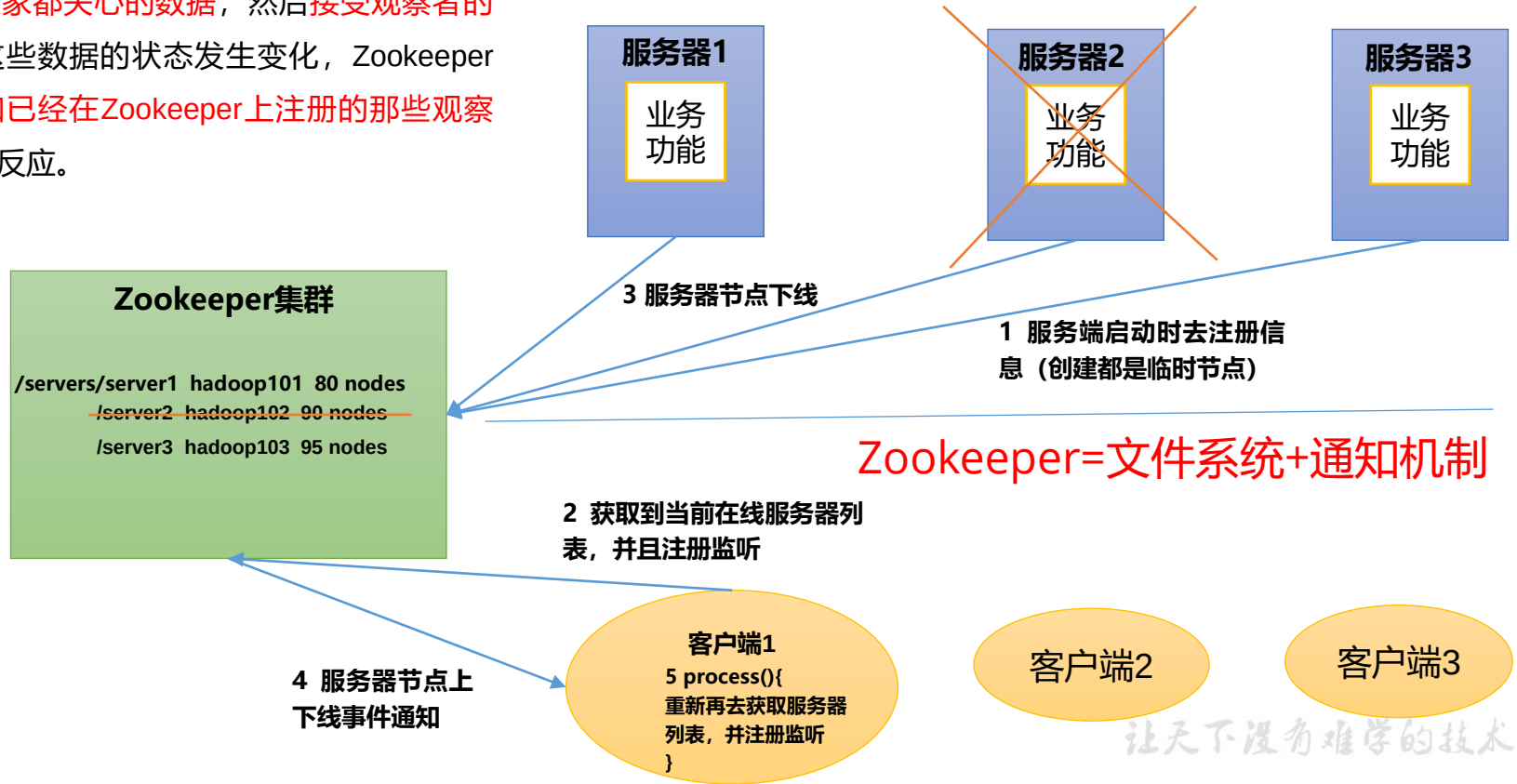
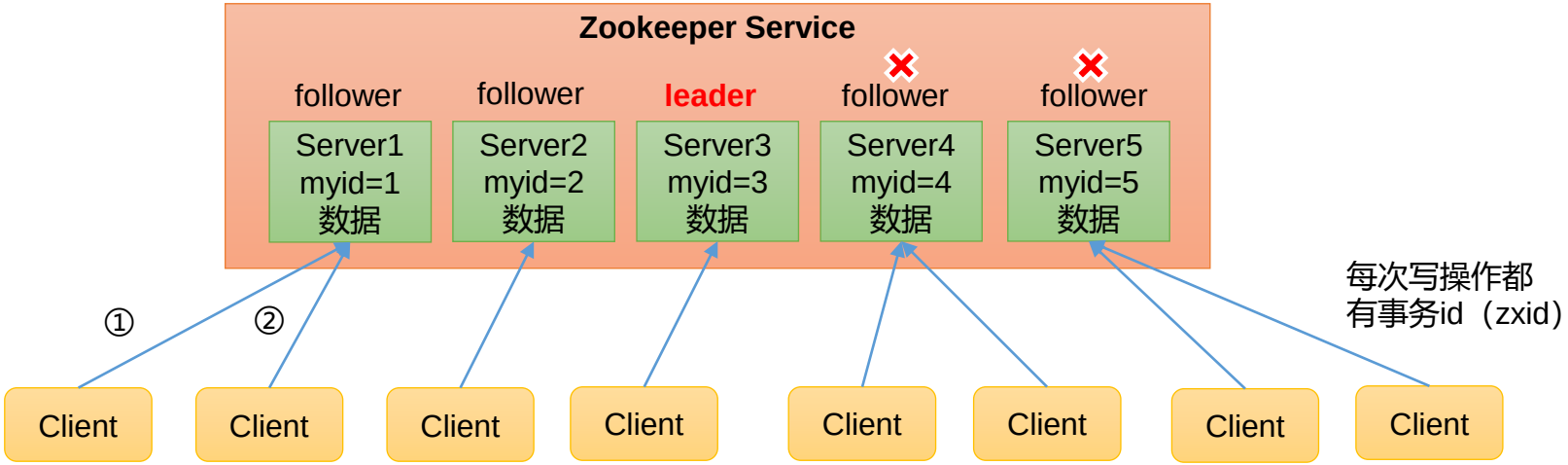
Zookeeper工作机制

Zookeeper从设计模式角度来理解：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。



Zookeeper特点



1）Zookeeper：一个领导者（Leader），多个跟随者（Follower）组成的集群。

2）集群中只要有半数以上节点存活，Zookeeper集群就能正常服务。所以Zookeeper适合安装奇数台服务器。

3）全局数据一致：每个Server保存一份相同的数据副本，Client无论连接到哪个Server，数据都是一致的。

4）更新请求顺序执行，来自同一个Client的更新请求按其发送顺序依次执行。

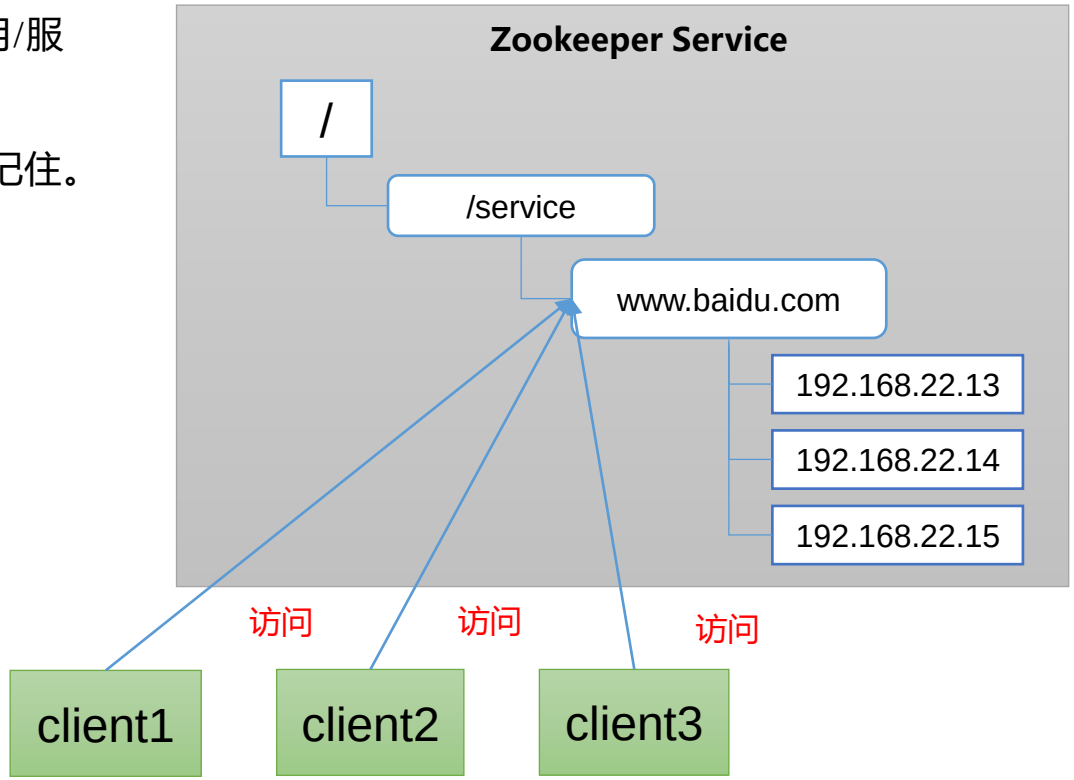
5）数据更新原子性，一次数据更新要么成功，要么失败。

6）实时性，在一定时间范围内，Client能读到最新数据。

统一命名服务

在分布式环境下，经常需要对应用/服务进行统一命名，便于识别。

例如：IP不容易记住，而域名容易记住。



统一配置管理

1）分布式环境下，配置文件同步非常常见。

（1）一般要求一个集群中，所有节点的配置信息是一致的，比如 Kafka 集群。

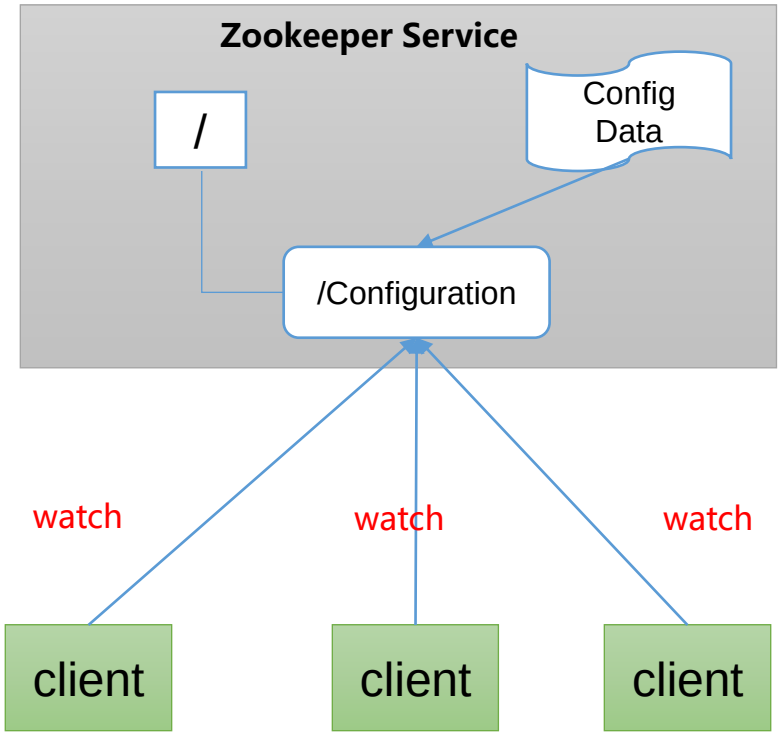
（2）对配置文件修改后，希望能够快速同步到各个节点上。

2）配置管理可交由ZooKeeper实现。

（1）可将配置信息写入ZooKeeper上的一个Znode。

（2）各个客户端服务器监听这个Znode。

（3）一旦Znode中的数据被修改，ZooKeeper将通知各个客户端服务器。



统一集群管理

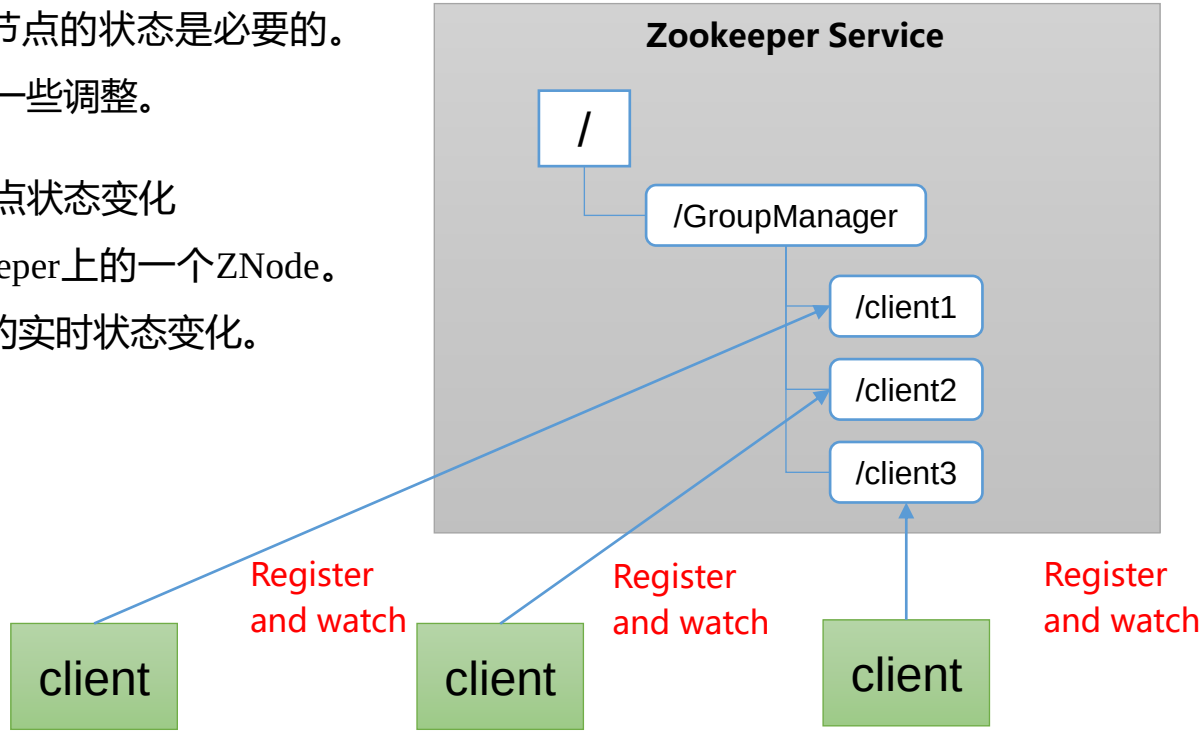
1）分布式环境中，实时掌握每个节点的状态是必要的。

（1）可根据节点实时状态做出一些调整。

2）ZooKeeper可以实现实时监控节点状态变化

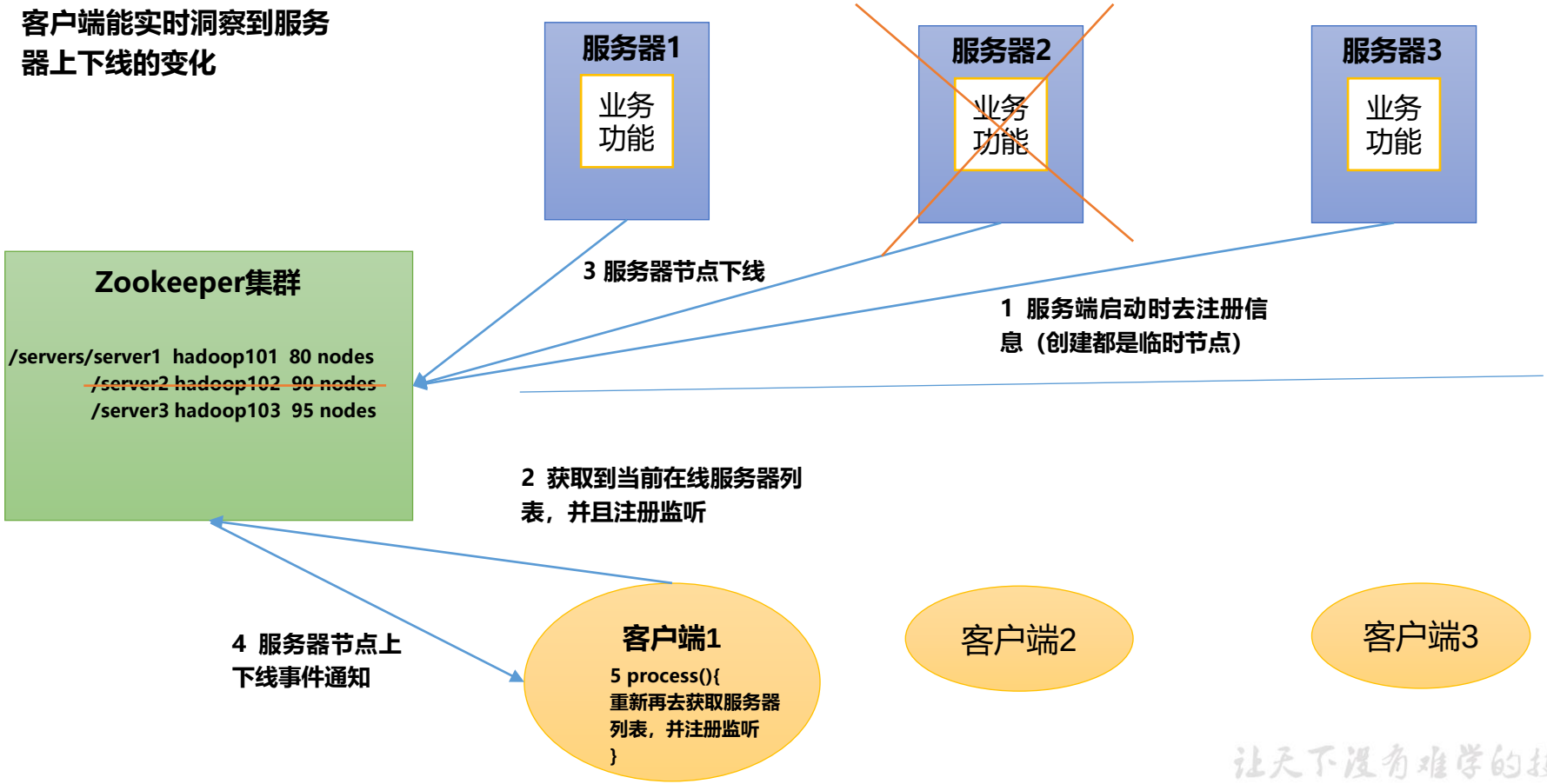
（1）可将节点信息写入ZooKeeper上的一个ZNode。

（2）监听这个ZNode可获取它的实时状态变化。



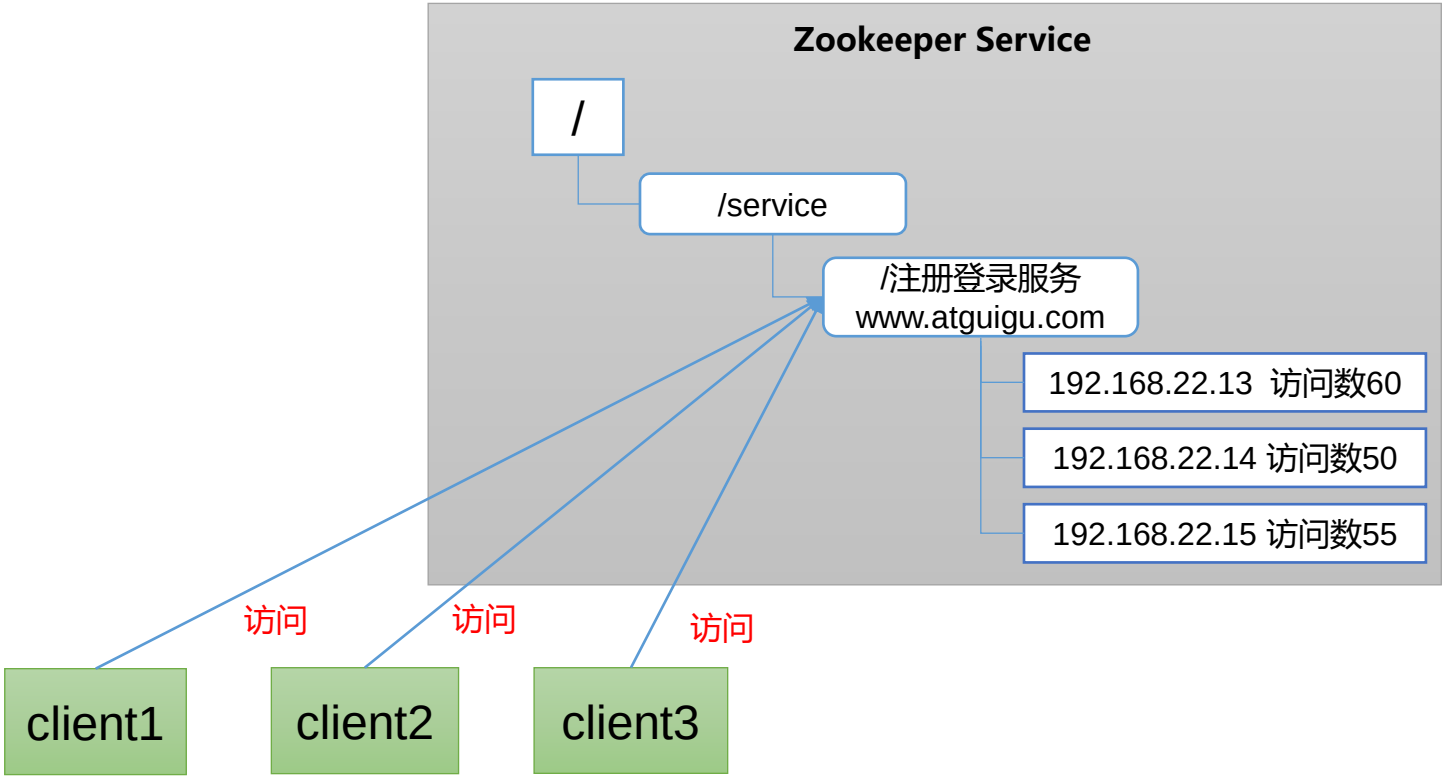
服务器动态上下线

客户端能实时洞察到服务器上下线的变化

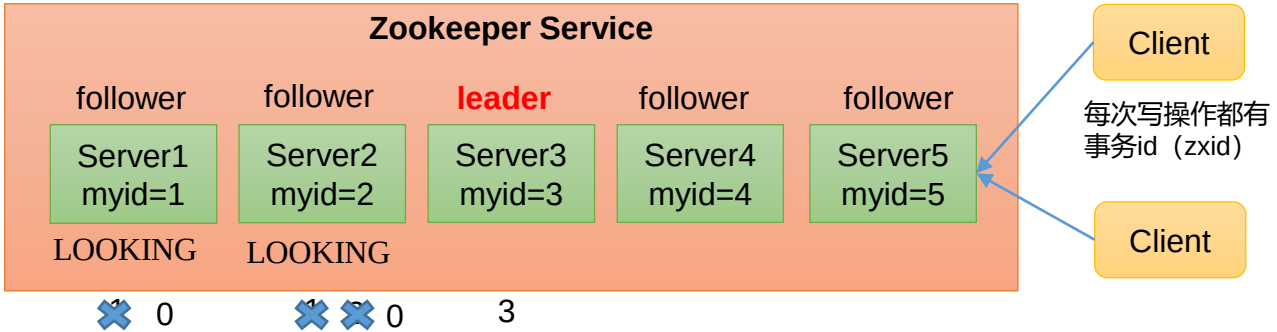


软负载均衡

在Zookeeper中记录每台服务器的访问数，让访问数最少的服务器去处理最新的客户端请求



Zookeeper选举机制——第一次启动



SID：服务器ID。用来唯一标识一台ZooKeeper集群中的机器，每台机器不能重复，和myid一致。

ZXID：事务ID。ZXID是一个事务ID，用来标识一次服务器状态的变更。在某一时刻，集群中的每台机器的ZXID值不一定完全一致，这和ZooKeeper服务器对于客户端“更新请求”的处理逻辑有关。

Epoch：每个Leader任期的代号。没有Leader时同一轮投票过程中的逻辑时钟值是相同的。每投完一次票这个数据就会增加

（1）服务器1启动，发起一次选举。服务器1投自己一票。此时服务器1票数一票，不够半数以上（3票），选举无法完成，服务器1状态保持为LOOKING；

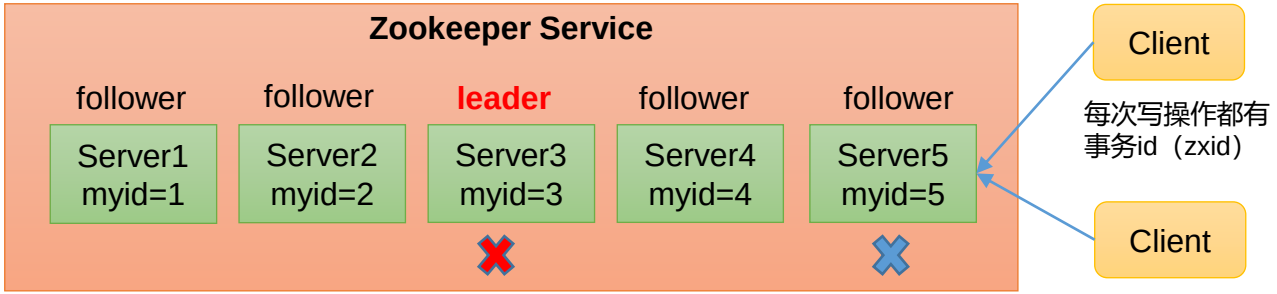
（2）服务器2启动，再发起一次选举。服务器1和2分别投自己一票并交换选票信息：此时服务器1发现服务器2的myid比自己目前投票推举的（服务器1）大，更改选票为推举服务器2。此时服务器1票数0票，服务器2票数2票，没有半数以上结果，选举无法完成，服务器1，2状态保持LOOKING

（3）服务器3启动，发起一次选举。此时服务器1和2都会更改选票为服务器3。此次投票结果：服务器1为0票，服务器2为0票，服务器3为3票。此时服务器3的票数已经超过半数，服务器3当选Leader。服务器1，2更改状态为FOLLOWING，服务器3更改状态为LEADING；

（4）服务器4启动，发起一次选举。此时服务器1，2，3已经不是LOOKING状态，不会更改选票信息。交换选票信息结果：服务器3为3票，服务器4为1票。此时服务器4服从多数，更改选票信息为服务器3，并更改状态为FOLLOWING；

（5）服务器5启动，同4一样当小弟。

Zookeeper选举机制——非第一次启动



SID：服务器ID。用来唯一标识一台ZooKeeper集群中的机器，每台机器不能重复，和myid一致。

ZXID：事务ID。ZXID是一个事务ID，用来标识一次服务器状态的变更。在某一时刻，集群中的每台机器的ZXID值不一定完全一致，这和ZooKeeper服务器对于客户端“更新请求”的处理逻辑有关。

Epoch：每个Leader任期的代号。没有Leader时同一轮投票过程中的逻辑时钟值是相同的。每投完一次票这个数据就会增加

（1）当ZooKeeper集群中的一台服务器出现以下两种情况之一时，就会开始进入Leader选举：

- 服务器初始化启动。
- 服务器运行期间无法和Leader保持连接。

（2）而当一台机器进入Leader选举流程时，当前集群也可能会处于以下两种状态：

- 集群中本来就已经存在一个Leader。
  对于第一种已经存在Leader的情况，机器试图去选举Leader时，会被告知当前服务器的Leader信息，对于该机器来说，仅仅需要和Leader机器建立连接，并进行状态同步即可。
- **集群中确实不存在Leader。**
  假设ZooKeeper由5台服务器组成，SID分别为1、2、3、4、5，ZXID分别为8、8、8、7、7，并且此时SID为3的服务器是Leader。某一时刻，3和5服务器出现故障，因此开始进行Leader选举。

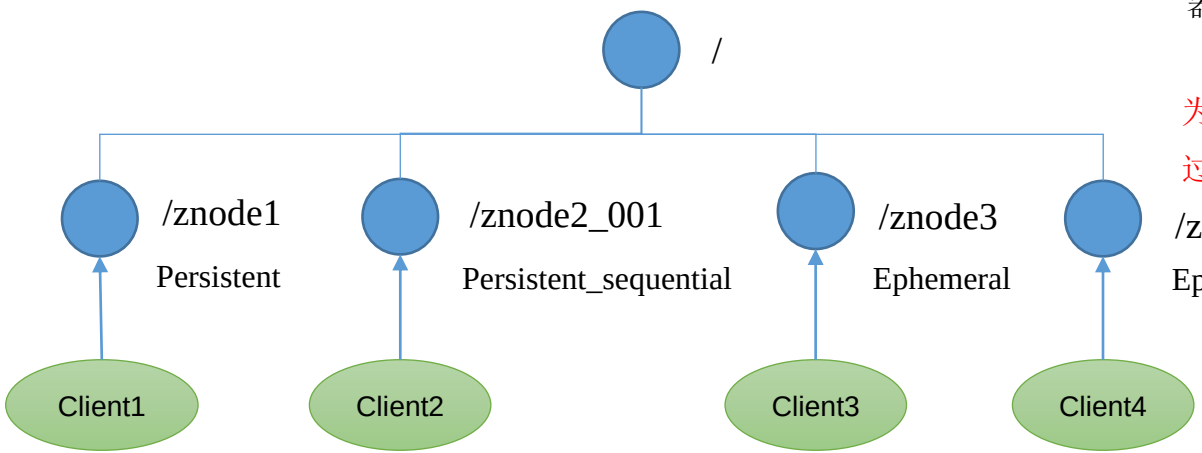
	（EPOCH，ZXID，SID ）	（EPOCH，ZXID，SID ）	（EPOCH，ZXID，SID ）
SID为1、2、4的机器投票情况：	（1，8，1）	（1，8，2）	（1，7，4）

选举Leader规则： ①EPOCH大的直接胜出　②EPOCH相同，事务id大的胜出　③事务id相同，服务器id大的胜出

节点类型

持久（Persistent）：客户端和服务器端断开连接后，创建的节点不删除

短暂（Ephemeral）：客户端和服务器端断开连接后，创建的节点自己删除



说明：创建znode时设置顺序标识，znode名称后会附加一个值，顺序号是一个单调递增的计数器，由父节点维护

注意：在分布式系统中，顺序号可以被用于为所有的事件进行全局排序，这样客户端可以通过顺序号推断事件的顺序

（1）持久化目录节点

客户端与Zookeeper断开连接后，该节点依旧存在

（2）持久化顺序编号目录节点

客户端与Zookeeper断开连接后，该节点依旧存在，只是Zookeeper给该节点名称进行顺序编号

（3）临时目录节点

客户端与Zookeeper断开连接后，该节点被删除

（4）临时顺序编号目录节点

客户端与Zookeeper断开连接后，该节点被删除，只是Zookeeper给该节点名称进行顺序编号。

监听器原理

1、监听原理详解

1）首先要有一个main()线程

2）在main线程中创建Zookeeper客户端，这时就会创建两个线程，一个负责网络连接通信（connet），一个负责监听（listener）。

3）通过connect线程将注册的监听事件发送给Zookeeper。

4）在Zookeeper的注册监听器列表中将注册的监听事件添加到列表中。

5）Zookeeper监听到有数据或路径变化，就会将这个消息发送给listener线程。

6）listener线程内部调用了process()方法。

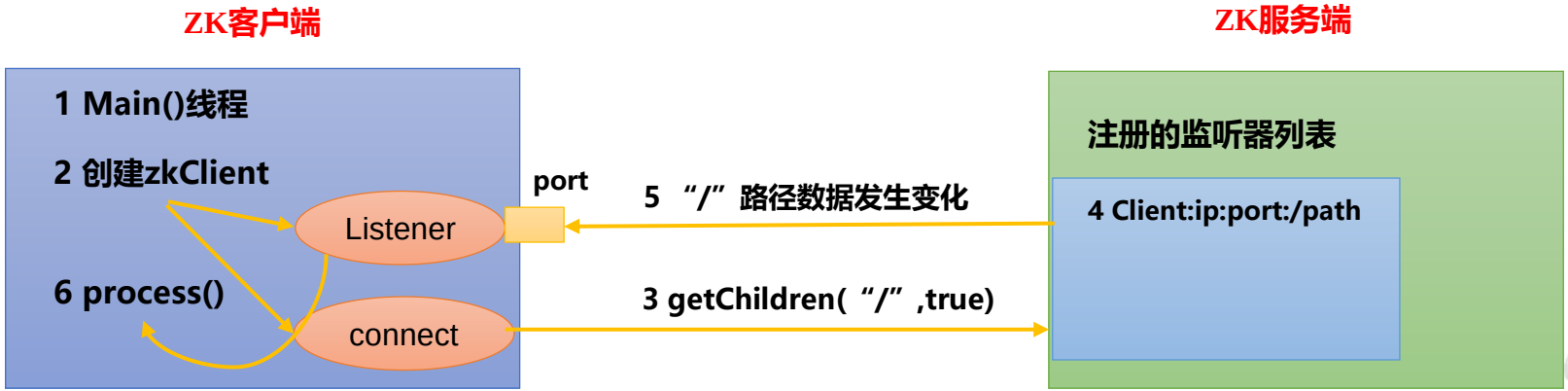
2、常见的监听

1）监听节点数据的变化

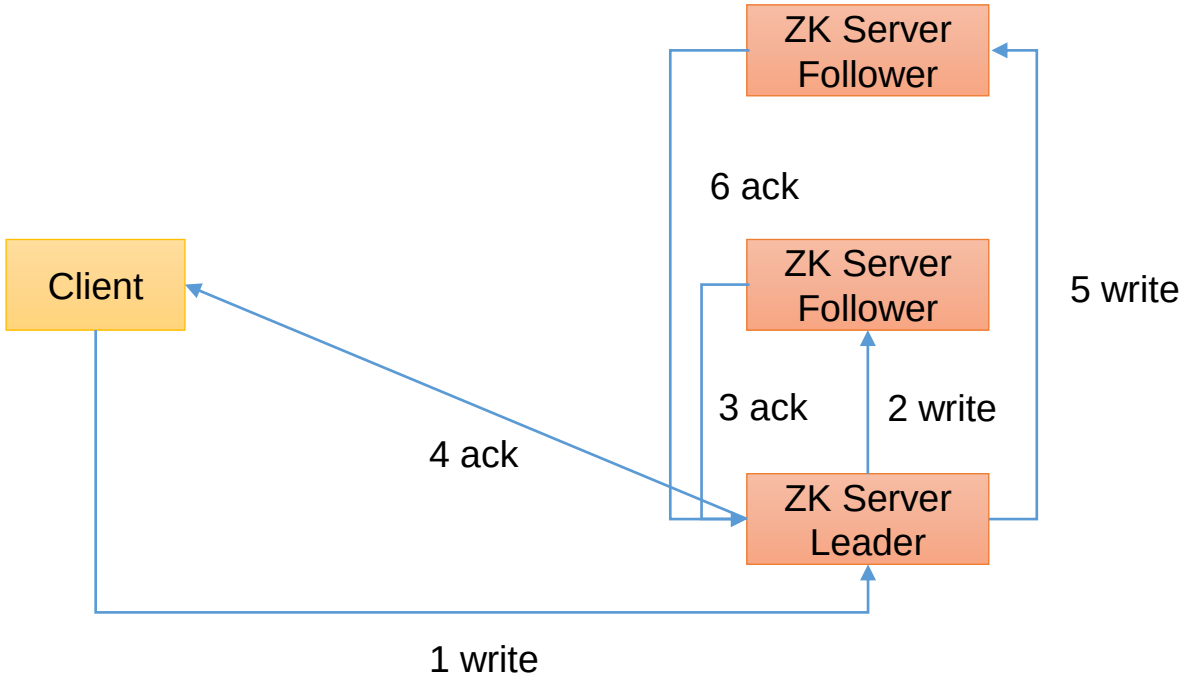
get path [watch]

2）监听子节点增减的变化

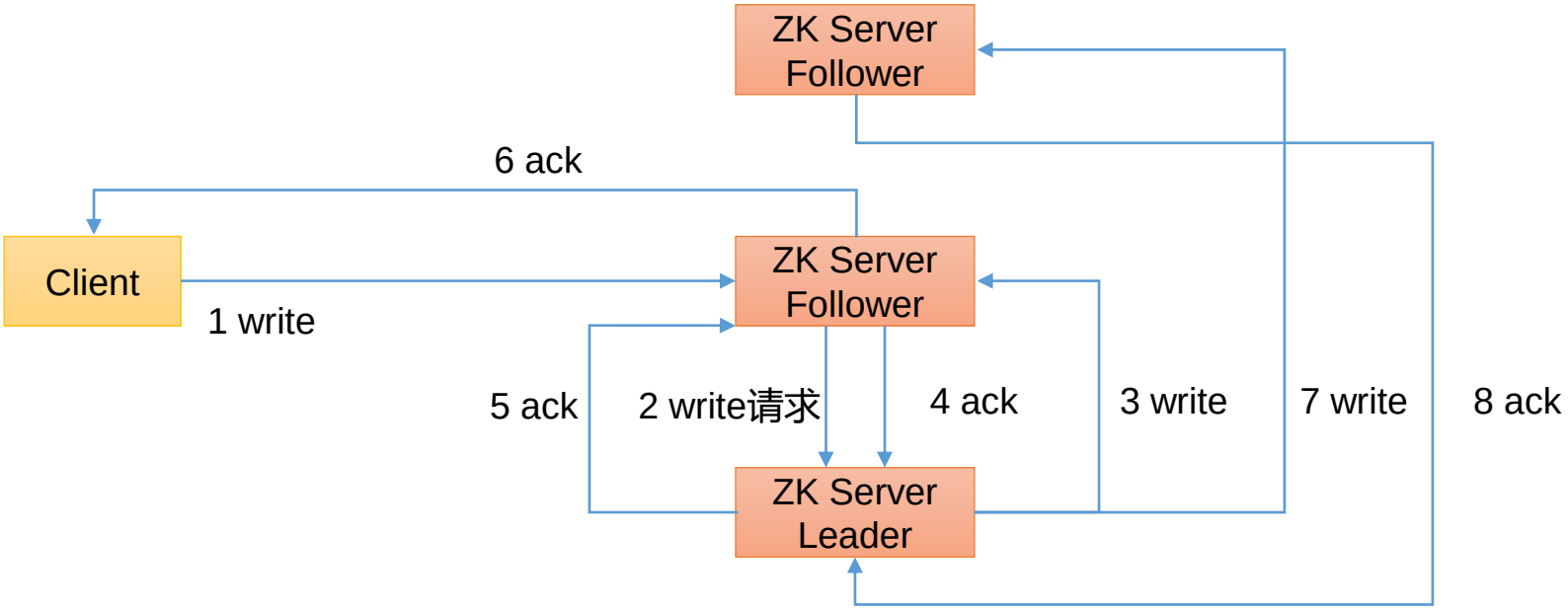
ls path [watch]



写流程之写入请求直接发送给Leader节点

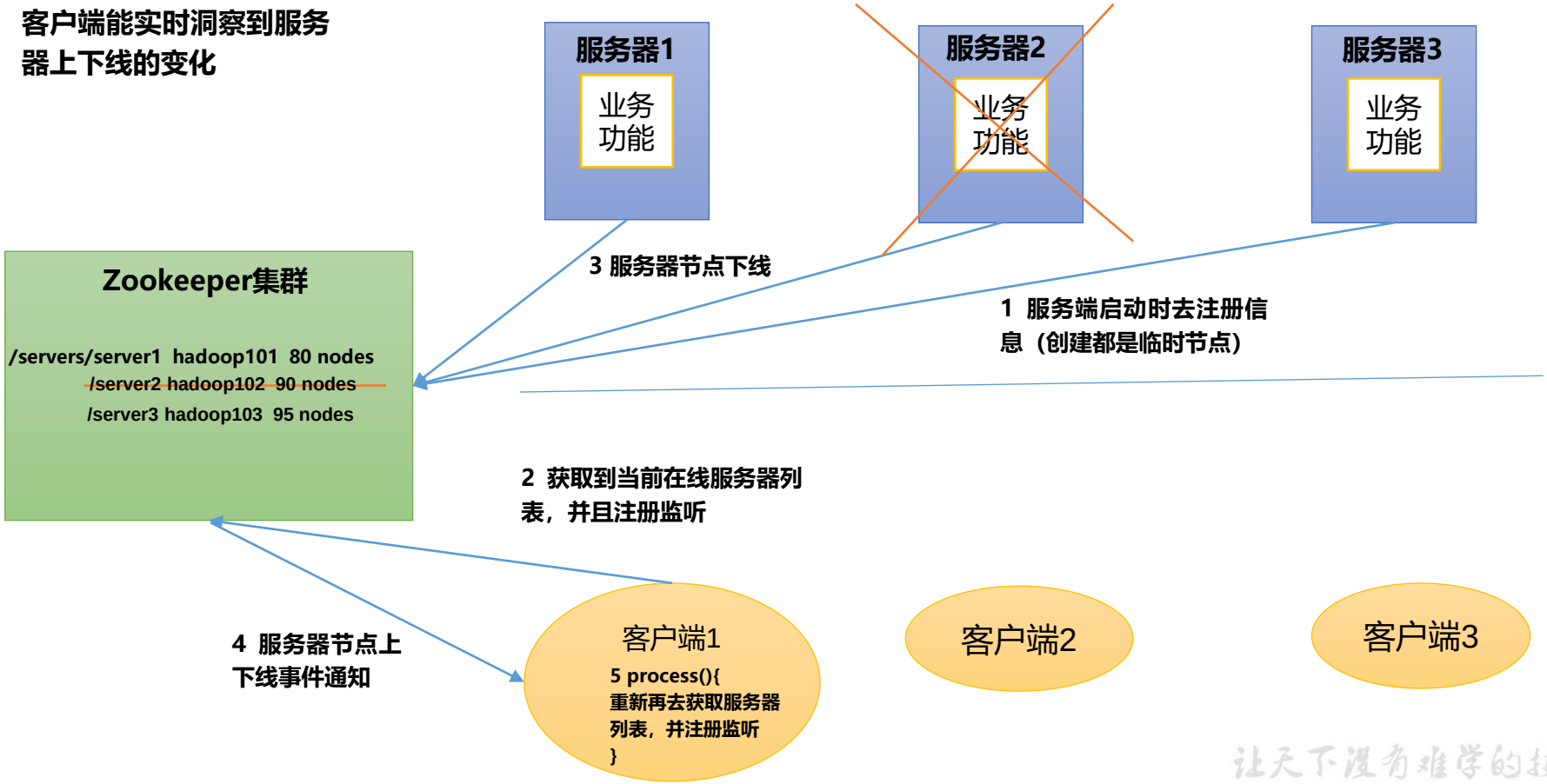


写流程之写入请求发送给follower节点

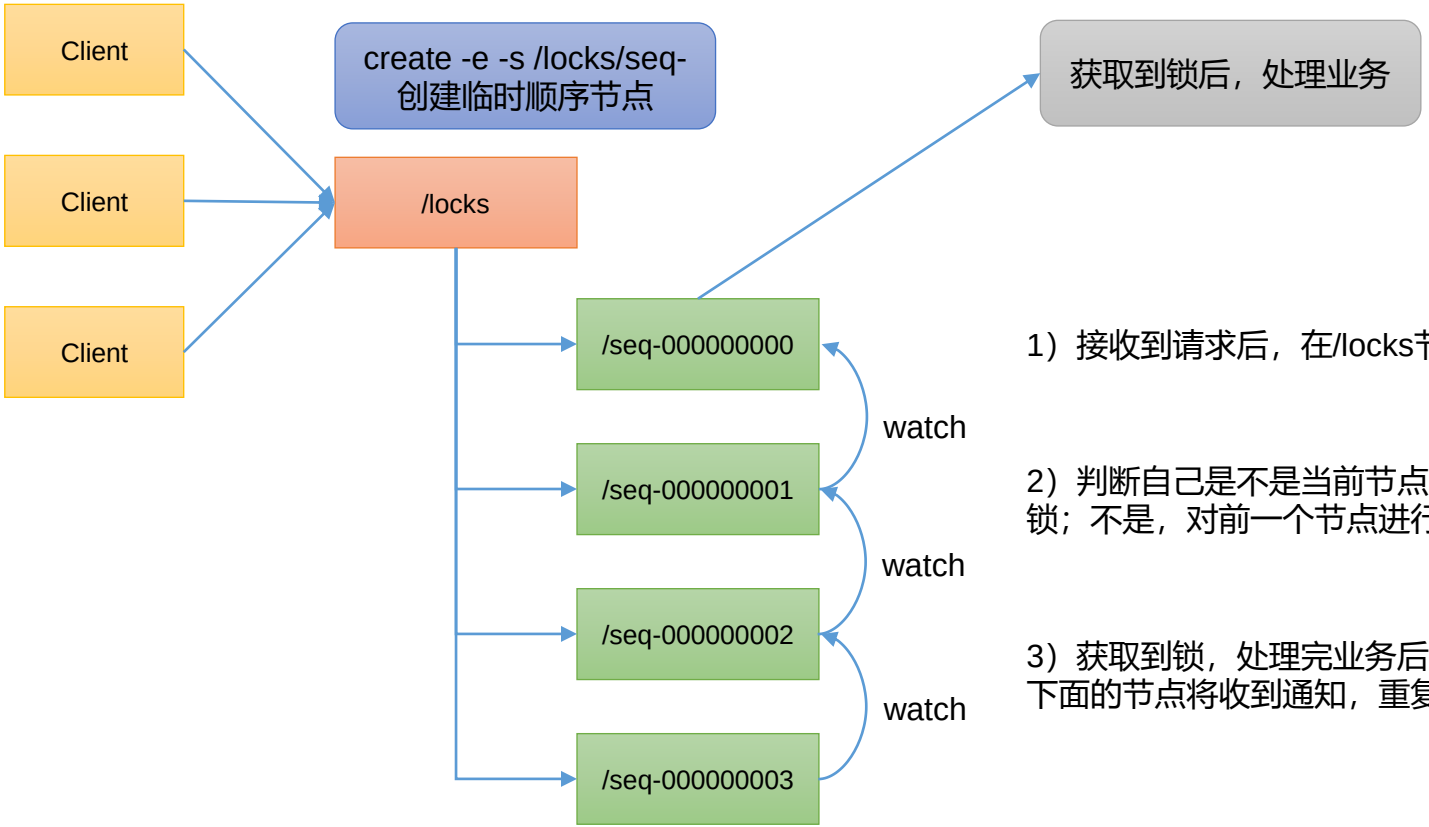


服务器动态上下线

客户端能实时洞察到服务器上下线的变化



分布式锁案例分析



1）接收到请求后，在/locks节点下创建一个临时顺序节点

2）判断自己是不是当前节点下最小的节点：是，获取到锁；不是，对前一个节点进行监听

3）获取到锁，处理完业务后，delete节点释放锁，然后下面的节点将收到通知，重复第二步判断

拜占庭将军问题

拜占庭将军问题是一个协议问题，拜占庭帝国军队的将军们必须全体一致的决定是否攻击某一支敌军。问题是这些将军在地理上是分隔开来的，并且将军中存在叛徒。叛徒可以任意行动以达到以下目标：**欺骗某些将军采取进攻行动；促成一个不是所有将军都同意的决定，如当将军们不希望进攻时促成进攻行动；或者迷惑某些将军，使他们无法做出决定**。如果叛徒达到了这些目的之一，则任何攻击行动的结果都是注定要失败的，只有完全达成一致的努力才能获得胜利。



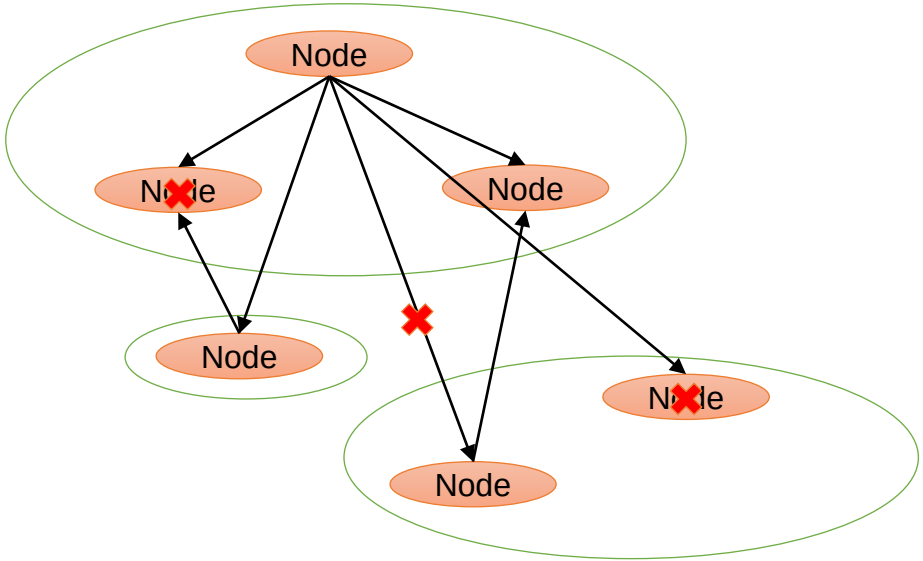





Paxos算法——解决什么问题

Paxos算法：一种基于消息传递且具有高度容错特性的**一致性算法**。

Paxos算法解决的问题：就是如何快速正确的在一个分布式系统中对某个数据值达成一致，并且保证不论发生任何异常，都不会破坏整个系统的一致性。

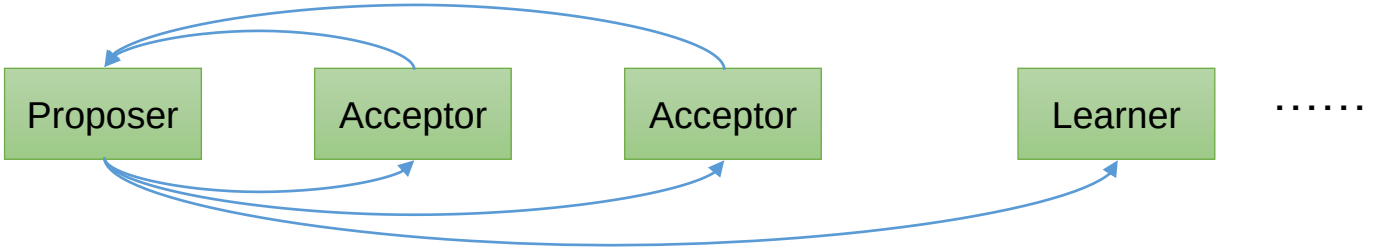


机器宕机

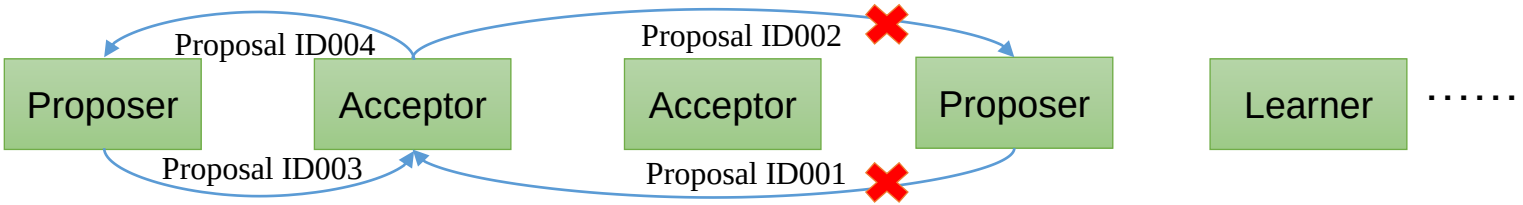
网络异常（延迟、重复、丢失）

Paxos算法描述：

- 在一个Paxos系统中，首先将所有节点划分为Proposer（提议者），Acceptor（接受者），和Learner（学习者）。（注意：每个节点都可以身兼数职）。



- 一个完整的Paxos算法流程分为三个阶段：
- Prepare准备阶段
  - Proposer向多个Acceptor发出Propose请求Promise（承诺）
  - Acceptor针对收到的Propose请求进行Promise（承诺）
- Accept接受阶段
  - Proposer收到多数Acceptor承诺的Promise后，向Acceptor发出Propose请求
  - Acceptor针对收到的Propose请求进行Accept处理
- Learn学习阶段：Proposer将形成的决议发送给所有Learners



（1）Prepare: Proposer生成全局唯一且递增的Proposal ID，向所有Acceptor发送Propose请求，这里无需携带提案内容，只携带Proposal ID即可。

（2）Promise: Acceptor收到Propose请求后，做出“两个承诺，一个应答”。

- 不再接受Proposal ID小于等于（注意：这里是<= ）当前请求的Propose请求。
- 不再接受Proposal ID小于（注意：这里是< ）当前请求的Accept请求。
- 不违背以前做出的承诺下，回复已经Accept过的提案中Proposal ID最大的那个提案的Value和Proposal ID，没有则返回空值。

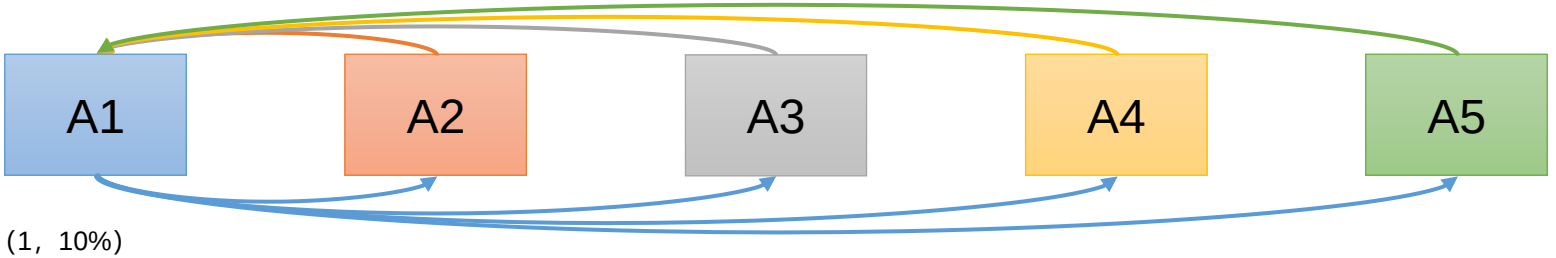
（3）Propose: Proposer收到多数Acceptor的Promise应答后，从应答中选择Proposal ID最大的提案的Value，作为本次要发起的提案。如果所有应答的提案Value均为空值，则可以自己随意决定提案Value。然后携带当前Proposal ID，向所有Acceptor发送Propose请求。

（4）Accept: Acceptor收到Propose请求后，在不违背自己之前做出的承诺下，接受并持久化当前Proposal ID和提案Value。

（5）Learn: Proposer收到多数Acceptor的Accept后，决议形成，将形成的决议发送给所有Learner。

情况1：

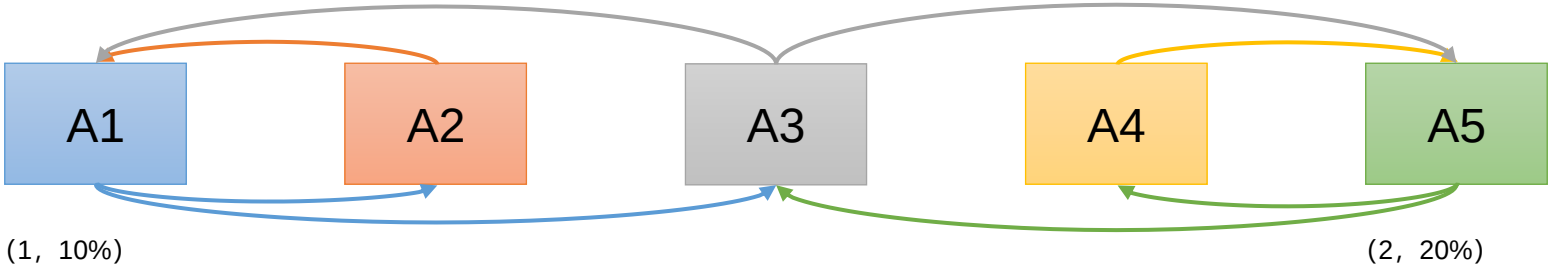
- 有A1, A2, A3, A4, A5 5位议员，就税率问题进行决议。



- A1发起1号Proposal的Propose，等待Promise承诺；
- A2-A5回应Promise；
- A1在收到两份回复时就会发起税率10%的Proposal；
- A2-A5回应Accept；
- 通过Proposal，税率10%。

情况2：

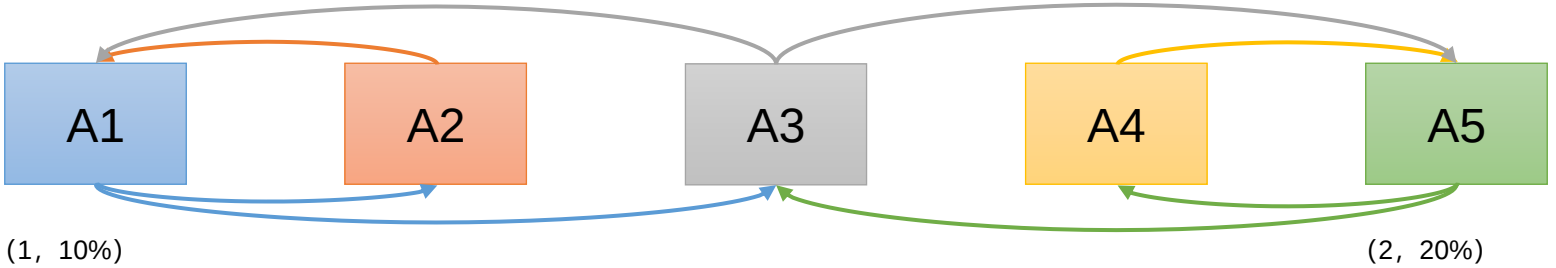
- 现在我们假设在A1提出提案的同时，A5决定将税率定为20%



- A1，A5同时发起Propose（序号分别为1，2）
- A2承诺A1，A4承诺A5，A3行为成为关键
- 情况1：A3先收到A1消息，承诺A1。
- A1发起Proposal（1，10%），A2，A3接受。
- 之后A3又收到A5消息，回复A1：（1，10%），并承诺A5。
- A5发起Proposal（2，20%），A3，A4接受。之后A1，A5同时广播决议。

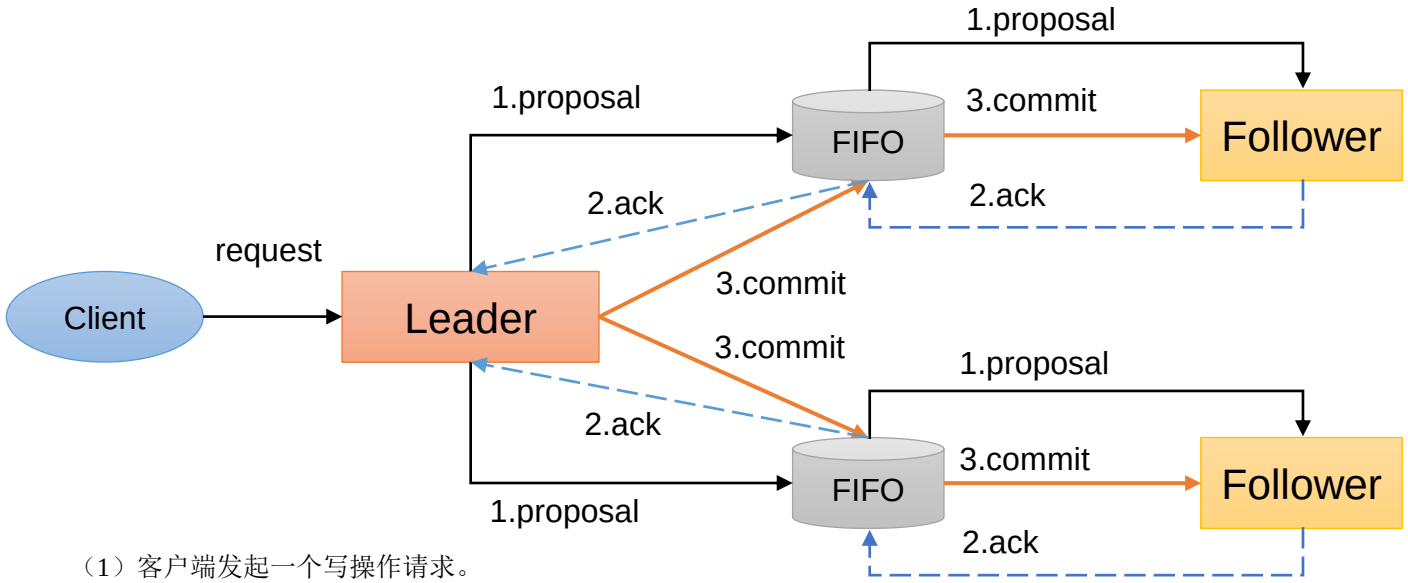
情况3：

- 现在我们假设在A1提出提案的同时，A5决定将税率定为20%



- A1，A5同时发起Propose（序号分别为1，2）
- A2承诺A1，A4承诺A5，A3行为成为关键
- 情况2：A3先收到A1消息，承诺A1。之后立刻收到A5消息，承诺A5。
- A1发起Proposal（1，10%），无足够响应，A1重新Propose （序号3），A3再次承诺A1。
- A5发起Proposal（2，20%），无足够相应。 A5重新Propose （序号4），A3再次承诺A5。
- ……

消息广播



ZAB协议针对事务请求的处理过程类似于一个两阶段提交过程

（1）广播事务阶段

（2）广播提交操作

这两阶段提交模型如下，有可能因为Leader宕机带来数据不一致，比如

（1）Leader发起一个事务Proposal1后就宕机，Follower都没有Proposal1

（2）Leader收到半数ACK宕机，没来得及向Follower发送Commit

怎么解决呢？**ZAB引入了崩溃恢复模式。**

（1）客户端发起一个写操作请求。

（2）Leader服务器将客户端的请求转化为事务Proposal 提案，同时为每个Proposal 分配一个全局的ID，即zxid。

（3）Leader服务器为每个Follower服务器分配一个单独的队列，然后将需要广播的 Proposal依次放到队列中去，并且根据FIFO策略进行消息发送。

（4）Follower接收到Proposal后，会首先将其以事务日志的方式写入本地磁盘中，写入成功后向Leader反馈一个Ack响应消息。

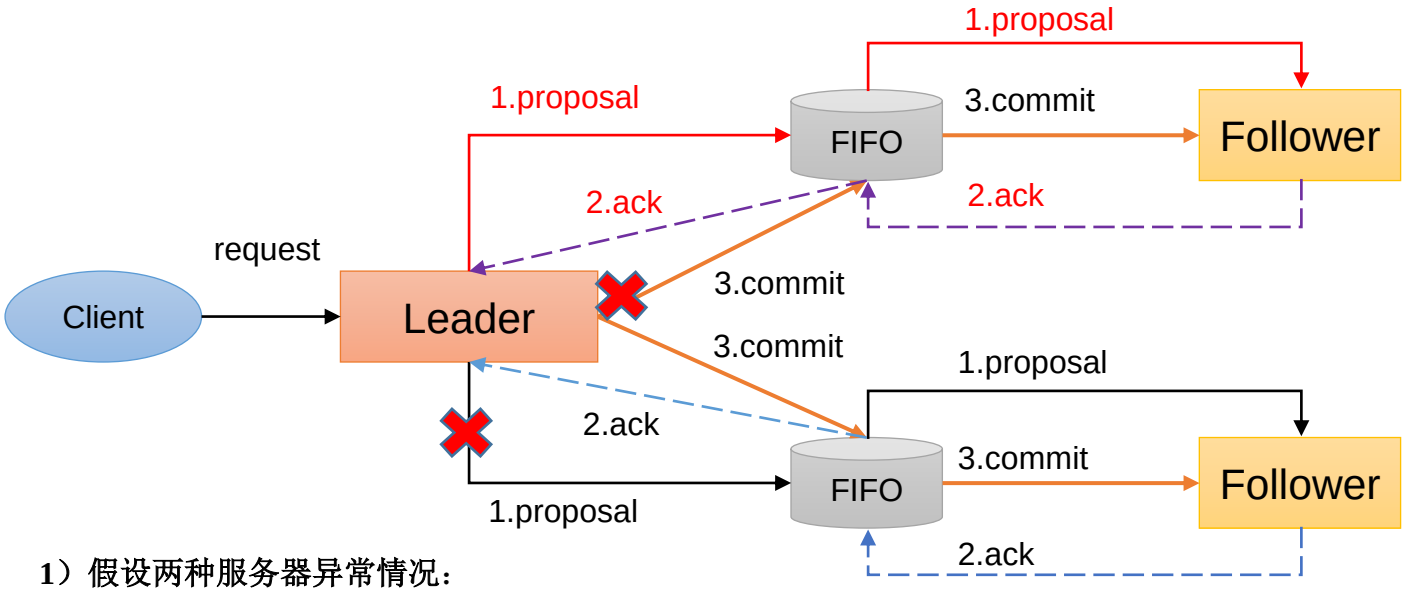
（5）Leader接收到超过半数以上Follower的Ack响应消息后，即认为消息发送成功，可以发送commit消息。

（6）Leader向所有Follower广播commit消息，同时自身也会完成事务提交。Follower 接收到commit消息后，会将上一条事务提交。

（7）Zookeeper采用Zab协议的核心，就是只要有一台服务器提交了Proposal，就要确保所有的服务器最终都能正确提交Proposal。

崩溃恢复——异常假设

一旦Leader服务器出现崩溃或者由于网络原因导致Leader服务器失去了与过半 Follower的联系，那么就会进入**崩溃恢复模式。**



1）假设两种服务器异常情况：

（1）假设一个事务在Leader提出之后，Leader挂了。

（2）一个事务在Leader上提交了，并且过半的Follower都响应Ack了，但是Leader在Commit消息发出之前挂了。

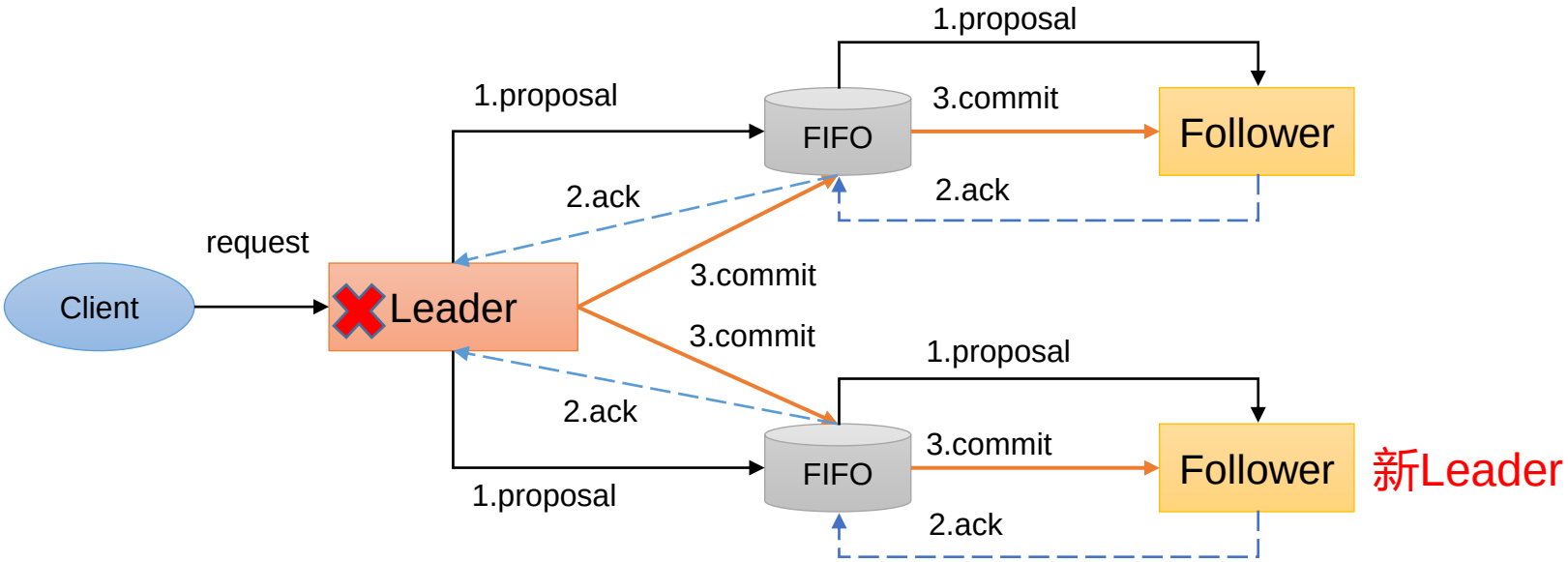
2）Zab协议崩溃恢复要求满足以下两个要求：

（1）确保已经被Leader提交的提案Proposal，必须最终被所有的Follower服务器提交。 （已经产生的提案，Follower必须执行）

（2）确保**丢弃**已经被Leader提出的，但是没有被提交的Proposal。（丢弃胎死腹中的提案）

崩溃恢复——Leader选举

崩溃恢复主要包括两部分：**Leader选举**和**数据恢复**。

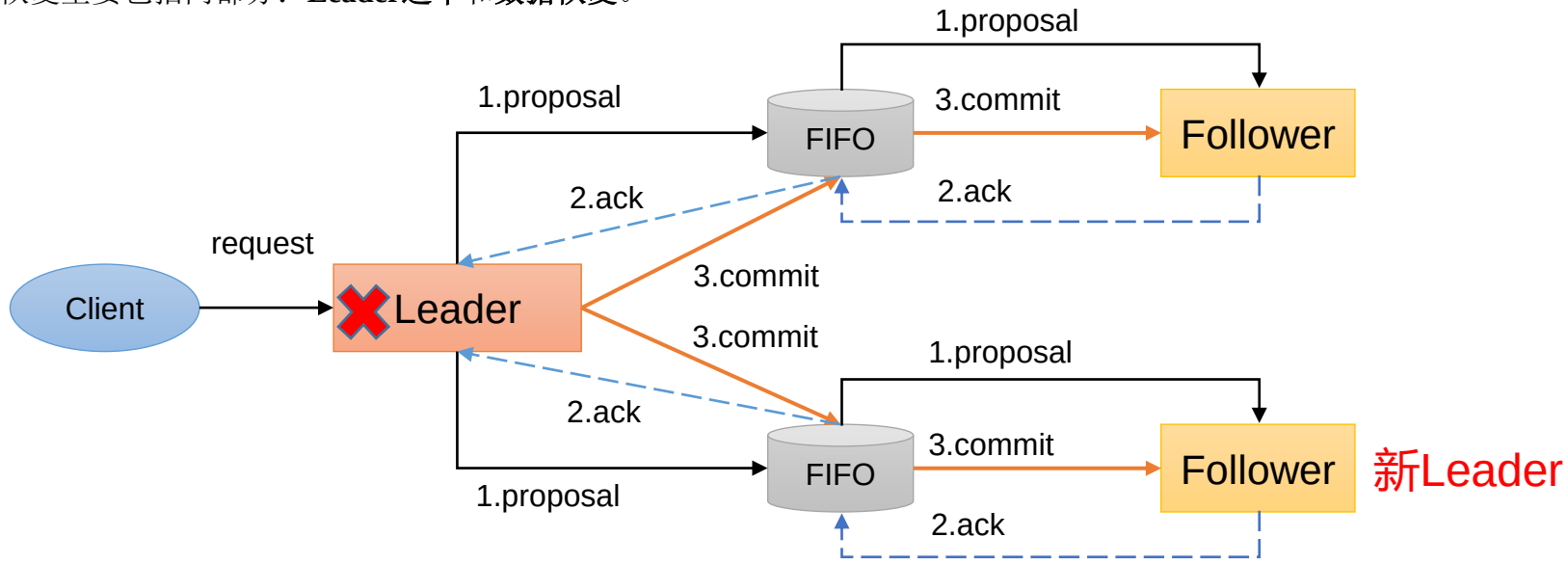


Leader选举：根据上述要求，Zab协议需要保证选举出来的Leader需要满足以下条件：

（1）新选举出来的Leader不能包含未提交的Proposal。**即新Leader必须都是已经提交了Proposal的Follower服务器节点。**

（2）**新选举的Leader节点中含有最大的zxid**。这样做的好处是可以避免Leader服务器检查Proposal的提交和丢弃工作。

崩溃恢复主要包括两部分：**Leader选举**和**数据恢复**。



Zab如何数据同步：

（1）完成Leader选举后，在正式开始工作之前（接收事务请求，然后提出新的Proposal），**Leader服务器会首先确认事务日志中的所有的Proposal 是否已经被集群中过半的服务器Commit。**

（2）Leader服务器需要确保所有的Follower服务器能够接收到每一条事务的Proposal，并且能将所有已经提交的事务Proposal应用到内存数据中。**等到Follower将所有尚未同步的事务Proposal都从Leader服务器上同步过，并且应用到内存数据中以后，**Leader才会把该Follower加入到真正可用的Follower列表中。

Zab数据同步过程中，如何处理需要丢弃的Proposal？

在Zab的事务编号zxid设计中，zxid是一个64位的数字。其中低32位可以看成一个简单的单增计数器，针对客户端每一个事务请求，Leader在产生新的Proposal事务时，都会对该计数器加1。而高32位则代表了Leader周期的epoch编号。

epoch编号可以理解为当前集群所处的年代，或者周期。每次Leader变更之后都会在 epoch的基础上加1，这样旧的Leader崩溃恢复之后，其他Follower也不会听它的了，因为 Follower只服从epoch最高的Leader命令。

每当选举产生一个新的 Leader，就会从这个Leader服务器上取出本地事务日志充最大编号Proposal的zxid，并从zxid中解析得到对应的epoch编号，然后再对其加1，之后该编号就作为新的epoch 值，并将低32位数字归零，由0开始重新生成zxid。

Zab协议通过epoch编号来区分Leader变化周期，能够有效避免不同的Leader错误的使用了相同的zxid编号提出了不一样的Proposal的异常情况。

基于以上策略，当一个包含了上一个Leader周期中尚未提交过的事务Proposal的服务器启动时，当这台机器加入集群中，以Follower角色连上Leader服务器后，Leader 服务器会根据自己服务器上最后提交的 Proposal来和Follower服务器的Proposal进行比对，比对的结果肯定是Leader要求Follower进行一个回退操作，回退到一个确实已经被集群中过半机器Commit的最新Proposal。

CAP理论告诉我们，一个分布式系统不可能同时满足以下三种

- 一致性（C:Consistency）
- 可用性（A:Available）
- 分区容错性（P:Partition Tolerance）

这三个基本需求，最多只能同时满足其中的两项，因为P是必须的，因此往往选择就在CP或者AP中。

1）一致性（C:Consistency）

在分布式环境中，一致性是指数据在多个副本之间是否能够保持数据一致的特性。在一致性的需求下，当一个系统在数据一致的状态下执行更新操作后，应该保证系统的数据仍然处于一致的状态。

2）可用性（A:Available）

可用性是指系统提供的服务必须一直处于可用的状态，对于用户的每一个操作请求总是能够在有限的时间内返回结果。

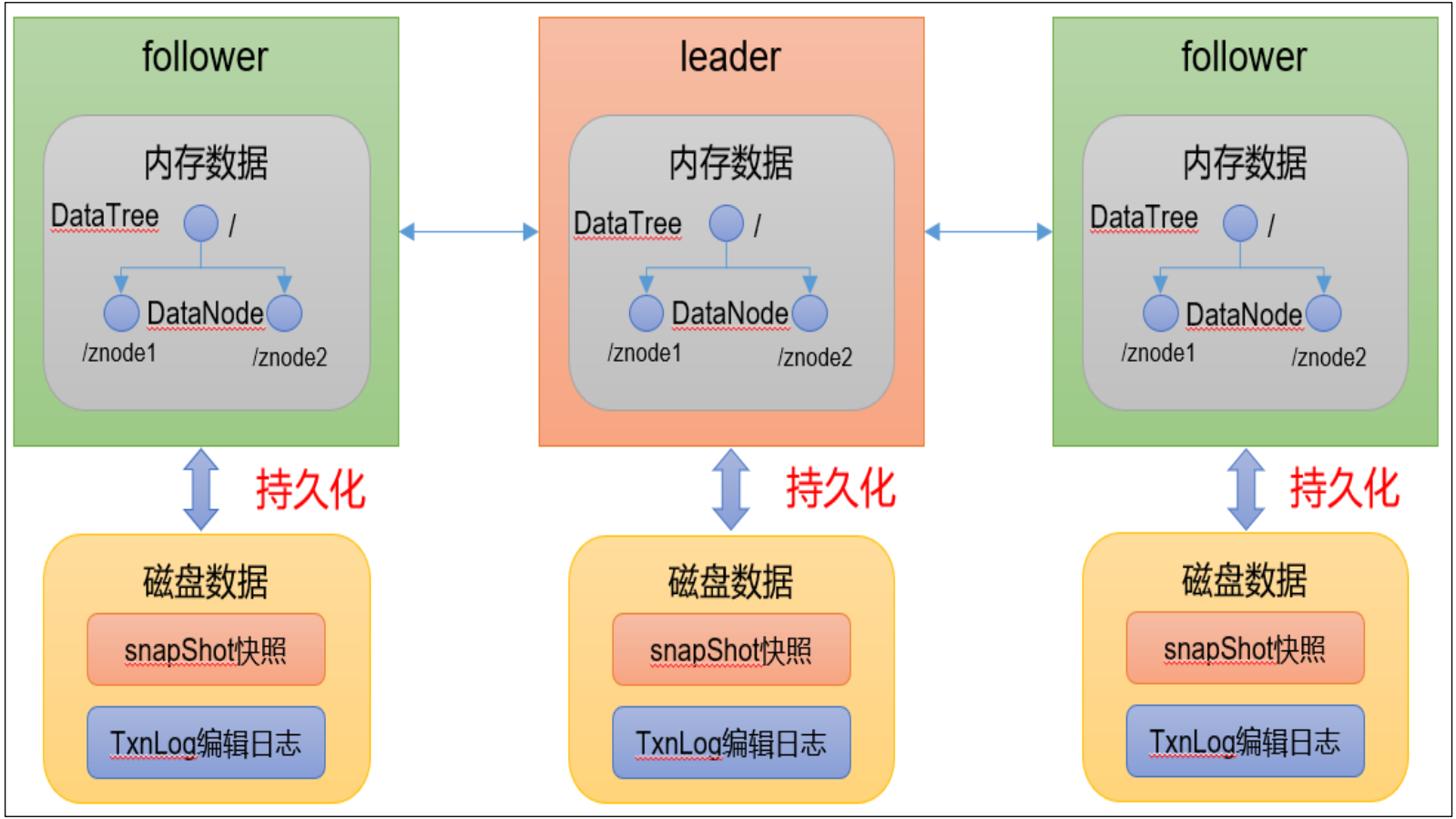
3）分区容错性（P:Partition Tolerance）

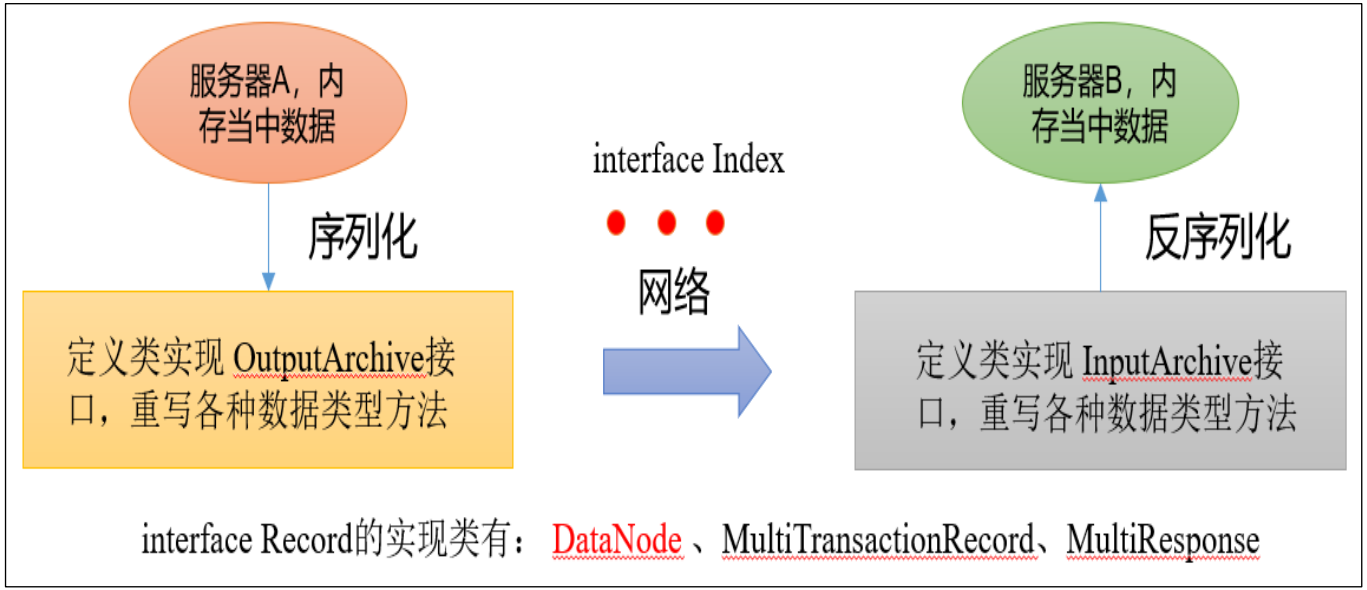
分布式系统在遇到任何网络分区故障的时候，仍然需要能够保证对外提供满足一致性和可用性的服务，除非是整个网络环境都发生了故障。

ZooKeeper保证的是CP

（1）ZooKeeper不能保证每次服务请求的可用性。（注：在极端环境下，ZooKeeper可能会丢弃一些请求，消费者程序需要重新请求才能获得结果）。所以说，ZooKeeper不能保证服务可用性。

（2）进行Leader选举时集群都是不可用。


follower
内存数据
DataTree
/
DataNode
/znode1
/znode2
leader
内存数据
DataTree
/
DataNode
/znode1
/znode2
follower
内存数据
DataTree
/
DataNode
/znode1
/znode2
持久化
持久化
持久化
磁盘数据
snapShot快照
TxnLog编辑日志
磁盘数据
snapShot快照
TxnLog编辑日志
磁盘数据
snapShot快照
TxnLog编辑日志


服务器A，内存当中数据
序列化
定义类实现 OutputArchive接口，重写各种数据类型方法
interface Index
网络
服务器B，内存当中数据
反序列化
定义类实现 InputArchive接口，重写各种数据类型方法
interface Record的实现类有： DataNode 、MultiTransactionRecord、MultiResponse

ZK服务端初始化源码解析

zkServer.sh start → nohup "$JAVA"
+ 一堆提交参数
+$ZOOMAIN （org.apache.zookeeper.server.quorum.QuorumPeerMain）
+ "$ZOOCFG" （zkEnv.sh文件中ZOOCFG="zoo.cfg"）

所以程序的入口
QuorumPeerMain.java

main()

// 1 服务端启动入口

- new QuorumPeerMain()
- initializeAndRun

// 解析参数

parse → parseProperties 解析zoo.cfg → setupQuorumPeerConfig → setupMyId 解析myid

// 过期快照删除

new DatadirCleanupManager → getSnapRetainCount()=3最少保留3个快照 → getPurgeInterval()=0关闭清除功能 → new PurgeTask 清理过期数据

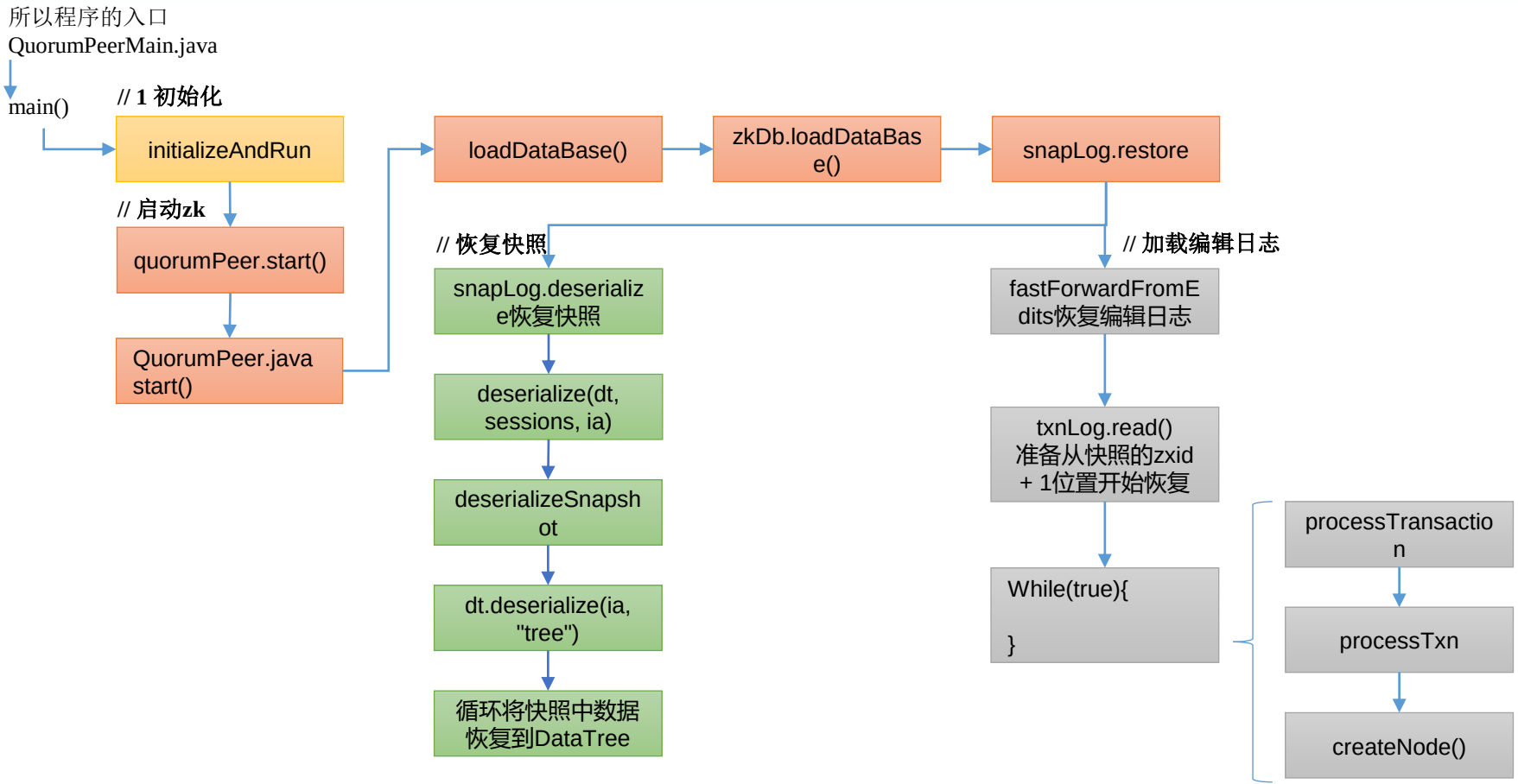
// 通信初始化

runFromConfig → createFactory → zookeeper.serverCnxnFactory → Default is `NIOServerCnxnFactory` 默认是NIO通信

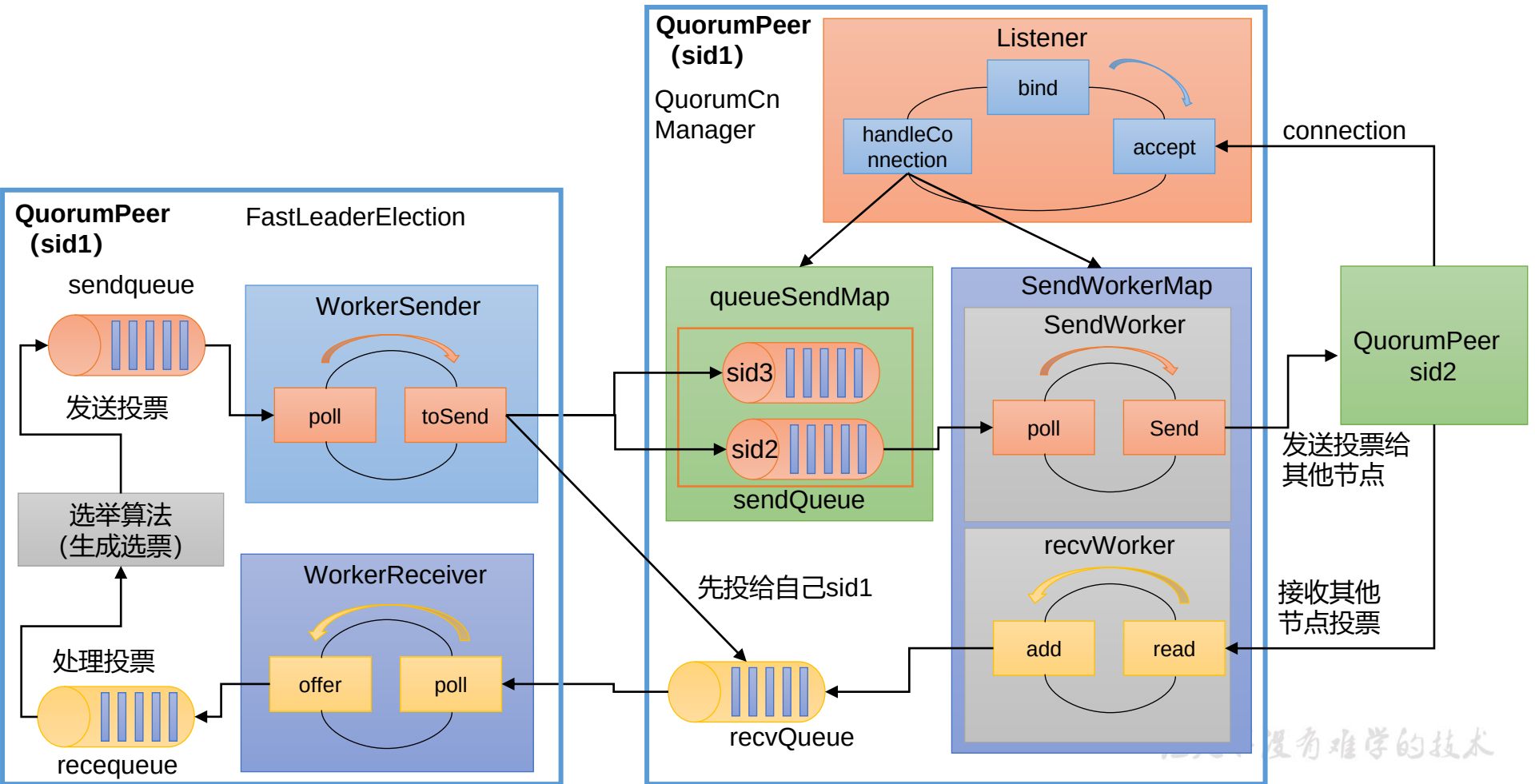
// 启动zk

- quorumPeer.start()
- configure → 初始化NIO服务端socket，绑定2181端口

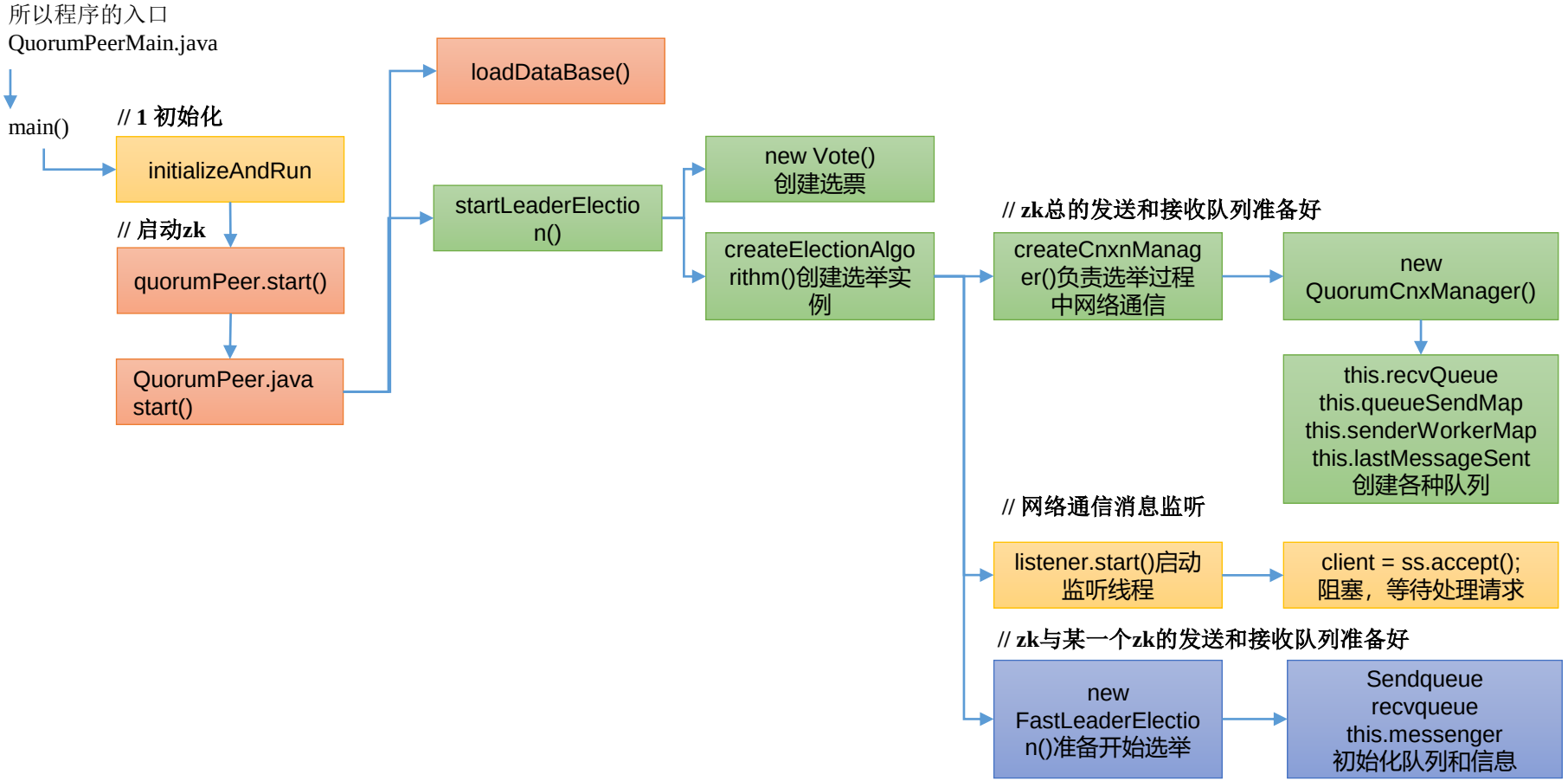
ZK服务端初始化源码解析



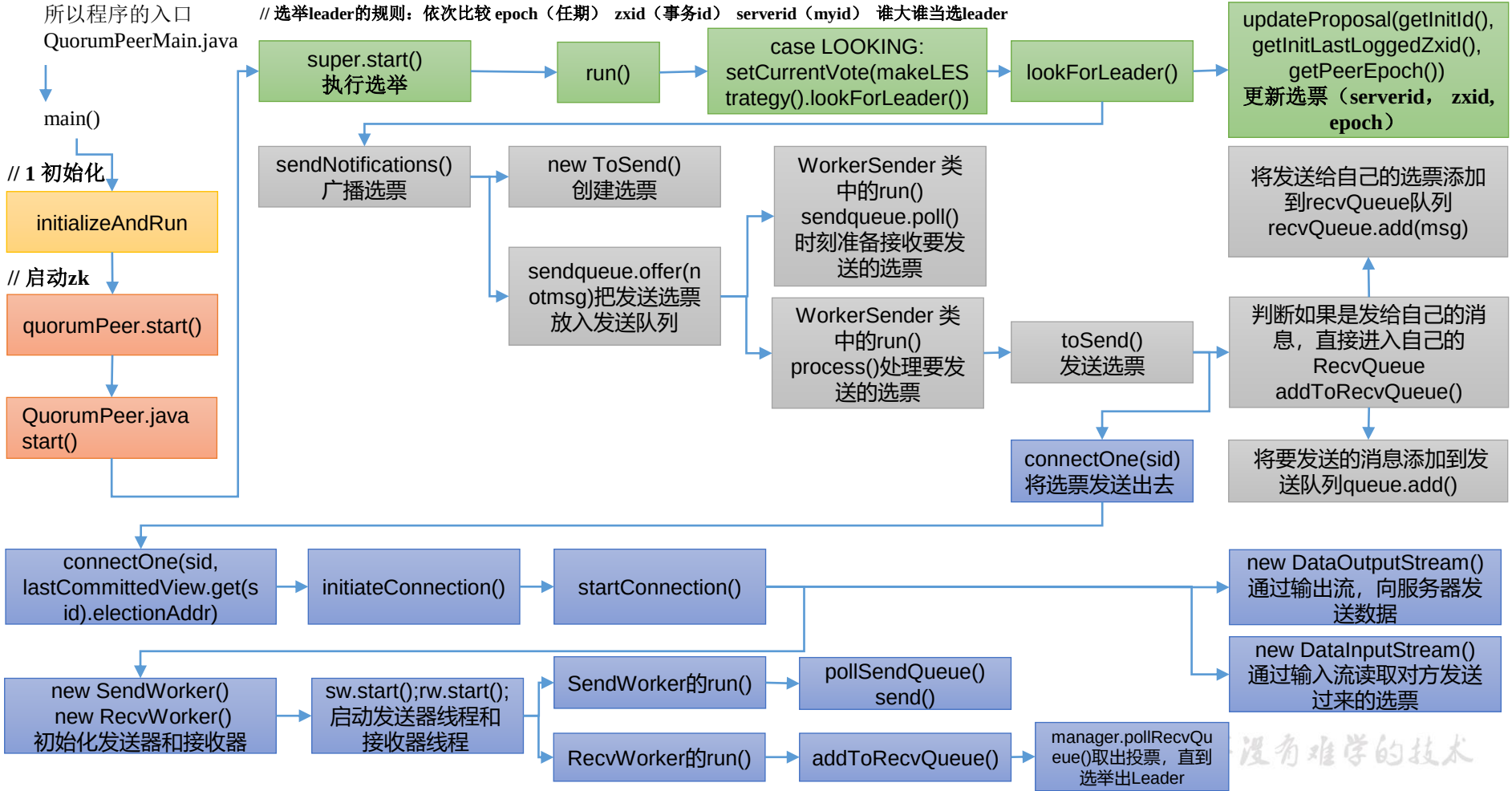
ZK选举源码解析


QuorumPeer
(sid1)
FastLeaderElection
sendqueue
发送投票
选举算法
(生成选票)
处理投票
recequeue
WorkerSender
poll
toSend
WorkerReceiver
offer
poll
QuorumPeer
(sid1)
QuorumCn
Manager
Listener
bind
handleCo
nnection
accept
connection
queueSendMap
sid3
sid2
sendQueue
SendWorkerMap
SendWorker
poll
Send
recvWorker
add
read
QuorumPeer
sid2
发送投票给
其他节点
接收其他
节点投票
先投给自己sid1
recvQueue

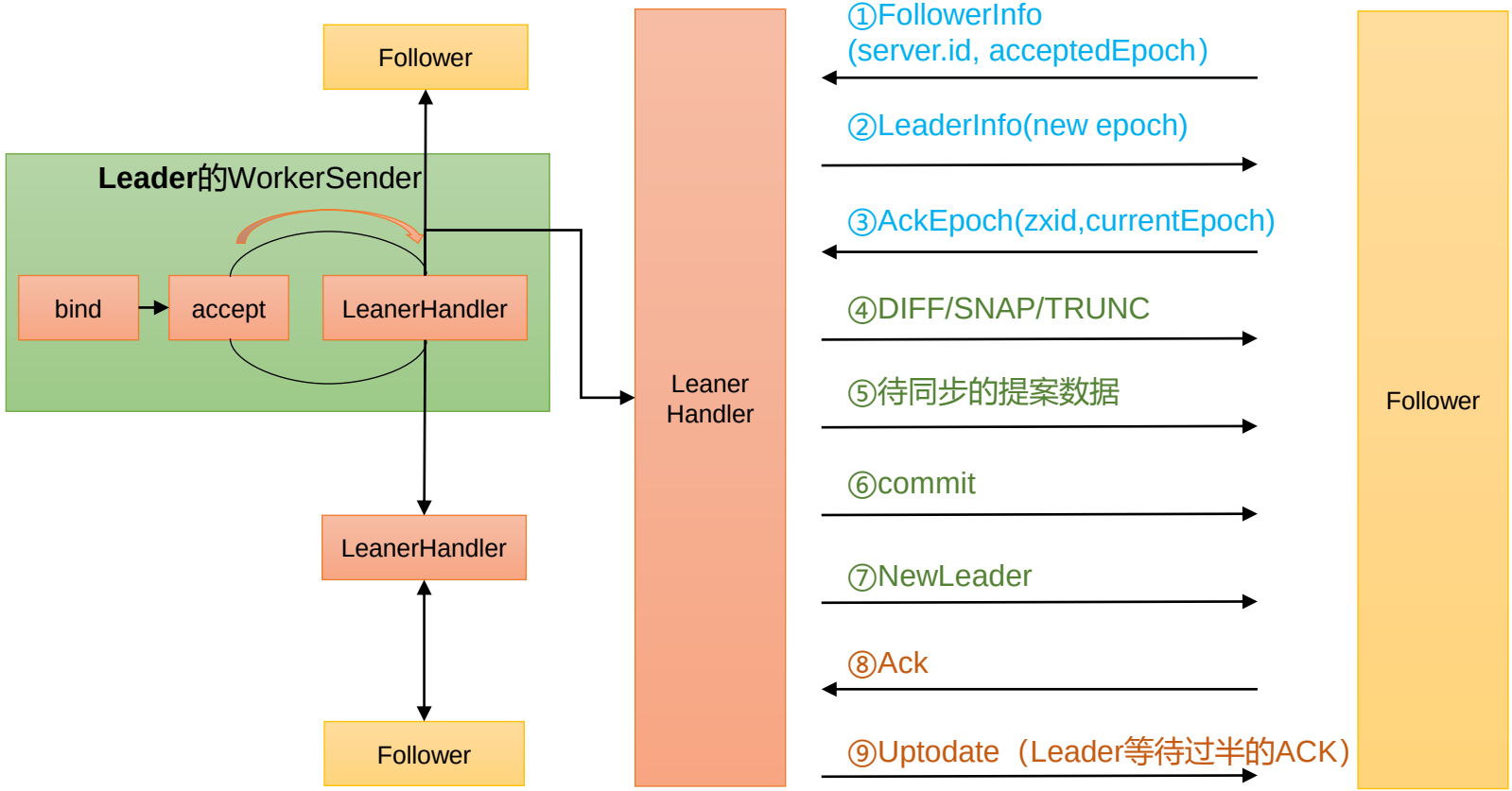
ZK选举准备源码解析



ZK选举执行源码解析


所以程序的入口
QuorumPeerMain.java
main()
// 1 初始化
initializeAndRun
// 启动zk
quorumPeer.start()
QuorumPeer.java
start()
// 选举leader的规则：依次比较 epoch（任期） zxid（事务id） serverid（myid） 谁大谁当选leader
super.start()
执行选举
run()
case LOOKING:
setCurrentVote(makeLES
trategy().lookForLeader())
lookForLeader()
updateProposal(getInitId(),
getInitLastLoggedZxid(),
getPeerEpoch())
更新选票（serverid， zxid,
epoch）
sendNotifications()
广播选票
new ToSend()
创建选票
sendqueue.offer(n
otmsg)把发送选票
放入发送队列
WorkerSender 类
中的run()
sendqueue.poll()
时刻准备接收要发
送的选票
WorkerSender 类
中的run()
process()处理要发
送的选票
toSend()
发送选票
connectOne(sid)
将选票发送出去
将发送给自己的选票添加
到recvQueue队列
recvQueue.add(msg)
判断如果是发给自己的消
息，直接进入自己的
RecvQueue
addToRecvQueue()
将要发送的消息添加到发
送队列queue.add()
connectOne(sid,
lastCommittedView.get(s
id).electionAddr)
initiateConnection()
startConnection()
new DataOutputStream()
通过输出流，向服务器发
送数据
new DataInputStream()
通过输入流读取对方发送
过来的选票
new SendWorker()
new RecvWorker()
初始化发送器和接收器
sw.start();rw.start();
启动发送器线程和
接收器线程
SendWorker的run()
pollSendQueue()
send()
RecvWorker的run()
addToRecvQueue()
manager.pollRecvQu
eue()取出投票，直到
选举出Leader

Follower和Leader状态同步源码解析



（1）注册

（2）同步策略

DIFF(差异化同步)

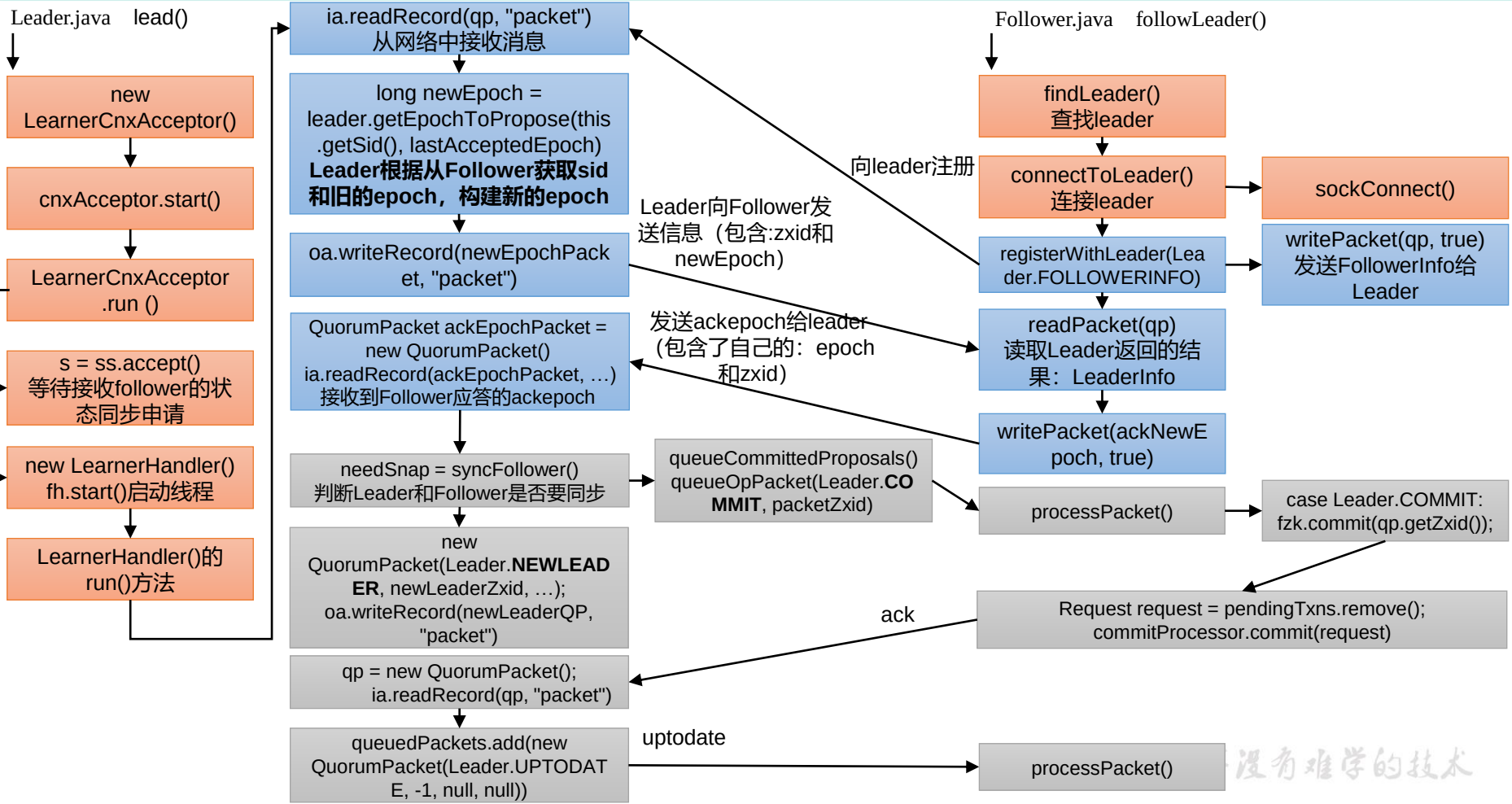
TRUNC(回滚同步)

SNAP(全量同步)

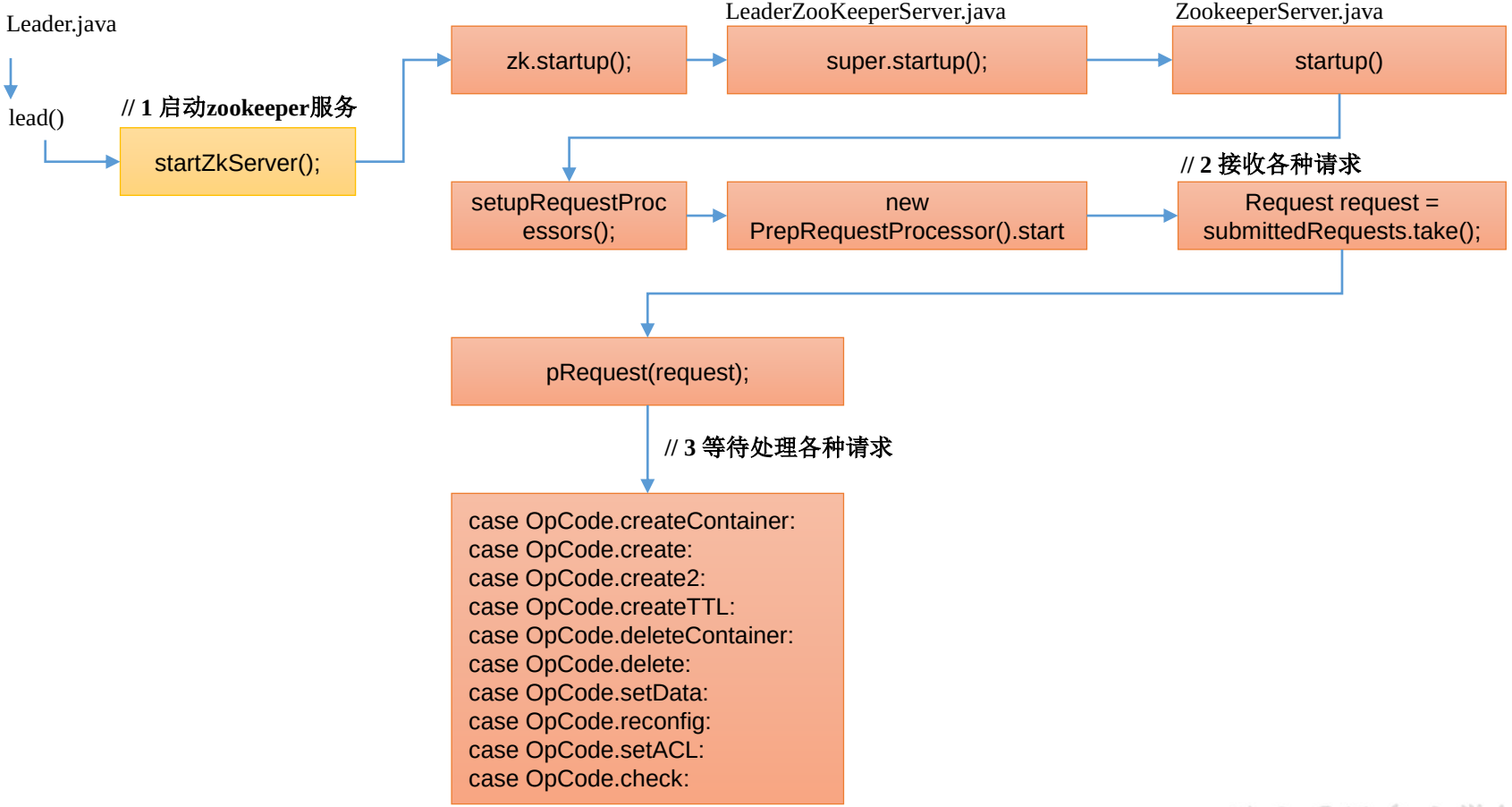
状态同步（同步过程中，有可能重复提议和提交）

（3）同步确认

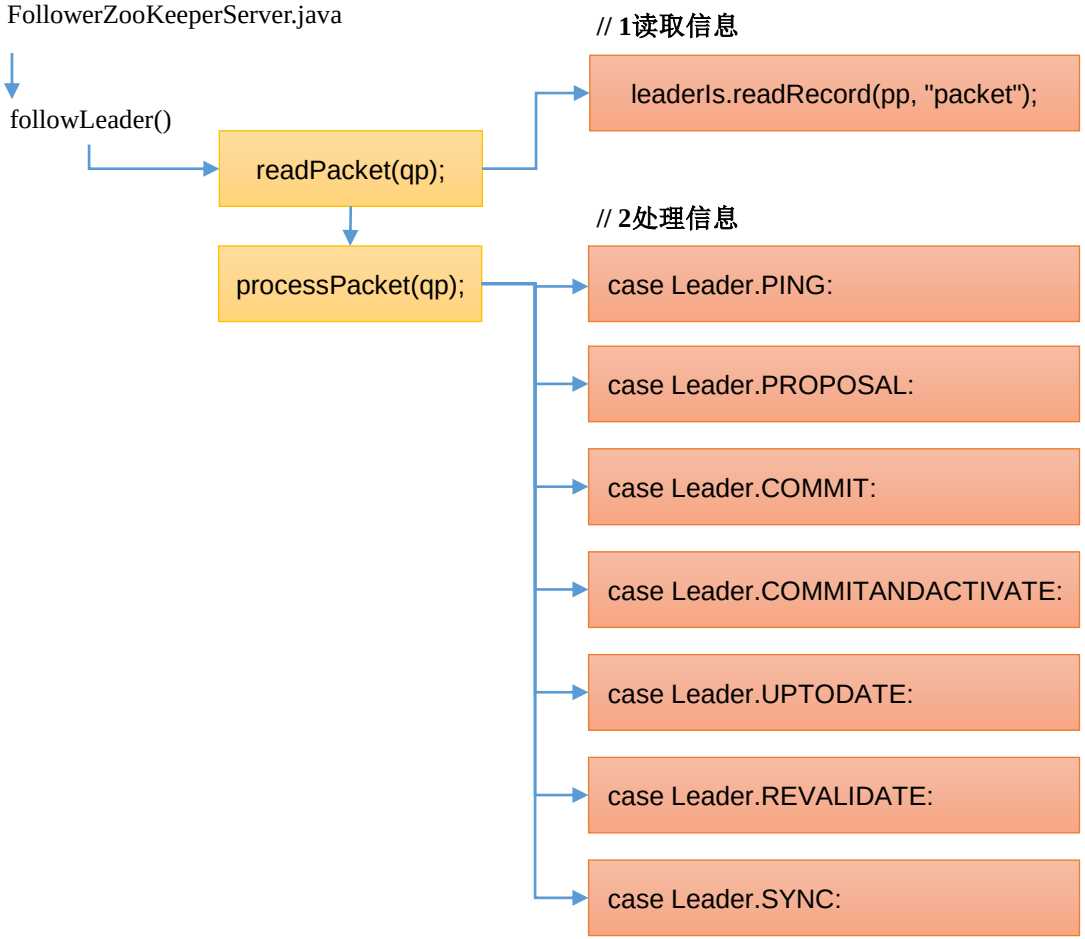
Follower和Leader状态同步源码解析



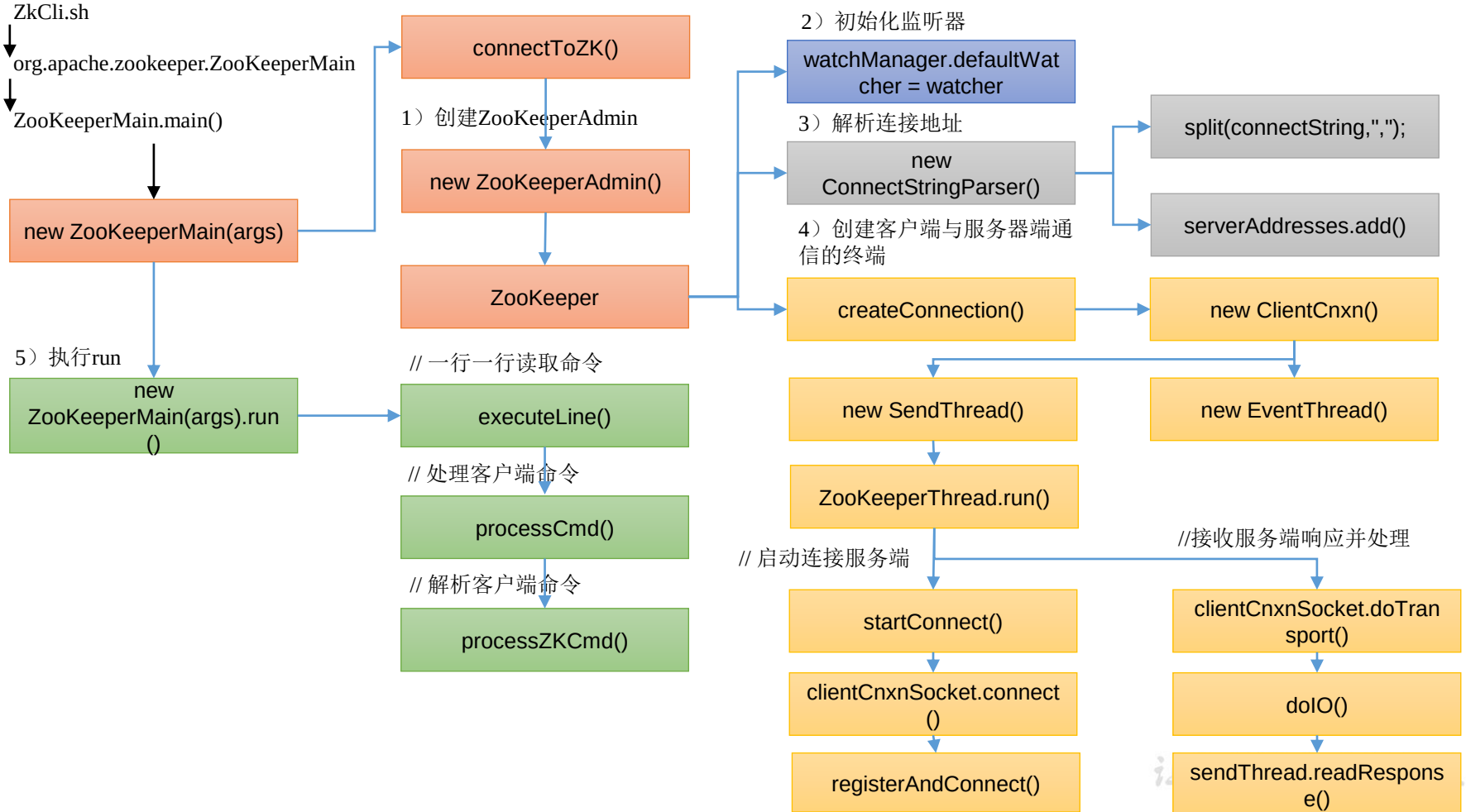
服务端Leader启动



服务端Follower启动



客户端初始化源码解析


ZkCli.sh
org.apache.zookeeper.ZooKeeperMain
ZooKeeperMain.main()
new ZooKeeperMain(args)
connectToZK()
1）创建ZooKeeperAdmin
new ZooKeeperAdmin()
ZooKeeper
2）初始化监听器
watchManager.defaultWatcher = watcher
3）解析连接地址
new ConnectStringParser()
split(connectString,",");
serverAddresses.add()
4）创建客户端与服务器端通信的终端
createConnection()
new ClientCnxn()
new SendThread()
new EventThread()
ZooKeeperThread.run()
// 启动连接服务端
//接收服务端响应并处理
startConnect()
clientCnxnSocket.connect()
registerAndConnect()
clientCnxnSocket.doTransport()
doIO()
sendThread.readResponse()
5）执行run
new ZooKeeperMain(args).run()
// 一行一行读取命令
executeLine()
// 处理客户端命令
processCmd()
// 解析客户端命令
processZKCmd()