

Zookeeper-Zookeeper leader选举 - 横刀天笑 - 博客园

Zookeeper-Zookeeper leader选举

2014-11-23 17:44 by 横刀天笑, 7075 阅读, 1 评论, 收藏, 编辑

在上一篇文章中我们大致浏览了zookeeper的启动过程，并且提到在Zookeeper的启动过程中leader选举是非常重要的而且最复杂的一个环节。那么什么是leader选举呢？zookeeper为什么需要leader选举呢？zookeeper的leader选举的过程又是什么样子的？本文的目的就是解决这三个问题。

首先我们来看看什么是leader选举。其实这个很好理解，leader选举就像总统选举一样，每人一票，获得多数票的人就当选为总统了。在zookeeper集群中也是一样，每个节点都会投票，如果某个节点获得超过半数以上的节点的投票，则该节点就是leader节点了。

国家选举总统是为了选一个最高统帅，治理国家。那么zookeeper集群选举的目的又是什么呢？其实这个要清楚明白的解释还是挺复杂的。我们可以简单点想这个问题：我们有一个zookeeper集群，有好几个节点。每个节点都可以接收请求，处理请求。那么，如果这个时候分别有两个客户端向两个节点发起请求，请求的内容是修改同一个数据。比如客户端c1，请求节点n1，请求是set a = 1; 而客户端c2，请求节点n2，请求内容是set a = 2;

那么最后a是等于1还是等于2呢？这在一个分布式环境里是很难确定的。解决这个问题有很多办法，而zookeeper的办法是，我们选一个总统出来，所有的这类决策都提交给总统一个人决策，那之前的问题不就没有了么。

那我们现在的问题就是怎么来选择这个总统呢？在现实中，选择总统是需要宣讲拉选票的，那么在zookeeper的世界里这又如何处理呢？我们还是show code吧。

在QuorumPeer的startLeaderElection方法里包含leader选举的逻辑。Zookeeper默认提供了4种选举方式，默认是第4种: FastLeaderElection。

我们先假设我们这是一个崭新的集群，崭新的集群的选举和之前运行过一段时间的选举是有稍许不同的，后面会提及。

节点状态：每个集群中的节点都有一个状态 LOOKING, FOLLOWING, LEADING, OBSERVING。都属于这4种，每个节点启动的时候都是LOOKING状态，如果这个节点参与选举但最后不是leader，则状态是FOLLOWING，如果不参与选举则是OBSERVING，leader的状态是LEADING。

开始这个选举算法前，每个节点都会在zoo.cfg上指定的监听端口启动监听(server.1=127.0.0.1:20881:20882)，这里的20882就是这里用于选举的端口。

在FastLeaderElection里有一个Manager的内部类，这个类里有启动了两个线程：

WorkerReceiver, WorkerSender。为什么说选举这部分复杂呢，我觉得就是这些线程就像左右互搏一样，

非常难以理解。顾名思义，这两个线程一个是处理从别的节点接收消息的，一个是向外发送消息的。对于外面的逻辑接收和发送的逻辑都是异步的。

这里配置好了，**QuorumPeer**的**run**方法就开始执行了，这里实现的是一个简单的状态机。因为现在是**LOOKING**状态，所以进入**LOOKING**的分支，调用选举算法开始选举了：

```
setCurrentVote(makeLEStrategy().lookForLeader());
```

而在**lookForLeader**里主要是干什么呢？首先我们会更新一下一个叫逻辑时钟的东西，这也是在分布式算法里很重要的一个概念，但是在这里先不介绍，可以参考后面的论文。然后决定我要投票给谁。不过**zookeeper**这里的选举真直白，每个节点都选自己(汗),选我，选我，选我..... 然后向其他节点广播这个选举信息。这里实际上并没有真正的发送出去，只是将选举信息放到由**WorkerSender**管理的一个队列里。




```
synchronized(this) {
    //逻辑时钟
    logicalclock++;
    //getInitLastLoggedZxid(), getPeerEpoch()这里先不关心是什么，后面会讨论
    updateProposal(getInitId(), getInitLastLoggedZxid(), getPeerEpoch());
}

//getInitId() 即是获取选谁，id就是myid里指定的那个数字，所以说一定要唯一
private long getInitId(){
    if(self.getQuorumVerifier().getVotingMembers().containsKey(self.getId()))
        return self.getId();
    else return Long.MIN_VALUE;
}

//发送选举信息，异步发送
sendNotifications();
```



现在我们去看看怎么把投票信息投递出去。这个逻辑在**WorkerSender**里，**WorkerSender**从**sendqueue**里取出投票，然后交给**QuorumCnxManager**发送。因为前面发送投票信息的时候是向集群所有节点发送，所以当然也包括自己这个节点，所以**QuorumCnxManager**的发送逻辑里会判断，如果这个要发送的投票信息是发送给自己的，则不发送了，直接进入接收队列。



```
public void toSend(Long sid, ByteBuffer b) {
    if (self.getId() == sid) {
        b.position(0);
        addToRecvQueue(new Message(b.duplicate(), sid));
    } else {
```

```

        //发送给别的节点，判断之前是不是发送过
        if (!queueSendMap.containsKey(sid)) {
            //这个SEND_CAPACITY的大小是1，所以如果之前已经有一个还在等待发送，则会把之前
            的一个删除掉，发送新的

            ArrayBlockingQueue<ByteBuffer> bq = new
ArrayBlockingQueue<ByteBuffer>(SEND_CAPACITY);
            queueSendMap.put(sid, bq);
            addToSendQueue(bq, b);

        } else {
            ArrayBlockingQueue<ByteBuffer> bq = queueSendMap.get(sid);
            if(bq != null){
                addToSendQueue(bq, b);
            } else {
                LOG.error("No queue for server " + sid);
            }
        }
        //这里是真正的发送逻辑了
        connectOne(sid);
    }
}

```



connectOne就是真正发送了。在发送之前会先把自己的id和选举地址发送过去。然后判断要发送节点的id是不是比自己的id大，如果大则不发送了。如果要发送又是启动两个线程：**SendWorker**,**RecvWorker**(这种一个进程内许多不同种类的线程，各自干活的状态真的很难理解)。发送逻辑还算简单，就是从刚才放到那个**queueSendMap**里取出，然后发送。并且发送的时候将发送出去的东西放到一个**lastMessageSent**的map里，如果**queueSendMap**里是空的，就发送**lastMessageSent**里的东西，确保对方一定收到了。

看完了**SendWorker**的逻辑，再来看看数据接收的逻辑吧。还记得前面提到的有个**Listener**在选举端口上启动了监听么，现在这里应该接收到数据了。我们可以看到**receiveConnection**方法。在这里，如果接收到的信息里的id比自身的id小，则断开连接，并尝试发送消息给这个id对应的节点(当然，如果已经有**SendWorker**在往这个节点发送数据，则不用了)。

如果接收到的消息的id比当前的大，则会有**RecvWorker**接收数据，**RecvWorker**会将接收到的数据放到**recvQueue**里。

而**FastLeaderElection**的**WorkerReceiver**线程里会不断地从这个**recvQueue**里读取**Message**处理。在**WorkerReceiver**会处理一些协议上的事情，比如消息格式等。除此之外还会看看接收到的消息是不是来自投票成员。如果是投票成员，则会看看这个消息里的状态，如果是**LOOKING**状态并且当前的逻辑时钟比投票消息里的逻辑时钟要高，则会发个通知过去，告诉谁是**leader**。在这里，刚刚启动的崭新集群，所以逻辑时钟基本上都是相同的，所以这里还没判断出谁是**leader**。不过在这里我们注意到如果当前节点的状态是**LOOKING**的话，接收逻辑会将接收到的消息放到**FastLeaderElection**的**recvqueue**里。而在**FastLeaderElection**会从这个**recvqueue**里读取东西。

这里就是选举的主要逻辑了：**totalOrderPredicate**

```
protected boolean totalOrderPredicate(long newId, long newZxid, long newEpoch, long
curId, long curZxid, long curEpoch) {return ((newEpoch > curEpoch) ||
        ((newEpoch == curEpoch) &&
        ((newZxid > curZxid) || ((newZxid == curZxid) && (newId > curId)))));
}
```

1. 判断消息里的**epoch**是不是比当前的大，如果大则消息里**id**对应的**server**我就承认它是**leader**
2. 如果**epoch**相等则判断**zxid**，如果消息里的**zxid**比我的大我就承认它是**leader**
3. 如果前面两个都相等那就比较一下**server id**吧，如果比我的大我就承认它是**leader**。

关于前面两个东西暂时我们不去关心它，对于新启动的集群这两者都是相等的。

那这样看来**server id**的大小也是**leader**选举的一环啊（有的人生下来注定就不平凡，这都是命啊）。

最后我们来看看，很多文章所介绍的，如果超过一半的人说它是**leader**，那它就是**leader**的逻辑吧



```
private boolean termPredicate(
    HashMap<Long, Vote> votes,
    Vote vote) {

    HashSet<Long> set = new HashSet<Long>();
    //遍历已经收到的投票集合，将等于当前投票的集合取出放到set中
    for (Map.Entry<Long,Vote> entry : votes.entrySet()) {
        if
(self.getQuorumVerifier().getVotingMembers().containsKey(entry.getKey())
        && vote.equals(entry.getValue())){
            set.add(entry.getKey());
        }
    }

    //统计set，也就是投某个id的票数是否超过一半
    return self.getQuorumVerifier().containsQuorum(set);
}

public boolean containsQuorum(Set<Long> ackSet) {
    return (ackSet.size() > half);
}
```



最后一关：如果选的是自己，则将自己的状态更新为**LEADING**，否则根据**type**，要么是**FOLLOWING**，要么是**OBSERVING**。

到这里选举就结束了。

这里介绍的是一个新集群启动时候的选举过程，启动的时候就是根据zoo.cfg里的配置，向各个节点广播投票，一般都是选投自己。然后收到投票后就会进行判断。如果某个节点收到的投票数超过一半，那么它就是leader了。

了解了这个过程，我们来看看另外一个问题：

一个集群有3台机器，挂了一台后的影响是什么？挂了两台呢？

挂了一台：挂了一台后就是收不到其中一台的投票，但是有两台可以参与投票，按照上面的逻辑，它们开始都投给自己，后来按照选举的原则，两个人都投票给其中一个，那么就有一个节点获得的票等于2， $2 > (3/2)=1$ 的，超过了半数，这个时候是能选出leader的。

挂了两台：挂了两台后，怎么弄也只能获得一张票，1 不大于 $(3/2)=1$ 的，这样就无法选出一个leader了。

在前面介绍时，为了简单我假设的是这是一个崭新的刚启动的集群，这样的集群与工作一段时间后的集群有什么不同呢？不同的就是epoch和zxid这两个参数。在新启动的集群里这两个一般是相等的，而工作一段时间后这两个参数有可能有的节点落后其他节点，至于为什么，这个还要在后面的存储和处理额胡断请求的文章里介绍。

* 关于逻辑时钟，我们的分布式大牛Leslie Lamport曾写过一篇论文：Time, Clocks, and the Ordering of Events in a Distributed System


好文要顶

关注我

收藏该文







横刀天笑

关注 - 15

粉丝 - 1073

00

(请您对文章做出评价)

荣誉：[推荐博客](#)

[+加关注](#)

« 上一篇：[Zookeeper-Zookeeper启动过程](#)
» 下一篇：[Zookeeper-Zookeeper client](#)