一、PCA理论

二、BASE理论

三、Redis集群，增加节点，数据是如何迁移的。

四、集群选举算法raft.

五、一致性hash

六、2PC、3PC、TCC

七、GC调优，fullGC相关

fullGC时间过长（STW）时节点间心跳检测断了怎么办?

jstack--通过jstack输出的线程信息主要包括：jvm自身线程、用户线程等。其中jvm线程会在jvm启动时就会存在。对于用户线程则是在用户访问时才会生成。

jstack工具可以用来获得core文件的java stack和native stack的信息。可以发现死锁等情况。

比如

locked <0xef63beb8> (ajava.util.ArrayList)

waiting on <0xef63beb8> (ajava.util.ArrayList)

这里需要解释一下，为什么先 lock了这个对象，然后又 waiting on同一个对象呢？让我们看看这个线程对应的代码：

synchronized(obj){

.....

obj.wait();

.....

}

"http-8080-10" daemon prio=10 tid=x0a949bb60 nid=0x884 waiting for monitor entry [...]

"http-8080-10" 这个线程处于等待状态。 waiting for monitor entry 如果在连续几次输出线程堆栈信息都存在于同一个或多个线程上时，则说明系统中有锁竞争激烈，死锁，或锁饿死的想象。

jmap--内存的使用情况

jvisualVM：可视化的工具，可以看到虚拟机的各种参数配置，以及线程堆栈和对象内存使用情况，可以分析dump文件，手动GC,堆上的内存一般时锯齿状最好。我在学习的时候用过。

）。找到问题的原因，进行GC调优，避免长时间的GC产生（G1可以控制停顿时间，我的电脑上默认时CMS）；

安全点/安全区域:对象引用不会发生变化的代码段。

2.垃圾回收算法

垃圾回收算法可以分为三类，都基于标记-清除（复制）算法：

- Serial算法（单线程）

- 并行算法
- 并发算法

JVM会根据机器的硬件配置对每个内存代选择适合的回收算法，比如，如果机器多于1个核，会对年轻代选择并行算法，关于选择细节请参考JVM调优文档。

稍微解释下的是，并行算法是用多线程进行垃圾回收，回收期间会暂停程序的执行，而并发算法，也是多线程回收，但期间不停止应用执行。所以，并发算法适用于交互性高的一些程序。经过观察，并发算法会减少年轻代的大小，其实就是使用了一个大的年老代，这反过来跟并行算法相比吞吐量相对较低。

还有一个问题是，垃圾回收动作何时执行?

- 当年轻代内存满时，会引发一次普通GC，该GC仅回收年轻代。需要强调的时，年轻代满是指Eden代满，Survivor满不会引发GC
- 当年老代满时会引发Full GC，Full GC将会同时回收年轻代、年老代
- 当永久代满时也会引发Full GC，会导致Class、Method元信息的卸载

另一个问题是，何时会抛出OutOfMemoryException，并不是内存被耗空的时候才抛出

- JVM98%的时间都花费在内存回收
- 每次回收的内存小于2%

5.回归问题

Q：为什么崩溃前垃圾回收的时间越来越长?

A:根据内存模型和垃圾回收算法，垃圾回收分两部分：内存标记、清除（复制），标记部分只要内存大小固定时间是不变的，变的是复制部分，因为每次垃圾回收都有一些回收不掉的内存，所以增加了复制量，导致时间延长。所以，垃圾回收的时间也可以作为判断内存泄漏的依据

Q：为什么Full GC的次数越来越多?

A：因此内存的积累，逐渐耗尽了年老代的内存，导致新对象分配没有更多的空间，从而导致频繁的垃圾回收

Q:为什么年老代占用的内存越来越大?

A:因为年轻代的内存无法被回收，越来越多地被Copy到年老代

3.JVM参数

在JVM启动参数中，可以设置跟内存、垃圾回收相关的一些参数设置，默认情况不做任何设置JVM会工作的很好，但对一些配置很好的Server和具体的应用必须仔细调优才能获得最佳性能。通过设置我们希望达到一些目标:

- GC的时间足够的小
- GC的次数足够的少
- 发生Full GC的周期足够的长

前两个目前是相悖的，要想GC时间小必须要一个更小的堆，要保证GC次数足够少，必须保证一个更大的堆，我们只能取其平衡。

（1）针对JVM堆的设置一般，可以通过-Xms -Xmx限定其最小、最大值，为了防止垃圾收集器在最小、最大之间收缩堆而产生额外的时间，我们通常把最大、最小设置为相同的值 （2）年轻代和年老代将根据默认的比例（1：2）分配堆内存，可以通过调整二者之间的比率NewRadio来调整二者之间的大小，也可以针对回收代，比如年轻代，通过 -XX:newSize -XX:MaxNewSize来设置其绝对大小。同样，为了防止年轻代的堆收缩，我们通常会把-XX:newSize -XX:MaxNewSize设置为同样大小

（3）年轻代和年老代设置多大才算合理？这个我问题毫无疑问是没有答案的，否则也就不会有调优。我们观察一下二者大小变化有哪些影响

- 更大的年轻代必然导致更小的年老代，大的年轻代会延长普通GC的周期，但会增加每次GC的时间；小的年老代会导致更频繁的Full GC
- 更小的年轻代必然导致更大年老代，小的年轻代会导致普通GC很频繁，但每次的GC时间会更短；大的年老代会减少Full GC的频率
- 如何选择应该依赖应用程序对象生命周期的分布情况：如果应用存在大量的临时对象，应该选择更大的年轻代；如果存在相对较多的持久对象，年老代应该适当增大。但很多应用都没有这样明显的特性，在抉择时应该根据以下两点：（A）本着Full GC尽量少的原则，让年老代尽量缓存常用对象，JVM的默认比例1：2也是这个道理 （B）通过观察应用一段时间，看其他在峰值时年老代会占多少内存，在不影响Full GC的前提下，根据实际情况加大年轻代，比如可以把比例控制在1：1。但应该给年老代至少预留1/3的增长空间

（4）在配置较好的机器上（比如多核、大内存），可以为年老代选择并行收集算法：**-XX:+UseParallelOldGC**，默认为Serial收集

（5）线程堆栈的设置：每个线程默认会开启1M的堆栈，用于存放栈帧、调用参数、局部变量等，对大多数应用而言这个默认值太了，一般256K就足用。理论上，在内存不变的情况下，减少每个线程的堆栈，可以产生更多的线程，但这实际上还受限于操作系统。

重新选取leader节点啊，恢复了之后自动变为fellower节点，然后回滚，并且和当前主节点同步。

八、自己设计一个RPC框架：

服务提供者（服务端）

服务发布者（注册中心）

服务请求者：（用户）

RPC框架的核心：就是一种进程间的通信过程，允许像调用本地服务一样调用远程服务。

其实就是把要执行的参数，方法，类，等信息经过socket传给服务发布者。然后服务发布者通过反射创建服务提供者的实例对象，并调用方法进行执行，然后把执行结果返回给服务请求方。但是给人的感觉就像是在调用本地方法一样。

涉及的知识：序列化（对象转二进制码流）。通信（TCP/UDP），远程代理对象（用来代理整个调用过程，创建连接，发送信息，接受消息，返回上层调用）。

通过反射获取指定类的实例，根据方法名和参数获得执行的方法。然后调用method.invoke(实例对象，参数)；

第一章 分布式架构

1.1 从集中式到分布式

集中式的特点：

部署结构简单（因为基于底层性能卓越的大型主机，不需考虑对服务多个节点的部署，也就不用考虑多个节点之间分布式协调问题）

分布式系统是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。

分布式的特点：

- 分布性：在空间随意分布
- 对等性：没有主从之分，都是对等的

- 并发性
- 缺乏全局时钟：很难定义两个事件谁先谁后
- 故障总是会发生

分布式环境的各种问题：

- 通信异常：主要是因为网络本身的不可靠性
- 网络分区：当网络发生异常时，导致部分节点之间的网络延时不断增大，最终导致部分节点可以通信，而另一部分节点不能。
- 三态（成功、失败与超时）
- 节点故障：组成分布式系统的服务器节点出现宕机或“僵死”现象

1.2 从ACID到CAP/BASE

事务是由一系列对系统中数据进行访问与更新的操作所组成的一个程序执行逻辑单元，狭义上的事务特指数据库事务。

事务有四个特性，分别是原子性（Atomicity）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability），简称为事务的ACID特性。

分布式事务是指事务的参与者、支持事务的服务器、资源服务器以及事务管理器分别位于分布式系统的不同节点之上。也可以被定义为一种嵌套型的事务，同时也具有了ACID事务特性。

CAP理论：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。

- 一致性：数据在多个副本之间是否能够保持一致的特性。
- 可用性：系统提供的服务必须一致处于可用的状态，对于用户的每一个操作请求总是能够在有限的时间内返回结果。
- 分区容错性：分布式系统在遇到任何网络分区故障的时候，仍然需要能够保证对外提供满足一致性和可用性的服务，除非是整个网络环境都发生了故障。

从CAP定理看出，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需求。对于一个分布式系统，分区容错性是一个最基本的需求。

BASE理论：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。核心思想是即使无法做到强一致性，但每个应用都可以根据自身的业务特点，采用适当的方式来使系统达到最终一致性。

- 基本可用：分布式系统在出现不可预知故障的时候，允许损失部分可用性。
- 弱状态：也称软状态，允许系统在不同节点的数据副本之间进行数据同步存在延时。
- 最终一致性：系统中所有的数据副本，在经过一段时间的同步后，最终能够达到一个一致的状态。

总得来说BASE理论面向的是大型高可用可扩展的分布式系统，完全不同于ACID的强一致性模型，而是通过牺牲强一致性来获取可用性，并允许数据在一段时间内是不一致的，但最终达到一致状态。

实际分布式场景中，ACID特性和BASE理论会结合在一起。

第二章 一致性协议

2.1 2PC与3PC

协调者：用来统一调度所有分布式节点的执行逻辑

参与者：被调度的分布式节点

2PC：

二阶段提交(Two-phaseCommit)是指，在计算机网络以及数据库领域内，为了使基于分布式系统架构下的所有节点在进行事务提交时保持原子性和一致性而设计的一种算法。

通常，二阶段提交也被称为是一种一致性协议。

大部分关系型数据库都是采用二阶段提交协议来完成分布式事务处理的。

阶段一：提交事务请求（投票阶段）

协调者节点向所有参与者节点询问是否可以执行提交操作(vote)，并开始等待各参与者节点的响应。

参与者节点执行询问发起为止的所有事务操作，并将Undo信息和Redo信息写入日志。

各参与者节点响应协调者节点发起的询问。如果参与者节点的事务操作实际执行成功，则它返回一个“YES”消息；如果参与者节点的事务操作实际执行失败，则它返回一个"NO”消息。

阶段二：执行事务请求（执行阶段）

分为以下两种情况：

当协调者节点从所有参与者节点获得的相应消息都为“YES”时:，**执行事物提交：**

协调者节点向所有参与者节点发出”正式提交(commit)”的请求。

参与者节点接受到COMMIT请求后正式完成操作，并释放在整个事务期间内占用的资源。

参与者节点向协调者节点发送"ACK”消息。

协调者节点受到所有参与者节点反馈的"ACK”消息后，完成事务。

如果任一参与者节点在第一阶段返回的响应消息为“NO”，或者 协调者节点在第一阶段的询问超时之前无法获取所有参与者节点的响应消息时,那么就会**中断事务：**

协调者节点向所有参与者节点发出”回滚操作(rollback)”的请求。

参与者节点利用之前写入的Undo信息执行回滚，并释放在整个事务期间内占用的资源。

参与者节点向协调者节点发送"ACK”消息。

协调者节点受到所有参与者节点反馈的"ACK”消息后，完成事务中断。

优点：原理简单，实现方便。

缺点：

同步阻塞问题：执行过程中，所有参与节点都是事务阻塞型的。当参与者占有公共资源时，其他第三方节点访问公共资源不得不处于阻塞状态。

单点故障：由于协调者的重要性，一旦协调者发生故障。参与者会一直阻塞下去。尤其在第二阶段，协调者发生故障，那么所有的参与者还都处于锁定事务资源的状态中，而无法继续完成事务操作。

数据不一致：在二阶段提交的阶段二中，当协调者向参与者发送commit请求之后，发生了局部网络异常或者在发送commit请求过程中协调者发生了故障，这回导致只有一部分参与者接受到了commit请求。而在这部分参与者接到commit请求之后就会执行commit操作。但是其他部分未接到commit请求的机器则无法执行事务提交。于是整个分布式系统便出现了数据部一致性的现象。

太过保守：二阶段提交缺乏较为完善的容错机制，任意一个节点的失败导致整个事务的失败。

3PC：

阶段一：CanCommit阶段

协调者向参与者发送CanCommit请求。询问是否可以执行事务提交操作。然后开始等待参与者的响应。

参与者接到CanCommit请求之后，正常情况下，如果其自身认为可以顺利执行事务，则返回Yes响应，并进入预备状态。否则反馈No

阶段二：PreCommit阶段

有以下两种可能：

假如协调者从所有的参与者获得的反馈都是Yes响应，那么就会执行事务的预执行：

发送预提交：请求协调者向参与者发送PreCommit请求，并进入Prepared阶段。

事务预提交：参与者接收到PreCommit请求后，会执行事务操作，并将undo和redo信息记录到事务日志中。

响应反馈：如果参与者成功的执行了事务操作，则返回ACK响应，同时开始等待最终指令。

假如有任何一个参与者向协调者发送了No响应，或者等待超时之后，协调者都没有接到参与者的响应，那么就执行事务的中断：

发送中断请求：协调者向所有参与者发送abort请求。

中断事务：参与者收到来自协调者的abort请求之后（或超时之后，仍未收到协调者的请求），执行事务的中断。

阶段三：doCommit阶段

分为以下两种情况：

执行提交

发送提交请求，协调者接收到参与者发送的ACK响应，那么他将从预提交状态进入到提交状态。并向所有参与者发送doCommit请求。

事务提交：参与者接收到doCommit请求之后，执行正式的事务提交。并在完成事务提交之后释放所有事务资源。

响应反馈：事务提交完之后，向协调者发送Ack响应。

完成事务：协调者接收到所有参与者的ack响应之后，完成事务。

协调者没有接收到参与者发送的ACK响应（可能是接受者发送的不是ACK响应，也可能响应超时），那么就会执行中断事务。

发送中断请求：协调者向所有参与者发送abort请求

事务回滚：参与者接收到abort请求之后，利用其在阶段二记录的undo信息来执行事务的回滚操作，并在完成回滚之后释放所有的事务资源。

反馈结果：参与者完成事务回滚之后，向协调者发送ACK消息

中断事务：协调者接收到参与者反馈的ACK消息之后，执行事务的中断。

阶段三，可能会出现：

协调者出现问题，协调者和参与者网络出现故障。

无论哪种异常都会导致参与者无法及时接收到来自协调者的doCommit或者rebort请求时，参与者会在等待超时之后，会继续进行事务的提交。

优缺点：

优点：3PC主要解决的单点故障问题，并减少阻塞范围，因为一旦参与者无法及时收到来自协调者的信息之后，他会**默认执行commit。**而不会一直持有事务资源并处于阻塞状态。

缺点：引入新问题，如果出现网络分区，协调者发送的abort响应没有及时被参与者接收到，那么参与者在等待超时之后执行了commit操作。这样就和其他接到abort命令并执行回滚的参与者之间存在数据不一致的情况。

TCC

所谓的TCC编程模式，也是两阶段提交的一个变种。TCC提供了一个编程框架，将整个业务逻辑分为三块：Try、Confirm和Cancel三个 操作。以在线下单为例，Try阶段会去扣库存，Confirm阶段则是去更新订单状态，如果更新订单失败，则进入Cancel阶段，会去恢复库存。总 之，TCC就是通过代码人为实现了两阶段提交，不同的业务场景所写的代码都不一样，复杂度也不一样，因此，这种模式并不能很好地被复用。

2.2 Paxos算法

算法中的参与者主要分为三个角色，同时每个参与者又可兼领多个角色:

proposer 提出提案，提案信息包括提案编号和提议的value;

acceptor 收到提案后可以接受(accept)提案;

learner 只能"学习"被批准的提案;

阶段一:

Proposer选择一个提案编号Mn，向Acceptor的某个超过半数的子集成员发送编号为Mn的Prepare请求

如果一个Acceptor收到一个Mn的请求，且Mn大于它已经响应的所有请求的编号，它会将它已批准过的最大编号的提案作为响应反馈给Proposer，同时它承诺不会再批准任何编号小于Mn的提案

比如一个Acceptor已经响应过的提案编号分别为1,2,7，那么它在收到一个编号为8的Prepare请求后，会将编号为7的提案反馈给Proposer

阶段二：

如果Proposer收到半数以上的Acceptor的响应，它会发一个针对[Mn，Vn]提案的Accept请求给Acceptor。Vn是收到响应中编号最大提案的值，如果响应中不包含任何提案，那么它就是任意值

如果Acceptor收到这个针对[Mn，Vn]提案的Accept请求，只要它尚未对编号大于Mn的Prepare请求做出响应，它就可以通过这个提案

提案获取:

方案一：一旦 Acceptor批准了一个提案，就发送给所有Learner

方案二：批准的提案先发给主Learner，再由其同步给其他Learner。（主Learner可能出现故障）

方案三：批准的提案先发给一个特定的Learner集合，再由集合里的Learner通知其他Learner。

活锁问题:

P1提出M1提案，并通过阶段一，同时P2提出M2提案，并通过阶段一，P1进入阶段二，会被忽略

然后P1提出M3，P2提出M4，以此类推，则会陷入死循环

解决办法：选择一个主Proposer，只有主Proposer才能提出提案

小结：

二阶段提交协议解决了分布式事务的原子性问题

三阶段提交协议避免了2PC中的无限期等待问题

Paxos算法引入少数服从多数原则，支持角色之间的轮换，避免了单点故障，无限期等待，网络分区问题，是最优秀的分布式一致性协议之一。

第四章 ZooKeeper与Paxos

ZooKeeper没有采用Paxos算法，而是采用了一种被称为ZAB的一致性协议。

4.1 初识ZooKeeper

ZooKeeper是一个分布式协调服务。

设计目标是将那些复杂的分布式一致性服务封装起来，以接口的形式提供给用户使用。

ZooKeeper是一个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诸如数据发布/订阅，分布式锁，Master选举等功能。

ZooKeeper作为一个分布式协调框架，主要用于解决分布式数据一致性的问题。

ZooKeeper可以保证如下分布式一致性特性：

顺序一致性：同一客户端发起的事务请求，会严格地按照其发起的顺序送到ZooKeeper

原子性：要么所有节点都成功执行了事务，要么全都没有执行

单一视图：无论客户端连接的是哪个ZooKeeper服务器，看到的数据模型都是一样的

可靠性：事务引起的服务端状态变更会被一直保留下来

实时性：ZooKeeper仅仅保证一定时间内，客户端能读到最新数据

设计目标：

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

简单的数据模型：使得分布式程序能通过一个共享的，树形结构的名字空间来相互协调。其数据模型类似于一个文件系统。

可以构建集群：集群的每台机器都会在内存中维护服务器状态，并且每台机器互相保持通信，只要有半数以上的机器正常工作，就能正常对外服务。

顺序访问：每个更新请求，都有有一个全局唯一的递增编号。

高性能：ZooKeeper将全量数据存储在内存中，适用于以读为主的应用场景。

ZooKeeper的基本概念：

集群角色：Leader：提供读和写。Follower和Observer都能提供读服务。

区别：Observer不参与Leader选举，也不参与写操作的“过半写成功”策略。主要用于提升读性能。

客户端连接：客户端与服务器之间的一个TCP长连接。

会话：会话周期从第一次建立连接开始，客户端能通过心跳检测与服务器保持有效会话。

4.2 ZooKeeper的ZAB协议

ZooKeeper并没有完全采用Paxos算法，而是采用一种称为ZooKeeper Atomic Broadcast(ZAB 原子消息广播协议)作为其数据一致性的核心算法。

ZAB是一种支持崩溃恢复的原子广播协议。

ZooKeeper使用一个单一主进程来处理所有事物请求，并采用ZAB，将服务器数据的状态变更广播到所有副本进程上。

ZAB 协议的核心是定义了对于那些会改变 ZooKeeper 服务器数据状态的事务请求的处理方式，即：

> 所有事务请求必须由一个全局唯一的服务器来协调处理，这样的服务器被称为 Leader 服务器，而余下的其他服务器则成为 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 Proposal（提议），并将该 Proposal 分发给集群中所有的 Follower 服务器。之后 Leader 服务器需要等待所有 Follower 服务器的反馈，一旦超过半数的 Follower 服务器进行了正确的反馈后，那么 Leader 就会再次向所有的 Follower 服务器分发 Commit 消息，要求其将前一个 Proposal 进行提交。

协议介绍：

ZAB包括两种基本模式，分别是崩溃恢复和消息广播。

当Leader服务器出现网络中断或者崩溃，ZAB协议会进入恢复模式并选举新的Leader。

当集群中过半的Follower完成了Leader的状态同步，整个服务框架就可以进入消息广播模式了。

如果非Leader节点收到客户端的事务请求，会转发给Leader。

消息广播：

在ZAB的二阶段提交过程中，移除了中断逻辑，意味着可以在过半节点反馈ACK之后提交事务。

需要崩溃恢复模式来解决Leader崩溃带来的数据不一致问题。

在消息广播的过程中，Leader会为每一个事务Proposal分配一个全局递增唯一的ID。

每一个Follower在接收到这个事务的Proposal之后，会将其以事务日志的形式写入到磁盘中去，并向Leader反馈ACK。

当Leader收到半数节点的ACK时，会广播一个commit消息通知其他节点进行事务提交，同时自己也完成事务提交。

崩溃恢复：

Leader选举算法应该保证：已经在Leader上提交的事务最终也被其他节点都提交，即使出现了Leader挂掉，Commit消息没发出去这种情况。

确保丢弃只在Leader上被提出的事务。Leader提出一个事务后挂了，集群中别的节点都没收到，当挂掉的节点恢复后，要确保丢弃那个事务。

让Leader选举算法能够保证新选举出来的Leader拥有最大的事务ID的Proposal。

数据同步：

在完成Leader选举后，Leader会首先确认事务日志中所有Proposal是否都被集群中过半的节点提交了，即是否完成数据同步。

事务ID：

是一个64位数，低32位是一个单调递增的计数器，每一个新的Proposal产生，该计数器都会+1.

高32位代表Leader周期epoch编号，每当选举出新Leader，会取得这个Leader的最大事务ID，对其高32位+1.低32位清零。

ZAB通过epoch来区分Leader周期变化的策略，简化和提升了数据恢复的流程。

Leader和Follower通过心跳检测来感知彼此的情况。

如果Leader在指定时间内无法从过半的Follower那收到心跳检测，或者是TCP连接本身断开了，会重新选举Leader。

心跳是从follower向leader发送心跳。

ZAB主要用于构建一个高可用的分布式数据主备系统。

Paxos算法是用于构建一个分布式的一致性状态机系统。

第六章 ZooKeeper的典型应用场景

数据发布/订阅：

两种模式：推模式和拉模式。

推模式：服务端主动将数据更新发送给所有订阅的客户端。

拉模式：客户端主动发起请求来获取最新数据。

ZooKeeper采用推拉结合的方式：客户端向服务端注册自己需要关注的节点，一旦节点的数据发生变更，那么服务端就会向相应的客户端发送Watcher事件通知。客户端收到这个通知之后，需要主动到服务端获取最新的数据。

心跳检测：

可以让不同的机器都在ZooKeeper的一个指定节点下创建临时子节点。不同机器之间可以根据这个临时节点来判断对应的客户端机器是否存活。

Master选举：

利用ZooKeeper的强一致性，能够保证在分布式高并发环境下节点的创建是全局唯一的。

选择一个根节点，例如/master_select,多台机器同时向该节点创建一个子节点/master_select/lock，利用ZooKeeper的特性，最终只有一台机器能够创建成功，成功的那台就是master。

同时，其他没能成功创建节点的客户端，都会在根节点上注册一个子节点变更的Watcher，用于监控当前master是否存活。

分布式锁：

分布式锁是控制分布式系统之间同步访问共享资源的一种方式。

排他锁：

定义锁：通过ZooKeeper上的一个数据节点来表示一个锁，例如/exclusive_lock/lock节点就可以被定义为一个锁。

获取锁：在需要获取锁时，所有客户端都会试图调用create（）接口，在/exclusive_lock节点下创建临时节点/exclusive_lock/lock。

最终只有一个客户端能成功，就可以认为该客户端获取了锁。其他没获取锁的客户端会在/exclusive_lock节点上注册监听，实时监听lock节点的变更情况。

释放锁：获取锁的客户端宕机了，临时节点会被移除。

正常执行完业务逻辑后，客户端会主动删除临时节点。

共享锁：

定义锁：通过ZooKeeper上的一个数据节点来表示一个锁，例如/shared_lock/host1-R-000001

获取锁：在需要获取锁时，所有客户端都会到/shared_lock下创建临时顺序节点。

如果是读请求，就创建例如/shared_lock/host1-R-000001的节点

如果是写请求，就创建例如/shared_lock/host2-W-000002的节点

写操作必须在当前没有任何事务进行读写操作的情况下进行。

对于读请求，如果比自己序号小的节点中有写请求，就进入等待。

对于写请求，如果自己不是序号最小的子序号，就进入等待。

释放锁：和排他锁一样。

羊群效应：

客户端无端收到过多和自己不相关的事件通知。

每个节点应该只需要关注比自己小的那个节点的变更。而不需要关心全局。

改进（当集群规模较大时适用）：

读请求：向比自己序号小的**最后一个写请求节点**注册监听。

写请求：向比自己序号小的节点注册监听