rpc全局配置加载

rpc的一些配置不能固定写死，需要让引入的消费者和提供者配置自定义rpc框架

常见的配置项有：注册中心地址（用于服务的注册和发现），通信协议（服务和消费者选择合适的通讯协议），序列化方式（两者都需要指定同样的序列化方式）等等。

先简单的四个配置项

name rpc名称

version 版本号

serverHost 服务提供者地址

serverPort 服务提供者端口

1. 项目初始化

新建ayouok-rpc-core模块，将之前的rpc-easy代码复制过去

将消费者和提供者依赖的模块换成ayouok-rpc-core

2. 实现配置加载

2.1. 新建rpcconfig配置类

```java
package com.yupi.yurpc.config;

import lombok.Data;

/**
 * RPC 框架配置
 */
@Data
public class RpcConfig {

    /**
     * 名称
     */
    private String name = "yu-rpc";

    /**
     * 版本号
     */
    private String version = "1.0";

    /**
     * 服务器主机名
     */
    private String serverHost = "localhost";

    /**
     * 服务器端口号
     */
    private Integer serverPort = 8080;

}
```

2.2. 加载工具类

使用流程

1. 参数是最后得到的配置类对象和配置文件中前缀
2. 传入RpcConfig类型和rpc
3. 拼好文件名，先找到明确的配置文件，然后再根据前缀读取指定的配置
4. 最后返回RpcConfig类型的对象

Java

```java
package com.yupi.yurpc.utils;

import cn.hutool.core.util.StrUtil;
import cn.hutool.setting.dialect.Props;

/**
 * 配置工具类
 */
public class ConfigUtils {

    /**
     * 加载配置对象
     *
     * @param tClass
     * @param prefix
     * @param <T>
     * @return
     */
    public static <T> T loadConfig(Class<T> tClass, String prefix) {
        return loadConfig(tClass, prefix, "");
    }

    /**
     * 加载配置对象，支持区分环境
     *
     * @param tClass
     * @param prefix
     * @param environment
     * @param <T>
     * @return
     */
    public static <T> T loadConfig(Class<T> tClass, String prefix, String environment) {
        StringBuilder configFileBuilder = new StringBuilder("application");
        if (StrUtil.isNotBlank(environment)) {
            configFileBuilder.append("-").append(environment);
        }
        configFileBuilder.append(".properties");
        Props props = new Props(configFileBuilder.toString());
        return props.toBean(tClass, prefix);
    }
}

```

2.3. 新建rpcconstant常量

```java
package com.yupi.yurpc.constant;

/**
 * RPC 相关常量
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>
 * @learn <a href="https://codefather.cn">鱼皮的编程宝典</a>
 * @from <a href="https://yupi.icu">编程导航学习圈</a>
 */
public interface RpcConstant {

    /**
     * 默认配置文件加载前缀
     */
    String DEFAULT_CONFIG_PREFIX = "rpc";
}

```

2.4. 可以读取的配置参数样式

```properties
rpc.name=yurpc
rpc.version=2.0
rpc.serverPort=8081
```

2.5. 封装全局配置对象

消费者和提供者引入并使用rpc框架时，提供一个方法获取rpc配置对象，并且是单例模式，不用每次调用都再重新读取配置文件再生成配置对象

使用流程

1. 在消费者/提供者项目启动的时候，调用init方法
2. init方法里首先会读取配置文件，生成rpcConfig对象
3. 然后再调用重载的int方法，给类变量赋值
4. 当再调用getConfig时，直接返回类变量

Properties

```
package com.yupi.yurpc;

import com.yupi.yurpc.config.RpcConfig;
import com.yupi.yurpc.constant.RpcConstant;
import com.yupi.yurpc.utils.ConfigUtils;
import lombok.extern.slf4j.Slf4j;

/**
 * RPC 框架应用
 * 相当于 holder，存放了项目全局用到的变量。双检锁单例模式实现
 */
@Slf4j
public class RpcApplication {

    private static volatile RpcConfig rpcConfig;

    /**
     * 框架初始化，支持传入自定义配置
     *
     * @param newRpcConfig
     */
    public static void init(RpcConfig newRpcConfig) {
        rpcConfig = newRpcConfig;
        log.info("rpc init, config = {}", newRpcConfig.toString());
    }

    /**
     * 初始化
     */
    public static void init() {
        RpcConfig newRpcConfig;
        try {
            newRpcConfig = ConfigUtils.loadConfig(RpcConfig.class, RpcConstant.DEFAULT_CONFIG_PREFIX);
        } catch (Exception e) {
            // 配置加载失败，使用默认值
            newRpcConfig = new RpcConfig();
        }
        init(newRpcConfig);
    }

    /**
     * 获取配置
     *
     * @return
```

```
     */
    public static RpcConfig getRpcConfig() {
        if (rpcConfig == null) {
            synchronized (RpcApplication.class) {
                if (rpcConfig == null) {
                    init();
                }
            }
        }
        return rpcConfig;
    }
}
```

3. 添加mock功能

mock功能类似于测试挡板，但是比挡板功能强大

使用流程

1. 消费者引入rpc后，提供者还没开发完成，没办法提供服务，消费者只能模拟一下调用提供者服务后，走通流程；
2. 使用动态代理，生成一个动态的生成一个服务代理对象，不用每次新调用一个服务，再另写一套代码了
3. 使用ServiceProxyFactory生成代理服务时，先判断是否配置开启mock，如果开启，调用生成mock代理对象的代理类。

Java

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

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

```

Java

```java
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;
    }
}
```

Java

```java
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());
```

```
46          }
47      }
48
```