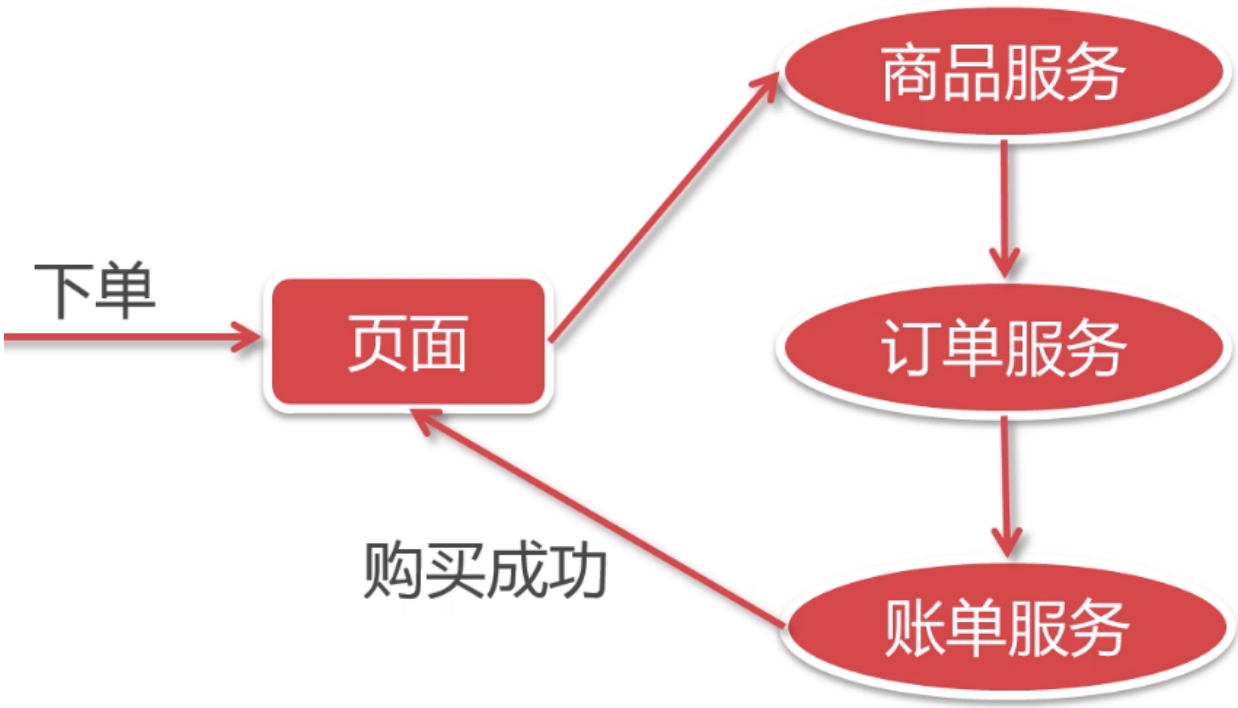
Zookeeper分布式专题与Dubbo微服务入门

第1章 分布式系统概念与Zookeeper简介

1-2 什么是分布式系统



面向服务式的开发

1-3 分布式系统的瓶颈以及zk的相关特性

- 一致性：数据一致性，数据按照顺序分批入库
- 原子性：事务要么成功要么失败，不会局部化
- 单一视图：客户端连接集群中的任一zk节点，数据都是一致的
- 可靠性：每次对zk的操作状态都会保存在服务端
- 实时性：客户端可以读取到zk服务端的最新数据

第2章 zookeeper的安装

下载加压`zookeeper-x.tar.gz`，配置环境变量

```
export JAVA_HOME=/usr/jdk8
export CLASSPATH=.:$JAVA_HOME/jre/lib/rt.jar:$JAVA_HOME/lib/dt.jar:$JAVA_HOME/lib/tools.jar
export ZOOKEEPER_HOME=/usr/local/zookeeper

export PATH=$PATH:$ZOOKEEPER_HOME/bin:$JAVA_HOME/bin
```

目录结构：

- bin：主要的一些运行命令
- conf：存放配置文件，其中需要修改`zk.cfg`
- contrib：附加的一些功能
- dist-maven：mvn编译后的目录
- docs：文档
- lib：需要依赖的jar包
- recipes：案例demo代码
- src：源码

2-4 zookeeper配置文件介绍，运行zk

把`conf/zoo_sample.cfg`复制为`zoo.cfg`，修改`zoo.cfg`

```
# The number of milliseconds of each tick
tickTime=2000
# The number of ticks that the initial
# synchronization phase can take
initLimit=10    # 用于集群，允许从节点连接并同步到master节点的初始化连接时间，以tickTime的倍数来表示
# The number of ticks that can pass between
# sending a request and getting an acknowledgement
syncLimit=5     # 用于集群，master主节点与从节点之间的心跳机制，以tickTime的倍数来表示
# the directory where the snapshot is stored.
# do not use /tmp for storage, /tmp here is just
# example sakes.
dataDir=/usr/local/zookeeper/dataDir            # 必须配置
dataLogDir=/usr/local/zookeeper/dataLogDir       # 日志目录，如果不配置会和dataDir共用
# the port at which the clients will connect
clientPort=2181             # 客户端连接服务器的端口，默认2181
# the maximum number of client connections.
# increase this if you need to handle more clients
#maxClientCnxns=60
#
# 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
#autopurge.purgeInterval=1
```

```
./zkServer.sh start
./zkServer.sh status
./zkServer.sh stop
./zkServer.sh restart
```

第3章 Zookeeper的基本数据模型

3-1 zk数据模型介绍

- 是一个树形结构。
- zk的数据模型也可以理解为linux\unix的文件目录。
- 每一个节点都称之为znode，它可以有子节点，也可以有数据。
- 每个节点分为临时节点和永久节点，临时节点在客户端断开后消失。
- 每个zk节点都有各自的版本号，可以通过命令行来显示节点信息。
- 每当节点数据发生变化，那么该节点的版本号会累加（和数据库中的乐观锁相似）。
- 删除/修改过时节点，版本号不匹配则会报错。
- 每个zk节点存储的数据不宜过大，几k即可。
- 节点可以设置权限acl，可以通过权限来限制用户的访问。

3-2 zk客户端连接/关闭服务端，查看znode

客户端连接

`bin/zkCli.sh`

查看znode结构

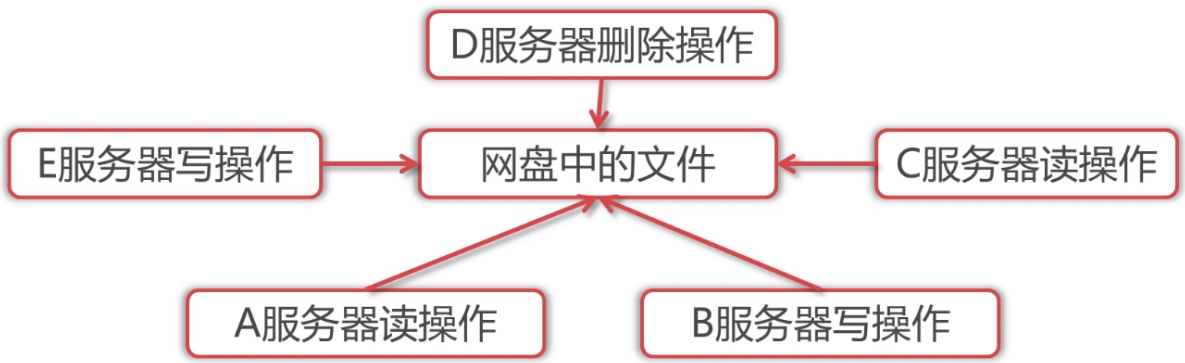
`ls /`

关闭客户端连接

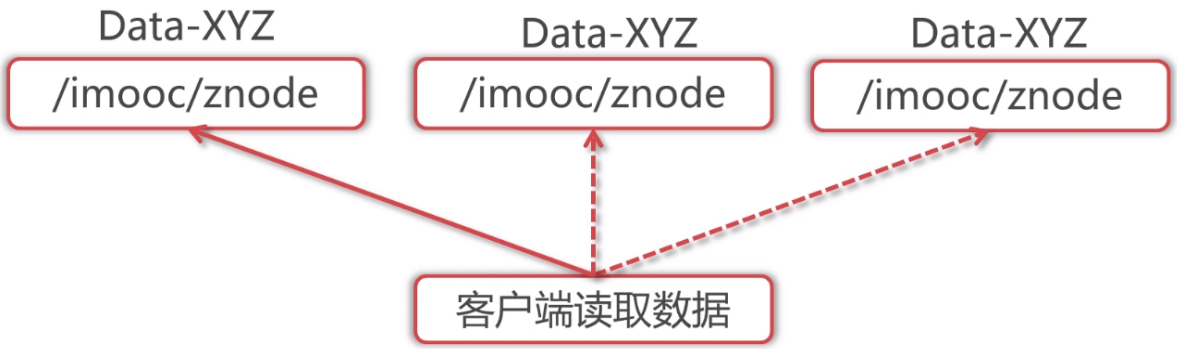
`quit`

3-3 zookeeper的作用

1. **master节点选举**：当主节点挂了后，从节点就会接手工作，并且保证这个节点是唯一的，这也是所谓的首脑模式，从而保证我们的集群是高可用的。
2. **统一配置文件管理**：只需要部署一台服务器，则可以把相同的配置文件同步更新到其他所有服务器，此操作在云计算机中用的特别多。
3. **发布与订阅**：类似消息队列MQ。dubbo发布者把数据存在znode上，订阅者会读取这个数据。
4. **提供分布式锁**：分布式环境中不同进程之间争夺资源，类似于多线程中的锁。



5. **集群管理**：集群中保证数据的强一致性，客户端不管链接任何节点的时候，都可以读到一致的数据。



第4章 ZK基本特性与基于Linux的ZK客户端命令行

4-2 session的基本原理与create命令的使用

session的基本原理

- 客户端与服务端之间的连接存在会话
- 每个会话都可以设置一个超时时间
- 心跳结束，session则过期
- session过期，则临时节点znode会被抛弃
- 心跳机制：客户端向服务端的ping包请求

zk常用命令行操作

- `create [-s] [-e] path data acl`：
  - `create /imooc imooc-data`：默认创建的节点，非顺序，持久化
  - `get /imooc`：

```
imooc-data
cZxid = 0x4
ctime = Fri Jun 15 13:49:52 CST 2018
mZxid = 0x4
mtime = Fri Jun 15 13:49:52 CST 2018
pZxid = 0x4
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 10
numChildren = 0
```

  - `create -e /imooc imooc-data`：创建临时的节点
  - `get /imooc`：

```
imooc-data
cZxid = 0x4
ctime = Fri Jun 15 13:49:52 CST 2018
mZxid = 0x4
mtime = Fri Jun 15 13:49:52 CST 2018
pZxid = 0x5
cversion = 1            # 版本号做了一次更新
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 10
numChildren = 1
```

- `get /imooc/tmp`：

```
imooc-data
cZxid = 0x5
ctime = Fri Jun 15 13:52:45 CST 2018
mZxid = 0x5
mtime = Fri Jun 15 13:52:45 CST 2018
pZxid = 0x5
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x100004196e80001          # 说明是临时节点
dataLength = 10
numChildren = 0
```

- `create -s /imooc/sec sec`：创建顺序的节点

```
Created /imooc/sec0000000001

# 如果再执行一次create -s /imooc/sec se
Created /imooc/sec0000000002
```

4-4 zk特性-理解watcher机制

在zkCli中可以通过help查看有哪些命令可以设置watcher

```
stat path [watch]
set path data [version]
ls path [watch]
delquota [-n|-b] path
ls2 path [watch]
setAcl path acl
setquota -n|-b val path
history
redo cmdno

printwatches on|off
```

```
delete path [version]
sync path
listquota path
rmr path
get path [watch]
create [-s] [-e] path data acl
addauth scheme auth
quit
getAcl path
close
connect host:port
```

- 针对每个节点的操作，都会有一个监督者->watcher
- 当监控的某个对象（znode）发生了变化，则触发watcher事件
- zk中的watcher是一次性的，触发后立即销毁
- 父节点、子节点的增删改都能够触发其watcher
- 针对不同类型的操作，触发的watcher事件也不同
  - （子）节点创建事件
  - （子）节点删除事件
  - （子）节点数据变化事件

4-5 父节点watcher事件

- 创建父节点触发：NodeCreated
  - `stat /imooc watch`
  - `create /imooc 123`

```
WATCHER::

WatchedEvent state:SyncConnected type:NodeCreated path:/imooc
Created /imooc
```

- 修改父节点数据触发：NodeDataChanged
  - `get /imooc watch`：也可以用`stat /imooc watch`
  - `set /imooc 789`

```
WATCHER::

WatchedEvent state:SyncConnected type:NodeDataChanged path:/imooc
cZxid = 0x14
ctime = Fri Jun 15 14:10:56 CST 2018
mZxid = 0x16
mtime = Fri Jun 15 14:14:11 CST 2018
pZxid = 0x14

cversion = 0
```

```
dataVersion = 2             # 修改数据的版本
aclVersion = 0              # acl的版本
ephemeralOwner = 0x0
dataLength = 3
numChildren = 0
```

- 删除父节点触发：NodeDeleted
  - `stat /imooc watch`
  - `delete /imooc`

```
WATCHER::

WatchedEvent state:SyncConnected type:NodeDeleted path:/imooc
```

4-6 子节点watcher事件

- `ls`为父节点设置watcher，创建子节点触发：NodeChildrenChanged
  - 如果父节点不存在，首先创建父节点（create /imooc leihou）
  - `ls /imooc watch`：注意必须用`ls`
  - `create /imooc/abc alielie`：

```
WATCHER::

WatchedEvent state:SyncConnected type:NodeChildrenChanged path:/imooc
Created /imooc/abc
```

- `ls`为父节点设置watcher，删除子节点触发：NodeChildrenChanged
  - `ls /imooc watch`
  - `delete /imooc/abc`：

```
WATCHER::

WatchedEvent state:SyncConnected type:NodeChildrenChanged path:/imooc
```

- **创建和删除子节点都只会触发父节点的NodeChildrenChanged事件**，因为父节点不关心子节点究竟怎么变化。
- `ls`为父节点设置watcher，修改子节点不触发事件：客户端
  - `create /imooc/xyz leihou`
  - `ls /imooc watch`
  - `set /imooc/xyz 9090`

```
cZxid = 0x21
ctime = Fri Jun 15 14:26:24 CST 2018
mZxid = 0x22
mtime = Fri Jun 15 14:26:43 CST 2018
pZxid = 0x21
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 4
numChildren = 0
```

4-7 watcher常用使用场景

统一资源配置

假设有一个zookeeper集群，主机更新了客户端的数据库配置信息（/imooc/sqlConfig），其他的节点也都跟着改变，因此都触发watcher，这样可以通知每个主机相连的客户端去更新配置信息。

4-8 权限acl详解，acl的构成：schema与id

- 针对节点可以设置相关读写等权限，目的是为了保障数据安全性
- 权限permissions可以指定不同的权限范围以及角色

ACL命令行

- `getAcl path`：获取某个节点的acl权限信息
- `setAcl path acl`：设置某个节点的acl权限信息
- `addauth scheme auth`：输入认证授权信息，注册时输入明文密码（登录），但是在zk的系统里，密码是以加密的形式存在的。

ACL的构成

- zk的acl通过`[scheme:id:permissions]`来构成权限列表
  - scheme：代表采用的某种权限机制
    - world：world下只有一个id：anyone，那么组合的写法就是`world:anyone:[permissions]`
    - auth：代表认证登录，需要注册用户有权限就可以，形式为`auth:user:password:[permissions]`
    - digest：需要对密码加密才能访问，与auth的区别：就是auth明文，digest密文。组合为：`digest:username:BASE64(SHA1(password)):[permissions]`
    - ip：当设置为ip指定的ip地址，此时限制ip进行访问，比如：`ip:192.168.1.1:[permissions]`
    - super：代表超级管理员，拥有所有的权限
  - id：代表允许访问的用户
  - permissions：权限组合字符串

    权限字符串缩写：c、r、d、w、a
    - CREATE：创建子节点
    - READ：获取节点/子节点
    - DELETE：删除子节点

- WRITE：设置节点数据
- ADMIN：设置权限

4-10 acl命令行讲解

1. world

`create /imooc/abc leihou`

`getAcl /imooc/abc`：

```
'world,'anyone
: cdrwa
```

`setAcl /imooc/abc world:anyone:crwa`：

```
cZxid = 0x29
ctime = Fri Jun 15 14:58:55 CST 2018
mZxid = 0x29
mtime = Fri Jun 15 14:58:55 CST 2018
pZxid = 0x29
cversion = 0
dataVersion = 0
aclVersion = 1              # 权限版本发生变化
ephemeralOwner = 0x0
dataLength = 2
numChildren = 0
```

`getAcl /imooc/abc`：

```
'world,'anyone
: crwa
```

`delete /imooc/abc/123`：

```
Authentication is not valid : /imooc/abc/123
```

2. auth

使用auth的话必须先注册

`setAcl /imooc/abc auth:imooc:imooc:cdrwa`

```
Acl is not valid : /names/imooc
```

`addauth digest imooc:imooc`

`setAcl /imooc/abc auth:imooc:imooc:cdrwa`：

- 这里也可以匿名设置：
  - `setAcl /imooc/abc auth::cdrwa`
  - 因为这个设置的用户名和密码其实和addauth中的一样的，也就是说即使这里的用户名和密码随便设置，但是getAcl的时候，得到的还是`imooc:XwEDaL3J0JQGkRQzM0DpO6zMzZs=`，因为addauth中设置的就是`imooc:XwEDaL3J0JQGkRQzM0DpO6zMzZs=`

`getAcl /imooc/abc`

```
'digest,'sherry:XwEDaL3J0JQGkRQzM0DpO6zMzZs=                # 数据库中存的是密文
: cdrwa
```

3. digest

假设已经注册好了用户imooc:imooc

`setAcl /names/test digest:imooc:XwEDaL3J0JQGkRQzM0DpO6zMzZs=:cdra`

`getAcl /names/test`

```
'digest,'imooc:XwEDaL3J0JQGkRQzM0DpO6zMzZs=
: cdra
```

`get /names/test`

```
Authentication is not valid : /names/test
```

`addauth digest imooc:imooc`

`get /names/test`

```
leihou
cZxid = 0x36
ctime = Fri Jun 15 15:20:44 CST 2018
mZxid = 0x36
mtime = Fri Jun 15 15:20:44 CST 2018
pZxid = 0x36
cversion = 0
dataVersion = 0
aclVersion = 1
ephemeralOwner = 0x0
dataLength = 6
numChildren = 0
```

4. super

1. 修改`zkServer.sh`增加super管理员
2. 重启`zkServer.sh`

4-15 acl的使用场景

- 开发/测试环境分离，开发者无权操作测试库的节点，只能看。
- 生产环境上控制指定ip的服务可以访问相关节点，防止混乱。

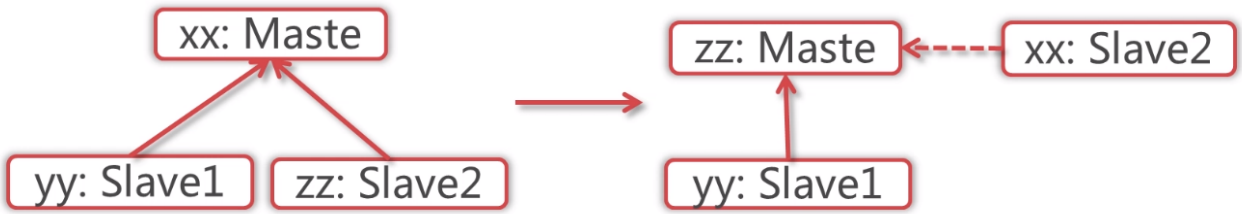
4-16 zk的四字命令

- zk可以通过它自身提供的简写命令来和服务器进行交互
- 需要使用到nc命令，安装：`yum install nc`
- `echo [command] | nc [ip] [port]`

`echo conf | nc localhost 2181`

```
clientPort=2181
dataDir=/usr/local/zookeeper/dataDir/version-2
dataLogDir=/usr/local/zookeeper/dataLogDir/version-2
tickTime=2000
maxClientCnxns=60
minSessionTimeout=4000
maxSessionTimeout=40000
serverId=0
```

第5章 选举模式和zookeeper的集群安装



当主节点挂掉后，zz这个从节点经过选举称为主节点，此时原来的主节点如果恢复了，就会变成从节点。

选举模式要求集群有奇数个节点，所以集群中最少有3个节点。

5-2 搭建伪分布式集群

1. 把zookeeper1复制两份：zookeeper2，zookeeper3
2. 修改zookeeper1下的conf中的`zoo.cfg`：

```
# 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=/usr/local/zookeeper1/dataDir                    # 需要修改
```

```
dataLogDir=/usr/local/zookeeper1/dataLogDir             # 需要修改
# the port at which the clients will connect
clientPort=2181                                       # 需要修改
# the maximum number of client connections.
# increase this if you need to handle more clients
#maxClientCnxns=60
#
# 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
#autopurge.purgeInterval=1


server.1=192.168.1.103:2888:3888                # 288x是集群中数据同步用的端口号
server.2=192.168.1.103:2889:3889                # 388x是选举模式用的端口号
server.3=192.168.1.103:2890:3890
```

3. 进入zookeeper1下的dataDir目录，创建一个文件`myid`，在`myid`中输入1
4. 修改zookeeper2和zookeeper3下的conf中的`zoo.cfg`，把对应的1分别换成2和3，也在zookeeper2和zookeeper3下重复步骤3，只是分别输入2和3

5-3 搭建真实的集群

和搭建伪分布式集群过程一样，只不过3个server的ip不相同，端口相同。

第6章 使用ZooKeeper原生Java API

- 会话连接与恢复
- 节点的增删改查
- watch与acl的相关操作

6-1 建立客户端与zk服务端的连接

1. pom中添加依赖

```
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.4.11</version>
</dependency>
```

2. 编写java客户端

```
public class ZKConnect implements Watcher {
```

```
    private final static Logger log = LoggerFactory.getLogger(ZKConnect.class);

    private static final String zkServerPath = "locahost:2181,locahost:2182,locahost:2183";
    private static final Integer timeout = 5000;

    public static void main(String[] args) throws Exception {
        /*
         * 客户端和zk服务端链接是一个异步的过程
         * 当连接成功后后，客户端会收的一个watch通知
         *
         *ZooKeeper(String connectString,
         *          int sessionTimeout,
         *          Watcher watcher,
         *          long sessionId,
         *          byte[] sessionPasswd,
         *          boolean canBeReadOnly)
         *
         * 参数：
         * connectString：连接服务器的ip字符串，
         *      比如: "192.168.1.1:2181,192.168.1.2:2181,192.168.1.3:2181"
         *      可以是一个ip，也可以是多个ip，一个ip代表单机，多个ip代表集群
         *      也可以在ip后加路径
         * sessionTimeout：超时时间，心跳收不到了，那就超时
         * watcher：如果收到watch通知，就会触发Watcher的process()方法；如果不需要，那就设置为null
         * canBeReadOnly：可读，当这个物理机节点断开后，还是可以读到数据的，只是不能写，
         *                          此时数据被读取到的可能是旧数据，此处建议设置为false，不推荐使用
         * sessionId：会话的id
         * sessionPasswd：会话密码    当会话丢失后，可以依据 sessionId 和 sessionPasswd 重新获取会话
         */
        ZooKeeper zk = new ZooKeeper(zkServerPath, timeout, new ZKConnect());

        log.warn("客户端开始连接zookeeper服务器...");
        log.warn("连接状态：{}", zk.getState());

        Thread.sleep(2000);

        log.warn("连接状态：{}", zk.getState());
    }

    @Override
    public void process(WatchedEvent event) {
        log.warn("接受到watch通知：{}", event);
    }
}
/*
 * 2018-06-19 09:58:29,439 [main] [com.imooc.zk.demo.ZKConnect.main(ZKConnect.java:44)] - [WARN]
客户端开始连接zookeeper服务器...
 * 2018-06-19 09:58:29,442 [main] [com.imooc.zk.demo.ZKConnect.main(ZKConnect.java:45)] - [WARN]
连接状态：CONNECTING



 * 2018-06-19 09:58:29,454 [main-EventThread]
```

```
[com.imooc.zk.demo.ZKConnect.process(ZKConnect.java:54)] - [WARN] 接受到watch通知：WatchedEvent
state:SyncConnected type:None path:null
 * 2018-06-19 09:58:31,443 [main] [com.imooc.zk.demo.ZKConnect.main(ZKConnect.java:49)] - [WARN]
连接状态：CONNECTED
 */
```

6-2 zk会话重连机制

```
public class ZKConnectSessionWatcher implements Watcher {

    private final static Logger log = LoggerFactory.getLogger(ZKConnectSessionWatcher.class);

    private static final String zkServerPath = "locahost:2181,locahost:2182,locahost:2183";
    private static final Integer timeout = 5000;

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

        ZooKeeper zk = new ZooKeeper(zkServerPath, timeout, new ZKConnectSessionWatcher());

        log.warn("客户端开始连接zookeeper服务器...");
        log.warn("连接状态：{}", zk.getState());
        Thread.sleep(1000);
        long sessionId = zk.getSessionId();
        String ssid = "0x" + Long.toHexString(sessionId);
        byte[] sessionPassword = zk.getSessionPasswd();
        log.warn("session Id为：{}, session Password为：{}", ssid, sessionPassword);
        log.warn("连接状态：{}", zk.getState());

        Thread.sleep(200);

        // 开始会话重连
        log.warn("开始会话重连...");

        ZooKeeper zkSession = new ZooKeeper(zkServerPath,
                timeout,
                new ZKConnectSessionWatcher(),
                sessionId,
                sessionPassword);
        log.warn("重新连接状态zkSession：{}", zkSession.getState());
        Thread.sleep(1000);
        log.warn("重新连接状态zkSession：{}", zkSession.getState());
    }

    @Override
    public void process(WatchedEvent event) {
        log.warn("接受到watch通知：{}", event);
    }
}
/*


 * 2018-06-19 10:01:28,562 [main]
```

```
[com.imooc.zk.demo.ZKConnectSessionWatcher.main(ZKConnectSessionWatcher.java:25)] - [WARN] 客户端开始连接zookeeper服务器...
 * 2018-06-19 10:01:28,565 [main]
[com.imooc.zk.demo.ZKConnectSessionWatcher.main(ZKConnectSessionWatcher.java:26)] - [WARN] 连接状态：CONNECTING
 * 2018-06-19 10:01:28,577 [main-EventThread]
[com.imooc.zk.demo.ZKConnectSessionWatcher.process(ZKConnectSessionWatcher.java:51)] - [WARN] 接受到watch通知：WatchedEvent state:SyncConnected type:None path:null
 * 2018-06-19 10:01:29,565 [main]
[com.imooc.zk.demo.ZKConnectSessionWatcher.main(ZKConnectSessionWatcher.java:31)] - [WARN] session Id为：0x300000469360005, session Password为：[-89, -25, -48, -39, 85, -108, 80, -119, 48, -31, 91, -57, 40, -18, -31, 41]
 * 2018-06-19 10:01:29,565 [main]
[com.imooc.zk.demo.ZKConnectSessionWatcher.main(ZKConnectSessionWatcher.java:32)] - [WARN] 连接状态：CONNECTED
 * 2018-06-19 10:01:30,566 [main]
[com.imooc.zk.demo.ZKConnectSessionWatcher.main(ZKConnectSessionWatcher.java:37)] - [WARN] 开始会话重连...，此时zk状态为：CONNECTED
 * 2018-06-19 10:01:30,567 [main]
[com.imooc.zk.demo.ZKConnectSessionWatcher.main(ZKConnectSessionWatcher.java:44)] - [WARN] 重新连接状态zkSession：CONNECTING
 * 2018-06-19 10:01:30,570 [main-EventThread]
[com.imooc.zk.demo.ZKConnectSessionWatcher.process(ZKConnectSessionWatcher.java:51)] - [WARN] 接受到watch通知：WatchedEvent state:SyncConnected type:None path:null
 * 2018-06-19 10:01:31,569 [main]
[com.imooc.zk.demo.ZKConnectSessionWatcher.main(ZKConnectSessionWatcher.java:46)] - [WARN] 重新连接状态zkSession：CONNECTED
 */
```

6-3 同步异步创建zk节点

1. 同步

```
/*
 * 同步或者异步创建节点，都不支持子节点的递归创建，异步有一个callback函数
 * 参数：
 * path：创建的路径
 * data：存储的数据的byte[]
 * acl：控制权限策略
 *          Ids.OPEN_ACL_UNSAFE --> world:anyone:cdrwa
 *          CREATOR_ALL_ACL --> auth:user:password:cdrwa
 * createMode：节点类型，是一个枚举
 *          PERSISTENT：持久节点
 *          PERSISTENT_SEQUENTIAL：持久顺序节点
 *          EPHEMERAL：临时节点
 *          EPHEMERAL_SEQUENTIAL：临时顺序节点
 */
String result = zookeeper.create(path, data, acls, CreateMode.EPHEMERAL);
System.out.println("创建节点： " + result + " 成功...");
```

2. 异步

```
String ctx = "{'create':'success'}";            //可以是任意形式（Object），灵活
zookeeper.create(path, data, acls, CreateMode.PERSISTENT, (rc, pa, ct, name) -> {
    System.out.println("创建节点："  + pa);
    System.out.println((String)ct);
}, ctx);
//lambda表达式是一个回调函数，
```

6-4 修改zk节点数据

和创建zk节点时一样，修改zk节点数据也有同步和异步两种方式。这里只举同步的例子，异步和创建节点时相似，也是传入回调函数和ctx。

```
/**
 * 参数：
 * path：节点路径
 * data：数据
 * version：数据版本号，需要和当前的数据版本号相同
 */
Stat status  = zookeeper().setData("/testnode", "xyz".getBytes(), 0);
System.out.println(status.getVersion());
```

6-5 删除zk节点

1. 同步

```
zookeeper().delete("/testnode", 1);
```

2. 异步

```
zookeeper().delete("/test-delete-node", 0, (rc, path, ctx) -> {
        System.out.println("删除节点" + path);
        System.out.println((String)ctx);
    }, ctx);
```

6-6 CountDownLatch的介绍

- 是一个计数器
- 多用于线程，可以暂停也可以继续

6-8 获取节点数据

```
/**
 * 参数：
 * path：节点路径
 * watch：true或者false，注册一个watch事件
 * stat：状态

 */
```

```
byte[] resByte = zookeeper().getData("/data", true, stat);
String result = new String(resByte);
System.out.println("当前值:" + result);
System.out.println("当前状态：" + stat);

/*
 * 当前值:alielie
 * 当前状态：8589934594,17179869235,1529052975251,1529378867665,1,0,0,0,7,0,8589934594
 *
 */
```

6-9 获取zk子节点列表

```
/**
 * 参数：
 * path：父节点路径
 * watch：true或者false，注册一个watch事件
 */
 //同步调用
List<String> strChildList = zookeeper().getChildren("/data", true);
for (String s : strChildList) {
    System.out.println(s);
}

//异步调用
String ctx = "{'callback':'ChildrenCallback'}";
zookeeper().getChildren("/imooc", true, new ChildrenCallBack(), ctx);
zookeeper().getChildren("/imooc", true, new Children2CallBack(), ctx);
```

6-10 判断zk节点是否存在

```
/**
 * 参数：
 * path：节点路径
 * watch：watch
 */
Stat stat = zookeeper().exists("/imooc-fake", true);
if (stat != null) {
    System.out.println("查询的节点版本为dataVersion：" + stat.getVersion());
} else {
    System.out.println("该节点不存在...");
}
```

也有异步的调用方式。

6-12 acl-自定义用户权限

```
List<ACL> acls = new ArrayList<ACL>();
Id imooc1 = new Id("digest", AclUtils.getDigestUserPwd("imooc1:123456"));       //id:password
Id imooc2 = new Id("digest", AclUtils.getDigestUserPwd("imooc2:123456"));
acls.add(new ACL(Perms.ALL, imooc1));
acls.add(new ACL(Perms.READ, imooc2));
acls.add(new ACL(Perms.DELETE | Perms.CREATE, imooc2));
zkServer.createZKNode("/aclimooc/testdigest", "testdigest".getBytes(), acls);
```

运行后，在命令行中运行一下命令：

`getAcl /aclimooc/testdigest` ：

```
'digest,'imooc1:ee8R/pr2P4sGnQYNGyw2M5S5IMU=
: cdrwa
'digest,'imooc2:eBdFG0gQw0YArfEFDCRP3LzIp6k=
: r
'digest,'imooc2:eBdFG0gQw0YArfEFDCRP3LzIp6k=
: cd
```

此时通过java API操作`/aclimooc/testdigest`节点：

```
zkServer.getZookeeper().addAuthInfo("digest", "imooc1:123456".getBytes());
zkServer.createZKNode("/aclimooc/testdigest/childtest",
                      "childtest".getBytes(),
                      Ids.CREATOR_ALL_ACL);
```

第7章 Apache Curator客户端的使用

常用的zk java客户端：

- zk原生api
- zkclient
- Apache curator

zk原生api的不足之处：

- 超时重连，不支持自动，需要手动操作
- Watch注册一次后会失效
- 不支持递归创建节点

1. 添加依赖

```
<dependency>
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-framework</artifactId>
    <version>4.0.0</version>
</dependency>


<dependency>
```

```
    <groupId>org.apache.curator</groupId>
    <artifactId>curator-recipes</artifactId>
    <version>4.0.0</version>
</dependency>

<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.4.11</version>
</dependency>
```

2. 代码

```
public class CuratorOperator {

    private CuratorFramework client = null;
    private static final String zkServerPath =
"192.168.1.103:2181,192.168.1.103:2182,192.168.1.103:2183";

    /**
     * 实例化zk客户端
     */
    public CuratorOperator() {
        /**
         * 同步创建zk示例，原生api是异步的
         *
         * curator链接zookeeper的策略:ExponentialBackoffRetry
         * baseSleepTimeMs：初始sleep的时间
         * maxRetries：最大重试次数
         * maxSleepMs：最大重试时间
         */
//      RetryPolicy retryPolicy = new ExponentialBackoffRetry(1000, 5);

        /**
         * curator链接zookeeper的策略:RetryNTimes
         * n：重试的次数
         * sleepMsBetweenRetries：每次重试间隔的时间
         */
        RetryPolicy retryPolicy = new RetryNTimes(3, 5000);

        /**
         * curator链接zookeeper的策略:RetryOneTime
         * sleepMsBetweenRetry:每次重试间隔的时间
         */
//      RetryPolicy retryPolicy2 = new RetryOneTime(3000);

        /**
         * 永远重试，不推荐使用
         */
//      RetryPolicy retryPolicy3 = new RetryForever(retryIntervalMs)

        /**
```

```
         * curator链接zookeeper的策略:RetryUntilElapsed
         * maxElapsedTimeMs:最大重试时间
         * sleepMsBetweenRetries:每次重试间隔
         * 重试时间超过maxElapsedTimeMs后，就不再重试
         */
//      RetryPolicy retryPolicy4 = new RetryUntilElapsed(2000, 3000);

        client = CuratorFrameworkFactory.builder()
                .connectString(zkServerPath)
                .sessionTimeoutMs(10000).retryPolicy(retryPolicy)
                .namespace("workspace").build();        //namespace：命令空间，在连接生成成功后，就会生成wordspace这个节点
        //就是在根节点（/）下创建一个/workspace，然后所有这个client中操作的节点都创建在/workspace下
        client.start();
    }

    /**
     *
     * @Description: 关闭zk客户端连接
     */
    public void closeZKClient() {
        if (client != null) {
            this.client.close();
        }
    }

    public static void main(String[] args) throws Exception {
        // 实例化
        CuratorOperator cto = new CuratorOperator();
        boolean isZkCuratorStarted = cto.client.isStarted();
        System.out.println("当前客户的状态：" + (isZkCuratorStarted ? "连接中" : "已关闭"));


        String nodePath = "/super/imooc";

        /*// 创建节点
        byte[] data = "superme".getBytes();
        cto.client.create().creatingParentsIfNeeded() //.creatingParentsIfNeeded(): 可以递归一层层创建，使用命令行必须要先创建/super，再创建/super/imooc
            .withMode(CreateMode.PERSISTENT)
            .withACL(ZooDefs.Ids.OPEN_ACL_UNSAFE)
            .forPath(nodePath, data);*/

        /*// 更新节点数据
        byte[] newData = "leihou".getBytes();
        cto.client.setData().withVersion(0).forPath(nodePath, newData);
        //如果版本不对会报错：KeeperException$BadVersionException: KeeperErrorCode = BadVersion for /workspace/super/imooc*/

        /*// 删除节点
        cto.client.delete()
                  .guaranteed()                 // 如果删除失败，那么在后端还是继续会删除，直到成功

                  .deletingChildrenIfNeeded()   // 如果有子节点，就删除
```

```
                .withVersion(1)
                .forPath(nodePath);           //这个会删除imooc及其子节点，但是不会删
除/workspace/super*/

        /*// 读取节点数据
        Stat stat = new Stat();
        byte[] data = cto.client.getData().storingStatIn(stat).forPath(nodePath);
        //storingStatIn(stat)：在获得这个节点数据的同时把节点的信息填在stat中，否则stat.getVersion()
== 0(初始值)
        System.out.println("节点" + nodePath + "的数据为: " + new String(data));
        System.out.println("该节点的版本号为: " + stat.getVersion());*/

        /*// 查询子节点
        List<String> childNodes = cto.client.getChildren().forPath(nodePath);
        System.out.println("开始打印子节点：");
        for (String s : childNodes) {
            System.out.println(s);
        }*/

        /*// 判断节点是否存在,如果不存在则为空
        Stat statExist = cto.client.checkExists().forPath(nodePath + "/a");
        System.out.println(statExist == null ? "该节点不存在" : "该节点存在");*/


        /*// watcher 事件  当使用usingWatcher的时候，监听只会触发一次，监听完毕后就销毁
        //CuratorWatcher和Watcher的区别就是CuratorWatcher的process()方法会抛出异常
        cto.client.getData().usingWatcher(new MyCuratorWatcher()).forPath(nodePath);
        //cto.client.getData().usingWatcher(new MyWatcher()).forPath(nodePath);*/

        /*// 为节点添加watcher，监听会一直存在，可以多次触发
        NodeCache nodeCache = new NodeCache(cto.client, nodePath);
        // NodeCache: 监听数据节点的变更，会触发事件
        // buildInitial为true : 初始化的时候获取node的值并且缓存
        nodeCache.start(true);
        if (nodeCache.getCurrentData() != null) {
            System.out.println("节点初始化数据为：" + new
String(nodeCache.getCurrentData().getData()));
        } else {
            System.out.println("节点初始化数据为空...");
        }
        nodeCache.getListenable().addListener(
                () -> {       //当监听到nodepath节点改变，就会调用这个方法
                    if (nodeCache.getCurrentData() == null) {
                        System.out.println("空");
                        return;
                    }
                    String data = new String(nodeCache.getCurrentData().getData());
                    System.out.println("节点路径：" + nodeCache.getCurrentData().getPath() + "数
据：" + data);
                }
        );*/
```

```
        // 为子节点添加watcher
        // PathChildrenCache: 监听数据节点的增删改，会触发事件
        String childNodePathCache =  nodePath;
        // cacheData: 设置缓存节点的数据状态
        PathChildrenCache childrenCache = new PathChildrenCache(cto.client, childNodePathCache,
true);
         //StartMode: 初始化方式
         //POST_INITIALIZED_EVENT：异步初始化，初始化之后会触发事件
         //NORMAL：异步初始化
         //BUILD_INITIAL_CACHE：同步初始化
        childrenCache.start(StartMode.POST_INITIALIZED_EVENT);
        List<ChildData> childDataList = childrenCache.getCurrentData();
        System.out.println("当前数据节点的子节点数据列表：");
        for (ChildData cd : childDataList) {
            String childData = new String(cd.getData());
            System.out.println(childData);
        }
        childrenCache.getListenable().addListener(
                (CuratorFramework client, PathChildrenCacheEvent event) -> {
                    if(event.getType().equals(PathChildrenCacheEvent.Type.INITIALIZED)){
                        System.out.println("子节点初始化ok...");
                    }

                    else if(event.getType().equals(PathChildrenCacheEvent.Type.CHILD_ADDED)){
                        String path = event.getData().getPath();
                        if (path.equals(ADD_PATH)) {
                            System.out.println("添加子节点:" + event.getData().getPath());
                            System.out.println("子节点数据:" + new
String(event.getData().getData()));
                        } else if (path.equals("/super/imooc/e")) {
                            System.out.println("添加不正确...");
                        }

                    }else if(event.getType().equals(PathChildrenCacheEvent.Type.CHILD_REMOVED)){
                        System.out.println("删除子节点:" + event.getData().getPath());
                    }else if(event.getType().equals(PathChildrenCacheEvent.Type.CHILD_UPDATED)){
                        System.out.println("修改子节点路径:" + event.getData().getPath());
                        System.out.println("修改子节点数据:" + new
String(event.getData().getData()));
                    }
                }
            );

        Thread.sleep(100000);

        cto.closeZKClient();
        boolean isZkCuratorStarted2 = cto.client.isStarted();
        System.out.println("当前客户的状态：" + (isZkCuratorStarted2 ? "连接中" : "已关闭"));
    }

    public final static String ADD_PATH = "/super/imooc/d";


}
```

```
class MyCuratorWatcher implements CuratorWatcher {

    @Override
    public void process(WatchedEvent event) throws Exception {
        System.out.println("触发watcher，节点路径为：" + event.getPath());
    }

}
```

第8章 Dubbo入门到重构服务

系统之间的调用方式：

- Webservice - wsdl
- httpclient
- rpc通信（dubbo）/ restful（springcloud）

8-2 dubbo入门简介

- 最大程度解耦，降低系统耦合性
- 生产者/消费者模式
- zk注册中心，admin监控中心，协议支持

第9章 分布式锁

9-1 死锁和活锁

死锁：一个进程获得锁，可以对数据库增删改查，其他进程等待这个进程结束后才有可能拿到锁访问数据库

活锁：一个进程对数据库增删改查，其他进程可以对数据库执行只读操作。

9-2 zookeeper分布式锁

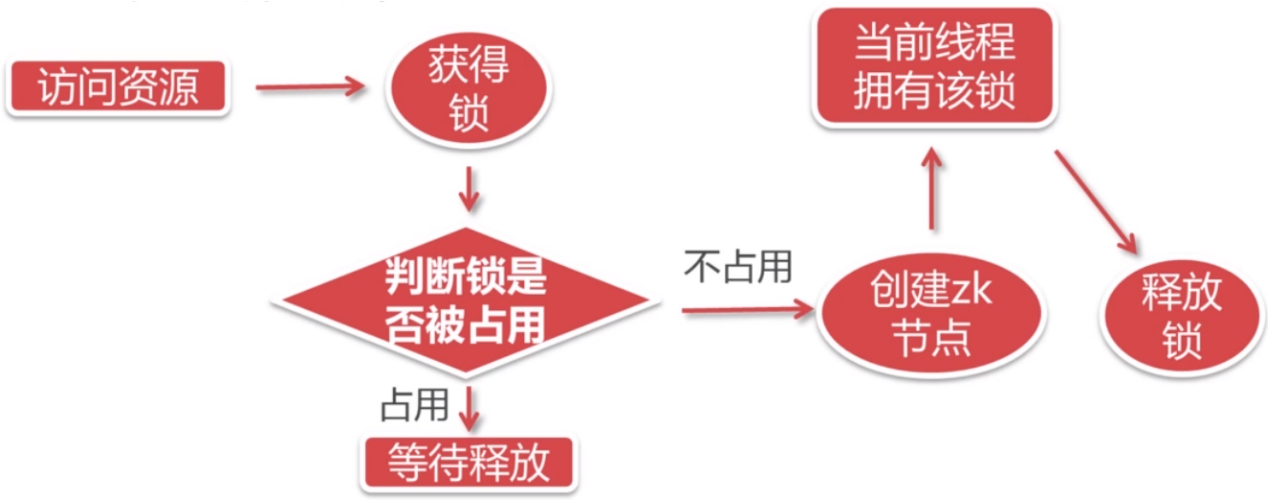
多个进程访问同一个数据库，可能会有最终数据不一致的现象。

举例：秒杀，第一个请求过来查询库存，此时库存数量为1，可以创建订单，在创建订单的过程中，另一个请求过来，查询库存也为1，也可以创建订单。最终数据对业务是有问题的。

解决数据不一致（防止库存为负）：**加分布式锁**

9-4 获取分布式锁的流程

分布式锁的流程图



用户如果不把节点delete，而是直接关闭会话，这个zk节点应该随着会话的关闭而删除，所以zk节点应该是**临时性**的。

分布式锁和分布式缓存本质是一样的，都是把锁或者缓存这种本来存在单机上的东西交给第三方来管理。

9-5 开发分布式锁

千万不能忘记释放锁

```
package com.imooc.curator.utils;

import java.util.concurrent.CountDownLatch;

import org.apache.curator.framework.CuratorFramework;
import org.apache.curator.framework.recipes.cache.PathChildrenCache;
import org.apache.curator.framework.recipes.cache.PathChildrenCache.StartMode;
import org.apache.curator.framework.recipes.cache.PathChildrenCacheEvent;
import org.apache.curator.framework.recipes.cache.PathChildrenCacheListener;
import org.apache.zookeeper.CreateMode;
import org.apache.zookeeper.ZooDefs.Ids;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

/**
 * @Description: 分布式锁的实现工具类
 */
public class DistributedLock {

    private CuratorFramework client = null;     // zk客户端

    final static Logger log = LoggerFactory.getLogger(DistributedLock.class);

    // 用于挂起当前请求，并且等待上一个分布式锁释放
    private static CountDownLatch zkLocklatch = new CountDownLatch(1);

    // 分布式锁的总节点名,为了区分不同的项目
    private static final String ZK_LOCK_PROJECT = "imooc-locks";

    // 分布式锁节点，当前业务的锁
```

```
    private static final String DISTRIBUTED_LOCK = "distributed_lock";

    // 构造函数
    public DistributedLock(CuratorFramework client) {
        this.client = client;
    }

    /**
     * @Description: 初始化锁
     */
    public void init() {

        // 使用命名空间
        client = client.usingNamespace("ZKLocks-Namespace");

        /**
         * 创建zk锁的总节点，相当于eclipse的工作空间下的项目
         *      ZKLocks-Namespace
         *          |
         *           —— imooc-locks
         *                  |
         *                   —— distributed_lock
         */
        try {
            if (client.checkExists().forPath("/" + ZK_LOCK_PROJECT) == null) {
                client.create()
                        .creatingParentsIfNeeded()
                        .withMode(CreateMode.PERSISTENT)
                        .withACL(Ids.OPEN_ACL_UNSAFE)
                        .forPath("/" + ZK_LOCK_PROJECT);
            }
            // 针对zk的分布式锁节点，创建相应的watcher事件监听
            addWatcherToLock("/" + ZK_LOCK_PROJECT);

        } catch (Exception e) {
            log.error("客户端连接zookeeper服务器错误... 请重试...");
        }
    }

    /**
     * @Description: 获得分布式锁
     */
    public void getLock() {
        // 使用死循环，当且仅当上一个锁释放并且当前请求获得锁成功后才会跳出
        while (true) {
            try {
                client.create()
                        .creatingParentsIfNeeded()
                        .withMode(CreateMode.EPHEMERAL)      //临时节点
                        .withACL(Ids.OPEN_ACL_UNSAFE)
                        .forPath("/" + ZK_LOCK_PROJECT + "/" + DISTRIBUTED_LOCK);
                log.info("获得分布式锁成功...");

                return;                                      // 如果锁的节点能被创建成功，则锁没有被
```

```
占用
            } catch (Exception e) {
                log.info("获得分布式锁失败...");
                try {
                    // 如果没有获取到锁，需要重新设置同步资源值
                    if(zkLocklatch.getCount() <= 0){
                        zkLocklatch = new CountDownLatch(1);
                    }
                    // 阻塞线程
                    zkLocklatch.await();
                } catch (InterruptedException ie) {
                    ie.printStackTrace();
                }
            }
        }
    }

    /**
     * @Description: 释放分布式锁
     */
    public boolean releaseLock() {
        try {
            if (client.checkExists().forPath("/" + ZK_LOCK_PROJECT + "/" + DISTRIBUTED_LOCK) !=
null) {
                client.delete().forPath("/" + ZK_LOCK_PROJECT + "/" + DISTRIBUTED_LOCK);
            }
        } catch (Exception e) {
            e.printStackTrace();
            return false;
        }
        log.info("分布式锁释放完毕");
        return true;
    }

    /**
     * @Description: 创建watcher监听
     */
    public void addWatcherToLock(String path) throws Exception {
        final PathChildrenCache cache = new PathChildrenCache(client, path, true);
        cache.start(StartMode.POST_INITIALIZED_EVENT);
        cache.getListenable().addListener(new PathChildrenCacheListener() {
            public void childEvent(CuratorFramework client, PathChildrenCacheEvent event) throws
Exception {
                if (event.getType().equals(PathChildrenCacheEvent.Type.CHILD_REMOVED)) {
                    String path = event.getData().getPath();66
                    log.info("上一个会话已释放锁或该会话已断开，节点路径为：" + path);
                    if(path.contains(DISTRIBUTED_LOCK)) {
                        log.info("释放计数器，让当前请求来获得分布式锁...");
                        zkLocklatch.countDown();
                    }
                }
            }

        });
```

```
    }
}
```

备注

- 这种较短的命令行是输入的命令：

  `create /imooc/abc leihou`

- 这种整行的命令行是输入命令后的显示结果

```
'world,'anyone
: cdrwa
```