

Hw_3 SpringCloud-Fi&Shipping 物流管理系统系统设计报告

🕒 报告完成时间：2023-6-17 20:11

撰写人：-20231266 禹浩男 -20271006 付柏逢

小组成员：-20231107 潘明豪

仓库链接：<https://github.com/laven00/trans>

用户测试账号：pmh 密码：123

管理员测试账号：yhn 密码：123

#0 系统开发简介

HomeWork3主要工作为将第二次的基于Vue和SpringBoot的普通前后端工程重建为基于SpringCloud的微服务架构。

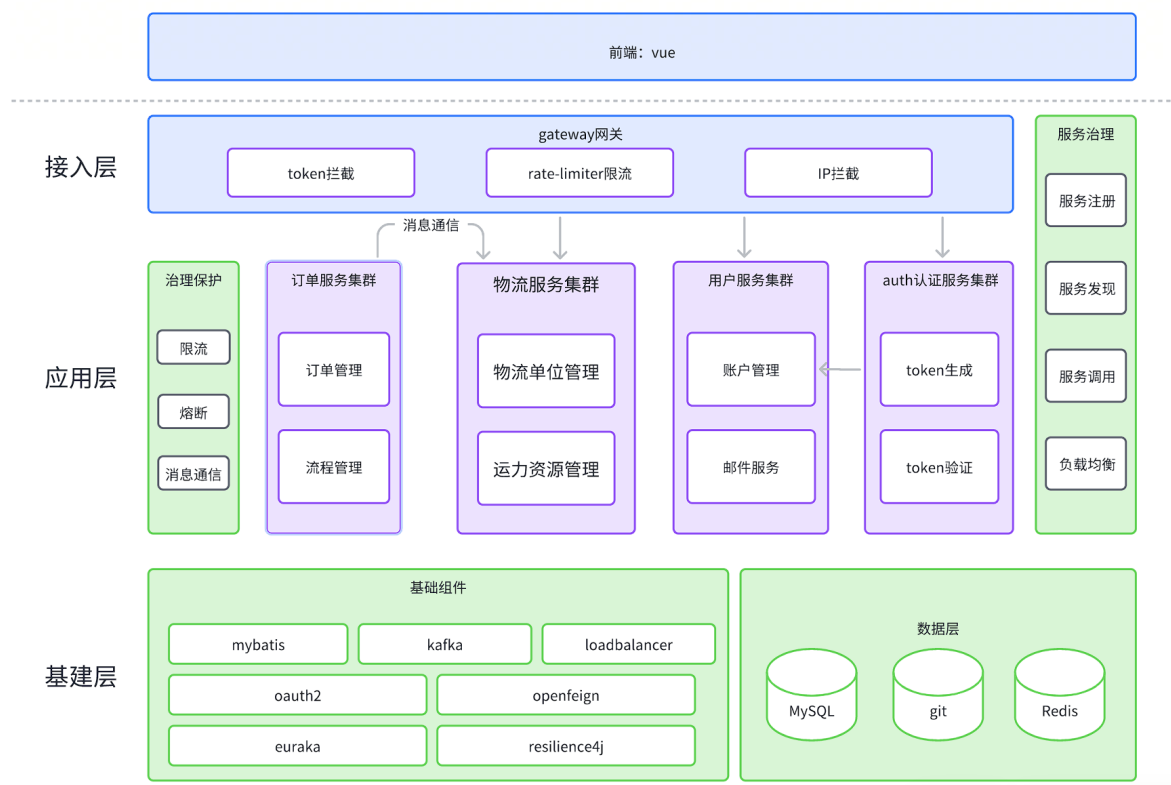
此部分工作需要将原先的后端工程根据业务逻辑拆分成诸个微服务，并使用注册到微服务管理中心（本项目使用Eureka进行管理）对于每个微服务，需要配置它们的服务提供者和消费者。服务提供者需要向Eureka注册中心注册自己的服务实例，服务消费者需要从Eureka注册中心获取服务提供者的信息。对于微服务之间的调用我们则使用openFeign进行调用，Feign是一种声明式的HTTP客户端，可以根据注解自动生成HTTP请求。

此外我们使用Spring Cloud Gateway来作为我们的网关服务，其最重要的作用是提供路由转发功能，可以根据请求的路由规则将请求转发到相应的微服务实例，通过Spring Cloud Gateway，我们可以将多个微服务的API聚合到一个统一的API入口，使得客户端可以通过一个接口访问多个微服务的功能，提高了整个应用的可维护性和可扩展性。此外网关中还可以配置过滤器来进行鉴权、限流等操作。本项目中我们通过token进行鉴权并对其可访问的接口进行限制。

最后在HomeWork2后期有一些仍在开发中的模块，我们也对其进行了补全开发，目前完成实现管理员对物流资源的获取以及分配，基本完成系统的整体开发。

#1 微服务模块划分

1.1 整体服务架构



1.2 Eureka微服务

Eureka是一个开源的微服务发现框架，本项目集成Eureka作为微服务注册中心，实现对微服务的管理和微服务之间的通信调用。其部分原理如下：

1. **注册中心：**Eureka Server作为注册中心，本项目的所有微服务，包括

gateway-service网关微服务

Jwt-service认证微服务

User-service账户微服务

Order-service订单微服务

Trans-unit-service物流微服务

都将自身的信息注册到Eureka Server上，包括服务名称、IP地址、端口号等信息。

2. **服务提供者：**当一个微服务启动并要提供其他微服务的接口时，它会向Eureka Server注册自己的信息，包括服务名称、IP地址、端口号等信息。
3. **服务消费者：**当一个微服务需要调用另一个微服务时，它会向Eureka Server发出一个请求，并在响应中获取可用的服务实例列表。然后，它可以根据负载均衡策略选择其中一个实例进行调用。

4. **心跳机制**：Eureka Server会定期向注册在上面的微服务发送心跳请求，以确保它们仍然可用。如果一个服务在一定时间内没有发送心跳请求，那么Eureka Server会将其从注册表中删除。

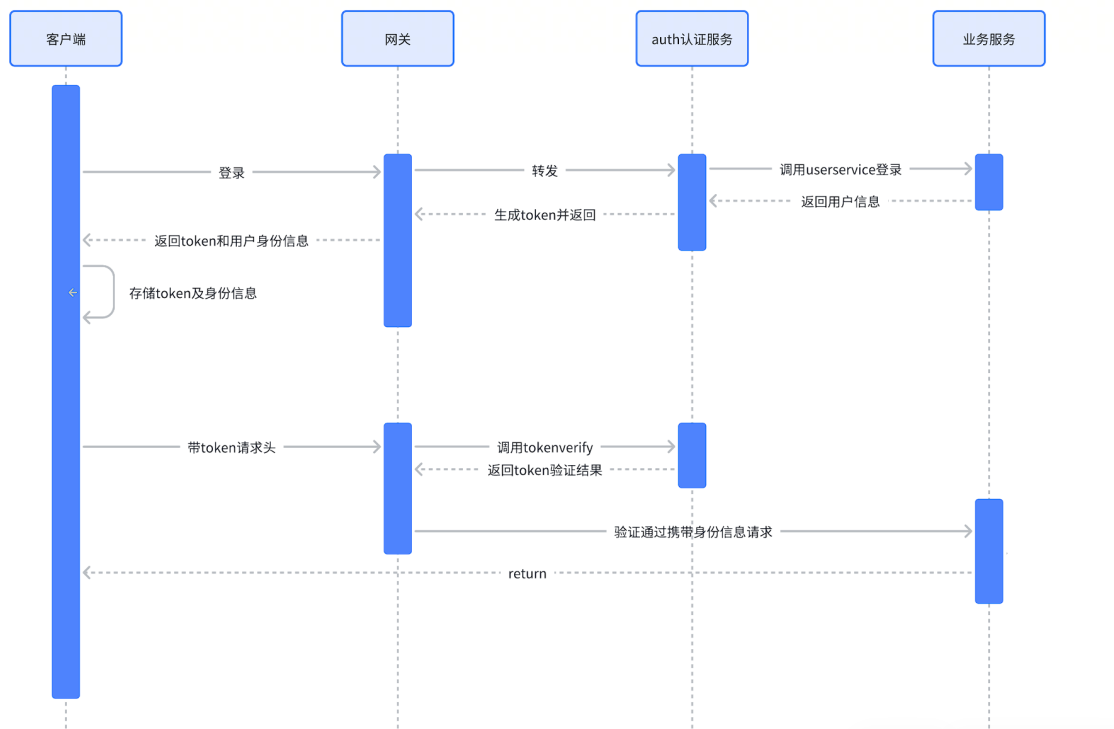
1.3 Jwt认证微服务

JWT是一种开放标准，用于在不同的应用程序之间安全地传输信息。它通过使用数字签名来验证和信任信息，从而允许在各种场景下安全地传输数据。

本系统使用JWT验证 的原理如下：

1. **生成 Token**：在用户登录时，服务器会生成一个包含用户用户名、id和identity信息的Token，Token 包含三个部分：头部、荷载和签名。头部包含加密算法和类型信息，荷载包含用户信息和过期时间等，签名则用于验证 Token 的真实性。
2. **发送 Token**：服务器将生成的Token 发送给客户端，同时将该用户的用户名，identity信息存储在云端redis数据库中，客户端将Token 存储在本地的存储介质中。
3. **发送请求**：客户端向服务器发送请求时，将Token 放在请求头中。
4. **验证 Token**：服务器在接收到请求后，使用之前生成Token 时使用的密钥对Token 进行解密和验证签名，然后检查Token 中包含的用户信息是否与云端redis中存储的用户信息匹配，以及Token 是否已过期。
5. **响应请求**：如果服务器验证Token 成功，则会响应请求并返回请求的资源，否则会返回错误信息。

以下为交互泳道图。



1.4 gateWay网关

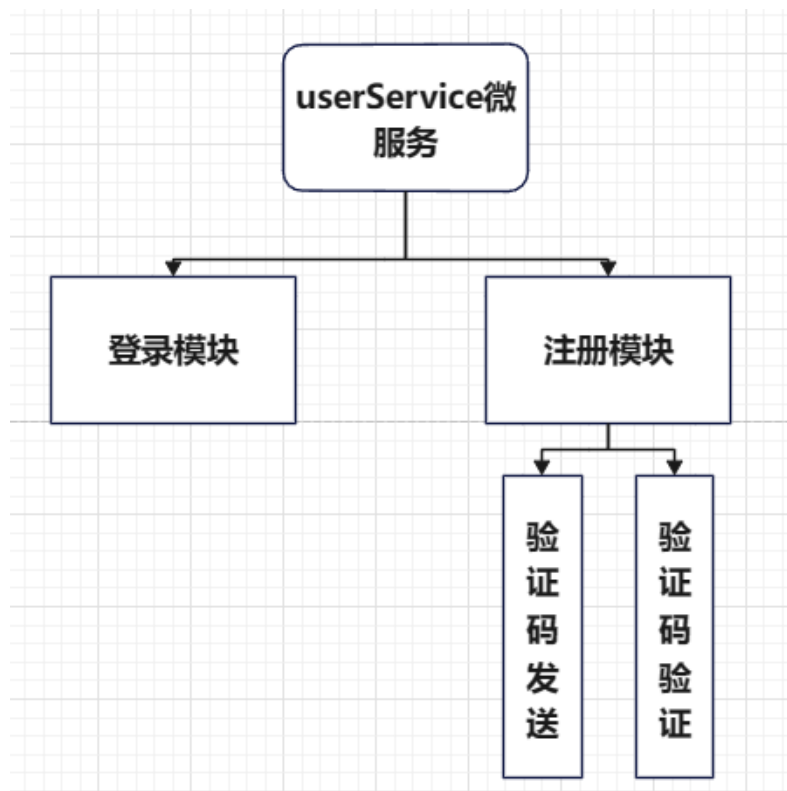
Spring Cloud Gateway 是基于 Spring Boot 2 和 Spring Framework 5 的 API 网关，核心原理是基于 Spring WebFlux 框架，它使用 Reactor 模式处理 HTTP 请求，提供了一种简单而有效的方式来管理微服务的请求路由、过滤和负载均衡：

在本项目中gateway有以下几种用途

- API 路由：通过Spring Cloud Gateway 将请求路由到不同的微服务实例或者服务群组，实现统一的 API 接口。
- 请求过滤：通过Spring Cloud Gateway 过滤器实现请求的过滤和修改，例如限流和对请求的token验证。
- 负载均衡：通过Spring Cloud Gateway 集成 Ribbon 实现负载均衡，将请求分发到多个微服务实例中。

1.5 UserService微服务

该微服务主要为用户相关功能的微服务。主要涉及的业务有用户的登录注册以及衍生的注册时邮箱验证码的发送与验证。下图为微服务模块图。前几次提交的报告已经提及不再赘述。



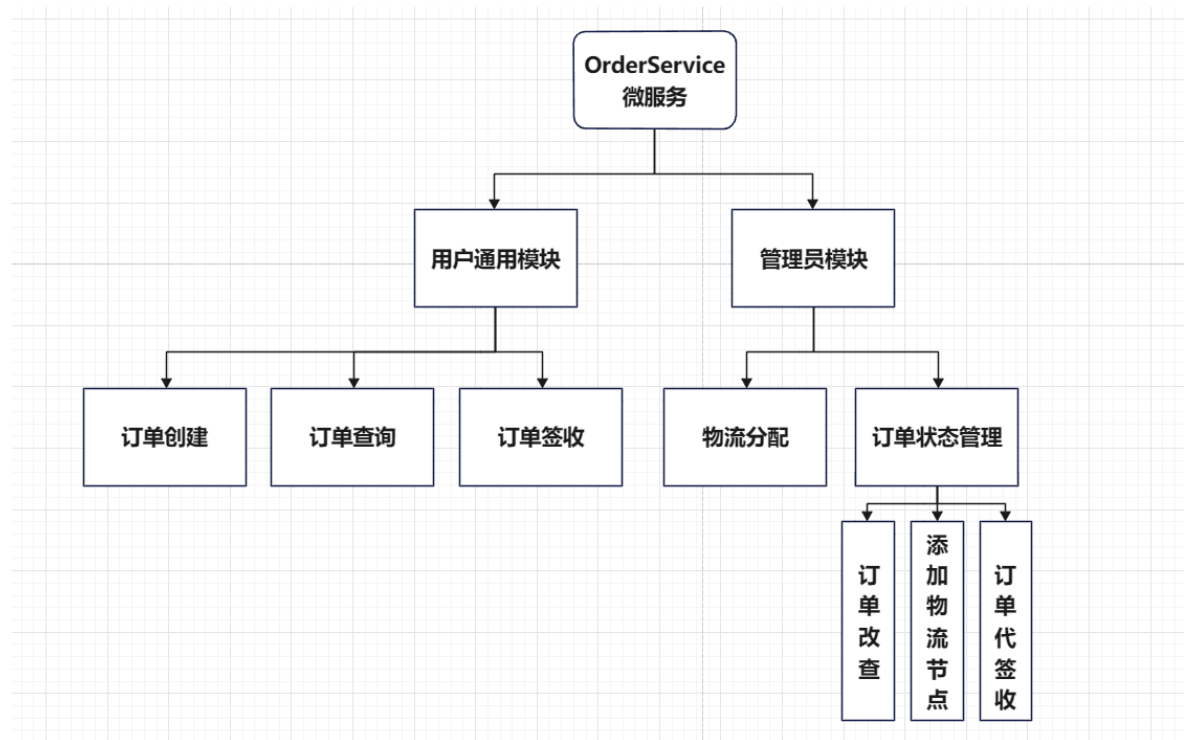
1.6 OrderService微服务

该微服务主要涉及订单的相关操作，下图为微服务模块图。主要分为用户通用模块和管理员模块。

用户通用模块主要包含订单的创建和订单的查询以及签收。进入页面时通过用户的手机号对数据库进行查询，分别返回前端其收件人为他和发件人为他的物流信息并展示。对于收件人为他的快递可以点击确认签收进行签收。用户也可以填写信息来发起一个物流订单。

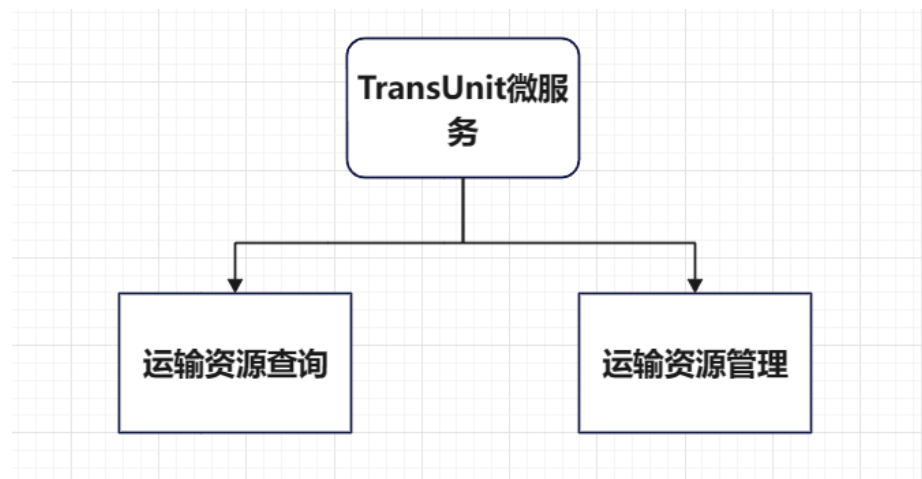
管理员模块主要包含订单状态管理和物流分配模块。物流分配模块会调用TranUnit微服务获取可用物流，前端选择物流后进入物流分配接口，对订单分配物流公司。订单管理模块则进行对订单的增删改查操作，对于运输中的订单，订单管理模块可以增加其运输的中间节点供用户查看。

t



1.7 TranUnit微服务

该微服务主要涉及运输单位的相关操作，目前含有的功能有运输单位的查询及返回、运力资源的增删改查。该服务未来还可能拓展与物流公司相关的其他业务逻辑，所以单独提取出来以备横向拓展。管理员需要分配订单时会访问该微服务，将可用的运输资源传回前端并渲染供管理员选择。OrderService也会调用该微服务的运输资源管理模块，分配资源时对运力资源的rest字段进行修改。




#2 数据库设计

3.1 订单信息表

该表主要用于记录存入的订单信息。除和订单相关的基础信息外，较重要字段为

- 1. id：自增的订单id，为订单的唯一标识
- 2. transunitid：运力资源预留字段，后期和表3 运力资源表查询订单承运单位等信息。
- 3. type：运输方式
- 4. content：备注

order	
	id: int
	transunitid: int
	userid: int
	createtime: datetime
	setout: varchar(255)
	destination: varchar(255)
	type: varchar(255)
	esttime: datetime
	state: varchar(10)
	weight: double
	sendphone: varchar(255)
	recphone: varchar(255)
	sendname: varchar(255)
	recname: varchar(255)
	content: varchar(255)
	sendaddress: varchar(255)
	recaddress: varchar(255)

3.2 用户信息表

该表主要用于记录注册的用户信息。

除用户基本字段外，**identity**字段用于记录用户等级，用于登录时的鉴权操作。

user	
	id: int
	username: varchar(255)
	email: varchar(255)
	password: varchar(255)
	identity: int
	phone: varchar(255)

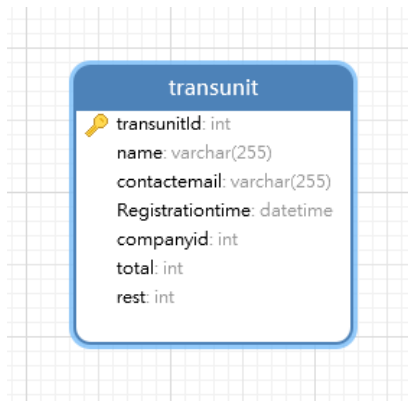
3.3 运输公司信息表

该表主要用于物流公司信息的管理。



3.4 运力资源表

该表主要为各物流公司旗下运输单位的管理，其rest是剩余可用的物流单位，total是总共的物流单位。



3.5 订单追踪记录表

该表主要为订单的各运输节点进行记录。



#3 系统模块设计

注：已有功能如系统登陆请查看报告2，此处展示新增物流分配功能

4.1 主页



4.1 物流公司选择页面

管理员点击分配发货后，会访问transunit微服务，微服务根据订单需要的运力资源返回可用的物流公司。



4.2 运输单位选择页面

管理员选择物流公司后，会显示其可用车队，管理员根据订单要求（如是否冷链运输）来选择运输单位。



4.3 运输进度管理页面

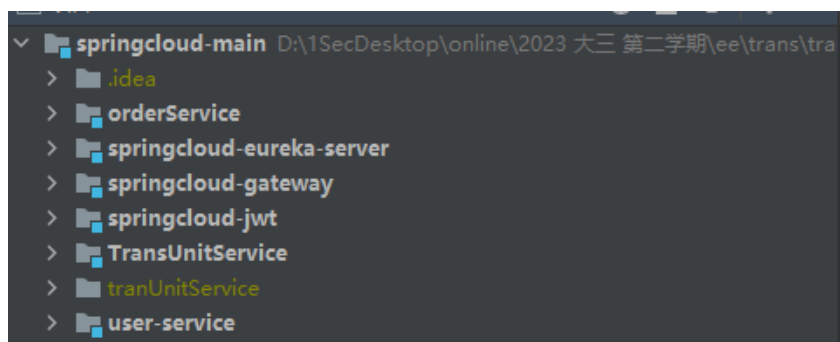
对于正在运输中的订单，可以点击更新状态来添加新的物流节点。点击确认到达后状态会变为待签收。



#4 考察功能点

5.1 重建微服务项目

上文已经提及，不再赘述，下图为重建后的项目结构。



5.2 Eureka微服务管理

System Status

Environment	test	Current time	2023-06-22T02:47:12 +0800
Data center	default	Uptime	00:04
		Lease expiration enabled	false
		Renews threshold	10
		Renews (last min)	8

DS Replicas

Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
GATEWAY-SERVICE	n/a (1)	(1)	UP (1) - localhost:gateway-service:8710
JWTSERVICE	n/a (1)	(1)	UP (1) - localhost:jwtService:8704
ORDERSERVICE	n/a (1)	(1)	UP (1) - localhost:orderservice:8707
TRANSUNITSERVICE	n/a (1)	(1)	UP (1) - localhost:transunitService:8708
USERSERVICE	n/a (1)	(1)	UP (1) - localhost:userservice:8705

General Info

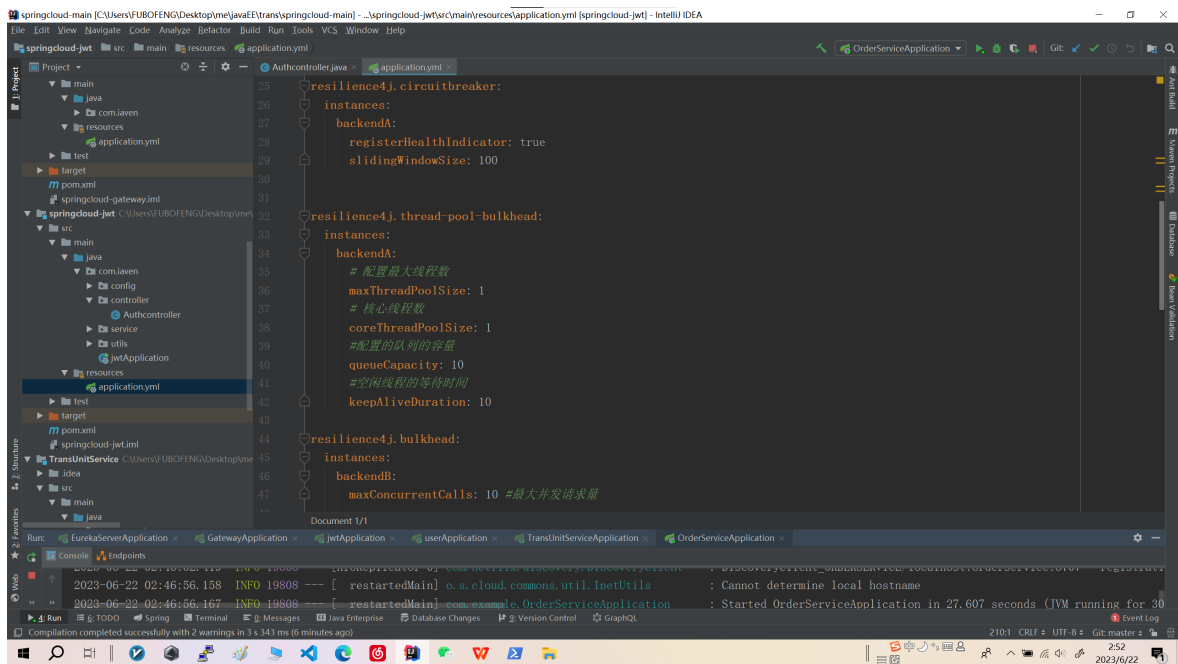
5.3 Resilience4j熔断

Resilience4j 是一种基于 JVM 的容错库，它提供了一系列的工具和函数式接口来帮助开发人员构建弹性和可靠的应用程序

Resilience4j 的核心原理是基于断路器模式、限流模式、重试模式等，通过在代码中添加不同的注解或者配置来实现相应的功能。例如，通过 `@CircuitBreaker` 注解来实现断路器模式，通过 `@RateLimiter` 注解来实现限流模式，通过 `@Retry` 注解来实现重试模式等。

Resilience4j 在本项目中的应用如下：

配置文件：



实现代码：

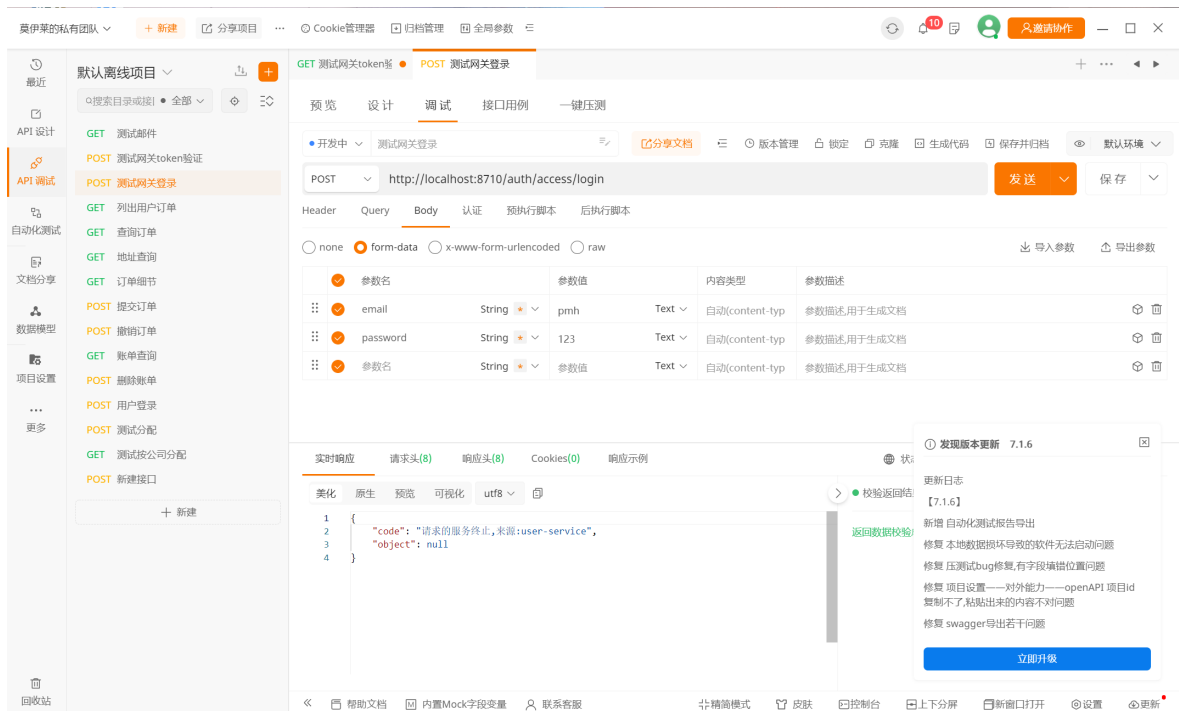
```
@PostMapping("/login")
@CircuitBreaker(name = "backendA", fallbackMethod = "loginHandler")
@Bulkhead(name = "backendA", fallbackMethod = "loginThreadHandler")
public ReturnObject<User> login(String email, String password) {...}

public ReturnObject<User> loginHandler(String email, String password, Throwable t) {
    System.out.println("请求的服务终止, 来源:user-service");
    return new ReturnObject<>("请求的服务终止, 来源:user-service", object: null);
}

public ReturnObject<User> loginThreadHandler(String email, String password, Throwable t) {
    System.out.println("线程池耗尽! :");
    return new ReturnObject<>("线程池耗尽!", object: null);
}
```

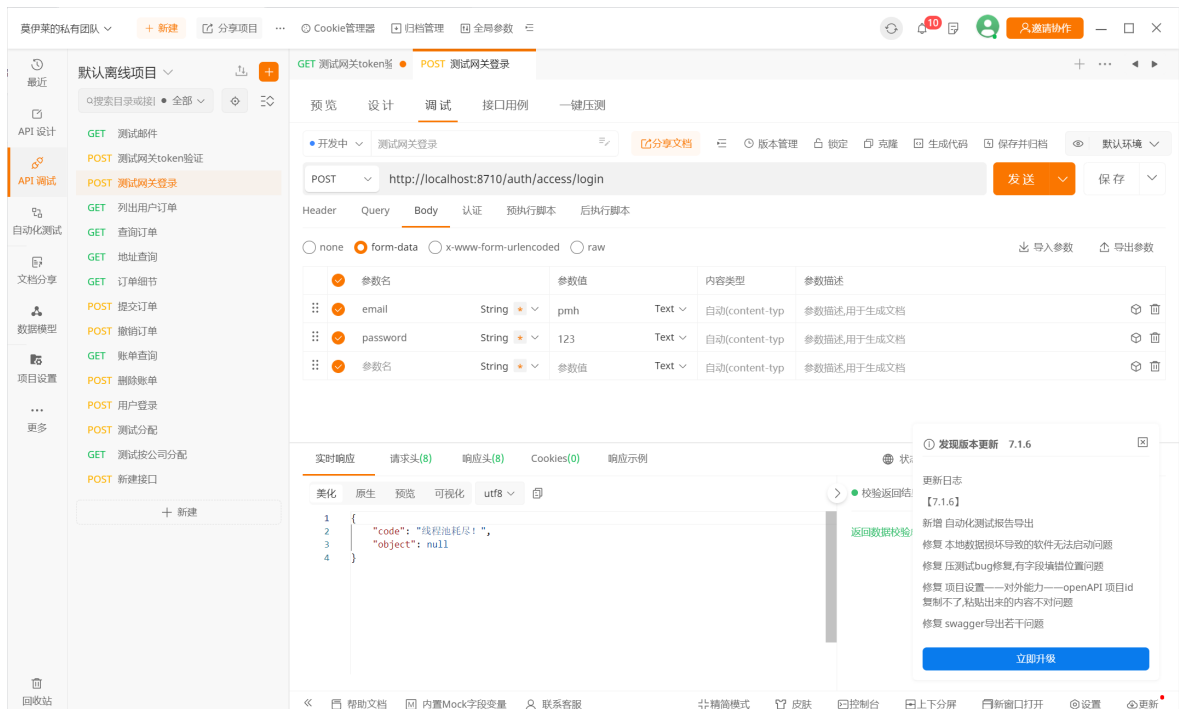
- **断路器模式：**Resilience4j 可以通过断路器模式防止故障的扩散，当某个服务或者接口出现故障时，断路器会快速切断对该服务或者接口的调用，并返回预设的失败结果。

测试结果：



- **限流模式：**Resilience4j 可以通过限流模式防止系统负载过高，当系统负载达到预设的阈值时，限流器会自动限制请求的流量，避免系统崩溃。

为了展示测试结果，这里暂时将线程池设置为1（即一秒只能有一个并发请求量）



5.4 集成Oauth2认证

OAuth 2.0 是一种授权框架，允许第三方应用程序以受限的方式访问资源。OAuth 2.0 实现了授权流程，使得用户可以授权一个应用程序代表自己访问另一个应用程序的资源，而无需提供自己的用户名和密码。OAuth 2.0 的授权流程通常涉及四个参与方：资源所有者（用户）、客户端（第三方应用程序）、授权服务器和资源服务器。授权服务器用于颁发令牌（token），而资源服务器用于根据令牌来控制对资源的访问权限。

OAuth 2.0 的授权流程包括以下步骤：

1. 客户端向资源所有者请求授权，并获得授权许可。
2. 客户端向授权服务器发送授权请求，并使用授权许可作为身份验证凭据。
3. 授权服务器验证客户端的身份，并向客户端颁发访问令牌。
4. 客户端使用访问令牌向资源服务器请求访问受保护的资源。
5. 资源服务器验证访问令牌的有效性，并根据访问令牌控制对资源的访问权限。

认证服务jwtservice的配置类：

```
@Configuration
@EnableResourceServer // 开启资源服务器功能
@EnableWebSecurity // 开启 web 访问安全
public class ResourceServerConfigurer extends ResourceServerConfigurerAdapter {

    private String sign_key = "shipSecretKey"; // jwt 签名密钥

    @Override
    public void configure(ResourceServerSecurityConfigurer resources) throws Exception {
        // 设置当前资源服务的资源 id 与认证 id 保持一致 可以不设置
        resources.resourceId("userservice");

        System.out.println("auth-----");
        // 定义 token 服务对象（token 校验就应该靠 token 服务对象）
        RemoteTokenServices remoteTokenServices = new RemoteTokenServices();
    }
}
```

```

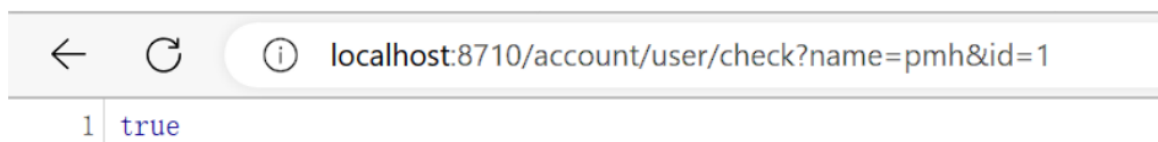
// 校验端点/接口设置
remoteTokenServices.setCheckTokenEndpointUrl("http://localhost:8704/access/jwt");
// 携带客户端 id 和客户端安全码
remoteTokenServices.setClientId("client_test");
remoteTokenServices.setClientSecret("abcxyz");
// 别忘了这一步
resources.tokenServices(remoteTokenServices);
}

@Override
public void configure(HttpSecurity http) throws Exception {
    http // 设置 session 的创建策略（根据需要创建即可）
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.IF_REQUIRED)
        .and()
        .authorizeRequests()
        .antMatchers("/user/testauth").authenticated() // autodeliver 为前缀的请求需要认证
        .anyRequest().permitAll(); // 其他请求不认证
}

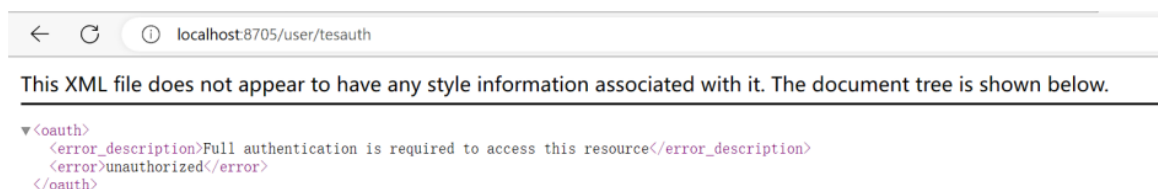
```

测试:

访问公共资源:



不带令牌访问受控制资源



5.5 GateWay网关

限流: 通过spring-boot-starter-data-redis-reactive包实现redis-limiter

配置文件:

```
- name: RequestRateLimiter #请求数限流 名字不能随便写
  args:
    key-resolver: "#{@ipKeyResolver}"
    redis-rate-limiter.replenishRate: 1
    redis-rate-limiter.burstCapacity: 1
```

路由转发:

```
cloud:
  gateway:
    routes:
      - id: service_1
        uri: lb://jwtService
        predicates:
          - Path=/auth/**
        filters:
          - StripPrefix=1
          - name: RequestRateLimiter #请求数限流 名字不能随便写
            args:
              key-resolver: "#{@ipKeyResolver}"
              redis-rate-limiter.replenishRate: 1
              redis-rate-limiter.burstCapacity: 1
```