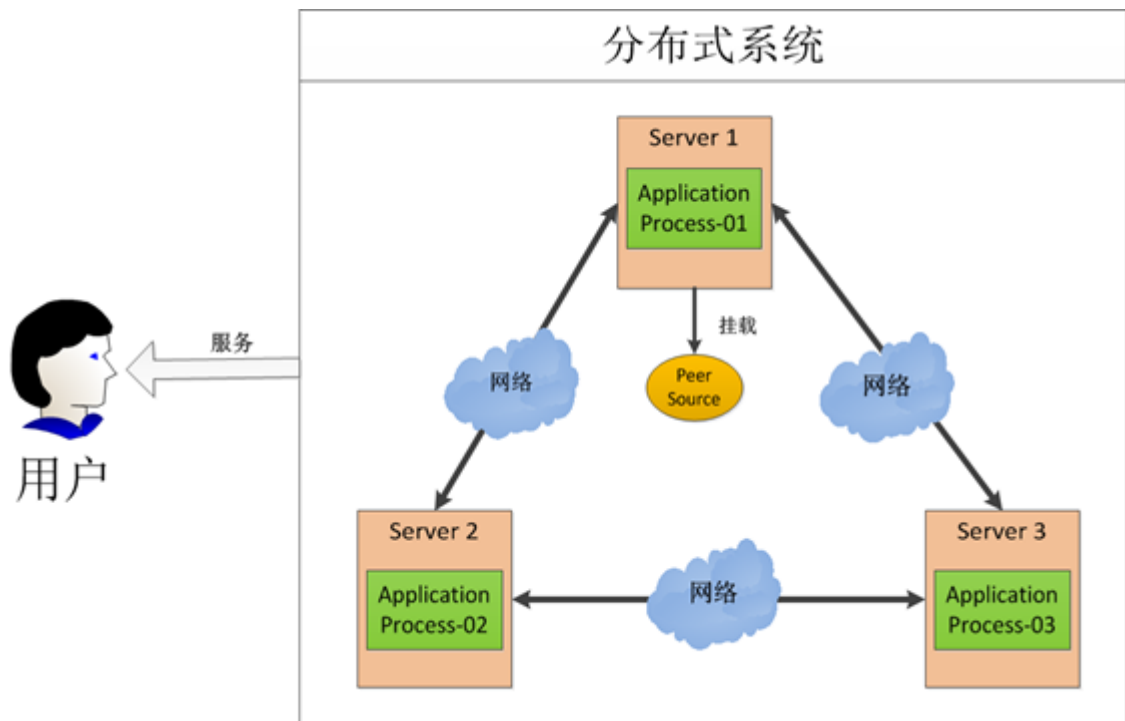


Zookeeper

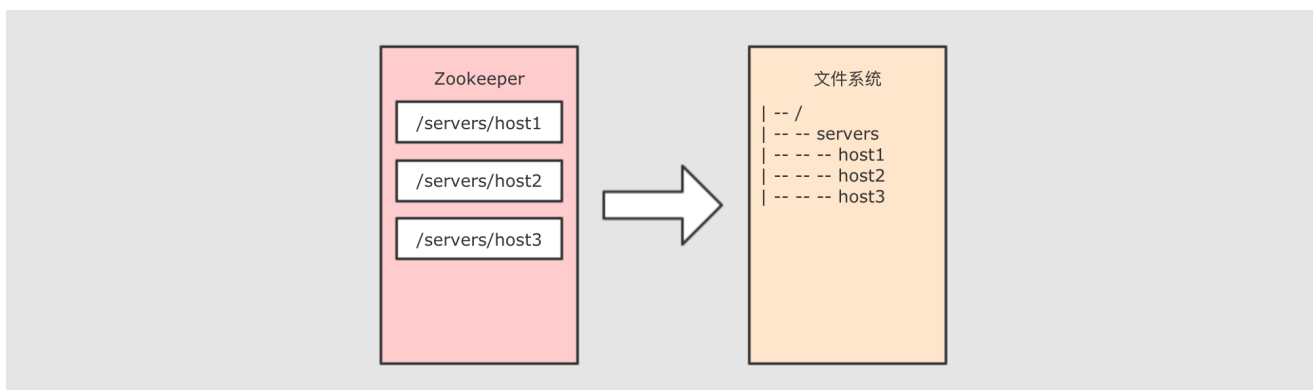
1 Zookeeper 的概述

- Zookeeper 是一个开源的分布式协调服务框架，主要用来解决分布式集群中应用系统的一致性问题 and 数据管理问题



2: Zookeeper 的特点

- Zookeeper 本质上是一个分布式文件系统, 适合存放小文件，也可以理解为一个数据库



- 在上图左侧, Zookeeper 中存储的其实是一个又一个 Znode, Znode 是 Zookeeper 中的节点

- Znode 是有路径的, 例如 `/data/host1` , `/data/host2` , 这个路径也可以理解为是 Znode 的 Name
- Znode 也可以携带数据, 例如说某个 Znode 的路径是 `/data/host1` , 其值是一个字符串 `"192.168.0.1"`
- 正因为 Znode 的特性, 所以 Zookeeper 可以对外提供出一个类似于文件系统的试图, 可以通过操作文件系统的方式操作 Zookeeper
 - 使用路径获取 Znode
 - 获取 Znode 携带的数据
 - 修改 Znode 携带的数据
 - 删除 Znode
 - 添加 Znode

3.Zookeeper的应用场景

3.1 数据发布/订阅

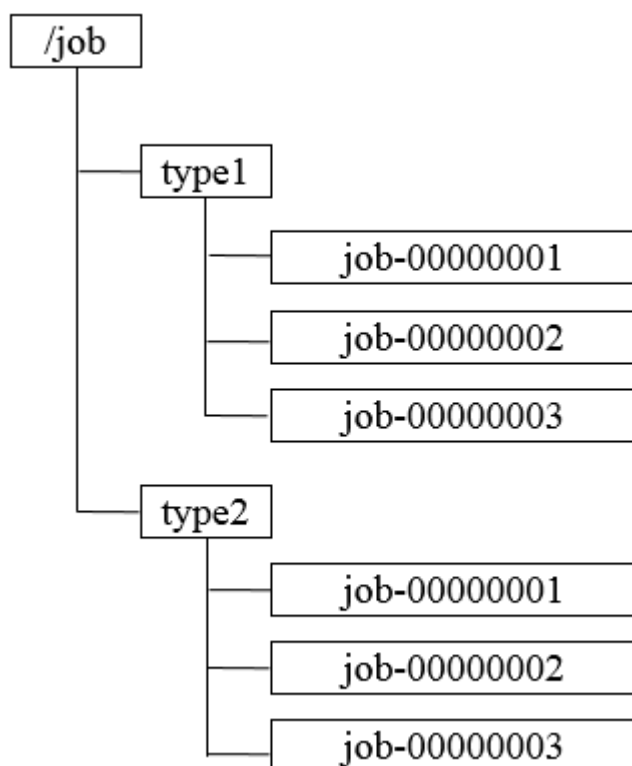
数据发布/订阅系统, 需要发布者将数据发布到Zookeeper的节点上, 供订阅者进行数据订阅, 进而达到动态获取数据的目的, 实现配置信息的集中式管理和数据的动态更新。

发布/订阅一般有两种设计模式: 推模式和拉模式, 服务端主动将数据更新发送给所有订阅的客户端称为推模式; 客户端主动请求获取最新数据称为拉模式。

Zookeeper采用了推拉相结合的模式, 客户端向服务端注册自己需要关注的节点, 一旦该节点数据发生变更, 那么服务端就会向相应的客户端推送Watcher事件通知, 客户端接收到此通知后, 主动到服务端获取最新的数据。

3.2 命名服务

命名服务是分步实现系统中较为常见的一类场景, 分布式系统中, 被命名的实体通常可以是集群中的机器、提供的服务地址或远程对象等, 通过命名服务, 客户端可以根据指定名字来获取资源的实体, 在分布式环境中, 上层应用仅仅需要一个全局唯一的名字。Zookeeper可以实现一套分布式全局唯一ID的分配机制。



通过调用Zookeeper节点创建的API接口就可以创建一个顺序节点，并且在API返回值中会返回这个节点的完整名字，利用此特性，可以生成全局ID，其步骤如下

1. 客户端根据任务类型，在指定类型的任务下通过调用接口创建一个顺序节点，如"job-"。
2. 创建完成后，会返回一个完整的节点名，如"job-00000001"。
3. 客户端拼接type类型和返回值后，就可以作为全局唯一ID了，如"type2-job-00000001"。

3.3 分布式协调/通知

Zookeeper中特有的Watcher注册于异步通知机制，能够很好地实现分布式环境下不同机器，甚至不同系统之间的协调与通知，从而实现对数据变更的实时处理。通常的做法是不同的客户端都对Zookeeper上的同一个数据节点进行Watcher注册，监听数据节点的变化（包括节点本身和子节点），若数据节点发生变化，那么所有订阅的客户端都能够接收到相应的Watcher通知，并作出相应处理。

在绝大多数分布式系统中，系统机器间的通信无外乎 **心跳检测、工作进度汇报和系统调度**。

① **心跳检测**，不同机器间需要检测到彼此是否在正常运行，可以使用Zookeeper实现机器间的心跳检测，基于其临时节点特性（临时节点的生存周期是客户端会话，客户端若当即后，其临时节点自然不再存在），可以让不同机器都在Zookeeper的一个指定节点下创建临时子节点，不同的机器之间可以根据这个临时子节点来判断对应的客户端机器是否存活。通过Zookeeper可以大大减少系统耦合。

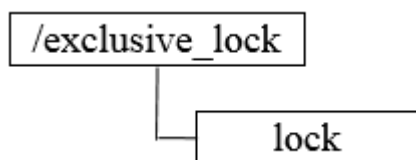
② **工作进度汇报**，通常任务被分发到不同机器后，需要实时地将自己的任务执行进度汇报给分发系统，可以在Zookeeper上选择一个节点，每个任务客户端都在这个节点下面创建临时子节点，这样不仅可以判断机器是否存活，同时各个机器可以将自己的任务执行进度写到该临时节点中去，以便中心系统能够实时获取任务的执行进度。

③ **系统调度**，Zookeeper能够实现如下系统调度模式：分布式系统由控制台和一些客户端系统两部分构成，控制台的职责就是需要将一些指令信息发送给所有的客户端，以控制他们进行相应的业务逻辑，后台管理人员在控制台上做一些操作，实际上就是修改Zookeeper上某些节点的数据，Zookeeper可以把数据变更以时间通知的形式发送给订阅客户端。

3.4分布式锁

分布式锁用于控制分布式系统之间同步访问共享资源的一种方式，可以保证不同系统访问一个或一组资源时的一致性，主要分为排它锁和共享锁。

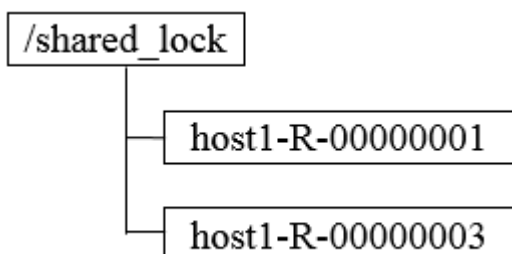
排它锁又称为写锁或独占锁，若事务T1对数据对象O1加上了排它锁，那么在整个加锁期间，只允许事务T1对O1进行读取和更新操作，其他任何事务都不能再对这个数据对象进行任何类型的操作，直到T1释放了排它锁。



① **获取锁**，在需要获取排它锁时，所有客户端通过调用接口，在/exclusive_lock节点下创建临时子节点/exclusive_lock/lock。Zookeeper可以保证只有一个客户端能够创建成功，没有成功的客户端需要注册/exclusive_lock节点监听。

② **释放锁**，当获取锁的客户端宕机或者正常完成业务逻辑都会导致临时节点的删除，此时，所有在/exclusive_lock节点上注册监听的客户端都会收到通知，可以重新发起分布式锁获取。

共享锁又称为读锁，若事务T1对数据对象O1加上共享锁，那么当前事务只能对O1进行读取操作，其他事务也只能对这个数据对象加共享锁，直到该数据对象上的所有共享锁都被释放。在需要获取共享锁时，所有客户端都会到/shared_lock下面创建一个临时顺序节点



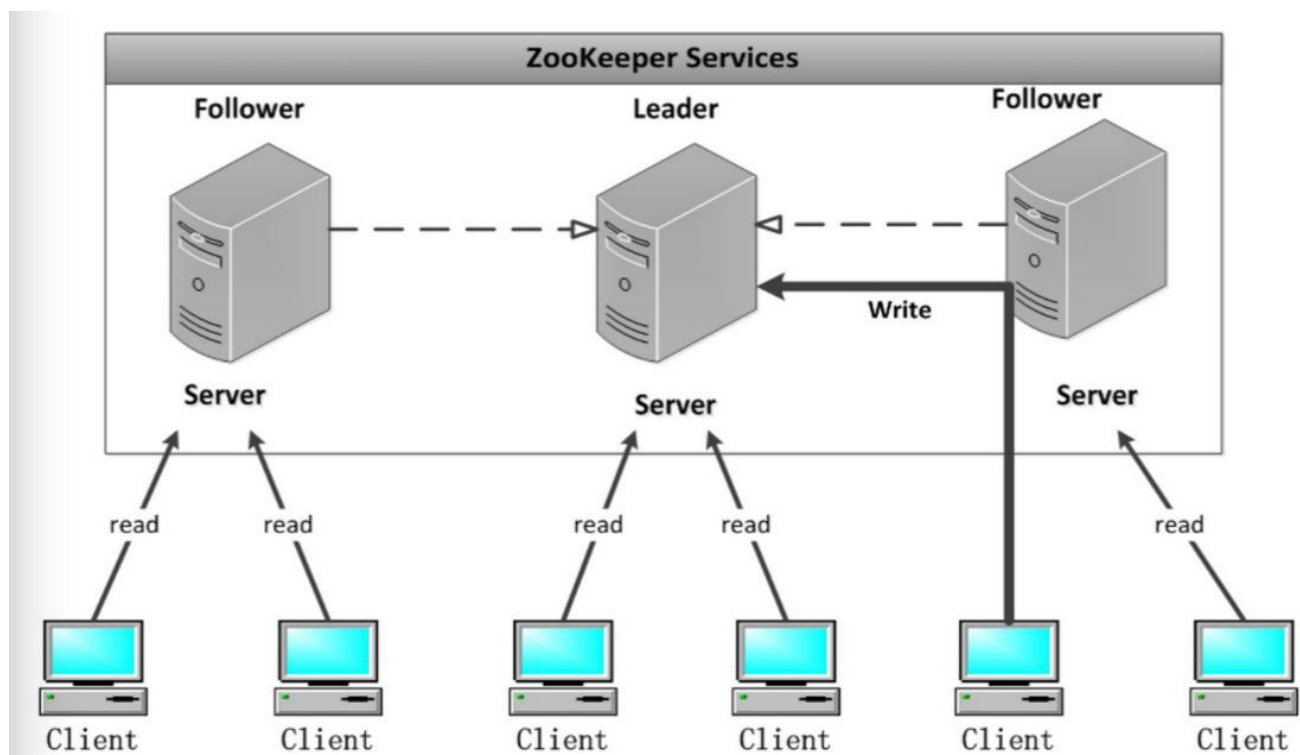
3.5 分布式队列

有一些时候，多个团队需要共同完成一个任务，比如，A团队将Hadoop集群计算的结果交给B团队继续计算，B完成了自己任务再交给C团队继续做。这就有点像业务系统的工作流一样，一环一环地传下去。

分布式环境下，我们同样需要一个类似单进程队列的组件，用来实现跨进程、跨主机、跨网络的数据共享和数据传递，这就是我们的分布式队列。

4.Zookeeper的架构

Zookeeper集群是一个基于主从架构的高可用集群



每个服务器承担如下三种角色中的一种

- **Leader** 一个Zookeeper集群同一时间只会有一个实际工作的Leader，它会发起并维护与各Follower及Observer间的心跳。所有的写操作必须要通过Leader完成再由Leader将写操作广播给其它服务器。
- **Follower** 一个Zookeeper集群可能同时存在多个Follower，它会响应Leader的心跳。Follower可直接处理并返回客户端的读请求，同时会将写请求转发给Leader处理，并且负责在Leader处理写请求时对请求进行投票。
- **Observer** 角色与Follower类似，但是无投票权。

角色		描述
领导者 (Leader)		1:leader 是集群工作的核心 集群内部各个服务器的调度者 2:Leader 负责进行投票选举 3:处理事务性（写操作）请求 4:参与集群投票
学习者 (Learner)	跟随者 (Follower)	1: Follower 用于接收客户端请求, 并向客户端返回结果 2:处理客户端非事务（读操作） 请求 3:转发事务请求给 Leader 4:参与集群投票
	观察者 (Observer)	1: Observer 用于接收客户端请求, 并向客户端返回结果 2:处理客户端非事务（读操作） 请求 3:转发事务请求给 Leader 4:不参与集群投票
客户端 (Client)		请求发起方

5:Zookeeper的选举机制

Leader选举是保证分布式**数据一致性**的关键所在。当Zookeeper集群中的一台服务器出现以下两种情况之一时，需要进入Leader选举。

5.1. 服务器启动时期的Leader选举

若进行Leader选举，则至少需要两台机器，这里选取3台机器组成的服务器集群为例。在集群初始化阶段，当有一台服务器Server1启动时，其单独无法进行和完成Leader选举，当第二台服务器Server2启动时，此时两台机器可以相互通信，每台机器都试图找到Leader，于是进入Leader选举过程。选举过程如下

(1) **每个Server发出一个投票**。由于是初始情况，Server1和Server2都会将自己作为Leader服务器来进行投票，每次投票会包含所推举的服务器的myid和ZXID，使用(myid, ZXID)来表示，此时Server1的投票为(1, 0)，Server2的投票为(2, 0)，然后各自将这个投票发给集群中其他机器。

(2) **接受来自各个服务器的投票**。集群的每个服务器收到投票后，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。

(3) **处理投票**。针对每一个投票，服务器都需要将别人的投票和自己的投票进行PK，PK规则如下

- **优先检查ZXID**。ZXID比较大的服务器优先作为Leader。
- **如果ZXID相同，那么就比较myid**。myid较大的服务器作为Leader服务器。

对于Server1而言，它的投票是(1, 0)，接收Server2的投票为(2, 0)，首先会比较两者的ZXID，均为0，再比较myid，此时Server2的myid最大，于是更新自己的投票为(2, 0)，然后重新投票，对于Server2而言，其无须更新自己的投票，只是再次向集群中所有机器发出上一次投票信息即可。

(4) **统计投票**。每次投票后，服务器都会统计投票信息，判断是否已经有过半机器接受到相同的投票信息，对于Server1、Server2而言，都统计出集群中已经有两台机器接受了(2, 0)的投票信息，此时便认为已经选出了Leader。

(5) **改变服务器状态**。一旦确定了Leader，每个服务器就会更新自己的状态，如果是Follower，那么就变更为FOLLOWING，如果是Leader，就变更为LEADING。

5.2. 服务器运行时期的Leader选举

在Zookeeper运行期间，Leader与非Leader服务器各司其职，即便当有非Leader服务器宕机或新加入，此时也不会影响Leader，但是一旦Leader服务器挂了，那么整个集群将暂停对外服务，进入新一轮Leader选举，其过程和启动时期的Leader选举过程基本一致过程相同。

6: Zookeeper安装

集群规划

服务器IP	主机名	myid的值
192.168.174.100	node01	1
192.168.174.110	node02	2
192.168.174.120	node03	3

第一步：下载zookeepeer的压缩包，下载网址如下

<http://archive.apache.org/dist/zookeeper/>

我们在这个网址下载我们使用的zk版本为3.4.9

下载完成之后，上传到我们的linux的/export/softwares路径下准备进行安装

第二步：解压

解压zookeeper的压缩包到/export/servers路径下去，然后准备进行安装

```
1 cd /export/software
2
3 tar -zxvf zookeeper-3.4.9.tar.gz -C ../servers/
```

第三步：修改配置文件

第一台机器修改配置文件

```
1 cd /export/servers/zookeeper-3.4.9/conf/
2
3 cp zoo_sample.cfg zoo.cfg
4
5 mkdir -p /export/servers/zookeeper-3.4.9/zkdatas/
```

vim zoo.cfg

```
1 dataDir=/export/servers/zookeeper-3.4.9/zkdatas
2 # 保留多少个快照
3 autopurge.snapRetainCount=3
4 # 日志多少小时清理一次
5 autopurge.purgeInterval=1
6 # 集群中服务器地址
7 server.1=node01:2888:3888
8 server.2=node02:2888:3888
9 server.3=node03:2888:3888
```

第四步：添加myid配置

在第一台机器的

/export/servers/zookeeper-3.4.9/zkdatas /这个路径下创建一个文件，文件名为myid ,文件内容为1


```
echo 1 > /export/servers/zookeeper-3.4.9/zkdatas/myid
```

第五步：安装包分发并修改myid的值

安装包分发到其他机器

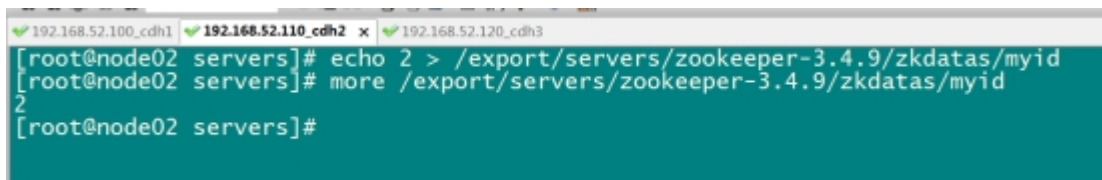
第一台机器上面执行以下两个命令

```
scp -r /export/servers/zookeeper-3.4.9/ node02:/export/servers/
```

```
scp -r /export/servers/zookeeper-3.4.9/ node03:/export/servers/
```

第二台机器上修改myid的值为2

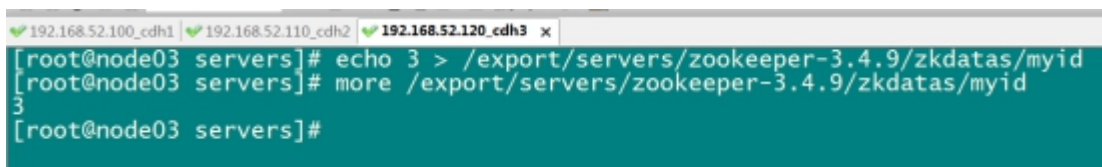
```
echo 2 > /export/servers/zookeeper-3.4.9/zkdatas/myid
```



```
[root@node02 servers]# echo 2 > /export/servers/zookeeper-3.4.9/zkdatas/myid
[root@node02 servers]# more /export/servers/zookeeper-3.4.9/zkdatas/myid
2
[root@node02 servers]#
```

第三台机器上修改myid的值为3

```
echo 3 > /export/servers/zookeeper-3.4.9/zkdatas/myid
```



```
[root@node03 servers]# echo 3 > /export/servers/zookeeper-3.4.9/zkdatas/myid
[root@node03 servers]# more /export/servers/zookeeper-3.4.9/zkdatas/myid
3
[root@node03 servers]#
```

第六步：三台机器启动zookeeper服务

三台机器启动zookeeper服务

这个命令三台机器都要执行

```
/export/servers/zookeeper-3.4.9/bin/zkServer.sh start
```

查看启动状态

```
/export/servers/zookeeper-3.4.9/bin/zkServer.sh status
```

3 Zookeeper的Shell 客户端操作

命令	说明	参数
<code>create [-s] [-e] path data acl</code>	创建Znode	-s 指定是顺序节点 -e 指定是临时节点
<code>ls path [watch]</code>	列出Path下所有子Znode	
<code>get path [watch]</code>	获取Path对应的Znode的数据和属性	
<code>ls2 path [watch]</code>	查看Path下所有子Znode以及子Znode的属性	
<code>set path data [version]</code>	更新节点	version 数据版本
<code>delete path [version]</code>	删除节点, 如果要删除的节点有子Znode则无法删除	version 数据版本
<code>rmr path</code>	删除节点, 如果有子Znode则递归删除	
<code>setquota -n -b val path</code>	修改Znode配额	-n 设置子节点最大个数 -b 设置节点数据最大长度
<code>history</code>	列出历史记录	

1: 创建普通节点

```
create /app1 hello
```

2: 创建顺序节点

```
create -s /app3 world
```

3: 创建临时节点

```
create -e /tempnode world
```

4: 创建顺序的临时节点

```
create -s -e /tempnode2 aaa
```

5:获取节点数据

```
get /app1
```

6:修改节点数据

```
set /app1 xxx
```

7:删除节点

delete /app1 删除的节点不能有子节点

rmr /app1 递归删除

Znode 的特点

- 文件系统的核心是 **Znode**
- 如果想要选取一个 **Znode** , 需要使用路径的形式, 例如 **/test1/test11**
- Znode 本身并不是文件, 也不是文件夹, Znode 因为具有一个类似于 Name 的路径, 所以可以从逻辑上实现一个树状文件系统
- ZK 保证 Znode 访问的原子性, 不会出现部分 ZK 节点更新成功, 部分 ZK 节点更新失败的问题
- **Znode** 中数据是有大小限制的, 最大只能为 **1M**
- **Znode** 是由三个部分构成
 - **stat** : 状态, Znode的权限信息, 版本等
 - **data** : 数据, 每个Znode都是可以携带数据的, 无论是否有子节点
 - **children** : 子节点列表

Znode 的类型

- 每个 **Znode** 有两大特性, 可以构成四种不同类型的 **Znode**
 - 持久性
 - **持久** 客户端断开时, 不会删除持有的Znode
 - **临时** 客户端断开时, 删除所有持有的Znode, **临时Znode不允许有子Znode**
 - 顺序性
 - **有序** 创建的Znode有先后顺序, 顺序就是在后面追加一个序列号, 序列号是由父节点管理的自增
 - **无序** 创建的Znode没有先后顺序

- **Znode** 的属性
 - **dataVersion** 数据版本, 每次当 **Znode** 中的数据发生变化的时候, **dataVersion** 都会自增一下
 - **cversion** 节点版本, 每次当 **Znode** 的节点发生变化的时候, **cversion** 都会自增
 - **aclVersion** **ACL(Access Control List)** 的版本号, 当 **Znode** 的权限信息发生变化的时候会自增
 - **zxid** 事务ID
 - **ctime** 创建时间
 - **mtime** 最近一次更新的时间
 - **ephemeralOwner** 如果 **Znode** 为临时节点, **ephemeralOwner** 表示与该节点关联的 **SessionId**

通知机制

- 通知类似于数据库中的触发器, 对某个**Znode**设置 **Watcher** , 当**Znode**发生变化的时候, **WatchManager** 会调用对应的 **Watcher**
- 当**Znode**发生删除, 修改, 创建, 子节点修改的时候, 对应的 **Watcher** 会得到通知
- **Watcher** 的特点
 - **一次性触发** 一个 **Watcher** 只会被触发一次, 如果需要继续监听, 则需要再次添加 **Watcher**
 - 事件封装: **Watcher** 得到的事件是被封装过的, 包括三个内容 **keeperState**, **eventType**, **path**

KeeperState	EventType	触发条件	说明
	None	连接成功	
SyncConnected	NodeCreated	Znode被创建	此时处于连接状态
SyncConnected	NodeDeleted	Znode被删除	此时处于连接状态
SyncConnected	NodeDataChanged	Znode数据被改变	此时处于连接状态
SyncConnected	NodeChildChanged	Znode的子Znode数据被改变	此时处于连接状态
Disconnected	None	客户端和服务端断开连接	此时客户端和服务端处于断开连接状态
Expired	None	会话超时	会收到一个SessionExpiredException
AuthFailed	None	权限验证失败	会收到一个AuthFailedException

会话

- 在ZK中所有的客户端和服务器的交互都是在某一个 `Session` 中的, 客户端和服务端创建一个连接的时候同时也会创建一个 `Session`
- `Session` 会在不同的状态之间进行切换: `CONNECTING` , `CONNECTED` , `RECONNECTING` , `RECONNECTED` , `CLOSED`
- ZK中的会话两端也需要进行心跳检测, 服务端会检测如果超过超时时间没收到客户端的心跳, 则会关闭连接, 释放资源, 关闭会话