11_启动机制和注解驱动

> 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

通过前面的教程，我们 RPC 框架的功能已经比较完善了，接下来我们就要思考如何优化这个框架。

框架是给开发者用的，让我们换位思考：如果你是一名开发者，会选择怎样的一款框架呢？

答案很简单，就是选择符合自身需求的呗！

往细了说，鱼皮会更关注框架的这些情况：

- 框架的知名度和用户数：尽量选主流的、用户多的，经过了充分的市场验证。
- 生态和社区活跃度：尽量选社区活跃的、能和其他技术兼容的。
- 简单易用易上手：最好能开箱即用，不用花很多时间去上手。这点可能是我们在做个人小型项目时最关注的，可以把精力聚焦到业务开发上。

选择框架的过程其实还有一个专业术语 —— 技术选型，大家可以阅读鱼皮的 这篇文章 <https://mp.weixin.qq.com/s?__biz=MzI1NDczNTAwMA==&mid=2247541448&idx=1&sn=fa08326ae7e25c85b7d18436b6d9ccff&chksm=e9c2c53fdeb54c29c6699260ed5cf343c4fee5fc3cf81e16fccc98c36aecb50a9bb94f29cae0&token=855278184&lang=zh_CN#rd> 详细了解技术选型。

回归到我们的 RPC 项目，其实框架目前是不够易用的。还记得么？光是我们的示例服务提供者，就要写下面这段又臭又长的代码！

```java
public class ProviderExample {

    public static void main(String[] args) {
        ProviderBootstrap.ServiceRegisterInfo
        // RPC 框架初始化
        ProviderBootstrap.init();

        // 注册服务
        String serviceName = UserService.class.getName();
        LocalRegistry.register(serviceName, UserServiceImpl.class);

        // 注册服务到注册中心
        RpcConfig rpcConfig = RpcApplication.getRpcConfig();
        RegistryConfig registryConfig = rpcConfig.getRegistryConfig();
        Registry registry = RegistryFactory.getInstance(registryConfig.getReg
        ServiceMetaInfo serviceMetaInfo = new ServiceMetaInfo();
        serviceMetaInfo.setServiceName(serviceName);
        serviceMetaInfo.setServiceHost(rpcConfig.getServerHost());
        serviceMetaInfo.setServicePort(rpcConfig.getServerPort());
        try {
            registry.register(serviceMetaInfo);
        } catch (Exception e) {
            throw new RuntimeException(e);
        }

        // 启动 web 服务
        VertxTcpServer vertxTcpServer = new VertxTcpServer();
        vertxTcpServer.doStart(8080);
    }
}
```

本节教程，我们就来优化框架的易用性，通过建立合适的启动机制和注解驱动机制，帮助开发者最少只用一行代码，就能轻松使用框架！

二、设计方案

让我们先来站在上帝视角，思考一下：怎么能让开发者用更少的代码启动框架？

启动机制设计

其实很简单，把所有启动代码封装成一个 **专门的启动类** 或方法，然后由服务提供者 / 服务消费者调用即可。

但有一点我们需要注意，服务提供者和服务消费者需要初始化的模块是不同的，比如服务消费者不需要启动 Web 服务器。

所以我们需要针对服务提供者和消费者分别编写一个启动类，如果是二者都需要初始化的模块，可以放到全局应用类 `RpcApplication` 中，复用代码的同时保证启动类的可维护、可扩展性。

在 Dubbo 中，就有类似的设计，参考文档：https://cn.dubbo.apache.org/zh-cn/overview/mannual/java-sdk/quick-start/api/ <https://cn.dubbo.apache.org/zh-cn/overview/mannual/java-sdk/quick-start/api/> 。

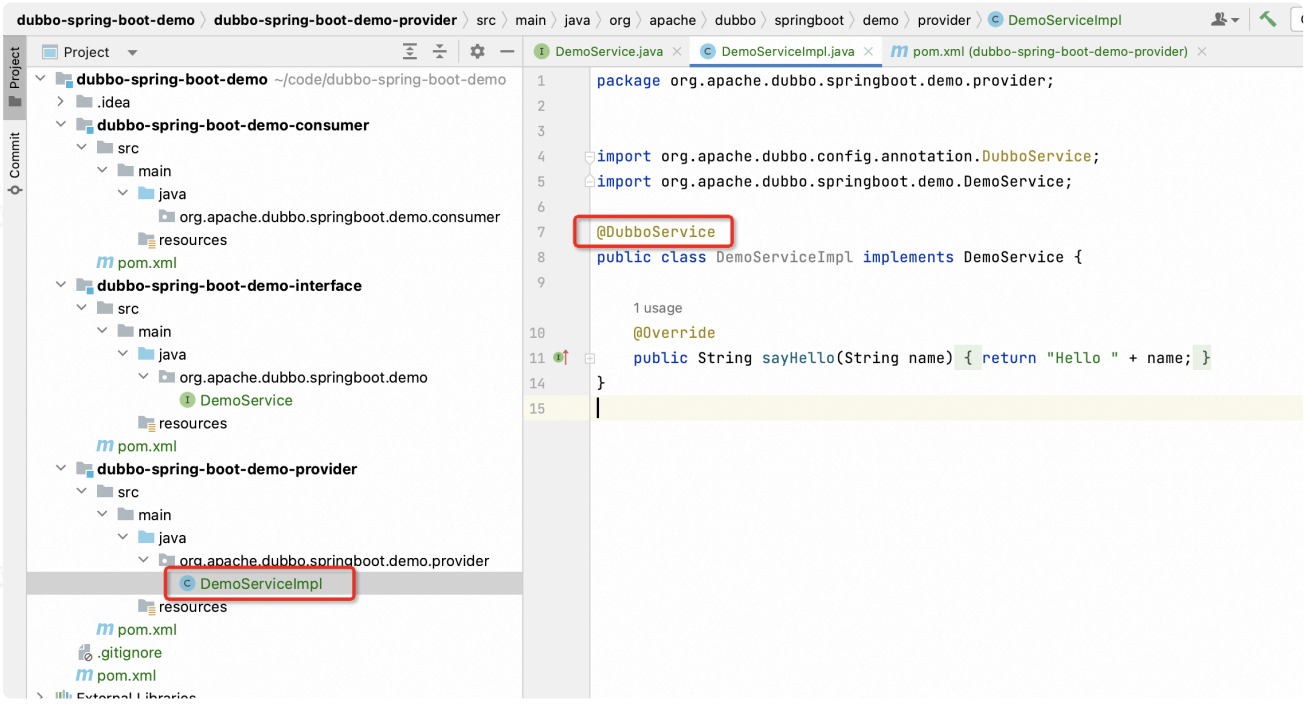
注解驱动设计

除了启动类外，其实还有一种更牛的方法，能帮助开发者使用框架。

学过 Dubbo 这款 RPC 框架的同学应该会有印象，Dubbo 中是如何让开发者快速使用框架的呢？

它的做法是 **注解驱动**，开发者只需要在服务提供者实现类打上一个 `DubboService` 注解，就能快速注册服务；同样的，只要在服务消费者字段打上一个 `DubboReference` 注解，就能快速使用服务。

如图：



由于现在的 Java 项目基本都使用 Spring Boot 框架，所以 Dubbo 还贴心地推出了 Spring Boot Starter，用更少的代码在 Spring Boot 项目中使用框架。

参考文档：https://cn.dubbo.apache.org/zh-cn/overview/mannual/java-sdk/quick-start/spring-boot/ <https://cn.dubbo.apache.org/zh-cn/overview/mannual/java-sdk/quick-start/spring-boot/>

那我们也可以有样学样，创建一个 Spring Boot Starter 项目，并通过注解驱动框架的初始化，完成服务注册和获取引用。

1）关于 Spring Boot Starter 的开发，鱼皮以前写过一篇 Starter 教程 <https://mp.weixin.qq.com/s?__biz=MzI1NDczNTAwMA==&mid=2247530836&idx=1&sn=2ed9f251e8ff211d7e1c2455e636d1ae&chksm=e9c29ea3deb517b58bd0643ce5c1da63516a1a5de60b4cd2448cce4c41d67d281f4c85d1a67e&token=720877586&lang=zh_CN#rd> ，并且在 编程导航 <https://yuyuanweb.feishu.cn/wiki/SePYwTc9tipQiCktw7Uc7kujnCd> 的 API 开放平台项目中，带大家实践过基于 Spring Boot Starter 的 SDK 开发。

2）实现注解驱动并不复杂，有 2 种常用的方式：

1. 主动扫描：让开发者指定要扫描的路径，然后遍历所有的类文件，针对有注解的类文件，执行自定义的操作。
2. 监听 Bean 加载：在 Spring 项目中，可以通过实现 BeanPostProcessor 接口，在 Bean 初始化后执行自定义的操作。

有了思路后，下面我们依次开发实现启动机制和注解驱动。

三、开发实现

启动机制

我们在 rpc 项目中新建包名 `bootstrap` ，所有和框架启动初始化相关的代码都放到该包下。

服务提供者启动类

新建 `ProviderBootstrap` 类，先直接复制之前服务提供者示例项目中的初始化代码，然后略微改造，支持用户传入自己要注册的服务。

在注册服务时，我们需要填入多个字段，比如服务名称、服务实现类，参考代码如下：

```
// 注册服务
String serviceName = UserService.class.getName();
LocalRegistry.register(serviceName, UserServiceImpl.class);
```

我们可以将这些字段进行封装，在 `model` 包下新建 `ServiceRegisterInfo` 类，代码如下：

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

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

/**
 * 服务注册信息类
 */
@Data
@AllArgsConstructor
@NoArgsConstructor
public class ServiceRegisterInfo<T> {

    /**
     * 服务名称
     */
    private String serviceName;

    /**
     * 实现类
     */
    private Class<? extends T> implClass;
}
```

这样一来，服务提供者的初始化方法只需要接受封装的注册信息列表作为参数即可，简化了方法。

服务提供者完整代码如下：

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

import com.yupi.yurpc.RpcApplication;
import com.yupi.yurpc.config.RegistryConfig;
import com.yupi.yurpc.config.RpcConfig;
import com.yupi.yurpc.model.ServiceMetaInfo;
import com.yupi.yurpc.model.ServiceRegisterInfo;
import com.yupi.yurpc.registry.LocalRegistry;
import com.yupi.yurpc.registry.Registry;
import com.yupi.yurpc.registry.RegistryFactory;
import com.yupi.yurpc.server.tcp.VertxTcpServer;

import java.util.List;

/**
 * 服务提供者初始化
 */
public class ProviderBootstrap {

    /**
     * 初始化
     */
    public static void init(List<ServiceRegisterInfo<?>> serviceRegisterInfoL
        // RPC 框架初始化（配置和注册中心）
        RpcApplication.init();
        // 全局配置
        final RpcConfig rpcConfig = RpcApplication.getRpcConfig();

        // 注册服务
        for (ServiceRegisterInfo<?> serviceRegisterInfo : serviceRegisterInfo
            String serviceName = serviceRegisterInfo.getServiceName();
            // 本地注册
            LocalRegistry.register(serviceName, serviceRegisterInfo.getImplCl

            // 注册服务到注册中心
            RegistryConfig registryConfig = rpcConfig.getRegistryConfig();
            Registry registry = RegistryFactory.getInstance(registryConfig.ge
            ServiceMetaInfo serviceMetaInfo = new ServiceMetaInfo();
            serviceMetaInfo.setServiceName(serviceName);
            serviceMetaInfo.setServiceHost(rpcConfig.getServerHost());
            serviceMetaInfo.setServicePort(rpcConfig.getServerPort());
            try {
                registry.register(serviceMetaInfo);
            } catch (Exception e) {
                throw new RuntimeException(serviceName + " 服务注册失败", e);
            }
        }

        // 启动服务器
        VertxTcpServer vertxTcpServer = new VertxTcpServer();
```

5277字

```
        vertxTcpServer.doStart(rpcConfig.getServerPort());
    }
}
```

现在，我们想要在服务提供者项目中使用 RPC 框架，就非常简单了。只需要定义要注册的服务列表，然后一行代码调用 `ProviderBootstrap.init` 方法即可完成初始化。

示例代码如下：

```
package com.yupi.example.provider;

import com.yupi.example.common.service.UserService;
import com.yupi.yurpc.bootstrap.ProviderBootstrap;
import com.yupi.yurpc.model.ServiceRegisterInfo;

import java.util.ArrayList;
import java.util.List;

/**
 * 服务提供者示例
 *
 * @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 ProviderExample {

    public static void main(String[] args) {
        // 要注册的服务
        List<ServiceRegisterInfo> serviceRegisterInfoList = new ArrayList<>()
        ServiceRegisterInfo serviceRegisterInfo = new ServiceRegisterInfo(Use
        serviceRegisterInfoList.add(serviceRegisterInfo);

        // 服务提供者初始化
        ProviderBootstrap.init(serviceRegisterInfoList);
    }
}
```

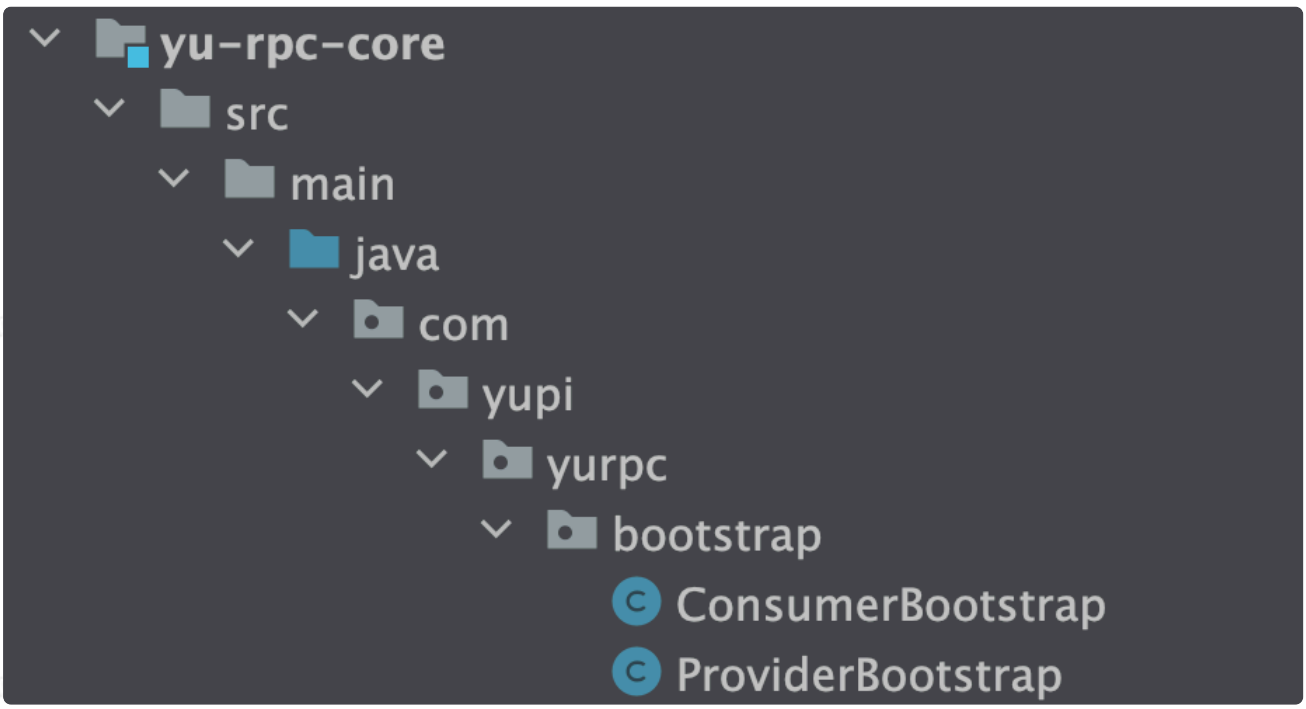
服务消费者启动类

服务消费者启动类的实现就更简单了，因为它不需要注册服务、也不需要启动 Web 服务器，只需要执行 `RpcApplication.init` 完成框架的通用初始化即可。

服务消费者启动类的完整代码如下：

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

import com.yupi.yurpc.RpcApplication;

/**
 * 服务消费者启动类（初始化）
 *
 * @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 class ConsumerBootstrap {

    /**
     * 初始化
     */
    public static void init() {
        // RPC 框架初始化（配置和注册中心）
        RpcApplication.init();
    }

}
```

目前的项目结构如图：



服务消费者示例项目的代码不会有明显的变化，只不过改为调用启动类了。

示例代码如下：

```java
public class ConsumerExample {

    public static void main(String[] args) {
        // 服务提供者初始化
        ConsumerBootstrap.init();

        // 获取代理
        UserService userService = ServiceProxyFactory.getProxy(UserService.cl
        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");
        }
    }
}
```

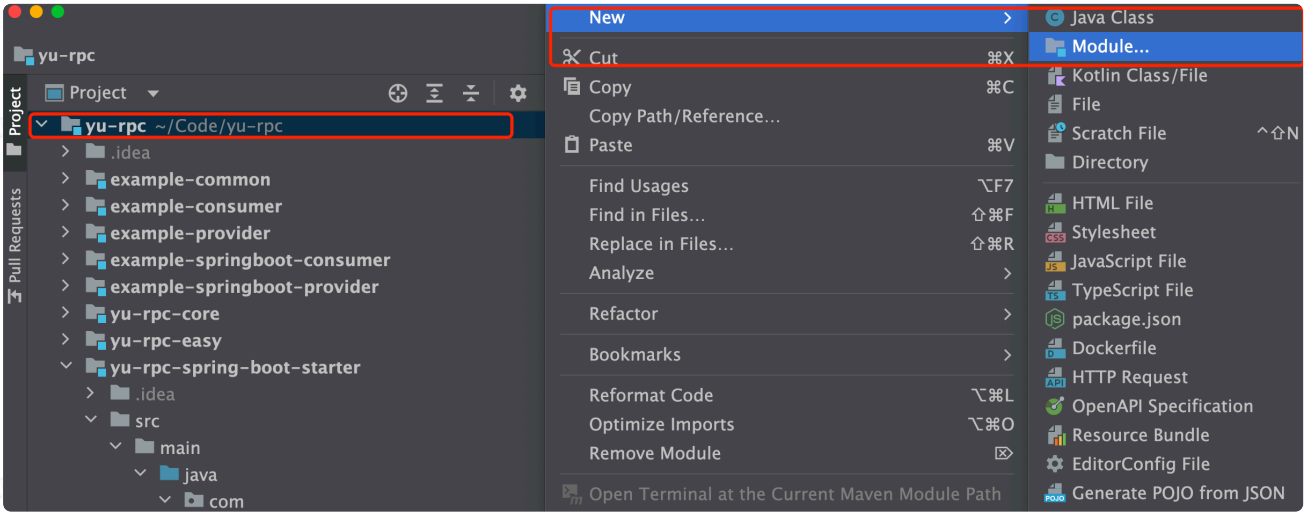
Spring Boot Starter 注解驱动

注意，**为了便于大家学习**，不要和已有项目的代码混淆，我们再来创建一个新的项目模块，专门用于实现 Spring Boot Starter 注解驱动的 RPC 框架。

Dubbo 是在框架内引入了 spring-context，会让整个框架更内聚，但是不利于学习理解。

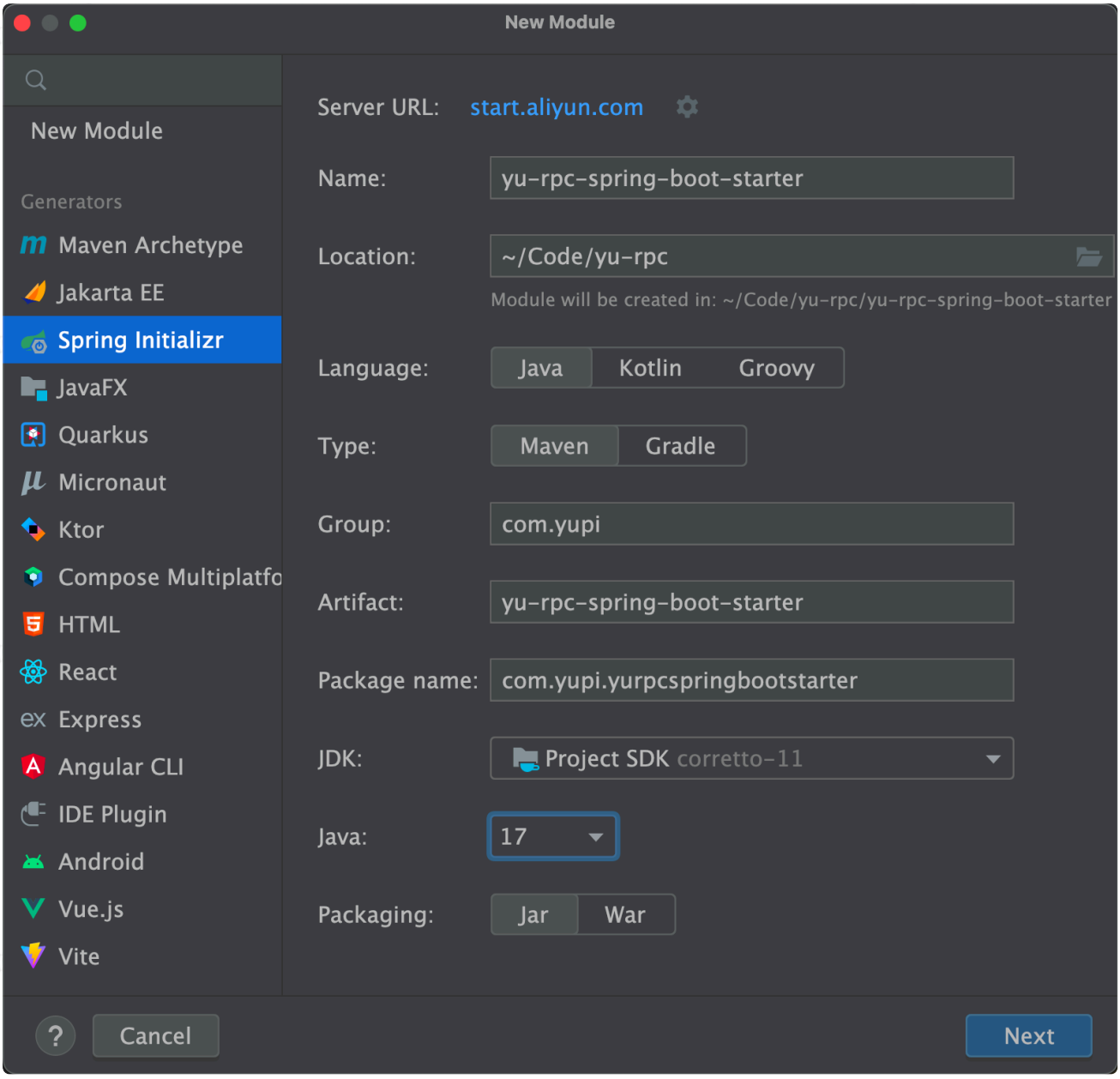
1、Spring Boot Starter 项目初始化

在项目根目录（yu-rpc）处右键新建模块：

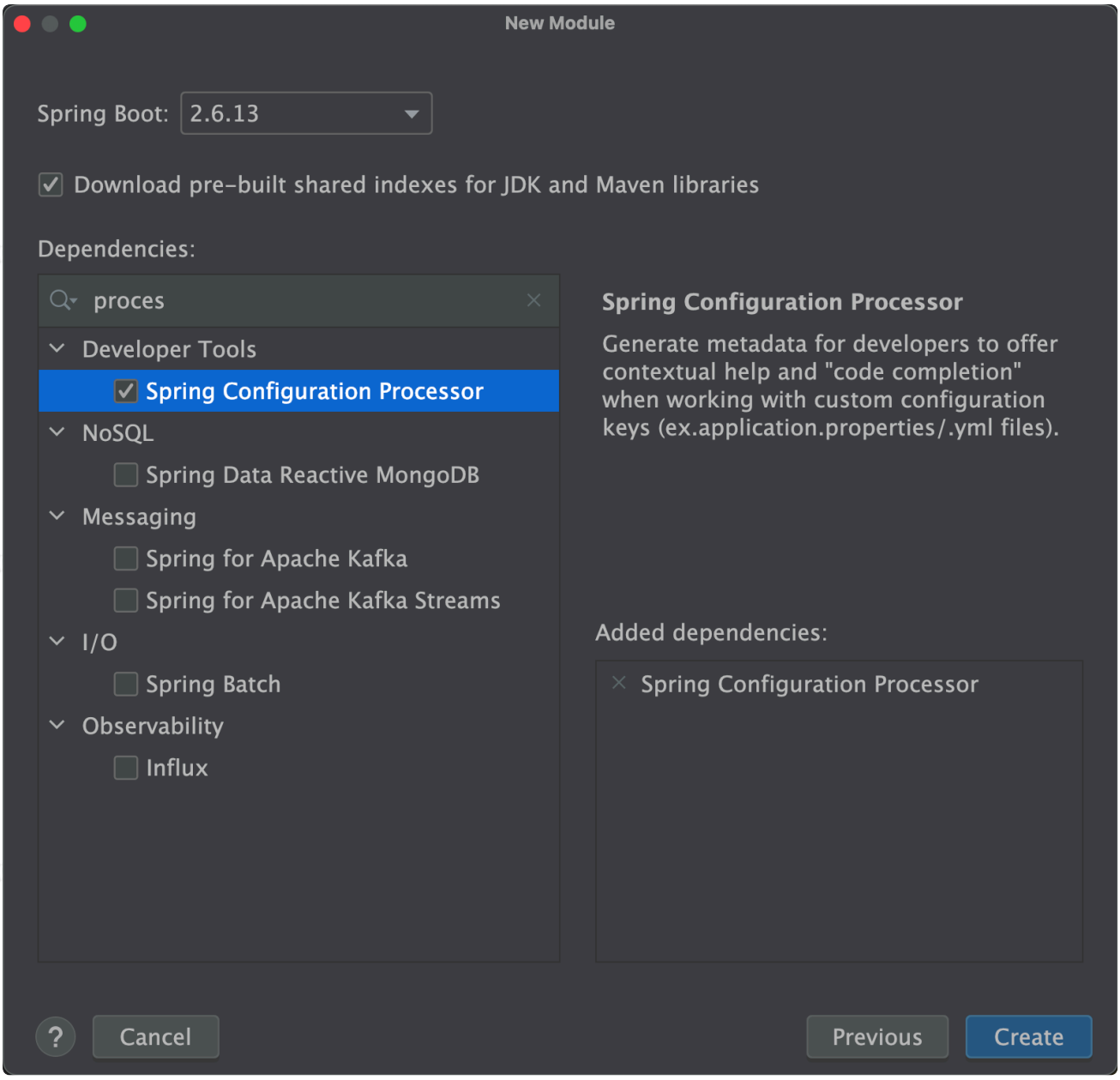


选择 Spring Initializr，将 Server URL 更改为 `start.aliyun.com`，然后创建一个名为 `yu-rpc-spring-boot-starter` 的模块，JDK 和 Java 版本选择 **>= 8** 即可。

如下图：



选择 Spring Boot 版本为 2.6，项目依赖如下：



创建好模块后，修改 pom.xml 文件，移除无用的插件代码：

```xml
<plugin>
    <groupId>org.springframework.boot</groupId>

    <artifactId>spring-boot-maven-plugin</artifactId>

    <version>${spring-boot.version}</version>

    <configuration>
        <mainClass>com.yupi.yurpc.springboot.starter.YuRpcSpringBootStarterAp

        <skip>true</skip>

    </configuration>

    <executions>
        <execution>
            <id>repackage</id>

            <goals>
                <goal>repackage</goal>

            </goals>

        </execution>

    </executions>

</plugin>

```

引入我们开发的 RPC 框架：

```xml
<dependency>
    <groupId>com.yupi</groupId>

    <artifactId>yu-rpc-core</artifactId>

    <version>1.0-SNAPSHOT</version>

</dependency>

```

至此，Spring Boot Starter 项目已经完成初始化。

2、定义注解

实现注解驱动的第一步是定义注解，要定义哪些注解呢？我们怎么知道应该定义哪些注解呢？

还是那句话，有样学样，可以参考知名框架 Dubbo 的注解。

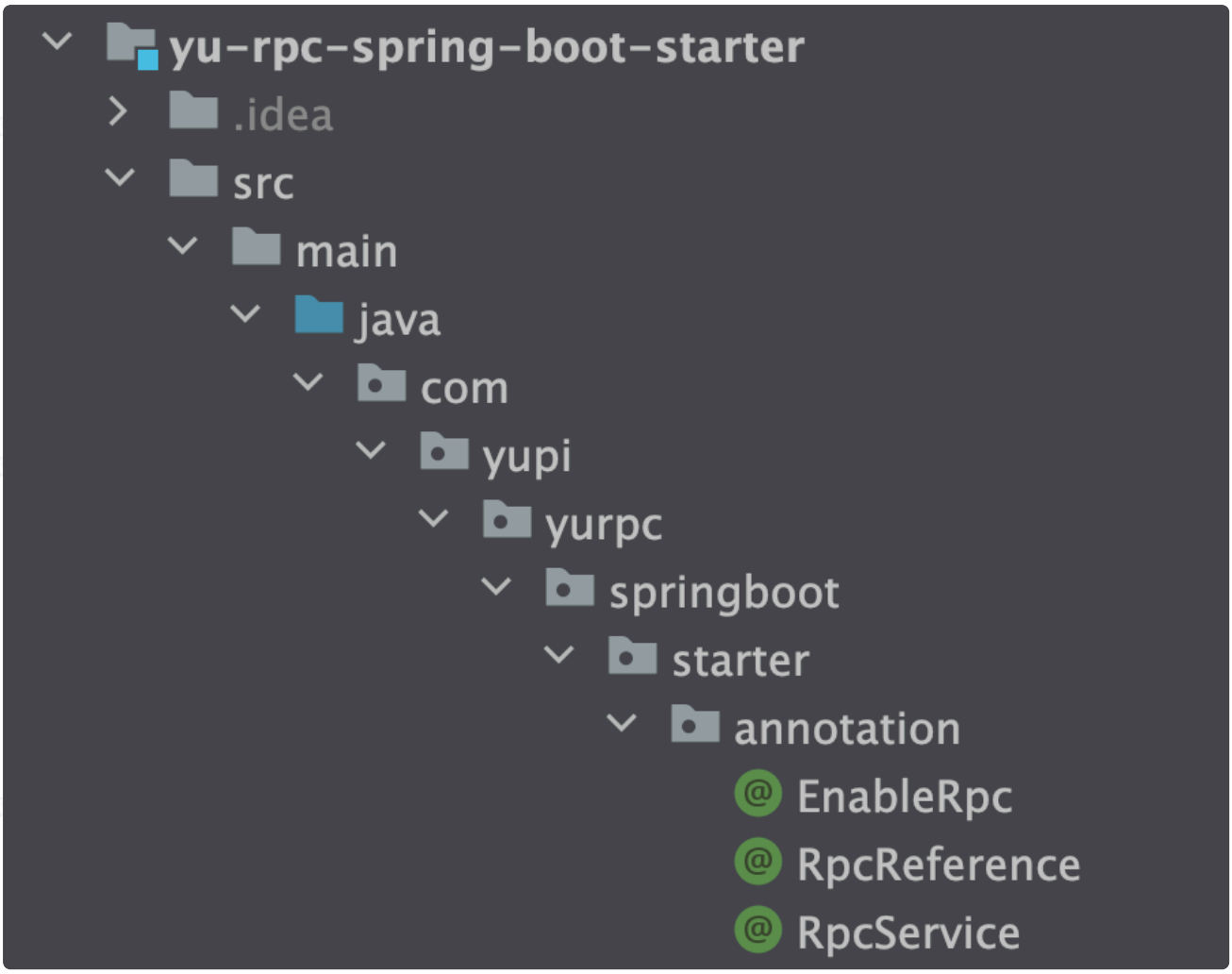
比如：

1. @EnableDubbo：在 Spring Boot 主应用类上使用，用于启用 Dubbo 功能。
2. @DubboComponentScan：在 Spring Boot 主应用类上使用，用于指定 Dubbo 组件扫描的包路径。
3. @DubboReference：在消费者中使用，用于声明 Dubbo 服务引用。
4. @DubboService：在提供者中使用，用于声明 Dubbo 服务。
5. @DubboMethod：在提供者和消费者中使用，用于配置 Dubbo 方法的参数、超时时间等。
6. @DubboTransported：在 Dubbo 提供者和消费者中使用，用于指定传输协议和参数，例如传输协议的类型、端口等。

当然，这些注解我们不需要全部用到，遵循最小可用化原则，我们只需要定义 3 个注解。

在 `yu-rpc-spring-boot-starter` 项目下新建 `annotation` 包，将所有注解代码放到该包下。

如下图：



1）@EnableRpc：用于全局标识项目需要引入 RPC 框架、执行初始化方法。

由于服务消费者和服务提供者初始化的模块不同，我们需要在 EnableRpc 注解中，指定是否需要启动服务器等属性。

代码如下：

```java
package com.yupi.yurpc.springboot.starter.annotation;

import com.yupi.yurpc.springboot.starter.bootstrap.RpcConsumerBootstrap;
import com.yupi.yurpc.springboot.starter.bootstrap.RpcInitBootstrap;
import com.yupi.yurpc.springboot.starter.bootstrap.RpcProviderBootstrap;
import org.springframework.context.annotation.Import;

import java.lang.annotation.*;

/**
 * 启用 Rpc 注解
 */
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
public @interface EnableRpc {

    /**
     * 需要启动 server
     *
     * @return
     */
    boolean needServer() default true;
}
```

当然，你也可以将 EnableRpc 注解拆分为两个注解（比如 EnableRpcProvider、EnableRpcConsumer），分别用于标识服务提供者和消费者，但可能存在模块重复初始化的可能性。

2）@RpcService：服务提供者注解，在需要注册和提供的服务类上使用。

RpcService 注解中，需要指定服务注册信息属性，比如服务接口实现类、版本号等（也可以包括服务名称）。

代码如下：

```java
package com.yupi.yurpc.springboot.starter.annotation;

import com.yupi.yurpc.constant.RpcConstant;
import org.springframework.stereotype.Component;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * 服务提供者注解（用于注册服务）
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>

 * @learn <a href="https://codefather.cn">程序员鱼皮的编程宝典</a>

 * @from <a href="https://yupi.icu">编程导航知识星球</a>

 */
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Component
public @interface RpcService {

    /**
     * 服务接口类
     */
    Class<?> interfaceClass() default void.class;

    /**
     * 版本
     */
    String serviceVersion() default RpcConstant.DEFAULT_SERVICE_VERSION;
}
```

3）@RpcReference：服务消费者注解，在需要注入服务代理对象的属性上使用，类似 Spring 中的 @Resource 注解。

RpcReference 注解中，需要指定调用服务相关的属性，比如服务接口类（可能存在多个接口）、版本号、负载均衡器、重试策略、是否 Mock 模拟调用等。

代码如下：

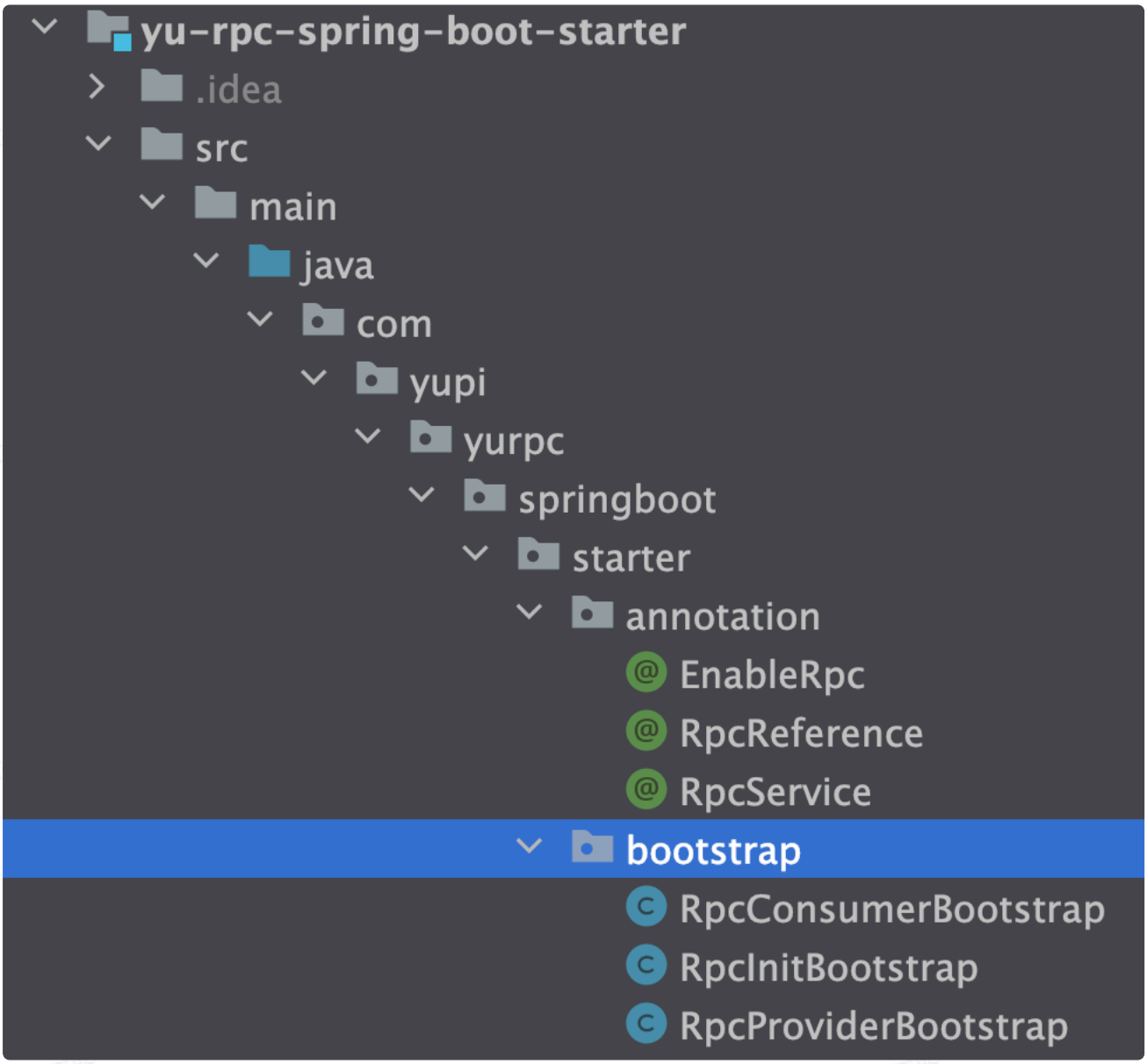
```java
package com.yupi.yurpc.springboot.starter.annotation;

import com.yupi.yurpc.constant.RpcConstant;
import com.yupi.yurpc.fault.retry.RetryStrategyKeys;
import com.yupi.yurpc.fault.tolerant.TolerantStrategyKeys;
import com.yupi.yurpc.loadbalancer.LoadBalancerKeys;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

/**
 * 服务消费者注解（用于注入服务）
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>

 * @learn <a href="https://codefather.cn">程序员鱼皮的编程宝典</a>

 * @from <a href="https://yupi.icu">编程导航知识星球</a>

 */
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface RpcReference {

    /**
     * 服务接口类
     */
    Class<?> interfaceClass() default void.class;

    /**
     * 版本
     */
    String serviceVersion() default RpcConstant.DEFAULT_SERVICE_VERSION;

    /**
     * 负载均衡器
     */
    String loadBalancer() default LoadBalancerKeys.ROUND_ROBIN;

    /**
     * 重试策略
     */
    String retryStrategy() default RetryStrategyKeys.NO;

    /**
     * 容错策略
     */
    String tolerantStrategy() default TolerantStrategyKeys.FAIL_FAST;
```

```
51
52      /**
53       * 模拟调用
54       */
55      boolean mock() default false;
56
57  }
```

3、注解驱动

在 starter 项目中新建 `bootstrap` 包，并且分别针对上面定义的 3 个注解新建启动类。

项目的目录结构如图：



1）Rpc 框架全局启动类 `RpcInitBootstrap` 。

我们的需求是，在 Spring 框架初始化时，获取 `@EnableRpc` 注解的属性，并初始化 RPC 框架。

怎么获取到注解的属性呢？

可以实现 Spring 的 `ImportBeanDefinitionRegistrar` 接口，并且在 `registerBeanDefinitions` 方法中，获取到项目的注解和注解属性。

完整代码如下：

```java
package com.yupi.yurpc.springboot.starter.bootstrap;

import com.yupi.yurpc.RpcApplication;
import com.yupi.yurpc.config.RpcConfig;
import com.yupi.yurpc.server.tcp.VertxTcpServer;
import com.yupi.yurpc.springboot.starter.annotation.EnableRpc;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.support.BeanDefinitionRegistry;
import org.springframework.context.annotation.ImportBeanDefinitionRegistrar;
import org.springframework.core.type.AnnotationMetadata;

/**
 * Rpc 框架启动
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>

 * @learn <a href="https://codefather.cn">程序员鱼皮的编程宝典</a>

 * @from <a href="https://yupi.icu">编程导航知识星球</a>

 */
@Slf4j
public class RpcInitBootstrap implements ImportBeanDefinitionRegistrar {

    /**
     * Spring 初始化时执行，初始化 RPC 框架
     *
     * @param importingClassMetadata
     * @param registry
     */
    @Override
    public void registerBeanDefinitions(AnnotationMetadata importingClassMeta
        // 获取 EnableRpc 注解的属性值
        boolean needServer = (boolean) importingClassMetadata.getAnnotationAt
                .get("needServer");

        // RPC 框架初始化（配置和注册中心）
        RpcApplication.init();

        // 全局配置
        final RpcConfig rpcConfig = RpcApplication.getRpcConfig();

        // 启动服务器
        if (needServer) {
            VertxTcpServer vertxTcpServer = new VertxTcpServer();
            vertxTcpServer.doStart(rpcConfig.getServerPort());
        } else {
            log.info("不启动 server");
        }

```

```
51        }
52    }
```

上述代码中，我们从 Spring 元信息中获取到了 EnableRpc 注解的 needServer 属性，并通过它来判断是否要启动服务器。

2）Rpc 服务提供者启动类 `RpcProviderBootstrap` 。

服务提供者启动类的作用是，获取到所有包含 `@RpcService` 注解的类，并且通过注解的属性和反射机制，获取到要注册的服务信息，并且完成服务注册。

怎么获取到所有包含 `@RpcService` 注解的类呢?

像前面设计方案中提到的，可以主动扫描包，也可以利用 Spring 的特性监听 Bean 的加载。

此处我们选择后者，实现更简单，而且能直接获取到服务提供者类的 Bean 对象。

只需要让启动类实现 `BeanPostProcessor` 接口的 `postProcessAfterInitialization` 方法，就可以在某个服务提供者 Bean 初始化后，执行注册服务等操作了。

完整代码如下：

```java
package com.yupi.yurpc.springboot.starter.bootstrap;

import com.yupi.yurpc.RpcApplication;
import com.yupi.yurpc.config.RegistryConfig;
import com.yupi.yurpc.config.RpcConfig;
import com.yupi.yurpc.model.ServiceMetaInfo;
import com.yupi.yurpc.registry.LocalRegistry;
import com.yupi.yurpc.registry.Registry;
import com.yupi.yurpc.registry.RegistryFactory;
import com.yupi.yurpc.springboot.starter.annotation.RpcService;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;

/**
 * Rpc 服务提供者启动
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>

 * @learn <a href="https://codefather.cn">程序员鱼皮的编程宝典</a>

 * @from <a href="https://yupi.icu">编程导航知识星球</a>

 */
@Slf4j
public class RpcProviderBootstrap implements BeanPostProcessor {

    /**
     * Bean 初始化后执行，注册服务
     *
     * @param bean
     * @param beanName
     * @return
     * @throws BeansException
     */
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName
        Class<?> beanClass = bean.getClass();
        RpcService rpcService = beanClass.getAnnotation(RpcService.class);
        if (rpcService != null) {
            // 需要注册服务
            // 1. 获取服务基本信息
            Class<?> interfaceClass = rpcService.interfaceClass();
            // 默认值处理
            if (interfaceClass == void.class) {
                interfaceClass = beanClass.getInterfaces()[0];
            }
            String serviceName = interfaceClass.getName();
            String serviceVersion = rpcService.serviceVersion();
            // 2. 注册服务
```

```
51                  // 本地注册
52                  LocalRegistry.register(serviceName, beanClass);
53
54                  // 全局配置
55                  final RpcConfig rpcConfig = RpcApplication.getRpcConfig();
56                  // 注册服务到注册中心
57                  RegistryConfig registryConfig = rpcConfig.getRegistryConfig();
58                  Registry registry = RegistryFactory.getInstance(registryConfig.ge
59                  ServiceMetaInfo serviceMetaInfo = new ServiceMetaInfo();
60                  serviceMetaInfo.setServiceName(serviceName);
61                  serviceMetaInfo.setServiceVersion(serviceVersion);
62                  serviceMetaInfo.setServiceHost(rpcConfig.getServerHost());
63                  serviceMetaInfo.setServicePort(rpcConfig.getServerPort());
64                  try {
65                      registry.register(serviceMetaInfo);
66                  } catch (Exception e) {
67                      throw new RuntimeException(serviceName + " 服务注册失败", e);
68                  }
69              }
70
71              return BeanPostProcessor.super.postProcessAfterInitialization(bean, b
72          }
```

其实上述代码中，绝大多数服务提供者初始化的代码都只需要从之前写好的启动类中复制粘贴，只不过换了一种参数获取方式罢了。

3）Rpc 服务消费者启动类 `RpcConsumerBootstrap` 。

和服务提供者启动类的实现方式类似，在 Bean 初始化后，通过反射获取到 Bean 的所有属性，如果属性包含 `@RpcReference` 注解，那么就为该属性动态生成代理对象并赋值。

完整代码如下：

```java
package com.yupi.yurpc.springboot.starter.bootstrap;

import com.yupi.yurpc.proxy.ServiceProxyFactory;
import com.yupi.yurpc.springboot.starter.annotation.RpcReference;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;

import java.lang.reflect.Field;

/**
 * Rpc 服务消费者启动
 *
 * @author <a href="https://github.com/liyupi">程序员鱼皮</a>

 * @learn <a href="https://codefather.cn">程序员鱼皮的编程宝典</a>

 * @from <a href="https://yupi.icu">编程导航知识星球</a>

 */
@Slf4j
public class RpcConsumerBootstrap implements BeanPostProcessor {

    /**
     * Bean 初始化后执行，注入服务
     *
     * @param bean
     * @param beanName
     * @return
     * @throws BeansException
     */
    @Override
    public Object postProcessAfterInitialization(Object bean, String beanName
        Class<?> beanClass = bean.getClass();
        // 遍历对象的所有属性
        Field[] declaredFields = beanClass.getDeclaredFields();
        for (Field field : declaredFields) {
            RpcReference rpcReference = field.getAnnotation(RpcReference.clas
            if (rpcReference != null) {
                // 为属性生成代理对象
                Class<?> interfaceClass = rpcReference.interfaceClass();
                if (interfaceClass == void.class) {
                    interfaceClass = field.getType();
                }
                field.setAccessible(true);
                Object proxyObject = ServiceProxyFactory.getProxy(interfaceCl
                try {
                    field.set(bean, proxyObject);
                    field.setAccessible(false);
                } catch (IllegalAccessException e) {
```

```
                    throw new RuntimeException("为字段注入代理对象失败", e);
                }
            }
        }
        return BeanPostProcessor.super.postProcessAfterInitialization(bean, b
    }

}
```

上述代码中，核心方法是 `beanClass.getDeclaredFields` ，用于获取类中的所有属性。看到这里的同学，必须要把反射的常用语法熟记于心了。

4）注册已编写的启动类。

最后，别忘了在 Spring 中加载我们已经编写好的启动类。

如何加载呢?

我们的需求是，仅在用户使用 `@EnableRpc` 注解时，才启动 RPC 框架。所以，可以通过给 EnableRpc 增加 `@Import` 注解，来注册我们自定义的启动类，实现灵活的可选加载。

修改后的 EnableRpc 注解代码如下：

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Import({RpcInitBootstrap.class, RpcProviderBootstrap.class, RpcConsumerBoots
public @interface EnableRpc {

    /**
     * 需要启动 server
     *
     * @return
     */
    boolean needServer() default true;
}
```

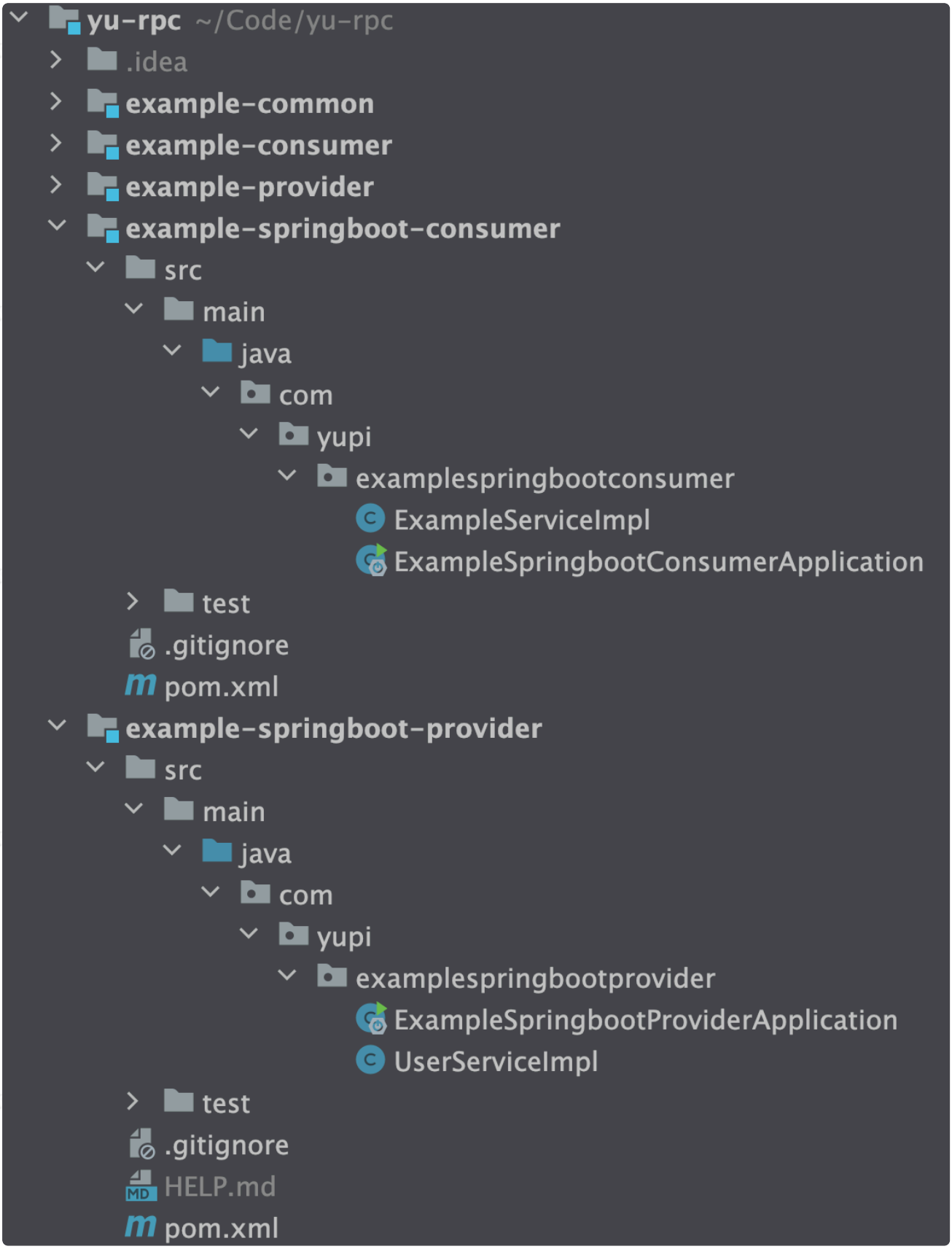
至此，一个基于注解驱动的 RPC 框架 Starter 开发完成。

四、测试

让我们使用 IDEA 新建 2 个使用 Spring Boot 2 框架的项目。

- 示例 Spring Boot 消费者：example-springboot-consumer
- 示例 Spring Boot 提供者：example-springboot-provider

项目的目录结构如下图：



每个项目都引入依赖：

```xml
    <dependency>
        <groupId>com.yupi</groupId>

        <artifactId>yu-rpc-spring-boot-starter</artifactId>

        <version>0.0.1-SNAPSHOT</version>

    </dependency>

    <dependency>
        <groupId>com.yupi</groupId>

        <artifactId>example-common</artifactId>

        <version>1.0-SNAPSHOT</version>

    </dependency>

```

1）示例服务提供者项目的入口类加上 `@EnableRpc` 注解，代码如下：

```java
package com.yupi.examplespringbootprovider;

import com.yupi.yurpc.springboot.starter.annotation.EnableRpc;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
@EnableRpc
public class ExampleSpringbootProviderApplication {

    public static void main(String[] args) {
        SpringApplication.run(ExampleSpringbootProviderApplication.class, arg
    }

}
```

服务提供者提供一个简单的服务，代码如下：

```java
package com.yupi.examplespringbootprovider;

import com.yupi.example.common.model.User;
import com.yupi.example.common.service.UserService;
import com.yupi.yurpc.springboot.starter.annotation.RpcService;
import org.springframework.stereotype.Service;

/**
 * 用户服务实现类
 *
 * @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>

 */
@Service
@RpcService
public class UserServiceImpl implements UserService {

    public User getUser(User user) {
        System.out.println("用户名：" + user.getName());
        return user;
    }
}
```

2）示例服务消费者的入口类加上 `@EnableRpc(needServer = false)` 注解，标识启动 RPC 框架，但不启动服务器。

代码如下：

```java
package com.yupi.examplespringbootconsumer;

import com.yupi.yurpc.springboot.starter.annotation.EnableRpc;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
@EnableRpc(needServer = false)
public class ExampleSpringbootConsumerApplication {

    public static void main(String[] args) {
        SpringApplication.run(ExampleSpringbootConsumerApplication.class, arg
    }

}
```

消费者编写一个 Spring 的 Bean，引入 UserService 属性并打上 `@RpcReference` 注解，表示需要使用远程服务提供者的服务。

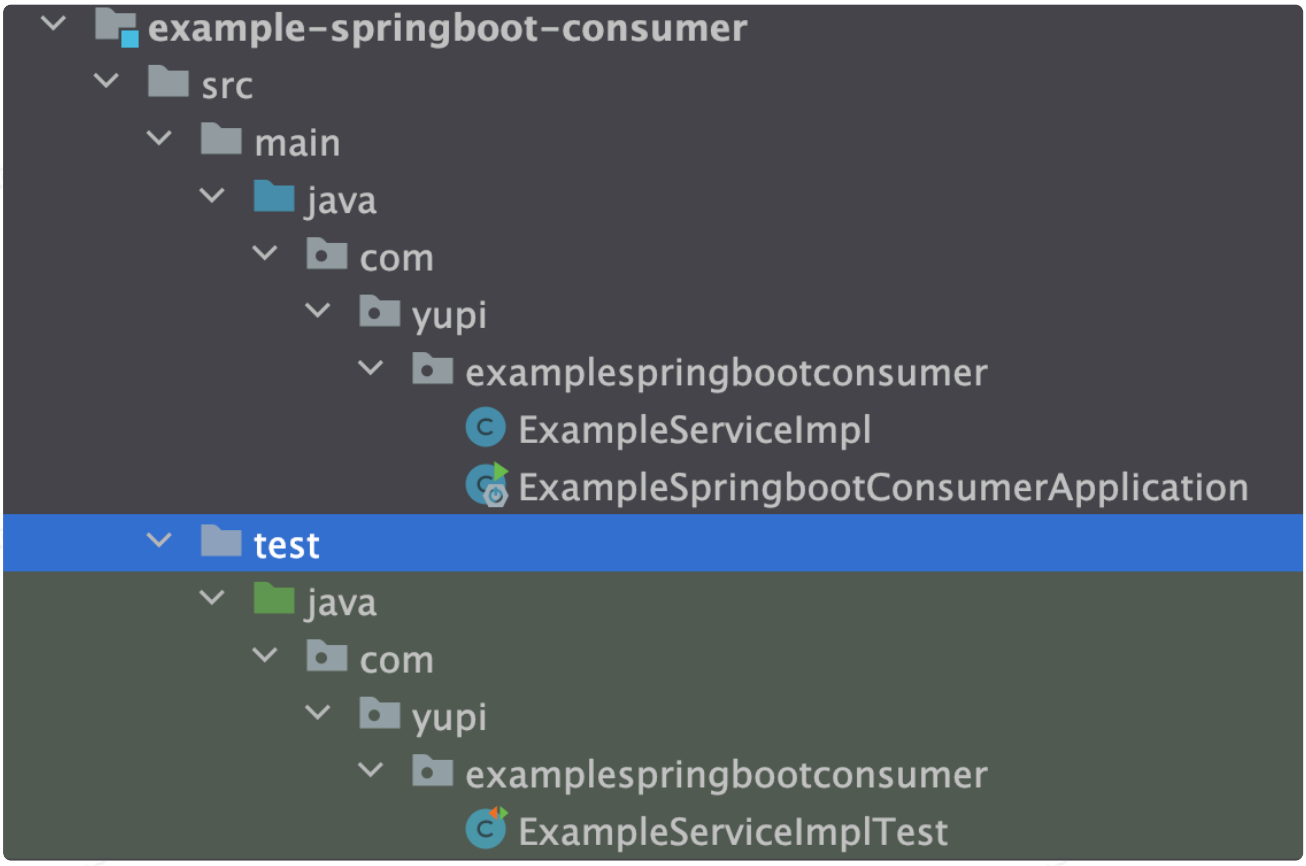
代码如下：

```
package com.yupi.examplespringbootconsumer;

import com.yupi.example.common.model.User;
import com.yupi.example.common.service.UserService;
import com.yupi.yurpc.springboot.starter.annotation.RpcReference;
import org.springframework.stereotype.Service;

@Service
public class ExampleServiceImpl {

    @RpcReference
    private UserService userService;

    public void test() {
        User user = new User();
        user.setName("yupi");
        User resultUser = userService.getUser(user);
        System.out.println(resultUser.getName());
    }

}
```

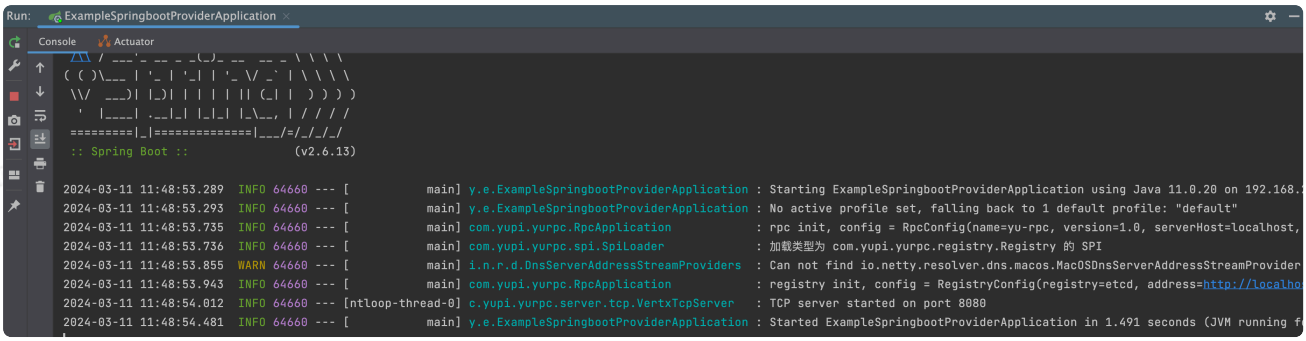
服务消费者编写单元测试，验证能否调用远程服务：

```java
package com.yupi.examplespringbootconsumer;

import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

import javax.annotation.Resource;

import static org.junit.jupiter.api.Assertions.*;

@SpringBootTest
class ExampleServiceImplTest {

    @Resource
    private ExampleServiceImpl exampleService;

    @Test
    void test1() {
        exampleService.test();
    }
}
```

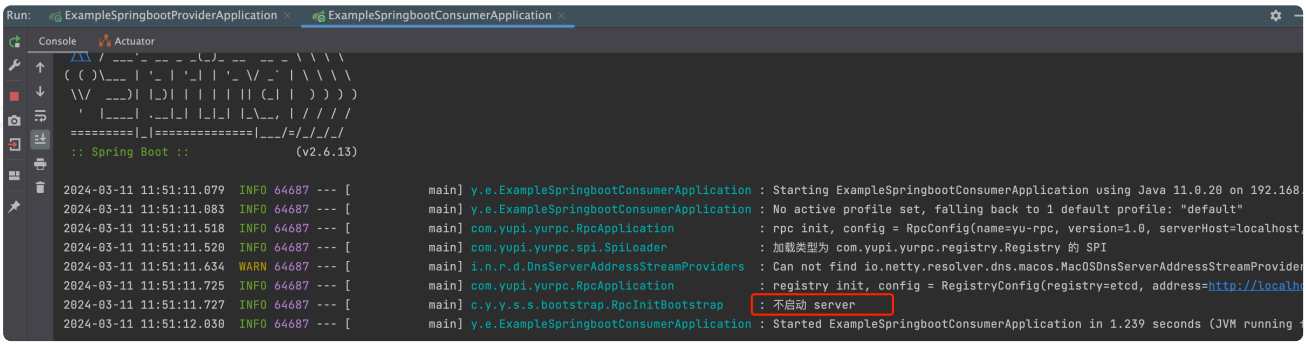
服务消费者的目录结构如图：



3）启动服务提供者入口类，如下图：



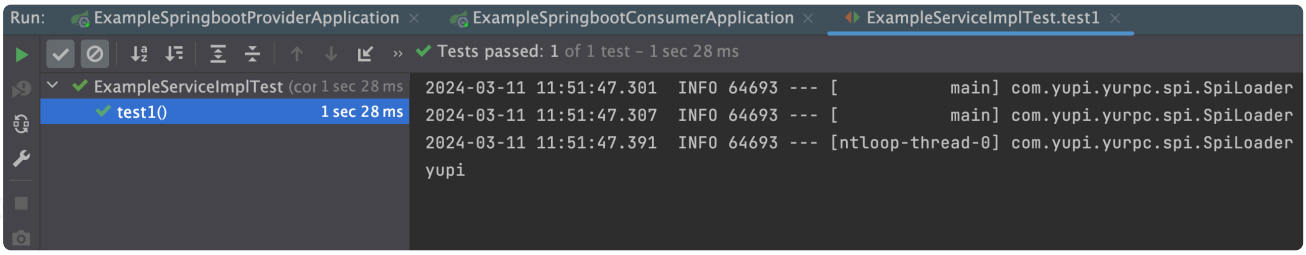
启动服务消费者的入口类，如下图：



可以看到 server 并没有启动，符合预期。

最后，执行服务消费者的单元测试，验证能否跑通整个流程。

如下图，调用成功：



服务提供者也收到了调用：