1、做服务注册与发现的时候用零时顺序节点的原因：a.临时节点是因为当客户端与zk会话断开以后会自动删除； b.顺序是因为节点名称是服务名，同一个节点下不能放多个同名的子节点，ip和port不是节点名称，是存在节点中的数据，没有顺序子节点就会不唯一，服务就没法注册。

2、每次获取节点内容后在册注册监听

```
}

private void updateServiceList() {
    try{
        List<String> children = zooKeeper.getChildren( path: BASE_SERVICES  + SERVICE_NAME, watch: true);
        List<String> newServerList = new ArrayList<String>();
        for(String subNode:children) {
            byte[] data = zooKeeper.getData( path: BASE_SERVICES  + SERVICE_NAME + "/" + subNode, watch: false, stat: null);
            String host = new String(data, charsetName: "utf-8");
            System.out.println("host:"+host);
            newServerList.add(host);
        }
        LoadBalance.SERVICE_LIST = newServerList;
    }catch (Exception e) {
        e.printStackTrace();
    }
}

@Override
public void contextDestroyed(ServletContextEvent sce) {
```

3、zk原生java api：

## 1、zookeeper原生Java API

Zookeeper客户端提供了基本的操作，比如，创建会话、创建节点、读取节点、更新数据、删除节点和检查节点是否存在等。但对于开发人员来说，Zookeeper提供的基本操纵还是有一些不足之处。

**Zookeeper API不足之处**

- (1) Watcher注册是一次性的，每次触发之后都需要重新进行注册；
- (2) Session超时之后没有实现重连机制；
- (3) 异常处理繁琐，Zookeeper提供了很多异常，对于开发人员来说可能根本不知道该如何处理这些异常信息；
- (4) 只提供了简单的byte[]数组的接口，没有提供针对对象级别的序列化；
- (5) 创建节点时如果节点存在抛出异常，需要自行检查节点是否存在；
- (6) 删除节点无法实现级联删除；

基于以上原因，直接使用Zookeeper原生API的人并不多。

4、zkClient：

## 2、ZkClient

```xml
<!-- https://mvnrepository.com/artifact/com.101tec/zkclient -->
<dependency>
    <groupId>com.101tec</groupId>
    <artifactId>zkclient</artifactId>
    <version>0.10</version>
</dependency>
```

ZkClient是一个开源客户端，在Zookeeper原生API接口的基础上进行了包装，更便于开发人员使用。解决如下问题：

- 1）session会话超时重连
- 2）解决Watcher反复注册
- 3）简化API开发

虽然 ZkClient 对原生 API 进行了封装，但也有它自身的不足之处：

- 几乎没有参考文档；
- 异常处理简化（抛出RuntimeException）；
- 重试机制比较难用；
- 没有提供各种使用场景的实现；

## 3、Apache Curator

5、Curator：

## 3、Apache Curator

相关依赖 参考文章： https://blog.csdn.net/xiaojin21cen/article/details/88538102

> Curator是Netflix公司开源的一套Zookeeper客户端框架，和ZkClient一样，解决了非常底层的细节开发工作，包括连接重连、反复注册Watcher和NodeExistsException异常等。目前已经成为 Apache 的顶级项目。

其特点：

1. Apache 的开源项目
2. 解决Watch注册一次就会失效的问题
3. 提供一套Fluent风格的 API 更加简单易用
4. 提供更多解决方案并且实现简单，例如：分布式锁
5. 提供常用的ZooKeeper工具类
6. 编程风格更舒服

除此之外，Curator中还提供了Zookeeper各种应用场景（Recipe，如共享锁服务、Master选举机制和分布式计算器等）的抽象封装。

6、curator创建多级节点（例如/services/product，当services节点不存在的时候不加**creatingParentsIfNeeded()**会报错）的时候需如图中这样处理：

```
String path = BASE_SERVICES + serverName;
if(curatorZkClient.checkExists().forPath(path) == null){
    curatorZkClient.create().creatingParentsIfNeeded().withMode(CreateMode.PERSISTENT).forPath(path,serverName.getBytes());
}
String data = InetAddress.getLocalHost().getHostAddress()+":"+port;
```

7、获取子节点返回的是节点名称，不是路径

```
public void updateServerMap() throws Exception{
    List<String> serverPath = curatorZkClient.getChildren().forPath(BASE_SERVICES);
    for(String path : serverPath){
        String serverName = new String(curatorZkClient.getData().forPath(BASE_SERVICES + "/" + path), charsetName: "UTF-8
        List<String> serverChild = curatorZkClient.getChildren().forPath(BASE_SERVICES + "/" + path);
        List<String> childList = new ArrayList<>(serverChild.size());
        for(String child : serverChild){
            byte[] bytes = curatorZkClient.getData().forPath(BASE_SERVICES + "/" + path + "/" + child);
            String data = new String(bytes, charsetName: "UTF-8");
            childList.add(data);
        截图(Alt + A)
        RamdomLoadBalances.SERVER_MAP.put(serverName,childList);
    }
}
/*
```

假如zk中有节点目录如下/services/product/product01，当我们获取services的子节点时会返回一个list，内容为product（而不是/services/product）。我在编写代码的时候把变量名定义为了path，所以将节点名和节点路径混淆了，以后要注意。

8、监听必须start()，不然不会生效：

```
curatorZkClient.start();
try {
    Stat state = curatorZkClient.checkExists().forPath(BASE_SERVICES);
    if(state == null){
        curatorZkClient.create().withMode(CreateMode.PERSISTENT).forPath(BASE_SERVICES);
    }
    TreeCache treeCache = new TreeCache(curatorZkClient,BASE_SERVICES);
    treeCache.getListenable().addListener(chiledListener);
    treeCache.start();
    updateServerMap();
} catch (Exception e) {
    e.printStackTrace();
}
}
```

9、Curator三种监听**NodeCache、PathChildrenCache、TreeCache**对比（https://blog.csdn.net/zifanyou/article/details/84883873）：

## 1. NodeCache

这是最简单的一种监听方式，只监听固定节点的内容变化。

## 2. PathChildrenCache

此种方式只监听某个节点的子节点变化，不关心孙子节点的变化。

## 3. TreeCache

此种方式已树形的方式监听某个节点下所有子孙节点的变化，例如子节点的子节点。

10、zk实现分布式锁***集群规模比较大***的情况下要注意***羊群效应：***

## 集群中的羊群效应

分布式中的羊群效应直白理解，生活中的羊群效应：

如果有接触过羊群的人可能会发现，当你在放羊的时候，如果某一羊犯错吃了农场的粮食，你可能会丢一块石头去驱逐该羊不要吃粮食，但是丢石头的时候，该羊被石头恐吓的同时也会惊慌其他羊，这也就是所谓的直白羊群意思。

但是在分布式系统中，例如Zokeeper集群中，例如某一节点A被大量client进行watch时，当节点A发生变化时候可能只对某一个客户端有影响，但是由于所有客户端都对该节点进行了watch，所以对于其他没有影响的client也会受到通知，这种不必要的通知就是分布式中的羊群效应。

标签：zookeeper 羊群效应

**羊群效应**

有一个问题需要注意，当变化发生时，ZooKeeper会触发一个特定的znode节点的变化导致的所有监视点的集合。如果有1000个客户端通过exists操作监视这个znode节点，那么当znode节点创建后就会发送1000个通知，因而被监视的znode节点的一个变化会产生一个尖峰的通知，该尖峰可能带来影响，例如，在尖峰时刻提交的操作延迟。可能的话，我们建议在使用ZooKeeper时，避免在一个特定节点设置大量的监视点，最好是每次在特定的znode节点上，只有少量的客户端设置监视点，理想情况下最多只设置一个。

解决该问题的方法并不适用于所有的情况，但在以下情况下可能很有用。假设有n个客户端争相获取一个锁（例如，主节点锁）。为了获取锁，一个进程试着创建/lock节点，如果znode节点存在了，客户端就会监视这个znode节点的删除事件。当/lock被删除时，所有监视/lock节点的客户端收到通知。另一个不同的方法，让客户端创建一个有序的节点/lock/lock-，回忆之前讨论的有序节点，ZooKeeper在这个znode节点上自动添加一个序列号，成为/lock/lock-xxx，其中xxx为序列号。我们可以使用这个序列号来确定哪个客户端获得锁，通过判断/lock下的所有创建的子节点的最小序列号。在该方案中，客户端通过/getChildren方法来获取所有/lock下的子节点，并判断自己创建的节点是否是最小的序列号。如果客户端创建的节点不是最小序列号，就根据序列号确定序列，并在前一个节点上设置监视点。例如：假设我们有三个节点：/lock/lock-001、/lock/lock-002和/lock/lock-003，在这个例子中情况如下：

11、zk实现分布式锁原理每个节点监听前置节点的变化：

```
public void waitLock() {
    IZkDataListener listener = new IZkDataListener() {

        public void handleDataDeleted(String dataPath) throws Exception {

            if(countDownLatch!=null){
                countDownLatch.countDown();
            }
        }

        public void handleDataChange(String dataPath, Object data) throws Exception {

        }
    };
    //给排在前面的的节点增加数据删除的watcher,本质是启动另外一个线程去监听前置节点
    this.zkClient.subscribeDataChanges(beforePath, listener);

    if(this.zkClient.exists(beforePath)){
        countDownLatch=new CountDownLatch(1);
        try {
            countDownLatch.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
    this.zkClient.unsubscribeDataChanges(beforePath, listener);
}
```

当前置节点发生变化以后当前线程继续执行再次去获取锁

注册监听前置znode

如果前置节点存在则阻塞当前线程，等待获取锁

12、Zk三种客户端没提供池化技术，不过Curator提供了start()、close()方法。我们也可以单例或者交给spring容器。但一般代码中对于zk的操作不是很频繁的话可以多例，通过new多个或者spring容器scope改成多例。

13、zookeeper的过半选举机制保证了不会出现脑裂现象
（https://www.jianshu.com/p/234a26ab1693）

14、zk满足了CAP理论中的CP

**15、ZK动态扩容分两种情况：**

- **如果原来是单机模式，则在扩容的时候，短暂的停止服务是不可避免的。因为修改了配置文件，需要将原机器进行重启，而其他系统都依赖于此单机zookeeper服务。在扩容的时候，我们需要先将扩容的机器配置部署完成，在最后阶段，修改原机器上的配置文件后对服务进行重启。这个时候就会出现短暂的停止服务。**
- **如果已经是集群模式，则扩容的时候先部署新机器，再重启老机器。在重启的过程中，需要保证一台机器重启完成后，再进行下一台机器的重启。这样就整个集群中每个时刻只有一台机器不能正常工作，而集群中有过半的机器是正常工作的，那么整个集群对外就是可用的。所以这个时候不会出现错误，也不会出现停止服务，整个扩容过程对用户是无感知的。**

**----------ZK复习----------**

**1、什么是ZK：Zookeeper致力于提供一个高性能、高可用、且具备严格的顺序访问控制能力的分布式协调服务。**

**2、zk数据结构：共享的树形结构，类似于文件系统，存储于内存。**

**3、高可用：可以集群部署，最好部署奇数个，超过半数正常工作就能对外提供服务。**

**4、顺序访问：对于每个请求，zk会分配一个全局唯一的递增编号，利用这个特性可以实现高级协调服务。**

**5、会话：会话（session）是zk非常重要的概念，客户端和服务端之间的任何操作都与会话有关，本质是TCP长连接，通过会话可以进行心跳检测和数据传输。**

6、zk数据模型：zk的视图结构和标准的Unix文件系统类似，其中每个节点称为 “数据节点” 或者 ZNode，每个ZNode不但可以存储数据还可以挂载子节点，因此可以称之为“树”。需要注意的是创建ZNode的时候都必须带值，否则节点会创建失败。

7、zk客户端常用命令：ls 、get、set、create、delete、rmr（递归删除）。

8、ACL权限：zk的ACL（访问控制列表）分为三个维度：scheme、id、permission，通常表示为【scheme:id:permission】：

- scheme代表授权策略，它有以下值：
  - world：默认方式，相当于全世界都能访问。
  - auth：代表已经认证通过的用户(通过addauth digest user:pwd 来添加授权用户)
  - digest：即用户名:密码这种方式认证，这也是业务系统中最常用的
  - ip：使用Ip地址认证
- id是验证模式，不同的scheme，id值也不一样：
  - scheme为auth时，id的值为username:password
  - scheme为digest时，id的值为username:BASE64(SHA1(password))
  - scheme为ip时，id的值为客户端的ip地址
  - scheme为world时，id的值为anyone
- permission代表权限，有crwda五种权限：
  - CREATE(c)：创建子节点的权限
  - DELETE(d)：删除节点的权限
  - READ(r)：读取节点数据的权限
  - WRITE(w)：修改节点数据的权限
  - ADMIN(a)：设置子节点权限的权限

9、zk节点类型（同一个Znode下节点的名称是唯一的）：

- 持久节点；

- 持久顺序节点；
- 临时节点；
- 临时顺序节点；

10、CAP理论（https://www.cnblogs.com/mingorun/p/11025538.html）：

- 分区容错性：指的分布式系统中的某个节点或者网络分区出现了故障的时候，整个系统仍然能对外提供满足一致性和可用性的服务。也就是说部分故障不影响整体使用。

- 可用性： 系统能正常相应客户端的请求。

- 一致性：要求分布式系统中的各节点时时刻刻保持数据的一致性。

11、CAP协议三者不可能同时满足。但分区容错性是必须要保证的，如果放弃分区容错性则意味着放弃了系统的扩展性，系统不再是分布式的，这有违分布式设计的初衷。工程师需要做的就是尽量在A、C之间找到一个平衡。

12、Base理论的核心思想是既然无法做到强一致性，那可以结合业务自身特点采用适当的方式来使系统达到最终一致性，BASE理论内容如下：

- 基本可用：当系统实现故障时，还可以对外提供服务，相对于正常的系统而言可能响应时间变长了，部分功能不可用了（服务降级）。
- 软状态：软状态是指可以允许系统中的数据存在中间状态，并认为该状态不影响系统的整体可用性，即允许系统的不同节点之间存在短暂的数据不一致。
- 最终一致性：最终一致性是说软状态是必须要有期限的，在期限过后，应当保证所有节点的数据副本是一致的，从而达到数据的最终一致性。

13、zab协议用来解决分布式系统的数据一致性问题。zk中只有leader处理客户端的事务请求（写请求）。

14、zab协议两种模式：

- 消息广播模式（数据副本的传递采用了消息广播模式）：zookeeper中数据副本的同步方式与二段提交（2pc）相似，但是却又不同。二段提交要求协调者必须等到所有的参与者全部反馈ACK确认消息后，再发送commit消息。而zk中只需半数以上follower成功反馈即可。消息广播具体步骤如下：
    - 客户端发起一个写请求；
    - Leader服务器会将客户端请求转化为事务提案（proposal），同时为每个proposal分配一个全局唯一的ID，即ZXID。
    - Leader服务器与每个follower服务器之间都有一个FIFO队列，leader将proposal发送到改队列中。
    - follower从队列中获取到消息并处理完成后（写入本地事务日志中）向leader服务器发送成功反馈。
    - leader收到半数以上follower的成功反馈后，紧接着就会向所有的follower发送commit消息。

- 奔溃恢复（选举的时候使用奔溃恢复模式，新选举的leader节点中含有最大的ZXID，这样做的好处就是可以避免leader服务器检查提案进行提交和丢弃工作，同时也是为了保证集群中的数据是最新的）：
    - 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。
    - 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。
    - 处理投票。针对每一个投票，服务器都需要将收到的投票和自己的投票进行PK，PK规则（1、优先检查ZXID。ZXID比较大的服务器优先作为Leader。2、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。**

- **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。**
- **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。**