# 课程介绍

每特教育第六期课程内容主要围绕Netty、Redis原理分析、高性能消息中间件、容器技术、微服务解决方案、微服务电商项目实战操作vue+SpringCloud

第六期的规划

社区平台 工单、面试宝典、核心资料

上课时间：每周二、周四、周六晚上20:30开课 周一强制必须听

# 网络编程基础

## OSI七层模型

应用层：Http协议、电子文件传输、文件服务器等

表示层：解决我们不同系统之间语法的通讯

会话层：建立与应用程序之间的通讯

传输层：提供了端口号和接口协议TPC/Udp

网络层：为数据包选择路由 路由器、交换机

定义了ip地址，可以根据ip地址找到对应的服务器

数据链路层：传输有地址的帧以及错误检测功能

物理层:以二进制形式，在物理机器上实现传输

（光纤、各种物理介质传输）

参考图来源：<http://www.colasoft.com.cn/download/protocols_map.php>

## Socket网络通讯技术

### TCP与UDP协议

#### Socket

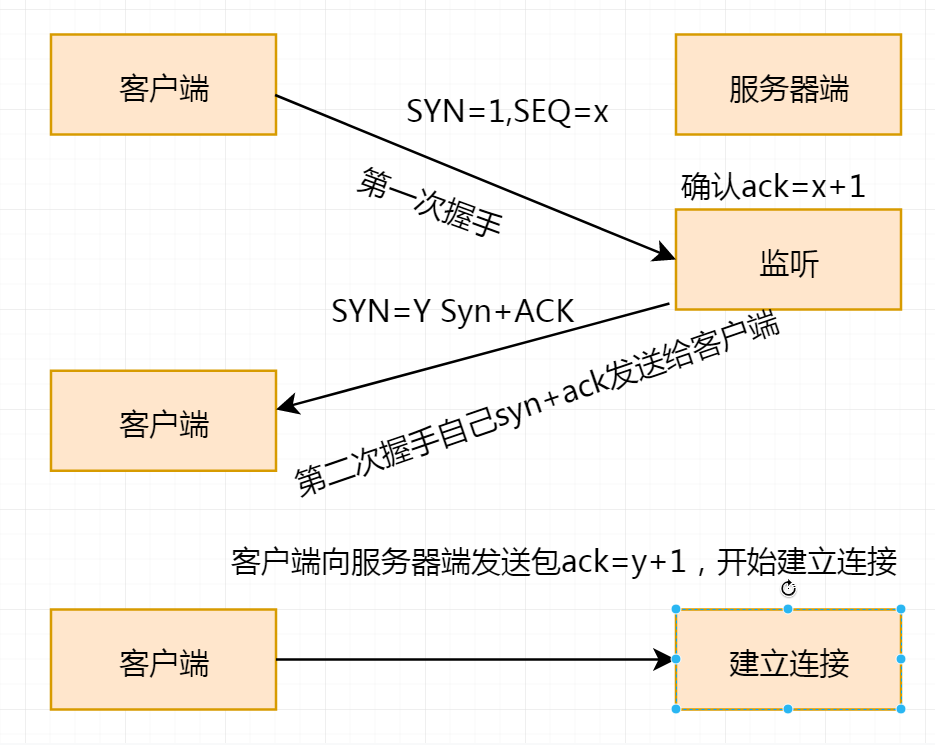
socket(套接字)是两个程序之间通过双向信道进行数据交换的端，可以理解为接口。使用socket编程也称为网络编程，socket只是接口并不是网络通信协议。

#### TCP与UDP区别

TCP是面向连接的可靠协议、通过三次握手建立连接，通讯完成时拆除连接

UDP是面向无连接通讯协议，udp通讯时不需要接受方确定，属于不可靠传输，可能会存在丢包的现象。

#### 三次握手和挥手（分手）概念



首先我们要知道在tcp建立连接中，有一些名词表示：

比如：syn就是建立连接、ack就是确认标志、fin终止标志

第一次握手：客户端会向服务器端发送码为syn=1，随机产生一个seq\_number=x的数据包到服务器端 （syn）

第二次握手：服务端接受到客户端请求之后，确认ack=x+1, 于是就向客户端发送syn（服务端独立生成 随机生成数字Y）+ack

第三次握手：客户端接受syn+ack，向服务器端发送ack=y+1，此包发送完毕即可 建立tcp连接。

白话文翻译：

第一次握手：客户端向服务器端发送 问服务器你在不在？

第二次握手：服务器端回应客户端说：我在的。

第三次握手：客户端发送给服务器端：ok，那我开始建立连接的

关闭连接：

第一次挥手： 客户端向服务器端发送释放的报文，停止发送数据 fin=1、生成一个序列号seq=u；

第二次挥手： 服务器端接受到释放的报文后，发送ack=u+1；随机生成的seq=v给客户端；当前状态为关闭等待状态

客户端收到了服务器确认通知之后，此时客户端就会进入到终止状态，等待服务器端发送释放报文。

第三次挥手：服务器端最后数据发送完毕之后，就向客户端发送连接释放报文，FIN=1，ack=u+1 当前为半关闭状态，随机生成一个随机树w

第四次挥手，客户端必须发出确认，ACK=1，ack=w+1，而自己的序列号是seq=u+1，此时，客户端就进入了TIME-WAIT（时间等待）状态。注意此时TCP连接还没有释放，必须经过2∗∗MSL（最长报文段寿命）的时间后，当客户端撤销相应的TCB后，才进入CLOSED状态。

服务器只要收到了客户端发出的确认，立即进入CLOSED状态。同样，撤销TCB后，就结束了这次的TCP连接。可以看到，服务器结束TCP连接的时间要比客户端早一些。

白话文翻译四次挥手：

第一次挥手 客户端向服务端发送一个释放连接通知；

第二次挥手 服务端接受到释放通知之后，告诉给客户端说等待一下，因为可能存在有其他的数据没有发送完毕，等待数据全部传输完毕之后就开始 关闭连接；

第三次挥手 服务器端所有的数据发送完毕之后，就告诉客户端说现在可以释放连接了。

第四次挥手： 客户端确认是最终释放连接通知，ok 就开始 就向服务区端发送我们可以开始关闭连接啦；

#### Socket Tcp通讯代码

##### 服务器端

|  |
| --- |
| **public class** SocketTcpServer {  **public static void** main(String[] args) **throws** IOException {  *// 创建Server Socket* ServerSocket serverSocket = **new** ServerSocket();  *// 创建我们的 Socket 监听连接地址和端口号* SocketAddress address = **new** InetSocketAddress(InetAddress.*getLocalHost*(), 8080);  *// 绑定我们的监听地址* serverSocket.bind(address);  *// 等待接受请求* System.***out***.println(**"等待客户端发送消息.."**);  Socket accept = serverSocket.accept();  *// 获取OutputStream流* PrintWriter socketOut = **new** PrintWriter(accept.getOutputStream());  **byte** buf[] = **new byte**[1024];  **if** (accept.getInputStream().read(buf) > 0) {  System.***out***.println(**"服务器端接受到客户端消息："** + **new** String(buf));  }  *// 服务器端响应消息* String sendStr = **"每特教育平均就业薪资突破了3万"**;  socketOut.write(sendStr);  socketOut.flush();   *// 关闭所有连接* socketOut.close();  accept.close();  serverSocket.close();  } } |

##### 客户端

|  |
| --- |
| **public class** SocketTcpClient {  **public static void** main(String[] args) **throws** IOException {  *// 创建socket* **final** Socket socket = **new** Socket();  *// 创建socket地址* SocketAddress address = **new** InetSocketAddress(InetAddress.*getLocalHost*(), 8080);  socket.connect(address);  *// 创建PrintWriter* PrintWriter socketOut = **new** PrintWriter(socket.getOutputStream());  BufferedReader socketIn = **new** BufferedReader(  **new** InputStreamReader(socket.getInputStream()));   *// 向服务器发送的内容* String sendStr = **"客户端问服务器端：你们每特教育第六期平均就业薪资突破多少？"**;  socketOut.write(sendStr);  socketOut.flush();  String receiveStr = socketIn.readLine();  System.***out***.println(**"服务器端回复：: "** + receiveStr);   *// 关闭连接* socketOut.close();  socketIn.close();  socket.close();  } } |

#### SocketUdp通讯代码

##### Udp服务器端

|  |
| --- |
| */\*  \* 接收客户端发送的数据  \*/ //1.创建服务器端DatagramSocket，指定端口* DatagramSocket socket = **new** DatagramSocket(8800); *//2.创建数据报，用于接收客户端发送的数据* **byte**[] data = **new byte**[1024]; *//创建字节数组，指定接收的数据包的大小* DatagramPacket packet = **new** DatagramPacket(data, data.**length**); *//3.接收客户端发送的数据* System.***out***.println(**"\*\*\*\*服务器端已经启动，等待客户端发送数据"**); *//此方法在接收到数据报之前会一直阻塞* socket.receive(packet); *//4.读取数据* String info = **new** String(data, 0, packet.getLength()); System.***out***.println(**"我是服务器，客户端说："** + info);  */\*  \* 向客户端响应数据  \*/ //1.定义客户端的地址、端口号、数据* InetAddress address = packet.getAddress(); **int** port = packet.getPort(); **byte**[] data2 = **"第六期已经开班啦~~"**.getBytes(); *//2.创建数据报，包含响应的数据信息* DatagramPacket packet2 = **new** DatagramPacket(data2, data2.**length**, address, port); *//3.响应客户端* socket.send(packet2); *//4.关闭资源* socket.close(); |

##### Udp客户端

|  |
| --- |
| */\*  \* 向服务器端发送数据  \*/ //1.定义服务器的地址、端口号、数据* InetAddress address = InetAddress.*getByName*(**"localhost"**); **int** port = 8800; **byte**[] data = **"每特教育第六期开班了吗？"**.getBytes(); *//2.创建数据报，包含发送的数据信息* DatagramPacket packet = **new** DatagramPacket(data, data.**length**, address, port); *//3.创建DatagramSocket对象* DatagramSocket socket = **new** DatagramSocket(); *//4.向服务器端发送数据报* socket.send(packet);  */\*  \* 接收服务器端响应的数据  \*/ //1.创建数据报，用于接收服务器端响应的数据* **byte**[] data2 = **new byte**[1024]; DatagramPacket packet2 = **new** DatagramPacket(data2, data2.**length**); *//2.接收服务器响应的数据* socket.receive(packet2); *//3.读取数据* String reply = **new** String(data2, 0, packet2.getLength()); System.***out***.println(**"我是客户端，服务器说："** + reply); *//4.关闭资源* socket.close(); |

### Http协议7个请求过程

Http协议一种超文本传输的协议，基于TCP/IP协议的包装，包含：img、css、js、html等。

Http协议的特征：

1. 无状态
2. 请求与响应模型
3. 简单快速
4. 灵活可以传输任何类型

Http分为 请求与响应

请求：

请求行

请求头

请求方法  
响应：

响应行

响应头

响应题

### 使用socket模拟Http服务器

|  |
| --- |
| **public class** HttpTest {  **public static void** main(String[] args) **throws** IOException {  ServerSocket serverSocket = **new** ServerSocket(8888);  *//一直监听，直到受到停止的命令* **while** (**true**) {  Socket socket = **null**;  **try** {  *//如果没有请求，会一直hold在这里等待，有客户端请求的时候才会继续往下执行* socket = serverSocket.accept();  *//获取输入流(请求)* BufferedReader bufferedReader = **new** BufferedReader(  **new** InputStreamReader(socket.getInputStream()));  StringBuilder stringBuilder = **new** StringBuilder();  String line = **null**;  *//得到请求的内容，注意这里作两个判断非空和""都要，只判断null会有问题* **while** ((line = bufferedReader.readLine()) != **null** && !line.equals(**""**)) {  stringBuilder.append(line).append(**"<br>"**);  }  String result = stringBuilder.toString();  System.***out***.println(result);  *//这里第二个参数表示自动刷新缓存* PrintWriter printWriter = **new** PrintWriter(  socket.getOutputStream(), **true**);  printWriter.println(**"HTTP/1.1 200 OK"**);  printWriter.println(**"Content-Type:text/html;charset=utf-8"**);  printWriter.println();   printWriter.println(**"<h5>你刚才发送的请求数据是：<br>"**);  *//将日志输出到浏览器* printWriter.write(result);  printWriter.println(**"</h5>"**);  *// release* printWriter.close();  bufferedReader.close();  socket.close();  } **catch** (IOException e) {  e.printStackTrace();  }  }  } } |

### 长连接与短连接区别

Http协议1.0： 规定客户端与浏览器保持短暂的连接，服务器完成数据传输数据之后就会立马关闭连接，如果频繁的向服务器端发送请求的话，这时候是非常消耗服务器资源。

Http协议1.1： Http协议从1.1开始就支持长连接，会根据特定时间保持当前连接状态，不会立马关闭掉tcp连接；

Http协议建立长连接之后，默认情况下在300s为空闲状态情况下会主动断开连接；

# Linux网络IO模型

### 同步和异步的区别

同步也就是程序从上往下实现执行；

异步从新开启一个新分支，相互不会影响；

站在Http协议上分析同步与异步区别：

我们的Http协议请求默认情况下同步形式调用，如果调用过程非常耗时的情况下 客户端等待时间就非常长， 这这种形式我们可以理解阻塞式；

解决办法：耗时的代码我们可以使用多线程实现处理，但是不能立马获取结果； 客户端可以主动查询

### Bio演变的NIO的过程

Bio是一个阻塞式的io，不能够支持并发请求访问；可以多线程优化代码

这种方式也存在缺点：如果每个请求过来都使用一个线程，这时候非常浪费CPU的资源。

所以在网络编程服务器中，是否使用单线程提高响应的效率问题，所以nio出现；

|  |
| --- |
| **public class** ServerTcpSocket {  **static byte**[] *bytes* = **new byte**[1024];   **public static void** main(String[] args) {  ExecutorService executorService = Executors.*newCachedThreadPool*();  **try** {  *// 1.创建一个ServerSocket连接* **final** ServerSocket serverSocket = **new** ServerSocket();  *// 2.绑定端口号* serverSocket.bind(**new** InetSocketAddress(8080));  *// 3.当前线程放弃cpu资源等待获取数据* System.***out***.println(**"等待获取数据..."**);  **while** (**true**) {  **final** Socket socket = serverSocket.accept();  executorService.execute(**new** Runnable() {  **public void** run() {  **try** {  System.***out***.println(**"获取到数据..."**);  *// 4.读取数据* **int** read = socket.getInputStream().read(*bytes*);  String result = **new** String(*bytes*);  System.***out***.println(result);  } **catch** (Exception e) {   }  }  });   }  } **catch** (Exception e) {   }  } } |

NIO非阻塞式代码

|  |
| --- |
| **public class** ServerNioTcpSocket {  **static** ByteBuffer *byteBuffer* = ByteBuffer.*allocate*(512);   **public static void** main(String[] args) {  **try** {  *// 1.创建一个ServerSocketChannel连接* **final** ServerSocketChannel serverSocketChannel = ServerSocketChannel.*open*();  *// 2.绑定端口号* serverSocketChannel.bind(**new** InetSocketAddress(8080));  *// 设置为非阻塞式* serverSocketChannel.configureBlocking(**false**);  *// 非阻塞式* SocketChannel socketChannel = serverSocketChannel.accept();  **if** (socketChannel != **null**) {  **int** j = socketChannel.read(*byteBuffer*);  **if** (j > 0) {  **byte**[] bytes = Arrays.*copyOf*(*byteBuffer*.array(), *byteBuffer*.limit());  System.***out***.println(**"获取到数据"** + **new** String(bytes));  }  }  System.***out***.println(**"程序执行完毕.."**);   } **catch** (Exception e) {  e.printStackTrace();  }  } } |

|  |
| --- |
| **public class** ClientTcpSocket {  **public static void** main(String[] args) {  **try** {  Socket socket = **new** Socket();  SocketAddress address = **new** InetSocketAddress(InetAddress.*getLocalHost*(), 8080);  socket.connect(address);  socket.getOutputStream().write(**"mayikt"**.getBytes());  } **catch** (Exception e) {   }  } } |

### 五种IO模型

Linux下的五种I/O模型

1)阻塞I/O（blocking I/O）

2)非阻塞I/O （nonblocking I/O）

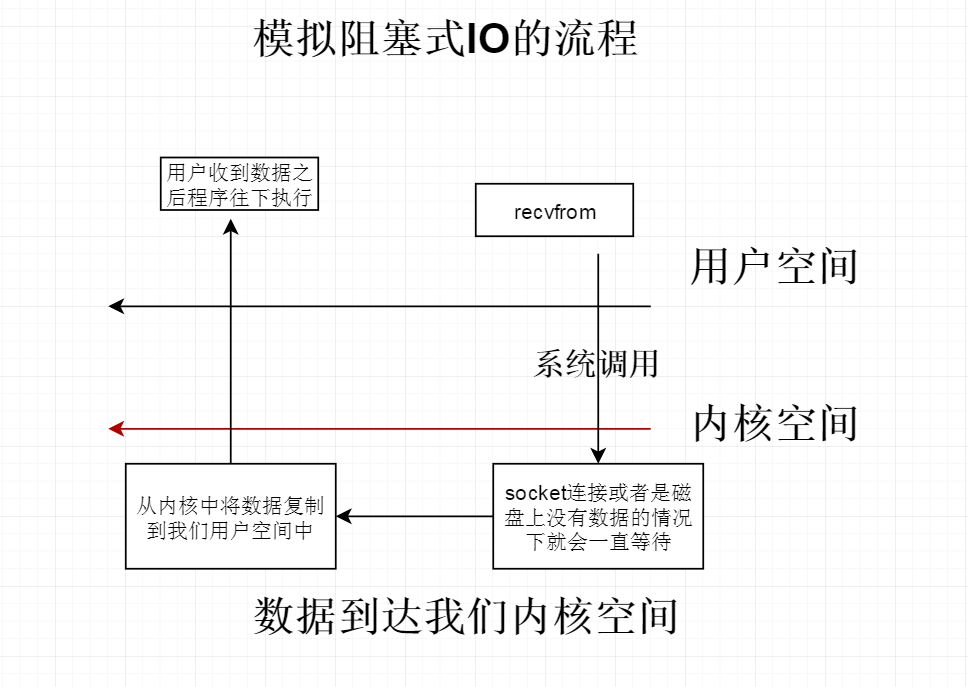
3) I/O复用(select 和poll) （I/O multiplexing）

4)信号驱动I/O （signal driven I/O (SIGIO)）

5)异步I/O （asynchronous I/O (the POSIX aio\_functions)）

前面四种都是同步io、第五种是异步IO；

#### 阻塞I/O



专业话术：

当我们在调用一个io函数的时候，如果没有获取到数据的情况下，那么就会一直等待；等待的过程中会导致整个应用程序一直是一个阻塞的过程，无法去做其他的实现。

*白话文：*

比如我们现在每特教育第六期要开班啦，由于口碑非常好、就业薪资非常高 这个时候有很多程序猿来排队报名学习；这个时候小军前面还有100个学员正在排队报名，小军必须要等待前面100个学员报名完成之后，轮训到小军才可以报名；小军为了保证当前的排队位置存在，也不能做其他的事情，这个过程我们可以称作为阻塞式；

用户空间与内核空间区别：

#### 非阻塞I/O

专业数据：

不管是否有获取到数据，都会立马获取结果，如果没有获取数据的话、那么就不间断的循环重试，但是我们整个应用程序不会实现阻塞。

白话文：

小军安排黄牛帮我们代替排队，每次间隔一段时间咨询黄牛 是否轮到自己呢。那么这个时候小军可以实现做其他的事情

这种非阻塞式的io还存在那些缺点？也是非常消耗cpu的资源，但是程序不会阻塞。

Cas、乐观锁与悲观锁

#### IO复用(select，poll，epoll)

IO复用机制：IO实际指的就是网络的IO、多路也就是多个不同的tcp连接；复用也就是指使用同一个线程合并处理多个不同的IO操作，这样的话可以减少CPU资源。

白话文：单个线程可以同时处理多个不同的io操作，应用场景非常广泛：redis原理。Mysql连接原理

下周二重点分析nio的复用机制

#### 信号驱动I/O

发出一个请求实现观察监听，当有数据的时候直接走我们异步回调；

白话文：小军在排队的时候 只需要去领取一个排队的号码，等到叫到了小军的时候才开始处理业务，这时候小军实际上还是可以去做其他的事情。 银行架构

#### 异步I/O(Aio)

异步io也就是发出请求数据之后，剩下的事情完全实现异步完成

# Nio与Bio的核心区别

## 什么是NIO

Java的nio是在Jdk1.4版本之后推出了一套新的io方案，这种io方案对原有io做了一次性能上的升级

NIO翻译成英文 no blocking io 简称为 nio 非阻塞io，不是new io。

比传统的io支持了面向缓冲区、基于通道实现的io的方案。

## NIO与BIO区别

|  |  |
| --- | --- |
| IO | Nio |
| 面向流(Stream oriented) | 面向缓冲区(Buffer oriented) |
| 阻塞式(Blocking IO) | 非阻塞式(Non blocking IO) |
|  | 选择器(Selectors) |

传统的bio（同步阻塞 ）是面向与流传输的，而NIO（同步非阻塞io）是面向与缓冲区非阻塞式的io，其中最大的亮点就是多路io复用机制。

阻塞式io：当我们没有获取到数据的时候，整个应用程序会实现阻塞等待，不能实现做其他的事情。

非阻塞式io：不管是否有获取到数据，都必须立马获取到结果，如果没有获取数据的情况下，就会不断的重试获取数据

类似于之前学习：cas、悲观和乐观锁

## 引入nio操作

# NIO的核心组件

## 通道（Channel）

通常我们nio所有的操作都是通过通道开始的，所有的通道都会注册到统一个选择器(Selector)上实现管理，在通过选择器将数据统一写入到 buffer中。

## 缓冲区(Buffer)

Buffer本质上就是一块内存区，可以用来读取数据，也就先将数据写入到缓冲区中、在统一的写入到硬盘上。

## 选择器(Selector)

Selector可以称做为选择器，也可以把它叫做多路复用器，可以在单线程的情况下可以去维护多个Channel，也可以去维护多个连接；

## NIO设计思想伪代码

|  |
| --- |
| **public class** SocketNioTcpServer {  **private static** List<SocketChannel> *listSocketChannel* = **new** ArrayList<>();  **private static** ByteBuffer *byteBuffer* = ByteBuffer.*allocate*(512);   **public static void** main(String[] args) {  **try** {  *// 1.创建一个ServerSocketChannel* ServerSocketChannel serverSocketChannel = ServerSocketChannel.*open*();  *// 2. 绑定地址* ServerSocketChannel bind = serverSocketChannel.bind(**new** InetSocketAddress(8080));  serverSocketChannel.configureBlocking(**false**);  **while** (**true**) {  SocketChannel socketChannel = serverSocketChannel.accept();  **if** (socketChannel != **null**) {  socketChannel.configureBlocking(**false**);  *listSocketChannel*.add(socketChannel);  }  **for** (SocketChannel scl : *listSocketChannel*) {  **try** {  **int** read = scl.read(*byteBuffer*);  **if** (read > 0) {  *byteBuffer*.flip();  Charset charset = Charset.*forName*(**"UTF-8"**);  String receiveText = charset.newDecoder().decode  (*byteBuffer*.asReadOnlyBuffer()).toString();  System.***out***.println(**"receiveText:"** + receiveText);  }  } **catch** (Exception e) {  e.printStackTrace();  }  }  }  } **catch** (Exception e) {  e.printStackTrace();  }  } } |

## 使用jdk原生api实现nio

|  |
| --- |
| **public class** NIOServer {   */\*\*  \* 创建一个选择器  \*/* **private** Selector **selector**;   **public void** initServer(**int** port) **throws** IOException {  *// 获得一个ServerSocketChannel通道* ServerSocketChannel serverSocketChannel = ServerSocketChannel.*open*();  *// 设置通道为非阻塞* serverSocketChannel.configureBlocking(**false**);  *// 将该通道对应的ServerSocket绑定到port端口* serverSocketChannel.bind(**new** InetSocketAddress(port));  *// 获得一个通道管理器* **this**.**selector** = Selector.*open*();  *// 将通道管理器和该通道绑定，并为该通道注册SelectionKey.OP\_ACCEPT事件,注册该事件后，  // 当该事件到达时，selector.select()会返回，如果该事件没到达selector.select()会一直阻塞。* serverSocketChannel.register(**selector**, SelectionKey.***OP\_ACCEPT***);  }   **public void** listen() **throws** IOException {  System.***out***.println(**"服务端启动成功！"**);  *// 轮询访问selector* **while** (**true**) {  *// 当注册的事件到达时，方法返回；否则,该方法会一直阻塞* **int** select = **selector**.select();  **if** (select == 0) {  **continue**;  }  *// 获得selector中选中的项的迭代器，选中的项为注册的事件* Iterator<SelectionKey> ite = **this**.**selector**.selectedKeys().iterator();  **while** (ite.hasNext()) {  SelectionKey key = (SelectionKey) ite.next();  *// 删除已选的key,以防重复处理* ite.remove();   **if** (key.isAcceptable()) {*// 客户端请求连接事件* ServerSocketChannel server = (ServerSocketChannel) key.channel();  *// 获得和客户端连接的通道* SocketChannel channel = server.accept();  *// 设置成非阻塞* channel.configureBlocking(**false**);   *// 在和客户端连接成功之后，为了可以接收到客户端的信息，需要给通道设置读的权限。* channel.register(**this**.**selector**, SelectionKey.***OP\_READ***);   } **else if** (key.isReadable()) {*// 获得了可读的事件* read(key);  }   }   }  }   **public void** read(SelectionKey key) **throws** IOException {  *// 服务器可读取消息:得到事件发生的Socket通道* SocketChannel channel = (SocketChannel) key.channel();  *// 创建读取的缓冲区* ByteBuffer buffer = ByteBuffer.*allocate*(512);  channel.read(buffer);  **byte**[] data = buffer.array();  String msg = **new** String(data).trim();  System.***out***.println(**"服务端收到信息："** + msg);  ByteBuffer outBuffer = ByteBuffer.*wrap*(msg.getBytes(**"utf-8"**));  channel.write(outBuffer);*// 将消息回送给客户端* }   **public static void** main(String[] args) **throws** IOException {  NIOServer server = **new** NIOServer();  server.initServer(8000);  server.listen();  } } |

## Redis是如何单线程能够非常好支持并发

首先Redis官方是没有windows版本的，只有redis版本

Redis的底层采用Nio中的多路IO复用的机制，能够非常好的支持这样的并发，从而保证线程安全问题；

但是Nio在不同的操作系统上实现的方式有所不同，在我们windows操作系统使用select实现轮训时间复杂度是为o(n)，而且还存在空轮训的情况，效率非常低， 其次是默认对我们轮训的数据有一定限制，所以支持上万的tcp连接是非常难。

所以在linux操作系统采用epoll实现事件驱动回调，不会存在空轮训的情况，只对活跃的 socket连接实现主动回调这样在性能上有大大的提升，所以时间复杂度是为o(1)

注意：windows操作系统是没有epoll，只有linux系统才有epoll

所以为什么nginx、redis都能够非常高支持高并发，最终都是linux中的IO多路复用机制epoll

# Netty框架

## 不选择Java原生NIO编程的原因

1.NIO的类库和API繁杂，使用麻烦，你需要熟练掌握Selector、ServerSocketChannel、SocketChannel、ByteBuffer等。

2.需要具备其他的额外技能做铺垫，例如熟悉Java多线程编程。这是因为NIO编程涉及到Reactor模式，你必须对多线程和网路编程非常熟悉，才能编写出高质量的NIO程序。

3.可靠性能力补齐，工作量和难度都非常大。例如客户端面临断连重连、网络闪断、半包读写、失败缓存、网络拥塞和异常码流的处理等问题，NIO编程的特点是功能开发相对容易，但是可靠性能力补齐的工作量和难度都非常大。

1. JDK NIO的BUG，例如臭名昭著的epoll bug，它会导致Selector空轮询，最终导致CPU 100%。官方声称在JDK1.6版本的update18修复了该问题，但是直到JDK1.7版本该问题仍旧存在，只不过该BUG发生概率降低了一些而已，它并没有被根本解决。该BUG以及与该BUG相关的问题单可以参见以下链接内容。

由于上述原因，在大多数场景下，不建议大家直接使用JDK的NIO类库，除非你精通NIO编程或者有特殊的需求。在绝大多数的业务场景中，我们可以使用NIO框架Netty来进行NIO编程，它既可以作为客户端也可以作为服务端，同时支持UDP和异步文件传输，功能非常强大。

## 为什么选择Netty框架

Netty是业界最流行的NIO框架之一，它的健壮性、功能、性能、可定制性和可扩展性在同类框架中都是首屈一指的，它已经得到成百上千的商用项目验证，例如Hadoop的RPC框架Avro就使用了Netty作为底层通信框架，其他还有业界主流的RPC框架，也使用Netty来构建高性能的异步通信能力。

通过对Netty的分析，我们将它的优点总结如下。

◎ API使用简单，开发门槛低；

◎ 功能强大，预置了多种编解码功能，支持多种主流协议；

◎ 定制能力强，可以通过ChannelHandler对通信框架进行灵活地扩展；

◎ 性能高，通过与其他业界主流的NIO框架对比，Netty的综合性能最优；

◎ 成熟、稳定，Netty修复了已经发现的所有JDK NIO BUG，业务开发人员不需要再为NIO的BUG而烦恼；

◎ 社区活跃，版本迭代周期短，发现的BUG可以被及时修复，同时，更多的新功能会加入；

◎ 经历了大规模的商业应用考验，质量得到验证。Netty在互联网、大数据、网络游戏、企业应用、电信软件等众多行业已经得到了成功商用，证明它已经完全能够满足不同行业的商业应用了。

正是因为这些优点，Netty逐渐成为了Java NIO编程的首选框架。

## 基于Netty框架构建Nio编程

### 创建一个Netty项目

#### Maven依赖

|  |
| --- |
| <**dependency**>  <**groupId**>io.netty</**groupId**>  <**artifactId**>netty-all</**artifactId**>  <**version**>4.1.42.Final</**version**> </**dependency**> |

#### 服务器端

|  |
| --- |
| **public class** NettyServer {   */\*\*  \* netty启动端口号  \*/* **private static int** *port* = 8080;   **public static void** main(String[] args) {  */\*\*  \* 客户端创建两个线程池组分别为 boss线程组和工作线程组  \*/  // 用于接受客户端连接的请求 （并没有处理请求）* NioEventLoopGroup bossGroup = **new** NioEventLoopGroup();  *// 用于处理客户端连接的读写操作* NioEventLoopGroup workGroup = **new** NioEventLoopGroup();  *// 用于创建我们的ServerBootstrap* ServerBootstrap serverBootstrap = **new** ServerBootstrap();  serverBootstrap.group(bossGroup, workGroup).channel(NioServerSocketChannel.**class**)  .childHandler(**new** ChannelInitializer<SocketChannel>() {  @Override  **protected void** initChannel(SocketChannel socketChannel) **throws** Exception {  socketChannel.pipeline().addLast(**new** ServerHandler());  }  });  *// 绑定我们的端口号码* **try** {  *// 绑定端口号，同步等待成功* ChannelFuture future = serverBootstrap.bind(*port*).sync();  System.***out***.println(**"服务器启动成功："** + *port*);  *// 等待服务器监听端口* future.channel().closeFuture().sync();  } **catch** (Exception e) {  e.printStackTrace();   } **finally** {  *// 优雅的关闭连接* bossGroup.shutdownGracefully();  workGroup.shutdownGracefully();  }   }  }  **public class** ServerHandler **extends** SimpleChannelInboundHandler {  @Override  **protected void** channelRead0(ChannelHandlerContext ctx, Object o) **throws** Exception {  *// 接受我们的数据* ByteBuf byteBuf = (ByteBuf) o;  String request = byteBuf.toString(CharsetUtil.***UTF\_8***);  System.***out***.println(**"request:"** + request);   *// 响应内容:* ctx.writeAndFlush(Unpooled.*copiedBuffer*(**"平均突破3万月薪"**, CharsetUtil.***UTF\_8***));    }   } |

#### 客户端

##### Socket原生客户端

|  |
| --- |
| **public class** SocketClient {   **public static void** main(String[] args) **throws** IOException {  Socket socket = **new** Socket(**"127.0.0.1"**, 8080);  OutputStream outputStream = socket.getOutputStream();  **for** (**int** i = 0; i < 10; i++) {  outputStream.write(**"mayikt"**.getBytes());  }  outputStream.close();  socket.close();  } } |

##### Netty客户端

|  |
| --- |
| **public class** NettyClient {  **public static void** main(String[] args) {  *//创建nioEventLoopGroup* NioEventLoopGroup group = **new** NioEventLoopGroup();  Bootstrap bootstrap = **new** Bootstrap();  bootstrap.group(group).channel(NioSocketChannel.**class**)  .remoteAddress(**new** InetSocketAddress(**"127.0.0.1"**, 8080))  .handler(**new** ChannelInitializer<SocketChannel>() {  @Override  **protected void** initChannel(SocketChannel ch) **throws** Exception {  ch.pipeline().addLast(**new** ClientHandler());  }  });  **try** {  *// 发起同步连接* ChannelFuture sync = bootstrap.connect().sync();  sync.channel().closeFuture().sync();  } **catch** (Exception e) {   } **finally** {  group.shutdownGracefully();  }   } }  **public class** ClientHandler **extends** SimpleChannelInboundHandler<ByteBuf> {    */\*\*  \* 活跃通道可以发送消息  \*  \** ***@param ctx*** *\** ***@throws*** *Exception  \*/* @Override  **public void** channelActive(ChannelHandlerContext ctx) **throws** Exception {  *// 发送数据* ctx.writeAndFlush(Unpooled.*copiedBuffer*(**"每特教育平均月薪突破多少?"**, CharsetUtil.***UTF\_8***));  }   */\*\*  \* 读取消息  \*  \** ***@param ctx*** *\** ***@param msg*** *\** ***@throws*** *Exception  \*/* @Override  **protected void** channelRead0(ChannelHandlerContext ctx, ByteBuf msg) **throws** Exception {  System.***out***.println(**"resp："** + msg.toString(CharsetUtil.***UTF\_8***));  } } |

### 粘包拆包解决方案：

什么 是粘包、拆包

原因的造成：

因为我们现在的tcp连接默认为长连接的形式实现通讯，发送请求之后不会立马关闭连接

客户端与服务器端建立连接，客户端发送一条消息，客户端与服务器端关闭连接

客户端与服务器端建立连接，客户端发送多条消息，客户端与服务器端关闭连接

什么是粘包：多次发送的消息，客户端一次合并读取 msg+msg

什么是拆包：第一次完整消息+第二次部分消息组合 、第二次缺失的消息

Msg Msg=msgmsg

Msg Msg=MsgM sg

粘包与拆包产生的背景：

Tcp协议为了高性能的传输，发送和接受的时候都采用了缓冲区

1. 当我们的应用程序发送的数据大于套字节缓冲区的时候，就会实现拆包。
2. 当我们的应用程序写入的数据小于套字节缓冲区的时候，多次发送的消息会合并到一起接受，这个过程我们可以称做为粘包。
3. 接受端不够及时的获取缓冲区的数据，也会产生粘包的问题
4. 进行mss（最大报文长度）大小的TCP分段，当TCP报文长度-TCP头部长度>mss的时候将发生拆包。

解决思路：

1.以固定的长度发送数据，到缓冲区

2.可以在数据之间设置一些边界(\n或者\r\n)

利用编码器LineBaseDFrameDecoder解决tcp粘包的问题

|  |
| --- |
| serverBootstrap.group(bossGroup, workGroup).channel(NioServerSocketChannel.**class**)  .childHandler(**new** ChannelInitializer<SocketChannel>() {  @Override  **protected void** initChannel(SocketChannel socketChannel) **throws** Exception {  *// 设置我们分割最大长度为1024* socketChannel.pipeline().addLast(**new** LineBasedFrameDecoder(1024));  *// 获取数据的结果为string类型* socketChannel.pipeline().addLast(**new** StringEncoder());  socketChannel.pipeline().addLast(**new** ServerHandler());  }  }); |

### 常用编码器

　DelimiterBasedFrameDecoder 解决TCP的粘包解码器

　StringDecoder 消息转成String解码器

　LineBasedFrameDecoder 自动完成标识符分隔解码器

　FixedLengthFrameDecoder 固定长度解码器，二进制

　Base64Decoder解码器

LineBasedFrameDecoder 以设置特定的字符

|  |
| --- |
| ByteBuf byteBuf = Unpooled.*copiedBuffer*(**"$\_"**.getBytes()); ch.pipeline().addLast(**new** DelimiterBasedFrameDecoder(1024, byteBuf)); |

FixedLengthFrameDecoder

## 序列化与反序列化

序列化：把对象转换成字节的过程，称为对象序列化

反序列化：把字节恢复成对象的过程，称为反序列化

1. 对象的持久化概念：把字节保存的硬盘上永久的存放
2. 网络传输对象概念:客户端将对象序列化为字节（序列化），变成二进制的形式发送到服务器端端，服务器端接受到字节对象后，反序列化成对象

注意序列化的类必须要实现Serializable接口 transient修士变量不被序列化

## Java实现序列化与反序列化

|  |
| --- |
| **import** com.mayikt.entity.MsgEntity;  **import** java.io.\*; **import** java.util.Date; **import** java.util.UUID;  */\*\*  \** ***@author*** *蚂蚁课堂创始人-余胜军QQ644064779  \** ***@title:*** *Test002  \** ***@description:*** *每特教育独创第五期互联网架构课程  \** ***@date*** *2019/10/2918:42  \*/* **public class** Test002 {  **public static void** main(String[] args) {  String fileName=**"D:\\codes\\msg"**;  *serialize*(fileName);  *deserialize*(fileName);  }   */\*\*  \* 序列化对象到文件  \*  \** ***@param fileName*** *\*/* **public static void** serialize(String fileName) {  **try** {  ObjectOutputStream out = **new** ObjectOutputStream(**new** FileOutputStream(fileName));  *//序列化一个字符串到文件* out.writeObject(**"序列化的日期是："**);  *//序列化一个当前日期对象到文件* out.writeObject(**new** Date());  MsgEntity msgEntity = **new** MsgEntity(UUID.*randomUUID*().toString(), **"每特教育真的牛逼!"**);  *//序列化一个MsgEntity对象* out.writeObject(msgEntity);  out.close();   } **catch** (FileNotFoundException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  }  }   */\*\*  \* 反序列化  \*  \** ***@param fileName*** *\*/* **public static void** deserialize(String fileName) {  **try** {  *//从本地文件中读取字节 变为对象* ObjectInputStream in = **new** ObjectInputStream(**new** FileInputStream(fileName));  String str = (String) in.readObject();  *//日期对象* Date date = (Date) in.readObject();  *//MsgEntity对象* MsgEntity userInfo = (MsgEntity) in.readObject();  System.***out***.println(str);  System.***out***.println(date);  System.***out***.println(userInfo);  } **catch** (FileNotFoundException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  } **catch** (ClassNotFoundException e) {  e.printStackTrace();  }  } } |

### 其他序列化实现方案

1. 将对象转换成json
2. 用XML格式序列化
3. ProtoBuf（谷歌第三方协议）

### 使用json格式实现rpc通讯

|  |
| --- |
| **public class** ServerHandler **extends** SimpleChannelInboundHandler<String> {  @Override  **protected void** channelRead0(ChannelHandlerContext channelHandlerContext, String o) **throws** Exception {  System.***out***.println(**"o:"** + o);  MsgEntity msgEntity = JSONObject.*parseObject*(o, MsgEntity.**class**);  System.***out***.println(**"msgEntity:"** + msgEntity);  } }  ch.pipeline().addLast(**new** StringEncoder()); ch.pipeline().addLast(**new** ClientHandler());  **public class** ClientHandler **extends** SimpleChannelInboundHandler<String> {    */\*\*  \* 活跃通道可以发送消息  \*  \** ***@param ctx*** *\** ***@throws*** *Exception  \*/* @Override  **public void** channelActive(ChannelHandlerContext ctx) **throws** Exception {  MsgEntity msgEntity = **new** MsgEntity(UUID.*randomUUID*().toString(), **"每特教育真牛逼"**);  String json = JSONObject.*toJSONString*(msgEntity);  *// 发送数据* ctx.writeAndFlush(json);  }    */\*\*  \* 读取消息  \*  \** ***@param channelHandlerContext*** *\** ***@param s*** *\** ***@throws*** *Exception  \*/* @Override  **protected void** channelRead0(ChannelHandlerContext channelHandlerContext, String s) **throws** Exception {  System.***out***.println(**"resp："** + s);  } }  socketChannel.pipeline().addLast(**new** StringDecoder()); socketChannel.pipeline().addLast(**new** ServerHandler()); |

### 业界主流的编解码框架

Facebook 的 Thrift

Jboos 的 Marshaling

MessagePack

谷歌protobuf

### MessagePack编码器

它像 JSON，但是更快更小。

MessagePack 是一种高效的二进制序列化格式。它允许您在 JSON 等多种语言之间交换数据，但它更快速更小巧。小整数被编码为单个字节，典型的短字符串除了字符串本身之外只需要一个额外的字节。

支持Python、Ruby、Java、C/C++等众多语言。宣称比Google Protocol Buffers还要快4倍。

#### API简单的使用

##### Maven的依赖

|  |
| --- |
| *<!-- MessagePack dependency -->* <**dependency**>  <**groupId**>org.msgpack</**groupId**>  <**artifactId**>msgpack</**artifactId**>  <**version**>0.6.12</**version**> </**dependency**> <**dependency**>  <**groupId**>io.netty</**groupId**>  <**artifactId**>netty-all</**artifactId**>  <**version**>4.1.32.Final</**version**> </**dependency**> |

##### 基本的API

|  |
| --- |
| *// 创建MessagePack* MessagePack messagePack = **new** MessagePack();  MsgEntity meite = **new** MsgEntity(UUID.*randomUUID*().toString(), **"每特教育"**);  *// 序列化* **byte**[] bs = messagePack.write(meite);  Value read1 = messagePack.read(bs);  System.***out***.println(read1); *// // 反序列化* MsgEntity read = messagePack.read(bs, MsgEntity.**class**);  System.***out***.println(read); |

### Netty使用 MessagePack编码器

#### 编码器

|  |
| --- |
| **public class** MsgpackEncoder **extends** MessageToByteEncoder {    */\*\*  \* 对我们数据实现编码  \*  \** ***@param channelHandlerContext*** *\** ***@param msg*** *\** ***@param byteBuf*** *\** ***@throws*** *Exception  \*/* @Override  **protected void** encode(ChannelHandlerContext channelHandlerContext, Object msg, ByteBuf byteBuf) **throws** Exception {  MessagePack msgpack = **new** MessagePack();  byteBuf.writeBytes(msgpack.write(msg));  } } |

#### 解码器

|  |
| --- |
| **public class** MsgpackDecoder **extends** MessageToMessageDecoder<ByteBuf> {  */\*\*  \* 服务器解码数据  \*  \** ***@param channelHandlerContext*** *\** ***@param byteBuf*** *\** ***@param list*** *\** ***@throws*** *Exception  \*/* @Override  **protected void** decode(ChannelHandlerContext channelHandlerContext, ByteBuf byteBuf, List<Object> list) **throws** Exception {  **final int** length = byteBuf.readableBytes();  **byte**[] b = **new byte**[length];  byteBuf.getBytes(byteBuf.readerIndex(), b, 0, length);  MessagePack msgpack = **new** MessagePack();  list.add(msgpack.read(b));  } } |

#### 客户端与服务器端

服务器端：

|  |
| --- |
| socketChannel.pipeline().addLast(**new** MsgpackDecoder()); socketChannel.pipeline().addLast(**new** ServerHandler()); |

客户端:

|  |
| --- |
| ch.pipeline().addLast(**new** MsgpackEncoder()); ch.pipeline().addLast(**new** ClientHandler()); |

# 基于Netty实现Dubbo

## 手写Dubbo需要设计的技术点

1. 基于Netty实现rpc框架
2. 使用Zookeeper实现服务列表注册
3. 设计Java序列化、反序列化技术
4. 多线程与并发编程技术
5. 自定义注解、反射技术

## Dubbo+Zookeeper环境搭建

### 什么是Dubbo

是一款高性能、轻量级的开源Java RPC框架，它提供了三大核心能力：面向接口的远程方法调用，智能容错和负载均衡，以及服务自动注册和发现。

Dubbo官网：<http://dubbo.apache.org/zh-cn/>



### Dubbo底层实现原理

1.服务提供者（Provider）：暴露服务的服务提供方，服务提供者在启动时，向注册中心注册自己提供的服务。

2.服务消费者（Consumer）: 调用远程服务的服务消费方，服务消费者在启动时，向注册中心订阅自己所需的服务，服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。

注册中心（Registry）：注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者

监控中心（Monitor）：服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心

### Dubbo环境搭建

模块划分：

mayikt\_member\_provider\_api --- 会员服务提供公共api

mayikt-member-provider-impl--- 会员服务提供公共实现

mayikt-member-impl-Consumer---订单服务消费者

#### API接口

|  |
| --- |
| **public interface** UserService {  String getUserName(Long userId); } |

#### 生产者

|  |
| --- |
| **public class** UserServiceImpl **implements** UserService {  **public** String getUserName(Long userId) {  **return "mayikt"**;  } } |

Xml：

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"  xsi:schemaLocation="http://www.springframework.org/schema/beans  http://www.springframework.org/schema/beans/spring-beans.xsd  http://code.alibabatech.com/schema/dubbo  http://code.alibabatech.com/schema/dubbo/dubbo.xsd"**>  *<!--定义了提供方应用信息，用于计算依赖关系；在 dubbo-admin 或 dubbo-monitor 会显示这个名字，方便辨识 -->* <**dubbo:application name="mayiktp"**/>  *<!--使用 zookeeper 注册中心暴露服务，注意要先开启 zookeeper -->* <**dubbo:registry address="zookeeper://localhost:2181"**/>  *<!-- 用dubbo协议在20880端口暴露服务 -->* <**dubbo:protocol name="dubbo" port="20880"**/>  *<!--使用 dubbo 协议实现定义好的 api.PermissionService 接口 -->* <**dubbo:service interface="com.mayikt.api.service.UserService"  ref="userService" protocol="dubbo"**/>  *<!--具体实现该接口的 bean -->* <**bean id="userService" class="com.mayikt.api.service.impl.UserServiceImpl"**/> </**beans**> |

启动生产者

|  |
| --- |
| **public class** Provider {   **public static void** main(String[] args) **throws** Exception {  System.*setProperty*(**"java.net.preferIPv4Stack"**, **"true"**);  ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext(**new** String[]{**"provider.xml"**});  context.start();  System.***out***.println(**"Provider started."**);  System.***in***.read(); *// press any key to exit* } } |

#### 消费者

|  |
| --- |
| **public class** Consumer {  **public static void** main(String[] args) **throws** Exception {  ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext(**new** String[]{**"consumer.xml"**});  context.start();  *//获取远程服务代理* UserService userService = (UserService) context.getBean(**"userService"**);  *// 执行远程方法* String userName = userService.getUserName(1L);  *// 现实调用结果* System.***out***.println(userName);  } } |

|  |
| --- |
| *<?***xml version="1.0" encoding="UTF-8"***?>* <**beans xmlns="http://www.springframework.org/schema/beans"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"  xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd  http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd"**>  <**dubbo:application name="order-consumer"**/>  *<!--向 zookeeper 订阅 provider 的地址，由 zookeeper 定时推送 -->* <**dubbo:registry address="zookeeper://localhost:2181"**/>  *<!--使用 dubbo 协议调用定义好的 api.PermissionService 接口 -->* <**dubbo:reference id="userService"  interface="com.mayikt.api.service.UserService"**/> </**beans**> |

### Dubbo支持那些协议

Dubbo支持dubbo、rmi、hessian、http、webservice、thrift、redis等多种协议，但是Dubbo官网是推荐我们使用Dubbo协议的。

## DubboAdmin部署流程

将dubbo-admin放入到tomcat webapps目录下 运行 修改dubbo.properties zk连接地址即可。

## 手写Dubbo核心思想与代码

mayikt-netty-dubbo-server --- dubbo核心的服务

netty-dubbo-common ---工具类

member-service-producer-api – 会员服务接口

member-service-producer-impl –会员服务接口实现

order-consumer-impl –订单消费者

### 服务注册

#### 将节点信息注册到zk上

|  |
| --- |
| **public interface** ServiceRegistration {   */\*\*  \** ***@param serviceName*** *com.mayikt.UserService  \** ***@param serviceAddres*** *mayikt://  \*/* **void** registr(String serviceName, String serviceAddres); } |

|  |
| --- |
| **public class** ServiceRegistrationImpl **implements** ServiceRegistration {  */\*\*  \* zk连接地址  \*/* **private final** String **zkServers** = **"127.0.0.1"**;  */\*\*  \* 会话时间  \*/* **private final int connectionTimeout** = 5000;  */\*\*\*  \* zkClient  \*/* **private** ZkClient **zkClient**;    **private** String **rootNamePath** = **"/mayikt\_rpc"**;   **public** ServiceRegistrationImpl() {  *// 1. 连接zk连接* **zkClient** = **new** ZkClient(**zkServers**, **connectionTimeout**);  }    **public void** registr(String serviceName, String serviceAddres) {  *// 创建我们的根路径 mayikt\_rpc* **if** (!**zkClient**.exists(**rootNamePath**)) {  **zkClient**.createPersistent(**rootNamePath**);  }  *// 创建我们的接口路径 /mayikt\_rpc/com.mayikt.UserService* String serviceNodePath = **rootNamePath** + **"/"** + serviceName;  **if** (!**zkClient**.exists(serviceNodePath)) {  **zkClient**.createPersistent(serviceNodePath);  }  *// 创建我们服务地址目录 /mayikt\_rpc/com.mayikt.UserService+"/providers"* String providerNodePath = serviceNodePath + **"/"** + **"providers"**;  **if** (!**zkClient**.exists(providerNodePath)) {  **zkClient**.createPersistent(providerNodePath);  }  *//创建我们服务地址 mayikt://192.168.11.11:8080/com.mayikt.sercice.UserSercice getUser* String serviceAddresNodePath = providerNodePath + **"/"** + URLEncoder.*encode*(serviceAddres);  **if** (**zkClient**.exists(serviceAddresNodePath)) {  **zkClient**.delete(serviceAddresNodePath);  }  **zkClient**.createEphemeral(serviceAddresNodePath);  } } |

#### 基于Netty启动我们的服务

##### MayiktRpcServer

|  |
| --- |
| **public class** MayiktRpcServer {   */\*\*  \* 存放注册的bean对象  \*/* **private** Map<String, Object> **serviceBean** = **new** HashMap<>();  **private** ServiceRegistration **serviceRegistration**;  */\*\*  \* 服务注册端口号  \*/* **private int port**;  */\*\*  \* 服务地址  \*/* **private** String **host**;   **public** MayiktRpcServer() {   }   **public** MayiktRpcServer(String host, **int** port) {  **this**.**host** = host;  **this**.**port** = port;  **serviceRegistration** = **new** ServiceRegistrationImpl();   }   */\*\*  \* 启动Netty服务器  \*/* **public void** start(Object object) {   *// 1.先将我们的服务注册到zk* bind(object);  *// 2.启动我们的netty* nettyStart();  }   **private void** bind(Object object) {  *// 获取需要发布的接口的注解class类* RpcAnnotation declaredAnnotation = object.getClass().getDeclaredAnnotation(RpcAnnotation.**class**);  **if** (declaredAnnotation == **null**) {  **return**;  }  Class value = declaredAnnotation.value();  String serviceName = value.toString().replace(**"interface "**, **""**);  String serviceAddres = **"mayikt://"** + **host** + **":"** + **port** + **"/"**;  **serviceRegistration**.registr(serviceName, serviceAddres);  System.***out***.println(**"serviceName:"** + serviceName + **",serviceAddres:"** + serviceAddres);  **serviceBean**.put(serviceName, object);  }   **private void** nettyStart() {  *// 用于接受客户端连接的请求 （并没有处理请求）* NioEventLoopGroup bossGroup = **new** NioEventLoopGroup();  *// 用于处理客户端连接的读写操作* NioEventLoopGroup workGroup = **new** NioEventLoopGroup();  *// 用于创建我们的ServerBootstrap* ServerBootstrap serverBootstrap = **new** ServerBootstrap();  serverBootstrap.group(bossGroup, workGroup).channel(NioServerSocketChannel.**class**)  .childHandler(**new** ChannelInitializer<SocketChannel>() {  @Override  **protected void** initChannel(SocketChannel socketChannel) **throws** Exception {  socketChannel.pipeline().addLast(MarshallingCodeCFactory.*buildMarshallingDecoder*());  socketChannel.pipeline().addLast(MarshallingCodeCFactory.*buildMarshallingEncoder*());  socketChannel.pipeline().addLast(**new** DubboServerHandler(**serviceBean**));  }  });  ;  *// 绑定我们的端口号码* **try** {  *// 绑定端口号，同步等待成功* ChannelFuture future = serverBootstrap.bind(**port**).sync();  System.***out***.println(**"会员服务启动成功:"** + **port**);  *// 等待服务器监听端口* future.channel().closeFuture().sync();  } **catch** (Exception e) {  e.printStackTrace();   } **finally** {  *// 优雅的关闭连接* bossGroup.shutdownGracefully();  workGroup.shutdownGracefully();  }   }   } |

##### DubboServerHandler

|  |
| --- |
| **public class** DubboServerHandler **extends** ChannelInboundHandlerAdapter {  */\*\*  \* 存放注册的bean对象  \*/* **private** Map<String, Object> **serviceBean** = **new** HashMap<>();   **public** DubboServerHandler(Map<String, Object> serviceBean) {  **this**.**serviceBean** = serviceBean;   }   @Override  **public void** channelRead(ChannelHandlerContext ctx, Object msg) **throws** Exception {  RpcRequest rpcRequest = (RpcRequest) msg;  String className = rpcRequest.getClassName();  Object objectImpl = **serviceBean**.get(className);  **if** (objectImpl == **null**) {  **return**;  }  Method method = objectImpl.getClass().getMethod(rpcRequest.getMethodName(), rpcRequest.getParameterTypes());  *// 使用反射技术执行我们的方法* Object result = method.invoke(objectImpl, rpcRequest.getParamsValue());  *// 响应给客户端* ctx.writeAndFlush(result);  } } |

##### MayiktRpcServer

|  |
| --- |
| **public class** RpcRequest **implements** Serializable {  **private static final long *SerialVersionUID*** = 1L;  */\*\*  \* 类的className  \*/* **private** String **className**;  */\*\*  \* 方法名称  \*/* **private** String **methodName**;  */\*\*  \* 参数类型  \*/* Class<?> **parameterTypes**[];  */\*\*  \* 参数value  \*/* Object **paramsValue**[];   **public** RpcRequest(String className, String methodName, Class<?>[] parameterTypes, Object[] paramsValue) {  **this**.**className** = className;  **this**.**methodName** = methodName;  **this**.**parameterTypes** = parameterTypes;  **this**.**paramsValue** = paramsValue;  }   **public** String getClassName() {  **return className**;  }   **public** String getMethodName() {  **return methodName**;  }   **public** Class<?>[] getParameterTypes() {  **return parameterTypes**;  }   **public** Object[] getParamsValue() {  **return paramsValue**;  }   **public void** setClassName(String className) {  **this**.**className** = className;  }   **public void** setMethodName(String methodName) {  **this**.**methodName** = methodName;  }   **public void** setParameterTypes(Class<?>[] parameterTypes) {  **this**.**parameterTypes** = parameterTypes;  }   **public void** setParamsValue(Object[] paramsValue) {  **this**.**paramsValue** = paramsValue;  }  } |

### 服务消费

#### 从注册中心上获取地址

|  |
| --- |
| **public interface** ServiceDiscover {   */\*\*  \* 根据服务名称查找对应服务列表  \*  \** ***@param serviceName*** *\** ***@return*** *\*/* List<String> getDiscover(String serviceName);   }  **public class** ServiceDiscoverImpl **implements** ServiceDiscover {  */\*\*  \* zk连接地址  \*/* **private final** String **zkServers** = **"127.0.0.1"**;  */\*\*  \* 会话时间  \*/* **private final int connectionTimeout** = 5000;  */\*\*\*  \* zkClient  \*/* **private** ZkClient **zkClient**;    **private** String **rootNamePath** = **"/mayikt\_rpc"**;   **public** ServiceDiscoverImpl() {  **zkClient** = **new** ZkClient(**zkServers**, **connectionTimeout**);  }   @Override  **public** List<String> getDiscover(String serviceName) {  String serviceNameNodePath = **rootNamePath** + **"/"** + serviceName + **"/providers"**;  List<String> children = **zkClient**.getChildren(serviceNameNodePath);  **return** children;  } } |

#### 负载均衡器

|  |
| --- |
| **public interface** RouteStrategy<T> {   */\*\*  \* 实现Dubbo负载均衡器 负载均衡、随机、一致性hash  \*  \** ***@param repos*** *\** ***@return*** *\*/* String select(List<T> repos); }  **public class** RandomLoadBalance **implements** RouteStrategy<String> {  @Override  **public** String select(List<String> repos) {  String value = repos.get(**new** Random().nextInt(repos.size()));  System.***out***.println(value);  **return** value;  } }  **public class** LoadBalancing **implements** RouteStrategy<String> {  **private int index** = 0;   @Override  **public synchronized** String select(List<String> repos) {  **int** size = repos.size();  **if** (**index** >= size) {  **index** = 0;  }  String value = repos.get(**index**++);  System.***out***.println(**"value:"** + value);  **return** value;  }   } |

#### 远程代理

|  |
| --- |
| **public class** RpcClientProxy {  **private** ServiceDiscover **serviceDiscover**;   **public** RpcClientProxy() {  **serviceDiscover** = **new** ServiceDiscoverImpl();  }   **public** <T> T create(Class<T> interfaceClass) {  **return** (T) Proxy.*newProxyInstance*(interfaceClass.getClassLoader(), **new** Class[]{interfaceClass}, **new** InvocationHandler() {  @Override  **public** Object invoke(Object proxy, Method method, Object[] args) **throws** Throwable {  *// 从zk上获取注册地址* String serviceName = interfaceClass.getName();  List<String> discover = **serviceDiscover**.getDiscover(serviceName);  *// 使用默认负载均衡器* RouteStrategy<String> loadBalancing = **new** LoadBalancing();  *// mayikt://192.168.212.1:8080 获取ip和端口号* String selectAddres = URLDecoder.*decode*(loadBalancing.select(discover));  String[] split = selectAddres.split(**":"**);  String host = split[1].replace(**"//"**, **""**);  String port = split[2].replace(**"/"**, **""**);  *// 建立Netty连接 发送数据* RpcRequest rpcRequest = **new** RpcRequest(serviceName, method.getName(), method.getParameterTypes(), args); *// return sendMsg(host, Integer.parseInt(port), rpcRequest);* DubboClientHandler dubboClientHandler = **new** DubboClientHandler();  *//创建nioEventLoopGroup* NioEventLoopGroup group = **new** NioEventLoopGroup();  Bootstrap bootstrap = **new** Bootstrap();  bootstrap.group(group).channel(NioSocketChannel.**class**)  .remoteAddress(**new** InetSocketAddress(host, Integer.*parseInt*(port)))  .handler(**new** ChannelInitializer<SocketChannel>() {  @Override  **protected void** initChannel(SocketChannel ch) **throws** Exception {  ch.pipeline().addLast(MarshallingCodeCFactory.*buildMarshallingDecoder*());  ch.pipeline().addLast(MarshallingCodeCFactory.*buildMarshallingEncoder*());  ch.pipeline().addLast(dubboClientHandler);  }  });  **try** {  *// 发起同步连接* ChannelFuture sync = bootstrap.connect().sync();  sync.channel().writeAndFlush(rpcRequest);  sync.channel().closeFuture().sync();  } **catch** (Exception e) {   } **finally** {  group.shutdownGracefully();  }  **return** dubboClientHandler.getResponse();   }  });  }   }  **public class** DubboClientHandler **extends** ChannelInboundHandlerAdapter {   **public** Object **response**;   **public** Object getResponse() {  **return response**;  }   @Override  **public void** channelRead(ChannelHandlerContext ctx, Object msg) **throws** Exception {  System.***out***.println(**"msg:"** + msg);  **this**.**response** = msg;  ctx.close();  } } |

### 使用marshalling实现编码器

|  |
| --- |
| **public final class** MarshallingCodeCFactory {   */\*\*  \* 创建Jboss Marshalling解码器MarshallingDecoder  \** ***@return*** *MarshallingDecoder  \*/* **public static** MarshallingDecoder buildMarshallingDecoder() {  *//首先通过Marshalling工具类的精通方法获取Marshalling实例对象 参数serial标识创建的是java序列化工厂对象。* **final** MarshallerFactory marshallerFactory = Marshalling.*getProvidedMarshallerFactory*(**"serial"**);  *//创建了MarshallingConfiguration对象，配置了版本号为5* **final** MarshallingConfiguration configuration = **new** MarshallingConfiguration();  configuration.setVersion(5);  *//根据marshallerFactory和configuration创建provider* UnmarshallerProvider provider = **new** DefaultUnmarshallerProvider(marshallerFactory, configuration);  *//构建Netty的MarshallingDecoder对象，俩个参数分别为provider和单个消息序列化后的最大长度* MarshallingDecoder decoder = **new** MarshallingDecoder(provider, 1024 \* 1024 \* 1);  **return** decoder;  }   */\*\*  \* 创建Jboss Marshalling编码器MarshallingEncoder  \** ***@return*** *MarshallingEncoder  \*/* **public static** MarshallingEncoder buildMarshallingEncoder() {  **final** MarshallerFactory marshallerFactory = Marshalling.*getProvidedMarshallerFactory*(**"serial"**);  **final** MarshallingConfiguration configuration = **new** MarshallingConfiguration();  configuration.setVersion(5);  MarshallerProvider provider = **new** DefaultMarshallerProvider(marshallerFactory, configuration);  *//构建Netty的MarshallingEncoder对象，MarshallingEncoder用于实现序列化接口的POJO对象序列化为二进制数组* MarshallingEncoder encoder = **new** MarshallingEncoder(provider);  **return** encoder;  }  } |

### Maven相关依赖

|  |
| --- |
| <**dependencies**>  *<!-- - ZK客户端工具 -->* <**dependency**>  <**groupId**>com.101tec</**groupId**>  <**artifactId**>zkclient</**artifactId**>  <**version**>0.10</**version**>  </**dependency**>  <**dependency**>  <**artifactId**>netty-dubbo-common</**artifactId**>  <**groupId**>com.mayikt</**groupId**>  <**version**>1.0-SNAPSHOT</**version**>  </**dependency**>  <**dependency**>  <**groupId**>io.netty</**groupId**>  <**artifactId**>netty-all</**artifactId**>  <**version**>4.1.42.Final</**version**>  </**dependency**>  *<!-- https://mvnrepository.com/artifact/com.alibaba/fastjson -->* <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>fastjson</**artifactId**>  <**version**>1.2.9</**version**>  </**dependency**>   <**dependency**>  <**groupId**>org.jboss.marshalling</**groupId**>  <**artifactId**>jboss-marshalling</**artifactId**>  <**version**>1.4.10.Final</**version**>  </**dependency**>  <**dependency**>  <**groupId**>org.jboss.marshalling</**groupId**>  <**artifactId**>jboss-marshalling-serial</**artifactId**>  <**version**>1.4.10.Final</**version**>  </**dependency**>    </**dependencies**> |

## 负载均衡器

### 负载均衡器在RPC调用中实现的作用

### 五种RPC框架负载均衡器的种类

### 基于策略模式设计负载均衡器

|  |
| --- |
| **public interface** LoadBalancer<T> {   */\*\*  \* 根据多个地址实现负载均衡轮训、权重、随机、一致性hash  \*  \** ***@param addres*** *\** ***@return*** *\*/* String getSingleAddres(List<T> addres); } |

#### 轮训机制

|  |
| --- |
| **public class** RotationLoadBalancer **implements** LoadBalancer<String> {  **private int index**;  **private** AtomicLong **atomicLong** = **new** AtomicLong(0);   */\*\*  \* 轮训机制 算法1  \*  \** ***@param addres*** *\** ***@return*** *\*/ // public synchronized String getSingleAddres(List<String> addres) { // if (index >= addres.size()) { // index = 0; // } // String ads = addres.get(index++); // System.out.println("ads:" + ads); // return ads; // }* **public** String getSingleAddres(List<String> addres) {  Long index = **atomicLong**.incrementAndGet() % 2;  String ads = addres.get(Integer.*parseInt*(index + **""**));  System.***out***.println(**"ads:"** + ads);  **return** ads;  } } |

#### 轮训权重

|  |
| --- |
| **public class** WeightLoadBalancer **implements** LoadBalancer<WeightEntity> {   **private** List<String> **addresList** = **new** ArrayList<String>();  **private** AtomicLong **atomicLong** = **new** AtomicLong(0);   **private int index**;   **public synchronized** String getSingleAddres(List<WeightEntity> addres) {  **if** (**addresList**.size() > 0) {  *// 对权重数据实现重新排序* **for** (**int** i = 0; i < addres.size(); i++) {  WeightEntity weightEntity = addres.get(i);  **int** weight = weightEntity.getWeight();  **for** (**int** j = 0; j < weight; j++) {  **addresList**.add(addres.get(i).getAddres());  }  }  }  **if** (**index** >= **addresList**.size()) {  **index** = 0;  }  String ads = **addresList**.get(Integer.*parseInt*(**index**++ + **""**));  System.***out***.println(**"ads:"** + ads);  **return** ads;  } } |

#### 一致性hash

|  |
| --- |
| **public class** IpHashLoadBalancer {   **public** String getSingleAddres(List<String> addres, String ipaddres) {  **int** index = ipaddres.hashCode() % addres.size();  String value = addres.get(index);  System.***out***.println(**"value:"** + value);  **return** value;  } } |

### 手写五种负载均衡器算法

RPC中负载均衡器有那些作用？

Rpc是远程调用技术 dubbo、feign、httpclient

本地负载均衡器任何 rpc框架都必须有本地负载均衡 ribbon、自带负载均衡器

负载均衡器在rpc作用？  
目的能够提高我们生产者服务器集群访问，分摊压力。

原理：从一个集合列表地址元素中，采用负载均衡器算法最终获取一个地址调用。

负载均衡算法：轮训、权重、一致性hash、随机权重、随机