

SHANGHAIJIAOTONGUNIVERSITY



分布式系统

Distributed Key-Value Storage System

实验报告

学生姓名： 林江浩

学生学号： 517021910674

专 业： 软件工程

指导教师： 吴 刚

学 院(系)： 电子信息与电气工程学院

目录

[1. 软硬件环境 1](#_Toc44610456)

[2. Zookeeper集群的搭建过程 1](#_Toc44610457)

[3. 系统架构与设计 3](#_Toc44610458)

[3.1 Zookeeper的使用 4](#_Toc44610459)

[3.2 基础功能的实现 4](#_Toc44610460)

[3.3 锁与并发 5](#_Toc44610461)

[3.4 持久化存储 6](#_Toc44610462)

[3.5 负载均衡 6](#_Toc44610463)

[3.6 容错性：Master 7](#_Toc44610464)

[3.7 容错性：Data Node 8](#_Toc44610465)

[3.8 可扩展性：Server-Level 8](#_Toc44610466)

[3.9 可扩展性：Group-Level 9](#_Toc44610467)

[3.10 小结 9](#_Toc44610468)

[4. 演示与测试 10](#_Toc44610469)

# 1. 软硬件环境

本次实验，我借助Zookeeper集群服务，实现了一个分布式的键值对存储系统。这个系统面向用户满足强一致性要求，并且通过一个进程模拟一台物理机的单机模型构建分布式运行环境。以下为本次实验的软硬件环境情况：

* 操作系统：Windows 10
* CPU：Intel(R) Core™ i7-6700HQ CPU @ 2.60GHz
* 内存：8GB
* Zookeeper：3.4.14
* 编程语言：Python 3.7
* 使用的第三方工具库：kazoo、xmlrpc
* 使用的第三方包管理工具：Anaconda3

# 2. Zookeeper集群的搭建过程

本次实验中，整个键值对存储系统的主动功能与服务都直接依赖于Zookeeper（如分布式锁服务、服务发现与配置管理等等），因此如何正确搭建一个具有高容错性的多节点（不少于三个）Zookeeper集群是分布式键值对存储系统正常运行的关键前提。Zookeeper集群的搭建可以分为二进制文件下载、修改配置文件、添加myid和启动集群服务四步，接下来我会按步骤进行详细解释。

1. **二进制文件下载**

从[Zookeeper官网](https://zookeeper.apache.org/releases.html)下载对应版本的工具包，并解压至相应文件夹。

1. **修改配置文件**

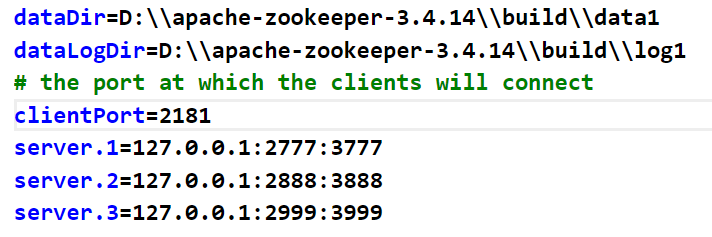
在Zookeeper主目录的conf文件夹下，复制Zookeeper提供的配置文件模板三份，并依次重命名为zoo1.cfg、zoo2.cfg和zoo3.cfg，依次对应之后Zookeeper集群中的三个服务节点的配置文件。接着，依次修改三个配置文件中的配置信息。

首先修改dataDir和dataLogDir，并且根据配置内容在本地手动创建对应对的Data和Log文件夹，需要注意，不同的配置文件应使用不同的文件夹。

然后还应关注clientPort配置项，该端口配置与用户的连接使用息息相关，此处我们采用默认端口2181不做修改。

最后，为了配置集群模式，需在文件末尾添加配置服务器信息server.x，用于表示其他Zookeeper实例端口，使能实例之间的联系。需要注意，上述三个配置文件的服务器配置项内容是一样的。

针对上述三个配置项的修改，具体如下图所示：



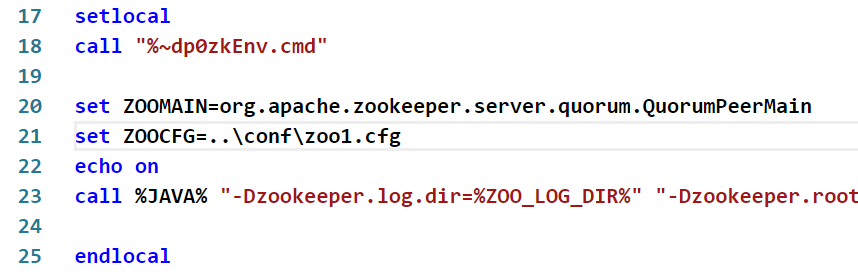
**图2.1 zoo1.cfg的配置文件修改**

1. **添加myid**

找到第二步中配置的三个dataDir文件夹，分别创建名为myid的文件，并写入对应的id（1~3），用于Zookeeper实例在产生联系之后的识别。

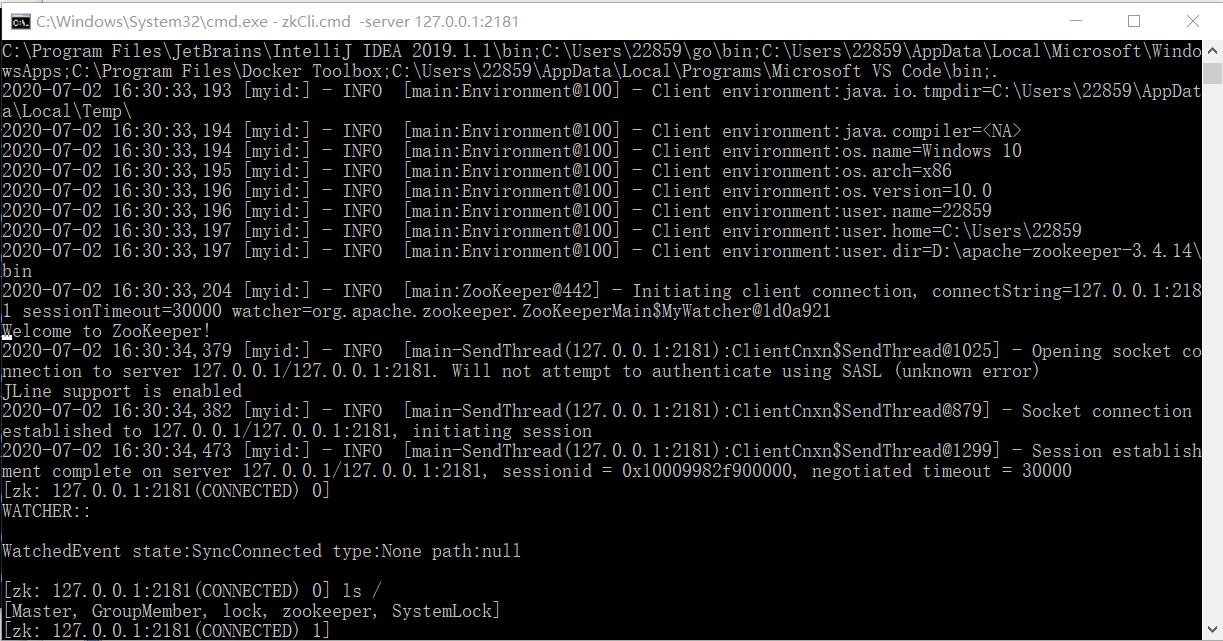
1. **启动集群服务**

因为我们需在Zookeeper集群中启动三个节点，因此首先将Zookeeper主目录下bin文件夹中的zkServer.cmd复制三份，依次命名为zkServer1.cmd、zkServer2.cmd和zkServer3.cmd。这三个.cmd文件与第一步中的.cfg文件一一对应，我们在对应.cmd文件中添加.cfg的路径信息，其他不做改动，具体见下图所示：



**图2.2 zkServer1.cmd的修改**

修改完毕后，依次运行三个.cmd文件，即可启动Zookeeper集群服务。作为验证，另启一个cmd窗口，进入主目录下的bin文件夹，输入命令zkCli.cmd -server 127.0.0.1:2181，即可打开一个连接到Zookeeper集群的客户端，可以进行相应的查看改动操作，具体情况如下图所示：

****

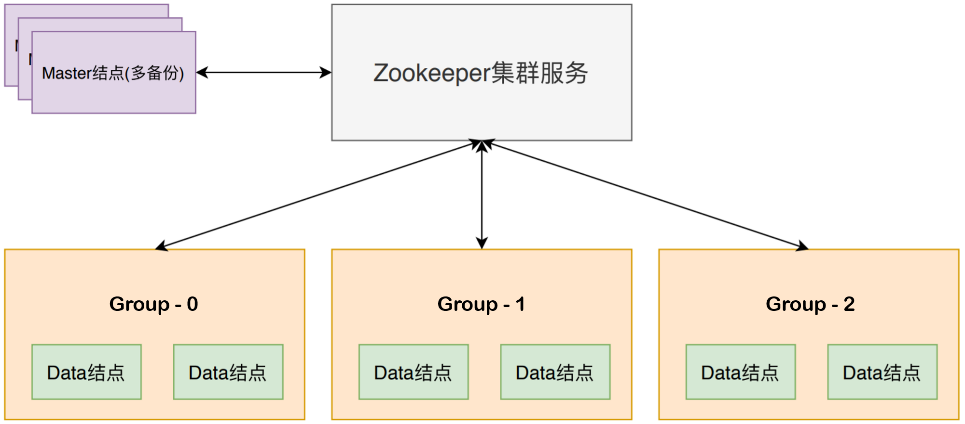
**图2.3 Zookeeper客户端建立连接后的示意图**

至此，我们成功搭建了一个三节点的Zookeeper服务集群，后续分布式键值对系统和Zookeeper集群服务的连接和使用通过第三方库kazoo进行建立。

# 3. 系统架构与设计

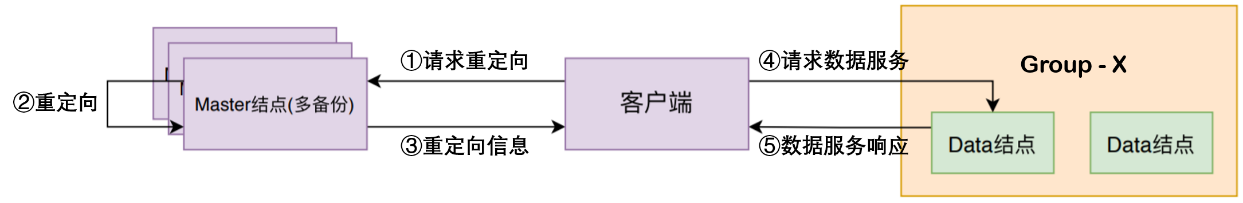
本系统采用C/S架构，用户通过不同的指令（get/put/delete）向服务端发出请求，向用户保证键值对存储的强一致性。

如图3.1所示，服务端则采用Master/Slave架构，结点可以分为Master结点和Data结点两种。Master结点是无状态的，本身不做任何存储相关的服务，而是以协调者的身份负责控制平面，比如对用户指令进行重定向、发现或处理Server结点的增减。Data结点是实际存储键值对的数据平面，负责数据的存储、迁移等；层次上，所有Data结点分属与不同的Group，同一个Group下的Data结点具备等价性，互为备份，从而提升可用性和性能。

****

**图3.1 服务端架构示意图**

如图3.2所示，对一条指令，客户端首先向Master结点发起重定向服务，Master根据该指令的key值等元数据将之重定向到某一个Group下的某一个Data结点，之后客户端向重定向的目标发起数据请求并被相应。客户端、Master结点和Data结点之间的所有通信均通过RPC完成，在这里我使用的是Python3自带的xmlrpc工具。

****

**图3.2 单条指令的基本业务流程**

实现基本的分布式键值对存储系统，只需要简单的转发请求即可。接下来我们将会考虑更多更复杂的需求，比如键值对存储系统的可扩展性、Master节点主备容错、Data节点的主备容错、如何用分布式锁处理读写并发、如何提供持久化操作等等。接下来，我将根据不同的功能需求进行分块讲解，描述我的设计以及实现过程中的问题与反思。

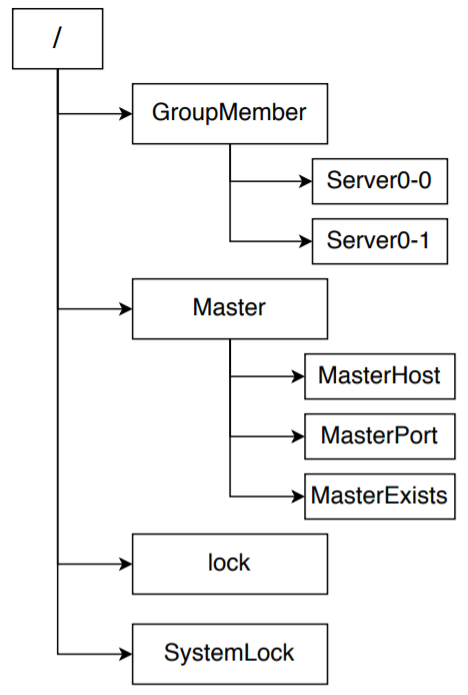
值得一提的是，接下来被讲解的功能顺序，就是我一步一步完善整个存储系统的过程，讲解顺序具备时间上的先后性，能够更好地体现我的思考过程和从小到大的迭代实现过程（可能后期的实现会被迫推翻前期的设计）。

但因为我加入了很多自己思考的过程，因此可能这部分的实验报告会有些冗长，如果老师和助教不感兴趣，可以直接移步本节的最后一小节，我在那里对需求功能的实现进行了一个设计小结。

## Zookeeper的使用

本系统实现中对Zookeeper的使用主要有以下几点：

1. 每一个Data节点在启动时会向/GroupMember中注册自己的临时节点，临时节点保存有该Data节点的所有有用信息，比如地址、端口、所属的Group等
2. 所有分布式读写锁统一注册到/lock下
3. 可以通过注册一个全局大锁/SystemLock来达到全局阻塞的目的
4. 将Master的信息保存在/Master中，比如Master的地址、主Master的临时节点等

****

**图3.1.1 ZooKeeper数据结构示意图**

## 基础功能的实现

如前文所说，分布式键值对存储系统的基础功能也就是PUT/GET/DELETE三个基本数据操作，我首先假设整个系统的状态为，单Master节点，每个Group下有两个Data节点，所有操作均由单客户端完成，不存在并发性，且不考虑操作过程中任何节点的崩溃。这样的实现是简单的，甚至连Zookeeper的集群服务都没有用上，此处不再赘述实现细节。

这里有两点值得一提：

第一，因为暂时没有涉及到可扩展性的问题，本着迭代开发的目的，这里Master节点对不同Key值的哈希重定向只是采用Python内置的简单哈希。在后面实现可扩展性需求时，我才引入了一致性哈希算法。

第二，为了维护同一个Group下多个Data节点的一致性。这里我采用的是二阶段提交（2PC）方法，通过阻塞执行、失败回滚的方式确保数据平面在用户视角下的强一致性。当然，还有一种更naïve的方法，即写操作到达的那个Data节点作为Primary，将这个写操作传播给同一Group的其他Peer节点，不用分两个阶段。

虽然我的实现上用了2PC，但是对比这两种方法，其实我是有一定的疑问的。

因为我们的分布式键值对存储系统是一个in-memory的系统，2PC的回滚日志其实没有被持久化，日志的地位和更新数据是一样的。所以2PC的方法看似有理，但其实和第二种naïve的方法的效果是一样的，他们都会受到网络阻隔和节点崩溃的影响。第二种naïve方法会遇上的Corner Case，在2PC的提交执行阶段都有可能碰上，那为什么我们还要采用实现复杂的2PC算法呢？说到这里，我其实也不确定这究竟是2PC本身的特性带来的（所以才有了后来3PC），还是因为我们实现的系统是in-memory导致的。若有机会，还请老师和助教能够不吝赐教。

## 锁与并发

在3.1的基础上，我开始考虑多客户端同时读写时如何处理并发数据请求（但仍不考虑任何节点崩溃的情况）。处理并发请求必然是要显式或隐式的利用锁的概念，根据加锁的粒度，一共有三种加锁的策略（这里只考虑对读写锁的应用）：

1. 加一个System-Level的全局大锁
2. 加一个Group-Level的组级锁
3. 加一个Key-Level的锁

显然，第一种加锁方式是不合理的，因为这样子所有的读写者都去尝试独占系统所有资源，明明两个客户端可以在不同Group节点进行操作，这时的存储系统并发性极差，因此不考虑第一种加锁方式。

和同学交流后发现，很多人采用的是第二种加锁策略，即一个Group下的Data节点可以接受多个读操作的并发或者一个写操作的独占，如此一来实现简单，甚至不用显式实现分布式锁就可以达到目的。

但我认为第二种加锁策略的锁粒度仍然太粗，当用户并发请求激增时，不同的用户的数据请求大概率会具备不同的Key值，不同Key值之间的任何读写操作都互不影响，但是却因为Group-Level的资源独占而被迫阻塞，这影响了系统响应和吞吐量。

因此，我采用的是第三种加锁策略。我显式实现了一个Key-Level的分布式读写锁锁，基本思想就是所有想要使用资源（发起数据请求）的客户端需要在询问Master之前在/lock/key下创建一个读锁或写锁的顺序临时节点，前列的人优先拿锁。而因为是临时节点，所以即使一个客户端拿锁后发生崩溃，锁资源也可以立即被释放，不会出现死锁。

通过上述设计，我们可以实现Key-Level的请求并行，相比于Group-Level的锁，吞吐量大大提升，当然因为我们对每一个Key的数据请求都会在Zookeeper中创建临时锁节点，因此可能在可扩展性上相对较差。

## 持久化存储

这里的持久化存储，我的设计是向客户端提供一个新的命令make\_persistence，用户向Master发送该指令，Master接收指令后，通知所有目前存活的Data节点进行写磁盘操作。但是这个持久化的操作如何与GET/PUT/DELETE进行并行呢？

首先，我认为持久化操作可以认为和GET一样是一个对内存的读操作，即写磁盘的过程不影响客户端进行GET操作，但是决不允许和PUT/DELETE操作同时进行。而实现这个阻塞目标，自然要引入锁的概念，那么对于持久化存储操作，什么粒度的锁不比较合适呢？

首先，我认为3.3中Key-Level的锁不合适，因为这需要对整个系统中所有存储的键值对都面向zookeeper建立至少一把锁，性能开销大，不具备可扩展性。至于System-Level和Group-Level的粒度，我认为Group-Level是更合适的，因为它不会导致全局阻塞，允许一定程度的并行。

但是，问题就来了，这里的Group-Level的加锁策略就和3.3中Key-Level的加锁策略冲突了！这意味着，如果我想用锁机制来解决持久化存储的并发问题，我还是必须将3.3中GET/PUT/DELETE的锁粒度提升到Group-Level！那么我在3.3中的设计就无效了，系统性能下降，并且代码回滚代价高昂。

作为妥协，我选择建立一把全局的读写锁（和3.3中Key-Level的读写锁相互独立），持久化操作以写者的身份去独占这把全局读写锁，而剩下的GET/PUT/DELETE则以读者身份共享全局锁。每一次向Master发送指令前，客户端都会有两次阻塞式的拿锁过程，先拿全局读写锁，再拿Key-Level对的读写锁，然后才能开始占用资源、请求服务。

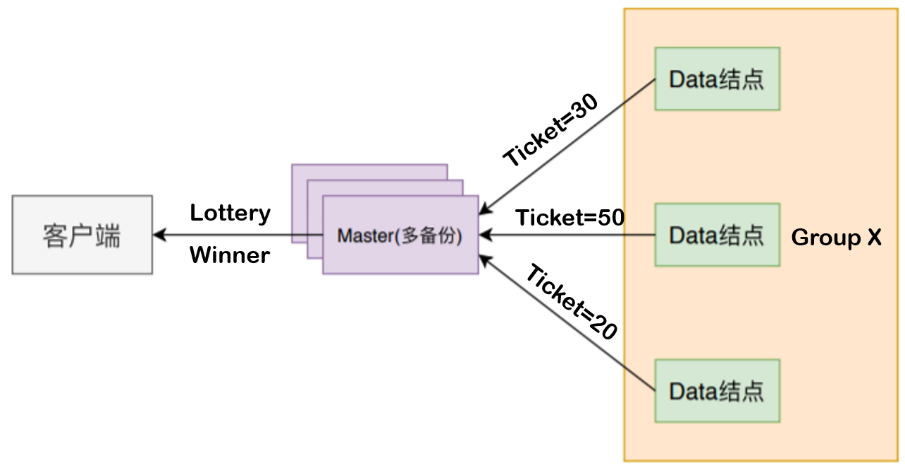
如此引入一把全局读写锁，可以说是我为了实现持久化存储的一种妥协，当然我们可以假设客户端的make\_persistence指令不会过于频繁。另一方面，在这里引入的全局读写锁又为我后面实现容错性和可扩展性提供了一定的便利，可以说是因祸得福吧~

## 负载均衡

因为我通过分布式读写锁和2PC来实现存储系统的强一致性，因此一个Group中的Data结点是等价的，没有谁是primary谁是standby的说法。不论一个读写操作被重定向到哪一个Data结点，最终都会通过阻塞传播的形式同步到其他同组Data结点中。因此我们可以在Master结点的重定向服务中加入负载均衡的设计，将读写操作均匀的分配的目标Group上的不同Data节点，以此来缓解单Primary Data结点的压力，提升系统的吞吐量。

那么如何合理的设计负载均衡算法呢？比较常见的思路就是，我们将请求压力均摊到每一个节点上。但是如果考虑到真实的多机分布式环境，不同的Data结点的部署环境和可使用的物理资源是不一样的，即不同Data结点的负载能力存在差异，简单的均摊对资源量低的Data结点不太友好。

因此，这里我采用了Lottery Algorithm（一种操作系统中的调度算法），由设计师、工程师根据部署环境，手动设置每一个Data结点的ticket权重，这个权重会在Data结点创建时被注册入Zookeeper中。这个权重代表了对应Data结点会被分配到的负载比例是多少，权重越大，则Data结点承担的负载越高。

****

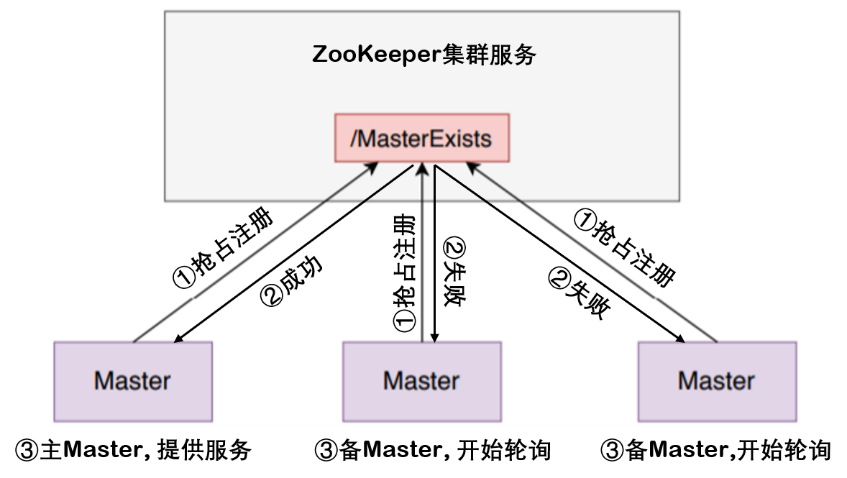
**图3.5.1 Lottery算法示意图**

在算法执行上，对每一个请求，Master将该请求重定向到对应Group，读取该Group下所有活跃的Data结点的权重，之后去一个随机数，这个随机数落在哪一个区间，则这个请求就由这个Data结点承担。显然，权重越高，则随机数落入其区间的概率越大，因此我们从期望上达成了按比例负载均衡的目的，甚至在运行过程中，我们仍然可以根据实际情况变化去调整不同Data结点的ticket权重。

## 容错性：Master

在这个分布式键值对存储系统中，因为所有的请求都需要经过Master进行重定向，Master处于一个单点故障的地位，因此我们需要为之添加备份Master，在主Master崩溃时能够迅速补位。

首先，所有的Data结点注册情况以及其他元数据信息都被存储在Zookeeper当中，在Master结点中仅仅是一份缓存拷贝，而Master结点本身是处于一个stateless的状态，只是实时监听Zookeeper集群中对应znode的变化并更新自己的信息缓存。而所有的重定向等服务，也都是Master基于这份实时更新的信息缓存做出的。

****

**图3.6.1 Master节点主备替换策略流程**

在保证了Master无状态的特点后，Master的主备替换设计相对简单。我们可以同时启动若干个Master节点，这些节点首先会抢占式的在Zookeeper集群中注册一个/MasterExists的临时节点，抢占注册成功的Master节点自动成为主Master，开始从Zookeeper中读取信息缓存，并对外提供服务；而没有抢占注册成功的Master则自动成为备份Master，处于不断轮询的状态，直到主Master崩溃，Zookeeper中的/MasterExists临时节点消失，则剩余的备份Master又同时进入抢占注册，最终决出一个主Master，剩余的继续轮询，等待下一次抢占竞争。

## 容错性：Data Node

我们在3.6节提到，Master的stateless性质为它的主备替换策略带来了便利，但是Data节点作为数据平面，是不可能做到无状态的，因此它的主备替换策略和Master有所不同。但值得注意，整个系统是向用户承诺强一致性的，同一个Group下所有的Data节点地位等价！这意味着，只要一个Group下仍有一个Data节点存活，那么这个Group就仍然可以对外提供等价的服务（只是吞吐量上会打折扣）。

由此，在我们之前的实现设计的铺垫下，Data节点的主备替换策略变得异常简单，总结就是“不作为”。当Master节点监听到/GroupMember下有节点崩溃后，只需判断每一个Group是否仍有至少一个Data节点存活，是则继续对外提供服务，略过本次节点变更；否则说明有一个Group完全崩溃，Group中的数据完全丢失（除非有持久化操作），系统无法正常对外提供服务，Master会主动挂起，不再对外提供重定向等服务，等待系统管理员来进行修复工作。

## 可扩展性：Server-Level

为了应对更高的负载和请求，或者是为了重启之前崩溃的服务器，我们可能需要在系统运行过程中，为某些Group动态新的Data节点。需要注意，这里的可扩展性是**Server层面的**，即加入新的Data节点后，**整个系统的Group数量不变**，因此每个Key值的重定向目标不变，**无需数据迁移操作**。

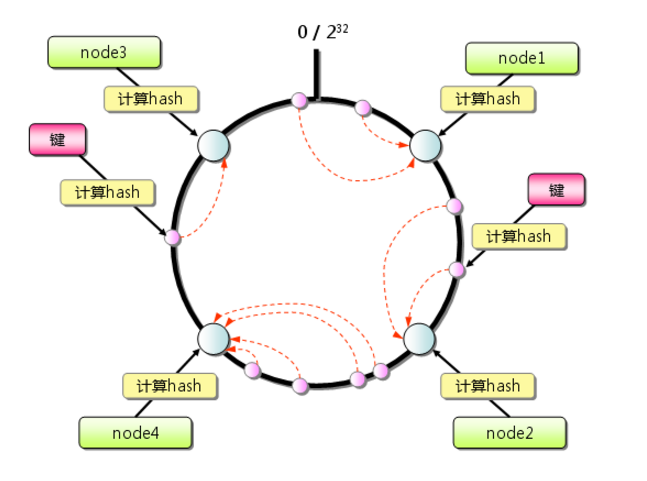
因为本系统向用户保证强一致性，一个Group中新扩展的Data节点处于无数据状态，需要来自同组其他Data节点的数据同步，而数据同步的这个过程通过拿3.4节中提到的System-Level Write Lock来达成阻塞，即同步完成之前，系统不再对外提供数据服务。

在我的设计中，可扩展的监听任务落在Master节点上，Master节点会实时监听/GroupMember下的节点变化。在监听到Server层面的有节点扩展后，Master首先去拿System-Level Write Lock（即在Zookeeper的相应路径注册临时锁节点），成功拿锁后，Master通过3.5中的Lottery算法向目标Group的一个旧Data节点发出sync\_send请求，让该节点将自己的所有键值对同步给新注册的Data节点，同步完成后，Master放锁，完成一轮Server层面的节点扩展。

## 可扩展性：Group-Level

区别于3.8中Server层面的扩展，Group层面的扩展即改变了整个系统的Group数量，这意味着所有Key值的哈希重定向目标都可能发生改变，需要数据迁移操作。

首先，为了尽可能减少需要迁移的数据，我引入了一致性哈希算法，每一个Group作为一致性哈希算法的一个真实节点，每个真实节点被映射为三个虚拟节点，在MD5的输出域上形成环形。



**图3.9.1 一致性哈希算法示意图**

通过引入一致性哈希算法，每次Group层面的节点扩展所需要迁移的数据量大大降低，但是数据迁移工作仍然不可避免。和3.8中的数据同步过程类似，数据迁移的监听任务由Master负责，并以全局阻塞的形式进行，以保证整个系统的强一致性。

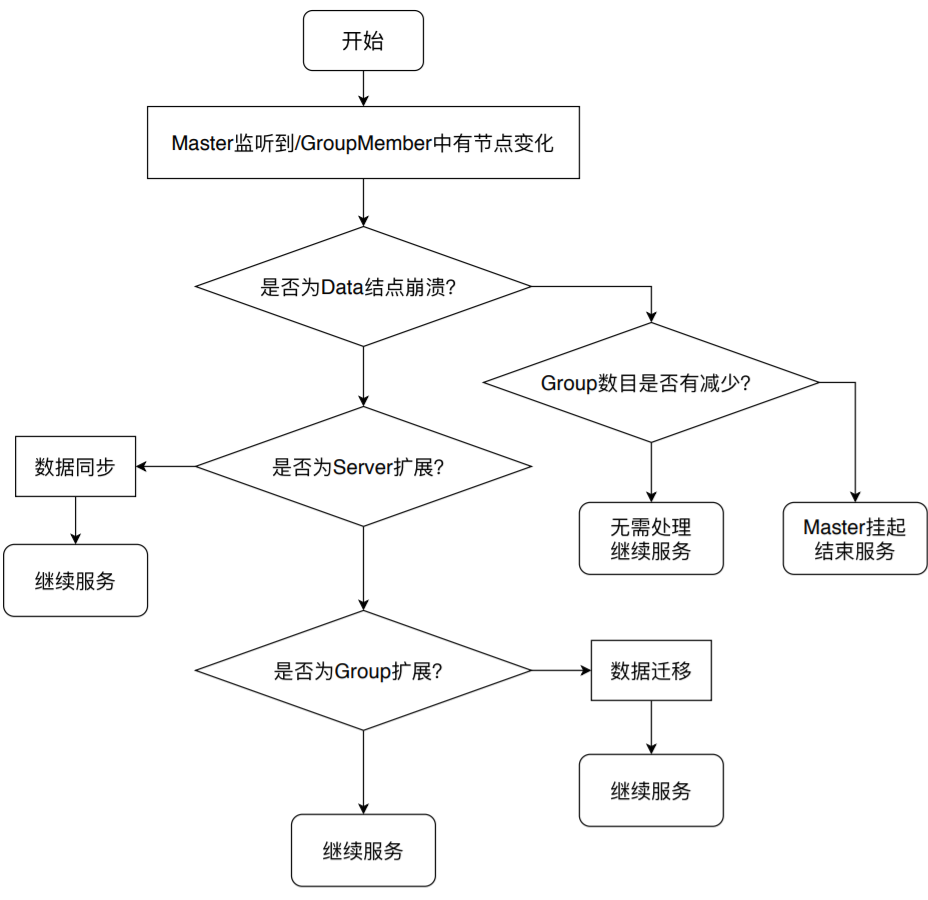
在监听到/GroupMember下有Group层面的节点扩展后， Master首先去拿3.4节中提到的得到System-Level Write Lock（即在Zookeeper的相应路径注册临时锁节点），成功拿锁后，Master遍历每一个Group，通过3.5中的Lottery算法向该Group的一个Data节点发出data\_transfer请求，让该节点将自己所有需要迁移的键值对传输个对应Group，而强一致性的保证下，这个迁移过程会被同步到同一Group下的其他Data节点上。迁移完成后，Master放锁，完成一轮Group层面的节点扩展。

## 小结

综上所述，我在实现基础的分布式键值对存储系统后，一共设计实现了如下扩展功能：

1. 通过Key-Level的分布式读写锁提升并发速度
2. 实现持久化存储（但是引入了全局读写锁，性能堪忧）
3. 通过Lottery算法实现了读写操作的负载均衡
4. 借助Master无状态的特点，实现了Master的主备替换
5. 实现了Data节点的主备替换
6. 实现Server层面的可扩展性
7. 实现Group层面的可扩展性

其中，5~7三个扩展功能本质上由需要Master节点对/GroupMember进行监听，因此在实现时被整合到一个模块之中。具体流程如下图所示：

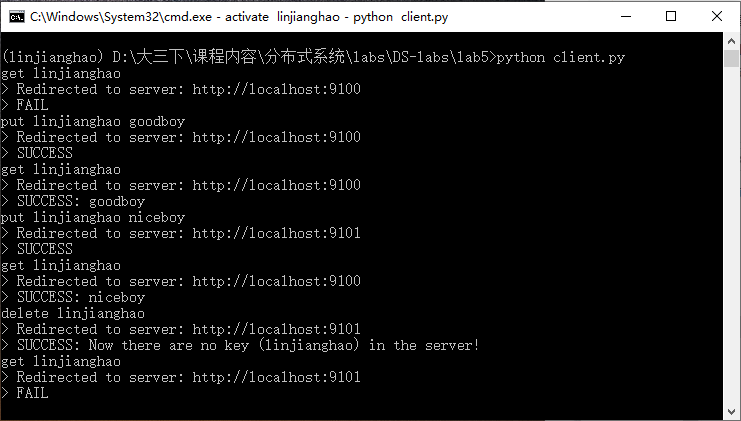
****

**图3.10.1 Master对/GroupMember监听的流程图**

# 4. 演示与测试

注意，这里提的是演示部分，即手工对client进行操作，方便向老师和助教展示实现效果，大规模自动化测试可见后文。针对我在第3节中实现的各类设计，我一共设计了7个演示用例，分别涵盖了功能性测试、并发性测试、Server节点的容错性测试、Master节点的容错性测试、Server-Level的可扩展性测试、Group-Level的可扩展性测试、持久化测试。具体内容可见同目录下的Demostration\_Doc.pdf。

个人演示测试结果显示，所有七个演示用例，分布式键值对系统全部通过测试。因为很多演示用例的成功判定是需要动态观察的，所以在报告中不方便呈现，可以在答辩的时候接受检阅。这里仅放出功能性测试的演示截图。

****

**图4.1 功能性演示结果截图**

# 5. 总结

本次课程实验可以说是非常有挑战性，并且给予了我们充分自由的发挥空间，我也能借此机会很好的将本学期所有的分布式系统的知识进行整理学习、融会贯通并最终加以实践。最后，再次感谢老师与助教这个学期的指导与帮助！