## Spring Cloud Feign

Spring Cloud Feign是一套基于Netflix Feign实现的声明式服务调用客户端。它使得编写Web服务客户端变得更加简单。我们只需要通过创建接口并用注解来配置它既可完成对Web服务接口的绑定。它具备可插拔的注解支持，包括Feign注解、JAX-RS注解。它也支持可插拔的编码器和解码器。Spring Cloud Feign还扩展了对Spring MVC注解的支持，同时还整合了Ribbon和Eureka来提供均衡负载的HTTP客户端实现。

接下来的示例将分为三个模块：

* 1.服务接口定义模块：通过Spring MVC注解定义抽象的interface服务接口
* 服务接口实现模块：实现服务接口定义模块的interface，该模块作为服务提供者注册到eureka
* 服务接口消费模块：服务接口定义模块的客户端实现，该模块通过注册到eureka来消费服务接口

## 使用示例

1.服务接口的定义

* 创建一个Spring Boot项目：eureka-feign-api，pom.xml的主要内容如下：

<parent>

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

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

<version>1.5.6.RELEASE</version>

<relativePath/>

</parent>

<dependencies>

<dependency>

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

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

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.SR2</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

* 使用Spring MVC注解来定义服务接口：

public interface HelloService {

@GetMapping("/hello")

String hello(@RequestParam(value = "name") String name);

}

* 完成了上述构建之后，我们使用mvn install将该模块构建到本地的Maven仓库中。

2.服务接口的实现

* 创建一个Spring Boot项目：eureka-feign-client，pom.xml的主要内容如下：

<parent>

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

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

<version>1.5.6.RELEASE</version>

<relativePath/>

</parent>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>com.didispace</groupId>

<artifactId>eureka-feign-api</artifactId>

<version>1.0.0</version>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.SR2</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

该模块需要依赖上面定义的eureka-feign-api，将使用上述定义的HelloService接口来实现对应的REST服务。同时依赖Eureka是为了将该服务注册到Eureka上供服务消费者发现。

* 创建应用主类。使用[@EnableDiscoveryClient](https://github.com/EnableDiscoveryClient" \o "@EnableDiscoveryClient)注解开启服务注册与发现，并实现HelloService接口的REST服务：

@EnableDiscoveryClient

@SpringBootApplication

public class Application {

@RestController

class HelloController implements HelloService {

@Override

public String hello(String name) {

return "hello " + name;

}

}

public static void main(String[] args) {

new SpringApplicationBuilder(Application.class).web(true).run(args);

}

}

* 编辑application.properties配置内容：

spring.application.name=eureka-feign-client

server.port=2101

eureka.client.serviceUrl.defaultZone=http://eureka.didispace.com/eureka/

配置了服务提供者的名称eureka-feign-client，服务提供者的端口号2101，并将该服务注册到我的公益Eureka注册中心上。启动该项目，我们可以通过访问：<http://eureka.didispace.com/> ，在该页面中找到它。

3.服务接口的消费

* 创建一个Spring Boot项目：eureka-feign-consumer，pom.xml的主要内容如下：

<parent>

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

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

<version>1.5.6.RELEASE</version>

<relativePath/>

</parent>

<dependencies>

<dependency>

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

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

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-feign</artifactId>

</dependency>

<dependency>

<groupId>com.didispace</groupId>

<artifactId>eureka-feign-api</artifactId>

<version>1.0.0</version>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>Dalston.SR2</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

该模块较服务提供者的依赖增加了Feign的依赖，因为这里将使用Feign来绑定服务接口的客户端。下面我们将使用Feign的继承特性来轻松的构建Feign客户端。

* 创建应用主类。使用[@EnableDiscoveryClient](https://github.com/EnableDiscoveryClient" \o "@EnableDiscoveryClient)注解开启服务注册与发现，并通过[@FeignClient](https://github.com/FeignClient" \o "@FeignClient)注解来声明服务绑定客户端：

@EnableFeignClients

@EnableDiscoveryClient

@SpringBootApplication

public class Application {

@FeignClient("eureka-feign-client")

interface HelloServiceClient extends HelloService {

}

@RestController

class TestController {

@Autowired

private HelloServiceClient helloServiceClient;

@GetMapping("/test")

public String test(String name) {

return helloServiceClient.hello(name);

}

}

public static void main(String[] args) {

new SpringApplicationBuilder(Application.class).web(true).run(args);

}

}

从上述代码中我们可以看到，利用Feign的继承特性，[@FeignClient](https://github.com/FeignClient" \o "@FeignClient)注解只需要通过声明一个接口来继承在API模块中定义的公共interface就能产生服务接口的Feign客户端了。而[@FeignClient](https://github.com/FeignClient" \o "@FeignClient)中的值需要填写该服务的具体服务名（服务提供者的spring.application.name配置值）。

* 编辑服务消费者的application.properties配置内容，将服务消费者注册到eureka上来消费服务：

spring.application.name=eureka-feign-consumer

server.port=2102

eureka.client.serviceUrl.defaultZone=http://eureka.didispace.com/eureka/

* 启动eureka-feign-consumer之后，我们可以通过访问：<http://localhost:2102/test> ，来实验eureka-feign-consumer对eureka-feign-client接口的调用。

## 原理

#### 一、开启注册

开启@EnableFeignClients注解到底给我们做了什么事情呢？

1、扫描EnableFeignClients注解上的配置信息，注册默认的配置类，这个配置类是对所有feignclient的都是生效的，即为全局的配置。

2、扫描带有@FeignClient注解的接口，并注册配置类（此时的配置类针对当前feignclient生效）和FeignClientFactoryBean，此bean实现了FactoryBean接口，我们知道spring有两种类型的bean对象，一种是普通的bean，另一种则是工厂bean（FactoryBean），它返回的其实是getObject方法返回的对象（更多关于FactoryBean的相关信息请查看spring官方文档）。getObject方法就是集成原生feign的核心方法，当spring注入feignclient接口时，getObject方法会被调用，得到接口的代理类。

备注：在FeignClient指定配置类时，切记不要被spring容器扫描到，不然会对全局生效。

在@EnableFeignClients标签中，import了一个FeignClientsRegistrar类，那么这个FeignClientsRegistrar#registerBeanDefinitions()在什么时候调用的呢？跟着Spring的源码走下去，看过源码的人都会直接看到AbstractApplicationContext#refresh()方法，整体整理一下代码：

@Override

public void refresh() throws BeansException, IllegalStateException {

synchronized (this.startupShutdownMonitor) {

// 扫描本项目里面的java文件，把bean对象封装成BeanDefinitiaon对象，然后调用DefaultListableBeanFactory#registerBeanDefinition()方法把beanName放到DefaultListableBeanFactory 的 List<String> beanDefinitionNames 中去

ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();

// Prepare the bean factory for use in this context.

prepareBeanFactory(beanFactory);

try {

postProcessBeanFactory(beanFactory);

// 在这里调用到FeignClientsRegistrar对象的registerBeanDefinitions()方法

invokeBeanFactoryPostProcessors(beanFactory);

//从DefaultListableBeanFactory里面的beanDefinitionNames中找到所有实现了BeanPostProcessor接口的方法，如果有排序进行排序后放到list中

registerBeanPostProcessors(beanFactory);

//Spring的国际化

initMessageSource();

//

initApplicationEventMulticaster();

// Initialize other special beans in specific context subclasses.

onRefresh();

//

registerListeners();

// Spring的IOC、ID处理。Spring的AOP。事务都是在IOC完成之后调用了BeanPostProcessor#postProcessBeforeInitialization()和postProcessBeforeInitialization()方法，AOP(事务)就是在这里处理的

finishBeanFactoryInitialization(beanFactory);

// 执行完之后调用实现了所有LifecycleProcessor接口的类的onRefresh()方法，同时调用所有观察了ApplicationEvent接口的事件(观察者模式)

finishRefresh();

}

catch (BeansException ex) {

// 找到所有实现了DisposableBean接口的方法，调用了destroy()方法，这就是bean的销毁

destroyBeans();

// Reset 'active' flag.

cancelRefresh(ex);

throw ex;

}

finally {

resetCommonCaches();

}

}

}

根据上面整理的代码发现，FeignClientsRegistrar#registerBeanDefinitions()方法是在扫描完bean之后，只放了一个beanname的情况下， 并没有进行IOC注册的时候调用的，这就是Spring动态扩展Bean，实现BeanDefinitionRegistryPostProcessor接口的所有方法也会在这里调用下postProcessBeanDefinitionRegistry()方法。

下面回到正题，分析FeignClientsRegistrar#registerBeanDefinitions()方法:

@Override

public void registerBeanDefinitions(AnnotationMetadata metadata,

BeanDefinitionRegistry registry) {

registerDefaultConfiguration(metadata, registry);//扫描EnableFeignClients标签里配置的信息，注册到beanDefinitionNames中。

registerFeignClients(metadata, registry);

}

public void registerFeignClients(AnnotationMetadata metadata,

BeanDefinitionRegistry registry) {

AnnotationTypeFilter annotationTypeFilter = new AnnotationTypeFilter(

FeignClient.class);

//省略代码...根据EnableFeignClients配置的basePackages找到包下所有FeignClient注解的类，Spring的Commponet也是这么干的

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);

/\*\*

\* 关键地方：Feign子容器概念：

\* 在注入FeignAutoConfiguration类的时候，注入了一个FeignContext对象，这个就是Feign的子容器。

\* 这里面装了List<FeignClientSpecification>对象，FeignClientSpecification对象的实质就是在@feignClient上配置的name为key，value为configuration对象的值

\* 比如feignclient 这样配置的@FeignClient(url="https://api.weixin.qq.com",name="${usercenter.name}", configuration = UserCenterFeignConfiguration.class, primary= false)

\* 那么在FeignContext中就会出现一个FeignClientSpecification{name='sms-server', configuration=[class com.jfbank.sms.configuration.FeignConfiguration]}这样的数据。

\* 这个地方比较关键，主要是因为后期对feign客户端的编码解码会用到自定义的类

\*/

//这个方法就是在ioc容器中塞入一个FeignClientSpecification对象，从而构建FeignContext子容器。

registerClientConfiguration(registry, name,

attributes.get("configuration"));

//重点分析这个

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);//对FeignClientFactoryBean对象生成一个BeanDefinition对象

...读取配置

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 });

//注册到beanDefinitionNames中对象

BeanDefinitionReaderUtils.registerBeanDefinition(holder, registry);//

}

#### 二、加载引入

自动加载配置类FeignAutoConfiguration，FeignClientsConfiguration，FeignRibbonClientAutoConfiguration，这三个类为feign提供了所有的配置类，默认情况下所加载的类情况：

feign.Feign.Builder 当引入了Hytrix并开启参数feign.hystrix.enabled=true后，则会加载feign.hystrix.HystrixFeign.Builder，此时feign就具备降级熔断的功能了。

feign.Client 此实现类的加载分两种情况：

使用url方式：feign.Client.Default，使用java原生的方式（java.net包）发起http请求，也可以自己扩展。

使用name方式：LoadBalancerFeignClient，集成了ribbon，实现服务发现与负载均衡，但是真正发起http请求还是java原生的方式，此处是一扩展点，当我们引入ApacheHttpClient时，http客户端就会使用apache的httpClient；当我们引入OkHttpClient时，http客户端就会使用okhttp3.OkHttpClient。

feign.Retryer 默认Retryer.NEVER\_RETRY，不进行重试，这里也可以自己实现Retryer接口实现自己的重试策略，但是feign在集成了ribbon的情况下，最好保持默认不进行重试，因为ribbon也会有重试策略，如果feign也开启重试，容易产生混乱；其实在低版本中spring-cloud-feing重试默认并不是NEVER\_RETRY，可能spring-cloud-feing也意识到这样做的问题，所以在D版中改成NEVER\_RETRY了。

feign.Request.Options 默认设置连接超时时间是10，读超时时间是60s。这里也可以更改，分两种情况：

使用url方式：必须通过这个参数来设置，才生效

@Configuration

public class MyConfig {

@Bean

public Request.Options options(){

Request.Options o = new Options(1000, 1000);

return o;

}

}

然后在注解上@FeignClient指定：

@FeignClient(name="",url="",configuration= {MyConfig.class})

注意此类不能被spring容器扫描到，否则会对全局生效。你也可以通过注解@EnableFeignClients来全局指定：

@EnableFeignClients(defaultConfiguration=MyConfig.class)

使用name方式：此时已经集成了ribbon，可以使用以下配置来设置，如果你此时也配置了Options，以下配置会被覆盖

# 对所有的feignclient生效

ribbon.ReadTimeout=10000

ribbon.ConnectTimeout=2000

# 对指定的feignclien生效

[feignclientName].ribbon.ReadTimeout=10000

[feignclientName].ribbon.ConnectTimeout=2000

如果开启Hytrix，hytrix也有超时时间设置，但是hytrix是封装在feign基础之上的，上文已有分析。

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=10000

你也可以关闭hytrix的超时时间

hystrix.command.default.execution.timeout.enabled=false

feign.codec.Decoder 解码器，默认使用了HttpMessageConverters来实现

feign.codec.Encoder 编码器，默认使用了HttpMessageConverters来实现

feign.Contract 默认提供springmvc的注解解析，支持@RequestMapping，@RequestBody，@RequestParam，@PathVariable

     读过Dubbo源码的同学都知道，当在DubboNamespaceHandler中解析reference标签的时候，传入了一个ReferenceBean对象，把xml中配置的属性都塞到这个对象上，也是装到了beanDefinitionNames中，然后发现ReferenceBean类和FeignClientFactoryBean都实现了FactoryBean的接口，并且里面都有getObject()和getObjectType()方法。当接口调用到这个feign客户端的时候，会从IOC中读取这个FeignClientFactoryBean并且调用getObject方法。下面就是分析getObject方法：

@Override

public Object getObject() throws Exception {

FeignContext context = applicationContext.getBean(FeignContext.class);

//从上文中的子容器中获取编码器，解码器等自定义类，然后封装一个Feign.Builder类

Feign.Builder builder = feign(context);

if (!StringUtils.hasText(this.url)) {//当@FeignClient没有配置url的时候

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));//集成了ribbon客户端负载均衡,下一篇分析

}

//当@FeignClient配置了url的时候

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 lod 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));

}

首先看配置了url的，指定了url的feignclient解析，一直跟着代码跟到了Feign.Builder#target()方法：

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);

}

直接看ReflectiveFeign#newInstance()方法：

//ReflectiveFeign#newInstance()

public <T> T newInstance(Target<T> target) {

//动态代理的handler类目前穿进来的是ParseHandlersByName类，所以这里要看ParseHandlersByName#apply()直接看下一个方法

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)) {//默认方法会走到这里，比如toString()，hashCode()等方法

DefaultMethodHandler handler = new DefaultMethodHandler(method);

defaultMethodHandlers.add(handler);

methodToHandler.put(method, handler);

} else {//这里才是装配的调用类，上文分析到计息的handler是SynchronousMethodHandler#invoke()

methodToHandler.put(method, nameToHandler.get(Feign.configKey(target.type(), method)));

}

}

InvocationHandler handler = factory.create(target, methodToHandler);

T proxy = (T) Proxy.newProxyInstance(target.type().getClassLoader(), new Class<?>[]{target.type()}, handler);//jdk动态代理

for(DefaultMethodHandler defaultMethodHandler : defaultMethodHandlers) {

defaultMethodHandler.bindTo(proxy);

}

return proxy;

}

//ParseHandlersByName#apply类，构建动态代理的handler

public Map<String, MethodHandler> apply(Target key) {

List<MethodMetadata> metadata = contract.parseAndValidatateMetadata(key.type());

Map<String, MethodHandler> result = new LinkedHashMap<String, MethodHandler>();

for (MethodMetadata md : metadata) {

BuildTemplateByResolvingArgs buildTemplate;

if (!md.formParams().isEmpty() && md.template().bodyTemplate() == null) {

buildTemplate = new BuildFormEncodedTemplateFromArgs(md, encoder);//通过自定义的encoder去解析参数

} else if (md.bodyIndex() != null) {

buildTemplate = new BuildEncodedTemplateFromArgs(md, encoder);//通过自定义的encoder去解析参数

} else {

buildTemplate = new BuildTemplateByResolvingArgs(md);

}

//创建handler，再看Factory#create()方法，下一个方法

result.put(md.configKey(),

factory.create(key, md, buildTemplate, options, decoder, errorDecoder));

}

return result;

}

//Factory#create(),构建一个SynchronousMethodHandler去处理请求，调用invoke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);

}

//SynchronousMethodHandler#invoke()方法：实际调用的方法

//@Override

public Object invoke(Object[] argv) throws Throwable {

RequestTemplate template = buildTemplateFromArgs.create(argv);//构建requestTemplate对象

Retryer retryer = this.retryer.clone();

while (true) {

try {

return executeAndDecode(template);//下面不分析了，就是执行execute方法并且解码饭后返回值

} catch (RetryableException e) {

retryer.continueOrPropagate(e);

if (logLevel != Logger.Level.NONE) {

logger.logRetry(metadata.configKey(), logLevel);

}

continue;

}

}

}

#### Feign源码总结：

