主讲老师：Fox

课前须知：
本节课需要zookeeper基础知识，新来的同学如果没有用过zookeeper建议从第一节开始听

```
文档：2. Zookeeper客户端使用与经典应用场景...
链接：http://note.youdao.com/noteshare?id=81cd2c0ecf8f018f1a6fd515e6402f5b&sub=35AA5B756CB346D1A826908C35BB7CC3
```


Zookeeper整合Java实战

ZooKeeper应用的开发主要通过Java客户端API去连接和操作ZooKeeper集群。可供选择的Java客户端API有：

- ZooKeeper官方的Java客户端API。
- 第三方的Java客户端API，比如Curator。

ZooKeeper官方的客户端API提供了基本的操作。例如，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。不过，对于实际开发来说，ZooKeeper官方API有一些不足之处，具体如下：

- ZooKeeper的Watcher监测是一次性的，每次触发之后都需要重新进行注册。·会话超时之后没有实现重连机制。
- 异常处理烦琐，ZooKeeper提供了很多异常，对于开发人员来说可能根本不知道应该如何处理这些抛出的异常。
- 仅提供了简单的byte[]数组类型的接口，没有提供Java POJO级别的序列化数据处理接口。
- 创建节点时如果抛出异常，需要自行检查节点是否存在。
- 无法实现级联删除。

总之，ZooKeeper官方API功能比较简单，在实际开发过程中比较笨重，一般不推荐使用。

Zookeeper 原生Java客户端使用

引入zookeeper client依赖

```
<!-- zookeeper client -->
<dependency>
 <groupId>org.apache.zookeeper</groupId>
 <artifactId>zookeeper</artifactId>
 <version>3.8.0</version>
</dependency>
```

注意：保持与服务端版本一致，不然会有很多兼容性的问题

ZooKeeper原生客户端主要使用org.apache.zookeeper.ZooKeeper这个类来使用ZooKeeper服务。

ZooKeeper常用构造器

```
ZooKeeper (connectString, sessionTimeout, watcher)
```

- connectString:使用逗号分隔的列表，每个ZooKeeper节点是一个host.port对，host 是机器名或者IP地址，port是ZooKeeper节点对客户端提供服务的端口号。客户端会任意选取connectString 中的一个节点建立连接。
- sessionTimeout : session timeout时间。
- watcher:用于接收到来自ZooKeeper集群的事件。

使用 zookeeper 原生 API,连接zookeeper集群

```
public class ZkClientDemo {

 private static final String CONNECT_STR="localhost:2181";
 private final static String CLUSTER_CONNECT_STR="192.168.65.156:2181,19
2.168.65.190:2181,192.168.65.200:2181";

 public static void main(String[] args) throws Exception {

 final CountDownLatch countDownLatch=new CountDownLatch(1);
 ZooKeeper zooKeeper = new ZooKeeper(CLUSTER_CONNECT_STR,
  4000, new Watcher() {
  @Override
  public void process(WatchedEvent event) {
  if(Event.KeeperState.SyncConnected==event.getState()
  && event.getType()== Event.EventType.None){
```

```
15  //如果收到了服务端的响应事件，连接成功
16  countDownLatch.countDown();
17  System.out.println("连接建立");
18  }
19  }
20  });
21  System.out.printf("连接中");
22  countDownLatch.await();
23  //CONNECTED
24  System.out.println(zooKeeper.getState());
25
26  //创建持久节点
27  zooKeeper.create("/user","fox".getBytes(),
28  ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
29
30  }
31
32 }
```

Zookeeper主要方法

- create(path, data, acl,createMode): 创建一个给定路径的 znode，并在 znode 保存 data[]的 数据，createMode指定 znode 的类型。
- delete(path, version):如果给定 path 上的 znode 的版本和给定的 version 匹配， 删除 znode。
- exists(path, watch):判断给定 path 上的 znode 是否存在，并在 znode 设置一个 watch。
- getData(path, watch):返回给定 path 上的 znode 数据，并在 znode 设置一个 watch。
- setData(path, data, version):如果给定 path 上的 znode 的版本和给定的 version 匹配，设置 znode 数据。
- getChildren(path, watch):返回给定 path 上的 znode 的孩子 znode 名字，并在 znode 设置一个 watch。
- sync(path):把客户端 session 连接节点和 leader 节点进行同步。

方法特点:

- 所有获取 znode 数据的 API 都可以设置一个 watch 用来监控 znode 的变化。
- 所有更新 znode 数据的 API 都有两个版本: 无条件更新版本和条件更新版本。

如果 version 为 -1，更新为无条件更新。否则只有给定的 version 和 znode 当前的

version 一样，才会进行更新，这样的更新是条件更新。

- 所有的方法都有同步和异步两个版本。同步版本的方法发送请求给 ZooKeeper 并等待服务器的响 应。异步版本把请求放入客户端的请求队列，然后马上返回。异步版本通过 callback 来接受来 自服务端的响应。

同步创建节点：

```
@Test
public void createTest() throws KeeperException, InterruptedException {
 String path = zooKeeper.create(ZK_NODE, "data".getBytes(), ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
 log.info("created path: {}",path);
}
```

异步创建节点：

```
@Test
public void createAsycTest() throws InterruptedException {
 zooKeeper.create(ZK_NODE, "data".getBytes(),
ZooDefs.Ids.OPEN_ACL_UNSAFE,
 CreateMode.PERSISTENT,
 (rc, path, ctx, name) -> log.info("rc {},path {},ctx {},name {}",rc,path,ctx,name),"context");
 TimeUnit.SECONDS.sleep(Integer.MAX_VALUE);
}
```

修改节点数据

```
@Test
public void setTest() throws KeeperException, InterruptedException {

 Stat stat = new Stat();
 byte[] data = zooKeeper.getData(ZK_NODE, false, stat);
 log.info("修改前: {}",new String(data));
 zooKeeper.setData(ZK_NODE, "changed!".getBytes(), stat.getVersion());
 byte[] dataAfter = zooKeeper.getData(ZK_NODE, false, stat);
 log.info("修改后: {}",new String(dataAfter));
}
```

Curator开源客户端使用

Curator是Netflix公司开源的一套ZooKeeper客户端框架，和ZkClient一样它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的问题以及NodeExistsException异常等。

Curator是Apache基金会的顶级项目之一，Curator具有更加完善的文档，另外还提供了一套易用性和可读性更强的Fluent风格的客户端API框架。

Curator还为ZooKeeper客户端框架提供了一些比较普遍的、开箱即用的、分布式开发用的解决方案，例如Recipe、共享锁服务、Master选举机制和分布式计算器等，帮助开发者避免了“重复造轮子”的无效开发工作。

> Guava is to Java that Curator to ZooKeeper

在实际的开发场景中，使用Curator客户端就足以应付日常的ZooKeeper集群操作的需求。

官网：https://curator.apache.org/

引入依赖

Curator 包含了几个包：

- curator-framework是对ZooKeeper的底层API的一些封装。
- curator-client提供了一些客户端的操作，例如重试策略等。
- curator-recipes封装了一些高级特性，如：Cache事件监听、选举、分布式锁、分布式计数器、分布式Barrier等。

```
<!-- zookeeper client -->
<dependency>
 <groupId>org.apache.zookeeper</groupId>
 <artifactId>zookeeper</artifactId>
 <version>3.8.0</version>
</dependency>

<!--curator-->
<dependency>
  <groupId>org.apache.curator</groupId>
  <artifactId>curator-recipes</artifactId>
  <version>5.1.0</version>
  <exclusions>
  <exclusion>
  <groupId>org.apache.zookeeper</groupId>
  <artifactId>zookeeper</artifactId>
  </exclusion>
  </exclusions>
 </dependency>
```

创建一个客户端实例

在使用curator-framework包操作ZooKeeper前，首先要创建一个客户端实例。这是一个CuratorFramework类型的对象，有两种方法：

- 使用工厂类CuratorFrameworkFactory的静态newClient()方法。

```
// 重试策略
RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000, 3)
//创建客户端实例
CuratorFramework client = CuratorFrameworkFactory.newClient(zookeeperConnectionString, retryPolicy);
//启动客户端
client.start();
```

- 使用工厂类CuratorFrameworkFactory的静态builder构造者方法。

```
RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000, 3);
CuratorFramework client = CuratorFrameworkFactory.builder()
  .connectString("192.168.128.129:2181")
  .sessionTimeoutMs(5000) // 会话超时时间
  .connectionTimeoutMs(5000) // 连接超时时间
  .retryPolicy(retryPolicy)
  .namespace("base") // 包含隔离名称
  .build();
client.start();
```

- connectionString：服务器地址列表，在指定服务器地址列表的时候可以是一个地址，也可以是多个地址。如果是多个地址，那么每个服务器地址列表用逗号分隔, 如 host1:port1,host2:port2,host3；port3 。

- retryPolicy：重试策略，当客户端异常退出或者与服务端失去连接的时候，可以通过设置客户端重新连接 ZooKeeper 服务端。而 Curator 提供了 一次重试、多次重试等不同种类的实现方式。在 Curator 内部，可以通过判断服务器返回的 keeperException 的状态代码来判断是否进行重试处理，如果返回的是 OK 表示一切操作都没有问题，而 SYSTEMERROR 表示系统或服务端错误。

策略名称	描述
ExponentialBackoffRetry	重试一组次数，重试之间的睡眠时间增加
RetryNTimes	重试最大次数
RetryOneTime	只重试一次
RetryUntilElapsed	在给定的时间结束之前重试

- 超时时间：Curator 客户端创建过程中，有两个超时时间的设置。一个是 sessionTimeoutMs 会话超时时间，用来设置该条会话在 ZooKeeper 服务端的失效时间。另一个是 connectionTimeoutMs 客户端创建会话的超时时间，用来限制客户端发起一个会话连接到接收 ZooKeeper 服务端应答的时间。sessionTimeoutMs 作用在服务端，而 connectionTimeoutMs 作用在客户端。

创建节点

创建节点的方式如下面的代码所示，回顾我们之前课程中讲到的内容，描述一个节点要包括节点的类型，即临时节点还是持久节点、节点的数据信息、节点是否是有序节点等属性和性质。

```
@Test
public void testCreate() throws Exception {
 String path = curatorFramework.create().forPath("/curator-node");
 curatorFramework.create().withMode(CreateMode.PERSISTENT).forPath("/curator-node","some-data".getBytes())
 log.info("curator create node :{} successfully.",path);
}
```

在 Curator 中，可以使用 create 函数创建数据节点，并通过 withMode 函数指定节点类型（持久化节点，临时节点，顺序节点，临时顺序节点，持久化顺序节点等），默认是持久化节点，之后调用 forPath 函数来指定节点的路径和数据信息。

一次性创建带层级结构的节点

```
@Test
public void testCreateWithParent() throws Exception {
 String pathWithParent="/node-parent/sub-node-1";
 String path = curatorFramework.create().creatingParentsIfNeeded().forPath(pathWithParent);
 log.info("curator create node :{} successfully.",path);
}
```

获取数据

```
@Test
public void testGetData() throws Exception {
```

```
3   byte[] bytes = curatorFramework.getData().forPath("/curator-node");
4   log.info("get data from node :{} successfully.",new String(bytes));
5  }
```

更新节点

我们通过客户端实例的 setData() 方法更新 ZooKeeper 服务上的数据节点，在setData 方法的后边，通过 forPath 函数来指定更新的数据节点路径以及要更新的数据。

```
1  @Test
2  public void testSetData() throws Exception {
3   curatorFramework.setData().forPath("/curator-
node","changed!".getBytes());
4   byte[] bytes = curatorFramework.getData().forPath("/curator-node");
5   log.info("get data from node /curator-node :{} successfully.",new
String(bytes));
6  }
```

删除节点

```
1  @Test
2  public void testDelete() throws Exception {
3   String pathWithParent="/node-parent";
4   curatorFramework.delete().guaranteed().deletingChildrenIfNeeded().forPat
h(pathWithParent);
5  }
```

guaranteed：该函数的功能如字面意思一样，主要起到一个保障删除成功的作用，其底层工作方式是：只要该客户端的会话有效，就会在后台持续发起删除请求，直到该数据节点在 ZooKeeper 服务端被删除。

deletingChildrenIfNeeded：指定了该函数后，系统在删除该数据节点的时候会以递归的方式直接删除其子节点，以及子节点的子节点。

异步接口

Curator 引入了BackgroundCallback 接口，用来处理服务器端返回来的信息，这个处理过程是在异步线程中调用，默认在 **EventThread** 中调用，也可以自定义线程池。

```
public interface BackgroundCallback
{
 /**
 * Called when the async background operation completes
 *
 * @param client the client
 * @param event operation result details
 * @throws Exception errors
 */
  public void processResult(CuratorFramework client, CuratorEvent event) throws Exception;
 }
```

如上接口，主要参数为 client 客户端， 和 服务端事件 event

inBackground 异步处理默认在EventThread中执行

```
@Test
public void test() throws Exception {
 curatorFramework.getData().inBackground((item1, item2) -> {
 log.info(" background: {}", item2);
 }).forPath(ZK_NODE);

 TimeUnit.SECONDS.sleep(Integer.MAX_VALUE);
}
```

指定线程池

```
@Test
public void test() throws Exception {
 ExecutorService executorService = Executors.newSingleThreadExecutor();

 curatorFramework.getData().inBackground((item1, item2) -> {
 log.info(" background: {}", item2);
 },executorService).forPath(ZK_NODE);

 TimeUnit.SECONDS.sleep(Integer.MAX_VALUE);
}
```

Curator 监听器：

```
/**
 * Receives notifications about errors and background events
 */
public interface CuratorListener
{
 /**
 * Called when a background task has completed or a watch has triggered
 *
 * @param client client
  * @param event the event
  * @throws Exception any errors
  */
  public void eventReceived(CuratorFramework client, CuratorEvent event)
throws Exception;
 }
```

针对 background 通知和错误通知。使用此监听器之后，调用inBackground 方法会异步获得监听

Curator Caches:

Curator 引入了 Cache 来实现对 Zookeeper 服务端事件监听，Cache 事件监听可以理解为一个本地缓存视图与远程 Zookeeper 视图的对比过程。Cache 提供了反复注册的功能。Cache 分为两类注册类型：节点监听和子节点监听。

node cache:

NodeCache 对某一个节点进行监听

```
public NodeCache(CuratorFramework client,
 String path)
Parameters:
client - the client
path - path to cache
```

可以通过注册监听器来实现，对当前节点数据变化的处理

```
public void addListener(NodeCacheListener listener)
 Add a change listener
Parameters:
```

```
listener - the listener
```

```
@Slf4j
public class NodeCacheTest extends AbstractCuratorTest{

 public static final String NODE_CACHE="/node-cache";

 @Test
 public void testNodeCacheTest() throws Exception {

 createIfNeed(NODE_CACHE);
  NodeCache nodeCache = new NodeCache(curatorFramework, NODE_CACHE);
  nodeCache.getListenable().addListener(new NodeCacheListener() {
  @Override
  public void nodeChanged() throws Exception {
  log.info("{} path nodeChanged: ",NODE_CACHE);
  printNodeData();
  }
  });

  nodeCache.start();
  }


  public void printNodeData() throws Exception {
  byte[] bytes = curatorFramework.getData().forPath(NODE_CACHE);
  log.info("data: {}",new String(bytes));
  }
 }
```

path cache:

PathChildrenCache 会对子节点进行监听，但是不会对二级子节点进行监听，

```
public PathChildrenCache(CuratorFramework client,
 String path,
 boolean cacheData)
Parameters:
client - the client
path - path to watch
cacheData - if true, node contents are cached in addition to the stat
```

可以通过注册监听器来实现，对当前节点的子节点数据变化的处理

```
public void addListener(PathChildrenCacheListener listener)
 Add a change listener
Parameters:
listener - the listener
```

```
@Slf4j
public class PathCacheTest extends AbstractCuratorTest{

 public static final String PATH="/path-cache";

 @Test
 public void testPathCache() throws Exception {

 createIfNeed(PATH);
  PathChildrenCache pathChildrenCache = new PathChildrenCache(curatorFramework, PATH, true);
  pathChildrenCache.getListenable().addListener(new PathChildrenCacheListener() {
  @Override
  public void childEvent(CuratorFramework client, PathChildrenCacheEvent event) throws Exception {
  log.info("event: {}",event);
  }
  });

  // 如果设置为true则在首次启动时就会缓存节点内容到Cache中
  pathChildrenCache.start(true);
  }
 }
```

tree cache:

TreeCache 使用一个内部类TreeNode来维护这个一个树结构。并将这个树结构与ZK节点进行了映射。所以TreeCache 可以监听当前节点下所有节点的事件。

```
public TreeCache(CuratorFramework client,
 String path,
 boolean cacheData)
Parameters:
client - the client
path - path to watch
```

```
7 cacheData - if true, node contents are cached in addition to the stat
```

可以通过注册监听器来实现，对当前节点的子节点，及递归子节点数据变化的处理

```
1 public void addListener(TreeCacheListener listener)
2  Add a change listener
3 Parameters:
4 listener - the listener
```

```
1 @Slf4j
2 public class TreeCacheTest extends AbstractCuratorTest{
3
4  public static final String TREE_CACHE="/tree-path";
5
6  @Test
7  public void testTreeCache() throws Exception {
8  createIfNeed(TREE_CACHE);
9  TreeCache treeCache = new TreeCache(curatorFramework, TREE_CACHE);
10  treeCache.getListenable().addListener(new TreeCacheListener() {
11  @Override
12  public void childEvent(CuratorFramework client, TreeCacheEvent event) t
hrows Exception {
13  log.info(" tree cache: {}",event);
14  }
15  });
16  treeCache.start();
17  }
18 }
```

Zookeeper在分布式命名服务中的实践