# 配置初始化

maven 依赖的 jar 包：

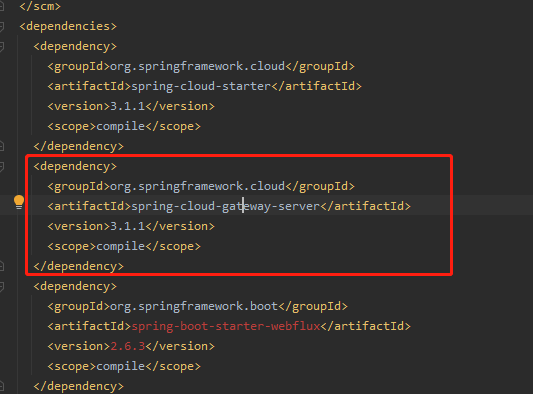
<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-webflux</artifactId>

</dependency>

其中gateway的核心依赖：



依旧使用spring.factories文件进行注册：

// Auto Configure

org.springframework.boot.autoconfigure.EnableAutoConfiguration=

// 依赖包的校验配置

org.springframework.cloud.gateway.config.GatewayClassPathWarningAutoConfiguration,

// 网关的核心配置

org.springframework.cloud.gateway.config.GatewayAutoConfiguration,

// 负载均衡相关依赖配置信息

org.springframework.cloud.gateway.config.GatewayLoadBalancerClientAutoConfiguration,

// 度量相关依赖配置信息

org.springframework.cloud.gateway.config.GatewayMetricsAutoConfiguration,

// 流控的依赖配置信息

org.springframework.cloud.gateway.config.GatewayRedisAutoConfiguration,

// 注册中心相关的依赖配置

org.springframework.cloud.gateway.discovery.GatewayDiscoveryClientAutoConfiguration

## GatewayClassPathWarningAutoConfiguration

GatewayClassPathWarningAutoConfiguration 用于检查项目是否正确导入 spring-boot-starter-webflux 依赖，而不是错误导入 spring-boot-starter-web 依赖，同时 GatewayClassPathWarningAutoConfiguration 在 EnableAutoConfiguration 配置加载前加载。

@Configuration(proxyBeanMethods = false)

// 执行顺序注解

// 当前注解标识需要在GatewayAutoConfiguration前加载此配置

@AutoConfigureBefore({GatewayAutoConfiguration.class})

@ConditionalOnProperty(

name = {"spring.cloud.gateway.enabled"},

matchIfMissing = true

)

public class GatewayClassPathWarningAutoConfiguration {

private static final Log log = LogFactory.getLog(GatewayClassPathWarningAutoConfiguration.class);

private static final String BORDER = "\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n";

public GatewayClassPathWarningAutoConfiguration() {

}

@Configuration(proxyBeanMethods = false)

// 条件判断注解

// classpath中不存在org.springframework.web.reactive.DispatcherHandler时起效，标识项目未导入了spring-boot-starter-webflux包

@ConditionalOnMissingClass({"org.springframework.web.reactive.DispatcherHandler"})

protected static class WebfluxMissingFromClasspathConfiguration {

public WebfluxMissingFromClasspathConfiguration() {

// 当前项目未导入了spring-boot-starter-webflux依赖时，打印警告日志

GatewayClassPathWarningAutoConfiguration.log.warn("\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\nSpring Webflux is missing from the classpath, which is required for Spring Cloud Gateway at this time. Please add spring-boot-starter-webflux dependency.\n\n\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n\n");

}

}

@Configuration(proxyBeanMethods = false)

// 条件判断注解

// classpath中存在org.springframework.web.servlet.DispatcherServlet时起效，标识项目导入了spring-boot-starter-web包

@ConditionalOnClass(name = {"org.springframework.web.servlet.DispatcherServlet"})

// 当前是web环境才执行

@ConditionalOnWebApplication(type = Type.SERVLET)

protected static class SpringMvcFoundOnClasspathConfiguration {

public SpringMvcFoundOnClasspathConfiguration() {

// 当前项目导入了spring-boot-starter-web依赖时，

throw new MvcFoundOnClasspathException();

}

}

}

## GatewayAutoConfiguration

GatewayAutoConfiguration 是 Spring Cloud Gateway 核心配置类，这里只列几个核心的，初始化如下 ：

* RoutePredicateHandlerMapping：查找匹配到 Route 并进行处理
* GatewayProperties：加载网关配置
* RouteDefinitionRouteLocator：创建一个根据 RouteDefinition 转换的路由定位器
* GatewayWebfluxEndpoint：管理网关的 HTTP API

## GatewayReactiveLoadBalancerClientAutoConfiguration

GatewayReactiveLoadBalancerClientAutoConfiguration作用是初始化 LoadBalancerClientFilter 路由的负载均衡拦截器

@Configuration (proxyBeanMethods = false)

// 条件判断注解

// classpath中存在ReactiveLoadBalancer和LoadBalancerAutoConfiguration和DispatcherHandler时此配置起效

@ConditionalOnClass({ReactiveLoadBalancer.class, LoadBalancerAutoConfiguration.class, DispatcherHandler.class})

// 执行顺序注解

@AutoConfigureAfter({LoadBalancerAutoConfiguration.class})

@EnableConfigurationProperties({GatewayLoadBalancerProperties.class})

public class GatewayReactiveLoadBalancerClientAutoConfiguration {

public GatewayReactiveLoadBalancerClientAutoConfiguration () {

}

@Bean

// 条件判断注解

// DI容器中存在LoadBalancerClientFactory类型Bean时起效

@ConditionalOnBean({LoadBalancerClientFactory.class})

@ConditionalOnMissingBean({ReactiveLoadBalancerClientFilter.class})

public ReactiveLoadBalancerClientFilter gatewayLoadBalancerClientFilter(LoadBalancerClient client, LoadBalancerProperties properties) {

return new ReactiveLoadBalancerClientFilter(client, properties);

}

@Bean

@ConditionalOnBean({ReactiveLoadBalancerClientFilter.class, LoadBalancerClientFactory.class})

@ConditionalOnMissingBean

@ConditionalOnEnabledGlobalFilter

public LoadBalancerServiceInstanceCookieFilter loadBalancerServiceInstanceCookieFilter(LoadBalancerClientFactory loadBalancerClientFactory) {

return new LoadBalancerServiceInstanceCookieFilter(loadBalancerClientFactory);

}

}

## GatewayMetricsAutoConfiguration

GatewayMetricsAutoConfiguration 作用是初始化 GatewayMetricsFilter 路由的度量拦截器。

## GatewayRedisAutoConfiguration

GatewayRedisAutoConfiguration 配置作用是初始化初始化 RedisRateLimiter 限流功能的，RequestRateLimiterGatewayFilterFactory 基于 RedisRateLimiter 实现网关的限流功能。

## GatewayDiscoveryClientAutoConfiguration

GatewayDiscoveryClientAutoConfiguration 的作用是初始化配置路由中的注册发现服务信息

# Route

## 构建方式

一般构建分为两种：外部化配置和编程方式。

### 外部化配置

spring:

cloud:

gateway:

routes:

- id: after\_route // ①

uri: https://example.org // ②

predicates:

- Cookie=mycookie,mycookievalue // ③

filters:

- AddRequestHeader=X-Request-Foo, Bar // ④

* 配置了一个 Route id 为 after\_route
* 客户端请求转发的目的地：https://example.org
* 在 request 中，当存在名字 mycookie 的 cookie 的值匹配 mycookievalue 则算成功
* 定义了一个 Filter，匹配成功后，会在请求头上添加 X-Request-Foo:Bar

### 编程方式

转换成编程方式

@Bean

public RouteLocator customRouteLocator(RouteLocatorBuilder builder) {

builder.routes()

.route(r -> r.cookie("mycookie", "mycookievalue")

.filters(f -> f.addRequestHeader("X-Request-Foo", "Bar"))

.uri("https://example.org")

)

.build();

}

## 构建原理

### 外部化配置

外部化配置是通过 GatewayProperties 进行构建的：

/\*\*

\* 网关配置信息加载

\* 从appliccation.yml中解析前缀为spring.cloud.gateway的配置

\*/

@ConfigurationProperties("spring.cloud.gateway")

@Validated

public class GatewayProperties {

private final Log logger = LogFactory.getLog(this.getClass());

/\*\*

\* 路由定义列表

\* 加载配置key=spring.cloud.gateway.routes 列表

\* List of Routes

\*/

@NotNull

@Valid

private List<RouteDefinition> routes = new ArrayList();

/\*\*

\* 默认的过滤器定义列表

\* 加载配置 key = spring.cloud.gateway.default-filters 列表

\* List of filter definitions that are applied to every route.

\*/

private List<FilterDefinition> defaultFilters = new ArrayList();

/\*\*

\* 网媒体类型列表

\* 加载配置 key = spring.cloud.gateway.streamingMediaTypes 列表

\* 默认包含{text/event-stream,application/stream+json}

\*/

private List<MediaType> streamingMediaTypes;

/\*\*

\* 路由定义错误时失败

\* 默认为true

\*/

private boolean failOnRouteDefinitionError;

public GatewayProperties() {

this.streamingMediaTypes = Arrays.asList(MediaType.TEXT\_EVENT\_STREAM, MediaType.APPLICATION\_STREAM\_JSON, new MediaType("application", "grpc"), new MediaType("application", "grpc+protobuf"), new MediaType("application", "grpc+json"));

this.failOnRouteDefinitionError = true;

}

...

}

#### RouteDefinition

用来对 Route 进行定义。也就是，通过 GatewayProperties 会与外部化配置进行绑定，把外部化配置比如 properties 或者 yml 绑定到 GatewayProperties 中。

/\*\*

\* 路由定义实体信息，包含路由的定义信息

\*/

@Validated

public class RouteDefinition {

/\*\*

\* 路由ID 编号，唯一

\*/

@NotEmpty

private String id = UUID.randomUUID().toString();

/\*\*

\* 谓语定义数组

\* predicates 属性，谓语定义数组

\* 请求通过 判断是否匹配。在 Route 里，PredicateDefinition 转换成 Predicate

\*/

@NotEmpty

@Valid

private List<PredicateDefinition> predicates = new ArrayList();

/\*\*

\* 过滤器定义数组

\* filters 属性，过滤器定义数组。

\* 在 Route 里，FilterDefinition 转换成 GatewayFilter

\*/

@Valid

private List<FilterDefinition> filters = new ArrayList();

/\*\*

\* 路由指向的URI

\*/

@NotNull

private URI uri;

/\*\*

\* 顺序

\*/

private int order = 0;

...

}

#### PredicateDefinition

/\*\*

\* 谓语定义,在 Route 里，PredicateDefinition 将转换成 Predicate

\*/

@Validated

public class PredicateDefinition {

/\*\*

\* 谓语定义名字

\* 通过 name 对应到 org.springframework.cloud.gateway.handler.predicate.RoutePredicateFactory 的实现类。

\* 例如: name=Query 对应到 QueryRoutePredicateFactory

\*/

@NotNull

private String name;

/\*\*

\* 参数数组

\* 例如，name=Host / args={"\_genkey\_0" : "iocoder.cn"} ，匹配请求的 hostname 为 iocoder.cn

\*/

private Map<String, String> args = new LinkedHashMap();

...

}

#### FilterDefinition

/\*\*

\* 过滤器定义，在 Route 里，FilterDefinition将转换成 GatewayFilter

\*/

@Validated

public class FilterDefinition {

/\*\*

\* 过滤器定义名字

\* 通过 name 对应到 org.springframework.cloud.gateway.filter.factory.GatewayFilterFactory 的实现类。

\* 例如，name=AddRequestParameter 对应到 AddRequestParameterGatewayFilterFactory

\*/

@NotNull

private String name;

/\*\*

\* 参数数组

\* 例如 name=AddRequestParameter / args={"\_genkey\_0": "foo", "\_genkey\_1": "bar"} ，添加请求参数 foo 为 bar

\*/

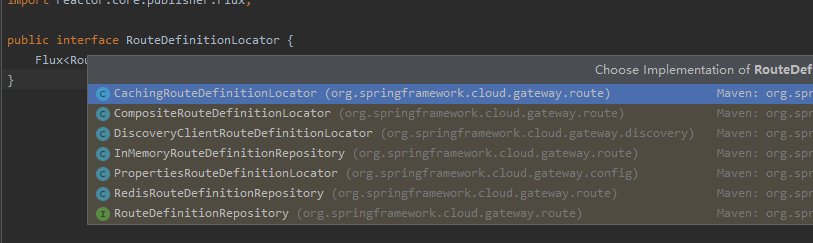
private Map<String, String> args = new LinkedHashMap();

...

}

#### RouteDefinitionLocator

GatewayAutoConfiguration 核心配置类中，有提到此类。Gateway 提供多种方式来获取外部的配置，而 RouteDefinitionLocator 就是父接口，下面有多种实现。



##### PropertiesRouteDefinitionLocator

从配置文件(YML、Properties 等) 读取路由配置。代码如下：

public class PropertiesRouteDefinitionLocator implements RouteDefinitionLocator {

private final GatewayProperties properties;

public PropertiesRouteDefinitionLocator(GatewayProperties properties) {

this.properties = properties;

}

public Flux<RouteDefinition> getRouteDefinitions() {

// 从 GatewayProperties 获取路由配置数组。

return Flux.fromIterable(this.properties.getRoutes());

}

}

##### InMemoryRouteDefinitionRepository

从存储器(内存、Redis、MySQL 等)读取、保存、删除路由配置。InMemoryRouteDefinitionRepository 是基于内存的。

public class InMemoryRouteDefinitionRepository implements RouteDefinitionRepository {

/\*\*

\* 路由配置映射 通过此来保存route

\* key ：路由编号 {@link RouteDefinition#id}

\*/

private final Map<String, RouteDefinition> routes = Collections.synchronizedMap(new LinkedHashMap());

public InMemoryRouteDefinitionRepository() {

}

public Mono<Void> save(Mono<RouteDefinition> route) {

return route.flatMap((r) -> {

this.routes.put(r.getId(), r);

return Mono.empty();

});

}

public Mono<Void> delete(Mono<String> routeId) {

return routeId.flatMap((id) -> {

if (this.routes.containsKey(id)) {

this.routes.remove(id);

return Mono.empty();

} else {

return Mono.defer(() -> {

return Mono.error(new NotFoundException("RouteDefinition not found: " + routeId));

});

}

});

}

public Flux<RouteDefinition> getRouteDefinitions() {

return Flux.fromIterable(this.routes.values());

}

}

基于内存，通过 Map routes 来保存 Route，缺点是如果重启，那么 route 会丢失。可以实现 RouteDefinitionRepository 接口自定义比如通过 Redis、Mysql 等来保存 Route。

##### DiscoveryClientRouteDefinitionLocator

获取在注册中心的服务列表，生成对应的 RouteDefinition 数组。

##### CompositeRouteDefinitionLocator

组合多种 RouteDefinitionLocator 的实现，为 RouteDefinitionRouteLocator 提供统一入口。

public class CompositeRouteDefinitionLocator implements RouteDefinitionLocator {

// RouteDefinitionLocator 数组

private final Flux<RouteDefinitionLocator> delegates;

public CompositeRouteDefinitionLocator(Flux<RouteDefinitionLocator> delegates) {

this.delegates = delegates;

}

// 将组合的 delegates 的路由定义全部返回。

public Flux<RouteDefinition> getRouteDefinitions() {

return this.delegates.flatMap(RouteDefinitionLocator::getRouteDefinitions);

}

}

### RouteDefinitionRouteLocator

RouteDefinitionRouteLocator 将外部化配置的 RouteDefinition、FilterDefinition、PredicateDefinition 转换成 Route、AsyncPredicate、GatewayFilter。上面讲到的核心配置类 GatewayAutoConfiguration作为分析的入口，代码如下：

@Bean

public RouteLocator routeDefinitionRouteLocator(GatewayProperties properties, List<GatewayFilterFactory> gatewayFilters, List<RoutePredicateFactory> predicates, RouteDefinitionLocator routeDefinitionLocator, ConfigurationService configurationService) {

// 这里进行转换

return new RouteDefinitionRouteLocator(routeDefinitionLocator, predicates, gatewayFilters, properties, configurationService);

}

外部化配置进行转换，RouteDefinitionRouteLocator 是 RouteLocator 的实现：

public class RouteDefinitionRouteLocator implements RouteLocator {

protected final Log logger = LogFactory.getLog(this.getClass());

public static final String DEFAULT\_FILTERS = "defaultFilters";

// RouteDefinition Locator，一个 RouteDefinitionLocator 对象。

private final RouteDefinitionLocator routeDefinitionLocator;

private final ConversionService conversionService;

/\*\*

\* predicates factories，Predicate 工厂列表，会被映射成 key 为 name, value 为 factory 的 Map。

\* 可以猜想出 gateway 是如何根据 PredicateDefinition 中定义的 name 来匹配到相对应的 factory 了。

\* key ：{@link RoutePredicateFactory#name()}

\*/

private final Map<String, RoutePredicateFactory> predicates = new LinkedHashMap();

/\*\*

\* filter factories，Gateway Filter 工厂列表，同样会被映射成 key 为 name, value 为 factory 的 Map。

\* key ：{@link GatewayFilterFactory#name()}

\*/

private final Map<String, GatewayFilterFactory> gatewayFilterFactories = new HashMap();

// gateway properties，外部化配置类。

private final GatewayProperties gatewayProperties;

private final SpelExpressionParser parser = new SpelExpressionParser();

private BeanFactory beanFactory;

private ApplicationEventPublisher publisher;

public RouteDefinitionRouteLocator(RouteDefinitionLocator routeDefinitionLocator, List<RoutePredicateFactory> predicates, List<GatewayFilterFactory> gatewayFilterFactories, GatewayProperties gatewayProperties, ConversionService conversionService) {

// 设置 RouteDefinitionLocator

this.routeDefinitionLocator = routeDefinitionLocator;

this.conversionService = conversionService;

// ① 初始化 RoutePredicateFactory

this.initFactories(predicates);

// ② 初始化 gatewayFilterFactories

gatewayFilterFactories.forEach((factory) -> {

GatewayFilterFactory var10000 = (GatewayFilterFactory)this.gatewayFilterFactories.put(factory.name(), factory);

});

// 设置 GatewayProperties

this.gatewayProperties = gatewayProperties;

}

public void setBeanFactory(BeanFactory beanFactory) throws BeansException {

this.beanFactory = beanFactory;

}

public void setApplicationEventPublisher(ApplicationEventPublisher publisher) {

this.publisher = publisher;

}

// ③ 实现 RouteLocator 的 getRoutes() 方法 获取 route

public Flux<Route> getRoutes() {

// 调用 convertToRoute 方法将 RouteDefinition 转换成 Route。

Flux<Route> routes = this.routeDefinitionLocator.getRouteDefinitions().map(this::convertToRoute);

if (!this.gatewayProperties.isFailOnRouteDefinitionError()) {

routes = routes.onErrorContinue((error, obj) -> {

if (this.logger.isWarnEnabled()) {

this.logger.warn("RouteDefinition id " + ((RouteDefinition)obj).getId() + " will be ignored. Definition has invalid configs, " + error.getMessage());

}

});

}

return routes.map((route) -> {

if (this.logger.isDebugEnabled()) {

this.logger.debug("RouteDefinition matched: " + route.getId());

}

return route;

});

}

...

}

#### 初始化RoutePredicateFactory

private void initFactories(List<RoutePredicateFactory> predicates) {

predicates.forEach((factory) -> {

String key = factory.name();

if (this.predicates.containsKey(key)) {

this.logger.warn("A RoutePredicateFactory named " + key + " already exists, class: " + this.predicates.get(key) + ". It will be overwritten.");

}

this.predicates.put(key, factory);

if (this.logger.isInfoEnabled()) {

this.logger.info("Loaded RoutePredicateFactory [" + key + "]");

}

});

}

#### 初始化gatewayFilterFactories

// FilterDefinition 转换成 GatewayFilter

private List<GatewayFilter> getFilters(RouteDefinition routeDefinition) {

List<GatewayFilter> filters = new ArrayList();

// 处理 GatewayProperties 中定义的默认的 FilterDefinition，转换成 GatewayFilter。

if (!this.gatewayProperties.getDefaultFilters().isEmpty()) {

filters.addAll(this.loadGatewayFilters("defaultFilters", this.gatewayProperties.getDefaultFilters()));

}

// 将 RouteDefinition 中定义的 FilterDefinition 转换成 GatewayFilter。

if (!routeDefinition.getFilters().isEmpty()) {

filters.addAll(this.loadGatewayFilters(routeDefinition.getId(), routeDefinition.getFilters()));

}

// 对 GatewayFilter 进行排序，排序的详细逻辑请查阅 spring 中的 Ordered 接口。

AnnotationAwareOrderComparator.sort(filters);

return filters;

}

实现 RouteLocator 的 getRoutes() 方法 获取 Route，真正转换的方法：

public Flux<Route> getRoutes() {

// 调用 convertToRoute 方法将 RouteDefinition 转换成 Route。

return this.routeDefinitionLocator.getRouteDefinitions().map(this::convertToRoute).map((route) -> {

if (this.logger.isDebugEnabled()) {

this.logger.debug("RouteDefinition matched: " + route.getId());

}

return route;

});

}

private Route convertToRoute(RouteDefinition routeDefinition) {

// 2.3.1 将 PredicateDefinition 转换成 AsyncPredicate。

AsyncPredicate<ServerWebExchange> predicate = this.combinePredicates(routeDefinition);

// 2.3.2 将 FilterDefinition 转换成 GatewayFilter。

List<GatewayFilter> gatewayFilters = this.getFilters(routeDefinition);

// 2.3.3 根据 1 和 2 两步骤定义的变量生成 Route 对象。

return ((AsyncBuilder)Route.async(routeDefinition).asyncPredicate(predicate).replaceFilters(gatewayFilters)).build();

}

将PredicateDefinition转换成AsyncPredicate：

private AsyncPredicate<ServerWebExchange> combinePredicates(RouteDefinition routeDefinition) {

List<PredicateDefinition> predicates = routeDefinition.getPredicates();

// ① 调用 lookup 方法，将列表中第一个 PredicateDefinition 转换成 AsyncPredicate。

AsyncPredicate<ServerWebExchange> predicate = this.lookup(routeDefinition, (PredicateDefinition)predicates.get(0));

AsyncPredicate found;

// ② 循环调用，将列表中每一个 PredicateDefinition 都转换成 AsyncPredicate。

// ③ 应用and操作，将所有的 AsyncPredicate 组合成一个 AsyncPredicate 对象。

for(Iterator var4 = predicates.subList(1, predicates.size()).iterator(); var4.hasNext(); predicate = predicate.and(found)) {

PredicateDefinition andPredicate = (PredicateDefinition)var4.next();

found = this.lookup(routeDefinition, andPredicate);

}

return predicate;

}

将 FilterDefinition 转换成 GatewayFilter：

// FilterDefinition 转换成 GatewayFilter

private List<GatewayFilter> getFilters(RouteDefinition routeDefinition) {

List<GatewayFilter> filters = new ArrayList();

// ① 处理 GatewayProperties 中定义的默认的 FilterDefinition，转换成 GatewayFilter。

if (!this.gatewayProperties.getDefaultFilters().isEmpty()) {

filters.addAll(this.loadGatewayFilters("defaultFilters", this.gatewayProperties.getDefaultFilters()));

}

// ② 将 RouteDefinition 中定义的 FilterDefinition 转换成 GatewayFilter。

if (!routeDefinition.getFilters().isEmpty()) {

filters.addAll(this.loadGatewayFilters(routeDefinition.getId(), routeDefinition.getFilters()));

}

// ③ 对 GatewayFilter 进行排序，排序的详细逻辑请查阅 spring 中的 Ordered 接口。

AnnotationAwareOrderComparator.sort(filters);

return filters;

}

生成 Route 对象：

public Route build() {

Assert.notNull(this.id, "id can not be null");

Assert.notNull(this.uri, "uri can not be null");

AsyncPredicate<ServerWebExchange> predicate = this.getPredicate();

Assert.notNull(predicate, "predicate can not be null");

return new Route(this.id, this.uri, this.order, predicate, this.gatewayFilters);

}

# Predicate

Predicate 对象是由 RoutePredicateFactory 工厂类创建。

## RoutePredicateFactory

@FunctionalInterface

public interface RoutePredicateFactory<C> extends ShortcutConfigurable, Configurable<C> {

String PATTERN\_KEY = "pattern";

default Predicate<ServerWebExchange> apply(Consumer<C> consumer) {

C config = this.newConfig();

consumer.accept(config);

this.beforeApply(config);

return this.apply(config);

}

default AsyncPredicate<ServerWebExchange> applyAsync(Consumer<C> consumer) {

C config = this.newConfig();

consumer.accept(config);

this.beforeApply(config);

return this.applyAsync(config);

}

default Class<C> getConfigClass() {

throw new UnsupportedOperationException("getConfigClass() not implemented");

}

default C newConfig() {

throw new UnsupportedOperationException("newConfig() not implemented");

}

default void beforeApply(C config) {

}

Predicate<ServerWebExchange> apply(C config);

default AsyncPredicate<ServerWebExchange> applyAsync(C config) {

return ServerWebExchangeUtils.toAsyncPredicate(this.apply(config));

}

default String name() {

return NameUtils.normalizeRoutePredicateName(this.getClass());

}

}

* #name() 默认方法，调用 NameUtils#normalizePredicateName(Class) 方法，获得 RoutePredicateFactory 的名字。该方法截取类名前半段，例如 QueryRoutePredicateFactory 的结果为 Query 。
* #apply() 接口方法，创建 Predicate 。

可以直接看到处理器类与相关谓词工厂类如下：

[处理器类与相关谓词工厂类](处理器类与相关谓词工厂类.png)

对相关谓词工厂进行分类：

[谓词分类](谓词分类.png)

### AfterRoutePredicateFactory

* Route 匹配 ：请求时间满足在配置时间之后
* 配置：

spring:

cloud:

gateway:

routes:

- id: after\_route

uri: http://example.org

predicates:

- After=2017-01-20T17:42:47.789-07:00[America/Denver]

* 代码：

public Predicate<ServerWebExchange> apply(AfterRoutePredicateFactory.Config config) {

return new GatewayPredicate() {

public boolean test(ServerWebExchange serverWebExchange) {

ZonedDateTime now = ZonedDateTime.now();

return now.isAfter(config.getDatetime());

}

public Object getConfig() {

return config;

}

public String toString() {

return String.format("After: %s", config.getDatetime());

}

};

}

### BeforeRoutePredicateFactory

* Route 匹配 ：请求时间满足在配置时间之前
* 配置：

spring:

cloud:

gateway:

routes:

- id: before\_route

uri: http://example.org

predicates:

- Before=2017-01-20T17:42:47.789-07:00[America/Denver]

* 代码：

public Predicate<ServerWebExchange> apply(BeforeRoutePredicateFactory.Config config) {

return new GatewayPredicate() {

public boolean test(ServerWebExchange serverWebExchange) {

ZonedDateTime now = ZonedDateTime.now();

return now.isBefore(config.getDatetime());

}

public Object getConfig() {

return config;

}

public String toString() {

return String.format("Before: %s", config.getDatetime());

}

};

}

### BetweenRoutePredicateFactory

* Route 匹配 ：请求时间满足在配置时间之间
* 配置：

spring:

cloud:

gateway:

routes:

- id: between\_route

uri: http://example.org

predicates:

- Betweeen=2017-01-20T17:42:47.789-07:00[America/Denver], 2017-01-21T17:42:47.789

代码：

public Predicate<ServerWebExchange> apply(BetweenRoutePredicateFactory.Config config) {

Assert.isTrue(config.getDatetime1().isBefore(config.getDatetime2()), config.getDatetime1() + " must be before " + config.getDatetime2());

return new GatewayPredicate() {

public boolean test(ServerWebExchange serverWebExchange) {

ZonedDateTime now = ZonedDateTime.now();

return now.isAfter(config.getDatetime1()) && now.isBefore(config.getDatetime2());

}

public Object getConfig() {

return config;

}

public String toString() {

return String.format("Between: %s and %s", config.getDatetime1(), config.getDatetime2());

}

};

}

### CookieRoutePredicateFactory

* Route 匹配 ：请求指定 Cookie 正则匹配指定值
* 配置：

spring:

cloud:

gateway:

routes:

- id: cookie\_route

uri: http://example.org

predicates:

- Cookie=chocolate, ch.p

* 代码：

public Predicate<ServerWebExchange> apply(CookieRoutePredicateFactory.Config config) {

return new GatewayPredicate() {

public boolean test(ServerWebExchange exchange) {

List<HttpCookie> cookies = (List)exchange.getRequest().getCookies().get(config.name);

if (cookies == null) {

return false;

} else {

Iterator var3 = cookies.iterator();

HttpCookie cookie;

do {

if (!var3.hasNext()) {

return false;

}

cookie = (HttpCookie)var3.next();

} while(!cookie.getValue().matches(config.regexp));

return true;

}

}

public Object getConfig() {

return config;

}

public String toString() {

return String.format("Cookie: name=%s regexp=%s", config.name, config.regexp);

}

};

}

### HeaderRoutePredicateFactory

* Route 匹配 ：请求头满足匹配
* 配置：

spring:

cloud:

gateway:

routes:

- id: header\_route

uri: http://example.org

predicates:

- Header=X-Request-Id, \d+

* 代码：

public Predicate<ServerWebExchange> apply(HeaderRoutePredicateFactory.Config config) {

boolean hasRegex = !StringUtils.isEmpty(config.regexp);

return (exchange) -> {

List<String> values = (List)exchange.getRequest().getHeaders().getOrDefault(config.header, Collections.emptyList());

if (values.isEmpty()) {

return false;

} else {

return hasRegex ? values.stream().anyMatch((value) -> {

return value.matches(config.regexp);

}) : true;

}

};

}

### HostRoutePredicateFactory

* Route 匹配 ：请求 Host 匹配指定值
* 配置：

spring:

cloud:

gateway:

routes:

- id: host\_route

uri: http://example.org

predicates:

- Host=\*\*.somehost.org

* 代码：

public Predicate<ServerWebExchange> apply(HostRoutePredicateFactory.Config config) {

return new GatewayPredicate() {

public boolean test(ServerWebExchange exchange) {

String host = exchange.getRequest().getHeaders().getFirst("Host");

String match = null;

for(int i = 0; i < config.getPatterns().size(); ++i) {

String pattern = (String)config.getPatterns().get(i);

if (HostRoutePredicateFactory.this.pathMatcher.match(pattern, host)) {

match = pattern;

break;

}

}

if (match != null) {

Map<String, String> variables = HostRoutePredicateFactory.this.pathMatcher.extractUriTemplateVariables(match, host);

ServerWebExchangeUtils.putUriTemplateVariables(exchange, variables);

return true;

} else {

return false;

}

}

public Object getConfig() {

return config;

}

public String toString() {

return String.format("Hosts: %s", config.getPatterns());

}

};

}

### MethodRoutePredicateFactory

* Route 匹配 ：请求 Method 匹配指定值
* 配置：

spring:

cloud:

gateway:

routes:

- id: method\_route

uri: http://example.org

predicates:

- Method=GET

* 代码：

public Predicate<ServerWebExchange> apply(MethodRoutePredicateFactory.Config config) {

return new GatewayPredicate() {

public boolean test(ServerWebExchange exchange) {

HttpMethod requestMethod = exchange.getRequest().getMethod();

return Arrays.stream(config.getMethods()).anyMatch((httpMethod) -> {

return httpMethod == requestMethod;

});

}

public String toString() {

return String.format("Methods: %s", Arrays.toString(config.getMethods()));

}

};

}

### PathRoutePredicateFactory

* Route 匹配 ：请求 Path 匹配指定值
* 配置：

spring:

cloud:

gateway:

routes:

- id: host\_route

uri: http://example.org

predicates:

- Path=/foo/{segment}

* 代码：

public Predicate<ServerWebExchange> apply(PathRoutePredicateFactory.Config config) {

final ArrayList<PathPattern> pathPatterns = new ArrayList();

synchronized(this.pathPatternParser) {

this.pathPatternParser.setMatchOptionalTrailingSeparator(config.isMatchTrailingSlash());

config.getPatterns().forEach((pattern) -> {

PathPattern pathPattern = this.pathPatternParser.parse(pattern);

pathPatterns.add(pathPattern);

});

}

return new GatewayPredicate() {

public boolean test(ServerWebExchange exchange) {

PathContainer path = PathContainer.parsePath(exchange.getRequest().getURI().getRawPath());

PathPattern match = null;

for(int i = 0; i < pathPatterns.size(); ++i) {

PathPattern pathPattern = (PathPattern)pathPatterns.get(i);

if (pathPattern.matches(path)) {

match = pathPattern;

break;

}

}

if (match != null) {

PathRoutePredicateFactory.traceMatch("Pattern", match.getPatternString(), path, true);

PathMatchInfo pathMatchInfo = match.matchAndExtract(path);

ServerWebExchangeUtils.putUriTemplateVariables(exchange, pathMatchInfo.getUriVariables());

exchange.getAttributes().put(ServerWebExchangeUtils.GATEWAY\_PREDICATE\_MATCHED\_PATH\_ATTR, match.getPatternString());

String routeId = (String)exchange.getAttributes().get(ServerWebExchangeUtils.GATEWAY\_PREDICATE\_ROUTE\_ATTR);

if (routeId != null) {

exchange.getAttributes().put(ServerWebExchangeUtils.GATEWAY\_PREDICATE\_MATCHED\_PATH\_ROUTE\_ID\_ATTR, routeId);

}

return true;

} else {

PathRoutePredicateFactory.traceMatch("Pattern", config.getPatterns(), path, false);

return false;

}

}

public Object getConfig() {

return config;

}

public String toString() {

return String.format("Paths: %s, match trailing slash: %b", config.getPatterns(), config.isMatchTrailingSlash());

}

};

}

### QueryRoutePredicateFactory

* Route 匹配 ：请求 QueryParam 匹配指定值
* 配置：

spring:

cloud:

gateway:

routes:

- id: query\_route

uri: http://example.org

predicates:

- Query=baz

- Query=foo, ba.

* 代码：

public Predicate<ServerWebExchange> apply(QueryRoutePredicateFactory.Config config) {

return new GatewayPredicate() {

public boolean test(ServerWebExchange exchange) {

if (!StringUtils.hasText(config.regexp)) {

return exchange.getRequest().getQueryParams().containsKey(config.param);

} else {

List<String> values = (List)exchange.getRequest().getQueryParams().get(config.param);

if (values == null) {

return false;

} else {

Iterator var3 = values.iterator();

String value;

do {

if (!var3.hasNext()) {

return false;

}

value = (String)var3.next();

} while(value == null || !value.matches(config.regexp));

return true;

}

}

}

public Object getConfig() {

return config;

}

public String toString() {

return String.format("Query: param=%s regexp=%s", config.getParam(), config.getRegexp());

}

};

}

### RemoteAddrRoutePredicateFactory

* Route 匹配 ：请求来源 IP 在指定范围内
* 配置：

spring:

cloud:

gateway:

routes:

- id: remoteaddr\_route

uri: http://example.org

predicates:

- RemoteAddr=192.168.1.1/24

* 代码：

public Predicate<ServerWebExchange> apply(RemoteAddrRoutePredicateFactory.Config config) {

final List<IpSubnetFilterRule> sources = this.convert(config.sources);

return new GatewayPredicate() {

public boolean test(ServerWebExchange exchange) {

InetSocketAddress remoteAddress = config.remoteAddressResolver.resolve(exchange);

if (remoteAddress != null && remoteAddress.getAddress() != null) {

String hostAddress = remoteAddress.getAddress().getHostAddress();

String host = exchange.getRequest().getURI().getHost();

if (RemoteAddrRoutePredicateFactory.log.isDebugEnabled() && !hostAddress.equals(host)) {

RemoteAddrRoutePredicateFactory.log.debug("Remote addresses didn't match " + hostAddress + " != " + host);

}

Iterator var5 = sources.iterator();

while(var5.hasNext()) {

IpSubnetFilterRule source = (IpSubnetFilterRule)var5.next();

if (source.matches(remoteAddress)) {

return true;

}

}

}

return false;

}

public Object getConfig() {

return config;

}

public String toString() {

return String.format("RemoteAddrs: %s", config.getSources());

}

};

}

## RoutePredicateHandlerMapping

Spring Cloud Gateway 官网提供的架构图：

[Spring Cloud Gateway 架构图](Spring%20Cloud%20Gateway%20架构图.png)

客户端发送请求过来，通过 HandlerMapping 进行 predicate 的匹配，匹配成功再进行下面的处理。

### DispatcherHandler

接收到请求，匹配 HandlerMapping ，此处会匹配到 RoutePredicateHandlerMapping。由于 Gateway 是构建在 reactive 上的，所以这边的 web 类型就是 reactive。

public class DispatcherHandler implements WebHandler, ApplicationContextAware {

private static final Exception HANDLER\_NOT\_FOUND\_EXCEPTION;

@Nullable

private List<HandlerMapping> handlerMappings;

@Nullable

private List<HandlerAdapter> handlerAdapters;

@Nullable

private List<HandlerResultHandler> resultHandlers;

public Mono<Void> handle(ServerWebExchange exchange) {

return this.handlerMappings == null ? this.createNotFoundError() :

// 顺序使用 handlerMappings 获得对应的 WebHandler

Flux.fromIterable(this.handlerMappings).concatMap((mapping) -> {

// 获得 Handler

return mapping.getHandler(exchange);

// 如果匹配不到 WebHandler ，返回 HANDLER\_NOT\_FOUND\_EXCEPTION 。

}).next().switchIfEmpty(this.createNotFoundError()).flatMap((handler) -> {

// 调用 invokeHandler() 方法，执行 Handler 。

return this.invokeHandler(exchange, handler);

}).flatMap((result) -> {

// 调用 handleResult() 方法，处理结果

return this.handleResult(exchange, result);

});

}

...

}

invokeHandler() 方法：

private Mono<HandlerResult> invokeHandler(ServerWebExchange exchange, Object handler) {

if (this.handlerAdapters != null) {

// 获取Adapters， WebHandler 的处理器适配器。

Iterator var3 = this.handlerAdapters.iterator();

while(var3.hasNext()) {

HandlerAdapter handlerAdapter = (HandlerAdapter)var3.next();

// 调用support方法 ，是否支持 WebHandler

if (handlerAdapter.supports(handler)) {

// 调用handle 方法，执行处理器

return handlerAdapter.handle(exchange, handler);

}

}

}

return Mono.error(new IllegalStateException("No HandlerAdapter: " + handler));

}

public boolean supports(Object handler) {

return WebHandler.class.isAssignableFrom(handler.getClass());

}

public Mono<HandlerResult> handle(ServerWebExchange exchange, Object handler) {

WebHandler webHandler = (WebHandler)handler;

// 执行处理器。例如，WebHandler 为 FilteringWebHandler 时，获得 Route 的 GatewayFilter 数组，创建 GatewayFilterChain 处理请求。

Mono<Void> mono = webHandler.handle(exchange);

// 在 WebHandler 执行完后 #then(Mongo)，然后返回 Mono.empty() 。

return mono.then(Mono.empty());

}

SimpleHandlerAdapter 返回的是 Mono.empty() ，所以不会触发该方法。

private Mono<Void> handleResult(ServerWebExchange exchange, HandlerResult result) {

return this.getResultHandler(result).handleResult(exchange, result).onErrorResume((ex) -> {

return result.applyExceptionHandler(ex).flatMap((exceptionResult) -> {

return this.getResultHandler(exceptionResult).handleResult(exchange, exceptionResult);

});

});

}

### RoutePredicateHandlerMapping

接收到请求，匹配Route，并返回处理Route的FilteringWebHandler。SimpleHandlerAdapter#handle(ServerWebExchange,Object)调用 FilteringWebHandler#handle(ServerWebExchange) 方法，处理请求。

public class RoutePredicateHandlerMapping extends AbstractHandlerMapping {

private final FilteringWebHandler webHandler;

private final RouteLocator routeLocator;

private final Integer managmentPort;

public RoutePredicateHandlerMapping(FilteringWebHandler webHandler, RouteLocator routeLocator, GlobalCorsProperties globalCorsProperties, Environment environment) {

this.webHandler = webHandler;

this.routeLocator = routeLocator;

if (environment.containsProperty("management.server.port")) {

this.managmentPort = new Integer(environment.getProperty("management.server.port"));

} else {

this.managmentPort = null;

}

// RequestMappingHandlerMapping 之后

this.setOrder(1);

this.setCorsConfigurations(globalCorsProperties.getCorsConfigurations());

}

protected Mono<?> getHandlerInternal(ServerWebExchange exchange) {

if (this.managmentPort != null && exchange.getRequest().getURI().getPort() == this.managmentPort) {

return Mono.empty();

} else {

// 设置 GATEWAY\_HANDLER\_MAPPER\_ATTR 为

RoutePredicateHandlerMappingexchange.getAttributes().put(ServerWebExchangeUtils.GATEWAY\_HANDLER\_MAPPER\_ATTR, this.getSimpleName());

// 匹配路由

return this.lookupRoute(exchange).flatMap((r) -> {

exchange.getAttributes().remove(ServerWebExchangeUtils.GATEWAY\_PREDICATE\_ROUTE\_ATTR);

if (this.logger.isDebugEnabled()) {

this.logger.debug("Mapping [" + this.getExchangeDesc(exchange) + "] to " + r);

}

// 设置 GATEWAY\_ROUTE\_ATTR 为 匹配的 Route

exchange.getAttributes().put(ServerWebExchangeUtils.GATEWAY\_ROUTE\_ATTR, r);

return Mono.just(this.webHandler);

}).switchIfEmpty(Mono.empty().then(Mono.fromRunnable(() -> { //匹配不到返回

exchange.getAttributes().remove(ServerWebExchangeUtils.GATEWAY\_PREDICATE\_ROUTE\_ATTR);

if (this.logger.isTraceEnabled()) {

this.logger.trace("No RouteDefinition found for [" + this.getExchangeDesc(exchange) + "]");

}

})));

}

}

}

跟一下 lookupRoute 匹配路由，这个方法是网关的核心：

protected Mono<Route> lookupRoute(ServerWebExchange exchange) {

// 获取所有路由

return this.routeLocator.getRoutes().concatMap((route) -> {

return Mono.just(route).filterWhen((r) -> {

exchange.getAttributes().put(ServerWebExchangeUtils.GATEWAY\_PREDICATE\_ROUTE\_ATTR, r.getId());

// 并调用 Predicate#apply(ServerWebExchange) 方法，顺序匹配一个 Route。

return (Publisher)r.getPredicate().apply(exchange);

// 未来会增加匹配过程中发生异常的处理。目前，任何一个 Predicate#test(ServerWebExchange) 的方法调用发生异常时，都会导致匹配不到 Route。一定要注意。

}).doOnError((e) -> {

this.logger.error("Error applying predicate for route: " + route.getId(), e);

}).onErrorResume((e) -> {

return Mono.empty();

});

}).next().map((route) -> {

if (this.logger.isDebugEnabled()) {

this.logger.debug("Route matched: " + route.getId());

}

this.validateRoute(route, exchange);

return route;

});

}

# Filter Chain过滤器链

[Spring Cloud Gateway 的整体流程图](Spring%20Cloud%20Gateway%20的整体流程图.png)

## FilteringWebHandler

// org.springframework.cloud.gateway.handler.FilteringWebHandler

/\*\*

\* 通过过滤器处理web请求的处理器

\*/

public class FilteringWebHandler implements WebHandler {

protected static final Log logger = LogFactory.getLog(FilteringWebHandler.class);

/\*\*

\* 全局过滤器

\*/

private final List<GatewayFilter> globalFilters;

public FilteringWebHandler(List<GlobalFilter> globalFilters) {

this.globalFilters = loadFilters(globalFilters);

}

/\*\*

\* 组成过滤链

\* 包装加载全局的过滤器，将全局过滤器包装成GatewayFilter

\*/

private static List<GatewayFilter> loadFilters(List<GlobalFilter> filters) {

return (List)filters.stream().map((filter) -> {

FilteringWebHandler.GatewayFilterAdapter gatewayFilter = new FilteringWebHandler.GatewayFilterAdapter(filter);

// 当 GlobalFilter 子类实现了 org.springframework.core.Ordered 接口，在委托一层 OrderedGatewayFilter 。

// 这样 AnnotationAwareOrderComparator#sort(List) 方法好排序。

if (filter instanceof Ordered) {

int order = ((Ordered)filter).getOrder();

return new OrderedGatewayFilter(gatewayFilter, order);

} else {

return gatewayFilter;

}

}).collect(Collectors.toList());

}

// 按照过滤链的顺序依次执行

public Mono<Void> handle(ServerWebExchange exchange) {

// 获取请求上下文设置的路由实例

Route route = (Route)exchange.getRequiredAttribute(ServerWebExchangeUtils.GATEWAY\_ROUTE\_ATTR);

// 获取路由定义下的网关过滤器集合

List<GatewayFilter> gatewayFilters = route.getFilters();

// 组合全局的过滤器与路由配置的过滤器

List<GatewayFilter> combined = new ArrayList(this.globalFilters);

// 添加路由配置过滤器到集合尾部

combined.addAll(gatewayFilters);

// 对过滤器进行排序

AnnotationAwareOrderComparator.sort(combined);

if (logger.isDebugEnabled()) {

logger.debug("Sorted gatewayFilterFactories: " + combined);

}

// 创建过滤器链表对其进行链式调用

return (new FilteringWebHandler.DefaultGatewayFilterChain(combined)).filter(exchange);

}

...

}

FilteringWebHandler 的执行顺序：

* 构建一个包含全局过滤器的集合（combined）
* 获取上下文中的路由信息 GATEWAY\_ROUTE\_ATTR
* 将路由里的过滤器添加到集合中（combined）
* 对过滤器集合进行排序操作
* 通过过滤器集合组装过滤器链表，并进行调用（DefaultGatewayFilterChain 与 Servlet 中的 FilterChain 的原理是一致的）
* 通过过滤器来处理请求到具体业务服务

## GatewayFilter 与 GlobalFilter 的关系

private final List<GatewayFilter> globalFilters;

public FilteringWebHandler(List<GlobalFilter> globalFilters) {

this.globalFilters = loadFilters(globalFilters);

}

### GatewayFilter 网关过滤器接口

GatewayFilter网关过滤器接口：

public interface GatewayFilter extends ShortcutConfigurable {

String NAME\_KEY = "name";

String VALUE\_KEY = "value";

Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain);

}

GatewayFilter 有三种类型的子类实现：

* OrderedGatewayFilter ：有序的网关过滤器实现类。在 FilterChain 里，过滤器数组首先会按照 order 升序排序，按照顺序过滤请求。

public class OrderedGatewayFilter implements GatewayFilter, Ordered {

// 委托的 GatewayFilter

private final GatewayFilter delegate;

// order代表顺序

private final int order;

public OrderedGatewayFilter(GatewayFilter delegate, int order) {

this.delegate = delegate;

this.order = order;

}

public GatewayFilter getDelegate() {

return this.delegate;

}

public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

return this.delegate.filter(exchange, chain);

}

public int getOrder() {

return this.order;

}

public String toString() {

return "[" + this.delegate + ", order = " + this.order + "]";

}

}

* GatewayFilterAdapter：网关过滤器适配器。在 GatewayFilterChain 使用 GatewayFilter 过滤请求，所以通过 GatewayFilterAdapter 将 GlobalFilter 适配成 GatewayFilter。

private static class GatewayFilterAdapter implements GatewayFilter {

private final GlobalFilter delegate;

GatewayFilterAdapter(GlobalFilter delegate) {

this.delegate = delegate;

}

public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

return this.delegate.filter(exchange, chain);

}

public String toString() {

StringBuilder sb = new StringBuilder("GatewayFilterAdapter{");

sb.append("delegate=").append(this.delegate);

sb.append('}');

return sb.toString();

}

}

* ModifyResponseGatewayFilter：用于修改 Response

public class ModifyResponseGatewayFilter implements GatewayFilter, Ordered {

private final ModifyResponseBodyGatewayFilterFactory.Config config;

public ModifyResponseGatewayFilter(ModifyResponseBodyGatewayFilterFactory.Config config) {

this.config = config;

}

public Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain) {

ServerHttpResponseDecorator responseDecorator = new ServerHttpResponseDecorator(exchange.getResponse()) {

public Mono<Void> writeWith(Publisher<? extends DataBuffer> body) {

Class inClass = ModifyResponseGatewayFilter.this.config.getInClass();

Class outClass = ModifyResponseGatewayFilter.this.config.getOutClass();

String originalResponseContentType = (String)exchange.getAttribute("original\_response\_content\_type");

HttpHeaders httpHeaders = new HttpHeaders();

httpHeaders.add("Content-Type", originalResponseContentType);

ModifyResponseBodyGatewayFilterFactory.ResponseAdapter responseAdapter = ModifyResponseBodyGatewayFilterFactory.this.new ResponseAdapter(body, httpHeaders);

DefaultClientResponse clientResponse = new DefaultClientResponse(responseAdapter, ExchangeStrategies.withDefaults());

Mono modifiedBody = clientResponse.bodyToMono(inClass).flatMap((originalBody) -> {

return ModifyResponseGatewayFilter.this.config.rewriteFunction.apply(exchange, originalBody);

});

BodyInserter bodyInserter = BodyInserters.fromPublisher(modifiedBody, outClass);

CachedBodyOutputMessage outputMessage = new CachedBodyOutputMessage(exchange, exchange.getResponse().getHeaders());

return bodyInserter.insert(outputMessage, new BodyInserterContext()).then(Mono.defer(() -> {

Flux<DataBuffer> messageBody = outputMessage.getBody();

HttpHeaders headers = this.getDelegate().getHeaders();

if (!headers.containsKey("Transfer-Encoding")) {

messageBody = messageBody.doOnNext((data) -> {

headers.setContentLength((long)data.readableByteCount());

});

}

return this.getDelegate().writeWith(messageBody);

}));

}

public Mono<Void> writeAndFlushWith(Publisher<? extends Publisher<? extends DataBuffer>> body) {

return this.writeWith(Flux.from(body).flatMapSequential((p) -> {

return p;

}));

}

};

return chain.filter(exchange.mutate().response(responseDecorator).build());

}

public int getOrder() {

return -2;

}

}

### GlobalFilter

全局过滤器接口，会作用到所有的Route上：

public interface GlobalFilter {

Mono<Void> filter(ServerWebExchange exchange, GatewayFilterChain chain);

}

## GatewayFilterFactory

GatewayFilterFactory子类分类图：

[GatewayFilterFactory子类分类图](GatewayFilterFactory子类分类图.png)