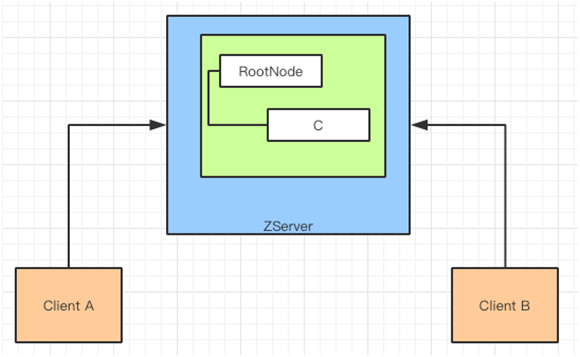
从一个简单的例子开始

在分布式系统中经常会遇到这种情况，多个应用读取同一个配置。例如：A、B两个应用都会读取配置C中的内容，一旦C中的内容出现变化，会通知A和B。

一般的做法是在A、B中按照时钟频率询问C的变化，或者使用观察者模式来监听C的变化，发现变化以后再更新A和B。那么ZooKeeper如何协调这种场景？

ZooKeeper会建立一个ZooKeeper服务器，暂且称为ZServer，用它来存放C的值。为A、B两个应用分别生成两个客户端，称作ClentA和ClientB。

这两个客户端连接到ZooKeeper的服务器，并获取其中存放的C。保存C值的地方在ZooKeeper服务端（Server）中称为ZNode。



ZNode

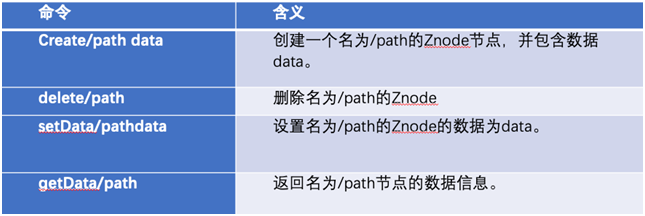
通过上面的例子，客户端ClientA和ClientB需要读取C的内容。这个C就作为树的叶子节点存放在ZooKeeper的ZNode中。

通常来说，为了提高效率ZNode是被存放在内存中的。ZNode的数据模型是一棵树（ZNode Tree）。

好像我们从上图中看到的一样，树中的每个节点都可以存放数据，并且每个节点下面都可以存放叶子节点。

ZooKeeper客户端通过”/”作为访问路径访问数据。例如，可以通过路径”/RootNode/C”来访问C变量。

为了方便客户端调用，ZooKeeper会暴露一些命令：



作为存储媒介来说，ZNode分为持久节点和临时节点：

* 持久节点（PERSISTENT）

该数据节点被创建后，就一直存在于ZooKeeper服务器上，除非删除操作（delete）清除该节点。

* 临时节点（EPHEMERAL）

该数据节点的生命周期会和客户端（Client）会话（Session）绑定在一起。如果客户端（Client）会话丢失了，那么节点就自动清除掉。

如果把临时节点看成资源的话，当客户端和服务端产生会话并生成临时节点，一旦客户端与服务器中断联系，节点资源会被从ZNode中删除。

* 顺序节点（SEQUENTIAL）

ZNode节点被分配为一个单调递增的整数。例如多个客户端在服务器/tasks上申请节点时，根据客户端申请的先后顺序，将数字追加到/tasks/task后面。

如果有三个客户端申请节点资源，那么在/tasks下面建立三个顺序节点，分别是/tasks/task1、/tasks/task2、/tasks/task3。

顺序节点，在处理分布式事务的时候非常有帮助，当多个客户端（Client）协作工作的时候，会按照一定的顺序执行。

如果将上面的两类节点和顺序节点进行组合的话，就有四种节点类型，分别是持久节点、持久顺序节点、临时节点、临时顺序节点。

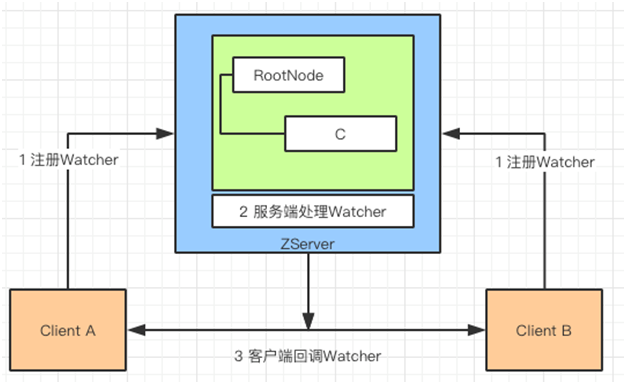
Watcher

上面说了ZooKeeper用来存放数据的ZNode，并且把C的值存储在里面。如果C被更新了，两个客户端（ClientA、ClientB）如何获得通知呢？

ZooKeeper客户端（Client）会在指定的节点（/RootNode/C）上注册一个Watcher，ZNode上的C被更新的时候，服务端就会通知ClientA和ClientB。

通过三步来实现：

1. 客户端注册Watcher
2. 服务端处理Watcher
3. 客户端回调Watcher



客户端注册Watcher

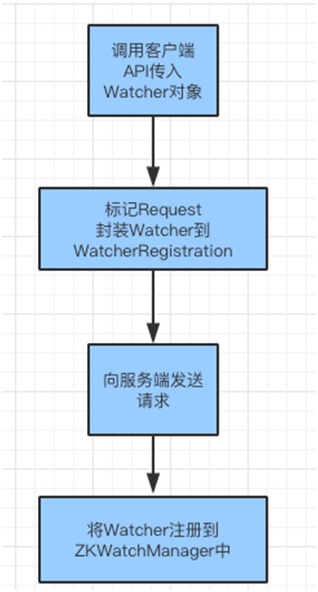
ZooKeeper客户端创建Watcher的实例对象：

D:\data\app\WeChat\WeChat Files\xiaohuxian292\FileStorage\Temp\f2f78cac4568b8dc4d8d98ada3398996.png

同时这个Watcher会保存在客户端本地，一直作为和服务端会话的Watcher。

D:\data\app\WeChat\WeChat Files\xiaohuxian292\FileStorage\Temp\0a02a13b0990141032d2e200d21f1e81.png

客户端可以通过getData、getChildren和exist方法来向服务端注册Watcher。



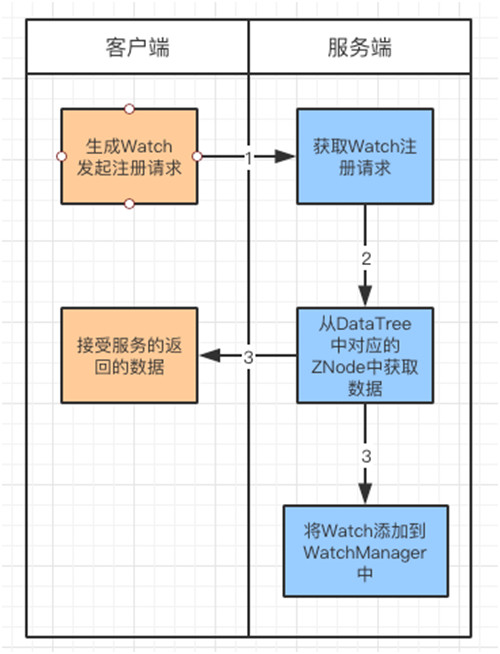
同时需要注意的是在客户端发送Watcher到服务端注册的时候，会将这个要发送的Watcher在本地的ZKWatchManager中保存。

这样做的好处，就是当获得服务端的注册成功的信息以后，就不用将Watcher的具体内容回传给客户端了。

客户端只用在接到服务端响应以后，从本地的ZKWatchManager中获取Watch的信息进行处理即可。

服务端处理Watcher

服务端收到客户端的请求以后，交给FinalRequestProcessor处理，这个进程会去ZNode中获取对应的数据，同时会把Watch加入到WatchManager中。这样下次这节点上的数据被更改了以后，就会通知注册Watch的客户端了。



客户端回调Watcher

客户端在响应客户端Watcher注册以后，会发送WatcherEvent事件。作为客户端有对应的回调函数接受这个消息。

这里会通过readResponse方法统一处理：



在SendTread接收到服务端的通知以后，会将事件通过EventThread.queueEvent发送给EventThread。

正如前面提到的，在客户端注册时，已经将Watcher的具体内容保存在ZKWatchManager一样了。

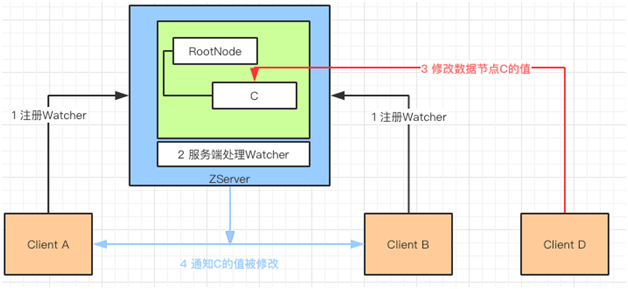
所以，EventTread通过EventType就可以知道哪个Watcher被响应了（数据变化了）。然后从ZKWatchManager取出具体Watch放到waitingEvent队列等待处理。

最后，由EventThrea中的processEvent方法依次处理数据更新的响应。

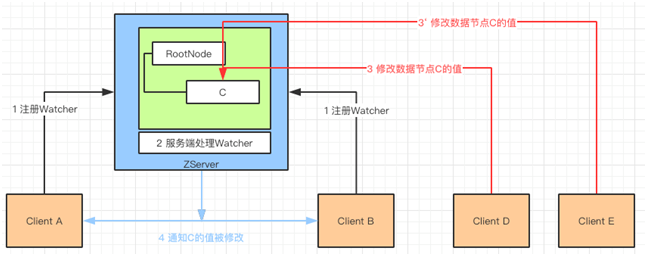


版本（Version）

介绍完了Watcher机制，回头再来谈谈ZNode的版本（version）。如果有一个客户端（ClientD），它尝试修改C的值，此时其他两个客户端会收到通知，并且进行后续的业务处理了。



那么在分布式系统中，会出现这么一种情况：在ClientD对C进行写入操作的时候，又有一个ClientE也对C进行写入操作。这两个Client会去竞争C资源，通常这种情况需要对C进行加锁操作。



因此引入ZNode版本（version）概念。版本是用来保证分布式数据原子性操作的。

ZNode的版本（version）信息保存在ZNode的Stat对象中，有如下三种：



本例只关注“数据节点内容的版本号”，也就是version。

如果说ClientD和ClientE对C进行写入操作视作是一个事务的话。在执行写入操作之前，两个事务分别会获取节点上的值，即节点保存的数据和节点的版本号（version）。

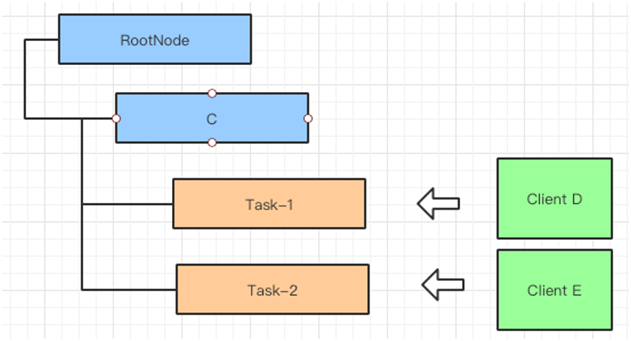
以乐观锁为例，对数据的写入会分成以下三个阶段：数据读取、写入校验和数据写入。例如C上的数据是1，version是0。

此时ClientD和ClientE都获取了这些信息。假设ClientD先做写入操作，在做写入校验的时候，发现之前获得的version和节点上的version是相同的，都是1，因此直接执行数据写入。

写入以后，version由原来的1变成了2。当ClientE做写入校验时，发现自己持有的version=1和节点当前的version=2不一样，于是，写入失败，重新获取version和节点数据，再次尝试写入。

除了上述方案以外，还可以利用ZNode的有序性，在C下面建立多个有序的子节点。每当一个Client准备写入数据的时候，创建一个临时有序的节点。

节点的顺序根据FIFO算法，保证先申请写入的Client排在其前面。每个节点都有一个序号，后面申请的节点按照序号依次递增。



每个Client在执行修改C操作的时候，都要检查有没有比自己序号小的节点，如果存在那么就进入等待，直到比自己序号小的节点进行完毕以后，才轮到自己执行修改操作。从而保证了事务处理的顺序性。

会话（Session）

说完版本（version）的概念，例子从原来的ClientAB已经扩充到了ClientDE。这些客户端都会和ZooKeeper的服务端进行通信，或读取数据、或修改数据。

我们将客户端与服务端完成的这种连接称为会话。ZooKeeper的会话有Connecting、Connected、Reconnecting、Reconnected和Close这几种状态。并且在服务端有专门的进程来管理它们，客户端初始化的时候就会根据配置自动连接服务器，从而建立会话，客户端连接服务器时会话处于Connecting状态。一旦连接完成，就会进入Connected状态。如果出现延迟或者短暂失联，客户端会自动重连，Reconnecting和Reconnected状态也就应运而生。如果长时间超时，或者客户端断开服务器，ZooKeeper会清理掉会话、以及该会话创建的临时数据节点，并且关闭和客户端的连接。

session作为会话实体，用来代表客户端会话，其包括4个属性：

* SessionID

用来全局唯一识别会话。

* TimeOut

会话超时事件。客户端在创造Session实例的时候，会设置一个会话超时的时间。

* TickTime

下次会话超时时间点。后面“分桶策略”会用到。

* isClosing

当服务端如果检测到会话超时失效了，会通过设置这个属性将会话关闭。

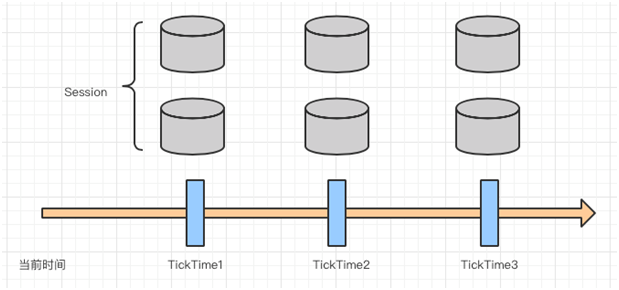
既然，会话是客户端与服务器之间的连接。在服务器端由SessionTracker管理会话。SessionTracker有一个工作就是，将超时的会话清除掉。于是“分桶策略”就登场了。

由于每个会话在生成的时候都会定义超时时间，通过当前时间+超时时间可以算出会话的过期时间。

由于SessionTracker不是实时监听会话超时，它是按照一定时间周期来监听的。也就是说，如果没有到达SessionTracker的检查时间周期，即使有会话过期，SessionTracker也不会去清除。由此，就引入会话超时计算公式，也就是TickTime的计算公式。

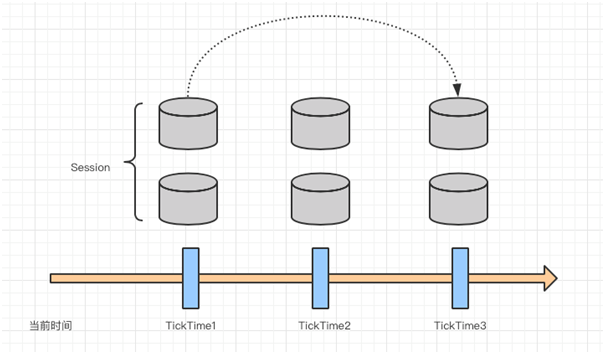
TickTime=( ( 当前时间 + 会话过期时间) /检查时间间隔 + 1 ) \*检查时间间隔

将这个值计算出来以后，SessionTracker会把对应的会话按照这个时间放在对应的时间轴上面。SessionTracker在对应的TickTime检查会话是否过期。



每当客户端连接上服务器都会做激活操作，同时每隔一段时间客户端会向服务器发送心跳检测。

服务器收到激活或者心跳检测以后，会重新计算会话过期时间，根据“分桶策略”进行重新调整。把会话从“老的区块”放到“新的区块”中去。



对于超时的会话，SessionTracker也会做如下清理工作：

* 标记会话状态为“已关闭”，也就是设置isClosing为True。
* 发起“会话关闭”的请求，让关闭操作在整个集群生效。
* 收集需要清理的临时节点。
* 添加“节点删除”的事务变更。
* 删除临时节点
* 移除会话
* 关闭客户端与服务端的连接

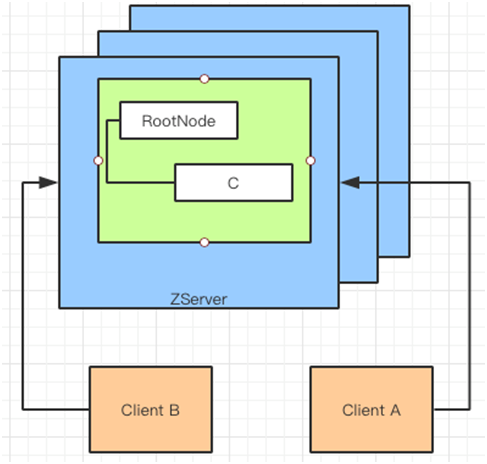
会话关闭以后客户端就无法从服务端获取/写入数据了。

服务群组（Leader，Follower，Observer）

前面提到了客户端如何通过会话与服务端保持联系，以及服务端是如何管理客户端会话（session）的。

我们继续思考一下，这么多的服务端都依赖一个ZooKeeper服务器。一旦服务挂了，客户端就无法工作了。

为了提高ZooKeeper服务的可靠性，引入服务器集群的概念。从原来的单个服务器，扩充成多个服务器，即使某一台服务器挂了，其他的服务器也可以顶上来。



这样看起来不错了，新的问题是，存在多个ZooKeeper服务器，那么客户端的请求发给哪台呢？服务器之间如何同步数据呢？如果一个服务挂掉了，其他的服务器如何替代？这里介绍两个概念Leader和Follower。

Leader服务器，是事务请求（写操作）的唯一调度者和处理者，保证集群事务处理的顺序性。也是集群内部服务器的调度者。

它是整个集群的老大，其他的服务器接到事务请求都会转交给它，让它协调处理。

Follower服务器，处理非事务请求（读操作），转发事务请求给Leader服务器。参与选举Leader的投票和事务请求Proposal的投票。

既然Leader是集群的老大，那么这个老大是如何产生的。ZooKeeper有仲裁机制，通过服务器的选举产生这个Leader，按照少数服从多数的原则。

因此，集群中服务器的个数一般都是奇数，例如：1、3、5，当然这里是建议。关于选举和仲裁都有一定的看法，一起来看看吧。

当众多服务器启动的时候，互相都不知道谁是Leader，因此都会进入Looking状态，也就是在网络中寻找Leader。

寻找的过程也是投票的过程，每个服务器会将服务器ID和事务ID作为投票信息发送给网络中其他的服务器，假设称它为投票信息VOTE，它包括：（ServerID, ZXID）。其中，ServerID是服务器注册的ID，随着服务器启动的顺序自动增加，后启动的服务器ServerID就大；ZXID是服务器处理事物的ID，随着事物的增加自动增加，同样后提交的事物ZXID也大一些。

其他的服务器收到VOTE信息以后会和自己的VOTE信息（ServerID，ZXID）进行比较。如果收到的VOTE（ServerID， ZXID）中的ZXID比自己的ZXID要大，那么把自己的VOTE修改成收到的VOTE。

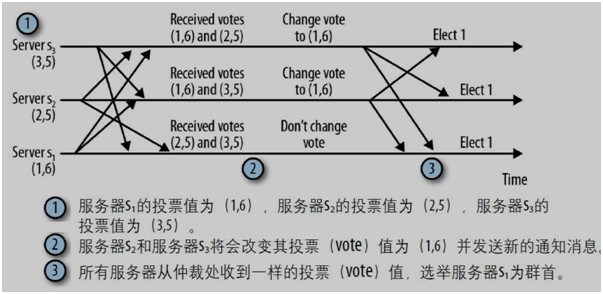
如果ZXID一样大，那么就比较ServerID，将大的那个ServerID作为自己VOTE的ServerID，转发给其他服务器。

再简单点说，如果事物ID（ZXID）比自己的事物ID（ZXID）要大，就把票投给这个服务器。如果事务ID一样，就把票投给ServerID大的服务器。

来个具体的例子，有三个服务器，他们的投票值分别是：

* S1( 1, 6 )
* S2( 2, 5 )
* S3( 3, 5 )

三个服务器分别把自己的VOTE发给其他两台服务器，S2和S3收到VOTE以后发现ZXID为6的来自S1的VOTE比自己持有的ZXID要大，因此把自己的VOTE修改为（1, 6）投出去，因此S1称为Leader。



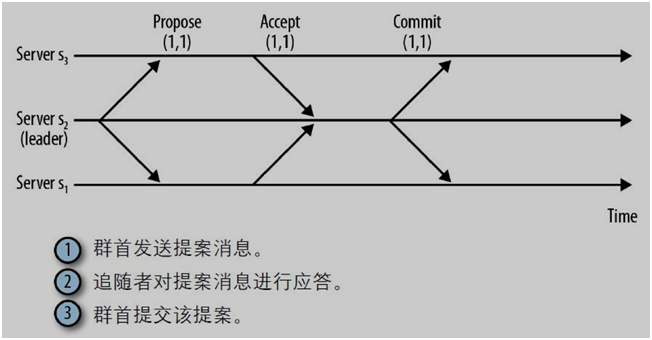
同样，如果S1作为Leader，因为某种原因挂掉或者长时间没有响应请求，其他的服务器也会进入Looking状态，开启投票仲裁模式寻找下一个Leader。

成为新Leader以后会通过广播的方式将ZNode上的数据同步到其他的Follower。

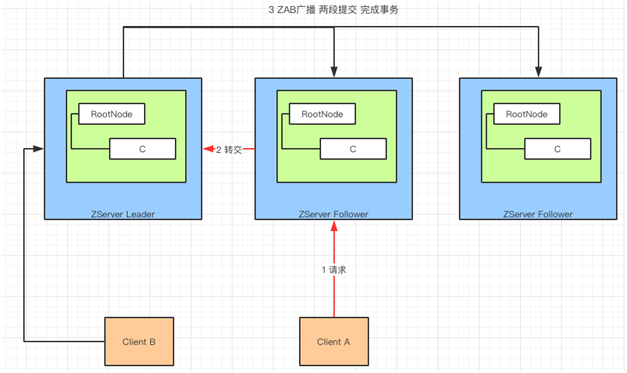
Leader有了，整个服务器集群有了领袖，它可以处理客户端的事务请求。客户端的请求可以发给集群中任意一台服务器，无论是哪个服务器都会将事务请求转交给Leader。Leader在将数据写入ZNode之前会向ZooKeeper的其它Follower进行广播。这里广播用到了ZAB协议（Atomic Broadcast Protocol）是Paxos协议的实践。说白了就是一个两段提交（两段提交和三段提交是分布式事务相关的概念）。

这里ZooKeeper通过以下方式实践两段提交：

* Leader向所有Follower发送一个PROPOSAL。
* 当Follower接收到PROPOSAL后，返回给Leader一个ACK消息，表示我收到PROPOSAL，并且准备好了。
* Leader仲裁数量（过半数）的Follower发送的ACK后（包括Leader自己），会发送消息通知Follower进行commit。
* 收到commit后，Follower就开始干活，将数据写入到ZNode中。



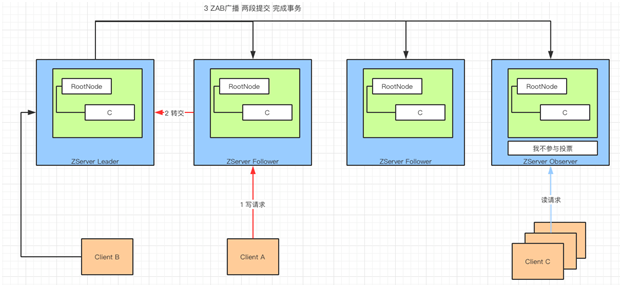
选举了Leader领导集群，Leader接受到Client的请求以后，也可以协调Follower工作了。



那么如果Client很多的情况下，特别是这些客户端都是做读操作的时候，ZooKeeper服务器如何处理如此多的请求呢？这里引入Observer的概念。

Observer和Follower基本一致，对于非事务请求（读操作），可以直接返回节点中的信息（数据从Leader中同步过来的）。对于事务请求（写操作），会转交给Leader做统一处理。Observer的存在就是为了解决大量客户端读请求。

Observer和Follower的区别是，Observer不参与仲裁投票和选举Leader。



总结

全文用了一个简单的例子讲ZooKeeper的主要特性和实现原理，最后做个总结。

ZooKeeper被用来协调和管理分布式系统，发挥着重要的作用。分布式系统由于其特性，应用分布在不同的物理主机或者网络中。

为了让它们协同工作，ZooKeeper中的ZNode成为统一协调的重要部分，客户端通过Client间接到服务端的ZNode上，监听ZNode数据的变化。同时ZNode支持的持久、临时和顺序性，以及版本控制，这些特性支持了分布式事务和锁的功能。

如果说，每一个ZooKeeperClient对Server的写入操作都是一次事务的话，ZooKeeper服务端维护了大量的事务，并且通过“分桶策略”来管理它们，保证了Client与Server端协调工作。

为了提高Server的可靠性，ZooKeeper引入了Server集群的概念。通过仲裁机制选举Leader来领导其他Follower。

事务都由Leader来处理，通过两段提交的方式对其他Server发起广播。为了增强对非事务请求的处理效率，ZooKeeper加入了Observer来帮忙。