

上海交通大学

SHANGHAI JIAOTONG UNIVERSITY



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
2. Zookeeper 集群的搭建过程.....	1
3. 系统架构与设计.....	3
3.1 Zookeeper 的使用.....	4
3.2 基础功能的实现.....	4
3.3 锁与并发.....	5
3.4 持久化存储.....	6
3.5 负载均衡.....	6
3.6 容错性: Master.....	7
3.7 容错性: Data Node.....	8
3.8 可扩展性: Server-Level.....	8
3.9 可扩展性: Group-Level.....	9
3.10 小结.....	9
4. 演示与测试.....	10
5. 总结.....	11

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。

(一) 二进制文件下载

从 [Zookeeper 官网](#) 下载对应版本的工具包，并解压至相应文件夹。

(二) 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应的 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：

```

dataDir=D:\\apache-zookeeper-3.4.14\\build\\data1
dataLogDir=D:\\apache-zookeeper-3.4.14\\build\\log1
# the port at which the clients will connect
clientPort=2181
server.1=127.0.0.1:2777:3777
server.2=127.0.0.1:2888:3888
server.3=127.0.0.1:2999:3999

```

图 2.1 zoo1.cfg 的配置文件修改

(三) 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。

(四) 启动集群服务

因为我们需要在 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：

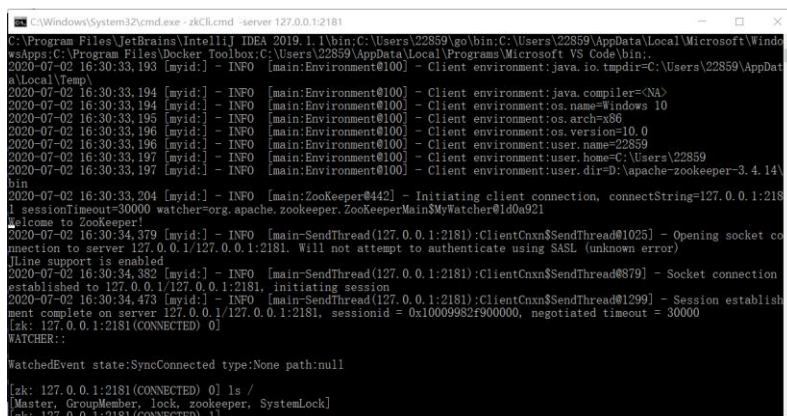
```

17 setlocal
18 call "%~dp0zkEnv.cmd"
19
20 set ZOO_MAIN=org.apache.zookeeper.server.quorum.QuorumPeerMain
21 set ZOO_CFG=..\conf\zoo1.cfg
22 echo on
23 call %JAVA% "-Dzookeeper.log.dir=%ZOO_LOG_DIR%" "-Dzookeeper.root
24
25 endlocal

```

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：



```

C:\Program Files\JetBrains\IntelliJ IDEA 2019.1.1\bin>zkCli.cmd -server 127.0.0.1:2181
2020-07-02 16:30:33.193 [myid:] - INFO [main:Environment@100] - Client environment:java.io.tmpdir=C:\Users\22859\AppData\Local\Temp\
2020-07-02 16:30:33.194 [myid:] - INFO [main:Environment@100] - Client environment:java.compiler=NA
2020-07-02 16:30:33.194 [myid:] - INFO [main:Environment@100] - Client environment:os.name=Windows 10
2020-07-02 16:30:33.195 [myid:] - INFO [main:Environment@100] - Client environment:os.arch=x86
2020-07-02 16:30:33.196 [myid:] - INFO [main:Environment@100] - Client environment:os.version=10.0
2020-07-02 16:30:33.196 [myid:] - INFO [main:Environment@100] - Client environment:user.name=22859
2020-07-02 16:30:33.197 [myid:] - INFO [main:Environment@100] - Client environment:user.home=C:\Users\22859
2020-07-02 16:30:33.197 [myid:] - INFO [main:Environment@100] - Client environment:user.dir=D:\apache-zookeeper-3.4.14\bin
2020-07-02 16:30:33.204 [myid:] - INFO [main:ZooKeeper@442] - Initiating client connection, connectString=127.0.0.1:2181
Welcome to ZooKeeper!
1 sessionTimeout=30000 watcher=org.apache.zookeeper.ZooKeeperMain$MyWatcher@1d0a921
2020-07-02 16:30:34.379 [myid:] - INFO [main-SendThread(127.0.0.1:2181):ClientCnxn$SendThread@1025] - Opening socket connection to server 127.0.0.1/127.0.0.1:2181. Will not attempt to authenticate using SASL (unknown error)
2020-07-02 16:30:34.382 [myid:] - INFO [main-SendThread(127.0.0.1:2181):ClientCnxn$SendThread@879] - Socket connection established to 127.0.0.1/127.0.0.1:2181, initiating session
2020-07-02 16:30:34.473 [myid:] - INFO [main-SendThread(127.0.0.1:2181):ClientCnxn$SendThread@1299] - Session establishment complete on server 127.0.0.1/127.0.0.1:2181, sessionId = 0x10009982f900000, negotiated timeout = 30000
[zk: 127.0.0.1:2181(CONNECTED) 0]
WATCHER::
WatchedEvent state:SyncConnected type:None path:null
[zk: 127.0.0.1:2181(CONNECTED) 0] ls /
[Master, GroupMember, lock, zookeeper, SystemLock]
[zk: 127.0.0.1:2181(CONNECTED) 1]

```

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。

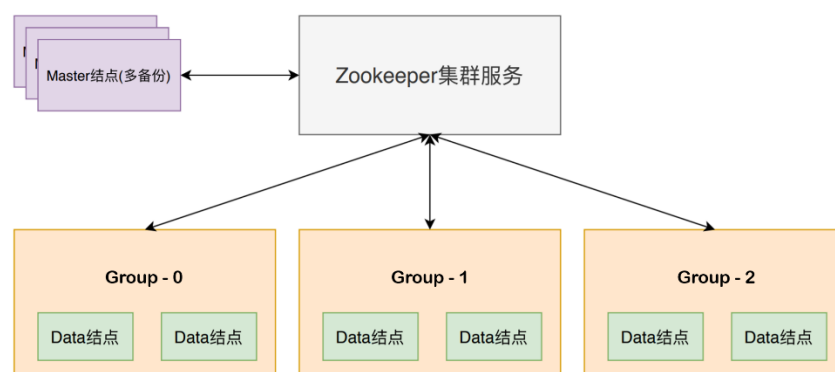


图 3.1 服务端架构示意图

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。

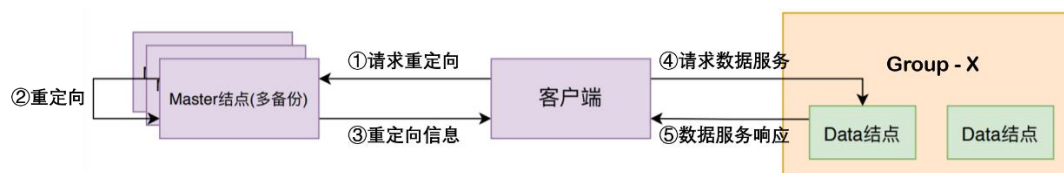


图 3.2 单条指令的基本业务流程

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。

3.1 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

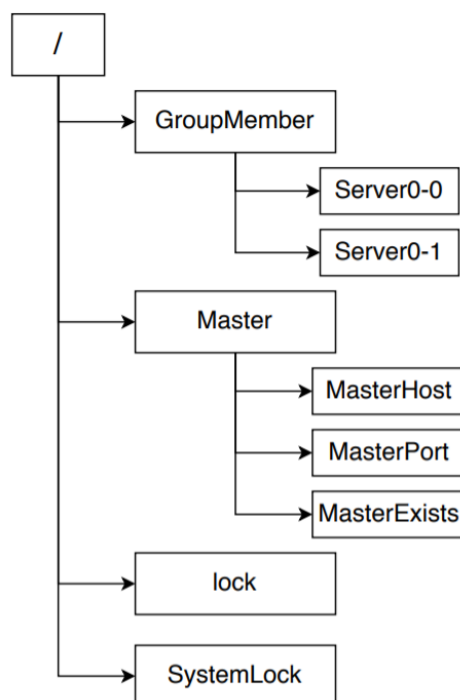


图 3.1.1 ZooKeeper 数据结构示意图

3.2 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。

3.3 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，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。

3.4 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~

3.5 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。

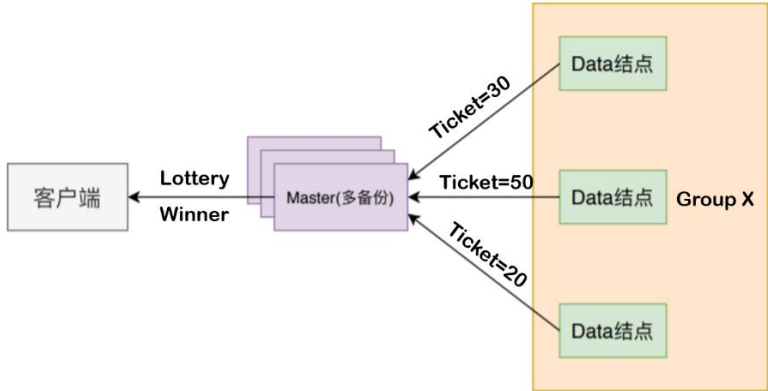


图 3.5.1 Lottery 算法示意图

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。

3.6 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。

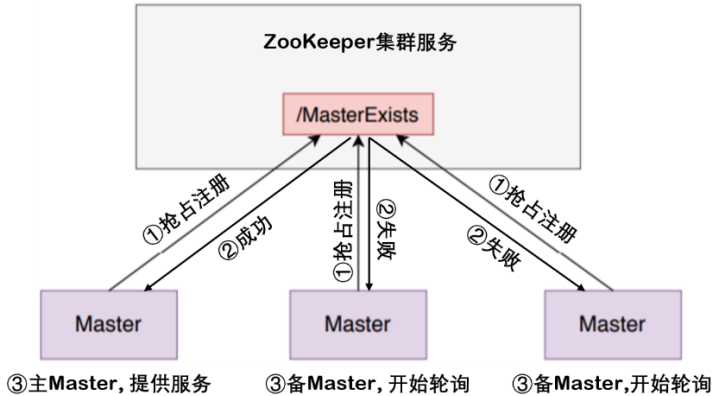


图 3.6.1 Master 节点主备替换策略流程

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。

3.7 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。

3.8 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。

3.9 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。

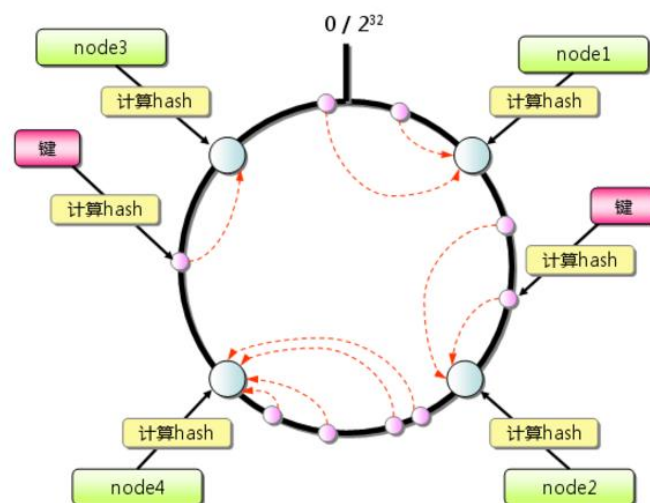


图 3.9.1 一致性哈希算法示意图

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。

3.10 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：

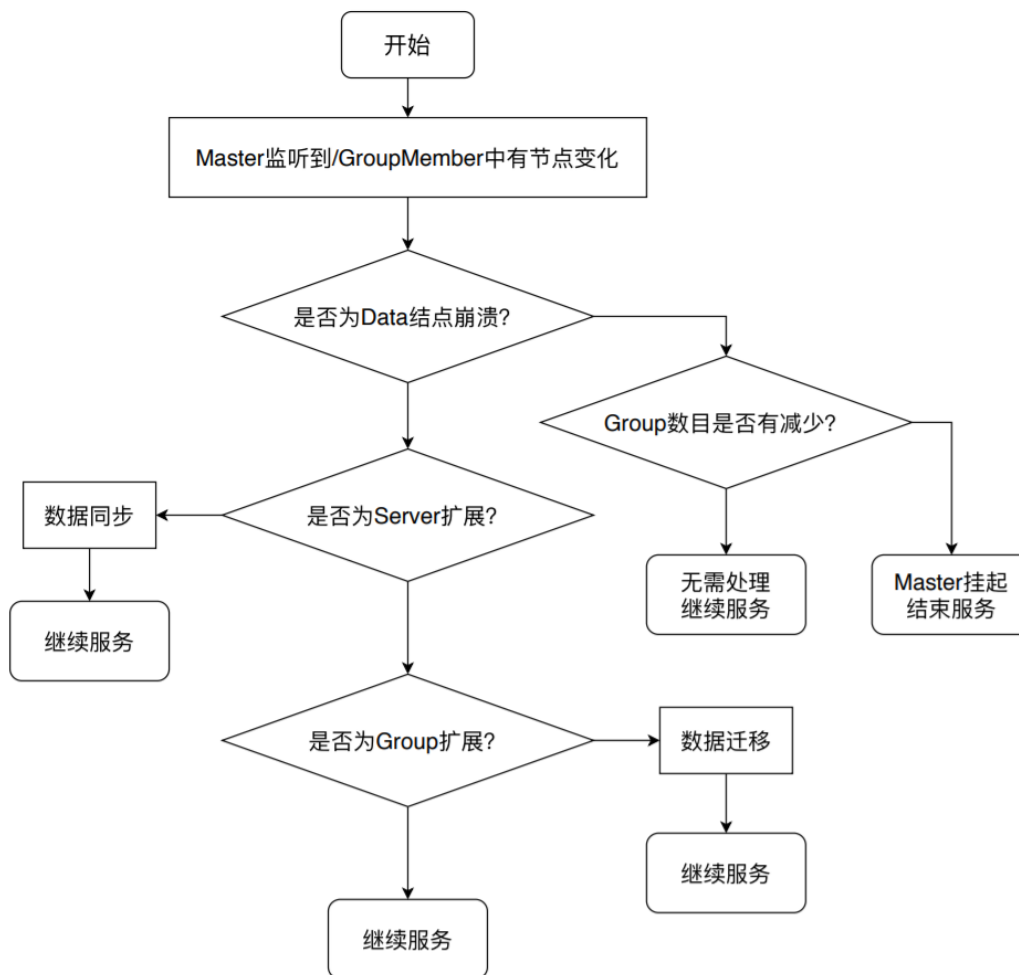


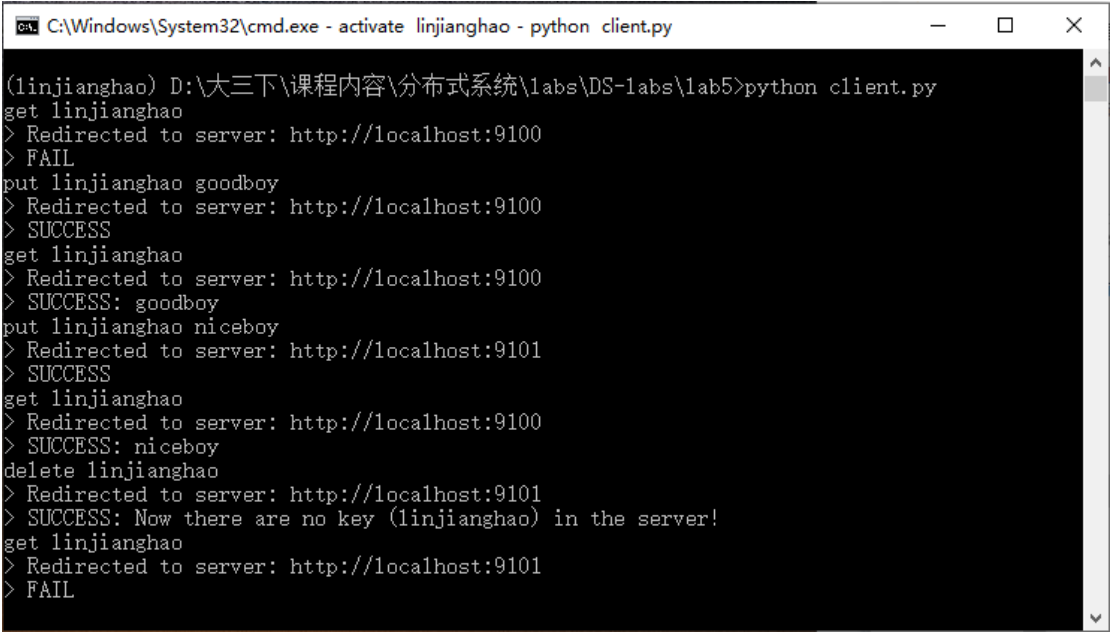
图 3.10.1 Master 对/GroupMember 监听的流程图

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 Demonstration_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。



```
(linjianghao) D:\大三下\课程内容\分布式系统\labs\DS-1abs\lab5>python client.py
get linjianghao
> Redirected to server: http://localhost:9100
> FAIL
put linjianghao goodboy
> Redirected to server: http://localhost:9100
> SUCCESS
get linjianghao
> Redirected to server: http://localhost:9100
> SUCCESS: goodboy
put linjianghao niceboy
> Redirected to server: http://localhost:9101
> SUCCESS
get linjianghao
> Redirected to server: http://localhost:9100
> SUCCESS: niceboy
delete linjianghao
> Redirected to server: http://localhost:9101
> SUCCESS: Now there are no key (linjianghao) in the server!
get linjianghao
> Redirected to server: http://localhost:9101
> FAIL
```

图 4.1 功能性演示结果截图

除了手工演示之外，我还在单机模式下进行了性能测试，并计算出相应的吞吐量。我设计的性能测试场景是，有 N 个客户端同时进行随机写操作，每个客户端会执行 10 条 PUT 指令，我分别取 N 为 1、10、50、75 与 100，测试结果如下表所示：

客户端并发量	1	10	50	75	100
总用时	89.11 s	90.21 s	93.91 s	96.14 s	115.51 s
吞吐量	0.11 req/s	1.11 req/s	5.32 req/s	7.80 req/s	8.65 req/s

通过上表我们可以清晰感受到分布式系统在面对并发请求时能够充分提高吞吐量。但是因为实验条件所限，我的部署和测试均只能在单机环境下进行，因此超过 100 的并发量会导致系统不稳、造成部分请求丢失的情况，测试结果不具备可靠性，因此没有列出。

可以合理推测，若将本系统部署至多机集群环境，能满足的并发量必然可以进一步上升，然后远程网络通信相比于本地 RPC 也会带来更高的时延，这都是我们从单机到多机所需要考虑的问题。

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！