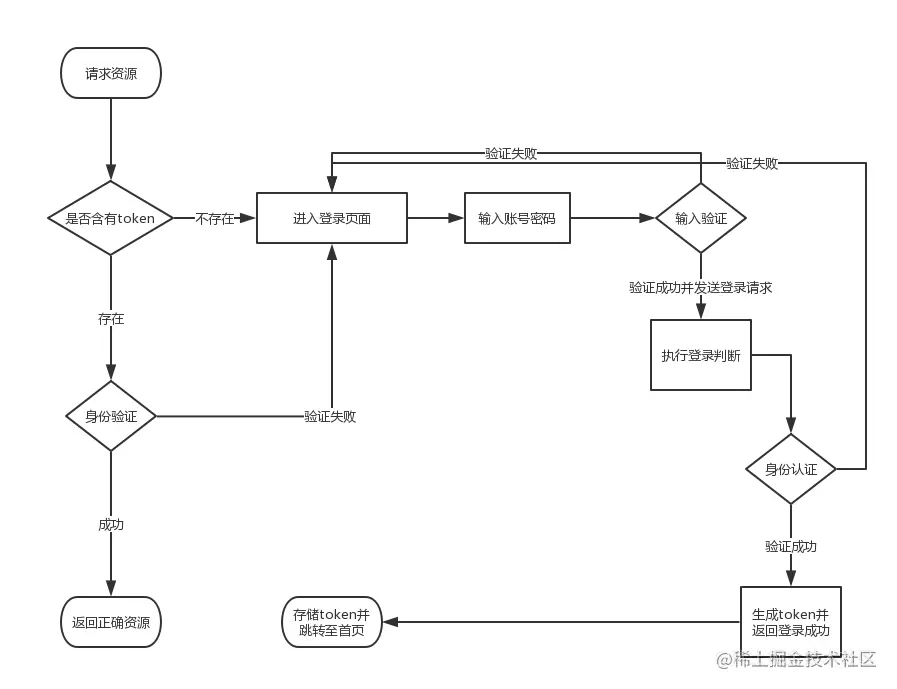
在前一个实验中我们实现了用户的登陆接口，该功能已经完成，但是身份认证的整个流程并没有完善，该流程中应该包括登陆功能、身份认证、访问拦截、退出功能，我们仅仅完成了第一步，因此本实验将会对该流程进行完善，将接下来的功能点完成。

新蜂商城第一个版本中，身份验证使用的方式是 session +拦截器实现，目的是为了验证是否登录，也就是简单的权限认证。现在这个版本是前后端分离的版本，我们用了一种新的方式，前文中我们也介绍到，登录成功后会有一个 token 值，那么我们该怎样使用这个 token 字符串呢？

## 身份认证

我们继续结合登录验证流程来讲一下：



前文中处理的流程分支是 token 不存在之后的登录处理流程，如果 token 存在我们该怎样进行身份认证？

#### 前端存储和使用 token

后端生成 token，前端存储和使用 token，在请求登录接口成功后，前端的处理方式首先会将 token 字符串存储到 localStorage 中，实现代码在 Login.vue 文件中：

**if** (this.type == 'login') {

**const** { data, resultCode } = **await** **login**({

"loginName": values.username,

"passwordMd5": this.$md5(values.password)

})

**setLocal**('token', data)

window.location.href = '/'

}

这是 web 端实现时的方案，将登录成功后的 token 值存储到 localStorage 对象中，如果是 iOS 开发或者安卓开发可能又有其对应的存储方式，不过存储的目的就是在后续请求中带上 token 值，使得后端在处理请求时可能进行身份认证。

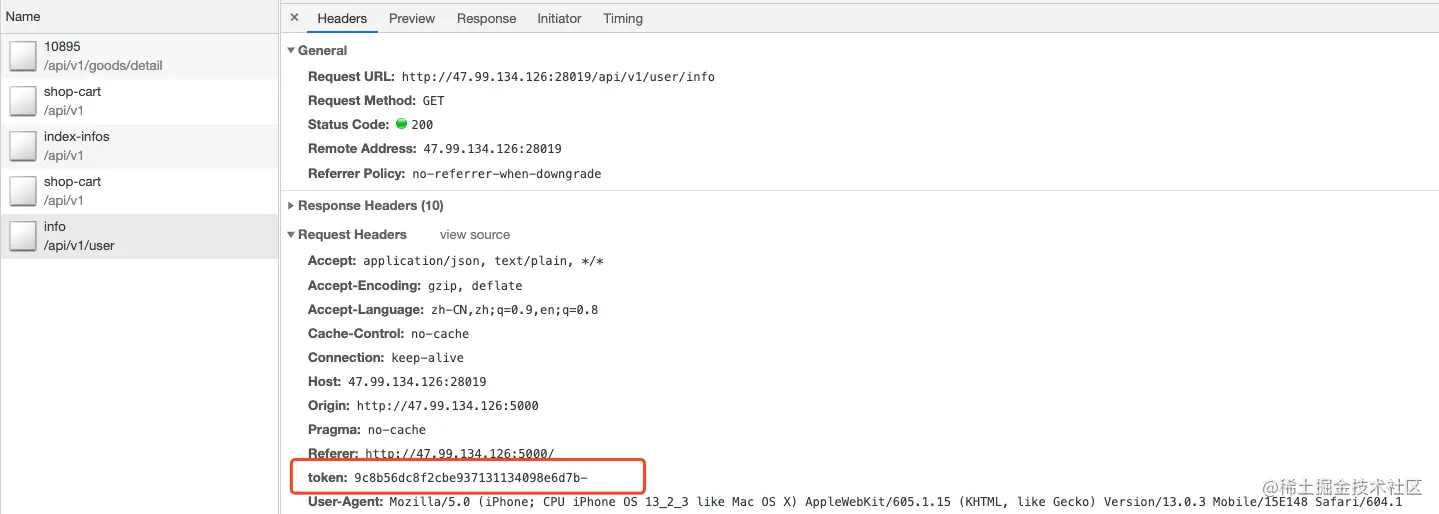
有了 token 值，在过期时间内我们都可以使用它来进行资源请求，新蜂商城 Vue 版本是如何将 token 值放到请求中的呢？

具体实现代码在 axios.js 中：

axios.defaults.headers['token'] = localStorage.**getItem**('token') || ''

axios.defaults.headers.post['Content-Type'] = 'application/json'

如果存在 token 值则将其放入到请求 header 对象中，此 header 参数的名称即为 token，打开控制台可以看到具体的请求样例：



这是一个请求个人信息的案例，在 Request Headers 中就有 token 参数，它的值就是登陆成功后存储到 localStorage中的值。

以上就是前端处理 token 值需要注意的点，接下来我们来讲解后端代码中是如何处理 token 的。

#### 后端处理 token 及身份验证

* 第一步是生成 token，这个在前一个章节中已经介绍。
* 第二步是获取到前端请求中的 token 值。
* 第三步是验证 token 值，是否存在、是否过期等等。

完成登录功能后，则需要对用户的登录状态进行验证，这里所说的登录状态保持即 “ Token 值是否存在及 Token 值是否有效 ” 。

而 Token 值是否有效则通过后端代码实现，由于大部分接口都需要进行登录验证，如果每个方法都添加查询用户数据的语句则有些多余，因此对方法做了抽取，通过注解切面的形式来返回用户信息。

* 自定义参数注解

我们自定义 @TokenToMallUser 注解，使用注解和 AOP 方式将用户对象注入到方法中：

**package** ltd.newbee.mall.config.annotation;

**import** java.lang.annotation.\*;

**@Target({ElementType.PARAMETER})@Retention(RetentionPolicy.RUNTIME)@Documentedpublic** **@interface** TokenToMallUser {

*/\*\**

*\* 当前用户在request中的名字*

*\**

*\* @return*

*\*/*

String **value**() **default** "user";

}

* 自定义方法参数解析器

新蜂商城第二版中用到的验证方式是方法参数解析器，在需要用户身份信息的方法中加上 @TokenToMallUser 注解，之后通过方法参数解析器来获得当前登录的对象信息。

自定义方法参数解析器 TokenToMallUserMethodArgumentResolver，需实现HandlerMethodArgumentResolver 类，代码如下：

**package** ltd.newbee.mall.config.handler;

**import** ltd.newbee.mall.common.Constants;**import** ltd.newbee.mall.common.NewBeeMallException;**import** ltd.newbee.mall.common.ServiceResultEnum;**import** ltd.newbee.mall.config.annotation.TokenToMallUser;**import** ltd.newbee.mall.dao.MallUserMapper;**import** ltd.newbee.mall.dao.NewBeeMallUserTokenMapper;**import** ltd.newbee.mall.entity.MallUser;**import** ltd.newbee.mall.entity.MallUserToken;**import** org.springframework.beans.factory.annotation.Autowired;**import** org.springframework.core.MethodParameter;**import** org.springframework.stereotype.Component;**import** org.springframework.web.bind.support.WebDataBinderFactory;**import** org.springframework.web.context.request.NativeWebRequest;**import** org.springframework.web.method.support.HandlerMethodArgumentResolver;**import** org.springframework.web.method.support.ModelAndViewContainer;

**import** javax.servlet.http.HttpServletRequest;**import** java.io.IOException;

**@Componentpublic** **class** **TokenToMallUserMethodArgumentResolver** **implements** **HandlerMethodArgumentResolver** {

**@Autowired**

**private** MallUserMapper mallUserMapper;

**@Autowired**

**private** NewBeeMallUserTokenMapper newBeeMallUserTokenMapper;

**public** **TokenToMallUserMethodArgumentResolver**() {

}

**public** **boolean** **supportsParameter**(MethodParameter parameter) {

**if** (parameter.hasParameterAnnotation(TokenToMallUser.class)) {

**return** true;

}

**return** false;

}

**public** Object **resolveArgument**(MethodParameter parameter, ModelAndViewContainer mavContainer, NativeWebRequest webRequest, WebDataBinderFactory binderFactory) {

**if** (parameter.getParameterAnnotation(TokenToMallUser.class) **instanceof** TokenToMallUser) {

**MallUser** mallUser = null;

*//获取请求头中的header*

**String** token = webRequest.getHeader("token");

*//验证token值是否存在*

**if** (null != token && !"".equals(token) && token.length() == Constants.TOKEN\_LENGTH) {

*//通过token值查询用户对象*

**MallUserToken** mallUserToken = newBeeMallUserTokenMapper.selectByToken(token);

**if** (mallUserToken == null || mallUserToken.getExpireTime().getTime() <= System.currentTimeMillis()) {

NewBeeMallException.fail(ServiceResultEnum.TOKEN\_EXPIRE\_ERROR.getResult());

}

mallUser = mallUserMapper.selectByPrimaryKey(mallUserToken.getUserId());

*//用户不存在*

**if** (mallUser == null) {

NewBeeMallException.fail(ServiceResultEnum.USER\_NULL\_ERROR.getResult());

}

*//是否封禁*

**if** (mallUser.getLockedFlag().intValue() == 1) {

NewBeeMallException.fail(ServiceResultEnum.LOGIN\_USER\_LOCKED\_ERROR.getResult());

}

*//返回用户对象供对应的方法使用*

**return** mallUser;

} **else** {

NewBeeMallException.fail(ServiceResultEnum.NOT\_LOGIN\_ERROR.getResult());

}

}

**return** null;

}

}

该方法的执行逻辑如下：

首先获取请求头中的 token 值，不存在则返回错误信息给前端，存在则继续后续流程。

通过 token 值来查询 MallUserToken 对象，是否存在或者是否过期，不存在或者已过期则返回错误信息给前端，正常则继续后续流程。

通过 MallUserToken 对象中的 userId 字段来查询 MallUser 用户对象，判断是否存在和是否已被封禁，用户状态正常则返回用户对象供对应的方法使用，否则返回错误信息。

* 配置方法参数解析器

最后在 WebMvcConfigurer 中配置 TokenToMallUserMethodArgumentResolver 使其生效，代码如下：

**package** ltd.newbee.mall.config;

**import** ltd.newbee.mall.config.handler.TokenToMallUserMethodArgumentResolver;**import** org.springframework.beans.factory.annotation.Autowired;**import** org.springframework.context.annotation.Configuration;**import** org.springframework.web.method.support.HandlerMethodArgumentResolver;**import** org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

**import** java.util.List;

**@Configurationpublic** **class** **NeeBeeMallWebMvcConfigurer** **implements** **WebMvcConfigurer** {

**@Autowired**

**private** TokenToMallUserMethodArgumentResolver tokenToMallUserMethodArgumentResolver;

*/\*\**

*\* TokenToMallUser 注解处理方法*

*\**

*\* @param argumentResolvers*

*\*/*

**public** **void** **addArgumentResolvers**(List<HandlerMethodArgumentResolver> argumentResolvers) {

argumentResolvers.add(tokenToMallUserMethodArgumentResolver);

}

}

## 身份验证测试

这样，需要进行登录判断的接口加上 @TokenToUser 注解即可，之后再进行相应的逻辑判断，十三也增加了两个接口进行状态保持的测试，请看下面代码：

**@GetMapping(value = "/test1")**

**@ApiOperation(value = "测试接口", notes = "方法中含有@TokenToMallUser注解")**

**public** Result<String> **test1**(**@TokenToMallUser** MallUser user) {

*//此接口含有@TokenToMallUser注解，即需要登陆验证的接口。*

**Result** result = null;

**if** (user == null) {

*//如果通过请求header中的token未查询到用户的话即token无效，登陆验证失败，返回未登录错误码。*

result = ResultGenerator.genErrorResult(416, "未登录！");

**return** result;

} **else** {

*//登陆验证通过。*

result = ResultGenerator.genSuccessResult("登陆验证通过");

}

**return** result;

}

**@GetMapping(value = "/test2")**

**@ApiOperation(value = "测试接口", notes = "方法中无@TokenToMallUser注解")**

**public** Result<String> **test2**() {

*//此接口不含@TokenToMallUser注解，即访问此接口无需登陆验证，此类接口在实际开发中应该很少，为了安全起见应该所有接口都会做登陆验证。*

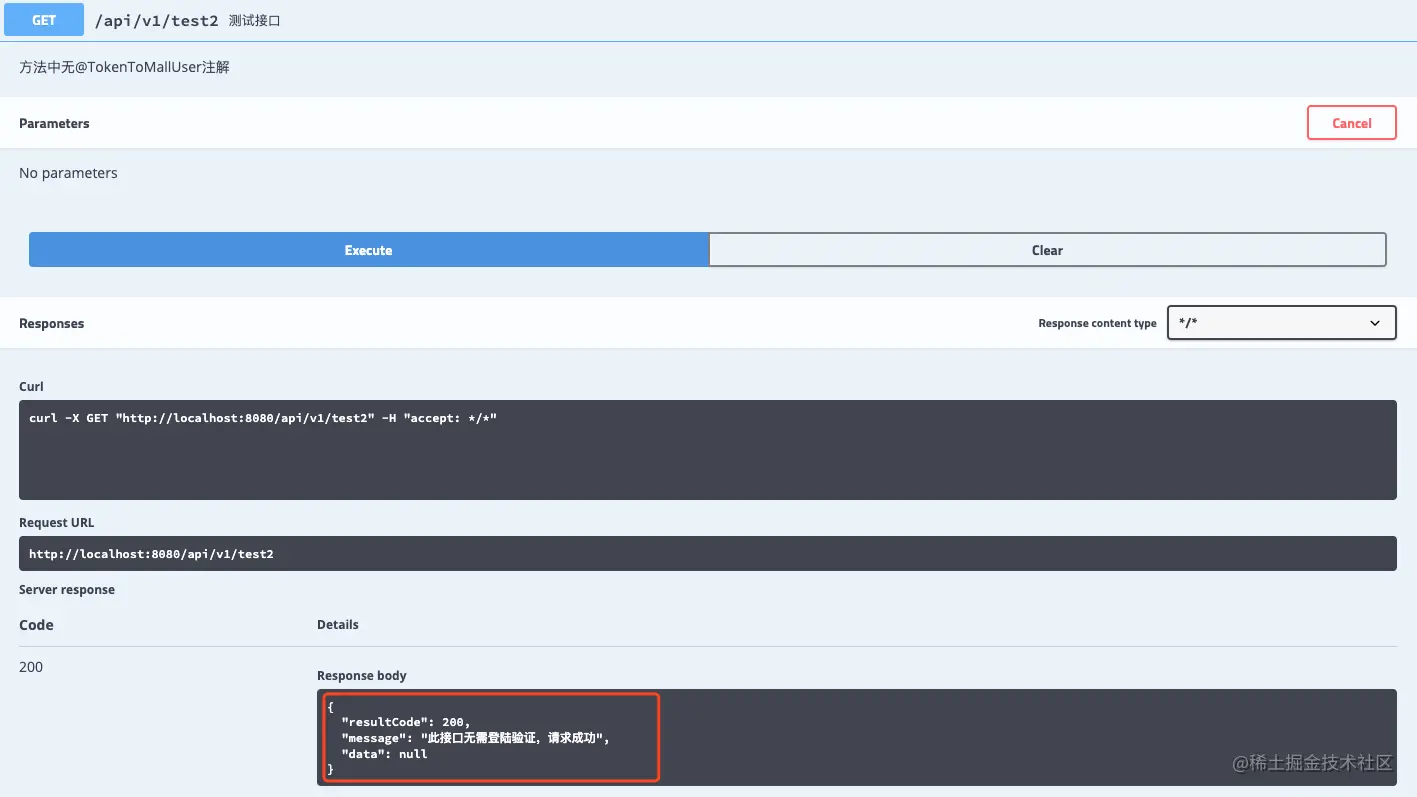
**Result** result = ResultGenerator.genSuccessResult("此接口无需登陆验证，请求成功");

*//直接返回业务逻辑返回的数据即可。*

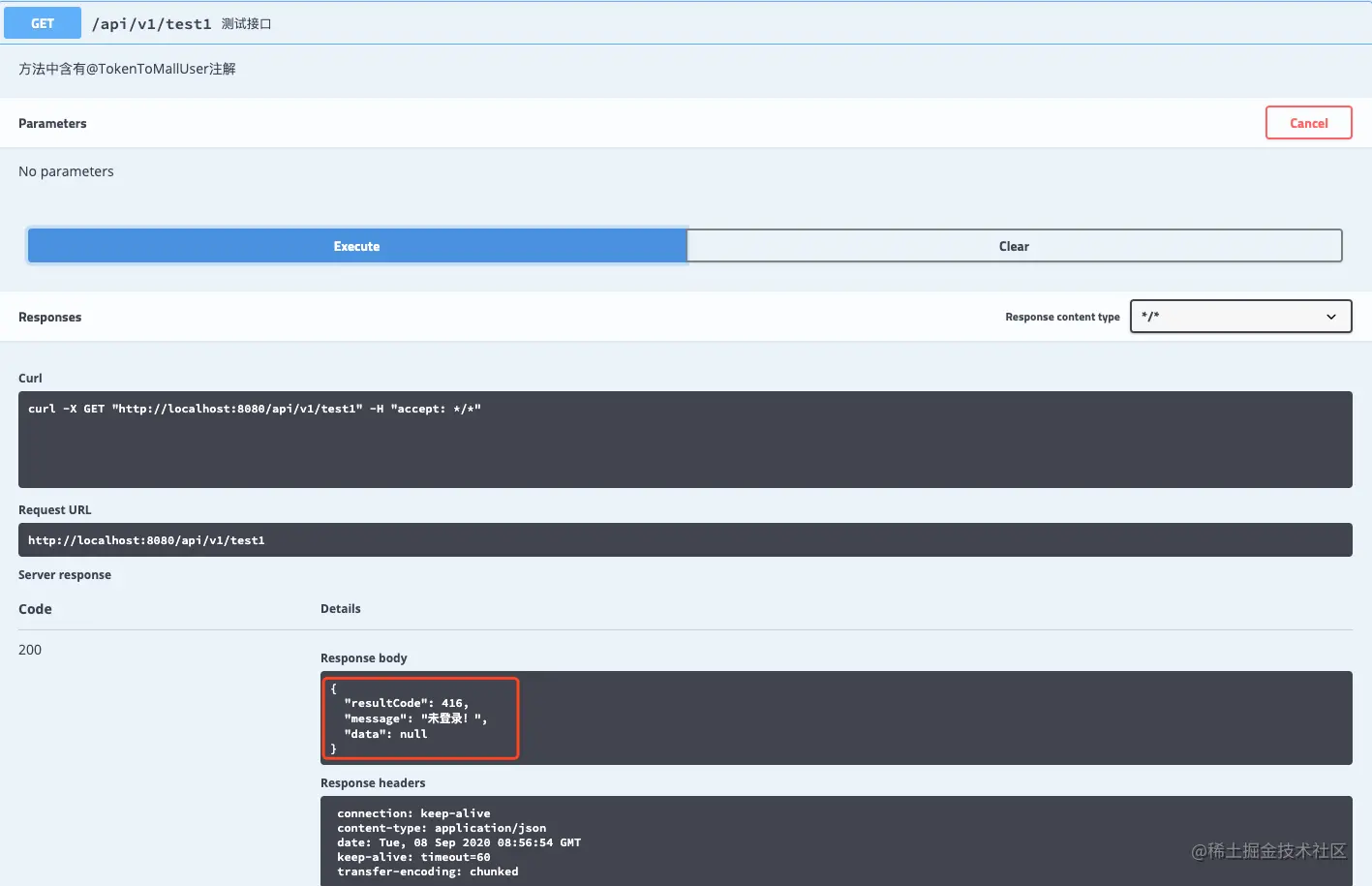
**return** result;

}

启动项目并访问 Swagger 接口页面，首先测试无需身份验证的接口，可以看到，由于不需要登录验证，接口直接返回了数据。



之后是测试需要身份验证的接口，如下图所示，由于是直接发起请求，并没有在请求头中放入 token 参数，在 Curl 那一栏中也可以看到，并没有 token 传输到后端，所以直接返回 416 错误码，提示未登录。



因为 token 参数是我们自定义的，如果没有在编码层面做修改，Swagger 肯定不会在页面中生成 token 参数，这里我们需要修改 Swagger 的配置，使得可以在 swagger-ui 页面可以传输 token 参数，Swagger 配置类修改如下：

**@Configuration@EnableSwagger2public** **class** **Swagger2Config** {

**@Bean**

**public** Docket **api**() {

**ParameterBuilder** tokenParam = **new** **ParameterBuilder**();

List<Parameter> swaggerParams = **new** **ArrayList**<Parameter>();

tokenParam.name("token").description("用户认证信息")

.modelRef(**new** **ModelRef**("string")).parameterType("header")

.required(false).build(); *//header中的ticket参数非必填，传空也可以*

swaggerParams.add(tokenParam.build()); *//根据每个方法名也知道当前方法在设置什么参数*

**return** **new** **Docket**(DocumentationType.SWAGGER\_2)

.apiInfo(apiInfo())

.ignoredParameterTypes(MallUser.class)

.select()

.apis(RequestHandlerSelectors.basePackage("ltd.newbee.mall.api"))*// 修改为自己的 controller 包路径*

.paths(PathSelectors.any())

.build()

.globalOperationParameters(swaggerParams);

}

**private** ApiInfo **apiInfo**() {

**return** **new** **ApiInfoBuilder**()

.title("新蜂商城接口文档")

.description("swagger接口文档")

.version("2.0")

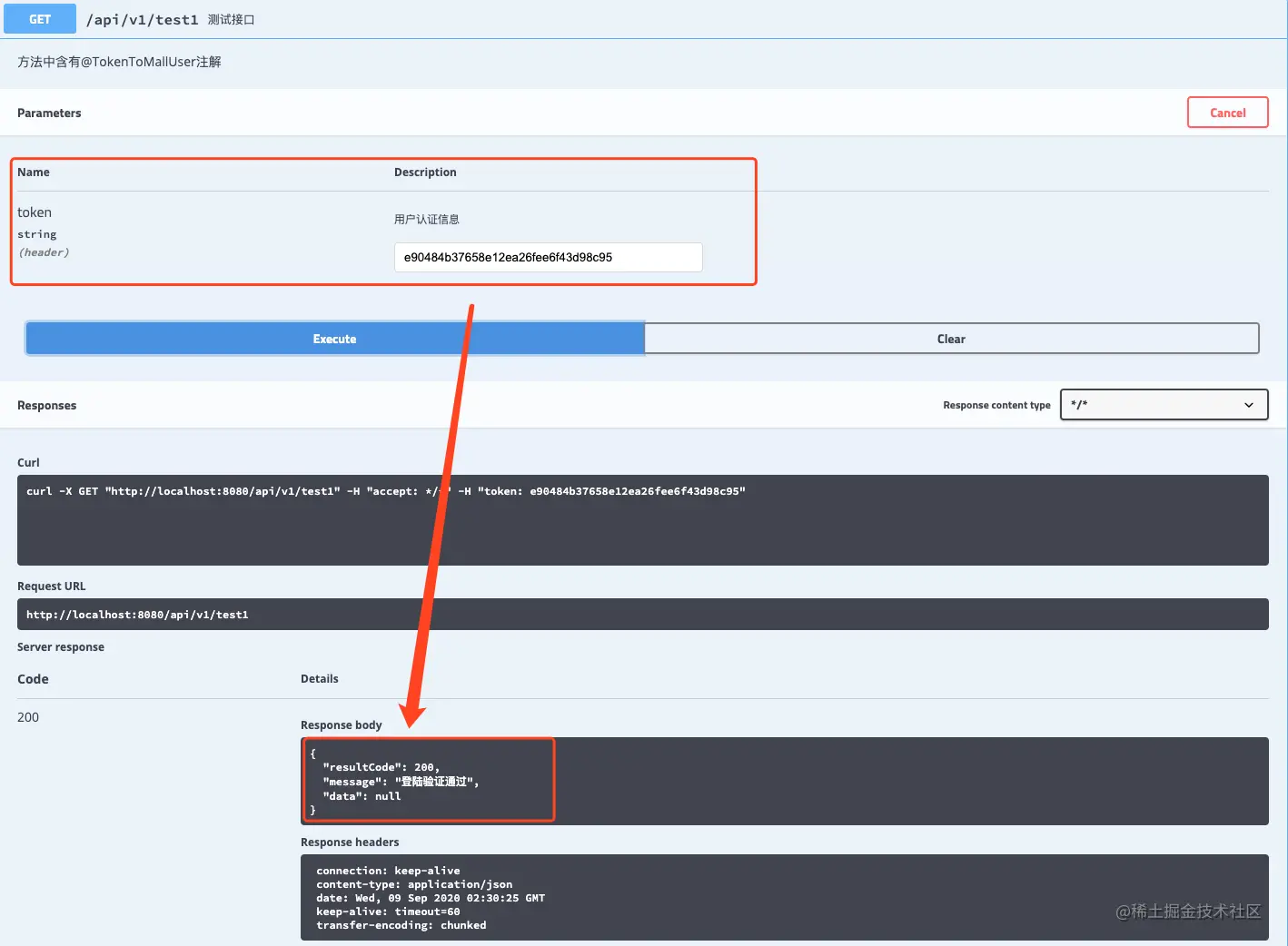
.build();

}

}

使用 ParameterBuilder 来定义 tokenParam，token 是一个字符串，且发送请求时是放在 Request Header 中，所以做了如下配置： modelRef(new ModelRef("string")).parameterType("header")，最后将该参数放入 globalOperationParameters 中，这里我们只配置了一个 token 参数，如果有其他全局参数也可以通过这种方式来配置。

重启项目并打开 Swagger 页面，可以看到在参数栏已经有 token 参数，且类型为请求头中的 string 字符串。执行登录接口获取一个 token 值并填入 input 框中，可以看到此时的返回码是 200，且 message 为”登录验证通过“，在 Curl 那一栏中也可以看到此时的传参中增加了 token 请求头，验证通过。



如果请求中不存在 Token 或者 Token 值是错误的，则验证身份失败，返回错误码 416，而如果填入正确的 Token 值，则返回登录验证成功，这样我们的身份验证编码就完成了。

## 用户模块接口完善

讲完注册登录和身份认证流程，我们继续来完善用户模块，如下图所示，分别是”我的“页面和”账号管理“页面：



”我的“页面需要展示用户信息，”账号管理“页面则需要展示用户的部分信息，同时还有更改用户信息以及退出登录的功能，这些都需要我们后端来进行处理，所以需要增加三个接口：

* 获取用户信息接口
* 修改用户信息接口
* 退出登录接口

新增代码如下：

**@GetMapping("/user/info")**

**@ApiOperation(value = "获取用户信息", notes = "")**

**public** Result<NewBeeMallUserVO> **getUserDetail**(**@TokenToMallUser** MallUser loginMallUser) {

*//已登录则直接返回*

**NewBeeMallUserVO** mallUserVO = **new** **NewBeeMallUserVO**();

BeanUtil.copyProperties(loginMallUser, mallUserVO);

**return** ResultGenerator.genSuccessResult(mallUserVO);

}

**@PutMapping("/user/info")**

**@ApiOperation(value = "修改用户信息", notes = "")**

**public** Result **updateInfo**(**@RequestBody** **@ApiParam("用户信息")** MallUserUpdateParam mallUserUpdateParam, **@TokenToMallUser** MallUser loginMallUser) {

**Boolean** flag = newBeeMallUserService.updateUserInfo(mallUserUpdateParam, loginMallUser.getUserId());

**if** (flag) {

*//返回成功*

**Result** result = ResultGenerator.genSuccessResult();

**return** result;

} **else** {

*//返回失败*

**Result** result = ResultGenerator.genFailResult("修改失败");

**return** result;

}

}

**@PostMapping("/user/logout")**

**@ApiOperation(value = "登出接口", notes = "清除token")**

**public** Result<String> **logout**(**@TokenToMallUser** MallUser loginMallUser) {

**Boolean** logoutResult = newBeeMallUserService.logout(loginMallUser.getUserId());

logger.info("logout api,loginMallUser={}", loginMallUser.getUserId());

*//登出成功*

**if** (logoutResult) {

**return** ResultGenerator.genSuccessResult();

}

*//登出失败*

**return** ResultGenerator.genFailResult("logout error");

}

由于这三个接口都需要用户登陆状态下才能正常的请求，所以方法定义时都使用了 @TokenToMallUser 注解。

接下来是这三个接口的实现逻辑。

****获取用户信息接口****

使用 @TokenToMallUser 注解 已经得到了当前登录的用户对象 MallUser，因此不用再去数据库中查询，直接返回即可，不过这里并没有直接返回 MallUser 对象，而是新定义了一个 NewBeeMallUserVO 对象，因为 MallUser 对象字段较多，但是有些字段前端用不到，所以就定义了 NewBeeMallUserVO 对象，返回三个页面所需要的字段即可。

****修改用户信息接口****

定义了 MallUserUpdateParam 对象来接收用户修改的信息字段，需要修改的字段主要有昵称、密码、个性签名，并使用 @RequestBody 注解来接收，之后调用业务层的 updateUserInfo() 方法来进行入库操作，将这些字段 update 掉。

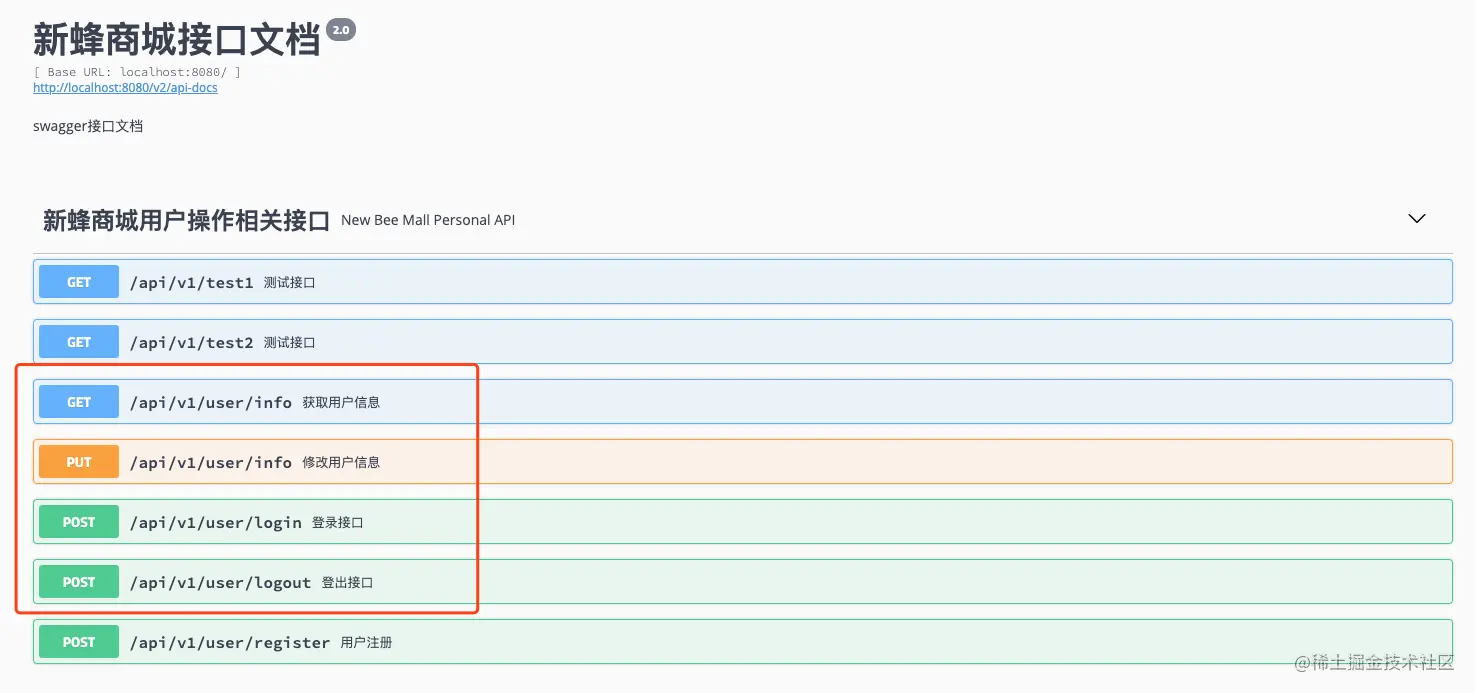
****登出接口****

这个接口的逻辑比较简单，只需要将该用户在 token 表中的记录删除掉即可，也就是将当前的 token 值设置为无效的，既然登出了肯定不能让当前的 token 值可以继续进行身份验证。

## 用户模块接口测试

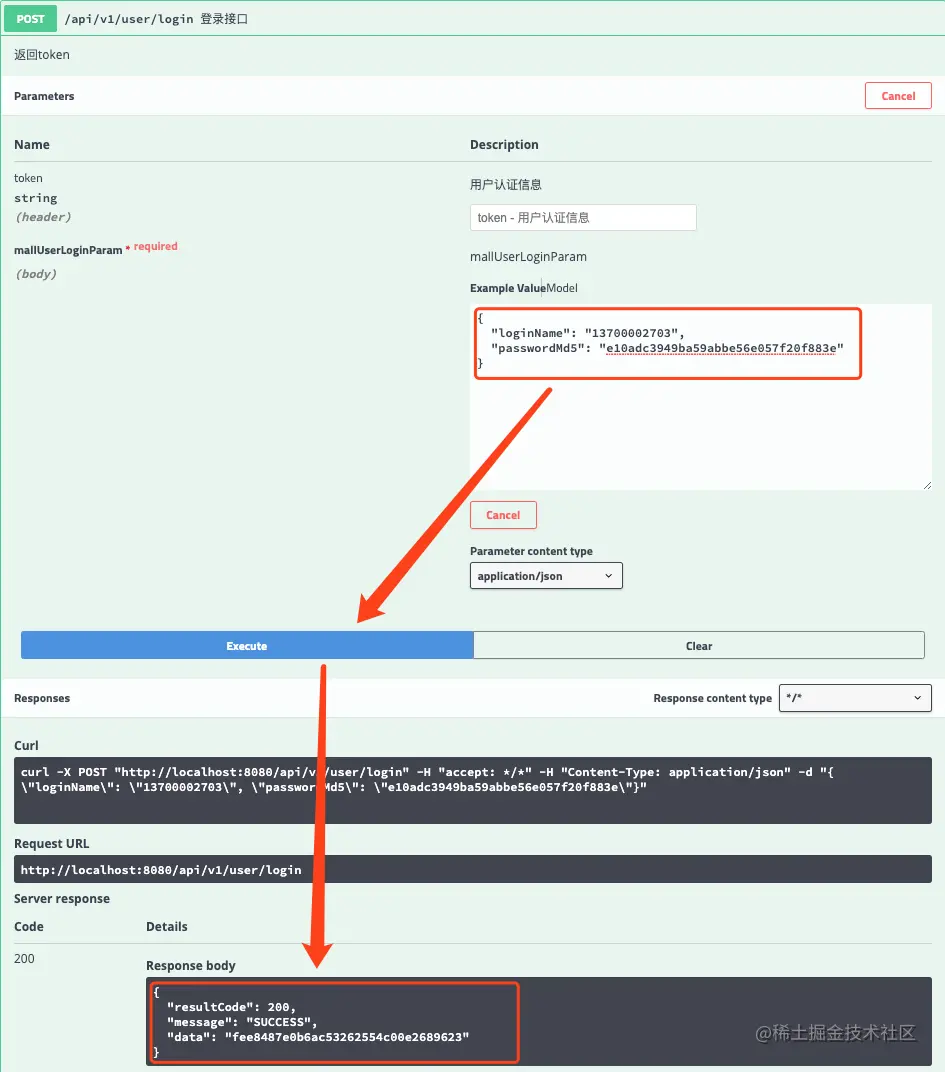
最后，我们通过 Swagger 页面来测试一下这些接口。

重启项目，打开 swagger-ui 页面：



#### 登录

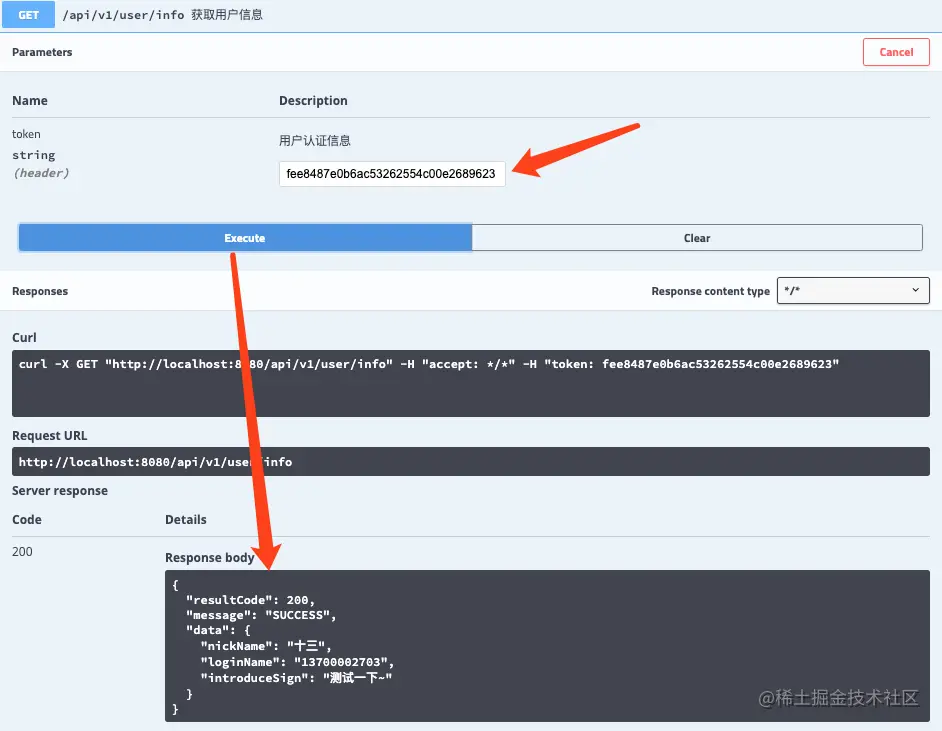
首先我们访问登录接口，拿到一个可以正常进行身份认证的 token 字符串，如下图所示：



由于在 Swagger 配置中，将 token 设置成全局的 header 参数，所以每个请求上都会有 token 参数，不过不填写也依然可以正常调用，有些接口不需要身份认证的。

#### 获取用户信息

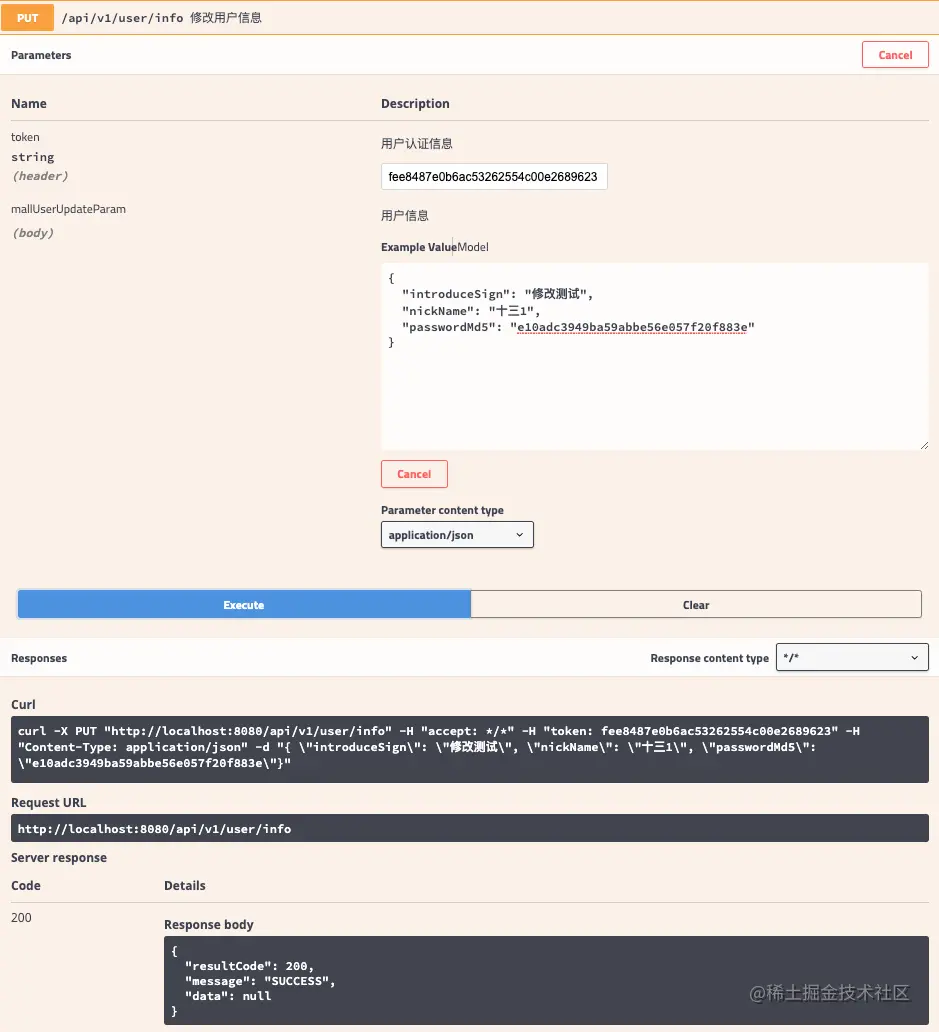
点开“获取用户信息”，在 token 输入框中填入登录接口返回的 token 值，之后点击”Execute“按钮，即可得到用户信息数据，如下图所示：



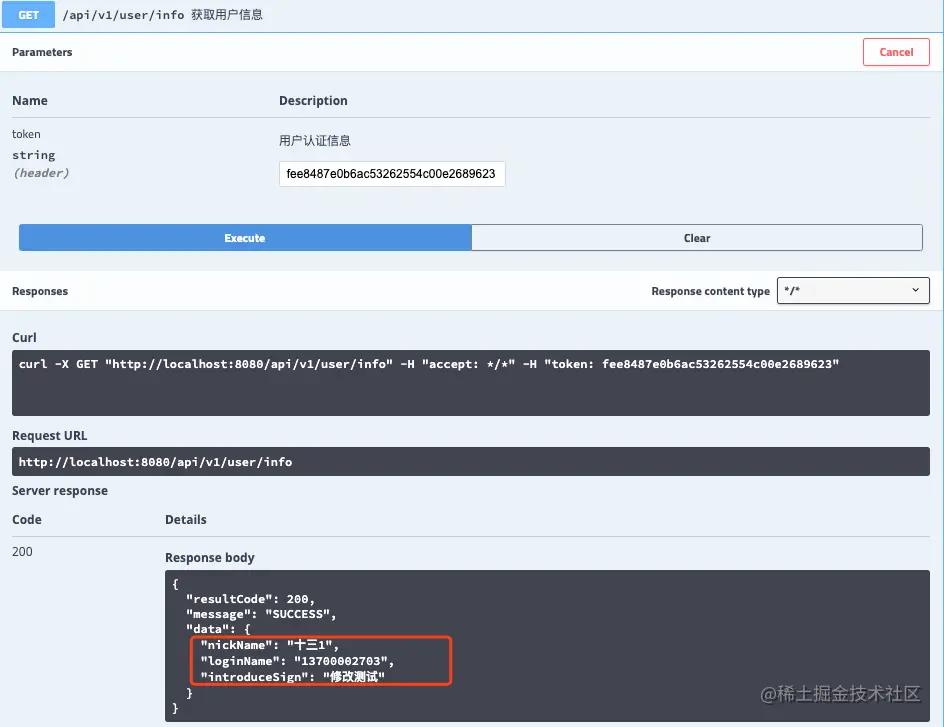
后端需要处理的内容就是这些，根据前端传入的参数返回相应的结果，至于前端怎么处理这些数据、怎么显示，都是前端开发人员所考虑的，我们后端保证接口参数和响应结果正常即可。

#### 修改用户信息

点开“修改用户信息”，在 token 输入框中填入登录接口返回的 token 值，并且将需要修改的字段内容放入请求体 body 中，之后点击”Execute“按钮，即可完成修改操作，如下图所示：



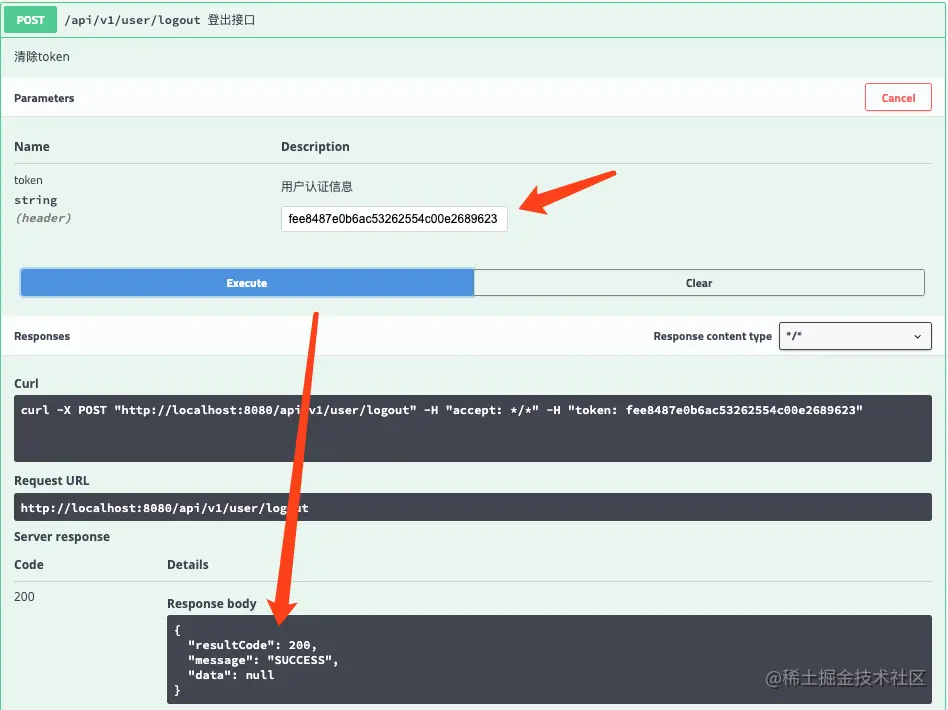
此时后端响应结果为修改成功，我们再次请求用户信息接口，得到了如下结果：



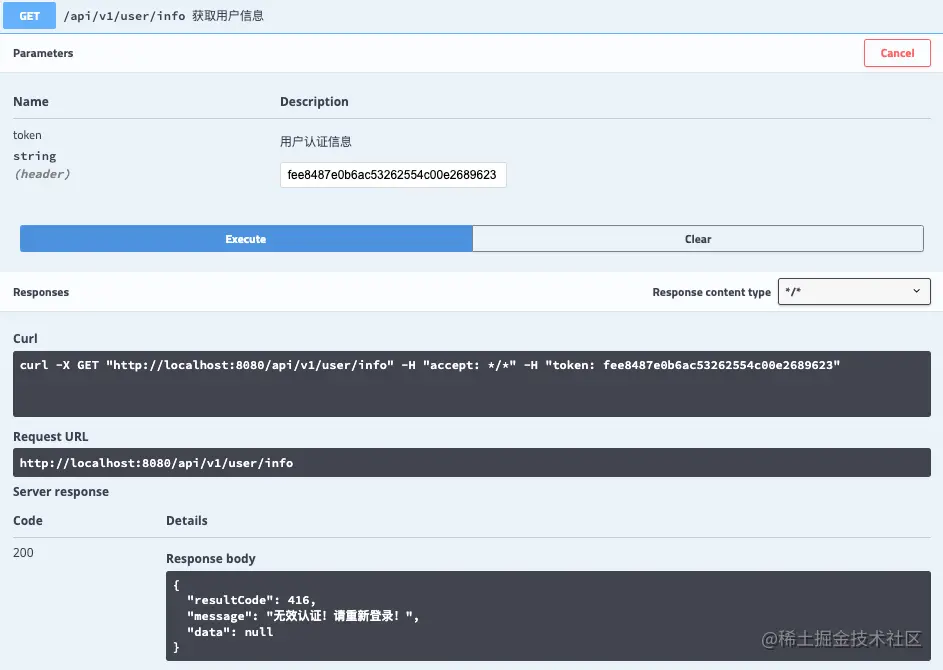
昵称、介绍字段都已经是修改后的内容了，修改接口测试成功。

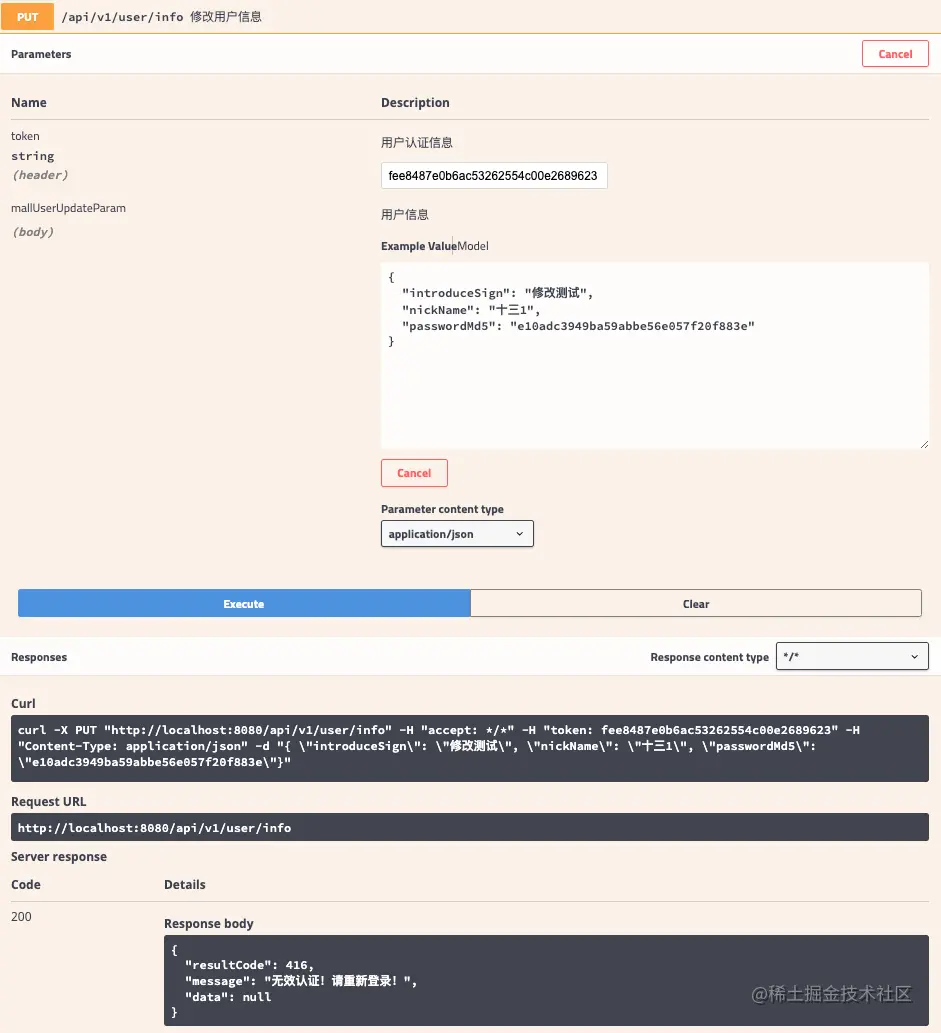
#### 退出登录

点开“登出接口”，在 token 输入框中填入登录接口返回的 token 值，之后点击”Execute“按钮，即可完成修改操作，如下图所示：



后端返回的是接口处理成功的响应，此时我们再用这个 token 进行接口请求，会得到如下结果：





因为执行了退出登录的逻辑，将 token 信息删除，所以原来的 token 已经失效了，再次使用失效的 token 值来进行接口请求，后端响应的肯定是”无效认证！请重新登录！“。

接口测试完成。