⾸先springcloud没有rpc，这就涉及rpc和微服务的区别。springcloud的模块通信⼯具feign跟httpclient和okhttp是⼀样的东西，都是对http请求封装的⼯具，其实feign可以选择httpclient或者okhttp作为底层实现（修改配置即可）。Feign使得写java http客户端更容易。选择Feign有如下原因：

1. Feign允许我们通过⼏个注解的⽅式实现http的第三⽅请求
2. 能够让我们以⾯向对象化的形式简单快速的完成第三⽅请求的开放，解决了代码的冗余
3. Feign插件化的开放定时允许我们⾃定义编码器、解码器、错误处理器、拦截器等等

Feign的作⽤：

①封装http请求，使开发⼈员对发送请求的过程⽆感知，给⼈⼀种伪rpc感觉（这也许是feign这个名字的由来吧，伪装~）。

②feign整合ribbon和hystrix，结合eureka起到负载均衡和熔断器、降级作⽤。

通过源码可知道：

1. Feign通过 Feign.Builder以建造者模式⽅式对各个开放组件进⾏配置

2. target()最终会调⽤new ReflectiveFeign(...)来⽣成Feign实例

3. SynchronousMethodHandler.Factory⽤于创建⼀个SynchronousMethodHandler对象

4. ParseHandlersByName将Target的所有接⼝⽅法转换为Map<String, MethodHandler>对象

5. ReflectiveFeign是Feign的具体实现类，最终会调⽤newInstance⽅法通过原⽣的动态代理框架⽣成最终的动态代理对象

本⽂就不做Feign过多的陈述了，接下来主要讲Feign中代理模式的实现。

4.3.2 ReflectiveFeign.newInstance()

public class ReflectiveFeign extends Feign {

//省略部分⽆关紧要的代码

/\*\*

\* 最终调⽤newInstance返回动态的代理对象

\* @param target

\* @param <T>

\* @return

\*/

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

//通过ParseHandlersByName对象apply⽅法⽣成⽅法名与MethodHandler的映射关系，见图1

Map<String, MethodHandler> nameToHandler = targetToHandlersByName.apply(target);

//构建Method与其对应的MethodHandler的关系存放⼊methodToHandler，主要⽤于在实际调⽤的时候，能够根据⽅法名获取对应的MethodHandler发起请求,见

Map<Method, MethodHandler> methodToHandler = new LinkedHashMap<Method, MethodHandler>();

List<DefaultMethodHandler> defaultMethodHandlers = new LinkedList<DefaultMethodHandler>();

//通过循环构建⽅法与其对应的MethodHandler的关系存放⼊methodToHandler

for (Method method : target.type().getMethods()) {

if (method.getDeclaringClass() == Object.class) {

continue;

} else if(Util.isDefault(method)) {

DefaultMethodHandler handler = new DefaultMethodHandler(method);

defaultMethodHandlers.add(handler);

methodToHandler.put(method, handler);

} else {

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

}

}

//通过InvocationHandlerFactory factory创建代理对象

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

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

//⽣成代理对象

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

for(DefaultMethodHandler defaultMethodHandler : defaultMethodHandlers) {

defaultMethodHandler.bindTo(proxy);

}

return proxy;

}

static class FeignInvocationHandler implements InvocationHandler {

private final Target target;

private final Map<Method, MethodHandler> dispatch;

FeignInvocationHandler(Target target, Map<Method, MethodHandler> dispatch) {

this.target = checkNotNull(target, "target");

this.dispatch = checkNotNull(dispatch, "dispatch for %s", target);

}

/\*\*

\* 请求接⼝的动态代理，在调⽤⽅法的过程中，实际是执⾏该⽅法进⾏调⽤

\* @param proxy

\* @param method

\* @param args

\* @return

\* @throws Throwable

\*/

@Override

public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {

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();

}

//根据⽅法名获取对应的MethodHandler发起请求

return dispatch.get(method).invoke(args);

}

@Override

public boolean equals(Object obj) {

if (obj instanceof FeignInvocationHandler) {

FeignInvocationHandler other = (FeignInvocationHandler) obj;

return target.equals(other.target);

}

return false;

}

@Override

public int hashCode() {

return target.hashCode();

}

@Override

public String toString() {

return target.toString();

}

}

}

}

具体步骤如下：

1. 根据target，解析⽣成MethodHandler对象

2. 构建method与MethodHandler的关系

3. 通过jdk动态代理⽣成代理对象

4. 将DefaultMethodHandler绑定到代理对象

https://www.jb51.net/article/240132.htm