3. 接口 Mock - 手写 RPC 框架项目教程 - 编程导航教程

> 仅供 [编程导航](https://www.code-nav.cn/post/1816420035119853569) 内部成员观看，请勿对外分享！一、需求分析什么是 Mock？RPC 框架的核心功能。

一、需求分析

什么是 Mock?

RPC 框架的核心功能是调用其他远程服务。但是在实际开发和测试过程中，有时可能无法直接访问真实的远程服务，或者访问真实的远程服务可能会产生不可控的影响，例如网络延迟、服务不稳定等。在这种情况下，就需要使用 mock 服务来模拟远程服务的行为，以便进行接口的测试、开发和调试。

mock 是指模拟对象，通常用于测试代码中，特别是在单元测试中，便于我们跑通业务流程。

举个例子，用户服务要调用订单服务，伪代码如下：1747226499385180161_0.19553958024558682

```
class UserServiceImpl {

    void test() {
        doSomething();
        orderService.order();
        doSomething();
    }
}
```

如果订单服务还没上线，那么这个流程就跑不通，只能先把调用订单服务的代码注释掉。

但如果给 orderService 设置一个模拟对象，调用它的 order 方法时，随便返回一个值，就能继续执行后续代码，这就是 mock 的作用。1747226499385180161_0.46483824053058354

为什么要支持 Mock?

虽然 mock 服务并不是 RPC 框架的核心能力，**但是它的开发成本并不高**。而且给 RPC 框架支持 mock 后，开发者就可以轻松调用服务接口、跑通业务流程，不必依赖真实的远程服务，提高使用体验，何乐而不为呢?

我们希望能够用最简单的方式 —— 比如一个配置，就让开发者使用 mock 服务。
1747226499385180161_0.7034308840037213

二、设计方案

前面也提到了，mock 的本质就是为要调用的服务创建模拟对象。

如何创建模拟对象呢？1747226499385180161_0.07807169160384242

在 RPC 项目第一期中，我们就提到了一种动态创建对象的方法 —— 动态代理。之前是通过动态代理创建远程调用对象。同理，我们通过动态代理创建一个 **调用方法时返回固定值** 的对象，不就好了？

三、开发实现

1）我们可以支持开发者通过修改配置文件的方式开启 mock，那么首先给全局配置类 `RpcConfig` 新增 mock 字段，默认值为 false。1747226499385180161_0.6503185041140092

修改的代码如下：

```
@Data
public class RpcConfig {
    ...

    /**
     * 模拟调用
     */
    private boolean mock = false;
}
```

2）在 Proxy 包下新增 `MockServiceProxy` 类，用于生成 mock 代理服务。1747226499385180161_0.5235891582241063

在这个类中，需要提供一个根据服务接口类型返回固定值的方法。

完整代码如下：

```
package com.yupi.yurpc.proxy;

import lombok.extern.slf4j.Slf4j;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

/**
 * Mock 服务代理（JDK 动态代理）
```

```
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @learn <a href="https://codefather.cn">编程宝典</a>
 * @from <a href="https://yupi.icu">编程导航知识星球</a>
 */
@Slf4j
public class MockServiceProxy implements InvocationHandler {

    /**
     * 调用代理
     *
     * @return
     * @throws Throwable
     */
    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        // 根据方法的返回值类型，生成特定的默认值对象
        Class<?> methodReturnType = method.getReturnType();
        log.info("mock invoke {}", method.getName());
        return getDefaultObject(methodReturnType);
    }

    /**
     * 生成指定类型的默认值对象（可自行完善默认值逻辑）
     *
     * @param type
     * @return
     */
    private Object getDefaultObject(Class<?> type) {
        // 基本类型
        if (type.isPrimitive()) {
            if (type == boolean.class) {
                return false;
            } else if (type == short.class) {
                return (short) 0;
            } else if (type == int.class) {
                return 0;
            } else if (type == long.class) {
                return 0L;
            }
        }
        // 对象类型
        return null;
    }
}
```

在上述代码中，通过 `getDefaultObject` 方法，根据代理接口的 class 返回不同的默认值，比如针对 boolean 类型返回 false、对象类型返回 null 等。

3）给 `ServiceProxyFactory` 服务代理工厂新增获取 mock 代理对象的方法 `getMockProxy` 。可以通过读取已定义的全局配置 `mock` 来区分创建哪种代理对象。

修改 ServiceProxyFactory，完整代码如下：1747226499385180161_0.793582310599189

```
package com.yupi.yurpc.proxy;

import com.yupi.yurpc.RpcApplication;

import java.lang.reflect.Proxy;

/**
 * 服务代理工厂（用于创建代理对象）
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @learn <a href="https://codefather.cn">编程宝典</a>
 * @from <a href="https://yupi.icu">编程导航知识星球</a>
 */
public class ServiceProxyFactory {

    /**
     * 根据服务类获取代理对象
     *
     * @param serviceClass
     * @param <T>
     * @return
     */
    public static <T> T getProxy(Class<T> serviceClass) {
        if (RpcApplication.getRpcConfig().isMock()) {
            return getMockProxy(serviceClass);
        }

        return (T) Proxy.newProxyInstance(
                serviceClass.getClassLoader(),
                new Class[]{serviceClass},
                new ServiceProxy());
    }

    /**
     * 根据服务类获取 Mock 代理对象
     *
     * @param serviceClass
     * @param <T>
     * @return
     */
    public static <T> T getMockProxy(Class<T> serviceClass) {
        return (T) Proxy.newProxyInstance(
                serviceClass.getClassLoader(),
                new Class[]{serviceClass},
                new MockServiceProxy());
    }
}
```

有些视频教程是把 mock 的逻辑写在之前的远程调用动态代理中，我会建议大家单独针对 mock 的场景写一套新的动态代理和代理工厂，不要和真实请求的代理逻辑混在一起。

四、测试

1）可以在 `example-common` 模块的 UserService 中写个具有默认实现的新方法。等下需要调用该方法来测试 mock 代理服务是否生效，即查看调用的是模拟服务还是真实服务。

代码如下：

```
package com.yupi.example.common.service;

import com.yupi.example.common.model.User;

/**
 * 用户服务
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @learn <a href="https://codefather.cn">编程宝典</a>
 * @from <a href="https://yupi.icu">编程导航知识星球</a>
 */
public interface UserService {

    /**
     * 获取用户
     *
     * @param user
     * @return
     */
    User getUser(User user);

    /**
     * 新方法 - 获取数字
     */
    default short getNumber() {
        return 1;
    }
}
```

2）修改示例服务消费者模块中的 `application.properties` 配置文件，将 mock 设置为 true：

```
rpc.name=yurpc
rpc.version=2.0
rpc.mock=true
```

3）修改 `ConsumerExample` 类，编写调用 `userService.getNumber` 的测试代码。1747226499385180161_0.6592082855645234

代码如下：

```
package com.yupi.example.consumer;

import com.yupi.example.common.model.User;
import com.yupi.example.common.service.UserService;
import com.yupi.yurpc.proxy.ServiceProxyFactory;

/**
 * 简易服务消费者示例
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @learn <a href="https://codefather.cn">编程宝典</a>
 * @from <a href="https://yupi.icu">编程导航知识星球</a>
 */
public class ConsumerExample {

    public static void main(String[] args) {
        // 获取代理
        UserService userService = ServiceProxyFactory.getProxy(UserService.class);
        User user = new User();
        user.setName("yupi");
        // 调用
        User newUser = userService.getUser(user);
        if (newUser != null) {
            System.out.println(newUser.getName());
        } else {
            System.out.println("user == null");
        }
        long number = userService.getNumber();
        System.out.println(number);
    }
}
```

应该能看到输出的结果值为 0，而不是 1，说明调用了 `MockServiceProxy` 模拟服务代理。当然也可以通过 Debug 的方式进行验证。1747226499385180161_0.6977309963979534

五、扩展

1）完善 Mock 的逻辑，支持更多返回类型的默认值生成。

参考思路：使用 Faker 之类的伪造数据生成库，来生成默认值。1747226499385180161_0.2581768446985808

1747226499385180161_0.3647999251532814