美多商城项目出彩内容

本项目引入了包括积极建立公共模块、接入 Nacos 配置中心、接入 Spring Boot Admin、接入 Zipkin、引入 JWT、结合 Spring Cloud Stream 配合 Rabbit MQ 的发布-订阅模式以及事物系统来打造高可靠性的订单服务，以及自行实现管理员后台面板等出彩内容。

一、 项目启动

本节也可以在源代码根目录下的 **README.md** 中找到。
以下是开发者所使用的环境软件版本表，可以根据实际情况进行调整：

环境准备

软件	版本
JDK	17.0.18-ms
Maven	3.9.9
MySQL	8.0.33
Nacos Server	2.3.2
Sentinel Dashboard	1.8.6
Zipkin Server	3.6.1
RabbitMQ Server	4.3.1
Erlang	27.3.4.13

数据库准备

导入 **spring-cloud.sql** 文件（此文件位于**代码库根目录**下）到 MySQL 数据库中，创建数据库并导入数据。
mysql -u root -p < spring-cloud.sql

启动服务

按以下顺序启动服务：

1. MySQL 数据库（端口 3306）
2. Nacos Server（端口 8848）
3. RabbitMQ Server（端口 5672/15672）
4. Sentinel Dashboard（端口 8080）
5. Zipkin Server（端口 9411）
6. admin-server（端口 9090）
7. user-provider（端口 8601）
8. user-consumer（端口 8893）
9. goods-provider（端口 8750）

10. goods-consumer（端口 8896）
11. admin-consumer（端口 8897）
12. gateway-service（端口 5000）

启动微服务

使用 **安装依赖.cmd** 来安装所有服务的依赖，安装完成后再执行下一步，也可以在命令行输入以下命令，命令行参见源代码根目录下的 **README.md**。
可以使用 **启动.cmd** 来快速启动所有服务，也可以在目录下，打开多个终端窗口，分别启动各个服务，命令行参见源代码根目录下的 **README.md**。

访问微服务

默认管理员账户的账户和密码均为 **admin**;
默认普通用户的账户和密码均为 **test**，可以通过以下方式访问各个微服务；
直接访问 http://localhost:5000/ 将 302 跳转至商品列表页

通过 Gateway 访问

页面	URL	说明
登录页	http://localhost:5000/user/tologin	用户登录页面
注册页	http://localhost:5000/user/toregister	用户注册页面
商品列表	http://localhost:5000/goods/getAll	展示所有商品
商品详情	http://localhost:5000/goods/detail?gid=1	查看商品详情
购物车	http://localhost:5000/goods/cart	查看购物车（需先登录）
我的订单	http://localhost:5000/goods/order	查看订单（需先登录）
后台管理	http://localhost:5000/admin/users	后台页面（需 admin 角色）

直接访问 Consumer

注意，本项目如果只访问 Consumer 将无法跨域共享登录状态，建议通过 Gateway 访问，因此本文略过此部分，如需访问，请参见源代码根目录下的 **README.md**。

运行备注

目前 Sentinel 规则为 QPS 10 才触达上限，如需测试限流，请修改规则。
Nacos 配置中心名为 **b2b-site-slogan**，将在 **user-provider** 启动时自动尝试创建，如需测试请自行修改。

二、 出彩内容：积极建立公共模块

项目抽取了 common-module 公共模块，将实体类、常量类、工具类、RabbitMQ 消息模型集中管理，避免多个微服务重复定义同一批类和字段不一致的问题。这样可以让各个服务共享同一套数据契约和认证语义，使项目从“多个服务拼接”提升为“有统一协议的微服务系统”，提前考虑了多模块协作、公共协议复用和后续扩展。

示例代码 / 统一导出 Header 名

```
public final class AuthConstants {

    public static final String AUTH_TOKEN = "AUTH_TOKEN";
    public static final String HEADER_AUTH_UID = "X-Auth-Uid";
    public static final String HEADER_AUTH_NAME = "X-Auth-Name";
    public static final String HEADER_AUTH_ROLE = "X-Auth-Role";

    private AuthConstants() {
    }
}
```

示例代码 / 统一管理 OrderMessage 对象

```
public class OrderMessage implements Serializable {

    private static final long serialVersionUID = 1L;

    private int gid;
    private String messageId;
    private Integer cartId;
    private String goodsname;
    private int number;
    private int price;
    private int uid;
}
```

示例代码 / 抽取出统一的 Sentinel 规则加载器

```
public final class SentinelRuleLoader {

    private SentinelRuleLoader() {
    }

    public static void loadDefaultRules(String... resources) {
        List<FlowRule> flowRules = new ArrayList<>();
        List<DegradeRule> degradeRules = new ArrayList<>();

        for (String resource : resources) {
            flowRules.add(createFlowRule(resource, RuleConstant.FLOW_GRADE_QPS));
            flowRules.add(createFlowRule(resource, RuleConstant.FLOW_GRADE_THREAD));
            degradeRules.add(createDegradeRule(resource));
        }

        FlowRuleManager.loadRules(flowRules);
        DegradeRuleManager.loadRules(degradeRules);
    }

    private static FlowRule createFlowRule(String resource, int grade) {
        FlowRule rule = new FlowRule();
        rule.setResource(resource);
        rule.setGrade(grade);
        rule.setCount(10);
        rule.setLimitApp("default");
```

```
        return rule;
    }

    private static DegradeRule createDegradeRule(String resource) {
        DegradeRule rule = new DegradeRule();
        rule.setResource(resource);
        rule.setGrade(RuleConstant.DEGRADE_GRADE_EXCEPTION_RATIO);
        rule.setCount(0.2);
        rule.setTimeWindow(10);
        rule.setMinRequestAmount(5);
        rule.setStatIntervalMs(1000);
        return rule;
    }
}
```

示例代码 / User Consumer 通过统一规则加载器加载规则

```
@Configuration
public class SentinelRuleConfig {

    @EventListener(ApplicationReadyEvent.class)
    public void initRules() {
        SentinelRuleLoader.loadDefaultRules(
                "/user/toregister",
                "/user/tologin",
                "/user/register",
                "/user/login"
        );
    }
}
```

示例代码 / 抽取出统一的解析上传目录路径的工具类

```
public final class UploadPathUtil {

    public static final String DEFAULT_UPLOAD_DIR = "uploads";
    public static final String UPLOAD_URL_PREFIX = "/uploads/";

    private UploadPathUtil() {
    }

    public static Path resolveUploadDir() {
        return resolveUploadDir(DEFAULT_UPLOAD_DIR);
    }

    public static Path resolveUploadDir(String uploadDir) {
        Path configuredPath = Paths.get(uploadDir.trim());
        if (configuredPath.isAbsolute()) {
            return configuredPath.normalize();
        }

        Path userDir =
Paths.get(System.getProperty("user.dir")).toAbsolutePath().normalize();
        Path nestedProjectDir = userDir.resolve("spring-cloud-demo");
        if (Files.exists(nestedProjectDir.resolve("pom.xml"))) {
            return nestedProjectDir.resolve(configuredPath).normalize();
        }
```

```
        Path parentDir = userDir.getParent();
        if (Files.exists(userDir.resolve("pom.xml"))
                && parentDir != null
                && Files.exists(parentDir.resolve("pom.xml"))) {
            return parentDir.resolve(configuredPath).normalize();
        }

        return userDir.resolve(configuredPath).normalize();
    }
}
```

示例代码 / Goods Consumer 通过统一工具类解析上传目录

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/goods" + UploadPathUtil.UPLOAD_URL_PREFIX + "**")
            .addResourceLocations(toResourceLocation(UploadPathUtil.resolveUploadDir
()));
}

private String toResourceLocation(Path path) {
    String location = path.toUri().toString();
    return location.endsWith("/") ? location : location + "/";
}
```

三、 出彩内容：接入 Nacos 配置中心

项目使用 Nacos 承担服务注册发现和配置中心两类职责。各服务启动后注册到 Nacos，Gateway 和 OpenFeign 通过服务名完成调用；同时，页面欢迎语等配置从 Nacos 统一读取，公共配置也可以集中维护。

示例代码 / 引入 Nacos 配置中心

```
spring:
    config:
        import: optional:nacos:b2b-site-slogan.yml
    cloud:
        nacos:
            server-addr: localhost:8848
            discovery:
                server-addr: localhost:8848
            config:
                server-addr: localhost:8848
                file-extension: yml
                data-id: b2b-site-slogan.yml
                group: DEFAULT_GROUP
                content: |
                    site:
                      welcome: "欢迎光临美多商城，Nacos 动态配置测试"
```

示例代码 / 读取 Nacos 动态配置

```
@RefreshScope
public class AdminController {
    @Value("${site.welcome:欢迎来到美多商城!}")
    private String welcome;
}
```

示例代码 / 自动化创建 Nacos 动态配置

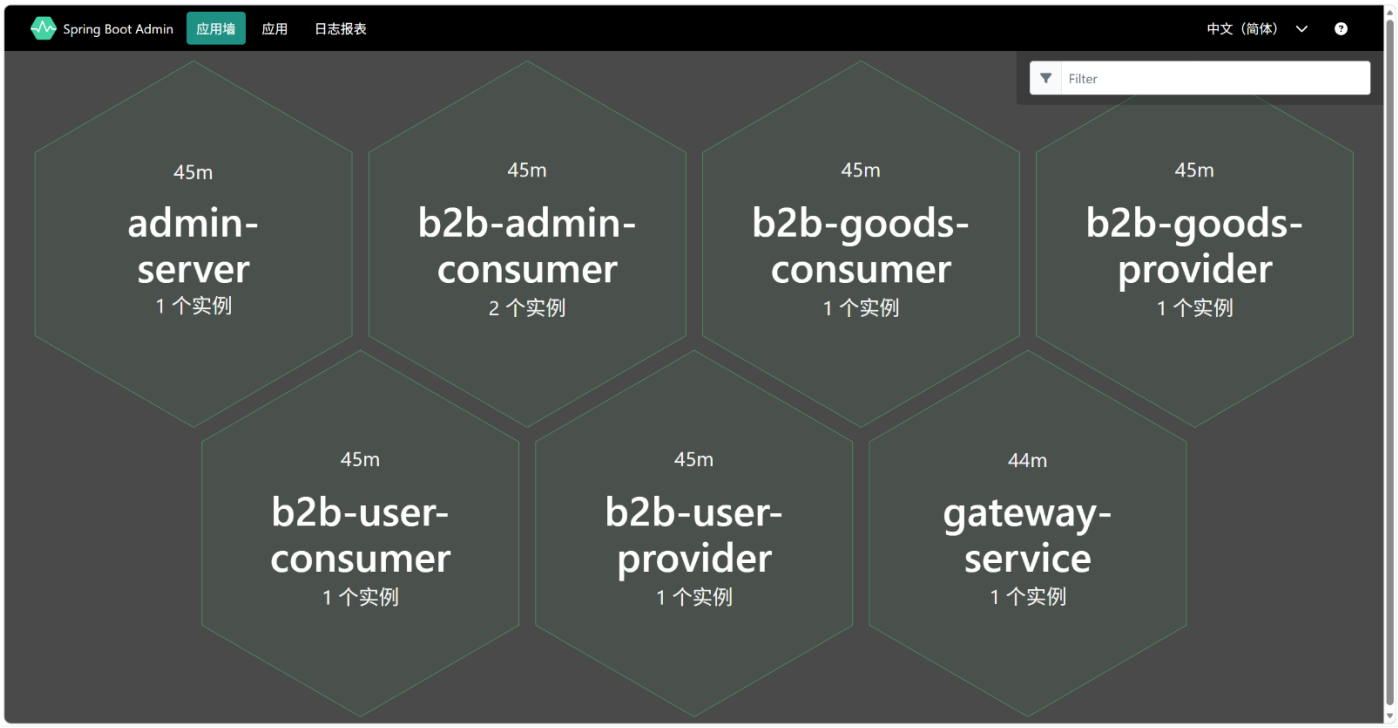
```
@Override
public void run(String... args) {

    try {
        // 创建配置服务
        Properties properties = new Properties();
        properties.put("serverAddr", serverAddr);
        ConfigService configService = NacosFactory.createConfigService(properties);

        // 检查配置是否已存在
        String existingConfig = null;
        try {
            existingConfig = configService.getConfig(dataId, group, 5000);
        } catch (NacosException e) {
            log.debug("配置不存在，将创建新配置 b2b-site-slogan");
        }

        if (existingConfig != null && !existingConfig.isEmpty()) {
            log.info("配置已存在:\n{}", existingConfig);
        } else {
            // 发布配置
            boolean isPublishOk = configService.publishConfig(dataId, group,
content);
            if (isPublishOk) {
                log.info("配置发布成功! ");
                log.info("配置内容:\n{}", content);
            } else {
                log.error("配置发布失败");
            }
        }

        // 验证配置
        String config = configService.getConfig(dataId, group, 5000);
        log.info("读取配置成功，内容:\n{}", config);

    } catch (NacosException e) {
        log.error("Nacos 配置初始化失败: {}", e.getMessage(), e);
    }
}
```

例图 / Nacos 动态配置效果图

四、 出彩内容：接入 Spring Boot Admin 侦测

项目单独建设 **admin-server** 模块作为 Spring Boot Admin 监控中心，用于查看微服务健康状态、运行指标、环境配置和 Actuator 信息。各业务服务暴露 Actuator 端点，并将 Spring Boot Admin Client 指向 **http://localhost:9090**，从而统一接入监控中心。这使项目具备“能运行、能观察、能定位”的基础运维能力。

例图 / Spring Boot Admin 服务监控



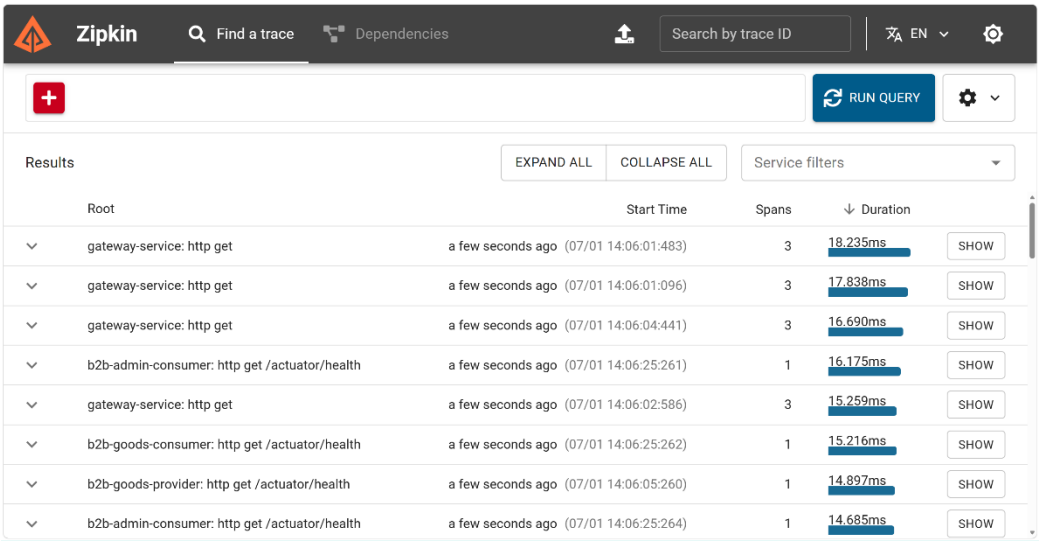
五、 出彩内容：接入 Zipkin 进行链路追踪

在微服务中，一次用户请求可能经过网关、页面服务、业务提供者、数据库等多个环节。如果没有链路追踪，接口变慢或调用失败时很难定位具体服务。本项目通过 Zipkin 收集调用链路，可以在页面中查看服务调用路径、耗时和 Trace 明细。这让项目再发现问题后可以快速发现和定位问题、梳理服务依赖。

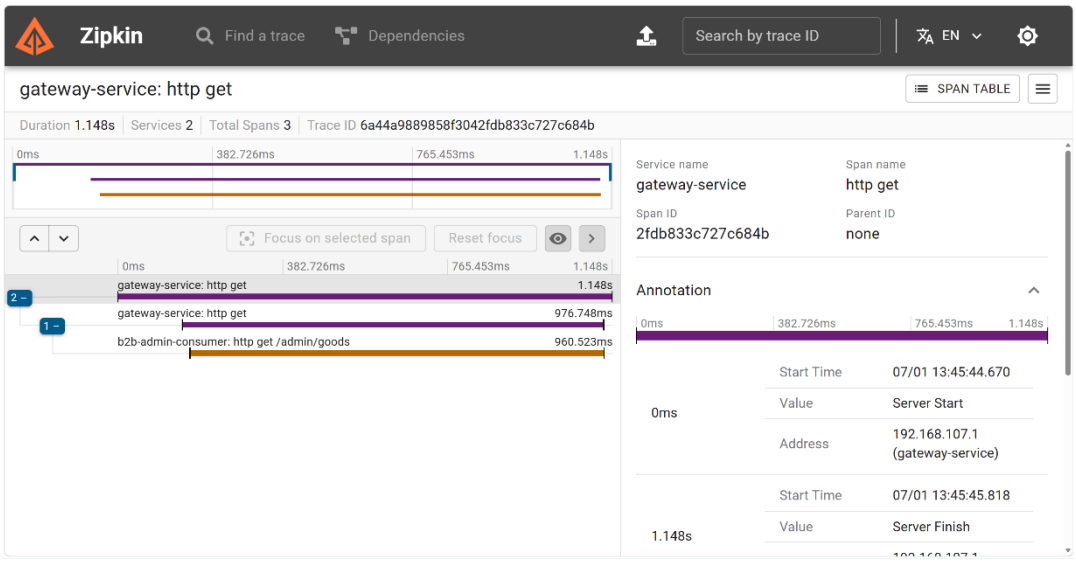
示例代码 / 将 Zipkin 引入项目

```yaml
management:
  tracing:
    sampling:
      probability: 1.0
  zipkin:
    tracing:
      endpoint: http://localhost:9411/api/v2/spans
```

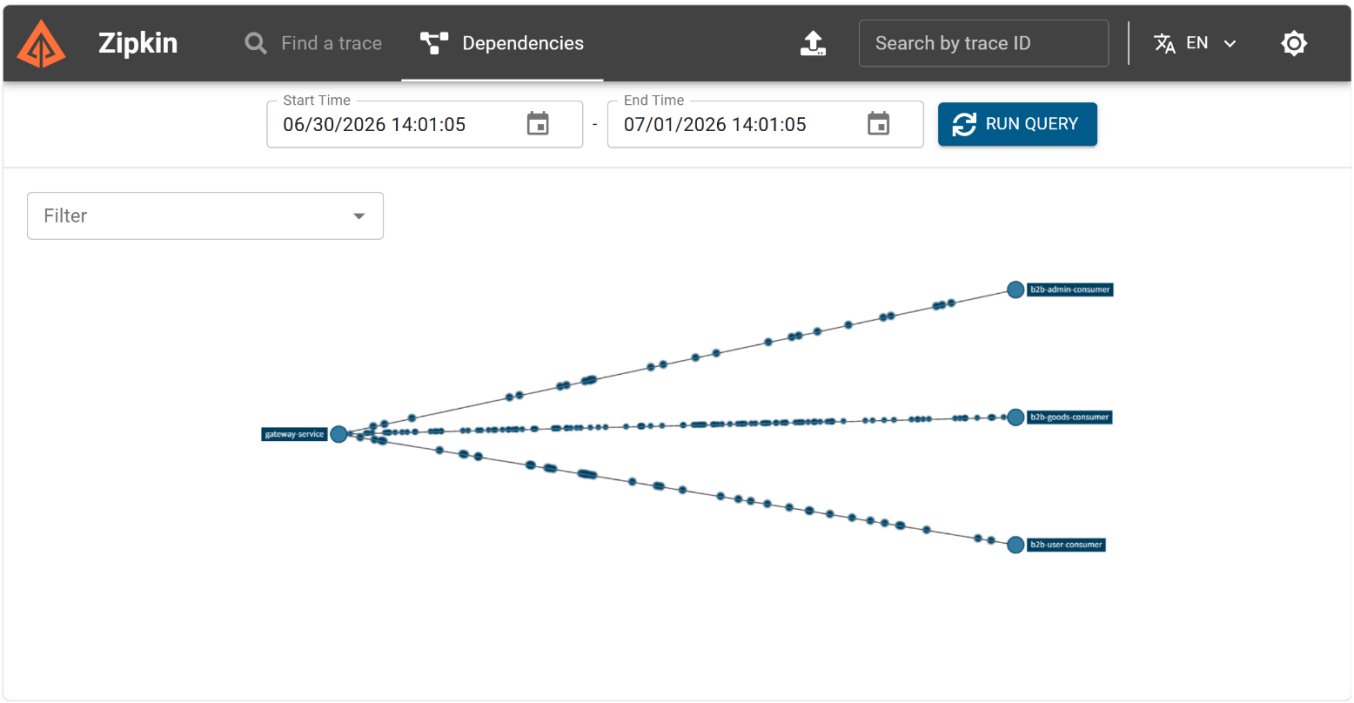
例图 / Zipkin 服务列表



例图 / Zipkin 链路追踪



Zipkin 依赖追踪



六、 出彩内容：引入 JWT 实现无状态身份认证

传统 Session 登录通常依赖单个应用内存或集中式 Session 存储，在微服务架构下会带来共享 Session、服务扩容和跨服务认证的问题。本项目使用 JWT 将用户账号、用户名、角色等信息写入 Token，登录成功后放入 **AUTH_TOKEN** Cookie。网关读取 Cookie 后解析 JWT，并将用户信息注入请求头，后续服务无需共享 Session，也能识别当前用户身份。下游服务（如管理员后台）将进一步根据注入的请求头或 JWT 中的 **urole**（或其他字段）判断当前用户是否具有足够的权限，是否允许访问。

示例代码 / 登录成功后生成 JWT 并写入 Cookie

```
User user = userService.login(uname, upassword);
if (user != null) {
    String token = JwtUtil.generateToken(user);
    httpResponse.addCookie(AuthCookieUtil.createAuthCookie(token));
    model.addAttribute("user", user);
    return "login";
}
```

示例代码 / 负责设置和使 Token Cookie 失效的工具类

```
public static Cookie createAuthCookie(String token) {
    Cookie cookie = new Cookie(AuthConstants.AUTH_TOKEN, token);
    cookie.setPath("/");
    cookie.setHttpOnly(true);
    cookie.setMaxAge(AuthConstants.TOKEN_MAX_AGE_SECONDS);
    return cookie;
}
```

```
public static void expireAuthToken(HttpServletResponse response) {
    Cookie cookie = new Cookie(AuthConstants.AUTH_TOKEN, "");
    cookie.setPath("/");
    cookie.setHttpOnly(true);
    cookie.setMaxAge(0);
    response.addCookie(cookie);
}
```

示例代码 / 负责生成和解析 Token 的 JWT 工具类

```
public final class JwtUtil {

    // 全局共享密钥
    private static final SecretKey KEY = Keys.hmacShaKeyFor(
            "spring-cloud-demo-jwt-secret-key".getBytes(StandardCharsets.UTF_8));

    public static String generateToken(User user) {
        Date now = new Date();
        String urole = user.getUrole() == null || user.getUrole().isBlank()
                ? AuthConstants.ROLE_USER
                : user.getUrole().trim();

        return Jwts.builder()
            .subject(String.valueOf(user.getUaccount()))
            .claim("uname", user.getUname())
            .claim("urole", urole)
            .issuedAt(now)
            .expiration(new Date(now.getTime() +
AuthConstants.TOKEN_MAX_AGE_MILLIS))
            .signWith(KEY)
            .compact();
    }

    public static JwtEntity parseToken(String token) {
        Claims claims = Jwts.parser()
            .verifyWith(KEY)
            .build()
            .parseSignedClaims(token)
            .getPayload();
        JwtEntity jwtEntity = new JwtEntity();
        jwtEntity.setSubject(claims.getSubject());
        try {
            jwtEntity.setUaccount(Integer.parseInt(claims.getSubject()));
        } catch (Exception e) {
            jwtEntity.setUaccount(0);
        }
        jwtEntity.setUname(claims.get("uname", String.class));
        jwtEntity.setUrole(claims.get("urole", String.class));
        jwtEntity.setIssuedAt(claims.getIssuedAt());
        jwtEntity.setExpiration(claims.getExpiration());
        return jwtEntity;
    }
}
```

示例代码 / Gateway 解析 JWT 并注入 Header

```
ServerHttpRequest sanitizedRequest = stripAuthHeaders(request);
```

```
HttpCookie cookie =
sanitizedRequest.getCookies().getFirst(AuthConstants.AUTH_TOKEN);

JwtEntity jwtEntity = JwtUtil.parseToken(cookie.getValue());
ServerHttpRequest.Builder requestBuilder = sanitizedRequest.mutate()
       .header(AuthConstants.HEADER_AUTH_UID, jwtEntity.getSubject())
       .header(AuthConstants.HEADER_AUTH_NAME, jwtEntity.getUname());

if (jwtEntity.getUrole() != null && !jwtEntity.getUrole().isBlank()) {
    requestBuilder.header(AuthConstants.HEADER_AUTH_ROLE, jwtEntity.getUrole());
}
```

示例代码 / 管理员后台根据角色控制访问

```
Optional<JwtEntity> jwtEntity = AuthTokenUtil.resolveJwtEntity(request);
if (jwtEntity.isPresent()
       && AuthConstants.ROLE_ADMIN.equalsIgnoreCase(jwtEntity.get().getUrole())) {
    return true;
}

if (jwtEntity.isEmpty() && AuthCookieUtil.findAuthToken(request).isPresent()) {
    AuthCookieUtil.expireAuthToken(response);
}
response.sendRedirect(LoginUrlUtil.resolveLoginUrl(request));
return false;
```

七、 出彩内容：引入 HiddenHttpMethodFilter

HTML 原生表单只支持 GET 和 POST，不能直接提交 PUT 或 DELETE。如果所有修改和删除都写成 POST 或 GET，虽然能实现功能，但接口语义不够清楚，尤其是删除操作如果用 GET，也不符合 Web 开发习惯。本项目在 **admin-consumer** 和 **goods-consumer** 中开启 HiddenHttpMethodFilter。页面仍然使用兼容性最好的 method="post"，但额外提交隐藏字段_method=put 或_method=delete。Spring MVC 接收到请求后，会把真实处理方法转换为 PUT 或 DELETE，从而匹配控制器中的@PutMapping、@DeleteMapping。

示例代码 / 在服务中开启 HiddenHttpMethodFilter

```
spring:
  mvc:
    hiddenmethod:
      filter:
        enabled: true
```

示例代码 / 前端删除表单通过隐藏字段实现指定处理方法

```
<form th:action="@{/admin/users/delete}" method="post">
    <input type="hidden" name="_method" value="delete">
    <input type="hidden" name="uaccount" th:value="${user.uaccount}">
    <button type="submit" class="delete-link">删除</button>
</form>
```

示例代码 / 正确后端使用 REST 语义编写端点示意

```
@DeleteMapping("/users/delete")
public String deleteUser(@RequestParam("uaccount") int uaccount, Model model) {
    model.addAttribute("msg", userAdminService.deleteUser(uaccount));
    return users(model);
}
```

八、出彩内容：接入 Spring Cloud Stream + Rabbit MQ 的发布-订阅模式配合事务系统提高订单处理服务的可靠性

项目使用 Spring Cloud Stream + RabbitMQ 将订单创建从同步调用改造成事件驱动流程，**goods-consumer** 发布订单事件，**goods-provider** 订阅并处理扣库存、写订单、清购物车。同时，消费端从设计之初就使用 @Transactional 做事务保障，把扣库存、写订单、删除购物车、记录消费日志放进同一个本地事务中，提升订单处理的可靠性。

下单场景并没有直接在页面服务中完成所有数据库操作，而是通过 **OrderMessage** 发布订单创建事件。这样可以降低服务耦合，让订单处理逻辑集中在商品提供者服务中。当前 **goods-provider** 作为订阅者消费 **order.create** 事件。假设未来新增短信通知、积分累计、销售统计等服务，只需要增加新的消费者订阅同一交换机即可，也不需要修改下单入口。

使用 @Transactional 主要解决消费端的一致性问题：订单消息到达后，扣减库存、插入订单、删除购物车属于一个业务整体。如果写订单失败或删除购物车失败，方法会抛出异常，数据库操作整体回滚，避免出现“库存扣了但订单没生成”这类不一致情况。项目还通过 **messageId** 和 order_**consume**_log 做消费记录，避免 RabbitMQ 重复投递时重复扣库存、重复生成订单。。

示例代码 / 生产者绑定 RabbitMQ

```
spring:
  cloud:
    stream:
      bindings:
        order-create-out-0:
          destination: b2b.order.exchange
      rabbit:
        bindings:
          order-create-out-0:
            producer:
              routing-key-expression: '''order.create'''
```

示例代码 / 生产者发送订单消息

```
public void sendOrderMessage(OrderMessage message) {
    log.info("发送订单消息: {}", message);
    boolean sent = streamBridge.send("order-create-out-0", message);
    if (!sent) {
        throw new IllegalStateException("订单消息发送失败");
    }
    log.info("订单消息发送成功");
}
```

示例代码 / 消费者订阅订单消息

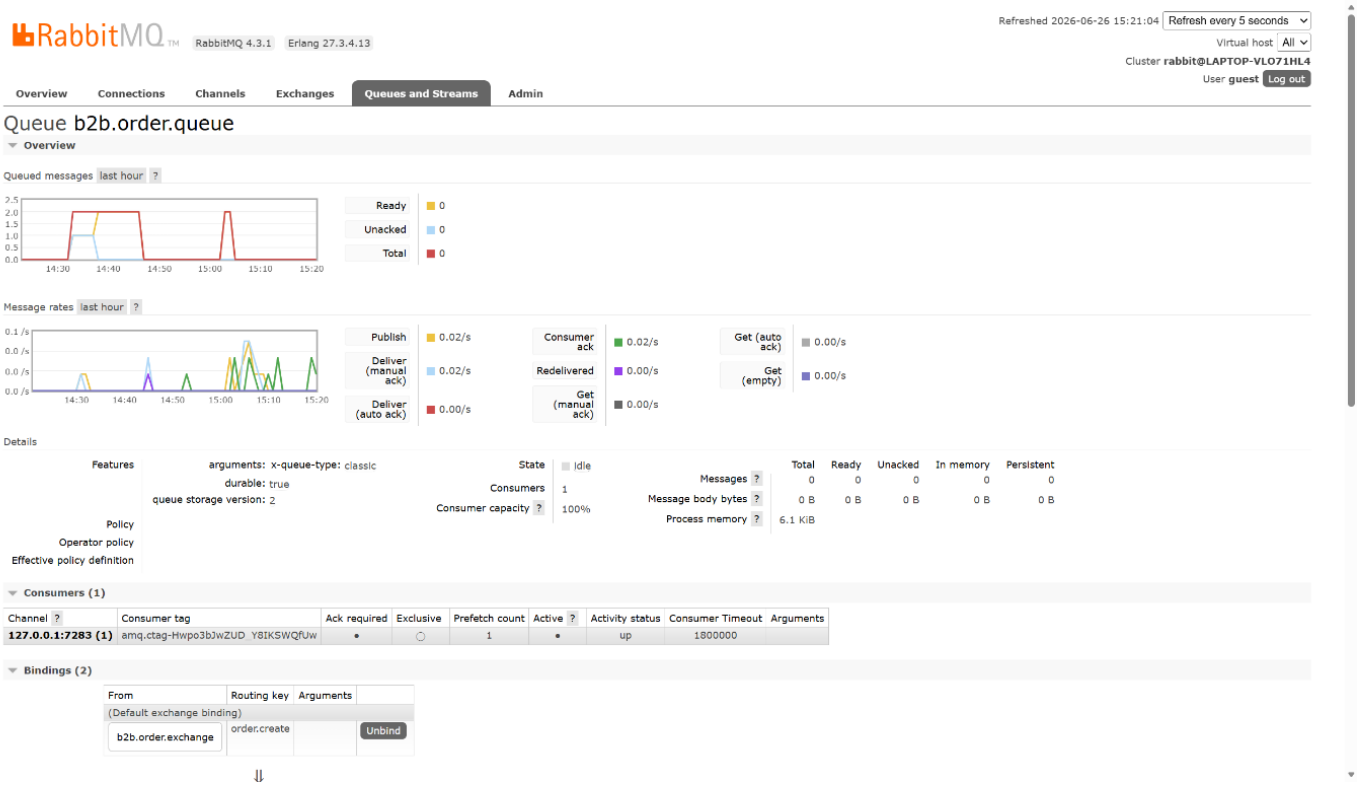
```
spring:
  cloud:
    stream:
      bindings:
        order-create-in-0:
          destination: b2b.order.exchange
          group: b2b.order.queue
          consumer:
            max-attempts: 1
      rabbit:
        bindings:
          order-create-in-0:
            consumer:
              binding-routing-key: order.create
              queue-name-group-only: true
              requeue-rejected: true
              prefetch: 1
```

示例代码 / 消费者处理订单事件

```
@Override
@Transactional(rollbackFor = Exception.class)
public void accept(OrderMessage msg) {
    log.info("收到订单消息: {}", msg);
    try {
        String messageId = msg.getMessageId();
        if (messageId == null || messageId.isBlank()) {
            messageId = UUID.randomUUID().toString();
            log.warn("订单消息缺少 messageId，将按非幂等消息处理: {}", msg);
        }
        int inserted = shoppingDao.insertConsumeLog(messageId);
        if (inserted == 0) {
            log.info("订单消息已处理，跳过重复消费: messageId={}", messageId);
            return;
        }
        int affected = goodsDao.updateStock(msg.getGid(), msg.getNumber());
        if (affected == 0) {
            log.warn("库存不足: gid={}, number={}", msg.getGid(), msg.getNumber());
            shoppingDao.deleteConsumeLog(messageId);
            return;
        }
```

```
        Userorder order = new Userorder();
        order.setGoodsname(msg.getGoodsname());
        order.setNumber(msg.getNumber());
        order.setPrice(msg.getPrice());
        order.setUid(msg.getUid());
        order.setTime(new java.util.Date());
        int orderInserted = shoppingDao.insertOrder(order);
        if (orderInserted == 0) {
            throw new IllegalStateException("订单写入失败");
        }
        Integer cartId = msg.getCartId();
        if (cartId != null && cartId > 0) {
            int deleted = shoppingDao.deleteCart(cartId, msg.getUid());
            if (deleted == 0) {
                throw new IllegalStateException("购物车条目不存在或已删除: cartId=" +
cartId + ", uid=" + msg.getUid());
            }
        }
        log.info("订单处理成功: messageId={}, gid={}, uid={}", messageId,
msg.getGid(), msg.getUid());
    } catch (Exception e) {
        log.error("订单处理失败", e);
        throw new IllegalStateException("订单处理失败", e);
    }
}
```

例图 / Rabbit MQ 消息队列页面



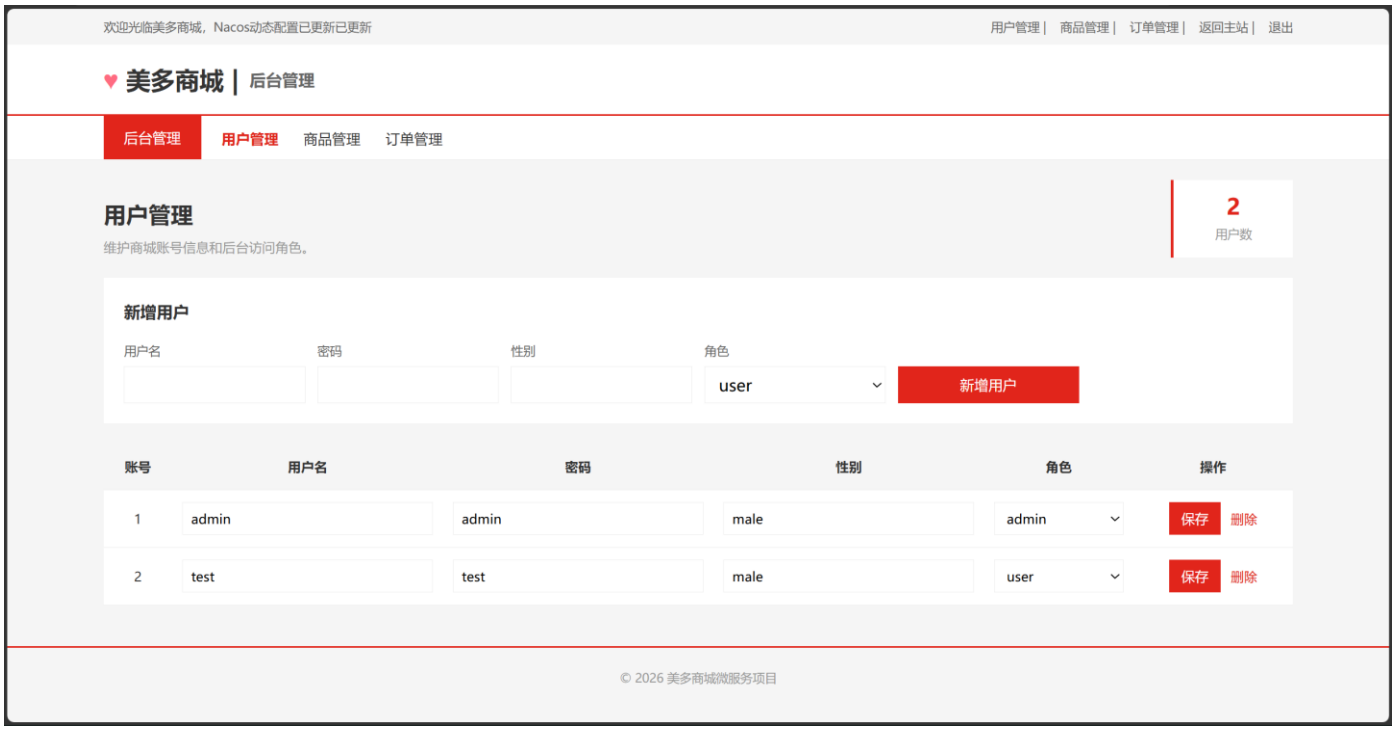
九、 出彩内容：实现管理员后台面板

admin-consumer 模块提供 **/admin/**** 后台页面，通过 Thymeleaf 渲染管理界面，通过 OpenFeign 调用用户服务和商品服务。后台不是简单展示静态页面，而是形成了完整的业务管理闭环：可以维护用户角色、调整商品

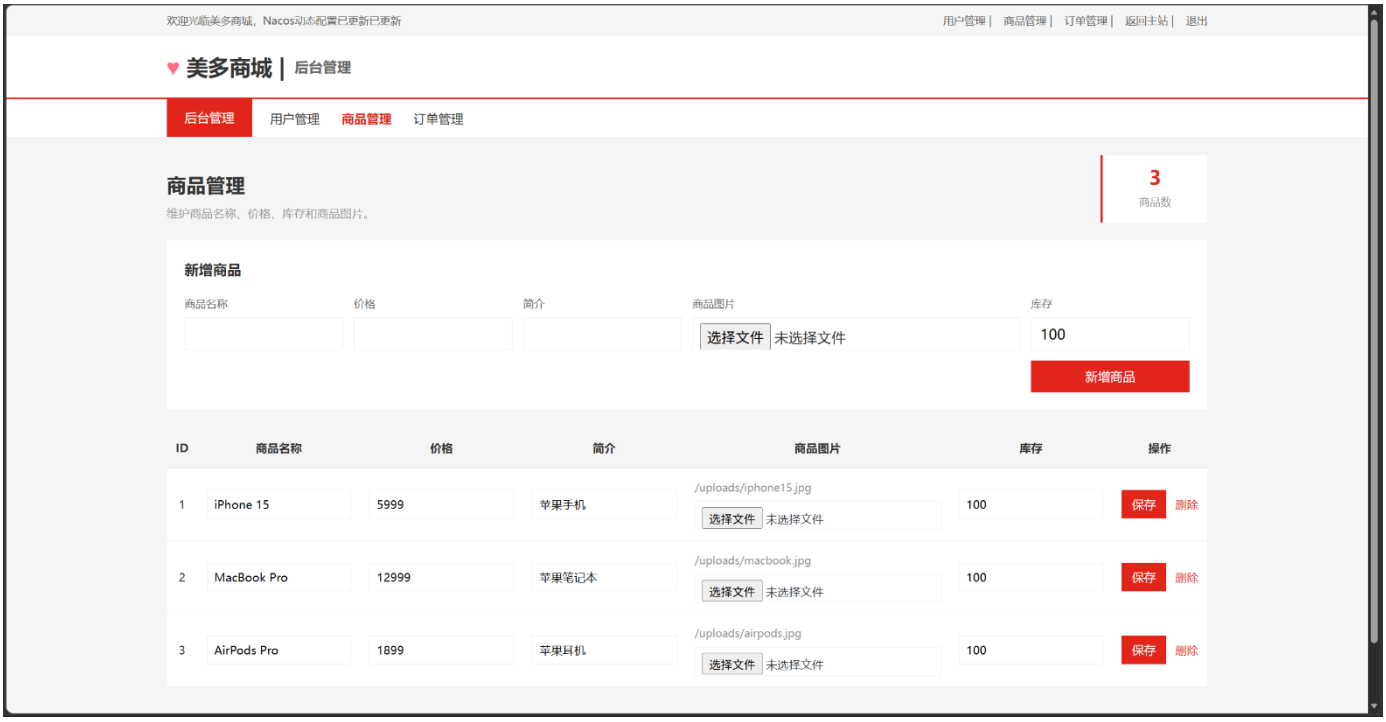
库存、查看订单记录。同时，管理员后台使用 JWT 进行保护，只有 admin（管理员）角色可以访问，这让后台有了基本的权限控制。

有关于管理员后台面板的代码，请参见**代码库**。

例图 / 管理员后台用户管理页



例图 / 管理员后台商品管理页



例图 / 管理员后台订单管理页