第 5 章

Spring Cloud Gateway

Spring Cloud Gateway 是一个基于多项技术开发的网关服务，它能够帮助开发者打造高效、稳定、安全的微服务应用程序。通过可扩展的路由、过滤器、负载均衡、安全性和监控等功能，能够轻松地将请求路由到微服务实例，优化接口响应，进而提升系统性能。本节将详细介绍如何使用 Spring Cloud Gateway 建立一个简单的微服务网关，涵盖了路由定义、过滤器添加及负载均衡实现等方面。

本节从动态路由、限流、负载均衡等方面为读者展示 Spring Cloud Gateway 的技术优势，同时还将对网关产品进行对比，帮助读者更好地选择适合自己的产品。同时还将介绍路由与更新、负载均衡策略、过滤器、限流方式、底层工作原理、高并发下的问题及解决方案等内容，使读者更加全面地了解 Spring Cloud Gateway 的优势和应用。希望通过本书，能够激发读者对 Spring Cloud Gateway 的兴趣和热情，同时也希望能够为读者提供实用的技术指南，帮助读者更好地应对分布式微服务架构的挑战。

5.1 动态路由/限流/负载均衡

在快速发展的微服务架构世界中，一群勇敢的开发者正在构建一个庞大的电子商务系统。在此系统中，多个微服务相互协同工作，例如订单管理、库存管理和购物车功能。作为核心组件的 API 网关，负责处理所有的请求。某个时刻，系统遭遇前所未有的流量冲击，大量请求涌入 API 网关，使处理速度显著降低。开发者意识到，如不及时解决问题，系统将面临崩溃的风险。在紧急情况下，开发者决定实施限流功能，以限制每个微服务的最大请求量。利用 API 网关限流功能，在请求进入网关之前进行限流处理。如此一来，即使流量高峰期，系统依然能够保持稳定运行。

除了限流功能，开发者还利用过滤器实现了用户身份验证。在请求到达微服务之前，过滤器会检查请求中的用户信息，确保仅有经过验证的用户方可访问系统。此举极大地提升了系统的安全性。

负载均衡方面，开发者借助 Spring Cloud Gateway 的动态路由功能。当有新请求到来时，API 网关会依据主机名、路径、方法和查询参数等元数据，从众多微服务中选取最合适的

目标地址。如此一来，系统能够根据实际需求动态调整负载，确保资源的合理分配。在 API 网关强大功能的支持下，开发者成功地应对了挑战，确保系统在高并发场景下稳定运行。

5.2 网关产品对比

随着互联网技术的飞速发展，网关服务已经成为各种应用程序和服务的关键组件。目前市面上的主流网关服务主要有 Spring Cloud Gateway、Zuul、OpenResty 和 Kong 等，其中，OpenResty 和 Kong 均依赖大量的 Lua 脚本，对开发者而言并非理想选择，而 Zuul 则存在两个主要版本：Zuul 1 和 Zuul 2。

Zuul 1 基于 Servlet 框架构建，采用阻塞和多线程方式，这导致了较高的线程资源占用。一个线程仅能处理一次连接请求，使处理流程较为复杂。Zuul 2 则引入了异步模型，运行于无阻塞框架上，降低了线程开销，然而，Zuul 2 在 2.x 版本后通过事件触发请求处理，导致请求流程易于中断，需要通过关联 ID 将整个请求串联起来。这使代码结构复杂且不易维护。

为解决 Zuul 的这些问题，Spring Cloud 推出了 Spring Cloud Gateway，以提升网关性能。它是基于 Spring Boot 2.x、Spring Webflux 和 Project Reactor 的 API 网关服务，专注于处理微服务之间的请求。

Spring Cloud Gateway 通过将负载均衡、路由、过滤等功能集成至其内部，简化了开发和维护过程。同时，Spring Cloud Gateway 还支持动态路由配置，实现了网关的高可用和灵活扩展。

5.3 路由与更新

在分布式系统中，路由是一种重要的功能，它决定了数据包从源节点到目标节点的传输路径。

5.3.1 静态路由

静态路由是一种预先配置的路由策略，它在系统启动时被加载并持久化到系统中。静态路由的优点是实现简单、配置方便，但缺点是无法适应网络拓扑的变化，容易造成路由表过大，增加系统开销。

为实现静态路由，可创建一个名为 staticStateRouteLocator 的自定义 RouteLocator 并对其进行配置。以下为简化后的实现步骤。

创建自定义 RouteLocator，代码如下：

```
//第 5 章/5.3.1 自定义 RouteLocator
import org.springframework.cloud.gateway.route.RouteLocator;
import org.springframework.cloud.gateway.route.builder.RouteLocatorBuilder;
import org.springframework.context.annotation.Bean;
```

```
import org.springframework.context.annotation.Configuration;
@Configuration
public class StaticStateRouteLocator {
    @Bean
    public RouteLocator staticStateRouteLocator(RouteLocatorBuilder
routeLocatorBuilder) {
        return routeLocatorBuilder
                .routes()
                .route("staticState-route", r ->r
                        .path("/user") //匹配所有路径
                        .uri("userService") //配置目标 URI
                )
                .build();
    }
}
```

在此例中，定义了一个名为 staticStateRouteLocator 的 RouteLocator 的 Bean，并使用 RouteLocatorBuilder 创建了一个包含一个路由的配置。该路由的 URI 为 userService，表示当路径匹配到/user 时，将请求路由到 userService。

在 Spring Cloud Gateway 配置文件中使用自定义 RouteLocator。在应用的 application.properties 或 application.yml 文件中添加以下配置以使用自定义的路由，代码如下：

```
//第 5 章/5.3.1 在配置文件中使用自定义的路由
spring:
  cloud:
    gateway:
      routes:
        #静态路由，将 id 指定为 staticState-route
        -id: staticState-route
          #uri 指向的服务名为 userService
          uri: userService
          predicates:
            #匹配所有的路径
            -Path=/**
```

在配置文件中定义一个名为 staticStateRoute 的 Routebean，它的 ID 为 staticStateroute。同时将 URI 指定为 userService，并添加了一个 Predicate(断言)，用于匹配所有路径。

完成以上步骤后，当请求路径匹配到/ * * 时，Spring Cloud Gateway 将自动将请求路由到 userService。这就是使用 Spring Cloud Gateway 实现静态路由的基本步骤。

本示例仅简化了代码，在实际使用时需要考虑处理动态路由配置、负载均衡等问题。同时，根据项目需求，可能需要修改 userService 的实际路径以满足业务需求。

5.3.2 动态路由

动态路由是一种网络技术，它可以实时监测网络拓扑的变化，并根据这些变化自动调整

路由选择，以确保网络流量的高效、稳定传输。与静态路由不同，动态路由不需要预先配置路由表，而是通过路由协议来实时更新路由信息，从而实现网络拓扑的动态管理。动态路由技术在现代网络架构中被广泛应用，尤其是在大型企业和数据中心网络中。

实现一个动态路由定位器的功能，可以从 Redis 中获取路由定义信息，将其添加到 Gateway 中。通过异步的方式实现添加、更新和删除路由，避免对主线程造成阻塞。同时，通过事件发布者的方式，实现路由信息的实时更新。这个功能对于需要动态更改路由规则的系统非常重要，可以极大地提高系统的灵活性和可维护性，代码如下：

```
//第 5 章/5.3.2 动态路由的实现
import com.alibaba.fastjson.json;
import org.springframework.cloud.gateway.event.RefreshRoutesEvent;
import org.springframework.cloud.gateway.route.RouteDefinition;
import org.springframework.cloud.gateway.route.RouteDefinitionWriter;
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.context.ApplicationEventPublisherAware;
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Component;
import reactor.core.publisher.Mono;
import javax.annotation.PostConstruct;
import javax.annotation.Resource;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
/*
 * 动态路由定位器
 */
@Component
public class DynamicRouteLocator implements ApplicationEventPublisherAware {
    //网关路由信息的 Redis 键
    private static final String GATEWAY_ROUTES ="GATEWAY_ROUTES";
    //路由定义的写入器
    @Resource
    private RouteDefinitionWriter routeDefinitionWriter;
    //Redis 操作模板
    @Resource
    private StringRedisTemplate redisTemplate;
    //应用事件发布者
    private ApplicationEventPublisher publisher;
    //线程池
    private ExecutorService executorService =Executors.newFixedThreadPool(10);
    //Bean 初始化方法，在构造后执行
    @PostConstruct
    public void init() {
        //从 Redis 中获取路由定义信息，并添加到 Gateway 中
        redisTemplate.opsForHash().values(GATEWAY_ROUTES).forEach(route ->{
            RouteDefinition definition = JSON.parseObject(route.toString(),
RouteDefinition.class);
```

```
            add(definition);
        });
    }
    //实现 ApplicationEventPublisherAware 接口的方法,用于设置应用事件发布者
    @Override
    public void setApplicationEventPublisher(ApplicationEventPublisher
applicationEventPublisher) {
        this.publisher =applicationEventPublisher;
    }
    //发布刷新路由事件
    private void notifyChanged() {
        this.publisher.publishEvent(new RefreshRoutesEvent(this));
    }
    /**
     * 增加路由
     */
    public String add(RouteDefinition definition) {
        //提交到线程池中执行
        executorService.submit(() ->{
            try {
                routeDefinitionWriter.save(Mono.just(definition)).subscribe();
                notifyChanged();
                //将路由定义信息添加到 Redis 中
                  redisTemplate.opsForHash().put(GATEWAY_ROUTES, definition.
getId(), JSON.toJSONString(definition));
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
        return "success";
    }
    /**
     * 更新路由
     */
    public String update(RouteDefinition definition) {
        //提交到线程池中执行
        executorService.submit(() ->{
            try {
                routeDefinitionWriter.delete(Mono.just(definition.getId()));
                routeDefinitionWriter.save(Mono.just(definition)).subscribe();
                notifyChanged();
                //将更新后的路由定义信息更新到 Redis 中
                redisTemplate.opsForHash().put(GATEWAY_ROUTES, definition.
getId(), JSON.toJSONString(definition));
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
        return "success";
```

```
    }
    /**
     * 删除路由
     */
    public String delete(String id) {
        //提交到线程池中执行
        executorService.submit(() ->{
            try {
                routeDefinitionWriter.delete(Mono.just(id)).subscribe();
                notifyChanged();
                //从 Redis 中删除路由定义信息
                redisTemplate.opsForHash().delete(GATEWAY_ROUTES, id);
            } catch (Exception e) {
                e.printStackTrace();
            }
        });
        return "delete success";
    }
}
```

首先，代码中定义了一个静态变量 GATEWAY_ROUTES，表示网关路由信息在 Redis 中的键值。同时，使用了 Spring Cloud Gateway 的路由定义写入器 RouteDefinitionWriter 和 Redis 操作模板 StringRedisTemplate，这两个对象都是通过注解@Resource 进行注入的。

在类初始化之后，通过@Bean 注解的 init()方法，从 Redis 中获取路由定义信息，并使用 RouteDefinitionWriter 的 save()方法将其添加到 Gateway 中，其中，获取的路由信息是以 JSON 格式存储的，需要使用 FastJson 的 JSON.parseObject()方法进行解析。

接下来是 3 个关键的方法：add()、update()和 delete()，分别用于添加、更新和删除路由。这 3 种方法均采用线程池的方式异步执行，避免对主线程造成阻塞。在添加路由的过程中，还需要将路由定义信息保存到 Redis，以便后续的更新和删除操作可以快速进行。

最后，代码中还定义了一个 notifyChanged()方法，用于发布刷新路由事件，以便将新的路由信息实时应用到 Gateway 中。这种方法调用了 ApplicationEventPublisher 的 publishEvent()方法，将 RefreshRoutesEvent 事件发布出去，告知 Gateway 需要更新路由信息。

5.3.3 底层全量更新和底层增量更新

底层全量更新：将整个路由表的内容全部重新计算和更新。更新实现简单，但网络容易中断，影响系统性能，代码如下：

```
//第 5 章/5.3.3 底层全量更新
import com.alibaba.fastjson.json; //引入 fastjson 库
import javassist.NotFoundException;
//引入 javassist 库的 NotFoundException 类
import org.slf4j.Logger; //引入日志库
```

```
import org.slf4j.LoggerFactory; //引入日志库
import org.springframework.beans.factory.annotation.Value;
//引入 Spring 框架的注解
import org.springframework.cloud.gateway.event.RefreshRoutesEvent;
//引入 gateway 的 RefreshRoutesEvent 类
import org.springframework.cloud.gateway.route.RouteDefinition;
//引入 gateway 的 RouteDefinition 类
import org.springframework.cloud.gateway.route.RouteDefinitionRepository;
//引入 gateway 的 RouteDefinitionRepository 接口
import org.springframework.context.ApplicationEventPublisher;
//引入 Spring 框架的 ApplicationEventPublisher 类
import org.springframework.context.ApplicationEventPublisherAware;
//引入 Spring 框架的 ApplicationEventPublisherAware 接口
import org.springframework.context.event.EventListener;
//引入 Spring 框架的 EventListener 注解
import org.springframework.stereotype.Component;
//引入 Spring 框架的 Component 注解
import reactor.core.publisher.Flux; //引入 Reactor 库的 Flux 类
import reactor.core.publisher.Mono; //引入 Reactor 库的 Mono 类
import javax.annotation.PostConstruct; //引入 javax 库的 PostConstruct 注解
import java.nio.file.Files; //引入 NIO 库的 Files 类
import java.nio.file.Paths; //引入 NIO 库的 Paths 类
import java.util.ArrayList; //引入 java.util 库的 ArrayList 类
import java.util.List; //引入 java.util 库的 List 类
import java.util.stream.Collectors;
 //引入 java.util.stream 库的 Collectors 类
@Component //标注为 Spring 组件
public class FileRouteDefinitionRepository implements RouteDefinitionRepository,
ApplicationEventPublisherAware {
//定义 FileRouteDefinitionRepository 类实现 RouteDefinitionRepository 和
//ApplicationEventPublisherAware 接口
    private static final Logger LOGGER =
LoggerFactory.getLogger(FileRouteDefinitionRepository.class);
    //声明静态 Logger 变量
    private ApplicationEventPublisher publisher;
    //定义 ApplicationEventPublisher 变量
    private List<RouteDefinition> routeDefinitionList =new ArrayList<>();
    //声明 List<RouteDefinition>变量,并初始化为 ArrayList
    @Value("${gateway.route.config.file}")
    //注入属性,需要去配置文件里面配置
    private String file; //声明文件路径变量
    @Override
       public  void  setApplicationEventPublisher  ( ApplicationEventPublisher
publisher) { //实现 setApplicationEventPublisher 方法
        this.publisher =publisher; //给 publisher 赋值
    }
    @PostConstruct //标注为在构造函数之后执行的方法
    public void init() { //定义 init 方法
        load(); //调用 load 方法
```

```
    }
    /**
     * 监听事件刷新配置
     */
    @EventListener //标注为事件监听方法
    public void listenEvent(RefreshRoutesEvent event) {
    //定义 listenEvent 方法
        load(); //调用 load 方法
        this.publisher.publishEvent(event); //发布事件
    }
    /**
     * 加载
     */
    private void load() { //定义 load 方法
        try { //捕获异常
            String jsonStr = Files.lines(Paths.get(file)).collect(Collectors.
joining());
//读取并拼接文件内容
             routeDefinitionList = JSON.parseArray(jsonStr, RouteDefinition.
class); //将 JSON 字符串解析为 List<RouteDefinition>对象
           LOGGER.info("路由配置已加载,加载条数:{}", routeDefinitionList.size());
//打印日志
        } catch (Exception e) { //捕获异常
            LOGGER.error("从文件加载路由配置异常", e); //打印错误日志
        }
    }
    @Override
    public Mono<Void>save(Mono<RouteDefinition>route) { //实现 save 方法
        return Mono.defer(() ->Mono.error(new NotFoundException("Unsupported
operation"))); //返回错误信息
    }
    @Override
    public Mono<Void>delete(Mono<String>routeId) { //实现 delete 方法
        return Mono.defer(() ->Mono.error(new NotFoundException("Unsupported
operation"))); //返回错误信息
    }
    @Override
    public Flux<RouteDefinition>getRouteDefinitions() {
    //实现 getRouteDefinitions 方法
        return Flux.fromIterable(routeDefinitionList);
        //返回 routeDefinitionList 的 Flux 流
    }
}
```

首先,gateway.route.config.file 属性要指向存在且格式正确的 JSON 文件,否则会抛出异常。其次,如果想动态更新路由配置,则需要在配置文件中开启 spring.cloud.gateway.discovery.locator.enabled 属性,并将其设置为 true,这样才能触发 RefreshRoutesEvent 事件来更新配置。注意,使用该事件更新配置时,需要确保已经开启了动态路由配置功能,否

则即使触发了事件，配置也无法更新。最后，使用 RouteDefinitionRepository 接口更新路由配置时，需要确保 Spring Cloud Gateway 版本支持该接口。

底层增量更新：更新发生变化的部分，由于数据量变小，容易出现网络中断的时间也变短。代码如下：

```
//第 5 章/5.3.3 底层增量更新

import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.cloud.gateway.event.RefreshRoutesEvent;
import org.springframework.cloud.gateway.route.RouteDefinition;
import org.springframework.cloud.gateway.route.RouteDefinitionRepository;
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.context.ApplicationEventPublisherAware;
import org.springframework.stereotype.Component;
import reactor.core.publisher.Flux;
import reactor.core.publisher.Mono;
import java.util.ArrayList;
import java.util.List;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

/**
 * 保存路由信息，优先级比配置文件高
 */
@Component
@RequiredArgsConstructor //使用构造函数注入
@Slf4j(topic ="RedisRouteDefinitionWriter") //使用 lombok 的 slf4j 注解
public class RedisRouteDefinitionWriter implements RouteDefinitionRepository,
ApplicationEventPublisherAware {
    private final Map<String, RouteDefinition>routeDefinitions =new
ConcurrentHashMap<>(); //使用 ConcurrentHashMap 存储路由信息
    private ApplicationEventPublisher publisher;
    //定义 ApplicationEventPublisher 对象
    @Override
    public Mono<Void>save(Mono<RouteDefinition>route) {
        return route.flatMap(r ->{
            log.info("保存路由信息 {}", r); //日志记录
            routeDefinitions.put(r.getId(), r);
            //将路由信息保存在 ConcurrentHashMap 中
            refreshRoutes(); //刷新路由信息
            return Mono.empty();
        });
    }
    @Override
    public Mono<Void>delete(Mono<String>routeId) {
        return routeId.flatMap(id ->{
            RouteDefinition routeDefinition =routeDefinitions.remove(id);
```

```
            //从 ConcurrentHashMap 中移除对应 id 的路由信息
            if (routeDefinition ! =null) {
                log.info("删除路由信息 {}", routeDefinition); //日志记录
                refreshRoutes(); //刷新路由信息
            }
            return Mono.empty();
        });
    }
    private void refreshRoutes() {
        this.publisher.publishEvent(new RefreshRoutesEvent(this));
        //使用 ApplicationEventPublisher 发布 RefreshRoutesEvent 事件,刷新路由信息
    }
    /**
     * 动态路由入口
     * @return Flux<RouteDefinition>
     */
    @Override
    public Flux<RouteDefinition>getRouteDefinitions() {
        List<RouteDefinition>definitions =new ArrayList<>(routeDefinitions.
values());
        //获取 ConcurrentHashMap 中保存的路由信息
        log.info("当前路由定义条数:{},definitions={}", definitions.size(),
definitions); //日志记录
        return Flux.fromIterable(definitions); //返回路由信息的 Flux 序列
    }
    @Override
    public void setApplicationEventPublisher(ApplicationEventPublisher
applicationEventPublisher) {
        this.publisher =applicationEventPublisher;
        //设置 ApplicationEventPublisher 对象
    }
}
```

为了保证实现路由信息存储的线程安全性,使用 ConcurrentHashMap 以避免多线程操作导致的潜在问题。利用 lombok 的 RequiredArgsConstructor 和 slf4j 注解简化了代码,提高了可读性。使用 ApplicationEventPublisher 发布 RefreshRoutesEvent 事件,以便网关能够及时刷新路由信息。为了提升性能,采用了 Reactive 编程实现异步操作,以减少线程等待时间。

5.4 负载均衡策略

作为网关,SpringCloudGateway 负责处理来自客户端的请求,并根据路由规则将请求转发至后端微服务。通过 SpringCloudGateway,开发者可以更加便捷地实现微服务的统一入口,并对请求进行统一的鉴权、限流等操作。

在现代的微服务架构中,负载均衡是一种至关重要的技术,它可以确保各个服务实例都