Zookeeper-3.3.5 源码分析

INF-DT-BQE 刘少伟 （liushaowei@baidu.com）

目 录

0. 声明

本文档部分文字陈述及图片来自网络，已经尽量注明出处。

1. 安装部署

很多手册、博客中已经提到过，zookeeper 是一种为分布式应用提供一致性服务的协调系统，是开源 hadoop 项目下的一个子项目。分布式应用可以用它来实现诸如同步、配置管理、分组、命名服务等功能。系统用途、使用场景、典型应用在这里不再赘述，详情可参考：http://en.wikipedia.org/wiki/Zookeeper

zookeeper 最新 release 版本是 3.4.5，笔者主要针对 3.3.5 版本进行了深入了解：http://zookeeper.apache.org/doc/r3.3.5/，而本章主要讲解 Zookeeper-3.3.5 的安装部署。安装步骤可参考：http://hadoop.apache.org/zookeeper/docs/current/zookeeperStarted.html，但是原文讲述不够详细，比较笼统，因此有必要对步骤重新进行简略说明。

zookeeper 的安装模式有三种：

- 单机模式（stand-alone）：单机单 server
- 集群模式：多机多 server，形成集群
- 伪集群模式：单机多个 server，形成伪集群

1.1 单机模式

第一步，获取 zookeeper 稳定版本安装包，假设 MY_HOME 为基路径：

```
MY_HOME=******

cd ${MY_HOME}

wget http://mirror.bjtu.edu.cn/apache/zookeeper/stable/zookeeper-3.3.5.tar.gz

tar zxf zookeeper-3.3.5.tar.gz
```

第二步，修改配置文件 zoo.cfg：

```
cd ${MY_HOME}/zookeeper-3.3.5/conf

cp zoo_sample.cfg zoo.cfg

vi zoo.cfg
```

需要对 zoo.cfg 做如下修改：

```
tickTime=2000

dataDir=${MY_HOME}/zookeeper-3.3.5/data

clientPort=2222
```

第三步，启动 server：

```
cd ${MY_HOME}/zookeeper-3.3.5/bin

./zkServer.sh start
```

1.2 集群模式

第一步，获取 zookeeper 稳定版本安装包：

```
MY_HOME=******

cd ${MY_HOME}

wget http://mirror.bjtu.edu.cn/apache/zookeeper/stable/zookeeper-3.3.5.tar.gz

tar zxf zookeeper-3.3.5.tar.gz
```

第二步，配置文件 zoo.cfg 修改：

```
cd ${MY_HOME}/zookeeper-3.3.5/conf

cp zoo_sample.cfg zoo.cfg

vi zoo.cfg
```

需要对 zoo.cfg 做如下修改：

```
tickTime=2000

initLimit=10

syncLimit=5

dataDir=*****/zookeeper-3.3.5/data

clientPort=2222

server.1=server1_ip:2888:3888

server.2=server2_ip:2888:3888

server.3=server3_ip:2888:3888
```

注 1：本章会在后面补充讲述各配置参数的作用。

第三步，创建 data 目录和 myid 文件：

```
cd ${MY_HOME}

mkdir data

cd data

vi myid
```

在这里需要写下该 server 对应的编号，与 zoo.cfg 的 server list 对应，比如 server1 就写 1，server2 就写 2，server3 就写 3。

需要注意：zkServer.sh 脚本存在 BUG，可能导致出现异常。BUG 在 zkServer.sh 中：

```
start)
...
  nohup $JAVA "-Dzookeeper.log.dir=${ZOO_LOG_DIR}" \
  "-Dzookeeper.root.logger=${ZOO_LOG4J_PROP}" \
  -cp "$CLASSPATH" $JVMFLAGS $ZOOMAIN "$ZOOCFG" > "$_ZOO_DAEMON_OUT" 2>&1 < /dev/null &
  if [ $? -eq 0 ]
  then
    if /bin/echo -n $! > "$ZOOPIDFILE"
    then
      sleep 1
      echo STARTED
    else
      echo FAILED TO WRITE PID     #这句说明写入 Pid 出现了问题
      exit 1
    fi
...
```

dataDir 是 Zookeeper 进程内部建立的，并且有一定延迟，因此将 zookeeper 进程 id 写入到 dataDir 下的 pidfile 时，dataDir 可能还没有建立好。对启动脚本稍作修改即可修复这个问题：

```
start)
...
  nohup $JAVA "-Dzookeeper.log.dir=${ZOO_LOG_DIR}" \
  "-Dzookeeper.root.logger=${ZOO_LOG4J_PROP}" \
  -cp "$CLASSPATH" $JVMFLAGS $ZOOMAIN "$ZOOCFG" > "$_ZOO_DAEMON_OUT" 2>&1 < /dev/null &
  zkpid=$!;
  if [ $? -eq 0 ]
```

```
  then
    while [ ! -d `dirname $ZOOPIDFILE` ]
    do
      sleep 1;
    done
    if /bin/echo -n $zkpid > "$ZOOPIDFILE"
    then
      sleep 1
      echo STARTED
    else
      echo FAILED TO WRITE PID
      exit 1
    fi
...
```

第四步，同步该文件夹到其他机器上，然后依次启动各 server：

```
cd ${MY_HOME}/zookeeper-3.3.5/bin
./zkServer.sh start
```

1.3 伪集群模式

安装方式和集群模式类似，只是需要额外注意两点即可：

1) 部署到不同的 MY_HOME，比如将 server 部署到同一目录下的 server1、server2、server3 目录中。

2) zoo.cfg 中配置信息不能重复，尤其是 dataDir、clientPort 和 server list 的端口设置不能重复。建议配置方式如下：

```
tickTime=2000
initLimit=10
syncLimit=5
dataDir=*****/zookeeper-3.3.5/data
clientPort=*****
server.1=127.0.0.1:2888:2889
```

```
server.2=127.0.0.1:3888:3889

server.3=127.0.0.1:4888:4889
```

注 2：集群/伪集群模式启动首个 server 时，可能会报大量错误，这是因为现在只起了 1 个 server，server 启动后会根据 zoo.cfg 的 server list 发起选举 leader 的请求，因为连不上其他机器而报错；当我们起第二个 server 后，leader 将会被选出，从而一致性服务开始可以使用。

注 3：启动后需要对 server 的状态进行查看，有两种方法：

1) ./zkServer.sh status，可以获取"Using config"和"Mode"信息。

2) jps 命令，可以查看部署在 zookeeper 节点上的 QuorumPeerMain 进程是否存在。

1.4 配置参数说明

dataDir

用于存放内存数据库快照的文件夹，同时用于集群的 myid 文件也存在这个文件夹里。

dataLogDir

用于单独设置 transaction log 的目录，transaction log 分离可以避免和普通 log 还有快照的竞争。

tickTime

心跳时间，为了确保 client-server 连接存在的，以毫秒为单位，最小超时时间为两个心跳时间。

clientPort

客户端监听端口。

globalOutstandingLimit

client 请求队列的最大长度，防止内存溢出，默认值为 1000。

preAllocSize

预分配的 Transaction log 空间 block 为 proAllocSize KB，默认 block 为 64M，一般不需要更改，除非 snapshot 过于频繁。

snapCount

在 snapCount 个 snapshot 后写一次 transaction log，默认值是 100,000。

traceFile

用于记录请求的 log，打开会影响性能，用于 debug，最好不要定义。

maxClientCnxns

最大并发客户端数，用于防止 DOS 的，默认值是 10，设置为 0 是不加限制。

clientPortBindAddress

可以设置指定的 client ip 以及端口，不设置的话等于 ANY:clientPort

minSessionTimeout

最小的客户端 session 超时时间，默认值为 2 个 tickTime，单位是毫秒

maxSessionTimeout

最大的客户端 session 超时时间，默认值为 20 个 tickTime，单位是毫秒

electionAlg

用于选举的实现的参数，0 为以原始的基于 UDP 的方式协作，1 为不进行用户验证的基于 UDP 的快速选举，2 为进行用户验证的基于 UDP 的快速选举，3 为基于 TCP 的快速选举，默认值为 3。

initLimit

多少个 tickTime 内，允许其他 server 连接并初始化数据，如果 zooKeeper 管理的数据较大，则应相应增大这个值。

syncLimit

多少个 tickTime 内，允许 follower 同步，如果 follower 落后太多，则会被丢弃。

leaderServes

leader 是否接受客户端连接。默认值为 yes。leader 负责协调更新。当更新吞吐量远高于读取吞吐量时，可以设置为不接受客户端连接，以便 leader 可以专注于同步协调工作。

server.x=[hostname]:nnnnn[:nnnnn]

配置集群里面的主机信息，其中 server.x 的 x 要写在 myid 文件中，决定当前机器的 id，第一个 port 用于连接 leader，第二个用于 leader 选举。如果 electionAlg 为 0，则不需要第二个 port。hostname 也可以填 ip。

group.x=nnnnn[:nnnnn]

分组信息，表明哪个组有哪些节点，例如 group.1=1:2:3 group.2=4:5:6 group.3=7:8:9。

weight.x=nnnnn

权重信息，表明哪个结点的权重是多少，例如 weight.1=1 weight.2=1 weight.3=1。

2. 架构和角色

2.1 系统架构

zookeeper 分为服务器端（server）和客户端（client），客户端可以连接到整个 zooKeeper 服务的任意服务器上（除非 leaderServes 参数被显式设置，leader 不允许接受客户端连接）。客户端使用并维护一个 TCP 连接，通过这个连接发送请求、接受响应、获取观察的事件以及发送心跳。如果这个 TCP 连接中断，客户端将自动尝试连接到另外的 zooKeeper 服务器。客户端第一次连接到 zooKeeper 服务时，接受这个连接的 zooKeeper 服务器会为这个客户端建立一个会话。当这个客户端连接到另外的服务器时，这个会话会被新的服务器重新建立。

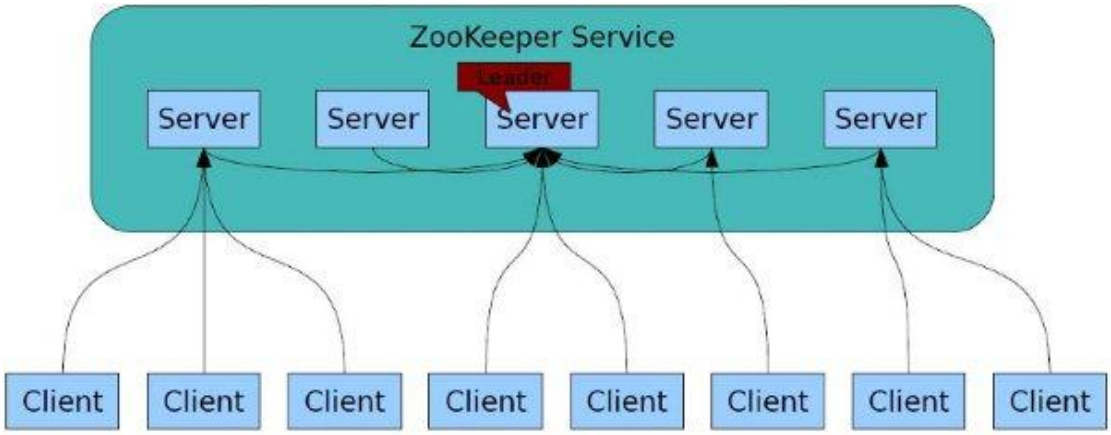

图 zookeeper 架构图（摘抄自网络）

在 org.apache.zookeeper.Watcher 中定义了 Event.KeeperState 类型，用于表示 client 和 server 之间的状态，包括：

```
/**
 * Unused, this state is never generated by the server
 */
Unknown (-1),
/**
 * The client is in the disconnected state - it is not connected
 * to any server in the ensemble.
 */
Disconnected (0),
/**
```

```
     * Unused, this state is never generated by the server
     */
    NoSyncConnected (1),
    /**
     * The client is in the connected state - it is connected
     * to a server in the ensemble (one of the servers specified
     * in the host connection parameter during ZooKeeper client
     * creation).
     */
    SyncConnected (3),
    /**
     * Auth failed state
     */
    AuthFailed(4),
    /**
     * The serving cluster has expired this session. The ZooKeeper
     * client connection (the session) is no longer valid. You must
     * create a new client connection (instantiate a new ZooKeeper
     * instance) if you with to access the ensemble.
     */
    Expired (-112);
```

需要注意 session 机制。client 连接 server 成功后，server 赋予该 client 一个 sessionid，client 需要不断发送心跳维持 session 有效，在 session 有效期内，可以使用 Zookeeper 提供的 API 进行操作。如果因为某些原因导致 client 无法正常发送心跳，在超时时长后，server 会判断该 client 的 session 失效，此时 client 发送的任何操作都会被拒绝，并触发 ExpiredException 异常，此时 KeeperState 处于 Expired 状态。

但 client 有自动重连 server 的机制，如果 client 的心跳无法正常连接 server，会在 session 超时前尝试连接其他 server，连接成功后可以继续操作。

如果 client 取消当前连接并连接其他 server，已存在的 watches 会丢失，取而代之的是 client 会生成一个特殊 WatchEvent 告诉本地 watcher 连接已经丢失，该 WatchEvent 是：EventType.None，KeeperState.DisConnected。本地 watcher 接收到该 WatchEvent 后会怎样？

2.2 server 角色

启动 zookeeper 服务器集群环境后，多个 Zookeeper 服务器在工作前会选举出一个 Leader。选举出 leader 前，所有 server 不区分角色，都需要平等参与投票（observer 除外，不参与投票）；选主过程完成后，存在以下几种角色：

- leader：领导者，可以接受 client 请求，也接收其他 server 转发的写请求，负责更新系统状态。
- follower：可以接收 client 请求，如果是写请求将转发给 leader 来更新系统状态。
- observer：同 follower，唯一区别就是不参与选主过程。

org.apache.zookeeper.server.quorum.QuorumPeer 定义了 server 的类型，其中 ServerState 表示 server 类型，LeanerType 表示当 ServerState 为 FOLLOWING 时是参与者还是观察者，前者称为 follower，后者称为 observer。

```
public enum ServerState {
        LOOKING, FOLLOWING, LEADING, OBSERVING;
}


/*
  * A peer can either be participating, which implies that it is willing to
  * both vote in instances of consensus and to elect or become a Leader, or
  * it may be observing in which case it isn't.
  *
  * We need this distinction to decide which ServerState to move to when
  * conditions change (e.g. which state to become after LOOKING).
  */
public enum LearnerType {
        PARTICIPANT, OBSERVER;
}
```

3. 数据模型和存储

3.1 数据模型

zookeeper 提供一种类似目录树结构的数据模型，每个节点（znode）具有唯一的路径标识，而路径是由斜线分隔开的路径名序列组成，和标准的文件系统非常类似，例如：/zookeeper/app/ugi/wangsan。

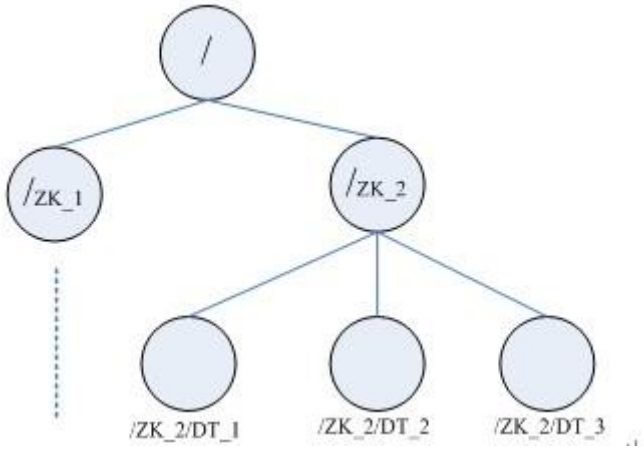

图　zk 目录树结构（摘抄自网络）

3.2 znode 节点

znode 兼具文件和目录两种特点，既像文件一样维护着数据、元信息、ACL、时间戳等数据结构，又像目录一样可以作为路径标识的一部分，并可以具有子 znode。用户对 znode 具有增、删、改、查等操作（权限允许的情况下）。

znode 具有原子性操作，每个 znode 的数据将被原子性地读写，读操作会读取与 znode 相关的所有数据，写操作会一次性替换所有数据。

zookeeper 并没有被设计为常规的数据库或者大数据存储，相反的是，它用来管理调度数据，比如分布式应用中的配置文件信息、状态信息、汇集位置等等。这些数据的共同特性就是它们都是很小的数据，通常以 KB 为大小单位。zooKeeper 的服务器和客户端都被设计为严格检查并限制每个 znode 的数据大小至多 1M，当时常规使用中应该远小于此值。

需要额外注意以下几点：

1) znode 中的数据可以有多个版本，在查询该 znode 数据时就需要带上版本信息。(set path version / delete path version）

2) znode 可以是临时 znode（create -e 生成的节点），一旦创建这个 znode 的 client 与 server 断开连接，该 znode 将被自动删除。client 和 server 之间通过 heartbeat 来确认连接正常，这种状态称之为 session，断开连接后 session 失效。

3) 临时 znode 不能有子 znode。

4) znode 可以自动编号（create -s 生成的节点），例如在 create -s /app/node 已存在时，将会生成 /app/node00***001 节点。

5) znode 可以被监控，该目录下某些信息的修改，例如节点数据、子节点变化等，可以主动通知监控注册的 client。事实上，通过这个特性，可以完成许多重要应用，例如配置管理、信息同步、分布式锁等等。

org.apache.zookeeper.CreateMode 中定义了四种节点类型，分别对应：

- PERSISTENT：永久节点
- EPHEMERAL：临时节点
- PERSISTENT_SEQUENTIAL：永久节点、序列化
- EPHEMERAL_SEQUENTIAL：临时节点、序列化

3.3 Stat 类

org.apache.zookeeper.data.Stat** 类定义 znode 节点的元信息，主要成员变量如下：

```
private long czxid;        // 创建时的 zxid
private long mzxid;        // 最新修改的 zxid
private long ctime;        // 创建时间
private long mtime;        // 修改时间
private int version;       // 版本号，对应 znode data
private int cversion;      // 版本号，对应子 znode
private int aversion;          // 版本号，对应 acl
private long ephemeralOwner;     // 临时节点对应的 client session id，默认为 0
private int dataLength;        // znode data 长度
private int numChildren;       // 子 znode 个数
private long pzxid;            // 最新修改的 zxid，貌似与 mzxid 重合了
```

注意 Stat、StatPersisited、StatPersisitedV1 三个类，其成员变量和逻辑基本一致，但 StatPersisited 类少了 dataLength 和 numChildren 属性，StatPersisitedV1 类少了 dataLength、numChildren 和 pzxid 属性，具体不同类用在什么地方待进一步分析。

3.4 DataNode 类

org.apache.zookeeper.server.DataNode 类记录了 znode 节点的所有信息，包括其父节点

（单个）、子节点（多个）、数据内容、ACL 信息、stat 元数据等，主要成员变量如下：

```
/** the parent of this datanode */
DataNode parent;
/** the data for this datanode */
byte data[];
/**
 * the acl map long for this datanode. the datatree has the map
 */
Long acl;
/**
 * the stat for this node that is persisted to disk.
 */
public StatPersisted stat;
/**
 * the list of children for this node. note that the list of children string
 * does not contain the parent path -- just the last part of the path. This
 * should be synchronized on except deserializing (for speed up issues).
 */
private Set<String> children = null;
```

需要注意的是 acl 和 children 两个成员变量。acl 是 Long 型值，相当于 acl key，具体的 ACL 信息实际上保存在 DataTree 中的 longKeyMap 和 aclKeyMap 中，前者保存了整个目录树所有节点的 ACL 信息，类型是 Map<Long, List<ACL>>，可以根据 acl key 获得某节点的 ACL 信息列表，后者则是该 map 的反向结构。

children 用于记录该节点的子节点列表信息，但保存的并不是 DataNode 类型，而是只保存了每个子节点路径名的最后部分，比如该节点为”/biglog/liushaowei”，子节点为”/biglog/liushaowei/test1”，那么 children 中保存”test1”这个相对路径，这么做的目的是：This should be synchronized on except deserializing (for speed up issues)。

3.5 DataTree 类

org.apache.zookeeper.serve.DataTree 类维护整个目录树结构，ConcurrentHashMap<String, DataNode> nodes 保存了从完整路径到 DataNode 的 hashtable，而 DataNode 中的 Set<String> children 保存了父子关系，即子节点的相对路径。通过某 DataNode 可以获知其任意子节点

的相对路径，然后拼装成完整路径，再去 DataTree 的 nodes 中查找。

所有对节点路径的访问都是通过 nodes 完成的。

主要成员变量如下：

```
/**
 * This hashtable provides a fast lookup to the datanodes. The tree is the
 * source of truth and is where all the locking occurs
 */
private final ConcurrentHashMap<String, DataNode> nodes =
new ConcurrentHashMap<String, DataNode>();
/**
 *  基于 znode data 的注册 watches 管理
 */
private final WatchManager dataWatches = new WatchManager();
/**
 *  基于 znode children 的注册 watches 管理
 */
private final WatchManager childWatches = new WatchManager();
/**
 * the path trie that keeps track fo the quota nodes in this datatree
 */
private final PathTrie pTrie = new PathTrie();
/**
 * This hashtable lists the paths of the ephemeral nodes of a session.
 * Long is sessionid
 */
private final Map<Long, HashSet<String>> ephemerals = new ConcurrentHashMap<Long,
HashSet<String>>();
/**
 * this is map from longs to acl's. It saves acl's being stored for each
 * datanode.
```

```
 */
public final Map<Long, List<ACL>> longKeyMap = new HashMap<Long, List<ACL>>();
/**
 * this a map from acls to long.
 */
public final Map<List<ACL>, Long> aclKeyMap = new HashMap<List<ACL>, Long>();
/**
 * these are the number of acls that we have in the datatree
 */
protected long aclIndex = 0;
```

1. DataTree 初始化

DataTree 初始化要完成的工作，需要建立系统用节点，包括/、/zookeeper、/zookeeper/quota 三个 znode。

```
public DataTree() {
    /* Rather than fight it, let root have an alias */
    nodes.put("", root);
    nodes.put(rootZookeeper, root);
    /** add the proc node and quota node */
    root.addChild(procChildZookeeper);
    nodes.put(procZookeeper, procDataNode);
    procDataNode.addChild(quotaChildZookeeper);
    nodes.put(quotaZookeeper, quotaDataNode);
}
```

2. quota 节点

对于 zookeeper 上的任意节点，其下都默认存在两个子节点：***/zookeeper_limits 和 ***/zookeeper_stats，其中 zookeeper_limits 定义了 znode 的 subtree 数量限制和数据量限制，而 zookeeper_stats 则保存了该 znode 的 substree 数量和数据量。在进行子节点更新时，需要调用 DataTree 的 updateCount 和 updateBytes 完成更新和限制判断（但 updateCount 和 updateBytes 中并没有限死，只是打了 warnning 日志）。

相关判断逻辑可以查阅 org.apache.zookeeper.Quota 类和 org.apache.zookeeper.StatsTrack 类。

3. createNode 过程

```
/**
 * @param path
 * @param data
 * @param acl
 * @param ephemeralOwner
 *                the session id that owns this node. -1 indicates this is not
 *                an ephemeral node.
 * @param zxid
 * @param time
 * @return the patch of the created node
 * @throws KeeperException
 */
public String createNode(String path, byte data[], List<ACL> acl,
        long ephemeralOwner, long zxid, long time)
        throws KeeperException.NoNodeException,
        KeeperException.NodeExistsException
```

具体的 createNode 过程如下：

1) 创建 StatPersisted stat 元数据，并 set 各种成员变量；
2) 创建 DataNode child 节点；
3) 解析父节点路径 parentName，并通过 DataNode parent = nodes.get(parentName)获取父节点，然后更新 parent 的 pzxid、cversion、ephemeralOwner；
4) 将 child 放入 parent 的 children 列表中，以及放入 DataTree 的 nodes 中：parent.addChild(childName); nodes.put(path, child);
5) 如果是临时节点，需要保存到 DataTree 的 ephemerals 中，key 是所属 owner 的 sessionid；
6) 判断该节点是否/zookeeper/quota/zookeeper_limits 或/zookeeper/quota/zookeeper_stat，如果是则？？？？；
7) 更新该节点的 quota 信息，即***/ zookeeper_stat 节点内容；
8) 调用 dataWatches.triggerWatch()触发该路径的 Event.EventType.NodeCreated 相关事件；

9) 调用 childWatches.triggerWatch() 触发父节点路径的 Event.EventType.NodeChildrenChanged 相关事件。

4. deleteNode 过程

```
/**
 * remove the path from the datatree
 *
 * @param path
 *                the path to of the node to be deleted
 * @param zxid
 *                the current zxid
 * @throws KeeperException.NoNodeException
 */
public void deleteNode(String path, long zxid)
        throws KeeperException.NoNodeException
```

具体的 deleteNode 过程如下：

1) 根据 DataNode node = nodes.get(path)获取该节点的 DataNode；

2) 根据 DataNode parent = nodes.get(parentName)获取该节点的父节点；

3) 更新 parent 的 children 列表、cversion、pzxid、ephemeralOwner，如果是临时节点，还要更新 DataTree 的 ephemerals；

4) 判断该节点是否/zookeeper/quota/zookeeper_limits 或/zookeeper/quota/zookeeper_stat，如果是则？？？？；

5) 更新该节点的 quota 信息，即***/ zookeeper_stat 节点内容；

6) 调用 dataWatches.triggerWatch()触发该路径的 Event.EventType.NodeDeleted 相关事件；

7) 调用 childWatches.triggerWatch() 触发父节点路径的 Event.EventType.NodeChildrenChanged 相关事件。

5. setData 过程

```
public Stat setData(String path, byte data[], int version, long zxid,
                long time) throws KeeperException.NoNodeException
```

具体的 setData 过程：

1) 根据 DataNode n = nodes.get(path)获取该节点 DataNode；

2) 更新 n 的 data、mtime、mzxid、version 信息；

3) 调用 DataTree 的 updateBytes 更新 Quota 信息；

4) 调用 dataWatches.triggerWatch() 触发该节点路径的 Event.EventType.NodeDataChanged 相关事件。

6. getData 过程

```
public byte[] getData(String path, Stat stat, Watcher watcher)
                    throws KeeperException.NoNodeException
```

具体的 getData 过程如下：

1) 根据 DataNode n = nodes.get(path)获取该节点 DataNode；

2) 如果 watcher 参数不为 NULL，调用 dataWatches.addWatch()添加 watcher；

3) 返回 n 的 data 信息。

7. statNode 过程

```
public Stat statNode(String path, Watcher watcher)
                    throws KeeperException.NoNodeException
```

8. getChildren 过程

```
public List<String> getChildren(String path, Stat stat, Watcher watcher)
                    throws KeeperException.NoNodeException
```

9. getCounts 过程

```
/**
 * this method gets the count of nodes and the bytes under a subtree
 *
 * @param path
 *                the path to be used
 * @param counts
 *                the int count
 */
private void getCounts(String path, Counts counts)
```

4. CLI 使用

使用 ./zkCli.sh -server host:port 登录 cli 后，执行 help 命令可获取命令列表，如下：

```
ZooKeeper -server host:port cmd args
connect host:port
get path [watch]
ls path [watch]
set path data [version]
delquota [-n|-b] path
quit
printwatches on|off
create [-s] [-e] path data acl
stat path [watch]
close
ls2 path [watch]
history
listquota path
setAcl path acl
getAcl path
sync path
redo cmdno
addauth scheme auth
delete path [version]
setquota -n|-b val path
```

下面具体讲解常见命令的一般用法。

1. create 操作

用法：create /app/node1 data

描述：创建 node1，数据内容为 data，前提是前置路径必须存在。注意，没有递归创建节点的原语。

用法：create -s /app/node1 data

描述：创建自增的 node1，数据内容为 data，如果 node1 已经存在，会自动创建 node00***001 节点。

用法：create -e /app/node1

描述：创建临时的 node1，数据内容为 data，client 和 server 断开连接后，该结点自动删除。

2. get 操作

用法：get /app/node1

描述：获取 node1 节点保存的数据，以及元信息。

3. set 操作

用法：set /app/node1 data

描述：设置 node1 节点保存的数据为 data，并显示元信息。

4. delete 操作

用法：delete /app/node1

描述：删除 node1 节点，前提是 node1 没有子节点。注意，没有递归删除节点的原语。

5. ls/ls2/stat 操作

用法：ls /app/node1

描述：展示 node1 节点下的子节点列表。

用法：ls2 /app/node1

描述：展示 node1 节点下的子节点列表，以及 node1 节点的元信息。

用法：stat /app/node1

描述：展示 node1 节点的元信息。

6. history/redo 操作

用法：history

描述：展示最近的历史命令。

用法：redo 10

描述：重新执行编号为 10 的命令。

7. close/quit 操作

用法：close

描述：关闭 session 和连接，但不退出 cli。

用法：quit

描述：关闭连接并退出 cli。

8. listquota/setquota/delquota 操作

略。

9. addauth/setAcl/getAcl 操作

略。

5. CAP 理论及一致性原则

分布式领域中存在 CAP 理论，且该理论已被证明：任何分布式系统只可同时满足两点，无法三者兼顾。

- C：Consistency，一致性，数据一致更新，所有数据变动都是同步的。
- A：Availability，可用性，系统具有好的响应性能。
- P：Partition tolerance，分区容错性。

因此，将精力浪费在思考如何设计能满足三者的完美系统上是愚钝的，应该根据应用场景进行适当取舍。

一致性是指从系统外部读取系统内部的数据时，在一定约束条件下相同，即数据变动在系统内部各节点应该是同步的。根据一致性的强弱程度不同，可以将一致性级别分为如下几种：

- 强一致性（strong consistency）。任何时刻，任何用户都能读取到最近一次成功更新的数据。示例 case：zookeeper。
- 单调一致性（monotonic consistency）。任何时刻，任何用户一旦读到某个数据在某次更新后的值，那么就不会再读到比这个值更旧的值。也就是说，可获取的数据顺序必是单调递增的。示例 case：暂无。
- 会话一致性（session consistency）。任何用户在某次会话中，一旦读到某个数据在某次更新后的值，那么在本次会话中就不会再读到比这个值更旧的值。会话一致性是在单调一致性的基础上进一步放松约束，只保证单个用户单个会话内的单调性，在不同用户或同一用户不同会话间则没有保障。示例 case：php 的 session 概念。
- 最终一致性（eventual consistency）。用户只能读到某次更新后的值，但系统保证数据将最终达到完全一致的状态，只是所需时间不能保障。示例 case：暂无。
- 弱一致性（weak consistency）。用户无法在确定时间内读到最新更新的值。

那么，zookeeper 到底提供的是什么样的一致性服务呢？

很多文章和博客里提到，zookeeper 是一种提供强一致性的服务，在分区容错性和可用性上做了一定折中，这和 CAP 理论是吻合的。但实际上 zookeeper 提供的只是单调一致性。

原因：

1. 假设有 2n+1 个 server，在同步流程中，leader 向 follower 同步数据，当同步完成的 follower 数量大于 n+1 时同步流程结束，系统可接受 client 的连接请求。如果 client 连接的并非同步完成的 follower，那么得到的并非最新数据，但可以保证单调性。

2. follower 接收写请求后，转发给 leader 处理；leader 完成两阶段提交的机制。向所有 server 发起提案，当提案获得超过半数（n+1）的 server 认同后，将对整个集群进行同步，超过半数（n+1）的 server 同步完成后，该写请求完成。如果 client 连接的并非同步完成的 follower，那么得到的并非最新数据，但可以保证单调性。

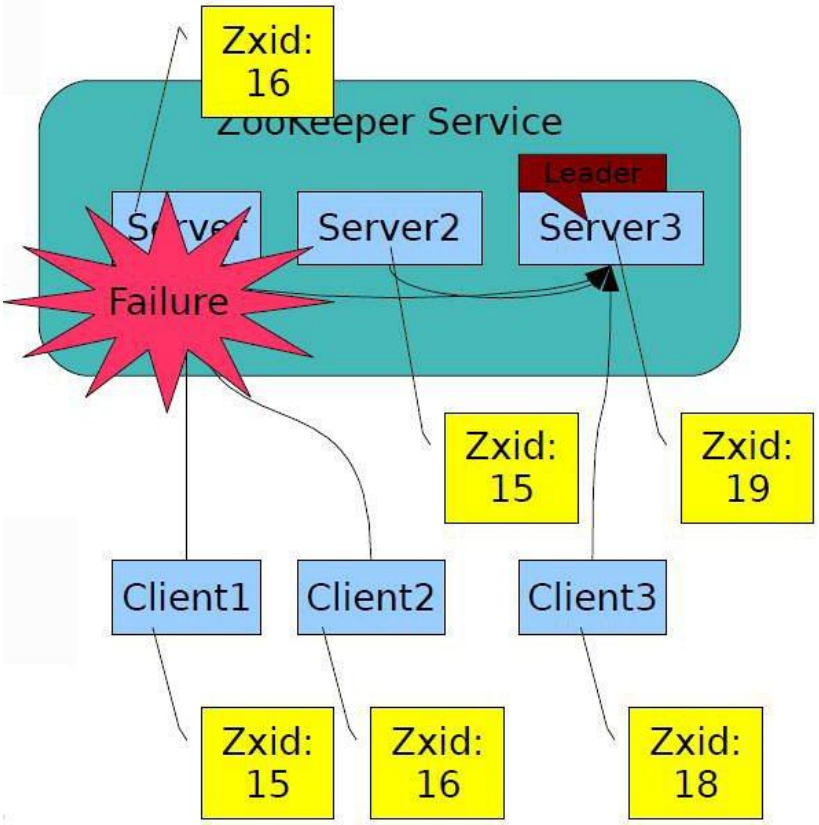

图 zookeeper 的一致性原则（摘抄自网络）

6. 选主流程

zookeeper 核心机制包括：恢复模式（选主流程）和广播模式（同步流程）。当服务刚启动、leader 崩溃、follower 不足半数时，系统就进入选主流程，此时不对外提供服务；当 leader 被选举出来后，系统就进入同步流程，server 之间完成状态同步，此后对外提供服务。本章主要讲解选主流程的原理。

选举策略主要基于 paxos 算法，一种称为 LeaderElection 算法，一种称为 FashLeaderElection 算法，系统默认使用 FashLeaderElection 算法完成。

6.1 LeaderElection 算法

流程简述如下：

1) 进入选主流程，当前发起选举的线程（选举线程）负责对投票结果进行统计，以及决定推荐哪个 server 作为 leader；

2) 选举线程初始化 currentVote，备选 leader id 默认是自己，然后向所有 server 发起一次询问，包括自己；Vote 类实际上可看成(id, zxid)这样的三元组，其中 id 表示备选 leader 的 id，zxid 表示当前最新事务 id。

```
self.setCurrentVote(new Vote(self.getId(), self.getLastLoggedZxid()));
.............
HashMap<InetSocketAddress, Vote> votes =
        new HashMap<InetSocketAddress, Vote>(self.getVotingView().size());
int xid = epochGen.nextInt();
```

3) 选举线程收到回复后，验证是否自己发起的询问，然后获取以下信息：第一，获取回复 server 的 id，存储到询问对象列表 heardFrom 中；第二，获取回复 server 提议的备选 leader 信息(id,zxid)，其中 id 是备选 leader 的 id，zxid 是事务 id 号，存储到投票记录表 votes 中。（什么是 zxid？为了保证事务的顺序一致性，zookeeper 采用了递增的事务 id 号（zxid）来标识事务。所有的提议（proposal）都在被提出的时候加上了 zxid。实现中 zxid 是一个 64 位的数字，它高 32 位是 epoch 用来标识 leader 关系是否改变，每次一个 leader 被选出来，它都会有一个新的 epoch，标识当前属于那个 leader 的统治时期。低 32 位用于递增计数）；

```
s.send(requestPacket);
responsePacket.setLength(responseBytes.length);
s.receive(responsePacket);
.............
int recvedXid = responseBuffer.getInt();
```

```
.....

long peerId = responseBuffer.getLong();

heardFrom.add(peerId);

Vote vote = new Vote(responseBuffer.getLong(),

      responseBuffer.getLong());

InetSocketAddress addr =

      (InetSocketAddress) responsePacket

      .getSocketAddress();

votes.put(addr, vote);
```

4) 所有 server 遍历请求一次后，选举线程进行投票记录计算：countVote()。每次计算需要统计两个值：result.vote 保存 zxid 最大的投票记录，result.count 是对应得票数；result.winner 记录得票数最多的投票记录，result.winningCount 是对应得票数。

```
result.vote = new Vote(Long.MIN_VALUE, Long.MIN_VALUE);

result.winner = new Vote(Long.MIN_VALUE, Long.MIN_VALUE);

Collection<Vote> votesCast = votes.values();

HashMap<Vote, Integer> countTable = new HashMap<Vote, Integer>();

// Now do the tally

for (Vote v : votesCast) {

      Integer count = countTable.get(v);

      if (count == null) {

            count = Integer.valueOf(0);

      }

      countTable.put(v, count + 1);

      .....

}

result.winningCount = 0;

LOG.info("Election tally: ");

for (Entry<Vote, Integer> entry : countTable.entrySet()) {

      if (entry.getValue() > result.winningCount) {

            result.winningCount = entry.getValue();
```

```
            result.winner = entry.getKey();
        }
        LOG.info(entry.getKey().id + "\t-> " + entry.getValue());
    }
    return result;
```

5) 对统计结果进行判断，如果 result.winner 对应的 winningCount> n/2+1，即获得了 n/2+1 的投票数，表明该备选 leader 获胜，将根据获胜 leader 的相关信息设置自己的状态；否则将 currentVote 设置为 result.vote，即选择 zxid 最大的那个 server 作为备选 leader，然后重复整个过程，直到 leader 被选举出来。

```
if (votes.size() == 0) {
        self.setCurrentVote(new Vote(self.getId(),
                        self.getLastLoggedZxid()));
} else {
        if (result.winner.id >= 0) {
                self.setCurrentVote(result.vote);
                // To do: this doesn't use a quorum verifier
                if (result.winningCount > (self.getVotingView().size() / 2)) {
                        self.setCurrentVote(result.winner);
                        s.close();
                        Vote current = self.getCurrentVote();
                        LOG.info("Found leader: my type is: " + self.getLearnerType());
                        /*
                         * We want to make sure we implement the state machine
                         * correctly. If we are a PARTICIPANT, once a leader
                         * is elected we can move either to LEADING or
                         * FOLLOWING. However if we are an OBSERVER, it is an
                         * error to be elected as a Leader.
                         */
                        if (self.getLearnerType() == LearnerType.OBSERVER) {
                                if (current.id == self.getId()) {
```

```
                        // This should never happen!
                        LOG.error("OBSERVER elected as leader!");
                        Thread.sleep(100);
                    }
                    else {
                        self.setPeerState(ServerState.OBSERVING);
                        Thread.sleep(100);
                        return current;
                    }
                } else {
                    self.setPeerState((current.id == self.getId())
                            ? ServerState.LEADING: ServerState.FOLLOWING);
                    if (self.getPeerState() == ServerState.FOLLOWING) {
                        Thread.sleep(100);
                    }
                    return current;
                }
            }
        }
    }
```

另外，从流程中可以分析得知：要使 leader 获得多数 server 的支持，server 的总数必须是 2n+1 的，且存活的 server 数目不能少于 n+1，否则选主流程无法成功，server 将会一直处于该 LOOKING 状态。

如果某 server 成为 leader，那么一定拥有最大的 zxid，也就是数据最新。此时会触发 leader 向其他 follower 状态同步的过程，保证所有 server 的数据一致。该恢复过程完毕后，才能重新接收 client 的请求。

observer 节点同样参与发送 request 和接收 response 的过程，但是 ResponderThread 的 run()方法中并不做任何处理。也就是说，observer 节点需要确定 leader 节点，但不参与选举。

6.2 FashLeaderElection 算法

流程简述如下：

1) 选举线程将 epoch+1，向所有 server 提议自己变成 leader，广播自身(id,zxid)信息。

2) 其他 server 收到提议后，需要解决 epoch 和 zxid 的冲突，然后向发起提议的 server 回复信息。（Q1：如何解决冲突？回复的是什么？）

3) 选举线程接收到其他 server 的回复，先判断自身 epoch 是否合法，然后更新 epoch 为对方推荐的 zxid、epoch，并向对方发送 ACK 消息，然后继续等待。等待一个周期结束后，如果仍未能选出 leader，则再次发送自己推荐的 leader 信息；否则可以结束。（Q2：过程不明？）

4) 重复该过程，最后一定能选举出 leader。（Q3：为什么？）

7. 状态同步

状态同步实际上就是选举完成后，leader 向 follower 同步数据的恢复过程，具体的流程分析可以按照不同角色进行阐述。

1. LEADER 流程

1) leader 设置最新的 epoch，即 zxid 的高 32 位++；

2) leader 构建 NEWLEADER 包，该包的数据是当前最大的事务 id，然后广播给所有的 follower 告知 leader 保存的事务 id 是多少，从而判断是否需要同步数据；leader 会给每个 follower 创建一个线程 LearnerHandler，负责接收它们的同步数据请求；

3) leader 主线程阻塞等待其他 follower 的回应，只有在超过半数的 follower 已经同步数据完毕，该过程才能结束，leader 才能正式成为 leader。

2. FOLLOWER 流程

follower 的同步逻辑，主要是 followLeader()方法：

1) findLeader()方法返回 leader 地址，然后 connectToLeader()连接到 leader，有重试机制，超时未连接上则退回到 LOOKING 状态；生成的 leaderIs、leaderOs 就是对应的 Archive。

2) registerLeader()方法向 leader 注册，需要传入 pktType 和 sentLastZxid 参数，前者是 packet 类型，后者是该 follower 最新的 zxid；方法返回 leader 反馈的 newLeaderZxid，同时 leader 开始向 follower 发送需要同步的信息，同步哪些信息是根据 sentLastZxid 和 newLeaderZxid 来确定的。

3) 进行 syncWithLeader()过程，与 leader 数据同步。该过程会循环收到不同类型的 packet，只有当收到 Leader.UPTODATE 类型的 packet 时，才表示同步结束。

4) 循环执行 processPacket()，直到服务停止为止。processPacket()主要是根据 leader 发送过来的 packet 类型进行不同处理。

   a) Leader.PING：执行 ping()。

   b) Leader.SYNC：执行 FollowerZookeeperServer 的 sync()。

   c) Leader.REVALIDATE：执行 revalidate()。

   d) Leader.UPTODATE：不可能，报错。

   e) Leader.COMMIT：执行 FollowerZookeeperServer 的 commit()。

   f) Leader.PROPOSAL：执行 FollowerZookeeperServer 的 logRequest()。

3. OBSERVER 流程

Observer 的逻辑所在，主要是 observeLeader()方法：

1) 与 Follower 同。

2) 与 Follower 同，不过传入的 pktType 不同。

3) 与 Follower 同。

4) processPacket()方法的处理逻辑与 Follower 不同，主要是：Leader.PROPOSAL、Leader.COMMIT 被忽略了。

8. 广播流程

经典的两阶段提交，即 leader 提起一个决议，由 followers 进行投票，leader 对投票结果进行计算决定是否通过该决议，如果通过执行该决议（事务），否则什么也不做。

9. Watch 机制

声明：该部分摘抄自网络。

Zookeeper 客户端在数据节点上设置监视，则当数据节点发生变化时，客户端会收到提醒。Zookeeper 中的各种读请求，如 getDate()，getChildren()，和 exists()，都可以选择加“监视点”(watch)。“监视点”指的是一种一次性的触发器(trigger)，当受监视的数据发生变化时，该触发器会通知客户端。

监视机制有三个关键点：

a) “监视点”是一次性的，当触发过一次之后，除非重新设置，新的数据变化不会提醒客户端。

b) “监视点”将数据改变的通知客户端。如果数据改变是客户端 A 引起的，不能保证“监视点”通知事件会在引发数据修改的函数返回前到达客户端 A。对于“监 视点”，Zookeeper 有如下保证：客户端一定是在接收到“监视”事件（watch event）之后才接收到数据的改变信息。

c) getData() 和 exists()设置关于节点数据的“监视点”，并返回节点数据信息；getChildren()设置关于子节点的“监视点”，并返回子节点信息。 setData()会触发关于节点数据的“监视点”。成功的 create()会触发所建立节点的数据“监视点”，和父节点的子节点“监视点”。成功的 delete()会触发所删除节点的数据“监视点”，和父节点的子节点“监视点”。

d) “监视点”保留在 Zookeeper 服务器上，则当客户端连接到新的 Zookeeper 服务器上时，所有需要被触发的相关“监视点”都会被触发。当客户端 断线后重连，与它的相关的“监视点”都会自动重新注册，这对客户端来说是透明的。在以下情况，“监视点”会被错过：客户端 B 设置了关于节点 A 存在性的“监视点”，但 B 断线了，在 B 断线过程中节点 A 被创建又被删除。此时，B 再连线后不知道 A 节点曾经被创建过。

Zookeeper 的“监视”机制保证以下几点：

a) “监视”事件的触发顺序和分发顺序与事件触发的顺序一致。

b) 客户端将先接收到“监视”事件，然后才收到新的数据

c) “监视”事件触发的顺序与 Zookeeper 服务器上数据变化的顺序一致

关于 Zookeeper“监视”机制的注意点：

a) “监视点”是一次性的

b) 由于“监视点”是一次性的，而且，从接收到“监视”事件到设置新“监视点”是有延时的，所以客户端可能监控不到数据的所有变化

c) 一个监控对象，只会被相关的通知触发一次。如，一个客户端设置了关于某个数据点 exists 和 getData 的监控，则当该数据被删除的时候，只会触发“文件被删除”的

通知。

d) 当客户端断开与服务器的连接时，客户端不再能收到“监视”事件，直到重新获得连接。所以关于 Session 的信息将被发送给所有 Zookeeper 服务器。由于当连接断开时收不到“监视”时间，所以在这种情况下，模块行为需要容错方面的设计。

10. ACL 控制

声明：该部分摘抄自网络。

znode 还具有原子性操作的特点：命名空间中，每一个 znode 的数据将被原子地读写。读操作读出与数据节点相关联的所有数据，写则替换该节点上的所有数据。每个节点上的“访问控制链”(ACL, Access Control List)保存了各客户端对于该节点的访问权限。

Zookeeper 利用访问控制链机制（Access Control List）控制客户端访问数据节点的权限，类似于 UNIX 文件的访问控制。但 Zookeeper 对于用户类别的区分，不止局限于所有者(owner)、组 (group)、所有人(world)三个级别。Zookeeper 中，数据节点没有“所有者”的概念。访问者利用 id 标识自己的身份，并获得与之相应的 不同的访问权限。

Zookeeper 支持可配置的认证机制。它利用一个三元组来定义客户端的访问权限：(scheme:expression, perms) 。其中，scheme 定义了 expression 的含义，如：host:host1.corp.com 标识了一个名为 host1.corp.com 的主机。Perms 标识了操作权限，如：（ip:19.22.0.0/16, READ）表示 IP 地址以 19.22 开头的主机有该数据节点的读权限。

Zookeeper 权限定义：

权限	描述	备注
CREATE	有创建子节点的权限	
READ	有读取节点数据和子节点列表的权限	
WRITE	有修改节点数据的权限	无创建和删除子节点的权限
DELETE	有删除子节点的权限	
ADMIN	有设置节点权限的权限	

Zookeeper 内置的 ACL 模式：

模式	描述
world	所有人
auth	已经被认证的用户
digest	通过 username:password 字符串的 md5 编码认证用户
host	匹配主机名后缀，如，host:corp.com 匹配 host:host1.corp.com, host:host2.corp.com，但不能匹配 host:host1.store.com
ip	通过 IP 识别用户，表达式格式为 addr/bits

11. client 启动流程

org.apache.zookeeper.ZookeeperMain 的 main 函数为入口，由 zkCli.sh 脚本调用启动。ZookeeperMain 的 commandMap 存储所有命令的 help 信息，history 则存储所有用户输入的命令，用于历史命令回朔，zk 则是 Zookeeper 类的实例化对象，是完成 client 命令的主体。

```
Map<String,String> commandMap = new HashMap<String,String>( );

HashMap<Integer,String> history = new HashMap<Integer,String>( );

Zookeeper zk;
```

client 启动流程简述如下：

```
public static void main(String args[])

    throws KeeperException, IOException, InterruptedException

{

    ZooKeeperMain main = new ZooKeeperMain(args);

    main.run();

}
```

1. 生成 ZookeeperMain 对象，构造函数中会调用 MyCommandOptions cl.parseOptions()方法解析”-server”和”-timeout”等参数信息，以及可能的输入命令，并构造 Zookeeper 对象 zk。然后执行 run()方法；
2. run()方法中，调用 cli.getCommand()方法，其返回值用于判断当前执行模式。返回值为空表示未进入交互模式，直接调用 processZKCmd(cl)处理命令，否则进入步骤 3；
3. 在交互模式中，循环获取用户输入命令并调用 executeLine()处理，首先调用 cl.parseCommand()解析命令，并将命令保存到历史中，然后调用 processZKCmd(cl)完成具体的命令执行。
    a) 判断命令是否正常；
    b) 对 quit、redo、history、printwatches、connect 等特殊命令进入处理；
    c) 对 zk.state 的状态进行判断，未连接的情况下不允许执行以下命令；
    d) 对 create、delete、set、get、ls/ls2、stat、getAcl、setAcl、setquota、listquota、delquota、close 等命令，分别调用 zk 的方法完成命令执行；
    e) 循环处理。

11.1 Zookeeper 类

```
/**
 * This is the main class of ZooKeeper client library. To use a ZooKeeper
 * service, an application must first instantiate an object of ZooKeeper class.
 * All the iterations will be done by calling the methods of ZooKeeper class.
 * The methods of this class are thread-safe unless otherwise noted.
 * <p>
 * Once a connection to a server is established, a session ID is assigned to the
 * client. The client will send heart beats to the server periodically to keep
 * the session valid.
 * <p>
 * The application can call ZooKeeper APIs through a client as long as the
 * session ID of the client remains valid.
 * <p>
 * If for some reason, the client fails to send heart beats to the server for a
 * prolonged period of time (exceeding the sessionTimeout value, for instance),
 * the server will expire the session, and the session ID will become invalid.
 * The client object will no longer be usable. To make ZooKeeper API calls, the
 * application must create a new client object.
 * <p>
 * If the ZooKeeper server the client currently connects to fails or otherwise
 * does not respond, the client will automatically try to connect to another
 * server before its session ID expires. If successful, the application can
 * continue to use the client.
 * <p>
 * Some successful ZooKeeper API calls can leave watches on the "data nodes" in
 * the ZooKeeper server. Other successful ZooKeeper API calls can trigger those
 * watches. Once a watch is triggered, an event will be delivered to the client
 * which left the watch at the first place. Each watch can be triggered only
```

```
 * once. Thus, up to one event will be delivered to a client for every watch it
 * leaves.
 * <p>
 * A client needs an object of a class implementing Watcher interface for
 * processing the events delivered to the client.
 *
 * When a client drops current connection and re-connects to a server, all the
 * existing watches are considered as being triggered but the undelivered events
 * are lost. To emulate this, the client will generate a special event to tell
 * the event handler a connection has been dropped. This special event has type
 * EventNone and state sKeeperStateDisconnected.
 *
 */
public class ZooKeeper {
      ……
}
```

Zookeeper 类实现 ZookeeperMain 提到的各命令功能，具体的方法实现可以参考《详细代码分析》章节。需要特别注意以下三个对象：

1. ClientCxcn 对象 cnxn

zookeeper 与 ClientCxcn 紧密相连。new Zookeeper()需要 new ClientCxcn()，其他例如 getData()、getChildren()的实现需要 cnxn.submitRequest()或者 cnxn.queuePacket()方法，前者为同步接口，或者为异步回调接口，实际上 submitRequest()会调用 queuePacket()但参数不同。前后者都需要 GetDataRequest request 和 GetDataResponse response 作为参数。

同步接口和异步接口的区别在于：同步接口需要阻塞等待 response 结果返回，并根据 response 进行后续判断；异步接口传入了 AsyncCallback 类型参数，用于对结果的回调处理。org.apache.zookeeper.AysncCallback 类需要实现 processResult()方法。

zkCli.sh 启动后的交互模式下，都是使用同步接口实现。

```
Packet queuePacket(RequestHeader h, ReplyHeader r, Record request,
            Record response, AsyncCallback cb, String clientPath,
            String serverPath, Object ctx, WatchRegistration watchRegistration) {
      Packet packet = null;
```

```
        synchronized (outgoingQueue) {
            if (h.getType() != OpCode.ping && h.getType() != OpCode.auth) {
                h.setXid(getXid());
            }
            packet = new Packet(h, r, request, response, null,
                    watchRegistration);
            packet.cb = cb;
            packet.ctx = ctx;
            packet.clientPath = clientPath;
            packet.serverPath = serverPath;
            if (!zooKeeper.state.isAlive() || closing) {
                conLossPacket(packet);
            } else {
                // If the client is asking to close the session then
                // mark as closing
                if (h.getType() == OpCode.closeSession) {
                    closing = true;
                }
                outgoingQueue.add(packet);
            }
        }

        sendThread.wakeup();
        return packet;
    }
```

ClientCxcn 的 SendThread sendThread 和 EventThread eventThread 也要特别注意。

- SendThread：真正的 cli 与 server 交互所在。功能 1：管理心跳；功能 2：管理 packet 出队列。
- EventThread：运行一个独立线程，维护一个 LinkedBlockingQueue<Object>类型的 waitingEvents，里面存储 WatcherSetEventPair 或者 Packet，也就是一个阻塞式的 FIFO

队列。EventThread 类是为了 watcher 以及 callback 而存在的。

1) queueEvent()负责将 WatchEvent 转化为 WatcherSetEventPair 并加入到 waitingEvents 中，需要使用到 Zookeeper 对象传入的 ClientWatchManager 对象。
2) queuePacket()负责将 Packet 加入到 waitingEvents 中。
3) run()：线程运行主体，循环处理 waitingEvents，当收到 eventOfDeath 时标识为 killed，等待 waitingEvents 被执行清空。processEvent()负责具体处理，当为 WatcherSetEventPair 时，负责执行各 watcher 的 process()方法；当为 Packet 且 CallBack 不为空时，负责执行 CallBack 的 processResult()方法。

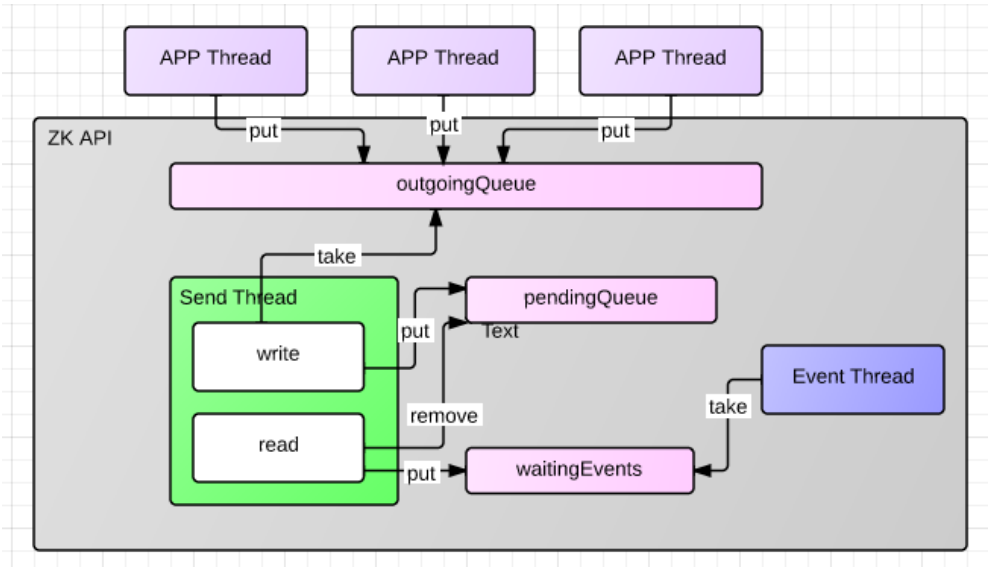

图 IO 模型（摘抄自网络）

2. ZKWatchManager

3. DataWatchRegistration 各子类

上述类都与 Watch 机制有关，重点放在《Watch 机制》章节讲述。

12. server 启动流程

org.apache.zookeeper.server.quorum.QuorumPeerMain 的 main 方法是 server 启动入口，被 zkServer.sh 脚本调用。main 主体会 new QuorumPeerMain 对象并调用 initializeAndRun()方法，开始 server 的启动流程。

```
protected void initializeAndRun(String[] args)
        throws ConfigException, IOException
{
        QuorumPeerConfig config = new QuorumPeerConfig();
        if (args.length == 1) {
                config.parse(args[0]);
        }

        if (args.length == 1 && config.servers.size() > 0) {
                runFromConfig(config);
        } else {
                LOG.warn("Either no config or no quorum defined in config, running "
                                + " in standalone mode");
                // there is only server in the quorum -- run as standalone
                ZooKeeperServerMain.main(args);
        }
}
```

具体流程如下：

1. 解析配置文件并获得 QuorumPeerConfig 对象，根据该对象判断运行何种模式。配置不存在或者 config.servers.size() <= 0 时，运行 standalone 模式，即 ZookeeperServerMain.main()；配置存在且 config.servers.size() > 0 时，运行 quorum 模式，即 QuorumPeer.start()和 QuorumPeer.join()，显然这是个新线程。我们在此仅分析 quorum 模式。

2. QuorumPeer 重写了 start()方法，负责具体的启动流程：

   a) zkDb.loadDatabase()：将 disk 的内容加载到 memory 中。
   b) cnxnFactory.start()：开启与 client 的通信。
   c) startLeaderElection()：选举过程。首先生成默认 Vote 对象，id 对应自身 id，zxid

对应自身处理过的最新 zxid；然后启动 ResponderThread.start()，监听其他节点的询问请求；最后生成选举算法，比如 LeaderElection、AuthFastLeaderElection、AuthFastLeaderElection、FastLeaderElection 等等。

d) super.start()：执行 run()。先是按序生成 QuorumBean、LocalPeerBean、RemotePeerBean 三个对象，并将对象 register 到 MBeanRegistry 实例中（有什么用不知道……）；然后进入 main 循环中，根据当前的服务状态判断如何处理，基本上可以认为这里负责了**选主过程、leader 服务过程、follower 服务过程、observer 服务过程**，并且使用循环解决当某过程完毕/退出后如何继续执行的问题，比如 leader 服务过程退出后进入选主过程、follower 服务过程退出后进入选主过程、observer 服务过程退出后进入选主过程（但不做任何投票，仅与 leader 同步）。

a) 选举过程执行 lookForLeader()方法，follower 服务过程执行 followLeader()方法，leader 服务过程执行 lead()方法，observer 服务过程执行 observeLeader()方法。

```
try {
    /*
     * Main loop
     */
    while (running) {
        switch (getPeerState()) {
        case LOOKING:
            try {
                LOG.info("LOOKING");
                setCurrentVote(makeLEStrategy().lookForLeader());
            } catch (Exception e) {
                LOG.warn("Unexpected exception",e);
                setPeerState(ServerState.LOOKING);
            }
            break;
        case OBSERVING:
            try {
                LOG.info("OBSERVING");
                setObserver(makeObserver(logFactory));
                observer.observeLeader();
```

```
                } catch (Exception e) {
                    LOG.warn("Unexpected exception",e );
                } finally {
                    observer.shutdown();
                    setObserver(null);
                    setPeerState(ServerState.LOOKING);
                }
                break;
            case FOLLOWING:
                try {
                    LOG.info("FOLLOWING");
                    setFollower(makeFollower(logFactory));
                    follower.followLeader();
                } catch (Exception e) {
                    LOG.warn("Unexpected exception",e);
                } finally {
                    follower.shutdown();
                    setFollower(null);
                    setPeerState(ServerState.LOOKING);
                }
                break;
            case LEADING:
                LOG.info("LEADING");
                try {
                    setLeader(makeLeader(logFactory));
                    leader.lead();
                    setLeader(null);
                } catch (Exception e) {
                    LOG.warn("Unexpected exception",e);
```

```
                } finally {
                    if (leader != null) {
                        leader.shutdown("Forcing shutdown");
                        setLeader(null);
                    }
                    setPeerState(ServerState.LOOKING);
                }
                break;
            }
        }
    } finally {
        LOG.warn("QuorumPeer main thread exited");
        try {
            MBeanRegistry.getInstance().unregisterAll();
        } catch (Exception e) {
            LOG.warn("Failed to unregister with JMX", e);
        }
        jmxQuorumBean = null;
        jmxLocalPeerBean = null;
    }
```

12.1 QuorumPeer 类

QuorumPeer 负责并管理 quorum 协议，implements Thread

- 内置 QuorumServer 类：addr 存储 leader 地址，electionAddr 存储选举用地址，id 存储 server 的 ID，type 存储节点类型。
- 内置 ServerState 枚举类型，表示服务状态：
  a) LOOKING
  b) FOLLOWING
  c) LEADING
  d) OBSERVING

- 内置 LearnerType 枚举类型，表示节点类型：
  a) PARTICIPANT
  b) OBSERVER
- 内置 ResponderThread 类，run()方法循环阻塞等待 udp socket，负责根据其他节点发来的询问请求返回应答信息，response 则根据本地服务状态进行处理。注意，该 ResponderThread 用于服务 LeaderElection 选举，其他选举机制例如 FastLeaderElection 貌似没有用到。
  a) LOOKING 状态：xid = xid，server id = myid，leader id = currentVote.id，zxid = currentVote.zxid。
  b) LEADING 状态：xid = xid，server id = myid，leader id = myid，zxid = leader.lastProposed。
  c) FOLLOWING 状态：xid = xid，server id = myid，leader id = currentVote.id，zxid = follower.zxid。
  d) OBSERVING 状态：忽略。

13. 开源客户端框架 Curator

略。

14. 应用场景

以下部分摘抄自其他文章，出处已不详。

zookeeper 从设计模式角度来看，是一个基于观察者模式设计的分布式服务管理框架，它负责存储和管理大家都关心的数据，然后接受观察者的注册，一旦这些数据的状态发生 变化，zookeeper 就将负责通知已经在 zookeeper 上注册的那些观察者做出相应的反应，从而实现集群中类似 Master/Slave 管理模式，关于 zookeeper 的详细架构等内部细节可以阅读 zookeeper 的源码。

14.1 统一命名服务（Name Service）

分布式应用中，通常需要有一套完整的命名规则，既能够产生唯一的名称又便于人识别和记住，通常情况下用树形的名称结构是一个理想的选择，树形 的名称结构是一个有层次的目录结构，既对人友好又不会重复。说到这里你可能想到了 JNDI，没错 Zookeeper 的 Name Service 与 JNDI 能够完成的功能是差不多的，它们都是将有层次的目录结构关联到一定资源上，但是 Zookeeper 的 Name Service 更加是广泛意义上的关联，也许你并不需要将名称关联到特定资源上，你可能只需要一个不会重复名称，就像数据库中产生一个唯一的数字主键一样。

Name Service 已经是 Zookeeper 内置的功能，你只要调用 Zookeeper 的 API 就能实现。如调用 create 接口就可以很容易创建一个目录节点。

14.2 配置管理（Configuration Management）

配置的管理在分布式应用环境中很常见，例如同一个应用系统需要多台 PC Server 运行，但是它们运行的应用系统的某些配置项是相同的，如果要修改这些相同的配置项，那么就必须同时修改每台运行这个应用系统的 PC Server，这样非常麻烦而且容易出错。

像这样的配置信息完全可以交给 Zookeeper 来管理，将配置信息保存在 Zookeeper 的某个目录节点中，然后将所有需要修改的应用机器监控配置信息的状态，一旦配置信息发生变化，每台应用机器就会收到 Zookeeper 的通知，然后从 Zookeeper 获取新的配置信息应用到系统中。

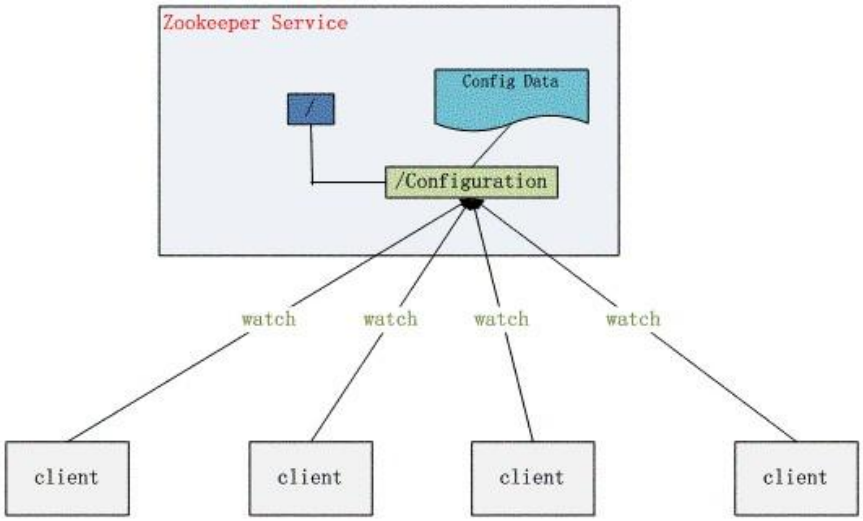

图 配置管理示例

14.3 集群管理（Group Membership）

Zookeeper 能够很容易的实现集群管理的功能，如有多台 Server 组成一个服务集群，那么必须要一个“总管”知道当前集群中每台机器的服务状态，一旦有机器不能提供服务，集群中其它集群必须知道，从而做出调整重新分配服务策略。同样当增加集群的服务能力时，就会增加一台或多台 Server，同样也必须让“总管”知道。

Zookeeper 不仅能够帮你维护当前的集群中机器的服务状态，而且能够帮你选出一个“总管”，让这个总管来管理集群，这就是 Zookeeper 的另一个功能 Leader Election。

它们的实现方式都是在 Zookeeper 上创建一个 EPHEMERAL 类型的目录节点，然后每个 Server 在它们创建目录节点的父目录节点上调用 getChildren(String path, boolean watch) 方法并设置 watch 为 true，由于是 EPHEMERAL 目录节点，当创建它的 Server 死去，这个目录节点也随之被删除，所以 Children 将会变化，这时 getChildren 上的 Watch 将会被调用，所以其它 Server 就知道已经有某台 Server 死去了。新增 Server 也是同样的原理。

Zookeeper 如何实现 Leader Election，也就是选出一个 Master Server。和前面的一样每台 Server 创建一个 EPHEMERAL 目录节点，不同的是它还是一个 SEQUENTIAL 目录节点，所以它是个 EPHEMERAL_SEQUENTIAL 目录节点。之所以它是 EPHEMERAL_SEQUENTIAL 目录节点，是因为我们可以给每台 Server 编号，我们可以选择当前是最小编号的 Server 为 Master，假如这个最小编号的 Server 死去，由于是 EPHEMERAL 节点，死去的 Server 对应的节点也被删除，所以当前的节点列表中又出现一个最小编号的节点，我们就选择这个节点为当前 Master。这样就实现了动态选择 Master，避免了传统意义上单 Master 容易出现单点故障的问题。

集群管理配置如图所示。

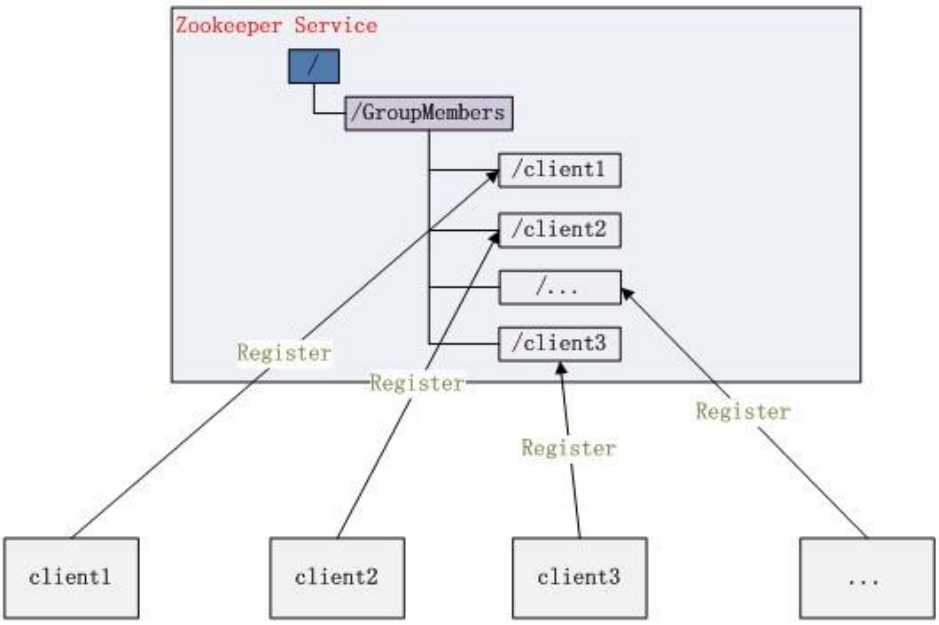

图 集群管理示例

14.4 共享锁（Locks）

共享锁在同一个进程中很容易实现，但是在跨进程或者在不同 Server 之间就不好实现了。Zookeeper 却很容易实现这个功能，实现方式也是需要获得锁的 Server 创建一个 EPHEMERAL_SEQUENTIAL 目录节点，然后调用 getChildren 方法获取当前的目录节点列表中最小的目录节点是不是就是自己创建的目录节点，如果正是自己创建的，那么它就获得了这个锁，如果不是那么它就调用 exists(String path, boolean watch) 方法并监控 Zookeeper 上目录节点列表的变化，一直到自己创建的节点是列表中最小编号的目录节点，从而获得锁，释放锁很简单，只要删除前面它自己所创建的目录节点就行了。

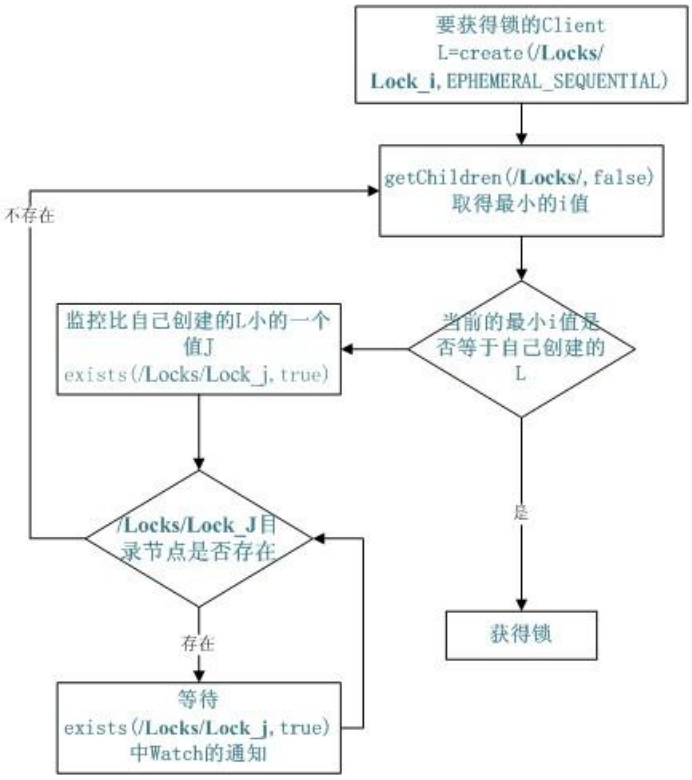

图 共享锁机制

14.5 队列管理

Zookeeper 可以处理两种类型的队列：

1) 当一个队列的成员都聚齐时，这个队列才可用，否则一直等待所有成员到达，这种是同步队列。

2) 队列按照 FIFO 方式进行入队和出队操作，例如实现生产者和消费者模型。

同步队列用 Zookeeper 实现的实现思路如下：

创建一个父目录 /synchronizing，每个成员都监控标志（Set Watch）位目录 /synchronizing/start 是否存在，然后每个成员都加入这个队列，加入队列的方式就是创建 /synchronizing/member_i 的临时目录节点，然后每个成员获取 / synchronizing 目录的所有目录节点，也就是 member_i。判断 i 的值是否已经是成员的个数，如果小于成员个数等待 /synchronizing/start 的出现，如果已经相等就创建 /synchronizing/start。

用流程图更容易理解：

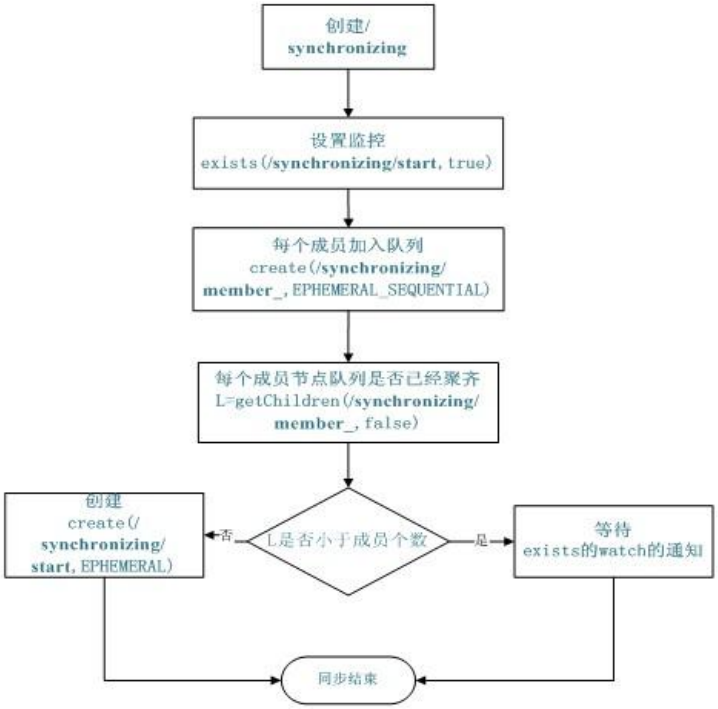

图 队列管理机制

FIFO 队列用 Zookeeper 实现思路如下：

实现的思路也非常简单，就是在特定的目录下创建 SEQUENTIAL 类型的子目录/queue_i，这样就能保证所有成员加入队列时都是有编号的，出队列时通过 getChildren( ) 方法可以返回当前所有的队列中的元素，然后消费其中最小的一个，这样就能保证 FIFO。

利用 Zookeeper 实现带优先级的队列，只需要简单地修改普通队列中节点的命名方式：以“queue-YY”来做队列元素的前缀，其中 YY 为节点的优先级，依据 Linux 的方式，数值越小，优先级越高。并且，处理队列的客户端，在处理完一个节点之后，需要调用 sync()以保证有优先级高的节点插入时，能够先获得处理资源。

14.6 障碍墙（Barriers）

分布式系统利用障碍墙来保证对于一组数据节点的处理被某个条件阻塞。直到条件被满足的时候，所有数据节点同时开始处理。在 Zookeeper 中，可以利用一个“障碍墙节点”（算法中使用 b 表示该节点）来实现这个功能：

1、客户端对于该“障碍墙”节点调用 exists(b, true)函数，设置 watch
2、如果 exists()返回 false，“障碍墙”节点不存在，客户端继续执行
3、如果 exists()返回 true，客户端等待“障碍墙”节点的 watch 事件
4、当 watch 事件被激发时，跳回 1 执行

14.7 双重障碍墙（Double Barriers）

双重障碍墙用于保证客户端同步开始和结束某个计算过程。当足够的客户端被障碍墙阻挡的时候，计算开始执行。当所有计算都结束的时候，所有客户端一起脱离障 碍墙。利用 Zookeeper 实现双重障碍墙的同步机制，在计算启动之前，客户端通过在“障碍墙节点”（算

法中使用 b 表示该节点）注册来加入同步过程。而 在计算结束时，客户端从“障碍墙节点”注销。

在以下算法代码中，n 是客户端的 Zookeeper 名字标识，p 标识了一个客户端，pmax 标识了编号最大的一个客户端，pmin 标识了编号最小的一个客户端。

进入
1、创建名字 n = b + “/” +p
2、设置“监视点”：exists(b + “/ready”, true)
3、创建子节点：create(n, EPHEMERAL)
4、L = getChildren(b, false)
5、如果 L 中的子节点个数少于 x（根据应用配置），客户端等待“监视”事件
否则 create(b + “/ready”, REGULAR)

如算法所示，在进入“障碍墙”的过程中，客户端在“障碍墙”节点下创建代表自己的临时节点。当最后一个客户端进入后，它能检测到“障碍墙”节点已经有 x 个子节点，此时，该客户端创建“ready”节点，通知等待“监视”事件的所有客户端同时开始计算过程。

退出
1、L = getChildren(b, false)
2、如果没有 L 中没有子节点，退出
3、如果 p 是 L 中唯一的子节点，delete(n)，并退出
4、如果 p 是 L 中编号最小的节点，等待在 pmax 之上
否则，delete(n)（如果 n 还存在），等待在 pmin 之上
5、跳回 1 执行

在计算结束，所有客户端需要删除各自子节点，并同时离开“障碍墙节点”。注意，在上述退出协议中，最后一个被删除的子节点是序号最小的子节点，该子节点对 应的客户端将“监视点”设置在当前序号最大子节点上。当该序号最大的子节点被删除后，继续选择当时序号最大的子节点等待。所有其他客户端都等待在序号最小 的子节点上，当该节点被删除之后，所有其他的客户端同时离开。

14.8 互斥锁

分布式互斥锁的定义是在任意时刻，分布式系统中的两个客户端不会同时认为自身持有同一个互斥锁。同样，首先我们需要定义一个锁节点，想要获得锁的客户端，按照以下过程操作：

获得锁
1、调用 create()函数，设置路径名“_locknode_/lock-”，并且设置“序列(sequence)”和“暂时性(ephemeral)”选项
2、对锁节点调用 getChildren()函数，并不设置“监视点”（注意，此处不能设置“监视点”）
3、如果 1 中创建的子节点序号是子节点中最小的序号，则该客户端获得了互斥锁，退出
4、对比 1 中创建的子节点序号小的最大的子节点调用 exists()函数，并设置“监视点”
5、如果 exists()返回 true，等待“监视”事件通知，并跳回 2
否则，跳回 2
释放锁
1、已经获得锁的客户端要释放锁的话，只需要删除之前创建的子节点便可

需要注意的几点：

1) 每次删除锁节点的子节点时，只会唤醒一个客户端。

2) 在这个实现方案中不需要轮询和超时处理

14.9 读写锁

利用 Zookeeper 实现的读写锁，是在互斥锁的基础上实现的。

获得读锁
1、调用 create()函数，设置路径名“_locknode_/read-”，并且设置“序列(sequence)”和“暂时性(ephemeral)”选项
2、对锁节点调用 getChildren()函数，并不设置“监视点”（注意，此处不能设置“监视点”）
3、如果不存在其子节点序号比 1 中创建的子节点序号小的“write-”子节点，当前的客户端获得读锁，退出
4、对于序号小于 1 中创建的子节点序号的“write-”调用 exists()，并设置“监视点”
5、如果 exists()返回 true，等待“监视事件”通知，并跳回 2
否则，跳回 2
释放读锁
1、删除客户端之前创建的子节点
获得写锁
1、调用 create()函数，设置路径名“_locknode_/read-”，并且设置“序列(sequence)”和“暂时性

(ephemeral)”选项
2、对锁节点调用 getChildren()函数，并不设置“监视点”（注意，此处不能设置“监视点”）
3、如果 1 中创建的子节点序号是子节点中最小的序号，则该客户端获得了写锁，退出
4、对比 1 中创建的子节点序号小的最大的子节点调用 exists()函数，并设置“监视点”
5、如果 exists()返回 true，等待“监视”事件通知，并跳回 2
否则，跳回 2
释放写锁
1、删除客户端之前创建的子节点

注意，此处实现之中有多个”read-”子节点同时等待在”write-”子节点上，但这样的设计不会有副作用。这是因为，”read-”子节点在被唤醒的时候，都能获得处理资源，而不是只有某个或者某些客户端能继续执行。

14.10 可恢复(recoverable)的读写锁

通过对读写锁做不大的修改，便可实现简单的可恢复读写锁（用于防止某些锁被客户端长期持有）。方法如下：

无论是在申请读锁还是申请写锁时，在调用 create()之后，立即对于该节点调用 getData()并设置“监视点”（注意，如果 getData()接 收到了 create 事件，则重新 getData()，并设置“监视点”）。getData()用于监视该节点数据中是否出现“unlock”字符串。需要 锁的其他客户端通过向该节点数据中写入”unlock”来提醒持有锁的客户端释放锁。

注意在这个可恢复读写锁的实现机制中，有一个关键点，即，持有锁的“客户端”必须同意释放锁。在很多情况下，持有锁的客户端需要保留锁来进行未完成的操作。

14.11 二阶段提交(Two-phased Commit)

二阶段提交机制用于保证分布式系统中的客户端都同意提交或者放弃某事务。

在 Zookeeper 中，可以在有协调客户端(coordinator)的基础上，实现二阶段提交机制。首先协调客户端创建一个事务节点（如：/app /Tx），为每个参与客户端建立一个子节点（如：/app/Tx/s_i），这些子节点的数据为空。当各个参与客户端接收到协调客户端发送的事务时，它们 别的客户端对应的节点设置“监视”。并通过写自己对应的节点的数据来告诉别的客户端自己是否确认提交事务。需要注意的是，由于很多情况下，只要有一个客户 端不能确认提交，事务就会被丢弃，所以整个处理时间可能很短。

15. 详细代码分析

org.apache.zookeeper.CreateMode

1. PERSISTENT：永久节点
2. EPHEMERAL：临时节点
3. PERSISTENT_SEQUENTIAL：永久节点、序列化
4. EPHEMERAL_SEQUENTIAL：临时节点、序列化

org.apache.zookeeper.Environment

static list()方法可以获取系统属性值列表，例如 java.home、java.class.path、user.home、user.dir 等。

org.apache.zookeeper.WatchEvent

属性包括：KeeperState、EventType、path

1. KeeperState：Disconnected（未连接）、SyncConnected（已连接）、AuthFailed（权限错误）、Expired（session 超时）
2. EventType：None（无）、NodeCreated（创建节点）、NodeDeleted（删除节点）、NodeDataChanged（更新节点内容）、NodeChildrenChanged（更新子节点）
3. path：节点所在路径

org.apache.zookeeper.Watcher

abstract process(WatchEvent event)，一般在 zk = new Zookeeper()时，可以指定 watcher，由该 watcher 接收任何更新信息。比如在 zk = new Zookeeper()后阻塞等待，当 watcher 的 process 线程接收到 event 的 KeeperState 为 SyncConnected 的事件后，唤醒主线程的阻塞。

org.apache.zookeeper.ZooDefs

定义了 OpCode、 Perms 和 Ids，其中：

1. OpCode：诸如 create、delete、setData、getData、getChildren 等操作的操作码。
2. Perms：诸如 ALL、READ、WRITE、CREATE、DELETE、ADMIN 等权限。
3. Ids：诸如 OPEN_ACL_UNSAFE、CREATOR_ALL_ACL、READ_ACL_UNSAFE 等列表信息。

org.apache.zookeeper.ZookeeperMain

zkCli.sh 脚本的启动入口，具体做了以下几件事情：

1. 定义 static List<String> commandMap，作为 help 帮助和判断是否支持该命令的条件。
2. 解析用户输入参数，获取-server、-timeout 参数信息，以及可能输入的 zk 命令。

3. 连接 zk 的 server。
4. 执行命令，有两种：如果在 zkCli.sh 启动时已经指定 zk 命令，则一次性执行完毕后打印结果退出；否则进入交互模式。
5. 交互模式下，去掉空命令行，先 parseCommand()再 processCmd()，并且记录输入历史到 history 列表中。parseCommand：解析命令到 MyCommandOptions 对象中；processCmd：不同的命令不同处理：
   a) help：usage()。
   b) quit：zk.close() && System.exit(0)。
   c) redo num：history 中获取历史命令，然后 processCmd()递归调用。
   d) history：history 中获取最新历史命令列表。
   e) printwatches：打印 printwatches 状态。
   f) connect [host]：先关闭上次连接，再连接新地址。
   g) create path data：调用 zk.create()。
   h) delete path [watch]：调用 zk.delete().
   i) set path data [version]：调用 zk.setData()。
   j) get path：调用 zk.getData()。
   k) aget path：调用 zk.getData()，但是多了 DataCallBack()回调对象。
   l) ls path：调用 zk.getChildren()。
   m) ls2 path：调用 zk.getChildren()，多了 stat 参数，也就是说会返回节点状态。
   n) getAcl/setAcl：略。
   o) stat path：调用 zk.exists()。
   p) listquota/setquota/delquota：略。
   q) close：调用 zk.close()。
   r) addauth schema:id:perms：调用 zk.addAuthInfo()，参数略。

org.apache.zookeeper.Zookeeper

实现了 ZookeeperMain 中 processCmd()调用的所有方法，主要包含三个主要实体：

1. ClientCxcn 对象 cnxn：负责所有方法的具体实现。
2. 类似 GetDataRequest 类型的对象 request：存储 request 的所有信息。
3. 类似 GetDataResponse 类型的对象 response：存储 response 的所有信息。

注意 1：所有 request/response 类型都在 org.apache.zookeeper.proto 中实现。

注意 2：Zookeeper 与 ClientCxcn 紧密相连。new Zookeeper()需要 new ClientCxcn()，其他例如 getData()、getChildren()的实现需要 cnxn.submitRequest()或者 cnxn.queryPacket()方法，前者为同步接口，或者为异步回调接口。前后者都需要 request 和 response 作为参数。

内置 ZkWatchManager 类，负责管理所有注册的 watcher。该类继承自 ClientWatchManager，需要实现 materialize()方法，该方法根据不同的 EventType 进行处理，返回需要通知到的 watcher 列表。

1. NodeChildChanged：将 clientPath 对应的 childWatches 列表加入 result 中。
2. NodeDeleted：将 clientPath 对应的 dataWatches、existsWatches 和 childWatches 列表加入 result 中。
3. NodeCreated/NodeDataChanged：将 clientPath 对应的 dataWatches 和 existsWatches 列表加入 result 中。
4. None：所有 dataWatches、existsWatches 和 childWatches 列表都通知。

注意 1：通过代码可以看到，从 dataWatches、existsWatches 和 childWatches 列表 remove 获取列表后加入 result 中，也就是注册的 watcher 仅触发一次就失效了。

注意 2：当 EventType 为 None 且 KeeperState 为 SyncConnected 时，不需要 remove，也就是说 watcher 没失效。

内置 WatchRegistration 类的子类，包括 DataWatchRegistration、ChildWatchRegistration、ExistsWatchRegistration，目的是记录一次注册的 watcher 和对应 path，register()方法在基类中已实现，会保存到 getWatches()得到的列表中，一般是 ZkWatchManager 对象中的对应 watcher 列表。

需要注意 session 机制。client 连接 server 成功后，server 赋予该 client 一个 sessionid，client 需要不断发送心跳维持 session 有效，在 session 有效期内，可以使用 Zookeeper 提供的 API 进行操作。

如果因为某些原因导致 client 无法正常发送心跳，在超时时长后，server 会判断该 client 的 session 失效，此时 client 发送的任何操作都会被拒绝，并触发 ExpiredException 异常。

如果 client 的心跳无法正常连接 server，会自动在 session 超时前尝试连接其他 server，连接成功后可以继续操作。

如果 client 取消当前连接并连接其他 server，已存在的 watches 会丢失，取而代之的是 client 会生成一个特殊 WatchEvent 告诉本地 watcher 连接已经丢失，该 WatchEvent 是：EventType：None，KeeperState：DisConnected。

Zookeeper 各功能 API 的具体实现：略。

org.apache.zookeeper.AysncCallback

各种 callback 类的定义，当使用 Zookeeper 的 API 时，可能会使用到异步 API，此时需要定义对应类型的 callback 对象并作为 API 参数，需要实现 processResult()方法。

org.apache.zookeeper.ClientCnxn

几个重要接口：

1. Packet 类：保存本次操作的所有信息，例如：RequestHeader、ReplyHeader、request、response、clientPath/serverPath、AsyncCallBack、WatchRegistration、chrootPath、serverAddrs 等等，以及包含了 header 和 request 的包 ByteBuffer。
2. submitRequest()：就是 queryPacket()的进一步封装，仅支持同步方式。
3. queryPacket()：将操作封装为 Packet，并 add 到 outgoingQueue 中。
4. SendThread 类：真正的 cli 与 server 交互所在。功能 1：管理心跳；功能 2：管理出队列。（略）
5. EventThread 类：运行一个独立线程，维护一个 LinkedBlockingQueue<Object>类型的 waitingEvents，里面存储 WatcherSetEventPair 或者 Packet，也就是一个阻塞式的 FIFO 队列。
    a) queryEvent() 负责将 WatchEvent 转化为 WatcherSetEventPair 并加入到 waitingEvents 中，需要使用到 Zookeeper 对象传入的 ClientWatchManager 对象。
    b) queryPacket()负责将 Packet 加入到 waitingEvents 中。
    c) run()：线程运行主体，循环处理 waitingEvents，当收到 eventOfDeath 时标识为 killed，等待 waitingEvents 被执行清空。processEvent()负责具体处理，当为 WatcherSetEventPair 时，负责执行各 watcher 的 process()方法；当为 Packet 且 CallBack 不为空时，负责执行对应的 processResult()方法。

简而言之，EventThread 类是为了 watcher 以及 callback 而存在的。

org.apache.zookeeper.proto.ReplyHeader

Record 的子类，存储三个属性：

1. xid：未知。
2. zxid：事务的 id 号。
3. err：错误号，0 表示无错误。

org.apache.zookeeper.proto.RequestHeader

Record 的子类，存储两个属性：

1. xid：未知。
2. type：操作类型编号。

org.apache.zookeeper.proto.***Request/***Response

Record 的子类，存储了对应类型操作的 request 信息和 response 信息，没有什么特别需要注意的地方，两者是一一映射的，什么类型的操作用什么类型的 request 和 response。

org.apache.zookeeper.server.quorum.QuorumPeerMain

zkServer.sh 的入口。

解析配置文件并获得 QuorumPeerConfig 对象，根据该对象判断运行何种模式：

1. 配置不存在或者 config.servers.size() <= 0 时，运行 standalone 模式，即 ZookeeperServerMain.main（）。
2. 配置存在且 config.servers.size() > 0 时，运行 quorum 模式，即 QuorumPeer.start() 和 QuorumPeer.join()，显然这是个新线程。

org.apache.zookeeper.server.quorum.QuorumPeerConfig

dataDir：数据存放路径

dataLogDir：日志存放路径

tickTime：基准时间片

clientPortAddress：client 可连接的地址端口

maxClientCnxns：最大 client 并发？

minSessionTimeout：最小 session 超时时间

maxSessionTimeout：最大 session 超时时间

peerType：节点类型，默认是 LearnerType. PARTICIPANT

serverId：节点 ID

serverWeight：节点权重列表，元素类型为 List<long, long>

serverGroup：节点组列表，元素类型为 List<long, long>

numGroups：节点组数量

electionAlg：选举参数？

electionPort：选举端口

servers：服务节点列表，元素类型为 List<long, QuorumServer>

observers：观察节点列表，元素类型为 List<long, QuorumServer>

quorumVerifier：？

initLimit：follower 向 leader 初始化同步的时间

syncLimit：follower 向 leader 一次请求往返的时间

org.apache.zookeeper.server.quorum.QuorumPeer

QuorumPeer 负责并管理 quorum 协议，implements Thread。

- 内置 QuorumServer 类：addr 存储 leader 地址，electionAddr 存储选举用地址，id

存储 server 的 ID，type 存储节点类型。

- 内置 ServerState 枚举类型，表示服务状态：
  a) LOOKING
  b) FOLLOWING
  c) LEADING
  d) OBSERVING
- 内置 LearnerType 枚举类型，表示节点类型：
  a) PARTICIPANT
  b) OBSERVER
- 内置 ResponderThread 类，run()方法循环阻塞等待 udp socket，负责根据其他节点发来的询问请求返回应答信息，response 则根据本地服务状态进行处理：
  a) LOOKING 状态：xid = xid，server id = myid，leader id = currentVote.id，zxid = currentVote.zxid。
  b) LEADING 状态：xid = xid，server id = myid，leader id = myid，zxid = leader.lastProposed。
  c) FOLLOWING 状态：xid = xid，server id = myid，leader id = currentVote.id，zxid = follower.zxid。
  d) OBSERVING 状态：忽略。
- 重写了 start()方法，执行步骤是：
  a) zkDb.loadDatabase()：将 disk 的内容加载到 memory 中。
  b) cnxnFactory.start()：开启与 client 的通信。
  c) startLeaderElection()：选举过程。首先生成默认 Vote 对象，id 对应自身 id，zxid 对应自身处理过的最新 zxid；然后启动 ResponderThread.start()，监听其他节点的询问请求；最后生成选举算法，比如 LeaderElection、AuthFastLeaderElection、AuthFastLeaderElection、FastLeaderElection 等等。
  d) super.start()：执行 run()。先是按序生成 QuorumBean、LocalPeerBean、RemotePeerBean 三个对象，并将对象 register 到 MBeanRegistry 实例中（有什么用不知道……）；然后进入 main 循环中，根据当前的服务状态判断如何处理，基本上可以认为这里负责了选主过程、leader 服务过程、follower 服务过程、observer 服务过程，并且使用循环解决当某过程完毕/退出后如何继续执行的问题，比如 leader 服务过程退出后进入选主过程、follower 服务过程退出后进入选主过程、observer 服务过程退出后进入选主过程（但不做任何投票，仅与 leader 同步）。

关注 election 的 lookForLeader()方法，关注 follower 的 followLeader()方法，关注 leader

的 lead()方法，关注 observer 的 observeLeader()方法。

org.apache.zookeeper.jmx.MBeanRegistry

略。

org.apache.zookeeper.server.quorum.Vote

记录投票信息：id、zxid、epoch、state。

org.apache.zookeeper.server.quorum.Election

- Election：接口，需要子类实现 lookForLeader()和 shutdown()方法。
- LeaderElection：在 QuorumPeer 的 run 方法中，当 KeeperState 为 LOOKING 时，会调用具体 Election 的 lookForLeader()方法进行选举。选举策略为：
  a) 在这里，self 表示 QuorumPeer 对象。
  b) 将 currentVote 设置为自身：(self.getId(), self. getLastLoggedZxid())，向其余 PARTICIPATION 节点发送 request 信息：(xid)，并依次获取 response 信息：(xid, peerId, leaderId, zxid)。需要注意 response 信息，xid 表示发送节点，peerId 表示反馈节点，leaderId 表示反馈节点认为的 leader 节点，zxid 表示最近的操作序列号；需要将 resonpse 信息转换为 votes 列表和 heardFrom 列表，前者由 leaderId、zxid 构造，后者由 peerId 构造。
  c) 进行 countVote()操作，主要几个作用：不在 heardFrom 中但在 votes 中的节点，需要清除；votes 中同样的节点，一致改为最大的 zxid；计算生成 result.vote 和 result.winner，前者记录 zxid 最大的 Vote，后者记录计数最多的 Vote。
  d) 如果 result.winner.id 有效，则 currentVote 改为 result.vote；如果 result.winner 的计数超过半数，则 currentVote 改为 result.winner，否则继续循环执行 a)步骤。超过半数的情况下，各节点更新自己的状态值，或者是 leader，或者是 follower，或者是 observer。
  e) 该节点的选举结束，ResponderThread 对象开启的端口依然存在，用于回答其他节点的请求并反馈应答，只不过应答的结果已经是固定的了（自身是 leader，则应答为：我是 leader；自身是 follower，则应答为：某个节点是 leader）；当 follower 超过半数后，集群趋于稳定。

  注意 1：observer 节点同样参与发送 request 和接收 response 的过程，但是 ResponderThread 的 run()方法中并不做任何处理。也就是说，observer 节点需要确定 leader 节点，但不参与选举。

  注意 2：observer 初始时会设置自身为 currentVote()，不知道对选举有没有影响？它只有通过 response 来更新。

  示例 1：有 5 个节点组成的集群，每个节点都是第一次启动，那么：

  a) 节点 1 启动，设置自身为 currentVote(1, MIN_VALUE)，发送 request(1)不成功；此时 votes 列表为空，节点 1 依然设置自身为 currentVote，并不断 sleep-try 尝试发送 request。

b) 节点 2 启动，设置自身为 currentVote(2, MIN_VALUE)，节点 1 与节点 2 互相交换选举结果，id 值较大的节点 2 胜出，此时节点 1 和节点 2 都设置 currentVote(2, MIN_VALUE)；但 winnerCount 仍未过半数，依然进行。

c) 节点 3 启动，设置自身为 currentVote(3, MIN_VALUE)，3 个节点相互交换选举结果，节点 3 的 id 值较大，此时节点 1、2、3 都设置 currentVote(3, MIN_VALUE)；winnerCount 已经超过半数，节点 3 设置自己为 LEADING 状态，节点 1、2 设置自己为 FOLLOWING 状态，leader 为节点 3。

d) 节点 4 启动，设置自身为 currentVote(4, MIN_VALUE)，4 个节点相互交换选举结果，虽然节点 4 的 id 值较大，但是接收到来自节点 1、2、3 超过半数的 winnerCount，故将自己设置为 FOLLOWING 状态，leader 为节点 3。

e) 节点 5 类似，略。

示例 2：有 5 个节点组成的集群，leader 是节点 3，由于某种原因导致节点 3 宕机，其余节点通过某种方式得知 leader 宕机，那么：

a) 节点 1 得知 leader 宕机，设置自身为 currentVote(1, MIN_VALUE)，发送 request(1)，其余节点尚未得知 leader 宕机，那么节点 1 获取到的 resopnse 列表均认为节点 3 为 leader，可惜节点 3 不在 heardFrom 列表中，因此 votes 列表被清空，节点 1 依然设置自身为 currentVote。

b) 节点 2 也得知 leader 宕机，进行同样的流程，得到的 votes 列表分别为节点 1->节点 1、节点 2->节点 2、节点 4->节点 3、节点 5->节点 3，由于节点 3 不在 heardFrom 列表，节点 4->节点 3、节点 5->节点 3 被清空。此时节点 2 的 id 较大，设置自身为 currentVote；节点 1 同样流程，设置结点 2 为 currentVote。

c) 节点 4 也得知 leader 宕机，进行同样的流程，结果是结点 4 的 id 较大，成为 leader，进行 LEADING 状态，节点 1 和结点 2 成为 follower，进入 FOLLOWING 状态。

d) 节点 5 同理，成为 follower，略。

示例 3：有 5 个节点组成的集群，leader 是节点 3，节点 1、2、4 宕机，此时 leader 发现生还的 follower 不足半数，从 LEADING 退化为 LOOKING 状态，节点 5 通过某种方式得知 leader 不存在后，也从 FOLLOWING 退化为 LOOKING 状态。

- AuthFastLeaderElection：略。
- AuthFastLeaderElection：略。
- FastLeaderElection：FastLeaderElection 与 LeaderElection 有本质的不同。LeaderElection 可以看做是向其余 PARTICIPATION 节点广播，并通过 ack 收集信息，然后进行判断处理的方式，如果不能选举成功则循环继续；而 FastLeaderElection 仅向其余 PARTICIPATION 节点广播，不需要 ack，其信息来源是其余节点的广播信息，而广播也并非循环继续的，只有两种情况下触发：一开始的时候，以及收到其他节点广播。FastLeaderElection 自身会启动两个线程负责 request 的发送队列和 response 接收队列的处理，分别为 WorkerSender 和 WorkerReveicer，而 sendqueue 和 recvqueue 则是对应的队列。

a) logicalclock 表示逻辑时钟，也就是本轮选举的 epoch，每次执行 lookForLeader()时加 1；同时设置自身(proposedLeader, proposedZxid)；然后发送 notification 给其余 PARTICIPATION 节点。

b) 从 recvqueue 中拉取 notification，如果 notification 为 null，表示未收到任何结果，需要重新发送 notification 给所有 PARTICIPATION 节点；如果 notification 不为 null，开始分情况处理。注意，recvset 存储收到的 Vote 结果。

c) 如果 notification.state 为 OBSERVING：不做任何处理。

d) 如果 notification.state 为 LOOKING：如果 notification.epoch > logicalclock，说明自身节点的选举轮次不是最新的，需要更新 logicalclock，清除 recvset 列表，根据(notification.leader, notification.zxid)判断是否更新(proposedLeader, proposedZxid)，并重新发送 notification 给其余 PARTICIPATION 节点；如果 notification.epoch < logicalclock，抛弃该 notification 不做任何处理；如果 notification.epoch = logicalclock，当(notification.leader, notification.zxid)比自身(proposedLeader, proposedZxid)更大时，更新(proposedLeader, proposedZxid)，然后重新发送 notification 给其余 PARTICIPATION 节点。注意：observer 节点也可能发送 notification，但是由于(notification.leader, notification.zxid) = (MIN_VALUE, MIN_VALUE)，因此不会被当做(proposedLeader, proposedZxid)使用，之所以让 observer 节点也发送 notification，可能是为了让触发选举流程更加敏感（否则只有 participation 节点才能触发）。上述处理完成后，如果发现收到的 notification 来源不是 observer 节点，则保存到 recvset 列表中，当 recvset 保存了所有 PARTICIPATION 节点的 notification 后，开始选举，如果选举成功则定位自身节点是 LEADING、FOLLOWING 还是 OBSERVERING 状态；如果选举失败，则进入下一次循环。

e) 如果 notification.state 为 default：

org.apache.zookeeper.server.quorum.Learner

Follower 和 Observer 的父类。

org.apache.zookeeper.server.quorum.Follower

Follower 的控制逻辑所在，主要是 followLeader()方法：

1. findLeader()方法返回 leader 地址，然后 connectToLeader()连接到 leader，有重试机制；生成的 leaderIs、leaderOs 就是对应的 Archive。

2. registerLeader()方法向 leader 注册，需要传入 pktType 和 sentLastZxid 参数，前者是 packet 类型，后者是该 follower 最新的 zxid；方法返回 leader 反馈的 newLeaderZxid，同时 leader 开始向 follower 发送需要同步的信息，同步哪些信息是根据 sentLastZxid 和 newLeaderZxid 来确定的。

3. 进行 syncWithLeader()过程，与 leader 数据同步。该过程会循环收到不同类型的 packet，只有当收到 Leader.UPTODATE 类型的 packet 时，才表示同步结束。

4. 循环执行 processPacket()，直到服务停止为止。processPacket()主要是根据 leader 发送过来的 packet 类型进行不同处理。

a) Leader.PING：执行 ping()。

b) Leader.SYNC：执行 FollowerZookeeperServer 的 sync()。

c) Leader.REVALIDATE：执行 revalidate()。

d) Leader.UPTODATE：不可能，报错。

e) Leader.COMMIT：执行 FollowerZookeeperServer 的 commit()。

f) Leader.PROPOSAL：执行 FollowerZookeeperServer 的 logRequest()。

org.apache.zookeeper.server.quorum.Observer

Observer 的逻辑所在，主要是 observeLeader()方法：

1. 与 Follower 同。
2. 与 Follower 同，不过传入的 pktType 不同。
3. 与 Follower 同。
4. processPacket()方法的处理逻辑与 Follower 不同，主要是：Leader.PROPOSAL、Leader.COMMIT 被忽略了。

org.apache.zookeeper.server.quorum.Leader

略，待补充。