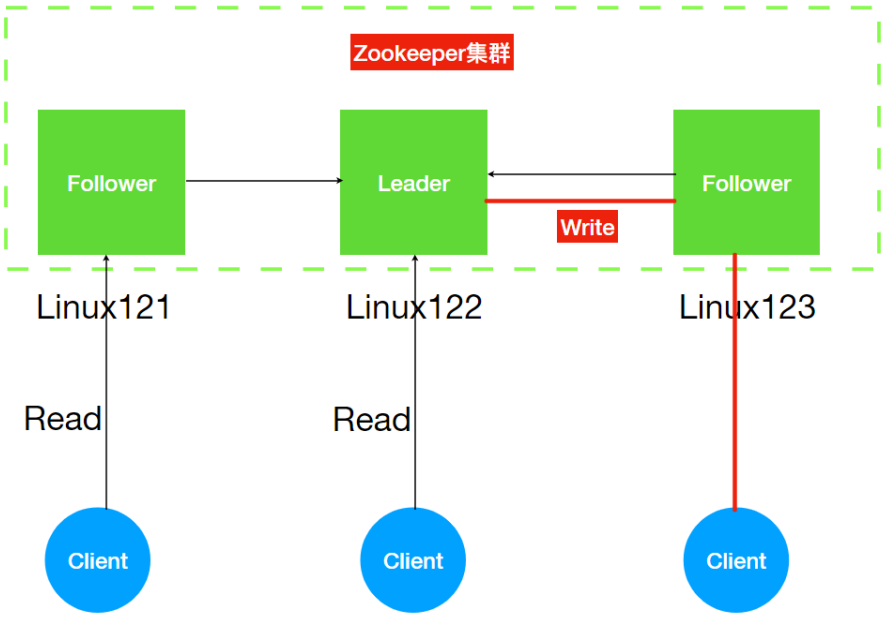
1. Zookeeper简介

1.1 Zookeeper是什么?

Zookeeper 是一个分布式协调服务的开源框架。 主要用来解决分布式集群中应用系统的一致性问题，例如怎样避免同时操作同一数据造成脏读的问题。分布式系统中数据存在一致性的问题！！

- **ZooKeeper 本质上是一个分布式的小文件存储系统。** 提供基于类似于文件系统的目录树方式的数据存储，并且可以对树中的节点进行有效管理。
- **ZooKeeper 提供给客户端监控存储在zk内部数据的功能**，从而可以达到基于数据的集群管理。 诸如： 统一命名服务(dubbo)、分布式配置管理(solr的配置集中管理)、分布式消息队列（sub/pub）、分布式锁、分布式协调等功能。

1.2 zookeeper的架构组成



Leader

- Zookeeper 集群工作的核心角色
- 集群内部各个服务器的调度者。
- 事务请求（写操作） 的唯一调度和处理者，保证集群事务处理的顺序性；对于 create，setData， delete 等有写操作的请求，则需要统一转发给leader 处理， leader 需要决定编号、执行操作，这个过程称为一个事务。

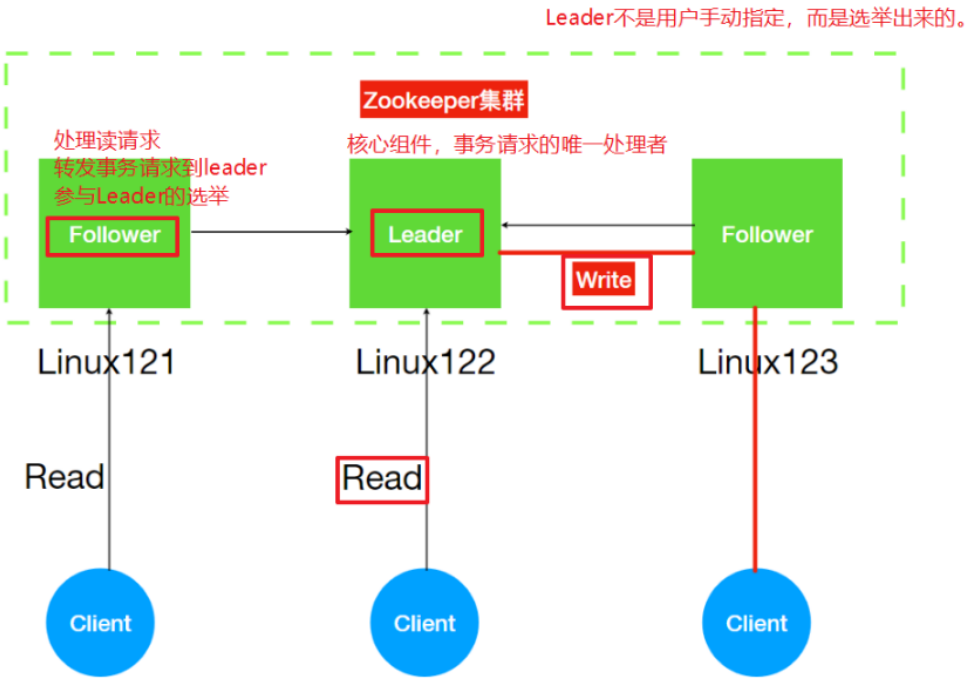
Follower

- 处理客户端非事务（读操作） 请求，
- 转发事务请求给 Leader；
- 参与集群 Leader 选举投票 2n-1台可以做集群投票。

此外，针对访问量比较大的 zookeeper 集群， 还可新增观察者角色。

Observer

- 观察者角色，观察 Zookeeper 集群的最新状态变化并将这些状态同步过来，其对于非事务请求可以进行独立处理，对于事务请求，则会转发给 Leader服务器进行处理。
- 不会参与任何形式的投票只提供非事务服务，通常用于在不影响集群事务处理能力的前提下提升集群的非事务处理能力。增加了集群增加**并发的读请求**。



ZK也是Master/slave架构，但是与之前不同的是zk集群中的Leader不是指定而来，而是通过选举产生。

1.3 Zookeeper 特点

1. Zookeeper：一个领导者（leader:老大），多个跟随者（follower:小弟）组成的集群。
2. Leader负责进行投票的发起和决议，更新系统状态(内部原理)
3. Follower用于接收客户请求并向客户端返回结果，在选举Leader过程中参与投票
4. 集群中只要有**半数以上节点存活**，Zookeeper集群就能正常服务。
5. 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。
6. 更新请求顺序进行(内部原理)
7. 数据更新原子性，一次数据更新要么成功，要么失败。

2. Zookeeper环境搭建

2.1 Zookeeper的搭建方式

Zookeeper安装方式有三种，单机模式和集群模式以及伪集群模式。

- 单机模式：Zookeeper只运行在一台服务器上，适合测试环境；
- 伪集群模式：就是在一台服务器上运行多个Zookeeper 实例；
- 集群模式：Zookeeper运行于一个集群上，适合生产环境，这个计算机集群被称为一个“集合体”

2.2 Zookeeper集群搭建

下载

首先我们下载稳定版本的zookeeper http://zookeeper.apache.org/releases.html

上传

下载完成后，将zookeeper压缩包 zookeeper-3.4.14.tar.gz上传到linux系统/opt/lagou/software

解压 压缩包

```
tar -zxvf zookeeper-3.4.14.tar.gz -C ../servers/
```

修改配置文件创建data与log目录

```
#创建zk存储数据目录
mkdir -p /opt/lagou/servers/zookeeper-3.4.14/data
#创建zk日志文件目录
mkdir -p /opt/lagou/servers/zookeeper-3.4.14/data/logs
#修改zk配置文件
cd /opt/lagou/servers/zookeeper-3.4.14/conf
#文件改名
mv zoo_sample.cfg zoo.cfg

vim zoo.cfg
#更新datadir
dataDir=/opt/lagou/servers/zookeeper-3.4.14/data
#增加logdir
dataLogDir=/opt/lagou/servers/zookeeper-3.4.14/data/logs
#增加集群配置
##server.服务器ID=服务器IP地址：服务器之间通信端口：服务器之间投票选举端口
server.1=linux121:2888:3888
server.2=linux122:2888:3888
server.3=linux123:2888:3888
#打开注释
#ZK提供了自动清理事务日志和快照文件的功能，这个参数指定了清理频率，单位是小时
autopurge.purgeInterval=1
```

```
# 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=/opt/lagou/servers/zookeeper-3.4.14/data
dataLogDir=/opt/lagou/servers/zookeeper-3.4.14/data/logs
# the port at which the clients will connect
clientPort=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
#集群配置
server.1=linux121:2888:3888
server.2=linux122:2888:3888
server.3=linux123:2888:3888
~
~
```

添加myid配置

1. 在zookeeper的 data 目录下创建一个 myid 文件，内容为1，这个文件就是记录每个服务器的ID

```
cd /opt/lagou/servers/zookeeper-3.4.14/data
echo 1 > myid
```

```
[root@linux121 data]# echo 1 > myid
[root@linux121 data]# ll
total 4
drwxr-xr-x 2 root root 6 Jul 16 03:46 logs
-rw-r--r-- 1 root root 2 Jul 16 03:58 myid
[root@linux121 data]#
```

安装包分发并修改myid的值

```
rsync-script /opt/lagou/servers/zookeeper-3.4.14
```

修改myid值 linux122

```
echo 2 >/opt/lagou/servers/zookeeper-3.4.14/data/myid
```

修改myid值 linux123

```
echo 3 >/opt/lagou/servers/zookeeper-3.4.14/data/myid
```

依次启动三个zk实例

启动命令（三个节点都要执行）

```
/opt/lagou/servers/zookeeper-3.4.14/bin/zkServer.sh start
```

查看zk启动情况

```
/opt/lagou/servers/zookeeper-3.4.14/bin/zkServer.sh status
```

```
[root@linux122 servers]# /opt/lagou/servers/zookeeper-3.4.14/bin/zkServer.sh start
ZooKeeper JMX enabled by default
Using config: /opt/lagou/servers/zookeeper-3.4.14/bin/../conf/zoo.cfg
Starting zookeeper ... STARTED
[root@linux122 servers]# /opt/lagou/servers/zookeeper-3.4.14/bin/zkServer.sh status
ZooKeeper JMX enabled by default
Using config: /opt/lagou/servers/zookeeper-3.4.14/bin/../conf/zoo.cfg
Mode: leader
[root@linux122 servers]#
```

集群启动停止脚本

```
vim zk.sh

#!/bin/sh
echo "start zookeeper server..."
if(($#==0));then
echo "no params";
exit;
fi
```

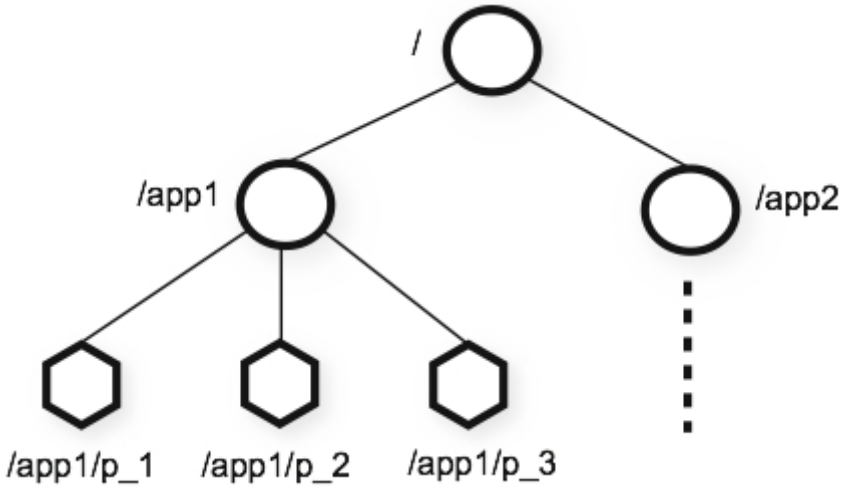
```
hosts="linux121 linux122 linux123"

for host in $hosts
do
ssh $host "source /etc/profile; /opt/lagou/servers/zookeeper-
3.4.14/bin/zkServer.sh $1"
done
```

3. Zookeeper数据结构与监听机制

ZooKeeper数据模型Znode

在ZooKeeper中，数据信息被保存在一个个数据节点上，这些节点被称为znode。ZNode 是 Zookeeper 中最小数据单位，在 ZNode 下面又可以再挂 ZNode，这样一层层下去就形成了一个层次化命名空间 ZNode 树，我们称为 ZNode Tree，它采用了类似文件系统的层级树状结构进行管理。见下图示例：



在 Zookeeper 中，每一个数据节点都是一个 ZNode，上图根目录下有两个节点，分别是：app1 和 app2，其中 app1 下面又有三个子节点,所有ZNode按层次化进行组织，形成这么一颗树，ZNode的节点路径标识方式和Unix文件系统路径非常相似，都是由一系列使用斜杠（/）进行分割的路径表示，开发人员可以向这个节点写入数据，也可以在这个节点下面创建子节点。

3.1 ZNode 的类型

刚刚已经了解到，Zookeeper的znode tree是由一系列数据节点组成的，那接下来，我们就对数据节点做详细讲解

Zookeeper 节点类型可以分为三大类：

持久性节点（Persistent）

临时性节点（Ephemeral）

顺序性节点（Sequential）

在开发中在创建节点的时候通过组合可以生成以下四种节点类型：持久节点、持久顺序节点、临时节点、临时顺序节点。不同类型的节点则会有不同的生命周期

持久节点：是Zookeeper中最常见的一种节点类型，所谓持久节点，就是指节点被创建后会一直存在服务器，直到删除操作主动清除

持久顺序节点：就是有顺序的持久节点，节点特性和持久节点是一样的，只是额外特性表现在顺序上。顺序特性实质是在创建节点的时候，会在节点名后面加上一个数字后缀，来表示其顺序。

临时节点：就是会被自动清理掉的节点，它的生命周期和客户端会话绑在一起，客户端会话结束，节点会被删除掉。与持久性节点不同的是，临时节点不能创建子节点。

临时顺序节点：就是有顺序的临时节点，和持久顺序节点相同，在其创建的时候会在名字后面加上数字后缀。

事务ID

首先，先了解，事务是对物理和抽象的应用状态上的操作集合。往往在现在的概念中，狭义上的事务通常指的是数据库事务，一般包含了一系列对数据库有序的读写操作，这些数据库事务具有所谓的ACID特性，即原子性（Atomic）、一致性（Consistency）、隔离性（Isolation）和持久性（Durability）。

而在ZooKeeper中，事务是指能够改变**ZooKeeper服务器状态**的操作，我们也称之为事务操作或更新操作，一般包括数据节点创建与删除、数据节点内容更新等操作。对于每一个事务请求，ZooKeeper都会为其分配一个全局唯一的事务ID，用 **ZXID** 来表示，通常是一个 64 位的数字。每一个 ZXID 对应一次更新操作，从这些ZXID中可以间接地识别出ZooKeeper处理这些更新操作请求的全局顺序

zk中的事务指的是对zk服务器状态改变的操作(create,update data,更新字节点)；zk对这些事务操作都会编号，这个编号是自增长的被称为ZXID。

3.2 ZNode 的状态信息

```
#使用bin/zkCli.sh 连接到zk集群
[zk: localhost:2181(CONNECTED) 2] get /zookeeper

cZxid = 0x0
ctime = Wed Dec 31 19:00:00 EST 1969
mZxid = 0x0
mtime = Wed Dec 31 19:00:00 EST 1969
pZxid = 0x0
cversion = -1
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 0
numChildren = 1
```

整个 ZNode 节点内容包括两部分：节点数据内容和节点状态信息。数据内容是空，其他的属于状态信息。那么这些状态信息都有什么含义呢?

```
 cZxid 就是 Create ZXID，表示节点被创建时的事务ID。
 ctime 就是 Create Time，表示节点创建时间。
 mZxid 就是 Modified ZXID，表示节点最后一次被修改时的事务ID。
 mtime 就是 Modified Time，表示节点最后一次被修改的时间。
 pZxid 表示该节点的子节点列表最后一次被修改时的事务 ID。只有子节点列表变更才会更新 pZxid，子节点内容变更不会更新。
 cversion 表示子节点的版本号。
 dataVersion 表示内容版本号。
 aclVersion 标识acl版本
 ephemeralOwner 表示创建该临时节点时的会话 sessionID，如果是持久性节点那么值为 0
 dataLength 表示数据长度。
 numChildren 表示直系子节点数。
```

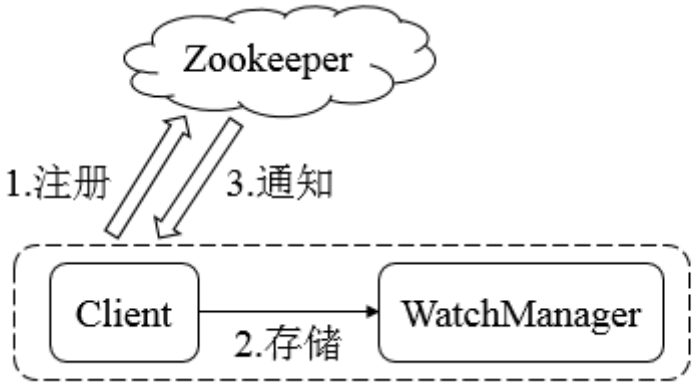
3.3 Watcher 机制

Zookeeper使用Watcher机制实现分布式数据的发布/订阅功能

一个典型的发布/订阅模型系统定义了一种 一对多的订阅关系，能够让多个订阅者同时监听某一个主题对象，当这个主题对象自身状态变化时，会通知所有订阅者，使它们能够做出相应的处理。

在 ZooKeeper 中，引入了 Watcher 机制来实现这种分布式的通知功能。ZooKeeper 允许客户端向服务端注册一个 Watcher 监听，当服务端的一些指定事件触发了这个 Watcher，那么Zk就会向指定客户端发送一个事件通知来实现分布式的通知功能。

整个Watcher注册与通知过程如图所示。



Zookeeper的Watcher机制主要包括**客户端线程、客户端WatcherManager、Zookeeper服务器**三部分。

具体工作流程为：

- 客户端在向Zookeeper服务器注册的同时，会将Watcher对象存储在客户端的WatcherManager当中
- 当Zookeeper服务器触发Watcher事件后，会向客户端发送通知
- 客户端线程从WatcherManager中取出对应的Watcher对象来执行回调逻辑

4. Zookeeper的基本使用

4.1 ZooKeeper命令行操作

现在已经搭建起了一个能够正常运行的zookeeper服务了，所以接下来，就是来借助客户端来对zookeeper的数据节点进行操作

首先，进入到zookeeper的bin目录之后

通过zkClient进入zookeeper客户端命令行

```
./zkcli.sh  连接本地的zookeeper服务器
./zkCli.sh -server ip:port(2181) 连接指定的服务器
```

连接成功之后，系统会输出Zookeeper的相关环境及配置信息等信息。输入help之后，屏幕会输出可用的Zookeeper命令，如下图所示

```
[zk: localhost:2181(CONNECTED) 3] help
ZooKeeper -server host:port cmd args
```

```
        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
```

创建节点

使用create命令，可以创建一个Zookeeper节点， 如

```
create [-s][-e] path data
其中，-s或-e分别指定节点特性，顺序或临时节点，若不指定，则创建持久节点
```

① 创建顺序节点

使用 **create -s /zk-test 123** 命令创建zk-test顺序节点

```
[zk: localhost:2181(CONNECTED) 4] create -s /zk-test 123
Created /zk-test0000000000
```

执行完后，就在根节点下创建了一个叫做/zk-test的节点，该节点内容就是123，同时可以看到创建的zk-test节点后面添加了一串数字以示区别

② 创建临时节点

使用 **create -e /zk-temp 123** 命令创建zk-temp临时节

```
[zk: localhost:2181(CONNECTED) 1] create -e /zk-temp 123
Created /zk-temp
[zk: localhost:2181(CONNECTED) 2] ls /
[zk-test0000000000, zookeeper, zk-temp]
```

临时节点在客户端会话结束后，就会自动删除，下面使用**quit**命令退出客户端

```
[zk: localhost:2181(CONNECTED) 3] quit
Quitting...
```

再次使用客户端连接服务端，并使用ls / 命令查看根目录下的节点

```
[zk: localhost:2181(CONNECTED) 0] ls /
[zk-test0000000000, zookeeper]
```

可以看到根目录下已经不存在zk-temp临时节点了

③ 创建永久节点

使用 **create /zk-permanent 123** 命令创建zk-permanent永久节点 ,可以看到永久节点不同于顺序节点，不会自动在后面添加一串数字

```
[zk-test0000000000, zookeeper]
[zk: localhost:2181(CONNECTED) 1] create /zk-permanent 123
Created /zk-permanent
[zk: localhost:2181(CONNECTED) 2] ls /
[zk-permanent, zk-test0000000000, zookeeper]
```

读取节点

与读取相关的命令有ls 命令和get 命令

ls命令可以列出Zookeeper指定节点下的所有子节点，但只能查看指定节点下的第一级的所有子节点；

```
ls  path
其中，path表示的是指定数据节点的节点路径
```

get命令可以获取Zookeeper指定节点的数据内容和属性信息。

```
get path
```

若获取根节点下面的所有子节点，使用**ls /** 命令即可

```
[zk: localhost:2181(CONNECTED) 2] ls /
[zk-permanent, zk-test0000000000, zookeeper]
```

若想获取/zk-permanent的数据内容和属性，可使用如下命令：**get /zk-permanent**

```
[zk: localhost:2181(CONNECTED) 3] get /zk-permanent
123
cZxid = 0x300000008
ctime = Thu Jul 16 04:33:41 EDT 2020
mZxid = 0x300000008
mtime = Thu Jul 16 04:33:41 EDT 2020
pZxid = 0x300000008
cversion = 0
dataVersion = 0
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 3
numChildren = 0
```

从上面的输出信息中，我们可以看到，第一行是节点/zk-permanent 的数据内容，其他几行则是创建该节点的事务ID（cZxid）、最后一次更新该节点的事务ID（mZxid）和最后一次更新该节点的时间（mtime）等属性信息

更新节点

使用set命令，可以更新指定节点的数据内容，用法如下

```
   set path data
```

其中，data就是要更新的新内容，version表示数据版本，在zookeeper中，节点的数据是有版本概念的，这个参数用于指定本次更新操作是基于Znode的哪一个数据版本进行的，如将/zk-permanent节点的数据更新为456，可以使用如下命令：**set /zk-permanent 456**

```
[zk: localhost:2181(CONNECTED) 4] set /zk-permanent 456
cZxid = 0x300000008
ctime = Thu Jul 16 04:33:41 EDT 2020
mZxid = 0x300000009
mtime = Thu Jul 16 05:07:00 EDT 2020
pZxid = 0x300000008
cversion = 0
dataVersion = 1
aclVersion = 0
ephemeralOwner = 0x0
dataLength = 3
numChildren = 0
```

现在dataVersion已经变为1了，表示进行了更新

删除节点

使用delete命令可以删除Zookeeper上的指定节点，用法如下

```
 delete path
```

其中version也是表示数据版本，使用**delete /zk-permanent** 命令即可删除/zk-permanent节点

```
[zk: localhost:2181(CONNECTED) 8] delete /zk-permanent
[zk: localhost:2181(CONNECTED) 9] ls /
[zk-test0000000000, zookeeper]
```

可以看到，已经成功删除/zk-permanent节点。值得注意的是，**若删除节点存在子节点，那么无法删除该节点，必须先删除子节点，再删除父节点**

4.2 Zookeeper-开源客户端

ZkClient

ZkClient是Github上一个开源的zookeeper客户端，在Zookeeper原生API接口之上进行了包装，是一个更易用的Zookeeper客户端，同时，zkClient在内部还实现了诸如Session超时重连、Watcher反复注册等功能

接下来，还是从创建会话、创建节点、读取数据、更新数据、删除节点等方面来介绍如何使用zkClient这个zookeeper客户端

添加依赖：

在pom.xml文件中添加如下内容

```
<dependency>
     <groupId>org.apache.zookeeper</groupId>
     <artifactId>zookeeper</artifactId>
     <version>3.4.14</version>
</dependency>
<dependency>
         <groupId>com.101tec</groupId>
         <artifactId>zkclient</artifactId>
         <version>0.2</version>
 </dependency>
```

1.创建会话

使用ZkClient可以轻松的创建会话，连接到服务端。

```
package com.hust.grid.leesf.zkclient.examples;

import java.io.IOException;
import org.I0Itec.zkclient.ZkClient;

public class CreateSession {

    /*
        创建一个zkClient实例来进行连接

     */

    public static void main(String[] args) {
        ZkClient zkClient = new ZkClient("127.0.0.1:2181");
        System.out.println("ZooKeeper session created.");

    }
}
```

运行结果：ZooKeeper session created.

结果表明已经成功创建会话。

2. 创建节点

ZkClient提供了递归创建节点的接口，即其帮助开发者先完成父节点的创建，再创建子节点

```
package com.hust.grid.leesf.zkclient.examples;
import org.I0Itec.zkclient.ZkClient;

public class Create_Node_Sample {

    public static void main(String[] args) {
        ZkClient zkClient = new ZkClient("127.0.0.1:2181");
        System.out.println("ZooKeeper session established.");

        //createParents的值设置为true，可以递归创建节点
        zkClient.createPersistent("/lg-zkClient/lg-c1",true);
        System.out.println("success create znode.");

    }
```

```
}
```

运行结果：success create znode.

结果表明已经成功创建了节点，值得注意的是，ZkClient通过设置createParents参数为true可以递归的先创建父节点，再创建子节点

3.删除节点

ZkClient提供了递归删除节点的接口，即其帮助开发者先删除所有子节点（存在），再删除父节点。

```
package com.hust.grid.leesf.zkclient.examples;

import org.IOItec.zkclient.ZkClient;

public class Del_Data_Sample {
    public static void main(String[] args) throws Exception {
        String path = "/lg-zkClient/lg-c1";
        ZkClient zkClient = new ZkClient("127.0.0.1:2181", 5000);
        zkClient.deleteRecursive(path);
        System.out.println("success delete znode.");
    }
}
```

运行结果: success delete znode.

结果表明ZkClient可直接删除带子节点的父节点，因为其底层先删除其所有子节点，然后再删除父节点

4. 监听节点变化

```
package com.lagou.zk.demo;

import org.IOItec.zkclient.IZkChildListener;
import org.IOItec.zkclient.ZkClient;
import org.apache.zookeeper.client.ZooKeeperSaslClient;

import java.util.List;

/*
演示zkClient如何使用监听器
 */
public class Get_Child_Change {
    public static void main(String[] args) throws InterruptedException {
        //获取到zkClient
        final ZkClient zkClient = new ZkClient("linux121:2181");

        //zkClient对指定目录进行监听(不存在目录:/lg-client)，指定收到通知之后的逻辑

        //对/lag-client注册了监听器，监听器是一直监听
        zkClient.subscribeChildChanges("/lg-client", new IZkChildListener() {
            //该方法是接收到通知之后的执行逻辑定义
            public void handleChildChange(String path, List<String> childs)
throws Exception {
                //打印节点信息
                System.out.println(path + " childs changes ,current childs " +
childs);
            }
        });
```

```
        //使用zkClient创建节点，删除节点，验证监听器是否运行
        zkClient.createPersistent("/lg-client");
        Thread.sleep(1000); //只是为了方便观察结果数据
        zkClient.createPersistent("/lg-client/c1");
        Thread.sleep(1000);
        zkClient.delete("/lg-client/c1");
        Thread.sleep(1000);

        zkClient.delete("/lg-client");

        Thread.sleep(Integer.MAX_VALUE);

        /*
        1 监听器可以对不存在的目录进行监听
        2 监听目录下子节点发生改变，可以接收到通知，携带数据有子节点列表
        3 监听目录创建和删除本身也会被监听到
         */
    }
}
```

运行结果：

```
/lg-zkClient 's child changed, currentChilds:[]
/lg-zkClient 's child changed, currentChilds:[c1]
/lg-zkClient 's child changed, currentChilds:[]
/lg-zkClient 's child changed, currentChilds:null
```

结果表明：

客户端可以对一个不存在的节点进行子节点变更的监听。

一旦客户端对一个节点注册了子节点列表变更监听之后，那么当该节点的子节点列表发生变更时，服务端都会通知客户端，并将最新的子节点列表发送给客户端

该节点本身的创建或删除也会通知到客户端。

5. 获取数据（节点是否存在、更新、删除）

```
package com.lagou.zk.demo;


import org.IOItec.zkclient.IZkDataListener;
import org.IOItec.zkclient.ZkClient;

//使用监听器监听节点数据的变化
public class Get_Data_Change {

    public static void main(String[] args) throws InterruptedException {
        // 获取zkClient对象
        final ZkClient zkClient = new ZkClient("linux121:2181");
        //设置自定义的序列化类型,否则会报错！！
        zkClient.setZkSerializer(new ZkStrSerializer());


        //判断节点是否存在，不存在创建节点并赋值
```

```
        final boolean exists = zkClient.exists("/lg-client1");
        if (!exists) {
            zkClient.createEphemeral("/lg-client1", "123");
        }

        //注册监听器，节点数据改变的类型，接收通知后的处理逻辑定义
        zkClient.subscribeDataChanges("/lg-client1", new IZkDataListener() {
            public void handleDataChange(String path, Object data) throws
Exception {
                //定义接收通知之后的处理逻辑
                System.out.println(path + " data is changed ,new data " + data);
            }

            //数据删除-->节点删除
            public void handleDataDeleted(String path) throws Exception {
                System.out.println(path + " is deleted!!");
            }
        });

        //更新节点的数据，删除节点，验证监听器是否正常运行
        final Object o = zkClient.readData("/lg-client1");
        System.out.println(o);

        zkClient.writeData("/lg-client1", "new data");
        Thread.sleep(1000);

        //删除节点
        zkClient.delete("/lg-client1");
        Thread.sleep(Integer.MAX_VALUE);
    }
}
```

```
package com.lagou.zk.demo;

import org.IOItec.zkclient.exception.ZkMarshallingError;
import org.IOItec.zkclient.serialize.ZkSerializer;

public class ZkStrSerializer implements ZkSerializer {

    //序列化，数据-->byte[]
    public byte[] serialize(Object o) throws ZkMarshallingError {
        return String.valueOf(o).getBytes();
    }

    //反序列化，byte[]--->数据
    public Object deserialize(byte[] bytes) throws ZkMarshallingError {
        return new String(bytes);
    }
}
```

运行结果：

```
123
/lg-client1 data is changed ,new data new data
/lg-client1 is deleted!!
```

结果表明可以成功监听节点数据变化或删除事件。

5. Zookeeper内部原理

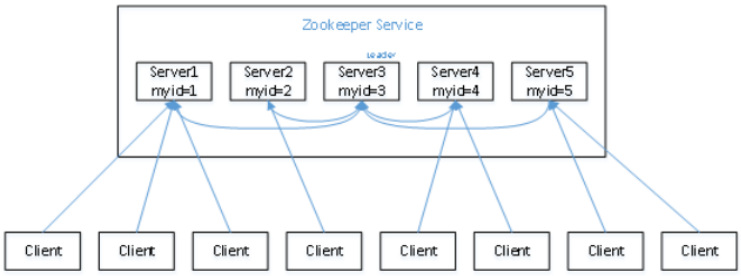
5.1 Leader选举

选举机制

- 半数机制：集群中半数以上机器存活，集群可用。所以Zookeeper适合安装奇数台服务器。
- Zookeeper虽然在配置文件中并没有指定Master和Slave。但是，Zookeeper工作时，是有一个节点为Leader，其它为Follower，Leader是通过内部的选举机制产生的。

集群首次启动

假设有五台服务器组成的Zookeeper集群，它们的id从1-5，同时它们都是最新启动的，也就是没有历史数据，在存放数据量这一点上，都是一样的。假设这些服务器依序启动，来看看会发生什么，



Zookeeper的选举机制

（1）服务器1启动，此时只有它一台服务器启动了，它发出去的报文没有任何响应，所以它的选举状态一直是LOOKING状态。

（2）服务器2启动，它与最开始启动的服务器1进行通信，互相交换自己的选举结果，由于两者都没有历史数据，所以id值较大的服务器2胜出，但是由于没有达到超过半数以上的服务器都同意选举它(这个例子中的半数以上是3)，所以服务器1、2还是继续保持LOOKING状态。

（3）服务器3启动，根据前面的理论分析，服务器3成为服务器1、2、3中的老大，而与上面不同的是，此时有三台服务器选举了它，所以它成为了这次选举的Leader。

（4）服务器4启动，根据前面的分析，理论上服务器4应该是服务器1、2、3、4中最大的，但是由于前面已经有半数以上的服务器选举了服务器3，所以它只能接收当小弟的命了。

（5）服务器5启动，同4一样称为follower。

集群非首次启动

每个节点在选举时都会参考自身节点的zxid值（事务ID）；优先选择zxid值大的节点称为Leader!!

5.2 ZAB一致性协议

1. 分布式数据一致性问题

为什么会出现分布式数据一致性问题?

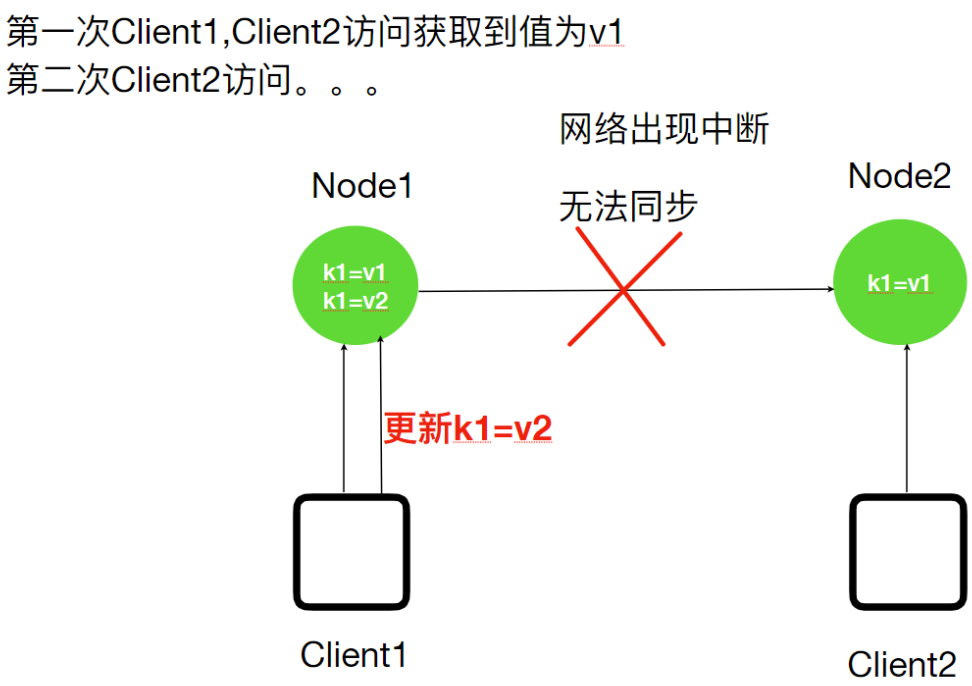
- 将数据复制到分布式部署的多台机器中，可以消除单点故障，防止系统由于某台（些）机器宕机导致的不可用。

- 通过负载均衡技术，能够让分布在不同地方的数据副本全都对外提供服务。有效提高系统性能。

在分布式系统中引入数据复制机制后，多台数据节点之间由于网络等原因很容易产生数据不一致的情况。

举例

当客户端Client1将系统中的一个值K1由V1更新为V2，但是客户端Client2读取的是一个还没有同步更新的副本，K1的值依然是V1,这就导致了数据的不一致性。其中，常见的就是主从数据库之间的复制延时问题。



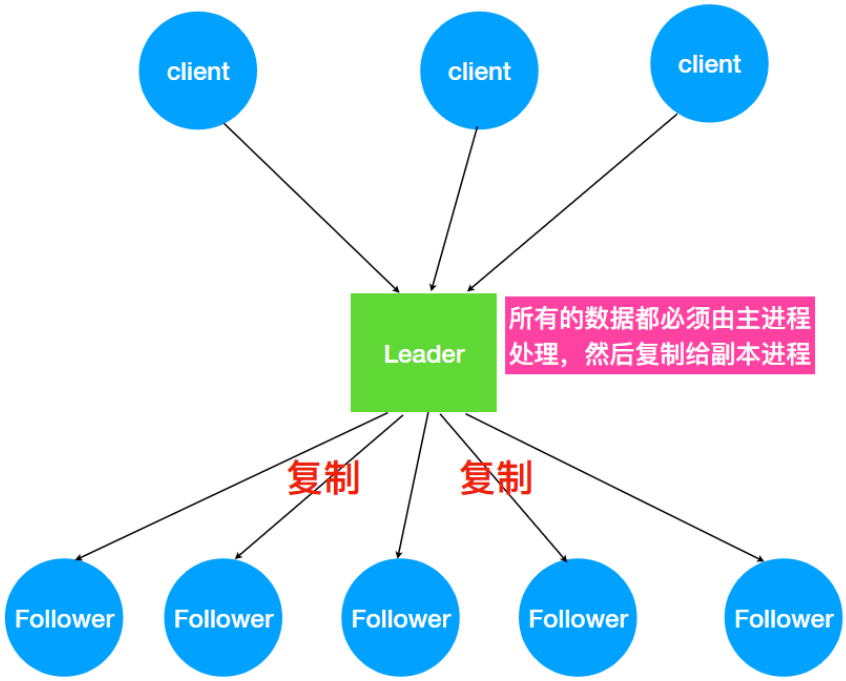
2. ZAB协议

ZK就是分布式一致性问题的工业解决方案，paxos是其底层理论算法(晦涩难懂著名)，其中zab，raft和众多开源算法是对paxos的工业级实现。ZK没有完全采用paxos算法，而是使用了一种称为Zookeeper Atomic Broadcast（ZAB，Zookeeper原子消息广播协议）的协议作为其数据一致性的核心算法。

ZAB协议

ZAB 协议是为分布式协调服务 Zookeeper 专门设计的一种支持崩溃恢复和原子广播协议。

主备模式保证一致性

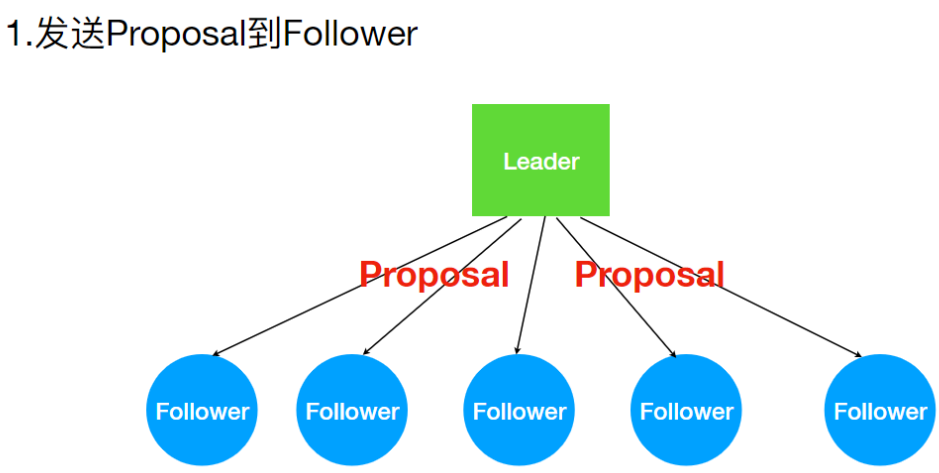


ZK怎么处理集群中的数据？所有客户端写入数据都是写入Leader中，然后，由 Leader 复制到Follower中。ZAB会将服务器数据的状态变更以事务Proposal的形式广播到所有的副本进程上，ZAB协议能够保证了事务操作的一个全局的变更序号(ZXID)。

广播消息

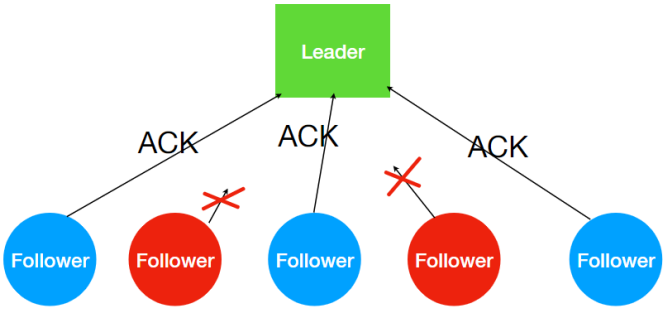
ZAB 协议的消息广播过程类似于 **二阶段提交过程**。对于客户端发送的**写请求**，全部由 Leader 接收，Leader 将请求封装成一个事务 Proposal(提议)，将其发送给所有 Follwer ，如果收到超过半数反馈 ACK，则执行 Commit 操作（先提交自己，再发送 Commit 给所有 Follwer）。

1. 发送Proposal到Follower



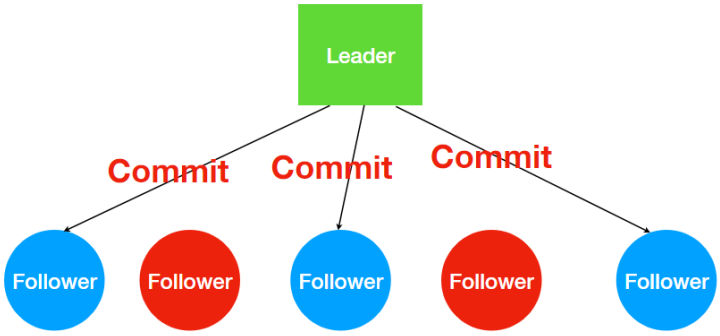
2. Leader接收Follower的ACK

2 超过半数Follower反馈ACK,Proposal被允许



3. 超过半数ACK则Commit

3 超过半数ACK，发送Commit到Follower,同时自己Commit



不能正常反馈Follower恢复正常后会进入数据同步阶段最终与Leader保持一致！！

细节

- Leader接收到Client请求之后，会将这个请求封装成一个事务，并给这个事务分配一个全局递增的唯一 ID，称为事务ID（ZXID），ZAB 协议要求保证事务的顺序，因此必须将每一个事务按照 ZXID 进行先后排序然后处理。
- ZK集群为了保证任何事务操作能够有序的顺序执行，只能是 Leader 服务器接受写请求，即使是 Follower 服务器接受到客户端的请求，也会转发到 Leader 服务器进行处理。

zk提供的应该是最终一致性的标准。zk所有节点接收写请求之后可以在一定时间内保证所有节点都能看到该条数据！！

Leader 崩溃问题

Leader宕机后，ZK集群无法正常工作，ZAB协议提供了一个高效且可靠的leader选举算法。

Leader宕机后，被选举的新Leader需要解决的问题

- ZAB 协议确保那些已经在 Leader 提交的事务最终会被所有服务器提交。
- ZAB 协议确保丢弃那些只在 Leader 提出/复制，但没有提交的事务。

基于上面的目的，ZAB协议设计了一个选举算法：能够确保已经被Leader提交的事务被集群接受，丢弃还没有提交的事务。

这个选举算法的关键点：保证选举出的新Leader拥有集群中所有节点最大编号(ZXID)的事务!!

6. Zookeeper应用实践

ZooKeeper是一个典型的发布/订阅模式的分布式数据管理与协调框架，我们可以使用它来进行分布式数据的发布与订阅。另一方面，通过对ZooKeeper中丰富的数据节点类型进行交叉使用，配合Watcher事件通知机制，可以非常方便地构建一系列分布式应用中都会涉及的核心功能，如数据发布/订阅、命名服务、集群管理、Master选举、分布式锁和分布式队列等。那接下来就针对这些典型的分布式应用场景来做下介绍

Zookeeper的两大特性：

1.客户端如果对Zookeeper的数据节点注册Watcher监听，那么当该数据节点的内容或是其子节点列表发生变更时，Zookeeper服务器就会向订阅的客户端发送变更通知。

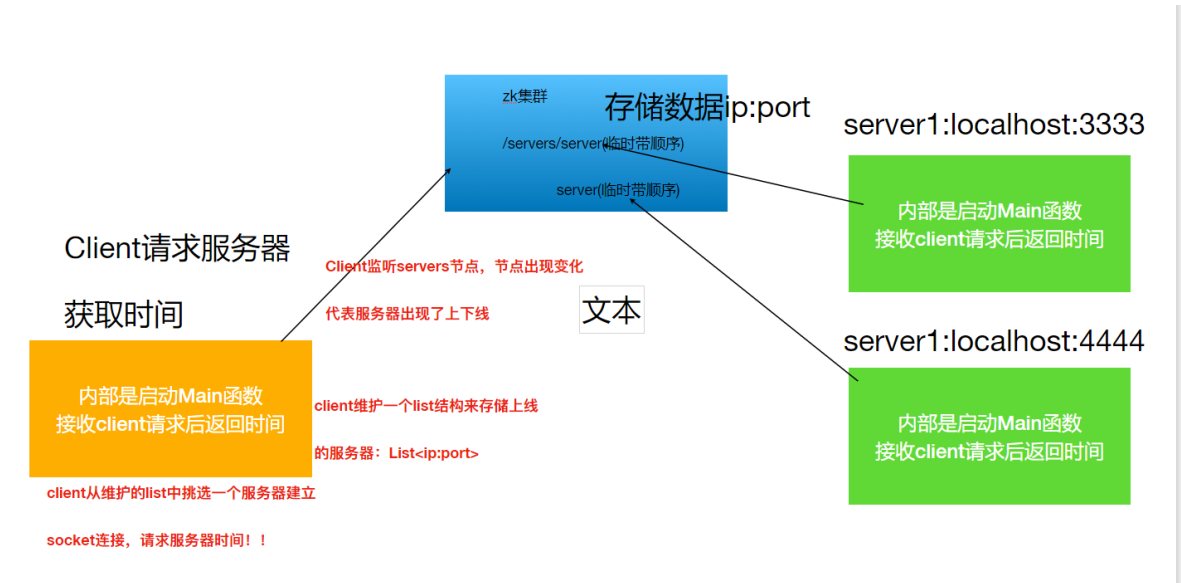
2.对在Zookeeper上创建的临时节点，一旦客户端与服务器之间的会话失效，那么临时节点也会被自动删除

利用其两大特性，可以实现集群机器存活监控系统，若监控系统在/clusterServers节点上注册一个Watcher监听，那么但凡进行动态添加机器的操作，就会在/clusterServers节点下创建一个临时节点：/clusterServers/[Hostname]，这样，监控系统就能够实时监测机器的变动情况。

6.1 服务器动态上下线监听

分布式系统中，主节点会有多台，主节点可能因为任何原因出现宕机或者下线，而任意一台客户端都要能实时感知到主节点服务器的上下线。

思路分析



具体实现

服务端

```
package com.lagou.zk.test;

import org.IOItec.zkclient.ZkClient;

public class ServerMain {

    private ZkClient zkClient = null;


    //获取到zk对象
```

```
    private void connectZK(){
         zkClient = new ZkClient("linux121:2181,linux122:2181,linux123:2181");
         if(!zkClient.exists("/servers")){
             zkClient.createPersistent("/servers");
         }
    }
    //注册服务端信息到zk节点

    private void registerServerInfo(String ip,String port){
        //创建临时顺序节点
        final String path =
zkClient.createEphemeralSequential("/servers/server", ip +":"+port);
        System.out.println("---->>> 服务器注册成功，ip="+ip+";port ="+port+";节点路
径信息="+path);
    }


    public static void main(String[] args) {
        final ServerMain server = new ServerMain();
        server.connectZK();
        server.registerServerInfo(args[0],args[1] );

        //启动一个服务线程提供时间查询
        new TimeServer(Integer.parseInt(args[1])).start();
    }
}
```

服务端提供时间查询的线程类

```
package com.lagou.zk.test;


import java.io.IOException;
import java.io.OutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.Date;

public class TimeServer extends Thread {


    private int port=0;

    public TimeServer(int port) {
        this.port = port;
    }

    @Override
    public void run() {
     //启动serversocket监听一个端口
        try {
            final ServerSocket serverSocket = new ServerSocket(port);
           while(true){
               final Socket socket = serverSocket.accept();
               final OutputStream out = socket.getOutputStream();
               out.write(new Date().toString().getBytes());
```

```
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

client端

```
package com.lagou.zk.onoffline;

import org.IOItec.zkclient.IZkChildListener;
import org.IOItec.zkclient.ZkClient;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;

// 注册监听zk指定目录，
//维护自己本地一个servers信息，收到通知要进行更新
//发送时间查询请求并接受服务端返回的数据
public class Client {
    //获取zkclient
    ZkClient zkClient = null;

    //维护一个serversi 信息集合
    ArrayList<String> infos = new ArrayList<String>();

    private void connectZk() {
        // 创建zkclient
        zkClient = new ZkClient("linux121:2181,linux122:2181");
        //第一次获取服务器信息,所有的子节点
        final List<String> childs = zkClient.getChildren("/servers");
        for (String child : childs) {
            //存储着ip+port
            final Object o = zkClient.readData("/servers/" + child);
            infos.add(String.valueOf(o));
        }

        //对servers目录进行监听
        zkClient.subscribeChildChanges("/servers", new IZkChildListener() {
            public void handleChildChange(String s, List<String> children)
throws Exception {
                //接收到通知，说明节点发生了变化，client需要更新infos集合中的数据
                ArrayList<String> list = new ArrayList<String>();
                //遍历更新过后的所有节点信息
                for (String path : children) {
                    final Object o = zkClient.readData("/servers/" + path);
                    list.add(String.valueOf(o));
                }

                //最新数据覆盖老数据
```

```
                infos = list;
                System.out.println("--》接收到通知，最新服务器信息为：" + infos);
            }
        });
    }


    //发送时间查询的请求
    public void sendRequest() throws IOException {
        //目标服务器地址
        final Random random = new Random();
        final int i = random.nextInt(infos.size());
        final String ipPort = infos.get(i);
        final String[] arr = ipPort.split(":");

        //建立socket连接

        final Socket socket = new Socket(arr[0], Integer.parseInt(arr[1]));
        final OutputStream out = socket.getOutputStream();
        final InputStream in = socket.getInputStream();
        //发送数据
        out.write("query time".getBytes());
        out.flush();
        //接收返回结果
        final byte[] b = new byte[1024];
        in.read(b);//读取服务端返回数据
        System.out.println("client端接收到server:+" + ipPort + "+返回结果：" + new
String(b));


        //释放资源
        in.close();
        out.close();
        socket.close();
    }


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

        final Client client = new Client();
        client.connectZk();   //监听器逻辑
        while (true) {
            try {
                client.sendRequest(); //发送请求
            } catch (IOException e) {
                e.printStackTrace();
                try {
                    client.sendRequest();
                } catch (IOException e1) {
                    e1.printStackTrace();
                }
            }
            //每隔几秒中发送一次请求到服务端
            Thread.sleep(2000);
        }
    }
}
```

6.2 分布式锁

1. 什么是锁

- 在单机程序中，当存在多个线程可以同时改变某个变量（可变共享变量）时，为了保证线程安全(数据不能出现脏数据)就需要对变量或代码块做同步，使其在修改这种变量时能够串行执行消除并发修改变量。
- 对变量或者堆代码码块做同步本质上就是加锁。目的就是实现多个线程在一个时刻同一个代码块只能有一个线程可执行

2. 分布式锁

分布式的环境中会不会出现脏数据的情况呢？类似单机程序中线程安全的问题。观察下面的例子

上面的设计是存在线程安全问题

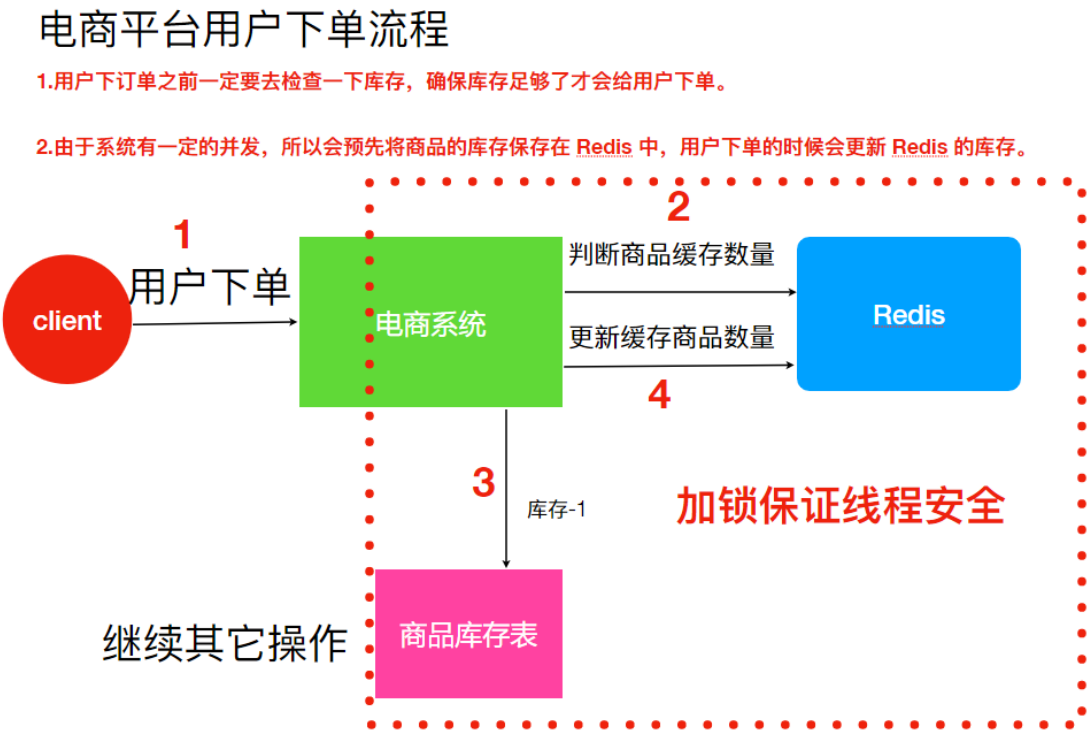
问题

假设Redis 里面的某个商品库存为 1；此时两个用户同时下单，其中一个下单请求执行到第 3 步，更新数据库的库存为 0，但是第 4 步还没有执行。

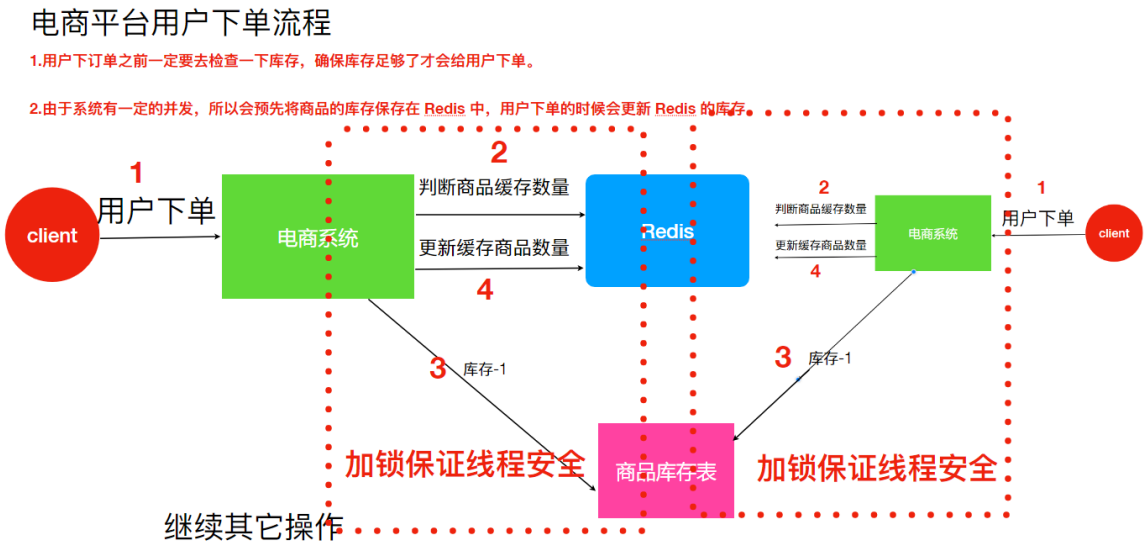
而另外一个用户下单执行到了第 2 步，发现库存还是 1，就继续执行第 3 步。但是商品库存已经为0，所以如果数据库没有限制就会出现超卖的问题。

解决方法

用锁把 2、3、4 步锁住，让他们执行完之后，另一个线程才能进来执行。



公司业务发展迅速，系统应对并发不断提高，解决方案是要增加一台机器，结果会出现更大的问题



假设有两个下单请求同时到来，分别由两个机器执行，那么这两个请求是可以同时执行了，依然存在超卖的问题。

因为如图所示系统是运行在两个不同的 JVM 里面，不同的机器上，增加的锁只对自己当前 JVM 里面的线程有效，对于其他 JVM 的线程是无效的。所以现在已经不是线程安全问题。需要保证两台机器加的锁是同一个锁，此时分布式锁就能解决该问题。

分布式锁的作用：在整个系统提供一个全局、唯一的锁，在分布式系统中每个系统在进行相关操作的时候需要获取到该锁，才能执行相应操作。

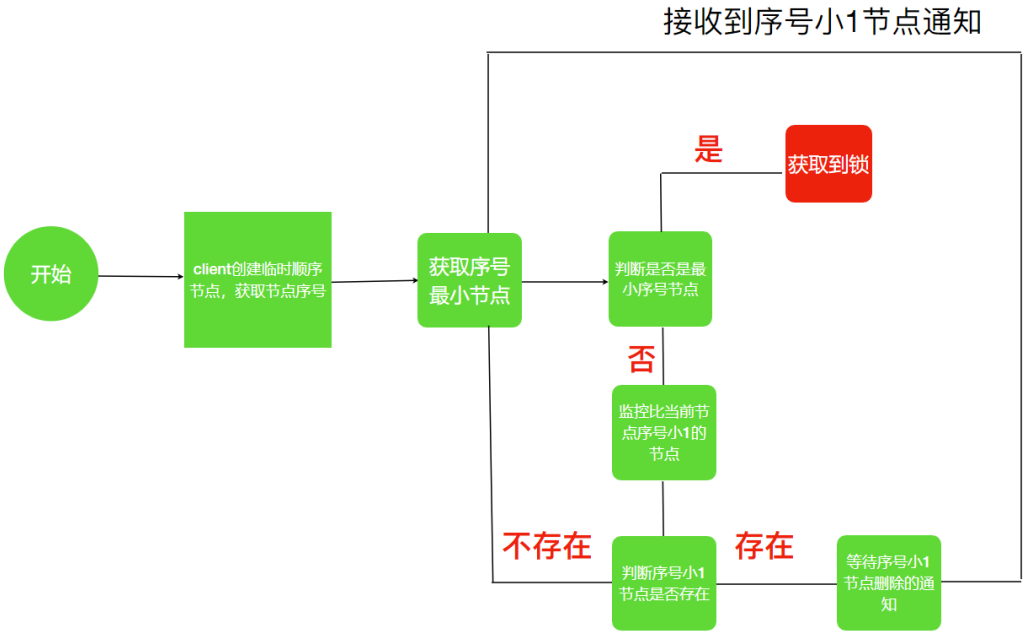
3 zk实现分布式锁

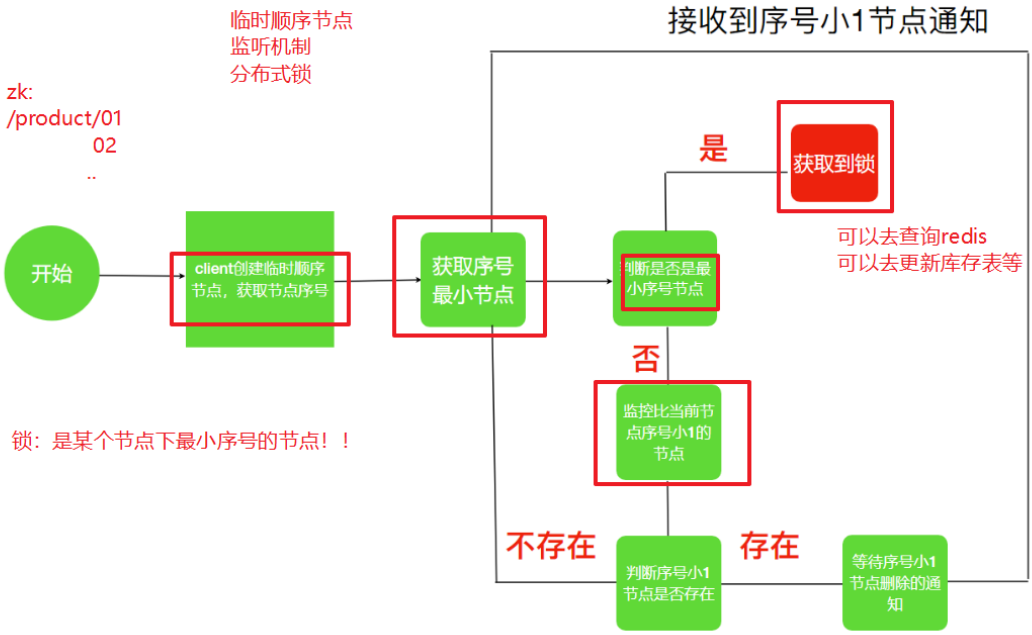
- 利用Zookeeper可以创建临时带序号节点的特性来实现一个分布式锁

实现思路

- 锁就是zk指定目录下序号最小的临时序列节点，多个系统的多个线程都要在此目录下创建临时的顺序节点，因为Zk会为我们保证节点的顺序性，所以可以利用节点的顺序进行锁的判断。
- 每个线程都是先创建临时顺序节点，然后获取当前目录下最小的节点(序号)，判断最小节点是不是当前节点，如果是那么获取锁成功，如果不是那么获取锁失败。
- 获取锁失败的线程获取当前节点上一个临时顺序节点，并对对此节点进行监听，当该节点删除的时候(上一个线程执行结束删除或者是掉线zk删除临时节点)这个线程会获取到通知，代表获取到了锁。

流程图





main方法

```
package com.lagou.zk.dislock;

//zk实现分布式锁
public class DisLockTest {
    public static void main(String[] args) {
        //使用10个线程模拟分布式环境
        for (int i = 0; i < 10; i++) {
            new Thread(new DisLockRunnable()).start();//启动线程
```

```
        }
    }

    static class DisLockRunnable implements Runnable {

        public void run() {
            //每个线程具体的任务，每个线程就是抢锁，
            final DisClient client = new DisClient();
            client.getDisLock();

            //模拟获取锁之后的其它动作
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            //释放锁
            client.deleteLock();
        }
    }
}
```

核心实现

```
package com.lagou.zk.dislock;

import org.IOItec.zkclient.IZkDataListener;
import org.IOItec.zkclient.ZkClient;

import java.util.Collections;
import java.util.List;
import java.util.concurrent.CountDownLatch;

//抢锁
//1. 去zk创建临时序列节点，并获取到序号
//2. 判断自己创建节点序号是否是当前节点最小序号，如果是则获取锁
//执行相关操作，最后要释放锁
//3. 不是最小节点，当前线程需要等待，等待你的前一个序号的节点
//被删除，然后再次判断自己是否是最小节点。。。
public class DisClient {

    public DisClient() {
        //初始化zk的/distrilocl节点,会出现线程安全问题
        synchronized (DisClient.class){
            if (!zkClient.exists("/distrilock")) {
                zkClient.createPersistent("/distrilock");
            }
        }

    }

    //前一个节点
    String beforNodePath;
```

```
    String currentNoePath;
    //获取到zkClient
    private ZkClient zkClient = new ZkClient("linux121:2181,linux122:2181");
    //把抢锁过程为量部分，一部分是创建节点，比较序号，另一部分是等待锁

    //完整获取锁方法
    public void getDisLock() {
        //获取到当前线程名称
        final String threadName = Thread.currentThread().getName();
        //首先调用tryGetLock
        if (tryGetLock()) {
            //说明获取到锁
            System.out.println(threadName + ":获取到了锁");
        } else {
            // 没有获取到锁,
            System.out.println(threadName + ":获取锁失败,进入等待状态");
            waitForLock();
            //递归获取锁
            getDisLock();
        }

    }

    CountDownLatch countDownLatch = null;

    //尝试获取锁
    public boolean tryGetLock() {
        //创建临时顺序节点,/distrilock/序号
        if (null == currentNoePath || "".equals(currentNoePath)) {
            currentNoePath = zkClient.createEphemeralSequential("/distrilock/",
"lock");
        }
        //获取到/distrilock下所有的子节点
        final List<String> childs = zkClient.getChildren("/distrilock");
        //对节点信息进行排序
        Collections.sort(childs); //默认是升序
        final String minNode = childs.get(0);
        //判断自己创建节点是否与最小序号一致
        if (currentNoePath.equals("/distrilock/" + minNode)) {
            //说明当前线程创建的就是序号最小节点
            return true;
        } else {
            //说明最小节点不是自己创建，要监控自己当前节点序号前一个的节点
            final int i = Collections.binarySearch(childs,
currentNoePath.substring("/distrilock/".length()));
            //前一个(lastNodeChild是不包括父节点)
            String lastNodeChild = childs.get(i - 1);
            beforNodePath = "/distrilock/" + lastNodeChild;
        }

        return false;
    }

    //等待之前节点释放锁,如何判断锁被释放，需要唤醒线程继续尝试tryGetLock
    public void waitForLock() {

        //准备一个监听器
        final IZkDataListener iZkDataListener = new IZkDataListener() {
```

```
            public void handleDataChange(String s, Object o) throws Exception {

            }

            //删除
            public void handleDataDeleted(String s) throws Exception {
                //提醒当前线程再次获取锁
                countDownLatch.countDown();//把值减1变为0，唤醒之前await线程
            }
        };
        //监控前一个节点
        zkClient.subscribeDataChanges(beforNodePath, iZkDataListener);

        //在监听的通知没来之前，该线程应该是等待状态，先判断一次上一个节点是否还存在
        if (zkClient.exists(beforNodePath)) {
            //开始等待,CountDownLatch:线程同步计数器
            countDownLatch = new CountDownLatch(1);
            try {
                countDownLatch.await();//阻塞，countDownLatch值变为0
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

        //解除监听
        zkClient.unsubscribeDataChanges(beforNodePath, iZkDataListener);
    }


    //释放锁
    public void deleteLock() {
        if (zkClient != null) {
            zkClient.delete(currentNoePath);
            zkClient.close();
        }
    }
}
```

分布式锁的实现可以是 Redis、Zookeeper，相对来说生产环境如果使用分布式锁可以考虑使用Redis实现而非Zk。

7 Hadoop HA

7.1 HA 概述

1. 所谓HA（High Available），即高可用（7*24小时不中断服务）。
2. 实现高可用最关键的策略是消除单点故障。Hadoop-HA严格来说应该分成各个组件的HA机制：HDFS的HA和YARN的HA。
3. Hadoop2.0之前，在HDFS集群中NameNode存在单点故障（SPOF）。
4. NameNode主要在以下两个方面影响HDFS集群

NameNode机器发生意外，如宕机，集群将无法使用，直到管理员重启

NameNode机器需要升级，包括软件、硬件升级，此时集群也将无法使用

HDFS HA功能通过配置Active/Standby两个NameNodes实现在集群中对NameNode的热备来解决上述问题。如果出现故障，如机器崩溃或机器需要升级维护，这时可通过此种方式将NameNode很快的切换到另外一台机器。

7.2 HDFS-HA 工作机制

通过双NameNode消除单点故障（Active/Standby）

2.1 HDFS-HA工作要点

1. 元数据管理方式需要改变

   内存中各自保存一份元数据；

   Edits日志只有Active状态的NameNode节点可以做写操作；

   两个NameNode都可以读取Edits；

   共享的Edits放在一个共享存储中管理（qjournal和NFS两个主流实现）；

2. 需要一个状态管理功能模块

   实现了一个zkfailover，常驻在每一个namenode所在的节点，每一个zkfailover负责监控自己所在NameNode节点，利用zk进行状态标识，当需要进行状态切换时，由zkfailover来负责切换，切换时需要防止brain split现象的发生（集群中出现两个Active的Namenode）。

3. 必须保证两个NameNode之间能够ssh无密码登录

4. 隔离（Fence），即同一时刻仅仅有一个NameNode对外提供服务

2.2 HDFS-HA工作机制

配置部署HDFS-HA进行自动故障转移。自动故障转移为HDFS部署增加了两个新组件：ZooKeeper和ZKFailoverController（ZKFC）进程，ZooKeeper是维护少量协调数据，通知客户端这些数据的改变和监视客户端故障的高可用服务。HA的自动故障转移依赖于ZooKeeper的以下功能：

- 故障检测

  集群中的每个NameNode在ZooKeeper中维护了一个临时会话，如果机器崩溃，ZooKeeper中的会话将终止，ZooKeeper通知另一个NameNode需要触发故障转移。

- 现役NameNode选择

  ZooKeeper提供了一个简单的机制用于唯一的选择一个节点为active状态。如果目前现役NameNode崩溃，另一个节点可能从ZooKeeper获得特殊的排外锁以表明它应该成为现役NameNode。

ZKFC是自动故障转移中的另一个新组件，是ZooKeeper的客户端，也监视和管理NameNode的状态。每个运行NameNode的主机也运行了一个ZKFC进程，ZKFC负责：

- 健康监测

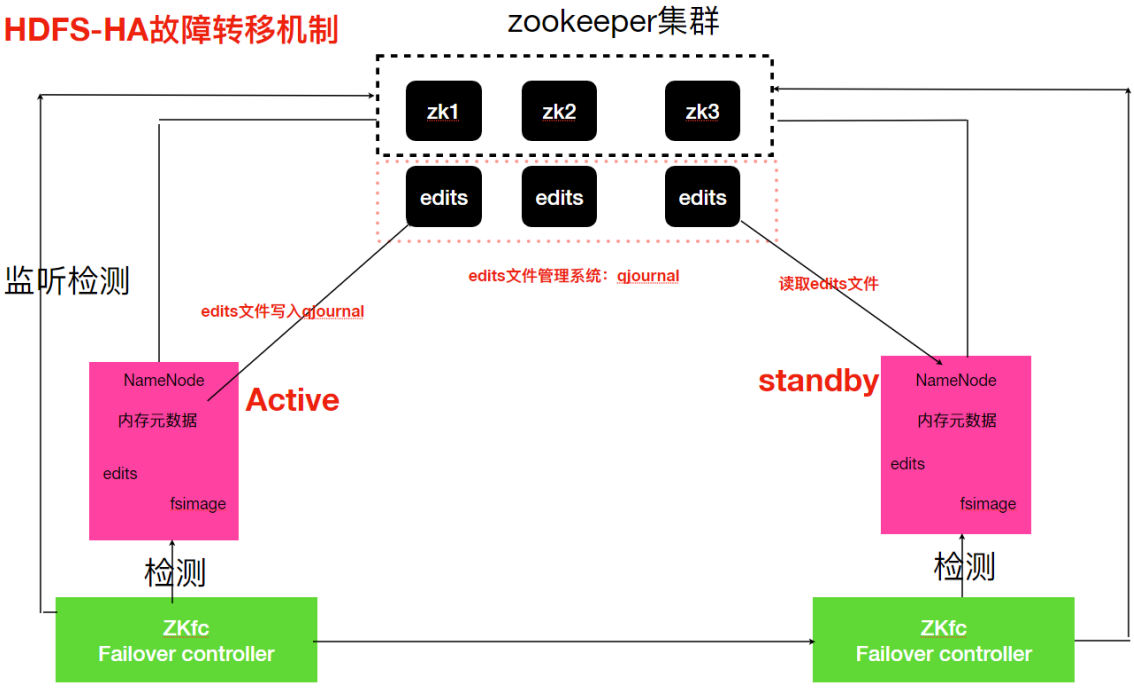
  ZKFC使用一个健康检查命令定期地ping与之在相同主机的NameNode，只要该NameNode及时地回复健康状态，ZKFC认为该节点是健康的。如果该节点崩溃，冻结或进入不健康状态，健康监测器标识该节点为非健康的。

- ZooKeeper会话管理

  当本地NameNode是健康的，ZKFC保持一个在ZooKeeper中打开的会话。如果本地NameNode处于active状态，ZKFC也保持一个特殊的znode锁，该锁使用了ZooKeeper对短暂节点的支持，如果会话终止，锁节点将自动删除。

- 基于ZooKeeper的选择

如果本地NameNode是健康的，且ZKFC发现没有其它的节点当前持有znode锁，它将为自己获取该锁。如果成功，则它已经赢得了选择，并负责运行故障转移进程以使它的本地NameNode为Active。故障转移进程与前面描述的手动故障转移相似，首先如果必要保护之前的现役NameNode，然后本地NameNode转换为Active状态。



7.3 HDFS-HA集群配置

```
https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithQJM.html
```

3.1 环境准备

1. 修改IP
2. 修改主机名及主机名和IP地址的映射
3. 关闭防火墙
4. ssh免密登录
5. 安装JDK，配置环境变量等

3.2 集群规划

linux121	linux122	linux123
NameNode	NameNode	
JournalNode	JournalNode	JournalNode
DataNode	DataNode	DataNode
ZK	ZK	ZK
	ResourceManager	
NodeManager	NodeManager	NodeManager

3.3 启动Zookeeper集群

启动zookeeper集群

```
zk.sh start
```

查看状态

```
zk.sh status
```

3.4 配置HDFS-HA集群

1. 停止原先HDFS集群

   ```
   stop-dfs.sh
   ```

2. 在所有节点，/opt/lagou/servers目录下创建一个ha文件夹

   ```
   mkdir /opt/lagou/servers/ha
   ```

3. 将/opt/lagou/servers/目录下的 hadoop-2.9.2拷贝到ha目录下

   ```
   cp -r hadoop-2.9.2 ha
   ```

4. 删除原集群data目录

   ```
   rm -rf /opt/lagou/servers/ha/hadoop-2.9.2/data
   ```

5. 配置hdfs-site.xml

   ```
   <property>
       <name>dfs.nameservices</name>
       <value>lagoucluster</value>
   </property>
   <property>
       <name>dfs.ha.namenodes.lagoucluster</name>
       <value>nn1,nn2</value>
   </property>
   <property>
       <name>dfs.namenode.rpc-address.lagoucluster.nn1</name>
       <value>linux121:9000</value>
   </property>
   <property>
       <name>dfs.namenode.rpc-address.lagoucluster.nn2</name>
       <value>linux122:9000</value>
   </property>
   <property>
       <name>dfs.namenode.http-address.lagoucluster.nn1</name>
       <value>linux121:50070</value>
   </property>
   <property>
       <name>dfs.namenode.http-address.lagoucluster.nn2</name>
   ```

```
    <value>linux122:50070</value>
</property>
<property>
    <name>dfs.namenode.shared.edits.dir</name>

<value>qjournal://linux121:8485;linux122:8485;linux123:8485/lagou</value>
</property>

<property>
    <name>dfs.client.failover.proxy.provider.lagoucluster</name>

<value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProv
ider</value>
</property>
<property>
    <name>dfs.ha.fencing.methods</name>
    <value>sshfence</value>
</property>
<property>
    <name>dfs.ha.fencing.ssh.private-key-files</name>
    <value>/root/.ssh/id_rsa</value>
</property>
<property>
    <name>dfs.journalnode.edits.dir</name>
    <value>/opt/journalnode</value>
</property>
<property>
    <name>dfs.ha.automatic-failover.enabled</name>
    <value>true</value>
</property>
```

6. 配置core-site.xml

```
<property>
    <name>fs.defaultFS</name>
    <value>hdfs://lagoucluster</value>
</property>
<property>
    <name>hadoop.tmp.dir</name>
    <value>/opt/lagou/servers/ha/hadoop-2.9.2/data/tmp</value>
</property>
<property>
    <name>ha.zookeeper.quorum</name>
    <value>linux121:2181,linux122:2181,linux123:2181</value>
</property>
```

7. 拷贝配置好的hadoop环境到其他节点

```
rsync-script /opt/lagou/servers/ha/hadoop-2.9.2/
```

3.5 启动HDFS-HA集群

1. 在各个JournalNode节点上，输入以下命令启动journalnode服务(去往HA安装目录，不要使用环境变量中命令)

```
/opt/lagou/servers/ha/hadoop-2.9.2/sbin/hadoop-daemon.sh start journalnode
```

2. 在[nn1]上，对其进行格式化，并启动

```
/opt/lagou/servers/ha/hadoop-2.9.2/bin/hdfs namenode -format

/opt/lagou/servers/ha/hadoop-2.9.2/sbin/hadoop-daemon.sh start namenode
```

3. 在[nn2]上，同步nn1的元数据信息

```
/opt/lagou/servers/ha/hadoop-2.9.2/bin/hdfs namenode -bootstrapStandby
```

4. 在[nn1]上初始化zkfc

```
/opt/lagou/servers/ha/hadoop-2.9.2/bin/hdfs zkfc -formatZK
```

5. 在[nn1]上，启动集群

```
/opt/lagou/servers/ha/hadoop-2.9.2/sbin/start-dfs.sh
```

6. 验证

将Active NameNode进程kill

kill -9 namenode的进程id

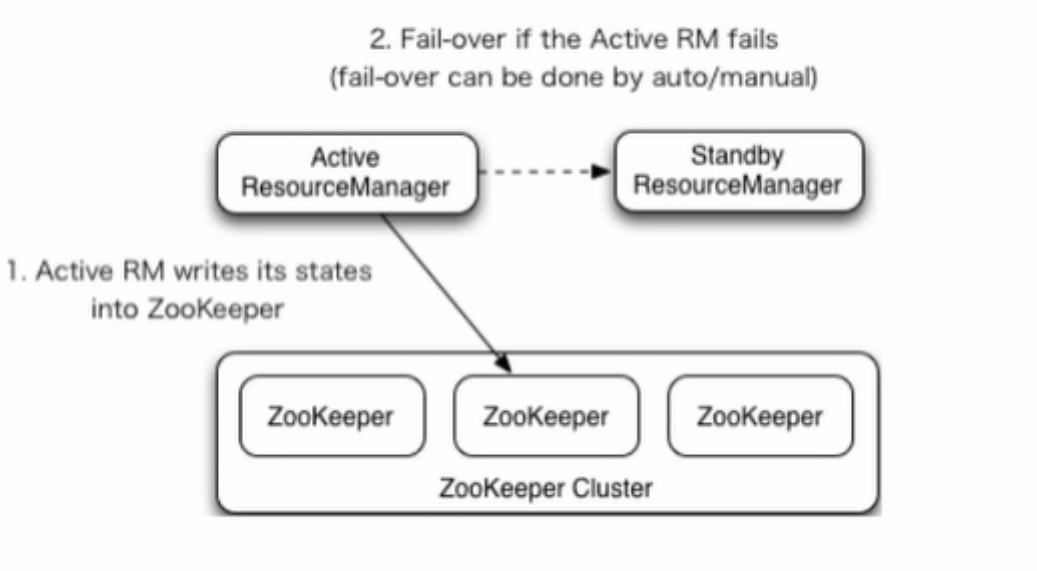
7.4 YARN-HA配置

4.1 YARN-HA工作机制

1. 官方文档

```
https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/ResourceManagerHA.html
```

2. YARN-HA工作机制，如图



4.2 配置YARN-HA集群

1. 环境准备

- 修改IP
- 修改主机名及主机名和IP地址的映射
- 关闭防火墙
- ssh免密登录
- 安装JDK，配置环境变量等
- 配置Zookeeper集群

2. 规划集群

linux121	linux122	linux123
NameNode	NameNode	
JournalNode	JournalNode	JournalNode
DataNode	DataNode	DataNode
ZK	ZK	ZK
	ResourceManager	ResourceManager
NodeManager	NodeManager	NodeManager

3. 具体配置
4. yarn-site.xml

```
<configuration>

    <property>
        <name>yarn.nodemanager.aux-services</name>
        <value>mapreduce_shuffle</value>
    </property>

    <!--启用resourcemanager ha-->
    <property>
        <name>yarn.resourcemanager.ha.enabled</name>
        <value>true</value>
    </property>

    <!--声明两台resourcemanager的地址-->
    <property>
        <name>yarn.resourcemanager.cluster-id</name>
        <value>cluster-yarn</value>
    </property>

    <property>
        <name>yarn.resourcemanager.ha.rm-ids</name>
        <value>rm1,rm2</value>
    </property>

    <property>
        <name>yarn.resourcemanager.hostname.rm1</name>
        <value>linux122</value>
```

```
    </property>

    <property>
        <name>yarn.resourcemanager.hostname.rm2</name>
        <value>linux123</value>
    </property>

    <!--指定zookeeper集群的地址-->
    <property>
        <name>yarn.resourcemanager.zk-address</name>
        <value>linux121:2181,linux122:2181,linux123:2181</value>
    </property>

    <!--启用自动恢复-->
    <property>
        <name>yarn.resourcemanager.recovery.enabled</name>
        <value>true</value>
    </property>

    <!--指定resourcemanager的状态信息存储在zookeeper集群-->
    <property>
        <name>yarn.resourcemanager.store.class</name>
<value>org.apache.hadoop.yarn.server.resourcemanager.recovery.ZKRMStateStore</va
lue>
</property>
```

2. 同步更新其他节点的配置信息

```
rsync-script yarn-site.xml
```

3. 启动hdfs

```
sbin/start-yarn.sh
```