一　基于 Netty 网络编程项目实战课程

1　项目介绍

2　Netty 介绍与相关基础知识

浏览器是无状态的，如果服务端需要实时推送信息，那么就要和服务器一直有一个连接

2.1 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

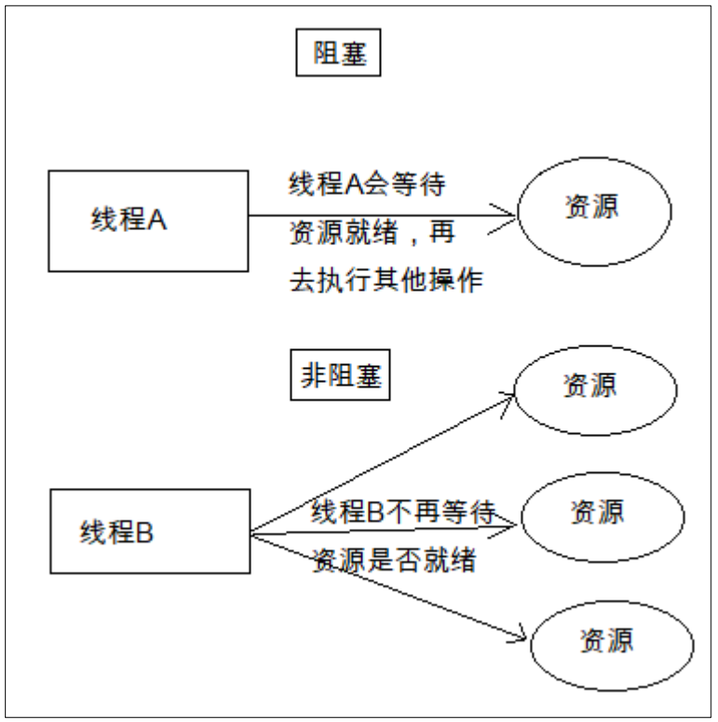
大数据：Apache Storm（Supervisor worker 进程间的通信也是基于 Netty 来实现的）

2.2 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）

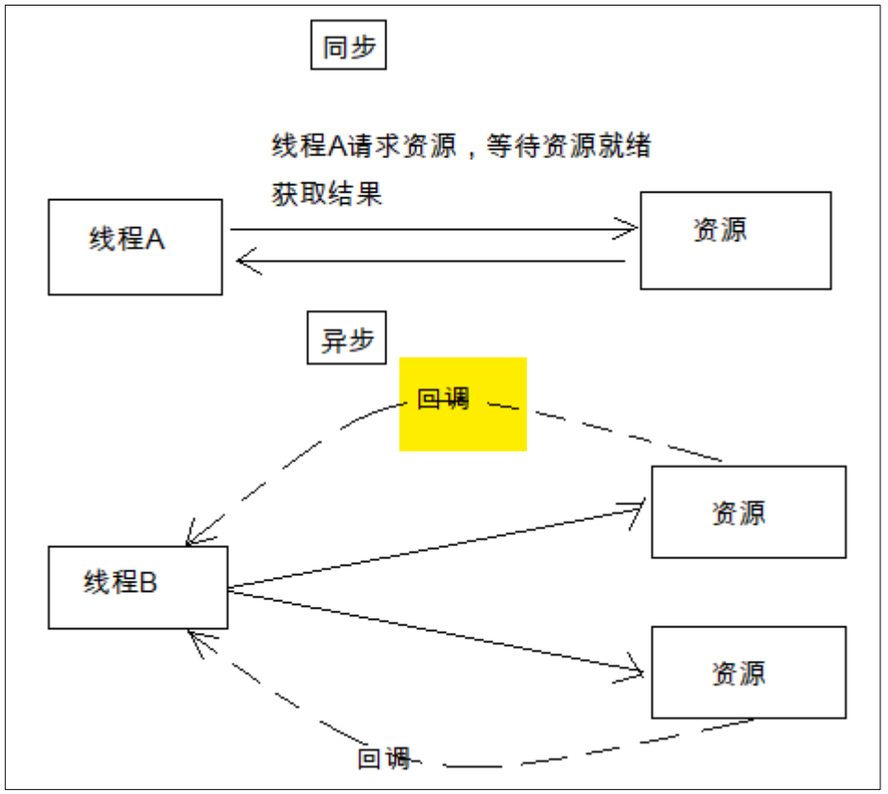
线程访问资源，该资源是否准备就绪的一种处理方式。



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



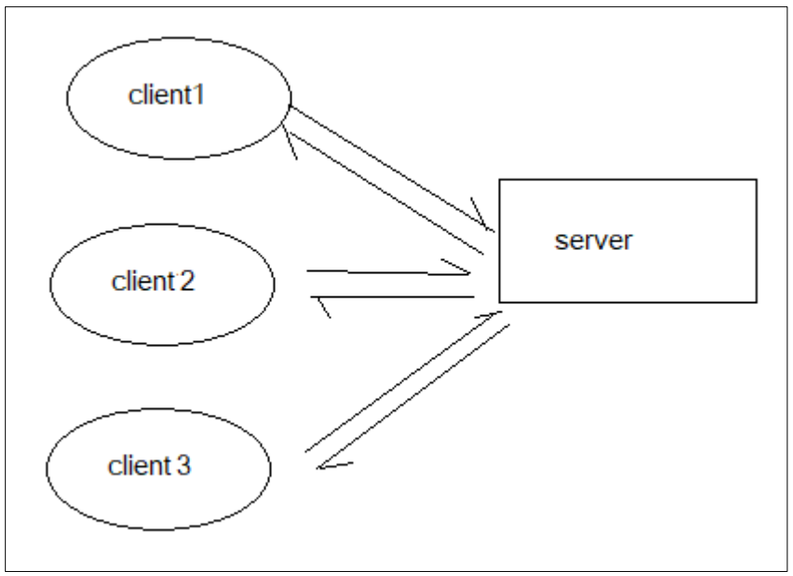
BIO----------同步阻塞 IO

开多几个服务器

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。

这里阻塞有几个概念：
服务器的线程没有读到客户端的socket发来的信息而阻塞，当前线程处于waiting状态
客户端的socket没有建立连接而阻塞，线程数量不够

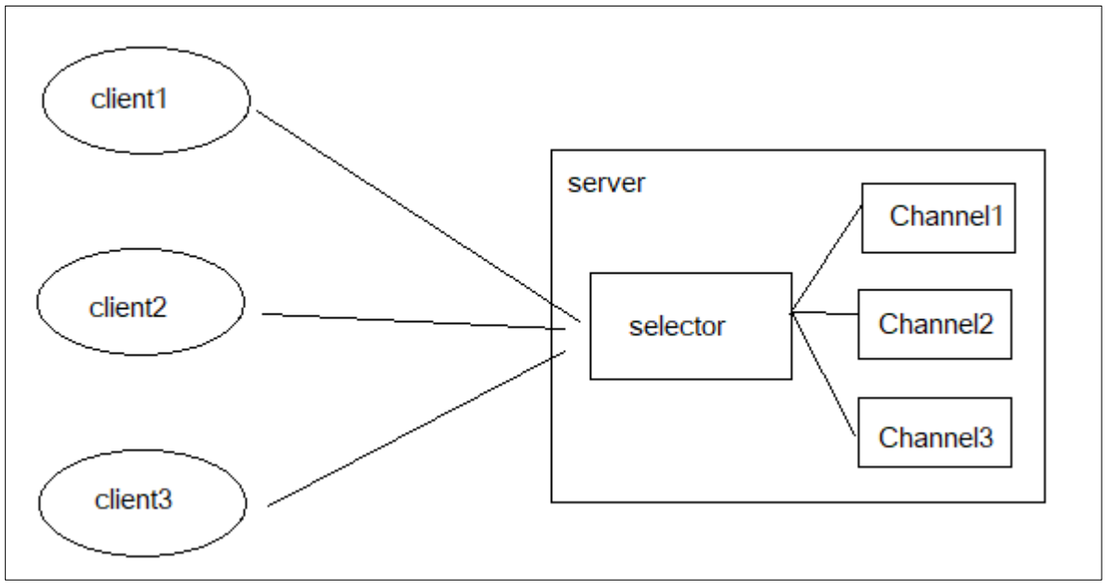


NIO-------同步非阻塞 IO

自己监控，没人通知

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

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。

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

2.3.1 单线程模型

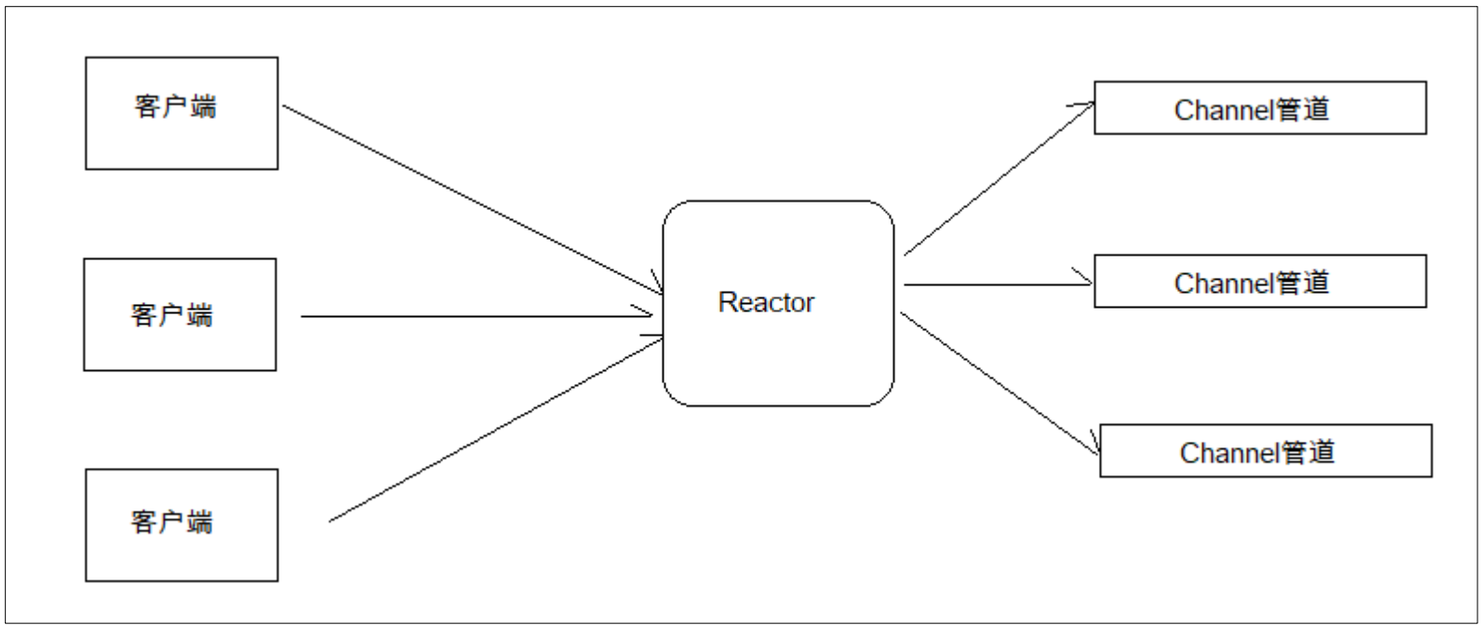
用户发起 IO 请求到 Reactor 线程

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。

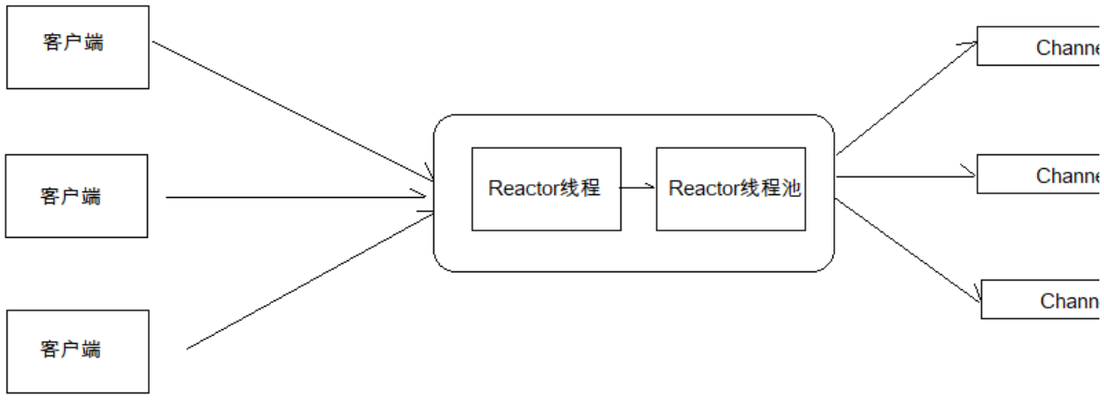
1. *但单线程的 Reactor 模型每一个用户事件都在一个线程中执行：*
2. *性能有极限，不能处理成百上千的事件*
3. *当负荷达到一定程度时，性能将会下降*
4. *某一个事件处理器发生故障，不能继续处理其他事件*



2.3.2 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

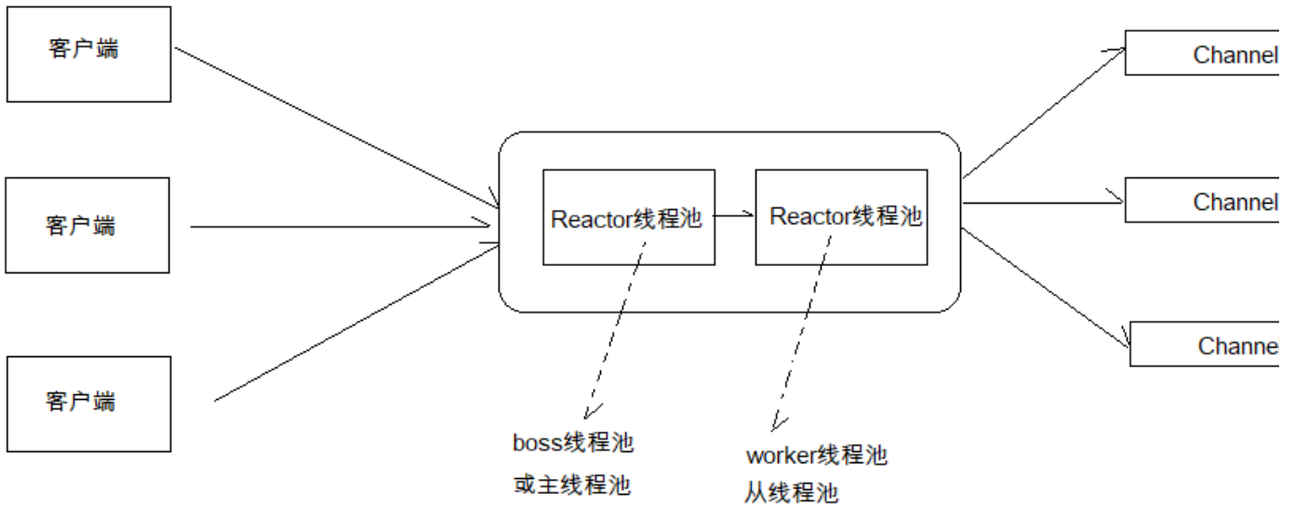


但是如果并发仍然很大，Reactor 仍然无法处理大量的客户端请求

2.3.3 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。



2.4 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());
```

```
    }
}
```

2.5 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>
```

2.6 MUI、HTML5+、HBuilder 介绍

MUI 介绍

```
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/

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#

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

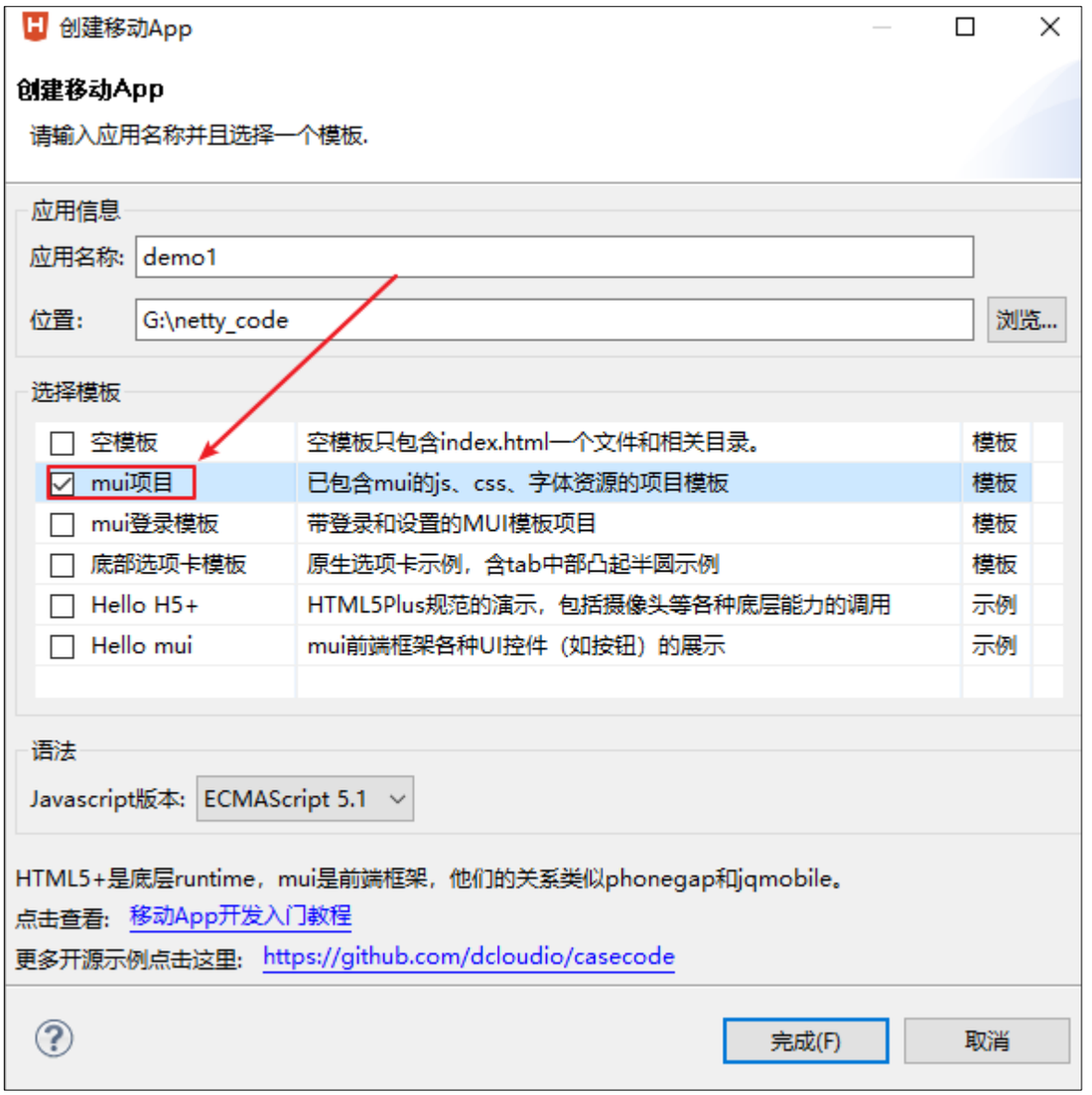
Android/IOS 的安装包。

http://www.dcloud.io/

2.7 MUI 前端开发

2.7.1 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

2.7.2 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");
});
```

2.7.3 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;
    }
}
```

2.7.4 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);
```

2.7.5 页面跳转

```
mui.openWindow({
    url: 'login_succss.html',
    id:'login_succss.html'
});
```

2.7.6 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");
```

3　构建项目

3.1 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

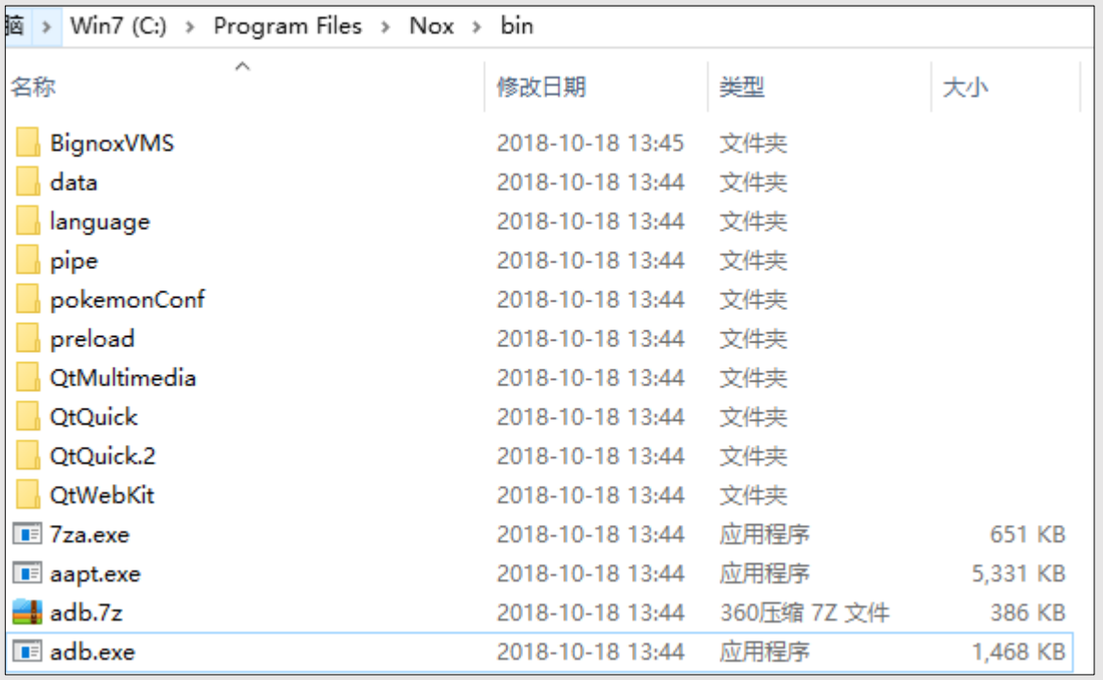
数据库：mysql

3.2 使用模拟器进行测试

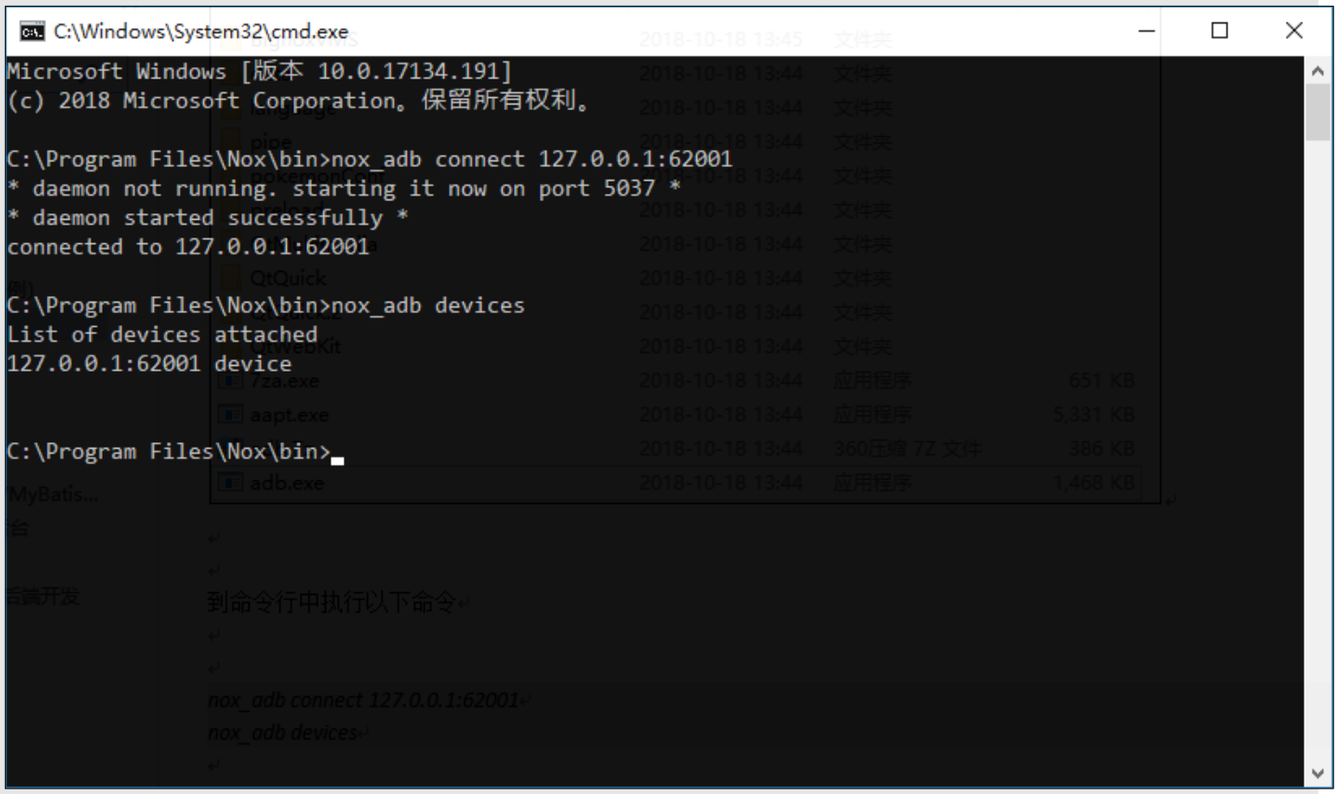
安装附件中的夜神 Android 模拟器（nox_setup_v6.2.3.8_full.exe）

双击桌面图标启动模拟器

安装后找到模拟器的安装目录



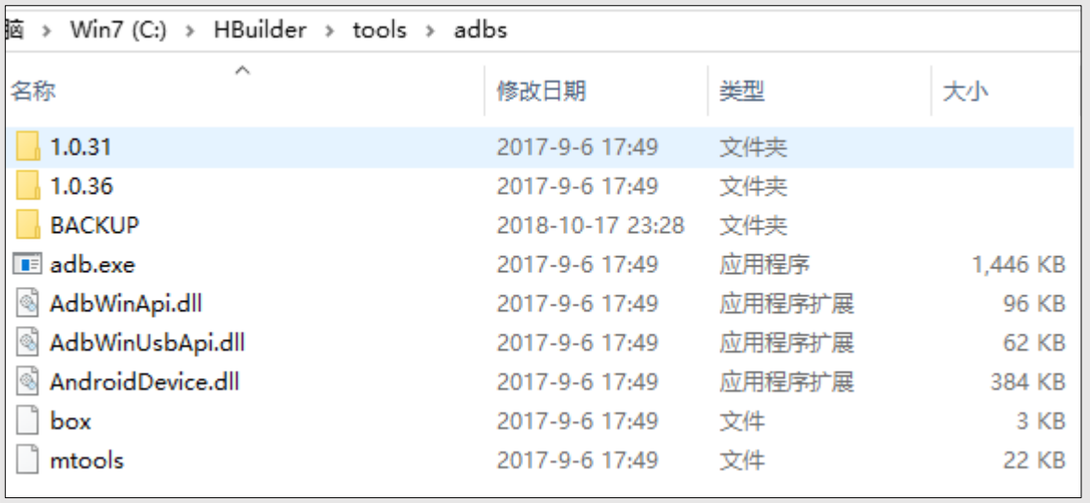
到命令行中执行以下命令



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

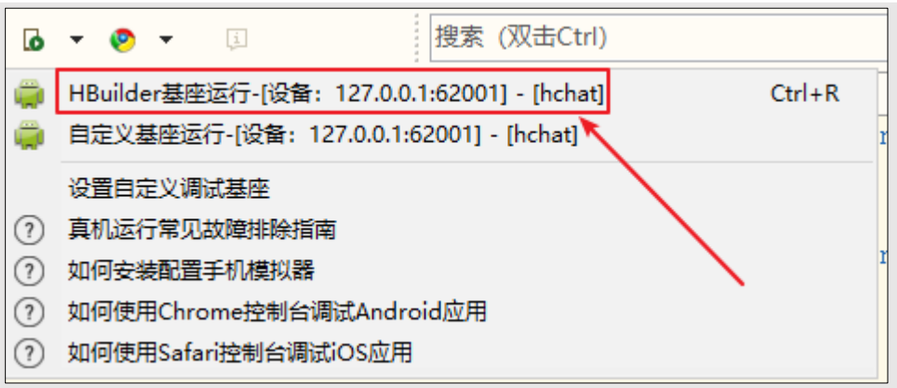


切换到命令行中执行以下命令

adb connect 127.0.0.1:62001

adb devices

打开 HBuilder 开始调试



3.3 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。

3.4 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

Field	Type	Comment
id	varchar(255) NOT NULL	ID
username	varchar(255) NULL	用户名
password	varchar(255) NULL	密码
pic_small	varchar(255) NULL	头像（小尺寸）
pic_normal	varchar(255) NULL	头像（正常尺寸）
nickname	varchar(255) NULL	昵称
qrcode	varchar(255) NULL	二维码
client_id	varchar(255) NULL	手机端唯一ID
sign	varchar(1024) NULL	签名
createtime	date NULL	注册日期
phone	varchar(255) NULL	绑定手机

tb_friend 朋友表

Field	Type	Comment
id	varchar(255) NOT NULL	
userid	varchar(255) NULL	用户id
friends_id	varchar(255) NULL	好友id
comments	varchar(255) NULL	备注
createtime	date NULL	添加好友日期

tb_friend_req 申请好友表

Field	Type	Comment
id	varchar(255) NOT NULL	id
from_userid	varchar(255) NULL	请求好友用户id
to_userid	varchar(255) NULL	被请求好友用户id
createtime	date NULL	请求时间
message	varchar(255) NULL	发送的消息
status	int(1) NULL	是否已处理

tb_chat_record 聊天记录表

Field	Type	Comment
id	varchar(255) NOT NULL	id
userid	varchar(255) NULL	用户id
friendid	varchar(255) NULL	好友id
has_read	int(1) NULL	是否已读
createtime	date NULL	聊天时间
has_delete	int(1) NULL	是否删除
message	varchar(1024) NULL	消息

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

3.5 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

4　业务开发 - 用户注册/登录/个人信息

4.1 用户登录功能 -后端开发

导入 IdWorker.java 雪花算法 ID 生成器　　有什么用

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");
    }
}
```

4.2 用户登录功能 - 前端&测试

4.3 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);
    }
}
```

4.4 注册功能 - 前端&测试

4.5 FASTDFS - 文件服务器介绍与搭建

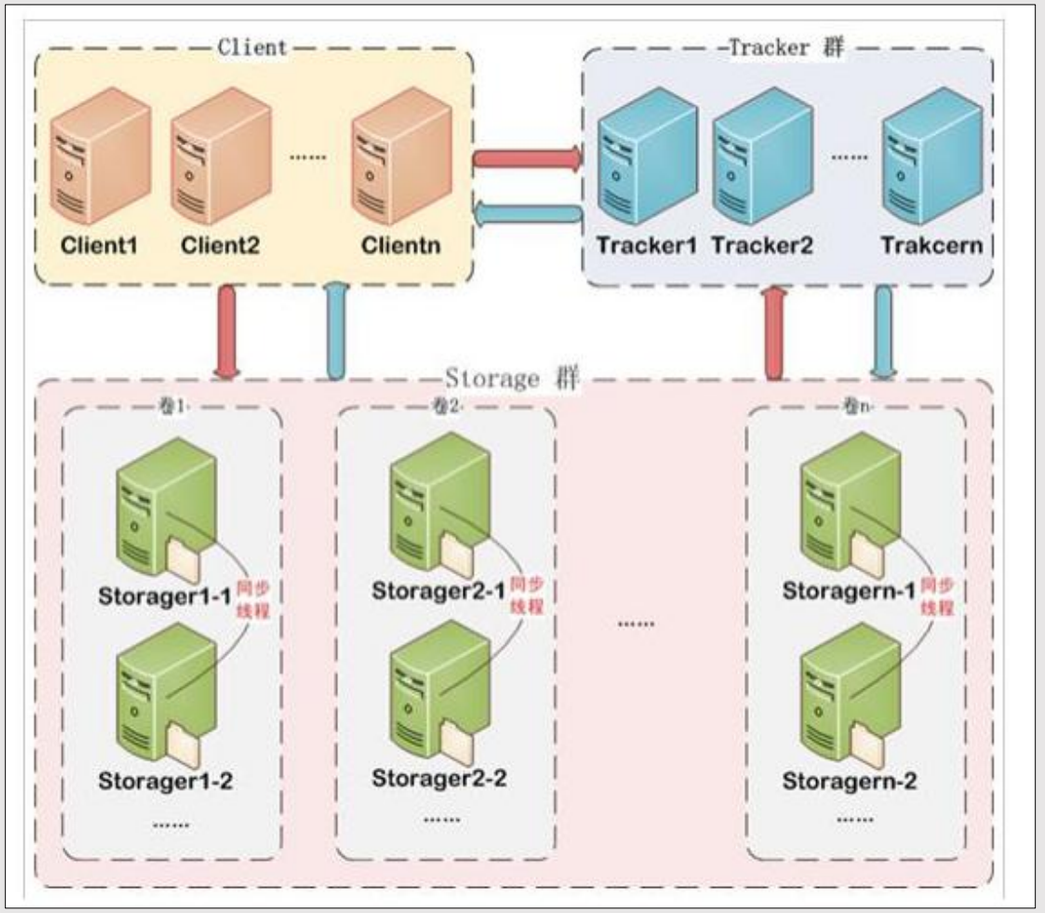
什么是 FastDFS

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。

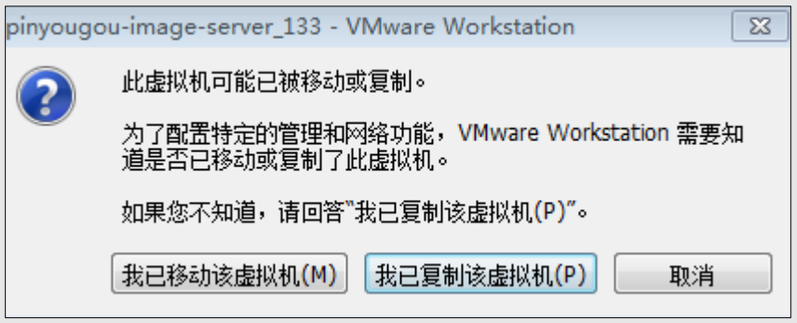
Storage：实际保存文件 Storage 分为多个组，每个组之间保存的文件是不同的。每个组内部可以有多个成员，组成员内部保存的内容是一样的，组成员的地位是一致的，没有主从的概念。

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

4.6 FASTDFS - 整合 Spring Boot

导入 ComponetImport.java 工具类

导入 FastDFSClient.java、FileUtils.java 工具类

4.7 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

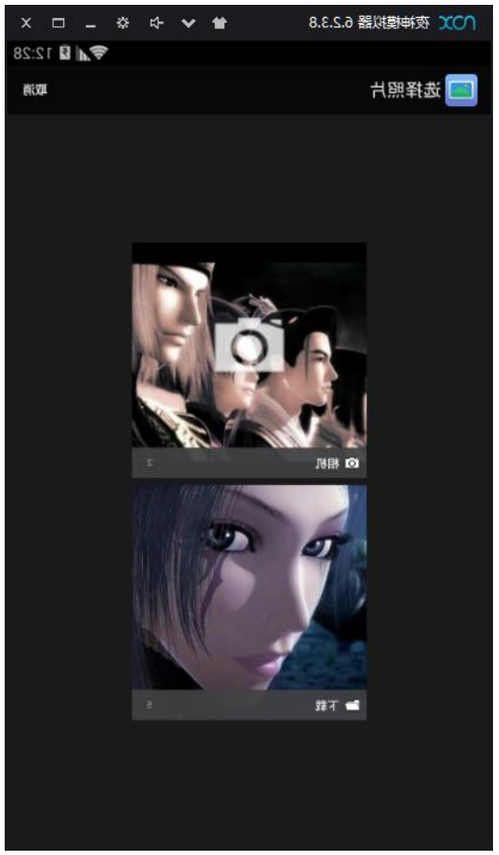
```
/**
 * 更新用户头像
 * @param userid
 * @param url
 * @param thumpImgUrl
 */
User updatePic(String userid, String url, String thumpImgUrl);
```

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;
}
```

4.8 个人信息 - 前端&测试头像上传



4.9 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);
}
```

4.10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);
}
```

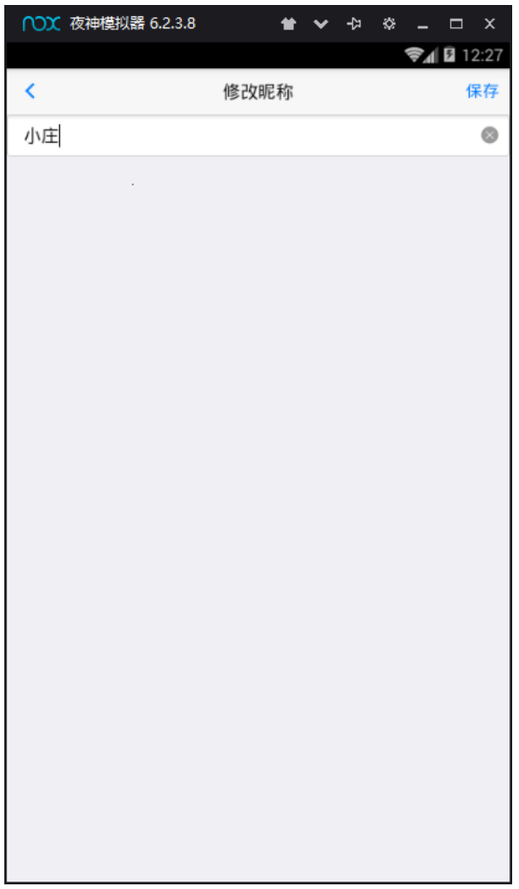
UserService

```
/**
 * 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;
}
```

4.11个人信息 - 修改昵称前端测试



4.12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);
    }
}
```

4.13个人信息 - 二维码生成前端测试

5 业务开发 - 发现页面与通信录

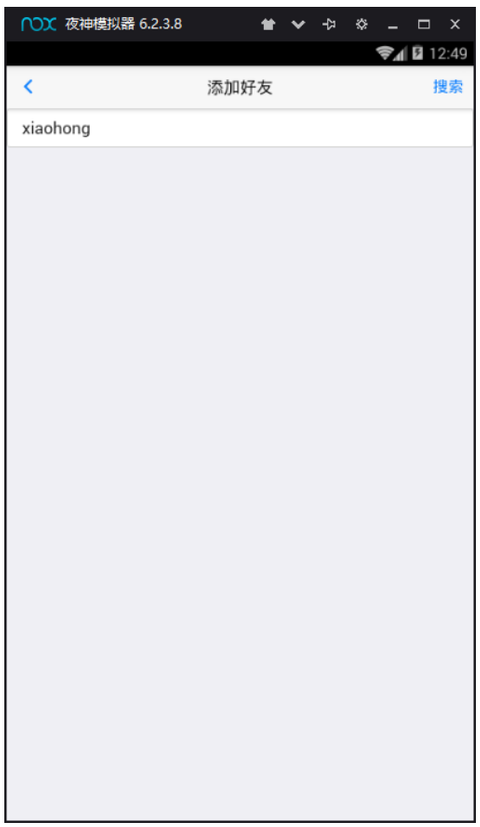
5.1 搜索朋友 - 后端开发

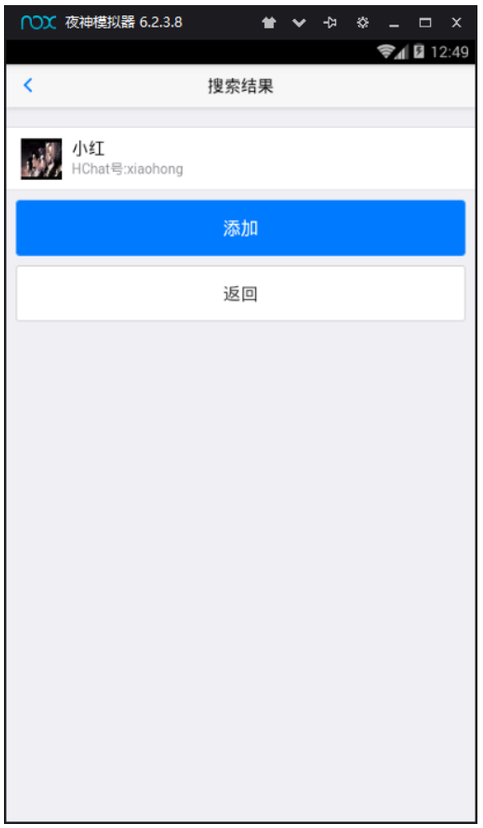
在搜索朋友的时候需要进行以下判断：

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;
}
```

5.2 搜索朋友 - 前端测试联调

5.3 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

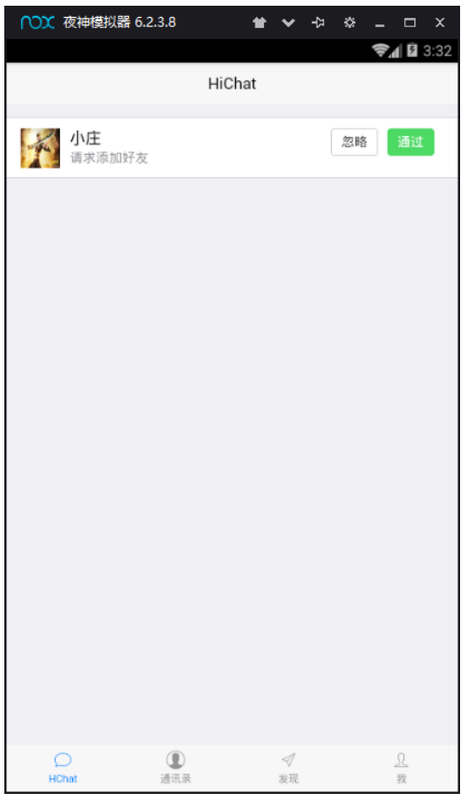
```
@Override
public void sendRequest(TbFriendReq friendReq) {

    // 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");
    }
}
```

5.4 添加好友 -前端测试

5.5 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;
}
```

5.6 展示好友请求 - 前端测试

5.7 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)));
    }
}
```

5.8 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);
```

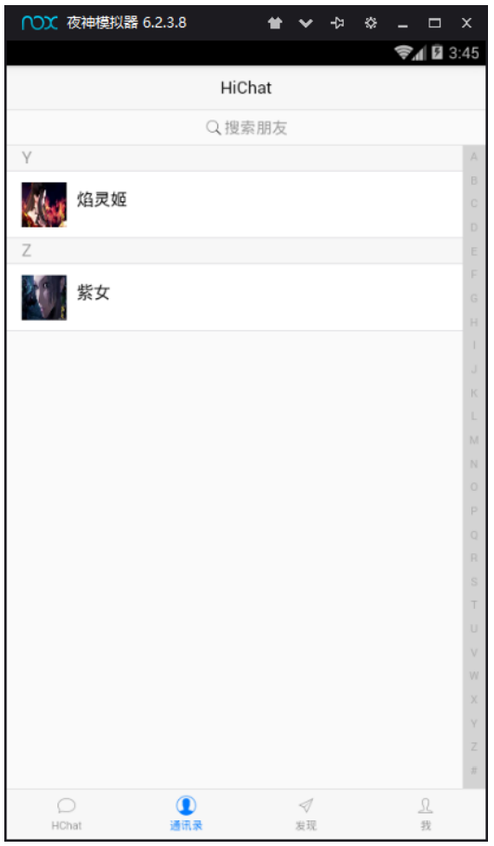
编写 FriendServiceImpl 实现

```
    @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);
    }
```

5.9 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
```

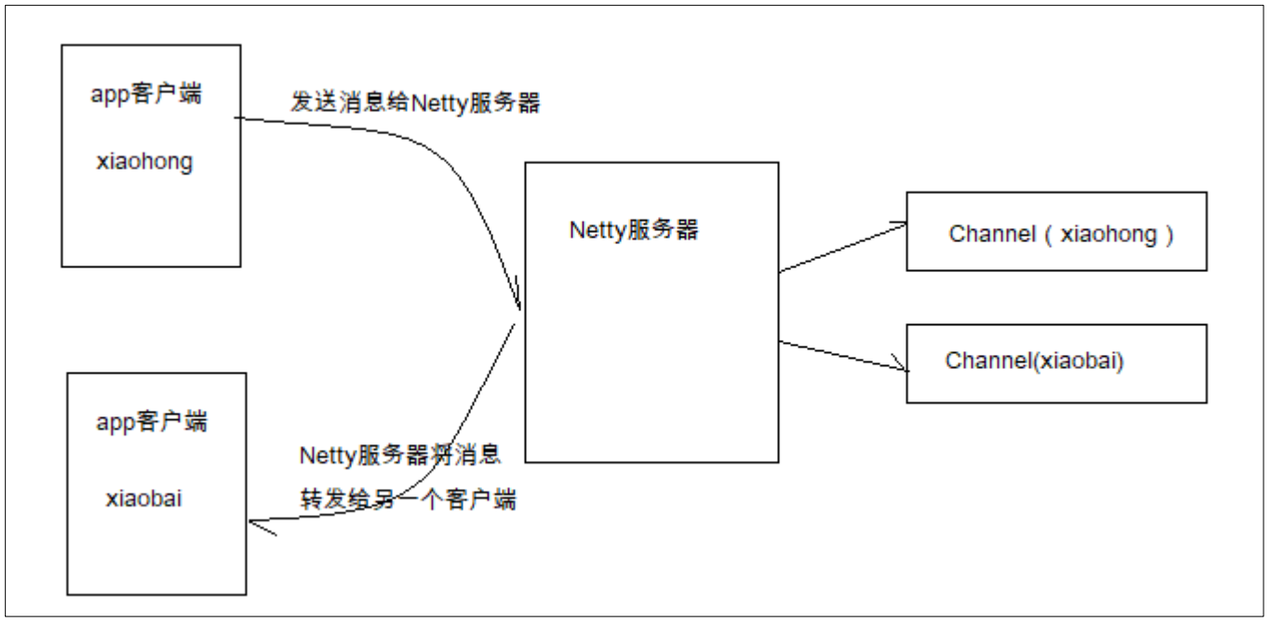
```
 */
List<User> findFriendsByUserid(String userid);
```

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;
}
```

6　业务开发 - 聊天业务

6.1 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

第一次主动发送消息

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());
    }
}
```

6.2 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();
}
```

6.3 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();
}
```

6.4 聊天业务 - 加载聊天记录功能

聊天记录不是通常保存在本地的嘛

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;
}
```

6.5 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;
```

6.6 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);
```

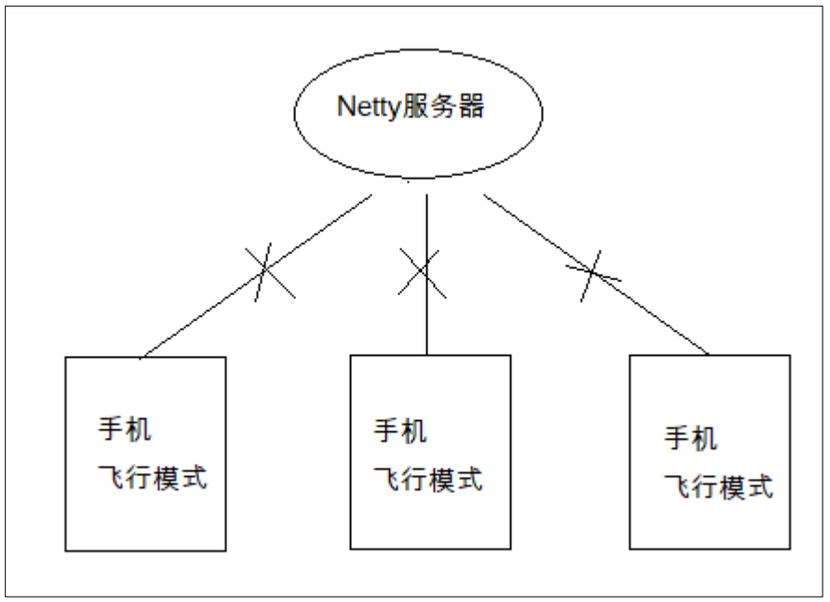
ChatRecordServiceImpl

```
@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);
}
```

7　业务开发 - 心跳机制

7.1 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());
```

7.2 测试 Netty 心跳机制