从几个方面来了解一个框架的（要看的点实在太多要不使用从重要到不重要）

## 原理方面

1. 框架使用何种语言，设计模式是什么，核心是什么，数据结构有哪些，核心之上搭建了哪些功能。

#### 实现：Java语言实现

#### 设计模式：Watcher 的观察者模式

#### 核心：znode数据结构

Note：类似于数的结构，每个节点都可以存放不大于1m的数据，一般是放置一些Note ：项目的公有数据。

节点分为：

永久节点，永久有序节点，临时节点，临时有序节点

永久节点：除非手动删除否则一直存在

永久有序节点：在永久节点上加zk自己维护的致增整数

临时节点：客户端链接存在才会存在的节点

临时有序节点：临时节点上加zk自己维护的致增整数

*Note：　Zookeeper 的核心是广播，这个机制保证了各个Server之间的同步。实现这个机制的协议叫做Zab协议。  
　　Zab协议有两种模式，它们分别是恢复模式（选主）和广播 模式（同步）。当服务启动或者在领导者崩溃后，Zab就进入了恢复模式，当领导者被选举出来，且大多数Server完成了和leader的状态同步以后， 恢复模式就结束了。  
　　状态同步保证了leader和Server具有相同的系统状态。为了保证事务的顺序一致性，zookeeper采用了递增的事务id号 （zxid）来标识事务。  
　　所有的提议（proposal）都在被提出的时候加上了zxid。实现中zxid是一个64位的数字，它高32位是epoch用 来标识leader关系是否改变，每次一个leader被选出来，它都会有一个新的epoch，标识当前属于那个leader的统治时期。低32位用于递 增计数。  
　　每个Server在工作过程中有三种状态：  
　　LOOKING：当前Server不知道leader是谁，正在搜寻。  
　　LEADING：当前Server即为选举出来的leader。  
　　FOLLOWING：leader已经选举出来，当前Server与之同步。*

#### 数据结构：树

#### Zookeeper 下 Server工作状态

服务器具有四种状态，分别是LOOKING、FOLLOWING、LEADING、OBSERVING。

* LOOKING：寻找Leader状态。当服务器处于该状态时，它会认为当前集群中没有Leader，因此需要进入Leader选举状态。
* FOLLOWING：跟随者状态。表明当前服务器角色是Follower。
* LEADING：领导者状态。表明当前服务器角色是Leader。
* OBSERVING：观察者状态。表明当前服务器角色是Observer。

#### 什么是 ZAB协议

包含了崩溃恢复协议和数据广播协议

刚启动或者leader服务挂了进入崩溃恢复协议选举产生leader，leader产生后退出恢复模式对外进行数据广播，提供服务。

## 内部流程/原理

##### 写数据/如何保证数据一致性：

客户端把写数据发到follower或者observer/leader,会把数据发送到leader leader对操作分配zid等数据，分发到各个节点当有半数Follower成功表示成功写入，zk leadr保证事务的顺序以及原子性。

*Note：这里有个点就是follower很多会占用带宽，写性能下降，所以产生observer模式。*

##### Watcher机制

1. 一次性，只触发一次
2. 客户端串行执行，指的是可能要回调多个客户端不要在一个客户端卡住。
3. 轻量只通知路径以及变化不传具体的数据
4. 注册watcher getData、exists、getChildren
5. 触发watcher create、delete、setData
6. 如果客户端exits一个未创建连接断开回来之前创建又删除则丢失通知，断开无法收到通知但是如果需要的话可以设置收到所有通知。

Note：为什么不是永久，因为这样如果频繁变动会产生很大压力且客户端不一定每次都需要知道数据变更

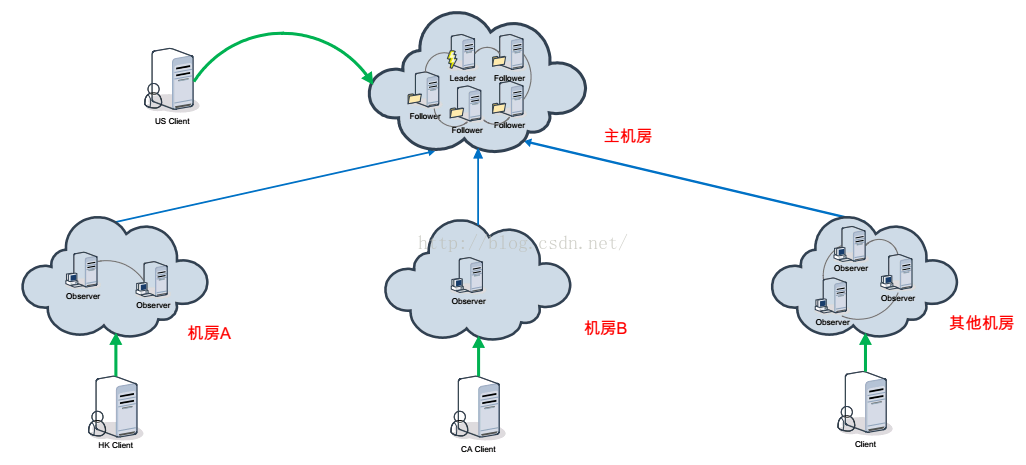
##### Chroot特性

可以设置命名空间，客户端所有节点操作都会限制在这个目录下。

##### observer模式：

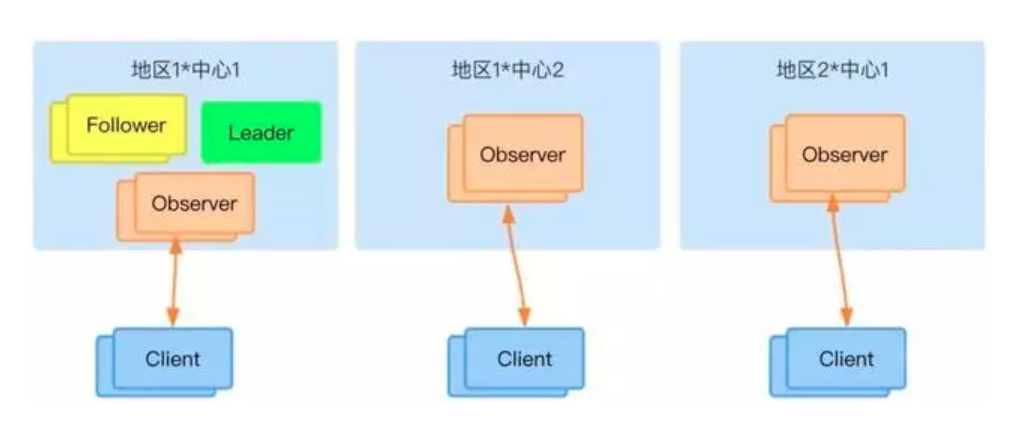
由于写follower多耗费性能所以产生 observer节点，只接受请求，不参与投票，写数据也不用返回。所以即使断开也不会影响集群，写的压力减小。

部署的话可以本地机房部署observer ，增加读取性能。



Note：observer，同步数据中心更新也是受限于网络速度，至少是增加增加读取性能。

容灾部署方案（上面的具体版本）：



Note1：leader和follower部署在同一个地区防止 出现网络故障产生大量选举不可用

Note2：也可以再加两个 follower实现leader follower 的 3 2 2机房实现完全容灾。

##### 使用Observer

可以在对应节点的配置文件添加如下配置：peerType=observer

上面仅仅是告诉Zookeeper该节点是Observer，其次，必须在配置文件指定哪些节点被指定为Observer，例如：server.1:localhost:2181:3181:observer

##### 静态扩容、动态扩容（做好幂等性）

Note：在极端情况下静态扩容可能会导致zookeeper集群出现严重的数据不一致问题，比如现有集群：A、B、C，现在需要进行静态扩容，停止ABC实例，拉入DE实例，此时如果C实例是ABC中最滞后的实例，如果AB启动的速度没有C快就会导致CDE组成新的集群，新的纪元号会覆盖原来的AB日志。当然现在基本上不会接受静态扩容，基本上都是动态扩容。

动态扩容在极端情况下也会出现类似问题，比如现在有三个机房，1、2、3，1机房方leader zid=200、100，2机房zid=80、50，3机房zid=40，假设上次的commit是在zid=200、100、50之间提交的，此时机房1出现断网，2机房zid=80、50与3机房zid=40开始组成新的集群，新的纪元在zid=50上产生。

Note2：解释-静态扩容导致 zid低的启动快，与其它zid低的组成集群导致zid丢失。

动态扩容，高zid的机器机房刚好断网，zid在低的纪元重新开始。解决做好幂等性以及follower多机房部署。

Note：如何实现幂等性？

##### 数据同步

在准leader、产生后发生，leader发送NEWLEADER数据包给每个follower并为每个follower创建一个线程进行处理，follower收到包返回，当前的zid，leader根据zid跟自己的比较做如下处理。

如果一样告知已同步。

如果这阶段有新的zid产生则同步差异过去，一个个发送commit包过去要求followe记录下来。

如果zid比自己大发送回滚指定删除多余数据。

如果这阶段没有新zid这发送uptodate数据包，等待follower的ack完成同步。

*Note：zookeeper节点间的数据同步*

*当各个节点已经自我恢复并选举出leader后，leader就开始和follows进行数据同步了，具体的逻辑可以见{org.apache.zookeeper.server.quorum.LearnerHandler}中：*

*leader构建NEWLEADER包，内含leader最大数据的zxid, 广播给follows，然后leader根据follower数量为每个follower创建一个LearnerHandler线程来处理同步请求：leader主线程阻塞，等待超过半数follower同步完数据之后成为正式leader。  
follower接收到NEWLEADER包后响应FOLLOWERINFO给leader，告知本方数据最大的zxid值； leader接收到回馈后开始判断：*

1. *如果follower和leader数据一致，则直接发送DIFF告知已经同步；*
2. *判断这一阶段内有无已经被提交的决议值，如果有，那么  
   a) 如果有部分数据没有同步，leader发送DIFF包将有差异的数据同步过去，同时将follower没有的数据逐个发送commit包给follower要求记录下来；  
   b) 如果follower数据zxid更大，发送TRUNC包给follower要求删除多余数据*
3. *如果这一阶段没有提交的决议，直接发送SNAP包将快照同步给follower*

*以上消息完毕后，LEADER发送UPTODATE包告知follower当前数据已同步，等待follower的ACK完成同步过程。*

#### 主要功能：

##### 发布/订阅：

利用节点的通知机制，当节点修改时候向订阅者发送通知，就好像发布/订阅模式。

##### 负载均衡：

mateQ 每个Broker都把自己注册到zk，发送消息采用轮训方式选择Broker，消费者也是注册到zk，在消费者端有对消费者进行分组，一个消息只会被一个group消费，当消费者或者生产者挂了可以收到收到通知进行移除重新计算负载。

##### 命名服务：

把服务注册到某个节点，多个服务实例增量添加，下面，调用方从节点获取ip port信息，zk负责服务的提供方健康检查，调用方可以采用轮训做负载均衡。

##### 分布式协调/通知：

当有多个任务需要协调多个任务进行处理，比如复制数据库，分发多个任务到处理器，失败触发机制被用机器，并报告自己的进度。

##### 集群管理：

健康检查，每个机器都可以同步自己状态到zk。

##### Master选举：

###### 初始化选举：zk的节点一般为奇数个，zk1，zk2，zk3

其实zk1启动处于looking状态，自己无法投票，当zk2启动也处于looking状态，两个机器互相通信开始投票。

1. 发出投票

Zk1，zk2都把自己当成leader进行选举，每张票包含（mid，zid），zk1，zk2给各个服务器发送自己的投票 k1（1，0），zk2（2，0）。

1. 接受投票

集群中的每个服务器接收到投票后，首先判断该投票的有效性，如检查是否是本轮投票、是否来自Looking状态的服务器等。

1. 处理投票

和自己的票进行比较

* 优先选择zid大的
* Zid一样选择mid大的

1. 统计投票

统计集群是否有半数接受了相同的投票信息，如果有投票结束。

本次 zk2（2，0）被两个机器接受所以zk2leader。

1. 更新状态

然后zk2发现自己是leader更新自己状态为leadering zk1发现自己是follower则更新自己为following状态。当zk3加入进来发现已经有leader则更新自己为following。

###### 运行期间产生选举：

当运行期间leader节点挂掉了，zk将对外暂停服务为。以上面的zk1，zk2，zk3为例。Zk2挂掉了

1.变更状态

所有的非observer节点变更为looking，进入leader选举流程。

*Note：这里如果不是使用observer异地部署参加选举将很慢*

2.发出投票

此时每个服务器上的zid可能不同，假设zk1的zid是 123，zk3的zid是 124，

发送自己票给所有服务器。

3.接受投票 （同上）

4.处理投票 （同上）

5.统计投票 （同上）

6.更新状态 （同上）

##### 分布式锁（写）

Note：所有连接一起件事某个节点，连接在下面创建临时有序子节点，所有连接检测自己是不是最小的临时节点，如果是，表示获取到锁，则处理逻辑，其它节点进来则发现自己不是最小向前一个一个注册watch，最小处理完成断开连接或者删除后。Watch机制使得其他节点收到通知得到锁。

##### 分布式锁（读）

也是创建临时有序节点（读取加个R标识eg： adfsad-R0001,写：adfas-W0002），对于读取判断

自己最小直接读取

前面都是读锁也读取

对于写请求依然同上需要等待

##### 分布式队列

在某个节点下面创建有序队列，数据依次插到节点，当处理数据的时候从中取出并删除节点

##### Note：在分布式系统领域有个著名的 CAP定理(C- 数据一致性;A-服务可用性;P-服务对网络分区故障的容错性，这三个特性在任何分布式系统中不能同时满足，最多同时满足两个);ZooKeeper是个

##### 几个随便命令：get set getchilder exits delete create

##### ZAB和Paxos算法的联系与区别？

* 相同点：
  + 两者都存在一个类似于Leader进程的角色，由其负责协调多个Follower进程的运行
  + Leader进程都会等待超过半数的Follower做出正确的反馈后，才会将一个提案进行提交
  + ZAB协议中，每个Proposal中都包含一个 epoch 值来代表当前的Leader周期，Paxos中名字为Ballot
* 不同点：  
  ZAB用来构建高可用的分布式数据主备系统（Zookeeper），Paxos是用来构建分布式一致性状态机系统。

##### 有哪些客户端

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

这里说的负载均衡是指软负载均衡。在分布式环境中，为了保证高可用性，通常同一个应用或同一个服务的提供方都会部署多份，达到对等服务。而消费者就须要在这些对等的服务器中选择一个来执行相关的业务逻辑，其中比较典型的是消息中间件中的生产者，消费者负载均衡。

消息中间件中发布者和订阅者的负载均衡，linkedin开源的KafkaMQ和阿里开源的metaq都是通过zookeeper来做到生产者、消费者的负载均衡。这里以metaq为例如讲下：

生产者负载均衡：metaq发送消息的时候，生产者在发送消息的时候必须选择一台broker上的一个分区来发送消息，因此metaq在运行过程中，会把所有broker和对应的分区信息全部注册到ZK指定节点上，默认的策略是一个依次轮询的过程，生产者在通过ZK获取分区列表之后，会按照brokerId和partition的顺序排列组织成一个有序的分区列表，发送的时候按照从头到尾循环往复的方式选择一个分区来发送消息。

消费负载均衡： 在消费过程中，一个消费者会消费一个或多个分区中的消息，但是一个分区只会由一个消费者来消费。MetaQ的消费策略是：

1. 每个分区针对同一个group只挂载一个消费者。

2. 如果同一个group的消费者数目大于分区数目，则多出来的消费者将不参与消费。

3. 如果同一个group的消费者数目小于分区数目，则有部分消费者需要额外承担消费任务。

在某个消费者故障或者重启等情况下，其他消费者会感知到这一变化（通过 zookeeper watch消费者列表）