HZ BOOKS
华章科技

内容全面，系统地讲解了Cassandra的所有功能特性及其使用方法
深入浅出，结合源代码分析了Cassandra的底层机制和工作原理

国内Cassandra领域的先驱者和实践者亲自执笔，多位数据库专家联袂推荐

数据库技术丛书

Cassandra in Action

Cassandra实战

郭 鹏◎著

机械工业出版社
China Machine Press

拼吾爱 —— 最新编程资源分享下载站

PS：PDF页面设置为100%，阅读效果最佳！

数据库技术丛书

Cassandra in Action

Cassandra实战

郭 鹏◎著

机械工业出版社
China Machine Press

前　言

为什么写本书

最早开始接触 Cassandra 是在 2010 年年初，那时 Cassandra 才刚刚进入人们的视线。由于传统的数据库已经无法满足项目业务的需求，公司便派我去寻找和尝试其他的数据库解决方案。当时有两种备选方案：Hadoop 项目中的 HBase 和 Facebook 开源的 Cassandra。

在对比了各方面的资料和分析了项目的业务需求之后，我们最终选择了配置简单、部署方便，使用高效的 Cassandra。与 HBase 不同，Cassandra 是一套独立的系统，不需要搭建和了解 HDFS 及 Zookeeper 集群即可开始使用，并且可以在 Windows 系统中直接进行测试。所以，我们在很短的时间之内就完成了功能和性能的评估，为后期的项目实施做好了技术储备。

到 2010 年 6 月份，部门开始正式搭建实时数据中心，需要将公司前台数据库产生的业务数据定期抽取到实时数据中心，提供给业务部门的相关人员使用。于是我们开始设计 Cassandra 的业务数据存储结构，并开始分析 Cassandra 内部的实现机制和源代码，从 Cassandra 的工作原理上指导我们的业务数据存储结构设计和测试方案。在整个设计和测试的过程中，我们遇到的一些问题，都可以通过活跃的 Cassandra 开发社区和开放的源代码找到答案。而且，我们还可以通过修改少量的源代码来修复一些 BUG 和增强系统管理的特性。最终，这些修改也都在开源社区提供的后续版本中得到了体现。得益于活跃的开源社区，

Cassandra 的功能和版本也在不断地更新。尤其是在最新的 Cassandra 0.7.x 版本中，提供了大家都非常期待的二级索引与在线更新 Schema 的功能。

Datastax 公司作为 Cassandra 的主要开发成员，开始为 Cassandra 的使用者提供相应的服务支持，并开发出了一系列强大的工具，进一步提高了 Cassandra 的实用性。另外，在 2011 年的 5 月，Datastax 将提供一款振奋人心的工具——Brisk。通过 Brisk，Cassandra 将取代 Hadoop 项目中的 HDFS 和 HBase 的角色，直接与 Hadoop 项目中的 MapReduce 和 Hive 项目进行集成，从而提高整个海量数据分析系统的性能和易用性。

相信未来 Cassandra 能够我们带来更多的惊喜。

本书面向的读者

初中级程序员。他们可以从本书中的实际例子了解 Cassandra 的基本概念，以及如何通过各种编程接口在 Cassandra 中写入和读取数据，如何建立二级索引，如何动态修改 Schema 信息，如何通过 MapReduce 做海量的数据分析，等等。

中高级程序员。他们可以通过本书对源代码的分析和讲解来了解 Cassandra 中的内部实现原理，从而能选用更加高效的实现方式，调整最适合的 Cassandra 集群运行配置，增强程序的整体运行性能和稳定性。

DBA 和系统运维人员。他们可以通过本书中的运维管理等内容，了解如何在 Cassandra 集群中安全高效地增加或减少服务器，以及如何恢复集群的错误，等等。

系统架构师。他们可以根据本书所讲的内容掌握 Cassandra 能做什么，不能做什么，适合做什么，从而为不同项目做出最合适的架构选型。

联系作者

正如大家所知，开源技术的更新速度非常之快，Cassandra 亦是如此。尽管我们已经很努力地将最新的内容融入其中，但仍难免会遗留部分内容。所以，如果你发现本书存在任何错误和问题，或者是你对书中的某些内容不理解需要与我们进一步探讨，再或者是你有好的意见或建议，都欢迎随时与我取得联系，我的邮箱是：gpcuster@gmail.com。

另外，我的个人博客也会经常更新关于 Cassandra 的最新动态和使用心得等内容，也欢迎大家来这里与我交流。博客的地址是：http://gpcuster.cnblogs.com。

致谢

感谢老张、许玉勤和姜迅，是你们的信任与支持给了我接触和学习 Cassandra 的机会和应用 Cassandra 的平台。

感谢童家旺、何勇、任振中、王海和陈晓峰，在学习和应用 Cassandra 的过程中，你们

给予了我很多帮助与支持。

感谢我的家人，一年来，我的大部分原来属于你们的业余时间都给了这本书，感谢你们的理解。

感谢杨福川和曾珊两位编辑为本书付出的耐心与努力，你们为本书提出了很多宝贵的建议，你们的敬业精神令我钦佩。

感谢本书的所有读者，希望你们能从本书中有所收获，大家的认可是我不断前进的动力。

郭　鹏

2011 年 4 月

目　　录

前　　言

第1章　认识 NoSQL/1

1.1　NoSQL 的起源和发展现状/2

1.2　为什么要使用 NoSQL/2

1.3　开源 NoSQL 产品介绍/3

1.3.1　Key/Value 的 NoSQL 数据库/3

1.3.2　面向文档的 NoSQL 数据库/4

1.3.3　面向列的 NoSQL 数据库/5

1.3.4　面向图的 NoSQL 数据库/6

1.4　本章小结/7

第2章　Cassandra 快速入门/9

2.1　在 Windows 环境运行单机版 Cassandra/10

2.1.1　配置 JRE/10

2.1.2　配置运行 Cassandra 0.6.x/11

2.1.3 配置运行 Cassandra 0.7.x/12
2.2 在 Linux 环境运行单机版 Cassandra/14
2.2.1 配置 JRE/14
2.2.2 配置运行 Cassandra 0.6.x/15
2.2.3 配置运行 Cassandra 0.7.x/16
2.3 Cassandra 的数据模型/18
2.3.1 Column/18
2.3.2 SuperColumn/18
2.3.3 ColumnFamily/19
2.3.4 Keyspace/20
2.4 Cassandra 的数据排序规则/20
2.5 配置数据类型/22
2.6 使用命令行工具与 Cassandra 交互/23
2.6.1 与 Cassandra 0.6.x 进行交互/23
2.6.2 与 Cassandra 0.7.x 进行交互/24
2.7 本章小结/26

第3章 理解 Cassandra 编程接口/27

3.1 多语言服务开发框架 Thrift/28
3.2 Cassandra 的数据类型/28
3.2.1 Column/28
3.2.2 SuperColumn/29
3.2.3 ColumnOrSuperColumn/29
3.2.4 ColumnParent/29
3.2.5 ColumnPath/30
3.2.6 SliceRange/30
3.2.7 SlicePredicate/30
3.2.8 Deletion/31
3.2.9 Mutation/31
3.2.10 KeyRange/31
3.2.11 KeySlice/32
3.2.12 TokenRange/32
3.2.13 AuthenticationRequest/32
3.2.14 ConsistencyLevel/33
3.2.15 NotFoundException/33

3.2.16 InvalidRequestException/34
3.2.17 UnavailableException/34
3.2.18 TimedOutException/34
3.2.19 AuthenticationException/34
3.2.20 AuthorizationException/35
3.3 Cassandra 的编程接口/35
3.3.1 get/35
3.3.2 get_slice/36
3.3.3 multiget_slice/36
3.3.4 get_count/37
3.3.5 get_range_slices/37
3.5.6 insert/38
3.3.7 remove/38
3.3.8 batch_mutate/39
3.3.9 describe_keyspaces/39
3.3.10 describe_keyspace/39
3.3.11 describe_cluster_name/40
3.3.12 describe_version/40
3.3.13 describe_ring/40
3.4 Cassandra 0.7.x 版本新增功能/40
3.4.1 二级索引/40
3.4.2 动态修改 Schema/44
3.4.3 自动清除过期数据/46
3.5 本章小结/47

第4章 基于 Cassandra 的在线交易系统/49

4.1 需求分析/50
4.2 数据模型设计/50
4.2.1 Seller/50
4.2.2 Buyer/51
4.2.3 Product/51
4.2.4 ProductCategory/52
4.2.5 Comment/53
4.3 编码实现/54
4.3.1 修改 Keyspace 设置/54
4.3.2 建立 Eclipse 项目/54

4.3.3 实体对象实现/55
4.3.4 Cassandra 数据操作接口实现/56

4.4 系统功能验证/60
4.4.1 BuyerDao 功能验证/60
4.4.2 SellerDao 功能验证/61
4.4.3 ProductDao 功能验证/62

4.5 迁移到 Cassandra 0.7.x/65
4.5.1 建立 Eclipse 项目/65
4.5.2 修改编译错误代码/65
4.5.3 新增 Schema 在线定义功能/69
4.5.4 功能验证/70

4.6 本章小结/71

第5章 Cassandra 的集群机制/73

5.1 一致性哈希/74
5.1.1 理解一致性哈希/74
5.1.2 一致性哈希在 Cassandra 中的应用/77

5.2 Gossip：集群节点之间的通信协议/81
5.2.1 FailureDetector/82
5.2.2 Gossiper/83

5.3 集群的数据备份机制/88
5.3.1 EndpointSnitch/88
5.3.2 ReplicationStrategy/91

5.4 集群状态变化的处理机制/96
5.4.1 StorageLoadBalancer/96
5.4.2 StorageService/97
5.4.3 MigrationManager/98

5.5 本章小结/99

第6章 Cassandra 的内部数据存储结构/101

6.1 Cassandra 中的数据存放规则/102

6.2 Commilog/102

6.3 Memtable/103

6.4 SSTable/105
6.4.1 Filter 文件/105
6.4.2 Index 文件/107
6.4.3 Data 文件/109
6.4.4 Statistics 文件/113
6.5 系统表空间/113
6.6 本章小结/114
第7章 Cassandra 的数据更新机制/115
7.1 数据更新流程/116
7.2 集群数据更新策略/116
7.2.1 ANY/120
7.2.2 ONE/121
7.2.3 QUORUM/121
7.2.4 LOCAL_QUORUM/121
7.2.5 EACH_QUORUM/121
7.2.6 ALL/121
7.3 二级索引/122
7.3.1 为什么需要二级索引/122
7.3.2 Cassandra 二级索引更新过程/123
7.4 本章小结/124
第8章 Cassandra 的数据读取机制/125
8.1 数据读取流程/126
8.1.1 弱读取/126
8.1.2 强读取/128
8.2 集群数据读取策略/131
8.2.1 ONE/132
8.2.2 QUORUM/132
8.2.3 LOCAL_QUORUM/132
8.2.4 EACH_QUORUM/132
8.2.5 ALL/133
8.3 读修复/133
8.4 数据缓存/134

8.4.1 RowCache/134
8.4.2 KeyCache/134
8.5 二级索引/135
8.6 本章小结/135

第9章 Cassandra的数据压缩机制/137

9.1 为什么要进行数据压缩/138
9.2 如何控制数据压缩/138
9.3 数据压缩流程/139
9.4 维护Cassandra中的数据/143
9.4.1 数据清理压缩/143
9.4.2 数据一致性校验压缩/144
9.5 本章小结/144

第10章 Cassandra的启动流程/145

10.1 Cassandra启动脚本/146
10.2 Cassandra启动流程/149
10.2.1 配置log4j/150
10.2.2 读取校验配置文件信息/150
10.2.3 加载所有的数据文件/152
10.2.4 修复数据/154
10.2.5 启动Gossiper服务/155
10.2.6 判断是否需要进行Bootstrap操作/156
10.2.7 监听Thrift端口，提供Thrift服务/157
10.3 本章小结/157

第11章 在分布式环境中使用的Cassandra/159

11.1 在Linux环境中搭建与使用Cassandra集群/160
11.1.1 配置JRE/160
11.1.2 部署Cassandra可执行文件/161
11.1.3 修改Cassandra配置文件/162
11.1.4 启动Cassandra/163
11.2 Cassandra运行配置项详解/166

11.3 Cassandra 集群的运行和维护/175
11.3.1 查看集群的运行情况/176
11.3.2 添加节点/179
11.3.3 删除节点/181
11.3.4 移动节点/183
11.3.5 数据维护/185
11.4 本章小结/187

第 12 章 Cassandra 与 Hadoop 的整合/189

12.1 Hadoop 快速入门/190
12.1.1 Hadoop 简介/190
12.1.2 HDFS/190
12.1.3 Map/Reduce/192
12.1.4 配置单机版 Hadoop/194
12.1.5 编写 Map/Reduce 程序/195
12.2 为什么要整合 Cassandra 与 Hadoop/200
12.3 使用 Map/Reduce 导入数据到 Cassandra 中/200
12.4 将 Cassandra 中的数据作为 Map/Reduce 输入/205
12.5 本章小结/209

第 13 章 Cassandra 最佳实践/211

13.1 避免 Cassandra 自身的限制/212
13.1.1 不要盲目使用 Super Column/212
13.1.2 硬盘的容量大小限制/212
13.1.3 注意系统大小限制/212
13.2 数据压缩策略/213
13.3 使用高级的客户端/213
13.3.1 Pycassa/213
13.3.2 Hector /215
13.3.3 FluentCassandra/218
13.3.4 Cassandra /220
13.3.5 phpcassa /221
13.4 负载均衡/222
13.4.1 随机选取/222

13.4.2 缓存集群信息/222

13.5 谨慎使用二级索引/223

13.6 通过 JMX 监测 Cassandra/223

13.7 调整 JVM 启动参数/229

13.8 使用适合的系统配置参数/231

13.9 本章小结/232

附录 A 在 Eclipse 中修改 Cassandra 源代码/233

附录 B CassSeller 代码/243

附录 C CassSeller－0.7 代码/273

第 1 章

认识 NoSQL

本章内容

- NoSQL 的起源和发展现状
- 为什么要使用 NoSQL
- 开源 NoSQL 产品介绍
- 本章小结

1.1 NoSQL 的起源和发展现状

对于 NoSQL 这个新兴的名词，大家的理解不尽相同。在网站 http://nosql-database.org/ 上对 NoSQL 有一个较为全面的解释：“下一代的数据库产品应该具备这几个特点：非关系型的、分布式的、开源的、可以线性扩展的。”这类数据库最初的目的在于提供现代网站可扩展的数据库解决方案，它开始于 2009 年年初，目前正在快速发展。这种类型的数据库具有以下特点：自由的 Schema，数据多处备份，简单的编程 API，数据的最终一致性等。所以，将这种类型的数据库称为 NoSQL（不仅仅是 SQL，全称为“not only sql”）数据库。

NoSQL 数据库并不是要取代现在广泛应用的传统数据库，而是采用一种非关系型的方式解决数据的存储和计算的问题。

由于 NoSQL 中没有像传统数据库那样定义数据的组织方式为关系型的，所以只要内部的数据组织采用了非关系型的方式，就可以称之为 NoSQL 数据库。

目前，可以将众多的 NoSQL 数据库按照内部的数据组织形式进行如下分类：

- Key/Value 的 NoSQL 数据库
- 面向文档的 NoSQL 数据库
- 面向列的 NoSQL 数据库
- 面向图的 NoSQL 数据库

不同的数据组织适合于不同的应用场景，后面将进行介绍。

1.2 为什么要使用 NoSQL

SQL 语言和关系型数据库（MySQL、PostgreSQL、Oracle 等）是通用的数据解决方案，占有绝大多数的市场。不过在最近兴起的 NoSQL 运动中，涌现出一批具备高可用性、支持线性扩展、支持 Map/Reduce 操作等特性的数据产品，它们具有如下特性：

- 频繁的写入操作、相对较少的读取统计信息的操作（如网站访问计数器），应该使用基于内存的 Key/Value（键/值）存储系统（如 Redis）或者是具备本地更新特性的文档存储系统（如 MongoDB）。
- 海量数据（如数据仓库中需要分析的数据）适合存储在一个结构松散、分布式的文件存储系统中，如 Hadoop。
- 存储二进制文件（如 mp3 或者 pdf 文档）并且能够直接为用户的浏览器提供下载功能，可以使用 Amazon S3。
- 临时性的数据（如网站的 session、缓存 HTML 页面信息等）适合存储在 Memcache 中。
- 如果希望数据具备高可用性，并且能够将数据丢失的风险降到最低，同时整个系统具备线性扩展的能力，可以考虑使用 Cassandra 和 HBase。

使用这些数据产品并不是要取代原有的数据产品，而是为不同的应用场景提供更多的选择。NoSQL 代表着：**选择合适的方案处理合适的业务场景**。上面介绍的几种 NoSQL 应用场景也许能够帮助我们选择合适的数据存储方案，网上也有不少值得参考的资源。和其他的技术方案一样，选择适合的业务场景才是最重要的。

绝大多数的应用都有非常复杂的应用场景，怎样才能找出一款 NoSQL 产品能够适用于所有的需求？答案是搭配使用多款 NoSQL 产品。传统数据库中的通用（One For All）的情况在 NoSQL 中是不存在的。

如图 1-1 所示，可以在一个网站中使用 4 款数据产品来提供服务。

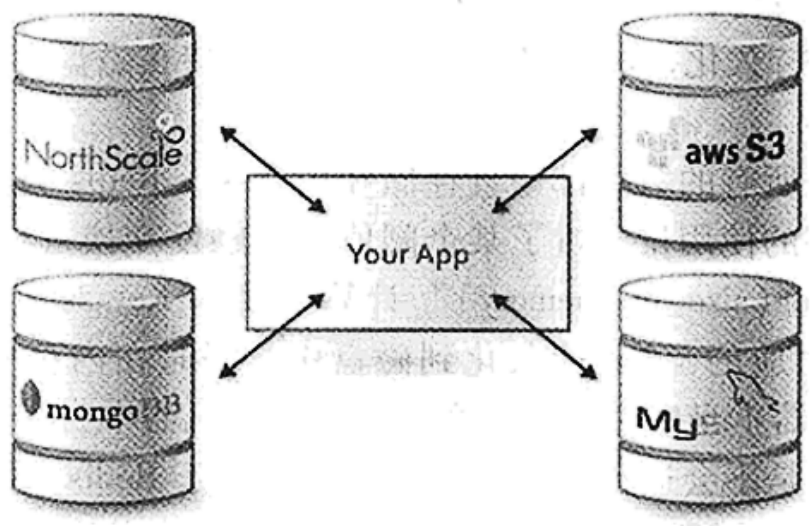


图 1-1　NoSQL 产品组合

- MySQL 用于存储敏感的数据，比如用户的资料、交易的信息等。
- MongoDB 用于存储大量的、相对不敏感的数据，比如博客文章的内容、文章访问次数等。
- Amazon S3 用于存储用户上传的文档、图片、音乐等数据。
- Memcached 用于存储临时性的信息，比如缓存 HTML 页面等。

选择多样的数据存储方案同样有利于提升我们对 NoSQL 的数据产品的理解，帮助我们从大量的解决方案中选择最适用的产品，而不是把眼光仅仅放在某一款产品上。

核心的思想是：**最适用的才是最好的**。

1.3　开源 NoSQL 产品介绍

1.3.1　Key/Value 的 NoSQL 数据库

1. Memcached

Memcached 是国外社区网站 LiveJournal 开发的高性能的内存 Key/Value 缓存服务器，目的是通过缓存数据库查询结果，减少数据库访问次数，以提高动态 Web 应用的速度，从而提高系统的可扩展性。

Memcached 虽然简单，但是却非常实用。它的简约设计非常适合于快速开发，并且能够解决大数据缓存的问题。同时 Memcached 为多种编程语言都提供了可以使用的应用编程接口（API）。

2. Redis

Redis 是一款先进的 Key/Value 存储系统。它与 Memcached 类似，区别如下：

- Redis 不仅支持简单的 Key/Value 类型的数据，同时还提供 list、set、hash 等数据结构的存储。
- Redis 支持数据的备份，即 master-slave 模式的数据备份。
- Redis 支持数据的持久化，可以将内存中的数据保持在磁盘中，重启的时候再次加载进行使用。

在 Redis 中，并不是所有的数据都一直存储在内存中。Redis 只会缓存所有的 Key 的信息，如果 Redis 发现内存的使用量超过了某个阈值，将触发交换（swap）的操作。Redis 根据“swappability = age * log(size_in_memory)”计算出哪些 Key 对应的 Value 需要交换到磁盘，然后再将这些 key 对应的 value 持久化到磁盘中，同时在内存中清除。这种特性使得 Redis 可以保持超过其机器本身内存大小的数据。当然，机器本身的内存必须要能够保持所有的 key，毕竟这些数据是不会进行交换操作的。同时由于 Redis 将内存中的数据交换到磁盘中的时候，提供服务的主线程和进行交换操作的子线程会共享这部分内存，所以如果更新需要交换的数据，Redis 将阻塞这个操作，直到子线程完成交换操作后才可以进行修改。

3. Dynamo

Dynamo 是亚马逊公司开发的一款分布式 Key/Value 存储系统，用于存储用户的购物车信息。Dynamo 与传统的 Key/Value 存储系统相比，最大的优势在于无单点故障，整个系统的可用性非常高，同时具备数据的最终一致性。

1.3.2 面向文档的 NoSQL 数据库

1. MongoDB

MongoDB 是一个高性能、开源、模式自由（schema-free）的文档型数据库，它在许多场景下可用于替代传统的关系型数据库或 Key/Value 存储方式。MongoDB 使用 C ++ 开发，具有以下特性：

- 面向文档的存储，适合存储对象及 JSON 形式的数据。
- 动态查询，MongoDB 支持丰富的查询表达式。查询指令使用 JSON 形式的标记，可轻易查询文档中内嵌的对象及数组。
- 完整的索引支持，包括文档内嵌对象及数组。MongoDB 的查询优化器会分析查询表达式，并生成一个高效的查询计划。
- 查询监视，MongoDB 包含一个监视工具用于分析数据库操作的性能。

- 复制及自动故障转移，MongoDB数据库支持服务器之间的数据复制，支持主-从模式（Master/Slave）及服务器之间的相互复制。复制的主要目标是提供冗余及自动故障转移。
- 高效的传统存储方式，支持二进制数据及大型对象（如照片或图片）。
- 自动分片以支持云级别的伸缩性，自动分片功能支持水平的数据库集群，可动态添加额外的机器。
- 模式自由，意味着对于存储在MongoDB数据库中的文件，我们不需要知道它的任何结构定义。
- 支持Map/Reduce计算，代表MongoDB具有强大的数据分析能力。

2. CouchDB

CouchDB是Apache社区中的一款文档型数据库服务器。与现在流行的关系数据库服务器不同，CouchDB是围绕一系列语义上自包含的文档而组织的。CouchDB中的文档是模式自由的，也就是说，并不要求文档具有某种特定的结构。CouchDB的这种特性使得它相对于传统的关系数据库而言，有自己的适用范围。一般来说，围绕文档来构建的应用都比较适合使用CouchDB作为其后台存储。CouchDB强调其中所存储的文档，在语义上是自包含的。这种面向文档的设计思路，更贴近很多应用的问题域的真实情况。对于这类应用，使用CouchDB的文档来进行建模，会更加自然和简单。与此同时，CouchDB也提供基于Map/Reduce编程模型的视图来对文档进行查询，可以提供类似于关系数据库中SQL语句的能力。CouchDB对于很多应用来说，提供了关系数据库之外的更好的选择。

1.3.3 面向列的NoSQL数据库

1. Cassandra

Cassandra是一款面向列的NoSQL数据库，和Google的Bigtable数据库属于同一类。此数据库比一个类似Dynamo的Key/Value数据库功能更多，但相比于面向文档的数据库（如MongoDB），它所支持的查询类型要少。

Cassandra结合了Dynamo的Key/Value与Bigtable的面向列的特点。

- 模式灵活：数据不需要像数据库一样使用预先设计的模式，增加或者删除字段非常方便（on the fly）。
- 支持范围查询：可以对任意Key进行范围查询。
- 支持二级索引查询：可以对任意列（Column）的值进行查询。
- 支持Map/Reduce计算：可以对Cassandra中的数据批量进行复杂的分析计算。
- 数据具备最终一致性，集群整体的可用性非常高。
- 高可用，可扩展：单点故障不影响集群服务，集群的性能可线性扩展。
- 数据可靠性高：一旦数据写入成功，数据就已经在机器的磁盘中完成了存储，不容易丢失。

2. HBase

HBase 是 Hadoop 项目中的数据库。它用于需要对大量的数据进行随机、实时的读写操作的场景中。HBase 的目标就是处理数据量非常庞大的表，可以用普通的计算机处理超过 10 亿行数据，还可处理有数百万列元素的数据表。

HBase 是一个开源的、分布式的、支持多版本的、面向列存储的 Google Bigtable 实现。HBase 的实现基于 Hadoop 分布式文件系统（HDFS），模仿并提供了基于 Google 文件系统的 Bigtable 数据库的所有功能。HBase 有如下特点：

- 可以直接从 HBase 中读取数据运行 Map/Reduce 任务，并可以将运行后的结果直接写入 HBase 中。
- 数据查询过滤和扫描操作在服务器端进行。
- 为实时查询做了特殊优化。
- 使用高性能的 Thrift 通信框架。
- 支持 REST、Protobuf 以及二进制形式的数据交互。
- 可以与 Cascading、Hive 和 Pig 配合使用，从而提高使用的效率。
- 提供可扩展的 JRuby（JIRB）的命令行工具。
- 支持 Ganglia 和 JMX，能够方便监视整个程序的运行状态。

1.3.4 面向图的 NoSQL 数据库

Neo4J 是一个用 Java 实现、完全兼容 ACID 的图形数据库。数据以一种针对图形网络进行过优化的格式保存在磁盘上。Neo4J 的内核是一种极快的图形引擎，具有数据库产品期望的所有特性，如恢复、两阶段提交、符合 XA 等。自 2003 年起，Neo4J 就已经作为 7 * 24 的产品使用。该项目已经发布了 1.2 版，它是关于伸缩性和社区测试的一个主要里程碑。通过联机备份实现的高可用性和主从复制功能目前处于测试阶段，预计在下一版本中发布。Neo4J 既可作为无须任何管理开销的内嵌数据库使用，也可以作为单独的服务器使用，在这种使用场景下，它提供了广泛使用的 REST 接口，能够方便地集成到基于 PHP、. NET 和 JavaScript 的环境里。

Neo4J 的特点如下：

- 用直观的图模型取代了严格定义的表模型，从而可以使用节点（node）、关系（relationship）、属性（property）来表达复杂的数据模型，如图 1-2 所示。

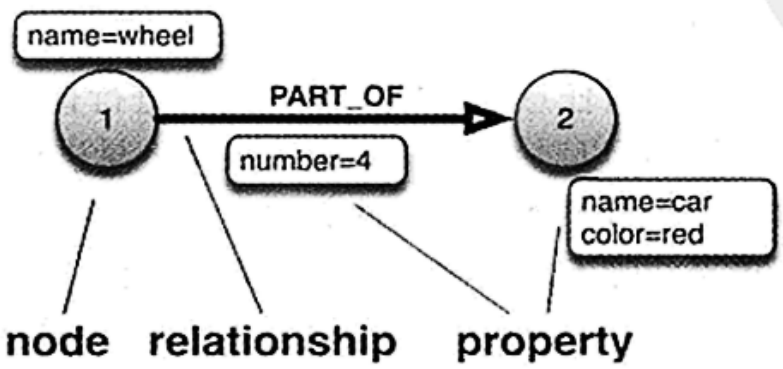
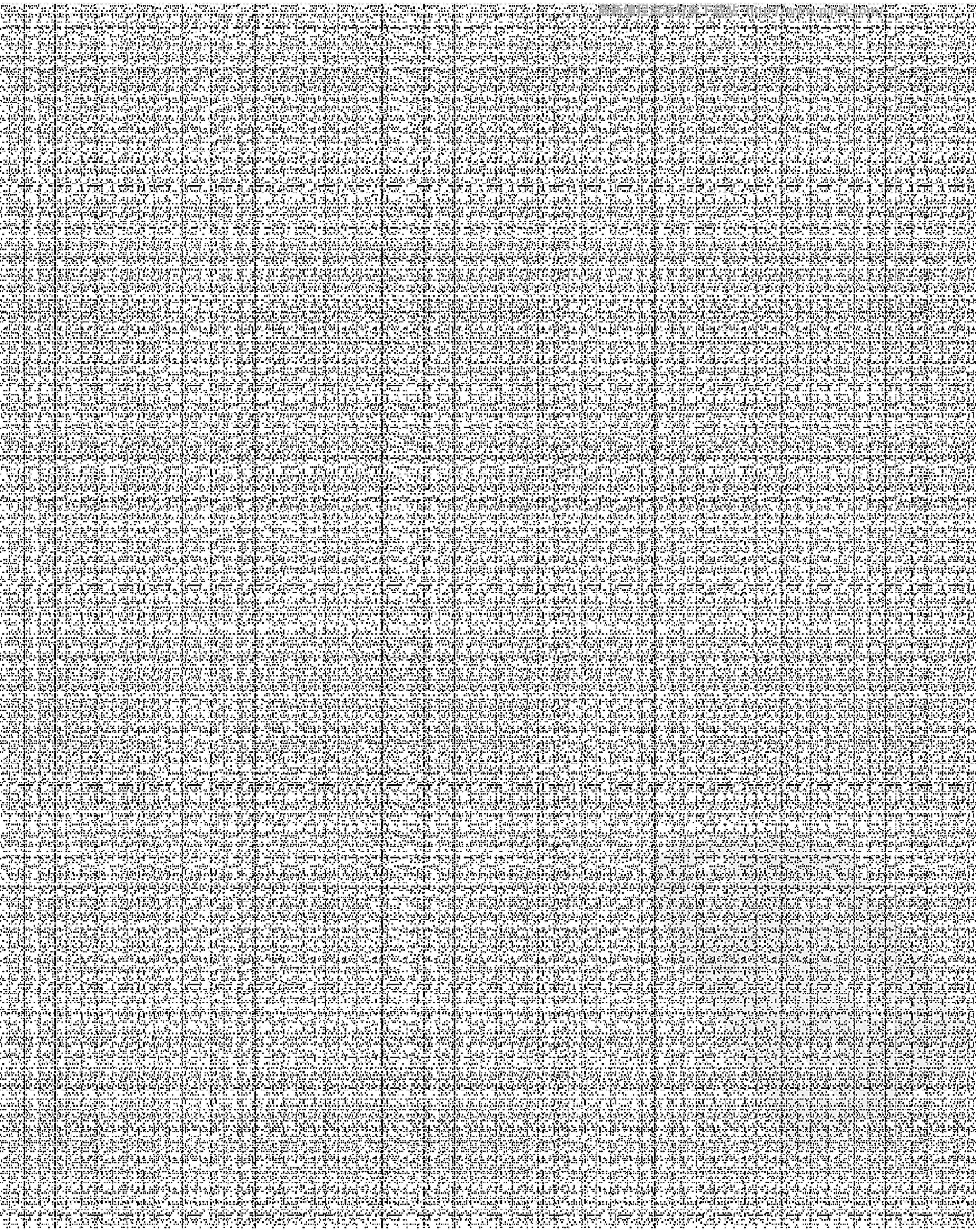


图 1-2 图模型的三大元素示意图

- 针对磁盘存储进行了特殊优化，使得其具备优异的性能和可扩展性。
- 每一台 Neo4J 服务器都可以处理上 10 亿的数据，并且可以通过水平拆分支持更大的数据量。
- 包含高效的图遍历算法，大大提高了数据的查询和分析能力。
- 程序本身非常简单小巧，核心功能的 Jar 包大小只有 500 KB。
- 具备简单好用的编程接口，方便程序的开发。

1.4　本章小结

本章首先介绍什么是 NoSQL，以及 NoSQL 的一些优势，然后介绍了各种开源 NoSQL 产品的功能和特性。从第 2 章开始，将详细讲解 Cassandra 的各种功能、特性以及使用方法。



第 2 章

Cassandra 快速入门

本章内容

- 在 Windows 环境运行单机版 Cassandra
- 在 Linux 环境运行单机版 Cassandra
- Cassandra 的数据模型
- Cassandra 的数据排序规则
- 配置数据类型
- 使用命令行工具与 Cassandra 交互
- 本章小结

在第 1 章中，我们介绍了 Cassandra 和各种 NoSQL 产品。在本章中，我们主要讲解如何在 Windows 和 Linux 平台上快速安装、配置并使用 Cassandra。

2.1 在 Windows 环境运行单机版 Cassandra

Cassandra 可以在多种环境中运行，包括 Windows、Linux 和 UNIX 系统。在 Windows 系统中运行 Cassandra 是非常方便的，只要配置好 JRE（Java 运行环境），然后下载 Cassandra，并进行简单的配置就可以运行了。

2.1.1 配置 JRE

JRE 是 Java 程序的运行环境。想要运行 Java 程序，必须先安装 JRE。JRE 的官方下载地址：http://www.java.com/zh_CN/download/manual.jsp。打开 JRE 的下载页面，如图 2-1所示。

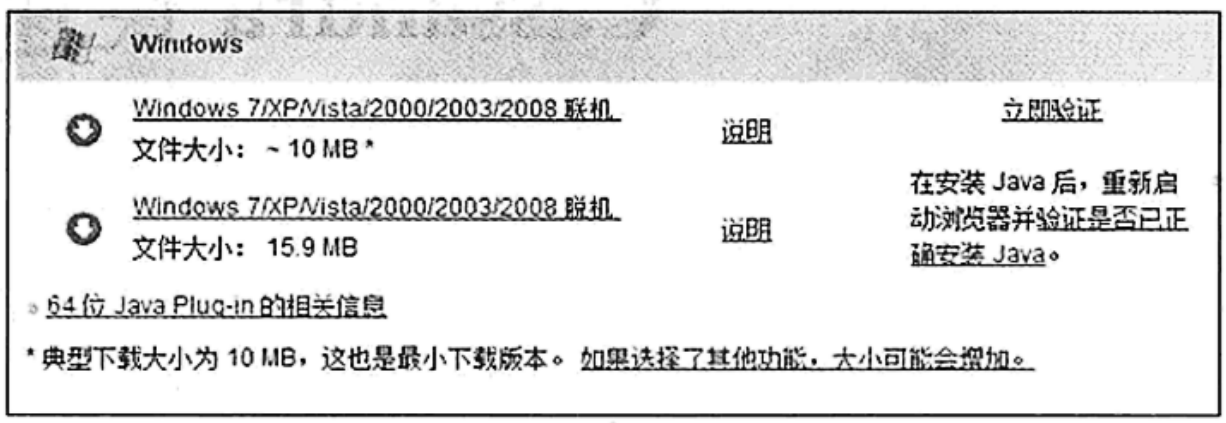


图 2-1 下载 JRE

下载完成之后，按照提示完成安装。如果没有更改 JRE 的安装路径，那么默认的安装路径为 C:\Program Files\Java\jre6\。完成安装后，还需要设置 JRE 的环境变量：JAVA_HOME = C:\Program Files\Java\jre6。完成设置后，可以打开“命令行窗口”检验 JRE 是否配置成功，如果输入“java - version”后，能够看到 JRE 的版本信息，说明 JRE 的配置成功了，如图 2-2 所示。

图 2-2 JRE 的版本信息

2.1.2　配置运行 Cassandra 0.6.x

在 Cassandra 官网下载 Cassandra 0.6.x 的压缩包。下载完成后，先将压缩包解压，假设解压的文件位置为：D:\apache-cassandra-0.6.x。然后，需要修改 Cassandra 的日志配置文件（conf/log4j.properties）和存储配置文件（conf/ storage-conf.xml）。在 conf/log4j.properties 文件中，需要修改 Cassandra 的运行日志的文件路径，将

```
log4j.appender.R.File = /var/log/cassandra/system.log
```

修改为

```
log4j.appender.R.File = /apache-cassandra-0.6.x/system.log
```

在文件 conf/ storage-conf.xml 中需要修改回滚日志（commitlog）、数据文件（data）和缓存数据文件（saved_cache）的存放路径。将

```
<SavedCachesDirectory>
   /var/lib/cassandra/saved_caches
</SavedCachesDirectory>
<CommitLogDirectory>
   /var/lib/cassandra/commitlog
</CommitLogDirectory>
<DataFileDirectories>
<DataFileDirectory>/var/lib/cassandra/data</DataFileDirectory>
</DataFileDirectories>
```

修改为

```
<SavedCachesDirectory>
   /apache-cassandra-0.6.x/saved_caches
</SavedCachesDirectory>
<CommitLogDirectory>
      /apache-cassandra-0.6.x/commitlog
</CommitLogDirectory>
<DataFileDirectories>
   <DataFileDirectory>/apache-cassandra-0.6.x/data</DataFileDirectory>
</DataFileDirectories>
```

最后，还需要设置 Cassandra 的环境变量：

```
CASSANDRA_HOME = D:\apache-cassandra-0.6.x
```

完成以上步骤以后，Cassandra 的环境就配置完毕了。

配置好 Cassandra 之后，打开“命令行窗口”，进入到 D:\apache-cassandra-0.6.x 目录下，然后直接执行“bin\cassandra.bat”命令就可以运行 Cassandra 了，如图 2-3 所示。同时，我们会在 D:\apache-cassandra-0.6.x 目录下看到 3 个目录（commitlog、data 和 saved_

cache）以及日志文件（system. log）。

图 2-3 在 Windows 环境运行 Cassandra 0. 6. x

2. 1. 3 配置运行 Cassandra 0. 7. x

在 Cassandra 官网下载 Cassandra 0. 7. x 的压缩包。下载完成后，先将压缩包解压，假设解压的文件位置为：C:\apache-cassandra-0. 7. x。然后，需要修改 Cassandra 的日志配置文件（conf/log4j-server. properties）和存储配置文件（conf/cassandra. yaml）。在 conf/log4j-server. properties 文件中，需要修改 Cassandra 的运行日志的文件路径，将

```
log4j.appender.R.File = /var/log/cassandra/system.log
```

修改为

```
log4j.appender.R.File = /apache-cassandra-0.7.x/system.log
```

在 conf/ cassandra. yaml 文件中，需要修改回滚日志（commitlog）、数据文件（data）和缓存数据（saved_caches）的存放路径。将

```
data_file_directories:
    - /var/lib/cassandra/data
commitlog_directory: /var/lib/cassandra/commitlog
saved_caches_directory: /var/lib/cassandra/saved_caches
```

修改为

```
data_file_directories:
    - /apache-cassandra-0.7.x/data
commitlog_directory: /apache-cassandra-0.7.x/commitlog
saved_caches_directory:
/apache-cassandra-0.7.x/saved_caches
```

最后，如果还没有设置 Java 的环境变量，请根据 Java 的安装目录设置 JAVA_HOME 路径，如

```
JAVA_HOME = C:\Program Files\Java\jre6
```

完成以上步骤以后，Cassandra 的环境就配置完毕了。

配置好 Cassandra 之后，打开“命令行窗口”，进入到 C:\apache-cassandra-0.7.x 目录下，然后直接执行“bin\cassandra.bat”命令就可以运行 Cassandra 了，如图 2-4 所示。同时，我们会在 D:\apache-cassandra-0.7.x 目录下发现多了 3 个目录（commitlog、data 和 saved_caches）以及日志文件（system.log）。

图 2-4　在 Windows 环境运行 Cassandra 0.7.x

另外，由于 Cassandra 0.7.x 版本中添加了在线修改元数据（Schema）信息的功能，所以系统在启动的时候，不会包含任何用户定义的元数据信息，需要自己手动创建，或者执行如下命令将配置文件中定义的元数据信息导入系统中。

```
C:\apache-cassandra-0.7.2>bin\schematool.bat localhost 8080 import
```

2.2 在 Linux 环境运行单机版 Cassandra

在 Linux 环境中配置 Cassandra 与在 Windows 环境中配置类似，区别主要在于 JRE 的配置方式和启动 Cassandra 的脚本不同。如启动 Cassandra，在 Windows 系统中使用的脚本名为 cassandra. bat，而在 Linux 系统中使用的脚本名为 cassandra。

下面演示的 Linux 系统版本为 32 位的 Ubuntu 10. 04。

2.2.1 配置 JRE

在大多数 Linux 发行版中，Java 都是默认安装的，可以直接在命令行中输入“java -version”来查看安装的 JRE 版本信息。

如果默认没有安装 JRE，可以到官网下载。JRE 的官方下载地址：http://www. java. com/zh_CN/download/manual. jsp。打开 JRE 的下载页面，如图 2-5 所示。

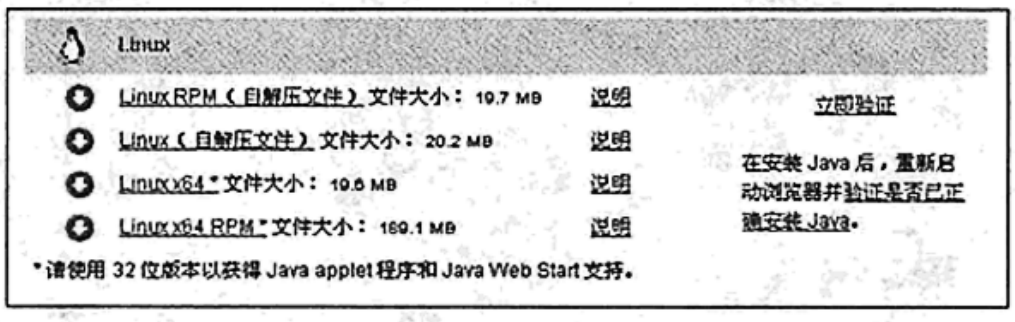


图 2-5 下载 JRE

下载 JRE 后，对其执行解压缩操作，设解压后的文件目录为/home/aaron/Downloads/jre1. 6. 0_21。在 ~/. bashrc 文件中将 JAVA_HOME 的环境变量设置到 JRE 目录的 bin 目录中，同时修改系统的 PATH 的环境变量。

```
export JAVA_HOME = /home/aaron/Downloads/jre1.6.0_21/bin
export PATH = $JAVA_HOME: $PATH
```

修改完成后，再执行以下命令，让修改的系统环境变量生效。

```
. ~/.bashrc
```

最后校验 JRE 环境是否配置成功，如图 2-6 所示。

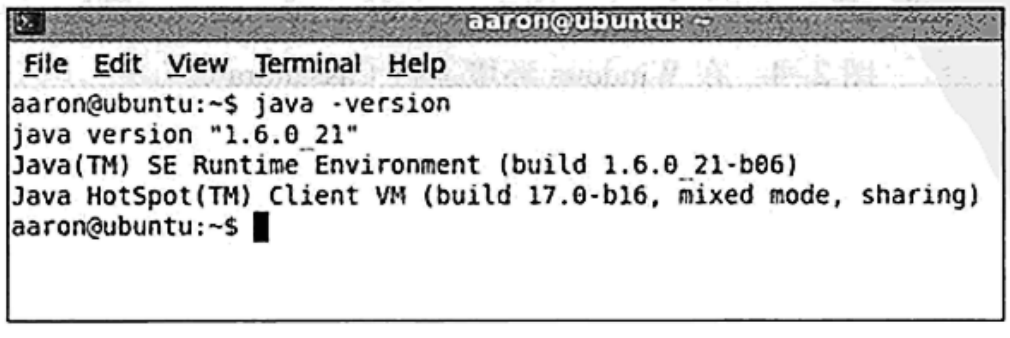


图 2-6 校验 JRE

从图 2-6 中可以看出，JRE 已经配置成功，并且可以正常使用。

2.2.2 配置运行 Cassandra 0.6.x

从官网中下载 Cassandra 0.6.x，然后将下载的压缩包解压，假设解压的文件位置为/home/aaron/Downloads/apache-cassandra-0.6.x。然后，需要修改 Cassandra 的日志配置文件（conf/log4j.properties）和存储配置文件（conf/storage-conf.xml）。在 conf/log4j.properties 文件中，需要修改 Cassandra 的运行日志的文件路径，将

```
log4j.appender.R.File = /var/log/cassandra/system.log
```

修改为

```
log4j.appender.R.File = /home/aaron/Downloads/apache - cassandra - 0.6.x/sys-
tem.log
```

在 conf/storage-conf.xml 文件中，需要修改回滚日志（commitlog）、数据文件（data）和缓存数据文件（saved_cache）的存放路径。将

```
<SavedCachesDirectory>
   /var/lib/cassandra/saved_caches
</SavedCachesDirectory>
<CommitLogDirectory>
   /var/lib/cassandra/commitlog
</CommitLogDirectory>
<DataFileDirectories>
<DataFileDirectory>/var/lib/cassandra/data</DataFileDirectory>
</DataFileDirectories>
```

修改为

```
<SavedCachesDirectory>
/home/aaron/Downloads/apache - cassandra - 0.6.x/saved_caches
    </SavedCachesDirectory>
    <CommitLogDirectory>
        /home/aaron/Downloads/apache - cassandra - 0.6.x/commitlog
    </CommitLogDirectory>
    <DataFileDirectories>
    <DataFileDirectory>/home/aaron/Downloads
/apache - cassandra - 0.6.x/data</DataFileDirectory>
    </DataFileDirectories>
```

完成以上步骤以后，Cassandra 的运行环境就配置完毕了。

配置好 Cassandra 之后，打开“命令行窗口”，进入到/home/aaron/Downloads/apache-cassandra-0.6.x 目录下，然后直接执行“bin/cassandra”命令就可以运行 Cassandra 了，如图 2-7 所示。同时，/home/aaron/Downloads/apache-cassandra-0.6.x 目录下多了 3 个目录

（commitlog、data 和 saved_cache）以及日志文件（system. log）。

```
aaron@ubuntu:~/Downloads/apache-cassandra-0.6.11$ bin/cassandra
aaron@ubuntu:~/Downloads/apache-cassandra-0.6.11$  INFO 00:24:57,647 JNA not found. Native methods will be
disabled.
 INFO 00:24:57,788 DiskAccessMode 'auto' determined to be standard, indexAccessMode is standard
 INFO 00:24:58,178 Saved Token not found. Using 136398396536561370822678890798127377580
 INFO 00:24:58,178 Saved ClusterName not found. Using Test Cluster
 INFO 00:24:58,185 Creating new commitlog segment /home/aaron/Downloads/apache-cassandra-0.6.11/commitlog/C
ommitLog-1298305498185.log
 INFO 00:24:58,231 LocationInfo has reached its threshold; switching in a fresh Memtable at CommitLogContex
t(file='/home/aaron/Downloads/apache-cassandra-0.6.11/commitlog/CommitLog-1298305498185.log', position=420)
 INFO 00:24:58,231 Enqueuing flush of Memtable-LocationInfo@22885256(169 bytes, 4 operations)
 INFO 00:24:58,233 Writing Memtable-LocationInfo@22885256(169 bytes, 4 operations)
 INFO 00:24:58,525 Completed flushing /home/aaron/Downloads/apache-cassandra-0.6.11/data/system/LocationInf
o-1-Data.db (360 bytes)
 INFO 00:24:58,543 Starting up server gossip
 INFO 00:24:58,626 Binding thrift service to localhost/127.0.0.1:9160
```

图 2-7　在 Linux 环境运行 Cassandra 0. 6. x

2. 2. 3　配置运行 Cassandra 0. 7. x

从官网中下载 Cassandra 0. 7. x，然后将下载的压缩包解压，假设解压的文件位置为 /home/aaron/Downloads/apache-cassandra-0. 7. x。然后，需要修改 Cassandra 的日志配置文件（conf/log4j-server. properties）和存储配置文件（conf/cassandra. yaml）。在 conf/log4j-server. properties 文件中，需要修改 Cassandra 的运行日志的文件路径，将

```
log4j.appender.R.File = /var/log/cassandra/system.log
```

修改为

```
log4j.appender.R.File = /home/aaron/Downloads/apache - cassandra - 0.7.x/sys-
tem.log
```

在 conf/ cassandra. yaml 文件中，需要修改回滚日志（commitlog）、数据文件（data）和缓存数据（caches）的存放路径。将

```
# directories where Cassandra should store data on disk.
data_file_directories:
    - /var/lib/cassandra/data

# commit log
commitlog_directory: /var/lib/cassandra/commitlog

# saved caches
saved_caches_directory: /var/lib/cassandra/saved_caches
```

修改为

```
# directories where Cassandra should store data on disk.
data_file_directories:
    - /home/aaron/Downloads/apache - cassandra - 0.7.x/data

# commit log
```

```
    commitlog_directory:
/home/aaron/Downloads/apache-cassandra-0.7.x/commitlog

    # saved caches
    saved_caches_directory:
/home/aaron/Downloads/apache-cassandra-0.7.x/saved_caches
```

完成以上步骤以后，Cassandra 的运行环境就配置完毕了。

配置好 Cassandra 之后，打开“命令行窗口”，进入到/home/aaron/Downloads/apache-cassandra-0.7.x 目录下，然后直接执行“bin/cassandra”命令就可以运行 Cassandra 了，如图 2-8 所示。同时，/home/aaron/Downloads/apache-cassandra-0.7.x 目录下多了 3 个目录（commitlog、data 和 saved_caches）以及日志文件（system.log）。

```
aaron@ubuntu:~/Downloads/apache-cassandra-0.7.0-rc3$ bin/cassandra
aaron@ubuntu:~/Downloads/apache-cassandra-0.7.0-rc3$  INFO 00:29:47,665 Heap size: 523698176/524746752
 INFO 00:29:47,671 JNA not found. Native methods will be disabled.
 INFO 00:29:47,736 Loading settings from file:/home/aaron/Downloads/apache-cassandra-0.7.0-rc3/conf/cassandra.yaml
 INFO 00:29:47,982 DiskAccessMode 'auto' determined to be standard, indexAccessMode is standard
 INFO 00:29:48,192 Creating new commitlog segment /home/aaron/Downloads/apache-cassandra-0.7.0-rc3/commitlog/CommitLog-1298305788192.log
 INFO 00:29:48,323 Opening /home/aaron/Downloads/apache-cassandra-0.7.0-rc3/data/system/IndexInfo-e-1
 INFO 00:29:48,406 Opening /home/aaron/Downloads/apache-cassandra-0.7.0-rc3/data/system/Schema-e-1
 INFO 00:29:48,428 Opening /home/aaron/Downloads/apache-cassandra-0.7.0-rc3/data/system/Schema-e-2
 INFO 00:29:48,446 Opening /home/aaron/Downloads/apache-cassandra-0.7.0-rc3/data/system/Migrations-e-1
 INFO 00:29:48,459 Opening /home/aaron/Downloads/apache-cassandra-0.7.0-rc3/data/system/LocationInfo-e-2
 INFO 00:29:48,489 Opening /home/aaron/Downloads/apache-cassandra-0.7.0-rc3/data/system/LocationInfo-e-1
 INFO 00:29:48,650 Loading schema version 31a8b1eb-0f0f-11e0-ae6e-e700f669bcfc
 WARN 00:29:48,989 Schema definitions were defined both locally and in cassandra.yaml. Definitions in cassandra.yaml were ignored.
 INFO 00:29:49,072 Replaying /home/aaron/Downloads/apache-cassandra-0.7.0-rc3/commitlog/CommitLog-1293161662044.log
 INFO 00:29:49,122 Finished reading /home/aaron/Downloads/apache-cassandra-0.7.0-rc3/commitlog/CommitLog-1293161662044.log
 INFO 00:29:49,125 Log replay complete
 INFO 00:29:49,195 Cassandra version: 0.7.0-rc3
 INFO 00:29:49,195 Thrift API version: 19.4.0
 INFO 00:29:49,209 Loading persisted ring state
 INFO 00:29:49,210 Starting up server gossip
 INFO 00:29:49,262 switching in a fresh Memtable for LocationInfo at CommitLogContext(file='/home/aaron/Downloads/apache-cassandra-0.7.0-rc3/commitlog/CommitLog-1298305788192.log', position=148)
 INFO 00:29:49,266 Enqueuing flush of Memtable-LocationInfo@26578114(29 bytes, 1 operations)
 INFO 00:29:49,267 Writing Memtable-LocationInfo@26578114(29 bytes, 1 operations)
 INFO 00:29:49,535 Completed flushing /home/aaron/Downloads/apache-cassandra-0.7.0-rc3/data/system/LocationInfo-e-3-Data.db (149 bytes)
 INFO 00:29:49,558 Using saved token 121011596126825432238618126362026260339
 INFO 00:29:49,559 switching in a fresh Memtable for LocationInfo at CommitLogContext(file='/home/aaron/Downloads/apache-cassandra-0.7.0-rc3/commitlog/CommitLog-1298305788192.log', position=444)
 INFO 00:29:49,560 Enqueuing flush of Memtable-LocationInfo@24080548(53 bytes, 2 operations)
 INFO 00:29:49,560 Writing Memtable-LocationInfo@24080548(53 bytes, 2 operations)
 INFO 00:29:50,251 Completed flushing /home/aaron/Downloads/apache-cassandra-0.7.0-rc3/data/system/LocationInfo-e-4-Data.db (301 bytes)
 INFO 00:29:50,252 Compacting [org.apache.cassandra.io.sstable.SSTableReader(path='/home/aaron/Downloads/apache-cassandra-0.7.0-rc3/data/system/LocationInfo-e-1-Data.db'),org.apache.cassandra.io.sstable.SSTableReader(path='/home/aaron/Downloads/apache-cassandra-0.7.0-rc3/data/system/LocationInfo-e-2-Data.db'),org.apache.cassandra.io.sstable.SSTableReader(path='/home/aaron/Downloads/apache-cassandra-0.7.0-rc3/data/system/LocationInfo-e-3-Data.db'),org.apache.cassandra.io.sstable.SSTableReader(path='/home/aaron/Downloads/apache-cassandra-0.7.0-rc3/data/system/LocationInfo-e-4-Data.db')]
 INFO 00:29:50,256 Will not load MX4J, mx4j-tools.jar is not in the classpath
 INFO 00:29:50,519 Binding thrift service to localhost/127.0.0.1:9160
 INFO 00:29:50,522 Using TFramedTransport with a max frame size of 15728640 bytes.
```

图 2-8　在 Linux 环境运行 Cassandra 0.7.x

另外，由于 Cassandra 0.7.x 版本中添加了在线修改元数据（Schema）信息的功能，所以系统在启动的时候不会包含任何用户定义的元数据信息，需要自己手动创建，或者执行

如下命令将配置文件中定义的元数据信息导入系统中。

```
sh bin/schematool localhost 8080 import
```

2.3 Cassandra 的数据模型

Cassandra 的数据模型与传统的键值对类型不同，我们可以将 Cassandra 的数据模型想象成一个四维或者五维的 HashMap。

在 Cassandra 中，数据类型有以下几种：Column、SuperColumn、ColumnFamily 和 Keyspace。下面将分别介绍这 4 种数据类型。

2.3.1 Column

Column 是 Cassandra 中最小的数据单元。它是一个三元的数据类型，包含 name、value 和 timestamp。

将一个 Column 用 JSON 的形式表现出来，如下所示：

```
{ //这是一个 Column
  name: "email address",
  value: "gpcuster@ gmali.com",
  timestamp: 123456789
}
```

这里提到的 name 和 value 都是 Java 中的 byte[ ]类型。

2.3.2 SuperColumn

我们可以将 SuperColumn 想象成 Column 的数组，它包含一个 name 以及一系列相应的 Column。

将一个 SuperColumn 用 JSON 的形式表现如下：

```
{    //这是一个 SuperColumn
     name: "email",
     //包含一系列的 Column
     value: {
         {name: "address", value: "gpcustr@ gmail.com",timestamp: 123456789},
         {name: "id", value: "gpcuster", timestamp: 123456789},
         {name: "password", value: "123456", timestamp: 123456789},
     }
}
```

Column 和 SuperColumn 都是 name 与 value 的组合，它们之间最大的不同在于 Column 的 value 是 Java 中的 String 类型，而 SuperColumn 的 value 是 Column 的 Map。

注意： SuperColumn 自身是不包含 timestamp 属性的。

2.3.3　ColumnFamily

ColumnFamily 是一个包含了许多 Row 的结构，可以将它想象成数据库中的表。Row 是由 Key 以及与这个 Key 所关联的 Column 组成的。

代码清单 2-1 代表了一个 ColumnFamily。

代码清单 2-1　ColumnFamily 示例

```
UserProfile = { //这是一个 ColumnFamily
   gpcuster: {  //这是对应 ColumnFamily 的 Key
      //这是 Key 下对应的 Column
      {name: "username", value: "aaron", timestamp: 123456789},
      {name: "email", value: "gpcuster@ gmail.com", timestamp: 123456789},
      {name: "phone", value: "13912345678", timestamp: 123456789}
     }, //第一个 Row 结束
     ieure: {    //这是 ColumnFamily 的另一个 Key
        //这是另一个 Key 对应的 column
        {name: "username", value: "bean", timestamp: 123456789},
        {name: "email", value: "bean@ gmail.com", timestamp: 123456789},
        {name: "phone", value: "13312341234", timestamp: 123456789},
        {name: "age", value: "23", timestamp: 123456789}
     } //第二个 Row 结束
}
```

上面例子是一个 Standard 类型的 ColumnFamily，Standard 类型的 ColumnFamily 包含了一系列 Column。

ColumnFamily 也可以是 Super 类型的，代码清单 2-2 代表了一个 SuperColumnFamily。

代码清单 2-2　SuperColumnFamily 示例

```
AddressBook = { //这是一个 Super 类型的 ColumnFamily
  gpcuster: {  //这是对应 ColumnFamily 的 Key
      {
        name : "John", //这是 SuperColumn 的 name
        value : {
            {name: "email", value:"john@ gmail.com", timestamp: 123456789},
            {name: "phone", value:"13412341234", timestamp: 123456789},
        }
      },
      {
        name : "Kim", //这是 SuperColumn 的 name
        value : {
            {name: "email", value:"kim@ gmail.com", timestamp: 123456789},
            {name: "phone", value:"13412340000", timestamp: 123456789},
        }
      },
      {
        name : "Tod", //这是 SuperColumn 的 name
```

```
            value : {
                {name: "email", value:"tod@ gmail.com", timestamp: 123456789},
                {name: "phone", value: "13412341111", timestamp: 123456789},
            }
        }
    }, //第一个 Row 结束
    ieure: {       //这是对应 ColumnFamily 的 Key
        {
          name : "joey", //这是 SuperColumn 的 name
          value : {
                {name: "email", value: "joey@ gmail.com", timestamp: 123456789},
                {name: "phone", value:"13400001234", timestamp: 123456789},
          }
        },
        {
          name : "William", //这是 SuperColumn 的 name
          value : {
                {name: "email", value:"william@ gmail.com", timestamp: 123456789},
                {name: "phone", value: "13011110000", timestamp: 123456789},
          }
        }
      } //第二个 Row 结束
}
```

2.3.4 Keyspace

每一个 Keyspace 包含多个 ColumnFamily，并且可以指定使用的数据备份策略和数据的备份数。

一般来说，Cassandra 集群只需要一个 Keyspace 就足够了。但是当 Cassandra 集群足够庞大、业务足够复杂的时候，一个 Keyspace 就无法满足需求了。

比如有两个博客网站共用同一个 Cassandra 集群，每一个博客网站都需要一个叫做 BlogEntry 的 ColumnFamily，而且第一个博客网站的数据备份数为 3，第二个博客网站的数据备份数为 5，那么就需要使用 2 个不同的 Keysapce 才能满足这个条件。

2.4 Cassandra 的数据排序规则

除了拥有灵活的数据模型，Cassandra 还可以指定 ColumnFamily 中 Column 的排序规则。这个排序操作是在数据插入 Cassandra 的时候完成的，读取的时候就已经是有序的了。

假设定义了一个 ColumnFamliy，并且将某一个 Key 下面的 4 条数据分别写入 Cassandra。这 4 条数据的写入顺序如下：

```
{name: 123, value: "first insert", timestamp: 123456789}
{name: 832416, value: "second insert", timestamp: 123456789}
```

```
{name: 3, value: "third insert", timestamp: 123456789}
{name: 976, value: "fourth insert", timestamp: 123456789}
```

如果指定的排序规则为 LongType，那么这 4 条数据写入 Cassandra 后的顺序如下：

```
{name: 3, value: "third insert", timestamp: 123456789}
{name: 123, value: "first insert", timestamp: 123456789}
{name: 976, value: "fourth insert", timestamp: 123456789}
{name: 832416, value: "second insert", timestamp: 123456789}
```

可以看到，按照 LongType 排序后，Key 下面的所有 Column 都是以 Column 的 name 数值大小排序的。

也可以指定排序规则为 UTF8Type，那么这 4 条数据写入 Cassandra 后的顺序如下：

```
{name: 123, value: "first insert", timestamp: 123456789}
{name: 3, value: "third insert", timestamp: 123456789}
{name: 832416, value: "second insert", timestamp: 123456789}
{name: 976, value: "fourth insert", timestamp: 123456789}
```

按照 UTF8Type 排序以后，排序的规则变为了按照 Column 的 name 字节大小。

除了 LongType 和 UTF8Type 的排序规则之外，Cassandra 还支持其他排序规则。表 2-1 详细介绍了 Cassandra 支持的所有排序规则。

表 2-1　Cassandra 支持的所有排序规则

排序名称	排序规则
BytesType	按照 Column 名称的 Byte 顺序进行排序
AsciiType	按照 Column 名称的 ASCII 顺序进行排序
UTF8Type	按照 Column 名称的 UTF8 顺序进行排序
LongType	按照 Column 名称的 Long 顺序进行排序
LexicalUUIDType	按照 Column 名称的 LexicalUUID 顺序进行排序
TimeUUIDType	按照 Column 名称的 TimeUUID 顺序进行排序

由于 Super 类型的 ColumnFamily 中包括的是 SuperColumn，所以除了可以指定 SuperColumn 与 SuperColumn 之间的排序规则外，还可以额外指定 SuperColumn 中所包括的 Column 的排序规则，并且这些排序规则都是通用的，如下所示：

```
{ //第一个 SuperColumn
  name: "workAddress",
  value: {
     {name: "street", value: "street one", timestamp:123456789},
     {name: "city", value: "Beijing", timestamp:123456789},
     {name: "zip", value: "100000", timestamp: 123456789}
  }
}
{ //第二个 SuperColumn
  name: "homeAddress",
```

```
    value: {
      {name: "street", value: "street two", timestamp: 123456789},
      {name: "city", value: "Hunan", timestamp: 123456789},
      {name: "zip", value: "414000", timestamp: 123456789}
    }
}
```

如果指定 SuperColumn 与 SuperColumn 之间的排序规则与 SuperColumn 中所包括的 Column 的排序规则为 UTF8Type，那么排序后的结果如下：

```
{ //第二个 SuperColumn
  name: "homeAddress",
  value: {
    {name: "city", value: "Hunan", timestamp: 123456789},
    {name: "street", value: "street two", timestamp:123456789},
    {name: "zip", value: "414000", timestamp: 123456789}
  }
},
{  //第一个 SuperColumn
  name: "workAddress",
  value: {
    {name: "city", value: "Beijing", timestamp:123456789},
    {name: "street", value: "street one", timestamp:123456789},
    {name: "zip", value: "100000", timestamp: 123456789}
  }
}
```

可以看到，经过排序后，名称为“homeAddress”的 SuperColumn 排在了名称为“workAddress”的 SuperColumn 前面。同时 SuperColumn 内部名称为“city”的 Column 排在了名称为“street”的 Column 前面。

2.5 配置数据类型

在 Cassandra 0.6.x 中，我们需要修改 storage-conf.xml 来配置使用的数据类型，并且修改后的数据模型需要重启 Cassandra 才能生效。

在 storage-conf.xml 文件中，可以任意指定 Keyspace 以及 Keyspace 下面的 ColumnFamily 和 ColumnFamily 的排序类型。

假设需要添加一个名为 School 的 Keyspace，并且下面包括 3 个 ColumnFamily，分别为 Student、Teacher 和 Class，我们可以在 storage-conf.xml 文件中的 <Keyspaces> 范围内添加如下内容：

```
<Keyspace Name = "School" >
    <ColumnFamily Name = "Student" CompareWith = "UTF8Type"/>
    <ColumnFamily Name = "Teacher" CompareWith = "UTF8Type"/>
```

```
    <ColumnFamily Name = "Class"
                  ColumnType = "Super"
                  CompareWith = "UTF8Type"
                  CompareSubcolumnsWith = "UTF8Type" />
    <ReplicaPlacementStrategy > org.apache.cassandra.locator.RackUnaware Strat-
egy < /ReplicaPlacementStrategy >
    <ReplicationFactor >1 < /ReplicationFactor >
    <EndPointSnitch > org.apache.cassandra.locator.EndPointSnitch < /EndPoint-
Snitch >
 < /Keyspace >
```

添加完这些内容后，我们再启动 Cassandra 就会多一个 Keyspace 可以使用。在指定 Keyspace 包含 ColumnFamily 的同时，也通过 CompareWith 指定了排序规则。对于 Super 类型的 ColumnFamily，我们需要指定 ColumnType，并且可以通过 CompareSubcolumnsWith 指定 SuperColumn 下 Column 的排序规则。

在指定新的 Keyspace 的同时，也指定了备份策略（ReplicaPlacementStrategy）、备份数（ReplicationFactor）和网络寻址规则（EndPointSnitch），这些配置的详细信息会在后面的章节中介绍。

在 Cassandra 0.7.x 中，可以直接在 conf/cassandra.yaml 文件中修改 schema，然后使用 bin/schematool 文件将新定义的 schema 文件信息导入 Cassandra 中，或者直接通过命令行执行需要修改的 schema 信息，无须重启 Cassandra 进程。

2.6 使用命令行工具与 Cassandra 交互

2.6.1 与 Cassandra 0.6.x 进行交互

打开一个命令行窗口，进入到 Cassandra 的安装目录下，在启动 Cassandra 命令行工具后，输入“connect localhost/9160”就可以连接到本机的 Cassandra 了。连接成功后，会输出“Connected to："Test Cluster" on localhost/9160”的字样，如图 2-9 所示。

图 2-9 启动 Cassandra 0.6.x 命令工具

接着，我们可以在 School 的 Keyspace 中插入如下两条数据到 Student 中：

```
cassandra > set School.Student['aaron']['no'] ='123456'
Value inserted.
cassandra > set School.Student['aaron']['age'] ='25'
Value inserted.
```

在命令“set School. Student['aaron']['no'] ='123456'”中，“set”表示要添加新的值，“School”表示 Keyspace 的名称，“Student”表示 ColumnFamily 的名称，“aaron”表示 Key，“no”表示 Column 的名称，“123456”表示 Column 的值。

然后，可以将插入的这两条数据取回来。

```
cassandra > get School.Student['aaron']
=> (column = no, value = 123456, timestamp = 1280502063365000)
=> (column = age, value = 25, timestamp = 1280502069028000)
Returned 2 results.
```

也可以分别取回 aaron 这个 Key 下面的 Column。

```
cassandra > get School.Student['aaron']['no']
=> (column = no, value = 123456, timestamp = 1280502063365000)
cassandra > get School.Student['aaron']['age']
=> (column = age, value = 25, timestamp = 1280502069028000)
```

更多关于 Cassandra 命令行提供的功能，可以输入“?”或者“help”进行查看，部分功能的预览界面如图 2-10 所示。

```
cassandra> ?
List of all CLI commands:
?                                                    Same as help.
help                                                 Display this help.
connect <hostname>/<port>                            Connect to thrift service.
describe keyspace <keyspacename>                     Describe keyspace.
exit                                                 Exit CLI.
quit                                                 Exit CLI.
show config file                                     Display contents of config file.
show cluster name                                    Display cluster name.
show keyspaces                                       Show list of keyspaces.
show api version                                     Show server API version.
get <ksp>.<cf>['<key>']                              Get a slice of columns.
get <ksp>.<cf>['<key>']['<super>']                   Get a slice of sub columns.
get <ksp>.<cf>['<key>']['<col>']                     Get a column value.
get <ksp>.<cf>['<key>']['<super>']['<col>']          Get a sub column value.
set <ksp>.<cf>['<key>']['<col>'] = '<value>'         Set a column.
set <ksp>.<cf>['<key>']['<super>']['<col>'] = '<value>'  Set a sub column.
del <ksp>.<cf>['<key>']                              Delete record.
del <ksp>.<cf>['<key>']['<col>']                     Delete column.
del <ksp>.<cf>['<key>']['<super>']['<col>']          Delete sub column.
count <ksp>.<cf>['<key>']                            Count columns in record.
count <ksp>.<cf>['<key>']['<super>']                 Count columns in a super column.
```

图 2-10　Cassandra 0. 6. x 命令工具提供的功能

2. 6. 2　与 Cassandra 0. 7. x 进行交互

打开一个命令行窗口，进入到 Cassandra 的安装目录下，在启动 Cassandra 命令行工具后，输入“connect localhost/9160”就可以连接到本机的 Cassandra 了。连接成功后，会输出“Connected to: "Test Cluster" on localhost/9160”的字样，如图 2-11 所示。

```
C:\apache-cassandra-0.7.2>bin\cassandra-cli.bat
Starting Cassandra Client
Welcome to cassandra CLI.

Type 'help;' or '?' for help. Type 'quit;' or 'exit;' to quit.
[default@unknown] connect localhost/9160;
Connected to: "Test Cluster" on localhost/9160
[default@unknown]
```

图 2-11　启动 Cassandra 0.7.x 命令工具

接着，我们可以在 Cassandra 中先删除之前通过 schematool 加载的 Keyspace1，然后创建一个 Keyspace1，如图 2-12 所示。

```
[default@unknown] drop keyspace Keyspace1;
32c0ca7-3dda-11e0-9cde-e700f669bcfc
[default@unknown] create keyspace Keyspace1;
4b4bab78-3dda-11e0-9cde-e700f669bcfc
[default@unknown] use Keyspace1;
Authenticated to keyspace: Keyspace1
[default@Keyspace1] create column family Users with comparator=UTF8Type and defa
ult_validation_class=UTF8Type;
57b6af90-3dda-11e0-9cde-e700f669bcfc
[default@Keyspace1]
[default@Keyspace1]
```

图 2-12　创建一个 Keyspace1

我们可以在 Keyspace1 中插入三条数据，然后再将这些数据读取出来，如图 2-13 所示。

```
[default@Keyspace1]
[default@Keyspace1] set Users[jsmith][first] = 'John';
Value inserted.
[default@Keyspace1]
[default@Keyspace1] set Users[jsmith][last] = 'Smith';
Value inserted.
[default@Keyspace1] set Users[jsmith][age] = long(42);
Value inserted.
[default@Keyspace1] get Users[jsmith];
=> (column=age, value=42, timestamp=1298307080342000)
=> (column=first, value=John, timestamp=1298307066774000)
=> (column=last, value=Smith, timestamp=1298307073589000)
Returned 3 results.
[default@Keyspace1]
```

图 2-13　在 Keyspace1 中插入三条数据，并读取出来

更多关于 Cassandra 命令行提供的功能，可以输入“?”或者“help”进行查看，部分功能的预览界面如图 2-14 所示。

```
cassandra> ?
List of all CLI commands:
?                                                    Same as help.
help                                                 Display this help.
connect <hostname>/<port>                            Connect to thrift service.
describe keyspace <keyspacename>                     Describe keyspace.
exit                                                 Exit CLI.
quit                                                 Exit CLI.
show config file                                     Display contents of config file.
show cluster name                                    Display cluster name.
show keyspaces                                       Show list of keyspaces.
show api version                                     Show server API version.
get <ksp>.<cf>['<key>']                              Get a slice of columns.
get <ksp>.<cf>['<key>']['<super>']                   Get a slice of sub columns.
get <ksp>.<cf>['<key>']['<col>']                     Get a column value.
get <ksp>.<cf>['<key>']['<super>']['<col>']          Get a sub column value.
set <ksp>.<cf>['<key>']['<col>'] = '<value>'         Set a column.
set <ksp>.<cf>['<key>']['<super>']['<col>'] = '<value>'  Set a sub column.
del <ksp>.<cf>['<key>']                              Delete record.
del <ksp>.<cf>['<key>']['<col>']                     Delete column.
del <ksp>.<cf>['<key>']['<super>']['<col>']          Delete sub column.
count <ksp>.<cf>['<key>']                            Count columns in record.
count <ksp>.<cf>['<key>']['<super>']                 Count columns in a super column.
```

图 2-14　Cassandra 0.7.x 命令工具提供的功能

提示：如果在 Linux 平台中使用该命令行工具，只需要在 Cassandra 的目录下输入 bin\cassandra-cli 即可。

2.7 本章小结

本章的目的是引导读者快速搭建 Cassandra 的使用环境和掌握 Cassandra 的基本操作方法。首先演示了如何在 Windows 和 Linux 系统中配置和运行 Cassandra，接着详细讲解了 Cassandra 的数据模型、排序规则以及如何配置数据类型，最后讲解了如何使用 Cassandra 自带的命令工具与 Cassandra 进行交互。

第3章

理解 Cassandra 编程接口

本章内容

- 多语言服务开发框架 Thrift
- Cassandra 的数据类型
- Cassandra 的编程接口
- Cassandra 0.7.x 版本新增功能
- 本章小结

第 2 章介绍了 Cassandra 的数据模型，并且使用 Cli 命令行工具对 Cassandra 进行了操作。在本章中，我们将讲解如何使用 Cassandra 提供多语言编程接口、编程接口的约定，以及如何使用 Java 语言来操作 Cassandra。

3.1 多语言服务开发框架 Thrift

开发客户端/服务器端（C/S）程序的时候，除了要编写服务器端的逻辑与通信模块外，还需要分别为不同的客户端编写相应的通信代码。比如服务器端是使用 C++ 编写的，但是服务器端需要交互的客户端可能是多种多样的，如果分别编写 C++ 版本的客户端、Java 版本的客户端以及其他各种语言版本的客户端，将是非常消耗时间的。为了解决这个问题，可以使用 Thrift。

Thrift 是一个多语言服务开发框架。它融合了代码自动生成引擎，通过这个引擎，开发者可以非常快速地开发程序的服务器端，并且自动生成各种客户端的代码，如 C++、Java、Python、PHP、Ruby 等。

Cassandra 提供了多种语言接口，如 Java、C++、C#、Python 等。正是由于使用了 Thrift 框架，Cassandra 的开发者只需要定义客户端与服务器端之间的相关通信的结构体和接口，并提供一个服务契约文件 cassandra. thrift，即可提供多种编程语言接口。

在我们下载的 Cassandra 发行包中，只包含了 Java 的编程接口。假设我们已经安装好了 Thrift，那么通过执行简单的命令就可以获得 C#或 C++ 的编程接口。

生成 C#编程接口如下：

```
thrift -gen csharp cassandra.thrift
```

生成 C++ 编程接口如下：

```
thrift -gen cpp cassandra.thrift
```

在生成编程接口的代码中，thrift 是我们安装的可执行文件，-gen 命令后面带的参数是需要生成代码的种类，csharp 代表 C#，cpp 代表 C++，最后一个参数 cassandra. thrift 代表的是服务的契约文件，这个文件在 Cassandra 的发行包中 interface 目录下。

更多关于 Thrift 的信息，可以登录 http://incubator. apache. org/thrift/进一步获得。

3.2 Cassandra 的数据类型

在 Cassandra 的通信契约文件（cassandra. thrift）中定义了 Cassandra 使用的所有数据类型，这些数据类型对应于我们之前讲到的数据模型中的各个概念，比如 Column、ColumnFamily 等。我们将分别根据 Java 编程接口讲解每一种数据类型。

3.2.1 Column

Column 是 Cassandra 中是最基本的存储单元，用于存储某一行的信息。

Column 在 cassandra. thrift 中的定义如下：

```
struct Column {
  1:required binary name,
  2:required binary value,
  3:required i64 timestamp,
}
```

Column 的 name 和 value 的类型都是 byte，其中 name 用于存储列的名称，value 用于存储列的值，timestamp 用于存储列被更新的时间，类型为 long。

3.2.2　SuperColumn

SuperColumn 是在 Super 类型的 ColumnFamily 中存储数据的单元，SuperColumn 中将包含多个 Column。

SuperColumn 在 cassandra. thrift 中的定义如下：

```
struct SuperColumn {
  1:required binary name,
  2:required list<Column> columns,
}
```

SuperColumn 中，name 用于存储 SuperColumn 的名称，类型为 byte，columns 是存储 Column 的数组。

3.2.3　ColumnOrSuperColumn

ColumnOrSuperColumn 在 cassandra. thrift 中的定义如下：

```
struct ColumnOrSuperColumn {
  1:optional Column column,
  2:optional SuperColumn super_column,
}
```

正如名字所描述的那样，ColumnOrSuperColumn 既可以是一个 Column，也可以是一个 SuperColumn。它的 column 字段和 super_column 中只能一个有值。

由于 Cassandra 所提供的接口中，返回的值可能是 Column 也可能是 SuperColumn，所以使用 ColumnOrSuperColumn 的类型，我们只需要判断究竟是哪个字段有值即可。

3.2.4　ColumnParent

ColumnParent 在 cassandra. thrift 中的定义如下：

```
struct ColumnParent {
  3:required string column_family,
  4:optional binary super_column,
}
```

我们可以将 ColumnParent 理解为文件系统中的一个目录，如果希望查找某一个 key 对应的 value 中，ColumnFamily 所有的值，只要指定 column_ family 字段就可以了。如果我们希望查找某一个 SuperColumn 的所有值，我们不仅需要指定 column_family 字段，还需要指定 super_column 字段。

3.2.5 ColumnPath

ColumnPath 在 cassandra. thrift 中的定义如下：

```
struct ColumnPath {
  3:required string column_family,
  4:optional binary super_column,
  5:optional binary column,
}
```

我们可以将 ColumnPath 理解为文件系统中的一个具体文件，如果希望查找某一个 key 对应的 value 中某一个 ColumnFamily 的某一个 Column 的值，我们需要指定 column_family 字段和 column 字段。如果我们希望查找某一个 SuperColumn 中某一个 Column 的值，我们不仅需要指定 column_family 字段和 super_column 字段，还需要指定 column 字段。

3.2.6 SliceRange

SliceRange 在 cassandra. thrift 中的定义如下：

```
struct SliceRange {
  1:required binary start,
  2:required binary finish,
  3:required bool reversed=0,
  4:required i32 count=100,
}
```

当我们查询某一个 Key 下面的 Value 的时候，可以通过 SliceRange 来指定需要返回的 Column 规则。

其中，start 字段代表按照 ColumnFamily 定义的排序规则需要返回的第一个 Column 名称，finish 字段代表按照 ColumnFamily 定义的排序规则需要返回的最后一个 Column 名称，reversed 字段代表返回的 Column 集合的顺序，如果为 false，则为顺序返回；如果为 true，则为逆序返回，默认为顺序返回。最后一个字段 count 代表返回的结果中 Column 的最大个数，默认为 100 个，如果实际的 Column 数量小于 count，则返回实际的 Column 数量。

假设我们希望返回某一个 Key 下面的所有 Column，那么将 start 和 finish 都设置为 ArrayUtils. EMPTY_BYTE_ARRAY，将 counter 设置为 Integer. MAX_VALUE 即可。

3.2.7 SlicePredicate

SlicePredicate 在 cassandra. thrift 中的定义如下：

```
struct SlicePredicate {
  1:optional list<binary> column_names,
  2:optional SliceRange   slice_range,
}
```

当我们查询某一个 Key 下面的 Value 的时候，可以通过 SlicePredicate 来指定需要返回哪些 Column。

如果希望返回某几个具体的 Column，那么指定 column_ names 字段的值就可以了。如果希望按照某一个规则返回相应的 Column，那么指定 slice_ range 字段就可以了。

3.2.8　Deletion

Deletion 在 cassandra. thrift 中的定义如下：

```
struct Deletion {
  1:required i64 timestamp,
  2:optional binary super_column,
  3:optional SlicePredicate predicate,
}
```

如果希望在 Cassandra 中删除某一个 Key 中的 Column，只需要指定 timestamp 字段和 predicate 字段即可。如果希望删除 Key 中的 SuperColumn 下的 Column，不仅需要指定 timestamp 字段和 predicate 字段，还需要指定需要删除的 super_column 字段。其中，指定的 timestamp 字段为当前的系统时间即可。

3.2.9　Mutation

Mutation 在 cassandra. thrift 中的定义如下：

```
struct Mutation {
  1:optional ColumnOrSuperColumn column_or_supercolumn,
  2:optional Deletion deletion,
}
```

在 Cassandra 中，所有的操作分为两种类型：修改和删除。其中，修改又可以分为插入和修改。我们对 Cassandra 中数据的所有修改都使用 Mutation 来操作。如果是插入新的数据，只需要指定 column_or_supercolumn 字段即可，如果是要修改现有的数据，也是直接指定 column_or_supercolumn 字段。删除数据指定 deletion 字段即可。

3.2.10　KeyRange

KeyRange 在 cassandra. thrift 中的定义如下：

```
struct KeyRange {
  1:optional string start_key,
  2:optional string end_key,
```

```
    3:optional string start_token,
    4:optional string end_token,
    5:required i32 count =100
}
```

Cassandra 不仅支持按照某一个规则查询 Key 下面的 Column 的值，还能够通过 KeyRange 指定一个规则去查询一批 Key 的值。

KeyRange 提供两种查询方式：按照 Key 查询，在 start_key 和 end_key 字段中指定起始的 Key 值即可；按照 token 查询，在 start_token 和 end_token 字段中指定其中的 token 值即可。

这里需要注意的是范围：如果指定 start_key = keyX，end_key = keyX，那么返回的只有 Key 为 keyX 的值；如果指定 start_token = tokenY，end_token = tokenY，那么会返回所有的 Key 值。

3.2.11 KeySlice

KeySlice 在 cassandra. thrift 中的定义如下：

```
struct KeySlice {
    1:required string key,
    2:required list<ColumnOrSuperColumn> columns,
}
```

我们从 Cassandra 查询出来的结果使用 KeySlice 来保存。在 KeySlice 中，字段 key 用来保存返回结果的 Key 值，columns 字段用来保存这个 Key 下面的 column 的信息。

3.2.12 TokenRange

TokenRange 在 cassandra. thrift 中的定义如下：

```
struct TokenRange {
    1:required string start_token,
    2:required string end_token,
    3:required list<string> endpoints,
}
```

我们可以根据 Token 在 Cassandra 查询数据。

通过 TokenRange，我们在 start_token 和 end_token 字段指定开始和结尾的 Token，并通过 endpoints 字段指定需要查询的机器地址即可。

3.2.13 AuthenticationRequest

AuthenticationRequest 在 cassandra. thrift 中的定义如下：

```
struct AuthenticationRequest {
    1:required map<string,string> credentials,
}
```

AuthenticationRequest 包含了我们录入 Cassandra 的信息，其中 credentials 字段的 key 为用户名，credentials 字段的 value 为用户密码。

3.2.14　ConsistencyLevel

ConsistencyLevel 在 cassandra. thrift 中的定义如下：

```
enum ConsistencyLevel {
  ZERO = 0,
  ONE = 1,
  QUORUM = 2,
  DCQUORUM = 3,
  DCQUORUMSYNC = 4,
  ALL = 5,
  ANY = 6,
}
```

在 Cassandra 的读取过程中，支持的一致性级别为 ONE、QUORUM 和 ALL。

假设我们的数据备份级别为 3，会有以下三种结果：

1）如果读取级别为 ONE，那么只要读取到 3 份数据中的 1 份数据就会返回结果。

2）如果读取的级别为 QUORUM，那么需要读取到 3 份数据中的 2 份数据才会返回结果。

3）如果读取的级别为 ALL，那么需要读取到所有的 3 份数据才会返回结果。

在 Cassandra 的写入过程中，支持的一致性级别为 ZERO、ANY、ONE、QUORUM 和 ALL。

假设我们的数据备份级别为 3，会有以下 5 种结果：

1）如果写入级别为 ZERO，那么只要 Cassandra 接受到这个写入请求后，就算写入成功，Cassandra 会在后台完成整个写入过程。

2）如果写入级别为 ANY，只要有任意 1 台 Cassandra 完成了写入操作（包括 hint 写入），就算写入成功。

3）如果写入级别为 ONE，只要有 1 台负责该数据的 Cassandra 完成了写入操作，就算写入成功。

4）如果写入级别为 QUORUM，只要有 2 台负责该数据的 Cassandra 完成了写入操作，就算写入成功。

5）如果写入级别为 ALL，需要 3 台负责该数据的 Cassandra 都完成了写入操作，才能算写入成功。

3.2.15　NotFoundException

NotFoundException 在 cassandra. thrift 中的定义如下：

```
exception NotFoundException {
}
```

如果在 Cassandra 中查询的值不存在，将抛出这个异常。

3.2.16 InvalidRequestException

InvalidRequestException 在 cassandra. thrift 中的定义如下：

```
exception InvalidRequestException {
  1:required string why
}
```

如果我们进行了不可执行的操作，将抛出这个异常。比如，在读取数据的时候，使用的一致性级别为 ConsistencyLevel. ZERO 或者需要查询的数据为 column，但是使用的查询规则为 superColumn。

3.2.17 UnavailableException

UnavailableException 在 cassandra. thrift 中的定义如下：

```
exception UnavailableException {
}
```

如果 Cassandra 在写入或者读取的过程中，发现需要操作的节点数少于实际存活的节点数，将抛出这个异常。

比如我们 Cassandra 的数据备份级别为 3，选择的读取级别为 QUORUM，那么要是实际包含这 3 份数据的机器中，只有一台在提供服务，其他两台机器无法提供服务，就会抛出 UnavailableException 的异常。

3.2.18 TimedOutException

TimedOutException 在 cassandra. thrift 中的定义如下：

```
exception TimedOutException{
}
```

如果 Cassandra 在写入或者读取的过程中，执行时间超过了 RPC 的时间限制（默认为 30 秒钟），那么将抛出这个异常。

3.2.19 AuthenticationException

AuthenticationException 在 cassandra. thrift 中的定义如下：

```
exception AuthenticationException {
  1:required string why
}
```

如果在登入 Cassandra 的过程中，用户名或者密码错误，将抛出这个异常。

3.2.20　AuthorizationException

AuthorizationException 在 cassandra. thrift 中的定义如下：

```
exception AuthorizationException {
  1:required string why
}
```

如果在登入 Cassandra 的过程中，指定的 Keyspace 不存在，将抛出这个异常。

3.3　Cassandra 的编程接口

在 Cassandra 0. 6. x 的版本中，cassandra. thrift 文件定义了 Cassandra 使用的所有编程接口。通过这些编程接口，使得开发人员可以通过编写代码的方式与 Cassandra 进行交互。

3.3.1　get

get 在 cassandra. thrift 中的定义如下：

```
ColumnOrSuperColumn
get(
1:required string keyspace,
2:required string key,
3:required ColumnPath column_path,
4:required ConsistencyLevel consistency_level = ONE
) throws (
1:InvalidRequestException ire,
2:NotFoundException nfe,
3:UnavailableException ue,
4:TimedOutException te),
```

获取某一个 Key 下面的某一个 Column 或者 SuperColumn。

其中 keyspace 为查询的 Keyspace 名称，key 为需要查询的 Key 名称，column_path 为需要查询的 Column 或者 SuperColumn 的路径，consistency_level 为读取一致性级别。

查询 ColumnFamily 下的某一个 Column，我们需要指定 column_path 中的 column_family 和 column。

查询 Super 类型的 ColumnFamily 下的某一个 SuperColumn，我们需要指定 column_path 中的 column_family 和 super_column。

查询 Super 类型的 ColumnFamily 下的某一个 SuperColumn 下的一个 Column，我们需要指定 column_path 中的 column_family、super_column 和 column。

这个方法返回一个 ColumnOrSuperColumn，如果 column 字段有值，代表结果是 Column；

如果 super_column 有值，代表结果是 SuperColumn。

3.3.2 get_slice

get_slice 在 cassandra. thrift 中的定义如下：

```
list <ColumnOrSuperColumn >
get_slice(
1:required string keyspace,
2:required string key,
3:required ColumnParent column_parent,
4:required SlicePredicate predicate,
5:required ConsistencyLevel consistency_level = ONE
) throws (
1:InvalidRequestException ire,
2:UnavailableException ue,
3:TimedOutException te),
```

按照指定规律获取某一个 Key 下面的 Column 或者 SuperColumn。

其中 keyspace 为查询的 Keyspace 名称，key 为需要查询的 Key 名称，column_parent 为需要查询的 Column 或者 SuperColumn 的上层路径，predicate 为 Column 的查询规则，consistency_level 为读取一致性级别。

查询 ColumnFamily 下的 Column 或 SuperColumn，我们需要指定 column_parent 中的 column_family。

查询 ColumnFamily 中的某一个 SuperColumn 下的 Column，我们需要指定 column_parent 中的 column_family 和 super_column。

这个方法返回一个 ColumnOrSuperColumn 数组，如果数组中 ColumnOrSuperColumn 的 column 字段有值，代表结果是 Column；如果 super_column 有值，代表结果是 SuperColumn。

3.3.3 multiget_slice

multiget_slice 在 cassandra. thrift 中的定义如下：

```
map <string,list <ColumnOrSuperColumn >>
multiget_slice(
1:required string keyspace,
2:required list <string > keys,
3:required ColumnParent column_parent,
4:required SlicePredicate predicate,
5:required ConsistencyLevel consistency_level = ONE
) throws (
1:InvalidRequestException ire,
2:UnavailableException ue,
3:TimedOutException te),
```

按照指定规律获取一批 Key 下面的 Column 或者 SuperColumn。

其中 keyspace 为查询的 Keyspace 名称，keys 为需要批量查询的 Key 名称，column_parent 为需要查询的 Column 或者 SuperColumn 的上层路径，predicate 为 Column 的查询规则，consistency_level 为读取一致性级别。

这个方法返回一个 Map，Map 的 key 为 Key 名称，Map 的 value 为 ColumnOrSuperColumn，如果 ColumnOrSuperColumn 的 column 字段有值，代表结果是 Column；如果 super_column 有值，代表结果是 SuperColumn。

3.3.4　get_count

get_count 在 cassandra. thrift 中的定义如下：

```
i32
get_count(
1:required string keyspace,
2:required string key,
3:required ColumnParent column_parent,
4:required ConsistencyLevel consistency_level = ONE
) throws (
1:InvalidRequestException ire,
2:UnavailableException ue,
3:TimedOutException te),
```

按照指定规律获取某一个 Key 下面的 Column 或者 SuperColumn 的个数。

其中 keyspace 为查询的 Keyspace 名称，key 为需要查询的 Key 名称，column_parent 为需要查询的 Column 或者 SuperColumn 的上层路径，predicate 为 Column 的查询规则，consistency_level 为读取一致性级别。

这个方法返回按照指定规律获取某一个 Key 下面的 Column 或者 SuperColumn 的个数。

3.3.5　get_range_slices

get_range_slices 在 cassandra. thrift 中的定义如下：

```
list<KeySlice>
get_range_slices(
1:required string keyspace,
2:required ColumnParent column_parent,
3:required SlicePredicate predicate,
4:required KeyRange range,
5:required ConsistencyLevel consistency_level = ONE
) throws (
1:InvalidRequestException ire,
2:UnavailableException ue,
3:TimedOutException te),
```

按照指定规律获取一批 Key 下面的 Column 或者 SuperColumn。

其中 keyspace 为查询的 Keyspace 名称，column_parent 为需要查询的 Column 或者 SuperColumn 的上层路径，predicate 为 Column 的查询规则，range 为 Key 的查询规则，consistency_level 为读取一致性级别。

这个方法返回一个 KeySlice 数组，数组中 KeySlice 中的 key 为 Key 的名称，KeySlice 中的 columns 数组为 Key 中对应的 Column 或 SuperColumn。

3.5.6 insert

insert 在 cassandra. thrift 中的定义如下：

```
void
insert(
1:required string keyspace,
2:required string key,
3:required ColumnPath column_path,
4:required binary value,
5:required i64 timestamp,
6:required ConsistencyLevel consistency_level = ONE
) throws (
1:InvalidRequestException ire,
2:UnavailableException ue,
3:TimedOutException te),
```

将一个 Column 写入 Cassandra 中。

其中 keyspace 为写入的 Keyspace 名称，key 为需要写入 Key 的名称，column_ path 为需要写入的 Column 或者 SuperColumn 的路径，timestamp 为写入的时间，consistency_level 为写入一致性级别。

如果这个方法没有抛出异常，那么写入就成功了。

3.3.7 remove

remove 在 cassandra. thrift 中的定义如下：

```
void
remove(
1:required string keyspace,
2:required string key,
3:required ColumnPath column_path,
4:required i64 timestamp,
5:ConsistencyLevel consistency_level = ONE
) throws (
1:InvalidRequestException ire,
2:UnavailableException ue,
3:TimedOutException te),
```

将一个 Column 从 Cassandra 中删除。

其中 keyspace 为删除的 Keyspace 名称，key 为需要删除 Key 的名称，column_ path 为需要删除的 Column 或者 SuperColumn 的路径，timestamp 为删除的时间，consistency_ level 为删除一致性级别。

如果这个方法没有抛出异常，那么删除就成功了。

3.3.8 batch_mutate

batch_mutate 在 cassandra. thrift 中的定义如下：

```
void
batch_mutate(
1:required string keyspace,
2:required map<string,map<string,list<Mutation>>>mutation_map,
3:required ConsistencyLevel consistency_level=ONE
) throws (
1:InvalidRequestException ire,
2:UnavailableException ue,
3:TimedOutException te),
```

批量将 Column 写入 Cassandra 中。

其中 keyspace 为写入的 Keyspace 名称，key 为需要写入 Key 的名称，mutation_map 为需要写入的 Column 或者 SuperColumn 集合，consistency_level 为写入一致性级别。

mutation_map 的 key 为 ColumnFamily 的名称，value 为需要在这个 ColumnFamily 做的修改集合，这个修改集合包括添加、修改和删除。

如果这个方法没有抛出异常，那么写入就成功了。

3.3.9 describe_keyspaces

describe_keyspaces 在 cassandra. thrift 中的定义如下：

```
set<string> describe_keyspaces(),
```

获取 Cassandra 中所有的 Keyspace 的描述信息。

3.3.10 describe_keyspace

describe_keyspace 在 cassandra. thrift 中的定义如下：

```
map<string,map<string,string>>
describe_keyspace(1:required string keyspace)
throws (1:NotFoundException nfe),
```

获取 Cassandra 中某一个 Keyspace 的描述信息。

其中，keyspace 为需要获取描述信息的 Keyspace 的名称。

在返回的结果（map < string，map < string，string >> ）中，第一个 string 为 ColumnFamily 的名称，第二个 string 和第三个 string 分别为 ColumnFamily 的属性名称和属性的值。

3.3.11 describe_cluster_name

describe_cluster_name 在 cassandra. thrift 中的定义如下：

```
string describe_cluster_name(),
```

获取 Cassandra 集群的名称。

3.3.12 describe_version

describe_version 在 cassandra. thrift 中的定义如下：

```
string describe_version(),
```

获取 Cassandra 的版本信息。

3.3.13 describe_ring

describe_ring 在 cassandra. thrift 中的定义如下：

```
list <TokenRange > describe_ring(1 :required string keyspace),
```

获取 Cassandra 集群中某一个 Keyspace 的节点之间的 token 信息。

其中 keyspace 为需要获取集群节点信息的 Keyspace 的名称。

3.4 Cassandra 0.7.x 版本新增功能

在 Cassandra 0.7.x 版本中，新增加了两项主要的功能：

- 二级索引
- 动态修改 Schema

为了实现这两项新功能，Cassandra 也在新版本的 cassandra. thrift 文件中定义了相关的数据类型和编程接口。

3.4.1 二级索引

在 Cassandra 中，对列值（column values）的索引叫做“二级索引”，它与列簇（ColumnFamilies）中对 Key 的索引不同。二级索引允许我们对列值进行查询，并且在读取和写入的时候不会引起操作阻塞。

1. 数据类型

在 cassandra. thrift 文件中定义的与二级索引相关的数据类型如下：

```
enum IndexOperator {
  EQ,
  GTE,
  GT,
  LTE,
  LT
}
struct IndexExpression {
  1:required binary column_name,
  2:required IndexOperator op,
  3:required binary value,
}
struct IndexClause {
  1:required list<IndexExpression> expressions
  2:required binary start_key,
  3:required i32 count=100,
}
enum IndexType {
  KEYS,
}
```

IndexOperator 定义了二级索引比较的关系：EQ 代表“=”，GTE 代表“>=”，GT 代表“>”，LTE 代表“<=”，LT 代表“<”。

IndexExpression 定义了进行二级索引查询的时候，需要查找的 Column 的相关条件。

1）column_name：需要进行查询的 Column 的名称。

2）op：需要进行查询的 Column 的值的比较关系。

3）value：需要进行查询的 Column 的值。

IndexClause 定义了进行二级索引查询的时候，需要查询的条件。

1）expressions：进行二级索引查询的条件集合。

2）start_key：进行二级索引查询时，开始筛选的起始 Key。

3）count：返回查询结果的最大个数，默认值为 100。

IndexType 定义了在 Column 中建立二级索引的类型。在 Cassandra 0.7.0 的版本中，只支持 KEYS 类型的二级索引。在 Cassandra 后续的版本中，将提供其他类型的二级索引，如 bitmap。

2. 编程接口

在 cassandra.thrift 文件中定义的与二级索引相关的编程接口如下：

```
list<KeySlice> get_indexed_slices(
1:required ColumnParent column_parent,
2:required IndexClause index_clause,
3:required SlicePredicate column_predicate,
```

```
4:required ConsistencyLevel consistency_level = ConsistencyLevel.ONE)
throws (1:InvalidRequestException ire,
2:UnavailableException ue,3:TimedOutException te),
```

通过二级索引对指定的 Column 进行查询。

如果希望对某一列建立二级索引，可以参考下面讲解的动态修改 Schema，在修改 ColumnDef 信息时，指定二级索引的信息。

3. 简单示例

理解二级索引最好的方式就是用实际的例子来说明。在这个例子中，我们使用 Cassandra 自带的命令行工具（CLI）进行操作，并且使用一个名为 users 的列簇。

```
$ bin/cassandra-cli --host localhost
Connected to:"Test Cluster" on localhost/9160
Welcome to cassandra CLI.

Type'help;'or '?'for help. Type'quit;'or'exit;'to quit.
[default@ unknown] create keyspace demo;
[default@ unknown] use demo;
[default@ demo] create column family users with comparator=UTF8Type
and column_metadata=[{column_name:full_name,validation_class:UTF8Type},
{column_name:birth_date,validation_class:LongType,index_type:KEYS}];
```

在上面的示例中，我们定义了一个 Keyspace，名为 demo，然后又定义了一个列簇，名为 users，最后在这个列簇中，对两个列定义了二级索引 full_name 和 birth_date。

接下来，我们在列簇 users 中添加一些测试数据。

```
[default@ demo] set users[bsanderson][full_name]='Brandon Sanderson';
[default@ demo] set users[bsanderson][birth_date]=1975;
[default@ demo] set users[prothfuss][full_name]='Patrick Rothfuss';
[default@ demo] set users[prothfuss][birth_date]=1973;
[default@ demo] set users[htayler][full_name]='Howard Tayler';
[default@ demo] set users[htayler][birth_date]=1968;
```

现在可以在 Cassandra 对刚刚写入的测试数据进行查询了。

```
[default@ demo] get users where birth_date=1973;
-------------------
RowKey:prothfuss
=> (column=birth_date,value=1973,timestamp=1291333944389000)
=> (column=full_name,value=Patrick Rothfuss,timestamp=1291333940538000)
```

现在假设有这样一个需求，我们想要对 State 这个列进行查询。要处理这个需求，只需要在 users 列簇中再添加一个二级索引即可。

首先，添加一些测试数据如下：

```
[default@ demo] set users[bsanderson][state] ='UT';
[default@ demo] set users[prothfuss][state] ='WI';
[default@ demo] set users[htayler][state] ='UT';
```

虽然现在 state 列还没有索引，但是依旧可以与其他包含索引的列进行联合查询。

```
[default@ demo] get users where state ='UT';
No indexed columns present in index clause with operator EQ

[default@ demo] get users where state ='UT'and birth_date > 1970;
No indexed columns present in index clause with operator EQ

[default@ demo] get users where birth_date =1968 and state ='UT';
-------------------
RowKey:htayler
=> (column = birth_date,value =1968,timestamp =1291334765649000)
=> (column = full_name,value = Howard Tayler,timestamp =1291334749160000)
=> (column = state,value =5554,timestamp =1291334890708000)
```

这里需要注意一点，说 KEYS 索引更像哈希索引而不是 BTree 索引的原因就在这里：即使 birth_date 有二级索引，但是 Cassandra 依旧不能提供范围查询，比如" > 1970"。

最后，我们给 state 列添加二级索引，这样就可以对 state 列的值进行单独的查询了。

```
[default@ demo] update column family users with comparator = UTF8Type
and column_metadata = [{column_name:full_name,validation_class:UTF8Type},
{column_name:birth_date,validation_class:LongType,index_type:KEYS},
{column_name:state,validation_class:UTF8Type,index_type:KEYS}];
```

添加完索引后，对 state 列的值进行单独的查询。

```
[default@ demo] get users where state ='UT';
--------------------
RowKey:bsanderson
=> (column = birth_date,value =1975,timestamp =1291333936242000)
=> (column = full_name,value = Brandon Sanderson,timestamp =1291333931790000)
=> (column = state,value = UT,timestamp =1291334909266000)
--------------------
RowKey:htayler
=> (column = birth_date,value =1968,timestamp =1291334765649000)
=> (column = full_name,value = Howard Tayler,timestamp =1291334749160000)
=> (column = state,value = UT,timestamp =1291334890708000)

[default@ demo] get users where state ='UT'and birth_date > 1970;
--------------------
RowKey:bsanderson
=> (column = birth_date,value =1975,timestamp =1291333936242000)
=> (column = full_name,value = Brandon Sanderson,timestamp =1291333931790000)
```

```
=> (column = state,value = UT,timestamp = 1291334909266000)
```

由于查询的列都包含二级索引，所以现在 Cassandra 可以进行范围查询了。在 Python 的客户端 pycassa 中，操作如下：

```
state_expr = pycassa.create_index_expression('state','UT')
birth_expr = pycassa.create_index_expression('birth_date',
1970,op = IndexOperator.GT)
clause = pycassa.create_index_clause([state_expr,bday_expr])
result = users.get_indexed_slices(clause):
```

在 Java 的客户端 Hector 中，操作如下：

```
StringSerializer ss = StringSerializer.get();
IndexedSlicesQuery < String,String,String > indexedSlicesQuery =
HFactory.createIndexedSlicesQuery(keyspace,ss,ss,ss);
indexedSlicesQuery.setColumnNames("full_name","birth_date","state");
indexedSlicesQuery.addGtExpression("birth_date",1970L);
indexedSlicesQuery.addEqualsExpression("state","UT");
indexedSlicesQuery.setColumnFamily("users");
indexedSlicesQuery.setStartKey("");
QueryResult < OrderedRows < String,String,String >> result =
indexedSlicesQuery.execute();
```

3.4.2 动态修改 Schema

ColumnFamily 在 Cassandra 中是没有 Schema 的，可以在其中创建任何 Column。但是我们必须在 Cassandra 中预先定义好所有需要使用的 ColumnFamily 和 Keyspace，只有这样，Cassandra 才知道集群中究竟有哪些 ColumnFamily 和 Keyspace 可用以及如何使用。

动态修改 Schema，是在 Cassandra 集群运行和维护的时候，不停止 Cassandra 集群，动态地添加、修改和删除 ColumnFamily 以及 Keyspace。

动态修改 Sechma 改变了之前 Cassandra 修改 ColumnFamily 和 Keyspace 的方式，修改 ColumnFamily 和 Keyspace 不再需要停止集群中的所有的节点，然后手动修改每一个节点的配置信息。这样大大地提高了集群的使用效率，同时降低了人为操作失误的可能。

1. 数据类型

在 cassandra. thrift 文件中定义的与动态修改 Schema 相关的数据类型如下：

```
struct ColumnDef {
  1:required binary name,
  2:required string validation_class,
  3:optional IndexType index_type,
  4:optional string index_name
}
struct CfDef {
```

```
    1:required string keyspace,
    2:required string name,
    3:optional string column_type = "Standard",
    5:optional string comparator_type = "BytesType",
    6:optional string subcomparator_type,
    8:optional string comment,
    9:optional double row_cache_size =0,
    11:optional double key_cache_size =200000,
    12:optional double read_repair_chance =1.0,
    13:optional list <ColumnDef > column_metadata,
    14:optional i32 gc_grace_seconds,
    15:optional string default_validation_class,
    16:optional i32 id,
    17:optional i32 min_compaction_threshold,
    18:optional i32 max_compaction_threshold,
    19:optional i32 row_cache_save_period_in_seconds,
    20:optional i32 key_cache_save_period_in_seconds,
    21:optional i32 memtable_flush_after_mins,
    22:optional i32 memtable_throughput_in_mb,
    23:optional double memtable_operations_in_millions,
}
struct KsDef {
    1:required string name,
    2:required string strategy_class,
    3:optional map <string,string > strategy_options,
    4:required i32 replication_factor,
    5:required list <CfDef > cf_defs,
}
```

在上面数据类型的定义中，ColumnDef 代表 Column 的元数据信息，其中包括了二级索引的信息。CfDef 代表 ColumnFamily 的元数据信息，所有在配置文件中对 ColumnFamily 的配置都可以通过 CfDef 表示。KsDef 代表 Keyspace 的元数据信息。

2. 编程接口

在 cassandra. thrift 文件中定义的与动态修改 Schema 相关的编程接口如下：

```
string system_add_column_family(1:required CfDef cf_def)
  throws (1:InvalidRequestException ire),
string system_drop_column_family(1:required string column_family)
  throws (1:InvalidRequestException ire),
string system_add_keyspace(1:required KsDef ks_def)
  throws (1:InvalidRequestException ire),
string system_drop_keyspace(1:required string keyspace)
  throws (1:InvalidRequestException ire),
string system_update_keyspace(1:required KsDef ks_def)
```

```
    throws (1:InvalidRequestException ire),
  string system_update_column_family(1:required CfDef cf_def)
    throws (1:InvalidRequestException ire),
```

在上面编程接口的定义中 system_add_column_family 代表添加 ColumnFamily，system_drop_column_family 代表删除 ColumnFamily，system_update_column_family 代表更新 ColumnFamily，system_add_keyspace 代表添加 Keyspace，system_drop_keyspace 代表删除 Keyspace，system_update_keyspace 代表更新 Keyspace。

3. 简单示例

假设现在 Cassandra 集群已经启动，我们可以通过 Cli 程序在集群中创建一个新的 Keyspace，取名为 Keyspace1，并且在这个 Keyspace1 中创建一个 ColumnFamily，取名为 Users，排序的规则为 UTF8Type。

```
[default@ unknown] create keyspace Keyspace1;
ece86bde-dc55-11df-8240-e700f669bcfc
[default@ unknown] use Keyspace1;
Authenticated to keyspace:Keyspace1
[default@ Keyspace1] create column family Users with comparator=UTF8Type and default_validation_class=UTF8Type;
737c7a71-dc56-11df-8240-e700f669bcfc
```

3.4.3 自动清除过期数据

当数据在 Cassandra 保存的时间超过一定的长度后，就会成为过期的数据。另外我们也不可能将所有的数据都保存起来，毕竟存储空间是有限的。所以，我们就需要一种能够自动将过期数据清除的机制。

在绝大多数的数据库系统中，清除过期数据的方式就是定期运行和维护一个任务，扫描数据库中的所有数据，看看哪些数据已经过期了，然后再将这些数据删除。这种方式虽然能够解决一些问题，但是却并不完美，原因如下：

1）有些数据可能是一周过期，有的数据可能是一个月才过期。

2）每一次清除数据都需要扫描大量的数据集合。

3）不能及时将空间清除，需要定时去扫描。

在 Cassandra 0.7.x 中，有更好的一种机制帮助我们清除过期的数据。

在将数据写入 Cassandra 中的时候，可以指定这条数据过期的时间 TTL（time to live）。那么这条数据将会在相应的时间过后，被系统自动删除。

1. 数据类型

在 cassandra.thrift 文件中定义的 Column 的数据类型如下：

```
struct Column {
  1:required binary name,
```

```
  2:required binary value,
  3:required i64 timestamp,
  4:optional i32 ttl,
}
```

在上面数据类型的定义中，与 Cassandra 0.6.x 相比，只是多了一个属性 ttl，在这个参数中指定数据过期的时间即可。经过相应的时间间隔，数据将会被系统自动删除。

其中 ttl 参数的单位为秒。

2. 简单示例

现在通过 Cli 程序在 Cassandra 集群中创建一个名为 demo 的 Keyspace，和一个名为 test 的 ColumnFamily。

```
Connected to:"Test Cluster" on localhost/9160
Welcome to cassandra CLI.

Type'help;'or'?'for help. Type'quit;'or'exit;'to quit.
[default@ unknown] create keyspace demo;
[default@ unknown] use demo;
[default@ demo] create column family test with comparator = UTF8Type and default_
validation_class = UTF8Type;
```

接下来，向 Cassandra 中插入两条数据，其中一条是正常数据，不指定自动删除的时间，另一条指定自动删除的时间为 60 秒。

```
[default@ demo] set test[row1][col1] ='val1';
[default@ demo] set test[row1][col2] ='val2'with ttl = 60;
[default@ demo] get test[row1];
=> (column = col1,value = val1,timestamp = 1291980736812000)
=> (column = col2,value = val2,timestamp = 1291987837942000,ttl = 60)
Returned 2 results.
```

可以看到，这两条数据都已经成功插入到 Cassandra 系统中了。我们等待 60 秒钟，然后再来查询刚刚插入的数据。

```
[default@ demo] get test[row1];
=> (column = col1,value = val1,timestamp = 1291980736812000)
```

之前插入的两条数据已经只剩下一条了，另一条数据已经被系统自动删除了。

3.5　本章小结

本章讲解了跨语言服务开发框架 Thrift，介绍了 Cassandra 提供的数据类型与编程接口，并且详细讲解了 Cassandra 0.7.x 新增的功能。在第 4 章中，我们将实践 Cassandra，设计并实现博客程序的后台逻辑。

第 4 章

基于 Cassandra 的在线交易系统

本章内容

- 需求分析
- 数据模型设计
- 编码实现
- 系统功能验证
- 迁移到 Cassandra 0.7.x
- 本章小结

在第 2 章和第 3 章中，我们了解了如何安装和配置 Cassandra，以及 Cassandra 所提供的编程接口。在本章中，我们将基于 Cassandra 0.6.x 的版本设计和实现一个在线交易系统，并将实现的在线交易系统修改为 Cassandra 0.7.x 的实现版本。

4.1 需求分析

在线交易系统的需求归纳为以下几点：

1）可供买家和卖家使用。

2）卖家出售的产品可以根据产品的种类进行分类，并且按照上架时间的先后顺序排序。

3）买家可以对产品进行评价，评价也是按照时间的先后顺序排序的。

4）买家的属性包括：用户名、姓名、年龄、性别、买家地址。

5）卖家的属性包括：用户名、姓名、年龄、性别、卖家地址。

6）产品的属性包括：卖家的用户名、产品名称、产品描述、产品价格。

7）评论的属性包括：买家的用户名、评论内容。

系统中需要 4 类实体对象，分别是：买家、卖家、产品和评论。

买家可以添加评论，卖家可以按照分类添加产品。

4.2 数据模型设计

根据之前的需求分析，系统需要为每一个实体对象建立一个相应的 ColumnFamily。同时注意到两个特殊的实体对象：产品和评论，它们不仅包含有自身的固有属性，同时也包含了与其他实体对象的关系，如评论和买家的关系，以及卖家和产品分类的关系。要维护实体之间的关系，我们可以通过 Super ColumnFamily 和单独的 ColumnFamily 来实现。

本系统的数据模型由 5 个 ColumnFamily 组成：Seller、Buyer、Product、ProductCategory 和 Comment。

4.2.1 Seller

Seller 用于保存卖家的信息，其中卖家的用户名作为 ColumnFamily 的 Key，该卖家的每一个属性都用一个 Column 来存储。Column 的名称为相应属性的名称，Column 的值为相应属性的值。

Seller 在 Cassandra 中的配置如下：

```
<ColumnFamily Name = "Seller"
              CompareWith = "UTF8Type" />
```

当实际有数据存储的时候，情况如下：

```
Seller = { //ColumnFamily
    aaron:{   //第一个卖家
        {name:"name",value:"郭鹏",timestamp:123456789},
        {name:"age",value:"26",timestamp:123456789},
        {name:"sex",value:"男",timestamp:123456789},
        {name:"address",value:"浙江省杭州市",timestamp:123456789}
    },
    lily:{   //第二个卖家
        {name:"name",value:"李娜",timestamp:123456789},
        {name:"age",value:"23",timestamp:123456789},
        {name:"sex",value:"女",timestamp:123456789},
        {name:"address",value:"辽宁省大连市",timestamp:123456789}
    }
    …
}
```

4.2.2　Buyer

Buyer 用于保存买家的信息，其中买家的用户名作为 ColumnFamily 的 Key，该买家的每一个属性都用一个 Column 来存储。其中 Column 的名称为相应属性的名称，Column 的值为相应属性的值。

Buyer 在 Cassandra 中的配置如下：

```
<ColumnFamily Name = "Buyer"
              CompareWith = "UTF8Type" />
```

当实际有数据存储的时候，情况如下：

```
Buyer = { //ColumnFamily
    gpcuster:{   //第一个买家
        {name:"name",value:"张斌",timestamp:123456789},
        {name:"age",value:"29",timestamp:123456789},
        {name:"sex",value:"男",timestamp:123456789},
        {name:"address",value:"湖南省岳阳市",timestamp:123456789}
    },
    lily:{   //第二个买家
        {name:"name",value:"王菲菲",timestamp:123456789},
        {name:"age",value:"38",timestamp:123456789},
        {name:"sex",value:"女",timestamp:123456789},
        {name:"address",value:"北京市",timestamp:123456789}
    }
    …
}
```

4.2.3　Product

Product 用于保存产品的信息，由于产品要在产品分类中根据产品上架的时间排序，所

以我们使用 TimeUUID 作为 ColumnFamily 的 Key，该产品的每一个属性都用一个 Column 来存储。其中 Column 的名称为相应属性的名称，Column 的值为相应属性的值。

Product 在 Cassandra 中的配置如下：

```
<ColumnFamily Name = "Product"
                    CompareWith = "UTF8Type" />
```

当实际有数据存储的时候，情况如下：

```
Product = { //ColumnFamily
    8177ec99 - b9df - 11df - 94a6 - 1b9323c915f2:{   //第一个产品
        {name:"name",value:"足球",timestamp:123456789},
        {name:"sellerUserName",value:"gpcuster",timestamp:123456789},
        {name:"desc",value:"白色真皮足球",timestamp:123456789},
        {name:"price",value:"98.8",timestamp:123456789}
    },
    684545e0 - b9df - 11df - 9a02 - 43b9b8ba2a56:{   //第二个产品
        {name:"name",value:"连衣裙",timestamp:123456789},
        {name:"sellerUserName",value:"lily",timestamp:123456789},
        {name:"desc",value:"粉红真丝",timestamp:123456789},
        {name:"price",value:"298.8",timestamp:123456789}
    }
    ...
}
```

4.2.4 ProductCategory

ProductCategory 用于保存产品的分类信息，产品类别的名称作为 ColumnFamily 的 Key，该产品分类下的每一个产品用一个 Column 来存储。其中 Column 的名称为相应产品的 TimeUUID，Column 的值为空。同时要求在产品分类下产品按照上架的时间排序，所以 ColumnFamily 选用的排序规则为 TimeUUIDType。

ProductCategory 在 Cassandra 中的配置如下：

```
<ColumnFamily Name = "ProductCategory"
                    CompareWith = "TimeUUIDType" />
```

当实际有数据存储的时候，情况如下：

```
ProductCategory = { //ColumnFamily
    Sport:{   //体育类产品
        (column = 8177ec99 - b9df - 11df - 94a6 - 1b9323c915f2,value:,timestamp =
1283795544638),
        (column = 684545e0 - b9df - 11df - 9a02 - 43b9b8ba2a56,value:,timestamp =
1283795502361),
        ...
    },
```

```
    Dress:{   //服装类产品
        (column = 864051d2 - b9dc - 11df - 8094 - 0f1e51755e0e,value:,timestamp =
1283795544638),
        (column = 7ebaa921 - b9dc - 11df - 95e1 - 55f06b262a7b,value:,timestamp =
1283795502361),
        …
    }
    …
}
```

4.2.5　Comment

Comment用于保存评论的信息，Super ColumnFamily的Key为被评论产品的TimeUUID，Super Column的名称为评论自身的TimeUUID，同时要求评论按照时间排序，所以Super Column选用的排序规则为TimeUUIDType。Super Column下的每一个Column用来存储评论的相应属性，其中Column的名称为相应属性的名称，Column的值为相应属性的值。

Comment在Cassandra中的配置如下：

```
<ColumnFamily Name = "Comment"
              ColumnType = "Super"
              CompareWith = "TimeUUIDType"
              CompareSubcolumnsWith = "UTF8Type"/>
```

当实际有数据存储的时候，情况如代码清单4-1所示。

代码清单4-1　Comment数据模型示例

```
Comment = { //Super ColumnFamily
    8177ec99 - b9df - 11df - 94a6 - 1b9323c915f2:{    //第一个产品的TimeUUID
        {
            name:"864051d2 - b9dc - 11df - 8094 - 0f1e51755e0e",//评论的TimeUUID
            value:{
                {name:"content",value:"容易掉颜色",timestamp:123456789},
                {name:"commentUserName",value:"aaron",timestamp:123456789},
            }
        },
        {
            name:"864051d2 - b9dc - 11df - 8094 - 55f06b262a7b",//评论的TimeUUID
            value:{
                {name:"content",value:"送货很快",timestamp:123456789},
                {name:"commentUserName",value:"lily",timestamp:123456789},
            }
        },
        …
    },
    684545e0 - b9df - 11df - 9a02 - 43b9b8ba2a56:{    //第二个产品的TimeUUID
        {
            name:"7ebaa921 - b9dc - 11df - 95e1 - 55f06b262a7b",//评论的TimeUUID
```

```
            value:{
                {name:"content",value:"质量不错",timestamp:123456789},
                {name:"commentUserName",value:"max Wang",timestamp:123456789},
            }
        },
        ...
    }
    ...
}
```

4.3 编码实现

完成数据模型的设计，修改 Cassandra 的 Keyspace 配置，就可以开始进行编码实现了。

4.3.1 修改 Keyspace 设置

打开 conf/storage-conf. xml 文件，在 Keyspace 配置项中，新添加一个 Keyspace 的配置信息，内容如下：

```
<Keyspace Name = "CassSeller" >
  <ColumnFamily Name = "Seller"
                CompareWith = "UTF8Type" />
  <ColumnFamily Name = "Buyer"
                CompareWith = "UTF8Type" />
  <ColumnFamily Name = "ProductCategory"
                CompareWith = "TimeUUIDType" />
  <ColumnFamily Name = "Product"
                CompareWith = "UTF8Type" />
  <ColumnFamily Name = "Comment"
                ColumnType = "Super"
                CompareWith = "TimeUUIDType"
                CompareSubcolumnsWith = "UTF8Type"/>

<ReplicaPlacementStrategy > org.apache.cassandra.locator.RackUnawareStrategy
</ReplicaPlacementStrategy >
    <ReplicationFactor >1 </ReplicationFactor >
<EndPointSnitch >org.apache.cassandra.locator.EndPointSnitch </EndPointSnitch >
  </Keyspace >
```

编辑完成后，保存并重新启动 Cassandra。

4.3.2 建立 Eclipse 项目

建立一个名为 CassSeller 的 Java 项目，将下载的 Cassnadra 发行版中 lib 目录下的需要使用的 jar 包添加到项目依赖包中。由于 Product 和 Comment 实体对象都使用了 TimeUUID，所

以需要到 http://jug.safehaus.org/Download 额外下载一个开源 TimeUUID 实现 jar 包 jug-asl-2.0.0.jar，并加入到项目的依赖包中。

依赖包的结构如图 4-1 所示。

在代码结构中，我们建立 4 个包，分别为：cassSeller.app、cassSeller.dao、cassSeller.dao.impl、cassSeller.model。其中，cassSeller.model 为实体对象定义，cassSeller.dao 为操作 Cassandra 的接口定义，cassSeller.dao.impl 为操作 Cassandra 的接口实现，cassSeller.app 为测试应用逻辑。

Eclipse 项目建立完毕以后，项目结构如图 4-2 所示。

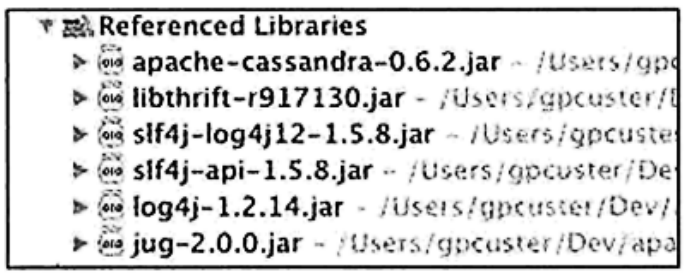


图 4-1　CassSeller 依赖包结构

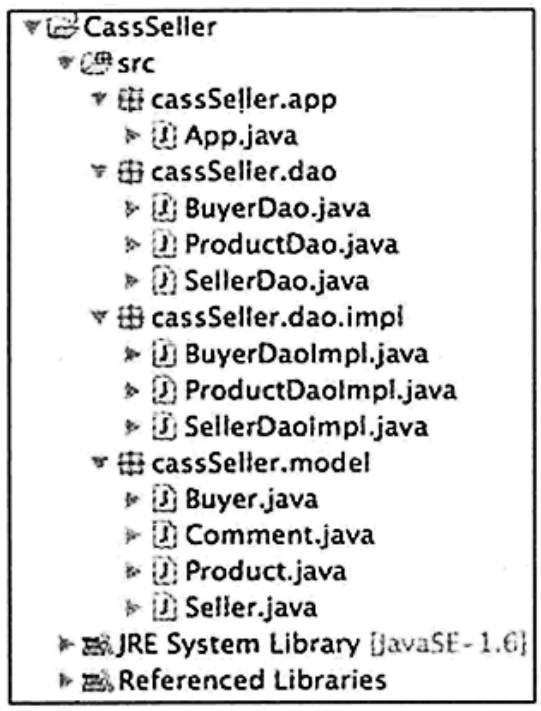


图 4-2　CassSeller 项目结构

4.3.3　实体对象实现

根据之前的数据模型定义，我们实现 4 个实体对象，都用 POJO（简单 Java 对象）形式实现，除了基本的属性定义之外，还有和每一个属性相对应的 getxxx 和 setxxx 方法。

Product 包含的属性如下：

```
private UUID uuid;
private String name;
private String sellerUserName;
private String desc;
private double price;
```

其中需要注意的是 uuid 这个属性，它的类型为 org.safehaus.uuid.UUID，并不是 JDK 中默认的 java.util.UUID 类型，本章中后面所有的 UUID 类型都是 org.safehaus.uuid.UUID。

Buyer 包含的属性如下：

```
private String userName;
```

```
private String name;
private int age;
private String sex;
private String address;
```

Seller 包含的属性如下：

```
private String userName;
private String name;
private int age;
private String sex;
private String address;
```

Comment 包含的属性如下：

```
private UUID uuid;
private String content;
private String commentUserName;
```

4.3.4 Cassandra 数据操作接口实现

实现的数据操作接口分为 3 个：BuyerDao、SellerDao 和 ProductsDao。BuyerDao 负责存取 Buyer 实例，接口的定义如下：

```
/**
 * 将 Buyer 实例插入到 Cassandra 中
 *
 * @ param buyer
 * 需要插入的 Buyer 实例
 * @ throws Exception
 */
public void insertBuyer(Buyer buyer) throws Exception;

/**
 * 根据 buyerUserName,从 Cassandra 中查询对应的 Buyer 实例
 *
 * @ param buyerUserName
 * Buyer ColumnFalimy 的 key
 * @ return Buyer 实例
 * @ throws Exception
 */
public Buyer getBuyer(String buyerUserName) throws Exception;
```

SellerDao 负责存取 Seller 实例，接口的定义如下：

```
/**
 * 将 Seller 实例插入到 Cassandra 中
 *
```

```
 * @param seller
 * 需要插入的Seller实例
 * @throws Exception
 */
public void insertSeller(Seller seller) throws Exception;

/**
 * 根据sellerUserName,从Cassandra中查询对应的Seller实例
 *
 * @param sellerUserName
 * Seller ColumnFamily的key
 * @return Seller实例
 * @throws Exception
 */
public Seller getSeller(String sellerUserName) throws Exception;
```

ProductDao不仅负责存取Product实例，还负责Product分类与评价的存取操作。接口的定义如代码清单4-2所示。

代码清单4-2　ProductDao接口定义

```
/**
 * 将Product实例插入到Cassandra中
 *
 * @param product
 *            需要插入的Product实例
 * @throws Exception
 */
public void insertProduct(Product product) throws Exception;

/**
 * 将产品分类信息插入到Cassandra中
 *
 * @param category
 * 产品的分类
 * @param productUUID
 * Product ColumnFamily的key
 * @throws Exception
 */
public void insertProductCategory(String category,UUID productUUID)
            throws Exception;

/**
 * 将产品评论信息插入到Cassandra中
 *
 * @param productUUID
 * Product ColumnFamily的key
 * @param comment
 * 评论的内容
 * @throws Exception
 */
public void insertComment(UUID productUUID,Comment comment)
```

```
        throws Exception;

/**
 * 根据 productUUID,从 Cassandra 中查询对应的 Product 实例
 *
 * @ param productUUID
 * Product ColumnFamily 的 key
 * @ return Product 实例
 * @ throws Exception
 */
public Product getProduct(UUID productUUID) throws Exception;

/**
 * 根据 category,从 Cassandra 中查询最新的 topNum 个 Product 实例
 *
 * @ param category
 * 产品的分类
 * @ param topNum
 * 查询最新的实例的个数
 * @ return 最新的 topNum 个 Product 实例
 * @ throws Exception
 */
public List<Product> getTopCategoryProducts(String category,int topNum)
            throws Exception;

/**
 * 根据 productUUID,从 Cassandra 中查询最新的 topNum 个 Product 评论实例
 *
 * @ param productUUID
 * Product ColumnFamily 的 key
 * @ param topNum
 * 查询最新的实例的个数
 * @ return 最新的 topNum 个 Product 评论实例
 * @ throws Exception
 */
public List<Comment> getTopProductComments(UUID productUUID,int topNum)
            throws Exception;
```

BuyerDaoImpl 的构造函数实现如下：

```
public BuyerDaoImpl(String host,int port) {
    TSocket socket = new TSocket(host,port);

    TBinaryProtocol binaryProtocol = new TBinaryProtocol(socket,false,
            false);
    cassandraClient = new Client(binaryProtocol);

    try {
        socket.open();
    } catch (Exception e) {
        e.printStackTrace();
        cassandraClient = null;
```

```
            return;
        }
    }
```

在构造函数中，通过传入的Cassandra主机地址和端口号，获得与Cassandra的连接。再用其他的数据操作方法便可使用这个连接进行操作了。

BuyerDaoImpl的insert方法实现的逻辑为：将用户传入的对象转换为相应的ColumnFamily，然后通过在构造函数中获得的Cassandra连接将数据发送给Cassandra即可。

将用户传入的对象转换为相应的ColumnFamily部分代码如下：

```
if (buyer.getAddress() != null && !buyer.getAddress().isEmpty()) {
    Column c = new Column();
    c.name = "address".getBytes("utf-8");
    c.value = buyer.getAddress().getBytes("utf-8");
    c.timestamp = System.currentTimeMillis();

    ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
    cosc.column = c;
    Mutation mutation = new Mutation();
    mutation.column_or_supercolumn = cosc;
    mutationList.add(mutation);
}
```

上面的代码中，将Buyer的address属性转换为了相应的Column。当所有需要保存的属性都转换为相应的Column后，就可以使用如下的逻辑将数据发送给Cassandra了：

```
cassandraClient.batch_mutate (Buyer.keySpace, mutationMap, ConsistencyLevel.
ONE);
```

BuyerDaoImpl的get方法为insert方法的逆向实现，逻辑为：从Cassandra查询获取相应的ColumnFamily，然后再构造一个Buyer对象返回即可。

从Cassandra中获取相应的ColumnFamily实现如下：

```
List<ColumnOrSuperColumn> buyerColumns =
  cassandraClient.get_slice(Buyer.keySpace,
                            buyerUserName,
                            columnParent,
                            slicePredicate,
                            ConsistencyLevel.ONE);
```

将ColumnFamily构造为Buyer对象的实现如下：

```
Buyer buyer = new Buyer();
buyer.setUserName(buyerUserName);
for (ColumnOrSuperColumn cosc:buyerColumns) {
    Column c = cosc.column;
    String columnName = new String(c.name,"utf-8");
```

```
        String columnValue = new String(c.value,"utf-8");
        if ("age".equals(columnName)) {
            buyer.setAge(Integer.parseInt(columnValue));
        } else if ("address".equals(columnName)) {
            buyer.setAddress(columnValue);
        } else if ("sex".equals(columnName)) {
            buyer.setSex(columnValue);
        } else if ("name".equals(columnName)) {
            buyer.setName(columnValue);
        }
    }
```

SellerDaoImpl 和 ProductDaoImpl 的实现与 BuyerDaoImpl 类似，基本就是实体对象与 Cassandra 中对应的 ColumnFamily 之间的互相转换。

4.4 系统功能验证

4.4.1 BuyerDao 功能验证

首先，输入 Cassandra 的主机名和端口号，获取 BuyerDao 的实例。

```
BuyerDao buyerDao = new BuyerDaoImpl("localhost",9160);
```

构造两个 Buyer 对象，并写入系统中。

```
Buyer buyer1 = new Buyer();
buyer1.setUserName("aaron");
buyer1.setName("郭鹏");
buyer1.setAddress("China Hangzhou");
buyer1.setAge(26);
buyer1.setSex("male");

Buyer buyer2 = new Buyer();
buyer2.setUserName("lily");
buyer2.setName("李娜");
buyer2.setAddress("China 大连");
buyer2.setAge(23);
buyer2.setSex("female");

try {
    buyerDao.insertBuyer(buyer1);
    buyerDao.insertBuyer(buyer2);
} catch (Exception e) {
    e.printStackTrace();
}
```

Buyer 对象写入成功后，再从系统中将 buyerUserName 为 aaron 的 Buyer 对象从系统中读取出来，并将获取到的实例打印在标准输出中。

```
try {
    Buyer aaron = buyerDao.getBuyer("aaron");
    System.out.println(getPropertyString(aaron));
} catch (Exception e) {
    e.printStackTrace();
}
```

可以在标准输出中看到如下结果：

```
userName:aaron
name:郭鹏
age:26
sex:male
address:China Hangzhou
```

4.4.2　SellerDao 功能验证

SellerDao 的使用与 BuyerDao 类似，创建 SellerDao 后并写入两条记录。

```
SellerDao sellerDao = new SellerDaoImpl("localhost",9160);

Seller seller1 = new Seller();
seller1.setUserName("sportSeller");
seller1.setName("体育用品专卖");
seller1.setAddress("China Wuhan");
seller1.setAge(30);
seller1.setSex("male");

Seller seller2 = new Seller();
seller2.setUserName("seller2");
seller2.setName("美丽服装");
seller2.setAddress("China Beijing");
seller2.setAge(37);
seller2.setSex("male");

try {
    sellerDao.insertSeller(seller1);
    sellerDao.insertSeller(seller2);
} catch (Exception e) {
    e.printStackTrace();
}
```

从系统中将 sellerUserName 为 sportSeller 的 Seller 对象从系统中读取出来，并将获取到的实例打印在标准输出中。

```
try {
    Seller sportSeller = sellerDao.getSeller("sportSeller");
```

```
    System.out.println(getPropertyString(sportSeller));
} catch (Exception e) {
    e.printStackTrace();
}
```

可以在标准输出中看到如下结果：

```
userName:sportSeller
name:体育用品专卖
age:30
sex:male
address:China Wuhan
```

4.4.3 ProductDao 功能验证

创建 ProductDao 后并写入两条记录。

```
ProductDao productDao = new ProductDaoImpl("localhost",9160);

Product product1 = new Product();
product1.setDesc("白色真皮足球");
product1.setSellerUserName("sportSeller");
product1.setName("Football");
product1.setPrice(98.8);
product1.setUuid(UUIDGenerator.getInstance().generateTimeBasedUUID());

Product product2 = new Product();
product2.setDesc("橡胶篮球");
product2.setSellerUserName("sportSeller");
product2.setName("Basketball");
product2.setPrice(29.8);
product2.setUuid(UUIDGenerator.getInstance().generateTimeBasedUUID());

try {
    productDao.insertProduct(product1);
    productDao.insertProduct(product2);
} catch (Exception e) {
    e.printStackTrace();
}
```

从系统中将 uuid 值为 product1 的 Product 对象从系统中读取出来，并将获取到的实例打印在标准输出中。

```
try {
    Product football = productDao.getProduct(product1.getUuid());
    System.out.println(getPropertyString(football));
} catch (Exception e) {
```

```
    e.printStackTrace();
}
```

可以在标准输出中看到如下结果：

```
uuid:5ca19e71-bb69-11df-aae5-2950f735babb
name:Football
sellerUserName:sportSeller
desc:白色真皮足球
price:98.8
```

然后将 sportSeller 的两款产品写入 Sports 分类中。

```
try {
    productDao.insertProductCategory("Sports",product1.getUuid());
    productDao.insertProductCategory("Sports",product2.getUuid());
} catch (Exception e) {
    e.printStackTrace();
}
```

再从系统中将 Sports 分类中的产品读取回来，并将获取到的产品信息按照时间先后顺序打印在标准输出中。

```
try {
    List<Product> products=productDao.getTopCategoryProducts(
            "Sports",2);
    for (Product product:products) {

    System.out.println(getPropertyString(product));
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

可以在标准输出中看到如下结果：

```
uuid:fdf76d71-bb6b-11df-b865-0fc6f2f80a40
name:Basketball
sellerUserName:sportSeller
desc:橡胶篮球
price:29.8

uuid:fdf74660-bb6b-11df-b865-0fc6f2f80a40
name:Football
sellerUserName:sportSeller
desc:白色真皮足球
price:98.8
```

最后，为 Basketball 写入两条评论。

```
Comment comment1 = new Comment();
comment1.setUuid(UUIDGenerator.getInstance().generateTimeBasedUUID());
comment1.setCommentUserName("aaron");
comment1.setContent("good product");

Comment comment2 = new Comment();
comment2.setUuid(UUIDGenerator.getInstance().generateTimeBasedUUID());
comment2.setCommentUserName("aaron");
comment2.setContent("送货速度快");

Comment comment3 = new Comment();
comment3.setUuid(UUIDGenerator.getInstance().generateTimeBasedUUID());
comment3.setCommentUserName("lily");
comment3.setContent("弟弟很喜欢这个礼物");

try {
    productDao.insertComment(product2.getUuid(),comment1);
    productDao.insertComment(product2.getUuid(),comment2);
    productDao.insertComment(product2.getUuid(),comment3);
} catch (Exception e) {
    e.printStackTrace();
}
```

从系统中将 basketball 产品的评论读取回来，然后将评论信息按照时间先后顺序排序，最后将排序后的产品信息打印在标准输出中。

```
try {
    List<Comment> comments = productDao.getTopProductComments(
            product2.getUuid(),2);
    for (Comment comment:comments) {

    System.out.println(getPropertyString(comment));
    }
} catch (Exception e) {
    e.printStackTrace();
}
```

可以在标准输出中看到如下结果：

```
uuid:e76ed0be-bb6e-11df-9d98-4f9a55d0dff8
content:弟弟很喜欢这个礼物
commentUserName:lily

uuid:e76ed0bd-bb6e-11df-9d98-4f9a55d0dff8
content:送货速度快
commentUserName:aaron
```

基于 Cassandra 0.6.x 实现的完整代码可以参考附录 B。

4.5　迁移到 Cassandra 0.7.x

本节将讲解如何将基于 Cassandra 0.6.x 实现的在线交易系统迁移到 Cassandra 0.7.x 中。

4.5.1　建立 Eclipse 项目

建立一个名为 CassSeller-0.7 的 Java 项目，将下载的 Cassnadra 发行版中 lib 目录下的需要使用的 jar 包添加到项目依赖包中。由于 Product 和 Comment 实体对象都使用了 TimeUUID，所以需要到 http://jug.safehaus.org/Download 额外下载一个开源 TimeUUID 实现 jar 包——jug-asl-2.0.0.jar，并加入到项目的依赖包中。

依赖包的结构如图 4-3 所示：

然后将 CassSeller 中的代码原封不动地复制到 CassSeller-0.7 中，由于 Cassandra 0.7.x 中部分编程接口有变动，所以代码中会包含错误信息，如图 4-4 所示。

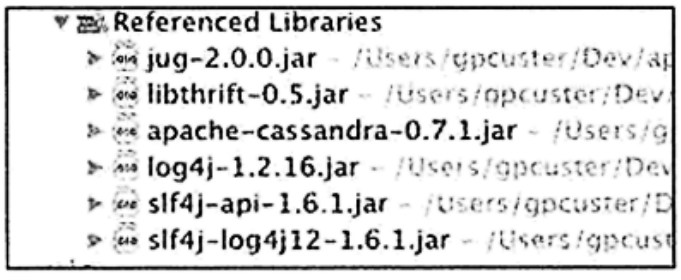


图 4-3　CassSeller-0.7 依赖包结构

图 4-4　CassSeller-0.7 项目结构

4.5.2　修改编译错误代码

完成项目的建立后，就需要修改代码，使用在 Cassandra 0.7.x 中变动的接口。

1. Column

在 Cassandra 0.7.x 中，我们定义的 Column 结构体中各个变量的类型发生了变化，如图 4-5 所示的错误代码。

在 Cassandra 0.6.x 中，Column 中的 name 和 value 属性的类型都是 byte[]。但是在 Cassandra 0.7.x 中，Column 中的 name 和 value 属性的类型变更为了 ByteBuffer，所以我们只需要将 byte[]类型转换成 ByteBuffer 类型即可，如图 4-6 所示。

```
if (product.getPrice() > 0) {
    Column c = new Column();
    c.name = "price".getBytes("utf-8");
    c.value = String.valueOf(product.getPrice()).getBytes("utf-8");
    c.timestamp = System.currentTimeMillis();
```

图 4-5 Column 错误代码 1

```
if (product.getPrice() > 0) {
    Column c = new Column();
    c.name = ByteBuffer.wrap("price".getBytes("utf-8"));
    c.value = ByteBuffer.wrap(String.valueOf(product.getPrice()).getBytes("utf-8"));
    c.timestamp = System.currentTimeMillis();
```

图 4-6 Column 修正代码 1

另外，在将 Column 的值读取出来的时候也会因为结构体属性的类型发生了变化而遇到编译错误，如图 4-7 所示。

```
comment.setUuid(new UUID(sc.name));
for (Column c : sc.columns) {
    String columnName = new String(c.name, "utf-8");
    String columnValue = new String(c.value, "utf-8");
```

图 4-7 Column 错误代码 2

同理，我们只需要将类型转换正确即可，如图 4-8 所示。

```
Comment comment = new Comment();
comment.setUuid(new UUID(sc.getName()));
for (Column c : sc.columns) {
    String columnName = new String(c.getName(), "utf-8");
    String columnValue = new String(c.getValue(), "utf-8");
```

图 4-8 Column 修正代码 2

2. SuperColumn

在 Cassandra 0.7.x 中，我们定义的 SuperColumn 结构体中各个变量的类型发生了变化，如图 4-9 所示的错误代码。

```
if (!columns.isEmpty()) {
    SuperColumn sc = new SuperColumn();
    sc.name = comment.getUuid().toByteArray();
    sc.columns = columns;
```

图 4-9 SuperColumn 错误代码

在 Cassandra 0.6.x 中，SuperColumn 中的 name 属性的类型是 byte[ ]。但是在 Cassandra 0.7.x 中，SuperColumn 中的 name 属性类型变更为了 ByteBuffer，所以我们只需要将 byte[ ]类型转换成 ByteBuffer 类型即可，如图 4-10 所示。

```
if (!columns.isEmpty()) {
    SuperColumn sc = new SuperColumn();
    sc.name = ByteBuffer.wrap(comment.getUuid().toByteArray());
    sc.columns = columns;
```

图4-10　SuperColumn修正代码

另外，在将SuperColumn的值读取出来的时候也会因为结构体属性的类型发生了变化而遇到编译错误，只需要将类型转换正确即可。

3. ColumnPath

在Cassandra 0.7.x中，我们定义的ColumnPath结构体中各个变量的类型发生了变化，如图4-11所示的错误代码。

```
ColumnPath columnPath = new ColumnPath();
columnPath.column_family = "ProductCategory";
columnPath.column = productUUID.toByteArray();
```

图4-11　ColumnPath错误代码

在Cassandra 0.6.x中，ColumnPath中的column属性的类型是byte[]。但是在Cassandra 0.7.x中，ColumnPath中的column属性类型变更为了ByteBuffer，所以我们只需要将byte[]类型转换成ByteBuffer类型即可，如图4-12所示。

```
ColumnPath columnPath = new ColumnPath();
columnPath.column_family = "ProductCategory";
columnPath.column = ByteBuffer.wrap(productUUID.toByteArray());
```

图4-12　ColumnPath修正代码

4. SliceRange

在Cassandra 0.7.x中，我们定义的SliceRange结构体中各个变量的类型发生了变化，如图4-13所示的错误代码。

```
SliceRange range = new SliceRange(new byte[] {}, new byte[] {}, true,
        Integer.MAX_VALUE);
```

图4-13　SliceRange错误代码

在Cassandra 0.6.x中，SliceRange中的start和finish属性的类型是byte[]。但是在Cassandra 0.7.x中，SliceRange中的start和finish属性类型变更为了ByteBuffer，所以我们只需要将byte[]类型转换成ByteBuffer类型即可，如图4-14所示。

5. Client

在Cassandra 0.7.x中，我们定义的Client结构体需要先指定一个具体的Keyspace，然后再执行相应的增删改查的操作，并且在这些操作的接口中，无须再次指定Keyspace，同时

将 String 类型的参数转换为 ByteBuffer 类型，如图 4-15 所示的错误代码。

```
SliceRange range = new SliceRange(ByteBuffer.wrap(new byte[] {}),
        ByteBuffer.wrap(new byte[] {}), true, Integer.MAX_VALUE);
```

图 4-14 SliceRange 修正代码

```
cassandraClient.batch_mutate(Buyer.keySpace, mutationMap,
        ConsistencyLevel.ONE);
```

图 4-15 Client 错误代码

图 4-15 中的代码修改为如图 4-16 所示。

```
cassandraClient.set_keyspace(Buyer.keySpace);
List<ColumnOrSuperColumn> buyerColumns = cassandraClient.get_slice(
        ByteBuffer.wrap(buyerUserName.getBytes("utf-8")), columnParent,
        slicePredicate, ConsistencyLevel.ONE);
```

图 4-16 Client 修正代码

另外，在 Cassandra 0.7.x 中，客户端通过 Thrift 与服务器端的通信开始默认使用 TFrame 的形式，所以构造 Client 实例的代码如下：

```
TSocket socket = new TSocket(host,port);
TBinaryProtocol binaryProtocol = new TBinaryProtocol(socket,false,
        false);
cassandraClient = new Client(binaryProtocol);

try {
    socket.open();
} catch (Exception e) {
    e.printStackTrace();
    cassandraClient = null;
    return;
}
```

要修改如下：

```
TSocket socket = new TSocket(host,port);

TTransport transport = new TFramedTransport(socket);

TBinaryProtocol binaryProtocol = new TBinaryProtocol(transport,true,true);
cassandraClient = new Client(binaryProtocol);

try {
```

```
        transport.open();
    } catch (Exception e) {
        e.printStackTrace();
        cassandraClient = null;
        return;
    }
```

4.5.3　新增Schema在线定义功能

在Cassandra 0.7.x中，系统启动的时候并不会从配置文件中加载Schema信息，用户必须通过编程接口自定义相关的Schema信息。

所以，我们在项目中新增加一个类，专门用于在Cassandra中定义该应用需要的Schema信息。类的全名为cassSeller.dao.impl.SchemaIniter，其中包含一个静态方法。

```
public static void init(Client cassandraClient) throws Exception {
    List<CfDef> cfDefs = new ArrayList<CfDef>();

    CfDef buyerCfDef = new CfDef();
    buyerCfDef.keyspace = Buyer.keySpace;
    buyerCfDef.name = Buyer.ColumnFamily;
    buyerCfDef.comparator_type = "UTF8Type";
    cfDefs.add(buyerCfDef);

    CfDef sellerCfDef = new CfDef();
    sellerCfDef.keyspace = Seller.keySpace;
    sellerCfDef.name = Seller.ColumnFamily;
    sellerCfDef.comparator_type = "UTF8Type";
    cfDefs.add(sellerCfDef);

    CfDef productCategoryCfDef = new CfDef();
    …

    KsDef ksDef = new KsDef();
    ksDef.name = Buyer.keySpace;
    ksDef.replication_factor = 1;
    ksDef.strategy_class = "org.apache.cassandra.locator.SimpleStrategy";
    ksDef.cf_defs = cfDefs;

    cassandraClient.system_add_keyspace(ksDef);
}
```

在这个方法中，我们定义了CassSeller程序需要使用的Schema信息，然后将这个变动通知Cassandra，使其生效。

然后，在构造cassSeller.dao.impl.BuyerDaoImpl等实例的时候，增加一个判断：如果系统中没有我们需要的Schema信息，则在Cassandra中创建相应的Schema信息。

```
try {
    cassandraClient.set_keyspace(Buyer.keySpace);
} catch (Exception e) {
    e.printStackTrace();
    try {
        SchemaIniter.init(cassandraClient);
        cassandraClient.set_keyspace(Buyer.keySpace);
    } catch (Exception e1) {
        e1.printStackTrace();
        cassandraClient = null;
    }
}
```

4.5.4 功能验证

在完成了上述功能后，我们就可以开始运行和维护整个程序进行测试了。

首先启动 Cassandra 0.7.x，然后执行 CassSeller 的程序，程序的完整输出如下：

```
userName:aaron
name:郭鹏
age:26
sex:male
address:China Hangzhou
userName:sportSeller
name:体育用品专卖
age:30
sex:male
address:China Wuhan

uuid:1df02fae-44ee-11e0-b74c-330488d86f62
name:Football
sellerUserName:sportSeller
desc:白色真皮足球
price:98.8

uuid:1df02faf-44ee-11e0-b74c-330488d86f62
name:Basketball
sellerUserName:sportSeller
desc:橡胶篮球
price:29.8

uuid:1df02fae-44ee-11e0-b74c-330488d86f62
name:Football
sellerUserName:sportSeller
desc:白色真皮足球
```

```
price:98.8

uuid:1e00f892-44ee-11e0-b74c-330488d86f62
content:弟弟很喜欢这个礼物
commentUserName:lily

uuid:1e00f891-44ee-11e0-b74c-330488d86f62
content:送货速度快
commentUserName:aaron
```

说明程序运行和维护成功了。

基于 Cassandra 0.7.x 实现的完整代码可以参考附录 C。

4.6　本章小结

本章分析、设计和实现了一个基于 Cassandra 的在线交易系统。通过本章的学习，读者可以了解如何分析和设计 Cassandra 的底层数据模型，并了解如何实现相关的应用。

第 5 章

Cassandra 的集群机制

本章内容

- 一致性哈希
- Gossip：集群节点之间的通信协议
- 集群的数据备份机制
- 集群状态变化的处理机制
- 本章小结

Cassandra 中的集群机制与大多数的分布式系统不同。例如，在 Hadoop 系统中，有一台机器有主节点（Master Node）和多台从节点（Slave Node），主节点控制所有的从节点。当 Hadoop 系统中的从节点出现故障时，整个系统的工作不会受到影响；而主节点出现故障后，整个 Hadoop 系统就不能正常使用了。由于 Cassandra 机器中每一台机器都是对等的，不存在主节点和从节点的概念，所以集群中任何一台机器出现故障，整个系统都不会受到影响，依旧可以正常工作。

本章将从基本原理入手，结合源代码讲解 Cassandra 的集群机制。

5.1 一致性哈希

5.1.1 理解一致性哈希

一致性哈希（Consistent Hash）是整个 Cassandra 集群的基础。

要理解什么是一致性哈希，为什么要使用一致性哈希，可以先回到 Cassandra 需要解决的根本问题——提供一个可以增加机器线性扩展性能的集群。

假设现在有 5 台服务器，需要实现一个集群以提供数据的存储与读取的功能。可以思考这样一种简洁的实现，如图 5-1 所示。

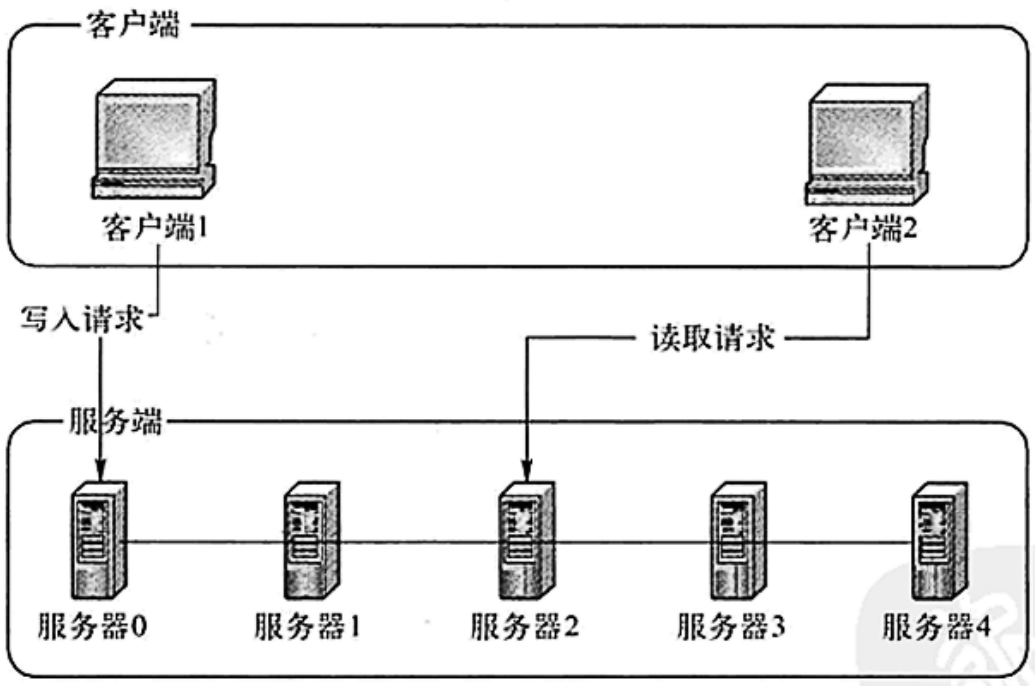


图 5-1 简单集群结构图

在图 5-1 所示的集群中，每一台服务器都有一个固定标志，分别为 0、1、2、3、4，并且每一台服务器都知道其他另外 4 台服务器的固定标志。

服务器端处理客户端的写入请求过程如下：

根据写入请求的 Key 求哈希值，然后用 Key 的哈希值与集群的服务器数量（这里为 5）取模，判断这个写入请求应该写入到集群中的哪台服务器中。如果写入请求应该由本服务器处理，就将数据写入磁盘；如果属于另外一台服务器，则转发写入请求到对应的服务器中。

服务器端处理客户端的写入请求过程的伪代码如下：

```
public void handleWrite(WriteMessage wm) {
    //从写入消息中获取 Key
    String key = wm.getKey();

    //计算实际处理该写入消息的节点编号
    int handleID = key.hashCode() % ClusterNodeNum;

    //判断写入消息是否由本服务器处理
    if (MyClusterID == handleID) {
        //该写入消息由本服务器处理,写入磁盘
        wirteMessage(wm);
    } else {
        //将写入消息发送给对应的服务器处理
        sendMessage(handleID, wm);
    }

    //通知客户端写入成功
    send2Client("write success");
}
```

服务器端处理客户端的读取请求过程如下：

根据读取请求的 Key 求哈希值，然后用 Key 的哈希值与集群的服务器数量（这里为 5）取模，判断这个读取请求应该从集群中的哪台服务器中读取。如果读取请求应该由本服务器处理，就从磁盘中读取数据；如果属于另外一台服务器，则转发读取请求到对应的服务器中。

服务器端处理客户端的读取请求过程的伪代码如下：

```
public void handleRead(ReadMessage rm) {
    //从读取消息中获取 Key
    String key = rm.getKey();

    //计算实际处理该读取消息的节点编号
    int handleID = key.hashCode() % ClusterNodeNum;

    //用于存储读取的数据
    String readData = "";

    //判断读取消息是否由本服务器处理
    if (MyClusterID == handleID) {
        //从磁盘中读取数据
        readData = readMessage(rm);
    } else {
        //将读取消息发送给对应的服务器处理
```

```
            readData = sendMessage(handleID, rm);
        }

        //将读取到的数据返回给客户端
        send2Client(readData);
    }
```

最后，客户端调用服务器端的伪代码如下：

```
public static void main(String[] args) {
    Random rnd = new Random();

    //所有可以使用的服务器端地址
    String[] servers = new String[] {"10.192.168.0", "10.192.168.1",
       "10.192.168.2", "10.192.168.3", "10.192.168.4"};

    //由于所有的服务器都可以提供相同的服务,为了做到负载均衡,随机从可用的服务器中选择一个
    Client client = new
    Client(servers[rnd.nextInt(servers.length)]);

    //向服务器端写入数据
    client.write(new WriteMessage("key1", "value1"));

    //从服务器端读取数据
    String value = client.read(new ReadMessage("key1"));
}
```

通过上面的设计，实现了一个简单的存储集群。

这种架构实现非常简单，同时可以利用多台机器共同提供服务。它的核心就在于对数据的 Key 进行哈希取模。

但是这种架构具有一个严重的缺陷：添加服务器或减少服务器困难。一旦集群中的机器发生变动，集群中的所有服务器都需要对现有的数据进行重新分布。

针对集群中的机器发生变动的情况，一致性哈希可以极大地降低数据重分布的影响。

那么一致性哈希是如何工作的呢?

首先求出集群中每一个节点的哈希值，并将其配置到 0 ～ 2^{32} 的圆环上。用同样的方法求出存储数据的 Key 的哈希值，并映射到圆上。然后从数据映射到的位置开始顺时针查找，将数据保存到找到的第一个服务器上。如果超过 2^{32} 仍然找不到相应服务器，就会保存到第一台服务器上，如图 5-2 所示。

从图 5-2 所示的状态中添加一台新的服务器，只有在圆环上增加服务器的地点逆时针方向的第一台服务器上的 Key 会受到影响，如图 5-3 所示。

因此，一致性哈希最大限度地抑制了 Key 的重新分布。而且，有的一致性哈希的实现方法还采用了虚拟节点的思想。使用一般的哈希函数的服务器的映射地点分布非常不均匀，而使用虚拟节点的思想（比如 Amazon 公司的一个分布式 Key/Value 存储引擎 Dynamo），为

每个物理节点（服务器）在圆环上分配 100 ～ 200 个点，这样就能抑制分布不均匀的现象，最大限度地减小服务器增减时的缓存重新分布。

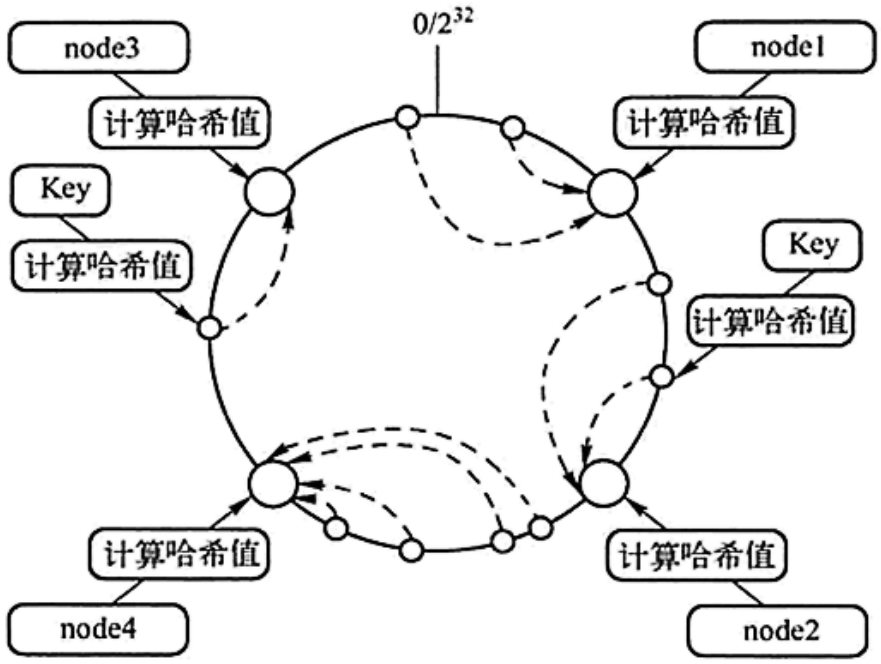


图 5-2　一致性哈希示意图

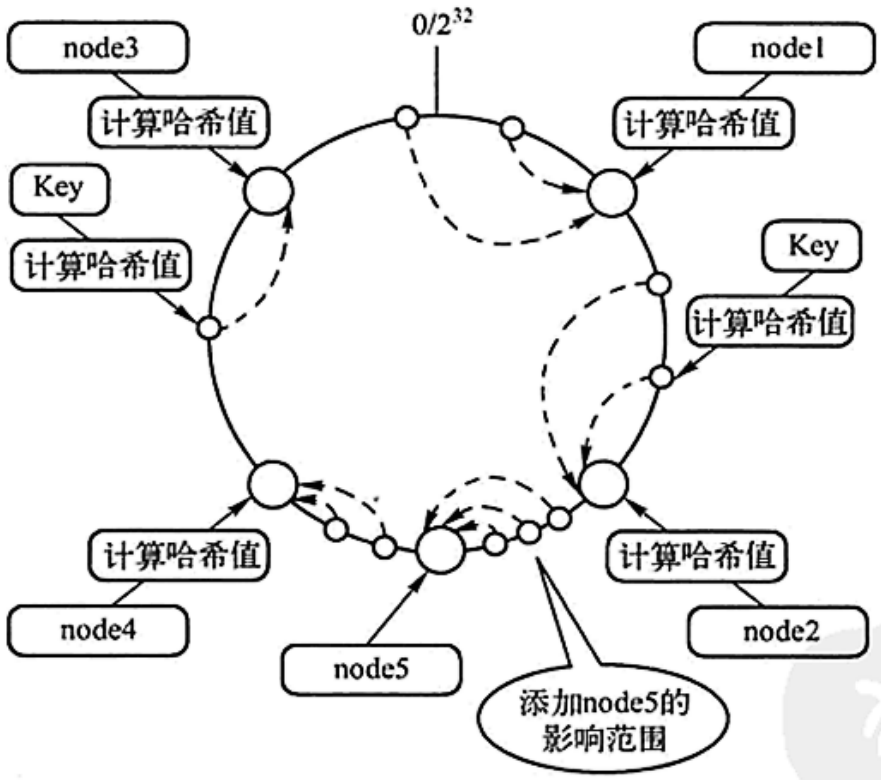


图 5-3　添加新的服务器

5.1.2　一致性哈希在 Cassandra 中的应用

了解了一致性哈希的原理，可以通过源代码了解 Cassandra 中是如何应用该原理的。按照附录 A 中的介绍，可以在 Eclipse 中编译和浏览 Cassandra 的所有源代码，如图 5-4 所示。

在 org. apache. cassandra. dht 包中，包含了分布式哈希表（Distribute Hash Table）的主要逻辑：Token、Range 和 Partitioner。

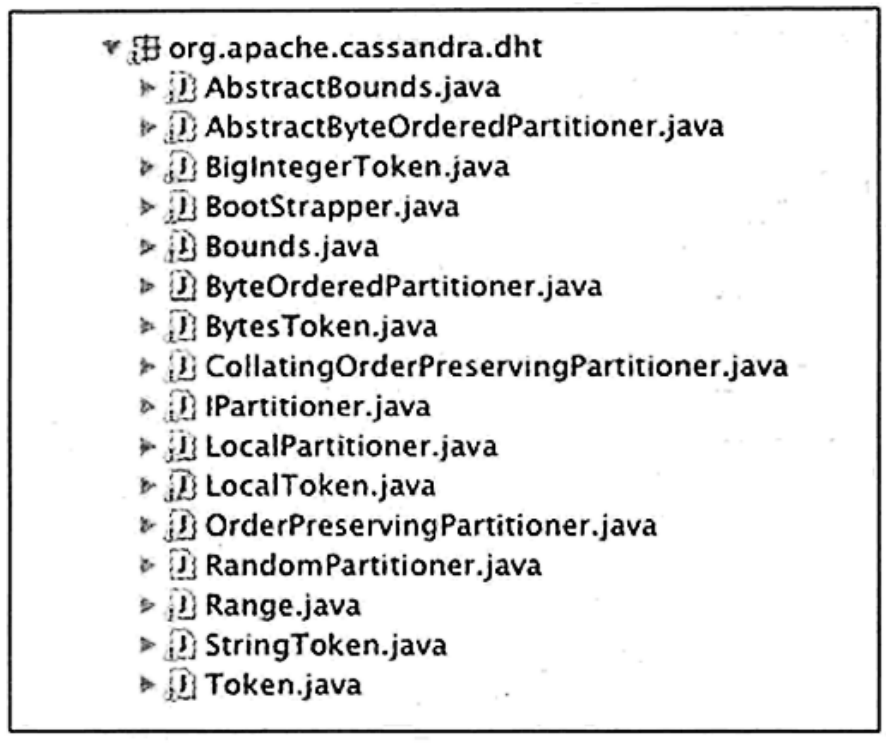


图 5-4 org. apache. cassandra. dht 代码结构

1. Token

在 Cassandra 中，每一个节点都对应一个唯一的 Token，相当于一致性哈希圆环中的一个节点地址。

org. apache. cassandra. dht. Token < T > 是一个抽象类，Cassandra 提供了以下 4 种实现：

1）BigIntegerToken：以 java. math. BigInteger 类型的实例代表一个 Token，由 org. apache. cassandra. dht. RandomPartitioner 使用。

2）BytesToken：以 byte[]类型的实例代表一个 Token，由 org. apache. cassandra. dht. ByteOrderedPartitioner 和 org. apache. cassandra. dht. CollatingOrderPreservingPartitioner 使用。

3）LocalToken：以 byte[]类型的实例和一个 org. apache. cassandra. db. marshal. AbstractType 类型的实例代表一个 Token，由 org. apache. cassandra. dht. LocalPartitioner 使用。

4）StringToken：以 java. lang. String 类型的实例代表一个 Token，由 org. apache. cassandra. dht. OrderPreservingPartitioner 使用。

2. Range

在 Cassandra 中，每一个节点负责处理一致性哈希圆环中的一段数据，从逆时针方向的上一个节点对应的 Token（开区间）到本节点对应的 Token（闭区间）范围内。这个范围就是 Range。

Range 包含如下两个重要的属性：

```
public final Token left;
public final Token right;
```

其中，right 代表一致性哈希圆环中本节点对应的 Token，left 代表一致性哈希圆环中逆时针方向的上一个节点对应的 Token，即(left,right]。

同时，Range还提供了一系列和圆环相关的操作方法：

1）public static boolean contains(Token left, Token right, Token bi)：判断bi是否在由left和right构造的Range之中。

2）public boolean contains(Range that)：判断that是否被该Range所包含。

3）public boolean intersects(Range that)：判断两个Range是否相交。

4）public Set < Range > intersectionWith(Range that)：判断两个Range相交的Range集合。

3. Partitioner

Partitioner用于管理Token在一致性哈希圆环中的生成规则，并且决定每一台机器中SSTable数据的排序规则。每一个Cassandra实例需要并且只能指定一个Partitioner实现。配置文件中默认的配置为partitioner：org. apache. cassandra. dht. RandomPartitioner。

org. apache. cassandra. dht. IPartitioner < T > 接口提供的方法功能如下：

1）public DecoratedKey < T > decorateKey(byte[] key)：将key转换为DecoratedKey（Token和key的组合）。这个方法在Cassandra对数据进行写入和读取操作时使用，用于数据key之前的排序判断。其中的排序规则是由对应的Token决定的。

2）public T midpoint(T left, T right)：计算left和right两个Token的中间值。这个方法在新节点加入Cassandra节点时调用。

3）public T getMinimumToken()：获取一致性哈希圆环中最小的Token。

4）public T getToken(byte[] key)：根据key计算对应的Token。

5）public T getRandomToken()：获取一个随机的Token。在集群初始化的时候，如果没有显式指定节点的Token值，将调用这个方法为每一个节点随机生成Token。

Cassandra提供了如下5种IPartitioner的实现：

（1）org. apache. cassandra. dht. RandomPartitioner

根据key生成Token的算法为md5，逻辑如下：

```
public BigIntegerToken getToken(byte[] key)
{
    if (key.length == 0)
        return MINIMUM;
    return new BigIntegerToken(FBUtilities.md5hash(key));
}
```

Token的排序规则与java. math. BigInteger的排序规则相同。

如果在配置文件中指定了InitialToken，读取的逻辑如下：

```
public Token<BigInteger> fromString(String string)
{
    return new BigIntegerToken(new BigInteger(string));
}
```

（2）org. apache. cassandra. dht. OrderPreservingPartitioner

根据 key 生成 Token 的算法为将 key 直接转换为 UTF-8 编码的字符串，逻辑如下：

```
public StringToken getToken(byte[] key)
{
    String skey;
    try
    {
        skey = FBUtilities.decodeToUTF8(key);
    }
    catch (CharacterCodingException e)
    {
        throw new RuntimeException("The provided key was not UTF8 encoded.", e);
    }
    return new StringToken(skey);
}
```

Token 的排序规则与 java. lang. String 的排序规则相同。

如果在配置文件中指定了 InitialToken，读取的逻辑如下：

```
public Token<BigInteger> fromString(String string)
{
    return new StringToken(string);
}
```

（3）org. apache. cassandra. dht. LocalPartitioner

根据 key 生成 Token 的算法为将 key 直接转换为 LocalToken，逻辑如下：

```
public LocalToken getToken(byte[] key)
{
    return new LocalToken(comparator, key);
}
```

Token 的排序规则与 org. apache. cassandra. db. marshal. AbstractType 的排序规则相同。org. apache. cassandra. db. marshal. AbstractType 的各个实现类的排序规则请参考表 2-1。

如果在配置文件中指定了 InitialToken，读取将出错，LocalPartitioner 不支持手动指定 InitialToken。

（4）org. apache. cassandra. dht. ByteOrderedPartitioner

根据 key 生成 Token 的算法为将 key 直接转换为 BytesToken，逻辑如下：

```
public BytesToken getToken(byte[] key)
{
    if (key.length==0)
        return MINIMUM;
    return new BytesToken(key);
}
```

Token 的排序规则与 byte 的排序规则相同。

如果在配置文件中指定了 InitialToken，读取的逻辑如下：

```
public Token<BigInteger> fromString(String string)
{
    return new BytesToken(FBUtilities.hexToBytes(string));
}
```

(5) org. apache. cassandra. dht. CollatingOrderPreservingPartitioner

根据 key 生成 Token 的算法为将 key 直接转换为 UTF-8 编码的字符串，逻辑如下：

```
public BytesToken getToken(byte[] key)
{
    if (key.length==0)
        return MINIMUM;

    String skey;
    try
    {
        skey = FBUtilities.decodeToUTF8(key);
    }
    catch (CharacterCodingException e)
    {
        throw new RuntimeException("The provided key was not UTF8 encoded.", e);
    }
    return new
BytesToken(collator.getCollationKey(skey).toByteArray());
}
```

Token 的排序规则与 byte 的排序规则相同。

如果在配置文件中指定了 InitialToken，读取的逻辑如下：

```
public Token<BigInteger> fromString(String string)
{
    return new BytesToken(FBUtilities.hexToBytes(string));
}
```

以上 5 种 Partitioner 的主要区别就在于将 Key 转换为 Token 的算法与底层数据存储中 key 的排序顺序。如果不想使用 Cassandra 随机分配的 Token，可以按照各种 Partitioner 从配置文件中读取 Token 的规则指定相应的 InitialToken。

5.2　Gossip：集群节点之间的通信协议

在 Cassandra 中实现集群中节点通信的代码在 org. apache. cassandra. gms 中，如图 5-5 所示。

图 5-5 所示的代码结构主要包括两个部分的实现：FailureDetector（失效节点的检验）

与 Gossiper（节点之间的状态传递）。

图 5-5 org. apache. cassandra. gms 代码结构

5.2.1 FailureDetector

Cassandra 中的 FailureDetector 原理基于 Hayashibara 的一篇论文 "*The Phi Accrual Failure Detector*"。简单地说：FailureDetector 会记录集群中其他节点与本节点的通信历史，然后根据当前时间和某一个节点的通信历史，判断某一个节点是否还存活。

FailureDetector 实现了 org. apache. cassandra. gms. IFailureDetector 接口，它的主要方法与相关作用如下。

1）public boolean isAlive(InetAddress ep)：判断集群中的某个节点是否存活。

2）public void interpret(InetAddress ep)：根据当前时间和该节点的通信历史，判断该节点是否还存活，如果该节点失效，则触发 org. apache. cassandra. gms. IFailureDetectionEventListener 的 convict 事件，即通知 Gossiper 从集群中将该节点设置为已经失效状态。该方法在 Gossiper 中每隔 1 s 对集群中的每一个节点进行一次判断。

3）public void report(InetAddress ep)：获取某一个节点的通信信息，记录到该节点的通信历史中。该方法在 Gossiper 每接收到集群中另一个节点的通信消息（GossipDigestSynMessage）时调用。

4）public void registerFailureDetectionEventListener(IFailureDetectionEventListener listener)：在 FailureDetector 中注册 IFailureDetectionEventListener，这个 IFailureDetectionEventListener 就是 Gossiper，即在 FailureDetector 发现某个节点失效时，自动调用 Gossiper 的 convict 方法，

将该节点的状态设置为失效。

5.2.2　Gossiper

Cassandra 集群没有中心节点，各个节点的地位完全相同，节点之间通过一种叫做 Gossip 的协议进行通信，用于维护集群的状态。通过 Gossip，每个节点都能知道集群中包含哪些节点，以及每一个节点的状态。这使得 Cassandra 集群中的任何一个节点都可以完成任意读取和写入操作[⊖]，若任意一个节点失效，整个集群依旧正常工作。

在 Gossip 初始化的时候，将构造 4 个集合，分别保存集群中存活的节点（liveEndpoints_）、失效的节点（unreachableEndpoints_）、种子节点（seeds_）和各个节点信息（endpointStateMap_）。

Cassandra 启动时会从配置文件中加载 seeds 的信息（这个配置项中指定了集群中的原始节点地址）到 seeds_中，然后启动一个 GossipTask 定时任务，每隔 1 s 钟执行一次，其实现逻辑见代码清单 5-1。

代码清单 5-1　GossipTask 实现逻辑

```
private class GossipTask implements Runnable
{
    public void run()
    {
        try
        {
            endpointStateMap_.get(localEndpoint_).getHeartBeatState().update-
HeartBeat();

            List<GossipDigest> gDigests = new ArrayList<GossipDigest>();

            Gossiper.instance.makeRandomGossipDigest(gDigests);

            if ( gDigests.size() > 0 )
            {
                Message message = makeGossipDigestSynMessage(gDigests);

                boolean gossipedToSeed = doGossipToLiveMember(message);

                doGossipToUnreachableMember(message);

                if (!gossipedToSeed || liveEndpoints_.size() < seeds_.size())
                    doGossipToSeed(message);

                doStatusCheck();
```

⊖　如果操作不属于本节点负责，则转发给合适的节点；如果属于本节点负责，则直接处理。

```
            }
        }
        catch (Exception e)
        {
            logger_.error("Gossip error", e);
        }
    }
}
```

首先更新本节点的心跳版本号，然后构造需要发送给其他节点的 GossipDigestSynMessage 消息，再将 GossipDigestSynMessage 消息发送给合适的节点，最后通过调用 FailureDetector 的 interpret 方法检查集群中是否有失效的节点。

GossipTask 中选择发送 GossipDigestSynMessage 消息给集群中那些节点的逻辑如下：

从集群存活的节点（liveEndpoints）中随机选择一个节点发送 GossipDigestSynMessage；然后根据一定的概率，从失效的节点（unreachableEndpoints_）中随机选取一个节点发送 GossipDigestSynMessage；最后，如果之前发送 GossipDigestSynMessage 消息的节点中不包含 seed 节点，或者当前活着的节点数少于 seed 节点数，则随机向一个 seed 发送 GossipDigestSynMessage 消息。

1. GossipDigestSynMessage

GossipDigestSynMessage 消息作为 Gossip 通信中的第一步，包含所有节点的地址、心跳版本号与节点状态版本号。接收到 GossipDigestSynMessage 消息的节点将执行以下操作：

1）根据接收到的 GossipDigest 集合，调用 FailureDetector 的 report 方法更新集群中节点的状态。

2）对接收到的 GossipDigestSynMessage 消息中的 GossipDigest 集合进行排序。

3）对比接收到的 GossipDigest 信息与本节点的 GossipDigest 差异，本节点需要进一步获取的节点信息由 deltaGossipDigestList 保存，本节点需要告诉发送 GossipDigest 信息节点的信息由 deltaEpStateMap 保存。

4）利用 deltaGossipDigestList 和 deltaEpStateMap 构建 GossipDigestAckMessage 消息，并将其发送给发送 GossipDigestSynMessage 消息的节点。

整个实现过程在代码 GossipDigestSynVerbHandler. java 中，如代码清单 5-2 所示。

代码清单 5-2　GossipDigestSynMessage 消息的处理逻辑

```
public void doVerb(Message message)
{
    InetAddress from = message.getFrom();
    if (logger_.isTraceEnabled())
        logger_.trace("Received a GossipDigestSynMessage from {}", from);

    byte[] bytes = message.getMessageBody();
```

```
    DataInputStream dis = new DataInputStream( new ByteArrayInputStream(bytes) );
    try
    {
        GossipDigestSynMessage gDigestMessage = GossipDigestSynMessage.serial-
izer().deserialize(dis);

        if
( !gDigestMessage.clusterId_.equals(DatabaseDescriptor.getCluster-Name()) )
        {
            logger_.warn ("ClusterName mismatch from " + from + " " + gDigestMes-
sage.clusterId_ + "!= " + DatabaseDescriptor.getClusterName());
            return;
        }

        List <GossipDigest > gDigestList = gDigestMessage.getGossipDigests();
        /* Notify the Failure Detector */

        Gossiper.instance.notifyFailureDetector(gDigestList);

        doSort(gDigestList);

        List <GossipDigest > deltaGossipDigestList = new ArrayList <GossipDigest > ();
        Map < InetAddress, EndpointState > deltaEpStateMap = new HashMap < InetAd-
dress, EndpointState > ();
        Gossiper.instance.examineGossiper(gDigestList, deltaGossipDigestList,
deltaEpStateMap);

        GossipDigestAckMessage gDigestAck = new GossipDigestAckMessage(deltaGos-
sipDigestList, deltaEpStateMap);
        Message gDigestAckMessage = Gossiper.instance.makeGossipDigestAckMess-
age(gDigestAck);
        if (logger_.isTraceEnabled())
            logger_.trace("Sending a GossipDigestAckMessage to {}", from);

    MessagingService.instance.sendOneWay(gDigestAckMessage, from);
    }
    catch (IOException e)
    {
        throw new RuntimeException(e);
    }
}
```

2. GossipDigestAckMessage

GossipDigestAckMessage消息是Gossip通信中的第二步，接收到GossipDigestAckMessage

消息的节点将执行以下操作：

1）在本地更新 GossipDigestAckMessage 消息中包含需要本节点更新的节点信息并调用 FailureDetector 的 report 方法更新集群中节点的状态。

2）将发送 GossipDigestAckMessage 消息的节点需要的其他节点的信息构造成 GossipDigestAck2Message 消息。

3）将 GossipDigestAck2Message 消息发送给发送 GossipDigestAckMessage 消息的节点。

整个实现过程在代码 GossipDigestAckVerbHandler. java 中，如代码清单 5-3 所示。

代码清单 5-3　GossipDigestAckMessage 消息的处理逻辑

```
public void doVerb(Message message)
{
    InetAddress from = message.getFrom();
    if (logger_.isTraceEnabled())
        logger_.trace("Received a GossipDigestAckMessage from {}", from);

    byte[] bytes = message.getMessageBody();
    DataInputStream dis = new DataInputStream( new
        ByteArrayInputStream(bytes) );

    try
    {
        GossipDigestAckMessage gDigestAckMessage =
GossipDigestAckMessage.serializer().deserialize(dis);
        List < GossipDigest > gDigestList =
gDigestAckMessage.getGossipDigestList();
        Map < InetAddress, EndpointState > epStateMap =
gDigestAckMessage.getEndpointStateMap();

        if ( epStateMap.size() > 0 )
        {
            /* Notify the Failure Detector */

            Gossiper.instance.notifyFailureDetector(epStateMap);
            Gossiper.instance.applyStateLocally(epStateMap);
        }

        Map < InetAddress, EndpointState > deltaEpStateMap =
new HashMap < InetAddress, EndpointState > ();
        for( GossipDigest gDigest : gDigestList )
        {
            InetAddress addr = gDigest.getEndpoint();
            EndpointState localEpStatePtr = Gossiper.instance.getStateFor-
VersionBiggerThan(addr, gDigest.getMaxVersion());
            if ( localEpStatePtr != null )
                deltaEpStateMap.put(addr, localEpStatePtr);
```

```
        }

        GossipDigestAck2Message gDigestAck2 = new GossipDigestAck2Message-
(deltaEpStateMap);
        Message gDigestAck2Message = Gossiper.instance.makeGossipDigestAck2-
Message(gDigestAck2);
        if (logger_.isTraceEnabled())
            logger_.trace("Sending a GossipDigestAck2Message to {}", from);

MessagingService.instance.sendOneWay(gDigestAck2Message, from);
    }
    catch ( IOException e )
    {
        throw new RuntimeException(e);
    }
}
```

3. GossipDigestAck2Message

GossipDigestAck2Message 消息是 Gossip 通信中的第三步也是最后一步，接收到 GossipDigestAck2Message 消息的节点将执行以下操作：在本地更新 GossipDigestAck2Message 消息中包含需要本节点更新的节点信息并调用 FailureDetector 的 report 方法更新集群中节点的状态。

整个实现过程在代码 GossipDigestAck2VerbHandler. java 中，如代码清单 5-4 所示。

代码清单 5-4　GossipDigestAck2Message 消息的处理逻辑

```
public void doVerb(Message message)
{
    InetAddress from = message.getFrom();
    if (logger_.isTraceEnabled())
        logger_.trace("Received a GossipDigestAck2Message from {}", from);

    byte[] bytes = message.getMessageBody();
    DataInputStream dis = new DataInputStream( new ByteArrayInputStream(bytes) );
    GossipDigestAck2Message gDigestAck2Message;
    try
    {
        gDigestAck2Message = GossipDigestAck2Message.serializer().deserialize
(dis);
    }
    catch (IOException e)
    {
        throw new RuntimeException(e);
    }
    Map < InetAddress, EndpointState > remoteEpStateMap =
gDigestAck2Message.getEndpointStateMap();
```

```
        Gossiper.instance.notifyFailureDetector(remoteEpStateMap);
        Gossiper.instance.applyStateLocally(remoteEpStateMap);
    }
```

通过上面的流程，Gossip 就完成了节点与节点之间信息的交换。简单来说，就是比较集群中不同节点之间的数据差异，然后将不一致的数据统一更新到最新的状态。这样就保证了集群中的每一个节点都了解集群中其他各个节点的状态。

5.3 集群的数据备份机制

Cassandra 是一个支持容灾的系统，即数据会在集群中保留多份，这样当某一个机器失效的时候，其他机器仍然有数据备份，从而保证整个服务正常。由于 Cassandra 要为每一台机器上面的数据都提供备份，当集群机器的数量比较大的时候，选择哪些机器作为数据的备份就尤为重要，特别是当需要跨数据中心的时候，就需要提供机架感应的相关功能。

所有和数据备份的相关代码都在 org. apache. cassandra. locator 中，如图 5-6 所示。

图 5-6 org. apache. cassandra. locator 代码结构

图 5-6 的代码结构主要包括两个部分的实现：EndpointSnitch（机架感应）与 ReplicationStrategy（数据的备份策略）。

5.3.1 EndpointSnitch

通过机架感应，Cassandra 集群中的每一个节点都可以知道哪几台节点和自己属于一个机架，哪几台节点和自己属于一个数据中心。

所有的机架感应策略都实现了 org. apache. cassandra. locator. IEndpointSnitch 接口。IendpointSnitch 接口包含以下几个重要的方法：

- public String getRack(InetAddress endpoint)：判断某一个节点所属的机架名称。
- public String getDatacenter(InetAddress endpoint)：判断某一个节点所属的数据中心名称。
- public List < InetAddress > sortByProximity(InetAddress address, List < InetAddress > addresses)：根据需要比较排序的地址，按照由近到远的规则对地址列表排序。

Cassandra 提供了 4 种实现，可以直接在配置文件中指定选择使用的实现类型。配置文件中默认的选项如下：

```
endpoint_snitch: org.apache.cassandra.locator.SimpleSnitch
dynamic_snitch: true
```

（1）SimpleSnitch

SimpleSnitch 是最简单的一种实现，它不提供机架和数据中心的功能，对节点距离排序就是直接返回，实现如下：

```
public List < InetAddress > sortByProximity(final InetAddress address, List < InetAddress > addresses)
{
    return addresses;
}
```

（2）PropertyFileSnitch

如果使用 PropertyFileSnitch，需要在 ClassPath 下添加一个名为“cassandra-rack. properties”的配置文件，里面为每一个节点的地址指定对应的数据中心和机架的名称。格式为：IP = Data Center:Rack，如：

```
10.21.119.13 = DC3 :RAC1
10.21.119.10 = DC3 :RAC1

10.0.0.13 = DC1 :RAC2
10.21.119.14 = DC3 :RAC2
10.20.114.15 = DC2 :RAC2

# default for unknown nodes
default = DC1 :r1
```

如果 Cassandra 启动后，这个文件被修改，修改后的内容也会在 1 分钟内生效。

PropertyFileSnitch 会根据 cassandra-rack. properties 文件中指定的内容返回某一个地址所属的机架与数据中心，对节点距离排序也会考虑机架和数据中心的因素，如代码清单 5-5 所示。

代码清单 5-5　考虑机架与数据中心的排序逻辑

```
public List < InetAddress > sortByProximity(final InetAddress address, List < InetAddress > addresses)
```

```
    {
        Collections.sort(addresses, new
Comparator < InetAddress > ()
        {
            public int compare(InetAddress a1, InetAddress a2)
            {
                return compareEndpoints(address, a1, a2);
            };
        });
        return addresses;
    }

    public int compareEndpoints(InetAddress address, InetAddress a1, InetAddress a2)
    {
        if (address.equals(a1) && !address.equals(a2))
            return -1;
        if (address.equals(a2) && !address.equals(a1))
            return 1;

        String addressRack = getRack(address);
        String a1Rack = getRack(a1);
        String a2Rack = getRack(a2);
        if (addressRack.equals(a1Rack) && !addressRack.equals(a2Rack))
            return -1;
        if (addressRack.equals(a2Rack) && !addressRack.equals(a1Rack))
            return 1;

        String addressDatacenter = getDatacenter(address);
        String a1Datacenter = getDatacenter(a1);
        String a2Datacenter = getDatacenter(a2);
        if (addressDatacenter.equals(a1Datacenter) && !addressDatacenter.equals
    (a2Datacenter))
            return -1;
        if (addressDatacenter.equals(a2Datacenter) && !addressDatacenter.equals
    (a1Datacenter))
            return 1;

        return 0;
    }
```

（3）RackInferringSnitch

RackInferringSnitch 的实现与 PropertyFileSnitch 类似，有同样的节点距离排序规则（参考代码清单 5-5），不同的地方在于判断节点所属机架和数据中心的逻辑，实现如下：

```
public String getRack(InetAddress endpoint)
{
```

```
    return Byte.toString(endpoint.getAddress()[2]);
}

public String getDatacenter(InetAddress endpoint)
{
    return Byte.toString(endpoint.getAddress()[1]);
}
```

（4）DynamicEndpointSnitch

DynamicEndpointSnitch 是一个特殊的实现，它并不能被单独使用，而必须和之前介绍的 3 种 EndpointSnitch 搭配使用。DynamicEndpointSnitch 能在原有 EndpointSnitch 的基础上，记录节点与节点之间通信的时间间隔，判断节点之间通信的快慢，从而达到根据实际的通信速度动态选择合适节点的目的。

如果希望将 DynamicEndpointSnitch 与 RackInferringSnitch 搭配使用，配置文件中的设置应该如下：

```
endpoint_snitch: org.apache.cassandra.locator. RackInferringSnitch
dynamic_snitch: true
```

如果希望单独使用 RackInferringSnitch，配置文件中的设置应该如下：

```
endpoint_snitch: org.apache.cassandra.locator. RackInferringSnitch
dynamic_snitch: false
```

这两种配置的区别就在于 dynamic_snitch 的设置：true 代表使用 DynamicEndpointSnitch，false 代表不使用 DynamicEndpointSnitch。

5.3.2　ReplicationStrategy

通过 ReplicationStrategy，Cassandra 集群可以知道任意一份数据备份的节点信息，同时在节点失效的时候，还能够计算出应该接收 HINT 消息的节点。

org. apache. cassandra. locator. AbstractReplicationStrategy 是所有 ReplicationStrategy 的基类，它包含以下几个重要的方法：

- public ArrayList < InetAddress > getNaturalEndpoints(Token searchToken)：从集群中找出负责所有 Token（searchToken）对应数据的节点集合。
- public abstract List < InetAddress > calculateNaturalEndpoints(Token searchToken, TokenMetadata tokenMetadata)：计算在指定的一致性哈希圆环（TokenMetadata）中，负责所有 Token（searchToken）对应数据的节点集合。
- public Multimap < InetAddress, InetAddress > getHintedEndpoints(Collection < InetAddress > targets)：如果目标节点中（targets）存在失效的节点，根据 endpointSnitch 从目标节点中计算出最合适的 HINT 节点。这个方法在 Cassandra 更新数据的时候使用，如果 Cassandra 在更新数据时发现某个节点不可用，将会把数据发送给另外一台 HINT 节点，

HINT 节点将缓存这部分数据到 SystemTable 中，等这个不可用的节点恢复后，再将缓存的数据发送给对应的节点，并将 HINT 节点存储在 SystemTable 中的数据删除。

Cassandra 提供了 3 种 ReplicationStrategy 实现：

（1）SimpleStrategy

这是最简单的 ReplicationStrategy 实现，它会根据指定的 Token 在指定的一致性哈希圆环中按照顺时针方向找出下 *N* 个需要备份的节点。SimpleStrategy 实现逻辑如代码清单 5-6 所示。

代码清单 5-6　SimpleStrategy 寻找备份节点逻辑

```
    public List<InetAddress> calculateNaturalEndpoints(Token token, TokenMetadata
metadata)
    {
        int replicas = getReplicationFactor();
        ArrayList<Token> tokens = metadata.sortedTokens();
        List<InetAddress> endpoints = new ArrayList<InetAddress>(replicas);

        if (tokens.isEmpty())
            return endpoints;

        Iterator<Token> iter = TokenMetadata.ringIterator(tokens, token);
        while (endpoints.size() < replicas && iter.hasNext())
        {
            endpoints.add(metadata.getEndpoint(iter.next()));
        }

        if (endpoints.size() < replicas)
            throw new IllegalStateException(String.format("replication factor (%s)
    exceeds number of endpoints (%s)", replicas, endpoints.size()));

        return endpoints;
    }
```

（2）OldNetworkTopologyStrategy

OldNetworkTopologyStrategy 的备份策略与 SimpleStrategy 的备份策略类似：根据指定的 Token 在指定的一致性哈希圆环中按照顺时针方向找出下 *N* 个需要备份的节点。不同的是：OldNetworkTopologyStrategy 在寻找第二个备份节点的时候，会找一个与第一个备份节点不在同一个数据中心的节点进行备份；寻找第三个备份节点的时候，会找一个与第二个备份节点同数据中心，但是不同机架的节点进行备份；接下来所有的备份节点寻找策略就按照 SimpleStrategy 的备份策略继续寻找。OldNetworkTopologyStrategy 实现逻辑如代码清单 5-7 所示。

代码清单 5-7　OldNetworkTopologyStrategy 寻找备份节点逻辑

```
    public List<InetAddress> calculateNaturalEndpoints(Token token, TokenMetadata
metadata)
```

```
{
    int replicas = getReplicationFactor();
    List < InetAddress > endpoints = new ArrayList < InetAddress > (replicas);
    ArrayList < Token > tokens = metadata.sortedTokens();

    if (tokens.isEmpty())
        return endpoints;

    Iterator < Token > iter = TokenMetadata.ringIterator(tokens, token);
    Token primaryToken = iter.next();
    endpoints.add(metadata.getEndpoint(primaryToken));

    boolean bDataCenter = false;
    boolean bOtherRack = false;
    while (endpoints.size() < replicas && iter.hasNext())
    {
        Token t = iter.next();
        if (! snitch.getDatacenter (metadata.getEndpoint (primaryToken)) .equals
(snitch.getDatacenter(metadata.getEndpoint(t))))
        {
            // If we have already found something in a diff datacenter no need to
find another
            if (!bDataCenter)
            {
                endpoints.add(metadata.getEndpoint(t));
                bDataCenter = true;
            }
            continue;
        }
            if (! snitch.getRack (metadata.getEndpoint (primaryToken)) .equals
(snitch.getRack (metadata.getEndpoint (t))) && snitch.getDatacenter (metada-
ta.getEndpoint (primaryToken)).equals (snitch.getDatacenter (metadata.getEndpoint
(t))))
        {
            //If we have already found something in a diff rack no need to find an-
other
            if (!bOtherRack)
            {
                endpoints.add(metadata.getEndpoint(t));
                bOtherRack = true;
            }
        }

    }

    if (endpoints.size() < replicas)
```

```
    {
        iter = TokenMetadata.ringIterator(tokens, token);
        while (endpoints.size() < replicas && iter.hasNext())
        {
            Token t = iter.next();
            if (!endpoints.contains(metadata.getEndpoint(t)))
                endpoints.add(metadata.getEndpoint(t));
        }

        if (endpoints.size() < replicas)
            throw new IllegalStateException(String.format("replication factor (%
s) exceeds number of endpoints (% s)", replicas, endpoints.size()));
    }

    return endpoints;
}
```

（3）NetworkTopologyStrategy

NetworkTopologyStrategy 在 OldNetworkTopologyStrategy 的基础上，可以更加详细地指定每一个数据中心需要备份的数据份数。比如我们需要在 DC1 中备份 3 份，DC2 中备份 2 份，那么在配置文件中的配置信息如下：

```
    replica_placement_strategy:
org.apache.cassandra.locator.NetworkTopologyStrategy
    strategy_options:
      DC1 : 3
      DC2 : 2
```

配置文件中就不需要再指定 replication_factor 的参数了，因为上面的设置中就已经决定了 replication_factor 的份数为 5。

NetworkTopologyStrategy 会先尝试在同一个数据中心中选择不同的机架作为该数据中心的备份节点，如果该数据中心找不到更多的机架，就会在同一个机架中寻找多个节点进行备份。最终保证每一个数据中心都有相应的备份数，并且每一个数据中心备份的节点尽可能在不同的机架中。NetworkTopologyStrategy 实现逻辑如代码清单 5-8 所示。

代码清单 5-8　NetworkTopologyStrategy 寻找备份节点逻辑

```
    public List < InetAddress > calculateNaturalEndpoints(Token searchToken, Token-
Metadata tokenMetadata)
    {
        int totalReplicas = getReplicationFactor();
        Map < String, Integer > remainingReplicas = new HashMap < String, Integer > (dat-
    acenters);
        Map < String, Set < String >> dcUsedRacks = new HashMap < String, Set < String >> ();
        List < InetAddress > endpoints = new ArrayList < InetAddress > (totalReplicas);
```

```
    for (Iterator<Token> iter=TokenMetadata.ringIterator(tokenMetadata.-
sortedTokens(), searchToken);
         endpoints.size() < totalReplicas && iter.hasNext();)
    {
        Token token=iter.next();
        InetAddress endpoint=tokenMetadata.getEndpoint(token);
        String datacenter=snitch.getDatacenter(endpoint);
        int remaining=remainingReplicas.containsKey(datacenter) ?remaining-
Replicas.get(datacenter) :0;
        if (remaining > 0)
        {
            Set<String> usedRacks=dcUsedRacks.get(datacenter);
            if (usedRacks==null)
            {
                usedRacks=new HashSet<String>();
                dcUsedRacks.put(datacenter, usedRacks);
            }
            String rack=snitch.getRack(endpoint);
            if (!usedRacks.contains(rack))
            {
                endpoints.add(endpoint);
                usedRacks.add(rack);
                remainingReplicas.put(datacenter, remaining-1);
            }
        }
    }

    for (Iterator<Token> iter=TokenMetadata.ringIterator(tokenMetadata.
sortedTokens(), searchToken);
         endpoints.size() < totalReplicas && iter.hasNext();)
    {
        Token token=iter.next();
        InetAddress endpoint=tokenMetadata.getEndpoint(token);
        if (endpoints.contains(endpoint))
            continue;

        String datacenter=snitch.getDatacenter(endpoint);
        int remaining=remainingReplicas.containsKey(datacenter) ?remaining-
Replicas.get(datacenter) :0;
        if (remaining > 0)
        {
            endpoints.add(endpoint);
            remainingReplicas.put(datacenter, remaining-1);
        }
    }
```

```
    for (Map.Entry<String, Integer> entry : remainingReplicas.entrySet())
    {
        if (entry.getValue() > 0)
            throw new IllegalStateException(String.format("datacenter (%s) has no
more endpoints, (%s) replicas still needed", entry.getKey(), entry.getValue()));
    }

    return endpoints;
}
```

5.4 集群状态变化的处理机制

在 Cassandra 中，如果节点与节点之间通过 Gossiper 协议发现集群中的状态发生了变化（机器失效、新的机器加入等），将以事件的形式通知这些事件的订阅者。所有对这些事件感兴趣的订阅者需要实现 org. apache. cassandra. gms. IendpointStateChangeSubscriber 接口，它包含的重要方法如下：

- public void onJoin(InetAddress endpoint, EndpointState epState)：当有新节点加入集群的时候，触发该事件。
- public void onChange(InetAddress endpoint, ApplicationState state, VersionedValue value)：当有新节点加入集群的时候，触发该事件。
- public void onAlive(InetAddress endpoint, EndpointState state)：当有失效节点恢复服务的时候，触发该事件。
- public void onDead(InetAddress endpoint, EndpointState state)：当有节点失效的时候，触发该事件。
- public void onRemove(InetAddress endpoint)：当有节点被移出集群的时候，触发该事件。

org. apache. cassandra. gms. IendpointStateChangeSubscriber 接口有 3 个实现类：StorageLoadBalancer、StorageService 和 MigrationManager。

5.4.1 StorageLoadBalancer

StorageLoadBalancer 能够计算集群中每一个节点的负载（磁盘中数据量的大小），并提供 Cassandra 集群的负载均衡。但是目前 Cassandra 的版本（0.7.x）中，并没有启用负载均衡的特性，而仅仅是计算集群中每一个节点的负载。

每当集群状态发生变化的时候，StorageLoadBalancer 所做的事情就是将节点负载变化传播出去。

当集群中某个节点的状态发生改变的时候，处理逻辑如下：

```
    public void onChange(InetAddress endpoint, ApplicationState state, VersionedVal-
ue value)
    {
        if (state != ApplicationState.LOAD)
            return;
        loadInfo_.put(endpoint, Double.parseDouble(value.value));
    }
```

当集群中有新节点加入的时候，处理逻辑如下：

```
    public void onJoin(InetAddress endpoint, EndpointState epState)
    {
        VersionedValue localValue = epState.getApplicationState(ApplicationState.LOAD);
        if (localValue != null)
        {
            onChange(endpoint, ApplicationState.LOAD, localValue);
        }
    }
```

5.4.2 StorageService

StorageService抽象了整个Cassandra集群的关系。通过它可以获取到整个集群的信息，并且可以管理整个集群，如删除某一个节点、移动某一个节点、为一个新加入的节点初始化数据等。

当集群中某个节点的状态发生改变的时候，处理逻辑如下：

```
    public void onChange(InetAddress endpoint, ApplicationState state, VersionedVal-
ue value)
    {
        if (state != ApplicationState.STATUS)
            return;

        String apStateValue = value.value;
        String[] pieces = apStateValue.split(VersionedValue.DELIMITER_STR, -1);
        assert (pieces.length > 0);

        String moveName = pieces[0];

        if (moveName.equals(VersionedValue.STATUS_BOOTSTRAPPING))
            handleStateBootstrap(endpoint, pieces);
        else if (moveName.equals(VersionedValue.STATUS_NORMAL))
            handleStateNormal(endpoint, pieces);
        else if (moveName.equals(VersionedValue.STATUS_LEAVING))
            handleStateLeaving(endpoint, pieces);
        else if (moveName.equals(VersionedValue.STATUS_LEFT))
            handleStateLeft(endpoint, pieces);
    }
```

当集群中有新节点加入的时候，处理逻辑如下：

```
public void onJoin(InetAddress endpoint, EndpointState epState)
{
  . for (Map.Entry < ApplicationState, VersionedValue > entry : epState.getAp-
plicationStateMap().entrySet())
    {
        onChange(endpoint, entry.getKey(), entry.getValue());
    }
}
```

当有失效节点恢复服务的时候，处理逻辑如下：

```
public void onAlive(InetAddress endpoint, EndpointState state)
{
    if (!isClientMode)
        deliverHints(endpoint);
}
```

当有节点被移出集群的时候，处理逻辑如下：

```
public void onRemove(InetAddress endpoint)
{
    tokenMetadata_.removeEndpoint(endpoint);
    calculatePendingRanges();
}
```

当有节点失效的时候，处理逻辑如下：

```
public void onDead(InetAddress endpoint, EndpointState state)
{
    MessagingService.instance.convict(endpoint);
}
```

5.4.3 MigrationManager

0.7.x 之前的 Cassandra 版本无法在集群运行的时候动态地修改 Schema 信息，如修改 ColumnFamily 的属性、新增 Keyspace 等。MigrationManager 的出现解决了这个问题。

如果集群中有节点的 Schema 信息发生了变更，将触发相应的事件，将变更信息应用到整个集群中。

当集群中某个节点的状态发生改变的时候，处理逻辑如下：

```
public void onChange(InetAddress endpoint, ApplicationState state, VersionedVal-
ue value)
    {
        if (state != ApplicationState.SCHEMA)
            return;
```

```
        UUID theirVersion = UUID.fromString(value.value);
        rectify(theirVersion, endpoint);
    }
```

当有失效节点恢复服务的时候，处理逻辑如下：

```
public void onChange(InetAddress endpoint, ApplicationState state, Versioned-
Value value)
{
    if (state != ApplicationState.SCHEMA)
        return;
    UUID theirVersion = UUID.fromString(value.value);
    rectify(theirVersion, endpoint);
}
```

5.5　本章小结

本章从原理和源代码实现上讲解了 Cassandra 集群的机制，包括一致性哈希、Gossiper 通信协议、集群的备份机制和集群状态变化的处理机制。这些机制是理解 Cassandra 内部如何工作的基础。基于这点，后面的章节将详细讲解 Cassandra 如何实现数据的写入与读取以及相应的操作。

只有理解了 Cassandra 内部的工作机制，才能更好地使用和优化 Cassandra。

第 6 章

Cassandra 的内部数据存储结构

本章内容

- Cassandra 中的数据存放规则
- Commilog
- Memtable
- SSTable
- 系统表空间
- 本章小结

在第 5 章中了解了 Cassandra 的集群机制，本章将深入 Cassandra 的内部数据存储结构，详细了解 SSTable 中的数据是如何写入以及如何被使用的。

6.1 Cassandra 中的数据存放规则

Cassandra 的配置文件可以对 Cassandra 中的数据存放信息进行配置。cassandra. yaml 中关于存放数据信息的配置如下：

```
# directories where Cassandra should store data on disk.
data_file_directories:
    - /var/lib/cassandra/data
# commit log
commitlog_directory: /var/lib/cassandra/commitlog
# saved caches
saved_caches_directory: /var/lib/cassandra/saved_caches
```

数据信息一共分为以下 3 类。

- data 目录：用于存储真正的数据文件，即后面将要讲到的 SSTable 文件。如果服务器有多个磁盘，可以指定多个目录，每一个目录都在不同的磁盘中。这样 Cassandra 就可以利用更多的硬盘空间。
- commitlog 目录：用于存储未写入 SSTable 中的数据，每次 Cassandra 系统中有数据写入，都会先将数据记录在该日志文件中，以保证 Cassandra 在任何情况下宕机都不会丢失数据。如果服务器有足够多的磁盘，可以将本目录设置在一个与 data 目录和 cache 目录不同的磁盘中，以提升读写性能。
- cache 目录：用于存储系统中的缓存数据。可以在 cassandra. yaml 文件中定义 Column Family 的属性中定义与缓存相关的信息，如缓存数据的大小（对应配置文件中的 keys_cached 和 rows_cached）、持久化缓存数据的时间间隔（对应配置文件中的 row_cache_save_period_in_seconds 和 key_cache_save_period_in_seconds）等。当 Cassandra 系统重启的时候，会从该目录下加载缓存数据。如果服务器有足够多的磁盘空间，可以将本目录设置在一个与 data 目录和 commitlog 目录不同的磁盘中，以提升读写性能。

在 data 目录下，Cassandra 会将每一个 Keyspace 中的数据存储在不同的文件目录下，并且 Keyspace 文件目录的名称与 Keyspace 名称相同。

假设有两个 Keyspace，分别为 ks1 和 ks2，但是在 data 目录下，将看到 3 个不同的目录：ks1，ks2 和 system。其中 ks1 和 ks2 用于存储系统定义的两个 Keyspace 的数据，另外一个 system 目录是 Cassandra 系统默认的一个 Keyspace，叫做 system，它用来存储 Cassandra 系统的相关元数据信息以及 HINT 数据信息，这个将会在后面的章节进行详细的介绍。

6.2 Commilog

当 Cassandra 有数据需要更新时，第一个记录这个更新的地方就是 Commitlog。

Commitlog 由如下两个部分构成：

```
CommitLog-1288024771844.log
CommitLog-1288024771844.log.header
```

在 CommitLog-1288024771844.log 文件中，保存了每一次更新操作的值。

在 CommitLog-1288024771844.log.header 文件中，记录了哪些数据已经从 memtable 中写入 SSTable 中。

通过 log.header 文件中记录的元数据信息，Cassandra 可以及时删除不必要的 Commitlog 文件，减少磁盘的占用量，并在 Cassandra 重启时，加快从 Commitlog 中恢复数据的速度。

Commitlog 文件的大小可以在配置文件中指定，默认是 128MB。

```
# Size to allow commitlog to grow to before creating a new segment
commitlog_rotation_threshold_in_mb: 128
```

当一个 Commitlog 文件大小超过设置的阈值后，将会新建一个 Commitlog，并将更新数据写入这个新的文件中。

Cassandra 提供了两种记录 Commitlog 的方式：周期记录（periodic）和批量记录（batch）。

如果使用周期记录的方式，需要在配置文件进行如下配置：

```
commitlog_sync: periodic
commitlog_sync_period_in_ms: 10000
```

Cassandra 会每次更新信息将写入 Commitlog 中，并且每隔一定的时间间隔（commitlog_sync_period_in_ms）调用 org.apache.cassandra.io.util.BufferedRandomAccessFile.sync() 同步 Commitlog 文件。

如果使用批量记录的方式，需要在配置文件进行如下配置：

```
commitlog_sync: batch
commitlog_sync_batch_window_in_ms: 100
```

Cassandra 会缓存每次更新信息，每隔一定的时间间隔（commitlog_sync_batch_window_in_ms）调用 org.apache.cassandra.io.util.BufferedRandomAccessFile.sync() 同步 Commitlog 文件，最后将之前缓存的更新信息写入 Commitlog 中。

如果不允许数据丢失，可以使用周期的方式记录 Commitlog。如果写入数据量非常大，同时可以承担由于机器可能宕机导致的数据丢失的风险，则使用批量记录的方式记录 Commitlog。

在实际的使用中，可以根据情况来选用合适的 Commitlog 记录方式。

6.3　Memtable

数据写入 Commitlog 后，将缓存在 Memtable 中。

Cassandra 中每一个 Memtable 只为一个 ColumnFamily 提供服务。

当下面 3 个条件中任意一个满足后，会将 Memtable 中缓存的数据写入磁盘，形成一个 SSTable 文件。

- Memtable 中缓存的数据达到容量（memtable_throughput_in_mb）上限。
- Memtable 中包含的数据的条数（memtable_operations_in_millions）超过上限。
- Memtable 距离上一次将缓存的数据写入磁盘的时间超过某一个间隔（memtable_flush_after_mins）。

上面提到的 3 个参数都可以在配置文件中进行设置，Cassandra 为每一个 ColumnFamily 提供单独的配置。

```
memtable_flush_after_mins: 59
memtable_throughput_in_mb: 255
memtable_operations_in_millions: 0.29
```

在 Memtable 中缓存数据的成员变量如下：

```
private final ConcurrentNavigableMap < DecoratedKey, ColumnFamily > columnFami-
lies = new ConcurrentSkipListMap < DecoratedKey, ColumnFamily > ();
```

每当有数据进入 Memtable 中时，会将数据保存到成员变量 ColumnFamilies 中，并解析这个数据，排除重复或者是已经过期的数据。具体实现如下：

```
void put (DecoratedKey key, ColumnFamily columnFamily)
{
    currentThroughput.addAndGet(cf.size());
    currentOperations.addAndGet(cf.getColumnCount());

    ColumnFamily oldCf = columnFamilies.putIfAbsent(key, cf);
    if (oldCf == null)
        return;

    oldCf.resolve(cf);
}
```

当 Cassandra 需要将 Memtable 中缓存的数据写入磁盘时，会按照内存中 Key 的顺序写入 SSTable 中。

```
private SSTableReader writeSortedContents() throws IOException
{
    logger.info("Writing " + this);
    SSTableWriter writer = new SSTableWriter(cfs.getFlushPath(), columnFamili-
es.size(), cfs.metadata, cfs.partitioner);

    for (Map.Entry < DecoratedKey, ColumnFamily > entry : columnFamilies.entrySet())
        writer.append(entry.getKey(), entry.getValue());

    SSTableReader ssTable = writer.closeAndOpenReader();
```

```
        logger.info("Completed flushing " + ssTable.getFilename());
        return ssTable;
    }
```

使用 Memtable 的优势在于：将随机 IO 写变为顺序 IO 写，降低大量的写操作对存储系统的压力。

6.4　SSTable

Cassandra 中的 Memtable 会缓存客户端写入的数据，当 Memtable 中缓存的某一个 ColumnFamily 中的数据量（对应配置文件中的 memtable_throughput_in_mb 和 memtable_operations_in_millions）或者超过上一次生成 SSTable 的时间（对应配置文件中的 memtable_flush_after_mins）后，Cassandra 会将 Memtable 中对应的 ColumnFamily 的数据持久化到磁盘中，生成一个 SSTable 文件。

如 ColumnFamily 名称为 Cf1 的一个 SSTable 文件由如下文件组成：

```
Cf1-e-1-Data.db
Cf1-e-1-Filter.db
Cf1-e-1-Index.db
Cf1-e-1-Statistics.db
```

其中，“Cf1”为 ColumnFamily 的名称；“e”为版本的标识（这个标识在 0.7 之前的版本中是没有的）；“1”代表这是名称为 Cf1 的 ColumnFamily 的第一个 SSTable，这个数字会随着新的 SSTable 文件的生成不断增加；“Data”、“Filter”、“Index”和“Statistics”分别代表 SSTable 4 个不同组成部分，它们的作用各不相同。

6.4.1　Filter 文件

Filter 文件用于快速定位某一个 Key 是否在该 SSTable 文件中存在。

为了判断某一个 Key 是否存在于 SSTable 文件中，Cassandra 使用了布隆过滤器（Bloom Filter）。Cassandra 使用布隆过滤器的好处在于可以减少占用的存储空间和内存空间，并提高查询的速度。

比如 Gmail 那样的公众电子邮件（E-mail）提供商，总是需要过滤垃圾邮件，一个办法就是记录下那些发垃圾邮件的 E-mail 地址[⊖]。由于那些发送者不停地注册新的地址，将它们都存起来需要大量的网络服务器。如果用哈希表，每存储 1 亿个 E-mail 地址，就需要 1.6GB 的内存——这是完全不能接受的！而使用布隆过滤器，解决同样的问题只需要哈希表 1/8 到 1/4 的大小。

假定我们存储 1 亿个电子邮件地址，我们先建立一个 16 亿二进制（比特）即 2 亿字节的向量，然后将这 16 亿个二进制全部设置为 0。对于每一个电子邮件地址 X，用 8 个不同

⊖　在全世界范围内，至少有几十亿个发送垃圾邮件的地址。

的随机数产生器(F1,F2,…,F8)产生8个信息指纹(f1,f2,…,f8)，再用一个随机数产生器G把这8个信息指纹映射到1到16亿中的8个自然数(g1,g2,…,g8)。把这8个位置的二进制全部设置为1。当我们对这1亿个E-mail地址进行这样的处理后，一个针对这些E-mail地址的布隆过滤器就建成了，如图6-1所示。

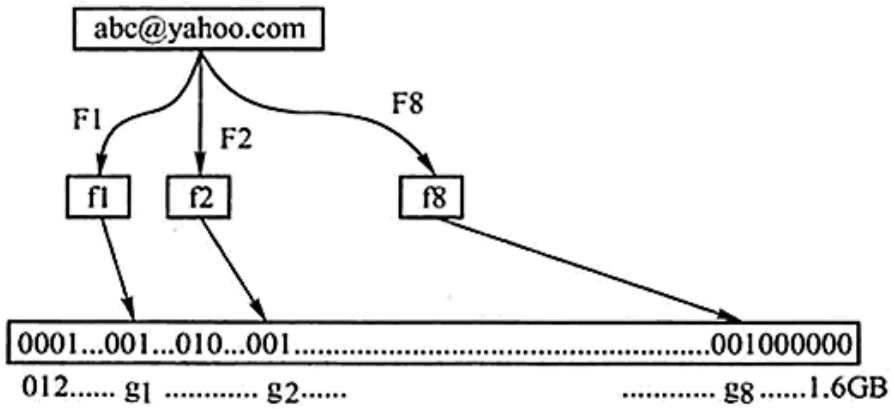


图6-1 布隆过滤器示意图

Cassandra中实现了org. apache. cassandra. utils. BloomFilter，它的实现原理与上面讲到的一致。每一个BloomFilter实例的内部包含一个java. util. BitSet类型的成员变量，filter_保存在内存中，判断某一个Key是否存在的逻辑如下：

```
public boolean isPresent(ByteBuffer key)
{
    for (int bucketIndex : getHashBuckets(key))
    {
        if (!filter_.get(bucketIndex))
        {
            return false;
        }
    }
    return true;
}
```

Cassandra将Memtable中的数据持久化为SSTable的过程中，将构造一个SSTableWriter实例，然后依次将Memtable中的数据传递给SSTableWriter实例，最后再关闭SSTableWiter实例。SSTable生成逻辑如代码清单6-1所示。

代码清单6-1 SSTable生成逻辑

```
SSTableWriter writer = new SSTableWriter(cfs.getFlushPath(), columnFamili-
es.size(), cfs.metadata, cfs.partitioner);

for (Map.Entry<DecoratedKey, ColumnFamily> entry : columnFamilies.entrySet())
    writer.append(entry.getKey(), entry.getValue());

SSTableReader ssTable = writer.closeAndOpenReader();
```

在初始化 SSTable 实例的时候，同时也会构造一个 BloomFilter 实例，在调用 writer. append(entry. getKey() , entry. getValue()) 的过程中，会将数据传输给 BloomFilter 实例处理，让其构建缓存位数组的 filter_，最后调用 writer. closeAndOpenReader() ，将 BloomFilter 实例中的数据写入到磁盘中，形成 Filter 文件，同时在这个过程中构建的 filter_将一直保存在内存中，用于判断某一个 Key 是否存在。

构建 filter_的逻辑如下：

```
for (int bucketIndex : getHashBuckets(key))
{
    filter_.set(bucketIndex);
}
```

写入 Filter 文件的逻辑如下：

```
FileOutputStream fos = new FileOutputStream(desc.filenameFor(SSTable.COMPONENT_
FILTER));
DataOutputStream stream = new DataOutputStream(fos);
BloomFilter.serializer().serialize(bf, stream);
stream.flush();
fos.getFD().sync();
stream.close();
```

当 Cassandra 重启时，将会读取 Filter 文件，从中加载到内存中。从 Fliter 文件中加载的逻辑如下：

```
DataInputStream stream = new DataInputStream(new FileInputStream(descrip-
tor.filenameFor(Component.FILTER)));
try
{
    bf = BloomFilter.serializer().deserialize(stream);
}
finally
{
    stream.close();
}
```

6.4.2　Index 文件

Filter 文件的作用在于判断某一个 Key 是否存在。如果 Key 存在，接下来的工作就是如何在 Index 文件中找到这个 Key 对应的 Column 值在 Data 文件中的具体位置。

在 Index 文件中，保存两类信息：Key 值和该 Key 在 Data 文件中对应的位置。Index 文件中保存 SSTable 中所有的 Key 以及该 Key 在 Data 文件中的位置，如图 6-2 所示。

如果要寻找某一个 Key 在 Data 文件中的位置，只需要顺序扫描 Index 文件就可以了。

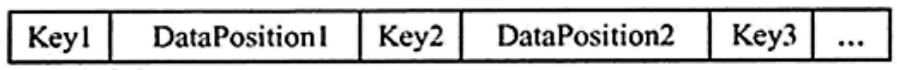


图 6-2 Index 文件存储结构

但是大多数情况下，SSTable 中会包含大量的 Key，顺序扫描 Index 文件将耗费大量的磁盘 IO 资源。另外注意一点：Index 文件中的 Key 是按照由小到大的顺序写的（详细原因请参考第 5 章的内容）。所以 Cassandra 在这一点上做了一个缓存优化：将 Index 文件中的部分 Key 以及该 Key 对应在 Index 文件中的位置（这里不是 Data 文件的位置了，而是 Index 文件的位置）按照某一个间距保存在内存中。假设这个间距为 128，一共有 600 个 Key，那么内存中缓存的 Key 分别为：Key1，Key128，Key256，Key384，Key512。当要查询某一个 Key 在 Data 文件中的位置时，会首先查询内存中的 Key，通过二分查找定位到 Key 所在的 Index 文件中的区间范围，然后再读取部分 Index 文件即可找到 Key 在 Data 文件中的位置了。如，要找 Key128，在内存中可以直接定位到 Index 文件的位置，然后读取 Index 文件即可。又如，要找 Key200 的位置，通过二分查找，知道 Key200 在 Index 文件的 Key128 和 Key256 之间，这样只要读取 Index 文件的一部分内容就可以迅速定位该 Key 的位置了。

上面所说的 Key 的间距可以在配置文件中如下设置：

```
# The Index Interval determines how large the sampling of row keys
# is for a given SSTable. The larger the sampling, the more effective
# the index is at the cost of space.
index_interval: 128
```

可见，这个参数设置得越小，读取 Key 的速度越快，但是消耗的内存越大。

Index 文件是在生成 SSTable 文件时产生的，根据代码清单 6-1，在初始化 SSTable 实例的时候，同时也会构造一个 IndexWriter 实例，在调用 writer. append (entry. getKey ( ) , entry. getValue( )) 的过程中，会将数据传输给 IndexWriter 实例处理，并构建自己的内存缓存 Summary，最后调用 writer. closeAndOpenReader( ) 时，将 IndexWriter 实例中的数据写入到磁盘中，形成 Index 文件，同时在这个过程中构建的 Summary 将一直保存在内存中，用于加速查找 Key 的位置。

构建 Summary 的逻辑如下：

```
if (keysWritten % DatabaseDescriptor.getIndexInterval() == 0)
    indexPositions.add(new KeyPosition(decoratedKey, indexPosition));
keysWritten ++;
```

写入 Index 文件的逻辑如下：

```
long indexPosition = indexFile.getFilePointer();
FBUtilities.writeShortByteArray(key.key, indexFile);
indexFile.writeLong(dataPosition);

summary.maybeAddEntry(key, indexPosition);
```

当 Cassandra 重启时，将会读取 Index 文件，从中加载到内存中。从 Index 文件中加载的逻辑如代码清单 6-2 所示。

代码清单 6-2　Index 文件加载逻辑

```
indexSummary = new IndexSummary(estimatedKeys);
if (recreatebloom)
    //estimate key count based on index length
    bf = BloomFilter.getFilter(estimatedKeys, 15);
while (true)
{
    long indexPosition = input.getFilePointer();
    if (indexPosition == indexSize)
        break;

    boolean shouldAddEntry = indexSummary.shouldAddEntry();
    ByteBuffer key = (ByteBuffer) ((shouldAddEntry || cacheLoading || recreate-
bloom)
                ? FBUtilities.readShortByteArray(input)
                : FBUtilities.skipShortByteArray(input));
    long dataPosition = input.readLong();
    if (key != null)
    {
        DecoratedKey decoratedKey = decodeKey(partitioner, descriptor, key);
        if (recreatebloom)
            bf.add(decoratedKey.key);
        if (shouldAddEntry)
            indexSummary.addEntry(decoratedKey, indexPosition);
        if (cacheLoading && keysToLoadInCache.contains(decoratedKey))
            keyCache.put(new Pair(descriptor, decoratedKey), dataPosition);
    }

    indexSummary.incrementRowid();
    ibuilder.addPotentialBoundary(indexPosition);
    dbuilder.addPotentialBoundary(dataPosition);
}
indexSummary.complete();
```

Cassandra 还提供一个叫做 KeyCache 的缓存，用于缓存上一次查找的 Key 在 Data 文件中的位置。这样当下一次查询同样的 Key 时，就不需要读取 Index 文件了。这个参数可以在配置文件 Keys_cached 中进行配置。

注意：在代码清单 6-2 中，加载 Index 文件的同时，如果存在 KeyCache，同样会加载到内存中。

6.4.3　Data 文件

上面介绍的 Filter 文件和 Index 文件都是为 Data 文件提供服务的，只有在 Data 文件中才

会存储真正的数据。但是 Data 文件又不仅仅存储了需要查询的数据，另外还存储了某一个 Key 对应的一些 Column 的索引信息。

在 Data 文件中，保存 3 类信息：Key 值、ColumnIndex 和 ColumnValue 的值，如图 6-3 所示。

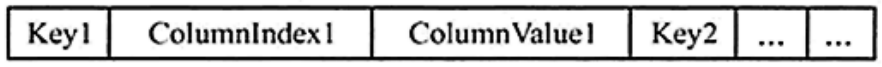


图 6-3 Data 文件存储结构

其中 ColumnIndex 包括两类信息：Bloom Filter 和 Index。

当读取 Data 文件中某一个 Key 中的某一个 Column 的时候，Cassandra 会先利用 ColumnIndex 中的 Bloom Filter 判断该 Column 是否存在，然后再利用 Index 快速找到实际的 Column 的值。这里所说的 Index 和上面提到的 SummaryIndex 工作原理是一致的，因为某一个 Key 下的所有 Column 也是按照一定固定的顺序由小到大存放的。在默认情况下，ColumnIndex 中的 Index 只会记录第一个 Column 和最后一个 Column 的位置，只有该 Key 下的 Column 大小超过了一个阈值之后才会记录其他的 Column 的位置。

这个参数可以在配置文件中进行配置：

```
# Add column indexes to a row after its contents reach this size.
# Increase if your column values are large, or if you have a very large
# number of columns. The competing causes are, Cassandra has to
# deserialize this much of the row to read a single column, so you want
# it to be small - at least if you do many partial-row reads - but all
# the index data is read for each access, so you don't want to generate
# that wastefully either.
column_index_size_in_kb: 64
```

另外需要注意的是，这里使用的 Index 只会对第一层级的 Column 建索引，即对标准类型的 ColumnFamliy 是非常有效的，一个 Key 下面包含成百上千个 Column 都没有问题，可以通过索引快速定位 Column 的位置。但是这个机制对于 Super 类型的 ColumnFamliy 就不起作用了。比如某一个 Key 下面的某一个 SuperColumn 下有成百上千个 Column，Cassandra 只会对 SuperColumn 建索引。所以对于这种查询，Cassandra 需要顺序遍历这个 SuperColumn 下的 Column 才能找到合适的值，这种情况下读取速度非常慢，不建议使用这种类型的数据建模方案。

Data 文件是在生成 SSTable 文件时产生的，根据代码清单 6-1，在初始化 SSTable 实例的时候，同时也会构造一个 BufferedRandomAccessFile（dataFile）实例，在调用 writer. append(entry. getKey() , entry. getValue()) 的过程中，会将数据传输给 dataFile 实例处理，并构建 ColumnIndex，最后调用 writer. closeAndOpenReader() 时，将 dataFile 实例中的数据写入磁盘中，形成 Data 文件。

构建 ColumnIndex 的逻辑如代码清单 6-3 所示。

代码清单 6-3　ColumnIndex 构建

```
    public static void serializeInternal(IIterableColumns columns, DataOutput dos)
throws IOException
    {
        int columnCount = columns.getEstimatedColumnCount();

        BloomFilter bf = BloomFilter.getFilter(columnCount, 4);

        if (columnCount == 0)
        {
            writeEmptyHeader(dos, bf);
            return;
        }

        //update bloom filter and create a list of IndexInfo objects marking the first
    and last column
        //in each block of ColumnIndexSize
        List < IndexHelper.IndexInfo > indexList = new ArrayList < IndexHelper.
    IndexInfo > ();
        int endPosition = 0, startPosition = -1;
        int indexSizeInBytes = 0;
        IColumn lastColumn = null, firstColumn = null;
        for (IColumn column : columns)
        {
            bf.add(column.name());

            if (firstColumn == null)
            {
                firstColumn = column;
                startPosition = endPosition;
            }
            endPosition += column.serializedSize();
            /* if we hit the column index size that we have to index after, go ahead and
index it. */
             if (endPosition - startPosition >= DatabaseDescriptor.getColumnInde-
xSize())
            {
                 IndexHelper.IndexInfo cIndexInfo = new IndexHelper.IndexInfo(first-
Column.name(), column.name(), startPosition, endPosition - startPosition);
                indexList.add(cIndexInfo);
                indexSizeInBytes += cIndexInfo.serializedSize();
                firstColumn = null;
            }

            lastColumn = column;
```

```
    }

    //all columns were GC'd after all
    if (lastColumn == null)
    {
        writeEmptyHeader(dos, bf);
        return;
    }

    //the last column may have fallen on an index boundary already. if not, index it
explicitly.
    if (indexList.isEmpty() || columns.getComparator().compare(indexList.get(in-
dexList.size() - 1).lastName, lastColumn.name()) != 0)
    {
         IndexHelper.IndexInfo cIndexInfo = new IndexHelper.IndexInfo(firstCol-
umn.name(), lastColumn.name(), startPosition, endPosition - startPosition);
        indexList.add(cIndexInfo);
        indexSizeInBytes += cIndexInfo.serializedSize();
    }

    /* Write out the bloom filter. */
    writeBloomFilter(dos, bf);

    //write the index. we should always have at least one computed index block, but
we only write it out if there is more than that.
    assert indexSizeInBytes > 0;
    if (indexList.size() > 1)
    {
        dos.writeInt(indexSizeInBytes);
        for (IndexHelper.IndexInfo cIndexInfo : indexList)
        {
            cIndexInfo.serialize(dos);
        }
    }
    else
    {
        dos.writeInt(0);
    }
}
```

写入 Data 文件的逻辑如下：

```
long startPosition = beforeAppend(decoratedKey);
FBUtilities.writeShortByteArray(decoratedKey.key, dataFile);
//write placeholder for the row size, since we don't know it yet
long sizePosition = dataFile.getFilePointer();
```

```
dataFile.writeLong(-1);
//write out row data
int columnCount = ColumnFamily.serializer().serializeWithIndexes(cf, dataFile);
//seek back and write the row size (not including the size Long itself)
long endPosition = dataFile.getFilePointer();
dataFile.seek(sizePosition);
dataFile.writeLong(endPosition - (sizePosition + 8));
//finally, reset for next row
dataFile.seek(endPosition);
```

当 Cassandra 重启时，不对 Data 文件进行特殊处理。

6.4.4　Statistics 文件

Statistics 文件是在 0.7 版本中新增的，它用来存储 SSTable 中所包含的 Column 的个数与 Row 的大小。

Statistics 文件是在生成 SSTable 文件时产生的，在调用 writer. append(entry. getKey() , entry. getValue()) 的过程中，Cassandra 会记录 Column 的个数与 Row 的大小。

```
estimatedRowSize.add(dataFile.getFilePointer() - currentPosition);
estimatedColumnCount.add(row.columnCount());
```

在调用 writer. closeAndOpenReader() 时，Cassandra 会将这些统计信息写入 Statistics 文件中。

```
    DataOutputStream out = new DataOutputStream(new FileOutputStream(desc.filename-
For(SSTable.COMPONENT_STATS)));
    EstimatedHistogram.serializer.serialize(rowSizes, out);
    EstimatedHistogram.serializer.serialize(rowSizes, out);
    out.close();
```

当 Cassandra 重启时，会从 Statistics 文件加载这些统计信息。

```
DataInputStream dis = new DataInputStream(new FileInputStream(statsFile));
rowSizes = EstimatedHistogram.serializer.deserialize(dis);
columnCounts = EstimatedHistogram.serializer.deserialize(dis);
dis.close();
```

6.5　系统表空间

在 Cassandra 中，除了用户自己定义的 Keyspace 之外，还有一个特殊的 Keyspace：名称为 system 的系统表空间。

用户不能在 Cassandra 中创建名为 system 的 Keyspace，只能由 Cassandra 系统自动创建。系统表空间的主要有以下两个作用：

- 管理 Cassandra 的系统元数据信息。

- 缓存 HINT 数据。

如果系统首次启动，Cassandra 将会自动在 data 目录下创建系统表空间，并将系统元数据信息存放在系统表空间中。以后启动的过程中，Cassandra 将会直接从系统表空间中读取系统元数据信息。

如果 Cassandra 发现某一个节点宕机，就会将发送给宕机节点的数据以 HINT 的形式发送给另外一台 Cassandra 服务器。接收到 HINT 数据的 Cassandra 服务器将数据缓存到系统表空间中，当其发现宕机的 Cassandra 恢复后，将缓存 HINT 数据发送给恢复的服务器，完成数据传输后，将缓存的 HINT 数据从系统表空间中删除。

6.6 本章小结

本章从原理上分析和讲解了 Cassandra 的内部数据存储结构 Commitlog、Memtable、SSTable 和构成 SSTable 的 4 个子文件。了解 Cassandra 的内部数据存储构造有利于为基于 Cassandra 的应用程序设计合理的数据模型，以及找出造成读写瓶颈的原因。另外还介绍了 Cassandra 的系统表空间，了解了整个系统元数据管理的机制。

第 7 章

Cassandra 的数据更新机制

本章内容

- 数据更新流程
- 集群数据更新策略
- 二级索引
- 本章小结

结合第6章讲解的Cassandra内部数据存储结构，本章将从整体上讲解整个Cassandra集群数据的更新机制，包括数据更新流程、不同的数据更新策略以及二级索引的构建方式。理解这部分内容有利于根据实际的情况选择合适的数据更新策略，从而更加高效地利用Cassandra。

在Cassandra中，数据的更新不仅仅包括写入操作，同时还包括数据的修改和数据的删除。这3类操作都适用于本章所讲解的内容。

7.1 数据更新流程

在Cassandra中更新数据需要经过3个过程：

- 更新数据写入Commitlog
- 更新数据写入Memtable
- 更新数据写入SSTable

整个流程如图7-1所示。

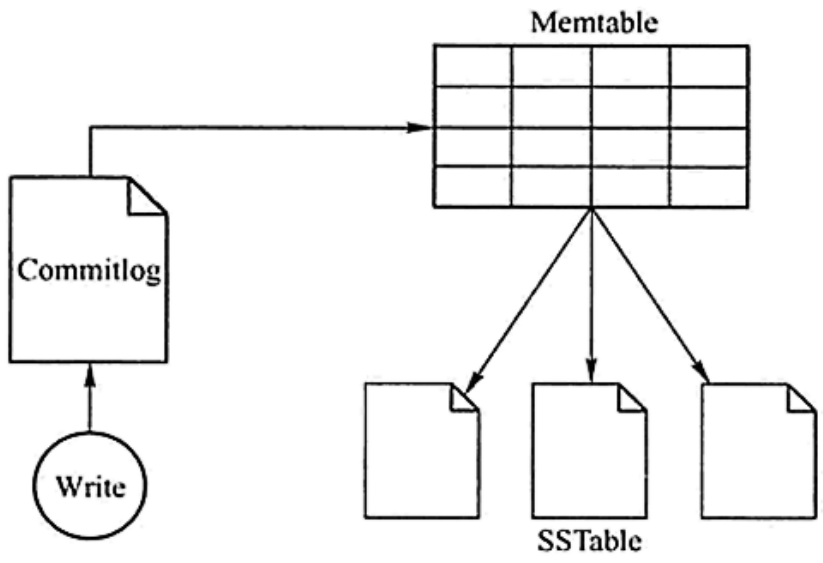


图7-1 数据更新流程

采用这个流程可以为Cassandra来带两个好处：数据不会丢失，只有顺序I/O写。

7.2 集群数据更新策略

在Cassandra集群中，数据是可以存在冗余的，这样可以保证在某几台服务器数据丢失后，提供数据冗余的服务器仍然可以提供服务。为了保证数据的冗余，每一个更新操作都会把数据发送给所有负责该数据的服务器。同时为了达到高可用性，不能等待所有提供数据冗余的服务器全部确认写入操作成功再提示客户端写入成功，Cassandra提供了6种一致性写入策略（如表7-1所示）来保证高可用性，写入操作无须等待所有的服务器都响应写入成功。

表 7-1　Cassandra 提供的 6 种一致性写入策略

名　称	描　　述
ANY	集群中任意一个服务器响应写入成功（包括 HINT 消息），则 Cassandra 就会通知客户端更新成功，否则通知客户端写入失败
ONE	集群中任意一个服务器响应写入成功（不包括 HINT 消息），则 Cassandra 就会通知客户端更新成功，否则通知客户端写入失败
QUORUM	集群中响应写入成功（不包括 HINT 消息）的服务器数量不小于“ReplicationFactor/2 + 1”，则 Cassandra 就会通知客户端更新成功，否则通知客户端写入失败 这种更新策略是最常用的更新策略，它是平衡数据一致性和高可用性的一个策略
LOCAL_QUORUM	要使用这种更新策略，必须使用 org. apache. cassandra. locator. NetworkTopologyStrategy 集群中响应写入成功（不包括 HINT 消息）的服务器数量为不小于“ReplicationFactor/2 +1”，同时写入成功的节点中有一台是同一个数据中心，则 Cassandra 就会通知客户端更新成功，否则通知客户端写入失败
EACH_QUORUM	要使用这种更新策略，必须使用 org. apache. cassandra. locator. NetworkTopologyStrategy 集群中响应写入成功（不包括 HINT 消息）的服务器数量为不小于“ReplicationFactor/2 +1”，同时写入成功的服务器中不是都在同一个数据中心，则 Cassandra 就会通知客户端更新成功，否则通知客户端写入失败
ALL	集群中响应写入成功的服务器数量等于 ReplicationFactor，则 Cassandra 就会通知客户端更新成功，否则通知客户端写入失败 当数据的一致性非常重要的时候，可以考虑使用这种更新策略

Cassandra 在更新数据的时候，可以通过 API 选择合适的更新策略。如使用 Thrift API 时，只需要再指定 ConsistencyLevel 即可。

如指定更新的级别为 ONE，则代码如下：

```
ConsistencyLevel.ONE
```

当集群中有服务器处于不可用状态，发送给不可用服务器的更新信息将缓存在集群其他机器中，等该不可用服务器恢复后，再将集群其他机器中的数据发送给恢复的服务器。这种消息在 Cassandra 中叫做 HINT 消息。HINT 消息缓存在系统自带的 system Keyspace 中。

整个数据更新的过程实现逻辑如代码清单 7-1 所示。

代码清单 7-1　数据更新逻辑

```
    public static void mutate(List < RowMutation > mutations, ConsistencyLevel con-
sistency_level) throws UnavailableException, TimeoutException
    {
        long startTime = System.nanoTime();
        ArrayList < IWriteResponseHandler > responseHandlers = new ArrayList < IWrit-
eResponseHandler > ();

        RowMutation mostRecentRowMutation = null;
        StorageService ss = StorageService.instance;

        try
        {
```

```
        for (RowMutation rm : mutations)
        {
            mostRecentRowMutation = rm;
            String table = rm.getTable();
            AbstractReplicationStrategy rs = Table.open(table).replicationStrategy;

            List < InetAddress > naturalEndpoints = ss.getNaturalEndpoints(table,
rm.key());
            Collection < InetAddress > writeEndpoints = ss.getTokenMetadata().
getWriteEndpoints(StorageService.getPartitioner().getToken(rm.key()), table,
naturalEndpoints);
            Multimap < InetAddress, InetAddress > hintedEndpoints = rs.getHinted-
Endpoints(writeEndpoints);

            //send out the writes, as in mutate() above, but this time with a call-
back that tracks responses
             final IWriteResponseHandler responseHandler = rs.getWriteResponse-
Handler(writeEndpoints, hintedEndpoints, consistency_level);
            responseHandler.assureSufficientLiveNodes();

            responseHandlers.add(responseHandler);
            Message unhintedMessage = null;
            for (Map.Entry < InetAddress, Collection < InetAddress > > entry : hint-
edEndpoints.asMap().entrySet())
            {
                InetAddress destination = entry.getKey();
                Collection < InetAddress > targets = entry.getValue();

                if (targets.size() ==1 && targets.iterator().next().equals(desti-
nation))
                {
                    //unhinted writes
                    if (destination.equals(FBUtilities.getLocalAddress()))
                    {
                        insertLocalMessage(rm, responseHandler);
                    }
                    else
                    {
                        //belongs on a different server. send it there.
                        if (unhintedMessage == null)
                        {
                            unhintedMessage = rm.makeRowMutationMessage();
                            MessagingService.instance.addCallback(responseHandler,
unhintedMessage.getMessageId());
                        }
                        if (logger.isDebugEnabled())
```

```
                        logger.debug ( " insert writing key " + FBUtili-
                        ties.bytesToHex(rm.key()) + " to " + unhintedMes-
                        sage.getMessageId() + "@ " +destination);

                    MessagingService.instance.sendOneWay(unhintedMessage,
destination);
                }
            }
            else
            {
                //hinted
                Message hintedMessage = rm.makeRowMutationMessage();
                for (InetAddress target : targets)
                {
                    if (!target.equals(destination))
                    {
                        addHintHeader(hintedMessage, target);
                        if (logger.isDebugEnabled())
                                logger.debug ("insert writing key " + FBUtili-
                                ties.bytesToHex(rm.key()) + " to " + hintedMes-
                                sage.getMessageId() + "@ " +destination + " for "
                                 +target);
                    }
                }
                        responseHandler.addHintCallback(hintedMessage, destination);

                        MessagingService.instance.sendOneWay(hintedMessage, des-
                        tination);
                    }
                }
            }
            //wait for writes. throws timeoutexception if necessary
            for (IWriteResponseHandler responseHandler : responseHandlers)
            {
                responseHandler.get();
            }
    }
    catch (IOException e)
    {
        if (mostRecentRowMutation == null)
            throw new RuntimeException("no mutations were seen but found an error dur-
ing write anyway", e);
            else
                throw new RuntimeException("error writing key " + FBUtilities.bytes-To-
Hex(mostRecentRowMutation.key()), e);
        }
```

```
        finally
        {
            writeStats.addNano(System.nanoTime() - startTime);
        }

    }
```

根据代码清单 7-1，整个数据更新过程如下：

1）Cassandra 获取到更新数据的所属的 Keyspace 的备份策略（ReplicationStrategy），通过备份策略和更新数据的 Key 计算出更新该数据需要写入哪些服务器中。

2）分析需要写入的这些服务器中是否发生宕机，如果有，需要另外寻找一个 HINT 服务器。

3）根据一致性级别，判断需要等待几个节点写入成功才能算整个写入操作成功。这个判断逻辑如下（LOCAL_QUORUM 和 EACH_QUORUM 除外）：

```
protected int determineBlockFor(String table)
{
    int blockFor = 0;
    switch (consistencyLevel)
    {
        case ONE:
            blockFor = 1;
            break;
        case QUORUM:
            blockFor = (writeEndpoints.size()/2) + 1;
            break;
        case ALL:
            blockFor = writeEndpoints.size();
            break;
        default:
            throw new UnsupportedOperationException("invalid consistency level:
" + consistencyLevel.toString());
    }
    return blockFor;
}
```

4）确保存活的服务器数量不小于需要响应写入成功服务器的数量。

5）向目标服务器发送更新数据，等待指定数量服务器响应成功后，告诉客户端更新成功，否则告诉客户端更新失败。

7.2.1 ANY

集群中任意一个服务器响应写入成功（包括 HINT 消息），则 Cassandra 就会通知客户端

更新成功，否则通知客户端写入失败。

这种数据更新策略的速度仅次于 ONE 的更新策略，但是数据的可靠性高于 ONE 的更新策略。

7.2.2　ONE

集群中任意一个服务器响应写入成功（不包括 HINT 消息），则 Cassandra 就会通知客户端更新成功，否则通知客户端写入失败。

这种数据更新策略是速度最快的，但是数据的可靠性较低，对于不重要但是更新比较频繁的数据可以采用这种更新策略。

7.2.3　QUORUM

集群中响应写入成功（不包括 HINT 消息）的服务器数量不小于“ReplicationFactor/2 +1”，则 Cassandra 就会通知客户端更新成功，否则通知客户端写入失败。

这种更新策略在不考虑数据中心的情况下，是最常用的更新策略，它是平衡数据一致性和高可用性的一个策略。

7.2.4　LOCAL_QUORUM

要使用这种更新策略，必须使用 org. apache. cassandra. locator. NetworkTopologyStrategy。

集群中响应写入成功（不包括 HINT 消息）的服务器数量不小于“ReplicationFactor/2 +1”，并且写入成功的节点中有一台与接受写入操作的服务器处于同一个数据中心时，Cassandra 就会通知客户端更新成功，否则通知客户端写入失败。

这种更新策略在考虑数据中心的情况下，是最常用的更新策略，它是平衡数据一致性和高可用性的一个策略。

7.2.5　EACH_QUORUM

要使用这种更新策略，必须使用 org. apache. cassandra. locator. NetworkTopologyStrategy。

集群中响应写入成功（不包括 HINT 消息）的服务器数量不小于“ReplicationFactor/2 +1”，并且写入成功的节点中有一台与接受写入操作的服务器不在同一个数据中心时，Cassandra 就会通知客户端更新成功，否则通知客户端写入失败。

这种更新策略在考虑数据中心的情况下，是最常用的更新策略，它是平衡数据一致性和高可用性的一个策略。

7.2.6　ALL

集群中响应写入成功的服务器数量等于 ReplicationFactor，则 Cassandra 就会通知客户端更新成功，否则通知客户端写入失败。

这种更新策略速度最慢，但是数据的可靠性最高，当数据的一致性非常重要的时候，

可以考虑使用这种更新策略。

7.3 二级索引

7.3.1 为什么需要二级索引

按照第 6 章 Cassandra 数据存储的结构，读取数据只能根据 Key 来获取 Key 下面的 Column 的值，而无法直接获取到那些 Column 的值满足某一个条件的值。

假如在实际的应用中我们的数据是这样保存的，如图 7-2 所示。

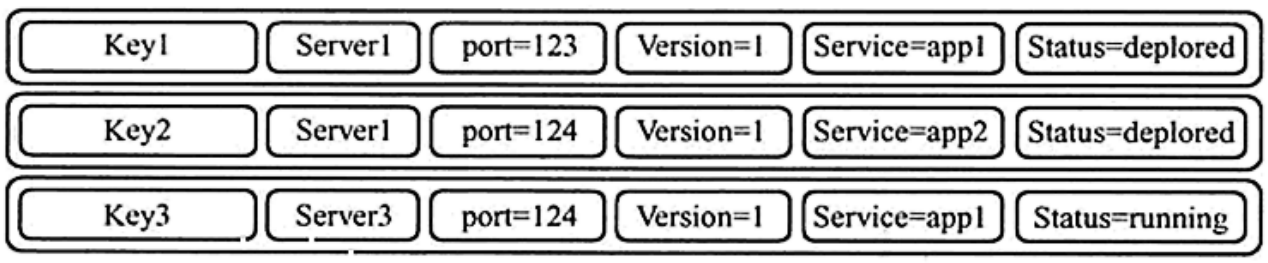


图 7-2 数据模型 1

每一个 Key 对应了一些列的 Column：port、Version、Service、Status 等。

通过某一个 Key，能够找到和这个 Key 相关的所有 Column 的值。

假设要找到 Column 的 name 为 Service 并且 Column 的值为 app2 的所有的 Key 值，那么应该如何操作呢？

根据 Cassandra 提供的 Thrift API，并不能直接找到这样的数据，唯一可以操作的方式就是通过 KeyRange 的方式，进行全表扫描，找到符合条件的值。

这种方式虽然可以解决问题，但是当数据量非常巨大的时候，进行这种查询的效率是非常有限的。

可以考虑使用二级索引的办法来解决这类的查询问题。

在原先数据模型的基础之上，根据实际的查询需求，建立以下额外的数据，如图 7-3 所示。

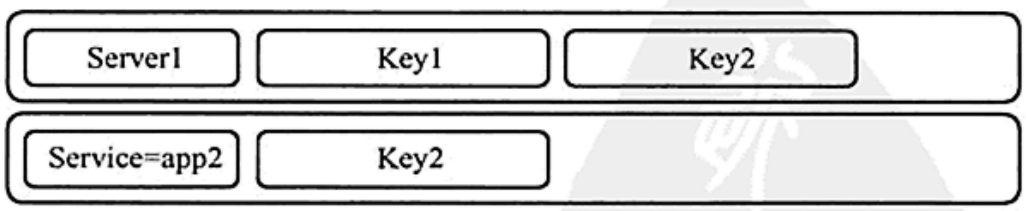


图 7-3 数据模型 2

建立 Key 为 Service = app2，其对应的值为 Key2。

这样，当需要查询所有的 Column 的 name 为 Service，并且 Column 的值为 app2 的所有的 Key 值时候，可以直接将 Service = app2 作为 Key 传给 Cassandra，然后就可以获得所有相应的 Key 了。

当想知道这样的 Key 中其他的 Column 值的时候，只要根据上一步找到的 Key，再次执行查询即可。

同样，想知道所有 Column 的 name 为 Service1 的所有 Key 值时候，可以直接将 Service1 作为 Key 传给 Cassandra，就可以获得所有相应的 Key 了。

这种方案的局限在于，需要索引的值越多，需要的额外空间就越大，同时当被索引的值进行修改的时候，索引的值也要同步进行修改。

7.3.2　Cassandra 二级索引更新过程

上面提到的自己手工建立二级索引的方式可以正常工作，但是带来了额外的人工维护成本。在 Cassandra 0.7 以后的版本中，提供了二级索引，其机制与上面讲述的原理一致，可以在配置文件中更加方便地配置和使用，而无须开发人员额外去维护二级索引。

要使用二级索引，只需要在配置文件中指定即可。

```
- name: Indexed1
  default_validation_class: LongType
  column_metadata:
    - name: birthdate
      validator_class: LongType
      index_type: KEYS
```

上面指定了一个二级索引，索引的名称为 Indexed1，索引的字段名称为 birthdate。

在 Cassandra 更新数据的过程中，将判断更新的数据是否需要建立索引，如果需要，将更新索引信息，如代码清单 7-2 所示。

代码清单 7-2　二级索引更新逻辑

```
SortedSet<ByteBuffer> mutatedIndexedColumns = null;
for (ByteBuffer column:cfs.getIndexedColumns())
{
    if (cf.getColumnNames().contains(column) || cf.isMarkedForDelete())
    {
        if (mutatedIndexedColumns == null)
            mutatedIndexedColumns = new TreeSet<ByteBuffer>();
        mutatedIndexedColumns.add(column);
    }
}

synchronized (indexLockFor(mutation.key()))
{
    ColumnFamily oldIndexedColumns = null;
    if (mutatedIndexedColumns != null)
    {
        //with the raw data CF, we can just apply every update in any order and let
        //read-time resolution throw out obsolete versions, thus avoiding read -
```

```
before-write.
        // but for indexed data we need to make sure that we're not creating
index entries
        //for obsolete writes
        oldIndexedColumns = readCurrentIndexedColumns(key, cfs, mutatedIndexed-
Columns);
        ignoreObsoleteMutations(cf, mutatedIndexedColumns, oldIndexedColumns);
    }

    Memtable fullMemtable = cfs.apply(key, cf);
    if (fullMemtable != null)
        memtablesToFlush = addFullMemtable(memtablesToFlush, fullMemtable);

    if (mutatedIndexedColumns != null)
    {
        // ignore full index memtables -- we flush those when the "master" one
is full
        applyIndexUpdates(mutation.key(), cf, cfs, mutatedIndexedColumns, oldIn-
dexedColumns);
    }
}
```

注意：在代码清单7-2中，如果在更新二级索引的过程中，Cassandra发现该索引信息已经存在，将更新之前的所有信息。

从本质上看，二级索引就是一个ColumnFamily，更新的数据同样先写入Commitlog，然后缓存到Memtable中，最后写入SSTable中。

7.4 本章小结

本章讲解了Cassandra的数据更新机制，在实际的应用中使用合适的集群数据更新策略是本章的重点。同时介绍了Cassandra中建立二级索引的机制，建立二级索引将会对Cassandra系统带来额外的负担，如何合理选用二级索引，这需要读者在实际的业务场景中多加思考。

第 8 章

Cassandra 的数据读取机制

本章内容

- 数据读取流程
- 集群数据读取策略
- 读修复
- 数据缓存
- 二级索引
- 本章小结

结合第 6 章中讲解的 Cassandra 内部数据存储结构，本章将从整体上讲解整个 Cassandra 集群数据的读取机制，包括数据读取流程、不同的数据读取策略、数据缓存以及如何利用二级索引快速查询相应的数据。理解这部分内容有利于根据实际的情况选择合适的数据读取策略、数据缓存方式以及二级索引，从而更加高效地利用 Cassandra。

8.1 数据读取流程

Cassandra 在写入数据的过程中，会为每一个 ColumnFamily 生成一个或者多个 SSTable 文件。所以在数据的读取过程中，Cassandra 会根据需要读取的 ColumnFamily 查询该 Column Family 下的 Memtable 以及所有的 SSTable，合并查询的结果，将最新的结果返回给客户端。Cassandra 读取数据的整体流程如图 8-1 所示。

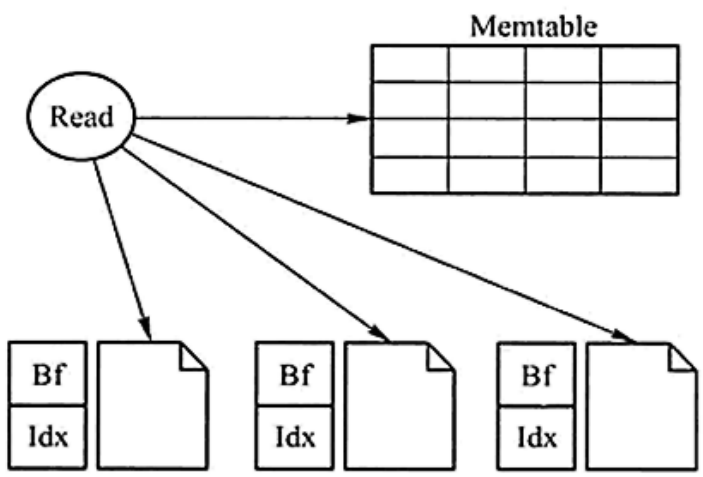


图 8-1 数据读取流程

Cassandra 从 Memtable 中获取数据，只要直接查询 Memtable 的成员变量 ColumnFamilies 即可。

Cassandra 从 SSTable 中获取数据，先要读取 Bloom Filer 文件判断该 Key 是否在本 SSTable 文件中，如果存在，再从 Index 文件中定位到数据的位置，最后从 Data 文件中读取需要查询的信息。

由于 Cassandra 的存储模型是面向列的。所以很有可能出现这样的情况：Key 为 k1 的数据，在 Memtable 中包含列名为 cm 的值 vm，而在第一个 SSTable 中包含列名为 s1 的值 vs1，在第二个 SSTable 中包含列名为 s2 的旧值 vs2old，并且在第三个 SSTable 中包含列名为 s2 的新值 vs2new。当客户端查询 Key 为 k1 的所有列的值的时候，Cassandra 会分别读取 Memtable、第一个 SSTable、第二个 SSTable 和第三个 SSTable 的值并进行汇总，最终的结果为 k1，有 3 个列：cm、s1 和 s2，列的值分别为 vm、vs1 和 vs2new。

Cassandra 的读取操作分为两类：弱读取（Weak Read）和强读取（Strong Read）。

8.1.1 弱读取

如果读取的一致性级别为 ONE，那么采用的读取方式为弱读取。

弱读取的实现逻辑如下：

```
    private static List < Row > weakRead (List < ReadCommand > commands) throws IOEx-
ception, UnavailableException, TimeoutException
    {
        List < Row > rows = new ArrayList < Row > ();

        //send off all the commands asynchronously
```

```
    List < Future < Object > > localFutures = null;
    List < IAsyncResult > remoteResults = null;
    for (ReadCommand command: commands)
    {
          InetAddress endPoint = StorageService.instance.findSuitableEndpoint
(command.table, command.key);
        if (endPoint.equals(FBUtilities.getLocalAddress()))
        {
            if (logger.isDebugEnabled())
                logger.debug("weakread reading " + command + " locally");

            if (localFutures == null)
                localFutures = new ArrayList < Future < Object > > ();
            Callable < Object > callable = new weakReadLocalCallable(command);

             localFutures.add(StageManager.getStage(Stage.READ).submit(calla-
ble));
        }
        else
        {
            if (remoteResults == null)
                remoteResults = new ArrayList < IAsyncResult > ();
            Message message = command.makeReadMessage();
            if (logger.isDebugEnabled())
                logger.debug("weakread reading " + command + " from " + mes-
sage.getMessageId() + "@ " + endPoint);
            if (randomlyReadRepair(command))
                message.setHeader(ReadCommand.DO_REPAIR, ReadCommand.DO_RE-
PAIR.getBytes());

             remoteResults.add(MessagingService.instance.sendRR(message, end-
Point));
        }
    }

    //wait for results
    if (localFutures != null)
    {
        for (Future < Object > future : localFutures)
        {
            Row row;
            try
            {
                row = (Row) future.get();
            }
            catch (Exception e)
```

```
            {
                throw new RuntimeException(e);
            }
            rows.add(row);
        }
    }
    if (remoteResults != null)
    {
        for (IAsyncResult iar: remoteResults)
        {
            byte[] body;
            body = iar.get(DatabaseDescriptor.getRpcTimeout(), TimeUnit.MILLISECONDS);
            ByteArrayInputStream bufIn = new ByteArrayInputStream(body);
            ReadResponse response = ReadResponse.serializer().deserialize(new
DataInputStream(bufIn));
            if (response.row() != null)
                rows.add(response.row());
        }
    }

    return rows;
}
```

在弱读取中，对于每一个读取命令，Cassandra 的执行流程如下：

1）从集群中找出来一台最适合读取的服务器。

2）从最适合读取的服务器读取数据。

- 如果最适合读取的服务器是本机，对本机的数据进行异步读取。
- 如果最适合读取的服务器不是本机，向该服务器请求需要读取的数据，并根据一定的概率（read_repair_chance）计算是否进行读修复操作（注意，这里的读修复是异步执行的，Cassandra 不会等待读修复操作完毕后再将结果返回给客户端）。

3）等待结果返回。

4）将返回的结果返回给客户端。

8.1.2 强读取

如果读取的一致性级别不是 ONE，则采用的读取方式为强读取。

强读取的实现逻辑如下：

```
    private static List < Row > strongRead(List < ReadCommand > commands, Consisten-
cyLevel consistency_level) throws IOException, UnavailableException, TimeoutExcep-
tion
    {
        List < QuorumResponseHandler < Row > > quorumResponseHandlers = new ArrayList
```

```
<QuorumResponseHandler<Row>>();
    List<InetAddress[]> commandEndpoints=new ArrayList<InetAddress[]>();
    List<Row> rows=new ArrayList<Row>();

    //send out read requests
    for (ReadCommand command: commands)
    {
        assert !command.isDigestQuery();
        ReadCommand readMessageDigestOnly=command.copy();
        readMessageDigestOnly.setDigestQuery(true);
        Message message=command.makeReadMessage();
        Message messageDigestOnly=readMessageDigestOnly.makeReadMessage();

        InetAddress dataPoint=StorageService.instance.findSuitableEndpoint
(command.table, command.key);
        List<InetAddress> endpointList=StorageService.instance.get-LiveNatu-
ralEndpoints(command.table, command.key);

        InetAddress[] endpoints=new InetAddress[endpointList.size()];
        Message messages[]=new Message[endpointList.size()];
          // data - request message is sent to dataPoint, the node that will
actually get
        //the data for us. The other replicas are only sent a digest query.
        int n=0;
        for (InetAddress endpoint : endpointList)
        {
            Message m=endpoint.equals(dataPoint)?message : messageDigestOnly;
            endpoints[n]=endpoint;
            messages[n++]=m;
            if (logger.isDebugEnabled())
                logger.debug("strongread reading "+(m==message ? "data" : "di-
gest")+" for "+command+" from "+m.getMessageId()+"@ "+endpoint);
        }
        AbstractReplicationStrategy rs=Table.open(command.table).replication-
Strategy;
        QuorumResponseHandler<Row> quorumResponseHandler=rs.getQuorum-Response-
Handler(new ReadResponseResolver(command.table), consistency_level);
        MessagingService.instance.sendRR(messages, endpoints, quorumResponse-
Handler);
        quorumResponseHandlers.add(quorumResponseHandler);
        commandEndpoints.add(endpoints);
    }

    //read results and make a second pass for any digest mismatches
    List<QuorumResponseHandler<Row>> repairResponseHandlers=null;
    for (int i=0; i < commands.size(); i++)
```

```
    {
        QuorumResponseHandler < Row > quorumResponseHandler = quorumResponseHan-
dlers.get(i);
        Row row;
        ReadCommand command = commands.get(i);
        try
        {
            long startTime2 = System.currentTimeMillis();
            row = quorumResponseHandler.get();
            if (row != null)
                rows.add(row);

            if (logger.isDebugEnabled())
                logger.debug("quorumResponseHandler: " + (System.current-TimeMil-
lis() - startTime2) + " ms.");
        }
        catch (DigestMismatchException ex)
        {
            AbstractReplicationStrategy rs = Table.open(command.table).replication-
Strategy;
            QuorumResponseHandler < Row > qrhRepair = rs.getQuorumResponseHandler
(new ReadResponseResolver(command.table), ConsistencyLevel.QUORUM);
            if (logger.isDebugEnabled())
                logger.debug("Digest mismatch:", ex);
            Message messageRepair = command.makeReadMessage();
            MessagingService.instance.sendRR(messageRepair, commandEndpoints.
get(i),qrhRepair);
            if (repairResponseHandlers == null)
                repairResponseHandlers = new ArrayList < QuorumResponseHandler <
Row >> ();
            repairResponseHandlers.add(qrhRepair);
        }
    }

    //read the results for the digest mismatch retries
    if (repairResponseHandlers != null)
    {
        for (QuorumResponseHandler < Row > handler : repairResponseHandlers)
        {
            try
            {
                Row row = handler.get();
                if (row != null)
                    rows.add(row);
            }
            catch (DigestMismatchException e)
```

```
            {
                throw new AssertionError(e); // full data requested from each node
here, no digests should be sent
            }
        }
    }

    return rows;
}
```

在强读取中，对于每一个读取命令，Cassandra 的执行流程如下：

1）从集群中找出一台最适合读取的服务器。

2）向该服务器请求需要读取的数据。

3）根据读取一致性级别，从集群中找出其他需要读取的服务器。

4）向其他需要读取的服务器请求需要读取数据的摘要信息，用于比对多台机器之间数据是否一致。

5）等待符合读取一致性要求结果的返回。判断需要等待的结果数的实现逻辑如下：

```
public int determineBlockFor(ConsistencyLevel consistencyLevel, String table)
{
    switch (consistencyLevel)
    {
        case ONE:
        case ANY:
            return 1;
        case QUORUM:
            return (DatabaseDescriptor.getReplicationFactor(table)/2)+1;
        case ALL:
            return DatabaseDescriptor.getReplicationFactor(table);
        default:
            throw new UnsupportedOperationException("invalid consistency level: "+
table.toString());
    }
}
```

6）从不同服务器返回的结果中，比对数据是否一致，对于不一致的数据，进行 QUORUM 级别的读修复操作。

7）将最终的结果返回给客户端。

8.2　集群数据读取策略

在 Cassandra 集群中，数据是可以存在冗余的，这样可以保证在某几台服务器数据丢失后，提供数据冗余的服务器仍然可以提供服务。

由于存在数据的冗余，不同的冗余数据之间就可以存在差异。所以 Cassandra 在读取数据的时候，需要读取所有的冗余数据，从多份数据中计算出最新的数据。同时 Cassandra 为了达到高可用性，不能等待所有提供数据冗余的服务器全部读取成功后，再将结果返回给客户端。Cassandra 提供了 5 种一致性读取策略来保证高可用性。

从 Cassandra 中获取数据时，可以通过 API 选择合适的更新策略，如使用 Thrift API 时，只需要在更新数据时指定 ConsistencyLevel 即可。

如指定更新的级别为 ONE，则代码如下：

```
ConsistencyLevel.ONE
```

8.2.1 ONE

集群中任意一个服务器返回读取，则 Cassandra 就会通知客户端读取成功，否则通知客户端读取失败。

这种读取策略速度最快。当应用对数据的读取速度要求很高，并且不需要获取最新的数据时，可以采用这种读取策略。

8.2.2 QUORUM

集群中读取成功的服务器数量不小于“ReplicationFactor/2 + 1”，则 Cassandra 就会通知客户端读取成功，否则通知客户端读取失败。

这种读取策略在不考虑数据中心的情况下，是最常用的读取策略，它能很好地平衡数据一致性和高可用性。

8.2.3 LOCAL_QUORUM

使用这种读取策略，必须使用 org. apache. cassandra. locator. NetworkTopologyStrategy。

集群中响应读取成功的服务器数量不小于“ReplicationFactor/2 + 1”，并且读取成功的节点中有一台与接受读取操作的服务器处于同一个数据中心时，Cassandra 就会通知客户端读取成功，否则通知客户端读取失败。

这种读取策略在考虑数据中心的情况下，是最常用的读取策略，它是平衡数据一致性和高可用性的一个策略。

8.2.4 EACH_QUORUM

使用这种读取策略，必须使用 org. apache. cassandra. locator. NetworkTopologyStrategy。

集群中响应读取成功的服务器数量不小于“ReplicationFactor/2 + 1”，并且读取成功的节点中有一台与接受读取操作的服务器不在同一个数据中心时，Cassandra 就会通知客户端读取成功，否则通知客户端读取失败。

这种读取策略在不考虑数据中心的情况下，是最常用的读取策略，它是平衡数据一致性和高可用性的一个策略。

8.2.5　ALL

集群中响应读取成功的服务器数量等于 ReplicationFactor，则 Cassandra 就会通知客户端读取成功，否则通知客户端读取失败。

当数据的一致性非常重要的时候，可以考虑使用这种更新策略。

8.3　读修复

读修复（Read Repair）是 Cassandra 中保证数据的最终一致性的很重要的功能。

大多数情况下，为了显示 Cassandra 集群的高可用性，会使用 ONE 的读取策略，这样会导致数据的不一致，所以需要进行读修复来修复过期的数据，使它们得到更新，从而保证数据的一致性。

但是读修复需要消耗一定的资源，显然对于每一个 ONE 级别的读操作都默认采取读修复操作是不合适的。

所以 Cassandra 提供了为每一个 Column Family 单独配置进行读修复的概率，如果希望每次读取都进行读修复操作，将其设置为 1.0 即可。配置文件中的如下配置代表有 1/10 的改进是针对读修复操作。

```
read_repair_chance: 0.1
```

读修复的实现逻辑如下：

```
ReadCommand readCommandDigestOnly = constructReadMessage(true);
try
{
    Message message = readCommandDigestOnly.makeReadMessage();
    if (logger_.isDebugEnabled())
        logger_.debug("Reading consistency digest for " + readCommand_.key + " from " +
message.getMessageId() + "@ [" + StringUtils.join(replicas_, ", ") + "]");

     MessagingService.instance.addCallback (new DigestResponseHandler (), mes-
sage.getMessageId());
    for (InetAddress endpoint : replicas_)
    {
        if (!endpoint.equals(FBUtilities.getLocalAddress()))
            MessagingService.instance.sendOneWay(message, endpoint);
    }
}
catch (IOException ex)
{
    throw new RuntimeException(ex);
}
```

在读修复的过程中，Cassandra 会向集群中每一个包含该数据并且存活的服务器发送该读取数据摘要信息的读取请求，然后进行比对找出最新的数据，并更新数据过期的服务器，从而让集群中每一个节点的数据都更新到最新的值，保证数据的一致性。

8.4 数据缓存

Cassandra 提供了两类缓存：列的缓存（RowCache）和 Key 的缓存（KeyCache）。Cassandra 可以为每一个 Column Family 单独配置缓存的个数。

```
keys_cached: 10000
rows_cached: 1000
```

除了定义缓存具体的个数之外，Cassandra 还提供了缓存的百分比。

```
keys_cached: 100%
rows_cached: 10%
```

另外，Cassandra 提供了周期性地将缓存中的数据持久化保存到硬盘中的功能，配置如下：

```
row_cache_save_period_in_seconds: 0
key_cache_save_period_in_seconds: 3600
```

如果设置为 0，表示不会将缓存中的数据持久化到硬盘中。持久化到硬盘中的具体位置也需要在配置文件中进行设置。

```
saved_caches_directory: /var/lib/cassandra/saved_caches
```

当 Cassandra 重启的时候，会从该目录中加载缓存信息到内存中。

8.4.1 RowCache

RowCache 缓存了读取的列信息，从 Cassandra 中读取数据会首先判断 RowCache 中是否有需要读取的数据。如果有，则直接从 RowCache 中找到相应的结果返回；如果没有，执行基本的读取流程，并将读取的结果缓存在 RowCache 中。

Cassandra 写入数据后会同步更新 RowCache，保证 Cassandra 从 RowCache 中可以读取到最新的信息。

8.4.2 KeyCache

KeyCache 与 RowCache 不同，KeyCache 缓存的是需要读取的数据在 SSTable Data 文件中的具体位置。

假设从 SSTable 中读取 Key = k1 的数据，Cassandra 会判断 KeyCache 中是否有 k1 的缓存，如果没有，通过 SSTable Index 文件寻找到 k1 在 SSTable Data 文件中的位置，并将该位

置缓存到 KeyCache 中。下次从 SSTable 中读取 k1 的数据时，就不需要从 SSTable Index 文件中寻找数据的位置了，而是直接从 SSTable Data 文件中读取。

8.5　二级索引

根据 7.3 节所讲的二级索引的机制，Cassandra 在使用二级索引查询的时候，不是直接去寻找存储数据的 ColumnFamily，而是先通过存储索引的 ColumnFamily 定位到存储数据的位置，再去存储数据的 ColumnFamily 中寻找实际的值。

当需要查找的值具有多个二级索引可以使用的时候，Cassandra 将判断使用哪个二级索引更加合适，这个特点有点类似于传统数据库中的执行计划。Cassandra 判断哪个二级索引更加适合的逻辑如下：

```
private IndexExpression highestSelectivityPredicate(IndexClause clause)
{
    IndexExpression best = null;
    int bestMeanCount = Integer.MAX_VALUE;
    for (IndexExpression expression : clause.expressions)
    {
        ColumnFamilyStore cfs = getIndexedColumnFamilyStore(expression.column_
name);
        if (cfs == null || !expression.op.equals(IndexOperator.EQ))
            continue;
        int columns = cfs.getMeanColumns();
        if (columns < bestMeanCount)
        {
            best = expression;
            bestMeanCount = columns;
        }
    }
    return best;
}
```

Cassandra 选用二级索引的判断依据非常简单：哪个二级索引关联的值少，就选用哪个二级索引。

8.6　本章小结

本章讲解了 Cassandra 的数据读取机制，在实际的应用中根据实际的情况使用合适的集群数据读取策略是本章的重点。同时介绍了 Cassandra 中读修复、数据缓存的机制，在实际使用中，如何为 Cassandra 设置合理的 RowCache、KeyCache 是提高系统读取速度的关键。

第9章

Cassandra的数据压缩机制

本章内容

- 为什么要进行数据压缩
- 如何控制数据压缩
- 数据压缩流程
- 维护 Cassandra 中的数据
- 本章小结

本章将详细讲解 Cassandra 中的数据压缩机制（Compaction）。先讲解为什么要进行数据压缩，然后讲解如何控制数据压缩行为以及压缩的流程，最后讲解其他形式的压缩操作。

9.1 为什么要进行数据压缩

通过前面的章节我们已经了解到，每一次 Cassandra 将 Memtable 中缓存的数据持久化到硬盘中都会形成一个全新的 SSTable 文件。SSTable 文件对于 Cassandra 而言是只读的，Cassandra 并不会修改已经存在的 SSTable 文件。

这样会造成如下 4 个严重的问题：

1）SSTable 文件的数量巨大，这将会耗尽操作系统的可操作文件数。

2）从 SSTable 文件中读取缓慢，因为每次读取某一个 Key 的值会查找所有包含该 Key 的 SSTable 文件，并且合并从不同 SSTable 文件中读取的值。

3）占用额外的硬盘。假设某一个 Key 在两个不同的 SSTable 文件中都存在，第一个 SSTable 文件中包含了该 Key 的 100 个 Column，另一个 SSTable 文件包含了该 Key 这 100 个 Column 的更新的值。那么第一个 SSTable 文件中保存的该 Key 的那 100 个 Column 的值就没有存在的必要了。

4）占用额外的内存。以上一个情况为例，不同的 SSTable 文件中存在重复的值或者过期的值，这些值都会增加 SSTable 文件中的 Filter 文件和 Index 文件的大小，而 Filter 文件和 Index 文件又会加载到内存中，从而降低了 Cassandra 的内存利用率。

Cassandra 提供的数据压缩机制可以将多个 SSTable 文件合并为一个 SSTable 文件，同时将多个 SSTable 文件中存在的重复的值、更新的值或者已经删除的值进行合并，从而解决上面提到的 4 个问题。

9.2 如何控制数据压缩

Cassandra 中的数据压缩是同步进行的，即 Cassandra 对外提供读写服务的同时，会在后台开启一个专门的线程进行数据压缩操作。

由于数据压缩操作将消耗大量的机器资源（如磁盘 IO、CPU、内存），为了尽可能减少压缩对读写服务的影响，可以在配置文件中对数据压缩线程的优先级进行设置，从而减少对读写服务的影响。

```
# change this to increase the compaction thread's priority. In java, 1 is the
# lowest priority and that is our default.
compaction_thread_priority: 1
```

Java 的线程数优先级分为 10 个级别，从 1 到 10，优先级依次递增。Cassandra 中所有线程的默认优先级为 5，压缩线程的默认优先级为 1。

另外，Cassandra 可以为每一个 ColunmFamily 指定当 SSTable 文件的数量等于或者超过

哪一个阈值后开始执行压缩的操作，如：

```
min_compaction_threshold: 5
max_compaction_threshold: 31
```

其中 min_compaction_threshold 代表当该 Colunm Family 中的 SSTable 文件的数量等于或者超过 min_compaction_threshold 后开始执行压缩操作，该参数的默认设置为 4。max_ compaction _ threshold 代表每次执行压缩操作最多合并的 SSTable 的文件个数，该参数的默认设置为 32。

如果上述两个参数中的任意一个为 0，代表取消 Cassandra 的数据压缩操作，这意味着除非手动触发数据压缩，否则 Cassandra 不会自动对该 ColunmFamily 中的 SSTable 文件进行压缩操作。

在 Cassandra 中配置的数据压缩参数也可以在 Cassandra 运行的过程中进行调整，只需要执行如下命令即可：

```
sh $CASSANDRA_HOME/bin/nodetool -h [hostname] -p [jmxport] setcompactionthreshold [cfname] [minthreshold] [maxthreshold]
```

执行的时候需要根据 Cassandra 的实际情况替换命令中用中括号包含的值。

比如运行 Cassandra 的机器名为 c1，JMX 监听端口为 8080，Cassandra 的安装目录为 /home/admin/cassandra-0.7，修改的 Column Family 的名称为 cf1，修改该 ColumnFamily 的最小、最大的压缩参数分别为 4、32。

```
sh /home/admin/cassandra-0.7/bin/nodetool -h c1 -p 8080 setcompactionthreshold cf1 4 32
```

如果要取消该 ColumnFamily 的自动压缩，则命令如下：

```
sh /home/admin/cassandra-0.7/bin/nodetool -h c1 -p 8080 setcompactionthreshold cf1 0 0
```

如果希望手动执行 Keyspace 名称为 k1 的压缩操作，可以执行如下命令：

```
sh /home/admin/cassandra-0.7/bin/nodetool -h c1 -p 8080 compact k1
```

Cassandra 接到该命令后，将对 Keyspace 名称为 k1 下的每一个 ColumnFamily 进行压缩，并且每一个 ColumnFamily 下的所有 SSTable 文件都将压缩成为 1 个 SSTable 文件。如果该操作一次性压缩较多的文件，需要谨慎使用。

9.3　数据压缩流程

在 Cassandra 中，整个数据压缩流程如下：

```
public Future<Integer> submitMinorIfNeeded(final ColumnFamilyStore cfs)
{
```

```
    Callable<Integer> callable = new Callable<Integer>()
    {
        public Integer call() throws IOException
        {
            Integer minThreshold = cfs.getMinimumCompactionThreshold();
            Integer maxThreshold = cfs.getMaximumCompactionThreshold();

            if (minThreshold == 0 || maxThreshold == 0)
            {
                logger.debug("Compaction is currently disabled.");
                return 0;
            }
            logger.debug("Checking to see if compaction of " + cfs.columnFamily + "
would be useful");
             Set<List<SSTableReader>> buckets = getBuckets(convertSSTable-
sToPairs(cfs.getSSTables()), 50L * 1024L * 1024L);
            updateEstimateFor(cfs, buckets);

            for (List<SSTableReader> sstables : buckets)
            {
                if (sstables.size() >= minThreshold)
                {
                    Collections.sort(sstables);
                    return doCompaction(cfs, sstables.subList(0, Math.min(sstables.
size(), maxThreshold)), (int) (System.currentTimeMillis() / 1000) -
cfs.metadata.gcGraceSeconds);
                }
            }
            return 0;
        }
    };
    return executor.submit(callable);
}
```

要进行数据压缩步骤如下：

1）判断是否符合压缩的条件。

2）如果 SSTable 的数量足够，就将需要压缩的 SSTable 文件按照大小进行分组。

3）对于已经分组的 SSTable 文件，如果数量满足压缩的要求，取分组中的前 max_compaction_threshold 个 SSTable 文件执行压缩。

Cassandra 在对 SSTabe 执行数据压缩的过程分为如下 3 个步骤：

1）判断是否需要在压缩前对需要压缩的文件做快照，同时判断需要压缩的磁盘空间是否满足，如果不满足，减少压缩文件的个数。

```
Table table = cfs.table;
if (DatabaseDescriptor.isSnapshotBeforeCompaction())
```

```
        table.snapshot("compact-" + cfs.columnFamily);
    logger.info("Compacting [" + StringUtils.join(sstables, ",") + "]");
    String compactionFileLocation = table.getDataFileLocation ( cfs.getExpected-
    CompactedFileSize(sstables));
    //If the compaction file path is null that means we have no space left for this com-
paction.
    //try again w/o the largest one.
    List<SSTableReader> smallerSSTables = new ArrayList<SSTableReader>(sstables);
    while (compactionFileLocation == null && smallerSSTables.size() > 1)
    {
        logger.warn("insufficient space to compact all requested files " + StringU-
tils.join(smallerSSTables, ", "));

smallerSSTables.remove(cfs.getMaxSizeFile(smallerSSTables));
        compactionFileLocation = table.getDataFileLocation(cfs.getExpectedCompactedFile-
Size(smallerSSTables));
    }
    if (compactionFileLocation == null)
    {
        logger.error("insufficient space to compact even the two smallest files,
aborting");
        return 0;
    }
    sstables = smallerSSTables;
```

在配置文件中可以指定是否在需要在压缩前对需要压缩的文件做快照。

```
# Whether or not to take a snapshot before each compaction. Be
# careful using this option, since Cassandra won't clean up the
# snapshots for you. Mostly useful if you're paranoid when there
# is a data format change.
snapshot_before_compaction: false
```

2）判断本次压缩操作是否属于主压缩（即该 Column Family 下面的所有 SSTable 文件都被压缩），构建遍历需要压缩的 SSTable 文件的迭代器，创建 SSTableWriter，开始遍历需要压缩的 SSTable 文件，将压缩后的值通过 SSTableWriter 写入新的 SSTable 文件中。

```
boolean major = cfs.isCompleteSSTables(sstables);

long startTime = System.currentTimeMillis();
long totalkeysWritten = 0;

int expectedBloomFilterSize = Math.max(DatabaseDescriptor.getIndexInterval(),
(int)SSTableReader.getApproximateKeyCount(sstables));

SSTableWriter writer;
```

```
    CompactionIterator ci = new CompactionIterator(cfs, sstables, gcBefore, major);
    //retain a handle so we can call close()
     Iterator < AbstractCompactedRow > nni = new FilterIterator (ci, PredicateU-
tils.notNullPredicate());
    executor.beginCompaction(cfs, ci);

    try
    {
        if (!nni.hasNext())
        {
            cfs.markCompacted(sstables);
            return 0;
        }

         String newFilename = new File(cfs.getTempSSTablePath(compactionFileLoca-
tion)).getAbsolutePath();
        writer = new SSTableWriter(newFilename, expectedBloomFilterSize, cfs.meta-
data, cfs.partitioner);
        while (nni.hasNext())
        {
            AbstractCompactedRow row = nni.next();
            try
            {
                writer.append(row);
            }
            catch (IOException ex)
            {
                writer.abort();
                //rethrow the exception so that caller knows compaction failed.
                throw ex;
            }
            totalkeysWritten ++;
        }
    }
    finally
    {
        ci.close();
    }
```

在压缩的过程中，Cassandra 会合并已经更新的值和已经删除的值，但是在合并 SSTable 的过程中，删除值的操作只有在进行主压缩或需要删除的 Key 只存在于需要压缩的 SSTable 文件中时才会进行。

另外，在 0.7 之前的 Cassandra 版本中，如果某一个 Key 对应的值非常大，那么就需要大量的内存来保持所有需要压缩的值。如果这些值超过了内存的限制，就会出现内存不足

问题。

在 Cassandra 0.7 中，Cassandra 在配置文件中增加了如下一个配置选项：

```
# Size limit for rows being compacted in memory. Larger rows will spill
# over to disk and use a slower two-pass compaction process. A message
# will be logged specifying the row key.
in_memory_compaction_limit_in_mb: 64
```

如果 Cassandra 在执行压缩的时候，发现某一列的大小超过了 in_memory_compaction_limit_in_mb 的限制，将使用 org. apache. cassandra. io. LazilyCompactedRow 来进行压缩，从而避免了内存不足的问题。

3）标记已经压缩完毕的 SSTable 文件，并将压缩后的文件加载至内存中，再判断是否需要再次对该 ColumnFamily 进行数据压缩操作。

```
SSTableReader ssTable = writer.closeAndOpenReader(getMaxDataAge(sstables));
cfs.replaceCompactedSSTables(sstables, Arrays.asList(ssTable));
submitMinorIfNeeded(cfs);
```

Cassandra 中删除已经压缩完毕的 SSTable 文件是非常讲究的：它并没有直接将该文件从磁盘中删除，而是在该文件上打了一个标记，然后加入幽灵引用（phantomReference）队列中，等待 JVM 回收内存时，再执行真正的删除操作。

假如在 Cassandra 停机之前没有将压缩完毕的 SSTable 文件删除，那么将在下次 Cassandra 启动的时候执行删除操作。

9.4　维护 Cassandra 中的数据

为了保证数据的一致性，Cassandra 还提供了两种额外的压缩模式用于维护 Cassandra 中的数据，它们是：数据清理模式和数据一致性校验模式。这两类数据压缩操作在 Cassandra 中都无法自动执行，需要系统管理人员手动触发。

9.4.1　数据清理压缩

由于 Cassandra 集群中服务器之间的状态在变化，比如有新的节点加入、旧的节点退出、某一些节点出现宕机等，将会导致某些 Cassandra 服务器的 SSTable 文件中保持了不属于该节点的数据。

通过数据清理压缩，Cassandra 将根据自己检测自己的 SSTable 文件，将不属于本节点的数据从 SSTable 中清除。

要使用这个功能，可以使用如下命令：

```
sh $CASSANDRA_HOME/bin/nodetool -h [hostname] -p [jmxport] cleanup [keyspace]
```

如果指定了 Keyspace，将只对这个 Keyspace 执行数据清理压缩。如果没有指定，将对

所有的 Keyspace 执行数据清理压缩。

9.4.2 数据一致性校验压缩

由于 Cassandra 集群中的数据存在冗余，可能会出现多个节点之间数据不一致的情况。为了避免这个问题，Cassandra 提供了数据一致性校验压缩，它将检测各个数据节点中的数据，将过期的数据更新到最新的值，使得该节点上的数据与其他节点一致。

要使用这个功能，可以使用如下命令：

```
sh $CASSANDRA_HOME/bin/nodetool -h [hostname] -p [jmxport] repair [keyspace] [column family name]
```

9.5 本章小结

本章从原理上分析和讲解了 Cassandra 的数据压缩机制，并讲解了如何通过配置文件和命令行控制数据压缩。

理解 Cassandra 的数据压缩机制有利于选择合适的压缩策略，更加高效地使用 Cassandra。同时可以通过数据清理压缩和数据一致性校验压缩来维护 Cassandra 中的数据。

第 10 章

Cassandra 的启动流程

本章内容

- Cassandra 启动脚本
- Cassandra 启动流程
- 本章小结

在第9章中，我们了解了 Cassandra 的数据压缩机制，本章将讲解 Cassandra 是如何启动的，使用了哪些配置文件，以及在 Cassandra 启动的过程中将开启哪些线程，执行哪些操作。

10.1 Cassandra 启动脚本

在 Linux 系统（Windows 系统与其类似）中，只需要配置好 Java 运行环境及相关参数（JAVA_HOME）即可启动 Cassandra。启动 Cassandra 的命令如下：

```
sh CASSANDRA_HOME/bin/cassandra
```

其中 CASSANDRA_HOME 是 Cassandra 发行包在服务器中安装的位置。

要了解执行 CASSANDRA_HOME/bin/cassandra 脚本究竟发生了什么事情，可以阅读源码，也可以使用如下命令显示该脚本的每一步相关操作：

```
sh-x CASSANDRA_HOME/bin/cassandra
```

在执行 sh 命令后面加入参数 x，将显示 shell 脚本执行的每一步操作，可能的输出如下：

```
aaron@ ubuntu:~/cassandra $ sh-x bin/cassandra
+[x=x]
+dirname bin/cassandra
+[-r /usr/share/cassandra/cassandra.in.sh]
+[-r /usr/local/share/cassandra/cassandra.in.sh]
+[-r /opt/cassandra/cassandra.in.sh]
+[-r /home/aaron/.cassandra.in.sh]
+[-r bin/cassandra.in.sh]
+. bin/cassandra.in.sh
+[x=x]
+dirname bin/cassandra
+CASSANDRA_HOME=bin/...
+[x=x]
+CASSANDRA_CONF=bin/.../conf
+cassandra_bin=bin/.../build/classes
+CLASSPATH=bin/.../conf:bin/.../build/classes
+ CLASSPATH = bin/ .../ conf: bin/ .../ build/ classes: bin/ .../ lib/ antlr
-3.1.3.jar
```

在 CASSANDRA_HOME/bin/cassandra 脚本中，执行的流程如下：

1）配置 Cassandra 所依赖的 JAR 包。

CASSANDRA_HOME/bin/cassandra 脚本中的实现逻辑如下：

```
if [ "x $CASSANDRA_INCLUDE" = "x" ]; then
    # Locations (in order) to use when searching for an include file.
    for include in /usr/share/cassandra/cassandra.in.sh \
                   /usr/local/share/cassandra/cassandra.in.sh \
                   /opt/cassandra/cassandra.in.sh \
                   ~/.cassandra.in.sh \
                   'dirname $0'/cassandra.in.sh; do
        if [ -r $include ]; then
```

```
            . $include
            break
        fi
    done
# ...otherwise, source the specified include.
elif [ -r $CASSANDRA_INCLUDE ]; then
    . $CASSANDRA_INCLUDE
fi
```

将执行 CASSANDRA_HOME/bin 目录中的另一个 shell 脚本 cassandra. in. sh。在这个脚本中，将配置所有 CASSANDRA_HOME/lib/目录下的 JAR 包。cassandra. in. sh 脚本的实现逻辑如下：

```
if [ "x $CASSANDRA_HOME" = "x" ]; then
   CASSANDRA_HOME ='dirname $0'/...
fi

if [ "x $CASSANDRA_CONF" = "x" ]; then
   CASSANDRA_CONF = $CASSANDRA_HOME/conf
fi

cassandra_bin = $CASSANDRA_HOME/build/classes

CLASSPATH = $CASSANDRA_CONF: $cassandra_bin

for jar in $CASSANDRA_HOME/lib/*.jar; do
   CLASSPATH = $CLASSPATH: $jar
done
```

2）配置 Java 运行环境。

CASSANDRA_HOME/bin/cassandra 脚本中的实现逻辑如下：

```
# Use JAVA_HOME if set, otherwise look for java in PATH
if [ -x $JAVA_HOME/bin/java ]; then
    JAVA = $JAVA_HOME/bin/java
else
    JAVA ='which java'
fi
```

Cassandra 将从系统的环境变量 JAVA_HOME 中寻找 Java 的运行路径，如果找不到，将从系统路径中寻找。

3）启动 Cassandra 进程，并指定相关的参数。

首先，会运行 CASSANDRA_HOME/conf/cassandra-env. sh 脚本。

```
if [ -f "$CASSANDRA_CONF/cassandra-env.sh" ]; then
    . "$CASSANDRA_CONF/cassandra-env.sh"
fi
```

在 CASSANDRA_HOME/conf/cassandra-env. sh 脚本中，将自动计算启动 Cassandra 应使用的内存大小，默认为服务器内存的一半。

```
calculate_heap_size()
{
    case "'uname'" in
        Linux)
            system_memory_in_mb ='free-m |awk'/Mem:/{print $2}"
            MAX_HEAP_SIZE = $((system_memory_in_mb /2))M
            return 0
        ;;
        FreeBSD)
            system_memory_in_bytes ='sysctl hw.physmem |awk'{print $2}"
            MAX_HEAP_SIZE = $((system_memory_in_bytes /1024 /1024 /2))M
            return 0
        ;;
        *)
            MAX_HEAP_SIZE =1024M
            return 1
        ;;
    esac
}

# MAX_HEAP_SIZE = "4G"

if [ "x $MAX_HEAP_SIZE" = "x" ]; then
    calculate_heap_size
fi
```

然后设置运行 Java 应用程序的 JVM 参数。

```
# Specifies the default port over which Cassandra will be available for
# JMX connections.
    JMX_PORT = "8080"


    # Here we create the arguments that will get passed to the jvm when
    # starting cassandra.

    # enable assertions. disabling this in production will give a modest
    # performance benefit (around 5% ).
    JVM_OPTS = " $JVM_OPTS-ea"

    # enable thread priorities, primarily so we can give periodic tasks
    # a lower priority to avoid interfering with client workload
    JVM_OPTS = " $JVM_OPTS-XX: +UseThreadPriorities"
    # allows lowering thread priority without being root. see
    #  http:// tech.stolsvik.com/ 2010/ 01/ linux- java- thread- priorities-
workaround.html
    JVM_OPTS = " $JVM_OPTS-XX:ThreadPriorityPolicy =42"

    # min and max heap sizes should be set to the same value to avoid
    # stop-the-world GC pauses during resize, and so that we can lock the
    # heap in memory on startup to prevent any of it from being swapped
    # out.
    JVM_OPTS = " $JVM_OPTS-Xms $MAX_HEAP_SIZE"
    JVM_OPTS = " $JVM_OPTS-Xmx $MAX_HEAP_SIZE"
    JVM_OPTS = " $JVM_OPTS-XX: +HeapDumpOnOutOfMemoryError"
```

```
if [ "'uname'" = "Linux" ] ; then
    # reduce the per-thread stack size to minimize the impact of Thrift
    # thread-per-client. (Best practice is for client connections to
    # be pooled anyway.) Only do so on Linux where it is known to be
    # supported.
    JVM_OPTS = "$JVM_OPTS-Xss128k"
fi

# GC tuning options.
JVM_OPTS = "$JVM_OPTS-XX:+UseParNewGC"
JVM_OPTS = "$JVM_OPTS-XX:+UseConcMarkSweepGC"
JVM_OPTS = "$JVM_OPTS-XX:+CMSParallelRemarkEnabled"
JVM_OPTS = "$JVM_OPTS-XX:SurvivorRatio=8"
JVM_OPTS = "$JVM_OPTS-XX:MaxTenuringThreshold=1"
JVM_OPTS = "$JVM_OPTS-XX:CMSInitiatingOccupancyFraction=75"
JVM_OPTS = "$JVM_OPTS-XX:+UseCMSInitiatingOccupancyOnly"

# jmx: metrics and administration interface
JVM_OPTS = "$JVM_OPTS-Dcom.sun.management.jmxremote.port=$JMX_PORT"
JVM_OPTS = "$JVM_OPTS-Dcom.sun.management.jmxremote.ssl=false"
JVM_OPTS = "$JVM_OPTS-Dcom.sun.management.jmxremote.authenticate=false"
```

以上的 JVM 参数中包括了：JXM、GC、内存大小、线程的配置信息。详细可以参考 JVM 参数文档。

最后，在 CASSANDRA_HOME/conf/cassandra-env.sh 脚本中指定日志的运行配置文件（log4j）、程序运行的入口类（Main Class），并启动 Cassandra 进程。

```
cassandra_parms = "-Dlog4j.configuration=log4j-server.properties"
classname = "org.apache.cassandra.thrift.CassandraDaemon"
exec $JAVA $JVM_OPTS $cassandra_parms-cp $CLASSPATH $props $class <&-&
```

经过以上 3 个步骤，Cassandra 就完成了所有相关的环境配置，并进入了启动流程。

10.2　Cassandra 启动流程

在 10.1 节中已经了解到，Cassandra 默认的入口类为 org. apache. cassandra. thrift. Cassandra Daemon，其 Main 函数的实现如下：

```
String pidFile = System.getProperty("cassandra-pidfile");
try
{
    setup();

    if (pidFile != null)
    {
        new File(pidFile).deleteOnExit();
    }

    if (System.getProperty("cassandra-foreground") == null)
    {
        System.out.close();
```

```
            System.err.close();
        }

        start();
    } catch (Throwable e)
    {
        String msg = "Exception encountered during startup.";
        logger.error(msg, e);

        //try to warn user on stdout too, if we haven't already detached
        System.out.println(msg);
        e.printStackTrace();

        System.exit(3);
    }
```

这里有两个主要的功能函数：start 和 setup。start 函数启动 Thrift 服务，setup 函数将进行相关配置并启动 Cassandra 自身的相关线程。

10.2.1 配置 log4j

在启动 Cassandra 的参数中，启动脚本中指定了环境变量：-Dlog4j.configuration=log4j-server.properties，所以 Cassandra 会根据指定的 log4j 配置文件进行初始化。

10.2.2 读取校验配置文件信息

在 org.apache.cassandra.config.DatabaseDescriptor 的静态构造函数中，会读取配置文件，然后将配置文件中的每一个配置选项加载到 DatabaseDescriptor 的成员变量中。

如果存放数据文件的文件夹不存在，就创建数据文件夹。

```
    public static void createAllDirectories() throws IOException
    {
        try {
            if (conf.data_file_directories.length = =0)
            {
                throw new ConfigurationException("At least one DataFileDirectory must
be specified");
            }
            for ( String dataFileDirectory : conf.data_file_directories )
                FileUtils.createDirectory(dataFileDirectory);
            if (conf.commitlog_directory = =null)
            {
                throw new ConfigurationException("commitlog_directory must be speci-
fied");
            }

                FileUtils.createDirectory(conf.commitlog_directory);
            if (conf.saved_caches_directory = =null)
            {
                 throw new ConfigurationException("saved_caches_directory must be
specified");
            }
```

```
            FileUtils.createDirectory(conf.saved_caches_directory);
        }
        catch (ConfigurationException ex) {
            logger.error("Fatal error: " + ex.getMessage());
            System.err.println("Bad configuration; unable to start server");
            System.exit(1);
        }
    }
```

如果系统表（System Table）存在，则对配置信息和存储在系统表中的信息进行校验，否则将信息写入系统表中，供下次启动校验使用。

```
    public static void checkHealth() throws ConfigurationException, IOException
    {
        Table table = null;
        try
        {
            table = Table.open(Table.SYSTEM_TABLE);
        }
        catch (AssertionError err)
        {
            //this happens when a user switches from OPP to RP.
             ConfigurationException ex = new ConfigurationException("Could not read
system table. Did you change partitioners?");
            ex.initCause(err);
            throw ex;
        }
        SortedSet<ByteBuffer> cols = new TreeSet<ByteBuffer>(BytesType.instance);
        cols.add(PARTITIONER);
        cols.add(CLUSTERNAME);
        QueryFilter filter = QueryFilter.getNamesFilter(decorate(LOCATION_KEY), new
QueryPath(STATUS_CF), cols);
         ColumnFamily cf = table.getColumnFamilyStore(STATUS_CF).getColumnFamily
(filter);

        if (cf == null)
        {
            //this is either a brand new node (there will be no files), or the partitio-
ner was changed from RP to OPP.
             for (String path : DatabaseDescriptor.getAllDataFileLocationsForTable
("system"))
            {
                File[] dbContents = new File(path).listFiles(new FilenameFilter()
                {
                    public boolean accept(File dir, String name)
                    {
                        return name.endsWith(".db");
                    }
                });
                if (dbContents.length > 0)
                    throw new ConfigurationException("Found system table files, but
they couldn't be loaded. Did you change the partitioner?");
```

```
        }

        //no system files. this is a new node.
        RowMutation rm = new RowMutation(Table.SYSTEM_TABLE, LOCATION_KEY);
        cf = ColumnFamily.create(Table.SYSTEM_TABLE, SystemTable.STATUS_CF);
        cf.addColumn(new Column(PARTITIONER, ByteBuffer.wrap(Database-
Descriptor.getPartitioner().getClass().getName().getBytes(UTF_8)), FBUtili-
ties.timestampMicros()));
        cf.addColumn(new Column(CLUSTERNAME, ByteBuffer.wrap(DatabaseDescrip-
tor.getClusterName().getBytes()), FBUtilities.timestampMicros()));
        rm.add(cf);
        rm.apply();

        return;
    }


    IColumn partitionerCol = cf.getColumn(PARTITIONER);
    IColumn clusterCol = cf.getColumn(CLUSTERNAME);
    assert partitionerCol != null;
    assert clusterCol != null;
    if (!DatabaseDescriptor.getPartitioner().getClass().getName().equals(Byte-
BufferUtil.string(partitionerCol.value(), UTF_8)))
        throw new ConfigurationException("Detected partitioner mismatch! Did you
change the partitioner?");
        String savedClusterName = ByteBufferUtil.string(clusterCol.value(), UTF_8);
        if (!DatabaseDescriptor.getClusterName().equals(savedClusterName))
            throw new ConfigurationException("Saved cluster name " + savedCluster-
Name + " != configured name " + DatabaseDescriptor.getClusterName());
    }
```

在校验的过程中，Cassandra 将从系统表中取得集群名称（Cluster Name）和分区器（Partitioner），然后将这两项与配置文件中的值进行比较，如果发现不一致，说明配置文件与之前运行的设置不符，Cassandra 不予启动。

Cassandra 0.6.x 的版本中，启动的时候会从配置文件中加载 Schema 信息，而 Cassandra 0.7.x 的版本中，启动的时候不会从配置文件中加载任何用户自己定义的 Schema 信息。

10.2.3 加载所有的数据文件

首先，Cassandra 会定位每一个 Keyspace 所在的数据文件夹和缓存数据文件夹，删除临时的、不完整的、已经成功执行的数据压缩文件。

```
public static void scrubDataDirectories(String table, String columnFamily)
{
    for (Map.Entry < Descriptor, Set < Component > > sstableFiles : files(table,
columnFamily, true).entrySet())
    {
        Descriptor desc = sstableFiles.getKey();
        Set < Component > components = sstableFiles.getValue();

        if (SSTable.conditionalDelete(desc, components))
            //was compacted or temporary: deleted.
```

```
                continue;

            File dataFile = new File(desc.filenameFor(Component.DATA));
            if (components.contains(Component.DATA) && dataFile.length() > 0)
                //everything appears to be in order... moving on.
                continue;

            //missing the DATA file! all components are orphaned
            logger.warn("Removing orphans for {}: {}", desc, components);
            for (Component component : components)
            {
                try
                {

            FileUtils.deleteWithConfirm(desc.filenameFor(component));
                }
                catch (IOException e)
                {
                    throw new IOError(e);
                }
            }
        }

        //cleanup incomplete saved caches
        Pattern tmpCacheFilePattern = Pattern.compile(table + "-" + columnFamily + "-
(Key|Row)Cache.* \.tmp$");
        File dir = new File(DatabaseDescriptor.getSavedCachesLocation());

        if (dir.exists())
        {
            assert dir.isDirectory();
            for (File file : dir.listFiles())
                if (tmpCacheFilePattern.matcher(file.getName()).matches())
                    if (!file.delete())
                        logger.warn("could not delete " + file.getAbsolutePath());
        }
    }
```

接下来，Cassandra 将加载所有的 Keyspace 中的数据文件，这包括读取 SSTable 文件中的 Filter 文件、内存索引 Index 文件，加载数据缓存文件，获取二级索引信息。

```
private ColumnFamilyStore(Table table, String columnFamilyName, IPartitioner
partitioner, int generation, CFMetaData metadata)
{
    ssTables = new SSTableTracker(table.name, columnFamilyName);
    Set<DecoratedKey> savedKeys = readSavedCache(DatabaseDescriptor.getSeri-
alizedKeyCachePath(table.name, columnFamilyName));
    logger.info("read " + savedKeys.size() + " from saved key cache");
    List<SSTableReader> sstables = new ArrayList<SSTableReader>();
    for (Map.Entry<Descriptor,Set<Component>> sstableFiles : files(table.name,
columnFamilyName, false).entrySet())
    {
        SSTableReader sstable;
        try
```

```
        {
                sstable = SSTableReader.open(sstableFiles.getKey(), sstableFiles.
getValue(), savedKeys, ssTables, metadata, this.partitioner);
        }
        catch (FileNotFoundException ex)
        {
             logger.error("Missing sstable component in " + sstableFiles + ";
skipped because of " + ex.getMessage());
            continue;
        }
        catch (IOException ex)
        {
            logger.error("Corrupt sstable " + sstableFiles + "; skipped", ex);
            continue;
        }
        sstables.add(sstable);
    }
    ssTables.add(sstables);

    //create the private ColumnFamilyStores for the secondary column indexes
    indexedColumns = new ConcurrentSkipListMap < ByteBuffer, ColumnFamilyStore >
(getComparator());
    for (ColumnDefinition info : metadata.column_metadata.values())
    {
        if (info.index_type != null)
            addIndex(info);
    }
}
```

10.2.4 修复数据

Cassandra 启动时的数据修改分为 3 个环节：读取 Commitlog、压缩数据、应用变更的元数据信息（Schema）。下面分别予以介绍。

1）读取 Commitlog，恢复没有持久化到 SSTable 中的数据。

```
public static void recover() throws IOException
{
    String directory = DatabaseDescriptor.getCommitLogLocation();
    File[] files = new File(directory).listFiles(new FilenameFilter()
    {
        public boolean accept(File dir, String name)
        {
                return CommitLogSegment.possibleCommitLogFile(name) && !in-
stance.manages(name);
        }
    });
    if (files.length == 0)
    {
        logger.info("No commitlog files found; skipping replay");
        return;
    }

    Arrays.sort(files, new FileUtils.FileComparator());
```

```
    logger.info("Replaying " + StringUtils.join(files, ", "));
    recover(files);
    for (File f : files)
    {

            FileUtils.delete  ( CommitLogHeader.getHeaderPathFromSegmentPath
(f.getAbsolutePath())); //may not actually exist
        if (!f.delete())
            logger.error("Unable to remove " + f + "; you should remove it manually
or next restart will replay it again (harmless, but time-consuming)");
    }
    logger.info("Log replay complete");
}
```

2）根据数据压缩的规则，如果任意 Column Family 下的 SSTable 数量过多，则对其执行数据压缩操作。

```
public void checkAllColumnFamilies() throws IOException
{
    //perform estimates
    for (final ColumnFamilyStore cfs : ColumnFamilyStore.all())
    {
        Runnable runnable = new Runnable()
        {
            public void run ()
            {
                logger.debug("Estimating compactions for " + cfs.columnFamily);
                final Set < List < SSTableReader > > buckets = getBuckets(conve-
rtSSTablesToPairs(cfs.getSSTables()), 50L * 1024L * 1024L);
                updateEstimateFor(cfs, buckets);
            }
        };
        executor.submit(runnable);
    }

    for (ColumnFamilyStore cfs : ColumnFamilyStore.all())
    {
        submitMinorIfNeeded(cfs);
    }
}
```

3）应用变更的元数据信息，保证本服务器的元数据信息与集群中其他的服务器一致。

```
    UUID currentMigration = DatabaseDescriptor.getDefsVersion();
    UUID lastMigration = Migration.getLastMigrationId();
    if ((lastMigration != null) && (lastMigration.timestamp() > currentMigration.tim-
estamp()))
    {
        MigrationManager.applyMigrations(currentMigration, lastMigration);
    }
```

10.2.5　启动 Gossiper 服务

在 Cassandra 集群内部，节点之间的通信使用的 IP 地址和监听端口可以在配置文件中进

行配置。

```
storage_port: 7000
listen_address: localhost
```

如果 listen _address 选择留空，则 Cassandra 将监听本机 hostname 所属的 IP 地址。

在初始化 Gossiper 的过程中，先初始化 FailureDetector，然后将 Gossiper 注册到 FailureDetector 中，当 FailureDetector 发现有节点失效的时候，执行 Gossiper. convict 将 endPointStateMap_中的节点置为节点不可用（Not live）。同时初始化 StorageLoadBalancer，将 StorageLoadBalancer 以 IEndPointStateChangeSubscriber 的形式注册到 Gossiper 中，使得 StorageLoadBalancer 关注 onJoin、onLive 等事件，启动定时任务 LoadDisseminator，将本机的 Load 信息添加到 Gossiper 中。最后将 StorageService 以 IEndPointStateChangeSubscriber 的形式注册到 Gossiper 中，使得 StorageService 关注 onJoin、onLive 等事件。

启动 Gossiper 服务时，将加载配置文件中的 seeds 信息和系统表中的集群节点信息，最后启动定时任务 GossipTimerTask，开始整个 Gossiper 的消息传播。

```
if (Boolean.valueOf(System.getProperty("cassandra.load_ring_state", "true")))
{
    logger_.info("Loading persisted ring state");
    for (Map.Entry<Token, InetAddress> entry : SystemTable.loadTokens().entry-
Set())
    {
        tokenMetadata_.updateNormalToken(entry.getKey(), entry.getValue());
        Gossiper.instance.addSavedEndpoint(entry.getValue());
    }
}

logger_.info("Starting up server gossip");

//have to start the gossip service before we can see any info on other nodes. this
is necessary
//for bootstrap to get the load info it needs.
// (we won't be part of the storage ring though until we add a nodeId to our state,
below.)
Gossiper.instance.register(this);
Gossiper.instance.register(migrationManager);
Gossiper.instance.start(FBUtilities.getLocalAddress(), SystemTable.increment-
AndGetGeneration()); //needed for node-ring gathering.
```

10.2.6 判断是否需要进行 Bootstrap 操作

在 Cassandra 集群中，如果有新的服务器希望加入已有的集群中，可以执行 Bootstrap 操作。新的服务器通过执行 Bootstrap 操作可以使得本服务器从已经集群的一致性哈希环中获取一个新的 Token，并且能够从已有集群中获取到本服务器所对应的数据。

新增服务器的 Token 值可以直接在配置文件中指定。

```
initial_token:
```

如果留空，将由 Partitioner 计算获得。

新增服务器获取 Token 的逻辑如下：

```
    public static Token getBootstrapToken(final TokenMetadata metadata, final Map <
InetAddress, Double > load) throws IOException, ConfigurationException
    {
        if (DatabaseDescriptor.getInitialToken() != null)
        {
                logger.debug ( " token manually specified as " + DatabaseDescrip-
tor.getInitialToken());
              Token token = StorageService.getPartitioner().getTokenFactory(). from-
String(DatabaseDescriptor.getInitialToken());
            if (metadata.getEndpoint(token) != null)
                throw new ConfigurationException("Bootstraping to existing token " +
token + " is not allowed (decommission/removetoken the old node first).");
            return token;
        }

        return getBalancedToken(metadata, load);
    }
```

如果需要进行 Bootstrap 操作，需要符合以下 3 个条件：

1）配置文件中的 auto_bootstrap 参数设置为 true。

```
auto_bootstrap: true
```

2）该服务器之前没有进行过 Bootstrap 操作，这个记录是存储在系统表中的。如果已经进行过 Bootstrap 操作，但是想再次进行，只能将系统表删除。

3）配置文件中的 seed 参数中不包含本服务。

满足上面 3 个条件，Cassandra 在启动的时候便会执行 Bootstrap 操作。

10.2.7　监听 Thrift 端口，提供 Thrift 服务

在配置文件中指定 Cassandra 对外服务监听的 IP 地址和端口号。启动 Cassandra 程序后，将对外提供 Thrift 服务。

服务监听的 IP 地址和端口号可以在配置文件中进行配置。

```
rpc_address: localhost
rpc_port: 9160
```

如果 rpc_address 选择留空，则 Cassandra 将监听本机 hostname 所属的 IP 地址。

10.3　本章小结

本章分析和讲解了 Cassandra 的整个启动流程，介绍了 Bootstrap 操作和启动相关的参数。

第 5 ～ 10 章系统概述了整个 Cassandra 内部的工作机制和流程。从第 11 章开始，我们将讲解 Cassandra 在实际环境中的高级应用。

第 11 章

在分布式环境中使用的 Cassandra

本章内容

- 在 Linux 环境中搭建与使用 Cassandra 集群
- Cassandra 运行配置项详解
- Cassandra 集群的运行和维护
- 本章小结

本书在第2章讲解了如何快速地在单机中使用Cassandra，本章将讲解如何在Linux服务器上搭建分布式的Cassandra集群，详细解释配置文件以及如何在生产环境中对Cassandra进行维护。

11.1 在Linux环境中搭建与使用Cassandra集群

在Linux系统中搭建Cassandra集群非常简单，只需要在每一台服务器上执行如下4个步骤即可：

1）配置JRE。

2）部署Cassandra可执行文件。

3）修改Cassandra配置文件。

4）启动Cassandra。

下面我们将在3台Linux服务器上演示如何搭建Cassandra集群。

演示用的3台服务器的配置如下：

- OS：Red Hat 5.3
- RAM：16GB
- CPU：Intel（R）Xeon（R）CPU 5110

3台服务器的主机名（hostname）分别为：hadoop12、hadoop13、hadoop14。使用的用户名（username）都是admin。服务器与服务器之间可以互相PING通（网络是互联的）。首先，在SUN的官网（http://www.java.com/en/download/manual.jsp）下载相应版本的JRE：Linux x64（如果是32位的服务器，可以下载Linux（self-extracting file）），然后在Cassandra的官网[⊖]下载相应版本的Cassandra可执行包：apache-cassandra-0.7.0-beta3-bin.tar.gz。

下载后在服务器HOME目录（/home/admin）下执行“ls -l”命令，可以看到如下结果：

```
[admin@hadoop12 ~]$ ls -l
total 74264
-rw-rw-r--1 admin admin 10786368 Nov 8 13:29 apache-cassandra-0.7.0-beta3-bin.tar.gz
-rw-rw-r--1 admin admin 20590332 Nov 8 18:34 jre-6u22-linux-x64.bin
```

11.1.1 配置JRE

从网上下载的jre-6u22-linux-x64.bin文件不具备可执行权限，所以首先需要给jre-6u22-linux-x64.bin文件赋权。

```
[admin@hadoop12 ~]$ chmod 777 jre-6u22-linux-x64.bin
```

⊖ 官网地址为：http://cassandra.apache.org/download/。

赋权后再次执行“ls -l”命令查看结果。

```
[admin@ hadoop12 ~]$ ls -l
total 74264
-rw-rw-r-- 1 admin admin 10786368 Nov 8 13:29 apache-cassandra-0.7.0-beta3
-bin.tar.gz
-rwxrwxrwx 1 admin admin 20590332 Nov 8 18:34 jre-6u22-linux-x64.bin
```

这次可以看到，jre-6u22-linux-x64.bin 文件的已经具备了执行的权限（-rwxrwxrwx）。

执行 jre-6u22-linux-x64.bin 文件的解压操作如下：

```
[admin@ hadoop12 ~]$ ./jre-6u22-linux-x64.bin
```

解压完成后，可以看到多了一个解压后的文件夹（jre1.6.0_22）。

```
[admin@ hadoop12 ~]$ ls -l
total 74268
-rw-rw-r-- 1 admin admin 10786368 Nov 8 13:29 apache-cassandra-0.7.0-beta3
-bin.tar.gz
drwxr-xr-x 8 admin admin 4096 Nov 10 21:22 jre1.6.0_22
-rwxrwxrwx 1 admin admin 20590332 Nov 8 18:34 jre-6u22-linux-x64.bin
```

然后设置 Java 的运行环境变量，打开 ~/.bashrc 文件。

```
[admin@ hadoop12 jre1.6.0_22]$ vi ~/.bash_profile
```

在 bashrc 文件底部添加如下内容：

```
JAVA_HOME = /home/admin/jre1.6.0_22
export JAVA_HOME
PATH = $JAVA_HOME/bin:$PATH
export PATH
```

添加完成之后，执行以下命令让系统应用该设置：

```
. ~/.bashrc
```

运行完成之后，就可以直接使用 Java。进行校验如下：

```
[admin@ hadoop12 jre1.6.0_22]$ java -version
java version "1.6.0_22"
Java(TM) SE Runtime Environment (build 1.6.0_22-b04)
Java HotSpot(TM) 64-Bit Server VM (build 17.1-b03, mixed mode)
```

这样，JRE 就算配置完毕了。

11.1.2　部署 Cassandra 可执行文件

首先需要对 apache-cassandra-0.7.0-beta3-bin.tar.gz 文件进行解压。

```
[admin@ hadoop12 ~]$ tar -xzf apache-cassandra-0.7.0-beta3-bin.tar.gz
```

解压完成后，解压的文件都在文件夹 apache-cassandra-0.7.0-beta3 中。

```
[admin@ hadoop12 ~]$ ls -l
total 74272
drwxrwxr-x 7 admin admin 4096 Nov 11 10:20 apache-cassandra-0.7.0-beta3
-rw-rw-r-- 1 admin admin 10786368 Nov 8 13:29 apache-cassandra-0.7.0-beta3
-bin.tar.gz
drwxr-xr-x 8 admin admin 4096 Nov 10 21:22 jre1.6.0_22
-rwxrwxrwx 1 admin admin 20590332 Nov 8 18:34 jre-6u22-linux-x64.bin
```

这样，Cassandra 可执行文件就部署成功了。

11.1.3 修改 Cassandra 配置文件

进入到 Cassandra 的配置文件目录。

```
cd /home/admin/apache-cassandra-0.7.0-beta3/conf
```

首先修改运行 Cassandra 的程序日志（log4j）的位置。

```
[admin@ hadoop12 conf]$ vi log4j-server.properties
```

将以下内容

```
log4j.appender.R.File=/var/log/cassandra/system.log
```

修改为

```
log4j.appender.R.File=/home/admin/apache-cassandra-0.7.0-beta3/system.log
```

修改后，Cassandra 程序日志就会记录在/home/admin/apache-cassandra-0.7.0-beta3/system.log 中。

接下来，修改启动 Cassandra 的环境参数。

```
[admin@ hadoop12 conf]$ vi cassandra-env.sh
```

将以下内容

```
# MAX_HEAP_SIZE="4G"
```

修改为

```
MAX_HEAP_SIZE="4G"
```

这样，Cassandra 进程在运行的时候就可以使用 4GB 的内存了。注意，这个参数需要根据服务器的实际内存大小来定。假设服务器可以使用的内存只有 2GB，这个参数就不能大于 2GB。

最后，修改 Cassandra 的运行参数。

```
[admin@ hadoop12 conf] $ vi cassandra.yaml
```

修改集群的名称。

```
cluster_name: 'MyCluster'
```

开启新节点加入集群数据自动分配参数。

```
auto_bootstrap: true
```

修改 Cassandra 数据的存储位置。

```
data_file_directories:
    - /home/admin/apache - cassandra - 0.7.0 - beta3 /data
```

修改 Cassandra commitlog 的存储位置。

```
commitlog_directory: /home/admin/apache - cassandra - 0.7.0 - beta3 /commitlog
```

修改 Cassandra Cache 的存储位置。

```
 saved_caches_directory: /home/admin/apache - cassandra - 0.7.0 - beta3 /saved
_caches
```

修改 seed 节点。

```
seeds:
    - hadoop12
    - hadoop13
```

修改使用虚拟内存的方式。

```
disk_access_mode: mmap_index_only
```

修改 Cassandra 集群内部监听地址。

```
listen_address:
```

修改 Cassandra 提供对外服务监听地址。

```
rpc_address:
```

修改远程调用超时时间。

```
rpc_timeout_in_ms: 30000
```

这样，Cassandra 的运行参数就配置完毕了。

11.1.4　启动 Cassandra

完成以上的配置后，就可以启动 Cassandra 了。执行以下命令：

```
[admin@ hadoop12 ~]$ sh ~/apache-cassandra-0.7.0-beta3/bin/Cassandra
```

执行该命令后，会看到如下输出，这说明启动成功了：

```
[admin@ hadoop12 ~]$ INFO 11:28:21,233 Heap size: 4286251008/4286251008
INFO 11:28:21,251 JNA not found. Native methods will be disabled.
INFO 11:28:21,263 Loading settings from file:/home/admin/apache-cassandra-
0.7.0-beta3/conf/cassandra.yaml
INFO 11:28:21,440 DiskAccessMode is standard, indexAccessMode is mmap
INFO 11:28:21,537 Creating new commitlog segment /home/admin/apache-cassandra
-0.7.0-beta3/commitlog/CommitLog-1289446101537.log
INFO 11:28:21,611 read 0 from saved key cache
INFO 11:28:21,615 read 0 from saved key cache
INFO 11:28:21,617 read 0 from saved key cache
INFO 11:28:21,618 read 0 from saved key cache
INFO 11:28:21,620 read 0 from saved key cache
INFO 11:28:21,623 loading row cache for LocationInfo of system
INFO 11:28:21,624 completed loading (1 ms; 0 keys) row cache for LocationInfo
of system
INFO 11:28:21,625 loading row cache for HintsColumnFamily of system
INFO 11:28:21,625 completed loading (0 ms; 0 keys) row cache for HintsColumnFamily
of system
INFO 11:28:21,626 loading row cache for Migrations of system
INFO 11:28:21,626 completed loading (0 ms; 0 keys) row cache for Migrations of sys-
tem
INFO 11:28:21,629 loading row cache for Schema of system
INFO 11:28:21,630 completed loading (1 ms; 0 keys) row cache for Schema of system
INFO 11:28:21,630 loading row cache for IndexInfo of system
INFO 11:28:21,631 completed loading (1 ms; 0 keys) row cache for IndexInfo of sys-
tem
INFO 11:28:21,669 Couldn't detect any schema definitions in local storage.
INFO 11:28:21,670 Found table data in data directories. Consider using JMX to call
org.apache.cassandra.service.StorageService.loadSchemaFromYaml().
INFO 11:28:21,674 No commitlog files found; skipping replay
INFO 11:28:21,700 Upgrading to 0.7. Purging hints if there are any. Old hints will
be snapshotted.
INFO 11:28:21,705 Cassandra version: 0.7.0-beta3
INFO 11:28:21,706 Thrift API version: 19.4.0
INFO 11:28:21,716 Loading persisted ring state
INFO 11:28:21,718 Starting up server gossip
INFO 11:28:21,728 switching in a fresh Memtable for LocationInfo at CommitLogCon-
text(file='/home/admin/apache-cassandra-0.7.0-beta3/commitlog/CommitLog-
1289446101537.log', position=700)
INFO 11:28:21,729 Enqueuing flush of Memtable-LocationInfo@1930362236(224
bytes, 4 operations)
INFO 11:28:21,730 Writing Memtable-LocationInfo@1930362236(224 bytes, 4 opera-
```

```
tions)
    INFO 11:28:21,931 Completed flushing /home/admin/apache-cassandra-0.7.0-be-
ta3/data/system/LocationInfo-e-1-Data.db
    INFO 11:28:21,984 This node will not auto bootstrap because it is configured to be
a seed node.
    WARN 11:28:21,998 Generated random token 116791188121120886117513664065968576037. Ran-
dom tokens will result in an unbalanced ring; see http://wiki.apache.org/cassandra/Oper-
ations
    INFO 11:28:21,999 switching in a fresh Memtable for LocationInfo at CommitLogCon-
text(file='/home/admin/apache-cassandra-0.7.0-beta3/commitlog/CommitLog-
1289446101537.log', position=848)
    INFO 11:28:22,000 Enqueuing flush of Memtable-LocationInfo@836293175(36 bytes,
1 operations)
    INFO 11:28:22,001 Writing Memtable-LocationInfo@836293175(36 bytes, 1 opera-
tions)
    INFO 11:28:22,120 Completed flushing /home/admin/apache-cassandra-0.7.0-be-
ta3/data/system/LocationInfo-e-2-Data.db
    INFO 11:28:22,125 Will not load MX4J, mx4j-tools.jar is not in the classpath
    INFO 11:28:22,163 Binding thrift service to hadoop12.hst.ali.dw.alidc.net/
172.16.197.112:9160
    INFO 11:28:22,166 Using TFramedTransport with a max frame size of 15728640 bytes.
```

现在可以通过 Cassandra 提供的监控工具查看集群的运行情况。

```
[admin@hadoop12 ~]$sh ~/apache-cassandra-0.7.0-beta3/bin/nodetool -h ha-
doop12 -p 8080 ring
    Address      Status State   Load      Token
    172.16.197.112 Up Normal 10.27 KB 116791188121120886117513664065968576037
```

可以看到，目前集群中只有一台服务器。下一步，看看如何构建一个 Cassandra 集群。

首先，按照上面的步骤，配置好 hadoop13 和 hadoop14 这两台机器，然后，在 hadoop13 这台服务器上执行以下命令启动 Cassandra 进程：

```
[admin@hadoop13 ~]$sh apache-cassandra-0.7.0-beta3/bin/cassandra
```

启动后的输出信息与第一台 Cassandra 启动的输出基本类似，区别如下：

```
[admin@hadoop13 ~]$ INFO 14:29:44,163 Heap size: 4286251008/4286251008
...
    INFO 14:29:45,361 Binding thrift service to hadoop13.hst.ali.dw.alidc.net/
172.16.197.113:9160
    INFO 14:29:45,362 InetAddress /172.16.197.112 is now UP
    INFO 14:29:45,362 Started hinted handoff for endpoint /172.16.197.112
    INFO 14:29:45,365 Finished hinted handoff of 0 rows to endpoint /172.16.197.112
...
```

可以看到，在启动第二台 Cassandra 服务器的时候，检测到了第一台 Cassandra 服务器：

InetAddress / 172. 16. 197. 112 is now UP。

再通过 Cassandra 提供的监控工具查看集群的运行情况。

```
[admin@ hadoop12 conf] $sh ~/apache - cassandra - 0.7.0 - beta3/bin/nodetool - h ha-
doop12 - p 8080 ring
Address    Status State  Load    Token
                                  116791188121120886117513664065968576037
172.16.197.113 Up  Normal 10.27 KB  76532511552026171774714881963084655022
172.16.197.112 Up  Normal 15.24 KB  116791188121120886117513664065968576037
```

现在可以看到，Cassandra 集群中已经有两台服务器在提供服务了。

这样就完成了在 Linux 环境中 Cassandra 集群的搭建。

11.2 Cassandra 运行配置项详解

Cassandra 0. 7. x 中所有的运行配置都在配置文件 cassandra. yaml 中进行配置。

YAML 的全称是：YAML Ain 't Markup Language（YAML 并不是标记语言），而 Cassandra 0. 7 之前的版本所采用配置文件格式为 XML，属于标记语言。

YAML 具备可读性强、数据序列化简单、适合跨编程语言开发等优点。如果希望了解更多关于 YAML 语言的规范与特点，可以参考官方文档[⊖]。

下面详细讲解 Cassandra 中的各个配置项的功能与含义。

1. cluster_name

该配置项用于设置 Cassandra 集群的名称。在 Cassandra 集群中，每一台服务器都必须具备相应的集群的名称。如果名称不一致，则当前 Cassandra 服务器无法加入集群。

用户可以通过 Thrift API 获取 Cassandra 集群的名称，接口方法为：string describe_cluster_name()。

2. initial_token

Cassandra 服务器的初始化 Token 值，这个值代表了 Cassandra 服务器在一致性哈希圆环中的位置。

当 Cassandra 第一次启动的时候，会从该配置项中读取，如果留空，将随机生成一个 Token 值。如果 Cassandra 不是第一次启动，将从系统表中读取该 Token 值。

选择合适的 Token 值能够将数据很好地平均分布到整个集群之中。

如果不知道数据的分布情况，可以将这个选项留空，Cassandra 将会自动生成一个随机的 Token 值。当集群中的数据量增多以后，就可以发现数据的分布规律，然后更加合理地

⊖ 链接为：http://www. yaml. org/spec/1. 2/spec. html。

手动指定 Token 值。

3. auto_bootstrap

第一次启动的时候，确定是否在加入 Cassandra 集群时从其他服务器获取属于本服务器的数据。如果当前 Cassandra 服务器不在 seed 配置选项中，并且是第一次启动，将从 Cassandra 集群中的其他服务器获取属于本服务器的数据。

在集群中，我们都需要新加入集群中的服务器能够自动获取相应的数据，所以可以将这个选项设置为 true。

4. hinted_handoff_enabled

是否开启当前 Cassandra 服务器的 HINT 操作。

如果开启该功能，Cassandra 服务器将缓存发送给暂时失效的其他 Cassandra 服务器的数据，等待失效的服务器恢复后，再将缓存的数据发送给恢复的服务器。

在实际的生成环境中，如果数据量特别大，读写的频率也非常高，并且数据的一致性不是特别重要的情况下，可以考虑将设置选择设置为 false，这样有利于提高系统的磁盘利用率和降低系统的负载。虽然关闭了 HINT 功能，Cassandra 集群还会有其他的机制来修改丢失的数据，如读修复（Read Repair）。

如果数据的一致性是非常重要的，则建议将这个选项设置为 true。

5. authenticator

验证使用 Cassandra 的用户是否合法，这是安全认证的第一步。

Cassandra 中定义了一系列验证用户的策略，可以选择的项如下：

- org. apache. cassandra. auth. AllowAllAuthenticator，表明所有的用户都是合法的。
- org. apache. cassandra. auth. SimpleAuthenticator，表明合法的用户和对应的密码都在 passwd. properties 文件中定义。

在进行 Cassandra 集群测试或者只有自己使用的时候，为简单起见，可以将该选项设置为 org. apache. cassandra. auth. AllowAllAuthenticator，这样就关闭了安全认证功能。

但是如果将 Cassandra 集群作为多个应用的公共存储，那么还是建议开启安全认证功能，将该选项设置为 org. apache. cassandra. auth. SimpleAuthenticator，并且在 passwd. properties 文件中定义可以使用 Cassandra 集群用户的用户名和密码信息，避免被其他人使用。

6. authority

验证该用户是否具备操作某一个 ColumnFamily 的权限，这是安全认证的第二步。

Cassandra 中定义了一系列验证用户权限的策略，可以选择的项为：

- org. apache. cassandra. auth. AllowAllAuthority，表明所有的用户具备所有的权限。
- org. apache. cassandra. auth. SimpleAuthority，表明合法的用户和对应的权限都在 ac-

cess. properties 文件中定义。

在进行 Cassandra 集群测试或者只有自己使用的时候，为简单起见，可以将该选项设置为 org. apache. cassandra. auth. AllowAllAuthority，这样就关闭了安全认证功能。

但是如果将 Cassandra 集群作为多个应用的公共存储，那么还是建议开启安全认证功能，将该选项设置为 org. apache. cassandra. auth. SimpleAuthority，并且在 access. properties 文件中定义使用 Cassandra 集群的用户的权限。

注意：authenticator 和 authority 是有区别的。authenticator 定义了允许哪些用户访问 Cassandra 集群，而 authority 定义了允许每一个用户访问哪些 Keyspace 和 ColumnFamily。

7. partitioner

Cassandra 集群中数据分区的策略。同一个 Cassandra 集群中的每一台服务器中的该配置应该一致。

Cassandra 中定义了一系列数据分区的策略，可以选择的项如下：

- org. apache. cassandra. dht. RandomPartitioner
- org. apache. cassandra. dht. ByteOrderedPartitioner
- org. apache. cassandra. dht. OrderPreservingPartitioner
- org. apache. cassandra. dht. CollatingOrderPreservingPartitioner

各个数据分区的策略的详细解释与区别可以参考第 5 章的内容。

8. data_file_directories

SSTable 文件在磁盘中的存储位置。

这个选项可以设置多个值，即如果服务器具有多个磁盘，可以将这几个磁盘都指定为存储 SSTable 文件的位置。

在生产环境中需要将 data_file_directories、saved_caches_directory 和 commitlog_directory 设置在不同的磁盘中，这样有利于分散系统的磁盘 I/O 的压力，使得系统获得更好的整体性能。

选择给 Cassandra 使用存放数据的磁盘最好是给 Cassandra 单独使用。比如某一个磁盘既存储 MySQL 的数据文件同时也存储 Cassandra 的数据文件，这样就不大适合。

9. commitlog_directory

commitlog 文件在磁盘中的存储位置。

10. saved_caches_directory

数据缓存文件在磁盘中的存储位置。

11. commitlog_rotation_threshold_in_mb

每一个 commitlog 文件的大小。可以直接使用默认配置。

12. commitlog_sync

记录 commitlog 的方式。可以选择的项如下：

- periodic，周期记录 commitlog，每一次有数据更新都将操作 commitlog。
- batch，批量记录 commitlog，每一段时间内数据的更新将批量一次性操作 commitlog。

在生产环境中，可以考虑使用 batch 的形式来记录 commitlog，这样可以提高系统的数据写入性能，但是系统在崩溃的时候，可能会有部分 commitlog 的数据丢失。这种数据丢失是允许的，因为 Cassandra 集群中还会有其他的节点写入成功，可以通过数据修复机制将这部分丢失的数据恢复。

13. commitlog_sync_period_in_ms

周期记录 commitlog 时，刷新 commitlog 文件的时间间隔。这个选项只有在 commitlog_sync = periodic 时才能设置。这个参数建议使用系统默认的值。

14. commitlog_sync_batch_window_in_ms

批量记录 commitlog 时，批量操作缓存的时间间隔。这个选项只有在 commitlog_sync = batch 时才能设置。

这个参数设置得越大，一次性批量写入的 commitlog 数据就会越多，系统整体写入性能也会更佳。但是不是越大越好，建议使用 10ms。

15. seeds

Cassandra 集群中的种子节点地址。

这个选项可以设置多个值，即 Cassandra 集群中有多个种子节点。

集群中所有的服务器在启动的时候，都将与 seed 节点进行通信，从而获取集群的相关信息。如果某一台服务器被设置为 seed 节点，那么在启动的时候将自动加入集群，并且不会执行 Bootstrap 的操作，即无法从集群的其他节点中获取相应的数据。

在生产环境中配置这个参数的时候，建议先考虑好集群的规模。比如打算使用 30 台服务器做 Cassandra 集群，那么将其中的 10 台机器的主机名配置在 seed 参数中，作为整个集群的种子节点。在后期的集群维护中，如果要卸载机器撤出集群，不要选择 seed 参数中的机器。

16. disk_access_mode

Cassandra 访问 SSTable 文件中的 Data 文件和 Index 文件时，是否使用虚拟内存映射的形式。可以选择的项如下：

- auto，自动选择合适的文件访问形式，如果是 64 位系统，则为 mmap 形式，否则为 standard 形式。

- mmap，访问 SSTable 文件中的 Data 文件和 Index 文件时，都采用虚拟内存映射的形式。要求服务器是 64 位系统。
- mmap_index_only，访问 SSTable 文件中的 Index 文件时，采用虚拟内存映射的形式。要求服务器是 64 位系统。
- standard，访问 SSTable 文件中的 Data 文件和 Index 文件时，都不采用虚拟内存映射的形式。

使用虚拟内存映射的形式访问文件能够加快对文件的读写速度，但是这是以消耗额外的内存作为代价的。所以要根据实际内存大小与文件大小来选择合适的文件访问方式。在生产环境中，可以使用 mmap_index_only 的形式，保证索引的高效使用，同时又不至于占用大量的虚拟内存空间。

17. concurrent_reads

并发读取的线程数。
这个选项设置得越大，Cassandra 在进行读取操作时可以使用的线程数就越多。
推荐的设置为：CPU 的个数 ×2。假设 Cassandra 服务器是 8 核的 CPU，那么可以将并发读取的参数设置为 16。

18. concurrent_writes

并发写入的线程数。
这个选项设置得越大，Cassandra 在进行写入操作时可以使用的线程数就越多。
这个参数可以根据每一个 Cassandra 服务的 Cassandra 客户端的数量来设置。一般来说，可以将这个参数设置为 32。

19. memtable_flush_writers

Memtable 中的数据写入磁盘成为 SSTable 文件的并发数。
这个选项使用默认配置即可，默认配置为 data_file_directories 中指定的目录的个数。

20. sliced_buffer_size_in_kb

进行范围读取操作时，读取 SSTable 文件使用的缓存大小。
这个选项使用默认配置即可。

21. storage_port

Cassandra 集群中服务器与服务器之间相互通信的端口号。这个参数在集群中所有的服务器中都必须一致。

22. listen_address

Cassandra 集群中服务器与服务器之间相互通信的地址。如果留空，将默认使用服务器

的机器名。

23. rpc_address

Cassandra 服务器对外提供服务的地址。如果留空，将默认使用服务器的机器名。

24. rpc_port

Cassandra 服务器对外提供服务的端口号。建议这个参数在集群中所有的服务器中保持一致。

25. rpc_keepalive

Cassandra 服务器对外提供服务连接是否使用长连接。
这个选项使用默认配置即可。

26. thrift_framed_transport_size_in_mb

使用 Thrift Frame 每次传递的数据大小。如果该选项为 0，则禁用 Thrift Frame 功能。
这个选项使用默认配置即可。在 Cassandra 0.7.x 中默认开启 Frame 功能。

27. thrift_max_message_length_in_mb

使用 Thrift 传递的数据最大值。
这个选项使用默认配置即可。

28. snapshot_before_compaction

Cassandra 在执行数据压缩操作前，是否对需要压缩的 SSTable 文件做数据快照（snapshot）。
这个选项使用默认配置即可。

29. binary_memtable_throughput_in_mb

数据批量导入的时候，binary memtable 中缓存数据大小。
这个选项一般只有在 Cassandra 集群第一次、初始化数据的时候才会使用。
这个选项使用默认配置即可。

30. column_index_size_in_kb

SSTable 文件中的 Data 文件对应 Column 索引的数据大小间隔。这个值越小，在 Column 索引中找到相应的值的速度就越快，但是会消耗更多的内存与磁盘空间。
这个选项使用默认配置即可。

31. in_memory_compaction_limit_in_mb

在 Cassandra 执行数据压缩时，如果某个 Key 对应的数据的大小超过了 in_memory_compaction_limit_in_mb 的限制，将采用延后压缩的机制进行压缩，避免使用过多的内存。

这个选项使用默认配置即可。如果机器的内存比较小，可以适当将这个参数调小。

32. rpc_timeout_in_ms

Cassandra 服务器在处理外部请求的时候，如果某一个请求执行的时间超过了 rpc_timeout_in_ms 的限制，将抛出超时异常给调用的客户端。

Cassandra 集群在实际的使用中，如果读写压力过大，很容易出现超时异常。原因是 Cassandra 服务器内部有太多的读写请求需要处理，无法在指定的时间内进行响应。对于这种问题，需要考虑应用的设计是否合理，能否再优化，或者是为 Cassandra 集群增加新的机器，以提高读写性能。

如果不是实时性非常高并且不是对外提供服务的应用，可以适当将这个参数调节成 30 000ms。

33. endpoint_snitch

Cassandra 集群中网络的选择策略。

Cassandra 中定义了一系列网络的选择策略，可以选择的项如下：

- org. apache. cassandra. locator. SimpleSnitch
- org. apache. cassandra. locator. RackInferringSnitch
- org. apache. cassandra. locator. PropertyFileSnitch

各个网络的选择策略的详细解释与区别可以参考第 5 章的内容。

34. dynamic_snitch

是否启用动态的节点选择策略。启动该选项可以有效地避免响应缓慢的节点。

和这个选项相关的其他选项如下：

- dynamic_snitch_update_interval_in_ms
- dynamic_snitch_reset_interval_in_ms
- dynamic_snitch_badness_threshold

和 dynamic_snitch 相关的信息可以参考第 5 章的内容。

35. request_scheduler

设置资源调度分配策略。

Cassandra 中定义了一系列网络的选择策略，可以选择的项如下：

- org. apache. cassandra. scheduler. NoScheduler，所有的请求分配的计算资源都是均

等的。

- org. apache. cassandra. scheduler. RoundRobinScheduler，对不同的 Keyspace 分配不同的计算资源。

在多租户的情况下适合使用 RoundRobinScheduler。

一般来说，如果 Cassandra 集群没有作为一个收费的服务提供给第三方使用，那么使用 org. apache. cassandra. scheduler. NoScheduler 即可。使用 RoundRobinScheduler 将会根据不同的用户给予不同的计算资源。

36. index_interval

SSTable 文件中的 Index 文件对应内存索引的数据大小间隔。这个值越小，在内存索引中找到相应的值的速度就越快，但是会消耗更多的内存。

这个选项使用默认配置即可，如果机器的内存比较小，可以适当将这个参数调小。

37. keyspaces

定义 Keyspace 的属性。包括如下属性。

1）name：定义 Keyspace 的名称。

2）replica_placement_strategy：定义数据的备份策略，可选的项如下：

- org. apache. cassandra. locator. SimpleStrategy
- org. apache. cassandra. locator. OldNetworkTopologyStrategy
- org. apache. cassandra. locator. NetworkTopologyStrategy
- org. apache. cassandra. locator. LocalStrategy

和数据的备份策略相关的信息可以参考第 5 章的内容。

3）replication_factor：定义数据的备份数。

4）column_families：定义 ColumnFamily 的属性。

5）column_type：定义 ColumnFamily 的类型。可以设置为 Super 或者 Standard，如果不设置，为 Standard 类型。

6）compare_with：Column 名称的排序规则，可选的项如下：

- AsciiType
- UTF8Type
- LexicalUUIDType
- TimeUUIDType
- LongType
- IntegerType

7）compare_subcolumns_with：SuperColumn 下的 Column 名称的排序规则。可选的项如下：

- AsciiType

- UTF8Type
- LexicalUUIDType
- TimeUUIDType
- LongType
- IntegerType

8）rows_cached：row 缓存的数量，可以为整数或者百分比。

9）Keys_cached：Key 缓存的数量，可以为整数或者百分比。

10）row_cache_save_period_in_seconds：定义 ColumnFamily 中的持久化 row 缓存的时间间隔，如果为 0，关闭持久化 row 缓存功能。

11）key_cache_save_period_in_seconds：定义 ColumnFamily 中的持久化 Key 缓存的时间间隔，如果为 0，关闭持久化 Key 缓存功能。

12）gc_grace_seconds：定义 ColumnFamily 中从数据标记为删除到真正进行物理删除的时间间隔，如果不设置，默认为 10 天（864 000s）。

13）memtable_flush_after_mins：定义 ColumnFamily 中 Memtable 最大的生存时间。

14）memtable_throughput_in_mb：定义 ColumnFamily 中 Memtable 最大缓存的数据大小。

15）memtable_operations_in_millions：定义 ColumnFamily 中 Memtable 最大缓存的数据条数。

16）min_compaction_threshold：定义 ColumnFamily 中执行数据压缩的最小 SSTable 文件数。

17）max_compaction_threshold：定义 ColumnFamily 中执行数据压缩的最大 SSTable 文件数。

18）default_validation_class：定义 ColumnFamily 中默认校验值的类型规则。可选的项如下：

- AsciiType
- UTF8Type
- LexicalUUIDType
- TimeUUIDType
- LongType
- IntegerType

19）column_metadata：定义二级索引的属性。

20）name：定义需要进行二级索引的 Column 名称。

21）validator_class：定义 ColumnFamily 中校验值的类型规则。可选的项如下：

- AsciiType
- UTF8Type
- LexicalUUIDType
- TimeUUIDType

- LongType
- IntegerType

22）index_type：定义二级索引的类型，目前支持的选项为：KEYS，这种方式将采取反转 Key 建索引。

11.3　Cassandra 集群的运行和维护

在 Cassandra 安装目录的 bin 目录下，提供了一系列帮助用户维护 Cassandra 集群的脚本。这些脚本包装了 Java 程序的调用命令，在使用的过程中，将调用 Cassandra 中 org. apache. cassandra. tools 包下的相关实现。

在 Cassandra 安装目录的 bin 目录下执行“ls -l”命令，查看当前目录下的文件如下：

```
[admin@ hadoop12 bin] $ ls -l
total 68
-rwxr-xr-x 1 admin admin 5706 Oct 28 23:31 cassandra
-rwxr-xr-x 1 admin admin 2555 Oct 28 23:31 cassandra.bat
-rwxr-xr-x 1 admin admin 1667 Oct 28 23:31 cassandra-cli
-rwxr-xr-x 1 admin admin 1753 Oct 28 23:31 cassandra-cli.bat
-rw-r--r-- 1 admin admin 1504 Oct 28 23:31 cassandra.in.sh
-rwxr-xr-x 1 admin admin 1839 Oct 28 23:31 clustertool
-rwxr-xr-x 1 admin admin 1663 Oct 28 23:31 config-converter
-rwxr-xr-x 1 admin admin 1696 Oct 28 23:31 json2sstable
-rwxr-xr-x 1 admin admin 2166 Oct 28 23:31 json2sstable.bat
-rwxr-xr-x 1 admin admin 2145 Oct 28 23:31 nodetool
-rwxr-xr-x 1 admin admin 1867 Oct 28 23:31 nodetool.bat
-rwxr-xr-x 1 admin admin 1650 Oct 28 23:31 schematool
-rwxr-xr-x 1 admin admin 1697 Oct 28 23:31 sstable2json
-rwxr-xr-x 1 admin admin 2355 Oct 28 23:31 sstable2json.bat
-rwxr-xr-x 1 admin admin 1780 Oct 28 23:31 sstablekeys
-rwxr-xr-x 1 admin admin 1175 Oct 28 23:31 stop-server
```

上面列出的脚本中，以 .bat 结尾的脚本是 Windows 系统版本。在非 Windows 系统中，启动 Cassandra 使用脚本的为 cassandra，而在 Windows 系统中，启动 Cassandra 使用的脚本为 cassandra. bat。

其中各个脚本的功能如下：

1）cassandra：启动 Cassandra 进程。

2）cassandra-cli：启动 Cassandra 的客户端，用户可以通过这个客户端直接使用命令插入数据到 Cassandra 中，也可以从 Cassandra 读取数据。不需要使用编程语言编写专用的程序。

3）clustertool：获取 Cassandra 集群中负责某一个 Keyspace 的节点信息，在整个集群中执行数据快照操作，删除某一个 Column Family 数据。

4）config-converter：将 Cassandra 0.7 版本之前的 XML 格式配置文件转换成 ymal 格式的配置文件。

5）json2sstable：将 JSON 数据导入 SSTable 中。

6）sstable2json：将 SSTable 中的指定的部分数据导出为 JSON 格式的数据。

7）nodetool：查看某台 Cassandra 服务器的运行状况，并可以对该服务器进行相关的操作。

8）schematool：导出 Cassandra 集群中的某一台服务器的配置信息（cassandra. ymal）或者指定某一台服务器重新加载配置信息。

9）sstablekeys：将指定的 SSTable 中所有的数据都导出为 JSON 格式的数据。

10）stop - server：使用指定的 PID 文件，停止正在运行的 Cassandra 进程。

所有的维护脚本和 Cassandra 服务器通信都是通过 JMX 进行的，默认的端口号为 8080，这个选项在 cassandra - env. sh 文件中设置如下：

```
JMX_PORT = "8080"
```

维护脚本执行的程序生成的日志信息可以在 log4j - tools. properties 文件中进行配置。默认所有的日志信息都直接输出到屏幕。

11.3.1 查看集群的运行情况

查看 Cassandra 集群的运行情况使用的脚本为 nodetool。

1. 查看集群整体的运行情况

查看集群中有哪些服务器在提供服务，并得知每一台服务器中存储的数据大小。

```
[admin@ dw - adw1 apache - cassandra - 0.6.5] $ sh bin/nodetool - h localhost - p
8080 ring
    Address  Status  Load  Range                Ring
                                 126692146432040178409928654073007590733
172.16.197.194Up  372.94 GB  29252733487478031053094602000287347581   |< -- |
172.16.197.191Up 394.92 GB  60436098281240948191708687738065373765   |  ^
172.16.197.193Up 294.11 GB  63689940583717811598708352333303827707   v  |
172.16.197.195Up 157.12 GB  86500034489056205514172642798068926029   |  ^
172.16.197.192Up  181.96 GB  126692146432040178409928654073007590733   |-- > |
```

该命令中，-h 代表查看的主机名称，-p 代表 JMX 的端口，ring 表示需要执行操作的名称。输出的结果中，Address 代表服务器的 IP 地址；Status 代表的含义为服务器的状态，Up 代表该服务器工作正常，Down 代表该服务器处于宕机状态，需要系统管理员去处理；Load 代表该服务器存储的数据大小；Range 代表该服务器使用的 Token 值；Ring 代表该服务器在一致性哈希环中所处的位置。

在实际的维护中，通常通过这个操作来查看集群中总体的运行状态，判断哪些服务器已经宕机，数据负载是否平衡。

2. 查看集群中某个服务器的运行状态

查看集群中某个服务器的运行状态。

```
[admin@ dw - adw1 apache - cassandra - 0.6.5] $sh bin/ nodetool - h localhost - p
8080 info
60436098281240948191708687738065373765
Load                 :394.92 GB
Generation No        :1288631480
Uptime (seconds) :997506
Heap Memory (MB) :1982.25 /5103.38
```

该命令中，-h 代表查看的主机名称，-p 代表 JMX 的端口，info 表示需要执行操作的名称。输出的结果中包含了该服务器使用的 Token 值；Load 代表该服务器存储的数据大小；Generation No 代表该服务器使用 Gossiper 通信的版本号；Uptime 代表该服务器启动后使用的时间长度；Heap Memory 代表该服务器目前使用的内存大小以及最大可使用的值。

在实际的维护中，通常通过这个操作来查看当前运行的服务器状态，内存是否足够。

3. 查看 ColumnFamily 执行记录

查看集群中某个服务器中所有的 ColumnFamily 执行记录。

```
[admin@ dw - adw1 apache - cassandra - 0.6.5] $sh bin/ nodetool  - h localhost  - p
8080 cfstats
    Keyspace: Keyspace1
        Read Count: 5335050
        Read Latency: 0.49735341955558054 ms.
        Write Count: 173910921
        Write Latency: 0.027932449417595807 ms.
        Pending Tasks: 0
            Column Family: Offer
            SSTable count: 1
            Space used (live): 346123443448
            Space used (total): 346123443448
            Memtable Columns Count: 0

            Memtable Data Size: 0
            Memtable Switch Count: 25937
            Read Count: 2352
            Read Latency: NaN ms.
            Write Count: 76653867
            Write Latency: NaN ms.
            Pending Tasks: 0
            Key cache: disabled
```

```
Row cache: disabled
Compacted row minimum size: 220
Compacted row maximum size: 4091
Compacted row mean size: 1443

Column Family: Member
SSTable count: 9
Space used (live): 69036641996
Space used (total): 69036641996
Memtable Columns Count: 363401
Memtable Data Size: 17237438
Memtable Switch Count: 5345
Read Count: 267989
Read Latency: NaN ms.
Write Count: 37838657
Write Latency: NaN ms.
Pending Tasks: 0
Key cache capacity: 56399616
Key cache size: 114830
Key cache hit rate: NaN
Row cache capacity: 1000000
Row cache size: 56543
Row cache hit rate: NaN
Compacted row minimum size: 288
Compacted row maximum size: 6891
Compacted row mean size: 1169
```

该命令中，-h 代表查看的主机名称，-p 代表 JMX 的端口，cfstats 表示需要执行操作的名称。输出的结果中包含了该服务器中存储的数据的记录条数，每一个 ColumnFamily 被读取的次数，读取的消耗的平均时间，每一个 ColumnFamily 被写入的次数，写入消耗的平均时间，SSTable 的个数，数据压缩的情况，RowCache 与 KeyCache 的情况等。

在实际的维护中，通常通过这个操作来查看当前运行的服务器提供服务的质量，数据读取和写入的速度是否满足业务的需求。

4. 查看 Cassandra 当前执行操作

查看集群中某个服务器中当前执行操作记录。

```
[admin@ dw - adw1 apache - cassandra - 0.6.5] $sh bin/ nodetool  - h localhost  - p
8080 tpstats
    Pool Name                    Active   Pending      Completed
    FILEUTILS - DELETE - POOL         0         0           1288
    STREAM - STAGE                    0         0              0
    RESPONSE - STAGE                  0         0      212283790
    ROW - READ - STAGE                0         0        5323514
```

```
LB - OPERATIONS                 0         0         0
MISCELLANEOUS - POOL            0         0         0
GMFD                            0         0         2992381
LB - TARGET                     0         0         0
CONSISTENCY - MANAGER           0         0         81
ROW - MUTATION - STAGE          0         0         174430896
MESSAGE - STREAMING - POOL      0         0         0
LOAD - BALANCER - STAGE         0         0         0
FLUSH - SORTER - POOL           0         0         0
MEMTABLE - POST - FLUSHER       0         0         1673
FLUSH - WRITER - POOL           0         0         1673
AE - SERVICE - STAGE            0         0         0
HINTED - HANDOFF - POOL         0         0         27
```

该命令中，-h 代表查看的主机名称，-p 代表 JMX 的端口，tpstats 表示需要执行操作的名称。输出的结果中包含了该服务器中每一个线程正在执行的操作个数，等待执行的操作个数，已经完成执行的操作个数。其中 RESPONSE - STAGE 代表响应其他服务器请求的次数；ROW - READ - STAGE 代表执行读取操作的次数；GMFD 代表执行 Gossiper 通信的次数；ROW - MUTATION - STAGE 代表执行更新操作的次数；MEMTABLE - POST - FLUSHER 代表数据从 Memtable 中写入 SSTable 文件中的次数。

在实际的维护中，通常通过这个操作来查看当前运行的服务器状态，找出操作过于频繁的线程。假如在 ROW - READ - STAGE 中有大量的操作处于 Pending 的状态，说明读取操作已经遇到瓶颈，可以在配置文件中适当增大并发读取的数量，或者考虑增加更多的 Cassandra 服务器。同理，如 ROW - MUTATION - STAGE 中有大量的操作处于 Pending 的状态，也可以采用类似的策略进行调整。

11.3.2　添加节点

在集群中添加一个节点，被添加的节点首先将获取集群中节点之间的信息，然后从其他服务器中获取本服务器需要的数据，最后开启对外的服务器端口和线程，开始对外提供服务。

回到本章开头的那个例子，共有 3 台服务器：hadoop12，hadoop13，hadoop14，其中 Cassandra 集群中已经包含了两台服务器：hadoop12，hadoop13，并且在这 3 台服务器的配置文件中，Bootstrap 的配置为：

```
auto_bootstrap: true
```

seeds 的配置为：

```
seeds:
    - hadoop12
    - hadoop13
```

如果要将 hadoop14 这台服务器加入到已有的集群中，首先保证 hadoop14 的配置文件与

另外两台服务器一致，并且各个服务器之间的网络是联通的。

最后在 hadoop14 上执行如下命令，直接启动 Cassandra 即可加入集群：

```
[admin@ hadoop14 ~ ] $ sh apache - cassandra - 0.7.0 - beta3 / bin / cassandra
```

该命令执行后，输出信息如下：

```
 [admin@ hadoop14 ~ ] $ INFO 15 :24 :32 ,624 Heap size: 4286251008 / 4286251008
    INFO 15 :24 :32 ,641 JNA not found. Native methods will be disabled.
    INFO 15 :24 :32 ,652 Loading settings from file:/ home/ admin/ apache - cassandra -
0.7.0 - beta3 / conf / cassandra.yaml
    INFO 15 :24 :32 ,827 DiskAccessMode is standard, indexAccessMode is mmap
    INFO 15 :24 :32 ,916 Creating new commitlog segment /home/admin/apache - cassandra
 - 0.7.0 - beta3 / commitlog / CommitLog - 1289633072916.log
    INFO 15 :24 :32 ,988 read 0 from saved key cache
    INFO 15 :24 :32 ,992 read 0 from saved key cache
    INFO 15 :24 :32 ,993 read 0 from saved key cache
    INFO 15 :24 :32 ,995 read 0 from saved key cache
    INFO 15 :24 :32 ,996 read 0 from saved key cache
    INFO 15 :24 :33 ,000 loading row cache for LocationInfo of system
    INFO 15 :24 :33 ,000 completed loading (0 ms; 0 keys) row cache for LocationInfo
of system
    INFO 15 :24 :33 ,001 loading row cache for HintsColumnFamily of system
    INFO 15 :24 :33 ,001 completed loading (0 ms; 0 keys) row cache for HintsColumnFamily
of system
    INFO 15 :24 :33 ,002 loading row cache for Migrations of system
    INFO 15 :24 :33 ,002 completed loading (0 ms; 0 keys) row cache for Migrations of sys-
tem
    INFO 15 :24 :33 ,005 loading row cache for Schema of system
    INFO 15 :24 :33 ,006 completed loading (1 ms; 0 keys) row cache for Schema of system
    INFO 15 :24 :33 ,006 loading row cache for IndexInfo of system
    INFO 15 :24 :33 ,007 completed loading (0 ms; 0 keys) row cache for IndexInfo of sys-
tem
    INFO 15 :24 :33 ,043 Couldn 't detect any schema definitions in local storage.
    INFO 15 :24 :33 ,044 Found table data in data directories. Consider using JMX to call
org. apache. cassandra. service. StorageService. loadSchemaFromYaml ().
    INFO 15 :24 :33 ,047 No commitlog files found; skipping replay
    INFO 15 :24 :33 ,073 Upgrading to 0.7. Purging hints if there are any. Old hints will
be snapshotted.
    INFO 15 :24 :33 ,078 Cassandra version: 0.7.0 - beta3
    INFO 15 :24 :33 ,079 Thrift API version: 19.4.0
    INFO 15 :24 :33 ,089 Loading persisted ring state
    INFO 15 :24 :33 ,091 Starting up server gossip
    INFO 15 :24 :33 ,100 switching in a fresh Memtable for LocationInfo at CommitLogCon-
text (file ='/ home/ admin/ apache - cassandra - 0.7.0 - beta3 / commitlog / CommitLog -
1289633072916. log ', position =700)
    INFO 15 :24 :33 ,101 Enqueuing flush of Memtable - LocationInfo @ 1930362236 (224
```

```
bytes, 4 operations)
    INFO 15:24:33,102 Writing Memtable-LocationInfo@1930362236(224 bytes, 4 operations)
    INFO 15:24:33,309 Completed flushing /home/admin/apache-cassandra-0.7.0-beta3/data/system/LocationInfo-e-1-Data.db
    INFO 15:24:33,352 Joining: getting load information
    INFO 15:24:33,353 Sleeping 90000 ms to wait for load information...
    INFO 15:24:35,188 Node /172.16.197.113 is now part of the cluster
    INFO 15:24:35,192 Node /172.16.197.112 is now part of the cluster
    INFO 15:24:35,260 InetAddress /172.16.197.112 is now UP
    INFO 15:24:35,261 Started hinted handoff for endpoint /172.16.197.112
    INFO 15:24:35,263 Finished hinted handoff of 0 rows to endpoint /172.16.197.112
    INFO 15:24:36,343 InetAddress /172.16.197.113 is now UP
    INFO 15:24:36,343 Started hinted handoff for endpoint /172.16.197.113
    INFO 15:24:36,344 Finished hinted handoff of 0 rows to endpoint /172.16.197.113
    INFO 15:26:03,371 Joining: getting bootstrap token
    INFO 15:26:03,384 New token will be 61078635599166706937511052402724559481 to assume load from /172.16.197.113
    INFO 15:26:03,386 Will not load MX4J, mx4j-tools.jar is not in the classpath
    INFO 15:26:03,420 Binding thrift service to hadoop14.hst.ali.dw.alidc.net/172.16.197.114:9160
    INFO 15:26:03,423 Using TFramedTransport with a max frame size of 15728640 bytes.
```

从启动的输出信息中可以看到，hadoop14 这台服务器在启动的时候执行了 Bootstrap 操作，从其他服务器中获取了属于本服务器的数据，然后启动，加入集群中，提供数据的读取与更新服务。

最后执行如下命令，确定服务器 hadoop14 已经成功加入集群中：

```
[admin@hadoop14 ~]$ sh apache-cassandra-0.7.0-beta3/bin/nodetool -h hadoop12 -p 8080 ring
Address         Status State   Load            Token
                                       116791188121120886117513664065968576037
172.16.197.114  Up     Normal  5.3 KB          61078635599166706937511052402724559481
172.16.197.113  Up     Normal  15.41 KB        76532511552026171774714881963084655022
172.16.197.112  Up     Normal  7.53 KB         116791188121120886117513664065968576037
```

可以看到，服务器 hadoop14 已经成功加入集群中。

11.3.3　删除节点

从集群中删除某一个节点，被删除的节点首先将关闭对外监听的所有服务和进程，然后将本服务器中的数据发送给其他需要该数据的服务器。

假设需要从集群中删除服务器 hadoop14，而该服务器中的 Cassandra 进程处于运行状态，可以使用如下命令：

```
[admin@hadoop14 ~]$ sh apache-cassandra-0.7.0-beta3/bin/nodetool -h ha-
```

```
doop14 -p 8080 decommission
```

注意，这个命令中，一定要小心设置 -h 后面的参数，这个参数代表让哪一台服务器执行删除的命令。假设在服务器 hadoop14 上面执行上述命令的 -h 参数为 hadoop12，那么集群中被删除的服务器就是 hadoop12 了。

命令执行后，程序在演示的环境中输出信息如下：

```
Exception in thread "main" java.lang.AssertionError
    at org.apache.cassandra.service.StorageService.getLocalToken(StorageSer-
vice.java:1125)
    at org.apache.cassandra.service.StorageService.startLeaving(StorageSer-
vice.java:1524)
    at org.apache.cassandra.service.StorageService.decommission(StorageSer-
vice.java:1543)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
```

从输出可以看出，程序运行出错了。原因是服务器从本地的系统表（System Keyspace）中读取 Token 信息失败。在我们测试的环境中，服务器 hadoop14 是刚刚启动的，系统表的数据并没有写入本地磁盘中。要解决这个问题，只要重启服务器 hadoop14 中的 Cassandra 进程即可。

再次执行删除服务器 hadoop14 的命令，没有任何输出，查看 Cassandra 程序的系统日志如下：

```
 INFO [RMI TCP Connection(4)-172.16.197.114] 2010-11-13 15:47:27,470 StorageS-
ervice.java (line 457) Leaving: sleeping 30000 ms for pending range setup
 INFO [RMI TCP Connection(4)-172.16.197.114] 2010-11-13 15:47:59,483 Messaging-
Service.java (line 360) Shutting down MessageService...
 INFO [RMI TCP Connection(4)-172.16.197.114] 2010-11-13 15:47:59,484 Messaging-
Service.java (line 377) Shutdown complete (no further commands will be processed)
 INFO [ACCEPT-hadoop14.hst.ali.dw.alidc.net/172.16.197.114] 2010-11-13 15:
47:59,484 MessagingService.java (line 541) MessagingService shutting down server
thread.
 INFO [RMI TCP Connection(4)-172.16.197.114] 2010-11-13 15:47:59,485 StorageS-
ervice.java (line 457) Decommissioned
```

但是该 Cassandra 进程依旧在服务器系统中执行，可以将这个进程关闭。

最后，查看集群的状态如下：

```
[admin@hadoop14 ~]$ sh apache-cassandra-0.7.0-beta3/bin/nodetool -h ha-
doop12 -p 8080 ring
Address         Status State   Load            Token
                                       116791188121120886117513664065968576037
172.16.197.113  Up     Normal  7.57 KB         76532511552026171774714881963084655022
172.16.197.112  Up     Normal  12.51 KB        116791188121120886117513664065968576037
```

说明该服务器已经从集群中删除成功。

如果需要从集群中删除的节点已经处于宕机状态，可是使用 removetoken 命令进行删除。

假设现在集群中的状态如下：

```
[admin@ hadoop14 ~ ] $ sh apache - cassandra - 0.7.0 - beta3 / bin / nodetool  - h ha-
doop12  - p 8080 ring
Address         Status State  Load        Token
                                  116791188121120886117513664065968576037
172.16.197.113 Up   Normal 12.51 KB    765325115520261717747148819630846550 22
172.16.197.114 Down Normal 5.3 KB      966618498365735289461142730145266155 29
172.16.197.112 Up   Normal 17.46 KB    116791188121120886117513664065968576037
```

其中有一台服务器（hadoop14）已经处于宕机状态了。现在需要将这台服务器从集群中删除。

```
[admin@ hadoop14 ~ ] $ sh apache - cassandra - 0.7.0 - beta3 / bin / nodetool  - h ha-
doop12  - p 8080 removetoken 966618498365735289461142730145266155 29
```

在上面的命令中，需要在 removetoken 命令后添加需要删除的 Token 的值，即需要删除的服务器的 Token 值。

执行成功后，再次查看集群中的状态如下：

```
[admin@ hadoop14 ~ ] $ sh apache - cassandra - 0.7.0 - beta3 / bin / nodetool  - h ha-
doop12  - p 8080 ring
Address         Status State  Load        Token
                                  116791188121120886117513664065968576037
172.16.197.113 Up   Normal 12.51 KB    765325115520261717747148819630846550 22
172.16.197.112 Up   Normal 17.46 KB    116791188121120886117513664065968576037
```

这样就完成了集群中宕机节点的删除操作。这个操作和上一步操作的区别为：使用 removetoken 的命令并不能将删除节点的数据传递给其他的节点。

11.3.4　移动节点

由于 Cassandra 集群目前不支持自动移动数据均衡，所以集群在运行一段时间后，将会出现数据分布不均匀的情况，比如一台机器中的数据量很大，另外两台机器中的数据量比较小。

这种情况下，可以使用移动节点的方式均衡数据的分布。

从集群中移动某一个节点，相当于先在集群中删除需要移动的节点，然后再对删除的节点执行 Bootstrap 操作。即被移动的节点先将本服务器中的数据发送给其他需要该数据的服务器，然后在新的位置中，再从其他的服务器中获取属于本节点的数据。

假设系统当前的集群状态如下：

```
Address         Status State  Load        Token
                                  154903721662098260367508836040371134953
```

```
172.16.197.113 Up    Normal 12.51 KB   76532511552026171774714881963084655022
172.16.197.112 Up    Normal 17.46 KB   116791188121120886117513664065968576037
172.16.197.114 Up    Normal 7.57 KB    154903721662098260367508836040371134953
```

需要在集群中移动服务器 hadoop14，移动后的 Token 值为：76532511552026171774714-881963084695032，并且该服务器中的 Cassandra 进程处于运行状态，可以使用如下命令：

```
[admin@ hadoop14 ~] $sh apache-cassandra-0.7.0-beta3/bin/nodetool -h hadoop14 -p 8080 move 76532511552026171774714881963084695032
```

命令执行成功后，没有任何输出，查看 Cassandra 程序的系统日志如下：

```
INFO [RMI TCP Connection(4)-172.16.197.114] 2010-11-13 16:38:55,332 StorageService.java (line 457) Leaving: sleeping 30000 ms for pending range setup
INFO[RMI TCP Connection(4)-172.16.197.114]2010-11-13 16:39:27,344 StorageService.java(line 1677)re-bootstrapping to new token 76532511552026171774714881963084695032
INFO [RMI TCP Connection(4)-172.16.197.114] 2010-11-13 16:39:27,345 ColumnFamilyStore.java (line 631) switching in a fresh Memtable for LocationInfo at CommitLogContext(file='/home/admin/apache-cassandra-0.7.0-beta3/commitlog/CommitLog-1289636309166.log', position=740)
INFO [RMI TCP Connection(4)-172.16.197.114] 2010-11-13 16:39:27,345 ColumnFamilyStore.java (line 930) Enqueuing flush of Memtable-LocationInfo@404343994(70 bytes, 3 operations)
INFO [FlushWriter:1] 2010-11-13 16:39:27,346 Memtable.java (line 154) Writing Memtable-LocationInfo@404343994(70 bytes, 3 operations)
INFO [FlushWriter:1] 2010-11-13 16:39:27,454 Memtable.java (line 161) Completed flushing /home/admin/apache-cassandra-0.7.0-beta3/data/system/LocationInfo-e-6-Data.db
INFO [RMI TCP Connection(4)-172.16.197.114] 2010-11-13 16:39:27,455 StorageService.java (line 457) Joining: sleeping 30000 ms for pending range setup
INFO [RMI TCP Connection(4)-172.16.197.114] 2010-11-13 16:39:57,462 StorageService.java (line 457) Bootstrapping
INFO [RMI TCP Connection(4)-172.16.197.114] 2010-11-13 16:39:57,467 ColumnFamilyStore.java (line 631) switching in a fresh Memtable for LocationInfo at CommitLogContext(file='/home/admin/apache-cassandra-0.7.0-beta3/commitlog/CommitLog-1289636309166.log', position=1036)
INFO [RMI TCP Connection(4)-172.16.197.114] 2010-11-13 16:39:57,468 ColumnFamilyStore.java (line 930) Enqueuing flush of Memtable-LocationInfo@427736678(53 bytes, 2 operations)
INFO [FlushWriter:1] 2010-11-13 16:39:57,468 Memtable.java (line 154) Writing Memtable-LocationInfo@427736678(53 bytes, 2 operations)
INFO [FlushWriter:1] 2010-11-13 16:39:57,592 Memtable.java (line 161) Completed flushing /home/admin/apache-cassandra-0.7.0-beta3/data/system/LocationInfo-e-7-Data.db
INFO [RMI TCP Connection(4)-172.16.197.114] 2010-11-13 16:39:57,595 StorageService.java (line 250) Bootstrap/move completed! Now serving reads.
```

从日志中可以看到，在节点执行 Move 的过程中，经历了 Remove 和 Bootstrap 两个阶段。

再次查看集群中的状态如下：

```
[admin@ hadoop14 ~ ] $ sh apache - cassandra - 0.7.0 - beta3 / bin / nodetool  - h hadoop14  - p 8080 ring
       Address         Status State  Load        Token
                                                 116791188121120886117513664065968576037
       172.16.197.113 Up  Normal 17.56 KB    76532511552026171774714881963084655022
       172.16.197.114 Up  Normal 17.83 KB    76532511552026171774714881963084695032
       172.16.197.112 Up  Normal 22.5 KB     116791188121120886117513664065968576037
```

可以看到，服务器 hadoop14 的位置已经变更，这样就完成了集群中节点的移动操作。

另外，如果不希望手工指定移动节点的 Token 的话，也可以让系统自动计算，使用如下命令即可：

```
[admin@ hadoop14 ~ ] $ sh apache - cassandra - 0.7.0 - beta3 / bin / nodetool  - h hadoop14  - p 8080 loadbalance
```

11.3.5　数据维护

Cassandra 对数据的维护体现在 3 个方面：数据压缩控制、数据修复和数据导入导出。

1. 数据压缩控制

在本书的第 9 章中，已经详细讲解了 Cassandra 数据压缩方面的工作机制。Cassandra 提供了以下功能可以对数据压缩进行相应的控制。

1）查看某一个 ColumnFamily 的数据压缩设置。

```
nodetool -h [hostname] -p [port] getcompactionthreshold [keyspace] [cfname]
```

其中［keyspace］和［cfname］分别为需要查看的 Keyspace 名称和 ColumnFamily 的名称。

2）设置某一个 ColumnFamily 的数据压缩。

```
nodetool -h [hostname] -p [port] setcompactionthreshold [keyspace] [cfname] [minthreshold] [maxthreshold]
```

其中［keyspace］和［cfname］分别为需要设置的 Keyspace 名称和 ColumnFamily 的名称，［minthreshold］和［maxthreshold］分别代表数据压缩的最小阈值和最大阈值。

3）让 Cassandra 将某一个 Keyspace 中的 Memtable 中缓存的数据写入 SSTable 文件中。

```
nodetool -h [hostname] -p [port] flush [keyspace]
```

其中［keyspace］为需要操作的 Keyspace 名称。

4）让 Cassandra 将某一个 Keyspace 中的所有的 SSTable 文件进行数据压缩。

```
nodetool -h [hostname] -p [port] compact [keyspace]
```

其中［keyspace］为需要压缩的 Keyspace 名称。

2. 数据修复

Cassandra 在数据修复的过程中，将判断本地的数据是否是集群需要的数据，如果不是，则删除这部分数据。同时判断本地数据是否有缺失与不一致现象，如果发现数据有缺失和不一致的情况，将从集群中的其他服务器获取相应的数据。

Cassandra 对数据修复的命令如下：

```
nodetool -h [hostname] -p [port] repair [keyspace]
```

3. 数据导入导出

Cassandra 中支持数据以 JSON 格式和 SSTable 格式互相转换。

将数据导入可以使用 json2sstable 脚本，它会将 JSON 格式的数据导入 Cassandra 成为 SSTable 文件，具体的使用方法如下：

```
json2sstable -K keyspace -c column_family <json> <sstable>
```

其中 <json> 为需要导入的 JSON 数据文件地址，<sstable> 为导入的 SSTable 名称。

将数据导出可以使用 sstable2json 脚本，它会将 SSTable 文件中指定 Key 的数据导出为 JSON 格式的文件，具体的使用方法如下：

```
json2sstable <sstable> [-k key [-k key [...]] -x key [-x key [...]]]
```

其中 <sstable> 为要导出的 SSTable 名称，-k key 代表需要导出数据包含的 Key，-x key 代表需要导出数据不包含的 Key。

另外，数据导出还可以使用 sstablekeys 脚本。这个脚本直接将 sstable 中的所有数据都导出到 JSON 格式的文件中，具体的使用方法如下：

```
sstablekeys <sstable>
```

其中 <sstable> 为要导出的 SSTable 名称。

4. 数据快照

通过数据快照（snapshot），可以很快地获得某一个 Keyspace 中当前的数据。这样就可以在任何时候将当前的数据进行备份，或者将数据恢复到做数据快照时候的情况。

通过以下命令，可以对指定的 Cassandra 服务器进行数据快照的操作。

```
bin/nodetool -h [hostname] -p [port] snapshot [Keyspace]
```

如执行如下命令：

```
bin/nodetool -h localhost-p 9160 snapshot
```

这个命令将会在 localost 机器上对所有的 Keyspace 数据做数据快照操作，输入如下：

```
DEBUG 14:25:15,385 Snapshot for Keyspace1 table data file
/var/lib/cassandra/data/Keyspace1/Standard2-1-Filter.db
created as /var/lib/cassandra/data/Keyspace1/snapshots/1277673915365/
Standard2-1-Filter.db
DEBUG 14:25:15,424 Snapshot for system table data file
/var/lib/cassandra/data/system/LocationInfo-9-Filter.db
created as /var/lib/cassandra/data/system/snapshots/1277673915389/
LocationInfo-9-Filter.d
```

请注意，从上面的输出中可以看到，这里不仅仅是对用户定义的 Keyspace 做了数据快照，同时对 system Keyspace 也做了一个数据快照。

另外，如果只需要对某一个特定的 Keyspace 做数据快照，可以使用如下的命令：

```
bin/nodetool -h localhost-p 9160 snapshot Keyspace1
```

在上面的命令中，Cassandra 就只会对本机的 Keyspace1 数据做数据快照。

同样的，我们也可以通过命令删除数据快照的数据。

```
bin/nodetool -h [hostname] -p [port] cleansnapshot [Keyspace]
```

与创建数据快照类似，如果不指定具体的 Keyspace，将删除本机所有的数据快照。如执行以下命令：

```
bin/nodetool -h localhost-p 9160 cleansnapshot
```

这个命令的输出如下：

```
DEBUG 14:45:00,490 Disseminating load info ...
DEBUG 14:45:11,797 Removing snapshot directory
/var/lib/cassandra/data/Keyspace1/snapshots
DEBUG 14:45:11,798 Deleting Standard2-1-Index.db
DEBUG 14:54:45,727 Deleting 1277675283388-Keyspace1
//...
DEBUG 14:54:45,728 Deleting snapshots
DEBUG 14:45:11,806 Cleared out all snapshot directories
```

也可以删除某一个具体的 Keyspace 下的数据快照。

```
bin/nodetool -h localhost-p 9160 cleansnapshot Keyspace1
```

11.4　本章小结

本章分析和讲解了 Cassandra 在 Linux 环境中如何搭建集群、Cassandra 中每一个配置项

的含义以及如何在生产环境中设置配置的值。另外讲解了 Cassandra 集群的运行和维护操作。

在生成环境中配置 Cassandra 需要特别小心，一定要根据服务器的实际情况和应用的特点来选择最佳的配置。另外在生产环境中使用 Cassandra 和测试环境有很大的不同，比如整个系统不能停机，服务不能中断，可以在线扩容等。

第 12 章

Cassandra 与 Hadoop 的整合

本章内容

- Hadoop 快速入门
- 为什么要整合 Cassandra 与 Hadoop
- 使用 Map/Reduce 导入数据到 Cassandra 中
- 将 Cassandra 中的数据作为 Map/Reduce 输入
- 本章小结

在第 11 章中，介绍了如何在 Linux 环境中搭建 Cassandra 集群，以及如何维护 Cassandra 集群。本章将讲解如何将 Cassandra 与 Hadoop 进行整合，在 Hadoop 与 Cassandra 中进行数据间的交换，从而完成相应的计算。

12.1 Hadoop 快速入门

12.1.1 Hadoop 简介

Hadoop 是 Apache 开源组织的一个分布式计算开源框架（http://hadoop.apache.org/），可运行于大规模集群上进行海量的存储与计算。

与 Hadoop 一脉相承的还有两个开源项目[⊖]：Lucene 和 Nutch。Lucene 是一个用 Java 开发的开源高性能全文检索工具包，它不是一个完整的应用程序，而是一套简单易用的 API。在全世界范围内，已有无数的软件系统、Web 网站基于 Lucene 实现了全文检索功能。后来 Doug Cutting 又开创了第一个开源的 Web 搜索引擎 Nutch（http://www.nutch.org），它在 Lucene 的基础上增加了网络爬虫和一些与 Web 相关的功能及一些解析各类文档格式的插件等。此外，Nutch 中还包含了一个分布式文件系统用于存储数据。从 Nutch 0.8.0 版本之后，Doug Cutting 把 Nutch 中的分布式文件系统以及实现 MapReduce 算法的代码独立出来，从而形成了一个新的开源项目——Hadoop。Nutch 也演化为基于 Lucene 全文检索以及 Hadoop 分布式计算平台的一个开源搜索引擎。

基于 Hadoop，可以轻松地编写可处理海量数据的分布式并行程序，并将其运行于由成百上千个节点组成的大规模计算机集群上。“云计算”是目前热门的技术名词，全球各大 IT 公司都在投资和推广这种新一代的计算模式，而 Hadoop 又被其中几家主要的公司用作其“云计算”环境中的重要基础软件，例如，雅虎除了资助 Hadoop 开发团队外，还在开发基于 Hadoop 的开源项目 Pig，这是一个专注于海量数据集分析的分布式计算程序；Facebook 系统内部也有大量的 Hadoop 集群，并且开源了 Hive，提供以 SQL 的方式进行海量数据处理的功能；Amazon 基于 Hadoop 推出了 Amazon S3（Amazon Simple Storage Service）提供可靠、快速、可扩展的网络存储服务，以及一个商用的云计算平台 Amazon EC2（Amazon Elastic Compute Cloud）。从目前的情况来看，Hadoop 注定会有辉煌的未来。

12.1.2 HDFS

Hadoop 分布式文件系统（HDFS）被设计成适合运行在通用硬件（commodity hardware）上。它和现有的分布式文件系统既有很多共同点，也有很明显的区别。HDFS 是一个具有高度容错性的系统，适合部署在廉价的机器上。HDFS 能提供高吞吐量的数据访问，非常适合大规模数据集上的应用。HDFS 放宽了一部分 POSIX 约束，来实现流式读取文件系统数据的目的。HDFS 最初是作为 Apache Nutch 搜索引擎项目的基础架构而开发的，是 Apache Ha-

⊖ 三者的创始人都是 Doug Cutting。

doop Core 项目[⊖]的一部分。

HDFS 在设计和实现上有以下几点考虑：

（1）硬件错误

硬件错误是常态而不是异常。HDFS 可能由成百上千台服务器构成，每个服务器上存储着文件系统的部分数据。我们面对的现实是：构成系统的组件数目是巨大的，而且任一组件都有可能失效，这意味着总是有一部分 HDFS 的组件是不工作的。因此，错误检测和快速、自动的恢复是 HDFS 最核心的架构目标。

（2）流式数据访问

运行在 HDFS 上的应用和普通的应用不同，需要流式访问它们的数据集。HDFS 的设计中更多地考虑了数据批处理，而不是用户交互处理。比数据访问的低延迟问题，更关键的在于数据访问的高吞吐量。POSIX 标准设置的很多硬性约束对 HDFS 应用系统不是必需的。为了提高数据的吞吐量，在一些关键方面对 POSIX 的语义做了一些修改。

（3）大规模数据集

运行在 HDFS 上的应用具有很大的数据集。HDFS 上的一个典型文件大小一般都在 G 字节至 T 字节。因此，HDFS 被调节以支持大文件存储。它应该能提供整体上高的数据传输带宽，能在一个集群里扩展到数百个节点。一个单一的 HDFS 实例应该能支撑数以千万计的文件。

（4）简单的一致性模型

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

（5）移动计算比移动数据更划算

一个应用请求的计算离它操作的数据越近就越高效，在数据达到海量级别的时候更是如此。因为这样能降低网络阻塞的影响，提高系统数据的吞吐量。将计算移动到数据附近，比将数据移动到应用所在显然更好。HDFS 为应用提供了将它们自己移动到数据附近的接口。

HDFS 采用 master/slave 架构。一个 HDFS 集群是由一个 Namenode 和一定数目的 Datanode 组成的。Namenode 是一个中心服务器，负责管理文件系统的名字空间（namespace）以及客户端对文件的访问。集群中的 Datanode 一般是一个节点一个，负责管理它所在节点上的存储。HDFS 暴露了文件系统的名字空间，用户能够以文件的形式在上面存储数据。从内部看，一个文件其实被分成一个或多个数据块，这些块存储在一组 Datanode 上。Namenode 执行文件系统的名字空间操作，比如打开、关闭、重命名文件或目录。它也负责确定数据块到具体 Datanode 节点的映射。Datanode 负责处理文件系统客户端的读写请求，在 Nameno-

---

⊖ 网站地址：http://hadoop.apache.org/core/。

de 的统一调度下进行数据块的创建、删除和复制。HDFS 架构图如图 12-1 所示。

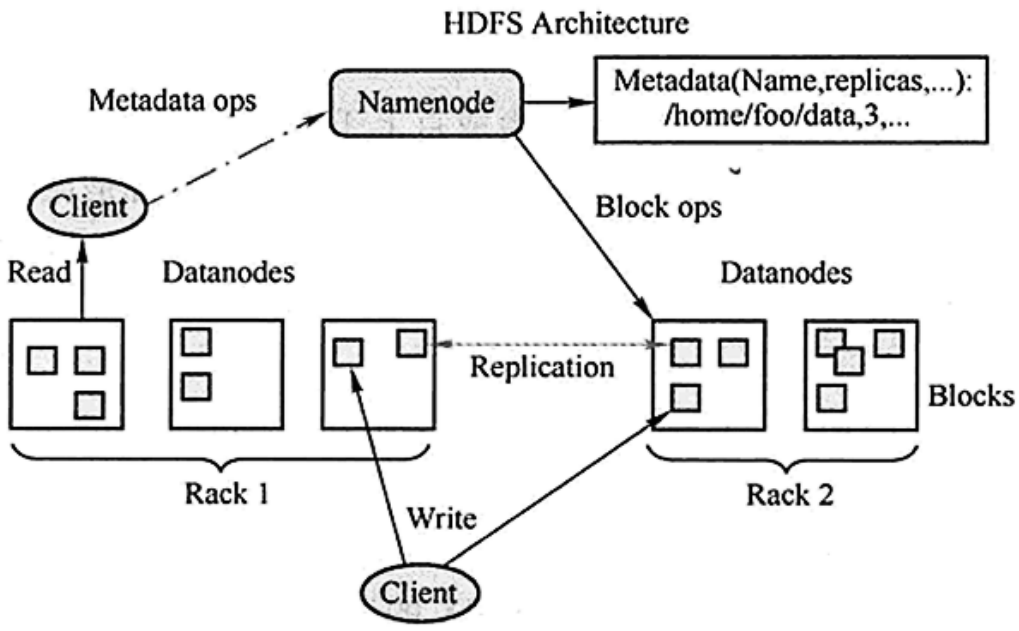


图 12-1 HDFS 架构图

Namenode 和 Datanode 被设计成可以在普通的商用机器上运行。这些机器一般运行 GNU/Linux 操作系统（OS）。HDFS 采用 Java 语言开发，因此任何支持 Java 的机器都可以部署 Namenode 或 Datanode。由于采用了可移植性极强的 Java 语言，使得 HDFS 可以部署到多种类型的机器上。一个典型的部署场景是一台机器上只运行一个 Namenode 实例，而集群中的其他机器分别运行一个 Datanode 实例。这种架构并不排斥在一台机器上运行多个 Datanode 的情况，只不过这样的情况比较少见。

集群中单一 Namenode 的结构大大简化了系统的架构。Namenode 是所有 HDFS 元数据的仲裁者和管理者，这样，用户数据永远不会流过 Namenode。

12.1.3 Map/Reduce

Hadoop Map/Reduce 是一个使用简易的软件框架，基于它写出来的应用程序能够运行在由上千个商用机器组成的大型集群上，并以一种可靠容错的方式并行处理 TB 级别的数据集。

一个 Map/Reduce 作业（job）通常会把输入的数据集切分为若干独立的数据块，由 map 任务（task）以完全并行的方式处理。框架会对 map 的输出先进行排序，然后把结果输入给 reduce 任务。通常作业的输入和输出都会被存储在分布式文件系统（HDFS）中。整个框架负责任务的调度和监控，以及重新执行已经失败的任务。

通常，Map/Reduce 框架和分布式文件系统是运行在一组相同的节点上的，也就是说，计算节点和存储节点通常在一起。这种配置允许框架在那些已经存好数据的节点上高效地调度任务，这可以使整个集群的网络带宽被非常高效地利用。

Map/Reduce 框架由一个单独的 JobTracker 和每个集群节点中的 TaskTracker 共同组成。JobTracker 负责调度构成一个作业的所有任务，这些任务分布在不同的 TaskTracker 上，Job-

Tracker 监控它们的执行，重新执行已经失败的任务。而 TaskTracker 仅负责执行由 JobTracker 指派的任务。

Map/Reduce 应用程序应该指明输入/输出的位置（路径），并通过实现合适的接口或抽象类提供 map 和 reduce 函数，再加上其他作业的参数，就构成了作业配置（job configuration）。然后，Hadoop 的 job client 提交作业（jar 包/可执行程序等）和配置信息给 JobTracker，后者负责分发这些软件和配置信息给 Task Tracker，调度任务并监控它们的执行，同时提供状态和诊断信息返回给 job client。

Map/Reduce 应用程序在执行的过程中，会对输入的数据进行拆分，然后将拆分后的数据输入到每一个 mapper 中进行计算。mapper 将计算后的结果进行分区，将同一个分区中的数据导入同一个 reducer 中。同时，reducer 将从不同的 mapper 中接收到的数据执行排序操作，对汇总后的数据再执行计算，输出最终的结果。

整个 Map/Reduce 应用程序的执行流程如图 12-2 所示。

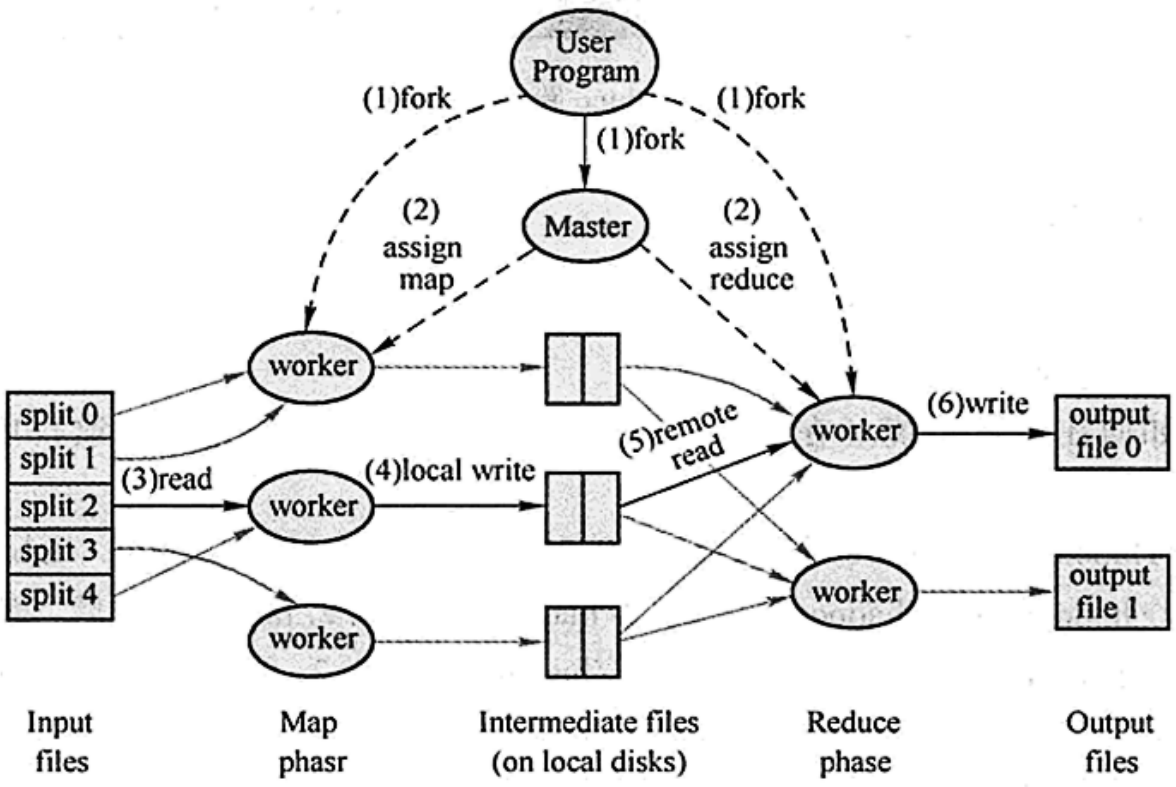


图 12-2　Map/Reduce 执行流程图

虽然 Hadoop 框架是用 Java 实现的，但 Map/Reduce 应用程序则不一定要用 Java 来写，如下面两种形式：

1）Hadoop Streaming 是一种运行作业的实用工具，它允许用户创建和运行任何可执行程序（例如：Shell、C ++ 、Python 等）来作为 mapper 和 reducer。

2）Hadoop Pipes 是一个与 SWIG 兼容的 C ++ API（没有基于 JNI 技术），它也可用于实现 Map/Reduce 应用程序。

不管采用哪一种形式，跨语言 Map/Reduce 框架实现的核心就是数据的重定向。如在 Hadoop Streaming 中，采用的是标准输入和输出的重定向。Map/Reduce 跨语言执行原理如图 12-3 所示。

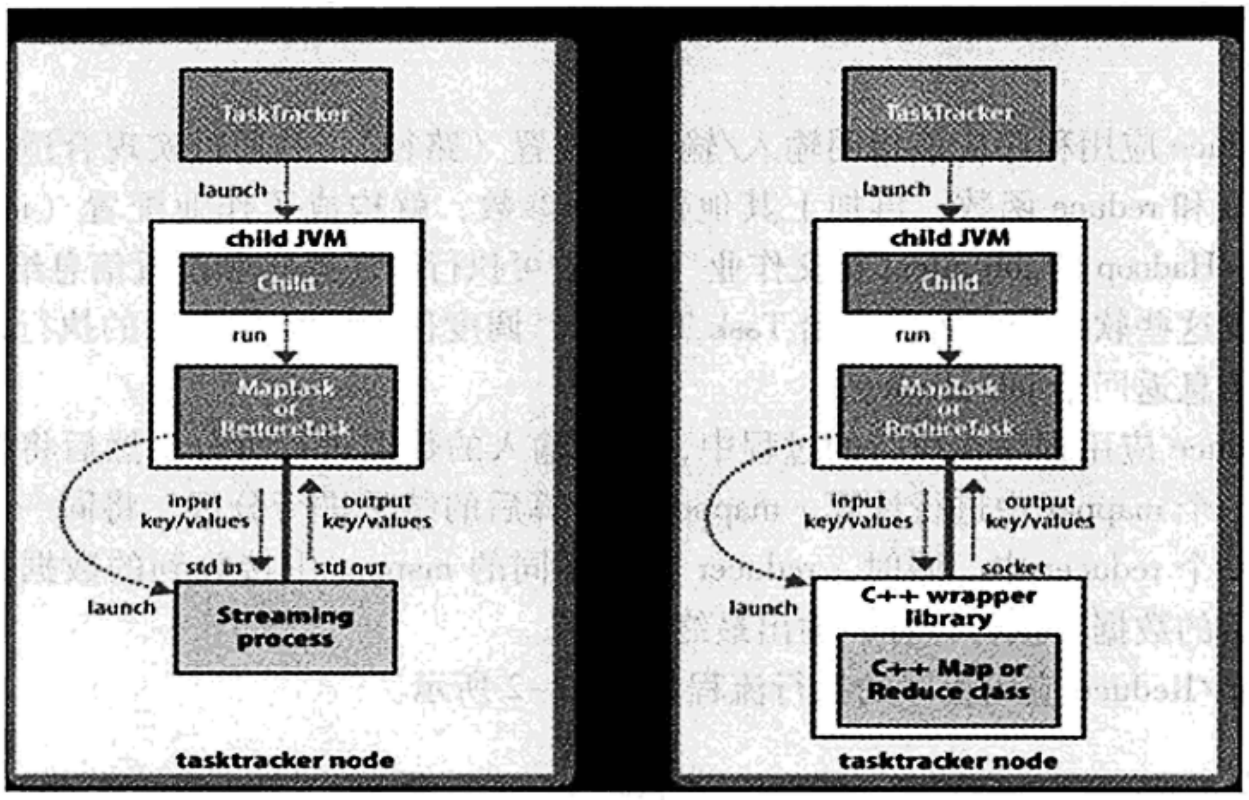


图 12-3 Map/Reduce 跨语言执行原理图

12.1.4 配置单机版 Hadoop

Hadoop 可以运行在所有的操作系统上，但是在 Windows 系统中需要额外安装 Cygwin，下载地址为：http://www.cygwin.com/。

下面介绍如何在 Mac 系统（在 Linux 系统中的配置步骤几乎完全一致）中配置单机版本的 Hadoop。

首先，在 Hadoop 官网中下载 Hadoop 0.20.2 的发行版本：http://labs.renren.com/apache-mirror//hadoop/core/hadoop-0.20.2/hadoop-0.20.2.tar.gz。然后，解压到本地磁盘中，假设解压的目录为/Users/gpcuster/Dev/hadoop-0.20.2。在 Linux 环境中，需要另外在 Sun 的官网上下载 Java 6 的运行环境。

然后修改 Hadoop 的运行环境配置文件/Users/gpcuster/Dev/hadoop-0.20.2/conf/hadoop-env.sh，添加如下 JAVA_HOME 的环境变量：

```
export JAVA_HOME = /usr
```

这样，单机运行的 Hadoop 就配置好了。

最后可以进行以下步骤来测试 Hadoop 是否可以正确执行。

1）进入到 Hadoop 的安装目录。

```
cd /Users/gpcuster/Dev/hadoop-0.20.2/
```

2）创建一个输入文件的文件夹，取名为 input。

```
mkdir input
```

3）将 conf 目录下的文件都复制到 input 目录下，作为测试的输入数据。

```
cp conf/*.xml input
```

4）运行 Map/Reduce 程序，将 input 目录中的文件作为输入数据，找出以 dfs 开头、后面包含一个或多个小写字母的单词，并将结果输出 output 目录中。

```
bin/hadoop jar hadoop-*-examples.jar grep input output 'dfs[a-z.]+'
```

运行的过程中，将看到如下输出：

```
10/11/17 23:41:58 INFO mapred.JobClient: Counters: 13
10/11/17 23:41:58 INFO mapred.JobClient: FileSystemCounters
10/11/17 23:41:58 INFO mapred.JobClient: FILE_BYTES_READ=978607
10/11/17 23:41:58 INFO mapred.JobClient: FILE_BYTES_WRITTEN=1032561
10/11/17 23:41:58 INFO mapred.JobClient: Map-Reduce Framework
10/11/17 23:41:58 INFO mapred.JobClient: Reduce input groups=1
10/11/17 23:41:58 INFO mapred.JobClient: Combine output records=1
10/11/17 23:41:58 INFO mapred.JobClient: Map input records=219
10/11/17 23:41:58 INFO mapred.JobClient: Reduce shuffle bytes=0
10/11/17 23:41:58 INFO mapred.JobClient: Reduce output records=1
10/11/17 23:41:58 INFO mapred.JobClient: Spilled Records=2
10/11/17 23:41:58 INFO mapred.JobClient: Map output bytes=17
10/11/17 23:41:58 INFO mapred.JobClient: Map input bytes=8660
10/11/17 23:41:58 INFO mapred.JobClient: Combine input records=1
10/11/17 23:41:58 INFO mapred.JobClient: Map output records=1
10/11/17 23:41:58 INFO mapred.JobClient: Reduce input records=1
```

这说明程序正常执行了。执行完毕，可以从输出目录 output 中查看结果数据。

```
cat output/*
```

输出的结果数据如下：

```
1  dfsadmin
```

这样，测试的 Map/Reduce 程序就通过了。

注意，在这个测试中，程序的输出目录 output 必须不存在，否则在执行的过程中将报错。

由于这是一个 Hadoop 的单机配置，所以整个测试的过程中并没有使用分布式文件系统（HDFS），而是使用本地文件系统和 Map/Reduce 计算框架。

12.1.5　编写 Map/Reduce 程序

在谷歌和百度上每天都会有成百上千个搜索的关键词，要解决统计每一个关键词被搜索次数的问题，Map/Reduce 程序就非常合适了。

假设程序输入的被搜索的关键词如下：

```
hello
hadoop
pengguo
Cassandra
hadoop
hello
```

在 mapper 中，输入文件中的每一行数据就是一个键值对，其中 key 是单词所在的文件的行数，value 是实际单词（如 key=1，value=hello；key=2，value=hadoop）。

mapper 每处理一行输入数据，输出的也是一个键值对。mapper 无须计算，直接将实际

的单词作为输出的 key，整数 1 作为输出的 value。

reducer 会将 mapper 输出数据中的 key 值相同的数据聚合到一起，形成一个数组。所以在 reducer 中输入的 key 就是实际单词，value 的长度就是 key 对应的这个单词实际出现的次数。最后将这个结果在 reducer 中输出即可。

要在 Eclipse 中编写这个程序，可以按照以下步骤执行：

1）在 Eclispe 中创建一个名为 hadoop - wordcount 的 Java 项目。

2）向该项目中导入编写 Map/Reduce 程序所需要的 JAR 包 hadoop - 0. 20. 2 - core. jar，如图 12-4 所示。

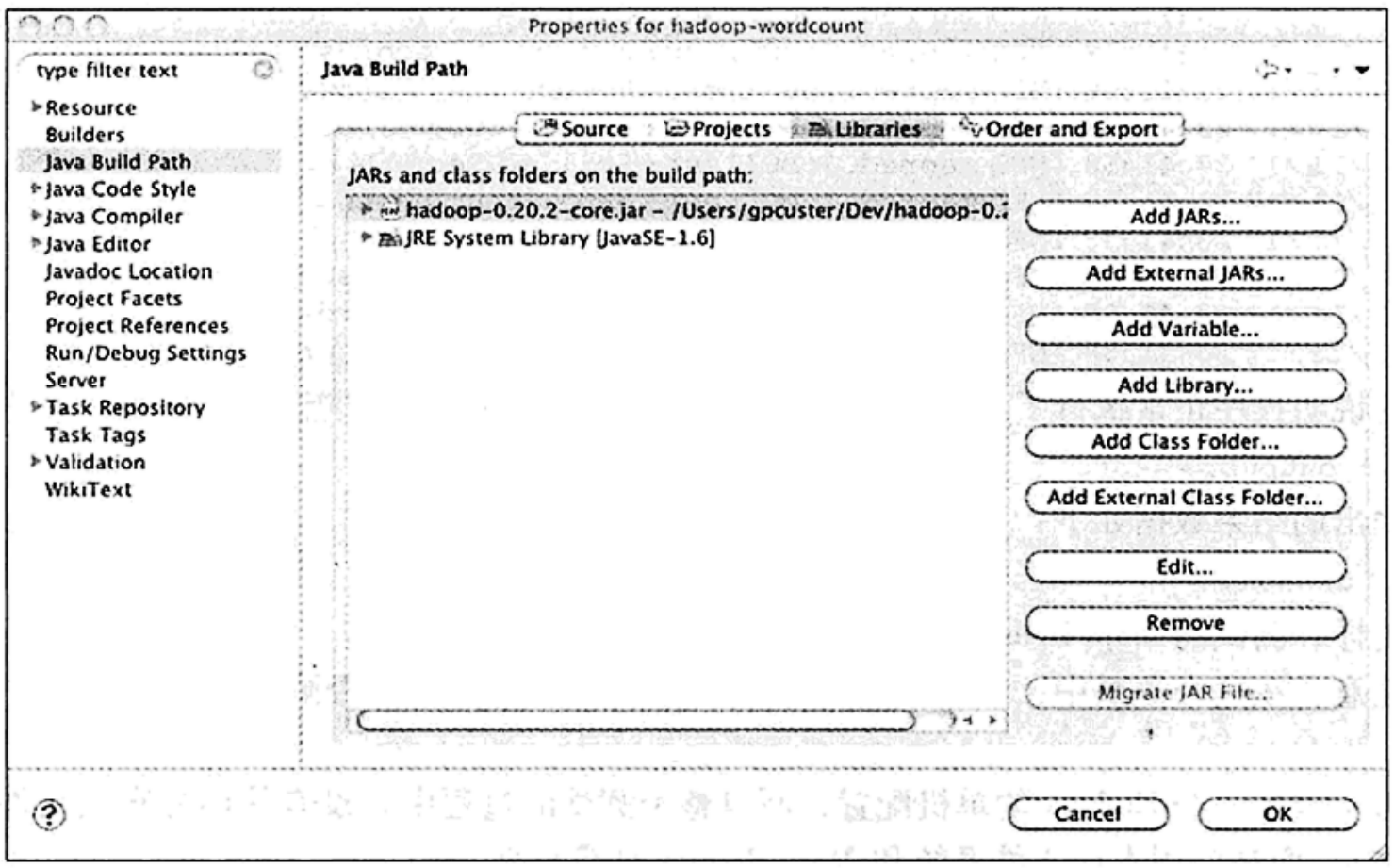


图 12-4　导入 Hadoop JAR 包

3）在 src 目录中创建名为 cassandra 的包，然后在 cassandra 包下创建名为 WordCount 的类，代码结构如图 12-5 所示。

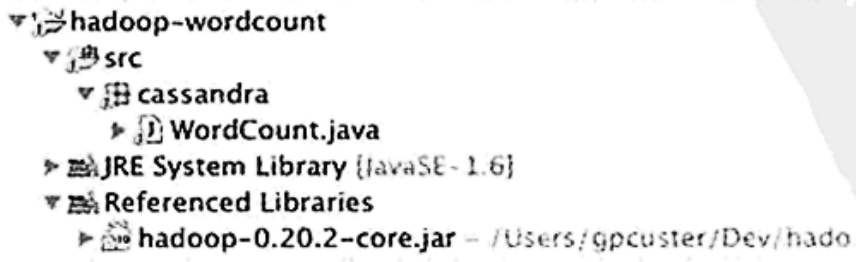


图 12-5　hadoop - wordcount 代码结构

4）在 WordCount. java 中编写 mapper 的逻辑、reducer 的逻辑，提交 job 的逻辑。最终的内容如下：

```
package cassandra;

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

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.TextOutputFormat;

public class WordCount {

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

        public void map(LongWritable key, Text value,
                OutputCollector<Text, IntWritable> output, Reporter reporter)
                throws IOException {
            String line = value.toString();
            StringTokenizer tokenizer = new StringTokenizer(line);
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                output.collect(word, one);
            }
        }
    }

    public static class Reduce extends MapReduceBase implements
            Reducer<Text, IntWritable, Text, IntWritable> {
        public void reduce(Text key, Iterator<IntWritable> values,
                OutputCollector<Text, IntWritable> output, Reporter reporter)
                throws IOException {
            int sum = 0;
            while (values.hasNext()) {
                sum += values.next().get();
            }
            output.collect(key, new IntWritable(sum));
        }
    }

    public static void main(String[] args) throws Exception {
```

```
        JobConf conf = new JobConf(WordCount.class);
        conf.setJobName("wordcount");

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

        conf.setMapperClass(Map.class);
        conf.setCombinerClass(Reduce.class);
        conf.setReducerClass(Reduce.class);

        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);

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

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

5）将编写的程序打成名为 hadoop – wordcount 的 JAR 包，如图 12–6 所示。指定入口类为 cassandra. WordCount，如图 12–7 所示。

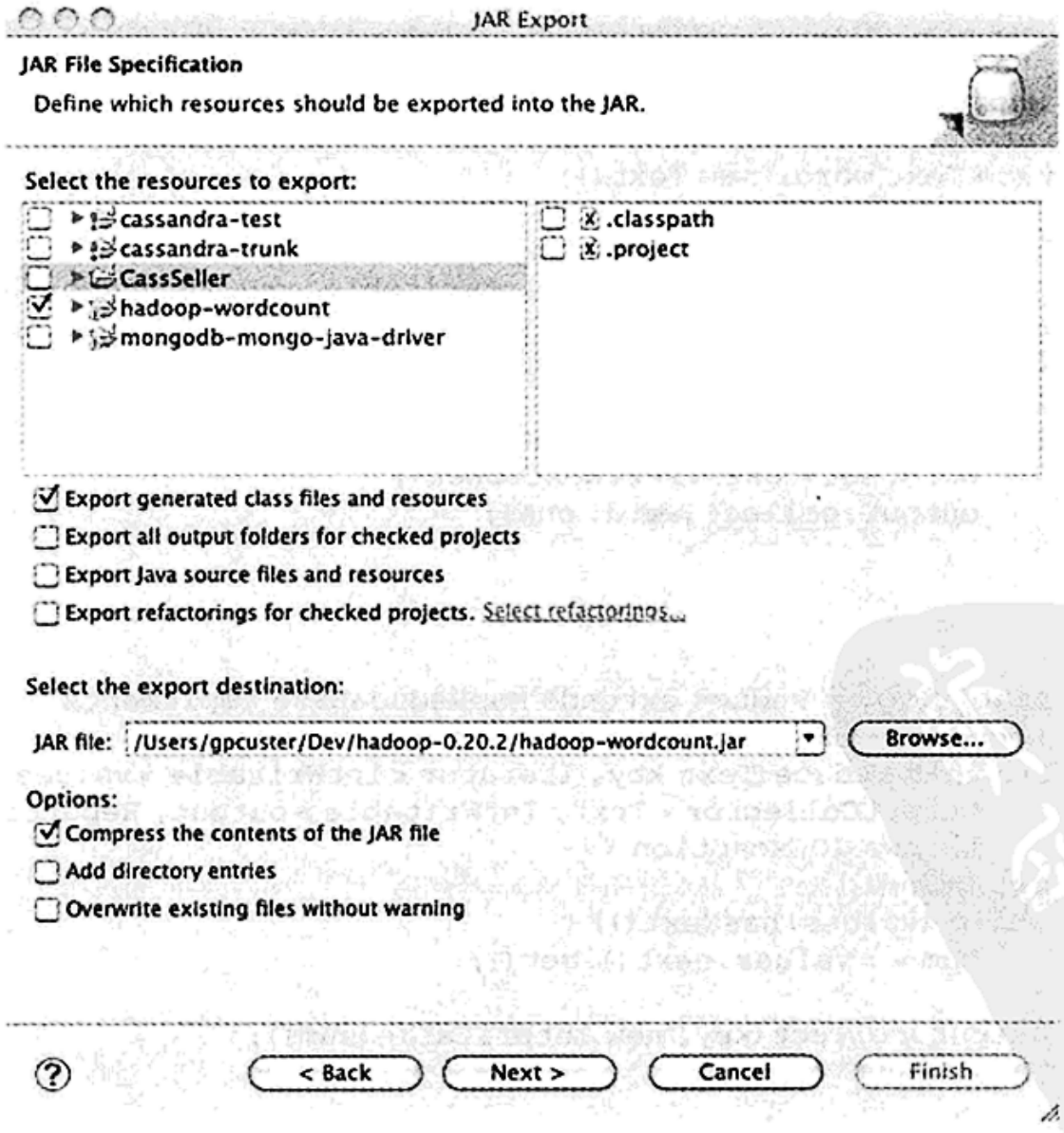


图 12–6　打包 hadoop – wordcount 程序

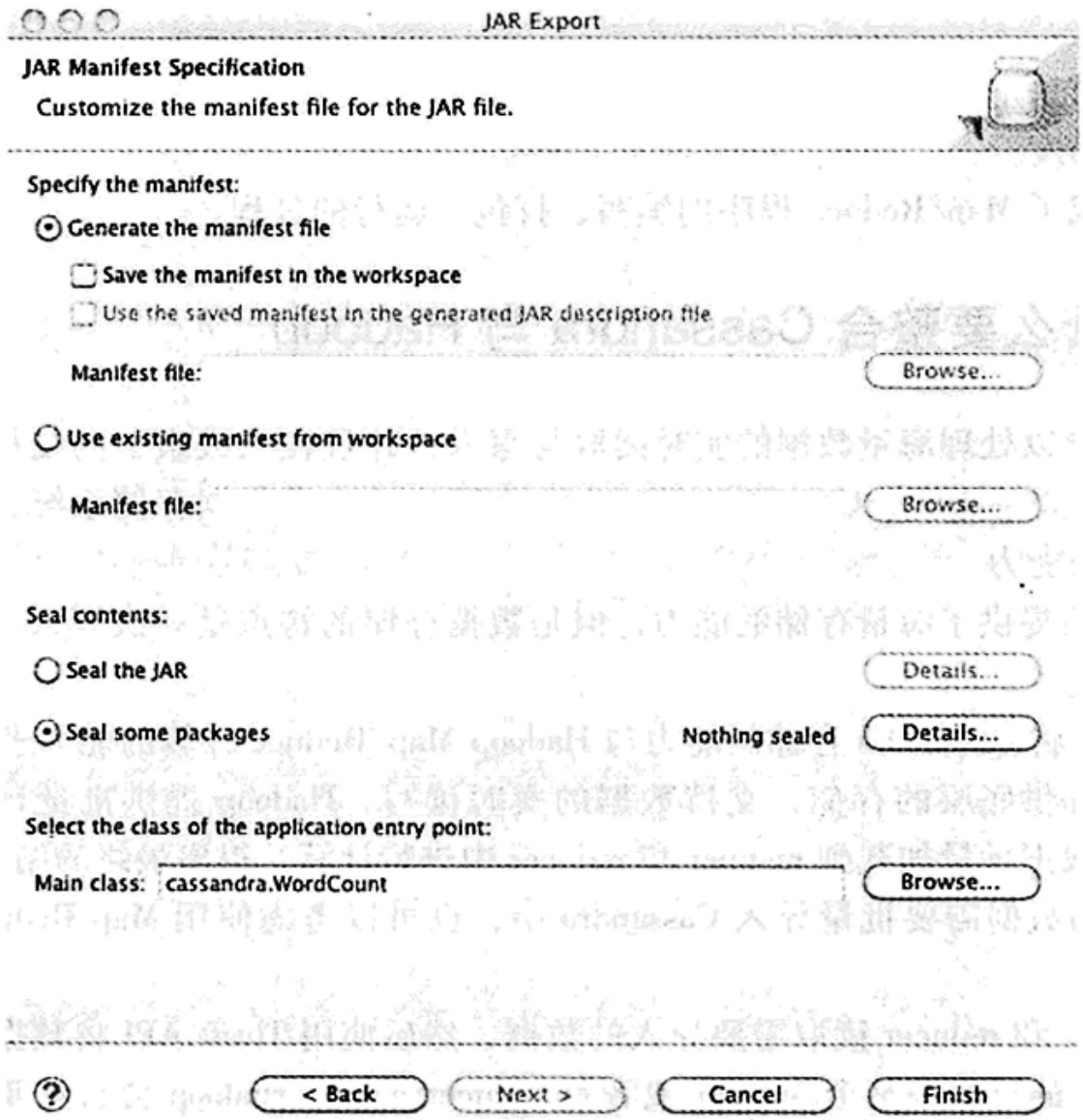


图 12-7　指定 hadoop - wordcount 程序入口类

这样，就完成了 Map/Reduce 应用程序的编写。执行完以上操作后，/Users/gpcuster/Dev/hadoop - 0.20.2 目录下就会新增一个 JAR 包 hadoop - wordcount.jar。

在/Users/gpcuster/Dev/hadoop - 0.20.2 目录下创建一个名为 wordcountinput.txt 的文件，文件中的内容如下：

```
hello
hadoop
pengguo
Cassandra
hadoop
hello
```

然后在/Users/gpcuster/Dev/hadoop - 0.20.2 目录下执行如下命令：

```
bin/hadoop jar hadoop - wordcount.jar wordcountinput.txt wordcountoutput
```

这样，我们编写的 WordCount 程序就运行起来了。执行成功后，可以在输出目录 wordcountoutput 中看到如下最终的执行结果：

```
Cassandra 1
hadoop 2
```

```
hello 2
pengguo 1
```

在输出的结果中，每一行记录代表一个单词出现的频度：左边是实际的单词，右边是该单词出现的频度。

这样就完成了 Map/Reduce 程序的编写、打包、运行的过程。

12.2 为什么要整合 Cassandra 与 Hadoop

Cassandra 可以处理海量数据的实时读取与写入，并且在二级索引的支持下，提供了对值的查询功能。总的来说，Cassandra 是一个非常不错的海量数据存储系统，但是缺乏了对数据进行分析的能力。

Hadoop 虽然提供了海量存储的能力，但是数据存储的特点是一次写入、多次读取，不支持数据的修改。

所以，可以将 Cassandra 存储的能力与 Hadoop Map/Reduce 计算的能力进行整合。

Cassandra 提供底层的存储，支持数据的实时读写，Hadoop 提供批量计算的能力，将 Cassandra 中的数据批量加载到 mapper 和 reducer 中进行计算，得出最终的结果。

如有大量的数据需要批量导入 Cassandra 中，也可以考虑使用 Map Reduce 进行批量导入。

使用 mapper 和 reducer 读取需要导入的数据，然后使用 Thrift API 将数据写入 Cassandra 中。如果在导入的过程中某个 mapper 或者是 reducer 失败，Hadoop 会自动重新运行失败的 mapper 或 reducer，并且 Cassandra 会自动处理重新导入的数据，所以使得整个数据的导入过程简单、高效和可靠。

以 12.1 节讲解的 WordCount 为例，系统需要记录每一个用户在搜索引擎上搜索的关键字，所以搜索的关键字数据实时写入 Cassandra 系统中，并且数据分析人员可以在任意时刻启动 Map/Reduce 分析程序，从 Cassandra 系统中读取所有的关键字数据进行统计分析。

12.3 使用 Map/Reduce 导入数据到 Cassandra 中

通过 Map/Reduce 程序，可以将 Hadoop 分布式文件系统中的海量文件批量导入 Cassandra 中。

输入的数据文件 wordcountinput. txt 内容如下：

```
user1 | hello
user1 | hadoop
user2 | pengguo
user1 | Cassandra
user1 | hadoop
user2 | hello
```

数据文件的每一行中，“ | ”符号前代表用户的名称，如 user1，“ | ”符号后代表用户

的搜索的关键字，如 hello。

Cassandra 用于存储用户搜索关键字的 ColumnFamily 可以采用 Standard 类型，Column 排序规则为 TimeUUIDType，Row 的 Key 为用户的名称，Column 的名称为值插入的时间（TimeUUID），Column 的值为搜索的关键字。该 ColumnFamily 在配置文件中可以定义如下：

```
- name: Keyspace1
  replica_placement_strategy: org.apache.cassandra.locator.SimpleStrategy
  replication_factor: 1
  column_families:
    - name: WordCount
      compare_with: TimeUUIDType
      keys_cached: 0
      rows_cached: 0
      row_cache_save_period_in_seconds: 0
      key_cache_save_period_in_seconds: 0
      memtable_flush_after_mins: 59
      memtable_throughput_in_mb: 64
      memtable_operations_in_millions: 1.0
```

接下来，开始编写 mapper 函数。

在 mapper 中，需要先初始化 Thrift 客户端，设置使用的 Keyspace，以将数据写入 Cassandra 服务器中，实现逻辑如下：

```
@ Override
public void configure(JobConf job) {
    super.configure(job);

    tr = new TFramedTransport(new TSocket("localhost", 9160));
    TProtocol proto = new TBinaryProtocol(tr);
    cassandraClient = new Cassandra.Client(proto);

    try {
        tr.open();
        cassandraClient.set_keyspace("Keyspace1");
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

初始化 Thrift 客户端后，在 mapper 函数中拆分输入数据，找出用户名和对应的搜索关键字，构建需要插入 Cassandra 的 Column，实现逻辑如下：

```
public void map(LongWritable key, Text value,
        OutputCollector<Text, IntWritable> output, Reporter reporter)
        throws IOException {
    String line = value.toString();

    int splitIndex = line.indexOf("|");

    if (splitIndex > 1 && splitIndex < line.length()) {
        String userName = line.substring(0, splitIndex);
        String searchKeyword = line.substring(splitIndex + 1);

        ColumnParent cp = new ColumnParent();
```

```
        cp.column_family = "WordCount";
        Column c = new Column();
        c.name = ByteBuffer.wrap(UUIDGenerator.getInstance()

                .generateTimeBasedUUID().toByteArray());
        c.timestamp = System.currentTimeMillis();
        c.value = ByteBuffer.wrap(searchKeyword.getBytes("utf-8"));

        try {
            cassandraClient.insert(
                    ByteBuffer.wrap(userName.getBytes("utf-8")), cp, c,
                                                    ConsistencyLevel.ONE);
                        } catch (Exception e) {
                    e.printStackTrace();
        }
    }
}
```

在 mapper 函数执行完毕后，还需要将 Thrift 客户端关闭，释放占用的资源。

```
@ Override
public void close() throws IOException {
    super.close();

    if (tr != null) {
        tr.close();
    }
}
```

这个程序只需要 Map 操作即可，所以在 Map/Reduce 程序的运行设置中将 Reduce 的个数设置为 0，这样在运行时就不会执行 Reduce 流程了。

```
conf.setNumReduceTasks(0);
```

完成代码的编写后，完整的 WordCount. java 代码如下：

```
package cassandra;

import java.io.IOException;
import java.nio.ByteBuffer;

import org.apache.cassandra.thrift.Cassandra;
import org.apache.cassandra.thrift.Cassandra.Client;
import org.apache.cassandra.thrift.Column;
import org.apache.cassandra.thrift.ColumnParent;
import org.apache.cassandra.thrift.ConsistencyLevel;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobClient;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.OutputCollector;
```

```
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.TextOutputFormat;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.protocol.TProtocol;
import org.apache.thrift.transport.TFramedTransport;
import org.apache.thrift.transport.TSocket;
import org.apache.thrift.transport.TTransport;
import org.safehaus.uuid.UUIDGenerator;

public class WordCount {

    public static class Map extends MapReduceBase implements
            Mapper<LongWritable, Text, Text, IntWritable> {
        Client cassandraClient = null;
        TTransport tr = null;

        @ Override
        public void configure(JobConf job) {
            super.configure(job);

            tr = new TFramedTransport(new TSocket("localhost", 9160));
            TProtocol proto = new TBinaryProtocol(tr);
            cassandraClient = new Cassandra.Client(proto);

            try {
                tr.open();

                cassandraClient.set_keyspace("Keyspace1");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        @ Override
        public void close() throws IOException {
            super.close();

            if (tr != null) {
                tr.close();
            }
        }

        public void map(LongWritable key, Text value,
            OutputCollector<Text, IntWritable> output, Reporter reporter)
            throws IOException {
            String line = value.toString();

            int splitIndex = line.indexOf("|");

            if (splitIndex > 1 && splitIndex < line.length()) {
                String userName = line.substring(0, splitIndex);
                String searchKeyword = line.substring(splitIndex + 1);

                ColumnParent cp = new ColumnParent();
```

```
                cp.column_family = "WordCount";
                Column c = new Column();
                c.name = ByteBuffer.wrap(UUIDGenerator.getInstance()

                        .generateTimeBasedUUID().toByteArray());
                c.timestamp = System.currentTimeMillis();
                c.value = ByteBuffer.wrap(searchKeyword.getBytes("utf-8"));

                                                    try {
                                                    cassandraClient.insert(

                        ByteBuffer.wrap(userName.getBytes("utf-8")), cp, c,

                        ConsistencyLevel.ONE);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }

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

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

        conf.setMapperClass(Map.class);
        conf.setNumReduceTasks(0);

        conf.setInputFormat(TextInputFormat.class);
        conf.setOutputFormat(TextOutputFormat.class);

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

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

最后将代码打成 JAR 包，名字为 wordcount – input. jar，入口类为 cassandra. WordCount。

下面就可以开始对这个 Map/Reduce 程序进行测试了。

首先，启动 Cassandra。

启动 Cassandra 后，执行如下命令导入新增加的 ColumnFamily 信息：

```
sh bin/schematool localhost 8080 import
```

然后，将 Cassandra 安装目录的 lib 目录下的 3 个 jar 包复制到每一台执行 Map/Reduce 程序的 Hadoop 安装目录的 lib 目录下。这 3 个 jar 包如下：

- apache – cassandra – 0. 7. 0 – rc1. jar
- jug – 2. 0. 0. jar

- libthrift – 0. 5. jar

输入文件为 wordcountinput. txt，执行如下命令启动 Map/Reduce 任务：

```
sh bin/hadoop jar wordcountinput.jar wordcountinput.txt wordcountoutput
```

程序执行过程中，输入的 Map/Reduce 执行信息如下：

```
10/11/20 15:48:51 INFO mapred.JobClient: map 100%  reduce 0%
10/11/20 15:48:51 INFO mapred.JobClient: Job complete: job_local_0001
10/11/20 15:48:51 INFO mapred.JobClient: Counters: 6
10/11/20 15:48:51 INFO mapred.JobClient: FileSystemCounters
10/11/20 15:48:51 INFO mapred.JobClient: FILE_BYTES_READ = 17415
10/11/20 15:48:51 INFO mapred.JobClient: FILE_BYTES_WRITTEN = 31094
10/11/20 15:48:51 INFO mapred.JobClient: Map - Reduce Framework
10/11/20 15:48:51 INFO mapred.JobClient: Map input records = 6
10/11/20 15:48:51 INFO mapred.JobClient: Spilled Records = 0
10/11/20 15:48:51 INFO mapred.JobClient: Map input bytes = 80
10/11/20 15:48:51 INFO mapred.JobClient: Map output records = 0
```

这样，使用 Map/Reduce 程序导入数据的测试就通过了。

当然，上面的程序仅仅是一个示例，在实际应用中应该考虑更多的细节，如以下几点：

- 在 mapper 或者 reducer 初始化的时候，可以随机连接一台 Cassandra 服务器，这样有助于平衡负载。
- 在 mapper 或者 reducer 中将数据导入 Cassandra 的时候，可以增加自动重试功能。比如一次导入失败了，可以让该线程暂停（Sleep）10s，然后再执行写入操作。
- 在 mapper 或者 reducer 中将数据导入 Cassandra 的时候，可以考虑采取批量写入的方式，比如 100 个 Column 一次性批量写入。

12. 4　将 Cassandra 中的数据作为 Map/Reduce 输入

在运行 Hadoop Map/Reduce 程序的过程中，可以将 Cassandra 中某个 Keyspace 下的 Column 中的所有数据作为 Map/Reduce 的数据输入。

这次，我们编写一个 Map/Reduce 程序，从 Cassandra 中读取 Keyspace1 下名为 WordCount 的 ColumnFamily 的数据，分析每一个关键词被搜索的次数。

1）编写 mapper 函数，mapper 函数中输入的数据为 Row 的 Key，以及该 Key 对应的所有 Column。在关键词中的统计中，只需要关注每一个 Column 的 Value 即可。

```
public static class TokenizerMapper
        extends
        Mapper<ByteBuffer, SortedMap<ByteBuffer, IColumn>, Text, IntWritable>{
    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    public void map(ByteBuffer key, SortedMap<ByteBuffer, IColumn> columns,
            Context context) throws IOException, InterruptedException {

        for (IColumn column : columns.values()) {
            if (column == null) {
```

```
                continue;
            }

            String value = ByteBufferUtil.string(column.value());
            word.set(value);
            context.write(word, one);
        }
    }
}
```

2）编写 reducer 函数，统计每一个搜索关键字出现的次数。

```
public static class ReducerToFilesystem extends
        Reducer<Text, IntWritable, Text, IntWritable> {
    private IntWritable result = new IntWritable();

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

        result.set(sum);
        context.write(key, result);
    }
}
```

3）配置 Map/Reduce 程序的运行配置信息，使用 ColumnFamilyInputFormat 作为输入的数据格式，同时指定获取 Keyspace1 的所有 ColumnFamily 为 WordCount 的数据。

```
job.setInputFormatClass(ColumnFamilyInputFormat.class);

ConfigHelper.setRpcPort(job.getConfiguration(), "9160");
ConfigHelper.setInitialAddress(job.getConfiguration(), "localhost");
ConfigHelper.setPartitioner(job.getConfiguration(),
        "org.apache.cassandra.dht.RandomPartitioner");
ConfigHelper.setInputColumnFamily(job.getConfiguration(), KEYSPACE,
        COLUMN_FAMILY);
SliceRange range = new SliceRange();
range.start = ByteBuffer.wrap(new byte[] {});
range.finish = ByteBuffer.wrap(new byte[] {});
range.reversed = false;
range.count = Integer.MAX_VALUE;
SlicePredicate predicate = new SlicePredicate().setSlice_range(range);
ConfigHelper.setInputSlicePredicate(job.getConfiguration(), predicate);
```

完成代码的编写后，完整的 WordCount. java 代码如下：

```
package cassandra;

import java.io.IOException;
import java.nio.ByteBuffer;
import java.util.SortedMap;

import org.apache.cassandra.db.IColumn;
import org.apache.cassandra.hadoop.ColumnFamilyInputFormat;
```

```
import org.apache.cassandra.hadoop.ConfigHelper;
import org.apache.cassandra.thrift.SlicePredicate;
import org.apache.cassandra.thrift.SliceRange;
import org.apache.cassandra.utils.ByteBufferUtil;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.conf.Configured;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.util.Tool;
import org.apache.hadoop.util.ToolRunner;

import cassandra.WordCount.ReducerToFilesystem;
import cassandra.WordCount.TokenizerMapper;

public class WordCount extends Configured implements Tool {

    static final String KEYSPACE = "Keyspace1";
    static final String COLUMN_FAMILY = "WordCount";

    public static void main(String[] args) throws Exception {
        ToolRunner.run(new Configuration(), new WordCount(), args);
        System.exit(0);
    }

    public static class TokenizerMapper
            extends
            Mapper<ByteBuffer, SortedMap<ByteBuffer, IColumn>, Text, IntWrit-
able> {
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        public void map(ByteBuffer key, SortedMap<ByteBuffer, IColumn> columns,
                Context context) throws IOException, InterruptedException {

            for (IColumn column : columns.values()) {
                if (column == null) {
                    continue;
                }

                String value = ByteBufferUtil.string(column.value());
                word.set(value);
                context.write(word, one);
            }
        }
    }

    public static class ReducerToFilesystem extends
            Reducer<Text, IntWritable, Text, IntWritable> {
        private IntWritable result = new IntWritable();
```

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

        result.set(sum);
        context.write(key, result);
    }
}

public int run(String[] args) throws Exception {
    Job job = new Job(getConf(), "wordcount");

    job.setJarByClass(WordCount.class);
    job.setMapperClass(TokenizerMapper.class);

job.setReducerClass(ReducerToFilesystem.class);
    job.setOutputKeyClass(Text.class);
    job.setOutputValueClass(IntWritable.class);
    FileOutputFormat.setOutputPath(job, new Path(args[0]));

job.setInputFormatClass(ColumnFamilyInputFormat.class);

ConfigHelper.setRpcPort(job.getConfiguration(), "9160");
ConfigHelper.setInitialAddress(job.getConfiguration(), "localhost");
ConfigHelper.setPartitioner(job.getConfiguration(),
          "org.apache.cassandra.dht.RandomPartitioner");
ConfigHelper.setInputColumnFamily(job.getConfiguration(), KEYSPACE,
          COLUMN_FAMILY);
        SliceRange range = new SliceRange();
        range.start = ByteBuffer.wrap(new byte[] {});
        range.finish = ByteBuffer.wrap(new byte[] {});
        range.reversed = false;
        range.count = Integer.MAX_VALUE;
        SlicePredicate predicate = new SlicePredicate().setSlice_range(range);
    ConfigHelper.setInputSlicePredicate(job.getConfiguration(), predicate);

        job.waitForCompletion(true);

        return 0;
    }
}
```

最后将代码打成 JAR 包，名字为 wordcount. jar，入口类为 cassandra. WordCount。

下面就可以开始对这个 Map/Reduce 程序进行测试了。

首先，保证 Cassandra 已经启动，并且导入了前面使用的数据。

然后，将 Cassandra 安装目录的 lib 目录下的两个 jar 包复制到每一台执行 Map/Reduce 程序的 Hadoop 安装目录的 lib 目录下。这两个 jar 包如下：

- commons – lang – 2. 4. jar
- guava – r05. jar

最后，执行如下命令启动 Map/Reduce 任务：

```
sh bin/hadoop jar wordcount.jar output
```

程序执行过程中，输入的 Map/Reduce 执行信息如下：

```
10/11/21 00:48:39 INFO mapred.JobClient: Job complete: job_local_0001
10/11/21 00:48:39 INFO mapred.JobClient: Counters: 12
10/11/21 00:48:39 INFO mapred.JobClient: FileSystemCounters
10/11/21 00:48:39 INFO mapred.JobClient: FILE_BYTES_READ=38194
10/11/21 00:48:39 INFO mapred.JobClient: FILE_BYTES_WRITTEN=67121
10/11/21 00:48:39 INFO mapred.JobClient: Map-Reduce Framework
10/11/21 00:48:39 INFO mapred.JobClient: Reduce input groups=4
10/11/21 00:48:39 INFO mapred.JobClient: Combine output records=0
10/11/21 00:48:39 INFO mapred.JobClient: Map input records=2
10/11/21 00:48:39 INFO mapred.JobClient: Reduce shuffle bytes=0
10/11/21 00:48:39 INFO mapred.JobClient: Reduce output records=4
10/11/21 00:48:39 INFO mapred.JobClient: Spilled Records=12
10/11/21 00:48:39 INFO mapred.JobClient: Map output bytes=68
10/11/21 00:48:39 INFO mapred.JobClient: Combine input records=0
10/11/21 00:48:39 INFO mapred.JobClient: Map output records=6
10/11/21 00:48:39 INFO mapred.JobClient: Reduce input records=6
```

程序执行完毕后，打开结果文件，可以看到最终计算的结果如下：

```
Cassandra 1
hadoop 2
hello 2
pengguo 1
```

这说明程序执行正确了。

12.5　本章小结

本章对 Hadoop 做了概要的介绍，分析了如何快速搭建与编写 Hadoop Map/Reduce 程序。

通过搜索关键字统计的应用，使读者了解如何将 Cassandra 与 Hadoop 进行整合，使得 Cassandra 具备大批量数据分析的能力。

第 13 章

Cassandra 最佳实践

本章内容

- 避免 Cassandra 自身的限制
- 数据压缩策略
- 使用高级的客户端
- 负载均衡
- 谨慎使用二级索引
- 通过 JMX 监测 Cassandra
- 调整 JVM 启动参数
- 使用适合的系统配置参数
- 本章小结

在之前的章节中，对 Cassandra 进行了全面和系统的介绍。在这一章中，将讲解 Cassandra 实际应用中的最佳实践。

13.1 避免 Cassandra 自身的限制

对于任何软件，没有所谓的“最好”，只有相对的“最适合”。

Cassandra 也是如此，在实际的使用中，需要避免 Cassandra 系统在设计和实现的过程中包含的限制。在 Cassandra 0.7 的版本中，应注意的问题如下。

13.1.1 不要盲目使用 Super Column

Cassandra 将客户端插入的数据写入 SSTable 文件中时，会对每一个 Key 对应的所有 Column 的名称建立索引，所以，如果某一个 Key 中包含了大量的 Column，那么这个索引就可以极大地提高对 Column 查找的速度。但是对于 Super 类型的 ColumnFamily，Cassandra 只会对 Super Column 的名称建立索引，当查找某一个 Super Column 下的 Column 时，就没有索引可以使用，需要依次遍历所有的 Column，直到找到所有合适的 Column 为止。如果某个 Super Column 下有大量的 Column，那么读取这个 Super Column 下的某个 Column 就将耗费大量的时间。

所以在设计 Cassandra 的数据模型时，不要盲目使用 Super Column，要仔细考虑项目的实际数据情况，如果采用 Super Column 后，在 Super Column 中将存在大量的 Column，就需要考虑是否采取另外一种思路来设计 Cassandra 的数据模型了。

要解决这个问题，需要一种新的数据存储格式，这个问题将会在未来的 Cassandra 发行版本（Cassandra 0.8.x）中解决。

13.1.2 硬盘的容量大小限制

Cassandra 中每一个 Key 对应的所有数据都是需要完整保存在一个 SSTable 文件中的，即一块硬盘中。如果某一个 Key 对应的数据超过了这个大小限制，系统会出现硬盘空间不足的错误异常。

13.1.3 注意系统大小限制

在 Cassnadra 中，会校验以下信息，保证数据的大小符合系统的要求：

- Key 的长度不能超过 65 535 个字节。
- Super Column 的名称和 Column 的名称长度不能超过 65 535 个字节。
- Column 对应值的长度不能超过 2GB。

由于 Cassnadra 使用 Thrift 与客户端进行通信，所以每次通信的数据都需要放在内存中，如果通信的数据大小超过了内存的限制，将出现内存不足的异常。

13.2　数据压缩策略

在实际的使用中，Cassandra 频繁地进行数据压缩会导致系统出现不稳定。原因是数据压缩将消耗大量的磁盘 I/O 和内存。如果关闭数据压缩功能，将导致数据文件夹下出现大量的 SSTable 文件，占用过多的磁盘空间，同时降低读取的效率。

所以，使用合理的压缩策略能有效地提高集群的稳定性和性能。

由于数据压缩操作是在后台与读写操作同步进行的，可以在配置文件中将数据压缩线程的优先级设置为最低。

```
compaction_thread_priority: 1
```

对于数据压缩特别频繁的 ColumnFamily，可以在配置文件中适当调节数据从 Memtable 中到导出 SSTable 中的限制条件，从而减少生成的 SSTable 的数量，如：

```
memtable_flush_after_mins: 120
memtable_throughput_in_mb: 256
memtable_operations_in_millions: 10.0
```

另外，还可以考虑关闭自动数据压缩操作，等到系统相对空闲的时刻，进行手工压缩操作，从而避免数据压缩操作对系统繁忙时候的影响。比如在系统中配置一个 crontab 的任务，在每天凌晨 1 点自动启动执行 Cassandra 数据压缩的操作。

13.3　使用高级的客户端

Cassandra 使用了 Facebook 开源的跨语言网络通信框架 Thrift，编写 C/S 程序只需要专注于 Server 端的逻辑即可，客户端与服务器端的通信协议只需要指定一个接口文件，Thrift 会自动生成不同语言的代码，比如 C + + 、Java、Python 等。Cassandra 中定义的客户端与服务器端的通信协议接口文件为 cassandra. thrift。该文件在 interface 目录下。

Cassandra 的 JAR 包中包含了与 Cassandra 进行通信的 Java 语言的 Thrift 客户端，如果希望使用其他语言版本的 Thrift 客户端，可以直接通过 cassandra. thrift 文件自动生成。

由于与 Cassandra 通信的客户端都是由 Thrift 自动生成的，所以其所具备的功能也是非常有限的。比如在其他编程框架中常用的连接池、线程安全等问题都需要客户端额外考虑。所以开源社区提供了不同编程语言的高级 Cassandra 客户端，它们都是以 Thrift 客户端为基础，额外提供了连接池、线程安全、错误处理等特性。

13.3.1　Pycassa

Pycassa 的官方网址为 http://github. com/pycassa/pycassa，Pycassa 是 Cassandra 的 Python 客户端，它提供的特性如下：

1）连接失败的时候，能够自动恢复。
2）提供连接池。
3）提供批量操作的接口。
4）提供比 Thrift 更加简便的接口。
5）提供 Cassandra 中 ColumnFamily 的映射方法，编程更加方便。

使用的范例代码如下：

```
import threading
import unittest

from nose.tools import assert_raises
from pycassa import connect, connect_thread_local
from pycassa.cassandra.ttypes import CfDef, KsDef

class ConnectionCase(unittest.TestCase):
    def test_connections(self):
        def version_check(connection, version):
            assert connection.describe_version() == version

        version = connect('Keyspace1').describe_version()

        thread_local = connect_thread_local('Keyspace1')
        threads = []
        for i in xrange(10):

threads.append(threading.Thread(target = version_check,
                                        args = (thread_local, version)))
            threads[-1].start()
        for thread in threads:
            thread.join()

        def test_api_version_check(self):
          import pycassa.connection
          pycassa.connection.API_VERSION = ['FOO']
          try:
              assert_raises(AssertionError, connect('Keyspace1').describe_version)
          finally:
              reload(pycassa.connection)

        def test_system_calls(self):
          conn = connect('Keyspace1')

          # keyspace modifications
          try:
              conn.drop_keyspace('TestKeyspace')
```

```
  except:
      pass
conn.add_keyspace(KsDef('TestKeyspace',
    'org.apache.cassandra.locator.SimpleStrategy', None, 3, []))
conn.update_keyspace(KsDef('TestKeyspace',
    'org.apache.cassandra.locator.SimpleStrategy', None, 2,[]))
conn.drop_keyspace('TestKeyspace')

# column family modifications
try:
    conn.drop_column_family('TestCF')
except:
    pass
conn.add_column_family(CfDef('Keyspace1', 'TestCF', 'Standard'))
cfdef = conn.get_keyspace_description()['TestCF']
cfdef.comment = 'this is a test'
conn.update_column_family(cfdef)
conn.drop_column_family('TestCF')
```

13.3.2　Hector

Hector 是 Cassandra 的 Java 客户端[⊖]，它提供的特性如下：

1）提供高度抽象的 Cassandra 编程接口。

2）提供错误处理。

3）提供连接池。

4）提供 JMX 监控和管理功能。

5）支持负载均衡。

使用的范例代码如下：

```
package me.prettyprint.cassandra.examples;

import static me.prettyprint.hector.api.factory.HFactory.createColumn;
import static me.prettyprint.hector.api.factory.HFactory.createColumnQuery;
import static me.prettyprint.hector.api.factory.HFactory.createKeyspace;
import static me.prettyprint.hector.api.factory.HFactory.createMultigetSliceQuery;
import static me.prettyprint.hector.api.factory.HFactory.createMutator;
import static me.prettyprint.hector.api.factory.HFactory.getOrCreateCluster;

import java.util.HashMap;
import java.util.Map;
```

⊖ Hector 的官方网址为 http://github.com/rantav/hector。

```
import me.prettyprint.cassandra.serializers.StringSerializer;
import me.prettyprint.hector.api.Cluster;
import me.prettyprint.hector.api.Keyspace;
import me.prettyprint.hector.api.Serializer;
import me.prettyprint.hector.api.beans.HColumn;
import me.prettyprint.hector.api.beans.Rows;
import me.prettyprint.hector.api.exceptions.HectorException;
import me.prettyprint.hector.api.mutation.Mutator;
import me.prettyprint.hector.api.query.ColumnQuery;
import me.prettyprint.hector.api.query.MultigetSliceQuery;
import me.prettyprint.hector.api.query.QueryResult;

/**
 * Thread Safe
 * @author Ran Tavory
 *
 */
public class ExampleDaoV2 {

 private final static String KEYSPACE = "Keyspace1";
 private final static String HOST_PORT = "localhost:9170";
 private final static String CF_NAME = "Standard1";
 /** Column name where values are stored */
 private final static String COLUMN_NAME = "v";
 private final StringSerializer serializer = StringSerializer.get();

 private final Keyspace keyspace;

 public static void main(String[] args) throws HectorException {
    Cluster c = getOrCreateCluster("MyCluster", HOST_PORT);
    ExampleDaoV2 ed = new ExampleDaoV2(createKeyspace(KEYSPACE, c));
    ed.insert("key1", "value1", StringSerializer.get());

    System.out.println(ed.get("key1", StringSerializer.get()));
 }

 public ExampleDaoV2(Keyspace keyspace) {
    this.keyspace = keyspace;
 }

/**
 * Insert a new value keyed by key
 *
 * @param key Key for the value
 * @param value the String value to insert
 */
```

```
 public <K> void insert(final K key, final String value, Serializer<K> keySer-
ializer) {
  createMutator(keyspace, keySerializer).insert(
    key, CF_NAME, createColumn(COLUMN_NAME, value, serializer, serializer));
 }

 /**
  * Get a string value.
  *
  * @return The string value; null if no value exists for the given key.
  */
 public <K> String get(final K key, Serializer<K> keySerializer) throws Hect-
orException {
    ColumnQuery<K, String, String> q = createColumnQuery(keyspace, keySeriali-
zer, serializer, serializer);
    QueryResult<HColumn<String, String>> r = q.setKey(key).
      setName(COLUMN_NAME).
      setColumnFamily(CF_NAME).
      execute();
    HColumn<String, String> c = r.get();
    return c == null ? null : c.getValue();
 }

 /**
  * Get multiple values
  * @param keys
  * @return
  */
 public <K> Map<K, String> getMulti(Serializer<K> keySerializer, K...
keys) {
  MultigetSliceQuery<K, String,String> q = createMultigetSliceQuery(keyspace,
keySerializer, serializer, serializer);
  q.setColumnFamily(CF_NAME);
  q.setKeys(keys);
  q.setColumnNames(COLUMN_NAME);

  QueryResult<Rows<K, String,String>> r = q.execute();
  Rows<K, String,String> rows = r.get();
  Map<K, String> ret = new HashMap<K, String>(keys.length);
  for (K k: keys) {
    HColumn<String,String> c = rows.getByKey(k).getColumnSlice().getColumnBy
Name(COLUMN_NAME);
    if (c != null && c.getValue() != null) {
      ret.put(k, c.getValue());
    }
  }
```

```
    return ret;
  }

  /**
   * Insert multiple values
   */
  public <K> void insertMulti(Map<K, String> keyValues, Serializer<K> keySer-
ializer) {
    Mutator<K> m = createMutator(keyspace, keySerializer);
    for (Map.Entry<K, String> keyValue: keyValues.entrySet()) {
        m.addInsertion(keyValue.getKey(), CF_NAME,
            createColumn(COLUMN_NAME, keyValue.getValue(), keyspace.createClock(),
serializer, serializer));
    }
    m.execute();
  }

  /**
   * Delete multiple values
   */
  public <K> void delete(Serializer<K> keySerializer, K... keys) {
    Mutator<K> m = createMutator(keyspace, keySerializer);
    for (K key: keys) {
        m.addDeletion(key, CF_NAME, COLUMN_NAME, serializer);
    }
    m.execute();
  }
}
```

13.3.3 FluentCassandra

FluentCassandra 的官方网址为 http://github.com/managedfusion/fluentcassandra，FluentCassandra 是 Cassandra 的 .Net 客户端，它支持 .Net 4.0 的动态关键字和 LINQ 表达式查询。

使用的范例代码如下：

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using FluentCassandra.Types;
using Apache.Cassandra;
using FluentCassandra.Connections;

namespace FluentCassandra.Test
{
    internal class _CassandraSetup
```

```
{
    public CassandraContext DB;
    public CassandraColumnFamily<AsciiType> Family;
    public CassandraSuperColumnFamily<AsciiType, AsciiType> SuperFamily;

    public const string TestKey1 = "Test1";
    public const string TestKey2 = "Test2";

    public const string TestStandardName = "Test1";
    public const string TestSuperName = "SubTest1";

    public _CassandraSetup()
    {
        var keyspaceName = "Testing";
        var server = new Server("localhost");

        if (!CassandraSession.KeyspaceExists(server, keyspaceName))

    CassandraSession.AddKeyspace(server, new KsDef {
              Name = "Testing",
              Replication_factor = 1,
              Strategy_class = "org.apache.cassandra.locator.SimpleStrategy",
              Cf_defs = new List<CfDef>()
          });

    var keyspace = new CassandraKeyspace(keyspaceName);
    if (!keyspace.ColumnFamilyExists(server, "Standard"))
        keyspace.AddColumnFamily(server, new CfDef {
            Name = "Standard",
            Keyspace = "Testing",
            Column_type = "Standard",
            Comparator_type = "AsciiType",
            Comment = "Used for testing Standard family."
        });

    DB = new CassandraContext(keyspaceName, server);
    DB.ThrowErrors = true;

    Family = DB.GetColumnFamily<AsciiType>("Standard");
    SuperFamily = DB.GetColumnFamily<AsciiType, AsciiType>("Super");

    Family.RemoveAllRows();
    SuperFamily.RemoveAllRows();

    Family.InsertColumn(TestKey1, "Test1", Math.PI);
    Family.InsertColumn(TestKey1, "Test2", Math.PI);
```

```
            Family.InsertColumn(TestKey1, "Test3", Math.PI);

            SuperFamily.InsertColumn(TestKey1, TestSuperName, "Test1", Math.PI);
            SuperFamily.InsertColumn(TestKey1, TestSuperName, "Test2", Math.PI);
            SuperFamily.InsertColumn(TestKey1, TestSuperName, "Test3", Math.PI);

            Family.InsertColumn(TestKey2, "Test1", Math.PI);
            Family.InsertColumn(TestKey2, "Test2", Math.PI);
            Family.InsertColumn(TestKey2, "Test3", Math.PI);

            SuperFamily.InsertColumn(TestKey2, TestSuperName, "Test1", Math.PI);
            SuperFamily.InsertColumn(TestKey2, TestSuperName, "Test2", Math.PI);
            SuperFamily.InsertColumn(TestKey2, TestSuperName, "Test3", Math.PI);
        }
    }
}
```

13.3.4 Cassandra

Cassandra 的官方网址为 http://github.com/fauna/cassandra，Cassandra 是 Cassandra 的 Ruby 客户端。

使用的范例代码如下：

```
require File.expand_path(File.dirname(__FILE__) + '/test_helper')

class CassandraClientTest < Test::Unit::TestCase
  include Cassandra::Constants

  def setup
    @twitter = Cassandra.new('Twitter', "127.0.0.1:9160", :retries => 2, :exception_classes => [])
  end

  def test_client_method_is_called
    assert_nil @twitter.instance_variable_get(:@client)
    @twitter.insert(:Statuses, key, {'1' => 'v', '2' => 'v', '3' => 'v'})
    assert_not_nil @twitter.instance_variable_get(:@client)
  end

  def key
    caller.first[/'(.*?)'/, 1]
  end

end
```

13.3.5　phpcassa

phpcassa 的官方网址为 http://github.com/thobbs/phpcassa，phpcassa 是 Cassandra 的 PHP 客户端，它提供的特性与 Pycassa 类似。

使用的范例代码如下：

```
<?php
require_once('simpletest/autorun.php');
require_once('../connection.php');
require_once('../columnfamily.php');

class TestColumnFamily extends UnitTestCase {

    private $client;
    private $cf;

    private static $KEYS = array('key1', 'key2','key3');

    public function setUp() {
        $this->client = new Connection('Keyspace1');
        $this->cf = new ColumnFamily($this->client, 'Standard1');
    }

    public function tearDown() {
        foreach(self::$KEYS as $key)
            $this->cf->remove($key);
    }

    public function test_opening_connection() {
        $this->client->connect();
    }

    public function test_empty() {
        try {
            $this->cf->get(self::$KEYS[0]);
            self::asertTrue(false);
        } catch (cassandra_NotFoundException $e) {
        }
    }

    public function test_insert_get() {
        $this->cf->insert(self::$KEYS[0], array('col'=>'val'));
        self::assertEqual($this->cf->get(self::$KEYS[0]), array('col'=>
'val'));
    }
```

```
public function test_insert_multiget() {
    $columns1 = array('1' => 'val1', '2' => 'val2');
    $columns2 = array('3' => 'val1', '4' => 'val2');
    $this->cf->insert(self::$KEYS[0], $columns1);
    $this->cf->insert(self::$KEYS[1], $columns2);
    $rows = $this->cf->multiget(self::$KEYS);
    self::assertEqual(count($rows), 2);
    self::assertEqual($rows[self::$KEYS[0]], $columns1);
    self::assertEqual($rows[self::$KEYS[1]], $columns2);
    self::assertFalse(in_array(self::$KEYS[2], $rows));
}

public function test_batch_insert() {
    $columns1 = array('1' => 'val1','2' => 'val2');
    $columns2 = array('3' => 'val1','4' => 'val2');
    $rows = array(self::$KEYS[0] => $columns1,
          self::$KEYS[1] => $columns2);
    $this->cf->batch_insert($rows);
    $rows = $this->cf->multiget(self::$KEYS);
    self::assertEqual(count($rows), 2);
    self::assertEqual($rows[self::$KEYS[0]], $columns1);
    self::assertEqual($rows[self::$KEYS[1]], $columns2);
    self::assertFalse(in_array(self::$KEYS[2], $rows));
}
}
?>
```

13.4 负载均衡

假设 Cassandra 集群由 10 台服务器构成，现在有 20 个客户端要进行读写操作，那么如何才能做到让集群中的 10 台服务器负载相对均衡呢？可以采用下面两种方式。

13.4.1 随机选取

每一个客户端从这 10 台服务器中随机选取一台服务器作为读写操作的对象。如果发现读写失败，则从这 10 台服务器中再随机选取一台服务器进行读写。

13.4.2 缓存集群信息

Cassandra 的内部读写流程如下：

1）客户端将读写请求发送给一台服务器。

2）接收到读写请求的服务器会判断需要查询的数据是否在本机中，如果在本机中，直接操作后应答；如果不在本机中，将请求转发给另外一台服务器来查询，并等待另外一台服务器的应答结果，并将这个应答结果返回给客户端。

可以看到，在上面的这个查询流程中，如果客户端将需要查询的数据发送给了一台不相关的 Cassandra 服务器，那么这个不相关的 Cassandra 服务器就充当了一个代理的角色。

最坏的情况是：所有的请求都发送到同一台 Cassandra 服务器，那么这台 Cassandra 服务器就成为了整个集群的瓶颈。

理想的状态是：客户端将查询的请求直接发送到存储有需要查询数据的 Cassandra 服务器中，这样就省掉了其中一层不必要的代理。

那么如何达到这种理想的状态呢？可以考虑使用 Cassandra 中自带的 RingCache。

使用 RingCache 非常简单，只需要编写如下代码即可获得负责某一个数据的实际服务器：

```
ringCache = new RingCache();
List < InetAddress > endPoints = ringCache.getEndPoint(keyspaceName, key);
```

在这里，只用两行代码就得到了某一个 Keyspace 下面的 Key 对应的所有服务器地址。这样客户端就能直接对这些服务器进行读写操作了。

Cassandra 集群内部的信息是时刻变动的，可能有服务器离开，也可能有新的服务器加入。但是 RingCache 是缓存在客户端的，所以可以考虑定期对 RingCache 进行更新。

13.5　谨慎使用二级索引

在 Cassandra 0.7.x 版本中，提供了二级索引的功能，使得用户可以按照 Column 的值进行查询。这种特性虽然非常实用，但是也为 Cassandra 带来了额外的开销。

对于需要建立二级索引的字段，Cassandra 除了要完成正常数据写入的操作，同时还要建立索引。相当于是二次写入。这会延长数据写入的时间。如果某一个 Column Family 中有大量的字段需要建立二级索引，那么这个数据写入的额外消耗就显得非常客观了。

所以在实际的应用中，需要谨慎考虑是否真的需要使用二级索引。

13.6　通过 JMX 监测 Cassandra

在 Java 程序的运行过程中，对 JVM 和系统的监测一直是 Java 开发人员在开发过程所需要的。一直以来，Java 开发人员必须通过一些底层的 JVM API，比如 JVMPI 和 JVMTI 等，才能监测 Java 程序运行过程中的 JVM 和系统的一系列情况。这种方式一直以来被人们所诟病，因为这需要大量的 C 程序和 JNI 调用，开发效率十分低下。于是出现了各种不同的专门做资源管理的程序包。为了解决这个问题，Sun 公司也在其 Java SE 5 版本中，正式提出了 Java 管理扩展（Java Management Extensions，JMX）用来管理监测 Java 程序（同时 JMX 也在 J2EE 1.4 中被发布）。JMX 的提出，让 JDK 中开发自监测程序成为可能，也提供了大量轻量级的监测 JVM 和运行中对象/线程的方式，从而提高了 Java 语言自己的管理监测能力。

我们可以通过 JMX 来对 Cassandra 的运行状况进行监测。在配置 Cassandra 启动参数的脚本 cassandra-env.sh 中，指定了 JMX 的如下参数：

```
JVM_OPTS = " $JVM_OPTS-Dcom.sun.management.jmxremote.port = $JMX_PORT"
JVM_OPTS = " $JVM_OPTS-Dcom.sun.management.jmxremote.ssl = false"
JVM_OPTS = " $JVM_OPTS-Dcom.sun.management.jmxremote.authenticate = false"
```

所以在 Cassandra 启动后，可以通过 JDK 提供的 JMX 的监测工具来监测 Cassandra 的运行情况。

在 JDK 的安装目录的 bin 子目录中，有一个名为：jconsole 的文件，通过这个工具可以监测 Cassandra 的运行情况。

系统在启动 Cassandra 后，再启动 jconsole，将看到如图 13-1 所示界面。

图 13-1　jconsole 启动界面

选择需要监测的 Java 进程（org. apache. cassandra. thrift. CassandraDaemon），然后选择“Connect”后，可以看到该进程的整体运行情况，包括内存的占用情况、线程数量、Class 的数量和 CPU 的使用率，如图 13-2 所示。

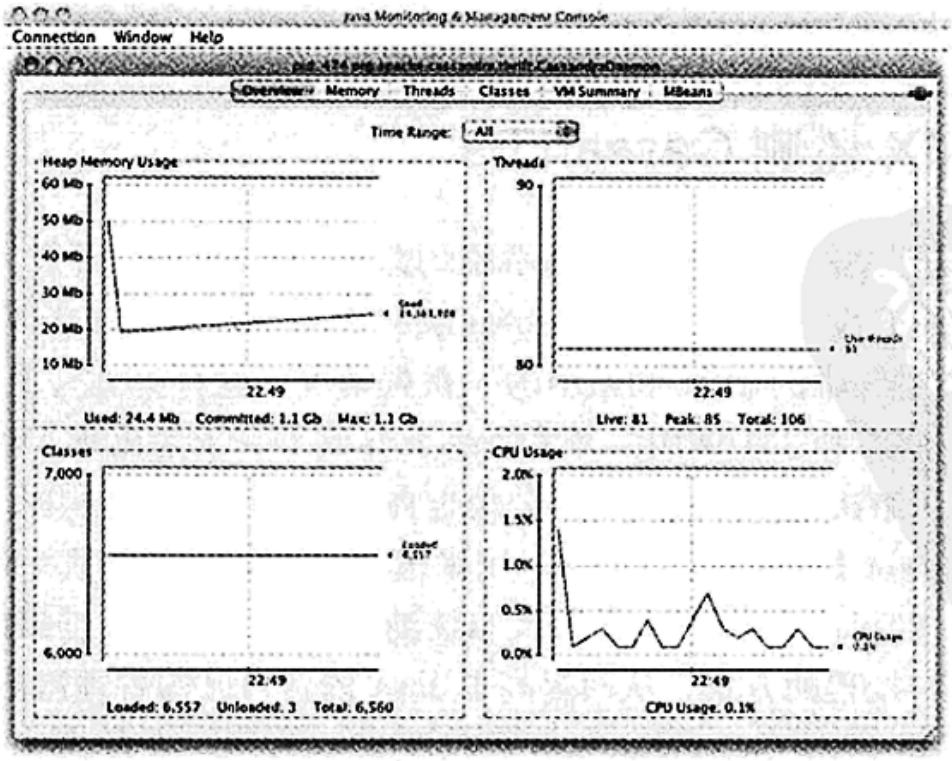


图 13-2　Cassandra 整体运行情况

点击“Memory”，可以看到 Cassandra 进程的内存使用情况，如图 13-3 所示。

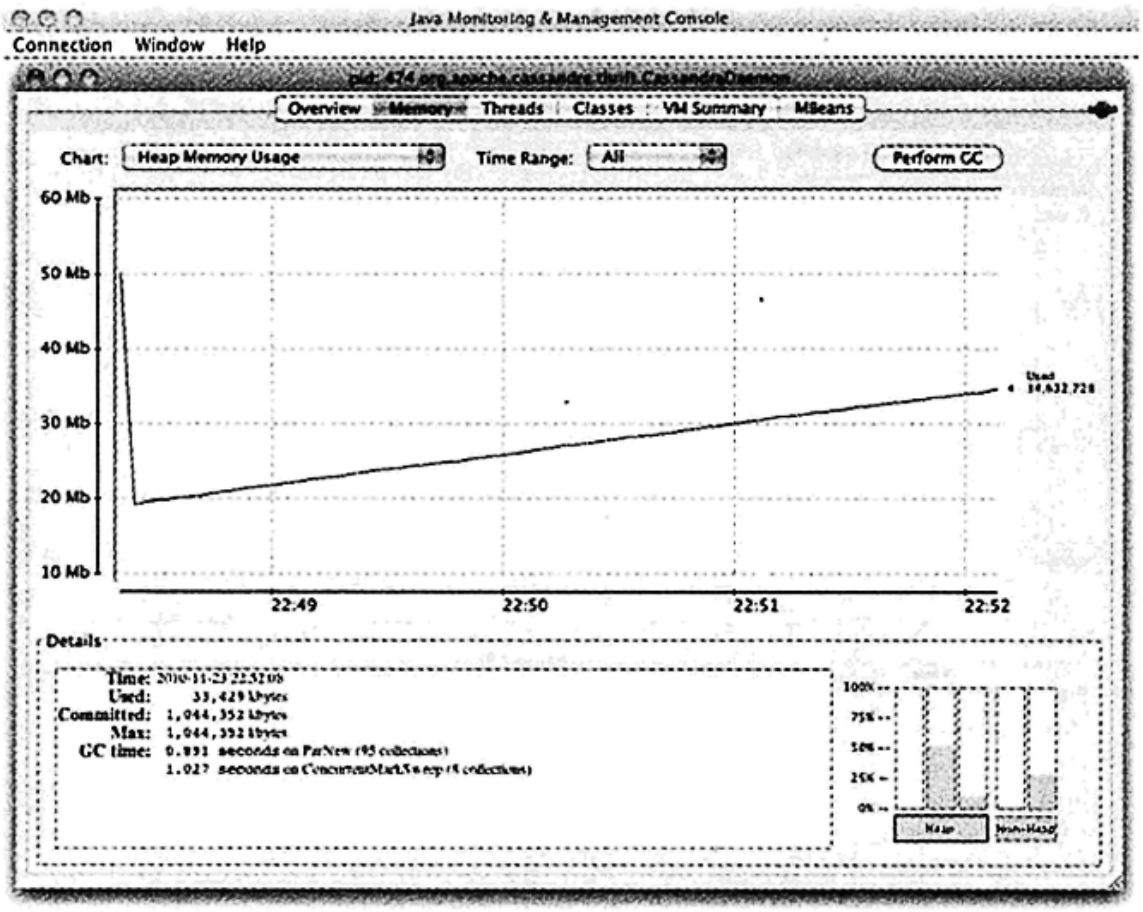


图 13-3　Cassandra 内存使用情况

点击“Threads”，可以看到 Cassandra 进程中所有的工作线程，以及每一个线程正在执行的操作，如图 13-4 所示。

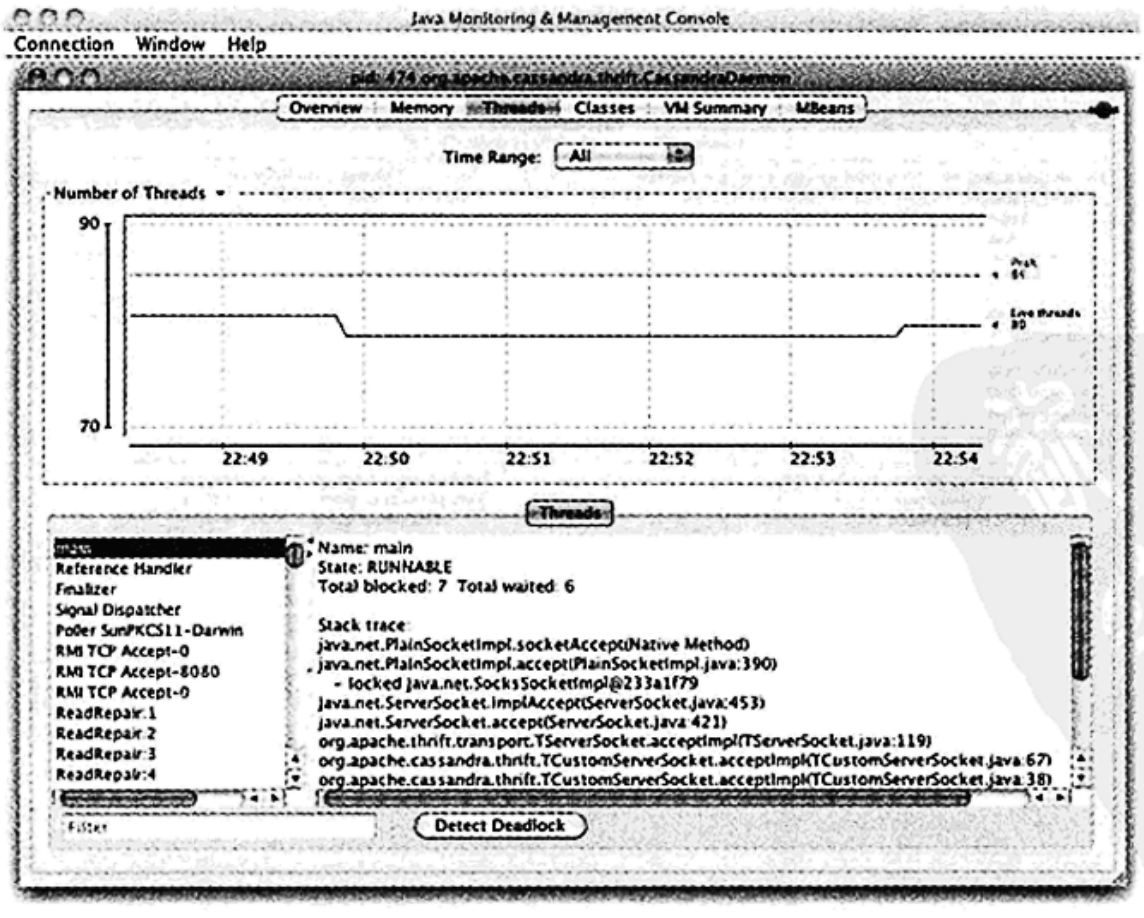


图 13-4　Cassandra 线程工作情况

点击“Classes”，可以看到 Cassandra 进程中实例化的类（Class）的个数，如图13-5所示。

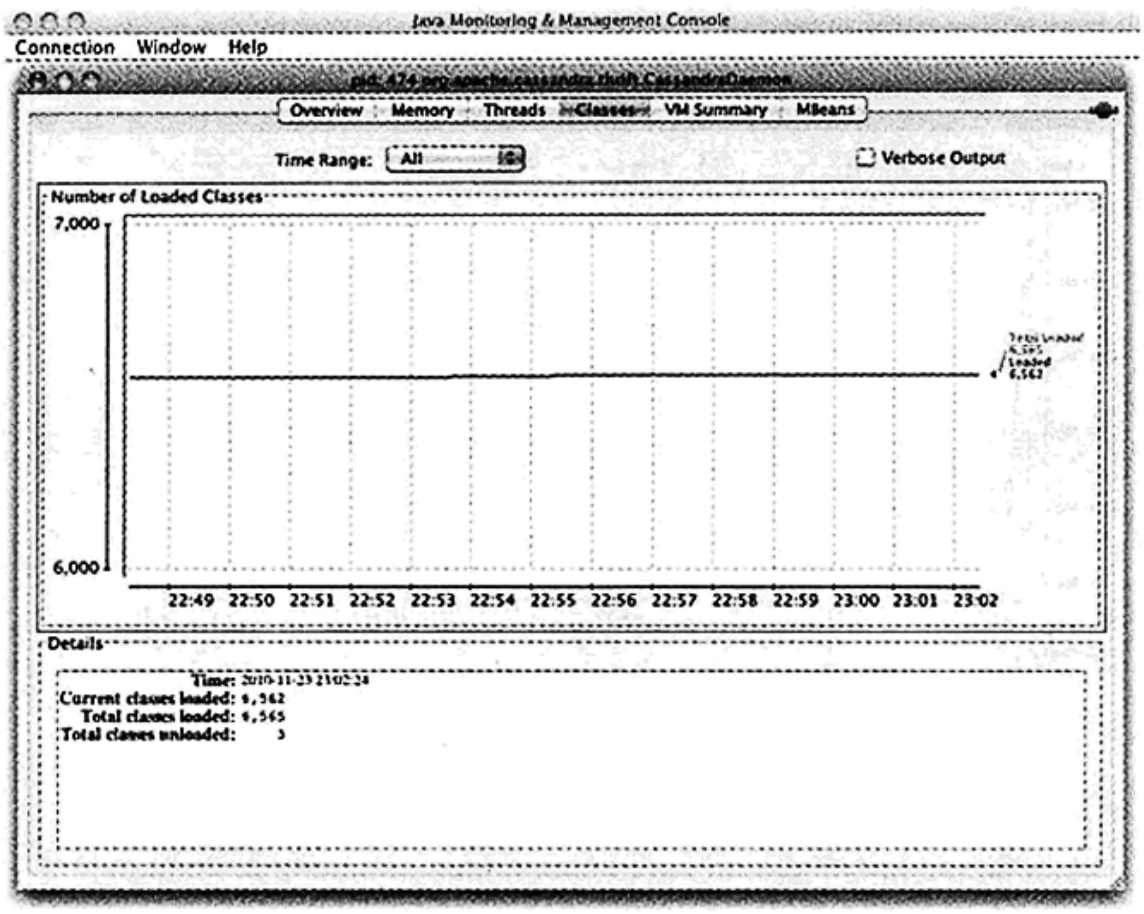


图 13-5 Cassandra 类的数量变化情况

点击“VM Summary”，可以看到启动 Cassandra 进程的所有参数以及 Java 虚拟机的运行参数，如图 13-6 所示。

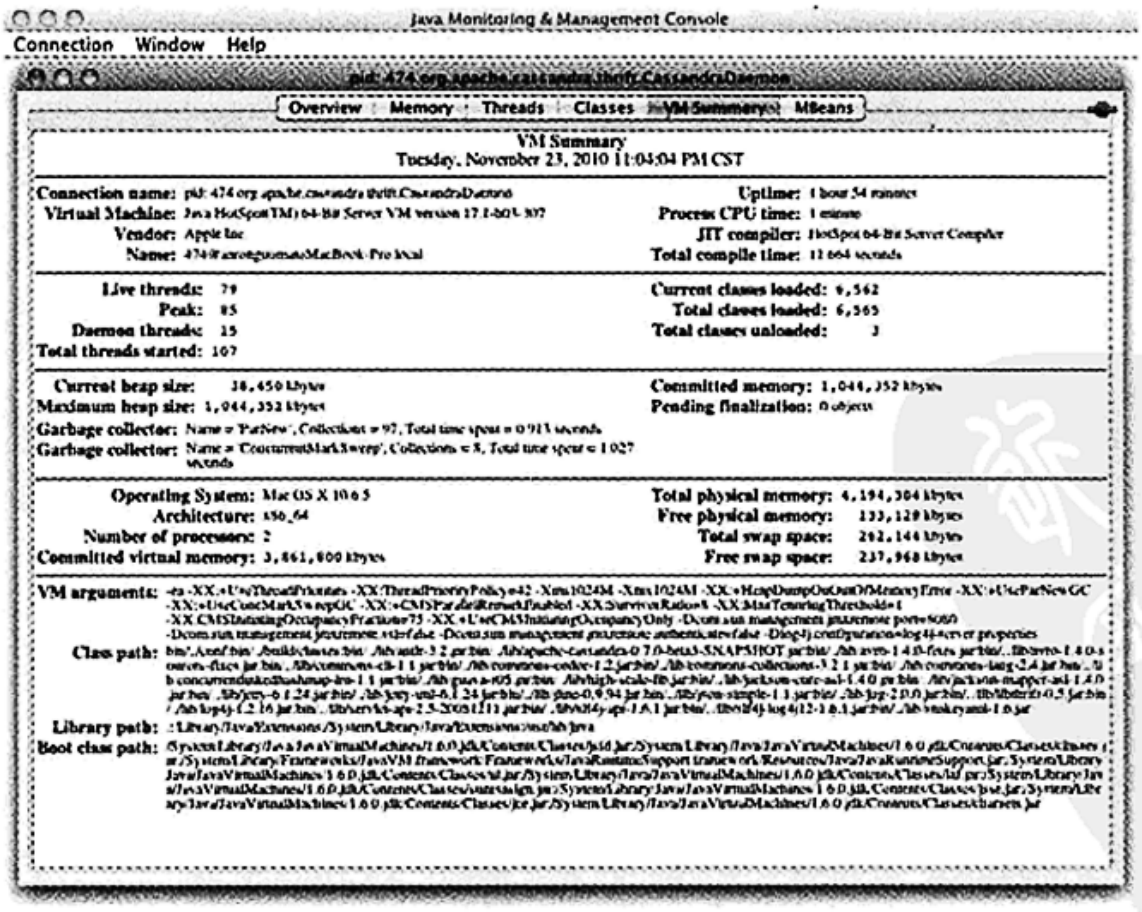


图 13-6 Cassandra 的参数

点击“MBeans”，可以看到 Cassandra 进程中所有注册到 JMX 中的变量的情况，如查看系统表空间中 IndexInfo 的值，如图 13-7 所示。

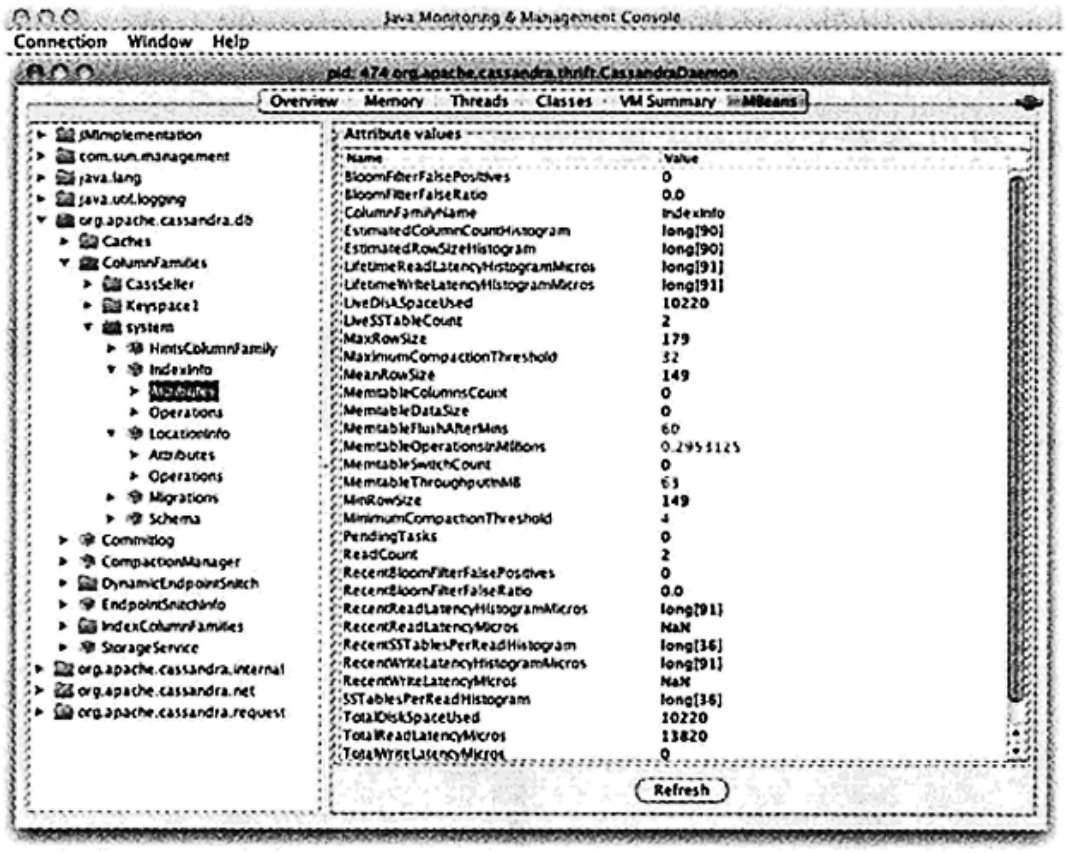


图 13-7　Cassandra 中的 JMX 变量的值

在“MBeans”中，还可以对 Cassandra 进程中暴露的 JMX 方法直接进行调用，如图 13-8所示。

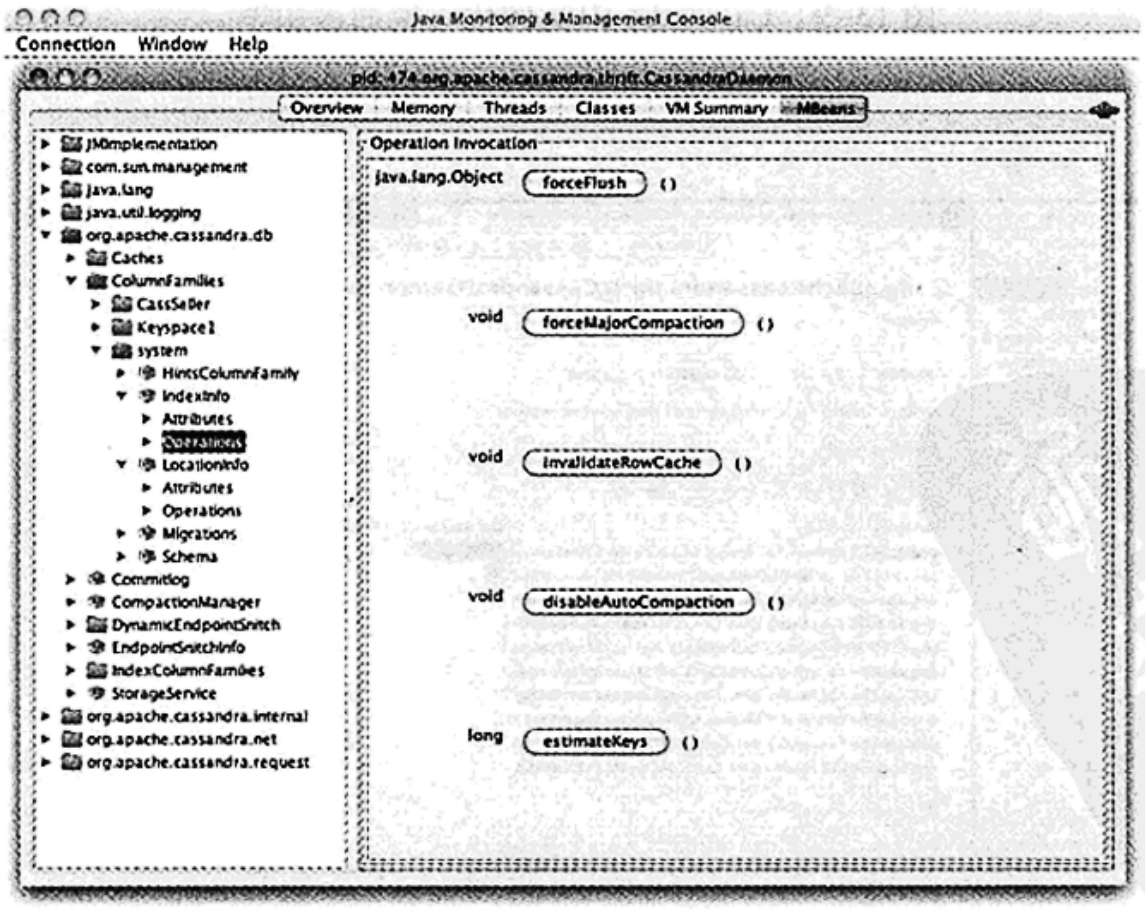


图 13-8　Cassandra 中的 JMX 方法

如果用鼠标点击“forceMajorCompaction”按钮，就可以对这个 ColumnFamily 进行主数据压缩的操作了。

JDK 除了提供 jconsole 以外，还提供了一个叫做 jvisualvm 的工具。这个工具同样在 JDK 的安装目录的 bin 子目录中。

通过 jvisualvm，可以查看 Cassandra 进程中每一个实例化的类在内存中的占用比率，如图 13-9 所示。

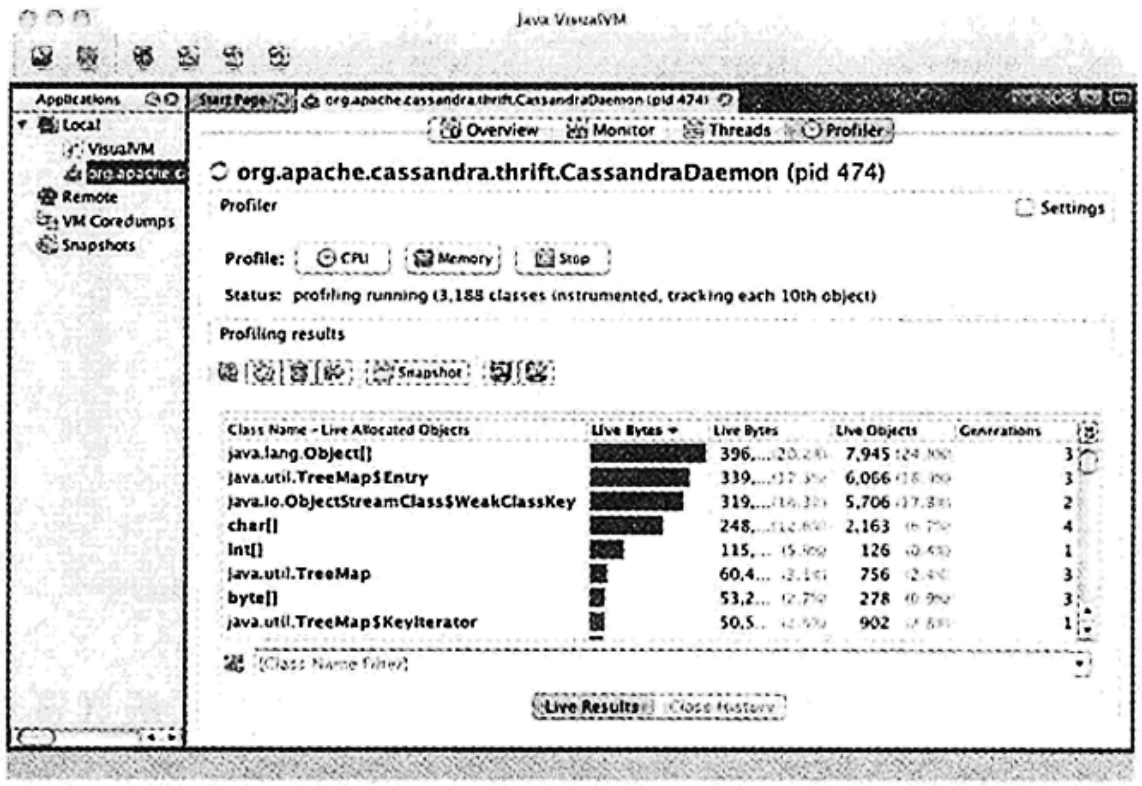


图 13-9　Cassandra 中实例化对象的占用率

还可以查看每一个方法调用的次数和占用时间，如图 13-10 所示。

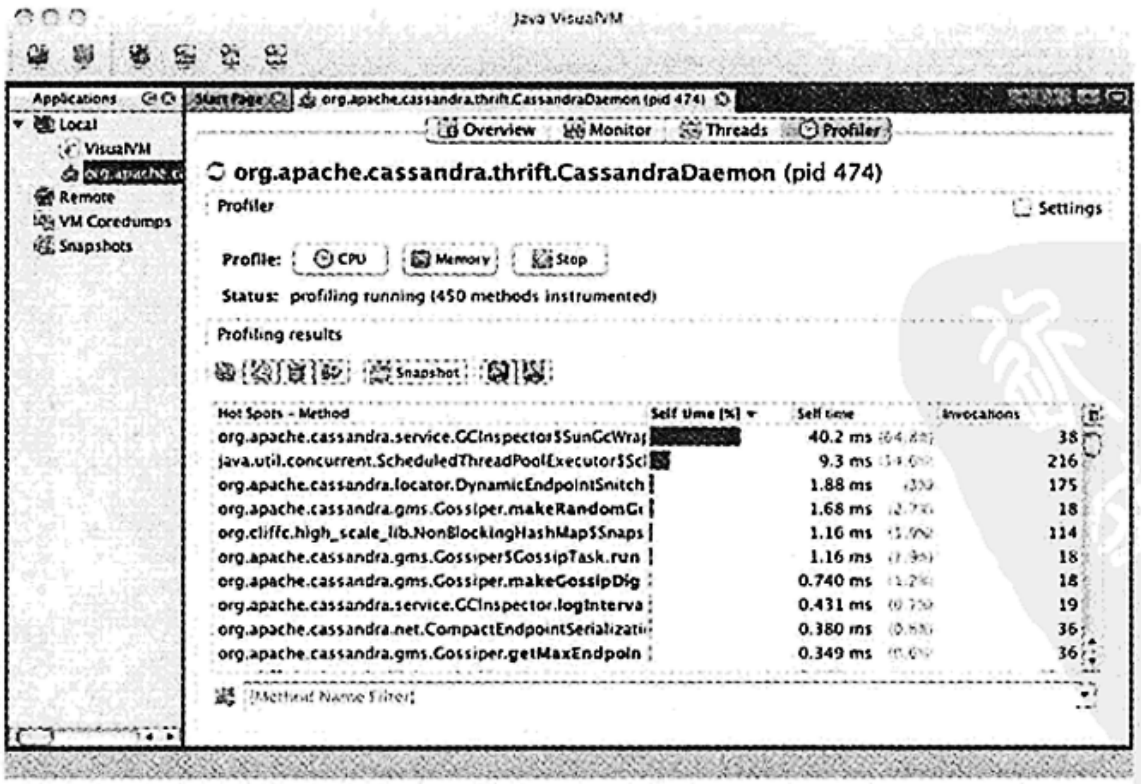


图 13-10　Cassandra 中方法调用次数和占用时间

使用好这两款 JMX 监测工具，可以帮助我们定位 Cassandra 的问题所在和性能瓶颈，从而更加高效地使用 Cassandra。

13.7　调整 JVM 启动参数

Cassandra 是基于 Java 的应用，我们可以通过修改启动 Cassandra 的 JVM 参数来达到性能调优的目的。

Cassandra 中配置 JVM 的启动参数的文件为 $CASSANDRA_HOME/conf/cassandra-env.sh，我们可以在这个文件中修改 JVM_OPTS 变量的值，然后重启 Cassandra，就可以修改 Cassandra 的 JVM 启动参数并使其生效。

在 Linux 系统中，Cassandra 默认的 JVM 启动参数如下：

```
-ea
-XX:+UseThreadPriorities
-XX:ThreadPriorityPolicy=42
-Xms $MAX_HEAP_SIZE
-Xmx $MAX_HEAP_SIZE
-XX:+HeapDumpOnOutOfMemoryError
-Xss128k
-XX:+UseParNewGC
-XX:+UseConcMarkSweepGC
-XX:+CMSParallelRemarkEnabled
-XX:SurvivorRatio=8
-XX:MaxTenuringThreshold=1
-XX:CMSInitiatingOccupancyFraction=75
-XX:+UseCMSInitiatingOccupancyOnly
```

其中 $MAX_HEAP_SIZE 为系统最大内存的一半。

Cassandra 默认使用的每一个 JVM 参数的解释如表 13-1 所示。

表 13-1　JVM 参数解释

JVM 参数名称	JVM 参数解释
ea	从 JDK1.4 开始，Java 可支持断言机制，用于诊断运行时问题。通常在测试阶段使断言有效，在正式运行时不需要运行断言。断言后的表达式的值是一个逻辑值，为 true 时断言不运行，为 false 时断言运行，抛出 java.lang.AssertionError 错误 参数 ea 用来设置虚拟机是否启动断言机制，默认时虚拟机关闭断言机制，用 -ea 可打开断言机制，不加 < packagename > 和 classname 时运行所有包和类中的断言，如果希望只运行某些包或类中的断言，可将包名或类名加到 -ea 之后。例如要启动包 com.foo.util 中的断言，可用命令 -ea：com.foo.util
UseThreadPriorities	开启 Java 线程优先级的功能
ThreadPriorityPolicy	Java 线程优先级的策略

（续）

JVM 参数名称	JVM 参数解释
Xms	设置虚拟机可用内存堆的初始大小，默认单位为字节，大小为 1024 的整数倍，并且要大于 1MB，可用 KB 或 MB 为单位来设置较大的内存数。初始堆大小为 2MB 例如：－Xms6400KB，－Xms256MB
Xmx	设置虚拟机内存堆的最大可用大小，默认单位为字节。该值必须为 1024 整数倍，并且要大于 2MB。可用 KB 或 MB 为单位来设置较大的内存数。默认堆最大值为 64MB 例如：－Xmx81920KB，－Xmx80MB 当应用程序申请了大内存运行时，虚拟机抛出 java. lang. OutOfMemoryError：Java heap space 错误，就需要使用－Xmx 设置较大的可用内存堆
HeapDumpOn OutOfMemoryError	在 JVM 内存不足而导致程序崩溃的时候，将内存中的数据导出
Xss	设置线程栈的大小，默认单位为字节。与－Xmx 类似，也可用 KB 或 MB 来设置较大的值。通常操作系统分配给线程栈的默认大小为 1MB 另外也可在 Java 中创建线程对象时设置栈的大小，构造函数原型为 Thread (ThreadGroup group，Runnable target，String name，long stackSize)
UseParNewGC	缩短 Java GC 执行 minor 收集的时间
UseConcMarkSweepGC	缩短 Java GC 执行 major 收集的时间
CMSParallelRemarkEnabled	缩短 Java GC 执行 remark 操作的中断时间
SurvivorRatio	设置 young generation 在 survivor spaces 中所占的大小比例
MaxTenuringThreshold	表示一个对象如果在 survivor spaces 移动多少次还没有被回收就放入年老代
CMSInitiatingOccupancyFraction	表示年老代占到约百分之多少的时候开始执行 CMS 操作
UseCMSInitiatingOccupancyOnly	这个参数让 JVM 只是用在 CMSInitiatingOccupancyFraction 中定义的值而不去使用在运行中计算

Sun 的官网网站中有完整 JVM 参数的详细说明，可以参考。

在设置 JVM 的启动参数时，有两个最为重要的参数：Xms 和 Xmx。在进行 JVM 参数调优的时候，可以先从 Xms 和 Xmx 这两个参数开始，然后再根据实际的应用运行情况调节其他的参数。

假设 Cassandra 集群中实际的服务器内存大小为 16GB，可以尝试使用如下 JVM 启动参数：

```
-da
-Xms12G
-Xmx12G
-XX:+UseParallelGC
-XX:+CMSParallelRemarkEnabled
-XX:SurvivorRatio=4
-XX:MaxTenuringThreshold=0
```

13.8　使用适合的系统配置参数

在本书的第 11 章中介绍了 Cassandra 的每一个系统配置参数，这里罗列出最为重要的系统配置参数。

1. data_file_directories，commitlog_directory，saved_caches_directory

data_file_directories 选项可以设置多个值，即如果服务器具有多个磁盘，可以将这几个磁盘都指定为存储 SSTable 文件的位置。

在生产环境中需要将 data_file_directories、saved_caches_directory 和 commitlog_directory 设置在不同的磁盘中，这样有利于分散整体系统的磁盘 I/O 的压力，使得系统获得更好的整体性能。

选择给 Cassandra 使用存放数据的磁盘最好是给 Cassandra 单独使用。比如某一个磁盘即存储 MySQL 的数据文件同时也存储 Cassandra 的数据文件，这样就不大适合。

2. commitlog_sync

在生产环境中，可以考虑使用 batch 的形式来记录 commitlog，这样可以提高系统的数据写入性能，但是系统在 Crash 的时候，可能会有部分 commitlog 的数据丢失。但是这种数据丢失是允许的，因为 Cassandra 集群中还会有其他的节点写入成功，可以通过数据修复机制将这部分丢失的数据恢复回来。

3. disk_access_mode

使用虚拟内存映射的形式访问文件能够加快对文件的读写速度，但是这是以消耗额外的内存作为代价的。所以要根据实际内存大小与文件大小来选择合适的文件访问方式。在生产环境中，可以使用 mmap_index_only 的形式，保证索引的高效使用，同时又不至于占用大量的虚拟内存空间。

4. concurrent_reads，concurrent_writes

concurrent_reads 选项设置得越大，Cassandra 在进行读取操作时可以使用的线程数就越多。推荐的配置为：CPU 的个数 ×2。假设 Cassandra 服务器是 8 核的 CPU，那么可以将并发读取的参数位置为 16。

concurrent_writes 选项设置得越大，Cassandra 在进行写入操作时可以使用的线程数就越多。这个参数可以根据每一个 Cassandra 服务的 Cassandra 客户端的数量来设置。一般来说，可以将这个参数设置为 32。

5. rpc_timeout_in_ms

Cassandra 集群在实际的使用中，如何读写压力过大，很容易出现超时异常。原因是

Cassandra 服务器内部有过多的读写请求需要处理，无法在指定的时间内进行响应。对于这种问题，需要考虑应用的设计是否合理，能否再优化，或者是为 Cassandra 集群增加新的机器，提高读写性能。

如果不是实时性非常高并且不是对外提供服务的应用，可以适当将这个参数调节成30 000 ms。

13.9 本章小结

本章从最佳实践的角度介绍和讲解了在设计 Cassandra 应用数据模型、程序开发、系统配置、程序配置以及线上环境运行和维护方面应该注意的问题，同时介绍了如何监测 Cassandra。

在真实的生产环境中，情况是多种多样的，各种异常情况都有可能发生，所以还需要根据实际的情况来考虑。

附录 A　在 Eclipse 中修改 Cassandra 源代码

本附录讲解如何通过代码管理工具 SVN 从官网下载 Cassandra 的源代码，完成编译，并执行单元测试。

A.1　配置环境

JDK 6，ANT 1.8

Eclipse 3.5

Eclipse SVN 插件：http://subclipse.tigris.org/update_1.6.x

安装 Eclipse SVN 插件的步骤如下：

1）在 Eclipse 的菜单中选择：Help -> Install New Software，如图 A-1 所示。

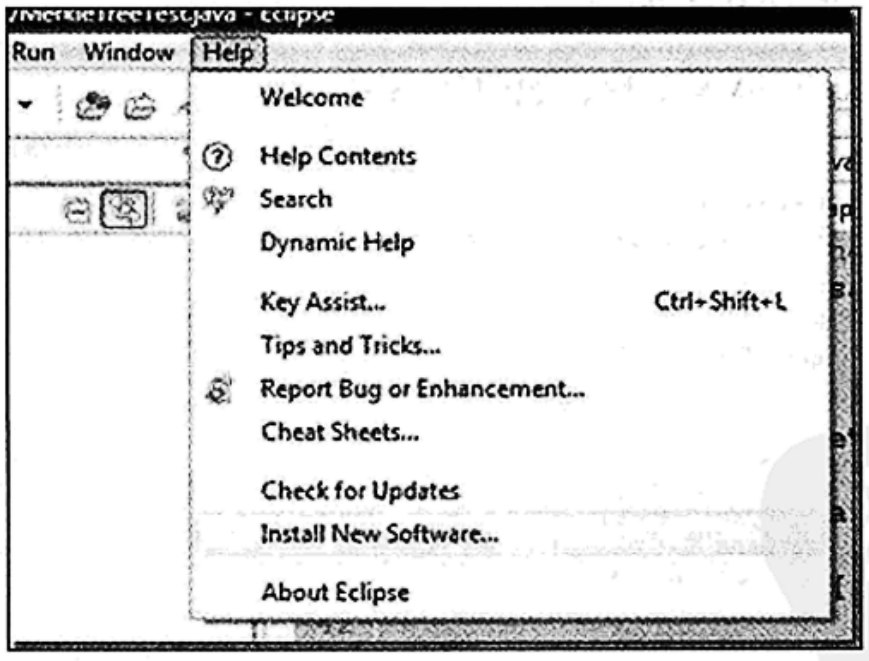


图 A-1　选择安装插件

2）输入需要安装的插件的地址 http://subclipse.tigris.org/update_1.6.x，然后勾选所有需要安装的选项，如图 A-2 所示。

3）点击 Next 按钮，按照系统提示的要求一步一步操作。

4）等待 subclipse 安装完成后重启 Eclipse。

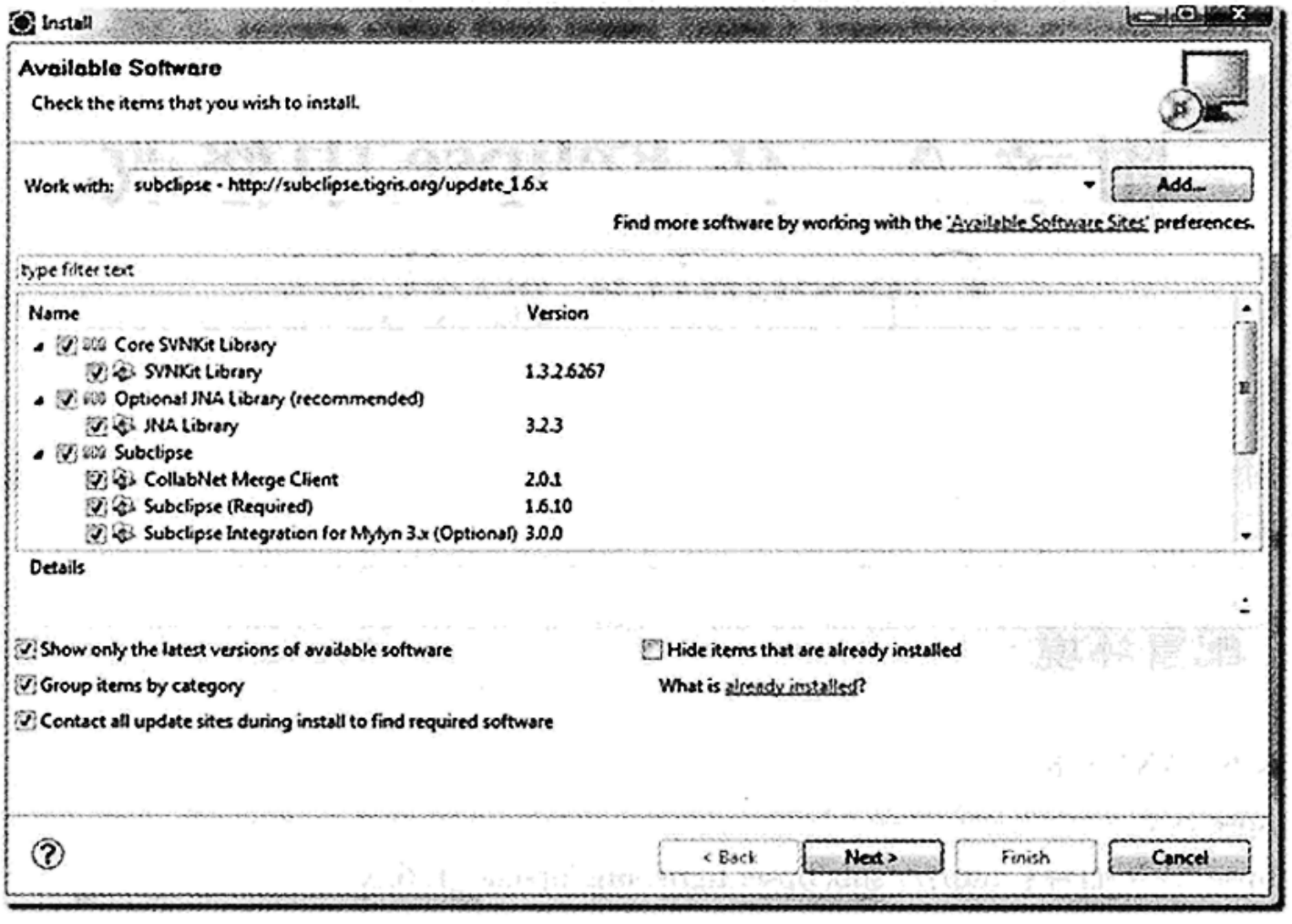


图 A-2 安装 SVN 插件

A.2 下载 Cassandra 源码

在 Eclipse 中新建一个 SVN 工程，如图 A-3 所示。

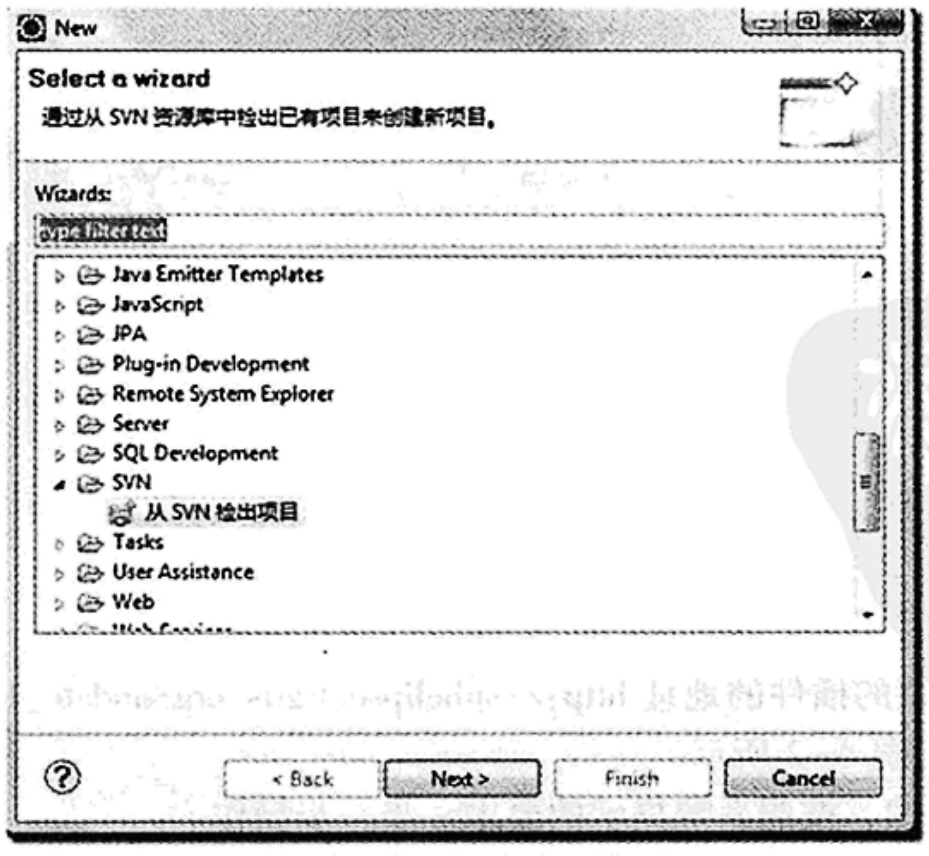


图 A-3 从 SVN 建立新项目

1）选择“创建新的资源库位置”，如图 A-4 所示。

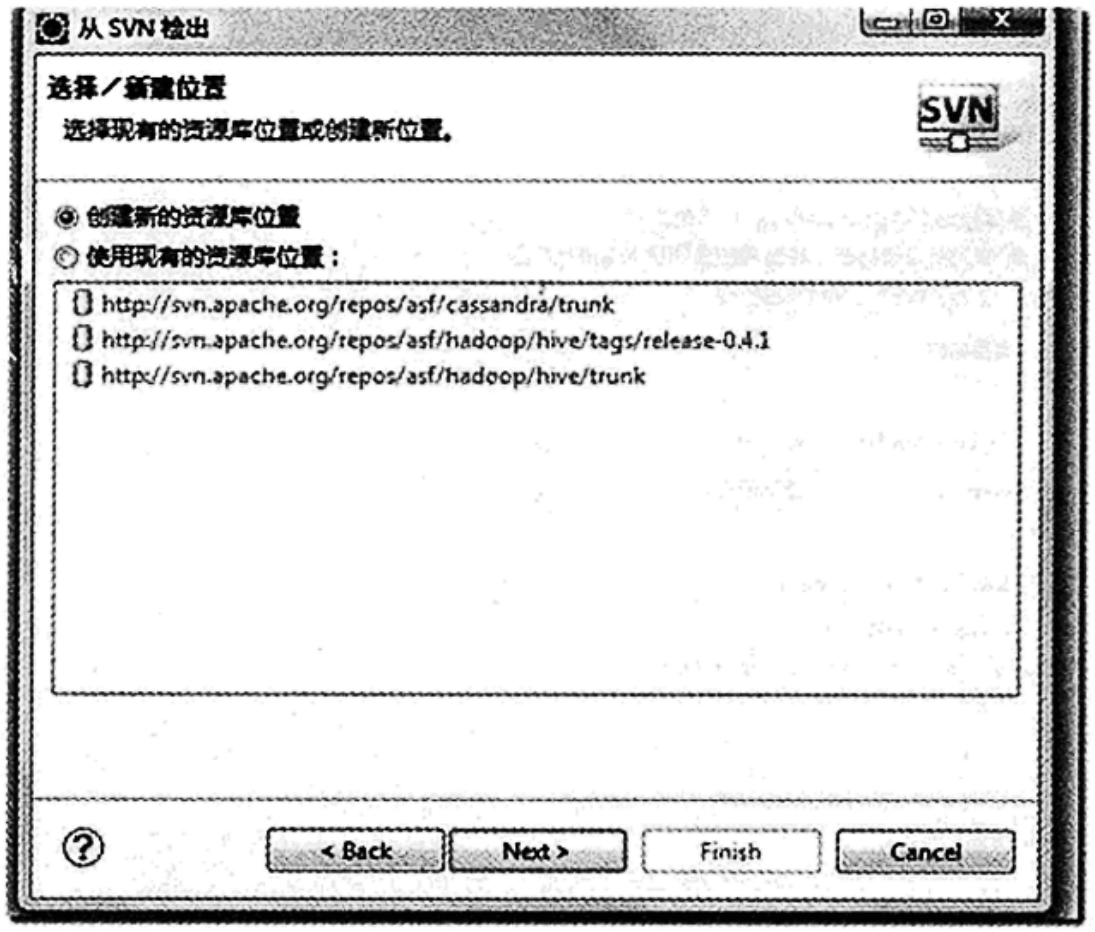


图 A-4　创建新的资源库位置

2）在获取 SVN 地址的 URL 栏中填写需要使用的源码版本，如 https://svn.apache.org/repos/asf/cassandra/tags/cassandra-0.6.2/，如图 A-5 所示。

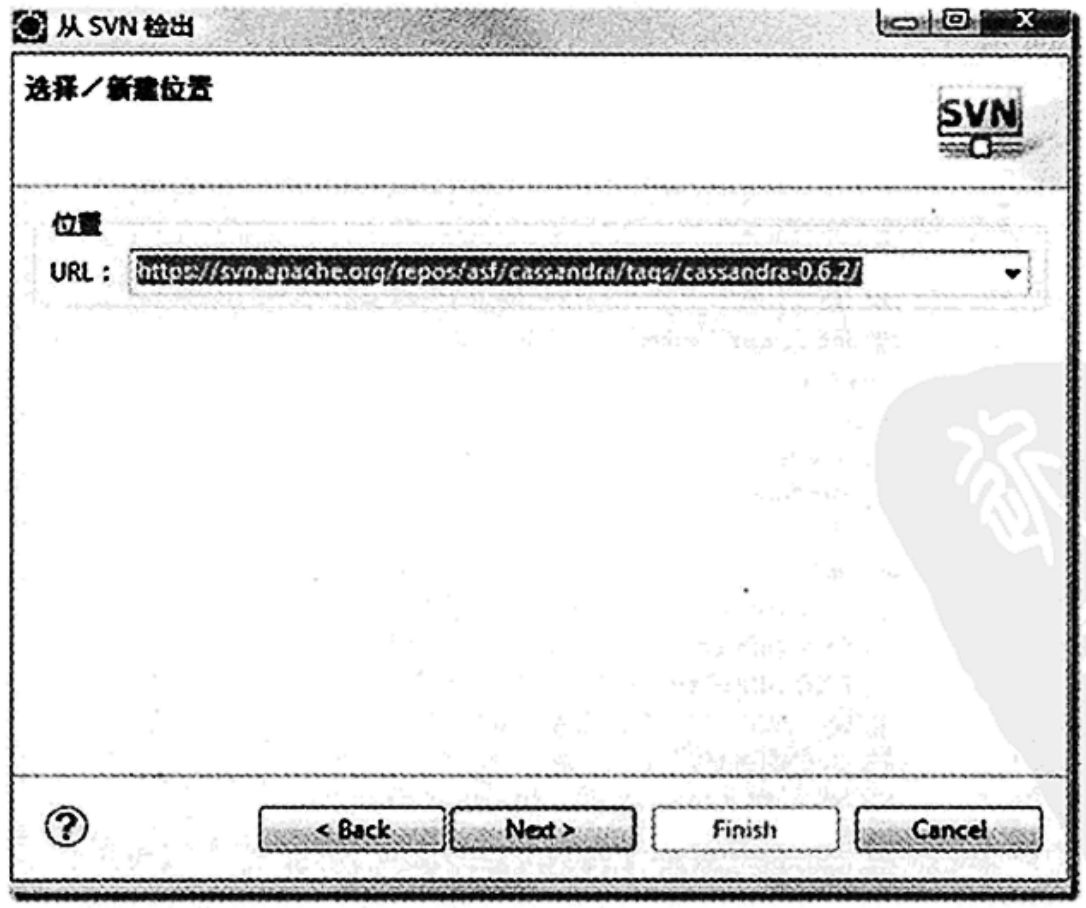


图 A-5　从 SVN 获取代码

3）点击 Next 按钮后，选择 SVN 检出方式，如图 A-6 所示。

图 A-6　选择 SVN 检出方式

4）根据提示创建一个 Java Project，SVN 将代码检出成功后，就可以看到刚刚创建的工程了，如图 A-7 所示。

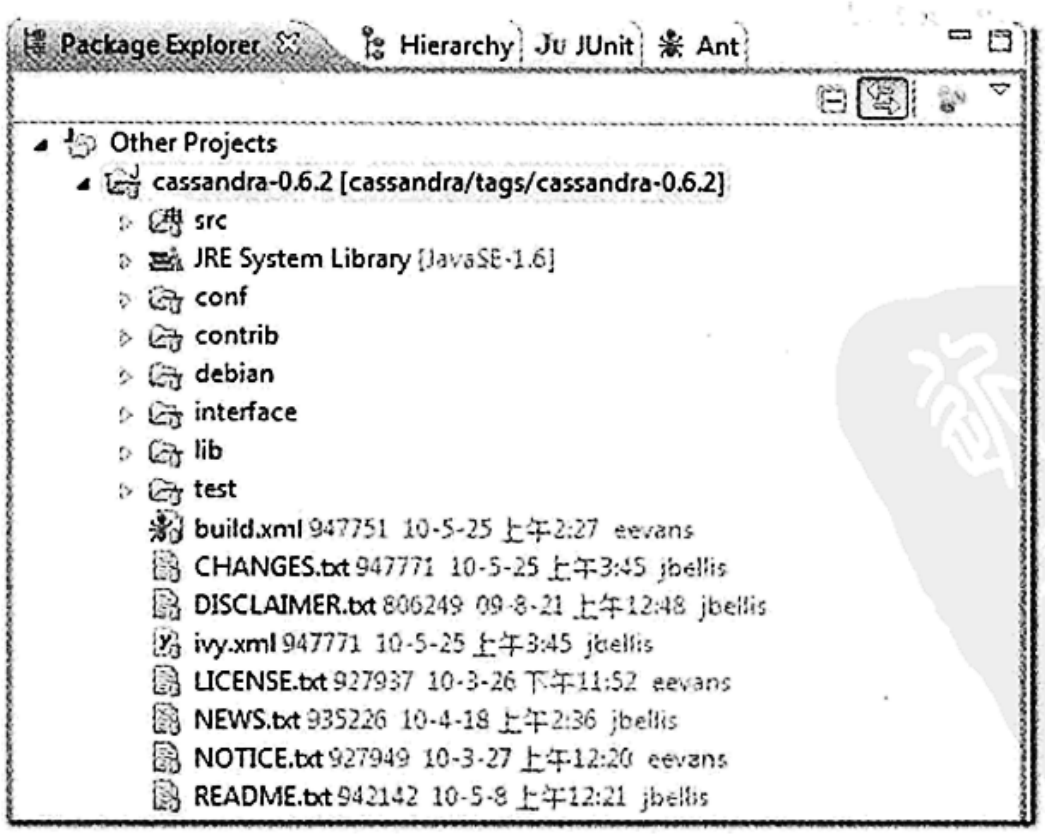


图 A-7　项目代码结构

A.3　编译 Cassandra

假设刚刚从 SVN 中下载的 Cassandra 源代码的位置为 D:\workspace\cassandra－0.6.2，就可以使用命令行工具在这个目录下通过 ANT 执行编译的操作了，如图 A-8 所示。

```
D:\workspace\cassandra-0.6.2>ant build
Buildfile: build.xml
```

图 A-8　编译 Cassandra

编译成功后，输出如图 A-9 所示。

```
build-project:
     [echo] apache-cassandra: D:\workspace\cassandra-0.6.2\build.xml
[parananer] Generating parameter names from D:\workspace\cassandra-0.6.2/inte
ce/avro/gen-java to D:\workspace\cassandra-0.6.2/build/classes

build:

BUILD SUCCESSFUL
Total time: 2 seconds
D:\workspace\cassandra-0.6.2>
```

图 A-9　Cassandra 编译结果

A.4　在 Eclipse 中修改 Cassandra 源码

1）将 D:\workspace\cassandra－0.6.2\lib 目录下的所有 jar 包添加到 Java 工程属性的 Build Path 中，如图 A-10 所示。

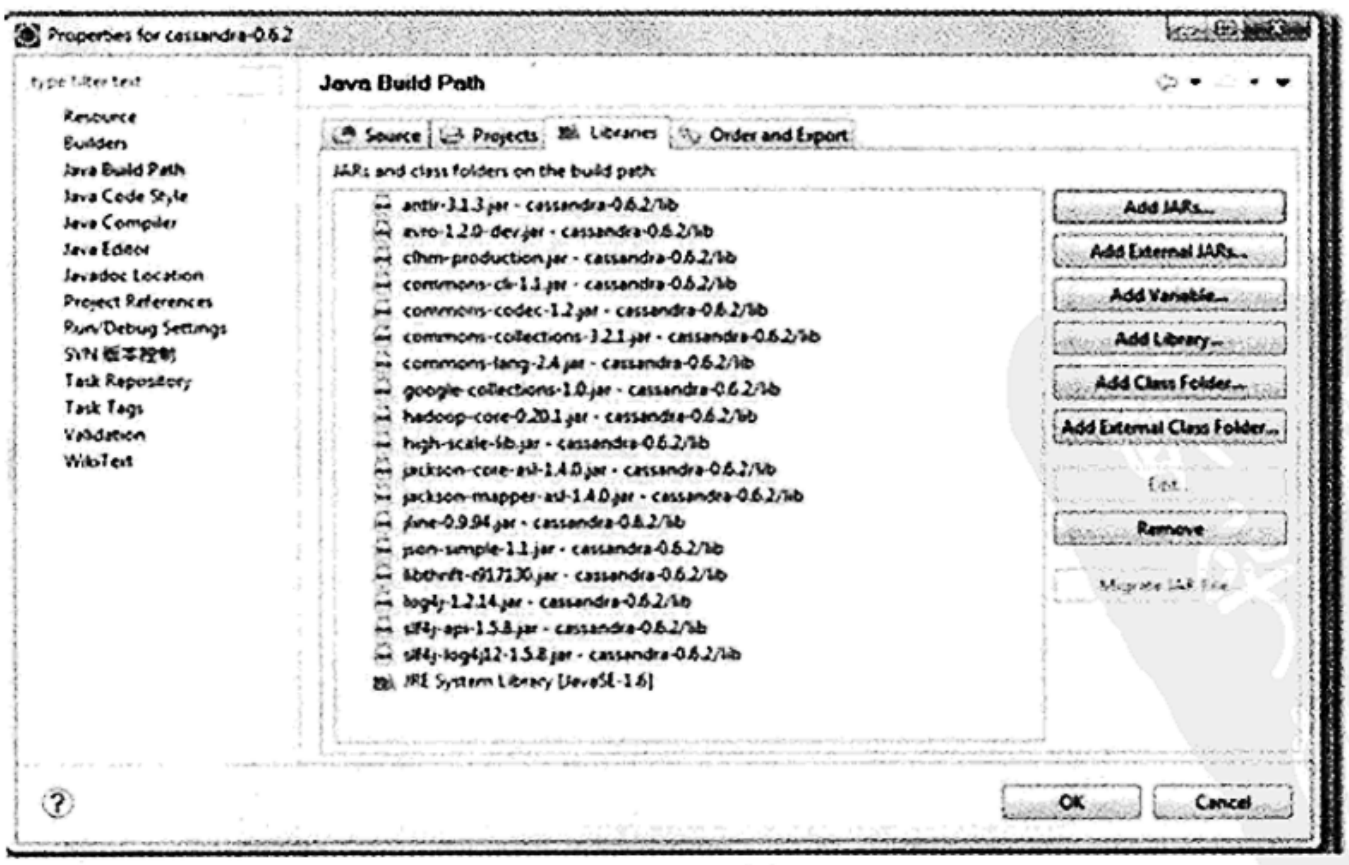


图 A-10　添加项目依赖 JAR 包

2）编辑 Source 目录的属性，默认的设置如图 A-11 所示。

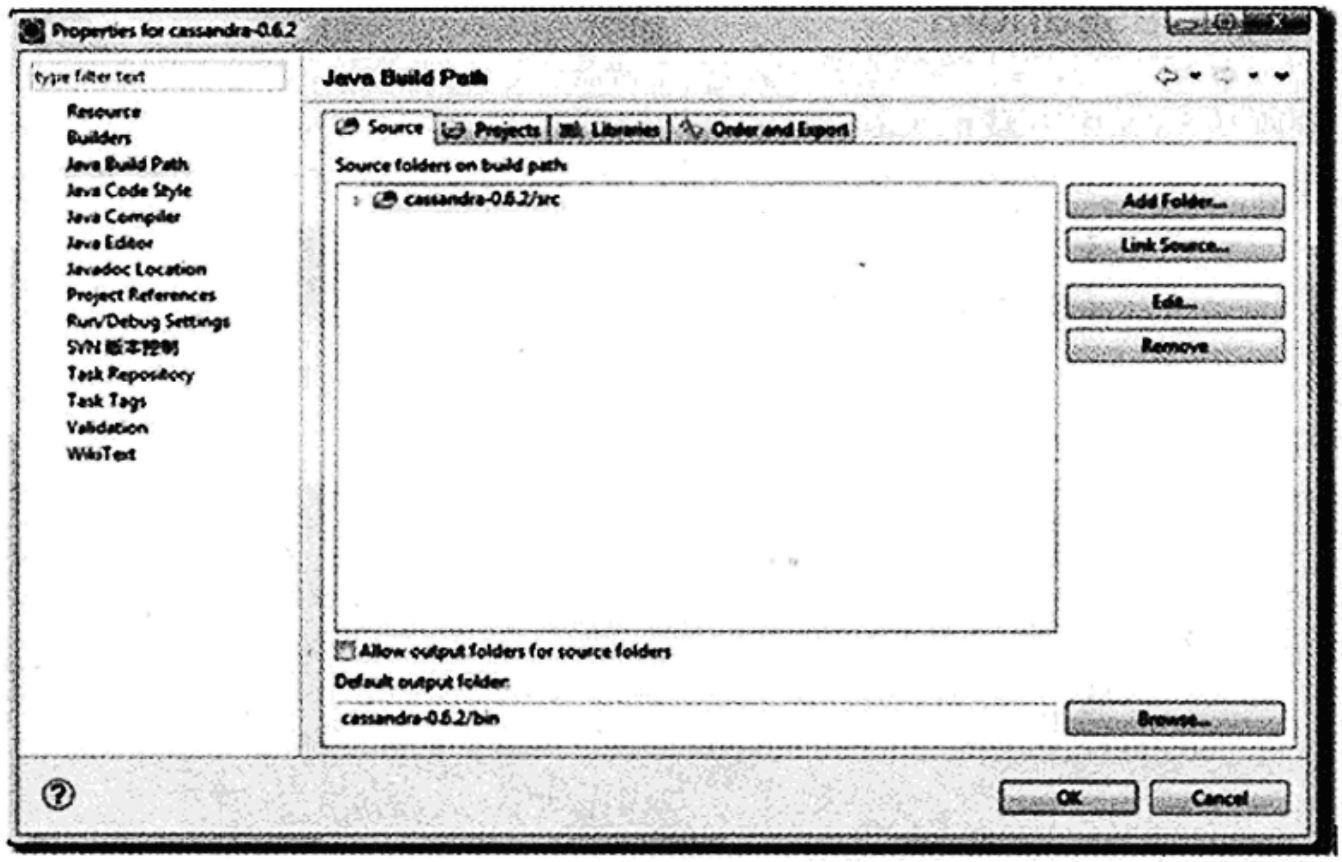


图 A-11　编辑 Source 目录属性

3）将默认的目录删除，然后添加以下 Source 目录，如图 A-12 所示。

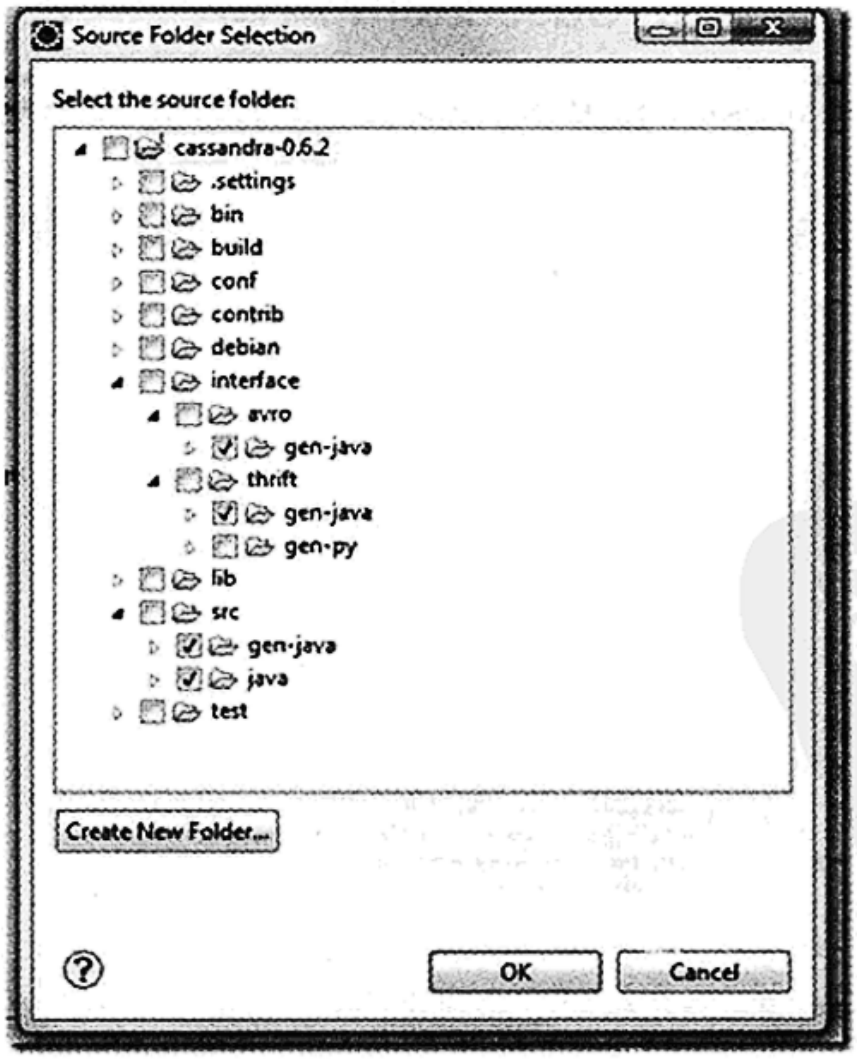


图 A-12　添加 Source 目录

添加完成之后，Java 工程目录结构如图 A-13 所示。

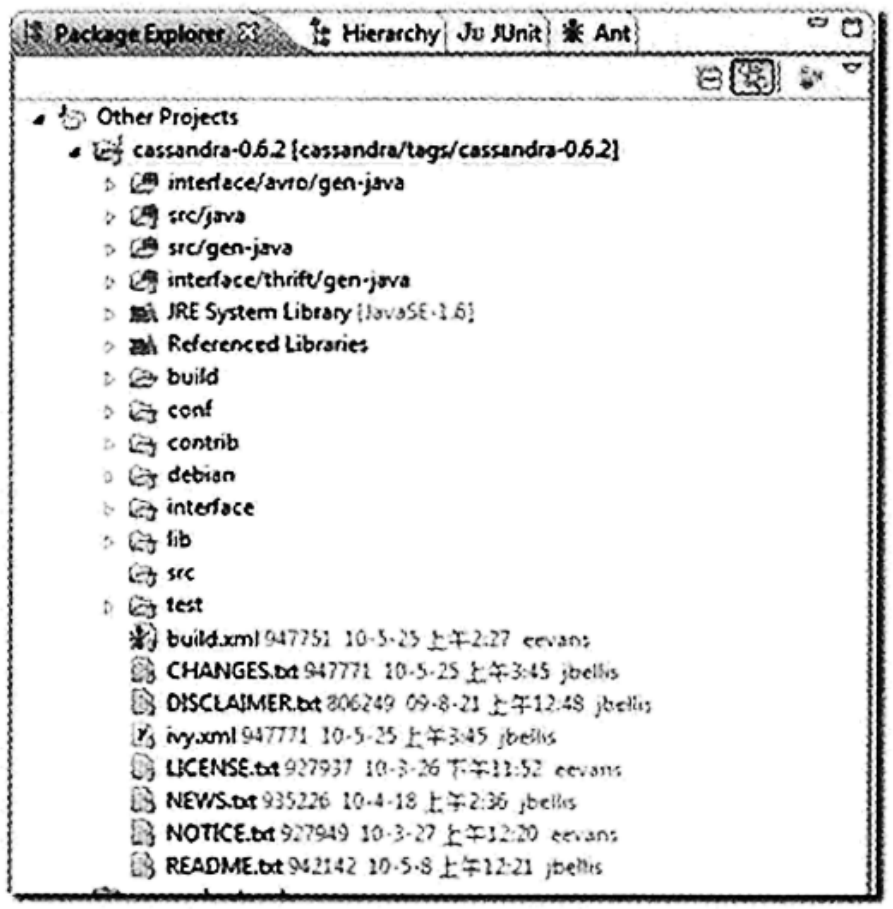


图 A-13　工程目录结构

按照正常的流程来说，对 Eclipse 的配置就算完成了，可以很方便地在 Eclipse 中编辑 Cassandra 源码了。

但是 Eclipse 还是有一些小 Bug 可能会影响到正常的功能，例如，如图 A-14 所示的错误提示。

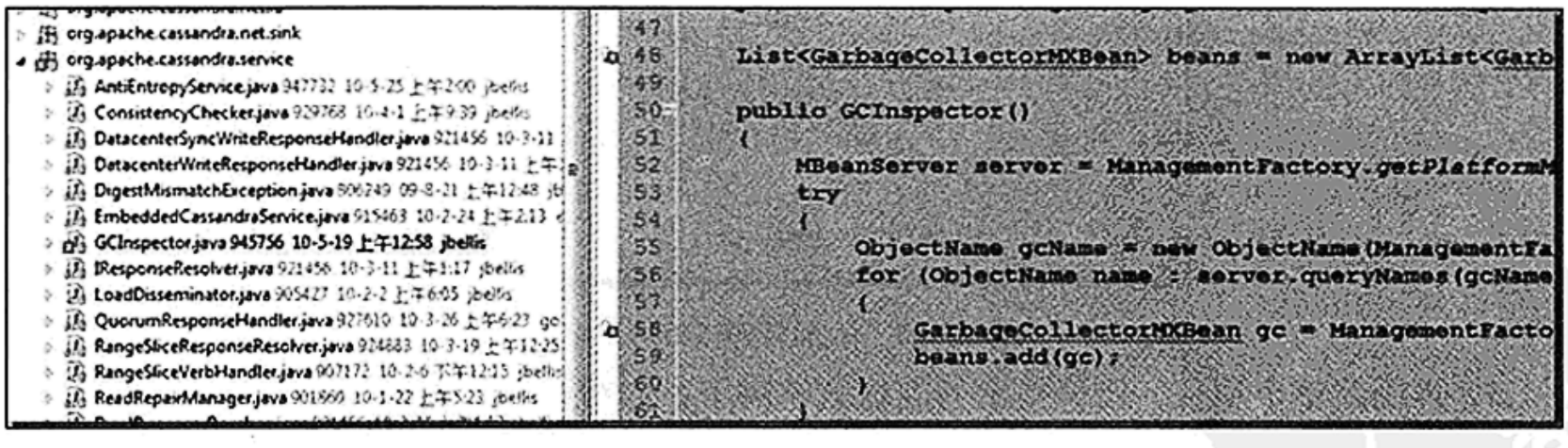


图 A-14　代码错误提示

可以看到，GarbageCollectorMXBean 其实是 JDK 中的一个类，但是 Eclipse 提示找不到该类。处理的办法如下：

1）打开 Java 工程的 Build Path，先移除 JRE System Library，如图 A-15 所示。

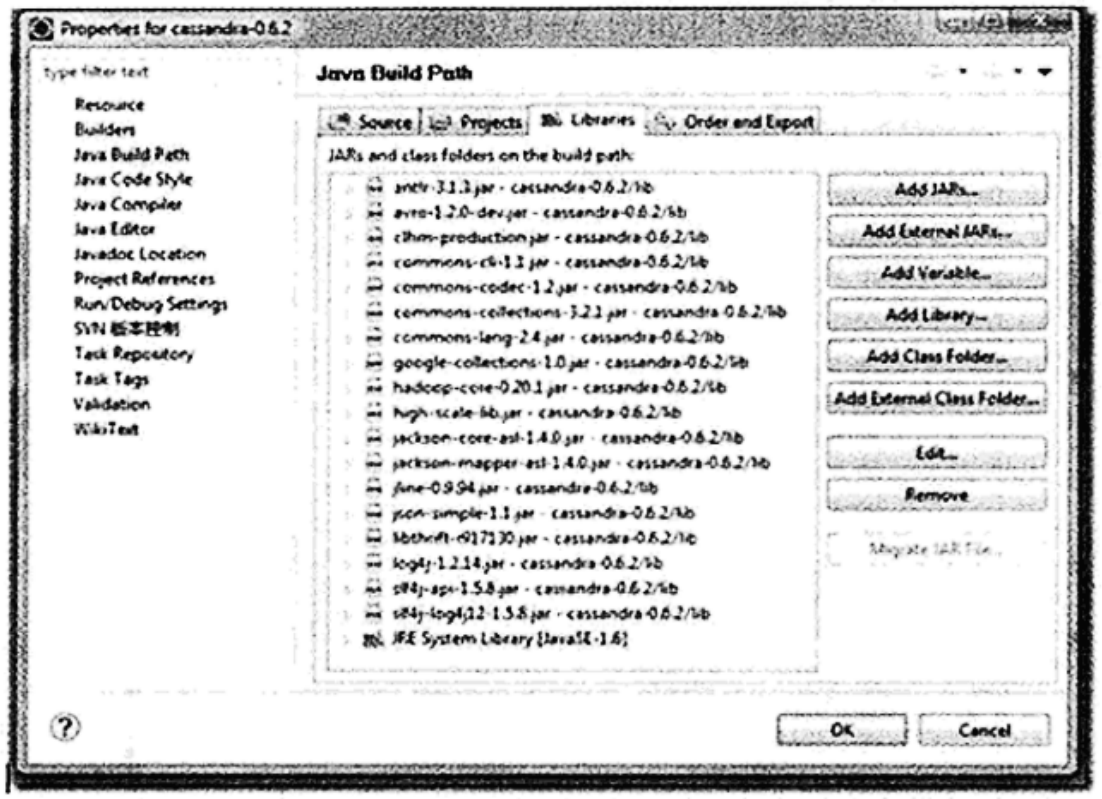


图 A-15 移除 JRE System Library

2）再将移除的 JRE System Library 添加回来，如图 A-16 所示。

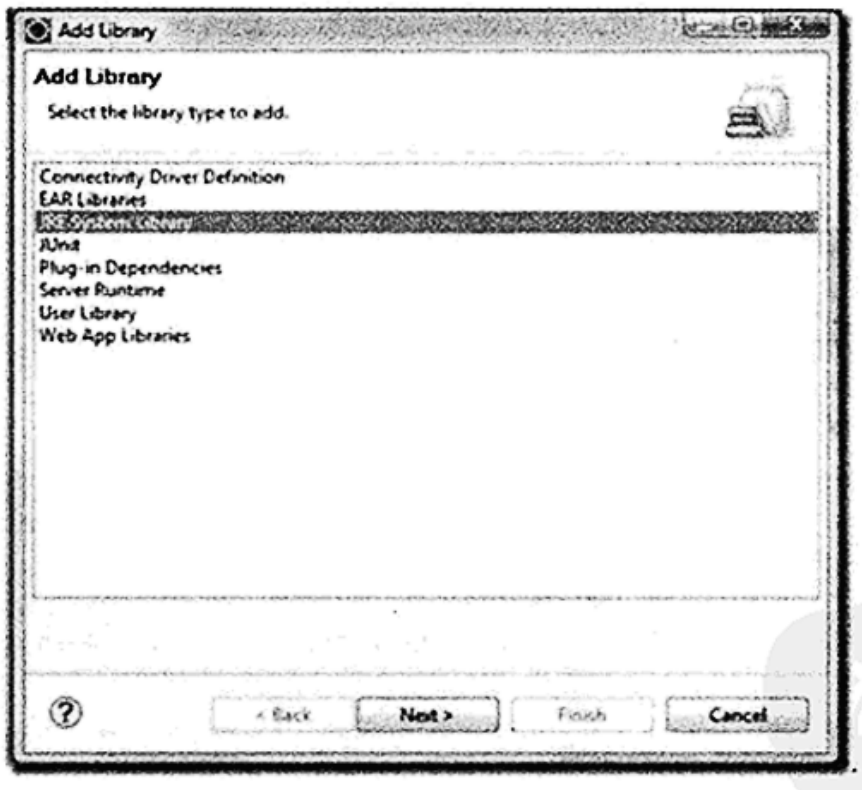


图 A-16 添加依赖 JRE 环境

3）再刷新整个项目，错误提示的问题就解决了，如图 A-17 所示。

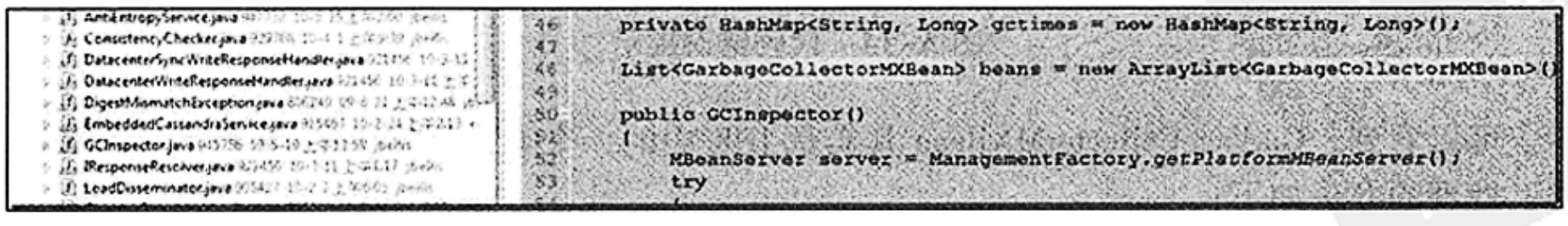


图 A-17 错误提示消失

可以看到，GarbageCollectorMXBean 已经能够被 Eclipse 识别出来了。

A.5　运行单元测试

可以将单元测试的代码也添加到 Java 工程的属性中。步骤如下：

1）添加 JUnit4 的 JAR 包到工程的 Build Path 中。

2）添加 test 目录下的测试代码到工程的 Source 中。

添加成功后工程的列表如图 A-18 所示。

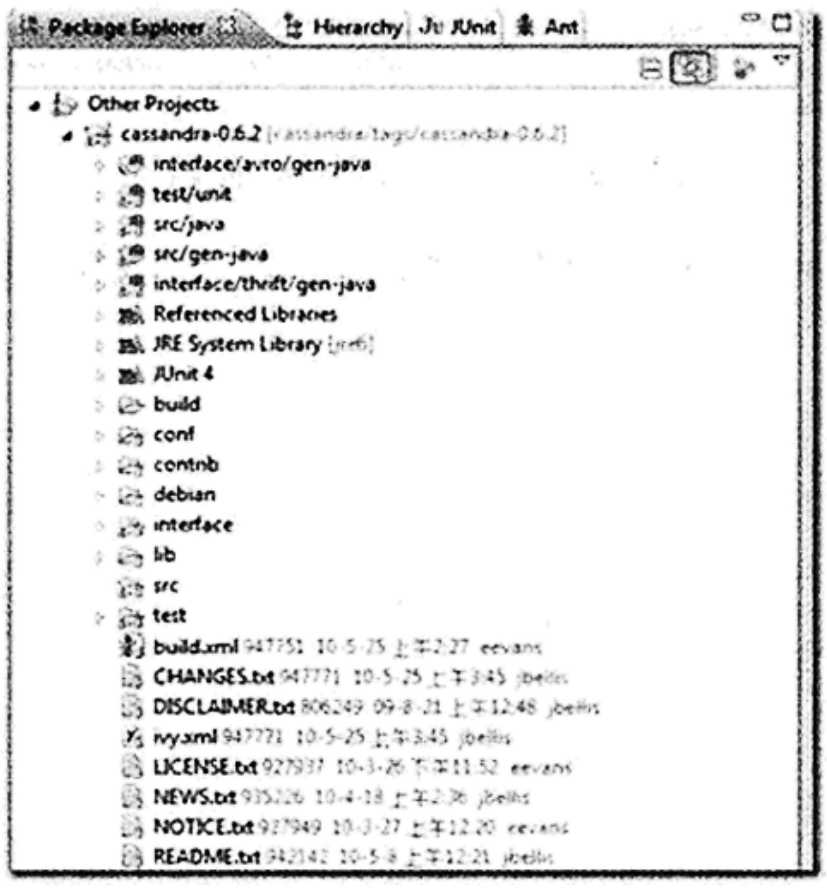


图 A-18　项目代码结构

这样就可以直接在 test 中运行已有的单元测试了。

A.6　打包发布

当完成对 Cassandra 源码的修改以后，可以将修改后的版本打包发布出来。使用 ANT 命令 ant release，结果如图 A-19 所示。

图 A-19　打包发布 Cassandra

命令执行完毕以后，可以在 D:\workspace\cassandra－0.6.2\build 目录中看到打包后的文件，如图 A-20 所示。

Name	Date modified	Type	Size
classes	2010/6/6 16:17	File Folder	
dist	2010/6/6 16:18	File Folder	
javadoc	2010/6/6 16:18	File Folder	
lib	2010/6/6 15:57	File Folder	
test	2010/6/6 15:57	File Folder	
apache-cassandra-0.6.2.jar	2010/6/6 16:17	WinRAR archive	1,254 KB
apache-cassandra-0.6.2-bin.rat.txt	2010/6/6 16:22	Text Document	141 KB
apache-cassandra-0.6.2-bin.tar.gz	2010/6/6 16:18	WinRAR archive	10,477 KB
apache-cassandra-0.6.2-src.rat.txt	2010/6/6 16:22	Text Document	0 KB
apache-cassandra-0.6.2-src.tar.gz	2010/6/6 16:22	WinRAR archive	9,398 KB
ivy-2.1.0.jar	2009/10/13 5:19	WinRAR archive	890 KB

图 A-20　Cassandra 打包后的文件

其中，apache－cassandra－0.6.2－bin.tar.gz 就是可以直接部署的文件。

附录 B　CassSeller 代码

本书第 4 章中编写了一个基于 Cassandra 的在线交易系统。本章将列出该项目中所有的源代码，Cassandra 的执行版本为 0.6.x。项目代码结构如图 B-1 所示。

项目依赖库如图 B-2 所示。

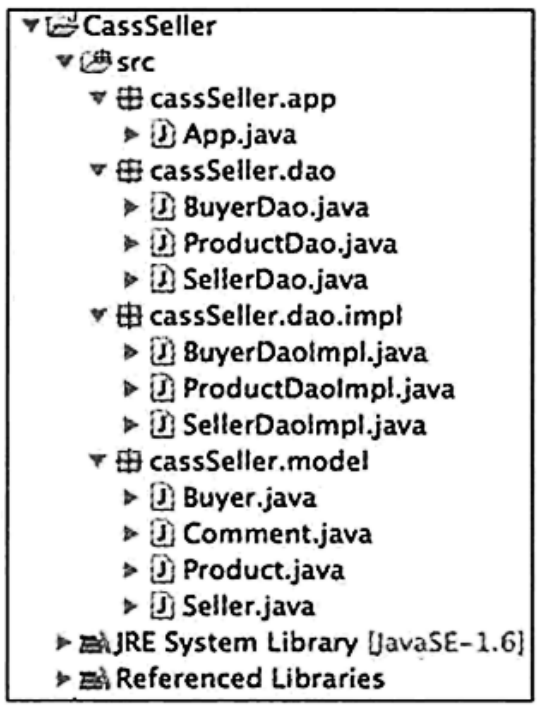


图 B-1　项目代码结构

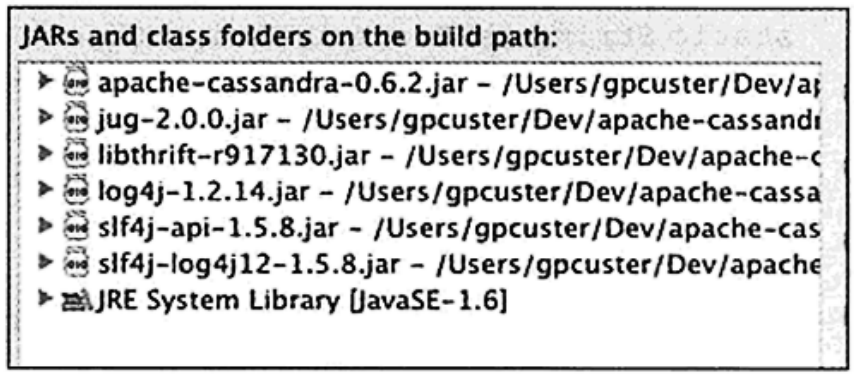


图 B-2　项目依赖库

下面分别给出各项目的源代码。

B.1　cassSeller.app.App

```
package cassSeller.app;

import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.util.List;

import org.safehaus.uuid.UUIDGenerator;

import cassSeller.dao.BuyerDao;
import cassSeller.dao.ProductDao;
```

```
import cassSeller.dao.SellerDao;
import cassSeller.dao.impl.BuyerDaoImpl;
import cassSeller.dao.impl.ProductDaoImpl;
import cassSeller.dao.impl.SellerDaoImpl;
import cassSeller.model.Buyer;
import cassSeller.model.Comment;
import cassSeller.model.Product;
import cassSeller.model.Seller;

public class App {

    /**
     * 获取对象实例中所包含的属性的名称和值的字符串
     *
     * @param entityName
     * 需要打印的对象实例
     * @return 对象实例中所包含的属性的名称和值的字符串
     * @throws Exception
     */
    static String getPropertyString(Object entityName)throws Exception {
            Class<?>c=entityName.getClass();
            Field field[]=c.getDeclaredFields();
            StringBuffer sb=new StringBuffer();
            for(Field f:field){
                if((f.getModifiers()&Modifier.STATIC)!=0){
                    continue;
                }
                sb.append(f.getName());
                sb.append(" : ");
                sb.append(invokeMethod(entityName, f.getName(), null));
                sb.append("\n");
            }
            return sb.toString();
    }

    static Object invokeMethod(Object owner, String methodName, Object[] args)
        throws Exception {
            Class<?>ownerClass=owner.getClass();

            methodName=methodName.substring(0,
                1).toUpperCase()+methodName.substring(1);
            Method method=null;
            try{
                method=ownerClass.getMethod("get"+methodName);
            }catch(Exception e){
```

```
        }

        return method.invoke(owner);
}

/**
 * @param args
 */
public static void main(String[] args) {
BuyerDao buyerDao = new BuyerDaoImpl("localhost", 9160);

Buyer buyer1 = new Buyer();
buyer1.setUserName("aaron");
buyer1.setName("郭鹏");
buyer1.setAddress("China Hangzhou");
buyer1.setAge(26);
buyer1.setSex("male");

Buyer buyer2 = new Buyer();
buyer2.setUserName("lily");
buyer2.setName("李娜");
buyer2.setAddress("China 大连");
buyer2.setAge(23);
buyer2.setSex("female");

try {
        buyerDao.insertBuyer(buyer1);
        buyerDao.insertBuyer(buyer2);
    } catch (Exception e) {
        e.printStackTrace();
}

try {
        Buyer aaron = buyerDao.getBuyer("aaron");
        System.out.println(getPropertyString(aaron));
} catch (Exception e) {
        e.printStackTrace();
}

SellerDao sellerDao = new SellerDaoImpl("localhost", 9160);

Seller seller1 = new Seller();
seller1.setUserName("sportSeller");
seller1.setName("体育用品专卖");
seller1.setAddress("China Wuhan");
seller1.setAge(30);
```

```
        seller1.setSex("male");

        Seller seller2 = new Seller();
        seller2.setUserName("seller2");
        seller2.setName("美丽服装");
        seller2.setAddress("China Beijing");
        seller2.setAge(37);
        seller2.setSex("male");

        try {
                sellerDao.insertSeller(seller1);
                sellerDao.insertSeller(seller2);
        } catch (Exception e) {
                e.printStackTrace();
        }

        try {
                Seller sportSeller = sellerDao.getSeller("sportSeller");
    System.out.println(getPropertyString(sportSeller));
        } catch (Exception e) {
                e.printStackTrace();
        }

        ProductDao productDao = new ProductDaoImpl("localhost", 9160);

        Product product1 = new Product();
        product1.setDesc("白色真皮足球");
        product1.setSellerUserName("sportSeller");
        product1.setName("Football");
        product1.setPrice(98.8);

    product1.setUuid(UUIDGenerator.getInstance().generateTimeBasedUUID());

        Product product2 = new Product();
        product2.setDesc("橡胶篮球");
        product2.setSellerUserName("sportSeller");
        product2.setName("Basketball");
        product2.setPrice(29.8);

    product2.setUuid(UUIDGenerator.getInstance().generateTimeBasedUUID());

        try {
                productDao.insertProduct(product1);
                productDao.insertProduct(product2);
        } catch (Exception e) {
          .     e.printStackTrace();
```

```
    }

    try {
            Product football = productDao.getProduct(product1.getUuid());
System.out.println(getPropertyString(football));
    } catch (Exception e) {
            e.printStackTrace();
    }

    try {
productDao.insertProductCategory("Sports", product1.getUuid());

productDao.insertProductCategory("Sports", product2.getUuid());
    } catch (Exception e) {
            e.printStackTrace();
    }

    try {
            List < Product > products =
                  productDao.getTopCategoryProducts("Sports", 2);
            for (Product product : products) {

System.out.println(getPropertyString(product));
            }
    } catch (Exception e) {
            e.printStackTrace();
    }

    Comment comment1 = new Comment();
comment1.setUuid(UUIDGenerator.getInstance().generateTimeBasedUUID());
    comment1.setCommentUserName("aaron");
    comment1.setContent("good product");

    Comment comment2 = new Comment();

comment2.setUuid(UUIDGenerator.getInstance().generateTimeBasedUUID());
    comment2.setCommentUserName("aaron");
    comment2.setContent("送货速度快");

    Comment comment3 = new Comment();

comment3.setUuid(UUIDGenerator.getInstance().generateTimeBasedUUID());
    comment3.setCommentUserName("lily");
    comment3.setContent("弟弟很喜欢这个礼物");

    try {
```

```
productDao.insertComment(product2.getUuid(), comment1);
productDao.insertComment(product2.getUuid(), comment2);
productDao.insertComment(product2.getUuid(), comment3);
    } catch (Exception e) {
            e.printStackTrace();
    }

    try {
            List<Comment> comments =
                  productDao.getTopProductComments(product2.getUuid(), 2);
            for (Comment comment : comments) {

System.out.println(getPropertyString(comment));
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

    }

}
```

B.2 cassSeller.dao.BuyerDao

```
package cassSeller.dao;

import cassSeller.model.Buyer;

public interface BuyerDao {

    /**
     * 将 Buyer 实例插入到 Cassandra 中
     *
     * @param buyer
     * 需要插入的 Buyer 实例
     * @throws Exception
     */
    public void insertBuyer(Buyer buyer) throws Exception;

    /**
     * 根据 buyerUserName,从 Cassandra 中查询对应的 Buyer 实例
     *
     * @param buyerUserName
     * Buyer ColumnFamily 的 key
```

```
     * @return Buyer 实例
     * @throws Exception
     */
    public Buyer getBuyer(String buyerUserName) throws Exception;
}
```

B.3　cassSeller.dao.ProductDao

```
package cassSeller.dao;

import java.util.List;

import org.safehaus.uuid.UUID;

import cassSeller.model.Comment;
import cassSeller.model.Product;

public interface ProductDao {

    /**
     * 将 Product 实例插入到 Cassandra 中
     *
     * @param product
     * 需要插入的 Product 实例
     * @throws Exception
     */
    public void insertProduct(Product product) throws Exception;

    /**
     * 将产品分类信息插入到 Cassandra 中
     *
     * @param category
     * 产品的分类
     * @param productUUID
     * Product ColumnFamily 的 key
     * @throws Exception
     */
    public void insertProductCategory(String category, UUID productUUID)
            throws Exception;

    /**
     * 将产品评论信息插入到 Cassandra 中
     *
     * @param productUUID
```

```
     * Product ColumnFamily 的 key
     * @ param comment
     * 评论的内容
     * @ throws Exception
     */
    public void insertComment(UUID productUUID, Comment comment)
                throws Exception;

    /**
     * 根据 productUUID,从 Cassandra 中查询对应的 Product 实例
     *
     * @ param productUUID
     * Product ColumnFamily 的 key
     * @ return Product 实例
     * @ throws Exception
     */
    public Product getProduct(UUID productUUID) throws Exception;

    /**
     * 根据 category,从 Cassandra 中查询最新的 topNum 个 Product 实例
     *
     * @ param category
     * 产品的分类
     * @ param topNum
     * 查询最新的实例的个数
     * @ return 最新的 topNum 个 Product 实例
     * @ throws Exception
     */
    public List<Product> getTopCategoryProducts(String category, int topNum)
                throws Exception;

    /**
     * 根据 productUUID,从 Cassandra 中查询最新的 topNum 个 Product 评论实例
     *
     * @ param productUUID
     * Product ColumnFamily 的 key
     * @ param topNum
     * 查询最新的实例的个数
     * @ return 最新的 topNum 个 Product 评论实例
     * @ throws Exception
     */
    public List<Comment> getTopProductComments(UUID productUUID, int topNum)
                throws Exception;
}
```

B. 4　cassSeller. dao. SellerDao

```
package cassSeller.dao;
import cassSeller.model.Seller;

public interface SellerDao {

    /* *
     * 将 Seller 实例插入到 Cassandra 中
     *
     * @ param seller
     * 需要插入的 Seller 实例
     * @ throws Exception
     * /
    public void insertSeller(Seller seller) throws Exception;

    /* *
     * 根据 sellerUserName,从 Cassandra 中查询对应的 Seller 实例
     *
     * @ param sellerUserName
     * Seller ColumnFamily 的 key
     * @ return Seller 实例
     * @ throws Exception
     * /
    public Seller getSeller(String sellerUserName) throws Exception;
}
```

B. 5　cassSeller. dao. impl. BuyerDaoImpl

```
package cassSeller.dao.impl;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.apache.cassandra.thrift.Cassandra.Client;
import org.apache.cassandra.thrift.Column;
import org.apache.cassandra.thrift.ColumnOrSuperColumn;
import org.apache.cassandra.thrift.ColumnParent;
import org.apache.cassandra.thrift.ConsistencyLevel;
import org.apache.cassandra.thrift.Mutation;
import org.apache.cassandra.thrift.SlicePredicate;
```

```
import org.apache.cassandra.thrift.SliceRange;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.transport.TSocket;

import cassSeller.dao.BuyerDao;
import cassSeller.model.Buyer;

/* *
 * @ author gpcuster
 *
 * /
public class BuyerDaoImpl implements BuyerDao {

    Client cassandraClient = null;

    public BuyerDaoImpl(String host, int port) {
            TSocket socket = new TSocket(host, port);

            TBinaryProtocol binaryProtocol =
                  new TBinaryProtocol(socket, false,false);
            cassandraClient = new Client(binaryProtocol);

            try {
                    socket.open();
            } catch (Exception e) {
                    e.printStackTrace();
                    cassandraClient = null;
                    return;
            }
}

@ Override
public void insertBuyer(Buyer buyer) throws Exception {
        if (cassandraClient == null) {
                throw new Exception("Can't connect to Cassandra.");
        }

        if (buyer == null) {
                throw new Exception("Can't insert null buyer to Cassandra.");
        }

        if (buyer.getUserName() == null ||buyer.getUserName().isEmpty()) {
                throw new Exception("Can't insert null
                      buyerUserName to Cassandra.");
        }
```

```
Map<String, Map<String, List<Mutation>>>mutationMap=new HashMap<String,
    Map<String, List<Mutation>>>();
Map<String, List<Mutation>>cfMutationMap=new HashMap<String, List
    <Mutation>>();
    List<Mutation>mutationList=new ArrayList<Mutation>();

    //设置买家信息
    if (buyer.getAge()>0) {
        Column c=new Column();
        c.name="age".getBytes("utf-8");
        c.value=String.valueOf(buyer.getAge()).getBytes("utf-8");
        c.timestamp=System.currentTimeMillis();

        ColumnOrSuperColumn cosc=new ColumnOrSuperColumn();
        cosc.column=c;
        Mutation mutation=new Mutation();
        mutation.column_or_supercolumn=cosc;
        mutationList.add(mutation);
            }

    if (buyer.getAddress()!=null && !buyer.getAddress().isEmpty()) {
        Column c=new Column();
        c.name="address".getBytes("utf-8");
        c.value=buyer.getAddress().getBytes("utf-8");
        c.timestamp=System.currentTimeMillis();

        ColumnOrSuperColumn cosc=new ColumnOrSuperColumn();
        cosc.column=c;
        Mutation mutation=new Mutation();
        mutation.column_or_supercolumn=cosc;
        mutationList.add(mutation);
            }

    if (buyer.getSex()!=null && !buyer.getSex().isEmpty()) {
        Column c=new Column();
        c.name="sex".getBytes("utf-8");
        c.value=buyer.getSex().getBytes("utf-8");
        c.timestamp=System.currentTimeMillis();

        ColumnOrSuperColumn cosc=new ColumnOrSuperColumn();
        cosc.column=c;
        Mutation mutation=new Mutation();
        mutation.column_or_supercolumn=cosc;
        mutationList.add(mutation);
            }
```

```
if (buyer.getName() != null && !buyer.getName().isEmpty()) {
    Column c = new Column();
    c.name = "name".getBytes("utf-8");
    c.value = buyer.getName().getBytes("utf-8");
    c.timestamp = System.currentTimeMillis();

    ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
    cosc.column = c;
    Mutation mutation = new Mutation();
    mutation.column_or_supercolumn = cosc;
    mutationList.add(mutation);
        }

        if (!mutationList.isEmpty()) {
            cfMutationMap.put(Buyer.ColumnFamily, mutationList);
            mutationMap.put(buyer.getUserName(), cfMutationMap);

cassandraClient.batch_mutate(Buyer.keySpace,
    mutationMap,ConsistencyLevel.ONE);
        }
}

@ Override
public Buyer getBuyer(String buyerUserName) throws Exception {
        if (cassandraClient == null) {
    throw new Exception("Can't connect to Cassandra.");
}

if (buyerUserName == null || buyerUserName.isEmpty()) {
    throw new Exception("Can't get null buyer to Cassandra.");
}

ColumnParent columnParent = new ColumnParent();
columnParent.column_family = Buyer.ColumnFamily;
SliceRange range = new SliceRange(new byte[] {}, new byte[] {}, true,
        Integer.MAX_VALUE);
SlicePredicate slicePredicate = new SlicePredicate();
slicePredicate.slice_range = range;

List<ColumnOrSuperColumn> buyerColumns =
        cassandraClient.get_slice(Buyer.keySpace, buyerUserName,
        columnParent, slicePredicate,ConsistencyLevel.ONE);

Buyer buyer = new Buyer();
buyer.setUserName(buyerUserName);
for (ColumnOrSuperColumn cosc : buyerColumns) {
```

```
                Column c = cosc.column;
                String columnName = new String(c.name, "utf-8");
                String columnValue = new String(c.value, "utf-8");
                if ("age".equals(columnName)) {
    buyer.setAge(Integer.parseInt(columnValue));
                } else if ("address".equals(columnName)) {
                        buyer.setAddress(columnValue);
                } else if ("sex".equals(columnName)) {
                        buyer.setSex(columnValue);
                } else if ("name".equals(columnName)) {
                        buyer.setName(columnValue);
                }
            }

            return buyer;
    }
}
```

B.6 cassSeller.dao.impl.ProductDaoImpl

```
package cassSeller.dao.impl;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.apache.cassandra.thrift.Cassandra.Client;
import org.apache.cassandra.thrift.Column;
import org.apache.cassandra.thrift.ColumnOrSuperColumn;
import org.apache.cassandra.thrift.ColumnParent;
import org.apache.cassandra.thrift.ColumnPath;
import org.apache.cassandra.thrift.ConsistencyLevel;

import org.apache.cassandra.thrift.Mutation;
import org.apache.cassandra.thrift.SlicePredicate;
import org.apache.cassandra.thrift.SliceRange;
import org.apache.cassandra.thrift.SuperColumn;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.transport.TSocket;
import org.safehaus.uuid.UUID;

import cassSeller.dao.ProductDao;
import cassSeller.model.Comment;
import cassSeller.model.Product;
```

```
public class ProductDaoImpl implements ProductDao {

    Client cassandraClient = null;
    public ProductDaoImpl(String host, int port) {
            TSocket socket = new TSocket(host, port);

            TBinaryProtocol binaryProtocol = new
                TBinaryProtocol(socket, false,false);
            cassandraClient = new Client(binaryProtocol);

            try {
                    socket.open();
            } catch (Exception e) {
                    e.printStackTrace();
                    cassandraClient = null;
                    return;
            }
    }

    @ Override
    public void insertProduct(Product product) throws Exception {
            if (cassandraClient == null) {
                throw new Exception("Can't connect to Cassandra.");
            }

if (product == null) {
    throw new Exception("Can't insert null product to Cassandra.");
}

if (product.getUuid() == null ||product.getUuid().isNullUUID()) {
    throw new Exception("Can't insert null product to Cassandra.");
}

Map<String, Map<String, List<Mutation>>>
      mutationMap = new HashMap<String, Map<String, List<Mutation>>>();
Map<String, List<Mutation> > cfMutationMap = new HashMap<String,
      List<Mutation> > ();
List<Mutation> mutationList = new ArrayList<Mutation> ();

//设置产品信息
if (product.getPrice() >0) {
        Column c = new Column();
        c.name = "price".getBytes("utf-8");
        c.value = String.valueOf(product.getPrice()).getBytes("utf-8");
        c.timestamp = System.currentTimeMillis();
```

```
            ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
            cosc.column = c;
            Mutation mutation = new Mutation();
            mutation.column_or_supercolumn = cosc;
            mutationList.add(mutation);
        }

        if (product.getDesc() != null && !product.getDesc().isEmpty()) {
            Column c = new Column();
            c.name = "desc".getBytes("utf-8");
            c.value = product.getDesc().getBytes("utf-8");
            c.timestamp = System.currentTimeMillis();

            ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
            cosc.column = c;
            Mutation mutation = new Mutation();
            mutation.column_or_supercolumn = cosc;
            mutationList.add(mutation);
        }

        if (product.getName() != null && !product.getName().isEmpty()) {
            Column c = new Column();
            c.name = "name".getBytes("utf-8");
            c.value = product.getName().getBytes("utf-8");
            c.timestamp = System.currentTimeMillis();

            ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
            cosc.column = c;
            Mutation mutation = new Mutation();
            mutation.column_or_supercolumn = cosc;
            mutationList.add(mutation);
        }

        if (product.getSellerUserName() != null
            &&! product.getSellerUserName().isEmpty()) {
            Column c = new Column();
            c.name = "sellerUserName".getBytes("utf-8");
            c.value = product.getSellerUserName().getBytes("utf-8");
            c.timestamp = System.currentTimeMillis();

            ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
            cosc.column = c;
            Mutation mutation = new Mutation();
            mutation.column_or_supercolumn = cosc;
            mutationList.add(mutation);
```

```
            }

            if (!mutationList.isEmpty()) {
                    cfMutationMap.put(Product.ColumnFamily, mutationList);
                    mutationMap.put(product.getUuid().toString(), cfMutationMap);

    cassandraClient.batch_mutate(Product.keySpace, mutationMap,ConsistencyLev-
el.ONE);
            }
    }

    @ Override
    public void insertComment(UUID productUUID, Comment comment)
                    throws Exception {
        if (cassandraClient == null) {
                    throw new Exception("Can't connect to Cassandra.");
        }

        if (productUUID == null || productUUID.isNullUUID()) {
                    throw new Exception("Can't insert null product to Cassandra.");
        }

        if (comment == null || comment.getUuid().isNullUUID()) {
                    throw new Exception("Can't insert null comment to Cassandra.");
        }

Map<String, Map<String, List<Mutation>>>mutationMap = new HashMap
    <String, Map<String, List<Mutation>>>();
Map<String, List<Mutation>>cfMutationMap = new HashMap<String,
    List<Mutation>>();
List<Mutation>mutationList = new ArrayList<Mutation>();

List<Column>columns = new ArrayList<Column>();
if (comment.getCommentUserName() != null
    && !comment.getCommentUserName().isEmpty()) {
        Column c = new Column();
        c.name = "commentUserName".getBytes("utf-8");
        c.value = comment.getCommentUserName().getBytes("utf-8");
        c.timestamp = System.currentTimeMillis();

        columns.add(c);
}
if (comment.getContent() != null && !comment.getContent().isEmpty()) {
        Column c = new Column();
        c.name = "content".getBytes("utf-8");
        c.value = comment.getContent().getBytes("utf-8");
```

```
                c.timestamp = System.currentTimeMillis();

                columns.add(c);
        }
        if (!columns.isEmpty()) {
                SuperColumn sc = new SuperColumn();
                sc.name = comment.getUuid().toByteArray();
                sc.columns = columns;

                ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
                cosc.super_column = sc;
                Mutation mutation = new Mutation();
                mutation.column_or_supercolumn = cosc;
                mutationList.add(mutation);

                cfMutationMap.put(Comment.ColumnFamily, mutationList);
                mutationMap.put(productUUID.toString(), cfMutationMap);

cassandraClient.batch_mutate(Product.keySpace,
    mutationMap,ConsistencyLevel.ONE);
      }

}

@ Override
    public Product getProduct(UUID productUUID) throws Exception {
            if (cassandraClient == null) {
                    throw new Exception("Can't connect to Cassandra.");
            }

            if (productUUID == null || productUUID.isNullUUID()) {
                    throw new Exception("Can't get null product to Cassandra.");
            }

            ColumnParent columnParent = new ColumnParent();
            columnParent.column_family = Product.ColumnFamily;
            SliceRange range = new SliceRange(new byte[] {}, new byte[] {}, true,In-
teger.MAX_VALUE);
            SlicePredicate slicePredicate = new SlicePredicate();
            slicePredicate.slice_range = range;

            List<ColumnOrSuperColumn> productColumns =
                    cassandraClient.get_slice(Product.keySpace, productUUID.toS-
tring(), columnParent,
                    slicePredicate, ConsistencyLevel.ONE);
```

```
        Product product = new Product();
        product.setUuid(productUUID);
        for (ColumnOrSuperColumn cosc : productColumns) {
                Column c = cosc.column;
                String columnName = new String(c.name, "utf-8");
                String columnValue = new String(c.value, "utf-8");
                if ("sellerUserName".equals(columnName)) {
product.setSellerUserName(columnValue);
                } else if ("desc".equals(columnName)) {
                        product.setDesc(columnValue);
                } else if ("name".equals(columnName)) {
                        product.setName(columnValue);
                } else if ("price".equals(columnName)) {
product.setPrice(Double.parseDouble(columnValue));
                }
        }

        return product;
}

@ Override
public void insertProductCategory(String category, UUID productUUID)
                throws Exception {
        if (cassandraClient == null) {
            throw new Exception("Can't connect to Cassandra.");
        }

        if (category == null || category.isEmpty()) {
            throw new Exception("Can't insert null category to Cassandra.");
        }

        if (productUUID == null || productUUID.isNullUUID()) {
            throw new Exception("Can't insert null product to Cassandra.");
        }

        ColumnPath columnPath = new ColumnPath();
        columnPath.column_family = "ProductCategory";
        columnPath.column = productUUID.toByteArray();

        cassandraClient.insert(Product.keySpace, category, columnPath,
            new byte[] {},
System.currentTimeMillis(), ConsistencyLevel.ONE);

}

@ Override
```

```
public List < Product > getTopCategoryProducts(String category, int topNum)
            throws Exception {
        if (cassandraClient == null) {
            throw new Exception("Can't connect to Cassandra.");
        }

        if (category == null || category.isEmpty()) {
            throw new Exception("Can't get null category to Cassandra.");
        }

        if (topNum < 1) {
            throw new Exception("topNum must larger than 1.");
        }

        ColumnParent columnParent = new ColumnParent();
        columnParent.column_family = "ProductCategory";
        SliceRange range = new SliceRange(new byte[] {}, new byte[] {}, true,
            topNum);
        SlicePredicate slicePredicate = new SlicePredicate();
        slicePredicate.slice_range = range;

        List < ColumnOrSuperColumn > productColumns =
            cassandraClient.get_slice(Product.keySpace, category,
            columnParent, slicePredicate,ConsistencyLevel.ONE);
        List < Product > products = new ArrayList < Product > ();
        for (ColumnOrSuperColumn cosc : productColumns) {

            Column c = cosc.column;
            UUID productUUID = new UUID(c.name);

            products.add(getProduct(productUUID));
        }

        return products;
}

@ Override
public List < Comment > getTopProductComments(UUID productUUID, int topNum)
            throws Exception {
        if (cassandraClient == null) {
            throw new Exception("Can't connect to Cassandra.");
        }

        if (productUUID == null || productUUID.isNullUUID()) {
            throw new Exception("Can't get null product to Cassandra.");
```

```
        }

        if (topNum < 1) {
            throw new Exception("topNum must larger than 1.");
        }
        ColumnParent columnParent = new ColumnParent();
        columnParent.column_family = Comment.ColumnFamily;
        SliceRange range = new SliceRange(new byte[] {}, new byte[] {}, true,
                topNum);
        SlicePredicate slicePredicate = new SlicePredicate();
        slicePredicate.slice_range = range;

        List<ColumnOrSuperColumn> commentColumns =
            cassandraClient.get_slice(Product.keySpace,
            productUUID.toString(), columnParent,slicePredicate,
            ConsistencyLevel.ONE);

        List<Comment> comments = new ArrayList<Comment>();
        for (ColumnOrSuperColumn cosc : commentColumns) {
            SuperColumn sc = cosc.super_column;
            Comment comment = new Comment();
            comment.setUuid(new UUID(sc.name));
            for (Column c : sc.columns) {
                String columnName = new String(c.name, "utf-8");
                String columnValue = new String(c.value, "utf-8");
                if ("commentUserName".equals(columnName)) {
                    comment.setCommentUserName(columnValue);
                } else if ("content".equals(columnName)) {
                    comment.setContent(columnValue);
                }
            }

            comments.add(comment);
        }

        return comments;
    }

}
```

B. 7 cassSeller. dao. impl. SellerDaoImpl

```
package cassSeller.dao.impl;
```

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.apache.cassandra.thrift.Column;
import org.apache.cassandra.thrift.ColumnOrSuperColumn;
import org.apache.cassandra.thrift.ColumnParent;
import org.apache.cassandra.thrift.ConsistencyLevel;
import org.apache.cassandra.thrift.Mutation;
import org.apache.cassandra.thrift.SlicePredicate;
import org.apache.cassandra.thrift.SliceRange;
import org.apache.cassandra.thrift.Cassandra.Client;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.transport.TSocket;

import cassSeller.dao.SellerDao;
import cassSeller.model.Seller;

public class SellerDaoImpl implements SellerDao{

    Client cassandraClient = null;

    public SellerDaoImpl(String host, int port) {
        TSocket socket = new TSocket(host, port);

        TBinaryProtocol binaryProtocol = new TBinaryProtocol(socket, false,
                false);
        cassandraClient = new Client(binaryProtocol);

        try {
                socket.open();
        } catch (Exception e) {
                e.printStackTrace();
                cassandraClient = null;
                return;
        }
}

@ Override
public void insertSeller(Seller seller) throws Exception {
        if (cassandraClient == null) {
            throw new Exception("Can't connect to Cassandra.");
        }

        if (seller == null) {
```

```
            throw new Exception("Can't insert null seller to Cassandra.");
        }

        if (seller.getUserName() == null || seller.getUserName().isEmpty()) {
            throw new Exception("Can't insert null sellerUserName
                to Cassandra.");
        }

        Map<String, Map<String, List<Mutation>>> mutationMap = new
                HashMap<String, Map<String, List<Mutation>>>();
        Map<String, List<Mutation>> cfMutationMap = new HashMap<String,
                List<Mutation>>();
        List<Mutation> mutationList = new ArrayList<Mutation>();

        //设置卖家信息
        if (seller.getAge() > 0) {
                Column c = new Column();
                c.name = "age".getBytes("utf-8");
                c.value = String.valueOf(seller.getAge()).getBytes("utf-8");
                c.timestamp = System.currentTimeMillis();

                ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
                cosc.column = c;
                Mutation mutation = new Mutation();
                mutation.column_or_supercolumn = cosc;
                mutationList.add(mutation);
        }

        if (seller.getAddress() != null && !seller.getAddress().isEmpty()) {
                Column c = new Column();
                c.name = "address".getBytes("utf-8");
                c.value = seller.getAddress().getBytes("utf-8");
                c.timestamp = System.currentTimeMillis();

                ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
                cosc.column = c;
                Mutation mutation = new Mutation();
                mutation.column_or_supercolumn = cosc;
                mutationList.add(mutation);
        }

        if (seller.getSex() != null && !seller.getSex().isEmpty()) {
                Column c = new Column();
                c.name = "sex".getBytes("utf-8");
                c.value = seller.getSex().getBytes("utf-8");
                c.timestamp = System.currentTimeMillis();
```

```
            ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
            cosc.column = c;
            Mutation mutation = new Mutation();
            mutation.column_or_supercolumn = cosc;
            mutationList.add(mutation);
    }

    if (seller.getName() != null && !seller.getName().isEmpty()) {
            Column c = new Column();
            c.name = "name".getBytes("utf-8");
            c.value = seller.getName().getBytes("utf-8");
            c.timestamp = System.currentTimeMillis();

            ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
            cosc.column = c;
            Mutation mutation = new Mutation();
            mutation.column_or_supercolumn = cosc;
            mutationList.add(mutation);
    }

    if (!mutationList.isEmpty()) {
        cfMutationMap.put(Seller.ColumnFamily, mutationList);
        mutationMap.put(seller.getUserName(), cfMutationMap);
        cassandraClient.batch_mutate(Seller.keySpace, mutationMap,ConsistencyLevel.ONE);
            }
    }

    @ Override
    public Seller getSeller(String sellerUserName) throws Exception {
            if (cassandraClient == null) {
                throw new Exception("Can't connect to Cassandra.");
            }

            if (sellerUserName == null || sellerUserName.isEmpty()) {
                throw new Exception("Can't get null buyer to Cassandra.");
            }

            ColumnParent columnParent = new ColumnParent();
            columnParent.column_family = Seller.ColumnFamily;
            SliceRange range = new SliceRange(new byte[] {}, new byte[] {}, true,In-
teger.MAX_VALUE);
            SlicePredicate slicePredicate = new SlicePredicate();
            slicePredicate.slice_range = range;
```

```
            List < ColumnOrSuperColumn > sellerColumns = cassandraClient.get _
slice(Seller.keySpace, sellerUserName, columnParent, slicePredicate,Consisten-
cyLevel.ONE);

            Seller seller = new Seller();
            seller.setUserName(sellerUserName);
            for (ColumnOrSuperColumn cosc : sellerColumns) {
                  Column c = cosc.column;
                  String columnName = new String(c.name, "utf-8");
                  String columnValue = new String(c.value, "utf-8");
                  if ("age".equals(columnName)) {
    seller.setAge(Integer.parseInt(columnValue));
                  } else if ("address".equals(columnName)) {
                        seller.setAddress(columnValue);
                  } else if ("sex".equals(columnName)) {
                        seller.setSex(columnValue);
                  } else if ("name".equals(columnName)) {
                        seller.setName(columnValue);
                  }
            }

            return seller;
      }
}
```

B. 8 cassSeller. model. Buyer

```
package cassSeller.model;


public class Buyer {
    public static final String keySpace = "CassSeller";
    public static final String ColumnFamily = "Buyer";

    private String userName;

    private String name;
    private int age;
    private String sex;
    private String address;

    public String getUserName() {
          return userName;
    }
```

```
public void setUserName(String userName) {
        this.userName = userName;
}

public String getName() {
        return name;
}

public void setName(String name) {
        this.name = name;
}

public int getAge() {
        return age;
}

public void setAge(int age) {
        this.age = age;
}

public String getSex() {
        return sex;
}

    public void setSex(String sex) {
            this.sex = sex;
    }

    public String getAddress() {
            return address;
    }

    public void setAddress(String address) {
            this.address = address;
    }
}
```

B. 9 cassSeller. model. Comment

```
package cassSeller.model;

import org.safehaus.uuid.UUID;

public class Comment {
```

```
public static final String keySpace = "CassSeller";
public static final String ColumnFamily = "Comment";

private UUID uuid;
private String content;
private String commentUserName;

public UUID getUuid() {
        return uuid;
}

public void setUuid(UUID uuid) {
        this.uuid = uuid;
}

public String getContent() {
        return content;
}

public void setContent(String content) {
        this.content = content;
}

public String getCommentUserName() {
        return commentUserName;
    }

    public void setCommentUserName(String commentUserName) {
        this.commentUserName = commentUserName;
    }

}
```

B. 10 cassSeller. model. Product

```
package cassSeller.model;

import org.safehaus.uuid.UUID;

public class Product {
    public static final String keySpace = "CassSeller";
    public static final String ColumnFamily = "Product";

    private UUID uuid;
```

```
private String name;
private String sellerUserName;
private String desc;
private double price;

public UUID getUuid() {
        return uuid;
}

public void setUuid(UUID uuid) {
        this.uuid = uuid;
}

public String getName() {
        return name;
}

public void setName(String name) {
        this.name = name;
}

public String getDesc() {
        return desc;
}

    public void setDesc(String desc) {
            this.desc = desc;
    }

    public double getPrice() {
            return price;
    }

    public void setPrice(double price) {
            this.price = price;
    }

    public String getSellerUserName() {
            return sellerUserName;
    }

    public void setSellerUserName(String sellerUserName) {
            this.sellerUserName = sellerUserName;
    }

}
```

B. 11 cassSeller. model. Seller

```
package cassSeller.model;

public class Seller {
    public static final String keySpace = "CassSeller";
    public static final String ColumnFamily = "Seller";

    private String userName;

    private String name;
    private int age;
    private String sex;
    private String address;

    public String getUserName() {
            return userName;
    }

    public void setUserName(String userName) {
            this.userName = userName;
    }

    public String getName() {
            return name;
    }

    public void setName(String name) {
            this.name = name;
    }

    public int getAge() {
            return age;
    }

    public void setAge(int age) {
            this.age = age;
    }

    public String getSex() {
            return sex;
    }

    public void setSex(String sex) {
```

```
            this.sex = sex;
        }

        public String getAddress() {
            return address;
        }

        public void setAddress(String address) {
            this.address = address;
        }
}
```

附录 C　CassSeller－0.7 代码

本书第 4 章中编写了一个基于 Cassandra 的在线交易系统。本章将列出该项目中所有的源代码，Cassandra 的执行版本为 0.7. x。项目代码结构如图 C-1 所示。

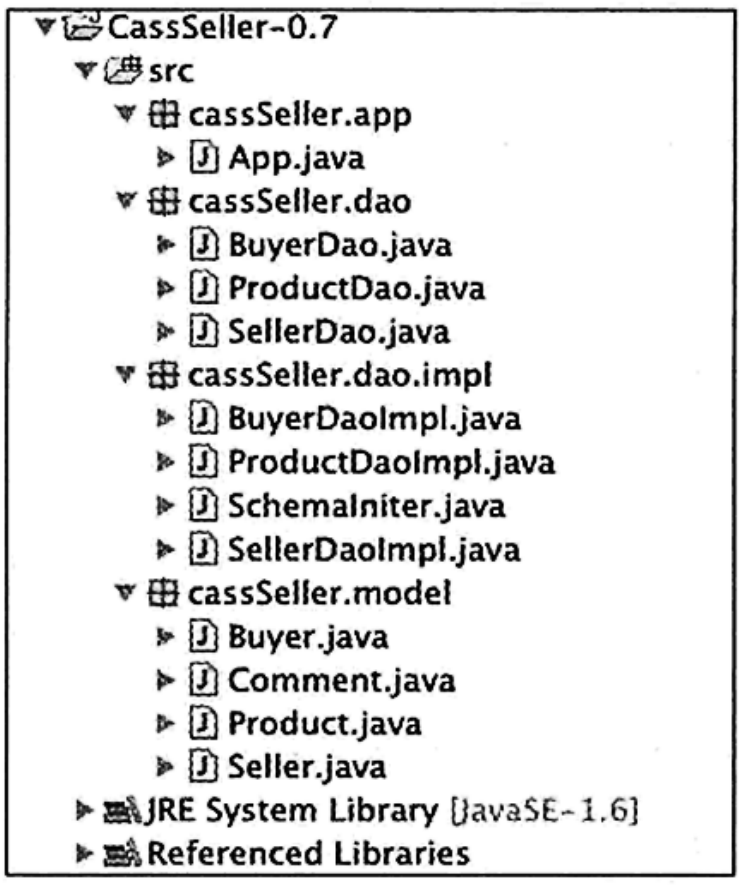


图 C-1　项目代码结构

项目依赖库如图 C-2 所示。

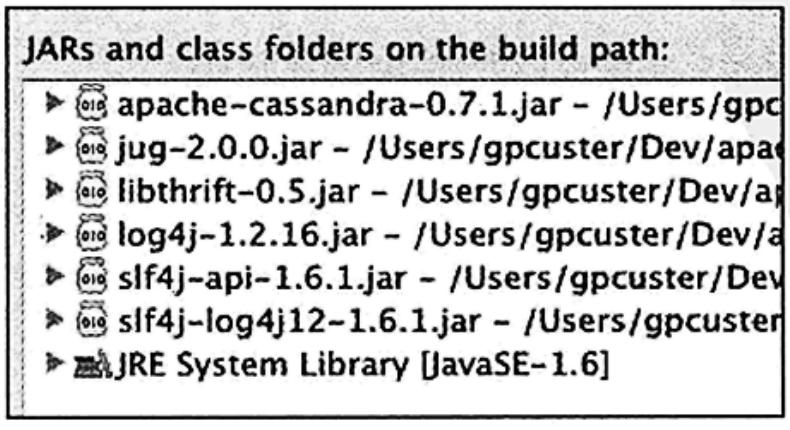


图 C-2　项目依赖库

下面分别给出各项目源代码。

C.1 cassSeller.app.App

```
package cassSeller.app;

import java.lang.reflect.Field;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.util.List;

import org.safehaus.uuid.UUIDGenerator;

import cassSeller.dao.BuyerDao;
import cassSeller.dao.ProductDao;
import cassSeller.dao.SellerDao;
import cassSeller.dao.impl.BuyerDaoImpl;
import cassSeller.dao.impl.ProductDaoImpl;
import cassSeller.dao.impl.SellerDaoImpl;
import cassSeller.model.Buyer;
import cassSeller.model.Comment;
import cassSeller.model.Product;
import cassSeller.model.Seller;

public class App {
    /**
     * 获取对象实例中所包含的属性的名称和值的字符串
     *
     * @param entityName
     * 需要打印的对象实例
     * @return 对象实例中所包含的属性的名称和值的字符串
     * @throws Exception
     */
    static String getPropertyString(Object entityName)
throws Exception {
        Class<?> c = entityName.getClass();
        Field field[] = c.getDeclaredFields();
        StringBuffer sb = new StringBuffer();

        for (Field f : field) {
            if ((f.getModifiers() &
                Modifier.STATIC) != 0) {
                continue;
                }
                sb.append(f.getName());
                sb.append(" : ");
                sb.append(invokeMethod(entityName,
```

```
                    f.getName(), null));
                sb.append("n");
        }
        return sb.toString();
}

static Object invokeMethod(Object owner, String
    methodName, Object[] args)
    throws Exception {
        Class<?> ownerClass = owner.getClass();

        methodName = methodName.substring(0,
            1).toUpperCase() + methodName.substring(1);
        Method method = null;
        try {
            method = ownerClass.getMethod("get" + methodName);
        } catch (Exception e) {
        }

        return method.invoke(owner);
}

/**
 * @param args
 */
public static void main(String[] args) {
        BuyerDao buyerDao = new BuyerDaoImpl("localhost",9160);

Buyer buyer1 = new Buyer();
buyer1.setUserName("aaron");
buyer1.setName("郭鹏");
buyer1.setAddress("China Hangzhou");
buyer1.setAge(26);
buyer1.setSex("male");

Buyer buyer2 = new Buyer();
buyer2.setUserName("lily");
buyer2.setName("李娜");
buyer2.setAddress("China 大连");
buyer2.setAge(23);
buyer2.setSex("female");

try {
        buyerDao.insertBuyer(buyer1);
        buyerDao.insertBuyer(buyer2);
} catch (Exception e) {
        e.printStackTrace();
}

try {
```

```
            Buyer aaron = buyerDao.getBuyer("aaron");
System.out.println(getPropertyString(aaron));
        } catch (Exception e) {
            e.printStackTrace();
        }

        SellerDao sellerDao = new SellerDaoImpl("localhost", 9160);

        Seller seller1 = new Seller();
        seller1.setUserName("sportSeller");
        seller1.setName("体育用品专卖");
        seller1.setAddress("China Wuhan");
        seller1.setAge(30);
        seller1.setSex("male");

        Seller seller2 = new Seller();
        seller2.setUserName("seller2");
        seller2.setName("美丽服装");
        seller2.setAddress("China Beijing");
        seller2.setAge(37);
        seller2.setSex("male");

        try {
            sellerDao.insertSeller(seller1);
            sellerDao.insertSeller(seller2);
        } catch (Exception e) {
            e.printStackTrace();
        }

        try {
            Seller sportSeller =sellerDao.getSeller("sportSeller");

System.out.println(getPropertyString(sportSeller));
        } catch (Exception e) {
            e.printStackTrace();
        }

        ProductDao productDao = new ProductDaoImpl("localhost", 9160);

        Product product1 = new Product();
        product1.setDesc("白色真皮足球");
        product1.setSellerUserName("sportSeller");
        product1.setName("Football");
        product1.setPrice(98.8);
product1.setUuid(UUIDGenerator.getInstance().generateTimeBasedUUID());

        Product product2 = new Product();
        product2.setDesc("橡胶篮球");
        product2.setSellerUserName("sportSeller");
```

```
        product2.setName("Basketball");
        product2.setPrice(29.8);
product2.setUuid(UUIDGenerator.getInstance().generateTimeBasedUUID());

        try {
                productDao.insertProduct(product1);
                productDao.insertProduct(product2);
        } catch (Exception e) {
                e.printStackTrace();
        }

        try {

                Product football = productDao.getProduct(product1.getUuid());
System.out.println(getPropertyString(football));
        } catch (Exception e) {
                e.printStackTrace();
        }

        try {
productDao.insertProductCategory("Sports", product1.getUuid());
productDao.insertProductCategory("Sports", product2.getUuid());
        } catch (Exception e) {
                e.printStackTrace();
        }

        try {
                List<Product> products =
                    productDao.getTopCategoryProducts("Sports", 2);
                for (Product product : products) {
System.out.println(getPropertyString(product));
                }
        } catch (Exception e) {
                e.printStackTrace();
        }

        Comment comment1 = new Comment();

comment1.setUuid(UUIDGenerator.getInstance().generateTimeBasedUUID());
        comment1.setCommentUserName("aaron");
        comment1.setContent("good product");

        Comment comment2 = new Comment();

comment2.setUuid(UUIDGenerator.getInstance().generateTimeBasedUUID());
        comment2.setCommentUserName("aaron");
        comment2.setContent("送货速度快");

        Comment comment3 = new Comment();
```

```
comment3.setUuid(UUIDGenerator.getInstance().generateTimeBasedUUID());
        comment3.setCommentUserName("lily");
        comment3.setContent("弟弟很喜欢这个礼物");

        try {
productDao.insertComment(product2.getUuid(), comment1);

productDao.insertComment(product2.getUuid(),comment2);

productDao.insertComment(product2.getUuid(),comment3);
      } catch (Exception e) {
                e.printStackTrace();
      }

        try {
                List<Comment> comments =
                    productDao.getTopProductComments(
                                product2.getUuid(), 2);
                for (Comment comment : comments) {

System.out.println(getPropertyString(comment));
                }
        } catch (Exception e) {
                e.printStackTrace();
        }

}
```

C.2 cassSeller. dao. BuyerDao

```
package cassSeller.dao;

import cassSeller.model.Buyer;

public interface BuyerDao {

    /**
     * 将 Buyer 实例插入到 Cassandra 中
     *
     * @param buyer
     * 需要插入的 Buyer 实例
     * @throws Exception
     */
     public void insertBuyer(Buyer buyer) throws Exception;

    /**
     * 根据 buyerUserName,从 Cassandra 中查询对应的 Buyer 实例
```

```
     *
     * @ param buyerUserName
     * Buyer ColumnFalimy 的 key
     * @ return Buyer 实例
     * @ throws Exception
     * /
    public Buyer getBuyer(String buyerUserName) throws Exception;
}
```

C.3　cassSeller. dao. ProductDao

```
package cassSeller.dao;

import java.util.List;

import org.safehaus.uuid.UUID;

import cassSeller.model.Comment;
import cassSeller.model.Product;

public interface ProductDao {

    /**
     * 将 Product 实例插入到 Cassandra 中
     *
     * @ param product
     * 需要插入的 Product 实例
     * @ throws Exception
     * /
    public void insertProduct(Product product) throws Exception;

    /**
     * 将产品分类信息插入到 Cassandra 中
     *
     * @ param category
     * 产品的分类
     * @ param productUUID
     * Product ColumnFamily 的 key
     * @ throws Exception
     /
    public void insertProductCategory(String category, UUID productUUID)throws
Exception;

    /**
     * 将产品评论信息插入到 Cassandra 中
     *
     * @ param productUUID
     * Product ColumnFamily 的 key
```

```
     * @ param comment
     * 评论的内容
     * @ throws Exception
     */
    public void insertComment(UUID productUUID, Comment comment)throws Excep-
tion;

    /**
     * 根据 productUUID,从 Cassandra 中查询对应的 Product 实例
     *
     * @ param productUUID
     * Product ColumnFamily 的 key
     * @ return Product 实例
     * @ throws Exception
     */
    public Product getProduct(UUID productUUID) throws Exception;

    /**
     * 根据 category,从 Cassandra 中查询最新的 topNum 个 Product 实例
     *
     * @ param category
     * 产品的分类
     * @ param topNum
     * 查询最新的实例的个数
     * @ return 最新的 topNum 个 Product 实例
     * @ throws Exception
     */
    public List < Product > getTopCategoryProducts(String category, int topNum)
throws Exception;

    /**
     * 根据 productUUID,从 Cassandra 中查询最新的 topNum 个 Product 评论实例
     *
     * @ param productUUID
     * Product ColumnFamily 的 key
     * @ param topNum
     * 查询最新的实例的个数
     * @ return 最新的 topNum 个 Product 评论实例
     * @ throws Exception
     */
    public List < Comment > getTopProductComments(UUID productUUID, int topNum)
throws Exception;
}
```

C. 4 cassSeller. dao. SellerDao

```
package cassSeller.dao;
```

```
import cassSeller.model.Seller;

public interface SellerDao {

    /**
     * 将 Seller 实例插入到 Cassandra 中
     *
     * @ param seller
     * 根据 sellerUserName,从 Cassandra 中查询对应的 Seller 实例
     *
     * @ param sellerUserName
     * Seller ColumnFamily 的 key
     * @ return Seller 实例
     * @ throws Exception
     */
    public Seller getSeller(String sellerUserName) throws Exception;
}
```

C.5　cassSeller. dao. impl. BuyerDaoImpl

```
package cassSeller.dao.impl;

import java.nio.ByteBuffer;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.apache.cassandra.thrift.Cassandra.Client;
import org.apache.cassandra.thrift.Column;
import org.apache.cassandra.thrift.ColumnOrSuperColumn;
import org.apache.cassandra.thrift.ColumnParent;
import org.apache.cassandra.thrift.ConsistencyLevel;
import org.apache.cassandra.thrift.Mutation;
import org.apache.cassandra.thrift.SlicePredicate;
import org.apache.cassandra.thrift.SliceRange;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.transport.TFramedTransport;
import org.apache.thrift.transport.TSocket;
import org.apache.thrift.transport.TTransport;

import cassSeller.dao.BuyerDao;
import cassSeller.model.Buyer;

/**
 * @ author gpcuster
 *
```

```
 */
public class BuyerDaoImpl implements BuyerDao {

    Client cassandraClient = null;

    public BuyerDaoImpl(String host, int port) {
            TSocket socket = new TSocket(host, port);

            TTransport transport = new TFramedTransport(socket);

            TBinaryProtocol binaryProtocol = new TBinaryProtocol(transport,
true, true);
            cassandraClient = new Client(binaryProtocol);

            try {
                    transport.open();
            } catch (Exception e) {
                    e.printStackTrace();
                    cassandraClient = null;
                    return;
            }

            try {
    cassandraClient.set_keyspace(Buyer.keySpace);
            } catch (Exception e) {
                    e.printStackTrace();
                    try {
SchemaIniter.init(cassandraClient);

cassandraClient.set_keyspace(Buyer.keySpace);
                } catch (Exception e1) {
                        e1.printStackTrace();
                        cassandraClient = null;
                }
        }
}

@ Override
public void insertBuyer(Buyer buyer) throws Exception {
        if (cassandraClient = = null) {
                throw new Exception("Can't connect to Cassandra.");
        }

        if (buyer = = null) {
            throw new Exception("Can't insert null buyer to Cassandra.");
        }

        if (buyer.getUserName() = = null ||
              buyer.getUserName().isEmpty()) {
              throw new Exception("Can't insert null
```

```
            buyerUserName to Cassandra.");
}

Map<ByteBuffer, Map<String, List<Mutation>>>
      mutationMap = new HashMap<ByteBuffer, Map<String, List<Muta-
      tion>>>();
Map<String, List<Mutation>> cfMutationMap = new
      HashMap<String, List<Mutation>>();
List<Mutation> mutationList = new ArrayList<Mutation>();

//设置买家信息
if (buyer.getAge() > 0) {
        Column c = new Column();
        c.name =
              ByteBuffer.wrap("age".getBytes("utf-8"));
        c.value =
                  ByteBuffer.wrap (String.valueOf (buyer.getAge ( ))
                  .getBytes("utf-8"));
        c.timestamp = System.currentTimeMillis();

        ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
        cosc.column = c;
        Mutation mutation = new Mutation();
        mutation.column_or_supercolumn = cosc;
        mutationList.add(mutation);
}

if (buyer.getAddress() != null && !buyer.getAddress().isEmpty()) {
        Column c = new Column();
        c.name =
            ByteBuffer.wrap("address".getBytes("utf-8"));
        c.value =
            ByteBuffer.wrap(buyer.getAddress().getBytes("utf-8"));
        c.timestamp =
            System.currentTimeMillis();

        ColumnOrSuperColumn cosc = new
            ColumnOrSuperColumn();
        cosc.column = c;
        Mutation mutation = new Mutation();
        mutation.column_or_supercolumn = cosc;
        mutationList.add(mutation);
}

if (buyer.getSex() != null
    && !buyer.getSex().isEmpty()) {
        Column c = new Column();
        c.name =
            ByteBuffer.wrap("sex".getBytes("utf-8"));
        c.value =
```

```
                ByteBuffer.wrap(buyer.getSex().getBytes("utf-8"));
            c.timestamp =
                System.currentTimeMillis();
            ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
            cosc.column = c;
            Mutation mutation = new Mutation();
            mutation.column_or_supercolumn = cosc;
            mutationList.add(mutation);
        }

        if (buyer.getName()!=null && !buyer.getName().isEmpty()) {
            Column c = new Column();
            c.name =
                ByteBuffer.wrap("name".getBytes("utf-8"));
            c.value =
                ByteBuffer.wrap(buyer.getName().getBytes("utf-8"));
            c.timestamp =
                System.currentTimeMillis();

            ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
            cosc.column = c;
            Mutation mutation=new Mutation();
            mutation.column_or_supercolumn=cosc;
            mutationList.add(mutation);
        }

        if (!mutationList.isEmpty()) {
            cfMutationMap.put(Buyer.ColumnFamily, mutationList);
            mutationMap.put(
                ByteBuffer.wrap(buyer.getUserName().getBytes("utf-8")),
cfMutationMap);

                cassandraClient.batch_mutate(mutationMap, ConsistencyLev-
el.ONE);
            }
    }

    @ Override
    public Buyer getBuyer(String buyerUserName) throws Exception {
        if (cassandraClient = = null) {
            throw new Exception("Can't connect to Cassandra.");
        }

        if (buyerUserName = = null ||buyerUserName.isEmpty()) {
            throw new Exception("Can't get null buyer to Cassandra.");
        }

        ColumnParent columnParent = new ColumnParent();
        columnParent.column_family =Buyer.ColumnFamily;
        SliceRange range = new SliceRange(ByteBuffer.wrap(new byte[] {}),
```

```
ByteBuffer.wrap(new byte[] {}), true, Integer.MAX_VALUE);
            SlicePredicate slicePredicate = new SlicePredicate();
            slicePredicate.slice_range = range;

            List < ColumnOrSuperColumn > buyerColumns = cassandraClient.get_
slice(ByteBuffer.wrap (buyerUserName.getBytes ( "utf - 8 ")), columnParent, sli-
cePredicate, ConsistencyLevel.ONE);

            Buyer buyer = new Buyer();
            buyer.setUserName(buyerUserName);
            for (ColumnOrSuperColumn cosc : buyerColumns) {
                  Column c = cosc.column;
                  String columnName = new String(c.getName(), "utf-8");
                  String columnValue = new String(c.getValue(), "utf-8");
                  if ("age".equals(columnName)) {
   buyer.setAge(Integer.parseInt(columnValue));
                  } else if ("address".equals(columnName)) {

                        buyer.setAddress(columnValue);
                  } else if ("sex".equals(columnName)) {
                        buyer.setSex(columnValue);
                  } else if ("name".equals(columnName)) {
                        buyer.setName(columnValue);
                  }
            }

            return buyer;
      }
 }
```

C.6　cassSeller. dao. impl. ProductDaoImpl

```
package cassSeller.dao.impl;

import java.nio.ByteBuffer;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.apache.cassandra.thrift.Cassandra.Client;
import org.apache.cassandra.thrift.Column;
import org.apache.cassandra.thrift.ColumnOrSuperColumn;
import org.apache.cassandra.thrift.ColumnParent;
import org.apache.cassandra.thrift.ConsistencyLevel;
import org.apache.cassandra.thrift.Mutation;
import org.apache.cassandra.thrift.SlicePredicate;
import org.apache.cassandra.thrift.SliceRange;
```

```
import org.apache.cassandra.thrift.SuperColumn;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.transport.TFramedTransport;
import org.apache.thrift.transport.TSocket;
import org.apache.thrift.transport.TTransport;
import org.safehaus.uuid.UUID;

import cassSeller.dao.ProductDao;
import cassSeller.model.Comment;
import cassSeller.model.Product;

public class ProductDaoImpl implements ProductDao {

      Client cassandraClient = null;

      public ProductDaoImpl(String host, int port) {
            TSocket socket = new TSocket(host, port);

            TTransport transport = new TFramedTransport(socket);

            TBinaryProtocol binaryProtocol = new TBinaryProtocol(transport, true,
true);
            cassandraClient = new Client(binaryProtocol);

            try {
                  transport.open();
            } catch (Exception e) {
                  e.printStackTrace();
                  cassandraClient = null;
                  return;
            }

            try {
cassandraClient.set_keyspace(Product.keySpace);
            } catch (Exception e) {
                  e.printStackTrace();
                  try {
SchemaIniter.init(cassandraClient);

cassandraClient.set_keyspace(Product.keySpace);
                  } catch (Exception e1) {
                        e1.printStackTrace();
                        cassandraClient = null;
                  }
            }
}

@ Override
public void insertProduct(Product product) throws Exception {
      if (cassandraClient = = null) {
```

```
                throw new Exception("Can't connect to Cassandra.");
            }

            if (product == null) {
                throw new Exception("Can't insert null product to Cassandra.");
            }

            if (product.getUuid() == null ||product.getUuid().isNullUUID()) {
                    throw new Exception ( " Can't insert null product to
Cassandra.");
            }

            Map<ByteBuffer, Map<String, List<Mutation>>>
mutationMap = new HashMap<ByteBuffer, Map<String, List<Mutation>>>();
              Map<String, List<Mutation>> cfMutationMap = new
                  HashMap<String, List<Mutation>>();
              List<Mutation> mutationList = new ArrayList<Mutation>();

              //设置产品信息
              if (product.getPrice() > 0) {
                    Column c = new Column();
                    c.name =
                       ByteBuffer.wrap("price".getBytes("utf-8"));
                    c.value =
                       ByteBuffer.wrap(String.valueOf(product.getPrice()).getBytes
("utf-8"));
                    c.timestamp = System.currentTimeMillis();

                    ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
                    cosc.column = c;
                    Mutation mutation = new Mutation();
                    mutation.column_or_supercolumn = cosc;
                    mutationList.add(mutation);
              }

              if (product.getDesc()!= null && !product.getDesc().isEmpty()) {
                    Column c = new Column();
                    c.name =
                        ByteBuffer.wrap("desc".getBytes("utf-8"));
                    c.value =
                        ByteBuffer.wrap(product.getDesc().getBytes("utf-8"));
                    c.timestamp =
                        System.currentTimeMillis();

                    ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
                    cosc.column = c;
                    Mutation mutation = new Mutation();
                    ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
                    cosc.column = c;
                    Mutation mutation = new Mutation();
```

```
                mutation.column_or_supercolumn = cosc;
                mutationList.add(mutation);
            }

            if (product.getName() != null && !product.getName().isEmpty()) {
                Column c = new Column();
                c.name =
                    ByteBuffer.wrap("name".getBytes("utf-8"));
                c.value =
                    ByteBuffer.wrap(product.getName().getBytes("utf-8"));
                c.timestamp =
                    System.currentTimeMillis();

                ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
                cosc.column = c;
                Mutation mutation = new Mutation();
                mutation.column_or_supercolumn = cosc;
                mutationList.add(mutation);
            }

            if (product.getSellerUserName()!= null && !product. getSellerUser-
Name().isEmpty()) {
                Column c = new Column();
                c.name =
                    ByteBuffer.wrap("sellerUserName".getBytes("utf-8"));
                c.value =
                    ByteBuffer.wrap(product.getSellerUserName().getBytes
("utf-8"));
                c.timestamp =
                    System.currentTimeMillis();

                ColumnOrSuperColumn cosc =new ColumnOrSuperColumn();
                cosc.column = c;
                Mutation mutation = new Mutation();
                mutation.column_or_supercolumn = cosc;
                mutationList.add(mutation);
            }

            if (!mutationList.isEmpty()) {
                cfMutationMap.put(Product.ColumnFamily, mutationList);
        mutationMap.put(ByteBuffer.wrap(product.getUuid().asByteArray()),cfMuta-
tionMap);

        cassandraClient.batch_mutate(mutationMap, ConsistencyLevel.ONE);
            }
        }

        @Override
        public void insertComment(UUID productUUID, Comment comment)
                throws Exception {
```

```
                if (cassandraClient = = null) {
                        throw new Exception("Can't connect to Cassandra.");
                }
                if (productUUID = = null ||productUUID.isNullUUID()) {
                            throw new Exception ( " Can't insert null product to
Cassandra.");
                }

                if (comment = = null ||comment.getUuid().isNullUUID()) {
                        throw new Exception("Can't insert null comment to Cassan-
dra.");
                }

                 Map < ByteBuffer, Map < String, List < Mutation > > > mutationMap =
new HashMap < ByteBuffer, Map < String, List < Mutation > > > ();
                  Map < String, List < Mutation > > cfMutationMap = new HashMap <
String, List < Mutation > > ();
                List < Mutation > mutationList = new ArrayList < Mutation > ();

                List < Column > columns = new ArrayList < Column > ();
                if (comment.getCommentUserName() ! = null && !comment. getCommen-
tUserName().isEmpty()) {
                        Column c = new Column();
                        c.name =
                            ByteBuffer.wrap("commentUserName".getBytes("utf - 8"));
                        c.value =
                            ByteBuffer.wrap(comment.getCommentUserName().getBytes
("utf - 8"));
                        c.timestamp =
                            System.currentTimeMillis();

                        columns.add(c);
                }
                 if (comment.getContent()!= null && !comment.getContent().isEmpty
()) {
                        Column c = new Column();
                        c.name =
                            ByteBuffer.wrap("content".getBytes("utf - 8"));
                        c.value =
                            ByteBuffer.wrap(comment.getContent().getBytes("utf - 8"));
                        c.timestamp =
                            System.currentTimeMillis();

                        columns.add(c);
                }

                if (!columns.isEmpty()) {
                        SuperColumn sc = new SuperColumn();
                        sc.name =
                            ByteBuffer.wrap(comment.getUuid().toByteArray());
```

```
                    sc.columns = columns;

                    ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
                    cosc.super_column = sc;
                    Mutation mutation = new Mutation();
                    mutation.column_or_supercolumn = cosc;
                    mutationList.add(mutation);
                    cfMutationMap.put(Comment.ColumnFamily, mutationList);
mutationMap.put(ByteBuffer.wrap(productUUID.asByteArray()),cfMutationMap);

cassandraClient.batch_mutate(mutationMap, ConsistencyLevel.ONE);
          }

}

@ Override
public Product getProduct(UUID productUUID) throws Exception {
       if (cassandraClient = = null) {
              throw new Exception("Can't connect to Cassandra.");
       }

       if (productUUID = = null ||productUUID.isNullUUID()) {
              throw new Exception("Can't get null product to Cassandra.");
       }

       ColumnParent columnParent = new ColumnParent();
       columnParent.column_family = Product.ColumnFamily;
              SliceRange range = new SliceRange(ByteBuffer.wrap(new byte[] {}),Byte-
Buffer.wrap(new byte[] {}), true, Integer.MAX_VALUE);
              SlicePredicate slicePredicate = new SlicePredicate();
              slicePredicate.slice_range = range;

               List < ColumnOrSuperColumn > productColumns = cassandraClient.get_
slice(
        ByteBuffer.wrap (productUUID.asByteArray ()), columnParent, slicePredicate,
ConsistencyLevel.ONE);
              Product product = new Product();
              product.setUuid(productUUID);
              for (ColumnOrSuperColumn cosc : productColumns) {
                    Column c = cosc.column;
                    String columnName = new String(c.getName(), "utf-8");
                    String columnValue = new String(c.getValue(), "utf-8");
                    if ("sellerUserName".equals(columnName)) {
      product.setSellerUserName(columnValue);
                    } else if ("desc".equals(columnName)) {
                          product.setDesc(columnValue);
                    } else if ("name".equals(columnName)) {
                          product.setName(columnValue);
                    } else if ("price".equals(columnName)) {
      product.setPrice(Double.parseDouble(columnValue));
```

```
                }
            }

            return product;
        }

        @ Override
        public void insertProductCategory(String category, UUID productUUID)throws
Exception {
            if (cassandraClient = = null) {
                throw new Exception("Can't connect to Cassandra.");
            }

            if (category = = null || category.isEmpty()) {
                throw new Exception("Can't insert null category to Cassandra.");
            }

            if (productUUID = = null || productUUID.isNullUUID()) {
                throw new Exception("Can't insert null product to Cassandra.");
            }

            ColumnParent columnParent = new ColumnParent();
            columnParent.column_family = "ProductCategory";

            Column column = new Column();
            column.name =
                ByteBuffer.wrap(productUUID.toByteArray());
            column.value = ByteBuffer.wrap(new byte[] {});
            column.timestamp = System.currentTimeMillis();

        cassandraClient.insert(ByteBuffer.wrap(category.getBytes("utf-8")),colum-
nParent, column, ConsistencyLevel.ONE);

        }
        @ Override
        public List<Product> getTopCategoryProducts(String
            category, int topNum)
            throws Exception {
                if (cassandraClient = = null) {
                    throw new Exception("Can't connect to Cassandra.");
                }

                if (category = = null || category.isEmpty()) {
                    throw new Exception("Can't get null category to Cassandra.");
                }

                if (topNum < 1) {
                    throw new Exception("topNum must larger than 1.");
                }
```

```
        ColumnParent columnParent = new ColumnParent();
        columnParent.column_family = "ProductCategory";
        SliceRange range = new
            SliceRange(ByteBuffer.wrap(new byte[] {}),ByteBuffer.wrap(new
            byte[] {}), true, topNum);
        SlicePredicate slicePredicate = new SlicePredicate();
        slicePredicate.slice_range = range;

         List <ColumnOrSuperColumn > productColumns = cassandraClient.get_
              slice(ByteBuffer.wrap(category.getBytes("utf-8")), columnPar-
              ent,slicePredicate, ConsistencyLevel.ONE);

        List<Product> products = new ArrayList<Product>();
        for (ColumnOrSuperColumn cosc : productColumns) {

              Column c = cosc.column;
              UUID productUUID = new UUID(c.getName());

              products.add(getProduct(productUUID));
        }

        return products;
}

@Override
public List<Comment> getTopProductComments(UUID productUUID, int topNum)
    throws Exception {
        if (cassandraClient == null) {
              throw new Exception("Can't connect to Cassandra.");
        }

        if (productUUID == null ||productUUID.isNullUUID()) {
              throw new Exception("Can't get null product to Cassandra.");
        }

        if (topNum < 1) {
              throw new Exception("topNum must larger than 1.");
        }

        ColumnParent columnParent = new ColumnParent();
        columnParent.column_family = Comment.ColumnFamily;
        SliceRange range = new
            SliceRange(ByteBuffer.wrap(new byte[] {}),ByteBuffer.wrap(new
            byte[] {}), true, topNum);
        SlicePredicate slicePredicate = new SlicePredicate();
        slicePredicate.slice_range = range;

         List <ColumnOrSuperColumn > commentColumns = cassandraClient.get_
slice(ByteBuffer.wrap(productUUID.asByteArray()), columnParent,slicePredi-
```

```
    cate, ConsistencyLevel.ONE);

            List<Comment> comments = new ArrayList<Comment>();
            for (ColumnOrSuperColumn cosc : commentColumns) {
                SuperColumn sc = cosc.super_column;
                Comment comment = new Comment();
                comment.setUuid(new UUID(sc.getName()));
                for (Column c : sc.columns) {
                    String columnName = new String(c.getName(), "utf-8");
                    String columnValue =new String(c.getValue(), "utf-8");

                    if ("commentUserName".equals(columnName)) {
    comment.setCommentUserName(columnValue);
                    } else if ("content".equals(columnName)) {
    comment.setContent(columnValue);
                    }
                }

                comments.add(comment);
            }

            return comments;
    }

}
```

C.7　cassSeller.dao.impl.SchemaIniter

```
package cassSeller.dao.impl;

import java.util.ArrayList;
import java.util.List;

import org.apache.cassandra.thrift.Cassandra.Client;
import org.apache.cassandra.thrift.CfDef;
import org.apache.cassandra.thrift.KsDef;

import cassSeller.model.Buyer;
import cassSeller.model.Comment;
import cassSeller.model.Product;
import cassSeller.model.Seller;

public class SchemaIniter {

    /**
     * 为 Cassandra 系统构造 Schema
     *
```

```
     * @ param cassandraClient
     * @ throws Exception
     */
    public static void init(Client cassandraClient) throws Exception {
            List<CfDef> cfDefs = new ArrayList<CfDef>();

            CfDef buyerCfDef = new CfDef();
            buyerCfDef.keyspace = Buyer.keySpace;
            buyerCfDef.name = Buyer.ColumnFamily;
            buyerCfDef.comparator_type = "UTF8Type";
            cfDefs.add(buyerCfDef);

            CfDef sellerCfDef = new CfDef();
            sellerCfDef.keyspace = Seller.keySpace;
            sellerCfDef.name = Seller.ColumnFamily;
            sellerCfDef.comparator_type = "UTF8Type";
            cfDefs.add(sellerCfDef);

            CfDef productCategoryCfDef = new CfDef();
            productCategoryCfDef.keyspace = Product.keySpace;
            productCategoryCfDef.name = "ProductCategory";
            productCategoryCfDef.comparator_type = "TimeUUIDType";
            cfDefs.add(productCategoryCfDef);

            CfDef productCfDef = new CfDef();
            productCfDef.keyspace = Product.keySpace;
            productCfDef.name = "Product";
            productCfDef.comparator_type = "UTF8Type";
            cfDefs.add(productCfDef);

            CfDef commentCfDef = new CfDef();
            commentCfDef.keyspace = Comment.keySpace;
            commentCfDef.name = Comment.ColumnFamily;
            commentCfDef.column_type = "Super";
            commentCfDef.comparator_type = "TimeUUIDType";

            commentCfDef.setSubcomparator_type("UTF8Type");
            cfDefs.add(commentCfDef);

            KsDef ksDef = new KsDef();
            ksDef.name = Buyer.keySpace;
            ksDef.replication_factor = 1;
            ksDef.strategy_class = "org.apache.cassandra.locator. SimpleStrategy";
            ksDef.cf_defs = cfDefs;

            cassandraClient.system_add_keyspace(ksDef);
        }
    }
```

C.8　cassSeller. dao. impl. SellerDaoimpl

```
package cassSeller.dao.impl;

import java.nio.ByteBuffer;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import org.apache.cassandra.thrift.Cassandra.Client;
import org.apache.cassandra.thrift.Column;
import org.apache.cassandra.thrift.ColumnOrSuperColumn;
import org.apache.cassandra.thrift.ColumnParent;
import org.apache.cassandra.thrift.ConsistencyLevel;
import org.apache.cassandra.thrift.Mutation;
import org.apache.cassandra.thrift.SlicePredicate;
import org.apache.cassandra.thrift.SliceRange;
import org.apache.thrift.protocol.TBinaryProtocol;
import org.apache.thrift.transport.TFramedTransport;
import org.apache.thrift.transport.TSocket;
import org.apache.thrift.transport.TTransport;

import cassSeller.dao.SellerDao;
import cassSeller.model.Seller;

public class SellerDaoImpl implements SellerDao {

      Client cassandraClient = null;

      public SellerDaoImpl(String host, int port) {
            TSocket socket = new TSocket(host, port);

            TTransport transport = new TFramedTransport(socket);

            TBinaryProtocol binaryProtocol = new TBinaryProtocol(transport, true,
true);
            cassandraClient = new Client(binaryProtocol);

            try {
                  transport.open();
            } catch (Exception e) {
                  e.printStackTrace();
                  cassandraClient = null;
                  return;
            }
```

```
            try {

        cassandraClient.set_keyspace(Seller.keySpace);
            } catch (Exception e) {
                e.printStackTrace();
                try {
        SchemaIniter.init(cassandraClient);

        cassandraClient.set_keyspace(Seller.keySpace);
                } catch (Exception e1) {
                    e1.printStackTrace();
                    cassandraClient = null;
                }
            }
        }

        @ Override
        public void insertSeller(Seller seller) throws Exception{
            if (cassandraClient = = null) {
                throw new Exception("Can't connect to Cassandra.");
            }

            if (seller = = null) {
                    throw new Exception ( " Can't insert null seller to
Cassandra.");
            }
            if (seller.getUserName() = = null || seller.getUserName().isEmpty
()) {
                throw new Exception(
                    "Can't insert null sellerUserName to Cassandra.");
            }
            Map < ByteBuffer, Map < String, List < Mutation > > > mutationMap = new
HashMap < ByteBuffer, Map < String, List < Mutation > > > ();
            Map < String, List < Mutation > > cfMutationMap = new
                HashMap < String, List < Mutation > > ();
            List < Mutation > mutationList = new ArrayList < Mutation > ();

            //设置卖家信息
            if (seller.getAge() > 0) {
                Column c = new Column();
                c.name =
                    ByteBuffer.wrap("age".getBytes("utf -8"));
                c.value =
                     ByteBuffer.wrap(String.valueOf(seller. getAge()).getBytes
("utf -8"));
                c.timestamp =
                    System.currentTimeMillis();

                ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
```

```
            cosc.column = c;
            Mutation mutation = new Mutation();
            mutation.column_or_supercolumn = cosc;
            mutationList.add(mutation);
        }

        if (seller.getAddress() != null && !seller.getAddress().isEmpty()) {
            Column c = new Column();
            c.name =
                ByteBuffer.wrap("address".getBytes("utf-8"));
            c.value =
                ByteBuffer.wrap(seller.getAddress().getBytes("utf-8"));
            c.timestamp =
                System.currentTimeMillis();

            ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
            cosc.column = c;
            Mutation mutation = new Mutation();
            mutation.column_or_supercolumn = cosc;
            mutationList.add(mutation);
        }

        if (seller.getSex()!= null && !seller.getSex().isEmpty()) {
            Column c = new Column();
            c.name =
                ByteBuffer.wrap("sex".getBytes("utf-8"));
            c.value =
                ByteBuffer.wrap(seller.getSex().getBytes("utf-8"));
            c.timestamp =
                System.currentTimeMillis();

            ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
            cosc.column = c;
            Mutation mutation = new Mutation();
            mutation.column_or_supercolumn = cosc;
            mutationList.add(mutation);
        }

        if (seller.getName()!=null && !seller.getName().isEmpty()) {
            Column c = new Column();
            c.name =
                ByteBuffer.wrap("name".getBytes("utf-8"));
            c.value =
                ByteBuffer.wrap(seller.getName().getBytes("utf-8"));
            c.timestamp =
                System.currentTimeMillis();

            ColumnOrSuperColumn cosc = new ColumnOrSuperColumn();
            cosc.column = c;
            Mutation mutation = new Mutation();
```

```
                    mutation.column_or_supercolumn = cosc;
                    mutationList.add(mutation);
                }

                if (!mutationList.isEmpty()) {
                    cfMutationMap.put(Seller.ColumnFamily,mutationList);
                    mutationMap.put (ByteBuffer.wrap (seller.getUserName ()
.getBytes("utf-8")),cfMutationMap);

        cassandraClient.batch_mutate(mutationMap,ConsistencyLevel.ONE);
                }
        }

        @ Override
        public Seller getSeller(String sellerUserName) throws Exception {
                if (cassandraClient = = null) {
                        throw new Exception("Can't connect to Cassandra.");
                }

                if (sellerUserName = = null || sellerUserName.isEmpty()) {
                        throw new Exception("Can't get null buyer to Cassandra.");
                }

                ColumnParent columnParent = new ColumnParent();
                columnParent.column_family = Seller.ColumnFamily;
                SliceRange range = new SliceRange(ByteBuffer.wrap(new byte[] {}),Byte-
Buffer.wrap(new byte[] {}), true, Integer.MAX_VALUE);
                SlicePredicate slicePredicate = new SlicePredicate();
                slicePredicate.slice_range = range;

                List < ColumnOrSuperColumn > sellerColumns = cassandraClient.get_
slice(

        ByteBuffer.wrap(sellerUserName.getBytes("utf-8")),columnParent, sli-
cePredicate, ConsistencyLevel.ONE);

                Seller seller = new Seller();
                seller.setUserName(sellerUserName);
                for (ColumnOrSuperColumn cosc : sellerColumns) {
                        Column c = cosc.column;
                        String columnName = new String(c.getName(), "utf-8");
                        String columnValue = new String(c.getValue(), "utf-8");
                        if ("age".equals(columnName)) {
        seller.setAge(Integer.parseInt(columnValue));
                        } else if ("address".equals(columnName)){
                                seller.setAddress(columnValue);
                        } else if ("sex".equals(columnName)) {
                                seller.setSex(columnValue);
                        } else if ("name".equals(columnName)) {
                                seller.setName(columnValue);
```

```
                }
            }

            return seller;
        }
    }
```

C.9　cassSeller. model. Buyer

```
package cassSeller.model;


public class Buyer {
    public static final String keySpace = "CassSeller";
    public static final String ColumnFamily = "Buyer";

    private String userName;

    private String name;
    private int age;
    private String sex;
    private String address;

    public String getUserName() {
        return userName;
    }

    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
```

```
    public String getSex() {
            return sex;
    }

    public void setSex(String sex) {
            this.sex = sex;
    }

    public String getAddress() {
            return address;
    }

    public void setAddress(String address) {
            this.address = address;
    }
}
```

C. 10 cassSeller. model. Comment

```
package cassSeller.model;

import org.safehaus.uuid.UUID;

public class Comment {
public static final String keySpace = "CassSeller";
public static final String ColumnFamily = "Comment";

private UUID uuid;
private String content;
private String commentUserName;

public UUID getUuid() {
        return uuid;
}

public void setUuid(UUID uuid) {
        this.uuid = uuid;
}

public String getContent() {
        return content;
}

public void setContent(String content) {
        this.content = content;
}
```

```
    public String getCommentUserName() {
            return commentUserName;
    }

    public void setCommentUserName(String commentUserName) {
            this.commentUserName = commentUserName;
    }

}
```

C.11　cassSeller. model. Product

```
package cassSeller.model;

import org.safehaus.uuid.UUID;

public class Product {
    public static final String keySpace = "CassSeller";
    public static final String ColumnFamily = "Product";

    private UUID uuid;
    private String name;
    private String sellerUserName;
    private String desc;
    private double price;

    public UUID getUuid() {
            return uuid;
    }

    public void setUuid(UUID uuid) {
            this.uuid = uuid;
    }

    public String getName() {
            return name;
    }

    public void setName(String name) {
            this.name = name;
    }

    public String getDesc() {
            return desc;
    }

    public void setDesc(String desc) {
```

```
            this.desc = desc;
        }

        public double getPrice() {
            return price;
        }

        public void setPrice(double price) {
            this.price = price;
        }

        public String getSellerUserName() {
            return sellerUserName;
        }

        public void setSellerUserName(String sellerUserName) {
            this.sellerUserName = sellerUserName;
        }

    }
```

C. 12 cassSeller. model. Seller

```
package cassSeller.model;

public class Seller {
    public static final String keySpace = "CassSeller";
    public static final String ColumnFamily = "Seller";

    private String userName;

    private String name;
    private int age;
    private String sex;
    private String address;

    public String getUserName() {
        return userName;
    }
    public void setUserName(String userName) {
        this.userName = userName;
    }

    public String getName() {
        return name;
    }
    public void setName(String name) {
```

```
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getSex() {
        return sex;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }
}
```

Cassandra实战

Cassandra in Action

毫无疑问，NoSQL数据库应用的出现让数据库的选择性变得更加丰富。Facebook公司推出的Cassandra无疑是开源NoSQL产品中的一颗闪亮新星。Facebook、Twitter、Digg等公司对Cassandra的使用已经证明了它不是一个玩具产品，其完全可以在复杂的生产环境下承担重要的角色。同时，越来越多的关系数据库也开始关注NoSQL技术的发展，如最近MySQL数据库下的InnoDB存储引擎已经提供了NoSQL的访问方式。本书对Cassandra的配置、应用、编译等内容进行了详细的介绍，想对Cassandra一探究竟的读者千万不能错过。

——姜承尧 资深MySQL数据库专家，著有经典著作《MySQL技术内幕：InnoDB存储引擎》

Cassandra最初是由Facebook开发的一套开源的分布式NoSQL数据库系统，它同时具备了Google BigTable的数据模型和Amazon Dynamo的完全分布式架构，具有良好的可扩展性，目前被很多大型的Web 2.0网站所使用，是一种流行的分布式结构化数据存储方案。本书作者维护Cassandra数据库已经有很长时间，具有丰富的一线工作经验，同时本书还结合源码对一些底层的机制和原理进行了分析，值得初中级读者参考。

——杨海朝 新浪网（中国）技术有限公司首席DBA/新浪微博数据库负责人

NoSQL是IT领域当下讨论最热烈的技术话题之一。2010年，Cassandra的去中心化和无缝扩展的特性吸引了众多NoSQL粉丝的眼球，成为了NoSQL阵营中一道亮丽的风景线。本书既能帮助我们全面掌握Cassandra的基础知识，又能引领我们深入了解Cassandra运行机制与原理。相信在阅读完本书后，你将会对Cassandra有一个全面而深入的认识。对于喜爱NoSQL的我来说，看到国内有相关的书籍出版，是相当兴奋的，特此向所有关注NoSQL的朋友推荐此书。

——孙　立 数据库专家/NoSQL先驱/去哪儿网高级系统架构师

有很多技术在出现后不久就淡出了人们的视野，导致这种情况发生的原因有很多，其中一个重要的原因就是没有合适的应用环境。在这一点上，NoSQL技术无疑是幸运的。因为我们所熟知的关系型数据库在很多超大规模、高并发的Web 2.0网站应用中面临很多无法克服的问题，所以近几年NoSQL技术越来越受到关注和重视。Amazon的Dynamo和Google的BigTable都是非常成功的商业NoSQL产品，很多开源的NoSQL产品也得到了长足的发展，Cassandra就是其中之一。

本书由浅入深地介绍了Cassandra的基础知识和它在生产环境中的应用，无疑是一本全面了解Cassandra的好书，本书将引领你走进无比精彩的NoSQL世界。

——张　勤 51CTO专家博客（http://onlyzq.blog.51cto.com/）/著有《linux服务器配置全程实录》

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com
网上购书：www.china-pub.com

上架指导：计算机/数据库
ISBN 978-7-111-34164-2
9 787111 341642
定价：59.00元