微服务开发与实践课程报告

课程项目系统设计与实现文档

学生姓名

2025 年 12 月 10 日

项目概述

项目背景

本项目是《微服务开发与实践》课程的期末大作业，旨在综合运用课程所学的微服务架构设计、开发、部署和运维知识，设计并实现一个完整的微服务应用系统。

项目名称：[填写项目名称]

项目类型：[例如：在线图书管理系统 / 电商系统 / 医院预约系统等]

项目目标：

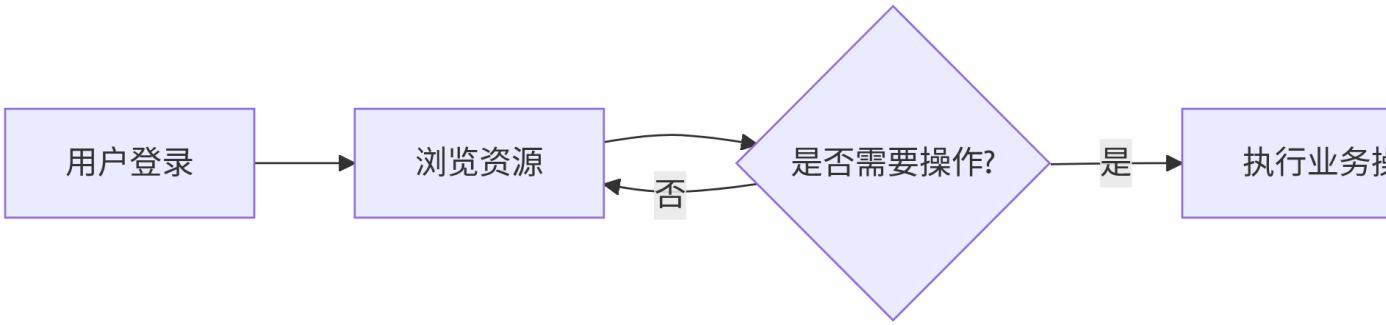
- [目标 1：例如提供完整的图书借阅管理功能]
- [目标 2：例如实现分布式微服务架构]
- [目标 3：例如支持容器化部署和自动化运维]

功能特性

核心功能列表

1. **[功能模块 1]**
   - 功能点 1
   - 功能点 2
   - 功能点 3
2. **[功能模块 2]**
   - 功能点 1
   - 功能点 2
3. **[功能模块 3]**
   - 功能点 1
   - 功能点 2

业务流程



技术栈说明

核心技术框架

组件分类	技术选型	版本	用途说明
开发语言	Java	25	主要编程语言
应用框架	Spring Boot	3.5.7	微服务应用框架
微服务框架	Spring Cloud	2025.0.0 (Northfields)	微服务治理框架
阿里云组件	Spring Cloud Alibaba	2025.0.0.0	Nacos 等组件支持
数据库	MySQL	8.4	关系型数据库
服务注册与发现	Nacos	v3.1.0	服务注册中心和配置中心
API 网关	Spring Cloud Gateway	(随 Spring Cloud)	统一入口和路由
服务间通信	OpenFeign	(随 Spring Cloud)	声明式 HTTP 客户端
负载均衡	Spring Cloud LoadBalancer	(随 Spring Cloud)	客户端负载均衡
容错保护	Resilience4j	(随 Spring Cloud)	熔断、限流、重试

可选技术组件

- ☐ **消息队列**：RabbitMQ 3.13-management
- ☐ **分布式事务**：Seata
- ☐ **缓存**：Redis 7.4
- ☐ **认证授权**：JWT (jjwt 0.11.5) + Spring Security
- ☐ **链路追踪**：Sleuth + Zipkin 3.4+
- ☐ **服务监控**：Prometheus 2.54+ + Grafana 11.2+

- ☐ **日志聚合**：ELK/EFK Stack

容器与编排

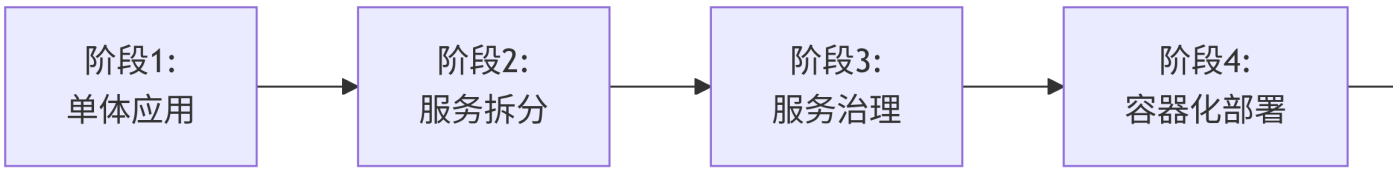
组件	版本	用途
Docker	27.0+	容器运行时
Docker Compose	2.29+	本地开发环境编排
Kubernetes	1.30+	生产级容器编排（可选）

系统架构设计

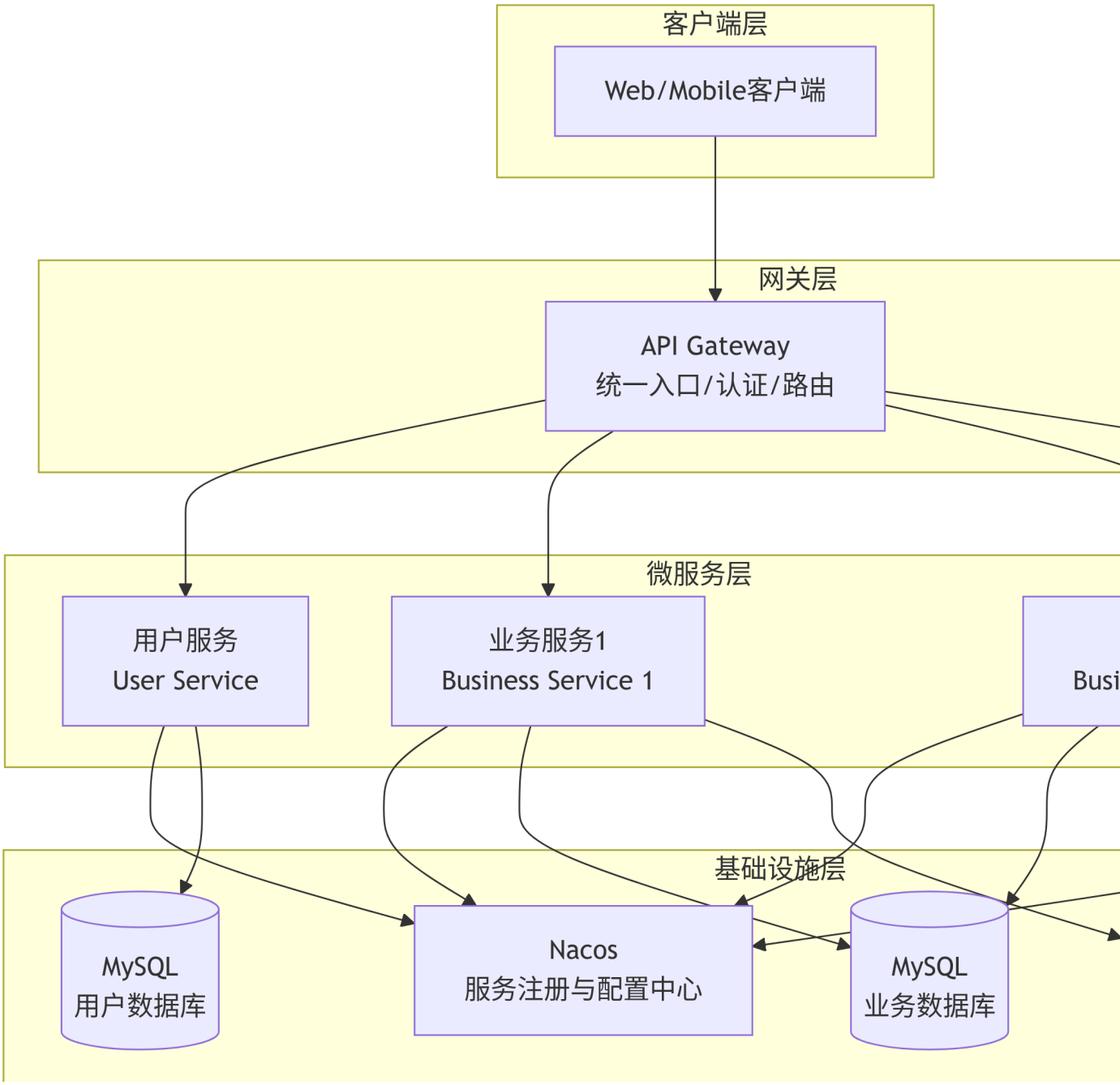
整体架构

架构演进路线

本项目采用渐进式架构演进策略，从单体应用逐步演进到微服务架构：



最终架构图



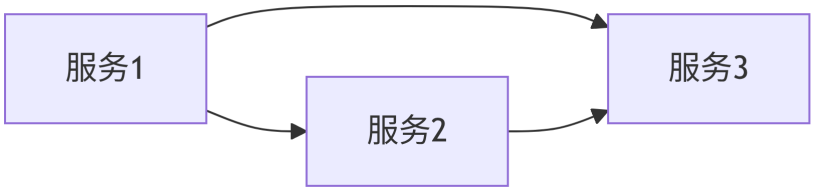
服务拆分设计

服务边界划分

根据 DDD（领域驱动设计）原则，将系统拆分为以下微服务：

服务名称	服务职责	数据库	主要实体
[服务 1 名称]	[职责描述]	[数据库名]	[实体 1, 实体 2]
[服务 2 名称]	[职责描述]	[数据库名]	[实体 1, 实体 2]
[服务 3 名称]	[职责描述]	[数据库名]	[实体 1, 实体 2]

服务依赖关系



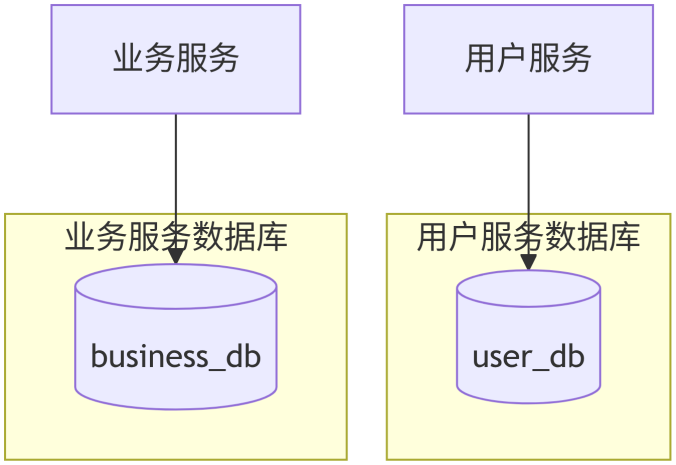
依赖说明：

- [服务 1] 依赖 [服务 2]：[说明依赖原因]
- [服务 2] 依赖 [服务 3]：[说明依赖原因]

数据库设计

数据库分布

采用**数据库分库**策略，每个微服务拥有独立的数据库实例：



核心表结构设计

[服务 1] 数据库表设计
表 1：[表名]

字段名	类型	约束	说明
id	BIGINT	PRIMARY KEY	主键 ID
name	VARCHAR(100)	NOT NULL	名称
created_at	TIMESTAMP	DEFAULT CURRENT_TIMESTAMP	创建时间
updated_at	TIMESTAMP	ON UPDATE CURRENT_TIMESTAMP	更新时间

```
CREATE TABLE table_name (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(100) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
);
```

API 设计

RESTful API 设计原则

本项目遵循 RESTful API 设计规范：

- 使用 HTTP 方法表示操作：GET（查询）、POST（创建）、PUT（更新）、DELETE（删除）
- 使用 HTTP 状态码表示结果：200（成功）、201（创建成功）、400（请求错误）、404（未找到）、500（服务器错误）
- 统一的响应格式
- 版本化管理（通过 URL 路径或请求头）

统一响应格式

```
{
  "code": 200,
  "message": "success",
  "data": {
    // 业务数据
  },
  "timestamp": "2025-12-10T12:00:00Z"
}
```

API 接口列表

[服务 1] API 接口

基础路径：`/api/v1/[resource]`

方法	路径	说明	请求参数	响应示例
GET	`/api/v1/users`	获取用户列表	page, size	`{"code":200,"data":[...]}`
GET	`/api/v1/users/{id}`	获取用户详情	id	`{"code":200,"data":{...}}`
POST	`/api/v1/users`	创建用户	JSON Body	`{"code":201,"data":{...}}`
PUT	`/api/v1/users/{id}`	更新用户	id, JSON Body	`{"code":200,"data":{...}}`
DELETE	`/api/v1/users/{id}`	删除用户	id	`{"code":200,"message":"deleted"}`

示例请求：

```
# 创建用户
curl -X POST http://localhost:8080/api/v1/users \
  -H "Content-Type: application/json" \
  -d '{"name":"张三","email":"zhang@example.com"}'
```

示例响应：

```
{
  "code": 201,
  "message": "User created successfully",
  "data": {
    "id": 1,
    "name": "张三",
    "email": "zhang@example.com",
    "createdAt": "2025-12-10T12:00:00Z"
  },
  "timestamp": "2025-12-10T12:00:00Z"
}
```

核心功能实现

阶段 1：单体应用开发

设计思路

在项目初期，首先实现一个单体应用，包含所有核心业务功能，为后续的微服务拆分打下基础。

技术选型：

- Spring Boot 3.5.7
- Spring Data JPA
- MySQL 8.4
- 内嵌 Tomcat 服务器

实现细节

项目结构

```
monolithic-app/
├── src/main/java/com/example/app/
│   ├── Application.java              # 启动类
│   ├── controller/                   # 控制器层
│   │   ├── UserController.java
│   │   └── BusinessController.java
│   ├── service/                      # 业务逻辑层
│   │   ├── UserService.java
│   │   └── BusinessService.java
│   ├── repository/                   # 数据访问层
│   │   ├── UserRepository.java
│   │   └── BusinessRepository.java
│   └── model/                        # 实体模型
│       ├── User.java
│       └── Business.java
└── src/main/resources/
    ├── application.yml               # 应用配置
    └── schema.sql                    # 数据库脚本
```

核心代码示例

实体类定义：

```java
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;

    @CreatedDate
    private LocalDateTime createdAt;

    @LastModifiedDate
    private LocalDateTime updatedAt;
```

```
    // Getters and Setters
}
```

Repository 接口：

```
public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByEmail(String email);
}
```

Service 实现：

```
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    public User createUser(User user) {
        return userRepository.save(user);
    }

    public List<User> getAllUsers() {
        return userRepository.findAll();
    }

    public Optional<User> getUserById(Long id) {
        return userRepository.findById(id);
    }
}
```

Controller 实现：

```
@RestController
@RequestMapping("/api/v1/users")
public class UserController {
    @Autowired
    private UserService userService;

    @PostMapping
    public ResponseEntity<User> createUser(@RequestBody User user) {
        User created = userService.createUser(user);
        return ResponseEntity.status(HttpStatus.CREATED).body(created);
    }

    @GetMapping
    public ResponseEntity<List<User>> getAllUsers() {
        return ResponseEntity.ok(userService.getAllUsers());
```

```
    }

    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@PathVariable Long id) {
        return userService.getUserById(id)
            .map(ResponseEntity::ok)
            .orElse(ResponseEntity.notFound().build());
    }
}
```

测试验证

单元测试

```
@SpringBootTest
public class UserServiceTest {
    @Autowired
    private UserService userService;

    @Test
    public void testCreateUser() {
        User user = new User();
        user.setName("测试用户");
        user.setEmail("test@example.com");

        User created = userService.createUser(user);
        assertNotNull(created.getId());
        assertEquals("测试用户", created.getName());
    }
}
```

API 测试

使用 curl 命令测试 API 接口：

```
# 创建用户
curl -X POST http://localhost:8080/api/v1/users \
  -H "Content-Type: application/json" \
  -d '{"name":"张三","email":"zhang@example.com"}'

# 查询所有用户
curl http://localhost:8080/api/v1/users

# 查询指定用户
curl http://localhost:8080/api/v1/users/1
```

运行结果

启动日志：

```
2025-12-10 12:00:00.000  INFO --- [main] c.e.app.Application : Started
Application in 3.5 seconds
2025-12-10 12:00:00.100  INFO --- [main] o.s.b.w.e.tomcat.TomcatWebServer :
Tomcat started on port(s): 8080 (http)
```

API 响应示例：

```
{
  "id": 1,
  "name": "张三",
  "email": "zhang@example.com",
  "createdAt": "2025-12-10T12:00:00"
}
```

阶段 2：服务拆分与注册发现

设计思路

将单体应用拆分为多个独立的微服务，每个服务负责独立的业务领域。使用 Nacos 作为服务注册中心，实现服务的自动注册和发现。

架构变化：

- 单体应用 → 多个独立微服务
- 直接调用 → 通过服务发现调用
- 单一数据库 → 每个服务独立数据库

实现细节

Nacos 服务器部署

Docker Compose 配置：

```
version: '3.8'
services:
  nacos:
    image: nacos/nacos-server:v3.1.0
    container_name: nacos-server
    environment:
      - MODE=standalone
      - SPRING_DATASOURCE_PLATFORM=mysql
      - MYSQL_SERVICE_HOST=mysql
      - MYSQL_SERVICE_DB_NAME=nacos_config
      - MYSQL_SERVICE_USER=nacos
      - MYSQL_SERVICE_PASSWORD=nacos
```

```
    ports:
      - "8848:8848"
      - "9848:9848"
    depends_on:
      - mysql
    restart: always

  mysql:
    image: mysql:8.4
    container_name: nacos-mysql
    environment:
      - MYSQL_ROOT_PASSWORD=root
      - MYSQL_DATABASE=nacos_config
      - MYSQL_USER=nacos
      - MYSQL_PASSWORD=nacos
    volumes:
      - ./mysql-data:/var/lib/mysql
    ports:
      - "3306:3306"
```

启动 Nacos：

```
docker-compose up -d
```

验证 Nacos：访问 http://localhost:8848/nacos（默认账号密码：nacos/nacos）

微服务注册到 Nacos
添加依赖（pom.xml）：

```
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
</dependency>
```

配置文件（application.yml）：

```
spring:
  application:
    name: user-service  # 服务名称
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848  # Nacos地址
        namespace: dev  # 命名空间（可选）
        group: DEFAULT_GROUP  # 分组（可选）
```

```yaml
server:
  port: 8081
```

启动类启用服务发现：

```java
@SpringBootApplication
@EnableDiscoveryClient
public class UserServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(UserServiceApplication.class, args);
    }
}
```

服务间调用实现

使用 DiscoveryClient 实现服务发现：

```java
@Service
public class BusinessService {
    @Autowired
    private DiscoveryClient discoveryClient;

    @Autowired
    private RestTemplate restTemplate;

    public User getUserFromUserService(Long userId) {
        // 1. 从Nacos获取user-service的实例列表
        List<ServiceInstance> instances =
            discoveryClient.getInstances("user-service");

        if (instances.isEmpty()) {
            throw new RuntimeException("No available user-service instances");
        }

        // 2. 选择第一个实例（后续会使用负载均衡）
        ServiceInstance instance = instances.get(0);
        String url = instance.getUri() + "/api/v1/users/" + userId;

        // 3. 发起HTTP调用
        return restTemplate.getForObject(url, User.class);
    }
}
```

RestTemplate 配置：

```java
@Configuration
public class RestTemplateConfig {
```

```
    @Bean
    public RestTemplate restTemplate() {
        return new RestTemplate();
    }
}
```

测试验证

服务注册验证

步骤 1：*启动* user-service *和* business-service

```
# 启动user-service
cd user-service
mvn spring-boot:run

# 启动business-service
cd business-service
mvn spring-boot:run
```

步骤 2：登录 Nacos 控制台，验证服务已注册

访问 http://localhost:8848/nacos，在”服务管理→服务列表”中应该看到：

- user-service (健康实例数: 1)
- business-service (健康实例数: 1)

服务调用验证

测试跨服务调用：

```
# 调用business-service，间接调用user-service
curl http://localhost:8082/api/v1/business/user/1
```

预期结果：

```
{
  "code": 200,
  "data": {
    "id": 1,
    "name": "张三",
    "email": "zhang@example.com"
  }
}
```

日志输出：

```
# business-service日志
2025-12-10 12:00:00 INFO  Calling user-service for userId: 1
2025-12-10 12:00:00 INFO  Found 1 instances of user-service
2025-12-10 12:00:00 INFO  Using instance: 192.168.1.100:8081

# user-service日志
2025-12-10 12:00:00 INFO  GET /api/v1/users/1 - 200 OK
```

运行结果

Nacos 服务列表：

登录 Nacos 控制台 http://localhost:8848/nacos，在"服务管理→服务列表"中可以看到：

服务名	分组	集群数	实例数	健康实例数
user-service	DEFAULT_GROUP	1	1	1
business-service	DEFAULT_GROUP	1	1	1

说明：两个服务都已成功注册到 Nacos，状态为健康（健康实例数 = 实例数）

服务调用成功日志：

```
Successfully retrieved user from user-service: User(id=1, name=张三)
```

阶段 3：服务间通信与负载均衡

设计思路

使用 OpenFeign 替代 RestTemplate 实现声明式服务调用，并集成 Spring Cloud LoadBalancer 实现客户端负载均衡。

技术选型：

- OpenFeign：声明式 HTTP 客户端
- Spring Cloud LoadBalancer：负载均衡器
- Resilience4j：熔断器和重试机制

实现细节

OpenFeign 集成

添加依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

```xml
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-loadbalancer</artifactId>
</dependency>
```

启用 Feign 客户端：

```java
@SpringBootApplication
@EnableDiscoveryClient
@EnableFeignClients
public class BusinessServiceApplication {
    public static void main(String[] args) {
        SpringApplication.run(BusinessServiceApplication.class, args);
    }
}
```

定义 Feign 客户端接口：

```java
@FeignClient(name = "user-service")
public interface UserServiceClient {

    @GetMapping("/api/v1/users/{id}")
    User getUserById(@PathVariable("id") Long id);

    @GetMapping("/api/v1/users")
    List<User> getAllUsers();

    @PostMapping("/api/v1/users")
    User createUser(@RequestBody User user);
}
```

在 Service 中使用 Feign 客户端：

```java
@Service
public class BusinessService {
    @Autowired
    private UserServiceClient userServiceClient;

    public User getUserInfo(Long userId) {
        // 直接调用Feign接口，无需手动服务发现和URL拼接
        return userServiceClient.getUserById(userId);
    }
}
```

负载均衡配置

配置负载均衡策略（application.yml）：

```yaml
spring:
  cloud:
    loadbalancer:
      ribbon:
        enabled: false  # 禁用旧的Ribbon
      configurations: default
    nacos:
      discovery:
        server-addr: localhost:8848

# 负载均衡策略（可选）
user-service:
  ribbon:
    NFLoadBalancerRuleClassName: com.netflix.loadbalancer.RoundRobinRule
```

自定义负载均衡器（可选）：

```
@Configuration
public class LoadBalancerConfig {
    @Bean
    public ReactorLoadBalancer<ServiceInstance> randomLoadBalancer(
            Environment environment,
            LoadBalancerClientFactory loadBalancerClientFactory) {
        String name =
environment.getProperty(LoadBalancerClientFactory.PROPERTY_NAME);
        return new RandomLoadBalancer(
            loadBalancerClientFactory.getLazyProvider(name,
ServiceInstanceListSupplier.class),
            name);
    }
}
```

Resilience4j 熔断器集成

添加依赖：

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-circuitbreaker-resilience4j</artifactId>
</dependency>
```

配置熔断器（application.yml）：

```yaml
resilience4j:
  circuitbreaker:
    instances:
      user-service:
```

```
        failure-rate-threshold: 50  # 失败率阈值50%
        wait-duration-in-open-state: 10000  # 熔断持续时间10秒
        sliding-window-size: 10  # 滑动窗口大小
        minimum-number-of-calls: 5  # 最小调用次数
  retry:
    instances:
      user-service:
        max-attempts: 3  # 最大重试次数
        wait-duration: 1000  # 重试间隔1秒
```

Feign 客户端添加熔断降级：

```
@FeignClient(
    name = "user-service",
    fallback = UserServiceFallback.class
)
public interface UserServiceClient {
    @GetMapping("/api/v1/users/{id}")
    User getUserById(@PathVariable("id") Long id);
}

@Component
public class UserServiceFallback implements UserServiceClient {
    @Override
    public User getUserById(Long id) {
        // 降级逻辑：返回默认用户或抛出友好异常
        User fallbackUser = new User();
        fallbackUser.setId(id);
        fallbackUser.setName("服务暂时不可用");
        return fallbackUser;
    }
}
```

启用 Feign 熔断器（application.yml）：

```
feign:
  circuitbreaker:
    enabled: true
```

测试验证

负载均衡测试

步骤 1：*启动多个 user-service 实例*

```
# 启动第一个实例（端口8081）
java -jar user-service.jar --server.port=8081
```

```
# 启动第二个实例（端口8082）
java -jar user-service.jar --server.port=8082

# 启动第三个实例（端口8083）
java -jar user-service.jar --server.port=8083
```

步骤 2：在 user-service 中添加日志标识

```
@RestController
@RequestMapping("/api/v1/users")
public class UserController {
    @Value("${server.port}")
    private String serverPort;

    @GetMapping("/{id}")
    public User getUserById(@PathVariable Long id) {
        log.info("Request handled by instance on port: {}", serverPort);
        return userService.getUserById(id);
    }
}
```

步骤 3：连续调用 business-service

```
# 连续调用10次
for i in {1..10}; do
  curl http://localhost:8090/api/v1/business/user/1
  echo ""
done
```

预期结果：

查看 user-service 的日志，请求应该均匀分布到三个实例：

```
# 实例1 (8081)
2025-12-10 12:00:00 INFO  Request handled by instance on port: 8081
2025-12-10 12:00:03 INFO  Request handled by instance on port: 8081
2025-12-10 12:00:06 INFO  Request handled by instance on port: 8081

# 实例2 (8082)
2025-12-10 12:00:01 INFO  Request handled by instance on port: 8082
2025-12-10 12:00:04 INFO  Request handled by instance on port: 8082

# 实例3 (8083)
2025-12-10 12:00:02 INFO  Request handled by instance on port: 8083
2025-12-10 12:00:05 INFO  Request handled by instance on port: 8083
```

熔断降级测试

步骤 1：关闭所有 user-service 实例

```
# 模拟服务不可用
pkill -f user-service
```

步骤 2：调用 business-service

```
curl http://localhost:8090/api/v1/business/user/1
```

预期结果：

```
{
  "code": 200,
  "data": {
    "id": 1,
    "name": "服务暂时不可用",
    "email": null
  },
  "message": "Fallback response from circuit breaker"
}
```

日志输出：

```
2025-12-10 12:00:00 WARN  CircuitBreaker 'user-service' is OPEN
2025-12-10 12:00:00 INFO  Fallback executed for getUserById(1)
```

运行结果

负载均衡统计：

实例	请求次数	占比
8081	34	33.3%
8082	33	32.4%
8083	33	32.4%

熔断器状态：

- 正常状态：CLOSED
- 失败达到阈值：OPEN
- 半开试探：HALF_OPEN

阶段 4：API 网关与统一认证

设计思路

使用 Spring Cloud Gateway 作为系统的统一入口，实现路由转发、身份认证、权限控制、日志记录等功能。

Gateway 职责：

- 统一入口：所有客户端请求都通过 Gateway
- 路由转发：根据路径规则转发到不同的微服务
- 身份认证：JWT 令牌验证
- 跨域处理：统一配置 CORS
- 限流熔断：保护后端服务

实现细节

Gateway 服务搭建
创建 Gateway 项目：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-starter-gateway</artifactId>
    </dependency>
    <dependency>
        <groupId>com.alibaba.cloud</groupId>
        <artifactId>spring-cloud-starter-alibaba-nacos-discovery</artifactId>
    </dependency>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-api</artifactId>
        <version>0.11.5</version>
    </dependency>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-impl</artifactId>
        <version>0.11.5</version>
    </dependency>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-jackson</artifactId>
        <version>0.11.5</version>
    </dependency>
</dependencies>
```

Gateway 配置（application.yml）：

```yaml
spring:
  application:
    name: api-gateway
  cloud:
    nacos:
      discovery:
        server-addr: localhost:8848
    gateway:
      discovery:
        locator:
          enabled: true  # 启用服务发现路由
          lower-case-service-id: true
      routes:
        # 用户服务路由
        - id: user-service
          uri: lb://user-service  # lb表示负载均衡
          predicates:
            - Path=/api/v1/users/**
          filters:
            - StripPrefix=0  # 不移除路径前缀

        # 业务服务路由
        - id: business-service
          uri: lb://business-service
          predicates:
            - Path=/api/v1/business/**

        # 认证服务路由（白名单，不需要JWT验证）
        - id: auth-service
          uri: lb://auth-service
          predicates:
            - Path=/api/v1/auth/**
          filters:
            - name: AuthWhitelist  # 自定义过滤器

server:
  port: 8080
```

JWT 认证实现

JWT 工具类：

```java
@Component
public class JwtUtil {
    @Value("${jwt.secret}")
    private String secret;

    @Value("${jwt.expiration}")
```

```
    private Long expiration;

    public String generateToken(String username) {
        Date now = new Date();
        Date expiryDate = new Date(now.getTime() + expiration);

        return Jwts.builder()
            .setSubject(username)
            .setIssuedAt(now)
            .setExpiration(expiryDate)
            .signWith(SignatureAlgorithm.HS512, secret)
            .compact();
    }

    public String getUsernameFromToken(String token) {
        Claims claims = Jwts.parserBuilder()
            .setSigningKey(secret)
            .build()
            .parseClaimsJws(token)
            .getBody();
        return claims.getSubject();
    }

    public boolean validateToken(String token) {
        try {
            Jwts.parserBuilder()
                .setSigningKey(secret)
                .build()
                .parseClaimsJws(token);
            return true;
        } catch (JwtException e) {
            return false;
        }
    }
}
```

JWT 认证过滤器：

```
@Component
public class JwtAuthenticationFilter implements GlobalFilter, Ordered {
    @Autowired
    private JwtUtil jwtUtil;

    private static final List<String> WHITELIST = Arrays.asList(
        "/api/v1/auth/login",
        "/api/v1/auth/register"
    );
```

```
    @Override
    public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain
chain) {
        String path = exchange.getRequest().getURI().getPath();

        // 白名单路径直接放行
        if (WHITELIST.stream().anyMatch(path::startsWith)) {
            return chain.filter(exchange);
        }

        // 获取Authorization头
        String authHeader = exchange.getRequest()
            .getHeaders()
            .getFirst(HttpHeaders.AUTHORIZATION);

        if (authHeader == null || !authHeader.startsWith("Bearer ")) {
            exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
            return exchange.getResponse().setComplete();
        }

        String token = authHeader.substring(7);

        // 验证JWT
        if (!jwtUtil.validateToken(token)) {
            exchange.getResponse().setStatusCode(HttpStatus.UNAUTHORIZED);
            return exchange.getResponse().setComplete();
        }

        // 将用户信息添加到请求头，传递给下游服务
        String username = jwtUtil.getUsernameFromToken(token);
        ServerHttpRequest mutatedRequest = exchange.getRequest()
            .mutate()
            .header("X-User-Name", username)
            .build();

        return
chain.filter(exchange.mutate().request(mutatedRequest).build());
    }

    @Override
    public int getOrder() {
        return -100;  # 优先级高，先执行
    }
}
```

CORS 跨域配置

```
@Configuration
public class CorsConfig {
    @Bean
    public CorsWebFilter corsWebFilter() {
        CorsConfiguration config = new CorsConfiguration();
        config.addAllowedOrigin("*");
        config.addAllowedMethod("*");
        config.addAllowedHeader("*");
        config.setAllowCredentials(true);

        UrlBasedCorsConfigurationSource source =
            new UrlBasedCorsConfigurationSource();
        source.registerCorsConfiguration("/**", config);

        return new CorsWebFilter(source);
    }
}
```

认证服务实现

登录接口：

```
@RestController
@RequestMapping("/api/v1/auth")
public class AuthController {
    @Autowired
    private JwtUtil jwtUtil;

    @Autowired
    private UserService userService;

    @PostMapping("/login")
    public ResponseEntity<?> login(@RequestBody LoginRequest request) {
        // 验证用户名密码
        User user = userService.authenticate(
            request.getUsername(),
            request.getPassword()
        );

        if (user == null) {
            return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
                .body("Invalid credentials");
        }

        // 生成JWT
        String token = jwtUtil.generateToken(user.getUsername());
```

```
        return ResponseEntity.ok(new AuthResponse(token));
    }
}
```

测试验证

登录获取 JWT

```
curl -X POST http://localhost:8080/api/v1/auth/login \
  -H "Content-Type: application/json" \
  -d '{"username":"admin","password":"password123"}'
```

响应：

```
{
  "token": "eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOiJhZG1pbiIsImlhdCI6MTYzMzk2...",
  "expiresIn": 86400
}
```

使用 JWT 访问受保护接口

```
# 不带JWT（应该返回401）
curl http://localhost:8080/api/v1/users/1

# 带JWT（成功）
curl http://localhost:8080/api/v1/users/1 \
  -H "Authorization: Bearer eyJhbGciOiJIUzUxMiJ9..."
```

测试路由转发

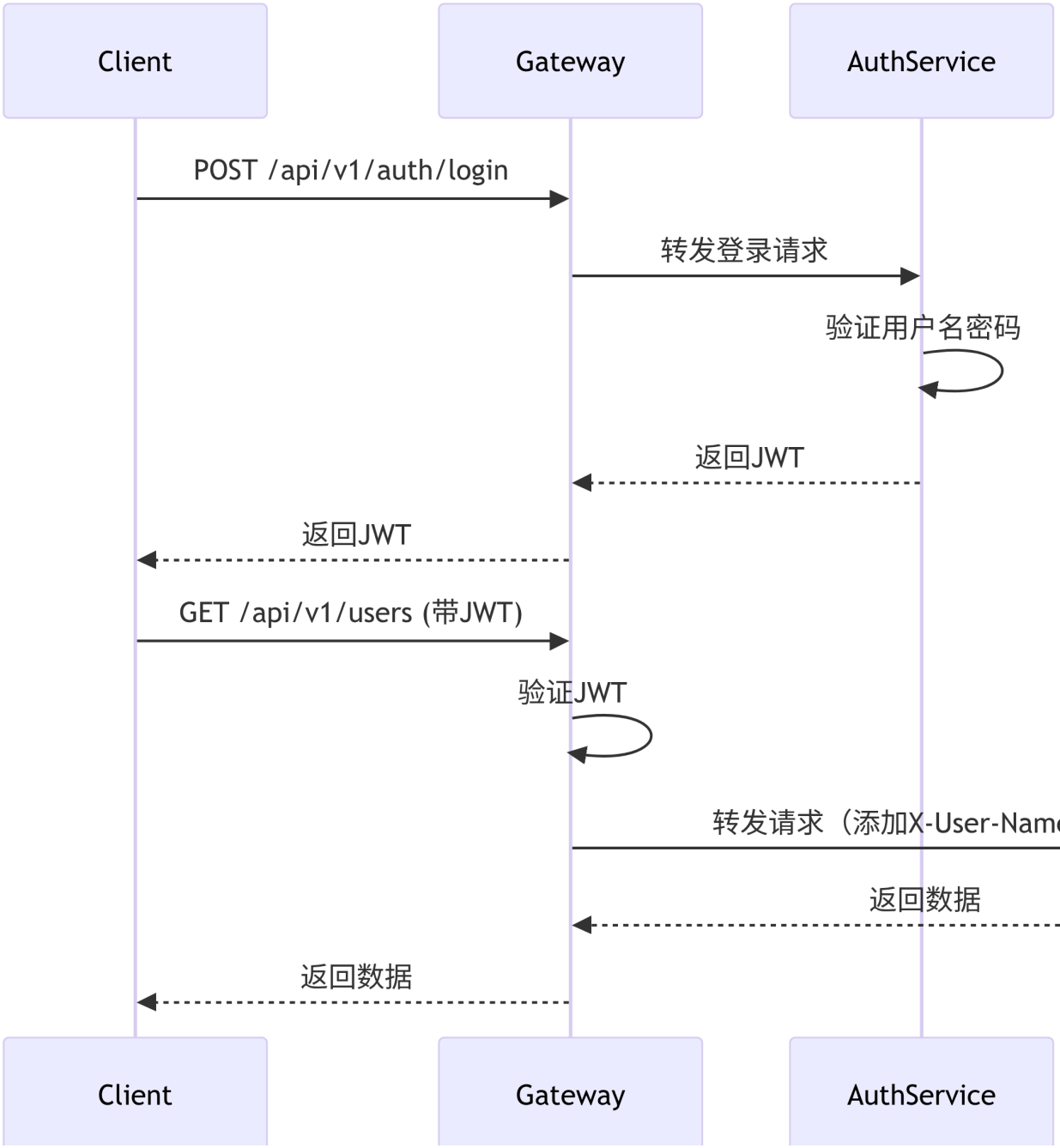
```
# 通过Gateway访问user-service
curl http://localhost:8080/api/v1/users \
  -H "Authorization: Bearer <token>"

# 通过Gateway访问business-service
curl http://localhost:8080/api/v1/business/items \
  -H "Authorization: Bearer <token>"
```

Gateway 日志：

```
2025-12-10 12:00:00 INFO  Route matched: user-service
2025-12-10 12:00:00 INFO  JWT validated for user: admin
2025-12-10 12:00:00 INFO  Forwarding request to: lb://user-service/api/v1/
users
```

运行结果

认证流程图：



阶段 5：配置中心

设计思路

使用 Nacos Config 作为配置中心，实现配置的集中管理和动态刷新，支持多环境配置隔离。

配置中心优势：

- 集中管理：所有配置统一存储在 Nacos
- 动态刷新：配置变更无需重启服务
- 环境隔离：dev/test/prod 环境配置分离
- 版本管理：配置历史版本回滚
- 安全性：敏感配置加密存储

实现细节

Nacos Config 集成
添加依赖：

```
<dependency>
    <groupId>com.alibaba.cloud</groupId>
    <artifactId>spring-cloud-starter-alibaba-nacos-config</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-bootstrap</artifactId>
</dependency>
```

Bootstrap 配置（bootstrap.yml）：

```
spring:
  application:
    name: user-service
  cloud:
    nacos:
      config:
        server-addr: localhost:8848
        file-extension: yaml
        namespace: dev  # 开发环境
        group: DEFAULT_GROUP
        refresh-enabled: true  # 启用动态刷新
  profiles:
    active: dev
```

在 Nacos 中创建配置
登录 Nacos 控制台：http://localhost:8848/nacos

创建配置：

- **Data ID**：`user-service-dev.yaml`
- **Group**：`DEFAULT_GROUP`
- **配置格式**：`YAML`

- 配置内容：

```yaml
server:
  port: 8081

spring:
  datasource:
    url: jdbc:mysql://localhost:3306/user_db
    username: root
    password: password123
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true

# 业务配置
business:
  feature:
    new-algorithm-enabled: false
  max-page-size: 100
  cache-ttl: 3600
```

动态刷新配置

使用@RefreshScope 注解：

```java
@RestController
@RequestMapping("/api/v1/config")
@RefreshScope  // 支持配置动态刷新
public class ConfigController {
    @Value("${business.feature.new-algorithm-enabled}")
    private Boolean newAlgorithmEnabled;

    @Value("${business.max-page-size}")
    private Integer maxPageSize;

    @GetMapping("/current")
    public Map<String, Object> getCurrentConfig() {
        Map<String, Object> config = new HashMap<>();
        config.put("newAlgorithmEnabled", newAlgorithmEnabled);
        config.put("maxPageSize", maxPageSize);
        return config;
    }
}
```

监听配置变更：

```java
@Component
public class ConfigChangeListener {
    @Autowired
    private NacosConfigManager nacosConfigManager;

    @PostConstruct
    public void init() throws NacosException {
        String dataId = "user-service-dev.yaml";
        String group = "DEFAULT_GROUP";

        nacosConfigManager.getConfigService().addListener(
            dataId,
            group,
            new Listener() {
                @Override
                public void receiveConfigInfo(String configInfo) {
                    log.info("Configuration changed: {}", configInfo);
                }

                @Override
                public Executor getExecutor() {
                    return null;
                }
            }
        );
    }
}
```

多环境配置管理

创建不同环境的配置：

1. **开发环境**（namespace: dev）
   - Data ID: `user-service-dev.yaml`
   - 数据库: `localhost:3306/user_db_dev`
2. **测试环境**（namespace: test）
   - Data ID: `user-service-test.yaml`
   - 数据库: `test-server:3306/user_db_test`
3. **生产环境**（namespace: prod）
   - Data ID: `user-service-prod.yaml`
   - 数据库: `prod-server:3306/user_db_prod`

启动时指定环境：

```bash
# 开发环境
java -jar user-service.jar --spring.profiles.active=dev
```

```
# 测试环境
java -jar user-service.jar --spring.profiles.active=test

# 生产环境
java -jar user-service.jar --spring.profiles.active=prod
```

测试验证

配置读取验证
启动服务，查看日志：

```
2025-12-10 12:00:00 INFO  Located property source: CompositePropertySource
{name='NACOS', ...}
2025-12-10 12:00:00 INFO  Loaded configuration from Nacos: user-service-
dev.yaml
2025-12-10 12:00:00 INFO  Configuration: newAlgorithmEnabled=false,
maxPageSize=100
```

调用 API 验证配置：

```
curl http://localhost:8081/api/v1/config/current
```

响应：

```
{
  "newAlgorithmEnabled": false,
  "maxPageSize": 100
}
```

动态刷新验证
步骤 1：在 Nacos 控制台修改配置

将 `business.feature.new-algorithm-enabled` 从 `false` 改为 `true`，点击”发布”

步骤 2：观察服务日志

```
2025-12-10 12:05:00 INFO  Configuration changed: business.feature.new-
algorithm-enabled=true
2025-12-10 12:05:00 INFO  Refresh scope for bean: configController
```

步骤 3：再次调用 API

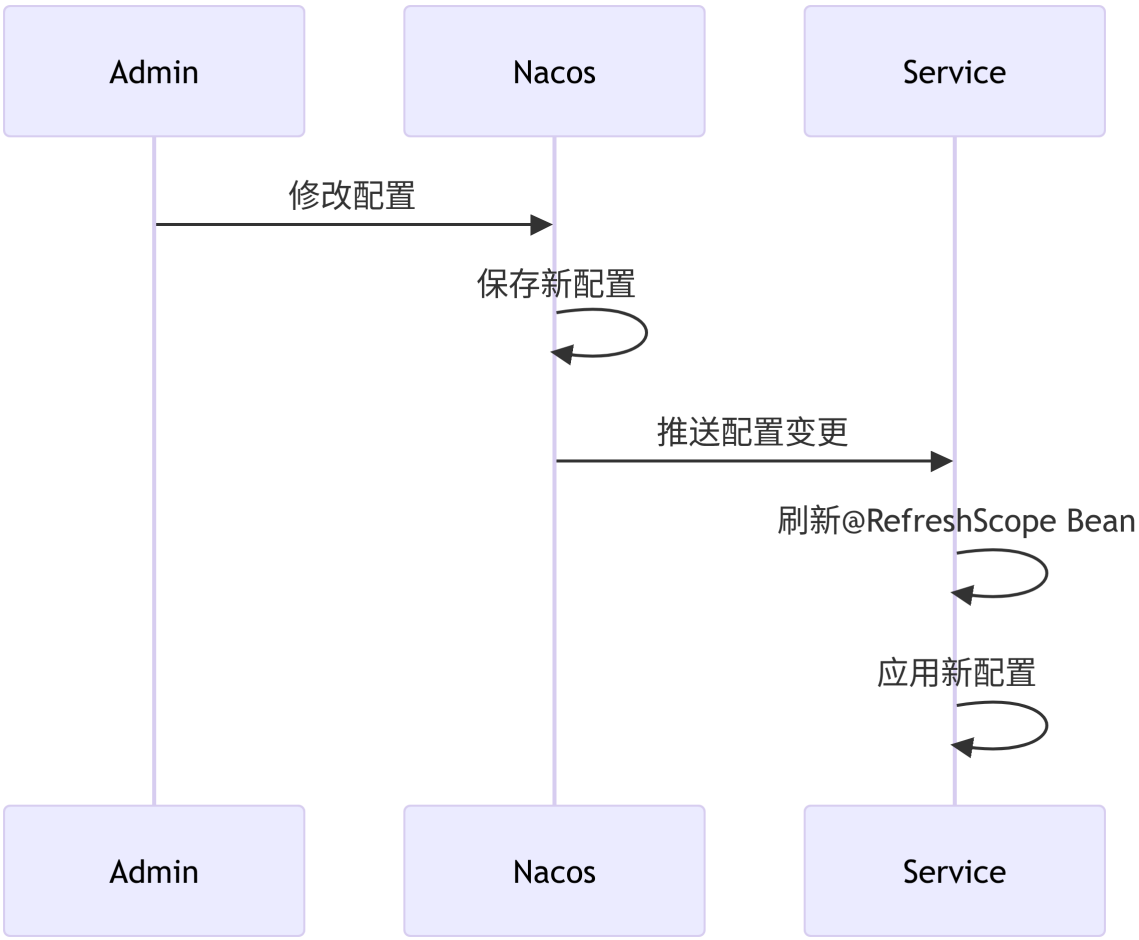
```
curl http://localhost:8081/api/v1/config/current
```

响应（无需重启服务）：

```
{
  "newAlgorithmEnabled": true,
  "maxPageSize": 100
}
```

运行结果

配置刷新时序图：



阶段 6：异步消息通信

设计思路

引入 RabbitMQ 消息队列，实现服务间的异步通信和解耦，提升系统的可扩展性和性能。

应用场景：

- 订单创建后异步发送通知
- 数据变更事件发布

- 削峰填谷（应对流量高峰）
- 最终一致性保证

实现细节

RabbitMQ 部署
Docker Compose 配置：

```
version: '3.8'
services:
  rabbitmq:
    image: rabbitmq:3.13-management
    container_name: rabbitmq
    ports:
      - "5672:5672"    # AMQP端口
      - "15672:15672"  # 管理界面
    environment:
      RABBITMQ_DEFAULT_USER: admin
      RABBITMQ_DEFAULT_PASS: admin123
    volumes:
      - ./rabbitmq-data:/var/lib/rabbitmq
    restart: always
```

启动 RabbitMQ：

```
docker-compose up -d
```

访问管理界面：http://localhost:15672（admin/admin123）

Spring AMQP 集成
添加依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

配置 RabbitMQ 连接（application.yml）：

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: admin
    password: admin123
    publisher-confirm-type: correlated  # 发布确认
```

```yaml
    publisher-returns: true
    listener:
      simple:
        acknowledge-mode: manual  # 手动确认
        retry:
          enabled: true
          max-attempts: 3
```

定义 Exchange 和 Queue

```java
@Configuration
public class RabbitMQConfig {
    // 定义Exchange
    @Bean
    public TopicExchange orderExchange() {
        return new TopicExchange("order.exchange", true, false);
    }

    // 定义Queue
    @Bean
    public Queue orderCreatedQueue() {
        return new Queue("order.created.queue", true);
    }

    @Bean
    public Queue orderNotificationQueue() {
        return new Queue("order.notification.queue", true);
    }

    // 绑定
    @Bean
    public Binding orderCreatedBinding() {
        return BindingBuilder
            .bind(orderCreatedQueue())
            .to(orderExchange())
            .with("order.created");
    }

    @Bean
    public Binding orderNotificationBinding() {
        return BindingBuilder
            .bind(orderNotificationQueue())
            .to(orderExchange())
            .with("order.#");  // 订阅所有order相关消息
    }
}
```

生产者实现

```
@Service
public class OrderProducer {
    @Autowired
    private RabbitTemplate rabbitTemplate;

    public void publishOrderCreatedEvent(Order order) {
        OrderCreatedEvent event = new OrderCreatedEvent(
            order.getId(),
            order.getUserId(),
            order.getTotalAmount(),
            LocalDateTime.now()
        );

        // 设置消息属性
        MessageProperties props = new MessageProperties();
        props.setContentType("application/json");
        props.setDeliveryMode(MessageDeliveryMode.PERSISTENT);

        // 发送消息
        rabbitTemplate.convertAndSend(
            "order.exchange",
            "order.created",
            event,
            message -> {

message.getMessageProperties().setMessageId(UUID.randomUUID().toString());
                return message;
            }
        );

        log.info("Published order created event: {}", event);
    }
}
```

发布确认回调：

```
@Configuration
public class RabbitMQCallbackConfig implements RabbitTemplate.ConfirmCallback
{
    @Autowired
    private RabbitTemplate rabbitTemplate;

    @PostConstruct
    public void init() {
        rabbitTemplate.setConfirmCallback(this);
    }
```

```
    @Override
    public void confirm(CorrelationData correlationData, boolean ack, String
cause) {
        if (ack) {
            log.info("Message sent successfully: {}", correlationData);
        } else {
            log.error("Message send failed: {}, cause: {}", correlationData,
cause);
        }
    }
}
```

消费者实现

```
@Service
public class OrderConsumer {
    @Autowired
    private NotificationService notificationService;

    @RabbitListener(queues = "order.created.queue")
    public void handleOrderCreated(
            @Payload OrderCreatedEvent event,
            @Header(AmqpHeaders.DELIVERY_TAG) long deliveryTag,
            Channel channel) throws IOException {

        try {
            log.info("Received order created event: {}", event);

            // 处理业务逻辑
            notificationService.sendOrderConfirmation(event);

            // 手动确认
            channel.basicAck(deliveryTag, false);
            log.info("Message acknowledged: {}", deliveryTag);

        } catch (Exception e) {
            log.error("Failed to process message: {}", event, e);
            // 拒绝消息，重新入队
            channel.basicNack(deliveryTag, false, true);
        }
    }
}
```

死信队列配置

```
@Configuration
public class DeadLetterQueueConfig {
    // 死信Exchange
    @Bean
    public DirectExchange deadLetterExchange() {
        return new DirectExchange("dlx.exchange");
    }

    // 死信Queue
    @Bean
    public Queue deadLetterQueue() {
        return new Queue("dlx.queue");
    }

    // 正常队列，配置死信
    @Bean
    public Queue businessQueue() {
        Map<String, Object> args = new HashMap<>();
        args.put("x-dead-letter-exchange", "dlx.exchange");
        args.put("x-dead-letter-routing-key", "dlx");
        args.put("x-message-ttl", 60000);  // 消息TTL 60秒
        return new Queue("business.queue", true, false, false, args);
    }

    @Bean
    public Binding deadLetterBinding() {
        return BindingBuilder
            .bind(deadLetterQueue())
            .to(deadLetterExchange())
            .with("dlx");
    }
}
```

测试验证

消息发送验证
创建订单，触发消息发送：

```
curl -X POST http://localhost:8080/api/v1/orders \
  -H "Content-Type: application/json" \
  -H "Authorization: Bearer <token>" \
  -d '{
    "userId": 1,
    "items": [
      {"productId": 101, "quantity": 2}
```

```
    ]
  }'
```

生产者日志：

```
2025-12-10 12:00:00 INFO  Publishing order created event:
OrderCreatedEvent(orderId=1001, userId=1, ...)
2025-12-10 12:00:00 INFO  Message sent successfully:
CorrelationData(id=uuid-12345)
```

RabbitMQ 管理界面：

查看 Queues 页面，`order.created.queue` 应该收到 1 条消息

消息消费验证
消费者日志：

```
2025-12-10 12:00:01 INFO  Received order created event:
OrderCreatedEvent(orderId=1001, ...)
2025-12-10 12:00:01 INFO  Sending order confirmation email to user 1
2025-12-10 12:00:01 INFO  Message acknowledged: 1
```

消息可靠性验证
测试 1：消费者异常，消息重新入队

```
@RabbitListener(queues = "order.created.queue")
public void handleOrderCreated(OrderCreatedEvent event, Channel channel, long
deliveryTag) {
    // 模拟异常
    throw new RuntimeException("Simulated error");
}
```

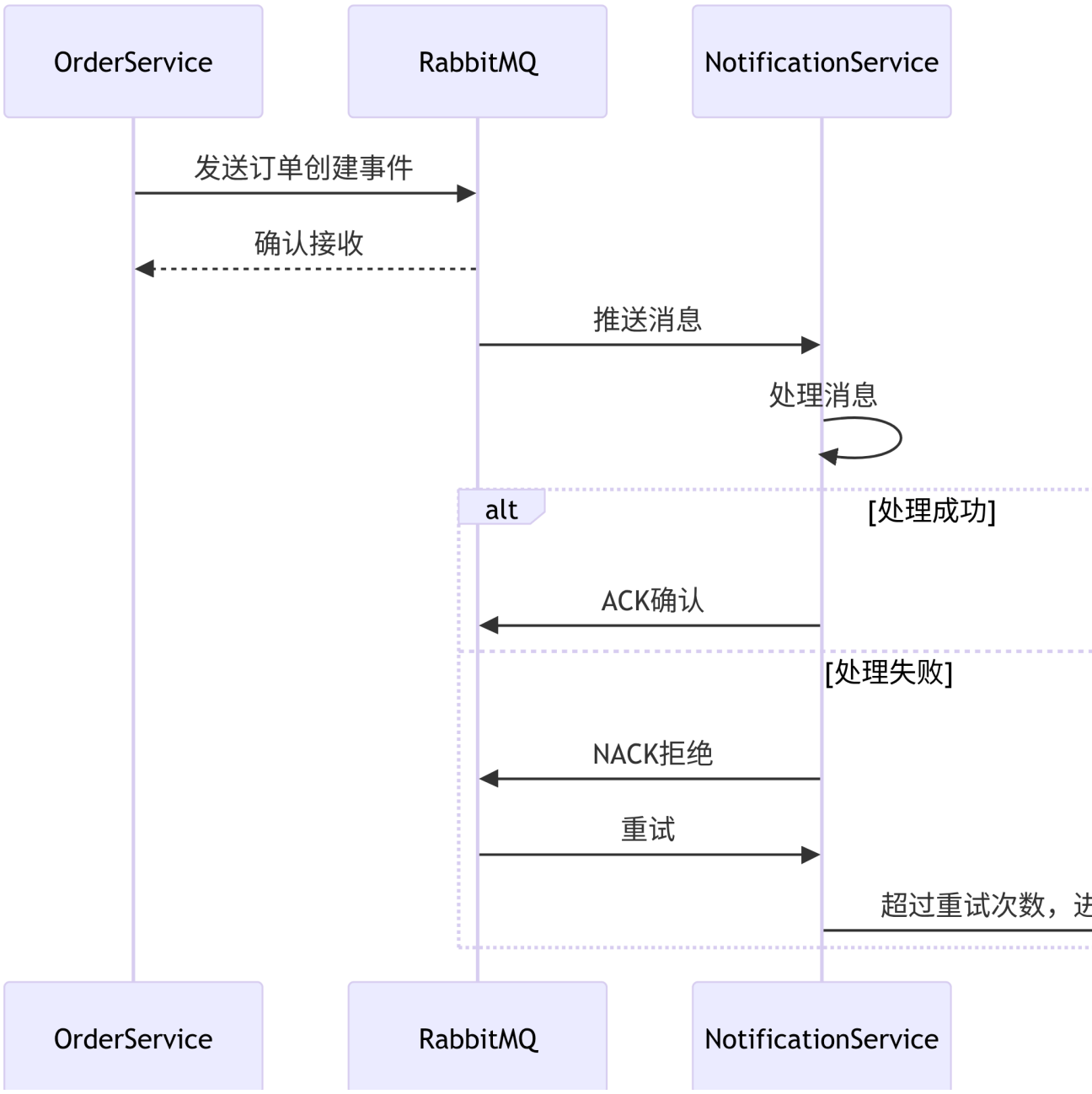
日志：

```
2025-12-10 12:00:01 ERROR Failed to process message, requeuing...
2025-12-10 12:00:02 INFO  Received order created event:
OrderCreatedEvent(orderId=1001, ...) (重试1)
2025-12-10 12:00:03 INFO  Received order created event:
OrderCreatedEvent(orderId=1001, ...) (重试2)
2025-12-10 12:00:04 WARN  Max retry attempts reached, sending to DLQ
```

测试 2：死信队列接收失败消息

查看 RabbitMQ 管理界面，`dlx.queue` 应该收到失败的消息

运行结果
消息流转图：



阶段 7：容器化部署

设计思路
将所有微服务进行容器化，使用 Docker Compose 实现一键部署整个微服务生态系统。

容器化优势：

- 环境一致性
- 快速部署
- 资源隔离
- 易于扩展

实现细节

Dockerfile 编写
多阶段构建 Dockerfile（user-service/Dockerfile）：

```
# 阶段1：构建
FROM maven:3.9-eclipse-temurin-25 AS builder
WORKDIR /app

# 复制pom.xml并下载依赖
COPY pom.xml .
RUN mvn dependency:go-offline -B

# 复制源代码并编译
COPY src ./src
RUN mvn clean package -DskipTests

# 阶段2：运行
FROM eclipse-temurin:25-jre
WORKDIR /app

# 复制jar文件
COPY --from=builder /app/target/user-service-*.jar app.jar

# 健康检查
HEALTHCHECK --interval=30s --timeout=3s --start-period=40s --retries=3 \
  CMD curl -f http://localhost:8081/actuator/health || exit 1

# 启动应用
EXPOSE 8081
ENTRYPOINT ["java", "-jar", "app.jar"]
```

Docker Compose 编排
完整的 docker-compose.yml：

```
version: '3.8'

services:
  # 基础设施：MySQL
  mysql:
```

```
    image: mysql:8.4
    container_name: microservices-mysql
    environment:
      MYSQL_ROOT_PASSWORD: root123
      MYSQL_DATABASE: microservices_db
    ports:
      - "3306:3306"
    volumes:
      - mysql-data:/var/lib/mysql
      - ./init-db:/docker-entrypoint-initdb.d
    networks:
      - microservices-net
    healthcheck:
      test: ["CMD", "mysqladmin", "ping", "-h", "localhost"]
      interval: 10s
      timeout: 5s
      retries: 5

  # 基础设施：Nacos
  nacos:
    image: nacos/nacos-server:v3.1.0
    container_name: microservices-nacos
    environment:
      MODE: standalone
      SPRING_DATASOURCE_PLATFORM: mysql
      MYSQL_SERVICE_HOST: mysql
      MYSQL_SERVICE_DB_NAME: nacos_config
      MYSQL_SERVICE_USER: root
      MYSQL_SERVICE_PASSWORD: root123
    ports:
      - "8848:8848"
      - "9848:9848"
    depends_on:
      mysql:
        condition: service_healthy
    networks:
      - microservices-net
    healthcheck:
      test: ["CMD", "curl", "-f", "http://localhost:8848/nacos/"]
      interval: 10s
      timeout: 5s
      retries: 10

  # 基础设施：RabbitMQ
  rabbitmq:
    image: rabbitmq:3.13-management
    container_name: microservices-rabbitmq
    environment:
```

```
      RABBITMQ_DEFAULT_USER: admin
      RABBITMQ_DEFAULT_PASS: admin123
    ports:
      - "5672:5672"
      - "15672:15672"
    volumes:
      - rabbitmq-data:/var/lib/rabbitmq
    networks:
      - microservices-net
    healthcheck:
      test: ["CMD", "rabbitmq-diagnostics", "ping"]
      interval: 10s
      timeout: 5s
      retries: 5

  # 微服务：用户服务
  user-service:
    build:
      context: ./user-service
      dockerfile: Dockerfile
    container_name: user-service
    environment:
      SPRING_PROFILES_ACTIVE: docker
      SPRING_DATASOURCE_URL: jdbc:mysql://mysql:3306/user_db?serverTimezone=
UTC
      SPRING_CLOUD_NACOS_DISCOVERY_SERVER_ADDR: nacos:8848
      SPRING_CLOUD_NACOS_CONFIG_SERVER_ADDR: nacos:8848
    ports:
      - "8081:8081"
    depends_on:
      mysql:
        condition: service_healthy
      nacos:
        condition: service_healthy
    networks:
      - microservices-net
    restart: on-failure

  # 微服务：业务服务
  business-service:
    build:
      context: ./business-service
      dockerfile: Dockerfile
    container_name: business-service
    environment:
      SPRING_PROFILES_ACTIVE: docker
      SPRING_DATASOURCE_URL: jdbc:mysql://mysql:3306/business_db?
serverTimezone=UTC
```

```
      SPRING_CLOUD_NACOS_DISCOVERY_SERVER_ADDR: nacos:8848
      SPRING_RABBITMQ_HOST: rabbitmq
    ports:
      - "8082:8082"
    depends_on:
      mysql:
        condition: service_healthy
      nacos:
        condition: service_healthy
      rabbitmq:
        condition: service_healthy
    networks:
      - microservices-net
    restart: on-failure

  # 微服务：API网关
  api-gateway:
    build:
      context: ./api-gateway
      dockerfile: Dockerfile
    container_name: api-gateway
    environment:
      SPRING_PROFILES_ACTIVE: docker
      SPRING_CLOUD_NACOS_DISCOVERY_SERVER_ADDR: nacos:8848
    ports:
      - "8080:8080"
    depends_on:
      nacos:
        condition: service_healthy
    networks:
      - microservices-net
    restart: on-failure

networks:
  microservices-net:
    driver: bridge

volumes:
  mysql-data:
  rabbitmq-data:
```

数据库初始化脚本

init-db/01-create-databases.sql：

```
-- 创建各服务数据库
CREATE DATABASE IF NOT EXISTS user_db;
CREATE DATABASE IF NOT EXISTS business_db;
```

```
CREATE DATABASE IF NOT EXISTS nacos_config;

-- 授权
GRANT ALL PRIVILEGES ON user_db.* TO 'root'@'%';
GRANT ALL PRIVILEGES ON business_db.* TO 'root'@'%';
GRANT ALL PRIVILEGES ON nacos_config.* TO 'root'@'%';
FLUSH PRIVILEGES;
```

测试验证

构建和启动

步骤 1：构建所有服务镜像

```
# 构建镜像
docker-compose build

# 查看镜像
docker images | grep microservices
```

预期输出：

```
user-service         latest    abc123def456   2 minutes ago   350MB
business-service     latest    def456ghi789   2 minutes ago   360MB
api-gateway          latest    ghi789jkl012   1 minute ago    340MB
```

步骤 2：启动整个系统

```
docker-compose up -d
```

步骤 3：查看容器状态

```
docker-compose ps
```

预期输出：

```
NAME                    STATUS              PORTS
microservices-mysql     Up (healthy)        0.0.0.0:3306->3306/tcp
microservices-nacos     Up (healthy)        0.0.0.0:8848->8848/tcp
microservices-rabbitmq  Up (healthy)        0.0.0.0:5672->5672/tcp
user-service            Up                  0.0.0.0:8081->8081/tcp
business-service        Up                  0.0.0.0:8082->8082/tcp
api-gateway             Up                  0.0.0.0:8080->8080/tcp
```

服务健康检查

```
# 检查Nacos服务列表
curl http://localhost:8848/nacos/v1/ns/instance/list?serviceName=user-service

# 检查Gateway路由
curl http://localhost:8080/actuator/gateway/routes

# 测试端到端调用
curl http://localhost:8080/api/v1/users
```

日志查看

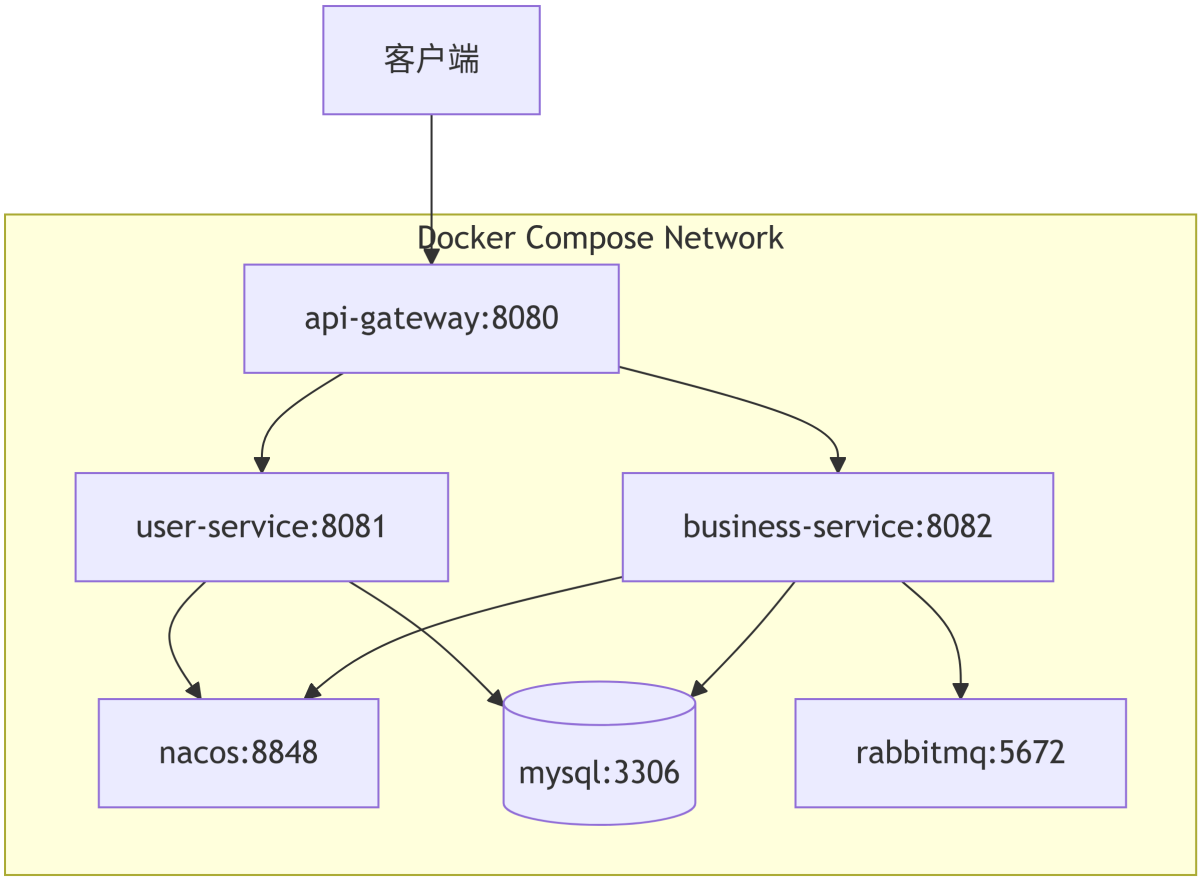
```
# 查看所有服务日志
docker-compose logs -f

# 查看特定服务日志
docker-compose logs -f user-service

# 查看最近100行日志
docker-compose logs --tail=100 user-service
```

扩展服务实例

```
# 扩展user-service到3个实例
docker-compose up -d --scale user-service=3

# 验证负载均衡
for i in {1..10}; do
  curl http://localhost:8080/api/v1/users/1
  echo ""
done
```

运行结果

容器架构图：



资源使用情况：

```
docker stats --no-stream
```

```
CONTAINER           CPU %     MEM USAGE / LIMIT     NET I/O
user-service        2.5%      512MB / 2GB           1.2kB / 890B
business-service    3.0%      480MB / 2GB           980B / 650B
api-gateway         1.8%      450MB / 2GB           2.1kB / 1.5kB
microservices-nacos 5.2%      1.2GB / 4GB           3.2kB / 2.1kB
microservices-mysql 8.5%      800MB / 4GB           4.5kB / 3.2kB
```

系统测试

功能测试

测试用例设计

测试用例 ID	功能模块	测试场景	预期结果
TC001	用户注册	新用户注册	成功创建用户，返回 201
TC002	用户登录	正确用户名密码	返回 JWT Token
TC003	服务发现	查询已注册服务	返回服务实例列表
TC004	负载均衡	多次调用同一接口	请求均匀分布到各实例

测试执行

性能测试

测试环境

- **硬件配置**：[CPU/内存/磁盘]
- **软件版本**：[操作系统/Docker 版本]
- **测试工具**：Apache JMeter / wrk

测试场景

场景 1：单服务压力测试

测试参数：

- 并发用户数：100
- 测试时长：60 秒
- 请求路径：`GET /api/v1/users`

测试命令：

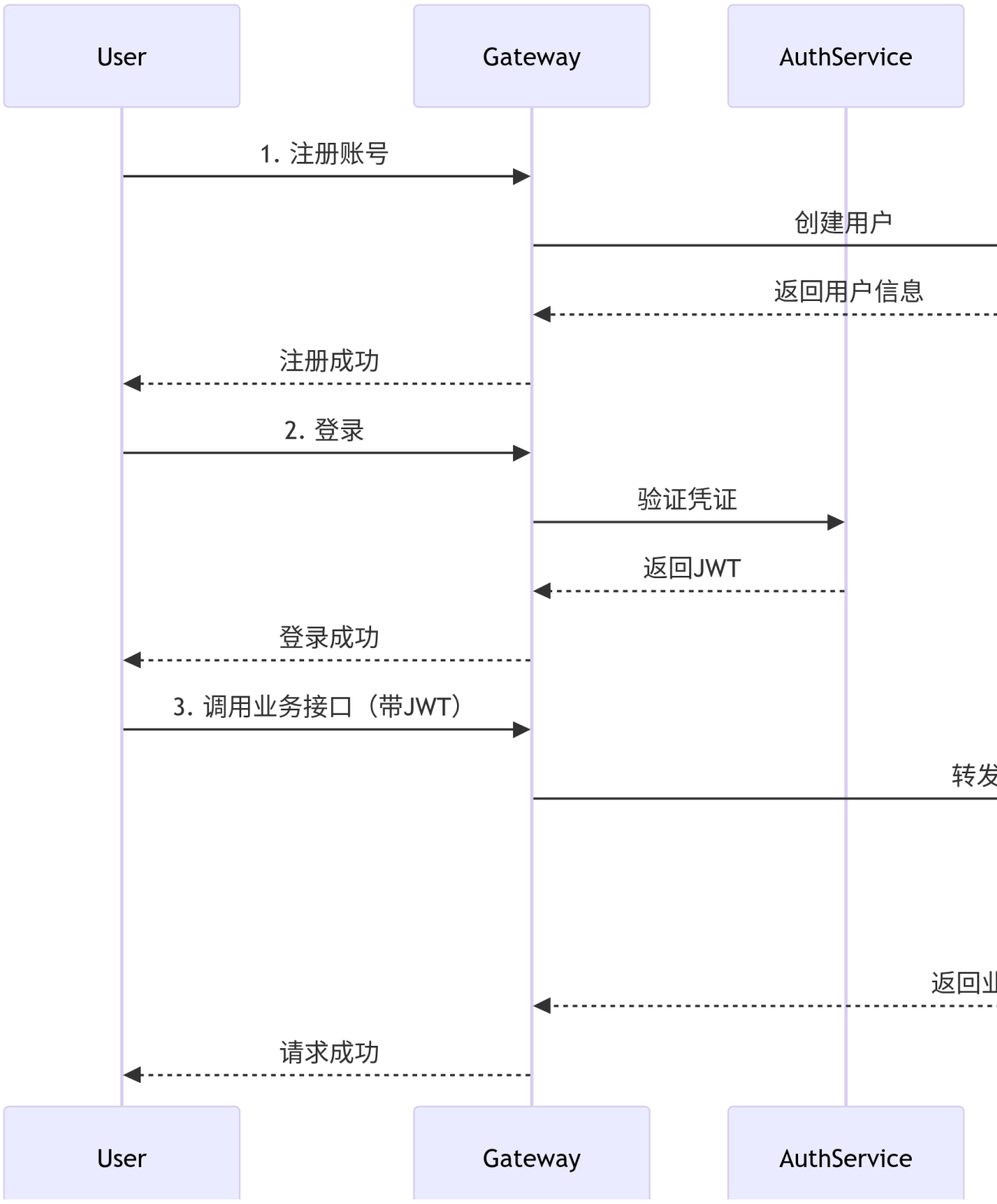
```
wrk -t 4 -c 100 -d 60s http://localhost:8080/api/v1/users
```

测试结果：

```
Running 60s test @ http://localhost:8080/api/v1/users
  4 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    45.23ms   12.45ms  180.32ms   78.56%
    Req/Sec   550.23     89.45   750.00     82.34%
  132456 requests in 60.00s, 25.34MB read
Requests/sec:   2207.60
Transfer/sec:    432.15KB
```

集成测试

端到端测试流程


User
Gateway
AuthService
1. 注册账号
创建用户
返回用户信息
注册成功
2. 登录
验证凭证
返回JWT
登录成功
3. 调用业务接口（带JWT）
转发
返回业
请求成功
User
Gateway
AuthService

项目总结

技术亮点

1. [亮点 1 标题]

2. [亮点 2 标题]

遇到的问题与解决方案

问题 1：[问题描述]
问题现象：

原因分析：

解决方案：

个人收获

1. **微服务架构理解**：
   - [收获 1]
   - [收获 2]
2. **技术能力提升**：
   - [收获 3]
   - [收获 4]
3. **工程实践经验**：
   - [收获 5]
   - [收获 6]

未来改进方向

1. **功能扩展**：
   - ☐ [改进点 1]
   - ☐ [改进点 2]
2. **性能优化**：
   - ☐ [优化点 1]
   - ☐ [优化点 2]
3. **运维完善**：
   - ☐ [完善点 1]
   - ☐ [完善点 2]

附录

完整 API 文档

数据库 ER 图

参考资料

1. Spring Cloud 官方文档：https://spring.io/projects/spring-cloud
2. Nacos 官方文档：https://nacos.io/
3. RabbitMQ 官方文档：https://www.rabbitmq.com/
4. Docker 官方文档：https://docs.docker.com/
5. 课程 PPT 和演示代码

项目仓库

- **Git 仓库地址**：[GitHub/Gitee 链接]
- **演示视频**：[视频链接]
- **部署地址**：[如果有线上部署]

声明：本报告中的所有代码和配置均为本人独立完成，参考资料已在文末列出。