基于JavaWeb架构的分布式缓存设计与实现

摘要

新世纪信息化技术的发展，互联网承载了越来越多的内容。我们现在拥有了更多的智能设备,触屏手机，平板电脑以及pc小型化的发展趋势越发明显。我们可以随时随地使用Wi-Fi无线网络和手机3G、4G网络来连接互联网，获取知识和信息。另一方面，随着计算机软件从单机单任务操作系统、单机多任务操作系统、到现在网络的发展历史，软件的发展思路产生了越来越多的变化，相对的计算机硬件而言，到现在半导体技术已经接近纳米技术的极限，单机计算节点的性能提升越来越不明显，升级提高单机性能的效果和性价比越来越低，并且单一个节点存在宕机和不稳定的风险。我们不得不使用更多的计算节点来处理越来越庞大的请求，分布式计算机以及对应的计算机集群，变得越来越流行。各个互联网巨头都拥有自己的大型分布式网络和计算系统。设备小型化和互联网大数据的趋势，让数据的数据集中化和低成本统一管理变成一种共识。我们将云服务中的云计算和大数据结合，能够高效地分析出数据中本质的信息。然而云服务中的服务器都是以分布式单机组成集群的形式来进行调用。分布式服务中的单点故障问题以及存储一致性都是很重要的问题。如果分布式网络中的主节点失效，从而导致整体系统的不可用，则会导致严重的问题。

本文在分析讨论分布式服务中的解决单点故障和存储一致性基础上，开发一个提供良好性能的基于Java web 架构的分布式缓存系统。

关键字： 云服务， 单点故障 ， Java web ， 分布式缓存

Distributed cache based on JavaWeb architecture design and implementation

Author:Airan Lu
Tutor:Zhao Yan

Abstract

With the development of information technology, the Internet brings more and more content. We now have more smart devices. The development of smart phones, tablet PCs and mini PC. We can use wireless network and mobile phone 4G 3G network to connect Internet, access to knowledge and information anytime and anywhere. As computer software, on the other hand, from the single machine single task operating system, LAN multitasking operating system, to the present network, the development history of software development ideas produced more and more changes, relative to the computer hardware, to the semiconductor technology is now approaching the limits of nanotechnology, the performance improvement of computing nodes is more and more is not obvious, upgrade to improve the effect of the single machine performance and cost performance is more and more low, and there is a single node goes down and the risk of instability. We had to use more computing nodes to handle the request of the growing, distributed computer and corresponding computer cluster, become more and more popular. The Internet giant has its own large distributed network and computing systems. Device miniaturization and the trend of Internet big data, make unified management of data centralization and low cost become a consensus. We will cloud services in the cloud computing and big data combination, can efficiently analyze the nature of the information in the data. However of single cluster server in the cloud services are distributed in the form of a call. A single point of failure problem in distributed service and storage consistency is very important problem. If the master node failure in a distributed network, leading to the

overall system is not available, will cause serious problems.

This paper discussed single point fault and store the consistency of distributed services. Based on the development of a good performance of the distributed cache system based on Java web framework.

Keywords: Cloud services, a single point of failure, Java web, distributed cache

目　录

第1章　绪论

1.1 研究内容

Java web 架构分布式缓存是　使用Java架构平台，部署在计算机服务器集群中，采用分布式算法和多机复制的 key-value 缓存中间件。

1.2 研究意义

基于Java web　架构的分布式缓存系统可以用于Java　以及其他高级语言开发的计算机应用以及大型web网站等需要暂时存储大量数据以及文件的需求。降低请求直接对于数据源如web系统的压力。并且通过分布式的多机备份，保证了数据的具有高度冗余，防止数据丢失。

1.3 研究现状和发展趋势

开源的分布式缓存和nosql 数据库有很多　常见的缓存有Memcache 和 redis 这些缓存本身并没有存在分布式,而是通过额外的配置来达 到分布式缓存的效果。　常见的分布式　计算平台是hadoop ,其实现了HDFS ,一个分布式的文件系统。　通过将文件按　照固定大小进行分块处理, 并通过数据冗余提高了系统的稳定性。无论是memcache　redis　还是hadoop　。都存在主从节点故障的问题。Zookeeper作为 一个分布式协调服务,实现了分布式一致性算法　Zab协议　其脱胎于Paxos算法[1]。能够进行自主的选举　主节点。 keyspace 数据库实现了类似的功能,但每个节点的数据的内容都一致,

导致数据存储成本过高。

第2章　系统分析

2.1 研究目标

了解Java web 应用程序的设计和开发流程

使用分布式算法，保证系统的高可用性和数据一致性

本软件使用Eclipse 的开发环境， 使用PaxosLease 选主算法 和2pc 数据一致性算法，一致性哈希算法，开发出基于Java web 架构的 分布式缓存系统。

2.2 需求分析

本软件本质上是一个JavaWeb应用。在同一网段下的不同机器可以设置Ip和不同端口来运行该程序。每台机器启动该程序后变成集群的一个节点，进行主从选举，确定主从关系，加入缓存节点。之后可以接收从主节点传递过来需要缓存的请求。

客户端使用该软件，可以直接作为本地的Hash表来进行读取。

通过综上描述，我们了解这个项目软件的基本功能需求：

1. 服务端启动程序

进行选举，分出主从节点。

确定主从关系后，进行缓存加载和其他设置

主节点要存储每个缓存数据的应存储从机位置

作为缓存，所有从节点，随时准备存储缓存数据

2. 客户端使用该程序

能够向缓存添加数据

能够从缓存中取到数据

第3章　系统开发环境

3.1 开发环境介绍

JavaWeb开发和Java EE 平台密不可分。Java EE（Java Platform，Enterprise Edition）是Sun公司（2009年4月20日甲骨文将其收购）推出的企业级应用程序版本。这个版本以前称为 J2EE。做企业版的JAVA环境，JavaEE提供了更多的功能和选择，最主要是在服务器端的开放的API 以及JAVA 通用的Servlet Web 开发环境，我们可以直接使用Java EE 开发出足够健壮

Java EE 平台 与 Java SE 平台有较大的区别，Java SE（Java Platform，Standard Edition）。Java SE 以前称为 J2SE。作为标准版的JAVA 开发工具包Java SE 包含了支持 Java Web 服务开发的类，并为 Java Platform，Enterprise Edition（Java EE）提供基础。

简而言之，JavaEE是Java SE的合集 。我们可以在Java EE使用更多的API。

开发环境需要以下软件：

Java SDK

Eclipse 4.4

Maven 3.2

3.2 开发环境搭建

1:首先到

http://www.oracle.com/technetwork/java/javase/downloads/index.html 下载安装 JAVA SE JDK 。

http://www.eclipse.org/downloads/ Eclipse IDE for Java EE Developers,

http://maven.apache.org/download.cgi 下载最新版本 Maven Binary zip

2：解压到 指定目录

3：配置环境变量

a.配置JDK环境

JAVA_HOME：Java安装目录

Path：添加Java 安装目录/bin

b.配置Maven环境

MAVEN_HOME : D:\apache-maven-3.2.2

Path : %MAVEN_HOME%\bin

(可选） MAVEN_OPTS : -Xms256m -Xmx512m

4：配置Eclipse：

a.打开Eclipse，点击Help 进入 Eclipse Marketplace （需要联网）。在搜索栏输入Maven Integration for Eclipse ，并进行安装。

b.点击window后，找到Preference页面点击。在左侧选中Java，下拉，会出现Installed JREs 标签，点击Add添加虚拟机。下一步选中JDK 的安装位置，Finish点击。

c.Preference页面点击。在左侧选中Maven，选中User Settings，并进行Maven库的设置。

5：检查环境正确：

在命令行模式下输入：javac –version，如果Java安装正确，会进行版本提示。

在命令行模式下输入：mvn –v 如果安装正确，会提示Maven插件的版本信息。

第4章　分布式系统相关技术及算法

4.1 分布式相关内容介绍

当我们按照线上生产环境，使用一台线上服务器来提供一些数据服务时。我们会碰到一下问题：

1. 单一线上服务器接受线上服务的网络和其他请求。性能压力过大。

2. 单一线上服务器一旦崩溃，会造成线上服务不可用，数据的丢失。

3. 当我们需要对线上环境做一些修改变化时， 单一的线上服务器必须停机，导致服务不可用。

面对以上这些情况，我们不得不对我们的服务器进行拓展，不再使用单一线上服务器，而是引入更多的线上服务器，作为整体向外服务。

一般来说，有两种办法来改变我们的数据服务的现状[11]：

1. 数据映像拓展：

2. 数据分区拓展：

数据映像拓展指的是让所有数据都存储到所有的服务器，让所有的服务器提供完全相同的内容。

数据分区拓展指的是让数据随机或者不随机在不同的服务器上面，通

过hash 算法分配等等，保证分散。

数据分区拓展时，数据丢失问题无法解决。当一台线上服务器产生问题死机或者下线，这台主机上的数据一定会不可见或者丢失。 所以我们提高数据可用性的办法都是通过数据映像拓展，也可以称之为数据冗余。常见的有hadoop 等等

但是我们引入更多的机器解决数据高可用的问题，会引起其他的影响，譬如多台主机间的数据同步事务性，以及数据的一致性问题。

我们举例说明问题：

现有一个需求，需要我们进行转账操作，即 从A账号向B账号汇款 。在标准的关系型数据库（RDBMS）事务中，这个操作会被分解成若干子操作：

1. 查询A账号的余额。是否有大于等于转账的金额
2. 将A账号的余额进行减法
3. 把结果在存入A账号
4. 查询B账号的余额
5. 将转账的金额加入B账号
6. 把结果写入B账号

作为关系型数据库为了保证数据的一致性，这六件事或者全部做完，或者全部不做。并且在这个执行的过程中，对A和B 账号的操作必须进行加锁。其他对于AB 的读写都必须等待这个事务操作的完成。如果不这样做的话，就会存在脏数据的问题。这是无法接受的。

然而上述解决办法只是在单机层面的。如果加入更多的服务器和机器，会引发许多棘手的问题。例如：

如果我们使用数据镜像方法，A和B 的操作可以在同一台主机完成，但是由于我们存在多个数据副本存储在其他的机器上面。这些副本我们也都需要进行更新。或者 A账户 同时要转账给B和C两个人，B和C的账户信息不在同一台服务器上面上面。那么在不同服务器上对同一个数据的写操作，需要保持其数据的一致性，保证数据不冲突。

如果我们使用数据分区的方法， 那么A和B 的数据不存储在一台主机上面，我们可能需要一个跨主机的事务，来完成A到B 的汇款操作，即如果A的扣款成功，而B的加钱操作不成功的话，我们还需要把钱回退给账户A。这样跨机器的分布式事务是很有难度的。

刨除以上的分析，我们还有性能和数据冗余关键点：所以我们在完成一个分布式数据缓存时需要关注的有：

1. 数据的安全冗余

2. 数据的一致性

3. 性能：吞吐量和响应时长

我们经过以上分析，很容易得出结论 ，在分布式的环境下：

如果想要数据具有高可用，就必须存储多份数据分散到不同机器。

写多份数据来保证可用性，又会导致数据需要同步性问题。

数据同步的问题的解决，会导致性能的下降。

上面的三条结论可以归纳成分布式领域的CAP理论：

Consistency(一致性)，数据同步更新，所有数据变动都是同时等待完成的

Availability(可用性),好的响应性能

Partition tolerance(分区容错性) 可靠性

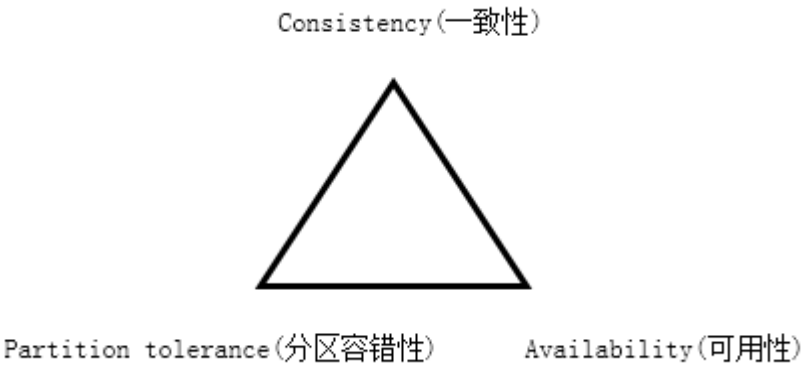


图 4-1

定理：任何分布式系统只可同时满足二点，没法三者兼顾。

我们在设计分布式系统无法完全同时满足这三个要素，而是应该进行取舍。

4.2 数据分区和一致性哈希

consistent hashing 算法于 1997 年 Consistent Hashing and Random Trees 论文[7]提出。解决了缓存失效和加权访问节点的问题。

我们先从一个简单的场景开始分析：

现在有N个一样的cache缓存服务器 ，我们把这所有的缓存服务器看成一个整体，作为一个服务Service，对外整体展示服务。

当我们有了一个对象需要我们进行缓存的时候。最朴素的想法是计算这个对象的hash值，然后把这个hash映射到N个服务器上面。

hash(Obj) %N

当对象的hash value 均匀散列的理想情况下的话， 那么这N个服务器可以完全均匀的接收这些缓存对象的读写。

然而有两种情况无法避免:

1. 一个cache 服务器m break down 崩溃 ， 这样所有映射到 m的 对象都会失效。 我们需要将m 从cache 集群中分离隔离出来。那么现在集群中拥有的机器是N-1台 。

现在映射的算法公式就变成了hash(obj)%(N-1)

2. 由于缓存服务压力越来越大， 需要添加节点。我们假设添加一台新的cache服务器。 那么现在集群中共有 N+1 台 cache 服务器。

现在映射的算法公式就变成了hash(obj)%(N+1)

3.后加入的节点可能运算存储能力要更强。 上述的算法公式无法满足这个条件。

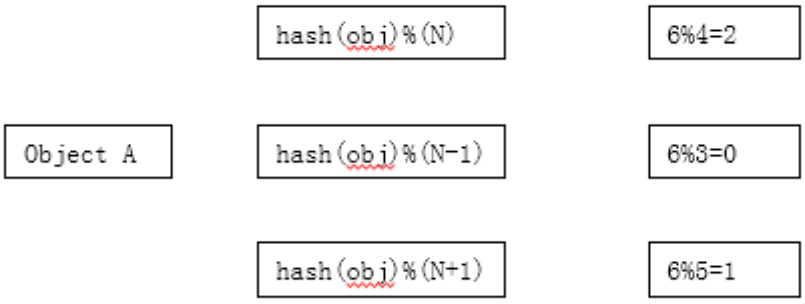


图 4-2

在1,2 的情况下， 同样的一个对象可能映射到不同的机器上，即所有缓存全部失效。这点是我们无法接受的。因为如果缓存全部失效 ， 那么全部访问对象请求就会涌入后台服务器。这点是无法接受的

由于存在上述问题， 说明了我们简单的 哈希hash 公式不适用在 分布式缓存中。

我们希望找到一种hash 算法，能够保证原有的hash 值尽量不收影响，还能增加基数。

一种良好的分布式hash算法 应该有以下几个特性：

平衡性：

平衡性指hash所得到的哈希值能够尽可能的分散到想要的最大区间的任何一个位置上。 大部分的区域都被覆盖到，从而负载均匀。

单调性：

单调性指的是如果原来已经有一些值通过哈希分派到了相应的cache区域，又有新的cache加入到集群中，那么原来使用hash 算法存储的旧数据仍能够被访问到。同时新加入的cache 缓存也能被hash 映射到。

分散性：

分布式环境中， 一台从服务器或者客户端可能无法得到所有cache 缓存服务器的信息。 而仅能看到一部分的缓存服务器。当这台机器开始通过hash 映射内容到缓存上时，由于不同的机器的使用和看见的hash 范围有所不同，从而导致不同机器通过hash 哈希算法向 cache 服务器所存储的内容 相同hash值的对象 被存储到不同 cache 服务器中。这种情况要尽可能的避免。因为这种情况降低了系统存储的效率。优秀的哈希算法会将

不一致的情况降低到最低。

负载:

负载指的是一台cache 服务器被不同的机器通过hash 来调用，那么这台cache server 就会有可能被不同机器存储不同内容。 好的hash 算法应该尽量避免这种情况。

平滑性

平滑性是指缓存服务器的数目平滑改变和缓存对象的平滑改变是一致的。

我们在上面所使用的简单的hash算法在很多方面都不合格， 尤其是单调性。

一致性哈希算法很好的解决了这个问题。 一致性hash 是将 hash算法的数学随机性和 链式数据结构的定向指向特征结合到一起，较好的解决了hash算法单调性的问题

简单来说，一致性hash 算法将整个hash算法映射所产生的值域 变成一个虚拟的，有方向性的圆环。我们假设某个普通线性hash 哈希算法所产生的值域范围为0-2^32-1（即哈希值是一个32位无符号整形）

那么整个hash空间我们按照顺时针方向组织。 0和2^32-1重合

接下来我们把每个cache 服务器 通过hash 算法 部署到 这个圆环上面

现在我们有三个 Object A、B、C 需要缓存到 cache ， 我们把这个三个对象的hash值也指向这个圆环

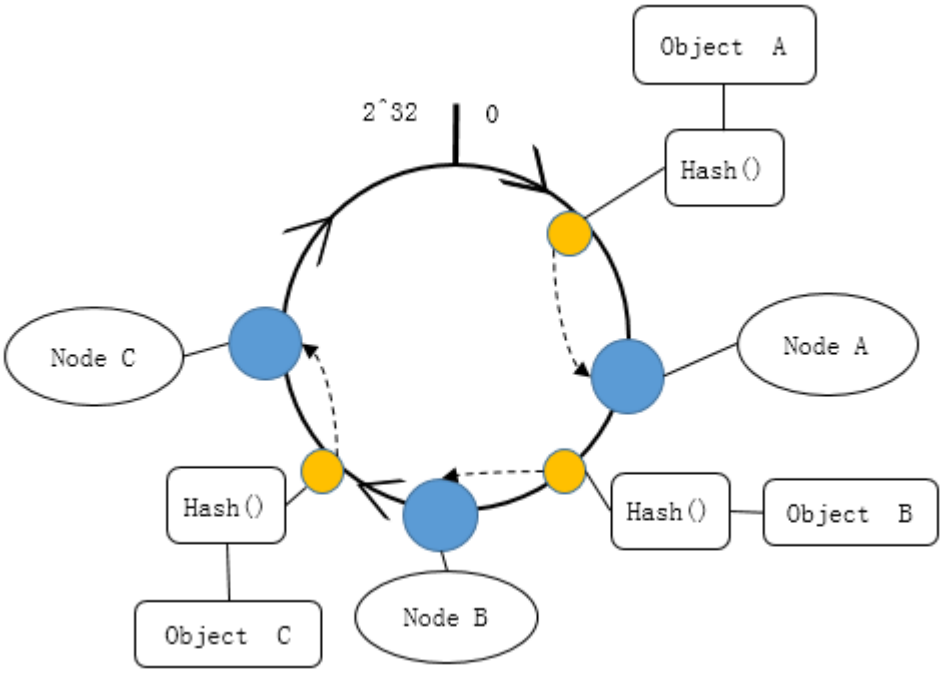


图 4-3

如果节点 B 的cache 服务器 崩溃，我们可以看到此时对象A、C的缓存加载不会产生问题，只有ObjectB的访问会沿着一致性哈希环路顺时针向前找到Node C。通常来说 在一致性哈希算法的实现中，一台服务器不可用宕机所产生的结果，损失的数据可能是这台服务器在一致性哈希圆环中到前一台服务器的部分，即逆时针从宕机节点到上一个节点之间数据，其它不会受到影响。

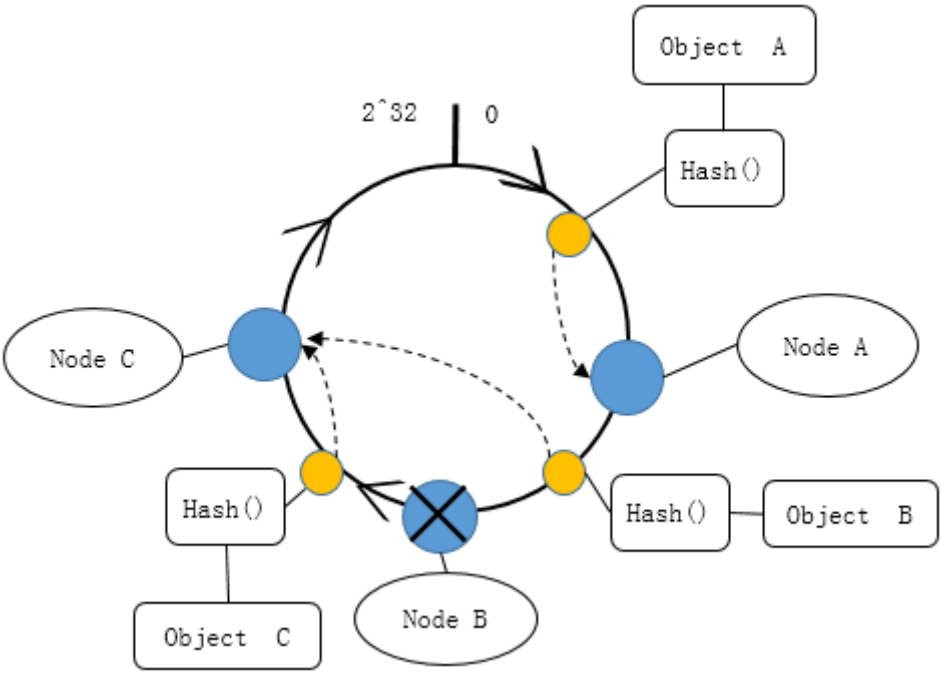


图 4-4

还有其他的情况，如果我们在圆环上添加一台服务器 D，如图

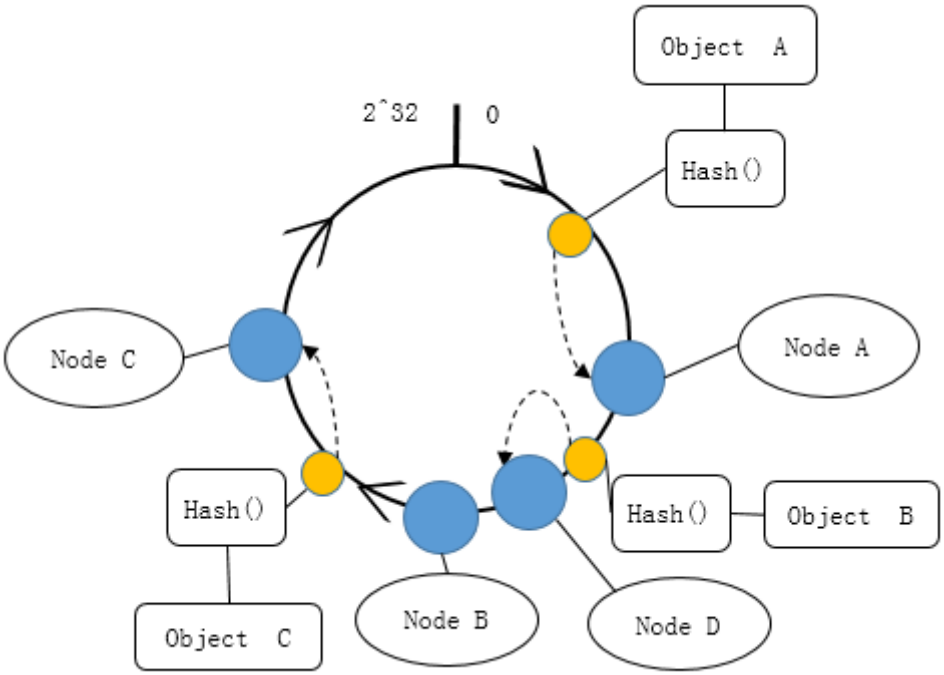


图 4-5

这时我们对Object A、C读取删除不会有任何影响，只有对象B需要重定位到新的Node D 。一致性哈希算法的具体实现中，一般来说，增加一台节点所影响的缓存数据不会很多，只有新节点到一致性哈希圆环中后一个节点的部分中之间数据，所有其他的数据根本不会受到影响。

通过我们上面的分析和讨论，我们可以得出结论，一致性哈希圆环对于节点服务器的添加删除所产生的改变都仅仅是产生极小的部分变化数据的位置。

还有，如果一致性哈希圆环上的节点如果过少的情况下，很容易导所有的请求都堆积在一台服务器上面。

面对这种情况，我们只需要去做一些虚拟节点来解决这个问题，而实际的节点可能对应大量的虚拟节点，假如我们要把NodeA节点进行扩充，“Node AX”、“Node AY”、“Node AZ”这样的虚拟节点放置到一致性哈希圆环上面。这样，我们就解决了圆环上面节点太少导致的问题。一般来说这个虚拟节点的数据倍数要设置的尽可能大些 可以是几百或者几十。

4.3 数据一致性算法

4.3.1 一致性模型

对于分布式环境中的数据一致性，一般来说，简单的存在三个模型：

1. 弱一致性： 当我们向系统中刷新一个值的时候，这个值很有可能读不出来，也有可能读出来。常见于某些cache系统

2. 最终一致性：当我们刷新一个值之后，这个值有可能在某个时间节点之前读不出来，但是通过某个时间窗口后一定能够查询到修改。

3. 强一致性：当我们写入一个值之后，不需要等待，我们能够立即查询这个结果。常见于关系型数据库等等。

从这三种数据一致性模型来说，弱一致性和最终一致性基本都是来进行异步冗余的，二千一致性模型一般来说都是要求立即返回结果，所以是同步冗余。通常来说异步的执行结果都能带来更好的效率和体验，但同时我们需要更富在去控制系统中同步异步之间的状态。同步可能很简单，但是多台机器之间的相互协调问题，保证这种同步需要大量的性能开销。

下面我们简要分析下分布式系统常用的数据一致算法：

4.3.2 两阶段/三阶段提交协议

这个算法的简称也叫做2/3PC Two/Three Phase Commit 。在分布式系统中，每个单独的服务虽然知道自己的操作的结果成功与否，但是并不知道其他节点操作结果的成功与否。每当出现这种情况的时候（一个事务跨越了多个节点），我们需要保证事务的ACID特性，需要有一个中间节点来汇总和调节所有节点的状态和进度。这个节点需要掌控整体数据一致性算法的执行进度，并指示这些节点把真正的操作结果进行提交。

下面我们简要描述2PC算法

一阶段：

协调者（中间节点或者主节点）向　所有的参与节点发送请求，询问

是否提交数据操作。

各个参与的节点（从节点）接收到请求，执行事务的准备操作： 为正式提交 进行资源的申请和锁定，写回滚日志等等..

各个从节点 判断事务准备情况 ， 如果准备成功，就回应可以提交，否则回应拒绝提交。

二阶段:

如果所从节点都同意提交，那么主节点向所有从节点发送正式提交的请求。从节点完成正式提交，取消事务所准备提交所占用的资源，回应主节点 已正确完成的信息。

如果有从节点回应主节点拒绝提交，那么主节点向所有的从节点都发送回滚操作 ，从而释放资源，从节点回应回滚完成信息，主节点接收到信息后，取消这个事务。

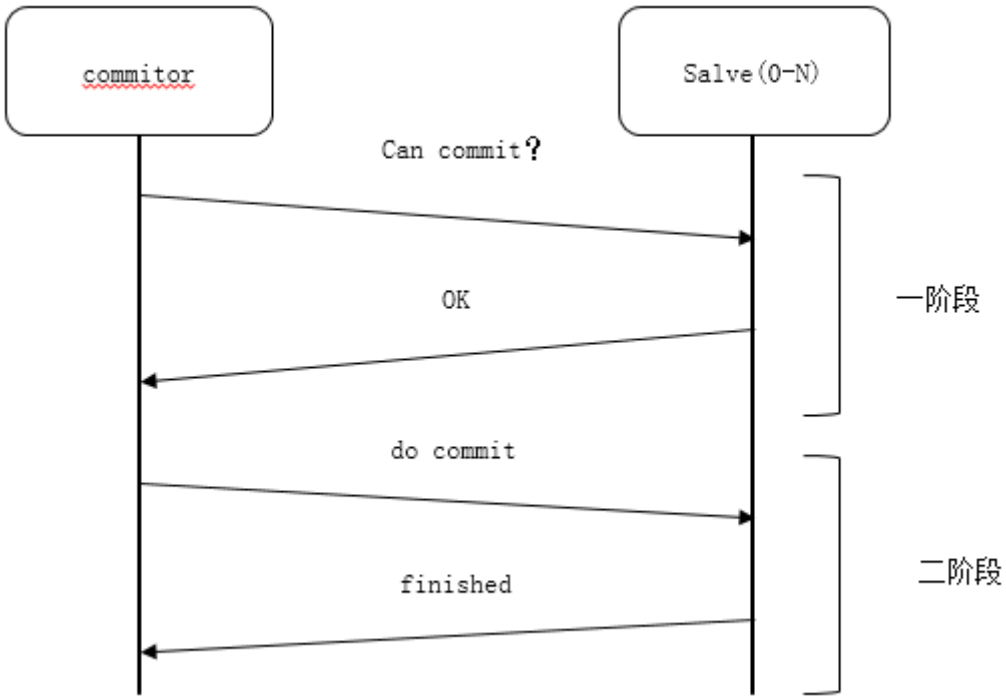


图 4-6

我们可以看到 ，其实2pc协议的第一阶段是去做询问，然后第二阶段来做决定。2pc也是一个强一致性算法。其步骤是先尝试再去做提交。

两阶段算法所占用的资源是比较多的，这个主节点需要等待同步所有从节点的返回值，如失败了，就要回退整个事务中的内容。整体流程比较复杂。

其次是网络连接超时的问题：

在一阶段中，如果发生了超时：

从节点收不到主节点发送的请求，这时候需要主节点进行超时处理，一旦请求超时，可以进行重试。

在二阶段如果发生了超时：

如果正式提交之后，从节点没有收到正式提交的命令，一旦从节点的返回超时，可以选择重试，也可以选择吧这个节点剔除出从节点。

如果从节点没有收到主节点的请求或者失败事务指令，那么从节点仍然占有这些准备提交的资源。这种情况可能发生在主节点宕机或者网络故障。此时从节点有几种选择，等协调者恢复，或者重发一阶段指令，或者设置主节点的信息超时时间。

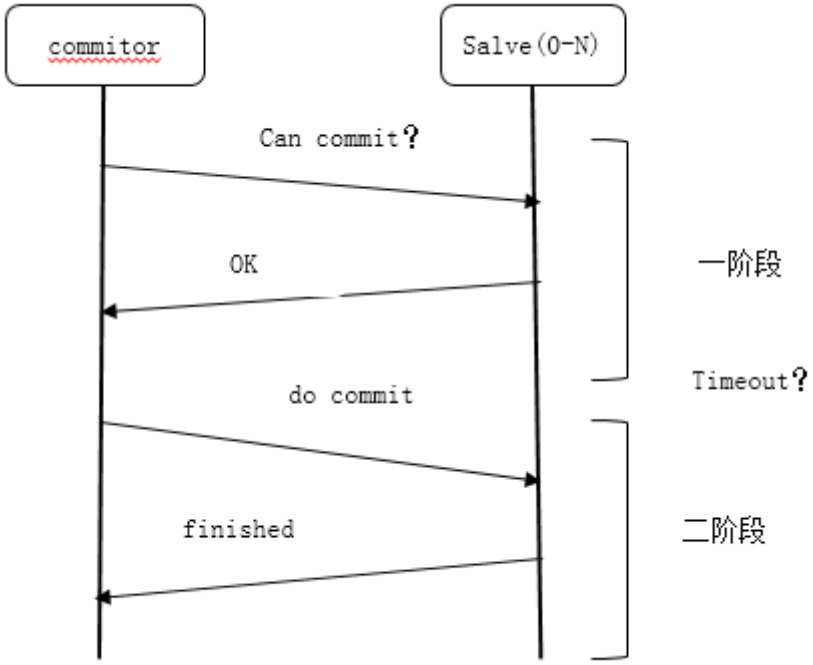


图 4-7

两阶段协议 最大隐患就在于此，如果第一阶段完成后，从节点在第二阶段没有收到决策，那么从节点不知道如何处理已经锁定的数据，并且会阻塞后来的所有对该数据的请求。

换句话说，主节点的可用性在2pc协议十分重要。因此我们可以引入三阶段提交算法来改进这一过程。

3PC算法的核心思想是 将 2pc 中第一个阶段分成两个阶段，先询问，得到结果再去锁定资源。询问的时候并不锁定资源，所有人同意，并将这个结果传递给了其他节点 ，才锁定。

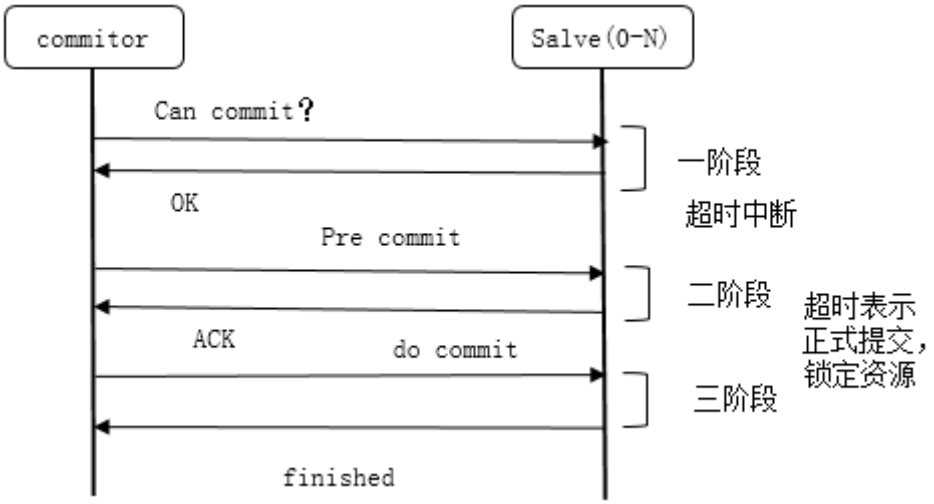


图 4-8

这样一来如果所有其他的从节点都同意提交，主节点才回去发送请求锁定资源的命令，这时如果主节点如果宕机，从节点会根据超时自动提交完成事务。

4.3.3 Paxos 协议（Basic Paxos）

在1975 年E. A. Akkoyunlu 和K. Ekanadham 发表论文中[6]，阐述了一个有意思的问题，两个黑帮 希望能够合伙拿下一个目标， 然而他们之间的通信是不可靠的。这个目标必须两个黑帮同时进攻才能拿下，现在他们需要沟通。这个有意思的问题在后来逐渐演变成有趣的两将军问题，之后又演变成更为困难的拜占庭将军问题：拜占庭王国疆域辽阔，为了防御，各个将领之间为了防御，距离很远。那么在现在要攻打另一个国家，

必要需要所有的将领同时去攻打才能拿下这一国家。各个将领之间需要信使互相交流。然而军队中存在叛徒和间谍。他们可能会误传信息通风报信。在这种情况下，如何才能打败别的国家。

这个问题的核心是我们需要在通信不可靠的情况下，保持一致性。

Paxos 协议[2]很大程度的能够解决分布式系统中这种一致性问题

Paxos 算法希望能够把所有请求都按照一定编号的进行编号，如果编号成功，那么所有请求都按照编号请求的顺序进行执行。这个编号如何选取，则是根据所有节点的选举制度。

首先，我们把所有节点分成了三种角色：

Proposer 提交者

Acceptor 审议者

Learner 学习者

proposer负责发送请求提案给acceptor ，acceptor负责审核提案，learner负责将已经通过的决议进行执行和处理。

提案的内容有：编号，决议内容，编号具有唯一性。

每个节点不分主从关系，并且每个节点可以同时是三种角色。

我们先简述这个算法的过程[3]：

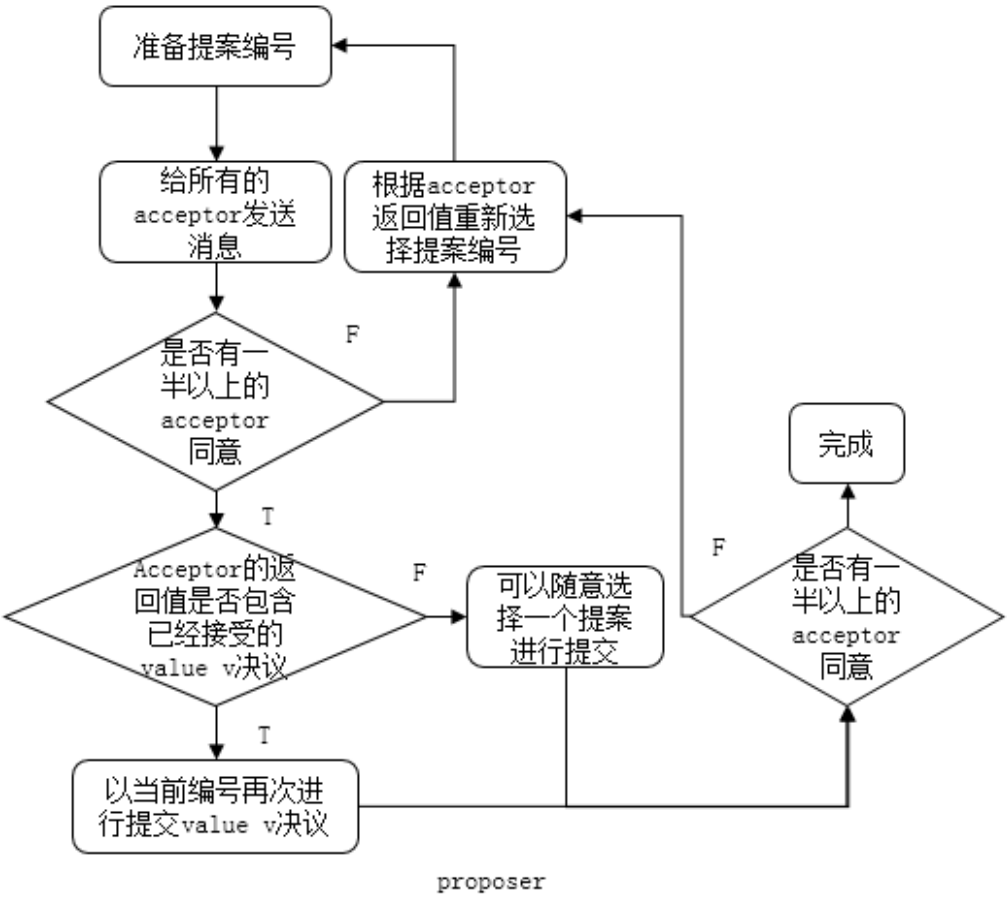

准备提案编号
给所有的acceptor发送消息
根据acceptor返回值重新选择提案编号
是否有一半以上的acceptor同意
F
T
Acceptor的返回值是否包含已经接受的value v决议
F
可以随意选择一个提案进行提交
T
以当前编号再次进行提交value v决议
是否有一半以上的acceptor同意
F
完成
proposer

图 4-9

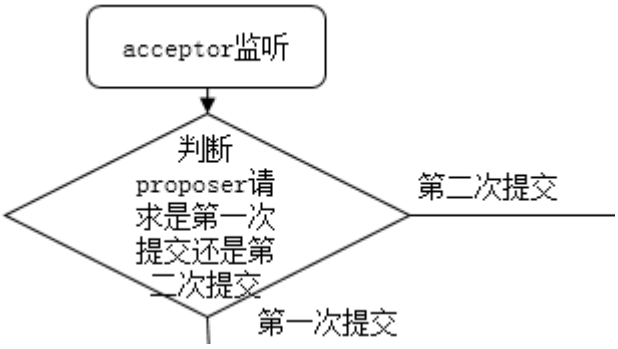

acceptor监听
判断proposer请求是第一次提交还是第二次提交
第二次提交
第一次提交

图 4-10

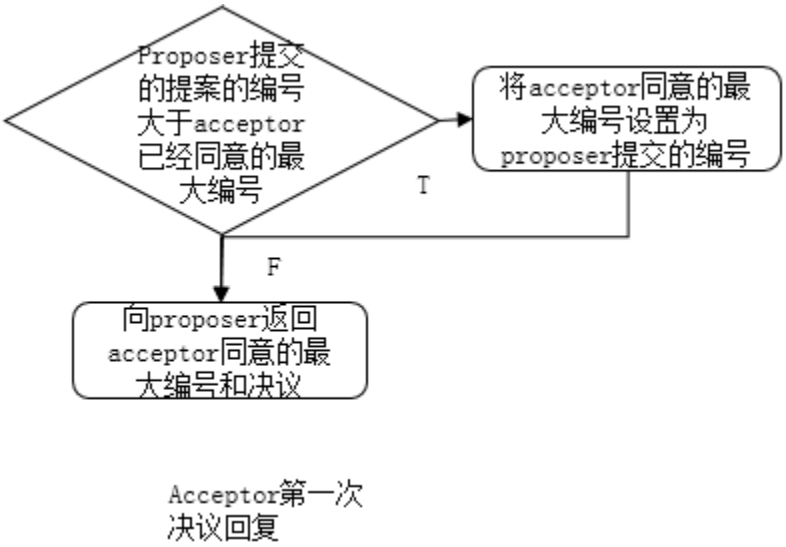


图 4-11

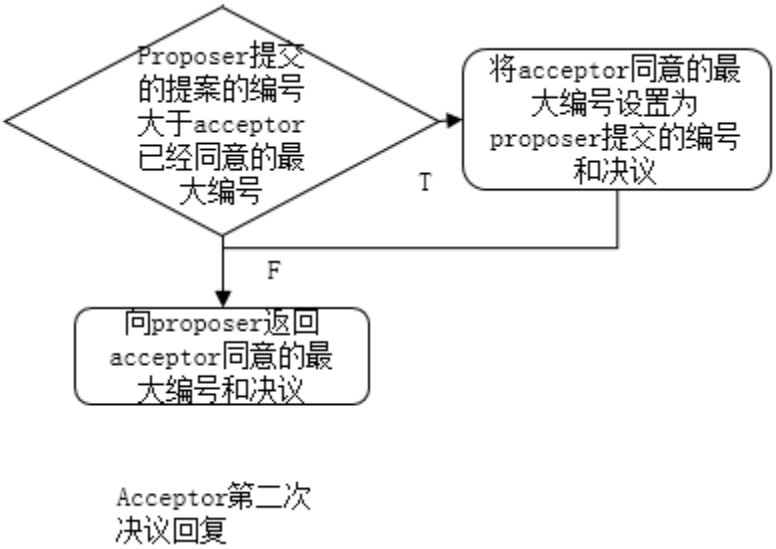


图 4-12

Prepare 阶段

a.proposer 提交者会选择一个新的proposal编号n，发送给acceptor接受者的所有人，当然也可以是大部分人，只要多数派回复消息即可。

b. acceptor接收者如果 发现编号n是它已知的编号最大的请求，他会记住这个编号，同时回信息，接受者已经接收的最大编号和对应的值（如果没有就是空），这时接收者不会再接受小于编号n的请求

Accept 阶段

a.假如proposer提交者接收到了多数接收者发给他同意的消息，那么他会发送正式的接受信息(编号为n，value v的proposal）到所有acceptor接收者。

最重要的是value v，如果acceptor接收者给提交者返回的值中包含了value不为空，那么就取返回中编号最大的value作为v；如果回应中不包含value为空，那么提交者可以自行选择一个值。

b.acceptor接收到accept消息后check，如果没有比n大的回应比n大的proposal，则accept对应的value；否则拒绝或不回应

关于多数派：由于选举制度的存在，paxos 算法需要一半以上的节点同意某个决议， 所以所有运行paxos 算法acceptor 节点个数应为奇数2N+1个，如果出现偶数个节点，没有办法分辨多数派的存在。同样如果如果acceptor宕机超过N个，由于无法形成多数派，所以算法无法继续执行。

我们可以通过一个例子来模拟这个算法：

古罗马城邦提交决议到长老院。

有2个公民和3个长老：

公民有权利提交决议给长老院。长老院审查后可以通过。

公民老王 希望罗马公民可以免税。

公民老李 希望去攻打希腊。

长老对待所有提案的态度都是，只要编号比我已经同意过的大就同意。

老王去找了三个长老，表述自己愿意提交第5号提案，免税提案。长老A、B都同意了这一提案，然而长老C发现老李已经先到他这里提交了6号提案，攻打希腊。于是回绝了5号提案。然而没关系,老王的5号提案已经被多数的长老认可，可以开始第二阶段的提交。

这时我们再来看老李，它的6号提案已经被长老C认可这时长老A,B发现这个6号提案编号更大，也能被接受。于是老李也可以第二阶段的提交。

老王现现在开始第二次提交，当他把提案交给三个长老时，长老发现已经有了6号提案，便不再同意5号提案。并将有新的提案的消息告诉了老王。

老王觉得不服气，便准备提交7号提案，还是希望免税。从头开始提交提案。

这时老王的新提案还没有递交到三位长老手中， 老李第二次提交被三位长老同意了。三位长老将6号提案设为将被执行的决议，交由行政处理执行，准备攻打希腊。老李收到三位长老多数派的回复的消息非常开心

老王的新提案交到了三位长老手中，由于7号提案大于6号议案，三位长老一致同意，并发布同意消息给老王，并把等待行政处理的决议告知了老王，老王接到多数派同意的消息，立即开始准备再次提交，然而发现 6号决议还在等待被行政执行，老王觉得应该先处理这个未完成的决议，把自己的7号提案的内容改成和6号决议一样，交给长老第二阶段表

决，三长老看到后也马上同意了这一过程，这样行政要执行的任务是7号决议：攻打希腊。老王受到多数派消息后也十分开心。

决议过程就这么结束了，老王的免税计划被取消了，然而保证了三位长老的对决议的同样认可。

4.4 主从选举算法

4.4.1 Paxos 原始协议的问题

Paxos 协议算法看上能够解决分布式数据一致性的问题，但是Paxos算法本身还有很多小问题需要解决：

1. 唯一编号的问题：

通过上面的paxos 算法描述，我们知道， 每一个提案都要有自己唯一的编号，并且这个编号需要不停的递增，来满足决议的不停选举。

单机上的唯一编号比较好实现，在不同机器上的唯一编号则不要不停的同步。

一旦提案在一阶段失败就需要重新选择编号。一般来说，不建议直接在编号上面直接+1，因为很可能存在其他节点也在选举。+1操作可能需要很多次的递归才能完成一阶段的提交。

Proposer提交人能够从两方面来确定自己应该提交的编号 ：提交人自己提交成功的自增长，或者接收者发回来的内容最大编号。

2. 活锁：

同时两个proposer 都处于一阶段的提交状态,有相当大的可能性两个人

都无法得到多数派的accept ，导致编号不停的上升。重复不停的提交，造成死循环活锁。

最好的解决办法是吧所有的提案　都通过Leader来提交，如果Leader主节点宕机的话，就马上再次选举其他的主节点。

主节点可以通过队列的方式控制提案的提交，保证编号排序。

3. Leader选举：

Paxos 算法本身就是靠选举来进行的。Paxos 需要Leader 对提案进行控制，而leader也需要选举出来。我们不可能再递归使用paxos来选举。需要一个新的算法来解决这个问题。PaxosLease算法可以解决这个问题。

4.4.2 PaxosLease协议

PaxosLease[4] [10]是Keyspace开发的基于Paxos、Lease[9]的Master选举算法。 基本想法和paxos一致，但是有以下几个区别：

不存在Leader。

选举的速度可以很慢。

需要引入随机时间。

我们需要引入以下变量：

T：每个Master的Lease时间（秒）

M：全局Lease时间，要确保 M > T

ballot number：每次发送的proposal编号

说明：

Timer1等待时间T仅运行于proposer，任何一个Node的Timer1超时便发起prepare请求

Timer2等待时间T仅运行于acceptor，如果超期则清空本次选举状态

Timer3等待时间T1 ＜ T，仅运行成为主节点的节点，目的是在成为主节点的状态 超期之前续租

下面我们简要说明下算法的基本步骤：

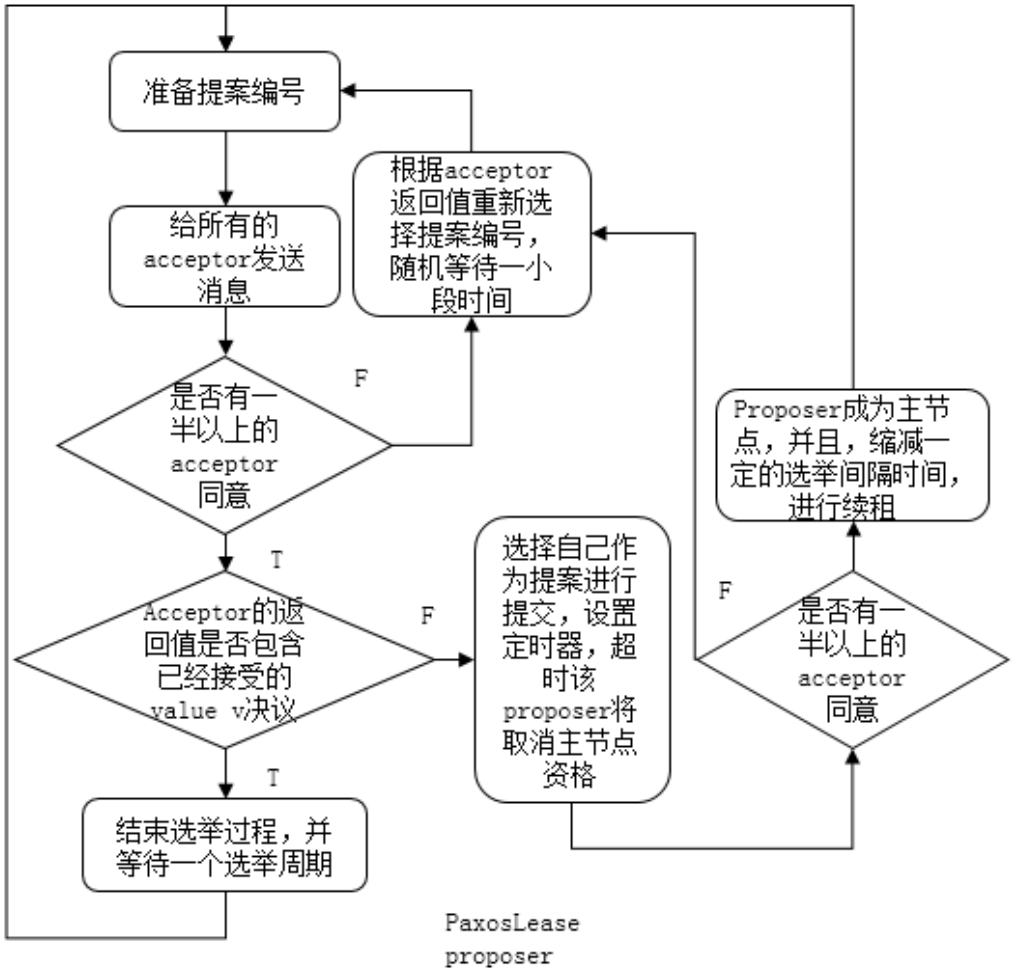


图 4-13

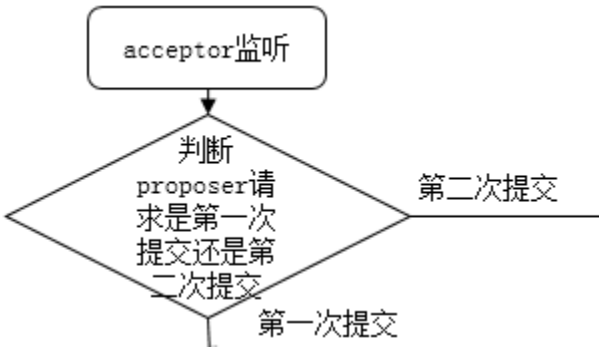

acceptor监听
判断proposer请求是第一次提交还是第二次提交
第二次提交
第一次提交

图 4-14

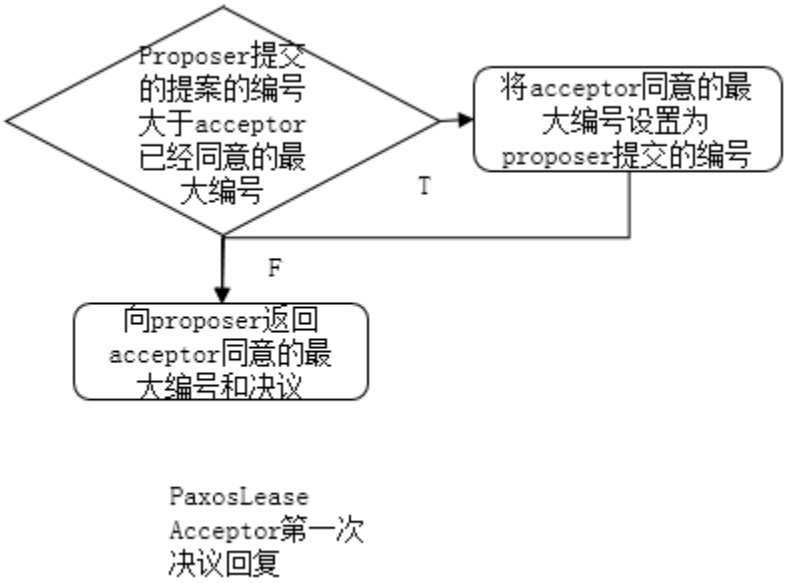
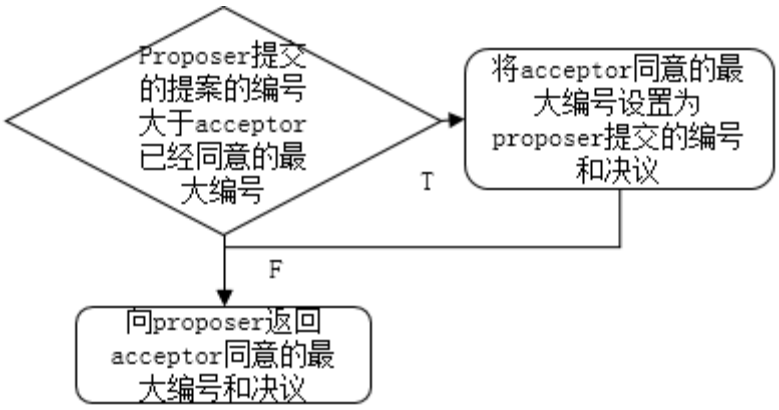

Proposer提交的提案的编号大于acceptor已经同意的最大编号
T
将acceptor同意的最大编号设置为proposer提交的编号
F
向proposer返回acceptor同意的最大编号和决议
PaxosLease Acceptor第一次决议回复

图 4-15



PaxosLease Acceptor第二次决议回复

1.proposer启动定时器Timer1，等待T秒便超时

2.proposer向acceptor发送(ballot number，proposer id，T）

3.acceptor接收到prepare消息后，判断：

如果接受proposer的编号小于本地已经accept的编号，则发送拒绝消息

否则，发送accept，包含已经确定最大编号和T

4.proposer接收acceptor的response

如果多数派accept，则进入promise阶段

否则，随机等待一段时间，提高编号重启prepare过程

5 proposer执行promise阶段

如果prepare阶段接收的value不为空，则终止promise

否则，发送(ballot number,proposer id ,T）

6.acceptor接收到promise请求

如果proposer的编号小于本地已经accept的编号，则回应拒绝消息

否则，启动定时器Timer2，等待T秒超时

7.proposer接收acceptor的response

如果接收到多数派的回应

删除Timer1

启动extend Timer3，等待时间T1 < T（停止Timer1，重新启动Timer1）

否则，重启prepare过程

我们可以看到，paxoslease算法和paxos在二阶段失败的处理有所不同：

Paxos 算法中，如果二阶段接收value v不为空 那么直接用这个v 作为value 来进行提交。

Paxoslease算法中，则是自己退出选举，不需要再去选举他人。

对于Paxos一阶段选举失败产生的活锁问题，PaxosLease算法的就是在一阶段提交失败的情况下 通过随机数，等待一个随机时间，再次重新触发选举算法 。这样的话每个节点面对一阶段提交失败都能通过不同的随机时间避开相互影响。。Paxos是否也可以这样做？ Paxos要求的一致性选举的频率高、延迟低，一阶段失败随机等待几秒这种办法会引发系统延时和性能问题。而Master选举频率低，所以可以忍受。

PaxosLease 有它的缺点：

1、 宕机重启后的情况

我们知道PaxosLease 的内容是不要持久化的。如果某一节点宕机后重启又立即加入了选举节点，可能会扰乱leader选举。我们举例说明：

现在有三个节点A B C ，他们的最大accept的编号是 A=2 ,B=2,C=2，这时

编号为（4，V1）的提案成功。

此时 A,B同时宕机，重启后初始化A=1,B=1,C=4 。如果此时出现编号为(3,V2)的决议，则会被接受，不符合算法的初始条件。

我们可以让宕机重启的节点在M秒之后再加入选举，这样能够进行至少一轮的学习。

2、无法按照权重分配节点

由于PaxosLease选举算法的限制，没有办法进行负载均衡。

4.4.3 Raft协议

除了上面介绍的Paxos算法可以解决选举问题之外，Raft协议[8]也是一种不错的的选择，因为我们不可能保证集群内所有机器都保持同一内容，那么我们希望能够大部分机器满足这一条件即可，已容错的方式解决这一问题，只要有所有节点一半以上的活节点统一意见就可以了，这点和Paxos类似。不同的是选举的具体过程和步骤。

我们可以把Paxos和Raft的选举过程看成是现实生活中的国家竞选。Raft协议包含候选人这一选项，每个节点都可以认为自己是候选人，向其它节点征集选票。而如果此时主节点存在，则会广播协议，禁止候选人拉选票的选举过程。一旦主节点失败，那么就可以继续选举。只要拿到

总人数一半以上的选票就可以成为主节点。当然，如果有两个人同时都是候选人，他们可能都拿不到50%以上的选票，那么他们可以选择重新开始再次选举，类似大选初选未胜出，进入复选环节。这样不停的进行。最后一方赢得选举。

第5章　程序设计

5.1 算法选择

基于Java web架构的分布式缓存，我们选择的是数据分区+数据镜像。Paxoslease选举主节点，主节点持有一致性hash哈希算法圆环。 每个一致性hash上的虚拟节点，都是多个实际服务器节点，这些节点之间组成多个数据镜像，存储相同的内容进行复制。复制的算法采用2PC两阶段提交算法。内存缓存使用LRU最近最少使用算法来进行更替。整体通讯使用HTTP 协议来进行交互。

这里会有一个问题，我们的主节点持有一致性hash 的内容，一致性哈希圆环上的节点都是我们随机的虚拟节点，当主节点宕机时 ， 需要将这些虚拟节点转移给下一个主节点。

所以当我们进行PaxosLease算法时，会把上一个节点生成的虚拟节点列表发送给acceptor进行保存。

这样的话，一致性hash算法就不存在节点失效的可能性，因为每个一致性hash 圆环上的节点都是虚拟节点，这些节点都是由若干个真实节点组成。而我们再启动所有的服务器之前就知道实际节点的个数，所以不存在节点的添加。

5.2 程序模块设计

网络通信模块使用的是HTTP通信， HTTP 协议作为应用层协议 ，底层使用TCP 可靠网络传输协议，保证网络通信的可靠性。HTTP作为应用层协议，被广泛使用在Web 服务和浏览器中。方便我们的调试和排查错误。

我们通过PaxosLease协议来选举主节点，两阶段提交算法来控制提交。保证提交内容的一致性。一致性哈希算法控制数据的分区。每个节点使用日志和LRU内存缓存来缓存对象。

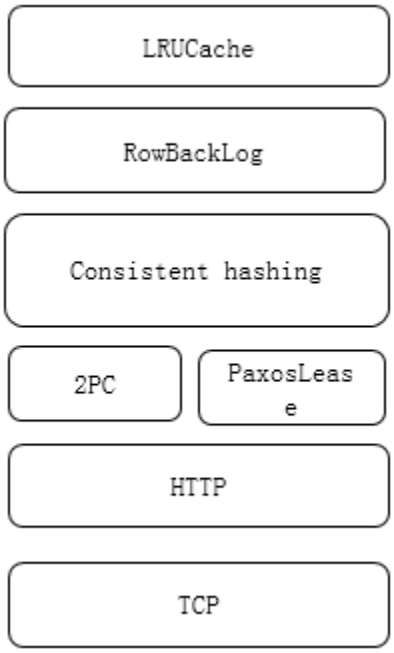


图 5-1

我们将系统进行模块划分，分为四个模块服务：缓存服务、数据分区服务、选主选举服务、选主监听服务。每个服务器节点都可以选择是否开启这些模块。由于数据分区服务是由主节点完成的，所以要判断主节点能否变成主节点进行开启。

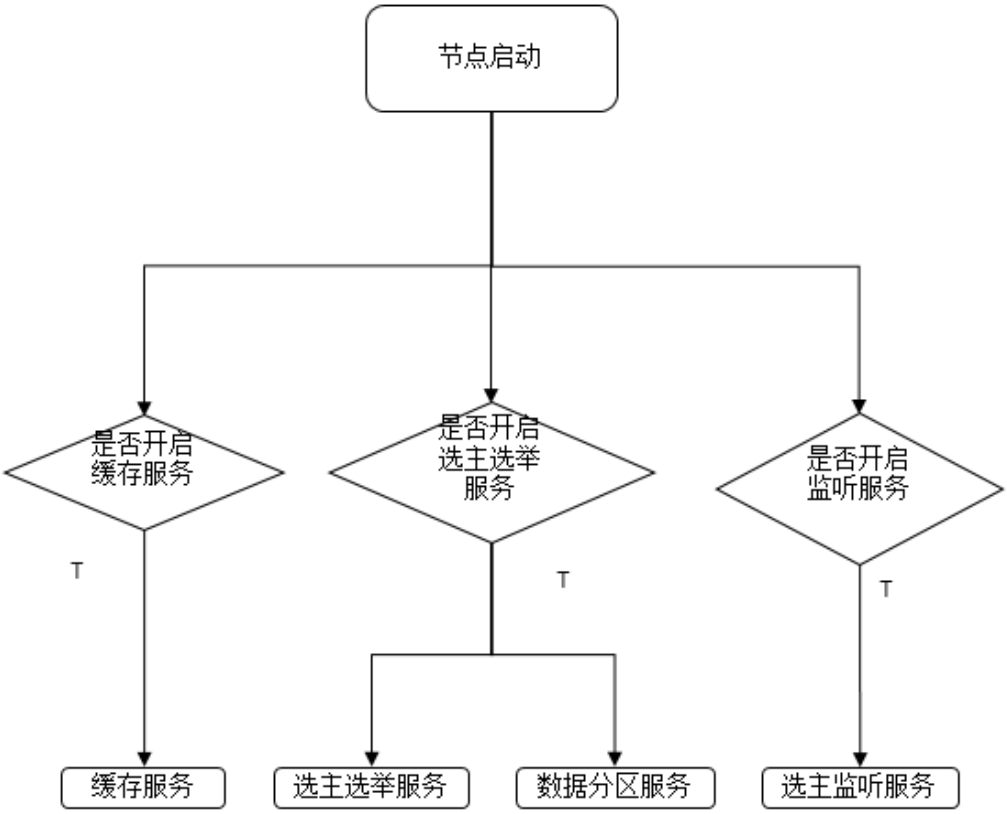

节点启动
是否开启
缓存服务
是否开启
选主选举
服务
是否开启
监听服务
T
T
T
缓存服务
选主选举服务
数据分区服务
选主监听服务

图 5-2

第6章　程序实现

6.1 文件结构和用途

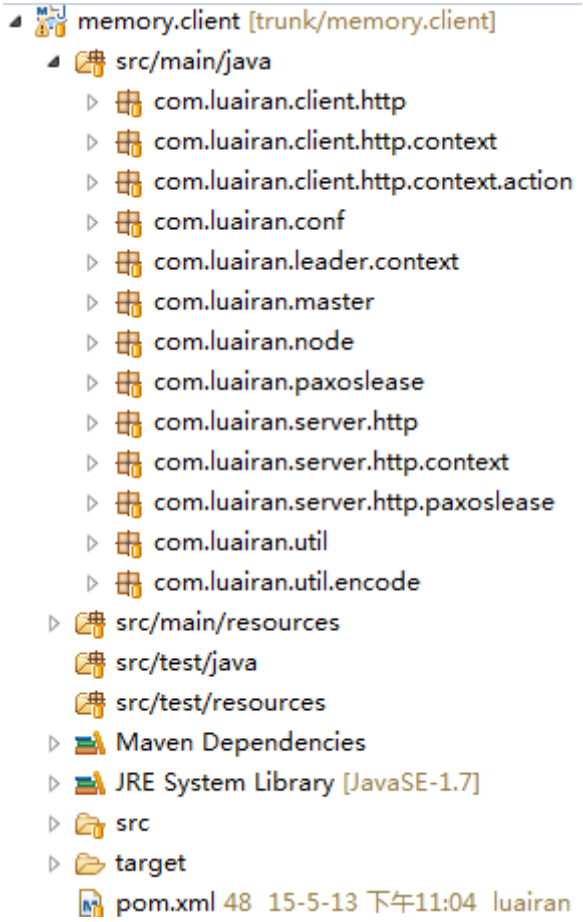


图 6-1

这是一个标准的Java EE Maven 工程，和普通Java 工程不同，Maven 工程拥有四个src文件夹，并且工程不需要导入jar包，我们只需要在pom.xml中填写对应的依赖即可。

表 6-1

src/main/java		主目录
	com.luairan.client.http	htt客户端连接工具
	com.luairan.client.http.context	客户端接入
	com.luairan.client.http.context.action	客户端具体实例
	com.luairan.conf	配置文件
	com.luairan.leader.context	选主实例内容
	com.luairan.master	主节点配置
	com.luairan.node	内存缓存内容
	com.luairan.paxoslease	选主算法工具
	com.luairan.server.http	http服务器工具
	com.luairan.server.http.context	http服务端具体服务实例
	com.luairan.server.http.paxoslease	http服务端具体选主服务实例
	com.luairan.util	工具类
	com.luairan.util.encode	加密以及编码包
src/main/resources		配置文件包
src/test/java		测试目录
src/test/resources		测试文件包
pom.xml		maven jar 依赖文件

Pom.xml 中我们需要的jar的名称 从maven仓库找到，填写相对应的groupId、artifactId、version、scope。项目环境的引jar包Maven会自动班

帮你完成。

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
        <artifactId>junit</artifactId>
        <version>4.10</version>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.slf4j</groupId>
        <artifactId>slf4j-log4j12</artifactId>
        <version>1.7.7</version>
    </dependency>
    <dependency>
        <groupId>com.googlecode.concurrentlinkedhashmap</groupId>
        <artifactId>concurrentlinkedhashmap-lru</artifactId>
        <version>1.4</version>
    </dependency>
    <dependency>
        <groupId>com.sun.net.httpserver</groupId>
        <artifactId>http</artifactId>
        <version>20070405</version>
    </dependency>
    <dependency>
        <groupId>dom4j</groupId>
        <artifactId>dom4j</artifactId>
        <version>1.6.1</version>
    </dependency>
```

</dependencies>

我们需要的外部jar 包有:

Junit

org.slf4j

concurrentlinkedhashmap

com.sun.net.httpserver

dom4j

Junit是java环境下常用的项目代码测试工具，我们可以使用它进行系统集成测试。Slf4j是一款常用的文件日志系统，我们可以通过它来记录对应的日志。Concurrentlinkedhashmap是google开源的一个Java下HashMap的实现，具有LRU最近最少使用算法，并且能够在并发环境下使用。com.sun.net.httpserver是一款极为轻量级的HTTP server 实现，相比标准的Java EE 下的Tomcat 和Jboss HTTPserver，它没有包含Java EE 下标准的Java Servlet规范，但是可以更好的嵌入开发，降低整体服务器的对于性能要求。Dom4j是java下用来解析标准dom xml文件的工具。

6.2 配置文件以及初始化

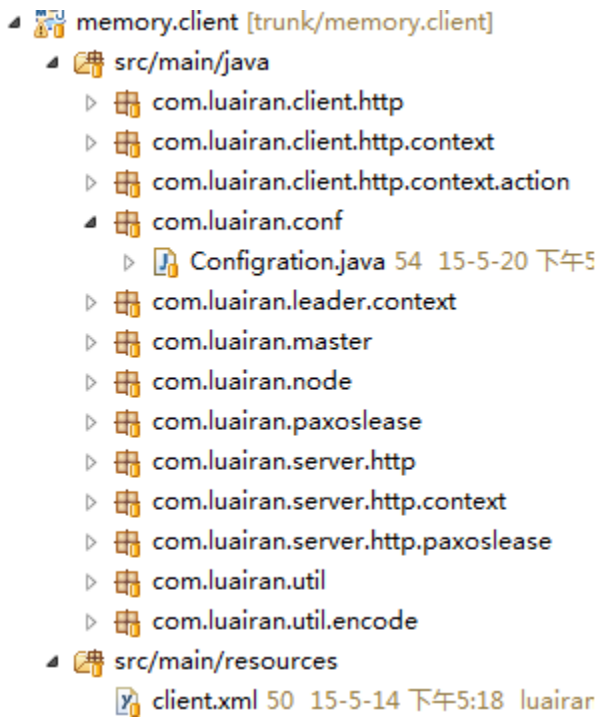


图 6-2

配置文件client.xml在src/main/resources目录下，我们可以通过该配置文件控制。

对应的，我们需要从程序中把xml文件中的内容读取到Java文件中,Configration.java 会将对应xml内容在程序开始执行之前载入到内存中。

6.3 主节点选举算法

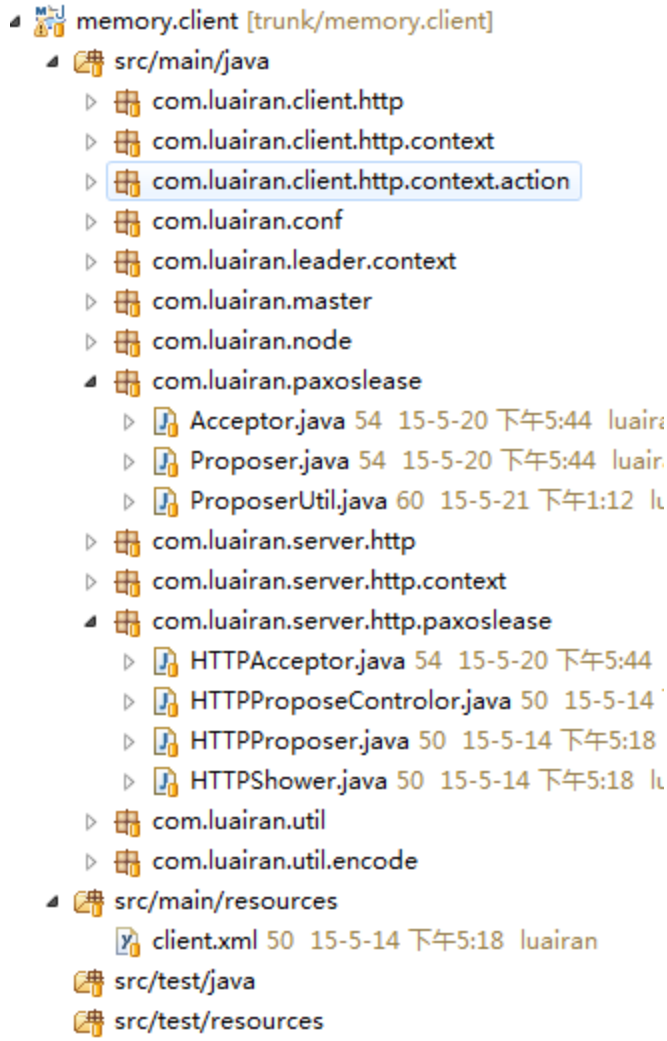


图 6-3

Acceptor.java 作为PaxosLease算法中的 acceptor 。Proposer.java 则代表一个提案，我们可以使用ProposerUtil.java 来不停的进行选举过程。HTTPAcceptor和HTTPProposer 作为触发器，用来接收请求。HTTPProposeControlor.java 和HTTPShower.java则是展示和控制整个的选举过程

```
public void proposerOnce() throws InterruptedException {
        CountDownLatch proposeOne = new CountDownLatch(urlList.size() / 2 + 1);
        CountDownLatch proposeTwo = new CountDownLatch(urlList.size() / 2 + 1);
        Proposer proposer = new Proposer(state, scheduler, proposeOne, proposeTwo,inetSocketAddress,singleNode,consistentHashNode.getConsistentHashLock());
        // 执行paxoslease 第一个阶段
        Request request = proposer.propose();
        for (String url : urlList) {
            ProposerOne popser = new ProposerOne(url+paxosAcceptor, proposer, request);
            executor.execute(popser);
        }
        boolean onestep = proposeOne.await(waitSeconds,TimeUnit.SECONDS);
        if(!onestep){
            Proposal proposal = proposer.getProposalOne();
    if (proposal != null && !proposal.getSingleNode().getAddress().equals(singleNode.getAddress())){
                scheduledFuture.cancel(false);
                scheduledFuture =scheduler.schedule(new TimePropose(),realProposeSeconds, TimeUnit.SECONDS);
                return ;
            }
            scheduledFuture.cancel(false);
```

```
            scheduledFuture =scheduler.schedule(new TimePropose(),
random.nextInt(oneFailRetrySeconds), TimeUnit.SECONDS);
            return;
        }
        if(scheduledFutureProposer!=null)
scheduledFutureProposer.cancel(false);
        request = proposer.prepareRequestOne();
        scheduledFutureProposer = proposer.getScheduledFuture();
        for (String url : urlList) {
            ProposerTwo proposerTwo = new
ProposerTwo(url+paxosAcceptor, proposer, request);
            executor.execute(proposerTwo);
        }
        boolean twostep =
proposeTwo.await(waitSeconds,TimeUnit.SECONDS);
        if(twostep){
            scheduledFuture.cancel(false);
            scheduledFuture =scheduler.scheduleAtFixedRate(new
TimePropose(), successReProposeSeconds, realProposeSeconds,
TimeUnit.SECONDS);
            proposer.proposeResonseTwo();
        }else{
            scheduledFuture.cancel(false);
            scheduledFuture =scheduler.scheduleAtFixedRate(new
TimePropose(), 0L, realProposeSeconds, TimeUnit.SECONDS);
        }
    }
```

上面代码是正常的一次选举过程的执行步骤。该步骤和PaxosLease流程中实现一致。

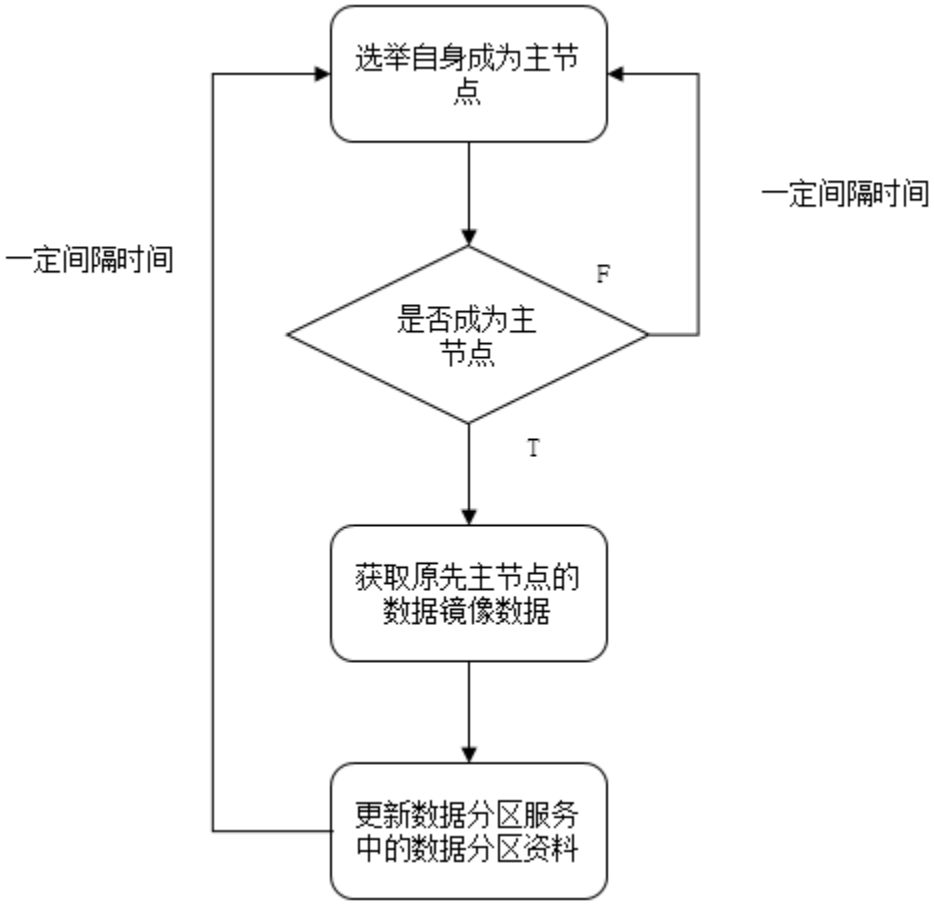


图 6-4

6.4 一致性哈希和数据冗余算法

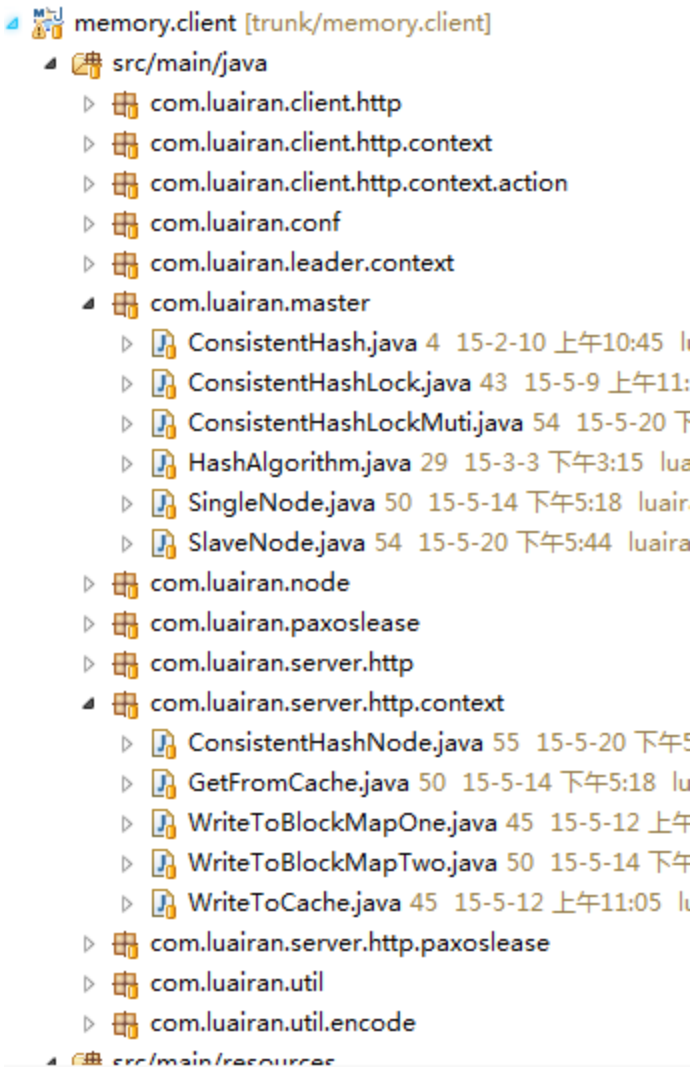


图 6-5

ConsistentHash.java是对应一致性hash 的Java 接口ConsistentHashLock.java ConsistentHashLockMuti.java 两个文件都是一致性hash的具体实现。在我们的程序中，都可以使用。HashAlgorithm.java 是对应的hash算法，我们可以使用多种多样的hash算法来保证散列充分。 SingleNode.java 表示真实的一个服务器节点SlaveNode.java 表示在一致性哈希圆环上的节点，这个节点可以由实际上的很多真实的服务器节点来组成。

下面对应的则是HTTP请求模型。

6.5 内存缓存LRU算法

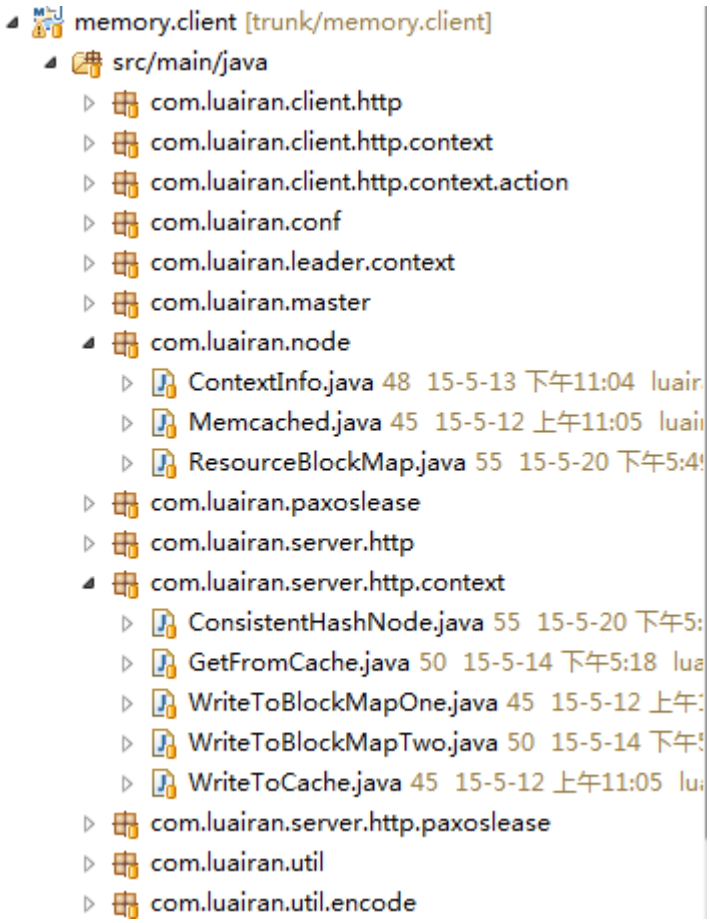


图 6-6

Memcached.java 文件是对应的内存缓存hashMap ，采用google开源的并发LRUHashMap。

我们在服务器实现代码的时候要注意多线程的问题。一般来说如果是单线程来用作缓存的是java自带类库中标准的HashMap，这个类是通过每个类的自带的hashcode方法计算对应每个对象的哈希值，从而使用key-value 的方式进行存储，如果对应的产生的hash key 值相同的话，采用链

表的方式将相同hash key 的不同对象进行存储。

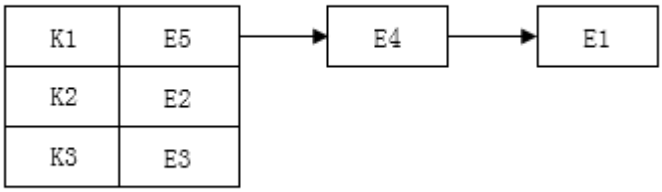


图 6-7

如图，E1、E4、E5　　对应的hash值相同，并且存放先后顺序为 E1、E4、E5。

但我们的HTTP使用的是多线程访问方式，来提高吞吐量。而HashMap并不是一个线程安全的哈希容器。当不同线程操作这个Map对象时可能会产生错误。

简单的来说，在java 中如果希望把一个线程不安全对象转换成线程安全的对象，最简单的办法就是加锁。这样可以保证每个线程同时只有一个对Map进行访问。但是这样做会导致严重的性能问题。

在jdk6版本以后我们有了更好的解决办法，使用ConcurrentHashMap 来进行多线程下的访问问题。ConcurrentHashMap解决该问题的办法采用的是锁分段技术。即我们把hashmap分成很多子HashMap片段，每个子片段使用加锁或CAS锁。

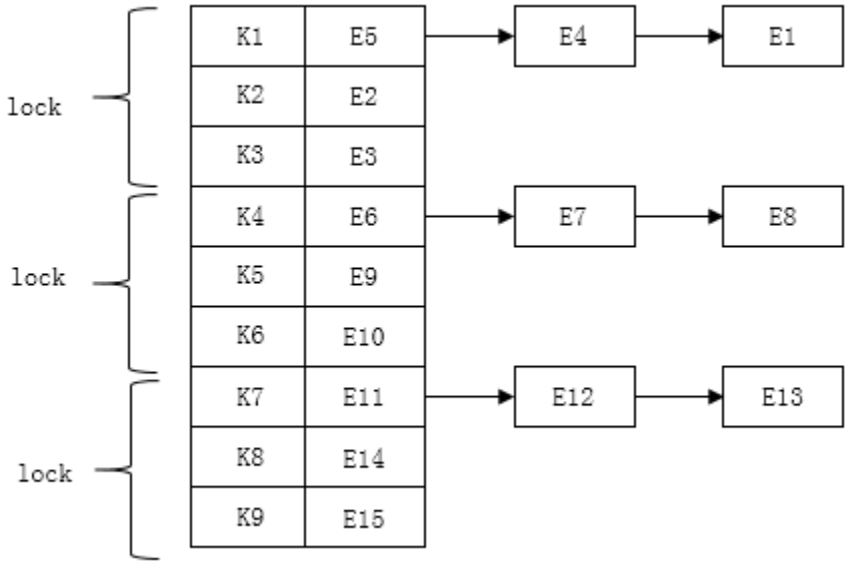


图 6-8

作为一个内存缓存来说，还要控制自己的缓存容量，LRU算法作为缓存中常用的算法，有很多优点：易于实现，能够很好的控制容量，能够根据使用频率调整。LRU最少使用算法，是根据每个元素的使用频率来进行调整缓存。LRU算法可以通过双向链表来实现，每次修改或者读取元素都是从链表头开始操作，如果超过缓存容量，就删掉链表末端的元素。

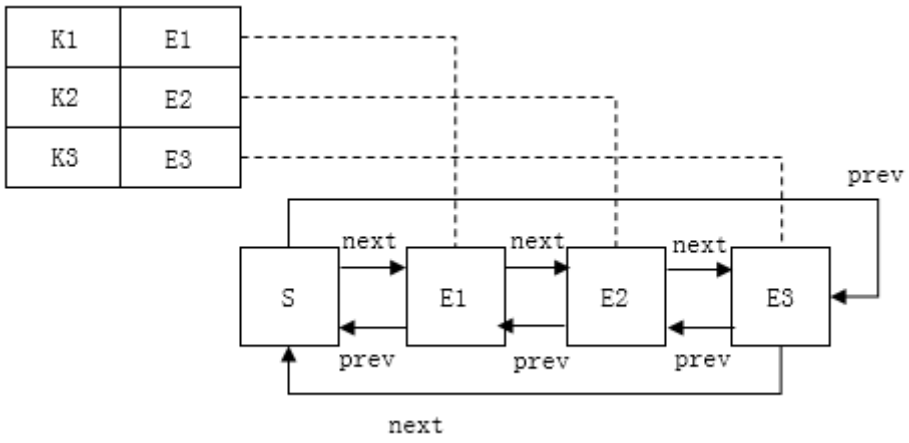


图 6-9

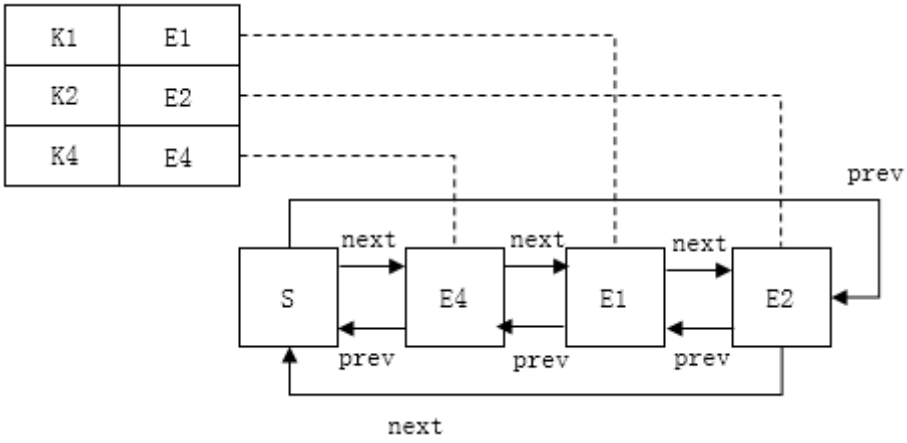


图 6-10

ResourceBlockMap.java则是将Memcached.java进行了封装，适用两阶段提交算法。ContextInfo.java则是真正存储在Memcache中的内容。包含了以下几项：

```
private  byte [] info;
private String md5;
private Long size;
private  String className;
```

上传的所有对象和类都是以byte数组为单位进行存储，这样做的原因是服务端永远不可能知道客户到底会选择上传那些数据结构，上传的内容有可能是Java对象，也有可能是文件和图片。我们在客户端会把对应的Java类和各种对象转化byte数据。

我们会用size 来表示上传的内容的大小，单位是字节Byte。

在比较同一项内容是否相等时，不会直接去比较字节数组中的每个值。

而是去比较每个对象的md5 。只要两个对象的md5 数据标签一致，我们就认为他们是相同的。

WriteToBlockMapOne.java WriteToBlockMapTwo.java 和GetFromCache.java则是对应缓存的HTTP请求类。用来处理 获取信息和两阶段提交。

第7章　程序测试

7.1 系统测试

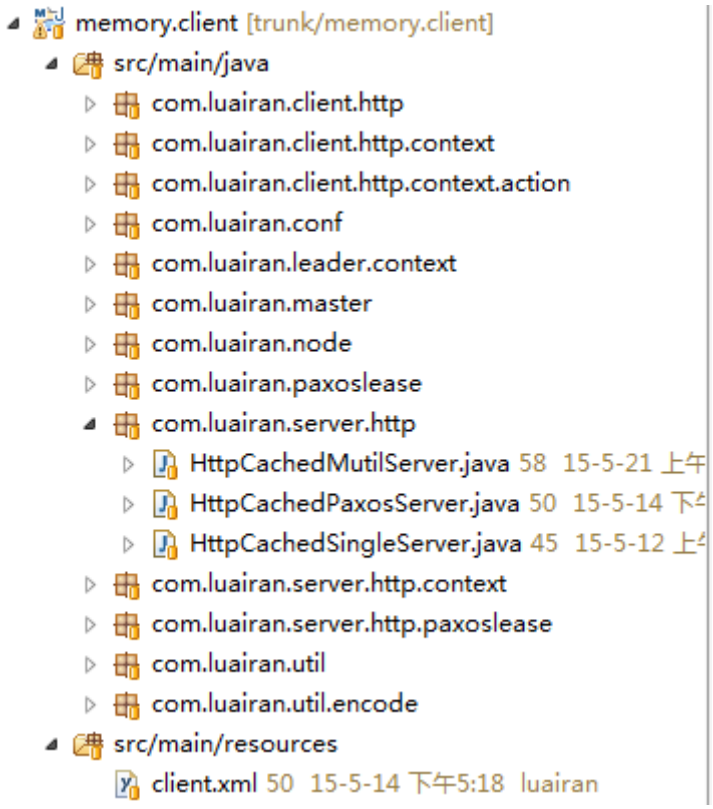


图 7-1

我们通过调用HttpCachedMutilServer.java中的main方法启动程序。

我们在client.xml中配置了两台实体机器在局域网下，一台机器启动四个端口来模拟四个节点，另一台机器启动三个节点。保证总节点数为奇数，保证选举算法的正确执行。

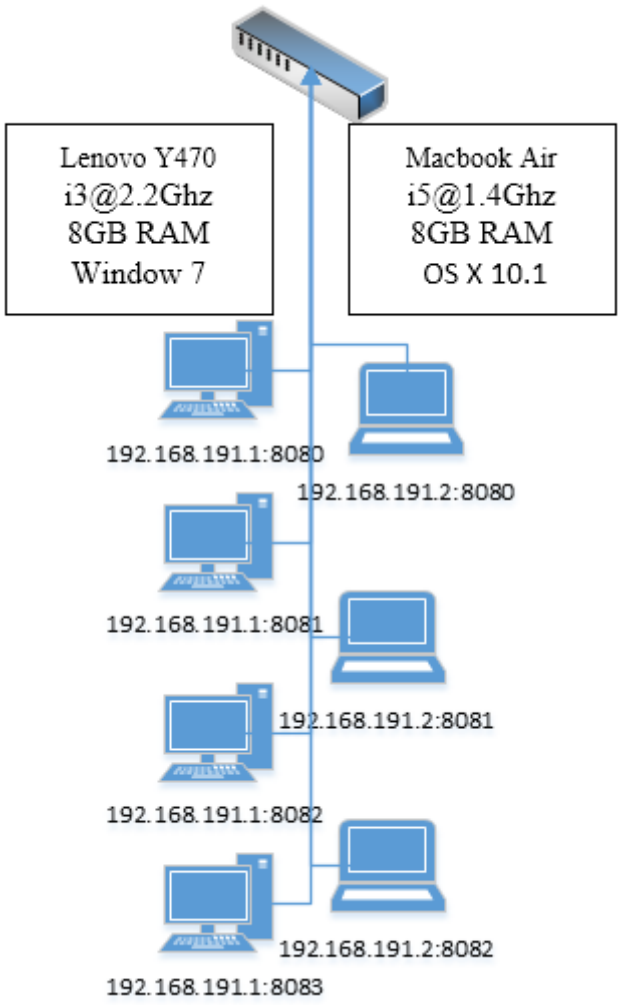

Lenovo Y470
i3@2.2Ghz
8GB RAM
Window 7
Macbook Air
i5@1.4Ghz
8GB RAM
OS X 10.1
192.168.191.1:8080
192.168.191.2:8080
192.168.191.1:8081
192.168.191.2:8081
192.168.191.1:8082
192.168.191.2:8082
192.168.191.1:8083

图 7-2

Thu May 21 15:49:03 CST 2015 /127.0.0.1:8083 PrepareRequest
Thu May 21 15:49:03 CST 2015 /127.0.0.1:8080 PrepareRequest
Thu May 21 15:49:03 CST 2015 /127.0.0.1:8083 PrepareRequest
Thu May 21 15:49:03 CST 2015 /127.0.0.1:8081 PrepareRequest
Thu May 21 15:49:03 CST 2015 /127.0.0.1:8080 PrepareRequest
Thu May 21 15:49:03 CST 2015 0.0.0.0/0.0.0.0:8082 获取null
Thu May 21 15:49:03 CST 2015 /127.0.0.1:8082 PrepareRequest
Thu May 21 15:49:03 CST 2015 0.0.0.0/0.0.0.0:8082 获取null
Thu May 21 15:49:03 CST 2015 0.0.0.0/0.0.0.0:8082 获取null
Thu May 21 15:49:03 CST 2015 0.0.0.0/0.0.0.0:8082 获取null
Thu May 21 15:49:03 CST 2015 /127.0.0.1:8084 PrepareRequest
Thu May 21 15:49:03 CST 2015 0.0.0.0/0.0.0.0:8082 获取null
Thu May 21 15:49:03 CST 2015 /127.0.0.1:8080 ProposeRequest
Thu May 21 15:49:03 CST 2015 /127.0.0.1:8082 ProposeRequest
Thu May 21 15:49:03 CST 2015 /127.0.0.1:8083 ProposeRequest
Thu May 21 15:49:03 CST 2015 /127.0.0.1:8084 ProposeRequest
Thu May 21 15:49:03 CST 2015 /127.0.0.1:8081 ProposeRequest
Thu May 21 15:49:03 CST 2015 0.0.0.0/0.0.0.0:8082成为主节点
Thu May 21 15:49:04 CST 2015 http://127.0.0.1:8081 null
Thu May 21 15:49:06 CST 2015 http://127.0.0.1:8080 null

图 7-3

7.2 系统可靠性

我们使用三台主机的测试环境，每台主机模拟三个节点，运行一周时间:

我们设置的正常选举时间为30s每个节点一次，续租时间为20s一次，paxoslease算法 一二阶段同步汇总时间均为3s, 一阶段失败重试随机时长最大为6s

我们一共监控到50241次选举请求，选举成功17143次。有3个节点一台主机因为电源断电关机，断开连接。但是分布式程序并没有停止执行，主节点仍然存在。

最大的不可用时间为34s，这是因为上一个节点成为主节点后立即被系统端口占用，而下一个节点在一阶段等待重试了几次。导致主节点访问失败。但只出现了一次，根据选举次数的概率，基本不影响系统的使用。

第8章　结论

8.1 结论

通过基于JavaWeb架构的分布式缓存设计与实践该论文，我们总结了分布式领域的数据冗余和数据分区的各种方法，并使用PaxosLease算法解决了分布式主节点单点故障的问题，通过一致性哈希，将数据的高可用和一致性问题进行了解决，通过具体的程序设计，完成了2PC算法、PaxosLease算法、一致性哈希的具体实现。

分布式是计算机领域重要的研究方向，随着晶体管以及工艺的限制，单个处理器的性能瓶颈越发明显，多核cpu以及多台计算并行计算变得越发重要。我个人认为分布式领域未待开发的领域还有很多。我们甚至可以分布式系统抽象成社会，每个人都是一个节点。选主算法就如同选举政府官员。完成并行计算任务就像工人在工厂的流水线一起完成产品。希望论文能为以后的分布式主从节点和数据分配存储研究提供一定的经验。

致　谢

本课题的设计与论文编写过程中，我的指导老师闫昭老师给予了我很大的帮助，帮助我理清题目的难点关键点，提供了大量的相关资料。在完成总体思想上指导也给予了我很多实践上的经验与指导，拓宽了我的知识和思路，使我按时顺利完成该课题的研究。在此我对闫昭的指导表示衷心感谢。

同时，我也衷心的感谢吉林大学软件学院四年来对我的悉心培养和支持，优秀的师资队伍和良好的校园环境让我受益匪浅，浓浓的校园氛围让我难以忘怀。

还要感谢阿里巴巴数据技术及产品部-数据质量团队，在校外实习期间，让我接触到了更多新鲜的技术。同时有机会接触到了大数据和并行计算等等先进的技术资料。

参考文献

1.Leslie Lamport.The Part-Time Parliament.[C]: ACM Transactions on Computer Systems,1998

2.Leslie Lamport. Paxos Made Simple.[A]: research microsoft,2001

3.Tushar Chandra,Robert Griesemer,Joshua Redstone. Paxos Made Live - An Engineering Perspective.[A]: Google Inc,2001

4.Mike Burrows. The Chubby lock service for loosely-coupled distributed systems.[A]: Google Inc,2001

5.Butler W. Lampson. How to Build a Highly Available System Using Consensus. [A]: Microsoft Inc

6.E. A. Akkoyunlu ,K. Ekanadham ,R. V. Hubert . SOME CONSTRAINTS AND TRADEOFFS IN THE DESIGN OF NETWORK COMMUNICATIONS.[C]

7. David Karger, Eric Lehman, Tom Leighton. Consistent Hashing and Random Trees.[A]: MIT,1997

8.Diego Ongaro ,John Ousterhout . In Search of an Understandable Consensus Algorithm. [A] : Stanford University

9.F.Hupfeld et al., FaTLease: Scalable Fault-Tolerant Lease Negotiation with Paxos.[A] :Boston, Massachusetts, USA. June 2327, 2008

10.Marton Trencseni,Attila Gazso,Holger Reinhardt. PaxosLease: Diskless Paxos for Leases .[A]

11.Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall , Werner Vogels. Dynamo: Amazon's Highly Available Key-value Store .[A]: Amazon.com