day02 ZooKeeper

一、内容回顾

- Hadoop的安装部署
  - 单机模式、伪分布式模式【了解】
  - 完全分布式模式【掌握】
- HDFS的Shell操作
- HDFS的块
- HDFS的体系结构
  - NameNode
  - DataNode
  - SecondaryNameNode
  - fsimage & edits
- 开机启动流程
- 集群安全模式
- HDFS的读写流程

二、教学目标

2.1. 教学重点

- ZooKeeper的命令
- ZooKeeper的数据模型
- ZooKeeper的节点类型

- ZooKeeper的选举制度
- ZooKeeper的监听原理
- ZooKeeper的写数据流程
- HA集群搭建

2.2. 教学难点

- ZooKeeper的选举制度
- ZooKeeper的监听原理
- HA集群搭建

三、教学导读

四、教学内容

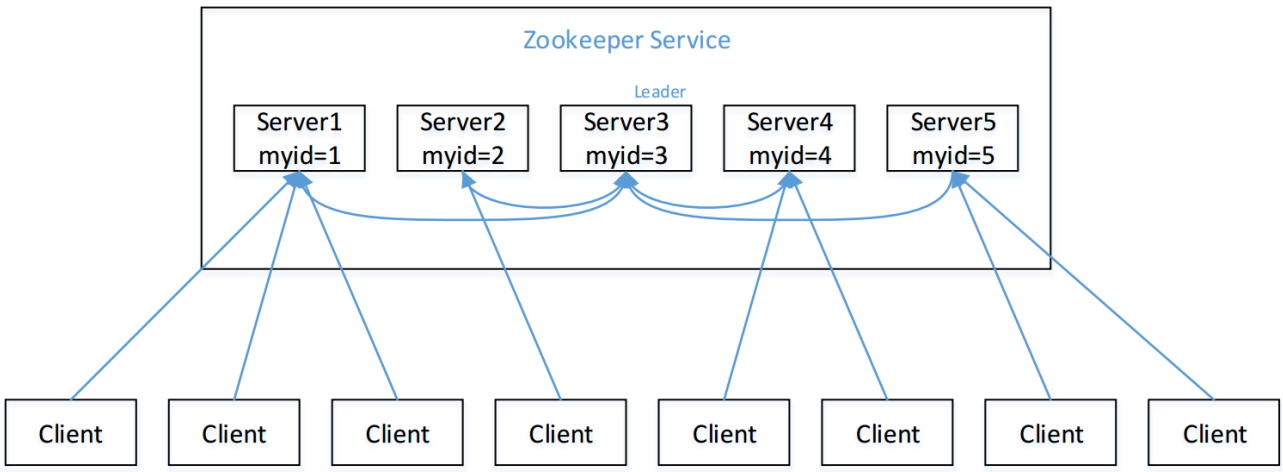
4.1. ZooKeeper概述【了解】

4.1.1. ZooKeeper是什么

1. zookeeper是一个为分布式应用程序提供的一个分布式开源协调服务框架。是Google的Chubby的一个开源实现，是Hadoop和Hbase的重要组件。主要用于解决分布式集群中应用系统的一致性问题。
2. 提供了基于类似Unix系统的目录节点树方式的数据存储。
3. 可用于维护和监控存储的数据的状态的变化，通过监控这些数据状态的变化，从而达到基于数据的集群管理
4. 提供了一组原语(机器指令)，提供了java和c语言的接口

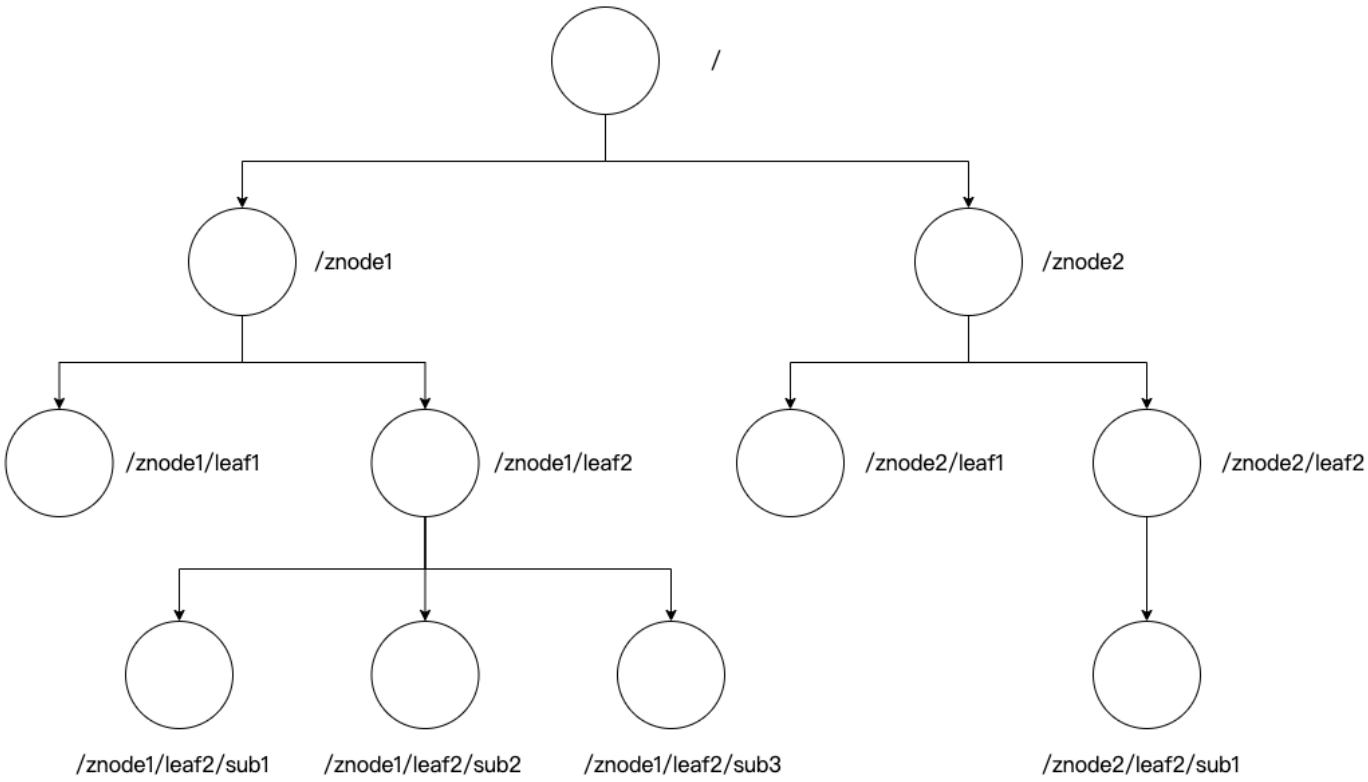
4.1.2. Zookeeper的特点

1. 也是一个分布式集群，一个领导者(leader),多个跟随者(follower).
2. 集群中只要有半数以上的节点存活，Zookeeper集群就能正常服务。
3. 全局数据一致性：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。
4. 更新请求按顺序进行：来自同一个client的更新请求按其发送顺序依次执行
5. 数据更新的原子性：一次数据的更新要么成功，要么失败
6. 数据的实时性：在一定时间范围内，client能读到最新数据。



4.1.3. Zookeeper的数据模型

Zookeeper的数据模型采用的与Unix文件系统类似的层次化的树形结构。我们可以将其理解为一个具有高可用特征的文件系统。这个文件系统中没有文件和目录，而是统一使用"节点"(node)的概念，称之为znode。znode既可以作为保存数据的容器(如同文件),也可以作为保存其他znode的容器(如同目录)。所有的znode构成了一个层次化的命名空间。



- Zookeeper被设计用来实现协调服务，这类服务通常使用小数据文件，而不是用于大容量数据存储，因此一个znode能存储的数据被限制在1MB以内，
- 每个znode都可以通过其路径唯一标识。

4.1.4. Zookeeper的应用场景

```
1．统一配置管理
2．统一集群管理
3．服务器节点动态上下线感知
4．软负载均衡等
5．分布式锁
6．分布式队列
```

4.1.5. zookeeper的作用

- 负责提供阶段的监听注册作用
- 负责接收用户的心跳,进行通信,感知用户的状态
- 负责为用户提供注册,查找信息
- 负责负载均衡

注意:

- 如果现在再新增一个服务器,要先完成在zookeeper中的注册,这时我就知道可以有一个新的服务器可用,分配任务时进行较平均分配(负载均衡)
- zookeeper可以单机版,但是他的压力也很大,所以一般zookeeper也是一个分布式的

4.2. Zookeeper的安装【掌握】

4.2.1. 环境变量的配置

1. 上传安装包

```
使用MobaXterm、FinalShell或者使用scp将安装包apache-zookeeper-3.6.3-bin.tar.gz上传到/root/softwares下
```

2. 解压安装

```
[root@qianfeng01 ~]# tar -zxvf apache-zookeeper-3.6.3-bin.tar.gz -C /usr/local
```

3. 更名

```
[root@qianfeng01 ~]# cd /usr/local/
[root@qianfeng01 local]# mv apache-zookeeper-3.6.3-bin zookeeper-3.6.3
```

4. 配置环境变量

```
[root@qianfeng01 local]# vim /etc/profile

export ZOOKEEPER_HOME=/usr/local/zookeeper-3.6.3
export PATH=$PATH:$ZOOKEEPER_HOME/bin
```

5. 使环境变量生效

```
[root@qianfeng01 local]# source /etc/profile
```

4.2.2. 集群模式的配置

4.2.2.1. Zookeeper的服务进程布局

```
qianfeng01    QuorumPeerMain
qianfeng02    QuorumPeerMain
qianfeng03    QuorumPeerMain
```

4.2.2.2. 修改zoo.cfg文件

```
[root@qianfeng01 local]# cd $ZOOKEEPER_HOME/conf/
# 复制出zoo.cfg文件
[root@qianfeng01 conf]# cp  zoo_sample.cfg  zoo.cfg
[root@qianfeng01 conf]# vim zoo.cfg
tickTime=2000        # 定义的时间单元(单位毫秒)，下面的两个值都是tickTime的倍数。
initLimit=10         # follower连接并同步leader的初始化连接时间。
syncLimit=5          # 心跳机制的时间(正常情况下的请求和应答的时间)
dataDir=/usr/local/zookeeper-3.6.3/zkData        # 修改zookeeper的存储路径，zkData目录一会要创建出来
clientPort=2181                  # 客户端连接服务器的port
server.1=qianfeng01:2888:3888              # 添加三个服务器节点
server.2=qianfeng02:2888:3888
server.3=qianfeng03:2888:3888

# 解析 Server.id=ip:port1:port2
# id:    服务器的id号，对应zkData/myid文件内的数字
# ip:      服务器的ip地址
# port1:  follower与leader交互的port
```

```
# port2:  选举期间使用的port

# 注意：此配置文件中，不支持汉字注释
```

4.2.2.3. 创建myid

```
# 在$ZOOKEEPER_HOME/zkData/目录下添加myid文件，内容为server的id号
[root@qianfeng01 conf]# cd $ZOOKEEPER_HOME
[root@qianfeng01 zookeeper]# mkdir zkData
[root@qianfeng01 zookeeper]# cd zkData
[root@qianfeng01 zkData]# echo "1" > myid
```

4.2.2.4. 搭建其他两个server节点

1. 使用scp命令将zookeeper环境 复制到qianfeng02和qianfeng03中

```
[root@qianfeng01 zkData]# cd /usr/local
[root@qianfeng01 local]# scp -r zookeeper-3.6.3 qianfeng02:/usr/local
[root@qianfeng01 local]# scp -r zookeeper-3.6.3 qianfeng03:/usr/local
```

2. 使用scp命令拷贝/etc/profile到两台机器上(别忘记source一下)

```
[root@qianfeng01 local]# scp /etc/profile qianfeng02:/etc/
[root@qianfeng01 local]# scp /etc/profile qianfeng03:/etc/
```

3. 修改qianfeng02的myid文件的内容为2

```
[root@qianfeng01 ~]# ssh qianfeng02
[root@qianfeng02 ~]# echo "2" > $ZOOKEEPER_HOME/zkData/myid
```

4. 修改qianfeng03的myid文件的内容为3

```
[root@qianfeng02 ~]# ssh qianfeng03
[root@qianfeng03 ~]# echo "3" > $ZOOKEEPER_HOME/zkData/myid
```

4.2.2.5. 启动服务

1. 在每一个节点启动zkServer的服务

```
[root@qianfeng01 ~]# zkServer.sh start
[root@qianfeng02 ~]# zkServer.sh start
[root@qianfeng03 ~]# zkServer.sh start

# 查看状态
[root@qianfeng01 ~]# zkServer.sh status
[root@qianfeng02 ~]# zkServer.sh status
[root@qianfeng03 ~]# zkServer.sh status
```

2. 便捷启动脚本

```
#!/bin/bash
```

```
2
3  # 获取参数
4  COMMAND=$1
5  if [ ! $COMMAND ]; then
6      echo "please input your option in [start | stop
   | status]"
7      exit -1
8  fi
9  if [ $COMMAND != "start" -a $COMMAND != "stop" -a
   $COMMAND != "status" ]; then
10     echo "please input your option in [start | stop
   | status]"
11     exit -1
12 fi
13
14 # 所有的服务器
15 HOSTS=( qianfeng01 qianfeng02 qianfeng03 )
16 for HOST in ${HOSTS[*]}
17 do
18     ssh -T $HOST << TERMINATOR
19     echo "---------- $HOST ----------"
20     zkServer.sh $COMMAND 2> /dev/null | grep -ivh
   SSL
21     exit
22 TERMINATOR
23 done
```

4.3. ZooKeeper的节点操作【掌握】

4.3.1. ZooKeeper的节点类型

ZooKeeper其实也是一个分布式集群，其中维护了一个目录树结构，在这个目录树中，组成的部分是一个个的节点。ZooKeeper的节点可以大致分为两种类型: **短暂类型** 和 **持久类型**

- 短暂类型ephemeral: 客户端和服务器断开后，创建的节点自己删除。
- 持久类型persistent: 客户端和服务器断开后，创建的节点不删除(默认情况)。

节点类型	描述信息
EPHEMERAL	临时节点，在会话结束后自动被删除。
EPHEMERAL_SEQUENTIAL	临时顺序节点，在会话结束后会自动被删除。会在给定的path节点名称后添加一个序列号。
PERSISTENT	永久节点，在会话结束后不会被自动删除。
PERSISTENT_SEQUENTIAL	永久顺序节点，在会话结束后不会被自动删除。会在给定的path节点名称后添加一个序列号。

4.3.2. ZooKeeper的Shell操作

4.3.2.1. 打开Shell客户端

1. 连接到当前节点的Server服务

```
[root@qianfeng01 ~]# zkCli.sh
```

2. 连接到其他节点的Server服务

```
[root@qianfeng01 ~]# zkCli.sh -server qianfeng02:2181
```

4.3.2.2. Shell操作

ls

```
作用：查看某个节点下的子节点
选项：
    -s 查看具体信息，包括time、version等信息
注意事项：需要使用绝对路径查看
示例：
    ls /
    ls /zookeeper
    ls -s /zookeeper/config
```

create

```
作用：创建一个节点，可以设置节点的初始内容
选项：
    -e：设置短暂类型节点
    -s：设置顺序节点
示例：
  create /test
  create /test2 "content message"
  create -e /test3 "content message"
  create -e -s /test "content message"
```

get

```
作用：获取节点存储的值
选项：
    -s：同时获取版本描述信息，例如：time、version等
示例：
  ls /zookeeper/config
  ls -s /zookeeper/config
```

> hehe 节点数据信息 cZxid = 0x800000002 节点创建时的zxid ctime = Thu May 09 03:41:15 CST 2019 节点创建的时间 mZxid = 0x800000002 对应节点最近一次修改的时间,与子节点无关 mtime = Thu May 09 03:41:15 CST 2019 节点最近一次更新的时间 pZxid = 0x800000002 对应节点与子节点(或者子节点)的修改的时间,与孙子节点无关 cversion = 0 子节点数据更新次数 dataVersion = 0 本节点数据更新次数 aclVersion = 0 节点授权信息(ACL)的更新次数 ephemeralOwner = 0x0 如果该节点为临时节点,ephemeralOwner值表示与该节点绑定的session id. 如果该节点不是临时节点,ephemeralOwner值为0 dataLength = 4 节点的数据长度 numChildren = 0 子节点的个数

set

```
作用：设置节点存储的值
示例：
  set /test "content message"
```

delete

```
作用：删除节点，只能删除空节点，即没有子节点的节点
示例：
  delete /test
```

deleteAll

```
作用：删除节点，可以递归删除所有的子节点
示例：
  deleteAll /test
```

addWatch

```
作用：监听节点，当这个节点发生变化（内容、创建、删除）会得到通知
示例：
  addWatch /test
```

removewatches

```
作用：移除对节点的监听
示例：
  removewatches /test
```

quit

```
作用：退出客户端
```

4.3.3. ZooKeeper的API操作

4.3.3.1. pom依赖

```
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
</dependency>
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.6.3</version>
</dependency>
```

4.3.3.2. 初始化ZooKeeper客户端对象

```
package com.qianfeng.zk;

import org.apache.zookeeper.*;
import org.apache.zookeeper.data.Stat;
import org.junit.Before;
import org.junit.Test;

import java.io.IOException;
import java.util.List;

/**
 * ZooKeeper的API操作
 *
 * @author 千锋大数据教研院-章鱼哥
 * @company 北京千锋互联科技有限公司
```

```
 */
public class ZkAPI {
    // Zookeeper客户端对象
    private ZooKeeper zkCli;

    @Before
    public void init() throws IOException, InterruptedException {
        // 连接到ZooKeeper的Server端
        String connectString = "qianfeng01:2181,qianfeng02:2181,qianfeng03:2181";
        // 连接超时时间
        int sessionTimeout = 5000;

        // 初始化一个ZooKeeper客户端实例，需要参数：服务端、连接超时时间、观察者做回调
        zkCli = new ZooKeeper(connectString, sessionTimeout, new Watcher() {
            @Override
            public void process(WatchedEvent watchedEvent) {
                // 暂不做任何处理
            }
        });
    }
}
```

4.3.3.3. 创建节点（同步）

> 注意: 以下的API操作，需要使用到zkCli对象。在上述的初始化部分已经完成了对zkCli对象的初始化。后面的所有操作，只需要将方法粘贴到ZkAPI类中即可。需要导入的包，也在上方的初始化部分导入完成了。

```java
// ACL权限类型：
//      OPEN_ACL_UNSAFE：完全开放的ACL，任何连接的客户端都可以操作该节点
//      CREATOR_ALL_ACL：只有创建者才有ACL权限
//      READ_ACL_UNSAFE：只能读取ACL
//
// CreateMode：
//      EPHEMERAL：临时型
//      EPHEMERAL_SEQUENTIAL：临时顺序型
//      PERSISTENT：永久型
//      PERSISTENT_SEQUENTIAL：永久顺序型
//
// 同步创建节点，遇到不正常的情况直接抛出异常。
@Test
public void createNode() throws InterruptedException, KeeperException {
    zkCli.create("/ApiNode1", "ApiContent1".getBytes(), ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT);
}
```

4.3.3.4. 创建节点（异步）

```java
@Test
public void createNodeAsync() throws InterruptedException {
    zkCli.create("/ApiNode2", "ApiContent2".getBytes(), ZooDefs.Ids.OPEN_ACL_UNSAFE, CreateMode.PERSISTENT, new CreateNodeCallBack(), "");
    // 因为是异步操作，如果不等待一下，操作直接就结束了，来不及等到回调事件的触发
    Thread.sleep(Integer.MAX_VALUE);
```

```
}

private static class CreateNodeCallBack implements AsyncCallback.StringCallback {
    /**
     * 异步创建节点回调方法
     * @param i 服务端响应代码：0 => 成功，-4 => 客户端与服务端连接断开，-110 => 指定节点已存在，-112 => 会话已过期
     * @param s 创建的节点路径
     * @param o 创建节点时传递的ctx
     * @param s1 在ZK上实际创建的节点名（针对顺序节点）
     */
    @Override
    public void processResult(int i, String s, Object o, String s1) {
        System.out.println("i = " + i + ", s = " + s + ", o = " + o + ", s1 = " + s1);
        switch (i) {
            case 0:
                System.out.println("节点创建成功");
                break;
            case -4:
                System.out.println("客户端与服务端连接已断开");
                break;
            case -110:
                System.out.println("指定节点已存在");
                break;
            case -112:
                System.out.println("会话已过期");
                break;
        }
    }
```

```
}
```

4.3.3.5. 删除节点（同步）

```
// 同步删除，遇到无法正常删除的情况，直接异常
@Test
public void deleteNode() throws Exception {
    zkCli.delete("/ApiNode1", -1);
}
```

4.3.3.6. 删除节点（异步）

```
@Test
public void deleteNodeAsync() throws Exception {
    zkCli.delete("/ApiNode1", -1, new
DeleteNodeCallBack(), "");
    Thread.sleep(Integer.MAX_VALUE);
}

/**
 * 删除节点回调
 */
private static class DeleteNodeCallBack implements
AsyncCallback.VoidCallback {
    /**
     * 异步删除节点回调方法
     * @param rc 服务端响应代码：0 => 成功，-4 => 客户端与服
务端连接断开，-110 => 指定节点已存在，-112 => 会话已过期
     * @param path 删除的节点路径
     * @param ctx 删除节点时传递的ctx
```

```
16          */
17         @Override
18         public void processResult(int rc, String path,
    Object ctx) {
19             System.out.println("删除结果：rc=" + rc + ",
    path=" + path + ", ctx=" + ctx);
20             switch (rc) {
21                 case 0:
22                     System.out.println("节点删除成功");
23                     break;
24                 case -4:
25                     System.out.println("客户端与服务端连接已断
    开");
26                     break;
27                 case -112:
28                     System.out.println("会话已过期");
29                     break;
30                 default:
31                     System.out.println("服务端响应码" + rc +
    "未知");
32                     break;
33             }
34         }
35     }
```

4.3.3.7. 修改节点内容（同步）

```
1  @Test
2  public void setNode() throws InterruptedException,
   KeeperException {
3      zkCli.setData("/ApiNode1", "hello".getBytes(), -1);
4  }
```

4.3.3.8. 修改节点内容（异步）

```
@Test
public void setNodeAsync() throws Exception {
    zkCli.setData("/ApiNode2", "hello".getBytes(), -1, new SetNodeCallBack(), "");
    Thread.sleep(Integer.MAX_VALUE);
}

private static class SetNodeCallBack implements AsyncCallback.StatCallback {
    /**
     * @param rc 服务端响应代码: 0 => 成功, -4 => 客户端与服务端连接断开, -110 => 指定节点已存在, -112 => 会话已过期
     * @param path 修改的节点路径
     * @param ctx  修改节点时传递的ctx
     * @param stat 节点状态, 由服务器端响应的新stat替换
     */
    @Override
    public void processResult(int rc, String path, Object ctx, Stat stat) {
        switch (rc) {
            case 0:
                System.out.println("节点数据设置成功");
                break;
            case -4:
                System.out.println("客户端与服务端连接已断开");
                break;
            case -112:
                System.out.println("会话已过期");
```

```
                break;
            default:
                System.out.println("服务端响应码" + rc + "未知");
                break;
        }
    }
}
```

4.3.3.9. 获取节点内容（同步）

```
@Test
public void getNode() throws InterruptedException, KeeperException {
    // 实例化对象，用于记录节点的状态信息
    Stat stat = new Stat();
    // 获取数据
    byte[] data = zkCli.getData("/ApiNode1", true, stat);
    // 打印数据
    System.out.println(new String(data));
    // 打印节点状态信息
    System.out.println(stat);
    System.out.println(stat.getVersion());
    System.out.println(stat.getCtime());
}
```

4.3.3.10. 获取节点内容（异步）

```
@Test
public void getNodeAsync() throws InterruptedException {
    zkCli.getData("/ApiNode1", true, new GetNodeCallBack(), "");
    Thread.sleep(Integer.MAX_VALUE);
}

private static class GetNodeCallBack implements AsyncCallback.DataCallback {
    /**
     * 获取数据的回调
     * @param rc 服务端响应代码：0 => 成功，-4 => 客户端与服务端连接断开，-110 => 指定节点已存在，-112 => 会话已过期
     * @param path 获取数据的节点路径
     * @param ctx 获取数据时传递的ctx
     * @param data 获取到的节点数据
     * @param stat 获取到的节点状态
     */
    @Override
    public void processResult(int rc, String path, Object ctx, byte[] data, Stat stat) {
        switch (rc) {
            case 0:
                System.out.println("节点数据获取成功：" + new String(data));
                break;
            case -4:
                System.out.println("客户端与服务端连接已断开");
                break;
```

```
            case -112:
                System.out.println("会话已过期");
                break;
            default:
                System.out.println("服务端响应码" + rc + "未知");
                break;
        }
    }
}
```

4.3.3.11. 获取所有子节点（同步）

```
@Test
public void getChildren() throws InterruptedException, KeeperException {
    List<String> children = zkCli.getChildren("/", true);
    System.out.println(children);
}
```

4.3.3.12. 获取所有子节点（异步）

```
@Test
public void getChildrenAsync() throws InterruptedException {
    zkCli.getChildren("/", true, new GetChildrenCallBack(), "");
    Thread.sleep(Integer.MAX_VALUE);
}
```

```
6
7  private static class GetChildrenCallBack implements
   AsyncCallback.ChildrenCallback {
8
9      /**
10      * 获取到所有的子节点的回调
11      * @param rc 服务端响应代码：0 => 成功，-4 => 客户端与服
   务端连接断开，-110 => 指定节点已存在，-112 => 会话已过期
12      * @param path 获取子节点的节点路径
13      * @param ctx 调用方法传递的ctx
14      * @param children 所有的子节点
15      */
16     @Override
17     public void processResult(int rc, String path,
   Object ctx, List<String> children) {
18         switch (rc) {
19             case 0:
20                 System.out.println("子节点获取成功： " +
   children);
21                 break;
22             case -4:
23                 System.out.println("客户端与服务端连接已断
   开");
24                 break;
25             case -112:
26                 System.out.println("会话已过期");
27                 break;
28             default:
29                 System.out.println("服务端响应码" + rc +
   "未知");
30                 break;
31         }
32     }
```

```
33  }
```

4.4. ZooKeeper的工作原理【重要】

4.4.1. 选举制度

4.4.1.1. 选举制度说明

ZooKeeper是一个分布式协调服务，但本身也是一个集群，并且是一个主从架构的集群。与HDFS类似的是，这个集群中也是需要一个主节点和若干个从节点的，但是与HDFS不同的是，ZooKeeper集群的主节点并不是我们通过配置文件设置的，而是所有的节点启动之后自己选举出来的！

> ZooKeeper集群中节点的角色状态: Leader 和 Follower

4.4.1.2. 选举中的几个概念

- Leader

  ```
  1  Zookeeper 集群工作的核心。
  2
  3  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；集群内部各个服务器的调度者。
  4  对于 create， setData， delete 等有写操作的请求，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。
```

- 投票相关
  - myid

    ```
    我们在搭建ZooKeeper集群的时候设置过的服务器的ID值，这个值在选举的投票过程中有一定的权重占比。
    ```

  - zxid

    ```
    事务ID，ZooKeeper会为每一个更新的操作分配一个事务ID。事务ID是一个64位的数字，且全局单调递增。
    在一个节点的状态信息中会看到这个值。
    ```

  - Epoch

    ```
    逻辑时钟。
    ```

    - 在进行选举的时候，epoch > zxid > myid

- 状态相关
  - LOOKING: 竞选状态
  - FOLLOWING: 随从状态，同步leader状态，参与选票
  - OBSERVING: 观察状态，同步leader状态，不参与选票
  - LEADING: 领导状态

4.4.1.3. 选举发生的时机

- 启动ZooKeeper集群的时候
- 集群运行过程中，Leader失联

4.4.1.4. 启动集群选举

我们以3个节点的ZooKeeper集群为例，启动的顺序是qianfeng01, qianfeng02, qianfeng03

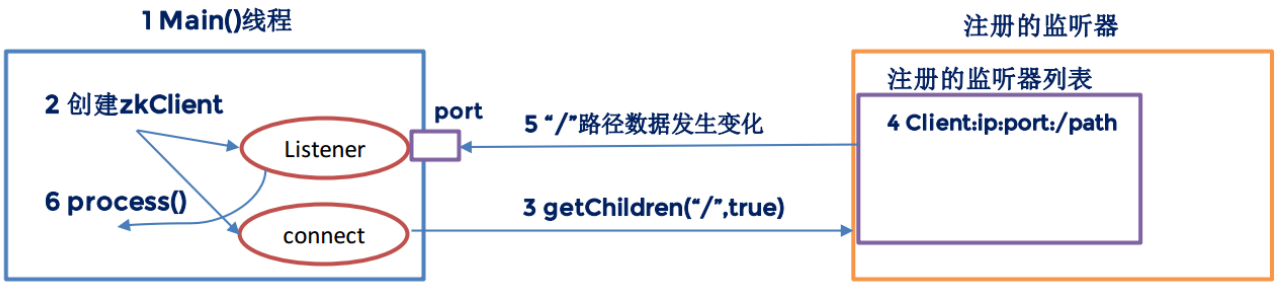
1. qianfeng01启动，投自己一票。由于未满足过半的条件，进入到LOOKING状态。
2. qianfeng02启动，投自己一票。现在活跃的节点已经过半，归票:
   - 优先比较Epoch，谁的大，谁当选Leader
   - 如果Epoch相同，比较zxid，谁的大，谁当选Leader
   - 如果zxid也相同，比较myid，谁的大，谁当选Leader
3. qianfeng03启动，由于集群中已经出现了Leader，因此qianfeng03直接进入FOLLOWING状态，成为Follower。

4.4.1.5. 集群运行中

集群在运行的过程中，Leader失联了。例如: 集群启动之后，qianfeng02当选为Leader，先在已经失联...

- qianfeng01、qianfeng03会进入到LOOKING状态，开始投票，并每人都给自己投了一票。
- 归票:
  - 优先比较Epoch，谁的大，谁当选Leader
  - 如果Epoch相同，比较zxid，谁的大，谁当选Leader
  - 如果zxid也相同，比较myid，谁的大，谁当选Leader

4.4.2. 监听原理



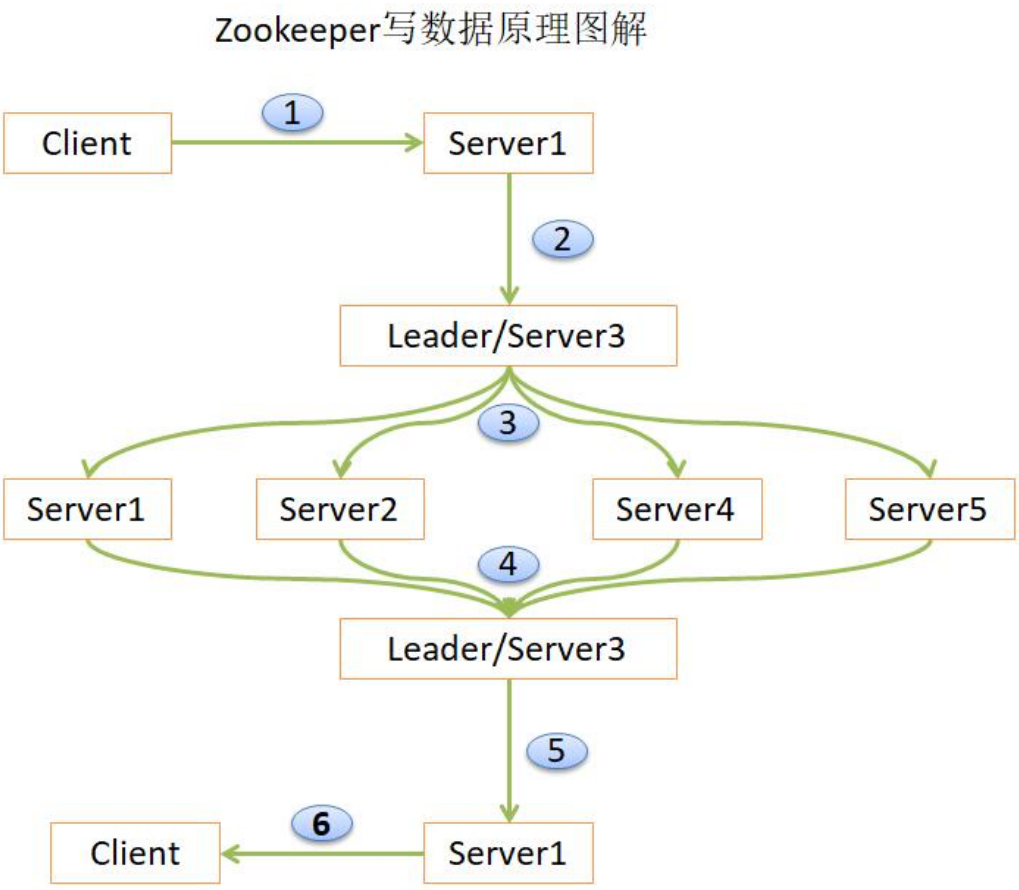
4.4.2.1. 图解

```
1. 首先要有一个main()线程
2. 在main线程中创建Zookeeper客户端， 这时就会创建两个线程， 一个负责网络连接通信(connet),一个负责监听(listener)。
3. 通过connect线程将注册的监听事件发送给Zookeeper。
4. 在Zookeeper的注册监听器列表中将注册的监听事件添加到列表中。
5. Zookeeper监听到有数据或路径变化， 就会将这个消息发送给listener线程。
6. listener线程内部调用了process（） 方法。
```

4.4.2.2. 用途

```
1.  监听节点数据的变化:  get /path  watch
2.  监听子节点增减的变化 : ls /path  watch
```

4.4.3. 写数据流程



1. Client向Zookeeper的server1上写数据，发送一个写请求
2. 如果server1不是leader,那么server1会把请求进一步转发给leader。
3. 这个leader会将写请求广播给所有server。
4. 各个Server写成功后就会通知leader。
5. 当leader收到半数以上的server写成功的通知，就说明数据写成功了。写成功后，leader会告诉server1数据写成功了。
6. server1会进一步通知Client数据写成功了。这时就认为整个写操作成功。

4.4.4. setAcl【了解】

设置节点Acl。

此处重点说一下acl，acl由大部分组成：1为scheme，2为user，3为permission，一般情况下表示为scheme:id:permissions。

其中scheme和id是相关的，下面将scheme和id一起说明。

scheme和id

world：它下面只有一个id，叫anyone，world:anyone代表任何人，zookeeper中对所有人有权限的结点就是属于world:anyone的

auth：它不需要id，只要是通过authentication的user都有权限（zookeeper支持通过kerberos来进行authencation，也支持username/password形式的authentication）

digest：它对应的id为username:BASE64(SHA1(password))，它需要先通过username:password形式的authentication

ip：它对应的id为客户机的IP地址，设置的时候可以设置一个ip段，比如ip:192.168.1.0/16，表示匹配前16个bit的IP段

super：在这种scheme情况下，对应的id拥有超级权限，可以做任何事情(cdrwa)

permissions

CREATE(c)：创建权限，可以在在当前node下创建child node

DELETE(d)：删除权限，可以删除当前的node

READ(r)：读权限，可以获取当前node的数据，可以list当前node所有的child nodes

WRITE(w)：写权限，可以向当前node写数据

ADMIN(a)：管理权限，可以设置当前node的permission

综上，一个简单使用setAcl命令，则可以为：

setAcl /zookeeper/node1 world:anyone:cdrw

详解：
1 Zookeeper ACL

ZooKeeper的权限管理亦即ACL控制功能通过Server、Client两端协调完成：
Server端：
一个ZooKeeper的节点（znode）存储两部分内容：数据和状态，状态中包含ACL信息。创建一个znode会产生一个ACL列表，列表中每个ACL包括：

l 验证模式(scheme)
l 具体内容(Id)（当scheme=“digest”时，Id为用户名密码，例如“root：J0sTy9BCUKubtK1y8pkbL7qoxSw=”）
l 权限(perms)
1.1 scheme

ZooKeeper提供了如下几种验证模式（scheme）：
l digest：Client端由用户名和密码验证，譬如user:password，digest的密码生成方式是Sha1摘要的base64形式
l auth：不使用任何id，代表任何已确认用户。
l ip：Client端由IP地址验证，譬如172.2.0.0/24
l world：固定用户为anyone，为所有Client端开放权限
l super：在这种scheme情况下，对应的id拥有超级权限，可以做任何事情(cdrwa)
注意的是，exists操作和getAcl操作并不受ACL许可控制，因此任何客户端可以查询节点的状态和节点的ACL。
节点的权限（perms）主要有以下几种：
l Create 允许对子节点Create操作
l Read 允许对本节点GetChildren和GetData操作
l Write 允许对本节点SetData操作
l Delete 允许对子节点Delete操作
l Admin 允许对本节点setAcl操作
Znode ACL权限用一个int型数字perms表示，perms的5个二进制位分别表示setacl、delete、create、write、read。比如0x1f=adcwr，0x1=----r，0x15=a-c-r。
1.1.1 world scheme固定id为anyone，表示对所有Client端开放权限：

```
[zk: localhost:2181(CONNECTED) 13] create /123 "123"
Created /123
[zk: localhost:2181(CONNECTED) 14] getAcl /123
'world,'anyone
: cdrwa
```

1.1.2 ip scheme设置可以访问的ip地址（比如127.0.0.1）或ip地址段（比如192.168.1.0/16）

10.194.157.58这台机器上创建/test并设置ip访问权限

```
[zk: 10.194.157.58:2181(CONNECTED) 0] create /test "123"
Created /test
[zk: 10.194.157.58:2181(CONNECTED) 1] setAcl /test ip:10.194.157.58:crwda
cZxid = 0x740021e467
ctime = Wed Dec 02 18:09:09 CST 2015
mZxid = 0x740021e467
mtime = Wed Dec 02 18:09:09 CST 2015
pZxid = 0x740021e467
cversion = 0
dataVersion = 0
aclVersion = 1
ephemeralOwner = 0x0
dataLength = 5
numChildren = 0
[zk: 10.194.157.58:2181(CONNECTED) 2] ls /test
[]
```

可以看到，本机是可以访问的。

在10.205.148.152上登陆

```
[zk: 10.194.157.58:2181(CONNECTED) 1] ls /test
Authentication is not valid : /test
```

可以看到，连接的ip不在授权中，提示访问错误。

1.1.3 digest scheme的id表示为username:BASE64(SHA1(password))

```
[root@rocket zookeeper-server1]# cd /usr/local/zookeeper-server1/
```

```
  [root@rocket zookeeper-server1]# pwd
  /usr/local/zookeeper-server1
  # 生成密文
  [root@rocket zookeeper-server1]# java -cp
./zookeeper-3.4.6.jar:./lib/log4j-
1.2.16.jar:./lib/slf4j-log4j12-1.6.1.jar:./lib/slf4j-
api-1.6.1.jar
org.apache.zookeeper.server.auth.DigestAuthenticationP
rovider test:test
  test:test->test:V28q/NynI4JI3Rk54h0r8O5kMug=
  创建acl
  clip_image002
  通过认证后，可以访问数据：
  [zk: localhost:2181(CONNECTED) 0]
  [zk: localhost:2181(CONNECTED) 0] ls /test_acl
  Authentication is not valid : /test_acl
  [zk: localhost:2181(CONNECTED) 1] getAcl /test_acl
  'digest,'test:V28q/NynI4JI3Rk54h0r8O5kMug=
  : cdrwa
  [zk: localhost:2181(CONNECTED) 2] addauth digest
test:test
  [zk: localhost:2181(CONNECTED) 3] ls /test_acl
  []
  [zk: localhost:2181(CONNECTED) 4] get /test_acl
  "test"
  cZxid = 0x33
  ctime = Wed Dec 02 00:10:47 PST 2015
  mZxid = 0x33
  mtime = Wed Dec 02 00:10:47 PST 2015
  pZxid = 0x33
  cversion = 0
  dataVersion = 0
  aclVersion = 1
```

ephemeralOwner = 0x0
dataLength = 6
numChildren = 0
1.2 SuperDigest超级管理员

当设置了znode权限，但是密码忘记了怎么办？还好Zookeeper提供了超级管理员机制。
一次Client对znode进行操作的验证ACL的方式为：
a）遍历znode的所有ACL：
i．对于每一个ACL，首先操作类型与权限（perms）匹配
ii．只有匹配权限成功才进行session的auth信息与ACL的用户名、密码匹配
b）如果两次匹配都成功，则允许操作；否则，返回权限不够error（rc=-102）
备注：如果znode ACL List中任何一个ACL都没有setAcl权限，那么就算superDigest也修改不了它的权限；再假如这个znode还不开放delete权限，那么它的所有子节点都将不会被删除。唯一的办法是通过手动删除snapshot和log的方法，将ZK回滚到一个以前的状态，然后重启，当然这会影响到该znode以外其它节点的正常应用。

superDigest设置的步骤
修改zkServer.sh，加入super权限设置
-Dzookeeper.DigestAuthenticationProvider.superDigest=super:gG7s8t3oDEtIqF6DM9LlI/R+9Ss=
clip_image004
clip_image006
重新启动Zookeeper
./zkServer.sh restart
这时候

不使用test:test进行认证，而是使用super:super进行认证：
[zk: localhost:2181(CONNECTED) 0] ls /test_acl

```
147   Authentication is not valid : /test_acl
148   [zk: localhost:2181(CONNECTED) 1] addauth digest
    super:super
149   [zk: localhost:2181(CONNECTED) 2] ls /test_acl
150   []
151   [zk: localhost:2181(CONNECTED) 3] get /test_acl
152   "test"
153   cZxid = 0x33
154   ctime = Wed Dec 02 00:10:47 PST 2015
155   mZxid = 0x33
156   mtime = Wed Dec 02 00:10:47 PST 2015
157   pZxid = 0x33
158   cversion = 0
159   dataVersion = 0
160   aclVersion = 1
161   ephemeralOwner = 0x0
162   dataLength = 6
163   numChildren = 0
164   1.3 ACL机制的缺陷
165
166   然而，ACL毕竟仅仅是访问控制，并非完善的权限管理，通过这种方式
    做多集群隔离，还有很多局限性：
167   ACL并无递归机制，任何一个znode创建后，都需要单独设置ACL，无
    法继承父节点的ACL设置。
168   除了ip这种scheme，digest和auth的使用对用户都不是透明的，这
    也给使用带来了很大的成本，很多依赖zookeeper的开源框架也没有加入
    对ACL的支持，例如hbase，storm。
```

4.5. 分布式锁

4.6. HA高可用集群

4.6.1. HA介绍

4.6.1.1. 什么是HA

HA: High Availability，高可用集群，指的是集群7*24小时不间断服务。

4.6.1.2. 为什么需要HA

在HDFS中，有NameNode、DataNode和SecondaryNameNode角色的分布，客户端所有的操作都是要与NameNode交互的，同时整个集群的命名空间信息也都保存在NameNode节点。但是，现在的集群配置中只有一个NameNode，于是就有一个问题: **单点故障**

那么，什么是单点故障呢？现在集群中只有一个NameNode，那么假如这个NameNode意外宕机、升级硬件等，导致NameNode不可用了，整个集群是不是也就不可用了？这就是单点故障的问题。

为了解决这样的问题，就需要高可用集群了。

4.6.1.3. 高可用的备份方式

- 主从模式（冷备）

```
1 准备两台服务器，准备相同的程序。 一台服务器对外提供服务，称为主节点(Active节点)；另外一台服务器平时不对外提供服务，主要负责和Active节点之间进行数据的同步，称为备份节点(Standby节点)。当主节点出现故障，Standby节点可以自动提升为Active节点，对外提供服务。
2 ZooKeeper实现的集群高可用，采用的就是这种模式。
```

```
1 　我作为一个班级的讲师，在班级负责授课的工作。如果我有一天生病请假了，是不是咱们班级就得自习一天了？为了解决这样的问题，教学部安排另外一个讲师，每天跟着我。我上课讲课，他就在旁边听着；我上课提问问题，他就在旁边看着；我去吃饭，他也在旁边跟着；我去上个厕所，他也在跟着！做为我的一个影子存在着。由于这个同时每天都会跟着我，因此我的一言一行，讲了什么内容，留了什么作业，吃了什么饭，抽了几根烟，他都知道！那么，如果有一天我生病请假了，他是不是就可以直接替代我为班级上课呢?
```

- 双主互备（热备）（了解）

```
1 准备两台服务器，准备相同的程序。同时对外提供服务(此时，这两台服务器彼此为对方的备份)，这样，当一台节点宕机的时候，另外一台节点还可以继续提供服务。
```

```
1 　小明到肯德基吃饭，找服务员点餐，这是正常的流程。但是，如果服务员只有一个，并且恰好生病了，那么小明是不是将没有办法正常点餐了。为了解决这个问题，肯德基雇了两个服务员，同时提供服务，这样一个服务员出问题了，另外一个服务员依然可以提供服务。
```

- 集群多备（了解）

```
基本上等同于双主互备，区别就在于同时对外提供服务的节点数量更多，备份数量更多
```

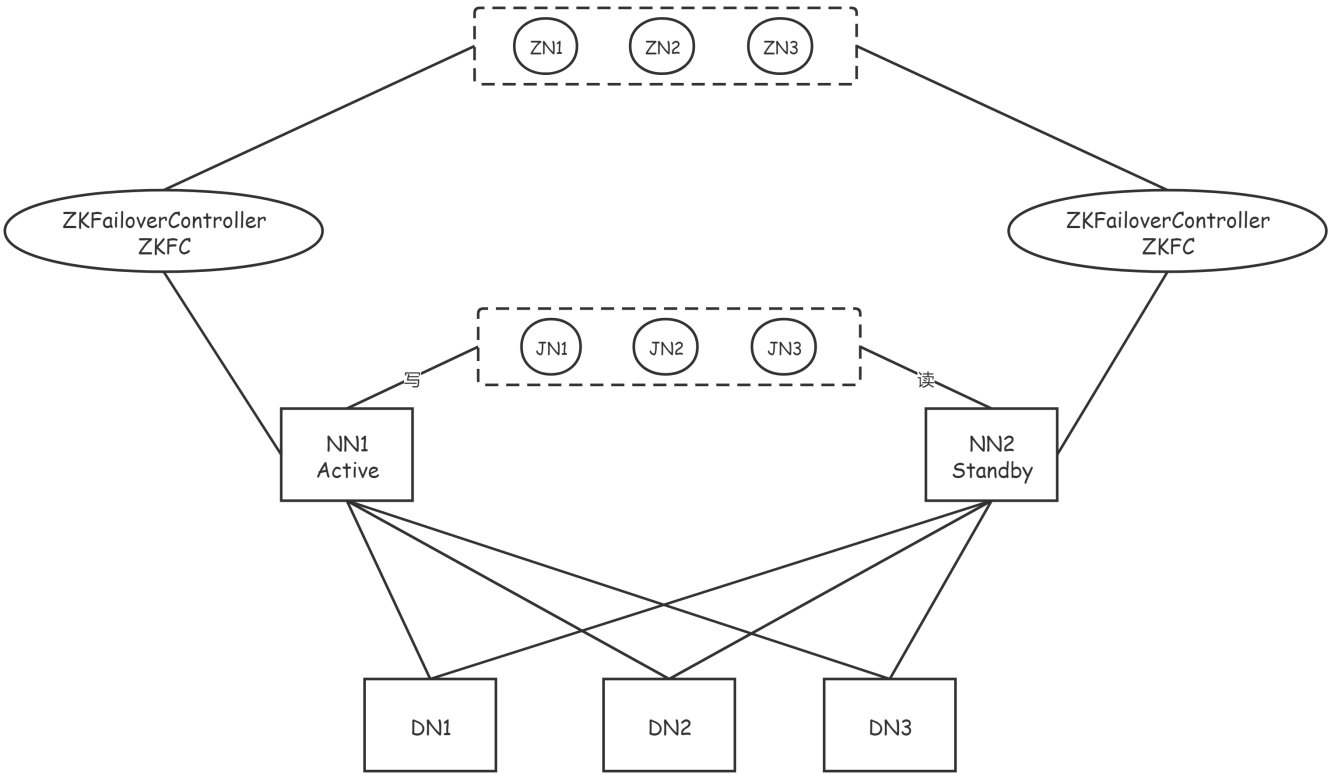
```
    肯德基觉得两个服务员也不保险，有两个同时生病的可能性，于是又多雇了几个服务员。
```

4.6.1.4. 高可用的实现

我们在这里采用的是主从模式的备份方式，也就是准备两个NameNode，一个对外提供服务，称为Active节点；另外一个不对外提供服务，只是实时的同步Active节点的数据，称为Standby的节点。

为了提供快速的故障转移，Standby节点还必须具有集群中块位置的最新信息。为了实现这一点，DataNodes被配置了两个NameNodes的位置，并向两者发送块位置信息和心跳信号。也就是说，DataNode同时向两个NameNode心跳反馈。

4.6.2. 高可用架构图



4.6.3. JournalNode

- **JournalNode的功能**

```
Hadoop2.x版本之后，Clouera提出了QJM/QuromJournal Manager，这是一个基于Paxos算法实现的HA的实现方案

1．基本的原理就是使用2N+1台JN存储EditLog，每次写入数据的时候，有半数以上的JN返回成功的信息，就表示本次的操作已经同步到了JN

2．在HA中，SecondaryNameNode这个角色已经不存在了，保证Standby节点的元数据信息与Active节点的元数据信息一致，需要通过若干个JN

3．当有任何的操作发生在Active节点上的时候，JN会记录这些操作到半数以上的节点中．Standby节点检测JN中的log日志文件发生了变化，会读取JN中的数据到自己的内存中，维护最新的目录树结构与元数据信息

4．当发生故障的时候，Active节点挂掉，此时Standby节点在成为新的Active节点之前，会将读取到的EditLog文件在自己的内存中进行推演，得到最新的目录树结构．此时再升为Active节点，可以无缝的继续对外提供服务．
```

- **防止脑裂的发生**

```
    对于HA群集的正确操作至关重要，一次只能有一个NameNode处于Active状态。否则，名称空间状态将在两者之间迅速分散，从而有数据丢失或其他不正确结果的风险。为了确保该属性并防止所谓的“裂脑情况”，JournalNode将一次仅允许单个NameNode成为作者。在故障转移期间，变为活动状态的NameNode将仅承担写入JournalNodes的角色，这将有效地防止另一个NameNode继续处于活动状态，从而使新的Active可以安全地进行故障转移。

- 怎么理解脑裂?
    就是Active节点处于网络震荡状态，假死状态，Standby就转为Active。等网络震荡过后，就有两个Active了，这就是脑裂。
```

- **JournalNode集群正常工作的条件**

```
- 至少3个Journalnode节点
- 运行个数建议奇数个(3,5,7等)
- 满足（n+1）/2个以上，才能正常服务。即能容忍（n-1）/2个故障。
```

- **JournalNode的缺点**

```
    在这种模式下，即使活动节点发生故障，系统也不会自动触发从活动NameNode到备用NameNode的故障转移，必须需要人为的操作才行。要是有一个能监视Active节点的服务功能就好了。
    这个时候，我们就可以使用zookeeper集群服务，来帮助我们进行自动容灾了。
```

4.6.4. 自动容灾原理

如果想进行HA的自动故障转移，那么需要为HDFS部署两个新组件：ZooKeeper quorum和ZKFailoverController进程（缩写为ZKFC）

4.6.4.1. Zookeeper quorum

Apache ZooKeeper是一项高可用性服务，用于维护少量的协调数据，将数据中的更改通知客户端并监视客户端的故障。HDFS自动故障转移的实现依赖ZooKeeper进行以下操作：

```
- 故障检测
  集群中的每个NameNode计算机都在ZooKeeper中维护一个持久性会话。如果计算机崩溃，则ZooKeeper会话将终止，通知另一个NameNode应触发故障转移。

- 活动的NameNode选举(HA的第一次启动)
  ZooKeeper提供了一种简单的机制来专门选举一个节点为活动的节点。如果当前活动的NameNode崩溃，则另一个节点可能会在ZooKeeper中采取特殊的排他锁，指示它应成为下一个活动的NameNode。
```

4.6.4.2. ZKFC

ZKFailoverController（ZKFC）是一个新组件，它是一个ZooKeeper客户端，它监视和管理namenode的状态。运行namenode的每台机器都会运行一个ZKFC，该ZKFC负责以下内容：

- 运行状况监视
  ZKFC使用运行状况检查命令定期ping其本地NameNode。只要NameNode以健康状态及时响应，ZKFC就会认为该节点是健康的。如果节点崩溃，冻结或以其他方式进入不正常状态，则运行状况监视器将其标记为不正常。

- ZooKeeper会话管理
  当本地NameNode运行状况良好时，ZKFC会在ZooKeeper中保持打开的会话。如果本地NameNode处于活动状态，则它还将持有一个特殊的“锁定”znode。该锁使用ZooKeeper对“临时”节点的支持。如果会话到期，则锁定节点将被自动删除。

- 基于ZooKeeper的选举
  如果本地NameNode运行状况良好，并且ZKFC看到当前没有其他节点持有锁znode，则它本身将尝试获取该锁。如果成功，则它“赢得了选举”，并负责运行故障转移以使其本地NameNode处于活动状态。故障转移过程类似于上述的手动故障转移：首先，如有必要，将先前的活动节点隔离，然后将本地NameNode转换为活动状态。

4.6.4.3. 自动容灾的过程描述

ZKFC（是一个进程，和NN在同一个物理节点上）有两只手，分别拽着NN和Zookeeper。(监控NameNode健康状态，并向Zookeeper注册NameNode)；集群一启动，2个NN谁是Active？谁又是Standby呢？

2个ZKFC先判断自己的NN是否健康，如果健康，2个ZKFC会向zoopkeeper集群抢着创建一个节点，结果就是只有1个会最终创建成功，从而决定active地位和standby位置。如果ZKFC1抢到了节点，ZKFC2没有抢到，ZKFC2也会监控watch这个节点。如果ZKFC1的Active NN异常退出，ZKFC1最先知道，就访问ZK，ZK就会把曾经创建的节点删掉。删除节点就是一个事件，谁监控这个节点，就会调用callback回调，ZKFC2就会把自己的地位上升到active，但在此之前要先确认ZKFC1的节点是否真的挂掉？这就引入了第三只手的概念。

ZKFC2通过ssh远程连接NN1尝试对方降级，判断对方是否挂了。确认真的不健康，才会真的 上升地位之active。所以ZKFC2的步骤是：

1.创建新节点。

2.第三只手把对方降级。

3.把自己升级

那如果NN都没毛病，ZKFC挂掉了呢？Zoopkeeper有一个客户端session机制，集群启动之后，2个ZKFC除了监控自己的NN，还要和Zoopkeeper建立一个tcp长连接，并各自获取自己的session。只要一方的session失效，Zoopkeeper 就会删除该方创建的节点，同时另一方创建节点，上升地位。

4.6.5. HA的配置

4.6.5.1. 守护进程布局

```
qianfeng01：NameNode、DataNode、JournalNode、QuorumPeerMain、ZKFC
qianfeng02：NameNode、DataNode、JournalNode、QuorumPeerMain、ZKFC
qianfeng03：DataNode、JournalNode、QuorumPeerMain
```

现在，先停止HDFS的进程，修改如下的配置文件吧！

4.6.5.2. hdfs-site.xml

```
<!-- 注意：高可用的集群，没有SecondaryNameNode的存在，因此在这个文件中之前存在的SecondaryNameNode的配置需要删除 -->
<configuration>
    <!-- 配置NameNode的逻辑名称 -->
    <!-- 注意：后面的很多参数配置都是需要使用到这个名称的 -->
    <property>
        <name>dfs.nameservices</name>
        <value>supercluster</value>
    </property>

    <!-- 配置两个NameNode的唯一标识符 -->
    <property>
        <name>dfs.ha.namenodes.supercluster</name>
        <value>nn1,nn2</value>
    </property>

    <!-- 针对每一个NameNode，配置自己的RPC通信地址和端口 -->
    <property>
        <name>dfs.namenode.rpc-address.supercluster.nn1</name>
```

```
        <value>qianfeng01:9820</value>
    </property>
    <property>
        <name>dfs.namenode.rpc-address.supercluster.nn2</name>
        <value>qianfeng02:9820</value>
    </property>

    <!-- 针对每一个NameNode，配置WebUI的地址和端口 -->
    <property>
        <name>dfs.namenode.http-address.supercluster.nn1</name>
        <value>qianfeng01:9870</value>
    </property>
    <property>
        <name>dfs.namenode.http-address.supercluster.nn2</name>
        <value>qianfeng02:9870</value>
    </property>

    <!-- 定义journalnode进程的数据存储的父路径，目录在上面已经定义好了的：journalData -->
    <property>
        <name>dfs.journalnode.edits.dir</name>
        <value>/usr/local/hadoop-3.3.1/tmp</value>
    </property>

    <!-- 配置journalnode的服务器地址和存储目录（数目为奇数个）-->
    <!-- 服务器地址使用分号“;”作为分隔符-->
    <property>
        <name>dfs.namenode.shared.edits.dir</name>
```

```
46
 <value>qjournal://qianfeng01:8485;qianfeng02:8485;qianfeng03:8485/journalData</value>
47      </property>
48
49      <!-- 指定客户端连接Active的namenode节点的java类型 -->
50      <property>
51
 <name>dfs.client.failover.proxy.provider.supercluster</name>
52
 <value>org.apache.hadoop.hdfs.server.namenode.ha.ConfiguredFailoverProxyProvider</value>
53      </property>
54
55      <!--为了保证系统的正确性，在任何时间只有一个NameNode处于Active状态，需要配置一个防护机制 -->
56      <property>
57          <name>dfs.ha.fencing.methods</name>
58          <value>sshfence</value>
59      </property>
60      <!--为了使该防护选项起作用，它必须能够在不提供密码的情况下SSH到目标节点。因此，还必须配置以下属性-->
61      <property>
62          <name>dfs.ha.fencing.ssh.private-key-files</name>
63          <value>/root/.ssh/id_rsa</value>
64      </property>
65      <!-- 免密登陆超时时间，超过此时间未连接上，则登陆失败，此配置可选-->
66      <property>
67          <name>dfs.ha.fencing.ssh.connect-timeout</name>
68          <value>30000</value>
```

```
    </property>

    <!-- 支持自动容灾属性 -->
    <property>
        <name>dfs.ha.automatic-failover.enabled</name>
        <value>true</value>
    </property>

    <!-- 块的副本数量 -->
    <property>
        <name>dfs.replication</name>
        <value>3</value>
    </property>
</configuration>
```

4.6.5.3. core-site.xml

```
<configuration>
    <!--注意：使用到的是在hdfs-site.xml中配置的逻辑名称 -->
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://supercluster</value>
    </property>

    <!-- hdfs的数据保存的路径，被其他属性所依赖的一个基础路径 -->
    <property>
        <name>hadoop.tmp.dir</name>
        <value>/usr/local/hadoop-3.3.1/tmp</value>
    </property>

    <!-- ZooKeeper服务的地址 -->
```

```
    <property>
        <name>ha.zookeeper.quorum</name>

 <value>qianfeng01:2181,qianfeng02:2181,qianfeng03:2181
</value>
    </property>

</configuration>
```

4.6.5.4. hadoop-env.sh

```
# 添加两行
export HDFS_JOURNALNODE_USER=root
export HDFS_ZKFC_USER=root
```

4.6.5.4. 分发配置文件到其他节点

```
[root@qianfeng01 ~]# cd $HADOOP_HOME/etc
[root@qianfeng01 hadoop]# scp -r hadoop qianfeng02:$PWD
[root@qianfeng01 hadoop]# scp -r hadoop qianfeng03:$PWD
```

4.6.5.5. 启动集群

现在，集群已经搭建成为了高可用的集群了。在启动集群之前，我们需要先明确一件事情: 集群现在的状态有两种:

- 这个集群我之前使用过，NameNode已经存储有数据了(fsimage和

edits已生成)

- 这个集群是我新搭建的，我直接搭建集群的时候就搭建的高可用的集群，之前从来没有启动过

如果你是第一种情况，请跳转到 **4.6.5.6. 启动: 普通集群转HA**

如果你是第二种情况，请跳转到 **4.6.5.7. 启动: 直接搭建HA**

4.6.5.6. 启动: 普通集群转HA

```
# 1. 启动集群的JournalNode服务。
#    注意事项：如果之前集群还在运行，需要先将其停止！使用命令 stop-dfs.sh
[root@qianfeng01 ~]# hdfs --daemon start journalnode
[root@qianfeng02 ~]# hdfs --daemon start journalnode
[root@qianfeng03 ~]# hdfs --daemon start journalnode

# 2. 启动以前节点上的namenode进程
[root@qianfeng01 ~]# hdfs --daemon start namenode

# 3. 在新的namenode节点上拉取镜像文件
[root@qianfeng02 ~]# hdfs namenode -bootstrapStandby

# 4. 同步日志到journalnode集群上,再启动集群
#    先关namenode
[root@qianfeng01 ~]# hdfs --daemon stop namenode
# 再同步日志
[root@qianfeng01 ~]# hdfs namenode -initializeSharedEdits

```

```
# 5. 格式化zkfc
# 5.1. 前提QuorumPeerMain服务必须处于开启状态，客户端zkfc才能格式化成功
[root@qianfeng01 ~]# zkServer.sh start
[root@qianfeng02 ~]# zkServer.sh start
[root@qianfeng03 ~]# zkServer.sh start
# 5.2. 选择其中一个namenode节点进行格式化zkfc
[root@qianfeng01 ~]# hdfs zkfc -formatZK

# 6. 你就可以快乐的开启HA集群进行测试了
[root@qianfeng01 ~]# start-all.sh

# 查看NameNode的状态
[root@qianfeng01 ~]# hdfs haadmin -getServiceState nn1

# 注意：以后开HA集群时，要先开zookeeper服务，再开HDFS。
```

4.6.5.7. 启动: 直接搭建HA

```
# 1. 启动三个节点上的journalnode服务
[root@qianfeng01 ~]# hdfs --daemon start journalnode
[root@qianfeng02 ~]# hdfs --daemon start journalnode
[root@qianfeng03 ~]# hdfs --daemon start journalnode

# 2. 格式化namenode
#    - 先删除所有节点的${hadoop.tmp.dir}/tmp/的数据(可选，这一步表示弃用fsimage.)
#    - 选择其中一个namenode进行格式化
[root@qianfeng01 ~]# hdfs namenode -format
#    - 并启动namenode进程
[root@qianfeng01 ~]# hdfs --daemon start namenode

```

```
# 3. 在另一台namenode上拉取已格式化的那台机器的镜像文件（数据的一致性）
[root@qianfeng02 ~]# hdfs namenode -bootstrapStandby

# 4. 然后关闭已经启动的namenode
[root@qianfeng01 ~]# hdfs --daemon stop namenode

# 5. 格式化zkfc
# 5.1. 前提QuorumPeerMain服务必须处于开启状态，客户端zkfc才能格式化成功
[root@qianfeng01 ~]# zkServer.sh start
[root@qianfeng02 ~]# zkServer.sh start
[root@qianfeng03 ~]# zkServer.sh start
# 5.2. 选择其中一个namenode节点进行格式化zkfc
[root@qianfeng01 ~]# hdfs zkfc -formatZK

# 6. 你就可以快乐的开启HA集群进行测试了
[root@qianfeng01 ~]# start-all.sh

# 注意：以后开HA集群时，要先开zookeeper服务，再开HDFS。
```

4.6.5.8. 自动容灾测试

> 由于CentOS7的minimal版本缺少容灾切换ActiveNameNode节点时所需要的组件，因此需要手动安装一下:
>
> yum install -y psmisc

```
1. 首先查看当前活跃的Active节点是谁
2. Kill掉活跃节点上的NameNode进程，模拟宕机
3. 观察另外一个节点，是否已经变成Active的状态
```

4.6.5.9. API操作

```
1  对于HA集群来说，如果想要使用JavaAPI进行集群的访问，需要在Configuration中设置好每一个属性，也就是再上方配置集群时进行的各种配置。如果觉得麻烦，最简单的方法就是将core-site.xml、hdfs-site.xml拷贝到项目的Resources目录下。
```

五、案例实战

六、教学总结

```
1  Zookeeper是一个分布式协调服务，也是大数据生态圈中非常重要的一个组件，在很多地方都起到了非常重要的作用，例如在集群高可用(HA)、在HBase、在Kafka中，都扮演了非常重要的角色。在Zookeeper中，我们必须要掌握的内容是：
2
3  1. Zookeeper的数据模型
4  2. Zookeeper的节点类型
5  3. Zookeeper的Shell操作
6  4. Zookeeper的选举制度
7  5. Zookeeper的监听原理
8  6. Zookeeper的写数据流程
```

七、课后作业

如何在Zookeeper上创建一个临时节点

```
create -e /test "contents"
```

八、解决方案

8.1. Zookeeper为什么是奇数节点

```
剩下的节点数必须大于宕掉的个数，也就是存活节点需要大于总数一半，zookeeper才可以继续使用。假设总结点为5，则需要3台存活，即使增加为6也是3台，也就是说只有节点数为增加到奇数时最小存活节点才会增加，所以设置节点为奇数节约资源。
```

8.2. Zookeeper的选举制度

```
暂停对外服务，

1．各节点会先选自己作为leader，然后将选票携带事务id:zxid发送出去
2．各节点拿到选票后，先排除非本轮的票，然后比对自己的选票跟各个节点发来的选票，先比较zxid（越大说明数据越新），相同时比较myid，大的一方获胜，将票投给获胜方，然后各自发回节点。（假如自己是1节点，3节点发来的票，比对后要么返回1要么返回3）
3．投票后，各个节点会统计投票信息，判断如果有过半选票则认为选出了leader，更新自身状态为follow或leader，如果没有则一直重复2直到满足条件为止。
4．选举出leader后，新节点或原leader节点宕机恢复后，会直接变为follow状态，不再进行选举。
```