


差不多先生的个人空间 > 微服务 > 正文

## Feign 源码解析 转

 xiaomin0322 发布于 2018/04/04 19:56 字数 2951 阅读 1394 收藏 3 点赞 1 评论 0

撸了今年阿里、头条和美团的面试，我有一个重要发现.....>>> 

解决两个问题：

1. 请求是怎么转到 Feign 的
2. Feign 是怎么工作的

前者跟 Spring Boot 有关，但是也关系到了后者，即通过前者才能发现是创建了哪些 Bean 或者代理来处理请求。

从注解开始看，使用 Feign 涉及到了两个注解，一个是@EnableFeignClients，用来开启 Feign，另一个是@FeignClient，用来标记要用 Feign 来拦截的请求接口。

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Import({FeignClientsRegistrar.class})
public @interface EnableFeignClients {

    String[] value() default {};

    String[] basePackages() default {};

    Class<?>[] basePackageClasses() default {};

    Class<?>[] defaultConfiguration() default {};

    Class<?>[] clients() default {};
}
```

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface FeignClient {

    @AliasFor("name")
    String value() default "";

    @Deprecated
    String serviceId() default "";

    @AliasFor("value")
    String name() default "";

    String qualifier() default "";

    String url() default "";

    boolean decode404() default false;

    Class<?>[] configuration() default {};

    Class<?> fallback() default void.class;

    Class<?> fallbackFactory() default void.class;

    String path() default "";

    boolean primary() default true;
}
```

@EnableFeignClients 是关于注解扫描的配置，@FeignClient 则是关于对该接口进行代理的时候，一些实现细节的配置，比如 fallback 方法，关于404的请求是抛错误还是正常返回。

我们先关注对于 EnableFeignClients 的处理。它使用了 @Import(FeignClientsRegistrar.class)，我们知道在 Spring Context 的处理里面，这个 Import 会在解析 Configuration 的时候当做提供了其他的 bean definition 的扩展，Spring 通过调用其 registerBeanDefinitions 方法来获取其提供的 bean definition。

```
public void registerBeanDefinitions(AnnotationMetadata metadata,
    BeanDefinitionRegistry registry) {
    registerDefaultConfiguration(metadata, registry);
    registerFeignClients(metadata, registry);
}

private void registerDefaultConfiguration(AnnotationMetadata metadata,
    BeanDefinitionRegistry registry) {
    Map<String, Object> defaultAttrs = metadata
        .getAnnotationAttributes(EnableFeignClients.class.getName(), true);

    if (defaultAttrs != null && defaultAttrs.containsKey("defaultConfiguration")) {
        String name;
        if (metadata.hasEnclosingClass()) {
            name = "default." + metadata.getEnclosingClassName();
        } else {
            name = "default." + metadata.getClassName();
        }
        registerClientConfiguration(registry, name,
            defaultAttrs.get("defaultConfiguration"));
    }
}

public void registerFeignClients(AnnotationMetadata metadata,
    BeanDefinitionRegistry registry) {
    ClassPathScanningCandidateComponentProvider scanner = getScanner();
    scanner.setResourceLoader(this.resourceLoader);

    Set<String> basePackages;

    Map<String, Object> attrs = metadata
        .getAnnotationAttributes(EnableFeignClients.class.getName());
    AnnotationTypeFilter annotationTypeFilter = new AnnotationTypeFilter(
        FeignClient.class);
    final Class<?>[] clients = attrs == null ? null
        : (Class<?>[]) attrs.get("clients");
    if (clients == null || clients.length == 0) {
        scanner.addIncludeFilter(annotationTypeFilter);
        basePackages = getBasePackages(metadata);
    } else {
        final Set<String> clientClasses = new HashSet<>();
        basePackages = new HashSet<>();
        for (Class<?> clazz : clients) {
            basePackages.add(ClassUtils.getPackageName(clazz));
            clientClasses.add(clazz.getCanonicalName());
        }
        AbstractClassTestingTypeFilter filter = new AbstractClassTestingTypeFilter() {
            @Override
            protected boolean match(ClassMetadata metadata) {
                String cleaned = metadata.getClassName().replaceAll("\\\\$", ".");
                return clientClasses.contains(cleaned);
            }
        };
        scanner.addIncludeFilter(
            new AllTypeFilter(Arrays.asList(filter, annotationTypeFilter)));
    }

    for (String basePackage : basePackages) {
        Set<BeanDefinition> candidateComponents = scanner
            .findCandidateComponents(basePackage);
        for (BeanDefinition candidateComponent : candidateComponents) {
            if (candidateComponent instanceof AnnotatedBeanDefinition) {
                // verify annotated class is an interface
                AnnotatedBeanDefinition beanDefinition = (AnnotatedBeanDefinition) candidateComponent;
                AnnotationMetadata annotationMetadata = beanDefinition.getMetadata();
                Assert.isTrue(annotationMetadata.isInterface(),
                    "@FeignClient can only be specified on an interface");

                Map<String, Object> attributes = annotationMetadata
                    .getAnnotationAttributes(
                        FeignClient.class.getCanonicalName());
            }
        }
    }
}
```

```
        String name = getClientName(attributes);
        registerClientConfiguration(registry, name,
                                   attributes.get("configuration"));

        registerFeignClient(registry, annotationMetadata, attributes);
    }
}

private void registerFeignClient(BeanDefinitionRegistry registry,
                                AnnotationMetadata annotationMetadata, Map<String, Object> attributes) {
    String className = annotationMetadata.getClassName();
    BeanDefinitionBuilder definition = BeanDefinitionBuilder
        .genericBeanDefinition(FeignClientFactoryBean.class);
    validate(attributes);
    definition.addPropertyValues("url", getUrl(attributes));
    definition.addPropertyValues("path", getPath(attributes));
    String name = getName(attributes);
    definition.addPropertyValues("name", name);
    definition.addPropertyValues("type", className);
    definition.addPropertyValues("decode404", attributes.get("decode404"));
    definition.addPropertyValues("fallback", attributes.get("fallback"));
    definition.addPropertyValues("fallbackFactory", attributes.get("fallbackFactory"));
    definition.setAutowireMode(AbstractBeanDefinition.AUTOWIRE_BY_TYPE);

    String alias = name + "FeignClient";
    AbstractBeanDefinition beanDefinition = definition.getBeanDefinition();

    boolean primary = (Boolean)attributes.get("primary"); // has a default, won't be null

    beanDefinition.setPrimary(primary);

    String qualifier = getQualifier(attributes);
    if (StringUtils.hasText(qualifier)) {
        alias = qualifier;
    }

    BeanDefinitionHolder holder = new BeanDefinitionHolder(beanDefinition, className,
        new String[] { alias });
    BeanDefinitionReaderUtils.registerBeanDefinition(holder, registry);
}
```

这里会往 Registry 里面添加两个 BeanDefinition，一个是 FeignClientSpecification，主要可调整项是通过 EnableFeignClients 注解的 defaultConfiguration 参数传入。另一个是负责注册 FeignClient，分为以下步骤

- 找到 basePackage 下面所有包含了 FeignClient 注解的类
- 读取类上面的 FeignClient 注解参数
- 如果该注解包括了 configuration 参数，则先注册 configuration 所指定的类。这个类也是包装在 FeignClientSpecification 里面的，也就是 bean 的类型其实是 FeignClientSpecification，在 FeignClient 上指定的 configuration 类是它的一个属性。
- 注册该注解了 FeignClient 的接口，生成 BeanDefinition 时是以 FeignClientFactoryBean 作为对象创建的，而使用了 FeignClient 注解的接口是作为该 Bean 的一个属性，同时，对于 FeignClient 注解配置的参数，比如 fallback 等都一并作为参数放入 BeanDefinition 中。**注意，configuration 没有传进去**

总结一下，在启动时，处理了 EnableFeignClients 注解后，registry 里面会多出一些关于 Feign 的 BeanDefinition，这些 BeanDefinition 分为两类：

- 一类是 FeignClientSpecification，包括了所有 FeignClient 上指定 configuration 以及在 EnableFeignClients 上指定的 defaultConfiguration。前者的名字为注解的 URL 或者 value 组成，比如 TEST-SERVICE，后者的名字为 default.\$CLASSNAME，比如 default.name.org.xiang.SpringBootApplication。
- 还有一类是 FeignClientFactoryBean，它包含了所有使用了 FeignClient 注解的接口信息以及注解上面的参数。它的名字为注解的 URL 或者 value，比如 TEST-SERVICE，跟它上面的 configuration 创建出来的 bean 定义是同一个名字。

来看下 FeignClientFactoryBean，它是一个工厂类，Spring Context 创建 Bean 实例时会调用它的 getObject 方法。

```
public Object getObject() throws Exception {
    FeignContext context = applicationContext.getBean(FeignContext.class);
    Feign.Builder builder = feign(context);

    if (!StringUtils.hasText(this.url)) {
        String url;
        if (!this.name.startsWith("http")) {
            url = "http://" + this.name;
        }
        else {
            url = this.name;
        }
        url += cleanPath();
        return loadBalance(builder, context, new HardCodedTarget<>(this.type,
            this.name, url));
    }
    if (StringUtils.hasText(this.url) && !this.url.startsWith("http")) {
        this.url = "http://" + this.url;
    }
    String url = this.url + cleanPath();
    Client client = getOptional(context, Client.class);
    if (client != null) {
        if (client instanceof LoadBalancerFeignClient) {
            // not load balancing because we have a url,
            // but ribbon is on the classpath, so unwrap
            client = ((LoadBalancerFeignClient)client).getDelegate();
        }
        builder.client(client);
    }
    Targeter targeter = get(context, Targeter.class);
    return targeter.target(this, builder, context, new HardCodedTarget<>(
        this.type, this.name, url));
}
```

在看具体的创建之前，我们还有一个问题没解决，那就是每个 FeignClient 实际上都有一个配套的 configuration 类，这个 configuration 怎么跟 client 关联起来？在前面注册 BeanDefinition 的时候，我们看到 configuration 其实也被作为参数，传给了 FeignClientSpecification。但是在 FeignClient 注册 BeanDefinition 的时候，configuration 被扔掉了。不过我们分析的时候看到了，两者的名字应该是一样的。

在通过 spring boot 自动配置的时候，spring-cloud 也提供了一个 Feign 的初始化配置类，FeignAutoConfiguration。它初始化了一个新的 FeignContext bean，并且把所有的 configuration 都放在 FeignContext 里面。

FeignContext 继承了 NamedContextFactory，它会管理一批 Context，外部调用的时候会指定用哪个 context 来寻找对应的 bean，而 context 如果不存在，则会创建一个新的 AnnotationConfigApplicationContext。创建 context 的时候，会用 name 去匹配已有的 configurations（加载该 FeignClient 注解里面提供的 configuration 属性类），如果有同名的，则就将该 configuration 注册进 context，另外如果有 default 开头的 configuration，也会将其注册到 context 里面，最后调用 refresh 方法对 context 进行初始化。初始化以后，相当于 configuration 里面提供的encoder，decoder 这些就逗号了。

在 FeignClientFactoryBean 创建 Bean 的时候，它先从 applicationContext 里面找到已经构建好的 feignContext 初始化一个 FeignBuilder，FeignBuilder 会利用 context 里面包含的对应的 configuration 指定的 bean，获取指定的类，比如 decoder，encoder，retryer，errorDecoder。然后再去寻找 context 中的 Client bean。这个 Client Bean 是什么呢？如果当前路径里面有 Ribbon，那么 Spring Boot 启动时就会创建一个 LoadBalancerFeignClient，如果没有，FeignAutoConfiguration 里面也会自己去创建 ApacheHttpClient 或者 OKHttpClient。FeignBuilder 会拿这个 client 配到自己里面。

然后是最关键的一步，获取 Targeter 来生成动态代理类，在 FeignAutoConfiguration 里面，指定了生成 HystrixTargeter。

```
return loadBalance(builder, context, new HardCodedTarget<>(this.type,
    this.name, url));

protected <T> T loadBalance(Feign.Builder builder, FeignContext context,
    HardCodedTarget<T> target) {
    Client client = getOptional(context, Client.class);
    if (client != null) {
        builder.client(client);
        Targeter targeter = get(context, Targeter.class);
        return targeter.target(this, builder, context, target);
    }

    throw new IllegalStateException(
        "No Feign Client for loadBalancing defined. Did you forget to include spring-cloud-starter-ribbon?");
}

public <T> T target(FeignClientFactoryBean factory, Feign.Builder feign, FeignContext context,
    Target.HardCodedTarget<T> target) {
    if (!(feign instanceof feign.hystrix.HystrixFeign.Builder)) {
        return feign.target(target);
    }
    feign.hystrix.HystrixFeign.Builder builder = (feign.hystrix.HystrixFeign.Builder) feign;
    SetterFactory setterFactory = getOptional(factory.getName(), context,
        SetterFactory.class);
    if (setterFactory != null) {
        builder.setterFactory(setterFactory);
    }
    Class<?> fallback = factory.getFallback();
    if (fallback != void.class) {
        return targetWithFallback(factory.getName(), context, target, builder, fallback);
    }
    Class<?> fallbackFactory = factory.getFallbackFactory();
    if (fallbackFactory != void.class) {
        return targetWithFallbackFactory(factory.getName(), context, target, builder, fallbackFactory);
    }

    return feign.target(target);
}
```



如果 FeignClient 没有定义 fallback, 或者说 Builder 不是 HystrixFeignBuilder, 则直接用 FeignBuidler 的 target 方法生成代理。

```
        public <T> T target(Target<T> target) {
            return build().newInstance(target);
        }

        public Feign build() {
            SynchronousMethodHandler.Factory synchronousMethodHandlerFactory =
                new SynchronousMethodHandler.Factory(client, retryer, requestInterceptors, logger,
                    logLevel, decode404);

            ParseHandlersByName handlersByName =
                new ParseHandlersByName(contract, options, encoder, decoder,
                    errorDecoder, synchronousMethodHandlerFactory);
            return new ReflectiveFeign(handlersByName, invocationHandlerFactory);
        }
    }
```

ReflectiveFeign 生成动态代理对象 (newInstance)

```
        public <T> T newInstance(Target<T> target) {
            //为每个方法创建一个SynchronousMethodHandler对象, 并放在 Map 里面。
            Map<String, MethodHandler> nameToHandler = targetToHandlersByName.apply(target);
            Map<Method, MethodHandler> methodToHandler = new LinkedHashMap<Method, MethodHandler>();
            List<DefaultMethodHandler> defaultMethodHandlers = new LinkedList<DefaultMethodHandler>();

            for (Method method : target.type().getMethods()) {
                if (method.getDeclaringClass() == Object.class) {
                    continue;
                } else if (Util.isDefault(method)) {
                    //如果是 default 方法, 说明已经有实现了, 用 DefaultHandler
                    DefaultMethodHandler handler = new DefaultMethodHandler(method);
                    defaultMethodHandlers.add(handler);
                    methodToHandler.put(method, handler);
                } else {
                    //否则就用上面的 SynchronousMethodHandler
                    methodToHandler.put(method, nameToHandler.get(Feign.configKey(target.type(), method)));
                }
            }
            // 创建动态代理, factory 是 InvocationHandlerFactory.Default, 创建出来的是 ReflectiveFeign.FeignInvocationHanlder, 也就是说后续对方法的调用都会进入到该对象的 inovke 方法。
            InvocationHandler handler = factory.create(target, methodToHandler);
            // 创建动态代理对象
            T proxy = (T) Proxy.newProxyInstance(target.type().getClassLoader(), new Class<?>[]{target.type()}, handler);

            for(DefaultMethodHandler defaultMethodHandler : defaultMethodHandlers) {
                defaultMethodHandler.bindTo(proxy);
            }
            return proxy;
        }
    }
```

```
        public Map<String, MethodHandler> apply(Target key) {
            //取出要实现的接口的所有方法
            List<MethodMetadata> metadata = contract.parseAndValidatateMetadata(key.type());
            Map<String, MethodHandler> result = new LinkedHashMap<String, MethodHandler>();
            for (MethodMetadata md : metadata) {
                //根据目标接口类和方法上的注解信息判断该用哪种 buildTemplate
                BuildTemplateByResolvingArgs buildTemplate;
                if (!md.formParams().isEmpty() && md.template().bodyTemplate() == null) {
                    buildTemplate = new BuildFormEncodedTemplateFromArgs(md, encoder);
                } else if (md.bodyIndex() != null) {
                    buildTemplate = new BuildEncodedTemplateFromArgs(md, encoder);
                } else {
                    buildTemplate = new BuildTemplateByResolvingArgs(md);
                }

                //调用synchronousMethodHandlerFactory来生成SynchronousMethodHandler对象。这个就是对接口某个方法的实现。
                result.put(md.configKey(),
                    factory.create(key, md, buildTemplate, options, decoder, errorDecoder));
            }
            return result;
        }
    }
```

synchronousMethodHandlerFactory 的 create 方法。

```
        public MethodHandler create(Target<?> target, MethodMetadata md,
            RequestTemplate.Factory buildTemplateFromArgs,
            Options options, Decoder decoder, ErrorDecoder errorDecoder) {
            return new SynchronousMethodHandler(target, client, retryer, requestInterceptors, logger,
                logLevel, md, buildTemplateFromArgs, options, decoder,
                errorDecoder, decode404);
        }
    }
```

ReflectiveFeign.FeignInvocationHanlder 的 invoke 方法。

```
        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
            //通过动态代理实现了几个通用方法, 比如 equals. toString. hashCode
            if ("equals".equals(method.getName())) {
                try {
                    Object
                        otherHandler =
                            args.length > 0 && args[0] != null ? Proxy.getInvocationHandler(args[0]) : null;
                    return equals(otherHandler);
                } catch (IllegalArgumentException e) {
                    return false;
                }
            } else if ("hashCode".equals(method.getName())) {
                return hashCode();
            } else if ("toString".equals(method.getName())) {
                return toString();
            }
            //找到具体的 method 的 Handler, 然后调用 invoke 方法。这样就又进入了SynchronousMethodHandler对象的 invoke 方法。
            return dispatch.get(method).invoke(args);
        }
    }
```

SynchronousMethodHandler 的 invoke 方法主要是应用 encoder, decoder 以及 retry 等配置, 并且自身对于调用结果有一定的处理逻辑。我们最关心的请求实现, 实际上是在组装 SynchronousMethodHandler 的 client 参数上, 即前面提到的, 如果当前路径里面有 Ribbon, 就是 LoadBalancerFeignClient, 如果没有, 根据配置生成 ApacheHttpClient 或者 OKHttpClient。在 Ribbon 里面, 实现了 Eureka 服务发现以及进行请求等动作。当然 Ribbon 里面还带了负载均衡逻辑。

```
        public Object invoke(Object[] argv) throws Throwable {
            RequestTemplate template = buildTemplateFromArgs.create(argv);
            Retryer retryer = this.retryer.clone();
            while (true) {
                try {
                    return executeAndDecode(template);
                } catch (RetryableException e) {
                    retryer.continueOrPropagate(e);
                    if (logLevel != Logger.Level.NONE) {
                        logger.logRetry(metadata.configKey(), logLevel);
                    }
                    continue;
                }
            }
        }
    }
```

```
        Object executeAndDecode(RequestTemplate template) throws Throwable {
            Request request = targetRequest(template);

            if (logLevel != Logger.Level.NONE) {
                logger.logRequest(metadata.configKey(), logLevel, request);
            }

            Response response;
            long start = System.nanoTime();
            try {
                //通过 client 获得请求的返回值
                response = client.execute(request, options);
                // ensure the request is set. TODO: remove in Feign 10
                response.toBuilder().request(request).build();
            } catch (IOException e) {
                if (logLevel != Logger.Level.NONE) {
                    logger.logIOException(metadata.configKey(), logLevel, e, elapsedTime(start));
                }
                throw errorExecuting(request, e);
            }
            long elapsedTime = TimeUnit.NANOSECONDS.toMillis(System.nanoTime() - start);

            boolean shouldClose = true;
            try {
                if (logLevel != Logger.Level.NONE) {
                    response =
                        logger.logAndRebufferResponse(metadata.configKey(), logLevel, response, elapsedTime);
                    // ensure the request is set. TODO: remove in Feign 10
                }
            }
```

```
        response.toBuilder().request(request).build();
    }
    if (Response.class == metadata.returnType()) {
        if (response.body() == null) {
            return response;
        }
        if (response.body().length() == null ||
            response.body().length() > MAX_RESPONSE_BUFFER_SIZE) {
            shouldClose = false;
            return response;
        }
        // Ensure the response body is disconnected
        byte[] bodyData = Util.toByteArray(response.body().asInputStream());
        return response.toBuilder().body(bodyData).build();
    }
    if (response.status() >= 200 && response.status() < 300) {
        if (void.class == metadata.returnType()) {
            return null;
        } else {
            return decode(response);
        }
    } else if (decode404 && response.status() == 404 && void.class != metadata.returnType()) {
        return decode(response);
    } else {
        throw errorDecoder.decode(metadata.configKey(), response);
    }
}
} catch (IOException e) {
    if (logLevel != Logger.Level.NONE) {
        logger.logIOException(metadata.configKey(), logLevel, e, elapsedTime);
    }
    throw errorReading(request, response, e);
} finally {
    if (shouldClose) {
        ensureClosed(response.body());
    }
}
```

## 总结

回到我们在开始想了解的两个问题。

### 请求是怎么转到 Feign 的？

分为两部分，第一是为接口定义的每个接口都生成一个实现方法，结果就是 SynchronousMethodHandler 对象。第二是为该服务接口生成了动态代理。动态代理的实现是 ReflectiveFeign.FeignInvocationHanlder，代理被调用的时候，会根据当前调用的方法，转到对应的 SynchronousMethodHandler。

### Feign 是怎么工作的？

当对接口的实例进行请求时（Autowire 的对象是某个ReflectiveFeign.FeignInvocationHanlder 的实例），根据方法名进入了某个 SynchronousMethodHandler 对象的 invoke 方法。

SynchronousMethodHandler 其实也并不处理具体的 HTTP 请求，它关心的更多的是请求结果的处理。HTTP 请求的过程，包括服务发现，都交给了当前 context 注册中的 Client 实现类，比如 LoadBalancerFeignClient。Retry 的逻辑实际上已经提出来了，但是 fallback 并没有在上面体现，因为我们上面分析动态代理的过程中，用的是 Feign.Builder，而如果有 fallback 的情况下，会使用 HystrixFeign.Builder，这是 Feign.Builder 的一个子类。它在创建动态代理的时候，主要改了一个东西，就是 invocationFactory 从默认的 InvocationHandlerFactory.Default 变成了一个内部匿名工厂，这个工厂的 create 方法返回的不是 ReflectiveFeign.FeignInvocationHandler，而是 HystrixInvocationHandler。所以动态代理类换掉了，invoke 的逻辑就变了。在新的逻辑里，没有简单的将方法转到对应的 SynchronousMethodHandler 上面，而是将 fallback 和 SynchronousMethodHandler一起封装成了 HystrixMethod，并且执行该对象。

本文转载自：<https://xli1224.github.io/2017/09/14/feign--anaylsis/>