Zookeeper

什么 Zookeeper

Zookeeper 是一个分布式开源框架，提供了协调分布式应用的基本服务，它向外部应用暴露一组通用服务——分布式同步（Distributed Synchronization）、命名服务（Naming Service）、集群维护（Group Maintenance）等，简化分布式应用协调及其管理的难度，提供高性能的分布式服务。ZooKeeper 本身可以以单机模式安装运行，不过它的长处在于通过分布式 ZooKeeper 集群（一个 Leader，多个 Follower），基于一定的策略来保证 ZooKeeper 集群的稳定性和可用性，从而实现分布式应用的可靠性。

1、zookeeper 是为别的分布式程序服务的

2、Zookeeper 本身就是一个分布式程序（只要有半数以上节点存活，zk 就能正常服务）

3、Zookeeper 所提供的服务涵盖：主从协调、服务器节点动态上下线、统一配置管理、分布式共享锁、统> 一名称服务等

4、虽然说可以提供各种服务，但是 zookeeper 在底层其实只提供了两个功能：

管理(存储，读取)用户程序提交的数据（类似 namenode 中存放的 metadata）；

并为用户程序提供数据节点监听服务；

Zookeeper 集群机制

Zookeeper 集群的角色： Leader 和 follower

只要集群中有半数以上节点存活，集群就能提供服务

Zookeeper 特性

1、Zookeeper：一个 leader，多个 follower 组成的集群

2、全局数据一致：每个 server 保存一份相同的数据副本，client 无论连接到哪个 server，数据都是一致的

3、分布式读写，更新请求转发，由 leader 实施

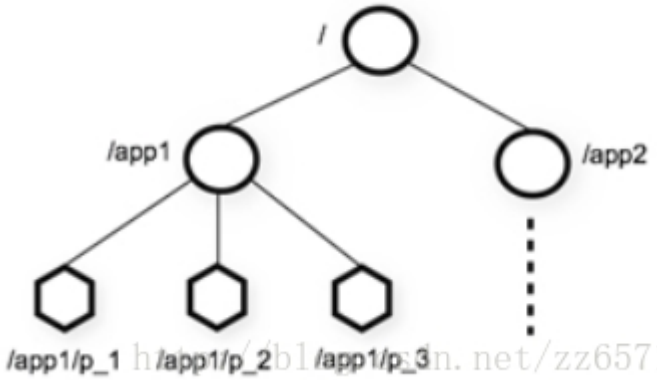
4、更新请求顺序进行，来自同一个 client 的更新请求按其发送顺序依次执行

5、数据更新原子性，一次数据更新要么成功，要么失败

6、实时性，在一定时间范围内，client 能读到最新数据

Zookeeper 数据结构

1、层次化的目录结构，命名符合常规文件系统规范(类似文件系统)



2、每个节点在 zookeeper 中叫做 znode,并且其有一个唯一的路径标识

3、节点 Znode 可以包含数据和子节点（但是 EPHEMERAL 类型的节点不能有子节点）
节点类型
a、Znode 有两种类型:
短暂（ephemeral）（create -e /app1/test1 “test1” 客户端断开连接 zk 删除 ephemeral 类型节点）
持久（persistent） （create -s /app1/test2 “test2” 客户端断开连接 zk 不删除 persistent 类型节点）
b、Znode 有四种形式的目录节点（默认是 persistent ）
PERSISTENT
PERSISTENT_SEQUENTIAL（持久序列/test0000000019 ）
EPHEMERAL
EPHEMERAL_SEQUENTIAL
c、创建 znode 时设置顺序标识，znode 名称后会附加一个值，顺序号是一个单调递增的计数器，由父节点维护

```
[zk: localhost:2181(CONNECTED) 11] ls /
[zookeeper, app1]
[zk: localhost:2181(CONNECTED) 12] create -s /app1/aa 100
Created /app1/aa0000000001
[zk: localhost:2181(CONNECTED) 13] create -s /app1/bb 100
Created /app1/bb0000000002
[zk: localhost:2181(CONNECTED) 14] create -s /app1/aa 100
Created /app1/aa0000000003
```

d、在分布式系统中，顺序号可以被用于为所有的事件进行全局排序，这样客户端可以通过顺序号推断事件的顺序

Zookeeper 应用场景

数据发布与订阅（配置中心）

发布与订阅模型，即所谓的配置中心，顾名思义就是发布者将数据发布到 ZK 节点上，供订阅者动态获取数据，实现配置信息的集中式管理和动态更新。例如全局的配置信息，服务式服务框架的服务地址列表等就非常适合使用。

负载均衡

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)，然后重新进行负载均衡，保证所有的分区都有消费者进行消费。

命名服务(Naming Service)

命名服务也是分布式系统中比较常见的一类场景。在分布式系统中，通过使用命名服务，客户端应用能够根据指定名字来获取资源或服务的地址，提供者等信息。被命名的实体通常可以是集群中的机器，提供的服务地址，远程对象等等——这些我们都可以统称他们为名字（Name）。其中较为常见的就是一些分布式服务框架中的服务地址列表。通过调用 ZK 提供的创建节点的 API，能够很容易创建一个全局唯一的 path，这个 path 就可以作为一个名称。阿里巴巴集团开源的分布式服务框架 Dubbo 中使用 ZooKeeper 来作为其命名服务，维护全局的服务地址列表， 点击这里查看 Dubbo 开源项目。在 Dubbo 实现中：

服务提供者在启动的时候，向 ZK 上的指定节点/dubbo/${serviceName}/providers 目录下写入自己的 URL 地址，这个操作就完成了服务的发布。

服务消费者启动的时候，订阅/dubbo/${serviceName}/providers 目录下的提供者 URL 地址，并向/dubbo/${serviceName} /consumers 目录下写入自己的 URL 地址。

注意，所有向 ZK 上注册的地址都是临时节点，这样就能够保证服务提供者和消费者能够自动感应资源的变化。 另外，Dubbo 还有针对服务粒度的监控，方法是订阅/dubbo/${serviceName}目录下所有提供者和消费者的信息。

分布式通知/协调

ZooKeeper 中特有 watcher 注册与异步通知机制，能够很好的实现分布式环境下不同系统之间的通知与协调，实现对数据变更的实时处理。使用方法通常是不同系统都对 ZK 上同一个 znode 进行注册，监听 znode 的变化（包括 znode 本身内容及子节点的），其中一个系统 update 了 znode，那么另一个系统能够收到通知，并作出相应处理

1. 另一种心跳检测机制：检测系统和被检测系统之间并不直接关联起来，而是通过 zk 上某个节点关联，大大减少系统耦合。
2. 另一种系统调度模式：某系统有控制台和推送系统两部分组成，控制台的职责是控制推送系统进行相应的推送工作。管理人员在控制台作的一些操作，实际上是修改了 ZK 上某些节点的状态，而 ZK 就把这些变化通知给他们注册 Watcher 的客户端，即推送系统，于是，作出相应的推送任务。
3. 另一种工作汇报模式：一些类似于任务分发系统，子任务启动后，到 zk 来注册一个临时节点，并且定时将自己的进度进行汇报（将进度写回这个临时节点），这样任务管理者就能够实时知道任务进度。

总之，使用 zookeeper 来进行分布式通知和协调能够大大降低系统之间的耦合

集群管理与 Master 选举

1. 集群机器监控：这通常用于那种对集群中机器状态，机器在线率有较高要求的场景，能够快速对集群中机器变化作出响应。这样的场景中，往往有一个监控系统，实时检测集群机器是否存活。过去的做法通常是：监控系统通过某种手段（比如 ping）定时检测每个机器，或者每个机器自己定时向监控系统汇报“我还活着”。 这种做法可行，但是存在两个比较明显的问题：

1. 集群中机器有变动的时候，牵连修改的东西比较多。
2. 有一定的延时。

利用 ZooKeeper 有两个特性，就可以实现另一种集群机器存活性监控系统：

1. 客户端在节点 x 上注册一个 Watcher，那么如果 x?的子节点变化了，会通知该客户端。
2. 创建 EPHEMERAL 类型的节点，一旦客户端和服务器的会话结束或过期，那么该节点就会消失。

例如，监控系统在 /clusterServers 节点上注册一个 Watcher，以后每动态加机器，那么就往 /clusterServers 下创建一个 EPHEMERAL 类型的节点：/clusterServers/{hostname}. 这样，监控系统就能够实时知道机器的增减情况，至于后续处理就是监控系统的业务了。

2. Master 选举则是 zookeeper 中最为经典的应用场景了。

在分布式环境中，相同的业务应用分布在不同的机器上，有些业务逻辑（例如一些耗时的计算，网络 I/O 处理），往往只需要让整个集群中的某一台机器进行执行，其余机器可以共享这个结果，这样可以大大减少重复劳动，提高性能，于是这个 master 选举便是这种场景下的碰到的主要问题。

利用 ZooKeeper 的强一致性，能够保证在分布式高并发情况下节点创建的全局唯一性，即：同时有多个客户端请求创建 /currentMaster 节点，最终一定只有一个客户端请求能够创建

成功。利用这个特性，就能很轻易的在分布式环境中进行集群选取了。
另外，这种场景演化一下，就是动态 Master 选举。这就要用到 EPHEMERAL_SEQUENTIAL 类型节点的特性了。
上文中提到，所有客户端创建请求，最终只有一个能够创建成功。在这里稍微变化下，就是允许所有请求都能够创建成功，但是得有个创建顺序，于是所有的请求最终在 ZK 上创建结果的一种可能情况是这样：/currentMaster/{sessionId}-1 ,/currentMaster/{sessionId}-2,/currentMaster/{sessionId}-3 …. 每次选取序列号最小的那个机器作为 Master，如果这个机器挂了，由于他创建的节点会马上小时，那么之后最小的那个机器就是 Master 了。
1. 在搜索系统中，如果集群中每个机器都生成一份全量索引，不仅耗时，而且不能保证彼此之间索引数据一致。因此让集群中的 Master 来进行全量索引的生成，然后同步到集群中其它机器。另外，Master 选举的容灾措施是，可以随时进行手动指定 master，就是说应用在 zk 在无法获取 master 信息时，可以通过比如 http 方式，向一个地方获取 master。
2. 在 Hbase 中，也是使用 ZooKeeper 来实现动态 HMaster 的选举。在 Hbase 实现中，会在 ZK 上存储一些 ROOT 表的地址和 HMaster 的地址，HRegionServer 也会把自己以临时节点（Ephemeral）的方式注册到 Zookeeper 中，使得 HMaster 可以随时感知到各个 HRegionServer 的存活状态，同时，一旦 HMaster 出现问题，会重新选举出一个 HMaster 来运行，从而避免了 HMaster 的单点问题

分布式锁

分布式锁，这个主要得益于 ZooKeeper 为我们保证了数据的强一致性。锁服务可以分为两类，一个是 保持独占，另一个是 控制时序。
1. 所谓保持独占，就是所有试图来获取这个锁的客户端，最终只有一个可以成功获得这把锁。通常的做法是把 zk 上的一个 znode 看作是一把锁，通过 create znode 的方式来实现。所有客户端都去创建 /distribute_lock 节点，最终成功创建的那个客户端也即拥有了这把锁。
2. 控制时序，就是所有视图来获取这个锁的客户端，最终都是会被安排执行，只是有个全局时序了。做法和上面基本类似，只是这里 /distributelock 已经预先存在，客户端在它下面创建临时有序节点（这个可以通过节点的属性控制: CreateMode.EPHEMERALSEQUENTIAL 来指定）。Zk 的父节点（/distribute_lock）维持一份 sequence，保证子节点创建的时序性，从而也形成了每个客户端的全局时序。

Zookeeper windows 环境安装

环境要求:必须要有 jdk 环境,本次讲课使用 jdk1.8

1.安装 jdk
2.安装 Zookeeper. 在官网 http://zookeeper.apache.org/下载 zookeeper.我下载的是 zookeeper-3.4.6 版本。
解压 zookeeper-3.4.6 至 D:\machine\zookeeper-3.4.6.
在 D:\machine 新建 data 及 log 目录。

3.ZooKeeper 的安装模式分为三种，分别为：单机模式（stand-alone）、集群模式和集群伪分布模式。ZooKeeper 单机模式的安装相对比较简单，如果第一次接触 ZooKeeper 的话，建议安装 ZooKeeper 单机模式或者集群伪分布模式。
安装单击模式。 至 D:\machine\zookeeper-3.4.6\conf 复制 zoo_sample.cfg 并粘贴到当前目录下，命名 zoo.cfg.

Zookeeper 集群环境搭建(linux)

环境要求:必须要有 jdk 环境,本次讲课使用 jdk1.8

结构

一共三个节点
(zk 服务器集群规模不小于 3 个节点),要求服务器之间系统时间保持一致。

上传 zk 并且解压

进行解压： tar -zxvf zookeeper-3.4.6.tar.gz
重命名： mv zookeeper-3.4.6 zookeeper

修改 zookeeper 环境变量

```
vi /etc/profile
export JAVA_HOME=/opt/jdk1.8.0_71
export ZOOKEEPER_HOME=/usr/local/zookeeper
export CLASSPATH=.:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
export PATH=$JAVA_HOME/bin:$ZOOKEEPER_HOME/bin:$PATH

source /etc/profile
```

修改 zoo_sample.cfg 文件

cd /usr/local/zookeeper/conf
mv zoo_sample.cfg zoo.cfg
修改 conf: vi zoo.cfg 修改两处
（1） dataDir=/usr/local/zookeeper/data（注意同时在 zookeeper 创建 data 目录）
（2）最后面添加
server.0=bhz:2888:3888
server.1=hadoop1:2888:3888
server.2=hadoop2:2888:3888

创建服务器标识

服务器标识配置:
创建文件夹: mkdir data
创建文件 myid 并填写内容为 0: vi myid（内容为服务器标识 : 0）

复制 zookeeper

进行复制 zookeeper 目录到 hadoop01 和 hadoop02
还有/etc/profile 文件
把 hadoop01、 hadoop02 中的 myid 文件里的值修改为 1 和 2
路径(vi /usr/local/zookeeper/data/myid)

启动 zookeeper

启动 zookeeper:
路径: /usr/local/zookeeper/bin
执行: zkServer.sh start
(注意这里 3 台机器都要进行启动)
状态: zkServer.sh status(在三个节点上检验 zk 的 mode,一个 leader 和俩个 follower)

常用命令

zkServer.sh status 查询状态

Zookeeper 配置文件介绍

```
# The number of milliseconds of each tick
tickTime=2000

# The number of ticks that the initial
# synchronization phase can take
initLimit=10

# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5

# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sakes.
dataDir=/home/myuser/zooA/data

# the port at which the clients will connect
clientPort=2181

# ZooKeeper server and its port no. # ZooKeeper ensemble should know about every other machine in the ensemble # specify server id by creating 'myid' file in the dataDir # use hostname instead of IP address for convenient maintenance
server.1=127.0.0.1:2888:3888
server.2=127.0.0.1:2988:3988
server.3=127.0.0.1:2088:3088

#
# Be sure to read the maintenance section of the
# administrator guide before turning on autopurge.
#
# http://zookeeper.apache.org/doc/current/zookeeperAdmin.html#sc_maintenance
#
# The number of snapshots to retain in dataDir
# autopurge.snapRetainCount=3
# Purge task interval in hours
# Set to "0" to disable auto purge feature <br>
#autopurge.purgeInterval=1
dataLogDir=/home/myuser/zooA/log
```

tickTime: 心跳时间，为了确保连接存在的，以毫秒为单位，最小超时时间为两个心跳时间
initLimit: 多少个心跳时间内，允许其他 server 连接并初始化数据，如果 ZooKeeper 管理的数据较大，则应相应增大这个值
clientPort: 服务的监听端口
dataDir: 用于存放内存数据库快照的文件夹，同时用于集群的 myid 文件也存在这个文件夹里（注意：一个配置文件只能包含一个 dataDir 字样，即使它被注释掉了。）
dataLogDir: 用于单独设置 transaction log 的目录，transaction log 分离可以避免和普通 log 还有快照的竞争
syncLimit: 多少个 tickTime 内，允许 follower 同步，如果 follower 落后太多，则会被丢弃。

server.A=B: C: D:
A 是一个数字,表示这个是第几号服务器,B 是这个服务器的 ip 地址
C 第一个端口用来集群成员的信息交换,表示的是这个服务器与集群中的 Leader 服务器交换信息的端口
D 是在 leader 挂掉时专门用来进行选举 leader 所用

Zookeeper 客户端

ZooKeeper 命令行工具类似于 Linux 的 shell 环境，不过功能肯定不及 shell 啦，但是使用它我们可以简单的对 ZooKeeper 进行访问，数据创建，数据修改等操作. 使用 zkCli.sh -server 127.0.0.1:2181 连接到 ZooKeeper 服务，连接成功后，系统会输出 ZooKeeper 的相关环境以及配置信息。
命令行工具的一些简单操作如下：
1. 显示根目录下、文件： ls / 使用 ls 命令来查看当前 ZooKeeper 中所包含的内容
2. 显示根目录下、文件： ls2 / 查看当前节点数据并能看到更新次数等数据
3. 创建文件，并设置初始内容： create /zk "test" 创建一个新的 znode 节点“ zk ”以及与它关联的字符串
4. 获取文件内容： get /zk 确认 znode 是否包含我们所创建的字符串
5. 修改文件内容： set /zk "zkbak" 对 zk 所关联的字符串进行设置
6. 删除文件： delete /zk 将刚才创建的 znode 删除
7. 退出客户端： quit
8. 帮助命令： help

Java 操作 Zookeeper

Zookeeper 说明

创建节点(znode)方法:

create:

提供了两套创建节点的方法，同步和异步创建节点方式。

口

同步方式:

参数 1，节点路径《名称)：InodeName (不允许递归创建节点，也就是说在父节点不存在的情况下，不允许创建子节点)

参数 2，节点内容:要求类型是字节数组(也就是说，不支持序列化方式，如果需要实现序列化，可使用 java 相关序列化框架，如 Hessian、Kryo 框架)

参数 3，节点权限:使用 Ids.OPEN_ACL_UNSAFE 开放权限即可。(这个参数一般在权展没有太高要求的场景下，没必要关注)

参数 4,节点类型:创建节点的类型:CreateMode，提供四种首点象型

PERSISTENT	持久化节点
PERSISTENT_SEQUENTIAL	顺序自动编号持久化节点，这种节点会根据当前已存在的节点数自动加 1
EPHEMERAL	临时节点，客户端 session 超时这类节点就会被自动删除
EPHEMERAL_SEQUENTIAL	临时自动编号节点

Maven 依赖信息

```
<dependency>
            <groupId>org.apache.zookeeper</groupId>
            <artifactId>zookeeper</artifactId>
            <version>3.4.6</version>
        </dependency>
```

Zookeeper 客户端连接

```
public class Test001 {


    //连接地址
    private static final String ADDRES = "127.0.0.1:2181";
    //session 会话
    private static final int SESSION_OUTTIME = 2000;
    //信号量,阻塞程序执行,用户等待zookeeper连接成功,发送成功信号,
```

```
    private static final CountDownLatch countDownLatch = new CountDownLatch(1);


    public static void main(String[] args) throws IOException, InterruptedException, KeeperException {
        ZooKeeper zk = new ZooKeeper(ADDRES, SESSION_OUTTIME, new Watcher() {


            public void process(WatchedEvent event) {
                // 获取事件状态
                KeeperState keeperState = event.getState();
                // 获取事件类型
                EventType eventType = event.getType();
                if (KeeperState.SyncConnected == keeperState) {
                    if (EventType.None == eventType) {
                        countDownLatch.countDown();
                        System.out.println("zk 启动连接...");
                    }


                }
            }
        });
        // 进行阻塞
        countDownLatch.await();
        String result = zk.create("/itmayeidu_Lasting", "Lasting".getBytes(), Ids.OPEN_ACL_UNSAFE,
                CreateMode.PERSISTENT);
        System.out.println(result);
        zk.close();
    }
}
```


创建 Zookeeper 节点信息

1. **创建持久节点，并且允许任何服务器可以操作**

```
    String result = zk.create("/itmayiedu_Lasting", "Lasting".getBytes(), Ids.OPEN_ACL_UNSAFE,
CreateMode.PERSISTENT);
    System.out.println("result:" + result);
```

2. **创建临时节点**

```
    String result = zk.create("/itmayiedu_temp", "temp".getBytes(), Ids.OPEN_ACL_UNSAFE, CreateMode.EPHEMERAL);
    System.out.println("result:" + result);
```

Watcher

在 ZooKeeper 中，接口类 Watcher 用于表示一个标准的事件处理器，其定义了事件通知相关的逻辑，包含 KeeperState 和 EventType 两个枚举类，分别代表了通知状态和事件类型，同时定义了事件的回调方法：process（WatchedEvent event）。

什么是 Watcher 接口

同一个事件类型在不同的通知状态中代表的含义有所不同，表 7-3 列举了常见的通知状态和事件类型。

表 7-3 Watcher 通知状态与事件类型一览

KeeperState	EventType	触发条件	说明
	None（-1）	客户端与服务端成功建立连接	
SyncConnected（0）	NodeCreated（1）	Watcher 监听的对应数据节点被创建	
	NodeDeleted（2）	Watcher 监听的对应数据节点被删除	此时客户端和服务器处于连
	NodeDataChanged（3）	Watcher 监听的对应数据节点的数据内容发生变更	
	NodeChildChanged（4）	Wather 监听的对应数据节点的子节点列表发生变更	
Disconnected（0）	None（-1）	客户端与 ZooKeeper 服务器断开连接	此时客户端和服务器处于断
Expired（-112）	Node（-1）	会话超时	此时客户端会话失效，通常 SessionExpiredException
AuthFailed（4）	None（-1）	通常有两种情况，1：使用错误的 schema 进行权限检查 2：SASL 权限检查失败	通常同时也会收到 AuthFail

表 7-3 中列举了 ZooKeeper 中最常见的几个通知状态和事件类型。

回调方法 process（）

process 方法是 Watcher 接口中的一个回调方法，当 ZooKeeper 向客户端发送一个 Watcher 事件通知时，客户端就会对相应的 process 方法进行回调，从而实现对事件的处理。process 方法的定义如下：

```
abstract public void process(WatchedEvent event);
```

这个回调方法的定义非常简单，我们重点看下方法的参数定义：WatchedEvent。

WatchedEvent 包含了每一个事件的三个基本属性：通知状态（keeperState），事件类型（EventType）和节点路径(path)，其数据结构如图 7-5 所示。ZooKeeper 使用 WatchedEvent 对象来封装服务端事件并传递给 Watcher，从而方便回调方法 process 对服务端事件进行处理。

提到 WatchedEvent，不得不讲下 WatcherEvent 实体。笼统地讲，两者表示的是同一个事物，都是对一个服务端事件的封装。不同的是，WatchedEvent 是一个逻辑事件，用于服务端和客户端程序执行过程中所需的逻辑对象，而 WatcherEvent 因为实现了序列化接口，因此可以用于网络传输。

服务端在生成 WatchedEvent 事件之后，会调用 getWrapper 方法将自己包装成一个可序列化的 WatcherEvent 事件，以便通过网络传输到客户端。客户端在接收到服务端的这个事件对象后，首先会将 WatcherEvent 还原成一个 WatchedEvent 事件，并传递给 process 方法处理，回调方法 process 根据入参就能够解析出完整的服务端事件了。

需要注意的一点是，无论是 WatchedEvent 还是 WatcherEvent，其对 ZooKeeper 服务端事件的封装都是机及其简单的。举个例子来说，当/zk-book 这个节点的数据发生变更时，服务端会发送给客户端一个“ZNode 数据内容变更”事件，客户端只能够接收到如下信

Watcher 代码

```
public class ZkClientWatcher implements Watcher {
	// 集群连接地址
	private static final String CONNECT_ADDRES = "192.168.110.159:2181,192.168.110.160:2181,192.168.110.162:2181";
	// 会话超时时间
	private static final int SESSIONTIME = 2000;
	// 信号量,让zk在连接之前等待,连接成功后才能往下走.
	private static final CountDownLatch countDownLatch = new CountDownLatch(1);
	private static String LOG_MAIN = "【main】 ";
	private ZooKeeper zk;

	public void createConnection(String connectAddress, int sessionTimeOut) {
		try {
			zk = new ZooKeeper(connectAddress, sessionTimeOut, this);
			System.out.println(LOG_MAIN + "zk 开始启动连接服务器....");
			countDownLatch.await();
		} catch (Exception e) {
			e.printStackTrace();
		}
	}

	public boolean createPath(String path, String data) {
		try {
			this.exists(path, true);
			this.zk.create(path, data.getBytes(), Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
			System.out.println(LOG_MAIN + "节点创建成功, Path:" + path + ",data:" + data);
		} catch (Exception e) {
			e.printStackTrace();
			return false;
```

```
        }
        return true;
    }

    /**
     * 判断指定节点是否存在
     *
     * @param path
     *            节点路径
     */
    public Stat exists(String path, boolean needWatch) {
        try {
            return this.zk.exists(path, needWatch);
        } catch (Exception e) {
            e.printStackTrace();
            return null;
        }
    }

    public boolean updateNode(String path,String data) throws KeeperException, InterruptedException {
        exists(path, true);
        this.zk.setData(path, data.getBytes(), -1);
        return false;
    }

    public void process(WatchedEvent watchedEvent) {
        // 获取事件状态
        KeeperState keeperState = watchedEvent.getState();
        // 获取事件类型
        EventType eventType = watchedEvent.getType();
        // 监听路径
        String path = watchedEvent.getPath();
        System.out.println("进入到 process() keeperState:" + keeperState + ", eventType:" + eventType + ", path:" + path
        // 判断是否建立连接
        if (KeeperState.SyncConnected == keeperState) {
            if (EventType.None == eventType) {
                // 如果建立建立成功,让后程序往下走
                System.out.println(LOG_MAIN + "zk 建立连接成功!");
                countDownLatch.countDown();
            } else if (EventType.NodeCreated == eventType) {
                System.out.println(LOG_MAIN + "事件通知,新增node节点" + path);
            } else if (EventType.NodeDataChanged == eventType) {
                System.out.println(LOG_MAIN + "事件通知,当前node节点" + path + "被修改....");
```

```
                }
                else if (EventType.NodeDeleted == eventType) {
                        System.out.println(LOG_MAIN + "事件通知,当前node节点" + path + "被删除....");
                }

            }
            System.out.println("----------------------------------------------------------");
        }

        public static void main(String[] args) throws KeeperException, InterruptedException {
            ZkClientWatcher zkClientWatcher = new ZkClientWatcher();
            zkClientWatcher.createConnection(CONNECT_ADDRES, SESSIONTIME);
//          boolean createResult = zkClientWatcher.createPath("/p15", "pa-644064");
            zkClientWatcher.updateNode("/pa2","7894561");
        }

}
```