# 分布式架构

## 集中式系统

大型主机时代的到来，集中式的计算机系统架构称为了主流；由于大型主机良好的性能和稳定性，其在单机处理能力方面的优势非常明显，it系统快速进入了集中式处理阶段，对应的计算机应用系统称为集中式系统

集中式系统就是由一台或多台计算机组成的中心节点，数据集中存储在这个中心节点上，并在整个系统的所有业务单元都集中部署在这个中心节点，系统的所有功能均由其集中处理；最大特点就是部署结构简单，无需考虑多节点之间的协助

存在的问题：

1. 成本较高：大型主机非常昂贵，大型主机的人才培养成本也非常高
2. 单点问题，一旦出现故障，整个系统就无法使用
3. 随着业务不断发展用户访问量提高应用系统的规模也会不断扩大，在单一大型主机上进行系统的扩容比较困难

## 分布式系统

一个硬件或者软件组件分布在不同的网络计算机上，彼此之间只是通过消息传递进行通信和协调的系统

### 特征

分布性：分布式系统中的计算机都会分布可以分布在不同的空间，机器可能会随时实际情况迁移

对等性：分布式系统中的计算机没有主从之分，即没有控制整个系统的主机，所有计算机节点都是对等的；

副本 ：是分布式系统中对数据和服务提供的一种冗余方式；为了提供高可用的服务，会对数据和服务进行副本处理，

数据副本：指的是不同节点上持久化的同一份数据，当某一节点上数据丢失，可用从其他节点上获取，这解决了分布式系统数据丢失的问题；

服务副本：多节点提供相同的服务，每个节点都有接收和处理外部请求的功能

并发性：分布式系统中多节点可能同时操作同一共享资源

缺乏全局时钟：分布式系统各节点之间是通过消息来相互通信，那么就缺乏一个全局的时钟序列控制

故障总会发生：考虑到的异常在运行到一定阶段后可能会发生，还会出现许多设计时未能考虑到的异常

### 存在的问题

通信异常：分布式系统节点之间需要进行网络通信，每次网络通信都会伴随着**网络不可用的风险**

网络分区：分布式系统的所有节点中只有部分节点之间可以进行正常网络通信，另一些节点不能

网络分区是指在分布式系统中，不同节点之间分布于不同的子网络（处于不同的网络环境------机房，异地网络），由于特殊原因导致这些子网络之间通信故障，但是子网络内部是可以通信的，从而导致整个网络环境被切分为若干个孤立的区域，（组成分布式系统的每一个节点的加入和退出都可以看着是一个特殊的网络分区）；这一现象称为网络分区，又叫脑裂；分布式系统会出现局部的小集群，在极端情况下，这些小集群会独立完成原本需要整个分布式系统才能完成的功能，包括对数据的事务处理，这就对分布式一致性提供了非常大的调整

三态：分布式环境中，网络状况可能会出现各种各样的问题，因此分布式系统每次请求与响应存在特有的“三态”概念，成功，超时，失败；

单系统环境下只存在成功或失败2中情况，分布式系统中处理请求和响应的成功或失败之外，因为网络异常出现超时的情况，

由于网络原因，该请求（消息）并没有发送到接收方，而是在消息发送的过程中出现了消息丢失现象；

接收方接收消息后，并进行了处理，但是在响应反馈到发送方的过程中，发送了消息丢失

当出现消息超时时，消息发送方无法确认当前请求是否已经被成功处理了。

节点故障：分布式系统中某个服务节点宕机了

### ACID

1. 原子性:对数据库的一系列的操作，要么都成功，要么都失败，不可能出现部分成功或者部分失败的情况

事务的操作全部失败：事务回滚。

1. 一致性：指的是数据库的完整性约束没有被破坏，**事务执行的前后都是合法的数据状态**。比如主键必须是唯一的，字段长度符合要求。除了数据库自身的完整性约束，还有一个是用户自定义的完整性。

比如说转账的这个场景，A 账户余额减少 1000，B 账户余额只增加了 500，这个时候因为两个操作都成功了，按照我们对原子性的定义，它是满足原子性的， 但是它没有满足一致性，因为它导致了会计科目的不平衡。

1. 隔离性：有了事务的定义以后，在数据库里面会有很多的事务同时去操作同一张表或者同一行数据，必然会产生一些并发或者干扰的操作；

**隔离性的定义，就是这些很多个的事务，对表或者行的并发操作，应该是透明的，互相不干扰的**

1. 持久性：对数据库的任意的操作，增删改，只要事务提交成功，那么结果就是永久性的，不可能因为我们系统宕机或者重启了数据库的服务器，它又恢复到原来的状态了

隔离级别

1. 未提交读（read uncommitted）:一个事务可以读取到其他事务未提交的数据，会出现脏读，所以叫做 RU，它没有解决任何的问题。
2. 已提交读（read committed）:一个事务只能读取到其他事务已提交的数据，不能读取到其他事务未提交的数据，它解决了脏读的问题，但是会出现不可重复读的问题。
3. 可重复读（repeatable read）:它解决了不可重复读的问题，也就是在同一个事务里面多次读取同样的数据结果是一样的，但是在这个级别下，没有定义解决幻读的问题。
4. 串行化（serializable）:所有的事务都是串行执行的，也就是对数据的操作需要排队，已经不存在事务的并发操作了，所以它解决了所有的问题。

### 分布式事务

一个分布式事务中会涉及到多个数据源或者业务系统的操作

分布式事务指的是事务的参与者，支持事务的服务器，资源服务器，以及事务的管理器分别位于分布式系统的不同节点上；

也就是说一次大的操作是由多个小的操作组成的，这些小操作分布于不同的服务器上，且属于不用的系统，分布式事务就是要保证这些小操作要么全部成功要么全部失败，

分布式事务本质上就是保证不同应用数据库的数据一致性；（跨行转账）

在分布式系统中如果要实现一套严格满足ACID特性的分布式事务，很可能出现的情况就是系统在可用性和严格的一致性之间出现冲突，分布式系统在可用性和一致性之间不存在一个完美的解决方案

### Cap定理

#### 定义

分布式系统不可能同时满足一致性C，可用性A，和分区容错性P这三项基本要求，最多只能满足2项

一致性：数据在多个副本之间是否保持一致的特性；在一致性的需求下，当一个系统数据更新操作完成后，应该保证系统副本之间的数据处于一致的状态；

如果一个副本更新了数据，另一个没有更新，那么获取这个未更新的副本数据就是老数据，就出现了分布式数据不一致的情况；

可用性：系统提供的服务必须处于可用的状态，对用户的请求总是能够在有限的时间内响应

有限时间内：系统必须在指定的时间内返回对应处理的结果，如果超过这个时间范围就被认定为不可用；不同系统对请求响应时间的要求不同，但是系统必须存在一个合理的响应时间

返回结果是可用性的另一个重要指标；系统在完成对用户请求处理后，返回一个正常的结果，正常的结果就是明确的告诉用户操作是成功还是失败了，而不是返回一个让用户感动困惑的结果（比如返回一个异常），那么系统就被认为是不可用的

分区容错性：分布式系统遇到任何网络分区故障的时候，仍然能够保证对外提供满足一致性和可用性的服务，除非整个网络都出现了故障

网络分区是指在分布式系统中，不同节点之间分布于不用的子网络（处于不同的网络环境------机房，异地网络），由于特殊原因导致这些子网络之间通信故障，但是子网络内部是可以通信的，从而导致整个网络环境被切分为若干个孤立的区域，（组成分布式系统的每一个节点的加入和退出都可以看着是一个特殊的网络分区）

#### 应用

一个分布式系统中不能同时满足CAP这3个需求，而只能满足其中两个，因此在使用cap定理的时候，我们需要抛出2其中一个

**放弃P**:意味着网络分区的问题，避免出现分区容错问题的一个简单的解决办法就是将所有的数据都放在一个分布式节点上，这样做虽然不能避免网络问题，但是不会出现网络分区带来的影响，放弃分区容错性也就是意味着放弃了系统的可扩展性

**放弃A:**当遇到网络分区或者其他故障时，那么受到影响的服务需要等待一定时间，因此在等待时间之内系统是无法向外提供服务的，即不可用

**放弃C**:这里的放弃一致性不是完全不需要数据的一致性，而是放弃数据的强一致性，实现数据的最终一致性；这样做就不能保证数据的实时性一致，但最终会达到一致性的，这就需要提供一个时间窗口，在这个时间范围内到达数据的最终一致性，具体多久就要看系统的设计；

对分布式系统而言**分区容错性是一个最基本的要求，必须满足的**；因为分布式系统中的组件必然是部署到不同的节点上，否则也就不能成为是分布式系统了；因此必然会出现子网络（面临不同的网络环境），所以分区容错就是分布式系统必须解决的问题，所以架构师一般根据业务需求在可用性和一致性之间寻找平衡

### Base理论

Base是basically available基本可用，soft state软状态和eventually consistent最终一致性三个短语的简写；base理论是对cap中一致性和可用性权衡的结果，核心思想是无法做到强一致性时，每个应用都可以根据自身的业务特点，采用适当的方式是系统达到最终一致性

**基本可用**：在分布式系统出现故障时允许损失部分可用性，但不意味着系统不可用

响应时间上的损失：比如原本的查询时间是0.5秒，出现故障后的查询时间是2秒

功能上的损失：在电商大促销时，访问量激增，为保护系统的稳定性，部分用户会被引导到一个降级页面(比如提示当前访问过多等）

**软状态：**指的是数据在系统中存在的中间状态，该中间状态不会影响系统的可用性，允许系统不同数据副本节点之间同步数据存在延迟；

**最终一致性：**系统所有数据副本在经过一定时间同步之后，最终能够达到一个一致的状态；本质就是保证系统数据的最终能够达到一致状态，不用保证实时的数据强一致性

最终一致性存在以下几种变种

|  |  |
| --- | --- |
|  | **说明** |
| 因果一致性 | 如果进程 A 通知进程 B 它已更新了一个数据项，那么，进程 B 的后续访问将返回更新后的值，且一次写入将保证取代前一次写入。与进程 A 无因果关系的进程 C 的访问遵守一般的最终一致性规则。 |
| 读己之所写（Read-Your-Writes） 一致性 | 当进程 A 自己更新一个数据项之后，它总是访问到更新过的值，且不会看到旧值。这是因果一致性模型的一个特例。 |
| 会话（Session）—致性 | 这是上一个模型的实用版本，它把访问存储系统的进程放到会话的上下文中。只要会话还存在，系统就保证“读己之所写”一致性。如果由于某些失败情形令会话终止，就要建立新的会话，而且系统保证不会延续到新的会话。 |
| 单调（Monotonic）读一致性 | 如果进程已经看到过数据对象的某个值，那么任何后续访问都不会返回在那个值之前的值。 |
| 单调写一致性 | 系统保证来自同一个进程的写操作顺序执行。 |

# 一致性协议

解决分布式一致性问题的协议和算法；当事务操作需要跨越多个分布式节点的时候，为了保证事务的acid特性，需要引入一个’**协调者’**组件来同意调度所有分布式节点的执行逻辑，被调度的节点就称为**‘参与者’，协调者负责调度参与者的行为，并最终决定参与者是否要把事务真正进行提交**

## 2pc

2-phase-commit二阶段提交：基于分布式系统架构下的所有节点在进行事务处理过程中能够保持原子性和一致性而设计的一种算法，用来保证分布式系统数据的一致性；

2pc将事务的过程分成了两个阶段来进行处理：

### 阶段一：提交事务请求

1. 事务询问：协调者向所有事务参与者发送事务内容，询问是否可以执行事务提交操作，并开始等待参与者的响应
2. 执行事务：各参与者节点执行事务操作，并将undo和redo信息记入事务日志中。
3. 参与者向协调者反馈事务询问的响应：如果参与者成功执行事务，那么就反馈给协调者Yes响应，表示事务可以执行，如果参与者没有成功执行事务，就反馈No响应，表示事务不执行

### 阶段二：执行事务提交

在二阶段中协调者会根据参与者的反馈情况来决定最终是否可以进行事务提交操作，正常情况下包含2中情况

**提交事务**

**当参与者反馈的都是yes响应，那么就会执行提交事务**

1. 发送提交请求

协调者向所有参与者发送commit请求

1. 事务提交

参与者收到commit请求后，会正式的执行事务提交操作，并在事务提交完成后释放整个事务期间占据的资源

1. 反馈事务提交结果

参与者在提交事务之后想协调者发送AcK消息

1. 完成事务

协调者接收到所有参与者返回的ack消息后，完成事务

**中断事务**

**当有任何一个参与者反馈的是no响应，那么就会中断事务**

1. 发送回滚请求

协调者向参与者发送rollback请求

1. 事务回滚

参与者收到rollback请求后，会正式执行事务回滚，并在事务回滚后，释放事务占据的资源

1. 反馈事务回滚结果

参与者回滚事物之后想协调者发送ack消息

1. 中断事务

协调者接收到ack消息后，完成事务中断

**二阶提交的优缺点**

**优点：原理简单，实现方便**

**缺点：同步阻塞、单点问题，脑裂，太过保守**

同步阻塞：执行过程中，参与该事务操作的逻辑都处于阻塞状态，也就是说，各个参与者都要等待其他参与者的响应，这个等待过程将无法进行其他操作，这会极大的限制分布式系统的性能；

单点问题：如果协调者出现了问题，那么整个二阶段提交流程将无法运转，如果协调者在二阶段出现问题的话，那么其他参与者将会一直处于锁定事务资源的状态中，无法完成事务操作

脑裂（网络分区）：数据不一致，在二阶段提交commit后，部分参与者网络故障，只有部分参与者收到了commit请求，收到commit请求的参与者会提交事务，没有收到的就不会提交事务，整个分布式系统就出现了数据不一致的情况

太过保守：没有完善的容错机制，任何一个节点失败都会导致整个事务的失败；进行事务是否提交 询问的过程中，参与者出现故障，始终无法获取所有参与者的响应消息，那么协调者只能依靠自身的超时机制来决定是否中断事务，这种策略太保守了

## 3pc

三阶段提交，是二阶提交的改进版，将二阶提交的事务提交请求过程一分为二，形成CanCommit，preCommit，do commit三个阶段组成的事务处理协议

### 阶段一：cancommit

1. 事务询问

协调者向所有参与者发送一个包含事务内容的canCommit请求,询问是否可以执行事务提交操作，并开始等待各参与者响应

1. 各个参与者向协调者反馈事务询问的响应

参与者接收到canCommit请求后，如果认为自身可以顺利执行事务，就反馈Yes响应，否则反馈No响应；

### 阶段二：Precommit

协调者根据参与者反馈的情况来决定是否可以进行事务的preCommit，正常情况有2种可能

执行事务预提交

1. 发送预提交请求

协调者向参与者发送preCommit请求，进入preCommit阶段

1. 事务预提交

参与者接收到preCommit后，会执行事务操作，并将undo和redo信息记录到事务日志中

1. 参与者向协调者反馈预事务执行的响应

参与者成功执行了事务操作，那么就会反馈给协调者Ack响应，同时等待最终的提交命令：commit(提交)或者abort（终止）

中断事务

当参与者有一个在canCommit阶段返回了No消息，或者在等待超时后，协调者无法接受到所有参与者的反馈结果，那么就会执行中断

1. 发送中断事务请求

协调者向所有参与者节点发送abort请求

1. 中断事务

所有参与者中断事务，无论是收到协调者的abort请求，或者还是在等待协调者发送请求中出现超时的参与者都中断事务

### 阶段三：docommit

这个阶段将会真正进行事务提交，会存在以下2种情况：

**执行提交**

1. 发送提交请求

进入这个阶段，当协调者处于正常工作状态，并且协调者接收到所有参与者的ack响应，那么就是从预提交转换为正式提交状态，向参与者发送doCommit请求

1. 事务提交

参与者接收到doCommit请求后，会正式执行事务提交操作，并在完成提交之后释放整个事务中占用的事务资源

1. 反馈事务提交结果

参与者在事务完成之后，相协调者发送ack消息

1. 完成事务

协调者接收到所有参与者反馈的ack消息后，完成事务

**中断事务**

1. 发送中断请求

协调者向参与者发送abort请求

1. 事务回滚

参与者接收到abort请求后，利用在二阶段的undo日志信息进行事务回滚，并在完成事务回滚之后释放事务占据的资源

1. 反馈事务回滚结果

参与者完成事务回滚之后，向协调者发送Ack消息

1. 中断事务

协调者接收到所有参与者发送的Ack消息后，中断事务

进入第三阶段之后，可能会存在2种故障

协调者出现问题

协调者和参与者之间出现了网络故障

无论哪种情况，参与者都会在等待超时后继续进行事务提交

### 优缺点

优点：三阶段提交和二阶段相比最大的优点就是降低了参与者的阻塞范围，并且能够在单点故障之后继续达成一致

缺点：参与者在接收到preCommit消息后，如果网络出现了分区协调者和参与者之间无法通信，在这种情况下参与者仍会进行事务的提交，这样必然出现的不一致性

## Paxos算法

# Zookeeper

## 什么是zookeeper

Zookeeper是一个典型的分布式数据一致性的解决方案，分布式应用程序可以基于它实现比如数据发布/订阅，负载均衡，命名服务，分布式协调/通知，集群管理，master选举，分布式锁和分布式队列等功能

Zookeeper致力于提供一个高性能，高可用，具有严格的顺序访问控制能力的分布式协调服务

Zookeeper提供了一个共享的，**树型结构的名字空间**来进行相互协调，树形结构的名字空间是指zookeeper服务内存中的一个数据模型，是由一系列被称为znode的数据节点组成的，数据模型类似一个文件系统，znode之间的层级关系，就像文件系统的目录结构一样；zookeeper将全量数据放在内存中的，以此来实现提高服务吞吐，减少延迟的目的

### Zookeeper的特性

顺序一致性：从客户端发起的请求最终将会严格按照顺序应用到zookeeper中去

原子性：整个集群所有的机器都成功应用了某一事务，要么都应用了，要么都没有应用，不会出现不会出现集群中部分机器应用了事务，部分没有应用事务的情况

单一视图：无论客户端连接的是哪个zookeeper服务器，其看到的服务器数据模型都是一致的；集群中所有机器上的数据模型都是一样的

可靠性：一旦服务端成功的应用了一个事务，并完成客户端响应，那么该事务引起的数据变化就会被一直保存下来，除非另一个事务引起了数据的变更

实时性：一旦事务成功，那么客户端就会立刻从服务器上获取最新的数据(保证一定时间内数据的实时性，接近实时)

### 集群角色

Leader，follower，observer

zookeeper集群中的所有机器通过leader选举过程来选举出一台服务器被称为leader的机器，leader服务为客户端提供了**读和写的服务**

Follower和Observer都能够**提供读服务**，唯一区别是Observer不参与leader选举，也不参与写操作的“过半写成功“策略

#### Leader

leader是zookeeper集群的核心。

1. 事务请求的唯一调度者和处理者，保证集群事务处理的顺序性
2. 集群内部各个服务器的调度者

#### follower

1. 处理客户端非事务请求，以及转发事务请求给leader服务器
2. 参与事务请求提议（proposal）的投票（客户端的一个事务请求，需要半数服务器投票通过以后才能通知leader commit； leader会发起一个提案，要求follower投票）
3. 参与leader选举的投票

#### observer

1. 观察zookeeper集群中最新状态的变化并将这些状态同步到observer服务器上
2. 增加observer不影响集群中事务处理能力，同时还能提升集群的非事务处理能力

### 会话

Session是指客户端会话；

在zookeeper中一个客户端连接是指客户端和服务端之间的一个长连接；Zookeeper对外的服务器端口是2181，客户端启动后就会与服务端建立连接

从第一次连接开始，客户端会话生命周期就开始了，通过这个连接客户端能通过心跳检测与服务器保存会话，能够向服务端发送请求并接受请求，还能够通过该连接接受来自服务器的watcher事件通知

Session timeout: 设置客户端会话的超时时间，客户端断开连接后，在session timeout时间之内重新连接到zookeeper，那么之前的会话就任然有效

### 数据节点(znode)

Zookeeper的数据模型中的数据单元，称之位数据节点---znode；zookeeper将所有的数据存放在内存中，数据模型就是一棵树(znode tree),由斜杠(/)进行分割的路径，就是一个znode,每个znode上都会保存自己的数据内容，同时还会保存一系列的属性信息

Znode分为**持久节点和临时节点**

持久节点就是指一旦创建就一直保存在zookeeper机器上，除非执行节点删除操作；

临时节点：生命周期和客户端会话绑定，一旦客户端会话失效，临时节点也就不存在了，

Zookeeper还允许为每个节点添加一个特殊属性：sequential，一旦这个节点被标记上这个属性，那么这个节点被创建的时候，zookeeper就会节点名后增加一个整型数字，

这个整型数字是父节点维护的自增数字

### 版本

对于每个ZNode zookeeper都会为其维护一个叫做Stat的数据结构，stat记录这个节点的3个数据版本，分别是version(当前ZNode的版本)，cversion(当前ZNode子节点版本)，aversion(当前节点的ACL版本)

### Watcher

事件监听器

在节点上注册watcher，在一些特定事件发生后触发

### ACL

采用access control list（ACL）策略来进行权限控制；zookeeper的5种权限

CREATE：创建子节点

READ：获取节点和子节点列表权限

WRITE：更新节点数据的权限

DELETE：删除子节点权限

ADMIN：设置节点的ACL权限

## Zab协议

Zookeeper并没有使用paxos算法，而是使用一种称为zookeeper atomic broadcast（zab）的协议作为数据一致性的核心算法

Zab协议是为分布式协调服务zookeeper设计的一种支持崩溃恢复的原子广播协议，在zookeeper中依赖zab协议来实现分布式数据一致性，基于该协议，zookeeper实现了一种准备模式的系统架构来保持集群中各个副本之间的数据一致性

Zookeeper使用一个单一的主进程来接收并处理客户端的所有事务请求，并采用zab的原子广播协议，将服务器数据的状态变更以事务proposal的形式广播到所有的副本进程中上去；Zab协议的这个主备模型架构保证了同一时刻集群中只能有一个主进程来广播服务器的状态变更，因此能够更好的处理客户端大量的并发请求

所有的zookeeper服务器数据变更的事务请求，必须由一个全局唯一的服务器来协调处理，这样的服务器称为Leader服务器，而余下的称为follower服务器；leader服务器负责将一个客户端事务请求转换成一个事务提议，并将该事务提议分发给集群中的所有follower服务器，之后leader服务器就等follower服务器反馈，一旦超过半数的follower服务器进行正确的反馈后，那么leader再次向follower发送commit消息，将之前的事务提议进行提交

### 协议介绍

Zab协议包括两种模式，分别是奔溃恢复和消息广播

当整个系统架构在**启动**过程中，或者是当leader服务器出现**网络中断、崩溃退出和重启**等异常情况时，ZAB协议就进入恢复模式选举新的服务器作为leader，同时集群中已有过半的机器和leader进行了数据同步（保证leader和集群中过半的集群的数据一致）之后，ZAB协议就退出恢复模式；（选举出leader，然后进行数据同步）

当集群中已有过半的follower服务器完成了和leader同步那么整个服务架构就进入**消息广播模式**；

当一个遵守ZAB协议的服务器进入集群中，如果存在leader服务器在负者进行消息广播，那么新加入的服务器就会自觉的进入数据恢复模式：找到leader，进行数据同步，然后参与到消息广播流程；

### 消息广播

Zab协议中的消息广播过程使用的是原子广播协议，类似一个二阶段提交的过程；ZAB协议中的二阶提交过程和二阶提交过程本来的定义略有不同：ZAB协议的二阶提交移除了中断逻辑，所以Follower服务器要么正常返回leader服务器提出的事务，要么就丢弃leader服务器；丢弃中断逻辑那么就可以在过半follower服务器反馈ACK消息之后leader服务器就发送提交事务命令，而不需要等到接收到所有的follower服务器返回的消息之后再决定是否提交；在这种提交模式下，是无法在leader服务器崩溃后保证数据的一致性的，所以就需要崩溃恢复模式来解决这个问题。

广播消息过程

Leader会为每个事务请求生成一个对应的proposal（提议）来进行广播，在广播之前为这个事务的proposal分配一个全局递增的唯一id，称之位事务ID(ZXID)，事务必须按照proposal事务ID的顺序来进行处理，广播消息协议是通过具有FIFO特性的tcp协议进行通信的，这样就保证了消息广播过程中消息接收和发送的顺序信息

具体步骤

1. leader服务器会为每个follower服务器各自分配一个单独的队列，将要广播的事务proposal放到队列中，并根据FIFO的策略发送
2. 每个follower服务器接收到这个事务proposal之后，首先会将事务日志写入本地磁盘，并且在成功写入后向leader服务器发送ACK消息
3. 当leader服务器接收到过半follower服务器反馈的ACK之后，就广播一个Commit消息给所有的follower服务器来提交事务，通过leader服务器也会完成自身事务的提交
4. Follower服务器接收到commit消息后进行事务提交

### 崩溃恢复

ZAB协议需要确保那些已经在leader服务器上提交的事务最终被所有的服务器都提交：就是当leader发出了提交事务消息后就挂了，其他的服务器仍会提交事务

ZAB协议需要确保丢弃那些只在leader服务器上被提交的事务：当leader服务器自身提交了事务，还没有发送提交命令时就挂了，那么其他的服务器就会丢弃这个事务

ZAB协议必须的leader选举算法：能够确保提交已经被leader提交的事务proposal（提交）,丢弃已经被跳过的事务，

数据同步过程

完成leader选举之后，在正式开始工作之前，leader服务器首先会确认事务日志中的所有proposal是否被过半的follower服务器提交，即是否完成数据同步过程

为每个follower服务器准备一个队列，并将那些没有被个follower服务器同步的事务以proposal消息的形式逐个发送到follower服务器，紧接着发送commit指令，以表示该事务已经被提交，等待follower服务器将所有未同步的事务都同步后，follower服务器就会被加入到真正的可用follower服务器列表中，并开始之后的流程；

丢弃事务（通过事务ID）

事务ID（ZXID）是一个64位的数字，其中低32位可用看作是单调递增的计数器，每生产一个新的事务proposal时，这个计数器就会+1；高32位表示leader周期epoch编号，每次新产生leader服务器后，就会从本地日志中取出最大事务ZXID,从ZAID中解析出epoch的值，然后对其+1，之后这个编码就作为新的epoch，并将ZXID的低32为从0开始计数生成新的ZXID，ZAB协议通过epoch编号来区分leader的周期变化，这样就能够避免leader选举之后出现相同事务id提交不同的事务，

（Zookeeper事务长期不提交会怎样）

### 深入zab协议

系统模型:一组进程组成的分布式系统中，每个进程都有自己独立的存储设备，各进程之间相互通信来实现消息的传递。每个进程都可以出现一次或者多次崩溃退出进程组；当进程恢复之后就再次加入进程组。

Zab协议规定任何时候都需要保证只有一个主进程负责进行消息广播，如果主进程崩溃，就需要选举出一个新的主进程，主进程的选举机制和消息广播机制是紧密关联的；随着时间的推移，会出现无限多个主进程并构成一个主进程序列，每个主进程都有个序列号，也被称为主进程周期

主进程周期

整个ZAB协议主要包括消息广播和崩溃恢复两个过程，进一步可以细分为三个阶段，分别是发现（Discovery)、同步（Synchronization)和广播（Broadcast)阶段。组成ZAB协议的每一个分布式进程，会循环地执行这三个阶段，我们将这样一个循环称为一个主进程周期。

Zab协议各阶段的算法流程

阶段一：发现

这个阶段主要是leader选举的过程

阶段二：同步

阶段三：广播

Leader选举

在zab协议中每个进程都有可能处于以下三种状态

LOOKING：leader选举阶段

FOLLOWING：follower服务器和leader保持同步状态

LEADING：leader服务器作为主进程领导状态

集群中所有zookeeper启动的时候，其初始化状态就是LOOKING状态，此时进程组中不存在leader。所有处于这种状态的进程，都会视图去选举出一个新的leader；当leader选出来之后，follower节点就是FOLLOWING状态，leader节点就是LEADER状态

当leader崩溃之后，follower节点就变为LOOKING状态，开始新一轮的leader选举

选举的时机

节点启动时和leader节点异常

暂停zab协议

## Zookeeper的操作

### 部署集群

1. 下载zookeeper安装包后，解压
2. 解压后，在conf文件夹下将zoo\_simple.cfg文件复制一份重命名为zoo.cfg，对zoo.cfg进行简单配置：

tickTime=2000

dataDir=D:/zookeeper-3.4.6/data

clientPort=2181

initLimit=5

syncLimit=2

server.1=IP1.2888.3888

server.2=IP2.2888.3888

server.3=IP3.2888.3888

配置简单说明：

在集群模式下集群中的每台机器都要感知整个集群是由那几台机器组成的，在配置文件中安装server.id=host:port:port这样的格式进行配置，每一行就代表一个服务器;这个配置中id被称为server id,用来标记该机器在集群环境中的机器序号，在通过dataDir配置的目录下创建一个myid文件，文件中只有一行内容，就是这个id

在集群中所有zookeeper服务的zoo.cfg文件内容都是一致的，最好使用svn或者Git把文件管理起来，确保每个机器都共享到一份相同的配置

myid文件内容只有一个数字，即为server id；比如server.1的myid文件内容是1，id的范围是1~255

1. 创建myid文件
2. 安装以上的步骤在其他机器上进行配置
3. 配置完成后启动服务

伪集群：集群中所有的zookeeper服务都部署在一台机器上，但是还是以集群的特性对外提供服务，配置基本上和集群模式一样，host是相同的，只是配置不一样的端口

### 启动zookeeper

1. java命令行

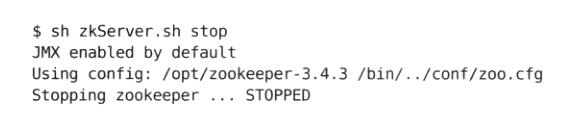
zookeeper的主入口QuorumPeerMain类就会启动zookeeper服务器，就是通过java命令来启动这个类，一般不使用这种方式

1. 使用自带脚本启动

在zookeeper的bin目录下提供了启动服务的脚本；在安装目录下的bin文件夹中有启动脚本：zKServer.cmd和zKserver.sh,分别是Windows环境和linux环境的启动脚本

### 停止服务

使用bin目录下的zkServer脚本的stop命令来完成：sh zkServer.sh stop



### 常用命令

1. 创建

create [-s] [-e] path data acl

-s表示节点是否有序

-e表示是否为临时节点

默认情况下，是持久化节点

acl是用来进行权限控制的，缺省时不做权限控制

1. 读取

ls path[watch]:返回指定节点下的所有子节点，只能返回当前节点的第一级子节点；

path表示指定数据节点的节点路径

get path [watch]：获得指定路径节点的节点信息，返回属性信息和数据内容

1. 更新

set path data [version]

data就是要更新的数据内容，

修改节点 path对应的data

节点的数据是由版本的概念，就是dataVersion：修改一次dataVersion就会改变，本次修改是基于节点的那个版本进行的

修改一次dataVersion就会+1

乐观锁的概念

数据库里面有一个 version字段去控制数据行的版本号

1. 删除

delete path [version]

删除节点

### Java客户端：curator

通过Zookeeper实例来连接点zookeeper服务器

#### 构造方法

4个构造方法

new ZooKeeper(connectString, sessionTimeout, watcher)

new ZooKeeper(connectString, sessionTimeout, watcher, canBeReadOnly)

new ZooKeeper(connectString, sessionTimeout, watcher, sessionId, sessionPasswd)

new ZooKeeper(connectString, sessionTimeout, watcher, sessionId, sessionPasswd, canBeReadOnly)

前面3个都会执行到最后一个

参数简介

ConnectString:指定zookeeper服务器列表，由英文逗号分开的host:port字符串组成：192.168.1.1:2181192.168.1.1:2181, 192.168.1.1:2181

sessionTimeout:会话超时时间，以毫秒为单位的整型值；在一个会话周期内，zookeeper客户端和服务器之间会通过心跳检测机制来维持会话的有效性，

watcher：

canBeReadOnly:boolean类型的参数，用于标记当前会话是read-only（只读）模式;

sessionId和sessionPasswd:会话id和会话秘钥，通过这两个参数可以唯一确定一个会话，同时客户端使用这个两个参数可以实现客户端会话复用

zookeeper客户端和服务端会话的建立是一个异步的过程，也就是在构造方法会话执行完成后，大多数情况下并没有建立会话，会话生命周期处于“CONNECTING”的状态

如果不阻塞主线程的话，运行完构造函数后，看不到process函数被回调，需要通过CountDownLatch来阻塞直到连接成功后消除阻塞，客户端会话创建完毕，这个创建的Zookeeper对象才能使用

**Watcher通知是一次性的，即一旦watcher通知触发后，该watcher失效，因此客户端需要反复注册watcher通知**

#### 创建节点

**public** String create(**final** String path, **byte** data[], List<ACL> acl, CreateMode createMode)

**public** **void**  create(**final** String path, **byte** data[], List<ACL> acl, CreateMode createMode, StringCallback cb, Object ctx)

参数

Path:创建的数据节点路径

Data:一个字节数组，节点创建后的初始内容

Acl:节点的Acl策略

createMode：节点的类型，是一个枚举类型，有4中可选的类型：持久（PERSISTENT）,持久顺序（PERSISTENT\_SEQUENTIAL），临时（EPHEMRAL），临时顺序（EPHEMRAL\_SEQUENTIAL）

cb：注册一个回调函数，cb对应的来需要实现StringCallBack接口，对processResult方法进行重写，当服务端创建节点完毕后，Zookeeper客户端会自动调用这个方法，这就可以处理相应的业务

ctx:用于传递一个对象，可以在回调方法中使用，通常是放一个上下文信息

Zookeeper无法在父节点不存在的情况下创建一个子节点，比如/node/node1，node节点不存在，但是创建的是node的子节点node1，这是要抛出异常的；如果创建的节点存在执行创建时也要抛出异常

目前Zookeeper节点存放的内容只支持字节数组（byte[]）类型；对应字符串，使用getByte方法获取字节数组即可，对应复杂对象，要使用其他方式将对象序列化（hessian或者kryo）

Acl：关于权限，如果对权限要求不高，只需要传入ZooDefs.Ids.OPEN\_ACL\_UNSAFE这个参数即可

**主节点是临时节点的话，不能构建子节点**

#### 删除节点

**public** **void** delete(**final** String path, **int** version)

**public** **void** delete(**final** String path, **int** version, VoidCallback cb, Object ctx)

path:删除节点的路径

version:指定节点的数据版本，表名本次删除是针对那个数据版本进行的；如果这个参数设置为-1，那么删除操作针对的是所有版本

cb：回调函数

ctx：传递给回调函数的参数

**删除只允许删除叶子节点**

#### 获取节点

1. 获取子节点列表

getChildren:获取一个节点的所有子节点

**public** List<String> getChildren(**final** String path, Watcher watcher)

**public** List<String> getChildren(String path, **boolean** watch)

**public** List<String> getChildren(**final** String path, Watcher watcher，Stat stat)

**public** List<String> getChildren(**final** String path, boolean watch,Stat stat)

**//没有返回的getChildren方法，参数中多了回调方法，其他的参数一样，作用不用，这个是在获取参数是处理某些业务的**

path:指定节点的路径

watcher：注册watcher,一旦这个节点的子节点列表发生了变化，那么就可以发送通知

watch：boolean类型，true表中注册当前Zookeeper实例默认的watcher，作用同上；false就是不需要对这个节点列表的改变做监听

stat：指定数据节点的节点状态信息

调用getChildren方法注册watcher后，当有子节点被添加，删除时，会向客户端发送一个NodeChildrenChanged通知，但是不包含子节点修改的

Stat记录节点的基本属性，比如创建时的事务ID（cZxid）,最后一次修改事务ID(mZxid)和节点数据内容的长度等

如果传入方法的是一个已经记录过程的实例，这个实例会被服务端响应的新实例替换，也就是指向新的引用

1. 获取节点数据

getData:获取节点内容

**public** **byte**[] getData(String path, **boolean** watch, Stat stat)

**public** **byte**[] getData(**final** String path, Watcher watcher, Stat stat)

path:节点的路径

watch：boolean类型，表示是否注册监听，true：要注册监听，使用Zookeeper默认的监听器

watcher:注册一个监听

注册监听器，监听的是NodeDataChanged事件，当监听的节点状态发生改变时会发送通知；节点状态的改变包括了数据内容的改变或者数据版本发送改变，这2个的改变都会触发NodeDataChanged事件

#### 修改节点

**public** Stat setData(**final** String path, **byte** data[], **int** version)

path：操作的节点

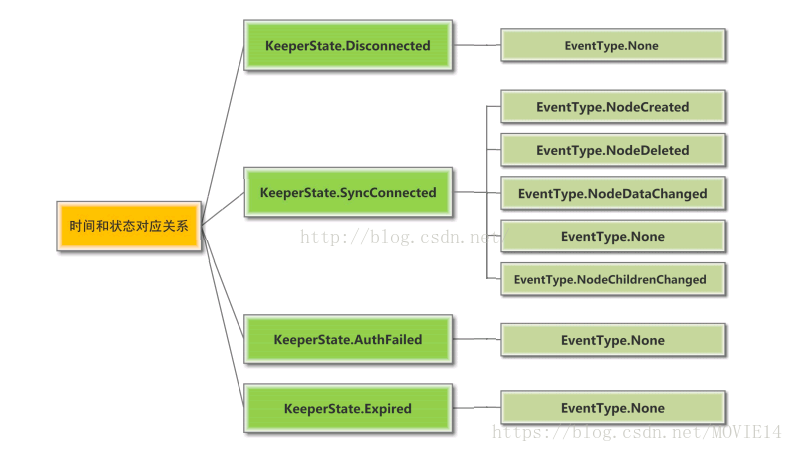
data[]：覆盖节点原数据内容的字符数组

version：节点的数据版本

version详解：指的是节点的数据版本，表明操作指定的数据版本；通过CAS原来来控制version值的变化，如果一个客户端试图进行更新操作时，它携带了最新的数据版本，而在同时，当前节点被其他的客户端更新，那么这个数据版本就一定会改变，那么就和当前客户端修改的版本不匹配，于是就无法更新，因此就避免了分布式并发问题，Zookeeper可以利用这个特性来构建较为复杂的应用场景：比如分布式锁

Zookeeper节点的数据版本是从0开始计数的，version为-1就不是合法的数据，它只是一个标记符，告诉客户端基于最新版本进行操作，如果对数据节点的变更操作没有原子性要求，那么就可以使用-1

基于version参数可以很好的控制Zookeeper节点数据的原子性



KeeperState.SyncConnected中的几种事件类型

NodeCreate:节点创建

NodeDeleted：删除节点

NodeDataChanged:访问节点

None：客户端和服务端建立连接

NodeChildrenChanged：子节点列表发送变更

exists,getData,getChildren这3个方法都能注册watcher监听,监听的节点必须存在；如果要对一个节点的状态进行监听，必须对这个节点路径添加watcher

当节点不存在是使用exists方法来注册

#### 权限控制

为了Zookeeper服务器上的数据被其他进程干扰或者修改，需要对Zookeeper上的数据访问进行权限控制，Zookeeper提供了ACL的权限控制机制来控制客户端对该节点数据访问的权限控制：通过设置Zookeeper服务器上数据节点的ACL，来控制客户端对该数据节点的方问；如果一个客户端符合ACL控制，那么就可以对其进行访问，否则无法操作

Zookeeper提供了多种权限控制模式

World，auth，digest,ip和super，

Zookeeper通过addAuthInfo(string scheme,byte[]auth)来进行权限信息的设置

Scheme:表示权限控制模式

auth:具体权限信息

ZooKeeper zk1 = **new** ZooKeeper(*host*, 3000, watcher);

Zk1.addAuthInfo(scheme, auth);

ZooKeeper zk2 = **new** ZooKeeper(*host*, 3000, watcher);

Zk2不能访问zk1创建的节点，zk2没有权限

### 开源客户端：curator

Apache Curator是一个比较完善的ZooKeeper客户端框架，通过封装的一套高级API 简化了ZooKeeper的操作。通过查看官方文档，可以发现Curator主要解决了三类问题：

1. 封装ZooKeeper client与ZooKeeper server之间的连接处理
2. 提供了一套Fluent风格的操作API
3. 提供ZooKeeper各种应用场景(recipe， 比如：分布式锁服务、集群领导选举、共享计数器、缓存机制、分布式队列等)的抽象封装

Curator主要从以下几个方面降低了zk使用的复杂性

1. 重试机制:提供可插拔的重试机制, 它将给捕获所有可恢复的异常配置一个重试策略，并且内部也提供了几种标准的重试策略(比如指数补偿)
2. 连接状态监控: Curator初始化之后会一直对zk连接进行监听，一旦发现连接状态发生变化将会作出相应的处理
3. zk客户端实例管理:Curator会对zk客户端到server集群的连接进行管理，并在需要的时候重建zk实例，保证与zk集群连接的可靠性
4. 各种使用场景支持:Curator实现了zk支持的大部分使用场景（甚至包括zk自身不支持的场景），这些实现都遵循了zk的最佳实践，并考虑了各种极端情况

基本应用

事件监听

Master选举

分布式锁

分布式计数器

分布式Barrier

# Zookeeper的典型应用场景

## 数据发布订阅

数据发布订阅模式就是所谓的配置中心，发布者将数据发送到zookeeper的一个或一系列节点上，供订阅者进行数据订阅，进而达到动态获取数据的目的，实现配置信息的集中式管理和数据的动态更新

发布订阅一般有2种设计方法，分别是推和拉；在推模式中，服务端主动将数据推送给所有订阅的客户端；拉模式就是客户端主动发送请求来获取数据，通常客户端都是采用定时进行轮询的方式来取数据

在zookeeper中采用的是推拉模式结合的方式：客户端向服务端注册自己需要关注的节点，一旦节点数据发生改变，那么服务端就会向客户端发送watcher事件通知，客户端接收到消息通知后，需要主动到服务端获取最新的数据

应用场景：系统中需要使用一些通用的配置信息，比如机器列表信息，运行时的开关配置，数据库配置等，这些全局的配置信息具备以下3个特性

1. 数据量通常比较小
2. 数据内容在运行时会发生改变
3. 集群中各个机器共享，配置一致

这些信息存储在本地配置文件或者内存变量中，在集群规模不大，配置变更不频繁时能方便的解决配置管理问题，但是一旦机器规模变大，且配置信息变更越来越频繁时这2种方式解决配置管理就变得很困难；

Zookeeper能非常高效的实现配置管理

## 负载均衡

## 命名服务

在分布式系统中，被命名的实体通常可以是集群中的机器，提供的服务地址或远程对象—这些都可以统称为名字，其中较为常见的就是分布式服务框架中的服务地址列表，通过使用命名服务，客户端可以根据指定的名字来获取资源的实体，服务地址和提供者的信息；比如可以存放dubbo服务地址，将信息存放在指定的节点（也就是是作为dubbo的配置中心）；

Zookeeper实现分布式全局唯一ID的分配机制，Id看作是唯一标识一个对象的标识符；比如在数据库中可以使用id自增来实现唯一id，但是当数据库实现分库之后，单个数据的id自增就不一定是全局唯一的；

可以使用uuid，但是uuid的缺点：长度过长（浪费空间），含义不明；

可以借助Zookeeper来实现全局id的唯一性

1. 通过create方法来创建一个顺序节点
2. 创建节点后返回节点名
3. 客户端拿到节点后进行简单的处理就是全局唯一id

## 分布式协调/通知

协调控制整个系统的运行流程，比如分布式式事务，机器间相互协调；映入一个协调者，便于将分布式协调的职责从应用中分离出来，减少系统之间的耦合，提示系统的可扩展性

Zookeeper中特有的**watcher注册和异步通知机制**，能够很好的实现分布式环境下不同机器，不同系统之间的协调与通知，从而实现对数据变更的实时处理

Zookeeper实现分布式协调和通知：不同客户端实现对zookeeper上的同一个数据节点进行watcher监听注册，监听节点数据的变化，如果节点发生改变，那么所有订阅的客户端能够接收到相应的watcher通知，并作出相应的处理

降低系统之间的耦合，能够非常方便的实现异构系统之间的灵活通信

## 集群管理

集群管理：集群监控和集群控制两大块，前者侧重对集群运行时状态的收集，后者则是对集群进行操作和控制

运维的需求：

1. 知道当前集群中有多少机器在工作
2. 收集每台机器的运行状态
3. 集群机器的上下线操作

Zookeeper具有两大特性

1. 客户端如果对zookeeper的一个数据节点注册watcher监听，那么当该节点的数据内容或是其子节点列表发生变更时，zookeeper服务器就会向订阅的客户端发送变更通知
2. 对在zookeeper上创建的临时节点，一旦客户端与服务器之间的会话断开，那么该临时节点就会被清除

利用这zookeeper的这2大特性，可以监控集群环境中工作的机器，实现集群机器存活性监控

后续监控实例

1. 分布式日志收集系统

收集分布式系统中不同机器上的系统日志

在分布式系统中把需要收集日志的机器分为多个分组，每个分组对应一个收集器，这个收集器就是一个后台机器，用于收集日志；对于大规模的分布式日志收集系统场景，通常需要解决2个问题：

1. 变化的日志源机器：在分布式系统中，伴随机器的变动，每个应用的机器几乎每天在发送改变，也就是说每个组别的日志源机器通常在发送变化
2. 变化的收集器机器：日志系统也会扩大，那么收集日志的机器也在变动

使用zookeeper来处理这2个问题

1. 注册收集器机器

使用zookeeper来进行日志系统收集器的注册，典型的做法就是创建一个收集器根节点，每个收集器启动的时候，会在收集器根节点下创建一个自己的节点（创建持久节点，因为节点上要存放分配的收集日志源机器列表）

1. 任务分发

创建好节点后，系统根据子节点的个数，将日志源机器分为若干组，然后将分组后的机器列表分别写到这些收集器创建节点的子节点上，这个收集器机器能够从自己节点上获取收集日志机器的列表，然后开始收集日志

1. 状态汇报

需要判断收集器机器是否存户：在收集器机器创建完自己专属的节点后，还需要在对应的子节点上创建一个状态节点，定期向该节点写入收集器机器的信息（心跳检测机制来实现），日志系统更具改状态子节点最后更新的时间来判断收集器机器是否存户

1. 动态分配

如果收集器机器挂掉或者新增，就需要动态的分配收集任务；

分配方式

全局动态分配：将整个日志源机器重新分配，比较简单；但是风险比较大，影响也比较大

局部动态分配：在小范围内进行任务的动态分配，更具负载情况，将高负载的分配到底负载机器上去

1. 在线云主机管理

这个管理器主要出现在那些虚拟主机的应用场景中；这类集群管理器中，有很重要的一块功能就是集群机器监控；对于机器状态，集群的在线率的统计要求较高，同时需要快速的对集群中机器的变更作出响应

机器的上线/下线

机器监控

## Master选举

分布式核心特性就是将具有独立计算能力的系统单元部署到不同的服务器上，构成一个分布式系统；在实际应用中也需要在这些分布在不同机器上的独立系统单元选举出一个Master；在分布式系统中Master是用来协调集群中其他系统单元，具有对分布式状态变更的绝对权，比如在读写分离的场景中，客户端的写请求往往是由master来完成的；或者是由master来处理一些复杂的逻辑，并将处理结果同步到其他系统单元；

场景：在集群中的所有系统单元需要对前端业务提供数据，比如根据一个id从海量的数据中获取想要的信息，如果让集群中所有的机器都要进行运算，那么就会耗费非常多的资源，一种比较好的处理方式就是只让集群中的部分甚至只有一台机器计算，将计算的结果同步到其他的机器上，这就大大的减少了重复劳动，提示性能

通过zookeeper实现Master选举

利用zookeeper节点的强一致性，能够很好的保证在分布式高并发情况下节点创建的唯一性；也就是说多个客户端访问同一个zookeeper服务端，同时创建相同路径的节点，那么就只会有一个客户端能创建这个节点；利用这个特性就非常容易的在分布式环境下实现Master选举

成功创建这个节点的客户端就成为了Master，并且为其他没有创建成功的客户端在这个节点上注册watcher监听，用来监控当前节点对应的Master机器是否存户，一旦Master挂掉，那么其他客户端就会重新进行master选举

## 分布式锁

分布式锁是在分布式系统中控制系统之间同步访问共享资源的一种方式；不同系统之间或者同一个系统的不同主机之间共享了一个或一组资源，那么访问资源的时候需要通过一些互斥手段来防止彼此之间的干扰，以保证一致性。

1. 排他锁

又称为写锁或独占锁，是一种基本的锁类型。

Zookeeper实现排他锁：通过zookeeper上的数据节点来表示一个锁；

在需要获取排他锁时，客户端创建一个临时节点来作为访问资源的锁标识；zookeeper保证只能有一个客户端成功创建同一名称的节点，如果客户端创建成功那么就获取到了锁，同时没有获取到锁的客户端就给该节点注册watcher监听，以便实时监听到这个锁节点的变更情况

释放锁：创建的是临时节点，那么就有2中方式释放锁

1. 客户端宕机，会话结束就释放锁了
2. 客户端主动删除该节点；
3. 共享锁

又称为读锁；如果事务t1对数据对象o1加上了共享锁，当前事务只能对o1进行读取操作，其他事务只能对这个数据对象加共享锁—直到该数据对象上的所有共享锁都被释放

共享锁和排他锁的区别：加上排他锁后，数据只对加锁的事务可见，而加上共享锁后，数据对事物都是可见的

Zookeeper实现共享锁：也是通过zookeeper创建临时节点来实现的，临时节点都是同一root节点下，不过不同的是共享锁创建的是临时顺序节点；通过不同的标记来区分是读锁还是写锁（比如R表示读，W表示写）

不同的事务都可以对同一数据进行读取操作，而更像操作必须在当前没有任何读和写操作的情况下进行。

释放锁：和排他锁一份，删除节点或者宕机会话终结

判断读写

1. 在root节点下创建子节点之后，获取root节点的所有子节点，对该节点注册子节点变更的watcher监听
2. 确定自己的节点序号在所有子节点中的顺序
3. 对于读请求：如果没有比自己序号小的节点，或者所有比自己序号小的节点都是读请求，那么就获取了共享锁，同时开始执行读逻辑
4. 对于写请求，如果自己节点不是序号最小的子节点，那么就等待
5. 接收到watcher通知后，重复以上步骤

上面的操作过程优化

存在问题：当释放锁之后，其他的机器都会收到这个节点被移除的通知，然后去从新获取一份子节点列表，然后再去判断自身节点所处的位置，获取到所得节点处理自身的逻辑，没有获取的就一直等待，当客户端移除自己的共享锁后，子节点变更会去通知所有机器，然而这个通知除了会影响比这个节点序号大的最近节点之外，对其他的都没有作用，所以其他节点就会存在许多的无用的重发操作

优化：判断自己是否是最小序号节点，于是每个客户端只需要关注比自己小（）的那个节点的变更情况就可以了（获取关注的节点，通过exists方法注册监听）

## 分布式队列

基于zookeeper实现的分布式队列，实现FIFO队列

所有客户端在zookeeper服务端的root节点下创建顺序临时节点，创建完成在来确定顺序

1. 首先获取root节点的所有子节点
2. 确定自身节点在所有子节点中的顺序
3. 如果自己的节点不是顺序序号最小的节点，那么就等待，同时向比自己序号小的最近的节点注册watcher监听，
4. 接收到通知后，重复以上步骤,就实现了FIFO

# Zookeeper内部实现

## 数据模型

### 数据节点znode

zookeeper的视图结构和文件系统类似，但是没有引入文件系统中目录和文件等相关概念，而是使用特有的数据节点概念称为ZNode;Znode是zookeeper中数据的最小单元，每个znode上都可以保存数据，同时还可以挂载子节点因此构成一个层次化的命名空间，这种树型接称之为树

在zookeeper中每个数据节点都被称为znode，znode按层次结构进行组织，形成一颗树；

**事务**

**在zookeeper中事务是指能够改变zookeeper服务器状态的操作，也称之为事务操作或者更新操作，**一般包括节点创建与删除，数据节点内容更新和客户端会话创建与失效等操作。对于每个事务zookeeper都会为其分配一个全局唯一的事务id，用zxid表示，通常是一个64位的数字。每个zxid对应一次更新操作，从这些zxid中可以间接的识别出zookeeper处理这些更新操作的全局顺序

### 节点特性

节点类型：在zookeeper中每个数据节点都是有生命周期的，其生命周期的长短取决于数据节点的类型。

节点类型可以分为持久节点，临时节点，顺序节点三大类，具体创建节点过程，通过使用组合，可以生成四种组合类型节点

1. 持久节点
2. 持久顺序节点
3. 临时节点
4. 临时顺序节点

### 节点的状态信息

节点创建是除了存储的数据内容之外，还存储了节点本身的状态信息；

Stat属性

Czxid :数据节点被创建时的事务id

Mzxid:节点最后一次更新的事务id

Ctime:节点创建时间

Mtime：节点最后一次被修改的时间

Version ：当前节点数据内容的版本号

Cversion：子节点的版本号

Aversion：节点的ACL变更的版本号

Ephemeralowner：创建该临时节点的sessionID,如果节点是持久的，那么就为0

Datalength：数据内容的长度

Numchildren：子节点的数量

Pzxid：子节点最后一次被修改的事务id，只有子节点列表发生改变才会更新这个值（创建或者删除节点），子节点的数据内容改变不会影响这个值

### 版本---保证分布式数据原子操作

Zookeeper数据节点引入版本的概念，每个数据节点都具有三个类型的版本信息，对数据节点的任何更新操作都会引起版本号的改变

**版本类型**

version:当前数据节点内容的版本号

cversion:当前数据节点子节点的版本号

aversion：当前数据节点ACL变更版本号

当节点创建后version就是0，对节点内容修改后就变为1，只要节点进行了变更操作无论操作前后值是否改变，version的值都会改变

### Watcher 机制

Zookeeper提供了分布式的数据发布/订阅功能，订阅者监听某个主题，主题发生改变，就会通知所有订阅者，订阅者接收到通知后就作出相应的处理

Zookeeper引入watcher机制来实现这种分布式功能；

Zookeeper允许客户端向服务器注册一个watcher监听，当服务器的一些指定的事件触发这个watcher，那么就会向客户端发送一个事件通知来实现分布的通知功能

Watcher接口

接口watcher表示一个标准的事务处理器，定义事件相关的逻辑，包括KeeperState和EventType两个枚举，分布表示通知状态和事件类型，同时定义事件的回调方法

process（WatchedEvent event）

watcher事件

|  |  |  |  |
| --- | --- | --- | --- |
| KeeperState | EventType | 触发条件 | 说明 |
| SyncConnected | None | 客户端和服务端建立会话 | 客户端和服务端处于连接状态 |
| NodeCreate | Watcher监听的对应节点被创建 |
| NodeDelete | Watcher监听的对应节点被删除 |
| NodeDataChange | Watcher监听的对应节点数据内容被修改（使用相同的数据来更新也会触发通知） |
| NodeChildrenChange | Watcher监听的对应节点的子节点列表发生变更（子节点列表变更是指子节点数量的变化，即删除或者增加节点，子节点的内容变更不会触发通知） |
| Disconnected | None | 客户端和服务端连接点断开 |  |
| Expired | None | 会话超时 |  |
| AuthFailed | None |  |  |

回调方法process，当zookeeper向客户端发送一个通知时，客户端会对相应的process方法进行回调，从而实现事件的处理

方法定义

**abstract** **public** **void** process(WatchedEvent event);

参数WatchedEvent包含每个事件的三个基本属性：通知状态（KeeperState），事件类型（EventType），节点路径（path）

WatcherEvent和watchedEvent：2者都是服务端对事件的封装，不同的是watchedEvent是一个逻辑事件，用于表示服务端和客户端程序执行过程的逻辑对象，

Watcherevent实现了serializable接口，可以序列化通过网络传播；服务端生成watchedEvent对象后，调用getWrapper方法，将自己包装成watcherevent，以便通过网络传输，客户端接收到服务端的这个对象后，首先从watcherevent中解析出watchedEvent,然后传递给process方法执行，

#### Watcher机制的源码实现

### ACL-保障数据的安全

权限模式：确定权限验证过程中使用的验证策略

Ip:通过ip地址来进行权限控制

Digest：使用类似username：password这种结构的权限标识来进行权限控制，

World：最开放的权限控制模式，实际上没有任何作用，数据节点的权限是对所有用户开发的，用户在不继续权限校验的情况下就能够操作节点

Super：超级用户模式，是一种特殊的digest，超级可以对任意Zookeeper上的数据节点进行任何操作。

授权对象：权限赋予的用户或者一个实体

在不同模式下授权对象不同

|  |  |
| --- | --- |
| 授权模式 | 授权对象 |
| IP | 通常是一个ip或者ip段 |
| Digest | 自定义：username:password |
| World | 只有一个id”anyone” |
| super | 也digest一样 |

权限：通过权限验证后允许的操作

权限操作分为5大类

Create

Delete

Read

Write

Admin

## 序列化与协议

Zookeeper的客户端和服务端之间会进行网络通信以实现数据的传输，对于一个网络通信，首先需要解决的就是对数据的序列化和反序列化，在zookeeper中使用**jute**这一序列化组件来进行数据的序列化和反序列化操作

使用jute组件来进行网络数据传送和本地磁盘数据存储的序列化和反序列化

#### Jute详解

Record接口:是jute定义的自己独特的序列化格式record,zookeeper中所需要进行网络传输或者本地磁盘存储的类型定义，都实现了这个接口

Record定义了2个最基本的方法，分别是serialize和deserialize,分别用于序列化和反序列化；参数archive是底层真正的序列化器和反序列化器；archive可以包含多个序列化和反序列化，因此2个方法都标记了参数tag,用于向序列化和反序列化标识对象自身的标记

**public** **interface** Record {

**public** **void** serialize(OutputArchive archive, String tag)

**throws** IOException;

**public** **void** deserialize(InputArchive archive, String tag)

**throws** IOException;

}

OutputArchive和InputArchive

这2个接口分布式jute定义的序列化和反序列化接口，目前有3种实现

BinaryOutputArchive/BinaryInputArchive:对数据对象进行序列化和反序列化，主要用于网络传输和本地磁盘的存储

CsvOutputArchive/CsvInputStream：对数据的序列化，方便对数据的可视化展示，因此被使用在toString方法中，

XmlOutputArchive/XmlInputStream：将数据已xml的形式保留和还原，目前没有使用

通信协议

Zookeeper的通信协议非常简单，对于请求，主要包含**请求头和请求体**，对于响应主要包含响应头和响应体

请求协议

请求头：RequestHeader

参数有xid和type

Xid用于记录客户端请求发起的先后序号，用来确保单个客户端请求的响应顺序

Type代表请求的操作类型，常见的包括创建节点，删除节点，获取节点

请求体：包含了请求的所有操作内容，不同的请求类型，器请求体部分是不相同的

ConnectRequest：创建会话,客户端和服务端建立会话的时候发送的请求

GetDataRequest:获取节点数据，客户端向服务端发送获取节点数据的请求

SetDataRequest:更新节点数据，客户端向服务端发送更新节点数据的请求

响应协议

响应头

ReplyHeader:基本信息xid,zxid,err

Xid和请求头中的一致，响应中只是将请求中的xid原值返回

Zxid是Zookeeper服务器上的最新的事务id,

Err是错误码，请求过程中出现异常情况，会在这个错误码中表示出来，

响应体，响应的主题内容部分，包含了响应的所有返回数据

Response

不同的响类型，响应体部分数据结构不同

ConnectResponse

GetDataResponse

SetDataResponse

## 客户端

Zookeeper的客户端的核心组件

Zookeeper实例:客户端的入口

ClientWatchManager：客户端watcher的管理器

HostPriovider：客户端地址列表管理器

ClientCnxn：客户端的核心线程，其中包括了两个线程，SendThread和EventThread；sendThread是I/o线程，负责客户端和服务端之间的io通信，EventThread是一个事件线程，主要负责对服务端事件进行处理

Zookeeper客户端的初始化和启动环节就是zookeeper对象的实例化过程，这个过程大致分为三个步骤：

1. 设置默认的watcher
2. 设置zookeeper的服务器地址列表
3. 创建ClientCnxn

### 会话的创建过程

#### 初始化阶段

1. 初始化zookeeper对象

创建zookeeper对象，在这个过程中会创建一个watcher管理器：ClientWatcherManager

1. 设置默认的watcher

如果在zookeeper的构造方法中传入了watcher对象，那么客户端会将这个watcher对象保存在ClientWatcherManager中

1. 构造zookeeper的服务器地址列表管理器，HostProvider

客户端会将构造方法中传入的服务器地址存放在服务器地址列表管理器HostProvider中

1. 创建并初始化ClientCnxn

创建网络连接器ClientCnxn，用来管理客户端和服务端的网络交互；同时还会创建两个队列outgoingQueue和pendingQueue，分别作为客户端的请求发送队列和服务端响应的等待队列，ClientCnxn连接器的底层i/o处理器是ClientCnxnSocket，同时客户端还会创建ClientCnxnSocket处理器

1. 初始化SendThread和EventThread

sendThread是I/o线程，负责客户端和服务端之间的io通信，EventThread是一个事件线程，主要负责对服务端事件进行处理；同时客户端还会将ClientCnxnSocket分配给sendThread作为底层网络io处理器，并初始化EventThread的待处理事件队列waitingEvents，用来存放所有等待被客户端执行的事件

#### 会话创建阶段

1. 启动SendThread和EventThread

sendThread首先会判断当前客户端的状态，做一些清理工作，**为客户端发送会话创建请求做准备**

1. 获取服务器地址

在开始创建tcp连接之前，sendthread首先需要获取一个zookeeper服务器的目标地址，通常是从hostprovider中随机获取一个，然后委托给ClientCnxnSocket去创建zookeeper服务器之间的tcp连接

1. 创建tcp连接

获取到服务器地址后，ClientCnxnSocket负责和服务器创建一个TCP长连接

1. 构造ConnectRequest请求

SendThread会负责根据当前客户端的实际设置，构造出一个ConnecteRequest请求，该请求代表客户端视图和服务端创建一个会话；zookeeper会将该请求进一步包装成网络i/o层的Packet对象，放到请求发送队列outgoingQueue中

1. 发送请求

Zookeeper客户端的请求装备完成后，ClientCnxnSocket负责从outgoingQueued中取出一个待发送的Packet对象，将其序列化成bytebuffer后向服务端发送；

#### 响应处理阶段

1. 接收服务器响应

CilentCnxnSocket接收到服务端的响应后，会首先判断当前的客户端状态是否已经初始化了，如果还没有完成初始化，那么就认定该响应就是会话创建请求的响应，直接readConnecteResult方法来处理响应

1. 处理Response

ClientCnxnSocket会对接收到的服务端响应进行反序列化，得到ConnectResponse对象，并从中获取到zookeeper服务端分配的会话sessionId

1. 连接成功

连接成功后，一方面要通知SendThread线程，进一步对客户端进行会话参数的设置，包括readTimeout和connectTimeout等，并更新客户端状态，另外通知地址管理器hostprovider当前成功连接的服务器地址

1. 生成事件：SyncConnected-None

为了让上层感知会话的成功创建，sendThread会生成一个事件，SyncConnected-None，代表客户端与服务端会话创建成功，并将该事件传递给EventThread

1. 查询watcher

EventThread线程收到事件后，会从ClientWatcherManager管理器中查询出对应的Watcher，针对SyncConnected-None事件，那么就直接找出初始化阶段步骤2中存储的默认Watcher，然后将其放到EventThread的waitingEvents队列中去。

1. 处理事件

EventThread不断的从waitingEvents队列中取出待处理的Watcher对象，然后直接调用该对象的process接口方法，以达到触发Watcher的目的。

### 服务器的地址管理

Zookeeper是如何从服务器列表地址中选择服务器，按顺序，或者随机

Zookeeper内部在选择接收到服务器地址列表后，会首先放入ConnectStringParser对象中封装起来，ConnectStringParser是一个服务器地址列表解析器，主要的作用：解析Chrootpath；保存服务器地址列表

#### Chroot：客户端隔离命名空间

Chroot:允许客户端为自己设置一个命名空间，如果设置了Chroot,那么该客户端对服务器的操作，都会被限制在自己的命名空间下；

通过设置Chroot，可以将zookeeper客户端和服务端的一颗子树相对应，在多应用公用一个zookeeper集群场景下，这对于实现不同应用之间的相互隔离非常有用

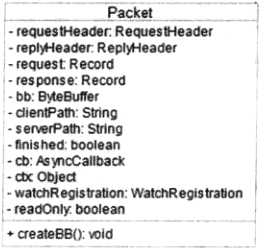
#### HostProvider:地址列表管理器

ConnectStringParser解析器会将服务器地址做些简单的处理，封装成InetSocketAddress对象，保存在ConnectStringParser的ArrayList类型的serverAddress属性中，然后会封装到staticHostProvider类中，staticHostProvider实现了接口HostProvider

### ClientCnxn：网络I/o

Packet

Packet是ClientCnxn内部定义的一个协议层封装，作为zookeeper请求和响应的载体，其数据结构如下



请求头requestHeader

响应头replayHeader

请求体request

响应体response

节点路径clientpath/serverpath

注册的watcher（WatchRegistration）

客户端和服务端进行网络传输的时候并没有将Packet的所有属性传输，Packet的createBB()方法负责对Packet对象进行序列化，最终生成用于网络传输的ByteBuffer对象，这个过程只会将requestHeader，request和readonly三个属性序列化，其余属性都保存在客户端，不进行网络传输

#### OutgoingQueue和pendingQueue

OutgoingQueue队列是一个请求发送队列，专门用于存储那些需要发送到服务端的packet集合，pending队列是为了存储那些从客户端发送到服务端的，但是需要等待服务端响应的Packet结合

#### 底层socket通信

ClientCnxnSocket定义了**底层Socket通信的接口**。在ZooKeeper3.4.0以前的版本中，客户端的这个底层通信层并没有被独立出来，而是混合在了ClientCnxn代码中。但后来为了使客户端代码结构更为清晰，同时也是为了便于对底层Socket层进行扩展（例如使用Netty来实现），因此3.4.0版本开始，抽取出了这个接口类。

在使用ZooKeeper客户端的时候，可以通过在zookeeper.clientCnxnSocket这个系统变量中配置ClientCnxnSocket实现类的全类名，以指定底层Socket通信层的自定义实现，

例如，-Dzookeeper.clientCnxnSocket=**org.apache.zookeeper.ClientCnxnSocketNIO**。在ZooKeeper中，其默认的实现是ClientCnxnSocketNIO。该实现类使用Java原生的NIO接口，其核心是doIO逻辑，主要负责对请求的发送和响应接收过程。

#### 请求发送

从outgoingQueue队列中取出一个可发送的Packat对象，生成客户端请求序号xid并将其设置到Packet请求头中，将其序列化后进行发送

请求发送完毕后，会立刻将该pendingqueue队列中，以便等待服务端响应返回后进行相应的处理

#### 响应接收

客户端接收到服务端的完整响应数据后，根据不同的客户端请求类型，做出对应的处理

1.如果检测到当前**客户端还尚未进行初始化**，那么说明当前客户端与服务端之间正在进行会话创建，那么就直接接收到的ByteBuffer（incomingBuffer）序列化成ConnectResponse对象。

2.如果当前客户端已经处于正常的会话周期，并且接收到的服务端响应是一个事件，那么ZooKeeper客户端会将接收到的ByteBuffer（incomingBuffer）序列化成WatcherEvent对象，并将该事件放入待处理队列中。

3.如果是一个**常规的请求响应**（指的是Create、GetData和Exist等操作请求），那么会从pendingQueue队列中取出一个Packet来进行相应的处理。ZooKeeper客户端首先会通过检验服务端响应中包含的XID值来确保请求处理的顺序性，然后再将接收到的ByteBuffer（incomingBuffer）序列化成相应的Response对象。

 最后，会在finishPacket方法中处理Watcher注册等逻辑。

#### SendThread

SendThread是客户端ClientCnxn内部一个核心的I/O调度线程，用于管理客户端和服务端之间的所有网络I/O操作。

一方面，SendThread**维护了客户端与服务端之间的会话生命周期**，其通过在一定的周期频率内向服务端发送一个PING包来实现心跳检测。同时，在会话周期内，如果客户端与服务端之间出现TCP连接断开的情况，那么就会自动且透明化的**完成重连操作**。

另一方面，SendThread管理了客户端所有的**请求发送和响应接收**操作，其将上层客户端API操作转换成相应的请求协议并发送到服务端，并完成对同步调用的返回和异步调用的回调。同时，SendThread还负责将来自服务端的事件传递给EventThread去处理。

#### EventThread

 EventThread是客户端ClientCnxn内部的另一个核心线程，**负责客户端的事件处理，并触发客户端注册的Watcher监听**。EventThread中有一个waitingEvents队列，用于临时存放那些需要被触发的Object,包括那些客户端注册的Watcher和异步接口中注册的回调器AsyncCallback。同时，EventThread会不断地从waitingEvents这个队列中取出Object，识别出其具体类型（Watcher或者AsyncCallback），并分别调用process和processResult接口方法来实现对事件的触发和回调。

## 会话

### 会话状态

Connecting，connected，reconnecting，reconnected，close

CONNECTING:客户端创建Zookeeper对象，还没和服务端建立连接时的状态

CONNECTED：客户端成功连接上服务器，客户端状态变为connected

当网络中断或者其他原因，客户端和服务端之间的连接断开，出现这种情况，客户端就会自动重连操作，同时客户端状态再次变更为Connecting，知道再次连上Zookeeper服务器后，状态变更为Connected，通常情况下，在Zookeeper运行期间，客户端状态总是介于connected和connecting两者之间；

RECONNECTING：

RECONNECTED：

CLOSE：出现会话超时，权限检查失败或者客户端主动退出，状态就变为close

### 会话创建

Session是zookeeper中会话的实体，代表一个客户端会话，包含以下4个基本属性：

sessionId:会话id,用来唯一标识一个会话，每个客户端创建一个新会话时，zookeeper都会为其分配一个全局唯一的sessionID

TimeOut:会话超时时间，

TickTime：下次会话超时时间点

isClosing：该属性用于标记一个会话是否已经被关闭

SessionId的创建

1. 获取当前时间的毫秒表示
2. 左移24位，为了防止负数的出现
3. 右移8位
4. 添加机器标识：SID
5. 将步骤3，4得到的两个64位表示的数值进行“|”操作

存在的问题：步骤1，ZooKeeper选择了使用Java语言自带的当前时间的毫秒数来作为该基数，针对当前时间的毫秒表示，通常情况下没有什么问题，但如果假设到了2022年04月08日时，在这种情况下，即使左移24位，还是有问题，因为24位后还是负数

解决：使用无符号右移，而非有符号右移，这样可以避免高位数值对SID的干扰了。该缺陷在3.4.6版本的ZooKeeper中已经得到了修复。

SessionTracker

SessionTracker是ZooKeeper服务端的会话管理器，负责会话的创建、管理和清理等工作。可以说，整个会话的生命周期都离不开SessionTracker的管理。每一个会话在SessionTracker内部都保留了三份，具体如下。

sessionById：这是一个HashMap<Long, SessionImpl>类型的数据结构，用于根据sessionID来管理Session实体。

sessionWithTimeout：这是一个ConcurrentHashMap<Long, Integer>类型的数据结构，用于根据sessionID来管理会话的超时时间。该数据结构和ZooKeeper内存数据库相连通，会被定期持久化到快照文件中去。

sessionSets：这是一个HashMap<Long, SessionSet>类型的数据结构，用于根据下次会话超时时间点来归档会话，便于进行会话管理和超时检查。

创建连接

**服务端**对于客户端的“会话创建”请求的处理，大体可以分为**四大步骤**，分别是**处理ConnectRequest请求**、**会话创建**、**处理器链路处理**和**会话响应**。在ZooKeeper服务端，首先将会由NIOServerCnxn来负责接收到客户端的“会话创建”请求，并反序列化出ConnectRequest请求，然后根据ZooKeeper服务端的配置完成会话超时时间的协商。随后，SessionTracker将会为该会话分配一个sessionID，并将其注册到sessionsById和sessionsWithTimeout中去，同时进行会话的激活。之后，该“会话请求”还会再ZooKeeper服务端的各个请求处理器之间进行顺序流转，最终完成会话的创建。

### 会话管理

分桶策略

Zookeeper的会话管理器主要是sessiontracker负责的，管理的方式成为分桶策略，所谓分桶策略，是指将类似的会话放在同一区块中进行管理, 以便于ZooKeeper对会话进行不同区块的隔离处理以及同一区块的统一处理

ZooKeeper将所有的会话都分配在了不同的区块之中，分配的原则是每个会话的“下次超时时间点”（ExpirationTime）, **ExpirationTime是指该会话最近一次可能超时的时间点**，对于一个新创建的会话而言，其会话创建完毕后，ZooKeeper就会为其计算ExpirationTime，计算方式如下：

ExpirationTime = CurrentTime + SessionTimeout

 其中CurrentTime指当前时间，单位是毫秒；SessionTimeout指该会话设置的超时时间，单位也是毫秒。

ZooKeeper的Leader服务器在运行期间会定时的进行会话超时检查，其时间间隔是ExpirationInterval，单位是毫秒，默认值是tickTime的值，即默认情况下，每隔2000毫秒进行一次会话超时检查。为了方便对多个会话同时进行超时检查，完整的ExpirationTime的计算方式如下：

ExpirationTime = CurrentTime + SessionTimeout

ExpirationTime = (ExpirationTime / ExpirationInterval+1)\* ExpirationInterval

ExpirationTime值总是ExpirationInterval的整数倍数

会话激活

 为了保持客户端会话的有效性，在ZooKeeper的运行过程中，客户端会在会话超时时间过期范围内向服务端发送PING请求来保持会话的有效性，我们俗称“心跳检测，同时，服务端需要不断地接收来自客户端的这个心跳检测，并且需要重新激活对应的客户端会话，我们将这个重新激活的过程称为TouchSession

流程：

1. 检验该会话是否已经被关闭
2. 计算该会话新的超时时间
3. 定位会话当前的区块
4. 迁移会话

会话超时检查

在ZooKeeper中，会话超时检查同样是由SessionTracker负责的。SessionTracker中有一个单独的线程专门进行会话超时检查，这里我们将其称为“超时检查线程 ”，其工作机制的核心思想其实非常简单：**逐个依次地对会话桶中剩下的会话进行清理。**

### 会话清理

当SessionTracker的会话超时检测线程整理出一些已经过期的会话后，就要对会话进行清理，清理的步骤分为以下7步：

标记会话状态为“已关闭”

由于整个会话清理过程需要一段的时间，因此为了保证在此期间不再处理来自该客户端的新请求，SessionTracker会首先将该会话的isClosing属性标记为true。这样，即使在会话清理期间接收到该客户端的新请求，也无法继续处理了。

发起“会话关闭”请求

为了使对该会话的关闭操作在整个服务端集群中都生效，ZooKeeper使用了提交“会话关闭”请求的方式，并立即交付给PrepRequestProcessor处理器进行处理。

收集需要清理的临时节点

在ZooKeeper中，一旦某个会话失效后，那么和该会话相关的临时（EPHEMERAL）节点都需要被一并清除掉。因此，在清理临时节点之前，首先需要将服务器上所有和该会话相关的临时节点都整理出来。

在ZooKeeper的内存数据库中，为每个会话都单独保存了一份由该会话维护的所有临时节点集合，因此在会话清理阶段，只需要根据当前即将关闭的会话的sessionID从内存数据库中获取到这份临时节点列表即可。

但是，在实际应用场景中，情况并没有那么简单，有如下的细节需要处理：在ZooKeeper处理会话关闭请求之前，正好有以下两类请求到达了服务端并正在处理中。

节点删除请求，删除的目标节点正好是上述临时节点中的一个。

临时节点创建请求，创建的目标节点正好是上述临时节点中的一个。

对于这两类请求，其共同点都是事务处理尚未完成，因此还没有应用到内存数据库中，所以上述获取到的临时节点列表在遇上这两类事务请求的时候，会存在不一致的情况。

假定我们当前获取的临时节点列表是ephemerals，那么针对第一类请求，我们需要将所有这些请求对应的数据节点路径从ephemerals中移除以避免重复删除。针对第二类请求，我们需要将所有这些请求对应的数据节点路径添加到ephemerals中去，以删除这些即将会被创建但是尚未保存到内存数据库中去的临时节点。

添加“节点删除”事务变更

完成该会话相关的临时节点收集后，ZooKeeper会逐个将这些临时节点转换成“节点删除”请求，并放入事务变更队列outstandingChanges中去。

删除临时节点

在上面的步骤中，我们已经收集了所有需要删除的临时节点，并创建了对应的“节点删除”请求，FinalRequestProcessor处理器会触发内存数据库，删除该会话对应的所有临时节点。

移除会话

完成节点删除后，需要将会话从SessionTracker中移除。主要就是从上面提到的三个数据结构（sessionById、sessionsWithTimeout和sessionSets）中将该会话移除掉。

关闭NIOServerCnxn

最后，从NIOServerCnxnFactory找到该会话对应的NIOServerCnxn，将其关闭。

### 重新连接

当客户端和服务器之间的网络连接断开时，ZooKeeper客户端会自动进行反复的重连，直到最终成功连接上ZooKeeper集群中的一台机器。在这种情况下，再次连接上服务端的客户端有可能会处于以下两种状态之一：

CONNECTED:如果在会话超时时间内重新连上了ZooKeeper集群中的一台机器，那么被认为重连成功。

EXPIRED:如果在会话超时时间以外重新连上了ZooKeeper集群中的一台机器，那么服务端其实已经对该会话进行了会话清理操作，因此在此连接上的会话会被是为非法会话

当客户端与服务端之间的连接断开后，用户在客户端可能会看到异常类型以及该怎么正确的处理这些异常：

CONNECTION\_LOSS(连接断开)

在这种情况下，客户端会自从地址列表种重新逐个选取新的地址并尝试进行重新连接，直到最终成功连接上服务器。

举个例子：假设某应用在使用ZooKeeper客户端进行setData操作的时候，正好出现了CONNECTION\_LOSS现象，那么客户端会立即收到事件None-Disconnected通知，同时会抛出异常org.apache.zookeeper.KeeperException$ConnectionLossException.

在这种情况下，我们的应用需要做的事情就是捕获住ConnectionLossException，然后等待客户端自动完成重连。

一旦客户端重连成功了，那么客户端就会收到事件None-SyncConnected通知，之后就可以重试刚刚出错的setData操作

SESSION\_EXPIRED （会话过期）

这种情况下用户需要重新实例化一个ZooKeeper对象，并且看应用的复杂情况，重新恢复临时数据。

SESSION\_MOVED(会话转移)

因为会话过期重连，导致会话从ZK服务器1转移到了ZK服务器2上。

ZK服务器从3.2.0版本之后提出了会话转移的概念。在处理客户端请求的时候，会首先检查会话的所有者（Owner），如果客户端请求的会话Owner不是当前服务器的话，那么就会直接抛出SessionMovedException异常。

## 服务器启动

### 单机启动

Zookeeper服务器的启动流程，大体可分为五个主要步骤，配置文件解析，初始化数据管理器，初始化网络i/o管理器，数据恢复和对外服务

启动流程

预启动

1. 统一由QuorumPeerMain作为启动类
2. 解析配置文件zoo.cfg
3. 创建并启动历史文件清理器DatadirCleanupManager
4. 判断当前是集群模式还是单机模式的启动
5. 再次进行配置文件zoo.zfg的解析
6. 创建服务器实例zookeeperServer

初始化

1. 创建服务器统计器ServerStatus
2. 创建zookeeper数据管理器fileTxnSnapLong。
3. 设置服务器tickTime和会话超时时间限制。
4. 创建ServerCnxnFactory
5. 初始化ServerCnxnFactory
6. 启动ServerCnxnFactory主线程
7. 恢复本地数据
8. 创建并启动会话管理器
9. 初始化zookeeper的请求处理链
10. 注册jmx服务
11. 注册zookeeper服务器实例

### 集群启动

预启动

1. 统一由QuorumPeerMain作为启动类
2. 解析配置文件zoo.cfg
3. 创建并启动历史文件清理器DatadirCleanupManager
4. 判断当前是集群模式还是单机模式的启动，集群模式下zoo.cfg中配置多个服务器地址

初始化

1. 创建ServerCnxnFactory。
2. 初始化ServerCnxnFactory。
3. 创建Zookeeper数据管理器FileTxnSnapLog。
4. 创建QuorumPeer实例。Quorum是集群模式下特有的对象，是Zookeeper服务器实例 (ZooKeeperServer)的托管者，QuorumPeer代表了集群中的一台机器，在运行期间， QuorumPeer会不断检测当前服务器实例的运行状态，同时根据情况发起Leader选举。
5. 创建内存数据库ZKDatabase。ZKDatabase负责管理ZooKeeper的所有会话记录以及 DataTree和事务日志的存储。
6. 初始化QuorumPeer。将核心组件如FileTxnSnapLog、ServerCnxnFactory、ZKDatabase 注册到QuorumPeer中，同时配置QuorumPeer的参数，如服务器列表地址、Leader选举算法和会话 超时时间限制等。
7. 恢复本地数据。
8. 启动ServerCnxnFactory主线程

## Leader选举

### 服务器启动时Leader选举流程

1. 每个server会发出一个投票：在初始情况，服务器都会将自己作为leader服务器来投票，每次投票包含的基本元素：所推荐的服务器的myid和zxid；也就是在初始化的时候服务器都会投自己一票，也就是将自己的myid和zxid组成投票信息，发送给集群的其他机器
2. 接收来自各个服务器的投票：每个服务器都会接收到投票后，首先会判断改投票的有效性，包括是否是本轮投票，是否来自looking状态的服务器
3. 处理投票：将接收到的投票和自己的投票进行pk,pk规则如下
4. 优先检查zxid，zxid比较大的服务器作为leader
5. 如果zxid相同，那么就比较myid
6. 统计投票

每次投票之后，服务器会统计所有投票，判断是否已经有过半的机器接收到相同的投票信息，当过半的机器收到相同的投票信息，就认为已经选出了leader

1. 改变服务器状态：一旦leader确定，每个服务器就会更新自己的状态，follower就是following，leader就是leading

### 服务器运行期间leader选举流程

在运行期间一旦leader挂了，那么整个集群将会暂时无法对外服务，进入新一轮的leader选举，选举流程基本上和启动一致

1. 变更状态
2. 每个server会发出一个投票
3. 接收来自各个服务器的投票
4. 处理投票
5. 统计投票
6. 改变服务器状态

### Leader选举算法细节

SID：服务器id，服务器的唯一标识，和myid一致

Zxid:事务id，服务器状态变更的唯一标识

Vote：投票

Quorum：过半机器数

#### 投票过程

第一轮投票

当服务器处于looking状态的时候，它就会向集群中所有其他机器发送消息，这个消息就是投票

投票包含2个基本信息：推荐服务器的sid和zxid，分别表示被推举服务器的唯一标识和事务id;

第一次投票的时候，由于无法检测到集群中其他机器的状态，所以每台机器都将自己作为被推荐的对象来进行投票

变更投票

服务器收到投票的处理，都是一个对比的过程，对比自身的投票和收到的投票

vote\_sid:接收到的投票中所推荐的leader服务器的sid

vote\_zxid：接收到的投票中所推荐的leader服务器的zxid,

self\_sid:当前服务器的sid

sef\_zxid:当前服务器的zxid

对比规则

1. vote\_ zxid大于self\_ zxid,就认可当前收到的投票，并将该次投票发送出去
2. vote\_ zxid小于self\_ zxid,就认坚持自己的投票，不做任何改变
3. 如果vote\_zxid和self\_zxid相等，那么就对比两者的sid，如果vote\_sid大于self\_sid,那么就认可当前接收到的投票，并将该次投票发送出去
4. Vote\_sid小于self\_sid,那么就坚持自己的投票，不做改变

对比完成后需要变更自身的投票

确定leader

经过二次投票后，集群中的每台机器都会再次收到其他机器的投票，然后开始统计投票；如果一台机器收到了超过半数的相同投票，那么这个投票对应的sid机器就为leader

#### 投票数据结构

Vote:

id:long,被推荐的leader的sid

zxid:long,被推荐的leader的事务id

electionEpoch:long，逻辑时钟。用来判断多个投票是否是在同一轮被选举周期中，改值在服务端时一个自增序列，每进入新的一轮投票后，都会对改值+1

peerEpoch:long，被推荐的leader的epoch

state:ServerState。当前服务器的状态

QuorumCnxnManager：网络io

每台服务器启动的时候都会创建一个quorumCnxnManager，负责各台服务器之间的底层leader选举过程的通信

#### 消息队列

QuorumCnxManager内部维护了一系列的队列，用来保存接收到的、待发送的消息以及消息的发送器，**除接收队列以外，其他队列都按照SID分组形成队列集合**

 recvQueue：消息接收队列，用于存放那些从其他服务器接收到的消息。 queueSendMap：消息发送队列，用于保存那些待发送的消息，按照SID进行分组。

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

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

#### 建立连接

为了能够相互投票，Zookeeper集群中的所有机器都需要两两建立起网络连接。QuorumCnxManager在启动时会创建一个ServerSocket来监听Leader选举的通信端口(默认为3888)。开启监听后，Zookeeper能够不断地接收到来自其他服务器的创建连接请求，在接收到其他服务器的TCP连接请求时，会进行处理。

为了避免两台机器之间重复地创建TCP连接**，Zookeeper只允许SID大的服务器主动和其他机器建立连接，否则断开连接。**

在receiveConnection函数中，服务器通过对比自己和远程服务器的sid值，来判断是否接收连接请求，如果当前服务器发现自己的sid较大，那么就断开当前连接，然后自己主动去和远程服务器建立连接；一旦连接建立，就会根据远程服务器的SID来创建相应的消息发送器SendWorker和消息接收器RecvWorker，并启动。

#### 消息接收与发送

RecvWorker:不断从tcp连接中读取信息，并将其保存到recvQueue队列中

SendWorker：从消息发送队列中获取一个消息来发送

#### FastLeaderElection：选举算法的核心部分

外部投票：其他服务器发来的投票

内部投票：服务器自身的投票

选举轮次：leader选举的轮次，即logicalclock

Pk:内部投票和外部投票进行对比来确定是否需要变更内部投票

选票管理组件

Sendqueue：选票发送队列，用来保存待发送的选票

Recvqueue：选票接收队列，用于保存接收到的选票

WorkerReciver:选票接收器

WorkerSender：选票发送器

算法的核心流程

Leader选举算法的基本流程就是lookforleader方法的逻辑，服务器状态为looking时，调用这个方法来进行leader选举

1. 自增选举轮次：  在FastLeaderElection实现中，有一个**logicalclock属性**，用于标识当前Leader的选举轮次，ZooKeeper规定了所有有效地投票都必须在同一轮次中。ZooKeeper在开始新一轮的投票时，会首先对logicalclock进行自增操作。
2. 初始化选票：在开始进行新一轮的投票之前，每个服务器都会首先初始化自己的选票。初始化选票也就是对Vote属性的初始化。在初始化阶段，每台服务器都会将自己推举为Leader
3. 发送初始化选票：   在完成选票的初始化后，服务器就会发起第一次投票。ZooKeeper会将刚刚初始化好的选票放入sendqueue队列中，由发送器WorkerSender负责发送出去。
4. 接收外部投票：每台服务器都会不断地从recvqueue队列中获取外部投票。如果服务器发现无法获取到任何的外部投票，那么就会立即确认自己是否和集群中其他服务器保持着有效连接。如果发现没有建立连接，那么就会马上建立连接。如果已经建立了连接，那么就再次发送自己当前的内部投票。
5. 判断选举轮次：当发送完初始化选票之后，接下来就要开始处理外部投票了。在处理外部投票的时候，会根据选举轮次来进行不同的处理。

**外部投票的选举轮次大于内部投票**。

如果服务器发现自己的选举轮次已经落后于该外部投票对应服务器的选举轮次，那么就会立即更新自己的选举轮次（logicalclock），并且清空所有已经收到的投票，然后使用初始化的投票来进行PK以确定是否变更内部投票（关于PK的逻辑会在步骤6中统一讲解），最终再将内部投票发送出去。

**外部投票的选举轮次小于内部投票。**

如果接收到的选票的选举轮次落后于服务器自身的，那么ZooKeeper就会直接忽略该外部投票，不做任何处理，并返回步骤4。

**外部投票的选举轮次和内部投票一致。**

这也是绝大多数投票的场景，如果外部投票的选举轮次和内部投票一致的话，那么就开始进行选票PK。

总的来说，只有在同一个选举轮次的投票才是有效的投票。

1. 选票pk：FastLeaderElection.totalOrderPredicate方法的核心逻辑。选票PK的目的是为了确定当前服务器是否需要变更投票，主要从选举轮次、ZXID和SID三个因素来考虑

**如果外部投票中被推举的Leader服务器的选举轮次大于内部投票**，那么就需要进行投票变更。

**如果选举轮次一致的话**，那么就对比两者的ZXID。如果外部投票的ZXID大于内部投票，那么就需要进行投票变更。

**如果两者的ZXID一致**，那么就对比两者的SID。如果外部投票的SID大于内部投票，那么就需要进行投票变更。

1. 变更投票： 通过选票PK后，如果确定了外部投票优于内部投票（所谓“优于”，是指外部投票所推举的服务器更适合成为Leader），那么就进行投票变更——使用外部投票的选票信息来覆盖内部投票。变更完成后，再次将这个变更后的内部投票发送出去。
2. 选票归档：无论是否进行了投票变更，都会将刚刚收到的那份外部投票放入“选票集合”recvset中进行归档。recvset用于记录当前服务器在本轮次的Leader选举中收到的所有外部投票——按照服务器对应的SID来区分，例如{(1, vote1) , (2, vote2), ...}。
3. 统计投票：完成了选票归档之后，就可以开始统计投票了。统计投票的过程就是为了统计集群中是否已经有过半的服务器认可了当前的内部投票。如果确定已经有过半的服务器认可了该内部投票，则终止投票，否则返回步骤4。
4. 更新服务器状态： 统计投票后，如果已经确定可以终止投票，那么就开始更新服务器状态。服务器会首先判断当前被过半服务器认可的投票状态更新为LEADING。如果自己不是被选举产生的Leader的话，那么就会根据具体情况来确定自己是FOLLOWING或是OBSERVING。

## 集群中的角色

**Leader:**事务请求的唯一调度者和处理者，保证集群事务处理的顺序；集群内部各服务器的调度者

**Follower：**处理客户端非事务请求，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：**观察Zookeeper集群的最新状态变化并将这些状态变更同步过来，工作原理和follower一致，对非事务的请求都能进行独立的处理，而对事物的处理都会转发给leader服务器

Observer不参与任何投票，和leader选举投票，在不影响集群事务处理能力前提下提升集群的非事务处理能力

**集群间消息的通信**

数据同步型

服务器初始化型

请求处理型

会话管理型

## 请求处理

### 会话创建请求

会话创建过程中zookeeper服务端的流程细节

### Setdata请求

服务端对setdata请求的处理

### 事务请求转发

Zookeeper的事务转发机制

### getData请求

服务端对getdata请求的处理

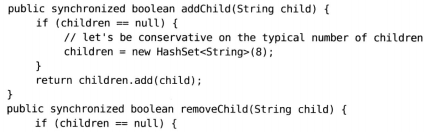
## 数据与内存

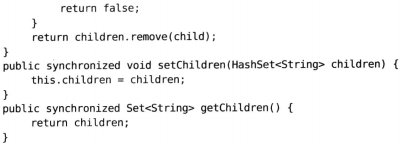
### 内存数据

**DataTree**：  DataTree是ZooKeeper内存数据存储的核心，是一个“树”的数据结构，代表了内存中的一份完整的数据。DataTree不包含任何与网络、客户端连接以及请求处理等相关的业务逻辑，是一个非常独立的ZooKeeper组件。

**DataNode**

   DataNode是数据存储的最小单元，其数据结构如上图所示。DataNode内部除了保存了节点的数据内容（data[]）、ACL列表（acl）和节点状态（stat）之外，正如最基本的数据结构中对树的描述，还记录了父节点（parent）的引用和子节点列表（children）两个属性。同时，DataNode还提供了对子节点列表操作的各个接口：





**nodes**

DataTree用于存储所有ZooKeeper节点的路径、数据内容及其ACL信息等，底层的数据结构其实是一个典型的ConcurrentHashMap键值对结构：



        在nodes这个Map中，存放了ZooKeeper服务器上所有的数据节点，可以说，对于ZooKeeper数据的所有操作，底层都是对这个Map结构的操作。nodes以数据节点的路径（path）为key，value则是节点的数据内容：DataNode。

        另外，对于所有的临时节点，为了便于实时访问和及时处理，DataTree中还单独将临时节点保存起来：



**ZKDatabase**

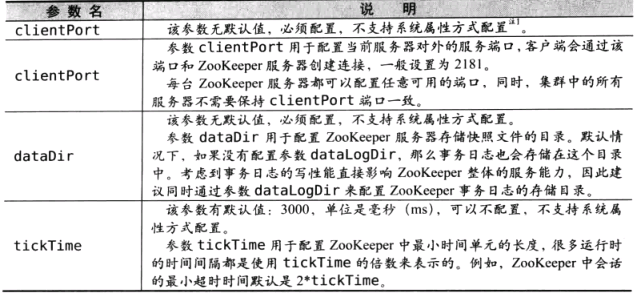
        ZKDatabase，正如其名字一样，是ZooKeeper的内存数据库，负责管理ZooKeeper的所有会话、DataTree存储和事务日志。ZKDatabase会定时向磁盘dump快照数据，同时在ZooKeeper服务器启动的时候，会通过磁盘上的事务日志和快照数据文件恢复成一个完整的内存数据库。

# Zookeeper其他

## Zookeeper配置详解

  Zookeeper的配置文件存放在主目录下的conf目录中，文件名为zoo.cfg。

基础配置



高级配置

**dataLogDir**：默认值为dataDir的值，不支持以系统属性方式配置，用于配置Zookeeper服务器存储事务日志的目录。默认情况下与快照数据保存在相同的目录中。在条件允许的情况下，可以将事务日志的存储配置在一个单独的磁盘上，因为事务日志记录对于磁盘的性能要求非常高，为了保证数据一致性，Zookeeper在返回客户端事务请求的响应之前，必须将本次请求对应的事务日志写入到磁盘中，因此，事务日志的写性能直接决定了Zookeeper在处理事务请求时的吞吐。针对同一块磁盘的其他并发读写操作，尤其是数据快照操作，会极大的影响事务日志的写性能。  
         **server.id=host:port:port**：无默认值，单机模式下不需要配置，不支持以系统属性方式配置。用于配置Zookeeper集群的机器列表。其中id为serverId，与每台服务器的myid文件中的数字对应，host为主机名。两个端口，分别为Follower与Leader进行通信和同步时使用的端口和用于Leader选举的端口。  
        **initLimit**：默认值为10，表示是tickTime的10倍，必须配置，并且是正整数，不支持以系统属性方式配置，用于配置Leader服务器等待Follower启动，并完成数据同步的时间。  
        **syncLimit**：默认值为5，表示是tickTime的5倍，必须配置，并且是正整数，不支持以系统属性方式配置，用于配置Leader服务器和Follower之间进行心跳检测的最大延时时间，在Zookeeper运行期间，Leader服务器会与所有的Follower进行心跳检测来确定该服务器是否存活。如果在syncLimit时间内无法获取到Follower的心跳检测响应，就认为Follower已经脱离了和自己的同步。在网络环境较差时，可以适当调大这个参数。  
        **snapCount**：默认为100000，表示每进行snapCount次事务日志输出后，触发一次快照，生成一个snapshot.\*文件。为了避免所有的服务器在同一时间进行快照操作而影响性能，会对snapCount进行随机数处理：snapCount/2+rand.int(snapCount/2)。此属性只支持系统属性方式配置：zookeeper.snapCount。  
         **preAllocSize**：默认值为65536，单位为Kb，即64Mb。仅支持系统属性方式配置：zookeeper.preAllocSize。用于配置Zookeeper事务日志文件预分配的磁盘空间大小。如果调整snapCount大小，preAllocSize也要随之做出变更。  
         **minSessionTimeout**/ **maxSessionTimeout**：默认值分别为2和20，即tickTime的2倍和20倍。表示会话的超时时间在2\*tickTime~20\*tickTime之间，单位为毫秒，不支持以系统属性方式配置。用于服务端对客户端的会话超时时间进行限制，如果客户端设置的超时时间不在该范围内，服务端会强制设置为最大或最小超时时间。  
         **maxClientCnxns**：默认为60，不支持以系统属性方式配置。从Socket层面限制单个客户端与单台服务器之间的并发连接数，即以ip地址来进行连接数的限制。如果设置为0，表示不做任何限制。仅仅是单台客户端与单个Zookeeper服务器连接数的限制，不能控制所有客户端的连接数总和。  
        **jute.maxbuffer**：默认值为1048575，单位为字节，仅支持系统属性方式配置：jute.maxBuffer。用于配置单个数据节点上可以存储的最大数据量大小。在变更该参数时，需要变更集群中所有机器以及客户端上均设置才能生效。  
         **clientPortAddress**：无默认值，不支持以系统属性方式配置。针对多网卡的机器，允许为每个IP地址指定不同的监听端口。  
         **autopurge.snapRetainCount**：默认值为3，不支持以系统属性方式配置。用于配置Zookeeper在自动清理的时候需要保留的快照数据文件数量和对应的事务日志文件。此参数的最小值为3，如果配置的值小于3会自动调整到3。  
         **autopurge.purgeInterval**：默认值为0，单位为小时，不支持以系统属性方式配置。用于配置Zookeeper进行历史文件自动清理的频率。如果配置为0或负数，表示不需要开启定时清理功能。  
         **fsync.warningthresholdms**：默认值为1000，单位为毫秒，仅支持系统属性方式配置：fsync.warningthresholdms。用于配置Zookeeper进行事务日志fsync操作时消耗时间的报警阈值。一旦fsync操作消耗的时间大于该参数指定的值，就在日志中打印出报警日志。  
         **forceSync**：默认值为yes，可配置项为“yes”和“no”，仅支持系统属性方式配置：zookeeper.forceSync。用于配置Zookeeper服务器是否在提交的时候，将日志写入操作强制刷入磁盘，默认情况下，每次事务日志写入操作都会实时刷入磁盘，如果设置为“no”，则能一定程度的提高ZooKeeper的写性能，但会存在类似机器断电等安全风险。  
         **globalOutstandingLimit**：默认值为1000，仅支持系统属性方式配置：zookeeper.globalOutstandingLimit。用于配置ZooKeeper服务器最大请求堆积数量。在Zookeeper服务器运行的过程中，客户端会源源不断的将请求发送到服务器，为了防止服务端资源耗尽，服务端必须限制同时处理的请求数，即最大请求堆积数量。  
         **leaderServes**：默认值为yes，可选配置项为“yes”和“no”，仅支持系统属性方式配置：zookeeper.leaderServes。用于配置Leader服务器是否能够接受客户端的连接，即是否允许Leader向客户端提供服务。默认情况下，Leader服务器能够接受并处理客户端的所有读写请求。在Zookeeper的架构设计中，Leader服务器主要用来进行对事务更新  
请求的协调以及集群本身的运行时协调，因此可以设置让Leader服务器不接受客户端的连接，以使其专注于进行分布式协调。  
         **skipAcl**：默认值为no，可选配置项为“yes”和“no”，仅支持系统属性方式配置：zookeeper.skipACL。用于配置ZooKeeper服务器是否跳过ACL权限检查，默认情况下，会对每一个客户端请求进行权限检查。如果设置为“yes”，则能一定程度的提高ZooKeeper的读写性能，但同时也将向所有客户端开放Zookeeper的数据，包括之前设置过ACL权限的数据节点，也不在接受权限控制。  
         **cnxTimeout**：默认值为5000，单位毫秒，仅支持系统属性方式配置：zookeeper.cnxTimeout。用于配置在Leader选举过程中各服务器之间进行TCP连接创建的超时时间。  
         **electionAlg**：用于配置选择ZooKeeper进行Leader选举时所使用的算法。目前Zookeeper中只保留了一种选举算法，所以不做讲解。

## 集群组成

### 过半存活即可用

一个 ZooKeeper 集群如果要对外提供可用的服务，那么集群中必须要有过半的机器正常工作并且彼此之间能够正常通信；

基于这个特性，如果想搭建一个能够允许 N 台机器 down 掉的集群，那么就要部署一个由 2\*N+1 台服务器构成的 ZooKeeper 集群。因此，一个由 3 台机器构成的 ZooKeeper 集群，能够在挂掉 1 台机器后依然正常工作，而对于一个由 5 台服务器构成的 ZooKeeper 集群，能够对 2 台机器挂掉的情况进行容灾。注意，如果是一个由6台服务器构成的 ZooKeeper 集群，同样只能够挂掉 2 台机器，因为如果挂掉 3 台，剩下的机器就无法实现过半了。

因此，从上面的讲解中，我们其实可以看出，对于一个由 6 台机器构成的 ZooKeeper 集群来说，和一个由 5 台机器构成的 ZooKeeper 集群，其在容灾能力上并没有任何显著的优势，反而多占用了一个服务器资源。基于这个原因，ZooKeeper 集群通常设计部署成奇数台服务器即可。

### 容灾

**所谓容灾**，在 IT 行业通常是指我们的计算机信息系统具有的一种在遭受诸如火灾、地震、断电和其他基础网络设备故障等毁灭性灾难的时候，依然能够对外提供可用服务的能力。

**单点问题**

单点问题是分布式环境中最常见也是最经典的问题之一，在很多分布式系统中都会存在这样的单点问题。

具体地说，**单点问题是指在一个分布式系统中，如果某一个组件出现故障就会引起整个系统的可用性大大下降甚至是处于瘫痪状态，那么我们就认为该组件存在单点问题**。

ZooKeeper 确实已经很好地解决了单点问题。我们已经了解到，**基于“过半”设计原则**，ZooKeeper 在运行期间，集群中至少有过半的机器保存了最新的数据。因此，只要集群中超过半数的机器还能够正常工作，整个集群就能够对外提供服务。

在进行 ZooKeeper 的容灾方案设计过程中，我们要充分考虑到“过半原则”。也就是说，无论发生什么情况，我们必须保证 ZooKeeper 集群中有超过半数的机器能够正常工作。因此，通常有以下两种部署方案。

**双机房部署**

在进行容灾方案的设计时，我们通常是以机房为单位来考虑问题。在现实中，很多公司的机房规模并不大，因此双机房部署是个比较常见的方案。但是遗憾的是，**在目前版本的 ZooKeeper 中，还没有办法能够在双机房条件下实现比较好的容灾效果**——**因为无论哪个机房发生异常情况，都有可能使得 ZooKeeper 集群中可用的机器无法超过半数**。当然，在拥有两个机房的场景下，通常有一个机房是主要机房（一般而言，公司会花费更多的钱去租用一个稳定性更好、设备更可靠的机房，这个机房就是主要机房，而另外一个机房则更加廉价一些）。我们唯一能做的，就是尽量在主要机房部署更多的机器。例如，对于一个由 7 台机器组成的 ZooKeeper 集群，通常在主要机房中部署 4 台机器，剩下的 3 台机器部署到另外一个机房中。

**三机房部署**

既然在双机房部署模式下并不能实现好的容灾效果，那么对于有条件的公司，选择三机房部署无疑是个更好的选择，**无论哪个机房发生了故障，剩下两个机房的机器数量都超过半数**。假如我们有三个机房可以部署服务，并且这三个机房间的网络状况良好，那么就可以在三个机房中都部署若干个机器来组成一个 ZooKeeper 集群。

我们假定构成 ZooKeeper 集群的机器总数为 N，在三个机房中部署的 ZooKeeper 服务器数分别为 N1、N2 和 N3，如果要使该 ZooKeeper 集群具有较好的容灾能力，我们可以根据如下算法来计算 ZooKeeper 集群的机器部署方案。

1. 计算 N1

如果 ZooKeeper 集群的服务器总数是 N，那么：

N1 = (N-1)/2

在 Java 中，“/” 运算符会自动对计算结果向下取整操作。举个例子，如果 N=8，那么 N1=3；如果 N=7，那么 N1 也等于 3。

2. 计算 N2 的可选值

N2 的计算规则和 N1 非常类似，只是 N2 的取值是在一个取值范围内：

N2 的取值范围是 1~(N-N1)/2

即如果 N=8，那么 N1=3，则 N2 的取值范围就是 1~2，分别是 1 和 2。注意，1 和 2 仅仅是 N2 的可选值，并非最终值——如果 N2为某个可选值的时候，无法计算出 N3 的值，那么该可选值也无效。

3. 计算 N3，同时确定 N2 的值

很显然，现在只剩下 N3 了，可以简单的认为 N3 的取值就是剩下的机器数，即：

N3 = N - N1 - N2

只是 N3 的取值必须满足 N3 < N1+N2。在满足这个条件的基础下，我们遍历步骤 2 中计算得到的 N2 的可选值，即可得到三机房部署时每个机房的服务器数量了。

现在我们以 7 台机器为例，来看看如何分配三机房的机器分布。根据算法的步骤 1，我们首先确定 N1 的取值为 3。根据算法的步骤 2，我们确定了 N2 的可选值为 1 和 2。最后根据步骤 3，我们遍历 N2 的可选值，即可得到两种部署方案，分别是（3、1、3）和（3、2、2）。以下是 Java 程序代码对以上算法的一种简单实现：

public class Allocation {

static final int n = 7;

public static void main(String[] args){

int n1,n2,n3;

n1 = (n-1) / 2;

int n2\_max = (n-n1) / 2;

for(int i=1; i<=n2\_max; i++){

n2 = i;

n3 = n - n1 -n2;

if(n3 >= (n1+n2)){

continue;

}

System.out.println("("+n1+","+n2+","+n3+")");

}

}

}

### 扩容与缩容

水平可扩容可以说是对一个分布式系统在高可用性方面提出的基本的，也是非常重要的一个要求，通过水平扩容能够帮助系统在不进行或进行极少改进工作的前提下，快速提高系统对外的服务支撑能力。简单地讲，水平扩容就是向集群中添加更多的机器，以提高系统的服务质量。

很遗憾的是，ZooKeeper 在水平扩容扩容方面做得并不十分完美，需要进行整个集群的重启。通常有两种重启方式，一种是集群整体重启，另外一种是逐台进行服务器的重启。

**整体重启**

所谓集群整体重启，就是先将整个集群停止，然后更新 ZooKeeper 的配置，然后再次启动。如果在你的系统中，ZooKeeper 并不是个非常核心的组件，并且能够允许短暂的服务停止（通常是几秒钟的时间间隔），那么不妨选择这种方式。在整体重启的过程中，所有该集群的客户端都无法连接上集群。等到集群再次启动，这些客户端就能够自动连接上——注意，整体启动前建立起的客户端会话，并不会因为此次整体重启而失效。也就是说，在整体重启期间花费的时间将不计入会话超时时间的计算中。

**逐台重启**

这种方式更适合绝大多数的实际场景。在这种方式中，每次仅仅重启集群中的一台机器，然后逐台对整个集群中的机器进行重启操作。这种方式可以在重启期间依然保证集群对外的正常服务。