# 大数据基础

目录

[1.高并发 1](#_Toc30548)

[1.1.进程和线程 1](#_Toc17011)

[1.1.1.什么是进程？ 1](#_Toc10337)

[1.1.2.为什么要引入进程？ 1](#_Toc25318)

[1.1.3.进程的状态 2](#_Toc26584)

[1.1.4.什么是线程 2](#_Toc3886)

[1.1.5.进程或线程的调度算法 3](#_Toc15256)

[1.2.NIO  3](#_Toc16459)

[1.2.1.NIO概述 3](#_Toc208)

[1.2.2.为什么要学习NIO 3](#_Toc19989)

[1.2.3.BIO和NIO的区别 4](#_Toc2492)

[1.2.4.BIO和NIO的适用场景 4](#_Toc15507)

[1.2.5.BIO 4种产生阻塞的方法： 4](#_Toc16920)

[1.2.6.Buffer—ByteBuffer 6](#_Toc10528)

[1.2.7.Socket实现 10](#_Toc22581)

[1.2.8.BIO实现 12](#_Toc32155)

[1.2.9.Selector服务通道API 15](#_Toc6120)

[1.2.10.粘包问题 17](#_Toc566)

[1.2.11.FileChannel 18](#_Toc19870)

[1.2.12.Java NIO的总结 19](#_Toc27296)

[1.3.并发工具包 20](#_Toc31576)

[1.3.1.ArrayBlockingQueue和LinkedBlockingQueue 20](#_Toc8608)

[1.3.2.SynchronousQueue 22](#_Toc13931)

[1.3.3.并发 Map(映射) ConcurrentMap 22](#_Toc28245)

[1.3.4.红黑二叉树 23](#_Toc31780)

[1.3.5.闭锁CountDownLatch！！！ 25](#_Toc10430)

[1.3.6.栅栏 CyclicBarrier 25](#_Toc31951)

[1.3.7.交换机 Exchanger 26](#_Toc7987)

[1.3.8.线程池 27](#_Toc6061)

[1.3.9.Callable 30](#_Toc7416)

[1.3.10.ScheduledExecutorService - 定时执行器服务 30](#_Toc22493)

[1.3.11.ForkJoinPool - 分叉合并池 31](#_Toc16913)

[1.3.12.锁 Lock 32](#_Toc321)

[1.3.13.原子性 36](#_Toc7928)

[2.Zookeeper 38](#_Toc30189)

[2.1.Zookeeper概述 38](#_Toc23284)

[2.1.1.Zookeeper介绍 38](#_Toc16284)

[2.1.2.分布式概念（集群） 38](#_Toc8726)

[2.1.3.分布式编程容易出现的问题 39](#_Toc9793)

[2.2.Zookeeper单机模式安装 39](#_Toc16424)

[2.3.Zookeeper指令与数据结构 40](#_Toc17327)

[2.4.Zk API 42](#_Toc14356)

[2.4.1.项目搭建 42](#_Toc17896)

[2.4.2.API演示 43](#_Toc25488)

[2.5.ZK集群搭建和配置 47](#_Toc23712)

[2.6.选举机制 49](#_Toc17676)

[2.6.1.Zookeeper事务概念 49](#_Toc29667)

[2.6.2.Zookeeper选举机制 50](#_Toc17978)

[2.7.Zookeeper集群指令 52](#_Toc11537)

[2.7.1.Zookeeper集群命令 52](#_Toc12198)

[2.7.2.Zookeeper配置详解 53](#_Toc20615)

[2.8.Zookeeper观察者 55](#_Toc18998)

[2.8.1.观察者概述 55](#_Toc1445)

[2.8.2.怎么使用观察者 56](#_Toc24974)

[2.9.Zookeeper特性总结 57](#_Toc19490)

[2.10.zookeeper的应用场景  57](#_Toc22185)

[2.11.2PC算法 59](#_Toc18624)

[2.12.Paxos算法 62](#_Toc11302)

[2.13.Zookeeper插件 64](#_Toc18657)

[3.AVRO 66](#_Toc541)

[3.1.数据序列化 66](#_Toc15247)

[3.2.Java原生序列化和反序列化 67](#_Toc25976)

[3.3.AVRO介绍 69](#_Toc5105)

[3.3.1.AVRO介绍 69](#_Toc14107)

[3.3.2.Avro Schema 70](#_Toc26697)

[3.3.3.AVRO实现对象的序列化和反序列化 74](#_Toc20846)

[4.RPC框架 76](#_Toc31205)

[4.1.概念 76](#_Toc1788)

[4.2.模式定义avdl文件 78](#_Toc30132)

[4.2.1.实现步骤： 78](#_Toc11077)

[4.2.2.AVRO—实现RPC加法运算 79](#_Toc31320)

[4.2.3.AVRO—实现RPC对象的传输 80](#_Toc7932)

[5.zebra项目  81](#_Toc6122)

[5.1.zebra项目介绍与分析 81](#_Toc25407)

[5.2.zebra项目整体架构 82](#_Toc24275)

[5.3.zebra项目业务实现 84](#_Toc5008)

# 1.高并发

## 1.1.进程和线程

### 1.1.1.什么是进程？

进程=一个活动的程序，把程序加载到内存里执行。

1）从内存的分配角度，每个进程都独占一块内存地址空间。

2）从执行的逻辑角度，每个进程都可以被cpu所处理，也可以暂时挂起，让其他进程得以被处理，针对单核架构，同一时刻，只有一个进程能被cpu所处理，因为cpu处理速度快，以及相关的进程算法的存在，使得宏观上感觉进程是并行处理的，但本质上是串行处理的。针对多核架构，宏观和微观都是并行的

3）从时序角度，每个进程经过一段时间后，进度都是向前推进的。

计算机生成了可选文字:
物 理 视 角 
（ 进 程 切 换 ） 
进 程 A 
进 程 B 
进 程 C 
逻 辑 视 角 
（ 多 道 并 发 ） 
： 程 ： 
时 序 视 角 
（ 持 续 推 进 ） 
进 程 A 
进 程 B 
进 程 C 
时 间 

### 1.1.2.为什么要引入进程？

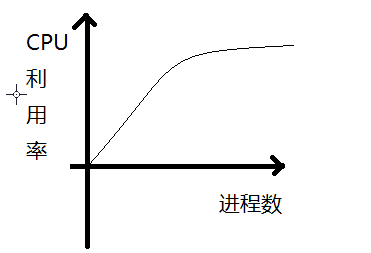
最初计算机是单道编程，比如一个程序，假设20%的时间在用cpu，80%的时间在做磁盘I/O。

所以引入进程的目的是满足多道编程，比如有两个进程，此时的利用率：1-0.8\*0.8=0.36

比如三个进程呢？cpu的利用率：1-0.8\*0.8\*0.8=0.48

多道编程的特点是：随着进程增多，cpu利用率越来越高

此外，也可以减少响应时间。



|  |  |
| --- | --- |
| 产生进程的情况 | 进程消亡的情况 |
| 1.系统初始化时，会产生一些系统进程，比如会话进程  壳进程，登录进程，安全管理进程等  2.父进程创建子进程  3.用户请求产生进程 | 1.一个进程执行完毕而自行退出消亡 2.进程在执行过程中出错或由于异常引起的退出消亡 3.被其他进程所杀死 |

### 1.1.3.进程的状态

进程的状态总体上分两种，执行态和挂起态

对于挂起态，我们要明确是什么原因而挂起的。

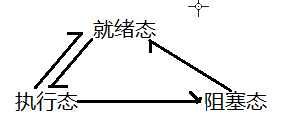
1）一个进程由于执行时间过长而主动挂起，让cpu处理其他的进程

2）一个进程在执行过程中由于发生某些I/O阻塞操作，而挂起

3）一个进程主动执行了一些阻塞操作，比如Sleep，而挂起

针对第一种情况的进程，给他cpu可以继续运行，而对于第二种和第三种的进程，即使给cpu，也处理不了，我们称之为阻塞态进程。

 进程的状态转变



### 1.1.4.什么是线程

一个进程至少有一个线程，也可以有多个线程。为什么要引入线程？

比如：拿word写笔记，这个word进程会监听打字；此外，每隔10分钟（默认值）自动保存一次。

总结：引入线程模型，目的就是为了让进程可以同时做多件事（线程是进程的分身）。此外，引入线程模型后，cpu的最小执行单位就编程了线程。

最后总结：进程相当于资源组织单位，线程时cpu最小执行单位。

### 1.1.5.进程或线程的调度算法

1）FCFS first come first server 先来服务调度算法

2）时间片轮转调度算法，局限性是可能使得某个短任务进程的处理时间延长

3）短任务优先算法，局限性是使得长任务进程长时间得不到处理，而且很多情况下，有些长进程是非常重要的进程

4）优先级调度算法，优先处理优先级高的进程，局限性就是优先级低的线程长期处理饥饿状态。

5）混合调度算法

## 1.2.NIO

### 1.2.1.NIO概述

Non-Blocking I/O，是一种非阻塞通信模型。不同的语言或操作系统都有其不同的实现。

我们主要学习基于Java语言的NIO，也称为java.nio。

java.nio是jdk1.4版本引入的一套API，我们可以利用这套API实现非阻塞的网络编程模型。

java.bio的通信模型其实就是socket（阻塞通信模型），还有之前学的流（InputStream,OutPutStream都是bio模型）

### 1.2.2.为什么要学习NIO

目前无论是何种应用，都是分布式架构，因为分布式架构能够抗高并发，实现高可用，负载均衡以及存储和处理海量数据，而分布式架构的基础是网络通信。因此， 因此，网络编程始终是分布式软件工程师和架构师的必备高端基础技能之一。

随着当前大数据和实时计算技术的兴起，高性能 RPC 框架与网络编程技术再次成为焦点。 比如Fackebook的Thrift框架，scala的 Akka框架，实时流领域的 Storm、Spark框架，又或者开源分布式数据库中的 Mycat、VoltDB，这些框架的底层通信技术都采用了 NIO（非阻塞通信）通信技术。而 Java 领域里大名鼎鼎的 NIO 框架——Netty，则被众多的开源项目或商业软件所采用。

### 1.2.3.BIO和NIO的区别

|  |  |
| --- | --- |
| BIO | NIO |
| ①流是有方向的，  比如输入流，输出流  ②流的是数据是连续不断的，所以决定了不能根据位置去灵活的操作数据      BIO是一种阻塞通信模型，比如java.bio的socket模型，accept(),connect(),read(),write()方法会产生阻塞。由于阻塞通信，所以BIO的模型是一个请求产生一个线程，所以请求数量越多，线程数量越多。由此带来的问题是内存的占用，以及cpu对应线程的调度管理，包括内存管理，内存碎片，内存环境等等。所以BIO阻塞通信模型不适用于高并发高访问量场景。 | ①通道Channel，在一个通道上，同时可以进行数据的输入和输出。 ②Buffer，传输数据的载体，是一个缓冲区，本质上是一个数组结构。  缓冲区的大小可以自己设定。  此外，注意在实际工作中缓存区最好不要超过32GB  NIO 非阻塞通信模型，可以利用一个线程或少量线程处理多用户请求。 |

### 1.2.4.BIO和NIO的适用场景

NIO的适用场景：高并发，高访问量，短请求

BIO的使用场景：访问量少，长请求（比如下载一个大文件等场景）

BIO实现的Socket——4种产生阻塞的方法

### 1.2.5.BIO 4种产生阻塞的方法：

**ServerDemo1代码（用于测试accpet,connect,read)：**

/\*\*

\* 这个类是作为socket的服务端

\*/

public class ServerDemo1 {

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

ServerSocket ss=new ServerSocket();

ss.bind(new InetSocketAddress(9999));

//accept方法会产生阻塞，直到有客户端连接

//传统的BIO会产生阻塞：

//1.服务端accept()方法会产生阻塞

Socket s=ss.accept();

//接下来做读入流的测试

//当有客户端接入时，accpet()方法不阻塞

//但是客户端没有任何的流输入，所以产生了阻塞

//2.即read()方法也会产生阻塞

InputStream in=s.getInputStream();

System.out.println("1");

in.read();

System.out.println("2");

}

}

**ClientDemo1代码（用于测试accpet,connect,read)：**

/\*\*

\* 这个类是客户端的Socket

\*/

public class ClientDemo1 {

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

//如果服务端未启动，客户端就连接的话会报错，Connection refused

//但是，需要留意的是，这个异常在程序启动后，并不是马上抛出的

//而是卡顿了一秒钟才出现的

//这个现象的原因：

//客户端程序启动=》尝试连接服务端=》等待服务端的链接响应=》服务端没有启动=》

//客户端收到服务端的响应，报出错误提示

//实际上，对应客户端，socket.connect()这个方法也会产生阻塞

Socket socket=new Socket();

socket.connect(new InetSocketAddress("127.0.0.1", 9999));

//while(true){}的意思是让客户端一直保持连接。

while(true){}

}

}

**ServerDemo2代码（用于测试write方法）:**

public class ServerDemo2 {

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

ServerSocket ss =new ServerSocket();

ss.bind(new InetSocketAddress(9998));

Socket s=ss.accept();

while(true){

}

}

}

**ClientDemo2代码（用于测试write方法）:**

/\*\*

\*这个类是用来测试客户端socket write()方法是否阻塞

\*/

public class ClientDemo2 {

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

Socket s=new Socket();

s.connect(new InetSocketAddress("127.0.0.1",9998));

OutputStream out=s.getOutputStream();

for(int i=0;i<1000000;i++){

System.out.println(i);

//结果证明，当不断向outputStream里写数据时，写到一定大小后，会产生阻塞

//即Write方法也会产生阻塞

out.write("a".getBytes());

}

out.flush();

out.close();

}

}

由于底层设备的缓存大小有限，当一直向缓冲区里写数据，却一直不往出取时，达到缓冲区大小上限时，就会造成阻塞，写不出去了。

### 1.2.6.Buffer—ByteBuffer

Buffer：缓冲区,内存里一段连续的空间

Buffer的子类，对应了8种基本数据类型里的七种（没有boolean类型）

ByteBuffer

**1.创建缓冲区**

static ByteBuffer allocate(int capacity)

参数：capacity 缓冲区的容量，以字节为单位

**相关代码：**

@Test

public void testCreateBuffer(){

//ByteBuffer是一个抽象类，我们得到的是他的实现子类对象，HeapByteBuffer

ByteBuffer buffer=ByteBuffer.allocate(1024);

}

**2.向缓冲区里写数据**

put(byte b)

put(byte[] src)

putXxx()

**代码：**

public void testPutData(){

ByteBuffer buffer=ByteBuffer.allocate(1024);

//利用put方法，存入的是字节数据，占一个字节

buffer.put((byte) 1);

//putInf，存入的是整数数据，占四个字节，在开发里，我们操作都是字节数据，所以掌握put方法即可

buffer.putInt(1);

//put方法也可以传入字节数组

buffer.put("123".getBytes());

}

**3.从缓冲区里读数据**

get()

但是，当向缓冲区里写入字节时，比如1，当调用get()方法时，得到的却是0，这是什么原因呢？

**代码：**

@Test

public void testGet(){

ByteBuffer buffer=ByteBuffer.allocate(1024);

buffer.put((byte) 1);

buffer.put((byte) 2);

buffer.put((byte) 3);

//当调用get方法时，之所以是0,是因为，在buffer缓冲区里，有一个position指针，每次put后，position位置+1

//所以，当put完之后，直接调用get()方法，get()是根据最新指针位置来取值的，最新位置肯定是没有数据的，所以是0

System.out.println(buffer.get());

//也可以通过get(position)来读取指定位置的数据

System.out.println(buffer.get(0));

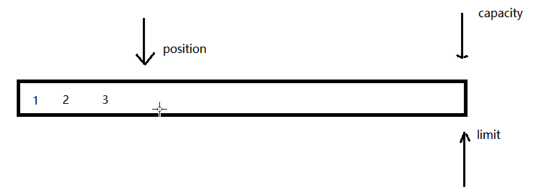
}

**4.Buffter缓冲区的3个关键元素**  
①capacity,缓存区总容量

②limit的大小<=capacity的大小，创建缓冲区时，默认=capacity

③position,初始位置在缓存区的0位，当写入数据时，数据的写入位置就是position的位置

写完后，position的位置+1。



**测试代码：**

@Test

public void testPositionAndLimit(){

ByteBuffer buffer=ByteBuffer.allocate(1024);

buffer.put((byte) 1);

buffer.put((byte) 2);

buffer.put((byte) 3);

//当一个缓冲区创建出来之后，position的初始位置是0,每put()一次或每get()一次，都会使得positon的位置+1

System.out.println("当前的position位置："+buffer.position());

//在写完数据后，将limit设置为当前position位置，然后将position位置重置为0

//这样做的目的是为了从头开始读数据，并且，用limit限制了读取下标，不会造成读出空数据的情况

buffer.limit(buffer.position());

//position位置重置为0

buffer.position(0);

}

Limit 
Position 
Capacity 

**5.filp()方法:**

flip()反转缓冲区，作用相当于buffer.limit(buffer.position());+        buffer.position(0);

**代码：**

@Test

public void testFlip(){

ByteBuffer buffer=ByteBuffer.allocate(1024);

buffer.put((byte) 1);

buffer.put((byte) 2);

buffer.put((byte) 3);

//flip()反转缓冲区，作用相当于buffer.limit(buffer.position());+buffer.position(0);

buffer.flip();

}

**HasRemaining方法：**

告知当前位置（position）和限制位（limit）之间是否还有元素

**代码：**

public void testHasRemaining(){

ByteBuffer buffer=ByteBuffer.allocate(1024);

buffer.put((byte) 1);

buffer.put((byte) 2);

buffer.put((byte) 3);

buffer.flip();

while(buffer.hasRemaining()){

System.out.println(buffer.get());

}

}

**6.rewind（）重绕缓冲区**

将position置为0

**7.clear()清空缓冲区**

@Test

public void testClear(){

ByteBuffer buffer=ByteBuffer.allocate(1024);

buffer.put((byte) 1);

buffer.put((byte) 2);

buffer.put((byte) 3);

//clear的作用是清空缓冲区，但并不真正的清除数据，而是把postion置为0，limit置为容量上限

//所以，当get(0)时，是可以得到原缓冲区数据的。但是我们一般都是在clear()方法之后，写数据，然后filp()

//所以，并不影响缓冲区的重用。

buffer.clear();

System.out.println(buffer.get(0));

}

**8.wrap()数组包装到缓冲区中**

**相关代码：**

@Test

public void testCreateBufferByWrap(){

//wrap方法也可以创建一个Buffer，接收的是一个字节数组，

//并且利用wrap方法创建完buffer之后，buffer里就有了字节数组里的数据

byte[] b={1,2,3,4};

ByteBuffer buffer=ByteBuffer.wrap(b);

ByteBuffer buffer2=ByteBuffer.wrap("helloworld".getBytes());

//也可以通过指定数组下标的界限来创建Buffer并填充数据，需要注意的是：

//利用此方法创建buffer，buffer的容量和数组的容量一致，buffer里也包含了数组的全部数据，

//不同的是limit的位置变化了

ByteBuffer buffer2=ByteBuffer.wrap(b,0,2);

}

Channel

Channel：通道，面向缓冲区，进行双向传输

总接口：Channel

其中重点关注他的4个子类：

操作TCP的

SocketChannel

ServerSocketChannel

操作UDP的

DatagramChannel

操作文件的

FileChannel

### 1.2.7.Socket实现

**1.ServerSocketChannel**

**代码：**

//这个方法用来测试ServerSocketChannel

@Test

public void testServerSocketChannel() throws IOException{

//ServerSocketChannel是一个抽象类，不能直接new,所以调用其静态方法open()

//创建出来的对象是ServerSocketChannelImpl的实例

ServerSocketChannel ssc=ServerSocketChannel.open();

ssc.socket().bind(new InetSocketAddress(9999));

//ServerSocketChannel 创建出来之后，默认是阻塞的，如果要设置成非阻塞模式，需要设置：

//ssc.configureBlocking(false); 属性为false表示非阻塞

ssc.configureBlocking(false);

ssc.accept();

System.out.println("NIO服务端收到客户端请求");

}

**2.SocketChannel**

**代码：**

@Test

public void testSocketChannel() throws Exception{

SocketChannel sc=SocketChannel.open();

//SocketChannel默认也是阻塞的，需要个更改下configureBlocking(false);

sc.configureBlocking(false);

sc.connect(new InetSocketAddress("127.0.0.1",9999));

}

3.read和write方法

**服务端代码（read)代码：**

public class ServerDemo1 {

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

ServerSocketChannel ssc=ServerSocketChannel.open();

ssc.configureBlocking(false);

ssc.socket().bind(new InetSocketAddress(9999));

//ServerSocketChannel不做任何数据上的处理，只是提供通道，负责连接。SocketChannel职责和net包下的socket类似

SocketChannel sc=null;

while(sc==null){

sc=ssc.accept();

}

ByteBuffer buffer=ByteBuffer.allocate(12);

//这里需要注意的是：当read读的时候，需要分配好容量。

//此外，read方法也是非阻塞的，需要用while(buffer.hasRemaining())来确保数据读取完整

while(buffer.hasRemaining()){

sc.read(buffer);

}

System.out.println("服务端接收数据："+new String(buffer.array()));

}

}

**客户端代码（write）代码：**

public class ClientDemo1 {

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

SocketChannel sc=SocketChannel.open();

sc.configureBlocking(false);

sc.connect(new InetSocketAddress("127.0.0.1", 9999));

while(!sc.isConnected()){

sc.finishConnect();

}

ByteBuffer buffer=ByteBuffer.wrap("hellobeijing".getBytes());

while(buffer.hasRemaining()){

//因为write方法是非阻塞的，那就是意味着write方法是否已经将buffer里的数据全部写完都会执行后面的代码

//所以要用到while(buffer.hasRemaining())这种形式来确保buffer数据全部写出

sc.write(buffer);

}

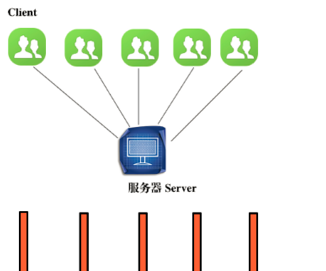
sc.close();

}

}

### 1.2.8.BIO实现

 问题的引入

使用BIO编写代码模拟一下

(编写一个服务器端和客户端程序，运行一次服务器程序，运行四次客户端程序模拟四个用户现场)

|  |
| --- |
| **public** **class** ServerDemo {  **public** **static** **void** main(String[] args) **throws** IOException {  System.***out***.println("服务器启动");  ServerSocket ss = **new** ServerSocket();  ss.bind(**new** InetSocketAddress(7777));  **while**(**true**){  Socket sk = ss.accept();  **new** ClientRunner(sk).start();  }  }  }  **class** ClientRunner **extends** Thread{  **private** Socket s;  **public** ClientRunner(Socket s){  **this**.s = s;  }  **public** **void** run(){  System.***out***.println("负责为客户端提供服务，当前线程的id："+Thread.*currentThread*().getId());  }  } |
| **public** **class** ClientDemo {  **public** **static** **void** main(String[] args) **throws** IOException {  Socket sk = **new** Socket();  sk.connect(**new** InetSocketAddress("127.0.0.1", 7777));  **while**(**true**);  }  } |

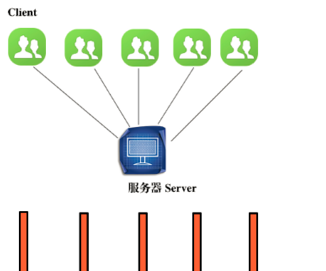
服务器启动

负责为客户端提供服务，当前线程的id：9

负责为客户端提供服务，当前线程的id：10

负责为客户端提供服务，当前线程的id：11

负责为客户端提供服务，当前线程的id：12



分析该模式的缺点：

缺点1：每有一个用户请求，就会创建一个新的线程为之提供服务。当用户请求量特别巨大，线程数量就会随之增大，继而内存的占用增大，所以不适用于高并发、高访问的场景。

缺点2：线程特别多，不仅占用内存开销，也会占用大量的cpu开销，因为cpu要做线程调度。

缺点3：如果一个用户仅仅是连入操作，并且长时间不做其他操作，会产生大量闲置线程。会使cpu做无意义的空转，降低整体性能。

缺点4：这个模型会导致真正需要被处理的线程（用户请求）不能被及时处理。

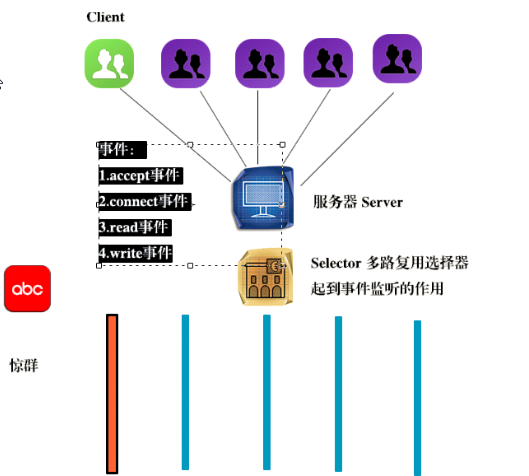
**解决方法**

针对缺点3和缺点4，可以将闲置的线程设置为阻塞态，cpu是不会调度阻塞态的线程，避免了cpu的空转。所以引入事件监听机制实现。

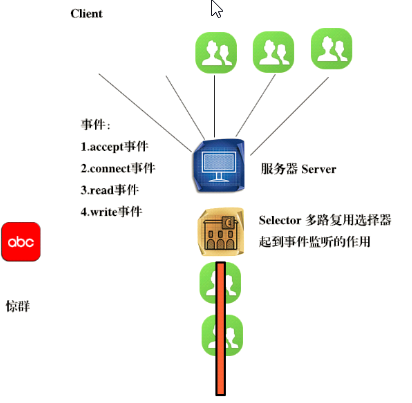
Selector多路复用选择器，起到事件监听的作用。

监听哪个用户执行操作，就唤醒对应的线程执行。那么都有哪些事件呢？

事件：1.accept事件、2.connect事件、3.read事件、4.write事件



针对缺点1和缺点2，可以利用非阻塞模型来实现，利用少量线程甚至一个线程来处理多用户请求。但是注意，这个模型是有使用场景的，适用于大量短请求场景。（比如用户访问电商网站），不适合长请求场景（比如下载大文件，这种场景，NIO不见得比BIO好）



Selector 多路复用器：

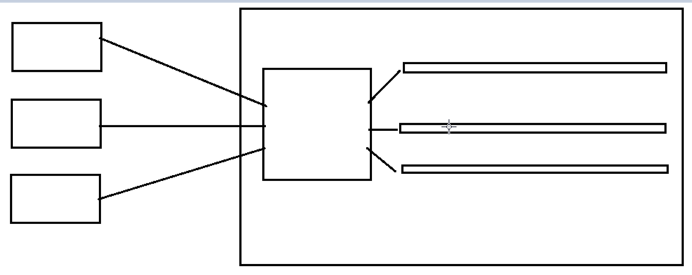
可以理解为路由器和交换机

在一个Selector上可以同时注册多个非阻塞的通道，从而只需要很少的线程数既可以管理许多通道。特别适用于开启许多通道但是每个通道中数据流量都很低的情况

**回顾BIO的缺陷：**

在高并发场景下，BIO的阻塞会产生一些无法解决的问题，

1.一个客户端连接，服务端就会分配一个线程来处理，如果有1000个客户端访问，就会有1000个线程。线程多所造成的问题



2.比如针对聊天程序，可能某一个线程没有做任何事情，却一直连着，导致线程大部门时间都是闲置的状态。

针对上面的模型，我们想出用下面的模型来处理：

计算机生成了可选文字:
一 个 线 程 处 理 多 个 用 户 请 求 

这种模型的核心思想：

有一个关键的元素，选择器。针对上图来说，三个客户端连接请求过来之后，先在选择器上进行注册，然后由选择器来决定当前时刻哪个客户端需要被处理，就交给线程来处理。

这样，就相当用很少的线程来处理很多的客户端的连接请求。

但需要注意，针对上述这种模型，传统的BIO做不了。为什呢？

如果用传统的IO,即阻塞式IO，当选择器选择处理某一个客户端时，可能会因accept,connect,read,write产生阻塞，如果线程产生阻塞，就会导致线程没有机会去处理其他的客户端连接请求。

### 1.2.9.Selector服务通道API

**accept事件**

编写服务器端程序：

|  |
| --- |
| **public** **class** Server {  **public** **static** **void** main(String[] args) **throws** IOException {  ServerSocketChannel ssc = ServerSocketChannel.*open*();  ssc.configureBlocking(**false**);  ssc.socket().bind(**new** InetSocketAddress(9999));  //创建一个Selector  Selector selector = Selector.*open*();  //要想用selector提供监听，要先注册事件，首先为服务器通道设置accept监听  ssc.register(selector,SelectionKey.***OP\_ACCEPT***);  **while**(**true**){  //select()方法会被阻塞，直到有监听的事件被触发  selector.select();  //拿到事件集合  Set<SelectionKey> set = selector.selectedKeys();  //获取事件集合的迭代器  Iterator<SelectionKey> iter = set.iterator();  **while**(iter.hasNext()){  SelectionKey sk = iter.next();  //证明有客户端接入  **if**(sk.isAcceptable()){  //拿到客户端的通道，并由服务端接收  ServerSocketChannel ss = (ServerSocketChannel) sk.channel();  ss.configureBlocking(**false**);  //接收连接  SocketChannel sc = ss.accept();  sc.configureBlocking(**false**);  System.***out***.println("有客户端连入，负责处理该请求的线程id:"+Thread.*currentThread*().getId());  //OP\_READ 0000 0001  //OP\_WRITE0000 0100  //OP\_READ|OP\_WRITE 0000 0101 相对注册了读"或"写的事件监听  sc.register(selector, SelectionKey.***OP\_READ***|SelectionKey.***OP\_WRITE***);  }  **if**(sk.isReadable()){  SocketChannel sc = (SocketChannel)sk.channel();  sc.configureBlocking(**false**);  }  **if**(sk.isWritable()){  }  //删除该事件，防止一个事件被重复处理  iter.remove();  }  }  }  } |

编写客户端代码：

|  |
| --- |
| **public** **class** Client {  **public** **static** **void** main(String[] args) **throws** IOException {  SocketChannel sc = SocketChannel.*open*();  sc.connect(**new** InetSocketAddress("127.0.0.1", 9999));  System.***out***.println("客户端连入");  **while**(**true**);  }  } |

服务器端启动一次，客户端启动三次，服务器端的控制台输出：

|  |
| --- |
| 服务器端启动  有客户端连入，负责处理该请求的线程id:1  有客户端连入，负责处理该请求的线程id:1  有客户端连入，负责处理该请求的线程id:1 |

处理多个请求使用同一个线程。

**read事件**

修改Server类

|  |
| --- |
| **if**(sk.isReadable()){  SocketChannel sc = (SocketChannel)sk.channel();  ByteBuffer buffer = ByteBuffer.*allocate*(10);  sc.read(buffer);  System.***out***.println("服务器端读到的数据："+**new** String(buffer.array()));  System.***out***.println("负责处理读请求的线程id："+Thread.*currentThread*().getId());  //去掉读监听  sc.register(selector, SelectionKey.***OP\_WRITE***);  } |

修改Client类

|  |
| --- |
| System.out.println("客户端连入");  ByteBuffer buffer = ByteBuffer.wrap(  "helloworld".getBytes());  sc.write(buffer);  while(true); |

**write事件**

修改Servet

|  |
| --- |
| **if**(sk.isWritable()){  SocketChannel sc = (SocketChannel)sk.channel();  ByteBuffer buffer = ByteBuffer.*wrap*("收到".getBytes("UTF-8"));  sc.write(buffer);  System.***out***.println("服务器端成功写出数据，负责处理写请求的线程id："+Thread.*currentThread*().getId());  //去掉写监听  sc.register(selector, SelectionKey.***OP\_READ***);  } |

修改Client类

|  |
| --- |
| ByteBuffer reBuf = ByteBuffer.*allocate*(6);  sc.read(reBuf);  System.***out***.println("客户端成功读到数据:"+**new** String(reBuf.array(),"UTF-8"));  while(true); |

### 1.2.10.粘包问题

从缓冲区的角度来看，前一个数据包的尾紧挨着后一数据包的头。（相当于两个数据包的数据 粘在一起了）。

场景

jaryman23#rosewoman26

如果发送的是结构化数据，此时就需要分包

分包策略

1.用标识符来做分割，存在的隐患是分隔符可能是正文内容

2.双方规定好发送的大小，局限性是不灵活，应用范围比较狭窄。

3.采用协议头的思想来做。

**补充：**不是所有的粘包问题都需要处理，比如传输一个文件，只要确保传输的连贯性和完整性 即可，不需要分包。

### 1.2.11.FileChannel

用于读取、写入、映射和操作文件的通道。

1.使用文件通道可以灵活根据位置指针去操作文件，即比基于流的形式操作文件灵活。

注解：当我们要对一个文件进行读取和书写的时候，可以通过指针去指定特定的位置，而流是连续不断的，并不能进行文件的截取，去读写文件的某一个位置。

2.要创建文件通道，如果是基于文件输入流获得的通道，只能读，输出流获得通道只能写。

注解：实际上当我们进行文件的读写的时候，以读取为例，首先输入流inputStream会先要创建一个读取流文件的通道或者是通信过程，先要将文件里的内容放到了通道里，并且人为规定了通道里的缓存区（缓存区实际上是一个缓冲区或者称之为数组结构）的大小来存放文件数据，从而调用文件通道的read方法来读取数据。输出流的性质一样，首先先将文件通过输出流取出来，后将要写入文件里的内容放到缓存区里，通过调用文件通道的write方法来将文件写到通道里。

代码如下：

|  |
| --- |
| /\*\*FileChannel不能单独创建，只能基于某一个流来创建  \* 基于输出流的通道只能写  \* 基于输入流的通道只能读  \*/  **public** **class** FileChannelDemo {  @Test  **public** **void** testWrite() **throws** Exception{  FileOutputStream out = **new** FileOutputStream("1.txt");  FileChannel fc = out.getChannel();  ByteBuffer buffer = ByteBuffer.*wrap*("1234".getBytes());  fc.position(4);  fc.write(buffer);  out.close();  }  @Test  **public** **void** testRead() **throws** Exception{  FileInputStream in = **new** FileInputStream("1.txt");  FileChannel fc = in.getChannel();  ByteBuffer buffer = ByteBuffer.*allocate*(4);  fc.position(4);  fc.read(buffer);  System.***out***.println(**new** String(buffer.array()));  }  /\*\*通过RandomAccessFile对象获取的通道，即可以读也可以写，前提是RandomAccessFile  \* 创建对象时的mode设置为“rw"  \*/  @Test  **public** **void** testRandom() **throws** Exception{  RandomAccessFile raf = **new** RandomAccessFile("1.txt", "rw");  FileChannel fc = raf.getChannel();  ByteBuffer writeBuf = ByteBuffer.*wrap*("5678".getBytes());  fc.position(8);  fc.write(writeBuf);  ByteBuffer readBuf = ByteBuffer.*allocate*(8);  fc.position(4);  fc.read(readBuf);  System.***out***.println(**new** String(readBuf.array()));  }  } |

### 1.2.12.Java NIO的总结

最后总结一下到底NIO给我们带来了些什么：

1）事件驱动模型

2）避免多线程

3）单线程处理多任务

4）非阻塞I/O，I/O读写不再阻塞

5）基于channel的传输，通常比基于流的传输更高效，缓冲区是可以被复用的

6）IO多路复用大大提高了Java网络应用的可伸缩性和实用性

相对于它的老前辈 BIO（阻塞通信）来说，NIO 模型非常复杂，以至于苦学了很久以后也很少有人能够精通它，难以编写出一个没有缺陷、高效且适应各种意外情况的稳定的 NIO 通信模块。所以我们一般使用已有的NIO框架，比如java的Netty框架

之所以会出现这样的问题，是因为 NIO 编程不是单纯的一个技术点，而是涵盖了一系列相关的技术、专业知识、编程经验和编程技巧的复杂工程。

## 1.3.并发工具包

**阻塞队列BlockingQueue**

**所有已知实现类：**

[ArrayBlockingQueue](mk:@MSITStore:F:\\APIdoc\\JdkAPI16.CHM::/java/util/concurrent/ArrayBlockingQueue.html" \o "java.util.concurrent 中的类), [DelayQueue](mk:@MSITStore:F:\\APIdoc\\JdkAPI16.CHM::/java/util/concurrent/DelayQueue.html" \o "java.util.concurrent 中的类), [LinkedBlockingDeque](mk:@MSITStore:F:\\APIdoc\\JdkAPI16.CHM::/java/util/concurrent/LinkedBlockingDeque.html" \o "java.util.concurrent 中的类), [LinkedBlockingQueue](mk:@MSITStore:F:\\APIdoc\\JdkAPI16.CHM::/java/util/concurrent/LinkedBlockingQueue.html" \o "java.util.concurrent 中的类), [PriorityBlockingQueue](mk:@MSITStore:F:\\APIdoc\\JdkAPI16.CHM::/java/util/concurrent/PriorityBlockingQueue.html" \o "java.util.concurrent 中的类), [SynchronousQueue](mk:@MSITStore:F:\\APIdoc\\JdkAPI16.CHM::/java/util/concurrent/SynchronousQueue.html" \o "java.util.concurrent 中的类)

### 1.3.1.ArrayBlockingQueue和LinkedBlockingQueue

|  |
| --- |
| **package** cn.tarean.queue;  **import** java.util.concurrent.ArrayBlockingQueue;  **import** java.util.concurrent.LinkedBlockingQueue;  **import** java.util.concurrent.PriorityBlockingQueue;  **import** java.util.concurrent.TimeUnit;  **import** org.junit.Test;  /\*\*  \* 知识点1:BlockingQueue是一个接口，我们重点学习两个实现子类，  \* 分别是ArrayBlockingQueue和LinkedBlockingQueue  \* 知识点2：ABQ底层的数据结构是数组，此外也兼具了队列的特点(FIFO)  \* 并且ABQ是有界的队列  \* 课外：ABQ支持设定公平策略，即让线程排队消费处理，避免出现饥饿线程。  \*/  **public** **class** TestQueue {  **public** **void** testCreate(){  ArrayBlockingQueue<Object> queue=  **new** ArrayBlockingQueue<>(10,**true**);  }  /\*  \* 知识点3:add()方法在队列满了之后，再次插入会抛queue full的异常  \* 知识点4:offer()在队列满之后，会抛出false。如果未满，则为true  \* 知识点5:put()会产生阻塞，阻塞放开的条件是队里未满（比如消费者消费了数据）  \* 知识点6:offer超时也会产生阻塞，阻塞放开条件有两个：①队列未满，②过了超时时间  \*/  @Test  **public** **void** produceMethod() **throws** Exception{  ArrayBlockingQueue<Object> queue=  **new** ArrayBlockingQueue<>(10);  **for**(**int** i=0;i<10;i++){  queue.add(i);  }  //queue.add(11);  //System.out.println(queue.offer(11));  // queue.put(11);  // System.out.println("hello");  queue.offer(11,3000,TimeUnit.***MILLISECONDS***);  System.***out***.println("hello");  }  /\*  \* 知识点7:LBQ底层的数据结构是链表，同时也满足FIFO原则。  \* 知识点8:LBQ队列是无界队列，所以不需要指定队列的容量。Interger.MaxValue 2^31  \* 知识点9:因为链表的特点是增删性能好，所以LBQ是最经典的队列实现  \*  \* 知识点10:remove 抛出NoSuchElement异常  \* 知识点11:poll会抛null(如果队列为空),如果不为空，则按FIFO原则对应的元素  \* 知识点12:take()会产生阻塞，阻塞放开条件是队列有数据可消费  \* 知识点13:poll超时也会阻塞，阻塞放开条件有两个，①队列有数据 ②过了超时时间  \* 知识点14:阻塞队列通过锁机制保证了线程的并发安全问题，底层用的锁是重入锁  \* 这种锁的吞吐量很高，因为支持非公平锁策略。  \*/  @Test  **public** **void** consumeMethod(){  LinkedBlockingQueue<Object> queue=**new** LinkedBlockingQueue<>();  //queue.remove();  //queue.add(10);  //System.out.println(queue.poll());  }  /\*  \* 知识点15:优先级队列里的元素必须要实现Comparable接口，可以对队列中的元素进行排序  \*/  **public** **class** Student **implements** Comparable<Student>{  **private** String name;  **private** **int** score;  **public** Student(String name,**int** score){  **this**.name=name;  **this**.score = score;  }  @Override  **public** **int** compareTo(Student o) {  **return** o.getScore()-score;  }  @Override  **public** String toString() {  **return** "Student [name=" + name + ", score=" + score + "]";  }  //getters和setters  }  @Test  **public** **void** createPBQ(){  PriorityBlockingQueue<Student> queue=**new** PriorityBlockingQueue<>();  Student s1=**new** Student("tom",50);  Student s2=**new** Student("rose",60);  Student s3=**new** Student("jim",40);    queue.add(s1);  queue.add(s2);  queue.add(s3);    **for**(**int** i=0;i<3;i++){  System.***out***.println(queue.poll());  }  }  } |

### 1.3.2.SynchronousQueue

SynchronousQueue 类实现了 BlockingQueue 接口。 SynchronousQueue 是一个特殊的队列，它的内部同时只能够容纳单个元素。如果该队列已 有一元素的话，试图向队列中插入一个新元素的线程将会阻塞，直到另一个线程将该元素从 队列中抽走。同样，如果该队列为空，试图向队列中抽取一个元素的线程将会阻塞，直到另 一个线程向队列中插入了一条新的元素。

### 1.3.3.并发 Map(映射) ConcurrentMap

介绍见文档。

HashMap非线程并发安全。Hashtable线程并发安全



查看Hashtable类的源码，发现它的读取、添加等方法都添加了同步锁，效率非常低。

那么如何提交并发的效率问题？

如果只锁定操作的一个元素，并发的效率特别高，但是实现太难。

解决思想：引入了分段锁（分段桶）

|  |
| --- |
| **public** **class** TestDemo2 {  /\*\*HashMap是非线程安全的  \* Hashtable:是线程安全的，当一个线程操作Hashtable时，会锁住整表，并且查询时也锁整表。  \* 引入思想：  \* 引入分段锁（分段桶）的概念  \* ConcurrentMap是利用这个思想来做的，分了16个桶（segment）  \* 性能是Hashtable的16倍。  \*/  @Test  **public** **void** testProduce(){  ConcurrentMap<String, String> cmap = **new** ConcurrentHashMap<String,String>();  cmap.put("one", "1");  cmap.put("two", "2");  cmap.put("three", "3");  System.***out***.println(cmap.get("one"));  }  /\*\*根据指定的key来获取符合条件的子map，  \* 方法：  \* headMap(key):获取的小于key  \* subMap(fromKey,toKey)获取的[fromKey,toKey)  \* tailMap(key)获取的[key  \*/  @Test  **public** **void** testNavigable(){  ConcurrentNavigableMap<Integer,String> map = **new** ConcurrentSkipListMap<Integer,String>();  map.put(1, "一");  map.put(2, "二");  map.put(3, "三");  map.put(4, "四");  map.put(5, "五");  ConcurrentNavigableMap<Integer,String> headMap = map.headMap(2);  System.***out***.println(headMap);  ConcurrentNavigableMap<Integer,String> subMap= map.subMap(3, 4);  System.***out***.println(subMap);  ConcurrentNavigableMap<Integer,String> tailMap = map.tailMap(4);  System.***out***.println(tailMap);  }  } |

### 1.3.4.红黑二叉树

在JDK1.8中，引入了红黑树机制。当桶中的元素个数超过8个时候，会扭转成一棵红黑树；如果节点个数不足7个，则会扭转回链表 - 扭转成红黑树的最小桶数为64

红黑树：

1.本质上一棵自平衡二叉查找树

2.查询效率的时间复杂度：O(logn)

3.二叉查找树：

基于二叉树

左子树都小于根，右子树都大于根

4.特点：

所有的节点颜色非红即黑

根节点一定是黑的

红节点的子节点是一定是黑节点，黑节点的子节点可以是红节点也可以是黑节点

最底层的叶子节点一定是黑色的空节点

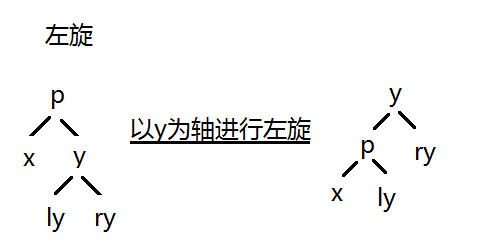
从根节点到任意一个叶子节点经过的黑色节点的个数一定相同，即黑色节点高度一致

新添加的节点的颜色一定是红节点

5.修正 - 红黑树的修正过程是一个链式过程：

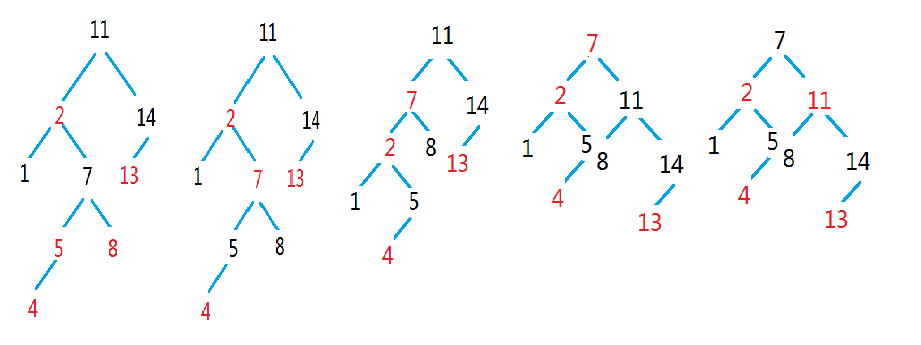
涂色：当前节点为红父节点为红，并且叔父节点为红，那么需要将父节点以及叔父节点涂黑，然后将祖父节点涂红

左旋：当前节点为红父节点为红，叔父节点为黑，并且当前节点为右子叶，那么需要以当前节点为轴进行左旋



右旋：当前节点为红父节点为红，叔父节点为黑，并且当前节点为左子叶，那么需要以父节点为轴进行右旋





### 1.3.5.闭锁CountDownLatch！！！

|  |
| --- |
| **package** cn.tarena.lock;  **import** java.util.concurrent.CountDownLatch;  /\*\*  \* 闭锁，也叫线程递减锁。  \* 做饭的场景  \* 比如启动三个线程  \* 线程1：买菜  \* 线程2：买锅  \* 线程3：炒菜  \*/  **public** **class** TestCountDownLatch {  **public** **static** **void** main(String[] args) **throws** Exception {  //--创建闭锁时，要指定一个初始的数量，这个要根据实际情况来定  CountDownLatch cdl=**new** CountDownLatch(2);  **new** Thread(**new** BuyFood(cdl)).start();  **new** Thread(**new** BuyPot(cdl)).start();  //--await()方法会产生阻塞  //--闭锁有个countdown()方法，这个方法每调用一次，初始数量减一，  //当数量变为0时,阻塞放开  cdl.await();  System.***out***.println("开始炒菜");  }  }  **class** BuyFood **implements** Runnable{  **private** CountDownLatch cdl;  **public** BuyFood(CountDownLatch cdl) {  **this**.cdl=cdl;  }  @Override  **public** **void** run() {  System.***out***.println("菜买回来了");  cdl.countDown();  }  }  **class** BuyPot **implements** Runnable{  **private** CountDownLatch cdl;  **public** BuyPot(CountDownLatch cdl) {  **this**.cdl=cdl;  }  @Override  **public** **void** run() {  System.***out***.println("锅买回来了");  cdl.countDown();  }  } |

CountDownLatch 以一个给定的数量初始化。countDown() 每被调用一次，这一数量就减一。 通过调用 await() 方法之一，线程可以阻塞等待这一数量到达零。

### 1.3.6.栅栏 CyclicBarrier

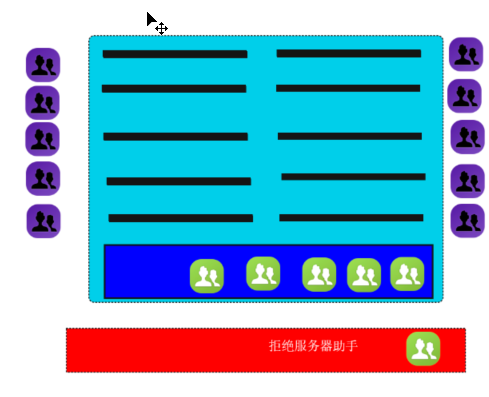
|  |
| --- |
| **package** cn.tarena.lock;  **import** java.util.concurrent.BrokenBarrierException;  **import** java.util.concurrent.CyclicBarrier;  /\*\*  \* 测试栅栏的使用  \* 场景:赛马，比如有两匹赛马  \* 线程1:赛马1  \* 线程2:赛马2  \* 要求就是所有的赛马到到达栅栏前，才能一起跑  \*/  **public** **class** TestCyclicBarrier {  **public** **static** **void** main(String[] args) {  CyclicBarrier cb=**new** CyclicBarrier(2);  **new** Thread(**new** Horse1(cb)).start();  **new** Thread(**new** Horse2(cb)).start();  }  }  **class** Horse1 **implements** Runnable{  **private** CyclicBarrier cb;  **public** Horse1(CyclicBarrier cb) {  **this**.cb=cb;  }  @Override  **public** **void** run() {  System.***out***.println("赛马1来到栅栏前");  **try** {  //--await()会产生阻塞，阻塞放开的条件是初始数量减为0  //此外，await()每调用一次，初始数量就会递减1。  cb.await();  } **catch** (InterruptedException | BrokenBarrierException e) {  e.printStackTrace();  }  System.***out***.println("赛马1开始跑");  }  }  **class** Horse2 **implements** Runnable{  **private** CyclicBarrier cb;  **public** Horse2(CyclicBarrier cb) {  **this**.cb=cb;  }  @Override  **public** **void** run() {  System.***out***.println("赛马2正在处理肚子的问题");  **try** {  Thread.*sleep*(4000);  System.***out***.println("赛马2来到栅栏前");  cb.await();  System.***out***.println("赛马2开始跑");  } **catch** (Exception e) {  e.printStackTrace();  }  }  } |

### 1.3.7.交换机 Exchanger

|  |
| --- |
| **package** cn.tarena.lock;  **import** java.util.concurrent.Exchanger;  /\*\*  \* 交换机，用于两个线程间的数据交换  \* 场景：两个间谍相互交换暗号  \* 线程1:间谍1,暗号:回眸一笑  \* 线程2:间谍2,暗号:寸草不生  \*/  **public** **class** TestExchanger {  **public** **static** **void** main(String[] args) {  Exchanger<String> ex=**new** Exchanger<>();  **new** Thread(**new** Spy1(ex)).start();  **new** Thread(**new** Spy2(ex)).start();  }  }  **class** Spy1 **implements** Runnable{  **private** Exchanger<String> ex;  **public** Spy1(Exchanger<String> ex) {  **this**.ex=ex;  }  @Override  **public** **void** run() {  String info="回眸一笑";  **try** {  //--exchange()用于向对方线程传递数据  //--exchange()的返回值是对方线程传来的数据  String result=ex.exchange(info);  System.***out***.println("1收到2发来的暗号:"+result);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  }  **class** Spy2 **implements** Runnable{  **private** Exchanger<String> ex;  **public** Spy2(Exchanger<String> ex) {  **this**.ex=ex;  }  @Override  **public** **void** run() {  String info="寸草不生";  **try** {  String result=ex.exchange(info);  System.***out***.println("2收到1发来的暗号:"+result);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  } |

### 1.3.8.线程池

使用线程池的目的：避免频繁的创建和销毁线程。



|  |
| --- |
| **package** cn.tarena.threadpool;  **import** java.util.concurrent.ArrayBlockingQueue;  **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;  **import** java.util.concurrent.LinkedBlockingQueue;  **import** java.util.concurrent.RejectedExecutionHandler;  **import** java.util.concurrent.SynchronousQueue;  **import** java.util.concurrent.ThreadPoolExecutor;  **import** java.util.concurrent.TimeUnit;  **import** org.junit.Test;  /\*\*  \* 学习线程池的使用  \* ①corePoolSize:核心线程数量  \* ②maximumPoolSize:最大线程数量  \* ③keepAliveTime:存活时间  \* ④unit:时间单位  \* ⑤workQueue:队列  \*  \* 知识点1：使用线程池的目的是避免线程的频繁创建和销毁，从而提高性能，节省资源。  \* 知识点2：当线程池中，有新的请求时，当线程数量小于核心线程数量，会创建新的线程来处理请求。  \* 知识点3：当核心线程满时，但等待未满时，有新的请求到达时，会进入队列进行排队等待  \* 知识点4：当核心线程满，队列满时，再有新线程达到时，会产生新线程，  \* 知识点5：当请求超过线程池的处理上限时，就将请求交给拒绝服务器助手来处理  \* 新线程数+核心线程数<=最大线程数  \* 知识点6：keepAliveTime限定的是临时线程的闲置时间，超过此时间，临时线程会被销毁  \* 核心线程是不会被销毁的，除非关闭线程池  \* 知识点7：shutdown()方法关闭线程池，在调用shutdown()后，线程池不会接受新的请求，  \* 知识点8：线程池通过execute方法启动线程  \* 此外，并不是马上关闭，而是等到池里所有线程完成工作后再关闭。  \*/  **public** **class** TestThreadPool {  @Test  **public** **void** testCreate(){  ExecutorService es=**new** ThreadPoolExecutor(  5,10,3000,TimeUnit.MILLISECONDS,  **new** ArrayBlockingQueue<Runnable>(5),**new** RejectedExecutionHandler() {  @Override  **public** **void** rejectedExecution(Runnable r, ThreadPoolExecutor executor) {  System.out.println("当前队列已满，请等待");  }  });  **for**(**int** i=0;i<16;i++){  es.execute(**new** ExRunner());  }  es.shutdown();  **while**(**true**);  }  /\*newCachedThreadPool线程池：  \* 0, Integer.MAX\_VALUE,60L, TimeUnit.SECONDS,  new SynchronousQueue<Runnable>()  \*①没有核心线程  \*②都是临时线程  \*③池子容量非常大(大池子)  小队列  \*大池子小队列的适用场景:很好的响应用户请求，可以处理很高的并发访问。  \*但是注意：如果是高并发但是长请求的话，会造成线程数量一直增大却得不到销毁，可能会造成堆溢出  \*此外，也会带来一定的线程创建和销毁的性能损耗。  \*  \*newFixedThreadPool线程池：  \*nThreads, nThreads,  0L, TimeUnit.MILLISECONDS,  new LinkedBlockingQueue<Runnable>()  \*①都是核心线程，没有临时线程  \*②未设定存活时间（因为没有临时线程）  \*③大队列  \*小池子大队列的适用场景:可以用于缓解高峰时的负载压力，相当于通过队列启动缓冲作用。  \*此外由于核心线程没有销毁的过程，所以性能很好。  \*缺点：可能不能及时的处理用户的请求。  \*  \*在设计线程池的时候，不要设计小池子小队列这种模式。  \*/  **public** **void** testCreate\_Simple(){  ExecutorService es=Executors.newCachedThreadPool();  ExecutorService es2=Executors.newFixedThreadPool(10);  }  }  **class** ExRunner **implements** Runnable{  @Override  **public** **void** run() {  System.out.println("hello");  **try** {  Thread.sleep(Integer.MAX\_VALUE);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  } |

### 1.3.9.Callable

|  |
| --- |
| **package** cn.tarena.thread;  **import** java.util.concurrent.Callable;  **import** java.util.concurrent.ExecutionException;  **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;  **import** java.util.concurrent.Future;  /\*\*  \* 知识点1：如何创建一个线程类，可以继承Thread类或实现Runnable接口  \* 在jdk1.5之后，引入了一个新的线程接口 callable  \* 知识点2：callable的call方法可以抛异常  \* 知识点3：call方法有返回值，返回值类型是由接口的泛型决定的  \* 知识点4：callable线程类只能通过线程池来启动，并且call方法的返回值是可以接到的  \* 知识点5：通过线程池的 submit来启动callable线程类  \* 知识点6：sumbit()返回值的一个Future对象，我们可以通过Future.get()获取call()返回值  \*/  **public** **class** TestCallable {  **public** **static** **void** main(String[] args) **throws** Exception {  ExecutorService es=Executors.*newCachedThreadPool*();  Future<String> future=es.submit(**new** CallRunner());  String result=future.get();  System.*out*.println("call的返回值:"+result);  }  }  **class** CallRunner **implements** Callable<String>{  @Override  **public** String call() **throws** Exception {  System.*out*.println("hello1708");  **return** "success";  }  } |

### 1.3.10.ScheduledExecutorService - 定时执行器服务

|  |
| --- |
| **package** cn.tedu.pool;  **import** java.util.concurrent.Executors;  **import** java.util.concurrent.ScheduledExecutorService;  **import** java.util.concurrent.TimeUnit;  **public** **class** ScheduledExecutorServiceDemo {  **public** **static** **void** main(String[] args) {  // 创建线程池  ScheduledExecutorService ses = Executors.*newScheduledThreadPool*(5);  // callable/command - 要执行的线程  // delay - 延时时间  // unit - 时间单位  // 推迟指定的时间再执行提交的任务  // ses.schedule(new ScheduledRunnable(), 5, TimeUnit.SECONDS);  // 从上一次的起始时间开始计时，推算下一次的起始时间  // 如果线程执行时间大于间隔时间，则以实际执行时间为准  ses.scheduleAtFixedRate(  **new** ScheduledRunnable(),  0, 5, TimeUnit.***SECONDS***);  // 从上一次的结束时间开始计时，推算下一次的起始时间  // ses.scheduleWithFixedDelay(  // new ScheduledRunnable(),  // 0, 5, TimeUnit.SECONDS);  }  }  **class** ScheduledRunnable **implements** Runnable {  @Override  **public** **void** run() {  System.***out***.println("hello~~~");  **try** {  Thread.*sleep*(8000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  }  } |

### 1.3.11.ForkJoinPool - 分叉合并池

1.分叉：将任务进行分解，将一个大任务拆分成多个小的任务分配到不同的核上来执行

2.合并：将分叉的任务的执行结果进行汇总

3.分叉合并的目的：提高CPU的利用率

4.分叉合并是适用于任务量比较大的场景

5.为了防止因为慢任务导致效率降低，采取了work-stealing(工作窃取)策略：即当一个核上的任务执行完成之后，不会空闲下来而是会**随机**的去其他核上去 "偷" 一个任务回来执行

|  |
| --- |
| **package** cn.tedu.pool;  **import** java.util.concurrent.ExecutionException;  **import** java.util.concurrent.ForkJoinPool;  **import** java.util.concurrent.Future;  **import** java.util.concurrent.RecursiveTask;  **public** **class** ForkJoinPoolDemo {  **public** **static** **void** main(String[] args) **throws** InterruptedException, ExecutionException {  **long** begin = System.*currentTimeMillis*();  // long sum = 0;  // for (long i = 1; i < 1000000000L; i++) {  // sum += i;  // }  // System.out.println(sum);  ForkJoinPool pool = **new** ForkJoinPool();  Future<Long> f = pool.submit(**new** Sum(1, 100000000L));  System.***out***.println(f.get());  **long** end = System.*currentTimeMillis*();  System.***out***.println(end - begin);  }  }  // RecursiveTask - 有返回值  // RecursiveAction - 没有返回值  **class** Sum **extends** RecursiveTask<Long> {  **private** **long** start;  **private** **long** end;  **public** Sum(**long** start, **long** end) {  **this**.start = start;  **this**.end = end;  }  @Override  **protected** Long compute() {  **if** (end - start > 10000) {  **long** mid = (start + end) / 2;  Sum left = **new** Sum(start, mid);  Sum right = **new** Sum(mid + 1, end);  // 分叉  left.fork();  right.fork();  // 合并  **return** left.join() + right.join();  } **else** {  **long** sum = 0;  **for** (**long** i = start; i <= end; i++) {  sum += i;  }  **return** sum;  }  }  } |

### 1.3.12.锁 Lock

|  |
| --- |
| **public** **class** TestDemo1 {  **public** **static** String *name* = "李雷";  **public** **static** String *gender* ="男";  **public** **static** **void** main(String[] args) {  **new** Thread(**new** SynRunner1()).start();  **new** Thread(**new** SynRunner2()).start();  }  }  **class** SynRunner1 **implements** Runnable{  **public** **void** run(){  **while**(**true**){  **if**("李雷".equals(TestDemo1.*name*)){  TestDemo1.*name* = "韩梅梅";  TestDemo1.*gender*="女";  }**else**{  TestDemo1.*name* = "李雷";  TestDemo1.*gender*="男";  }  }  }  }  **class** SynRunner2 **implements** Runnable{  **public** **void** run(){  **while**(**true**){  System.***out***.println(TestDemo1.*name*+":"+TestDemo1.*gender*);  }  }  } |

运行程序，控制台可能出现以下结果：

韩梅梅:男 李雷:女

如何解决？我们添加同步代码块。

**synchronized**

|  |
| --- |
| **public** **class** TestDemo1 {  **public** **static** String *name* = "李雷";  **public** **static** String *gender* ="男";  **public** **static** **void** main(String[] args) {  **new** Thread(**new** SynRunner1()).start();  **new** Thread(**new** SynRunner2()).start();  }  }  **class** SynRunner1 **implements** Runnable{  **public** **void** run(){  **while**(**true**){  **synchronized**(TestDemo1.**class**){  **if**("李雷".equals(TestDemo1.*name*)){  TestDemo1.*name* = "韩梅梅";  TestDemo1.*gender*="女";  }**else**{  TestDemo1.*name* = "李雷";  TestDemo1.*gender*="男";  }  }  }  }  }  **class** SynRunner2 **implements** Runnable{  **public** **void** run(){  **while**(**true**){  **synchronized**(TestDemo1.**class**){  System.***out***.println(TestDemo1.*name*+":"+TestDemo1.*gender*);  }  }  }  } |

**Lock lock = new ReentrantLock();**

|  |
| --- |
| **public** **class** TestDemo2 {  **public** **static** String *name* = "李雷";  **public** **static** String *gender* ="男";  **public** **static** **void** main(String[] args) {  /\*\*ReentrantLock:重入锁  \* 重入锁的性能较高，原因是默认使用的非公平锁策略。  \* ReentrantLock(false):非公平锁，默认值  \* ReentrantLock(true):公平锁  \* 注意：使用重入锁，必须注意锁的释放的问题，要在finally中来释放。  \* 补充：同步代码块的锁的释放是由JVM完成的  \* 总结：非公平锁策略是允许插队，能够表现出很高的吞吐量，性能就高，建议使用.  \*/  Lock lock = **new** ReentrantLock();  **new** Thread(**new** SynRunner3(lock)).start();  **new** Thread(**new** SynRunner4(lock)).start();  }  }  **class** SynRunner3 **implements** Runnable{  **private** Lock lock;  **public** SynRunner3(Lock lock) {  **this**.lock = lock;  }  **public** **void** run(){  **while**(**true**){  lock.lock();  **if**("李雷".equals(TestDemo2.*name*)){  TestDemo2.*name* = "韩梅梅";  TestDemo2.*gender*="女";  }**else**{  TestDemo2.*name* = "李雷";  TestDemo2.*gender*="男";  }  lock.unlock();  }  }  }  **class** SynRunner4 **implements** Runnable{  **private** Lock lock;  **public** SynRunner4(Lock lock) {  **this**.lock = lock;  }  **public** **void** run(){  **while**(**true**){  lock.lock();  System.***out***.println(TestDemo2.*name*+":"+TestDemo2.*gender*);  lock.unlock();  }  }  } |



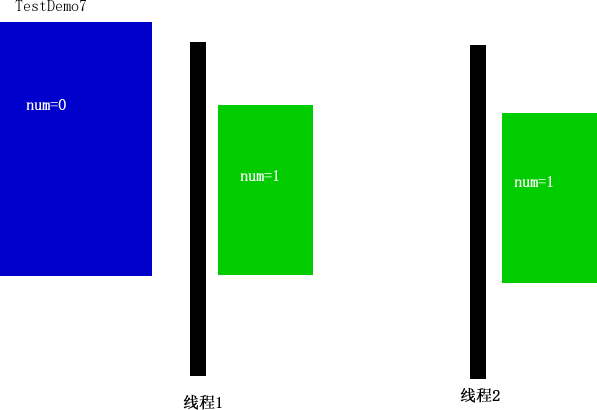
**读写锁ReadWriteLock**

|  |
| --- |
| **public** **class** TestDemo3 {  **public** **static** String *name* = "李雷";  **public** **static** String *gender* ="男";  **public** **static** **void** main(String[] args) {  /\*\*读写锁，更细粒度锁的控制，synchronized关键字和ReentrantLock 同一时间只能有一个线程进行访问被锁定的代码，读写锁的机制则不是，本质上分为两把锁，读锁和写锁，在读锁情况下，多个线程可以并发访问资源，只有当是写锁时只能一个一个的顺序执行，适合 读多写少的场景  \*/  ReadWriteLock lock = **new** ReentrantReadWriteLock();  **new** Thread(**new** SynRunner5(lock)).start();  **new** Thread(**new** SynRunner6(lock)).start();  }  }  **class** SynRunner5 **implements** Runnable{  **private** ReadWriteLock lock;  **public** SynRunner5(ReadWriteLock lock) {  **this**.lock = lock;  }  **public** **void** run(){  **while**(**true**){  lock.writeLock().lock();  **if**("李雷".equals(TestDemo3.*name*)){  TestDemo3.*name* = "韩梅梅";  TestDemo3.*gender*="女";  }**else**{  TestDemo3.*name* = "李雷";  TestDemo3.*gender*="男";  }  lock.writeLock().unlock();;  }  }  }  **class** SynRunner6 **implements** Runnable{  **private** ReadWriteLock lock;  **public** SynRunner6(ReadWriteLock lock) {  **this**.lock = lock;  }  **public** **void** run(){  **while**(**true**){  lock.readLock().lock();  System.***out***.println(TestDemo3.*name*+":"+TestDemo3.*gender*);  lock.readLock().unlock();  }  }  } |

### 1.3.13.原子性

|  |
| --- |
| /\*\*  \* 背景:  \* 使用两个线程同时对一个静态变量做加法运算，比如每个线程执行1万次加法运算  \* 最后查看静态变量的结果是多少  \*  \* 知识点1:原子性类型有很多种，比如原子性整型，原子性布尔类型。  \* 创建的形式:AtomicInteger num=new AtomicInteger(0)  \* 知识点2：比如原子整型的++操作，可以调用addAndGet()  \*/  **public** **class** TestDemo {  **public** **static** **int** *num* = 0;  **public** **static** **void** main(String[] args) **throws** InterruptedException {  CountDownLatch cdl = **new** CountDownLatch(2);  **new** Thread(**new** AddRunner1(cdl)).start();  **new** Thread(**new** AddRunner2(cdl)).start();  cdl.await();  System.***out***.println("num="+*num* );  }  }  **class** AddRunner1 **implements** Runnable{  **private** CountDownLatch cdl;  **public** AddRunner1(CountDownLatch cdl) {  **this**.cdl = cdl;  }  **public** **void** run(){  **for** (**int** i = 0; i < 100000; i++) {  TestDemo.*num*++;  }  cdl.countDown();  }  }  **class** AddRunner2 **implements** Runnable{  **private** CountDownLatch cdl ;  **public** AddRunner2(CountDownLatch cdl) {  **this**.cdl = cdl;  }  **public** **void** run(){  **for** (**int** i = 0; i < 100000; i++) {  TestDemo.*num*++;  }  cdl.countDown();  }  } |

多次测试发现num值总是小于200000。原因：



如何解决？

方法1：使用同步代码块

方法2：使用锁。

方法3：原子性

修改:

public static int num = 0;

改为：

public static AtomicInteger num = new AtomicInteger(0);

修改两个线程中

TestDemo.num++;

改为：

TestDemo.num.addAndGet();

# 2.Zookeeper

## 2.1.Zookeeper概述

### 2.1.1.Zookeeper介绍

Zookeeper是分布式应用程序的协调服务框架，是Hadoop的重要组件。ZK要解决的问题：

1.分布式环境下的数据一致性。

2.分布式环境下的统一命名服务

3.分布式环境下的配置管理

4.分布式环境下的分布式锁

5.集群管理问题

### 2.1.2.分布式概念（集群）

分布式的思想就是人多干活快，即用多台机器同时处理一个任务。分布式的编程和单机的编程 思想是不同的，随之也带来新的问题和挑战。

### 2.1.3.分布式编程容易出现的问题

1.死锁

2.活锁。活锁定义：在程序里，由于某些条件的发生碰撞，导致重新执行，再碰撞=》再执 行，如此循环往复，就形成了活锁。活锁的危害：多个线程争用一个资源，但是没有任何一个 线程能拿到这个资源。（死锁是有一个线程拿到资源，但相互等待互不释放造成死锁），活锁 是死锁的变种。补充：活锁更深层次的危害，很耗尽Cpu资源（在做无意义的调度）

扩展：zk是根据Google的一篇论文

《The Chubby lock service for loosely coupled distributed systems》

3.需要考虑集群的管理问题，需要有一套机制来检测到集群里节点的状态变化。（可以用心跳 机制来做，但zk不是用心跳机制来做的）

4.如果用一台机器做集群管理，存在单点故障问题，所以针对集群管理，也需要形成一个集群

5.管理集群里Leader的选举问题（要根据一定的算法和规则来选举），包括要考虑Leader挂掉 之后，如何从剩余的follower里选出Leader

6.分布式锁的实现，用之前学的重入锁，同步代码块是做不了的

**Zookeeper的名字**

动物园管理员

## 2.2.Zookeeper单机模式安装

实现步骤：

1. 关闭虚拟机的防火墙 ,执行：service iptables stop  
   1.准备虚拟机，安装并配置jdk，1.6以上  
   配置示例：  
   JAVA\_HOME=/home/software/jdk1.8  
   CLASSPATH=.:$JAVA\_HOME/lib/dt.jar:$JAVA\_HOME/lib/tools.jar  
   PATH=$JAVA\_HOME/bin:$PATH  
   export JAVA\_HOME PATH CLASSPATH  
   2.上传zookeeper的安装包 3.4.7版本  
   3.解压安装 tar -xvf …………  
   4.进入zookeeper安装目录下的conf目录，有一个zoo\_sample.cfg的文件  
    复制一份，并重命名为zoo.cfg文件，这个名字固定写死，因为zookeeper启动会检查这个文件，根据这个配置文件里的信息来启动服务

编辑zoo.cfg，修改

dataDir=/home/software/zookeeper-3.4.7/tmp

注意，此目录需要手动创建出来，之前为临时目录  
5.启动zk

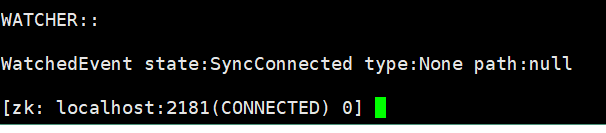
进入bin目录，执行：

sh zkServer.sh start 或者./zkServer.sh start

可以通过jps指令查看活动的java进程

zk的进程是：QuorumPeerMain

6.进入zookeeper客户端，操作zookeeper  
执行：sh zkCli.sh



目录结构：

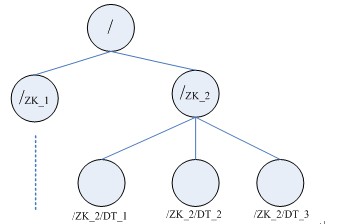
bin 指令

conf 配置文件

lib 运行jar包库

## 2.3.Zookeeper指令与数据结构

Zk数据结构



1. ZK有一个最开始的节点，叫做根节点
2. ZK的节点叫做znode节点
3. 每个znode节点都可存储数据
4. 每个znode节点都可创建自己的子节点
5. 多个znode节点共同形成了znode树
6. Znode树的维系实在内存中，目的是供用户快速的查询
7. 每个znode节点都是一个路径（通过路径来定位这个节点）
8. 每个路径名都是唯一的。
9. znode节点分4类，①普通持久节点（create) ②普通临时节点（create -e)③顺序持久节点（create -s) ④临时顺序（create -e -s )
10. 不要用zookeeper存储大量数据，从功能来讲，zookeeper做的集群的协调服务，所以存储的信息是很少量的。此外，因为znode树是维系在内存中，海量数据也会占用大量的内存。

ZK指令(客户端)

|  |  |
| --- | --- |
| 指令 | 示例 |
| ls查看指令 | ls / |
| create创建节点指令，注意，在创建节点时，要分配初始数据。 | create /zk01  hellobeijing  create /zk02 “” |
| get查看节点数据指令  hellobeijing 数据  cZxid = 0x2 创建此节点的事务id  ctime = Mon May 15 05:58:32 PDT 2018创建节点的时间戳  mZxid = 0x2 修改此节点的事务id  mtime = Mon May 15 05:58:32 PDT 2018修改次节点数据的时间戳  pZxid = 0x2 修改子节点的事务id  cversion = 0 版本号  dataVersion = 0数据版本号，每当数据发生变化，版本号递增1  aclVersion = 0  ephemeralOwner = 0x0 如果此节点不是临时节点，则为0  dataLength = 9数据大小  numChildren = 0子节点个数 | get /zk01 |
| set更新节点数据指令(执行后mtime、dataVersion可能会发生变化，dataLength可能会变化) | set /zk01 hellozk |
| delete删除子节点为空的节点 | delete /zk02 |
| rmr 递归删除指定节点 | rmr /zk02 |
| Create指令补充：   1. 创建子节点 2. Zk节点分四种类型：分别是：   **普通持久节点**:  **普通临时节点**：创建此临时节点的客户端失去和zk连接后，此节点消失.zk是通过临时节点监控哪个服务器挂掉的。  **顺序持久节点**：会根据用户指定的节点路径，自动分配一个递增的顺序号。（顺序节点实现分布式锁的效果，服务器1抢到zk05分配zk050001，服务器2抢到zk05分配zk050002）  **顺序临时节点：** | 1. create /zk01/node01 hello   2.1．create /zk01 hello  2.2. create –e /zk02 abc 2.3.create –s /zk03 abc  2.4.create –s -e /zk05 abcd  zk050000000003  再创建一个就是：  zk050000000004 |
| quit退出zk客户端（或ctrl+c） |  |

## 2.4.Zk API

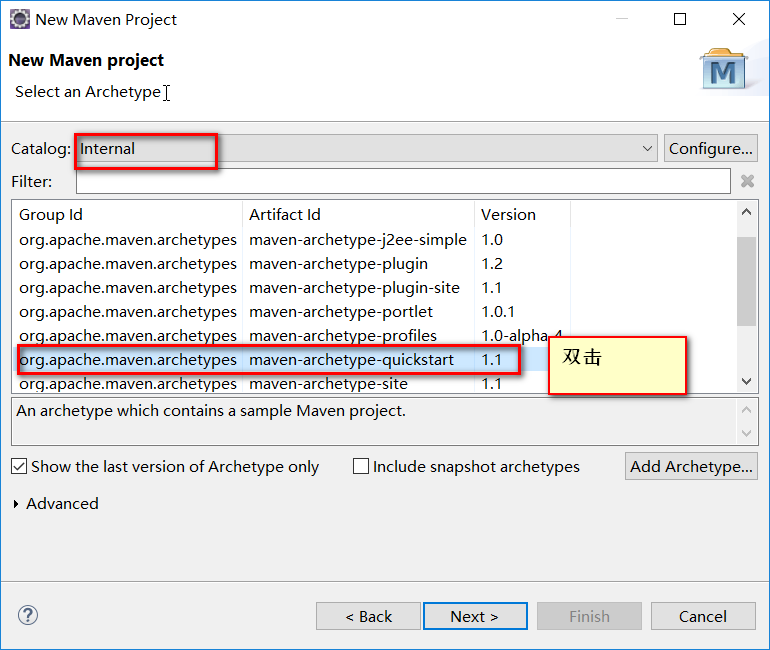
### 2.4.1.项目搭建

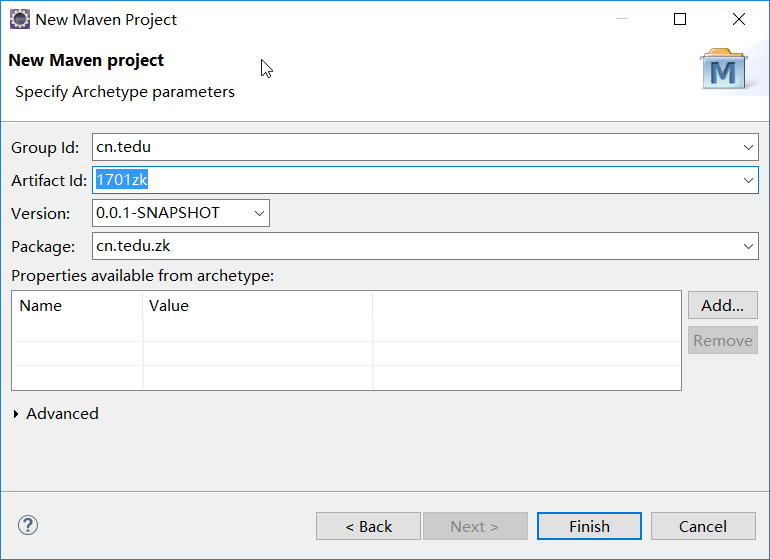
实现步骤：

1. 关闭虚拟机的防火墙 service iptables stop（临时关闭）

永久关闭：chkconfig iptables off

1. 创建一个maven工程

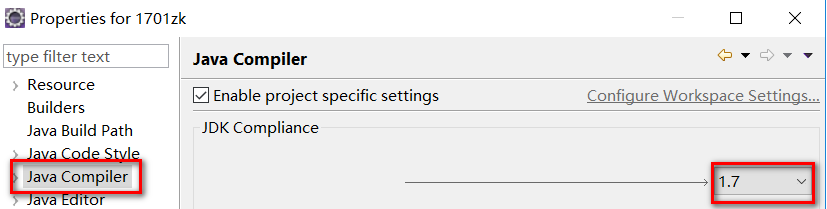




修改maven工程的几个参数

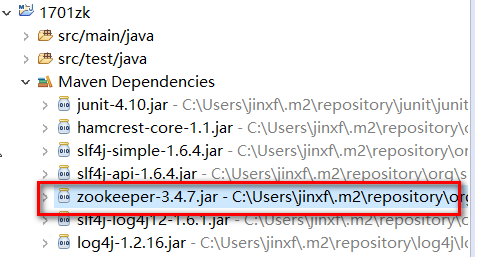
(1).修改默认的jdk，从1.5改为1.7

(2).修改JavaCompiler版本，从1.5改为1.7



点击Apply->OK

(3).从提供的资料中将pom.xml文件拷贝过来替换项目中的同名文件。

zk需要的核心jar包

环境就搭建好了，接下来我们建测试类。（删除不需要的App.java）

### 2.4.2.API演示

**编写测试类，测试连接**

|  |
| --- |
| **package** cn.tedu.zk;  **import** java.io.IOException;  **import** org.apache.zookeeper.WatchedEvent;  **import** org.apache.zookeeper.Watcher;  **import** org.apache.zookeeper.Watcher.Event.KeeperState;  **import** org.apache.zookeeper.ZooKeeper;  **import** org.junit.Test;  **public** **class** TestDemo {  //①connectString:指定zk服务器的ip地址:端口号  //②sessionTimeout:客户端连接服务的超时时间  //③监听器,可以指定事件的发生  @Test  **public** **void** testConect() **throws** IOException{  ZooKeeper zk = **new** ZooKeeper("192.168.80.18:2181",3000,**new** Watcher(){  @Override  **public** **void** process(WatchedEvent event) {  // KeeperState.SyncConnected表示连接成功事件  **if**(event.getState()==KeeperState.***SyncConnected***){  System.***out***.println("成功连接zk服务器");  }  }  });  //测试没有打印连接成功，添加以下代码  **while**(**true**);  //再次测试，连接成功(如果不成功，检查防火墙是否关闭)  }  } |

为何添加while(true)就可以，原因是：

zk连接是一个非阻塞连接方法，连接还没有来的急建立，该线程已经结束。那么如何保证后面的代码都是在连接成功后执行呢，可以使用闭锁。

|  |
| --- |
| **package** cn.tedu.zk;  **import** java.io.IOException;  **import** java.util.concurrent.CountDownLatch;  **import** org.apache.zookeeper.WatchedEvent;  **import** org.apache.zookeeper.Watcher;  **import** org.apache.zookeeper.Watcher.Event.KeeperState;  **import** org.apache.zookeeper.ZooKeeper;  **import** org.junit.Test;  **public** **class** TestDemo {  //①connectString:指定zk服务器的ip地址:端口号  //②sessionTimeout:客户端连接服务的超时时间  //③监听器,可以指定事件的发生  @Test  **public** **void** testConect() **throws** IOException{  **final** CountDownLatch cdl=**new** CountDownLatch(1);  ZooKeeper zk = **new** ZooKeeper("192.168.80.18:2181",3000,**new** Watcher(){  @Override  **public** **void** process(WatchedEvent event) {  // KeeperState.SyncConnected表示连接成功事件 **if**(event.getState()==KeeperState.***SyncConnected***){  System.***out***.println("成功连接zk服务器");  cdl.countDown();  }  }  });  cdl.await();  }  } |

**编写测试节点的CURD**

|  |
| --- |
| @Test  **public** **void** CRUD() **throws** Exception{  **final** CountDownLatch cdl=**new** CountDownLatch(1);  ZooKeeper zk=**new** ZooKeeper("192.168.234.231:2181",30000,**new** Watcher(){  @Override  **public** **void** process(WatchedEvent event) {  **if**(event.getState().equals(KeeperState.***SyncConnected***)){  System.***out***.println("连接成功");  cdl.countDown();  }  }  });  cdl.await();  //参1：path:节点路径  //参2：data:节点数据 要的是字节数组类型  //参3：acl:节点权限 一般赋予此节点所有权限，即所有人对此节点做CRUD:OPEN\_ACL\_UNSAFE  //参4：createMode:节点类型  //PERSISTENT普通持久（持久固定的）  //PERSISTENT\_SEQUENTIAL顺序持久  //EPHEMERAL:普通临时  //EPHEMERAL\_SEQUENTIAL:顺序临时  zk.create("/park06","hello1708".getBytes(),  Ids.***OPEN\_ACL\_UNSAFE***, CreateMode.***EPHEMERAL***);    //--更新节点：①节点路径 ②更新的数据 ③节点的数据版本号  //每次更新节点数据，dataversion都会递增1，此外，如果版本号不一致，会报错  //如果想做到无论如何都更新，版本号写成-1  zk.setData("/park01","hello123".getBytes(),-1);    //--获取节点数据：①节点路径 ②监听器 ③节点状态  **byte**[] data=zk.getData("/park01",**null**,**null**);  System.***out***.println(**new** String(data));    //--删除节点：①节点路径 ②版本号，-1表示无视版本号删除  zk.delete("/park01",-1);    **while**(**true**);  } |

**获取子节点**

|  |
| --- |
| @Test  **public** **void** testGetChild() **throws** Exception{  **省略公用的连接**;  /\* getChildren获取的是子节点的名字，不是路径，  \* 所以在很多场景下，需要自己拼成一个完整路径  \*/  List<String> paths = zk.getChildren("/zk01", **null**);  **for**(String path:paths)  System.***out***.println(path);  } |

**监听节点数据变化**

|  |
| --- |
| **public** **void** testGetDataWatch() **throws** Exception{  **……**  zk.getData("/zk01", **new** Watcher(){  @Override  **public** **void** process(WatchedEvent event) {  //EventType.NodeDataChanged节点数据发送变化的事件  **if**(event.getType()==EventType.***NodeDataChanged***){  System.***out***.println("数据发生变化");  }  }  }, **null**);  **while**(**true**);  } |

运行程序后，在终端上执行以下命令：

set /zk01 newdata

控制台输出“数据发生变化”

目前只能监听一次，如何实现永久监听：

|  |
| --- |
| **for**(;;){  **final** CountDownLatch cdll=**new** CountDownLatch(1);  zk.getData("/zk01",**new** Watcher(){  @Override  **public** **void** process(WatchedEvent event) {  //EventType.NodeDataChanged=》节点数据发生变化的事件 **if**(event.getType()==EventType.***NodeDataChanged***){  System.***out***.println("数据发生变化");  cdll.countDown();  }  }  },**null**);  cdll.await();  }  } |

**监听子节点变化（创建或删除）**

|  |
| --- |
| **public** **void** testGetChildWatch() **throws** Exception{  **。。。。。。**  zk.getChildren("/zk01",**new** Watcher(){  @Override  **public** **void** process(WatchedEvent event) {  //NodeChildrenChanged 子节点事件，包括子节点创建或删除  **if**(event.getType()==EventType.***NodeChildrenChanged***){  System.***out***.println("子节点变化");  }    }  });  **while**(**true**);  } |

运行程序，终端上执行以下命令（在/zk01下创建子节点）

create /zk01/zk02 jxf

控制台输出“子节点变化”

**监听节点删除**

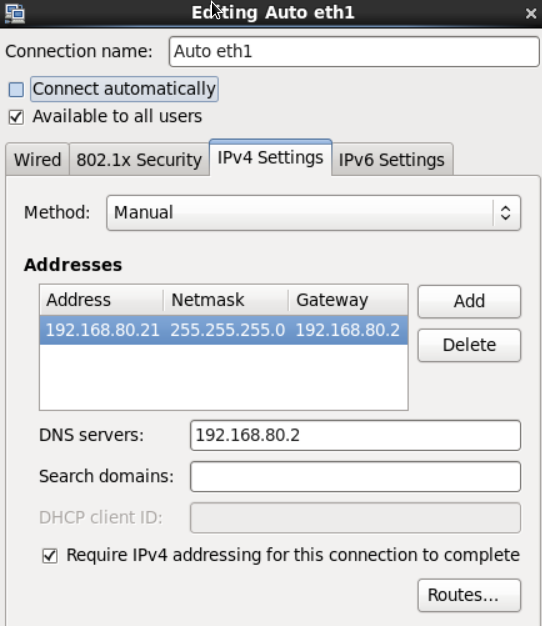
|  |
| --- |
| @Test  **public** **void** testExistWatch() **throws** Exception{  **…….**  zk.exists("/zk02",**new** Watcher(){  @Override  **public** **void** process(WatchedEvent event) {  //EventType.NodeDeleted 节点删除事件  **if**(event.getType()==EventType.***NodeDeleted***){  System.***out***.println("节点已被删除");  }    }    });  **while**(**true**);  } |

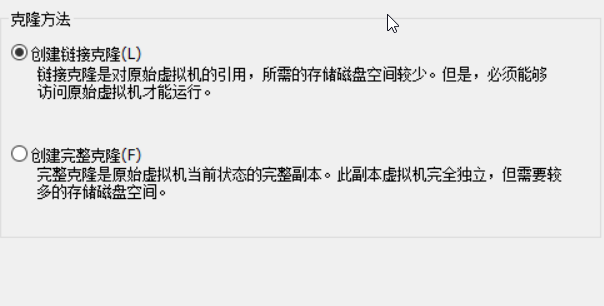
运行程序，在终端上执行删除：delete /zk02

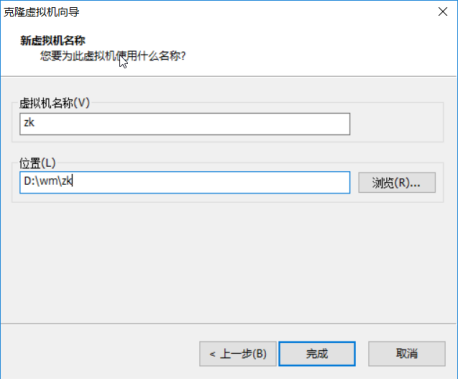
控制台输出：节点被删除

## 2.5.ZK集群搭建和配置

1. 克隆三台空虚拟机（含有lrzsz），修改网络ip，并关闭虚拟机的防火墙







临时关闭：service iptables stop

永久关闭：chkconfig iptables off

1. 安装和配置jdk(克隆的母机中安装配置过jdk，此步可省)
2. 安装和配置zookeeper

mkdir /home/software

tar –zxvf /home/software/zookeeper-3.4.7.tar.gz

[root@localhost conf]# cp zoo\_sample.cfg zoo.cfg

1. 配置zoo.cfg

配置说明：

tickTime=2000 心跳间隔周期 毫秒。

initLimit=10初始连接超时阈值=10\*tickTime。指的是follower初始连接leader的超时时间。 如果网络环境不好，适当调大。

syncLimit=5连接超时阈值=syncLimit\*tickTime。指的是follower和leader做数据交互的超 时时间。如果网络环境不好，适当调大。

dataDir=/home/software/zookeeper-3.4.7/tmp dataDir数据目录指的是zookeeper znode树的 持久化目录，

clientPort=2181

server.1=192.168.80.21:2888:3888

server.2=192.168.80.22:2888:3888

server.3=192.168.80.23:2888:3888

server后的数字是选举id，在选举过程中会用到。注意:数字一定要能比较出大小。

2888 端口原子广播端口，可以自定义

3888 端口选举端口，可以自定义

在zk安装目录下创建tmp文件，创建myid文件（名字固定），并编辑当前虚拟机的选举id

mkdir /home/software/zookeeper-3.4.7/tmp

tmp#vim myid (内容为1)

远程拷贝zk的安装目录到zk2、zk3上

[root@localhost software]# scp -r zookeeper-3.4.7/ [root@192.168.80.22:/home/](mailto:root@192.168.80.22:/home/)software

[root@localhost software]# scp -r zookeeper-3.4.7/ [root@192.168.80.23:/home/](mailto:root@192.168.80.23:/home/)software

分别修改myid,zk2->2,zk3->3

5.启动zk集群测试

分别启动zk

|  |
| --- |
| [root@localhost bin]# ls  README.txt zkCli.cmd zkEnv.cmd zkServer.cmd  zkCleanup.sh zkCli.sh zkEnv.sh zkServer.sh  [root@localhost bin]# ./zkServer.sh start  ZooKeeper JMX enabled by default  Using config: /home/software/zookeeper-3.4.7/bin/../conf/zoo.cfg  Starting zookeeper ... STARTED  [root@localhost bin]# jps  3490 Jps  3464 QuorumPeerMain |

[root@localhost bin]# ./zkServer.sh status

|  |
| --- |
| ZooKeeper JMX enabled by default  Using config: /home/software/zookeeper-3.4.7/bin/../conf/zoo.cfg  Mode: follower |

如果启动顺序为zk1->zk2->zk3,通常zk2为leader。

Leader是如何选举出来的？

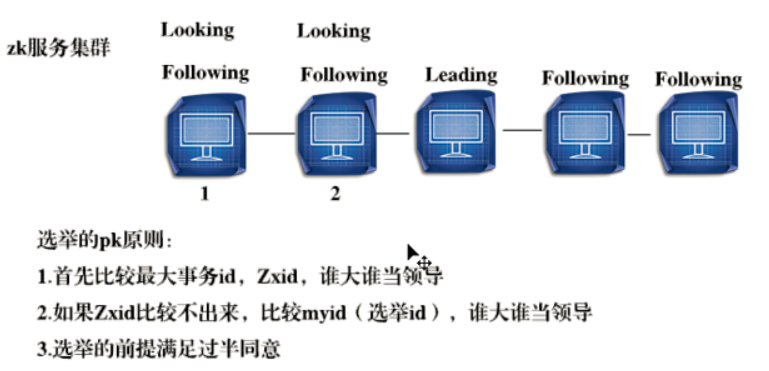
## 2.6.选举机制

### 2.6.1.Zookeeper事务概念

1.每一个写操作都是一个事务，每一个事务都用一个事务id来代表，叫：zxid

2.zxid 是全局唯一，并且全局递增的。作用就是可以根据最大事务id，找到最新的事务

### 2.6.2.Zookeeper选举机制



**选举分两个阶段**

数据恢复阶段：

当zk服务器启动时，会先从本地磁盘找到本机的最大事务id。

选举阶段：

zk服务器会提交选举协议

1.Zxid（最大事务id）

2.本机的选举id（myid文件里的数字）

3.逻辑时钟值 记录当前的选举轮数，确保每个zk在同一轮选举中

4.当前zk服务器状态，分4种： Looking=>选举阶段 Following=>当小弟阶段 Leading=>当领导阶段 Observering=>观察者阶段

**选举的pk原则**

1.首先比较最大事务id，Zxid，谁大谁当领导

2.如果Zxid比较不出来，比较myid（选举id），谁大谁当领导

3.选举的前提满足过半同意

思考：

启动顺序对leader的影响，比如启动顺序变为231、321、312等

3台集群如果挂掉一个能否继续工作，如果挂掉两个呢

集群能不能配置偶数台

答：1、按照比较规则，例如321，事务id相同时，leader就是3

2、挂掉一个可以，挂掉两个就不可以了，因为要满足过半存活机制

可以配置偶数台，但是最好奇数台，更好的满足过半性

**Leader选举出来之后**

首先要做的是数据同步，目的是确保zk集群的数据一致性。一是可以保证当Leader挂掉之后，其他follower可以顶替工作。此外要确保客户端无论从哪个zk服务器获取数据都是一致的。

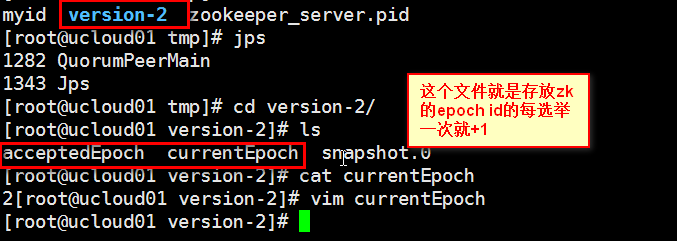
这种实现数据一致的过程称为原子广播（Atomic Brodcast）

对于客户端的读请求，任何一个zk服务器都可以处理，

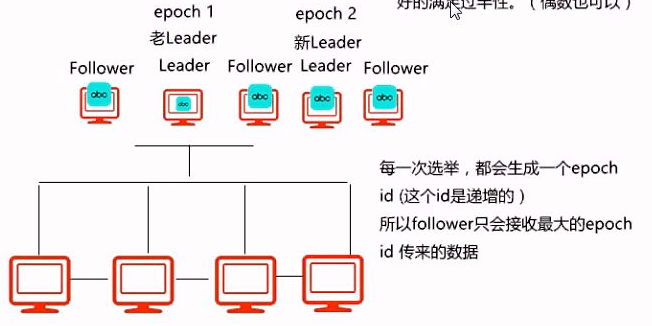
但是对于写请求，会交给Leader来处理，Leader会通过原子广播端口，将写请求（事务）广播给其他的节点，还会收集每个台服务器的ack信息，然后统计是否满足过半性，如果满足，则此事务成功提交。最后反馈客户端一个确认信息。

请记住：过半性（过半选举，过半事务提交，过半存活）

每一次选举，都会生成一个epoch id（这个id是递增的），所以follower只是会接受最大的epoch id传来的数据



注意：当老的leader宕机之后，就会通过选举产生新的leader，就会要求进行数据同步，但是在老的leader重启之后，就会认为自己还是leader，就会要求重新进行数据同步，这个时候就需要引入epoch id，来表明，谁的id最大，就接受谁的数据。详情见下图：



事务id（zsid）是一个64位的2进制数，高32位是epoch id，低32位是事务id

修改testCreateCallBack()方法

|  |
| --- |
| ZooKeeper zk=new ZooKeeper("192.168.80.21:2181,  192.168.80.22:2181,192.168.80.22:2181",3000,new Watcher(){… |

测试发现，三台zk上都创建了该节点。

扩展 Zookeeper的选举机制根据Paxos 算法来实现。 Paxos算法解决的问题：在分布式环境下就某一个决议达成一致性问题。 Paxos算法存在活锁问题，Zk用的是Fast Paxos算法，解决了活锁问题。

## 2.7.Zookeeper集群指令

### 2.7.1.Zookeeper集群命令

可以通过Linux nc 工具来查看Zookeeper集群服务状态（掌握3个即可）

1） 执行：echo stat | nc 127.0.0.1 2181

查看哪个节点（想看哪个节点，就写那个节点的ip即可）被选择作为follower或者leader

Clients:

/127.0.0.1:40743[0](queued=0,recved=1,sent=0)

Latency min/avg/max: 0/0/0

Received: 2

Sent: 1

Connections: 1

Outstanding: 0

Zxid: 0x500000002

Mode: follower

Node count: 8

2） 执行：echo ruok|nc 127.0.0.1 2181

测试是否启动了该Server，若回复imok表示已经启动。

3）echo conf | nc 127.0.0.1 2181 ,输出相关服务配置的详细信息。

clientPort=2181

dataDir=/home/software/zookeeper-3.4.7/tmp/version-2

dataLogDir=/home/software/zookeeper-3.4.7/tmp/version-2

tickTime=2000

maxClientCnxns=60

minSessionTimeout=4000

maxSessionTimeout=40000

serverId=1

initLimit=10

syncLimit=5

electionAlg=3

electionPort=3888

quorumPort=2888

peerType=0

4）echo kill | nc 127.0.0.1 2181 ,关掉server

### 2.7.2.Zookeeper配置详解

|  |  |
| --- | --- |
| **参数名** | **说明** |
| clientPort | 客户端连接server的端口，即对外服务端口，一般设置为2181吧。 |
| dataDir | 存储快照文件snapshot的目录。默认情况下，事务日志也会存储在这里。  ZK会在特定条件下会触发一次快照（snapshot），将当前服务节点的状态以快照文件的形式dump到磁盘上去，即snapshot文件。此外，每生成一次快照文件，就会生成一个对应的事务日志文件 |
| dataLogDir | 事务日志输出目录。  正常运行过程中，针对所有事务操作，在返回客户端“事务成功”的响应前，ZK会确保已经将本次事务操作的事务日志写到磁盘上，只有这样，事务才会生效。 |
| tickTime | ZK中的一个时间单元。ZK中所有时间都是以这个时间单元为基础，进行整数倍配置的。例如，session的最小超时时间是2\*tickTime。 |
| initLimit | Follower在启动过程中，会从Leader同步所有最新数据，然后确定自己能够对外服务的起始状态。Leader允许F在 **initLimit** 时间内完成这个工作。通常情况下，我们不用太在意这个参数的设置。如果ZK集群的数据量确实很大了，F在启动的时候，从Leader上同步数据的时间也会相应变长，因此在这种情况下，有必要适当调大这个参数了。 **默认是：10\*ticktime** |
| syncLimit | 在运行过程中，Leader负责与ZK集群中所有机器进行通信，例如通过一些心跳检测机制，来检测机器的存活状态。如果L发出心跳包在syncLimit之后，还没有从F那里收到响应，那么就认为这个F已经不在线了。 **默认是：5\*ticktime** |
| minSessionTimeout  maxSessionTimeout | Session超时时间限制，如果客户端设置的超时时间不在这个范围，那么会被强制设置为最大或最小时间。默认的Session超时时间是在2 \*  **tickTime ~ 20 \* tickTime 这个范围** |
| snapCount | 每进行snapCount次事务日志输出后，触发一次快照(snapshot), 此时，ZK会生成一个snapshot.\*文件，同时创建一个新的事务日志文件log.\*。默认是100000。这是一种情况  此外，在产生新Leader时，也会生成新的快照文件，（同时会生成对应的事务文件） |
| autopurge.purgeInterval | 在上文中已经提到，3.4.0及之后版本，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，需要配置一个1或更大的整数，默认是0，表示不开启自动清理功能。  注意：如果是手动清理，不要清理掉最新的事务日志文件和快照文件。  比如在做集群数据迁移时，只需要把这两个文件拷贝即可。 |
| server.x=[hostname]:nnnnn[:nnnnn] | 这里的x是一个数字，与myid文件中的id是一致的。右边可以配置两个端口，第一个端口用于F和L之间的数据同步和其它通信，第二个端口用于Leader选举过程中投票通信。 |
| jute.maxbuffer | 每个节点最[大数据](http://lib.csdn.net/base/20)量，是默认是1M。这个限制必须在server和client端都进行设置才会生效。 |
| globalOutstandingLimit | 最大请求堆积数。默认是1000。ZK运行的时候， 尽管server已经没有空闲来处理更多的客户端请求了，但是还是允许客户端将请求提交到服务器上来，以提高吞吐性能。当然，为了防止Server内存溢出，这个请求堆积数还是需要限制下的。超过则拒绝 |
| preAllocSize | 预先开辟磁盘空间，用于后续写入事务日志。默认是64M，每个事务日志大小就是64M。如果ZK的快照频率较大的话，建议适当减小这个参数。 |
| traceFile | 用于记录所有请求的log，一般调试过程中可以使用，但是生产环境不建议使用，会严重影响性能。 |
| electionAlg | 默认为3，即 fast paxos election 选举算法。在3.4版本后，1 2对应的选举算已弃用，所以此项配置不要更改。 |
| leaderServes | 默认情况下，Leader是会接受客户端连接，并提供正常的读写服务。但是，如果你想让Leader专注于集群中机器的协调，那么可以将这个参数设置为no，这样一来，会提高整个zk集群性能。 |

## 2.8.Zookeeper观察者

### 2.8.1.观察者概述

虽然客户端直接连接到投票选举的Zookeeper成员执行良好，但这个架构很难扩展到大量的客户端。问题就是因为我们添加了更多的投票成员，写入性能下降。这是由于这样的事实：一个写入操作要求共识协议至少是整体的一半，因此投票的成本随着投票者越多会显著增加。

我们引入了一个新的Zookeeper节点类型叫做Observer，它帮助处理这个问题并进一步完善了Zookeeper的可扩展性。观察者不参与投票，它只监听投票的结果。除了这个简单的区别，观察者精确的和追随者一样运行 - 客户端可能链接他们并发送读取和写入请求。观察者像追随者一样转发这些请求到领导者，而他们只是简单的等待监听投票的结果。正因为如此，我们可以尽可能多的增加观察者的数量，而不影响投票的性能。

观察者还有其他优势。因为他们不投票，他们不是Zookeeper整体的主要组件。因此他们可以故障，或者从集群断开连接，而不影响Zookeeper服务的可用性。对用户的好处是观察者可以连接到比追随者更不可靠的网络。事实上，观察者可以用于从其他数据中心和Zookeeper服务通信。观察者的客户端会快速的读取，因为所有的读取都在本地，并且写入导致最小的网络开销，因为投票协议所需的消息数量更小。

**observer观察者的特点：   
1.不参与投票   
2.观察和监听投票结果，然后去更新状态。**

### 2.8.2.怎么使用观察者

使用观察者设置Zookeeper全员非常简单，只需要在原来的配置文件上改两个地方。

第一，在要设置的那个节点的配置文件设置为观察者，必须放置这一行：(先停止集群所有机器，如果是三台机器，可以修改其中一个follower为观察者)

peerType=observer

 在zoo.cfg文件下进行配置

计算机生成了可选文字:
# The number of ticks that the initial 
# synchronization phase C a n take 
InltL1m1t— 
# The number Of ticks that C a n pass b 巳 t 谣 巳 巳 n 
# sending a request and getting a n acknowledg 
syncL1m1t— 
# the directo ry Vhere the snapshot is sto red ． 
# do not u s 巳 /tmp fo 「 storage, /tmp here j 
sakes. 
7ataDir= ． h s f 。 4 。 7/tmp 

这一行告诉Zookeeper的服务是一个观察者。

1. 在每个服务配置文件里，必须在观察者定义行添加:observer(所有机器都要配置)。

例如：server.3=localhost:2181:3181:observer

计算机生成了可选文字:
# Pu 「 g 巳 task interval in hours 
# Set t 0 《 ℃ “ t 0 disable a ut 0 purge featu re 
#autopurge.purgelnterval= 
S · 1 
S e rVe r · 2 
Server · 3 = 
1g2 。 1 巧 。 234 。 1 ： 2 ： 3 ： observer 

这个告诉其他服务server.3是一个观察者，并且他们不需要期望他选举。这是你在Zookeeper集群中添加观察者需要的所有配置。

可以启动集群并查看状态，发现配置观察者的机器状态已经变为observer

## 2.9.Zookeeper特性总结

**数据一致性**

client不论连接到哪个Zookeeper，展示给它都是同一个视图，即查询的数据都是一样的。这是zookeeper最重要的性能。

**原子性**

对于事务决议的更新，只能是成功或者失败两种可能，没有中间状态。要么都更新成功，要么都不更新。即，要么整个集群中所有机器都成功应用了某一事务，要么都没有应用，一定不会出现集群中部分机器应用了改事务，另外一部分没有应用的情况。

**可靠性**

一旦服务端成功的应用了一个事务，并完成对客户端的响应，那么该事务所引起的服务端状态变更将会一直保留下来，除非有另一个事务又对其进行了改变。

**实时性**

Zookeeper保证客户端将在非常短的时间间隔范围内获得服务器的更新信息，或者服务器失效的信息，或者指定监听事件的变化信息。（前提条件是：网络状况良好）因为znode树是维系在内存。所以不能用zk存储大量数据。

**顺序性**

如果在一台服务器上消息a在消息b前发布，则在所有Server上消息a都将在消息b前被发布。

**过半性**

zookeeper集群必须有半数以上的机器存活才能正常工作。因为只有满足过半数，才能满足选举机制选出Leader。因为只有过半，在做事务决议时，事务才能更新。

所以一般来说，zookeeper集群的数量最好是奇数个。

## 2.10.zookeeper的应用场景

**统一命名服务（Name Service）**

分布式应用中，通常需要有一套完整的命名规则，既能够产生唯一的名称又便于人识别和记住， 就像数据库中产生一个唯一的数字主键一样。我们利用Zookeeper 可以轻松实现这一功能——利用znode路径。因为znode路径是全局唯一的（可以用某个路径来代表服务名）。

**集群管理**

通过zookeeper知道集群里机器的状态，实现思路：集群里每台机器都在zookeeper里注册自己的临时节点，并上传自己的运行状态，我们可以查看这些临时节点，来得知节点的数据信息，假如某个临时节点消失了，意味着这台节点挂掉了，从而也可以达到集群监控的目的。

**数据订阅发布**

这个最典型的就是集群配置信息要发布到集群的客户机节点上，实现配置信息的集中式管理和动态更新。

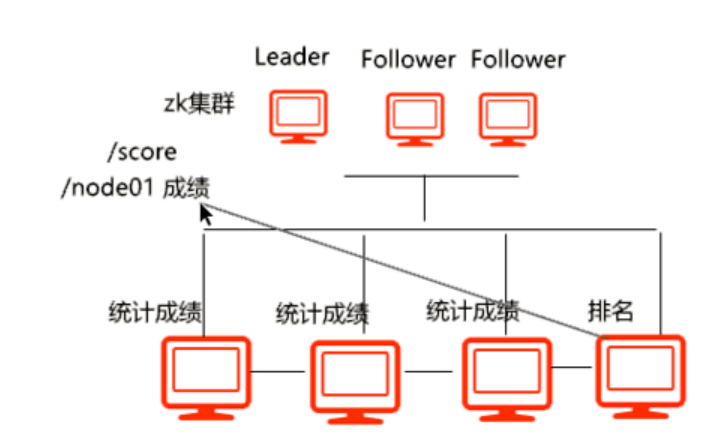
**配置管理（Configuration Management）**

配置的管理在分布式应用环境中很常见，例如同一个应用系统需要多台 PC Server 运行，但是它们运行的应用系统的某些配置项是相同的，如果要修改这些相同的配置项，那么就必须同时修改每台运行这个应用系统的 PC Server，这样非常麻烦而且容易出错。

像这样的配置信息完全可以交给 Zookeeper 来管理，将配置信息保存在 Zookeeper 的某个目录节点中，然后将所有需要修改的应用机器监控配置信息的状态，一旦配置信息发生变化，每台应用机器就会收到 Zookeeper 的通知，然后从 Zookeeper 获取新的配置信息应用到系统中。

**分布式同步协调通知**

可以联想赛马的案例，在分布式环境下，这种效果称为屏障。



**任务分配（后面案例讲）**

**负载均衡**

实现思路：每台机器可以定期将自己的状态信息注册zk服务上（比如cpu，带宽，磁盘等使用率）zk可以根据这些信息做负载均衡。

**分布式锁**

可以通过顺序节点来实现。比如多个客户端同时抢同一个节点，可以为每个客户端创建临时顺序节点，按照顺序进行分配

## 2.11.2PC算法

**分布式下的数据一致性问题**

对于一个将数据副本分布在不同分布式节点上的系统来说，如果对第一个节点的数据进行了更新操作并且更新成功后，却没有使得第二个节点上的数据得到相应的更新，于是在对第二个节点的数据进行读取操作时，获取的依然是老数据（或称为脏数据），这就是典型的分布式数据不一致情况。

为了解决分布式一致性问题，在长期的探索研究过程中，涌现出了一大批经典的一致性协议和算法，其中最著名的就是二阶段提交协议、三阶段提交协议和Paxos算法。

**2PC**

2PC，是Two-Phase Commit的缩写，即二阶段提交，是计算机网络尤其是在数据库领域内，为了使基于分布式系统架构下的所有节点在进行事务处理过程中能够保持原子性和一致性而设计的一种算法。通常，二阶段提交协议也被认为是一种一致性协议，用来保证分布式系统数据的一致性。目前，绝大部分的关系型数据库都是采用二阶段提交协议来完成分布式事务处理的，利用该协议能够非常方便地完成所有分布式事务参与者的协调，统一决定事务的提交或回滚，从而能够有效地保证分布式数据一致性，因此二阶段提交协议被广泛地应用在许多分布式系统中。

**提交过程**

二阶段提交协议是将事务的提交过程分成了两个阶段来进行处理，其执行流程如下。

**阶段一：提交事务请求+执行事务**

1. 事务询问。

协调者向所有的参与者发送事务内容，询问是否可以执行事务提交操作，并开始等待各参与者的响应。

2. 执行事务。

各参与者节点执行事务操作，并将Undo和Redo信息记入事务日志中。

3. 各参与者向协调者反馈事务询问的响应。

如果参与者成功执行了事务操作，那么就反馈给协调者Yes响应，表示事务可以执行；如果参与者没有成功执行事务，那么就反馈给协调者No响应，表示事务不可以执行。

由于上面讲述的内容在形式上近似是协调者组织各参与者对一次事务操作的投票表态过程，因此二阶段提交协议的阶段一也被称为“投票阶段”，即各参与者投票表明是否要继续执行接下去的事务提交操作。

**阶段二：事务提交**

在阶段二中，协调者会根据各参与者的反馈情况来决定最终是否可以进行事务提交操作，正常情况下，包含以下两种可能。

**执行事务提交**

假如协调者从所有的参与者获得的反馈都是Yes响应，那么就会执行事务提交。

1. 发送提交请求。

协调者向所有参与者节点发出Commit请求。

2. 事务提交。

参与者接收到Commit请求后，会正式执行事务提交操作，并在完成提交之后释放在整个事务执行期间占用的事务资源。

3. 反馈事务提交结果。

参与者在完成事务提交之后，向协调者发送Ack消息。

4. 完成事务。

协调者接收到所有参与者反馈的Ack消息后，完成事务。

**中断事务**

假如任何一个参与者向协调者反馈了No响应，或者在等待超时之后，协调者尚无法接收到所有参与者的反馈响应，那么就会中断事务。

1. 发送回滚请求。

协调者向所有参与者节点发出Rollback请求。

2. 事务回滚。

参与者接收到Rollback请求后，会利用其在阶段一中记录的Undo信息来执行事务回滚操作，并在完成回滚之后释放在整个事务执行期间占用的资源。

3. 反馈事务回滚结果。

参与者在完成事务回滚之后，向协调者发送Ack消息。

4. 中断事务。

协调者接收到所有参与者反馈的Ack消息后，完成事务中断。

以上就是二阶段提交过程中，前后两个阶段分别进行的处理逻辑。简单地讲，二阶段提交将一个事务的处理过程分为了投票和执行两个阶段，其核心是对每个事务都采用先尝试后提交的处理方式，因此也可以将二阶段提交看作一个强一致性的算法，下图分别展示了二阶段提交过程中“事务提交”和“事务中断”两种场景下的交互流程。

计算机生成了可选文字:
事 务 捏 交 
阶 段 一 
Prepar•iA* 
Yes 
0 与 甾 
prepar•i'* 
0 与 0 
阶 段 二 
C “ tn 
Commitii* 
寺 与 者 

计算机生成了可选文字:
事 务 中 斯 
阶 段 一 
Yes 
所 ep 豇 e 请 求 
讲 调 
Pr 叩 a 俺 清 求 
0 与 者 
与 者 
赫 与 者 
R b 求 
Ack 
老 与 甾 
Ro № 茈 k 凿 求 
Ack 
与 者 
RO 腩 茈 k 濱 下 
饬 与 者 

**优缺点**

二阶段提交协议的优点：原理简单，实现方便。

二阶段提交协议的缺点：

同步阻塞：只有处理完一个事务才能再处理下一个请求

单点问题：只是指事务里只有一个leader

脑裂：一个集群中，同时出现了两个Leader，脑裂的危害是会造成集群管理的混乱，并且是不可控的。只要是Master-slave架构，就有可能出现脑裂。 ---就是在一个集群里有两个网络分区，就会有两个leader

太过保守：只要有一个失败，则整个事务失败

## 2.12.Paxos算法



**Leslie Lamport（莱斯利·兰伯特）**

Paxos算法的作者Leslie Lamport（莱斯利·兰伯特）及其对计算机科学尤其是分布式计算领域的杰出贡献。作为2013年的新科图灵奖得主，现年73岁的Lamport是计算机科学领域一位拥有杰出成就的传奇人物，其先后多次荣获ACM和IEEE以及其他各类计算机重大奖项。Lamport对时钟同步算法、面包店算法、拜占庭将军问题以及Paxos算法的创造性研究，极大地推动了计算机科学尤其是分布式计算的发展，全世界无数工程师得益于他的理论，其中Paxos算法的提出，正是Lamport多年的研究成果。

说起Paxos理论的发表，还有一段非常有趣的历史故事。Lamport早在1990年就已经将其对Paxos算法的研究论文The Part-Time Parliament提交给ACM TOCS Jnl.的评审委员会了，但是由于Lamport“创造性”地使用了故事的方式来进行算法的描述，导致当时委员会的工作人员没有一个能够正确地理解他对算法的描述，时任主编要求Lamport使用严谨的数据证明方式来描述该算法，否则他们将不考虑接受这篇论文。遗憾的是，Lamport并没有接收他们的建议，当然也就拒绝了对论文的修改，并撤销了对这篇论文的提交。在后来的一个会议上，Lamport还对此事耿耿于怀：“为什么这些搞理论的人一点幽默感也没有呢？”

幸运的是，还是有人能够理解Lamport那公认的令人晦涩的算法描述的。1996年，来自微软的Butler Lampson在WDAG96上提出了重新审视这篇分布式论文的建议，在次年的WDAG97上，麻省理工学院的Nancy Lynch也公布了其根据Lamport的原文重新修改后的Revisiting the Paxos Algorithm，“帮助”Lamport用数学的形式化术语定义并证明了Paxos算法。于是在1998年的ACM TOCS上，这篇延迟了9年的论文终于被接受了，也标志着Paxos算法正式被计算机科学接受并开始影响更多的工程师解决分布式一致性问题。

后来在2001年，Lamport本人也做出了让步，这次他放弃了故事的描述方式，而是使用了通俗易懂的语言重新讲述了原文，并发表了Paxos Made Simple——当然，Lamport甚为固执地认为他自己的表述语言没有歧义，并且也足够让人明白Paxos算法，因此不需要数学来协助描述，于是整篇文章还是没有任何数学符号。好在这篇文章已经能够被更多的人理解，相信绝大多数的Paxos爱好者也都是从这篇文章开始慢慢进入了Paxos的神秘世界。

由于Lamport个人自负固执的性格，使得Paxos理论的诞生可谓一波三折。关于Paxos理论的诞生过程，后来也成为了计算机科学领域被广泛流传的学术趣事。

**拜占庭将军问题的提出**

1982年，Lamport与另两人共同发表了论文The Byzantine Generals Problem，提出了一种计算机容错理论。在理论描述过程中，为了将所要描述的问题形象的表达出来，Lamport设想出了下面这样一个场景：

拜占庭帝国有许多支军队，不同军队的将军之间必须制订一个统一的行动计划，从而做出进攻或者撤退的决定，同时，各个将军在地理上都是被分隔开来的，只能依靠军队的通讯员来进行通讯。然而，在所有的通讯员中可能会存在叛徒，这些叛徒可以任意篡改消息，从而达到欺骗将军的目的。

这就是著名的“拜占庭将军问题”。实际上拜占庭将军问题是一个分布式环境下的协议问题，拜占庭帝国军队的将军们必须全体一致的决定是否攻击某一支敌军。

**Paxos算法的诞生**

Lamport在1990年提出了一个理论上的一致性解决方案，同时给出了严格的数学证明。鉴于之前采用故事类比的方式成功的阐述了“拜占廷将军问题”，因此这次Lamport同样用心良苦地设想出了一个场景来描述这种一致性算法需要解决的问题，及其具体的解决过程：

在古希腊有一个叫做Paxos的小岛，岛上采用议会的形式来通过法令，议会中的议员通过信使进行消息的传递。值得注意的是，议员和信使都是兼职的，他们随时有可能会离开议会厅，并且信使可能会重复的传递消息，也可能一去不复返。因此，议会协议要保证在这种情况下法令仍然能够正确的产生，并且不会出现冲突。

这就是论文The Part-Time Parliament中提到的兼职议会，而Paxos算法名称的由来也是取自论文中提到的Paxos小岛。在这个论文中，Lamport压根没有说Paxos小岛是虚构出来的，而是煞有介事的说是考古工作者发现了Paxos议会事务的手稿，从这些手稿猜测Paxos人开展议会的方法。因此，在这个论文中，Lamport从问题的提出到算法的推演论证，通篇贯穿了对Paxos议会历史的描述。

**算法陈述**

Paxos算法实际上也是一个类2pc算法，而重点是引入了“过半性”的投票理念，通俗地讲就是少数服从多数的原则。此外，Paxos算法支持分布式节点角色之间的轮换，即当协调者出现问题后，参与者可以变成协调者工作。这极大地避免了分布式单点的出现，因此Paxos算法既解决了无限期等待问题，也提高了性能，是目前来说最优秀的分布式一致性协议之一。

Zookeeper用的是fast paxos 算法，如果采用paxos算法，可能会出现活锁问题。解决了活锁问题。此外，Google的Chubby lock service 也基于paxos算法来实现的。

## 2.13.Zookeeper插件

实现步骤：

1.需要联网

2.打开Eclipse=》help=》 install new software

3.点击add，添加如下的url:

***<http://www.massedynamic.org/eclipse/updates/>***

计算机生成了可选文字:
90什以“@ 
囗 洫 “ 
0 

4.选择好插件，三个全部勾选。然后点击next进行安装，（有一步是接收许可协议）

计算机生成了可选文字:
《 nstall Details 
Review the items to be installed. 
0 Masse Dynamic Eclipse Core Library 
0 
Plug-in for JMX 
0 
Plug-in for ZooKeeper 土 2 ． 2 
VerSlOn 
1 ． 0 ． 0 

5.显示正在下载和安装

计算机生成了可选文字:
Installing Software 
Installing Software 
Fetching artifacts.xml.xz from http：//d 一 509040015 / 0.06kB of 55 ． 05kB at 0B/s) 
囗 Always run in background 
Run in Background 
Cancel 
Details > > 

6.调出Zookeeper的视图

计算机生成了可选文字:
， re r 
ZooKeeper 3 ． 2 ． 2 Explorer 

7.新建Zookeeper连接服务，并填好name和 RootPath，然后点击add，添加Zookeeper的连接服务地址，

8.出现下图，证明安装成功

计算机生成了可选文字:
Package Explorer 
奥 My ZooKeeper 
Servers 
eng•nel 
hdfs 
@ jobtracker 
zookeeper 
ZooKeeper 3 ． 2 ． 2 Explorer 

# 3.AVRO

## 3.1.数据序列化

**数据序列化(Serialization)**

数据序列化就是将对象或者数据结构转化成特定的格式，**使其可在网络中传输，或者可存储在内存或者文件中**。反序列化则是相反的操作，将对象从序列化数据中还原出来。

而数据序列化后的数据格式可以是二进制，可以是XML，也可以是JSON等任何格式。

我们要明确的是：数据序列化的重点在于数据的交换和传输。

**序列化的关注点/衡量标准**

1.序列化后的数据大小

因为序列化后的字节流（二进制数据）通常是通过网络进行传输的，因此序列化后的内容越少传输耗时也就越短

2.序列化和反序列化的耗时及占用的cpu

3.是否支持跨语言，跨平台

在异构的网络系统中，网络通信双方可能是不同的语言，比如一端是Java，另一端是C。或者一端是Windows操作系统，另一端是Linux操作。

你可以这样理解：假设你传递的是字符串，没有问题，所有的机器都可以识别正常的字符串。

那么现在假设你传递的参数是一个 Java 对象，比如叫 cat。服务器并没有那么智能，它并不会知道你传递的是一个 Java 对象，而不是其他类型的数据，它识别不了 Java 对象。

因为Java 对象本质上是 class 字节码，服务器并不能根据这个字节码识别出该 Java 对象。所以，要提供一个公共的格式，不仅 Windows 能识别，你的服务器也能识别的公共的格式。

## 3.2.Java原生序列化和反序列化

**序列化**

**Person对象代码（注意要实现Serializable接口）：**

|  |
| --- |
| **package** cn.tedu;  **import** java.io.Serializable;  **public** **class** Person **implements** Serializable{  **private** String name;  **private** **int** age;  **public** String getName() {  **return** name;  }  **public** **void** setName(String name) {  **this**.name = name;  }  **public** **int** getAge() {  **return** age;  }  **public** **void** setAge(**int** age) {  **this**.age = age;  }  } |
| **测试代码：**  /\*这个方法是测试Person对象序列化的，利用的是java提供的原生序列化机制  \*主要掌握的是ObjectOutputStream.writeObject()方法  \*需要注意的是被序列化的对象要实现Serializabel接口  \*/  @Test  **public** **void** testSerialize01() **throws** Exception{  Person p=**new** Person();  p.setAge(23);  p.setName("jary");  ObjectOutputStream oos=**new** ObjectOutputStream(**new**  FileOutputStream("person.data"));  oos.writeObject(p);  oos.flush();  oos.close();  } |

**反序列化**

|  |
| --- |
| **测试代码：**  /\*这个方法是测试Person对象反序列化的，利用的是java原生提供的反序列  \* 需要掌握的是ObjectInputStream.readObject()方法  \*/  @Test  **public** **void** testDeserialize01() **throws** Exception{  ObjectInputStream ois=**new** ObjectInputStream(**new**  FileInputStream("person.data"));  Person p =(Person) ois.readObject();  System.***out***.println(p.getName());  System.***out***.println(p.getAge());  } |

**SerializableId的作用**

|  |
| --- |
| **测试代码：**  /\*  \* 这个方法是测试SerializeId的作用  \* 当Person对象序列化完毕后，如果更改Person类的结构，比如新增一个私有属性等，再次反序列化时会报错。  \* 原因是：当序列化时，会自动生成一个SerializeId,反序列化时，也会生成一个SerializeId，如果更改了Person对 象会导致两个SerializeId不一样，从而反序列化失败。  \* 所以解决办法是：手动加上SerializeId  \*/  @Test  **public** **void** testSerializeId() **throws** Exception{  ObjectInputStream ois=**new** ObjectInputStream(**new**  FileInputStream("person.data"));  Person p =(Person) ois.readObject();  System.***out***.println(p.getAge());  System.***out***.println(p.getName());  } |
| **Person中加入如下代码即可：**  **private** **static** **final** **long** ***serialVersionUID*** = 1500208610485192912L; |

**SerializableId的作用**

在某些场景下，对于某些字段我们不想序列化，出于安全的角度，比如密码。 此时，我们就可以用transient关键字，在序列化时忽略此字段。

|  |
| --- |
| **示例代码：**  **public** **class** Person **implements** Serializable{  **private** **static** **final** **long** ***serialVersionUID*** = 1500208610485192912L;  **private** String name;  **private** **transient** String password; |

**总结Java原生序列化**

1.java原生序列化在实际开发中用的非常少，因为序列化完后的数据文件很大。  
2.不能跨语言。 java序列化后的数据文件，不能用c语言或其他语言还原回来  
3.速度慢  
4.有成千上万(打个比方,不止这么多)的对象要反序列化, 而java序列化机制不能复用对象,java反序列化的时候, 每次要构造出新的对象. 在hadoop的序列化机制中, 反序列化的对象是可以复用的.

## 3.3.AVRO介绍

### 3.3.1.AVRO介绍



**官方网址：**http://avro.apache.org/

Avro是一个数据序列化框架（系统），提供了：

1.丰富的数据结构类型，8种基本数据类型以及6种复杂类型

2.快速可压缩的二进制形式

3.提供容器文件用于持久化数据

4.远程过程调用RPC框架

5.简单的动态语言结合功能，Avro 和动态语言结合后，读写数据文件和使用 RPC 协议都不需要生成代码，而代码生成作为一种可选的优化只值得在静态类型语言中实现。

通过avro，每次进行序列化，根据模式（schema)文件来序列化，可以提高性能。

**同类主流产品**

Google的：Protobuffer

facebook的：Thrift

**Google protobuffer**

**优点**

1）二进制消息，性能好/效率高（空间和时间效率都很不错）

2）proto文件生成目标代码，简单易用

3）序列化反序列化直接对应程序中的数据类，不需要解析后在进行映射(XML,JSON都是这种方式)

4）支持向前兼容（新加字段采用默认值）和向后兼容（忽略新加字段），简化升级

5）支持多种语言（可以把proto文件看做IDL文件）

6）Netty等一些框架集成

**Thrift**

应用

Facebook的开源的日志收集系统(scribe: <https://github.com/facebook/scribe>)

淘宝的实时数据传输平台(TimeTunnel <http://code.taobao.org/p/TimeTunnel/wiki/index>)

Evernote开放接口(https://github.com/evernote/evernote-thrift)

Quora(http://www.quora.com/Apache-Thrift)

HBase( [http://abloz.com/hbase/book.html#thrift](http://abloz.com/hbase/book.html" \l "thrift) )

…

**优点**

支持非常多的语言绑定

thrift文件生成目标代码，简单易用

消息定义文件支持注释

数据结构与传输表现的分离，支持多种消息格式

包含完整的客户端/服务端堆栈，可快速实现RPC

支持同步和异步通信

### 3.3.2.Avro Schema

**Schemas**

Avro是依赖于模式（schema），模式文件是用json格式来表示的。

如果是想利用avro实现序列化或rpc通信，需要遵守schema的格式要求。

基于模式的好处是使得序列化快速而又轻巧。（你可以把模式文件理解为一个模板，基于这个模板可以生成对象）

**Avro简单格式列表（8种）**

|  |  |
| --- | --- |
| **原生类型** | **说明** |
| null | 表示没有值 |
| boolean | 表示一个二级制布尔值 |
| int | 表示32位有符号整数 |
| long | 表示64位有符号整数 |
| float | 表示32位单精度浮点数 |
| double | 表示64位双精度浮点数 |
| bytes | 表示8位无符号字节序列 |
| string | 表示字符序列 |

**Avro复杂格式列表（6种）**

**只需掌握：record**

|  |  |  |
| --- | --- | --- |
| **复杂类型** | **属性** | **说明** |
| **Records** | **type name** | **record** |
|  | **name** | a JSON string providing the name of the record (**required**). |
|  | namespace | a JSON string that qualifies the name(optional). |
|  | doc | a JSON string providing documentation to the user of this schema (optional). |
|  | aliases | a JSON array of strings, providing alternate names for this record (optional). |
|  | **fields** | a JSON array, listing fields (**required**). |
|  | name | a JSON string. |
|  | type | a schema/a string of defined record. |
|  | default | a default value for field when lack. |
|  | order | ordering of this field. |
| **Enums** | **type name** | **enum** |
|  | **name** | a JSON string providing the name of the enum (**required**). |
|  | namespace | a JSON string that qualifies the name. |
|  | doc | a JSON string providing documentation to the user of this schema (optional). |
|  | aliases | a JSON array of strings, providing alternate names for this enum (optional) |
|  | **symbols** | a JSON array, listing symbols, as JSON strings (**required**). All symbols in an enum must be unique. |
| **Arrays** | **type name** | **array** |
|  | items | the schema of the array’s items. |
| **Maps** | **type name** | **map** |
|  | values | the schema of the map’s values. |
| **Fixed** | **type name** | **fixed** |
|  | **name** | a string naming this fixed (**required**). |
|  | namespace | a string that qualifies the name. |
|  | aliases | a JSON array of strings, providing alternate names for this enum (optional). |
|  | **size** | an integer, specifying the number of bytes per value (**required**). |
| **Unions** |  | a JSON arrays |

**Test.avsc文件 所有格式示例**

**说明：Test.avsc文件，利用avro的插件可生成对应的Test 类，这个类可以利用avro的API序列化和反序列化）**

{"namespace": "avro.domain",

 "type": "record",

 "name": "Test",

 "fields": [

   {"name": "stringVar", "type": "string"},

   {"name": "bytesVar", "type": ["bytes", "null"]},

   {"name": "booleanVar",  "type": "boolean"},

   {"name": "intVar",  "type": "int", "order":"descending"},

   {"name": "longVar",  "type": ["long", "null"], "order":"ascending"},

   {"name": "floatVar",  "type": "float"},

   {"name": "doubleVar",  "type": "double"},

   {"name": "enumVar",  "type": {"type": "enum", "name": "Suit", "symbols" : ["SPADES", "HEARTS", "DIAMONDS", "CLUBS"]}},

   {"name": "strArrayVar", "type": {"type": "array", "items": "string"}},

   {"name": "intArrayVar", "type": {"type": "array", "items": "int"}},

     {"name": "mapVar", "type": {"type": "map", "values": "long"}},

   {"name": "fixedVar", "type": {"type": "fixed", "size": 16, "name": "md5"}}

 ]

}

AVRO pom坐标

**坐标**

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>4.10</version>

</dependency>

         <dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-simple</artifactId>

<version>1.6.4</version>

<scope>compile</scope>

</dependency>

<dependency>

<groupId>org.apache.avro</groupId>

<artifactId>avro</artifactId>

<version>1.7.5</version>

</dependency>

<dependency>

<groupId>org.apache.avro</groupId>

<artifactId>avro-ipc</artifactId>

<version>1.7.5</version>

</dependency>

</dependencies>

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-compiler-plugin</artifactId>

<version>2.3.2</version>

</plugin>

<plugin>

<groupId>org.apache.avro</groupId>

<artifactId>avro-maven-plugin</artifactId>

<version>1.7.5</version>

<executions>

<execution>

<id>schemas</id>

<phase>generate-sources</phase>

<goals>

<goal>schema</goal>

<goal>protocol</goal>

<goal>idl-protocol</goal>

</goals>

<configuration>

<sourceDirectory>${project.basedir}/src/main/avro/</sourceDirectory>

<outputDirectory>${project.basedir}/src/main/java/</outputDirectory>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

### 3.3.3.AVRO实现对象的序列化和反序列化

**模式定义avsc文件**

Avro通过avsc文件，会生成对应的java类，这个java类专门用来被avro实现序列化和反序列化。

**实现步骤：**

1. 在工程下创建src\main\avro 的maven目录（注意环境，java build path和java compiler改为自己jdk版本）
2. 将pom.xml拷贝到项目中

如果有报错，并非文件错误，maven自身问题，可以通过设置：

window->preferences->Maven->Errors/Warnings中全部设置为Ignore即可

1. 在项目中创建源目录：/src/main/avro
2. 在/src/main/avro目录下建立Xxx.avsc的文件，例如User.avsc
3. 根据avro的schema格式，定义文件avsc文件内容
4. 在项目上右键=》run as=>Maven generate source 如果控制台输出BUILD SUCCESS，则代表成功，刷新工程，会在生成avsc文件对应的java对象

**user.avsc代码：**

{"namespace": "example.avro", //生成User对象后，所在的包路径

"type": "record", //序列化类型

"name": "User", //类名，User

"fields": [

{"name": "name", "type": "string"}, //属性名 name ,类型 String

{"name": "age", "type": ["int", "null"]} //属性名 age,类型 int，默认值null

]

}

**补充：**如果不用maven来生成，需要手动来处理，操作如下：

①下载这4个jar包，并放在同一目录下

计算机生成了可选文字:
jackson-mapper-asl-l 四 的 3.j 
avro 17 · 5.j 
avro-tools-l .7.4.jar 
Jackson-core-asl-l 的 3.jar 

②把user.avsc文件放在此目录下

③在此目录下进入cmd命令窗口

④执行：java -jar avro-tools-1.7.4.jar compile schema Person.avsc ./

计算机生成了可选文字:
教 字 (D:) 
名 称 
avro-l .7.5.Jar 
“ ro 工 冥 包 及 使 说 明 
0 
囗 
0 
avro-tools-l .7.4.jar 
avrol#ih*.txt 
DemoService.avdl 
jackson-core-asl-l ． 90 3.jar 
jackson-mapper-asl-l ． 90 3.jar 
[ 」 QueryParameter.avsc 
修 改 日 耻 
2015 / 8 / 24 12 ： 12 
2015 / 8 / 24 1 3 ： 05 
2015 / 8 / 24 1335 
2015 / 8 / 24 13 ： 22 
2015 / 4 / 23 20 ： 10 
2015 / 5 / 1 14 ： 45 
2015 / 8 / 24 1 2 ： 37 
2015 / 8 / 24 12 ： 37 
Executable Jar File 
Executable Jar File 
文 本 文 档 
AVDL 文 亻 牛 
Executable Jar File 
Executable Jar File 
AVSC2 亻 牛 
AVSC 文 亻 牛 
392 KB 
1L535 KB 
1 KB 
1 KB 
227 KB 
753 KB 
1 KB 
1 KB 

**7.利用AVRO的API实现对象的序列化和反序列化**

**avro创建对象的三种形式：**

|  |
| --- |
| @Test  **public** **void** create(){  User u1=**new** User();  u1.setName("tom");  u1.setAge(22);    User u2=**new** User("小明",11);    //利用newBuilder(Obeject).setXxx().build()的方式来创建User，更加灵活  //即可以基于某一个对象来创建一个新的对象，如果不做任何修改，就相当于复制这个对象。此外，也可以针对某一个值进行修改  User u3=**new** User().*newBuilder*(u2).setName("小华").build();  System.***out***.println(u3.getName());  System.***out***.println(u3.getAge());  } |

**序列化代码示例：**

|  |
| --- |
| /\*  \* 这个方法是测试利用avro将User对象序列化为本地文件  \* avro序列化的特点是可以根据模式文件做对象结构的复用，这样的好处可以节省序列后的数据大小尤其在对象数量多了之后，优势更为明显。  \*/ @Test  **public** **void** write() **throws** Exception{  DatumWriter<User> dw=**new** SpecificDatumWriter<>(User.**class**);  DataFileWriter<User> dfw=**new** DataFileWriter<>(dw);    User u1=**new** User("tom",23);  User u2=**new** User("rose",27);  User u3=**new** User("tom",29);  //①参:序列化类型的模式  //②参:序列化文件的路径  dfw.create(u1.getSchema(),**new** File("1.txt"));  //append方法将数据添加到序列化文件里  dfw.append(u1);  dfw.append(u2);  dfw.append(u3);  dfw.close();  } |

**反序列化代码：**

|  |
| --- |
| @Test  **public** **void** read() **throws** Exception{  DatumReader<User> dr=**new** SpecificDatumReader<>(User.**class**);  //①参:反序列化的文件路径  //②参:DatumReader 对象  DataFileReader<User> dfr=  **new** DataFileReader<>(**new** File("1.txt"), dr);  //DataFileReader本身就实现了迭代器接口，每次迭代返回一个User对象  **while**(dfr.hasNext()){  User u=dfr.next();  System.***out***.println(u);  }  } |

# 4.RPC框架

## 4.1.概念

在理解什么是RPC框架之前，首先要了解什么是RPC 。

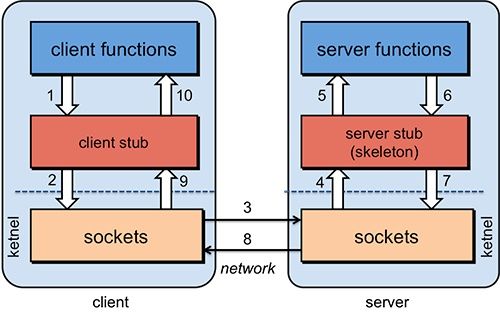
RPC的翻译是：Remote Procedure Call Protocol——远程过程调用协议。

**应用场景：**基于RPC，我们可以让一台计算机通过网络通信从远程计算机（另一台计算机）上请求服务，并得到远程计算机返回的结果。

所以，使用RPC的目的就是无法在一个进程内，甚至一个计算机内通过本地调用的方式完成的需求，比如不同的系统间的通讯，甚至不同的组织间的通讯。由于计算能力需要横向扩展，需要在多台机器组成的集群上部署应用，

**最常见应用于分布式集群环境中，集群里各节点（计算机）之间的网络通信。一是远程方法调用，二是做数据的传输。**

**一次RPC的过程**



两台服务器A，B，一个应用部署在A服务器上，A想要调用B服务器上应用提供的函数/方法，由于不在一个内存空间，不能直接调用，需要通过网络来表达调用的语义和传达调用的数据。

比如说，一个方法可能是这样定义的：

int add(int a,int b）

那么：

首先，要解决通讯的问题，主要是通过在客户端和服务器之间建立TCP连接，远程过程调用的所有交换的数据都在这个连接里传输。

第二，要解决寻址的问题，也就是说，A服务器上如何连接到B服务器（如主机或IP地址）以及特定的端口，方法的名称名称是什么，这样才能完成调用。

第三，当A服务器上的应用发起远程过程调用时，方法的参数需要通过底层的网络协议如TCP传递到B服务器，由于网络协议是基于二进制的，所以需要将数据进行序列化后发送给B服务器。

第四，B服务器收到请求后，需要对参数进行反序列化（序列化的逆操作），恢复数据。然后找到对应的方法进行本地调用，然后得到返回值。

第五，返回值还要发送回服务器A上的应用，也要经过序列化的方式发送，服务器A接到后，再反序列化，恢复为内存中的表达方式，交给A服务器上的应用。

**主流的RPC框架**

**Thrift**

facebook的开源RPC框架，现在贡献给apache。支持多语言之间的RPC通信，facebook当时在开发Thrift的时候，目的就是为了实现facebook系统内之间的各语言通信。2007年贡献给apache。

**Protocol Buffer**

是Google开源的RPC框架，可以跨平台使用，目前支持的语言：C++，java，python。（没有thrift多）。

**Apache Avro**

**RPC框架具有的特点：**

1.基于rpc协议

2.有自定义的一套序列化和反序列的机制  
3.客户端通过代理机制调用远程方法

4.服务端通过回调机制执行方法及返回结果

## 4.2.模式定义avdl文件

avdl文件用于avro生成协议方法的。

### 4.2.1.实现步骤：

1.在src\main\avro目录下新建一个后缀为avdl的文件，比如AddService.avdl文件

2.根据avro格式要求以及业务要求编辑这个文件

3.利用maven -run as -generate sources 生成协议方法类

**AddService.avdl代码：**

@namespace("rpc.service")//接口生成的包路径

protocol AddService{//协议+接口名

int add(int i,int y);//接口方法

}

**协议方法里想传递某个avsc对象的代码**

@namespace("rpc.service")

protocol AddService{

**import schema "User.avsc";**

int add(int i,int y);

void parseUser(**avro.domain.User user**);

}

**协议方法里想传递一个map，并且map里包含一个对象的代码：**

@namespace("rpc.service")

protocol AddService{

**import schema "User.avsc";**

int add(int i,int y);

void parseUser(avro.domain.User user);

**void parseUserMap(map<avro.domain.User> userMap);**

}

### 4.2.2.AVRO—实现RPC加法运算

**一、创建工程**

分别创建两个maven工程：avro-add-client和avro-add-server，创建方式与avro工程创建方式一致，拷贝avro中的pom.xml文件，修改jdk环境，并创建src/main/avro的源目录.

**二、编写存根并生成接口**

**在src/main/avro包下创建文件AddService.avdl，此文件为存根文件，按照存根文件的模板规范写：**

**AddService.avdl代码：**

|  |
| --- |
| @namespace("rpc.service") //接口生成的包路径  protocol AddService{ //协议+接口名  int add(int a,int b); //接口方法  } |

在项目上右键=》run as=>Maven generate source 如果控制台输出BUILD SUCCESS，则代表成功，刷新工程，会生成avdl文件对应的接口

**在服务端项目中重复刚才的动作。**

1. **avro-add-server工程实现接口并创建启动类**

在服务端实现接口并编写算法

|  |
| --- |
| **package** rpc.service;  **import** org.apache.avro.AvroRemoteException;  **public** **class** AddServiceImpl **implements** AddService{  @Override  **public** **int** add(**int** a, **int** b) **throws** AvroRemoteException {  **return** a+b;  }  } |

在服务端创建服务端的启动类：

|  |
| --- |
| **package** rpc.service;  **import** java.net.InetSocketAddress;  **import** org.apache.avro.ipc.NettyServer;  **import** org.apache.avro.ipc.specific.SpecificResponder;  **public** **class** Start {  **public** **static** **void** main(String[] args) {  NettyServer server=**new** NettyServer(  **new** SpecificResponder(AddService.**class**,**new** AddServiceImpl()),  **new** InetSocketAddress(8888));  System.***out***.println("服务端启动");  }  } |

1. **avro-add-client工程创建客户端的启动类**

|  |
| --- |
| **package** rpc.service;  **import** java.io.IOException;  **import** java.net.InetSocketAddress;  **import** org.apache.avro.ipc.NettyTransceiver;  **import** org.apache.avro.ipc.specific.SpecificRequestor;  **public** **class** Start {  **public** **static** **void** main(String[] args) **throws** IOException {  NettyTransceiver client=**new** NettyTransceiver(**new** InetSocketAddress("127.0.0.1",8888));  //调用存根中的方法，此方法会生成一个代理类  AddService proxy=SpecificRequestor.*getClient*(AddService.**class**, client);  **int** result=proxy.add(2,3);  System.***out***.println("客户端传递了2,3,调用服务端的addService方法，运算结果为："+result);  client.close();  }  } |

### 4.2.3.AVRO—实现RPC对象的传输

1. **创建类的模板文件**

可以直接将之前avro中的User.avsc拷贝到avro-add-client工程的src/main/avro下，并修改AddService.avdl为：

|  |
| --- |
| @namespace("rpc.service")  protocol AddService{  import schema "User.avsc";  example.avro.User getUser(example.avro.User user);  int add(int a,int b);  } |

在项目上右键=》run as=>Maven generate source 如果控制台输出BUILD SUCCESS，则代表成功，刷新工程

**在服务端项目中重复刚才的动作。**

1. **avro-add-server工程修改AddServiceImpl并启动**

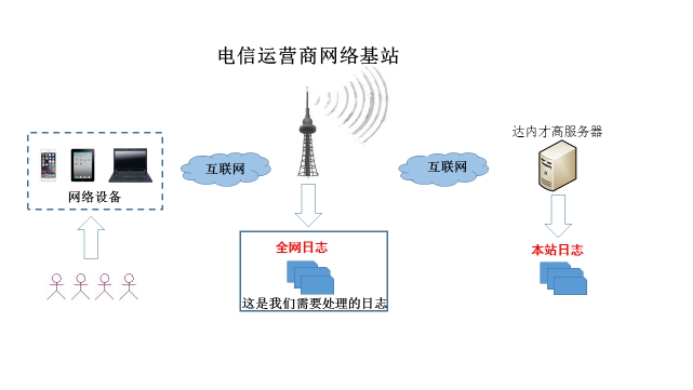
|  |
| --- |
| @Override  **public** User getUser(User user) **throws** AvroRemoteException {  User u = user;  **if**(u.getAge()!=**null**){  u.setAge(u.getAge()+1);  }  **return** u;  } |

1. **avro-add-client工程修改客户端的启动类**

|  |
| --- |
| **public** **static** **void** main(String[] args) **throws** IOException {  NettyTransceiver client=**new** NettyTransceiver(**new** InetSocketAddress("127.0.0.1",8888));  //调用存根中的方法，此方法会生成一个代理类  AddService proxy=SpecificRequestor.*getClient*(AddService.**class**, client);  User user = **new** User();  user.setName("tom张三");  user.setAge(18);  User u = proxy.getUser(user);  System.***out***.println("客户端收到数据："+u);  client.close();  } |

# 5.zebra项目

## 5.1.zebra项目介绍与分析



如图所示，电信运营商的用户通过连接到互联网中的各种网络设备访问一个网站时，其访问信息会通过基站在网络中传递，一个基站负责收集某一片小区用户的上网数据，这些收集的数据都以日志信息进行存储。所有的用户上网行为都会有所记录。比如用户通过3G/4G下载某个app应用，或者登陆、使用某一个App软件，以及通过App发送的数据都会记录。

这样一来，我们就可以根据日志文件，来分析和统计数据，比如可以统计：

1.App下载热度排行

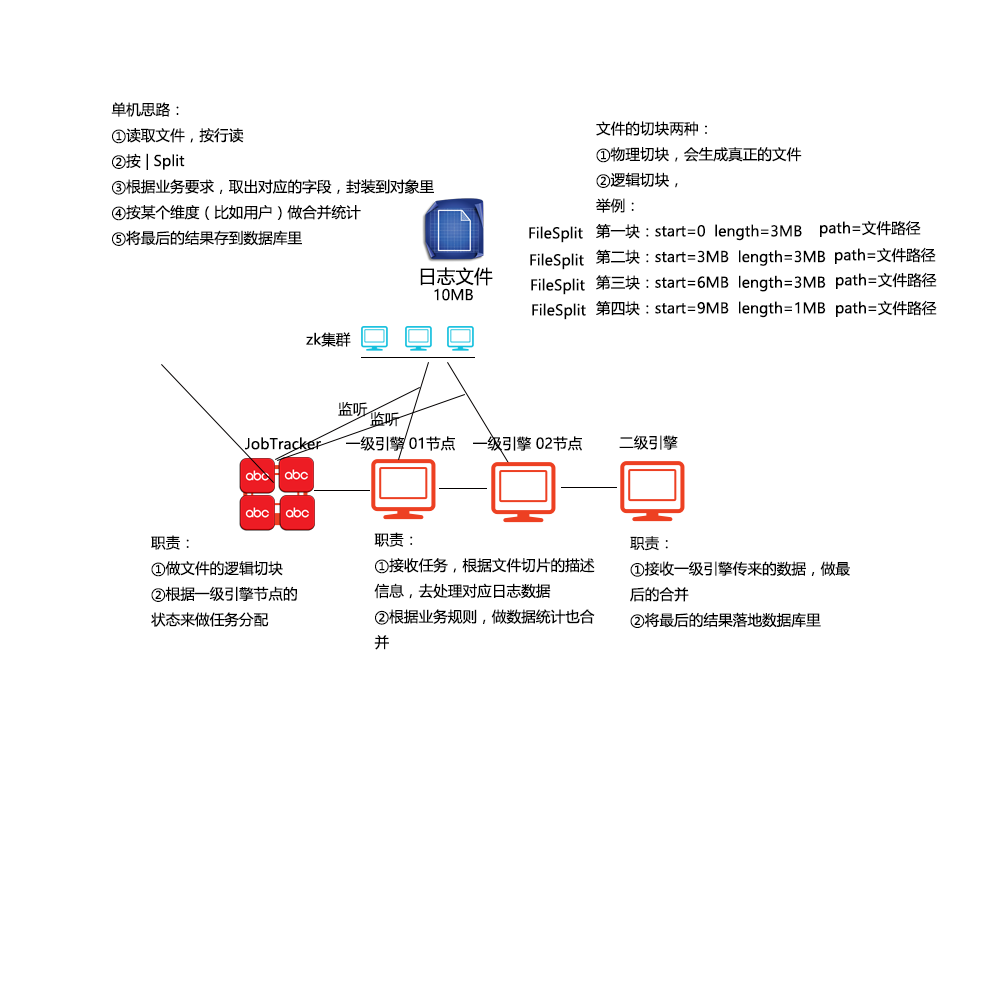
2.用户使用哪个App平均用时最长

3.用户使用哪个App耗费流量最多

…………

既可以根据数据，多维度来统计数据

这就是zebra项目的目的



## 5.2.zebra项目整体架构

**技术架构：**

1.maven

利用Maven工程来管理项目。主要是利用Maven来管理jar包，以及利用maven生成avro的rpc方法及序列化对象

2.avro

利用avro实现对象的序列化以及节点间的rpc通信

3.zookeeper

利用zookeeper达到分布式环境的协调服务

**工程架构：**

**1.zebra-contract 合同工程。此工程作用：**

①管理整个项目的pom.xml文件

②定义全局变量参数及工具类

③定义avro相关的avsc文件及avdl文件

**2.zebra-jobtracker 工程。此工程作用：**

①定时扫描指定目录下是否有待处理文件

②根据用户定义的参数，对文件进行逻辑切块。

③根据文件切块数量生成对应的任务数量。（一个文件块相当于一个任务）

④将任务分发给一级引擎节点（TaskTracker )去处理。并通过zookeeper监控TaskTracker的状态来决定任务分配

**3.zebra-engine1 工程。此工程作用：**

①接收jobtracker发来的任务，根据任务进行对文件的处理

②将文件数据进行清洗和整理。（根据zebra要求的业务逻辑进行数据整理）

③将处理完的数据发给二级引擎，二级引擎做最后的合并

④通过zookeeper，注册自身节点信息状态，便于集群其他机器监控继而做相关的业务逻辑处理

**4.zebra-engine2 工程。此工程作用：**

①接收（一个或多个）一级引擎发来的数据。

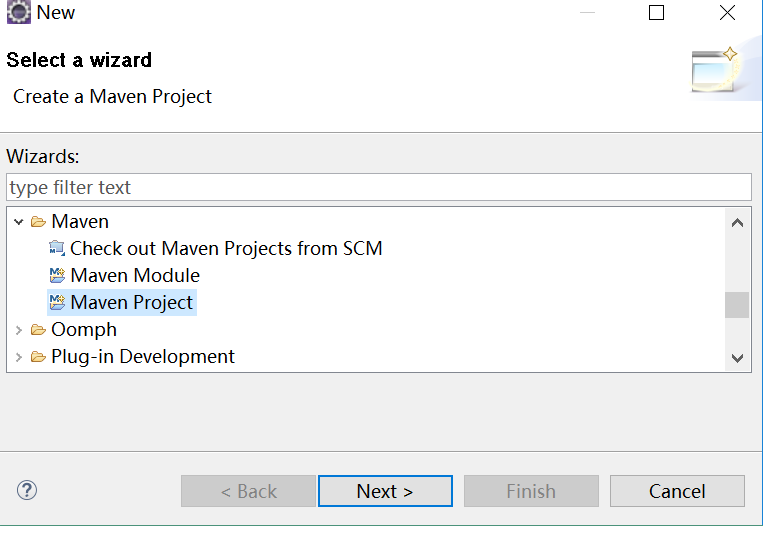
②对数据进行归并处理

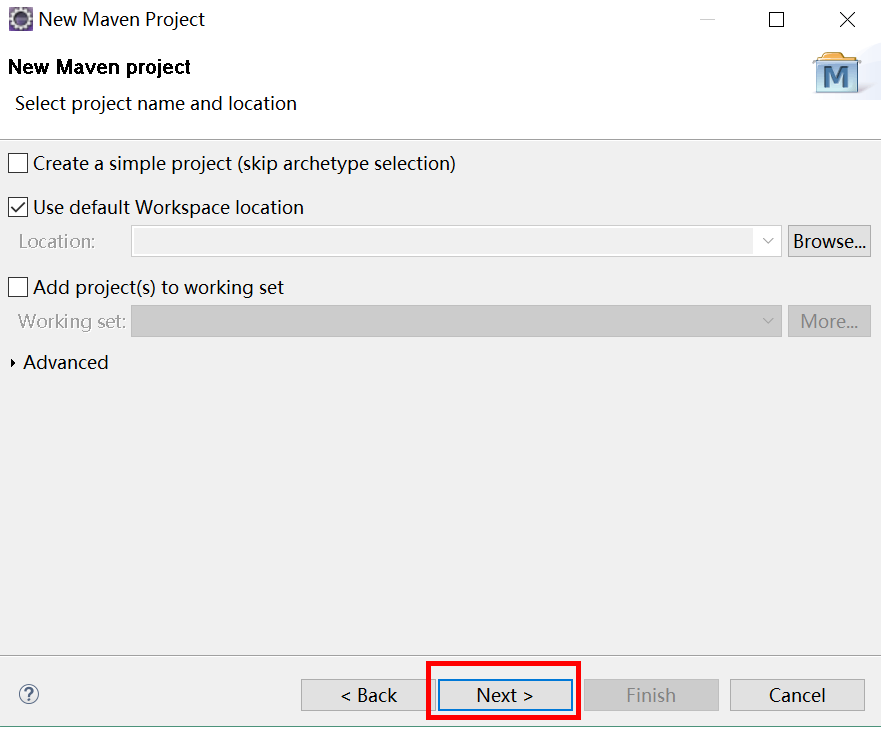
③将最后处理的数据结果落地（写文件或写数据库）

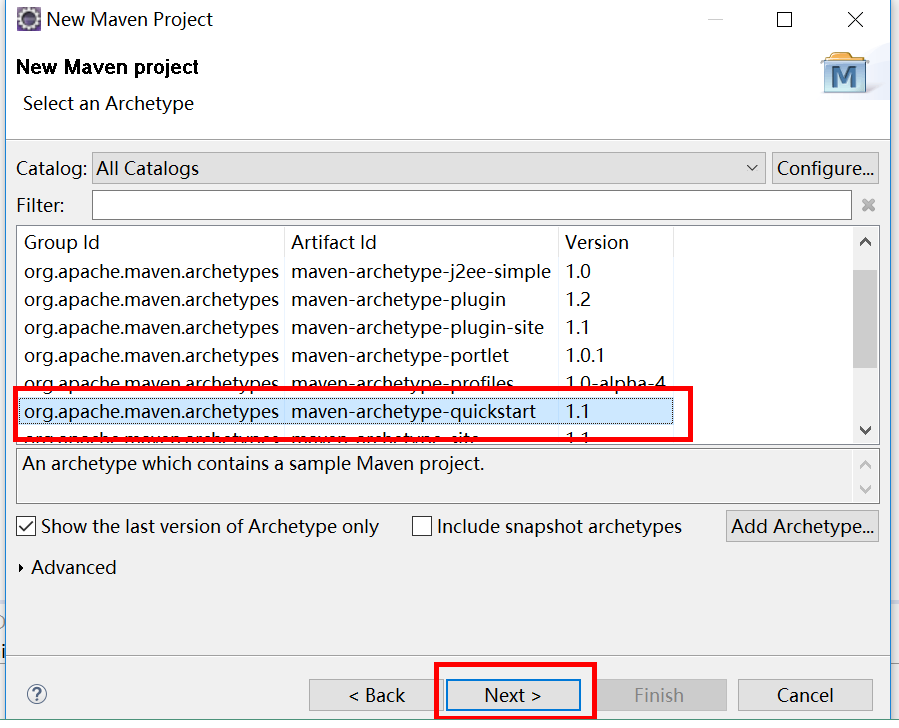
## 5.3.zebra项目业务实现

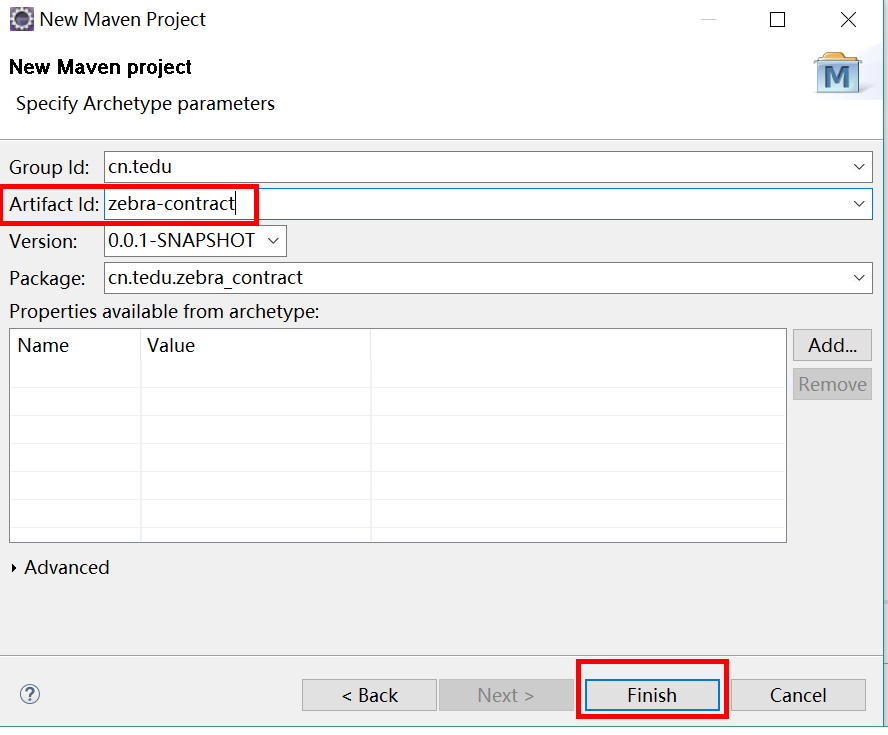
**1.创建父工程contract**

**创建maven项目：zebra-contract,注意环境，java build path和java compiler改为自己jdk版本**

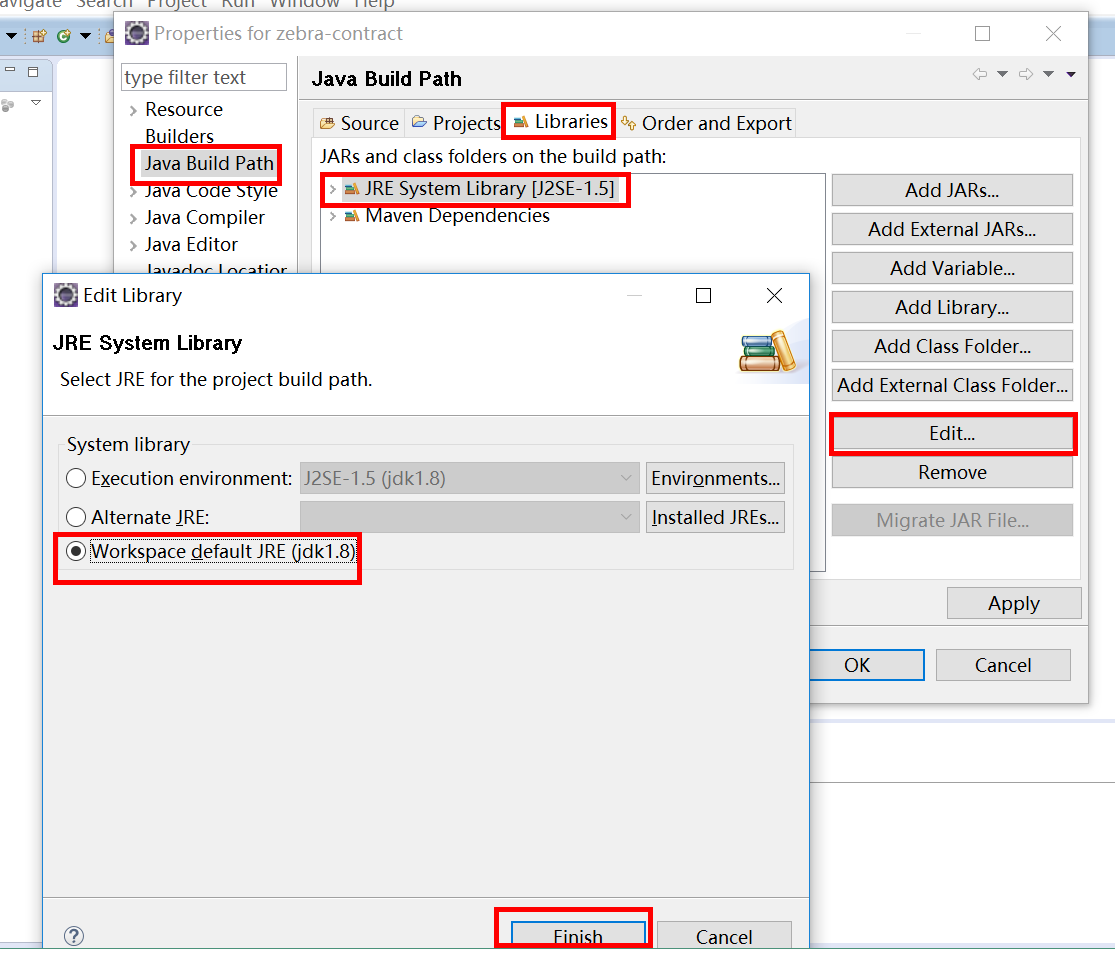




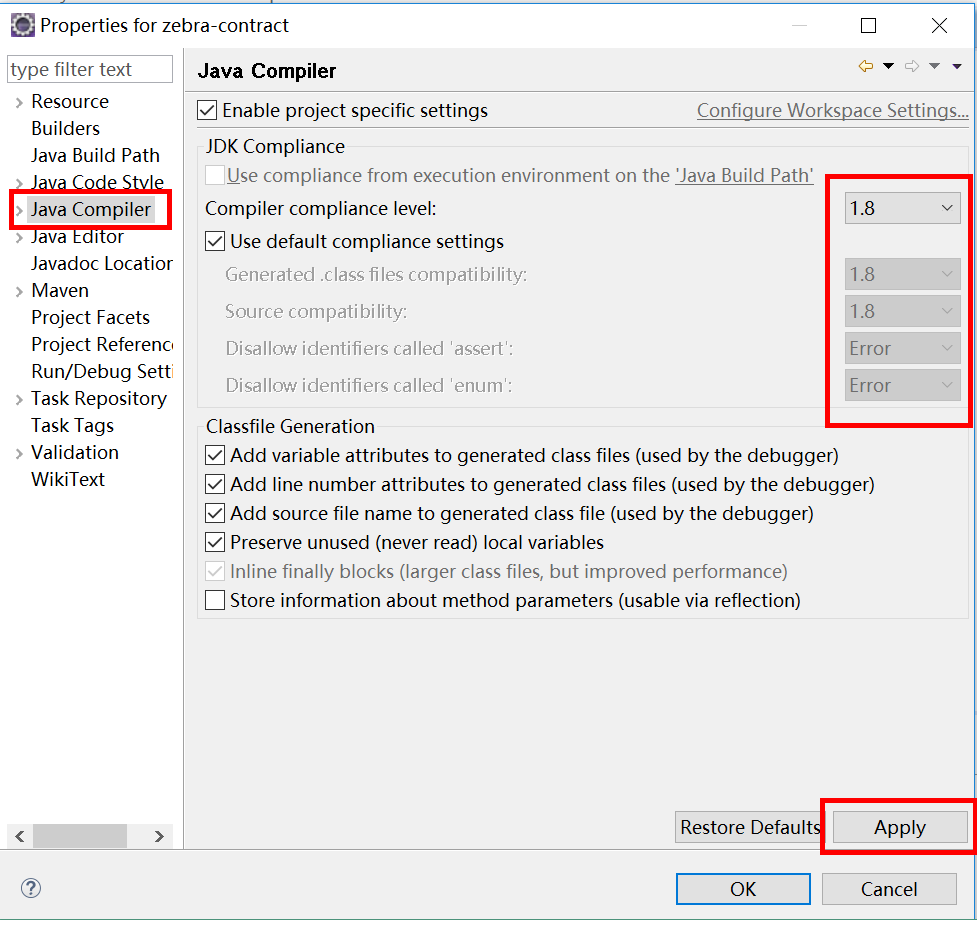




修改jdk为自己的jdk：



修改jdk版本信息



1. **引入pom文件**

**将工程pom.xml引入zebra-contract（在课前资料中）**

1. **创建模式文件**

**在工程下创建源目录：src\main\avro，并在此目录下分别创建如下模式文件：**

**FileSplit.avsc：用来生成文件切片对象**

|  |
| --- |
| {"namespace": "rpc.domain",  "type": "record",  "name": "FileSplit",  "fields": [  {"name": "path", "type": ["string", "null"]},  {"name": "start", "type": ["long", "null"]},  {"name": "length", "type": ["long", "null"]}  ]  } |

**HttpAppHost.avsc：用来生成业务字段对象**

|  |
| --- |
| {"namespace": "rpc.domain",  "type": "record",  "name": "HttpAppHost",  "fields": [  {"name": "reportTime", "type": ["string", "null"]},  {"name": "cellid", "type": ["string", "null"]},  {"name": "appType", "type": ["int", "null"]},  {"name": "appSubtype", "type": ["int", "null"]},  {"name": "userIP", "type": ["string", "null"]},  {"name": "userPort", "type": ["int", "null"]},  {"name": "appServerIP", "type": ["string", "null"]},  {"name": "appServerPort", "type": ["int", "null"]},  {"name": "host", "type": ["string", "null"]},  {"name": "attempts", "type": ["int", "null"]},  {"name": "accepts", "type": ["int", "null"]},  {"name": "trafficUL", "type": ["long", "null"]},  {"name": "trafficDL", "type": ["long", "null"]},  {"name": "retranUL", "type": ["long", "null"]},  {"name": "retranDL", "type": ["long", "null"]},  {"name": "failcount", "type": ["int", "null"]},  {"name": "transDelay", "type": ["long", "null"]}  ]  } |

**RpcFileSplit.avdl:rpc传输切片的存根**

|  |
| --- |
| @namespace("rpc.service")  protocol RpcFileSplit{  import schema "FileSplit.avsc";  void sendFileSplit(rpc.domain.FileSplit fileSplit);  } |

**RpcSendHttpAppHost.avdl:rpc传输对象和所有对象组成的map的存根**

|  |
| --- |
| @namespace("rpc.service")  protocol RpcSendHttpAppHost{  import schema "HttpAppHost.avsc";  void sendHttpAppHost(rpc.domain.HttpAppHost httpAppHost);  void sendMap(map<rpc.domain.HttpAppHost> hahMap);  } |

**创建完模式文件后，在项目上右键->run as->Maven generate source 如果控制台输出BUILD SUCCESS，则代表成功，刷新工程，会生成对应的类和接口**

1. **加入配置文件及工具类**

**在src/main/java目录下创建配置文件：env.properties，代码如下：**

|  |
| --- |
| zebra.dir=D:\\zebra\\data  zebra.scanninginterval=30000  zebra.blocksize=3000000  zebra.zk.serverip=192.168.234.158:2181,192.168.234.159:2181,192.168.234.160:2181  zebra.zk.sessiontimeout=30000  zebra.zk.jobtrackerpath=/jobtracker  zebra.zk.engine1path=/engine1  zebra.zk.engine2path=/engine2 |

zebra.dir=D:\\zebra\\data //日志文件所在目录，可以自定义

zebra.scanninginterval=30000 //扫描周期，按照每30秒扫描上述目录

zebra.blocksize=3000000 //切片大小，可以自己设置

zebra.zk.serverip //zookeeper集群ip，需要改为自己的ip

zebra.zk.sessiontimeout=30000 //会话超时时间，30秒

zebra.zk.jobtrackerpath=/jobtracker //jobtracker在zookeeper中注册节点的路径

zebra.zk.engine1path=/engine1 //一级引擎在zookeeper中注册节点的路径

zebra.zk.engine2path=/engine2 //二级引擎在zookeeper中注册节点的路径

**将日志文件拷贝到对应目录下（注意有两个文件）**

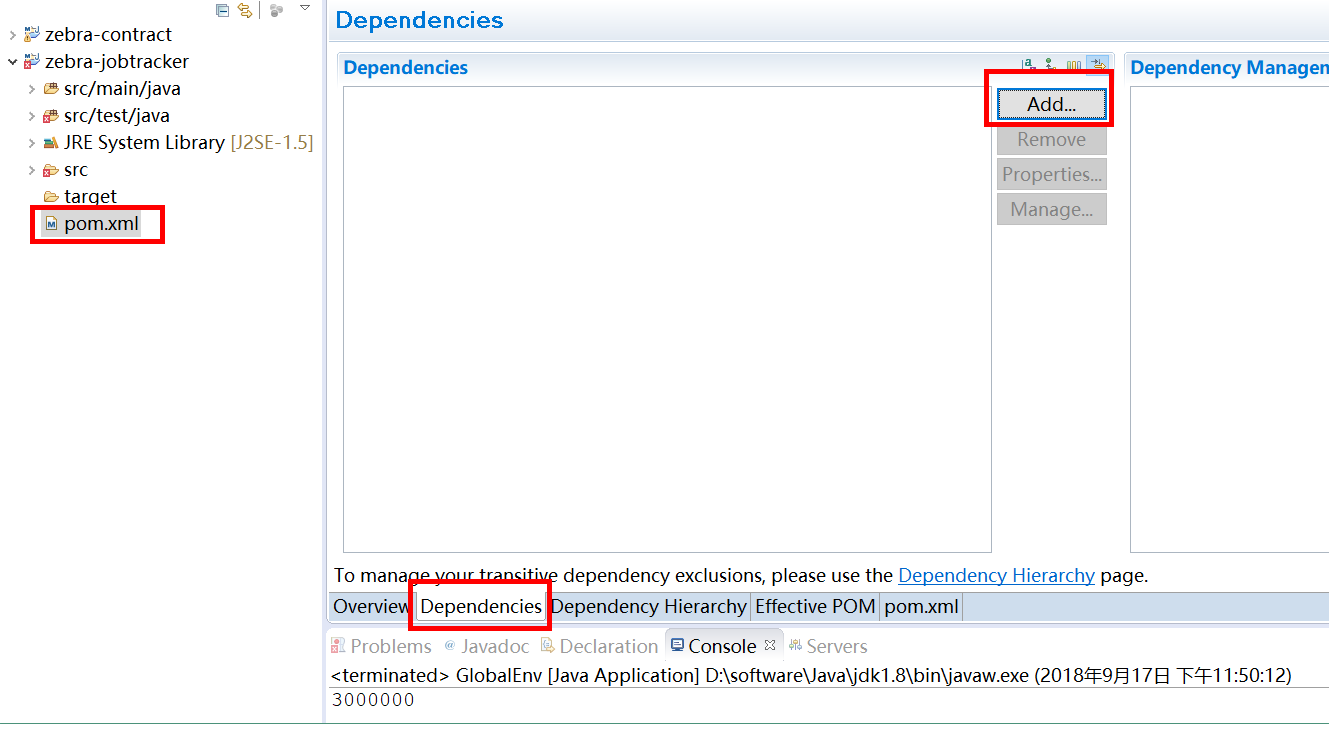
**在此目录下创建包：cn.tedu.zebra.commen，并在此包下创建工具类：GlobalEnv，用来读取配置文件信息，并保存在类的内部，同时定义两个队列，分别保存日志文件和文件切片对象，并提供get方法，同时创建获取zookeeper连接的方法，代码如下：**

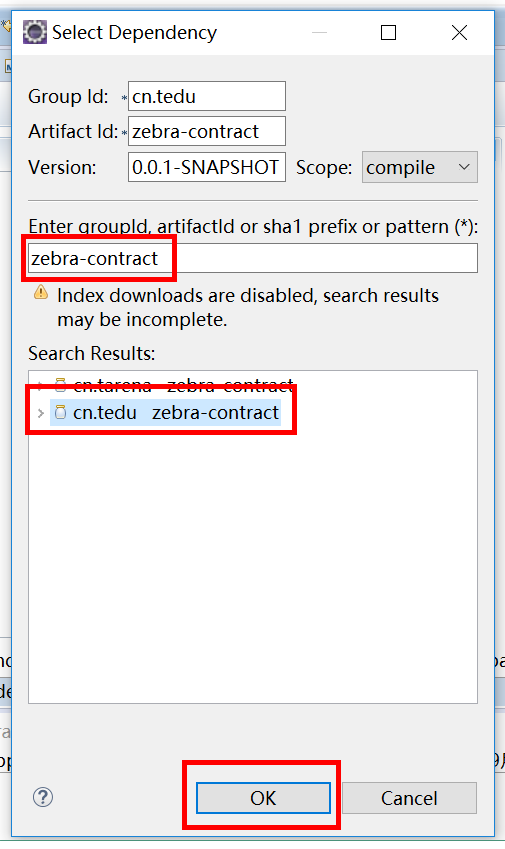
|  |
| --- |
| **package** cn.tedu.zebra.common;  **import** java.io.File;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.util.Properties;  **import** java.util.concurrent.BlockingQueue;  **import** java.util.concurrent.CountDownLatch;  **import** java.util.concurrent.LinkedBlockingQueue;  **import** org.apache.zookeeper.WatchedEvent;  **import** org.apache.zookeeper.Watcher;  **import** org.apache.zookeeper.Watcher.Event.KeeperState;  **import** org.apache.zookeeper.ZooKeeper;  **import** rpc.domain.FileSplit;  /\*\*  \* 这个类是用定义全局变量，以及加载属性文件和读取属性的  \*/  **public** **class** GlobalEnv {  //别忘了，在静态代码里初始化  **static**{  *initParam*();  }  //文件目录路劲  **private** **static** String *dir*;  //日志文件队列，因为每30秒扫描一次目录，如果发现有新文件，则放入此队列  **private** **static** BlockingQueue<File> *fileQueue*=**new** LinkedBlockingQueue<>();  //文件切片队列，将文件切片对象存入此队列  **private** **static** BlockingQueue<FileSplit> *splitQueue*=**new** LinkedBlockingQueue<>();  **private** **static** **long** *scannningInterval*;  **private** **static** **long** *blocksize*;  **private** **static** String *zkServerip*;  **private** **static** **int** *sessiontimeout*;  **private** **static** String *jobtrackerpath*;  **private** **static** String *engine1path*;  **private** **static** String *engine2path*;  /\*  \* 这个方法是用于分配zk连接对象的方法  \*/  **public** **static** ZooKeeper connectZkServer(){  **try** {  **final** CountDownLatch cdl=**new** CountDownLatch(1);  ZooKeeper zk=**new** ZooKeeper(*zkServerip*, *sessiontimeout*,**new** Watcher(){  @Override  **public** **void** process(WatchedEvent event) {  **if**(event.getState()==KeeperState.***SyncConnected***){  System.***out***.println("连接zk服务成功！");  cdl.countDown();  }  }  });  cdl.await();  **return** zk;  } **catch** (Exception e) {  e.printStackTrace();  **return** **null**;  }  }  /\*  \* 这个方法用来加载和属性文件的  \*/  **public** **static** **void** initParam(){  **try** {  Properties pro=**new** Properties();  InputStream in=GlobalEnv.**class**.getResourceAsStream("/env.properties");  pro.load(in);  in.close();  **if**(pro.containsKey("zebra.dir")){  *dir*=pro.getProperty("zebra.dir");  }  **if**(pro.containsKey("zebra.scanninginterval")){  *scannningInterval*=Long.*parseLong*(pro.getProperty("zebra.scanninginterval","30000"));  }  **if**(pro.containsKey("zebra.blocksize")){  *blocksize*=Long.*parseLong*(pro.getProperty("zebra.blocksize"));  }  **if**(pro.containsKey("zebra.zk.serverip")){  *zkServerip*=pro.getProperty("zebra.zk.serverip");  }  **if**(pro.containsKey("zebra.zk.sessiontimeout")){  *sessiontimeout*=Integer.*parseInt*(pro.getProperty("zebra.zk.sessiontimeout"));  }  **if**(pro.containsKey("zebra.zk.jobtrackerpath")){  *jobtrackerpath*=pro.getProperty("zebra.zk.jobtrackerpath");  }  **if**(pro.containsKey("zebra.zk.engine1path")){  *engine1path*=pro.getProperty("zebra.zk.engine1path");  }  **if**(pro.containsKey("zebra.zk.engine2path")){  *engine2path*=pro.getProperty("zebra.zk.engine2path");  }  } **catch** (Exception e) {  e.printStackTrace();  }  }  **public** **static** String getDir() {  **return** *dir*;  }  **public** **static** BlockingQueue<File> getFileQueue() {  **return** *fileQueue*;  }  **public** **static** **long** getScannningInterval() {  **return** *scannningInterval*;  }  **public** **static** **long** getBlocksize() {  **return** *blocksize*;  }  **public** **static** BlockingQueue<FileSplit> getSplitQueue() {  **return** *splitQueue*;  }  **public** **static** String getJobtrackerpath() {  **return** *jobtrackerpath*;  }  **public** **static** String getEngine1path() {  **return** *engine1path*;  }  **public** **static** String getEngine2path() {  **return** *engine2path*;  }  } |

**5.创建jobtracker工程**

**创建maven项目：zebra-jobtracker,注意环境，java build path和java compiler改为自己jdk版本**

**因为zebra-jobtracker相当于继承zebra-contract，所以需要对pom.xml文件进行依赖：**





最后需要保存

**6.创建一级引擎并添加配置文件和工具类**

**创建maven项目：zebra-engine1-01,注意环境，java build path和java compiler改为自己jdk版本**

**同样zebra-engine1继承zebra-contract，所以需要对pom.xml文件进行依赖，同上**

**一级引擎需要一些自己的配置，在src/main/java目录下创建配置文件：ownenv.properties**

|  |
| --- |
| zebra.rpcport=9991  zebra.zk.znodepath=/node01 |

分别为一级引擎的rpc服务端端口号和在zookeeper中的节点路径

**在此目录下创建包：cn.tedu.zebra.commen，并在此包下创建工具类：OwnEnv，用来读取配置文件信息，并保存在类的内部，同时定义两个队列，分别保存jobtracher发送过来的文件切片对象和处理完切片后生成的map对象，并提供get方法**

|  |
| --- |
| **package** cn.tedu.zebra.common;  **import** java.io.IOException;  **import** java.io.InputStream;  **import** java.util.Map;  **import** java.util.Properties;  **import** java.util.concurrent.BlockingQueue;  **import** java.util.concurrent.LinkedBlockingQueue;  **import** rpc.domain.FileSplit;  **import** rpc.domain.HttpAppHost;  **public** **class** OwnEnv {  **static**{  *initParam*();  }  //用来存储从jobtracher发送过来的文件切片对象  **private** **static** BlockingQueue<FileSplit> *spiltQueue*=**new** LinkedBlockingQueue<>();  //将每个文件切片中处理完的数组组成的map放入此队列，到时候发送给引擎2  **private** **static** BlockingQueue<Map<CharSequence,HttpAppHost>> *mapQueue*=**new** LinkedBlockingQueue<Map<CharSequence,HttpAppHost>>();  **private** **static** **int** *rpcport*;  **private** **static** String *znodepath*;  **public** **static** **void** initParam(){  **try** {  Properties pro=**new** Properties();  InputStream in=OwnEnv.**class**.getResourceAsStream("/ownenv.properties");  pro.load(in);  in.close();  **if**(pro.containsKey("zebra.rpcport")){  *rpcport*=Integer.*parseInt*(pro.getProperty("zebra.rpcport"));  }  **if**(pro.containsKey("zebra.zk.znodepath")){  *znodepath*=pro.getProperty("zebra.zk.znodepath");  }  } **catch** (IOException e) {  e.printStackTrace();  }  }  **public** **static** **int** getRpcport() {  **return** *rpcport*;  }  **public** **static** String getZnodepath() {  **return** *znodepath*;  }  **public** **static** BlockingQueue<FileSplit> getSpiltQueue() {  **return** *spiltQueue*;  }  **public** **static** BlockingQueue<Map<CharSequence, HttpAppHost>> getMapQueue() {  **return** *mapQueue*;  }  } |

**7.创建二级引擎**

**创建maven项目：zebra-engine2,注意环境，java build path和java compiler改为自己jdk版本**

**同样zebra-engine2继承zebra-contract，所以需要对pom.xml文件进行依赖，同上**

1. **定期扫描指定目录日志文件**

**zebra-jobtracker工程：创建包cn.tedu.file，包下创建类FileHandle，此类是一个线程类，定期扫描指定目录下是否有待处理日志文件，如果有，将此日志文件放到队列里，供后续处理**

|  |
| --- |
| **package** cn.tedu.file;  **import** java.io.File;  **import** cn.tedu.zebra.common.GlobalEnv;  /\*\*  \* 是一个线程类，定期扫描指定目录下是否有待处理日志文件  \* 如果有，将此日志文件放到队列里，供后续处理  \*/  **public** **class** FileHandle **implements** Runnable{  @Override  **public** **void** run() {  **try** {  **while**(**true**){  //获取日志文件所在目录  File dir=**new** File(GlobalEnv.*getDir*());  //获取目录下所有文件  File[] files=dir.listFiles();  **for**(File file:files){  //引入标识文件，根据标识文件找还未被处理的日志文件，  //当日志文件加到队列之后，删掉对应的标识文件，这样可以避免重复处理  **if**(file.getName().endsWith(".ctr")){  String csvName=file.getName().split(".ctr")[0]+".csv";  //找到了待处理的日志文件  File csvFile=**new** File(dir,csvName);  GlobalEnv.*getFileQueue*().add(csvFile);  //删除标识文件  file.delete();  }  }  //设定扫描周期  Thread.*sleep*(GlobalEnv.*getScannningInterval*());  }  } **catch** (Exception e) {  e.printStackTrace();  }  }  } |

1. **文件进行逻辑切块**

**zebra-jobtracker工程：在cn.tedu.file下创建类FileToBlock，此类是一个线程类，用于从日志队列里取出文件进行逻辑切块，并将切片对象存入切片队列中。**

**可以打桩测试代码是否正确**

|  |
| --- |
| **package** cn.tedu.file;  **import** java.io.File;  **import** cn.tedu.zebra.common.GlobalEnv;  **import** rpc.domain.FileSplit;  /\*\*  \* 是一个线程类，用于从日志队列里取出文件进行 逻辑切块  \* 文件总大小/用户指定的大小，要考虑是否整除的情况  \* 做文件切片（FileSplit)的封装，存的是任务的描述信息:  \* 文件路径：path 处理的起始长度：start 处理长度：length  \* 第一个切片：start 0 length 30000  \* 第二个切片：start 30000 length 30000  \* 第三个切片：start 60000 length 30000  \* 第四个切片：start 90000 length 10mb-9mb  \*/  **public** **class** FileToBlock **implements** Runnable{  @Override  **public** **void** run() {  **try** {  **while**(**true**){  File file=GlobalEnv.*getFileQueue*().take();  **long** length=file.length();  **long** num=length%GlobalEnv.*getBlocksize*()==0?length/GlobalEnv.*getBlocksize*():length/GlobalEnv.*getBlocksize*()+1;  **for**(**int** i=0;i<num;i++){  FileSplit split=**new** FileSplit();  split.setPath(file.getPath());  split.setStart(i\*GlobalEnv.*getBlocksize*());  **if**(i==num-1){  split.setLength(file.length()-split.getStart());  }**else**{  split.setLength(GlobalEnv.*getBlocksize*());  }  //打桩输出切片信息  System.***out***.println(split);  GlobalEnv.*getSplitQueue*().add(split);  }  }  } **catch** (Exception e) {  e.printStackTrace();  }  }  } |

**编写测设类进行测试，创建包：cn.tedu.common，创建类Start，代码如下：**

|  |
| --- |
| **package** cn.tedu.common;  **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;  **import** cn.tedu.file.FileHandle;  **import** cn.tedu.file.FileToBlock;  **public** **class** Start {    **public** **static** **void** main(String[] args) {  System.***out***.println("jobtracker启动");  //创建线程池启动多个线程，newCachedThreadPool：大池子小队列，适用于高并发短时任务  ExecutorService es=Executors.*newCachedThreadPool*();  es.submit(**new** FileHandle());  es.submit(**new** FileToBlock());  }  } |

**运行此类，进行测试，如果控制台输出对应切片信息，说明前面代码正确**

1. **一级引擎注册zookeeper**

**jobtracker已经对文件进行逻辑切块，接下来要通过rpc传输给一级引擎，但是如何获取一级引擎的ip和端口号（在一级引擎自己的配置文件中）？**

**可以通过zookeeper，一级引擎在zookeeper的/engine1下注册自己的节点/node，并在此节点写入数据为自己的ip和端口号jobtracher通过获取/engine1节点，然后获取其子节点和节点上的数据。**

**zebra-engine1-01工程在zookeeper上注册自己的节点，将本机的ip和rpc服务端端口号注册：创建包cn.tedu.zk，创建线程类：ZkConnectRunner，代码如下**

|  |
| --- |
| **package** cn.tedu.zk;  **import** java.net.InetAddress;  **import** org.apache.zookeeper.CreateMode;  **import** org.apache.zookeeper.ZooDefs.Ids;  **import** org.apache.zookeeper.ZooKeeper;  **import** cn.tedu.zebra.common.GlobalEnv;  **import** cn.tedu.zebra.common.OwnEnv;  /\*\*  \* 用于一级引擎注册自己节点信息，将本机的ip和rpc服务端端口号注册  \*/  **public** **class** ZkConnectRunner **implements** Runnable{  **private** **static** ZooKeeper *zk*;  @Override  **public** **void** run() {  **try** {  *zk*=GlobalEnv.*connectZkServer*();  // 10.8.42.106/9991  String data=InetAddress.*getLocalHost*().getHostAddress()+"/"+OwnEnv.*getRpcport*();  //判断/engine1节点是否存在，如果不存在，则创建此节点  **if**(*zk*.exists(GlobalEnv.*getEngine1path*(), **null**) == **null**){  *zk*.create(GlobalEnv.*getEngine1path*(), "".getBytes(), Ids.***OPEN\_ACL\_UNSAFE***, CreateMode.***PERSISTENT***);  }  // /engine1/node01  *zk*.create(GlobalEnv.*getEngine1path*()+OwnEnv.*getZnodepath*(),  data.getBytes(),Ids.***OPEN\_ACL\_UNSAFE***, CreateMode.***EPHEMERAL***);  } **catch** (Exception e) {  e.printStackTrace();  }  }  } |

**在cn.tedu.zebra.commen包下创建线启动类进行测试**

|  |
| --- |
| **package** cn.tedu.zebra.common;  **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;  **import** cn.tedu.zk.ZkConnectRunner;  **public** **class** Start {  **public** **static** **void** main(String[] args){  System.***out***.println("一级引擎01节点启动");  ExecutorService es=Executors.*newCachedThreadPool*();  es.submit(**new** ZkConnectRunner());  }  } |

**运行此类，如果zookeeper中出现对应节点和数据，证明注册成功**

1. **建立一级引擎的rpc服务端**

**一级引擎已经将ip和端口号注册到zookeeper上，接着开发一级引擎的rpc服务端，分别创建rpc存根RpcFileSplit的实现类RpcFileSplitImpl和用于提供rpc服务的线程类RpcServer，并最后将线程加入到启动类的线程池**

**zebra-engine1-01工程下创建包cn.tedu.rpc，创建实现类RpcFileSplitImpl，代码如下：**

|  |
| --- |
| **package** cn.tedu.rpc;  **import** org.apache.avro.AvroRemoteException;  **import** rpc.domain.FileSplit;  **import** rpc.service.RpcFileSplit;  **public** **class** RpcFileSplitImpl **implements** RpcFileSplit{  @Override  **public** Void sendFileSplit(FileSplit fileSplit) **throws** AvroRemoteException {  //暂时先不处理，先打印出来，进行测试  System.***out***.println("一级引擎收到切片："+fileSplit);  **return** **null**;  }  } |

**同时在此包下创建提供rpc服务的线程类RpcServer，代码如下：**

|  |
| --- |
| **package** cn.tedu.rpc;  **import** java.net.InetSocketAddress;  **import** org.apache.avro.ipc.NettyServer;  **import** org.apache.avro.ipc.specific.SpecificResponder;  **import** cn.tedu.zebra.common.OwnEnv;  **import** rpc.service.RpcFileSplit;  /\*\*  \* 是一个线程类，用于提供Rpc服务  \*/  **public** **class** RpcServer **implements** Runnable{  @Override  **public** **void** run() {  **try** {  NettyServer server=**new** NettyServer(**new** SpecificResponder(RpcFileSplit.**class**,**new** RpcFileSplitImpl()), **new** InetSocketAddress(OwnEnv.*getRpcport*()));  } **catch** (Exception e) {  e.printStackTrace();  }  }  } |

**在包cn.tedu.zebra.common包下的Start类下将刚才的线程加入线程池，代码如下：**

|  |
| --- |
| **public** **static** **void** main(String[] args){  System.***out***.println("一级引擎01节点启动");  ExecutorService es=Executors.*newCachedThreadPool*();  es.submit(**new** ZkConnectRunner());  **es.submit(new RpcServer());**  } |

1. **jobtracker创建rpc客户端与一级引擎连接**

**一级引擎服务端已经建好，jobtracker需要通过rpc连通一级引擎，但是在连通前需要先获取一级引擎的ip和rpc服务端端口号，通过zookeeper获取，可以创建线程类ZkConnectRunner来获取一级引擎的数据，并传给rpc客户端RpcClient的线程类，将切片数据发送给一级引擎，并将ZkConnectRunner加入到线程池进行启动，测试一级引擎是否能收到数据。**

**zebra-jobtracker工程下创建包cn.tedu.zk，包下创建类：ZkConnectRunner，代码如下：**

|  |
| --- |
| **package** cn.tedu.zk;  **import** java.util.List;  **import** org.apache.zookeeper.ZooKeeper;  **import** cn.tedu.rpc.RpcClient;  **import** cn.tedu.zebra.common.GlobalEnv;  /\*\*  \* 用于通过zk获取一级引擎节点信息  \*/  **public** **class** ZkConnectRunner **implements** Runnable{  **private** ZooKeeper zk;  @Override  **public** **void** run() {  **try** {  zk=GlobalEnv.*connectZkServer*();    List<String> paths=zk.getChildren(GlobalEnv.*getEngine1path*(),**null**);  // node01  **for**(String path:paths){  //每获取一个子节点，则启动一个客户端与服务端相连，并将path和zk对象传给客户端,zk将来可以监听客户端是否繁忙来发送数据  **new** Thread(**new** RpcClient(path,zk)).start();  }  } **catch** (Exception e) {  e.printStackTrace();  }  }  } |

**创建包cn.tedu.rpc，在包下创建rpc客户端类RpcClient，代码如下：**

|  |
| --- |
| **package** cn.tedu.rpc;  **import** java.net.InetSocketAddress;  **import** org.apache.avro.ipc.NettyTransceiver;  **import** org.apache.avro.ipc.specific.SpecificRequestor;  **import** org.apache.zookeeper.ZooKeeper;  **import** cn.tedu.zebra.common.GlobalEnv;  **import** rpc.service.RpcFileSplit;  /\*\*  \* 用于向服务端发送FileSpilt  \*/  **public** **class** RpcClient **implements** Runnable{  **private** String path;  **private** ZooKeeper zk;  **public** RpcClient(String path, ZooKeeper zk) {  **this**.path=path;  **this**.zk=zk;  }  @Override  **public** **void** run() {  **try** {  // /engine1 / node01 获取数据  **byte**[] data=zk.getData(GlobalEnv.*getEngine1path*()+"/"+path,**null**,**null**);  String ip=**new** String(data).split("/")[0];  **int** port=Integer.*parseInt*(**new** String(data).split("/")[1]);  NettyTransceiver client=**new** NettyTransceiver(**new** InetSocketAddress(ip, port));  **final** RpcFileSplit proxy=SpecificRequestor.*getClient*(RpcFileSplit.**class**,client);  //从队列中获取切片对象，并通过rpc发送，take()方法会产生阻塞  proxy.sendFileSplit(GlobalEnv.*getSplitQueue*().take());  } **catch** (Exception e) {  e.printStackTrace();  }  }  } |

**将ZkConnectRunner加入common包下的Start线程池：**

|  |
| --- |
| **public** **static** **void** main(String[] args) {  System.***out***.println("jobtracker启动");  //创建线程池启动多个线程，newCachedThreadPool：大池子小队列，适用于高并发短时任务  ExecutorService es=Executors.*newCachedThreadPool*();  es.submit(**new** FileHandle());  es.submit(**new** FileToBlock());  es.submit(**new** ZkConnectRunner());  } |

**进行测试：首先检查日志文件的标识文件是否存在，然后先启动一级引擎，然后启动二级引擎，看控制台是否能打印出文件切片，如果正确打印，则联通成功**

1. **一级引擎对切片位置追溯，并进行字段封装与合并**

**一级引擎收到切片后，将切片存入队列中，创建切片处理类MapperRunner，获取切片，并进行位置追溯，例如开始和结束的位置不一定是一行的开始和结束，统一向前追溯，利用通道中可以通过指针获取数据来判断。**

**zebra-engine1-01工程rpc包中的RpcFileSplitImpl修改为将收到的切片存入队列，修改代码如下：**

|  |
| --- |
| **public** Void sendFileSplit(FileSplit fileSplit) **throws** AvroRemoteException {  **OwnEnv.*getSpiltQueue*().add(fileSplit);**  **return** **null**;  } |

**创建包cn.tedu.mapper，包下创建线程类MapperRunner，进行位置的追溯与字段的处理与合并，并将map放入队列中以便交个二级引擎合并，代码如下：**

|  |
| --- |
| **package** cn.tedu.mapper;  **import** java.io.BufferedReader;  **import** java.io.ByteArrayInputStream;  **import** java.io.File;  **import** java.io.FileInputStream;  **import** java.io.InputStreamReader;  **import** java.nio.ByteBuffer;  **import** java.nio.channels.FileChannel;  **import** java.util.Arrays;  **import** java.util.HashMap;  **import** java.util.List;  **import** java.util.Map;  **import** cn.tedu.zebra.common.OwnEnv;  **import** rpc.domain.FileSplit;  **import** rpc.domain.HttpAppHost;  /\*\*  \* 是一个线程类，从队列里取出切片，根据任务描述信息，处理对应的数据  \* 根据zebra业务规则做字段封装，最后按照用户合并  \*/  **public** **class** MapperRunner **implements** Runnable{  @Override  **public** **void** run() {  **try** {  **while**(**true**){  //后面用于封装日志对象  Map<CharSequence,HttpAppHost> map=**new** HashMap<>();  FileSplit split=OwnEnv.*getSpiltQueue*().take();  String path=split.getPath().toString();  **long** start=split.getStart();  **long** end=start+split.getLength();  File file=**new** File(path);    //如果要一行一行读，为了保证是完整一行，要做文件的位置追溯  FileChannel fc=**new** FileInputStream(**new** File(path)).getChannel();    //处理start位置  **if**(start==0){    }**else**{  **long** headposition=start;  **while**(**true**){  fc.position(headposition);  ByteBuffer buffer=ByteBuffer.*allocate*(1);  fc.read(buffer);  **if**(**new** String(buffer.array()).equals("\n")){  start=headposition+1;  **break**;  }**else**{  headposition=headposition-1;  }  }  }    //处理end位置  **if**(end==file.length()){    }**else**{  **long** tailposition=end;  **while**(**true**){  fc.position(tailposition);  ByteBuffer buffer=ByteBuffer.*allocate*(1);  fc.read(buffer);  **if**(**new** String(buffer.array()).equals("\n")){  end=tailposition;  **break**;  }**else**{  tailposition=tailposition-1;  }  }    }  //代码到这，start和end的位置都处理完毕    //创建缓冲区存储切片数据  ByteBuffer buffer=ByteBuffer.*allocate*((**int**) (end-start));  fc.position(start);  fc.read(buffer);  //可以通过BufferedReader一行一行读取buffer中的数据  BufferedReader br=**new** BufferedReader(**new** InputStreamReader(**new** ByteArrayInputStream(buffer.array())));    //一行一行处理数据  String line=**null**;  **while**((line=br.readLine())!=**null**){  String[] data=line.split("\\|");  HttpAppHost hah=**new** HttpAppHost();  //封装日志生成的时间  hah.setReportTime(path.split("\_")[1]);  //应用大类  hah.setAppType(Integer.*parseInt*(data[22]));  //应用子类  hah.setAppSubtype(Integer.*parseInt*(data[23]));  //用户ip  hah.setUserIP(data[26]);  //用户port  hah.setUserPort(Integer.*parseInt*(data[28]));  //访问的服务ip  hah.setAppServerIP(data[30]);  //访问的服务port  hah.setAppServerPort(Integer.*parseInt*(data[32]));  //域名  hah.setHost(data[58]);  //上网小区的id  **if**(data[16].equals("")){  hah.setCellid("000000000");  }**else**{  hah.setCellid(data[16]);  }  **int** appTypeCode=Integer.*parseInt*(data[18]);  String transStatus=data[54];    //业务逻辑处理  **if**(appTypeCode==103){  hah.setAttempts(1);  }  String[] range = "10,11,12,13,14,15,32,33,34,35,36,37,38,48,49,50,51,52,53,54,55,199,200,201,202,203,204,205,206,302,304,306".split("\\,");  List<String> list = Arrays.*asList*(range);  **if**(appTypeCode==103&&list.contains(data[54])&&data[67].equals("")){  hah.setAccepts(1);  }**else**{  hah.setAccepts(0);  }  **if**(appTypeCode == 103){  hah.setTrafficUL(Long.*parseLong*(data[33]));  hah.setTrafficDL(Long.*parseLong*(data[34]));  hah.setRetranUL(Long.*parseLong*(data[39]));  hah.setRetranDL(Long.*parseLong*(data[40]));  }  **if**(appTypeCode==103&&data[54].equals("1")&&data[67].equals("0")){  hah.setFailcount(1);  }**else**{  hah.setFailcount(0);  }  **if**(appTypeCode==103){  hah.setTransDelay(Long.*parseLong*(data[20]) -Long.*parseLong*(data[19]));  }  CharSequence key=hah.getReportTime() + "|" + hah.getAppType() + "|" + hah.getAppSubtype() + "|" + hah.getUserIP() + "|" + hah.getUserPort() + "|" + hah.getAppServerIP() + "|" + hah.getAppServerPort() +"|" + hah.getHost() + "|" + hah.getCellid();  //将对象存入map中，如果map中已经存入此对象，则进行累加，否则直接存入  **if**(map.containsKey(key)){  HttpAppHost mapHah=map.get(key);  mapHah.setAccepts(mapHah.getAccepts()+hah.getAccepts());  mapHah.setAttempts(mapHah.getAttempts()+hah.getAttempts());  mapHah.setTrafficUL(mapHah.getTrafficUL()+hah.getTrafficUL());  mapHah.setTrafficDL(mapHah.getTrafficDL()+hah.getTrafficDL());  mapHah.setRetranUL(mapHah.getRetranUL()+hah.getRetranUL());  mapHah.setRetranDL(mapHah.getRetranDL()+hah.getRetranDL());  mapHah.setFailcount(mapHah.getFailcount()+hah.getFailcount());  mapHah.setTransDelay(mapHah.getTransDelay()+hah.getTransDelay());  map.put(key, mapHah);  }**else**{  map.put(key,hah);  }  }  //将map存入队列中，最终发给二级引擎进行合并  OwnEnv.*getMapQueue*().add(map);  //测试看是否能输出map的大小  System.***out***.println("map大小:"+map.size());  }  } **catch** (Exception e) {  e.printStackTrace();  }  }  } |

**将MapperRunner加入线程池，在common包下Start中添加如下代码：**

|  |
| --- |
| **public** **static** **void** main(String[] args){  System.***out***.println("一级引擎01节点启动");  ExecutorService es=Executors.*newCachedThreadPool*();  es.submit(**new** ZkConnectRunner());  es.submit(**new** RpcServer());  es.submit(**new** MapperRunner());  } |

**进行测试，先检查日志标识文件是否存在，然后启动一级引擎，再启动jobtracker，看控制台是否能打印出map的大小**

1. **设置一级引擎工作状态**

**现在只是发送了一条数据，发给了一个一级引擎，如果有多个一级引擎，首先每个引擎先发一个数据，但是后续发送时需要根据一级引擎是否繁忙来发送，所以需要一级引擎向zookeeper注册自己的繁忙状态信息**

**zebra-engine1-01工程zk包下的ZkConnectRunner类中分别添加设置节点繁忙和空闲的方法，代码如下：**

|  |
| --- |
| **public** **void** run() {  ......  }  **public** **static** **void** setBusy() {  **try** {  String data=InetAddress.*getLocalHost*().getHostAddress()+"/"+OwnEnv.*getRpcport*()+"/busy";  *zk*.setData(GlobalEnv.*getEngine1path*()+OwnEnv.*getZnodepath*(), data.getBytes(),-1);  } **catch** (Exception e) {  e.printStackTrace();  }  }  **public** **static** **void** setFree() {  **try** {  String data=InetAddress.*getLocalHost*().getHostAddress()+"/"+OwnEnv.*getRpcport*()+"/free";  *zk*.setData(GlobalEnv.*getEngine1path*()+OwnEnv.*getZnodepath*(), data.getBytes(),-1);  } **catch** (Exception e) {  e.printStackTrace();  }  } |

**修改mapper包下的MapperRunner，当从队列中取出切片后，即进入繁忙状态，当处理完的map存入队列后，即进入空闲状态，修改代码如下：**

|  |
| --- |
| FileSplit split=OwnEnv.*getSpiltQueue*().take();  **//此节点处于繁忙状态**  **ZkConnectRunner.*setBusy*();**  **......**  **......**  //将map存入队列中，最终发给二级引擎进行合并  OwnEnv.*getMapQueue*().add(map);  **//此节点处于空闲状态**  **ZkConnectRunner.*setFree*();** |

1. **jobtracker根据节点状态发送任务**

**jobtracker时刻监听zookeeper中一级引擎的节点状态，如果发生变化，并且为空闲，则发送任务**

**zebra-jobtracker工程rpc包下RpcClient中添加如下代码：**

|  |
| --- |
| @Override  **public** **void** run() {  **try** {  ......  proxy.sendFileSplit(GlobalEnv.*getSplitQueue*().take());  **for**(;;){  **final** CountDownLatch cdl=**new** CountDownLatch(1);  zk.getData(GlobalEnv.*getEngine1path*()+"/"+path,**new** Watcher(){  @Override  **public** **void** process(WatchedEvent event) {  **if**(event.getType()==EventType.***NodeDataChanged***){  **try** {  **byte**[] data=zk.getData(GlobalEnv.*getEngine1path*()+"/"+path, **null**, **null**);  **if**(**new** String(data).contains("free")){  //证明空闲，则发送任务  FileSplit split=GlobalEnv.*getSplitQueue*().take();  proxy.sendFileSplit(split);  cdl.countDown();  }**else**{  cdl.countDown();  }  } **catch** (Exception e) {  e.printStackTrace();  }  }  }  }, **null**);  cdl.await();  }  } **catch** (Exception e) {  e.printStackTrace();  }  } |

**进行测试，先检查日志标识文件是否存在，启动一级引擎，在启动jobtracker，如果打印出4次map大小，证明一个一级引擎可以收到所有任务。**

**再复制一份zebra-engine1-01起名为：zebra-engine1-02，修改配置文件中的端口号和节点名称，例如：**

|  |
| --- |
| zebra.rpcport=9992  zebra.zk.znodepath=/node02 |

**再次测试，先启动一级引擎的两个项目，在启动jobtracker，看两个一级引擎是否都能收到jobtracker发来的map（也可以复制三个一级引擎测试）**

1. **建立二级引擎rpc服务端及注册zookeeper**

**一级引擎处理完所有的map后，需要交给二级引擎做最后的合并，所以需要二级引擎建立rpc服务端，并且一级引擎需要获取二级引擎的ip与端口号，所以二级引擎同样需要在zookeeper中注册自己的节点信息**

**zebra-engine2项目下创建包cn.tedu.rpc，包下首先编写rpc存根的实现类RpcSendHttpAppHostImpl，并添加如下代码：**

|  |
| --- |
| **package** cn.tedu.rpc;  **import** java.util.Map;  **import** java.util.concurrent.BlockingQueue;  **import** java.util.concurrent.LinkedBlockingDeque;  **import** org.apache.avro.AvroRemoteException;  **import** rpc.domain.HttpAppHost;  **import** rpc.service.RpcSendHttpAppHost;  **public** **class** RpcSendHttpAppHostImpl **implements** RpcSendHttpAppHost{  @Override  **public** Void sendHttpAppHost(HttpAppHost httpAppHost) **throws** AvroRemoteException {  **return** **null**;  }  @Override  **public** Void sendMap(Map<CharSequence, HttpAppHost> hahMap) **throws** AvroRemoteException {  //打桩用于测试  System.***out***.println("二级引擎收到map："+hahMap.size());  **return** **null**;  }  } |

**rpc包下创建rpc服务端线程类RpcServer，用来接收一级引擎发来的数据，代码如下：**

|  |
| --- |
| **package** cn.tedu.rpc;  **import** java.net.InetSocketAddress;  **import** org.apache.avro.ipc.NettyServer;  **import** org.apache.avro.ipc.specific.SpecificResponder;  **import** rpc.service.RpcSendHttpAppHost;  /\*\*  \* Rpc服务端，用于接收一级引擎发来的数据  \*/  **public** **class** RpcServer **implements** Runnable{  @Override  **public** **void** run() {  **try** {  NettyServer server =**new** NettyServer(**new** SpecificResponder(RpcSendHttpAppHost.**class**,**new** RpcSendHttpAppHostImpl()),**new** InetSocketAddress(7777));  } **catch** (Exception e) {  e.printStackTrace();  }  }  } |

**二级引擎在zookeeper中注册自己的节点信息**

**创建包cn.tedu.zk，包下创建类ZkConnectRunner，并添加如下代码：**

|  |
| --- |
| **package** cn.tedu.zk;  **import** java.net.InetAddress;  **import** org.apache.zookeeper.CreateMode;  **import** org.apache.zookeeper.ZooDefs.Ids;  **import** org.apache.zookeeper.ZooKeeper;  **import** cn.tedu.zebra.common.GlobalEnv;  /\*\*  \* 二级引擎注册自身节点信息的  \*/  **public** **class** ZkConnectRunner **implements** Runnable{  **private** ZooKeeper zk;  @Override  **public** **void** run() {  **try** {  zk=GlobalEnv.*connectZkServer*();  // 端口号目前是写死的，有兴趣可以写到配置文件中，动态读取  String data=InetAddress.*getLocalHost*().getHostAddress()+"/7777";  zk.create(GlobalEnv.*getEngine2path*(), data.getBytes(),Ids.***OPEN\_ACL\_UNSAFE***, CreateMode.***EPHEMERAL***);  } **catch** (Exception e) {  e.printStackTrace();  }  }  } |

**创建包cn.tedu.commen，包下创建线启动类Start，创建线程池并将ZkConnectRunner和RpcServer添加到线程池，代码如下：**

|  |
| --- |
| **package** cn.tedu.common;  **import** java.util.concurrent.ExecutorService;  **import** java.util.concurrent.Executors;  **import** cn.tedu.rpc.RpcServer;  **import** cn.tedu.zk.ZkConnectRunner;  **public** **class** Start {  **public** **static** **void** main(String[] args) {  System.***out***.println("二级引擎启动");  ExecutorService es=Executors.*newCachedThreadPool*();  es.submit(**new** ZkConnectRunner());  es.submit(**new** RpcServer());  }  } |

1. **一级引擎和二级引擎连通**

**二级引擎rpc服务端已经创建好，一级引擎创建客户端将处理好的map发送给二级引擎，并测试二级引擎是否能收到。**

**zebra-engine1-01项目（可以先删除zebra-engine1-02，因为代码完全一样）包rpc下创建rpc客户端的线程类RpcClient，读取zookeeper中二级引擎的ip和端口号并将数据发送给二级引擎，代码如下：**

|  |
| --- |
| **package** cn.tedu.rpc;  **import** java.net.InetSocketAddress;  **import** org.apache.avro.ipc.NettyTransceiver;  **import** org.apache.avro.ipc.specific.SpecificRequestor;  **import** org.apache.zookeeper.ZooKeeper;  **import** cn.tedu.zebra.common.GlobalEnv;  **import** cn.tedu.zebra.common.OwnEnv;  **import** rpc.service.RpcSendHttpAppHost;  **public** **class** RpcClient **implements** Runnable{  **private** ZooKeeper zk;  @Override  **public** **void** run() {  **try** {  zk=GlobalEnv.*connectZkServer*();  //获取ip和端口号信息  **byte**[] data=zk.getData(GlobalEnv.*getEngine2path*(), **null**, **null**);  String ip=**new** String(data).split("/")[0];  **int** port=Integer.*parseInt*(**new** String(data).split("/")[1]);  NettyTransceiver client=**new** NettyTransceiver(**new** InetSocketAddress(ip, port));  RpcSendHttpAppHost proxy=SpecificRequestor.*getClient*(RpcSendHttpAppHost.**class**, client);  **while**(**true**){  proxy.sendMap(OwnEnv.*getMapQueue*().take());  }  } **catch** (Exception e) {  e.printStackTrace();  }  }  } |

**将RpcClient加入线程池，在common包下Start中添加如下代码：**

|  |
| --- |
| **public** **static** **void** main(String[] args){  System.***out***.println("一级引擎01节点启动");  ExecutorService es=Executors.*newCachedThreadPool*();  es.submit(**new** ZkConnectRunner());  es.submit(**new** RpcServer());  es.submit(**new** MapperRunner());  **es.submit(new RpcClient());**  } |

**进行联调测试，先检查日志标识文件，再启动二级引擎，然后启动一级引擎，最后启动jobtracker，如果控制台打印出二级引擎收到的map，则全部联通**

1. **二级引擎对所有的map进行合并**

**如果二级引擎收到一级引擎发送的所有map，则开始map的最后合并工作，当所有的map合并完之后，将数据通过jdbc落地到数据库中**

**zebra-engine2项目RpcSendHttpAppHostImpl中，接收所有的map并判断是否全部接收，如果全部接收，可以启动一个线程进行合并，RpcSendHttpAppHostImpl代码如下：**

|  |
| --- |
| **public** **class** RpcSendHttpAppHostImpl **implements** RpcSendHttpAppHost{  **public** **static** BlockingQueue<Map<CharSequence,HttpAppHost>> *queue*=**new** LinkedBlockingDeque<>();  ......  @Override  **public** Void sendMap(Map<CharSequence, HttpAppHost> hahMap) **throws** AvroRemoteException {  *queue*.add(hahMap);  //任务数量 jobtracker，可以通过注册zk，获取  **if**(*queue*.size()==4){  //启动合并  **new** Thread(**new** ReducerRunner()).start();  }  **return** **null**;  }  } |

**创建包cn.tedu.reduce，包下创建类ReducerRunner，用于合并数据，代码如下：**

|  |
| --- |
| **package** cn.tedu.reduce;  **import** java.util.HashMap;  **import** java.util.Map;  **import** java.util.Map.Entry;  **import** cn.tedu.rpc.RpcSendHttpAppHostImpl;  **import** rpc.domain.HttpAppHost;  /\*\*  \* 从队列里，取出4个map,最后合成一个map(包括用户的最后合并)  \*/  **public** **class** ReducerRunner **implements** Runnable{  //创建结果集map  **private** Map<String,HttpAppHost> map=**new** HashMap<>();  @Override  **public** **void** run() {  **while**(**true**){  Map<CharSequence,HttpAppHost> reduceMap=RpcSendHttpAppHostImpl.*queue*.poll();  **if**(reduceMap==**null**){  //证明已经所有任务都归并完，跳出循环  System.***out***.println("跳出循环");  **break**;  }**else**{  **for**(Entry<CharSequence,HttpAppHost> entry:reduceMap.entrySet()){  //用户标识  String key=entry.getKey().toString();  HttpAppHost hah=entry.getValue();  **if**(map.containsKey(key)){  //做数据累加  HttpAppHost mapHah=map.get(key);  mapHah.setAccepts(mapHah.getAccepts()+hah.getAccepts());  mapHah.setAttempts(mapHah.getAttempts()+hah.getAttempts());  mapHah.setTrafficUL(mapHah.getTrafficUL()+hah.getTrafficUL());  mapHah.setTrafficDL(mapHah.getTrafficDL()+hah.getTrafficDL());  mapHah.setRetranUL(mapHah.getRetranUL()+hah.getRetranUL());  mapHah.setRetranDL(mapHah.getRetranDL()+hah.getRetranDL());  mapHah.setTransDelay(mapHah.getTransDelay()+hah.getTransDelay());  map.put(key, mapHah);  }**else**{  map.put(key, hah);  }  }  }  }  System.***out***.println("最后结果map的大小"+map.size());  //map已经处理完数据，进行落地  ZebraDB.toDb(map);  }  } |

**合并完后，通过jdbc将数据写入到数据库，首先，将log4j.properties和c3p0-config.xml加入到src/main/java目录下，并创建包cn.tedu.zebradb，并创建类ZebraDB，通过批处理将数据插入到数据库中，代码如下：**

|  |
| --- |
| **package** cn.tedu.zebradb;  **import** java.sql.Connection;  **import** java.sql.PreparedStatement;  **import** java.sql.SQLException;  **import** java.util.Map;  **import** com.mchange.v2.c3p0.ComboPooledDataSource;  **import** rpc.domain.HttpAppHost;  **public** **class** ZebraDB {  **private** **static** ComboPooledDataSource *source* = **new** ComboPooledDataSource();  **public** **static** **void** toDb(Map<String, HttpAppHost> reduceMap) {  Connection conn = **null**;  PreparedStatement ps = **null**;  **try** {  conn=*source*.getConnection();  conn.setAutoCommit(**false**);  String sql = "insert into f\_http\_app\_host values (?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?)";  ps=conn.prepareStatement(sql);  **int** i=0;  **for**(Map.Entry<String, HttpAppHost> entry:reduceMap.entrySet()){  HttpAppHost hah=entry.getValue();  String reportTime=hah.getReportTime().toString();  String year=reportTime.substring(0, 4);  String month=reportTime.substring(4,6);  String day=reportTime.substring(6,8);  reportTime=year+"-"+month+"-"+day;    ps.setDate(1,java.sql.Date.*valueOf*(reportTime));  ps.setInt(2, hah.getAppType());  ps.setInt(3, hah.getAppSubtype());  ps.setString(4, hah.getUserIP().toString());  ps.setInt(5, hah.getUserPort());  ps.setString(6, hah.getAppServerIP().toString());  ps.setInt(7, hah.getAppServerPort());  ps.setString(8,hah.getHost().toString());  ps.setString(9,hah.getCellid().toString());  ps.setLong(10, hah.getAttempts());  ps.setLong(11, hah.getAccepts());  ps.setLong(12, hah.getTrafficUL());  ps.setLong(13, hah.getTrafficDL());  ps.setLong(14, hah.getRetranUL());  ps.setLong(15, hah.getRetranDL());  ps.setLong(16, hah.getTransDelay());  ps.setLong(17, hah.getTransDelay());    ps.addBatch();  i++;  **if**(i%1000 == 0){  ps.executeBatch();  ps.clearBatch();  }  }  //防止有残余的信息没有处理  ps.executeBatch();  conn.commit();  } **catch** (Exception e) {  **try** {  conn.rollback();  } **catch** (SQLException e1) {  e1.printStackTrace();  }  e.printStackTrace();  }**finally** {  **if**(ps!=**null**){  **try** {  ps.close();  } **catch** (Exception e) {  e.printStackTrace();  }**finally**{  ps = **null**;  }  }  **if**(conn!=**null**){  **try** {  conn.close();  } **catch** (SQLException e) {  e.printStackTrace();  }**finally** {  conn=**null**;  }  }  }  System.***out***.println("数据已经全部写到数据库里");  }  } |

**最后总测试，先检查日志文件，检查数据库中的表是否建好，然后启动二级引擎，再启动一级引擎，再启动jobtracker，运行结束后查看数据库数据是否写入，写入成功，则项目结束。**