# [ZooKeeper学习总结（1）——ZooKeeper入门介绍](https://blog.csdn.net/u012562943/article/details/52963477)

目录

[概述](https://blog.csdn.net/u012562943/article/details/52963477#t0)

[ZooKeeper的安装](https://blog.csdn.net/u012562943/article/details/52963477#t1)

[ZooKeeper监控](https://blog.csdn.net/u012562943/article/details/52963477#t2)

[咱们在JMXLOCALONLY后边添加jmx的相关参数配置](https://blog.csdn.net/u012562943/article/details/52963477#t3)

[Zookeeper的存储模型](https://blog.csdn.net/u012562943/article/details/52963477#t4)

[Zookeeper客户端的使用](https://blog.csdn.net/u012562943/article/details/52963477#t5)

[可使用zkClish -server localhost来连接到Zookeeper服务上](https://blog.csdn.net/u012562943/article/details/52963477#t6)

[使用ls 可查看根节点下有哪些子节点可以双击Tab键查看更多命令](https://blog.csdn.net/u012562943/article/details/52963477#t7)

[Zookeeper创建Znode](https://blog.csdn.net/u012562943/article/details/52963477#t8)

[Zookeeper删除Znode](https://blog.csdn.net/u012562943/article/details/52963477#t9)

[Zookeeper的相关操作](https://blog.csdn.net/u012562943/article/details/52963477#t10)

ZooKeeper的API

观察触发器

## 1. 概述

Zookeeper是Hadoop的一个子项目，它是分布式系统中的协调系统，可提供的服务主要有：配置服务、名字服务、分布式同步、组服务等。

它有如下的一些特点：

* **简单**

Zookeeper的核心是一个精简的文件系统，它支持一些简单的操作和一些抽象操作，例如，排序和通知。

* **丰富**

Zookeeper的原语操作是很丰富的，可实现一些协调数据结构和协议。例如，分布式队列、分布式锁和一组同级别节点中的“领导者选举”。

* **高可靠**

Zookeeper支持集群模式，可以很容易的解决单点故障问题。

* **松耦合交互**

不同进程间的交互不需要了解彼此，甚至可以不必同时存在，某进程在zookeeper中留下消息后，该进程结束后其它进程还可以读这条消息。

* **资源库**

Zookeeper实现了一个关于通用协调模式的开源共享存储库，能使开发者免于编写这类通用协议。

## 2. ZooKeeper的安装

* **独立模式安装**

Zookeeper的运行环境是需要java的，建议安装oracle的java6.

可去官网下载一个稳定的版本，然后进行安装： [***http://zookeeper.apache.org/***](http://zookeeper.apache.org/)

解压后在zookeeper的conf目录下创建配置文件zoo.cfg，里面的配置信息可参考统计目录下的zoo\_sample.cfg文件，我们这里配置为：

tickTime=**2000**initLimit=**10**syncLimit=**5**dataDir=/opt/zookeeper-data/  
clientPort=**2181**

tickTime ： 指定了ZooKeeper的基本时间单位（以毫秒为单位）；

initLimit ： 指定了启动zookeeper时，zookeeper实例中的随从实例同步到领导实例的初始化连接时间限制，超出时间限制则连接失败（以tickTime为时间单位）；

syncLimit ： 指定了zookeeper正常运行时，主从节点之间同步数据的时间限制，若超过这个时间限制，那么随从实例将会被丢弃；

dataDir ： zookeeper存放数据的目录；

clientPort ： 用于连接客户端的端口。

###环境变量配置 编辑 .bashrc 文件, 在文件末尾添加以下环境变量配置:

# ZooKeeper Env

export ZOOKEEPER\_HOME=/opt/zookeeper

export PATH=$PATH:$ZOOKEEPER\_HOME/bin

运行以下命令使环境变量生效:

source .bashrc

* **启动一个本地的ZooKeeper 实例**

*%* zkServer.sh start

检查ZooKeeper是否正在运行

echo ruok | *nc* localhost **2181**

若是正常运行的话会打印“imok”。

## 3. ZooKeeper监控

* **远程JMX 配置**

默认情况下，zookeeper是支持本地的jmx监控的。若需要远程监控zookeeper，则需要进行进行如下配置。

默认的配置有这么一行：

ZOOMAIN="-Dcom.sun.management.jmxremote -Dcom.sun.management.jmxremote.local.only=$JMXLOCALONLY org.apache.zookeeper.server.quorum.QuorumPeerMain"

#### 咱们在$JMXLOCALONLY后边添加jmx的相关参数配置：

ZOOMAIN="-Dcom.sun.management.jmxremote  
 -Dcom.sun.management.jmxremote.local.only=$JMXLOCALONLY  
 -Djava.rmi.server.hostname=192.168.1.8  
 -Dcom.sun.management.jmxremote.port=1911  
 -Dcom.sun.management.jmxremote.ssl=false  
 -Dcom.sun.management.jmxremote.authenticate=false  
 org.apache.zookeeper.server.quorum.QuorumPeerMain"

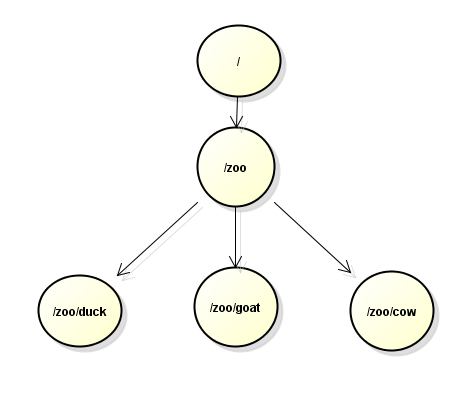
这样就可以远程监控了，可以用jconsole.exe或jvisualvm.exe等工具对其进行监控。

* **身份验证**

这里没有配置验证信息，如果需要请参见我的博文 jvisualvm 远程监控tomcat ：http://www.cnblogs.com/leocook/p/jvisualvmandtomcat.html

## 4. Zookeeper的存储模型

Zookeeper的数据存储采用的是结构化存储，结构化存储是没有文件和目录的概念，里边的目录和文件被抽象成了节点（node），zookeeper里可以称为znode。Znode的层次结构如下图：



最上边的是根目录，下边分别是不同级别的子目录。

## 5. Zookeeper客户端的使用

* **zkCli.sh**

#### 可使用./zkCli.sh -server localhost来连接到Zookeeper服务上。

#### 使用ls /可查看根节点下有哪些子节点，可以双击Tab键查看更多命令。

* **Java 客户端**

可创建org.apache.zookeeper.ZooKeeper对象来作为zk的客户端，注意，java api里创建zk客户端是异步的，为防止在客户端还未完成创建就被使用的情况，这里可以使用同步计时器，确保zk对象创建完成再被使用。

* **C 客户端**

可以使用zhandle\_t指针来表示zk客户端，可用zookeeper\_init方法来创建。可在ZK\_HOME\src\c\src\ cli.c查看部分示例代码。

## 6. Zookeeper创建Znode

Znode有两种类型：短暂的和持久的。短暂的znode在创建的客户端与服务器端断开（无论是明确的断开还是故障断开）连接时，该znode都会被删除；相反，持久的znode则不会。

**public class** CreateGroup **implements** Watcher{  
 **private static final int** SESSION\_TIMEOUT = **1000**;//会话延时  
  
 **private** ZooKeeper zk = **null**;  
 **private** CountDownLatch *countDownLatch* = **new** CountDownLatch(**1**);//同步计数器  
  
 **public void** process(WatchedEvent event) {  
 **if**(event.getState() == KeeperState.SyncConnected){  
 *countDownLatch*.countDown();//计数器减一  
 }  
 }  
  
 /\*\*  
 \* 创建zk对象  
 \* 当客户端连接上zookeeper时会执行process(event)里的countDownLatch.countDown()，计数器的值变为0，则countDownLatch.await()方法返回。  
 \* @param hosts  
 \* @throws IOException  
 \* @throws **InterruptedException** \*/  
 **public void** connect(**String** hosts) **throws** IOException, InterruptedException {  
 zk = **new** ZooKeeper(hosts, SESSION\_TIMEOUT, **this**);  
 countDownLatch.await();//阻塞程序继续执行  
 }  
  
 /\*\*  
 \* 创建group  
 \*  
 \* @param groupName 组名  
 \* @throws KeeperException  
 \* @throws InterruptedException  
 \*/  
 **public void** create(String groupName) **throws** KeeperException, InterruptedException {  
 String path = "/" + groupName;  
 String createPath = zk.create(path, **null**, Ids.OPEN\_ACL\_UNSAFE*/\*允许任何客户端对该znode进行读写\*/*, CreateMode.PERSISTENT*/\*持久化的znode\*/*);  
 System.out.println("Created " + createPath);  
 }  
  
 /\*\*  
 \* 关闭zk  
 \* @throws InterruptedException  
 \*/  
 **public void** close() **throws** InterruptedException {  
 **if**(zk != **null**){  
 **try** {  
 zk.close();  
 } **catch** (InterruptedException e) {  
 **throw** e;  
 }**finally**{  
 zk = **null**;  
 System.gc();  
 }  
 }  
 }  
}

这里我们使用了同步计数器CountDownLatch，在connect方法中创建执行了zk = new ZooKeeper(hosts, SESSION\_TIMEOUT, this);之后，下边接着调用了CountDownLatch对象的await方法阻塞，因为这是zk客户端不一定已经完成了与服务端的连接，在客户端连接到服务端时会触发观察者调用process()方法，我们在方法里边判断一下触发事件的类型，完成连接后计数器减一，connect方法中解除阻塞。

还有两个地方需要注意：这里创建的znode的访问权限是open的，且该znode是持久化存储的。

测试类如下：

**public class CreateGroupTest** {  
 **private static String** *hosts* = "192.168.1.8";  
 **private static String** *groupName* = "zoo";  
  
 **private** CreateGroup *createGroup* = **null**;  
  
 /\*\*  
 \* init  
 \* @throws **InterruptedException** \* @throws KeeperException  
 \* @throws IOException  
 \*/  
 @Before  
 **public void** init() **throws** KeeperException, **InterruptedException**, IOException {  
 *createGroup* = **new** CreateGroup();  
 *createGroup*.connect(*hosts*);  
 }  
  
 @Test  
 **public void** testCreateGroup() **throws** KeeperException, **InterruptedException** {  
 *createGroup*.create(*groupName*);  
 }  
  
 /\*\*  
 \* 销毁资源  
 \*/  
 @After  
 **public void** destroy() {  
 **try** {  
 *createGroup*.close();  
 *createGroup* = **null**;  
 **System**.gc();  
 } **catch** (**InterruptedException** e) {  
 e.printStackTrace();  
 }  
 }  
}

由于zk对象的创建和销毁代码是可以复用的，所以这里我们把它分装成了接口：

/\*\*  
 \* 连接的观察者，封装了zk的创建等  
 \* @author leo  
 \*  
 \*/  
**public class ConnectionWatcher implements** Watcher {  
 **private static final int *SESSION\_TIMEOUT*** = **5000**;  
  
 **protected** ZooKeeper *zk* = **null**;  
 **private** CountDownLatch *countDownLatch* = **new** CountDownLatch(**1**);  
  
 **public void** process(WatchedEvent event) {  
 KeeperState state = event.getState();  
  
 **if**(state == KeeperState.SyncConnected){  
 *countDownLatch*.countDown();  
 }  
 }  
  
 /\*\*  
 \* 连接资源  
 \* @param hosts  
 \* @throws IOException  
 \* @throws **InterruptedException** \*/  
 **public void** connection(**String** hosts) **throws** IOException, **InterruptedException** {  
 *zk* = **new** ZooKeeper(hosts, ***SESSION\_TIMEOUT***, **this**);  
 *countDownLatch*.await();  
 }  
  
 /\*\*  
 \* 释放资源  
 \* @throws **InterruptedException** \*/  
 **public void** close() **throws InterruptedException** {  
 **if** (**null** != *zk*) {  
 **try** {  
 *zk*.close();  
 } **catch** (**InterruptedException** e) {  
 **throw** e;  
 }**finally**{  
 *zk* = **null**;  
 **System**.gc();  
 }  
 }  
 }  
}

## 7. Zookeeper删除Znode

/\*\*  
 \* 删除分组  
 \* @author leo  
 \*  
 \*/  
**public class DeleteGroup extends** ConnectionWatcher {  
 **public void** delete(**String** groupName) {  
 **String** path = "/" + groupName;  
  
 **try** {  
 List<**String**> children = zk.getChildren(path, **false**);  
  
 **for**(**String** child : children){  
 zk.delete(path + "/" + child, -**1**);  
 }  
 zk.delete(path, -**1**);//版本号为-1，  
 } **catch** (KeeperException e) {  
 e.printStackTrace();  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 }  
}

zk.delete(path,version)方法的第二个参数是znode版本号，如果提供的版本号和znode版本号一致才会删除这个znode，这样可以检测出对znode的修改冲突。通过将版本号设置为-1，可以绕过这个版本检测机制，无论znode的版本号是什么，都会直接将其删除。

测试类：

**public class DeleteGroupTest** {  
 **private static final String *HOSTS*** = "192.168.1.137";  
 **private static final String *groupName*** = "zoo";  
  
 **private** DeleteGroup *deleteGroup* = **null**;  
  
 @Before  
 **public void** init() **throws** IOException, **InterruptedException** {  
 *deleteGroup* = **new** DeleteGroup();  
 *deleteGroup*.connection(***HOSTS***);  
 }  
  
 @Test  
 **public void** testDelete() **throws** IOException, **InterruptedException**, KeeperException {  
 *deleteGroup*.delete(***groupName***);  
 }  
  
 @After  
 **public void** destroy() **throws InterruptedException** {  
 **if**(**null** != *deleteGroup*){  
 **try** {  
 *deleteGroup*.close();  
 } **catch** (**InterruptedException** e) {  
 **throw** e;  
 }**finally**{  
 *deleteGroup* = **null**;  
 **System**.gc();  
 }  
 }  
 }  
}

## 8. Zookeeper的相关操作

ZooKeeper中共有9中操作：

create：创建一个znode

delete：删除一个znode

exists：测试一个znode

getACL，setACL：获取/设置一个znode的ACL（权限控制）

getChildren：获取一个znode的子节点

getData，setData：获取/设置一个znode所保存的数据

sync：将客户端的znode视图与ZooKeeper同步

这里更新数据是必须要提供znode的版本号（也可以使用-1强制更新，这里可以执行前通过exists方法拿到znode的元数据Stat对象，然后从Stat对象中拿到对应的版本号信息），如果版本号不匹配，则更新会失败。因此一个更新失败的客户端可以尝试是否重试或执行其它操作。

## 9. ZooKeeper的API

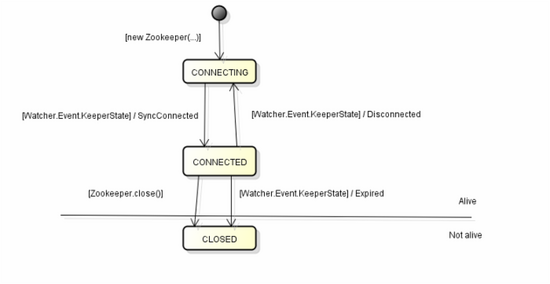
ZooKeeper的api支持多种语言，在操作时可以选择使用同步api还是异步api。同步api一般是直接返回结果，异步api一般是通过回调来传送执行结果的，一般方法中有某参数是类AsyncCallback的内部接口，那么该方法应该就是异步调用，回调方法名为processResult。

## 10. 观察触发器

可以对客户端和服务器端之间的连接设置观察触发器（后边称之为zookeeper的状态观察触发器），也可以对znode设置观察触发器。

* **状态观察器**

zk的整个生命周期如下：



可在创建zk对象时传入一个观察器，在完成CONNECTING状态到CONNECTED状态时，观察器会触发一个事件，该触发的事件类型为NONE，通过event.getState()方法拿到事件状态为SyncConnected。有一点需要注意的就是，在zk调用close方法时不会触发任何事件，因为这类的显示调用是开发者主动执行的，属于可控的，不用使用事件通知来告知程序。这一块在下篇博文还会详细解说。

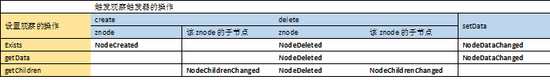
* **设置znode 的观察器**

可以在读操作exists、getChildren和getData上设置观察，在执行写操作create、delete和setData将会触发观察事件，当然，在执行写的操作时，也可以选择是否触发znode上设置的观察器，具体可查看相关的api。

当观察的znode 被创建、删除或其数据被更新时 ，设置在exists上的观察将会被触发；

当观察的znode 被删除或数据被更新时 ，设置在getData上的观察将会被触发；

当观察的znode 的子节点被创建、删除或znode 自身被删除时 ，设置在getChildren上的观察将会被触发，可通过观察事件的类型来判断被删除的是znode还是它的子节点。



对于 NodeCreated 和 NodeDeleted 根据路径就能发现是哪个znode被写；对于 NodeChildrenChanged 可根据getChildren来获取新的子节点列表。

注意：在收到收到触发事件到执行读操作之间，znode的状态可能会发生状态，这点需要牢记。

# [ZooKeeper学习总结（2）——ZooKeeper开源Java客户端ZkClient使用](https://blog.csdn.net/u012562943/article/details/52963506)

zkclient是zookeeper的Java客户端。它让Zookeeper API 使用起来更简单；它非常方便订阅各种事件并自动重新绑定事件（会话建立、节点修改、节点删除、子节点变更等）；它提供了session过期自动重连等机制。今天我们主要讲解的是如何使用zkclient的基础api。

首先我们创建一个maven工程，将jar引进来，下面是maven的依赖：



## 1、创建会话 ZooKeeper的开源客户端ZkClient

（1）zkServers 指的是zk的服务器列表，由英文状态逗号分开的host：port字符串组成，每一个都代表一台zk机器，例如192,168,11,160:2181,192,168,11,161:2181...

（2）sessionTimeout 会话超时时间，单位为毫秒，默认为30000ms

（3）connectionTimeout 连接创建的超市时间，单位为毫秒，此参数表明如果在这个时间段内还是无法和zk建立连接，那么就放弃连接直接抛出异常

（4）connection IZKConnection 接口的实现类

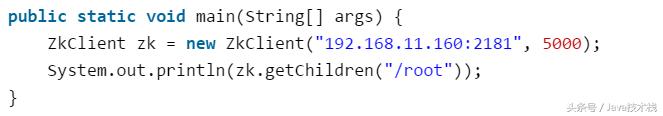
（5）zkSerializer 自定义的序列化器

在使用zk的api创建连接的时候我们知道，创建连接的过程是一个异步过程，我们需要自己通过线程阻塞来监听链接是否创建成功，而在zkclient里，将异步方式转为了同步方式。这样会更加方便。

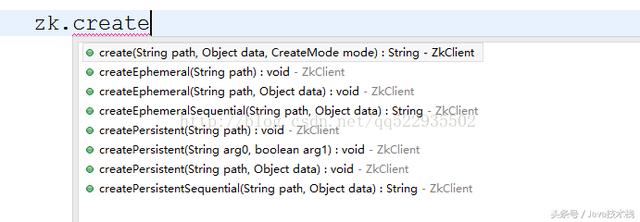
IZKconnection接口。这个接口是对zk原生接口最直接的包装，也是和zk最直接的交互层，里面包含了增，删，改，查等接口。zkClient有两种实现方式分别是ZkConnection和InMemoryConnection ，而ZkConnection是常用的。

ZkSerializer接口。这个接口是用来完成data数据序列化用的，默认情况下zkc使用Java自带的序列化方式对对象进行序列化。

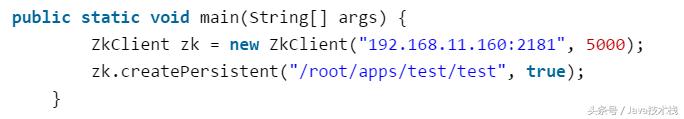
在zkclient的构造方法里我们没有找到对watcher的传入，那如何实现监听呢，zkclient使用listener来实现watcher注册

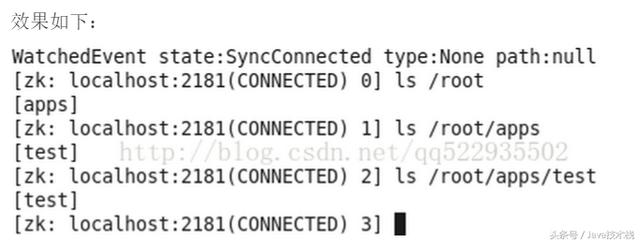


## 2、创建节点



（1）path 节点路径 （2）data 数据 （3）mode 节点类型 （4）cal 权限策略 （5）callback 异步回调函数 （6）context 上下文对象 （7）createParents 指定是否创建父节点。在这些参数里，和zk自带的api都差不多，其中data因为有了自定义的序列化工具，所以可以传入更为复杂的对象。

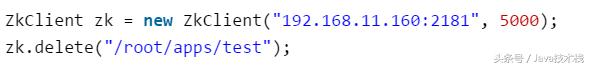




## 3、删除节点

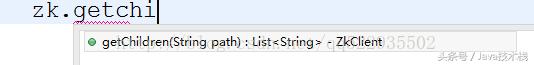
刚才我们创建了一个永久的节点，下面我们来使用delete方法删除这个节点：





但还有一个方法：deleteRecursive 这个方法可以帮助我们逐层遍历删除节点的工作。

## 4、获取节点，注册监听



这个方法没有注册watcher，所以我们现在先讲一个如何注册listener。下面这个listener接口可以完成节点的监听。



问题是我们如何将这个监听器注册呢，我们可以使用subscribeChildChanges方法完成注册，下面我们来看一下demo



经过测试，得出以下结论：

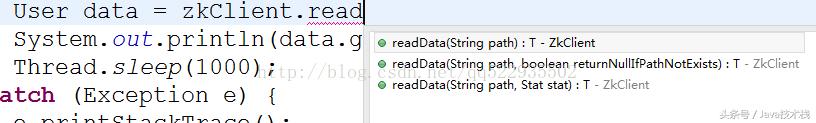
（1）客户端可以对一个不存在的节点进行子节点变更的监听

（2）一旦客户端对一个节点注册了子节点列表变更监听之后，那么该节点的字节列表发生变化的时候，服务端都会通知客户端，并将最新的子节点列表发送给客户端。

（3）该节点本身的创建或删除会通知到客户端。

（4）listener只需要注册一次会一直生效。

## 5、获取与更新数据



（1）returnNullIfPathNotExists 默认情况下 在调用这个方法的时候 如果指定的节点不存在会抛出异常，如果设置了这个参数 如果节点不存在就返回null 而不会抛出异常

（2）stat 指定数据节点的节点状态信息 用户是在接口中传入一个旧的stat变量 该stat变量会在方法执行过程中被来自服务器响应的新的stat对象替换。

来看一下DEMO案例





# [ZooKeeper学习总结（3）——ZooKeeper常见面试题](https://blog.csdn.net/u012562943/article/details/76128183)

**Zookeeper是什么框架**  
分布式的、开源的分布式应用程序协调服务，原本是Hadoop、HBase的一个重要组件。它为分布式应用提供一致性服务的软件，包括：配置维护、域名服务、分布式同步、组服务等。  
**应用场景**  
Zookeeper的功能很强大，应用场景很多，结合我实际工作中使用Dubbo框架的情况，Zookeeper主要是做注册中心用。基于Dubbo框架开发的提供者、消费者都向Zookeeper注册自己的URL，消费者还能拿到并订阅提供者的注册URL，以便在后续程序的执行中去调用提供者。而提供者发生了变动，也会通过Zookeeper向订阅的消费者发送通知。  
**Paxos算法& Zookeeper使用协议**  
Paxos算法是分布式选举算法，Zookeeper使用的 ZAB协议（Zookeeper原子广播），二者有相同的地方，比如都有一个Leader，用来协调N个Follower的运行；Leader要等待超半数的Follower做出正确反馈之后才进行提案；二者都有一个值来代表Leader的周期。  
不同的地方在于：  
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。  
Paxos算法、ZAB协议要想讲清楚可不是一时半会的事儿，自1990年莱斯利·兰伯特提出Paxos算法以来，因为晦涩难懂并没有受到重视。后续几年，兰伯特通过好几篇论文对其进行更进一步地解释，也直到06年谷歌发表了三篇论文，选择Paxos作为chubby cell的一致性算法，Paxos才真正流行起来。  
对于普通开发者来说，尤其是学习使用Zookeeper的开发者明确一点就好：分布式Zookeeper选举Leader服务器的算法与Paxos有很深的关系。  
**选举算法和流程**  
详情可参看我之前的文章《ZooKeeper集群安装配置使用》第6节“ZooKeeper选举机制”，有个简单的描述。  
**Zookeeper有哪几种节点类型**  
持久：创建之后一直存在，除非有删除操作，创建节点的客户端会话失效也不影响此节点。  
持久顺序：跟持久一样，就是父节点在创建下一级子节点的时候，记录每个子节点创建的先后顺序，会给每个子节点名加上一个数字后缀。  
临时：创建客户端会话失效（注意是会话失效，不是连接断了），节点也就没了。不能建子节点。  
临时顺序：不用解释了吧。  
**Zookeeper对节点的watch监听通知是永久的吗？**  
不是。官方声明：一个Watch事件是一个一次性的触发器，当被设置了Watch的数据发生了改变的时候，则服务器将这个改变发送给设置了Watch的客户端，以便通知它们。  
为什么不是永久的，举个例子，如果服务端变动频繁，而监听的客户端很多情况下，每次变动都要通知到所有的客户端，这太消耗性能了。  
一般是客户端执行getData(“/节点A”,true)，如果节点A发生了变更或删除，客户端会得到它的watch事件，但是在之后节点A又发生了变更，而客户端又没有设置watch事件，就不再给客户端发送。  
在实际应用中，很多情况下，我们的客户端不需要知道服务端的每一次变动，我只要最新的数据即可。  
**部署方式？集群中的机器角色都有哪些？集群最少要几台机器**  
单机，集群。Leader、Follower。集群最低3（2N+1）台，保证奇数，主要是为了选举算法。  
**集群如果有3台机器，挂掉一台集群还能工作吗？挂掉两台呢？**  
记住一个原则：过半存活即可用。  
**集群支持动态添加机器吗？**  
其实就是水平扩容了，Zookeeper在这方面不太好。两种方式：  
全部重启：关闭所有Zookeeper服务，修改配置之后启动。不影响之前客户端的会话。  
逐个重启：顾名思义。这是比较常用的方式。