1. 上课须知

每次课前的约定：

20:00 准时开始上课！ 20:00 准时开始上课！ 20:00 准时开始上课！

能听到音乐，能看到画面的小伙伴，请在直播间评论栏扣 666， 如果有其他小伙伴扣 666 证明我的直播环境是OK 的，然后听不见，或者看不到的小伙伴赶紧调整自己的上课环境。

一个晚上的上课时间是：20:00 - 23:00，中间会找机会休息一次。10分钟左右。

2. 一个逻辑思维题

题目：

```
总共64匹马，每次只能同时8匹马赛跑。请问，最少用多少次比赛，就能找出来最快的4匹马？
```

解题的核心原则：

```
能先排除的先排除，能确定的先确定，然后进行比赛，一场比赛尽可能多的排除更多的马。
```

3. 课程安排

ZooKeeper 2次
MapReduce 2次
Hive 3次
HBase 阿里大佬

4. 作业

1、搭建zookeeper集群，熟练使用 zk 的 shell 命令

2、基于zookeeper集群搭建 HA 的 HDFS 集群

5. 背景

咱们一上来，先从 **服务部署架构** 的发展历程讲起！

```
集中式：什么事情都由一台服务器搞定
分布式：多台服务器联合完成
```

扩展的方式：

```
横向扩展
纵向扩展
```

5.1. 集中式服务

所谓集中式系统就是指由一台或多台主计算机组成中心节点，数据集中存储于这个中心 节点中，并且整个系统的所有业务单元都集中部署在这个中心节点上，系统所有的功能均由其集中处理。也就是说，集中式系统中，每个终端或客户端及其仅仅负责数据的录入和输出，而数据的存储与控制处理完全交由主机来完成。

集中式服务优点：

1、结构简单
2、部署简单
3、项目架构简单

集中式服务缺点：

1、大型主机的研发人才和维护人才培养成本非常高
2、大型主机非常昂贵
3、单点故障问题，主机一挂，所有服务终止
4、大型主机的性能扩展受限于摩尔定律

补充一下摩尔定律：

> 摩尔定律是由英特尔（Intel）创始人之一戈登·摩尔（Gordon Moore）提出来的。其内容为：当价格不变时，集成电路上可容纳的元器件的数目，约每隔18-24个月便会增加一倍，性能也将提升一倍。换言之，每一美元所能买到的电脑性能，将每隔18-24个月翻一倍以上。
>
> 摘自：百度百科

摩尔定律告诉我们：纵向扩展理论上是受限，所以只能考虑横向扩展，而且理论上来说，横向扩展理论上不受限！

```
纵向扩展：提升服务器性能，上限的
横向扩展：提升服务器数量（分布式）
```

简单说：一台高性能机器搞定所有事情！集中式架构的主机就是战斗力为 1E 的浩克！

5.2. 发展趣事

不得不提阿里巴巴发起的 "去IOE" 运动(IOE 指的是 IBM 小型机、Oracle 数据库、EMC 的高端存储)。为什么要去IOE?

```
1、企业成本越来越高，升级单机处理能力的性价比越来越低
2、单机处理能力存在瓶颈
3、稳定性和可用性这两个指标很难达到
```

5.3. 分布式服务

在《分布式系统概念与设计》注 一书中，对分布式系统做了如下定义：**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。**

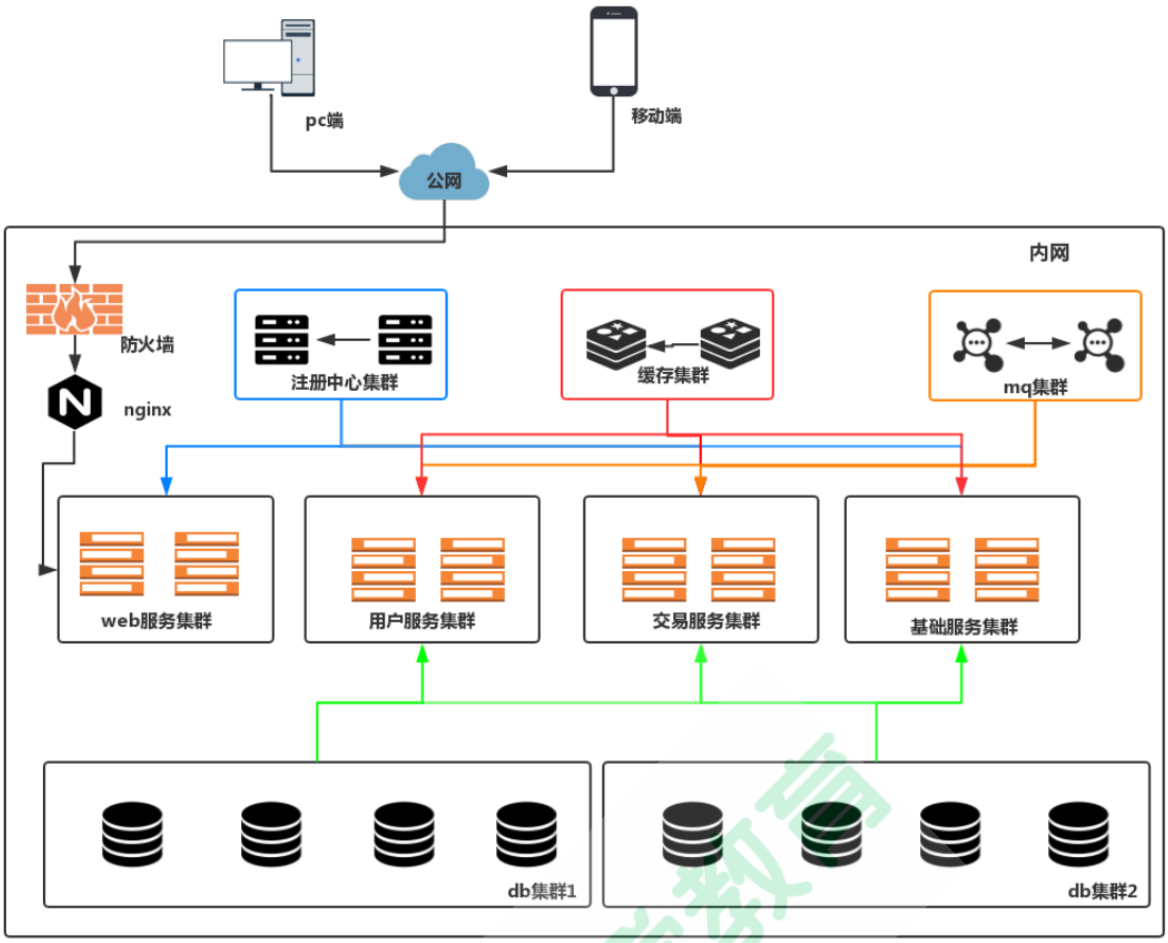
简单来说就是一群独立计算机集合共同对外提供服务，但是对于普通的用户来说，就像是一台计算机在提供服务一样。

分布式意味着可以采用更多的普通计算机（相对于昂贵的大型机）组成分布式集群对外提供服务。计算机越多，CPU、内存、存储资源等也就越多，能够处理的并发访问量也就越大。

一个由分布式系统实现的电子商城，在功能上可能被拆分成多个应用，分别提供不同的功能，组成一个分布式系统对外提供服务。

而系统内的各个子系统之间通过网络进行通信和协调，如异步消息或者 RPC/HTTP 请求调用等。

所以，分布式系统中的计算机在空间上几乎没有任何限制，这些计算机可能被放在不同的机柜上，也可能被部署在不同的机房中，还可能在不同的城市中，对于大型的网站甚至可能分布在不同的国家和地区。



分布式系统的特点：

```
分布性：分布式系统中的多台计算机都会在空间上随意分布，同时，它们的分布情况也会随时变动
对等性：集群中的每个工作节点的角色是一样的。注意副本这个概念
并发性：多个机器可能同时操作一个数据库或者存储系统，可能引发数据不一致的问题（串行，并行，并发）
缺乏全局时钟：分布式系统中的多个主机上的事件的先后顺序很难界定（分布式场景中最复杂的一个问题之一）
故障总发生（服务器宕机，网络拥堵和延迟）：组成分布式系统的所有计算机，都有可能发生任何形式的故障。
```

先注意两个概念的区别：

```
传统集群：大量兄弟组成团队，大家做一样的事，只是这些事的数量比较大
分布式：大量兄弟组成团队，每个成员完成其中的一部分，大家做的是不同的事情。
```

和集中式系统相比，分布式系统的性价比更高、处理能力更强、可靠性更高、也有很好的扩展性。

但是，分布式在解决了网站的高并发问题的同时也带来了一些其他问题。

首先，分布式的必要条件就是网络，这可能对性能甚至服务能力造成一定的影响。其次，一个集群中的服务器数量越多，服务器宕机的概率也就越大。另外，由于服务在集群中分布式部署，用户的请求只会落到其中一台机器上，所以，一旦处理不好就很 **容易产生数据一致性问题。**

简单说：分布式集群发挥人多力量的优势，分布式集群就是一个每个兵的战斗力都为10000但是有10000个兵的军队！

5.4. 分布式异常问题

1、通信异常：网络不可用（消息延迟或者丢失），会导致分布式系统内部无法顺利进行一次网络通信，所以可能造成多节点数据丢失和状态不一致，还有可能造成数据乱序。

2、网络分区：网络不连通，但各个子网络的内部网络是正常的，从而导致整个系统的网络环境被切分成了若干个孤立的区域，分布式系统就会出现局部小集群造成数据不一致。

3、节点故障/机器宕机：服务器节点出现的宕机或"僵死"现象，这是常态，而不是异常

4、分布式三态：即成功、失败和超时，分布式环境中的网络通信请求发送和结果响应都有可能丢失，所以请求发起方无法确定消息是否处理成功。

5、存储数据丢失：对于有状态节点来说，数据丢失意味着状态丢失，通常只能从其他节点读取、恢复存储的状态。解决方案：副本协议

6、异常处理原则：被大量工程实践所检验过的异常处理黄金原则是：任何在设计阶段考虑到的异常情况一定会在系统实际运行中发生，但在系统实际运行遇到的异常却很有可能在设计时未能考虑，所以，除非需求指标允许，在系统设计时不能放过任何异常情况。

5.5. 衡量分布式系统的性能指标

1、性能：下面三个性能指标往往会相互制约，追求高吞吐的系统，往往很难做到低延迟；系统平均响应时间较长时，也很难提高QPS。（并行，串行，并发）

```
系统的吞吐能力，指系统在某一时间可以处理的数据总量，通常可以用系统每秒处理的总数据量来衡量；
系统的响应延迟，指系统完成某一功能需要使用的时间；
系统的并发能力，指系统可以同时完成某一功能的能力，通常也用QPS(query per second)来衡量。
```

2、可用性：系统的可用性(availability)指系统在面对各种异常时可以正确提供服务的能力。系统的可用性可以用系统停服务的时间与正常服务的时间的比例来衡量，也可以用某功能的失败次数与成功次数的比例来衡量。可用性是分布式的重要指标，衡量了系统的鲁棒性，是系统容错能力的体现。（5个9的可靠性：一年只有5分钟的宕机时间！,6个9的可靠性，也就是31秒）

3、可扩展性：系统的可扩展性(scalability)指分布式系统通过扩展集群机器规模提高系统性能（吞吐、延迟、并发）、存储容量、计算能力的特性。好的分布式系统总在追求“线性扩展性”，也就是使得系统的某一指标可以随着集群中的机器数量线性增长。

4、一致性：分布式系统为了提高可用性，总是不可避免的使用副本的机制，从而引发副本一致性的问题。越是强的一致的性模型，对于用户使用来说使用起来越简单。

5.6. 一致性理解

1、强一致性：写操作完成之后，读操作一定能读到最新数据。在分布式场景中，很难实现，后续的 Paxos 算法，Quorum 机制，ZAB 协议等能实现！

2、弱一致性：不承诺立即可以读到写入的值，也不承诺多久之后数据能够达到一致，但会尽可能地保证到某个时间级别（比如秒级别）后，数据能够达到一致状态。

3、读写一致性：用户读取自己写入结果的一致性，保证用户永远能够第一时间看到自己更新的内容。比如我们发一条朋友圈，朋友圈的内容是不是第一时间被朋友看见不重要，但是一定要显示在自己的列表上。

解决方案：

```
1、一种方案是对于一些特定的内容我们每次都去主库读取。（问题主库压力大）
2、我们设置一个更新时间窗口，在刚更新的一段时间内，我们默认都从主库读取，过了这个窗口之后，我们会挑选最近更新的从库进行读取
3、我们直接记录用户更新的时间戳，在请求的时候把这个时间戳带上，凡是最后更新时间小于这个时间戳的从库都不予以响应。
```

4、单调读一致性：本次读到的数据不能比上次读到的旧。多次刷新返回旧数据出现灵异事件。解决方案：通过hash映射到同一台机器。

5、因果一致性： 如果节点 A 在更新完某个数据后通知了节点 B，那么节点 B 之后对该数据的访问和修改都是基于 A 更新后的值。于此同时，和节点 A 无因果关系的节点 C 的数据访问则没有这样的限制。

6、最终一致性： 是所有分布式一致性模型当中最弱的。不考虑中间的任何状态，只保证经过一段时间之后，最终系统内数据正确。它最大程度上保证了系统的并发能力，也因此，在高并发的场景下，它也是使用最广的一致性模型。

5.7. 分布式一致性的作用

分布式一致性的作用：

- 为了提高系统的可用性，以防止单点故障引起的系统不可用
- 提高系统的整体性能，通过负载均衡技术，能够让分布在不同地方的数据副本，都能够为用户提供服务

其实上面这么多的内容组中只有一个用处：引出一个问题：

分布式系统的数据一致性的问题！

解决方案：

```
事务 + 分布式事务
分布式一致性算法
Quorum机制
CAP 和 BASE 理论
```

6. 分布式事务

事务：单机存储系统中。用来保证存储系统的数据状态的一致性的

```
广义上的概念：一个事务中的所有操作，要么都成功，要么都不成功，没有中间状态
狭义上的事务：  数据库的事务
```

特征：

```
ACID ：  原子性，  一致性，持久性，  隔离性
```

举例：

转账：单机 ===> 分布式

```
A 账户 - 1000
B 账户 + 1000
```

前提：分布式系统中，每个节点都能知道自己的事务操作是否成功，但是没法知道系统中的其他节点的事务是否成功。这就有可能会造成分布式系统中的各节点的状态出现不一致。因此当一个事务需要跨越服务器节点，并且要保证事务的ACID特性时，就必须引入一个 "协调者" 的角色。那么其他的各个进行事务操作的节点就都叫做 "参与者"。

典型的两种分布式事务的提交模式：2PC 和 3PC

6.1. 2PC两阶段提交

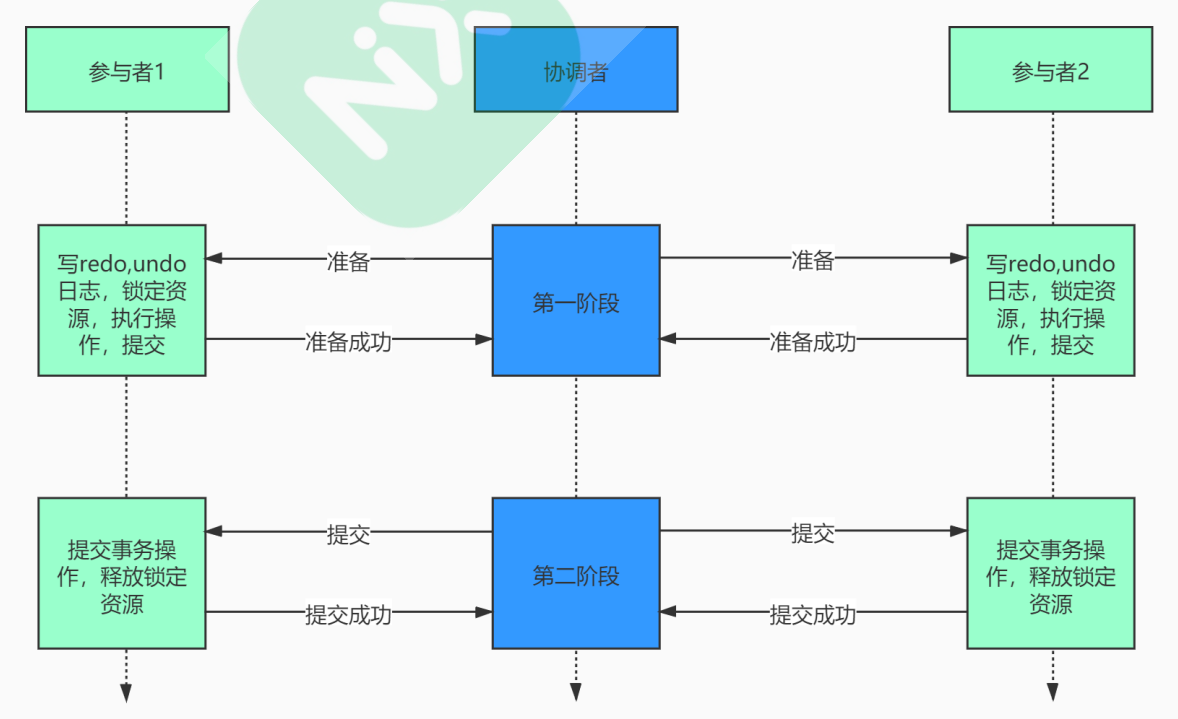
6.1.1. 执行过程解析

第一阶段：请求/表决阶段

```
1、在分布式事务发起者向分布式事务协调者发送请求的时候，事务协调者向所有参与者发送事务预处理请求（vote request）
2、这个时候参与者会开启本地事务并开始执行本地事务，执行完成后不会commit，而是向事务协调者报告是否可以处理本次事务
```

第二阶段：提交/执行/回滚阶段

```
分布式事务协调者收到所有参与者反馈后，所有参与者节点均响应可以提交，则通知参与者和发起者执行commit，否则rollback
```



6.1.2. 2PC的问题

第一点：性能问题（同步阻塞）

从流程上面可以看出，最大的缺点就是在执行过程中节点都处于阻塞状态。各个操作数据库的节点都占用着数据库资源，只有当所有节点准备完毕，事务协调者才会通知进行全局commit/rollback，参与者进行本地事务commit/rollback之后才会释放资源，对性能影响较大。

第二点：单点故障问题（协调者可能宕机）

事务协调者是整个分布式事务的核心，一旦事务协调者出现故障，会导致参与者收不到commit/rollback的通知，从而导致参与者节点一直处于事务无法完成的中间状态。

第三点：数据不一致（消息丢失问题）

在第二阶段的时候，如果发生局部网络问题，一部分事务参与者收不到 commit/rollback 消息，那么就会导致节点间数据不一致。

第四点：太过保守（没有容错机制）

必须收到所有参与的正反馈才提交事务：如果有任意一个事务参与者的响应没有收到，则整个事务失败回滚。

6.2. 3PC三阶段提交

3PC(three-phase commit)即三阶段提交，是2阶段提交的改进版，其将二阶段提交协议的 "提交事务请求" 一分为二，形成了**cancommit，precommit，docommit** 三个阶段。

除了在 2PC 的基础上 **增加了CanCommit阶段**，还引入了**超时机制。**一旦事务参与者指定时间没有收到协调者的 commit/rollback 指令，就会自动本地 commit，这样可以解决协调者单点故障的问题。

6.2.1. 执行过程解析

第一阶段：CanCommit阶段（提交询问）

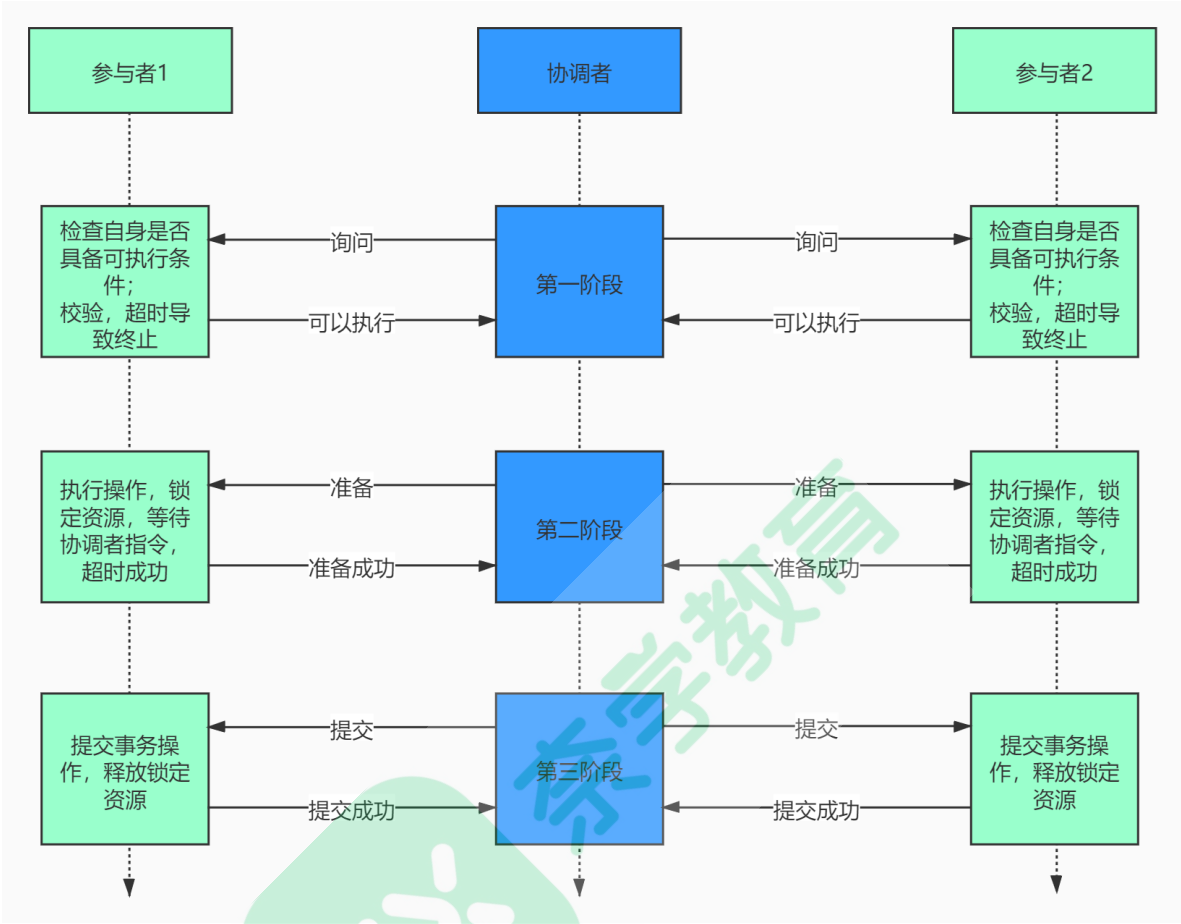
分布式事务协调者询问所有参与者是否可以进行事务操作，参与者根据自身健康情况，是否可以执行事务操作响应Y/N。

第二阶段：PreCommit阶段（预提交）

1、如果参与者返回的都是同意，协调者则向所有参与者发送预提交请求，并进入prepared阶段。
2、参与者收到预提交请求后，执行事务操作，并保存Undo和Redo信息到事务日志中。
3、参与者执行完本地事务之后（uncommitted），会向协调者发出Ack表示已准备好提交，并等待协调者下一步指令。
4、如果协调者收到预提交响应为拒绝或者超时，则执行中断事务操作，通知各参与者中断事务（abort）。
5、参与者收到中断事务（abort）或者等待超时，都会主动中断事务/直接提交。

第三阶段：doCommit阶段（最终提交）

1、协调者收到所有参与者的Ack，则从预提交进入提交阶段，并向各参与者发送提交请求。
2、参与者收到提交请求，正式提交事务（commit），并向协调者反馈提交结果Y/N。
3、协调者收到所有反馈消息，完成分布式事务。
4、如果协调者超时没有收到反馈，则发送中断事务指令（abort）。
5、参与者收到中断事务指令后，利用事务日志进行rollback。
6、参与者反馈回滚结果，协调者接收反馈结果或者超时，完成中断事务。



6.2.2. 3PC的问题

第一点：降低阻塞范围

相对于二级段提交协议，三阶段提交协议的最大的优点就是降低了事务参与者的阻塞的范围，并且能够在出现单点故障后继续达成一致。对于协调者和参与者都设置了超时机制（在2PC中，只有协调者拥有超时机制，即如果在一定时间内没有收到参与者的消息则默认失败），主要是避免了参与者在长时间无法与协调者节点通讯（协调者挂掉了）的情况下，无法释放资源的问题，因为参与者自身拥有超时机制会在超时后，自动进行本地commit从而进行释放资源。而这种机制也侧面降低了整个事务的阻塞时间和范围。

第二点：最后提交以前状态一致

通过CanCommit、PreCommit、DoCommit三个阶段的设计，相较于2PC而言，多设置了一个缓冲阶段保证了在最后提交阶段之前各参与节点的状态是一致的 。

第三点：依然可能数据不一致

三阶段提交协议在去除阻塞的同时也引入了新的问题，那就是参与者接收到 precommit 消息后，如果出现网络分区，此时协调者所在的节点和参与者无法进行正常的网络通信，在这种情况下，该参与者依然会进行事务的提交，这必然出现数据的不一致性。

7. 分布式一致性算法详解

网络分区一定存在！（消息的延迟和丢失的可能性一定是有的！）就需要在就算发生了网络分区，也能保证分布式数据一致！

7.1. Paxos算法

Paxos 算法是 Lesile Lamport 提出的一种基于消息传递且具有高度容错特性的一致性算法。

分布式系统中的节点通信存在两种模型: 共享内存和消息传递。

基于消息传递通信模型的分布式系统，不可避免会发生进程变慢被杀死，消息延迟、丢失、重复等问题，Paxos算法就是在存在以上异常的情况下仍能保持一致性的协议。

Paxos 算法使用一个希腊故事来描述，在 Paxos 中，存在三种角色，分别为

```
1、Proposer(提议者，用来发出提案proposal)，
2、Acceptor(接受者，可以接受或拒绝提案)，
3、Learner(学习者，学习被选定的提案，当提案被超过半数的Acceptor接受后为被批准)。
```

映射到 zookeeper 集群：

```
leader：发起提案          主席，  单点故障（解决方案：leader选举机制）
follower：参与投票          人大代表
observer：  被动接受          全国所有人
```

议会制：

```
弱一致性：保证超过半数达成一致即可的协议
```

下面更精确的定义 Paxos 要解决的问题：

```
1、决议(value)只有在被proposer提出后才能被批准
2、在一次Paxos算法的执行实例中，只批准(chose)一个value，  multi-paxos
3、learner只能获得被批准(chosen)的value
```

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，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。

7.2. Raft算法

Etcd 的底层实现： Raft算法

想要了解 Raft 算法，给你一个九阳神功密卷：http://thesecretlivesofdata.com/raft/
只要花费不到10分钟的时间就可以搞定这个算法了！

```
follower: 没有线条
candidate: 候选人  虚线
leader: 领导者  实线
```

7.3. ZAB协议

ZooKeeper的底层实现: ZAB 协议

ZooKeeper的底层工作机制，就是依靠 ZAB 实现的。实现 崩溃回复 和 消息广播 两个主要功能。

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服务器都提交。
ZAB协议需要确保丢弃那些只在 leader 服务器上被提出的事务。

如果让 leader 选举算法能够保证新选举出来的 leader 服务器拥有集群中所有机器最高事务编号（ZXID）的事务proposal，那么就可以保证这个新选举出来的 leader 一定具有所有已经提交的提案。

ZAB两种基本的模式：崩溃恢复和消息广播。

8. 抽屉原理/鸽巢原理

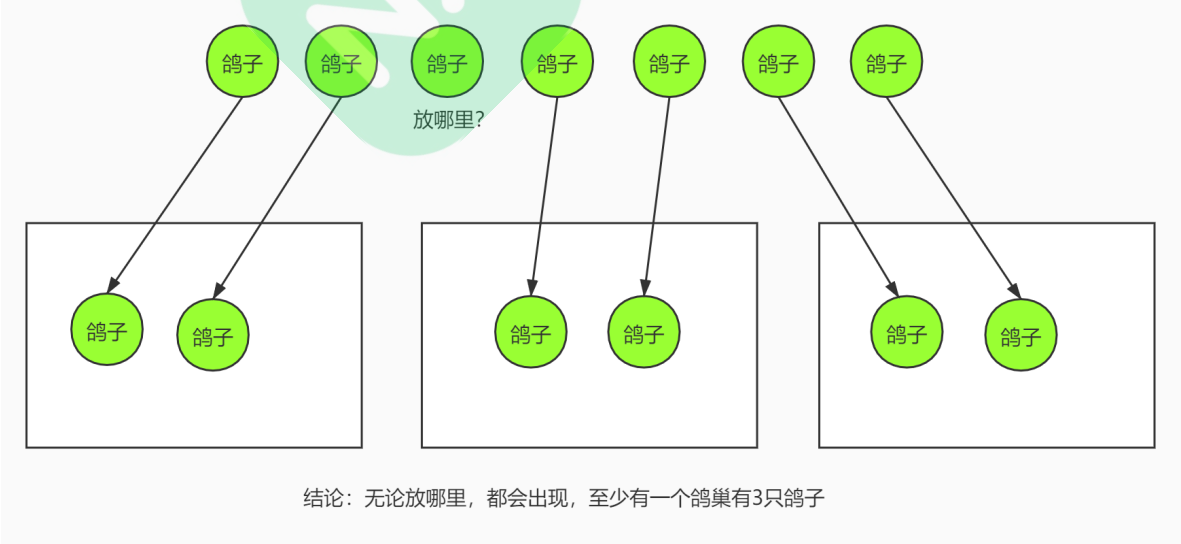
鸽巢原理，又名狄利克雷抽屉原理、鸽笼原理。

其中一种简单的表述法为：

- 若有n个笼子和n+1只鸽子，所有的鸽子都被关在鸽笼里，那么至少有一个笼子有至少2只鸽子。

另一种为：

- 若有n个笼子和kn+1只鸽子，所有的鸽子都被关在鸽笼里，那么至少有一个笼子有至少k+1只鸽子。



为什么从抽屉原理说起？一来大家对这个比较熟悉，也容易理解，二来它与 Quorum 机制有异曲同工的地方。

回顾抽屉原理，2个抽屉每个抽屉最多容纳2个苹果，现在有3个苹果无论怎么放，其中的一个抽屉里面肯定会有2个苹果。那么我们把抽屉原理变变型，2个抽屉一个放了2个红苹果，另一个放了2个青苹果，我们取出3个苹果，无论怎么取至少有1个是红苹果，这个理解起来也很简单。我们把红苹果看成更新了的有效数据，青苹果看成未更新的无效数据。便可以看出来，不需要更新全部数据（并非全部是红苹

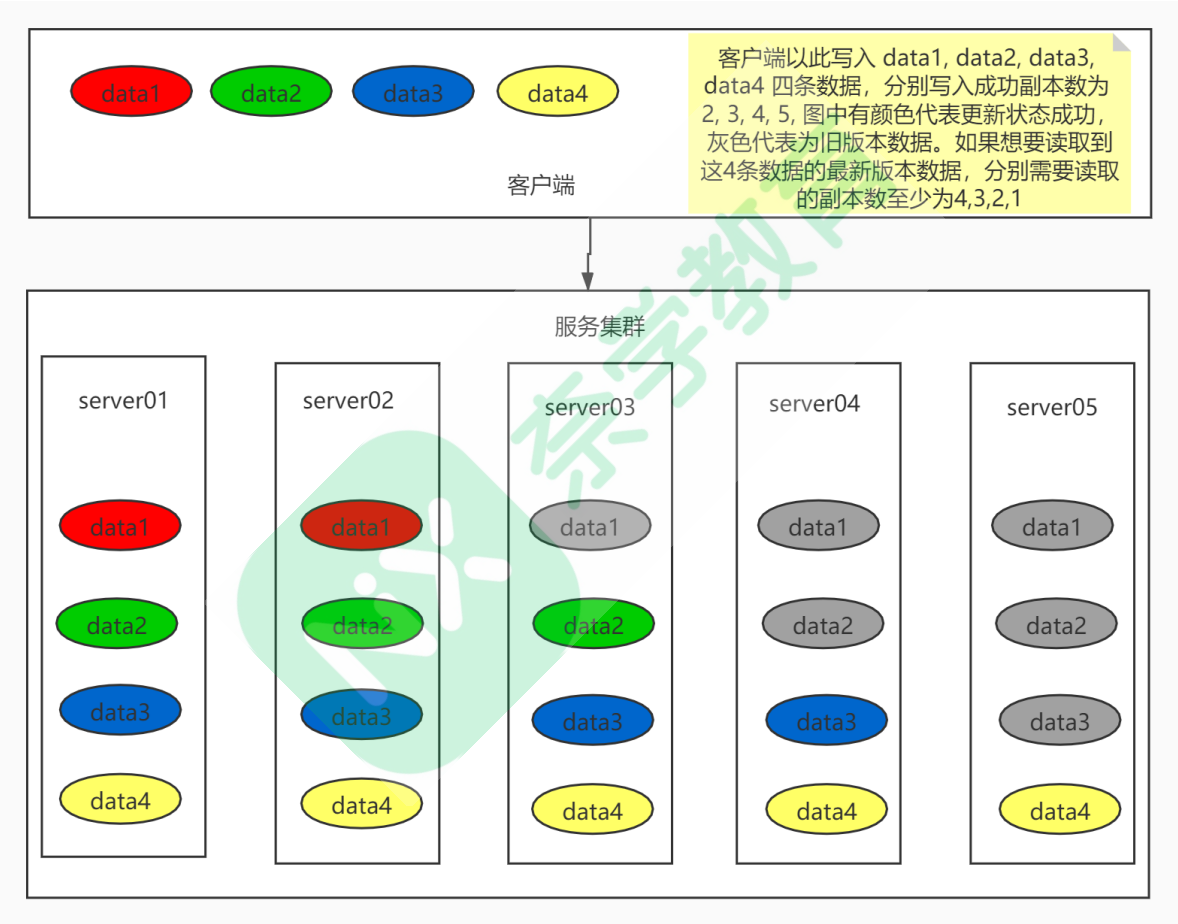
果）我们就可以得到有效数据，当然我们需要读取多个副本完成（取出多个苹果）。

9. Quorum NWR机制

Quorum NWR：Quorum 机制是分布式场景中常用的，用来保证数据安全，并且在分布式环境中实现最终一致性的投票算法。这种算法的主要原理来源于鸽巢原理。它最大的优势，既能实现强一致性，而且还能自定义一致性级别！

> Write to all copies with latest version N, wait synchronously for W success Read from all copies, wait for first R responses, pick the highest version number

N：复制的节点数，即一份数据被保存的副本数。
W：写操作成功的节点数，即每次数据写入写成功的副本数。W 肯定是小于等于 N 的。
R：读操作获取最新版本数据所需的最小节点数，即每次读取成功至少需要读取的副本数。



总结：**这三个因素决定了可用性，一致性 和 分区容错性。只要保证（W + R > N）就一定能读取到最新的数据，数据一致性级别完全可以根据读写副本数的约束来达到强一致性！**

分以下三种情况讨论：前提，当 N 已经固定了。

- **W = 1, R = N，Write Once Read All**

  在分布式环境中，写一份，那么如果要读取到最新数据，就必须要读取所有节点，然后取最新版本的值了。写操作高效，但是读操作效率低。一致性高，分区容错性差，可用性低

- **R = 1, W = N, Read Only Write All**

  在分布式环境中，所有节点都同步完毕，才能读取，所以只要读取任意一个节点就可以读取到最新数据。读操作高效，但是写操作效率低。分区容错性好，一致性差，实现难度更高，可用性高

- **W = Q, R = Q where Q = N/2 + 1**

可以简单理解为写超过一半节点，那么读也超过一半节点，取得读写性能平衡。一般应用适用，读写性能之间取得平衡。如 N=3, W=2, R=2，分区容错性，可用性，一致性取得一个平衡。

ZooKeeper 就是这么干的！采用了第三种情况！

分布式数据库：

```
HBase:  CP 数据的一致性
Cassandra:   AP 追求可用性
```

有没有 CA 的分布式系统？ 没有！ 单机系统是 CA 类型！ MySQL Oracle

10. CAP理论和BASE理论详解

10.1. CAP理论

CAP 理论：2000 年 7 月份被首次提出，CAP 理论告诉我们，一个分布式系统不可能同时满足 C,A,P 三个需求

- **C：Consistency，强一致性**

  分布式环境中多个数据副本保持一致

- **A：Availability，高可用性**

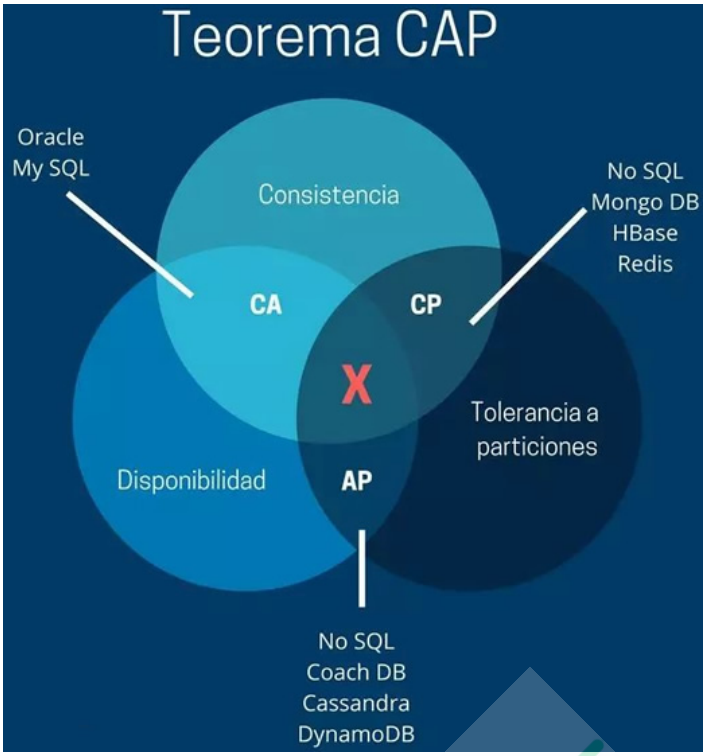
  系统提供的服务必须一直处于可用，对于用户的每一个操作请求总是能在有限时间内返回结果

- **P：Partition Tolerance 分区容错性**

  分布式系统在遇到任何网络分区故障时，仍然需要能够保证对外提供满足一致性和可用性的服务

既然一个分布式系统不能同时满足 C,A,P 三个需求，那么如何抉择呢？

- **放弃 P**：最简单的极端做法，就是放置在一个节点上，也就只有一个数据副本，所有读写操作就集中在一台服务器上，有单点故障问题。放弃 P 也就意味着放弃了系统的可扩展性，所以分布式系统一般来说，都会保证 P
- **放弃 A**：一旦系统遇到网络分区或者其他故障时，服务需要等待一段时间，在等待时间内就无法正常对外提供服务，即服务不可用
- **放弃 C**：事实上，放弃一致性是指放弃数据的强一致性，而保留最终一致性，具体多久达到数据同步取决于存储系统的设计



CAP只能3选2，因为在分布式系统中，容错性P肯定是必须有的，所以这时候无非就两种情况，网络问题导致要么错误返回，要么阻塞等待，前者牺牲了一致性，后者牺牲了可用性。

经验总结：

- 架构师不要花费精力浪费在设计同时满足CAP的分布式系统
- 分区容错性往往是分布式系统必然要面对和解决的问题。所以架构师应该把精力放在如何根据业务特点在A和C之间寻求平衡。
- 对于单机软件，因为不用考虑P，所以肯定是 CA 型，比如 MySQL
- 对于分布式软件，因为一定会考虑P，所以又不能兼顾A和C的情况下，只能在A和C做权衡，比如 HBase, Redis 等。做到服务基本可用，并且数据最终一致即可。

所以，就产生了 BASE 理论。

10.2. BASE理论

多数情况下，其实我们也并非一定要求强一致性，部分业务可以容忍一定程度的延迟一致，所以为了兼顾效率，发展出来了最终一致性理论 BASE，来自 ebay 的架构师提出。BASE理论全称：全称：Basically Available(基本可用)，Soft state（软状态）,和 Eventually consistent（最终一致性）三个 短语的缩写。**核心思想是：即使无法做到强一致性，但每个应用都可 以根据自身业务特点，采用适当的方式来使系统达到最终一致性。**一句话概括，做事别走极端，BASE 是对 CAP 理论中的 C 和 A 进行权衡得到的结果。

```
不是强一致，而是最终一致
不是高可用，而是基本可用。
```

- **Basically Available（基本可用）**：基本可用是指分布式系统在出现故障的时候，允许损失部分可用性，即保证核心可用

  响应时间的损失：出现故障或者高峰，查询结果可适当延长，以用户体验上限为主。
  功能上的损失：例如淘宝双11，为保护系统稳定性，正常下单，其他边缘服务可暂时不可用。

- **Soft State（软状态）**：软状态是指允许系统存在中间状态，而该中间状态不会影响系统整体可用性。分布式存储中一般一份数据至少会有三个副本，允许不同节点间副本同步的延时就是软状态的体现。通俗的讲：允许存在不同节点同步数据时出现延迟，且出现数据同步延迟时存在的中间状态也不会影响系统的整体性能
- **Eventually Consistent（最终一致）**：最终一致性是指系统中的所有数据副本经过一定时间后，最终能够达到一致的状态。弱一致性和强一致性相反，最终一致性是弱一致性的一种特殊情况，要求最终达到一致，而不是实时强一致

以上所有理论的总结：

```
集中式 和 分布式服务部署架构的分析
    设计分布式系统会遇到的各种难题：数据一致性的问题
2PC  3PC
    通用的思路实现，还是有缺点
Paxos  Raft  ZAB
    就算出现了分布式网络通信异常等相关棘手的问题，以上这些算法也能实现一致性
    拜占庭将军 问题！
        消息丢失
        消息被恶意更改
议会制 QuorumNWR机制
    R + W > N  ===> 少数服从多数
    一致性 和 可用性的冲突问题
CAP 和  BASE
    分布式系统一定要满足 P
    当然就只能在 C 和 A 中做权衡。！
    绝大部分系统都是 BASE 系统（基本可用 + 最终一致）
```

以后，开发分布式系统，根据业务来决定到底追求 高可用 还是追求 强一致性！

区块链：POW协议！ Seata TCC

11. ZooKeeper介绍

11.1. ZooKeeper介绍

看官网吧：https://zookeeper.apache.org/

> What is ZooKeeper?
>
> **Apache ZooKeeper is an effort to develop and maintain an open-source server which enables highly reliable distributed coordination**
>
> ZooKeeper is a centralized service for **maintaining configuration information**, **naming**, **providing distributed synchronization**, and **providing group services**. All of these kinds of services are used in some form or another by distributed applications. Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable. Because of the difficulty of implementing these kinds of services, applications initially usually skimp on them ,which make them brittle in the presence of change and difficult to manage. Even when done correctly, different implementations of these services lead to management complexity when the applications are deployed

ZooKeeper 是一个分布式的，开放源码的分布式应用程序协调服务，是 Google 的 Chubby 一个开源的实现。它提供了简单原始的功能，分布式应用可以基于它实现更高级的服务，比如**分布式同步，配置管理，集群管理，命名管理，队列管理**。它被设计为易于编程，使用文件系统目录树作为数据模型。服务端跑在 Java 上，提供 Java 和 C 的客户端 API。

```
GFS         HDFS          分布式文件系统
BigTable     HBase           分布式数据库
MapReduce        MapReduce       分布式计算系统
```

他们都共同提到了一个技术：chubby 服务协调的！ ZooKeeper就是用来解决这些分布式系统共同存在的疑难杂症！

zoo + keeper hadoop 大象 hbase 海豚 hive 蜜蜂 pig 猪， ...

```
但凡你发现  分布式领域中，有各种疑难杂症。 找zookeeper  动物园管理员
hadoop 小象，  hbase 海豚， hive 蜜蜂 ..
```

众所周知，协调服务非常容易出错，但是却很难恢复正常，例如，协调服务很容易处于竞态以至于出现死锁。我们设计 ZooKeeper 的目的是为了减轻分布式应用程序所承担的协调任务。

ZooKeeper 是集群的管理者，监视着集群中各节点的状态，根据节点提交的反馈进行下一步合理的操作。最终，将简单易用的接口和功能稳定，性能高效的系统提供给用户。

官网地址：http://ZooKeeper.apache.org/
官网快速开始地址：http://zookeeper.apache.org/doc/current/zookeeperStarted.html
官网API地址：http://ZooKeeper.apache.org/doc/r3.4.10/api/index.html

解决问题的解决方案

```
1、遇到问题，能找人问，就理解找人问
2、自己思考问题可能出现的原因，逐个排查
3、第二步花费的时间，最多半个小时，上网搜
4、来问我
5、官网详细解读（不全，晦涩难懂）和源码
```

12. ZooKeeper的核心架构设计和工作机制

Zookeeper是一个分布式数据一致性的解决方案，分布式应用可以基于它实现诸如数据发布/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、且具有严格的顺序访问控制能力的分布式协调系统。

```
zookeeper分布式集群，当中的所有节点，都保存了这个集群中的所有数据。
任意一条数据，都被保存在了zookeeper的所有的服务器节点
zookeeper集群有9个节点，每条数据，就被写入了9个副本
```

zookeeper集群没有单点故障的！ HDFS的架构：伪分布式，普通分布式，HA集群，联邦集群

HA 就是基于zookeeper , 那就让这个集群中的所有节点的状态（数据的状态）都是一致的

zookeeper paxos实现的， 其实每个follower都可以发起提议的！

```
propersor
acceptor        中谁发起了提议，谁就是 propersor，
learner
```

如果让 zk 集群中的所有节点，都具备发起提议的功能，: 没法保证全局时钟序列

zk 集群会专门选择一个节点用来处理所有的 提议发起（只有一个节点，具备发起提议的权利）

```
保证严格有序：
    1、全局有序：服务端先接收到的请求，也一定会先处理。
    2、偏序： 客户端先发送的请求，服务端一定会先处理
zk内部的所有的事务，都是串行排队执行
```

zk内部就实现了一个选举算法: 正常来说，所有的节点都是 looking 状态（找leader状态） 必然会有节点发起提议来选leader，如果选举成功，则有一个 looking 状态的节点成为 leader< 则其他节点自动成为follower节点

这个leader负责这个zk集群内部的所有事务操作，也就相当于是一个分布式协调者。

leader和所有的follower 都是负责处理读数据请求

12.1. 概述

ZooKeeper 作为一个集群提供数据一致的协调服务，自然，最好的方式就是在整个集群中的各服务节点进行数据的复制和同步。通俗的讲，就是 ZooKeeper 以一个集群的方式对外提供协调服务，**集群内部的所有节点都保存了一份完整的数据**。其中一个主节点用来做集群管理提供写数据服务，其他的从节点用来同步数据，提供读数据服务。这些从节点必须保持和主节点的数据状态一致。

总结要点：

```
1、集群角色：在ZooKeeper中，没有沿用Master/Slave(主备)概念，而是引入了Leader、Follwer、Observer三种角色。通过Leader选举来选定一台Leader机器，Leader机器为客户端提供读写服务，其他角色提供读服务，唯一区别就是Observer不参与Leader选举过程、写操作过半成功策略，因此Oberver可以在不影响写性能情况下提高集群性能。
2、ZooKeeper集群中的所有节点的数据状态通过ZAB协议保持一致
```

数据复制的好处：

```
1、容错：一个节点出错，数据不丢失，不至于让整个集群无法提供服务
2、扩展性：通过增加服务器节点能提高 ZooKeeper 系统的负载能力，把读写负载分布到多个节点上
3、高性能：客户端可访问本地 ZooKeeper 节点或者访问就近的节点，依次提高用户的访问速度
```

12.2. ZooKeeper的设计目的/架构特点

- **最终一致性**

```
client 不论连接到哪个 Server，展示给它都是同一个数据视图，这是 ZooKeeper 最重要的性能。
```

- **可靠性**

具有简单、健壮、良好的性能，如果消息 message 被到一台服务器接受，那么它将被所有的服务器接受。

- **实时性**

ZooKeeper 保证客户端将在一个时间间隔范围内获得服务器的更新信息，或者服务器失效的信息。但由于网络延时等原因，ZooKeeper不能保证两个客户端能同时得到刚更新的数据，如果需要最新数据，应该在读数据之前调用 sync() 接口。

- **等待无关（wait-free）**

慢的或者失效的 client 不得干预快速的 client 的请求，使得每个 client 都能有效的等待。

- **原子性**

事务操作只能成功或者失败，没有中间状态。通过ZAB协议实现！

- **顺序性**

TCP协议的保证消息的全序特性，先发的消息，服务器先收到。
Leader的因果顺序，包括全局有序和偏序两种：
1、全局有序：如果在一台服务器上消息 a 在消息 b 前发布，则在所有 Server 上消息 a 都将在消息 b 前被发布；
2、偏序：指如果一个消息 b 在消息 a 后被同一个发送者发布，a 必将排在 b 前面。

13. ZooKeeper集群安装

见安装文档

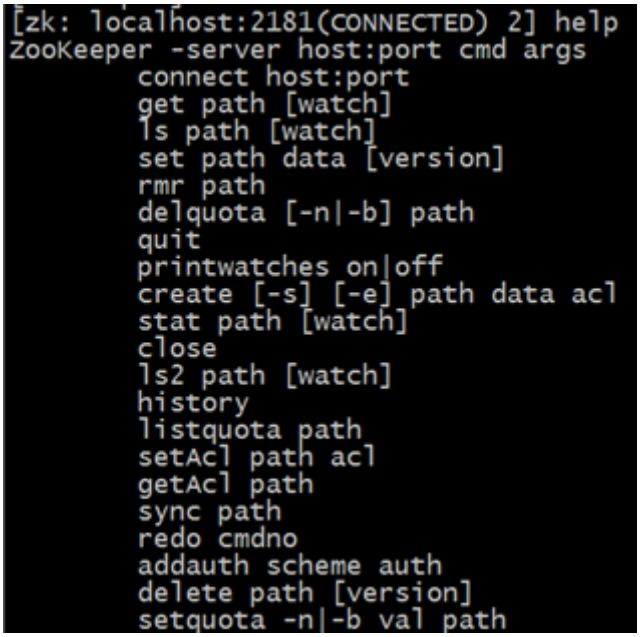
14. ZooKeeper Shell使用

14.1. 集群的命令使用

首先，我们可以是用命令 bin/zkCli.sh 进入 ZooKeeper 的命令行客户端，这种是直接连接本机的 ZooKeeper 服务器，还有一种方式，可以连接其他的 ZooKeeper 服务器，只需要我们在命令后面接一个参数 -server 就可以了。例如：

```
[bigdata@bigdata02 ~]# zkCli.sh -server bigdata02:2181
```

进入命令行之后，键入help可以查看简易的命令帮助文档，如下图：



常用命令详解：

命令	作用
ls / ls /ZooKeeper	查看znode子节点列表
create /zk "myData"	创建znode节点
get /zk get /zk/node1	获取znode数据
set /zk "myData1"	设置znode数据
ls /zk watch	就对一个节点的**子节点变化事件**注册了监听
get /zk watch	就对一个节点的**数据内容变化事件**注册了监听
create -e /zk "myData"	创建临时znode节点
create -s /zk "myData"	创建顺序znode节点
create -e -s /zk "myData"	创建临时的顺序znode节点
delete /zk	只能删除没有子znode的znode
rmr /zk	不管里头有多少znode，统统删除
stat /zk	查看/zk节点的状态信息

znode数据信息字段解释：

```
cZxid = 0x400000093 节点创建的时候的zxid
# The zxid of the change that caused this znode to be created.

ctime = Fri Dec 02 16:41:50 PST 2016  节点创建的时间
# The time in milliseconds from epoch when this znode was created.

mZxid = 0x400000093  节点修改的时候的zxid，与子节点的修改无关
```

```
# The zxid of the change that last modified this znode.

mtime = Fri Dec 02 16:41:50 PST 2016  节点的修改的时间
# The time in milliseconds from epoch when this znode was last modified.

pZxid = 0x400000093  和子节点的创建/删除对应的zxid，和修改无关，和孙子节点无关
# The zxid of the change that last modified children of this znode.

cversion = 0  子节点的更新次数
# The number of changes to the children of this znode.

dataVersion = 0  节点数据的更新次数
# The number of changes to the data of this znode.

aclVersion = 0  节点（ACL）的更新次数
# The number of changes to the ACL of this znode.

ephemeralOwner = 0x0  如果该节点为ephemeral节点，ephemeralOwner值表示与该节点绑定的
session id. 如果该节点不是ephemeral节点，ephemeralOwner值为0
# The session id of the owner of this znode if the znode is an ephemeral node.
If it is not an ephemeral node, it will be zero.

dataLength = 6  节点数据的字节数
# The length of the data field of this znode.

numChildren = 0  子节点个数，不包含孙子节点
# The number of children of this znode.
```

kafka离开zookeeper就是个废物

hbase离开zookeeper也是个废物