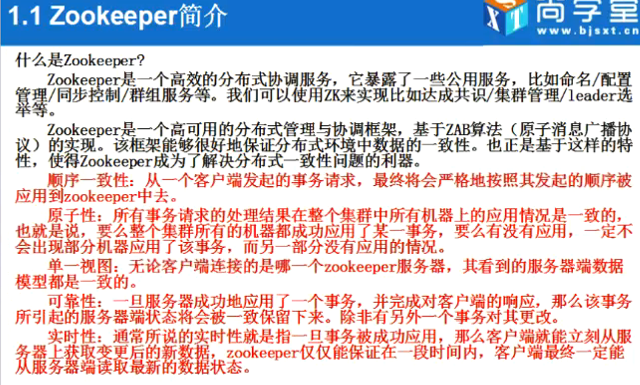
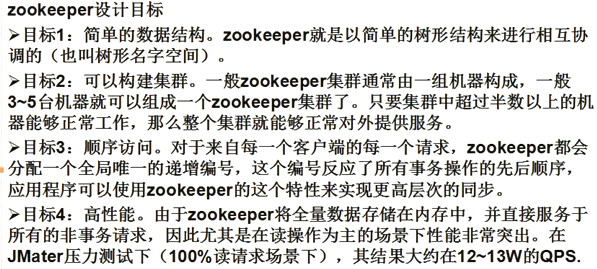
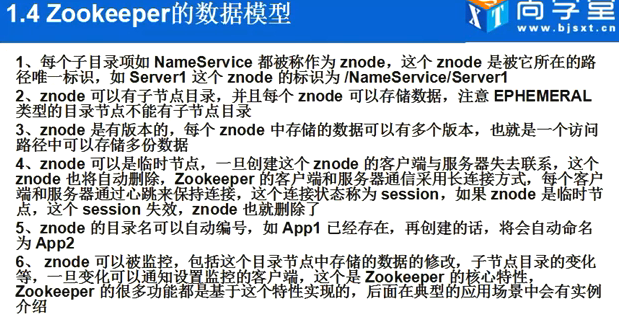
## Zookeeper简介

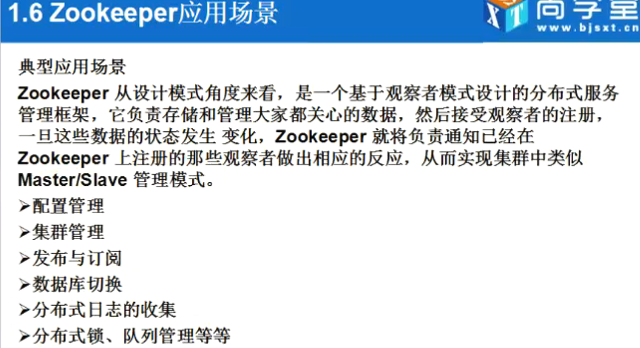


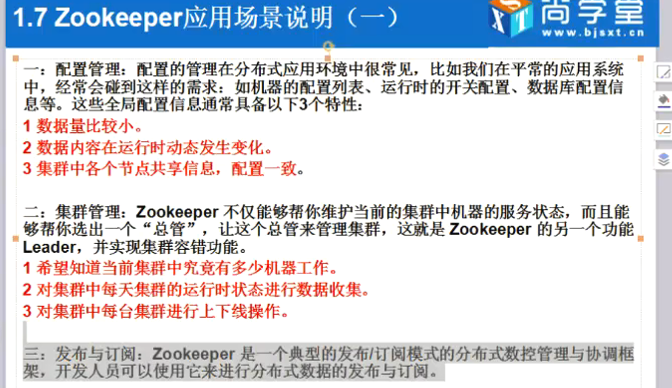


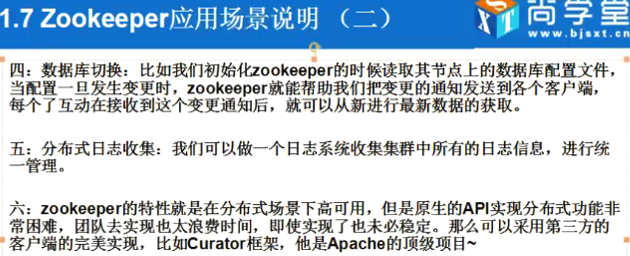




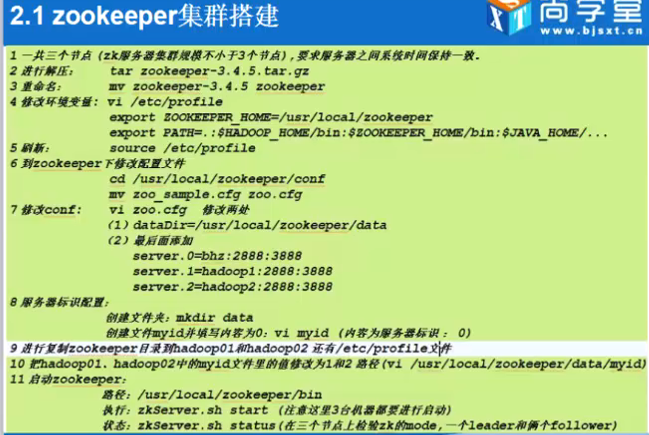






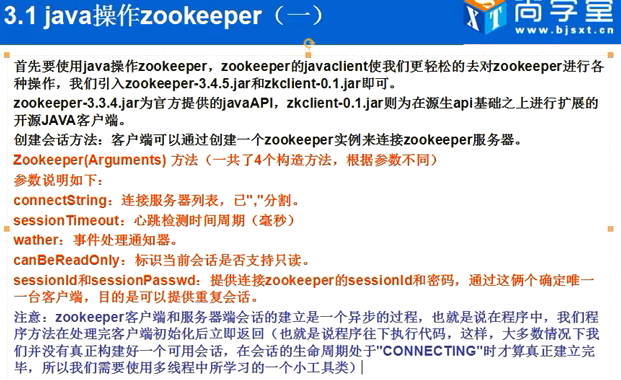


## Zookeeper搭建





## Java操作zookeeper





持久节点：当节点创建后，就一直存在，直至有删除操作来主动删除这个节点。该节点不会因创建该节点的客户端会话失效而消失。

持久顺序节点：和持久节点类似。额外的特性是，每个父节点会为它的第一级子节点维护一份时序，会记录每个子节点创建的先后顺序。在创建节点时，zk会自动为给定节点名后加一个单调递增的数字后缀，作为新的节点名。

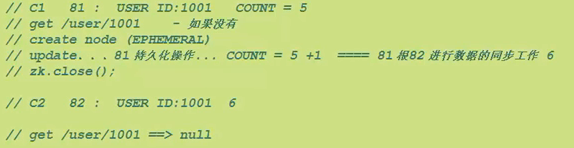
临时节点：生命周期和客户端的会话绑定。如果客户端会话失效，那么这个节点就会被删除。这里的会话失效而不是连接断开。在临时节点下面不能创建子节点。

临时顺序节点：可以用来实现分布式锁。

分布式锁实现：

（这里要求父节点必须存在）客户端调用create方法创建名为/super/guid\_lock\_ 的节点，这里的节点类型需要为EPHEMERAL\_SEQUECTIAL,客户端调用getChildren(“/super”)方法获取已经创建的子节点，注意这里不需要注册任何watcher（避免羊群效应，如果同一时刻有大量客户端断开连接，则会触发很多通知给其他客户端），客户端获取到所有子节点的path之后，进行比较，如果发现自己是所有节点序号最小的，那么就认为该客户端获取了锁。如果不是最小的，则说明自己没有获得锁，此时客户端需要找到比自己小的那个节点，然后对其调用exist方法，同时注册监听事件。之后当这个被关注的节点被移除了，客户端就会收到相应的通知，这个时候客户端需要再次调用getChildren方法来获取已经创建的子节点，确保自己是最小的节点。

zookeeper分布式锁的大致原理就是利用临时节点。比如，两个服务器s1和s2，运行相同的web应用，采用nginx作为前端负载均衡器，当某个请求是对数据库中进行操作时，s1会向zookeeper查看是否存在节点id=1的临时节点，如果不存在则创建，这样就保证了在同一时刻只会有一个请求对数据库进行操作。对于zookeeper来说，不管并发如何，它能够保证在创建节点时只能有一个客户端进行操作。

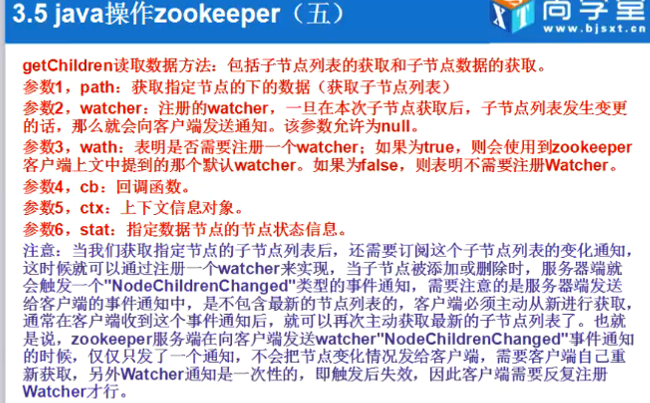


Zookeeper的get效率非常高，一般来说都是先get查看节点存在不存在，如果不存在再去create，极少情况下会出现两个client去get时发现节点再同时create的情况。





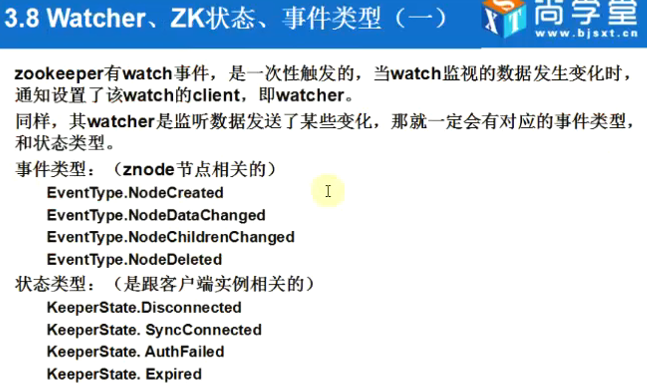




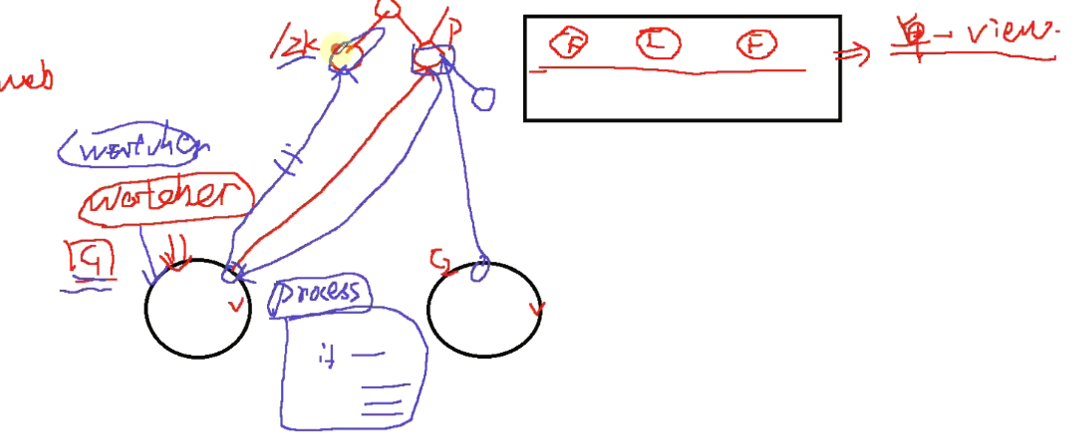




## Watcher ZK状态 事件类型

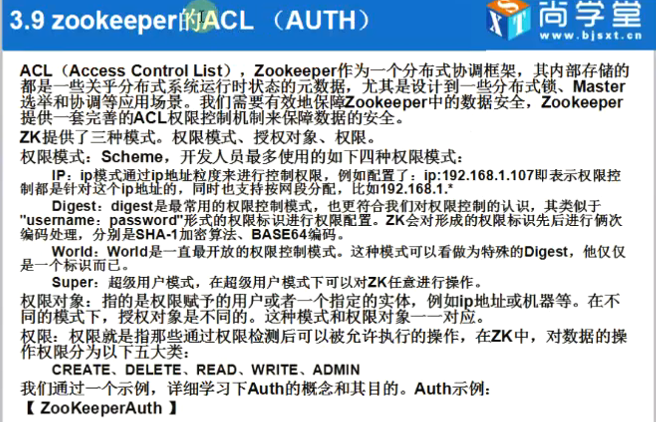


这里始终监控的是父节点，当父节点发生变化或者子节点新增删除时，都会触发该父节点的事件。



客户端可以监控zk中节点的变化，这样在客户端就可以存在一个或者多个watcher（客户端单独开一个线程用于watch）。这样当zk节点发生变化时，client就能收到事件通知，然后在process中进行处理，需要在process中判断事件的类型。（watch是一次性的，如果需要多次watch，两种方法：1 在一次触发之后，手动修改boolean值为true 2 重新new出一个watcher



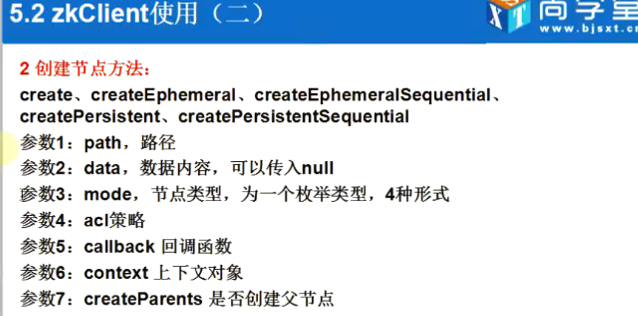






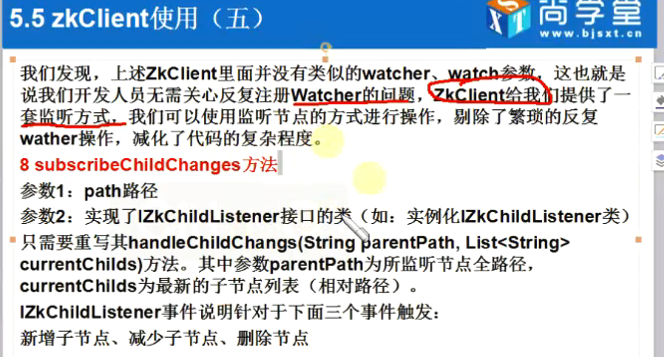
## Zkclient

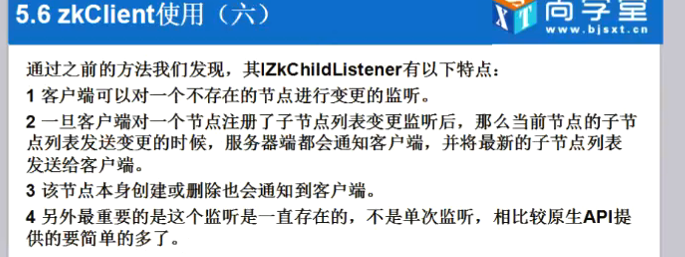












## Curator





### 连接状态

1 当客户端和zk服务端成功建立连接之后，zk会在和客户端之间维持一个session，默认的超时时间是60s。

2 短暂的失去连接：由于网络原因，客户端和zk服务端失去连接，如果在session超时时间之内，客户端重新连接上了服务端，由于session没有过期，zk能够保证连接恢复后保持正常服务。

3 客户端重新连接上zk服务端时间比较长，已经超过了session的超时时间，此时zk服务端会把该客户端相关的临时节点、watcher等都删除。因此客户端再次连上zk服务端上时，会获取到session expired异常，curator会销毁之前的session，创建新的session，但是之前相关的临时节点和watcher都不会存在，需要开发者监听连接状态事件，当session过期后，重新进行创建临时节点。这里需要对Lost事件进行监听。

4 客户端重新启动：不论zk的session是否过期，都需要重新创建临时节点和watcher事件，先前的session会在稍后一段时间内过期。

5 zk服务端重新启动：zk将session信息存放到硬盘上，因此重启之后，先前未过期的session依然存在，在zk服务器启动之后，客户端与zk服务端创建新的连接，使用先前的session。

6 当session过期了，在session过期期间另外的客户端修改了zk的值，那么这个修改在客户端重新连接到zk上时，zk客户端不会接收到这个修改的watch事件（即便添加了watch），如果需要严格的watch逻辑，需要在curator的状态监控中添加逻辑。

项目中需要对连接状态进行处理，因为在生产环境中，网络不确定因素会导致客户端和zk服务端失去连接，一旦超过了session过期时间，那么客户端创建的临时节点也就会被删除，因此需要监听该状态事件，在session过期之后，重新创建临时节点。





查看ConnectionState.Lost的API，解释为：

当zk认为session已经过期了，就会触发LOST事件。

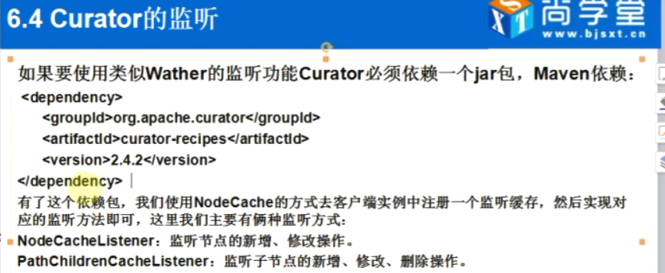
在检测到LOST事件之后，使用while(true)死循环，不断尝试重新连接zk sever，直至连接上为止。

blockUntilConnectedOrTimedOut()方法：当客户端连接上zk sever后，返回true，如果连接超时返回false。

（可以通过防火墙禁止某些流量，达到session过期的目的）



### curator监听（Watcher）



这里监听的方法相当于空间换时间，在客户端新增一个cache，将cache和zookeeper服务端进行比较，如果有变更，则将变更的数据保存在cache中。

而原生api和zkclient是通过重复注册实现的。

三种watcher：

1 Path cache:监视一个路径下 子节点的创建 删除以及数据的更新

2 Node cache 监视一个节点的创建 删除 更新，将节点的数据缓存在本地

3 Tree cache：Path cache和Node cache的合体，监视路径下的创建 删除 更新，并缓存路径下所有子节点的数据

#### PathCache

可以监听子节点的增加 删除 和修改，但是子节点的子节点变化不再监听

#### NodeCache

只会监听当前节点的增加 删除和修改

#### TreeCache

是PathCache和NodeCache的结合，既可以监听当前节点，也可以监听子节点的变化



分布式的锁全局同步，任何一个客户端不会有两个客户端都拥有相同的锁。

在集群环境中，都是多个jvm协同工作，当需要一些全局锁时就要用到上面介绍的分布式锁了。这种锁的缺点在于每次客户端需要获取锁时都需要与zk服务端交互，创建znode，等待着自己获取锁，网络通信会给服务器带来一定压力。

场景：更新缓存

处理逻辑：

1、 首选根据key获取资源，如果资源存在，用之。

2、如果不存在，则申请获取锁（使用共享锁)。

3、获取到锁以后，再次判断资源是否存在（防止重复更新），如果存在说明已经有人更新了，方法退出，否则更新缓存。

4、释放锁。

假设现在有10（1-10）个线程同时执行上诉逻辑，如果资源不存在，那么它们全部会执行第（2）步获取锁，在同一时刻，只会有1个线程获取锁，其它9个线程阻塞，等待获取锁。现在我们假设线程1获取到锁，开始执行（3-4）步动作，在第（3步）当中，再次判断资源是否存在，（肯定不存在因为它是第一个进去的），所以它负责加载资源放入缓存，然后释放锁， 再说其它线程（2-10）它们依次获取到锁，然后执行（3，4）动作，再次判断资源是否存在（已经存在了，因为1号线程已经放进去了），所以他们直接退出，释放锁。由此可见只有1号线程获取锁是有意义的，但是它们都需要与zook进行网络通讯，因此会给网络带来压力。

如果说我们有A,B 二台服务器进行集群，这10个线程获取锁的请求分别来自这2台服务器，例如（1-5）号线程来自A服务器， （6-10）号线程来自B服务器，那么它们与zk交互了10次，创建10个znode来申请锁，但是如果我们进行一定的优化，它们只要与zk交互2次就够了，我们来把上面的逻辑改造一下。

1、 首选根据key获取资源，如果资源存在，用之。

2、如果不存在，则申请获取锁（JVM进程内的锁)。

3、获取到锁(JVM进程内的锁)，再次判断资源是否存在，如果资源存在就退出，没啥好所的。

4、如果资源不存在，则申请获取锁(分布式锁)。

5、获取到锁（分布式锁）再次判断资源是否存在（防止重复更新），如果存在说明已经有人更新了，方法退出，否则更新缓存。

6、释放分布式锁，释放JVM进程内的锁。

http://blog.csdn.net/luckyzhoustar/article/details/50628419

http://www.cnblogs.com/luxiaoxun/p/4889764.html



<http://www.cnblogs.com/LiZhiW/p/4941771.html>

curator还有一个好处：当某个客户端突然挂了，本地上的信息都丢失了，当再次连接成功时，会收到已经存储在zk上节点变化的通知。比如，client1创建节点/super/1，client2创建节点/super/2，client3创建节点/super/3，这样，当client3挂掉了，再次连接上zk时，本地就收到/super/1和/super/2的通知。。不需要我们再主动去获取/super子节点的数据了。

## Leader选举

Curator提供两种leader选举方式：

1 LeaderLatch LeaderLatchListener

2 LeaderSelector LeaderSelectorListener LeaderSelectorListenerAdapter

(Spark中使用的是第一种方式)

### Leaderlatch

需要为每一个节点创建一个CuratorFramework，LeaderLatch，LeaderLatchListener

一个CuratorFramework就相当于一个zk客户端，启动，然后创建LeaderLatch，并指定刚才创建的CuratorFramework和leaderPath，注意不同节点的leaderPath必须一致。创建好LeaderLatch之后，要添加一个监听器，也就是为他注册一个LeaderLatchListener回调，如果某个节点成为leader，就会去调用这个节点的LeaderLatchListener的isLeader()方法，可以在这里写自己的业务逻辑。最后，调用LeaderLatch的start()方法，这个LeaderLatch会参加选举。

### 案例

CuratorFramework cf = CuratorFrameworkFactory.builder…

cf.start();

LeaderLatch latch = new LeaderLatch(cf,”/leader”);

latch.addListener(new LeaderLatchListener() {

public void notLeader() {

}

public void isLeader() {

}

});

latch.start();

### 方法介绍

LeaderLatch提供了如下方法；

start()/close(0 启动/停止LeaderLatch

addListener(LeaderLatchListener)/removeListener(LeaderLatchListener) 添加/删除LeaderLatchListener

hasLeadership() 如果Leaderlatch是leader，那么返回true，否则返回false

getLeader() 获取Leader

await() 等待LeaderLatch成为Leader

LeaderLatchListener的方法：

isLeader() 当Leaderlatch的hasleadership()从false到true后，会调用isLeader方法

notLeader() 当LeaderLatch的hasLeadership从true到false后，就会调用notLeader()方法

## 面试题

百度搜索zk面试题

Zookeeper是怎么实现的？ Zab协议 paxos协议

zk怎么实现分布式锁？

会话超时 连接超时