11 RPC 框架代码分析之其他模块

动态代理屏蔽网络传输细节

我们在前面的章节讲到过我们需要用到动态代理来屏蔽复杂的网络传输细节。对应的代码：`RpcClientProxy.java`

Java Copy

```java
@Slf4j
public class RpcClientProxy implements InvocationHandler {
    public <T> T getProxy(Class<T> clazz) {
        return (T) Proxy.newProxyInstance(clazz.getClassLoader(), new Class<?>[]{clazz}, this);
    }
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) {
    }
}
```

当我们去调用一个远程的方法的时候，实际上是通过代理对象调用的。

获取代理对象的方法如下：

Java Copy

```java
    public <T> T getProxy(Class<T> clazz) {
        return (T) Proxy.newProxyInstance(clazz.getClassLoader(), new Class<?>[]{clazz}, this);
    }
```

网络传输细节都被封装在了 `invoke()` 方法中。

```java
    public Object invoke(Object proxy, Method method, Object[] args) {
        log.info("invoked method: [{}]", method.getName());
        RpcRequest rpcRequest = RpcRequest.builder().methodName(method.getName())
                .parameters(args)
                .interfaceName(method.getDeclaringClass().getName())
                .paramTypes(method.getParameterTypes())
                .requestId(UUID.randomUUID().toString())
                .group(rpcServiceProperties.getGroup())
                .version(rpcServiceProperties.getVersion())
                .build();
        RpcResponse<Object> rpcResponse = null;
        if (rpcRequestTransport instanceof NettyRpcClient) {
            CompletableFuture<RpcResponse<Object>> completableFuture = (CompletableFuture<RpcResponse<Object>>) rpcRequestTransport.sendRpcRequest(rpcRequest);
            rpcResponse = completableFuture.get();
        }
        if (rpcRequestTransport instanceof SocketRpcClient) {
            rpcResponse = (RpcResponse<Object>) rpcRequestTransport.sendRpcRequest(rpcRequest);
        }
        this.check(rpcResponse, rpcRequest);
        return rpcResponse.getData();
    }
```

通过注解注册/消费服务

我们这里借用了 Spring 容器相关的功能。核心代码都放在了：`src/main/java/github/javaguide/spring` 包下面。

我们定义两个注解：

- `RcpService` ：注册服务
- `RpcReference` ：消费服务

`RcpService.java`

```java
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
@Inherited
public @interface RpcService {

    /**
     * Service version, default value is empty string
     */
    String version() default "";

    /**
     * Service group, default value is empty string
     */
    String group() default "";

}
```


`RpcReference.java`

```java
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD})
@Inherited
public @interface RpcReference {

    /**
     * Service version, default value is empty string
     */
    String version() default "";

    /**
     * Service group, default value is empty string
     */
    String group() default "";

}
```

简单说一下原理。

我们实现需要 `BeanPostProcessor` 接口并重写 `postProcessBeforeInitialization()`方法和 `postProcessAfterInitialization()` 方法。

Spring bean 在实例化之前会调用 `postProcessBeforeInitialization()` 方法，在 Spring bean 实例化之后会调用 `postProcessAfterInitialization()` 方法。

Java Copy

```java
@Slf4j
@Component
public class SpringBeanPostProcessor implements BeanPostProcessor {

    @Override
    public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {

    }

    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException {

    }
}
```

被我们使用 `RpcService` 和 `RpcReference` 注解的类都算是 Spring Bean。

- 我们可以在 `postProcessBeforeInitialization()` 方法中去判断类上是否有 `RpcService` 注解。如果有的话，就取出 `group` 和 `version` 的值。然后，再调用 `ServiceProvider` 的 `publishService()` 方法发布服务即可！
- 我们可以在 `postProcessAfterInitialization()` 方法中遍历类的属性上是否有 `RpcReference` 注解。如果有的话，我们就通过反射将这个属性赋值即可！