清 华 大 学

综 合 论 文 训 练

题目：分布式二级哈希表

系　　别：计算机科学与技术系

专　　业：计算机科学与技术

姓　　名：冯时

指导教师：陈文光教授

辅导教师：陈康副教授

2011 年 6 月

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定,即:学校有权保留学位论文的复印件,允许该论文被查阅和借阅;学校可以公布该论文的全部或部分内容,可以采用影印、缩印或其他复制手段保存该论文。

(涉密的学位论文在解密后应遵守此规定)

签　名:_____ 导师签名:_____ 日　　期:_____

中文摘要

随着互联网的发展和云存储的兴起,传统的分布式哈希表越来越难以满足丰富的个人在线存储应用对存储后端的需求。二级哈希表存储的是具有二级索引的数据,因此能够提供更丰富的语义。本论文阐述了分布式二级哈希表的架构设计和实现细节,说明了性能对于分布式存储系统的重要性。分布式二级哈希表综合了分布式系统领域的一些经典技术,利用配置服务器集中管理控制信息,并采用一致性哈希进行数据的分配和备份。

关键词:分布式　哈希表　存储　性能　一致性哈希

ABSTRACT

With the development of Internet and boom of cloud storage, traditional distributed hash table is too simple to support the upper layer applications which provide personal online storage service. A two-level DHT stores data with a major index plus a secondary one, resulting in more complex semantics. The present thesis describes both architecture design and implementation details of two-level DHTs, and comes to a conclusion that high performance and efficiency is fundamental aspect of a distributed system. The two-level DHT is a combination of classical technologies in distribute computing, with a config server to monitor the cluster. It also employs consistent hashing to solve data partition and replication.

Key words: distributed hash table storage performance consistent hashing

目　录

第 1 章　引言

1.1　研究背景

分布式存储系统越来越成为整个互联网赖以存在和发展的强大依托。从 20 世纪 90 年代初互联网兴起开始,我们进入了一个信息爆炸的时代。2002 年世界上共产生了五百亿亿字节的数据,其中 92% 的信息存储于电子介质中,这相当于人类历史上所有说过的话语所包含的信息量的总和。[1] 随着互联网的进一步发展和普及,信息产生的速度还在加快。单一的存储单元已经不能容纳如此巨大的信息量,分布式存储系统通过将多个存储单元组织起来,充分利用每个存储单元的资源,使得整个系统的存储容量成倍的增长,可以很好的满足大容量存储的需求。另一方面,随着云计算的迅猛发展,互联网用户希望更多的个人数据存储在云端,而不是本地个人计算机上面。分布式存储系统为云计算和云存储提供强大的后端支持,在满足海量个人数据存储的同时,使得用户不必关心数据的组织方式和存储实现。

存储系统归根结底解决的是从索引到值的映射关系,分布式哈希表是最基本的分布式存储系统。任何一种分布式存储系统归根结底都是一种广义上的分布式哈希表。只不过在这样的系统中,索引可能是比字符串更复杂的数据结构。例如在像 Google File System[2] 这样的分布式文件系统中,数据被看作文件。分布式文件系统按照层级目录的方式组织数据,用户则通过指定完整路径来索引文件,并通过读、顺序写等基本文件操作来存取数据。另一类分布式存储系统则通过支持更复杂的值类型来提供更丰富的语义。例如在分布式存储系统 PNUTS[3] 中,存储的对象可以是字符串、整数等基本数据类型,也可以是没有子结构和含义的二进制数据块 ①。此外,分布式数据库还维护了数据之间的关系,支持简单的查询语义。

分布式存储系统的根本设计原则之一,是根据上层应用的需求,在系统的简洁性和功能的丰富性之间找到一个最优平衡点。不考虑设计者的因素,功能越强大,系统势必越复杂庞大,系统的运行效率可能越低。Dynamo[4] 等经典分布式

① http://en.wikipedia.org/wiki/Binary_large_object

系统不止一次阐明了这个原则:一个分布式系统不是具备越丰富的功能越出色,而是能够高效率实现足够的功能。具体到分布式存储系统,在相同实验条件下,最基本的分布式哈希表支持的语义最简单,执行效率也最高。与之相比,分布式文件系统和分布式数据库等存储系统功能更强大,方便上层应用调用,但运行效率相对较低。

随着存储成本的降低、网络条件的改进以及安全技术的发展,云存储越来越受到人们的青睐。更多的用户希望把个人数据存放在互联网上,以增强数据存储的可靠性,节省本地存储空间,实现数据的离线传输,以及方便不同终端之间数据同步。此类应用在互联网上也层出不穷,比如以 Gmail① 为代表的电子邮件系统,以 flickr② 为例的个人在线图片存储系统等,都能满足用户在线存储数据的需要。这些应用需要存储的数据具有共同的特点:数据是按照用户分开存储的;每个用户可以存储多个数据;不要求系统维护数据之间的关系。

针对上述云存储应用的特点,我设计并实现了分布式二级哈希表。该存储系统提供简洁而易用的接口,可以作为这些应用的底层存储后端。在一个分布式二级哈希表中,数据是二进制块,对于数据的索引通过指定二级 ID 实现。这样,第一级 ID 可以用于区分不同用户,第二级 ID 则用来指定同一个用户的不同数据。数据则根据第一级用户 ID 分配到不同的存储服务器上,实现分布式存储。为了保证系统的运行效率,我在实现分布式二级哈希表的过程中借鉴并使用了部分开源代码。其中,每台存储服务器上都部署了 Redis ③ 实现本地哈希表存储,数据分布和备份采用了著名的一致性哈希算法[5],基于开源项目 libconhash ④ 作出修改实现。

1.2 相关工作

1.2.1 分布式哈希表

分布式哈希表是最基本的分布式存储系统。单机哈希表解决的是从索引到值的映射关系,而分布式哈希表则是根据索引将不同的数据分配到不同的存储服务器上,从而实现数据的分布式存储。一般方法是:先确定一个哈希函数,将此

① http://mail.google.com/mail
② http://www.flickr.com/
③ http://redis.io/
④ http://sourceforge.net/projects/libconhash/

哈希函数的值域空间按照某种方式分割成多个子空间,每一个子空间对应一台存储服务器。当我们需要确定某个数据在哪台服务器上存放时,就把这个哈希函数作用在它的索引上,得到的哈希值所在的子空间对应的服务器上就存储了或者应该存放此数据。由于分布式哈希表原理简单,实现一个高效率的系统并不难。现在已经有很多成熟的实现,比如豆瓣的 BeansDB ①,亚马逊的 Dynamo 等分布式哈希表,都具有很高的执行效率。

使用基本的分布式哈希表难以实现高效率的分布式二级哈希表。由于单个一级 ID 可能对应多个二级 ID,使用基本分布式哈希表很难罗列出某个一级 ID 对应的所有二级 ID 以及数据。我们希望能够实现一个原生支持二级哈希语义的系统,既要保证操作的原子性,又能够高效率运行。

1.2.2 分布式文件系统和分布式数据库

Google File System 等分布式文件系统和以 Bigtable[6] 为代表的分布式数据库支持比分布式哈希表更复杂的语义。在分布式文件系统中,数据被看作文件,通过路径来索引。分布式数据库则侧重数据之间的关系,支持一些匹配查询。这些系统在处理大块数据时表现出很好的性能,但在操作小数据(小于 1MB)时,系统开销则相对过大。在这种情形下,采用简单的分布式哈希表来处理小规模数据更合适一些。[4] 虽然分布式文件系统和分布式数据库可以实现分布式二级哈希表的全部语义,但是我们希望设计一种更轻量级的系统,它在处理小数据的时候能够表现出很高的性能。

1.3 论文组织

本章介绍分布式二级哈希表的研究背景和已有的分布式存储系统。第2 章定义了分布式二级哈希表要解决的问题,主要说明了用到的数据结构、系统为上层应用实现的调用接口以及系统设计和实现所基于的重要的实验假设。第 3章阐述了分布式二级哈希表的架构设计。第 4章分模块详细说明了系统的原理和实现细节。第 5章通过实验数据分析了分布式二级哈希表的性能。第 6章挖掘了系统当前设计和实现的缺陷,并提出了可能的解决方案,作为今后的改进方向。第7章对本论文作出总结。

① http://code.google.com/p/beansdb/

第 2 章　问题定义

在第3章介绍系统架构之前,本章先定义需要解决的问题。

2.1　数据结构

要存储的目标数据被称为 blob,它是一个没有子结构和含义的二进制数据块。上层应用在写入数据之前,需要先将待存储的数据转换成二进制数据块,再调用接口函数将数据写入服务器。读出数据之后,系统直接将二进制数据块返回给上层应用,具体的解析工作由上层应用完成。这样设计的优点,一是可以简化系统实现,从而提高执行效率;二是使系统的应用范围更广泛,由于任何数据在计算机内的终极存储表示都是二进制数据块,所以理论上该系统可以支持任何应用。这样设计的缺点是降低了系统的易用性,需要上层应用完成更多的序列化/解析工作。①

每一个 blob 拥有一个字符串类型的 ID,称作 blobID。多个 blob 逻辑上被划在一个 “桶” 中,这通过给每个 blob 附加另一个字符串类型的 ID 实现,该 ID 被称作 bucketID,如图 2.1所示。所有具有相同 bucketID 的 blob 被认为装在同一个桶中,同一个桶中的 blobID 是唯一的,不同桶中的的不同 blob 则可能具有相同的 blobID。这样,对于 blob 的索引是通过给出两级 ID 实现的。与最基本的分布式哈希表相比,多出的一级 ID 实际上维护了数据之间的归类关系,即按照 bucketID 是否相同将数据归入不同的类别。

2.2　接口

分布式二级哈希表作为各种云存储应用的底层存储后端,为不同的应用提供了统一的调用接口。接口的设计既要保证通用性,考虑系统实现的效率,又不能损失易用性。分布式哈希表的调用接口及说明参见表2.1。

由于分布式二级哈希表可能为多个上层应用提供存储服务,因而每一个独立的应用在使用分布式二级哈希表之前需要先创建一个服务实例② 。在调用具

① 在当前实验条件下,高效和普适性比易用性更重要。

② 类似 MFC 中的 HANDLE

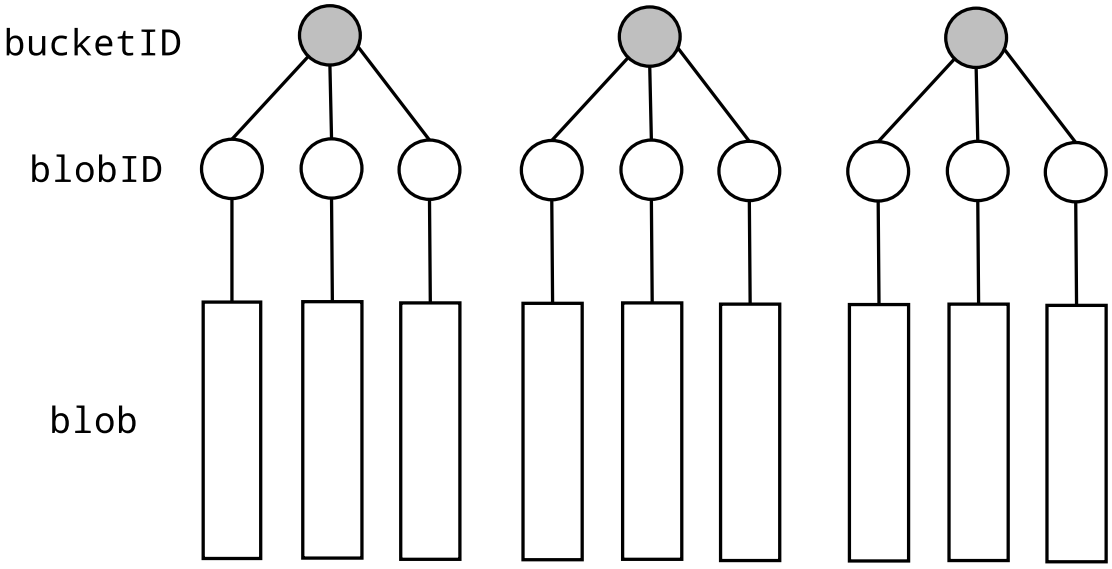

图 2.1　分布式二级哈希表数据结构。每个 blob 拥有一个 blobID 和一个 bucketID。共享相同 bucketID 的多个 blob 被归为同一类。

体的接口函数时,此实例也要作为一个参数传递给系统。出于简化的目的,本论文的叙述将略去关于实例的说明,只考虑仅有一个实例的情况。

2.3　实验假设

任何一个系统都不可能解决所有的问题,分布式二级哈希表也不例外。下面的假设在很大程度上影响了系统架构的设计:

1. 单个 blob 的大小在 1B 到 1MB 左右。具备这个规模数据的最典型应用是文本系统。比如个人邮件的文本部分,不包括附件和多媒体等内容,尺寸一般小于一兆字节。问答系统和论坛留言也大致如此。此外,矢量图和尺寸较小的压缩图也属于这个尺寸范围。这些应用一般只要求知道数据属于哪个用户,至于数据之间的关系则没有过多要求,而且一般由上层应用自己来维护关系信息,因此分布式二级哈希表可以很好的满足这些应用的存储需求。另一方面,虽然单个数据的尺寸不大,但是数据的总量可能很大。这就要求系统的算法复杂度不能太高,能在海量数据中迅速定位到指定的目标。后面我们会看到,由于采用了哈希算法,数据的定位复杂度是 O(1),能够保证系统在应对海量数据请求时保持较高的执行效率。
2. 存储服务器集群的规模大概是 50 台上下,每一台服务器的硬盘容量在 1TB 左右。由于该系统的集群规模与典型数据中心相差甚远,机器的故障发生

表 2.1　分布式二级哈希表调用接口及说明。这些接口是分布式二级哈希表为上层应用提供的调用接口,以标准 C 语法为例。

函数原型	说明
int createBucket(const char *bucketID);	创建一个空桶,该桶内不包含任何 blob。当用户刚刚注册云存储服务时,可以调用此函数。
int deleteBucket(const char *bucketID);	删除一个桶中的所有 blob。当用户注销删除账户全部数据时,可以调用此函数。
int existBucket(const char *bucketID);	查看一个桶是否存在。当需要查看一个用户是否存在时,可以调用此函数。
int deleteBlob(const char *bucketID, const char *blobID);	删除一个 blob。当需要删除用户的某个特定数据时,可以调用此函数。
int existBlob(const char *bucketID, const char *blobID);	查看一个 blob 是否存在。当需要查看用户的某个特定数据是否存在时,可以调用此函数。
int loadBlob(const char *bucketID, const char *blobID, int *blobLength, void *blob);	读取某个 blob 的内容。当需要读取用户的某个特定数据时,可以调用此函数。
int saveBlob(const char *bucketID, const char *blobID, const int blobLength, const void *blob);	将 blob 存入服务器,覆盖具有相同索引的数据。当需要存储或更新用户的某个特定数据时,可以调用此函数。

率并不频繁,[4] 可以假设大部分时间所有机器都是正常运转的。另一方面,机器故障也是有可能发生的。如果数据大部分时间保存在内存中,则需要系统定期将数据写入硬盘以应对系统错误;如果数据大部分时间保存在硬盘上,则需要利用数据局部性在内存中构建数据缓存以提高效率。为了解决硬盘发生错误带来的灾难性后果,我们还需要将数据进行备份,即在多台服务器上存储同一数据的多个副本。也就是说,同样的操作需要在多台存储服务器上进行。比如,上层应用调用了分布式哈希表的写入函数,假设数据在系统中有了三个副本,那么系统需要向这三台服务器发送写请求。这里系统需要解决的一个关键问题是如何保证副本间的数据一致性。强一致性要求数据在不同副本之间总是相同的,但是理论界和工业界普遍认可实现强一致性不仅技术难度较大,这样的系统运行效率也往往很低。[7] 与

之相对的另一种解决方案是弱一致性,它不保证数据的不同副本总是完全相同的,而是保证读出的数据总是最近一次写入的数据。本质上讲,系统可以利用数据从开始写入到最终读出之间的时间间隔实现数据在不同副本之间的同步,因而操作的平均响应时间大大缩短,而吞吐率则得以提高。

3. 要求系统在饱和运转时,操作成功率大于 99.999%,每台服务器平均每秒处理请求数不小于 50 左右,平均响应时间小于百毫秒数量级;另一方面,操作的语义成功率不作太高要求,即允许极个别情况下,读操作返回的数据版本不是最新写入的。当系统要应对大量数据请求时,由于处在高负荷运转状态,系统的响应时间会降低。要在保证系统正确性的前提下,尽量提高系统的运行效率。
4. 系统由标准 C 代码实现。存储服务器上运行 Linux 操作系统,使用标准 C 代码实现分布式二级哈希表既保证了系统的运行效率,又不会影响代码的可移植性。

第 3 章　系统架构设计

在第2章定义了要解决的问题,并确定了实验的有关假设之后,本章我来说明分布式二级哈希表的系统架构设计,而在第4章中将详细介绍系统各模块的实现细节。

对于像分布式二级哈希表这样的分布式存储系统,需要解决的关键问题主要包括以下几点:

1. 数据在集群中的单个结点上是如何存储的?
2. 如何知道集群中哪些服务器结点当前是正常运行的?哪些已经发生了故障无法响应?进一步的,如果有结点发生了故障,那么是什么类型的故障?是因为系统负荷过大暂时无法响应,还是服务器程序崩溃或者系统故障需要重新启动,亦或是硬盘错误本地数据无法恢复?
3. 数据在不同存储节点上是如何分配的?数据是如何存为多个副本的?给定索引如何确定数据存放在哪台(些)服务器上?
4. 如何保证不同副本的数据一致性?

分布式二级哈希表基于一些经典技术,在保证效率的前提下解决了这些问题,其架构如图 3.1所示。

集群中有一台特殊的配置服务器,通过与集群中其他的存储节点进行心跳通信,来确定当前有哪些机器在正常运行。当前的系统设计不区分不同的故障类型,当配置服务器在一段时限内与某存储服务器心跳通信失败,认为该服务器发生了错误,之后的操作不再涉及该服务器。由于数据有多份副本,信息不会丢失。配置服务器的另一个功能是实现了一致性哈希,对于某个特定的目标数据,根据它的 bucketID,配置服务器负责指定多个服务器结点存放该数据,并维护该信息,从而实现了数据的分配和备份。在第 4.1.1小节我将详细介绍一致性哈希算法。

分布式二级哈希表提供了一个客户端库,里面包含表2.1罗列的接口函数供上层应用调用,上层应用需要嵌入客户端库代码。当上层应用发起请求时,客户端首先通过远程过程调用[①] 从配置服务器获取存放着该数据的多台目标服务器的 IP 地址,然后按照 Redis 规定的协议,通过 socket 通信[②] 同时向这些目标服务

① http://en.wikipedia.org/wiki/Remote_procedure_call
② http://en.wikipedia.org/wiki/Computer_network_programming

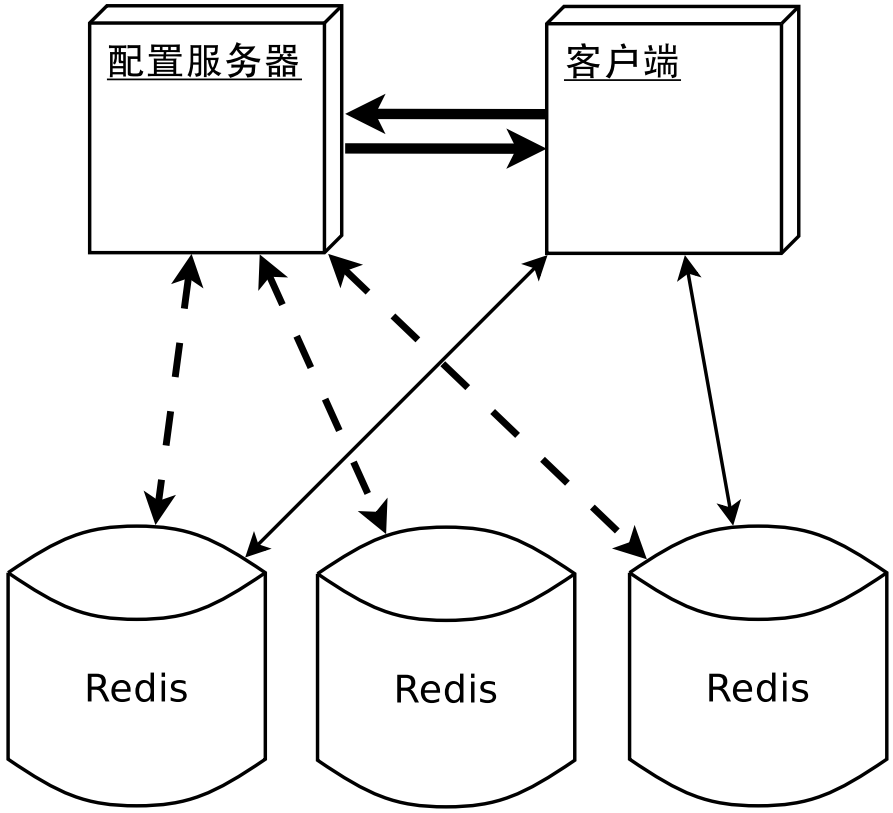

图 3.1　分布式二级哈希表系统架构。集群中设置一台配置服务器,与各节点进行心跳通信(虚线箭头),并且实现一致性哈希算法。上层应用嵌入客户端库代码,调用其中提供的接口函数,首先通过远程过程调用(粗实线箭头)从配置服务器获取某个 bucketID 对应的一系列存储服务器的地址,再遵照协议与这些服务器上的 Redis 进程进行 socket 通信(细实线箭头),完成相应操作。服务器集群中每个存储节点上运行一个 Redis 服务器进程。

器发起请求并接收应答。为了提高系统的运行效率,客户端还做了多项优化,并且通过线程池实现了异步请求,详见第4.2节。

在存储服务器集群中的每个结点上,都运行着一个 Redis 存储服务器进程。Redis 是一个开源本地哈希表项目。与一般的哈希表不同,Redis 的值类型除了可以是一般的字符串,还可以是列表、集合甚至哈希。所以 Redis 实际上已经解决了本地二级哈希的问题。Redis 的另一个优点是它在应用层实现了一个虚拟存储层,将所有数据存于内存和应用层虚存中大大提高了系统运行效率。同时,Redis 是基于日志的事务型存储系统,保证了操作的原子性,并具备一定的容错能力。有关 Redis 的更多细节我将在第4.3节做进一步介绍。

分布式哈希表的架构设计与主流分布式存储系统有很多相似之处,其优点显而易见。

1. 数据被分配到多台存储服务器之上,包含多个副本,数据的分配和备份采用一致性哈希算法。系统中包含的单个存储单元越多,单个存储单元的容量越大,系统的总存储容量就越大。数据分别存放在多台存储服务器上,提升了系统的运行效率。单个存储单元的硬盘读写速率和网络带宽有一个上限,如果客户端能够同时与多台服务器进行数据通信,便提高了单位时间内的数据交换速率,使得系统的总体运行效率大大提升。另一方面,由于系统对数据进行了备份,在不同的存储结点上保留了多个副本,即便某个存储节点由于发生了灾难性的故障导致数据永久丢失,系统仍可以通过该数据在其他正常运转的服务器上的副本将该数据恢复,这大大增强了系统的容错能力。一致性哈希算法作为经典的数据分配算法,被很多著名的分布式存储系统所采用。[4] 由于已知分布式二级哈希表的集群规模在几十台服务器左右,机器故障并不会频繁发生,当前的系统进行了数据备份,不支持存储结点动态加入和离开集群。但是一致性哈希算法自诞生之日起,受到青睐的原因之一在于它很好的支持了系统中存储结点的动态变化,所以分布式二级哈希表具备实现该功能的潜能。事实上,只要在当前系统上添加一个后台数据迁移模块,即可实现存储节点的动态添加和移除功能。
2. 集群中有一台特殊的配置服务器,监控集群中所有存储结点是否运转正常,并实现一致性哈希算法。配置服务器与集群中各存储结点进行心跳通信,了解各存储服务器的运转情况。相比传输的数据,心跳信息所占用的网络带宽和消耗的系统计算资源基本可以忽略不计。因而使用单一的服务器维护控制信息对系统性能造成的影响是微乎其微的。另一方面,集中管理控制信息与非集中策略相比,不用考虑信息同步问题,不仅简化了系统实现,也方便客户端获得此信息。此外,当前的系统架构设计假设配置服务器不会发生系统故障。如果确实需要解决这个问题,只需要再安置一台配置服务器,存储控制信息的副本,并定期从主配置服务器获取信息更新。一旦主配置服务器发生系统故障,该替补配置服务器立刻接替其工作,继承原配置服务器的属性 ① ,即可实现配置服务器的无缝替换。
3. 上层应用需要嵌入分布式二级哈希表系统提供的客户端库代码,以调用其

① 如 IP 地址。

中提供的接口函数。这种机制在一定程度上影响了系统的易用性,优点是客户端和上层应用之间不需要进行网络通信。另一种机制使得上层应用不必嵌入任何系统代码,而是按照系统规定的协议与任何一台存储结点或入口服务器通信,再由此机器经过一次路由选择将该请求发送至目标服务器,操作完成后再将返回值传回上层应用。缺点是需要附加一次网络通信,对网络带宽和操作延迟提出了更高的要求。在 [4] 中详细讨论了两种机制的优劣和适用场合。由于分布式二级哈希表在设计之初就本着尽量提高系统运行效率的原则,而其所支撑的上层应用目前都是由我们自己设计和实现的工程,所以分布式二级哈希表采用了第一种机制实现客户端。

4. 客户端通过远程过程调用从配置服务器获取存储服务器的 IP 地址,再与这些存储服务器通过 socket 通信进行实际的数据传输。在分布式二级哈希表系统中,由于控制信息只包含目标存储服务器的 IP 地址,一次操作需要获取的控制信息只有 10 字节左右。所以尽管整个系统只有一台配置服务器提供控制信息,该服务器依然可以承担相应的负担。另一方面,由于实际存储的一份数据最大有 1MB,因此实际的数据传输直接在客户端和相应的存储服务器之间进行。这种数据分散传输的机制充分利用了各个存储服务器的网络带宽和硬盘读写能力,不会由于出现某个瓶颈,而造成整个系统的运行效率降低,因而这种设计被很多著名的分布式存储系统所采纳。[2]

第 4 章　原理与实现

分布式二级哈希表是一个偏向工程的项目,涉及到的大多是已经发展成熟的技术,是对分布式领域一些经典算法的综合。但是分布式二级哈希表对系统的运行效率要求很高,因此要求在实现细节上不能有瑕疵。本章是论文中篇幅最大的一章,一方面详细阐述了分布式二级哈希表用到的经典技术的原理,另一方面也深入说明了系统的实现细节。

对于像分布式二级哈希表这样并不复杂的系统,在实现时仍然体现了模块化的思想,整个系统是由一系列模块搭建而成的。每个模块完成特定的功能,基本独立,但是在实现时也要考虑模块之间的协作关系。这些模块有些是由我独立实现的,有些在实现时参考了别人的代码,有些将别人的代码加以修改后移植到系统里来,有些则直接使用了别人的代码。所有的引用和参考都基于开源代码,并且符合作者的协议和使用条款。

分布式二级哈希表的所有模块都用标准 C 语言写成。相比与其他的脚本语言或者面向对象编程语言,C 语言有着更高的执行效率。C 语言的灵活度比较高,对于像分布式二级哈希表这样对系统执行效率要求颇高的工程,用 C 语言开发有更大的优化自由度。因为 C++ 语言可以兼容 C 语言,但反之不行,所以 C 语言有更高的可移植性。鉴于存储服务器上运行的都是 Linux 系统,C 语言无疑是最佳的选择,所以我决定选用标准 C 语言进行分布式二级哈希表的全部开发。另一方面,相比 C++ 语言和其他面向对象编程语言,C 语言的标准库和第三方库都不够丰富,这也迫使我自己实现了像线程池这样的模块,反而锻炼了自己的编程能力。

下面我将分模块详细阐述每一部分的原理和实现细节。

4.1　配置服务器

配置服务器是整个分布式二级哈希表的核心。它通过与集群中每一个存储服务器进行心跳通信,掌控集群中所有结点的运转情况。配置服务器实现了一致性哈希算法,负责决定数据如何在存储结点间进行分配和备份。当上层应用调用系统的接口函数时,客户端首先通过远程过程调用,从配置服务器获取目标数

据所有副本所处服务器的 IP 地址。在当前的系统设计中,由于数据已经有多份备份,并且机器故障发生概率很低,所以配置服务器忽略可能发生的单个结点故障。如果希望配置服务器能够解决存储结点动态加入和离开集群的情况,需要在当前的系统实现上添加数据迁移模块。

4.1.1 一致性哈希算法

一致性哈希算法自诞生之日起,就因其优雅的设计而备受青睐,并且被众多著名分布式存储系统所采纳。一致性哈希算法的优势在于,当存储结点动态增多和减少时,需要在结点间转移的数据量最小。虽然分布式二级表的当前设计并不支持结点的动态加入和移除,但是为了增强系统的可扩展性,我仍然采用了一致性哈希算法来解决数据分配和备份问题。

下面先介绍一致性哈希算法的原理。将哈希函数作用于数据的索引空间,得到的值域空间称作*哈希空间*。将哈希空间首尾相接,回绕成一个环状空间,称作*哈希环*,如图4.1所示。每一个索引的哈希值在哈希环上有唯一的一个点与之对应,我们用这个点代表具有这个索引的数据。每一个存储结点被称作一个*物理结点*,使用某个字符串作为其唯一标识,比如 IP 地址。每一个物理结点有若干*虚拟结点*与之对应,虚拟结点数一般设定为与物理结点的存储容量成正比。每一个虚拟节点也有一个唯一的字符串标识,一般通过在其对应物理节点的标识后面附加结点编号构成。图4.1中,画出了两个物理结点,其中一个物理节点画出了两个虚拟结点(深灰色),另一个物理结点只画出了一个虚拟结点(浅灰色)。将所有虚拟节点的字符串标识求哈希值,结果也是哈希环上的一点,我们用这个点代表这个虚拟节点对应的物理结点。这样,一个物理结点有几个虚拟结点与之对应,哈希环上就有几个点与这个物理结点对应。由于哈希函数的性质,与某个物理结点对应的那些点在哈希环上的分布并无规律可循,当物理结点包含足够多的虚拟结点时,这些点可以看作是随机分布的。后面我们会看到,正是这个性质保证了一致性哈希在结点动态加入和移出时,数据迁移量最小。

现在我们有了一个哈希环;对每一个数据的索引求哈希后,在哈希环上都有一点和这个数据对应;对于集群中的每一台存储服务器,在哈希环上都有若干点与之对应,并且这些点是随机分布的。在图4.1中,最大的圆环是哈希环。在哈希环上,标有字母的大圆代表数据,上面的单词是该数据的索引。哈希环上的小圆代表虚拟结点,旁边标注的字符串是该虚拟结点的唯一标识。同样灰度的

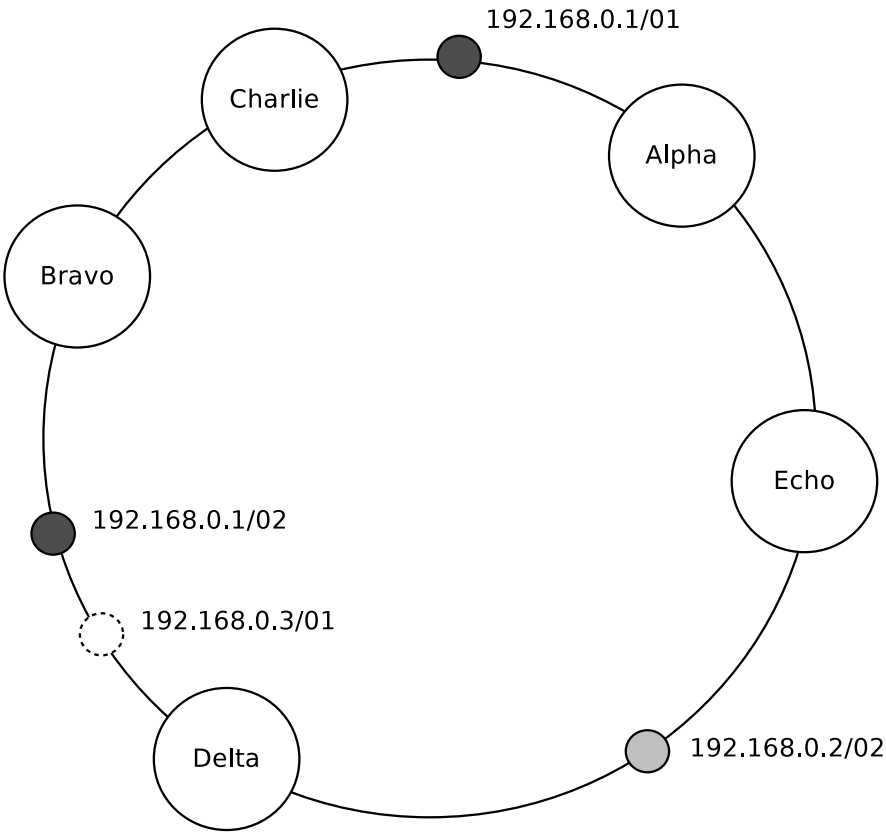

图 4.1　一致性哈希算法。最大的圆环是哈希环。在哈希环上,标有字母的大圆代表数据,上面的单词是该数据的索引,图中画出了五个数据。哈希环上的小圆代表虚拟结点,旁边标注的字符串是该虚拟结点的标识,同样灰度的小圆对应相同的物理结点。系统稳定时有两个物理结点,IP 地址为 192.168.0.1 的物理结点配置了两个虚拟结点,用深灰色小圆表示;IP 地址为 192.168.0.2 的物理结点配置了一个虚拟结点,用浅灰色小圆表示。对于任何一个数据,其索引的哈希值在哈希环上有唯一的一点与之对应。从该点开始,顺着哈希环沿逆时针方向找到 M 个虚拟结点,这 M 个结点对应的 N 个不同物理结点即存放该数据的存储服务器,N 为数据在系统中的副本数。当系统中有新的结点加入,如虚线小圆表示的 IP 地址为 192.168.0.3 的存储服务器,假设 N 等于 1,需要迁移的数据只有那些索引的哈希值落到从浅灰色虚拟结点到新加入虚拟结点之间的劣弧上的数据,这些数据将从深灰色虚拟结点对应的物理结点上移动到新加入的物理结点上,即索引值为 Delta 的数据将从 IP 地址为 192.168.0.1 的存储结点上移动到新加入的 IP 地址为 192.168.0.3 的存储结点上。除此之外,索引的哈希值落在从新加入的虚拟结点到浅灰色虚拟结点的优弧上的所有数据都不必做出移动。期望的数据移动量为系统中数据总量的 (C + 1) 分之一,C 为结点加入前集群的规模,这也是在保证系统存储负载平衡的前提下,任何算法所能达到的数据迁移量下界。此外,一致性哈希算法保证了集群中存在结点动态变化时,变化前后系统的存储负载都是均衡的。

小圆对应相同的物理结点。那么如何决定数据的分配和备份呢？一致性哈希算法规定,对于任何一个数据,将哈希函数作用在它的索引之上,得到的哈希值在哈希环上有唯一一点与之对应。从该点开始,顺着哈希环沿逆时针方向找到前M个虚拟结点。如果这M个虚拟结点对应N个不同物理结点(N为数据在整个系统中的副本数,并且显然N不大于M),那么该数据就应当存放在这N个物理结点上。在图4.1中,假设N等于1,那么索引为Alpha的数据应当存放在IP地址为192.168.0.2的物理结点上,索引为Bravo的数据应当存放在IP地址为192.168.0.1的物理结点上;如果N等于2,那么图中画出的所有数据都应当在两台服务器上各存放一个副本。

另一种决定数据在存储服务器之间分配的方法,在实现上比一致性哈希算法简单。首先对集群中的C台机器从0到C-1依次编号。对于每一个数据,计算其索引的哈希值。由于数据在计算机中的最终表示是二进制,所以哈希值也可以当成一个整数来处理。将这个整数除以C,得到的余数即为N等于1时,应当存储该数据的服务器的编号。对于N大于1的情形,只需要找到编号不小于该余数的连续N台服务器即可。[①] 这种除留余数法与一致性哈希算法相比,思路更加直观,实现也方便,对于集群结点固定的系统,不失为一种理想的解决方案。

但是在大型的数据中心里,集群规模一般为几千台甚至更多的服务器,结点故障不是偶尔发生,而是可能一直存在。这要求有一种算法能够应对结点的动态加入和移出,将这种变化造成的数据迁移量降至最低。对于除留余数法,假设集群中总数据量为D,当有一个新存储结点加入原本规模为C的集群,那么需要迁移的数据量由公式4-1计算得:

$$(\frac{D}{C}-\frac{D}{C-1})*(1+2+\cdots+(C-1))=\frac{D}{2} \tag{4-1}$$

也就是说,当有一个结点动态加入集群时,平均有一半的数据要从原存储服务器移动到另一台服务器。结点移出集群是加入的逆过程,所以当有一个结点因为故障而不再正常运转时,平均有一半的数据将发生转移。当结点故障异常频繁时,这种数据迁移规模将耗费大量的网络带宽和硬盘读写资源,对分布式存储系统将是一个极大的负担。

再回到一致性哈希算法。假设系统稳定后有一个新的存储服务器加入了集群,并且它设置了一个虚拟结点,如图4.1中虚线小圆所示。当N等于1时,需要

① 如果超过C-1则从0开始数

迁移的数据只有那些索引的哈希值落到从浅灰色虚拟结点到新加入虚拟结点之间的劣弧上的数据,这些数据将从深灰色虚拟结点对应的物理结点上移动到新加入的物理结点上,即索引值为 Delta 的数据将从 IP 地址为 192.168.0.1 的存储结点上移动到新加入的 IP 地址为 192.168.0.3 的存储结点上。除此之外,索引的哈希值落在从新加入的虚拟结点到浅灰色虚拟结点的优弧上的所有数据都不必做出移动。当 N 大于 1 时,情况与之类似,请读者以图 4.1为例自行判断哪些数据应该做出移动。那么系统中需要移动的数据总量是多少呢? 考虑结点加入的逆过程,即有一个结点离开集群,需要移动的所有数据就是这个结点原先负责存储的所有数据。由于每个物理结点对应很多的虚拟结点,而这些虚拟结点又是随机分布的,因此在结点离开集群之前,数据在集群中的分布是近似平均的。也就是说,规模为 C 的集群,一个结点动态离开带来的数据迁移量是 $\frac{D}{C}$,一个结点动态加入造成的数据迁移量是 $\frac{D}{C+1}$。显然这两个数值是在保证系统存储负载均衡的性质下,数据迁移量的下界。因而当结点动态移入和移出系统时,一致性哈希算法在数据迁移量方面是最优的算法,并且达到了理论下界。一致性哈希算法的另一个优点是在集群中结点动态变化时,它仍然保持了系统存储负载均衡的性质。前面已经分析过,当系统稳定时,一致性哈希算法保证了存储负载均衡。当某个物理结点离开集群后,原先那些由它某个虚拟结点负责的数据,将被这个虚拟结点按顺时针方向看的下一个虚拟结点所承担。由于物理结点的所有虚拟结点是随机分布的,那么当一个物理结点对应足够多的虚拟结点时,这些被选中的虚拟结点将以相同的数目对应全部的物理结点。换句话说,原先由那个离开的物理结点存储的数据将被平均分成若干份,集群中其余的每个物理结点将得到相同的份数,所以得到的数据总量也相同。因此,一致性哈希算法保证了集群中存在结点动态变化时,变化前后系统的存储负载都是均衡的。

分布式二级哈希表的配置服务器实现了一致性哈希算法。开源项目 libconhash 是一个用标准 C 语言写就的,能在 Windows 和 Linux 平台下运行的一致性哈希算法实现。但是它不支持在系统中进行数据备份,并且其中一些功能的实现策略不适合分布式二级哈希表。因此我在原工程的框架下重写了部分代码,添加了诸多功能,比如使得系统可以支持数据备份,最终将其成功移植到分布式二级哈希表系统中来。

在分布式二级哈希表中,由于系统设计不考虑结点动态加入和离开的情形,因此配置服务器通过读取配置文件获取集群中所有存储服务器的主机名或 IP 地

址。由于集群中各存储结点是同构的,因此直接设定每个物理结点对应 1024 个虚拟结点。哈希函数选用 128 位 MD5 哈希[8],保证了虚拟结点在哈希环上的随机分布性质。

实现一致性哈希的另一个关键点是如何在哈希环上快速找到顺时针方向下一个虚拟结点。由于 C 语言的标准库中不具有像 Java 语言中 HashMap 那样的数据结构可以直接调用实现此功能的接口函数,在分布式二级哈希表中,这个问题通过自行实现了一棵红黑树 ① 来解决。红黑树是一种平衡二叉搜索树,它具有以下性质:

1. 结点被涂以红色或者黑色。
2. 根结点被涂以黑色。
3. 所有叶结点被涂以黑色。
4. 所有红色结点的左右子结点都被涂以黑色。
5. 给定一个结点,从这个结点开始到它任何一个后代叶结点的所有路径包含相同数目的黑色结点。

这些性质保证了红黑树是一棵基本平衡的二叉树,在它上面进行搜索的最差时间复杂度为 O(log(N)),其中 N 为红黑树中结点的个数。在分布式二级哈希表中,所有的虚拟结点都有一个在红黑树中的叶子结点与之对应,排序依据就是这个虚拟结点的标识字符串的 MD5 值。当我们需要查看一个某个数据存储在哪些物理结点上时,就把这个数据的索引取 MD5 值,再拿这个值在红黑树中去搜索,找到第一个哈希值不小于这个值的虚拟结点,再从这个虚拟结点的哈希值开始,往后找 N - 1 个哈希值严格增大,并且对应不同物理结点的虚拟结点。如果在这个过程中需要找比某个值大的虚拟结点,但是这个值比红黑树中最大的哈希值还大,那么直接返回哈希值最小的那个虚拟结点。给定索引确定数据所在存储服务器的 IP 地址的函数伪代码如下:

```
FUNC getIpsByKey(IN string key, IN int N, OUT IP[] ips)
{
  GLOBE RBTREE rbtree;
  VAR VNODE vnode, start;
```

① http://en.wikipedia.org/wiki/Red-black_tree

```
    vnode = rbtree.geq( MD5(key) );
    ips[0] = vnode.node.ip;
    start = vnode;
    for i from 1 to (N - 1)
    {
1:    vnode = rbtree.ge( vnode.hash );
      if ( vnode == start ) ERROR;
      for j from 0 to i - 1
      {
        if ( ips[j] == vnode.node.ip ) goto 1;
      }
      ips[i] = vnode.node.ip;
    }
  }
```

前面我们说到在红黑树中搜索的时间复杂度为 O(log(N)),其中 N 为树中结点总个数。在分布式二级哈希表的一致性哈希实现中,红黑树中的结点数就是系统中虚拟结点的总数。在第2.3节中的一个假设是系统的典型规模为 50 台存储服务器,按照上文提到的每个物理结点配置 1024 个虚拟结点计算,一次查询需要花费的时间规模与处理器进行 16 次运算相当,因此用红黑树实现一致性哈希具有很高的查询效率。

此外,之前在原理介绍中提到的数据索引,在分布式二级哈希表的实现中,指的是 blob 的 bucketID。也就是说,同一个桶中的 blob 将存储在相同的服务器上。一致性哈希算法对于数据的分配粒度是相对于桶的,它不能区分同一个桶中不同的 blob。在这种情况下,集群的存储负载平衡是由另一个假设保证的。即我们假设单个 blob 较小,但是 blob 和桶的个数很多。如果我们将桶中所有 blob 的大小求和作为桶的大小,那么虽然桶的大小可能差异很大,但是由于每个物理结点都存储了很多的桶,而且这些桶是随机分配的,那么当桶的数目足够多时,每个物理结点的总存储量是近似相同的。

4.1.2 远程过程调用

除了一致性哈希算法,配置服务还包含另一个重要模块,即远程过程调用模块。在分布式二级哈希表中,客户端通过远程过程调用从配置服务器获取某个数据所在的目标存储服务器的 IP 地址。

远程过程调用是一种客户端和服务器交互的技术,它使得客户端程序可以调用执行在远端服务器上的一段程序,而调用方法就好像在执行本地的函数一样,不必关心客户端和服务器的网络交互,以及其他实现细节。分布式二级哈希表采用 ONC RPC [①] 标准规定的远程过程调用协议,代码执行流如图4.2所示。客户端执行 clnt_call 系统调用,并将参数按照本地函数调用的方式压栈。clnt_call 从中提取出目标服务器例程的信息,并将该例程需要的参数按照事先定义的方式打包,然后通过 TCP 或 UDP 协议将参数发送至指定服务器。目标服务器事先已经注册了一段例程,并监听 ONC RPC 使用的默认端口 111。一旦收到客户端发来的打包数据,按照事先定义的方式从中提取出参数,并传递给服务器例程。接着服务器例程在用户态执行,执行完毕后,进入内核态,由操作系统将返回值按照事先约定的方式打包,再依照相同的传输层协议,通过刚才建立的连接,将打包后的返回值传回客户端。至此,此次远程过程调用服务器端的工作完成,服务器端准备应答下一个请求。客户端收到服务器发送来的打包数据,按照事先约定的方式从中提取出返回值。然后 clnt_call 系统调用返回,将远程过程调用的返回值传递给用户态的客户端主程序,本次远程过程调用执行完毕。

远程过程调用这种客户端和服务器的交互机制,简化了程序员的编程,其中,参数和返回值的打包,数据在客户端和服务器之间的可靠性传输等工作均由系统来实现。但是,上述代码执行流程对于纯粹写应用的程序员仍然略显复杂,比如,程序员需要告知系统如何对数据进行打包和提取,需要初始化网络参数等。为了进一步简化程序员编程,使程序员将主要精力放在将要实现的功能上,而不是远程过程调用本身的实现上,编程工具 rpcgen[②] 应运而生。程序员只需要书写一个简单的配置文件,声明服务器例程的原型,以及各参数和返回值的数据结构,rpcgen 就能据此产生出服务器和客户端将要用到的全部 C 代码。程序员只需要在服务器端例程的框架里填入具体实现,再将客户端代码和服务器端代码分别编译,便实现了远程过程调用。比如,在分布式二级哈希表中,客户端通

① http://en.wikipedia.org/wiki/ONC_RPC
② http://en.wikipedia.org/wiki/Rpcgen

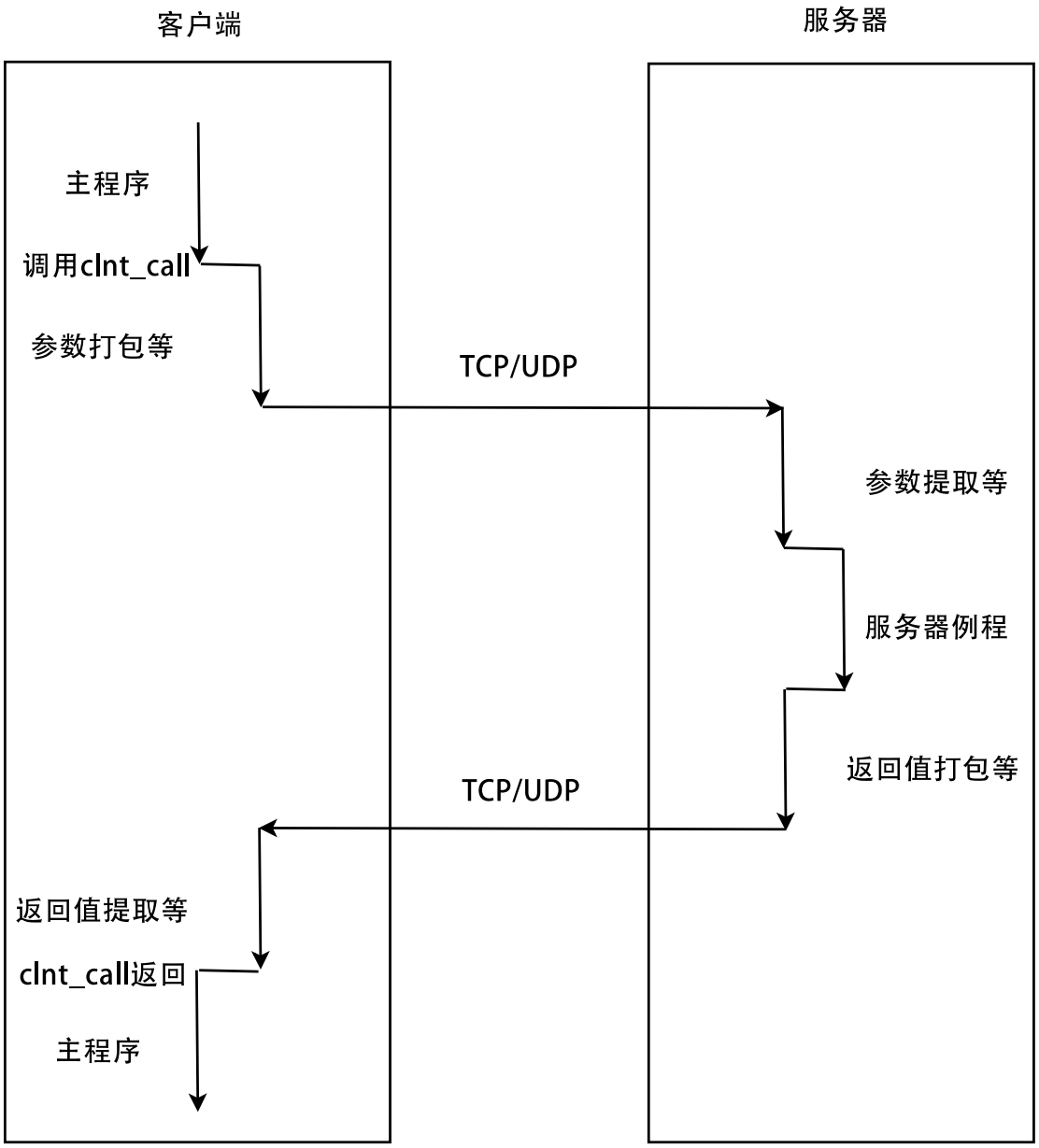

图 4.2　ONC RPC 代码执行流。客户端用户态主程序执行 clnt_call 系统调用。clnt_call 将服务器目标例程需要的参数按照事先定义的方式打包,然后通过 TCP 或 UDP 协议将打包数据发送至指定服务器。目标服务器事先已经注册了一段例程,并监听 ONC RPC 使用的默认端口 111。收到客户端发来的打包数据后按照事先定义的方式从中提取出参数,并传递给用户态服务器例程。服务器例程执行完毕后,进入内核态,由操作系统将返回值按照事先约定的方式打包,再通过刚才建立的连接将打包后的返回值传回客户端。至此,此次远程过程调用服务器端工作完成,端准备应答下一个请求。客户端收到服务器发送来的数据,按照事先约定的方式从中提取返回值。然后 clnt_call 返回,将远程过程调用的返回值传递给用户态的客户端主程序,本次远程过程调用执行完毕。

过远程过程调用,给定一个索引,从配置服务器得到具备这个索引的数据所在的存储服务器的 IP 地址。那么 rpcgen 的配置文件 cfgsrv.x 内容非常简单:

```
typedef u_int ips<>;

program CFGSRVPROG {
  version CFGSRVVERS {
    ips GET_HOSTS_BY_KEY(string) = 1;
  } = 1;
} = 1;
```

其中指定了服务器端例程的原型。参数只有一个,即字符串类型的索引值。返回值是无符号类型整数的可变长度数组,其中每个元素是一个目标存储服务器的 IP 地址。rpcgen 据此生成了四个文件:

1. cfgsrv.h: 基本的声明文件。编译客户端和服务器端程序时都需要引入此头文件。
2. cfgsvr_clnt.c: 客户端辅助文件,以实现远程过程调用机制,其中只定义了一个函数 get_hosts_by_key_1。客户端主程序调用这个本地函数,就能从配置服务器获取目标存储服务器的 IP 地址。
3. cfgsrv_svc.c: 服务器端辅助文件,以实现远程过程调用机制。
4. cfgsrv_xdr.c: 服务器端例程的参数和返回值打包程序。

为了完成远程过程调用机制,服务器端还需要实现 cfgsrv.h 中声明的 get_hosts_by_key_1 和 get_hosts_by_key_1_svc 两个函数。其中第一个函数是服务器端真正完成通过索引获取目标存储服务器 IP 地址的例程,可以调用第4.1.1小节介绍的一致性哈希算法的实现函数。第二个函数只是对第一个函数的包装。之后通过 rpcgen -m 命令可以得到服务器端初始化远程过程调用的代码。至此,将上述文件合理编译,即可得到实现了远程过程调用的服务器和客户端模块。

远程过程调用简化了客户端和服务器端实现通信的机制。如果不使用远程过程调用,一种实现方式是直接在客户端和服务器端建立 TCP/UDP 连接,再按照自定义的协议传送数据。在第2.3实验假设中提到分布式二级哈希表平均每台服务器每秒处理的请求数不小于 50,平均响应时间为百毫秒数量级,这说明客户

端一直会发起大量的请求。如果每一个请求完成之后都关闭 socket 连接,下个请求到来时再重新建立连接,那么创建 socket 和回收资源的系统开销将会非常大,严重影响系统运行性能。这要求每个客户端和配置服务器的 socket 连接一直持续,直到客户端程序全部运行结束。基于这个前提,由于要保证系统的平均响应时间和吞吐量,请求必须设定为异步的,而不能像同步系统那样将请求放置在一个队列里,在上一个请求结束之后再处理下一个请求。因此,请求的返回顺序可能和发起顺序不同,客户端可能先收到后发出的请求的返回值,再收到先发出的请求的返回值。在同一个 socket 连接中,客户端需要为每个请求附上一个唯一的流水号,服务器则要为返回值贴上相同的流水号。另一方面,在编程实现上,异步请求也较同步请求更为复杂,容易出错且不易调试。如果维护返回值和请求的对应关系,以及请求的异步化都由远程过程调用的框架来完成,那么对于程序员来说,客户端的每次请求就是对本地函数的一次调用,不必过多考虑上述实现细节。

4.2 客户端

在第3章中曾介绍过,分布式二级哈希表中的客户端是一个独立的模块,是分布式二级哈希表的一个函数库,上层应用需要将这些代码编译到可执行文件中去。客户端直接从配置服务器拿到目标存储服务器的 IP 地址,再与这些服务器进行通信,执行操作,并将返回值传递给上层应用。这样,完成一次操作总共需要进行两次网络通信。第一次是客户端通过远程过程调用从配置服务器拿到路由信息,第二次是客户端将请求发送给具体的存储结点,得到返回值。这种决策的缺点是上层应用需要嵌入分布式二级哈希表的标准 C 代码,降低了分布式二级哈希表的易用性和通用性。另一种不需要嵌入系统框架代码的解决方案是,让客户端直接与某一台入口服务器按照某种协议进行通信,再由这台入口服务器完成客户端的功能。这样入口服务器和客户端之间还要多进行一次网络通信,势必增加请求的平均响应时间,并且消耗网络带宽。由于分布式二级哈希表更在意系统的运行效率,而且上层应用是由我们自己搭建的,因此最终我选用了第一种方案来实现客户端。

客户端采用了分层结构,如图4.3所示。最上层是接口层,对分布式二进制哈希表提供的各项功能做出了最外层的包装,为上层应用提供调用接口。接口层通过调用路由层的相应函数,将上层应用的请求向下面传递。路由层收到请求后,

首先通过远程过程调用从配置服务器得到数据所在的存储服务器的 IP 地址,然后按照 Redis 规定的交互协议构建命令闭包,将其丢进线程池的任务队列中。最后在规定的时限内检查该命令闭包的执行结果,并将返回值传递给上层函数。为了提高系统的性能,我在路由层的下面实现了一个线程池。该线程池有一个任务队列,其中的元素是 Redis 命令闭包,包含一条 Redis 命令的全部信息。线程池会自动执行队列中的任务,并将返回值填入命令闭包。这样路由层就可以方便的将上层应用的请求转化为 Redis 命令,将构建好的命令闭包丢进线程池的任务队列。在线程池之下是通信层,主要实现了 Redis 规定的交互协议,与存储服务器上 Redis 服务器进程进行交互,执行各种命令并获取返回值。总体来说,除了接口层,分布式二级哈希表中的每一层都通过调用下层的某一函数,实现请求逐级向下传递。而下层执行完该函数之后,通过函数返回将结果数据传递回上一层。横向来看,路由层和配置服务器之间有一次网络通信,获取数据所在存储服务器的 IP 地址;通信层和存储服务器之间也有一次网络通信,执行真正的 Redis 命令并取得返回值。两层之间的线程池实现了请求的异步化,充分利用了网络带宽,大大提高了系统的运行效率。下面我将详细介绍分布式二级哈希表客户端的各个子模块。

4.2.1 接口层

接口层对分布式二级哈希表实现的各项功能做了最外层的包装,将其呈现给上层应用。在接口层的头文件 api.h 里声明了可供上层应用调用的所有函数,具体功能详见表 2.1。数据的返回是通过上层应用在调用接口函数时,传入指针实现的。调用者负责为该指针申请足够的空间,接口层则会在从该指针所指地址开始填入返回的数据,如果该函数需要返回数据的话。每个函数的返回值则标识了该请求是否成功执行。在上层应用调用任何接口函数之前,还需要调用接口层的初始化函数。初始化函数将该层需要用到的模块和全局变量等资源进行初始化,并调用下层的初始化函数。与此类似,客户端所有请求执行完成后,调用接口层的终结函数。

4.2.2 路由层

回顾一下,在第3章中我说明了为什么同一个数据在分布式二进制哈希表中要存储多份,以及不同副本之间的数据保证了弱一致性,即不同副本之间的数据

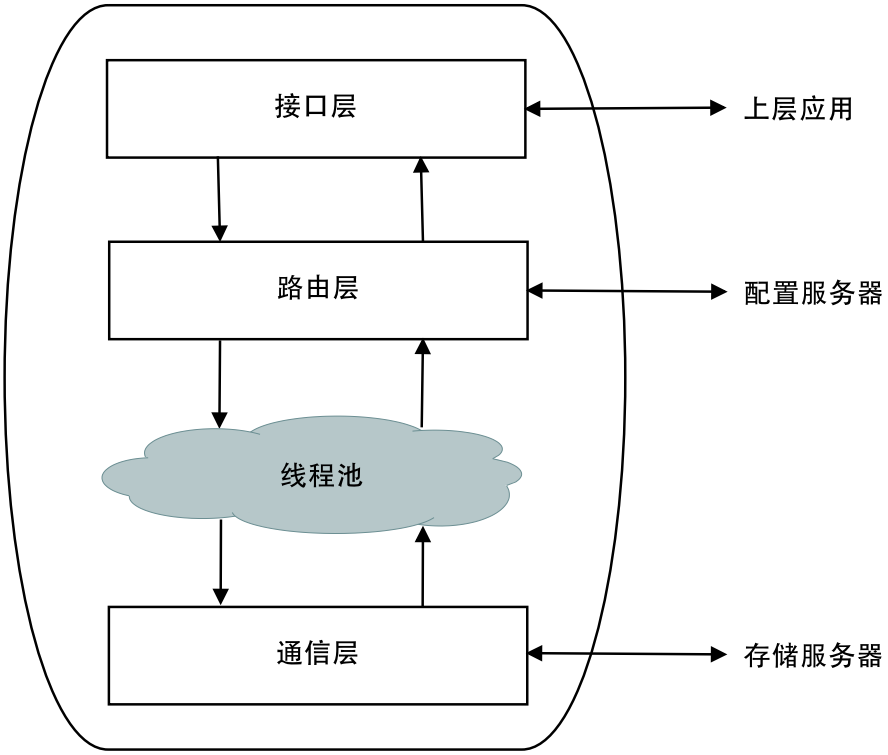

图 4.3　分布式二级哈希表客户端分层结构。最上层是接口层,对分布式二进制哈希表提供的各项功能做出了最外层的包装,为上层应用提供调用接口。接口层之下是路由层,通过远程过程调用从配置服务器得到数据所在的存储服务器的 IP 地址,然后按照 Redis 规定的交互协议构建命令闭包,将其丢进线程池的任务队列,再异步检查其执行结果。线程池会自动执行任务队列中的命令,并将返回值填回命令闭包。最下层的通信层按照 Redis 规定的协议与存储服务器上的 Redis 服务器进程进行通信,将指令发送给存储服务器并取得返回值。纵向来看,请求和参数由上而下逐级传递,返回值则由下而上逐级返回。横向来看,路由层从配置服务器获取存储服务器的 IP 地址有一次网络通信,通信层和目标存储服务器之间还有一次网络通信。

并不是时刻保证相同的,但读操作总能读取到对该数据的最近一次写操作结果。在分布式二级哈希表中,数据的弱一致性通过$R + W > N$ 语义实现。

4.2.2.1　$R + W > N$ 语义

$R + W > N$ 语义是分布式系统中保证数据弱一致性的经典算法之一,它指的是如下算法。在数据有 N 个副本的分布式系统中,每个读写操作都要对 N 个副本同时进行。对于写操作,只要在至少 R 个副本上成功写入了,即认为该写操作成功了。对于读操作,只要读出了至少 W 个副本上的数据,那么其中至少有一份数据是最新写入的数据。其中,R 和 W 的数值人为设定,但必须满足二者之和大

于 N。原理的正确性是显然的。

相比与维护数据的强一致性，$R + W > N$ 语义的优点如下：

1. $R + W > N$ 语义提高了系统的操作成功率，降低了响应延迟，大大提高了系统性能。强一致性系统中，如果由于网络阻塞或者其他原因，在一个副本上写操作执行失败了，那么整个写操作就执行失败了。即便能够执行成功，这次写操作的响应时间也取决于 N 个操作中最慢的那个。而在 $R + W > N$ 语义中，写操作取决于这 N 个操作中第 R 快的那个操作。由于 R 小于 N，平均响应时间必定减小。另一方面，由于只要在 R 副本上成功写入了数据，即宣告该写操作成功执行了，系统的操作成功率必定增加。
2. $R + W > N$ 语义可以针对不同的应用需求，对系统的性能作出调整。R 越大，系统对于读操作的响应时间越小，数据的一致性越强；W 越小，系统对于写操作的响应时间越小。
3. $R + W > N$ 语义可以简化系统实现。在强一致性系统中，如果某个操作失败，需要将系统恢复到这 N 个子操作进行之前的状态。也就是说，即便 N 个写入操作中的某些成功了，也要将数据恢复到写入之前的数值。这将大大增加系统的实现难度，降低系统的运行效率。[7] 而在弱一致性系统里，操作的成功率很高，因此读操作总能读到最新写入的数据。

回忆第2.3节，我们要求系统在饱和运转时，操作成功率大于 99.999%，而平均响应时间在百毫秒数量级，如此严苛的条件必然要通过采用 $R + W > N$ 语义来满足。

在分布式二级哈希表中，路由层针对每一个请求，将其转化为一条 Redis 命令，构建出 N 个命令闭包后丢入线程池的任务队列。线程池对任务的执行是异步的，路由层需要在一定的时限内去检查执行情况，并将最终结果返回给接口层。关于命令闭包、线程池和异步结果提取将在稍后第4.2.3小节详细介绍。

与强一致性分布式存储系统相比，$R + W > N$ 语义带来了系统性能的显著提升，但是延长了读操作的平均响应时间。原先 N 个读操作只要有一个成功执行了，即可以直接返回其结果，现在不仅要等 R 个结果，而且由于数据是弱一致性的，系统只能保证这 R 个结果中至少有一个是最新版本的数据，但可能不都是最新版本的数据。这要求系统能够通过某种机制从中挑选出最新版本的数据，这一过程称为版本冲突解决。

4.2.2.2 版本冲突解决

在仅保证数据弱一致性分布式系统中，$R + W > N$ 语义确保读到的 R 个版本的数据中有一份是最新写入的，但这 R 份数据可能不全是最新的版本。分布式系统领域里一般通过如下方式解决这个问题：在写入时为数据添加版本信息，在读数据时根据版本信息判断它们的时序关系，选取最新的数据。添加版本信息的最常用方法是向量时钟[9]，比如在 Dynamo[4] 的实现中就使用了这种方法。但是向量时钟不能解决所有的版本冲突问题，当有无法解决的冲突时，Dynamo 将所有版本的数据返回给上层应用，由他们自己来解决。

分布式二级哈希表的设计不允许有多个版本的数据反馈给上层应用。由于版本冲突是异步分布式系统的固有性质，不可能通过任何算法彻底解决。而且系统存储的是没有任何子结构的二进制数据块，不可能有任何智能的算法将冲突的数据综合出一个新的版本。因此当有不可解决的数据冲突发生时，分布式二级哈希表必然只能从中随机选取一个版本的数据作为返回值。另一方面，尽管为了降低写操作的响应时间，N 个副本中只要有 W 个成功写入了数据，该操作就成功返回了，但是在不发生错误的情况下，这 N 个副本上的数据最终都会被更新。此外，如果采用向量时钟标记数据版本，那么在执行写入操作时，必然要附加一次读取操作取得数据的版本信息，这必然会增加写入操作的平均响应时间。第 2.3节提到，容许个别情况下读到不是最近一次写入的数据。综合以上考虑，为了简化系统设计和实现，最终我决定直接从读到的 R 个副本中随机选取一份数据作为返回值。为了减少这种策略可能带来的风险，对于 blob 的读写操作，设定 $(N, W, R) = (3, 2, 2)$，这是主流分布式存储系统的一般配置方法；对于桶和 blob 的删除操作，以及检测指定桶和 blob 是否存在的操作，设定 $(N, W, R) = (3, 3, 1)$，即删除操作要求在所有副本上全部成功执行，而只要有一个副本上还残留有数据，即认为该数据存在。由于删除操作和检测操作的出现频率读写操作低很多，这种配置在对系统性能造成可以忽略的影响的前提下，很大程度上提高了操作的语义正确性。

4.2.3 线程池

$R + W > N$ 语义通过减小写操作的响应时间来提高系统的性能，具体来说，由原本需要等待 N 个副本上操作全部完成，改进为只需要等待至少 R 个副本上成功写入了数据即可。系统满足这个性质的前提是，在 N 个副本上的操作是并

行执行的,其中每个操作包括客户端将命令请求发送至存储服务器,存储服务器在本地执行相应操作,存储服务器将操作的返回值发送回客户端。为了实现操作的并行执行,必然要用到多线程或者多进程技术。进程间通信需要用到特定的技术[①],相比起来,同一进程下的线程共享内存地址空间、文件描述符等系统资源,不同线程之间交换数据几乎不用引入额外的系统开销。另一方面,线程比进程更轻量级,其创建和资源回收比线程的开销要小很多。综合以上两点考虑,我最终选择多线程技术实现操作的并行化。在第2.3节中介绍过,在系统饱和运转的情况下,单位时间内客户端发起的操作请求数是非常大的。如果针对每个操作都重新创建一个线程,执行完成后再回收资源,那么资源分配和回收的系统开销仍将是巨大的,这不符合分布式二级哈希表要尽量保证系统运行效率的根本设计原则。为了进一步提高系统运行效率,我实现了线程池。[②]

在分布式二级哈希表中,客户端实现的线程池结构如图4.4。在线程池初始化的时候,会创建若干个工作线程,起初它们都处于空闲状态。线程池还有一个任务队列,其中的元素是由命令闭包构建的任务。路由层构建好命令闭包后,会将其抛入任务队列。一旦任务队列非空,而且至少有一个工作线程处于空闲状态,该任务将被一个工作线程执行,并且从任务队列中移除。该任务被执行完成后,工作线程重新进入空闲状态,准备执行下一个任务。完成的任务所包含的命令闭包中已经填充上了返回值,它们被放置在一个虚拟的结果队列里。这时路由层会在一定时限内异步检测任务的执行情况,如果操作成功执行,路由层从中提取结果后返回。

由于工作线程是在线程池初始化的时候创建的,因此当系统饱和运转时,不会有线程创建和回收资源的开销。不过 C 语言的标准库中没有线程池的实现,因此我自己利用互斥锁和条件变量实现了一个高效的线程池。

```
typedef struct ThreadPool {
  struct ThreadTask *taskQueue;
  pthread_mutex_t *lock;
  pthread_cond_t *cond;
  pthread_t *threads;
} ThreadPool;
```

① http://en.wikipedia.org/wiki/Inter-process_communication
② http://en.wikipedia.org/wiki/Thread_pool

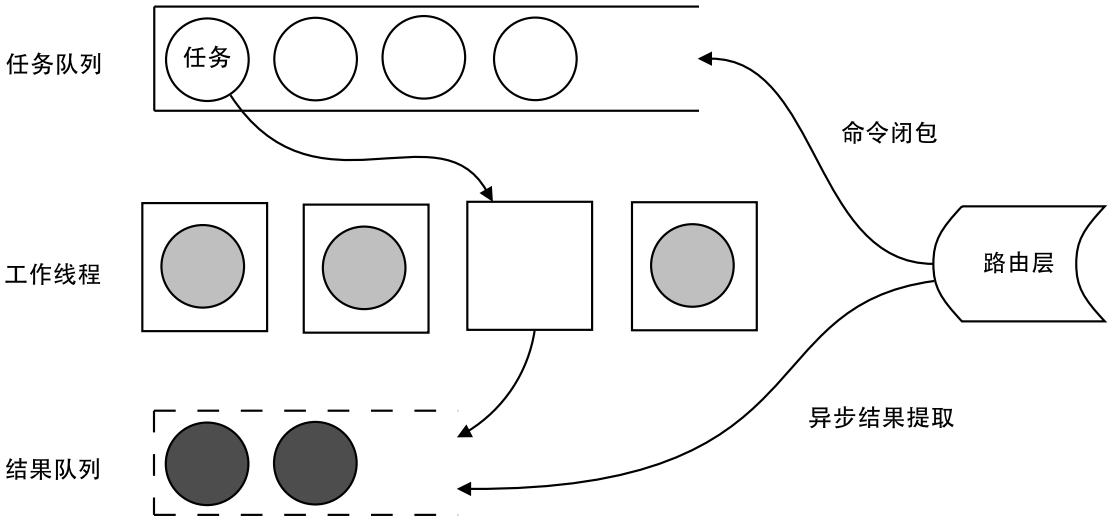

图 4.4　分布式二级哈希表客户端线程池工作原理。线程池将路由层丢入的命令闭包包装成任务丢入任务队列。空闲的工作线程自动执行排在队列中的任务,并将其从任务队列中移出。执行完毕后,该工作线程重新进入空闲状态准备执行下个任务。路由层会在一定时限内异步检测命令闭包的执行情况,并从中提取结果。

其中,taskQueue 是一个用链表实现的任务队列。当路由层请求执行某个命令闭包时,线程池相应的会创建 N 个任务,并把它们加入到任务队列的末尾。lock 是线程同步互斥锁,用于保证对线程池结构体任何操作的原子性。cond 是线程条件变量,它与 lock 一起,用于通知空闲的工作线程任务队列里有等待被执行的任务,实现任务对工作线程的分配。threads 是工作线程,这些线程在线程池初始化的时候创建,之后每个线程进入忙等[①]循环。在这个循环的一开始,该工作线程阻塞在 pthread_cond_wait,进入休眠状态并等待条件变量被触发。起初任务队列为空,一旦有新的任务加入,线程池会触发条件变量。这个时候,某个处于休眠状态的工作线程被唤醒,将该任务从任务队列中移出,然后判断这时任务队列中是否还有等待被执行的任务,如果队列不为空,它再触发一次条件变量,最后执行刚刚从任务队列中取出的任务。线程池实现了任务的异步执行,保证了系统的运行效率,主要体现在:

1. 所有工作线程在线程池初始化的时候被创建。当系统稳定运行之后,不存在因为线程创建和资源回收带来的开销,系统性能不会受到影响。
2. 空闲的工作线程取得某个等待被执行的任务之后:
   (a) 将其从任务队列中移除。

① http://en.wikipedia.org/wiki/Busy_waiting

(b) 判断此时任务队列是否为空,如果仍然有任务等待被执行,再次触发条件变量。

(c) 执行2a中从任务队列移除的任务。

这三步必须按顺序执行。2a在2b之前保证了所有任务都会被执行,而且同一个任务只能被一个工作线程执行。2b在2c之前则使得任务可以被并行执行,保证了系统的运行效率。

4.2.3.1 命令闭包

在分布式二级哈希表中,命令闭包是这样一个结构体,它包含一个操作的全部信息。

```
typedef struct RedisCommand {
  int (*command) (void *);
  char *key;
  char *field;
  void *value;
  int successNum;
  int exist;
  void *blob;
  pthread_mutex_t *lock;
  int refNum;
} RedisCommand;
```

其中,command 函数指针是将要被工作线程执行的例程,上层应用调用接口层的某一函数时,路由层根据其语义选择一条 Redis 命令来实现,即为该函数指针所指向的函数。key, field, value 三个指针分别指向对应数据的两级索引和内容,如果不涉及则为 NULL。由于该命令闭包对应的操作将要在 N 个副本上执行,因此 successNum 用于存放成功操作的计数。exist 专门用于表示桶或 blob 是否存在。假设每个 blob 的最大长度为 L 字节,对于读取 blob 操作,blob 指针用于存放返回值,它指向一块大小为 $N * L$ 的已经申请的内存空间,第 i 个完成该操作的工作线程将读取的结果填放在起始地址为 $blob + (i - 1) * L$ 的内存空间中。lock 是该命令闭包的同步互斥锁,用于保证计数等操作正确执行。

4.2.3.2 任务

在分布式二级哈希表中,任务是这样一个结构体,它是任务队列中的元素,可以被任何一个工作线程执行。

```
typedef struct ThreadTask {
  RedisCommand *redisCommand;
  struct in_addr *ip;
  struct ThreadTask *next;
} ThreadTask;
```

对于上层应用的每个请求,路由层都会创建一个命令闭包,该操作将分别在 N 个副本上并行执行,因此线程池将为每个命令闭包创建 N 个任务。这 N 个任务的 redisCommand 指针都指向这个命令闭包。同时,路由层从配置服务器获取到的 N 个目标存储服务器的 IP 地址也将被分配到这些任务中去。这样,每个任务结构体就包含了执行任何一个请求所需要的全部信息。

4.2.3.3 异步结果提取

线程池实现了操作的异步化,也使得路由层需要定期查看命令闭包的执行情况。路由层会在一定时限内以一定频率不断检查命令闭包的 successNum 域,如果按照 $R + W > N$ 语义达到了指定的数值,该操作被认为执行成功。如果有返回值,则按照上文提到的方法提取返回值。由于结果提取可能在 N 个操作都成功执行之后,也可能在没有全部执行之后,因此需要某种机制来保证命令闭包在结果提取之后再进行资源回收。在分布式二级哈希表中,这通过命令闭包结构体的引用计数域 refNum 实现。该域在命令闭包创建时初始化为 1。之后线程池将创建 N 个任务结构体,每当有一个任务指向该命令闭包,这个命令闭包的 refNum 域加 1。这样,在这些任务被执行之前,它们所包含的命令闭包的引用计数达到 $N + 1$。之后这 N 个任务并行执行,其间路由层可能异步提取结果。每当一个任务执行完毕,命令闭包的引用计数减 1;当路由层成功提取了结果,命令闭包的引用计数也减 1。一旦命令闭包的引用计数减至 0,该命令闭包结构体的内存资源立即被回收。通过设置命令闭包的引用计数域,保证路由层在异步提取结果之后,再对其占用的内存资源进行回收。

表 4.1　分布式二级哈希表使用的 Redis 命令。客户端的通信层按照 Redis 规定的交互协议实现了这些基本 Redis 命令,与存储服务器上的 Redis 服务器进程交互,传输指令和数据,完成存储服务器上 Redis 二级哈希表的本地操作,以表达分布式二级哈希表的语义。

接口函数	Redis 命令	说明
createBucket	hSet	在本地 Redis 二级哈希表中存入两级索引分别为 bucketID 和 EXIST_BLOB_ID 的数据,内容为 EXIST_BLOB。其中,EXIST_BLOB_ID 为分布式二级哈希表设定的一个特殊字符串,不会与任何实际数据的 blobID 相同。
deleteBucket	del	删除本地 Redis 二级哈希表中一级索引为 bucketID 的所有数据。
existBucket	exists	查看本地 Redis 二级哈希表中一级索引为 bucketID 的数据是否存在。
deleteBlob	hDel	删除本地 Redis 二级哈希表中两级索引分别为 bucketID 和 blobID 的数据。
existBlob	hExists	查看本地 Redis 二级哈希表中两级索引分别为 bucketID 和 blobID 的数据是否存在。
loadBlob	hGet	读取本地 Redis 二级哈希表中两级索引分别为 bucketID 和 blobID 的数据。
saveBlob	hSet	在本地 Redis 二级哈希表中存入两级索引分别为 bucketID 和 blobID 的数据,内容为 blob。

4.2.4　通信层

通信层是客户端分层结构的最底层,与存储结点上运行的 Redis 服务器进程交互,传输数据和指令,完成存储服务器上 Redis 二级哈希表的本地操作。通信层按照 Redis 的交互协议 ①,实现了一些基本 Redis 命令 ②,通过这些操作来表达分布式二级哈希表接口函数的语义,其对应关系详见表4.1。

路由层在创建命令闭包的时候,会将其函数指针域指向通信层实现的 Redis 命令,然后被线程池中的工作线程所执行。对于某个 Redis 命令,首先建立到目标存储服务器的 socket 连接,然后按照 Redis 规定的协议向该服务器发送指令和

① http://redis.io/topics/protocol
② http://redis.io/commands

数据,之后在一定的时限之内,异步读取服务器传回的结果。如果超时没有收到结果,则认为在该存储服务器上的操作失败。如果成功读到结果,将其填回命令闭包中,路由层会自动从命令闭包读取操作的结果数据。最后关闭到远端存储服务器的连接。

一次 Redis 命令的执行包括客户端将指令和数据发送至目标存储服务器,服务器在 Redis 二级哈希表上执行本地操作,服务器返回结果。如果出现网络阻塞或者存储服务器负载过重等情况,从客户端发出请求到服务器返回结果可能经过很长的时间,甚至最终没有结果返回。如果客户端采用阻塞方式从 socket 通道读取结果,会严重影响系统性能。为此,客户端采取异步方式读取结果,在预先设定的时限内,不断的去检测是否有数据传回。如果读到结果立即返回;如果超时没有读到结果,则认为此次操作失败。因此,时限的设置尤为关键,时限大则操作成功率较高,但平均响应时间也会增大;反之,时限小则操作成功率和平均响应时间都会降低。由于在当前的系统实现里,时限的设置实在系统运行之前静态确定的,因此需要综合考虑第2.3节中关于系统操作成功率和平均响应时间的要求,具体的数值设置参见第5章。

4.3 存储服务器

存储服务器是分布式二级哈希表存储集群中的单个结点,是一致性哈希算法中的一个物理结点,通过 IP 地址来区分。存储服务器内部的存储称为本地存储,解决的是本机上索引到数据的映射关系。在第3中曾提到过,分布式二级哈希表的本地存储采用 Redis 开源项目。分布式二级哈希表的客户端直接与存储服务器上的 Redis 服务器进程交互,对存储服务器上的数据进行操作,并获取返回值。Redis 是一个增强的哈希表,其数据可以是哈希,也就是说,Redis 实现了单机的二级哈希表。分布式二级哈希表则通过一致性哈希,将数据以桶为基本单位,在各个存储服务器上进行分配和备份,实现了表2.1罗列的调用接口函数。

本地存储是分布式二级哈希表需要解决的问题之一,它在很大程度上决定了系统的吞吐率和平均响应时间。不过,分布式存储系统往往更关心数据是如何在存储集群的结点间进行分配和备份的,而单个结点上的存储方式则使用已经成熟的本地存储技术。这一方面是由于分布式存储系统和其他的分布式系统一样,呈现给用户的是独立的接口。在上层应用看来,它在一个分布式存储系统中操作数据和它在一个单独的存储单元上操作数据,在使用方式上不应该有任何

的差别,只是性能有所不同。分布式存储系统只要解决好数据在系统中多个存储结点上的分配和备份即可,因为组成系统的最小存储单元可能是一台计算机,也可能是另一个子分布式存储系统。另一方面,为了提高系统的可扩展性,增强系统的普适性,不应该对系统中单个结点上的本地存储方式提出过多的要求。随着对分布式存储系统的利用,可用的存储空间越来越小,势必要求系统通过增加存储结点来提高总存储容量。对组成系统的单个结点的限制越少,那么可以加入系统的结点越多,系统的可扩展性越强。例如像 Google File System[2] 那样的分布式文件系统,主要解决的是数据如何在不同存储结点上进行分配和备份的问题,以及用户如何能够找到想要的数据的问题。系统完全没有对单个结点上的文件系统提出任何要求。只要具备可用的本地文件系统,任何一台计算机,甚至任何一个分布式文件系统,都可以作为一个基本的存储单元,加入到 Google File System 集群中去。

在当前的应用假设下,分布式二级哈希表被设计为不支持存储结点的动态加入和移除。但是实现数据分配和备份采用了一致性哈希算法,使得系统具备很强的扩展潜能。选择一个已经很成熟的开源项目 Redis 作为本地存储模块,在保证系统运行效率的同时,也使我将主要精力投放在改进和优化数据分配和备份上,着重解决那些在分布式存储系统中存在,而在本地存储系统中不必考虑的问题。

当然,Redis 本身是一个非常出色的本地增强哈希表,支持二级哈希表的语义。此外,系统在用户态实现了一个虚拟内存层,将全部数据放置在内存或者用户态的虚拟内存中,大大提升了系统的处理速度,降低了操作的平均响应时间。将数据存放在内存中具有一定的风险,因为一旦断电,内存中的数据就会全部丢失且无法恢复。每经过一段固定的时间,或者系统中的数据进行了固定次数的更新操作,Redis 就会将内存和用户态虚拟内存中的数据压缩后写入硬盘。此外,Redis 还通过日志文件机制 ① 将各个操作以日志的方式记录下来,提高数据的恢复速度,并且保证数据不会丢失。受篇幅限制,本论文不对 Redis 做过多讨论,其架构设计和实现细节可以通过阅读 Redis 的源代码 ② 来深入了解。本节只粗略说明 Redis 是如何实现本地二级哈希表存储的。

Redis 是用标准 C 语言写成的。由于 C 语言的标准库没有哈希表这种数据

① http://en.wikipedia.org/wiki/Journaling_file_system
② https://github.com/antirez/redis

表 4.2　Redis 二级哈希表哈希串结构。哈希串是多个子串的顺序连接。所有的长度均以字节为单位。

子串	长度	含义	说明
$< H >$	1	哈希串长度	当数值大于或等于 254 时,该值忽略。
$< L_i >$	1 或 5	第 i 个数据的索引长度	把第一字节当作无符号整数。如果小于或等于 252,此数表示索引长度,该子串长度为 1;如果等于 253,该子串长度为 5,后面 4 字节表示的无符号整数为索引长度;如果等于 254,这个位置可以插入新的数据(原数据可能由于删除操作被清空);如果等于 255,说明这是哈希串的最后一个字节。
$< k_i >$		第 i 个数据的索引	
$< I_i >$	1 或 5	第 i 个数据的长度	同 $< L_i >$。
$< F_i >$	1	$< v_i >$ 之后的无效字节数	可能由于更新操作使 v_i 长度变短。
$< v_i >$		第 i 个数据	

结构,Redis 自己实现了一个哈希表。与分布式二级哈希表的一致性哈希模块相同,Redis 本地哈希表也采用 128 位 MD5 哈希函数。为了解决哈希冲突,Redis 中的桶结构用链表实现,在桶内查找元素直接遍历链表中的元素,复杂度为 O(N),其中 N 为链表长度。在二级哈希表中,数据的类型又是一级哈希,这一级哈希的映射关系用一个特定格式的字符串表示,称为**哈希串**。哈希串的结构为:

$$< H >< L_1 >< k_1 >< I_1 >< F_1 >< v_1 > \cdots < L_N >< k_N >< I_N >< F_N >< v_N >$$

各子串的含义和编码规则参见表4.2。举例来说,如果某个映射关系为 $\{foo : bar, hello : world\}$,那么表示它的哈希串的一种可能形式为 \x02\x03foo\x03\x00bar\x05hello\x05\x00world\xFF。在二级哈希表中查找元素的复杂度为 O(N),其中 N 为这个哈希表中元素的的个数。

第 5 章　性能测试与结果分析

本章介绍测试分布式二级哈希表性能的实验,并对实验结果进行了简单的分析。第 2.3节我阐述了分布式二级哈希表的应用假设,后续章节则说明了实际的运行环境是怎样决定了系统的设计和实现。很遗憾我并没有得到那样理想的环境,用来对分布式二级哈希表做一番彻底的测试。因此本章的数据和结论只能在一定程度上反映出分布式二级哈希表的性能,关于系统的最终评价还要在系统投入实际运行之后,通过检测其运行情况来得出。

在本次实验中,我测试了分布式二级哈希表对于写入 blob 的两项性能,包括系统在单位时间内进行的写入操作数,以及进行一次写入操作的平均时间两项内容。在每个单项测试中,绘制出了上述性能指标随客户端总写入进程数的变化曲线,其中,客户端总写入进程数均从 1 变化到 12。这些进程一刻不停的执行写入操作,对于测试平均写入执行时间的实验,在统计结果时去除了准备数据等无关行为的时间,只计算从开始调用系统接口函数,到该函数返回的时间;对于测试系统在单位时间内进行写入操作数的实验,事先将数据准备好,避免无关操作的影响。对于某一个测试点,比如客户端总写入进程数为 N 的情况,这些进程并不是同时开始工作的,而是依次提前 10 秒开始一刻不停的实行写操作,最后一个开始的进程总共运行 30 秒钟,因此倒数第二个开始的进程共运行 40 秒钟。这样,所有的进程将同时停止,并且在最后 30 秒钟同时运行。实验结果即最后 30 秒钟里,所有进程的所有写入操作的加权平均值。这样做的目的是希望尽快使系统达到一个稳定状态,记录该状态下系统的性能。

分布式二级哈希表读取数据的性能很高。用类似的方法测试分布式二级哈希表读取数据的能力,读取操作的平均用时在 10 毫秒以内,并且随客户端并发读取进程数的增加变化不大。这主要是因为存储服务器上的 Redis 的读取性能远远高于写入性能。由于读取操作的失败不会影响数据的完整性,而 Redis 中所有的数据存储在内存或者应用层虚拟内存中,因此绝大部分情况下,读取操作需要的数据都可以直接获取,而不必进行硬盘的读操作,这大大缩短了一次读取操作需要的时间。相反的,写入操作的失败可能影响数据的完整性,在4.3节中提到过,每次写入操作都会被记录在一个日志文件中,并且每当对数据进行过一定次数的写入操作后,Redis 就会将内存中和用户态虚拟内存中的数据压缩后存入硬

盘。对于像本次实验这种写入数据的速率和规模,触发写入硬盘操作的频率很高,因此写入操作的性能会远远低于读取操作,这也是本次实验着重探讨写入分布式二级哈希表写入性能的原因。

本实验的硬件配置参见表5.1。其中,配置服务器同时在运行着其他的程序,因此并没有彻底发挥所有硬件的最大性能。对于客户端的配置,所有数据的索引最大长度为 15 字节,数据值的大小为 1 字节到 64KB 随机分布,索引和值的内容是随机生成的;每个客户端进程的线程池共有 255 个工作线程;$R + W > N$ 语义按照 $(N, W) = (3, 2)$ 配置。

表 5.1　分布式二级哈希表性能测试实验硬件配置。共设置四台存储服务器,配置同构。其中一台服务器同时作为配置服务器。客户端在一台单独的计算机上。所有的存储服务器通过千兆以太网交换机进行连接,客户端计算机和存储服务器的连接额外经过一台千兆以太网交换机,客户端与此交换机通过百兆网线相连。

配置	**存储服务器**	**客户端**
数量	4	1
内存	8GB	8GB
CPU	4 核 Intel(R) Core(TM)2 Q4900 2.66GHz	4 核 Intel(R) Core(TM) i5 750 2.67GHz
硬盘	1TB SATA	1TB SATA
操作系统	Linux 2.6.32-30-server x86_64	Linux 2.6.38-8-generic x86_64
网络	所有的存储服务器通过千兆以太网交换机进行连接,客户端计算机和存储服务器的连接额外经过一台千兆以太网交换机,客户端与此交换机通过百兆网线相连。	

实验过程中,所有的写入操作均成功返回,没有一次操作失败,即满足第2.3节对于操作成功率大于 99.999% 的要求,因此在试验结果中将不再对操作成功率作额外说明。

5.1　单位时间写入数据次数

分布式二级哈希表在稳定运行时,单位时间内完成的写入操作数与客户端并发写入进程数的关系如图5.1所示。从图中可以看出,进程数较小时,单位时间写入数据次数随进程数增加而增加,说明系统的运行能力尚未饱和。当并发写入

进程数达到 7 个以后，系统单位时间完成的写入数据次数不再随写入进程数增加而增大，而是稳定在一个固定数值，即每台服务器平均每秒最多处理的写入操作数约为 30。

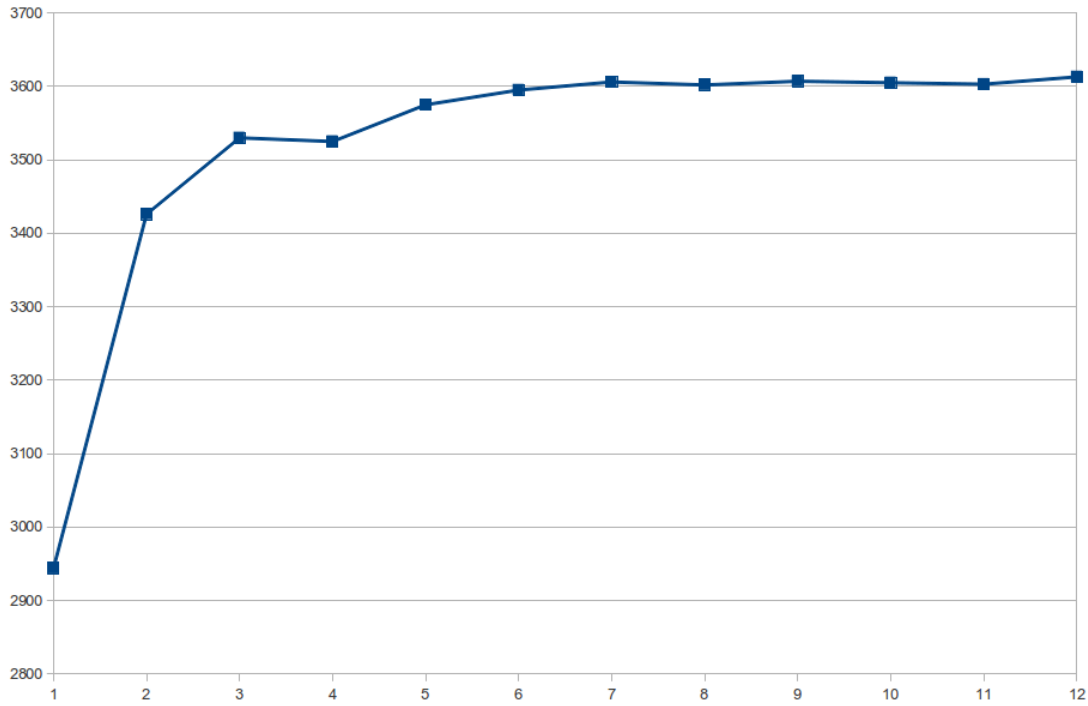

图 5.1　分布式二级哈希表单位时间写入数据次数与并发写入进程数的关系。横轴代表写入进程数，纵轴代表半分钟内系统完成的写入请求总数。每台服务器平均每秒最多处理的写入操作数约为 30。

5.2　平均写入时间

分布式二级哈希表在稳定运行时，写入操作的平均执行时间与客户端并发写入进程数的关系如图5.2所示。从图中可以看出，写入操作的平均执行时间随客户端并发写入进程数近似线性增加。

5.3　结果分析

由于实验只计算最后半分钟所有写进程同时工作时的数值，而进程又是逐个加入的，因此当进程数较少时，系统的最后半分钟并未达到饱和运行状态。当进程数比较多时，在最后半分钟的运行过程中，系统已经饱和，因此单位时间内

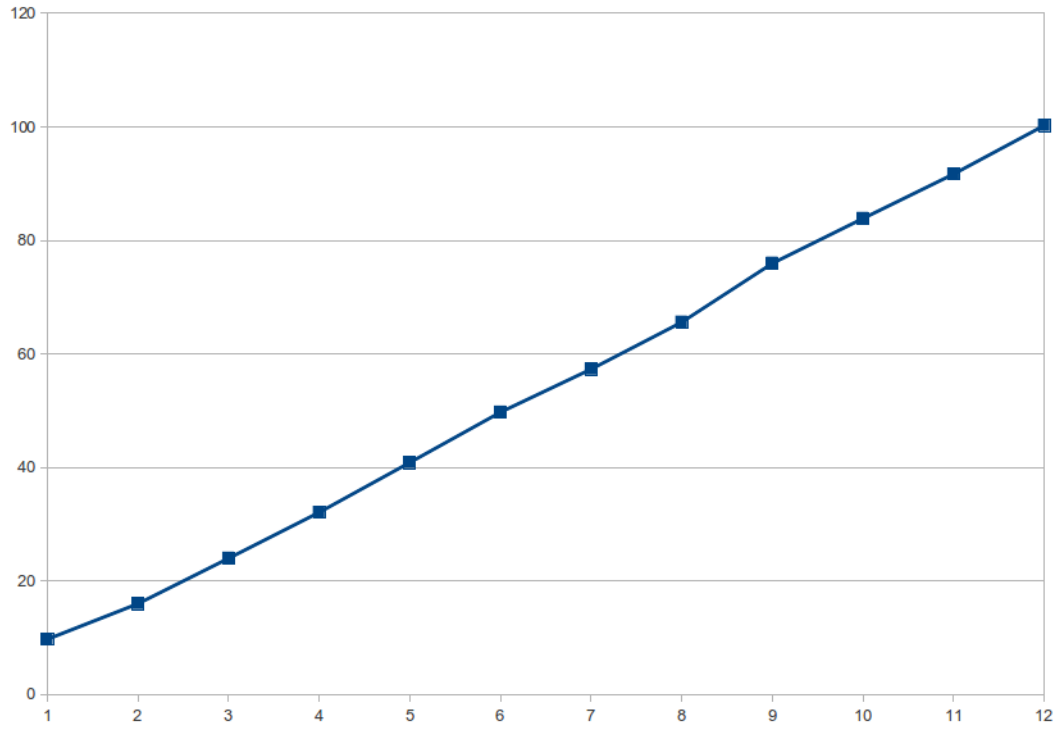

图 5.2　分布式二级哈希表平均写入时间与客户端并发写入进程数的关系。横轴代表写入进程数，纵轴代表平均写入时间，单位是毫秒。平均写入时间随写入进程数增加近似线性增大。

的写入操作次数不再发生变化。普通 7200RPM SATA 硬盘的 IOPS 约为 90[①]，假设一次写入操作涉及到一次硬盘读写操作，那么在当前实验条件下并没有达到硬盘的写入极限，即系统的瓶颈并不是存储服务器的物理存储介质。另一方面，平均写入时间随并发写入进程数近似线性增加也不是出于偶然。为了探究其中的原因，找到系统的瓶颈，我做了进一步的实验，将每一个操作分解，统计各个子操作的时间，得到图 5.3所示的结果。其中，蓝线、红线、黄线均表示网络通信的时间，并且红线与黄线之和跟绿线的差值即为客户端本地计算的开销，比如线程的调度等。绿线与蓝线的和与深红线的差值也表示本地开销。比较意外的是，几乎所有子操作的时间都随写进程数线性增加，因此基本可以断定瓶颈在于客户端而不是存储服务器的处理能力。图 5.1已经说明当线程数不大于 4 个时，存储服务器并没达到饱和状态，因此黄线呈线性变化的原因应该是客户端读取 socket 结果的时间呈线性变化。减小线程池规模得到了近似的结果，排除了调度开销过

① http://en.wikipedia.org/wiki/IOPS

大的可能。另一方面,如果在另一台直接与存储服务器通过千兆以太网网线和交换机相连的计算机上运行客户端程序,系统的效率显著增加。12 个写入线程时,平均写入时间控制在几十毫秒左右,而每台服务器平均每秒处理的写入操作数达到了 100 左右。至此,系统的瓶颈已经确定。由于客户端和与之相连的交换机之间的网线带宽不足,只有百兆,影响了系统的运行效率。

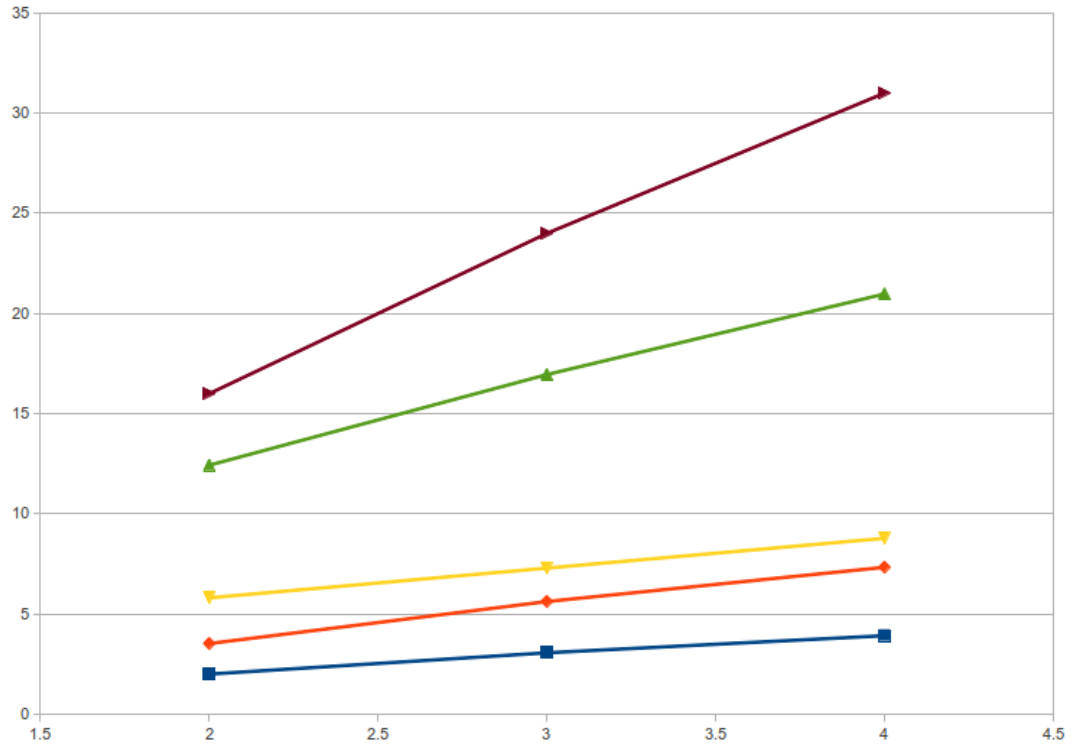

图 5.3　分布式二级哈希表平均写入时间拆分。横轴代表写入客户端并发写入进程数,纵轴代表平均写入时间,单位是毫秒。蓝线是客户端从配置服务器获取目标存储服务器 IP 地址平均时间,红线是客户端向存储服务器发送指令和数据的平均时间,黄线是客户端从发送完指令和数据到读到存储服务器传回结果数据的平均时间,绿线是任务进入线程池到读到结果的平均时间,深红线是上层应用调用接口函数到得到返回值的平均时间。各个子操作的时间均随客户端写入线程数近似线性增加。

第 6 章　可能的改进

分布式二级哈希表的当前设计能够满足上层应用的基本需求,主要功能也都在保证系统运行效率的前提下得以实现。随着系统的应用范围不断扩大,上层应用的种类和数目逐渐增多,当前的系统设计和实现必然要随之作出调整,以满足日益变化的需求。下面我将对未来作出展望,预测可能出现的问题,并讨论了解决方案。

6.1　可扩展性

可扩展性是任何分布式系统需要考虑的问题。对于分布式存储系统来说,随着对系统的利用,可用存储空间将越来越少,必然要通过向集群中增加存储单元来扩大系统的总存储容量。另一方面,当结点出现故障不再正常工作,或者局部结点无法通过网络访问,这些结点不应当被系统认定为工作结点。可扩展性指的是系统支持结点的加入和离开,要在集群规模发生变化时,仍然保证较高的运行效率。随着系统规模的增大,虽然单个结点出现故障的概率并未发生变化,但是系统中有结点出现故障的概率却大大增加。故障的解决也可能不是在统一的时间,比如当系统管理员检测到一台存储单元上的操作系统无法响应,他可能立即将该机器重新启动。因此,系统需要在结点随时出现故障和随时恢复正常的情况下高效率运转,也就是说,系统还要支持结点的动态加入和离开。分布式二级哈希表在设计之初就考虑到了这一潜在的需求,因此数据分配和备份采用了一致性哈希算法。只要在当前的系统模块上添加数据迁移功能,即可使系统支持结点的动态加入和移除。

数据迁移是指在一致性哈希算法中,当有结点动态加入和离开集群时,为了保证算法的正确性和数据的负载均衡,需要将某个存储服务器上的数据移动到另一个存储服务器上。其原理和数据移动规则在图4.1中已经说明,这里不再赘述。数据迁并不像想象中的那么简单,需要考虑很多的问题。比如,在数据迁移的过程中,如果有结点加入和离开集群,而且与迁移过程相关,那么可能造成数据的丢失。数据迁移应当由后台进程异步执行,那么可能造成对弱一致性性质的破坏。要实现数据迁移,需要将诸如此类的问题考虑清楚,并提出合理的解决方

案。

6.2 容错性

在有局部错误发生时,系统仍能保证语义的正确性,甚至仍然能够保证系统运行效率的能力,就是系统的容错性。在分布式系统中,错误的出现率比单机要大的多,而且错误种类可能五花八门。比如,单个存储结点上的工作进程可能无响应,单个存储结点的操作系统可能没有响应,单个存储结点的存储介质可能出现故障,网络可能阻塞等等,这些故障都是分布式二级哈希表需要应对的情况。目前,系统通过一致性哈希算法对数据进行分配和备份,保证数据的完整性,不会有数据丢失。而在本地存储上,则依赖 Redis 的一系列容错机制来提高系统的整体容错性。

为了进一步提高分布式二级哈希表的容错性,还可以对系统作出改进和优化。比如,目前客户端路由层异步提取结果的时限是静态配置的,这不能适应网络条件变化剧烈的情况。在今后的实现中,可以将引入反馈机制,使该时限根据网络延迟动态调整,在保证操作成功率的同时,尽可能降低系统的平均响应时间。此外,还可以对 Redis 恢复数据的代码作出优化,提高系统从故障中恢复的速度,减少由此带来的开销。

6.3 其他改进和优化

此外,还可以通过一系列的改进和优化提高分布式二级哈希表的性能。比如依照当前的实现,每当上层应用发出请求,路由层都需要通过远程过程调用从配置服务器获取目标存储服务器的 IP 地址,这需要进行一次网络通信。如果引入缓存机制,那么可以在大部分情况下节省一次网络通信带来的延迟,减少操作的平均响应时间。不过引入缓存机制的前提是保证缓存内容和配置服务器上控制信息的一致性。可见,分布式二级哈希表的性能还可以提升,今后的工作应当在考虑为系统添加功能的同时,充分挖掘当前设计和实现中的缺陷,进一步提升系统性能。

第 7 章　结论

本论文介绍了分布式二级哈希表,详细阐述了其架构设计和实现细节。分布式二级哈希表采用分布式领域一些经典技术,在分布式系统上实现了二级哈希的存储语义,为邮件系统、个人云存储等上层应用提供后端的存储支持。分布式二级哈希表由配置服务器、客户端和存储服务器集群构成。其中,配置服务器监控各存储结点的运行状况,集中管理控制信息,并通过一致性哈希算法实现数据的分配的备份。客户端采用分层结构,利用远程过程调用从配置服务器获取数据的目标存储服务器的 IP 地址,借助线程池实现请求的并行异步执行,并通过 $R + W > N$ 语义保证了数据在系统中的弱一致性。单个存储结点上采用开源项目 Redis,实现了本地二级哈希存储,具备很高的性能。

分布式二级哈希表在设计之初,就本着尽量提高系统运行效率的根本原则。实验证明,当前的系统实现也确实达到了较高的性能。当然,分布式二级哈希表仍然存在性能提升的空间,今后的工作可以从增强系统的可扩展性和容错性入手,进一步完善和改进分布式二级哈希表。

插图索引

表格索引

公式索引

参考文献

[1] 马少平, 刘亦群. 去伪存真, 去粗取精——页面质量评估及其在网络信息检索中的应用. 2006

[2] Ghemawat S, Gobioff H, Leung S. The Google file system. Proceedings of ACM SIGOPS Operating Systems Review, volume 37. ACM, 2003. 29–43

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

[4] Hastorun D, Jampani M, Kakulapati G, et al. Dynamo: amazon's highly available key-value store. Proceedings of In Proc. SOSP. Citeseer, 2007

[5] Karger D, Lehman E, Leighton T, et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. Proceedings of Proceedings of the twenty-ninth annual ACM symposium on Theory of computing. ACM, 1997. 654–663

[6] Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 2008, 26(2):1–26

[7] Fox A, Gribble S, Chawathe Y, et al. Cluster-based scalable network services. Proceedings of ACM SIGOPS Operating Systems Review, volume 31. ACM, 1997. 78–91

[8] Rivest R. RFC1321: The MD5 message-digest algorithm. RFC Editor United States, 1992.

[9] Lamport L. Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 1978, 21(7):558–565

致　谢

衷心感谢我的导师陈文光教授对我完成毕业设计给予的悉心指导,感谢我的辅导教师陈康副教授对我的热心辅导与帮助。

感谢清华大学计算机科学与技术系高性能所集群计算组实验室的各位学长学姐,长期以来对我提供了各方面的热心帮助,耐心地为我答疑解惑,使我能够顺利完成毕业设计。

在写作论文的过程中,我使用了 LaTeX 工具。开源模板 ThuThesis大大简化了写作论文的工作量,特此感谢。

声　明

本人郑重声明:所呈交的学位论文,是本人在导师指导下,独立进行研究工作所取得的成果。尽我所知,除文中已经注明引用的内容外,本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体,均已在文中以明确方式标明。

签　名:_____ 日　期:_____

附录 A　外文资料的的调研阅读报告

A Brief Report on Distributed Storage System

Introduction

Distributed system is a major topic of current computer science. Among it, a distributed storage system usually takes advantage of multiple machines to gain capacity, reliability, availability, while for people who operates data, it seems that he is facing with a single machine and needn't care about the whole backend framework.

Categories

There are a lot of typical distributed storage systems. Google File System [1], MooseFS① and many others are classified as *distributed file system*. A client can communicate with the cluster to store and retrieve data much like operating files in a local file system. Usually, they support full namespace hierarchy and the data is accessed with a path.

Another category is classified as *distributed database*. The significant feature of the data stored in a distributed database is that the data is stored and accessed aligned by *columns* and *rows*. Sometimes, more strengthful databases also record relationship between data and support complicated query semantics.

Apart from the two categories mentioned above, a novel kind of distributed storage system is getting more and more popular. It is not only light-weighted, as it doesn't support relationship between data or just supports weak relationship, but also flat, usually because it doesn't support complicated namespace hierarchy but just a map of key-value pairs. On the opposite, the system which belongs to this category is usually of high performance, to be measured by throughput, latency, availability, reliability and other criteria. Such systems are called *distributed key-value stores*. Douban's BeansDB

① http://www.moosefs.org/

[①], Kyoto Cabinet[②] from Japan and many others are all successful distributed key-value stores. As such storage systems are light-weighted, some systems, like MemcacheDB [③], decide to put data in memory or virtual memory to gain performance burst.

Fundamental Concepts

Before I summarize existing distributed storage systems, I would like to explain some fundamental concepts related to distributed system. Without a clear introduction on these conceptions, it would be impossible to comprehend the beauty and elegance of architecture designs on famous distributed storage system.

Replication

Replication is copies of same data. In distributed system, data is distributed to many computing and storage machines. Under most cases, data is evenly stored on different machines. The world will be easy but fragile if we don't make copy of data. Let's take a closer look at why data backup is necessary with some simple calculation. If the possibility of failture for a single machine is p, and for simplicity, we assume all the machines are independently identical, i.e., the possibility of failure for each machine equals p and a machine never notices whether his buddy is alive or dead. What is the possibility that a system containing n identical machines goes down with one or more machine failing to work at some time? Yes, you are right! It is $1-(1-p)^n$. So what does this mean? Although p may be small, when n gets larger and larger, the result tends to reach 1! Commonly, a datacenter of companies like Google contains from thousands of to millions of machines with moderate disks. Disk failures are not rare but a common case. If the data is not replicated, it would be impossible to ensure the integrity of data, as recovery of data from a failed disk costs time, computation resource and network bandwith, yet this is not always possible. Hence making copies of data is critical in large systems like distributed key-value store.

① http://beansdb.googlecode.com/files/Inside%20BeansDB.pdf
② http://fallabs.com/kyotocabinet/kyotoproducts.pdf
③ http://memcachedb.org/memcachedb-guide-1.0.pdf

Consistent Hashing

Replication of data may not be as simple as it seems at first glance. There are many tricky technologies behind to provide the correctness of the replication mechanism, and further the improvement of the performance. Situation becomes even worse when the system is distributed as we have to make decision on the choice of machine for different block of data.

Usually, the data is distributed by its key. First, a hash function is needed to calculate a hash value for the key. The key may be a string or even any binary stream, while the result of the hash function always falls into a finite range. The range, called *hash space*, is divided into several segments, each representing a machine. That machine takes responsibility of all the data with keys whose hash value is within that range.

Different distributed algorithms vary in their hash functions. A simple instance could be a distributed system which adopts *Cyclic Redundancy Check* as its hash function. As we mentioned above, each machine is associated with a segment in the range of hash value. However, in most cases, this range is further made up of several subsegments, called hash slots. Often, hash slots within a segment are not located adjacently to each other in the hash space. Note that this is why the algorithm distributes data evenly among different machines, while the impact of a single machine failure is minimized.

Consistent hashing[4] is a method to distribute data evenly within a storage cluster, regardless of the specific hashing function employed. If we concatenate the tail of the hash space to its head, the hash space will rewind like a ring, called the *hash ring*. We put some nodes on the ring and the ring is broken up into some arcs. These nodes are called *virtual nodes*. A physical node is a storage server, which is a collection of same amount of virtual nodes. The assignment of virtual node to physical one is random, which means that all the virtual nodes, which belong to the same physical node, may not be adjacent on hash ring. Different virtual nodes, which belong to different physical nodes, may not appear on the hash ring in a round-robin manner. They are just randomly distributed.

How to decide which storage server should take responsibility of the data associated with a specific key? If we don't take data replication into consideration, the server

is the physical node found as follows. We traverse clockwise from the point on the hash ring representing the hash value of that key. The physical node, where the first virtual node met like this belongs to, is the storage machine which is responsible for that data.

When a machine refuses to work any more due to failure, the virtual nodes associated with it should be taken care by other physical nodes. It is obvious that each of such virtual nodes should share the same physical node, to which the next virtual node belongs, counting clockwise. As the virtual nodes conducted previously by the failed machine are distributed randomly, it is probably expected that the data handled previously by the failed machine will be distributed evenly across other active machines. It goes the same when a new machine joins the cluster.

In the real world, data is replicated into many copies. The data associated with a single key is stored on several machines, namely N physical nodes. These machines are determined by tranversing clockwise from the point on the hash ring representing the hash value of that key. The traversal stops when the first M virtual nodes belongs to N different physical nodes. These N physical nodes are the target machines.

Example Distributed Storage Systems

In this short article, I'll make a brief summary on several famous distributed storage system with introductions on their significant features.

Dynamo: Amazon's Highly Available Key-value Store

I believe the most famous distributed key-value store is Amazon's Dynamo [2]. Although it is neither open source nor available for organizations outside Amazon to use, it is highly recognized by scientists and engineers in the area of distributed computing.

Perhaps Dynamo gains popularity mainly because of its elegant design. It is a perfect example of minimizing system functionality to satisify basic requirements of application. Dynamo acts as an internal infrastructure for Amazon's many services, such as the on-line book stores. In most of the senarios, the service beyond Dynamo has such a requirement that data is highly writable. Considering this specific requirement, Dynamo is designed at first day to support high throughput and low latency of write

request, while to sacrifice consistency hence increasing of read request both operation time and possibility of version conflict, which is tolerable within these services.

Bigtable: A Distributed Storage System for Structured Data

Bigtable[3] is an outstanding representative of the brand new *NoSQL*. It differs from traditional database by not supporting complex relationship between data, yet more flexible. Bigtable is considered as a multi-dimensional mapping of data. The last dimension is usually timestamp which means the database records historical snapshots of data. The database is flexible that dimension names of different data may be different.

Redis: An Open Source, Advanced Key-value Store

Redis① is an open source key-value store. It beats other hash tables for its rich type of values, such as binary stream, lists, sets, sorted sets and even hashes, while still maintaining high performance because all the data resides in memory. Redis not only performs data compression but also implements a virtual memory layer in user space to solve memory shortage. It is also fully journaled to enhance ability of fault tolerance.

参考文献

[1] Ghemawat S, Gobioff H, Leung S. The Google file system. Proceedings of ACM SIGOPS Operating Systems Review, volume 37. ACM, 2003. 29–43

[2] Hastorun D, Jampani M, Kakulapati G, et al. Dynamo: amazon’highly available key-value store. Proceedings of In Proc. SOSP. Citeseer, 2007

[3] Chang F, Dean J, Ghemawat S, et al. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 2008, 26(2):1–26

[4] Karger D, Lehman E, Leighton T, et al. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. Proceedings of Proceed-ings of the twenty-ninth annual ACM symposium on Theory of computing. ACM, 1997. 654–663

① http://redis.io/

综合论文训练记录表

<table>
<tr><td>学生姓名</td><td></td><td>学号</td><td></td><td>班级</td><td></td></tr>
<tr><td>论文题目</td><td colspan="5"></td></tr>
<tr><td>主要内容以及进度安排</td><td colspan="5">指导教师签字：
考核组组长签字：
年　　月　　日</td></tr>
<tr><td>中期考核意见</td><td colspan="5">考核组组长签字：
年　　月　　日</td></tr>
</table>

指导教师评语	指导教师签字：_____ 年　　月　　日
评阅教师评语	评阅教师签字：_____ 年　　月　　日
答辩小组评语	答辩小组组长签字：_____ 年　　月　　日

总成绩：_____

教学负责人签字：_____

年　　月　　日