# 会员系统

## 总体目标

搭建环境

会员登录注册

发起众筹项目

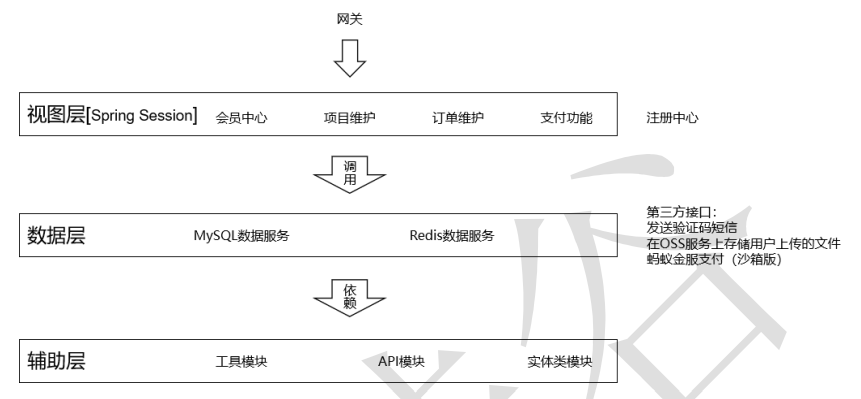
展示众筹项目

支持众筹项目

订单

支付

## 整体架构



## 工程概述

父工程、聚合工程：唯一的pom工程 crowd-07-member-parent

注册中心：crowd-08-member-eureka

实体类模块：crowd-09-member-entity

MySQL 数据服务：crowd-10-member-mysql-provider

Redis 数据服务：crowd-11-member-redis-provider

会员中心：crowd-12-member-authentication-consumer

项目维护：crowd-13-member-project-consumer

订单维护：crowd-14-member-order-consumer

支付功能：crowd-15-member-pay-consumer

网关：crowd-16-member-zuul

API 模块：crowd-17-member-api

# 环境搭建

## 约定

1.包名约定：新创建的包都作为 com.atguigu.crowd 的子包

2.主启动类类名：CrowdMainClass

3.端口号：

eureka 1000

mysql-provider 2000

redis-provider 3000

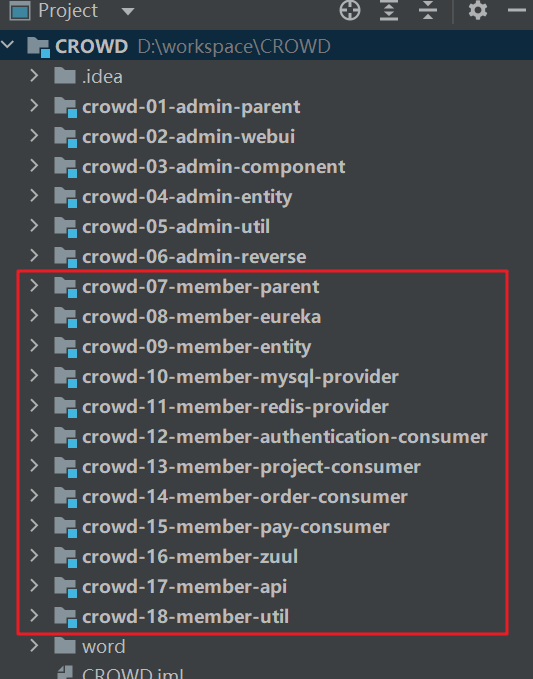
authentication-consumer 4000

project-consumer 5000

order-consumer 7000

pay-consumer 8000

zuul 80



## eureka工程

### 依赖

<dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>  
 </dependency>  
</dependencies>

### 主启动类

加上@EnableEurekaServer注解

@EnableEurekaServer  
@SpringBootApplication  
public class CrowdMainClass {  
 public static void main(String[] args) {  
 SpringApplication.*run*(CrowdMainClass.class);  
 System.*out*.println("Hello world!");  
 }  
}

### application.yml

server:  
 port: 1000  
  
spring:  
 application:  
 name: eureka  
  
eureka:  
 instance:  
 *#配置当前注册中心的服务主机地址* hostname: localhost  
 client:  
 *# 自己就是注册中心，所以自己不注册自己* register-with-eureka: false  
 *# 自己就是注册中心，所以不需要“从注册中心取回信息”* fetch-registry: false  
 service-url:  
 *#客户端（consumer和provider）访问 Eureka 时使用的地址* defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka

## entity工程

### 实体类的进一步划分

1.VO：View Object 视图对象

用途 1：接收浏览器发送过来的数据

用途 2：把数据发送给浏览器去显示 

2.PO：Persistent Object 持久化对象

用途 1：将数据封装到 PO 对象存入数据库

用途 2：将数据库数据查询出来存入 PO 对象

所以 PO 对象是和数据库表对应，一个数据库表对应一个PO 对象

3.DO：Data Object 数据对象

用途 1：从 Redis 查询得到数据封装为 DO 对象

用途 2：从 ElasticSearch 查询得到数据封装为 DO 对象

用途 3：从 Solr 查询得到数据封装为 DO 对象 ……

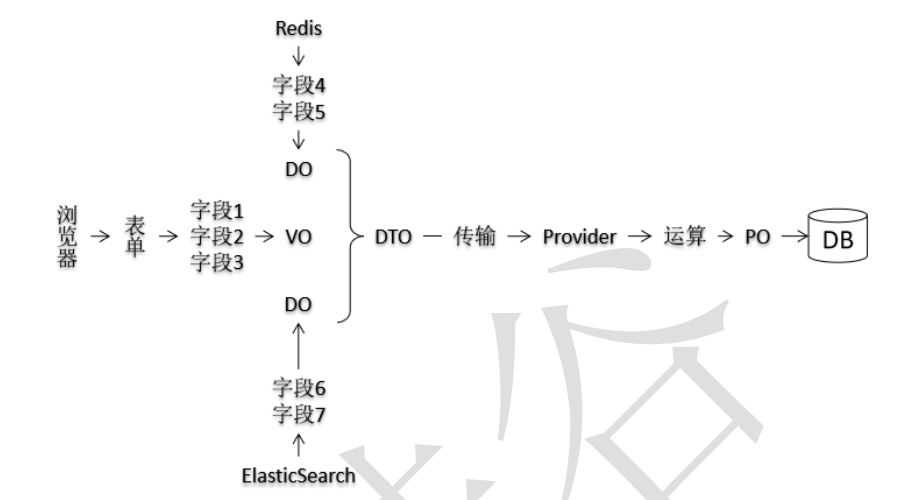
从中间件或其他第三方接口查询到的数据封装为 DO 对象

4.DTO：Data Transfer Object 数据传输对象

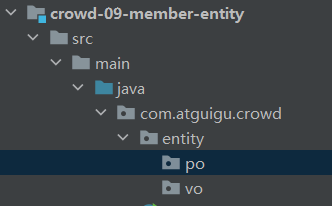
用途 1：从 Consumer 发送数据到 Provider

用途 2：Provider 返回数据给 Consumer

使用 org.springframework.beans.BeanUtils.copyProperties(Object, Object)在不同实体类之间复制属性。 MemberVO→复制属性→MemberPO



### 创建包



## MySQL工程

抽取整个项目中所有针对数据库的操作。

### 建表语句

成员信息表

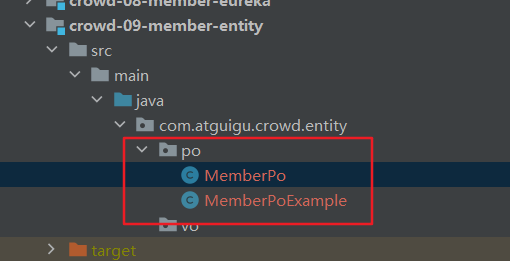
create table t\_member(  
 id int(11) not null auto\_increment,  
 loginacct varchar(255) not null,  
 userpswd varchar(255) not null,  
 username varchar(255),  
 email varchar(255),  
 authstatus int(4) comment '实名认证状态 0-未认证，1-申请中，2-已认证',  
 usertype int(4) comment '0-个人，1-企业',  
 realname varchar(255),  
 cardnum varchar(255),  
 accttype int(4) comment '0-企业，1-个体，2-个人，3-政府',  
 primary key(id)  
);

### 逆向工程

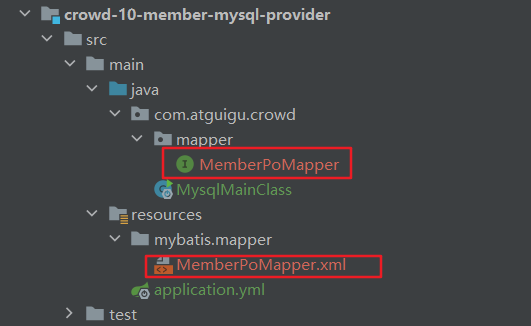
1.逆向工程，generatorConfig.xml配置文件修改表名和类名



2.实体类放到entity工程的po包下



3.mapper接口和mapper.xml放到mybatis包下



### 依赖

<dependencies>  
 <!-- 指定groupId和artifactId即可，版本已在父工程中定义 -->  
  
 <!--SpringBoot测试-->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 </dependency>  
 <!--MyBatis -->  
 <dependency>  
 <groupId>org.mybatis.spring.boot</groupId>  
 <artifactId>mybatis-spring-boot-starter</artifactId>  
 </dependency>  
 <!--mysql驱动-->  
 <dependency>  
 <groupId>mysql</groupId>  
 <artifactId>mysql-connector-java</artifactId>  
 </dependency>  
 <!--连接池-->  
 <dependency>  
 <groupId>com.alibaba</groupId>  
 <artifactId>druid</artifactId>  
 </dependency>  
 <!-- 对外暴露服务 -->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <!-- 作为客户端访问 Eureka 注册中心 -->  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
 </dependency>  
  
 <!-- 为了能够使用实体类 -->  
 <dependency>  
 <groupId>com.atguigu.crowd</groupId>  
 <artifactId>crowd-09-member-entity</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 </dependency>  
 <!-- 为了能够使用工具类 -->  
 <dependency>  
 <groupId>com.atguigu.crowd</groupId>

<artifactId>crowd-18-member-util</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 </dependency>  
  
</dependencies>

### 主启动类

注意加上@MapperScan注解，扫描mapper接口所在包

//扫描mapper接口所在包  
@MapperScan("com.atguigu.crowd.mapper")  
@SpringBootApplication  
public class CrowdMainClass {  
 public static void main(String[] args) {  
 SpringApplication.*run*(CrowdMainClass.class);  
 System.*out*.println("Hello world!");  
 }  
}

### application.yml

配置mybatis、数据源、eureka和日志等

server:  
 port: 2000  
spring:  
 application:  
 *#配置当前微服务名称，在Eureka注册中心，是通过微服务名称来查找调用微服务* name: mysql  
  
 *#数据源配置* datasource:  
 name: mydb  
 driver-class-name: com.mysql.cj.jdbc.Driver  
 type: com.alibaba.druid.pool.DruidDataSource  
 url: jdbc:mysql://localhost:3306/crowd?useUnicode=true&characterEncoding=UTF-8&serverTimezone=UTC  
 username: root  
 password: 123456  
*#MyBatis配置*mybatis:  
 mapper-locations: classpath:/mybatis/mapper/\*Mapper.xml  
*#日志*logging:  
 level:  
 com.atguigu.crowd.mapper: debug  
 com.atguigu.crowd.test: debug  
*#eureka配置*eureka:  
 client:  
 service-url:  
 *#Eureka客户端访问Eureka服务器时使用的地址* defaultZone: http://localhost:1000/eureka/

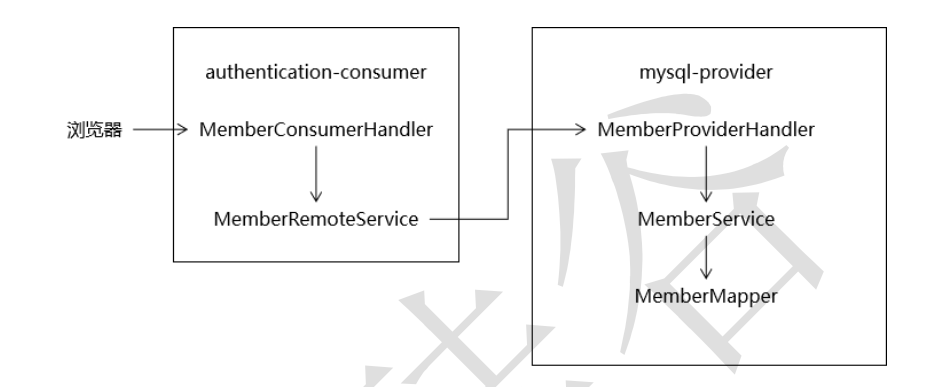
### 测试类

@RunWith(SpringRunner.class)  
@SpringBootTest  
public class MyBatisTest {  
 Logger logger = LoggerFactory.*getLogger*(MyBatisTest.class);  
  
 @Autowired  
 private DataSource dataSource;  
  
 @Autowired  
 private MemberPoMapper mapper;  
  
 @Test  
 public void testConnetion() throws SQLException {  
 Connection connection = dataSource.getConnection();  
 logger.info("============"+connection.toString());  
 }  
  
 @Test  
 public void testDatabase(){  
 MemberPo memberPo = mapper.selectByPrimaryKey(1001);  
 logger.info("======="+memberPo.toString());  
 }  
  
}

### MySQL工程对外服务

1.对外的服务放在mysql工程

2.接口和实现放在api工程



#### api工程

1.引入依赖

引入util工程是为了ResultEntity类，引入entity工程是为了MemberPo类

<dependencies>  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-openfeign</artifactId>  
 </dependency>  
 <dependency>  
 <groupId>com.atguigu.crowd</groupId>  
 <artifactId>crowd-18-member-util</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 </dependency>  
 <dependency>  
 <groupId>com.atguigu.crowd</groupId>  
 <artifactId>crowd-09-member-entity</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 </dependency>  
</dependencies>

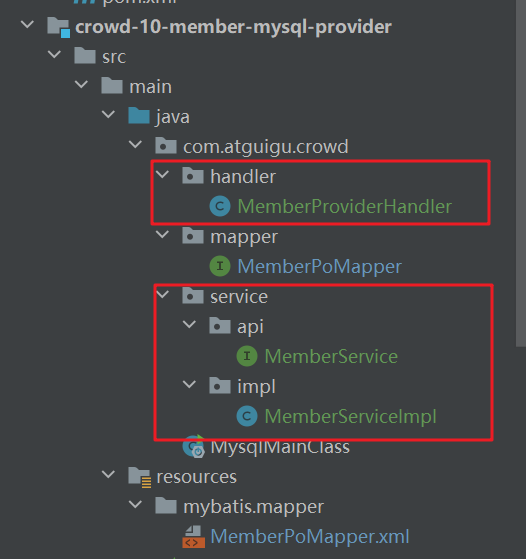
2.service类

注意要加 @FeignClient 注解，远程接口要和provider工程的方法保持一致

即要和mysql-provider工程中的控制器的方法一致

//@FeignClient注解表示和一个provider对应，value属性指定要调用的Provider的微服务名称;  
// fallbackFactory属性指定 consumer 调用 provider 时如果失败所采取的备用方案  
@FeignClient(value = "mysql-provider")  
public interface MySQLRemoteService {  
 //远程调用的接口方法  
 //注意：这里的方法要和 Provider中的具体实现的方法的声明要一致  
 //@RequestMapping注解的url地址、@RequestParam、@RequestBody、@PathVariable两边一致  
 @RequestMapping("/get/member/by/loginacct/remote")  
 ResultEntity<MemberPo> getMemberByLoginacctRemote(@RequestParam("loginacct") String loginacct);  
}

#### mysql工程



1.控制器方法

@RestController  
public class MemberProviderHandler {  
 @Autowired  
 private MemberService memberService;  
  
 //provider工程的远程方法，要和api工程有 @FeignClient 注解标记的方法一致  
 @RequestMapping("/get/member/by/loginacct/remote")  
 ResultEntity<MemberPo> getMemberByLoginacctRemote(@RequestParam("loginacct") String loginacct){  
 //*todo 异常处理* MemberPo member = memberService.getMemberByLoginacct(loginacct);  
 return ResultEntity.*successWithData*(member);  
 }  
}

2.service接口

public interface MemberService {  
  
 MemberPo getMemberByLoginacct(String loginacct);  
}

3.service实现类，加上事务控制

// 在类上使用@Transactional(readOnly = true)针对查询操作设置事务属性  
@Transactional(readOnly = true)  
@Service  
public class MemberServiceImpl implements MemberService {  
 @Autowired  
 private MemberPoMapper mapper;  
  
 @Override  
 public MemberPo getMemberByLoginacct(String loginacct) {  
 return mapper.selectByLoginAcct(loginacct);  
 }  
}

4.mapper接口

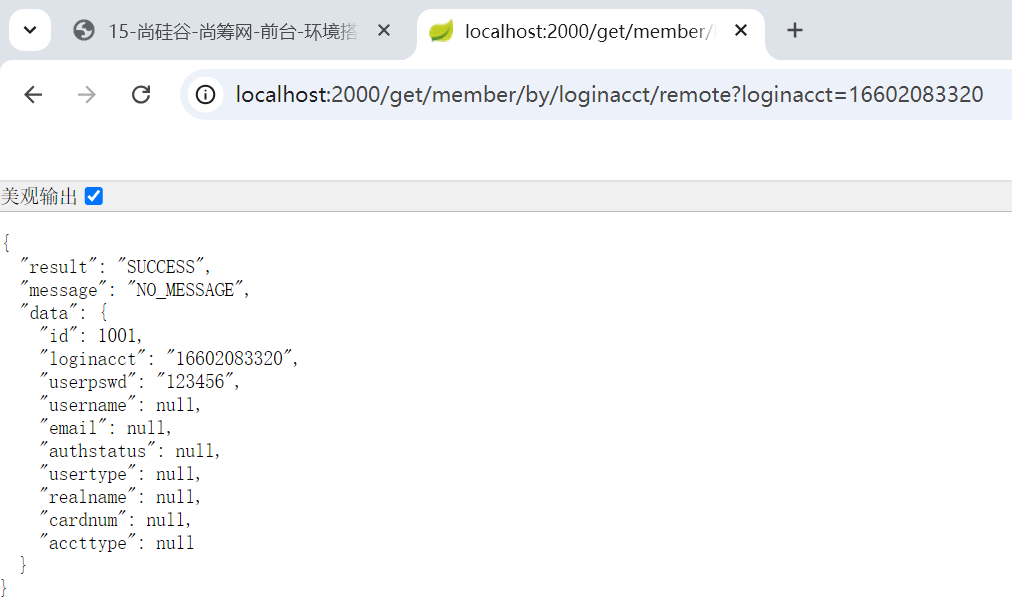
*/\*\*  
\* 根据登录账号查询成员信息  
\** ***@param*** *loginAcct  
\** ***@return****\*/*MemberPo selectByLoginAcct(@Param("loginAcct") String loginAcct);

5.mapper.xml的sql

<select id="selectByLoginAcct" parameterType="java.lang.String" resultMap="BaseResultMap">  
 select  
 <include refid="Base\_Column\_List" />  
 from t\_member  
 where loginacct = #{loginAcct}  
</select>

#### 测试

http://localhost:2000/get/member/by/loginacct/remote?loginacct=16602083320



## Redis工程

抽取项目中所有访问 Redis 的操作。

### 依赖

<dependencies>  
 <!-- 整合 Redis -->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-data-redis</artifactId>  
 </dependency>  
 <!-- 测试 -->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-test</artifactId>  
 <scope>test</scope>  
 </dependency>  
 <!-- 对外暴露服务 -->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <!-- 作为客户端访问 Eureka 注册中心 -->  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
 </dependency>  
 <!-- 为了能够使用实体类 -->  
 <dependency>  
 <groupId>com.atguigu.crowd</groupId>  
 <artifactId>crowd-09-member-entity</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 </dependency>  
 <!-- 为了能够使用工具类 -->  
 <dependency>  
 <groupId>com.atguigu.crowd</groupId>  
 <artifactId>crowd-18-member-util</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 </dependency>  
</dependencies>

### 主启动类

@SpringBootApplication  
public class CrowdMainClass {  
 public static void main(String[] args) {  
 SpringApplication.*run*(CrowdMainClass.class);  
 System.*out*.println("Hello world!");  
 }  
}

### application.yml

主要配置：微服务名称、redis、eureka

server:  
 port: 3000  
  
spring:  
 application:  
 *#配置当前微服务名称，在Eureka注册中心，是通过微服务名称来查找调用微服务* name: redis-provider  
  
 *#redis相关配置* redis:  
 *#默认使用 0 号数据库* database: 0  
 *#redis服务器地址，如果redis的配置文件的 bind属性绑定了地址，需要用该地址* host: 192.168.64.133  
 *#rdis默认端口6379* port: 6379  
 password: 123456  
 timeout: 5000  
 jedis:  
 pool:  
 *#连接池中最大空闲连接，默认值是8* max-idle: 500  
 *#连接池中的最小空闲连接，默认值是0* min-idle: 50  
 *#如果赋值为 -1，则表示不限制；如果pool已经分配了maxActive个jedis实例，则此时pool的状态为exhansted（耗尽）* max-active: 1000  
 *#等待可用连接的最大时间，单位毫秒，默认值为 -1，表示永不超时。如果超过等待时间，则直接抛出 JedisConnectionException* max-wait: 2000  
  
*#eureka配置*eureka:  
 client:  
 service-url:  
 *#Eureka客户端访问Eureka服务器时使用的地址* defaultZone: http://localhost:1000/eureka/

### 测试类

@RunWith(SpringRunner.class)  
@SpringBootTest  
public class RedisTest {  
 Logger logger = LoggerFactory.*getLogger*(RedisTest.class);  
 @Autowired  
 private StringRedisTemplate redisTemplate;  
 @Test  
 public void testRedis(){  
 ValueOperations<String, String> operations = redisTemplate.opsForValue();  
 operations.set("name1","shuyun");  
 }  
}

### redis工程对外服务

#### api工程接口

提供外部服务：string类型的get、set方法，分带超时时间和不带超时时间的方法，以及删除key的操作

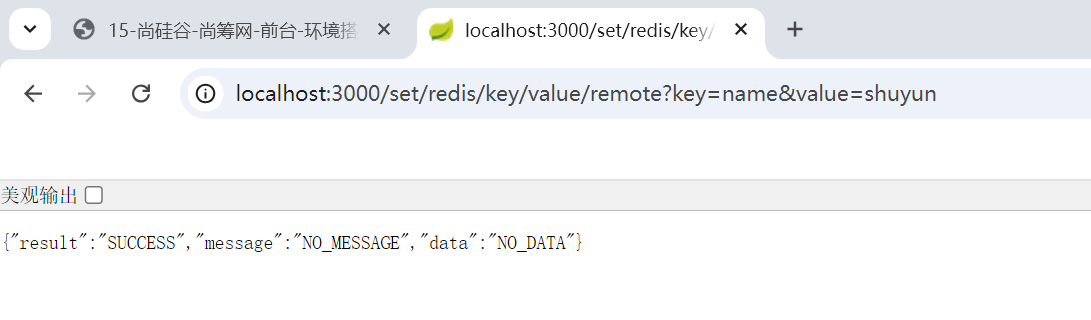
*/\*\*  
\** ***@author*** *shuyun  
\** ***@date*** *2024-10-11 08:52:27  
\** ***@FeignClient注解表示和一个provider对应，value属性指定要调用的Provider的微服务名称;****\* fallbackFactory属性指定 consumer 调用 provider 时如果失败所采取的备用方案  
 \*/*@FeignClient(value = "redis-provider")  
public interface RedisRemoteService {  
  
 */\*\*  
 \* redis操作，不带超时的set方法  
 \*  
 \** ***@param*** *key  
 \** ***@param*** *value  
 \** ***@return*** *\*/* @RequestMapping("/set/redis/key/value/remote")  
 ResultEntity<String> setRedisKeyValueRemote(@RequestParam("key") String key, @RequestParam("value") String value);  
  
  
 */\*\*  
 \* redis操作，带超时的set方法  
 \** ***@param*** *key  
 \** ***@param*** *value  
 \** ***@param*** *time  
 \** ***@param*** *timeUnit  
 \** ***@return*** *\*/* @RequestMapping("/set/redis/key/value/remote/with/timeout")  
 ResultEntity<String> setRedisKeyValueRemoteWithTimeout(  
 @RequestParam("key") String key,  
 @RequestParam("value") String value,  
 @RequestParam("time") long time,  
 @RequestParam("timeUnit") TimeUnit timeUnit);  
  
 */\*\*  
 \* reddis操作，根据key获取value  
 \** ***@param*** *key  
 \** ***@return*** *\*/* @RequestMapping("/get/redis/value/by/key/remote")  
 ResultEntity<String> getRedisValueByKeyRemote(@RequestParam("key") String key);  
  
 */\*\*  
 \* redis操作，删除key  
 \** ***@param*** *key  
 \** ***@return*** *\*/* @RequestMapping("/remove/redis/key/remote")  
 ResultEntity<String> removeRedisKeyRemote(@RequestParam("key") String key);  
}

#### redis工程handler

@RestController  
public class RedisHandler {  
  
 @Autowired  
 private StringRedisTemplate redisTemplate;  
  
 */\*\*  
 \* redis操作，不带超时的set方法  
 \*  
 \** ***@param*** *key  
 \** ***@param*** *value  
 \** ***@return*** *\*/* @RequestMapping("/set/redis/key/value/remote")  
 ResultEntity<String> setRedisKeyValueRemote(@RequestParam("key") String key, @RequestParam("value") String value){  
 try{  
 ValueOperations<String, String> operations = redisTemplate.opsForValue();  
 operations.set(key,value);  
 }catch (Exception e){  
 return ResultEntity.*failed*(e.getMessage());  
 }  
 return ResultEntity.*successWithoutData*();  
 }  
  
  
 */\*\*  
 \* redis操作，带超时的set方法  
 \** ***@param*** *key  
 \** ***@param*** *value  
 \** ***@param*** *time  
 \** ***@param*** *timeUnit  
 \** ***@return*** *\*/* @RequestMapping("/set/redis/key/value/remote/with/timeout")  
 ResultEntity<String> setRedisKeyValueRemoteWithTimeout(  
 @RequestParam("key") String key,  
 @RequestParam("value") String value,  
 @RequestParam("time") long time,  
 @RequestParam("timeUnit") TimeUnit timeUnit){  
  
 try{  
 ValueOperations<String, String> operations = redisTemplate.opsForValue();  
 operations.set(key,value,time,timeUnit);  
 }catch (Exception e){  
 return ResultEntity.*failed*(e.getMessage());  
 }  
 return ResultEntity.*successWithoutData*();  
 }  
  
 */\*\*  
 \* reddis操作，根据key获取value  
 \** ***@param*** *key  
 \** ***@return*** *\*/* @RequestMapping("/get/redis/value/by/key/remote")  
 ResultEntity<String> getRedisValueByKeyRemote(@RequestParam("key") String key){  
 try{  
 ValueOperations<String, String> operations = redisTemplate.opsForValue();  
 String value = operations.get(key);  
 return ResultEntity.*successWithData*(value);  
 }catch (Exception e){  
 return ResultEntity.*failed*(e.getMessage());  
 }  
 }  
  
 */\*\*  
 \* redis操作，删除key  
 \** ***@param*** *key  
 \** ***@return*** *\*/* @RequestMapping("/remove/redis/key/remote")  
 ResultEntity<String> removeRedisKeyRemote(@RequestParam("key") String key){  
 try{  
 redisTemplate.delete(key);  
 return ResultEntity.*successWithoutData*();  
 }catch (Exception e){  
 return ResultEntity.*failed*(e.getMessage());  
 }  
 }  
}

#### 测试

http://localhost:3000/set/redis/key/value/remote?key=name&value=shuyun



## 认证工程

### 依赖

<dependencies>  
 <!--对外服务-->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-web</artifactId>  
 </dependency>  
 <!--thymeleaf模板-->  
 <dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-thymeleaf</artifactId>  
 </dependency>  
 <!--eureka客户端-->  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
 </dependency>  
 <!--api工程-->  
 <dependency>  
 <groupId>com.atguigu.crowd</groupId>  
 <artifactId>crowd-17-member-api</artifactId>  
 <version>1.0-SNAPSHOT</version>  
 </dependency>  
</dependencies>

<!--热部署工具，开发使用，生产不用-->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-devtools</artifactId>  
 <version>2.1.6.RELEASE</version>  
</dependency>  
<!--热部署工具，开发使用，生产不用-->  
<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-loader</artifactId>  
 <version>2.1.6.RELEASE</version>  
</dependency>

### 主启动类

@EnableDiscoveryClient // 当前版本可以不写  
@SpringBootApplication  
public class AuthenticationMainClass {  
 public static void main(String[] args) {  
 SpringApplication.*run*(AuthenticationMainClass.class);  
 System.*out*.println("Hello world!");  
 }  
}

### application.yml

server:  
 port: 4000  
  
spring:  
 application:  
 *#注册到eureka的微服务名称* name: authentication-consumer  
  
 thymeleaf:  
 *#视图前缀* prefix: classpath:/templates/  
 *#视图后缀* suffix: .html  
  
eureka:  
 client:  
 service-url:  
 *#eureka服务器地址，客户端需要往这个地址去注册* defaultZone: http://localhost:1000/eureka

### 首页handler

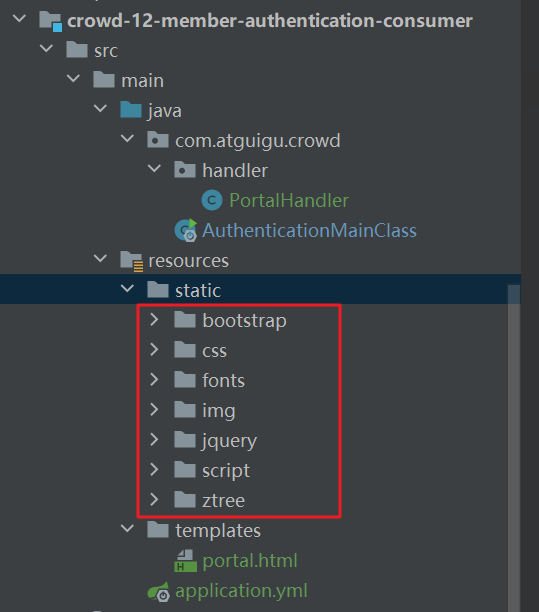
注意：这里要返回视图，所以不要用@RestController注解

@Controller  
public class PortalHandler {  
  
 */\*\*  
 \* portal首页  
 \** ***@return*** *\*/* @RequestMapping("/")  
 public String showPortal(){  
 return "portal";  
 }  
}

### 首页portal.html 页面

#### 加入静态资源

SpringBoot 要求在 static 目录下存放静态资源。



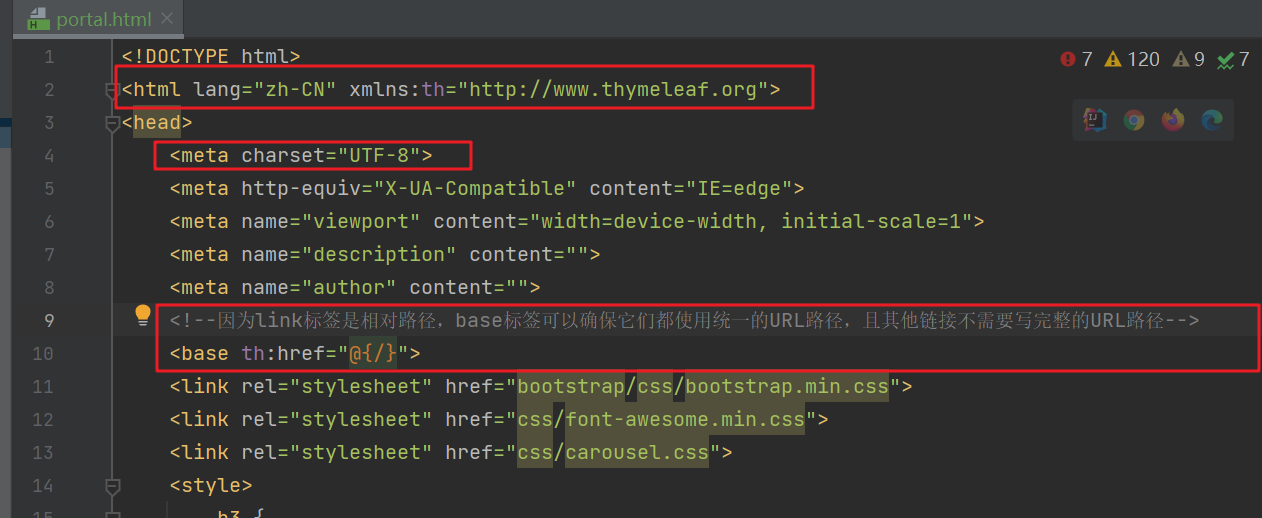
#### portal页面修改

从index.html拿源码过来修改

1.加入thymeleaf模板

2.修改字符集为UTF-8

3.引入base标签，简化链接路径，确保所有资源使用同一个URL进行解析



## zuul网关

### 依赖

<dependencies>  
 <!--eureka客户端-->  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>  
 </dependency>  
 <!--zuul网关-->  
 <dependency>  
 <groupId>org.springframework.cloud</groupId>  
 <artifactId>spring-cloud-starter-netflix-zuul</artifactId>  
 </dependency>  
</dependencies>

### 主启动类

主启动类，加上@EnableZuulProxy注解，启用zuul代理功能

//开启zuul代理功能  
@EnableZuulProxy  
@SpringBootApplication  
public class ZuulMainClass {  
 public static void main(String[] args) {  
 SpringApplication.*run*(ZuulMainClass.class);  
 System.*out*.println("Hello world!");  
 }  
}

### application.yml

1.禁用通过微服务名访问，此时需通过 zuul网关地址+路由规则指定的别名+具体url地址来访问

2.指定路由规则，/\*\*表示多层匹配

server:  
 port: 80  
  
spring:  
 application:  
 *#注册到eureka的微服务名称* name: zuul  
  
eureka:  
 client:  
 service-url:  
 *#eureka服务器地址，客户端需要往这个地址去注册* defaultZone: http://localhost:1000/eureka  
zuul:  
 *#不允许通过微服务名进行访问，\*表示所有微服务* ignored-services: "\*"  
 *#在 Zuul 向其他微服务重定向时保持原本头信息（请求头、响应头)* sensitive-headers: "\*"  
 routes:  
 *#自定义路由规则* crowd-portal:  
 *#将该微服务路径替换成path路径* service-id: authentication-consumer  
 *#这里一定要使用两个“\*”号，不然“/”路径后面的多层路径将无法访问* path: /\*\*

### 测试

通过zuul访问首页

1.启动eureka、zuul、authentication工程

2.通过zuul网关访问首页 http://localhost:80/

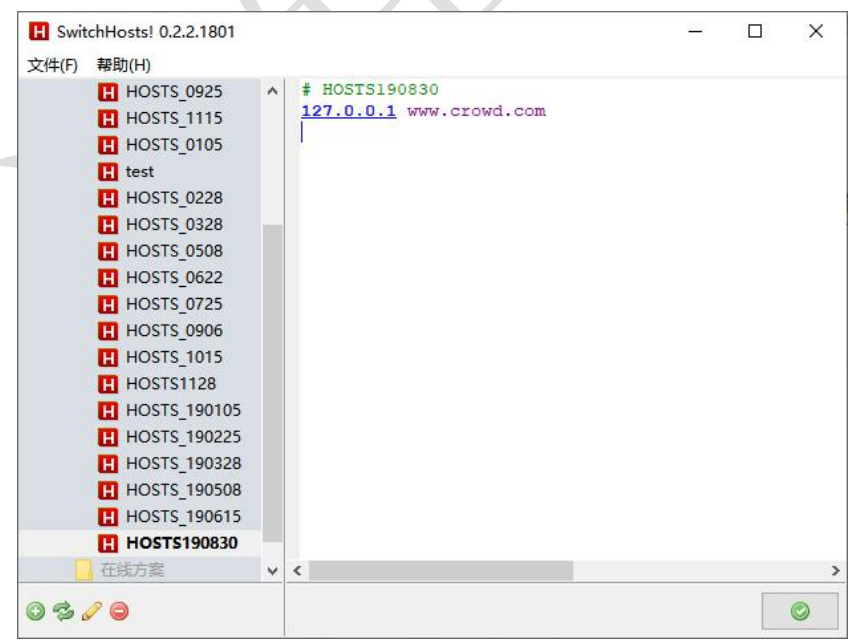
效果：



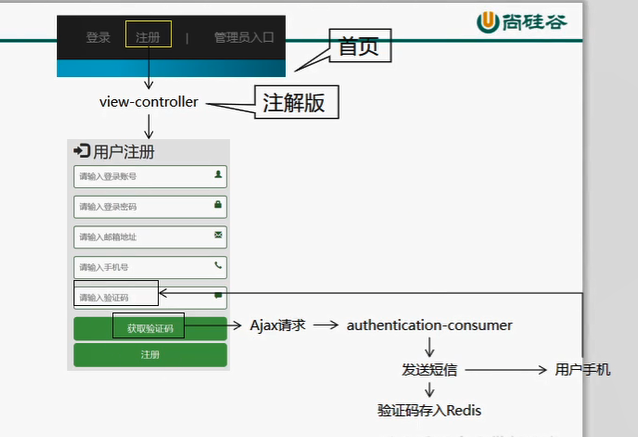
### 域名配置（可选）

暂时不做

就是将本地ip替换成域名，可以通过域名直接访问



# 会员登录



## 短信验证码

### 第三方接口

需要从阿里云网站引用第三方接口

https://www.aliyun.com/

登录阿里云网站---云市场---数据与API---短信与运营商

要先登录= =

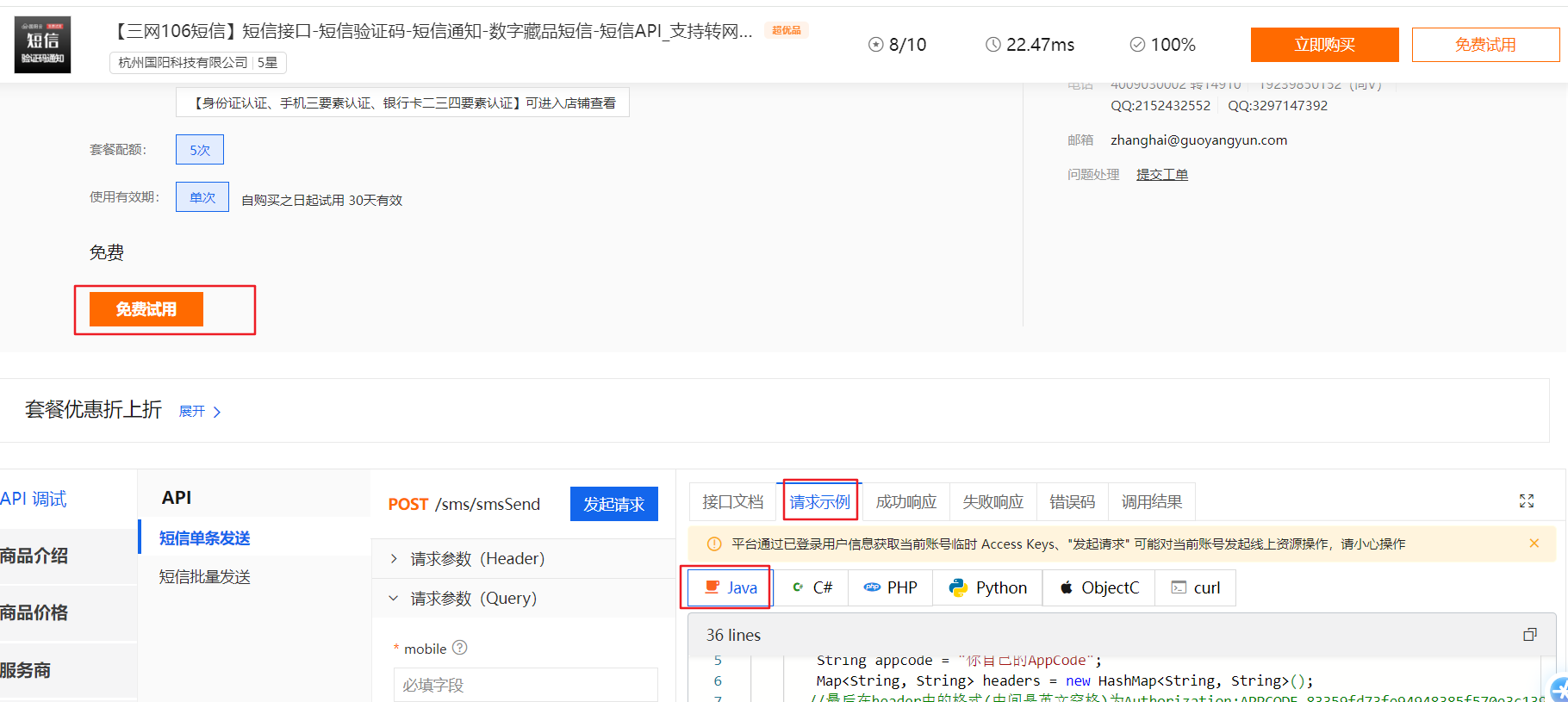




搜索“短信”，然后随便找一个点进去，免费使用，在“买家控制台”可以看使用列表，里面就有APPCode



点击“请求示例”，把里面的代码拷贝到项目中，再进行调整



### 引入短信验证到项目

将短信验证接口引入到util工程，然后再由authentication工程引用util工程

将 host path method appcode phone作为配置参数，调用接口的时候再读取传入

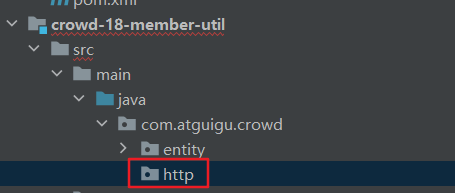
手机号的参数名也可以作为配置参数，调用时再传进去

#### HttpUtil

打开提供的地址



在util工程新建一个包存放



新建HttpUtil类，然后把代码拷贝过来

再引入httputil的依赖，打开这个网址，然后把依赖拷贝过来，放到pom.xml中

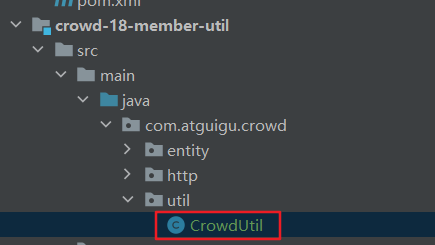


APPCode在买家控制台可以查看



#### 封装短信发送类

新建工具类，然后将请求示例的代码拷贝到里面，做成一个静态方法



将可变参数抽取出来，作为配置项，放到配置文件里面，方便修改

public class CrowdUtil {  
 */\*\*  
 \* 调用第三方接口，给用户手机发送短信验证码，发送成功后将验证码返回  
 \*  
 \** ***@param*** *host 接口地址  
 \** ***@param*** *path 路径  
 \** ***@param*** *method http请求方式 GET POST  
 \** ***@param*** *appcode APPCode  
 \** ***@param*** *mobile 接收短信的手机号  
 \** ***@param*** *smsSignId 短信前缀，即短信签名id  
 \** ***@param*** *templateId 短信模板  
 \** ***@return*** *\*/* public static ResultEntity<String> sendShortMessage(  
 String host,  
 String path,  
 String method,  
 String appcode,  
 String mobile,  
 String smsSignId,  
 String templateId  
 ) {  
 Map<String, String> headers = new HashMap<String, String>();  
 //最后在header中的格式(中间是英文空格)为Authorization:APPCODE 83359fd73fe94948385f570e3c139105  
 headers.put("Authorization", "APPCODE " + appcode);  
 Map<String, String> querys = new HashMap<String, String>();  
 querys.put("mobile", mobile);  
 //获取6位验证码  
 String messageCode = *getMessageAhtenticationCode*(6);  
  
 querys.put("param", "\*\*code\*\*:"+messageCode+",\*\*minute\*\*:5");  
  
 //smsSignId（短信前缀）和templateId（短信模板），可登录国阳云控制台自助申请。参考文档：http://help.guoyangyun.com/Problem/Qm.html  
  
 querys.put("smsSignId", smsSignId);  
 querys.put("templateId", templateId);  
 Map<String, String> bodys = new HashMap<String, String>();  
  
 try {  
 */\*\*  
 \* 重要提示如下:  
 \* HttpUtils请从\r\n\t \t\* https://github.com/aliyun/api-gateway-demo-sign-java/blob/master/src/main/java/com/aliyun/api/gateway/demo/util/HttpUtils.java\r\n\t \t\* 下载  
 \*  
 \* 相应的依赖请参照  
 \* https://github.com/aliyun/api-gateway-demo-sign-java/blob/master/pom.xml  
 \*/* HttpResponse response = HttpUtils.*doPost*(host, path, method, headers, querys, bodys);  
 StatusLine statusLine = response.getStatusLine();  
 //调用状态码  
 int statusCode = statusLine.getStatusCode();  
 String reasonPhrase = statusLine.getReasonPhrase();  
  
 if (200 == statusCode) {  
 //发送成功后，返回验证码，存到redis中  
 return ResultEntity.*successWithData*(messageCode);  
 } else {  
 return ResultEntity.*failed*(reasonPhrase);  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 return ResultEntity.*failed*(e.getMessage());  
 }  
  
 }  
  
 */\*\*  
 \* 生成指定位数的随机验证码，只包含数字  
 \** ***@param*** *length 验证码长度  
 \** ***@return*** *\*/* public static String getMessageAhtenticationCode(int length) {  
 StringBuilder code = new StringBuilder();  
  
 for (int i = 0; i < length; i++) {  
 int num = (int) (Math.*random*() \* 10);  
 code.append(num);  
 }  
 return code.toString();  
 }  
}

#### 依赖分析

util工程中引入了第三方接口的依赖，且项目中多个工程引用了util工程，为了避免依赖冲突，需要把已经存在的依赖排除掉。

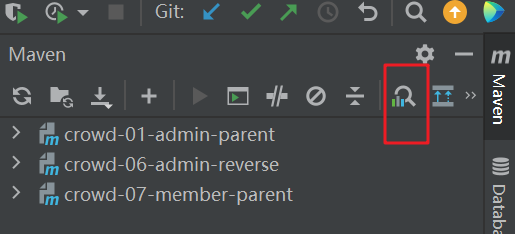
因为 发送短信验证码的功能是authentication认证工程在使用，所以主要和该工程做比较

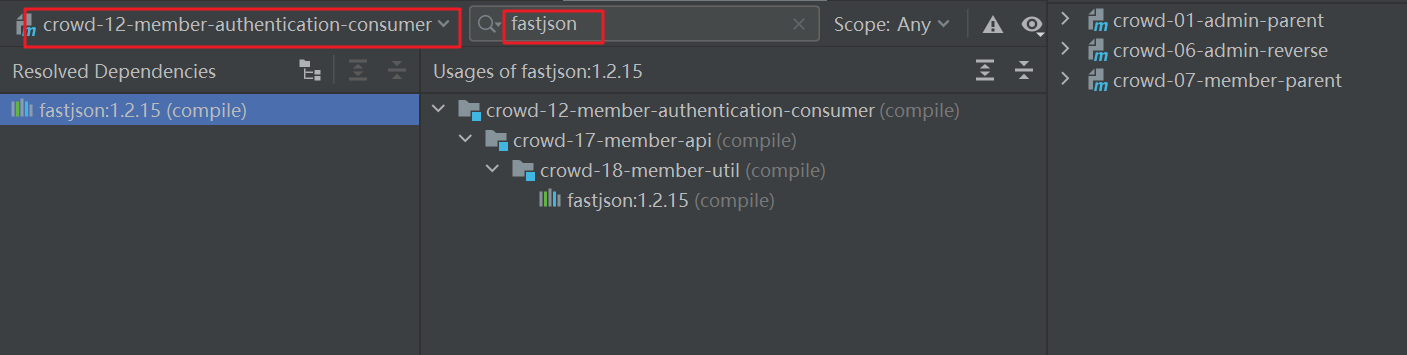
引入的依赖如下：

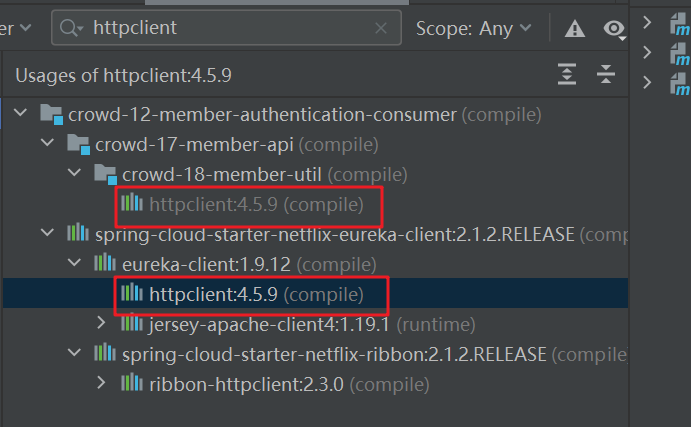
fastjson、httpclient、httpcore、commons-lang、jetty-util、junit

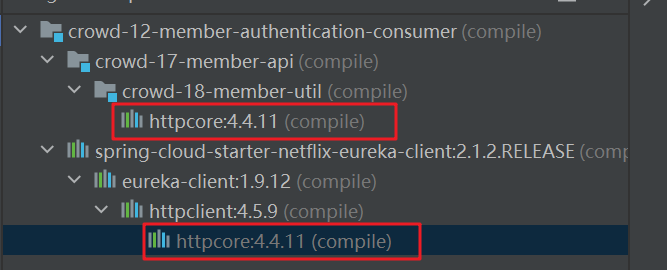
要确保util工程引入的依赖版本和已有的一致，否则会冲突

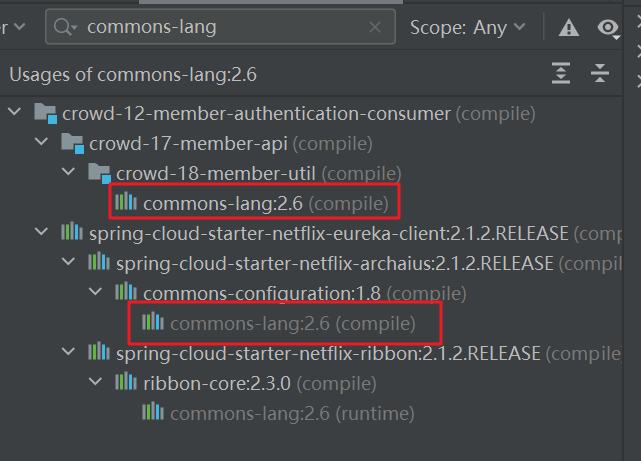
如果两个都存在时，优先用版本高的

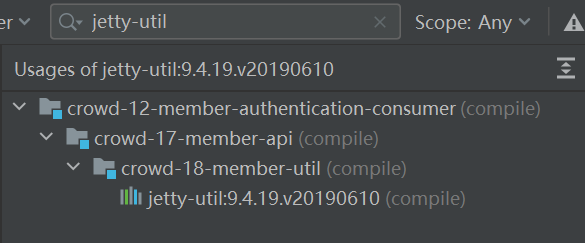












## 会员注册



### 前往注册界面

点击首页的 "注册" 按钮，弹出注册的界面

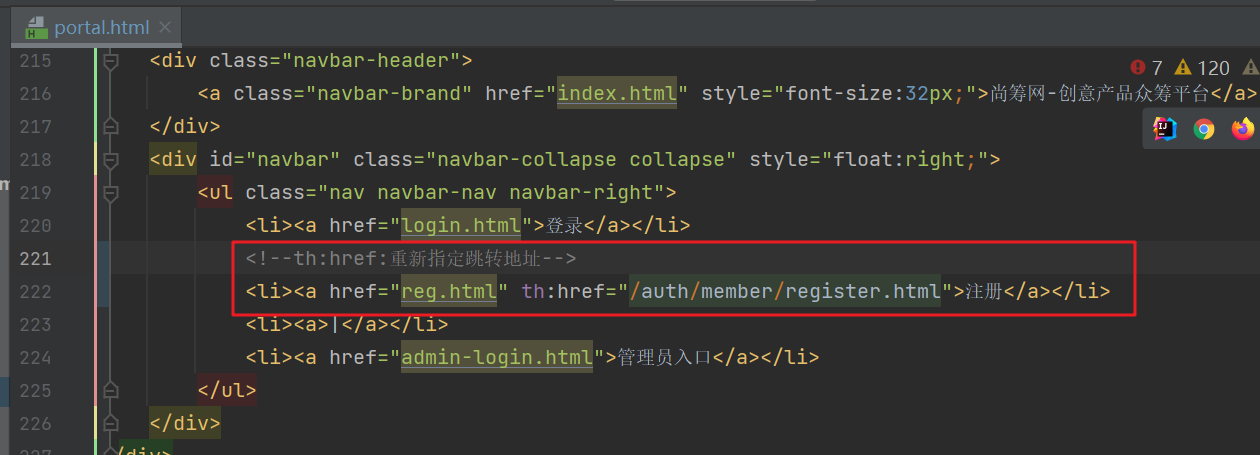
1.给首页的 "注册" 按钮指定跳转地址

2.控制器根据"注册"按钮的地址，跳转到"注册"页面

3.准备注册html页面

#### 首页"注册"按钮

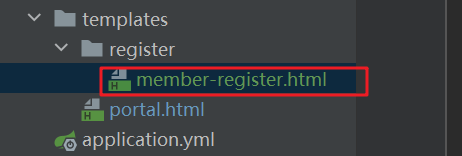
portal.html页面中，修改"注册"按钮的地址为：



#### 注册界面

页面注意分类存放，且在application.yml中配置了视图的前后缀，要放在templates目录下

注册界面放在：/templates/register目录下，视图名称：member-register.html

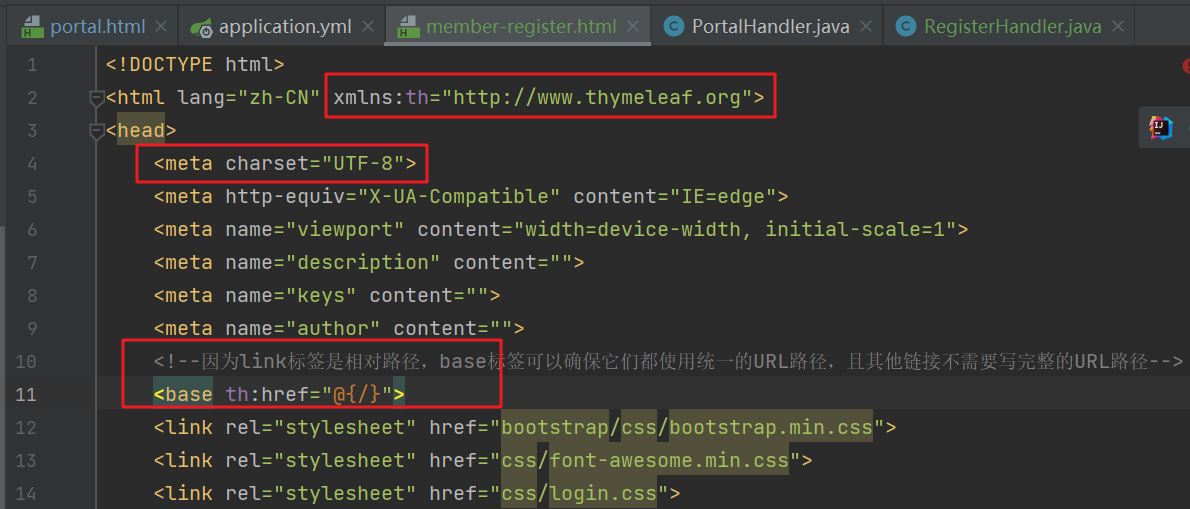


然后从原型中把源码拿过来修改即可

1.增加thymeleaf模板

2.修改字符集为 UTF-8

3.增加base标签，统一URL前缀地址



#### 跳转

因为跳转到注册界面不需要业务逻辑处理，所以可以使用配置或者使用控制器跳转。配置可以使用xml配置，也可以使用注解来配置。这三种方法下面都介绍下，本次使用注解配置。

方法一：控制器方法跳转

新建一个RegisterHandler类，专门处理会员注册的请求

@Controller  
public class RegisterHandler {  
  
 @RequestMapping("/auth/member/register.html")  
 public String toMemberRegister(){  
 return "register/member-register";  
 }  
}

方法二：xml配置

这个是在未使用springboot前的xml配置

在spring-web-mvc.xml中配置地址和跳转的视图，方法三就是使用注解方式实现



方法三：注解配置

新增一个配置类，直接实现WebMvcConfigurer，重写addViewControllers()方法，该方法中有path和viewname两个属性，分别表示请求路径和目标视图名称

@Configuration  
public class MemberRegisterConfig implements WebMvcConfigurer {  
 @Override  
 public void addViewControllers(ViewControllerRegistry registry) {  
 String path = "/auth/member/register.html";  
 String viewName = "register/member-register";  
 registry.addViewController(path).setViewName(viewName);  
 }  
}

### 发送短信验证码



给"获取短信验证码"绑定单击事件，发送ajax请求后端，调用第三方接口，发送短信验证码。

1.获取输入的手机号

2.发送ajax请求

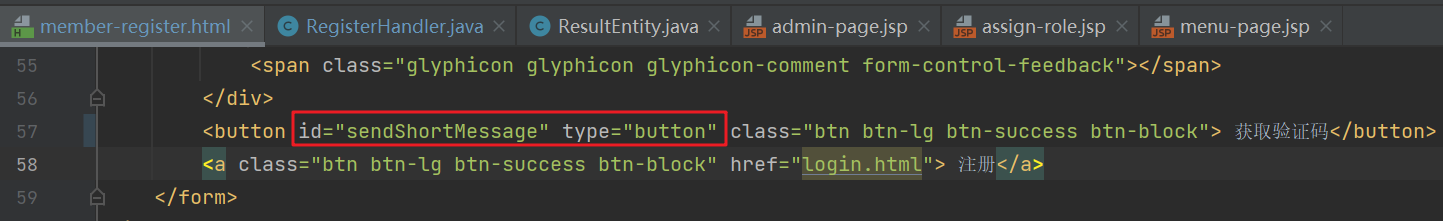
#### 页面

1.给"获取验证码"按钮增加 id属性和type属性

2.绑定单击事件

注意：如果"获取验证码"按钮没有type="button"这个属性，点击该按钮的时候会提交表单，而当前form表单没有指定action属性，就会在提交表单后返回本页面，导致页面刷新。

按钮增加两个属性



JS代码，绑定单击事件，发送ajax请求

<script type="text/javascript" src="jquery/jquery-2.1.1.min.js"></script>  
<script type="text/javascript" src="bootstrap/js/bootstrap.min.js"></script>  
<script type="text/javascript" src="layer/layer.js"></script>  
<script type="text/javascript">  
  
 $(function () {  
 $("#sendShortMessage").click(function () {  
 //1.获取用户输入的手机号  
 var phoneNum = $.trim($("[name=phoneNum]").val());  
 //2.发送ajax请求后端，发送验证码到该手机上  
 $.ajax({  
 "url": "/auth/member/sendShortMessage",  
 "type": "post",  
 "data": {  
 "phoneNum": phoneNum  
 },  
 "dataType": "json",  
 "success": function (response) {  
 let result = response.result;  
 if ("SUCCESS" == result){  
 layer.msg("验证码发送成功");  
 }else {  
 layer.msg("验证码发送失败，请稍后再试"+response.message);  
 return ;  
 }  
 },  
 "error": function (response) {  
 layer.msg("验证码发送失败，请稍后再试")  
 }  
 });  
 });  
 });  
</script>

#### 后端

application.yml配置发送短信接口的参数

增加以下参数配置：

message:  
 host: https://gyytz.market.alicloudapi.com  
 path: /sms/smsSend  
 method: POST  
 appcode: 2fc1a2c00e5f4b22adb8571bd2b6b6ea  
 smsSignId: 2e65b1bb3d054466b82f0c9d125465e2  
 templateId: 908e94ccf08b4476ba6c876d13f084ad

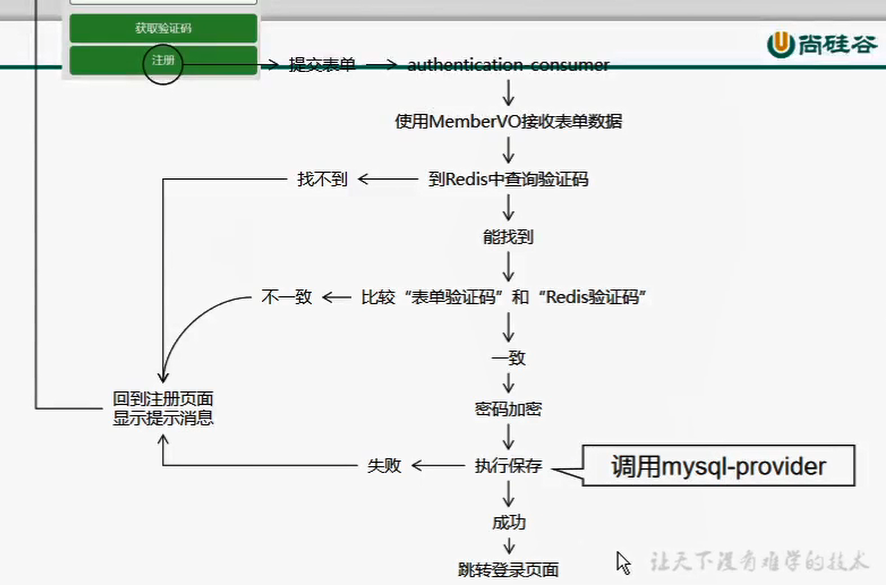
控制器方法：

@ResponseBody  
@RequestMapping("/auth/member/sendShortMessage")  
public ResultEntity<String> sendMessage(@RequestParam("phoneNum") String phoneNum) {  
 //调用第三方接口，发送短信验证码到用户手机上  
 ResultEntity<String> sendShortMessageEntity = CrowdUtil.*sendShortMessage*(host, path, method, appcode, phoneNum, smsSignId, templateId);  
  
 if (ResultEntity.*SUCCESS*.equals(sendShortMessageEntity.getResult())) {  
 logger.info("短信发送成功");  
 //短信发送成功  
 //验证码存入redis中的key,确保每个号码唯一  
 String key = CrowdConstant.*REDIS\_MESSAGE\_CODE\_PREFIX* + phoneNum;  
 //验证码  
 String value = sendShortMessageEntity.getData();  
  
 //短信发送成功，将验证码存入到redis中  
 ResultEntity<String> redisstringResultEntity = redisRemoteService.setRedisKeyValueRemoteWithTimeout(key, value, 5, TimeUnit.*MINUTES*);  
  
 logger.info("redis操作："+redisstringResultEntity.getResult());  
 //不管redis成功失败，直接返回即可  
 return redisstringResultEntity;  
 } else {  
 logger.info("短信发送失败");  
 //短信发送失败，直接将结果返回前端  
 return sendShortMessageEntity;  
 }  
}

注意，主启动类要加上 @EnableFeignClients注解，否则远程调用接口时会报错

### 执行注册

点击"注册"按钮，将整个表单发送给后端，执行数据校验，校验通过数据入库，注册成功。



1.给"注册"按钮绑定单击事件

2.获取表单数据，发送ajax请求后端

3.后端接收"注册"表单的数据，转换成VO

4.检查"注册"表单的数据有效性

5.到redis查询验证码，和表单验证码进行比较

6.验证码一致（需要删除redis中存的验证码），则检查该账号是否已存在，不存在继续下一步

7.账号未存在，则对密码进行加密，保存到数据库

8.保存成功，重定向到“登录”页面

#### mysql工程

提供查询和保存操作

1.sql

<select id="selectByLoginAcct" parameterType="java.lang.String" resultMap="BaseResultMap">  
 select  
 <include refid="Base\_Column\_List" />  
 from t\_member  
 where loginacct = #{loginAcct}  
</select>  
  
<insert id="saveMemberPo" parameterType="com.atguigu.crowd.entity.po.MemberPo">  
 insert into t\_member  
 <trim prefix="(" suffix=")" suffixOverrides=",">  
 <if test="id != null and id !=''">id,</if>  
 <if test="loginacct != null and loginacct !=''">loginacct,</if>  
 <if test="userpswd != null and userpswd !=''">userpswd,</if>  
 <if test="username != null and username !=''">username,</if>  
 <if test="email != null and email !=''">email,</if>  
 <if test="authstatus != null and authstatus !=''">authstatus,</if>  
 <if test="usertype != null and usertype !=''">usertype,</if>  
 <if test="realname != null and realname !=''">realname,</if>  
 <if test="cardnum != null and cardnum !=''">cardnum,</if>  
 <if test="accttype != null and accttype !=''">accttype,</if>  
 </trim>  
 <trim prefix="values(" suffix=")" suffixOverrides=",">  
 <if test="id != null and id !=''">#{id},</if>  
 <if test="loginacct != null and loginacct !=''">#{loginacct},</if>  
 <if test="userpswd != null and userpswd !=''">#{userpswd},</if>  
 <if test="username != null and username !=''">#{username},</if>  
 <if test="email != null and email !=''">#{email},</if>  
 <if test="authstatus != null and authstatus !=''">#{authstatus},</if>  
 <if test="usertype != null and usertype !=''">#{usertype},</if>  
 <if test="realname != null and realname !=''">#{realname},</if>  
 <if test="cardnum != null and cardnum !=''">#{cardnum},</if>  
 <if test="accttype != null and accttype !=''">#{accttype},</if>  
 </trim>  
</insert>

2.mapper接口

*/\*\*  
\* 根据登录账号查询成员信息  
\** ***@param*** *loginAcct  
\** ***@return****\*/*MemberPo selectByLoginAcct(@Param("loginAcct") String loginAcct);  
  
*/\*\*  
\* 新增会员  
\** ***@param*** *po  
\*/*void saveMemberPo(MemberPo po);

3.service实现类

注意事务的配置： @Transactional注解的三个属性：

propagation = Propagation.REQUIRES\_NEW：表示不管当前线程上有没有事务，都要自己开事务，在自己的事务中运行。

rollbackFor = Exception.class：编译时异常和运行时异常都回滚

readOnly = false：是否只读，false表示关闭，即可以读写

// 在类上使用@Transactional(readOnly = true)针对查询操作设置事务属性  
@Transactional(readOnly = true)  
@Service  
public class MemberServiceImpl implements MemberService {  
 @Autowired  
 private MemberPoMapper mapper;  
  
 @Override  
 public MemberPo getMemberByLoginacct(String loginacct) {  
 return mapper.selectByLoginAcct(loginacct);  
 }  
  
  
 */\*\*  
 \* REQUIRES\_NEW 表示不管当前线程上有没有事务，都要自己开事务，在自己的事务中运行。  
 \* rollbackFor 编译时异常和运行时异常都回滚  
 \* readOnly 是否只读  
 \** ***@param*** *po  
 \*/* @Transactional(propagation = Propagation.*REQUIRES\_NEW*,rollbackFor = Exception.class,readOnly = false)  
 @Override  
 public void saveMemberPo(MemberPo po) {  
 mapper.saveMemberPo(po);  
 }  
}

4.控制器

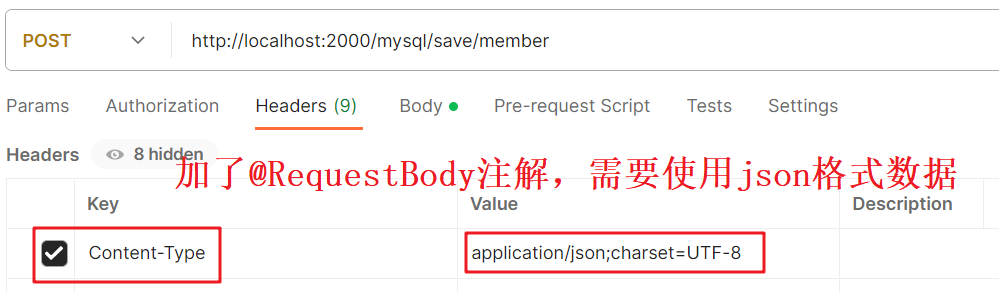
@RestController  
public class MemberProviderHandler {  
 @Autowired  
 private MemberService memberService;  
  
  
 */\*\*  
 \* provider工程的远程方法，要和api工程有 @FeignClient 注解标记的方法一致  
 \* 根据账号查询成员信息  
 \** ***@param*** *loginacct  
 \** ***@return*** *\*/* @RequestMapping("/get/member/by/loginacct/remote")  
 public ResultEntity<MemberPo> getMemberByLoginacctRemote(@RequestParam("loginacct") String loginacct){  
 //*todo 异常处理* MemberPo member = memberService.getMemberByLoginacct(loginacct);  
 return ResultEntity.*successWithData*(member);  
 }  
  
 */\*\*  
 \* 新增会员  
 \** ***@param*** *po  
 \** ***@return*** *\*/* @ResponseBody  
 @RequestMapping("/mysql/save/member/remote")  
 public ResultEntity<String> saveMemberPoRemote(@RequestBody MemberPo po){  
 try {  
 memberService.saveMemberPo(po);  
 }catch (Exception e){  
 if (e instanceof DuplicateKeyException){  
 return ResultEntity.*failed*("该账号已存在");  
 }  
 return ResultEntity.*failed*(e.getMessage());  
 }  
 return ResultEntity.*successWithoutData*();  
 }  
}

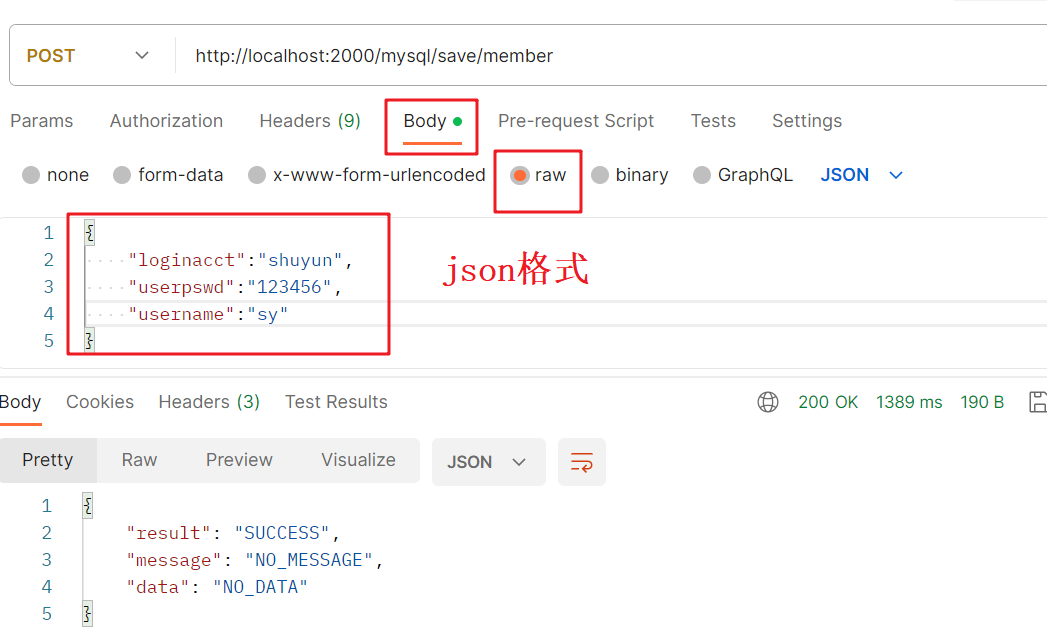
5.测试

因为是要MemberPo接收数据，所以不能直接用浏览器发请求，因为浏览器发起的是get请求，没有请求体。所以需要使用postman发请求

加了@RequestBody注解，请求头要使用application/json;charset=UTF-8 格式

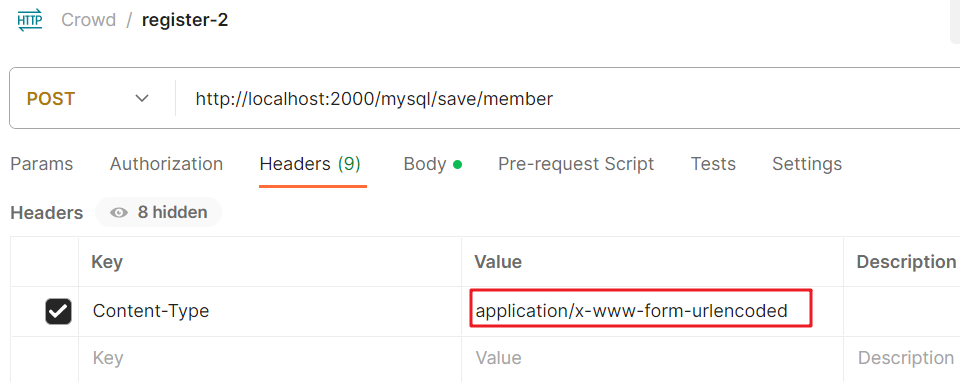
请求体使用json格式

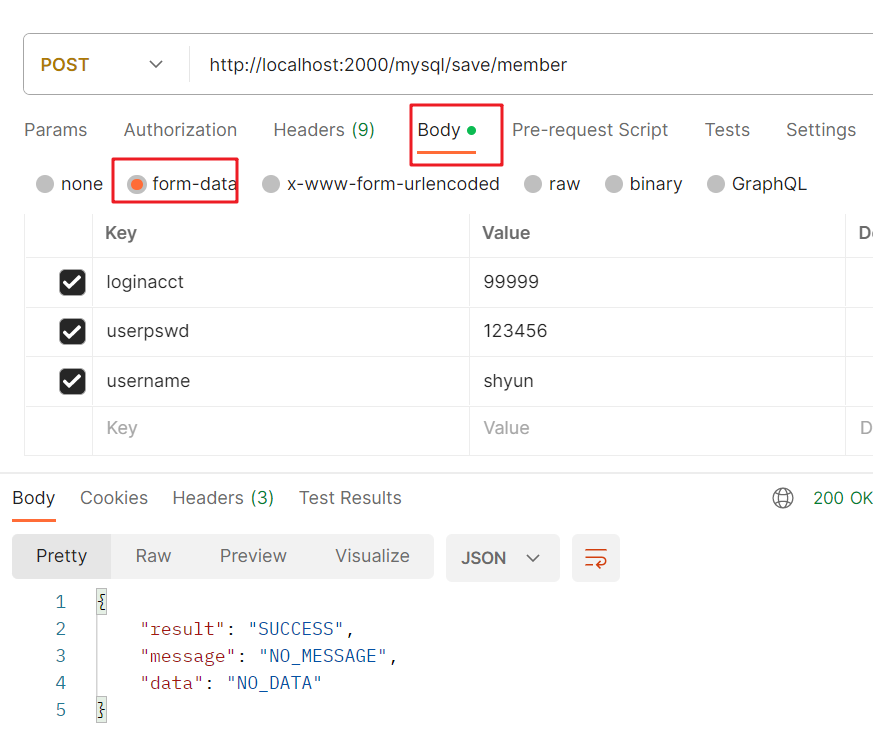




如果不加 @RequestBody注解，请求头要使用application/x-www-form-urlencoded

请求体要使用表单形式





#### entity工程

提供vo类，即存放和浏览器交互数据的实体类

引入lombok依赖

<dependency>  
 <groupId>org.projectlombok</groupId>  
 <artifactId>lombok</artifactId>  
 <version>1.18.22</version>  
</dependency>

VO实体类

@Data  
@NoArgsConstructor  
@AllArgsConstructor  
public class MemberVo {  
  
 private Integer id;  
  
 private String loginacct;  
  
 private String userpswd;  
  
 private String username;  
  
 private String email;  
  
 private String phoneNum;  
  
 private String messageCode;  
}

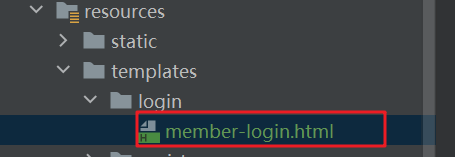
#### api工程

api工程的方法要和远程Provider工程方法保持一致

//@FeignClient注解表示和一个provider对应，value属性指定要调用的Provider的微服务名称;  
// fallbackFactory属性指定 consumer 调用 provider 时如果失败所采取的备用方案  
@FeignClient(value = "mysql-provider")  
public interface MySQLRemoteService {  
  
 */\*\*  
 \* 根据账号查询会员信息  
 \* 会调用远程mysql-Provider工程的同名方法  
 \* 注意：这里的方法要和 Provider中的具体实现的方法的声明要一致  
 \** ***@RequestMapping注解的url地址、@RequestParam、@RequestBody、@PathVariable两边一致*** *\** ***@param*** *loginacct  
 \** ***@return*** *\*/* @RequestMapping("/get/member/by/loginacct/remote")  
 ResultEntity<MemberPo> getMemberByLoginacctRemote(@RequestParam("loginacct") String loginacct);  
  
 */\*\*  
 \* 新增会员  
 \** ***@param*** *po  
 \** ***@return*** *\*/* @ResponseBody  
 @RequestMapping("/mysql/save/member/remote")  
 ResultEntity<String> saveMemberPoRemote(@RequestBody MemberPo po);  
}

#### auth工程

1.准备临时的登录界面



2.控制器方法，进行注册验证

主要操作：

①接收表单数据，转换成Vo，注意@RequestBody注解的使用。如果是ajax请求，则需要加上@RequestBody注解。如果是直接提交form表单，不加@RequestBody注解

②检查Vo数据的有效性，主要是检查必传参数是否为空，如果为空，返回注册页面进行消息提示

③调用redis远程接口，查询该手机的验证码，检查校验码是否一致

④检查注册的账号是否已存在

⑤对密码加密，然后调用mysql工程的远程接口，进行保存操作

⑥保存成功后，调用redis远程接口，删除该用户手机的验证码

⑦重定向到登录页面

@Controller  
public class RegisterHandler {  
 private Logger logger = LoggerFactory.*getLogger*(RegisterHandler.class);  
  
 */\*\*  
 \* 从首页的"注册"按钮，跳转到注册页面，不使用，使用配置类的注解方式跳转  
 \*  
 \** ***@return*** *\*/*// @RequestMapping("/auth/member/register.html")  
// public String toMemberRegister(){  
// return "register/member-register";  
//  
// }  
 @Autowired  
 private RedisRemoteService redisRemoteService;  
  
 @Autowired  
 private MySQLRemoteService mySQLRemoteService;  
  
 @Value("${message.host}")  
 private String host;  
  
 @Value("${message.path}")  
 private String path;  
  
 @Value("${message.method}")  
 private String method;  
  
 @Value("${message.appcode}")  
 private String appcode;  
  
 @Value("${message.smsSignId}")  
 private String smsSignId;  
  
 @Value("${message.templateId}")  
 private String templateId;  
  
 */\*\*  
 \* 发送短信验证码到用户手机  
 \** ***@param*** *phoneNum  
 \** ***@return*** *\*/* @ResponseBody  
 @RequestMapping("/auth/member/sendShortMessage")  
 public ResultEntity<String> sendMessage(@RequestParam("phoneNum") String phoneNum) {  
 //调用第三方接口，发送短信验证码到用户手机上  
 ResultEntity<String> sendShortMessageEntity = CrowdUtil.*sendShortMessage*(host, path, method, appcode, phoneNum, smsSignId, templateId);  
  
 if (ResultEntity.*SUCCESS*.equals(sendShortMessageEntity.getResult())) {  
 logger.info("短信发送成功");  
 //短信发送成功  
 //验证码存入redis中的key,确保每个号码唯一  
 String key = CrowdConstant.*REDIS\_MESSAGE\_CODE\_PREFIX* + phoneNum;  
 //验证码  
 String value = sendShortMessageEntity.getData();  
  
 //短信发送成功，将验证码存入到redis中  
 ResultEntity<String> redisstringResultEntity = redisRemoteService.setRedisKeyValueRemoteWithTimeout(key, value, 5, TimeUnit.*MINUTES*);  
  
 logger.info("redis操作："+redisstringResultEntity.getResult());  
 //不管redis成功失败，直接返回即可  
 return redisstringResultEntity;  
 } else {  
 logger.info("短信发送失败");  
 //短信发送失败，直接将结果返回前端  
 return sendShortMessageEntity;  
 }  
 }  
  
 */\*\*  
 \* 执行会员注册  
 \** ***@param*** *memberVo  
 \** ***@param*** *modelMap  
 \** ***@return*** *\*/* @RequestMapping("/auth/member/do/register")  
 public String memberRegister(MemberVo memberVo, ModelMap modelMap){  
 String registerView = "/register/member-register";  
 //1.VO参数检查，必要参数是否为空  
 String tips = checkReq(memberVo,modelMap);  
 if (!StringUtils.*isEmpty*(tips)){  
 modelMap.addAttribute(CrowdConstant.*REGISTER\_CHECK*,tips);  
 return registerView;  
 }  
  
 //2.从redis中拿出验证码  
 String messageCodeKey = CrowdConstant.*REDIS\_MESSAGE\_CODE\_PREFIX*+memberVo.getPhoneNum();  
 ResultEntity<String> messageCodeEntity = redisRemoteService.getRedisValueByKeyRemote(messageCodeKey);  
 //redis请求失败，返回注册页面，提示错误消息  
 if (ResultEntity.*FAILED*.equals(messageCodeEntity.getResult())){  
 modelMap.addAttribute(CrowdConstant.*REGISTER\_CHECK*,CrowdConstant.*REGISTER\_FAIL*);  
 return registerView;  
 }  
  
 //3.比较表单的验证码和redis中拿出的验证码是否一致  
 String userCode = memberVo.getMessageCode();  
 String redisCode = messageCodeEntity.getData();  
 //验证码不一致  
 if (!Objects.*equals*(userCode, redisCode)){  
 modelMap.addAttribute(CrowdConstant.*REGISTER\_CHECK*,CrowdConstant.*MESSAGE\_CODE\_ERROR*);  
 return registerView;  
 }  
  
 //4.检查账号是否存在  
 ResultEntity<MemberPo> memberEntity = mySQLRemoteService.getMemberByLoginacctRemote(memberVo.getLoginacct());  
 if (!Objects.*isNull*(memberEntity.getData())){  
 modelMap.addAttribute(CrowdConstant.*REGISTER\_CHECK*,CrowdConstant.*LOGINACCT\_IS\_EXIST*);  
 return registerView;  
 }  
  
 //5.密码加密  
 BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();  
 String encodedPwd = encoder.encode(memberVo.getUserpswd());  
 memberVo.setUserpswd(encodedPwd);  
  
 //6.执行保存  
 MemberPo memberPo = new MemberPo();  
 BeanUtils.*copyProperties*(memberVo,memberPo);  
 ResultEntity<String> saveMemberPoEntity = mySQLRemoteService.saveMemberPoRemote(memberPo);  
 //mysql工程插入失败，返回注册页面，提示错误消息  
 if (ResultEntity.*FAILED*.equals(saveMemberPoEntity.getResult())){  
 modelMap.addAttribute(CrowdConstant.*REGISTER\_CHECK*,CrowdConstant.*REGISTER\_FAIL*);  
 return registerView;  
 }  
  
 //7.保存成功后，删除redis中的验证码  
 redisRemoteService.removeRedisKeyRemote(messageCodeKey);  
  
 //8.重定向到登录页面，重定向的地址在配置类里面转发  
 return "redirect:/auth/member/login";  
 }

*/\*\*  
 \* VO参数检查  
 \** ***@return*** *\*/* private String checkReq(MemberVo memberVo, ModelMap modelMap) {  
 //检查不通过，返回原来注册页面，提示错误消息  
 if (Objects.*isNull*(memberVo)){  
 return CrowdConstant.*REQUEST\_IS\_EMPTY*;  
 }  
 if (StringUtils.*isEmpty*(memberVo.getLoginacct())){  
 return CrowdConstant.*LOGINACCT\_IS\_EMPTY*;  
 }  
 if (StringUtils.*isEmpty*(memberVo.getUserpswd())){  
 return CrowdConstant.*PASD\_IS\_EMPTY*;  
 }  
 if (StringUtils.*isEmpty*(memberVo.getPhoneNum())){  
 return CrowdConstant.*PHONENUM\_IS\_EMPTY*;  
 }  
 if (StringUtils.*isEmpty*(memberVo.getMessageCode())){  
 return CrowdConstant.*MESSAGE\_CODE\_IS\_EMPTY*;  
 }  
 return null;  
 }  
}

重定向后，在配置类配置重定向地址

重定向是为了避免表单重复提交，多次注册

@Configuration  
public class MemberRegisterConfig implements WebMvcConfigurer {  
  
 @Override  
 public void addViewControllers(ViewControllerRegistry registry) {  
 String path = "/auth/member/register";  
 String viewName = "register/member-register";  
 //从首页跳转到注册页面  
 registry.addViewController(path).setViewName(viewName);  
 //从首页跳转到登录界面  
 registry.addViewController("/auth/member/login").setViewName("login/member-login");  
 }  
}

#### 页面表单和按钮修改

1.给表单的各个字段增加name属性

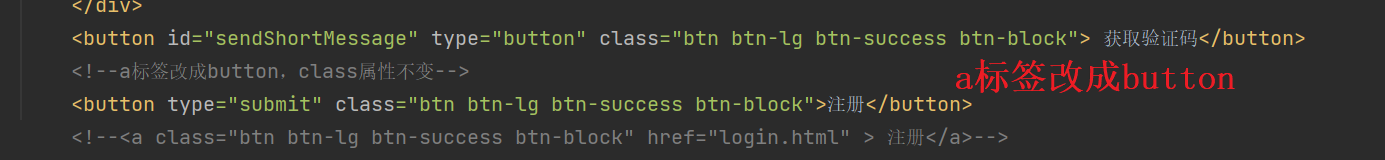
2.给form表单增加action属性，指定提交地址

3.<a>标签，改成button标签，点击后提交表单

4.增加一个字段，用户昵称

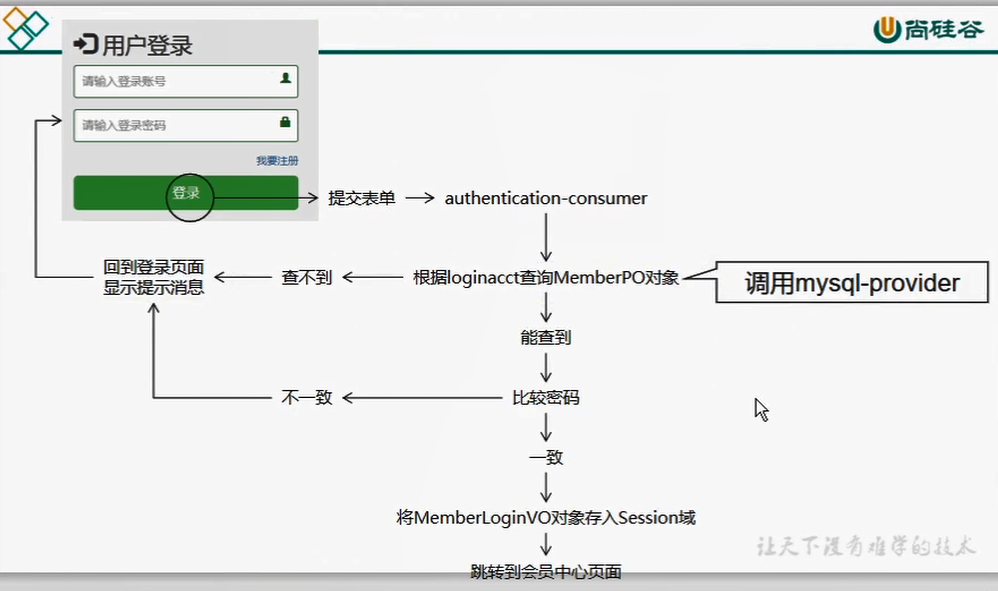
5.页面上获取错误消息提示





## 会员登录

检查账号密码正确后，将用户信息存入到Session域，表示用户登录成功



### 首页跳转到登录页面

修改a标签的跳转地址



### 登录页面

1.加上thymeleaf模板

2.修改字符集

3.加上base标签

4.表单增加name属性

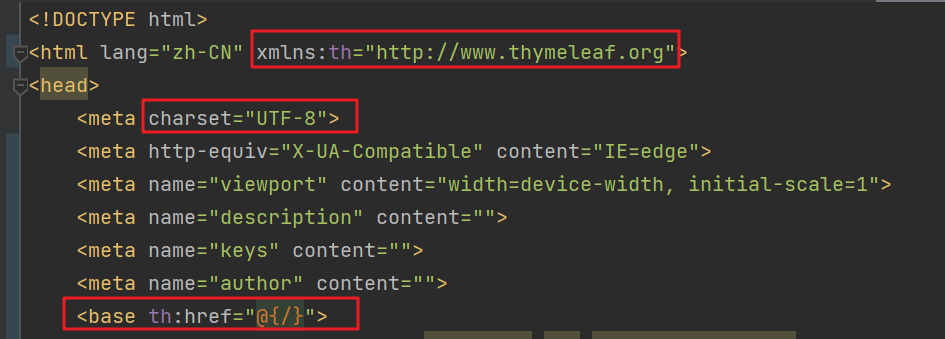
5.form表单增加action属性

6.注册标签的地址修改

7.登录的a标签改成button

8.页面错误消息提示

member-login.html页面修改：





### entity工程

将登录后的会员信息存入到Session域

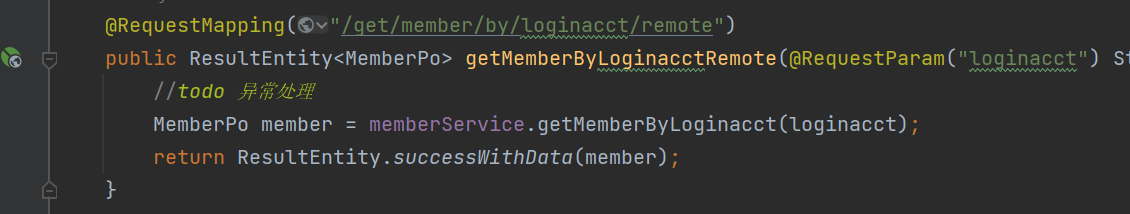
主要存：昵称、邮箱、id

后期在Session域中拿到数据，说明用户是已经登录了

@Data  
@AllArgsConstructor  
@NoArgsConstructor  
public class MemberLoginVo {  
 private Integer id;  
  
 private String username;  
  
 private String email;  
}

### mysql工程

已有相应接口，根据账号查询会员信息



### auth工程

RegisterHandler改名成MemberHandler

#### 控制器方法

*/\*\*  
\* 用户登录  
\** ***@param*** *memberVo  
\** ***@param*** *modelMap  
\** ***@param*** *session  
\** ***@return****\*/*@RequestMapping("/auth/member/do/login")  
public String login(MemberVo memberVo, ModelMap modelMap, HttpSession session){  
 String loginView = "login/member-login";  
  
 //1.根据账号查询用户信息  
 String loginacct = memberVo.getLoginacct();  
 ResultEntity<MemberPo> memberEntity= mySQLRemoteService.getMemberByLoginacctRemote(loginacct);  
 if (ResultEntity.*FAILED*.equalsIgnoreCase(memberEntity.getResult())){  
 //查询失败，返回登录页面，提示错误消息  
 modelMap.addAttribute(CrowdConstant.*LOGIN\_CHECK*,CrowdConstant.*LOGIN\_FAIL*);  
 return loginView;  
 }  
 MemberPo memberPo = memberEntity.getData();  
 if (Objects.*isNull*(memberPo)){  
 //查询不到用户，账号不存在  
 modelMap.addAttribute(CrowdConstant.*LOGIN\_CHECK*,CrowdConstant.*LOGINACCT\_IS\_NOT\_EXIST*);  
 return loginView;  
 }  
  
 //2.远程调用mysql工程接口，根据账号查询会员信息，进行密码校验  
 String userpswd = memberVo.getUserpswd();  
 String userpswdDataBase = memberPo.getUserpswd();  
 BCryptPasswordEncoder encoder = new BCryptPasswordEncoder();  
 //注意要使用这个方法来进行密码校验，因为就算同一个密码，每次加密的盐值都不同  
 boolean matches = encoder.matches(userpswd, userpswdDataBase);  
 if (!matches){  
 //密码不正确  
 modelMap.addAttribute(CrowdConstant.*LOGIN\_CHECK*,CrowdConstant.*PSWD\_ERROR*);  
 return loginView;  
 }  
  
 //3.密码校验通过，将登录信息存入到Session域  
 MemberLoginVo memberLoginVo = new MemberLoginVo(memberPo.getId(),memberPo.getUsername(),memberPo.getEmail());  
  
 session.setAttribute(CrowdConstant.*LOGIN\_MEMBER*,memberLoginVo);  
  
 //4.重定向到会员中心页面，避免表单重复提交  
 return "redirect:/auth/to/member-center";  
}

#### 配置类

配置类MemberRegisterConfig，配置 登录成功后跳转到个人中心的地址



#### 个人中心页面

# 发起项目

# 展示项目

# 支付案例

# 订单

# 支付

# 部署

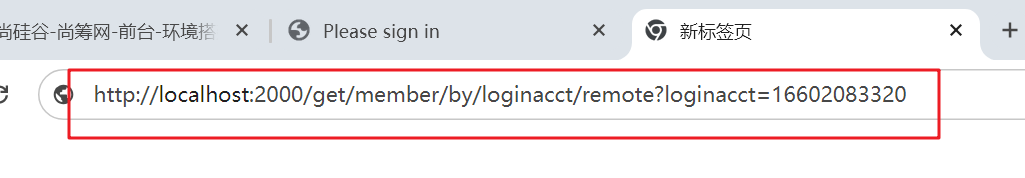
# 总结

# 出错汇总

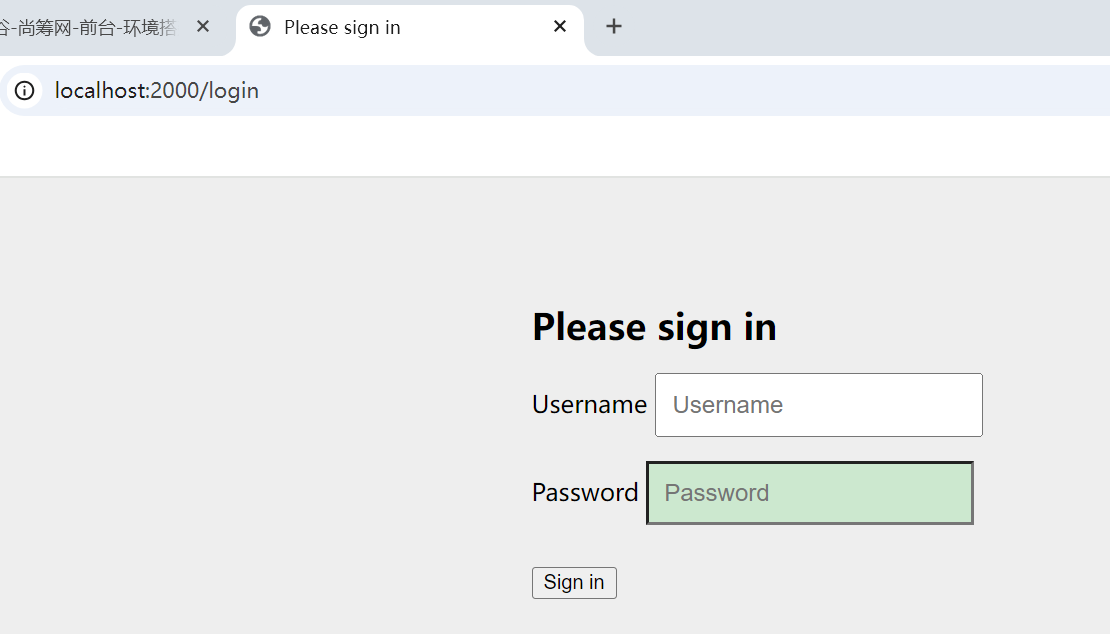
## 环境搭建

搭建mysql和redis工程的时候，单元测试没问题，在浏览器通过控制器方法测试时，请求被拦截了，要求登录

输入这个地址



跳转到了这里



出错原因：Mysql工程和redis工程引入的resultentity所在的util工程是后台的工程，这个util工程的依赖是全依赖于其父工程依赖（完全引用），导致引入SpringSecurity，所以就拦截请求，要求登录后访问。

修改：新建一个util工程，新建resultentity类，Mysql工程和redis工程引入新的util工程



## 会员登录

### 会员注册

authencation工程通过api工程的接口，调用远程redis工程的接口报错：

Field redisRemoteService in com.atguigu.crowd.handler.RegisterHandler required a bean of type 'com.atguigu.crowd.api.RedisRemoteService' that could not be found.

这是因为 authencation 工程的主启动类没加上feign客户端功能开启的注解