**作业报告**

**需求分析**

本项目需要实现一个支持高并发的在线商城系统后端。

**基本功能需求**

1. **用户系统**：包括用户注册、登录退出、用户等级。
2. **商品服务**：
3. 管理员可进行商品维护（商品信息、商品分类、商品价格、商品库存数量）。
4. 普通用户可以进行商品搜索（按名称、分类等搜索筛选，按价格、评价数量等进行排序）和查看。
5. **评论服务**：普通用户可查看商品信息及评论，已经发布评论，按发布时间排序。
6. **订单**：支持订单的创建、查询，库存量不足时下单失败，未支付订单可以撤销。
7. **订单支付**：模拟调用接口完成订单支付，订单不能重复支付

**选做设计实现**

1. 使用**缓存**技术提升系统性能，项目中使用的缓存数据库是Redis
2. 使用**无状态服务**来提升系统性能
3. 使用**负载均衡**技术提升系统性能，项目使用Nginx来实现负载均衡

**性能测试需求**

不少于两个应用节点部署，使用JMeter等工具进行性能测试，测试QPS、响应时间等。

**设计**

**设计方案**

**数据库关系设计**

本系统使用 SQLite 作为关系型数据库，用于存储用户、产品、订单和评论等数据。数据库关系设计如下：

**用户表（users）设计**

用户表（users）用于存储用户的基本信息和状态，包括用户名、密码、邮箱、用户等级和登录令牌。用户表支持用户注册、登录、退出和用户等级管理。

用户表的完整关系模式如下：

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 数据类型 | 约束条件 | 说明 |
| id | INTEGER | PRIMARY KEY AUTOINCREMENT | 用户唯一标识 |
| username | VARCHAR(100) | UNIQUE NOT NULL | 用户名，唯一 |
| password | TEXT | NOT NULL | 用户密码，已加密 |
| email | VARCHAR(100) | UNIQUE | 用户邮箱，唯一 |
| level | INTEGER | DEFAULT 1 | 用户等级，1为默认用户，2为管理员 |
| token | TEXT |  | 用户登录令牌 |
| created\_at | DATETIME | DEFAULT CURRENT\_TIMESTAMP | 创建时间 |
| updated\_at | DATETIME | DEFAULT CURRENT\_TIMESTAMP | 更新时间 |
| deleted\_at | DATETIME |  | 删除时间（软删除） |

SQL 创建表语句：

|  |
| --- |
| SQL CREATE TABLE users (  id INTEGER PRIMARY KEY AUTOINCREMENT,  username VARCHAR(100) UNIQUE NOT NULL,  password TEXT NOT NULL,  email VARCHAR(100) UNIQUE,  level INTEGER DEFAULT 1,  token TEXT,  created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,  updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,  deleted\_at DATETIME ); |

**商品表（products）设计**

商品表（products）用于存储产品的基本信息，包括名称、分类、价格和库存。

商品表的完整关系模式：

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 数据类型 | 约束条件 | 说明 |
| id | INTEGER | PRIMARY KEY AUTOINCREMENT | 产品唯一标识 |
| name | VARCHAR(100) | NOT NULL | 产品名称 |
| category | VARCHAR(100) |  | 产品分类 |
| price | FLOAT | NOT NULL | 产品价格 |
| stock | INTEGER | NOT NULL | 产品库存 |
| created\_at | DATETIME | DEFAULT CURRENT\_TIMESTAMP | 创建时间 |
| updated\_at | DATETIME | DEFAULT CURRENT\_TIMESTAMP | 更新时间 |
| deleted\_at | DATETIME |  | 删除时间（软删除） |

SQL 建表语句：

|  |
| --- |
| SQL CREATE TABLE products (  id INTEGER PRIMARY KEY AUTOINCREMENT,  name VARCHAR(100) NOT NULL,  category VARCHAR(100),  price FLOAT NOT NULL,  stock INTEGER NOT NULL,  created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,  updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,  deleted\_at DATETIME ); |

**订单表（orders）设计**

订单表（orders）用于存储订单的详细信息，包括用户ID、产品ID、购买数量、总金额和订单状态。

订单表的完整关系模式：

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 数据类型 | 约束条件 | 说明 |
| id | INTEGER | PRIMARY KEY AUTOINCREMENT | 订单唯一标识 |
| user\_id | INTEGER | NOT NULL | 用户ID（外键） |
| product\_id | INTEGER | NOT NULL | 产品ID（外键） |
| quantity | INTEGER | NOT NULL | 购买数量 |
| total | FLOAT | NOT NULL | 总金额 |
| status | VARCHAR(20) | DEFAULT 'unpaid' | 订单状态（未支付、已支付、已取消） |
| created\_at | DATETIME | DEFAULT CURRENT\_TIMESTAMP | 创建时间 |
| updated\_at | DATETIME | DEFAULT CURRENT\_TIMESTAMP | 更新时间 |
| deleted\_at | DATETIME |  | 删除时间（软删除） |

SQL 建表语句：

|  |
| --- |
| SQL CREATE TABLE orders (  id INTEGER PRIMARY KEY AUTOINCREMENT,  user\_id INTEGER NOT NULL,  product\_id INTEGER NOT NULL,  quantity INTEGER NOT NULL,  total FLOAT NOT NULL,  status VARCHAR(20) DEFAULT 'unpaid',  created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,  updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,  deleted\_at DATETIME ); |

**评论表（comments）设计**

评论表的完整关系模式：

|  |  |  |  |
| --- | --- | --- | --- |
| 字段名 | 数据类型 | 约束条件 | 说明 |
| id | INTEGER | PRIMARY KEY AUTOINCREMENT | 评论唯一标识 |
| user\_id | INTEGER | NOT NULL | 用户ID（外键） |
| product\_id | INTEGER | NOT NULL | 产品ID（外键） |
| content | TEXT | NOT NULL | 评论内容 |
| created\_at | DATETIME | DEFAULT CURRENT\_TIMESTAMP | 创建时间 |
| updated\_at | DATETIME | DEFAULT CURRENT\_TIMESTAMP | 更新时间 |
| deleted\_at | DATETIME |  | 删除时间（软删除） |

SQL 建表语句：

|  |
| --- |
| SQL CREATE TABLE comments (  id INTEGER PRIMARY KEY AUTOINCREMENT,  user\_id INTEGER NOT NULL,  product\_id INTEGER NOT NULL,  content TEXT NOT NULL,  created\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,  updated\_at DATETIME DEFAULT CURRENT\_TIMESTAMP,  deleted\_at DATETIME ); |

**性能需求设计**

为了满足高并发访问和快速响应的性能需求，本系统采用以下设计和技术：

1. **缓存技术**：使用 Redis 作为缓存数据库，缓存频繁访问的数据，如产品列表、用户信息等，减少对 SQLite 数据库的直接访问，提升系统读取性能。
2. **负载均衡**：使用 NGINX 作为反向代理服务器，实现负载均衡，将请求分发到多个应用实例，提升系统的并发处理能力。
3. **无状态应用**：设计无状态应用，将所有状态信息存储在 Redis 或数据库中，服务实例可随时重启或扩展，保证系统的高可用性。
4. **自动扩展**：在高负载情况下，可以通过增加应用实例的方式进行自动扩展，确保系统在高并发访问下仍能保持稳定运行。
5. **消息中间件**：未来可扩展使用消息中间件（如 RabbitMQ、Kafka）处理异步任务，实现下订单请求和下订单处理、支付请求和支付处理的解耦，订单10分钟未支付自动取消。

**部署方案**

本系统采用 Docker 容器化部署，使用 Docker Compose 管理多个容器，确保各个服务独立运行，互不影响。部署方案如下：

1. **服务划分**：将系统划分为多个独立的服务，包括用户服务、产品服务、订单服务、评论服务、Redis 服务和 NGINX 服务。
2. **Docker 镜像**：每个服务分别打包成 Docker 镜像，使用 Dockerfile 定义构建过程。
3. **Docker Compose**：使用 Docker Compose 定义服务间的依赖关系和网络配置，确保各个服务可以互相通信。
4. **共享数据库文件**：通过 Docker 卷挂载方式，将本地的 SQLite 数据库文件挂载到多个应用实例，确保数据一致性。
5. **负载均衡**：使用 NGINX 作为反向代理服务器，实现负载均衡，将请求分发到多个应用实例，提升系统的并发处理能力。

**实现**

**接口实现**

1. **用户注册接口**

逻辑说明：

* 用户输入用户名（username）和用户密码（password）进行注册。
* 服务器验证用户名的唯一性。
* 服务器对用户密码进行加密存储。
* 注册成功后，服务器生成一个唯一的token返回给用户。

代码实现：

|  |
| --- |
| Go router.POST("/register", func(c \*gin.Context) {  var user User  if err := c.ShouldBindJSON(&user); err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})  return  }  // 密码加密  hashedPassword, \_ := bcrypt.GenerateFromPassword([]byte(user.Password), bcrypt.DefaultCost)  user.Password = string(hashedPassword)  // 默认用户等级为1  user.Level = 1  // 生成用户登录令牌  uuid := uuid.New()  user.Token = uuid.String()  if err := db.Create(&user).Error; err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": "Registration failed, please try again."})  return  }  c.JSON(http.StatusOK, gin.H{"data": user.Token}) }) |

1. **用户登录接口**

逻辑说明：

* 用户输入用户名（username）和用户密码（password）进行登录。
* 服务器验证用户名和密码的正确性。
* 登录成功后，服务器生成一个新的token返回给用户。

代码实现：

|  |
| --- |
| Go router.POST("/login", func(c \*gin.Context) {  var user User  var loginUser User  if err := c.ShouldBindJSON(&loginUser); err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})  return  }  if err := db.Where("username = ?", loginUser.Username).First(&user).Error; err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": "User does not exist."})  return  }  if err := bcrypt.CompareHashAndPassword([]byte(user.Password), []byte(loginUser.Password)); err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": "Incorrect password."})  return  }  uuid := uuid.New()  user.Token = uuid.String()  db.Save(&user)  c.JSON(http.StatusOK, gin.H{"data": user.Token}) }) |

1. **用户退出接口**

逻辑说明：

* 用户发送当前登录的token。
* 服务器验证token的有效性。
* 退出登录成功后，服务器删除用户的token。

代码实现：

|  |
| --- |
| Go router.POST("/logout", func(c \*gin.Context) {  var user User  if err := c.ShouldBindJSON(&user); err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})  return  }  if err := db.Where("token = ?", user.Token).First(&user).Error; err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": "User does not exist."})  return  }  user.Token = ""  db.Save(&user)  c.JSON(http.StatusOK, gin.H{"data": "Logout successful."}) }) |

1. **管理员添加商品接口**

逻辑说明：

* 管理员输入产品信息（name、category、price、stock）添加产品。
* 服务器验证管理员权限。
* 产品信息存储到数据库中。

代码实现：

|  |
| --- |
| Go router.POST("/admin/product", func(c \*gin.Context) {  authHeader := c.GetHeader("Authorization")  tokenParts := strings.Split(authHeader, " ")  if CheckPermission(tokenParts[1]) == 1 {  c.JSON(http.StatusBadRequest, gin.H{"error": "Permission denied."})  return  }  var product Product  if err := c.ShouldBindJSON(&product); err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})  return  }  if err := db.Create(&product).Error; err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": "Product creation failed, please try again."})  return  }  c.JSON(http.StatusOK, gin.H{"data": product}) }) |

1. **管理员获取商品接口**

逻辑说明：

* 管理员根据产品ID获取产品信息。
* 服务器验证管理员权限。
* 服务器从数据库中查询产品信息并返回。

代码实现：

|  |
| --- |
| Go router.GET("/admin/product/:id", func(c \*gin.Context) {  authHeader := c.GetHeader("Authorization")  tokenParts := strings.Split(authHeader, " ")  if CheckPermission(tokenParts[1]) == 1 {  c.JSON(http.StatusBadRequest, gin.H{"error": "Permission denied."})  return  }  var product Product  if err := db.Where("id = ?", c.Param("id")).First(&product).Error; err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": "Product does not exist."})  return  }  c.JSON(http.StatusOK, gin.H{"data": product}) }) |

1. **管理员删除商品接口**

逻辑说明：

* 管理员根据产品ID删除产品。
* 服务器验证管理员权限。
* 服务器从数据库中删除产品。

代码实现：

|  |
| --- |
| Go router.DELETE("/admin/product/:id", func(c \*gin.Context) {  authHeader := c.GetHeader("Authorization")  tokenParts := strings.Split(authHeader, " ")  if CheckPermission(tokenParts[1]) == 1 {  c.JSON(http.StatusBadRequest, gin.H{"error": "Permission denied."})  return  }  var product Product  if err := db.Where("id = ?", c.Param("id")).First(&product).Error; err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": "Product does not exist."})  return  }  db.Delete(&product)  c.JSON(http.StatusOK, gin.H{"data": "Product deletion successful."}) }) |

1. **用户获取商品接口**

逻辑说明：

* 用户获取产品列表。
* 服务器首先尝试从Redis缓存中获取产品列表。
* 如果缓存不存在，查询数据库并将结果存储到Redis缓存。

代码实现：

|  |
| --- |
| Go router.GET("/product", func(c \*gin.Context) {  val, err := client.Get("products").Result()  if errors.Is(err, redis.Nil) {  var products []Product  if err := db.Find(&products).Error; err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": "Error retrieving products."})  return  }  productsJson, \_ := json.Marshal(products)  err = client.Set("products", productsJson, 5\*time.Second).Err()  if err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": "Error caching products."})  return  }  c.JSON(http.StatusOK, gin.H{"data": products})  } else if err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": "Error retrieving products from cache."})  } else {  var products []Product  json.Unmarshal([]byte(val), &products)  c.JSON(http.StatusOK, gin.H{"data": products})  } }) |

1. **用户下订单接口**

逻辑说明：

* 用户创建订单，输入产品ID和购买数量。
* 服务器验证用户登录状态。
* 服务器检查产品库存。
* 订单信息存储到数据库中。

代码实现：

|  |
| --- |
| Go router.POST("/order", func(c \*gin.Context) {  authHeader := c.GetHeader("Authorization")  tokenParts := strings.Split(authHeader, " ")  user := CheckLogin(tokenParts[1])  if user == 0 {  c.JSON(http.StatusBadRequest, gin.H{"error": "Permission denied."})  return  }  var order Order  if err := c.ShouldBindJSON(&order); err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})  return  }  var product Product  if err := db.First(&product, order.ProductID).Error; err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": "Product does not exist."})  return  }  if product.Stock < order.Quantity {  c.JSON(http.StatusBadRequest, gin.H{"error": "Insufficient product stock."})  return  }  product.Stock -= order.Quantity  order.Total = float64(order.Quantity) \* product.Price  order.Status = "unpaid"  order.UserID = user  db.Save(&product)  if err := db.Create(&order).Error; err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": "Order creation failed, please try again."})  return  }  c.JSON(http.StatusOK, gin.H{"data": order}) }) |

1. **用户获取订单接口**

逻辑说明：

* 用户获取自己的订单列表。
* 服务器验证用户登录状态。
* 服务器从数据库中查询用户的订单并返回。

代码实现：

|  |
| --- |
| Go router.GET("/order", func(c \*gin.Context) {  authHeader := c.GetHeader("Authorization")  tokenParts := strings.Split(authHeader, " ")  user := CheckLogin(tokenParts[1])  if user == 0 {  c.JSON(http.StatusBadRequest, gin.H{"error": "Permission denied."})  return  }  var orders []Order  if err := db.Where("user\_id = ?", user).Order("created\_at desc").Find(&orders).Error; err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": "Error retrieving orders."})  return  }  c.JSON(http.StatusOK, gin.H{"data": orders}) }) |

1. **用户支付订单接口**

逻辑说明：

* 用户支付订单，输入订单ID。
* 服务器验证用户登录状态。
* 服务器更新订单状态为已支付。

代码实现：

|  |
| --- |
| Go router.POST("/order/pay", func(c \*gin.Context) {  authHeader := c.GetHeader("Authorization")  tokenParts := strings.Split(authHeader, " ")  if CheckLogin(tokenParts[1]) == 0 {  c.JSON(http.StatusBadRequest, gin.H{"error": "Permission denied."})  return  }  var payInfo struct {  OrderID uint  }  if err := c.ShouldBindJSON(&payInfo); err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})  return  }  var order Order  if err := db.First(&order, payInfo.OrderID).Error; err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": "Order does not exist."})  return  }  if order.Status != "unpaid" {  c.JSON(http.StatusBadRequest, gin.H{"error": "Order has been paid or cancelled."})  return  }  order.Status = "paid"  db.Save(&order)  c.JSON(http.StatusOK, gin.H{"data": "Payment successful."}) }) |

1. **获取商品评论接口**

逻辑说明：

* 用户获取产品的评论列表，输入产品ID。
* 服务器验证用户登录状态。
* 服务器从数据库中查询产品的评论并返回。

代码实现：

|  |
| --- |
| Go router.GET("/comment/:productID", func(c \*gin.Context) {  authHeader := c.GetHeader("Authorization")  tokenParts := strings.Split(authHeader, " ")  if CheckLogin(tokenParts[1]) == 0 {  c.JSON(http.StatusBadRequest, gin.H{"error": "Permission denied."})  return  }  var comments []Comment  if err := db.Where("product\_id = ?", c.Param("productID")).Order("created\_at desc").Find(&comments).Error; err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": "Error retrieving comments."})  return  }  c.JSON(http.StatusOK, gin.H{"data": comments}) }) |

1. **添加商品评论接口**

逻辑说明：

* 用户对产品添加评论，输入产品ID和评论内容。
* 服务器验证用户登录状态。
* 评论信息存储到数据库中。

代码实现：

|  |
| --- |
| Go router.POST("/comment", func(c \*gin.Context) {  authHeader := c.GetHeader("Authorization")  tokenParts := strings.Split(authHeader, " ")  user := CheckLogin(tokenParts[1])  if user == 0 {  c.JSON(http.StatusBadRequest, gin.H{"error": "Permission denied."})  return  }  var comment Comment  if err := c.ShouldBindJSON(&comment); err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": err.Error()})  return  }  comment.UserID = user  if err := db.Create(&comment).Error; err != nil {  c.JSON(http.StatusBadRequest, gin.H{"error": "Comment creation failed, please try again."})  return  }  c.JSON(http.StatusOK, gin.H{"data": comment}) }) |

**部署实现**

项目使用本地单机部署和Docker多服务部署两种方式。

**本地单机部署**

1. 使用GoLand打开项目文件夹backend
2. 打开设置，选择Go->GOROOT为1.22版本
3. 本地启动Redis，端口号为6379
4. 运行go build backend

**Docker多服务部署**

使用 Dockerfile 来构建程序的 Docker 镜像。在 Dockerfile中使用与本地化测试相同的go语言镜像版本，安装必要的构建工具和数据库工具，设置go语言代理，将本地的go项目构建文件上传到docker，之后进行构建并启动应用程序：

|  |
| --- |
| Dockerfile *# 使用官方的Go镜像作为基础镜像* FROM golang:1.22.4-alpine3.20   *# 安装必要的构建工具和 SQLite3 库* RUN apk add --no-cache gcc musl-dev sqlite-dev  *# 设置工作目录* WORKDIR /app  *# 设置 Go Proxy* ENV GOPROXY=https://goproxy.cn,direct ENV CGO\_ENABLED=1  *# 复制 go.mod 和 go.sum 文件到工作目录* COPY go.mod go.sum ./  *# 下载依赖项* RUN go mod download  *# 复制当前目录内容到工作目录* COPY . .  *# 显示当前工作目录和文件* RUN pwd && ls -al  *# 构建应用程序* RUN go build -v -o main .  *# 暴露应用程序端口* EXPOSE 8080  *# 启动应用程序* CMD ["./main"] |

创建一个docker-compose.yml文件来定义服务和依赖关系。在其中定义使用的redis缓存、nginx负载均衡、以及启动的多个后端服务（在此启动了四个后端服务，分别是app1、app2、app3、app4），都使用上面的Dockerfile来构建：

|  |
| --- |
| YAML version: '3'  services:  redis:  image: redis:latest  container\_name: redis  ports:  - "6379:6379"   app1:  build: .  container\_name: app1  volumes:  - ./gorm.db:/app/data/gorm.db  ports:  - "8081:8080"  depends\_on:  - redis   app2:  build: .  container\_name: app2  volumes:  - ./gorm.db:/app/data/gorm.db  ports:  - "8082:8080"  depends\_on:  - redis    app3:  build: .  container\_name: app3  volumes:  - ./gorm.db:/app/data/gorm.db  ports:  - "8083:8080"  depends\_on:  - redis   app4:  build: .  container\_name: app4  volumes:  - ./gorm.db:/app/data/gorm.db  ports:  - "8084:8080"  depends\_on:  - redis   nginx:  image: nginx:latest  container\_name: nginx  ports:  - "80:80"  volumes:  - ./nginx.conf:/etc/nginx/nginx.conf  depends\_on:  - app1  - app2 |

修改Redis数据库连接信息，由于使用Docker部署Redis，实现容器间的相互通信，修改main.go第53行为Addr: "redis:6379",来重新配置Redis。修改SQLite数据库连接信息，修改main.go第78行为db, \_ = gorm.Open(sqlite.Open("/app/data/gorm.db"), &gorm.Config{})，代表多个服务共享同一个数据库。

使用nginx.conf文件来配置负载均衡。定义每个工作进程可以同时打开的最大连接数1024。定义一个名为backend的上游服务器池，服务器池中包含了四个服务器条目app1、app2、app3、app4。nginx默认使用轮询的方式实现负载均衡。因此，在收到的请求中，有四分之一分别分配给app1、app2、app3、app4。

|  |
| --- |
| YAML events {  worker\_connections 1024*;* }  http {  upstream backend {  server app1:8080*;*  server app2:8080*;*  server app3:8080*;*  server app4:8080*;*  }   server {  listen 80*;*   location / {  proxy\_pass http://backend*;*  proxy\_set\_header Host $host*;*  proxy\_set\_header X-Real-IP $remote\_addr*;*  proxy\_set\_header X-Forwarded-For $proxy\_add\_x\_forwarded\_for*;*  proxy\_set\_header X-Forwarded-Proto $scheme*;*  }  } } |

在配置好上述文件后（本地需要安装 Docker 和 Docker-compose），在项目文件夹目录中启动终端来进行部署：

|  |
| --- |
| Bash docker-compose up --build |

使用docker ps命令可以查看所有启动的容器，包含一个负载均衡服务器nginx，四个后端请求处理服务器app1~app4，一个redis缓存服务器。注意到，nginx的测试端口是80，因此我们需要将测试中的请求端口修改为80。



第二次启动服务：

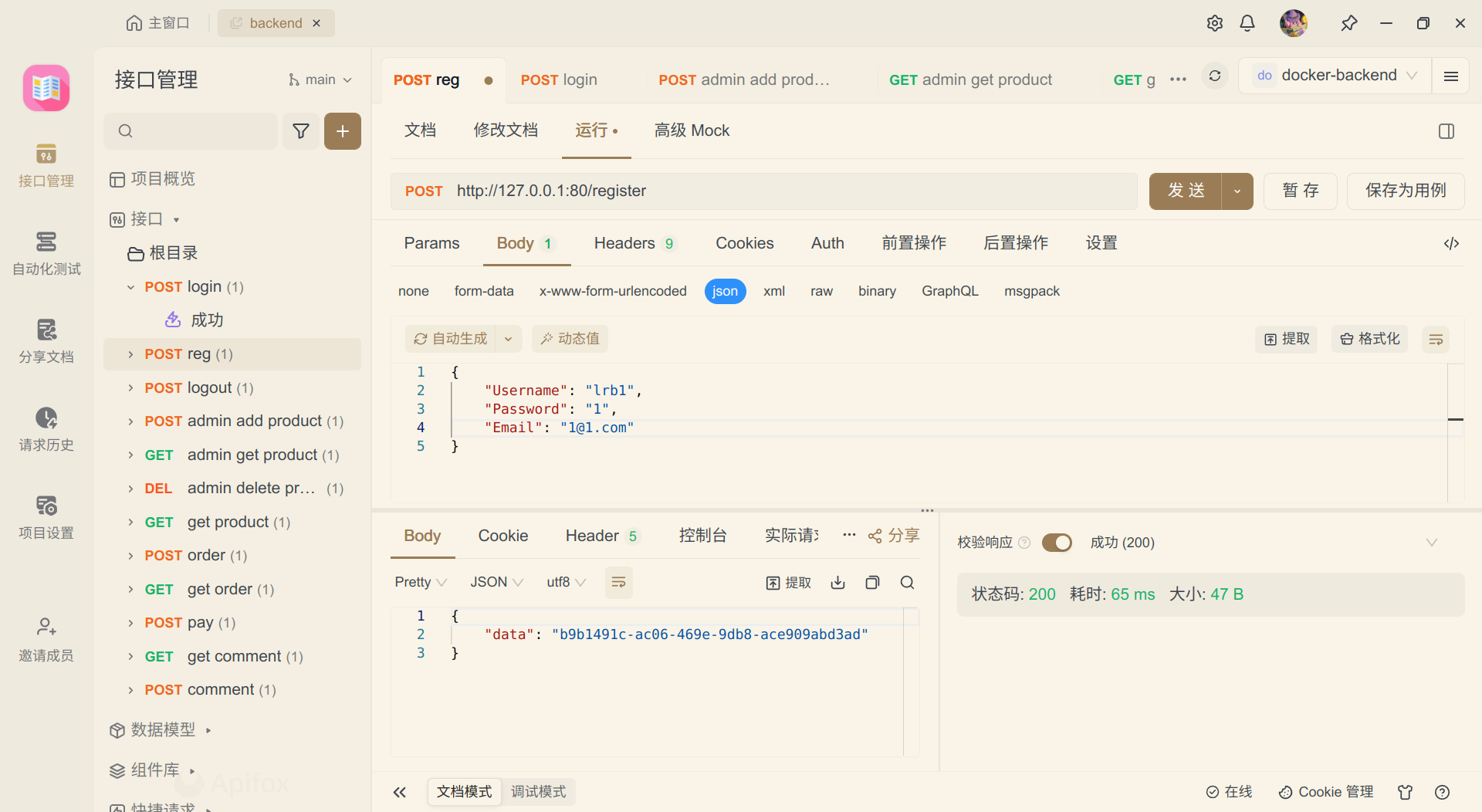
|  |
| --- |
| Bash docker-compose up |

**测试**

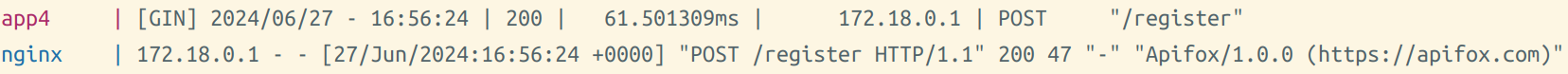
**接口功能测试**

1. **用户注册接口测试**

请求体中需要包含用户名、密码、邮箱，响应中有服务器端生成的 token：

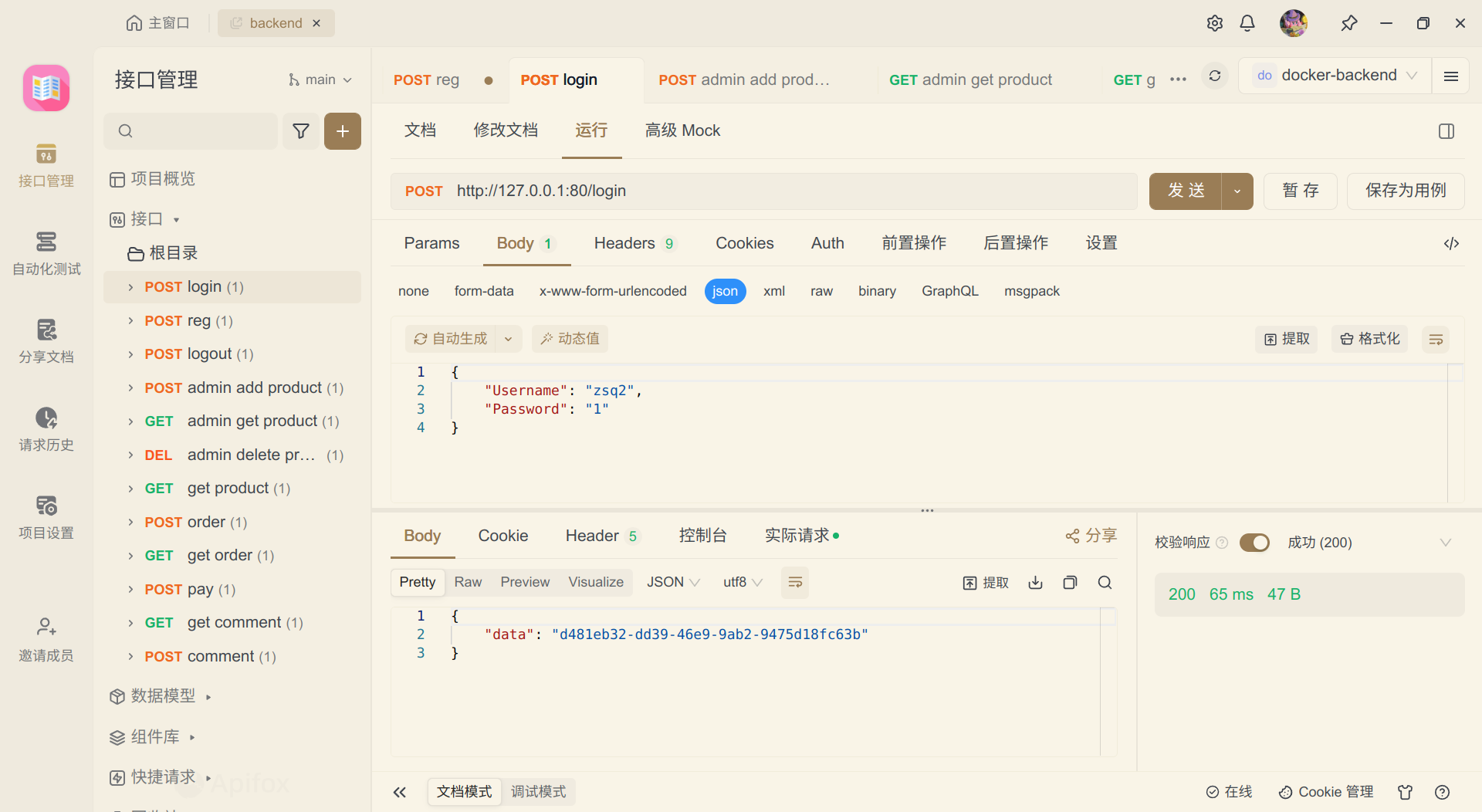


运行输出中可以看到 Nginx 将本次请求分配给 app4 来处理：

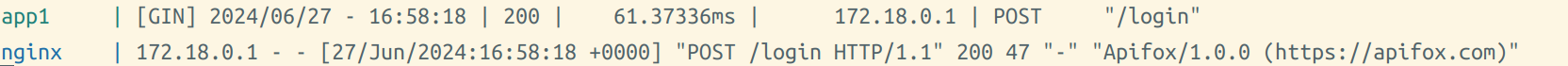


1. **用户登录接口测试**

请求体中需要包含用户名和密码，响应中有服务器生成的 token：

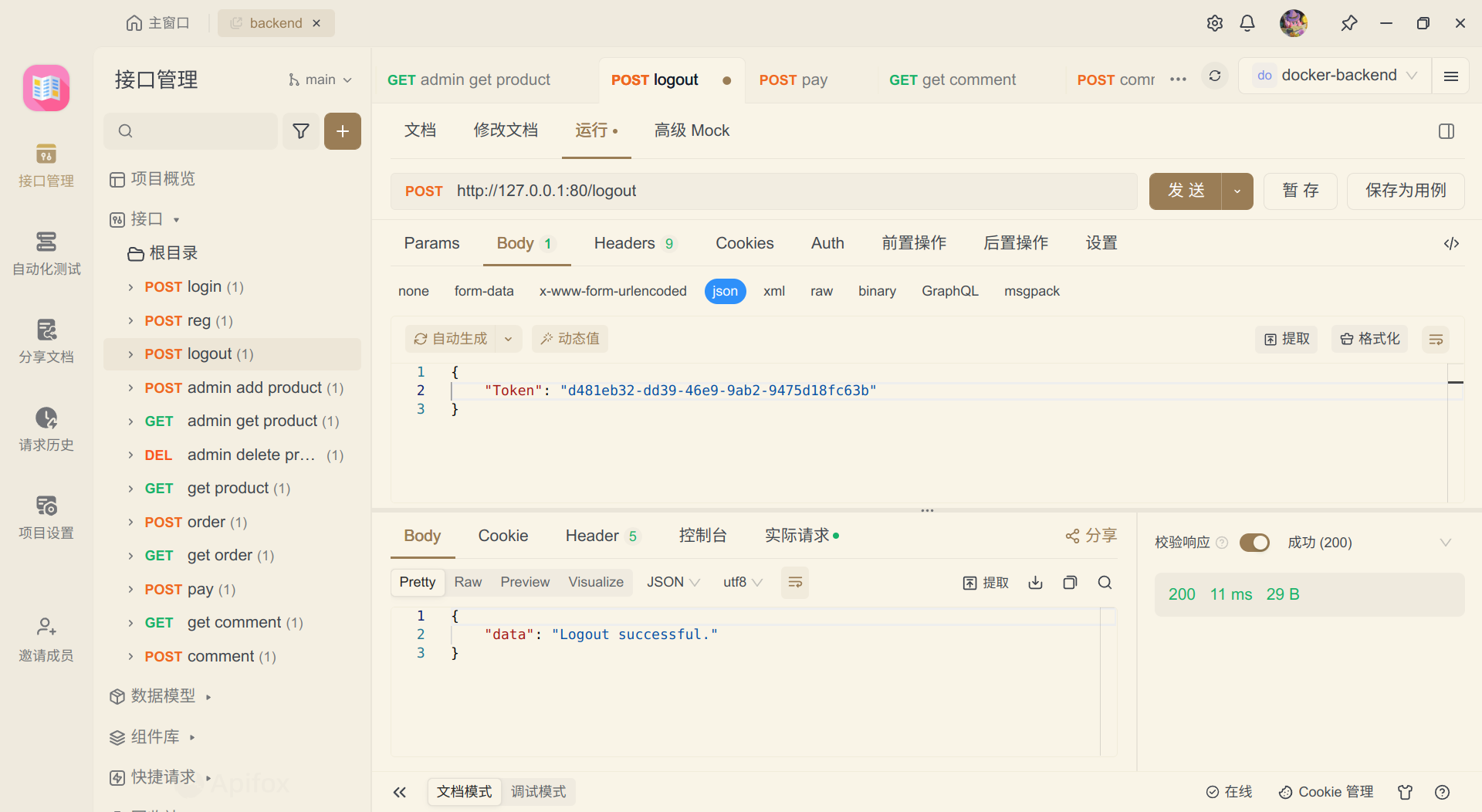


运行输出中可以看到 Nginx 将本次请求分配给 app1 来处理：

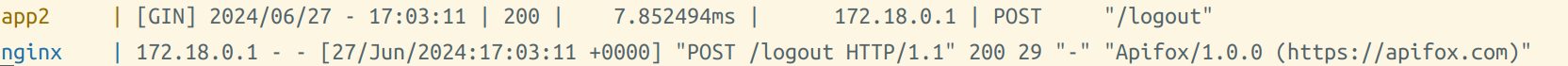


1. **用户退出接口测试**

请求体中需要包含用户 token，响应中含有登出成功提示，服务器端删除该用户 token：

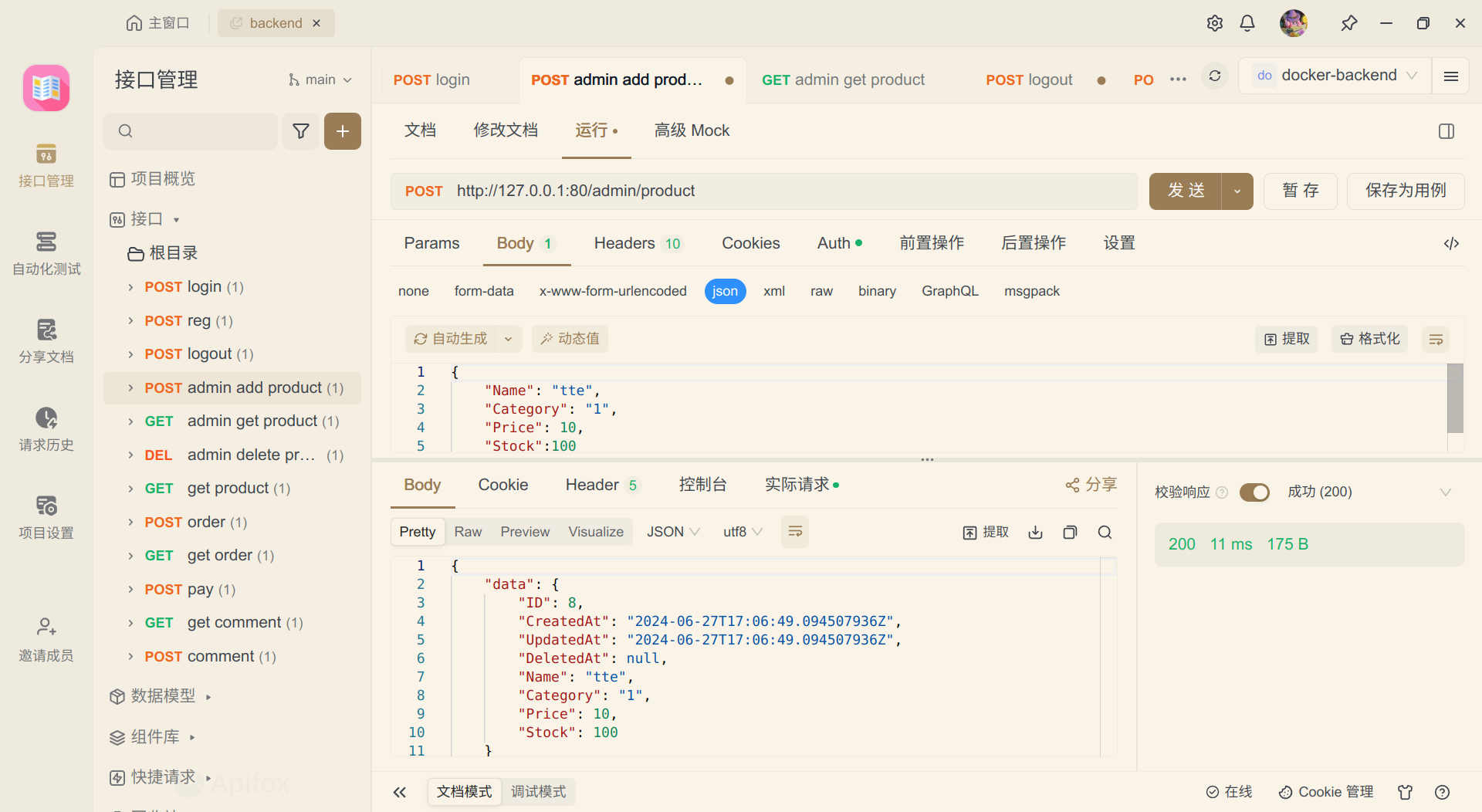


运行输出中可以看到 Nginx 将本次请求分配给 app2 来处理：

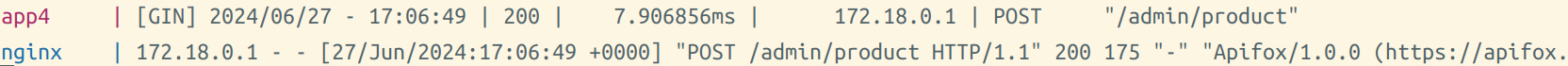


1. **管理员添加商品接口测试**

请求体中需要包含商品的商品名、商品分类、商品价格、商品库存，在请求认证中需要包含登录成功获取的 token，请求响应中含有添加成功的商品信息：

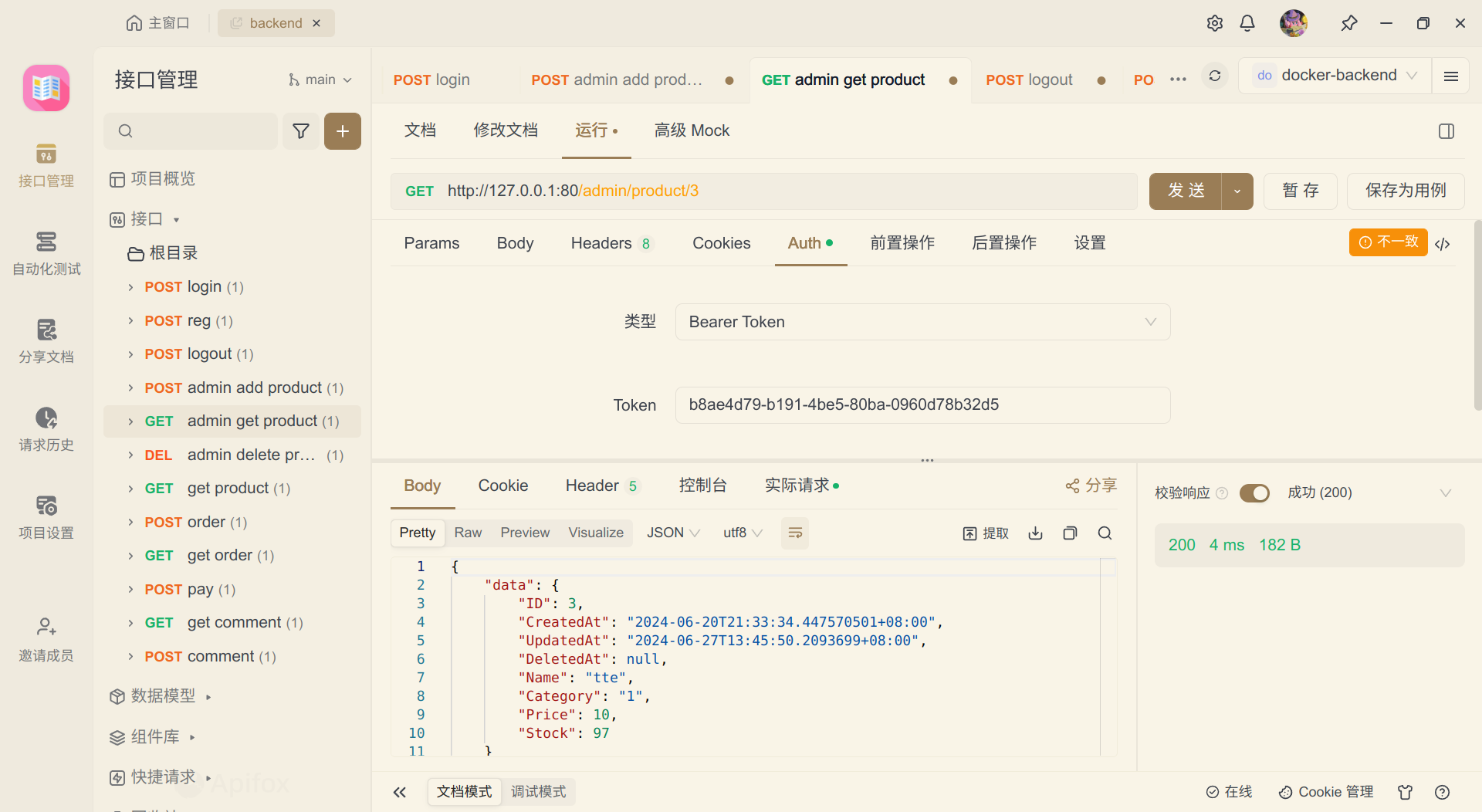


运行输出中可以看到 Nginx 将本次请求分配给 app4 来处理：

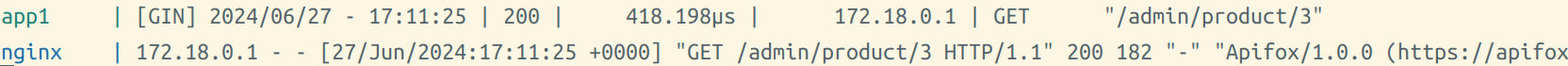


1. **管理员获取商品接口测试**

请求 url 中需要包含商品 ID，请求认证中需要包含登录成功获取的 token，请求响应中包含对应 ID的商品详情：

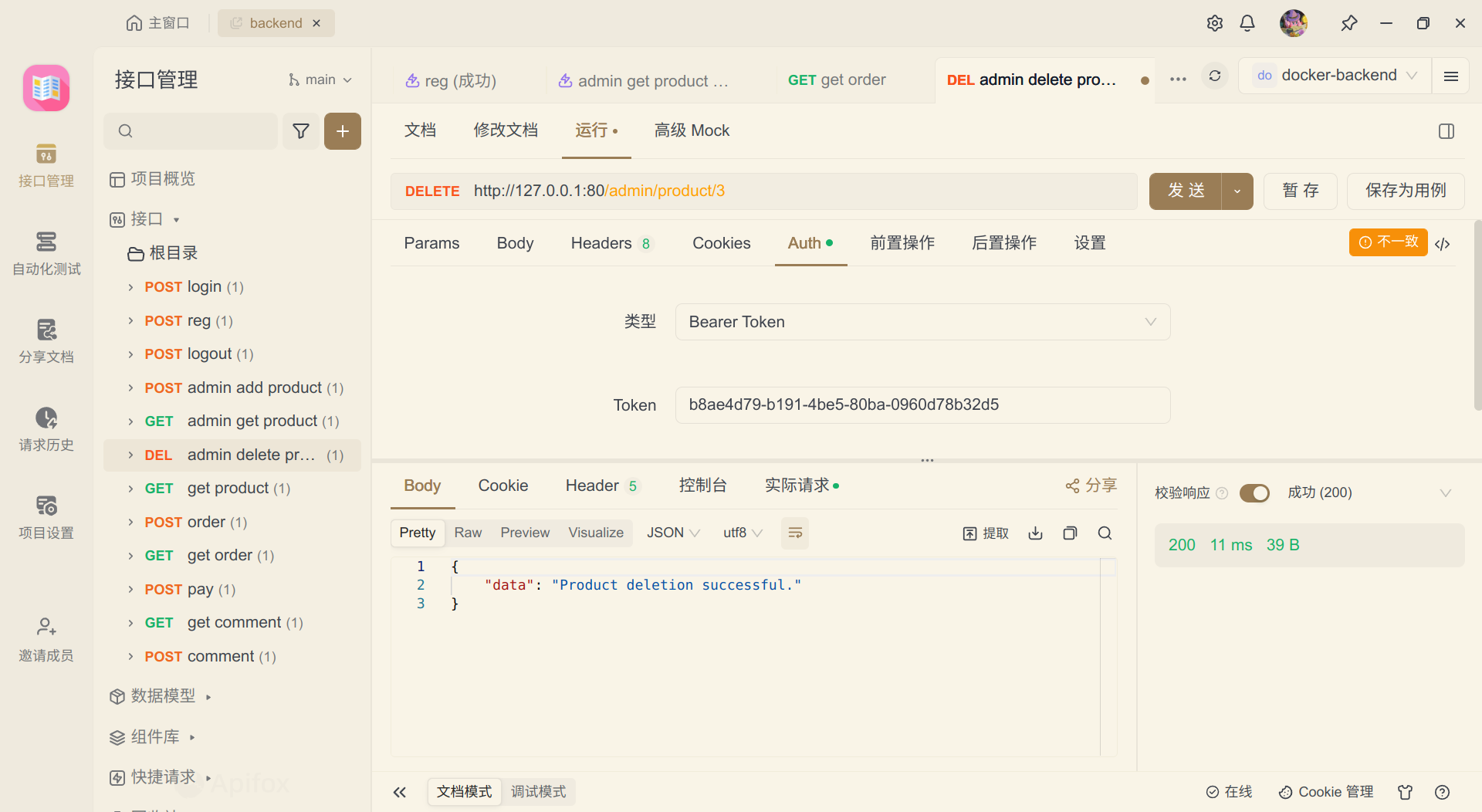


运行输出中可以看到 Nginx 将本次请求分配给 app1 来处理：

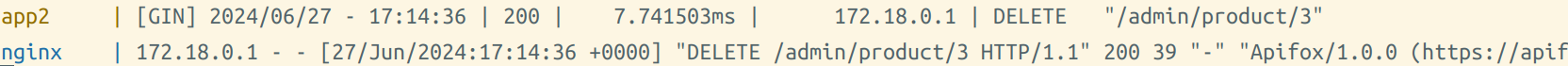


1. **管理员删除商品接口测试**

请求 url 中需要包含商品 ID，请求认证中需要包含登录成功获取的 token，请求响应中包含删除成功的提示：



运行输出中可以看到 Nginx 将本次请求分配给 app2 来处理：

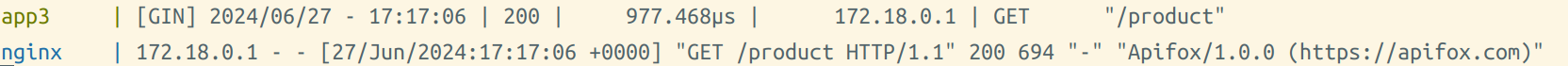


1. **用户获取商品接口**

请求认证中需要包含登录成功获取的 token，请求响应中包含当前所有商品列表，商品按照创建时间进行排序：



运行输出中可以看到 Nginx 将本次请求分配给 app3 来处理：

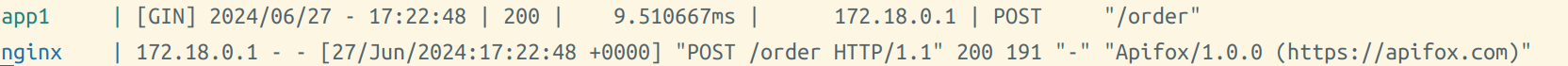


1. **用户下订单接口**

请求体中需要包含商品 ID，商品数量，请求认证需要包含登录成功获取的 token，请求响应中包含添加成功商品的详细信息，订单与用户 ID 和商品 ID 绑定：

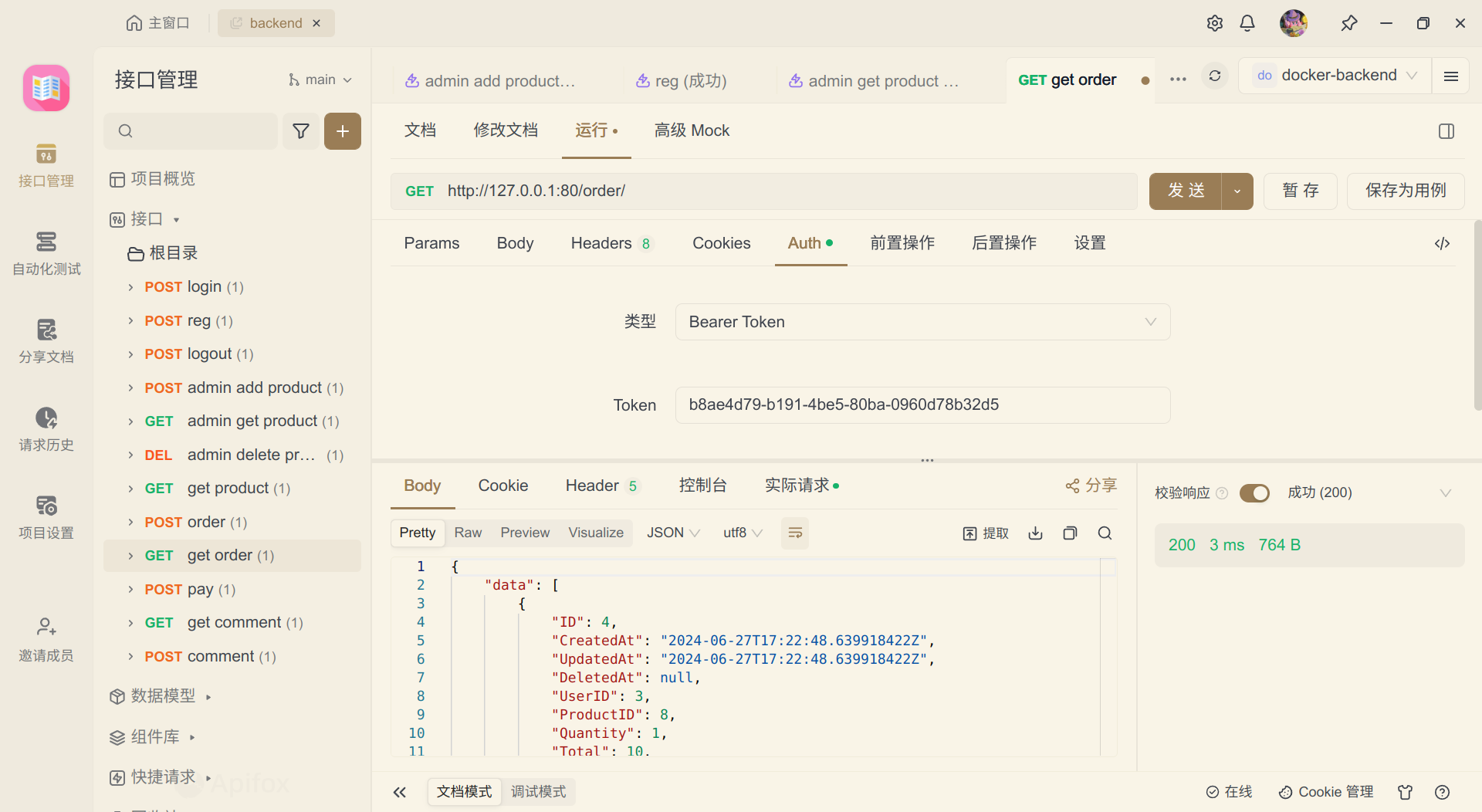


运行输出中可以看到 Nginx 将本次请求分配给 app1 来处理：

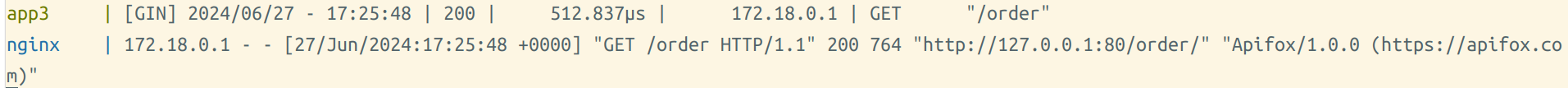


1. **用户获取订单接口**

请求认证需要包含登录成功获取的 token，请求响应中包含当前购物车的清单，按加入购物车时间进行排序：

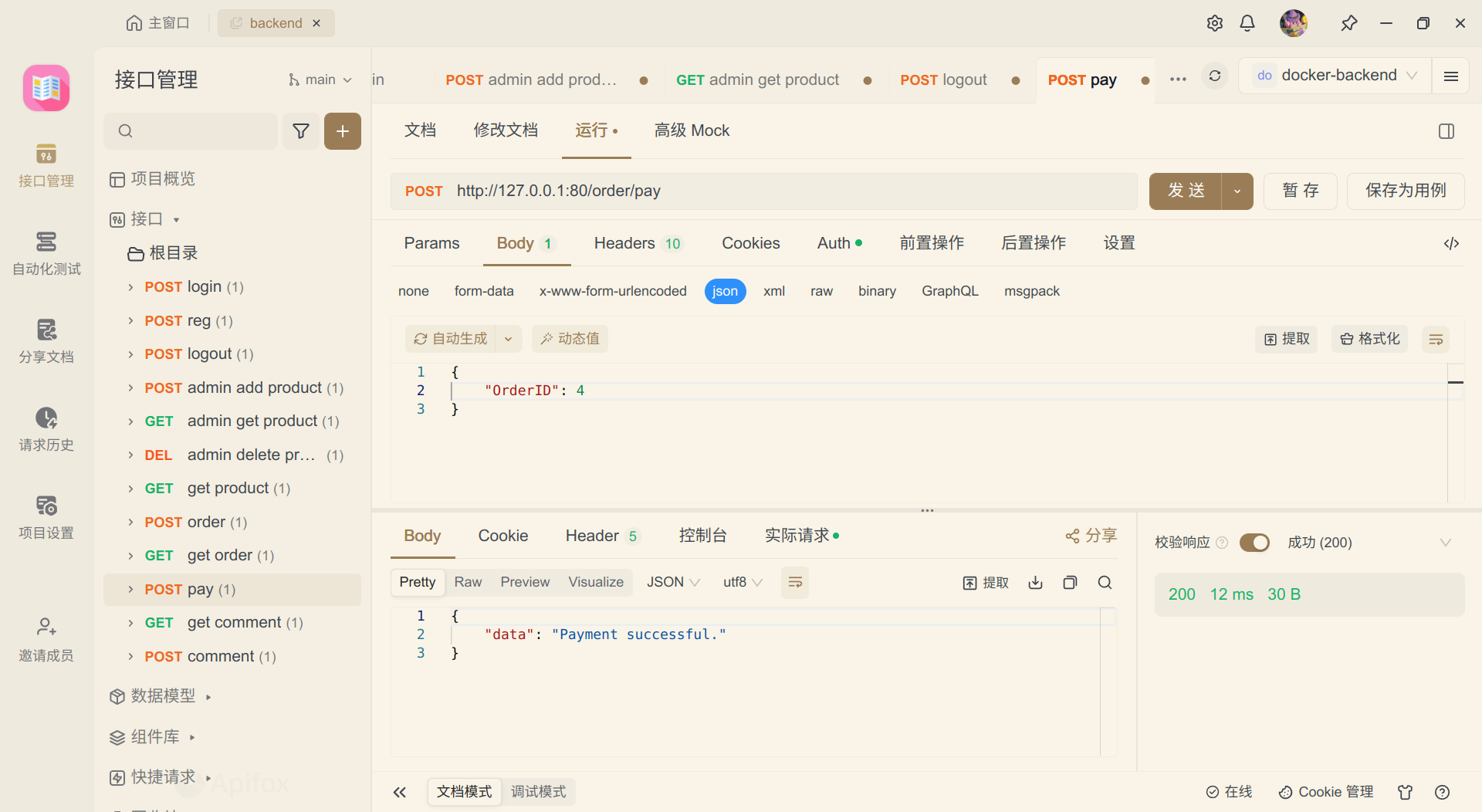


运行输出中可以看到 Nginx 将本次请求分配给 app3 来处理：

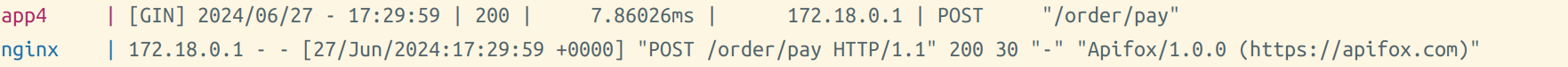


1. **用户支付订单接口**

请求体中需要包含订单 ID，请求认证需要包含登录成功获取的 token，请求响应中包含支付成功提示：



运行输出中可以看到 Nginx 将本次请求分配给 app4 来处理：

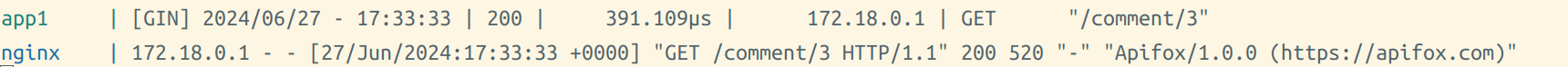


1. **获取商品评论接口**

请求 url 中需要包含商品 ID，请求认证中需要包含登录成功获取的 token，请求响应包含该商品所有评论详情，按照评论添加时间排序：

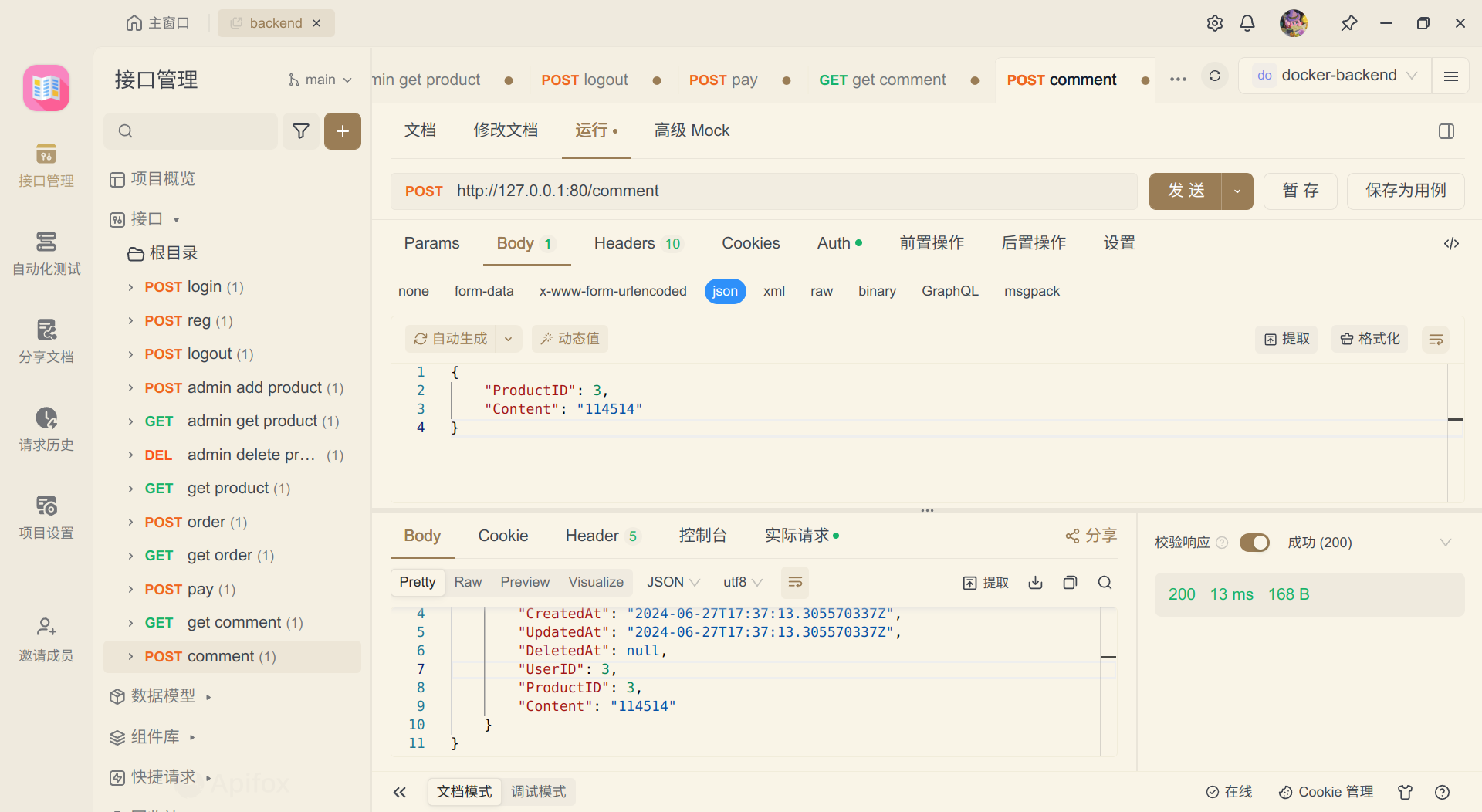


运行输出中可以看到 Nginx 将本次请求分配给 app1 来处理：



1. **添加商品评论接口**

请求体中需要包含商品 ID 和评论内容，请求认证中需要包含登录成功获取的 token，请求响应包含该评论详情，评论与用户 ID 和商品 ID 绑定：



运行输出中可以看到 Nginx 将本次请求分配给 app2 来处理：



**接口性能测试**

**Redis缓存测试**

针对实现了Redis缓存功能的用户获取商品接口进行测试。

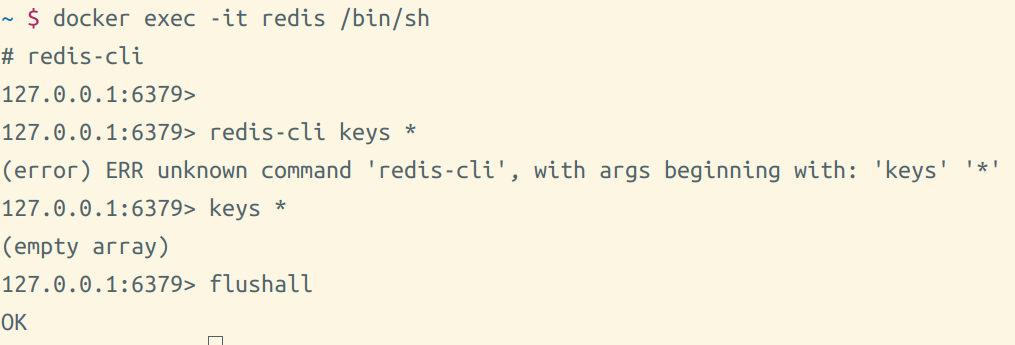
1. 首先进入Redis的容器。

|  |
| --- |
| Bash docker exec -it redis /bin/sh |

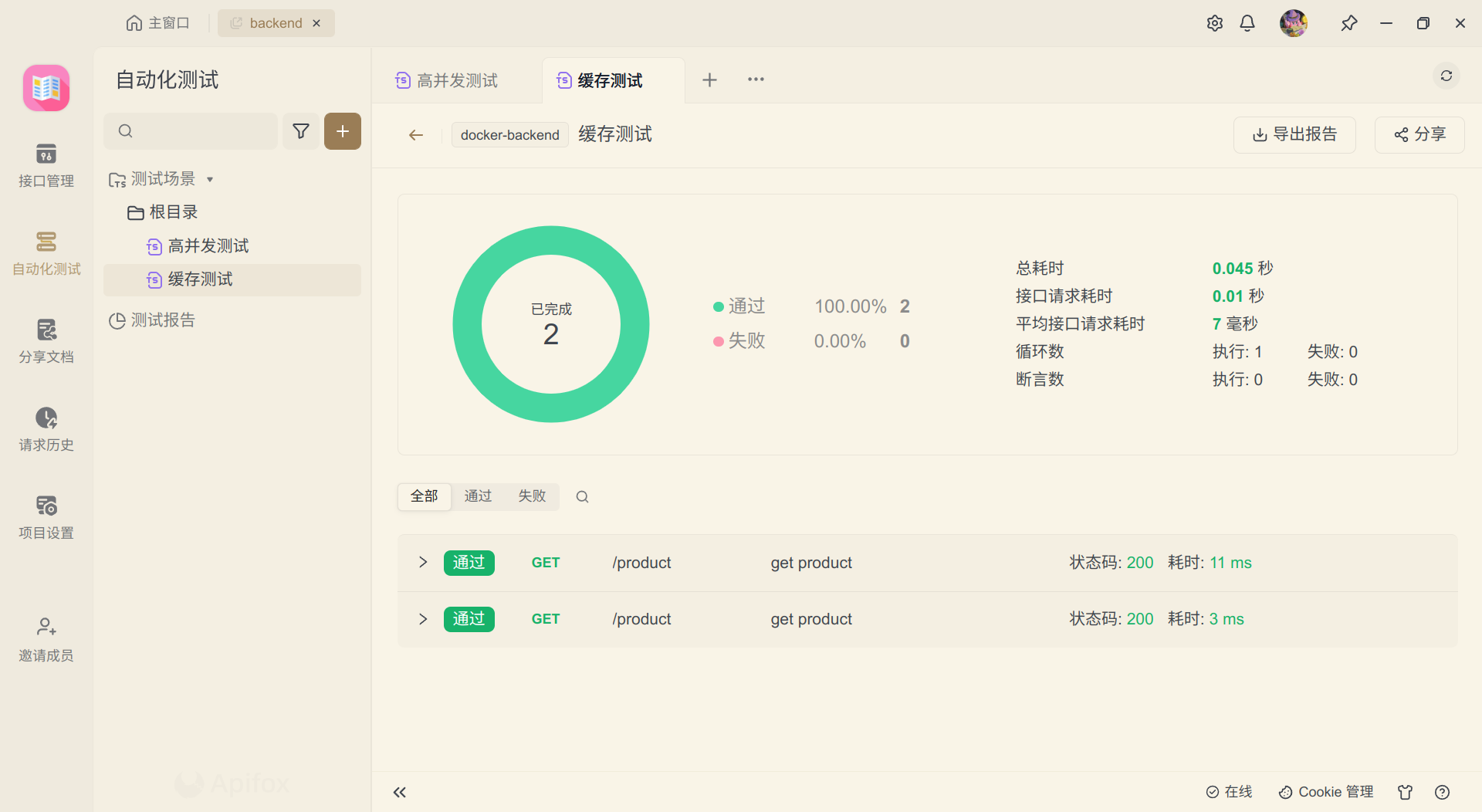
使用Redis CLI操控Redis：

|  |
| --- |
| Bash redis-cli |

清空当前Redis缓存：



1. 在没有缓存时测试用户获取商品接口。测试工具使用apifox，测试为单线程两次，第二次时将有缓存：可以看到，在没有缓存时响应时间为11ms，有缓存时响应时间为3ms，计算没有缓存时的QPS：有缓存时的QPS：。可以看到响应时间显著减少，QPS显著增加，性能有了明显提升。

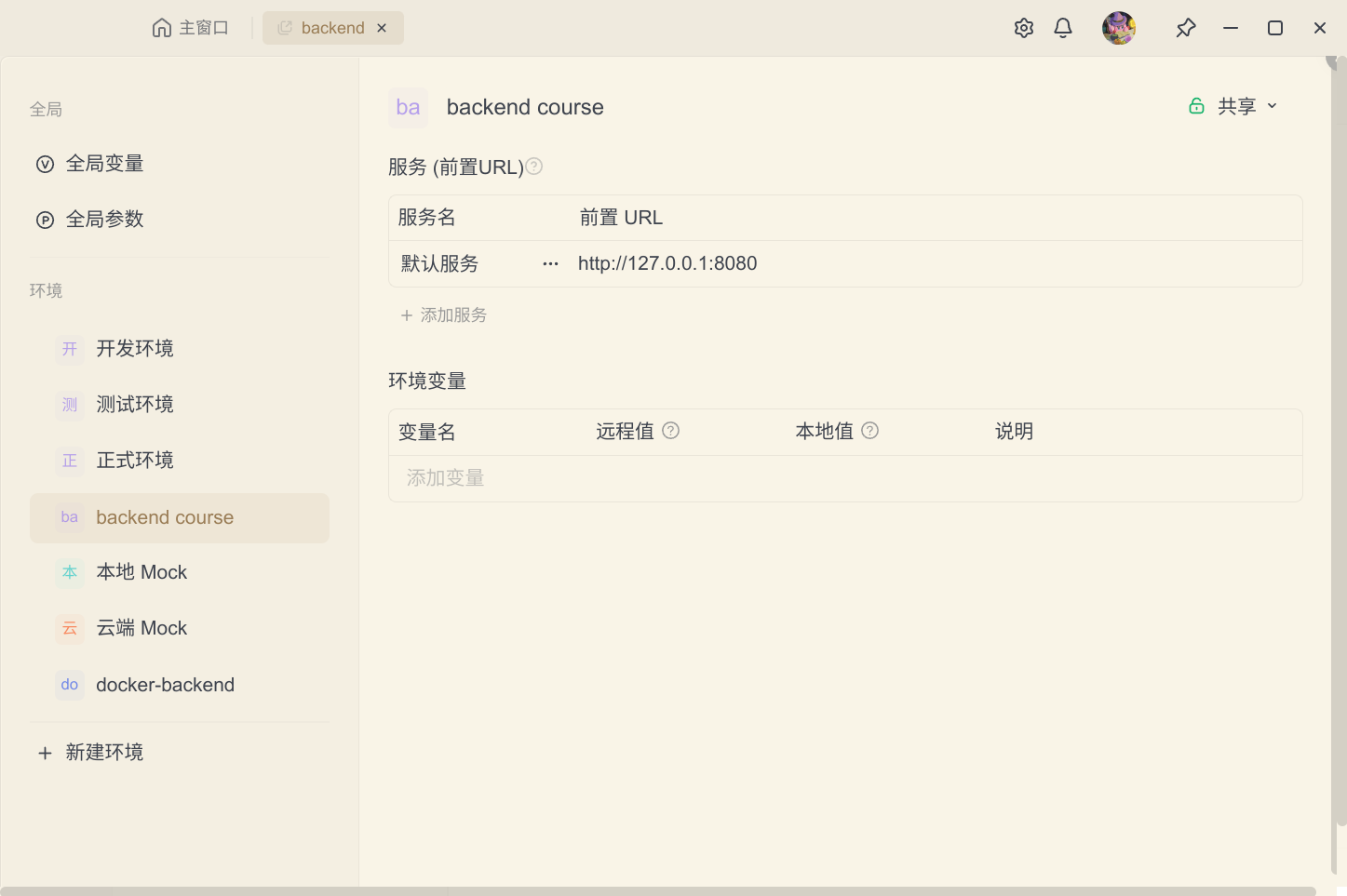


**Nginx负载均衡测试**

使用apifox来进行高并发测试，主要针对用户获取商品接口进行测试。

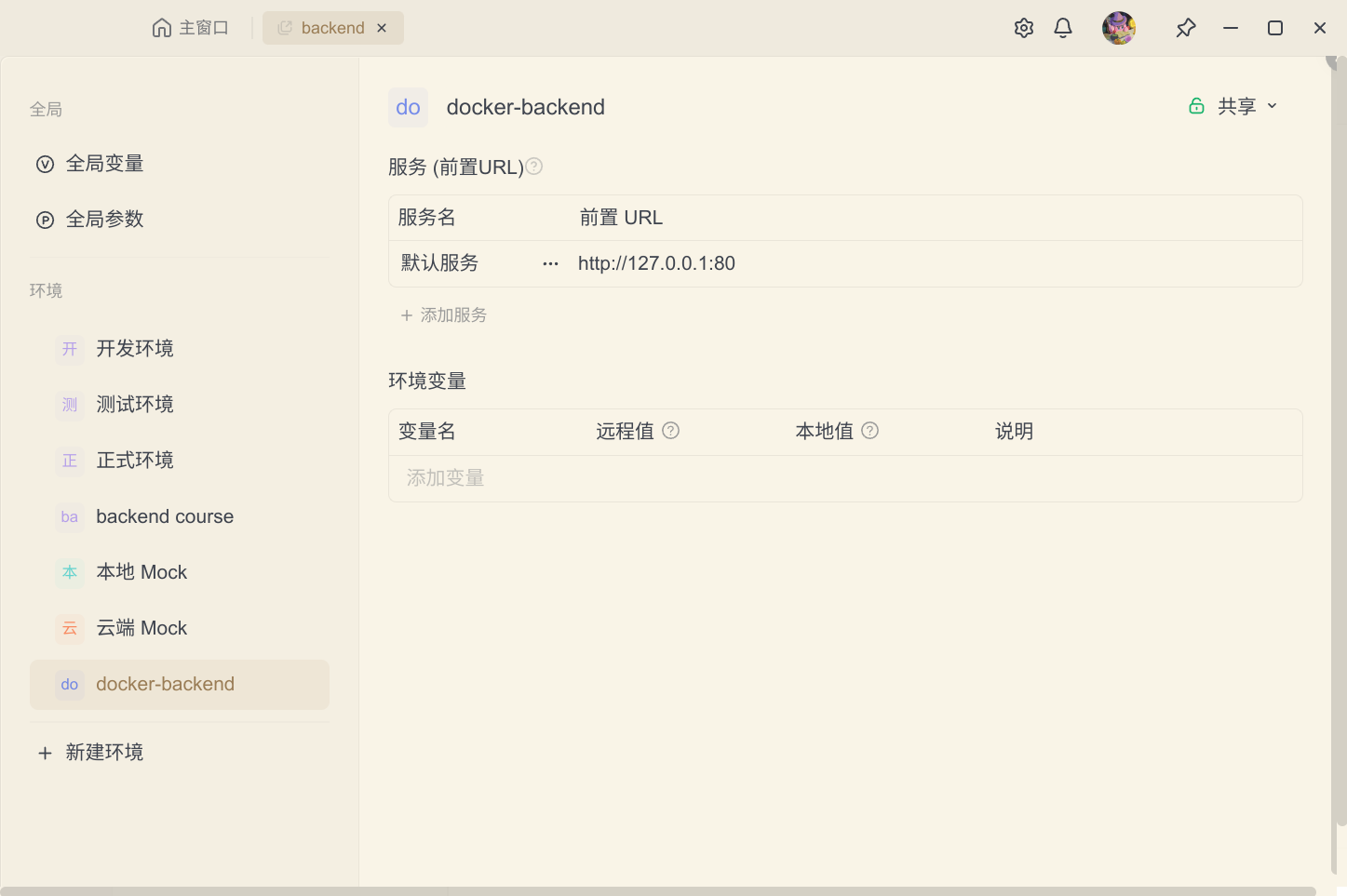
1. 10线程，一个线程分为一次登录和10次获取商品：

* 使用负载均衡前，在本地（对应端口8080，环境名为backend course）进行运行和测试：可以看到，请求110个接口总耗时为3.873秒，接口请求耗时为1.43秒，平均接口请求耗时为13毫秒。这表明响应时间为13ms，没有负载均衡时的QPS：





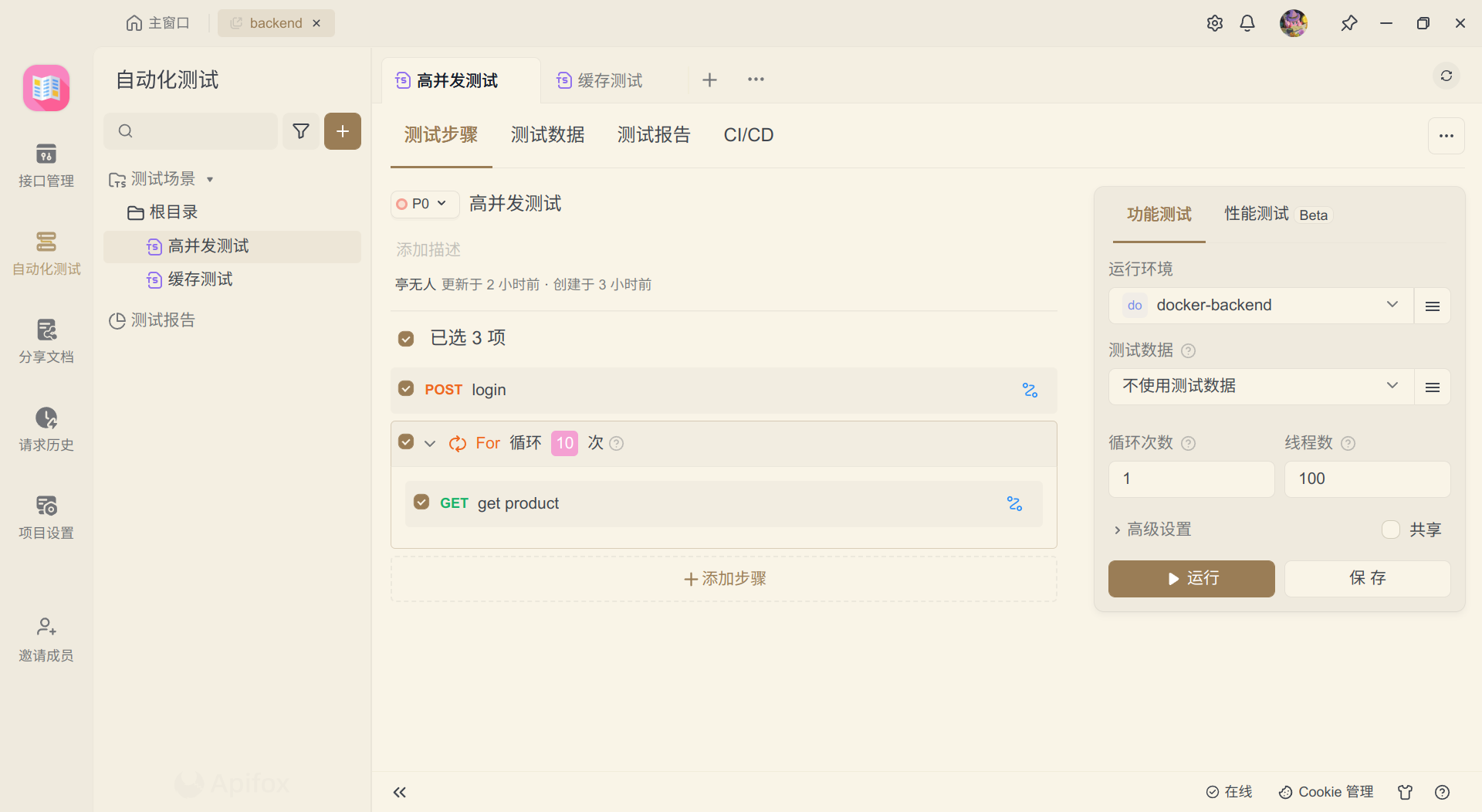
* 使用负载均衡后（对应端口80，环境名为docker-backend）测试：可以看到，请求110个接口总耗时为3.66秒，接口请求耗时为1.43秒，平均接口请求耗时为13毫秒。这表明响应时间为13ms，有负载均衡时的QPS：。性能整体有了一些提升。





1. 100线程，一个线程分为一次登录和10次获取商品：

* 使用负载均衡前测试：请求1100个接口的总耗时为：14.09秒，接口请求耗时为61.06秒，平均接口请求耗时为56ms。这表明响应时间为56ms，无负载均衡时的QPS：

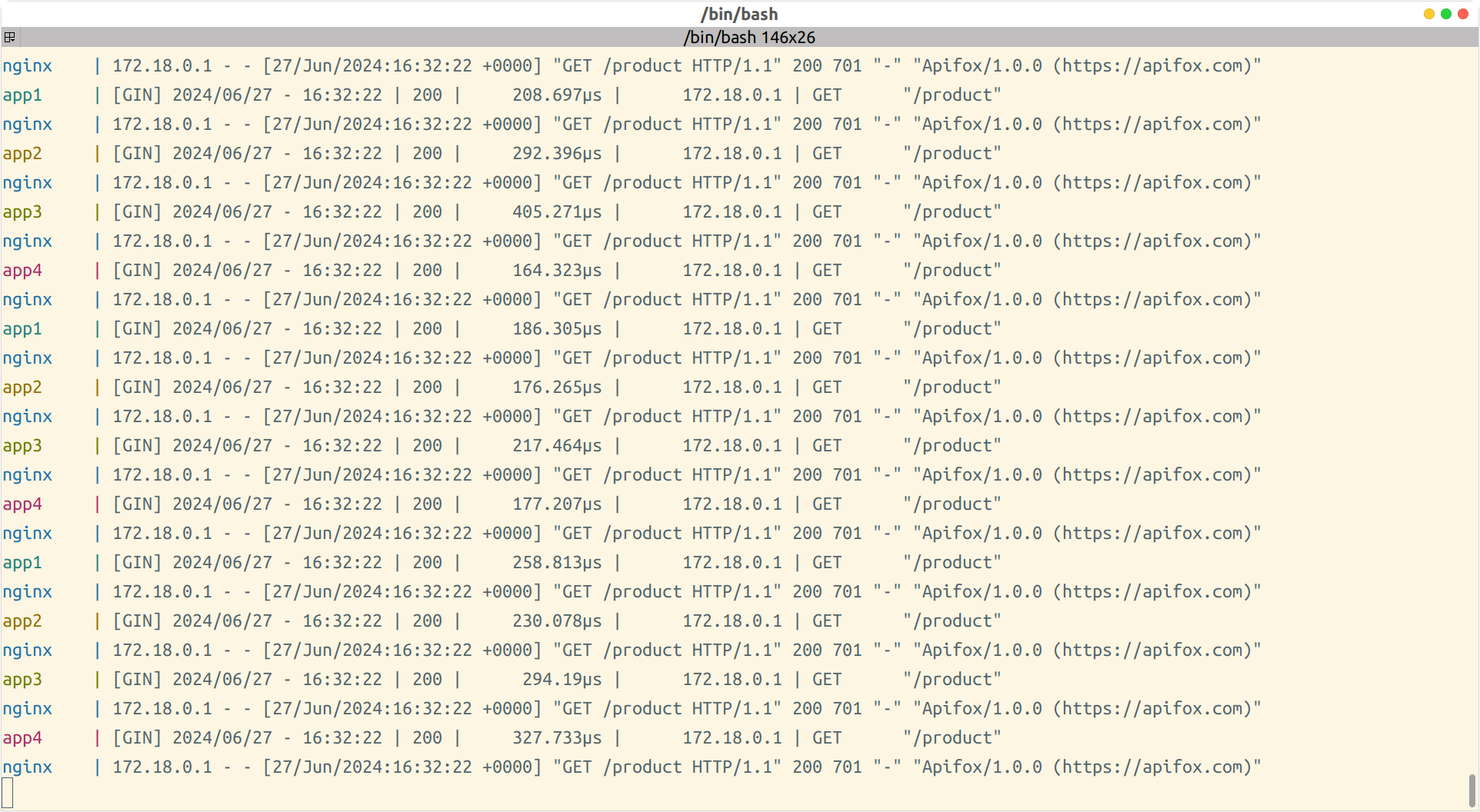




* 使用负载均衡后：请求1100个接口的总耗时为：13.054秒，接口请求耗时为52.8秒，平均接口请求耗时为48ms。这表明响应时间为48ms，无负载均衡时的QPS：。发现提升并不明显，怀疑是单机 Docker 部署性能受限的问题。在实际部署中，需要将该服务部署在多个阿里云ESC服务器上。



* 可以看到在 Docker-compose 服务启动的终端中看到负载均衡的详细过程：



**问题分析**

1. 问题一：Redis连接问题

刚开始将项目运行在Docker上时出现了Redis缓存连接失败的问题，分析可能是Docker内部网络问题。通过将Redis也部署在Docker上，保证网络配置正确，解决了这个问题。

1. 问题二：Nginx配置问题

在开始部署时对Nginx配置和负载均衡机制缺乏了解，初始部署仅有少量服务节点，未能充分利用负载均衡的优势。通过查阅相关资料，发现无状态服务具有可靠的扩展性，通过将多个无状态服务部署在多个节点，他们操控相同的数据库。这样，Nginx就可以将请求分发给它们中的任一个。在需要更好的性能时，可以通过拓展更多的无状态服务来轻松实现。

1. 问题三：部署方案问题

在选择部署方案时，为了简化测试和部署，将所有服务统一部署在本机的Docker容器中。这种方式虽然方便，但在高并发测试中存在潜在问题，因为所有的请求和请求处理都由同一台机器完成。这可能导致测试结果不准确，无法真实反映系统在分布式环境下的性能。在实际部署中，为了获得更准确的测试结果，应将服务部署在不同的主机上，并将请求和服务的处理分离。这意味着，高并发的请求应该由一组专门的主机发送，而服务应该由另一组主机处理。这样可以更真实地模拟实际生产环境下的分布式系统性能。

**前端界面功能操作示意图**

**用户注册界面**

**功能**：用户输入用户名、密码和邮箱进行注册。

**前端示意图**：

|  |
| --- |
| 用户注册 |
| 用户名: [\_\_\_\_\_\_\_\_\_\_\_\_\_] |
| 密码: [\_\_\_\_\_\_\_\_\_\_\_\_\_] |
| 邮箱: [\_\_\_\_\_\_\_\_\_\_\_\_\_] |
| [注册] |

**后端接口**：

* **URL**: /register
* **方法**: POST
* **请求数据**: { "username": "example", "password": "password", "email": "example@example.com" }
* **响应数据**: { "data": "user\_token" }

**用户登录界面**

**功能**：用户输入用户名和密码进行登录。

**前端示意图**：

|  |
| --- |
| 用户登录 |
| 用户名: [\_\_\_\_\_\_\_\_\_\_\_\_\_] |
| 密码: [\_\_\_\_\_\_\_\_\_\_\_\_\_] |
| [登录] |

**后端接口**：

* **URL**: /login
* **方法**: POST
* **请求数据**: { "username": "example", "password": "password" }
* **响应数据**: { "data": "user\_token" }

**用户退出界面**

**功能**：用户点击退出登录按钮退出登录

**前端示意图**：

|  |
| --- |
| 用户登出 |
| [退出登录] |

**后端接口**：

* **URL**: /logout
* **方法**: POST
* **请求数据**: { "token": "uesr\_token" }
* **响应数据**: { "data": "Logout successful." }

**商品管理界面**

**功能**：管理员添加商品，获取商品，删除商品。

**前端示意图**：

|  |
| --- |
| 产品列表 |
| 产品1 |
| 产品2 |
| 产品3 |
| ... |

**后端接口**：

* **管理员添加商品**：
* **URL**：/admin/product
* **方法**：POST
* **请求数据**：{ "Name": "Product1", "Category": "category", "Price": "10.0", "Stock": "100" }
* **响应数据**：{ "data": { "id": 1, "CreateAt": "time", "UpdateAt": "time", "DeleteAt": null, "name": "Product1", "category": "Category1", "price": 10.0, "stock": 100 }}
* **管理员获取商品**：
* **URL**：/admin/product/productID
* **方法**：GET
* **请求数据**：null
* **响应数据**：{ "data": { "id": 1, "CreateAt": "time", "UpdateAt": "time", "DeleteAt": null, "name": "Product1", "category": "Category1", "price": 10.0, "stock": 100 }}
* **管理员删除商品**：
* **URL**：/admin/product/productID
* **方法**：DELETE
* **请求数据**：null
* **响应数据**：{ "data": "Product deletion successful."}

**商品列表界面**

**功能**：显示商品的列表，将商品添加到购物车。

**前端示意图**：

|  |
| --- |
| 产品列表 |
| 产品1 |
| 产品2 |
| 产品3 |
| ... |

**后端接口**：

* **获取商品列表**：
* **URL**: /product
* **方法**: GET
* **请求数据**: 无
* **响应数据**: { "data": [{ "id": 1, "CreateAt": "time", "UpdateAt": "time", "DeleteAt": null, "name": "Product1", "category": "Category1", "price": 10.0, "stock": 100 }, ...] }
* **用户下订单**：
* **URL**：/order
* **方法**：POST
* **请求数据**：{ "ProductID": 1, "Quantity": 2 }
* **响应数据**：{ "data": { "id": 1, "CreateAt": "time", "UpdateAt": "time", "DeleteAt": null, "user\_id": 1, "product\_id": 1, "quantity": 2, "total": 20.0, "status": "unpaid" }

**订单页面**

**功能**：用户查看和支付订单。

**前端示意图**：

|  |
| --- |
| 我的订单 |
| 订单1 |
| 产品: 产品1 |
| 数量: 2 |
| 总价: 20.0 |
| 状态: 未支付 |
| [支付] |

**后端接口**：

* **获取订单**:
* **URL**: /order
* **方法**: GET
* **请求数据**: 无
* **响应数据**: { "data": [{ "id": 1, "CreateAt": "time", "UpdateAt": "time", "DeleteAt": null, "user\_id": 1, "product\_id": 1, "quantity": 2, "total": 20.0, "status": "unpaid" }, ...] }
* **支付订单**:
* **URL**: /order/pay
* **方法**: POST
* **请求数据**: { "order\_id": 1 }
* **响应数据**: { "data": "Payment successful." }

**评论界面**

**功能**：用户查看和添加产品评论。

**前端示意图**：

|  |
| --- |
| 产品评论 |
| 评论1 |
| 用户: 用户1 |
| 内容: 很好！ |
| 评论2 |
| 用户: 用户2 |
| 内容: 不错 |
| ... |
| [添加评论] |
| 评论内容: [\_\_\_\_\_\_\_\_\_\_\_\_\_] |
| [提交] |

**后端接口**：

* **获取评论**：
* **URL**: /comment/:productID
* **方法**: GET
* **请求数据**: 无
* **响应数据**: { "data": [{ "id": 1, "CreateAt": "time", "UpdateAt": "time", "DeleteAt": null, "user\_id": 1, "product\_id": 1, "content": "很好！" }, ...] }
* **添加评论**：
* **URL**: /comment
* **方法**: POST
* **请求数据**: { "product\_id": 1, "content": "很好！" }
* **响应数据**: { "data": { "id": 1, "CreateAt": "time", "UpdateAt": "time", "DeleteAt": null, "user\_id": 1, "product\_id": 1, "content": "很好！" } }