基于 Curator 实现分布式锁

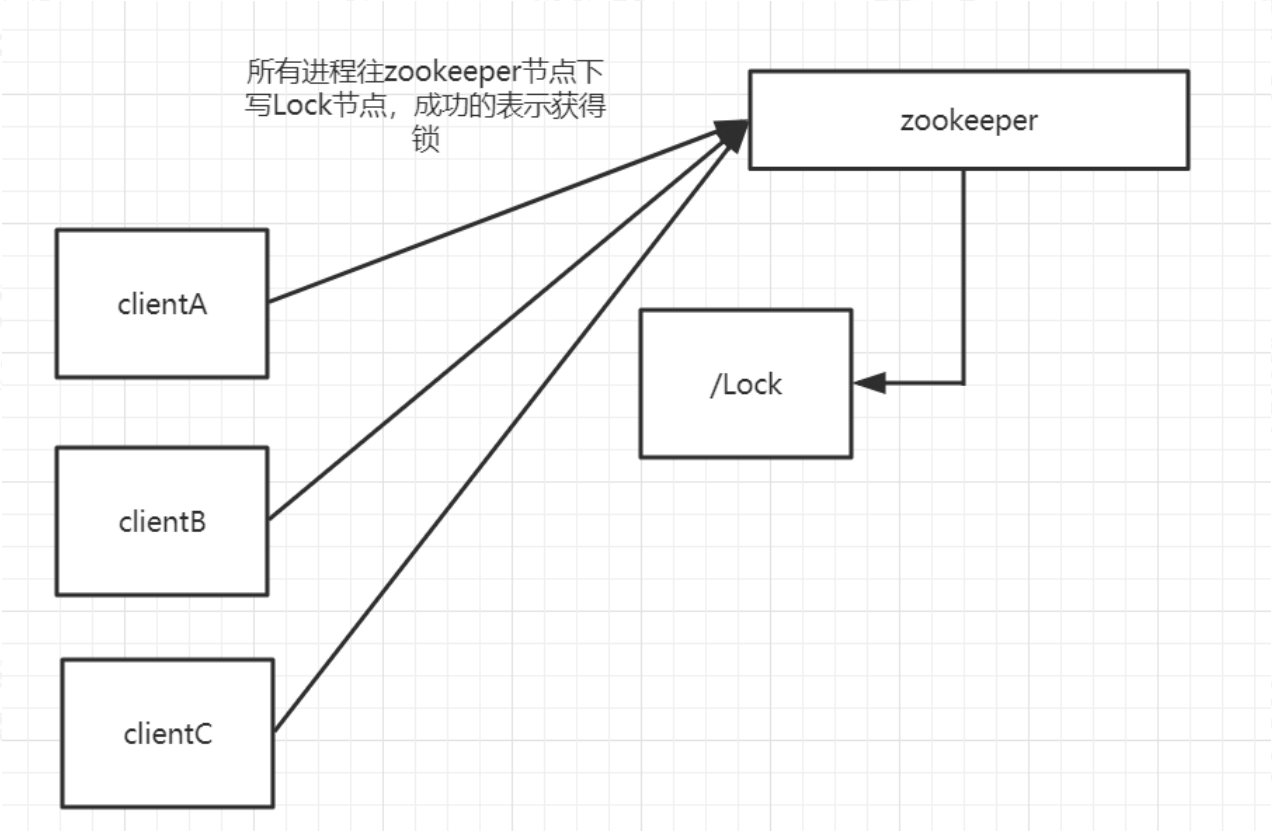
分布式锁的基本场景

如果在多线程并行情况下去访问某一个共享资源，比如说共享变量，那么势必会造成线程安全问题。那么我们可以用很多种方法来解决，比如 synchronized、 比如 Lock 之类的锁操作来解决线程安全问题，那么在分布式架构下，涉及到多个进程访问某一个共享资源的情况，比如说在电商平台中商品库存问题，在库存只有 10 个的情况下进来 100 个用户，如何能够避免超卖呢？所以这个时候我们需要一些互斥手段来防止彼此之间的干扰。
然后在分布式情况下，synchronized 或者 Lock 之类的锁只能控制单一进程的资源访问，在多进程架构下，这些 api 就没办法解决我们的问题了。怎么办呢?

用 zookeeper 来实现分布式锁

结合我们前面对 zookeeper 特性的分析和理解，我们可以利用 zookeeper 节点的特性来实现独占锁，就是同级节点的唯一性，多个进程往 zookeeper 的指定节点下创建一个相同名称的节点，只有一个能成功，另外一个是创建失败；创建失败的节点全部通过 zookeeper 的 watcher 机制来监

听 zookeeper 这个子节点的变化，一旦监听到子节点的删除事件，则再次触发所有进程去写锁；



这种实现方式很简单，但是会产生“惊群效应”，简单来说就是如果存在许多的客户端在等待获取锁，当成功获取到锁的进程释放该节点后，所有处于等待状态的客户端都会被唤醒，这个时候 zookeeper 在短时间内发送大量子节点变更事件给所有待获取锁的客户端，然后实际情况是只会有一个客户端获得锁。如果在集群规模比较大的情况下，会对 zookeeper 服务器的性能产生比较的影响。

利用有序节点来实现分布式锁

我们可以通过有序节点来实现分布式锁，每个客户端都往

指定的节点下注册一个临时有序节点，越早创建的节点，节点的顺序编号就越小，那么我们可以判断子节点中最小的节点设置为获得锁。如果自己的节点不是所有子节点中最小的，意味着还没有获得锁。这个的实现和前面单节点实现的差异性在于，每个节点只需要监听比自己小的节点，当比自己小的节点删除以后，客户端会收到 watcher 事件，此时再次判断自己的节点是不是所有子节点中最小的，如果是则获得锁，否则就不断重复这个过程，这样就不会导致羊群效应，因为每个客户端只需要监控一个节点。

curator 分布式锁的基本使用

curator 对于锁这块做了一些封装，curator 提供了 InterProcessMutex 这样一个 api。除了分布式锁之外，还提供了 leader 选举、分布式队列等常用的功能。
InterProcessMutex：分布式可重入排它锁
InterProcessSemaphoreMutex：分布式排它锁
InterProcessReadWriteLock：分布式读写锁

```
public class Demo {
```

```
    public static void
main(String[] args) {
        CuratorFramework
curatorFramework=null;

curatorFramework=CuratorFrameworkF
actory.builder().

connectString(ZkConfig.ZK_CONNECT_
STR).

sessionTimeoutMs(ZkConfig.ZK_SESSI
ON_TIMEOUT).
                retryPolicy(new
ExponentialBackoffRetry(1000,10)).
build();
        curatorFramework.start();

        final InterProcessMutex
lock=new
InterProcessMutex(curatorFramework
,"/locks");
```

```
        for(int i=0;i<10;i++){
            new Thread(()->{

System.out.println(Thread.currentT
hread().getName()+"->尝试获取锁");
                try {
                    lock.acquire();

System.out.println(Thread.currentT
hread().getName()+"->获得锁成功");
                } catch (Exception e)
{

e.printStackTrace();
                }
                try {

Thread.sleep(4000);
                    lock.release();

System.out.println(Thread.currentT
hread().getName()+"->释放锁成功");
```

```
                } catch (Exception e)
{

e.printStackTrace();
                }


            },"t"+i).start();
        }

    }
}
```

Curator 实现分布式锁的基本原理

构造函数

```
// 最常用
public InterProcessMutex(CuratorFramework client,
String path){
    // Zookeeper 利用 path 创建临时顺序节点，实现公
平锁的核心
    this(client, path, new
```

```
StandardLockInternalsDriver());
}

public InterProcessMutex(CuratorFramework client, String path, LockInternalsDriver driver){
    // maxLeases=1, 表示可以获得分布式锁的线程数量（跨 JVM）为 1，即为互斥锁
    this(client, path, LOCK_NAME, 1, driver);
}

// protected 构造函数
InterProcessMutex(CuratorFramework client, String path, String lockName, int maxLeases, LockInternalsDriver driver){
    basePath = PathUtils.validatePath(path);
    // internals 的类型为 LockInternals，InterProcessMutex 将分布式锁的申请和释放操作委托给 internals 执行
    internals = new LockInternals(client, driver, path, lockName, maxLeases);
}
```

InterProcessMutex.acquire

```
// 无限等待
public void acquire() throws Exception{
    if ( !internalLock(-1, null) ){
        throw new IOException("Lost connection while trying to acquire lock: " + basePath);
    }
}

// 限时等待
public boolean acquire(long time, TimeUnit unit) throws Exception{
    return internalLock(time, unit);
}
```

InterProcessMutex.internalLock

```
    private boolean internalLock(long time, TimeUnit unit) throws Exception{
        Thread currentThread = Thread.currentThread();
        LockData lockData = threadData.get(currentThread);
```

```
            if ( lockData != null ){
                // 实现可重入
                // 同一线程再次 acquire，首先判断当前的映射表内（threadData）是否有该线程的锁信息，如果有则原子+1，然后返回
                lockData.lockCount.incrementAndGet();
                return true;
            }

            // 映射表内没有对应的锁信息，尝试通过 LockInternals 获取锁
            String lockPath = internals.attemptLock(time, unit, getLockNodeBytes());
            if ( lockPath != null ){
                // 成功获取锁，记录信息到映射表
                LockData newLockData = new LockData(currentThread, lockPath);
                threadData.put(currentThread, newLockData);
                return true;
            }
            return false;
```

```
    }
    // 映射表
    // 记录线程与锁信息的映射关系
    private final ConcurrentMap<Thread, LockData>
threadData = Maps.newConcurrentMap();
    // 锁信息
    // Zookeeper 中一个临时顺序节点对应一个“锁”，但
让锁生效激活需要排队（公平锁），下面会继续分析
    private static class LockData{
        final Thread owningThread;
        final String lockPath;
        final AtomicInteger lockCount = new
AtomicInteger(1); // 分布式锁重入次数

        private LockData(Thread owningThread,
String lockPath){
            this.owningThread = owningThread;
            this.lockPath = lockPath;
        }
    }
```

LockInternals.attemptLock

```
// 尝试获取锁，并返回锁对应的 Zookeeper 临时顺序节点的路径
String attemptLock(long time, TimeUnit unit, byte[] lockNodeBytes) throws Exception{
    final long startMillis = System.currentTimeMillis();
    // 无限等待时，millisToWait 为 null
    final Long millisToWait = (unit != null) ? unit.toMillis(time) : null;
    // 创建 ZNode 节点时的数据内容，无关紧要，这里为 null，采用默认值（IP 地址）
    final byte[] localLockNodeBytes = (revocable.get() != null) ? new byte[0] : lockNodeBytes;
    // 当前已经重试次数，与 CuratorFramework 的重试策略有关
    int retryCount = 0;

    // 在 Zookeeper 中创建的临时顺序节点的路径，相当于一把待激活的分布式锁
    // 激活条件：同级目录子节点，名称排序最小
```

```
（排队，公平锁），后续继续分析
        String ourPath = null;
        // 是否已经持有分布式锁
        boolean hasTheLock = false;
        // 是否已经完成尝试获取分布式锁的操作
        boolean isDone = false;

        while ( !isDone ){
            isDone = true;
            try{
                // 从 InterProcessMutex 的构造函数可知实际 driver 为 StandardLockInternalsDriver 的实例
                // 在 Zookeeper 中创建临时顺序节点
                ourPath = driver.createsTheLock(client, path, localLockNodeBytes);
                // 循环等待来激活分布式锁，实现锁的公平性，后续继续分析
                hasTheLock = internalLockLoop(startMillis, millisToWait, ourPath);
            } catch ( KeeperException.NoNodeException e ) {
```

```
                    // 容错处理，不影响主逻辑的理解，可跳过
                    // 因为会话过期等原因，StandardLockInternalsDriver 因为无法找到创建的临时顺序节点而抛出 NoNodeException 异常
                    if ( client.getZookeeperClient().getRetryPolicy().allowRetry(retryCount++,
                            System.currentTimeMillis() - startMillis, RetryLoop.getDefaultRetrySleeper()) ){
                        // 满足重试策略尝试重新获取锁
                        isDone = false;
                    } else {
                        // 不满足重试策略则继续抛出 NoNodeException
                        throw e;
                    }
                }
            }
            if ( hasTheLock ){
                // 成功获得分布式锁，返回临时顺序节点的路径，上层将其封装成锁信息记录在映射表，方便锁重
```

```
入
                return ourPath;
            }
            // 获取分布式锁失败，返回 null
            return null;
        }
```

createsTheLock

```
        // From StandardLockInternalsDriver
        // 在 Zookeeper 中创建临时顺序节点
        public String createsTheLock(CuratorFramework client, String path, byte[] lockNodeBytes) throws Exception{
            String ourPath;
            // lockNodeBytes 不为 null 则作为数据节点内容，否则采用默认内容（IP 地址）
            if ( lockNodeBytes != null ){
                // 下面对 CuratorFramework 的一些细节做解释，不影响对分布式锁主逻辑的解释，可跳过
                // creatingParentContainersIfNeeded：用于创建父节点，如果不支持 CreateMode.CONTAINER
                // 那么将采用 CreateMode.PERSISTENT
```

```
                // withProtection：临时子节点会添加GUID
前缀
                ourPath                                    =
client.create().creatingParentContainersIfNeeded()
                        //
CreateMode.EPHEMERAL_SEQUENTIAL： 临时顺序节
点，Zookeeper 能保证在节点产生的顺序性
                        // 依据顺序来激活分布式锁，从而也
实现了分布式锁的公平性，后续继续分析
                        .withProtection().withMode(CreateM
ode.EPHEMERAL_SEQUENTIAL).forPath(path,
lockNodeBytes);
            } else {
                ourPath                                    =
client.create().creatingParentContainersIfNeeded()
                        .withProtection().withMode(CreateM
ode.EPHEMERAL_SEQUENTIAL).forPath(path);
            }
            return ourPath;
        }
```

LockInternals.internalLockLoop

```
    // 循环等待来激活分布式锁，实现锁的公平性
    private boolean internalLockLoop(long startMillis,
Long millisToWait, String ourPath) throws Exception {
        // 是否已经持有分布式锁
        boolean haveTheLock = false;
        // 是否需要删除子节点
        boolean doDelete = false;
        try {
            if (revocable.get() != null) {

client.getData().usingWatcher(revocableWatcher).forPa
th(ourPath);
            }

            while ((client.getState() ==
CuratorFrameworkState.STARTED) && !haveTheLock) {
                // 获取排序后的子节点列表
                List<String> children =
getSortedChildren();
                // 获取前面自己创建的临时顺序子节
```

```
点的名称
                String sequenceNodeName =
ourPath.substring(basePath.length() + 1);
                // 实现锁的公平性的核心逻辑，看下
面的分析
                PredicateResults predicateResults =
driver.getsTheLock(client,

children , sequenceNodeName , maxLeases);
                if (predicateResults.getsTheLock()) {
                    // 获得了锁，中断循环，继续返回
上层
                    haveTheLock = true;
                } else {
                    // 没有获得到锁，监听上一临时
顺序节点
                    String previousSequencePath =
basePath + "/" + predicateResults.getPathToWatch();
                    synchronized (this) {
                        try {
                            // exists()会导致导致资
源泄漏，因此 exists()可以监听不存在的 ZNode，因此采
```

用 getData()

// 上一临时顺序节点如果被删除，会唤醒当前线程继续竞争锁，正常情况下能直接获得锁，因为锁是公平的

client.getData().usingWatcher(watcher).forPath(previousSequencePath);

if (millisToWait != null) {

millisToWait -= (System.currentTimeMillis() - startMillis);

startMillis = System.currentTimeMillis();

if (millisToWait <= 0) {

doDelete = true; // 获取锁超时，标记删除之前创建的临时顺序节点

break;

}

wait(millisToWait); // 等待被唤醒，限时等待

} else {

wait(); // 等待被唤

```
醒，无限等待
                                    }
                                } catch
(KeeperException.NoNodeException e) {
                                    // 容错处理，逻辑稍微有点
绕，可跳过，不影响主逻辑的理解
                                    // client.getData()可能调用
时抛出 NoNodeException，原因可能是锁被释放或会话
过期（连接丢失）等
                                    // 这里并没有做任何处理，
因为外层是 while 循环，再次执行 driver.getsTheLock 时
会调用 validateOurIndex
                                    // 此时会抛出
NoNodeException，从而进入下面的 catch 和 finally 逻
辑，重新抛出上层尝试重试获取锁并删除临时顺序节点
                                }
                            }
                        }
                    }
                } catch (Exception e) {
                    ThreadUtils.checkInterrupted(e);
                    // 标记删除，在 finally 删除之前创建的临
```

```
时顺序节点（后台不断尝试）
                doDelete = true;
                // 重新抛出，尝试重新获取锁
                throw e;
            } finally {
                if (doDelete) {
                    deleteOurPath(ourPath);
                }
            }
            return haveTheLock;
        }
```

getTheLock

```
        // From StandardLockInternalsDriver
        public PredicateResults getsTheLock(CuratorFramework client, List<String> children, String sequenceNodeName, int maxLeases) throws Exception{
            // 之前创建的临时顺序节点在排序后的子节点列表中的索引
            int ourIndex = children.indexOf(sequenceNodeName);
```

```
            // 校验之前创建的临时顺序节点是否有效
            validateOurIndex(sequenceNodeName, ourIndex);
            // 锁公平性的核心逻辑
            // 由 InterProcessMutex 的构造函数可知，maxLeases 为 1，即只有 ourIndex 为 0 时，线程才能持有锁，或者说该线程创建的临时顺序节点激活了锁
            // Zookeeper 的临时顺序节点特性能保证跨多个 JVM 的线程并发创建节点时的顺序性，越早创建临时顺序节点成功的线程会更早地激活锁或获得锁
            boolean getsTheLock = ourIndex < maxLeases;
            // 如果已经获得了锁，则无需监听任何节点，否则需要监听上一顺序节点（ourIndex-1）
            // 因为锁是公平的，因此无需监听除了（ourIndex-1）以外的所有节点，这是为了减少羊群效应，非常巧妙的设计!!
            String pathToWatch = getsTheLock ? null : children.get(ourIndex - maxLeases);
            // 返回获取锁的结果，交由上层继续处理（添加监听等操作）
            return new PredicateResults(pathToWatch,
```

```
getsTheLock);
        }

        static void validateOurIndex(String sequenceNodeName, int ourIndex) throws KeeperException{
            if ( ourIndex < 0 ){
                // 容错处理，可跳过
                // 由于会话过期或连接丢失等原因，该线程创建的临时顺序节点被 Zookeeper 服务端删除，往外抛出 NoNodeException
                // 如果在重试策略允许范围内，则进行重新尝试获取锁，这会重新重新生成临时顺序节点
                // 佩服 Curator 的作者将边界条件考虑得如此周到！
                throw new KeeperException.NoNodeException("Sequential path not found: " + sequenceNodeName);
            }
        }
```

释放锁的逻辑

InterProcessMutex.release

```
        public void release() throws Exception{
            Thread currentThread = Thread.currentThread();
            LockData lockData = threadData.get(currentThread);
            if ( lockData == null ){
                // 无法从映射表中获取锁信息，不持有锁
                throw new IllegalMonitorStateException("You do not own the lock: " + basePath);
            }

            int newLockCount = lockData.lockCount.decrementAndGet();
            if ( newLockCount > 0 ){
                // 锁是可重入的，初始值为 1，原子-1 到 0，锁才释放
                return;
            }
```

```
            if ( newLockCount < 0 ){
                // 理论上无法执行该路径
                throw new IllegalMonitorStateException("Lock count has gone negative for lock: " + basePath);
            }
            try{
                // lockData != null && newLockCount == 0，释放锁资源
                internals.releaseLock(lockData.lockPath);
            } finally {
                // 最后从映射表中移除当前线程的锁信息
                threadData.remove(currentThread);
            }
        }
```

LockInternals.releaseLock

```
    void releaseLock(String lockPath) throws Exception{
        revocable.set(null);
        // 删除临时顺序节点，只会触发后一顺序节点去获取锁，理论上不存在竞争，只排队，非抢占，公平锁，
```

```
先到先得
            deleteOurPath(lockPath);
        }
```

```
        // Class:LockInternals
        private void deleteOurPath(String ourPath) throws
Exception{
            try{
                // 后台不断尝试删除

client.delete().guaranteed().forPath(ourPath);
            } catch ( KeeperException.NoNodeException
e ) {
                // 已经删除(可能会话过期导致)，不做处理
                // 实际使用 Curator-2.12.0 时，并不会抛
出该异常
            }
        }
```

使用 Zookeeper 实现 leader 选举

在分布式计算中，leader election 是很重要的一个功能，这个选举过程是这样子的：指派一个进程作为组织者，将任务分发给各节点。在任务开始前，哪个节点都不知道谁

是 leader 或者 coordinator。当选举算法开始执行后，每个节点最终会得到一个唯一的节点作为任务 leader。除此之外，选举还经常会发生在 leader 意外宕机的情况下，新的 leader 要被选举出来。
Curator 有两种选举 recipe（Leader Latch 和 Leader Election）

Leader Latch

参与选举的所有节点，会创建一个顺序节点，其中最小的节点会设置为 master 节点，没抢到 Leader 的节点都监听前一个节点的删除事件，在前一个节点删除后进行重新抢主，当 master 节点手动调用 close（）方法或者 master 节点挂了之后，后续的子节点会抢占 master。
其中 spark 使用的就是这种方法

LeaderSelector

LeaderSelector 和 Leader Latch 最的差别在于，leader 可以释放领导权以后，还可以继续参与竞争

LeaderSelector 案例演示

```
public class SelectorClient2 extends
LeaderSelectorListenerAdapter implements Closeable
{
```

```
    private final String name;
    private final LeaderSelector leaderSelector;

    public SelectorClient2(CuratorFramework
client, String path, String name) {
        this.name = name;
        // 利用一个给定的路径创建一个 leader
selector
        // 执行 leader 选举的所有参与者对应的路径必
须一样
        // 本例中 SelectorClient 也是一个
LeaderSelectorListener，但这不是必须的。
        leaderSelector = new
LeaderSelector(client, path, this);
        // 在大多数情况下，我们会希望一个 selector
放弃 leader 后还要重新参与 leader 选举
        leaderSelector.autoRequeue();
    }

    public void start(){
        leaderSelector.start();
```

```
    }

    @Override
    public void close() throws IOException {
        leaderSelector.close();
    }

    @Override
    public void takeLeadership(CuratorFramework
curatorFramework) throws Exception {
        System.out.println(name + " 现在是 leader
了，持续成为 leader ");
        //选举为 master，
        System.in.read();//阻塞，让当前获得 leader
权限的节点一直持有，直到该进程关闭

    }
    private static String
CONNECTION_STR="192.168.13.102:2181,192.168.13.103
:2181,192.168.13.104:2181";

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

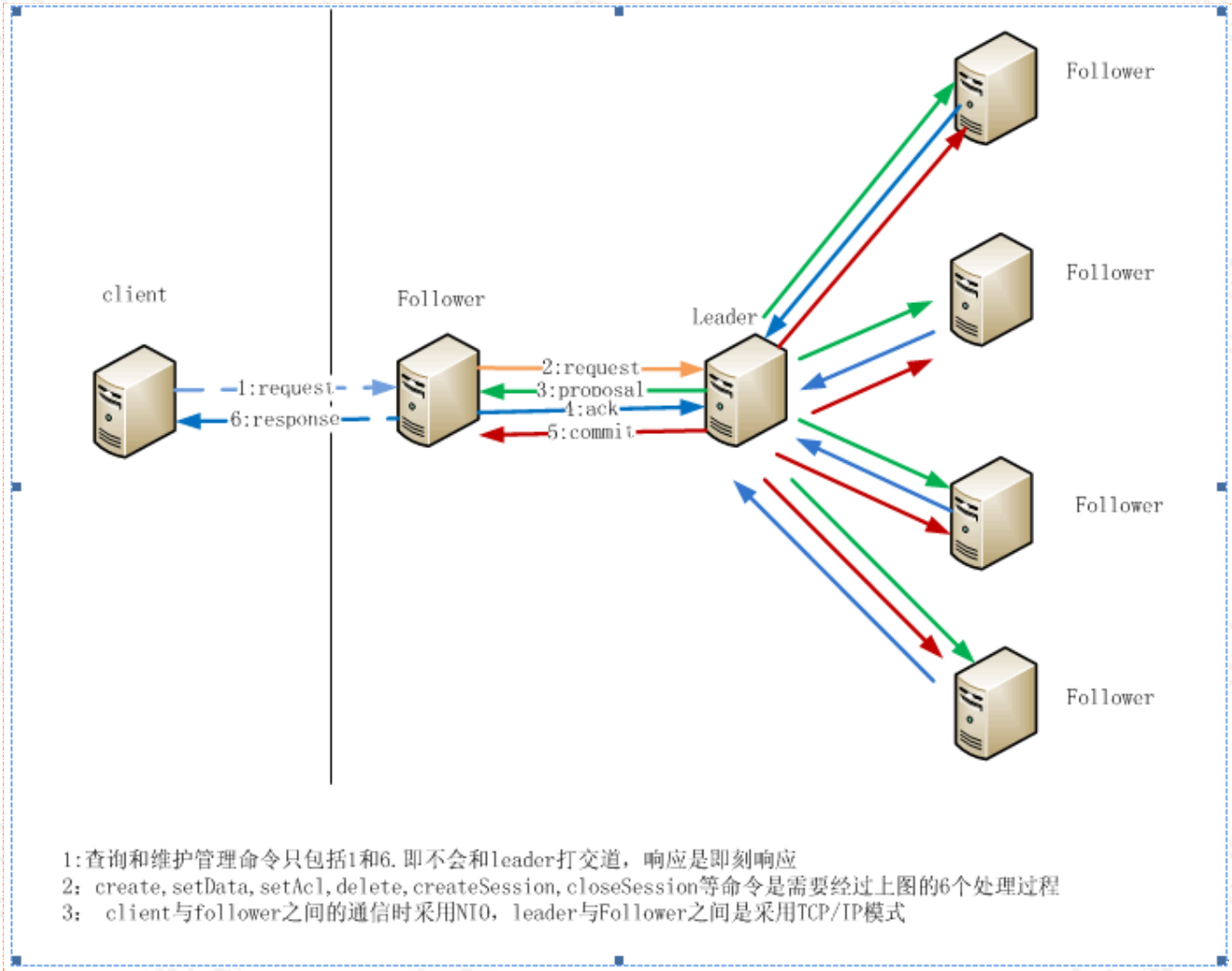
```
IOException {
        CuratorFramework curatorFramework=
CuratorFrameworkFactory.builder().

connectString(CONNECTION_STR).sessionTimeoutMs(500
0).
                retryPolicy(new
ExponentialBackoffRetry(1000,3)).build();
        curatorFramework.start();
        SelectorClient2 sc=new
SelectorClient2(curatorFramework,"/leader","Client
B");
        sc.start();
        System.in.read();
    }
```

Zookeeper 数据的同步流程

在第一节课，我们了解了 zk 的集群组成， zookeeper 通过三种不同的集群角色来组成整个高性能集群的

在 zookeeper 中,客户端会随机连接到 zookeeper 集群中的一个节点，如果是读请求，就直接从当前节点中读取数据,如果是写请求,那么请求会被转发给 leader 提交事务，

然后 leader 会广播事务，只要有超过半数节点写入成功，那么写请求就会被提交（类 2PC 事务）



那么问题来了

1. 集群中的 leader 节点如何选举出来?
2. leader 节点崩溃以后，整个集群无法处理写请求，如何快速从其他节点里面选举出新的 leader 呢?
3. leader 节点和各个 follower 节点的数据一致性如何保证

ZAB 协议

ZAB（Zookeeper Atomic Broadcast） 协议是为分布式协调服务 ZooKeeper 专门设计的一种支持崩溃恢复的原子广播协议。在 ZooKeeper 中，主要依赖 ZAB 协议来实现分布式数据一致性，基于该协议，ZooKeeper 实现了一种主备模式的系统架构来保持集群中各个副本之间的数据一致性。

zab 协议介绍

ZAB 协议包含两种基本模式，分别是

1. 崩溃恢复
2. 原子广播

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。

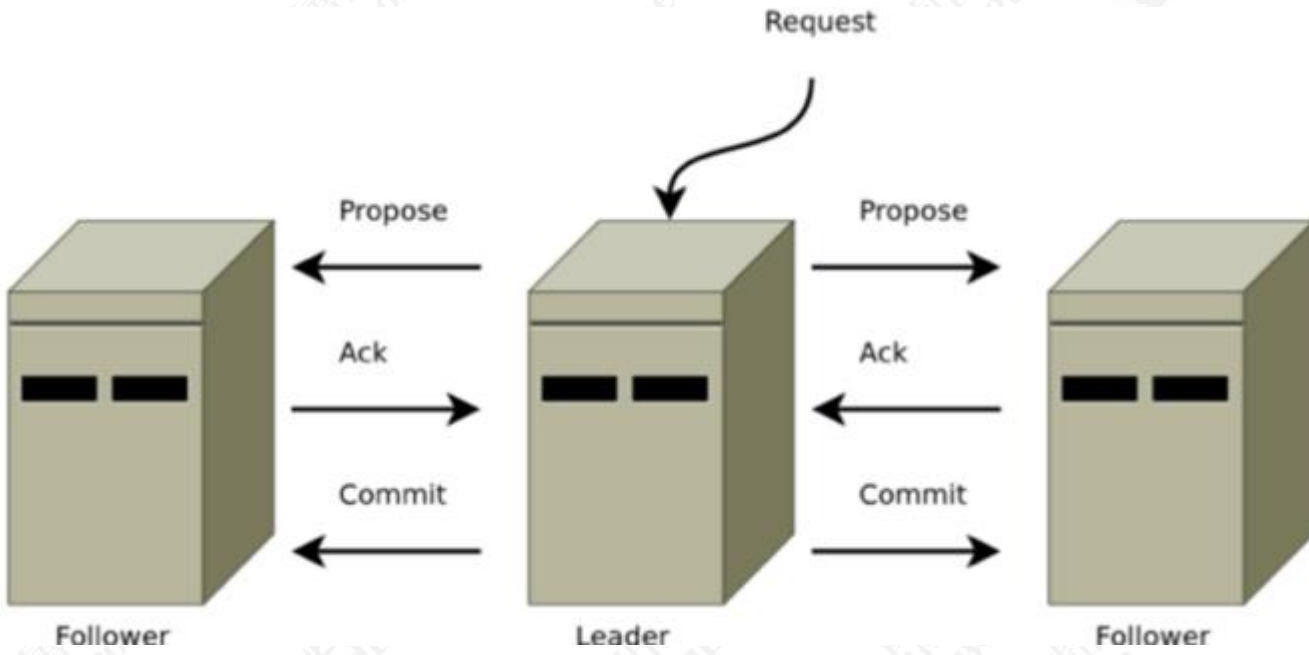
需要注意的是：leader 节点可以处理事务请求和非事务请求，follower 节点只能处理非事务请求，如果 follower 节点接收到非事务请求，会把这个请求转发给 Leader 服务器

消息广播的实现原理

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



ps: 和完整的 2pc 事务不一样的地方在于，zab 协议不能终止事务，follower 节点要么 ACK 给 leader，要么抛弃 leader，只需要保证过半数的节点响应这个消息并提交了即可，虽然在某一个时刻 follower 节点和 leader 节点的状态会不一致，但是也是这个特性提升了集群的整体性能。 当然这种数据不一致的问题，zab 协议提供了一种恢复模式来进行数据恢复，后续讲解

这里需要注意的是：

leader 的投票过程，不需要 Observer 的 ack，也就是 Observer 不需要参与投票过程，但是 Observer 必须要同步 Leader 的数据从而在处理请求的时候保证数据的一致性

崩溃恢复的实现原理

前面我们已经清楚了 ZAB 协议中的消息广播过程，ZAB 协议的这个基于原子广播协议的消息广播过程，在正常情况下是没有任何问题的，但是一旦 Leader 节点崩溃，或者由于网络问题导致 Leader 服务器失去了过半的 Follower 节点的联系（leader 失去与过半 follower 节点联系，可能是 leader 节点和 follower 节点之间产生了网络分区，那么此时的 leader 不再是合法的 leader 了），那么就会进入到崩溃恢复模式。崩溃恢复状态下 zab 协议需要做两件事

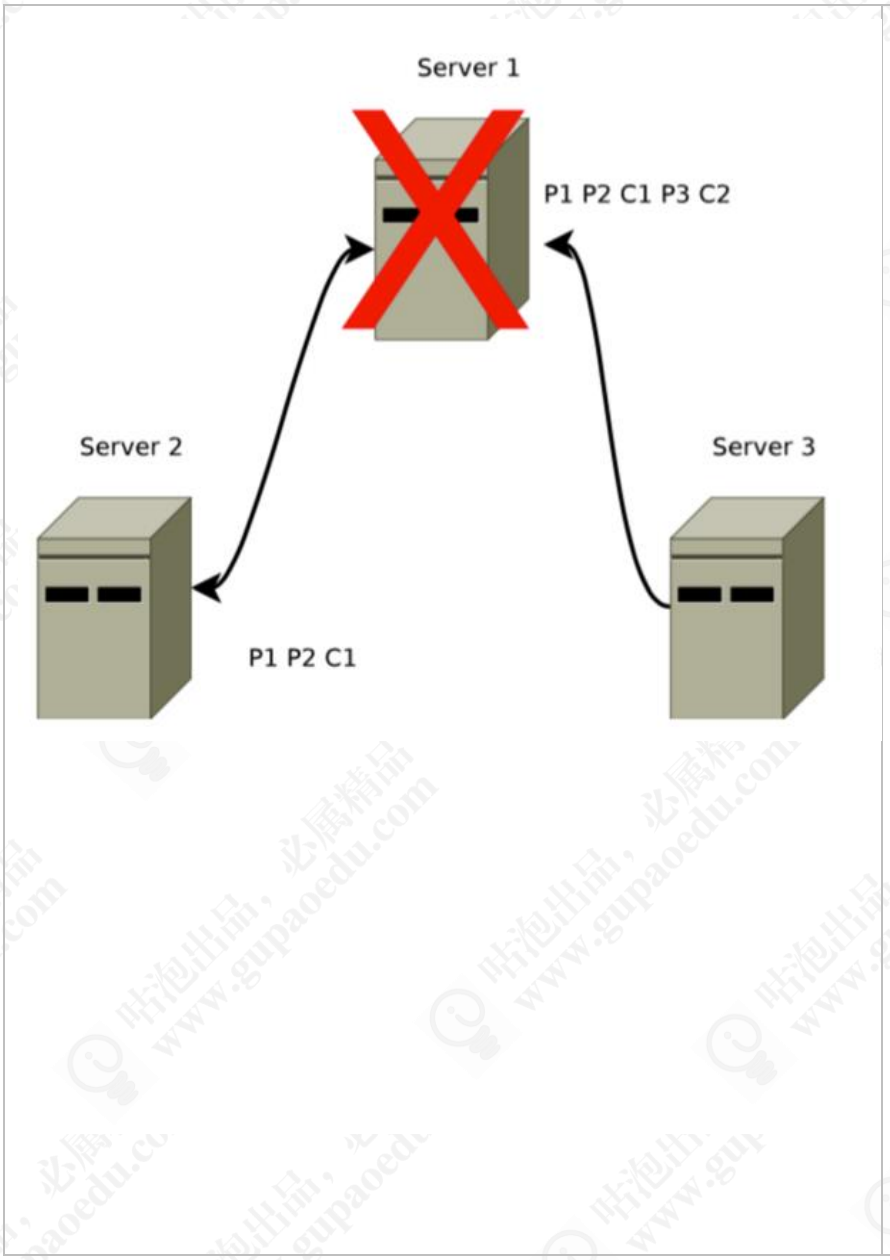
1. 选举出新的 leader
2. 数据同步

前面在讲解消息广播时，知道 ZAB 协议的消息广播机制是简化版本的 2PC 协议，这种协议只需要集群中过半的节点响应提交即可。但是它无法处理 Leader 服务器崩溃带来的数据不一致问题。因此在 ZAB 协议中添加了一个“崩溃恢复模式”来解决这个问题。

那么 ZAB 协议中的崩溃恢复需要保证，如果一个事务 Proposal 在一台机器上被处理成功，那么这个事务应该在所有机器上都被处理成功，哪怕是出现故障。为了达到这个目的，我们先来设想一下，在 zookeeper 中会有哪些场景导致数据不一致性，以及针对这个场景，zab 协议中的崩溃恢复应该怎么处理。

已经被处理的消息不能丢

当 leader 收到合法数量 follower 的 ACKs 后，就向各个 follower 广播 COMMIT 命令，同时也会在本地执行 COMMIT 并向连接的客户端返回「成功」。但是如果在各个 follower 在收到 COMMIT 命令前 leader 就挂了，导致剩下的服务器并没有执行都这条消息。

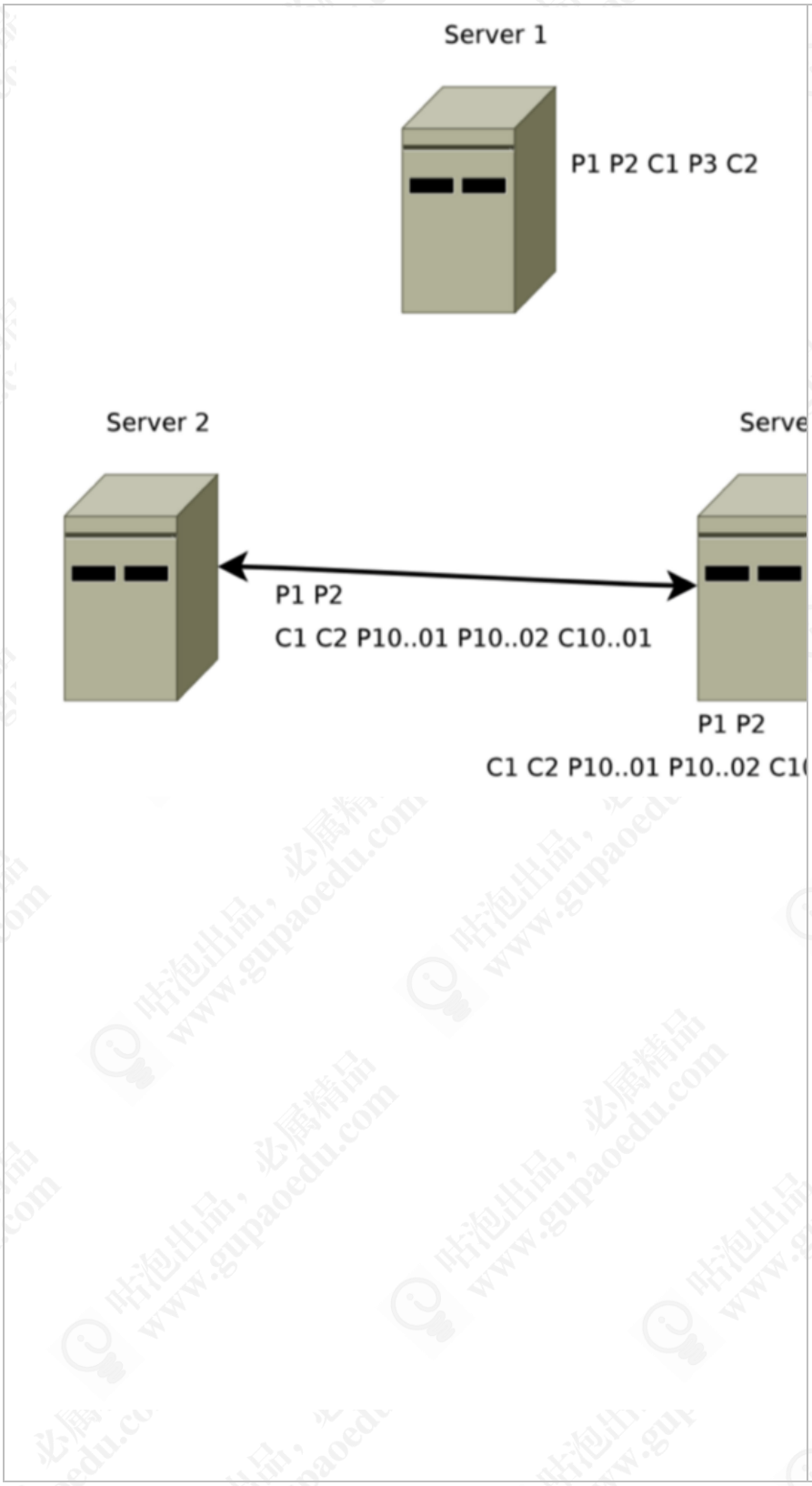


图中的 C2 就是一个典型的例子，在集群正常运行过程的某一个时刻，Server1 是 leader 服务器，先后

	广播了消息 P1、P2、C1、P3 和 C2.其中当 leader 服务器把消息 C2(Commit 事务 proposal2)发出后就立即崩溃退出了,那么针对这

	种情况，ZAB 协议就需要确保事务 Proposal2 最终能够在所有的服务器上都能被提交成功，否则将会出现不一致

被丢弃的消息不能再次出现



当 leader 接收到消息请求生成 proposal 后就挂了，其他 follower 并没有收到此 proposal，因此经过恢复模

	式重新选了 leader 后，这条消息是被跳过的。此时，之前挂了的 leader 重新启动并注册成了 follower，他保留了被跳过消息的

	proposal 状态，与整个系统的状态是不一致的，需要将其删除。

ZAB 协议需要满足上面两种情况，就必须要设计一个 leader 选举算法：能够确保已经被 leader 提交的事务 Proposal 能够提交、同时丢弃已经被跳过的事务 Proposal。针对这个要求

1. 如果 leader 选举算法能够保证新选举出来的 Leader 服务器拥有集群中所有机器最高编号（ZXID 最大）的事务 Proposal，那么就可以保证这个新选举出来的 Leader 一定具有已经提交的提案。因为所有提案被 COMMIT 之前必须有超过半数的 follower ACK，即必须有超过半数

节点的服务器的事务日志上有该提案的 proposal，因此，只要有合法数量的节点正常工作，就必然有一个节点保存了所有被 COMMIT 消息的 proposal 状态

2. 另外一个，zxid 是 64 位，高 32 位是 epoch 编号，每经过一次 Leader 选举产生一个新的 leader，新的 leader 会将 epoch 号+1，低 32 位是消息计数器，每接收到一条消息这个值+1，新 leader 选举后这个值重置为 0.这样设计的好处在于老的 leader 挂了以后重启，它不会被选举为 leader，因此此时它的 zxid 肯定小于当前新的 leader。当老的 leader 作为 follower 接入新的 leader 后，新的 leader 会让它将所有的拥有旧的 epoch 号的未被 COMMIT 的 proposal 清除

关于 ZXID

前面一直提到 zxid，也就是事务 id，那么这个 id 具体起什么作用，以及这个 id 是如何生成的，简单给大家解释下

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（ZAB 协议通过 epoch 编号来区分 Leader 周期变化的策略）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=（原来的 epoch+1），标识当前属于那个 leader 的统治时期。低 32 位用于递增计数。

epoch：可以理解为当前集群所处的年代或者周期，每个 leader 就像皇帝，都有自己的年号，所以每次改朝换代，leader 变更之后，都会在前一个年代的基础上加 1。这样就算旧的 leader 崩溃恢复之后，也没有人听他的了，因为 follower 只听从当前年代的 leader 的命令。