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：

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 Kafka 技术改造成基于 Kafka 消息队列集成的微服务架构，并使用 Spring Cloud Config 技术进行集中配置和 Spring Cloud Sleuth 技术进行集中式日志跟踪。

本报告将详细介绍使用了基于 Kafka 消息队列集成的微服务架构，采用 Spring Cloud Config 进行集中配置，Spring Cloud Sleuth 进行集中式日志跟踪实现的物流控制系统的技术实现。

技术栈：

kafka：

Kafka 是一个分布式、事件驱动的消息传递平台，允许高度可扩展、容错和实时的数据处理。它是为高吞吐量、低延迟和持久的数据流而设计的。

其优点包括：
1.可扩展性：Kafka 具有水平可扩展性，可以处理大量数据流和并发消费者。
2.容错：Kafka 旨在通过跨集群中的多个节点复制数据来提供容错。它确保了数据在发生故障时的持久性和弹性。
3.实时处理：Kafka 支持近乎实时的数据流，允许应用程序实时处理数据并对其做出反应。
4.事件驱动架构：Kafka 的发布-订阅模型支持事件驱动架构的开发，其中服务通过异步事件进行通信。

Zookeeper：

ZooKeeper 是一个开源的分布式协调服务，用于管理和协调分布式应用程序中的大规模集群。

其优点包括：
1. 高可用性：ZooKeeper 采用了分布式的架构，数据存储在多个节点上，当其中一个节点故障时，其他节点可以接替其职责，确保服务的持续可用性。
2. 一致性：ZooKeeper 提供了强一致性的数据模型，所有对数据的更新操作都会被线性化地执行，保证了多个客户端之间的数据一致性。
3. 可靠性：ZooKeeper 将数据存储在内存中，并将其持久化到磁盘上，这样即使服务器发生故障或重启，数据也不会丢失。

4.简单易用的 API：ZooKeeper 提供了简单易用的 API，使得开发人员可以轻松地进行分布式应用程序的开发和管理。
5.顺序访问：ZooKeeper 可以为每个数据节点分配一个全局唯一的顺序号，使得客户端可以按照顺序访问数据节点，实现有序性。

事件驱动架构

事件驱动体系结构是一种体系结构模式，其中服务通过生成和使用事件进行通信。事件是表示系统中发生的更改或重大事件的消息，服务对这些事件作出异步反应。

其优点包括：
1.松耦合：事件驱动体系结构中的服务是解耦的，允许独立的开发、可扩展性和灵活性。
2.响应能力：服务可以实时对事件做出反应，从而能够对系统中的更改做出快速及时的响应。
3.可扩展性：事件驱动的体系结构可以通过添加更多的事件生产者和消费者来水平扩展，从而确保高吞吐量并处理事件负载的峰值。
4.可扩展性：可以通过订阅相关事件轻松添加新服务，而不会影响现有服务。

集中式配置

在集中式配置方法中，专用系统或服务用于在一个中心位置存储和管理多个应用程序或服务的配置属性。

其优点包括：
1.一致性：集中化配置可确保所有服务使用相同的配置设置，从而减少整个系统中出现配置不一致的可能性。
2.易于管理：配置属性可以在一个中心位置进行更新和管理，简化了部署和维护过程。
3.动态更新：集中式配置允许在不需要重新启动服务的情况下进行动态更新，从而实现灵活的配置更改。
4.版本控制和历史记录：集中式配置系统通常提供版本控制和配置更改的历史跟踪，从而在出现问题时更容易恢复到以前的版本。

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：

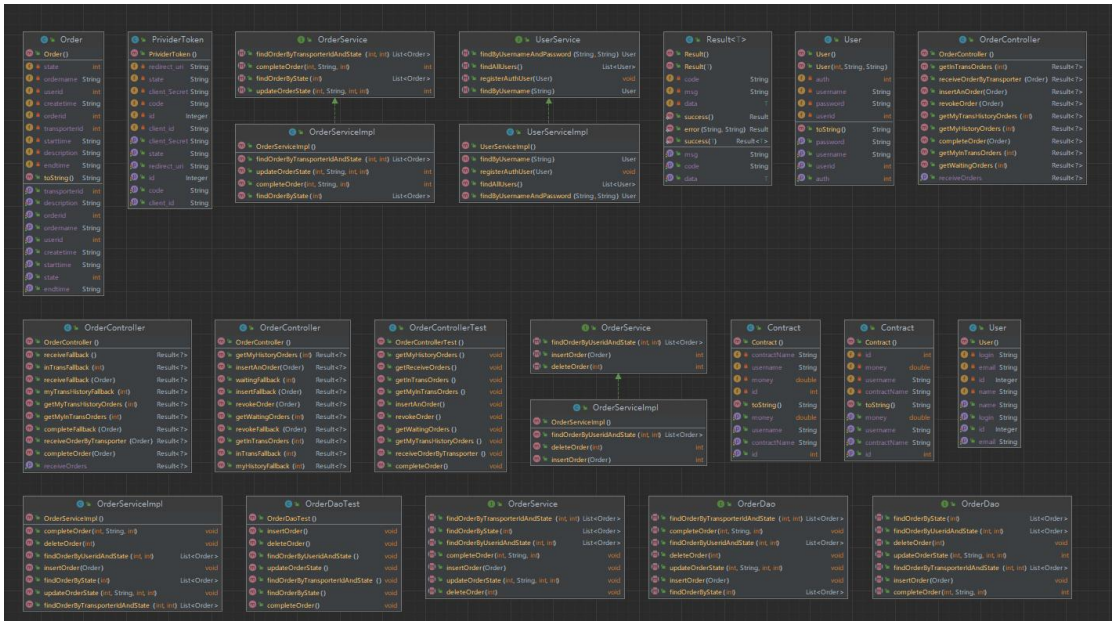
Spring Cloud Sleuth 是 Spring 生态系统中微服务架构的分布式跟踪框架，它会自动向系统的日志和服务间通信中添加一些跟踪/元数据（通过请求标头），跟踪特定的请求日志。

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：

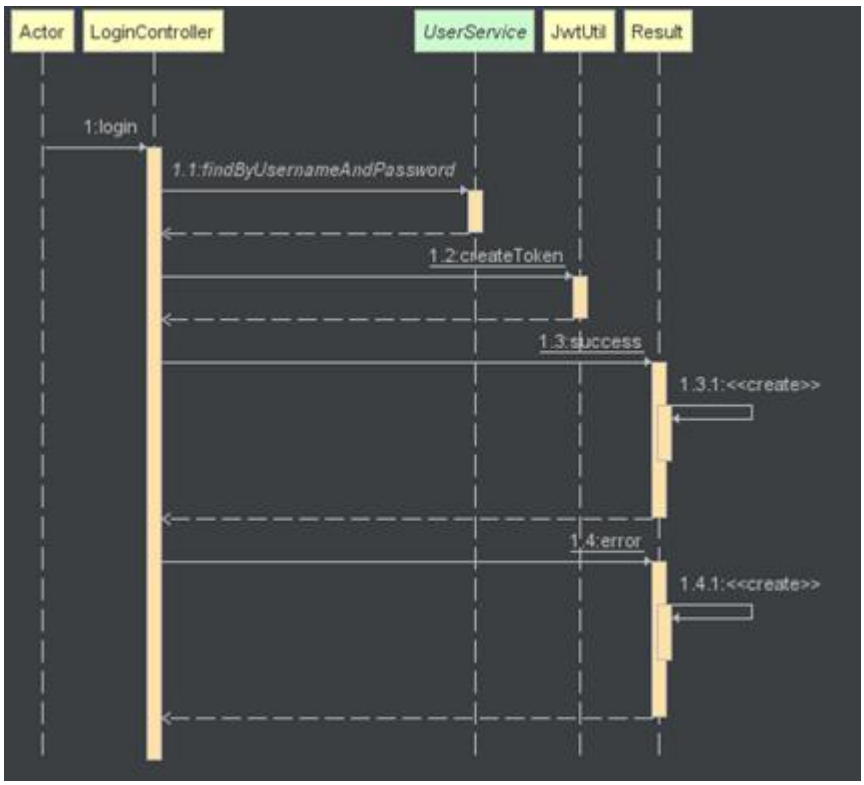
用于服务的注册与发现，服务通过注册中心进行服务间的通信。

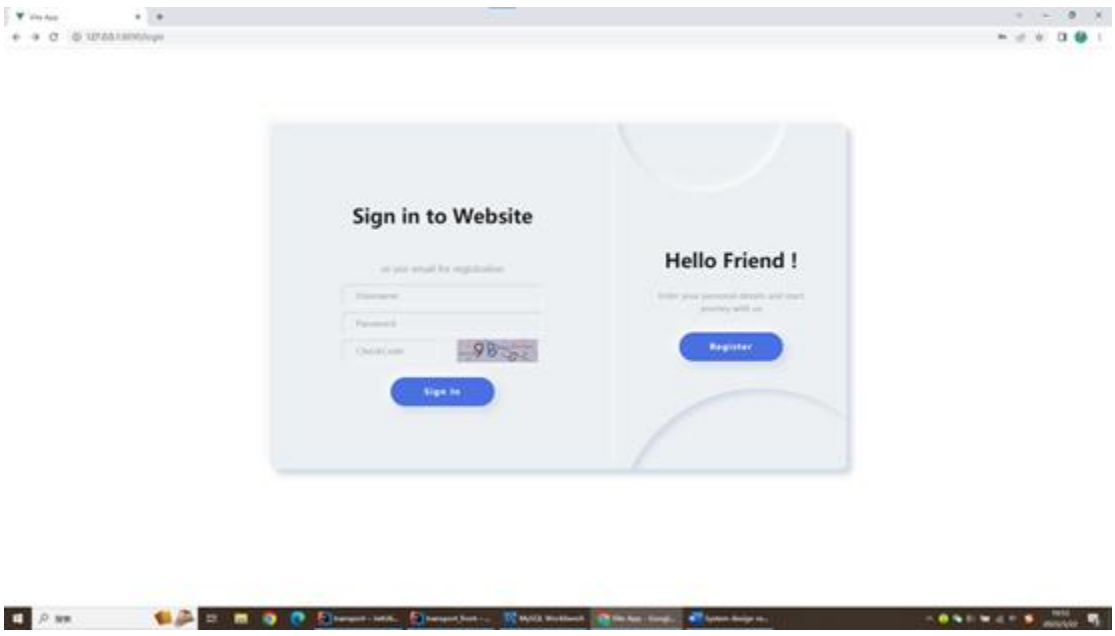
gatewayService 网关服务模块：负责请求的路由和过滤，统一各服务模块的访问接口，是系统对外的唯一入口。

auth_service 授权服务模块：

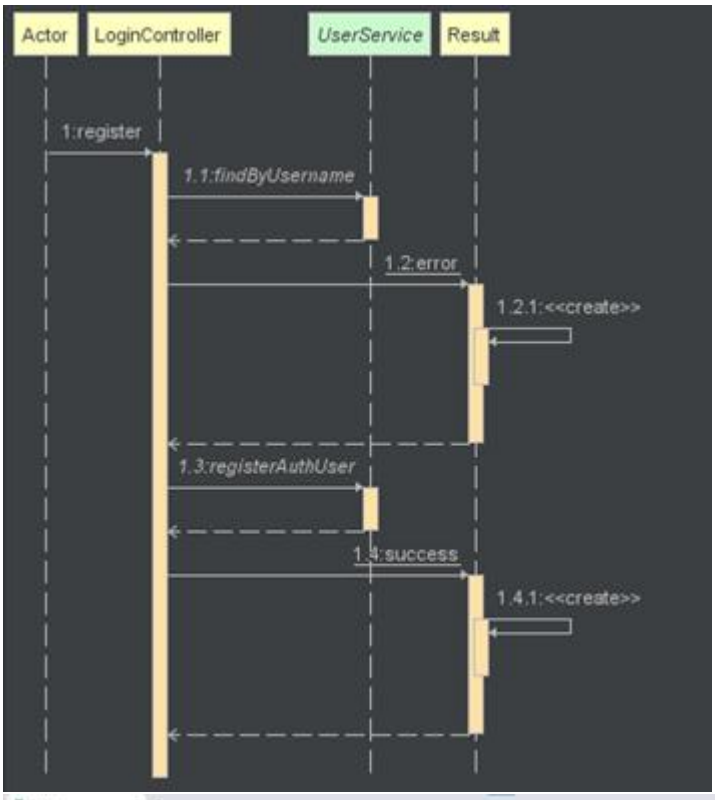
负责处理用户和运输者的登录注册相关功能，提供用户认证、权限管理、JWT token 生成、登录、注册、登出等相关认证功能。

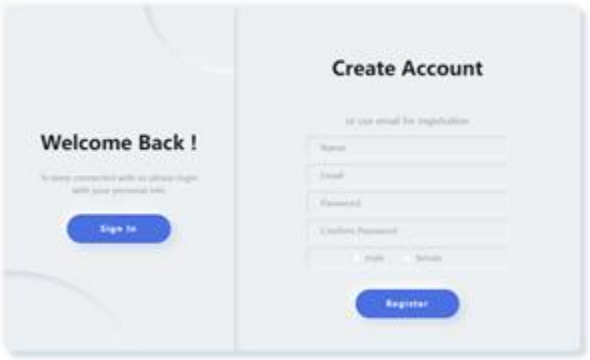
用户登录：


Actor
LoginController
UserService
JwtUtil
Result
1:login
1.1:findByUsernameAndPassword
1.2:createToken
1.3:success
1.3.1:<<create>>
1.4:error
1.4.1:<<create>>


Sign in to Website
Hello Friend !

用户注册：

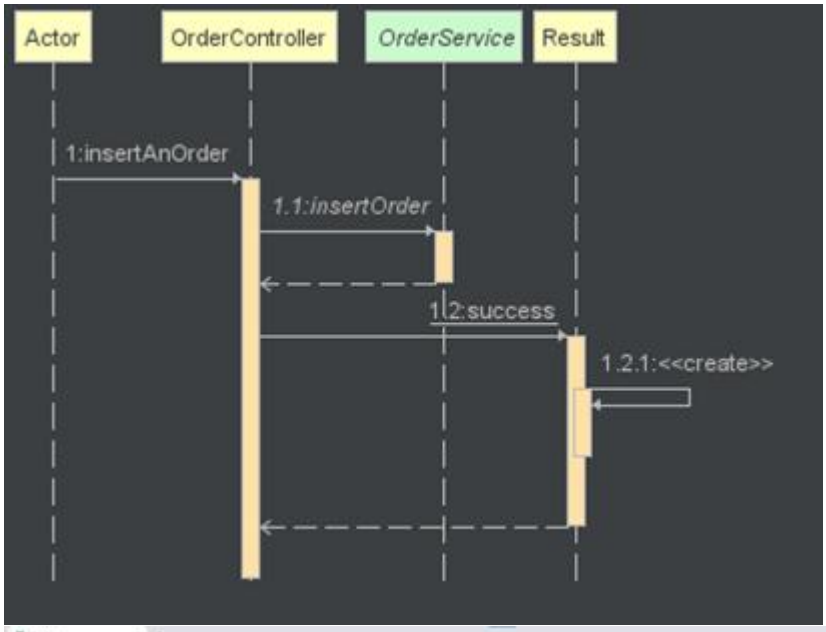


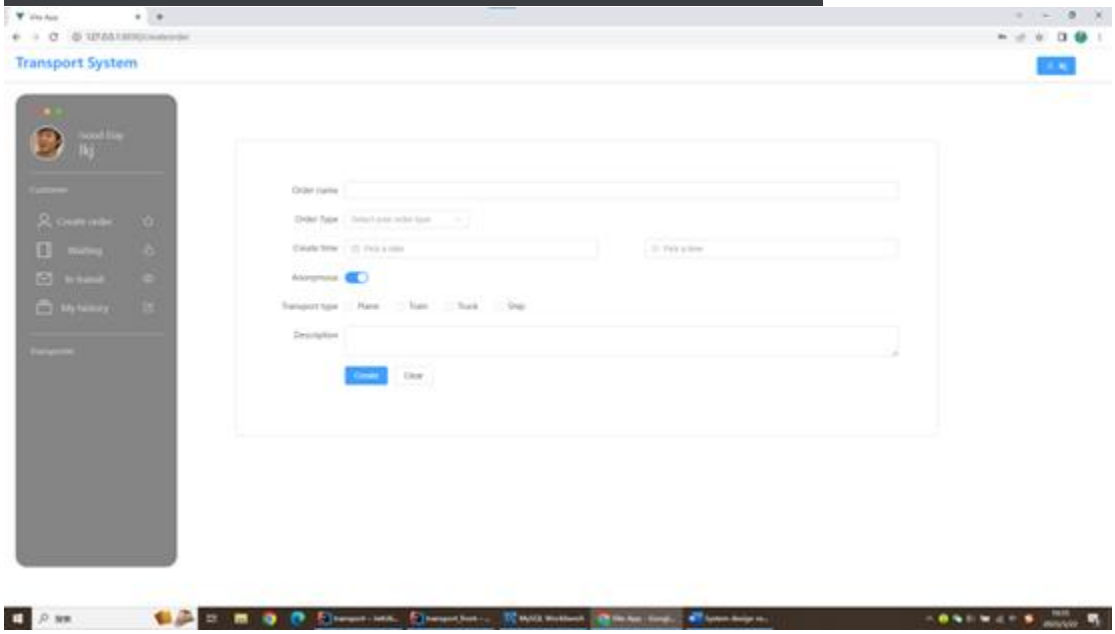


user_service 客户服务模块：

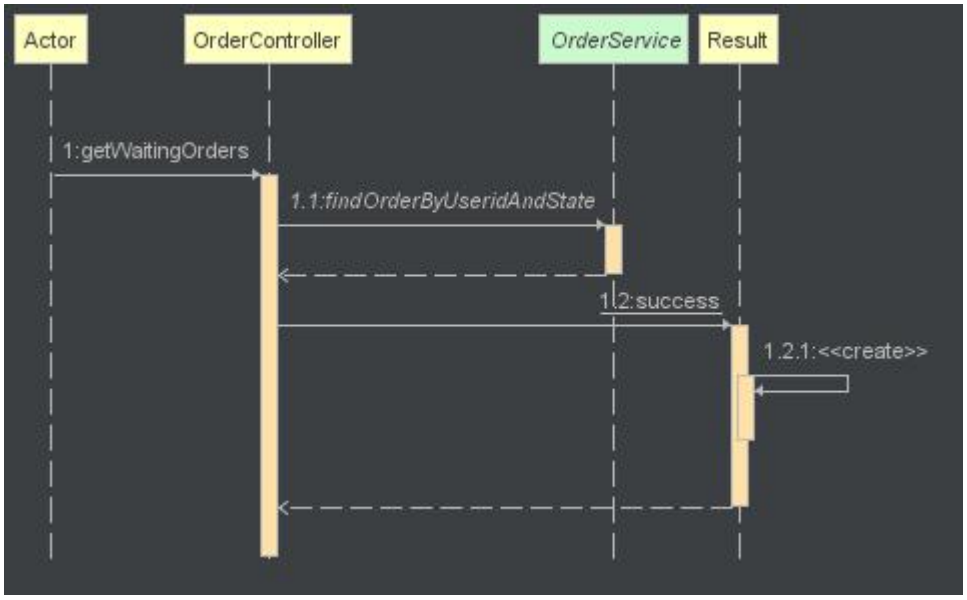
负责处理用户相关的业务逻辑，包括发起订单、撤回订单、查看等待中订单、查看运输中订单、查看已完成订单等用户业务功能。

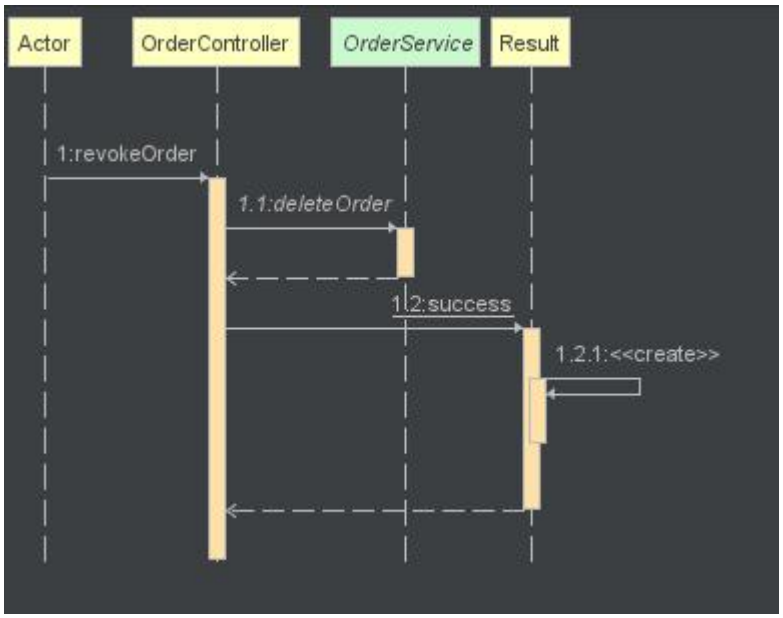
创建订单：


Actor
OrderController
OrderService
Result
1:insertAnOrder
1.1:insertOrder
1.2:success
1.2.1:<<create>>


Transport System

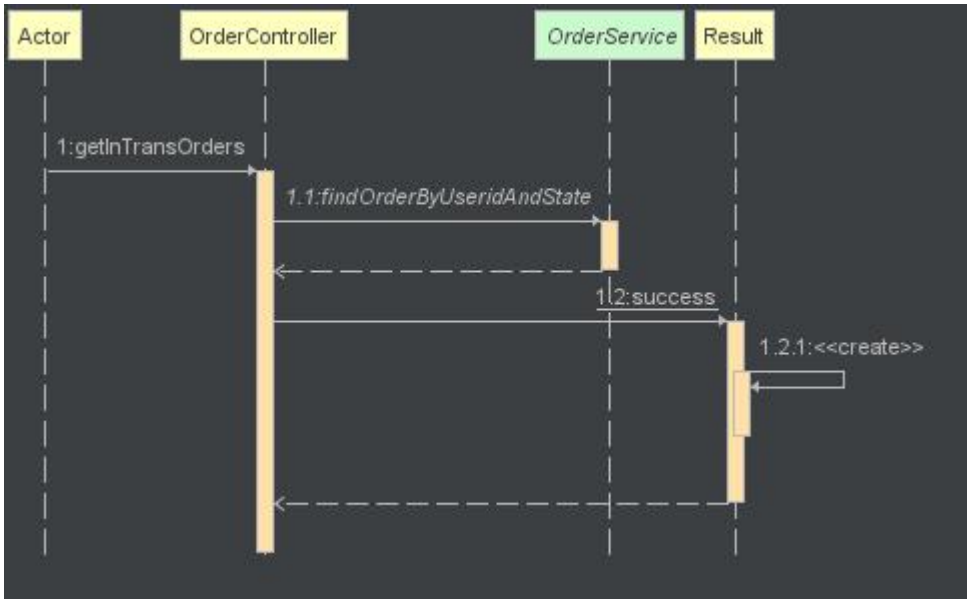
查看等待中的订单：





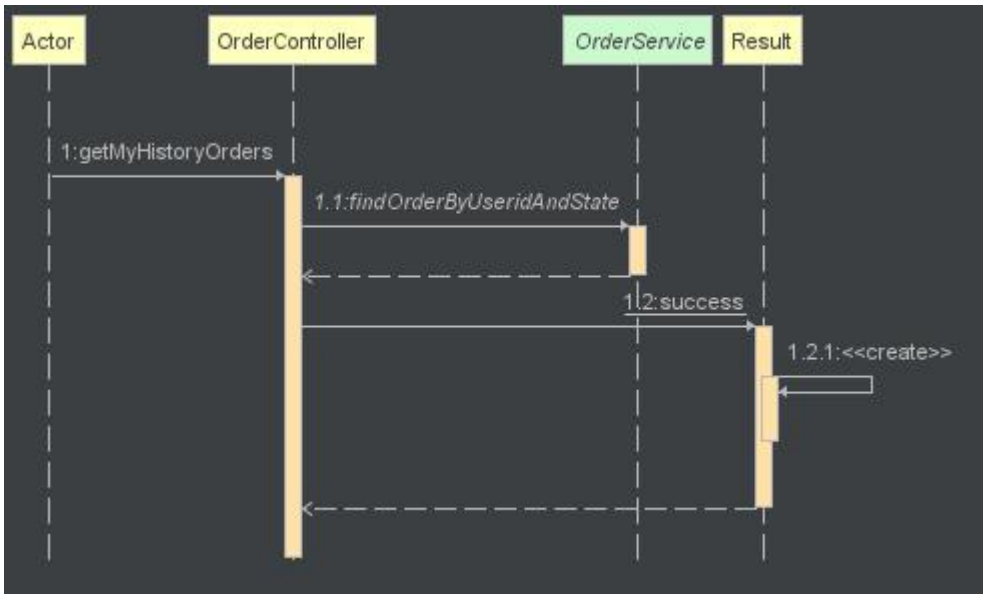
Name	Description	Create Time	User Id	State	Operations
222	dasdad	2023-05-Mo 19:55:52	1	Waiting be transported	Revoke
444	Ship project	2023-05-Mo 19:56:10	1	Waiting be transported	Revoke

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：

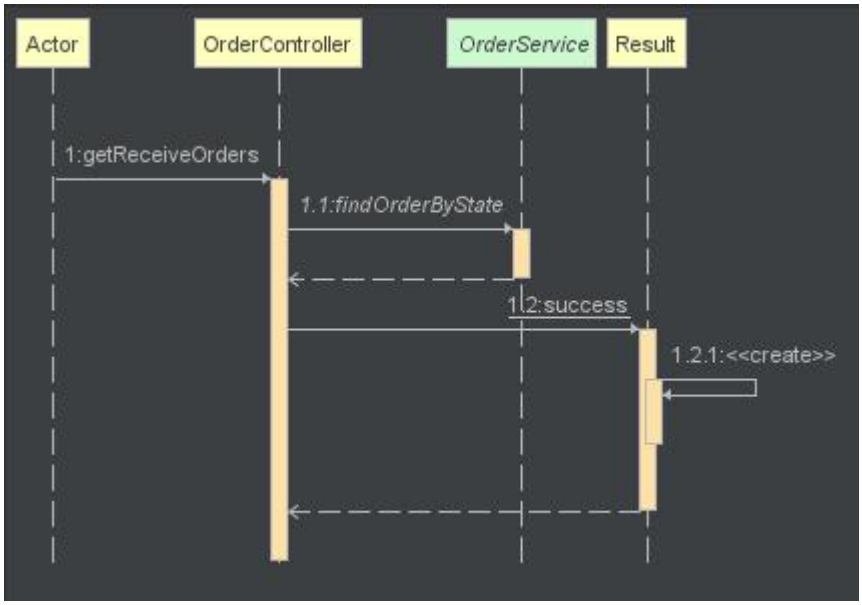


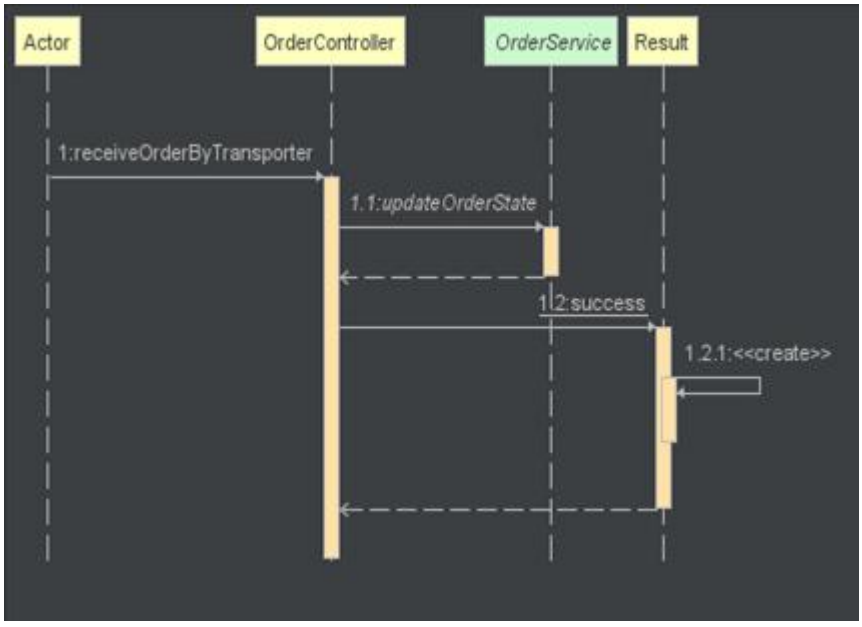
Name	Description	Create time	Userid	Transporterid	Take order time	Completed time	State
firstorder	nothing	2023-05-Mo 12:42:42	1	2	2023-05-Mo 12:42:42	2023-05-22 15:01:44	Completed
third	nothing	2023-05-Mo 12:42:42	1	2	2023-05-Mo 12:42:42	2023-05-Mo 12:42:42	Completed
four	nothing	2023-05-Mo 12:42:42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

transporter_service 运输者服务模块：

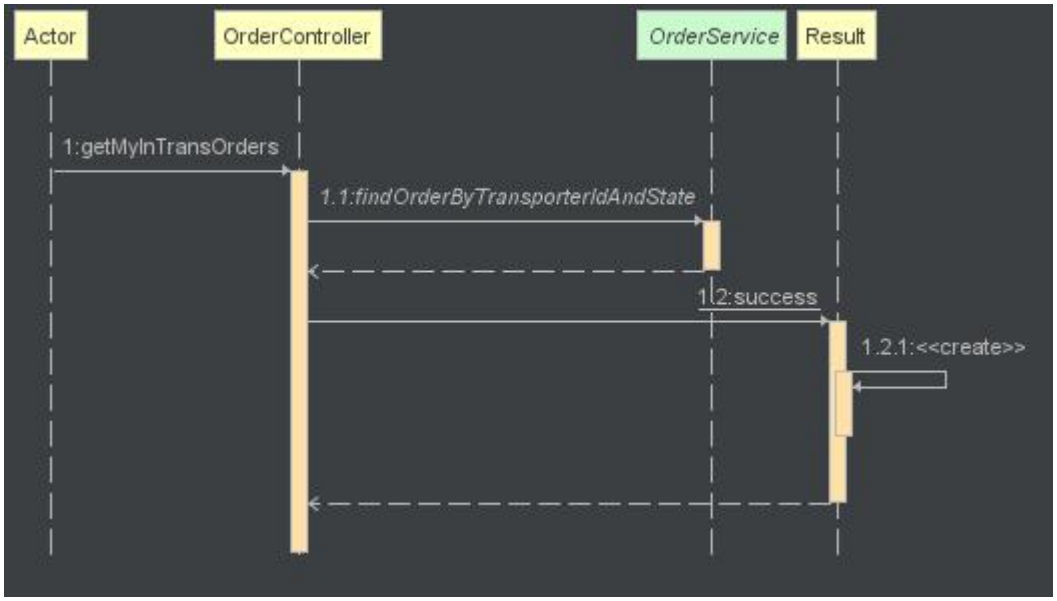
负责处理运输者相关的业务逻辑，包括接收订单、查看己方正在运输的的的订单，完成订单，查看运输历史等运输者业务功能。

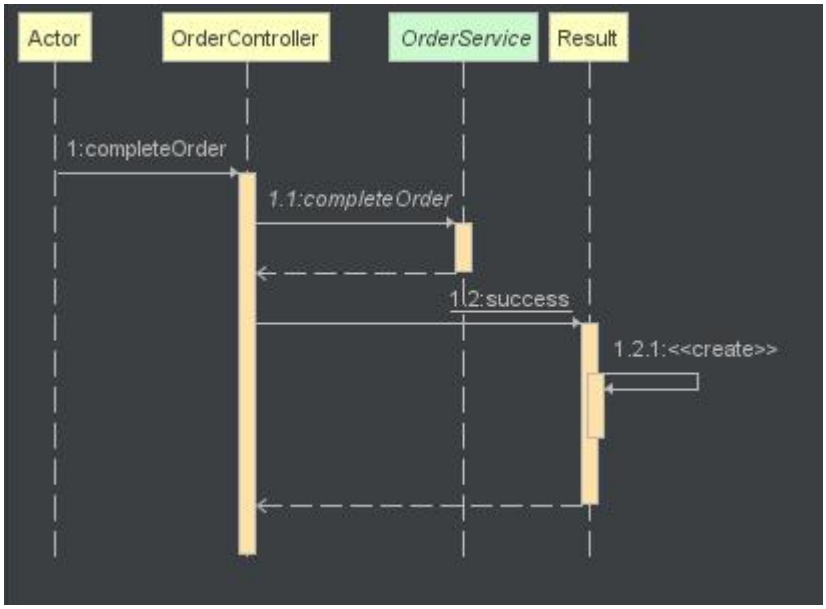
承接订单：





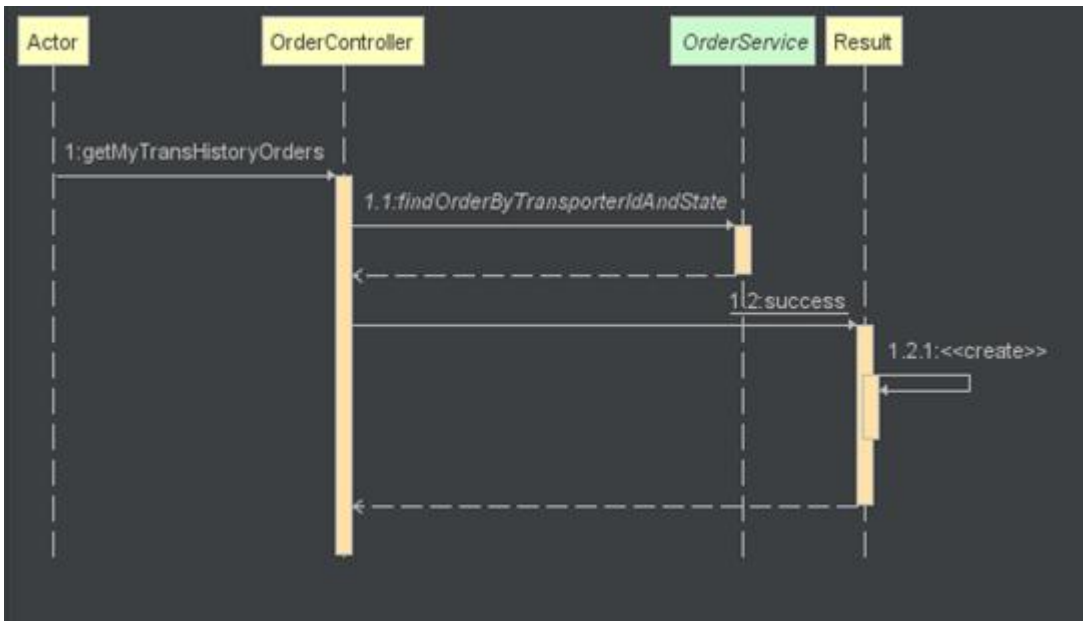
查看运输中的订单：





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

查看历史运输记录：

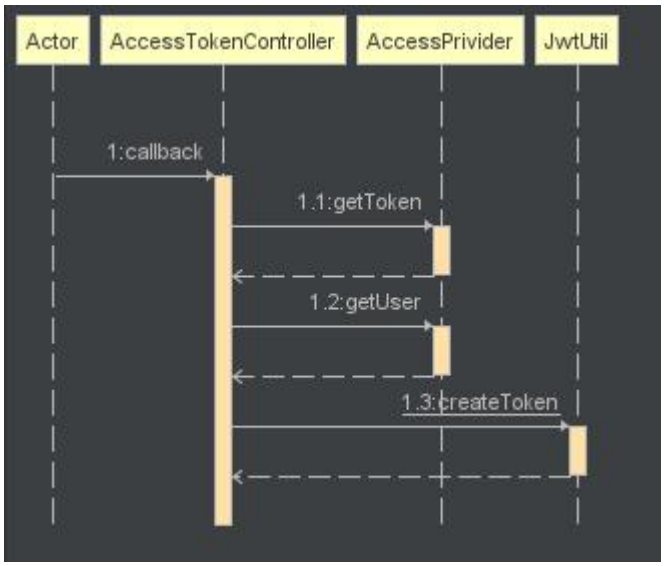


Name	Description	Create time	UserId	TransporterId	Take order time	Completed time	State
firstorder	nothing	2023-05-Mo 12:42:42	1	2	2023-05-Mo 12:42:42	2023-05-22 15:01:44	Completed
third	nothing	2023-05-Mo 12:42:42	1	2	2023-05-Mo 12:42:42	2023-05-Mo 12:42:42	Completed
four	nothing	2023-05-Mo 12:42:42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

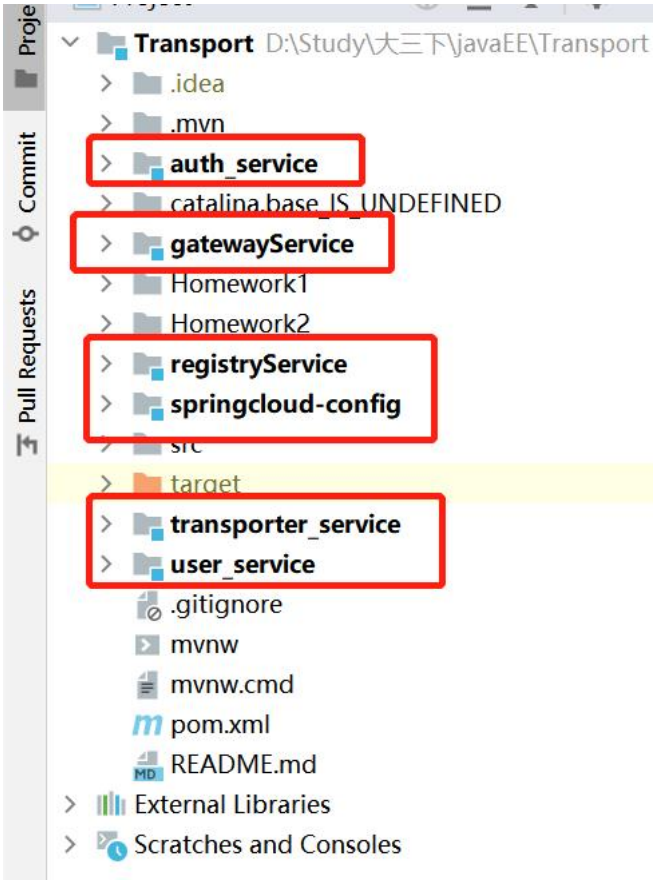
oauth2_service 第三方登录模块：

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：



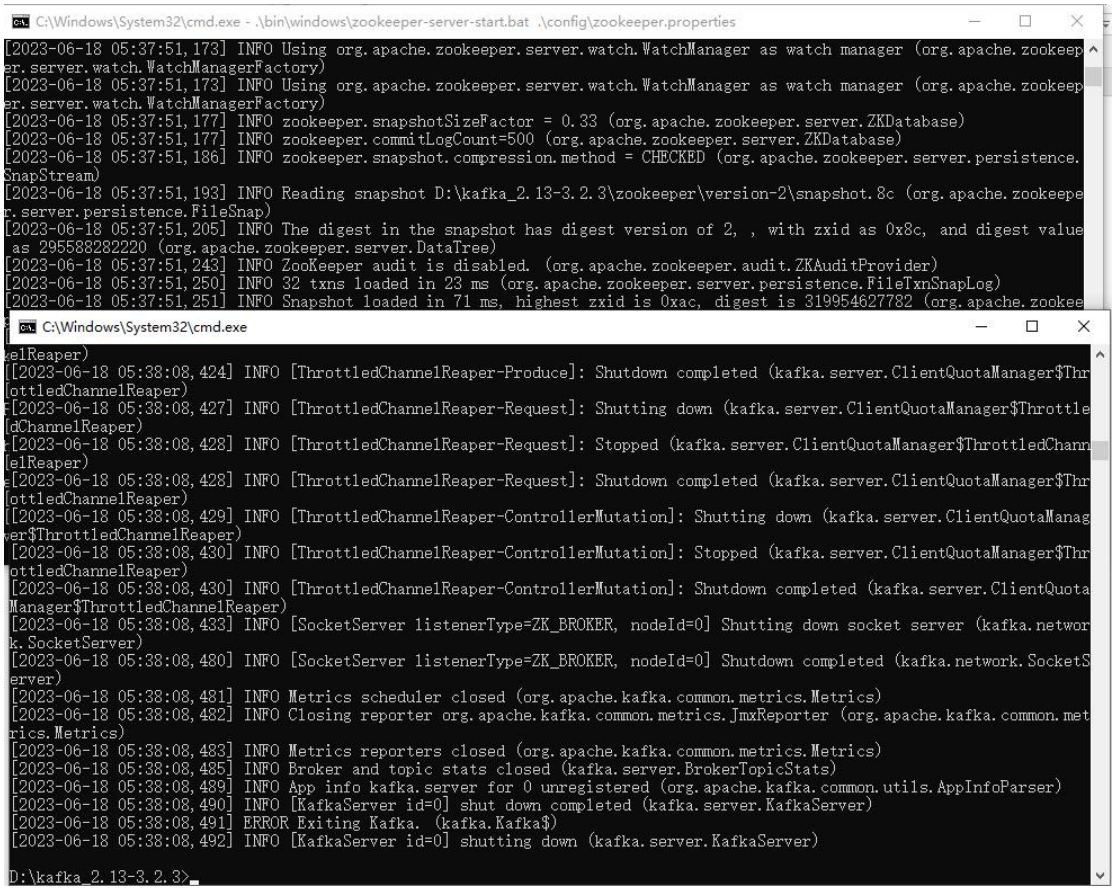
服务拆分

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

集成 Kafka

本地启动 zookeeper 和 kafka：



配置封装 kafka 生产者，消费者：

```java
@Component
public class EventProducer {
    2 usages
    private final KafkaTemplate<String, String> kafkaTemplate;
    no usages
    private final String topic = "topic"; // Kafka主题名称
    no usages  new *
    @Autowired
    public EventProducer(KafkaTemplate<String, String> kafkaTemplate) { this.kafkaTemplate = kafkaTemplate; }
    2 usages  new *
    public void sendEvent(String topic,
                          String message) {
        kafkaTemplate.send(topic, message);
    }
}
```

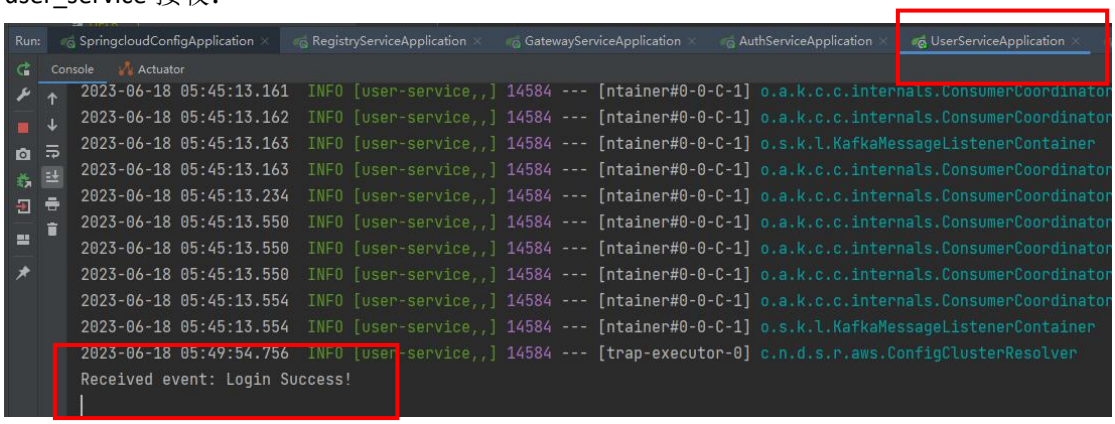
```java
@Component
public class EventConsumer {

    no usages  new *
    @KafkaListener(topics = "auth", groupId = "my_group")
    public void receiveEvent(String message) {
        // 处理接收到的事件消息
        System.out.println("Received event: " + message);
    }
```

示例：

auth_service 使用封装好的 sendEvent 函数将消息发到 kafka 队列：

```
if(result.getAuth() == 0){
    eventProducer.sendEvent( topic: "user", message: "Login Success!");
}else{
    eventProducer.sendEvent( topic: "trans",  message: "Login Success!");
}
```

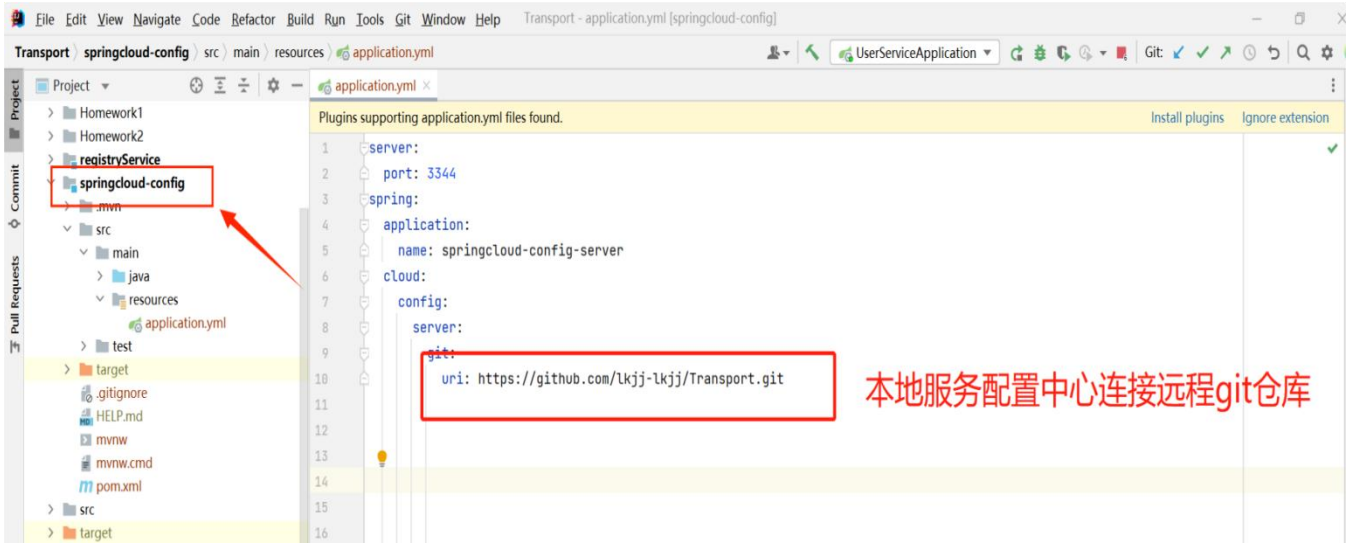
user_service 接收：



配置管理

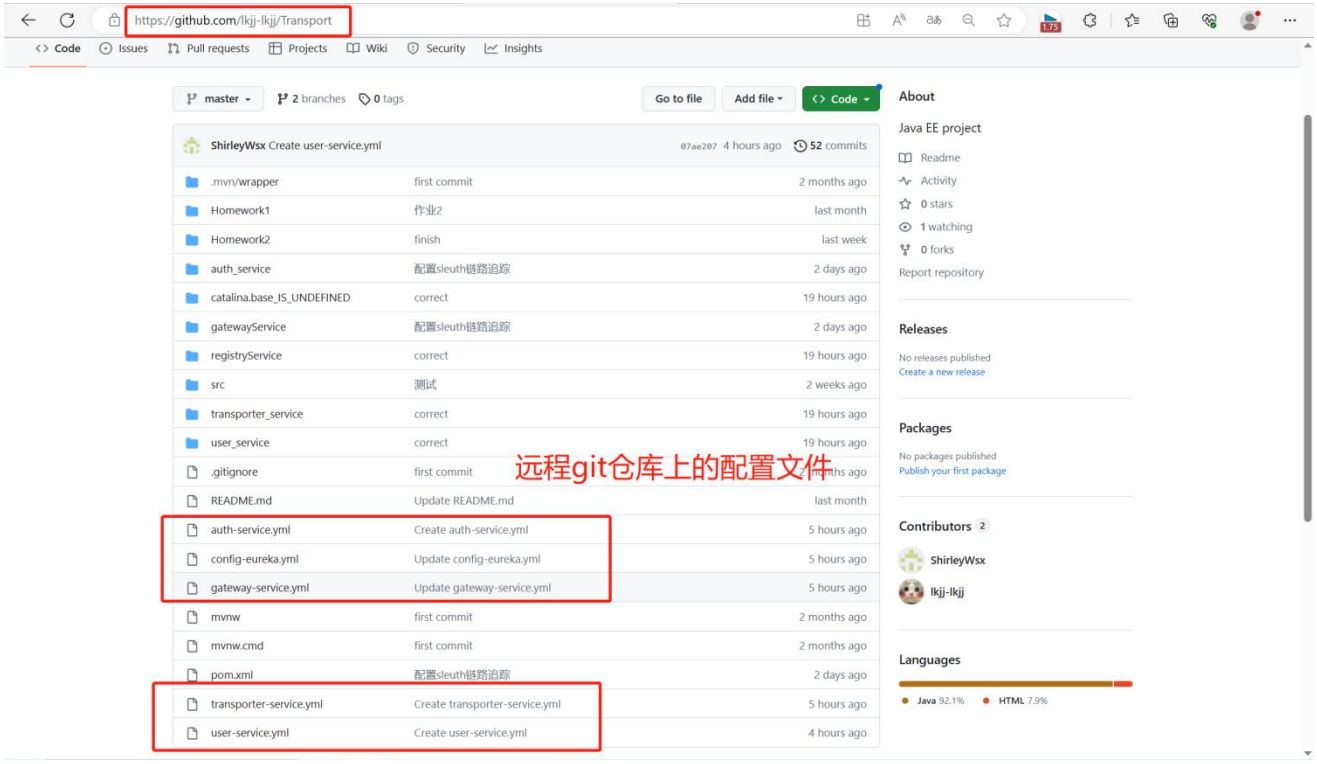
为了方便管理系统的配置信息，引入 Spring Cloud Config。Spring Cloud Config 可以集中管理各个微服务的配置文件，并提供 REST 接口供其他服务获取配置信息。

springcloud-config 本地服务配置中心：

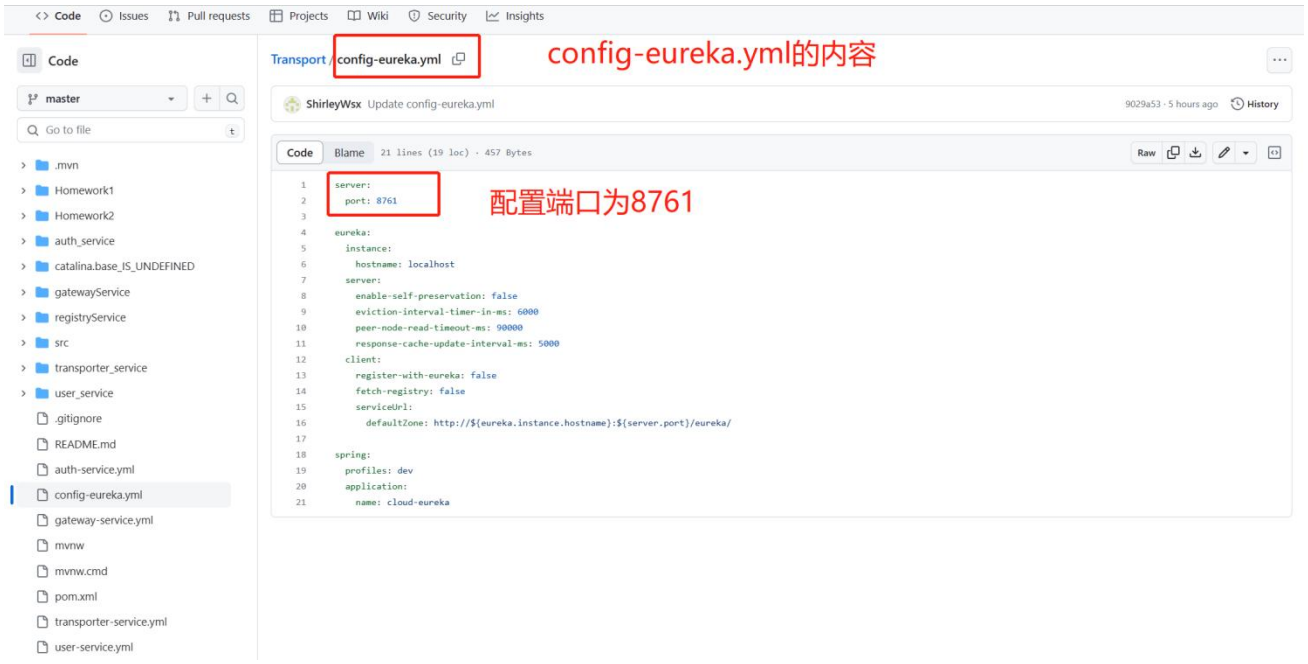


远程 GitHub 仓库上配置文件，分别有 auth-service.yml、config-eureka.yml、gateway-service.yml、

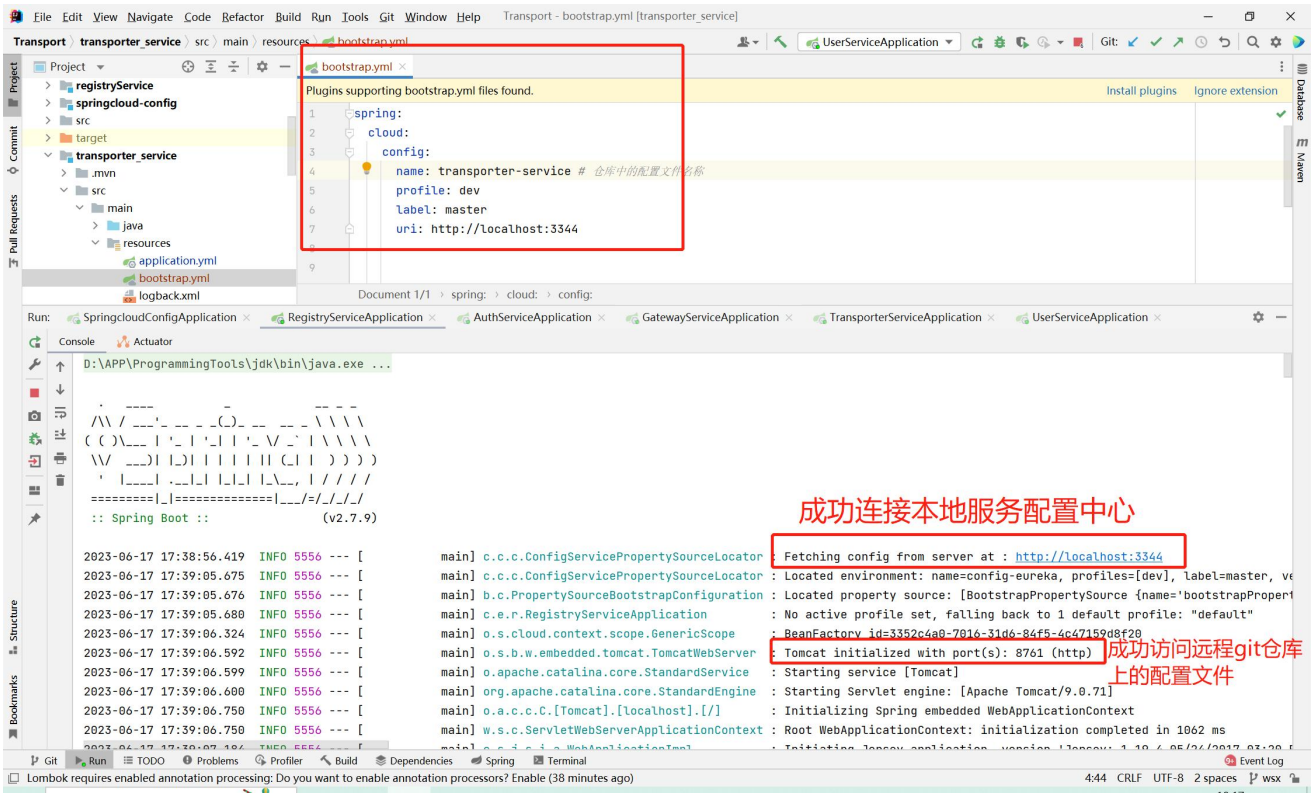
transporter-service.yml、user-service.yml：



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

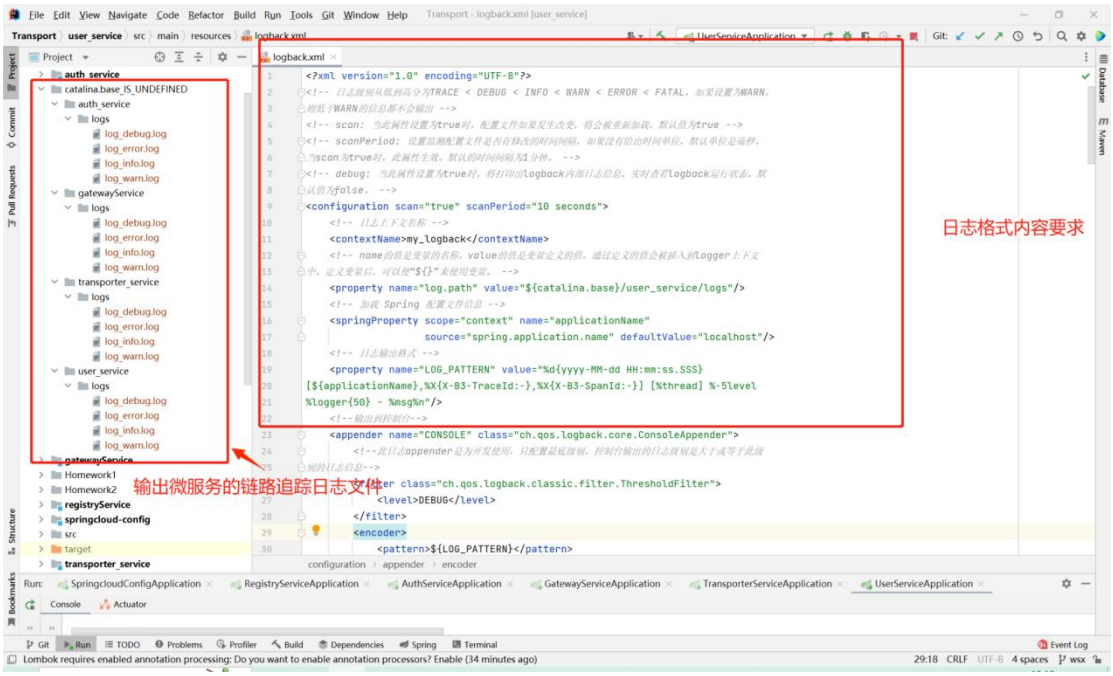


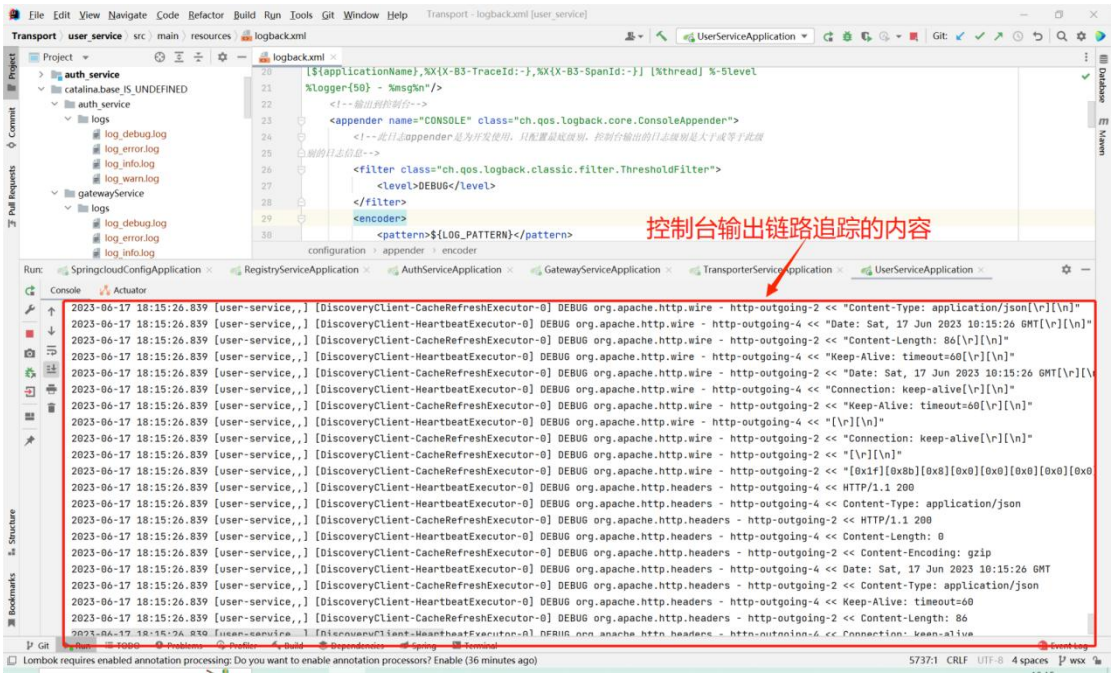
测试通过连接本地服务配置中心访问远程 git 仓库上的配置文件：



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。





总结：

本报告详细介绍了基于 Kafka 消息队列集成的微服务架构，采用 Spring Cloud Config 进行集中配置，Spring Cloud Sleuth 进行集中式日志跟踪实现的物流控制系统的设计，实现了物流项目中 service 间的互相通信，配置管理与日志追踪等功能。

其中：

1.微服务架构与事件驱动：系统采用微服务架构，将功能拆分为独立的微服务。为了实现松耦合和高扩展性，引入事件驱动的架构模式。微服务之间通过使用 Kafka 消息队列进行异步通信，通过发布和订阅事件实现解耦和灵活性。

2.Kafka 消息队列：作为事件驱动架构的中心组件，Kafka 提供了可靠的、高吞吐量的消息传递机制。微服务可以发布事件到 Kafka 主题，同时订阅感兴趣的主题进行处理。这种异步通信模式使得系统具备了松耦合、可伸缩和弹性的特性。

3.Spring Cloud Config：系统使用 Spring Cloud Config 进行集中配置管理。通过将配置文件存储在 Git 仓库中，Spring Cloud Config 实现了配置的集中化管理、版本控制和动态刷新。各个微服务可以通过 Config Server 获取最新的配置信息，实现了配置的灵活性和可维护性。

4.Spring Cloud Sleuth：通过集成 Spring Cloud Sleuth，系统实现了集中式的日志跟踪功能。Spring Cloud Sleuth 为每个请求分配唯一的跟踪 ID，并将跟踪 ID 添加到日志消息中，从而实现了请求在各个微服务之间的流程追踪。这有助于排查和解决分布式系统中的问题，并提供了对系统性能和异常情况的全面可观察性。

通过采用上述技术，该物流控制系统基于 Kafka 消息队列实现了事件驱动的微服务架构。采用 Spring Cloud Config 进行集中配置管理，可以快速调整系统行为和属性。Spring Cloud Sleuth 提供了集中式的日志跟踪，帮助定位和解决分布式系统中的问题。这样的设计提供了高度灵活性、可伸缩性和可观察性，使得系统能够满足需求的变化并具备高性能和可靠性。