Apache ZooKeeper 是一个高可靠的分布式协调中间件。它是 Google Chubby 的一个开源

实现，那么它主要是解决什么问题的呢？那就得先了解 Google Chubby

Google Chubby 是谷歌的一个用来解决分布式一致性问题的组件，同时，也是粗粒度的分布

式锁服务

分布式一致性（数据同步，leader选举之类的）

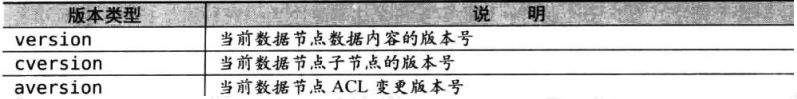
分布式锁（leader选举）

节点类型：持久化节点，持久化有序节点（为子节点维持顺序），临时节点，临时有序节点

Zookeeper中的节点除了保存内容还存在数据节点状态信息



zookeeper 为数据节点引入了版本的概念，每个数据节点都有三 类版本信息，对数据节点任何更新操作都会引起版本号的变化



Watcher机制

zookeeper 提供了分布式数据的发布/订阅功能，zookeeper允许客户端向服务端注册一个 watcher 监听，当服务端的一些指定事件触发了 watcher，那么服务端就会向客户端发送一个事件监听。Watcher通知是一次性的。常用客户端curator（优先）或者zkclient。

Watcher 监听机制是 Zookeeper 中非常重要的特性，我们基于 zookeeper 上创建的节点，可以对这些节点绑定监听事件，比如可以监听节点数据变更、节点删除、子节点状态变更等

事件，通过这个事件机制，可以基于 zookeeper 实现分布式锁、集群管理等功能 。

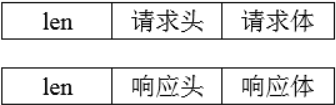
### Watcher监听机制

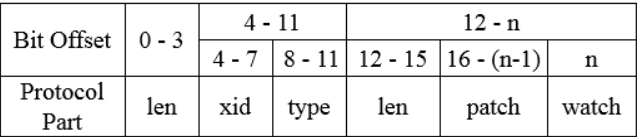
ZooKeeper 的 Watcher 机制，总的来说可以分为三个过程：客户端注册 Watcher、服务器处理 Watcher 和客户端回调 Watcher

客户端初始化：

在创建一个 ZooKeeper 客户端对象实例时，我们通过 new Watcher()向构造方法中传入一个默认的 Watcher, 这个 Watcher 将作为整个 ZooKeeper 会话期间的默认Watcher，会一直被保存在客户端 ZKWatchManager 的 defaultWatcher 中。在注册监听时会将路径和默认监听器放入一个类中以便后面添加到一个map中。

Zookeeper自己的协议：



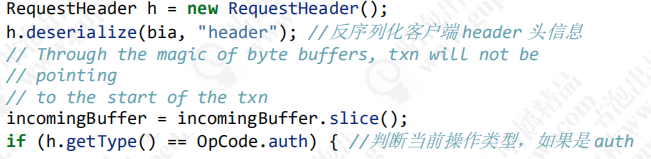


集群模式下服务器端的处理流程：

1.NIOServerCnxn 类，用来处理客户端发送过来的请求

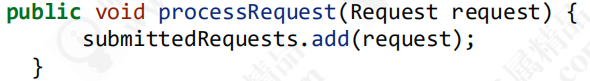
ZookeeperServer-zks.processPacket(this, bb);

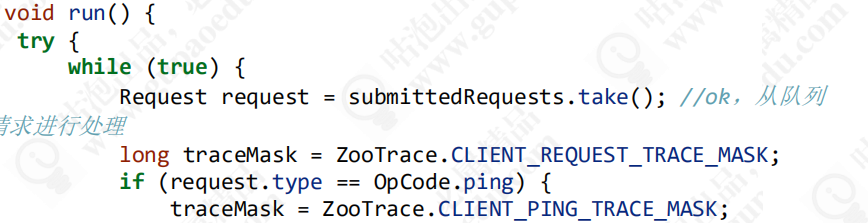
processPacket(ServerCnxn cnxn, ByteBuffer incomingBuffer)

1. 在处理过程中首先反序列化header头部信息，判断当前操作类型，根据具体的操作类型进行相应的处理
2. 

3.进行链式调用处理请求：

以链（processor）的方式调用，各种processor实现了线程，每个processor内部都存在下一个processor的属性，从而形成链式调用。Processor调用processRequest()方法时会将数据放入该processor的阻塞队列中，然后由于processor类实现了线程类，在其run方法中会取出数据，并调用下一个链的processRequest()方法。每个链中的run()方法会进行一些处理。





1597901884(1)

以最后一个链的处理FinalRequestProcessor. processRequest 为例：



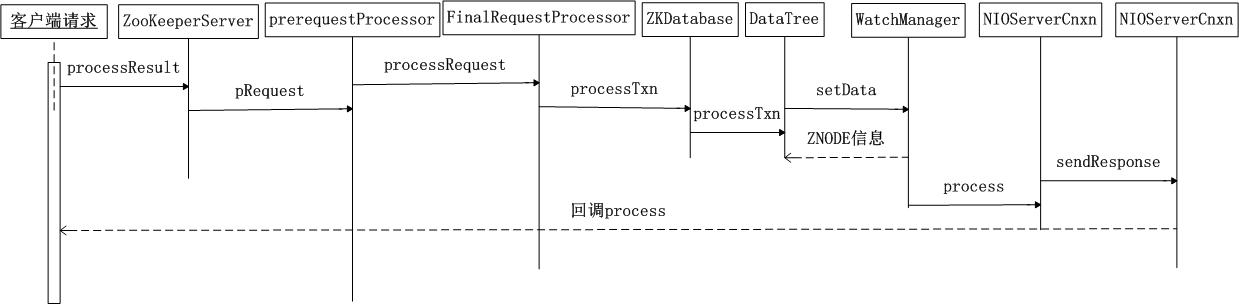
在上述代码中判断了existsRequest.getWatch时候存在，存在则传入cnxn.

服务端处理完成以后，会通过 NIOServerCnxn.sendResponse 发送返回的响应信息

SendThread.readResponse,接收服务端的信息进行读取，sendthread线程在客户端初始化的时候已经创建了。这个方法里面主要的流程如下，首先读取 header，如果其 xid == -2，表明是一个 ping 的 response，return 。如果 xid 是 -4 ，表明是一个 AuthPacket 的 response return 。如果 xid 是 -1，表明是一个 notification,此时要继续读取并构造一个 enent，通过

EventThread.queueEvent 发送，return。***其它情况下：从 pendingQueue 拿出一个 Packet（这里会校验拿出的packet中的xid与返回的xid是否相同），校验后更新 packet 信息。即将packet中的信息更新为从服务端接受到的信息。这个packet是初始发送时加入到阻塞队列中并等待回应的packet，在收到相应后就在watcherManager中放入路径和监听器。将数据包放入等待事件通知队列中。***

***服务器端：当服务端发生变更时，由于服务器端的监听器与一个连接绑定，所以直接发送一个数据包， WatchManager 负责 Watcher 事件的触发，它是一个统称，在服务端 DataTree 会托管两个 WatchManager，分别是 dataWatches 和 childWatches，分别对应数据变更 Watcher 和子节点变更 Watcher***

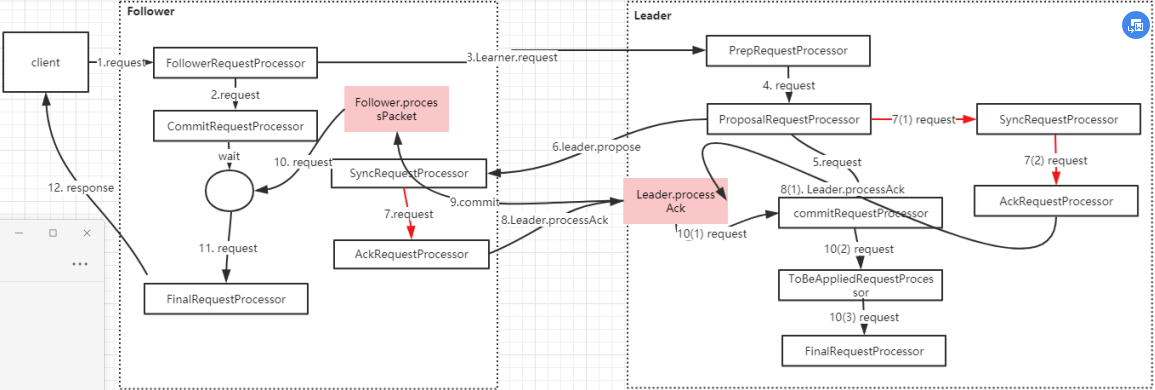


***客户端，判断由SendThread.readResponse 收到数据包后，交给eventThread.queueEvent()进行处理。这里面有具体的处理逻辑（注意，凡是涉及到线程的，都要从run()方法找到处理逻辑）***

process 方法是 Watcher 接口中的一个回调方法，当 ZooKeeper 向客户端发送一个 Watcher 事件通知时，客户端就会对相应的 process 方法进行回调，从而实现对事件的处理。

process 方法包含 WatcherEvent 类型的参数，WatchedEvent 包含了每一个事件的三个基本属性：通知状态（KeeperState）、事件类型（EventType）和节点路径（Path），ZooKeeper 使用 WatchedEvent 对象来封装服务端事件并传递给 Watcher，从而方便回调方法 process 对服务端事件进行处理

***参考网址：<https://developer.ibm.com/zh/articles/os-cn-apache-zookeeper-watcher/>***



### Zookeeper的分布式锁和选举:

数据一致性：

zookeeper 基于 zxid 以及阻塞队列的方式来实现请求的顺序一致性。如果一个 client 连接到一个最新的 follower 上，那么它 read 读取到了最新的数据，然后 client 由于网络原因重新连接到 zookeeper 节点，而这个时候连接到一个还没有完成数据同步的 follower 节点，那么这一次读到的数据不久是旧的数据吗？实际上 zookeeper 处理了这种情况，client 会记录自己已经读取到的最大的 zxid，如果 client 重连到 server 发现 client 的 zxid 比自己大。连接会失败。

#### Leader选举：

Zab协议包含两种模式，原子广播和崩溃恢复当整个集群在启动时，或者当leader节点出现网络中断、崩溃等情况时，ZAB协议就会进入恢复模式并选举产生新的Leader，当leader服务器选举出来后，并且集群中有过半的机器和该leader节点完成数据同步后（同步指的是数据同步，用来保证集群中过半的机器能够和leader服务器的数据状态保持一致），ZAB协议就会退出恢复模式。当集群中已经有过半的Follower节点完成了和Leader状态同步以后，那么整个集群就进入了消息广播模式。这个时候，在Leader节点正常工作时，启动一台新的服务器加入到集群，那这个服务器会直接进入数据恢复模式，和leader节点进行数据同步。同步完成后即可正常对外提供非事务请求的处理。

消息广播：如果大家了解分布式事务的2pc和3pc协议的话（不了解也没关系，我们后面会讲），消息广播的过程实际上是一个简化版本的二阶段提交过程

1. leader接收到消息请求后，将消息赋予一个全局唯一的64位自增id，叫：zxid，通过zxid的大小比较既可以实现因果有序这个特征
2. leader为每个follower准备了一个FIFO队列（通过TCP协议来实现，以实现了全局有序这一个特点）将带有zxid的消息作为一个提案（proposal）分发给所有的follower，
3. 当follower接收到proposal，先把proposal写到磁盘写入成功以后再向leader回复一个ack

4.当leader接收到合法数量（超过半数节点）的ACK后，leader就会向这些follower发送commit命令，同时会在本地执行该消息

5.当follower收到消息的commit命令以后，会提交该消息。

选举的关键信息：服务器id, 事务id, epoch(逻辑时钟或者投票的次数)：

服务器启动时的 leader 选举 ：

每个节点启动的时候状态都是 LOOKING，处于观望状态，接下来就开始进

行选主流程：

若进行 Leader 选举，则至少需要两台机器，这里选取 3 台机器组成的服务器集群为例。在集群初始化阶段，当有一台服务器 Server1 启动时，其单独无法进行和完成 Leader 选举，当第二台服务器 Server2 启动时，此时两台机器可以相互通信，每台机器都试图找到 Leader，于是进入 Leader选举过程。选举过程如下：

(1) 每个 Server 发出一个投票。由于是初始情况，Server1 和 Server2 都会将自己作为 Leader 服务器来进行投票，每次投票会包含所推举的服务器的 myid 和 ZXID、epoch，使用(myid, ZXID,epoch)来表示，此时 Server1 的投票为(1, 0)，Server2 的投票为(2, 0)，然后各自将这个投票发给集群中其他机器。

(2) 接受来自各个服务器的投票。集群的每个服务器收到投票后，首先判断该投票的有效性，如检查是否是本轮投票（epoch）、是否来自LOOKING 状态的服务器。

(3) 处理投票。针对每一个投票，服务器都需要将别人的投票和自己的投

票进行 PK，PK 规则如下 i. 优先比较 epochii. 其次检查 ZXID。ZXID 比较大的服务器优先作为 Leader iii. 如果 ZXID 相同，那么就比较 myid。myid 较大的服务器作为Leader 服务器。 对于 Server1 而言，它的投票是(1, 0)，接收 Server2 的投票为(2, 0)，首先会比较两者的 ZXID，均为 0，再比较 myid，此时 Server2 的myid 最大，于是更新自己的投票为(2, 0)，然后重新投票，对于Server2 而言，其无须更新自己的投票，只是再次向集群中所有机器发出上一次投票信息即可。

(4) 统计投票。每次投票后，服务器都会统计投票信息，判断是否已经有过半机器接受到相同的投票信息，对于 Server1、Server2 而言，都统计出集群中已经有两台机器接受了(2, 0)的投票信息，此时便认为已经选出了 Leader。

(5) 改变服务器状态。一旦确定了 Leader，每个服务器就会更新自己的

状态，如果是 Follower，那么就变更为 FOLLOWING，如果是 Leader(就是确定自己已经超过半数投票了)， 就变更为 LEADING。

运行过程中的 leader 选举 ：

当集群中的 leader 服务器出现宕机或者不可用的情况时，那么整个集群

将无法对外提供服务，而是进入新一轮的 Leader 选举，服务器运行期间

的 Leader 选举和启动时期的 Leader 选举基本过程是一致的。

(1) 变更状态。Leader 挂后，余下的非 Observer 服务器都会将自己的服务器状态变更为 LOOKING，然后开始进入 Leader 选举过程。

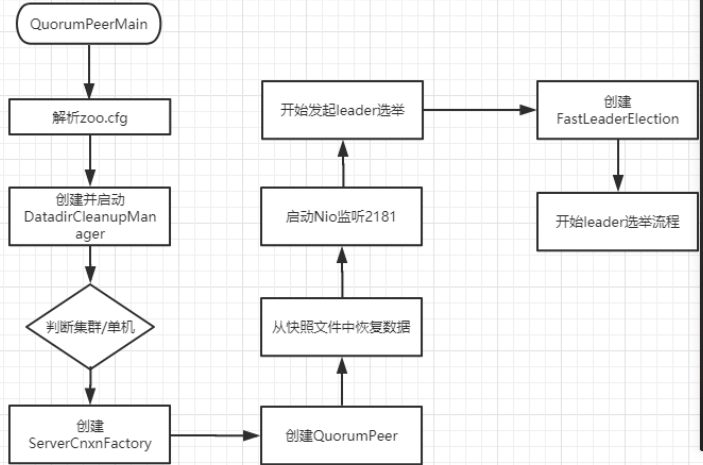
(2) 每个 Server 会发出一个投票。在运行期间，每个服务器上的 ZXID 可能不同，此时假定 Server1 的 ZXID 为 123，Server3 的 ZXID 为 122；在第一轮投票中，Server1 和 Server3 都会投自己，产生投票(1, 123)，(3, 122)，然后各自将投票发送给集群中所有机器。接收来自各个服务器的投票。与启动时过程相同。

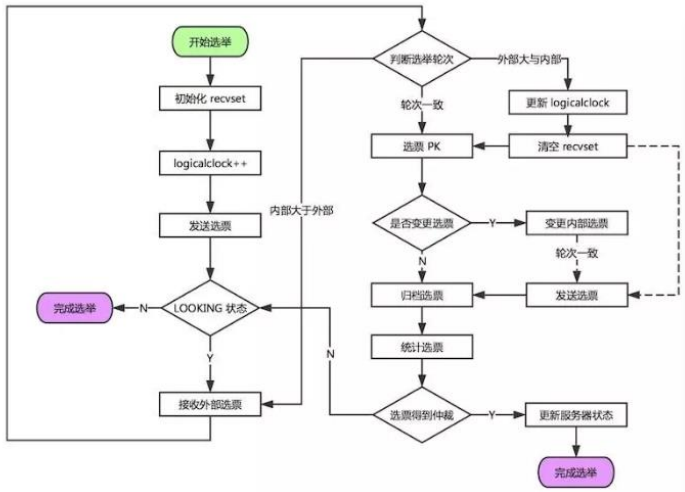
(3) 处理投票。与启动时过程相同，

(4) 统计投票。与启动时过程相同。

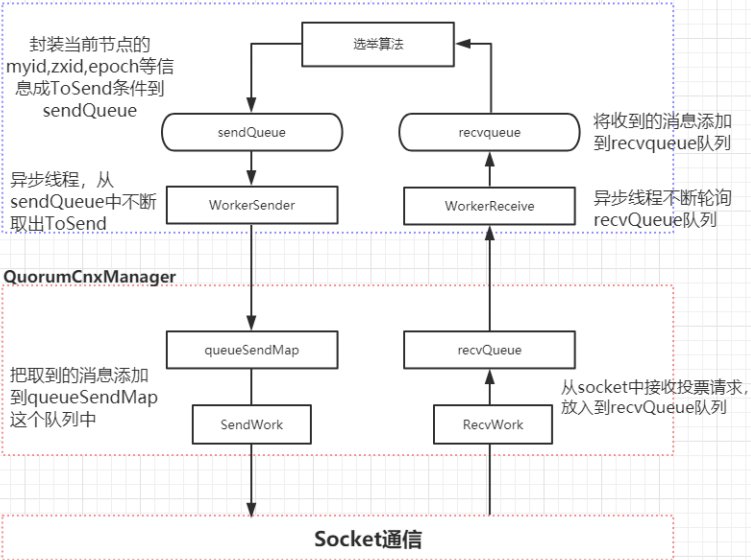
(5) 改变服务器的状态。与启动时过程相同

### 选举流程:





通信流程图：



#### leader 选举完成之后的处理逻辑 ：

通过 lookForLeader 方法选举完成以后，会设置当前节点的 PeerState，要么为 Leading、要么就是 FOLLOWING、或者 OBSERVING到这里，只是表示当前的 leader 选出来了，但是 QuorumPeer.run 方法里面还没执行完，我们再回过头看看后续的处理过程：

**Follower服务器**会初始化一个follower对象，构建一个 FollowerZookeeperServer，表示 follower 节点的请求处理服务，follower.followLeader(); 进行数据同步。

**Leader服务器初始化一个Leader对象，**构建一个 LeaderZookeeperServer，表示 Leader 节点的请求处理服务，Leader.lead()与follower进行交互。

### 分布式锁：

惊群效应是因为多个节点监听一个节点的变化。利用临时 有序节点可以实现一个节点只监听上一个节点。

InterProcessMutex：分布式可重入排它锁

InterProcessSemaphoreMutex：分布式排它锁

InterProcessReadWriteLock：分布式读写锁

**分布式锁的逻辑比较简单：大体如下：每次lock.acquire()的时候都会先看一下当前线程是否已经获取锁，如果存在一个代表了锁的本地变量LockData，则表明获取了锁，就将当前获取锁的个数加一，。如果没有获取锁则，用节点路径和当前线程初始化一个LockData，放入本地变量，并创建临时有序节点。如果当前节点不是创建的有序节点中的第一个节点，则循环获取锁或者超时获取锁。创建的节点会监听前一个节点的变化。如果是第一个节点则直接获取锁成功。当一个节点使用完毕会释放锁，lock.release()会删除当前节点，监听当前节点的下一个节点会被通知并激活当前处于wait()状态的线程。当前线程会判断自己创建的临时节点是否在第一个，如果是则获取锁，否则监听其上一个节点。（注意，这里为什么要确定时候是顺序节点中的第一个节点。（原则上，上一个节点（即第一个节点）被删除，只有下一个节点会被激活（当前第一个节点），不需要判断是否是第一个节点，但是由于一些节点会因为某种原因被删除，这个被删除的节点的下一个节点会被唤醒，这时候它属于异常唤醒，不应该获得锁，所以需要判断是否是第一个节点）**

**可以使用zookeeper的这种特性为其他集群进行leader选举：**

**例如：集群中的每个ap往zookeeper中创建顺序节点，第一个成功节点的ap为leader. 当leader手动关闭或者宕机后，第一个节点被删除，下一个节点所代表的ap晋升为leader。**

**LeaderSelector和leaderLatch的主要差别是，LeaderSelector在leader释放领导权后可以重新参加选举。**