# Learn Feign

# 基本介绍

Feign是一个用于Spring Boot应用程序的声明式REST客户端。

官方原话是 **Feign makes writing java http clients easier**。

Feign 基于每个请求是同步的。因此，Feign 会阻塞，直到它收到响应。

# Chapter 1. Netflix Feign 和 Open Feign

最初，Feign 是由 Netflix 创建并发布的，作为其 Netflix OSS 项目的一部分。后来 Netflix 决定在内部停止使用 Feign，并停止开发。由于这一决定，Netflix 在一个名为 OpenFeign 的新项目下将 Feign 完全转移到了开源社区。Feign 项目被贡献给了开源组织，于是才有了我们今天使用的 OpenFeign 组件。

Spring Cloud Netflix 将 Netflix OSS 产品整合到了 Spring Cloud 生态系统中，包括 **Feign**、**Eureka**、**Ribbon** 和许多其他工具和实用程序。

# Chapter 2. Spring Cloud OpenFeign

Spring-Cloud-OpenFeign 是在 整合了 `OpenFeign` 的基础上加入了 `SpringMVC` 注解支持。

Spring Boot 1.4.7或更早版本 用的是 spring-cloud-starter-feign，之后用的是spring-cloud-starter-openfeign。

# Feign 的组成

# Chapter 3. Feign 的组成

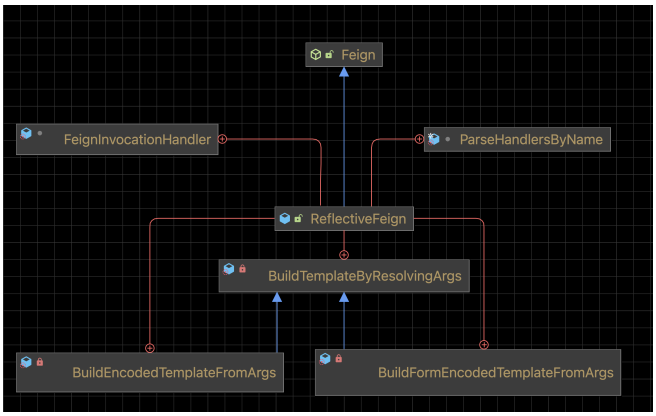
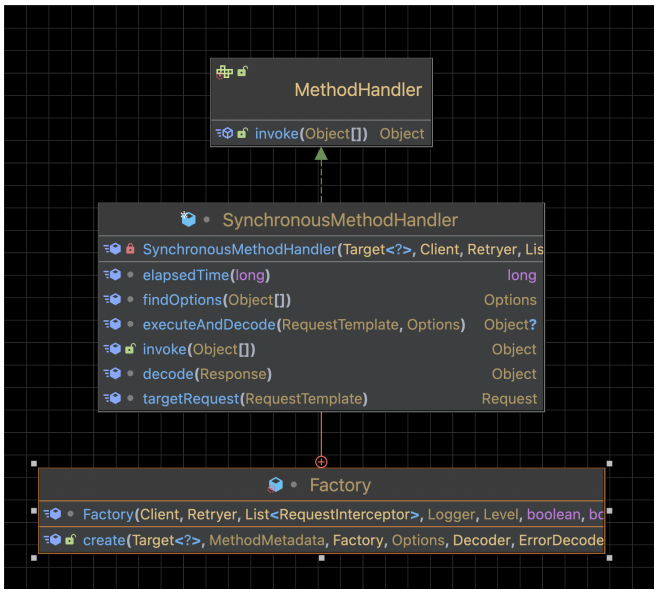
| 接口 | 作用 | 默认值 |
| --- | --- | --- |
| Feign.Builder | Feign的入口 | |
| Client | Feign底层用什么去请求 | Feign.Client.Default<br>(和Ribbon配合时LoadBalancerFeignClient) |
| Contract | 解析 HTTP 请求 | feign.Contract.Default |
| Encoder | 编码器，用于将对象转换成HTTP请求消息体 | feign.codec.Encoder.Default |
| Decoder | 解码器，将响应消息体转换成对象 | feign.codec.Decoder.Default |
| ErrorDecoder | 错误解码器，处理 2XX、404<br>之外的响应信息 | feign.codec.ErrorDecoder.Default |
| Logger | 日志管理器 | feign.Logger.NoOpLogger |
| Retryer | 重试器，默认重试 5<br>次（以1.5的整数指数幂倍厘秒为间隔） | feign.Retryer.Default#Default |
| RequestInterceptor | 请求拦截器，用于为每个请求添加通用逻辑 | 无 |
| hystrix | 服务熔断组件 | |
| ribbon | 负载均衡组件 | |

# Feign 的核心流程

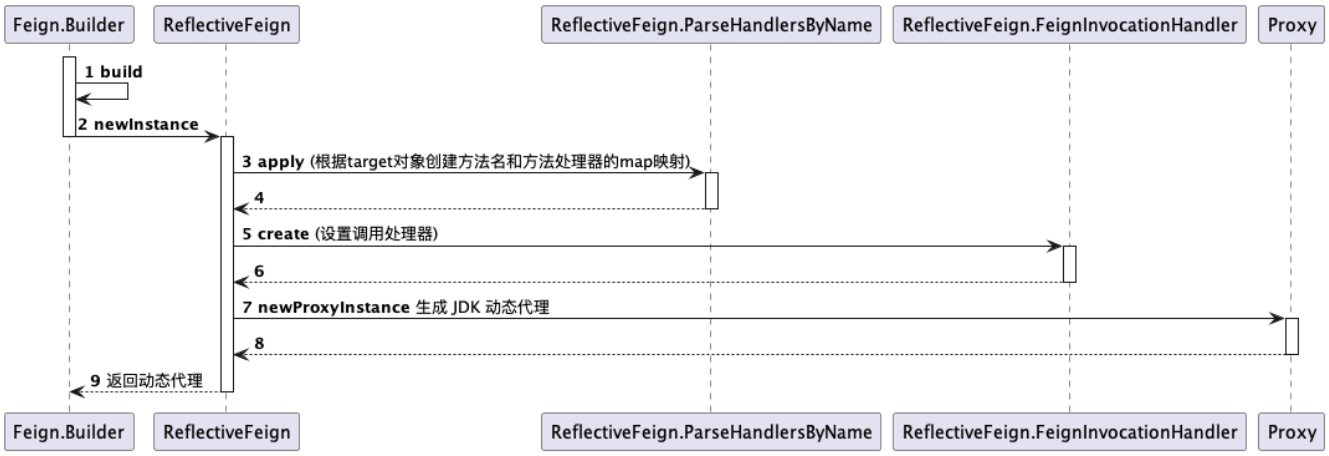
基本流程：

- 使用 JDK 动态代理为每一个 `@FeignClient` 标记的接口生成代理类
- 生成代理类时为每个非默认方法添加了一个代理方法
- 最终执行代理类中的代理方法

核心类：

| ReflectiveFeign | SynchronousMethodHandler |
| --- | --- |
|  |  |

# Chapter 4. 时序图



# Chapter 5. Feign 的入口方法

通过Feign.builder()生成Feign.Builder的构造者对象，然后设置相关的参数，再调用target方法构造出动态代理。

Feign.builder()

```
public abstract class Feign {
  public static Builder builder() {
    return new Builder();
  }
    public static class Builder {
      //...
  }
}
```

# Chapter 6. 构建 ReflectFeign

target 方法内部先调用 build 方法新建一个 ReflectFeign 对象，然后调用 ReflectFeign 的 newInstance 方法创建动态代理。

feign.Feign

```
public <T> T target(Target<T> target) {
  // spring-cloud-openfeign 注入最终执行方法，最后注入到 Spring
容器中的是接口的代理对象。
  return build().newInstance(target);
}

public Feign build() {
  // 调用处理器工厂，设置属性
  SynchronousMethodHandler.Factory synchronousMethodHandlerFactory =
      new SynchronousMethodHandler.Factory(client, retryer, requestInterceptors,
logger,
          logLevel, decode404, closeAfterDecode, propagationPolicy);
  // 解析处理器handlersByName将所有参数进行封装，只有一个 apply 方法
  ParseHandlersByName handlersByName =
      new ParseHandlersByName(contract, options, encoder, decoder, queryMapEncoder,
          errorDecoder, synchronousMethodHandlerFactory);
            return new ReflectiveFeign(
                    handlersByName,
                    // InvocationHandlerFactory 控制反射方法的调度
                    invocationHandlerFactory, ①
                    // QueryMapEncoder 按照 name 和 value 转换 object 到 map
                    queryMapEncoder); ②
}
```

① Builder 的属性，控制反射方法的调度，默认值是 InvocationHandlerFactory.Default()

② Builder 的属性，接口参数注解 @QueryMap 时，参数的编码器，默认值是 QueryMapEncoder.Default()

# Chapter 7. 创建动态代理对象

ReflectiveFeign 有五个内部类：

| Inner Class | Description |
| --- | --- |
| FeignInvocationHandler | 实现了 java.lang.reflect.InvocationHandler，定义 JDK 动态代理执行的方法 |
| ParseHandlersByName | 解析处理程序 |
| BuildTemplateByResolvingArgs | 实现了 RequestTemplate.Factory ，HTTP 请求生成器工厂。 |
| BuildEncodedTemplateFromArgs | 继承 BuildTemplateByResolvingArgs，重载 resolve 方法，解析 body 请求。 |
| BuildFormEncodedTemplateFromArgs | 继承 BuildTemplateByResolvingArgs，重载 resolve 方法，解析 form 表单请求。 |

ReflectiveFeign.newInstance 方法创建接口动态代理对象

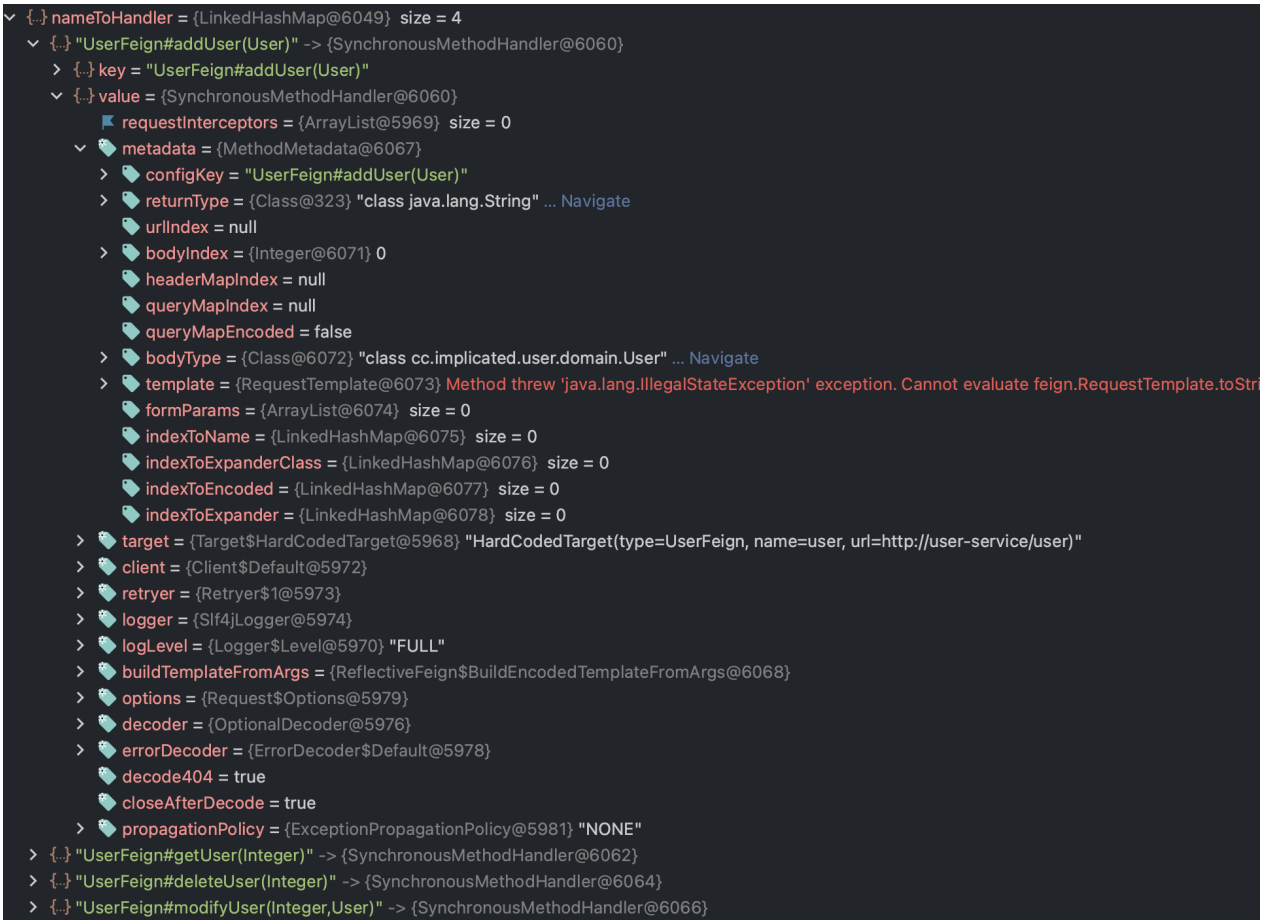
feign.ReflectiveFeign

```
// 构造器
ReflectiveFeign(ParseHandlersByName targetToHandlersByName,
                InvocationHandlerFactory factory,
                QueryMapEncoder queryMapEncoder) {
    this.targetToHandlersByName = targetToHandlersByName;
    this.factory = factory;
    this.queryMapEncoder = queryMapEncoder;
}

// 构造器
@Override
public <T> T newInstance(Target<T> target) {
    // 1.解析方法元数据
    // targetToHandlersByName 是构造器传入的 ParseHandlersByName 对象
    // 根据 target 对象创建方法名和方法处理器的 map 映射
    Map<String, MethodHandler> nameToHandler = targetToHandlersByName.apply(target);
①
    // 方法处理器的 map 映射
    Map<Method, MethodHandler> methodToHandler = new LinkedHashMap<Method,
MethodHandler>();
    // 默认方法处理器的 map 映射
    List<DefaultMethodHandler> defaultMethodHandlers = new LinkedList
<DefaultMethodHandler>();

    // target.type() is @FeignClient 标识的class
    // 遍历接口所有方法，构建 Method  ->  MethodHandler 的映射
    for (Method method : target.type().getMethods()) {
        if (method.getDeclaringClass() == Object.class) {
            continue;
        } else if (Util.isDefault(method)) {
            // 默认方法，执行原方法
            DefaultMethodHandler handler = new DefaultMethodHandler(method);
            defaultMethodHandlers.add(handler);
```

```
            methodToHandler.put(method, handler);
        } else {
            // 一般方法，执行代理类方法
            methodToHandler.put(method, nameToHandler.get(Feign.configKey(target.type
(), method)));
        }
    }
    // 2.设置调用处理器
    // 创建动态代理，factory 是 InvocationHandlerFactory.Default，创建出来的是
    // feign.ReflectiveFeign.FeignInvocationHandler.dispatch 持有 methodToHandler 引用
    //
ReflectiveFeign.FeignInvocationHandler，也就是说后续对方法的调用都会进入到该对象的
invoke 方法
    InvocationHandler handler = factory.create(target, methodToHandler); ②

    // 3.生成 JDK 动态代理
    T proxy = (T) Proxy.newProxyInstance(target.type().getClassLoader(), ③
            new Class<?>[]{target.type()}, handler);

    // 绑定默认方法到代理上
    for (DefaultMethodHandler defaultMethodHandler : defaultMethodHandlers) {
        defaultMethodHandler.bindTo(proxy);
    }
    return proxy;
}
```

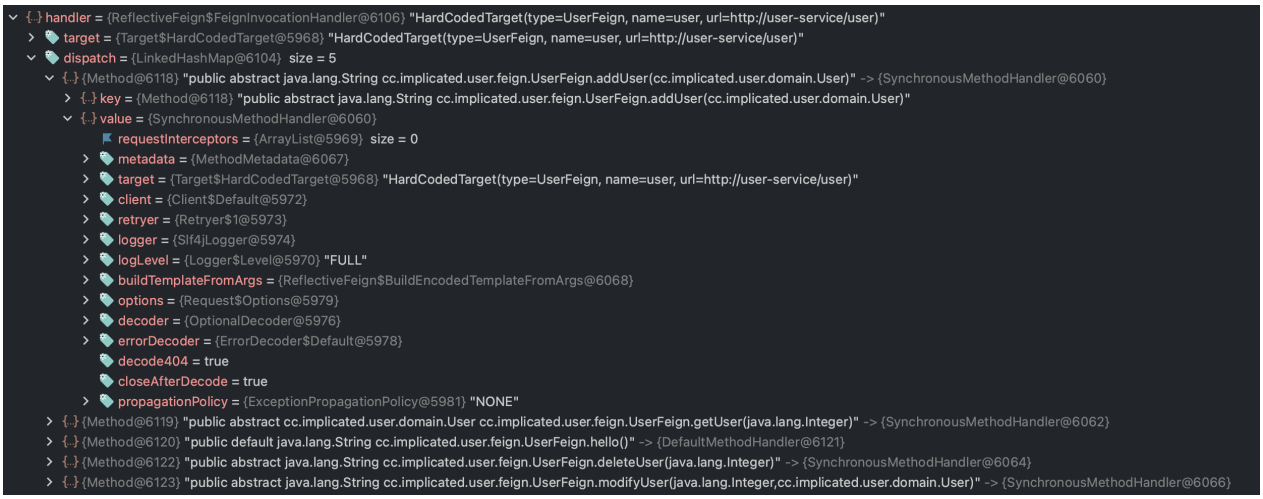
① 根据指定的 contract 解析 target，根据 MethodMetadata 生成 MethodHandler 。

▼ 方法元数据



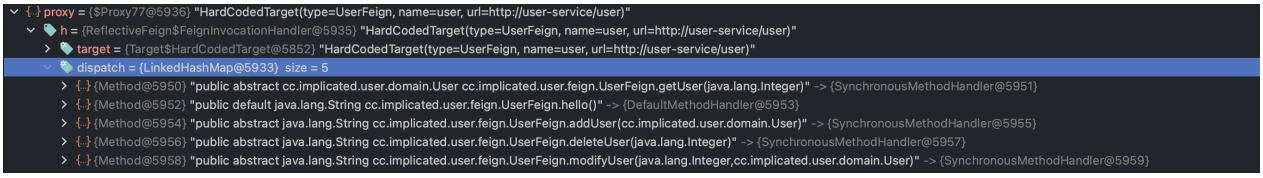

② 设置调用处理器，最终执行 HTTP 请求的地方。

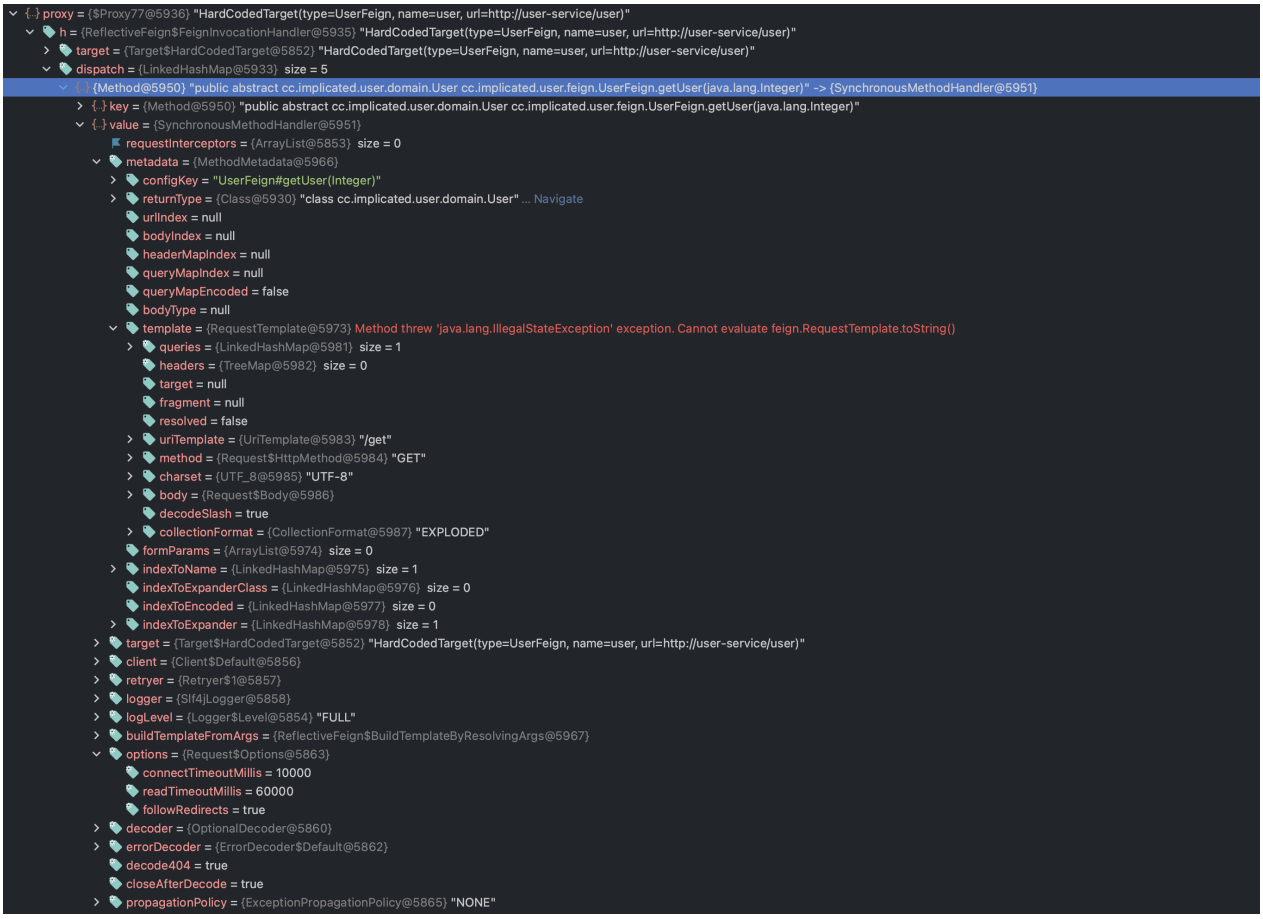
▼ 调用处理器




③ 生成 JDK 动态代理。

▼ 动态代理

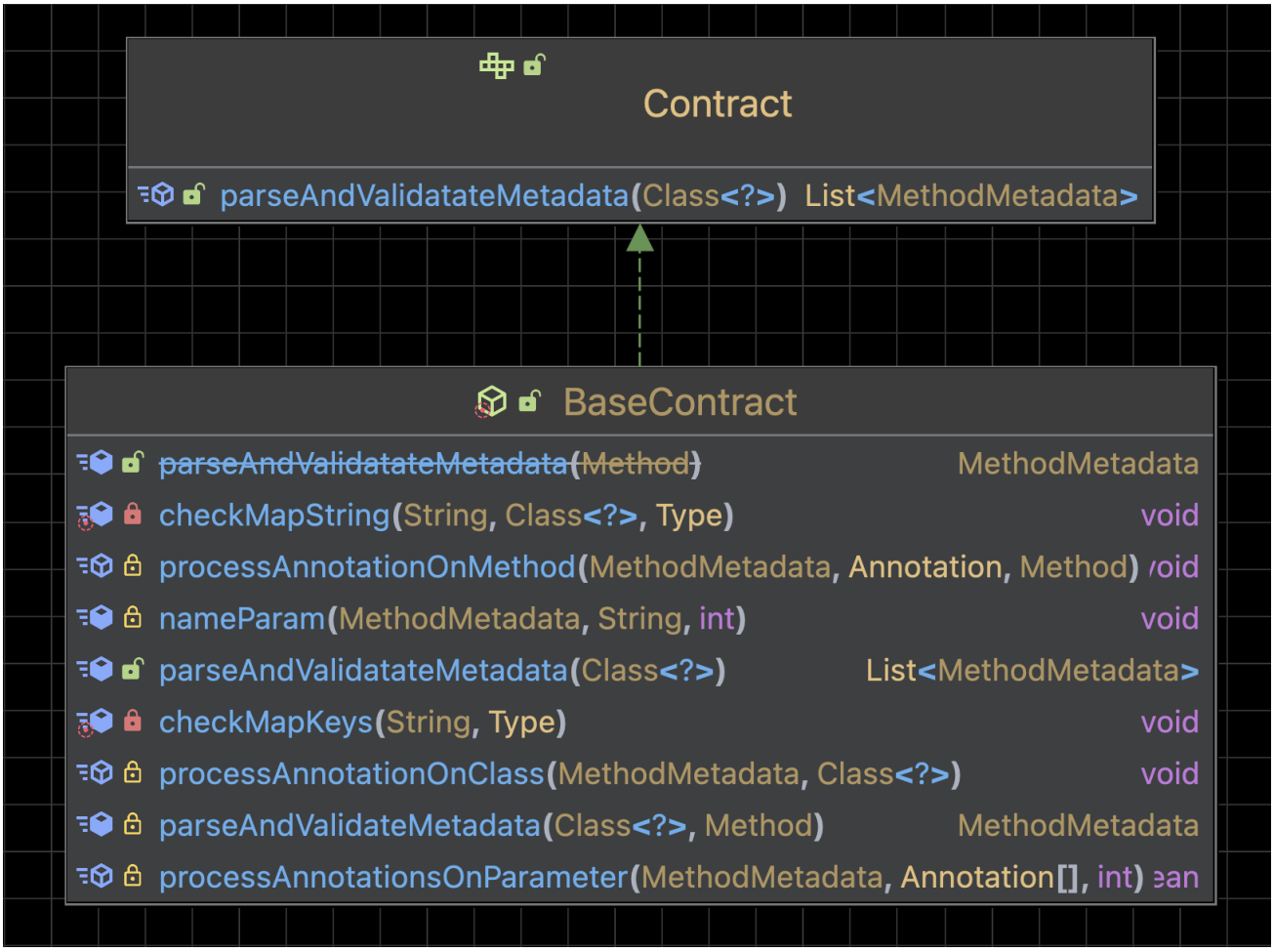



proxy = {$Proxy77@5936} "HardCodedTarget(type=UserFeign, name=user, url=http://user-service/user)"
h = {ReflectiveFeign$FeignInvocationHandler@5935} "HardCodedTarget(type=UserFeign, name=user, url=http://user-service/user)"
target = {Target$HardCodedTarget@5852} "HardCodedTarget(type=UserFeign, name=user, url=http://user-service/user)"
dispatch = {LinkedHashMap@5933} size = 5
{Method@5950} "public abstract cc.implicated.user.domain.User cc.implicated.user.feign.UserFeign.getUser(java.lang.Integer)" -> {SynchronousMethodHandler@5951}
key = {Method@5950} "public abstract cc.implicated.user.domain.User cc.implicated.user.feign.UserFeign.getUser(java.lang.Integer)"
value = {SynchronousMethodHandler@5951}
requestInterceptors = {ArrayList@5853} size = 0
metadata = {MethodMetadata@5966}
configKey = "UserFeign#getUser(Integer)"
returnType = {Class@5930} "class cc.implicated.user.domain.User" ... Navigate
urlIndex = null
bodyIndex = null
headerMapIndex = null
queryMapIndex = null
queryMapEncoded = false
bodyType = null
template = {RequestTemplate@5973} Method threw 'java.lang.IllegalStateException' exception. Cannot evaluate feign.RequestTemplate.toString()
queries = {LinkedHashMap@5981} size = 1
headers = {TreeMap@5982} size = 0
target = null
fragment = null
resolved = false
uriTemplate = {UriTemplate@5983} "/get"
method = {Request$HttpMethod@5984} "GET"
charset = {UTF_8@5985} "UTF-8"
body = {Request$Body@5986}
decodeSlash = true
collectionFormat = {CollectionFormat@5987} "EXPLODED"
formParams = {ArrayList@5974} size = 0
indexToName = {LinkedHashMap@5975} size = 1
indexToExpanderClass = {LinkedHashMap@5976} size = 0
indexToEncoded = {LinkedHashMap@5977} size = 0
indexToExpander = {LinkedHashMap@5978} size = 1
target = {Target$HardCodedTarget@5852} "HardCodedTarget(type=UserFeign, name=user, url=http://user-service/user)"
client = {Client$Default@5856}
retryer = {Retryer$1@5857}
logger = {Slf4jLogger@5858}
logLevel = {Logger$Level@5854} "FULL"
buildTemplateFromArgs = {ReflectiveFeign$BuildTemplateByResolvingArgs@5967}
options = {Request$Options@5863}
connectTimeoutMillis = 10000
readTimeoutMillis = 60000
followRedirects = true
decoder = {OptionalDecoder@5860}
errorDecoder = {ErrorDecoder$Default@5862}
decode404 = true
closeAfterDecode = true
propagationPolicy = {ExceptionPropagationPolicy@5865} "NONE"

# Chapter 8. 解析方法元数据

Spring-cloud 继承 Contract.BaseContract，实现处理参数的方法

▼ 查看 BaseContract 类图




feign.ReflectiveFeign.ParseHandlersByName#apply

```
public Map<String, MethodHandler> apply(Target key) {
    // 待处理的方法元数据
    List<MethodMetadata> metadata = contract.parseAndValidatateMetadata(key.type());
①
    // 保存生成的方法处理器
    Map<String, MethodHandler> result = new LinkedHashMap<String, MethodHandler>();
    // 处理方法元数据
    for (MethodMetadata md : metadata) {
        BuildTemplateByResolvingArgs buildTemplate;
        // form 表单
        if (!md.formParams().isEmpty() && md.template().bodyTemplate() == null) {
            buildTemplate = new BuildFormEncodedTemplateFromArgs(md, encoder,
queryMapEncoder);
        }
        // body
        else if (md.bodyIndex() != null) {
            buildTemplate = new BuildEncodedTemplateFromArgs(md, encoder,
queryMapEncoder);
        }
```

```
        // 其它请求
        else {
            buildTemplate = new BuildTemplateByResolvingArgs(md, queryMapEncoder);
        }
        // 创建方法处理器
        result.put(md.configKey(),
                factory.create(key, md, buildTemplate, options, decoder, errorDecoder
)); ②
    }
    return result;
}
```

① 获取方法元数据

② 为每个方法创建自己的方法处理器

## 8.1. 获取方法元数据

解析每个非默认方法获取方法元数据。

feign.Contract.BaseContract#parseAndValidatateMetadata

```
@Override
public List<MethodMetadata> parseAndValidatateMetadata(Class<?> targetType) {
    //...

    // 解析每个方法
    Map<String, MethodMetadata> result = new LinkedHashMap<String, MethodMetadata>();
    for (Method method : targetType.getMethods()) {
        // 跳过默认方法
        if (method.getDeclaringClass() == Object.class ||
                (method.getModifiers() & Modifier.STATIC) != 0 ||
                Util.isDefault(method)) {
            continue;
        }
        // 解析方法元数据，进入 SpringMvcContract
        MethodMetadata metadata = parseAndValidateMetadata(targetType, method);
        checkState(!result.containsKey(metadata.configKey()), "Overrides unsupported:
%s",
                metadata.configKey());
        result.put(metadata.configKey(), metadata);
    }
    return new ArrayList<>(result.values());
}
```

解析每个具体方法。

feign.Contract.BaseContract#parseAndValidateMetadata

```
protected MethodMetadata parseAndValidateMetadata(Class<?> targetType, Method method)
{
    MethodMetadata data = new MethodMetadata();
```

```
    data.returnType(Types.resolve(targetType, targetType, method.getGenericReturnType
()));
    // 方法唯一key
    data.configKey(Feign.configKey(targetType, method));

    // 解析class上注解
    if (targetType.getInterfaces().length == 1) {
        processAnnotationOnClass(data, targetType.getInterfaces()[0]);
    }
    processAnnotationOnClass(data, targetType);


    // 解析method上注解
    for (Annotation methodAnnotation : method.getAnnotations()) {
        processAnnotationOnMethod(data, methodAnnotation, method);
    }
    Class<?>[] parameterTypes = method.getParameterTypes();
    Type[] genericParameterTypes = method.getGenericParameterTypes();

    // 解析参数
    Annotation[][] parameterAnnotations = method.getParameterAnnotations();
    int count = parameterAnnotations.length;
    for (int i = 0; i < count; i++) {
        boolean isHttpAnnotation = false;
        if (parameterAnnotations[i] != null) {
            isHttpAnnotation = processAnnotationsOnParameter(data,
parameterAnnotations[i], i);
        }
        if (parameterTypes[i] == URI.class) {
            data.urlIndex(i);
        } else if (!isHttpAnnotation && parameterTypes[i] != Request.Options.class) {
            data.bodyIndex(i);
            data.bodyType(Types.resolve(targetType, targetType, genericParameterTypes
[i]));
        }
    }
    return data;
}
```

## 8.2. 创建方法处理器

feign.SynchronousMethodHandler.Factory#create

```
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, closeAfterDecode, propagationPolicy);
}
```

# Chapter 9. 设置调用处理器

通过工厂方法生成调用处理器。

feign.InvocationHandlerFactory.Default

```
static final class Default implements InvocationHandlerFactory {

    @Override
    public InvocationHandler create(Target target, Map<Method, MethodHandler>
dispatch) {
        return new ReflectiveFeign.FeignInvocationHandler(target, dispatch);
    }
}
```

## 9.1. 不同方法的路由

MethodHandler 有两个实现，DefaultMethodHandler 和 SynchronousMethodHandler。 默认方法走 DefaultMethodHandler ，其它方法走 SynchronousMethodHandler。

feign.ReflectiveFeign.FeignInvocationHandler#invoke

```
@Override
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
    // 自定义equals、hashCode和toString方法的处理
    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();
    }

    // dispatch 就是 newInstance 方法中的 methodToHandler
    return dispatch.get(method).invoke(args);
}
```

## 9.2. DefaultMethodHandler

执行代理接口自己的方法。

feign.DefaultMethodHandler#invoke

```
@Override
```

```
public Object invoke(Object[] argv) throws Throwable {
    if (handle == null) {
        throw new IllegalStateException(
                "Default method handler invoked before proxy has been bound.");
    }
    return handle.invokeWithArguments(argv);
}
```

## 9.3. SynchronousMethodHandler

定义了发起 HTTP 请求的方法

feign.SynchronousMethodHandler#invoke

```
@Override
public Object invoke(Object[] argv) throws Throwable {
    // RequestTemplate 定义了完整的HTTP请求信息
    RequestTemplate template = buildTemplateFromArgs.create(argv);
    // Options 定义了连接超时时间、请求超时时间、是否允许重定向
    Options options = findOptions(argv);
    // 重试设置
    Retryer retryer = this.retryer.clone();
    // 成功返回，失败抛异常
    while (true) {
        try {
            // 执行和解码
            return executeAndDecode(template, options);
        } catch (RetryableException e) {
            try {
                // 判断是否继续重试
                retryer.continueOrPropagate(e);
            } catch (RetryableException th) {
                Throwable cause = th.getCause();
                if (propagationPolicy == UNWRAP && cause != null) {
                    throw cause;
                } else {
                    throw th;
                }
            }
            if (logLevel != Logger.Level.NONE) {
                logger.logRetry(metadata.configKey(), logLevel);
            }
            // 重试
            continue;
        }
    }
}
```

### 发起请求和解析响应

feign.SynchronousMethodHandler#executeAndDecode

```
Object executeAndDecode(RequestTemplate template, Options options) throws Throwable {
    // RequestTemplate 转换为 Request
    Request request = targetRequest(template);

    // 打印请求参数
    if (logLevel != Logger.Level.NONE) {
        logger.logRequest(metadata.configKey(), logLevel, request);
    }

    // 打印接口响应时间
    Response response;
    long start = System.nanoTime();
    try {
        // 发起请求
        response = client.execute(request, options);
    } catch (IOException e) {
        if (logLevel != Logger.Level.NONE) {
            logger.logIOException(metadata.configKey(), logLevel, e, elapsedTime(
start));
        }
        // 抛出重试异常 RetryableException()
        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);
        }
        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
            // 读取body字节数组，返回response
            byte[] bodyData = Util.toByteArray(response.body().asInputStream());
            return response.toBuilder().body(bodyData).build();
        }
        // 处理 2XX
        if (response.status() >= 200 && response.status() < 300) {
```

```
            // 接口返回void
            if (void.class == metadata.returnType()) {
                return null;
            }
            // 解码response，直接调用decoder解码
            else {
                Object result = decode(response);
                shouldClose = closeAfterDecode;
                return result;
            }
        }
        // 处理 404
        else if (decode404 && response.status() == 404 && void.class != metadata
.returnType()) {
            Object result = decode(response);
            shouldClose = closeAfterDecode;
            return result;
        }
        // 其他返回码，使用errorDecoder解析，抛出异常
        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) {
            // 关流
            ensureClosed(response.body());
        }
    }
}
```

# Chapter 10. 发送HTTP请求

`feign.Client` 是一个接口，默认实现类是 `feign.Client.Default`， 使用 `java.net.HttpURLConnection` 发送 `HTTP` 请求。

feign还实现了：

- ApacheHttpClient
- OkHttpClient
- RibbonClient
- Java 11 Http2

## 10.1. 默认请求方式。

feign.Client.Default#execute

```
@Override
public Response execute(Request request, Options options) throws IOException {
    HttpURLConnection connection = convertAndSend(request, options); ①
    return convertResponse(connection, request); ②
}
```

① 发送请求

② 解析响应

### 发送请求

feign.Client.Default#convertAndSend

```
HttpURLConnection convertAndSend(Request request, Options options) throws IOException
{
    final URL url = new URL(request.url());
    // 获取 conniption
    final HttpURLConnection connection = this.getConnection(url);
    //...

    connection.setConnectTimeout(options.connectTimeoutMillis());
    connection.setReadTimeout(options.readTimeoutMillis());
    connection.setAllowUserInteraction(false);
    connection.setInstanceFollowRedirects(options.isFollowRedirects());
    connection.setRequestMethod(request.httpMethod().name());

    //..
}
```

### 解析响应

feign.Client.Default#convertResponse

```
Response convertResponse(HttpURLConnection connection, Request request) throws
IOException {
```

```
    int status = connection.getResponseCode();
    String reason = connection.getResponseMessage();

    if (status < 0) {
        throw new IOException(format("Invalid status(%s) executing %s %s", status,
                connection.getRequestMethod(), connection.getURL()));
    }

    Map<String, Collection<String>> headers = new LinkedHashMap<>();
    for (Map.Entry<String, List<String>> field : connection.getHeaderFields().
entrySet()) {
        // response message
        if (field.getKey() != null) {
            headers.put(field.getKey(), field.getValue());
        }
    }

    Integer length = connection.getContentLength();
    if (length == -1) {
        length = null;
    }
    InputStream stream;
    if (status >= 400) {
        stream = connection.getErrorStream();
    } else {
        stream = connection.getInputStream();
    }
    return Response.builder()
            .status(status)
            .reason(reason)
            .headers(headers)
            .request(request)
            .body(stream, length)
            .build();
}
```

## 10.2. Apache HttpClient

To Be Continued.

## 10.3. OkHttp

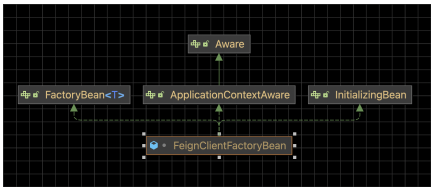
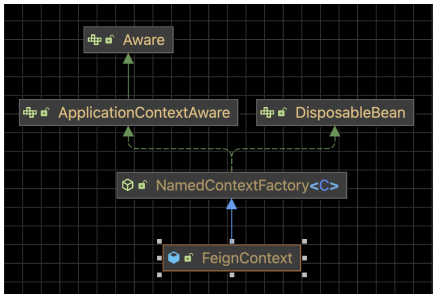
To Be Continued.

# Spring 整合 Feign 原理

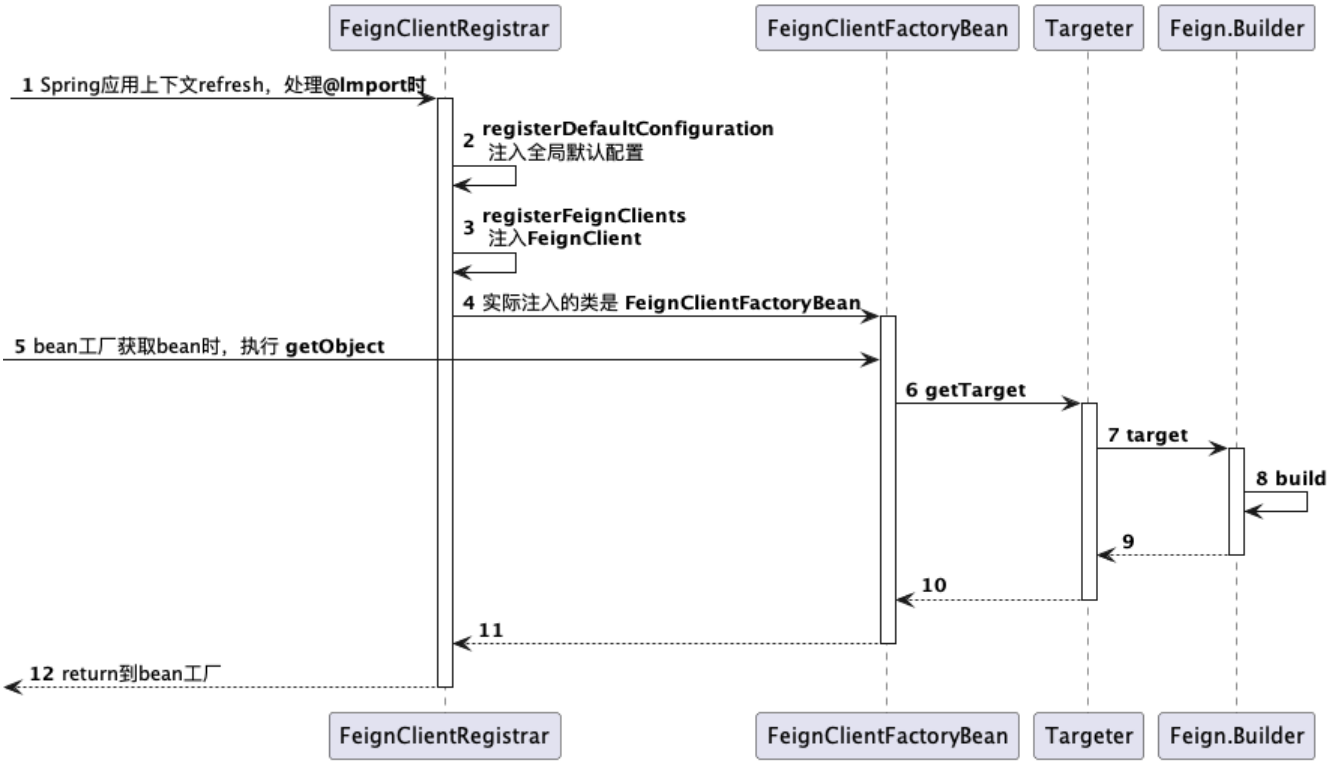
基本流程：

- 使用 FeignClientsRegistrar 扫包并
  - 注入 FeignClient 配置信息（实际是注入 FeignClientSpecification）。
  - 注入 FeignClient 客户端（实际是注入 FeignClientFactoryBean）。
- 使用 FeignClientSpecification 和 FeignClientFactoryBean 为每个 @FeignClient 标记的类创建子容器，每个容器相互隔离，都拥有完整的 FeignClient 配置信息。

核心类：

| FeignClientSpecification | FeignClientFactoryBean | FeignContext |
| --- | --- | --- |
|  |  |  |

# Chapter 11. 时序图



# Chapter 12. 入口

org.springframework.cloud.openfeign.EnableFeignClients

```
@Retention()
@Target(ElementType.TYPE)
@Documented
@Import(FeignClientsRegistrar.class) ①
public @interface EnableFeignClients {
   // ...
}
```

① 导入 FeignClient 注入器

# Chapter 13. FeignClient 注入器

org.springframework.cloud.openfeign.FeignClientsRegistrar

```
@Override
public void registerBeanDefinitions(AnnotationMetadata metadata,
BeanDefinitionRegistry registry) {
    registerDefaultConfiguration(metadata, registry); ①
    registerFeignClients(metadata, registry); ②
}
```

① 注入 `@EnableFeignClients` 的全局默认配置（defaultConfiguration）

② 注入 `@FeignClient`

## 13.1. 注入 @EnableFeignClients

org.springframework.cloud.openfeign.FeignClientsRegistrar#registerDefaultConfiguration

```
private void registerClientConfiguration(BeanDefinitionRegistry registry, Object name,
            Object configuration) {
    BeanDefinitionBuilder builder = BeanDefinitionBuilder
            .genericBeanDefinition(FeignClientSpecification.class);
    builder.addConstructorArgValue(name);
    builder.addConstructorArgValue(configuration);
    // 注入 FeignClientSpecification （包含 ） EnableFeignClients.defaultConfiguration
    registry.registerBeanDefinition(
            name + "." + FeignClientSpecification.class.getSimpleName(),
            builder.getBeanDefinition());
    }
```

## 13.2. 注入 @FeignClient

org.springframework.cloud.openfeign.FeignClientsRegistrar#registerFeignClients

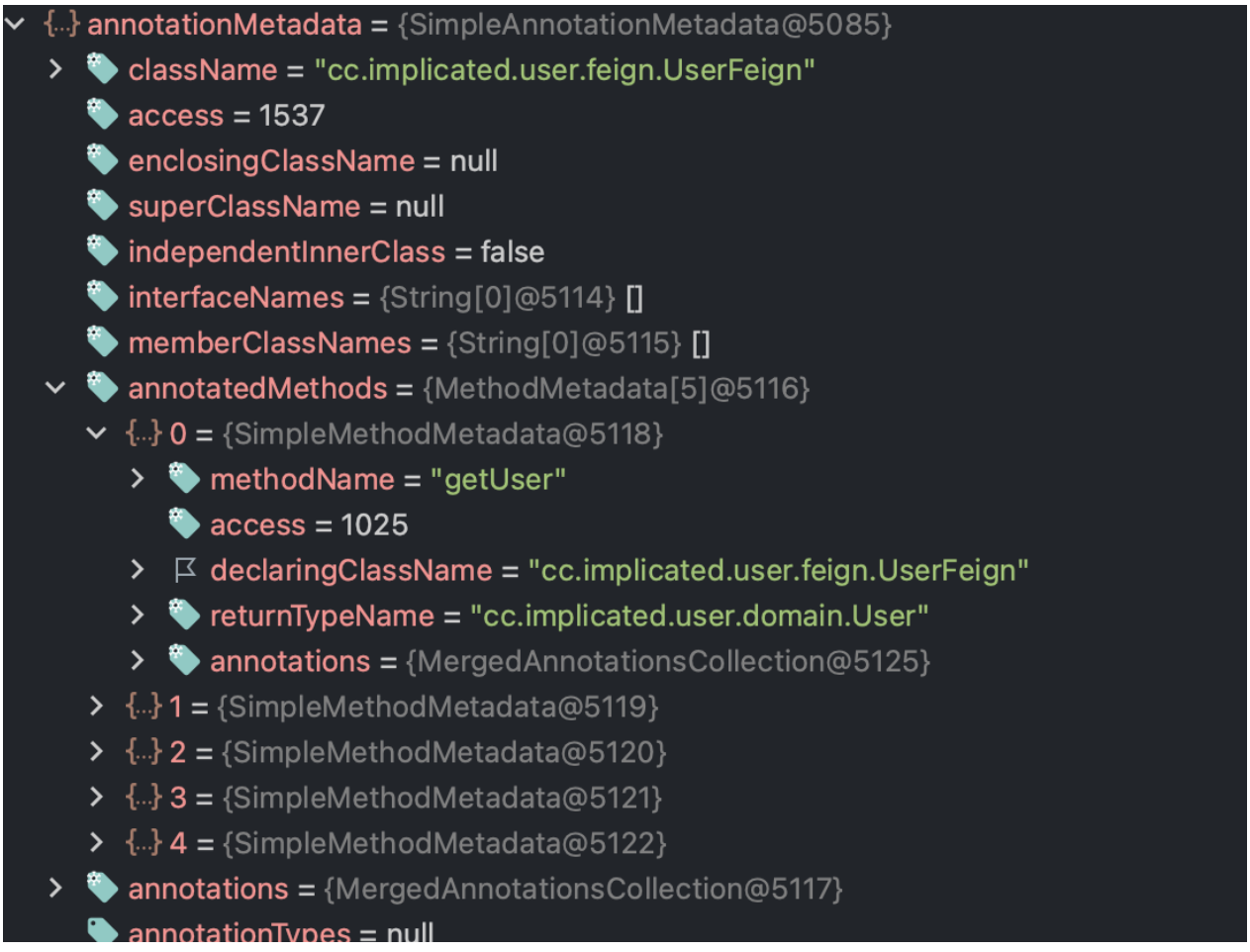
```
public void registerFeignClients(AnnotationMetadata metadata, BeanDefinitionRegistry
registry) {
    // 获取ClassPath扫描器
    ClassPathScanningCandidateComponentProvider scanner = getScanner();
    // 为扫描器设置资源加载器
    scanner.setResourceLoader(this.resourceLoader);

    Set<String> basePackages;

    // 获取 @EnableFeignClients 注解的属性 map
    Map<String, Object> attrs = metadata
            .getAnnotationAttributes(EnableFeignClients.class.getName());
    AnnotationTypeFilter annotationTypeFilter = new AnnotationTypeFilter(
            FeignClient.class);
    // 获取 @EnableFeignClients 注解的 clients 属性
    final Class<?>[] clients = attrs == null ? null
```

```
            : (Class<?>[]) attrs.get("clients");
    // 如果 @EnableFeignClients 注解未指定 clients
属性则扫描添加（扫描过滤条件为：标记有 @FeignClient 的类）
    if (clients == null || clients.length == 0) {
        scanner.addIncludeFilter(annotationTypeFilter);
        basePackages = getBasePackages(metadata);
    }
    // 如果 @EnableFeignClients 注解已指定 clients
属性，则直接添加，不再扫描（从这里可以看出，为了加快容器启动速度，建议都指定 clients
属性）
    else {
        final Set<String> clientClasses = new HashSet<>();
        basePackages = new HashSet<>();
        // 扫描 clients
        for (Class<?> clazz : clients) {
            basePackages.add(ClassUtils.getPackageName(clazz));
            clientClasses.add(clazz.getCanonicalName());
        }
        // 过滤标记有 @FeignClient 的类
        AbstractClassTestingTypeFilter filter = new AbstractClassTestingTypeFilter() {
            @Override
            protected boolean match(ClassMetadata metadata) {
                String cleaned = metadata.getClassName().replaceAll("\\$", ".");
                return clientClasses.contains(cleaned);
            }
        };
        // 同时满足
        scanner.addIncludeFilter(
                new AllTypeFilter(Arrays.asList(filter, annotationTypeFilter)));
    }

    // 遍历最终获取到的 @FeignClient 注解类的集合
    for (String basePackage : basePackages) {
        Set<BeanDefinition> candidateComponents = scanner
                .findCandidateComponents(basePackage);
        for (BeanDefinition candidateComponent : candidateComponents) {
            // 判断是否是带有注解的 Bean
            if (candidateComponent instanceof AnnotatedBeanDefinition) {
                // 验证带注释的类必须是接口，不是接口则直接抛出异常
                // 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");

                // 获取 @FeignClient 注解的属性值
                Map<String, Object> attributes = annotationMetadata
                        .getAnnotationAttributes(
                                FeignClient.class.getCanonicalName());
```

```
                // 获取 clientName 的值，也就是在构造器的参数值
                String name = getClientName(attributes);
                // 同上一个方法最后调用的方法，注入 @FeignClient中的
configuration对象到容器中，name: name.feignClientSpecification
                registerClientConfiguration(registry, name, attributes.get(
"configuration"));

                // 循环注入 @FeignClient 对象，name: className
                registerFeignClient(registry,
                    annotationMetadata, ①
                    attributes); ②

            }
        }
    }
}
```

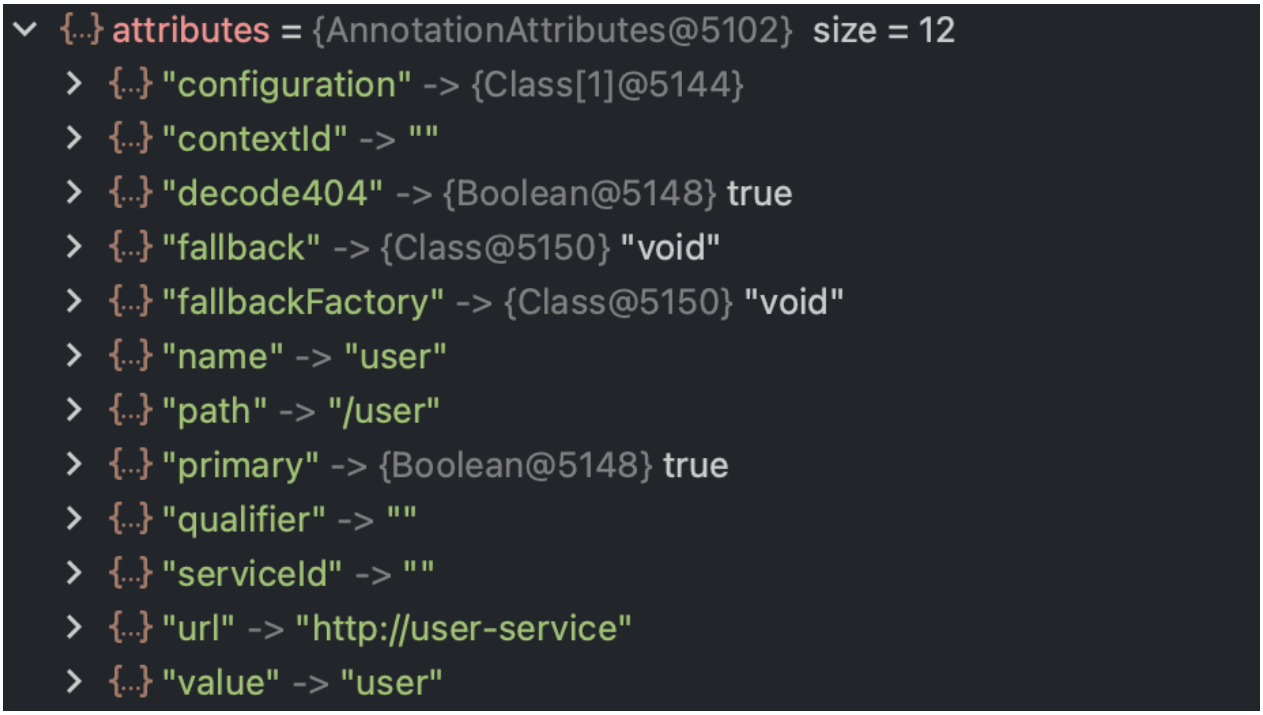
① 接口元数据（拿了个名字）

▼ Details




② `@FeignClient` 注解的属性值

▼ Details




org.springframework.cloud.openfeign.FeignClientsRegistrar#registerFeignClient

```
private void registerFeignClient(BeanDefinitionRegistry registry, AnnotationMetadata
annotationMetadata, Map<String, Object> attributes) { // 获取类名称
    String className = annotationMetadata.getClassName();
    // BeanDefinitionBuilder
的主要作用就是构建一个AbstractBeanDefinition,AbstractBeanDefinition类最终被构建成一个B
eanDefinitionHolder然后注册到Spring中
    // 注意：beanDefinition类为FeignClientFactoryBean，故在
Spring获取类的时候实际返回的是FeignClientFactoryBean类
    BeanDefinitionBuilder definition = BeanDefinitionBuilder
            .genericBeanDefinition(FeignClientFactoryBean.class);
    validate(attributes);
    // 添加FeignClientFactoryBean的属性，
    definition.addPropertyValue("url", getUrl(attributes));
    definition.addPropertyValue("path", getPath(attributes));
    String name = getName(attributes);
    definition.addPropertyValue("name", name);
    // Feign 容器名称
    String contextId = getContextId(attributes);
    definition.addPropertyValue("contextId", contextId);
    definition.addPropertyValue("type", className);
    definition.addPropertyValue("decode404", attributes.get("decode404"));
    definition.addPropertyValue("fallback", attributes.get("fallback"));
    definition.addPropertyValue("fallbackFactory", attributes.get("fallbackFactory"));
    definition.setAutowireMode(AbstractBeanDefinition.AUTOWIRE_BY_TYPE);

    // 设置别名
    String alias = contextId + "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;
    }

    // 定义BeanDefinitionHolder，name是 Feign标记的className
    BeanDefinitionHolder holder = new BeanDefinitionHolder(beanDefinition, className,
            new String[] { alias });
    // 注入FeignClientFactoryBean
    BeanDefinitionReaderUtils.registerBeanDefinition(holder, registry);
}
```

# Chapter 14. FeignClientFactoryBean

`FeignClientFactoryBean` 实现了 `FactoryBean<Object>` 接口。

在注册 `FeignClientFactoryBean` 时，会把 `getObject()` 方法返回的对象注入到 Spring 容器中

org.springframework.cloud.openfeign.FeignClientFactoryBean#getObject

```
@Override
public Object getObject() throws Exception {
    // 返回jdk动态代理，最终注入 Spring 容器的是每个 @FeignClient 标记接口的代理类
    return getTarget();
}
```

## 14.1. `getTarget()`

org.springframework.cloud.openfeign.FeignClientFactoryBean#getTarget

```
<T> T getTarget() {
    // FeignContext 容器，所有配置信息都从这里取
    // 从 FeignAutoConfiguration 注入
    FeignContext context = this.applicationContext.getBean(FeignContext.class); ①
    Feign.Builder builder = feign(context);

    // 未指定 url 属性
    if (!StringUtils.hasText(this.url)) {
        if (!this.name.startsWith("http")) {
            this.url = "http://" + this.name;
        }
        else {
            this.url = this.name;
        }
        // 格式化 url
        this.url += cleanPath();
        // HystrixTargeter.target
        // url 是 name（服务名），根据服务名去负载均衡
        return (T) loadBalance(builder, context,
                new HardCodedTarget<>(this.type, this.name, this.url)); ②
    }
    // 指定 url 属性（url协议头不是必须的）
    if (StringUtils.hasText(this.url) && !this.url.startsWith("http")) {
        this.url = "http://" + this.url;
    }
    // 格式化 url
    String url = this.url + cleanPath();
    // 获取 http client
    Client client = getOptional(context, Client.class); ④
    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();
        }
        if (client instanceof FeignBlockingLoadBalancerClient) {
            // not load balancing because we have a url,
            // but Spring Cloud LoadBalancer is on the classpath, so unwrap
            client = ((FeignBlockingLoadBalancerClient) client).getDelegate();
        }
        builder.client(client);
    }
    Targeter targeter = get(context, Targeter.class); ④
    // DefaultTargeter.target()
    // url 是 @FeignClient 中指定的 URL
    return (T) targeter.target(this, builder, context,
            new HardCodedTarget<>(this.type, this.name, url)); ③
}
```

① 注入 FeignContext，入口在 FeignAutoConfiguration

② 如果配置了 Hystrix，执行此方法

org.springframework.cloud.openfeign.FeignClientFactoryBean#loadBalance

```
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-netflix-ribbon?");
}
```

③ 默认执行的方法

④ get() 和 getOptional() 都是从 Feign 容器中去获取需要的类。其中通过 get() 方法获取的类不能为空。
org.springframework.cloud.openfeign.FeignClientFactoryBean#get

```
protected <T> T get(FeignContext context, Class<T> type) {
    // contextId 是容器名称，type 是要获取的类
    T instance = context.getInstance(this.contextId, type);
    if (instance == null) {
        throw new IllegalStateException(
                "No bean found of type " + type + " for " +
                this.contextId);
    }
    return instance;
```

```
}
```

org.springframework.cloud.openfeign.FeignClientFactoryBean#getOptional

```
protected <T> T getOptional(FeignContext context,Class<T> type) {
    return context.getInstance(this.contextId, type);
}
```

## 14.2. Targeter.target()

Targeter 接口有两个实现 DefaultTargeter 和 HystrixTargeter，根据是否配置熔断分开处理，最终都是执行 Feign.Builder 的 target方法。

org.springframework.cloud.openfeign.DefaultTargeter

```
class DefaultTargeter implements Targeter {

    @Override
    public <T> T target(FeignClientFactoryBean factory, Feign.Builder feign,
            FeignContext context, Target.HardCodedTarget<T> target) {
        // 最终执行 Feign.Builder 的 target方法
        return feign.target(target);
    }
}
```

org.springframework.cloud.openfeign.HystrixTargeter#target

```
class HystrixTargeter implements Targeter {

    @Override
    public <T> T target(FeignClientFactoryBean factory, Feign.Builder feign,
FeignContext context, Target.HardCodedTarget<T> target) {
        // 校验
        if (!(feign instanceof feign.hystrix.HystrixFeign.Builder)) {
            return feign.target(target);
        }

        feign.hystrix.HystrixFeign.Builder builder = (feign.hystrix.HystrixFeign
.Builder) feign;
        String name = StringUtils.isEmpty(factory.getContextId()) ? factory.getName()
            : factory.getContextId();
        SetterFactory setterFactory = getOptional(name, context, SetterFactory.class);
        if (setterFactory != null) {
            builder.setterFactory(setterFactory);
        }
        // fallback 熔断
        Class<?> fallback = factory.getFallback();
        if (fallback != void.class) {
            return targetWithFallback(name, context, target, builder, fallback);
        }
```

```
        // fallbackFactory 熔断
        Class<?> fallbackFactory = factory.getFallbackFactory();
        if (fallbackFactory != void.class) {
            return targetWithFallbackFactory(name, context, target, builder,
                fallbackFactory);
        }

        return feign.target(target);
    }

    private <T> T targetWithFallbackFactory(String feignClientName, FeignContext
context,
                                            Target.HardCodedTarget<T> target,
HystrixFeign.Builder builder,
                                            Class<?> fallbackFactoryClass) {
        FallbackFactory<? extends T> fallbackFactory = (FallbackFactory<? extends T>)
getFromContext(
            "fallbackFactory", feignClientName, context, fallbackFactoryClass,
            FallbackFactory.class);
        return builder.target(target, fallbackFactory);
    }

    private <T> T targetWithFallback(String feignClientName, FeignContext context,
                                     Target.HardCodedTarget<T> target, HystrixFeign
.Builder builder,
                                     Class<?> fallback) {
        T fallbackInstance = getFromContext("fallback", feignClientName, context,
            fallback, target.type());
        return builder.target(target, fallbackInstance);
    }
}
```

feign.hystrix.HystrixFeign.Builder

```
public <T> T target(Target<T> target, T fallback) {
  return build(fallback != null ? new FallbackFactory.Default<T>(fallback) : null)
.newInstance(target);
}

public <T> T target(Target<T> target, FallbackFactory<? extends T> fallbackFactory) {
  return build(fallbackFactory).newInstance(target);
}
```

# Chapter 15. FeignContext

FeignContext 继承自 NamedContextFactory ，实现为每个 @FeignClient 标记的接口创建单独的子容器。

## 15.1. 初始化

FeignContext 由 FeignAutoConfiguration 在启动时注入，FeignAutoConfiguration 被 EnableConfigurationProperties 修饰，启用对 @ConfigurationProperties 注释Bean的支持。。

org.springframework.cloud.openfeign.FeignAutoConfiguration

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(Feign.class)
@EnableConfigurationProperties({FeignClientProperties.class,
FeignHttpClientProperties.class })
@Import(DefaultGzipDecoderConfiguration.class)
public class FeignAutoConfiguration {

    @Autowired(required = false)
    private List<FeignClientSpecification> configurations = new ArrayList<>();

    @Bean
    public HasFeatures feignFeature() {
        return HasFeatures.namedFeature("Feign", Feign.class);
    }

    @Bean
    public FeignContext feignContext() {
        // 注入 FeignContext
        FeignContext context = new FeignContext();
        context.setConfigurations(this.configurations);
        return context;
    }
    //..
}
```

org.springframework.cloud.openfeign.FeignContext#FeignContext

```
public class FeignContext extends NamedContextFactory<FeignClientSpecification> {

    public FeignContext() {
        // FeignClientsConfiguration 是 Feign的默认全局配置
        super(FeignClientsConfiguration.class, "feign", "feign.client.name");
    }
}
```

通过调用父类的 getInstance 方法返回容器中的 Bean。

org.springframework.cloud.context.named.NamedContextFactory#getInstance

```
public <T> T getInstance(String name, Class<T> type) {
```

```
    // 先判断内存中有没有，有直接返回，没有会创建一个
    // 所有子容器保存在一个 ConcurrentHashMap 中
    AnnotationConfigApplicationContext context = getContext(name);
    if (BeanFactoryUtils.beanNamesForTypeIncludingAncestors(context,
            type).length > 0) {
        return context.getBean(type);
    }
    return null;
}
```

## 15.2. 调用

FeignClientFactoryBean 中创建 Feign.Builder 和给 Feign.Builder 设置属性时调用，根据名字从中 ApplicationContext 获取 FeignContext 子容器。

org.springframework.cloud.openfeign.FeignClientFactoryBean

```
protected Feign.Builder feign(FeignContext context) {
    FeignLoggerFactory loggerFactory = get(context, FeignLoggerFactory.class);
    Logger logger = loggerFactory.create(this.type);

    // build()设置默认值
    Feign.Builder builder = get(context, Feign.Builder.class)
            // required values
            .logger(logger)
            .encoder(get(context, Encoder.class))
            .decoder(get(context, Decoder.class))
            .contract(get(context, Contract.class));

    configureFeign(context, builder);

    return builder;
}

protected void configureUsingConfiguration(FeignContext context,
                                           Feign.Builder builder) {
    Logger.Level level = getOptional(context, Logger.Level.class);
    if (level != null) {
        builder.logLevel(level);
    }
    Retryer retryer = getOptional(context, Retryer.class);
    if (retryer != null) {
        builder.retryer(retryer);
    }
    ErrorDecoder errorDecoder = getOptional(context, ErrorDecoder.class);
    if (errorDecoder != null) {
        builder.errorDecoder(errorDecoder);
    }
    Request.Options options = getOptional(context, Request.Options.class);
    if (options != null) {
        builder.options(options);
```

```
    }
    Map<String, RequestInterceptor> requestInterceptors = context
        .getInstances(this.contextId, RequestInterceptor.class);
    if (requestInterceptors != null) {
        builder.requestInterceptors(requestInterceptors.values());
    }
    QueryMapEncoder queryMapEncoder = getOptional(context, QueryMapEncoder.class);
    if (queryMapEncoder != null) {
        builder.queryMapEncoder(queryMapEncoder);
    }
    if (this.decode404) {
        builder.decode404();
    }
}
```

# SpringMVC 注解支持原理

# Chapter 16. SpringMVC 注解支持原理

继承 `Contract.BaseContract`，实现处理参数的方法

```
public class SpringMvcContract extends Contract.BaseContract implements
ResourceLoaderAware {

    public SpringMvcContract(List<AnnotatedParameterProcessor>
annotatedParameterProcessors, ConversionService conversionService) {
        Assert.notNull(annotatedParameterProcessors,
                "Parameter processors can not be null.");
        Assert.notNull(conversionService, "ConversionService can not be null.");

        List<AnnotatedParameterProcessor> processors;
        if (!annotatedParameterProcessors.isEmpty()) {
            processors = new ArrayList<>(annotatedParameterProcessors);
        }
        else {
            processors = getDefaultAnnotatedArgumentsProcessors();
        }
        this.annotatedArgumentProcessors = toAnnotatedArgumentProcessorMap(processors
);
        this.conversionService = conversionService;
        this.convertingExpanderFactory = new ConvertingExpanderFactory
(conversionService);
    }

    private List<AnnotatedParameterProcessor> getDefaultAnnotatedArgumentsProcessors()
{
        List<AnnotatedParameterProcessor> annotatedArgumentResolvers = new ArrayList<
>();

        // 具体实现
        annotatedArgumentResolvers.add(new PathVariableParameterProcessor());
        annotatedArgumentResolvers.add(new RequestParamParameterProcessor());
        annotatedArgumentResolvers.add(new RequestHeaderParameterProcessor());
        annotatedArgumentResolvers.add(new QueryMapParameterProcessor());

        return annotatedArgumentResolvers;
    }

    // 解析方法元数据，方法中会调用下面三个方法
    @Override
    public MethodMetadata parseAndValidateMetadata(Class<?> targetType, Method method)
{
        //...
    }

    // 解析class上注解
    @Override
```

```
    protected void processAnnotationOnClass(MethodMetadata data, Class<?> clz) {
        //...
    }

    // 解析method上注解
    @Override
    protected void processAnnotationOnMethod(MethodMetadata data, Annotation
methodAnnotation, Method method) {
        //...
    }

    // 解析参数
    @Override
    protected boolean processAnnotationsOnParameter(MethodMetadata data, Annotation[]
annotations, int paramIndex) {
        //...
    }

    //...
}
```

# Spring Cloud Open Feign 配置项

# Chapter 17. `@EnableFeignClients`

| param | 作用 |
| --- | --- |
| value | 同basePackages |
| basePackages | 类扫描路径（字符串数组） |
| basePackageClasses | 扫描指定的每个类所在的包下面的所有被 `@FeignClient` 修饰的类 |
| defaultConfiguration | `FeignClient` 的全局配置 |
| clients | 用 `@FeignClient` 注释的类的列表。 |

- `clients` 与类路径扫描互斥。
- 同时指定 `clients` 和类路径扫描，则 `clients` 优先。

# Chapter 18. @FeignClient

| param | 作用 |
| --- | --- |
| value | 同 name |
| contextId | 替换 name 作为 bean 名称 |
| name | FeignClient 的名称，不能为空 |
| qualifier | 指定别名 |
| url | 手动指定 @FeignClient 调用的地址 |
| decode404 | 404解码开关，默认 false |
| configuration | 指定 Feign 的配置类 |
| fallback | 定义容错的处理类， |
| fallbackFactory | 定义容错的处理工厂类 |
| path | 定义当前 FeignClient 的统一前缀 |
| primary | 是否将 Feign 代理 bean<br>标记为主Bean。默认为True。 |

- fallback 指定的类必须实现 @FeignClient 标记的接口。
  - org.springframework.cloud.openfeign.FeignClientsRegistrar#validateFallback
  - org.springframework.cloud.openfeign.FeignClientsRegistrar#validateFallbackFactory
- 在 name 和 url 属性中支持占位符。

```
@FeignClient(name = "${feign.name}", url = "${feign.url}")
public interface FooClient {
    //..
}
```

- 同时指定 Java Bean 和配置属性，则配置属性优先。

# Chapter 19. FeignClientsConfiguration

全局默认值。

| 组件 | 默认值 |
| --- | --- |
| Contract | SpringMvcContract |
| Encoder | SpringEncoder |
| Decoder | SpringDecoder |
| ErrorDecoder | feign.codec.ErrorDecoder.Default |
| Logger | Slf4jLogger |
| Retryer | Retryer.NEVER_RETRY |
| RequestInterceptor | 无 |

# Chapter 20. FeignClient Properties

Feign 客户端配置。

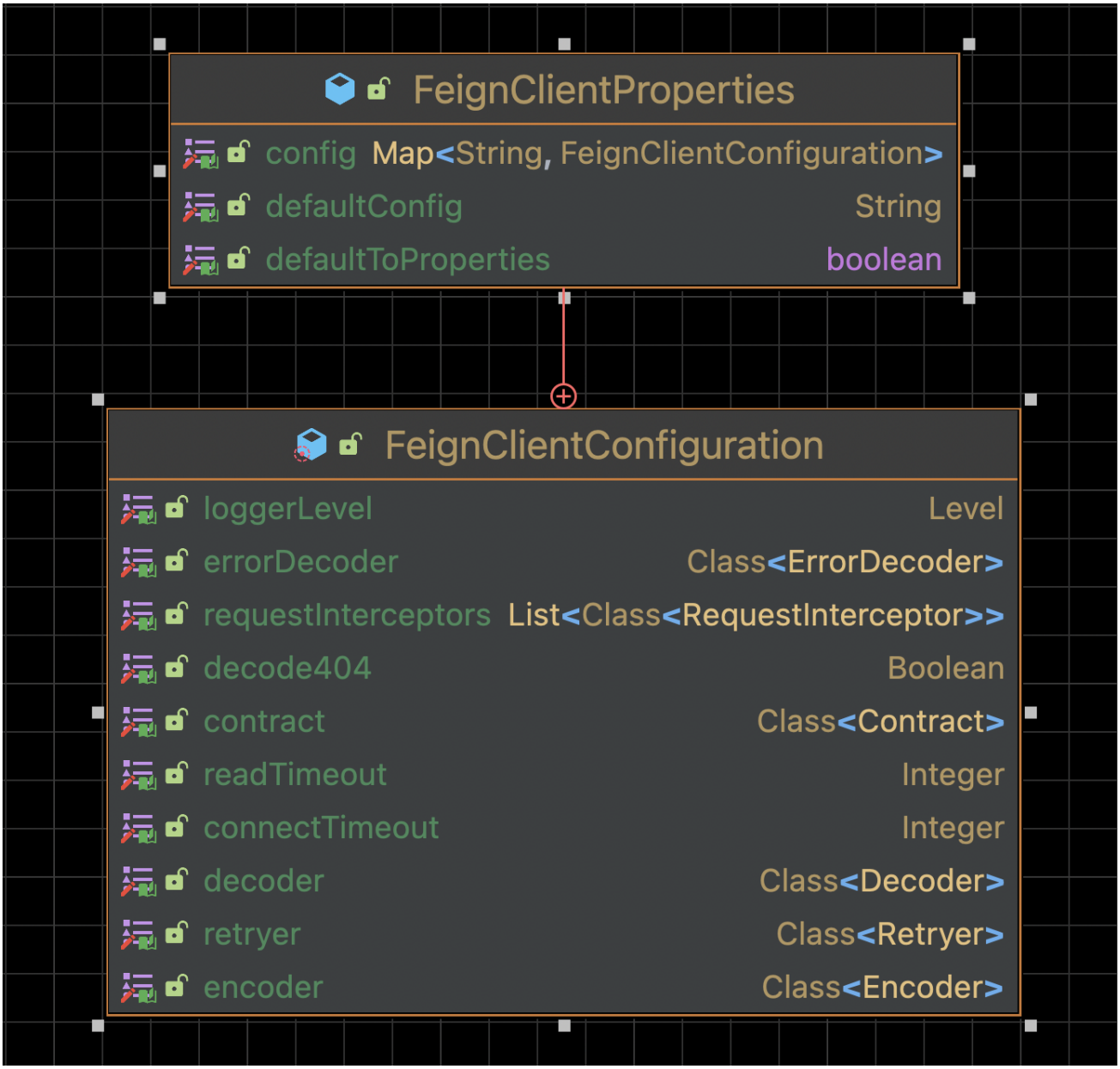
| 配置 | 默认值（Feign默认） | 描述 |
| --- | --- | --- |
| feign.client.default-config | default | 全局配置key |
| feign.client.default-to-properties | true | 配置优先 |
| feign.client.config.<name>.loggerLevel | NONE | 日志级别 |
| feign.client.config.<name>.connectTimeout | 10s | 连接超时时间 |
| feign.client.config.<name>.readTimeout | 60s | 请求超时时间 |
| feign.client.config.<name>.retryer | Retryer.NEVER_RETRY | 重试器 |
| feign.client.config.<name>.decode404 | false | 404解码开关 |
| feign.client.config.<name>.errorDecoder | feign.codec.ErrorDecoder.Default | 错误解码器 |
| feign.client.config.<name>.requestInterceptors | | |
| feign.client.config.<name>.decoder | SpringDecoder | 解码器 |
| feign.client.config.<name>.encoder | SpringEncoder | 编码器 |
| feign.client.config.<name>.contract | SpringMvcContract | 请求解析器 |

# Chapter 21. FeignHttpClient Properties

连接池配置。

| 配置 | 默认值 | 描述 |
| --- | --- | --- |
| feign.httpclient.connection-timeout | 2000 | 连接超时时间 |
| feign.httpclient.connection-timer-repeat | 3000 | 回收超时连接的时间间隔 |
| feign.httpclient.disable-ssl-validation | false | 是否开启 SSL 校验 |
| feign.httpclient.enabled | true | 是否开启 |
| feign.httpclient.follow-redirects | true | 是否允许重定向 |
| feign.httpclient.max-connections | 200 | 线程池最大连接数 |
| feign.httpclient.max-connections-per-route | 50 | 每个 FeignClient 最大连接数 |
| feign.httpclient.time-to-live | 900 | 线程存活时间 |
| feign.httpclient.time-to-live-unit | 秒 | 线程存活时间单位 |

▼ Details



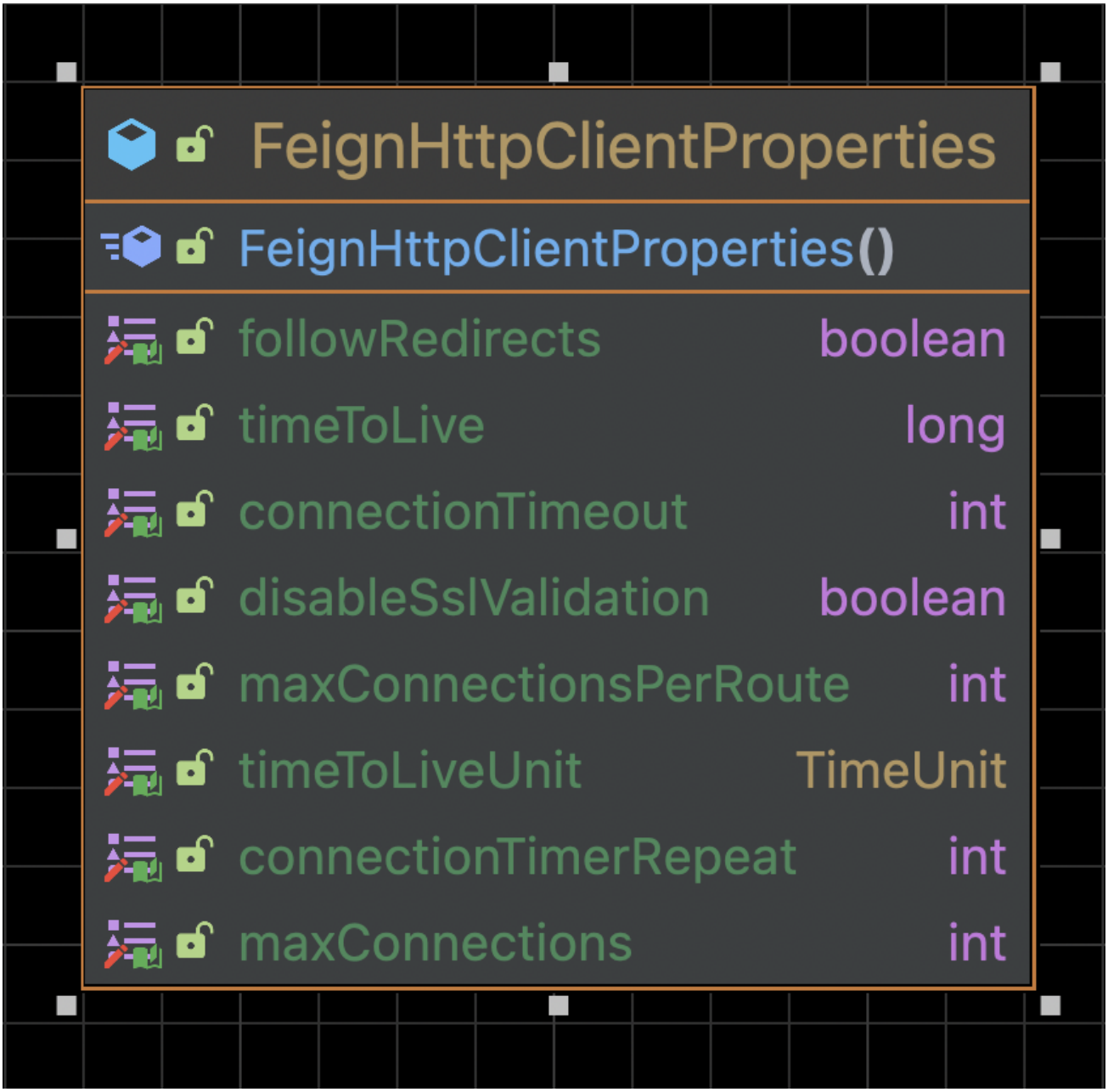
```java
@ConfigurationProperties("feign.client")
public class FeignClientProperties {

    private boolean defaultToProperties = true;

    private String defaultConfig = "default";

    private Map<String, FeignClientConfiguration> config = new HashMap<>();

    //...
}
```

# Chapter 22. Feign Compression Properties

HTTP 压缩配置。

| 配置 | 默认值 | 描述 |
| --- | --- | --- |
| feign.compression.request.enabled | false | 开启请求压缩 |
| feign.compression.request.mime-types | [text/xml, application/xml, application/json] | 配置压缩支持的mime类型列表 |
| feign.compression.request.min-request-size | 2048 | 压缩数据大小的最小阈值 |
| feign.compression.response.enabled | false | 开启响应压缩 |
| feign.compression.response.useGzipDecoder | false | 启用默认的gzip解码器 |

▼ Details




```
@ConfigurationProperties(prefix = "feign.httpclient")
```

```
public class FeignHttpClientProperties {

    /**
     * Default value for disabling SSL validation.
     */
    public static final boolean DEFAULT_DISABLE_SSL_VALIDATION = false;

    /**
     * 默认 httpclient 最大连接数
     */
    public static final int DEFAULT_MAX_CONNECTIONS = 200;

    /**
     * 默认每个 FeignClient 最大连接数
     */
    public static final int DEFAULT_MAX_CONNECTIONS_PER_ROUTE = 50;

    /**
     * 默认存活时间
     */
    public static final long DEFAULT_TIME_TO_LIVE = 900L;

    /**
     * 默认超时时间
     */
    public static final int DEFAULT_CONNECTION_TIMEOUT = 2000;

    /**
     * 默认时间单位 s
     */
    public static final TimeUnit DEFAULT_TIME_TO_LIVE_UNIT = TimeUnit.SECONDS;
    //...
}
```

# Open Feign 的性能优化

# Chapter 23. 设置日志级别

| 级别 | 打印内容 |
| --- | --- |
| NONE （默认值） | 不记录任何日志 |
| BASIC | 仅记录请求方法、URL、响应状态代码以及执行时间 |
| HEADERS | 记录BASIC级别的基础上，记录请求和响应的header |
| FULL | 记录请求和响应的header、body和元数据 |

Feign 日志记录仅响应 `DEBUG` 级别，需要同时设置 Feign 和 Feign 所在包的日志级别为 `DEBUG`

feign.SynchronousMethodHandler#executeAndDecode

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

feign.slf4j.Slf4jLogger#logRequest

```
@Override
protected void logRequest(String configKey, Level logLevel, Request
request) {
    if (logger.isDebugEnabled()) {
      super.logRequest(configKey, logLevel, request);
    }
}
```

# Chapter 24. 设置请求/响应压缩

## 24.1. 概述

gzip 是一种数据格式，采用 deflate 算法压缩数据`，gzip`是一种流行的文件压缩算法，应用十分广泛，尤其是在 Linux 平台。

当 Gzip 压缩一个纯文本文件时，效果是非常明显的，大约可以减少 70% 以上的文件大小。

网络数据经过压缩后实际上降低了网络传输的字节数，最明显的好处就是可以加快网页加载的速度。网页加载速度加快的好处不言而喻，除了节省流量，改善用户的浏览体验外，另一个潜在的好处是 Gzip 与搜索引擎的抓取工具有着更好的关系。

## 24.2. HTTP 协议中关于压缩传输的规定

客户端向服务器请求中带有：Accept-Encoding:gzip，deflate 字段，向服务器表示客户端支持的压缩格式（gzip 或者 deflate），如果不发送该消息头，服务端默认是不会压缩的。

服务端在收到请求之后，如果发现请求头中含有 Accept-Encoding 字段，并且支持该类型压缩，就会对响应报文压缩之后返回给客户端，并且携带 Content-Encoding:gzip 消息头，表示响应报文是根据该格式进行压缩的。

客户端接收到请求之后，先判断是否有 Content-Encoding 消息头，如果有，按该格式解压报文。否则按正常报文处理。

## 24.3. 配置

```yaml
feign:
  compression:
    request:
      enabled: true
      mime-types: text/xml,application/xml,application/json
      min-request-size: 2048
    response:
      enabled: true
```

# Chapter 25. 替换默认的HTTP实现

## 25.1. Apache HttpClient

```
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-httpclient</artifactId>
    <version>${openfeign.version}</version>
</dependency>
```

```
feign:
  httpclient:
    max-connections: 200
    max-connections-per-route: 50
```

## 25.2. OkHttp

性能最好。

```
<dependency>
    <groupId>io.github.openfeign</groupId>
    <artifactId>feign-okhttp</artifactId>
    <version>${openfeign.version}</version>
</dependency>
```

```
feign:
  httpclient:
    max-connections: 200
    max-connections-per-route: 50
  okhttp:
    enable: true
```

# Chapter 26. 添加缓存

可以使用 `@EnableCaching` 为 `Feign` 开启缓存。

```
public interface DemoClient {

    @GetMapping("/demo/{filterParam}")
    @Cacheable(cacheNames = "demo-cache", key = "#keyParam")
    String demoEndpoint(String keyParam, @PathVariable String filterParam);
}
```

还可以通过属性 `spring.cloud.openfeign.cache.enabled=false` 禁用该功能。

# Chapter 27. 自定义异常处理

```
public class CustomErrorDecoder implements ErrorDecoder {
    @Override
    public Exception decode(String methodKey, Response response) {

        switch (response.status()){
            case 400:
                return new BadRequestException();
            case 404:
                return new NotFoundException();
            default:
                return new Exception("error");
        }
    }
}
```

```
public class ClientConfiguration {

    @Bean
    public ErrorDecoder errorDecoder() {
        return new CustomErrorDecoder();
    }
}
```

# Chapter 28. Feign-Form

Feign-form 增加了对 application/x-www-form-urlencoded 和 multipart/form-data 的编码支持。

```
<dependency>
    <groupId>io.github.openfeign.form</groupId>
    <artifactId>feign-form-spring</artifactId>
    <version>3.8.0</version>
</dependency>
```

## 28.1. 上传文件 (multipart/form-data)

```
@FeignClient(configuration = AliyunOssClient.MultipartSupportConfig.class) {

    @Bean
    class MultipartSupportConfig {
        @Bean
        public Encoder feignFormEncoder() {
            return new SpringFormEncoder();
        }
    }
}
```

## 28.2. Form表单提交 (application/x-www-form-urlencoded)

```
@FeignClient(configuration = AliyunOssClient.MultipartSupportConfig.class) {

    @Bean
    class MultipartSupportConfig {
        @Autowired
        private ObjectFactory<HttpMessageConverters> messageConverters;

        @Bean
        public Encoder feignFormEncoder() {
            return new SpringFormEncoder(new SpringEncoder(messageConverters));
        }
    }
}
```

# Feign 的未来

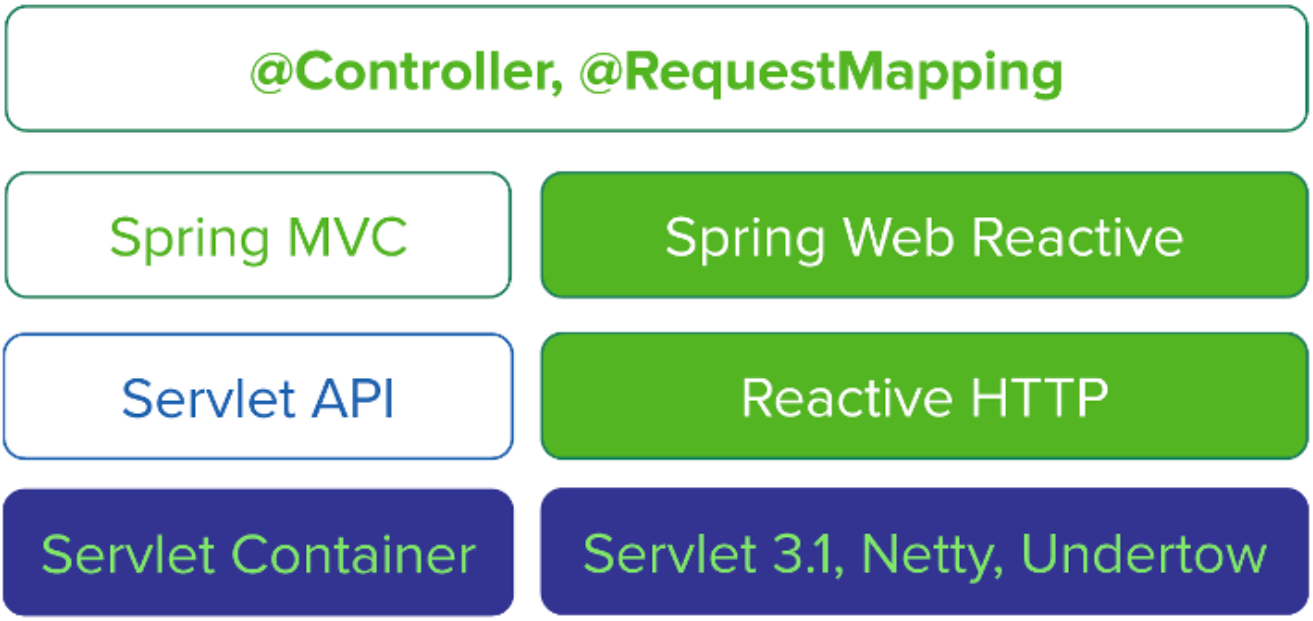
# Chapter 29. 10.8 新特性

Feign 10.8引入了一个新的构建器 AsyncFeign ，允许方法返回 CompletableFuture 个实例。在10.8之前 Feign 都是同步的。

```
interface GitHub {
  @RequestLine("GET /repos/{owner}/{repo}/contributors")
  CompletableFuture<List<Contributor>> contributors(@Param("owner") String owner,
@Param("repo") String repo);
}

public class MyApp {
  public static void main(String... args) {
    GitHub github = AsyncFeign.builder()
                        .decoder(new GsonDecoder())
                        .target(GitHub.class, "https://api.github.com");

    // Fetch and print a list of the contributors to this library.
    CompletableFuture<List<Contributor>> contributors = github.contributors("
OpenFeign", "feign");
    for (Contributor contributor : contributors.get(1, TimeUnit.SECONDS)) {
      System.out.println(contributor.login + " (" + contributor.contributions + ")");
    }
  }
}
```

# Chapter 30. Spring WebClient

`WebClient` 是一个异步非阻塞网络请求客户端，在 `Spring 5` 中引入，属于 `spring-web-reactive` 的一部分。




> Spring Web Reactive Framework
>
> 如何使用OpenFeign+WebClient实现非阻塞的接口聚合 - 掘金
>
> Spring Boot FeignClient vs. WebClient | Baeldung

# 参考文档


> OpenFeign/feign
>
> spring-cloud-openfeign
>
> Spring Cloud OpenFeign Docs
>
> Spring Boot FeignClient vs. WebClient | Baeldung
>
> 10000字 | 深入理解 OpenFeign 的架构原理