在线聊天系统系统分析及设计报告

1. 需求分析
2. 核心用户分析

在线聊天系统主要针对一些年轻用户群体以及因为工作需求而对于实时交流以及非实时交流有较大需求的群里。就青年群体而言，这一用户群体特征比较鲜明，其主要需求为基础聊天需求以及一些能够凸显个性的功能需求。在线聊天对于青年人来说也逐渐成为一种主流的设计方式。年轻人们通过在线交流和好友印象的可以了解到对方的性格，而且可以通过相互添加好友保持关系。而对于有工作需求的人来说能够实时交流以及处理未读消息就显得十分重要。

1. 系统的主要功能的概述

首先未注册的用户可以注册账号，已经注册的用户可以使用账号密码进行登录。

用户可以搜索好友，搜索之后可以进行添加好友

主界面分为两个部分，一个部分为消息盒子，一个部分为好友盒子

消息盒子主要存放未读消息，如果有一个好友向你发送消息你没有点到聊天框里查看的话就会在消息盒子界面显示

好友盒子显示如下几个部分，好友列表，添加好友的入口，个人信息的入口，朋友验证的入口

所有的好友会在好友列表中展示，一开始所有的好友的在默认分组。点击好友之后可以进入好友的资料卡页面

可以在好友资料卡中可以查看好友的基本消息，以及会显示好友的印象，当点击某个印象标签的时候会提示你可以进行删除。还可以在好友资料卡页面点击发送消息进入聊天窗口。除此之外右上角点击之后可以有删除好友，移动好友，添加标签的选项

删除好友：点击之后好友将被删除，你可以通过再次发送验证消息进行添加

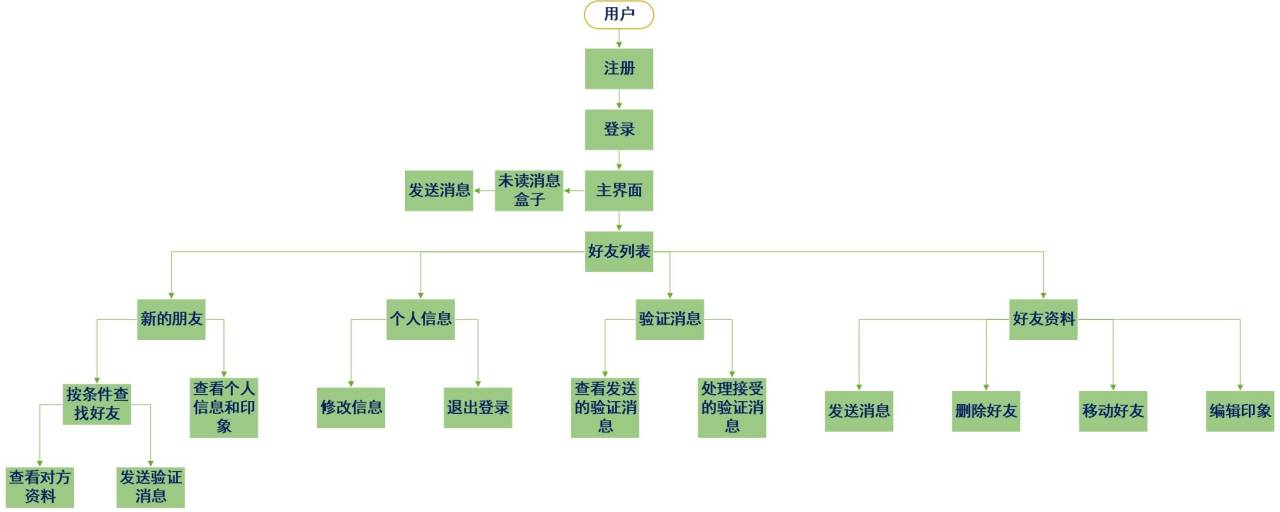
移动好友：可以将好友移动到指定的分组，如果分组不存在则创建分组，若移动后分组内没有成员则删除分组。

添加标签：可以为好友添加一个标签。

当进入聊天框之后发送消息对方就可以发收到，点击下载聊天记录的按钮就可以下载所有的聊天记录，点击删除聊天记录可以删除和当前用户所有的云端记录。

个人信息，在这里可以修改个人信息包括修改头像，以及删除别人给自己的标签，并且可以在此处退出登录

1. 项目操作流程图



1. 功能详解
2. 登录

使用账号密码进行登录，登录成功之后跳转到主页面中的消息盒子的页面

1. 注册

账号采用邮箱格式，密码要求大于八位

1. 消息盒子

消息盒子显示你的所有的未读消息，一旦消息已读就会从消息盒子中去除

1. 好友盒子

好友盒子有如下这些部分组成：新的朋友，我的账号，朋友验证，好友列表

1. 好友列表

按照分组展示所有的好友，点击好友可以进入好友资料卡页面

1. 朋友验证

当你发送的请求别人已经处理完了或者别人向你发送了请求的话此处会有一个红点表示消息数量。点击进入之后进入验证消息模块

1. 我的账号

点击之后进入个人资料卡，在这里可以修改姓名，头像，性别，头像要求小于30kb，年龄要求不能为负数，性别要求只能是男或者女，还可以在此处删除自己的标签，也可以退出登录。

1. 新的朋友

可以进行全局搜索，即不进行任何输入直接回车可以显示所有的好友，并且可以进行模糊搜索，只输入名字的部分也可搜索到。并且可以添加年龄和性别的限制条件。点击搜索结果可以进入好友资料卡。在这里可以填写验证消息，并且发送好友验证，自己不能添加自己，不能添加以及添加的好友，如果已经发送过依次请求对方为响应也不能发送。当这里发送之后对方的朋友验证会出现红点。

1. 验证消息

当我们点击朋友验证之后，进入验证消息页面，如果我们发送的消息被处理了，则会有一个红点标记，别人发送的请求我们可以选择拒绝和接受。如果我们进入了此页面的话，如果存在我们发送的消息被处理了且我们自己之前未读的，则会被设置为已读。对于别人发给自己的请求，则必须在处理完之后才会被设置为已读。

1. 好友资料卡

显示好友的基本信息，好友的标签，点击标签可以进行删除，并且可以在此页面点击发送消息进入聊天框进行聊天，此页面中点击右上角还可以进行删除好友，移动好友，添加标签。

1. 删除好友：将好友从列表中删除，删除后可以再次发送验证消息
2. 移动好友

输入要移动的分组如果不存在则创建分组，若某个分组内没有了用户则删除分组，所有用户默认在默认分组中

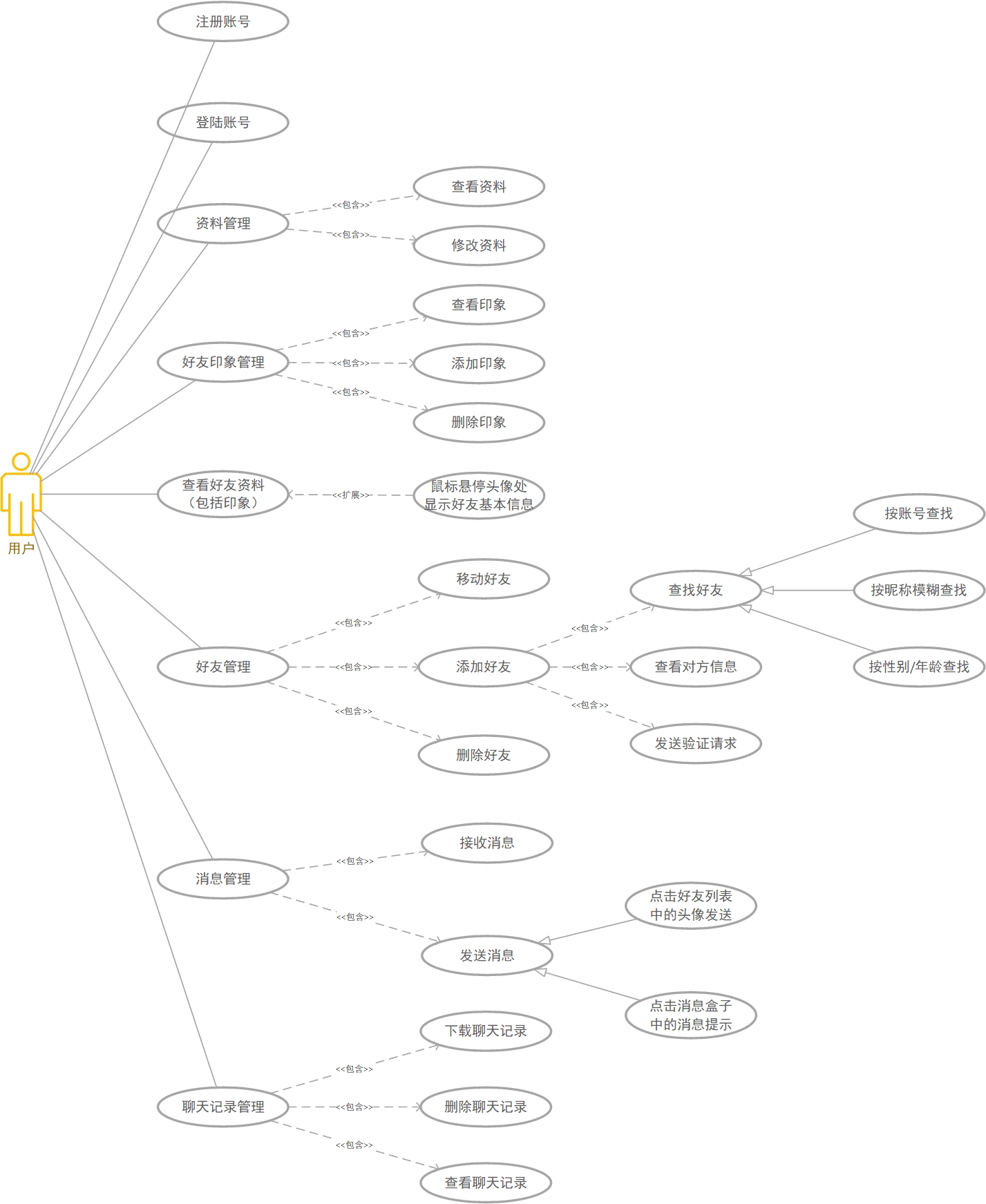
1. 添加标签

可以对一个用户添加一个标签，添加重复标签没有用

1. 聊天界面

聊天界面可以双方可以实时发送消息，显示的时候自己的消息在右侧，对方的消息在左侧，且按时间排序，点击下载按钮可以进行聊天记录下载，点击删除按钮可以删除云端数据

1. 系统的顶级用例图



1. 系统的原型图设计

原型图主要是用图片的形式站输出之前的功能模块，并且也是后面前端UI的主要依据

登陆界面，注册界面类似 消息盒子界面

聊天界面 好友列表界面

搜索界面 好友申请界面

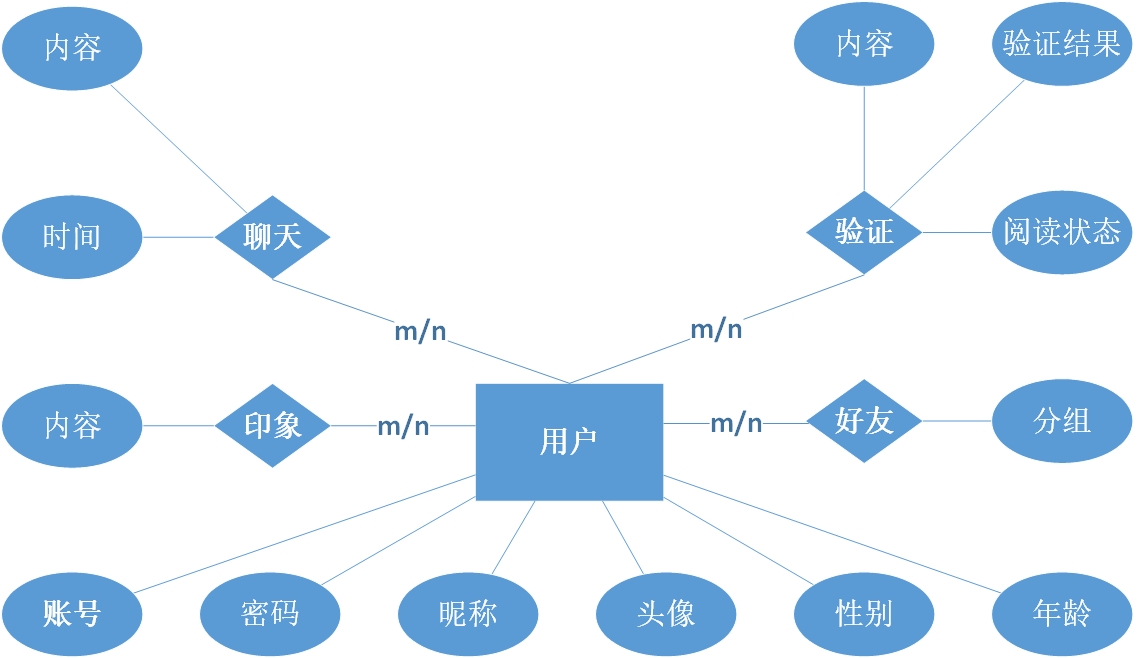
个人资料页面 验证消息界面

项目的页面和原型图过多这里就不一一展示，详情可见压缩文件

二、数据库设计

因为聊天系统的所有功能基本上都是围绕着用户进行的。聊天是用户和用户聊天，添加好友也是用户添加用户，印象管理也是用户给用户添加印象。所以主要的联表操作都和user表有关。这里就先给出整个项目的ER图



根据ER图可以构建数据库的物理设计如下

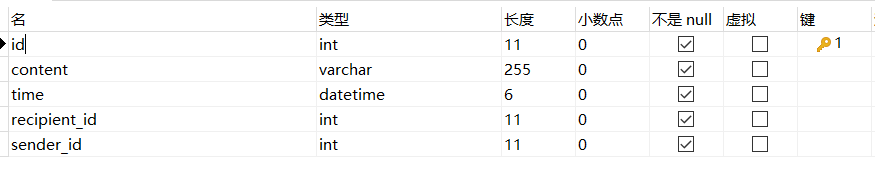
1. 好友关系表 friendship



1. 好友印象表 impression



1. 聊天记录表 record



1. 好友验证表 validation



1. 用户表 user



Redis中的存储结构的说明，因为Redis的Nosql的性质很适合用于存储未读的聊天记录，我是用Redis中的哈希结构进行存储未读消息。每一条记录规则如下键为unread+userId，值为一个Java中的Map<String, Map<String, String>>的类型。Map<String, Map<String, String>>的类型数据格式如下:

senderId: {

nickname:xxx,

content:xxx,

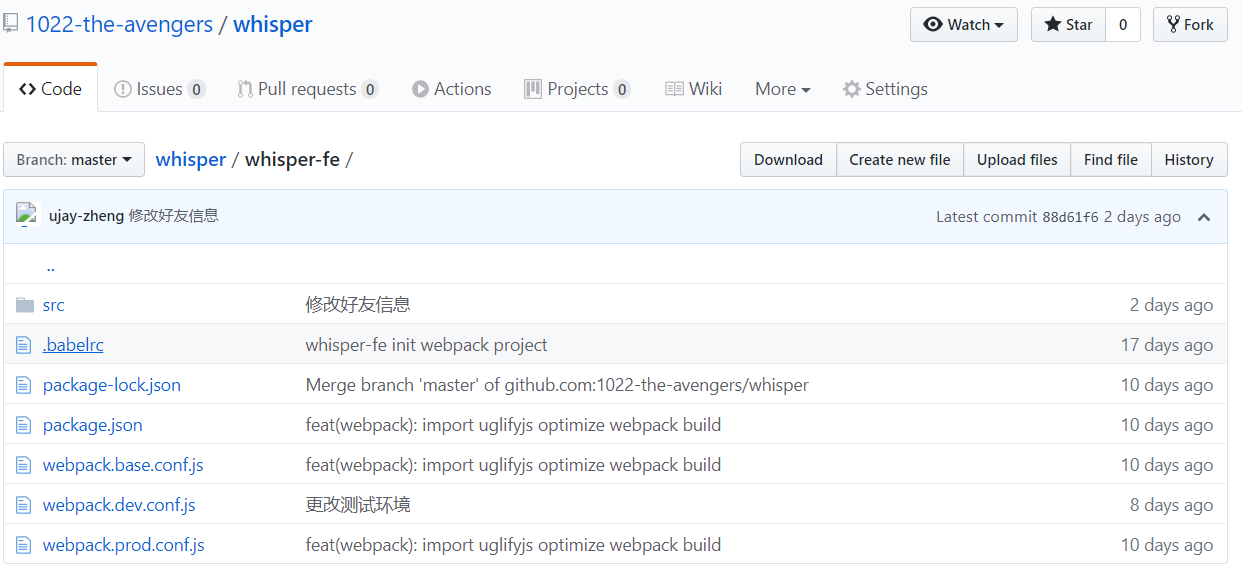
pic:xxx

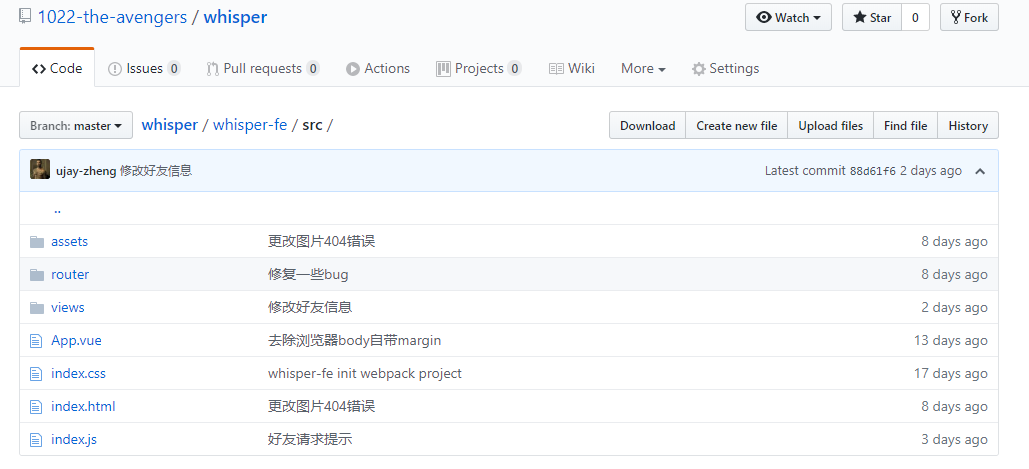
}

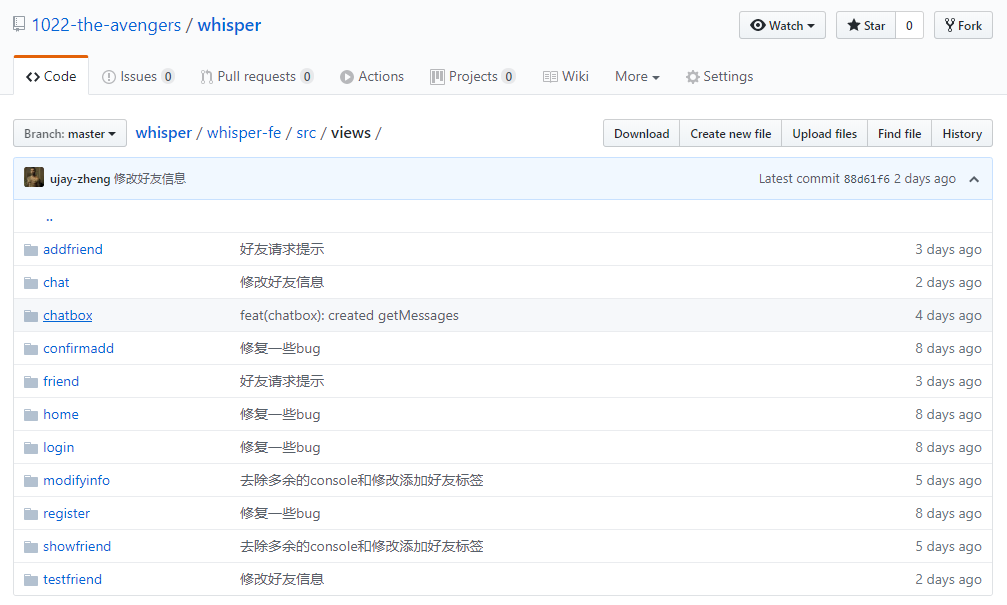
除此之外在Redis中单独存储一个哈希结构，键为unreadNum+userId，值为未读消息数量。

1. 架构设计
   1. 技术栈
      1. 前端
         1. Vue作为前端框架
         2. vue-router进行前端路由管理
         3. webpack开发SPA（单页面应用）
         4. mint-UI作为UI框架
         5. STOMP实现Socket通信的框架
         6. axios发送请求
         7. sass(css预处理器，进行css代码的编写)

前端架构说明:







Webpack搭建项目前端框架，打包生成SPA（单页面）的移动端应用。

1. Webpack配置文件

webpack.base.conf.js为基础配置 一些最基本的loader与plugin都在这里面 webpack.dev.conf.js为开发环境配置，配置了适合开发环境的sourceMap，能快速的定位开发环境代码报错位置 webpack.prod.conf.js为生产环境下配置，关闭了sourceMap，极大的减小了线上环境代码包的大小，启用了UglifyJsPlugin进行代码压缩，使首屏加载速度低于500ms。

1. Src目录下为主要代码， assets文件夹下存储着图片，iconfont等静态资源。Router文件夹下为vue的路由，控制着页面的跳转。 Views文件夹下为视图组件，我们开发的代码主要在这里。每个人负责不同的模块，采用gitflow工作流，几乎没有产生冲突。
   * 1. 后端
        1. Spring Boot作为后端框架
        2. Shiro作为安全验证框架
        3. STOMP协议作为通信协议
        4. Redis存储未读消息
        5. Mysql作为用户信息等的存储
     2. 部署
        1. Docker部署
     3. 测试
        1. Selemiun
        2. textNG
     4. 项目管理平台
        1. github+git

后端架构说明：



1. 后端package说明如下:
   1. config表示配置文件，里面存放诸如Shiro，STOMP的配置
   2. controller放置api接口
   3. dao放置数据库操作类和接口，其下的impl这个package表示JAP扩展实现类
   4. dto存放数据库表的交互数据结构
   5. entity存放数据库表中的映射
   6. pojo：存放普通Java类，一般是存放工具类
   7. processor：存放过滤器，拦截器和监听器
   8. service：供其他模块调用的服务类，采用接口和实现的方式。impl为其实现类
   9. vo和前端交互的数据类（主要用于将数据格式转化为和前端约定的格式）
2. 使用如此结构的原因
   1. 这些package中需要实现config中的文件和entity中的文件，只要这两个package的文件配置了，整个项目就可以运行。之后的编写只需要在其他包中添加自己的代码即可
   2. 在完成了config和entity的文件之后，其他功能的编写只需要注重代码逻辑的实现即可
   3. 当多个人同时写后端的时候，在写自己的功能的时候只需要在对应的package中编写对应的代码即可互不干扰。

四、功能实现

后端实现：

1. 基础配置：
2. RedisConfig

主要是注入RedisTemplate<Object, Object>这个bean

1. Shiro

首先建立自己的匹配器TokenCredentialsMatcher用于验证比较，此比较器实现了CredentialsMatcher接口，实现类doCredentialsMatch方法。我们通过token比较进行验证用户登录状态是否合法。

String token = (String) authenticationToken.getCredentials();  
return TokenManager.verify(token);

然后再建立自己的Reanlm：TokenRealm，自己配置的TokenRealm继承自AuthorizingRealm，并且再构造函数的时候就将我们自己的TokenCredentialsMatcher作为此类的匹配器加入。这样的话进行比较的时候就不会调用默认的比较匹配器而是调用我们自定义的比较器。我们重写supports方法，只有当support方法返回true的时候才会之后的操作。

return token instanceof MyToken

然后再重写doGetAuthenticationInfo方法和doGetAuthorizationInfo方法，但是doGetAuthorizationInfo为授权方法，在我们本系统中并没有使用，所以只给出前者核心代码

String token = (String)authenticationToken.getCredentials();  
Integer userId = TokenManager.getId(token);  
if(userId==null)  
 userId=0;  
return new SimpleAuthenticationInfo(userId, token, getName());

MyToken是我自己建立的Token最主要的方式是重写getPrincipal和getCredentials，前者表示标识，后者表示验证的凭证

@Override  
public Object getPrincipal() {  
 Integer userId = -1;  
 if(TokenManager.verify(this.token)){  
 userId = TokenManager.getId(token);  
 }  
 return userId;  
}  
  
@Override  
public Object getCredentials() {  
 return token;  
}

做好基础文件之后开始编写ShiroConfig，主要完成两个函数一个是注册ShiroFilterFactoryBean的函数，另一个是注册SessionsSecurityManager的函数，在前中我们配置我们的过滤器，以及过滤器拦截的url，在后这种我们进行的工作主要是关闭Shiro自带的session功能。

@Bean  
public ShiroFilterFactoryBean shiroFilterFactoryBean(SecurityManager securityManager) {  
 ShiroFilterFactoryBean shiroFilterFactoryBean = new ShiroFilterFactoryBean();  
  
 shiroFilterFactoryBean.setSecurityManager(securityManager);  
  
 Map<String, Filter> filterMap = new HashMap<>();  
 filterMap.put("verification", new AuthFilter());  
 shiroFilterFactoryBean.setFilters(filterMap);  
 Map<String, String> filterRuleMap = new LinkedHashMap<>();  
 filterRuleMap.put("/verification/\*\*", "verification");  
 shiroFilterFactoryBean.setFilterChainDefinitionMap(filterRuleMap);  
 return shiroFilterFactoryBean;  
}  
  
@Bean  
public SessionsSecurityManager securityManager() {  
 DefaultWebSecurityManager securityManager = new DefaultWebSecurityManager();  
  
 securityManager.setRealm(tokenRealm);  
 //关闭Shiro自带的Session。  
 DefaultSubjectDAO subjectDAO = new DefaultSubjectDAO();  
 DefaultSessionStorageEvaluator defaultSessionStorageEvaluator = new DefaultSessionStorageEvaluator();  
 defaultSessionStorageEvaluator.setSessionStorageEnabled(false);  
 subjectDAO.setSessionStorageEvaluator(defaultSessionStorageEvaluator);  
 securityManager.setSubjectDAO(subjectDAO);  
 return securityManager;  
}

1. 然后再来看过滤器的编写，这里主要是做登录验证的

其主要思路如下：编写的过滤器继承自BasicHttpAuthenticationFilter类，然后重写了isLoginAttempt方法用于判断是否是进行需要登录验证的操作，如果判断返回true则才会进行之后的操作。然后重写executeLogin方法即这个方法进行token的验证，通过我们自定义的TokenReanlm进行验证。如果验证成功则返回true即继续执行原来的请求，如果验证失败则进入onAccessDenied方法。我们通过冲洗onAccessDenied方法将降入此方法后就返回401异常说明，token验证失败。

@Override  
protected boolean isLoginAttempt(ServletRequest request, ServletResponse response) {  
 return ((HttpServletRequest) request).getHeader("Authorization") != null;  
}  
  
@Override  
protected boolean executeLogin(ServletRequest request, ServletResponse response){  
 HttpServletRequest httpServletRequest = (HttpServletRequest) request;  
 String authorization = httpServletRequest.getHeader("Authorization");  
 MyToken token = new MyToken(authorization.substring(7));  
 getSubject(request, response).login(token);  
 return true;  
}  
  
@Override  
protected boolean isAccessAllowed(ServletRequest request, ServletResponse response, Object mappedValue) {  
 try {  
 if (isLoginAttempt(request, response)) {  
 executeLogin(request, response);  
 } else {  
 throw new Exception();  
 }  
 } catch (Exception e){  
 return false;  
 }  
 return true;  
}  
  
@Override  
protected boolean onAccessDenied(ServletRequest request, ServletResponse response){  
 HttpServletResponse httpServletResponse = (HttpServletResponse)response;  
 AuthMessage authMessage = AuthMessage.getAuthMessage(  
 "请先登录", "", "login /auth/login",  
 "/auth/login", "login", "application/json");  
 httpServletResponse.setCharacterEncoding("UTF-8");  
 httpServletResponse.setHeader("Content-Type", "application/json;charset=UTF-8");  
 String ret = JSON.toJSONString(authMessage);  
 httpServletResponse.setStatus(401);  
 try {  
 response.getWriter().write(ret);  
 response.getWriter().close();  
 } catch (IOException ex) {  
 ex.printStackTrace();  
 }  
 return false;  
}

1. WebSocket配置文件：

通过重写registerStompEndpoints方法规定socket建立连接时候的接口。然后再configureMessageBroker方法中设置订阅url和发送消息的前缀

@Override  
public void registerStompEndpoints(StompEndpointRegistry registry) {  
 registry.addEndpoint("/chat").setAllowedOrigins("\*").withSockJS(); //设置连接url并且设置跨域  
}  
  
@Override  
public void configureMessageBroker(MessageBrokerRegistry config) {  
 config.enableSimpleBroker("/subscription"); //设置订阅的url  
 config.setApplicationDestinationPrefixes("/socket"); //设置访问url前缀  
}

1. 数据库实体entity的类在这里就不展示，详情可见代码压缩包，因为这个package中的文件和数据库中的表紧密联系，而数据库中的表在之前已经详细说明过了
2. dto作为entity中的对外交互类这里也不给出具体代码，具体代码可以见代码压缩包
3. controller
4. 登录

首先通过前端传送上来的数据去数据库中寻找数据库中的User，这里调用了UserService的setUser方法，此方法将user放置到UserSerivce对象中的时候同时会去数据库查找账号对应的user，且放入到此类的userInDataBase中，调用UserService的encode方法将当前收到的密码进行加密，然后进行判断，如果userInDataBase为空也就是UserService方法中的isRegistered()返回了false，则表示账号还没被注册，如果checkPassword方法返回false也就是encode之后的密码和userInDataBase中的密码不一样，说明密码错误。如果二者都不满足则登录成功，这里如果发送错误的话就会抛出异常然后有一个全局的异常捕捉函数进行处理

userService.setUser(user);

userService.encode();

if(!userService.isRegistered()){  
throw new HttpException(HttpStatus.UNAUTHORIZED, AuthMessage.getAuthMessage(  
"账号未注册","", "registration /auth/registration",  
"/auth/registration", "registration", "application/json"));

} else if(!userService.checkPassword()){  
throw new HttpException(HttpStatus.UNAUTHORIZED, AuthMessage.getAuthMessage(  
"账号密码不匹配","", "login /auth/login",  
"/auth/login", "login", "application/json"));

}else{  
response.setStatus(200);  
return AuthMessage.getAuthMessage(  
 "登录成功",TokenManager.sign(userService.getUserInDataBase().getId()), "messages /management/messages",  
 "/management/messages", "messages", "application/json");  
}

1. 注册和登录逻辑类似，通过UserService的setUser方法将注册的用户放置到UserService类中，然后进行是否注册过账号密码是否小于8的判断，如果都没有的话，随机分配一个头像，年龄默认为1，性别默认为女，然后进行注册

userService.setUser(user);  
if(userService.isRegistered()){  
 throw new HttpException(HttpStatus.FORBIDDEN, AuthMessage.getAuthMessage(  
 "账号已注册","", "registration /auth/registration",  
 "/auth/registration", "registration", "application/json"));  
}else if(user.getPassword().length()<8){  
 throw new HttpException(HttpStatus.FORBIDDEN, AuthMessage.getAuthMessage(  
 "密码长度小于八位","", "registration /auth/registration",  
 "/auth/registration", "registration", "application/json"));  
}else{  
 userService.encode();  
 userService.initPic();  
 userService.setAge(1);  
 userService.setGender("男");  
 userService.save();  
 response.setStatus(201);  
 return AuthMessage.getAuthMessage(  
 "注册成功", TokenManager.sign(userService.getUser().getId()), "messages /management/messages",  
 "/management/messages", "messages", "application/json");  
}

1. Socket实现实时聊天
2. 规定如下，一个用户之间里一个socket连接，当点击进入聊天框的时候建立连接，离开聊天框的时候断开连接。每个用户订阅的评到就是/subscription/userId。然后每一个用户要发送消息的时候消息格式如下：

{

senderId:xxx,

recipientId:xxx,

content:{“me”:”xxx”}

}

后端接收到消息之后将content转化为{“he”,”xxx”}这是因为前端渲染需要，然后通过recipientId可以知道要发送得了路径，可以通过MessageingTemplate.converAndSend来主动推送消息。除此之外还要将消息送入Reids和Mysql。

@MessageMapping("/chat")  
 public void sayHello(ChatMessage message){  
 User sender = userService.findById(message.getSenderId());  
 User recipient = userService.findById(message.getRecipientId());  
 if(sender==null||recipient==null)  
 return;  
 Map<String, String> content = message.getContent();;  
 String destination = "/subscription/" + recipient.getId();  
 String origin = "/subscription/" + sender.getId();  
 if(relationshipService.isFriend(recipient, sender)){  
 Record record = new Record(content.get("my"), new Date(), sender, recipient);  
 userMessage.save(record);  
 userMessage.addMessage(recipient.getId(), sender.getId(), sender.getNickname(), content.get("my"), sender.getPic());  
 messagingTemplate.convertAndSend(destination, new HashMap<String, Object>(){{  
 put("senderId", String.valueOf(sender.getId()));  
 put("senderPic", sender.getPic());  
 put("senderName", sender.getNickname());  
 put("content", new HashMap<String, String>(){{  
 put("he", content.get("my"));  
 }}  
 );  
 }});  
 messagingTemplate.convertAndSend(origin, new HashMap<String, String>(){{  
 put("content","发送成功");  
 }});  
 }else{  
 messagingTemplate.convertAndSend(origin, new HashMap<String, String>(){{  
 put("content","对方不是你好友");  
 }});  
 }  
 }

1. 消息盒子的消息获取

因为消息盒子是采用轮询方式获取，我们通过UserService的无参数的setUser方法可以将Shiro框架中通过验证的用户放入到UserService的user中去，然后我们获取到Redis的所有存储消息并且返回给前端，返回给前端的数据包括Redis中的未读消息（发送方头像，姓名，Id，内容），以及对于每一个发送方的未读消息数量，以及自己的头像，和基础信息。

@RequestMapping(value = "/index", method = {RequestMethod.GET})  
public InfoMessage getIndex(@RequestParam Integer customer){  
 User user = userService.setUser();  
 if(!(customer==null||customer==-1)){  
 userMessage.readMessage(user.getId(),customer);  
 }  
 Map<Object, Object> unreadMessages = userMessage.getAllUnread(user.getId());  
 Map<Object, Object> unreadNum = userMessage.getAllUnreadNum(user.getId());  
 LinkMessage linkMessage = new LinkMessage(  
 "",  
 "",  
 "",  
 "");  
 return new InfoMessage("获取成功", unreadMessages, unreadNum, new UserInformation(  
 user.getId(), user.getNickname(), user.getPic(), user.getGender(), user.getAge()  
 ), linkMessage);  
}

1. 获取所有的历史记录

这个逻辑也比较简单，也是获取到当前登录的用户之后，去查找当前登录的用户和对方的所有聊天记录，然后将消息转化为前端要求的格式返回即可。这里不给出代码，具体可见代码压缩包

@RequestMapping(value = "/history/download", method = {RequestMethod.GET})  
public ResponseEntity<FileSystemResource> download(@RequestParam Integer customer){  
 User user = userService.setUser();  
 List<RecordDto> records = recordRepository.getChatRecord(user.getId(), customer, -1);  
 HttpHeaders headers = new HttpHeaders();  
 headers.add("Cache-Control", "no-cache, no-store, must-revalidate");  
 headers.add("Content-Disposition", "attachment; filename=" + System.currentTimeMillis() + ".xls");  
 headers.add("Pragma", "no-cache");  
 headers.add("Expires", "0");  
 headers.add("Last-Modified", new Date().toString());  
 headers.add("ETag", String.valueOf(System.currentTimeMillis()));  
 try {  
 File file = new File("history/"+user.getId()+"with"+customer+".txt"); // 相对路径，如果没有则要建立一个新的output.txt文件  
 if(!file.exists()) {  
 file.createNewFile(); // 创建新文件,有同名的文件的话直接覆盖  
 }  
 FileWriter writer = new FileWriter(file);  
 BufferedWriter out = new BufferedWriter(writer);  
 for(RecordDto record: records){  
 out.write("at "+ record.getTime()+"|"+record.getSenderId()+"-->"+record.getRecipientId()+"="+record.getContent()+"\r\n");  
 }  
 out.flush(); // 把缓存区内容压入文件  
 out.close();  
 return ResponseEntity.ok().headers(headers) .contentLength(file.length()).contentType(MediaType.parseMediaType("application/octet-stream")) .body(new FileSystemResource(file));  
 } catch (IOException e) {  
 e.printStackTrace();  
 }  
 File file = new File("history/"+user.getId()+"with"+customer+".txt");  
 return ResponseEntity.ok().headers(headers) .contentLength(file.length()).contentType(MediaType.parseMediaType("application/octet-stream")) .body(new FileSystemResource(file));  
}

1. 历史记录下载

下载分为两部分，第一步为查询聊天记录，查询的流程在上面已经说过了，这里不再赘述，主要是第二部。我们更改httpheader，加入文件流的控制。然后将我们查询到的聊天记录根据规定格式写入到Java的File类中，然后将这个File类返回给前端即可。

1. 删除历史记录

删除历史记录的逻辑很简单，知道知道获取到对方Id和自己的Id然后将数据库中的所有这两个人的记录都删除即可(sender=a，customer=b||sender=b，customer=a)这里就不给出代码，具体见压缩包

前端实现：

**Home主界面的编写**

使用mint-ui的mt-tabbar标签来实现消息盒子组件与好友列表组件之间的切换，同时添加了图片实现点击可以切换状态。

<mt-tabbar v-model="selected">

      <mt-tab-item id="chatbox">

        <img slot="icon" v-show="selected!=='chatbox'" src="/assets/message.png" />

        <img slot="icon" v-show="selected=='chatbox'" src="/assets/message\_click.png" />

        消息

      </mt-tab-item>

      <mt-tab-item id="friend">

        <img slot="icon" v-show="selected!=='friend'" src="/assets/friend.png" />

        <img slot="icon" v-show="selected=='friend'" src="/assets/friend\_click.png" />

        好友列表

      </mt-tab-item>

    </mt-tabbar>

**好友列表的渲染**

根据后端传来的双层数组进行列表渲染，使用Vue中value 和 key的列表渲染方式，将不同分组的人员分开渲染，同时使用URL的传值方式，使点击不同的用户，好友展示中展示不同的好友信息

      <div v-for="(value,key) in friends" :key="key">

        <mt-cell :title="key"></mt-cell>

        <mt-cell

          v-for="(value2,key2) in value"

          :key="key2"

          :to="'/home/showfriend?group='+key+'&id='+key2"

        >

          <span>{{value2.nickname}}</span>

          <img slot="icon" v-bind:src="value2.pic" width="28" height="28" />

        </mt-cell>

      </div>

**好友验证提醒功能**

使用http轮询，查看是否有好友验证，若有则使用红点标记出来

    this.getNotRead = setInterval(function() {    //轮询请求是否有未读消息

      that.axios

        .get("/verification/user/validation-amount")

        .then(response => {

          that.notReadNum = response.data.data;

        })

        .catch(err => {

          console.log(error); //异常

        });

    }, 2000);

**添加好友模块**

使用mint-ui的搜索框组件进行昵称的输入，同时使用两个下拉框实现年龄和性别的筛选，使用Vue的原生回车事件进行提交

    <mt-search

      v-model="searchContent"

      cancel-text="取消"

      placeholder="输入昵称搜索"

      @keyup.enter.native="search"

    ></mt-search>

**发送好友请求模块**

模块初始化时，使用LocalStorage取出存储的用户信息，使用axios的错误捕获来判断是否成功发送好友请求

        .catch(err => {

          MessageBox.alert("不能重复添加，或者添加自己").then(action => {});

        });

**个人信息修改模块**

个人修改的信息通过表单获取，同时使用正则表达式判断是否符合要求。图片文件则进行后缀以及大小的合法性判断，不符合要求则不进行修改

      var regPos = /^\d+$/;

      if (

        this.info.nickname == "" ||

        this.info.account == "" ||

        this.info.pic == "" ||

        this.info.gender == ""

      ) {

        MessageBox.alert("所填项不能为空").then(action => {});

        return false;

      }else if(this.info.gender!="男" && this.info.gender !="女"){

        MessageBox.alert("性别只能为男或女").then(action => {});

        return false;

      }else if(!regPos.test(this.info.age)){

        MessageBox.alert("年龄只能为非负整数").then(action => {});

        return false;

      }

     var img = e.target.files[0];

      if (

        img.type !== "image/jpeg" &&

        img.type !== "image/png" &&

        img.type !== "image/gif"

      ) {

        MessageBox.alert("请选择图片文件").then(action => {});

        return false;

      } else if (img.size > 1024 \* 30) {

        console.log(img.size);

        MessageBox.alert("选择小于30kb的图片").then(action => {});

        return false;

      }

**好友展示模块**

**通过mint-ui的上拉框进行功能的选择，可以选择删除好友，添加标签，移动好友的功能，添加标签时与以往的标签进行比较，若有相同的则进行去重操作**

      <mt-button icon="more" slot="right" @click.native="actionSheet"></mt-button>

          var add = {};

          add.targetId = this.info.id;

          add.contents = this.info.impressions;

          const impressions = new Set(add.contents);

          if(!impressions.has(value)){

            this.info.impressions.push(value);

          }

          console.log(impressions);

          add.contents = [...impressions.values()];

**好友验证模块**

使用两个列表渲染，展示自己发送的请求，以及别人发送给你的请求，同时组件被创建时，自动请求后端，将所有的未读消息置为已读消息

    this.axios    //标记好友请求已读

      .post("/verification/user/validation-reading")

      .then(response => {

      })

      .catch(err => {

        console.log(err); //异常

      });

**聊天模块**

Connection方法，建立stomp连接，并且为了防止websocket连接中断，我采用了心跳保活技术，每60s发送一条无用消息，确保连接正常。 并且推出页面会断开socket连接，以减小服务器压力。



发送消息，进行了内容判断 发送内容不能为空或者内容长度不能超过200

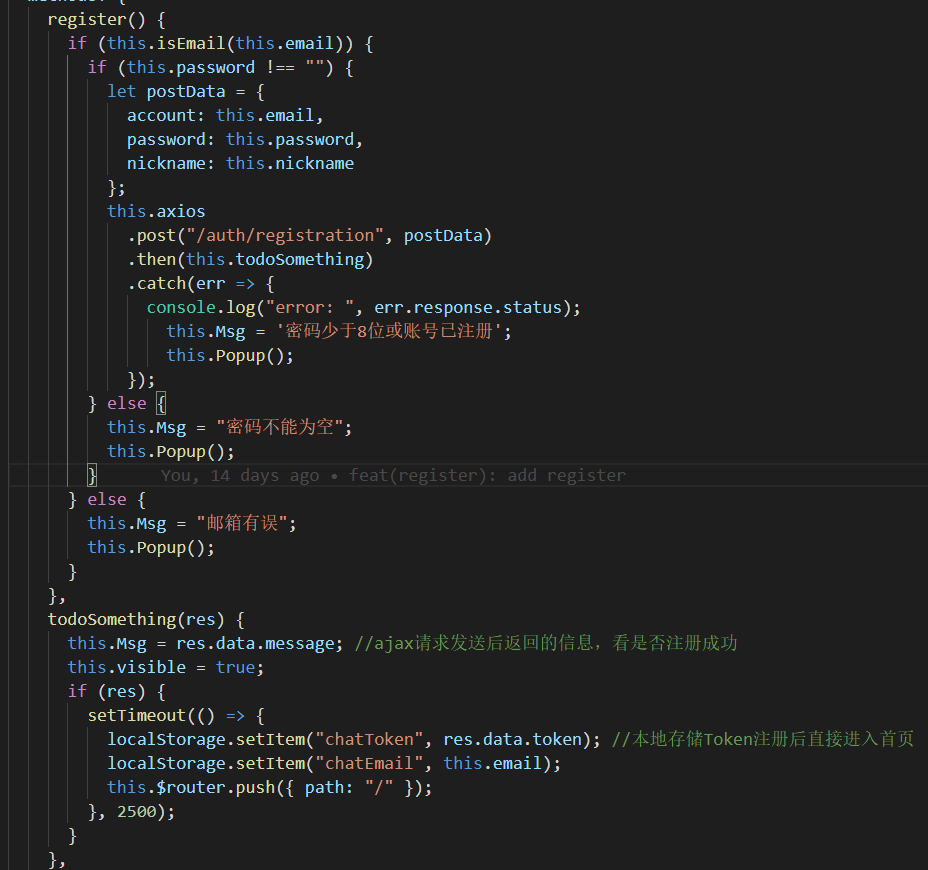


**登录注册模块**

登录后 将token与个人信息相关的数据存入localstorage内，退出登录就直接清除localstorage内全部数据。



注册后，直接跳转到主页面，有良好的错误处理机制。



**测试模块**

**TESTNG文件配置**

使用xml配置要测试的模块，包括注册功能，登录功能，添加好友功能，聊天功能等

*<?***xml version="1.0" encoding="UTF-8"***?>***<!DOCTYPE suite SYSTEM "http://testng.org/testng-1.0.dtd"*>***<**suite name="Link聊天自动化测试"**>  
 <**test name="测试"**>  
 <**classes**>  
 <**class name="register.Register"**/>  
 <**class name="login.Login"** />  
 <**class name="addfriend.Addfriend"**/>  
 <**class name="impression.Impression"**/>  
 <**class name="chat.Chat"**/>  
 <**class name="deletefriend.Deletefriend"**/>  
 </**classes**>  
 </**test**>  
</**suite**>

**浏览器驱动公共代码**

公共代码中实现公共操作，包括启动浏览器驱动，浏览器驱动关闭，共享变量DRIVER，供所有模块使用

@BeforeClass  
**public static void** setUp() **throws** Exception {  
  
 System.***out***.println(**"启动浏览器"**);  
 System.*setProperty*(**"webdriver.chrome.driver"**,**"./src/chromedriver.exe"**);  
 *driver* = **new** ChromeDriver();  
 *driver*.manage().timeouts().implicitlyWait(10, TimeUnit.***SECONDS***);  
}

**注册模块自动化测试代码**

使用selenium进行浏览器的操作，输入待注册的账号，注册完成后跳转到主页面进行登录，判断是否注册成功

Main.*setUp*();  
Main.*driver*.get(**RegisterURL**);  
Main.*driver*.findElement(By.*name*(**"user\_register"**)).sendKeys(**this**.**email**);  
Main.*driver*.findElement(By.*name*(**"user\_name"**)).sendKeys(**this**.**name**);  
Main.*driver*.findElement(By.*name*(**"user\_pass"**)).sendKeys(**this**.**psw**);  
Main.*driver*.findElement(By.*cssSelector*(**".btn\_register"**)).click();  
Thread.*sleep*(3000);  
String current\_url = Main.*driver*.getCurrentUrl();  
**try** {  
 Assert.*assertEquals*(current\_url,**this**.**SuccessURL**,**"注册验证失败！！"**);  
}**catch** (Exception e){  
 Main.*tearDown*();  
}  
Main.*tearDown*();

**登录模块自动化代码**

在输入框中输入账号与密码，点击登录，页面加载完成后判断是否转跳到主页面

Main.setUp();  
Main.driver.get(Main.BaseURL);  
WebElement email = Main.driver.findElement(By.name(**"user\_login"**));  
WebElement psw = Main.driver.findElement(By.name(**"user\_pass"**));  
WebElement button = Main.driver.findElement(By.cssSelector(**".btn\_login"**));  
email.sendKeys(Main.email);  
psw.sendKeys(Main.psw);  
button.click();

**印象模块自动化测试代码**

根据提供的账号登录到主页面，点击好友框，添加印象，然后匹配数据库中的印象表，查看是否添加成功

Main.*driver*.findElement(By.*xpath*(**"//\*[text()='康王']"**)).click();  
Main.*driver*.findElement(By.*className*(**"mintui-more"**)).click();  
Thread.*sleep*(2000);  
Main.*driver*.findElement(By.*xpath*(**"//\*[text()='添加标签']"**)).click();  
Main.*driver*.findElement(By.*cssSelector*(**"input"**)).sendKeys(**this**.**impre**);  
Main.*driver*.findElement(By.*xpath*(**"//\*[text()='确定']"**)).click();

**好友删除自动化测试代码**

点击页面删除好友按钮，退回到好友列表界面，查看是否删除该好友

Main.*driver*.findElement(By.*className*(**"mintui-more"**)).click();  
Main.*driver*.findElement(By.*xpath*(**"//\*[text()='删除好友']"**)).click();  
Main.*driver*.findElement(By.*xpath*(**"//\*[text()='确定']"**)).click();  
Main.*driver*.findElement(By.*xpath*(**"//\*[text()='确定']"**)).click();

**try**{  
 WebElement a = Main.*driver*.findElement(By.*xpath*(**"//\*[text()='康王']"**));  
 Thread.*sleep*(4000);  
}**catch** (Exception e){  
 Main.*tearDown*();  
}  
Main.*tearDown*();

**聊天界面自动化测试代码**

点击预订的好友，点击发送消息，跳转到消息界面，输入框中输入消息，点击发送，查看页面中是否渲染出该条消息，并登陆另一账号查看是否接受到该消息

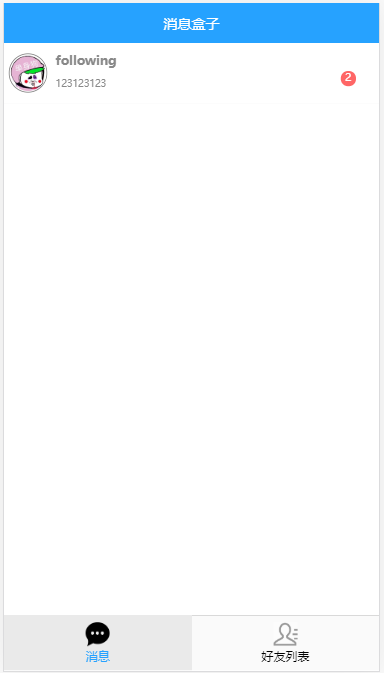
Main.*driver*.findElement(By.*className*(**"mint-button--large"**)).click();  
Main.*driver*.findElement(By.*className*(**"footer\_inpuit"**)).sendKeys(**"呆呆呆呆"**);  
Main.*driver*.findElement(By.*className*(**"footer\_button"**)).click();  
**try**{  
 WebElement a = Main.*driver*.findElement(By.*xpath*(**"//\*[text()='确定']"**));  
}**catch**(Exception e){  
 Assert.*fail*(**"发送消息失败！！"**);  
 Main.*tearDown*();  
}

五、成果展示

登录页面 注册页面

消息盒子页面(无消息) 消息盒子（有消息）

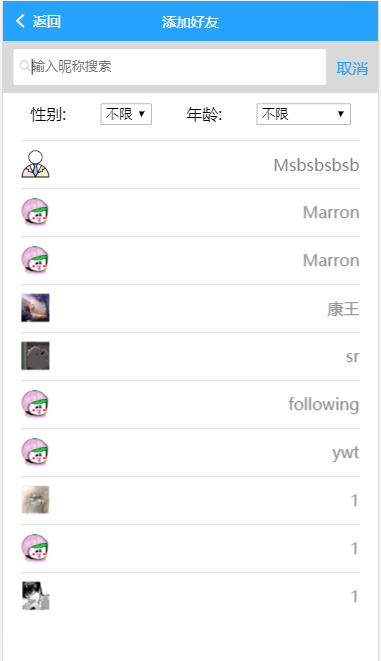
好友盒子（无朋友验证） 好友盒子(有朋友验证)

在好友列表中点击好友之后显示的页面 点击右上角的点之后的页面

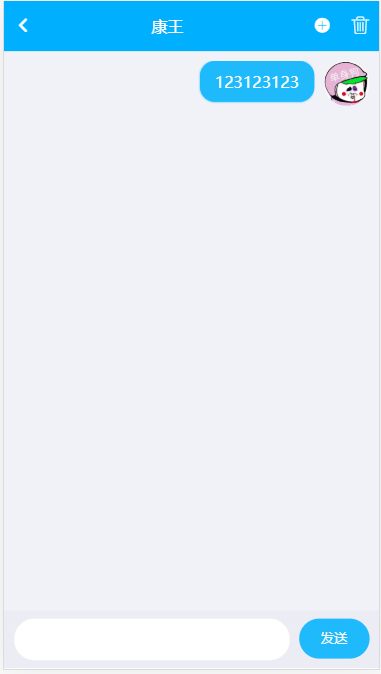
个人信息 搜索结果界面

搜索点击好友后的界面 可以发送验证消息（带信息）

聊天界面



验证消息