# 1、dubbo项目引入

dubbo 阿里开源的一个SOA服务治理框架，从目前来看把它称作是一个RPC远程调用框架更为贴切。单从RPC框架来说，功能较完善，支持多种传输和序列化方案。所以想必大家已经知道他的核心功能了：就是远程调用。

中文官网：

https://dubbo.gitbooks.io/dubbo-user-book/content/references/xml/dubbo-service.html

## 1.1 引入dubbo的三种方式

### 1.1 直接引入dubbo

实现步骤

* 创建服务端项目
  + 引入dubbo 依赖
  + 编写服务端代码
* 创建客户端项目
  + 引入dubbo 依赖
  + 编写客户端调用代码

dubbo 引入：

<dependency>

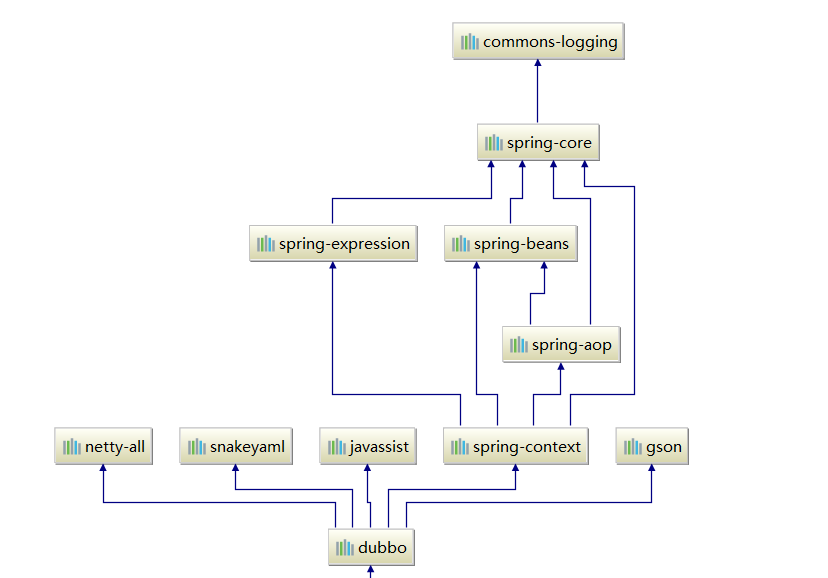
<groupId>org.apache.dubbo</groupId>

<artifactId>dubbo</artifactId>

<version>2.7.4.1</version>

</dependency>

dubbo 默认必填依懒：



服务端代码：

public void openServer(int port) {

ApplicationConfig config = new ApplicationConfig();

config.setName("simple-app");

ProtocolConfig protocolConfig=new ProtocolConfig();

protocolConfig.setName("dubbo");

protocolConfig.setPort(port);

protocolConfig.setThreads(20);

ServiceConfig<UserService> serviceConfig=new ServiceConfig();

serviceConfig.setApplication(config);

serviceConfig.setProtocol(protocolConfig);

serviceConfig.setRegistry(new RegistryConfig(RegistryConfig.NO\_AVAILABLE));

serviceConfig.setInterface(UserService.class);

serviceConfig.setRef(new UserServiceImpl());

serviceConfig.export();

}

客户端代码：

static String remoteUrl = "dubbo://127.0.0.1:12345/tuling.dubbo.server.UserService";

// 构建远程服务对象

public UserService buildRemoteService(String remoteUrl) {

ApplicationConfig application = new ApplicationConfig();

application.setName("young-app");

ReferenceConfig<UserService> referenceConfig = new ReferenceConfig<>();

referenceConfig.setApplication(application);

referenceConfig.setInterface(UserService.class);

referenceConfig.setUrl(remoteUrl);

UserService userService = referenceConfig.get();

return userService;

}

演示基于注册中心实现服集群：

* 修改服务端代码，添加multicast 注册中心。
* 修改客户端代码，添加multicast 注册中心。
* 观察 多个服务时，客户端如何调用。
* 观察 动态增减服务，客户端的调用。

# 服务端连接注册中心

serviceConfig.setRegistry(new RegistryConfig("multicast://224.1.1.1:2222"));

# 客户端连接注册中心

referenceConfig.setRegistry(new RegistryConfig("multicast://224.1.1.1:2222"));

#查看 基于UDP 占用的2222 端口

netstat -ano|findstr 2222

### 1.2 基于spring IOC 引入dubbo

在前面两个例子中 出现了,ApplicationConfig、ReferenceConfig、RegistryConfig、com.alibaba.dubbo.config.ServiceConfig等实例 ，很显然不需要每次调用的时候都去创建该实例那就需要一个IOC 容器去管理这些实例，spring 是一个很好的选择。

**提供者配置----------------------------------**

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd">

<!-- 提供方应用信息，用于计算依赖关系 -->

<dubbo:application name="simple-app" />

<!-- 使用multicast广播注册中心暴露服务地址 -->

<dubbo:registry address="multicast://224.5.6.7:1234" />

<!-- 用dubbo协议在20880端口暴露服务 ，当port设置为-1时代表自动分配端口号，此为生产时配置-->

<dubbo:protocol name="dubbo" port="20880" />

<!-- 声明需要暴露的服务接口 -->

<dubbo:service interface="tuling.dubbo.server.UserService" ref="userService" />

<!-- 和本地bean一样实现服务 -->

<bean id="userService" class="tuling.dubbo.server.impl.UserServiceImpl" />

</beans>

提供者服务暴露代码：

ApplicationContext context = new ClassPathXmlApplicationContext("/spring-provide.xml");

((ClassPathXmlApplicationContext) context).start();

System.in.read();

**消费者配置---------------------------------------**

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:dubbo="http://dubbo.apache.org/schema/dubbo"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd

http://dubbo.apache.org/schema/dubbo http://dubbo.apache.org/schema/dubbo/dubbo.xsd">

<dubbo:application name="young-app"/>

<dubbo:registry address="multicast://224.5.6.7:1234"/>

<dubbo:reference id="userService" interface="tuling.dubbo.server.UserService"/>

</beans>

消费者调用代码：

ApplicationContext context = new ClassPathXmlApplicationContext("/spring-consumer.xml");

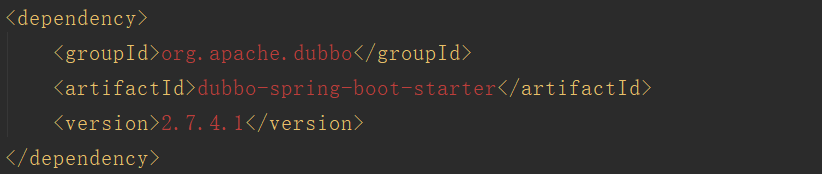
UserService userService = context.getBean(UserService.class);

UserVo u = userService.getUser(1111);

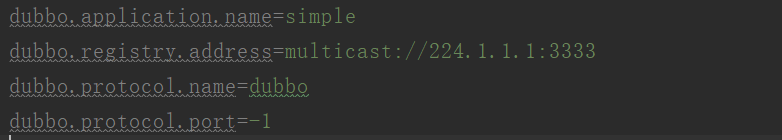
System.out.println(u);

### 1.3 spring boot方式引入dubbo

引入依赖



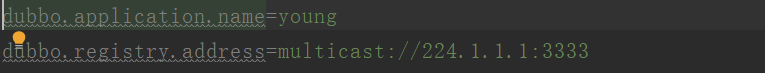
Server端配置



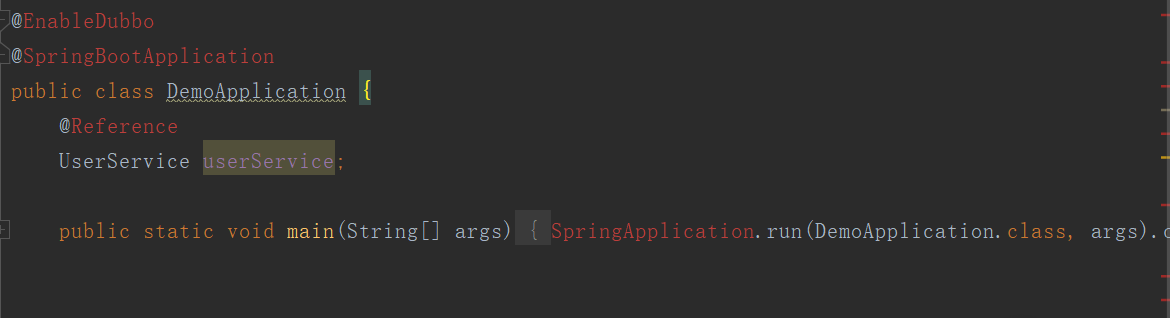
使用@EnableDubbo注解引入dubbo配置



客户端配置

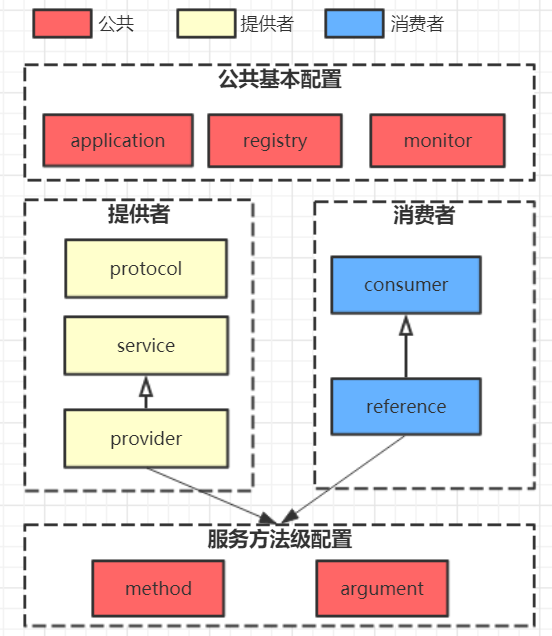


使用@Reference 注解注入RPC调用的类实例



## 1.2 dubbo的配置属性

|  |  |  |
| --- | --- | --- |
| **标签** | **用途** | **解释** |
| <dubbo:application/> | 公共 | 用于配置当前应用信息，不管该应用是提供者还是消费者 |
| <dubbo:registry/> | 公共 | 用于配置连接注册中心相关信息 |
| <dubbo:protocol/> | 服务 | 用于配置提供服务的协议信息，协议由提供方指定，消费方被动接受 |
| <dubbo:service/> | 服务 | 用于暴露一个服务，定义服务的元信息，一个服务可以用多个协议暴露，一个服务也可以注册到多个注册中心 |
| <dubbo:provider/> | 服务 | 当 ProtocolConfig 和 ServiceConfig 某属性没有配置时，采用此缺省值，可选 |
| <dubbo:consumer/> | 引用 | 当 ReferenceConfig 某属性没有配置时，采用此缺省值，可选 |
| <dubbo:reference/> | 引用 | 用于创建一个远程服务代理，一个引用可以指向多个注册中心 |
| <dubbo:method/> | 公共 | 用于 ServiceConfig 和 ReferenceConfig 指定方法级的配置信息 |
| <dubbo:argument/> | 公共 | 用于指定方法参数配置 |



### Application：用于配置应用的信息

Name：必填，当前应用名称，用于注册中心计算应用间依赖关系，注意：消费者和提供者应用名不要一样，此参数不是匹配条件，你当前项目叫什么名字就填什么，和提供者消费者角色无关，比如：kylin应用调用了morgan应用的服务，则kylin项目配成kylin，morgan项目配成morgan，可能kylin也提供其它服务给别人使用，但kylin项目永远配成kylin，这样注册中心将显示kylin依赖于morgan

Version：当前应用的版本

Owner：应用负责人，用于服务治理，请填写负责人公司邮箱前缀

Organization：组织名称(BU或部门)，用于注册中心区分服务来源，此配置项建议不要使用autoconfig，直接写死在配置中，如china,intl,itu,crm,asc,dw,aliexpress

Architecture：用于服务分层对应的架构。如，intl、china。不同的架构使用不同的分层。

Environment：应用环境，如：develop/test/product，不同环境使用不同的缺省值，以及作为只用于开发测试功能的限制条件

Complier：默认javassist，Java字节码编译器，用于动态类的生成，可选：jdk或javassist

Logger：slf4j默认值，日志输出方式，可选：slf4j,jcl,log4j,log4j2,jdk

### Registry：注册中心配置

id: string 注册中心引用BeanId，可以在<dubbo:service registry=””>或<dubbo:reference registry=””>中引用此Id

address：<host:port> 服务发现必填，注册中心服务器地址，如果地址没有端口缺省为9090，同一集群内的多个地址用逗号分隔，如：ip:port,ip:port，不同集群的注册中心，请配置多个<dubbo:registry>标签

protocol:默认dubbo协议，注册中心地址协议，支持dubbo, multicast, zookeeper, redis, consul(2.7.1), sofa(2.7.2), etcd(2.7.2), nacos(2.7.2)等协议

port：9090，注册中心缺省端口，当address没有带端口时使用此端口做为缺省值

username：登陆注册中心的用户名，如果注册中心不需要验证，则可以不填

password：注册中心密码，不需要可以不填

transport：默认netty，网络传输方式，可选mina，netty

timeout：默认5000毫秒，注册中心请求超时时间（毫秒）

session：60000，注册中心会话超时时间（毫秒），用于检测提供者非正常断线后的脏数据，比如用心跳检测的实现，此时间就是心跳间隔，不同注册中心实现不一样

file： 使用文件缓存注册中心地址列表及服务提供者列表，应用重启时将基于此文件恢复，注意：两个注册中心不能使用同一文件存储

wait：默认0，停止时等待通知完成的时间（毫秒）

check：布尔值，默认true，注册中心不存在时或者宕机是否报错

register：布尔值默认true，是否向此注册中心注册服务，如果设为false，将只订阅，不注册，即只调用注册中心的服务，而本身不注册上去不会被调用

subscribe：布尔值，默认为true，是否向此注册中心订阅服务，如果设为false，将只注册，不订阅

dynamic：布尔值，默认为true，服务是否动态注册，如果设为false，注册后将显示为disable状态，需人工启用，并且服务提供者停止时，也不会自动取消注册，需人工禁用。

Group：默认为dubbo分组，服务注册分组，跨组的服务不会相互影响，也无法相互调用，适用于环境隔离。

Simplified：布尔值，默认为false，注册到注册中心的URL是否采用精简模式的（与低版本兼容）

Extra-keys：在simplified=true时，extraKeys允许你在默认参数外将额外的key放到URL中，格式："interface,key1,key2"。

### Provider：服务提供者，可以为service、protocol提供默认配置，类似于模板

Id：默认值dubbo，协议BeanId，可以在<dubbo:service proivder="">中引用此ID，如果service中没有配置则使用第一个

Protocol：默认dubbo，协议

Host：服务主机名，多网卡选择或指定VIP及域名时使用，为空则自动查找本机IP，建议不要配置，让Dubbo自动获取本机IP

Threads：服务线程池大小（固定大小），默认200

Payload：默认8M，请求及响应数据包大小限制，单位：字节

Path：提供者上下文路径，为服务path的前缀

Server：dubbo协议缺省为netty，http协议缺省为servlet，协议的服务器端实现类型，比如：dubbo协议的mina,netty等，http协议的jetty,servlet等

Client：dubbo协议的默认值为netty，协议的客户端实现类型，比如：dubbo协议的mina,netty等

Codec：协议编码方式，默认dubbo

Serialization：dubbo协议缺省为hessian2，rmi协议缺省为java，http协议缺省为json，协议序列化方式，当协议支持多种序列化方式时使用，比如：dubbo协议的dubbo,hessian2,java,compactedjava，以及http协议的json,xml等

Default：默认false，是否为缺省协议，用于多协议

Filter：服务提供方远程调用过程拦截器名称，多个名称用逗号分隔

Listener：服务提供方导出服务监听器名称，多个名称用逗号分隔

Threadpool：线程池，默认固定线程池线程池类型，可选：fixed/cached/limit(2.5.3以上)/eager(2.6.x以上)

Accepts：默认0（0应该是不限制），服务提供者最大可接受连接数

Version：默认值0.0.0，服务版本，建议使用两位数字版本，如：1.0，通常在接口不兼容时版本号才需要升级

Group：服务分组，当一个接口有多个实现，可以用分组区分

Delay：默认值0，延迟注册服务时间(毫秒)- ，设为-1时，表示延迟到Spring容器初始化完成时暴露服务

Timeout：远程服务调用超时时间

Retries：默认值2，远程服务调用重试次数，不包括第一次调用，不需要重试请设为0

Connections：默认0，对每个提供者的最大连接数，rmi、http、hessian等短连接协议表示限制连接数，dubbo等长连接协表示建立的长连接个数，如果不配置则所有接口共用单一服务者与消费者的单一通道

Loadbalance：默认random，负载均衡策略，可选值：random,roundrobin,leastactive，分别表示：随机，轮询，最少活跃调用

Async：默认false，是否缺省异步执行，不可靠异步，只是忽略返回值，不阻塞执行线程

Stub：默认false，设为true，表示使用缺省代理类名，即：接口名 + Local后缀。

Mock：设为true，表示使用缺省Mock类名，即：接口名 + Mock后缀

Token：默认false，令牌验证，为空表示不开启，如果为true，表示随机生成动态令牌

Registry：默认向所有与注册中心注册，向指定注册中心注册，在多个注册中心时使用，值为<dubbo:registry>的id属性，多个注册中心ID用逗号分隔，如果不想将该服务注册到任何registry，可将值设为N/A

Dynamic：默认true，服务是否动态注册，如果设为false，注册后将显示后disable状态，需人工启用，并且服务提供者停止时，也不会自动取消册，需人工禁用。

Accesslog：默认false，值既可以为布尔值也可以是字符串，设为true，将向logger中输出访问日志，也可填写访问日志文件路径，直接把访问日志输出到指定文件

Owner：服务负责人，用于服务治理，请填写负责人公司邮箱前缀

Document：服务文档URL

Weight：服务权重

Excutes：默认0，服务提供者每服务每方法最大可并行执行请求数

Actives：默认0，每服务消费者每服务每方法最大并发调用数

Proxy：默认javassist，生成动态代理方式，可选：jdk/javassist

Cluster：默认failover，集群方式，可选：failover/failfast/failsafe/failback/forking

Deprecated：默认false，服务是否过时，如果设为true，消费方引用时将打印服务过时警告error日志

Queues：默认0，线程池队列大小，当线程池满时，排队等待执行的队列大小，建议不要设置，当线程池满时应立即失败，重试其它服务提供机器，而不是排队，除非有特殊需求。

Charset：默认utf-8，序列化编码

Buffer：默认8192，网络读写缓冲区大小

Iothreads：默认CPU+1，IO线程池，接收网络读写中断，以及序列化和反序列化，不处理业务，业务线程池参见threads配置，此线程池和CPU相关，不建议配置。

Talnet：所支持的telnet命令，多个命令用逗号分隔

Layer：服务提供者所在的分层。如：biz、dao、intl:web、china:acton。

<dubbo:service>:contextpath,服务治理

### Consumer：消费端，reference的缺省配置，模板

Timeout：默认1000毫秒，远程服务调用超时时间

Retries：默认2，远程服务调用重试次数，不包括第一次调用，不需要重试请设为0,仅在cluster为failback/failover时有效

Connections：默认0，对每个提供者的最大连接数，rmi、http、hessian等短连接协议表示限制连接数，dubbo等长连接协表示建立的长连接个数，如果不配置则所有接口共用单一服务者与消费者的单一通道

Loadbalance：默认random，负载均衡策略，可选值：random,roundrobin,leastactive，分别表示：随机，轮询，最少活跃调用

Async：默认false，是否缺省异步执行，不可靠异步，只是忽略返回值，不阻塞执行线程

Generic：默认false，是否缺省泛化接口，如果为泛化接口，将返回GenericService

Check：默认true，启动时检查提供者是否存在，true报错，false忽略

Proxy：默认Javassist，生成动态代理的方式，可选择jdk、Javassist

Owner：调用服务负责人，用于服务治理，请填写负责人公司邮箱前缀

Actives：默认0，每服务消费者每服务每方法最大并发调用数

Cluster：默认failover，集群方式，可选：failover/failfast/failsafe/failback/forking

Filter：服务消费方远程调用过程拦截器名称，多个名称用逗号分隔

Listener：服务消费方引用服务监听器名称，多个名称用逗号分隔

Registry：默认向所有的注册中心注册，向指定注册中心注册，在多个注册中心时使用，值为<dubbo:registry>的id属性，多个注册中心ID用逗号分隔，如果不想将该服务注册到任何registry，可将值设为N/A

## 1.3 配置中的套路

### 1.1 服务端配置客户端使用

其参数传递机制是 服务端所有配置都会封装到URL参数，在通过注册中心传递到客户端

如retries/riː'traɪ/(重试次数)、async/əˈsɪŋk/（是否异步）、loadbalance(负载均衡)。。。timeout等。

Dubbo中可以通过 <dubbo:provider> 来实现服务端缺省配置。它可以同时为 <dubbo:service> 和 <dubbo:protocol> 两个标签提供缺省配置。如：

#相当于每个服务提供者设置了超时时间 和重试次数

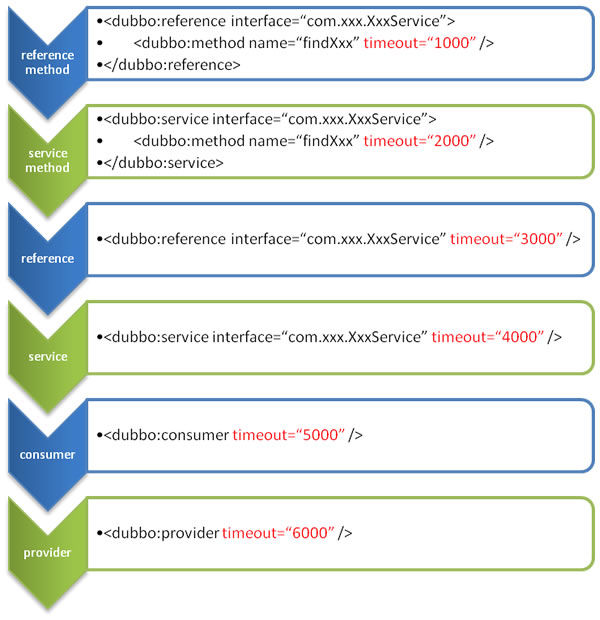
<dubbo:provider timeout="2000" retries="2"></dubbo:provider>

同样客户端也有缺省配置标签：<dubbo:consumer>，这些缺省设置可以配置多个 通过 <dubbo:service provider="providerId"> ,**如果没指定就用第一个**。

### 1.2<dubbo:provider>与<dubbo:service> ，<dubbo:consumer>与<dubbo:reference>傻傻分不清楚

Provider是服务端服务提供者，service服务端服务，consumer客户端消费者，reference客户端的服务引用

服务端与客户端配置的优先级，比如客户端服务端都能配置超时时间共有6处可以配置，则哪个会生效,下图优先级由上到下逐级递减



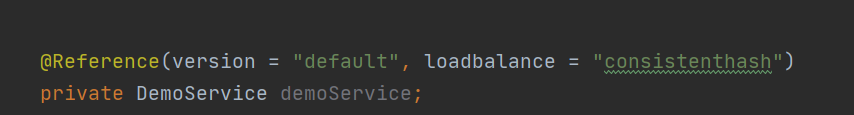
## 1.4 dubbo的高级应用

https://www.yuque.com/books/share/f2394ae6-381b-4f44-819e-c231b39c1497（密码：kyys）

### 1、负载均衡

Random随机，RoundRobin轮询，LeastActive最少活跃数，ConsistentHash一致性hash

默认使用随机策略



这其中⽐较难理解的就是最少活跃调⽤数是如何进⾏统计的？

讲道理，最少活跃数应该是在服务提供者端进⾏统计的，服务提供者统计有多少个请求正在执⾏中。

但在Dubbo中，就是不讲道理，它是在消费端进⾏统计的，为什么能在消费端进⾏统计？

逻辑是这样的：

1. 消费者会缓存所调⽤服务的所有提供者，⽐如记为p1、p2、p3三个服务提供者，每个提供者内都有⼀个属性记为active，默认位0

2. 消费者在调⽤次服务时，如果负载均衡策略是leastactive

3. 消费者端会判断缓存的所有服务提供者的active，选择最⼩的，如果都相同，则随机

4. 选出某⼀个服务提供者后，假设位p2，Dubbo就会对p2.active+1

5. 然后真正发出请求调⽤该服务

6. 消费端收到响应结果后，对p2.active-1

### 2、服务超时

在服务提供者和服务消费者上都可以配置服务超时时间，这两者是不⼀样的。

消费者调⽤⼀个服务，分为三步：

1. 消费者发送请求（⽹络传输）

2. 服务端执⾏服务

3. 服务端返回响应（⽹络传输）

如果在服务端和消费端只在其中⼀⽅配置了timeout，那么没有歧义，表示消费端调⽤服务的超时时间，消费端如果超过时间还没有收到响应结果，则消费端会抛超时异常，但，服务端不会抛异常，服务端在执⾏

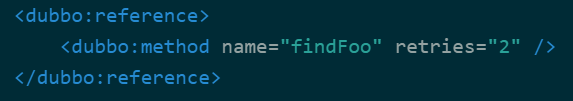
### 3集群容错

Dubbo调用失败后会有多种容错策略

Failover Cluster：

失败自动切换，当出现失败，重试其它服务器。通常用于读操作，但重试会带来更长延迟。可通过 retries="2" 来设置重试次数(不含第一次)。





Failfast Cluster：

快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录。

Failsafe Cluster

失败安全，出现异常时，直接忽略。通常用于写入审计日志等操作。

Failback Cluster：

失败自动恢复，后台记录失败请求，定时重发。通常用于消息通知操作。

Forking Cluster

并行调用多个服务器，只要一个成功即返回。通常用于实时性要求较高的读操作，但需要浪费更多服务资源。可通过 forks="2" 来设置最大并行数。

Broadcast Cluster：

广播调用所有提供者，逐个调用，任意一台报错则报错。通常用于通知所有提供者更新缓存或日志等本地资源信息。

### 4、服务降级

服务降级表示：服务消费者在调⽤某个服务提供者时，如果该服务提供者报错了，所采取的措施。

集群容错和服务降级的区别在于：

1. 集群容错是整个集群范围内的容错

2. 服务降级是单个服务提供者的⾃身容错

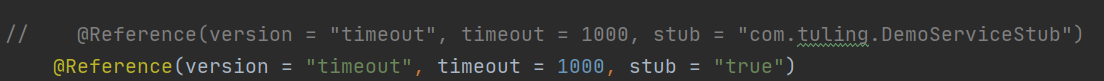
mock=force:return+null 表示消费方对该服务的方法调用都直接返回 null 值，不发起远程调用。用来屏蔽不重要服务不可用时对调用方的影响。

 mock=fail:return+null 表示消费方对该服务的方法调用在失败后，再返回 null 值，不抛异常。用来容忍不重要服务不稳定时对调用方的影响。

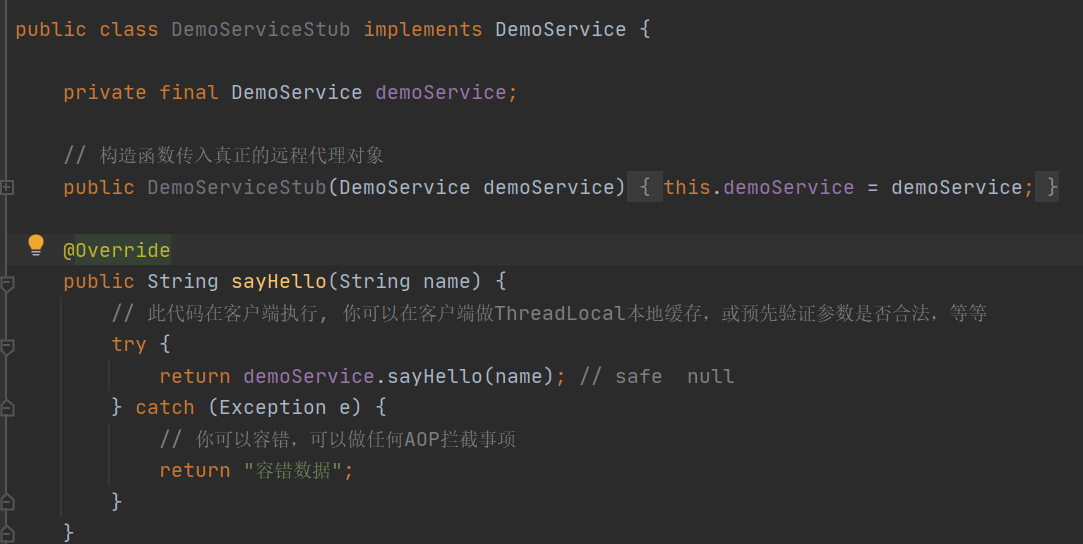


### 5、本地存根

远程服务后，客户端通常只剩下接口，而实现全在服务器端，但提供方有些时候想在客户端也执行部分逻辑，比如：做 ThreadLocal 缓存，提前验证参数，调用失败后伪造容错数据等等，此时就需要在 API 中带上 Stub，客户端生成 Proxy 实例，会把 Proxy 通过构造函数传给 Stub [1](https://dubbo.apache.org/zh/docs/v2.7/user/examples/local-stub/" \l "fn:1)，然后把 Stub 暴露给用户，Stub 可以决定要不要去调 Proxy。

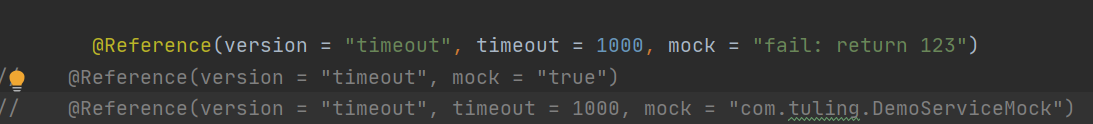


客户端配置本地存根有两种方式，当配置为stub=“true”时会默认去找接口后面加上Stub的实现类接口



### 6、本地伪装

本地伪装就是Mock，Dubbo中Mock的功能相对于本地存根更简单⼀点，Mock其实就是Dubbo中的服务容错的解决⽅案



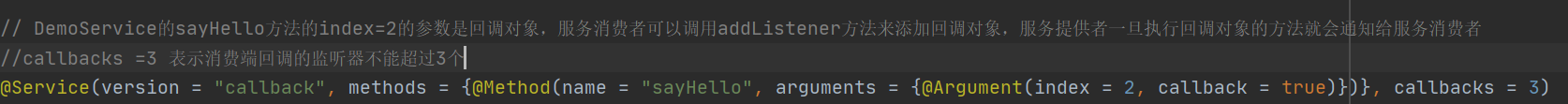
在 2.6.6 以上的版本，可以开始在 Spring XML 配置文件中使用 fail: 和 force:。force: 代表强制使用 Mock 行为，在这种情况下不会走远程调用。fail: 与默认行为一致，只有当远程调用发生错误时才使用 Mock 行为。force: 和 fail: 都支持与 throw 或者 return 组合使用。

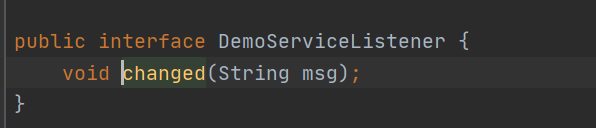
### 7、参数回调

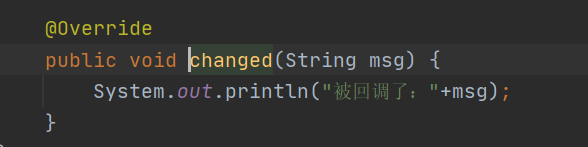
⾸先，如果当前服务⽀持参数回调，意思就是：对于某个服务接⼝中的某个⽅法，如果想⽀持消费者在调⽤这个⽅法时能设置回调逻辑，那么该⽅法就需要提供⼀个⼊参⽤来表示回调逻辑。

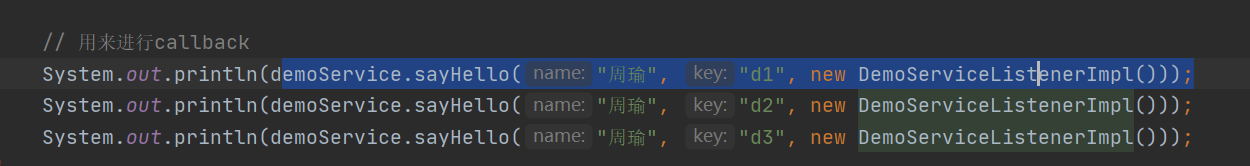
因为Dubbo协议是基于⻓连接的，所以消费端在两次调⽤同⼀个⽅法时想指定不同的回调逻辑，那么就需要在调⽤时在指定⼀定key进⾏区分。

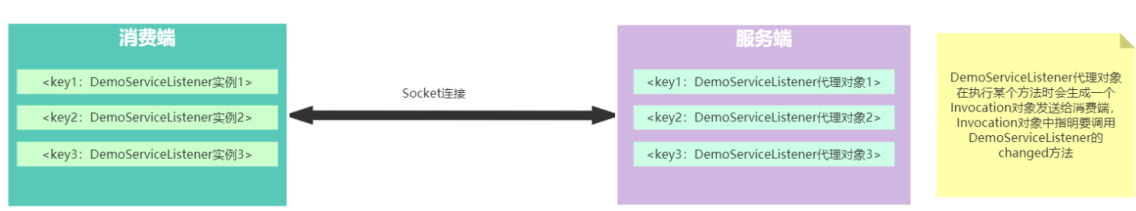
参数回调方式与调用本地 callback 或 listener 相同，只需要在 Spring 的配置文件中声明哪个参数是 callback 类型即可。Dubbo 将基于长连接生成反向代理，这样就可以从服务器端调用客户端逻辑





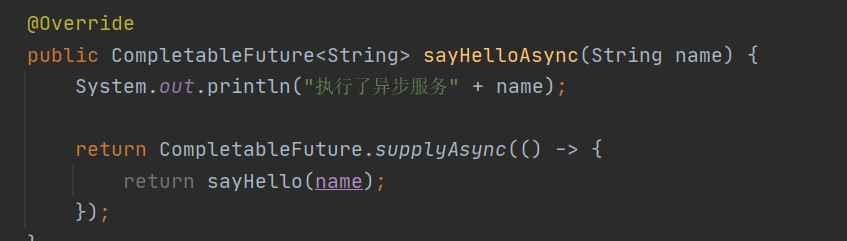




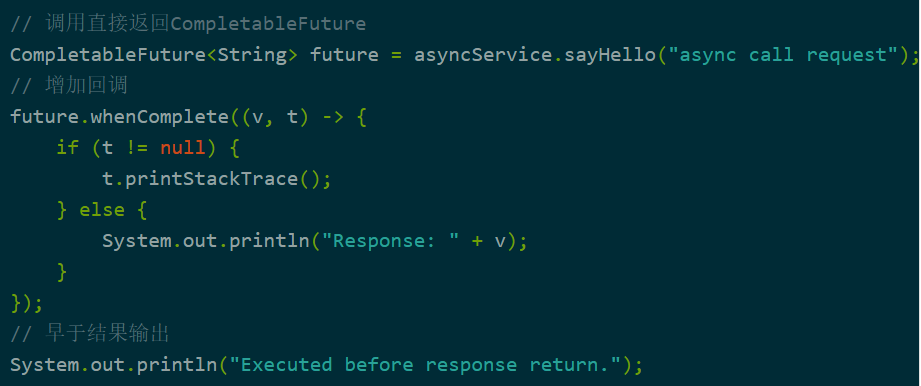


### 8、异步调⽤

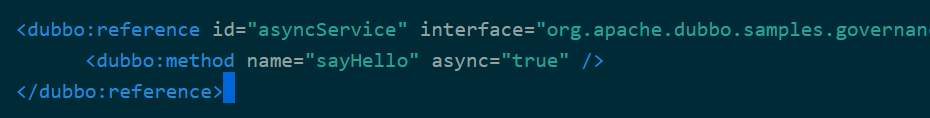
异步调用的服务提供方



消费者调用有多种方式

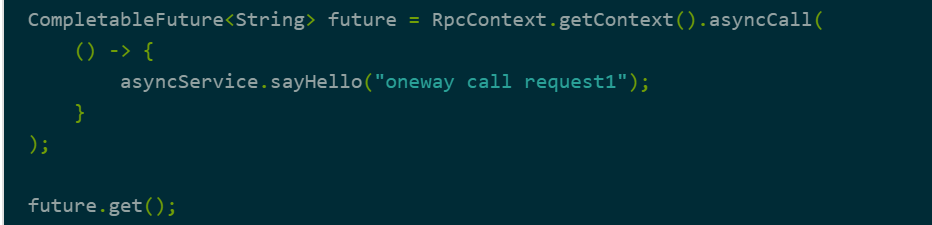


第二种方式





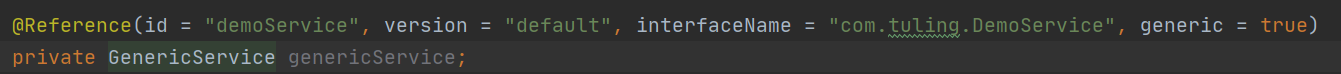
或者



### 9、泛化调用

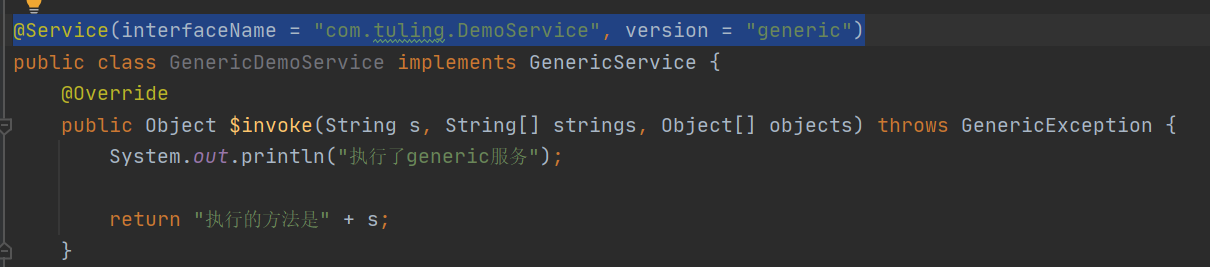
泛化调⽤可以⽤来做服务测试。

在Dubbo中，如果某个服务想要⽀持泛化调⽤，就可以将该服务的generic属性设置为true，那对于服务消费者来说，就可以不⽤依赖该服务的接⼝，直接利⽤GenericService接⼝来进⾏服务调⽤。



该实例调用的方法是demoService default版本的方法

### 10、泛化服务

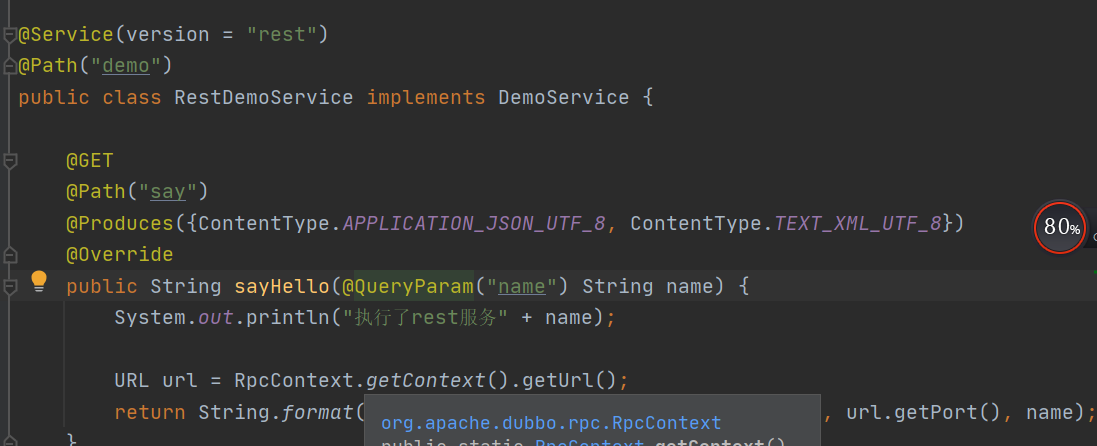


泛化服务是一个正常的服务实例，消费端可以引入，但是调用的时候不会调用demoService实现类，而是直接调用invoke方法，主要用于开发测试，在服务提供者没有开发完成时用于自身测试

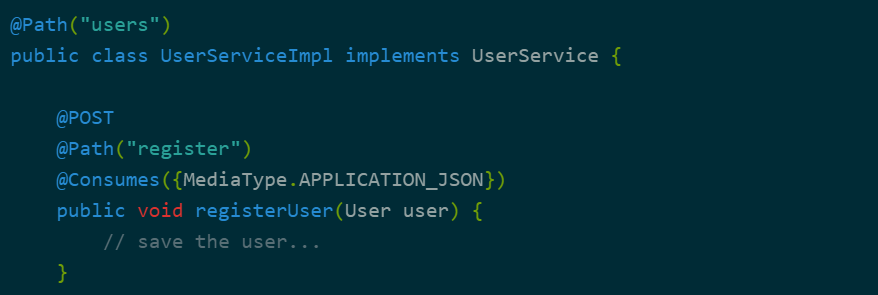
### 11、rest支持

注意Dubbo的REST也是Dubbo所⽀持的⼀种协议。

当我们⽤Dubbo提供了⼀个服务后，如果消费者没有使⽤Dubbo也想调⽤服务，那么这个时候我们就可以让我们的服务⽀持REST协议，这样消费者就可以通过REST形式调⽤我们的服务了。



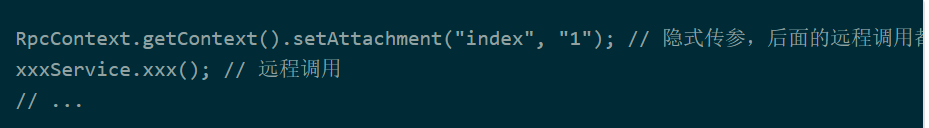
Path注解表明的访问路径，produces注解表示将返回数据转换成指定格式

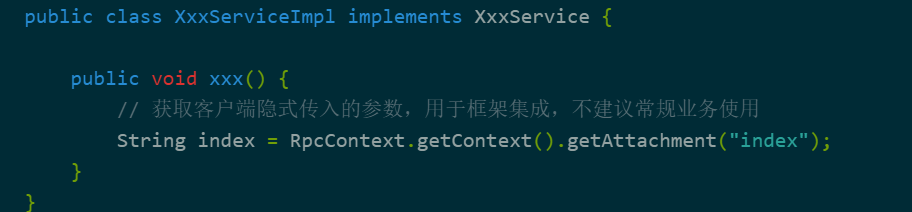


@Consumes({MediaType.APPLICATION\_JSON})：指定registerUser()接收JSON格式的数据。REST框架会自动将JSON数据反序列化为User对象

### 12、隐式传参

可以通过 RpcContext 上的 setAttachment 和 getAttachment 在服务消费方和提供方之间进行参数的隐式传递。





**消费方：**

在AbstractClusterInvoker类的invoke方法中会获取隐士参数，然后将隐式参数赋值给

RpcInvocation，可用于选择调用的具体服务

通过开发cluster的Wrapper包装类，在里面添加隐士参数

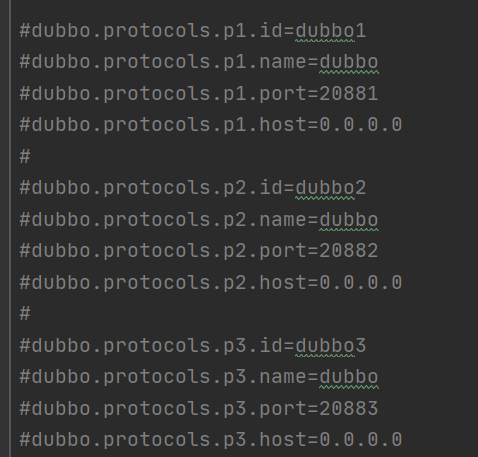
研究RpcContext的初始化过程添加隐士参数

**提供方：**

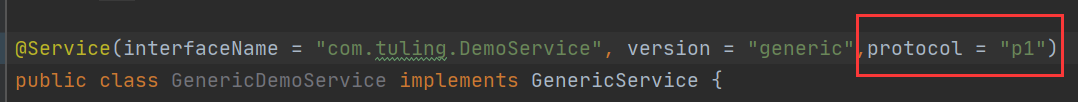
### 13、admin管理台动态配置与路由规则

注意动态配置修改的是服务参数，并不能修改服务的协议、IP、PORT、VERSION、GROUP，因为这5个信息是服务的标识信息，是服务的身份证号，是不能修改的。

### 14、多协议同时使用



可以同时定义多种协议，dubbo在暴露服务的时候会为每种协议均暴露服务，可以在service中指定使用哪种服务



## 1.5 dubbo提供的协议

### 1、dubbo 协议 (默认)

1、dubbo 缺省协议 采用单一长连接和NIO异步通讯，适合于小数据量大并发的服务调用，以及服务消费者机器数远大于服务提供者机器数的情况  
2、不适合传送大数据量的服务，比如传文件，传视频等，除非请求量很低。

可以设置消费者与服务提供者的连接个数，默认0，因dubbo协议采用单一长连接，假设网络为千兆网卡(1024Mbit=128MByte)，根据测试经验数据每条连接最多只能压满7MByte(不同的环境可能不一样，供参考)，理论上1个服务提供者需要20个服务消费者才能压满网卡

<dubbo:protocol name="dubbo" connections="2" />

<dubbo:service connections=”0”>或<dubbo:reference connections=”0”>

设置服务提供者最大的连接个数

<dubbo:protocol name="dubbo" accepts="1000" />

为什么采用异步单一长连接？

因为服务的现状大都是服务提供者少，通常只有几台机器，而服务的消费者多，可能整个网站都在访问该服务，比如Morgan的提供者只有6台提供者，却有上百台消费者，每天有1.5亿次调用，如果采用常规的hessian服务，服务提供者很容易就被压跨，通过单一连接，保证单一消费者不会压死提供者，长连接，减少连接握手验证等，并使用异步IO，复用线程池，防止C10K问题。

### 2、rmi协议

1、RMI协议采用JDK标准的java.rmi.\*实现，采用阻塞式短连接和JDK标准序列化方式 。

使用TCP传输协议，同步传输

如果正在使用RMI提供服务给外部访问（公司内网环境应该不会有攻击风险），同时应用里依赖了老的common-collections包（dubbo不会依赖这个包，请排查自己的应用有没有使用）的情况下，存在反序列化安全风险。

### 3、hessian 协议

Hessian 协议用于集成 Hessian 的服务，Hessian底层采用 Http 通讯，Hessian序列化，多连接，同步传输，采用 Servlet 暴露服务，Dubbo 缺省内嵌 Jetty 作为服务器实现。

Dubbo 的 Hessian 协议可以和原生 Hessian 服务互操作，即：

提供者用 Dubbo 的 Hessian 协议暴露服务，消费者直接用标准 Hessian 接口调用

或者提供方用标准 Hessian 暴露服务，消费方用 Dubbo 的 Hessian 协议调用。

Hessian 是 Caucho 开源的一个 RPC 框架，其通讯效率高于 WebService 和 Java 自带的序列化。

### http协议

短连接，多连接，同步传输，表单序列化即json

### webservice 协议

短连接，多连接，同步传输，soap序列化

基于 WebService 的远程调用协议，基于 Apache CXF的 frontend-simple 和 transports-http 实现。可以和原生 WebService 服务互操作，即：

提供者用 Dubbo 的 WebService 协议暴露服务，消费者直接用标准 WebService 接口调用，

或者提供方用标准 WebService 暴露服务，消费方用 Dubbo 的 WebService 协议调用。

### 6、thrift [θrift] 协议

当前 dubbo 支持的 thrift 协议是对 thrift 原生协议 [2] 的扩展，在原生协议的基础上添加了一些额外的头信息，比如 service name，magic number 等。

使用 dubbo thrift 协议同样需要使用 thrift 的 idl compiler 编译生成相应的 java 代码，后续版本中会在这方面做一些增强。

Thrift协议不允许null值存在，不能在协议中传null

### 7、memcached 协议

基于 memcached实现的 RPC 协议。

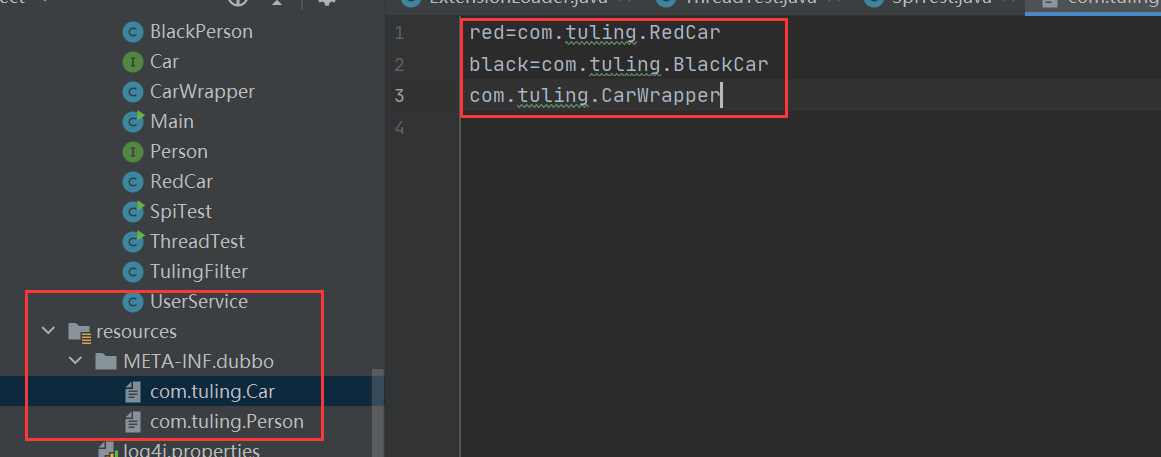
### 8、redis 协议

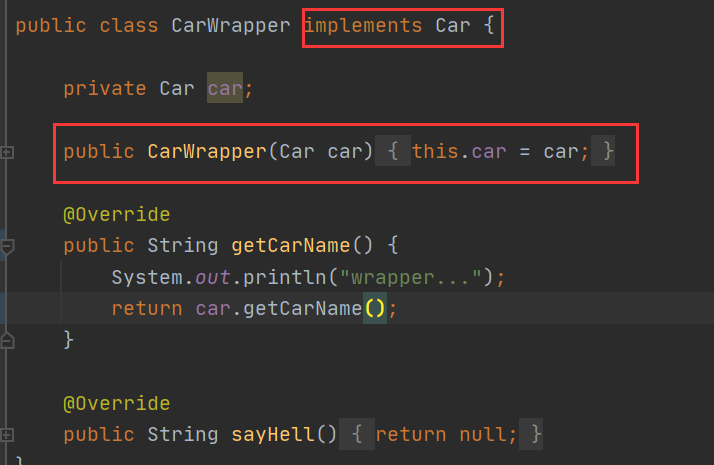
基于 Redis实现的 RPC 协议。

### rest协议

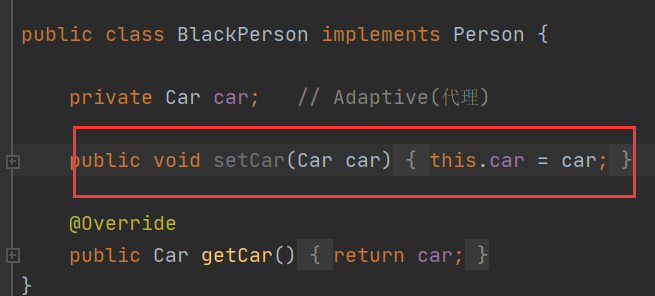
restful了协议，使用@Path实现http调用

## 2.1 dubbo的spi机制





此时加载car时会将对象包装为wrapper，carWrapper不会解析实例，而是被作为wrapper类缓存，扩展点其他类实例化时则调用进行包装



当一个类引用另一个接口实例且没有指定具体实现类，则会生成一个代理类实例，调用时再获取具体的实现类

大致的加载顺序如下：

获取指定扩展点的加载器----->获取实例-------》如果name传的true则会触发获取默认实例的逻辑------》其中getExtensionClasses方法比较重要------》loadExtensionClasses()此方法会加载扩展点的所有的实例并缓存-------》cacheDefaultExtensionName此方法获取默认的实现类并返回，获取的方式是去接口获取其配置的spi注解的value值，下面在调用loadDirectory获取所有的实现类并缓存，而且会处理wrapper与adapttive，此处不做详解

-------》当name不为true时调用正常逻辑------->createExtension(name)----->调用getExtensionClasses()方法加载实现类---->loadExtensionClasses()------>loadDirectory---------->loadResource--------------->loadClass-------->此方法首先判断实现类是否标注adaptive注解(自适应扩展点,自动生成一个动态adaptive类,根据@SPI注解的参数决定扩展点具体使用的实现类)，如果有缓存，没有则判断是否为wrapper类型，即有无以当前接口实例为参数的构造函数，如果没有进入最后的逻辑，首先判断类上有无activate注解，有的话缓存，没有继续往下走，检查实现类上有无extension注解指定的name值，然后缓存为普通的extension

回到createExtension(name)----->解析完实现类后创建实例，然后进行属性赋值，与wrpper调用首先看属性赋值----------->injectExtension(instance)找出实现类的所有set方法，判断是否需要依赖注入，使用objectFactory.getExtension获取需要注入的实例，工厂实例使用的AdaptiveExtensionFactory因为该类标注有adaptive注解，在该类中首先使用SpringExtensionFactory去ioc容器中获取实例，前提是类不是接口且不包含spi注解，如果此工厂返回空则调用SpiExtensionFactory工厂获取代理对象，如果接口的实现类没有标注adaptive注解则dubbo自动创建一个实现类，如果有则使用该类实例注入，此时注入接口上必须有adaptive标注的方法，而且只有此种方法才会被代理，同时方法有要求

1. 该⽅法如果是⽆参的，那么则会报错

2. 该⽅法有参数，可以有多个，并且其中某个参数类型是URL，那么则可以进⾏代理

3. 该⽅法有参数，可以有多个，但是没有URL类型的参数，那么则不能进⾏代理

4. 该⽅法有参数，可以有多个，没有URL类型的参数，但是如果这些参数类型，对应的类中存在getUrl⽅法（返回值类型为URL），那么也可以进⾏代理

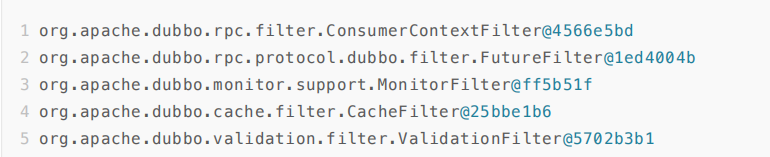
属性注入结束后，调用所有的wrapper缓存的类进行包装，实质是直接调用其构造函数创建对象

injectExtension((T) wrapperClass.getConstructor(type).newInstance(instance))

如果想一次获取多个扩展点实例



可以得到五个filter

、

前三个是通过CommonConstants.CONSUMER找到的

CacheFilter是通过url中的参数找到的

ValidationFilter是通过指定的name找到的

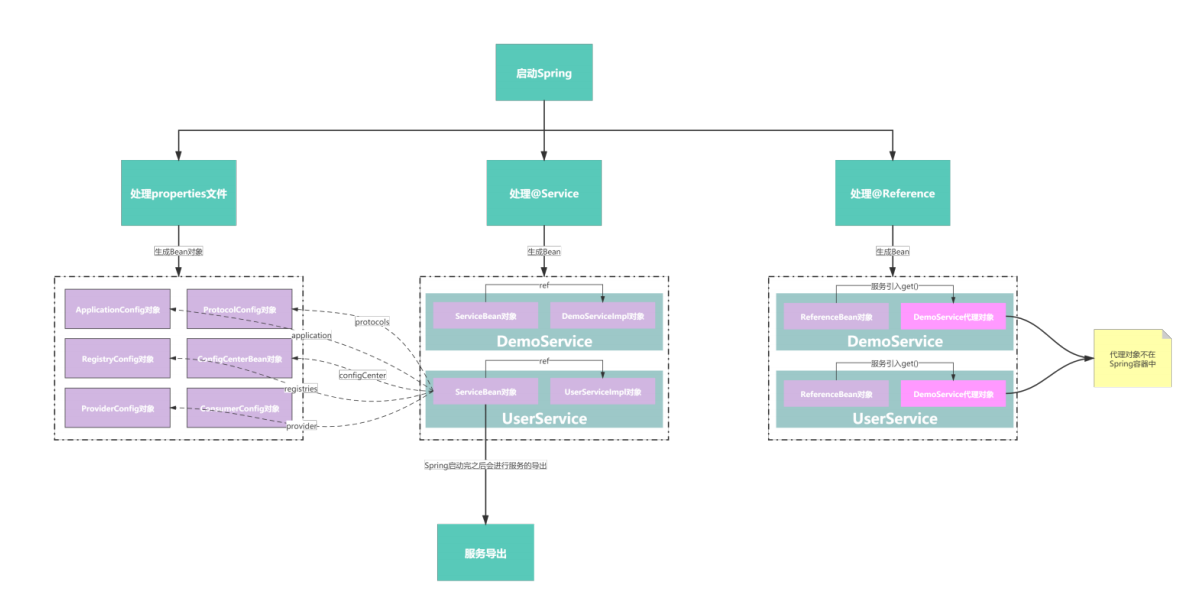
在⼀个扩展点类上，可以添加@Activate(自动激活扩展点,属性值进行过滤是否激活)注解，这个注解的属性有：

1. String[] group()：表示这个扩展点是属于哪组的，这⾥组通常分为PROVIDER和CONSUMER，表示

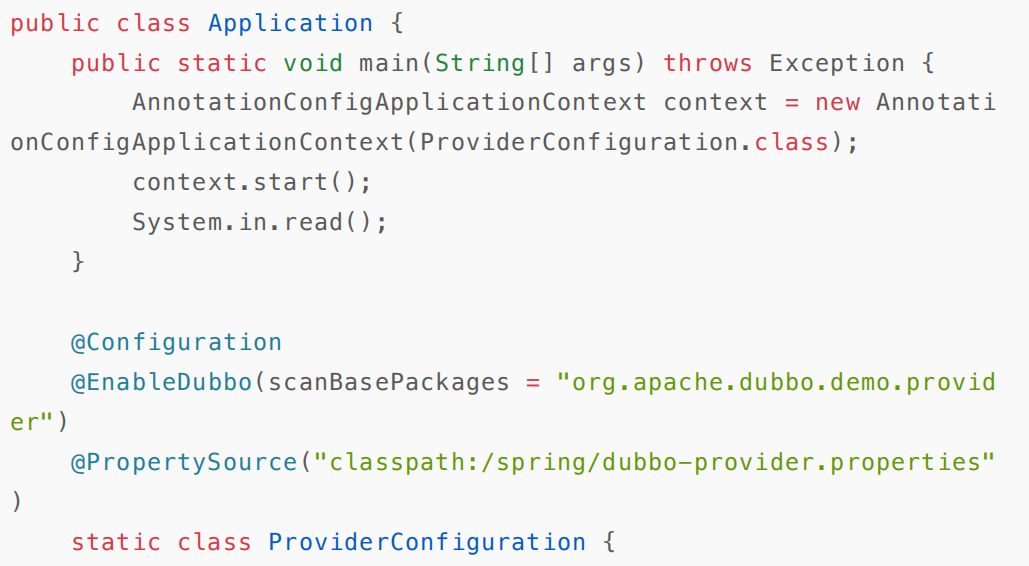
该扩展点能在服务提供者端，或者消费端使⽤

2. String[] value()：表示的是URL中的某个参数key，当利⽤getActivateExtension⽅法来寻找扩展点时，如果传⼊的url中包含的参数的所有key中，包括了当前扩展点中的value值，那么则表示当前url可

## 2.2 spring整合dubbo



### 2.1 配置文件解析



应⽤配置类为ProviderConfiguration, 在配置上有两个⽐较重要的注解

1. @PropertySource表示将dubbo-provider.properties中的配置项添加到Spring容器中，可以通过@Value的⽅式获取到配置项中的值

2. @EnableDubbo(scanBasePackages = "org.apache.dubbo.demo.provider")表示对指定包下的类进⾏扫描，扫描@Service与@Reference注解，并且进⾏处理

在EnableDubbo注解上，有另外两个注解，也是研究Dubbo最重要的两个注解

1. @EnableDubboConfig

2. @DubboComponentScan

注意两个注解中对应的@Import注解所导⼊的类：

1. DubboConfigConfigurationRegistrar

2. DubboComponentScanRegistrar

Spring在启动时会解析这两个注解，并且执⾏对应的Registrar类中的registerBeanDefinitions⽅法（这是Spring中提供的扩展功能。）

Spring启动时，会调⽤DubboConfigConfigurationRegistrar的registerBeanDefinitions⽅法，该⽅法是利⽤Spring中的AnnotatedBeanDefinitionReader来读取：

1. DubboConfigConfiguration.Single.class

2. DubboConfigConfiguration.Multiple.class

这两个类上的注解@EnableDubboConfigBindings

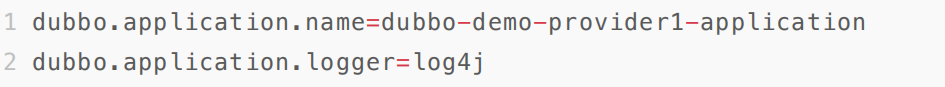
@EnableDubboConfigBindings注解上也有⼀个@Import注解，导⼊的是

DubboConfigBindingsRegistrar.class。该类会获取@EnableDubboConfigBindings注解中的value，

也就是多个@EnableDubboConfigBinding注解，然后利⽤DubboConfigBindingRegistrar去处理这些@EnableDubboConfigBinding注解

**DubboConfigBindingRegistrar**

此类中的主要⽅法是registerDubboConfigBeans()⽅法，主要功能就是获取⽤户所设置的properties⽂件中的内容，对Properties⽂件进⾏解析，根据Properties⽂件的每个配置项的前缀、参数名、参数值⽣成对应的BeanDefinition

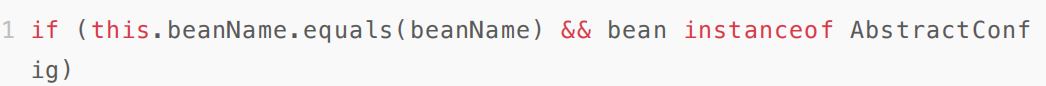


前缀为"dubbo.application"的配置项，会⽣成⼀个ApplicationConfig类型的BeanDefinition，并且name和logger属性为对应的值

并且还会针对⽣成的每个BeanDefinition⽣成⼀个和它⼀对⼀绑定的BeanPostProcessor，类型为DubboConfigBindingBeanPostProcessor.class。

DubboConfigBindingBeanPostProcessor

DubboConfigBindingBeanPostProcessor是⼀个BeanPostProcessor，在Spring启动过程中，会针对所有的Bean对象进⾏后置加⼯，但是在DubboConfigBindingBeanPostProcessor中有如下判断：

所以DubboConfigBindingBeanPostProcessor并不会处理Spring容器中的所有Bean，它只会处理上⽂由Dubbo所⽣成的Bean对象。

并且，在afterPropertiesSet()⽅法中，会先创建⼀个DefaultDubboConfigBinder。

**DefaultDubboConfigBinder**

当某个AbstractConfig类型的Bean，在经过DubboConfigBindingBeanPostProcessor处理时，此时Bean对象中的属性是没有值的，会利⽤DefaultDubboConfigBinder进⾏赋值。底层就是利⽤Spring中的 DataBinder技术，结合properties⽂件对对应的属性进⾏赋值。

对应⼀个AbstractConfig类型（针对的其实是⼦类，⽐如ApplicationConfig、RegistryConfig）的Bean，每个类都有⼀些属性，⽽properties⽂件是⼀个key-value对，所以实际上DataBinder就是将属性名和properties⽂件中的key进⾏匹配，如果匹配成功，则把value赋值给属性。具体DataBinder技术是如何⼯作的，请⾃⾏学习（不难）。

### 2.2 @service注解

SpringBoot整合在DubboAutoConfiguration内直接添加引入导出服务的注解处理的后置处理器

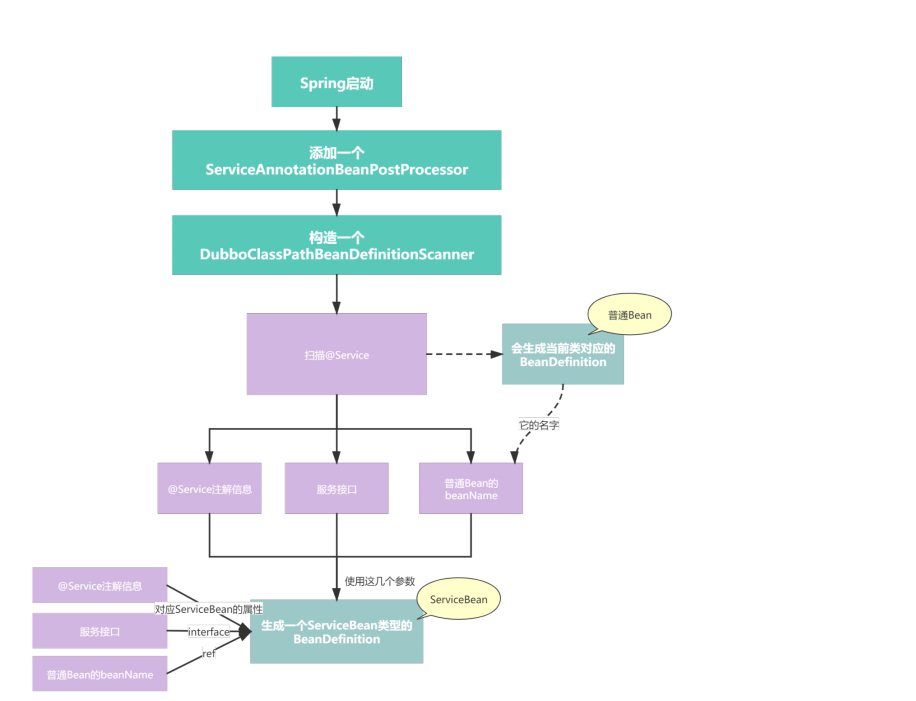
DubboComponentScanRegistrar

DubboConfigConfigurationRegistrar的作⽤是向Spring容器中注册两个Bean:

1. ServiceAnnotationBeanPostProcessor

2. ReferenceAnnotationBeanPostProcessor

ServiceAnnotationBeanPostProcessor



ServiceAnnotationBeanPostProcessor是⼀个BeanDefinitionRegistryPostProcessor，是⽤来注册

BeanDefinition的。它的主要作⽤是扫描Dubbo的@Service注解，⼀旦扫描到某个@Service注解就把它以及被它注解的类当做⼀个Dubbo服务，进⾏服务导出。

**DubboClassPathBeanDefinitionScanner**

DubboClassPathBeanDefinitionScanner是Dubbo⾃定义的扫描器，继承了Spring中的

ClassPathBeanDefinitionScanner了。

DubboClassPathBeanDefinitionScanner相对于ClassPathBeanDefinitionScanner并没有做太多的改变，只是把useDefaultFilters设置为了false，主要是因为Dubbo中的@Service注解是Dubbo⾃定义的，在这个注解上并没有⽤@Component注解（因为Dubbo不是⼀定要结合Spring才能⽤），所以为了能利⽤ Spring的扫描逻辑，需要把useDefaultFilters设置为false。

每扫描到⼀个@Service注解，就会得到⼀个BeanDefinition，这个BeanDefinition的beanClass属性就是具体的服务实现类。 但，如果仅仅只是这样，这只是得到了⼀个Spring中的Bean，对于Dubbo来说此时得到的Bean是⼀个服务，并且，还需要解析@Service注解的配置信息，因为这些都是服务的参数信息，所以在扫描完了之后，会针对所得到的每个BeanDefinition，都会额外的再registry⽣成⼀个ServiceBean类型的Bean对象。

ServiceBean

ServiceBean表示⼀个Dubbo服务，它有⼀些参数，⽐如：

1. ref，表示服务的具体实现类

2. interface，表示服务的接⼝

3. parameters，表示服务的参数（@Service注解中所配置的信息）

4. application，表示服务所属的应⽤

5. protocols，表示服务所使⽤的协议

6. registries，表示服务所要注册的注册中⼼

所以在扫描到⼀个@Service注解后，其实会得到两个Bean:

1. ⼀个就是服务实现类本身⼀个Bean对象

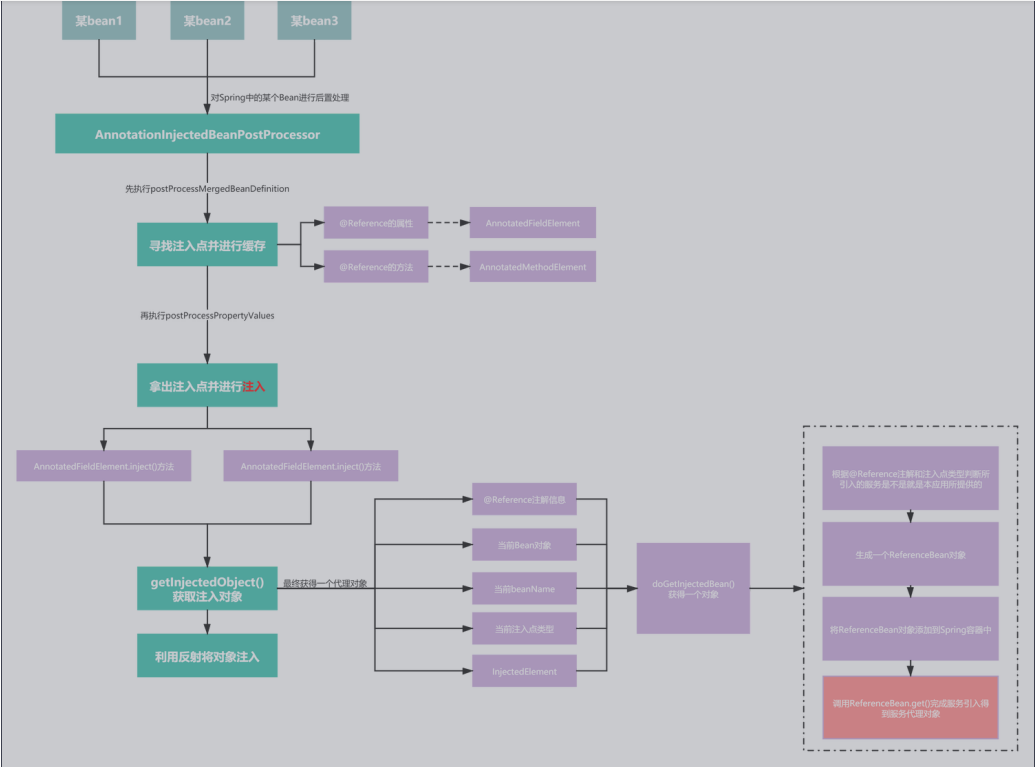
2. ⼀个就是对应的ServiceBean类型的⼀个Bean对象

并且需要注意的是，ServiceBean实现了ApplicationListener接⼝，所以当Spring启动完成后会触发 onApplicationEvent()⽅法的调⽤，⽽在这个⽅法内会调⽤export()，这个⽅法就是服务导出的⼊⼝⽅

Springboot：

DubboBootstrapApplicationListener实现了ApplicationListener接⼝在容器初始化完成之后会通过finishRefresh方法调用实现ApplicationListener的所有监听器，调用DubboBootstrapApplicationListener的onApplication方法调用到服务的导出逻辑

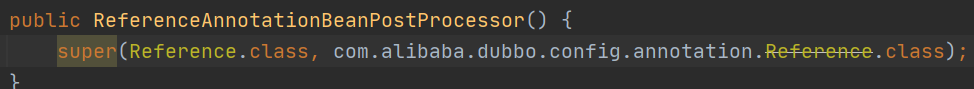
### 2.3 @reference注解



ReferenceAnnotationBeanPostProcessor的⽗类是AnnotationInjectedBeanPostProcessor，是⼀个 InstantiationAwareBeanPostProcessorAdapter，是⼀个BeanPostProcessor。

Spring在对Bean进⾏依赖注⼊时会调⽤AnnotationInjectedBeanPostProcessor的 postProcessPropertyValues()⽅法来给某个Bean按照ReferenceAnnotationBeanPostProcessor的逻辑进⾏依赖注⼊。 在注⼊之前会查找注⼊点，被@Reference注解的属性或⽅法都是注⼊点。

ReferenceAnnotationBeanPostProcessor的无参构造函数会调用父类的构造函数将需要扫描的注解类型传给AnnotationInjectedBeanPostProcessor，由父类扫描注入点



针对某个Bean找到所有注⼊点之后，就会进⾏注⼊了，注⼊就是给属性或给set⽅法赋值，但是在赋值之前得先得到⼀个值，此时就会调⽤ReferenceAnnotationBeanPostProcessor的doGetInjectedBean() ⽅法来得到⼀个对象，⽽这个对象的构造就⽐较复杂了，因为对于Dubbo来说，注⼊给某个属性的应该是当前这个属性所对应的服务接⼝的代理对象。

但是在⽣成这个代理对象之前，还要考虑问题：

1. 当前所需要引⼊的这个服务，是不是在本地就存在？不存在则要把按Dubbo的逻辑⽣成⼀个代理对象

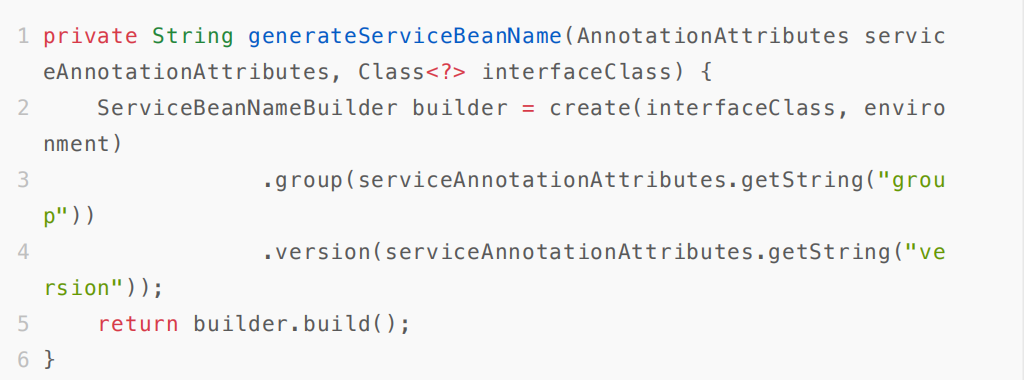
2. 当前所需要引⼊的这个服务，是不是已经被引⼊过了（是不是已经⽣成过代理对象了），如果是应该是不⽤再重复去⽣成了。

⾸先如何判断当前所引⼊的服务是本地的⼀个服务（就是当前应⽤⾃⼰所提供的服务）。

我们前⾯提到，Dubbo通过@Service来提供⼀个服务，并且会⽣成两个Bean：

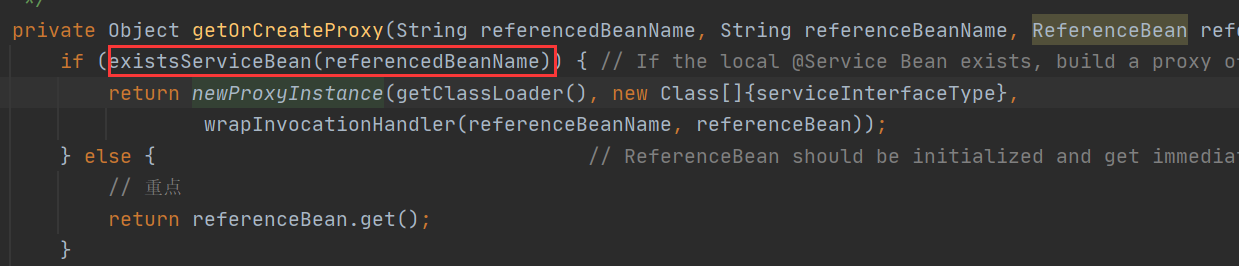
1. ⼀个服务实现类本身Bean

2. ⼀个ServiceBean类型的Bean，这个Bean的名字是这么⽣成的：



是通过接⼝类型+group+version来作为ServiceBean类型Bean的名字的。

所以现在对于服务引⼊，也应该提前根据@Reference注解中的信息和属性接⼝类型去判断⼀下当前Spring容器中是否存在对应的ServiceBean对象，如果存在则直接取出ServiceBean对象的ref属性所对应的对象，作为要注⼊的结果。



然后如何判断当前所引⼊的这个服务是否已经被引⼊过了（是不是已经⽣成过代理对象了）。

这就需要在第⼀次引⼊某个服务后（⽣成代理对象后）进⾏缓存（记录⼀下）。Dubbo中是这么做的:

1. ⾸先根据@Reference注解的所有信息+属性接⼝类型⽣成⼀个字符串

2. 然后@Reference注解的所有信息+属性接⼝类型⽣成⼀个ReferenceBean对象（ReferenceBean对象中的get⽅法可以得到⼀个Dubbo⽣成的代理对象，可以理解为服务引⼊的⼊⼝⽅法）

3. 把字符串作为beanName，ReferenceBean对象作为bean注册到Spring容器中，同时也会放⼊referenceBeanCache中。

有了这些逻辑，@Reference注解服务引⼊的过程是这样的：

1. 得到当前所引⼊服务对应的ServiceBean的beanName（源码中叫referencedBeanName）

2. 根据@Reference注解的所有信息+属性接⼝类型得到⼀个referenceBeanName

3. 根据referenceBeanName从referenceBeanCache获取对应的ReferenceBean，如果没有则创建⼀个ReferenceBean

4. 根据referencedBeanName（ServiceBean的beanName）判断Spring容器中是否存在该bean，如果存在则给ref属性所对应的bean取⼀个别名，别名为referenceBeanName。

a. 如果Spring容器中不存在referencedBeanName对应的bean，则判断容器中是否存在

referenceBeanName所对应的Bean，如果不存在则将创建出来的ReferenceBean注册到 Spring 容器中（此处这么做就⽀持了可以通过@Autowired注解也可以使⽤服务了， ReferenceBean是⼀个FactoryBean）

5. 如果referencedBeanName存在对应的Bean，则额外⽣成⼀个代理对象，代理对象的

InvocationHandler会缓存在localReferenceBeanInvocationHandlerCache中，这样如果引⼊的是同⼀个服务，并且这个服务在本地，

1. 如果referencedBeanName不存在对应的Bean，则直接调⽤ReferenceBean的get()⽅法得到⼀个代理对象

**后续在导入中具体说代理对象的创建**

## 2.3 服务导出

### 3.1 服务导出过程

**服务导出⼤概原理（⾯试时可以这么答）**

服务导出的⼊⼝为ServiceBean中的export()⽅法，当Spring启动完之后，通过接收Spring的ContextRefreshedEvent事件来触发export()⽅法的执⾏。

⼀个ServiceBean对象就表示⼀个Dubbo服务，ServiceBean对象中的参数就表示服务的参数，⽐如timeout，该对象的参数值来⾄@Service注解中所定义的。

服务导出主要得做两件事情：

1. 根据服务的参数信息，启动对应的⽹络服务器（netty、tomcat、jetty等），⽤来接收⽹络请求

2. 将服务的信息注册到注册中⼼

但是在做这两件事情之前得先把服务的参数确定好，因为⼀个Dubbo服务的参数，除开可以在@Service注解中去配置，还会继承Dubbo服务所属应⽤（Application）上的配置，还可以在配置中⼼或JVM环境变量中去配置某个服务的参数，所以⾸先要做的是确定好当前服务最终的（优先级最⾼）的参数值。

确定好服务参数之后，就根据所配置的协议启动对应的⽹络服务器。在启动⽹络服务器时，并且在⽹络服务器接收请求的过程中，都可以从服务参数中获取信息，⽐如最⼤连接数，线程数，socket超时时间等等。

启动完⽹络服务器之后，就将服务信息注册到注册中⼼。同时还有向注册中⼼注册监听器，监听Dubbo中的动态配置信息变更。

服务导出要做的⼏件事情：

1. 确定服务的参数

2. 确定服务⽀持的协议

3. 构造服务最终的URL

4. 将服务URL注册到注册中⼼去

5. 根据服务⽀持的不同协议，启动不同的Server，⽤来接收和处理请求

6. 因为Dubbo⽀持动态配置服务参数，所以服务导出时还需要绑定⼀个监听器Listener来监听服务的参数

**确定服务的参数**

在执⾏ServiceConfig.export()时，此时ServiceConfig对象就代表⼀个服务，我们已经知道了这个服务的名字（就是接⼝的名字），并且此时这个服务可能已经有⼀些参数了，就是@Service注解上所定义的参数。

但是在Dubbo中，除开可以在@Service注解中给服务配置参数，还有很多地⽅也可以给服务配置参数，⽐如：

1. dubbo.properties⽂件，你可以建⽴这个⽂件，dubbo会去读取这个⽂件的内容作为服务的参数，Dubob的源码中叫做PropertiesConfiguration

2. 配置中⼼，dubbo在2.7版本后就⽀持了分布式配置中⼼，你可以在Dubbo-Admin中去操作配置中⼼，分布式配置中⼼就相当于⼀个远程的dubbo.properties⽂件，你可以在Dubbo-Admin中去修改这个dubbo.properties⽂件，当然配置中⼼⽀持按应⽤进⾏配置，也可以按全局进⾏配置两种，在Dubbo的源码中AppExternalConfiguration表示应⽤配置，ExternalConfiguration表示全局配置。

3. 系统环境变量，你可以在启动应⽤程序时，通过-D的⽅式来指定参数，在Dubbo的源码中叫SystemConfiguration

4. 再加上通过@Service注解所配置的参数，在Dubbo的源码中叫AbstractConfig

服务的参数可以从这四个位置来，这四个位置上如果配了同⼀个参数的话，优先级从⾼到低如下：SystemConfiguration -> AppExternalConfiguration -> ExternalConfiguration ->

AbstractConfig -> PropertiesConfiguration

可以看出，-D⽅式配置的参数优先级最⾼，配置中⼼次之，注解随后，dubbo.properties最后。

你可以在dubbo.properties⽂件或配置中⼼中通过⼀下形式去给⼀个服务配置参数：

dubbo.service.{interface-name}[.{method-name}].timeout=3000

-D⽅式也类似。

从以上分析我们可以看出，在服务导出时，⾸先得确定服务的参数。

当然，服务的参数除开来⾃于服务的⾃身配置外，还可以来⾃其上级。

⽐如如果服务本身没有配置timeout参数，但是如果服务所属的应⽤的配置了timeout，那么这个应⽤下的服务都会继承这个timeout配置。

所以在确定服务参数时，需要先从上级获取参数，获取之后，如果服务本身配置了相同的参数，那么则进⾏覆盖。

**确定服务⽀持的协议**

确定服务所⽀持的协议还是⽐较简单的，就是看⽤户配了多少个Protocol。和服务参数意义，Protocol也是可以在各个配置点进⾏配置的。

1. ⾸先在SpringBoot的application.properties⽂件中就可能配置了协议

2. 也可能在dubbo.properties⽂件中配置了协议

3. 也可能在配置中⼼中也配置了协议

4. 也可能通过-D的⽅式也配置了协议

所以在服务导出时，需要从以上⼏个地⽅获取协议，结果可能是⼀个协议，也可能是多个协议，从⽽确定出协议。构造服务最终的URL

有了确定的协议，服务名，服务参数后，⾃然就可以组装成服务的URL了。

但是还有⼀点是⾮常重要的，在Dubbo中⽀持服务动态配置，注意，这个和配置中⼼不是同⼀概念，动态配置是可以在服务导出后动态的去修改服务配置的，⽽配置中⼼则不能达到这⼀的效果（这个我要在确定⼀下）。

动态配置，其实就是继续给服务增加了⼀些参数，所以在把服务的URL注册到注册中⼼去之前，得先按照动态配置中所添加的配置重写⼀下URL，也就是应⽤上动态配置中的参数。

只有这样作完之后得到的URL才是真正准确的服务提供者URL。

**将服务URL注册到注册中⼼去**

有了准确的服务URL之后，就可以把URL注册到注册中⼼上去了。

这个步骤并不麻烦，只不过这⾥要去寻找⽤户是否配置了多个注册中⼼，将服务URL注册到每个注册中⼼去。

**根据服务URL启动Server**

在服务URL中指定了协议，⽐如Http协议、Dubbo协议。根据不同的协议启动对应的Server。

⽐如Http协议就启动Tomcat、Jetty。

⽐如Dubbo协议就启动Netty。

不能只启动Server，还需要绑定⼀个RequestHandler，⽤来处理请求。

⽐如，Http协议对应的就是InternalHandler。Dubbo协议对应的就是ExchangeHandler。

这⾥来详细分析⼀下Dubbo协议所启动的Server。

1. 调⽤DubboProtocol的openServer(URL url)⽅法开启启动Server

2. 调⽤DubboProtocol的createServer(url)⽅法，在createServer()⽅法中调Exchangers.bind(url,

requestHandler)得到⼀个ExchangeServer

3. 其中requestHandler表示请求处理器，⽤来处理请求

4. 在Exchangers.bind(url, requestHandler)中，先会根据URL得到⼀个Exchanger，默认为

HeaderExchanger

5.

6. HeaderExchangeClient负责发送⼼跳，HeaderExchangeServer负责接收⼼跳，如果超时则会关闭channel

7. 在构造HeaderExchangeServer之前，会通过调⽤Transporters.bind(url, new DecodeHandler(new HeaderExchangeHandler(handler)))⽅法的到⼀个Server

8. 默认会使⽤getTransporter去bind(URL url, ChannelHandler listener)从⽽得到⼀个Servlet，此时的listener就是外部传进来的DecodeHandler

9. 在NettyTransporter的bind⽅法中会去new NettyServer(url, listener)，所以上⾯返回的Server默认就是NettyServer

10. 在构造NettyServer时，会调⽤ChannelHandlers.wrap(handler, ExecutorUtil.setThreadName(url, SERVER\_THREAD\_POOL\_NAME))再构造⼀个ChannelHandler。

11. wrap中的handler就是上⾯的listener

12. 在wrap⽅法中会调⽤new MultiMessageHandler(new

HeartbeatHandler(ExtensionLoader.getExtensionLoader(Dispatcher.class).getAdaptiveExtens

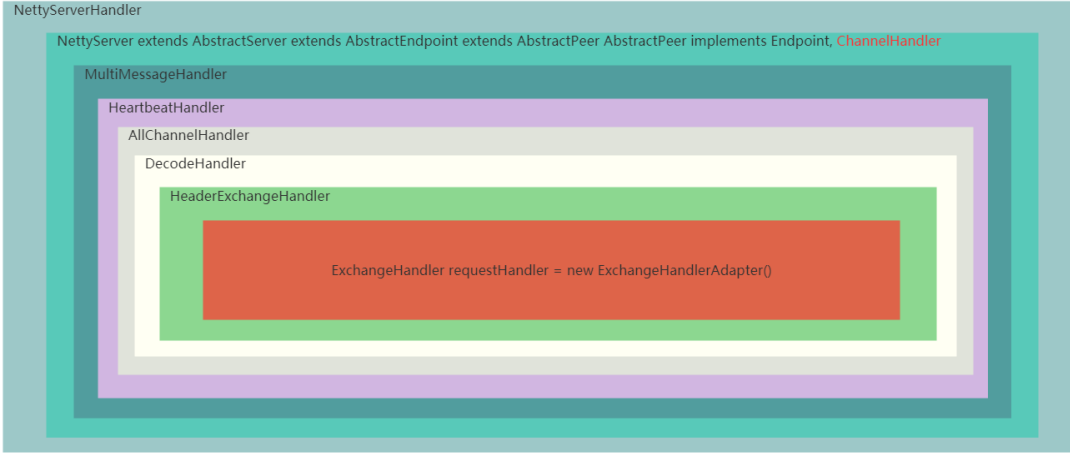
ion().dispatch(handler, url)));构造⼀个ChannelHandler。

13. 构造完ChannelHandler后，就是真正的去开启Server了，会调⽤AbstractServer抽象类的doOpen⽅法。

14. 在NettyServer中，会实现doOpen⽅法，会调⽤new NettyServerHandler(getUrl(), this)构造⼀个NettyServerHandler，并bind地址

15. ⾄此，DubboProtocol协议的启动Server流程就结束。

总结⼀下DubboProtocol协议的RequestHandler链路：



1. NettyServerHandler：与NettyServer直接绑定的请求处理器，负责从Netty接收到请求，

channelRead()⽅法获取到请求，然后调⽤下⼀层的Handler（NettyServer）的received()⽅法将请求传递下去，此时的请求还是Object msg

2. NettyServer：NettyServer的⽗类AbstractPeer中存在received()，该⽅法没有做什么，直接把msg传递给下⼀层Handler（MultiMessageHandler）

3. MultiMessageHandler：此Handler会判断msg是否是⼀个MultiMessage，如果是，则对

MultiMessage进⾏拆分，则把拆分出来的msg传递给下层Handler（HeartbeatHandler），如果不是，则直接把msg传递给下层Handler（HeartbeatHandler）

4. HeartbeatHandler：此Handler通过received()⽅法接收到msg，然后判断该msg是不是⼀个⼼跳请求或⼼跳响应，如果是⼼跳请求，则此Handler返回⼀个Response对象（很简单的⼀个对象），如果是⼼跳响应，则打印⼀个⽇志，不会有其他逻辑，如果都不是，则把msg传递给下层Handler（AllChannelHandler）。

5. AllChannelHandler：此Handler通过received()⽅法接收到msg，然后把msg封装为⼀个

ChannelEventRunnable对象，并把ChannelEventRunnable扔到线程池中去，异步去处理该msg。在ChannelEventRunnable中会把msg交给下⼀个Handler（DecodeHandler）

6. DecodeHandler：此Handler通过received()⽅法接收到msg，会对msg解析decode解码，然后交给下⼀个Handler(HeaderExchangeHandler)

7. HeaderExchangeHandler：此Handler通过received()⽅法接收到msg，会判断msg的类型

a. 如果Request是TwoWay，则会调⽤下⼀个Handler(DubboProtocol中的requestHandler)的reply⽅法得到⼀个结果，然后返回

b. 如果Request不是TwoWay，则会调⽤下⼀个Handler(DubboProtocol中的requestHandler)的 received⽅法处理该请求，不会返回结果

8. requestHandler：此Handler是真正的处理请求逻辑，在received()⽅法中，如果msg是Invocation，则会调⽤reply⽅法，但不会返回reply⽅法所返回的结果，在reply⽅法中把msg强制转换为Invocation 类型 inv，然后根据inv得到对应的服务Invoker，然后调⽤invoke(inv)⽅法，得到结果。

**服务导出源码流程**

1. ServiceBean.export()⽅法是导出的⼊⼝⽅法，会执⾏ServiceConfig.export()⽅法完成服务导出，导出完了之后会发布⼀个Spring事件ServiceBeanExportedEvent

2. 在ServiceConfig.export()⽅法中会先调⽤checkAndUpdateSubConfigs()，这个⽅法主要完成

AbstractConfig的参数刷新(从配置中⼼获取参数等等)，AbstractConfig是指ApplicationConfig、

ProtocolConfig、ServiceConfig等等，刷新完后会检查stub、local、mock等参数是否配置正确

3. 参数刷新和检查完成了之后，就会开始导出服务，如果配置了延迟导出，那么则按指定的时间利⽤ ScheduledExecutorService来进⾏延迟导出

4. 否则调⽤doExport()进⾏服务导出

5. 继续调⽤doExportUrls()进⾏服务导出

6. ⾸先通过loadRegistries()⽅法获得所配置的注册中⼼的URL，可能配了多个配置中⼼，那么当前所导出的服务需要注册到每个配置中⼼去，这⾥，注册中⼼的是以URL的⽅式来表示的，使⽤的是什么注册中⼼、注册中⼼的地址和端⼝，给注册中⼼所配置的参数等等，都会存在在URL上，此URL以registry://开始

7. 获得到注册中⼼的registryURLs之后，就会遍历当前服务所有的ProtocolConfig，调⽤

doExportUrlsFor1Protocol(protocolConfig, registryURLs);⽅法把当前服务按每个协议每个注册中⼼分别进⾏导出

8. 在doExportUrlsFor1Protocol()⽅法中，会先构造⼀个服务URL，包括

a. 服务的协议dubbo://,

b. 服务的IP和PORT，如果指定了就取指定的，没有指定IP就获取服务器上⽹卡的IP，

c. 以及服务的PATH，如果没有指定PATH参数，则取接⼝名

d. 以及服务的参数，参数包括服务的参数，服务中某个⽅法的参数

e. 最终得到的URL类似： dubbo://192.168.1.110:20880/com.tuling.DemoService?

timeout=3000&&sayHello.loadbalance=random

9. 得到服务的URL之后，会把服务URL作为⼀个参数添加到registryURL中去，然后把registryURL、服务的接⼝、当前服务实现类ref⽣成⼀个Invoker代理对象，再把这个代理对象和当前ServiceConfig对象包装成⼀个DelegateProviderMetaDataInvoker对象，DelegateProviderMetaDataInvoker就表示了完整的⼀个服务

10. 接下来就会使⽤Protocol去export导出服务了，导出之后将得到⼀个Exporter对象（该Exporter对象，可以理解为主要可以⽤来卸载（unexport）服务，什么时候会卸载服务？在优雅关闭Dubbo应⽤的时候）

11. 接下来我们来详细看看Protocol是怎么导出服务的？

12. 但调⽤protocol.export(wrapperInvoker)⽅法时，因为protocol是Protocol接⼝的⼀个Adaptive对象，所以此时会根据wrapperInvoker的genUrl⽅法得到⼀个url，根据此url的协议找到对应的扩展点，此时扩展点就是RegistryProtocol，但是，因为Protocol接⼝有两个包装类，⼀个是ProtocolFilterWrapper、ProtocolListenerWrapper，所以实际上在调⽤export⽅法时，会经过这两个包装类的export⽅法，但是在这两个包装类的export⽅法中都会Registry协议进⾏了判断，不会做过多处理，所以最终会直接调⽤到RegistryProtocol export(Invoker<T> originInvoker)⽅法

13. 在RegistryProtocol的export(Invoker<T> originInvoker)⽅法中，主要完成了以下⼏件事情：

a. ⽣成监听器，监听动态配置中⼼此服务的参数数据的变化，⼀旦监听到变化，则重写服务URL，并且在服务导出时先重写⼀次服务URL

b. 拿到重写之后的URL之后，调⽤doLocalExport()进⾏服务导出，在这个⽅法中就会调⽤

DubboProtocol的export⽅法去导出服务了，导出成功后将得到⼀个ExporterChangeableWrapper

i. 在DubboProtocol的export⽅法中主要要做的事情就是启动NettyServer，并且设置⼀系列的RequestHandler，以便在接收到请求时能依次被这些RequestHandler所处理

ii. 这些RequestHandler在上⽂已经整理过了

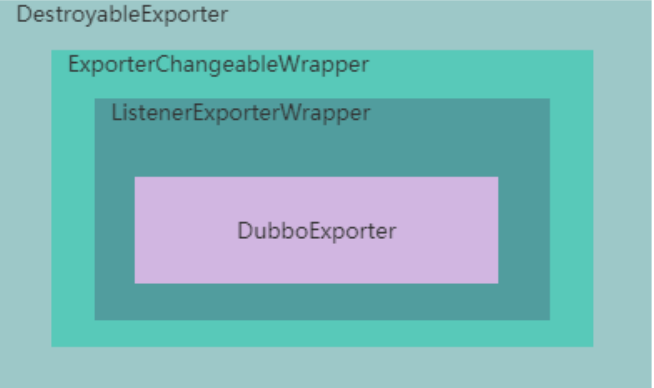
c. 从originInvoker中获取注册中⼼的实现类，⽐如ZookeeperRegistry

d. 将重写后的服务URL进⾏简化，把不⽤存到注册中⼼去的参数去除

e. 把简化后的服务URL调⽤ZookeeperRegistry.registry()⽅法注册到注册中⼼去

f. 最后将ExporterChangeableWrapper封装为DestroyableExporter对象返回，完成服务导出

Exporter架构



⼀个服务导出成功后，会⽣成对应的Exporter：

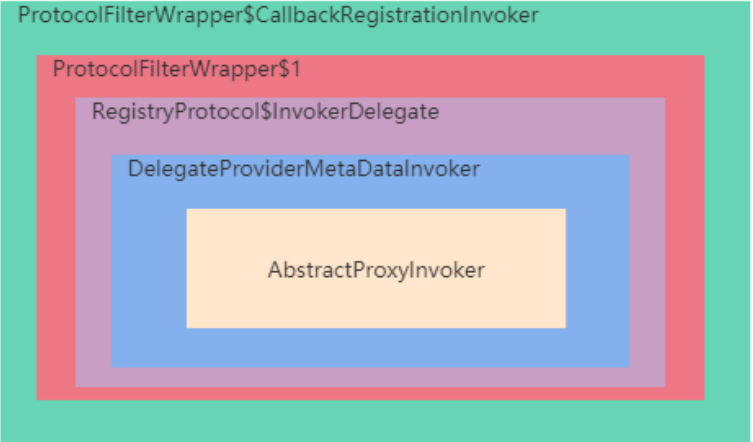
1. DestroyableExporter：Exporter的最外层包装类，这个类的主要作⽤是可以⽤来unexporter对应的服务

2. ExporterChangeableWrapper：这个类主要负责在unexport对应服务之前，把服务URL从注册中⼼中移除，把该服务对应的动态配置监听器移除

3. ListenerExporterWrapper：这个类主要负责在unexport对应服务之后，把服务导出监听器移除

4. DubboExporter：这个类中保存了对应服务的Invoker对象，和当前服务的唯⼀标志，当NettyServer接收到请求后，会根据请求中的服务信息，找到服务对应的DubboExporter对象，然后从对象中得到Invoker对象

服务端Invoker架构



1. ProtocolFilterWrapper$CallbackRegistrationInvoker：会去调⽤下层Invoker，下层Invoker执⾏完了之后，会遍历过滤器，查看是否有过滤器实现了ListenableFilter接⼝，如果有，则回调对应的onResponse⽅法，⽐如TimeoutFilter，当调⽤完下层Invoker之后，就会计算服务的执⾏时间

2. ProtocolFilterWrapper$1：ProtocolFilterWrapper中的过滤器组成的Invoker，利⽤该Invoker，可以执⾏服务端的过滤器，执⾏完过滤器之后，调⽤下层Invoker

3. RegistryProtocol$InvokerDelegate：服务的的委托类，⾥⾯包含了

DelegateProviderMetaDataInvoker对象和服务对应的providerUrl，执⾏时直接调⽤下层Invoker

4. DelegateProviderMetaDataInvoker：服务的的委托类，⾥⾯包含了AbstractProxyInvoker对象和ServiceConfig对象，执⾏时直接调⽤下层Invoker

5. AbstractProxyInvoker：服务接⼝的代理类，绑定了对应的实现类，执⾏时会利⽤反射调⽤服务实现类实例的具体⽅法，并得到结果

服务监听器原理总结

服务在导出的过程中需要向动态配置中⼼的数据进⾏订阅，以便当管理⼈员修改了动态配置中⼼中对应服务的参数后，服务提供者能及时做出变化。

此功能涉及到版本兼容，因为在Dubbo2.7之前也存在此功能，Dubbo2.7开始对此功能进⾏了调整。

在Dubbo2.7之前，仅⽀持某个服务的动态配置

在Dubbo2.7之后，不仅⽀持对单个服务的动态配置，也⽀持对某个应⽤的动态配置（相当于对这个应⽤下的所有服务⽣效）

9为了达到这个功能，需要利⽤Zookeeper的Watcher机制，所以对于服务提供者⽽⾔，我到底监听哪个Zookeeper节点的数据变化呢？

这个节点是有规则的，并且在Dubbo2.7前后也不⼀样：

Dubbo2.7之前：

监听的zk路径是：

/dubbo/org.apache.dubbo.demo.DemoService/configurators/override://0.0.0.0/org.apache.dub

bo.demo.DemoService?

category=configurators&compatible\_config=true&dynamic=false&enabled=true&timeout=6000

注意，注意监听的是节点名字的变化，⽽不是节点内容

Dubbo2.7之后

监听的zk路径是：

服务： /dubbo/config/dubbo/org.apache.dubbo.demo.DemoService.configurators节点的内容

应⽤： /dubbo/config/dubbo/dubbo-demo-provider-application.configurators节点的内容

注意，要和配置中⼼的路径区分开来，配置中⼼的路径是：

应⽤：/dubbo/config/dubbo/org.apache.dubbo.demo.DemoService/dubbo.properties节点的内

容

全局：/dubbo/config/dubbo/dubbo.properties节点的内容

所以在⼀个服务进⾏导出时，需要在服务提供者端给当前服务⽣成⼀个对应的监听器实例，这个监听器实例为OverrideListener，它负责监听对应服务的动态配置变化，并且根据动态配置中⼼的参数重写服务URL。

除开有OverrideListener之外，在Dubbo2.7之后增加了另外两个：

1. ProviderConfigurationListener：监听的是应⽤的动态配置数据修改，所以它是在RegistryProtocol类中的⼀个属性，并且是随着RegistryProtocol实例化⽽实例化好的，⼀个应⽤中只有⼀个

2. ServiceConfigurationListener：监听的是服务的动态配置数据修改，和OverrideListener类似，也是对应⼀个服务的，所以在每个服务进⾏导出时都会⽣成⼀个，实际上ServiceConfigurationListener的内部有⼀个属性就是OverrideListener，所以当ServiceConfigurationListener监听数据发⽣了变化时，就会把配置中⼼的最新数据交给OverrideListener去重写服务URL。

3. 同时在RegistryProtocol类中报存了所有服务所对应的OverrideListener，所以实际上当

ProviderConfigurationListener监听到数据发⽣了变化时，也会把它所得到的最新数据依次调⽤每个

OverrideListener去重写服务对应的服务URL。

104. ProviderConfigurationListener会监听/dubbo/config/dubbo/dubbo-demo-provider-

application.configurators节点

5. ServiceConfigurationListener会监

听/dubbo/config/dubbo/org.apache.dubbo.demo.DemoService.configurators节点

整理修改动态配置触发流程：

1. 修改服务动态配置，底层会修改Zookeeper中的数据，

a. /dubbo/config/dubbo/org.apache.dubbo.demo.DemoService.configurators节点的内容

2. ServiceConfigurationListener会监听到节点内容的变化，会触发ServiceConfigurationListener的

⽗类AbstractConfiguratorListener的process(ConfigChangeEvent event)⽅法

3. ConfigChangeEvent表示⼀个事件，事件中有事件类型，还有事件内容（节点内容），还有触发这个

事件的节点名字，事件类型有三个：

a. ADDED

b. MODIFIED

c. DELETED

4. 当接收到⼀个ConfigChangeEvent事件后，会根据事件类型做对应的处理

a. ADDED、MODIFIED：会根据节点内容去⽣成override://协议的URL，然后根据URL去⽣成

Configurator， Configurator对象很重要，表示⼀个配置器，根据配置器可以去重写URL

b. DELETED：删除ServiceConfigurationListener内的所有的Configurator

5. ⽣成了Configurator后，调⽤notifyOverrides()⽅法对服务URL进⾏重写

6. 注意，每次重写并不仅仅只是⽤到上⾯所⽣成的Configurator，每次重写要⽤到所有的

Configurator，包括本服务的Configurator，也包括本应⽤的Configurator，也包括⽼版本管理台的

Configurator，重写URL的逻辑如下：

a. 从exporter中获取⽬前已经导出了的服务URL-currentUrl

b. 根据⽼版本管理台的Configurator重写服务URL

c. 根据providerConfigurationListener中的Configurator重写服务URL

d. 根据serviceConfigurationListeners中对应的服务的Configurator重写服务URL

e. 如果重写之后newUrl和currentUrl相等，那么不需要做什么了

f. 如果重写之后newUrl和currentUrl不相等，则需要进⾏服务重新导出：

i. 根据newUrl进⾏导出，注意，这⾥只是就是调⽤DubboProtocol的export，再次去启动

NettyServer

ii. 对newUrl进⾏简化，简化为registeredProviderUrl

iii. 调⽤RegistryProtocol的unregister()⽅法，把当前服务之前的服务提供URL从注册中⼼删掉

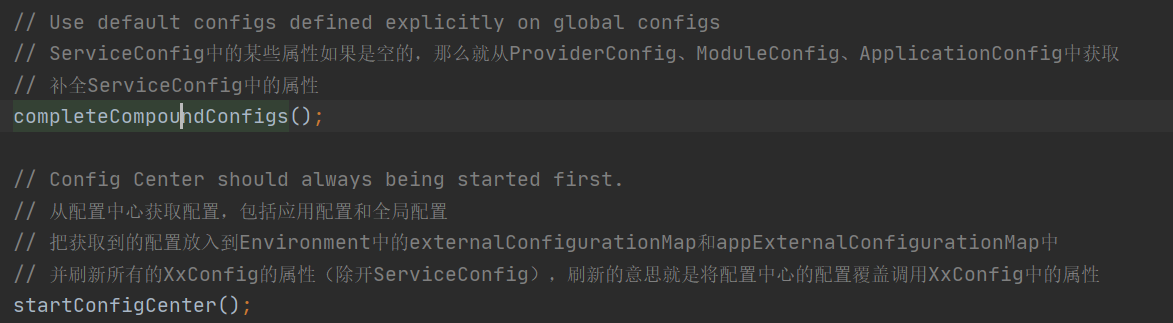
iv. 调⽤RegistryProtocol的register()⽅法，把新的registeredProviderUrl注册到注册中⼼

### 3.2 源码

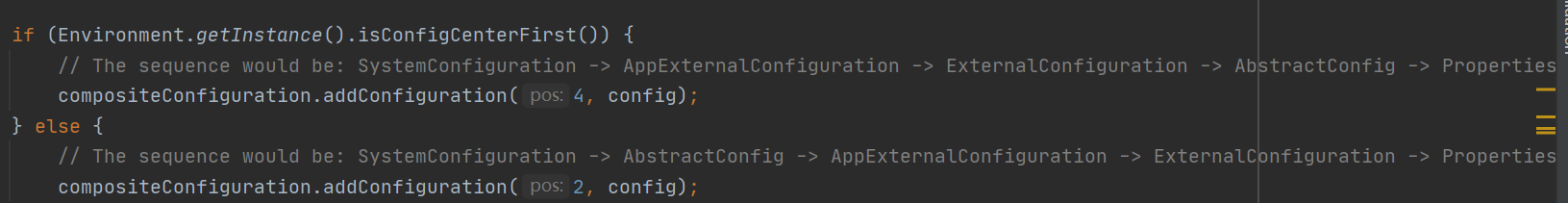
首先是在serviceBean的监听方法中触发导出功能，导出后会发布ServiceBeanExportedEvent事件，可监听事件，做导出后的日志或者逻辑，现在重点说导出

#### checkAndUpdateSubConfigs()

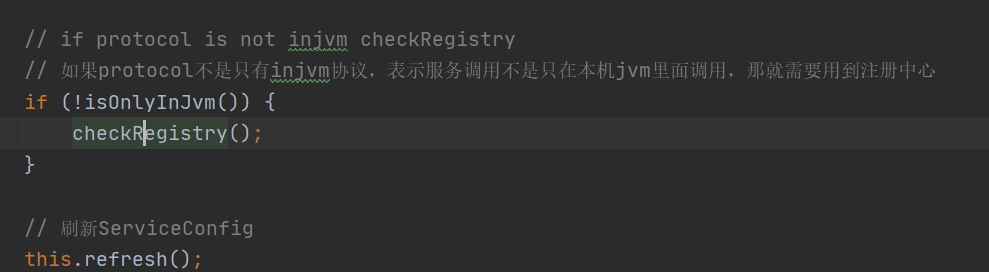
首先调用checkAndUpdateSubConfigs()，对属性进行填充



可通过配置决定配置中心与serviceConfig是的优先级高，数值越小优先级越高



如果需要注册中心，则checkRegistry中会启动注册中心获取配置中心的配置与动态配置



#### doExport()

1. 继续调⽤doExportUrls()进⾏服务导出

遍历所有的协议为每种协议注册服务

6. ⾸先通过loadRegistries()⽅法获得所配置的注册中⼼的URL，可能配了多个配置中⼼，那么当前所导出的服务需要注册到每个配置中⼼去，这⾥，注册中⼼的是以URL的⽅式来表示的，使⽤的是什么注册中⼼、注册中⼼的地址和端⼝，给注册中⼼所配置的参数等等，都会存在在URL上，此URL以registry://开始

7. 获得到注册中⼼的registryURLs之后，就会遍历当前服务所有的ProtocolConfig，调⽤

doExportUrlsFor1Protocol(protocolConfig, registryURLs);⽅法把当前服务按每个协议每个注册中⼼分别进⾏导出

8. 在doExportUrlsFor1Protocol()⽅法中，会先构造⼀个服务URL，包括

a. 服务的协议dubbo://,

b. 服务的IP和PORT，如果指定了就取指定的，没有指定IP就获取服务器上⽹卡的IP，

c. 以及服务的PATH，如果没有指定PATH参数，则取接⼝名

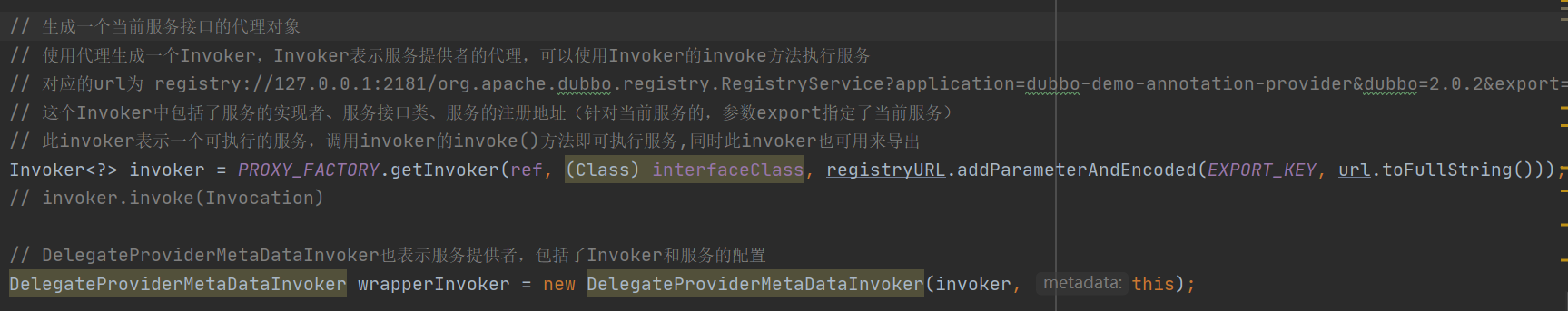
d. 以及服务的参数，参数包括服务的参数，服务中某个⽅法的参数

e. 最终得到的URL类似： dubbo://192.168.1.110:20880/com.tuling.DemoService?

timeout=3000&&sayHello.loadbalance=random

如果使用token会在该方法中添加

1. 得到服务的URL之后，会把服务URL作为⼀个参数添加到registryURL中去，然后把registryURL、服务的接⼝、当前服务实现类ref⽣成⼀个Invoker代理对象，再把这个代理对象和当前ServiceConfig对象包装成⼀个DelegateProviderMetaDataInvoker对象，DelegateProviderMetaDataInvoker就表示了完整的⼀个服务



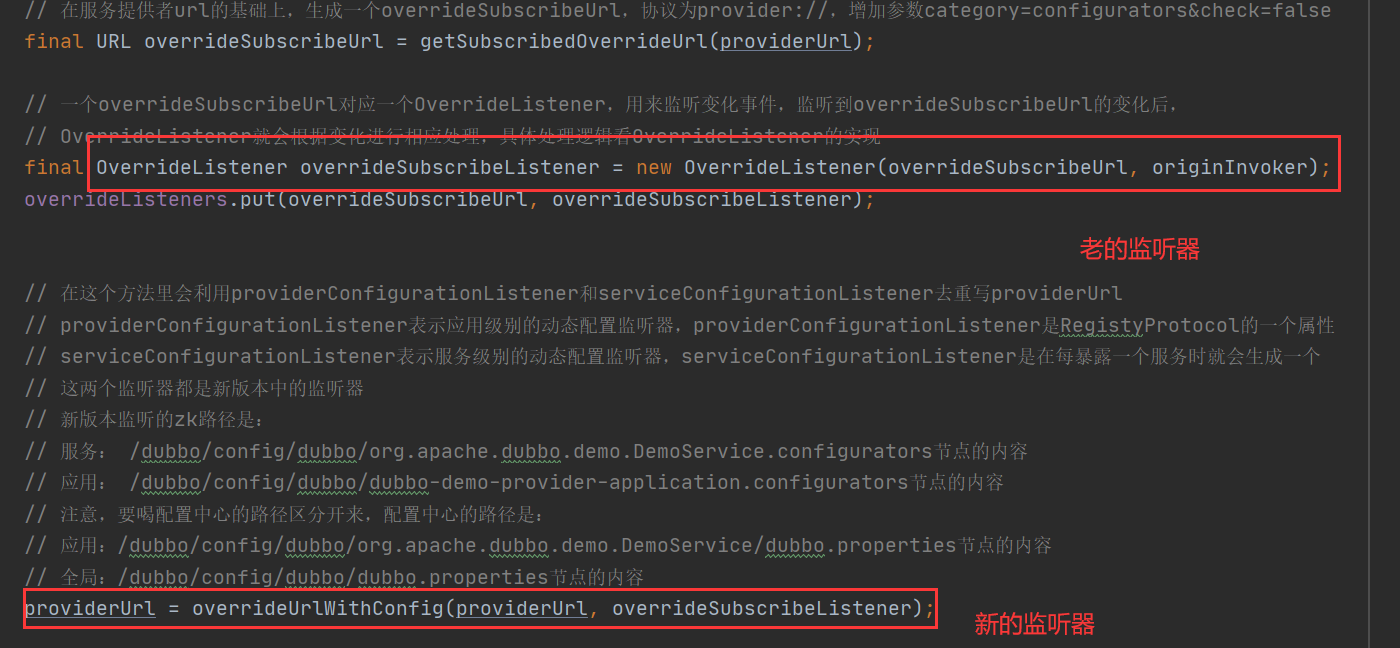
10. 接下来就会使⽤Protocol去export导出服务了，导出之后将得到⼀个Exporter对象（该Exporter对象，可以理解为主要可以⽤来卸载（unexport）服务，什么时候会卸载服务？在优雅关闭Dubbo应⽤的时候）

11. 接下来我们来详细看看Protocol是怎么导出服务的？

12. 但调⽤protocol.export(wrapperInvoker)⽅法时，因为protocol是Protocol接⼝的⼀个Adaptive对象，所以此时会根据wrapperInvoker的genUrl⽅法得到⼀个url，根据此url的协议找到对应的扩展点，此时扩展点就是RegistryProtocol，但是，因为Protocol接⼝有两个包装类，⼀个是ProtocolFilterWrapper、ProtocolListenerWrapper，所以实际上在调⽤export⽅法时，会经过这两个包装类的export⽅法，但是在这两个包装类的export⽅法中都会Registry协议进⾏了判断，不会做过多处理，所以最终会直接调⽤到RegistryProtocol export(Invoker<T> originInvoker)⽅法

13. 在RegistryProtocol的export(Invoker<T> originInvoker)⽅法中，主要完成了以下⼏件事情：

a. ⽣成监听器，监听动态配置中⼼此服务的参数数据的变化，⼀旦监听到变化，则重写服务URL，并且在服务导出时先先获取一次动态配置重写⼀次服务URL



b. 拿到重写之后的URL之后，调⽤doLocalExport()进⾏服务导出，在这个⽅法中就会调⽤

DubboProtocol的export⽅法去导出服务了，导出成功后将得到⼀个ExporterChangeableWrapper （包装该wrapper用于服务注销）

i. 在DubboProtocol的export⽅法中主要要做的事情就是启动NettyServer，并且设置⼀系列的RequestHandler，以便在接收到请求时能依次被这些RequestHandler所处理



在open Server中启动netty，开启netty的调用链比较长createServer(url)------>getExchanger(url).bind(url, handler),HeaderExchanger调用-------->继续调用bind方法直到nettyServer，然后调用doOpen方法

Reset方法在修改动态配置后由监听器触发调用重新导出

ii. 这些RequestHandler在上⽂已经整理过了

c. 从originInvoker中获取注册中⼼的实现类，⽐如ZookeeperRegistry

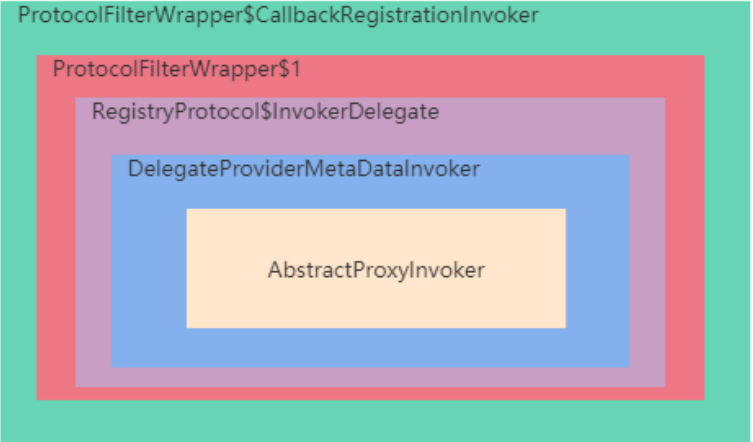
d. 将重写后的服务URL进⾏简化，把不⽤存到注册中⼼去的参数去除

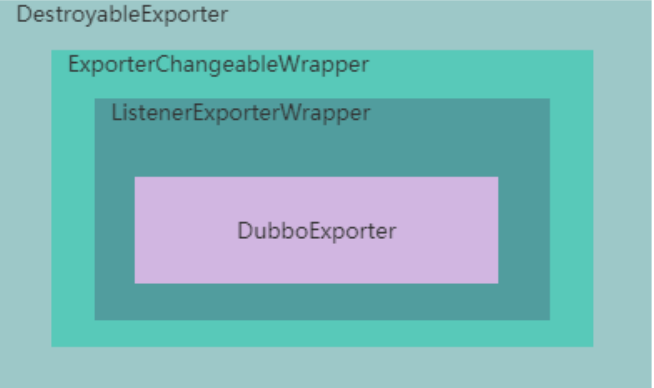
e. 把简化后的服务URL调⽤ZookeeperRegistry.registry()⽅法注册到注册中⼼去

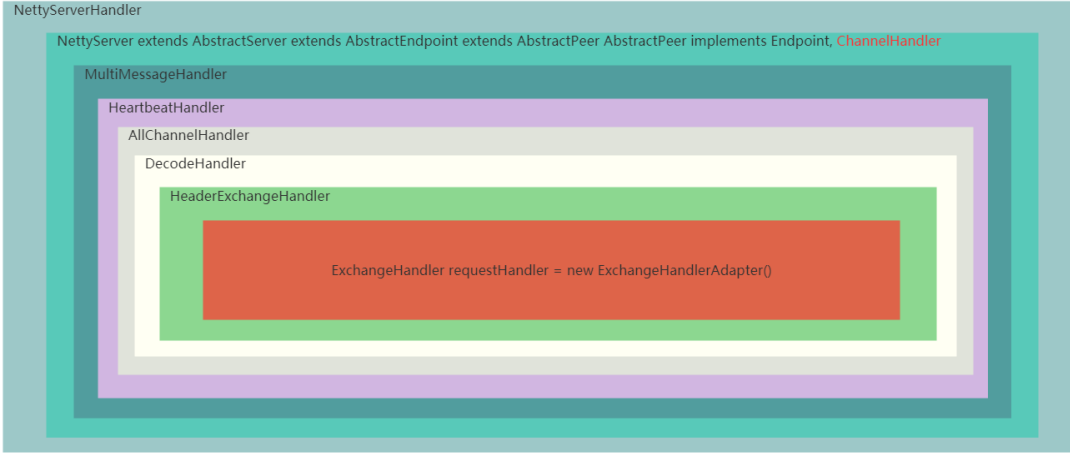
注册时首先跳到FailbackRegistry，之后的doRegistry才会进入ZookeeperRegistry

在注册前会调用前面注册的监听器，先手动触发监听器逻辑，后面才真正的监听ZK

f. 最后将ExporterChangeableWrapper封装为DestroyableExporter对象返回，完成服务导出







## 2.4 服务导入

服务导入的入口在referenceBean的getObject方法

1. 调⽤checkAndUpdateSubConfigs()，检查和更新参数，和服务提供者类似，把ReferenceBean⾥的属性的值更新为优先级最⾼的参数值 （包括配置中心与动态配置）

2. 调⽤init()去⽣成代理对象ref，get()⽅法会返回这个ref

3. 在⽣成代理对象ref之前，先把消费者所引⼊服务设置的参数添加到⼀个map中，等会根据这个map中的参数去从注册中⼼查找服务

4. 把消费者配置的所有注册中⼼获取出来

如果不是使用的injvm协议，则先看@reference注解有没有url参数，没有才会去配置 文件中查找配置中心

a. 如果只有⼀个注册中⼼，那么直接调⽤Protocol的refer(interfaceClass, urls.get(0));得 到⼀个Invoker对象

b. 如果有多个注册中⼼，则遍历每个注册中⼼，分别调⽤Protocol的refer(interfaceClass, url);得到⼀个Invoker对象添加到invokers中，然后把invokers调⽤CLUSTER.join(new

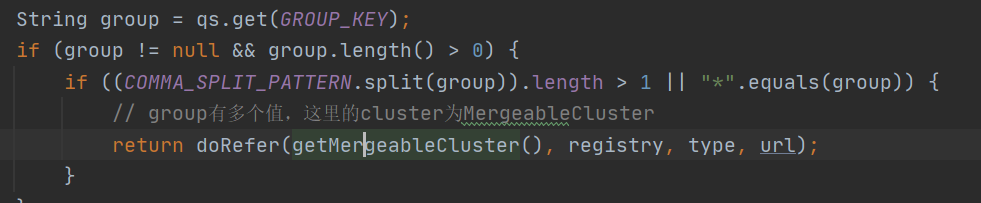
StaticDirectory(u, invokers));封装所有invokers得到⼀个invoker，

5. 把最终得到的invoker对象调⽤PROXY\_FACTORY.getProxy(invoker);得到⼀个代理对象，并返回，这个代理对象就是ref

6. 总结：上⽂的Invoker对象，表示服务执⾏者，从注册中⼼refer下来的是⼀个服务执⾏者，合并invokers后得到的invoker也是⼀个服务执⾏者（抽象范围更⼤了）

接下来，来看Protorol.refer(interfaceClass, url)⽅法是怎么⽣成⼀个Invoker的

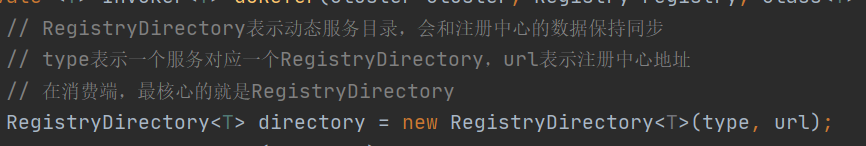
1. ⾸先interfaceClass表示要引⼊的服务接⼝，url是注册中⼼的url（registry://），该url中有⼀个refer参数，参数值为当前所要引⼊服务的参数



当引入参数配置多个分组时则调用dorefer时传入MergeabledCluster否则传默认实现类

2. 调⽤doRefer(cluster, registry, type, url)

3. 在doRefer⽅法中会⽣成⼀个RegistryDirectory



服务目录包含了以下属性

1. serviceType：表示服务接⼝

2. serviceKey：表示引⼊的服务key，serviceclass+version+group

3. queryMap：表示引⼊的服务的参数配置

4. configurators：动态配置

5. routerChain：路由链

6. invokers：表示服务⽬录当前缓存的服务提供者Invoker

7. ConsumerConfigurationListener：监听本应⽤的动态配置

8. ReferenceConfigurationListener：监听所引⼊的服务的动态配置

4. 然后获取新版本中的路由器链，并添加到RegistryDirectory中去

构建路由链在下面介绍

5. RegistryDirectory监听⼏个⽬录（注意，完成监听器的订阅绑定后，会⾃动触发⼀次去获取这些⽬录上的当前数据）

a. 当前所引⼊的服务的动态配置⽬录：

/dubbo/config/dubbo/org.apache.dubbo.demo.DemoService:1.1.1:g1.configurators

b. 当前所引⼊的服务的提供者⽬录：/dubbo/org.apache.dubbo.demo.DemoService/providers

c. 当前所引⼊的服务的⽼版本动态配置⽬录：

/dubbo/org.apache.dubbo.demo.DemoService/configurators

d. 当前所引⼊的服务的⽼版本路由器⽬录：

/dubbo/org.apache.dubbo.demo.DemoService/routers

服务目录订阅的流程directory.subscribe

FailbackRegistry.subscribe-->ZookeeperRegistry.doSubscribe---->notify主动触发监听方法----->AbstractRegistry.notify--------->RegistryDirectory.notify----------->refreshOverrideAndInvoker---->refreshInvoker在该方法内构建invoker

**创建DubboInvoker在下面详解**

6. 调⽤cluster.join(directory)得到⼀个invoker

7. 返回invoker（如果消费者引⼊了多个group中的服务，那么这⾥返回的是new

MergeableClusterInvoker<T>(directory);，否则返回的是new FailoverClusterInvoker<T>

(directory);）

8. 但是，上⾯返回的两个Invoker都会被MockClusterInvoker包装，所以最终返回的是

MockClusterInvoker。

1. 当urls有多个时，配了多个注册中心或者dubbo直连的url，则首先为每个url调用refer方法获取服务提供者，当多个url中有注册中心地址则看图



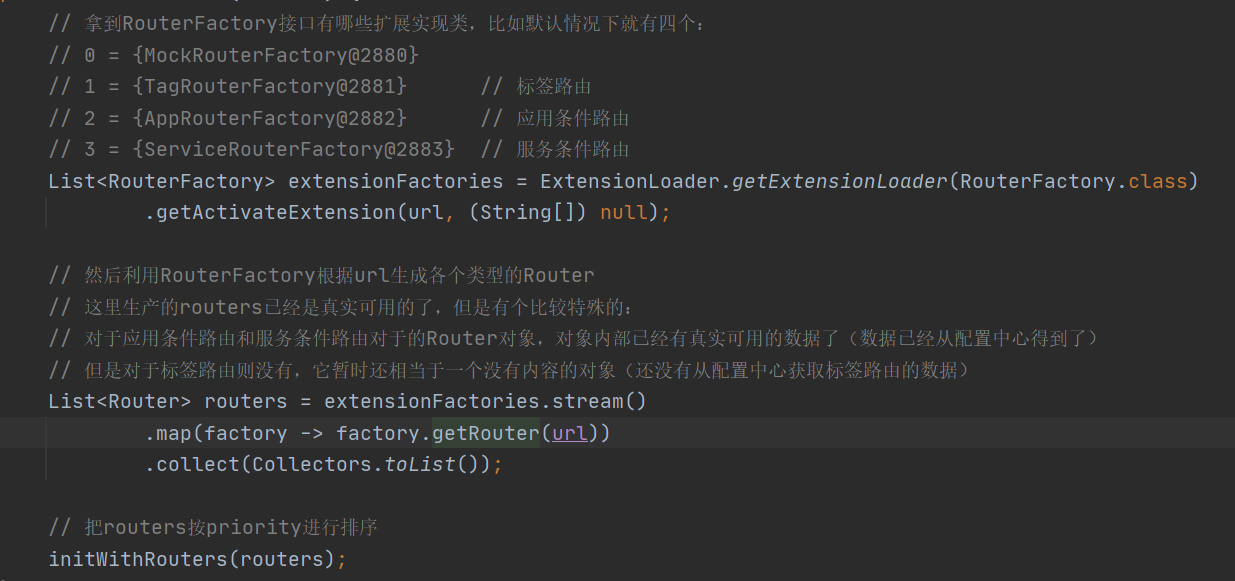
**路由链的构建**

RouterChain.buildChain(url)⽅法赋值得到路由链。

这⾥的url是这样的：consumer://192.168.0.100/org.apache.dubbo.demo.DemoService?

application=dubbo-demo-consumer-application&dubbo=2.0.2&group=g1&interface=org.apache.dubbo.demo.DemoService&lazy=false&methods=sayHello&pid=19852&release=2.7.0&revision=1.1.1&side=consumer&sticky=false&timestamp=1591332529643&version=1.1.1

表示所引⼊的服务的参数，在获得路由链时就要根据这些参数去匹配得到符合当前的服务的Router.



1. RouterChain.buildChain(url)

2. new RouterChain<>(url)

3. List<RouterFactory> extensionFactories =

ExtensionLoader.getExtensionLoader(RouterFactory.class).getActivateExtension(url,

(String[]) null);根据url去获取可⽤的RouterFactory，可以拿到四个：

a. MockRouterFactory：Mock路由，没有order，相当于order=0

b. TagRouterFactory: 标签路由，order = 100

c. AppRouterFactory: 应⽤条件路由，order = 200

d. ServiceRouterFactory: 服务条件路由，order = 300

4. 遍历每个RouterFactory，调⽤getRouter(url)⽅法得到Router，存到List<Router> routers中

5. 对routers按order从⼩到⼤的顺序进⾏排序

AppRouter和ServiceRouter是⾮常类似，他们的⽗类都是ListenableRouter，在创建AppRouter和ServiceRouter时，会绑定⼀个监听器，⽐如：

1. AppRouter监听的是：/dubbo/config/dubbo/dubbo-demo-consumer-application.condition-

router节点的内容

1. ServiceRouter监听的是：
2. /dubbo/config/dubbo/org.apache.dubbo.demo.DemoService:1.1.1:g1.condition-router节

点的内容绑定完监听器之后，会主动去获取⼀下对应节点的内容（也就是所配置的路由规则内容），然后会去解析内容得到ConditionRouterRule routerRule，再调⽤generateConditions(routerRule);⽅法解析出⼀个或多个ConditionRouter，并存⼊到List<ConditionRouter> conditionRouters中。

注意routerRule和conditionRouters是ListenableRouter的属性，就是在AppRouter和ServiceRouter中的。

对于TagRouter就⽐较特殊，⾸先标签路由是⽤在，当消费者在调⽤某个服务时，通过在请求中设置标签，然后根据所设置的标签获得可⽤的服务提供者地址。⽽且⽬前TagRouter只⽀持应⽤级别的配置(⽽且是服务提供者应⽤，给某个服务提供者应⽤打标)。

所以对于服务消费者⽽⾔，在引⽤某个服务时，需要知道提供这个服务的应⽤名，然后去监听这个应⽤名对应的.tag-router节点内容，⽐如

/dubbo/config/dubbo/dubbo-demo-provider-application.tag-router。

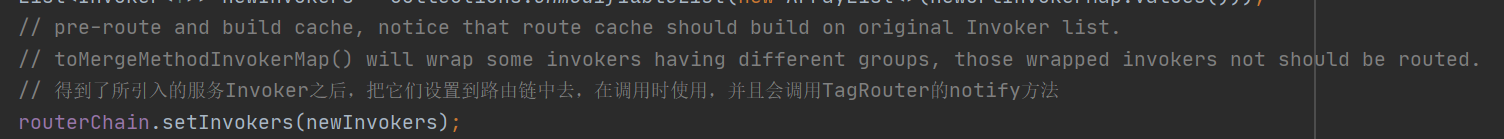
那么问题来了，怎么才能知道提供这个服务的服务提供者的应⽤名呢？

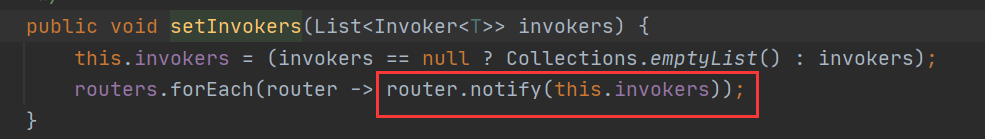
答案是，需要先获取到当前所引⼊服务的服务提供者URL，从URL中得到服务提供者的应⽤名拿到应⽤名之后才能去应⽤名对应的.tag-router节点去绑定监听器。

这就是TagRouter和AppRouter、ServiceRouter的区别，对于AppRouter⽽⾔，监听的是本消费者应⽤的路由规则，对于ServiceRouter⽽⾔，监听的是所引⼊服务的路由规则，都⽐较简单。

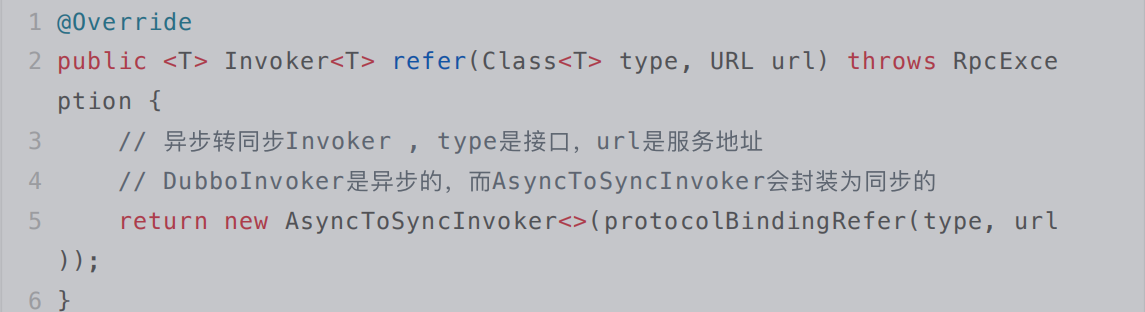
所以，TagRouter是在引⼊服务时，获取到服务的提供者URL之后，才会去监听.tag-router节点中的内容，并⼿动获取⼀次节点中的内容，设置TagRouter对象中tagRouterRule属性，表示标签路由规则。

创建完invoker*后面初始化标签路由*





**创建DubboInvoker**



创建DubboInvoker的调用链比较长入口在RegistryProtocol类里服务目录添加监听里触发最终调用到AbstractProtocol的refer方法，用debug去看调用链

**在这个过程中会去zk获取所有的服务，并将服务使用routerChain进行过滤**

**在这个过程中会调用ProtocolFilterWrapper的构建过滤器的逻辑buildInvokerChain**

调⽤protocolBindingRefer()⽅法得到⼀个Invoker后，会包装为⼀个AsyncToSyncInvoker然后作为refer⽅法的结果返回。

在DubboProtocol的protocolBindingRefer()⽅法中会new⼀个DubboInvoker，然后就返回了。

在构造DubboInvoker时，有⼀个⾮常重要的步骤，构造clients。DubboInvoker作为消费端服务的执⾏者，在调⽤服务时，是需要去发送Invocation请求的，⽽发送请求就需要client，之所以有多个client，是因为DubboProtocol⽀持多个。

假如在⼀个DubboInvoker中有多个Client，那么在使⽤这个DubboInvoker去调⽤服务时，就可以提⾼效率，⽐如⼀个服务接⼝有多个⽅法，那么在业务代码中，可能会不断的调⽤该接⼝中的⽅法，并且由于DubboProtocol底层会使⽤异步去发送请求，所以在每次需要发送请求时，就可以从clients轮询⼀个client去发送这个数据，从⽽提⾼效率。

接下来，来看看clients是如何⽣成的。

1. ⾸先，⼀个DubboInvoker到底⽀持多少个Client呢？这是可以配置的，参数为connections，按指定的数字调⽤initClient(url)得到ExchangeClient。

2. initClient(url)的实现逻辑为

a. 获取client参数，表示是⽤netty还是mina等等

b. 获取codec参数，表示数据的编码⽅式

c. 获取heartbeat参数，表示⻓连接的⼼跳时间，超过这个时间服务端没有收到数据则关闭socket，默认为1分钟

d. 如果所指定的client没有对应的扩展点，则抛异常

e. 获取lazy参数，默认为false，如果为true，那么则直接返回⼀个LazyConnectExchangeClient，

表示真正在发送数据时才建⽴socket

f. 否则调⽤Exchangers.connect(url, requestHandler)获得⼀个client

g. 在connect()⽅法中调⽤HeaderExchanger的connect⽅法去建⽴socket连接并得到⼀个

HeaderExchangeClient

h. 在构造HeaderExchangeClient时需要先执⾏Transporters.connect()⽅法得到⼀个Client

i. 会调⽤NettyTransporter的connect()去构造⼀个NettyClient

j. 在构造NettyClient的过程中，会去初始化Netty的客户端，然后连接Server端，建⽴⼀个Socket

**在服务消费端有⼏个监听器：**

1. ConsumerConfigurationListener：监听本应⽤的动态配置，当应⽤的动态配置发⽣了修改后，会调⽤RegistryDirectory的refreshInvoker()⽅法，对应的路径为："/dubbo/config/dubbo/dubbo-

demo-consumer-application.configurators"

2. ReferenceConfigurationListener：监听所引⼊的服务的动态配置，当服务的动态配置发⽣了修改后，会调⽤RegistryDirectory的refreshInvoker()⽅法，对应的路径

为："/dubbo/config/dubbo/org.apache.dubbo.demo.DemoService:1.1.1:g1.configurators"

3. RegistryDirectory：RegistryDirectory本身也是⼀个监听器，它会监听所引⼊的服务提供者、服务动态配置（⽼版本）、服务路由，路径分别为：

a. "/dubbo/org.apache.dubbo.demo.DemoService/providers"

b. "/dubbo/org.apache.dubbo.demo.DemoService/configurators"

c. "/dubbo/org.apache.dubbo.demo.DemoService/routers"

4. 路由器Router：每个Router⾃⼰本身也是⼀个监听器，负责监听对应的路径

a. AppRouter：应⽤路由，监听的路径为"/dubbo/config/dubbo/dubbo-demo-consumer-

application.condition-router"

b. ServiceRouter: 服务路由，监听的路径

为"/dubbo/config/dubbo/org.apache.dubbo.demo.DemoService:1.1.1:g1.condition- router"

c. TagRouter: 标签路由，标签路由和应⽤路由、服务路由有所区别，应⽤路由和服务路由都是在消费者启动，在构造路由链时会进⾏监听器的绑定，但是标签路由不是消费者启动的时候绑定监听器的，是在引⼊服务时，获取到服务的提供者URL之后，才会去监听.tag-router节点中的内容，监听的路径为

"/dubbo/config/dubbo/dubbo-demo-provider-application.tag-router"

当ConsumerConfigurationListener接收到了消费者应⽤的动态配置数据变化后，会调⽤当前消费者应⽤中的所有RegistryDirectory的refreshInvoker()⽅法，表示刷新消费者应⽤中引⼊的每个服务对应的Invoker

当ReferenceConfigurationListener接收到了某个服务的动态配置数据变化后，会调⽤该服务对应的RegistryDirectory的refreshInvoker()⽅法，表示刷新该服务对应的Invoker

当AppRouter和ServiceRouter接收到条件路由的数据变化后，就会更新Router内部的routerRule和 conditionRouters属性。这两个属性在服务调⽤过程中会⽤到。

当TagRouter接收到标签路由的数据变化后，就会更新TagRouter内部的tagRouterRule的属性，这个属性在服务调⽤过程中会⽤到。

当RegistryDirectory接收到"/dubbo/org.apache.dubbo.demo.DemoService/configurators"节点数据变化后，会⽣成configurators

当RegistryDirectory接收到"/dubbo/org.apache.dubbo.demo.DemoService/routers"节点数据变化后，会⽣成Router并添加到routerChain中

当RegistryDirectory接收到"/dubbo/org.apache.dubbo.demo.DemoService/providers"节点数据变化后，会调⽤refreshOverrideAndInvoker()⽅法。这个⽅法就是⽤来针对每个服务提供者来⽣成Invoker的。

1. refreshOverrideAndInvoker⽅法中⾸先调⽤overrideDirectoryUrl()⽅法利⽤Configurators重写⽬录地址，⽬录地址是这样的：

zookeeper://127.0.0.1:2181/org.apache.dubbo.registry.RegistryService?application=dubbo-

demo-consumer-application&dubbo=2.0.2&group=g1&interface=org.apache.dubbo.demo.DemoService&lazy=false&methods=sayHello&pid=49964&register.ip=192.168.40.17&release=2.7.0&revision=1.1.1 &side=consumer&sticky=false&timestamp=1591339005022&version=1.1.1，在注册中⼼URL基础上把当前引⼊服务的参数作为URL的Parameters，所以这个地址既包括了注册中⼼的信息，也包括了当前引⼊服务的信息

2. 利⽤⽼版本configurators，Consumer应⽤的configurators，引⼊的服务的configurators去重写⽬录地址。

3. 重写往⽬录地址后，调⽤refreshInvoker(urls)⽅法去刷新Invoker

4. 在refreshInvoker(urls)⽅法中会把从注册中⼼获取到的providers节点下的服务URL，调⽤

toInvokers(invokerUrls)⽅法得到Invoker

5. 先按Protocol进⾏过滤，并且调⽤DubboProtocol.refer⽅法得到Invoker

6. 将得到的invokers设置到RouterChain上，并且调⽤RouterChain上所有的routers的

notify(invokers)⽅法，实际上这⾥只有TagRouter的notify⽅法有⽤

1. 再把属于同⼀个group中的invoker合并起来

## 2.5 服务调用

Dubbo整个调用过程

https://www.processon.com/view/link/60110b847d9c08426cf10e49

### 5.1 消费端调用

第一步调用InvokerInvocationHandler的invoke方法



1. MockClusterInvoker.invoke(new RpcInvocation(method, args))：Mock逻辑

2. AbstractClusterInvoker.invoke(invocation)：把RpcContext中设置的Attachments添加到

invocation对象上，此处即为隐式传参，消费端可以通过开发cluster Wrapper类实现隐式传参，调⽤路由链从服务⽬录上筛选出适合的服务Invoker，获得服务均衡策略 loadbalance

3. FailoverClusterInvoker.doInvoke(invocation, invokers, loadbalance)：根据负载均衡策略选出⼀

个invoker，然后执⾏

4. InvokerWrapper.invoke(invocation)：没做什么事情

5. CallbackRegistrationInvoker.invoke(invocation)：开始执⾏Filter链，执⾏完得到结果后，会获取ListenableFilter中的listener，执⾏listener的onResponse⽅法

6. ConsumerContextFilter.invoke(invocation)：设置RpcContext中LocalAddress、

RemoteAddress、RemoteApplicationName参数

7. FutureFilter.invoke(invocation)：

8. MonitorFilter.invoke(invocation)：⽅法的执⾏次数+1

9. ListenerInvokerWrapper.invoke(invocation)：没做什么事情

10. AsyncToSyncInvoker.invoke(invocation)：异步转同步，会先⽤下层Invoker去异步执⾏，然后阻塞Integer.MAX\_VALUE时间，直到拿到了结果

11. AbstractInvoker.invoke(invocation)：主要调⽤DubboInvoker的doInvoke⽅法，如果doInvoker⽅法出现了异常，会进⾏包装，包装成AsyncRpcResult

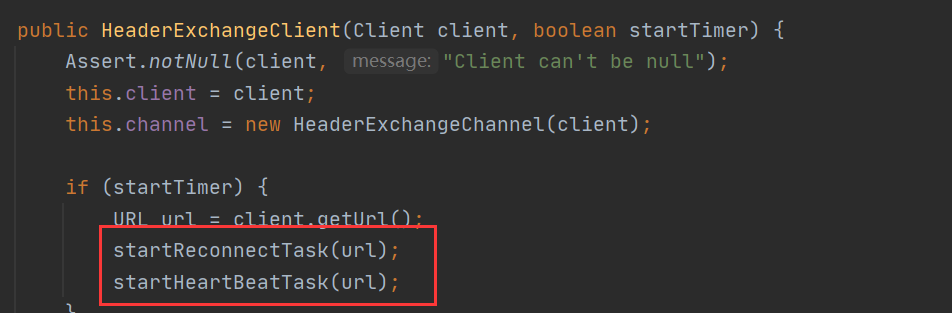
12. DubboInvoker.doInvoke(invocation)：从clients轮询出⼀个client进⾏数据发送，如果配置了不关⼼结果，则调⽤ReferenceCountExchangeClient的send⽅法，否则调⽤

ReferenceCountExchangeClient的request⽅法

13. ReferenceCountExchangeClient.request(Object request, int timeout)：没做什么事情

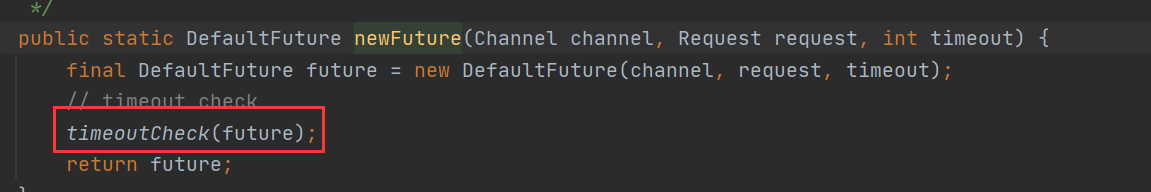
14. HeaderExchangeClient.request(Object request, int timeout)：没做什么事情

构造方法中添加心跳与重连任务，当使用的网络框架本身有心跳功能则此处不会添加心跳任务



15. HeaderExchangeChannel.request(Object request, int timeout)：构造⼀个Request对象，并且会构造⼀个DefaultFuture对象来阻塞timeout的时间来等待结果，在构造DefaultFuture对象时，会把DefaultFuture对象和req的id存⼊FUTURES中，FUTURES是⼀个Map，当

HeaderExchangeHandler接收到结果时，会从这个Map中根据id获取到DefaultFuture对象，然后返回Response。



创建DefaultFuture对象时会添加调用超时检测任务

16. AbstractPeer.send(Object message)：从url中获取send参数，默认为false

17. AbstractClient.send(Object message, boolean sent)：没做什么

18. NettyChannel.send(Object message, boolean sent)：调⽤NioSocketChannel的writeAndFlush

发送数据，然后判断send如果是true，那么则阻塞url中指定的timeout时间，因为如果send是false，在HeaderExchangeChannel中会阻塞timeout时间

19. NioSocketChannel.writeAndFlush(Object msg)：最底层的Netty⾮阻塞式的发送数据

总结⼀下上⾯调⽤流程：

1. 最外层是Mock逻辑，调⽤前，调⽤后进⾏Mock

2. 从服务⽬录中，根据当前调⽤的⽅法和路由链，筛选出部分服务Invoker（DubboInvoker）

3. 对服务Invoker进⾏负载均衡，选出⼀个服务Invoker

4. 执⾏Filter链

5. AsyncToSyncInvoker完成异步转同步，因为DubboInvoker的执⾏是异步⾮阻塞的，所以如果是同步调⽤，则会在此处阻塞，知道拿到响应结果

6. DubboInvoker开始异步⾮阻塞的调⽤

7. HeaderExchangeChannel中会阻塞timeout的时间来等待结果，该timeout就是⽤户在消费端所配置

### 5.2 服务端调用逻辑

1. NettyServerHandler：接收数据

2. MultiMessageHandler：判断接收到的数据是否是MultiMessage，如果是则获取MultiMessage中的单个Message，传递给HeartbeatHandler进⾏处理

3. HeartbeatHandler：判断是不是⼼跳消息，如果不是则把Message传递给AllChannelHandler

4. AllChannelHandler：把接收到的Message封装为⼀个ChannelEventRunnable对象，扔给线程池进⾏处理 ，此处很重要，dubbo的经典线程模型，dubbo内的线程均为io线程，业务交由新的线程处理

5. ChannelEventRunnable：在ChannelEventRunnable的run⽅法中会调⽤DecodeHandler处理

Message

6. DecodeHandler：按Dubbo协议的数据格式，解析当前请求的path，versio，⽅法，⽅法参数等等，然后把解析好了的请求交给HeaderExchangeHandler

7. HeaderExchangeHandler：处理Request数据，⾸先构造⼀个Response对象，然后调⽤

ExchangeHandlerAdapter得到⼀个CompletionStage future，然后给future通过whenComplete绑定⼀个回调函数，当future执⾏完了之后，就可以从回调函数中得到ExchangeHandlerAdapter的执⾏结果，并把执⾏结果设置给Response对象，通过channel发送出去。

8. ExchangeHandlerAdapter：从本机已经导出的Exporter中根据当前Request所对应的服务key，去寻找Exporter对象，从Exporter中得到Invoker，然后执⾏invoke⽅法，此Invoker为

ProtocolFilterWrapper$CallbackRegistrationInvoker

9. ProtocolFilterWrapper$CallbackRegistrationInvoker：负责执⾏过滤器链，并且在执⾏完了之后回调每个过滤器的onResponse或onError⽅法

10. EchoFilter：判断当前请求是不是⼀个回升测试，如果是，则不继续执⾏过滤器链了（服务实现者Invoker也不会调⽤了）

11. ClassLoaderFilter：设置当前线程的classloader为当前要执⾏的服务接⼝所对应的classloader

12. GenericFilter：把泛化调⽤发送过来的信息包装为RpcInvocation对象

13. ContextFilter：设置RpcContext.getContext()的参数

14. TraceFilter：先执⾏下⼀个invoker的invoke⽅法，调⽤成功发送调⽤信息

15. TimeoutFilter：调⽤时没有特别处理，只是记录了⼀下当前时间，当整个filter链都执⾏完了之后回调TimeoutFilter的onResponse⽅法时，会判断本次调⽤是否超过了timeout

16. MonitorFilter：记录当前服务的执⾏次数

17. ExceptionFilter：调⽤时没有特别处理，在回调onResponse⽅法时，对不同的异常进⾏处理，详解Dubbo的异常处理

18. DelegateProviderMetaDataInvoker：过滤器链结束，调⽤下⼀个Invoker

19. AbstractProxyInvoker：在服务导出时，根据服务接⼝，服务实现类对象⽣成的，它的invoke⽅法就会执⾏服务实现类对象的⽅法，得到结果

### 5.3 Dubbo的异常处理

当服务消费者在调⽤⼀个服务时，服务提供者在执⾏服务逻辑时可能会出现异常，对于Dubbo来说，服务消费者需要在消费端抛出这个异常，那么这个功能是怎么做到的呢？

服务提供者在执⾏服务时，如果出现了异常，那么框架会把异常捕获，捕获异常的逻辑在

AbstractProxyInvoker中，捕获到异常后，会把异常信息包装为正常的AppResponse对象，只是AppResponse的value属性没有值，exception属性有值。

此后，服务提供者会把这个AppResponse对象发送给服务消费端，服务消费端是在

InvokerInvocationHandler中调⽤AppResponse的recreate⽅法重新得到⼀个结果，在recreate⽅法中会去失败AppResponse对象是否正常，也就是是否存在exception信息，如果存在，则直接throw这个exception，从⽽做到服务执⾏时出现的异常，在服务消费端抛出。

那么这⾥存在⼀个问题，如果服务提供者抛出的异常类，在服务消费者这边不存在，那么服务消费者也就抛不出这个异常了，那么dubbo是怎么处理的呢？

这⾥就涉及到了ExceptionFilter，它是服务提供者端的⼀个过滤器，它主要是在服务提供者执⾏完服务后会去识别异常：

1. 如果是需要开发⼈员捕获的异常，那么忽略，直接把这个异常返回给消费者

2. 如果在当前所执⾏的⽅法签名上有声明，那么忽略，直接把这个异常返回给消费者

3. 如果抛出的异常不需要开发⼈员捕获，或者⽅法上没有申明，那么服务端或记录⼀个error⽇志

4. 异常类和接⼝类在同⼀jar包⾥，那么忽略，直接把这个异常返回给消费者

5. 如果异常类是JDK⾃带的异常，那么忽略，直接把这个异常返回给消费者

6. 如果异常类是Dubbo⾃带的异常，那么忽略，直接把这个异常返回给消费者

7. 否则，把异常信息包装成RuntimeException，并覆盖AppResponse对象中的exception属性