基于key-value的NoSQL数据存储系统的设计与实现

软件工程 罗晨 指导教师 穆斌

【摘要】 （略）

【关键词】 NoSQL 数据库 分布式系统 CAP理论 键值

【Abstract】 （略）

【Keywords】 NoSQL, Database, Distributed System, CAP, Key-Value

**目录**

[1 引言 4](#_Toc356240928)

[1.1 课题背景 4](#_Toc356240929)

[1.2 研究目标 4](#_Toc356240930)

[1.3 论文结构 4](#_Toc356240931)

[2 相关概念 5](#_Toc356240932)

[2.1 数据存储系统 5](#_Toc356240933)

[2.2 传统数据库瓶颈 5](#_Toc356240934)

[2.3 NoSQL数据库 6](#_Toc356240935)

[2.4 系统扩展性 6](#_Toc356240936)

[2.4.1 垂直扩展 6](#_Toc356240937)

[2.4.2 水平扩展 7](#_Toc356240938)

[2.5 CAP理论 7](#_Toc356240939)

[3 系统编程接口 7](#_Toc356240940)

[3.1 数据模型 8](#_Toc356240941)

[3.1.1 键值结构 8](#_Toc356240942)

[3.1.2 Region结构 8](#_Toc356240943)

[3.2 键值操作 9](#_Toc356240944)

[3.2.1 set操作 9](#_Toc356240945)

[3.2.2 get操作 9](#_Toc356240946)

[3.2.3 incr操作 9](#_Toc356240947)

[3.2.4 delete操作 10](#_Toc356240948)

[3.2.5 stat操作 10](#_Toc356240949)

[4 系统依赖 10](#_Toc356240950)

[4.1 分布式文件系统 10](#_Toc356240951)

[4.2 分布式协调系统 10](#_Toc356240952)

[4.3 异步网络通信框架 11](#_Toc356240953)

[5 系统架构 12](#_Toc356240954)

[5.1 架构概览 12](#_Toc356240955)

[5.1.1 总体架构 12](#_Toc356240956)

[5.1.2 设计风格 13](#_Toc356240957)

[5.2 主服务器 14](#_Toc356240958)

[5.2.1 架构概述 14](#_Toc356240959)

[5.2.2 集群状态监听 15](#_Toc356240960)

[5.2.3 定时任务 15](#_Toc356240961)

[5.3 数据服务器 17](#_Toc356240962)

[5.3.1 内存存储引擎 17](#_Toc356240963)

[5.3.2 持久化存储引擎 19](#_Toc356240964)

[5.3.3 文件格式 20](#_Toc356240965)

[5.3.4 索引结构 20](#_Toc356240966)

[5.3.5 任务线程 21](#_Toc356240967)

[5.4 操作日志 23](#_Toc356240968)

[5.5 客户端 24](#_Toc356240969)

[5.5.1 同步模型 25](#_Toc356240970)

[5.5.2 异步模型 26](#_Toc356240971)

[5.6 通信模型 26](#_Toc356240972)

[5.6.1 模型概述 26](#_Toc356240973)

[5.6.2 消息格式 29](#_Toc356240974)

[6 功能实现 30](#_Toc356240975)

[6.1 集群启动与关闭 30](#_Toc356240976)

[6.1.1 启动顺序 30](#_Toc356240977)

[6.1.2 系统关闭 30](#_Toc356240978)

[6.1.3 领导选举 31](#_Toc356240979)

[6.2 故障恢复 32](#_Toc356240980)

[6.2.1 可靠性保证 32](#_Toc356240981)

[6.2.2 主服务器故障 33](#_Toc356240982)

[6.2.3 Data服务器故障 33](#_Toc356240983)

[6.3 响应请求 33](#_Toc356240984)

[6.3.1 set操作 33](#_Toc356240985)

[6.3.2 get操作 34](#_Toc356240986)

[6.3.3 incr操作 34](#_Toc356240987)

[6.3.4 delete操作 34](#_Toc356240988)

[6.3.5 stat操作 34](#_Toc356240989)

[6.4 负载均衡 35](#_Toc356240990)

[6.4.1 自动路由 35](#_Toc356240991)

[6.4.2 Region分配 36](#_Toc356240992)

[6.4.3 Region卸载 37](#_Toc356240993)

[6.4.4 Region拆分 37](#_Toc356240994)

[6.4.5 Region合并 39](#_Toc356240995)

[6.5 垃圾回收 40](#_Toc356240996)

[6.5.1 主服务器目录 40](#_Toc356240997)

[6.5.2 数据服务器目录 41](#_Toc356240998)

[6.6 系统配置 42](#_Toc356240999)

[6.7 系统监控 43](#_Toc356241000)

[6.7.1 监控模块 43](#_Toc356241001)

[6.7.2 使用方法 44](#_Toc356241002)

[7 性能数据 45](#_Toc356241003)

[7.1 测试环境 46](#_Toc356241004)

[7.2 测试方法 46](#_Toc356241005)

[7.3 性能结果 46](#_Toc356241006)

[7.3.1 内存存储引擎 46](#_Toc356241007)

[7.3.2 持久化存储引擎 48](#_Toc356241008)

[8 使用场景 49](#_Toc356241009)

[8.1 缓存系统 49](#_Toc356241010)

[8.2 持久化存储 51](#_Toc356241011)

[9 相关工作 52](#_Toc356241012)

[10 结论与展望 52](#_Toc356241013)

[11 主要参考文献 53](#_Toc356241014)

# 引言

## 课题背景

随着互联网的发展， web应用的用户数量以及数据规模开始呈现爆炸式的增加。对用此类新兴的互联网应用，其特点通常为高并发访问、海量数据存储以及相对弱的事务操作。在这种大数据时代的背景下，单机系统早已不能满足软件系统的需求，因此人们提出了许多扩展方案以提高系统性能。但传统的数据库由于模型复杂、提供ACID属性的事务，因而很难进行分布式扩展，并开始不能满足日益增长的大数据存储需求。而在这种背景下，NoSQL数据库应运而生。同时，NoSQL具有模型简单、高度扩展、分布式等特点，而受到开发人员的青睐。目前业界IT公司都实现了自己的NoSQL数据存储系统，并搭建了基于数以千计机器的集群，展示了NoSQL数据存储系统的强大魅力。

本课题借鉴了目前业界主流的NoSQL数据存储系统的实现方案，并在前人工作的基础上，设计并实现了名为“kvstore”的基于键值的分布式数据存储系统的原型。

## 研究目标

本课题最终应实现分布式的键值数据存储系统的原型，并应支持如下几大功能：键值操作、内存与持久两种存储方式、负载均衡、故障自动恢复、支持多语言客户端等。系统完成之后，应进行相应的功能测试和性能测试，以证明系统的正确性以及高可用性。

## 论文结构

本文第2章介绍了分布式数据存储系统所涉及的相关概念及理论，第3章介绍了系统提供的编程接口API。之后的第4、5、6章介绍了系统的实现方案，其中第4章介绍了系统所依赖的开源项目级系统，第5章大致介绍了系统的总体架构以及各组件的功能，而第6章则介绍了系统功能的实现算法与过程。之后的第7章给出了系统的性能数据，第8章则通过实例介绍了本系统的使用场景。最后第9和10章分别介绍了本论文的相关工作以及结论和展望。

# 相关概念

## 数据存储系统

数据存储系统也可被称作数据库管理系统，由一个互相关联的数据的集合和一组用以访问这些数据的程序组成[1] 。数据库技术最初产生于20世纪60年代中期，随着计算机管理数据的规模越来越大，应用越来越广泛。数据库技术也在不断地发展和提高。尤其在1970年IBM的研究员发表的“A Relational Model of Data for Large Shared Data Banks”[2] 奠定了现代关系数据库的基础。

关系数据库是创建在[关系模型](https://zh.wikipedia.org/wiki/%E5%85%B3%E7%B3%BB%E6%A8%A1%E5%9E%8B)基础上的[数据库](https://zh.wikipedia.org/wiki/%E6%95%B0%E6%8D%AE%E5%BA%93)，借助于[集合代数](https://zh.wikipedia.org/wiki/%E9%9B%86%E5%90%88%E4%BB%A3%E6%95%B0)等[数学](https://zh.wikipedia.org/wiki/%E6%95%B0%E5%AD%A6)概念和方法来处理数据库中的数据。关系数据库提供完整的关系操作（增、删、改、查等），完整性约束（实体完整性、参照完整性和用户定义完整性）以及关系数据语言SQL。关系数据库中的数据通过关系（表）来组成，结构清晰，并且功能完备，因此被广为使用。目前比较流行的关系数据库有Oracle，MySQL，微软的SQLSERVER等。

除了关系数据库之外，目前业界比较流行还有对象数据库。对象数据库是人们为了适配当前面向对象语言（如C++，Java）而尝试开发的一种数据库。面向对象语言可以直接操作数据库中的对象，而无需进行繁琐的转换。比较有代表性的对象数据库有Versant公司的Versant以及db4o数据库，主要运用在电信、金融等领域。但由于对象数据库的诞生时间较晚，以及缺少形式化的数学基础而导致的查询上的弱势，对象数据库并没有广泛的流行起来。

由于关系数据库在业界所占非常大的比重，因此本论文中主要考虑关系数据库，除了特殊的说明之外，下文中的“数据库”也代指的是关系数据库。

## 传统数据库瓶颈

由于SQL数据库对数据的存储与操作提供了强大的支持，人们也因此普遍采用关系数据库作为应用的数据存储系统。但随着互联网的发展和互联网用户的增多导致了数据集的急剧膨胀以及高并发的数据访问，例如Google、Facebook此类网站，传统的关系数据库在这种环境下开始逐渐成为应用的瓶颈之一：

#### 随着数据集的增大和高并发的访问，单机数据库难以处理大规模的数据存储与访问；

#### 关系数据库通常为了提供数据的强一致性约束和ACID属性的事务，会在并发的数据访问时对数据进行加锁，因此限制了并发能力；

#### 同样由于强一致性的保证，关系数据库难以实现分布式。虽然可以通过two-phase commit[3] 等技术来实现分布式事务，但通常会涉及集群中的所有机器，同样限制了集群的规模；

#### 关系数据库提供存储过程以实现数据操作的业务逻辑，但随着SOA架构的流行，人们逐渐开始将数据操作与数据存储解耦，即数据操作放在业务层以web服务或远程调用的方式提供，而数据库中只负责存储数据。

虽然关系数据库比较难以实现分布式，但目前仍有一些公司通过放松数据一致性的要求以及放弃关系数据库中的某些特性（如表连接查询）实现了关系数据库的分布式集群，并得到了较好的效果。以此为代表的是eBay所采用的以分库分表的方式[4] 构建数据库集群。通过对数据的主键或某些业务属性进行分组，可以将数据分配到不同的表或不同的数据库中。但需要额外的路由程序或查询解释器将数据分配到对应的表或数据库中，同时也失去了数据的强一致性保证。

## NoSQL数据库

NoSQL通常也是Not Only SQL的缩写，是对不同于传统的关系数据库的数据库管理系统的统称。NoSQL数据库与传统的关系数据库在很多方面上都有区别，但最重要的特点是NoSQL数据库不使用SQL作为查询语言，并且数据存储不采用固定的表格模式，因而也更加灵活，并具有良好的水平扩展性。

目前NoSQL数据库通常包括以下几类[5] ：

#### 键-值存储：系统根据用户定义的键来存储并索引对应的值。键-值存储系统就像一个哈希表，但是通常提供了额外的功能，例如数据副本，版本控制，加锁，事务，键排序等。键-值存储系统的代表有Redis([http://redis.io](http://redis.io/))，Amazon公司研发的Dynamo[6] 等。

#### 文档存储：系统存储定义好的文档，并提供索引和简单的检索机制。与键值存储不同的是，这类系统通常提供二级索引，多种类型的文档以及嵌套文档。文档存储系统的代表有MongoDB（<http://www.mongodb.org/>）。

#### 可扩展的记录存储：系统存储可被垂直分区以及水平分布在不同节点上的可扩展的记录。基本的数据模型通常为行和列，行通过唯一的键指定，而列中通常包含若干预定义好的组以及任意多的属性。可扩展的记录存储系统的代表有Google公司研发的BigTable[7] ，Cassandra[8] 等。

以上几种NoSQL数据库都包含一些相似的特征，例如数据的无结构或弱结构性，提供弱一致性或最终一致性[9] 的保证，易于水平扩展，很少提供事务支持等。而通常的互联网应用如社交网站、论坛、邮箱等，业务逻辑较为简单，对数据的一致性要求不是太高，并且通常具有大数据、高并发等特性，因此比较适合采用NoSQL数据库作为数据存储系统。而与此同时，NoSQL数据库也在这些大型的互联网公司的推动以及大数据的背景下不断发展，与关系数据库进行更好的互补。

## 系统扩展性

随着大数据时代的来临，传统的单机数据库模式早已不能适应日益增长的数据规模和高并发的访问。对于数据库系统的扩展方式，通常有如下两种不同的方案。

### 垂直扩展

垂直扩展即对一个计算实体增加资源而提高计算实体的性能。例如使用更多核的CPU，添加更多的内存和硬盘等。垂直扩展的优点是简单方便，而且软件系统无需对代码进行修改。目前市场上许多服务器厂商如Dell、IBM、联想等都提供有超高性能的服务器产品，另外SAP的HANA数据库可支持TB级的内存计算。然而垂直扩展的缺点也是显而易见，首先超高性能服务器的价格远远高于普通机器，其次服务器很难无限制的扩展，容易达到瓶颈，并且性能越高，进一步扩展的开销也就越大。因此垂直扩展的架构方案并没有被广泛的采用。

### 水平扩展

水平扩展也被称作横向扩展。与垂直扩展相反，水平扩展将多台计算实体通过特定的软硬件连接成为一个逻辑的实体以提供更强大的计算能力。水平扩展通常采用性能普通的服务器甚至PC机，通过特定的分布式算法将任务分配到不同的机器上，以获得强大的计算能力。例如Google的BigTable[7] 存储系统就已经实现了可运行在上千台机器组成的集群中。水平扩展方案价格低廉，使用普通的机器便可获得强大的计算能力，而且集群的总体性能也不受单台机器性能的限制，因而也受到许多IT公司的青睐。

然而水平扩展在实施上通常也比垂直扩展更加复杂：

#### 数据存储系统的水平扩展需要软件层面的支持。例如eBay公司的分库分表方案[4] ，需要专门的路由程序的支持，以将数据根据主键或特定的业务属性分配到对应的机器中

#### 集群具有较高的维护难度和成本。随着机器数量的增加，集群的维护成本也相应的增加，并且需要自动化工具的支持以实现日常的维护工作。

#### 分布式集群为系统提出了额外的问题和复杂性，例如网络分区问题，节点失效问题，节点间协调问题等等，都需要软件系统予以额外的考虑。

垂直扩展与水平扩展具有各自的优缺点，二者适应的场景也有所不同。垂直扩展的简单方便比较受非IT企业的青睐，如电信、金融、能源等领域的公司，并且垂直扩展带来的性能也可基本满足其业务需求。而水平扩展由于比较复杂，目前只有较大的IT公司如Google、Facebook、Amazon等才有能力自主研发数据存储系统，并搭建大规模的集群。本论文中所实现的系统基于横向扩展，将多服务器搭建成统一的集群，以提供数据存储服务。

## CAP理论

2000年，加州大学伯克利分校的Eric Brewer教授在2000年的分布式原则研讨会上提出了著名的CAP理论，即对于一个分布式系统来说，不可能同时满足以下三点：

#### 一致性（Consistency），即所有节点在同一时间有相同的数据；

#### 可用性（Availability），保证每个请求不管成功或者失败都有响应；

#### 分区容忍性（Partition Tolerance），系统中任意信息的丢失或失败不会影响系统的继续运作。[11]

而CAP理论的正确性也在2002年麻省理工学院的Seth Gilbert等人的“Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services”[12] 论文中被证明。

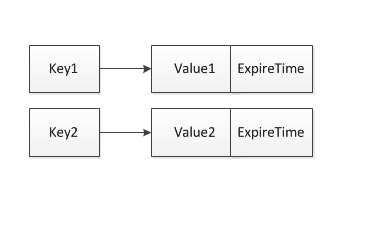
由于分区容忍性是分布式系统的基础，因此人们通常会在一致性和可用性之间进行取舍，或对二者进行平衡。具有一致性与分区容忍性（CP）的代表系统有Google公司的BigTable[13] [7] 系统，其通过Google File System[13] 实现了较高的数据一致性。而Amazon的Dynamo[6] 系统则通过版本号机制和数据多备份等实现了高可用性，然而应用程序则需要解决有可能读取不到最新数据的问题。而另一类系统如Cassandra（<http://cassandra.apache.org/>）则允许用户配置成功写操作的最小节点数（W）、成功读操作的最小节点数（R）和副本节点的数量（N）来调整系统的一致性和可用性。当W+R越大，则一致性越高，可用性越低，反之亦然。

# 系统编程接口

## 数据模型

### 键值结构

kvstore采用键值模型作为数据模型结构，类似于一个大的哈希表，其中每一个键都惟一确定了一个值，并且系统不允许有重复的键以及空值出现。另外键和值在系统中都存储为二进制字节数组，而由用户来负责解释键值的含义，例如可以是字符串、序列化的对象或者其他任意的结构。

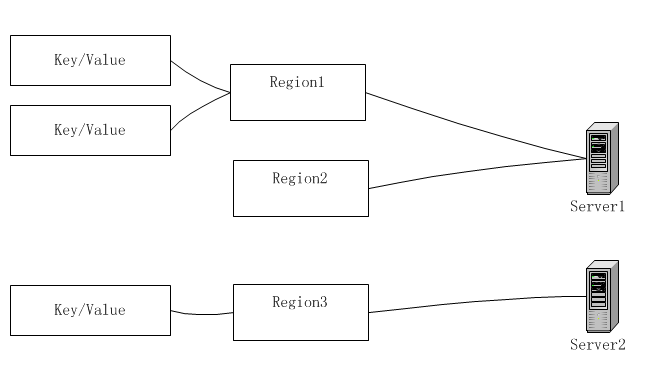


图表 3‑1

如图表 3‑1所示，每个键分别惟一对应了一个值和失效时间。失效时间为long型的时间戳，如果服务器的当前时间超过了失效时间，那么该键值对将会被自动删除，即用户再也无法通过键取得对应值。

### Region结构

Region是从某个开始键到结束键的所有键值对的集合，也是系统负载均衡的基本单位。如图表 3‑2所示，Region会被分配到某个唯一的服务器中，之后所有对该区域内的键值对的请求都会由该服务器所负责。



图表 3‑2

Region中包含如下的属性：开始键，结束键，以及由主服务器分配的唯一的Id。另外每个Region还对应了统计信息，统计的数据项包括键值对数量、大小、读次数和写次数。这些统计信息会作为负载均衡的依据供主服务器使用。 键顺序由字节序决定，即首先按照数组的长度，长度短者小；如果长度相等，这依次按无符号字节的格式比较数组的每一位，首先出现小字节的字节数组小。

## 键值操作

kvstore作为一个键值存储系统，支持如下几种常见的键值操作。下文中如未进行特殊说明，将按照operation param1 [param2=value]…的格式进行说明各个操作语义，其中operation是操作名称，param1是必须参数，param2是可选参数，并且默认值为value。

### set操作

Set操作的语义为设置某个键值对，即将一个键值对存储到系统当中。Set操作的指令为：set key value [ttl=0]。其中key、value都为二进制数组，ttl为int整数，指明该键值对应存活的毫秒值，默认为0，即永久保存。

### get操作

Get操作的语义为获取某个键对应的值以及存活时间的毫秒值，并且如果键不存在，则返回空。Get操作的指令为：get key。其中key为二进制数组。

### incr操作

Incr是increment的缩写，即增加某个键对应的整数计数器的值。Incr操作的指令为incr key [incremental = 1] [initValue=0] [ttl=0]。其中key为二进制数组，incremental为要为计数器增加的整数值，默认为1；initValue为计数器的初始值，默认为0；ttl为计数器存活的时间毫秒值，默认为0，即永久保存。另外计数器实际上在系统中对应了一个长度为4的二进制数组，如果尝试为一个值长度不为4的键执行incr操作，那么系统将返回错误信息。

### delete操作

Delete操作是在系统中删除某个键指定的键值对。Delete操作的指令为delete key。其中key为二进制数组。

### stat操作

Stat是statistics的简写，系统将返回包含数据服务器信息的数组，其中每个数据服务器信息对象包含了当前服务器的地址，权值，使用情况，以及其中所包含的Region情况。服务器的权值为用户指定的整数值，会在主服务器进行负载均衡时使用。服务器的使用情况包括服务器总的内存、当前剩余的内存以及CPU的使用率。由于kvstore基于分布式的文件系统，因此这里并未统计真实文件系统的使用情况。用户可以通过HDFS的管理页面进行查看。

# 系统依赖

Kvstore作为分布式数据存储系统，需要数据的持久化存储、副本、分布式协调、网络通信等基础功能。以下系统或框架作为kvstore的基础设施，为kvstore的实现提供了基础。

## 分布式文件系统

分布式文件系统是建立在网络之上的文件系统，即通过软件的方式，将多台服务器的文件系统集中成统一的文件系统，为用户提供文件的存储与访问服务，具有高度的透明性。

本系统采用HDFS（[Hadoop Distributed File System](http://hadoop.apache.org/docs/stable/hdfs_user_guide.html)）作为文件系统，以存储操作日志文件和数据文件。HDFS是Hadoop项目的一部分，也是Google File System[10] 的开源Java版本实现。HDFS具有如下几大特性：

#### 适于利用普通服务器搭建分布式存储集群，并且具有容错性和易于扩展性。

#### 具有高度配置性，可通过配置系统参数如文件块大小、副本数量等适应不同的需求。

#### 由Java编写，可支持所有主流平台。

#### 内置了web服务器，可以方便检查集群的状态。

#### 支持文件的负载均衡，尽可能的将文件均匀分布到每个节点上。

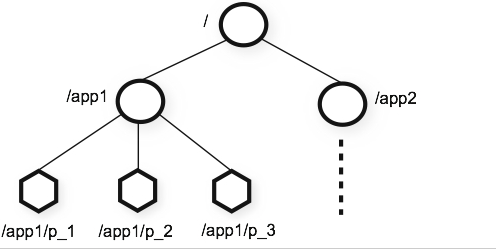
#### 提供类似Java IO的流式IO接口，简单易用。

由于HDFS通过文件的副本机制提供了数据的容错性，因此本系统中将通过在HDFS中存储操作日志和数据文件，即利用HDFS自身提供的容错性机制，来保障数据的可靠性。

另外由于Apache官方提供的发行版本中并未提供HDFS单独包，而是作为Hadoop系统的一部分进行发行。因此为减少本系统的依赖，作者基于Hadoop r1.0.4版本的源码，将其中的HDFS部分单独打包以供本系统使用。

## 分布式协调系统

本系统选用ZooKeeper作为分布式协调系统。ZooKeeper(<http://zookeeper.apache.org/>)是Hadoop的子项目，也是Google Chubby[14] 服务的Java开源实现。作为分布式协调系统，ZooKeeper提供了配置信息管理、命名服务、分布式同步、分组等服务。



图表 4‑1

如图表 4‑1所示，ZooKeeper中采用了类似文件系统的树状结构，用户可以在某个路径上创建节点或者读写节点上附加的数据，也可以以回调的方式监听某个节点。ZooKeeper中的节点分为暂时性和永久性，暂时性节点当客户端失去连接时，会被ZooKeeper删除，而永久性节点则会被永久保存。此外ZooKeeper为用户的更新操作提供了如下保证：

#### 顺序一致性，即客户端的更新操作会按照发送的顺序执行。

#### 原子性，更新操作要么成功要么失败。

#### 单系统镜像，客户端不管连接到哪台ZooKeeper服务器，看到的都是统一的视图。

#### 可靠性，一旦更新生效，它将一直持续到某个客户端覆盖了此次更新。

#### 时效性，客户端可以在固定的时间延迟内，看到系统的最新状态。

通过ZooKeeper提供的操作以及以上几点保证，用户可以方便的实现各种分布式协调算法，如通过监控暂时性节点可监控集群中节点的状态，通过顺序创建多个暂时性节点来实现Paxos领导选举算法[15] ，以及实现分布式队列、分布式锁服务等等。

## 异步网络通信框架

作为分布式的数据存储系统，网络通信是系统中必不可少也是影响性能的一部分。kvstore系统采用Mina作为网络通信框架。Mina([http://mina.apache.org](http://mina.apache.org/))是一款开源、基于Java NIO技术的网络通信框架，它具有异步、高性能、模型简单、易于使用等诸多优点：

#### 基于异步的事件回调模型，具有高并发性。

#### Mina支持多种传输协议，如TCP/IP、UDP/IP、RS232等，封装了底层实现并提供统一的编程接口。

#### 提供类似于Java Servlet Filter的过滤器，过滤器可作为扩展点，方便开发人员对消息进行预处理以及后处理。

#### 提供底层及高级的API，开发人员既可以直接操作二进制的消息流，也可以提供消息的编码、解码器以面向对象的方式操作消息。

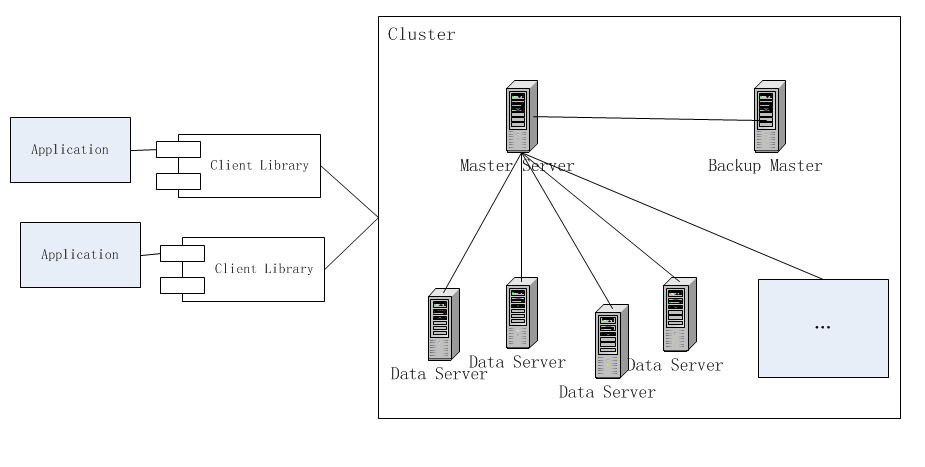
#### 高度订制的线程模型，包括单线程、单线程池和多线程池。

系统中的主服务器、数据服务以及Java客户端之间的网络通信完全基于Mina框架以及TCP/IP协议。具体的消息格式将在5.6通信进行详细介绍。另外系统提供了基于C#的客户端程序，该客户端则直接采用了Socket直接连接到主服务器和数据服务器。

# 系统架构

## 架构概览

### 总体架构



图表 5‑1

图表 5‑1为系统的总体架构图。系统集群采用Master-Slave模式，并提供Backup Master以避免单点失效问题。另外应用程序通过系统提供的相应语言版本的客户端库程序访问集群，以使用系统提供的数据存储服务。

#### 主服务器（Master Server）

主服务器是系统中的“管理者”，负责监听集群状态（例如数据服务器的加入与移除），定期收集各个数据服务器的状态信息，并定期的对集群数据服务器的负载进行负载均衡操作。此外，主服务还会响应客户端的Region表请求，以返回目前集群中的Region表。集群只能同时允许一个主服务器存在。主服务器将在章节5.2主服务器中详细介绍。

#### 备份主服务器

为了避免主服务器的单点失效问题，集群中通常还会有备份主服务器。备份主服务器在功能上与主服务器完全相同，但是所有的主服务器在启动时都会参与到领导选举算法中，被选作领导的服务器会成为主服务器，而其他的服务器都会成为备份主服务器，等待下一次领导选举算法的执行。

#### 数据服务器（Data Server）

数据服务器负责存储系统中的键值对数据。每个数据服务器会被主服务器分配若干个Region，数据服务器会负责这些Region范围内的键值对的操作。数据服务器提供内存和持久化两种存储引擎，内存存储引擎完全使用内存存储，但在内存容量满之后会置换出某些键值对，因而适合用作缓存系统；而持久化存储引擎则使用内存加文件的方式保证数据的持久性存储。数据服务器将在章节5.3数据服务器中详细介绍。

#### 客户端库程序（Client Library）

客户端库程序是应用程序以访问本系统数据存储服务的桥梁。客户端库程序主要封装了键路由、网络连接管理等工作。开发人员只需指定主服务器的地址，便可通过get、set等操作访问本系统，而无需繁琐的初始化、连接建立与释放等工作。理论上系统支持任意语言的客户端访问，只要客户端按照网络消息协议的规范与主服务器以及数据服务器交互即可。目前系统提供了Java和C#两个版本的客户端库程序。客户端库程序将在章节5.4操作日志中详细介绍。

#### 应用程序（Application）

应用程序是系统的最终用户，通过系统提供的客户端库程序访问集群以使用系统的数据存储服务。目前Java和C#的应用程序可直接使用系统提供的客户端库，而其他语言的应用则需要按照系统的网络消息协议，开发对应的客户端库程序。应用程序可自己解释键值的含义，例如序列化的对象、文档等，同时也可根据不同的需求定制系统的功能，如用作缓存系统、持久存储系统等。

### 设计风格

kvstore作为系统级应用和框架级应用，应具有较高的性能、较好的可扩展性、可维护性以及提供简单易用的API。为达到这些目标，本系统的设计与编码遵循如下几点原则和风格：

#### KISS原则

KISS是Keep it simple stupid的缩写，即应尽可能的将系统设计的简单易用。kvstore系统实现过程中尽可能的将复杂的操作拆分并简化，通过简单来提高性能，提高可靠性。同时系统提供的接口也应保持简洁，方便开发人员学习和使用。

#### 面向接口编程

面向接口编程即系统的模块与模块之间通过接口进行依赖，而不是直接依赖于实现类。面向接口编程有益于系统中模块间的解耦，增加系统的灵活性和可扩展性。在kvstore系统中，所有可能发生变化的实现都被抽象为接口，并通过接口进行调用。因而当需要更换某个接口的实现类时，可以尽可能的避免其他模块的修改。

#### 事件驱动

为提高扩展性，系统在许多操作中都提供了事件点，并允许调用者注册监听器进行回调。通过事件驱动机制，可以灵活的向已有实现中添加功能。例如，系统在每个get、set等操作后都提供了事件点，可以以此添加日志监听器、统计监听器等功能。

#### 使用设计模式

设计模式是前人对已有设计问题的总结，也是最佳实践。通过使用设计模式，可以帮助实现高内聚低耦合的目标，提高系统的灵活性、可扩展性和可维护性；同时由于大多数开发人员对设计模式都有所了解，因此通过遵循已有的设计模式，也可让开发人员更快的使用及开发本系统。例如系统中广泛用到了工厂模式、代理模式、策略模式、装饰模式、迭代器模式等。

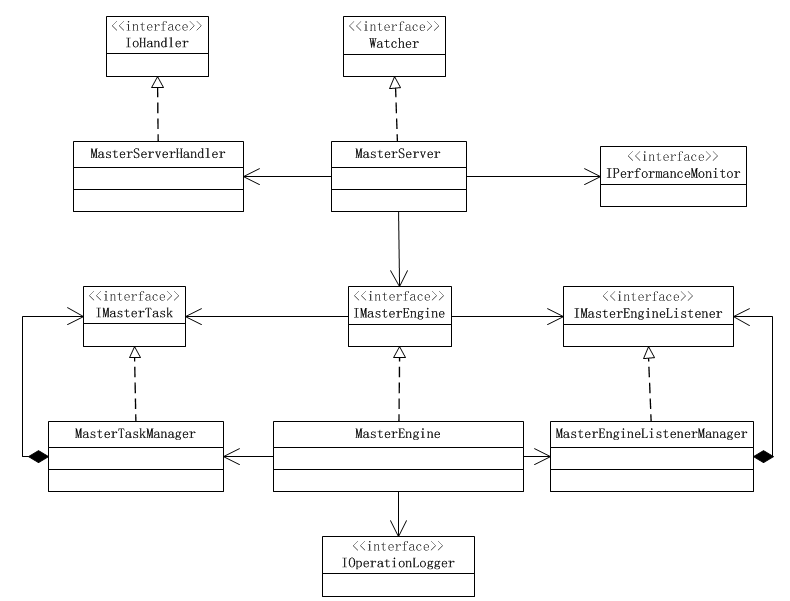
#### 可配置性

为保证灵活性，系统将各种参数都作为可配置参数。用户可在配置文件以键值对的形式指定参数名和值。可配置性允许系统满足不同的应用需求，例如作为缓存系统或持久性存储系统，作为单机系统或分布式系统。同时可配置性也允许用户根据不同的环境对系统进行不同的优化，例如对不同配置的机器设置大小不同的缓存，以逐渐在系统的运维中找到最优的配置方案。

## 主服务器

主服务器（Master Server）在系统扮演管理员的角色，主要负责监听集群状态、文件系统垃圾回收、Region负载均衡等功能。集群中允许拥有多个主服务器，但是在某个时间段只有一个活动的主服务器存在。

### 架构概述



图表 5‑2

图表 5‑2为主服务器中的主要类图，其中IoHandler为Mina框架提供的接口，当某些网络连接相关的事件（例如接收消息、会话失效等）发生时的回调接口。Watch为ZooKeeper系统提供的接口，为监控的节点发生变化时的回调接口。主服务器通过实现Watcher接口来实现主服务器的领导选举算法。

#### MasterServer

MasterServer类是主服务器的入口点，同时负责主服务器中其他组件的初始化与资源释放工作。同时，MasterServer类也负责响应网络连接的建立与释放，消息接收，以及监听ZooKeeper中节点的创建与删除动作（创建表示机器加入，删除表示机器退出）。

#### IMasterEngine

IMasterEngine封装了主服务器的主要逻辑，例如添加删除服务器、Region分配、同步数据服务器信息等。IMasterEngine中的方法通常由MasterServer在某个特定的事件发生时所调用，例如收到数据服务器发送的请求或响应时。

#### IMasterEngineListener

IMasterEngineListener为IMasterEngine提供的事件监听器，IMasterEngine在许多操作中都提供了事件点，例如数据服务器卸载，Region拆分等，允许IMasterEngine的调用者动态向其中添加功能。

#### IMasterTask

IMasterTask定义了主服务器定时任务的接口，这些任务以单独线程的方式，并且以一个固定的时间间隔运行。由IMasterEngine负责任务的启动与关闭。主服务器定时任务将在5.2.3定时任务中详细介绍。

另外主服务器中涉及网络通信的部分，将在5.6通信中详细介绍。

### 集群状态监听

主服务器会负责对集群的状态监听。其中集群的状态包括主服务器的状态和数据服务器的状态。

#### 主服务器状态监听

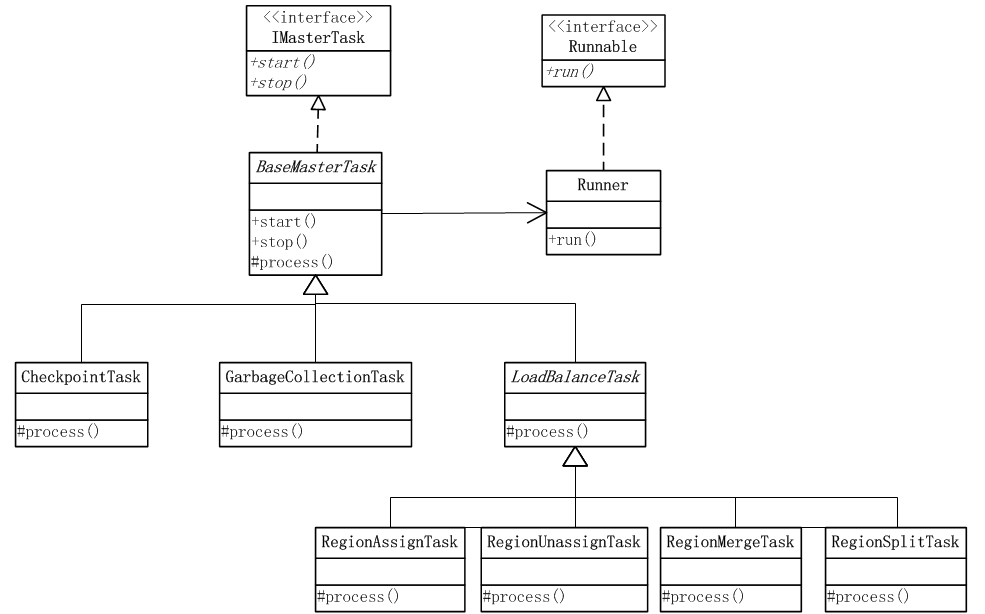
主服务器状态的监听通过ZooKeeper实现。当主服务器启动时，会向ZooKeeper中创建一个暂时性的顺序节点。其他主服务器将依次监视对方。当某个节点被ZooKeeper删除时，通常是由于主服务器发生故障或网络连通故障，相应的主服务器会被ZooKeeper触发，并开始新的领导选举的过程。同时新的领导也会向ZooKeeper的某节点写入当前主服务器的地址，以通知数据服务器连接。主服务器领导选举的部分将在6.1.3领导选举中详细介绍。

#### 数据服务器状态监听

数据服务器状态的监听通过Mina框架的会话功能实现。当数据服务器启动时，它会尝试与主服务器建立连接，并通过定时的心跳消息汇报自身的情况。与此同时，主服务器会保存每一个数据服务器的状态与会话信息。当数据服务器或网络连接发生故障时，数据服务器与主服务器之间的会话会发生超时，并且主服务器的MasterServerHandler会收到相应的通知。

### 定时任务

在主服务器中，每个定时任务都是一个单独的线程，并且定时重复执行某项操作。



图表 5‑3

图表 5‑3为主服务器中定时任务的类图。任务的设计采用了模板方法模式，即每个BaseMasterTask中定义了抽象的process方法，并且该方法在线程Runner中被循环调用，而每个实体Task都实现了具体的process方法。主服务器中每个任务的介绍如下：

#### 检查点任务（CheckpointTask）

检查点任务定时的将主服务器的状态保存到新的检查点文件中，并切换新的操作日志，以控制操作日志的大小。检查点文件需要保存的状态信息主要包括每个Region的信息，并且主服务器在启动时会加载最新的检查点以及相应的操作日志文件。检查点任务的执行频率通常以天为单位，并且用户可指定检查点文件的保留时间。

#### 垃圾回收任务（GarbageCollectionTask）

垃圾回收任务定时的检查HDFS中的垃圾文件以及过期的数据文件，并将其删除。由于系统在运行过程中的失败操作可能产生垃圾文件，因此通过垃圾回收任务，可以定期的检查并删除这些垃圾文件，释放存储空间。垃圾回收任务将在6.5垃圾回收小结中详细介绍。

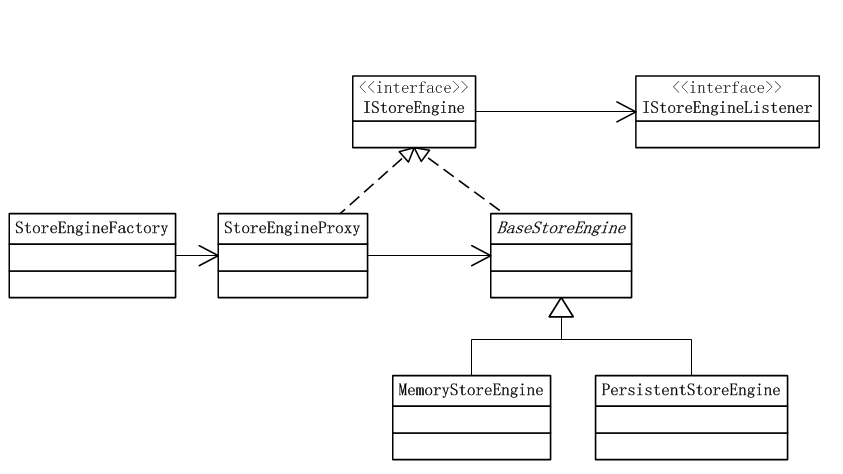
#### 负载均衡任务（LoadBalanceTask）

负载均衡任务主要负责检查每个数据服务器中的Region情况，包括每个Region的大小以及当前数据服务器的负载情况。问了简化问题，系统将负载均衡任务分为四个独立的子任务，分别为Region分配任务（RegionAssignTask）、Region卸载任务（RegionUnassignTask）、Region拆分任务（RegionSplitTask）和Region合并任务（RegionMergeTask）。四个子任务独立执行，但是系统保证同一个Region在同一个时间内只会被某一个子任务处理。例如如果Region A在某一时刻被Region分配任务选为目标，那么在得到目标数据服务器答复之前，其他子任务不会再处理Region A。

负载均衡任务的时间间隔以秒为单位，通常Region分配任务的间隔应设定的比较短，以确保Region被卸载后可以及时被分配，而其他子任务的时间间隔可比较长，以减少主服务器的资源利用。负载均衡任务将在章节6.4负载均衡中详细介绍。

## 数据服务器

数据服务器是系统负责中存储键值数据的服务器。与主服务器不同，通常系统中有多个数据服务器存在。另外为适应不同的应用需求，数据服务器提供了两种存储引擎，即内存存储引擎和持久化存储引擎。

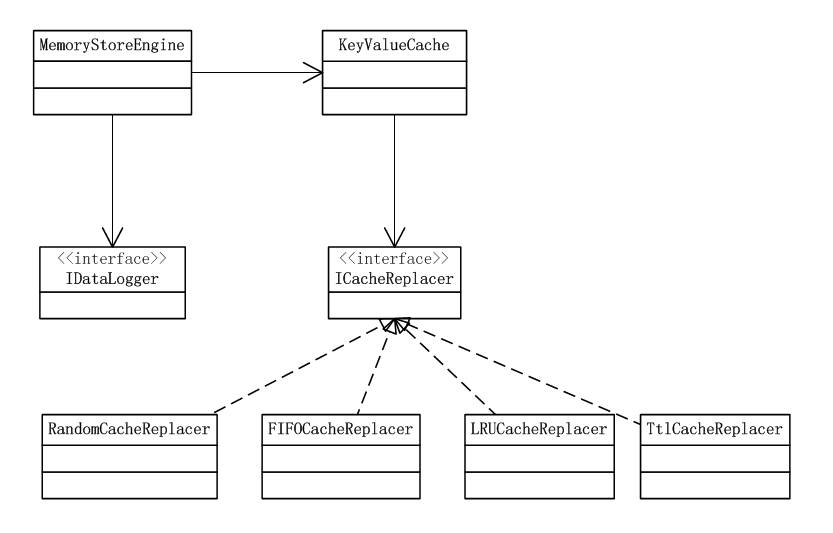


图表 5‑4

图表 5‑4描述了存储引擎的基本结构。其中系统采用代理模式在某些函数的调用后回调存储引擎的监听器。同时采用工厂模式，根据调用者指定的存储引擎类型，创建并返回相应的存储引擎。

### 内存存储引擎

内存存储引擎将所有的数据都保存在内存中，同时为保证数据的可恢复性，内存存储引擎在每次写操作或删除操作时都会将操作写入操作日志中。但当内存到达上限之后，系统会根据用户指定的置换算法删除某些键值对以释放空间。



图表 5‑5

图表 5‑5为内存存储引擎的主要类图。内存存储引擎的实现较为简单，其中主要用到了KeyValueCache，以及每个Region对应了一个操作日志。IDataLogger定义了操作日志的接口，具体的结构将在章节5.4操作日志中详细介绍。KeyValueCache是责管理内存中的键值对，并可以设置内存上限以及置换算法。目前系统中提供的置换算法包括：

#### 随机置换算法（RandomCacheReplacer）

随机置换算法将随机选择一个键值对进行置换。其内部采用了哈希集合的实现方式。索引键添加、置换和删除的复杂度都为O(1)。

#### 先进先出置换算法（FIFOCacheReplacer）

先进先出置换算法按照键加入的顺序进行置换，即先进入的键将先被删除。其内部采用了队列的实现方式。索引键添加与置换的复杂度为O(1)，而删除需要顺序查找，因此复杂度为O(N)。

#### 最近最少使用置换算法

最近最少使用置换算法优先选择最近最少使用的键进行置换，即对键按照最近访问的时间顺序排序。其内部采用LinkedHashMap实现。索引键的添加、删除与置换的复杂度都为O(1)。

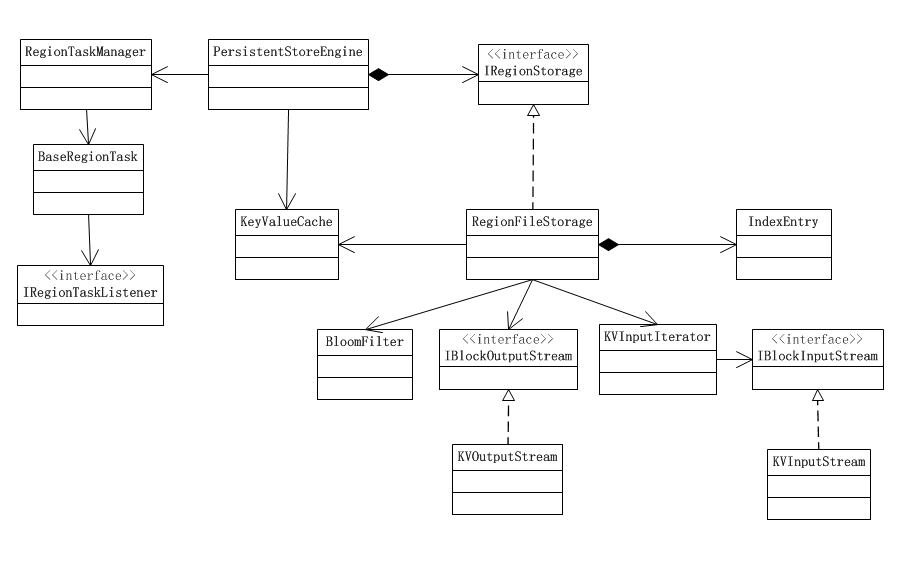
#### 键值生存时间置换算法

由于前三种置换算法没有考虑到键值的生存时间，因此有可能将不会过期的键先于将要过期的键置换出内存。键值生存时间置换算法则将键按照存活时间排序，每次选择最先过期的键进行置换。其内部采用红黑树有序集合实现。索引键添加、删除的复杂度都为O()，而置换的复杂度则为O(1)。

内存存储引擎中的响应客户端请求操作将在章节6.3响应请求中介绍，而Region的加载、卸载、拆分以及合并等操作将在章节6.4负载均衡中详细介绍。

### 持久化存储引擎

持久存储引擎通过将键值对数据写入到文件中以达到持久化存储的目的。同时为了兼顾性能，键值对会被写入内存以及操作日志中，当内存缓冲区满时，缓冲区中的数据将被合并到数据文件中。



图表 5‑6

图表 5‑6介绍了持久化存储引擎的主要类图。其中实现了IRegionStorage接口的RegionFileStorage负责某个Region的存储与读取。同时为减少文件系统的IO操作，写操作也会被直接写入到内存缓存当中。

#### PersistentStoreEngine

PersistentStoreEngine是存储引擎的持久化实现。在持久化存储引擎中，每个Region都有一个对应的IRegionStorage来实现的Region中数据的持久化存储。而持久化存储引擎则负责调用IRegionStorage中的方法，来响应客户端及主服务器的请求。

#### IRegionStorage

IRegionStorage定义了Region持久化存储的接口。同时系统提供了基于分布式文件系统HDFS的RegionFileStorage实现。IRegionStorage提供的方法包括将键值写入缓冲、读取键值数据、获得数据文件路径以及提交内存缓冲区等，以供PersistentStoreEngine调用。

#### BloomFilter

BloomFilter是1970年由布隆提出的高效的一种用于检查元素是否在集合中的算法[16] 。它实际上由一个长二进制串和一系列的哈希算法组成，并且具有很高的空间效率和时间效率，但是一定几率把不存在的元素判定为存在。持久化存储引擎通过使用BloomFilter以减少文件系统的IO操作，即如果Bl oomFilter判定某个键肯定不在集合中时，便无需执行IO操作。

#### IO组件

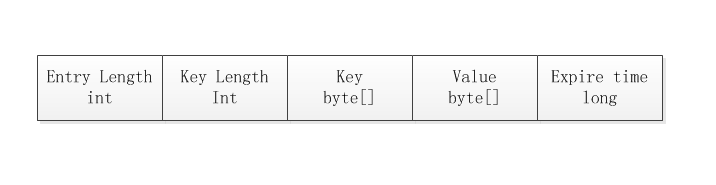
系统提供了适合键值文件结构的文件输入流和输出流以满足读写文件的需求，同时采用迭代器的模式对文件读取进行封装，使得读取文件可以遍历集合一样操作。关于键值数据文件的具体格式将在章节5.3.3文件中介绍。

#### RegionTask

由于持久化存储引擎Region相关的操作（例如Region拆分、合并、加载、卸载等）比较复杂，并且持续时间较长，因此为了兼顾性能并且提高系统并发性，系统采用异步任务的方式执行Region操作。Region任务将在章节5.3.5任务线程中详细介绍。

### 文件格式

在持久化存储引擎中，Region数据文件采用二进制流的方式进行存储，同时文件分成大小固定的块（通常为64K）以方便对文件进行索引。



图表 5‑7

图表 5‑7描述了每个键值项在数据文件中存储的格式。其中值的长度可以根据项的长度与键的长度推算出来。同时，由于文件块的存在，因此有可能出现某个键值项在结束之前遇到了块的末尾，甚至跨越多个文件块的情况。在这种情况下，如果块中剩余的字节数不足以容纳单独的int或long，那么文件中将补0，同时将数据写入下一个块中。另外数据文件末尾块没有填满，那么末尾块空白的部分也将全部补0。

系统定义了IBlockInputStream和IBlockOutputStream接口以进行数据文件的读写操作，同时提供了KVInputStream和KVOutputStream默认实现。

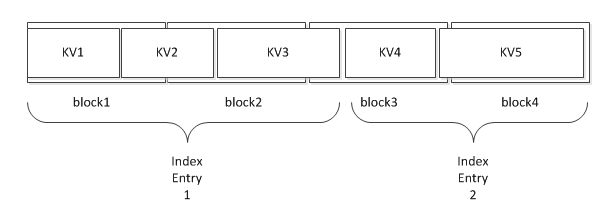
在KVInputStream中，读取操作基于块来进行，即每次KVInputStream将文件中的一个块读入内存中，然后基于内存流进行读取。同时，KVInputStream允许在一个内存块之内进行随机读取。

同样在KVOutputStream中，会保存当前块的编号以及偏移量，并且在写入int以及long等数据时，会检查如果当前如果已经到达块末尾，那么将进行补0。同时写入数据后，会更新当前块号以及偏移量。

KVInputIterator通过迭代器模式对文件读取进行封装。迭代器首先会尝试读取4个字节，即下一个项的长度。如果长度大于0，说明文件中仍有数据项存在，此时迭代器会将流的当前偏移前移四位，并允许调用者读取下一个数据项；否则说明文件中已无数据项存在，迭代器将关闭当前数据文件。

### 索引结构

Region数据文件中的键值项按照键的升序排列。因此为了提高键值项文件查找的效率，持久化存储引擎在加载某个Region时，会首先遍历Region的数据文件并建立索引。每个索引项中包含开始键、结束键、开始块编号、结束块编号，以及开始块偏移。



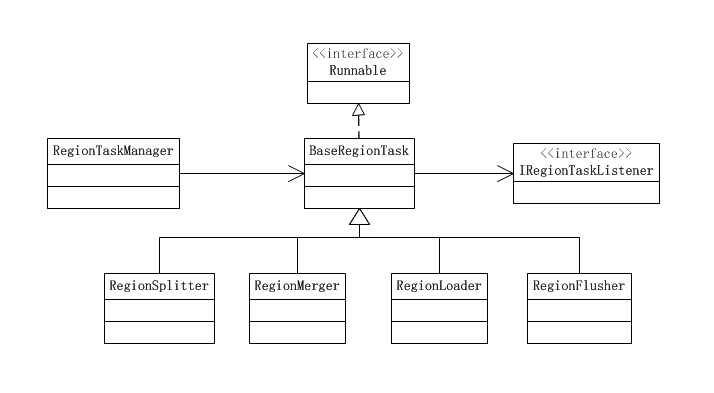
图表 5‑8

系统允许用户指定每个索引项中涵盖的块数目，但是由于可能有某个键值项跨越多个块的情况存在，系统会保证每个索引项的块数目为不小于用户指定值的并且可行的最小值。例如图表 5‑8描述了四个数据块、五个键值项以及每个索引号中涵盖的块数目为2的情况。其中索引项1为：开始键key1，结束键key3，开始块block1，结束块block3，开始块偏移0；索引项2为：开始键key4，结束键key5，开始块block3，结束块block4，开始块偏移为KV4在block3中的开始位置。

当系统从文件中读取某个键值项时，系统会首先查找到对应的索引项，然后将索引项中涵盖的数据块全部加载入内存。因此用户通过调整数据文件块大小以及索引项中的块数目，可以对系统的查找性能进行优化。一个典型的配置为块大小64KB，而索引项中的块数目为5，以平衡索引项的数目以及每次从文件中载入的数据大小。

### 任务线程

在持久化存储引擎中，Region的操作例如合并、拆分以及提交等较为复杂，并且涉及到文件IO处理，因此为了避免此类操作阻塞持久化存储引擎的其他操作，系统采用多线程任务的方式实现此类操作。另外，由于这类操作执行中需要对内存进行加写锁，并会阻止其他线程读写内存，因此为减少对其他线程的阻塞，系统规定某一个时间只允许有一个任务线程运行。



图表 5‑9

图表 5‑9描述了持久化存储引擎中用到的任务线程。其中每个线程都实现了Java的Runnable接口，因此都可作为单独的线程启动。

#### RegionSplitter

RegionSplitter主要负责将某个容量过大的Region拆分成两个。在拆分过程中，RegionSplitter会尽量将Region按照当前大小平均分成两个，并且以监听器回调的方式通过存储引擎拆分的结果。

#### RegionMerger

RegionMerger主要负责将两个容量都过小的Region合并为一个新的Region，两个Region在键范围上必须连续。

#### RegionLoader

RegionLoader主要负责加载某个给定的Region。RegionLoader需要遍历数据文件以建立索引，同时将操作日志中的数据载入缓存。另外由于RegionLoader不会阻塞数据服务器请求线程，因此通常以同步的方式调用。

#### RegionFlusher

RegionFlusher主要负责当某个Region写缓冲容量满之后，将写缓冲的数据写入数据文件。RegionFlusher会保证结果文件中的键值数据以键升序排列，同样为了保证可靠性，旧的数据文件不会被删除，而是等主服务器的垃圾回收任务进行回收。

#### RegionTaskManager

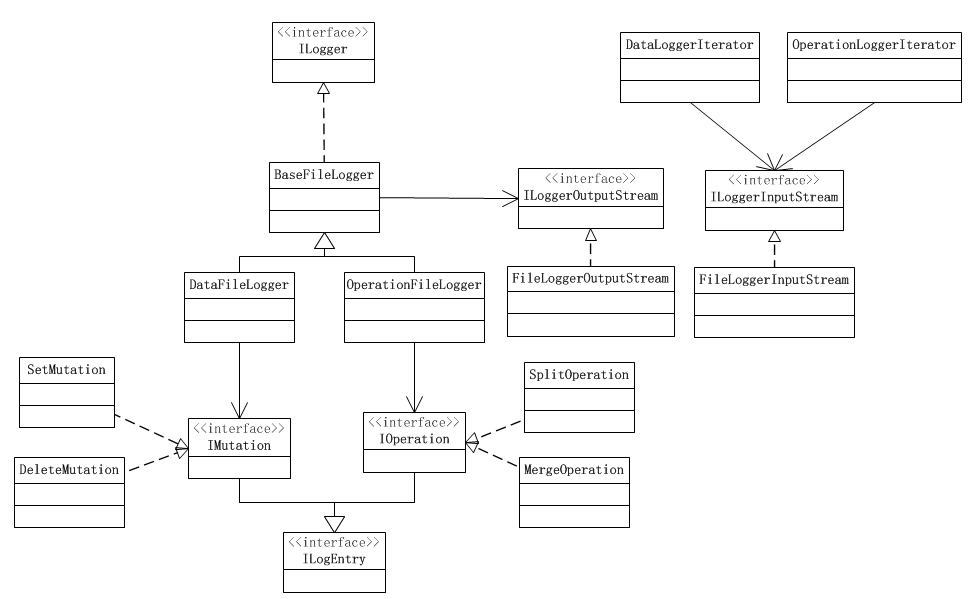
#### RegionTaskManager负责控制任务线程的启动，并且通过互斥信号量以确保同一时间内只有一个任务线程执行，线程执行完之后会将信号量修改为闲置状态。当尝试启动任务线程时，RegionTaskManager会检查互斥信号量，并返回是否启动成功，如果失败，则只需重试调用即可。

操作线程中具体的算法将在章节6.4负载均衡中详细介绍。

## 操作日志

在主服务器与数据服务器中，都采用了预写式日志技术（Write ahead log）来提供操作的持久性与高效性。对于主服务器，Region操作在提交之前会被写入操作日志中，以保证主服务器下次启动时Region信息不会丢失；而数据服务器在执行键值对的更新及删除操作提交之前也会写入操作日志，以保证变更操作的持久性。

由于操作日志都是以向末尾添加的方式写入数据，无需磁盘寻道等操作，因而效率很高，甚至可以与内存写入操作不相上下。



图表 5‑10

图表 5‑10描述了操作日志的主要类图。其中对于主服务器和数据服务器，系统分别提供了OperationFileLogger和DataFileLogger供二者使用。

#### 日志文件输入/输出流

ILoggerInputStream和ILoggerOutputStream定义了日志文件输入输出流接口，并且系统提供了基于HDFS的FileLoggerInputStream和FileLoggerOutputStream默认实现。

#### ILogger

ILogger定义了系统的日志记录器的接口，例如向日志中写入日志项，获得当前日志文件名，关闭日志等等。其中日志以二进制流的形式保存。OperationFileLogger和DataFileLogger分别为记录主服务器的Region操作和数据服务器的键值更新操作实现了ILogger接口。

#### ILogEntry

ILogEntry定义了日志项的接口，其中包含获得当前操作类型函数。IOperation扩展了ILogEntry，并且用于记录主服务器中的Region操作，目前共有拆分和合并两种操作需要记录。IMutation扩展了ILogEntry，并用于记录数据服务器中的键值操作，目前共有更新（set）和删除（delete）被需要记录。

#### LoggerIterator

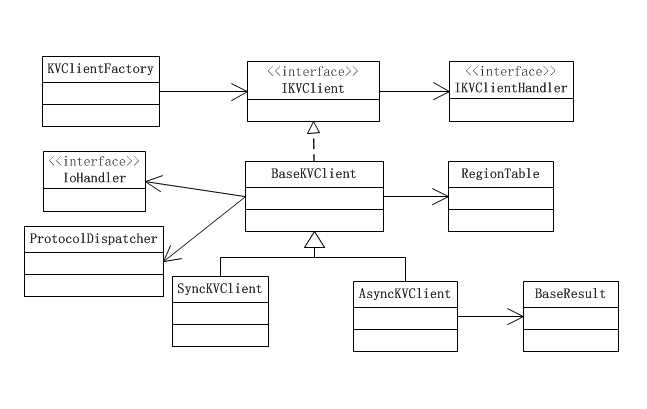
LoggerIterator通过迭代器的方式封装日志的读取操作，每次迭代返回一个IOperation实例和IMutation实例。主服务器或数据服务器可根据日志项的类型，进行相应的类型转换，并且执行相应的操作。

为避免单个日志文件过大，在主服务器中，每次创建检查点之后，操作日志也会被重新创建；同样在数据服务器的持久化存储引擎中，每次内存缓冲中的数据写入文件时，系统也会创建新的日志文件。

## 客户端

客户端程序库（下文简称客户端）是系统提供给应用程序以使用系统数据存储服务的程序库。目前系统已提供了基于Java和C#的客户端，并且只要按照5.6.2消息格式定义的消息传输规范，便可基于Socket连接实现任意语言的客户端。

客户端应包括动态路由、集群通信等功能，并实现系统定义的编程接口。其中动态路由的功能将在6.4.1自动路由中详细介绍。



图表 5‑11

图表 5‑11描述了系统Java客户端的主要类图。其中IoHandler、ProtocolDispatcher等组件将在5.6通信模型中统一介绍。

#### IKVClient

IKVClient定义了系统客户端的主要接口，例如get、set、更新Region表、关闭客户端等操作。

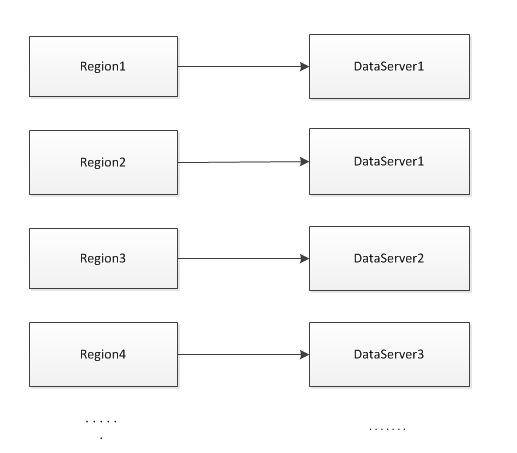
#### SyncKVClient

SyncKVClient为IKVClient基于同步模型的实现，即所有的调用操作都为同步执行。SyncKVClient的具体实现将在5.5.1同步模型中详细介绍。

#### AsyncKVClient

AsyncKVClient为IKVClient基于异步模型的实现，开发者需要实现IKVClientHandler并注册到客户端中，并且会由AsyncKVClient取得结果时进行回调。AsyncKVClient的具体实现将在5.5.2异步模型中详细介绍。

#### Region表（RegionTable）



图表 5‑12

Region中定义了键到数据服务器的映射关系，并且由主服务器负责管理。图表 5‑12描述了Region表的基本结构。在Region表中，包含有序的Region数组和Region到数据服务器地址的哈希表。当需要查找某个键对应的数据服务器时，首先通过二分查找法在Region数组中查到相应的Region，然后再从哈希表中拿到对应的地址即可。整个查找过程的时间复杂度为O()，其中n为Region数量。

### 同步模型

在SyncKVClient中，所有的接口都以同步的方式进行调用。但是由于Mina框架本身是异步的网络通信框架，因此系统在Mina传输消息之后，会阻塞当前线程，并且当Mina的事件处理线程收到消息之后，再唤醒当前线程。

同步模型使用简单，易于理解，符合开发人员常规的编程习惯。但缺点是会阻塞当前线程，并且由于网络传输的等待时间可能较长，因此不利于程序的快速响应以及高并发性。

### 异步模型

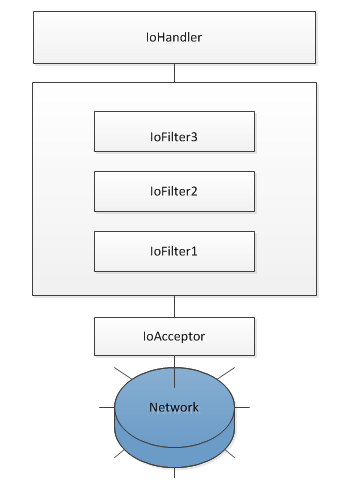
在AsyncKVClient中，开发人员需要实现客户端定义的IKVClientHandler接口，并且客户端会以回调的方式通知开发人员。例如对于get操作，开发人员需要首先调用AsyncKVClient中的get方法，之后当数据服务器返回结果时，客户端会调用IKVClientHandler中的onGet方法，并传入GetResult参数。GetResult中包含了键、值、生存时间信息以及异常信息，当发生异常时，GetResult会在开发人员尝试获得结果值时抛出异常，以此强制开发人员处理错误情况。另外AsyncKVClient中的方法都将返回空值或0，因此在异步模型中，AsyncKVClient的方法的返回值通常是无意义的。

异步模型不会阻塞当前线程，而是通过客户端的线程池以回调的方式告知应用程序结果，因此可以多个请求并发执行，有利于降低应用程序的响应时间。然而异步模型较为复杂，需要开发人员将传统的串行执行的代码，拆分成以监听器回调的方式，可能会增加程序的复杂度。因此异步模型较适合数据访问密集型而非计算密集型的应用。

## 通信模型

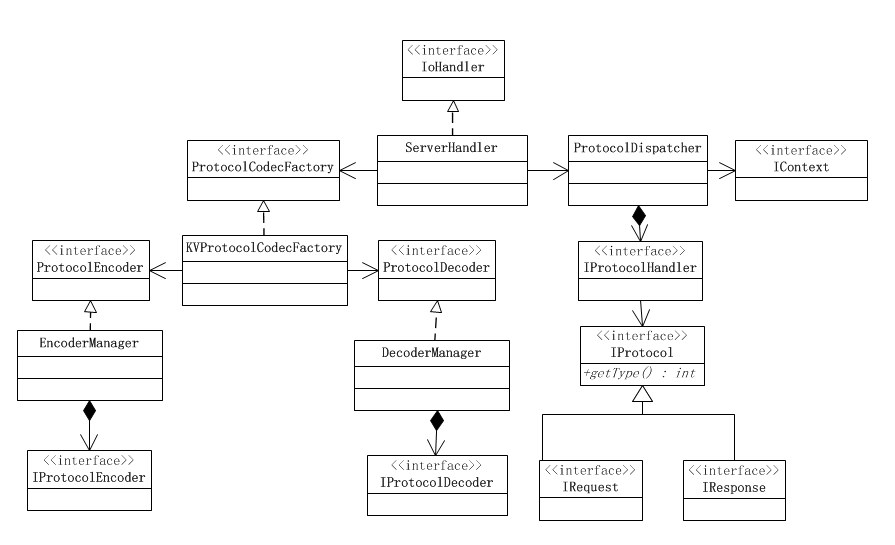
### 模型概述

系统的网络通信模型采用采用Mina的异步通信模式，即基于异步消息机制完成网络IO操作。



图表 5‑13

图表 5‑13描述了Mina构建的TCP服务器的网络通信模型。最底层的IoAcceptor负责创建Socket连接点，并绑定到机器的IP和端口上；中间层为过滤器组成的调用链，过滤器可对消息进行编码、解码，或者记录日志等操作；最顶层为用户定义的IoHandler，当网络事件发生时被Mina回调。



图表 5‑14

系统在Mina网络通信模型的基础上加以封装，将每个请求与响应都封装为对象，并根据消息类型，分派到对应的消息处理器。同时每个消息都有对应的编码器和解码器，以允许对象进行网络传输。图表 5‑14描述了系统通信框架，其中省去了IProtocol、IProtocolEncoder和IProtocolDecoder的具体实现类。

#### ServerHandler

ServerHandler是IoHandler的实现类。当网络事件发生时（收到消息），ServerHandler中相应的方法会被Mina调用，并且将根据消息的类型通过ProtocolDispatcher，把消息交由对应的Handler进行处理。

#### IProtocol

IProtocol接口定义了服务器间以及客户端与服务器间交互的消息类。其中每个消息都必须有唯一的类型值，供ProtocolDispatcher进行消息分发以及消息的编码器、解码器使用。

#### IProtocolHandler

IProtocolHandler是系统中定义的消息处理类，其中每一种消息类型都对应了一个IProtocolHandler的实现，并且每一个实现类都根据相应的消息类型注册到ProtocolDispatcher中，并由顶层的ServerHandler进行调用。

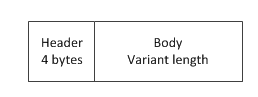
#### 编码器/解码器

编码器/解码器的作用是将消息对象转为二进制流或从二进制流中读出消息对象。对于服务器之间的消息对象传输，为了简单，直接使用对象序列化机制进行传输；而对于客户端与服务器之间的消息，为了支持多语言版本的客户端，则需要提供相应的编码器/解码器实现消息对象与二进制流之间的转换。具体的消息格式将在章节5.6.2消息格式中详细介绍。

通过上述的通信模型，为系统添加新的消息只需要定义IProtocol的实现类，实现对应的IProtocolHandler，同时根据消息类型注册到ProtocolDispatcher中。另外如果消息涉及到客户端交互，则另外需要按照系统的消息对象格式定义消息的编码器和解码器。

### 消息格式

消息格式定义了实现了IProtocol的消息对象到二进制流的相互转换规则。系统的网络消息采用定长头+变长体的格式（图表 5‑15），即每个消息开头都为四字节的整数以标明变长消息体的长度。



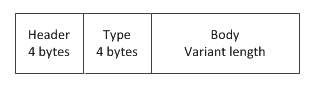
图表 5‑15

关于消息格式有如下几点约定：

#### 数据的网络传输都采用大字节序（Big-Endian）的格式。

#### 对于二进制数组的传输，采用四字节整数的长度头加内容的格式。

#### 字符串当作二进制数组进行传输，并且字符集都为UTF-8。



图表 5‑16

图表 5‑16简要描述了系统中的消息格式。每个消息开头都是四字节的整数以指明消息的长度，和四字节整数以指明消息类型。对于服务器之间通信的消息对象，消息体中直接包含是序列化后的消息对象，同样接收者也可直接通过反序列化的方式读出消息对象；而对于客户端与服务器之间通信的消息对象，则按照消息对象中字段的顺序，依次将字段值写入二进制流，读取时同样依次按照字段顺序读出即可。

例如，GetRequest为客户端根据键查找值的请求，GetRequest中的字段包括（表格 5‑1）：

|  |  |  |
| --- | --- | --- |
| 类型 | 名称 | 含义 |
| boolean | retry | 客户端是否要进行重试 |
| byte[] | key | 要查找的键 |

表格 5‑1

对应的消息格式为（表格 5‑2）

|  |  |
| --- | --- |
| 类型 | 含义 |
| int | 消息长度 |
| int | 消息类型 |
| byte | 0表示false，1表示true |
| int | 键长度 |
| byte[] | 键数据 |

表格 5‑2

由于系统中定义的消息对象较多，由于篇幅限制，本文中不再一一列举。具体可参考系统提供的JavaDoc以及源码。

# 功能实现

## 集群启动与关闭

系统中包含有主服务器、备份主服务器和数据服务器三种类型的服务器，因此为保证系统的正常工作和运行，三种服务器应有正确的启动和关闭顺序。

### 启动顺序

对于集群中的系统，推荐应首先启动所有的主服务器，等主服务器全部就绪之后，再启动数据服务器。

#### 启动主服务器

主服务器可通过运行系统提供的脚本“master\_server.bat”启动，同时应保证系统所需的配置文件在类路径中。主服务器可以同时启动多个实例。

#### 领导选举

所有的主服务器启动后将参与到领导选举算法中，其中被选举出来的主服务器将在ZooKeeper中创建“/kvstore/master/addr”节点，并将自己的ip地址和端口存入该节点中。另外，被选举出的主服务器将加载HDFS中“/kvstore/master/checkpoint/”目录下最新的检查点以及“/kvstore/master/log/”中相应的日志文件的Region信息，并将所有的Region放入待分配Region集合中。而备份主服务器将在领导选举后暂停，以等待下一次领导选举算法的执行。主服务器的领导选举算法将在章节6.1.3领导选举中详细介绍。

#### 启动数据服务器

数据服务器可通过运行系统提供的脚本“data\_server.bat”启动。数据服务器启动后，会在ZooKeeper中查看“/kvstore/master/addr”节点，如果节点不存在，则等待一段时间后重试；否则取得节点中保存的ip地址和端口，并尝试进行连接相应的主服务器。

#### 建立连接

主服务器收到数据服务器的连接建立请求时，会检查当前已连接的数据服务器中是否包含新连接的数据服务器，以避免重复连接的情况。之后主服务器会将新的数据服务器添加到自己的数据服务器表中。如果数据服务器先于主服务器启动，并且自身已拥有了某些Region，则数据服务器会通过定期心跳的方式将自身包含的Region汇报给主服务器，主服务器会进行同步，即将Region从未分配的Region集合中删除，并且分配给对应的数据服务器。

当所有数据服务器启动完毕时，主服务器的任务线程将开始定期执行（用户可设置任务线程启动前的等待时间）。之后通过Region分配任务，系统中所有之前已存在但未被分配的Region将分配给对应的数据服务器，整个系统便可正常的响应客户端请求。

### 系统关闭

对于主服务器或数据服务器的关闭，只需通过ctrl+c命令关闭“master\_server.bat”和“data\_server.bat”即可。

#### 关闭备份主服务器

首先应关闭所有的备份主服务器，以防止当主服务器先关闭时，后续的备份服务器又会被选举为新的主服务器。

#### 关闭主服务器

备份主服务器关闭后，便可主服务器。主服务器关闭时会将相应的日志文件关闭，并断开与ZooKeeper、HDFS以及所有的数据服务器之间的连接。

#### 关闭数据服务器

当主服务器与数据服务器断开时，数据服务器会自动尝试进行重连，但如果用户设置了最大尝试次数，那么数据服务器将在超过次数之后自动关闭。如果用户设置最大尝试次数为无限，那么用户需要手动的关闭数据服务器。

由于主服务器与数据服务器在关闭时会执行相应的资源清理、关闭文件、写入缓存等操作，因此尽量不用使用杀死进程的方式进行关闭，这样会使某些系统关闭应执行的代码无法执行，从而可能导致数据丢失的情况发生。

### 领导选举

主服务器的领导选举算法的实现基于Paxos算法[15] ，具体步骤如下：

#### 创建顺序节点

每个主服务器启动之后，都会在ZooKeeper中的“/kvstore/master/”目录下创建暂时顺序性节点。节点的名称由ZooKeeper以递增序列的形式分配。

#### 选举

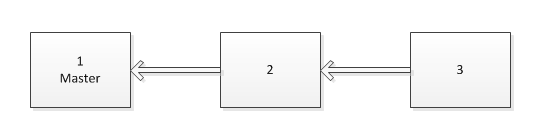
主服务器在创建完节点之后，会检查“/kvstore/master”中的所有节点。如果当前服务器创建的节点值为节点中的最小值，那么该主服务器就成为领导，否则它将成为备份主服务器，并且监听它前面一个节点的状态变化。

#### 监听

对于备份主服务器，如果它所监听的节点被删除（即对应的服务器发生故障，与ZooKeeper失去连接），那么它将重新执行第二步的选举算法。

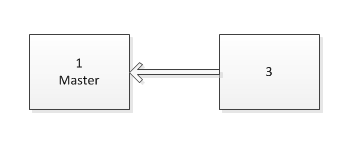
基于Paxos的领导选举算法假定每个主服务器都是对等的，并且严格按照主服务器在ZooKeeper中创建节点的时间顺序进行选举，以此保证了公平性。同时，每次当某台主服务器离开集群中，只会唤醒一台主服务器重新执行选举算法，由此避免了羊群效应。

例如，三台主服务器启动后，分别在ZooKeeper中创建了名为1、2、3的三个节点。因此基于该算法，1将成为主服务器，2和3成为备份主服务器，并且2将监听1，3将监听2（图表 6‑1）。



图表 6‑1

在某一时刻，2因为故障失去与ZooKeeper的连接，此时节点2会被ZooKeeper删除，并且3将被唤醒以重新执行选举算法。算法的结果为3将监听1（图表 6‑2）。



图表 6‑2

最后，1也失去了与ZooKeeper的连接，此时节点1会被ZooKeeper删除，并且唤醒3。3通过执行选举算法发现自己成为主服务器（图表 6‑3）。



图表 6‑3

## 故障恢复

在系统运行过程中，各个组件难免发生故障。系统通过基础设施、功能实现、以及对故障情况的具体分析等方面，以提供较好的容错机制。

### 可靠性保证

#### HDFS

HDFS本身就是一个具有高度容错性的文件系统，适合在廉价的机器上提供高可靠性的文件系统服务。HDFS认为硬件故障是常态而非异常，因此HDFS提供对数据的副本机制以避免数据丢失，以及基于文件块的校验码以快速检测故障。因此系统假定存储在HDFS中的文件是可靠的，容错的，而无需系统额外通过数据副本机制来保证数据的可靠性。

#### 操作日志

对于任何会导致系统状态变化的操作，系统都会将其写入到操作日志中，以允许当故障发生时，系统可以从操作日志中对状态进行重建。同时，系统保证了此类操作的幂等性，即操作被执行一次或多次的效果等价。主服务器中的操作日志主要记录Region的分割与合并操作，数据服务器中主要记录键值的set和delete操作。

#### 功能实现

系统在功能实现的编码层面上，同样对可靠性进行了考虑。例如主服务器中会同时保存一段时间之内的检查点文件，以允许最新检查点文件损坏时，系统可以从老的检查点文件恢复状态。另外主服务器的负载均衡任务将连续的事务分解为相互独立的子任务，以保证某一个任务的失败不会影响到其他任务的执行。同时，为保证系统的正确性，系统做了充分的单元测试、系统测试等，并通过模拟在真实环境下应用程序的请求，对系统进测试。

### 主服务器故障

对于传统的主从式服务器架构，容易出现单点失效问题，即当主服务器失效时，会导致整个系统的瘫痪。系统通过引入备份主服务器机制，可以有效的避免单点失效问题。

当主服务器失效时，某个备份主服务器会被唤醒，并将成为新的主服务器。之后，新的主服务器会加载检查点文件和对应的操作日志，以获得系统中所有存在的Region，重建主服务器之前的状态。同时所有的数据服务器在主服务器失效时也会收到通知，并尝试连接到新的主服务器中。连接建立后，数据服务器通过定时心跳汇报自己拥有的Region信息。通过上述机制，当主服务器失效时，新的主服务器会马上接替原主服务器的工作，整个系统状态不会丢失。

即使当主服务器全部失效，数据服务器在一定时间内仍处于部分可用的状态。对于那些已经从主服务器中拿到Region表的客户端，仍然可以通过老的Region表向相应的数据服务器发起请求，并且数据服务器也可进行响应。

### Data服务器故障

当数据服务器发生故障时，主服务器将收到相应的通知，并且将其从数据服务器表移除。另外，对于该数据服务器所负责的Region，将被放入未分配Region集合中。当主服务器的Region分配任务执行时，会将未分配的Region按照Region分配算法，分配给其他的数据服务器。数据服务器收到加载Region的请求时，会读取文件系统中的数据文件以及相应的日志文件，以加载Region中的数据。经过上述的一系列工作之后，发生故障的数据服务器中的数据会被分配到其他数据服务器中，因而系统可以继续提供服务。

## 响应请求

对于客户端的键值请求，如get、set等，数据服务器都会先检查负责的Region中是否包含请求的key。如果不包含，则直接返回客户端“Invalid\_Key”。因此下面的操作流程都假定数据服务器所负责的Region中包含请求的key。另外，当键值对操作执行完后，引擎代理类调用所有引擎监听器中的相应的回调方法，以通知监听器。

### set操作

set操作比较简单。对于内存存储引擎，需要执行如下操作：将键值对放入内存中，并在内存满时置换出某些键值对，同时写入操作日志。而对于持久存储引擎，除了上述之外，同时需要将键值对放入写缓存中。当写缓存满时，持久存储引擎将尝试启动RegionFlusher任务，将写缓存中的数据写入数据文件，具体步骤如下：

#### 检查当前是否有任务线程在执行，如果是，则退出；否则开始执行写缓存提交操作。

#### 在Region目录下创建名为“$timestamp”的临时数据文件，其中“$timestamp”为当前时间戳。

#### 创建新的写缓存，并读入之前的数据文件，与老的写缓存中的数据一起按键的顺序写入到新的数据文件中，类似于归并算法中的归并操作。

#### 完全写入后，持久存储引擎尝试遍历新的数据文件，并建立索引，以保证数据文件的正确性。当索引创建无误后，则清空老的写缓存中的数据，否则将老写缓存与新写缓存中的数据合并。

#### 将之前创建的临时数据文件重命名为“$regionid-$timestamp.data”，并且创建新的名为“$regionid-$timestamp.log”的日志文件，其中“$regionid”为Region的Id，“timestamp”为当前时间戳，并且数据文件与日志文件的时间戳相同。

持久存储引擎采用两步提交的方式以保证数据写入的正确性，当写入文件失败时，操作会异常退出，同时临时数据文件的名字不会变为合法的数据文件名。在之后主服务器的垃圾回收操作中，由于失败操作产生的临时文件会被删除。

### get操作

内存存储引擎的get操作需检查内存中是否存在该键值对，并且如果存在还需要检查键值对是否存活可，只有当键值对存在并且键值对仍旧存活时，内存存储引擎才会返回该键值对。

对于持久化存储引擎，则比较复杂，需要检查写缓存、内存以及文件等，具体步骤如下：

#### 从写缓存中通过键查找键值对，如果键值对存在，直接转到(5)。

#### 从内存中通过键查找键值对，如果键值对存在，直接转到(5)。

#### 通过BloomFilter初步判断是否存在与数据文件中，如果不存在，则存储引擎直接返回空。

#### 找到键对应的索引项，并将索引项对应的块全部载入内存，同时查找是否存在键对应的键值对。如果不存在，存储引擎直接返回空。

#### 现在已经取得了键值对，需要检查键值对是否仍然存活和删除标志位（isDeleted，只针对写缓存中的键值对）。如果键值对存活并且未被删除，则返回键值对，否则返回空。

### incr操作

Incr操作可以拆分为get操作和set操作，具体的incr操作如下：

#### 通过get操作，得到对应的键值对。

#### 如果键值对为空，则新建键值对，并且值为用户指定的计数器初始值；否则如果值得长度为4，则将值加上用户指定的增量，否则抛出异常。

#### 将新的键值对通过set操作放入存储引擎中。

对于内存存储引擎和持久化存储引擎，只是get与set的具体实现不同，而incr操作的流程二者则基本相同。

### delete操作

Delete操作比较简单，对于内存存储引擎，只需要将对应的键值对从内存中删除，并且记录操作日志；而对于持久化存储引擎，除了上述操作外，还需要在写缓存中放入一个键为请求键、并且标志位isDeleted为真的键值项，以在下一次缓存提交操作中，删除原有数据文件中的键值对。

### stat操作

为了简化系统的编程模型，stat操作采用定时更新的方式进行对Region的统计操作，而非每次get、set操作更新Region的统计信息。因此，Region的统计信息会有一定时间的延迟和一定的误差，但对于大规模的数据，该误差属于可接受的范围内。

对于内存存储引擎，stat操作的步骤如下：

#### 注册存储引擎监听器，当键值操作执行时，更新相应Region的读次数和写次数。同时当set、delete操作执行时，将Region的dirty位置为true。

#### 当心跳任务启动时，开始执行Region统计操作。

#### 遍历数据服务器中的所有Region，并将dirty位为true的Region放入dirtyList中。如果dirtyList的长度为0，则可以直接返回。

#### 将dirtyList中所有Region的键值数量以及大小都清0。

#### 遍历内存中所有的键值对，对于每一个键值对，尝试在dirtyList中找到对应的Region。如果Region不为空，那么将Region包含的键值数目加一，并且大小加上该键值对的大小。

#### 将dirtyList中所有Region的dirty位置为false，统计操作完成。并向主服务器通过心跳的方式发送统计信息。

而对于持久化存储引擎，stat的操作步骤为：

#### 注册存储引擎监听器，当键值操作执行时，更新相应Region的读次数和写次数。同时当set、delete操作执行时，将Region的dirty位置为true。

#### 当心跳任务启动时，开始执行Region统计操作。

#### 遍历数据服务器中的所有Region，并对所有dirty位为true的Region，进行下列操作。

#### 将Region统计信息中的键值数量设为数据文件中的键值数量，大小设为数据文件大小。（其中数据文件中的键值数量和数据文件大小在每次重建索引时进行统计）。

#### 检查写缓存中的数据，对于正常的键值对，则将键值数量加一，键值大小加上键值对的大小；而对于需要删除的键值对，则将键值数量减去一，并将键值大小减去键值对的大小。

#### 将Region的dirty位置为false。如果数据服务器仍有待处理的Region，则转入(4)，否则退出。

通过数据服务器与主服务器之间定期的汇报机制，使得统计操作可以在系统较为空闲的时间进行后台处理。而客户端调用stat操作时，主服务器直接将上次心跳的结果返回即可，而无需涉及数据服务器参加运算。不过该机制也会引入统计机制的一些误差，统计结果有一定的延迟性，另外对于持久化存储引擎写缓存中的数据，可能会出现重复统计的情况。

## 负载均衡

为平衡每个数据服务器的负载，并提高集群的总体利用率，系统提供了两种的负载均衡机制（简单均衡策略和高级均衡策略），用户可根据需求以及硬件配置状况灵活选择。

### 自动路由

由于系统中的Region被分散到不同的数据服务器中，因此应用程序使用客户端访问集群时，客户端应自动根据应用程序所访问的键值对，连接到对应的数据服务器。这一过程被称为自动路由，具体过程如下：

#### 客户端检查本地是否存在Region表，如果有直接转到(3)。

#### 客户端根据用户指定的主服务器地址（可以为多个）尝试连接主服务器，并且请求最新的Region表。

#### 根据键查找Region表，取得对应的数据服务器地址。

#### 连接数据服务器，并发送相应的操作请求。

#### 收到回复后，检查消息的返回值。如果返回值为“Invalid\_Key”，并且没有重试过，则转到(2)，否则退出。

对于数据服务器，如果某个客户端所请求的键值对不属于该数据服务器时，数据服务器会返回值为“Invalid\_Key”的消息。而客户端收到该消息后，发现本地的Region表已经过期，将向主服务器请求更新Region表后重新连接到新的数据服务器。理论上客户端重试一次之后将连接到正确的数据服务器，但是为了避免当系统的Region表出现故障时客户端无限重试的情况，系统要求客户端必须最多只能重试一次，否则将直接向调用者返回错误信息。

### Region分配

当主服务器中的未分配Region集合非空时， Region分配任务会将集合中的Region分配到某些数据服务器中。通常有如下几种情况会导致主服务器中的未分配Region集合非空：主服务器刚启动、数据服务器退出或某个数据服务器的Region负载过多，以至于被Region卸载任务所卸载。

简单均衡策略的目的是将所有的Region按数量平均分配给数据服务器，以保证每个数据服务器中的Region数量基本相同。为了分配Region，简单策略会首先将所有的数据服务器按Region数量升序排序。之后，依次向数据服务器中分配Region，直到其数量达到所有数据服务器中Region数量的最大值。如果未分配的Region还有剩余，则逐个依次向每个数据服务器分配Region，直到所有的Region被分配完成即可。

而高级均衡策略的目的是将所有的Region按照其负载量以及每个数据服务器的负载能力进行分配。Region负载量需要考虑Region的大小、读次数和写次数，数据服务器负载能力则需要考虑空闲内存、总内存、CPU使用率以及权值。另外为了叙述方便，定义符号，即Ai与A1-An的平均值的比值，系统用该比值来衡量系统的某个Region或数据服务器的负载量。高级均衡策略中Region分配的详细计算步骤如下：

#### 计算每个Region的负载量，计算方法为。由于RegionSize对Region负载的影响较大，因此将其权值设为2。

#### 计算每个数据服务器的负载因子，计算公式为。

#### 计算每个数据服务器已分配的Region容量，计算公式为。

#### 将数据服务器按照容量升序排序，并依次向数据服务器中添加Region，同时将Capacity(Si)加上Load(Ri)/Factor(Si)，直至当前数据服务器的负载超过所有数据服务器的最大负载。

#### 如果Region还有剩余，则逐个依次向每个数据服务器中分配Region，直到所有的Region被分配完。

根据计算过程可以看出，高级的系统策略基于如下考虑：每个Region的负载可由大小、读次数和写次数决定；每个数据服务器的能力可由空闲内存、总内存、CPU使用情况以及用户指定的加权因子决定；同时，对于同一个Region，放到不同的数据服务器上，对数据服务器带来的负载也不同，例如放到性能强大的数据服务器上可能占用的资源比例较小，而放到性能较弱的数据服务器上则占用的比例较大。

当主服务器的Region分配任务计算出每个Region的目标数据服务器后，会向对应的数据服务器发送Region加载请求。之后数据服务器会根据请求的内容，加载对应的Region，并返回操作结果。对于内存存储引擎和持久化存储引擎，有不同的操作过程。

内存存储引擎直接根据Region加载请求的Id，找到Region的存放目录，并且将最新的日志文件载入内存即可。同时在载入的过程中，会检查内存的使用情况，并当内存已满时置换出相应的键值对。

持久化存储引擎找到Region的存放目录之后，会按照时间倒序依次尝试加载数据文件，如果数据文件加载成功，则加载相应的日志文件。在加载数据文件的过程中，持久化存储引擎会对其进行遍历，并建立索引。如果索引建立失败，则说明数据文件在生成过程中有错误发生，持久化存储引擎则会尝试加载前一个数据文件。

### Region卸载

Region卸载主要发生在某个数据服务器的Region负载远超过其他数据服务器的负载的情况。系统根据某个数据服务器的负载量与数据服务器的最小负载量比值进行判断，如果该比值超过了用户指定的阈值，则需要对该数据服务器中的某些Region进行卸载。

简单均衡策略直接根据每个数据服务器的Region数量进行处理。如果某个数据服务器的Region数量与数据服务器Region数量最小值的比值超过了用户指定的阈值，则系统会随机选择数据服务器中的某些Region进行卸载，直至Region数量小于所有数据服务器Region数量的平均值。

复杂均衡策略则按照章节6.4.2Region分配描述的数据服务器容量Capacity(Si)的计算方法。如果某个数据服务器的容量与数据服务器中的最少容量比值超过了用户指定的阈值，则系统会随机选择数据服务器中的某些Region进行卸载，直至容量少于所有数据服务器容量的平均值。

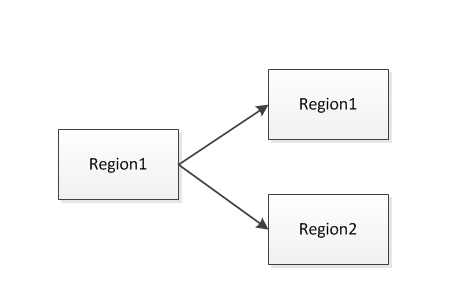
主服务器中的Region卸载任务选择要卸载的Region后，会向对应的数据服务器发送Region卸载请求。之后，数据服务器会根据收到的请求，对Region进行卸载，并向主服务器报告结果。如果Region卸载成功，则该Region将被主服务器放在未分配Region集合中，等待下一次Region分配任务启动时进行分配。

数据服务器中Region的卸载操作比较简单，只需要关闭相应的日志文件，从缓存中清除该Region所包含的数据，并将该Region从Region集合中删除即可。日志文件被关闭时，HDFS会将内存中的数据写入磁盘中，如果日志文件不能被正常关闭，则可能出现日志文件末尾数据丢失的情况。

另外，由于Region的卸载计算比较耗时，并且大部分时间内没有数据服务器需要卸载Region，因此为节约主服务器的计算资源，最好将Region卸载任务的间隔时间设置的较长。

### Region拆分

Region拆分操作发生在当某个Region的大小超过了用户设定的阈值时，Region的默认最大值为1GB。如图表 6‑4所示，Region拆分操作将某个Region拆分成两个Region，并且拆分后的两个Region所包含的键值对大小基本相同。对于Region的拆分操作，不区分简单策略和高级策略，都是通过大小进行判断。



图表 6‑4

主服务器中的Region拆分任务线程会定期遍历系统中的Region，如果发现某个Region的大小超过了用户指定的阈值，便会生成新的Region Id，并向相应的数据服务器发送Region拆分的请求。Region Id采用递增序列的方式产生，并且当前值保存在ZooKeeper中。

当数据服务器收到Region拆分请求时，便会对指定的Region进行拆分操作，并向主服务器报告拆分结果。对于内存存储引擎和持久化存储引擎，有不同的拆分方式。

对于内存存储引擎，Region拆分的步骤如下：

#### 遍历内存中的所有键值对，并将属于将要拆分Region中的键值对放到链表中，同时统计这些键值对的大小。

#### 遍历链表中的键值对，并同时累加键值对的大小，直至大小超过Region键值对的总大小的一半，同时记录下键的结束位置。则新Region的开始键为该结束键加一，结束键为老Region的结束键，而老Region的结束键设为该结束键。

#### 为新、老Region分别创建日志文件，并且遍历老的日志文件，将记录项分别写入到新、老Region的日志文件中。

内存存储引擎的Region拆分操作主要计算出新老Region的开始结束键，以及将日志文件进行分割，而对内存中的键值对没有更改。

而对于持久化存储引擎，Region拆分的步骤如下：

#### 检查当前是否有任务线程正在运行，如果有，则直接退出；否则启动Region拆分任务线程。

#### 分别创建新老Region的临时数据文件，并且都置于老Region的目录下。老Region文件的命名为“$timestamp”，新Region文件的命名为“$timestamp-new”，其中“$timestamp”为当前时间戳。

#### 遍历写缓存并检查老Region的数据文件的大小，统计出所有键值对的大小。

#### 遍历写缓存和老Region的数据文件，按照大小平均的方式分别写入到新老Region的临时数据文件，同时记录下二者的边界键。

#### 分别遍历新老Region的临时文件，并建立索引。

#### 索引建立无误后，将新老Region的数据文件命名为“$regionid-$timestamp.data”，其中“$regionid”为Region的Id，并将新Region的数据文件移动到新Region的目录下。之后分别为新老Region建立名为“$regionid-$timestamp.log”的日志文件。

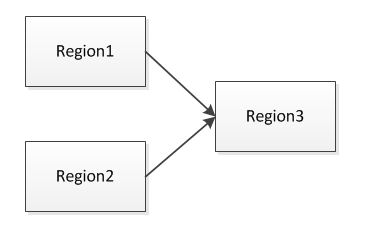
#### 最后将新Region的开始键设为边界键加一，结束键设为老Region的结束键，而老Region的结束键设为边界键。

在持久化存储引擎的Region拆分操作过程中，如果有错误发生，那么临时数据文件将不会被命名为合法的数据文件，并且将在一段时间之后被主服务器的垃圾回收线程所删除。

在系统的启动过程中，如果主服务器没有发现任何检查点文件或日志文件（通常是新系统启动），则会默认创建一个Region并包括所有的键值对。在系统的运行过程中，该Region会被逐渐拆分，并均衡到系统中其他的数据服务器中。

### Region合并

Region合并操作发生在某两个相邻Region所含有的键值对的总大小小于用户设定的Region最大值的一半时。如图表 6‑5所示，Region合并操作将两个相邻的Region为一个新的Region。所谓相邻Region，即一个Region的开始键应为另一个Region的结束键加一，只有两个Region相邻，才能进行合并。



图表 6‑5

主服务器中的Region合并任务线程会定期的对数据服务器的Region进行遍历，并依次检查两个相邻的Region是否可以合并。由于数据服务器中的Region以有序集合的方式存储，因此只需对Region进行顺序检查即可。如果两个Region可以合并，那么主服务器会生成新的Region Id，并向数据服务器发送合并Region请求。

当数据服务器收到Region合并请求时，会进行Region合并操作，并将合并结果报告给主服务器。

对于内存存储引擎，Region合并的操作步骤如下：

#### 创建新的Region对象，其键范围包括了要合并的两个Region的键范围，同时统计信息为两个Region统计信息的和。

#### 为新的Region创建日志文件，并且遍历待合并的两个Region的日志文件，按照顺序添加到新的日志文件当中。

#### 从数据存储引擎中将原有的两个Region删除，并添加新的Region。

而对于持久存储引擎，Region合并的操作步骤如下：

#### 检查系统中是否有任务线程正在执行，如果有，则退出；否则启动Region合并线程。

#### 为新的Region创建名为“$timestamp”的临时数据文件，其中“$timestamp”为当前时间戳。

#### 将待合并的两个Region的写缓存和数据文件按顺序分别写入到Region的临时数据文件中，类似于两次写缓存提交操作。

#### 遍历临时数据文件，并建立索引。

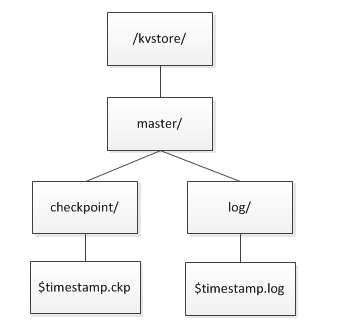
#### 索引建立无误后，将临时数据文件重命名为“$regionid-$timestamp.data”，其中“$regionid”为新Region的ID，并为新Region产生日志文件。

为避免Region合并失败时的影响，系统的Region合并操作将产生一个全新的Region，而并非在待合并的Region上进行更新。

## 垃圾回收

由于系统运行过程中可能由于某些失败操作产生一些垃圾文件，并且会有过期的检查点文件或数据文件，因此主服务器会定期执行垃圾回收任务，对文件系统进行清理。

### 主服务器目录



图表 6‑6

图表 6‑6描述了主服务器的目录结构。检查点文件位于“/kvstore/master/checkpoint/”，并且命名格式为“$timestamp.ckp”，日志文件位于“/kvstore/master/log/”，并且命名格式为“$timestamp.log”，其中“$timestamp”为文件创建时的系统时间戳。

主服务的检查点目录和日志目录中只应当存在检查点文件、日志文件以及系统在生成检查点失败时产生的临时文件。用户可以分别指定检查点文件和临时文件保存的时间。

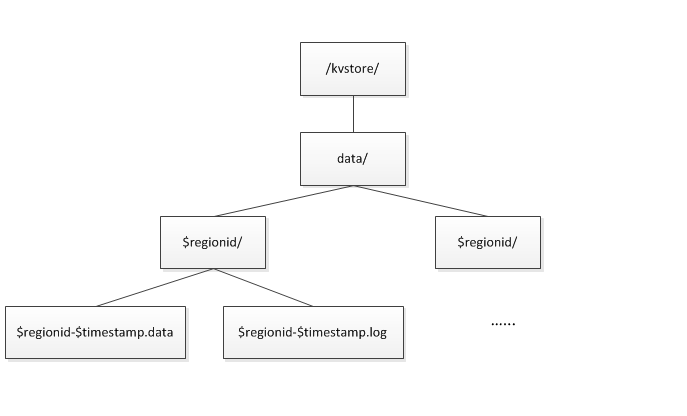
具体的检查步骤如下，对于检查点目录和日志目录下的每一个文件，执行如下操作：

#### 检查文件命名规则，如果属于检查点文件或日志文件，转到(2)，否则转到(3)

#### 从文件名中解析出文件的创建时间，并于当前时间比较，如果差值大于检查点文件的保存时间，则将其删除。

#### 通过HDFS获得临时文件的创建时间，并于当前时间比较，如果差值大于临时文件的保存时间，则将其删除。

### 数据服务器目录



图表 6‑7

图表 6‑7描述了数据服务器的目录结构。数据服务器中的Region目录为“/kvstore/data/$regionid”中，其中“$regionid”为Region的ID值。在内存存储引擎中，该目录只会存放命名格式为“$regionid-$timestamp.log”的日志文件，而在持久存储引擎中，该目录下会存放命名格式为“$regionid-$timestamp.data”的数据文件、日志文件以及由Region操作失败时产生的临时文件，其中“$timestamp”为文件创建时的系统时间戳。用户可以分别指定Region数据文件和临时文件的保存时间。

具体的检查步骤如下，对于“/kvstore/data/”下的所有子目录，执行如下操作：

#### 如果当前Region目录为空，并且主服务器的Region表中不包含该Region，那么删除该目录，否则转到(2)。

#### 列出当前Region目录中的所有文件，并遍历每个文件。

#### 检查文件命名规则，如果属于数据文件，则放入数据文件集合中；如果属于日志文件，则放入日志文件集合中，并转到(5)。

#### 通过HDFS获得临时文件的创建时间，并于当前时间比较，如果间隔大于临时文件的保存时间，则将其删除。

#### 检查目录中是否还有文件存在，如果有，那么选择下一个文件，并转到(3)。

#### 将数据文件集合和日志文件集合按照文件名升序排序，即先创建的文件在前，后创建的文件在后。

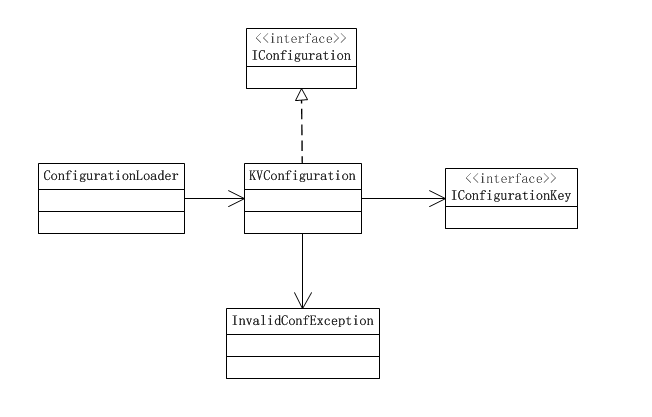
#### 分别对于数据文件集合和日志文件集合的前n-1个文件，如果文件名中的时间戳与当前时间的差值大于Region数据文件的保存时间，就将其删除，否则退出循环。其中n为数据文件集合和日志文件集合的大小。

第(7)步的目的是因为Region的数据文件不是以固定的时间间隔产生，而是取决于Region数据的写入量。因此通过上述算法可以避免当某个Region长时间没有新的数据文件产生时系统将所有的Region数据文件全部删除的情况。

## 系统配置

为提高系统的灵活性与可配置性，系统尽可能的将所有可能变化的参数都作为从配置文件读取的方式。系统的配置文件采用Java属性文件的格式，即每行对应了用等号分割的键值对。

系统中通常有两个配置文件，一个名为“kvstore.default.properties”，其中定义了系统中所有可能出现的配置信息，并提供了相应的默认参数和注释信息。而另一个名为“kvstore.properties”的配置文件需要用户提供，并且其中的配置信息将覆盖“kvstore.default.properties”文件中的配置信息。



图表 6‑8

图表 6‑8描述了系统配置相关的主要类图。其中IConfiguration定义了配置类的接口，如根据键取得相应值。同时，IConfiguration的实现类应负责值的单位转换，如将配置文件中的天数转换为毫秒数。IConfiguration的调用者会在调用时进行检查，如果用户提供的参数值有误，那么将抛出InvalidConfException异常。

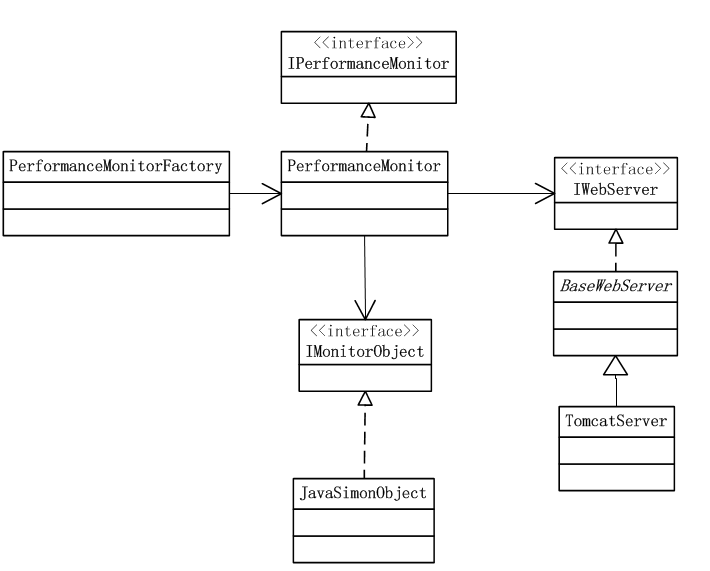
ConfigurationLoader负责对配置文件的加载。在系统启动过程中，ConfigurationLoader将首先加载“kvstore.default.properties”文件，之后将尝试加载“kvstore.properties”文件，如果该文件存在，那么该文件中的键值对将覆盖之前读取的配置信息。

系统的配置文件中包含了如主服务器地址、数据服务器地址、HDFS地址、ZooKeeper地址、数据服务器存储引擎、内存置换算法、负载均衡策略、Region大小等内容，具体可参考系统提供的“kvstore.default.properties”中的配置项以及相应的注释。

## 系统监控

### 监控模块

系统中集成了[JavaSimon](https://code.google.com/p/javasimon/)和[JavaMelody](https://code.google.com/p/javamelody/)两款开源的Java性能监控系统。JavaSimon可用于记录系统中关键操作的执行时间，并以图表的方式展现；而JavaMolody可监控Java虚拟机的运行情况，例如内存使用、异常错误信息、线程情况等。同时系统中内置了tomcat服务器，当开启监控功能时，tomcat服务器也会随主服务器或数据服务器启动时启动。



图表 6‑9

图表 6‑9描述了系统监控模块的主要类图。

#### IPerformanceMonitor

#### IPerformanceMonitor定义了系统性能监控模块的主要接口，例如根据字符串键获得对应的监控对象，以及启动、停止等。另外WebServer也由IPerformanceMonitor管理。

#### IWebServer

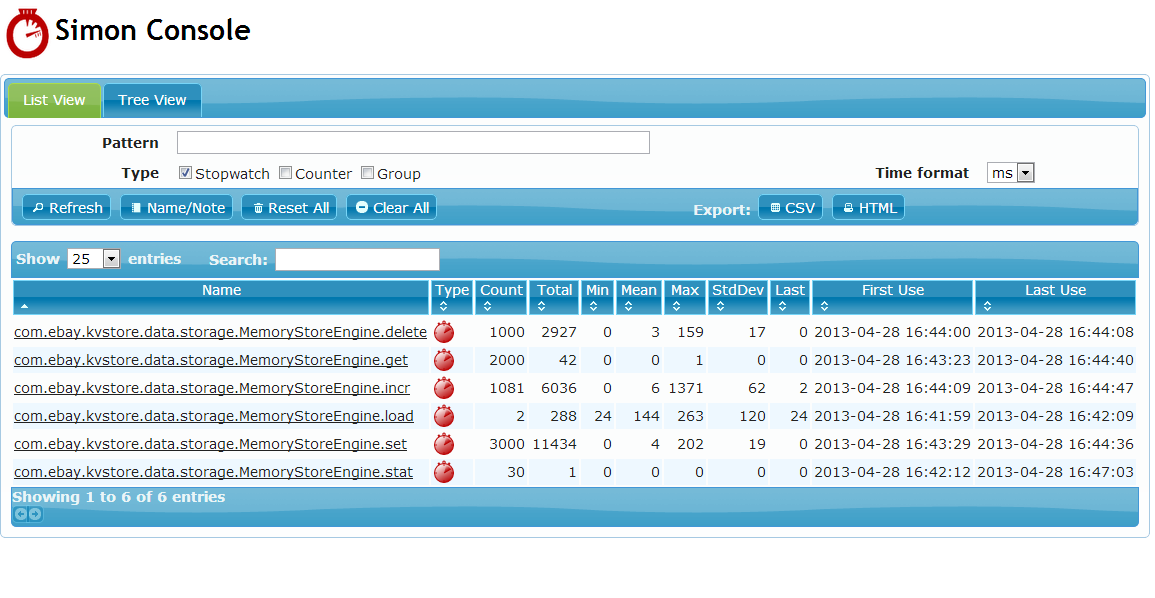
#### IWebServer封装了web服务器的接口，并且系统中提供了基于Tomcat7的TomcatServer实现。

#### IMonitorObject

IMonitorObject对应了需要监控执行时间的一个代码段。其中包含了开始和结束方法。

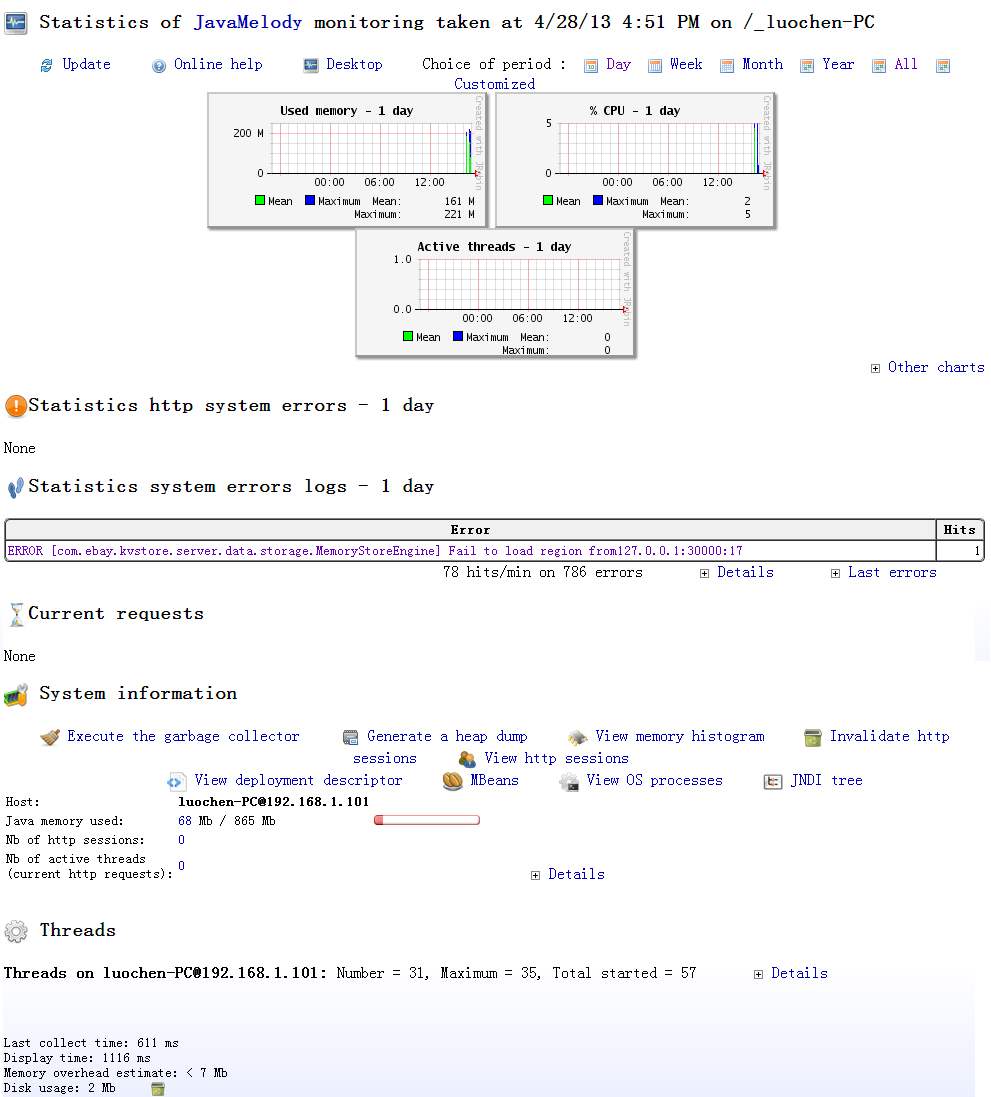
### 使用方法

对于主服务器，系统只使用JavaMolody监视机器的内存、CPU、线程等情况，而对于数据服务器，除了监视上述内容之外，系统还对get、set等键值操作以及拆分、合并等Region操作的执行时间通过JavaSimon进行监控。对于需要监控操作时间的代码段，需要在代码开始前，通过PerformanceMonitor拿到相应的IMonitorObject，并且调用开始方法；在代码结束之后，再调用IMonitorObject的结束方法。



图表 6‑10

图表 6‑10为数据服务器操作时间的统计结果页面。用户可通过地址“http://ip:port/stat”访问该页面，其中ip为数据服务器的地址，port为数据服务器的tomcat监听端口，默认为8080。从结果图中，可以看到每个关键操作的执行次数、总时间、最小时间、平均时间等数据，并且可将数据导出为CVS或HTML格式。



图表 6‑11

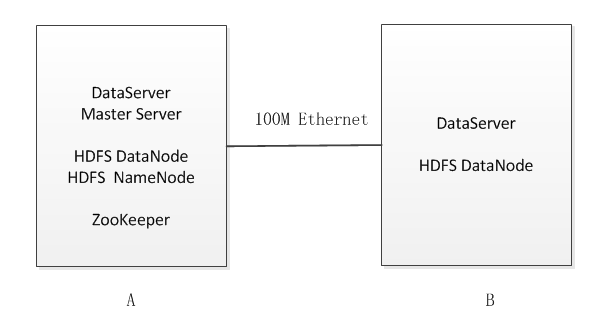
图表 6‑11为系统性能监控的页面。用户可通过地址http://ip:port/monitoring访问，其中ip为服务器地址，port为主服务器或数据服务器中tomcat默认监听的端口，主服务器为8000，数据服务器为8080。在该页面中，用户可看到系统运行期间的内存、CPU使用情况，错误日志，当前线程情况以及操作系统的详细信息等。

通过上述的监控机制，允许开发人员在系统运营时随时监控系统的运营状态，并发现潜在问题。但由于两个监控程序同样需要消耗一定的服务器资源，因此开发人员可根据自身需要决定是否使用该监控功能。

# 性能数据

## 测试环境

由于环境限制，我们只在两台机器搭建的小型集群中对系统进行了性能测试。其中机器A的硬件配置为2.8GHZ四核CPU，4G DDR3内存和500G机械硬盘， 机器B的硬件配置为2.4G四核CPU，4G DDR3内存和500G机械硬盘。两台机器通过百兆以太网的局域网相连。



图表 7‑1

如图表 7‑1所示，ZooKeeper被安装在机器A中。HDFS的命名节点也被安装在机器A中，而HDFS的数据节点被同时安装在A和B中，文件的块大小为64MB，块的副本数量为2。另外系统全部采用默认配置，并且主服务器安装在A中，数据服务器分别安装在A和B中。

## 测试方法

我们采用通过客户端调用的方式对系统的性能进行测试。在测试程序中，一定数量的线程并发的向服务器发送请求，以测试系统的性能。请求的键值对随机生成，大小由10字节到10KB不等。测试的内容包括内存存储引擎与持久化存储引擎的读操作性能（get）、写操作性能（incr、delete、set）以及随机操作性能。同时，对于持久化存储引擎，还需要观察写缓存提交、Region拆分、合并等操作对客户端响应时间的影响。

在测试过程中，测试程序将会以同步的方式进行客户端调用，同时记录下每次调用的时间，并对测试结果进行汇总。同时我们将通过系统的性能监控程序，观察数据服务器的操作时间记录，以及系统资源（包括CPU、内存和线程）的使用情况。

## 性能结果

### 内存存储引擎

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| TPS  操作 | 10 | 20 | 30 | 40 | 50 |
| 读 | 0.067/0.144/1.3(ms) | 0.063/0.21/2.8 | 0.066/0.152/2.4 | 0.048/0.111/2.091 | 0.026/0.174/2.5 |
| 写 | 0.032/0.058/0.821 | 0.031/0.049/0.463 | 0.026/0.046/0.31 | 0.02/0.048/0.838 | 0.023/0.046/1.82 |
| 随机 | 0.027/0.11/1.0 | 0.031/0.143/1.868 | 0.029/0.45/2.287 | 0.015/0.109/1.988 | 0.028/0.098/1.018 |

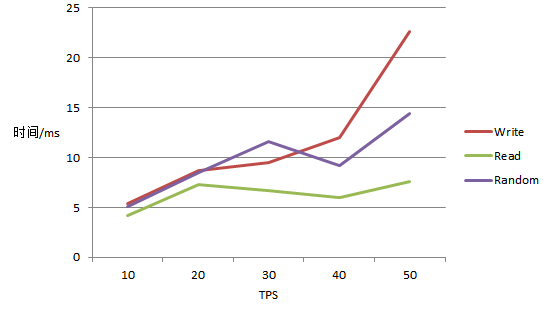
表格 7‑1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| TPS  操作 | 10 | 20 | 30 | 40 | 50 |
| 写 | 0.7/5.43/27.7(ms) | 0.512/8.68/92.5 | 0.327/9.51/62.7 | 0.851/12/73.4 | 0.621/22.6/124 |
| 读 | 0.565/4.2/34.9 | 0.369/7.27/49 | 0.336/6.7/65.6 | 0.498/5.97/59.2 | 0.341/7.6/67.7 |
| 随机 | 0.4/5.06/42.2 | 0.392/8.49/92.2 | 0.304/11.6/87 | 0.302/9.16/82.2 | 0.509/14.4/83.7 |

表格 7‑2

表格 7‑1为内存存储引擎在不同的操作以及每秒访问次数下的操作时间。从表格中可以看出，操作时间的数量级都为毫秒级。由于内存存储引擎的读、写以及随机操作都是直接操作内存以及顺序的写入操作日志文件，因此每次操作的响应时间也完成较快。

而表格 7‑2为同样的操作情况下，客户端程序的从调用开始到结束的时间。从表格可以看出，该时间要比表格 7‑1中描述的时间长很多。由于网络连接、传输以及高并发情况下多线程之间的竞争都需要占用时间，因此导致了客户端程序观察到的操作时间要大于服务器端记录的时间。同时，如图表 7‑2所示，随着并发访问量的提高，操作的所需的时间也随之提高。



图表 7‑2

### 持久化存储引擎

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| TPS  操作 | 10 | 20 | 30 | 40 | 50 |
| 写 | 0.028/0.12/1.18(ms) | 0.062/0.194/1.981 | 0.045/0.151/1.125 | 0.022/0.141/1.455 | 0.041/0.146/1.043 |
| 读 | 0.071/3.96/45.114 | 0.036/2.629/54.3 | 0.032/3.394/45.876 | 0.029/3.4/48.23 | 0.022/5.82/49.73 |
| 随机 | 0.028/2.79/16.6 | 0.038/2.19/31.1 | 0.037/2.95/22.6 | 0.017/3.1/52.9 | 0.032/4.34/62.2 |

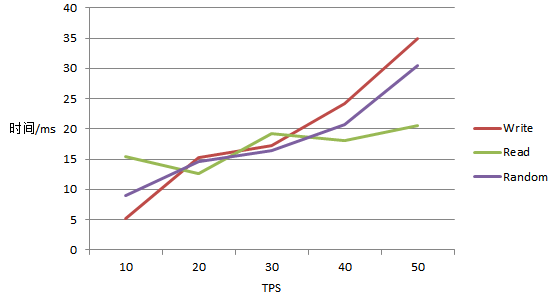
表格 7‑3

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| TPS  操作 | 10 | 20 | 30 | 40 | 50 |
| 写 | 0.322/5.14/24.5ms | 0.488/15.3/125 | 0.323/17.3/111 | 0.273/24.2/215 | 0.416/35/219.4 |
| 读 | 0.58/15.36/89 | 0.297/12.59/104 | 0.511/19.28/106.3 | 0.238/18.01/86.4 | 0.346/20.6/92 |
| 随机 | 0.273/8.96/47.2 | 0.314/14.6/87.8 | 0.314/16.42/295 | 0.278/20.7/282 | 0.444/30.4/327 |

表格 7‑4

与内存存储引擎的测试方法类似，表格 7‑3给出了持久化存储引擎在不同操作以及不同的每秒访问次数下服务器相应的操作时间。操作时间同样以毫秒级为单位。不过从表格中可以看出，读操作所需的时间却通常大于写操作，这是因为写操作是直接的内存操作，而读操作在检查布隆过滤器后有可能去进行文件读取操作，因而测试结果也与系统的实现一致。

表格 7‑4给出了在同样条件下，客户端完成相应的操作所需的时间。与内存存储引擎的测试结果类似，客户端完成整个操作所需的时间会大于服务器键值操作的时间，并且随着并发访问量的提高而提高（如所示）。



图表 7‑3

从测试结果可以看出，数据服务器中键值操作的效率很高。但由于网络连接与传输、多线程竞争以及其他的开销，导致服务器的请求响应时间仍有很大的优化余地。因此下一步的工作将包括对数据服务器的多线程并发、代码结构以及网络连接方面的优化，以降低系统的响应时间。

# 使用场景

基于键值的NoSQL数据存储系统在应用系统中通常被用作缓存系统，以减轻数据库或应用服务器的压力。同时也适用于存储业务逻辑较为简单的持久化数据，例如用户信息、文档信息等。但由于缺少事务支持，以及模型较为简单，基于键值的数据存储系统不适用于处理业务逻辑复杂的数据。

## 缓存系统

### 概述

为减轻数据库或应用服务器的压力，应用开发者通常会缓存某些计算的结果。而本系统非常适合用作缓存系统。本系统作为缓存系统具备的优点如下：

#### 系统的内存存储引擎具有较好的读写性能，同时当内存满时被置换出的某些键值对不会影响应用程序的运行。

#### 通过指定键值对的存活时间，可使得键值对在某个固定时间之后失效，以达到更新缓存的目的。应用程序也可手动对缓存进行更新，但操作较为复杂。

#### 系统对数据库服务器提供了故障恢复以及自动路由等功能，避免了当某台服务器失效时的缓存大面积失效问题。

#### 集中式缓存相比于应用服务器的本地缓存命中率更高，只要当一台应用服务器被访问时，缓存便会产生。

通常在应用程序中，开发人员可将以下一些内容进行缓存：

#### 数据库查询结果

开发人员可缓存数据库查询的结果，以减少数据库的访问压力。对于简单的数据库查询，例如主键查询，可以直接用记录的主键作为缓存键；而对于复杂的多表连接查询，可以将SQL语句与参数拼接为缓存键，进行缓存。

#### Web服务调用结果

目前大多数应用系统都采用SOA的架构，即将应用程序功能作为服务的方式进行部署和调用。因此应用程序可缓存web服务的调用结果，以减少应用服务器的压力，同时提高系统的响应速度。

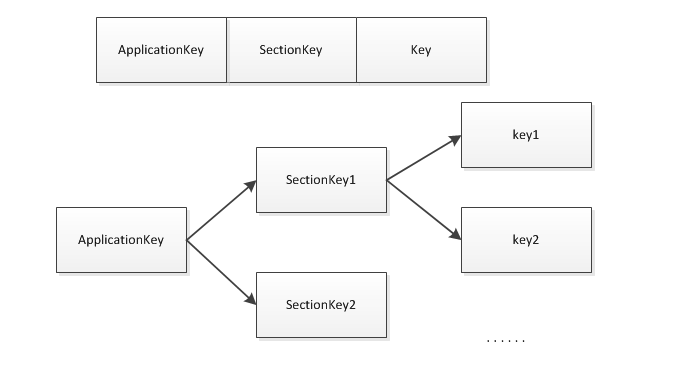
#### 序列化对象

MVC是目前主流的应用程序架构。而其中的M对应了Model，即通过一系列的计算得到的模型，并且被用于进行结果展示。因此开发人员可将序列化后的模型对象进行缓存，下次访问时直接进行缓存中反序列化即可，而无需再次进行复杂的计算操作。

#### 动态页面

目前大多数应用程序都采用动态页面的方式进行展示，例如jsp、php等技术。对于某些较长时间不变化的页面或部分页面，开发人员可将其进行缓存，以允许下次用户访问时不必重新进行渲染，极大的提高响应速度。

### 缓存模型



图表 8‑1

缓存系统可以使用图表 8‑1所示的存储模型，键通过ApplicationKey、SectionKey以及Key三者唯一生成，以对系统中的缓存数据键进行区别。另外键可以通过字节数组组合的方式进行生成。

#### ApplicationKey

ApplicationKey是唯一标识应用程序的键。由于本系统通常作为集中式的缓存系统运行，因此通过采用ApplicationKey，可以避免不同应用程序之间缓存的冲突。如果某些缓存需要在多个应用程序间共享，只需将ApplicationKey设为相同即可。

#### SectionKey

SectionKey标识了某一个应用程序中的不同的缓存数据模块，例如可以是不同的SQL数据库表，或者web服务。通常一个应用程序中会有不同的缓存数据模块，而SectionKey可以避免模块间的冲突。

#### Key

Key是唯一表示应用程序某个缓存数据模块的唯一标识，例如数据库表的主键或者web服务调用的参数。

例如，假定名为“A”的应用程序中有名为“User”的数据库表，每个用户通过唯一的数字ID进行标识，并且键之间采用“：”进行组合。因此对于主键“100000”标识的用户，其在缓存中的键为“A：User：100000”。通过以上描述的数据模型，应用开发人员可对系统提供的客户端库程序进行进一步的封装，以适应其应用场景，方便其他开发人员的使用

## 持久化存储

除了用作缓存系统之外，用户同样可以使用本系统的持久化存储引擎来持久化存储某些数据。但是由于系统的键值模型较为简单，并且没有事务等特性的支持，因此较适合存储某些数量大、访问频繁但是结构相对简单的数据，例如图片、多媒体文件对象，文档等数据。同样，由于应用程序可自己解释二进制数据的内容，因此也可存储序列化后的对象、JSON对象以及其他自定义格式的二进制数据。

考虑一个简单的文档分享系统，用户可对文档进行编辑、分享以及查看等操作。由于本系统中文档为主要数据，并且由于包含格式信息，文档的大小可能会比较大（数KB到数MB不等），并且访问比较频繁。因此为减轻SQL数据库的访问压力，应用程序可将文档数据作为键值对存放在本系统中，而其他的业务逻辑相关的数据如用户数据、用户文档关联数据等则可存储在SQL数据库中。同样，应用程序也可使用8.1缓存系统描述的数据模型，以避免应用程序之间和模块之间的数据冲突。

# 相关工作

2000年加州大学伯克利分校的Eric Brewer教授提出著名的CAP理论[11] 为奠定了分布式存储系统的基础。人们在设计分布式存储系统时，不再盲目追求三者，而是尝试为分区容忍性、数据一致性和可用性进行取舍或三者的平衡，并且学术界和工业界都出现了许多优秀的分布式存储系统。随着Google的BigTable[7] 以及Map Reduce[17] 等论文的发表，人们开始为集群的强大威力所惊叹，NoSQL数据库也开始进入了蓬勃发展的年代，各大公司都逐渐公布了自己的NoSQL数据存储系统，例如实现了高可用性的键值数据库Dynamo[6] ，面向地理范围分布的键值数据库Pnuts[18] 。同时，许多开源的NoSQL数据库也随之产生，例如高性能的键值数据库Redis，面向文档存储的NoSQL数据库mongodb等，并且被业界广泛使用。另外，学术界早先对分布式算法的研究也为分布式NoSQL数据库奠定了基础，其中Paxos算法[15] 、Gossip协议[10] 等算法被广泛应用在分布式数据库中。另外，NoSQL数据库也从传统的SQL数据库的研究中借鉴了许多思想，如并发控制[3] 、数据备份与恢复[19] 以及事务控制[20] 等内容。

# 结论与展望

本文首先介绍了分布式数据存储系统所涉及的相关概念，之后分别从架构和算法的角度，详细介绍了分布式键值数据存储系统kvstore的实现方案，并通过简单的服务器集群对系统的性能进行的测试。同时，本文对键值数据存储系统的应用场景做了简单介绍，并给出若干实例进行分析。

回顾NoSQL数据存储系统的发展历程，人们逐渐意识到NoSQL与SQL并非替代关系，而应为互补关系，这点从No被看作是“Not Only”的缩写也可看出。随着软件系统规模以及数据规模的日益增大，系统的可伸缩性也逐渐的开始被架构师所重视。而NoSQL数据库的模型简单、易于扩展等优点，相信会被大规模系统的开发人员所青睐。但与此同时，SQL数据库由于功能强大、模型完整等优点，并且性能足以满足中小型应用的业务需求，因此也不会退出历史舞台。相信在未来的发展过程中，NoSQL与SQL数据库将相互取长补短，共同发展，以更好的适应现实中的应用需求。

# 主要参考文献

Silberschatz A, Korth H F, Sudarshan S. Database system concepts[M]. Hightstown: McGraw-Hill, 1997.

Codd, E.F. (1970). "A Relational Model of Data for Large Shared Data banks”. Communications 13 (6): 377–387. [doi](https://en.wikipedia.org/wiki/Digital_object_identifier):10.1145/362384.362685

Bernstein P A, Hadzilacos V, Goodman N. Concurrency control and recovery in database systems[M]. New York: Addison-wesley, 1987.

Pritchett D. Base: An acid alternative[J]. Queue, 2008, 6(3): 48-55.

Cattell R. Scalable SQL and NoSQL data stores[J]. ACM SIGMOD Record, 2011, 39(4): 12-27.

DeCandia G, Hastorun D, Jampani M, et al. Dynamo: amazon's highly available key-value store[C]//ACM Symposium on Operating Systems Principles: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles. 2007, 14(17): 205-220.

Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data[J]. ACM Transactions on Computer Systems (TOCS), 2008, 26(2): 4.

Lakshman A, Malik P. Cassandra: a decentralized structured storage system[J]. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35-40.

Vogels W. Eventually consistent[J]. Communications of the ACM, 2009, 52(1): 40-44.

Allavena A, Demers A, Hopcroft J E. Correctness of a gossip based membership protocol[C]//Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing. ACM, 2005: 292-301.

Brewer E. CAP twelve years later: How the[J]. Computer, 2012, 45(2): 23-29.

Gilbert S, Lynch N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services[J]. ACM SIGACT News, 2002, 33(2): 51-59.

Ghemawat S, Gobioff H, Leung S T. The Google file system[C]//ACM SIGOPS Operating Systems Review. ACM, 2003, 37(5): 29-43.

Burrows M. The Chubby lock service for loosely-coupled distributed systems[C]//Proceedings of the 7th symposium on Operating systems design and implementation. USENIX Association, 2006: 335-350.

Lamport L. Paxos made simple[J]. ACM SIGACT News, 2001, 32(4): 18-25.

Bloom B H. Space/time trade-offs in hash coding with allowable errors[J]. Communications of the ACM, 1970, 13(7): 422-426.

Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.

Cooper B F, Ramakrishnan R, Srivastava U, et al. PNUTS: Yahoo!'s hosted data serving platform[J]. Proceedings of the VLDB Endowment, 2008, 1(2): 1277-1288.

Ng F K. Database backup method: U.S. Patent 4,686,620[P]. 1987-8-11.

Bortvedt G K, Gerber R H. Method of commitment in a distributed database transaction: U.S. Patent 5,799,305[P]. 1998-8-25.

【另注页面设置要求：上、下各设定4.1厘米；左、右各设定3厘米；页眉、页脚各为3.5厘米；文章内容设置行距为“单倍行距”。】