

1. Zookeeper 综述

1.1. 课程总体介绍



如上图所示，整个 ZK 体系会从入门开始，到基础，进阶以及实战，最后把相关的理论给介绍一下。

ZK 在后面讲 dubbo 和 kafka 都会用到，请务必重视起来。

另外对 zk 了解比较深入的同学可以课后通过视频的方式加快学习进度。

1.1.1. 为什么学习 ZK

应该重点掌握分布式环境的演进过程，从一个单节点开始，慢慢过渡到分布式，为什么单节点不行，传统一个 tomcat 打天下有什么有点，缺点又是什么，当一个 tomcat 搞不定的时候，分布式的架构图又是什么样的，

传统的单节点架构自然有问题，到了分布式的架构中，问题肯定也有不少，这些问题就是我们学习 ZK 要解决的，但学习这些解决方案之前，还是需要有点理论基础。

接下来就要了解下什么是 zk,为什么学习 zk,Zk 在分布式架构中扮演了什么样的角色。以及面试的时候经常会问到的问题，心里要有个大概的了解。

1.1.2. zookeeper 基础

了解 zk 是什么玩意后，接下来就把 zk 安装好，先来讲解的是 zk 单机部署，这个非常容易，而且后面课程绝大部分的时间都是使用单节点来使用的，至于集群的配置放到后面来讲，正在工作中，集群的维护应该是运维来做，哪怕没有运维，在后面通过详细的学习通过也能在 20 分钟内完全搭建起来，同学们不用着急。

Zk 的特性：这块是重点之中的重点，后面学习的一切操作，包括实战，都是建立在这基础之上的。其中数据模型和 watch 机制又是最最重要的。

1.1.3. zk 进阶

安装好了 zk，对 zk 基础有一定的了解后，接下来就学习怎么操作 zk，首先了解基础 zk 客户端的使用。

客户端的简单使用能解决一些问题，方便查看信息，简单省事，但真正对于我们 java 架构师来说最重要的是 java 客户端，包括原生的，zkclient 以及 curator，同学们最少要熟练使用其中的一种，另外的在工作中也要很快的百度得出来。

1.1.4. zk 的实战

一般来说 zk 实战是放到最后来讲得，但这一块确是面试得时候经常会碰到的，所以先提前，这一块是整个课程的重点，我会带同学一行行的代码来学习，也请同学务必能够跟着 deer 老师一个个案例吃透。

会讲到配置中心，负载均衡，分布式锁，集群选举，尤其是负载均衡，如果这块没听懂 peter 老师会找我麻烦的，后面他讲 dubbo 的核心就是这玩意，至于具体的请到后面课程中收听。

1.1.5. zk 高级知识

这一块不需要同学完全听懂，掌握 40%-50% 就够了，工作中很少用到，但确实面试的时候的加分项，

包括 2pc,3pc 提交协议，集群的安装等等

1.1.6. 总结：

不管怎么样，其中的 zk 基础，zk 进阶客户端的使用，以及 zk 项目实战请务必重视起来。

1.2. 分布式系统基础知识

一个 tomcat 打天下的时代，不能说完全淘汰了，在一个管理系统，小型项目中还经常使用，这并不过分，出于成本的考虑，这反而值得提倡。

但同学们最终是要成为架构师，架构师就必然要了解分布式系统：

1.2.1. 分布式系统是什么

分布式系统：一个硬件或软件组件分布在不同的网络计算机上，彼此之间仅仅通过消息传递进行通信和协调的系统

这是分布式系统，在不同的硬件，不同的软件，不同的网络，不同的计算机上，仅仅通过消息来进行通讯与协调

这是他的特点，更细致的看这些特点又可以有：分布性、对等性、并发性、缺乏全局时钟、故障随时会发生。

1.2.1.1. 分布性

既然是分布式系统，最显著的特点肯定就是分布性，从简单来看，如果我们做的是个电商项目，整个项目会分成不同的功能，专业点就不同的微服务，比如用户微服务，产品微服务，订单微服务，这些服务部署在不同的 tomcat 中，不同的服务器中，甚至不同的集群中，整个架构都是分布在不同的地方的，在空间上是随意的，而且随时会增加，删除服务器节点，这是第一个特性

1.2.1.2. 对等性

对等性是分布式设计的一个目标，还是以电商网站为例，来说明下什么是对等性，要完成一个分布式的系统架构，肯定不是简单的把一个大的单一系统拆分成一个个微服务，然后部署在不同的服务器集群就够了，其中拆分完成的每一个微服务都有可能发现问题，而导致整个电商网站出现功能的丢失。

比如订单服务，为了防止订单服务出现问题，一般情况需要有一个备份，在订单服务出现问题的时候能顶替原来的订单服务。

这就要求这两个（或者 2 个以上）订单服务完全是对等的，功能完全是一致的，其实这就是一种服务副本的冗余。

还一种是数据副本的冗余，比如数据库，缓存等，都和上面说的订单服务一样，为了安全考虑需要有完全一样的备份存在，这就是对等性的意思。

1.2.1.3. 并发性

并发性其实对我们来说并不陌生，在学习多线程的时候已经或多或少学习过，多线程是并发的基础。

但现在我们要接触的不是多线程的角度，而是更高一层，从多进程，多JVM的角度，例如在一个分布式系统中的多个节点，可能会并发地操作一些共享资源，如何准确并高效的协调分布式并发操作。

后面实战部分的分布式锁其实就是解决这问题的。

1.2.1.4. 缺乏全局时钟

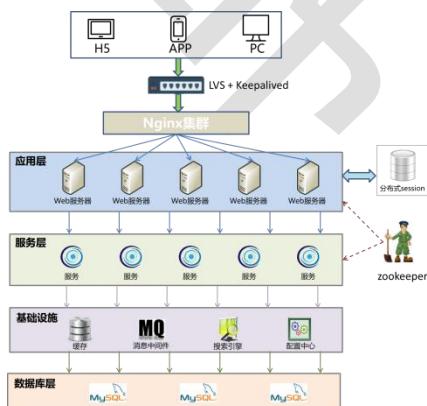
在分布式系统中，节点是可能反正任意位置的，而每个位置，每个节点都有自己的时间系统，因此在分布式系统中，很难定义两个事务纠结谁先谁后，原因就是因为缺乏一个全局的时钟序列进行控制，当然，现在这已经不是什么大问题了，已经有大把的时间服务器给系统调用

1.2.1.5. 故障随时会发生

任何一个节点都可能出现停电，死机等现象，服务器集群越多，出现故障的可能性就越大，随着集群数目的增加，出现故障甚至都会成为一种常态，怎么样保证在系统出现故障，而系统还是正常的访问者是作为系统架构师应该考虑的。

1.2.2. 大型网站架构图回顾

知道什么是分布式系统，接下来具体来看下大型网站架构图，这个图在前面分布式架构演进应该已经讲过，首先整个架构分成很多个层，应用层，服务层，基础设施层与数据服务层，每一层都由若干节点组成，这是典型的分布式架构，后面一大把的时间就是系统的学习里面的每一个部分



那么zookeeper在其中又是扮演什么角色呢，如果可以把zk扮演成交警的角色，而各个节点就是马路上的各种汽车（汽车，公交车），为了保证整个交通（系统）的可用性，zookeeper

必须知道每一节点的健康状态（公交车是否出了问题，要派新的公交【服务注册与发现】），公路在上下班高峰是否拥堵，在某一条很窄的路上只允许单独一个方向的汽车通过【分布式锁】。

如果交通警察是交通系统的指挥官，而 zookeeper 就是各个节点组成分布式系统的指挥官。

1.2.2.1. 分布式系统协调“方法论”

1.2.2.1.1. 分布式系统带来的问题

如果把分布式系统和平时的交通系统进行对比，哪怕再稳健的交通系统也会有交通事故，分布式系统也有很多需要攻克的问题，比如：通讯异常，网络分区，三态，节点故障等。

1.2.2.1.1.1. 通信异常

通讯异常其实就是网络异常，网络系统本身是不可靠的，由于分布式系统需要通过网络进行数据传输，网络光纤，路由器等硬件难免出现问题。只要网络出现问题，也就会影响消息的发送与接受过程，因此数据消息的丢失或者延长就会变得非常普遍。

1.2.2.1.1.2. 网络分区

网络分区，其实就是脑裂现象，本来有一个交通警察，来管理整个片区的交通情况，一切井然有序，突然出现了停电，或者出现地震等自然灾难，某些道路接受不到交通警察的指令，可能在这种情况下，会出现一个零时工，片警零时来指挥交通。

但注意，原来的交通警察其实还在，只是通讯系统中断了，这时候就会出现问题了，在同一个片区的道路上有不同人在指挥，这样必然引擎交通的阻塞混乱。

这种由于种种问题导致同一个区域（分布式集群）有两个相互冲突的负责人的时候就会出现这种精神分裂的情况，在这里称为脑裂，也叫网络分区。

1.2.2.1.1.3. 三态

三态是什么？三态其实就是成功，与失败以外的第三种状态，当然，肯定不叫变态，而叫超时态。

在一个 jvm 中，应用程序调用一个方法函数后会得到一个明确的相应，要么成功，要么失败，而在分布式系统中，虽然绝大多数情况下能够接受到成功或者失败的相应，但一旦网络出现异常，就非常有可能出现超时，当出现这样的超时现象，网络通讯的发起方，是无法确定请求是否成功处理的。

1.2.2.1.4. 节点故障

这个其实前面已经说过了，节点故障在分布式系统下是比较常见的问题，指的是组成服务器集群的节点会出现的宕机或“僵死”的现象，这种现象经常会发生。

1.2.2.1.2. CAP 理论

前面花费了很大的篇幅来了解分布式的特点以及会碰到很多会让人头疼的问题，这些问题肯定会有一定的理论思想来解决问题的。

接下来花点时间来谈谈这些理论，其中 CAP 和 BASE 理论是基础，也是面试的时候经常会问到的

首先看下 CAP，CAP 其实就是一致性，可用性，分区容错性这三个词的缩写

1.2.2.1.2.1. 一致性

一致性是事务 ACID 的一个特性【原子性(Atomicity)、一致性(Consistency)、隔离性(Isolation)、持久性(Durability)】，学习数据库优化的时候 deer 老师讲过。

这里讲的一致性其实大同小异，只是现在考虑的是分布式环境中，还是不单一的数据库。

在分布式系统中，一致性是数据在多个副本之间是否能够保证一致的特性，这里说的一致性和前面说的对等性其实差不多。如果能够在分布式系统中针对某一个数据项的变更成功执行后，所有用户都可以马上读取到最新的值，那么这样的系统就被认为具有【强一致性】。

1.2.2.1.2.2. 可用性

可用性指系统提供服务必须一直处于可用状态，对于用户的操作请求总是能够在有限的时间内访问结果。

这里的重点是【有限的时间】和【返回结果】

为了做到有限的时间需要用到缓存，需要用到负载，这个时候服务器增加的节点是为性能考虑；

为了返回结果，需要考虑服务器主备，当主节点出现问题的时候需要备份的节点能最快的顶替上来，千万不能出现 OutOfMemory 或者其他 500, 404 错误，否则这样的系统我们会认为是不可用的。

1.2.2.1.2.3. 分区容错性

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。

不能出现脑裂的情况

1.2.2.1.2.4. 具体描述

来看下 CAP 理论具体描述：

一个分布式系统不可能同时满足一致性、可用性和分区容错性这三个基本需求，最多只能同时满足其中的两项

序号	被抛弃的谁	说明
1	放弃P (满足AC)	将数据和服务都放在一个节点上，避免因网络引起的负面影响，充分保证系统的可用性和一致性。但放弃P意味着放弃了系统的可扩展性
2	放弃A (满足PC)	当节点故障或者网络故障时，受到影响的服务需要等待一定的世界，因此在等待时间里，系统无法对外提供正常服务，因此是不可用的；
3	放弃C (满足AP)	系统无法保证数据的实时一致性，但是承诺数据最终会保证一致性。因此存在数据不一致的窗口期，至于窗口期的长短取决于系统的设计

TIPS：不可能把所有应用全部放到一个节点上，因此架构师的精力往往就花在怎么样根据业务场景在 A 和 C 直接寻求平衡；

1.2.2.1.3. BASE 理论

根据前面的 CAP 理论，架构师应该从一致性和可用性之间找平衡，系统短时间完全不可用肯定是不允许的，那么根据 CAP 理论，在分布式环境下必然也无法做到强一致性。

BASE 理论：即使无法做到强一致性，但分布式系统可以根据自己的业务特点，采用适当的方式来使系统达到最终的一致性；

1.2.2.1.3.1. Basically Available 基本可用

当分布式系统出现不可预见的故障时，允许损失部分可用性，保障系统的“基本可用”；体现在“时间上的损失”和“功能上的损失”；

e.g：部分用户双十一高峰期淘宝页面卡顿或降级处理；

1.2.2.1.3.2. Soft state 软状态

其实就是前面讲到的三态，既允许系统中的数据存在中间状态，既系统的不同节点的数据副本之间的数据同步过程存在延时，并认为这种延时不会影响系统可用性；

e.g: 12306 网站卖火车票，请求会进入排队队列；

1.2.2.1.3.3. Eventually consistent 最终一致性

所有的数据在经过一段时间的数据同步后，最终能够达到一个一致的状态；

e.g: 理财产品首页充值总金额短时不一致；

1.3. Zookeeper 简介

1.3.1. Zookeeper 简介 (what)

ZooKeeper 致力于提供一个高性能、高可用，且具备严格的顺序访问控制能力的分布式协调服务，是雅虎公司创建，是 Google 的 Chubby 一个开源的实现，也是 Hadoop 和 Hbase 的重要组件。

1.3.1.1. 设计目标

- 简单的数据结构：共享的树形结构，类似文件系统，存储于内存；
- 可以构建集群：避免单点故障，3-5 台机器就可以组成集群，超过半数正常工作就能对外提供服务；
- 顺序访问：对于每个读请求，zk 会分配一个全局唯一的递增编号，利用这个特性可以实现高级协调服务；
- 高性能：基于内存操作，服务于非事务请求，适用于读操作为主的业务场景。3 台 zk 集群能达到 13w QPS；

1.3.2. 哪些常见需要用到 ZK (why)

数据发布订阅
负载均衡
命名服务
Master 选举
集群管理
配置管理
分布式队列
分布式锁

1.3.3. 为什么要学习 zookeeper? (why)

互联网架构师必备技能

高端岗位必考察的知识点

zk 面试问题全解析

- Zookeeper 是什么框架
- 应用场景
- Paxos 算法& Zookeeper 使用协议
- 选举算法和流程
- Zookeeper 有哪几种节点类型
- Zookeeper 对节点的 watch 监听通知是永久的吗?
- 部署方式? 集群中的机器角色都有哪些? 集群最少要几台机器
- 集群如果有 3 台机器, 挂掉一台集群还能工作吗? 挂掉两台呢?
- 集群支持动态添加机器吗?

2. Zookeeper 基础

2.1. 部署

先把 ZK 安装起来, 后面的很多操作, 都是的前提都是由 ZK 的操作环境, 先来把 ZK 安装好,

2.1.1. Zookeeper windows 环境安装

环境要求: 必须要有 jdk 环境, 本次讲课使用 jdk1.8

1. 安装 jdk
2. 安装 Zookeeper. 在官网 <http://zookeeper.apache.org/> 下载 zookeeper. 我下载的是 zookeeper-3.4.12 版本。

解压 zookeeper-3.4.6 至 D:\machine\zookeeper-3.4.12.

在 D:\machine 新建 data 及 log 目录。

3. ZooKeeper 的安装模式分为三种, 分别为: 单机模式 (stand-alone)、集群模式和集群伪分布模式。ZooKeeper 单机模式的安装相对比较简单, 如果第一次接触 ZooKeeper 的话, 建议安装 ZooKeeper 单机模式或者集群伪分布模式。

安装单击模式。至 D:\machine\zookeeper-3.4.12\conf 复制 zoo_sample.cfg 并粘贴到当前目录下, 命名 zoo.cfg.

2.1.2. Zookeeper 集群配置

1. 安装 jdk 运行 jdk 环境

上传 jdk1.8 安装包

2. 安装 jdk1.8 环境变量

```
vi /etc/profile
```

```
export JAVA_HOME=/usr/local/jdk1.8.0_181
export ZOOKEEPER_HOME=/usr/local/zookeeper
export CLASSPATH=.:${JAVA_HOME}/lib/dt.jar:${JAVA_HOME}/lib/tools.jar
export PATH=${JAVA_HOME}/bin:${ZOOKEEPER_HOME}/bin:$PATH
```

刷新 profile 文件

```
source /etc/profile
```

关闭防火墙

3. 下载 zookeeper 安装包

```
wget https://mirrors.tuna.tsinghua.edu.cn/apache/zookeeper/zookeeper-3.4.10/zookeeper-3.4.10.tar.gz
```

4. 解压 Zookeeper 安装包

```
tar -zxvf zookeeper-3.4.10.tar.gz
```

5. 修改 Zookeeper 文件夹名称

```
重命名: mv zookeeper-3.4.10 zookeeper
```

6. 修改 zoo_sample.cfg 文件

```
cd /usr/local/zookeeper/conf
```

```
mv zoo_sample.cfg zoo.cfg
```

修改 conf: vi zoo.cfg 修改两处

(1) dataDir=/usr/local/zookeeper/data (注意同时在 zookeeper 创建 data 目录)

(2) 最后面添加

```
server.0=192.168.212.154:2888:3888
```

```
server.1=192.168.212.156:2888:3888
```

```
server.2=192.168.212.157:2888:3888
```

7. 创建服务器标识

服务器标识配置:

创建文件夹: mkdir data

创建文件 myid 并填写内容为 0: vi

`myid` (内容为服务器标识 : 0)

8. 复制 zookeeper

进行复制 `zookeeper` 目录到 `node1` 和 `node2`

还有 `/etc/profile` 文件

把 `node1`、`node2` 中的 `myid` 文件里的值修改为 1 和 2

路径(`vi /usr/local/zookeeper/data/myid`)

9 启动 zookeeper

启动 `zookeeper`:

路径: `/usr/local/zookeeper/bin`

执行: `zkServer.sh start`

(注意这里 3 台机器都要进行启动)

状态: `zkServer.sh`

`status`(在三个节点上检验 zk 的 mode, 一个 leader 和两个 follower)

```
scp -r /soft root@zk2:/
```

```
scp -r /soft root@zk3:/
```

2.1.3. 目录结构

<code>bin</code>	存放系统脚本
<code>conf</code>	存放配置文件
<code>contrib</code>	zk 附加功能支持
<code>dist-maven</code>	maven 仓库文件
<code>docs</code>	zk 文档
<code>lib</code>	依赖的第三方库
<code>recipes</code>	经典场景样例代码
<code>src</code>	zk 源码

其中 `bin` 和 `conf` 是非常重要的两个目录, 平时也是经常使用的。

2.1.3.1. `bin` 目录

先看下 `bin` 目录

	README.txt	2018/3/27 12:32	文本文档 1 KB
	zkCleanup.sh	2018/3/27 12:32	Shell Script 2 KB
	zkCli.cmd	2018/3/27 12:32	Windows 命令脚本 2 KB
	zkCli.sh	2018/3/27 12:32	Shell Script 2 KB
	zkEnv.cmd	2018/3/27 12:32	Windows 命令脚本 2 KB
	zkEnv.sh	2018/3/27 12:32	Shell Script 3 KB
	zkServer.cmd	2018/3/27 12:32	Windows 命令脚本 2 KB
	zkServer.sh	2018/3/27 12:32	Shell Script 7 KB

其中

zkServer 为服务器，启动后默认端口为 2181

zkCli 为命令行客户端

2.1.3.2. conf 目录

Conf 目录为配置文件存放的目录，zoo.cfg 为核心的配置文件

这里面的配置很多，这配置是运维的工作，目前没必要，也没办法全部掌握。

序号	参数名	说明
1	clientPort	客户端连接server的端口，即对外服务端口，一般设置为2181吧。
2	dataDir	存储快照文件snapshot的目录。默认情况下，事务日志也会存储在这里。建议同时配置参数dataLogDir，事务日志的写性能直接影响zk性能。
3	tickTime	ZK中的一个时间单元。ZK中所有时间都是以这个时间单元为基础，进行整数倍配置的。例如，session的最小超时时间是2*tickTime。
4	dataLogDir	事务日志输出目录。尽量给事务日志的输出配置单独的磁盘或是挂载点，这将极大的提升ZK性能。（No Java system property）
5	globalOutstandingLimit	最大请求堆积数。默认是1000。ZK运行的时候，尽管server已经没有空闲来处理更多的客户端请求了，但是还是允许客户端将请求提交到服务器上来，以提高吞吐性能。当然，为了防止Server内存溢出，这个请求堆积数还是需要限制下的。（Java system property: zookeeper.globalOutstandingLimit.）
6	preAllocSize	预先开辟磁盘空间，用于后续写入事务日志。默认是64M，每个事务日志大小就是64M。如果ZK的快照频率较大的话，建议适当减小这个参数。（Java system property: zookeeper.preAllocSize）

序号	参数名	说明
7	snapCount	每进行snapCount次事务日志输出后，触发一次快照(snapshot)，此时，ZK会生成一个snapshot.*文件，同时创建一个新的事务日志文件log.*。默认是100000. (真正的代码实现中，会进行一定的随机数处理，以避免所有服务器在同一时间进行快照而影响性能) (Java system property: zookeeper.snapCount)
8	traceFile	用于记录所有请求的log，一般调试过程中可以使用，但是生产环境不建议使用，会严重影响性能。 (Java system property: requestTraceFile)
9	maxClientCnxns	单个客户端与单台服务器之间的连接数的限制，是ip级别的，默认是60，如果设置为0，那么表明不作任何限制。请注意这个限制的使用范围，仅仅是单台客户端机器与单台ZK服务器之间的连接数限制，不是针对指定客户端IP，也不是ZK集群的连接数限制，也不是单台ZK对所有客户端的连接数限制。
10	clientPortAddress	对于多网卡的机器，可以为每个IP指定不同的监听端口。默认情况是所有IP都监听 clientPort指定的端口。 New in 3.3.0
11	minSessionTimeout maxSessionTimeout	Session超时时间限制，如果客户端设置的超时时间不在这个范围，那么会被强制设置为最大或最小时间。默认的Session超时时间是在 $2 * tickTime \sim 20 * tickTime$ 这个范围 New in 3.3.0

序号	参数名	说明
12	fsync.warningthresholdms	事务日志输出时，如果调用fsync方法超过指定的超时时间，那么会在日志中输出警告信息。默认是1000ms。 (Java system property: fsync.warningthresholdms) New in 3.3.4
13	autopurge.purgeInterval	在上文中已经提到，3.4.0及之后版本，ZK提供了自动清理事务日志和快照文件的功能，这个参数指定了清理频率，单位是小时，需要配置一个1或更大的整数，默认是0，表示不开启自动清理功能。 (No Java system property) New in 3.4.0
14	autopurge.snapRetainCount	这个参数和上面的参数搭配使用，这个参数指定了需要保留的文件数目。默认是保留3个。 (No Java system property) New in 3.4.0
15	electionAlg	在之前的版本中，这个参数配置是允许我们选择leader选举算法，但是由于在以后的版本中，只会留下一种“TCP-based version of fast leader election”算法，所以这个参数目前看来没有用了，这里也不详细展开说了。 (No Java system property)
16	initLimit	Follower在启动过程中，会从Leader同步所有最新数据，然后确定自己能够对外服务的起始状态。Leader允许F在 initLimit时间内完成这个工作。通常情况下，我们不用太在意这个参数的设置。如果ZK集群的数据量确实很大了，F在启动的时候，从Leader上同步数据的时间也会相应变长，因此在这种情况下，有必要适当调大这个参数了。 (No Java system property)

序号	参数名	说明
17	syncLimit	在运行过程中, Leader负责与ZK集群中所有机器进行通信, 例如通过一些心跳检测机制, 来检测机器的存活状态。如果L发出心跳包在syncLimit之后, 还没有从F那里收到响应, 那么就认为这个F已经不在线了。注意: 不要把这个参数设置得过大, 否则可能会掩盖一些问题。(No Java system property)
18	leaderServes	默认情况下, Leader是会接受客户端连接, 并提供正常的读写服务。但是, 如果你想让Leader专注于集群中机器的协调, 那么可以将这个参数设置为no, 这样一来, 会大大提高写操作的性能。(Java system property: zookeeper.leaderServes)。
19	server.x=[hostname]:nnnnn[:nnnnn]	这里的x是一个数字, 与myid文件中的id是一致的。右边可以配置两个端口, 第一个端口用于F和L之间的数据同步和其它通信, 第二个端口用于Leader选举过程中投票通信。(No Java system property)
20	group.x=nnnnn[:nnnn]weight.x=nnn nn	对机器分组和权重设置, 可以参见这里(No Java system property)
21	cnxTimeout	Leader选举过程中, 打开一次连接的超时时间, 默认是5s。(Java system property: zookeeper.cnxTimeout)

序号	参数名	说明
22	zookeeper.DigestAuthenticationProvider.superDigest	ZK权限设置相关, 具体参见《使用super身份对有权限的节点进行操作》和《ZooKeeper权限控制》
23	skipACL	对所有客户端请求都不作ACL检查。如果之前节点上设置有权限限制, 一旦服务器上打开这个开头, 那么也将失效。(Java system property: zookeeper.skipACL)
24	forceSync	这个参数确定了是否需要在事务日志提交的时候调用FileChannel.force来保证数据完全同步到磁盘。(Java system property: zookeeper.forceSync)
25	jute.maxbuffer	每个节点最大数据量, 是默认是1M。这个限制必须在server和client端都进行设置才会生效。(Java system property: jute.maxbuffer)

在这挑选几个讲解:

clientPort: 参数无默认值, 必须配置, 用于配置当前服务器对外的服务端口, 客户端必须使用这端口才能进行连接

dataDir: 用于存放内存数据库快照的文件夹, 同时用于集群的 myid 文件也存在这个文件夹里(注意: 一个配置文件只能包含一个 dataDir 字样, 即使它被注释掉了。)

dataLogDir: 用于单独设置 transaction log 的目录, transaction log 分离可以避免和普通 log 还有快照的竞争

dataDir: 新安装 zk 这文件夹里面是没有文件的, 可以通过 snapCount 参数配置产生快照的时机

以下配置集群中才会使用，后面再讨论

tickTime: 心跳时间，为了确保连接存在的，以毫秒为单位，最小超时时间为两个心跳时间
initLimit: 多少个心跳时间内，允许其他 server 连接并初始化数据，如果 ZooKeeper 管理的数据较大，则应相应增大这个值

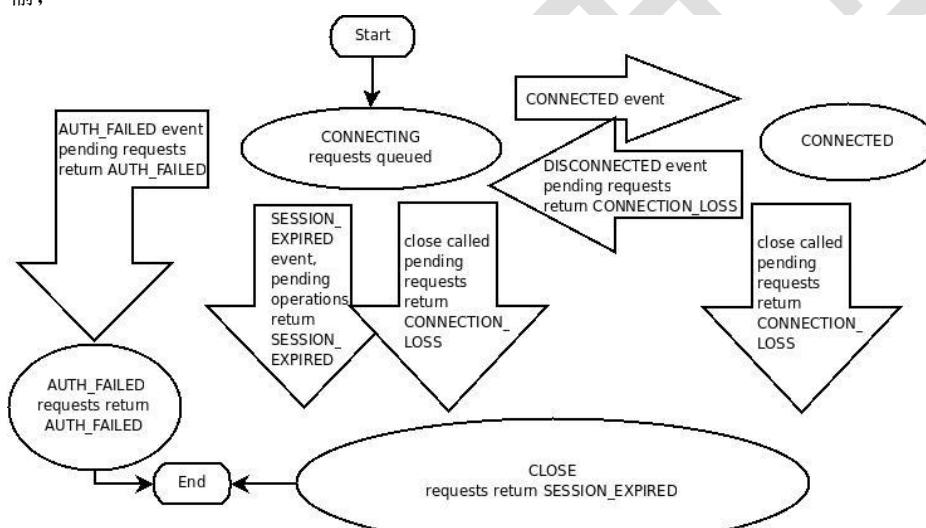
syncLimit: 多少个 tickTime 内，允许 follower 同步，如果 follower 落后太多，则会被丢弃。

2.2. ZK 的特性

Zk 的特性会从会话、数据节点，版本，Watcher，ACL 权限控制，集群角色这些部分来了解，其中重点需要掌握的[数据节点与 Watcher](#)

2.2.1. 会话

客户端与服务端的一次会话连接，本质是 TCP 长连接，通过会话可以进行心跳检测和数据传输；



会话(session)是 zookeeper 非常重要的概念，客户端和服务端之间的任何交互操作都与会话有关

会话状态

看下这图，Zk 客户端和服务端成功连接后，就创建了一次会话，ZK 会话在整个运行期间的生命周期中，会在不同的会话状态之间切换，这些状态包括：

CONNECTING、CONNECTED、RECONNECTING、RECONNECTED、CLOSE

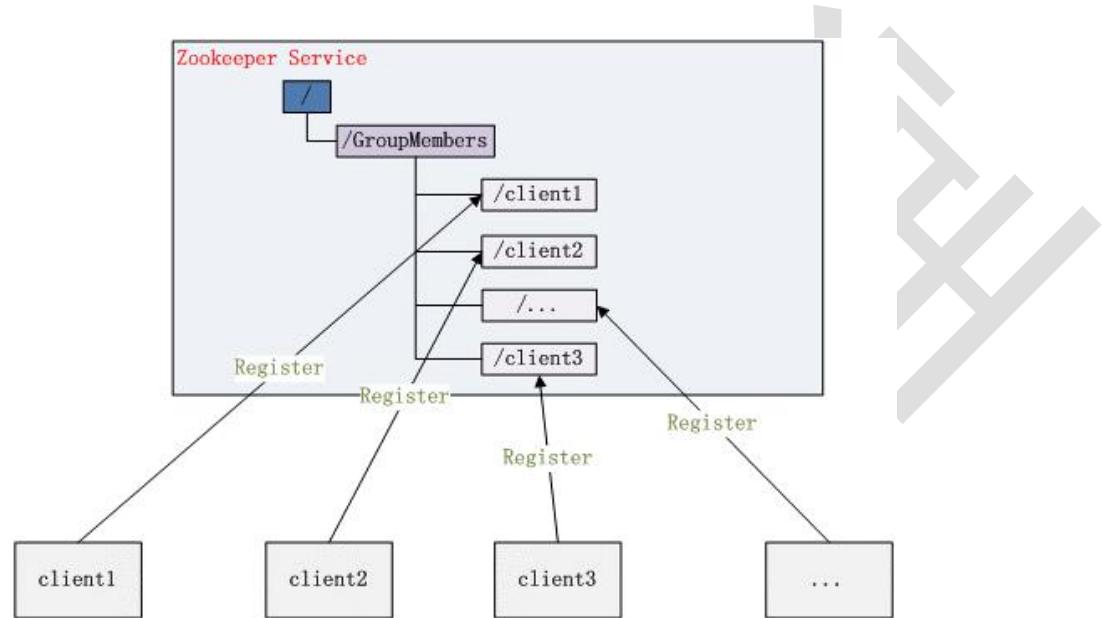
一旦客户端开始创建 Zookeeper 对象，那么客户端状态就会变成 CONNECTING 状态，同时客户端开始尝试连接服务端，连接成功后，客户端状态变为 CONNECTED，通常情况下，由于断网或其他原因，客户端与服务端之间会出现断开情况，一旦碰到这种情况，Zookeeper 客户端会自动进行重连服务，同时客户端状态再次变成 CONNECTING，直到重新连上服务端后，状态又变为 CONNECTED，在通常情况下，客户端的状态总是介于 CONNECTING 和 CONNECTED 之间。但是，如果出现诸如会话超时、权限检查或是客户端主动退出程序等

情况，客户端的状态就会直接变更为 CLOSE 状态

2.2.2. ZK 数据模型

ZooKeeper 的视图结构和标准的 Unix 文件系统类似，其中每个节点称为“数据节点”或 ZNode，每个 znode 可以存储数据，还可以挂载子节点，因此可以称之为“树”

第二点需要注意的是，每一个 znode 都必须有值，如果没有值，节点是不能创建成功的。



- 在 Zookeeper 中，znode 是一个跟 Unix 文件系统路径相似的节点，可以往这个节点存储或获取数据
- 通过客户端可对 znode 进行增删改查的操作，还可以注册 watcher 监控 znode 的变化。

2.2.3. Zookeeper 节点类型

节点类型非常重要，是后面项目实战的基础。

a、Znode 有两种类型：

短暂（ephemeral）（`create -e /app1/test1 "test1"` 客户端断开连接 zk 删除 ephemeral 类型节点）

持久（persistent）（`create -s /app1/test2 "test2"` 客户端断开连接 zk 不删除 persistent 类型节点）

b、Znode 有四种形式的目录节点（默认是 persistent ）

PERSISTENT

PERSISTENT_SEQUENTIAL (持久序列/test00000000019)

EPHEMERAL

EPHEMERAL_SEQUENTIAL

c、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

```
[zk: localhost:2181(CONNECTED) 11] ls /
[zookeeper, appl]
[zk: localhost:2181(CONNECTED) 12] create -s /appl/aa 100
Created /appl/aa0000000001
[zk: localhost:2181(CONNECTED) 13] create -s /appl/bb 100
Created /appl/bb0000000002
[zk: localhost:2181(CONNECTED) 14] create -s /appl/aa 100
Created /appl/aa0000000003
```

d、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

2.2.4. Zookeeper 节点状态属性

序号	属性	数据结构	描述
1	czxid	long	节点被创建的Zxid值
2	mzxid	long	节点被修改的Zxid值
3	pzxid	long	子节点最有一次被修改时的事务ID
4	ctime	long	节点被创建的时间
5	mtime	long	节点最后一次被修改的时间
6	versoin	long	节点被修改的版本号,
7	cversion	long	节点的所拥有子节点被修改的版本号
8	aversion	long	节点的ACL被修改的版本号
9	ephemeralOwner	long	如果此节点为临时节点，那么它的值为这个节点拥有者的会话ID；否则，它的值为0
10	dataLength	int	节点数据域的长度
11	numChildren	int	节点拥有的子节点个数

2.2.5. ACL 保障数据的安全

ACL 机制，表示为 scheme:id:permissions，第一个字段表示采用哪一种机制，第二个 id 表示用户， permissions 表示相关权限（如只读，读写，管理等）。

zookeeper 提供了如下几种机制（scheme）：

world: 它下面只有一个 id，叫 anyone, world:anyone 代表任何人，zookeeper 中对所有人有权限的结点就是属于 world:anyone 的

auth: 它不需要 id, 只要是通过 authentication 的 user 都有权限(zookeeper 支持通过 kerberos 来进行 authencation, 也支持 username/password 形式的 authentication)

digest: 它对应的 id 为 username:BASE64(SHA1(password)), 它需要先通过 username:password 形式的 authentication

ip: 它对应的 id 为客户机的 IP 地址, 设置的时候可以设置一个 ip 段, 比如 ip:192.168.1.0/16, 表示匹配前 16 个 bit 的 IP 段

现在看这可能懵懵懂懂, 不过没关系, 等会在客户端操作的时候会有详细的操作

2.3. 命令行

2.3.1. 服务端常用命令

在准备好相应的配置之后, 可以直接通过 zkServer.sh 这个脚本进行服务的相关操作

启动 ZK 服务: sh bin/zkServer.sh start

查看 ZK 服务状态: sh bin/zkServer.sh status

停止 ZK 服务: sh bin/zkServer.sh stop

重启 ZK 服务: sh bin/zkServer.sh restart

2.3.2. 客户端常用命令

使用 zkCli.sh -server 127.0.0.1:2181 连接到 ZooKeeper 服务, 连接成功后, 系统会输出 ZooKeeper 的相关环境以及配置信息。命令行工具的一些简单操作如下:

- 显示根目录下、文件: ls / 使用 ls 命令来查看当前 ZooKeeper 中所包含的内容
- 显示根目录下、文件: ls2 / 查看当前节点数据并能看到更新次数等数据
- 创建文件, 并设置初始内容: create /zk "test" 创建一个新的 znode 节点 “ zk ” 以及与它关联的字符串 [-e] [-s] 【-e 零时节点】 【-s 顺序节点】
- 获取文件内容: get /zk 确认 znode 是否包含我们所创建的字符串 [watch]【watch 监听】
- 修改文件内容: set /zk "zkbak" 对 zk 所关联的字符串进行设置
- 删除文件: delete /zk 将刚才创建的 znode 删除, 如果存在子节点删除失败
- 递归删除: rmr /zk 将刚才创建的 znode 删除, 子节点同时删除
- 退出客户端: quit
- 帮助命令: help

2.3.3. ACL 命令常用命令

再回过头来看下 ACL 权限

Zookeeper 的 ACL(Access Control List), 分为三个维度: scheme、id、permission
通常表示为: scheme:id:permission

- schema:代表授权策略
- id:代表用户
- permission:代表权限

2.3.3.1. Scheme

world:

默认方式, 相当于全世界都能访问

auth:

代表已经认证通过的用户(可以通过 addauth digest user:pwd 来添加授权用户)

digest:

即用户名:密码这种方式认证, 这也是业务系统中最常用的

ip:

使用 Ip 地址认证

2.3.3.2. id

id 是验证模式, 不同的 scheme, id 的值也不一样。

scheme 为 auth 时:

username:password

scheme 为 digest 时:

username:BASE64(SHA1(password))

scheme 为 ip 时:

客户端的 ip 地址。

scheme 为 world 时

anyone。

2.3.3.3. Permission

CREATE、READ、WRITE、DELETE、ADMIN 也就是 增、删、改、查、管理权限, 这 5 种权限
简写为 crwda(即: 每个单词的首字符缩写)

CREATE(c): 创建子节点的权限

DELETE(d): 删除节点的权限

READ(r): 读取节点数据的权限

WRITE(w): 修改节点数据的权限

ADMIN(a): 设置子节点权限的权限

2.3.3.4. ACL 命令

2.3.3.4.1. getAcl

获取指定节点的 ACL 信息

```
create /testDir/testAcl deer # 创建一个子节点  
getAcl /testDir/testAcl      # 获取该节点的 acl 权限信息
```

2.3.3.4.2. setAcl

设置指定节点的 ACL 信息

```
setAcl /testDir/testAcl world:anyone:crwa    # 设置该节点的 acl 权限  
getAcl /testDir/testAcl      # 获取该节点的 acl 权限信息, 成功后, 该节点就少了 d 权限  
  
create /testDir/testAcl/xyz xyz-data    # 创建子节点  
delete /testDir/testAcl/xyz      # 由于没有 d 权限, 所以提示无法删除
```

2.3.3.4.3. addauth

注册会话授权信息

2.3.3.4.3.1. Auth

```
addauth digest user1:123456          # 需要先添加一个用户  
setAcl /testDir/testAcl auth:user1:123456:crwa  # 然后才可以拿着这个用户去设置权限  
  
getAcl /testDir/testAcl      # 密码是以密文的形式存储的
```

```
create /testDir/testAcl/testa aaa  
delete /testDir/testAcl/testa    # 由于没有 d 权限, 所以提示无法删除
```

退出客户端后:

```
ls /testDir/testAcl  # 没有权限无法访问  
create /testDir/testAcl/testb bbb # 没有权限无法访问
```

```
addauth digest user1:123456 # 重新新增权限后可以访问了
```

2.3.3.4.3.2. Digest

auth 与 digest 的区别就是，前者使用明文密码进行登录，后者使用密文密码进行登录

```
create /testDir/testDigest data  
addauth digest user1:123456  
setAcl /testDir/testDigest digest:user1:HYGa7IZRm2PUBFiFFu8xY2pPP/s=:crwa    # 使用 digest  
来设置权限
```

[注意：这里如果使用明文，会导致该 znode 不可访问](#)

通过明文获得密文

```
shell>  
java -Djava.ext.dirs=/soft/zookeeper-3.4.12/lib -cp /soft/zookeeper-3.4.12/zookeeper-3.4.12.jar  
org.apache.zookeeper.server.auth.DigestAuthenticationProvider deer:123456  
  
deer:123456->deer:ACFm5rWnnKn9K9RN/Oc8qEYGYDs=
```

2.3.3.4.4. acl 命令行 ip

```
create /testDir/testIp data  
setAcl /testDir/testIp ip:192.168.30.10:cdrwa  
  
getAcl /testDir/testIp
```

2.3.3.4.5. ACL 权限补充

很多同学练习 setAcl 权限时由于失误，导致节点无法删除

```
create /enjoy1/node1 enjoy  
setAcl /enjoy1 world:anyone:r
```

这个时候无论是 delete 还是 rmr 都没有权限删除

解决方式:启用 super 权限

使用 DigestAuthenticationProvider.generateDigest("super:admin")；获得密码

1. 修改 zkServer 启动脚本增加

```
"-Dzookeeper.DigestAuthenticationProvider.superDigest=super:xQJmxLMiHGwaqBvst5y6rkB6  
HQs=""
```

2. 启动客户端用管理员登陆

```
addauth digest super:admin
```

2.3.4. 常用四字命令

ZooKeeper 支持某些特定的四字命令字母与其的交互。用来获取 ZooKeeper 服务的当前状态及相关信息。可通过 telnet 或 nc 向 ZooKeeper 提交相应的命令：

当然，前提是安装好了 nc

```
yum install nc
```

echo stat|nc 127.0.0.1 2181 来查看哪个节点被选择作为 follower 或者 leader

使用 echo ruok|nc 127.0.0.1 2181 测试是否启动了该 Server, 若回复 imok 表示已经启动。

echo dump| nc 127.0.0.1 2181 ,列出未经处理的会话和临时节点。

echo kill | nc 127.0.0.1 2181 ,关掉 server

echo conf | nc 127.0.0.1 2181 ,输出相关服务配置的详细信息。

echo cons | nc 127.0.0.1 2181 ,列出所有连接到服务器的客户端的完全的连接 / 会话的详细信息

echo envi |nc 127.0.0.1 2181 ,输出关于服务环境的详细信息（区别于 conf 命令）。

echo reqs | nc 127.0.0.1 2181 ,列出未经处理的请求。

echo wchs | nc 127.0.0.1 2181 ,列出服务器 watch 的详细信息。

echo wchc | nc 127.0.0.1 2181 ,通过 session 列出服务器 watch 的详细信息，它的输出是一个与 watch 相关的会话的列表。

echo wchp | nc 127.0.0.1 2181 ,通过路径列出服务器 watch 的详细信息。它输出一个与 session 相关的路径。

2.3.5. ZooKeeper 日志可视化

前面以及讲了两个非常重要的配置一个是 dataDir, 存放的快照数据，一个是 dataLogDir, 存放的是事务日志文件

```
java -cp  
/soft/zookeeper-3.4.12/zookeeper-3.4.12.jar:/soft/zookeeper-3.4.12/lib/slf4j-api-1.7.25.jar  
org.apache.zookeeper.server.LogFormatter log.1
```

```
java -cp  
/soft/zookeeper-3.4.12/zookeeper-3.4.12.jar:/soft/zookeeper-3.4.12/lib/slf4j-api-1.7.25.jar  
org.apache.zookeeper.server.SnapshotFormatter log.1
```

2.4. Java 客户端框架 (*重要)

2.4.1. Zookeeper 原生客户端

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>enjoy</groupId>
    <artifactId>zookeeperJavaApi</artifactId>
    <version>1.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
        <dependency>
            <groupId>org.apache.zookeeper</groupId>
            <artifactId>zookeeper</artifactId>
            <version>3.4.12</version>
        </dependency>
        <dependency>
            <groupId>com.101tec</groupId>
            <artifactId>zkclient</artifactId>
            <version>0.10</version>
        </dependency>
        <dependency>
            <groupId>org.apache.curator</groupId>
            <artifactId>curator-framework</artifactId>
            <version>4.0.0</version>
        </dependency>
        <dependency>
            <groupId>org.apache.curator</groupId>
            <artifactId>curator-recipes</artifactId>
            <version>4.0.0</version>
        </dependency>
    </dependencies>

```

```
</dependency>
</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <configuration>
                <source>1.8</source>
                <target>1.8</target>
            </configuration>
        </plugin>
    </plugins>
</build>

</project>
```

2.4.1.1. 创建会话

```
package cn.enjoy.javaapi;

import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.ZooKeeper;
import org.junit.Test;

import java.util.concurrent.CountDownLatch;

/**
 * Created by VULCAN on 2018/11/5.
 */
public class TestCreateSession {
    //ZooKeeper 服务地址
    private static final String SERVER = "192.168.30.10:2181";

    //会话超时时间
    private final int SESSION_TIMEOUT = 30000;

    @Test
    /**
     * 获得 session 的方式，这种方式可能会在 ZooKeeper 还没有获得连接的时候就已经对
```

ZK 进行访问了

```
/*
public void testSession1() throws Exception {
    ZooKeeper zooKeeper = new ZooKeeper(SERVER,SESSION_TIMEOUT,null);
    System.out.println(zooKeeper);
    System.out.println(zooKeeper.getState());
}

//发令枪
private CountDownLatch countDownLatch = new CountDownLatch(1);

@Test
/**
 * 对获得 Session 的方式进行优化，在 ZooKeeper 初始化完成以前先等待，等待完成后
再进行后续操作
*/
public void testSession2() throws Exception {
    ZooKeeper zooKeeper = new ZooKeeper(SERVER, SESSION_TIMEOUT, new Watcher() {
        @Override
        public void process(WatchedEvent watchedEvent) {
            if(watchedEvent.getState() == Event.KeeperState.SyncConnected) {
                //确认已经连接完毕后再进行操作
                countDownLatch.countDown();
                System.out.println("已经获得了连接");
            }
        }
    });
    //连接完成之前先等待
    countDownLatch.await();
    System.out.println(zooKeeper.getState());
}
}
```

2.4.1.2. 客户端基本操作

```
package cn.enjoy.javaapi;

import org.apache.zookeeper.*;
```

```
import java.io.IOException;
import java.util.concurrent.CountDownLatch;

public class TestJavaApi implements Watcher {

    private static final int SESSION_TIMEOUT = 10000;
    private static final String CONNECTION_STRING = "192.168.30.10:2181";
    private static final String ZK_PATH = "/leader";
    private ZooKeeper zk = null;

    private CountDownLatch connectedSemaphore = new CountDownLatch(1);

    /**
     * 创建 ZK 连接
     *
     * @param connectString ZK 服务器地址列表
     * @param sessionTimeout Session 超时时间
     */
    public void createConnection(String connectString, int sessionTimeout) {
        this.releaseConnection();
        try {
            zk = new ZooKeeper(connectString, sessionTimeout, this);
            connectedSemaphore.await();
        } catch (InterruptedException e) {
            System.out.println("连接创建失败，发生 InterruptedException");
            e.printStackTrace();
        } catch (IOException e) {
            System.out.println("连接创建失败，发生 IOException");
            e.printStackTrace();
        }
    }

    /**
     * 关闭 ZK 连接
     */
    public void releaseConnection() {
        if (null != this.zk) {
            try {
                this.zk.close();
            } catch (InterruptedException e) {
                // ignore
                e.printStackTrace();
            }
        }
    }
}
```

```
        }

    }

    /**
     * 创建节点
     *
     * @param path 节点 path
     * @param data 初始数据内容
     * @return
     */
    public boolean createPath(String path, String data) {
        try {
            System.out.println("节点创建成功, Path: "
                    + this.zk.create(path, // 节点路径
                        data.getBytes(), // 节点内容
                        ZooDefs.Ids.OPEN_ACL_UNSAFE, // 节点权限
                        CreateMode.EPHEMERAL) // 节点类型
                    + ", content: " + data);
        } catch (KeeperException e) {
            System.out.println("节点创建失败, 发生 KeeperException");
            e.printStackTrace();
        } catch (InterruptedException e) {
            System.out.println("节点创建失败, 发生 InterruptedException");
            e.printStackTrace();
        }
        return true;
    }

    /**
     * 读取指定节点数据内容
     *
     * @param path 节点 path
     * @return
     */
    public String readData(String path) {
        try {
            System.out.println("获取数据成功, path: " + path);
            return new String(this.zk.getData(path, false, null));
        } catch (KeeperException e) {
            System.out.println("读取数据失败, 发生 KeeperException, path: " + path);
            e.printStackTrace();
            return "";
        } catch (InterruptedException e) {
            System.out.println("读取数据失败, 发生 InterruptedException, path: " + path);
        }
    }
}
```

```
        e.printStackTrace();
        return "";
    }

}

/***
 * 更新指定节点数据内容
 *
 * @param path 节点 path
 * @param data 数据内容
 * @return
 */
public boolean writeData(String path, String data) {
    try {
        System.out.println("更新数据成功, path: " + path + ", stat: " +
                           this.zk.setData(path, data.getBytes(), -1));
    } catch (KeeperException e) {
        System.out.println("更新数据失败, 发生 KeeperException, path: " + path);
        e.printStackTrace();
    } catch (InterruptedException e) {
        System.out.println("更新数据失败, 发生 InterruptedException, path: " + path);
        e.printStackTrace();
    }
    return false;
}

/***
 * 删除指定节点
 *
 * @param path 节点 path
 */
public void deleteNode(String path) {
    try {
        this.zk.delete(path, -1);
        System.out.println("删除节点成功, path: " + path);
    } catch (KeeperException e) {
        System.out.println("删除节点失败, 发生 KeeperException, path: " + path);
        e.printStackTrace();
    } catch (InterruptedException e) {
        System.out.println("删除节点失败, 发生 InterruptedException, path: " + path);
        e.printStackTrace();
    }
}
```

```

public static void main(String[] args) {

    TestJavaApi sample = new TestJavaApi();
    sample.createConnection(CONNECTION_STRING, SESSION_TIMEOUT);
    if (sample.createPath(ZK_PATH, "我是节点初始内容")) {
        System.out.println();
        System.out.println("数据内容: " + sample.readData(ZK_PATH) + "\n");
        sample.writeData(ZK_PATH, "更新后的数据");
        System.out.println("数据内容: " + sample.readData(ZK_PATH) + "\n");
        sample.deleteNode(ZK_PATH);
    }

    sample.releaseConnection();
}

/**
 * 收到来自 Server 的 Watcher 通知后的处理。
 */
@Override
public void process(WatchedEvent event) {
    System.out.println("收到事件通知: " + event.getState() + "\n");
    if (Event.KeeperState.SyncConnected == event.getState()) {
        connectedSemaphore.countDown();
    }
}
}

```

2.4.1.3. Watch 机制

```

package cn.enjoy.javaapi;

import org.apache.zookeeper.*;
import org.apache.zookeeper.data.Stat;

import java.util.List;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.atomic.AtomicInteger;

public class ZooKeeperWatcher implements Watcher {

```

```
/** 定义原子变量 */
AtomicInteger seq = new AtomicInteger();
/** 定义 session 失效时间 */
private static final int SESSION_TIMEOUT = 10000;
/** zookeeper 服务器地址 */
private static final String CONNECTION_ADDR = "192.168.30.10:2181";
/** zk 父路径设置 */
private static final String PARENT_PATH = "/testWatch";
/** zk 子路径设置 */
private static final String CHILDREN_PATH = "/testWatch/children";
/** 进入标识 */
private static final String LOG_PREFIX_OF_MAIN = "【Main】";
/** zk 变量 */
private ZooKeeper zk = null;
/** 信号量设置，用于等待 zookeeper 连接建立之后 通知阻塞程序继续向下执行 */
private CountDownLatch connectedSemaphore = new CountDownLatch(1);

/**
 * 创建 ZK 连接
 * @param connectAddr ZK 服务器地址列表
 * @param sessionTimeout Session 超时时间
 */
public void createConnection(String connectAddr, int sessionTimeout) {
    this.releaseConnection();
    try {
        zk = new ZooKeeper(connectAddr, sessionTimeout, this);
        System.out.println(LOG_PREFIX_OF_MAIN + "开始连接 ZK 服务器");
        connectedSemaphore.await();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

/**
 * 关闭 ZK 连接
 */
public void releaseConnection() {
    if (this.zk != null) {
        try {
            this.zk.close();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
}

/**
 * 创建节点
 * @param path 节点路径
 * @param data 数据内容
 * @return
 */
public boolean createPath(String path, String data) {
    try {
        //设置监控(由于 zookeeper 的监控都是一次性的所以 每次必须设置监控)
        this.zk.exists(path, true);
        System.out.println(LOG_PREFIX_OF_MAIN + "节点创建成功, Path: " +
            this.zk.create(/**路径*/
                path,
                /**数据*/
                data.getBytes(),
                /**所有可见*/
                ZooDefs.Ids.OPEN_ACL_UNSAFE,
                /**永久存储*/
                CreateMode.PERSISTENT ) +
            ", content: " + data);
    } catch (Exception e) {
        e.printStackTrace();
        return false;
    }
    return true;
}

/**
 * 读取指定节点数据内容
 * @param path 节点路径
 * @return
 */
public String readData(String path, boolean needWatch) {
    try {
        return new String(this.zk.getData(path, needWatch, null));
    } catch (Exception e) {
        e.printStackTrace();
        return "";
    }
}

/**
```

```
* 更新指定节点数据内容
* @param path 节点路径
* @param data 数据内容
* @return
*/
public boolean writeData(String path, String data) {
    try {
        System.out.println(LOG_PREFIX_OF_MAIN + "更新数据成功, path: " + path + ","
stat: " +
            this.zk.setData(path, data.getBytes(), -1));
    } catch (Exception e) {
        e.printStackTrace();
    }
    return false;
}

/**
 * 删除指定节点
 *
 * @param path
 *          节点 path
 */
public void deleteNode(String path) {
    try {
        this.zk.delete(path, -1);
        System.out.println(LOG_PREFIX_OF_MAIN + "删除节点成功, path: " + path);
    } catch (Exception e) {
        e.printStackTrace();
    }
}

/**
 * 判断指定节点是否存在
 * @param path 节点路径
 */
public Stat exists(String path, boolean needWatch) {
    try {
        return this.zk.exists(path, needWatch);
    } catch (Exception e) {
        e.printStackTrace();
        return null;
    }
}
```

```
/**  
 * 获取子节点  
 * @param path 节点路径  
 */  
private List<String> getChildren(String path, boolean needWatch) {  
    try {  
        return this.zk.getChildren(path, needWatch);  
    } catch (Exception e) {  
        e.printStackTrace();  
        return null;  
    }  
}  
  
/**  
 * 删除所有节点  
 */  
public void deleteAllTestPath() {  
    if(this.exists(CHILDREN_PATH, false) != null){  
        this.deleteNode(CHILDREN_PATH);  
    }  
    if(this.exists(PARENT_PATH, false) != null){  
        this.deleteNode(PARENT_PATH);  
    }  
}  
  
/**  
 * 收到来自 Server 的 Watcher 通知后的处理。  
 */  
@Override  
public void process(WatchedEvent event) {  
  
    System.out.println("进入 process . . . . . event = " + event);  
  
    try {  
        Thread.sleep(200);  
    } catch (InterruptedException e) {  
        e.printStackTrace();  
    }  
  
    if (event == null) {  
        return;  
    }  
  
    // 连接状态
```

```
Watcher.Event.KeeperState keeperState = event.getState();
// 事件类型
Watcher.Event.EventType eventType = event.getType();
// 受影响的 path
String path = event.getPath();

String logPrefix = "【Watcher-" + this.seq.incrementAndGet() + "】 ";

System.out.println(logPrefix + "收到 Watcher 通知");
System.out.println(logPrefix + "连接状态:\t" + keeperState.toString());
System.out.println(logPrefix + "事件类型:\t" + eventType.toString());

if (Event.KeeperState.SyncConnected == keeperState) {
    // 成功连接上 ZK 服务器
    if (Event.EventType.None == eventType) {
        System.out.println(logPrefix + "成功连接上 ZK 服务器");
        connectedSemaphore.countDown();
    }
    //创建节点
    else if (Event.EventType.NodeCreated == eventType) {
        System.out.println(logPrefix + "节点创建");
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        this.exists(path, true);
    }
    //更新节点
    else if (Event.EventType.NodeDataChanged == eventType) {
        System.out.println(logPrefix + "节点数据更新");
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(logPrefix + "数据内容：" + this.readData(PARENT_PATH
true));
    }
    //更新子节点
    else if (Event.EventType.NodeChildrenChanged == eventType) {
        System.out.println(logPrefix + "子节点变更");
        try {
            Thread.sleep(3000);
        }
    }
}
```

```
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println(logPrefix + " 子 节 点 列 表 : " +
this.getChildren(PARENT_PATH, true));
    }
    //删除节点
    else if (Event.EventType.NodeDeleted == eventType) {
        System.out.println(logPrefix + "节点 " + path + " 被删除");
    }
}
else if (Watcher.Event.KeepAlive == keeperState) {
    System.out.println(logPrefix + "与 ZK 服务器断开连接");
}
else if (Watcher.Event.KeepAlive == keeperState) {
    System.out.println(logPrefix + "权限检查失败");
}
else if (Watcher.Event.KeepAlive == keeperState) {
    System.out.println(logPrefix + "会话失效");
}

System.out.println("-----");
}

public static void main(String[] args) throws Exception {

    //建立 watcher
    ZooKeeperWatcher zkWatch = new ZooKeeperWatcher();
    //创建连接
    zkWatch.createConnection(CONNECTION_ADDR, SESSION_TIMEOUT);
    //System.out.println(zkWatch.zk.toString());

    Thread.sleep(1000);

    // 清理节点
    zkWatch.deleteAllTestPath();

    if (zkWatch.createPath(PARENT_PATH, System.currentTimeMillis() + "")) {

        // 读取数据，在操作节点数据之前先调用 zookeeper 的 getData()方法是为了可以 watch 到对节点的操作。watch 是一次性的，
```

```
// 也就是说，如果第二次又重新调用了 setData()方法，在此之前需要重新调用一次。
System.out.println("----- read parent -----");
zkWatch.readData(PARENT_PATH, true);
// 更新数据
zkWatch.writeData(PARENT_PATH, System.currentTimeMillis() + "");

/** 读取子节点，设置对子节点变化的 watch，如果不写该方法，则在创建子节点是只会输出 NodeCreated，而不会输出 NodeChildrenChanged，也就是说创建子节点时没有 watch。
如果是递归的创建子节点，如 path="/p/c1/c2"的话，getChildren(PARENT_PATH, true)只会在创建 c1 时 watch，输出 c1 的 NodeChildrenChanged，而不会输出创建 c2 时的 NodeChildrenChanged，如果 watch 到 c2 的 NodeChildrenChanged，则需要再调用一次 getChildren(String path, true)方法，其中 path="/p/c1"
*/
System.out.println("----- read children path -----");
zkWatch.getChildren(PARENT_PATH, true);

Thread.sleep(1000);

// 创建子节点，同理如果想要 watch 到 NodeChildrenChanged 状态，需要调用
getChildren(CHILDREN_PATH, true)
zkWatch.createPath(CHILDREN_PATH, System.currentTimeMillis() + "");

Thread.sleep(1000);

zkWatch.readData(CHILDREN_PATH, true);
zkWatch.writeData(CHILDREN_PATH, System.currentTimeMillis() + "");
}

Thread.sleep(50000);
// 清理节点
zkWatch.deleteAllTestPath();
Thread.sleep(1000);
zkWatch.releaseConnection();
}

}
```

2.4.1.4. ZK 认证机制

```
package cn.enjoy.javaapi;

import org.apache.zookeeper.Watcher;
import org.apache.zookeeper.*;
import org.apache.zookeeper.data.ACL;
import org.apache.zookeeper.data.Stat;

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.CountDownLatch;
import java.util.concurrent.atomic.AtomicInteger;

public class TestZookeeperAuth implements Watcher {

    /** 连接地址 */
    final static String CONNECT_ADDR = "192.168.30.10:2181";
    /** 测试路径 */
    final static String PATH = "/testAuth";
    final static String PATH_DEL = "/testAuth/delNode";
    /** 认证类型 */
    final static String authentication_type = "digest";
    /** 认证正确方法 */
    final static String correctAuthentication = "123456";
    /** 认证错误方法 */
    final static String badAuthentication = "654321";

    static ZooKeeper zk = null;
    /** 计时器 */
    AtomicInteger seq = new AtomicInteger();
    /** 标识 */
    private static final String LOG_PREFIX_OF_MAIN = "【Main】";

    private CountDownLatch connectedSemaphore = new CountDownLatch(1);

    @Override
    public void process(WatchedEvent event) {
        try {
            Thread.sleep(200);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
if (event==null) {
    return;
}
// 连接状态
Event.KeeperState keeperState = event.getState();
// 事件类型
Event.EventType eventType = event.getType();
// 受影响的 path
String path = event.getPath();

String logPrefix = "【Watcher-" + this.seq.incrementAndGet() + "】 ";

System.out.println(logPrefix + "收到 Watcher 通知");
System.out.println(logPrefix + "连接状态:\t" + keeperState.toString());
System.out.println(logPrefix + "事件类型:\t" + eventType.toString());
if (Event.KeeperState.SyncConnected == keeperState) {
    // 成功连接上 ZK 服务器
    if (Event.EventType.None == eventType) {
        System.out.println(logPrefix + "成功连接上 ZK 服务器");
        connectedSemaphore.countDown();
    }
} else if (Event.KeeperState.Disconnected == keeperState) {
    System.out.println(logPrefix + "与 ZK 服务器断开连接");
} else if (Event.KeeperState.AuthFailed == keeperState) {
    System.out.println(logPrefix + "权限检查失败");
} else if (Event.KeeperState.Expired == keeperState) {
    System.out.println(logPrefix + "会话失效");
}
System.out.println("-----");
}

/**
 * 创建 ZK 连接
 *
 * @param connectString
 *          ZK 服务器地址列表
 * @param sessionTimeout
 *          Session 超时时间
 */
public void createConnection(String connectString, int sessionTimeout) {
    this.releaseConnection();
    try {
        zk = new ZooKeeper(connectString, sessionTimeout, this);
        //添加节点授权
        zk.addAuthInfo(authentication_type,correctAuthentication.getBytes());
    }
}
```

```
        System.out.println(LOG_PREFIX_OF_MAIN + "开始连接 ZK 服务器");
        //倒数等待
        connectedSemaphore.await();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

/**
 * 关闭 ZK 连接
 */
public void releaseConnection() {
    if (this.zk!=null) {
        try {
            this.zk.close();
        } catch (InterruptedException e) {
        }
    }
}

/**
 *
 * @param args
 * @throws Exception
 */
public static void main(String[] args) throws Exception {

    TestZookeeperAuth testAuth = new TestZookeeperAuth();
    testAuth.createConnection(CONNECT_ADDR,2000);

    List<ACL> acls = new ArrayList<ACL>(1);
    for (ACL ids_acl : ZooDefs.Ids.CREATOR_ALL_ACL) {
        acls.add(ids_acl);
    }

    try {
        zk.create(PATH, "init content".getBytes(), acls, CreateMode.PERSISTENT);
        System.out.println("使用授权 key: " + correctAuthentication + "创建节点: " + PATH
+ ", 初始内容是: init content");
    } catch (Exception e) {
        e.printStackTrace();
    }
    try {
        zk.create(PATH_DEL, "will be deleted! ".getBytes(), acls, CreateMode.PERSISTENT);
    }
}
```

```
        System.out.println("使用授权 key: " + correctAuthentication + "创建节点: " + PATH_DEL + ", 初始内容是: init content");
    } catch (Exception e) {
        e.printStackTrace();
    }

    // 获取数据
    getDataByNoAuthentication();
    getDataByBadAuthentication();
    getDataByCorrectAuthentication();

    // 更新数据
    updateDataByNoAuthentication();
    updateDataByBadAuthentication();
    updateDataByCorrectAuthentication();

    // 删除数据
    deleteNodeByBadAuthentication();
    deleteNodeByNoAuthentication();
    deleteNodeByCorrectAuthentication();
    //
    Thread.sleep(1000);

    deleteParent();
    //释放连接
    testAuth.releaseConnection();
}

/** 获取数据: 采用错误的密码 */
static void getDataByBadAuthentication() {
    String prefix = "[使用错误的授权信息]";
    try {
        ZooKeeper badzk = new ZooKeeper(CONNECT_ADDR, 2000, null);
        //授权
        badzk.addAuthInfo(authentication_type,badAuthentication.getBytes());
        Thread.sleep(2000);
        System.out.println(prefix + "获取数据: " + PATH);
        System.out.println(prefix + "成功获取数据: " + badzk.getData(PATH, false, null));
    } catch (Exception e) {
        System.err.println(prefix + "获取数据失败, 原因: " + e.getMessage());
    }
}

/** 获取数据: 不采用密码 */
static void getDataByNoAuthentication() {
```

```
String prefix = "[不使用任何授权信息]";
try {
    System.out.println(prefix + "获取数据: " + PATH);
    ZooKeeper nozk = new ZooKeeper(CONNECT_ADDR, 2000, null);
    Thread.sleep(2000);
    System.out.println(prefix + "成功获取数据: " + nozk.getData(PATH, false, null));
} catch (Exception e) {
    System.err.println(prefix + "获取数据失败, 原因: " + e.getMessage());
}
}

/** 采用正确的密码 */
static void getDataByCorrectAuthentication() {
    String prefix = "[使用正确的授权信息]";
    try {
        System.out.println(prefix + "获取数据: " + PATH);

        System.out.println(prefix + "成功获取数据: " + zk.getData(PATH, false, null));
    } catch (Exception e) {
        System.out.println(prefix + "获取数据失败, 原因: " + e.getMessage());
    }
}

/**
 * 更新数据: 不采用密码
 */
static void updateDataByNoAuthentication() {

    String prefix = "[不使用任何授权信息]";

    System.out.println(prefix + "更新数据: " + PATH);
    try {
        ZooKeeper nozk = new ZooKeeper(CONNECT_ADDR, 2000, null);
        Thread.sleep(2000);
        Stat stat = nozk.exists(PATH, false);
        if (stat!=null) {
            nozk.setData(PATH, prefix.getBytes(), -1);
            System.out.println(prefix + "更新成功");
        }
    } catch (Exception e) {
        System.err.println(prefix + "更新失败, 原因是: " + e.getMessage());
    }
}
```

```
/**  
 * 更新数据：采用错误的密码  
 */  
  
static void updateDataByBadAuthentication() {  
  
    String prefix = "[使用错误的授权信息]";  
  
    System.out.println(prefix + "更新数据: " + PATH);  
    try {  
        ZooKeeper badzk = new ZooKeeper(CONNECT_ADDR, 2000, null);  
        //授权  
        badzk.addAuthInfo(authentication_type,badAuthentication.getBytes());  
        Thread.sleep(2000);  
        Stat stat = badzk.exists(PATH, false);  
        if (stat!=null) {  
            badzk.setData(PATH, prefix.getBytes(), -1);  
            System.out.println(prefix + "更新成功");  
        }  
    } catch (Exception e) {  
        System.err.println(prefix + "更新失败，原因是: " + e.getMessage());  
    }  
}  
  
/**  
 * 更新数据：采用正确的密码  
 */  
  
static void updateDataByCorrectAuthentication() {  
  
    String prefix = "[使用正确的授权信息]";  
  
    System.out.println(prefix + "更新数据: " + PATH);  
    try {  
        Stat stat = zk.exists(PATH, false);  
        if (stat!=null) {  
            zk.setData(PATH, prefix.getBytes(), -1);  
            System.out.println(prefix + "更新成功");  
        }  
    } catch (Exception e) {  
        System.err.println(prefix + "更新失败，原因是: " + e.getMessage());  
    }  
}  
  
/**  
 * 不使用密码 删除节点  
 */
```

```
/*
static void deleteNodeByNoAuthentication() throws Exception {

    String prefix = "[不使用任何授权信息]";

    try {
        System.out.println(prefix + "删除节点: " + PATH_DEL);
        ZooKeeper nozk = new ZooKeeper(CONNECT_ADDR, 2000, null);
        Thread.sleep(2000);
        Stat stat = nozk.exists(PATH_DEL, false);
        if (stat!=null) {
            nozk.delete(PATH_DEL,-1);
            System.out.println(prefix + "删除成功");
        }
    } catch (Exception e) {
        System.err.println(prefix + "删除失败, 原因是: " + e.getMessage());
    }
}

/**
 * 采用错误的密码删除节点
 */
static void deleteNodeByBadAuthentication() throws Exception {

    String prefix = "[使用错误的授权信息]";

    try {
        System.out.println(prefix + "删除节点: " + PATH_DEL);
        ZooKeeper badzk = new ZooKeeper(CONNECT_ADDR, 2000, null);
        //授权
        badzk.addAuthInfo(authentication_type,badAuthentication.getBytes());
        Thread.sleep(2000);
        Stat stat = badzk.exists(PATH_DEL, false);
        if (stat!=null) {
            badzk.delete(PATH_DEL, -1);
            System.out.println(prefix + "删除成功");
        }
    } catch (Exception e) {
        System.err.println(prefix + "删除失败, 原因是: " + e.getMessage());
    }
}

/**
 * 使用正确的密码删除节点
*/
```

```

/*
static void deleteNodeByCorrectAuthentication() throws Exception {

    String prefix = "[使用正确的授权信息]";

    try {
        System.out.println(prefix + "删除节点: " + PATH_DEL);
        Stat stat = zk.exists(PATH_DEL, false);
        if (stat!=null) {
            zk.delete(PATH_DEL, -1);
            System.out.println(prefix + "删除成功");
        }
    } catch (Exception e) {
        System.out.println(prefix + "删除失败, 原因是: " + e.getMessage());
    }
}

/**
 * 使用正确的密码删除节点
 */
static void deleteParent() throws Exception {
    try {
        Stat stat = zk.exists(PATH_DEL, false);
        if (stat == null) {
            zk.delete(PATH, -1);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}

```

2.4.2. ZkClient

2.4.2.1. 基本操作

```

package cn.enjoy.zkclient;

import org.I0Itec.zkclient.ZkClient;
import org.I0Itec.zkclient.ZkConnection;

```

```
import java.util.List;

/**
 * Created by VULCAN on 2018/11/7.
 */
public class ZkClientOperator {

    /** zookeeper 地址 */
    static final String CONNECT_ADDR = "192.168.30.10:2181";
    /** session 超时时间 */
    static final int SESSION_OUTTIME = 10000;//ms

    public static void main(String[] args) throws Exception {
        // ZkClient zkc = new ZkClient(new ZkConnection(CONNECT_ADDR), SESSION_OUTTIME);
        ZkClient zkc = new ZkClient(CONNECT_ADDR, SESSION_OUTTIME);

        //1. create and delete 方法
        zkc.createEphemeral("/temp");
        zkc.createPersistent("/super/c1", true);
        Thread.sleep(10000);
        zkc.delete("/temp");
        zkc.deleteRecursive("/super");

        //2. 设置 path 和 data 并且读取子节点和每个节点的内容
        zkc.createPersistent("/super", "1234");
        zkc.createPersistent("/super/c1", "c1 内容");
        zkc.createPersistent("/super/c2", "c2 内容");
        List<String> list = zkc.getChildren("/super");
        for(String p : list){
            System.out.println(p);
            String rp = "/super/" + p;
            String data = zkc.readData(rp);
            System.out.println("节点为: " + rp + ", 内容为: " + data);
        }

        //3. 更新和判断节点是否存在
        zkc.writeData("/super/c1", "新内容");
        System.out.println(zkc.readData("/super/c1").toString());
        System.out.println(zkc.exists("/super/c1"));

        // 4.递归删除/super 内容
        zkc.deleteRecursive("/super");
    }
}
```

```
    }  
}
```

2.4.2.2. 监听机制

```
package cn.enjoy.zkclient;  
  
import org.I0Itec.zkclient.IZkChildListener;  
import org.I0Itec.zkclient.IZkDataListener;  
import org.I0Itec.zkclient.ZkClient;  
import org.I0Itec.zkclient.ZkConnection;  
import org.junit.Test;  
  
import java.util.List;  
  
public class TestZkClientWatcher {  
  
    /** zookeeper 地址 */  
    static final String CONNECT_ADDR = "192.168.30.10:2181";  
    /** session 超时时间 */  
    static final int SESSION_OUTTIME = 10000;//ms  
  
    @Test  
    /**  
     * subscribeChildChanges 方法 订阅子节点变化  
     */  
    public void testZkClientWatcher1() throws Exception {  
        ZkClient zkc = new ZkClient(new ZkConnection(CONNECT_ADDR), SESSION_OUTTIME);  
  
        //对父节点添加监听子节点变化。  
        zkc.subscribeChildChanges("/super", new IZkChildListener() {  
            @Override  
            public void handleChildChange(String parentPath, List<String> currentChilds)  
throws Exception {  
                System.out.println("parentPath: " + parentPath);  
                System.out.println("currentChilds: " + currentChilds);  
            }  
        });  
  
        Thread.sleep(3000);  
    }  
}
```

```
zkc.createPersistent("/super");
Thread.sleep(1000);

zkc.createPersistent("/super" + "/" + "c1", "c1 内容");
Thread.sleep(1000);

zkc.createPersistent("/super" + "/" + "c2", "c2 内容");
Thread.sleep(1000);

zkc.delete("/super/c2");
Thread.sleep(1000);

zkc.deleteRecursive("/super");
Thread.sleep(Integer.MAX_VALUE);

}

@Test
/**
 * subscribeDataChanges 订阅内容变化
 */
public void testZkClientWatcher2() throws Exception {
    ZkClient zkc = new ZkClient(new ZkConnection(CONNECT_ADDR), SESSION_OUTTIME);

    zkc.createPersistent("/super", "1234");

    //对父节点添加监听子节点变化。
    zkc.subscribeDataChanges("/super", new IZkDataListener() {
        @Override
        public void handleDataDeleted(String path) throws Exception {
            System.out.println("删除的节点为:" + path);
        }

        @Override
        public void handleDataChange(String path, Object data) throws Exception {
            System.out.println("变更的节点为:" + path + ", 变更内容为:" + data);
        }
    });

    Thread.sleep(3000);
    zkc.writeData("/super", "456", -1);
    Thread.sleep(1000);
}
```

```
        zkc.delete("/super");
        Thread.sleep(Integer.MAX_VALUE);

    }

}
```

2.4.3. Curator

2.4.3.1. 基本操作

```
package cn.enjoy.curator;

import org.apache.curator.RetryPolicy;
import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.framework.CuratorFrameworkFactory;
import org.apache.curator.framework.api.BackgroundCallback;
import org.apache.curator.framework.api.CuratorEvent;
import org.apache.curator.framework.api.CuratorListener;
import org.apache.curator.framework.api.transaction.CuratorOp;
import org.apache.curator.framework.api.transaction.CuratorTransactionResult;
import org.apache.curator.retry.ExponentialBackoffRetry;
import org.apache.zookeeper.CreateMode;
import org.apache.zookeeper.data.Stat;
import org.junit.Before;
import org.junit.Test;

import java.util.List;

import static com.sun.xml.internal.ws.dump.LoggingDumpTube.Position.Before;

/**
 * 测试 Apache Curator 框架的基本用法
 */
public class OperatorTest {
    //ZooKeeper 服务地址
    private static final String SERVER = "192.168.30.10:2181";

    //会话超时时间
    private final int SESSION_TIMEOUT = 30000;
```

```
//连接超时时间
private final int CONNECTION_TIMEOUT = 5000;

//创建连接实例
private CuratorFramework client = null;

/**
 * baseSleepTimeMs: 初始的重试等待时间
 * maxRetries: 最多重试次数
 *
 *
 * ExponentialBackoffRetry: 重试一定次数，每次重试时间依次递增
 * RetryNTimes: 重试 N 次
 * RetryOneTime: 重试一次
 * RetryUntilElapsed: 重试一定时间
 */
RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000, 3);

@org.junit.Before
public void init(){
    //创建 CuratorFrameworkImpl 实例
    client      =      CuratorFrameworkFactory.newClient(SERVER,
CONNECTION_TIMEOUT, retryPolicy);

    //启动
    client.start();
}

/**
 * 测试创建节点
 * @throws Exception
 */
@Test
public void testCreate() throws Exception{
    //创建永久节点
    client.create().forPath("/curator","/curator data".getBytes());

    //创建永久有序节点

client.create().withMode(CreateMode.PERSISTENT_SEQUENTIAL).forPath("/curator_sequential",
"/curator_sequential data".getBytes());

    //创建临时节点
    client.create().withMode(CreateMode.EPHEMERAL)
```

```
.forPath("/curator/ephemeral","/curator/ephemeral data".getBytes());  
  
        //创建临时有序节点  
        client.create().withMode(CreateMode.EPHEMERAL_SEQUENTIAL)  
            .forPath("/curator/ephemeral_path1","/curator/ephemeral_path1  
data".getBytes());  
  
    }  
  
    /**  
     * 测试检查某个节点是否存在  
     * @throws Exception  
     */  
    @Test  
    public void testCheck() throws Exception{  
        Stat stat1 = client.checkExists().forPath("/curator");  
        Stat stat2 = client.checkExists().forPath("/curator2");  
  
        System.out.println("/curator'是否存在: " + (stat1 != null ? true : false));  
        System.out.println("/curator2'是否存在: " + (stat2 != null ? true : false));  
    }  
  
    /**  
     * 测试异步设置节点数据  
     * @throws Exception  
     */  
    @Test  
    public void testSetDataAsync() throws Exception{  
        //创建监听器  
        CuratorListener listener = new CuratorListener() {  
  
            @Override  
            public void eventReceived(CuratorFramework client, CuratorEvent event)  
                throws Exception {  
                System.out.println(event.getPath());  
            }  
        };  
  
        //添加监听器  
        client.getCuratorListenable().addListener(listener);  
    }
```

```
//异步设置某个节点数据
client.setData().inBackground().forPath("/curator","sync".getBytes());

//为了防止单元测试结束从而看不到异步执行结果，因此暂停 10 秒
Thread.sleep(10000);
}

/**
 * 测试另一种异步执行获取通知的方式
 * @throws Exception
 */
@Test
public void testSetDataAsyncWithCallback() throws Exception{
    BackgroundCallback callback = new BackgroundCallback() {

        @Override
        public void processResult(CuratorFramework client, CuratorEvent event)
            throws Exception {
            System.out.println(event.getPath());
        }
    };

    //异步设置某个节点数据
    client.setData().inBackground(callback).forPath("/curator","/curator modified data with
Callback".getBytes());

    //为了防止单元测试结束从而看不到异步执行结果，因此暂停 10 秒
    Thread.sleep(10000);
}

/**
 * 测试删除节点
 * @throws Exception
 */
@Test
public void testDelete() throws Exception{
    //创建测试节点
    client.create().orSetData().creatingParentsIfNeeded()
        .forPath("/curator/del_key1","/curator/del_key1 data".getBytes());

    client.create().orSetData().creatingParentsIfNeeded()
```

```
.forPath("/curator/del_key2","/curator/del_key2 data".getBytes());  
  
client.create().forPath("/curator/del_key2/test_key","test_key data".getBytes());  
  
//删除该节点  
client.delete().forPath("/curator/del_key1");  
  
//级联删除子节点  
client.delete().guaranteed().deletingChildrenIfNeeded().forPath("/curator/del_key2");  
}  
  
/*  
 * 测试事务管理：碰到异常，事务会回滚  
 * @throws Exception  
 */  
@Test  
public void testTransaction() throws Exception{  
    //定义几个基本操作  
    CuratorOp createOp = client.transactionOp().create()  
        .forPath("/curator/one_path","some data".getBytes());  
  
    CuratorOp setDataOp = client.transactionOp().setData()  
        .forPath("/curator","other data".getBytes());  
  
    CuratorOp deleteOp = client.transactionOp().delete()  
        .forPath("/curator");  
  
    //事务执行结果  
    List<CuratorTransactionResult> results = client.transaction()  
        .forOperations(createOp,setDataOp,deleteOp);  
  
    //遍历输出结果  
    for(CuratorTransactionResult result : results){  
        System.out.println("执行结果是: " + result.getPath() + "--" + result.getType());  
    }  
}
```

2.4.3.2. 监听机制

```
package cn.enjoy.curator;

import org.apache.curator.RetryPolicy;
import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.framework.CuratorFrameworkFactory;
import org.apache.curator.framework.api.CuratorEvent;
import org.apache.curator.framework.api.CuratorListener;
import org.apache.curator.framework.recipes.cache.*;
import org.apache.curator.retry.ExponentialBackoffRetry;
import org.apache.zookeeper.CreateMode;
import org.apache.zookeeper.WatchedEvent;
import org.apache.zookeeper.Watcher;
import org.junit.Test;

import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class EventTest {

    //ZooKeeper 服务地址
    private static final String SERVER = "192.168.30.10:2181";

    //会话超时时间
    private final int SESSION_TIMEOUT = 30000;

    //连接超时时间
    private final int CONNECTION_TIMEOUT = 5000;

    //创建连接实例
    private CuratorFramework client = null;

    /**
     * baseSleepTimeMs: 初始的重试等待时间
     * maxRetries: 最多重试次数
     *
     *
     * ExponentialBackoffRetry: 重试一定次数，每次重试时间依次递增
     * RetryNTimes: 重试 N 次
     * RetryOneTime: 重试一次
     * RetryUntilElapsed: 重试一定时间
     */
}
```

```
/*
RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000, 3);

@org.junit.Before
public void init(){
    //创建 CuratorFrameworkImpl 实例
    client      =      CuratorFrameworkFactory.newClient(SERVER,      SESSION_TIMEOUT,
CONNECTION_TIMEOUT, retryPolicy);

    //启动
    client.start();
}

/**
*
* @描述：第一种监听器的添加方式：对指定的节点进行添加操作
* 仅仅能监控指定的本节点的数据修改,删除 操作 并且只能监听一次 --->不好
*/
@Test
public void TestListenterOne() throws Exception{

client.create().orSetData().withMode(CreateMode.PERSISTENT).forPath("/test","test".getBytes());

    // 注册观察者，当节点变动时触发
    byte[] data = client.getData().usingWatcher(new Watcher() {
        @Override
        public void process(WatchedEvent event) {
            System.out.println("获取 test 节点 监听器 :" + event);
        }
    }).forPath("/test");



client.create().orSetData().withMode(CreateMode.PERSISTENT).forPath("/test","test".getBytes());
    Thread.sleep(1000);

client.create().orSetData().withMode(CreateMode.PERSISTENT).forPath("/test","test".getBytes());
    Thread.sleep(1000);
    System.out.println("节点数据: " + new String(data));
    Thread.sleep(10000);
}

/**
```

```
/*
 * @描述: 第二种监听器的添加方式: Cache 的三种实现
 *      Path Cache: 监视一个路径下 1) 孩子结点的创建、2) 删除, 3) 以及结点数据的
更新。
 *
 *          产生的事件会传递给注册的 PathChildrenCacheListener。
 *      Node Cache: 监视一个结点的创建、更新、删除, 并将结点的数据缓存在本地。
 *      Tree Cache: Path Cache 和 Node Cache 的“合体”, 监视路径下的创建、更新、删
除事件, 并缓存路径下所有孩子结点的数据。
 */

//1.path Cache 连接 路径 是否获取数据
//能监听所有的字节点 且是无限监听的模式 但是 指定目录下节点的子节点不再监听
@Test
public void setListenterTwoOne() throws Exception{
    ExecutorService pool = Executors.newCachedThreadPool();
    PathChildrenCache childrenCache = new PathChildrenCache(client, "/test", true);
    PathChildrenCacheListener childrenCacheListener = new PathChildrenCacheListener() {
        @Override
        public void childEvent(CuratorFramework client, PathChildrenCacheEvent event)
throws Exception {
            System.out.println("开始进行事件分析:----");
            ChildData data = event.getData();
            switch (event.getType()) {
                case CHILD_ADDED:
                    System.out.println("CHILD_ADDED : "+ data.getPath() + " 数据:"+
data.getData());
                    break;
                case CHILD_REMOVED:
                    System.out.println("CHILD_REMOVED : "+ data.getPath() + " 数
据:"+ data.getData());
                    break;
                case CHILD_UPDATED:
                    System.out.println("CHILD_UPDATED : "+ data.getPath() + " 数
据:"+ data.getData());
                    break;
                case INITIALIZED:
                    System.out.println("INITIALIZED : "+ data.getPath() + " 数 据 :"+
data.getData());
                    break;
                default:
                    break;
            }
        }
    };
}
```

```
childrenCache.getListenable().addListener(childrenCacheListener);
System.out.println("Register zk watcher successfully!");
childrenCache.start(PathChildrenCache.StartMode.POST_INITIALIZED_EVENT);

//创建一个节点

client.create().orSetData().withMode(CreateMode.PERSISTENT).forPath("/test","test".getBytes());

client.create().orSetData().withMode(CreateMode.EPHEMERAL).forPath("/test/node01","enjoy".
getBytes());
Thread.sleep(1000);

client.create().orSetData().withMode(CreateMode.EPHEMERAL).forPath("/test/node02","deer".g
etBytes());
Thread.sleep(1000);

client.create().orSetData().withMode(CreateMode.EPHEMERAL).forPath("/test/node02","demo".
getBytes());
Thread.sleep(1000);
client.delete().forPath("/test/node02");
Thread.sleep(10000);
}

//2.Node Cache 监控本节点的变化情况 连接 目录 是否压缩
//监听本节点的变化 节点可以进行修改操作 删除节点后会再次创建(空节点)
@Test
public void setListenterTwoTwo() throws Exception{
    ExecutorService pool = Executors.newCachedThreadPool();
    //设置节点的 cache
    final NodeCache nodeCache = new NodeCache(client, "/test", false);
    nodeCache.getListenable().addListener(new NodeCacheListener() {
        @Override
        public void nodeChanged() throws Exception {
            System.out.println("the test node is change and result is :");
            System.out.println("path : "+nodeCache.getCurrentData().getPath());
            System.out.println("data : "+new String(nodeCache.getCurrentData().getData()));
            System.out.println("stat : "+nodeCache.getCurrentData().getStat());
        }
    });
    nodeCache.start();
```

```
client.create().orSetData().withMode(CreateMode.PERSISTENT).forPath("/test","test".getBytes());
    Thread.sleep(1000);

client.create().orSetData().withMode(CreateMode.PERSISTENT).forPath("/test","enjoy".getBytes());
    Thread.sleep(10000);
}

//3.Tree Cache
// 监控 指定节点和节点下的所有的节点的变化--无限监听 可以进行本节点的删除
(不在创建)
@Test
public void TestListenterTwoThree() throws Exception{
    ExecutorService pool = Executors.newCachedThreadPool();
    //设置节点的 cache
    TreeCache treeCache = new TreeCache(client, "/test");
    //设置监听器和处理过程
    treeCache.getListenable().addListener(new TreeCacheListener() {
        @Override
        public void childEvent(CuratorFramework client, TreeCacheEvent event) throws
Exception {
            ChildData data = event.getData();
            if(data !=null){
                switch (event.getType()) {
                    case NODE_ADDED:
                        System.out.println("NODE_ADDED : "+ data.getPath() +"
据:"+ new String(data.getData()));
                        break;
                    case NODE_REMOVED:
                        System.out.println("NODE_REMOVED : "+ data.getPath() +"
数据:"+ new String(data.getData()));
                        break;
                    case NODE_UPDATED:
                        System.out.println("NODE_UPDATED : "+ data.getPath() +"
数据:"+ new String(data.getData()));
                        break;

                    default:
                        break;
                }
            }else{
                System.out.println( "data is null : "+ event.getType());
            }
        }
    });
}
```

```
//开始监听
treeCache.start();

//创建一个节点

client.create().orSetData().withMode(CreateMode.PERSISTENT).forPath("/test","test".getBytes());

    Thread.sleep(1000);

client.create().orSetData().withMode(CreateMode.EPHEMERAL).forPath("/test/node01","enjoy".
getBytes());
    Thread.sleep(1000);

client.create().orSetData().withMode(CreateMode.EPHEMERAL).forPath("/test/node01","deer".g
etBytes());

    Thread.sleep(1000);

client.create().orSetData().creatingParentsIfNeeded().withMode(CreateMode.EPHEMERAL).forPat
h("/test/node02/node02_2","deer".getBytes());

    Thread.sleep(10000);

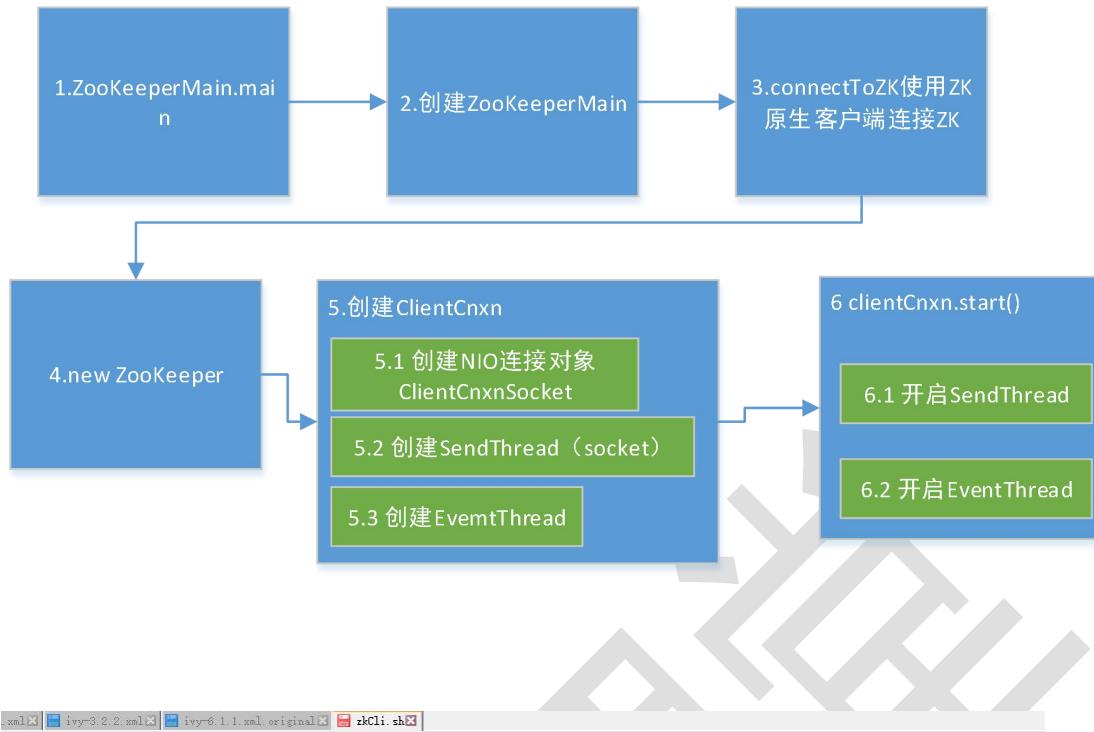
}

}
```

3. 单节点源码解读

3.1. 客户端源码

3.1.1. 总体流程



启动 zkCli.sh 脚本

```

#!/bin/bash
# use POSTIX interface, symlink is followed automatically
ZOOBIN="${BASH_SOURCE-$0}"
ZOOBIN=$(dirname "${ZOOBIN}")
ZOOBINDIR=$(cd "${ZOOBIN}"; pwd)

if [ -e "$ZOOBIN/../libexec/zkEnv.sh" ]; then
    . "$ZOOBINDIR"/../libexec/zkEnv.sh
else
    . "$ZOOBINDIR"/zkEnv.sh
fi

"$JAVA" "-Dzookeeper.log.dir=${ZOO_LOG_DIR}" "-Dzookeeper.r
        cp "$CLASSPATH" $CLIENT_JVMFLAGS $JVMFLAGS \
        org.apache.zookeeper.ZooKeeperMain "$@"

```

启动客户端 zkCli.sh 文件里面的配置

实际运行

```

286 public static void main(String args[])
287     throws KeeperException, IOException, InterruptedException
288 {
289     ZooKeeperMain main = new ZooKeeperMain(args);
290     main.run();
291 }
292

```

```

public static void main(String args[])
throws KeeperException, IOException, InterruptedException

```

```
{  
    ZooKeeperMain main = new ZooKeeperMain(args);  
    main.run();  
}
```

Main 方法流程:

1. new ZooKeeperMain 对象
2. 调用 run () 方法

在 ZookeeperMain 的构造方法里面, 重点是

```
public ZooKeeperMain(String args[]) throws IOException, InterruptedException {  
    cl.parseOptions(args);  
    System.out.println("Connecting to " + cl.getOption("server"));  
    //连接上 ZK  
    connectToZK(cl.getOption("server"));  
}
```

```
protected void connectToZK(String newHost) throws InterruptedException, IOException {  
    if (zk != null && zk.getState().isAlive()) {  
        zk.close();  
    }  
    host = newHost;  
    boolean readOnly = cl.getOption("readonly") != null;  
    zk = new ZooKeeper(host,  
                      Integer.parseInt(cl.getOption("timeout")),  
                      new MyWatcher(), readOnly);  
}
```

最终在 connectToZK 方法里面也就是使用原生的 Zk 客户端进行连接的。

```
public ZooKeeper(String connectString, int sessionTimeout, Watcher watcher,  
                 boolean canBeReadOnly)  
throws IOException  
{  
    LOG.info("Initiating client connection, connectString=" + connectString  
          + " sessionTimeout=" + sessionTimeout + " watcher=" + watcher);  
  
    watchManager.defaultWatcher = watcher;  
  
    ConnectStringParser connectStringParser = new ConnectStringParser(  
        connectString);  
    HostProvider hostProvider = new StaticHostProvider(
```

```

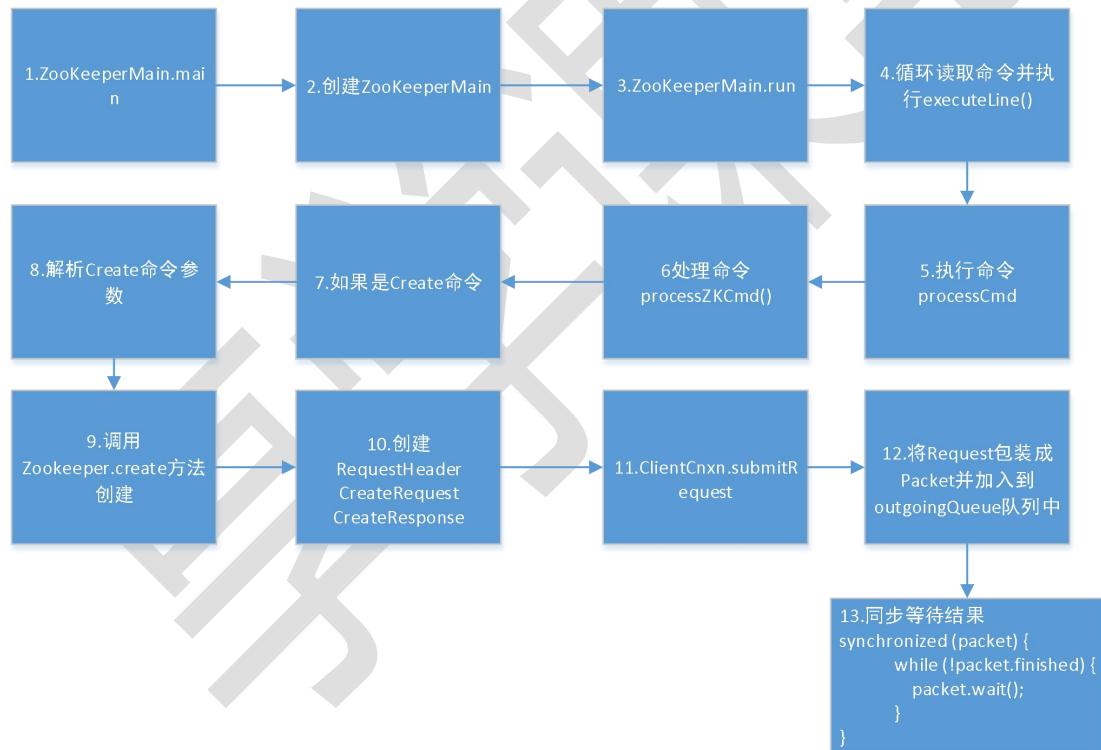
        connectStringParser.getServerAddresses());
        cnxn = new ClientCnxn(connectStringParser.getChrootPath(),
            hostProvider, sessionTimeout, this, watchManager,
            //获得和服务端连接的对象
            getClientCnxnSocket(), canBeReadOnly);
        //调用 start()
        cnxn.start();
    }
}

```

```

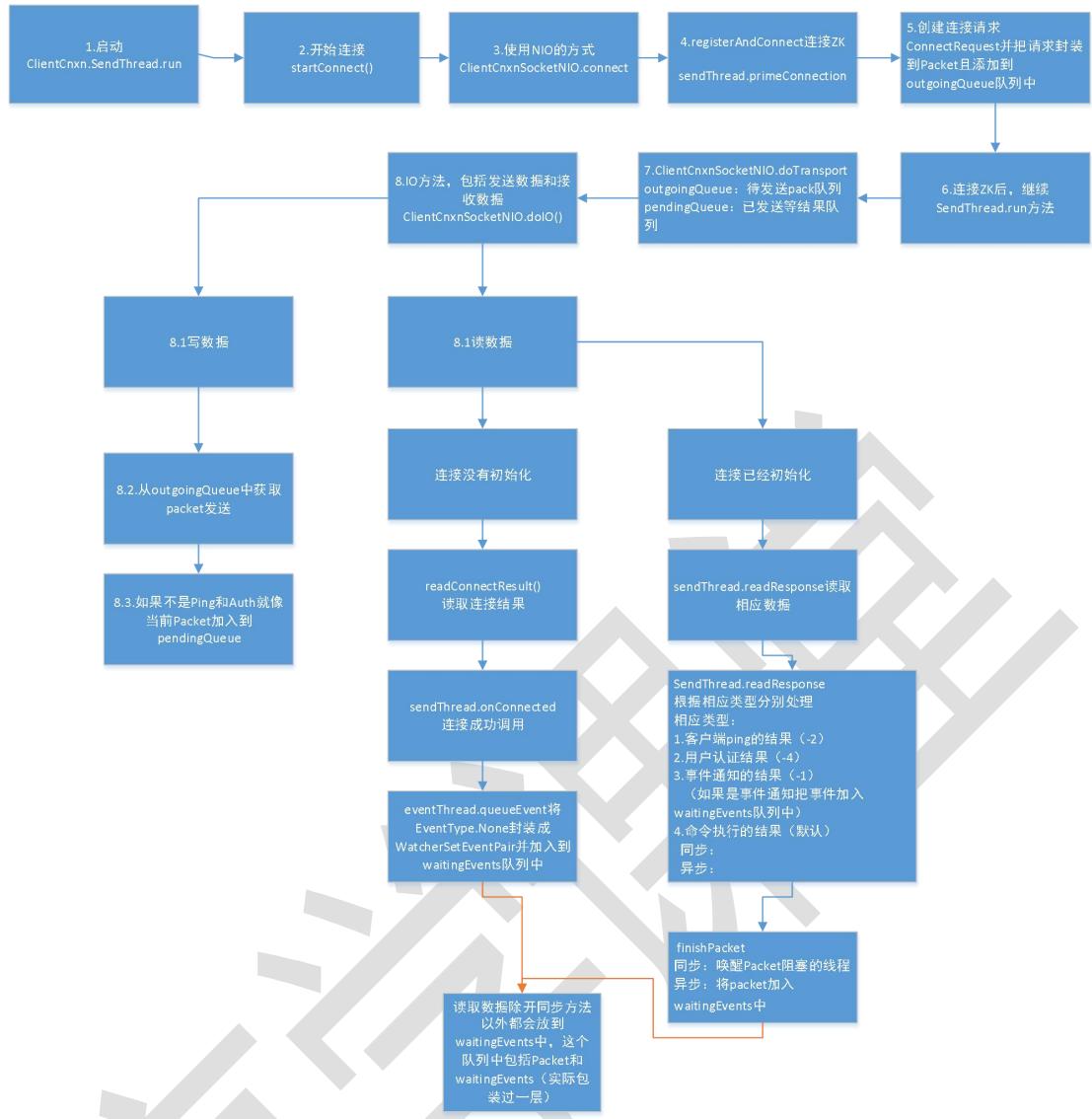
public void start() {
    sendThread.start();
    eventThread.start();
}

```



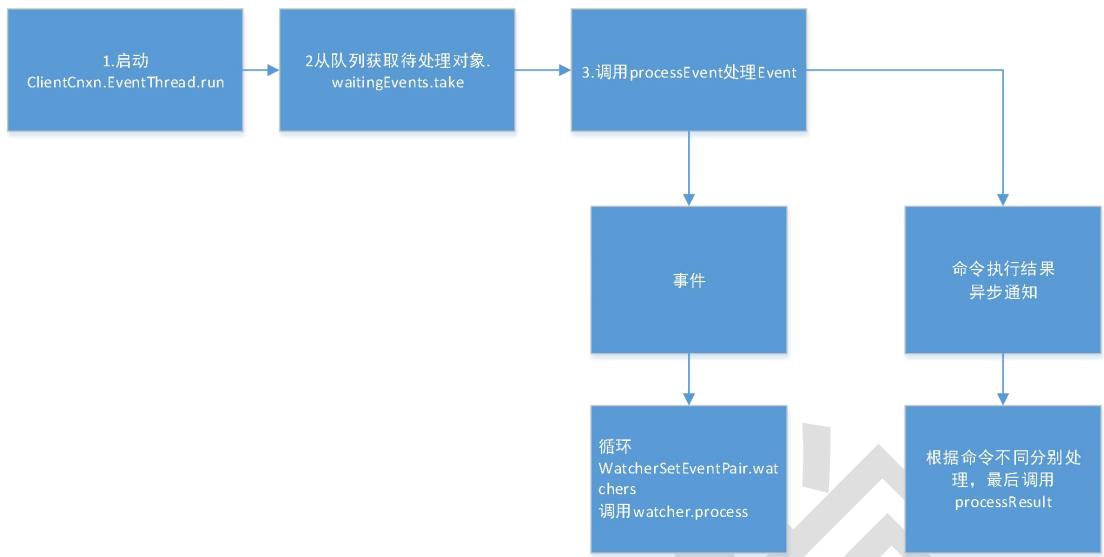
3.1.2. 开启 SendThread 线程

org.apache.zookeeper.ClientCnxn.SendThread#run

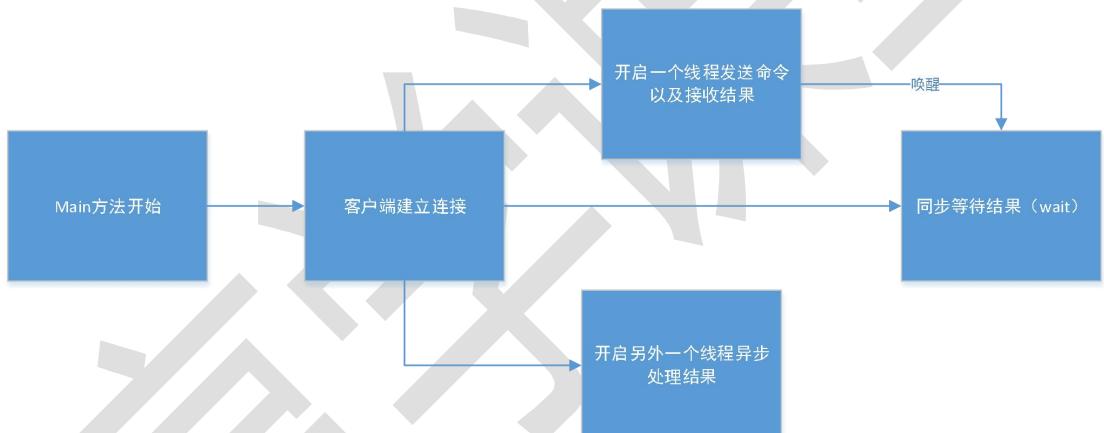


3.1.3. 开启 EventThread

org.apache.zookeeper.ClientCnxn.EventThread.run



3.1.4. 总结:



3.2. 服务端源码（单机）

3.2.1. 总体流程



3.2.2. 具体处理流程



4. Zookeeper 高级

4.1. 一致性协议概述

前面已经讨论过，在分布式环境下，有很多不确定性因素，故障随时都回发生，也讲了 CAP 理论，BASE 理论

我们希望达到，在分布式环境下能搭建一个高可用的，且数据高一致性的服务，目标是这样，但 CAP 理论告诉我们要达到这样的理想环境是不可能的。这三者最多完全满足 2 个。

在这个前提下，P（分区容错性）是必然要满足的，因为毕竟是分布式，不能把所有的应用全放到一个服务器里面，这样服务器是吃不消的，而且也存在单点故障问题。

所以，只能从一致性和可用性中找平衡。

怎么个平衡法？在这种环境下出现了 BASE 理论：

即使无法做到强一致性，但分布式系统可以根据自己的业务特点，采用适当的方式来使系统

达到最终的一致性；

BASE 由 Basically Available 基本可用、Soft state 软状态、Eventually consistent 最终一致性组成，一句话概括就是：平时系统要求是基本可用，除开成功失败，运行有可容忍的延迟状态，但是，无论如何经过一段时间的延迟后系统最终必须达成数据是一致的。

其实可能发现不管是 CAP 理论，还是 BASE 理论，他们都是理论，这些理论是需要算法来实现的，今天讲的 2PC、3PC、Paxos 算法，ZAB 算法就是干这种事情。

所以今天要讲的这些的前提一定是分布式，解决的问题全部都是在分布式环境下，怎么让系统尽可能的高可用，而且数据能最终能达到一致。

4.1.1. 两阶段提交 two-phase commit (2PC)

首先来看下 2PC，翻译过来叫两阶段提交算法，它本身是一致强一致性算法，所以很适合用作数据库的分布式事务。其实数据库的经常用到的 TCC 本身就是一种 2PC.

回想下数据库的事务，数据库不管是 MySQL 还是 MSSql，本身都提供的很完善的事务支持。

MySQL 后面学分表分库的时候会讲到在 innodb 存储引擎，对数据库的修改都会写到 undo 和 redo 中，不只是数据库，很多需要事务支持的都会用到这个思路。

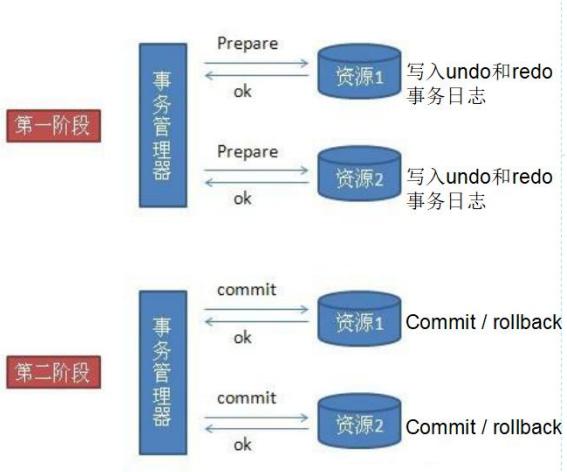
对一条数据的修改操作首先写 undo 日志，记录的数据原来的样子，接下来执行事务修改操作，把数据写到 redo 日志里面，万一捅娄子，事务失败了，可从 undo 里面回复数据。

不只是数据库，在很多企业里面，比如华为等提交数据库修改都回要求这样，你要新增一个字段，首先要把修改数据库的字段 SQL 提交给 DBA (redo)，这不够，还需要把删除你提交字段，把数据还原成你修改之前的语句也一并提交者叫 (undo)

数据库通过 undo 与 redo 能保证数据的强一致性，要解决分布式事务的前提就是当个节点是支持事务的。

这在个前提下，2pc 借鉴这失效，首先把整个分布式事务分两节点，首先第一阶段叫准备节点，事务的请求都发送给一个个的资源，这里的资源可以是数据库，也可以是其他支持事务的框架，他们会分别执行自己的事务，写日志到 undo 与 redo，但是不提交事务。

当事务管理器收到了所有资源的反馈，事务都执行没报错后，事务管理器再发送 commit 指令让资源把事务提交，一旦发现任何一个资源在准备阶段没有执行成功，事务管理器会发送 rollback，让所有的资源都回滚。这就是 2pc，非常非常简单。



说他是强一致性的是他需要保证任何一个资源都成功，整个分布式事务才成功。

4.1.1.1. 优点：

优点：原理简单，实现方便

4.1.1.2. 缺点：

缺点：同步阻塞，单点问题，数据不一致，容错性不好

4.1.1.2.1. 同步阻塞

在二阶段提交的过程中，所有的节点都在等待其他节点的响应，无法进行其他操作。这种同步阻塞极大的限制了分布式系统的性能。

4.1.1.2.2. 单点问题

协调者在整个二阶段提交过程中很重要，如果协调者在提交阶段出现问题，那么整个流程将无法运转。更重要的是，其他参与者将会处于一直锁定事务资源的状态中，而无法继续完成事务操作。

4.1.1.2.3. 数据不一致

假设当协调者向所有的参与者发送 commit 请求之后，发生了局部网络异常，或者是协调者在尚未发送完所有 commit 请求之前自身发生了崩溃，导致最终只有部分参与者收到了

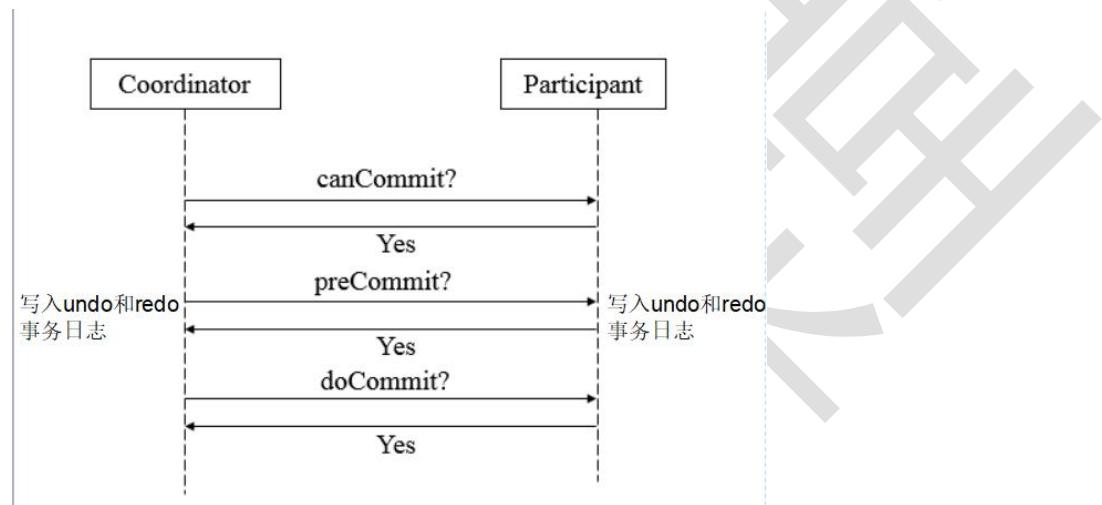
commit 请求。这将导致严重的数据不一致问题。

4.1.1.2.4. 容错性不好

二阶段提交协议没有设计较为完善的容错机制，任意一个节点失败都会导致整个事务的失败。

4.1.2. 三阶段提交 three-phase commit (3PC)

由于二阶段提交存在着诸如同步阻塞、单点问题，所以，研究者们在二阶段提交的基础上做了改进，提出了三阶段提交。



4.1.2.1. 第一阶段 canCommit

确认所有的资源是否都是健康、在线的，以约女孩举例，你会打个电话问下她是不是在家，而且可以约个会。

如果女孩有空，你在去约她。

就因为有了这一阶段，大大的减少了 2 段提交的阻塞时间，在 2 段提交，如果有 3 个数据库，恰恰第三个数据库出现问题，其他两个都会执行耗费时间的事务操作，到第三个却发现连接不上。3 段优化了这种情况

4.1.2.2. 第二阶段 PreCommit

如果所有服务都 ok，可以接收事务请求，这一阶段就可以执行事务了，这时候也是每个资源都回写 redo 与 undo 日志，事务执行成功，返回 ack (yes)，否则返回 no

4.1.2.3. 第三阶段 doCommit

这阶段和前面说的 2 阶段提交大同小异, 这个时候协调者发现所有提交者事务提交者事务都正常执行后, 给所有资源发送 commit 指令。

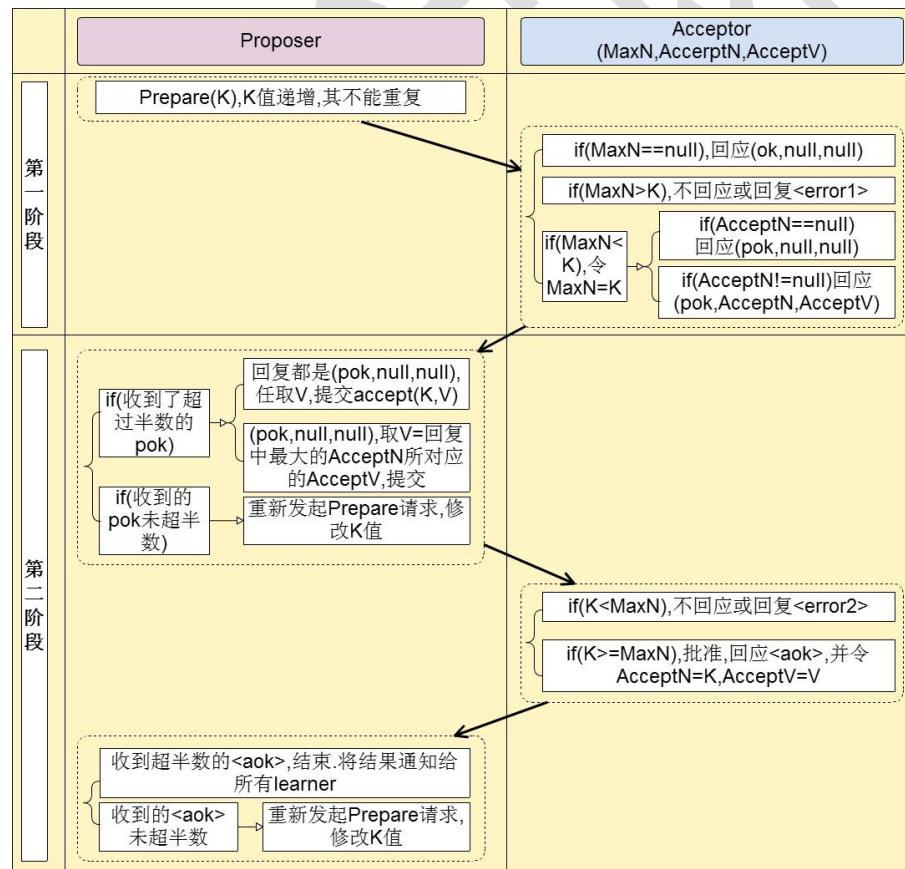
和二阶段提交有所不同的是, 他要求所有事务在协调者出现问题, 没给资源发送 commit 指令的时候, 三阶段提交算法要求资源在一段时间超时后回默认提交做 commit 操作。

这样的要求就减少了前面说的单点故障, 万一事务管理器出现问题, 事务也回提交。

但回顾整个过程, 不管是 2pc, 还是 3pc, 同步阻塞, 单点故障, 容错机制不完善这些问题都没本质上得到解决, 尤其是前面说得数据一致性问题, 反而更糟糕了。

所有数据库的分布式事务一般都是二阶段提交, 而三阶段的思想更多的被借鉴扩散成其他的算法。

4.1.3. Paxos 算法



这个算法还是有点难度的，本身这算法的提出者莱斯利·兰伯特在前面几篇论文中都不是以严谨的数学公式进行的。

其实这个 paxos 算法也分成两阶段。首先这个图有 2 个角色，提议者与接收者

4.1.3.1. 第一阶段

提议者对接收者吼了一嗓子，我有个事情要告诉你们，当然这里接受者不只一个，它也是个分布式集群

相当于星期一开早会，可耻的领导吼了句：“要开会了啊，我要公布一个编号为 001 的提案，收到请回复”。

这个时候领导就会等着，等员工回复 1 “好的”，如果回复的数目超过一半，就会进行下一步。

如果由于某些原因（接收者死机，网络问题，本身业务问题），导通过的协议未超过一半，

这个时候的领导又会再吼一嗓子，当然气势没那凶残：“好了，怕了你们了，我要公布一个新的编号为 002 的提案，收到请回复 1”【就其实和老师讲课很像，老师经常问听懂了吗？听懂的回 1，没懂的回 2，只有回复 1 的占了大多数，才能讲下个知识点】

4.1.3.2. 第二阶段

接下来到第二阶段，领导苦口婆心的把你们叫来开会了，今天编号 002 提案的内容是：“由于项目紧张，今天加班到 12 点，同意的请举手”这个时候如果绝大多数的接收者都同意，那么好，议案就这么决定了，如果员工反对或者直接夺门而去，那么领导又只能从第一个阶段开始：“大哥，大姐们，我有个新的提案 003，快回会议室吧。。”

4.1.3.3. 详细说明：

【注意：不懂没事，记住上面那简单情况就好，面试足够】

上面那个故事描绘的是个苦逼的领导和凶神恶煞的员工之间的斗争，通过这个故事你们起码要懂 paxos 协议的流程是什么样的（paxos 的核心就是少数服从多数）。

上面的故事有两个问题：

苦逼的领导（单点问题）：有这一帮凶残的下属，这领导要不可能被气死，要不也会辞职，这是单点问题。

凶神恶煞的下属（一致性问题）：如果员工一种都拒绝，故意和领导抬杠，最终要产生一个一致性的解决方案是不可能的。

所以 paxos 协议肯定不会只有一个提议者，作为下属的员工也不会那么强势
协议要求：如果接收者没有收到过提案编号，他必须接受第一个提案编号
如果接收者没有收到过其他协议，他必须接受第一个协议。

举一个例子：

有 2 个 Proposer(老板，老板之间是竞争关系)和 3 个 Acceptor(政府官员)：

4.1.3.3.1. 阶段一

1. 现在需要对一项议题来进行 paxos 过程，议题是“**A 项目我要中标！**”，这里的“我”指每个带着他的秘书 Proposer 的 Client 老板。
2. Proposer 当然听老板的话了，赶紧带着议题和现金去找 Acceptor 政府官员。
3. 作为政府官员，当然想谁给的钱多就把项目给谁。
4. Proposer-1 小姐带着现金同时找到了 Acceptor-1~Acceptor-3 官员，1 与 2 号官员分别收取了 10 比特币，找到第 3 号官员时，没想到遭到了 3 号官员的鄙视，3 号官员告诉她，Proposer-2 给了 11 比特币。不过没关系，Proposer-1 已经得到了 1,2 两个官员的认可，形成了多数派(如果没有形成多数派，Proposer-1 会去银行提款来来找官员们给每人 20 比特币，这个过程一直重复每次+10 比特币，直到多数派的形成)，满意的找老板复命去了，但是此时 Proposer-2 保镖找到了 1,2 号官员，分别给了他们 11 比特币，1,2 号官员的态度立刻转变，都说 Proposer-2 的老板懂事，这下子 Proposer-2 放心了，搞定了 3 个官员，找老板复命去了，当然这个过程是第一阶段提交，只是官员们初步接受贿赂而已。故事中的比特币是编号，议题是 value。

这个过程保证了在某一时刻，某一个 proposer 的议题会形成一个多数派进行初步支持

4.1.3.3.2. 阶段二

5. 现在进入第二阶段提交，现在 proposer-1 小姐使用分身术(多线程并发)分了 3 个自己分别去找 3 位官员，最先找到了 1 号官员签合同，遭到了 1 号官员的鄙视，1 号官员告诉他 proposer-2 先生给了他 11 比特币，因为上一条规则的性质 proposer-1 小姐知道 proposer-2 第一阶段在她之后又形成了多数派(至少有 2 位官员的赃款被更新了);此时她赶紧去提款准备重新贿赂这 3 个官员(重新进入第一阶段)，每人 20 比特币。刚给 1 号官员 20 比特币，1 号官员很高兴初步接受了议题，还没来得及见到 2,3 号官员的时候

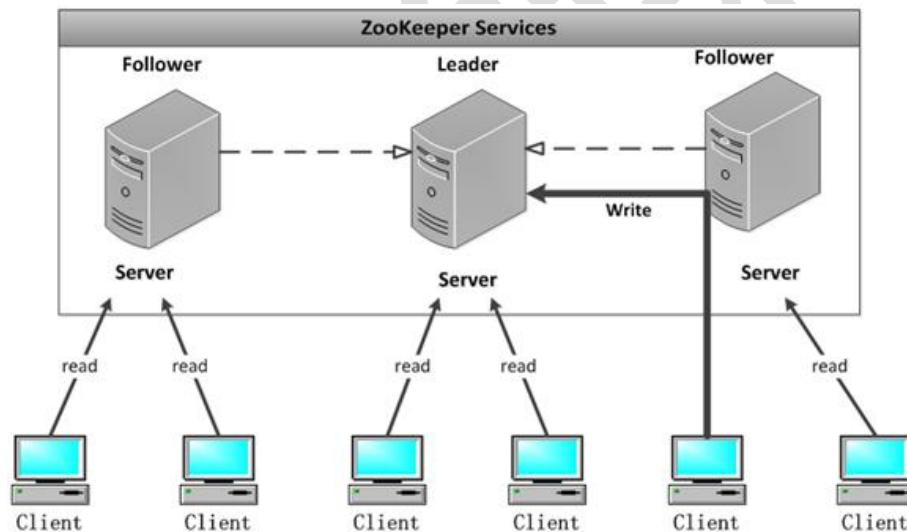
这时 proposer-2 先生也使用分身术分别找 3 位官员(注意这里是 proposer-2 的第二阶段)，被第 1 号官员拒绝了告诉他收到了 20 比特币，第 2,3 号官员顺利签了合同，这时 2,3 号官员记录 client-2 老板用了 11 比特币中标，因为形成了多数派，所以最终接受了 Client2 老板中标这个议题，对于 proposer-2 先生已经出色的完成了工作；

这时 proposer-1 小姐找到了 2 号官员，官员告诉她合同已经签了，将合同给她看，proposer-1 小姐是一个没有什么职业操守的聪明人，觉得跟 Client1 老板混没什么前途，所以将自己的议题修改为“Client2 老板中标”，并且给了 2 号官员 20 比特币，这样形成了一个多数派。顺利的再次进入第二阶段。由于此时没有人竞争了，顺利的找 3 位官员签合同，3 位官员看到议题与上次一次的合同是一致的，所以最终接受了，形成了多数派，proposer-1 小姐跳槽到 Client2 老板的公司去了。

总结：Paxos 过程结束了，这样，一致性得到了保证，算法运行到最后所有的 proposer 都投“client2 中标”所有的 acceptor 都接受这个议题，也就是说在最初的第二阶段，议题是先入为主的，谁先占了先机，后面的 proposer 在第一阶段就会学习到这个议题而修改自己本身的议题，因为这样没职业操守，才能让一致性得到保证，这就是 paxos 算法的一个过程。原来 paxos 算法里的角色都是这样的不靠谱，不过没关系，结果靠谱就可以了。该算法就是为了追求结果的一致性。

4.2. ZK 集群解析

4.2.1. Zookeeper 集群特点



前面一种研究的单节点，现在来研究下 zk 集群，首先来看下 zk 集群的特点。

- 顺序一致性
客户端的更新顺序与它们被发送的顺序相一致。
- 原子性
更新操作要么成功要么失败，没有第三种结果。
- 单一视图
无论客户端连接到哪一个服务器，客户端将看到相同的 ZooKeeper 视图。
- 可靠性

一旦一个更新操作被应用，那么在客户端再次更新它之前，它的值将不会改变。

- 实时性
连接上一个服务端数据修改，所以其他的服务器都会实时的跟新，不算完全的实时，有一点延时的
- 角色轮换避免单点故障
当 leader 出现问题的时候，会选举从 follower 中选举一个新的 leader

4.2.2. 集群中的角色

- Leader 集群工作机制中的核心
事务请求的唯一调度和处理者，保证集群事务处理的顺序性
集群内部个服务器的调度者(管理 follower, 数据同步)
- Follower 集群工作机制中的跟随者
处理非事务请求，转发事务请求给 Leader
参与事务请求 proposal 投票
参与 leader 选举投票
- Observer 观察者
3.30 以上版本提供，和 follower 功能相同，但不参与任何形式投票
处理非事务请求，转发事务请求给 Leader
提高集群非事务处理能力

4.2.3. Zookeeper 集群配置

1. 安装 jdk 运行 jdk 环境

上传 jdk1.8 安装包

2. 安装 jdk1.8 环境变量

vi /etc/profile

```
export JAVA_HOME=/usr/local/jdk1.8.0_181
export ZOOKEEPER_HOME=/usr/local/zookeeper
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
export PATH=$JAVA_HOME/bin:$ZOOKEEPER_HOME/bin:$PATH
```

刷新 profile 文件

source /etc/profile

关闭防火墙

3. 下载 zookeeper 安装包

```
 wget https://mirrors.tuna.tsinghua.edu.cn/apache/zookeeper/zookeeper-3.4.10/zookeeper-3.4.10.tar.gz
```

4. 解压 Zookeeper 安装包

```
 tar -zxvf zookeeper-3.4.10.tar.gz
```

5. 修改 Zookeeper 文件夹名称

```
 重命名: mv zookeeper-3.4.10 zookeeper
```

7. 修改 zoo_sample.cfg 文件

```
 cd /usr/local/zookeeper/conf  
 mv zoo_sample.cfg zoo.cfg  
 修改 conf: vi zoo.cfg 修改两处  
 (1) dataDir=/usr/local/zookeeper/data (注意同时在 zookeeper 创建 data 目录)  
 (2) 最后面添加  
 server.0=192.168.212.154:2888:3888  
 server.1=192.168.212.156:2888:3888  
 server.2=192.168.212.157:2888:3888
```

7. 创建服务器标识

服务器标识配置:

创建文件夹: mkdir data

创建文件 myid 并填写内容为 0: vi

myid (内容为服务器标识 : 0)

8. 复制 zookeeper

进行复制 zookeeper 目录到 node1 和 node2

还有/etc/profile 文件

把 node1、node2 中的 myid 文件里的值修改为 1 和 2

路径(vi /usr/local/zookeeper/data/myid)

9 启动 zookeeper

启动 zookeeper:

路径: /usr/local/zookeeper/bin

执行: zkServer.sh start

(注意这里 3 台机器都要进行启动)

状态: zkServer.sh

status(在三个节点上检验 zk 的 mode, 一个 leader 和两个 follower)

```
 scp -r /soft root@zk2:/
```

```
 scp -r /soft root@zk3:/
```

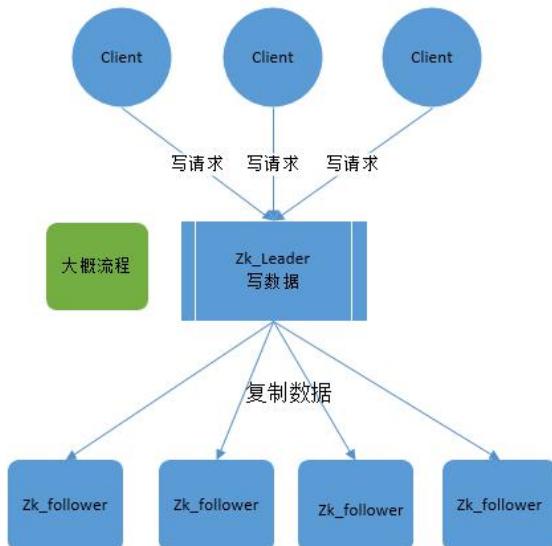
4.2.4. Zookeeper 集群一致性协议 ZAB 解析

4.2.4.1. 总览

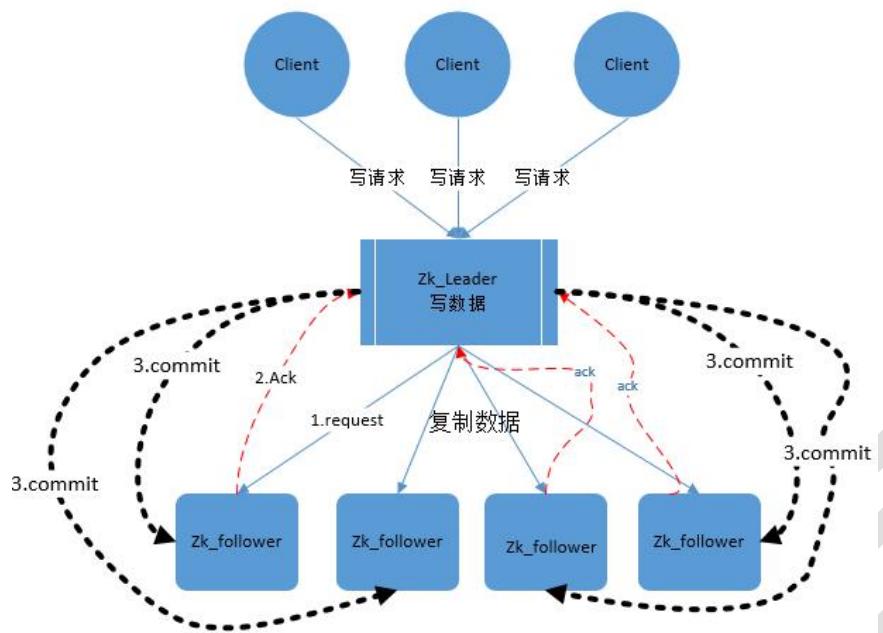
懂了 paxos 算法，其实 zab 就很好理解了。很多论文和资料都证明 zab 其实就是 paxos 的一种简化实现，但 Apache 自己的立场说 zab 不是 paxos 算法的实现，这个不需要去计较。

zab 协议解决的问题和 paxos 一样，是解决分布式系统的数据一致性问题

zookeeper 就是根据 zab 协议建立了主备模型完成集群的数据同步（保证数据的一致性），前面介绍了集群的各种角色，这说所说的主备架构模型指的是，在 zookeeper 集群中，只有一台 leader（主节点）负责处理外部客户端的事务请求（写操作），leader 节点负责将客户端的写操作数据同步到所有的 follower 节点中。



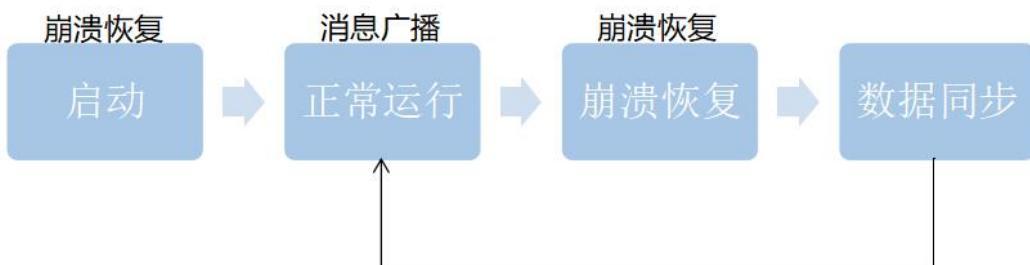
zab 协议核心是在整个 zookeeper 集群中只有一个节点既 leader 将所有客户端的写操作转化为事务（提议 proposal）。leader 节点再数据写完之后，将向所有的 follower 节点发送数据广播请求（数据复制），等所有的 follower 节点的反馈，在 zab 协议中，只要超过半数 follower 节点反馈 ok, leader 节点会向所有 follower 服务器发送 commit 消息，既将 leader 节点上的数据同步到 follower 节点之上。



发现，整个流程其实和 paxos 协议其实大同小异。说 zab 是 paxos 的一种实现方式其实并不过分。

Zab 再细看可以分成两部分。第一的消息广播模式，第二是崩溃恢复模式。

集群生命周期



正常情况下当客户端对 zk 有写的数据请求时，leader 节点会把数据同步到 follower 节点，这个过程其实就是消息的广播模式

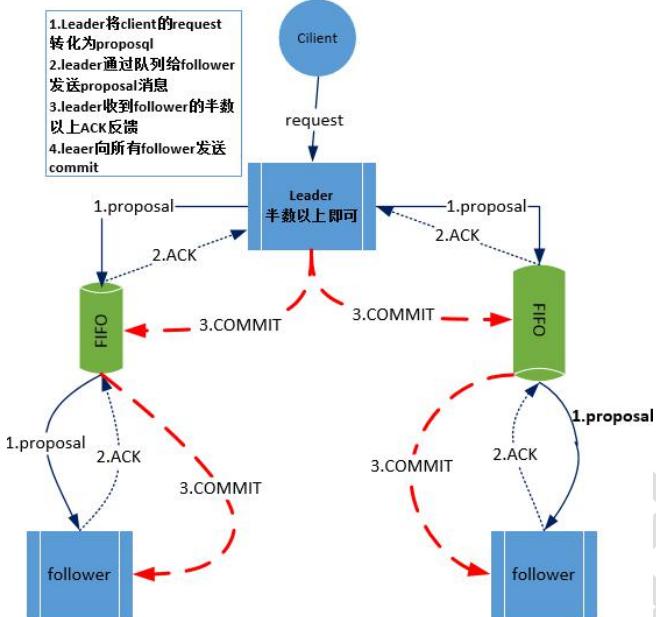
在新启动的时候，或者 leader 节点崩溃的时候会要选举新的 leader，选好新的 leader 之后会进行一次数据同步操作，整个过程就是崩溃恢复。

4.2.4.2. 消息广播模式

为了保证分区容错性，zookeeper 是要让每个节点副本必须是一致的

1. 在 zookeeper 集群中数据副本的传递策略就是采用的广播模式

2. Zab 协议中的 leader 等待 follower 的 ack 反馈，只要半数以上的 follower 成功反馈就好，不需要收到全部的 follower 反馈。



zookeeper 中消息广播的具体步骤如下：

1. 客户端发起一个写操作请求
2. Leader 服务器将客户端的 request 请求转化为事物 proposql 提案，同时为每个 proposal 分配一个全局唯一的 ID，即 ZXID。
3. leader 服务器与每个 follower 之间都有一个队列，leader 将消息发送到该队列
4. follower 机器从队列中取出消息处理完(写入本地事物日志中)毕后，向 leader 服务器发送 ACK 确认。
5. leader 服务器收到半数以上的 follower 的 ACK 后，即认为可以发送 commit
6. leader 向所有的 follower 服务器发送 commit 消息。

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。这也是 CAP/BASE 最终实现一致性的一个体现。

回顾一下：前面还讲了 2pc 协议，也就是两阶段提交，发现流程 2pc 和 zab 还是挺像的，zookeeper 中数据副本的同步方式与二阶段提交相似但是却又不同。二阶段提交的要求协调者必须等到所有的参与者全部反馈 ACK 确认消息后，再发送 commit 消息。要求所有的参与者要么全部成功要么全部失败。二阶段提交会产生严重阻塞问题，但 paxos 和 zab 没有这要求。

为了进一步防止阻塞，leader 服务器与每个 follower 之间都有一个单独的队列进行收发消息，使用队列消息可以做到异步解耦。leader 和 follower 之间只要往队列中发送了消息即可。如果使用同步方式容易引起阻塞。性能上要下降很多

4.2.4.3. 崩溃恢复



4.2.4.3.1. 背景（什么情况下会崩溃恢复）

zookeeper 集群中为保证任何所有进程能够有序的顺序执行，只能是 leader 服务器接受写请求，即使是 follower 服务器接受到客户端的请求，也会转发到 leader 服务器进行处理。

如果 leader 服务器发生崩溃(重启是一种特殊的崩溃，这时候也没 leader)，则 zab 协议要求 zookeeper 集群进行崩溃恢复和 leader 服务器选举。

4.2.4.3.2. 最终目的（恢复成什么样）

ZAB 协议崩溃恢复要求满足如下 2 个要求：

确保已经被 leader 提交的 proposal 必须最终被所有的 follower 服务器提交。

确保丢弃已经被 leader 出的但是没有被提交的 proposal。

新选举出来的 leader 不能包含未提交的 proposal，即新选举的 leader 必须都是已经提交了的 proposal 的 follower 服务器节点。同时，新选举的 leader 节点中含有最高的 ZXID。这样做的好处就是可以避免了 leader 服务器检查 proposal 的提交和丢弃工作。

- 每个 Server 会发出一个投票,第一次都是投自己。投票信息：(myid, ZXID)
- 收集来自各个服务器的投票
- 处理投票并重新投票，处理逻辑：优先比较 ZXID,然后比较 myid
- 统计投票，只要超过半数的机器接收到同样的投票信息，就可以确定 leader
- 改变服务器状态

问题：为什么优先选大的 zxid

4.2.5. Java 客户端连接集群

```
ZkClient client = new ZkClient("host1,host2,host3,host4,host5");
```

ZK 连接集群很简单，只需要把连接地址用逗号分隔就好。

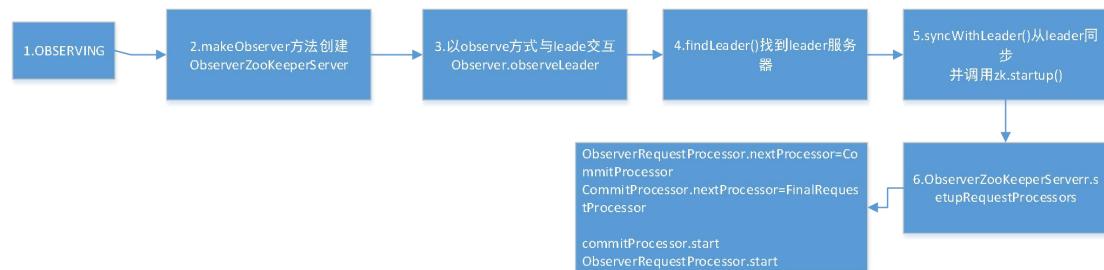
5. 集群源码解读

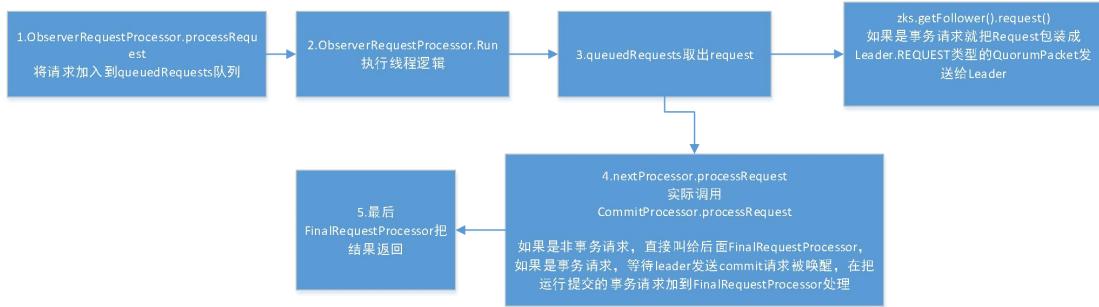
5.1. 集群模式

5.1.1. 数据同步总流程

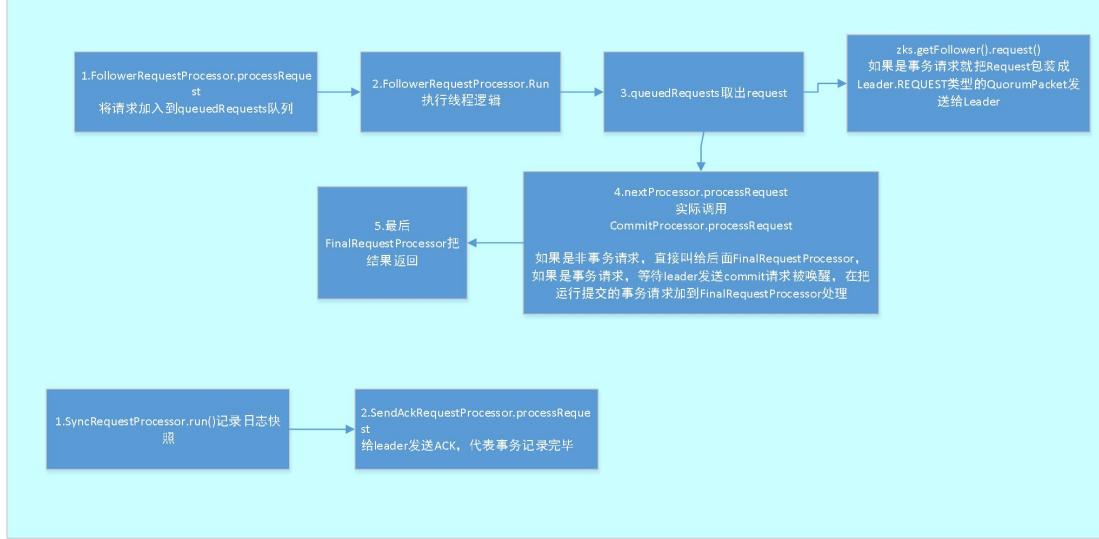
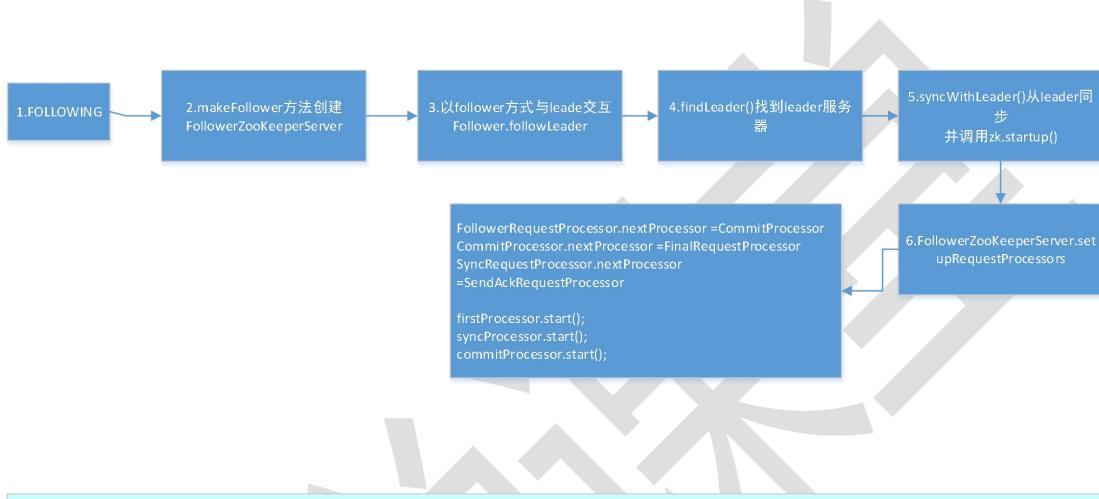


5.1.1.1. OBSERVING

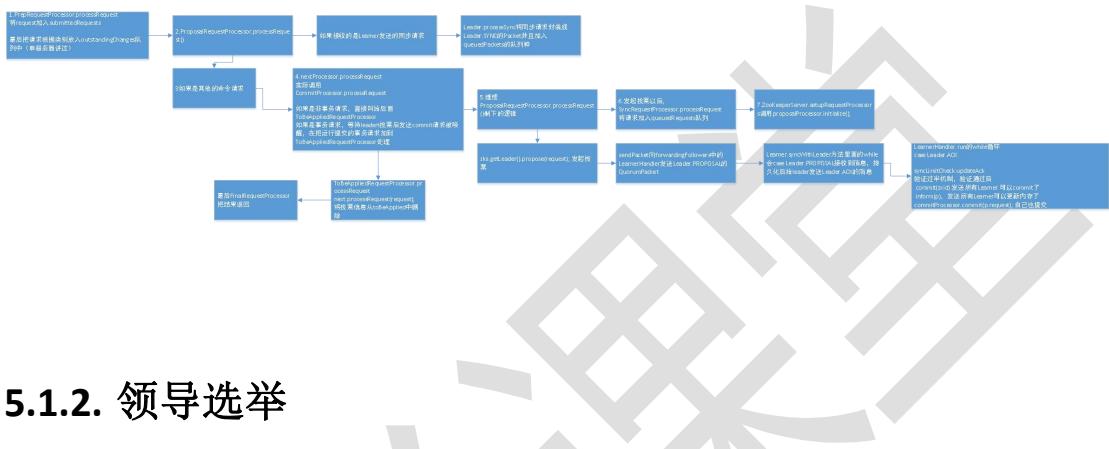
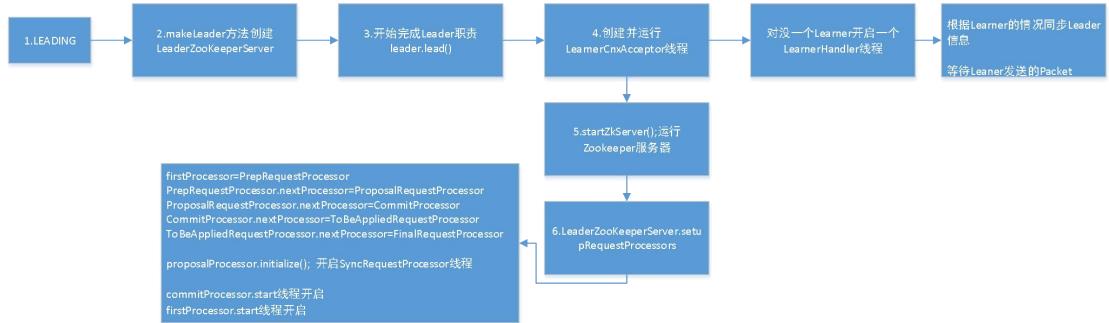




5.1.1.2. FOLLOWING

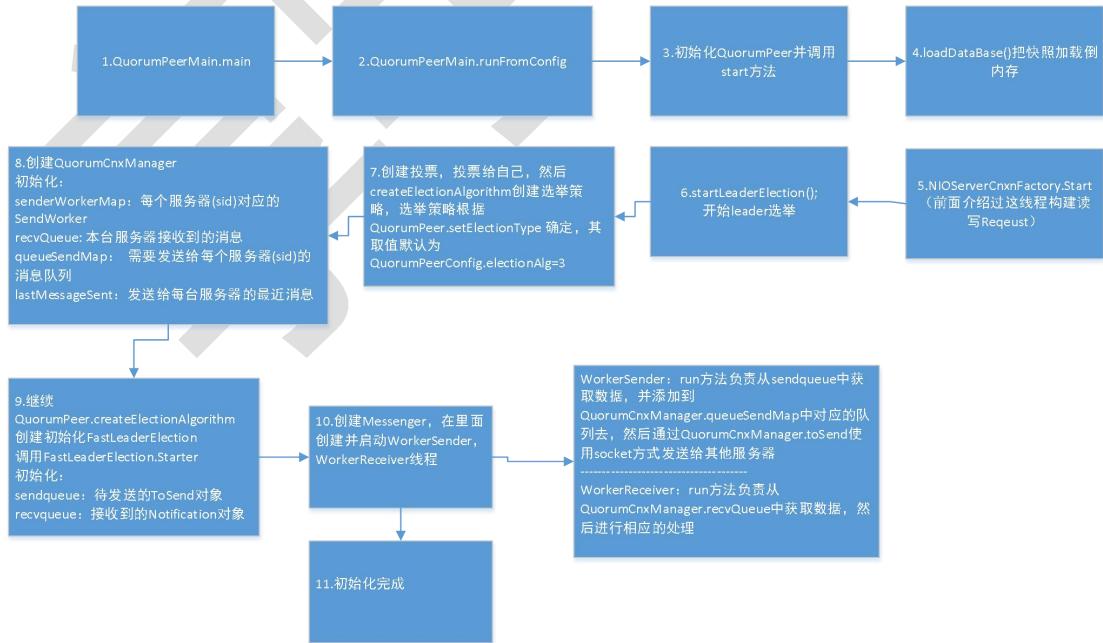


5.1.1.3. LEADING



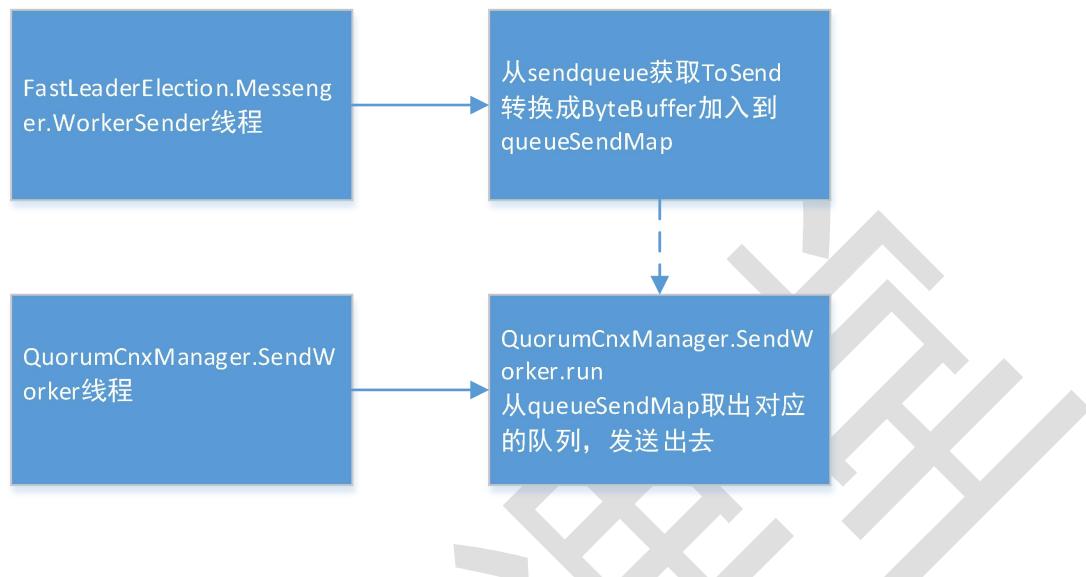
5.1.2. 领导选举

5.1.2.1. 领导选举初始化

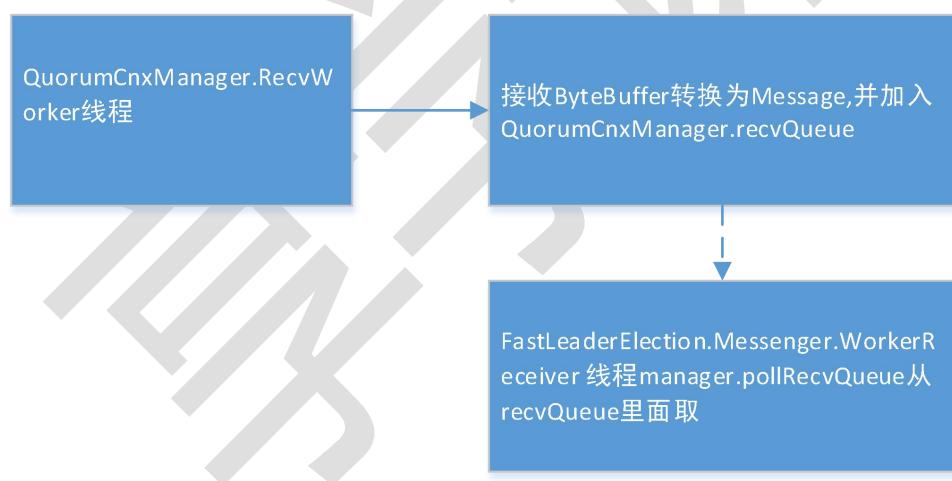


5.1.2.2. 线程逻辑

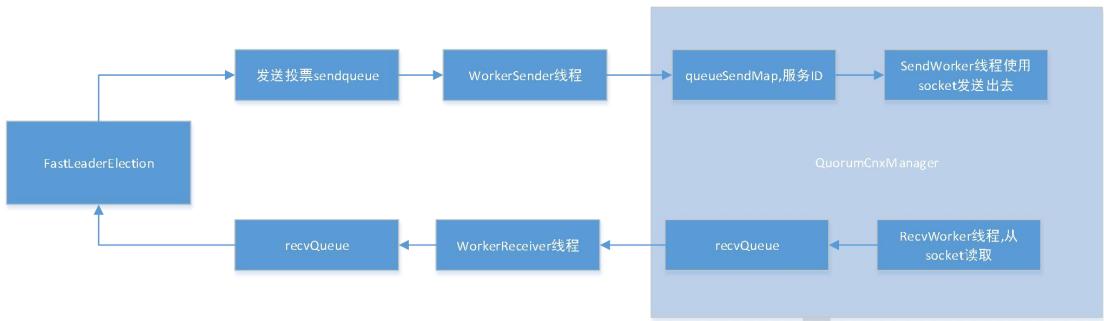
5.1.2.2.1. 发送请求用到的线程



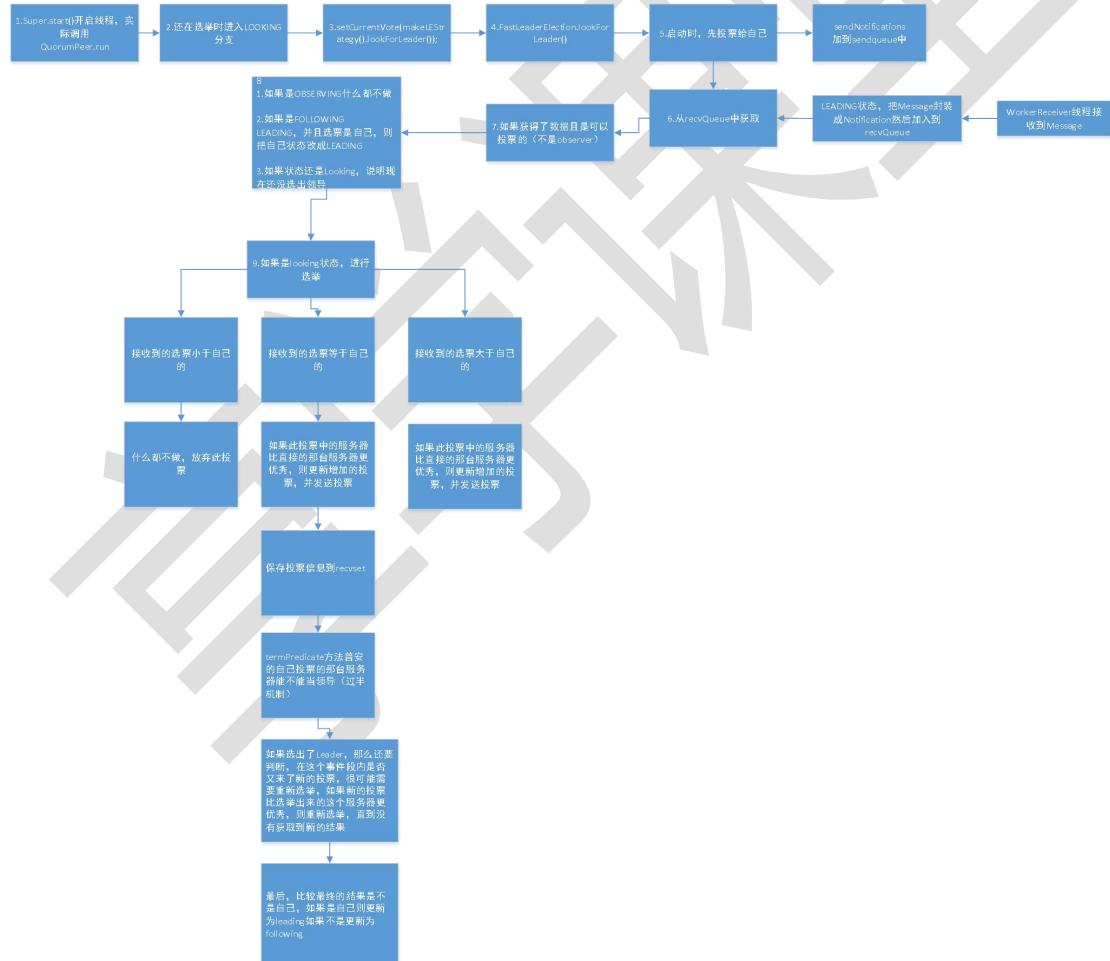
5.1.2.2.2. 接收请求用到的线程



5.1.2.2.3. 发送/接收总览



5.1.2.3. 领导选举算法



6. 常见面试题

6.1. ZAB 协议是什么？

ZAB 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。

ZAB 协议包括两种基本的模式：崩溃恢复和消息广播。

当整个 zookeeper 集群刚刚启动或者 Leader 服务器宕机、重启或者网络故障导致不存在过半的服务器与 Leader 服务器保持正常通信时，所有进程（服务器）进入崩溃恢复模式，首先选举产生新的 Leader 服务器，然后集群中 Follower 服务器开始与新的 Leader 服务器进行数据同步，当集群中超过半数机器与该 Leader 服务器完成数据同步之后，退出恢复模式进入消息广播模式，Leader 服务器开始接收客户端的事务请求生成事物提案来进行事务请求处理。

6.2. Znode 有哪几种类型

PERSISTENT-持久节点

除非手动删除，否则节点一直存在于 Zookeeper 上

EPHEMERAL-临时节点

临时节点的生命周期与客户端会话绑定，一旦客户端会话失效（客户端与 zookeeper 连接断开不一定会话失效），那么这个客户端创建的所有临时节点都会被移除。

PERSISTENT_SEQUENTIAL-持久顺序节点

基本特性同持久节点，只是增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。

EPHEMERAL_SEQUENTIAL-临时顺序节点

基本特性同临时节点，增加了顺序属性，节点名后边会追加一个由父节点维护的自增整型数字。

6.3. ACL 权限控制机制

UGO (User/Group/Others)

目前在 Linux/Unix 文件系统中使用，也是使用最广泛的权限控制方式。是一种粗粒度的文件系统权限控制模式。

ACL (Access Control List) 访问控制列表

包括三个方面：

权限模式（Scheme）

IP: 从 IP 地址粒度进行权限控制

Digest: 最常用，用类似于 `username:password` 的权限标识来进行权限配置，便于区分不同应用来进行权限控制

World: 最开放的权限控制方式，是一种特殊的 digest 模式，只有一个权限标识“`world:anyone`”

Super: 超级用户

授权对象

授权对象指的是权限赋予的用户或一个指定实体，例如 IP 地址或是机器名。

权限 Permission

CREATE: 数据节点创建权限，允许授权对象在该 Znode 下创建子节点

DELETE: 子节点删除权限，允许授权对象删除该数据节点的子节点

READ: 数据节点的读取权限，允许授权对象访问该数据节点并读取其数据内容或子节点列表等

WRITE: 数据节点更新权限，允许授权对象对该数据节点进行更新操作

ADMIN: 数据节点管理权限，允许授权对象对该数据节点进行 ACL 相关设置操作

6.4. ZK 的角色有哪些

Leader

事务请求的唯一调度和处理者，保证集群事务处理的顺序性

集群内部各服务的调度者

Follower

处理客户端的非事务请求，转发事务请求给 Leader 服务器

参与事务请求 Proposal 的投票

参与 Leader 选举投票

Observer

3.3.0 版本以后引入的一个服务器角色，在不影响集群事务处理能力的基础上提升集群的非事务处理能力

处理客户端的非事务请求，转发事务请求给 Leader 服务器

不参与任何形式的投票

6.5. ZK Server 的工作状态有哪些，能否具体描述

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。

6.6. ZK 的 watch 机制是否永久

否。

6.7. ZK 的常见客户端有哪些

java 客户端: zk 自带的 zkclient 及 Apache 开源的 Curator

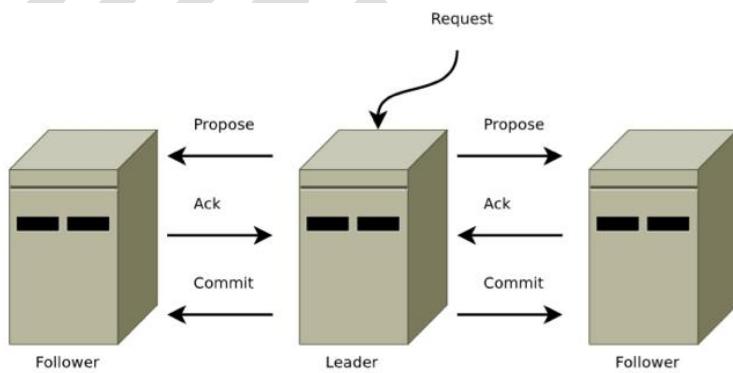
6.8. 分布式锁用 zk 怎么实现

有两种实现方式, 具体看课程代码

6.9. Zk 默认的通信框架是什么

Nio, 可以改成 netty

6.10. 消息广播的流程



- 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, 叫做: zxid, 通过 zxid 的大小比较即可实现因果有序这一特性。
- Leader 通过先进先出队列(通过 TCP 协议来实现, 以此实现了全局有序这一特性)将带有 zxid 的消息作为一个提案(proposal)分发给所有 follower。
- 当 follower 接收到 proposal, 先将 proposal 写到硬盘, 写硬盘成功后再向 leader 回

一个 ACK。

- 当 leader 接收到合法数量的 ACKs 后，leader 就向所有 follower 发送 COMMIT 命令，会在本地执行该消息。
- 当 follower 收到消息的 COMMIT 命令时，就会执行该消息

6.11. 领导选举的流程



- 每个 Server 会发出一个投票,第一次都是投自己。投票信息: (myid, ZXID)
- 收集来自各个服务器的投票
- 处理投票并重新投票, 处理逻辑: 优先比较 ZXID,然后比较 myid
- 统计投票, 只要超过半数的机器接收到同样的投票信息, 就可以确定 leader
- 改变服务器状态