# 基于Netty网络编程项目实战课程

## 项目介绍

## Netty介绍与相关基础知识

### Netty介绍

**简介**

*Netty是由JBOSS提供的一个java开源框架。Netty提供异步的、事件驱动的网络应用程序框架和工具，用以快速开发高性能、高可靠性的网络服务器和客户端程序。*

*也就是说，Netty 是一个基于NIO的客户、服务器端编程框架，使用Netty 可以确保你快速和简单的开发出一个网络应用，例如实现了某种协议的客户、服务端应用。Netty相当于简化和流线化了网络应用的编程开发过程，例如：基于TCP和UDP的socket服务开发。*

*“快速”和“简单”并不用产生维护性或性能上的问题。Netty 是一个吸收了多种协议（包括FTP、SMTP、HTTP等各种二进制文本协议）的实现经验，并经过相当精心设计的项目。最终，Netty 成功的找到了一种方式，在保证易于开发的同时还保证了其应用的性能，稳定性和伸缩性。*

1. Netty提供了简单易用的API
2. 基于事件驱动的编程方式来编写网络通信程序
3. 更高的吞吐量
4. 学习难度低

应用场景：

JavaEE： Dubbo

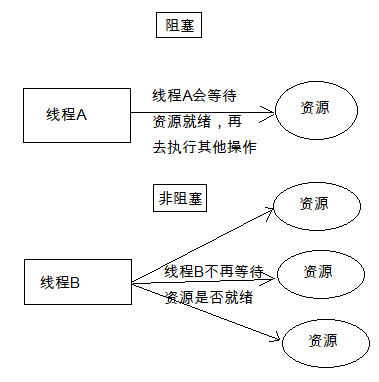
大数据：Apache Storm（Supervisor worker进程间的通信也是基于Netty来实现的）

### BIO、NIO、AIO介绍与区别

**阻塞与非阻塞**

*主要指的是访问IO的线程是否会阻塞（或者说是等待）*

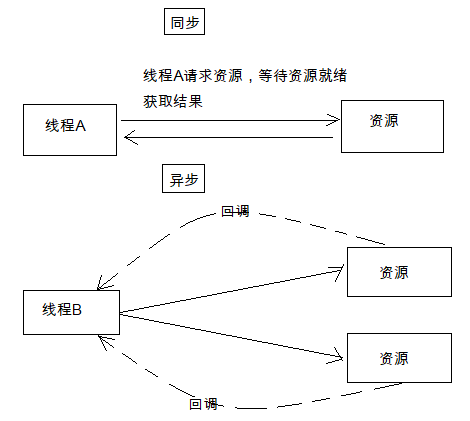
*线程访问资源，该资源是否准备就绪的一种处理方式。*



**同步和异步**

*主要是指的数据的请求方式*

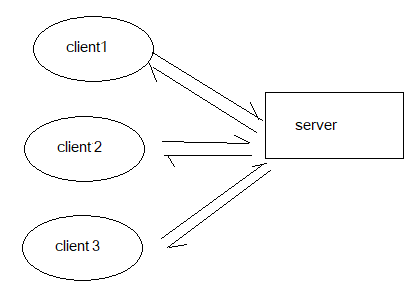
*同步和异步是指访问数据的一种机制*



**BIO**

*同步阻塞IO，Block IO，IO操作时会阻塞线程，并发处理能力低。*

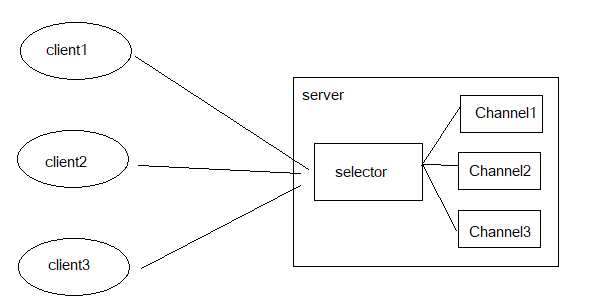
*我们熟知的Socket编程就是BIO，一个socket连接一个处理线程（这个线程负责这个Socket连接的一系列数据传输操作）。阻塞的原因在于：操作系统允许的线程数量是有限的，多个socket申请与服务端建立连接时，服务端不能提供相应数量的处理线程，没有分配到处理线程的连接就会阻塞等待或被拒绝。*



**NIO**

*同步非阻塞IO，None-Block IO*

*NIO是对BIO的改进，基于Reactor模型。我们知道，一个socket连接只有在特点时候才会发生数据传输IO操作，大部分时间这个“数据通道”是空闲的，但还是占用着线程。NIO作出的改进就是“一个请求一个线程”，在连接到服务端的众多socket中，只有需要进行IO操作的才能获取服务端的处理线程进行IO。这样就不会因为线程不够用而限制了socket的接入。*



**AIO（NIO 2.0）**

*异步非阻塞IO*

*这种IO模型是由操作系统先完成了客户端请求处理再通知服务器去启动线程进行处理。AIO也称NIO2.0，在JDK7开始支持。*

### Netty Reactor模型 - 单线程模型、多线程模型、主从多线程模型介绍

#### 单线程模型

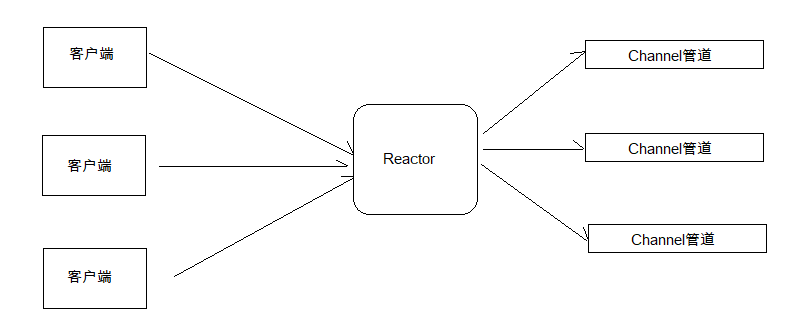
*用户发起IO请求到Reactor线程*

*Ractor线程将用户的IO请求放入到通道，然后再进行后续处理*

*处理完成后，Reactor线程重新获得控制权，继续其他客户端的处理*

***这种模型一个时间点只有一个任务在执行，这个任务执行完了，再去执行下一个任务。***

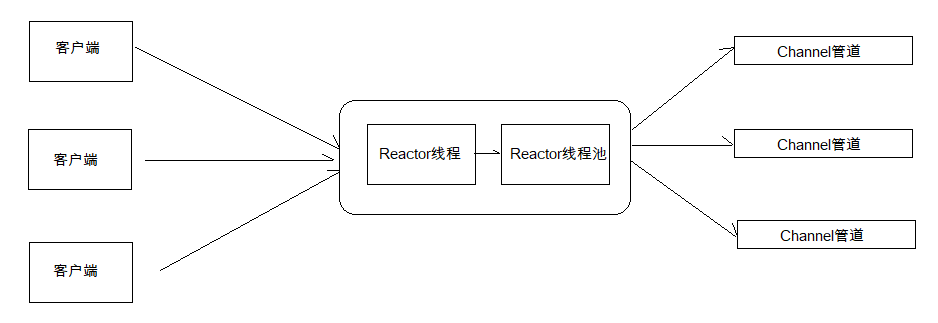
1. *但单线程的Reactor模型每一个用户事件都在一个线程中执行：*
2. *性能有极限，不能处理成百上千的事件*
3. *当负荷达到一定程度时，性能将会下降*
4. *某一个事件处理器发生故障，不能继续处理其他事件*



#### Reactor多线程模型

*Reactor多线程模型是由一组NIO线程来处理IO操作（之前是单个线程），所以在请求处理上会比上一中模型效率更高，可以处理更多的客户端请求。*

*这种模式使用多个线程执行多个任务，任务可以同时执行*

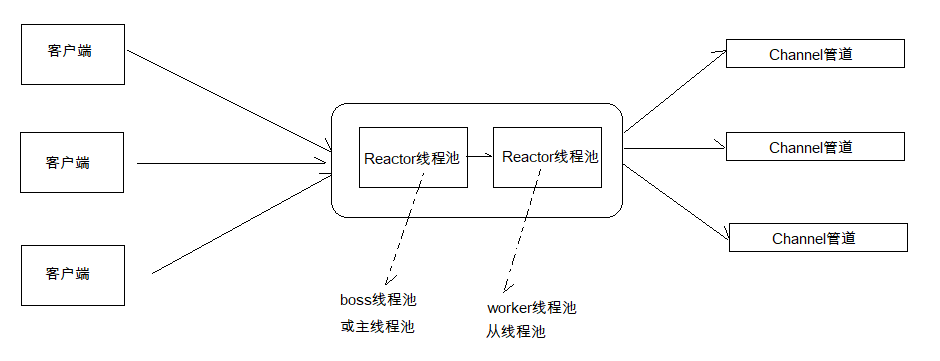


***但是如果并发仍然很大，Reactor仍然无法处理大量的客户端请求***

#### Reactor主从多线程模型

*这种线程模型是Netty推荐使用的线程模型*

*这种模型适用于高并发场景，一组线程池接收请求，一组线程池处理IO。*



### Netty - 基于web socket简单聊天DEMO实现

**后端编写**

导入依赖

*<build>*

*<plugins>*

*<plugin>*

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

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

*<version>****3.1****</version>*

*<configuration>*

*<target>****1.8****</target>*

*<source>****1.8****</source>*

*</configuration>*

*</plugin>*

*</plugins>*

*</build>*

*<dependencies>*

*<dependency>*

*<groupId>****io.netty****</groupId>*

*<artifactId>****netty-all****</artifactId>*

*<version>****4.1.15.Final****</version>*

*</dependency>*

*</dependencies>*

编写Netty Server

***public class*** *WebsocketServer {* ***public static void*** *main(String[] args)* ***throws*** *InterruptedException {  
 // 初始化主线程池（boss线程池）  
 NioEventLoopGroup mainGroup =* ***new*** *NioEventLoopGroup();  
 // 初始化从线程池(worker线程池)  
 NioEventLoopGroup subGroup =* ***new*** *NioEventLoopGroup();* ***try*** *{  
 // 创建服务器启动器  
 ServerBootstrap b =* ***new*** *ServerBootstrap();  
  
 // 指定使用主线程池和从线程池  
 b.group(mainGroup, subGroup)  
 // 指定使用Nio通道类型  
 .channel(NioServerSocketChannel.****class****)  
 // 指定通道初始化器加载通道处理器  
 .childHandler(****new*** *WsServerInitializer());  
  
 // 绑定端口号启动服务器，并等待服务器启动  
 // ChannelFuture是Netty的回调消息  
 ChannelFuture future = b.bind(9090).sync();  
 // 等待服务器socket关闭  
 future.channel().closeFuture().sync();  
 }* ***finally*** *{  
 // 优雅关闭boos线程池和worker线程池  
 mainGroup.shutdownGracefully();  
 subGroup.shutdownGracefully();  
 }  
 }  
}*

编写通道初始化器

***public class*** *WsServerInitializer* ***extends*** *ChannelInitializer<SocketChannel> {* ***@Override  
 protected void*** *initChannel(SocketChannel ch)* ***throws*** *Exception {  
 ChannelPipeline pipeline = ch.pipeline();  
  
 // ------------------  
 // 用于支持Http协议  
  
 // websocket基于http协议，需要有http的编解码器  
 pipeline.addLast(****new*** *HttpServerCodec());  
 // 对写大数据流的支持  
 pipeline.addLast(****new*** *ChunkedWriteHandler());  
 // 添加对HTTP请求和响应的聚合器:只要使用Netty进行Http编程都需要使用  
 // 对HttpMessage进行聚合，聚合成FullHttpRequest或者FullHttpResponse  
 // 在netty编程中都会使用到Handler  
 pipeline.addLast(****new*** *HttpObjectAggregator(1024 \* 64));  
  
  
 // ---------支持Web Socket -----------------  
  
 // websocket服务器处理的协议，用于指定给客户端连接访问的路由: /ws  
 // 本handler会帮你处理一些握手动作: handshaking(close, ping, pong) ping + pong = 心跳  
 // 对于websocket来讲，都是以frames进行传输的，不同的数据类型对应的frames也不同  
 pipeline.addLast(****new*** *WebSocketServerProtocolHandler(****"/ws"****));  
  
 // 添加自定义的handler  
 pipeline.addLast(****new*** *ChatHandler());  
 }  
}*

编写处理消息的ChannelHandler

/\*\*  
 \* 处理消息的handler  
 \* TextWebSocketFrame: 在netty中，是用于为websocket专门处理文本的对象，frame是消息的载体  
 \*/  
**public class** ChatHandler **extends** SimpleChannelInboundHandler<TextWebSocketFrame> {  
  
 // 用于记录和管理所有客户端的Channel  
 **private static** ChannelGroup clients = **new** DefaultChannelGroup(GlobalEventExecutor.**INSTANCE**);  
  
 **@Override  
 protected void** channelRead0(ChannelHandlerContext ctx, TextWebSocketFrame msg) **throws** Exception {  
 // 获取从客户端传输过来的消息  
 String text = msg.text();  
 System.**out**.println(**"接收到的数据:"** + text);  
  
 // 将接收到消息发送到所有客户端  
 **for**(Channel channel : clients) {  
 // 注意所有的websocket数据都应该以TextWebSocketFrame进行封装  
 channel.writeAndFlush(**new** TextWebSocketFrame(**"[服务器接收到消息:]"** + LocalDateTime.now() + **",消息为:"** + text));  
 }  
 }  
  
 /\*\*  
 \* 当客户端连接服务端之后（打开连接）  
 \* 获取客户端的channel，并且放入到ChannelGroup中去进行管理  
 \* **@param ctx** \* **@throws** Exception  
 \*/  
 **@Override  
 public void** handlerAdded(ChannelHandlerContext ctx) **throws** Exception {  
 // 将channel添加到客户端  
 clients.add(ctx.channel());  
 }  
  
 **@Override  
 public void** handlerRemoved(ChannelHandlerContext ctx) **throws** Exception {  
 // 当触发handlerRemoved，ChannelGroup会自动移除对应客户端的channel  
 //clients.remove(ctx.channel());  
  
 // asLongText()——唯一的ID  
 // asShortText()——短ID（有可能会重复)  
 System.**out**.println(**"客户端断开, channel对应的长id为:"** + ctx.channel().id().asLongText());  
 System.**out**.println(**"客户端断开, channel对应的短id为:"** + ctx.channel().id().asShortText());  
 }  
}

### websocket以及前端代码编写

WebSocket protocol 是HTML5一种新的协议。它实现了浏览器与服务器全双工通信(full-duplex)。一开始的握手需要借助HTTP请求完成。Websocket是应用层第七层上的一个应用层协议，它必须依赖 HTTP 协议进行一次握手，握手成功后，数据就直接从 TCP 通道传输，与 HTTP 无关了。

**前端编写**

*<!DOCTYPE* ***html****>  
<****html****>  
 <****head****>  
 <****meta charset="UTF-8"****>  
 <****title****></****title****>  
 </****head****>  
 <****body****>  
 <****div****>发送消息</****div****>  
 <****input type="text" id="msgContent"*** */>  
 <****input type="button" value="点击发送" onclick="CHAT****.chat()****"****/>  
   
 <****div****>接收消息:</****div****>  
 <****div id="recMsg" style="background-color****:* ***gainsboro****;****"****></****div****>  
   
 <****script type="application/javascript"****>* ***window****.****CHAT*** *= {* ***socket****:* ***null****,  
 init:* ***function****() {  
 // 判断浏览器是否支持websocket* ***if****(****window****.WebSocket) {  
 // 支持WebScoekt  
 // 连接创建socket，注意要添加ws后缀* ***CHAT****.****socket*** *=* ***new*** *WebSocket(****"ws://127.0.0.1:9001/ws"****);* ***CHAT****.****socket****.onopen =* ***function****() {* ***console****.log(****"连接建立成功"****);  
 };* ***CHAT****.****socket****.onclose =* ***function****() {* ***console****.log(****"连接关闭"****)  
 };* ***CHAT****.****socket****.onerror =* ***function****() {* ***console****.log(****"发生错误"****);  
 };* ***CHAT****.****socket****.onmessage =* ***function****(e) {* ***console****.log(****"接收到消息:"*** *+ e.****data****);* ***var*** *recMsg =* ***document****.getElementById(****"recMsg"****);* ***var*** *html = recMsg.****innerHTML****;  
 recMsg.****innerHTML*** *= html +* ***"<br/>"*** *+ e.****data****;  
 };  
   
 }* ***else*** *{  
 alert(****"浏览器不支持websocket协议"****);  
 }  
 },  
 chat:* ***function****() {* ***var*** *msg =* ***document****.getElementById(****"msgContent"****);* ***CHAT****.****socket****.send(msg.****value****);  
 }  
 }* ***CHAT****.init();  
 </****script****>  
 </****body****>  
</****html****>*

### MUI、HTML5+、HBuilder介绍

**MUI介绍**

[***http://dev.dcloud.net.cn/mui/***](http://dev.dcloud.net.cn/mui/)

*MUI是一个轻量级的前端框架。MUI以iOS平台UI为基础，补充部分Android平台特有的UI控件。MUI不依赖任何第三方JS库，压缩后的JS和CSS文件仅有100+K和60+K，可以根据自己的需要，自定义去下载对应的模块。并且MUI编写的前端，可以打包成APK和IPA安装文件，在手机端运行。也就是，编写一套代码，就可以在Android、IOS下运行。*

*API地址：*[*http://dev.dcloud.net.cn/mui/ui/*](http://dev.dcloud.net.cn/mui/ui/)

**H5+**

*H5+提供了对HTML5的增强，提供了40WAPI给程序员使用。使用H5+ API可以轻松开发二维码扫描、摄像头、地图位置、消息推送等功能*

**

*API地址：*[*http://www.html5plus.org/doc/zh\_cn/accelerometer.html#*](http://www.html5plus.org/doc/zh_cn/accelerometer.html)

**HBuilder**

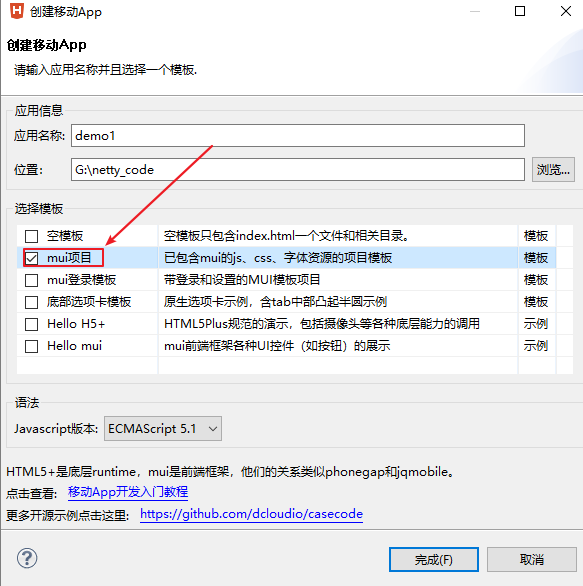
*前端开发工具。本次项目所有的前端使用HBuilder开发。在项目开发完后，也会使用HBuilder来进行打包Android/IOS的安装包。*

<http://www.dcloud.io/>

### MUI前端开发

#### 创建项目/页面/添加MUI元素

创建MUI移动App项目



页面创建，添加组件

*<header class="mui-bar mui-bar-nav">*

*<h1 class="mui-title">登录页面</h1>*

*</header>*

*<div class="mui-content">*

*<form class="mui-input-group">*

*<div class="mui-input-row">*

*<label>用户名</label>*

*<input type="text" class="mui-input-clear" placeholder="请输入用户名">*

*</div>*

*<div class="mui-input-row">*

*<label>密码</label>*

*<input type="password" class="mui-input-password" placeholder="请输入密码">*

*</div>*

*<div class="mui-button-row">*

*<button type="button" class="mui-btn mui-btn-primary">确认</button>*

*<button type="button" class="mui-btn mui-btn-danger">取消</button>*

*</div>*

*</form>*

*</div>*

<http://dev.dcloud.net.cn/mui/ui/#accordion>

#### 获取页面元素/添加点击事件

获取页面元素

*mui.plusReady(function() {*

*// 使用document.getElementById来获取Input组件数据*

*var username = document.getElementById("username");*

*var password = document.getElementById("password");*

*var confirm = document.getElementById("confirm");*

*// 绑定事件*

*confirm.addEventListener("tap", function() {*

*alert("按下按钮");*

*});*

*});*

批量绑定页面元素的点击事件

*mui(".mui-table-view").on('tap','.mui-table-view-cell',function(){*

*});*

使用原生JS的事件绑定方式

*// 绑定事件*

*confirm.addEventListener("tap", function() {*

*alert("按下按钮");*

*});*

#### 发起ajax请求

**前端**

当我们点击确认按钮的时候，将用户名和密码发送给后端服务器



*// 发送ajax请求*

*mui.ajax('http://192.168.1.106:9000/login', {*

*data: {*

*username: username.value,*

*password: password.value*

*},*

*dataType: 'json', //服务器返回json格式数据*

*type: 'post', //HTTP请求类型*

*timeout: 10000, //超时时间设置为10秒；*

*headers: {*

*'Content-Type': 'application/json'*

*},*

*success: function(data) {*

*// 可以使用console.log打印数据，一般用于调试*

*console.log(data);*

*},*

*error: function(xhr, type, errorThrown) {*

*//异常处理；*

*console.log(type);*

*}*

*});*

**后端**

基于SpringBoot编写一个web应用，主要是用于接收ajax请求，响应一些数据到前端

***@RestController  
public class*** *LoginController {* ***@RequestMapping****(****"/login"****)* ***public*** *Map login(****@RequestBody*** *User user) {  
 System.****out****.println(user);  
  
 Map map =* ***new*** *HashMap<String, Object>();* ***if****(****"tom"****.equals(user.getUsername()) &&* ***"123"****.equals(user.getPassword())) {  
 map.put(****"success"****,* ***true****);  
 map.put(****"message"****,* ***"登录成功"****);  
 }* ***else*** *{  
 map.put(****"success"****,* ***false****);  
 map.put(****"message"****,* ***"登录失败，请检查用户名和密码是否输入正确"****);  
 }* ***return*** *map;  
 }  
}*

#### 字符串转JSON对象以及JSON对象转字符串

将JSON对象转换为字符串

*// 使用JSON.stringify可以将JSON对象转换为String字符串*

*console.log(JSON.stringify(data));*

将字符串转换为JSON对象

*var jsonObj = JSON.parse(jsonStr);*

#### 页面跳转

*mui.openWindow({*

*url: 'login\_succss.html',*

*id:'login\_succss.html'*

*});*

#### App客户端缓存操作

大量的App很多时候都需要将服务器端响应的数据缓存到手机App本地。

<http://www.html5plus.org/doc/zh_cn/storage.html>

在App中缓存的数据，就是以key-value键值对来存放的。



将数据放入到本地缓存中

*var user = {*

*username: username.value,*

*password: password.value*

*}*

*// 将对象数据放入到缓存中，需要转换为字符串*

*plus.storage.setItem("user", JSON.stringify(user));*

从本地缓存中读取数据

*// 从storage本地缓存中获取对应的数据*

*var userStr = plus.storage.getItem("user");*

## 构建项目

### 项目功能需求、技术架构介绍

**功能需求**

*登录/注册*

*个人信息*

*搜索添加好友*

*好友聊天*

**技术架构**

前端

*开发工具：HBuilder*

*框架：MUI、H5+*

后端

*开发工具：IDEA*

*框架：Spring Boot、MyBatis、Spring MVC、FastDFS、Netty*

*数据库：mysql*

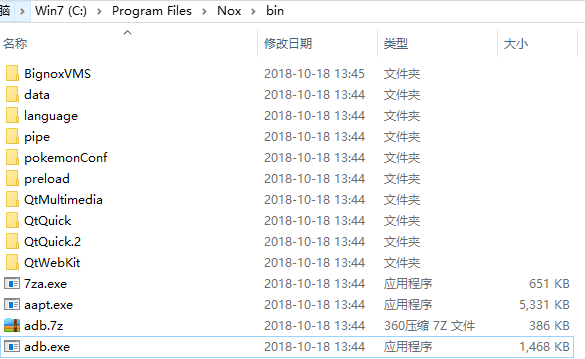
### 使用模拟器进行测试

**安装附件中的夜神Android模拟器（nox\_setup\_v6.2.3.8\_full.exe）**

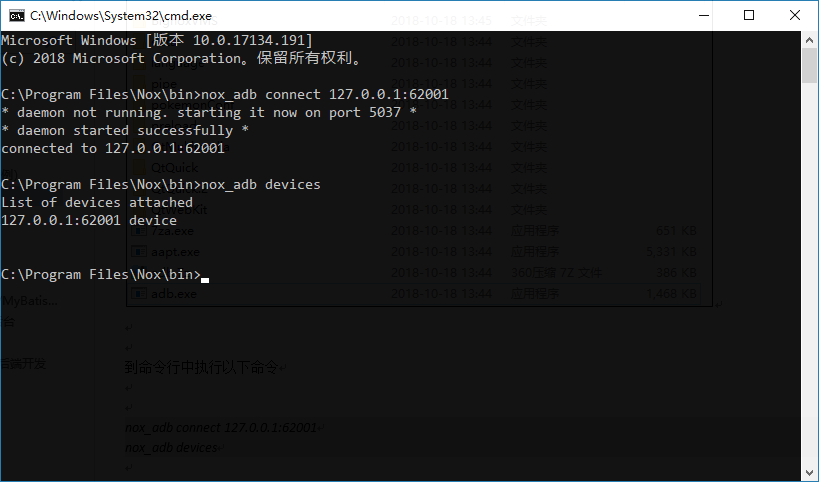
*双击桌面图标启动模拟器*

**

*安装后找到模拟器的安装目录*

**

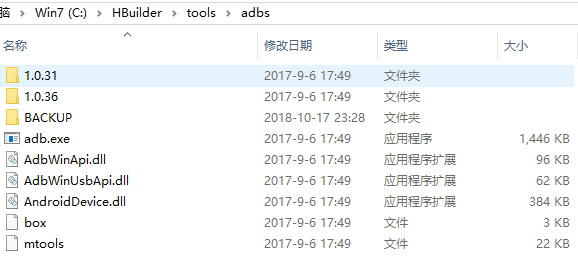
*到命令行中执行以下命令*

**

*nox\_adb connect 127.0.0.1:62001*

*nox\_adb devices*

*进入到Hbuilder安装目录下的tools/adbs目录*

**

*切换到命令行中执行以下命令*

*adb connect 127.0.0.1:62001  
adb devices*

**打开HBuilder开始调试**

**

### 前端 - HBuilder前端项目导入

*将资料中的heima-chat.zip解压，并导入到HBuilder中。*

### 后端 - 导入数据库/SpringBoot项目/MyBatis逆向工程

**导入数据库**

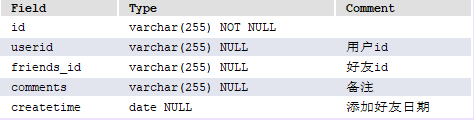
将资料中的hchat.sql脚本在开发工具中执行

**数据库表结构介绍**

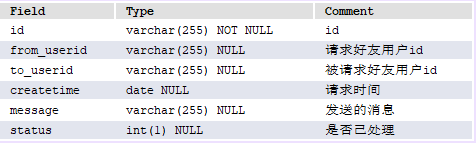
tb\_user用户表



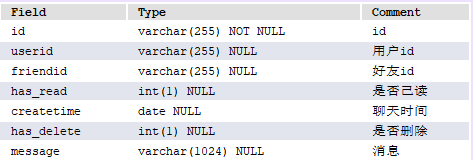
tb\_friend朋友表



tb\_friend\_req申请好友表



tb\_chat\_record聊天记录表



**使用MyBatis逆向工程生成代码**

将资料中的generatorSqlmapCustom项目导入到IDEA中，并配置项目所使用的JDK

**创建Spring Boot项目**

拷贝资料pom.xml依赖

拷贝资料中的application.properties配置文件

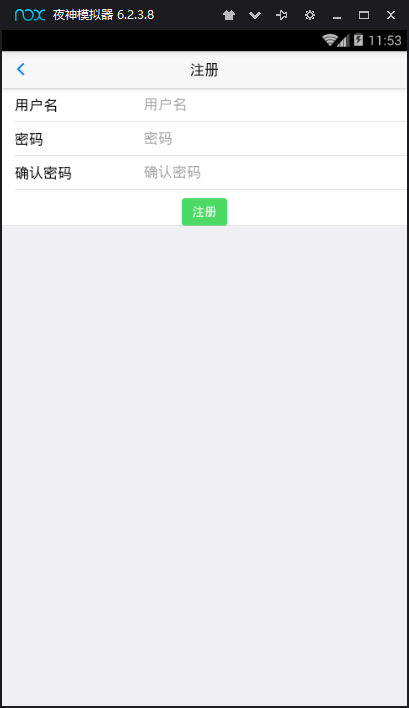
### 后端 - Spring Boot整合Netty搭建后台

**spring boot整合Netty**

导入资料中配置文件中的spring-netty文件夹中的java文件

**启动Spring Boot，导入HTML页面,使用浏览器打开测试Netty是否整合成功**

## 业务开发 - 用户注册/登录/个人信息

### 用户登录功能 -后端开发

**导入IdWorker.java雪花算法ID生成器**

**初始化IdWorker**

***@SpringBootApplication  
@MapperScan****(basePackages =* ***"com.itheima.hchat.mapper"****)****public class*** *Application {* ***public static void*** *main(String[] args) {  
 SpringApplication.run(Application.****class****);  
 }* ***@Bean  
 public*** *IdWorker idWorker() {* ***return new*** *IdWorker(0, 0);  
 }  
}*

**创建Result实体类**

*/\*\*  
 \* 将返回给客户端的数据封装到实体类中  
 \*/****public class*** *Result {* ***private boolean success****; // 是否操作成功* ***private*** *String* ***message****; // 返回消息* ***private*** *Object* ***result****; // 返回附件的对象* ***public*** *Result(****boolean*** *success, String message) {* ***this****.****success*** *= success;* ***this****.****message*** *= message;  
 }* ***public*** *Result(****boolean*** *success, String message, Object result) {* ***this****.****success*** *= success;* ***this****.****message*** *= message;* ***this****.****result*** *= result;  
 }* ***public boolean*** *isSuccess() {* ***return success****;  
 }* ***public void*** *setSuccess(****boolean*** *success) {* ***this****.****success*** *= success;  
 }* ***public*** *String getMessage() {* ***return message****;  
 }* ***public void*** *setMessage(String message) {* ***this****.****message*** *= message;  
 }* ***public*** *Object getResult() {* ***return result****;  
 }* ***public void*** *setResult(Object result) {* ***this****.****result*** *= result;  
 }  
}*

**创建返回给客户端的User实体类**

*/\*\*  
 \* 用来返回给客户端  
 \*/****public class*** *User {* ***private*** *String* ***id****;* ***private*** *String* ***username****;* ***private*** *String* ***picSmall****;* ***private*** *String* ***picNormal****;* ***private*** *String* ***nickname****;* ***private*** *String* ***qrcode****;* ***private*** *String* ***clientId****;* ***private*** *String* ***sign****;* ***private*** *Date* ***createtime****;* ***private*** *String* ***phone****;* ***public*** *String getId() {* ***return id****;  
 }* ***public void*** *setId(String id) {* ***this****.****id*** *= id;  
 }* ***public*** *String getUsername() {* ***return username****;  
 }* ***public void*** *setUsername(String username) {* ***this****.****username*** *= username;  
 }* ***public*** *String getPicSmall() {* ***return picSmall****;  
 }* ***public void*** *setPicSmall(String picSmall) {* ***this****.****picSmall*** *= picSmall;  
 }* ***public*** *String getPicNormal() {* ***return picNormal****;  
 }* ***public void*** *setPicNormal(String picNormal) {* ***this****.****picNormal*** *= picNormal;  
 }* ***public*** *String getNickname() {* ***return nickname****;  
 }* ***public void*** *setNickname(String nickname) {* ***this****.****nickname*** *= nickname;  
 }* ***public*** *String getQrcode() {* ***return qrcode****;  
 }* ***public void*** *setQrcode(String qrcode) {* ***this****.****qrcode*** *= qrcode;  
 }* ***public*** *String getClientId() {* ***return clientId****;  
 }* ***public void*** *setClientId(String clientId) {* ***this****.****clientId*** *= clientId;  
 }* ***public*** *String getSign() {* ***return sign****;  
 }* ***public void*** *setSign(String sign) {* ***this****.****sign*** *= sign;  
 }* ***public*** *Date getCreatetime() {* ***return createtime****;  
 }* ***public void*** *setCreatetime(Date createtime) {* ***this****.****createtime*** *= createtime;  
 }* ***public*** *String getPhone() {* ***return phone****;  
 }* ***public void*** *setPhone(String phone) {* ***this****.****phone*** *= phone;  
 }* ***@Override  
 public*** *String toString() {* ***return "User{"*** *+* ***"username='"*** *+* ***username*** *+* ***'\''*** *+* ***", picSmall='"*** *+* ***picSmall*** *+* ***'\''*** *+* ***", picNormal='"*** *+* ***picNormal*** *+* ***'\''*** *+* ***'}'****;  
 }  
}*

**UserController实现**

***@RequestMapping****(****"/login"****)****public*** *Result login(****@RequestBody*** *TbUser user) {* ***try*** *{  
 User \_user =* ***userService****.login(user.getUsername(), user.getPassword());* ***if****(\_user ==* ***null****) {* ***return new*** *Result(****false****,* ***"登录失败，将检查用户名或者密码是否正确"****);  
 }* ***else*** *{* ***return new*** *Result(****true****,* ***"登录成功"****, \_user);  
 }  
 }* ***catch*** *(Exception e) {  
 e.printStackTrace();* ***return new*** *Result(****false****,* ***"登录错误"****);  
 }  
}*

**UserService接口定义**

*/\*\*  
 \* 登录  
 \** ***@param user*** *\** ***@return*** *\*/  
User login(TbUser user);*

**编写UserServiceImpl实现**

***@Override  
public*** *User login(TbUser user) {  
 TbUserExample example =* ***new*** *TbUserExample();  
 TbUserExample.Criteria criteria = example.createCriteria();  
 criteria.andUsernameEqualTo(user.getUsername());  
  
 List<TbUser> userList =* ***userMapper****.selectByExample(example);* ***if****(userList !=* ***null*** *&& userList.size() == 1) {  
 TbUser userInDB = userList.get(0);  
 // MD5加密认证* ***if****(userInDB.getPassword().equals(DigestUtils.md5DigestAsHex(user.getPassword().getBytes()))) {* ***return*** *loadUserById(userInDB.getId());  
 }* ***else*** *{* ***throw new*** *RuntimeException(****"用户名或密码错误"****);  
 }  
 }* ***else*** *{* ***throw new*** *RuntimeException(****"用户不存在"****);  
 }  
}*

### 用户登录功能 - 前端&测试

### 注册功能 - 后端

**UserController**

***@RequestMapping****(****"/register"****)****public*** *Result register(****@RequestBody*** *TbUser user) {* ***try*** *{* ***userService****.register(user);* ***return new*** *Result(****true****,* ***"注册成功"****);  
 }* ***catch*** *(RuntimeException e) {* ***return new*** *Result(****false****, e.getMessage());  
 }  
}*

**UserService接口**

***void*** *register(TbUser user);*

**UserServiceImpl实现**

***@Override  
public void*** *register(TbUser user) {  
 // 1. 查询用户是否存在  
 TbUserExample example =* ***new*** *TbUserExample();  
 TbUserExample.Criteria criteria = example.createCriteria();  
 criteria.andUsernameEqualTo(user.getUsername());  
  
 List<TbUser> userList =* ***userMapper****.selectByExample(example);  
  
 // 1.1 如果存在抛出异常* ***if****(userList !=* ***null*** *&& userList.size() > 0 ) {* ***throw new*** *RuntimeException(****"用户名已经存在!"****);  
 }* ***else*** *{  
 user.setId(****idWorker****.nextId());  
 // MD5加密保存  
 user.setPassword(DigestUtils.md5DigestAsHex(user.getPassword().getBytes()));  
 user.setPicSmall(****""****);  
 user.setPicNormal(****""****);  
 user.setNickname(user.getUsername());  
 user.setQrcode(****""****);   
 user.setCreatetime(****new*** *Date());* ***userMapper****.insert(user);  
 }  
}*

### 注册功能 - 前端&测试

### FASTDFS - 文件服务器介绍与搭建

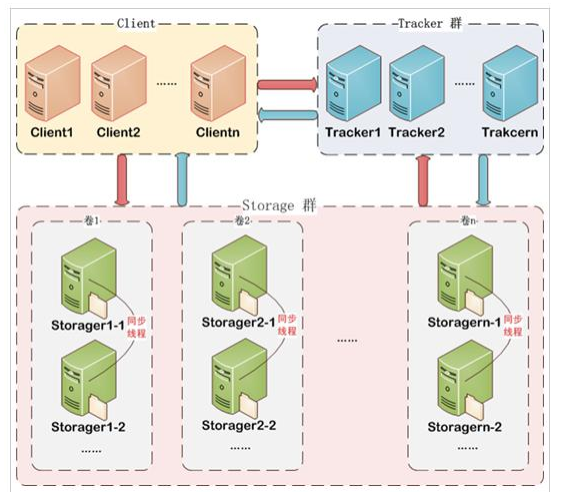
**什么是FastDFS**

*FastDFS 是用 c 语言编写的一款开源的分布式文件系统。FastDFS 为互联网量身定制，充分考虑了冗余备份、负载均衡、线性扩容等机制，并注重高可用、高性能等指标，使用 FastDFS很容易搭建一套高性能的文件服务器集群提供文件上传、下载等服务。*

*FastDFS 架构包括 Tracker server 和 Storage server。客户端请求 Tracker server 进行文件上传、下载，通过 Tracker server 调度最终由 Storage server 完成文件上传和下载。*

*Tracker server 作用是负载均衡和调度，通过 Tracker server 在文件上传时可以根据一些策略找到 Storage server 提供文件上传服务。可以将 tracker 称为追踪服务器或调度服务器。*

*Storage server 作用是文件存储，客户端上传的文件最终存储在 Storage 服务器上，Storageserver 没有实现自己的文件系统而是利用操作系统 的文件系统来管理文件。可以将storage称为存储服务器。*

**

*服务端两个角色：*

*Tracker：管理集群，tracker 也可以实现集群。每个 tracker 节点地位平等。收集 Storage 集群的状态。*

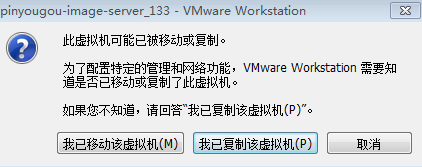
*Storage：实际保存文件 Storage 分为多个组，每个组之间保存的文件是不同的。每个组内部可以有多个成员，组成员内部保存的内容是一样的，组成员的地位是一致的，没有主从的概念。*

**在Linux中搭建FastDFS**

*解压缩fastdfs-image-server.zip*

*双击vmx文件，然后启动。*

*注意：遇到下列提示选择“我已****移动****该虚拟机”！*

**

*IP地址已经固定为192.168.25.133 ，请设置你的仅主机网段为25。*

*登录名为root 密码为itcast*

### FASTDFS - 整合Spring Boot

**导入ComponetImport.java工具类**

**导入FastDFSClient.java、FileUtils.java工具类**

### 个人信息 - 后端照片上传功能开发

**注入FastDFS相关Bean**

***@Autowired  
private*** *Environment* ***env****;****@Autowired  
private*** *FastDFSClient* ***fastDFSClient****;*

**编写UserController update Handler上传照片**

***@RequestMapping****(****"/upload"****)****public*** *Result upload(MultipartFile file, String userid) {* ***try*** *{  
 // 上传  
 String url =* ***fastDFSClient****.uploadFace(file);  
 String suffix =* ***"\_150x150."****;  
 String[] pathList = url.split(****"\\."****);  
 String thumpImgUrl = pathList[0] + suffix + pathList[1];  
  
 // 更新用户头像  
 User user =* ***userService****.updatePic(userid, url, thumpImgUrl);  
 user.setPicNormal(****env****.getProperty(****"fdfs.httpurl"****) + user.getPicNormal());  
 user.setPicSmall(****env****.getProperty(****"fdfs.httpurl"****) + user.getPicSmall());* ***return new*** *Result(****true****,* ***"上传成功"****, user);  
 }* ***catch*** *(IOException e) {  
 e.printStackTrace();* ***return new*** *Result(****false****,* ***"上传失败"****);  
 }  
}*

**编写UserService**

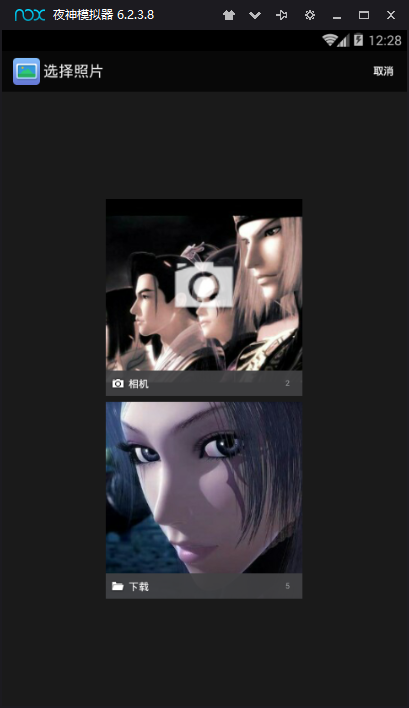
将新上传的图片保存到用户信息数据库中

*/\*\*  
 \* 更新用户头像  
 \** ***@param userid*** *\** ***@param url*** *\** ***@param thumpImgUrl*** *\*/  
User updatePic(String userid, String url, String thumpImgUrl);*

**编写UserServiceImpl**

***@Override  
public*** *User updatePic(String userid, String url, String thumpImgUrl) {  
 TbUser user =* ***userMapper****.selectByPrimaryKey(userid);  
 user.setPicNormal(url);  
 user.setPicSmall(thumpImgUrl);;* ***userMapper****.updateByPrimaryKey(user);  
  
 User userVo =* ***new*** *User();  
 BeanUtils.copyProperties(user, userVo);* ***return*** *userVo;  
}*

### 个人信息 - 前端&测试头像上传

### 个人信息 - 修改昵称后端实现

**编写UserController**

***@RequestMapping****(****"/updateNickname"****)****public*** *Result updateNickname(****@RequestBody*** *TbUser user) {* ***try*** *{* ***userService****.updateNickname(user.getId(), user.getNickname());* ***return new*** *Result(****true****,* ***"修改成功"****);  
 }* ***catch*** *(Exception e) {  
 e.printStackTrace();* ***return new*** *Result(****false****,* ***"修改失败"****);  
 }  
}*

**UserSevice接口**

/\*\*  
 \* 根据用户id更新用户昵称  
 \* **@param userid** \* **@param nickname** \*/  
**void** updateNickname(String userid, String nickname);

**UserServiceImpl实现**

***@Override  
public void*** *updateNickname(String userid, String nickname) {  
 System.****out****.println(userid);  
 TbUser user =* ***userMapper****.selectByPrimaryKey(userid);  
 user.setNickname(nickname);* ***userMapper****.updateByPrimaryKey(user);  
}*

### 个人信息 -重新加载用户信息后端实现

Controller

***@RequestMapping****(****"/findById"****)****public*** *User findById(String userid) {* ***return userService****.findById(userid);  
}*

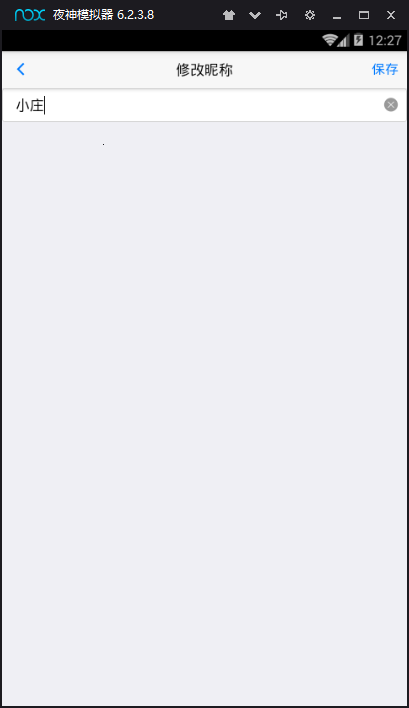
UserService

*/\*\*  
 \* 根据用户id查找用户信息  
 \** ***@param userid*** *用户id  
 \** ***@return*** *用户对象  
 \*/  
User findById(String userid);*

UserServiceImpl

***@Override  
public*** *User findById(String userid) {  
 TbUser tbUser =* ***userMapper****.selectByPrimaryKey(userid);  
 User user =* ***new*** *User();  
 BeanUtils.copyProperties(tbUser, user);* ***return*** *user;  
}*

### 个人信息 - 修改昵称前端测试

### 个人信息 - 二维码生成后端编写

二维码是在用户注册的时候，就根据用户的用户名来自动生成一个二维码图片，并且保存到FastDFS中。

需要对注册的方法进行改造，在注册用户时，编写逻辑保存二维码。并将二维码图片的链接保存到数据库中。

**二维码前端页面展示**



**导入二维码生成工具类**

导入QRCodeUtils.java文件

**UserServiceImpl**

修改注册方法，在注册时，将使用二维码生成工具将二维码保存到FastDFS，并保存链接更新数据库

***@Override  
public void*** *register(TbUser user) {  
 // 1. 查询用户是否存在  
 TbUserExample example =* ***new*** *TbUserExample();  
 TbUserExample.Criteria criteria = example.createCriteria();  
 criteria.andUsernameEqualTo(user.getUsername());  
  
 List<TbUser> userList =* ***userMapper****.selectByExample(example);  
  
 // 1.1 如果存在抛出异常* ***if****(userList !=* ***null*** *&& userList.size() > 0 ) {* ***throw new*** *RuntimeException(****"用户名已经存在!"****);  
 }* ***else*** *{  
 user.setId(****idWorker****.nextId());  
 // MD5加密保存  
 user.setPassword(DigestUtils.md5DigestAsHex(user.getPassword().getBytes()));  
 user.setPicSmall(****""****);  
 user.setPicNormal(****""****);  
 user.setNickname(user.getUsername());  
  
 // 获取临时目录  
 String tmpFolder =* ***env****.getProperty(****"hcat.tmpdir"****);  
 String qrCodeFile = tmpFolder +* ***"/"*** *+ user.getUsername() +* ***".png"****;* ***qrCodeUtils****.createQRCode(qrCodeFile,* ***"user\_code:"*** *+ user.getUsername());* ***try*** *{  
 String url =* ***fastDFSClient****.uploadFile(****new*** *File(qrCodeFile));  
 user.setQrcode(url);  
 }* ***catch*** *(IOException e) {  
 e.printStackTrace();* ***throw new*** *RuntimeException(****"上传文件失败"****);  
 }*  
 *user.setCreatetime(****new*** *Date());* ***userMapper****.insert(user);  
 }  
}*

### 个人信息 - 二维码生成前端测试

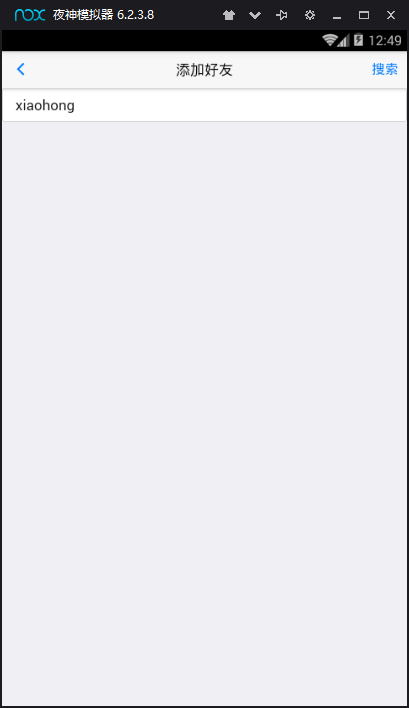
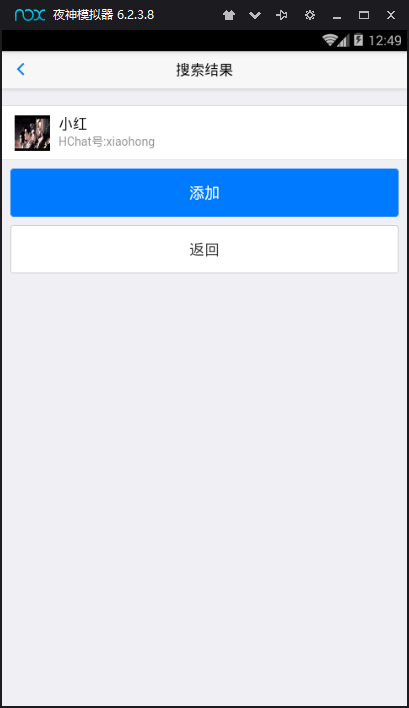
## 业务开发 - 发现页面与通信录

### 搜索朋友 - 后端开发

在搜索朋友的时候需要进行以下判断：

1. *不能添加自己为好友*
2. *如果搜索的用户已经是好友了，就不能再添加了*
3. *如果已经申请过好友并且好友并没有处理这个请求了，也不能再申请。*

**前端页面展示**

搜索朋友其实就是用户搜索，所以我们只需要根据用户名将对应的用户搜索出来即可。

**编写UserController**

***@RequestMapping****(****"/findUserById"****)****public*** *User findUserById(String userid) {  
 System.****out****.println(userid);* ***return userService****.loadUserById(userid);  
}*

**编写UserService接口**

*/\*\*  
 \* 根据用户id加载用户信息  
 \** ***@param userid*** *\** ***@return*** *\*/  
User findUserById(String userid);*

**编写UserServiceImpl实现**

***@Override  
public*** *User findUserById(String userid) {  
 TbUser tbUser =* ***userMapper****.selectByPrimaryKey(userid);  
 User user =* ***new*** *User();  
 BeanUtils.copyProperties(tbUser, user);* ***if****(StringUtils.isNotBlank(user.getPicNormal())) {  
 user.setPicNormal(****env****.getProperty(****"fdfs.httpurl"****) + user.getPicNormal());  
 }* ***if****(StringUtils.isNotBlank(user.getPicSmall())) {  
 user.setPicSmall(****env****.getProperty(****"fdfs.httpurl"****) + user.getPicSmall());  
 }  
 user.setQrcode(****env****.getProperty(****"fdfs.httpurl"****) + user.getQrcode());* ***return*** *user;  
}*

### 搜索朋友 - 前端测试联调

### 添加好友 - 发送好友请求后端开发

添加好友需要发送一个好友请求。

**编写FriendController**

***@RequestMapping****(****"/sendRequest"****)****public*** *Result sendRequest(****@RequestBody*** *TbFriendReq tbFriendReq) {* ***try*** *{* ***friendService****.sendRequest(tbFriendReq);* ***return new*** *Result(****true****,* ***"发送请求成功"****);  
 }* ***catch*** *(RuntimeException e) {* ***return new*** *Result(****false****, e.getMessage());  
 }* ***catch*** *(Exception e) {  
 e.printStackTrace();* ***return new*** *Result(****false****,* ***"发送请求失败"****);  
 }  
}*

**编写FriendService**

*/\*\*  
 \* 发送好友请求  
 \*/****void*** *sendRequest(TbFriendReq friendReq);*

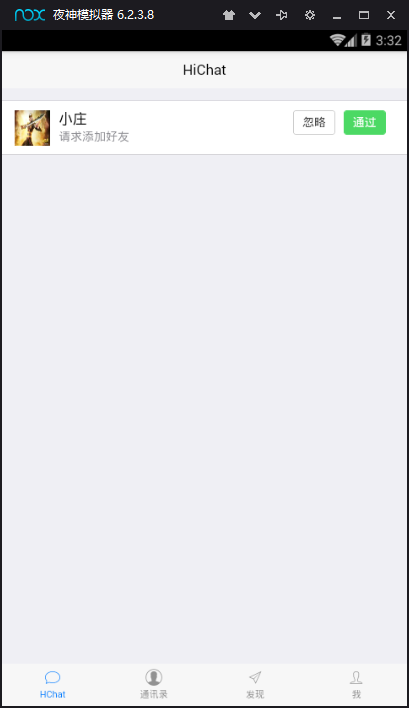
**编写FriendServiceImpl实现**

***@Override  
public void*** *sendRequest(TbFriendReq friendReq) {  
  
 // 判断用户是否已经发起过好友申请  
 TbFriendReqExample example =* ***new*** *TbFriendReqExample();  
 TbFriendReqExample.Criteria criteria = example.createCriteria();  
 criteria.andFromUseridEqualTo(friendReq.getFromUserid());  
 criteria.andToUseridEqualTo(friendReq.getToUserid());  
  
 List<TbFriendReq> friendReqList =* ***friendReqMapper****.selectByExample(example);* ***if****(friendReqList ==* ***null*** *|| friendReqList.size() == 0) {  
 friendReq.setId(****idWorker****.nextId());  
 friendReq.setCreatetime(****new*** *Date());  
 // 设置请求未处理  
 friendReq.setStatus(0);* ***friendReqMapper****.insert(friendReq);  
 }* ***else*** *{* ***throw new*** *RuntimeException(****"您已经请求过了"****);  
 }  
}*

### 添加好友 -前端测试

### 展示好友请求 -后端开发

**前端页面展示**



**编写Controller**

***@RequestMapping****(****"/findFriendReqByUserid"****)****public*** *List<FriendReq> findMyFriendReq(String userid) {* ***return friendService****.findMyFriendReq(userid);  
}*

**编写FriendService**

*/\*\*  
 \* 根据用户id查找好友请求  
 \** ***@param userid*** *\** ***@return*** *\*/  
List<FriendReq> findMyFriendReq(String userid);*

**编写FriendServiceImpl实现**

***@Override  
public*** *List<FriendReq> findMyFriendReq(String userid) {  
 // 查询好友请求  
 TbFriendReqExample example =* ***new*** *TbFriendReqExample();  
 TbFriendReqExample.Criteria criteria = example.createCriteria();  
 criteria.andToUseridEqualTo(userid);  
 // 查询没有处理的好友请求  
 criteria.andStatusEqualTo(0);  
  
 List<TbFriendReq> tbFriendReqList =* ***friendReqMapper****.selectByExample(example);  
 List<FriendReq> friendReqList =* ***new*** *ArrayList<FriendReq>();  
  
 // 加载好友信息* ***for*** *(TbFriendReq tbFriendReq : tbFriendReqList) {  
 TbUser tbUser =* ***userMapper****.selectByPrimaryKey(tbFriendReq.getFromUserid());  
 FriendReq friendReq =* ***new*** *FriendReq();  
 BeanUtils.copyProperties(tbUser, friendReq);  
 friendReq.setId(tbFriendReq.getId());  
 // 添加HTTP前缀  
 friendReq.setPicSmall(****env****.getProperty(****"fdfs.httpurl"****) + friendReq.getPicSmall());  
 friendReq.setPicNormal(****env****.getProperty(****"fdfs.httpurl"****) + friendReq.getPicNormal());  
  
 friendReqList.add(friendReq);  
 }* ***return*** *friendReqList;  
}*

### 展示好友请求 - 前端测试

### 添加好友 - 接受好友请求后端开发

*添加好友需要双方互相添加。*

*例如：A接受B的好友申请，则将A成为B的好友，同时B也成为A的好友。*

**编写FriendController**

***@RequestMapping****(****"/acceptFriendReq"****)****public*** *Result acceptFriendReq(String reqid) {* ***try*** *{* ***friendService****.acceptFriendReq(reqid);* ***return new*** *Result(****true****,* ***"添加好友成功"****);  
 }* ***catch*** *(Exception e) {  
 e.printStackTrace();* ***return new*** *Result(****false****,* ***"添加好友失败"****);  
 }  
}*

**编写FriendService**

*/\*\*  
 \* 接受好友请求  
 \** ***@param reqid*** *好友请求ID  
 \*/****void*** *acceptFriendReq(String reqid);*

**编写FriendServiceImpl**

***@Override  
public void*** *acceptFriendReq(String reqid) {  
 // 设置请求状态为1  
 TbFriendReq tbFriendReq =* ***friendReqMapper****.selectByPrimaryKey(reqid);  
 tbFriendReq.setStatus(1);* ***friendReqMapper****.updateByPrimaryKey(tbFriendReq);  
  
 // 互相添加为好友  
 // 添加申请方好友  
 TbFriend friend1 =* ***new*** *TbFriend();  
 friend1.setId(****idWorker****.nextId());  
 friend1.setUserid(tbFriendReq.getFromUserid());  
 friend1.setFriendsId(tbFriendReq.getToUserid());  
 friend1.setCreatetime(****new*** *Date());  
  
 // 添加接受方好友  
 TbFriend friend2 =* ***new*** *TbFriend();  
 friend2.setId(****idWorker****.nextId());  
 friend2.setFriendsId(tbFriendReq.getFromUserid());  
 friend2.setUserid(tbFriendReq.getToUserid());  
 friend2.setCreatetime(****new*** *Date());* ***friendMapper****.insert(friend1);* ***friendMapper****.insert(friend2);  
  
 // 发送消息更新通信录  
 // 获取发送好友请求方Channel  
 Channel channel = UserChannelMap.get(tbFriendReq.getFromUserid());* ***if****(channel !=* ***null****){  
 Message message =* ***new*** *Message();  
 message.setType(4);  
  
 channel.writeAndFlush(****new*** *TextWebSocketFrame(JSON.toJSONString(message)));  
 }  
}*

### 添加好友 -拒绝添加好友后端开发

在用户选择忽略好友请求时，我们只需要将之前的好友请求状态（status）设置为1。无需添加好友。

**编写FriendController**

***@RequestMapping****(****"/ignoreFriendReq"****)****public*** *Result ignoreFriendReq(String reqid) {* ***try*** *{* ***friendService****.ignoreFriendReq(reqid);* ***return new*** *Result(****true****,* ***"忽略成功"****);  
 }* ***catch*** *(Exception e) {  
 e.printStackTrace();* ***return new*** *Result(****false****,* ***"忽略失败"****);  
 }  
}*

**编写FriendService接口**

*/\*\*  
 \* 忽略好友请求  
 \** ***@param reqid*** *好友请求id  
 \*/****void*** *ignoreFriendReq(String reqid);*

**编写FriendServiceImpl实现**

***@Override  
public void*** *ignoreFriendReq(String reqId) {  
 // 设置请求状态为1  
 TbFriendReq tbFriendReq =* ***friendReqMapper****.selectByPrimaryKey(reqId);  
 tbFriendReq.setStatus(1);* ***friendReqMapper****.updateByPrimaryKey(tbFriendReq);  
}*

### 通信录功能 - 后端

通信录功能就是要根据当前登录用户的id，获取到用户的好友列表。

**前端页面效果**



**编写FriendController**

*/\*\*  
 \* 根据用户id查询好友  
 \** ***@param userid*** *\** ***@return*** *\*/****@RequestMapping****(****"/findFriendsByUserid"****)****public*** *List<User> findFriendsByUserid(String userid) {* ***return friendService****.findFriendsByUserid(userid);  
}*

**编写FriendService**

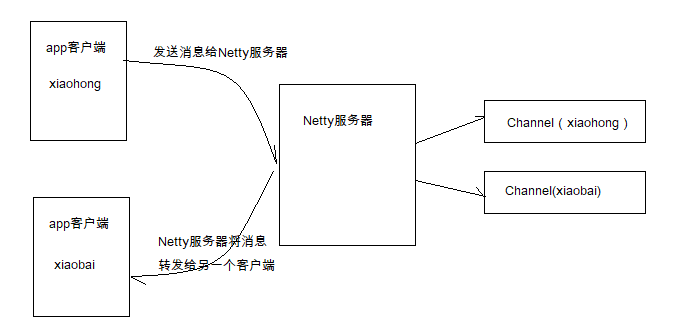
*/\*\*  
 \* 根据用户id查找好友  
 \** ***@param userid*** *\** ***@return*** *\*/  
List<User> findFriendsByUserid(String userid);*

**编写FriendServiceImpl**

***@Override  
public*** *List<User> findFriendsByUserid(String userid) {  
 TbFriendExample example =* ***new*** *TbFriendExample();  
 TbFriendExample.Criteria criteria = example.createCriteria();  
 criteria.andUseridEqualTo(userid);  
  
 List<TbFriend> tbFriendList =* ***friendMapper****.selectByExample(example);  
 List<User> userList =* ***new*** *ArrayList<User>();* ***for*** *(TbFriend tbFriend : tbFriendList) {  
 TbUser tbUser =* ***userMapper****.selectByPrimaryKey(tbFriend.getFriendsId());  
 User user =* ***new*** *User();  
 BeanUtils.copyProperties(tbUser, user);  
 // 添加HTTP前缀  
 user.setPicSmall(****env****.getProperty(****"fdfs.httpurl"****) + user.getPicSmall());  
 user.setPicNormal(****env****.getProperty(****"fdfs.httpurl"****) + user.getPicNormal());  
  
 userList.add(user);  
 }* ***return*** *userList;  
}*

## 业务开发 - 聊天业务

### 聊天业务 - 用户id关联Netty通道后端开发



要使用netty来进行两个客户端之间的通信，需要提前建立好用户id与Netty通道的关联。

服务器端需要对消息进行保存。

每一个App客户端登录的时候，就需要建立用户id与通道的关联。

**导入SpringUtil工具类**

此工具类主要用来在普通Java类中获取Spring容器中的bean

**定义消息实体类**

***public class*** *Message* ***implements*** *Serializable{* ***private*** *Integer* ***type****; // 消息类型* ***private*** *TbChatRecord* ***chatRecord****; // 消息体* ***private*** *String* ***ext****; // 扩展字段  
  
 // getter/setter  
}*

**定义UserChannelMap用来保存用户id与Channel通道关联**

***public class*** *UserChannelMap {* ***public static*** *HashMap<String, Channel> userChannelMap =* ***new*** *HashMap<>();* ***public static void*** *put(String userid, Channel channel) {  
 userChannelMap.put(userid, channel);  
 }* ***public static*** *Channel get(String userid) {* ***return*** *userChannelMap.get(userid);  
 }  
}*

**编写ChatHandller**

用户在第一次登陆到手机App时，会自动发送一个type为0的消息，此时，需要建立用户与Channel通道的关联。后续，将会根据userid获取到Channel，给用户推送消息。

*/\*\*  
 \* 处理消息的handler  
 \* TextWebSocketFrame: 在netty中，是用于为websocket专门处理文本的对象，frame是消息的载体  
 \*/****public class*** *ChatHandler* ***extends*** *SimpleChannelInboundHandler<TextWebSocketFrame> {  
  
 // 用于记录和管理所有客户端的Channel* ***private static*** *ChannelGroup clients =* ***new*** *DefaultChannelGroup(GlobalEventExecutor.****INSTANCE****);* ***@Override  
 protected void*** *channelRead0(ChannelHandlerContext ctx, TextWebSocketFrame msg)* ***throws*** *Exception {  
 // 1. 获取从客户端传输过来的消息  
 String text = msg.text();  
  
 // 2. 判断消息的类型，根据不同的消息类型执行不同的处理  
 System.****out****.println(text);  
 Message message = JSON.parseObject(text, Message.****class****);  
 Integer type = message.getType();* ***switch*** *(type) {* ***case*** *0:  
 // 2.1 当websocket第一次Open的时候，初始化channel，channel关联到userid  
 String userid = message.getChatRecord().getUserid();  
 // 保存userid对应的channel  
 UserChannelMap.put(userid, channel);* ***for*** *(Channel client : clients) {  
 System.****out****.println(****"客户端连接id："*** *+ client.id());  
 }  
  
 // 打印当前在线用户* ***for****(String uid : UserChannelMap.userChannelMap.keySet()) {  
 System.****out****.print(****"用户id:"*** *+ uid +* ***"\n\n"****);  
 System.****out****.println(****"Channelid:"*** *+ UserChannelMap.get(uid));  
 }* ***break****;* ***case*** *1:  
 // 2.2 聊天记录保存到数据库，标记消息的签收状态[未签收]* ***break****;* ***case*** *2:  
 // 2.3 签收消息，修改数据库中的消息签收状态[已签收]  
 // 表示消息id的列表* ***break****;* ***case*** *3:  
 // 2.4 心跳类型的消息* ***break****;  
 }  
 }  
  
 /\*\*  
 \* 当客户端连接服务端之后（打开连接）  
 \* 获取客户端的channel，并且放入到ChannelGroup中去进行管理  
 \** ***@param ctx*** *\** ***@throws*** *Exception  
 \*/* ***@Override  
 public void*** *handlerAdded(ChannelHandlerContext ctx)* ***throws*** *Exception {  
 // 将channel添加到客户端  
 clients.add(ctx.channel());  
 }* ***@Override  
 public void*** *handlerRemoved(ChannelHandlerContext ctx)* ***throws*** *Exception {  
 // 当触发handlerRemoved，ChannelGroup会自动移除对应客户端的channel  
 clients.remove(ctx.channel());  
 }* ***@Override  
 public void*** *exceptionCaught(ChannelHandlerContext ctx, Throwable cause)* ***throws*** *Exception {  
 // 抛出异常时移除通道  
 cause.printStackTrace();  
 ctx.channel().close();  
 clients.remove(ctx.channel());  
 }  
}*

### 聊天业务 - 用户断开连接、连接异常取消关联通道

服务器端应该根据通道的ID，来取消用户id与通道的关联关系。

UserChannelMap类

*/\*\*  
 \* 根据通道id移除用户与channel的关联  
 \** ***@param channelId*** *通道的id  
 \*/****public static void*** *removeByChannelId(String channelId) {* ***if****(!StringUtils.isNotBlank(channelId)) {* ***return****;  
 }* ***for*** *(String s : userChannelMap.keySet()) {  
 Channel channel = userChannelMap.get(s);* ***if****(channelId.equals(channel.id().asLongText())) {  
 System.****out****.println(****"客户端连接断开,取消用户"*** *+ s +* ***"与通道"*** *+ channelId +* ***"的关联"****);  
 userChannelMap.remove(s);* ***break****;  
 }  
 }  
}*

ChatHandler类

***@Override  
public void*** *exceptionCaught(ChannelHandlerContext ctx, Throwable cause)* ***throws*** *Exception {  
 UserChannelMap.removeByChannelId(ctx.channel().id().asLongText());  
 ctx.channel().close();  
}****@Override  
public void*** *handlerRemoved(ChannelHandlerContext ctx)* ***throws*** *Exception {  
 System.****out****.println(****"关闭通道"****);  
 UserChannelMap.removeByChannelId(ctx.channel().id().asLongText());  
 UserChannelMap.print();  
}*

### 聊天业务 - 发送聊天消息后端开发

将消息发送到好友对应的Channel通道，并将消息记录保存到数据库中

**编写ChatHandler**

获取ChatRecordService服务

*Channel channel = ctx.channel();  
ChatRecordService chatRecordService = (ChatRecordService) SpringUtil.getBean(****"chatRecordServiceImpl"****);*

***case*** *1:  
 // 2.2 聊天记录保存到数据库，标记消息的签收状态[未签收]  
 TbChatRecord chatRecord = message.getChatRecord();  
 String msgText = chatRecord.getMessage();  
 String friendid = chatRecord.getFriendid();  
 String userid1 = chatRecord.getUserid();  
  
 // 保存到数据库，并标记为未签收  
  
 String messageId = chatRecordService.insert(chatRecord);  
 chatRecord.setId(messageId);  
  
 // 发送消息  
 Channel channel1 = UserChannelMap.get(friendid);* ***if****(channel1 !=* ***null****) {  
 // 从ChannelGroup查找对应的额Channel是否存在  
 Channel channel2 = clients.find(channel1.id());* ***if****(channel2 !=* ***null****) {  
 // 用户在线,发送消息到对应的通道  
 System.****out****.println(****"发送消息到"*** *+ JSON.toJSONString(message));  
 channel2.writeAndFlush(****new*** *TextWebSocketFrame(JSON.toJSONString(message)));  
 }  
 }* ***break****;*

**编写ChatRecordService接口**

*/\*\*  
 \* 保存聊天记录到服务器  
 \** ***@param chatRecord*** *\*/  
String insert(TbChatRecord chatRecord);*

**编写ChatRecordServiceImpl实现**

***@Override  
public*** *String insert(TbChatRecord chatRecord) {  
 chatRecord.setId(****idWorker****.nextId());  
 chatRecord.setHasRead(0);  
 chatRecord.setCreatetime(****new*** *Date());  
 chatRecord.setHasDelete(0);* ***chatRecordMapper****.insert(chatRecord);* ***return*** *chatRecord.getId();  
}*

### 聊天业务 - 加载聊天记录功能

根据userid和friendid加载未读的聊天记录

**编写ChatRecordController**

***@RequestMapping****(****"/findUnreadByUserIdAndFriendId"****)****public*** *List<TbChatRecord> findUnreadByUserIdAndFriendId(String userid, String friendid) {* ***return chatRecordService****.findUnreadByUserIdAndFriendId(userid, friendid);  
}*

**编写ChatRecordService**

*/\*\*  
 \* 根据用户ID和朋友ID获取未读的消息  
 \** ***@param userid*** *\** ***@param friendId*** *\** ***@return*** *\*/  
List<TbChatRecord> findUnreadByUserIdAndFriendId(String userid, String friendId);*

**编写ChatRecordServiceImpl实现**

***@Override  
public*** *List<TbChatRecord> findUnreadByUserIdAndFriendId(String userid, String friendid) {  
 TbChatRecordExample example =* ***new*** *TbChatRecordExample();  
 TbChatRecordExample.Criteria criteria1 = example.createCriteria();  
  
 criteria1.andUseridEqualTo(friendid);  
 criteria1.andFriendidEqualTo(userid);  
 criteria1.andHasReadEqualTo(0);  
 criteria1.andHasDeleteEqualTo(0);  
  
 TbChatRecordExample.Criteria criteria2 = example.createCriteria();  
 criteria2.andUseridEqualTo(userid);  
 criteria2.andFriendidEqualTo(friendid);  
 criteria2.andHasReadEqualTo(0);  
 criteria2.andHasDeleteEqualTo(0);  
  
 example.or(criteria1);  
 example.or(criteria2);  
  
 // 加载未读消息  
 List<TbChatRecord> chatRecordList =* ***chatRecordMapper****.selectByExample(example);  
  
 // 将消息标记为已读* ***for*** *(TbChatRecord tbChatRecord : chatRecordList) {  
 tbChatRecord.setHasRead(1);* ***chatRecordMapper****.updateByPrimaryKey(tbChatRecord);  
 }* ***return*** *chatRecordList;  
}*

### 聊天业务 - 已读/未读消息状态标记

**已读消息**

当用户接收到聊天消息，且聊天窗口被打开，就会发送一条用来签收的消息到Netty服务器

用户打开聊天窗口，加载所有聊天记录，此时会把发给他的所有消息设置为已读

**未读消息**

如果用户没有打开聊天窗口，就认为消息是未读的

ChatRecordController

***@RequestMapping****(****"/findUnreadByUserid"****)****public*** *List<TbChatRecord> findUnreadByUserid(String userid) {* ***try*** *{* ***return chatRecordService****.findUnreadByUserid(userid);  
 }* ***catch*** *(Exception e) {  
 e.printStackTrace();* ***return new*** *ArrayList<TbChatRecord>();  
 }  
}*

ChatRecordService

*/\*\*  
 \* 设置消息为已读  
 \** ***@param id*** *聊天记录的id  
 \*/****void*** *updateStatusHasRead(String id);*

ChatRecordServiceImpl

***@Override  
public void*** *updateStatusHasRead(String id) {  
 TbChatRecord tbChatRecord =* ***chatRecordMapper****.selectByPrimaryKey(id);  
 tbChatRecord.setHasRead(1);* ***chatRecordMapper****.updateByPrimaryKey(tbChatRecord);  
}*

ChatHandler

***case*** *2:  
 // 将消息记录设置为已读  
 chatRecordService.updateStatusHasRead(message.getChatRecord().getId());* ***break****;*

### 聊天业务 - 未读消息读取

在用户第一次打开App的时候，需要将所有的未读消息加载到App

ChatRecordController

***@RequestMapping****(****"/findUnreadByUserid"****)****public*** *List<TbChatRecord> findUnreadByUserid(String userid) {* ***try*** *{* ***return chatRecordService****.findUnreadByUserid(userid);  
 }* ***catch*** *(Exception e) {  
 e.printStackTrace();* ***return new*** *ArrayList<TbChatRecord>();  
 }  
}*

ChatRecordService

*/\*\*  
 \* 根据用户id，查询发给他的未读消息记录  
 \** ***@param userid*** *用户id  
 \** ***@return*** *未读消息列表  
 \*/  
List<TbChatRecord> findUnreadByUserid(String userid);*

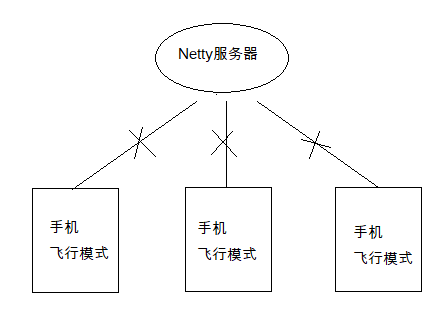
ChatRecordServiceImpl

***@Override  
public*** *List<TbChatRecord> findUnreadByUserid(String userid) {  
 TbChatRecordExample example =* ***new*** *TbChatRecordExample();  
 TbChatRecordExample.Criteria criteria = example.createCriteria();  
  
 // 设置查询发给userid的消息  
 criteria.andFriendidEqualTo(userid);  
 criteria.andHasReadEqualTo(0);* ***return chatRecordMapper****.selectByExample(example);  
}*

## 业务开发 - 心跳机制

### Netty心跳处理以及读写超时设置

Netty并不能监听到客户端设置为飞行模式时，自动关闭对应的通道资源。我们需要让Netty能够定期检测某个通道是否空闲，如果空闲超过一定的时间，就可以将对应客户端的通道资源关闭。



编写后端Netty心跳检查的Handler

*/\*\*  
 \* 检测Channel的心跳Handler  
 \*/****public class*** *HeartBeatHandler* ***extends*** *ChannelInboundHandlerAdapter {  
 // 客户端在一定的时间没有动作就会触发这个事件* ***@Override  
 public void*** *userEventTriggered(ChannelHandlerContext ctx, Object evt)* ***throws*** *Exception {  
 // 用于触发用户事件，包含读空闲/写空闲* ***if****(evt* ***instanceof*** *IdleStateEvent) {  
 IdleStateEvent event = (IdleStateEvent)evt;* ***if****(event.state() == IdleState.****READER\_IDLE****) {  
 System.****out****.println(****"读空闲..."****);  
 }* ***else if****(event.state() == IdleState.****WRITER\_IDLE****) {  
 System.****out****.println(****"写空闲..."****);  
 }* ***else if****(event.state() == IdleState.****ALL\_IDLE****) {  
 System.****out****.println(****"关闭客户端通道"****);  
 // 关闭通道，避免资源浪费  
 ctx.channel().close();  
 }  
 }  
 }  
}*

在通道初始化器中（WebSocketInitailizer）添加心跳检查

*// 增加心跳事件支持  
// 第一个参数: 读空闲4秒  
// 第二个参数： 写空闲8秒  
// 第三个参数： 读写空闲12秒  
pipeline.addLast(****new*** *IdleStateHandler(4, 8, 12));  
  
pipeline.addLast(****new*** *HeartBeatHandler());*

### 测试Netty心跳机制