



System Design Report

姓 名: 王姝昕

学 号: 20301118

姓 名: 李可佳

学 号: 20301105

目录

项目概述：	3
技术栈：	3
Eureka	3
Hystrix	3
Oauth2	4
Spring Cloud Gateway	4
Spring Cloud Config	4
Spring Cloud Sleuth	5
系统总体架构的设计与实现	5
服务拆分	15
统一网关	16
服务通信	17
认证授权：	18
异常处理与容错机制	20
配置管理	23
日志追踪	25
总结	26

项目概述：

Transport 项目是一个基于微服务架构的物流控制系统的网页应用。该项目采用前后端分离框架。前端采用 **Vue** 框架；后端采用 **Spring Boot** 框架，使用 **Spring Cloud** 等技术改造成微服务架构并进行集中配置管理。

本报告将详细介绍使用了一系列 **Spring Cloud** 技术包括 **Spring Cloud Netflix**（**Eureka**、**Hystrix**）、**Spring Cloud Security**（**Oauth2**）、**Spring Cloud Gateway**、**Spring Cloud Config** 和 **Spring Cloud Sleuth** 技术实现的系统设计。

技术栈：

Eureka：

Eureka 是 **Netflix** 开发的服务发现框架，本身是一个基于 **REST** 的服务，主要用于定位运行在 **AWS** 域中的中间层服务，以达到负载均衡和中间层服务故障转移的目的，其包含两个组件：**Eureka Server** 和 **Eureka Client**。

Eureka Server 提供服务注册服务，各个节点启动后，会在 **Eureka Server** 中进行注册，这样 **EurekaServer** 中的服务注册表中将会存储所有可用服务节点的信息，服务节点的信息可以在界面中直观的看到。

Eureka Client 是一个 **java** 客户端，用于简化与 **Eureka Server** 的交互，客户端同时也就是一个内置的、使用轮询(**round-robin**)负载算法的负载均衡器。

其优点包括：

1. 服务注册和发现：**Eureka** 使微服务能够在注册中心注册并动态发现其他服务。它简化了服务发现过程，有助于实现负载平衡和故障切换。
2. 高可用性和容错性：**Eureka** 采用高可用性架构设计，使其能够在网络分区或组件故障期间从故障中恢复并保持服务可用性。
3. 动态路由和负载平衡：**Eureka** 与 **Spring Cloud Gateway** 等其他组件集成良好，实现了基于服务发现的动态路由和负荷平衡。

Hystrix：

Netflix Hystrix 是 **SOA/微服务** 架构中提供服务隔离、熔断、降级机制的工具/框架，它也是断路器的一种实现，用于高微服务架构的可用性，是防止服务出现雪崩的利器。

其优点包括：

1. 容错和恢复能力: Hystrix 提供了一种容错机制, 用于处理分布式系统中的故障和延迟问题。它有助于防止级联故障, 并通过提供后备响应来实现优雅的降级。
2. 断路: Hystrix 监控微服务的运行状况, 并实现断路器, 以防止请求淹没不健康或过载的服务。它隔离并包含故障, 提高了整个系统的稳定性。
3. 实时监控和度量: Hystrix 提供有关服务性能和运行状况的见解和度量。它允许您监控和收集数据, 以分析微服务的行为和效率。

Oauth2:

OAuth2 是一种认证授权框架, 用于允许第三方应用经用户授权后访问对应的 http 服务或资源。它为用户提供了一种安全和标准化的方式, 允许他们访问自己的数据, 而无需与第三方应用程序共享凭据。其定义了四种角色, 分别是: 资源拥有者 (Resource Owner), 即真实用户; 客户端 (Client), 即第三方应用; 资源服务器 (Resource Server), 即管理资源的服务; 授权服务器 (Authorization Server), 即用于授权客户端访问的服务。

其优点包括:

1. 增强的安全性: 用户可以授予对其数据的有限访问权限, 而无需将其登录凭据暴露给第三方应用程序。
2. 以用户为中心的控制: 用户可以控制授予和撤销对其数据的访问, 从而提高隐私和数据安全性。
3. 简化了集成: OAuth 2.0 为身份验证和授权提供了标准化协议, 使开发人员更容易与不同的服务和应用程序集成。

Spring Cloud Gateway:

Spring Cloud Gateway 是 Spring Cloud 微服务生态下的网关组件, 它作为请求入口, 可以对微服务环境起到增强和保护的作用。

其优点包括:

1. API 网关和路由: Spring Cloud gateway 充当客户端请求的入口点, 并提供路由功能将请求转发到适当的微服务。它支持动态路由, 并可以与 Eureka 等服务发现机制集成以实现动态负载平衡。
2. 安全和身份验证: 网关为实现 JWT 验证、速率限制和请求过滤等安全和身份认证机制提供了一个集中的位置。
3. 请求转换和聚合: Gateway 允许您转换或聚合来自多个微服务的请求和响应, 从而在处理不同的客户端需求方面提供灵活性。

Spring Cloud Config:

Spring Cloud Config 为微服务架构中的微服务提供集中化的外部配置支持, 配置服务器

为各个不同微服务应用的所有环境提供了一个中心化的外部配置，其分为服务端和客户端两部分。

服务端也称为分布式配置中心，它是一个独立的微服务应用，用来连接配置服务器并为客户端提供获取配置信息，加密/解密信息等访问接口。

客户端则是通过指定的配置中心来管理应用资源，以及与业务相关的配置内容，并在启动的时候从配置中心获取和加载配置信息。

配置服务器默认采用 git 来存储配置信息，这样就有助于对环境配置进行版本管理，并且可以通过 git 客户端工具来方便的管理和访问配置内容。

其优点包括：

1. 集中式配置管理：Spring Cloud Config 允许您以集中式方式管理多个微服务的配置。它提供了一个服务器，用于存储和服务每个微服务的配置属性。
2. 动态配置更新：使用 Spring Cloud Config，您可以在不重新启动微服务的情况下更新微服务的配置，从而实现动态配置更改。
3. 配置版本控制和历史记录：它为配置提供版本控制和跟踪历史记录，使管理和跟踪更改变得更容易。

Spring Cloud Sleuth：

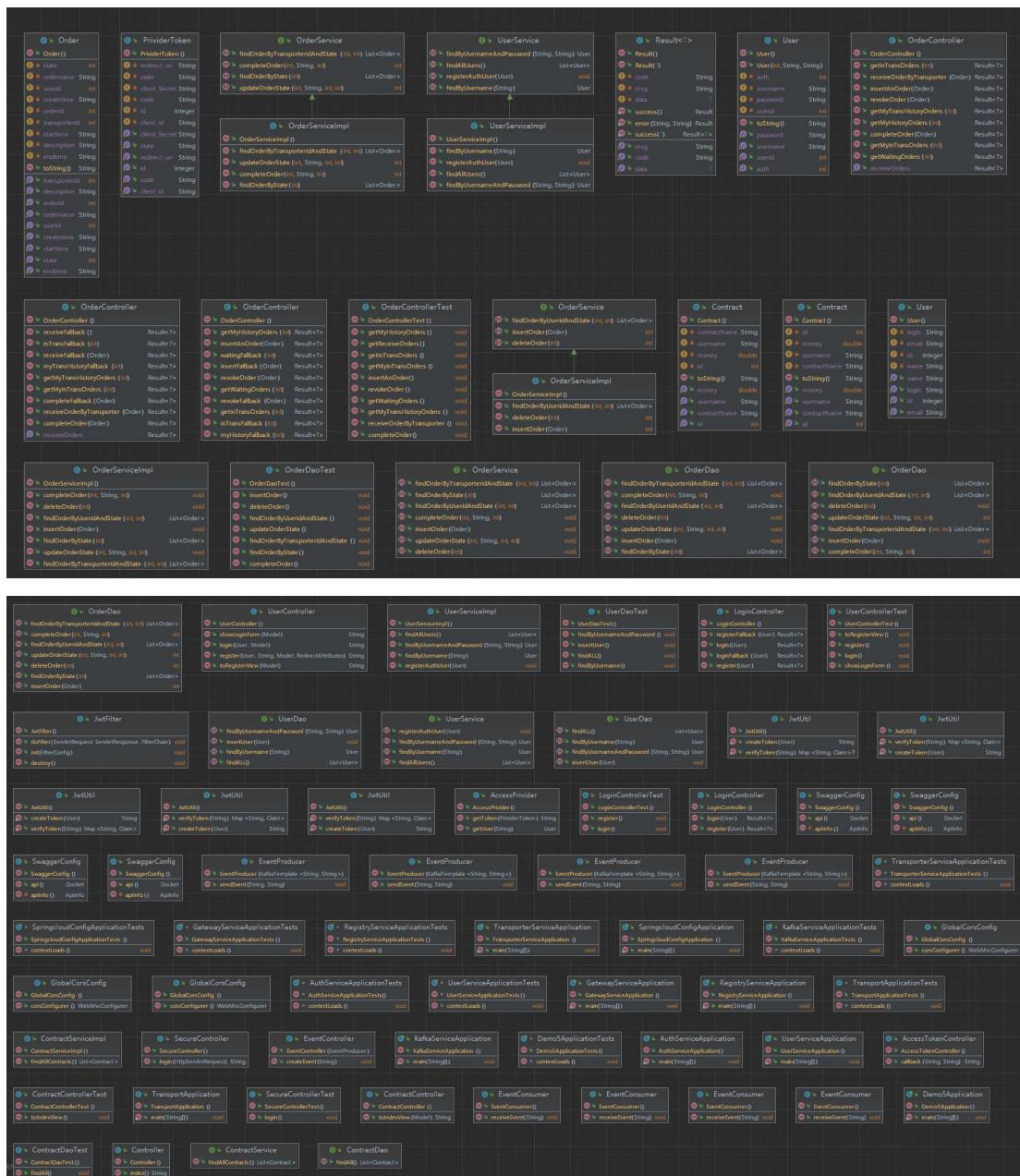
Spring Cloud Sleuth 是 Spring 生态系统中微服务架构的分布式跟踪框架，它会自动向系统的日志和服务间通信中添加一些跟踪/元数据（通过请求标头），跟踪特定的请求日志。

其优点包括：

1. 分布式跟踪：Spring Cloud Sleuth 有助于跟踪和监控请求，因为它们在分布式系统中跨多个微服务传播。它为每个请求分配并传播唯一的标识符（如 Trace ID、Span ID），从而实现端到端的可见性。
2. 性能优化：Sleuth 提供了对单个服务及其交互性能的深入了解。它有助于识别和优化分布式系统中的性能瓶颈。
3. 故障排除和调试：使用 Sleuth，您可以轻松地跟踪和分析请求流，从而更容易地在分布式环境中排除问题和调试微服务。

系统总体架构的设计与实现：

项目类图：



本物流系统采用基于 Spring Cloud 微服务的架构，将项目拆分为 6 个独立的微服务模块，每个微服务模块提供不同的服务和功能。以下是各微服务模块的简介：

springcloud-config 配置管理服务模块：

负责管理系统的配置信息，为其他服务提供配置信息的获取。

registryService 注册中心服务模块：

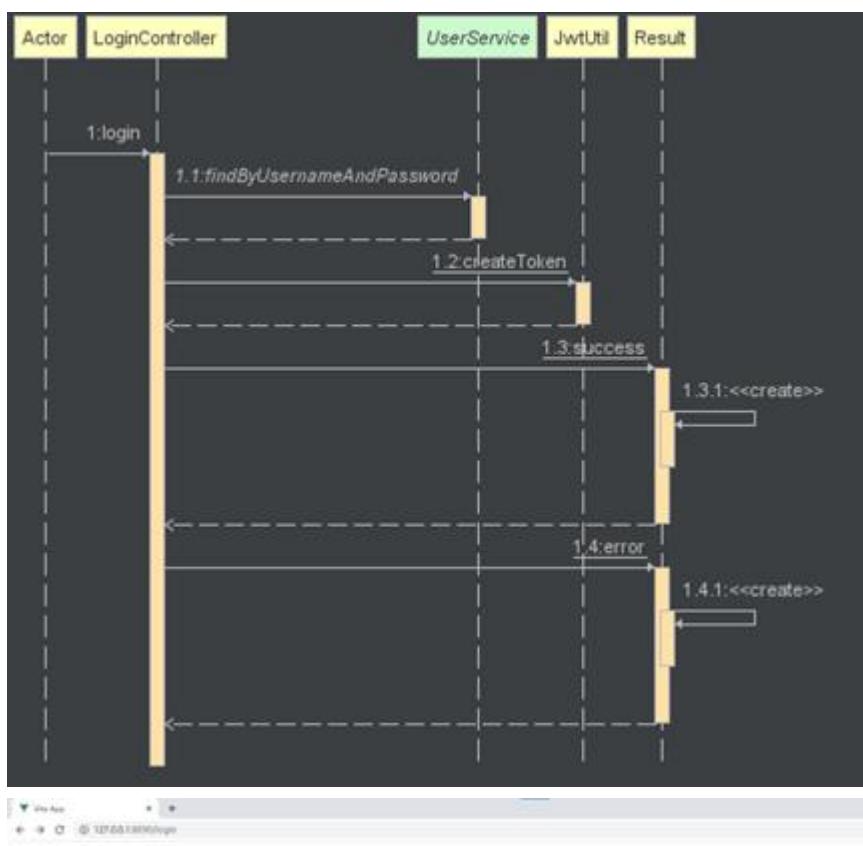
用于服务的注册与发现，服务通过注册中心进行服务间的通信。

gatewayService 网关服务模块: 负责请求的路由和过滤, 统一各服务模块的访问接口, 是系统对外的唯一入口。

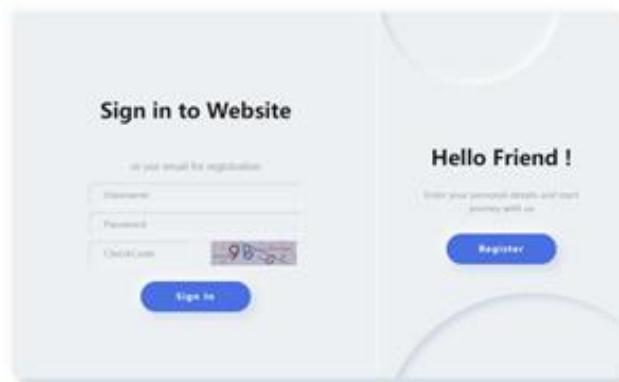
auth service 授权服务模块:

负责处理用户和运输者的登录注册相关功能，提供用户认证、权限管理、JWT token 生成、登录、注册、登出等相关认证功能。

用户登录：

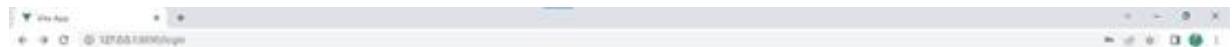
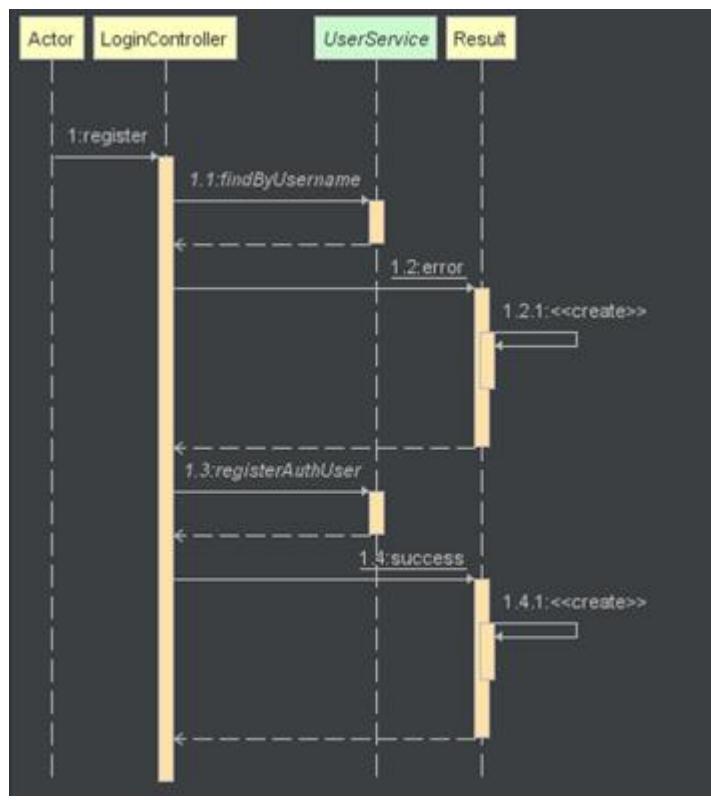


127.0.0.1:8080/login



127.0.0.1:8080/login

用户注册：



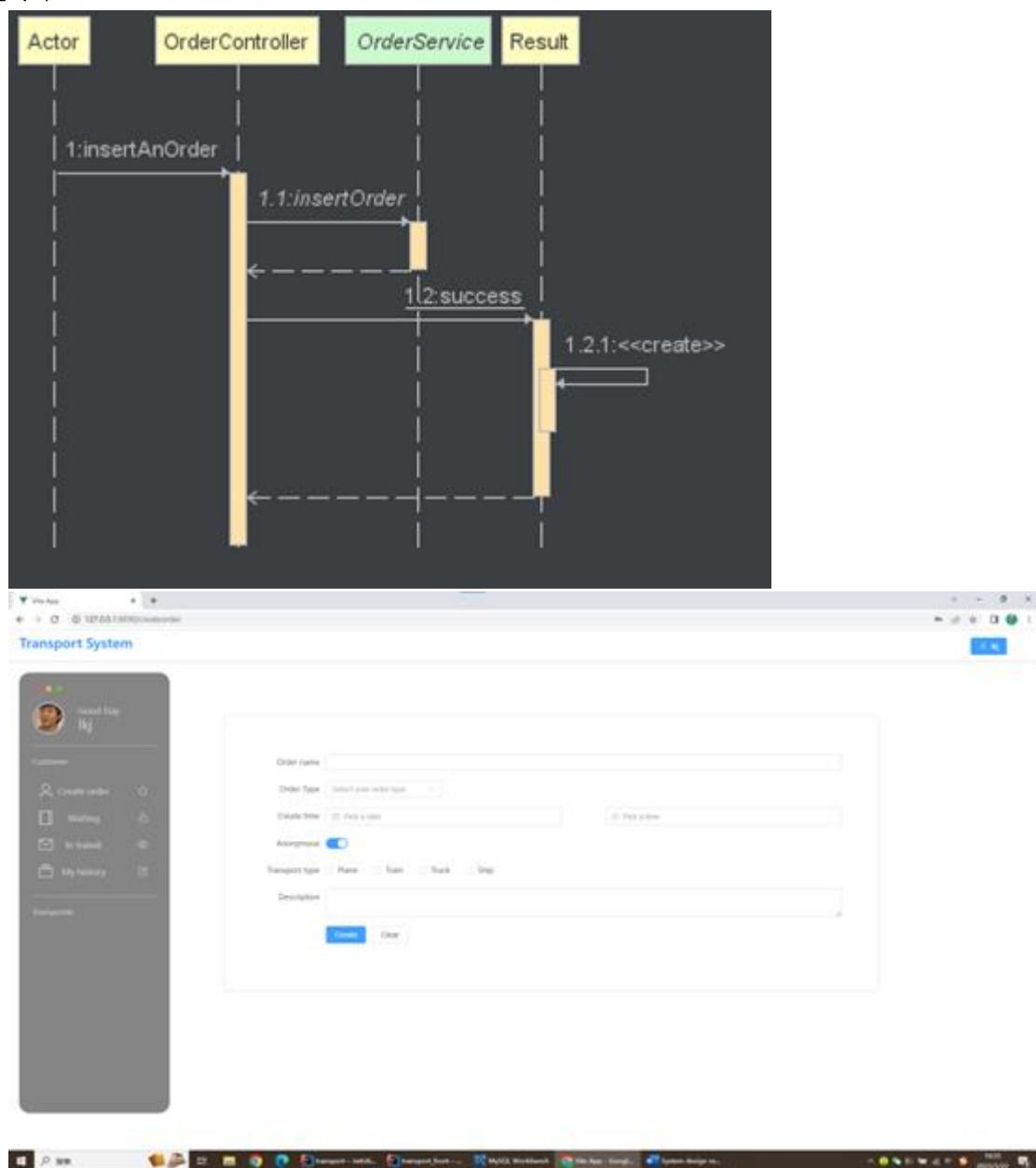
A detailed view of the 'Create Account' form. It includes a 'Welcome Back !' section with a 'Sign In' button, a 'Create Account' title, and a form with fields for Name, Email, Password, and Confirm Password. A 'Register' button is at the bottom right. The background shows a light blue gradient with white curves.



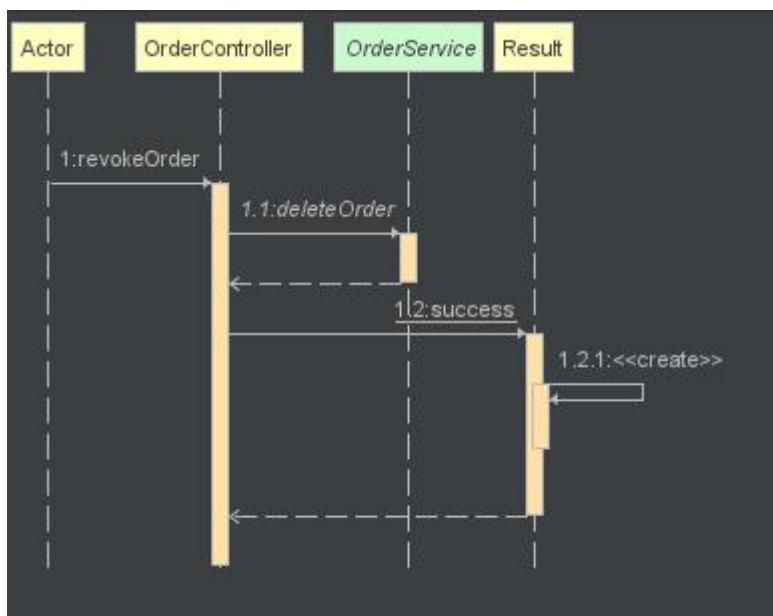
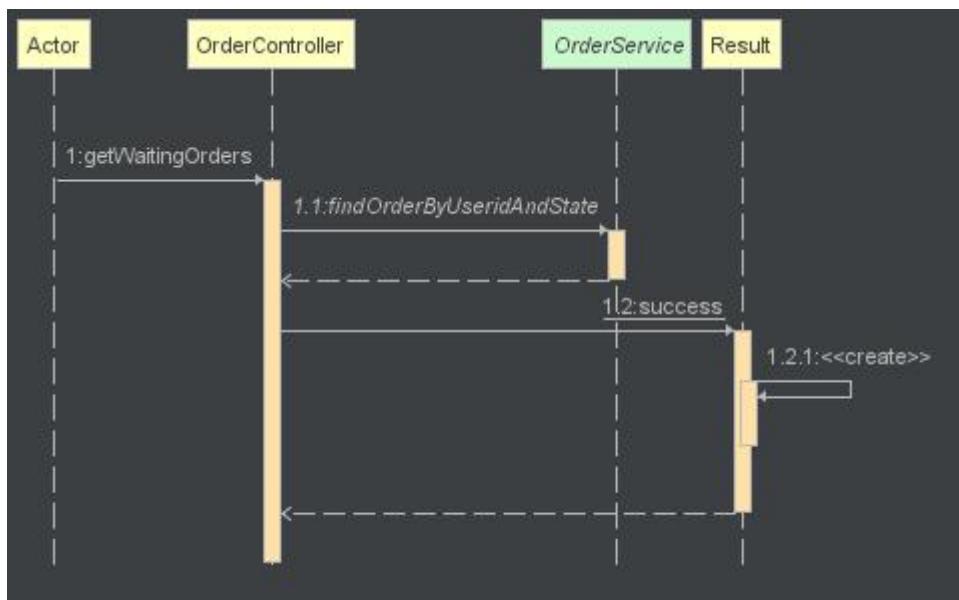
user_service 客户服务模块:

负责处理用户相关的业务逻辑，包括发起订单、撤回订单、查看等待中订单、查看运输中订单、查看已完成订单等用户业务功能。

创建订单：

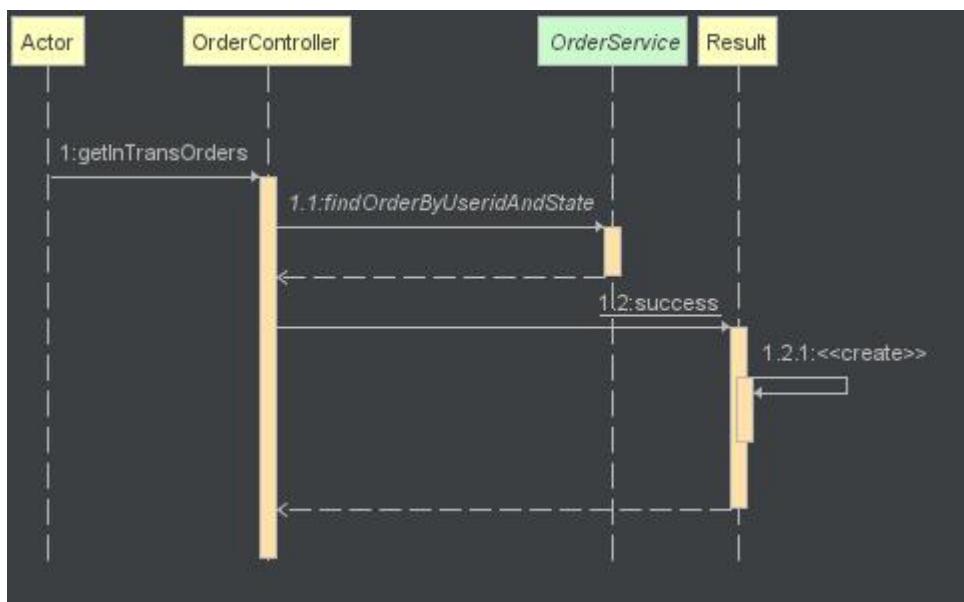


查看等待中的订单：



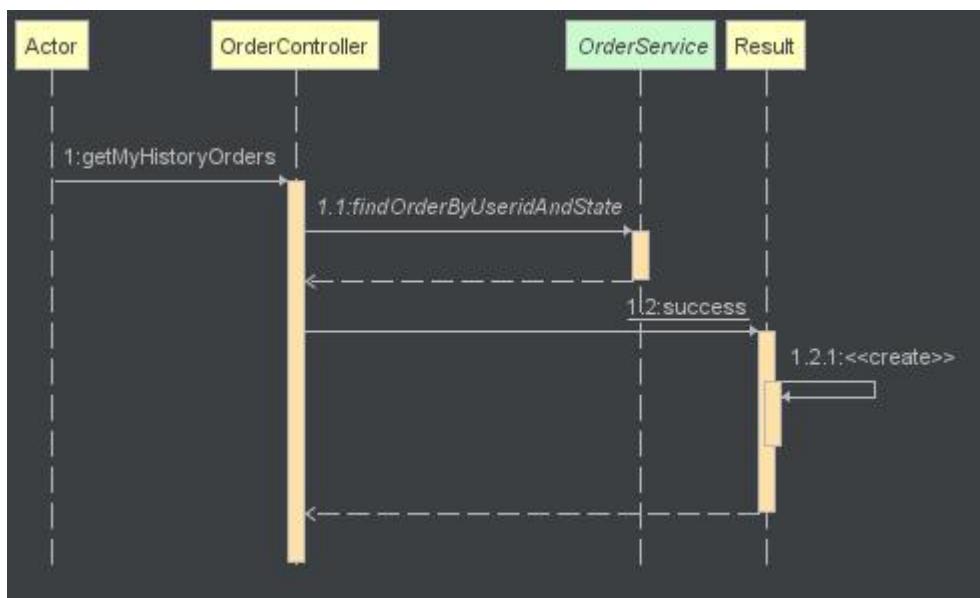
Name	Description	Create Time	User Id	Status	Operations
222	deadlined	2023-05-10 19:55:52	1	Waiting be transported	Revoke
444	Ship project	2023-05-10 19:56:10	1	Waiting be transported	Revoke

查看运输中的订单：



Name	Description	Create time	Userid	TransporterId	Take order time	State
second	nothing	2023-05-Mo 12:42:42	1	2	2023-05-22 14:51:12	In transit

查看已完成订单：

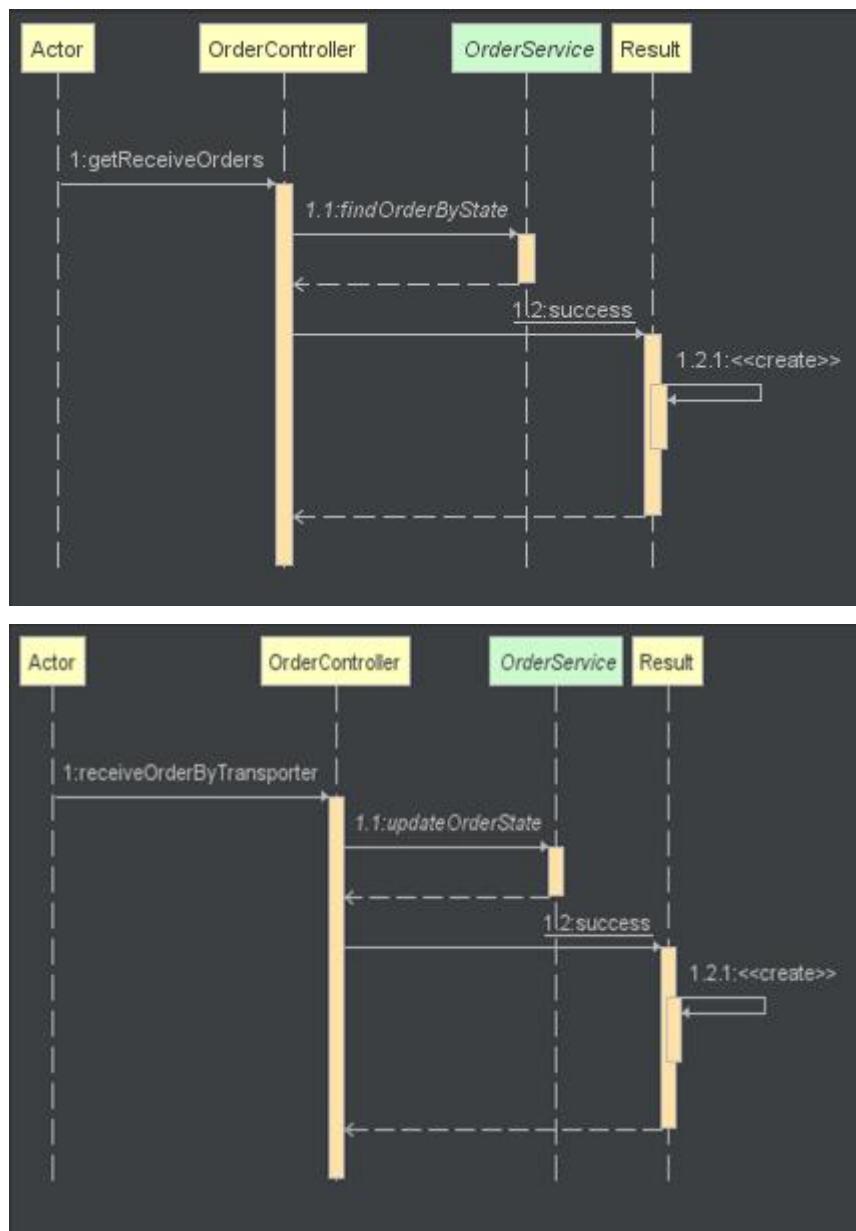


Name	Description	Create time	Userid	TransporterId	Take order time	Completed time	State
firstorder	nothing	2023-05-Mo 12:42:42	1	2	2023-05-Mo 12:42:42	2023-05-22 15:01:44	Completed
third	nothing	2023-05-Mo 12:42:42	1	2	2023-05-Mo 12:42:42	2023-05-Mo 12:42:42	Completed
four	nothing	2023-05-Mo 12:42:42	1	2	2023-05-Mo 12:42:42	2023-05-Mo 12:42:42	Completed

transporter_service 运输者服务模块:

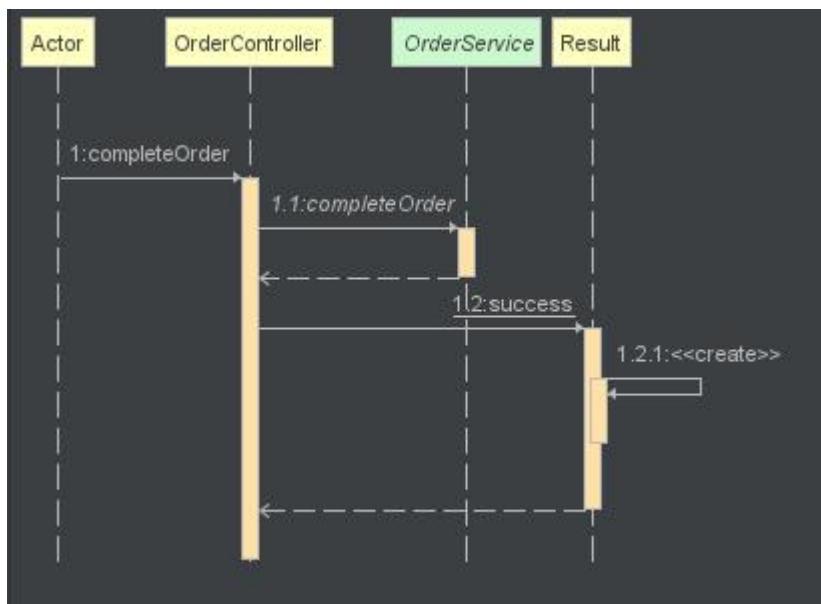
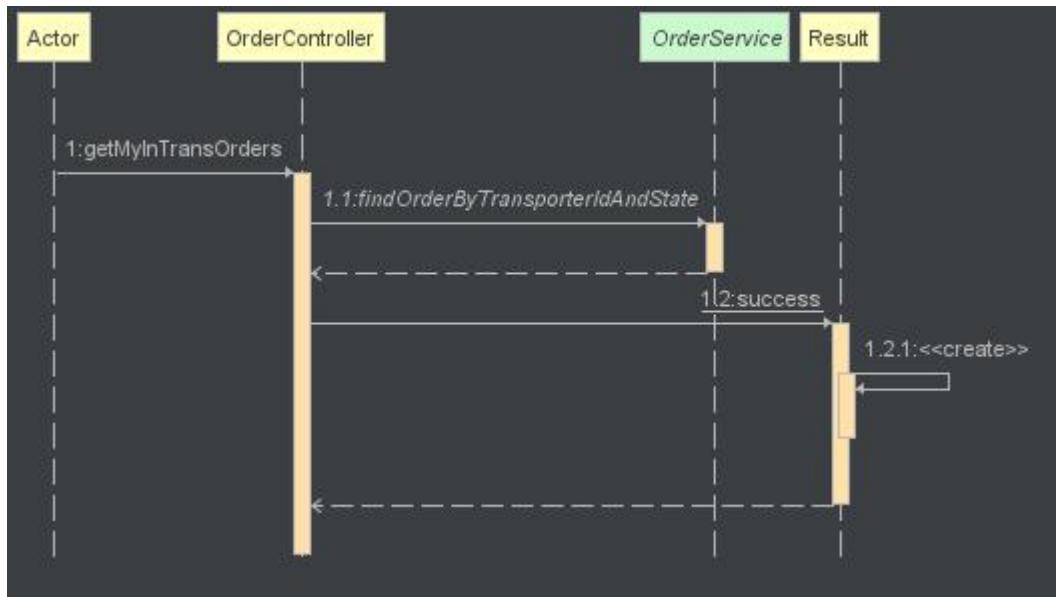
负责处理运输者相关的业务逻辑，包括接收订单、查看己方正在运输的的订单，完成订单，查看运输历史等运输者业务功能。

承接订单：



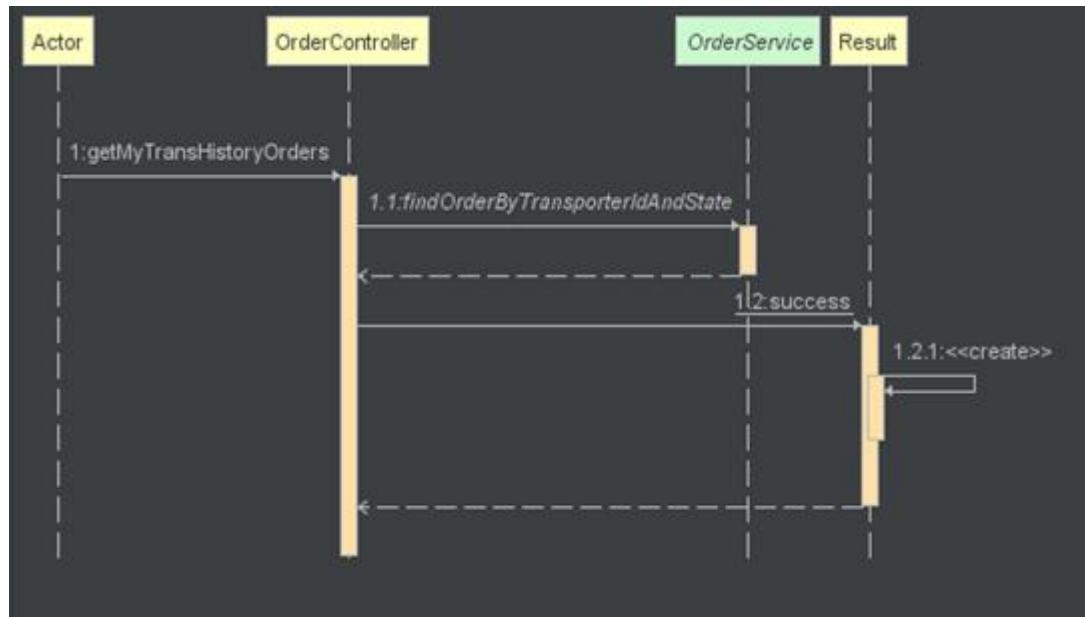
Name	Description	Create Time	User Id	Status	Operations
222	desined	2023-05-Mo 19:55:52	1	Waiting be transported	Receive
444	Ship project	2023-05-Mo 19:56:10	1	Waiting be transported	Receive

查看运输中的订单：



Name	Description	Create time	Userid	TransporterId	Take order time	Status	Operations
second	nothing	2023-05-22 12:42:42	1	2	2023-05-22 14:51:12	In transit	Complete

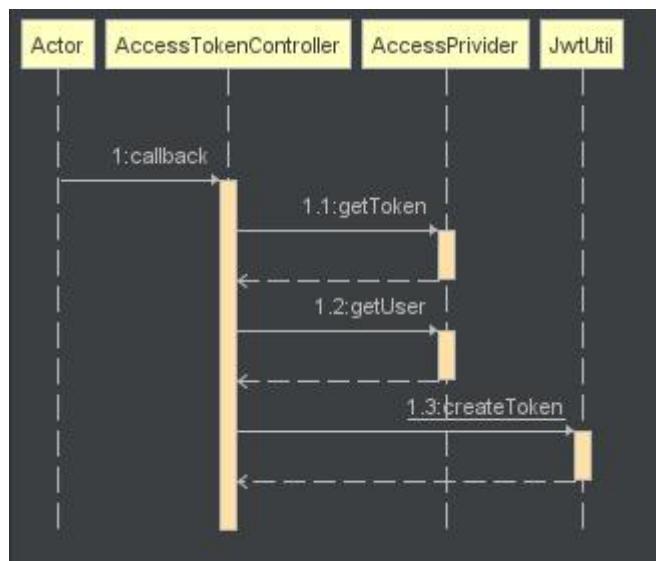
查看历史运输记录：



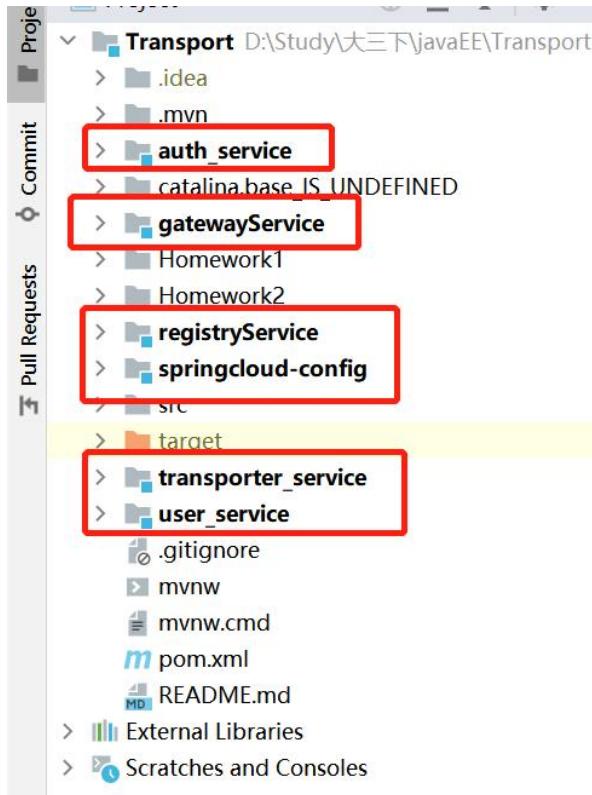
Name	Description	Create time	Userid	TransporterId	Take order time	Completed time	State
firstorder	nothing	2023-05-Mo 12:42:42	1	2	2023-05-Mo 12:42:42	2023-05-22 15:01:44	Completed
third	nothing	2023-05-Mo 12:42:42	1	2	2023-05-Mo 12:42:42	2023-05-Mo 12:42:42	Completed
four	nothing	2023-05-Mo 12:42:42	1	2	2023-05-Mo 12:42:42	2023-05-Mo 12:42:42	Completed

oauth2_service 第三方登录模块：

负责第三方登录功能，本项目使用 gitee 结合 oauth2 协议标准可实现第三方登录。
回调函数时序图：



项目目录:



服务拆分

本物流系统拆分为 7 个独立的微服务，每个服务具有独立的职责。服务之间的通信通过 HTTP 协议进行。

1. gatewayService 通过网关服务进行请求路由和过滤，实现请求的负载均衡和安全性控制。
2. auth_service 向用户管理服务发送请求进行用户注册、登录等操作，通过 Eureka 进行服务发现。
3. user_service 通过网关服务向物流服务发起创建订单、查询订单等请求，通过 Eureka 进行服务发现。
4. transporter_service 通过 Eureka 进行服务注册，接收来自网关服务和客户端的请求，处理物流相关的业务逻辑。
5. registryService 提供注册中心服务
6. springcloud-config 开启 spring cloud config server 服务
7. oauth_service 提供第三方（gitee）登录功能

功能模块

Eureka:

完成 Eureka 注册中心的搭建，各微服务成功注册到 Eureka 上，并实现服务发现功能：

The screenshot shows the Spring Eureka dashboard at localhost:8761. The top navigation bar includes links for 'HOME' and 'LAST 1000 SINCE STARTUP'. The main content area is titled 'System Status' and displays various system metrics. Below this, a note states: 'THE SELF PRESERVATION MODE IS TURNED OFF. THIS MAY NOT PROTECT INSTANCE EXPIRY IN CASE OF NETWORK/OTHER PROBLEMS.' The 'DS Replicas' section lists four services: AUTH-SERVICE, GATEWAY-SERVICE, TRANSPORTER-SERVICE, and USER-SERVICE, each with its status, IP, and port number. A red box highlights the URL in the browser address bar, and another red box highlights the service details in the 'Instances currently registered with Eureka' table. A red arrow points from the text 'Eureka的服务注册成功，各微服务名称和端口号如下：' to the highlighted table row for GATEWAY-SERVICE.

Application	AMIs	Availability Zones	Status
AUTH-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-B79J07/auth-service:8082
GATEWAY-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-B79J07/gateway-service:8888
TRANSPORTER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-B79J07/transporter-service:8083
USER-SERVICE	n/a (1)	(1)	UP (1) - DESKTOP-B79J07/user-service:8081

统一网关

网关配置信息：

```

1  server:
2    port: 8888
3
4  spring:
5    profiles: dev
6    application:
7      name: gateway-service
8      main:
9        allow-bean-definition-overriding: true
10   cloud:
11     gateway:
12
13   routes:
14     - id: user_service
15       uri: lb://user-service
16       predicates:
17         - Path=/user-service/**
18       filters:
19         - StripPrefix=1
20
21     - id: auth_service
22       uri: lb://auth-service
23       predicates:
24         - Path=/auth-service/**
25       filters:
26         - StripPrefix=1
27
28     - id: transporter_service
29       uri: lb://transporter-service
30       predicates:
31         - Path=/transporter-service/**
32       filters:
33         - StripPrefix=1
34
35   discovery:
36     locator:
37       enabled: true
38       lower-case-service-id: true
39
40   eureka:
41     client:
42       fetch-registry: true
43       register-with-eureka: true
44       service-url:

```

配置统一对外暴露的API端口号8888

设置各微服务的路由转发

服务通信

成功通过统一网关 8888 端口号对 auth-service 微服务的 login 接口进行访问，并返回相应的 token 值：

The screenshot shows a Postman interface with the following details:

- Request URL: `http://127.0.0.1:8888/auth-service/login`
- Method: POST
- Body content:

```
1 ... "username": "lkj",
2 ... "password": "123"
```
- Response status: 200 OK
- Response body (Pretty):

```
1 {
2   "code": "200",
3   "msg": "成功",
4   "data": {
5     "auth": 0,
6     "id": 1,
7     "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
eyJwYXNzd29yZC16IjEyMyIsIm1kIjoxLCJ1c2VyTmFtZS16ImxraiIsImV4cCI6MTY5MDU5NTQ0NSviaWF0IjoxNjg2OTk1NDQ1fQ.
mRMd44mf_aWTqM1HkiTSFKo2x06azSMfMyBUD14Xpkg"
```

成功通过统一网关 8888 端口号对 user-service 微服务的 waiting 接口进行访问：

The screenshot shows a Postman interface with the following details:

- Request URL: `http://127.0.0.1:8888/user-service/secure/waiting/1`
- Method: GET
- Headers:

KEY	VALUE	DESCRIPTION	...	Bulk Edit	Presets
authorization	eyJhbGciOiJIUzI1NilsInR5cCI6IkpXVCJ9 eyJwYXNzd29...				
Key	Value	Description			
- Response status: 200 OK
- Response body (Pretty):

```
1 {
2   "code": "200",
3   "msg": "成功",
4   "data": [
5     {
6       "orderid": 2,
7       "ordername": "second",
8       "description": "nothing",
9       "createtime": "2023-05-22 12:42:42",
10      "starttime": "2023-05-22 14:51:12",
11      "endtime": "2023-06-14 22:47:18",
12      "userid": 1,
13      "transporterid": 2,
14      "state": 0
15    }
16  ]
```

成功通过统一网关 8888 端口号对 transporter-service 微服务的 receive 接口进行访问：

The screenshot shows the Postman interface with the following details:

- Request URL: `http://127.0.0.1:8888/transporter-service/secure/receive`
- Method: GET
- Headers tab selected, showing:
 - `authorization`: eyJhbGciOiJIUzI1NlslnR5cCl6IkpxVCJ9.eyJwYXNzd29...
- Body tab selected, showing the response JSON:

```
1 "code": "200",
2 "msg": "成功",
3 "data": [
4     {
5         "orderid": 2,
6         "ordername": "second",
7         "description": "nothing",
8         "createtime": "2023-05-Mo 12:42:42",
9         "starttime": "2023-05-22 14:51:12",
10        "endtime": "2023-06-14 22:47:18",
11        "userid": 1,
12        "transporterid": 2,
13        "state": 0
14    }
15 ]
16 ]
```

认证授权

Oauth2 实现 Gitee 登录：

The screenshot shows a browser window with the following details:

- Address bar: `127.0.0.1:8890/login`
- Page title: 前端登录页面展示
- Form fields: Username, Password, CheckCode
- Buttons: Sign In, Gitee login, Register
- Text: Sign in to Website, Hello Friend !, Enter your personal details and start Journey with us.

A red box highlights the "Gitee login" button, and a red arrow points to it from the text "可使用Gitee方式进行登录".

点击 Gitee login 即可跳转到 gitee 登陆网站:



gitee 登陆成功后返回用户信息并生成 JWT token 并跳转到登录成功页面:

```
no usages new *
@GetMapping("/callback")
public String callback(@RequestParam(name = "code") String code,
                      @RequestParam(name = "state") String state) {

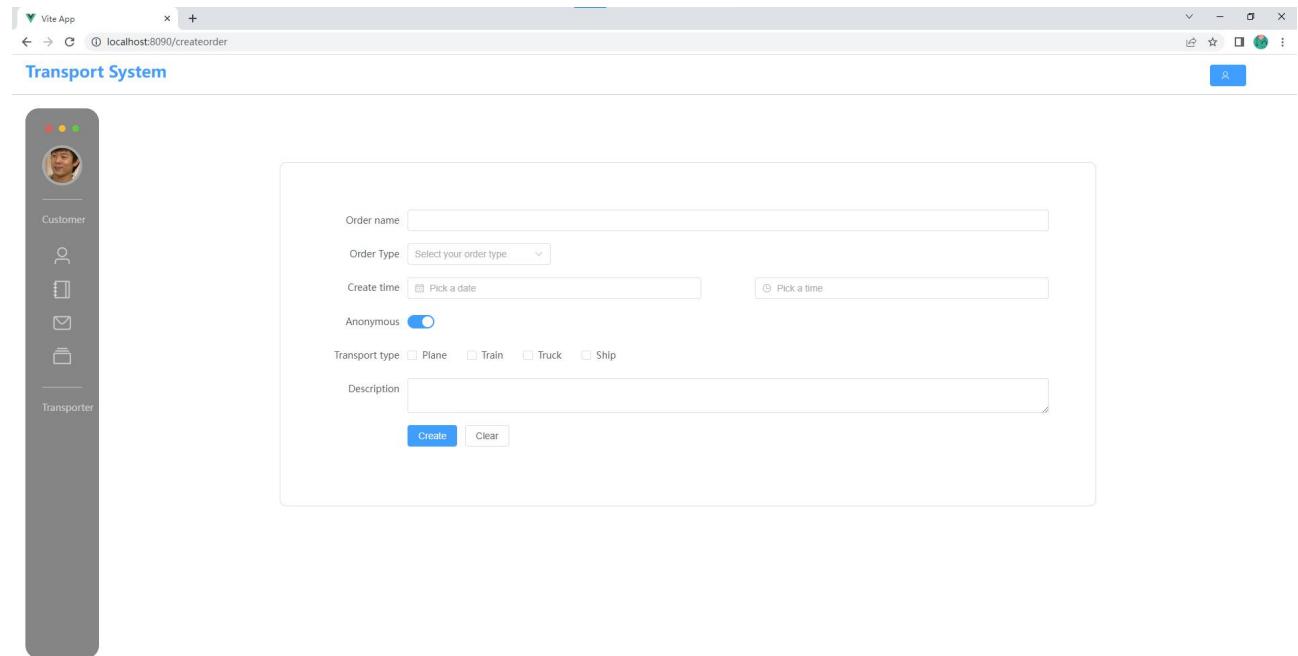
    providerToken.setClient_id("8e70556e33da0ea034e85276607e26b93c5fc6dd893061c9d92e44800051cc51");
    providerToken.setCode(code);
    providerToken.setRedirect_uri("http://localhost:8080/callback");
    providerToken.setState(state);
    providerToken.setClient_Secret("b139eb710179a336a7927287ca0166ec96d617ca65f06a30fb5e9f99286aa85d");
    String Token = accessProvider.getToken(providerToken);
    user = accessProvider.getUser(Token);
    user1.setLogin(user.getLogin());
    user1.setId(user.getId());
    user1.setEmail(user.getEmail());
    String JWTtoken = JwtUtil.createToken(user1);
    System.out.println("JWT token: " + JWTtoken);
```

打印生成的 JWT token:

```
2023-06-18 05:30:48.560 INFO 16444 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 6 ms
token: dc13e4efa4004c5f2e0f2653e741b0e8
{"access_token":"dc13e4efa4004c5f2e0f2653e741b0e8","token_type":"bearer","expires_in":86400,"refresh_token":"fb11883baee120efd9ff1dadb9e22273c6f37d9fa0028e70f1389d23032a4450","scope":"user_info","creat
lkjjjj
11060120
JWT token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCQ9eyJwYXNzd29yZC16ImxrampqaiIsImlkIjoxMTA2MDEyMCwiZXhwIjoxNjkwNjM3NDQ5LCJpYXQiOjE20DcwMzc0NDl9.vustT23Xasih22j_Pcj33FyGBeisa92qvahtWQCW-E
```

```
2023-06-18 05:30:48.560 INFO 16444 ---
token: dc13e4efa4004c5f2e0f2653e741b0e8
{"access_token":"dc13e4efa4004c5f2e0f2653e741b0e8","token_type":"bearer","expires_in":86400,"refresh_token":"fb11883baee120efd9ff1dadb9e22273c6f37d9fa0028e70f1389d23032a4450","scope":"user_info","creat
lkjjjj
11060120
JWT token: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCQ9eyJwYXNzd29yZC16ImxrampqaiIsImlkIjoxMTA2MDEyMCwiZXhwIjoxNjkwNjM3NDQ5LCJpYXQiOjE20DcwMzc0NDl9.vustT23Xasih22j_Pcj33FyGBeisa92qvahtWQCW-E
```

此页面生成时通过 `mount()` 方法请求拿到 JWT token，并进行后续的数据请求和处理操作：



异常处理与容错机制

为了保证系统的稳定性和可靠性，引入 `Hystrix` 来实现服务的容错机制。`Hystrix` 可以在服务出现故障或超时的情况下，提供降级策略或者熔断处理。

在各微服务中，使用 `Hystrix` 实现对数据库操作与远程服务调用的超时与容错处理，通过熔断机制防止服务雪崩，避免单个服务的故障影响整个系统。

微服务模块的 `controller` 层接口上添加服务熔断注解 `@HystrixCommand` 和回退方法 `fallbackMethod` 实现服务熔断，以 `transporter_service` 为例如下图所示：

实现Hystrix的服务熔断

```

19   @Slf4j
20   @RestController
21   @RequestMapping(@RequestMapping("/secure"))
22   public class OrderController {
23       @Autowired
24       private OrderService orderService;
25
26       @HystrixCommand(fallbackMethod = "receiveFallback") //失败了就会调用下面的这个备选方案
27       @GetMapping("receive")
28       public Result<?> getReceiveOrders(){
29           List<Order> orders = orderService.findOrderByState(0);
30           if(orders.isEmpty()){
31               throw new RuntimeException("orders throw RuntimeException");
32           }
33           return Result.success(orders);
34       }
35
36       //备选方案
37       public Result<?> receiveFallback(){
38           return Result.error(code: "204", msg: "No receive orders");
39       }
40
41       @HystrixCommand(fallbackMethod = "inTransFallback")
42       @GetMapping("myintrans/{transporterid}")
43       public Result<?> getMyInTransOrders(@PathVariable("transporterid") int transporterid){
44           List<Order> orders = orderService.findOrderByTransporterIdAndState(transporterid, state: 1);
45           if(orders.isEmpty()){
46               throw new RuntimeException("orders throw RuntimeException");
47           }
48           return Result.success(orders);
49       }
    
```

启动类上添加@EnableHystrix注解开启服务熔断：

```

3   import org.springframework.boot.SpringApplication;
4   import org.springframework.boot.autoconfigure.SpringBootApplication;
5   import org.springframework.boot.web.servlet.ServletComponentsScan;
6   import org.springframework.cloud.client.discovery.EnableDiscoveryClient;
7   import org.springframework.cloud.netflix.hystrix.EnableHystrix;
8   import springfox.documentation.swagger2.annotations.EnableSwagger2;
9
10  @EnableDiscoveryClient
11  @EnableHystrix //开启服务熔断
12  @SpringBootApplication
13  @ServletComponentsScan(basePackages = "com.example.transporter_service.authorization")
14  public class TransporterServiceApplication {
15      public static void main(String[] args) { SpringApplication.run(TransporterServiceApplication.class, args); }
16  }
    
```

测试断路器的服务熔断功能：

当数据库中存在资源时，可成功访问该资源，不需要开启服务熔断：

The screenshot shows a POST request to `http://127.0.0.1:8888/transporter-service/secure/send`. The response status is 200 OK, time is 573 ms, size is 456 B, and the response body is:

```
1  {
2      "code": "200",
3      "msg": "成功",
4      "data": [
5          {
6              "orderid": 2,
7              "ordername": "second",
8              "description": "nothing",
9              "createtime": "2023-05-22 12:42:42",
10             "starttime": "2023-05-22 14:51:12",
11             "endtime": "2023-06-14 22:47:18",
12             "userid": 1,
13             "transporterid": 2,
14             "state": 0
15         }
16     ]
17 }
```

当数据库中不存在该资源时，不能进行访问并开启服务熔断告知用户：

The screenshot shows the Postman interface with the following details:

- URL:** http://127.0.0.1:8888/transporter-service/secure/receive
- Method:** GET
- Headers:** authorization (value: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJwYXNzd29...)
- Body:** JSON response:

```
1 "code": "204",
2 "msg": "No receive orders",
3 "data": null
```
- Status:** 200 OK
- Time:** 22 ms
- Size:** 271 B

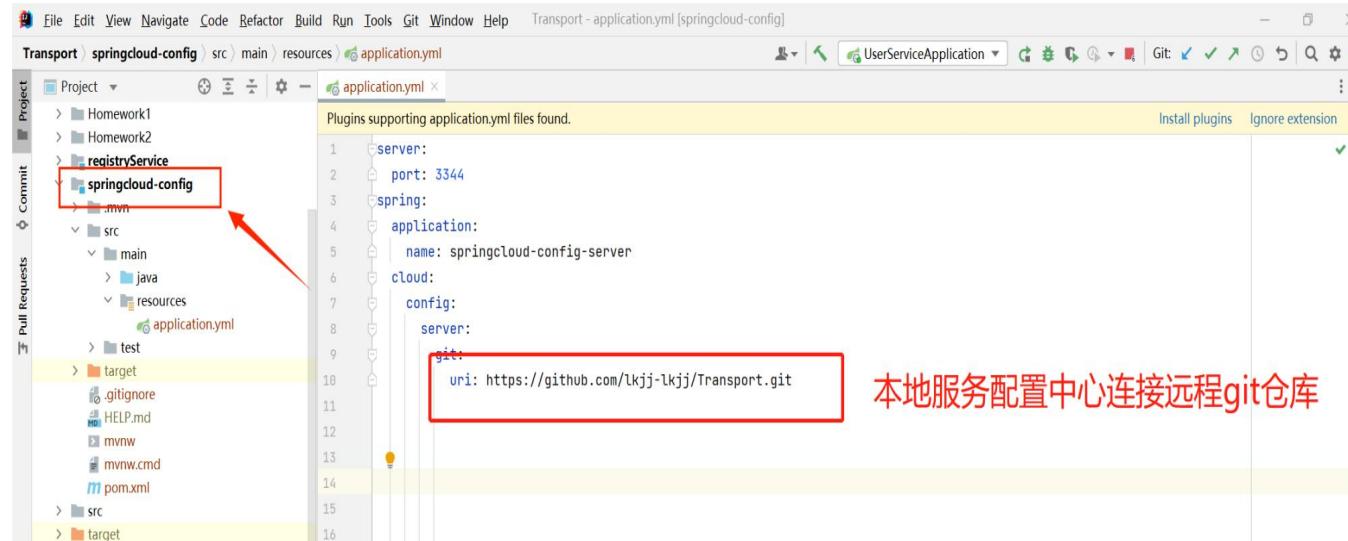
A large red box highlights the URL field and the JSON response body.

当数据库不存在该资源时，断路器实现熔断处理

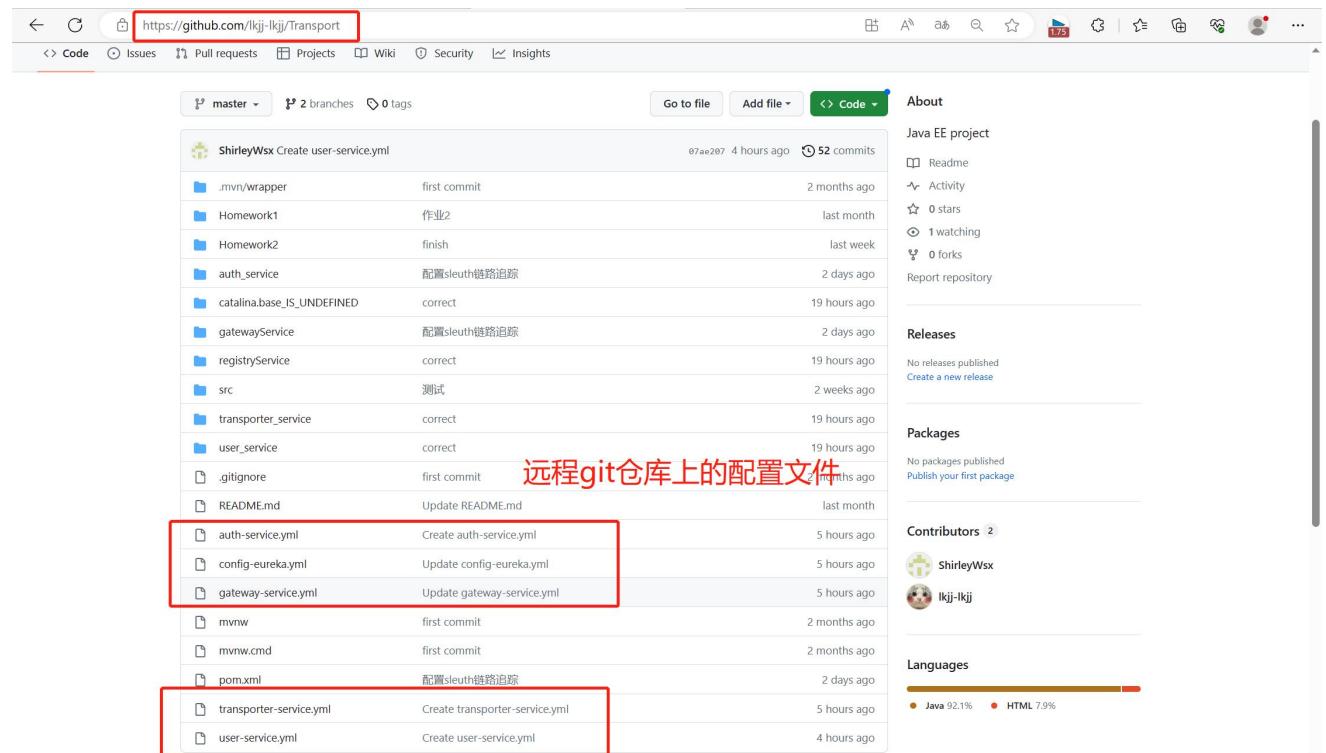
配置管理

为了方便管理系统的配置信息，引入 Spring Cloud Config。Spring Cloud Config 可以集中管理各个微服务的配置文件，并提供 REST 接口供其他服务获取配置信息。

springcloud-config 本地服务配置中心：



远程 GitHub 仓库上配置文件，分别有 auth-service.yml、config-eureka.yml、gateway-service.yml、transporter-service.yml、user-service.yml：



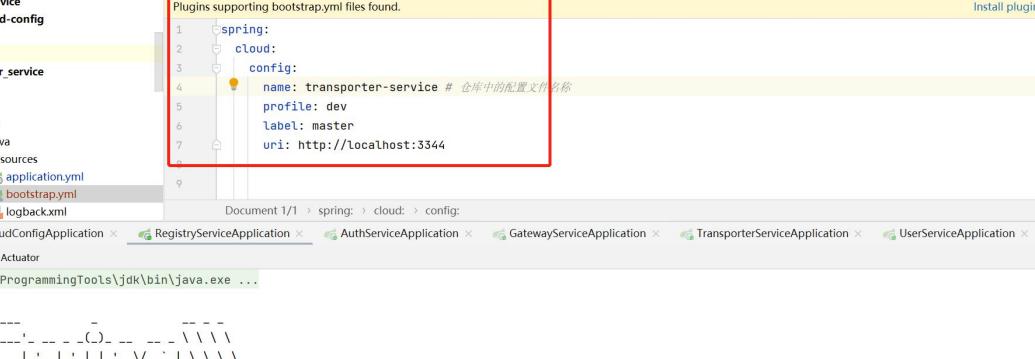
远程 GitHub 仓库上配置文件的内容，以 config-eureka.yml 为例如下图所示：



The screenshot shows a GitHub repository interface. The left sidebar lists files and folders, including config-eureka.yml, which is currently selected and highlighted with a blue background. The main area displays the contents of config-eureka.yml. A red box highlights the line 'port: 8761'. To the right of the code, the text '配置端口为8761' (Configure port to 8761) is overlaid in red. The GitHub UI includes standard navigation elements like Code, Issues, Pull requests, Projects, Wiki, Security, Insights, and a search bar.

```
server:  
  port: 8761  
  
eureka:  
  instance:  
    hostname: localhost  
  server:  
    enable-self-preservation: false  
    eviction-interval-timer-in-ms: 6000  
    peer-node-read-timeout-ms: 90000  
    response-cache-update-interval-ms: 5000  
  client:  
    register-with-eureka: false  
    fetch-registry: false  
    serviceUrl:  
      defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/  
  
spring:  
  profiles: dev  
  application:  
    name: cloud-eureka
```

测试通过连接本地服务配置中心访问远程 git 仓库上的配置文件：



File Edit View Navigate Code Refactor Build Run Tools Git Window Help Transport - bootstrap.yml [transporter_service]

Transport transporter_service src main resources bootstrap.yml

Project registryService springcloud-config target transporter_service .mvn src main java resources application.yml bootstrap.yml logback.xml

bootstrap.yml

Plugins supporting bootstrap.yml files found.

1 spring:
2 cloud:
3 config:
4 name: transporter-service # 仓库中的配置文件名称
5 profile: dev
6 label: master
7 uri: http://localhost:3344

Document 1/1 spring: cloud: config:

Run: SpringcloudConfigApplication RegistryServiceApplication AuthServiceApplication GatewayServiceApplication TransporterServiceApplication UserServiceApplication

Console Actuator

D:\APP\ProgrammingTools\jdk\bin\java.exe ...

成功连接本地服务配置中心

Fetching config from server at : http://localhost:3344

Located environment: name=config-eureka, profiles=[dev], label=master, ve

Located property source: [BootstrapPropertySource {name='bootstrapProper

No active profile set, falling back to 1 default profile: "default"

BeanFactory id=3352c4a0-7010-3106-84f5-4c47159d8520

Tomcat initialized with port(s): 8761 (http) 成功访问远程git仓库

Starting service [Tomcat]

Starting Servlet engine: [Apache Tomcat/9.0.73]

Initializing Spring embedded WebApplicationContext

Root WebApplicationContext: initialization completed in 1062 ms

Initiating Jersey application, version: Jersey 1.10.6, 05/02/2017, 03:20:10

Git Run TODO Profiler Build Dependencies Spring Terminal

Lombok requires enabled annotation processing: Do you want to enable annotation processors? Enable (38 minutes ago)

4:44 CRLF UTF-8 2 spaces Event Log

日志追踪

使用 Spring Cloud Sleuth 来实现分布式跟踪，通过 Sleuth 可以跟踪记录每个请求在系统中的流转情况，帮助我们进行系统监控和故障定位。

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- 日志级别从低到高为TRACE < DEBUG < INFO < WARN < ERROR < FATAL, 如果设置为WARN, 则明示于WARN的消息都不会输出 -->
<!-- scan: 当此属性设置为true时, 配置文件如果发生改变, 将会被重新加载, 默认值为true -->
<!-- scanPeriod: 设置启动配置文件是否有修改的时间间隔, 如果没有给出时间单位, 默认单位是毫秒。 -->
<!-- debug: 此属性设置为true时, 将打印出logback内部日志信息, 实时查看logback运行状态, 默认值为false. -->
<configuration scan="true" scanPeriod="10 seconds">
    <!-- 日志上下文名称 -->
    <contextName>my_logback</contextName>
    <!-- name的值是父级的名称, value的是要定义的值, 通过定义的值会插入到Logger上下文 -->
    <property name="log_path" value="${catalina.base}/user_service/logs"/>
    <!-- 加载Spring 应用上下文 -->
    <springProperty scope="context" name="applicationName"
        source="spring.application.name" defaultValue="localhost"/>
    <!-- 日志输出格式 -->
    <property name="LOG_PATTERN" value="%d{yyyy-MM-dd HH:mm:ss.SSS}
        [%{applicationName},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-}] [%thread] %-5level
        [%logger{50} - %msg%n"/>
    <!-- 配置Appender -->
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <!-- 此Appender是为开发使用, 只配置最低级别, 抑制台输出的日志级别是大于或等于其级别的日志信息 -->
        <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
            <level>DEBUG</level>
        </filter>
        <encoder>
            <pattern>${LOG_PATTERN}</pattern>
        </encoder>
    </appender>
</configuration>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- 日志级别从低到高为TRACE < DEBUG < INFO < WARN < ERROR < FATAL, 如果设置为WARN, 则明示于WARN的消息都不会输出 -->
<!-- scan: 当此属性设置为true时, 配置文件如果发生改变, 将会被重新加载, 默认值为true -->
<!-- scanPeriod: 设置启动配置文件是否有修改的时间间隔, 如果没有给出时间单位, 默认单位是毫秒。 -->
<!-- debug: 此属性设置为true时, 将打印出logback内部日志信息, 实时查看logback运行状态, 默认值为false. -->
<configuration scan="true" scanPeriod="10 seconds">
    <!-- 日志上下文名称 -->
    <contextName>my_logback</contextName>
    <!-- name的值是父级的名称, value的是要定义的值, 通过定义的值会插入到Logger上下文 -->
    <property name="log_path" value="${catalina.base}/user_service/logs"/>
    <!-- 加载Spring 应用上下文 -->
    <springProperty scope="context" name="applicationName"
        source="spring.application.name" defaultValue="localhost"/>
    <!-- 日志输出格式 -->
    <property name="LOG_PATTERN" value="%d{yyyy-MM-dd HH:mm:ss.SSS}
        [%{applicationName},%X{X-B3-TraceId:-},%X{X-B3-SpanId:-}] [%thread] %-5level
        [%logger{50} - %msg%n"/>
    <!-- 配置Appender -->
    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <!-- 此Appender是为开发使用, 只配置最低级别, 抑制台输出的日志级别是大于或等于其级别的日志信息 -->
        <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
            <level>DEBUG</level>
        </filter>
        <encoder>
            <pattern>${LOG_PATTERN}</pattern>
        </encoder>
    </appender>
</configuration>
```

控制台输出链路追踪的内容

总结：

本报告详细介绍了基于微服务架构的物流控制系统的设计，使用了一系列 Spring Cloud 技术包括 Spring Cloud Netflix（Eureka、Hystrix）、Spring Cloud Security（Oauth2）、Spring Cloud Gateway、Spring Cloud Config 和 Spring Cloud Sleuth，实现了物流项目的服务拆分、统一网关、服务通信、认证授权、异常处理与容错机制、配置管理与日志追踪等功能。

其中：

1. 微服务架构：系统采用微服务架构，将功能拆分为多个独立的服务，每个服务专注于特定的业务功能。这种架构提供了松耦合、可扩展和易于维护的优势。

2. Spring Cloud Netflix：

Eureka：作为服务注册和发现的组件，Eureka 允许各个微服务在运行时注册和发现彼此，实现动态扩展、负载均衡和故障恢复等功能。

Hystrix：通过实现断路器模式，Hystrix 提供了容错能力，保护系统免受服务故障和延迟的影响，并提供了监控和统计功能。

3. Spring Cloud Security（Oauth2）：采用 Oauth2 作为授权框架，实现了安全的认证和授权机制，确保只有经过授权的用户和服务才能访问系统的敏感数据和功能。

4. Spring Cloud Gateway：作为系统的 API 网关，Spring Cloud Gateway 提供了集中式路由和过滤的功能，对外统一暴露 API，并处理鉴权、请求转发和负载均衡等功能，同时支持动态路由配置。

5. Spring Cloud Config：使用 Spring Cloud Config 来管理和集中管理系统的配置文件，实现了配置的集中化管理、动态刷新和版本控制，使得配置更易于管理和更新。

6. Spring Cloud Sleuth：通过集成 Spring Cloud Sleuth，系统实现了分布式追踪和日志跟踪功能，可以跟踪请求在各个微服务之间的流动，并记录关键日志信息，有助于排查和解决分布式系统中的问题。

通过采用上述技术，物流控制系统实现了高可用性、弹性扩展、安全性和可维护性。微服务架构提供了灵活性和可扩展性，Spring Cloud 技术栈提供了丰富的组件和工具，使得系统在开发、部署和维护过程中更加容易，同时满足了安全性、性能和可观测性等要求。总体而言，该系统设计报告展示了高效和可靠的物流控制系统的架构和实现方案。