



A Shipping and Transportation Web  
application Development with Spring MVC  
and More

**Assignment 4**

学院：软件学院

专业：软件工程

项目成员：

20301093 赵天舒

20301115 祁麟

指导教师：曾立刚

创建时间：2023.6.15

---

# 目录

1. 需求分析 .....	3
2. 系统架构设计 .....	3
3. 数据库设计 .....	12
4. 接口设计 .....	14
5. 系统测试 .....	15
6. 安全设计 .....	18
7. 性能优化 .....	21
8. 部署方案 .....	21
9. 总结 .....	23

## 1. 需求分析

本系统是一个航运和运输 Web 应用程序，主要目的是为用户提供货物运输的服务，包括货物的运输、物流跟踪、订单管理等功能。本系统主要面向以下用户：

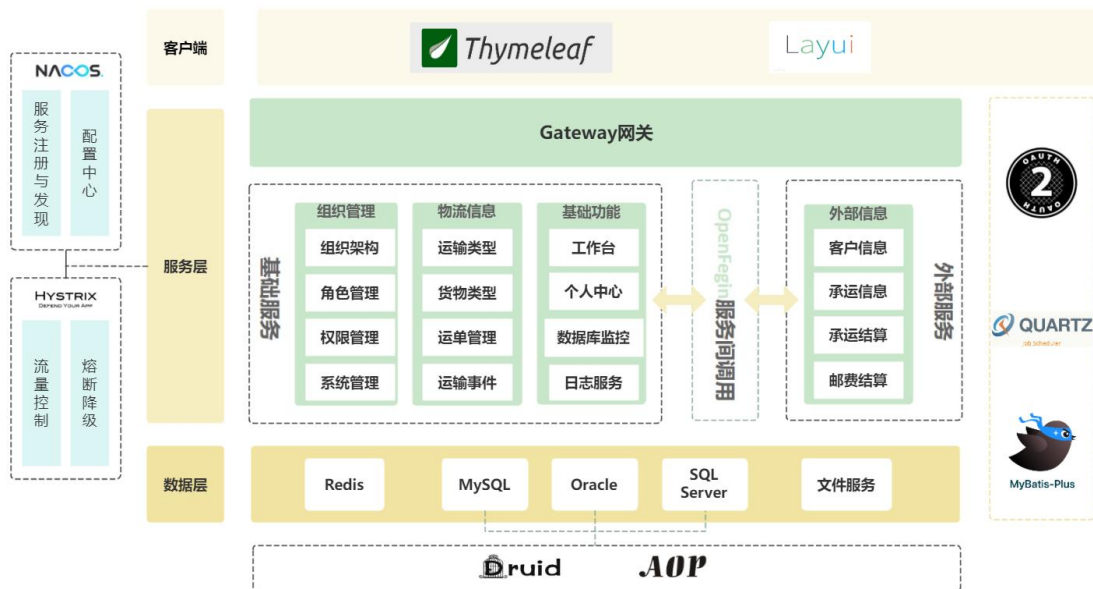
管理员：具备系统所有功能权限；

业务员：负责运输信息管理、运单管理、财务管理等。

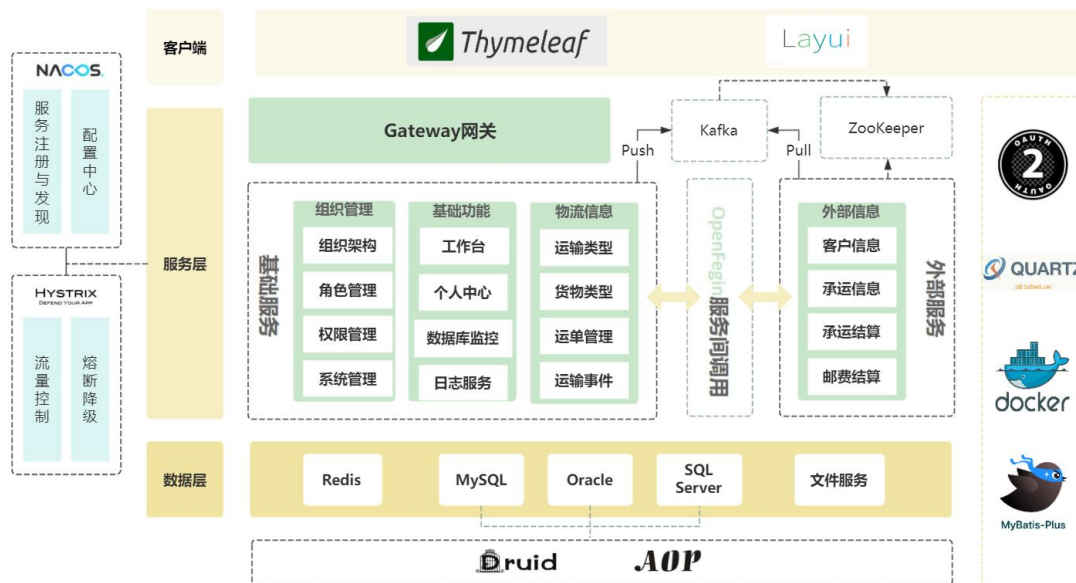
在本系统中，管理员需要进行组织管理，包括组织架构信息管理、角色管理、权限管理与系统管理。业务员需要管理运输信息，包括运输类型、货物类型、客户信息、承运信息的管理；需要管理运单，包括运单的创建、状态管理、运输事件管理等；需要管理财务收支，包括承运结算与邮费结算等。

## 2. 系统架构设计

本系统采用微架构，在上一阶段中系统架构图如下所示：



本阶段开发主要完成了 zookeeper 与 kafka 的配置，以及 DockerFile 文件的编写，系统架构如图所示：



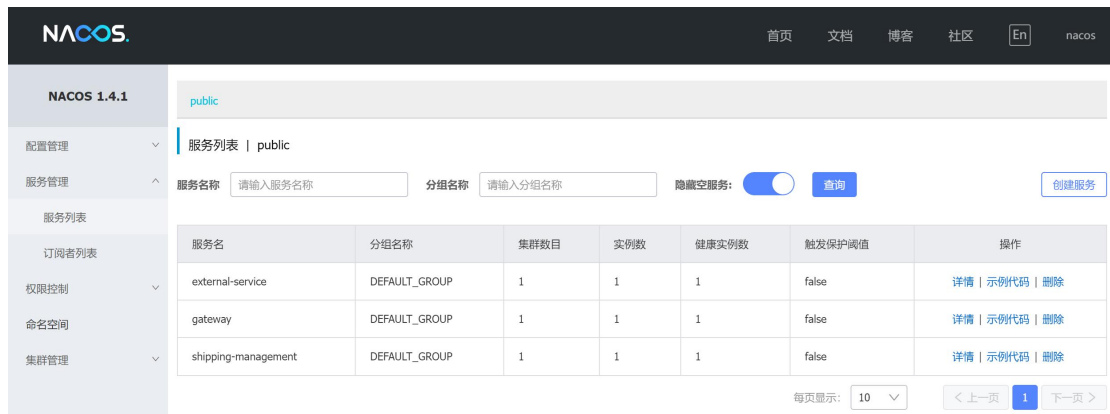
## 微服务拆分

基于 Spring Cloud Alibaba 框架将服务按照物流系统基础服务 shipping-management 和外部服务 external-service 拆分为两部分，并使用 OpenFeign 实现服务间调用。通过服务拆分，shipping-management 服务可专注于处理物流系统的核心功能，例如运单管理、物流流程等，而 external-service 服务则负责与外部系统进行交互，获取客户信息和承运信息等。

## 使用 Nacos 实现服务注册与发现

为了能够有效地管理和调用这些微服务，我们使用了 Nacos 作为服务注册与发现的解决方案。

Nacos 提供了一个易于使用的、高可用的、支持微服务环境的服务注册与发现功能。当微服务启动时，它们会将自身的服务信息注册到 Nacos 服务注册中心。然后，其他的微服务可以通过 Nacos 服务注册中心发现并调用这些服务。



## Gateway

在我们的物流管理系统中，Spring Cloud Gateway 起着关键作用，它作为所有微服务的前端入口，暴露出统一的 API 接口给外部的用户。其主要功能包括：

**路由转发：**Spring Cloud Gateway 能够根据请求信息（如 URL 路径、HTTP 方法、请求参数等）将请求转发到相应的微服务。关于 ShippingApplication 的请求会被路由到 /shipping/\*\*，而关于 ExternalApplication 的请求会被路由到 /external/\*\*。

**身份验证与授权：**Gateway 可以在将请求转发到内部服务前完成身份验证和授权，这样可以提高系统的安全性。我们系统中通过 Spring Security 和 OAuth2 实现了统一的身份验证和授权。

## Hystrix 集成 Gateway

在我们的物流管理系统中，我们使用 Netflix 的 Hystrix 作为熔断器。并在 Gateway 中进行配置。当 /external/\*\* 路径的请求到达时，将使用名为 externalHystrixFilter 的 Hystrix 过滤器进行处理，并将失败的请求转发到 /fallback/external 路径。同样，对于 /shipping/\*\* 路径的请求，将使用名为 shippingHystrixFilter 的 Hystrix 过滤器，并将失败的请求转发到 /fallback/shipping 路径。

Hystrix 将在我们的系统中提供两个主要功能：服务降级和服务熔断。

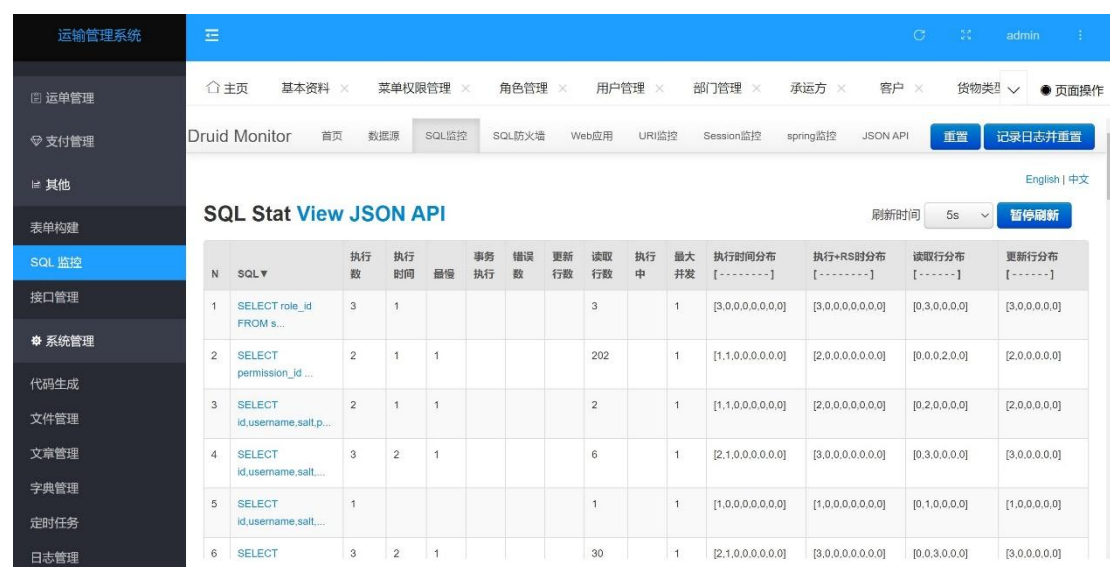
**服务降级：**当某个服务出现故障或超时，Hystrix 提供了降级机制，允许系统继续运行并提供有限的功能。在我们的物流系统中，为每个微服务定义了简单的回退规则。当主要功能无法正常执行时，将触发这些回退方法，并返回事先定义好的降级响应。

**服务熔断：**当 Hystrix 连续检测到服务故障时，熔断器将打开并阻止进一步的服务调用，直到服务恢复正常。防止故障的服务对系统产生过大的影响，并加快恢复时间。

我们采用自定义的 HystrixFilterFactory，通过继承 AbstractGatewayFilterFactory，并在 apply 方法中添加 Hystrix 过滤器逻辑。实现包装目标微服务的调用，并在调用失败时执行回退逻辑，返回预定义的降级响应。

## Druid Spring Boot Starter 集成

在我们的物流管理系统中，我们已经集成了 Druid Spring Boot Starter 来优化我们的数据库连接管理，并实现监控功能。



The screenshot shows the Druid Monitor web interface. The left sidebar contains navigation links for various system management tasks. The main content area displays the 'SQL Stat View JSON API' with a table of execution statistics. The table includes columns for N, SQL, execution count, execution time, error count, update count, read count, execution status, max open connections, execution time distribution, execution + RS distribution, read distribution, and update distribution. The table lists six SQL queries and their corresponding statistics.

N	SQL	执行数	执行时间	错误数	更新行数	读取行数	执行中	最大并发	执行时间分布	执行+RS分布	读取分布	更新分布
1	SELECT role_id FROM s...	3	1			3		1	[3,0,0,0,0,0,0]	[3,0,0,0,0,0,0]	[0,3,0,0,0,0]	[3,0,0,0,0,0]
2	SELECT permission_id ...	2	1	1		202		1	[1,1,0,0,0,0,0]	[2,0,0,0,0,0,0]	[0,0,0,2,0,0]	[2,0,0,0,0,0]
3	SELECT id,username,salt.p...	2	1	1		2		1	[1,1,0,0,0,0,0]	[2,0,0,0,0,0,0]	[0,2,0,0,0,0]	[2,0,0,0,0,0]
4	SELECT id,username,salt...	3	2	1		6		1	[2,1,0,0,0,0,0]	[3,0,0,0,0,0,0]	[0,3,0,0,0,0]	[3,0,0,0,0,0]
5	SELECT id,username,salt...	1				1		1	[1,0,0,0,0,0,0]	[1,0,0,0,0,0,0]	[0,1,0,0,0,0]	[1,0,0,0,0,0]
6	SELECT	3	2	1		30		1	[2,1,0,0,0,0,0]	[3,0,0,0,0,0,0]	[0,0,3,0,0,0]	[3,0,0,0,0,0]

**数据库连接管理：**Druid 是阿里巴巴的开源数据库连接池，由于其强大而高效的数据库连接管理能力，它被广泛使用。通过使用 Druid Spring Boot Starter，我们可以管理和维护一个数据库连接池，当需要时可以重复使用这些连接，从而减少初始化新连接的开销。

---

**监控功能：**除了提供高效的数据库连接管理之外，Druid 还提供了一个称为 Druid Stat 的统计模块，用于监控数据库访问性能。我们的物流管理系统已经启用了这个模块，可以实时收集数据库查询的详细信息，包括查询的执行时间、结果集的行数等等。所有这些信息都被汇集在一个 web 页面上，供我们进行查看和分析。

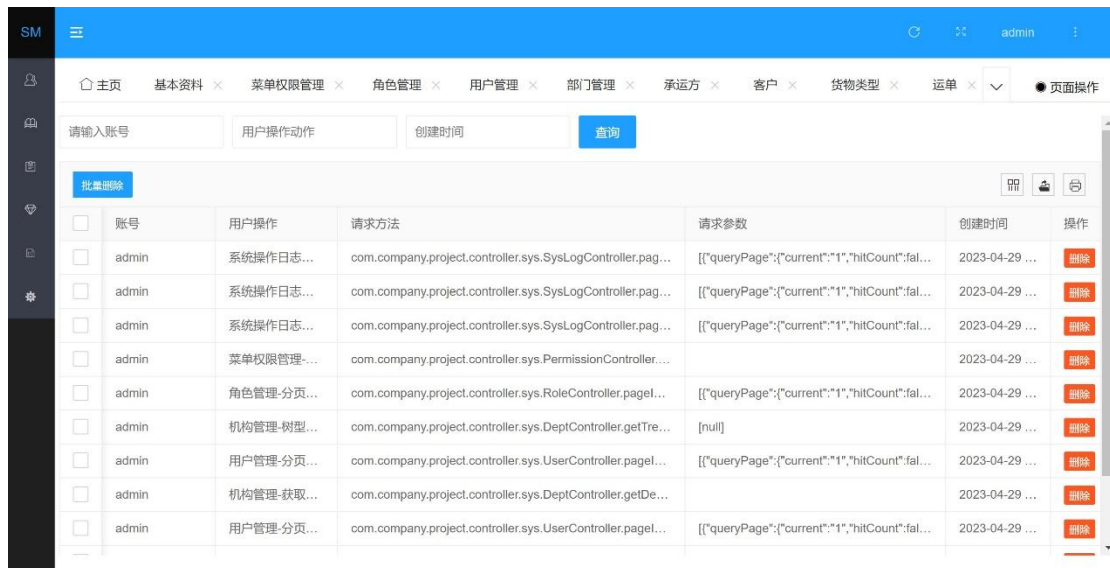
在我们的物流管理系统中，我们面临着高并发访问和数据读写负载均衡的需求。为了解决这些问题，我们决定采用多数据源的方式，同时选择使用 Druid 作为我们的数据库连接池，并通过 Spring AOP 进行多数据源的动态切换。使用 Druid Spring Boot Starter 进行数据库连接管理和监控，不仅可以提高数据库操作的性能，还可以帮助我们快速定位和解决可能存在的数据库相关问题。

**Spring AOP 动态数据源切换：**为了实现多数据源，我们决定采用 Spring AOP 对数据源进行动态切换。在我们的设计中，我们首先定义了一个自定义的@TargetDataSource 注解，用来标注使用哪个数据源。然后，我们创建一个自定义的数据源切换切面，它在@TargetDataSource 注解的方法执行前，将数据源切换到指定的数据源，在方法执行后，将数据源切换回默认数据源。

**使用 Druid 和 Spring AOP 实现多数据源的优点：**使用 Druid 连接池，我们不仅可以从连接池方面优化我们的系统，提高系统对数据库操作的响应速度；同时，Druid 内置的监控功能也可以方便我们对系统的性能进行监控和调优。通过 Spring AOP 动态切换数据源，我们可以根据具体的业务需求，选择最合适的数据源，从而提高系统的性能，同时也增加了系统的灵活性和可维护性。通过使用 Druid 连接池和 Spring AOP，我们的系统能更好地应对高并发访问和复杂的业务需求，同时也更易于监控和维护。

## 使用 Quartz 框架实现定时任务管理

在我们的物流管理系统中，存在着定期运行的后台任务的需求，例如定期生成报告、定期清理日志等。为了满足这个需求，我们选择使用 Quartz 框架来实现定时任务的创建、调度和管理。



<input type="checkbox"/>	账号	用户操作	请求方法	请求参数	创建时间	操作
<input type="checkbox"/>	admin	系统操作日志...	com.company.project.controller.sys.SysLogController.pag...	[{"queryPage":{"current":"1","hitCount":fal...	2023-04-29 ...	删除
<input type="checkbox"/>	admin	系统操作日志...	com.company.project.controller.sys.SysLogController.pag...	[{"queryPage":{"current":"1","hitCount":fal...	2023-04-29 ...	删除
<input type="checkbox"/>	admin	系统操作日志...	com.company.project.controller.sys.SysLogController.pag...	[{"queryPage":{"current":"1","hitCount":fal...	2023-04-29 ...	删除
<input type="checkbox"/>	admin	菜单权限管理...	com.company.project.controller.sys.PermissionController...		2023-04-29 ...	删除
<input type="checkbox"/>	admin	角色管理-分页...	com.company.project.controller.sys.RoleController.pagel...	[{"queryPage":{"current":"1","hitCount":fal...	2023-04-29 ...	删除
<input type="checkbox"/>	admin	机构管理-树型...	com.company.project.controller.sys.DeptController.getTre...	[null]	2023-04-29 ...	删除
<input type="checkbox"/>	admin	用户管理-分页...	com.company.project.controller.sys.UserController.pagel...	[{"queryPage":{"current":"1","hitCount":fal...	2023-04-29 ...	删除
<input type="checkbox"/>	admin	机构管理-获取...	com.company.project.controller.sys.DeptController.getDe...		2023-04-29 ...	删除
<input type="checkbox"/>	admin	用户管理-分页...	com.company.project.controller.sys.UserController.pagel...	[{"queryPage":{"current":"1","hitCount":fal...	2023-04-29 ...	删除

**Quartz 框架：**Quartz 是一个开源的强大的 Java 作业调度框架，能够满足各种复杂的调度需求。Quartz 提供了丰富的配置选项，可以设置作业的开始时间、结束时间、重复次数、重复间隔等。

**Quartz 作业创建：**在我们的系统中，我们将每一个需要定期执行的任务定义为一个 Quartz 作业(Job)。每个 Job 类都需要实现 org.quartz.Job 接口，并实现其 execute(JobExecutionContext context)方法。在 execute 方法中，我们定义了当调度器触发该作业时需要执行的具体任务。

**Quartz 作业调度：**为了调度 Job，我们需要创建一个 Trigger，用来定义 Job 的调度规则。Quartz 提供了 SimpleTrigger 和 CronTrigger 两种触发器。SimpleTrigger 用于设置作业的开始时间、结束时间、重复次数和重复间隔；CronTrigger 则可以使用 Cron 表达式来定义更复杂的调度规则。



**Quartz 作业管理：**为了方便管理所有的 Job 和 Trigger，我们使用 Quartz 提供的 Scheduler。Scheduler 负责管理和调度所有的 Job 和 Trigger。我们可以通过 Scheduler 的 API 来创建、删除、暂停和恢复 Job 和 Trigger。

**使用 Quartz 框架的优点：**通过使用 Quartz 框架，我们可以方便地创建、调度和管理所有的定时任务，大大提高了系统的可维护性和扩展性。同时，Quartz 的丰富的配置选项和强大的调度功能也可以满足我们各种复杂的调度需求。

## 使用 ZooKeeper 进行分布式协调和服务发现

为了实现系统的分布式协调和服务发现，我们选择使用 ZooKeeper 作为分布式协调服务。ZooKeeper 是一个开源的分布式协调系统，提供了高可用性、一致性和可靠性的服务。

```
2023-06-19 23:04:04,033 [myid:] - INFO [main:DigestAuthenticationProvider@47] - ACL digest algorithm is: SHA1
2023-06-19 23:04:04,034 [myid:] - INFO [main:DigestAuthenticationProvider@61] - zookeeper.DigestAuthenticationProvider.enabled = true
2023-06-19 23:04:04,044 [myid:] - INFO [main:FileTxnSnapLog@124] - zookeeper.snapshot.trust.empty : false
2023-06-19 23:04:04,082 [myid:] - INFO [main:ZookeeperBanner@42] -
2023-06-19 23:04:04,083 [myid:] - INFO [main:ZookeeperBanner@42] -
2023-06-19 23:04:04,087 [myid:] - INFO [main:ZookeeperBanner@42] -
2023-06-19 23:04:04,097 [myid:] - INFO [main:ZookeeperBanner@42] -
2023-06-19 23:04:04,098 [myid:] - INFO [main:ZookeeperBanner@42] -
2023-06-19 23:04:04,111 [myid:] - INFO [main:ZookeeperBanner@42] -
2023-06-19 23:04:04,112 [myid:] - INFO [main:ZookeeperBanner@42] -
2023-06-19 23:04:04,114 [myid:] - INFO [main:ZookeeperBanner@42] -
2023-06-19 23:04:04,119 [myid:] - INFO [main:ZookeeperBanner@42] -
2023-06-19 23:04:04,127 [myid:] - INFO [main:ZookeeperBanner@42] -
2023-06-19 23:04:04,166 [myid:] - INFO [main:Environment@98] - Server environment:zookeeper.version=3.7.1-a2fb57c55f8e59cdd76c34b357ad5181df1258d5, built on
2022-05-07 06:45 UTC
2023-06-19 23:04:04,167 [myid:] - INFO [main:Environment@98] - Server environment:host.name=LAPTOP-H3CBR1HD
2023-06-19 23:04:04,170 [myid:] - INFO [main:Environment@98] - Server environment:java.version=13.0.2
2023-06-19 23:04:04,177 [myid:] - INFO [main:Environment@98] - Server environment:java.vendor=Oracle Corporation
2023-06-19 23:04:04,179 [myid:] - INFO [main:Environment@98] - Server environment:java.home=D:\software\Java\jdk13
```

### ZooKeeper 的优势：

- 高可用性：ZooKeeper 采用分布式架构，通过在多个节点上复制数据来实现高可用性。即使某个节点发生故障，系统仍然可以正常运行。
- 一致性：ZooKeeper 使用 ZAB 协议（ZooKeeper Atomic Broadcast）来保证数据的一致性和顺序性。每个更新都会被顺序广播给所有节点，保证了数据的一致性。
- 可靠性：ZooKeeper 将数据存储在内存中，并定期将数据持久化到磁盘，以防止数据丢失。即使系统发生故障或重启，数据也可以被恢复。
- 灵活性：ZooKeeper 提供了丰富的 API 和功能，可以用于实现分布式锁、选举、配置管理和服务发现等场景。

### ZooKeeper 的主要功能：

- 
- 分布式协调：ZooKeeper 提供了分布式锁和顺序节点等机制，用于实现分布式系统的协调和同步。
  - 服务发现：通过在 ZooKeeper 中注册服务和监听节点变化，可以实现动态的服务发现和负载均衡。
  - 配置管理：ZooKeeper 可以用于集中管理系统的配置信息，当配置发生变化时，可以及时通知相关的服务进行更新。
  - 选举：ZooKeeper 提供了选举算法和机制，可以用于实现分布式系统中的领导者选举。

通过使用 ZooKeeper 进行分布式协调和服务发现，我们可以构建一个高可用、可靠和灵活的分布式系统。

## 使用 Kafka 进行消息传递

为了实现基础服务与外部信息之间的高效通信和解耦，我们选择使用 Kafka 作为消息传递系统。Kafka 是一个分布式流处理平台，可以实现高吞吐量、可扩展性和可靠性的消息传递。

### Kafka 的优势：

- 高吞吐量：Kafka 采用分布式、并行的方式处理消息，可以处理大量的消息并实现高吞吐量的数据传输。
- 可扩展性：Kafka 具有良好的可扩展性，可以通过添加更多的 Kafka 节点来增加系统的容量和处理能力。
- 持久性：Kafka 将消息持久化到磁盘，确保消息的可靠性和持久性存储，即使在系统故障或重启后仍能保留消息。

- 
- 解耦和松耦合：使用 Kafka 作为消息传递系统可以实现基础服务和外部信息之间的解耦和松耦合，提高系统的灵活性和可维护性。

#### Kafka 消息传递流程：

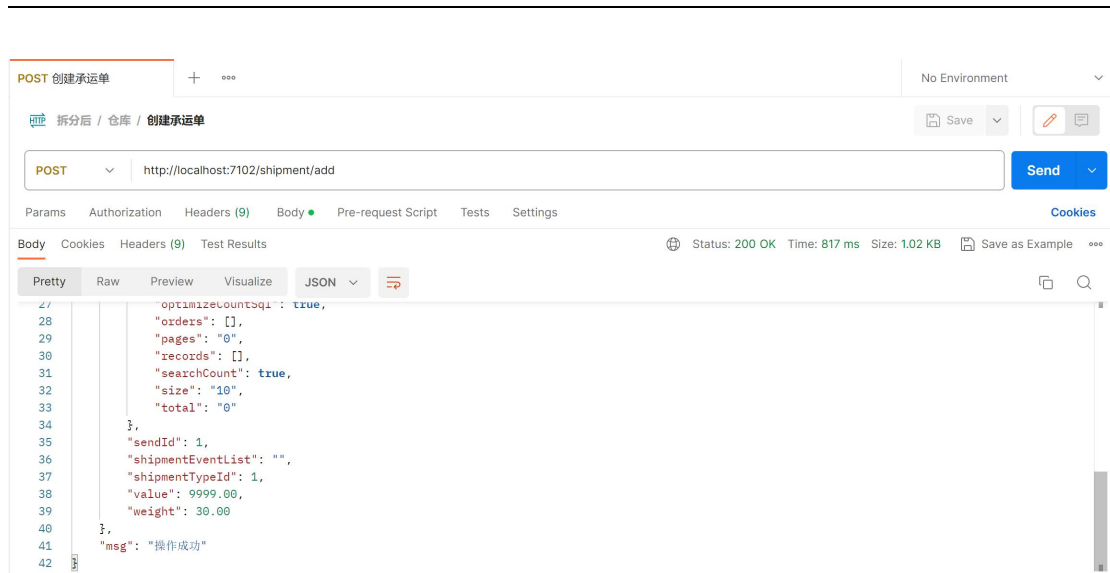
- 创建 Topic：在 Kafka 中创建一个或多个 Topic，用于存储消息。每个 Topic 可以包含多个分区，每个分区存储一部分消息。
- 消息生产者：基础服务将组织管理、物流信息等消息作为生产者，通过 Kafka Producer API 将消息发布到指定的 Topic 中。
- 消息消费者：外部信息服务作为消息消费者，通过 Kafka Consumer API 订阅指定的 Topic，接收和处理消息。
- 消息传递：Kafka 会将生产者发送的消息持久化到磁盘，并将其传递给订阅了该 Topic 的消费者。消费者可以按照自己的需求对消息进行处理和解析。
- 消息确认：消费者在处理完消息后可以发送确认消息给 Kafka，表示已成功处理该消息，Kafka 将删除该消息。

本系统在 shipping-management 服务中创建了 KafkaMessageSender 类，在 external-service 服务中创建了 KafkaMessageReceiver 类，shipping-management 在运单创建完成后会发送一个创建承运结算单据（carrier-billing-events）的消息，external-service 接收该消息并处理，实现了基于 zookeeper 和 kafka 的消息传递功能。

启动 external-service 后，kafka 显示新增 1 个消费者：

```
[2023-06-19 23:38:30.397] INFO [GroupCoordinator 0]: Dynamic Member with unknown member id joins group payment in Empty state. Created a new member id consumer-2-c095fd02-fa8f-4da2-89eb-866f3399e05d for this member and add to the group. (kafka.coordinator.group.GroupCoordinator)
[2023-06-19 23:38:30.408] INFO [GroupCoordinator 0]: Preparing to rebalance group payment in state PreparingRebalance with old generation 0 (__consumer_offsets-38) (reason: Adding new member consumer-2-c095fd02-fa8f-4da2-89eb-866f3399e05d with group instance id None; client reason: not provided) (kafka.coordinator.group.GroupCoordinator)
[2023-06-19 23:38:30.425] INFO [GroupCoordinator 0]: Stabilized group payment generation 1 (__consumer_offsets-38) with 1 members (kafka.coordinator.group.GroupCoordinator)
[2023-06-19 23:38:30.443] INFO [GroupCoordinator 0]: Assignment received from leader consumer-2-c095fd02-fa8f-4da2-89eb-866f3399e05d for group payment for generation 1. The group has 1 members, 0 of which are static. (kafka.coordinator.group.GroupCoordinator)
```

调用创建运单接口：



数据库新增一条待支付承运单信息：

9	1	30 (Null)	150.00	0 2023-06-19 23:44:19	(Null)
---	---	-----------	--------	-----------------------	--------

说明通过 kafka 成功实现消息传递。

### 3. 数据库设计

本系统中需要保存的数据包括用户信息、货物信息、订单信息等。其中承运信息表、客户信息表、运单表的数据库设计如下所示：

#### 3.1 承运信息表

字段名	数据类型	注释
id	int	承运人 id
name	varchar(100)	公司名
contact_name	varchar(50)	联系人姓名
email	varchar(100)	邮箱
phone	varchar(20)	电话

#### 3.2 客户信息表

字段名	数据类型	注释
id	int	客户 id

字段名	数据类型	注释
name	varchar(100)	姓名
email	varchar(100)	邮箱
phone	varchar(20)	手机号
company_name	varchar(100)	公司名
address	varchar(200)	地址
city	varchar(50)	城市
country	varchar(50)	国家
postal_code	varchar(20)	邮编

### 3.3 运单信息表

字段名	数据类型	注释
id	int	运单 id
customer_id	int	客户 id
carrier_id	int	承运方 id
send_id	int	发货人 id
shipment_type_id	int	物流类型
origin_address	varchar(255)	发货详细地址
origin_city	varchar(255)	发货地
origin_country	varchar(255)	发货地国家
origin_postal_code	varchar(255)	发货地邮编
destination_address	varchar(255)	收货详细地址
destination_city	varchar(255)	收货地
destination_country	varchar(255)	收货地国家

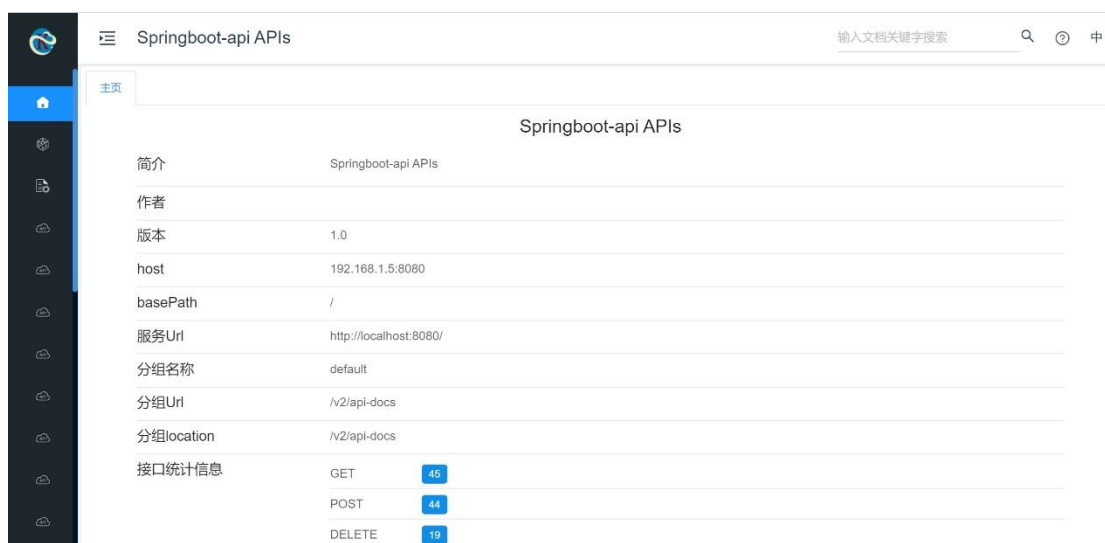
字段名	数据类型	注释
destination_postal_code	varchar(255)	收货地邮编
des	varchar(255)	备注
goods_id	int	物资类型
num	int	物资数量
weight	decimal(10,2)	总重量
value	decimal(10,2)	物资价值
order_status	varchar(255)	运单状态
pickup_date	datetime	发货日期
delivery_date	datetime	收货日期

## 4. 接口设计

本系统主要采用 RESTful API 进行通信，共包含两类接口，分别用于系统管理（sys）与运单管理（shipping），基于增删改查实现一系列系统管理的基础功能。

### 使用 OpenAPI 文档

系统集成 knife4j 生成接口文档，启动项目后可访问 <http://localhost:8080/doc.html> 进行查看。



OpenAPI 是一种定义 API 的规范，能够提供 API 的整体视图，包括：API 的所有端点 (endpoint)，每个端点的输入/输出，认证方法，联系信息等等。开发团队和 API 的消费者都可以方便地查看和理解 API 的设计和函数。

## 5. 系统测试

本系统目前完成了对运单管理服务、财务管理服务的单元测试，采用 JUnit+mockito+Jacoco 实现，测试用例如下表所示：

测试用例编号	Shipment -01	用例类型	服务实现层
用例目的	测试统计承运总金额的功能		
测试步骤			
1. 创建一个 Shipment 对象，并设置其各个属性的值。			
2. 调用 shipmentService.saveShipment() 方法，将 Shipment 对象保存到数据库中，并获取保存后的 Shipment 对象。			
3. 验证保存后的 Shipment 对象是否符合预期，包括其各个属性的值是否正确。			
4. 使用 JUnit 的 Assert 断言方法，验证保存后的 Shipment 对象是否符合预期，包括其 id、originAddress、destinationAddress、weight、value、pickupDate、orderStatus 等属性的值是否正确。			
代码覆盖分析			
<pre>34 public Shipment saveShipment(Shipment shipment) { 35     LambdaQueryWrapper&lt;Customer&gt; q1 = Wrappers.lambdaQuery(); 36     q1.eq(Customer::getId, shipment.getSendId()); 37     Customer send = customerMapper.selectOne(q1); 38     shipment.setOriginAddress(send.getAddress()); 39     shipment.setOriginCountry(send.getCountry()); 40     shipment.setOriginCity(send.getCity()); 41     shipment.setOriginPostalCode(send.getPostalCode()); 42     LambdaQueryWrapper&lt;Customer&gt; q2 = Wrappers.lambdaQuery(); 43     q2.eq(Customer::getId, shipment.getCustomerId()); 44     Customer customer = customerMapper.selectOne(q2); 45     shipment.setDestinationAddress(customer.getAddress()); 46     shipment.setDestinationCountry(customer.getCountry()); 47     shipment.setDestinationCity(customer.getCity()); 48     shipment.setDestinationPostalCode(customer.getPostalCode()); 49     LambdaQueryWrapper&lt;Goods&gt; q3 = Wrappers.lambdaQuery(); 50     q3.eq(Goods::getId, shipment.getGoodsId()); 51     Goods goods = goodsMapper.selectOne(q3); 52     shipment.setWeight(goods.getWeight().multiply(shipment.getNum())); 53     shipment.setValue(goods.getPrice().multiply(shipment.getNum())); 54     shipment.setPickupDate(new Date()); 55     shipment.setOrderStatus(0); 56     shipmentMapper.insert(shipment); 57     return shipment; 58 }</pre>			

测试用例编号	CarrierBilling-01	用例类型	服务实现层
用例目的	测试根据运单创建承运结算记录的功能		
测试步骤			
<div>1. 创建一个 Shipment 对象，并设置其属性值。</div> <div>2. 创建一个 ShipmentType 对象，并设置其属性值。</div> <div>3. 使用 Mockito 框架模拟 shipmentTypeMapper 的 selectOne 方法，使其返回上一步中创建的 ShipmentType 对象。</div> <div>4. 使用 Mockito 框架模拟 carrierBillingMapper 的 insert 方法，使其返回 1，表示插入数据成功。</div> <div>5. 调用 carrierBillingService 的 saveCarrierBilling 方法，并传入第 1 步中创建的 Shipment 对象作为参数。</div> <div>6. 使用断言方法 assertTrue 判断 saveCarrierBilling 方法返回值是否为 true。</div>			
代码覆盖分析			
<pre>31 public boolean saveCarrierBilling(Shipment shipment) { 32     CarrierBilling carrierBilling = new CarrierBilling(); 33     carrierBilling.setCarrierId(shipment.getCarrierId()); 34     carrierBilling.setOrderId(shipment.getId()); 35     carrierBilling.setCreateDate(new Date()); 36     LambdaQueryWrapper&lt;ShipmentRate&gt; queryWrapper = Wrappers.lambdaQuery(); 37     queryWrapper.eq(ShipmentRate::getShipmentTypeId, shipment.getShipmentTypeId()); 38     queryWrapper.eq(ShipmentRate::getRelatedId, shipment.getCarrierId()); 39     ShipmentRate shipmentRate = shipmentRateMapper.selectOne(queryWrapper); 40     carrierBilling.setFreightCharge(shipment.getWeight().multiply(shipmentRate.getPricePerKg())); 41     return SqlHelper.retBool(this.getBaseMapper().insert(carrierBilling)); 42 }</pre>			

测试用例编号	CarrierBilling -02	用例类型	服务实现层
用例目的	测试统计承运总金额的功能		
测试步骤			
<div>1. 定义一个 expected 变量，用于存储预期的结果值。</div> <div>2. 调用 carrierBillingService 对象的 carrierCount() 方法，并将返回值赋值给 actual 变量，用于存储实际的结果值。</div>			



3. 使用 `Assert.assertEquals()` 方法, 将预期结果值和实际结果值进行比较。如果两个值相等, 则测试通过, 否则测试失败。

#### 代码覆盖分析

```
public Double carrierCount() {  
    LambdaQueryWrapper<CarrierBilling> queryWrapper = Wrappers.lambdaQuery();  
    queryWrapper.eq(CarrierBilling::getState, 1);  
    List<CarrierBilling> list = this.baseMapper.selectList(queryWrapper);  
    Double total = 0.0;  
    for(CarrierBilling c:list) {  
        total += Double.parseDouble(c.getFreightCharge().toString());  
    }  
    System.out.println(total);  
    return Double.parseDouble(total.toString());  
}
```

测试用例编号	CustomerBilling-01	用例类型	服务实现层
用例目的	测试根据运单创建邮费结算记录的功能		
测试步骤			
<div>1. 创建一个 Shipment 对象，并设置其属性值。</div> <div>2. 创建一个 ShipmentType 对象，并设置其属性值。</div> <div>3. 使用 Mockito 框架模拟 shipmentTypeMapper 的 selectOne 方法，使其返回上一步中创建的 ShipmentType 对象。</div> <div>4. 使用 Mockito 框架模拟 customerBillingMapper 的 insert 方法，使其返回 1，表示插入数据成功。</div> <div>5. 调用 customerBillingService 的 saveCustomerBilling 方法，并传入第 1 步中创建的 Shipment 对象作为参数。</div> <div>6. 使用断言方法 assertTrue 判断 saveCustomerBilling 方法返回值是否为 true。</div>			
代码覆盖分析			

```

25     public boolean saveCustomerBilling(Shipment shipment) {
26         CustomerBilling customerBilling = new CustomerBilling();
27         customerBilling.setSendId(shipment.getSendId());
28         customerBilling.setOrderId(shipment.getId());
29         customerBilling.setCreateDate(new Date());
30         LambdaQueryWrapper<ShipmentType> queryWrapper = Wrappers.lambdaQuery();
31         queryWrapper.eq(ShipmentType::getId, shipment.getShipmentTypeId());
32         ShipmentType shipmentType = shipmentTypeMapper.selectOne(queryWrapper);
33         customerBilling.setPaymentAmount(shipmentType.getPrice().multiply(shipment.getWeight()));
34         return SqlHelper.retBool(this.getBaseMapper().insert(customerBilling));
35     }

```

测试用例编号	CustomerBilling-02	用例类型	服务实现层
用例目的	测试统计邮费总金额的功能		
测试步骤			
<div>1. 定义一个 expected 变量，用于存储预期的结果值。</div> <div>2. 调用 customerBillingService 对象的 customerCount() 方法，并将返回值赋值给 actual 变量，用于存储实际的结果值。</div> <div>3. 使用 Assert.assertEquals() 方法，将预期结果值和实际结果值进行比较。如果两个值相等，则测试通过，否则测试失败。</div>			
代码覆盖分析			
<pre>37     public Double customerCount() { 38         LambdaQueryWrapper&lt;CustomerBilling&gt; queryWrapper = Wrappers.lambdaQuery(); 39         queryWrapper.eq(CustomerBilling::getState, 1); 40         List&lt;CustomerBilling&gt; list = this.baseMapper.selectList(queryWrapper); 41         Double total = 0.0; 42         for (CustomerBilling c: list) { 43             total += Double.parseDouble(c.getPaymentAmount().toString()); 44         } 45         System.out.println(total); 46         return Double.parseDouble(total.toString()); 47     }</pre>			

## 6. 安全设计

在我们的物流管理系统中，我们使用 OAuth 2.0 协议来保护我们的资源和服务。OAuth 2.0 是一种授权框架，允许我们的应用为用户提供安全的授权服务，不需要分享密码。

---

## OAuth 2.0 在物流管理系统中的使用

当用户需要访问受保护的资源或服务时，他们将被引导到我们的授权服务器。授权服务器将要求用户登录并授权我们的应用访问他们的数据。用户在我们的服务器上成功授权后，将会得到一个授权码。

用户浏览器将使用这个授权码重定向回到我们的应用。我们的应用将使用这个授权码，再加上应用的 ID 和密钥，请求授权服务器一个访问令牌。授权服务器验证授权码和应用的凭据后，将会发放一个访问令牌给我们的应用。

我们的应用现在可以使用这个访问令牌来代表用户访问受保护的资源和服务。

## 微服务中的 OAuth 2.0 集成

在我们的微服务架构中，每个微服务 - CompanyProjectApplication、ExternalApplication、GatewayApplication、ShippingApplication，都将接入我们的 OAuth 2.0 授权服务。

在这个体系中，GatewayApplication 充当授权服务器的角色，为其他的微服务提供授权服务。其他的微服务在处理请求时，将需要检查请求的访问令牌，以验证用户的身份和权限。令牌的验证可能会在每个微服务内部处理，或者在 API 网关层处理。

通过在我们的微服务中集成 OAuth 2.0，我们可以有效地保护我们的资源和服务，只允许经过授权的用户访问。

## SSL 连接的创建

在我们的物流管理系统中，我们对数据的安全性非常重视。因此，我们选择使用 SSL 连接来加密客户端和服务端之间的通信，从而保证数据在传输过程中的安全性。

**SSL 连接的配置：**在我们的系统中，我们配置了 SSL 连接来确保客户端与服务端之间的数据传输过程是安全的。具体来说，我们生成了一个 SSL 证书，并在服务端进行了配置。

---

当客户端尝试与服务器建立连接时，服务器将 SSL 证书发送到客户端。客户端将使用证书来验证服务器的身份，并建立一个加密的连接。

**SSL 连接的优点：**使用 SSL 连接的最大优点是它可以提供数据传输的安全性。SSL 连接使用强大的加密算法来加密传输的数据，确保即使数据在传输过程中被拦截，也无法被解密和阅读。此外，SSL 证书还能验证服务器的身份，防止中间人攻击。

**SSL 连接在系统中的应用：**在我们的物流管理系统中，所有的客户端到服务器的连接都使用 SSL。这包括客户端应用程序与后端服务之间的连接，以及各个微服务之间的连接。这确保了系统中所有敏感信息的安全性，如用户的登录信息，物流的详细信息等。

通过使用 SSL 连接，我们的物流管理系统可以提供一个安全、可靠的网络环境，保护用户数据的安全，并防止各种网络攻击。

## 使用 Hibernate Validator 进行数据校验

在我们的物流管理系统中，数据的完整性和准确性是非常关键的。为了确保系统接收到的数据满足我们的业务规则，我们使用了 Hibernate Validator 进行数据校验。

Hibernate Validator 是 Bean Validation 的参考实现。它提供了一种基于注解的方式，可以将数据校验逻辑与业务代码分离，从而使得代码更易于维护，并且可以复用数据校验逻辑。

**Hibernate Validator 的配置：**在我们的系统中，我们通过在实体类的字段上添加 Hibernate Validator 提供的注解，如 @NotNull, @Min, @Max 等，来定义数据校验规则。当我们需要对一个实体类进行数据校验时，我们会创建一个 Validator 实例，并调用其 validate 方法进行数据校验。校验结果会被封装成 ConstraintViolation 集合返回，每一个 ConstraintViolation 对象都包含了一条违反的约束信息。

**Hibernate Validator 的优点：**使用 Hibernate Validator 的最大优点是它将数据校验逻辑与业务代码分离，使得数据校验逻辑更易于管理和复用。此外，Hibernate Validator 提供了

---

丰富的校验注解，可以满足我们大部分的数据校验需求。对于更复杂的数据校验需求，我们也可以通过创建自定义注解和对应的校验器来满足。

**Hibernate Validator 在系统中的应用：**在我们的物流管理系统中，我们广泛使用 Hibernate Validator 进行数据校验。比如，在处理用户注册请求时，我们会校验用户名是否为空，密码的长度是否符合要求等；在处理物流信息更新请求时，我们会校验物流信息是否完整，物流状态是否合法等。

Hibernate Validator 在保证我们系统数据完整性和准确性方面起到了非常重要的作用。

## 7. 性能优化

本系统采用 Redis 缓存 Token 信息，提高系统的性能和安全性。同时，系统还可以采用以下方式进一步优化性能：

- 使用 Nginx 等反向代理服务器，提高系统的并发处理能力；
- 使用 Redis 集群，提高系统的可用性和性能；
- 使用分布式文件系统（如 FastDFS）存储文件，提高文件上传和下载的性能；
- 对数据库进行优化，如建立索引、分表等，提高数据库的查询性能。

## 8. 部署方案

本系统可采用以下部署方案：

前端部署：前端页面可以部署到静态文件服务器上，如 Nginx；

后端部署：后端可以采用 Docker 容器化部署，方便快捷；

数据库部署：数据库可以采用 MySQL 或者 PostgreSQL，也可以采用 Docker 容器化部署。

---

## 使用 Docker 进行容器化部署

为了提高应用程序的可移植性和部署效率，我们选择使用 Docker 进行容器化部署。

Docker 是一种开源的容器化平台，可以将应用程序及其依赖项打包为一个独立的容器，从而实现跨平台和快速部署。

### Docker 的优势：

- 环境一致性：Docker 容器包含了应用程序及其依赖项，可以确保在不同环境中运行的一致性，避免了因环境差异导致的问题。
- 快速部署：Docker 容器可以在几秒钟内启动，大大缩短了应用程序的部署时间，提高了开发和运维效率。
- 资源隔离：每个 Docker 容器都运行在独立的隔离环境中，互不干扰，确保了应用程序之间的隔离性和安全性。
- 可扩展性：Docker 容器可以根据需求进行水平扩展，通过简单的命令即可启动多个相同的容器来处理更大的负载。

**Docker 容器化部署流程：**我们将应用程序及其依赖项打包为 Docker 镜像，并通过 Docker 容器来运行应用程序。以下是 Docker 容器化部署的主要步骤：

- 编写 Dockerfile：创建一个 Dockerfile，定义了如何构建 Docker 镜像。在 Dockerfile 中，我们指定了基础镜像、添加依赖项、设置环境变量和启动命令等。
- 构建 Docker 镜像：使用 Docker 命令根据 Dockerfile 构建 Docker 镜像。通过执行 `docker build` 命令，Docker 会根据 Dockerfile 的定义自动构建镜像，并将其保存到本地镜像仓库。

- 
- 推送 Docker 镜像：将构建好的 Docker 镜像推送到远程镜像仓库，以便在其他环境中使用。可以使用 Docker 命令或者容器注册中心（如 Docker Hub、阿里云容器镜像服务等）来进行镜像推送。
  - 部署 Docker 容器：在目标环境中使用 Docker 命令根据镜像启动 Docker 容器。通过执行 `docker run` 命令，Docker 会根据镜像创建一个容器，并运行应用程序。

## 9. 总结

在本物流管理系统的四次迭代中，通过集成 Spring Cloud Alibaba、Nacos、Gateway、Hystrix、Druid、Spring AOP、Quartz、OAuth 2.0 和 Hibernate Validator 等组件，初步实现了微服务系统的注册中心、配置中心、网关、熔断器、数据库连接池、数据源切换、定时任务管理、安全身份验证和数据校验等功能。

在第四阶段，通过使用 Zookeeper 和 Kafka，我们实现了物流服务作为生产者向外部服务发送消息的功能。这为不同服务之间的异步通信提供了解决方案，提高了系统的可扩展性和灵活性。我们还编写了 Dockerfile，为系统的部署提供了便利。虽然目前还没有将系统部署到云端，但我们的准备工作已经完成，为将来的部署奠定了基础。

通过这四次迭代，本系统初步实现了用户需求并验证了系统的可行性，意识到项目开发是一个持续学习和不断改进的过程，每个阶段都有新的挑战 and 机会。我们不仅提升了对现代化 JAVAEE 技术栈和架构的理解和应用能力，还加深了团队合作和解决问题的能力。期待在以后的项目中能够继续学习和成长，为用户提供更好的服务和解决方案。