**主要内容：**

* 客户端
  + zookeeper客户端简介，C客户端
  + 客户端连接参数说明
  + 客户端CRUD
  + 客户端监听
* 集群
  + 集群架构说明
  + 集群配置及参数说明
  + 选举 投票机制
  + 主从复制机制

一、客户端API常规应用

zookeeper 提供了java与C两种语言的客户端。我们要学习的就是java客户端。引入最新的maven依赖：

<dependency>

            <groupId>org.apache.zookeeper</groupId>

            <artifactId>zookeeper</artifactId>

            <version>3.5.5</version>

</dependency>

**知识点：**

1. 初始连接
2. 创建、查看节点
3. 监听节点
4. 设置节点权限
5. 第三方客户端ZkClient

**初始连接：**

常规的客户端类是 org.apache.zookeeper.ZooKeeper，实例化该类之后将会自动与集群建立连接。构造参数说明如下：

|  |  |  |
| --- | --- | --- |
| 参数名称 | 类型 | 说明 |
| connectString | String | 连接串，包括ip+端口 ,集群模式下用逗号隔开  192.168.0.149:2181,192.168.0.150:2181 |
| sessionTimeout | **int** | 会话超时时间，该值不能超过服务端所设置的  *minSessionTimeout 和maxSessionTimeout* |
| Watcher | Watcher | 会话监听器，服务端事件将会触该监听 |
| sessionId | **long** | 自定义会话ID |
| sessionPasswd | **byte**[] | 会话密码 |
| canBeReadOnly | **boolean** | 该连接是否为只读的 |
| hostProvider | HostProvider | 服务端地址提供者，指示客户端如何选择某个服务来调用，默认采用StaticHostProvider实现 |

2.创建、查看节点

**创建节点**

通过org.apache.zookeeper.ZooKeeper#create()即可创建节点，其参数说明如下：

|  |  |  |
| --- | --- | --- |
| 参数名称 | 类型 | 说明 |
| Path | String |  |
| Data | byte[] |  |
| Acl | List<ACL> |  |
| createMode | CreateMode |  |
| cb | StringCallback |  |
| Ctx | Object |  |

**查看节点：**

通过org.apache.zookeeper.ZooKeeper#getData()即可创建节点，其参数说明如下：

|  |  |  |
| --- | --- | --- |
| 参数名称 | 类型 | 说明 |
| Path | String |  |
| Watch | **boolean** |  |
| Watcher | Watcher |  |
| Cb | DataCallback |  |
| Ctx | Object |  |

**查看子节点：**

通过org.apache.zookeeper.ZooKeeper#getChildren()即可获取子节点，其参数说明如下：

|  |  |  |
| --- | --- | --- |
| 参数名称 | 类型 | 说明 |
| Path | String |  |
| Watch | **boolean** |  |
| Watcher | Watcher |  |
| Cb | Children2Callback |  |
| Ctx | Object |  |

3.监听节点

在getData() 与getChildren()两个方法中可分别设置监听数据变化和子节点变化。通过设置watch为true，当前事件触发时会调用zookeeper()构建函数中Watcher.process()方法。也可以添加watcher参数来实现自定义监听。一般采用后者。

注：所有的监听都是一次性的，如果要持续监听需要触发后在添加一次监听。

4.设置节点ACL权限

ACL包括结构为scheme:id:permission（有关ACL的介绍参照第一节课关于ACL的讲解）

客户端中由org.apache.zookeeper.data.ACL 类表示，类结构如下：

1. ACL
   1. Id
      1. scheme // 对应权限模式scheme
      2. id // 对应模式中的id值
   2. perms // 对应权限位permission

关于权限位的表示方式：

每个权限位都是一个唯一数字，将其合时通过或运行生成一个全新的数字即可

@InterfaceAudience.Public

public interface Perms {

int READ = 1 << 0;

int WRITE = 1 << 1;

int CREATE = 1 << 2;

int DELETE = 1 << 3;

int ADMIN = 1 << 4;

int ALL = READ | WRITE | CREATE | DELETE | ADMIN;

}

5.第三方客户端ZkClient

zkClient 是在zookeeper客户端基础之上封装的，使用上更加友好。主要变化如下：

* 可以设置持久监听，或删除某个监听
* 可以插入JAVA对象，自动进行序列化和反序列化
* 简化了基本的增删改查操作。

二、Zookeeper集群

知识点：

1. 集群部署
2. 集群角色说明
3. 选举机制
4. 数据提交机制
5. 集群配置说明

zookeeper集群的目的是为了保证系统的性能承载更多的客户端连接设专门提供的机制。通过集群可以实现以下功能：

* 读写分离：提高承载，为更多的客户端提供连接，并保障性能。
* 主从自动切换：提高服务容错性，部分节点故障不会影响整个服务集群。

**半数以上运行机制说明：**

集群至少需要三台服务器，并且强烈建议使用奇数个服务器。因为zookeeper 通过判断大多数节点的存活来判断整个服务是否可用。比如3个节点，挂掉了2个表示整个集群挂掉，而用偶数4个，挂掉了2个也表示其并不是大部分存活，因此也会挂掉。

1. 集群部署

配置语法：

server.<节点ID>=<ip>:<数据同步端口>:<选举端口>

* **节点ID**：服务id手动指定1至125之间的数字，并写到对应服务节点的 {dataDir}/myid 文件中。
* **IP地址：**节点的远程IP地址，可以相同。但生产环境就不能这么做了，因为在同一台机器就无法达到容错的目的。所以这种称作为伪集群。
* **数据同步端口：**主从同时数据复制端口，（做伪集群时端口号不能重复）。
* **远举端口：**主从节点选举端口，（做伪集群时端口号不能重复）。

配置文件示例：

tickTime=2000

dataDir=/var/lib/zookeeper/

clientPort=2181

initLimit=5

syncLimit=2

#以下为集群配置，必须配置在所有节点的zoo.cfg文件中

server.1=zoo1:2888:3888

server.2=zoo2:2888:3888

server.3=zoo3:2888:3888

**集群配置流程：**

1. 分别创建3个data目录用于存储各节点数据

mkdir data

mkdir data/1

mkdir data/3

mkdir data/3

1. 编写myid文件

echo 1 > data/1/myid

echo 3 > data/3/myid

echo 2 > data/2/myid

3、编写配置文件

*conf/zoo1.cfg*

tickTime=2000

initLimit=10

syncLimit=5

dataDir=data/1

clientPort=2181

#集群配置

server.1=127.0.0.1:2887:3887

server.2=127.0.0.1:2888:3888

server.3=127.0.0.1:2889:3889

*conf/zoo2.cfg*

tickTime=2000

initLimit=10

syncLimit=5

dataDir=data/2

clientPort=2182

#集群配置

server.1=127.0.0.1:2887:3887

server.2=127.0.0.1:2888:3888

server.3=127.0.0.1:2889:3889

*conf/zoo3.cfg*

tickTime=2000

initLimit=10

syncLimit=5

dataDir=data/3

clientPort=2183

#集群配置

server.1=127.0.0.1:2887:3887

server.2=127.0.0.1:2888:3888

server.3=127.0.0.1:2889:3889

4.分别启动

./bin/zkServer.sh start conf/zoo1.cfg

./bin/zkServer.sh start conf/zoo2.cfg

./bin/zkServer.sh start conf/zoo3.cfg

5.分别查看状态

./bin/zkServer.sh status conf/zoo1.cfg

Mode: follower

./bin/zkServer.sh status conf/zoo2.cfg

Mode: leader

./bin/zkServer.sh status conf/zoo3.cfg

Mode: follower

**检查集群复制情况：**

1、分别连接指定节点

zkCli.sh 后加参数-server 表示连接指定IP与端口。

./bin/zkCli.sh -server 127.0.0.1:2181

./bin/zkCli.sh -server 127.0.0.1:2182

./bin/zkCli.sh -server 127.0.0.1:2183

* 任意节点中创建数据，查看其它节点已经同步成功。

注意： -server参数后同时连接多个服务节点，并用逗号隔开 127.0.0.1:2181,127.0.0.1:2182

1. 集群角色说明

zookeeper 集群中总共有三种角色，分别是leader（主节点）follower(子节点) observer（次级子节点）

|  |  |
| --- | --- |
| 角色 | 描述 |
| **Leader** | 主节点，又名领导者。用于写入数据，通过选举产生，如果宕机将会选举新的主节点。 |
| **Follower** | 子节点，又名追随者。用于实现数据的读取。同时他也是主节点的备选节点，并用拥有投票权。 |
| **Observer** | 次级子节点，又名观察者。用于读取数据，与follower区别在于没有投票权，不能选为主节点。并且在计算集群可用状态时不会将observer计算入内。 |

**observer配置：**

只要在集群配置中加上observer后缀即可，示例如下：

server.3=127.0.0.1:2889:3889:observer

3.选举机制

通过 ./bin/zkServer.sh status <zoo配置文件> 命令可以查看到节点状态

./bin/zkServer.sh status conf/zoo1.cfg

Mode: follower

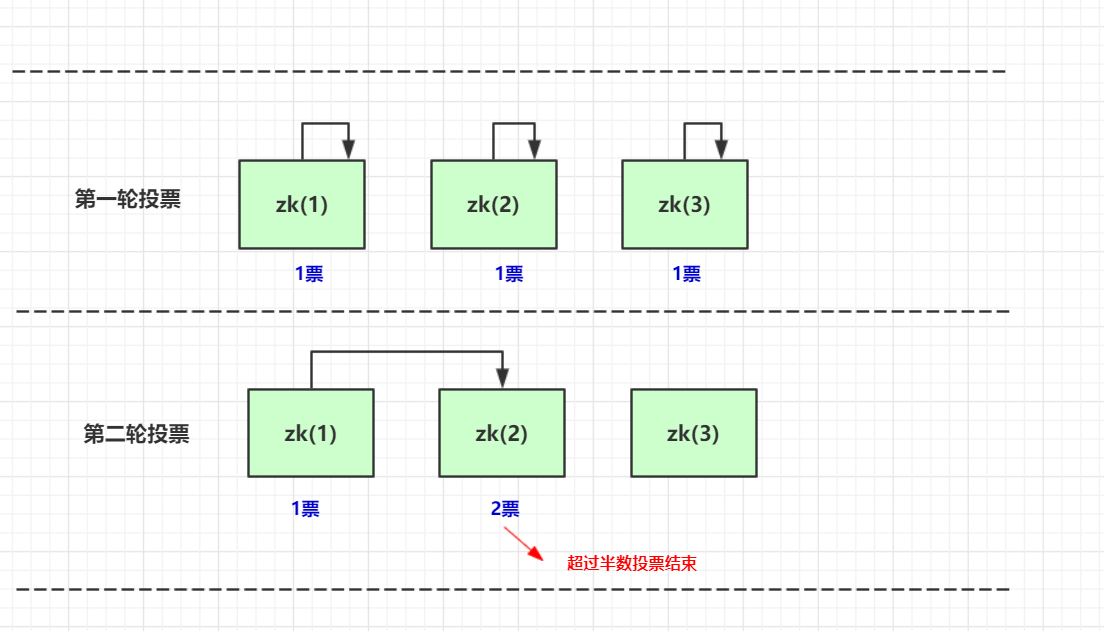
./bin/zkServer.sh status conf/zoo2.cfg

Mode: leader

./bin/zkServer.sh status conf/zoo3.cfg

Mode: follower

可以发现中间的2182 是leader状态.其选举机制如下图：



**投票机制说明：**

第一轮投票全部投给自己

第二轮投票给myid比自己大的相邻节点

如果得票超过半数，选举结束。

* 在第一轮中，按照“我最牛逼，我怕谁”的原则，每个节点都推荐它自己为集群的leader节点。
* 按照我们假设的理想条件，节点S1首先收到了S2发送来的推荐者“2”，节点S1发现“2”要比它之前推荐的“1”（也就是它自己）牛。根据谁牛推荐谁的原则，“S1”清空自己的票箱，重新选举“2”（注意，此时“S1”的新票箱中已经有两票选举“2”了，一票是它自己，另外一票是”S2”，并且所有节点都是Looking状态）
* 同样的事情发生在“S2”身上：”S2”收到了”S3”发过来的推荐信息，发现“3”这个被推举者比之前自己推举的“2”要牛，于是也清空自己的票箱，发起一轮新的投票，此时“S2”选举“3”。依次类推”S3”、”S4”。
* 这里要注意S5这个节点，在第一轮接受到了来源于“S1”——“S4”的推举者（一定注意，每一次接受信息，都会广播一次“我坚持推举的人”），发现“还是推荐的5最牛”，于是“我继续推举S5吧”。
* **以上这个过程在整个理想的网络环境上一直持续。**到了第四轮，“S1”收到了“S2”发送来的推举者“5”，发现“5”要比当前“S1”推荐的“4”要牛。**所以“S1”清空了自己的票箱，重新推举“5”（发送给其他所有节点）**。
* 关键的第五轮来了，我们再重复一下，经过之前的选举，现在“S2”——“S5”都已经推举“5”为Leader了，而且都处于第四轮。这时他们收到了”S1”发来的新的“第五轮”投票，于是都和之前一样，做相同的一件事：清空自己的票箱，重新向其他所有节点广播自己的第五轮投票“5”。
* **于是，节点X，收到了大于N / 2 +１的选举“5”的投票，且都是第五轮投票。这样每个节点就都知道了自己的角色，**选举结束。**所有将成为Follower状态的节点，向将要成为Leader的节点发起最后一次“工作是否正常”的询问。得到肯定的ack后，整个集群的工作状态就确认了**。

**选举触发：**

当集群中的服务器出现已下两种情况时会进行Leader的选举

1. 服务节点初始化启动
2. 半数以上的节点无法和Leader建立连接

当节点初始起动时会在集群中寻找Leader节点，如果找到则与Leader建立连接，其自身状态变化**follower**或**observer。**如果没有找到Leader，当前节点状态将变化LOOKING，进入选举流程。

在集群运行其间如果有follower或observer节点宕机只要不超过半数并不会影响整个集群服务的正常运行。但如果leader宕机，将暂停对外服务，所有follower将进入LOOKING 状态，进入选举流程。

Leader选举是保证分布式数据一致性的关键所在。当Zookeeper集群中的一台服务器出现以下两种情况之一时，需要进入Leader选举。

　　(1) 服务器初始化启动;

　　(2) 服务器运行期间无法和Leader保持连接。

　　下面就两种情况进行分析讲解。

　　1. 服务器启动时期的Leader选举

　　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，使用(myid, ZXID)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、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

　　　　· 优先检查ZXID。ZXID比较大的服务器优先作为Leader。

　　　　· 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，就变更为LEADING。

　　2. 服务器运行时期的Leader选举

　　在Zookeeper运行期间，Leader与非Leader服务器各司其职，即便当有非Leader服务器宕机或新加入，此时也不会影响Leader，但是一旦Leader服务器挂了，那么整个集群将暂停对外服务，进入新一轮Leader选举，其过程和启动时期的Leader选举过程基本一致。假设正在运行的有Server1、Server2、Server3三台服务器，当前Leader是Server2，若某一时刻Leader挂了，此时便开始Leader选举。选举过程如下

　　(1) 变更状态。Leader挂后，余下的非Observer服务器都会讲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。

　　(3) 接收来自各个服务器的投票。与启动时过程相同。

　　(4) 处理投票。与启动时过程相同，此时，Server1将会成为Leader。

　　(5) 统计投票。与启动时过程相同。

　　(6) 改变服务器的状态。与启动时过程相同。

　　2.2 Leader选举算法分析

　　在3.4.0后的Zookeeper的版本只保留了TCP版本的FastLeaderElection选举算法。当一台机器进入Leader选举时，当前集群可能会处于以下两种状态

　　　　· 集群中已经存在Leader。

　　　　· 集群中不存在Leader。

　　对于集群中已经存在Leader而言，此种情况一般都是某台机器启动得较晚，在其启动之前，集群已经在正常工作，对这种情况，该机器试图去选举Leader时，会被告知当前服务器的Leader信息，对于该机器而言，仅仅需要和Leader机器建立起连接，并进行状态同步即可。而在集群中不存在Leader情况下则会相对复杂，其步骤如下

　　(1) 第一次投票。无论哪种导致进行Leader选举，集群的所有机器都处于试图选举出一个Leader的状态，即LOOKING状态，LOOKING机器会向所有其他机器发送消息，该消息称为投票。投票中包含了SID（服务器的唯一标识）和ZXID（事务ID），(SID, ZXID)形式来标识一次投票信息。假定Zookeeper由5台机器组成，SID分别为1、2、3、4、5，ZXID分别为9、9、9、8、8，并且此时SID为2的机器是Leader机器，某一时刻，1、2所在机器出现故障，因此集群开始进行Leader选举。在第一次投票时，每台机器都会将自己作为投票对象，于是SID为3、4、5的机器投票情况分别为(3, 9)，(4, 8)， (5, 8)。

　　(2) 变更投票。每台机器发出投票后，也会收到其他机器的投票，每台机器会根据一定规则来处理收到的其他机器的投票，并以此来决定是否需要变更自己的投票，这个规则也是整个Leader选举算法的核心所在，其中术语描述如下

　　　　· vote\_sid：接收到的投票中所推举Leader服务器的SID。

　　　　· vote\_zxid：接收到的投票中所推举Leader服务器的ZXID。

　　　　· self\_sid：当前服务器自己的SID。

　　　　· self\_zxid：当前服务器自己的ZXID。

　　每次对收到的投票的处理，都是对(vote\_sid, vote\_zxid)和(self\_sid, self\_zxid)对比的过程。

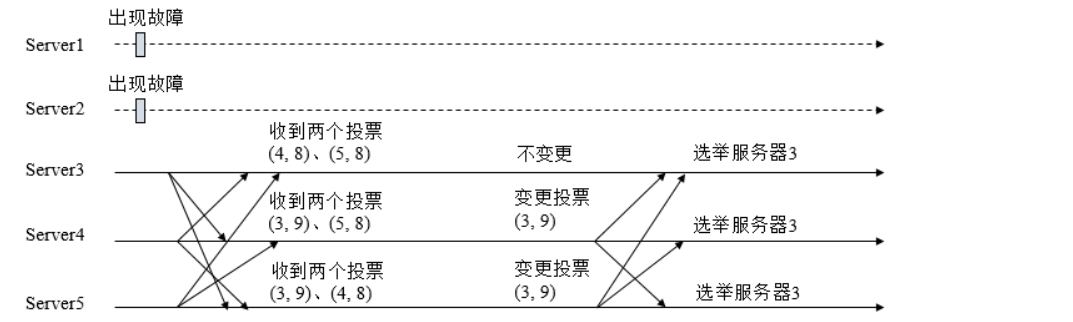
　　　　规则一：如果vote\_zxid大于self\_zxid，就认可当前收到的投票，并再次将该投票发送出去。

　　　　规则二：如果vote\_zxid小于self\_zxid，那么坚持自己的投票，不做任何变更。

　　　　规则三：如果vote\_zxid等于self\_zxid，那么就对比两者的SID，如果vote\_sid大于self\_sid，那么就认可当前收到的投票，并再次将该投票发送出去。

　　　　规则四：如果vote\_zxid等于self\_zxid，并且vote\_sid小于self\_sid，那么坚持自己的投票，不做任何变更。

结合上面规则，给出下面的集群变更过程。



(3) 确定Leader。经过第二轮投票后，集群中的每台机器都会再次接收到其他机器的投票，然后开始统计投票，如果一台机器收到了超过半数的相同投票，那么这个投票对应的SID机器即为Leader。此时Server3将成为Leader。

　　由上面规则可知，通常那台服务器上的数据越新（ZXID会越大），其成为Leader的可能性越大，也就越能够保证数据的恢复。如果ZXID相同，则SID越大机会越大。

　　2.3 Leader选举实现细节

　　1. 服务器状态

　　服务器具有四种状态，分别是LOOKING、FOLLOWING、LEADING、OBSERVING。

　　LOOKING：寻找Leader状态。当服务器处于该状态时，它会认为当前集群中没有Leader，因此需要进入Leader选举状态。

　　FOLLOWING：跟随者状态。表明当前服务器角色是Follower。

　　LEADING：领导者状态。表明当前服务器角色是Leader。

　　OBSERVING：观察者状态。表明当前服务器角色是Observer。

　　2. 投票数据结构

　　每个投票中包含了两个最基本的信息，所推举服务器的SID和ZXID，投票（Vote）在Zookeeper中包含字段如下

　　id：被推举的Leader的SID。

　　zxid：被推举的Leader事务ID。

　　electionEpoch：逻辑时钟，用来判断多个投票是否在同一轮选举周期中，该值在服务端是一个自增序列，每次进入新一轮的投票后，都会对该值进行加1操作。

　　peerEpoch：被推举的Leader的epoch。

　　state：当前服务器的状态。

　　3. QuorumCnxManager：网络I/O

　　每台服务器在启动的过程中，会启动一个QuorumPeerManager，负责各台服务器之间的底层Leader选举过程中的网络通信。

　　(1) 消息队列。QuorumCnxManager内部维护了一系列的队列，用来保存接收到的、待发送的消息以及消息的发送器，除接收队列以外，其他队列都按照SID分组形成队列集合，如一个集群中除了自身还有3台机器，那么就会为这3台机器分别创建一个发送队列，互不干扰。

　　　　· recvQueue：消息接收队列，用于存放那些从其他服务器接收到的消息。

　　　　· queueSendMap：消息发送队列，用于保存那些待发送的消息，按照SID进行分组。

　　　　· senderWorkerMap：发送器集合，每个SenderWorker消息发送器，都对应一台远程Zookeeper服务器，负责消息的发送，也按照SID进行分组。

　　　　· lastMessageSent：最近发送过的消息，为每个SID保留最近发送过的一个消息。

　　(2) 建立连接。为了能够相互投票，Zookeeper集群中的所有机器都需要两两建立起网络连接。QuorumCnxManager在启动时会创建一个ServerSocket来监听Leader选举的通信端口(默认为3888)。开启监听后，Zookeeper能够不断地接收到来自其他服务器的创建连接请求，在接收到其他服务器的TCP连接请求时，会进行处理。为了避免两台机器之间重复地创建TCP连接，Zookeeper只允许SID大的服务器主动和其他机器建立连接，否则断开连接。在接收到创建连接请求后，服务器通过对比自己和远程服务器的SID值来判断是否接收连接请求，如果当前服务器发现自己的SID更大，那么会断开当前连接，然后自己主动和远程服务器建立连接。一旦连接建立，就会根据远程服务器的SID来创建相应的消息发送器SendWorker和消息接收器RecvWorker，并启动。

　　(3) 消息接收与发送。消息接收：由消息接收器RecvWorker负责，由于Zookeeper为每个远程服务器都分配一个单独的RecvWorker，因此，每个RecvWorker只需要不断地从这个TCP连接中读取消息，并将其保存到recvQueue队列中。消息发送：由于Zookeeper为每个远程服务器都分配一个单独的SendWorker，因此，每个SendWorker只需要不断地从对应的消息发送队列中获取出一个消息发送即可，同时将这个消息放入lastMessageSent中。在SendWorker中，一旦Zookeeper发现针对当前服务器的消息发送队列为空，那么此时需要从lastMessageSent中取出一个最近发送过的消息来进行再次发送，这是为了解决接收方在消息接收前或者接收到消息后服务器挂了，导致消息尚未被正确处理。同时，Zookeeper能够保证接收方在处理消息时，会对重复消息进行正确的处理。

　　4. FastLeaderElection：选举算法核心

　　· 外部投票：特指其他服务器发来的投票；

　　· 内部投票：服务器自身当前的投票；

　　· 选举轮次：Zookeeper服务器Leader选举的轮次，即logicalclock；

　　· PK：对内部投票和外部投票进行对比来确定是否需要变更内部投票。

　　(1) 选票管理

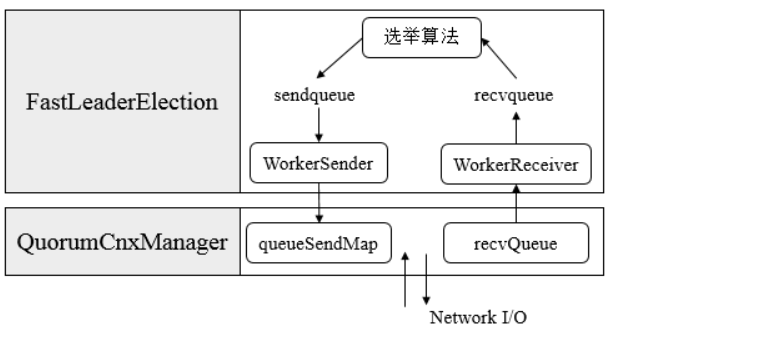
　　· sendqueue：选票发送队列，用于保存待发送的选票。

　　· recvqueue：选票接收队列，用于保存接收到的外部投票。

　　· WorkerReceiver：选票接收器。其会不断地从QuorumCnxManager中获取其他服务器发来的选举消息，并将其转换成一个选票，然后保存到recvqueue中，在选票接收过程中，如果发现该外部选票的选举轮次小于当前服务器的，那么忽略该外部投票，同时立即发送自己的内部投票。

　　· WorkerSender：选票发送器，不断地从sendqueue中获取待发送的选票，并将其传递到底层QuorumCnxManager中。

　　(2) 算法核心



上图展示了FastLeaderElection模块是如何与底层网络I/O进行交互的。Leader选举的基本流程如下

　　1. 自增选举轮次。Zookeeper规定所有有效的投票都必须在同一轮次中，在开始新一轮投票时，会首先对logicalclock进行自增操作。

　　2. 初始化选票。在开始进行新一轮投票之前，每个服务器都会初始化自身的选票，并且在初始化阶段，每台服务器都会将自己推举为Leader。

　　3. 发送初始化选票。完成选票的初始化后，服务器就会发起第一次投票。Zookeeper会将刚刚初始化好的选票放入sendqueue中，由发送器WorkerSender负责发送出去。

　　4. 接收外部投票。每台服务器会不断地从recvqueue队列中获取外部选票。如果服务器发现无法获取到任何外部投票，那么就会立即确认自己是否和集群中其他服务器保持着有效的连接，如果没有连接，则马上建立连接，如果已经建立了连接，则再次发送自己当前的内部投票。

　　5. 判断选举轮次。在发送完初始化选票之后，接着开始处理外部投票。在处理外部投票时，会根据选举轮次来进行不同的处理。

　　　　· 外部投票的选举轮次大于内部投票。若服务器自身的选举轮次落后于该外部投票对应服务器的选举轮次，那么就会立即更新自己的选举轮次(logicalclock)，并且清空所有已经收到的投票，然后使用初始化的投票来进行PK以确定是否变更内部投票。最终再将内部投票发送出去。

　　　　· 外部投票的选举轮次小于内部投票。若服务器接收的外选票的选举轮次落后于自身的选举轮次，那么Zookeeper就会直接忽略该外部投票，不做任何处理，并返回步骤4。

　　　　· 外部投票的选举轮次等于内部投票。此时可以开始进行选票PK。

　　6. 选票PK。在进行选票PK时，符合任意一个条件就需要变更投票。

　　　　· 若外部投票中推举的Leader服务器的选举轮次大于内部投票，那么需要变更投票。

　　　　· 若选举轮次一致，那么就对比两者的ZXID，若外部投票的ZXID大，那么需要变更投票。

　　　　· 若两者的ZXID一致，那么就对比两者的SID，若外部投票的SID大，那么就需要变更投票。

　　7. 变更投票。经过PK后，若确定了外部投票优于内部投票，那么就变更投票，即使用外部投票的选票信息来覆盖内部投票，变更完成后，再次将这个变更后的内部投票发送出去。

　　8. 选票归档。无论是否变更了投票，都会将刚刚收到的那份外部投票放入选票集合recvset中进行归档。recvset用于记录当前服务器在本轮次的Leader选举中收到的所有外部投票（按照服务队的SID区别，如{(1, vote1), (2, vote2)...}）。

　　9. 统计投票。完成选票归档后，就可以开始统计投票，统计投票是为了统计集群中是否已经有过半的服务器认可了当前的内部投票，如果确定已经有过半服务器认可了该投票，则终止投票。否则返回步骤4。

　　10. 更新服务器状态。若已经确定可以终止投票，那么就开始更新服务器状态，服务器首选判断当前被过半服务器认可的投票所对应的Leader服务器是否是自己，若是自己，则将自己的服务器状态更新为LEADING，若不是，则根据具体情况来确定自己是FOLLOWING或是OBSERVING。

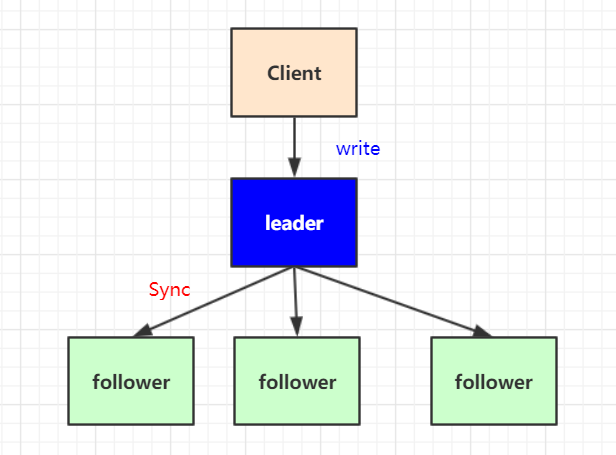
　　以上10个步骤就是FastLeaderElection的核心，其中步骤4-9会经过几轮循环，直到有Leader选举产生。

1. 数据同步机制

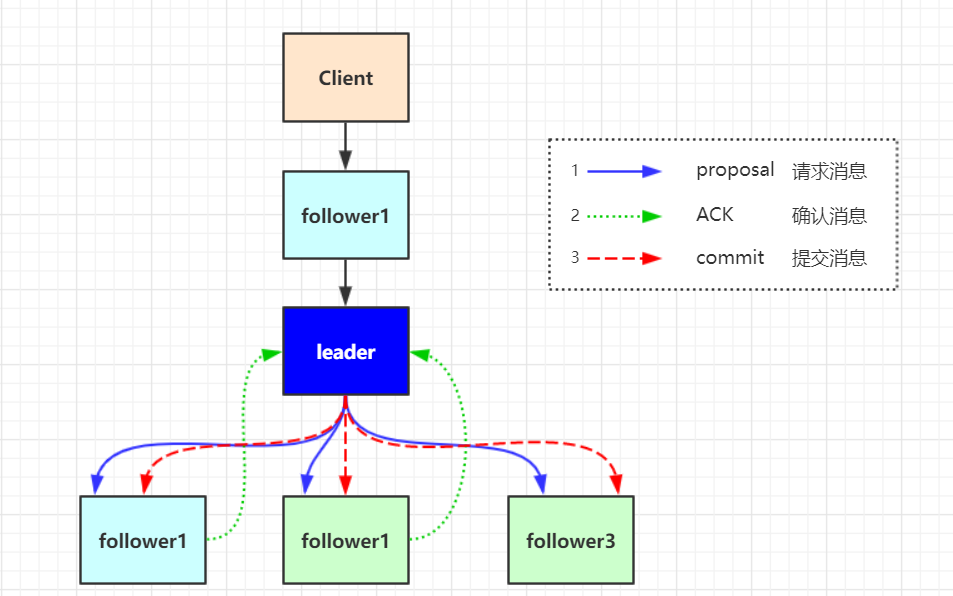
zookeeper 的数据同步是为了保证各节点中数据的一至性，同步时涉及两个流程，一个是正常的客户端数据提交，另一个是集群某个节点宕机在恢复后的数据同步。

**客户端写入请求：**

写入请求的大至流程是，收leader接收客户端写请求，并同步给各个子节点。如下图：



但实际情况要复杂的多，比如client 它并不知道哪个节点是leader 有可能写的请求会发给follower ，由follower在转发给leader进行同步处理



客户端写入流程说明：

1. client向zk中的server发送写请求，如果该server不是leader，则会将该写请求转发给leader server，leader将请求事务以proposal形式分发给follower；
2. 当follower收到收到leader的proposal时，根据接收的先后顺序处理proposal；
3. 当Leader收到follower针对某个proposal过半的ack后，则发起事务提交，重新发起一个commit的proposal
4. Follower收到commit的proposal后，记录事务提交，并把数据更新到内存数据库；
5. 当写成功后，反馈给client。

**服务节点初始化同步：**

在集群运行过程当中如果有一个follower节点宕机，由于宕机节点没过半，集群仍然能正常服务。当leader 收到新的客户端请求，此时无法同步给宕机的节点。造成数据不一至。为了解决这个问题，当节点启动时，第一件事情就是找当前的Leader，比对数据是否一至。不一至则开始同步,同步完成之后在进行对外提供服务。

如何比对Leader的数据版本呢，这里通过ZXID事物ID来确认。比Leader就需要同步。

**ZXID说明：**

ZXID是一个长度64位的数字，其中低32位是按照数字递增，任何数据的变更都会导致,低32位的数字简单加1。高32位是leader周期编号，每当选举出一个新的leader时，新的leader就从本地事物日志中取出ZXID,然后解析出高32位的周期编号，进行加1，再将低32位的全部设置为0。这样就保证了每次新选举的leader后，保证了ZXID的唯一性而且是保证递增的。

**思考题：**

如果leader 节点宕机，在恢复后它还能被选为leader吗？

5.四字运维命令

ZooKeeper响应少量命令。每个命令由四个字母组成。可通过telnet或nc向ZooKeeper发出命令。

这些命令默认是关闭的，需要配置4lw.commands.whitelist来打开，可打开部分或全部示例如下：

#打开指定命令

4lw.commands.whitelist=stat, ruok, conf, isro

#打开全部

4lw.commands.whitelist=\*

echo “4lw.commands.whitelist=\*”>>conf/zoo1.cfg //所有配置文件均需要写入

./bin/zkServer.sh restart conf/zoo1.cfg 重启zookeeper使四字运维命令生效

安装Netcat工具，已使用nc命令

#安装Netcat 工具

yum install -y nc

#查看服务器及客户端连接状态

echo stat | nc localhost 2181

**命令列表**

1. conf：3.3.0中的新增功能：打印有关服务配置的详细信息。
2. 缺点：3.3.0中的新增功能：列出了连接到该服务器的所有客户端的完整连接/会话详细信息。包括有关已接收/已发送的数据包数量，会话ID，操作等待时间，最后执行的操作等信息。
3. crst：3.3.0中的新增功能：重置所有连接的连接/会话统计信息。
4. dump：列出未完成的会话和临时节点。这仅适用于领导者。
5. envi：打印有关服务环境的详细信息
6. ruok：测试服务器是否以非错误状态运行。如果服务器正在运行，它将以imok响应。否则，它将完全不响应。响应“ imok”不一定表示服务器已加入仲裁，只是服务器进程处于活动状态并绑定到指定的客户端端口。使用“ stat”获取有关状态仲裁和客户端连接信息的详细信息。
7. srst：重置服务器统计信息。
8. srvr：3.3.0中的新功能：列出服务器的完整详细信息。
9. stat：列出服务器和连接的客户端的简要详细信息。
10. wchs：3.3.0中的新增功能：列出有关服务器监视的简要信息。
11. wchc：3.3.0中的新增功能：按会话列出有关服务器监视的详细信息。这将输出具有相关监视（路径）的会话（连接）列表。请注意，根据手表的数量，此操作可能会很昂贵（即影响服务器性能），请小心使用。
12. dirs：3.5.1中的新增功能：以字节为单位显示快照和日志文件的总大小
13. wchp：3.3.0中的新增功能：按路径列出有关服务器监视的详细信息。这将输出具有关联会话的路径（znode）列表。请注意，根据手表的数量，此操作可能会很昂贵（即影响服务器性能），请小心使用。
14. mntr：3.4.0中的新增功能：输出可用于监视集群运行状况的变量列表。