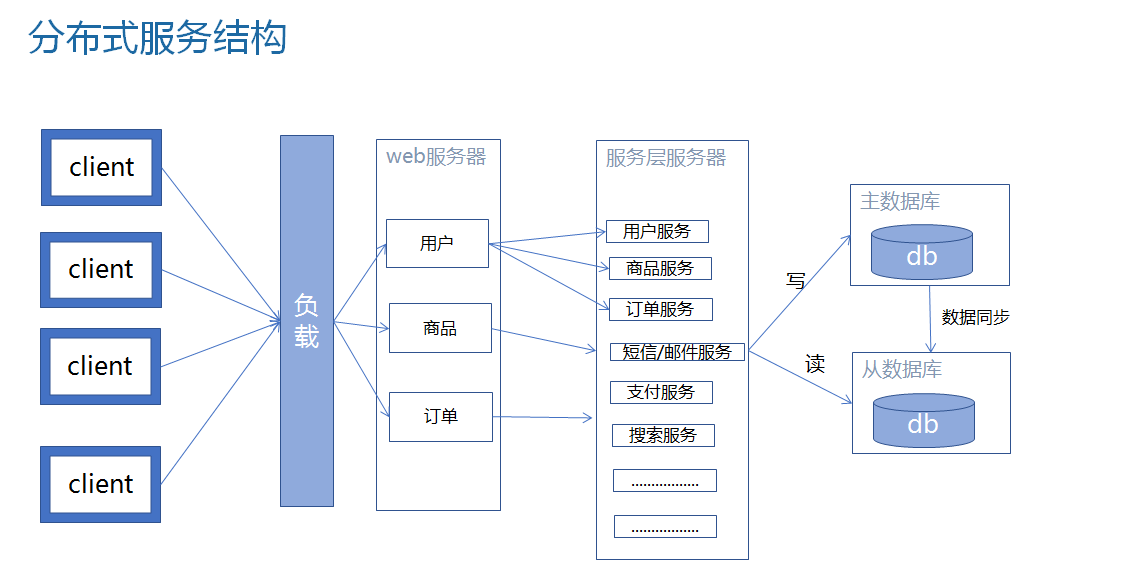
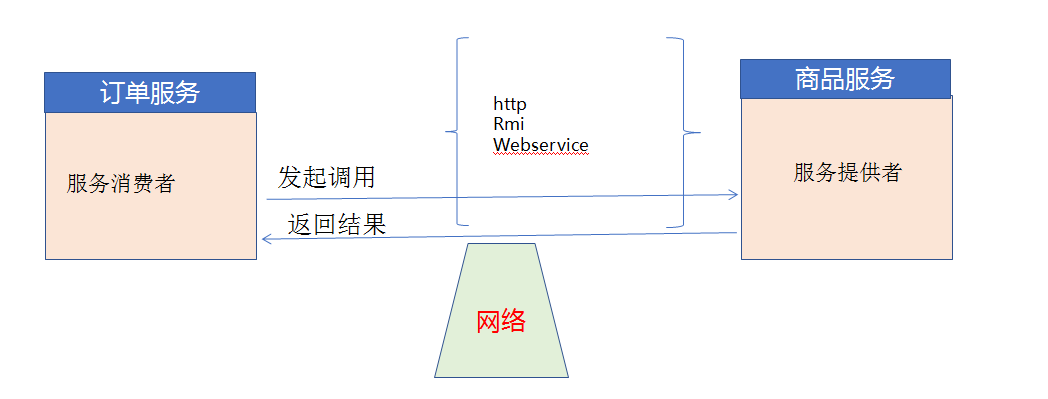
# RPC场景和过程

## RPC场景

在微服务环境下，存在大量的跨JVM进行方法调用的场景，如下图：



具体到某一个调用来说，希望A机器能通过网络，调用B机器内的某个服务方法，并得到返回值：



## RPC的实现切入口

从本质上来讲，某个jvm内的对象方法，是无法在jvm外部被调用的，如下图的代码：

OrderService orderService = (OrderService) ctx.getBean("orderService");  
OrderEntiry entiry = orderService.getDetail("1");

orderService.getDetail("1")的这一句调用，是无法脱离本地jvm环境被调用的。但是，好在java中的对象方法的调用，还有反射模式的调用：

Method method = target.getClass().getMethod(methodName, argTypes);  
return method.invoke(target, args);

只要我们传入反射需要的目标对象**orderService，**方法名称**getDetail**和参数值**"1",**反射就能将**orderService.getDetail**调用起来

于是，我们可以把调用某个方法的过程，改造成这样

Map<String,String> info = new HashMap();  
info.put("target","orderService");  
info.put("methodName","getDetail");  
info.put("arg","1");

//反射调用  
Object result = InvokeUtils.call(info,ctx);

现在，只要谁告诉我了反射需要信息，target/method/arg，我就能调用本地的任何对象方法了

## 网络通信传递反射信息

在上一节的步骤中，本地方法已经宣称，只要传递给他target/method/arg，它就能帮助我们执行想要的目标服务方法，那么现在，我们只需要解决target/method/arg三种信息的网络传输问题。

网络通信的方法很多，如http/rmi/webservice等等，我们只需要选用任意一种即可，为简便起见，我们选用jdk的rmi方式，其使用方式如下：

### 定义一个继承自remote接口的类

public interface InfoService extends Remote {//继承remote接口  
 String RMI\_URL = "rmi://127.0.0.1:9080/InfoService";  
 int port = 9080;  
 Object passInfo(Map<String,String> info) throws RemoteException;  
}

创建一个实现类（为简化实现，继承UnicastRemoteObject类）

public class InfoServiceImpl extends UnicastRemoteObject implements InfoService {  
 public InfoServiceImpl() throws RemoteException {  
 super();  
 }  
 @Override  
 public Object passInfo(Map<String, String> info) {  
 System.out.println("恭喜你，调通了，参数："+JSON.toJSONString(info));  
 info.put("msg","你好，调通了！");  
 return info;  
 }  
}

### RMI开放服务到指定URL

只需要将实例绑定注册到指定的URL和port上,远程即可调用此实例

InfoService infoService = new InfoServiceImpl();  
//注冊通讯端口  
LocateRegistry.createRegistry(9080);  
//注冊通讯路径  
Naming.bind("rmi://127.0.0.1:9080/InfoService

", infoService);

### RMI远程通过URL连接并调用

//取远程服务实现  
infoService = (InfoService) Naming.lookup(InfoService.RMI\_URL);  
//呼叫远程反射方法  
Map<String,String> info = new HashMap();  
info.put("target","orderService");  
info.put("methodName","getDetail");  
info.put("arg","1");  
Object result = infoService.passInfo(info);

至此，已经实现了通过RMI跨机器传递target/method/arg

## 远程调用的融合

可以看到，前两步，已经实现了跨机器的反射信息收发和反射动作调用。我们现在只需要在InfoService的实现上，对传递过来的info信息，直接发起反射调用即可

InfoService infoService = new InfoServiceImpl(){  
 public Object passInfo(Map<String, String> info) {//对象，方法，参数  
  
 super.passInfo(info);//info内包含的信息，是反射需要的信息  
 Object result = InvokeUtils.call(info,ctx);  
 System.out.println("测试InvokeUtils.call调用功能，调用结果：" + JSON.toJSONString(result));  
 return result;  
 }  
};

这样，远程机器只要通过infoService 传递信息过来，就自动将目标服务反射调用，并返回结果值回去，整个RPC过程完成

## 对客户端友好的透明化封装

虽然在上一节中，RPC的整个调用链条已经拉通，但是我们发现还有一个易出错的地方，就是客户端封装反射信息的地方，功能不够内聚，容易出现手误，代码易读性也很差

//呼叫远程反射方法  
Map<String,String> info = new HashMap();  
info.put("target","orderService");  
info.put("methodName","getDetail");  
info.put("arg","1");  
Object result = infoService.passInfo(info);

有什么改善的办法呢？

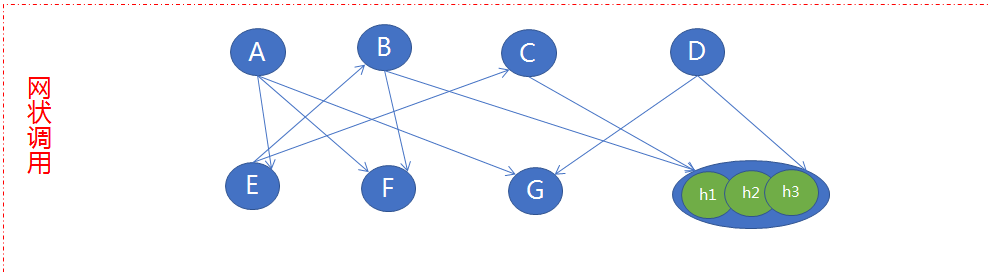
仔细核对，其实target/method/arg这三个信息，全部都可以从接口方法调用OrderService.getDetail("1")中得到，于是，我们可以为此接口做一个代理对象，在代理对象内部完成反射信息的包装：

OrderService service = new OrderService(){  
 @Override  
 public OrderEntiry getDetail(String id) {  
 Map<String,String> info = new HashMap();  
 //写死了反射的目标，静态代理  
 info.put("target","orderService");//对象  
 info.put("methodName","getDetail");//方法  
 info.put("arg",id);//参数  
  
 OrderEntiry result = null;  
 try {  
 result = (OrderEntiry)infoService.passInfo(info);  
 } catch (RemoteException e) {  
 e.printStackTrace();  
 }  
 return result;  
 }  
};

大功告成，从此，客户端远程传递反射信息的过程，直接变成调用接口的代理对象即可。调用者，甚至不再需要区分，此接口代理对象到底是谁，像调用正常的本地服务一起使用就ok

## Dubbo使命

在上面的章节，我们重点详述了，一个具体的RPC调用的全过程。那么在现实工作中，服务节点间的RPC调用是非常普遍并且错踪复杂的



我们除了要关心RPC的过程实现，还需要考虑：

1.服务方是集群时，如何挑选一台机器来响应客户端？

2.因网络抖动引起的调用失败，如何重试来弥补？

3.服务方机器的动态增减，如何能够让客户端及时了解到并做出调整？

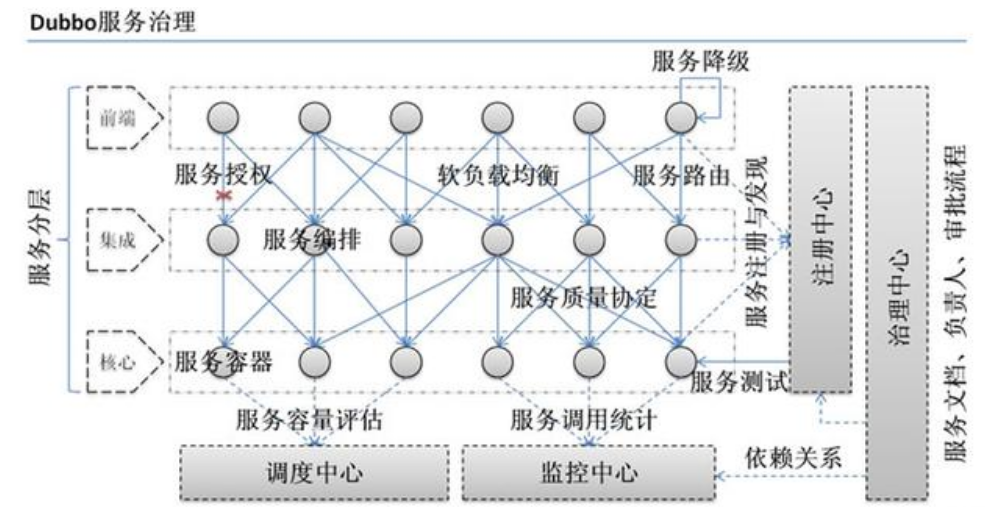
.....

Dubbo的使命，即是解决上述围绕RPC过程的一览子问题

# Dubbo简介

在分布式服务架构下，各个服务间的相互rpc调用会越来越复杂。最终形成网状结构，此时服务的治理极为关键。

Dubbo是一个带有服务治理功能的RPC框架，提供了一套较为完整的服务治理方案，其底层直接实现了rpc调用的全过程，并尽力做事rpc远程对使用者透明。下图展示了Dubbo服务治理的功能。



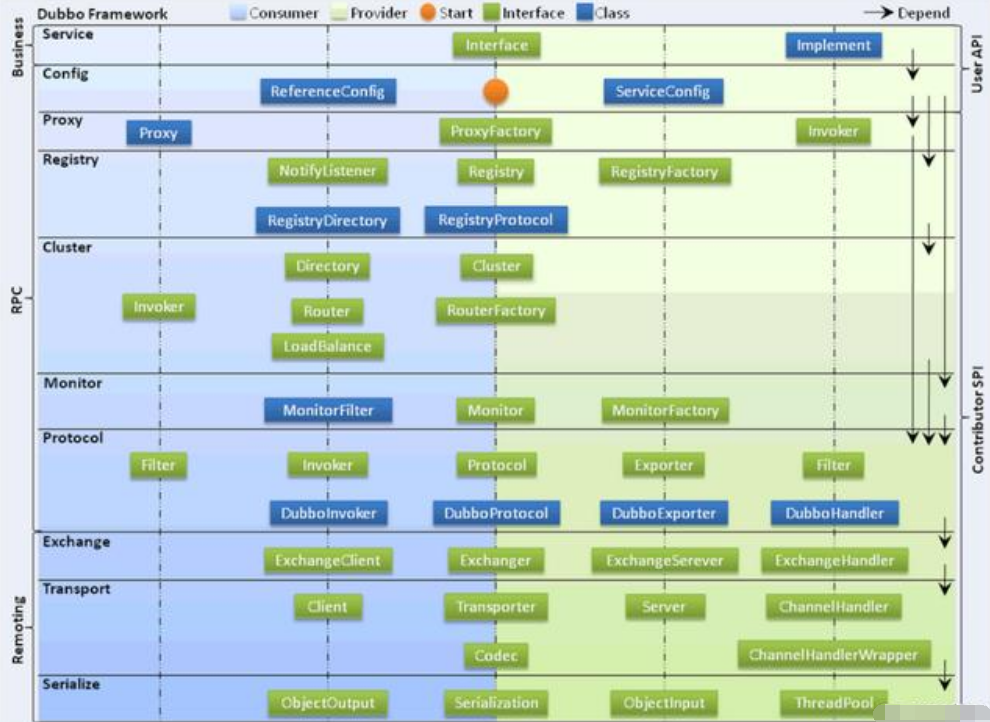
简单的说，Dubbo就是个服务调用的框架，如果没有分布式的需求，其实是不需要用的，只有在分布式的时候，才有使用Dubbo这样的分布式服务框架的需求，并且本质上是个服务调用的东东。

其核心部分包含:

* 远程通讯：提供对多种基于长连接的NIO框架抽象封装，包括多种线程模型、序列化以及“请求-响应”模式的信息交换方式。
* 集群容错：提供基于接口方法的透明远程过程调用，包括多协议支持以及软负载均衡，失败容错、地址路由、动态配置等集群支持。
* 自动发现：基于注册中心目录服务，使服务消费方能动态的查×××提供方，使地址透明，使服务提供方可以平滑增加或减少机器。

## dubbo的架构及特点

下图展示了dubbo的整体结构



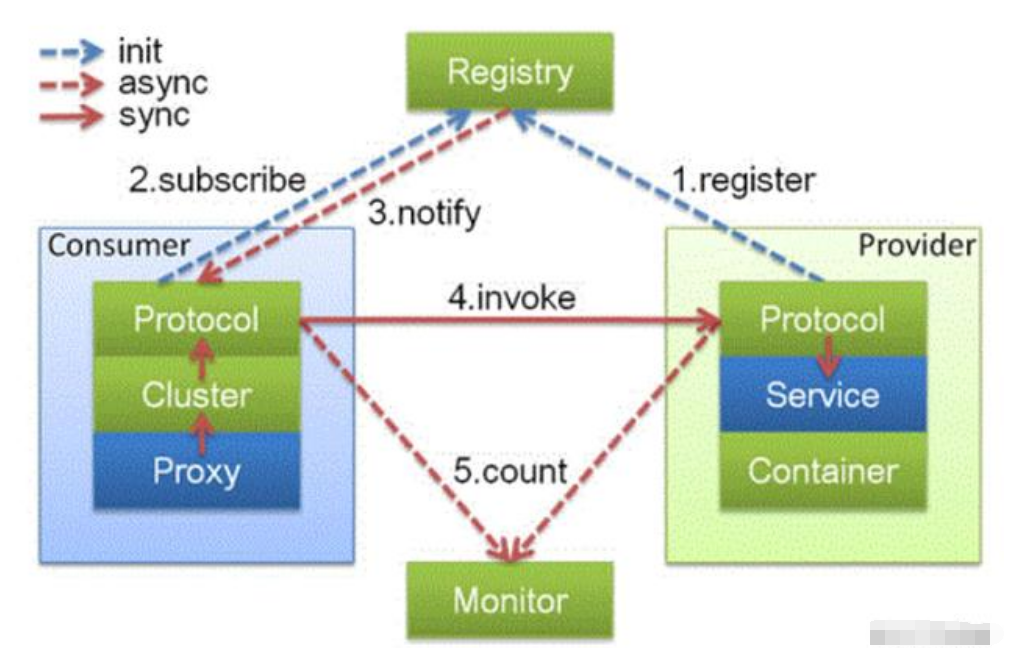
Dubbo总体架构设计一共划分了10层，而最上面的Service层是留给实际想要使用Dubbo开发分布式服务的开发者实现业务逻辑的接口层。图中左边淡蓝背景的为服务消费方使用的接口，右边淡绿色背景的为服务提供方使用的接口，位于中轴线上的为双方都用到的接口。

* 服务接口层(Service)：该层是与实际业务逻辑相关的，根据服务提供方和服务消费方的业务设计对应的接口和实现。
* 配置层(Config)：对外配置接口，以ServiceConfig和ReferenceConfig为中心，可以直接new配置类，也可以通过Spring解析配置生成配置类。
* 服务代理层(Proxy)：服务接口透明代理，生成服务的客户端Stub和服务器端Skeleton，以ServiceProxy为中心，扩展接口为ProxyFactory。
* 服务注册层(Registry)：封装服务地址的注册与发现，以服务URL为中心，扩展接口为RegistryFactory、Registry和RegistryService。可能没有服务注册中心，此时服务提供方直接暴露服务。
* 集群层(Cluster)：封装多个提供者的路由及负载均衡，并桥接注册中心，以Invoker为中心，扩展接口为Cluster、Directory、Router和LoadBalance。将多个服务提供方组合为一个服务提供方，实现对服务消费方透明，只需要与一个服务提供方进行交互。
* 监控层(Monitor)：RPC调用次数和调用时间监控，以Statistics为中心，扩展接口为MonitorFactory、Monitor和MonitorService。
* 远程调用层(Protocol)：封将RPC调用，以Invocation和Result为中心，扩展接口为Protocol、Invoker和Exporter。Protocol是服务域，它是Invoker暴露和引用的主功能入口，它负责Invoker的生命周期管理。Invoker是实体域，它是Dubbo的核心模型，其他模型都向它靠扰，或转换成它，它代表一个可执行体，可向它发起invoke调用。它有可能是一个本地的实现，也可能是一个远程的实现，也可能是一个集群实现。
* 信息交换层(Exchange)：封装请求响应模式，同步转异步，以Request和Response为中心，扩展接口为Exchanger、ExchangeChannel、ExchangeClient和ExchangeServer。
* 网络传输层(Transport)：抽象mina和netty为统一接口，以Message为中心，扩展接口为Channel、Transporter、Client、Server和Codec。
* 数据序列化层(Serialize)：可复用的一些工具，扩展接口为Serialization、 ObjectInput、ObjectOutput和ThreadPool。

从上图可以看出，Dubbo对于服务提供方和服务消费方，从框架的10层中分别提供了各自需要关心和扩展的接口，构建整个服务生态系统(服务提供方和服务消费方本身就是一个以服务为中心的)。

## Dubbo服务的角色关系

服务提供方和服务消费方之间的调用关系，如图所示：



节点角色说明：



调用关系说明：

0：服务容器负责启动，加载，运行服务提供者。

1：服务提供者在启动时，向注册中心注册自己提供的服务。

2：服务消费者在启动时，向注册中心订阅自己所需的服务。

3：注册中心返回服务提供者地址列表给消费者，如果有变更，注册中心将基于长连接推送变更数据给消费者。

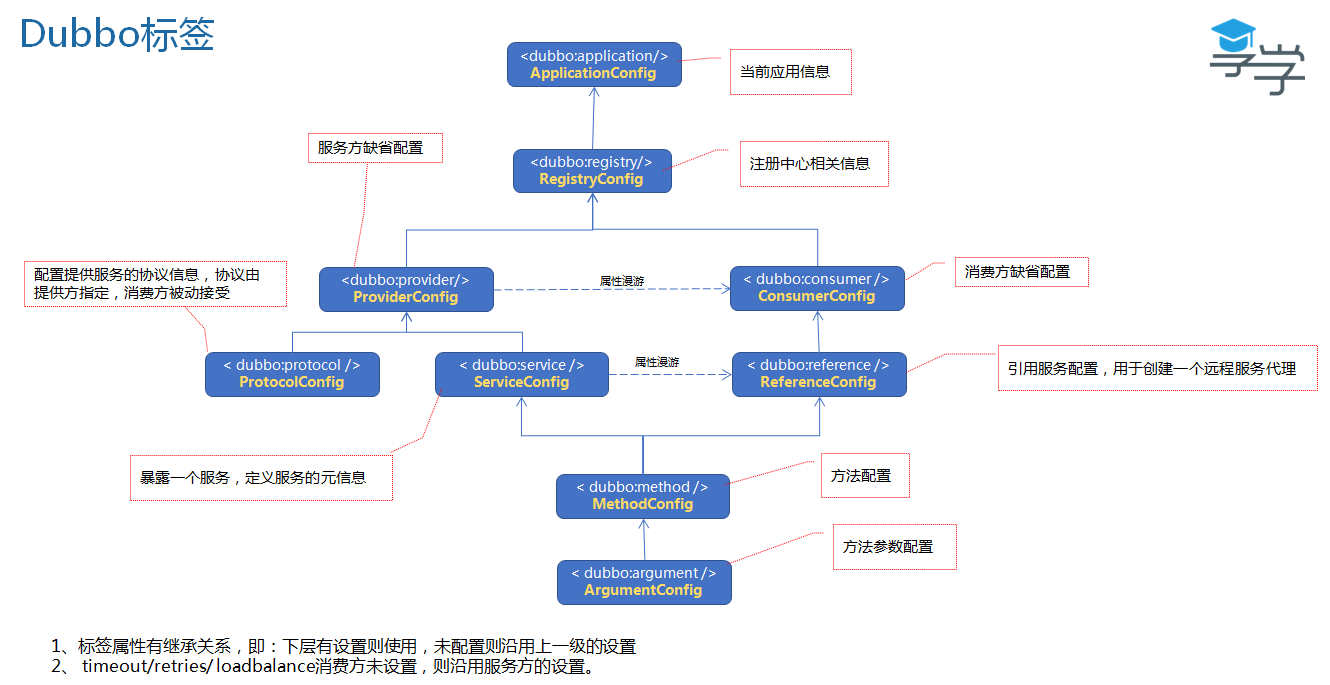
4：服务消费者，从提供者地址列表中，基于软负载均衡算法，选一台提供者进行调用，如果调用失败，再选另一台调用。

5：服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

# Dubbo的基础配置使用

## xml配置方式

### dubbo功能标签集



<dubbo:service/> 服务配置，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:reference/> 引用配置，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:protocol/> 协议配置，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:application/> 应用配置，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/> 注册中心配置，用于配置连接注册中心相关信息。  
<dubbo:module/> 模块配置，用于配置当前模块信息，可选。  
<dubbo:monitor/> 监控中心配置，用于配置连接监控中心相关信息，可选。  
<dubbo:provider/> 提供方的缺省值，当ProtocolConfig和ServiceConfig某属性没有配置时，采用此缺省值，可选。  
<dubbo:consumer/> 消费方缺省配置，当ReferenceConfig某属性没有配置时，采用此缺省值，可选。  
<dubbo:method/> 方法配置，用于ServiceConfig和ReferenceConfig指定方法级的配置信息。  
<dubbo:argument/> 用于指定方法参数配置。  
dubbo标签的使用以及标签属性详解

其实

### 标签详解

所有配置项分为三大类，参见下表中的"作用"一列。

* 服务发现：表示该配置项用于服务的注册与发现，目的是让消费方找到提供方。
* 服务治理：表示该配置项用于治理服务间的关系，或为开发测试提供便利条件。
* 性能调优：表示该配置项用于调优性能，不同的选项对性能会产生影响。

所有配置最终都将转换为URL表示，并由服务提供方生成，经注册中心传递给消费方，各属性对应URL的参数，参见配置项一览表中的"对应URL参数"列。

注意：只有group，interface，version是服务的匹配条件，三者决定是不是同一个服务，其它配置项均为调优和治理参数。

#### <dubbo:service/>

服务提供者暴露服务配置：  
配置类：com.alibaba.dubbo.config.ServiceConfig

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **标签** | **属性** | **对应URL参数** | **类型** | **是否必填** | **缺省值** | **作用** | **描述** | **兼容性** |
| <dubbo:service> | interface |  | class | 必填 |  | 服务发现 | 服务接口名 | 1.0.0以上版本 |
| <dubbo:service> | ref |  | object | 必填 |  | 服务发现 | 服务对象实现引用 | 1.0.0以上版本 |
| <dubbo:service> | version | version | string | 可选 | 0.0.0 | 服务发现 | 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 | 1.0.0以上版本 |
| <dubbo:service> | group | group | string | 可选 |  | 服务发现 | 服务分组，当一个接口有多个实现，可以用分组区分 | 1.0.7以上版本 |
| <dubbo:service> | path | <path> | string | 可选 | 缺省为接口名 | 服务发现 | 服务路径 (注意：1.0不支持自定义路径，总是使用接口名，如果有1.0调2.0，配置服务路径可能不兼容) | 1.0.12以上版本 |
| <dubbo:service> | delay | delay | int | 可选 | 0 | 性能调优 | 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 | 1.0.14以上版本 |
| <dubbo:service> | timeout | timeout | int | 可选 | 1000 | 性能调优 | 远程服务调用超时时间(毫秒) | 2.0.0以上版本 |
| <dubbo:service> | retries | retries | int | 可选 | 2 | 性能调优 | 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 | 2.0.0以上版本 |
| <dubbo:service> | connections | connections | int | 可选 | 100 | 性能调优 | 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 | 2.0.0以上版本 |
| <dubbo:service> | loadbalance | loadbalance | string | 可选 | random | 性能调优 | 负载均衡策略，可选值：random,roundrobin,leastactive，分别表示：随机，轮循，最少活跃调用 | 2.0.0以上版本 |
| <dubbo:service> | async | async | boolean | 可选 | false | 性能调优 | 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 | 2.0.0以上版本 |
| <dubbo:service> | stub | stub | class/boolean | 可选 | false | 服务治理 | 设为true，表示使用缺省代理类名，即：接口名 + Local后缀，服务接口客户端本地代理类名，用于在客户端执行本地逻辑，如本地缓存等，该本地代理类的构造函数必须允许传入远程代理对象，构造函数如：public XxxServiceLocal(XxxService xxxService) | 2.0.0以上版本 |
| <dubbo:service> | mock | mock | class/boolean | 可选 | false | 服务治理 | 设为true，表示使用缺省Mock类名，即：接口名 + Mock后缀，服务接口调用失败Mock实现类，该Mock类必须有一个无参构造函数，与Local的区别在于，Local总是被执行，而Mock只在出现非业务异常(比如超时，网络异常等)时执行，Local在远程调用之前执行，Mock在远程调用后执行。 | 2.0.0以上版本 |
| <dubbo:service> | token | token | string/boolean | 可选 | false | 服务治理 | 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，否则使用静态令牌，令牌的作用是防止消费者绕过注册中心直接访问，保证注册中心的授权功能有效，如果使用点对点调用，需关闭令牌功能 | 2.0.0以上版本 |
| <dubbo:service> | registry |  | string | 可选 | 缺省向所有registry注册 | 配置关联 | 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 | 2.0.0以上版本 |
| <dubbo:service> | provider |  | string | 可选 | 缺使用第一个provider配置 | 配置关联 | 指定provider，值为<dubbo:provider>的id属性 | 2.0.0以上版本 |
| <dubbo:service> | deprecated | deprecated | boolean | 可选 | false | 服务治理 | 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 | 2.0.5以上版本 |
| <dubbo:service> | dynamic | dynamic | boolean | 可选 | true | 服务治理 | 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。 | 2.0.5以上版本 |
| <dubbo:service> | accesslog | accesslog | string/boolean | 可选 | false | 服务治理 | 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 | 2.0.5以上版本 |
| <dubbo:service> | owner | owner | string | 可选 |  | 服务治理 | 服务负责人，用于服务治理，请填写负责人公司邮箱前缀 | 2.0.5以上版本 |
| <dubbo:service> | document | document | string | 可选 |  | 服务治理 | 服务文档URL | 2.0.5以上版本 |
| <dubbo:service> | weight | weight | int | 可选 |  | 性能调优 | 服务权重 | 2.0.5以上版本 |
| <dubbo:service> | executes | executes | int | 可选 | 0 | 性能调优 | 服务提供者每服务每方法最大可并行执行请求数 | 2.0.5以上版本 |
| <dubbo:service> | actives | actives | int | 可选 | 0 | 性能调优 | 每服务消费者每服务每方法最大并发调用数 | 2.0.5以上版本 |
| <dubbo:service> | proxy | proxy | string | 可选 | javassist | 性能调优 | 生成动态代理方式，可选：jdk/javassist | 2.0.5以上版本 |
| <dubbo:service> | cluster | cluster | string | 可选 | failover | 性能调优 | 集群方式，可选：failover/failfast/failsafe/failback/forking | 2.0.5以上版本 |
| <dubbo:service> | filter | service.filter | string | 可选 | default | 性能调优 | 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 | 2.0.5以上版本 |
| <dubbo:service> | listener | exporter.listener | string | 可选 | default | 性能调优 | 服务提供方导出服务监听器名称，多个名称用逗号分隔 |  |
| <dubbo:service> | protocol |  | string | 可选 |  | 配置关联 | 使用指定的协议暴露服务，在多协议时使用，值为<dubbo:protocol>的id属性，多个协议ID用逗号分隔 | 2.0.5以上版本 |
| <dubbo:service> | layer | layer | string | 可选 |  | 服务治理 | 服务提供者所在的分层。如：biz、dao、intl:web、china:acton。 | 2.0.7以上版本 |
| <dubbo:service> | register | register | boolean | 可选 | true | 服务治理 | 该协议的服务是否注册到注册中心 | 2.0.8以上版本 |

#### <dubbo:reference/>

服务消费者引用服务配置：  
配置类：com.alibaba.dubbo.config.ReferenceConfig

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **标签** | **属性** | **对应URL参数** | **类型** | **是否必填** | **缺省值** | **作用** | **描述** | **兼容性** |
| <dubbo:reference> | id |  | string | 必填 |  | 配置关联 | 服务引用BeanId | 1.0.0以上版本 |
| <dubbo:reference> | interface |  | class | 必填 |  | 服务发现 | 服务接口名 | 1.0.0以上版本 |
| <dubbo:reference> | version | version | string | 可选 |  | 服务发现 | 服务版本，与服务提供者的版本一致 | 1.0.0以上版本 |
| <dubbo:reference> | group | group | string | 可选 |  | 服务发现 | 服务分组，当一个接口有多个实现，可以用分组区分，必需和服务提供方一致 | 1.0.7以上版本 |
| <dubbo:reference> | timeout | timeout | long | 可选 | 缺省使用<dubbo:consumer>的timeout | 性能调优 | 服务方法调用超时时间(毫秒) | 1.0.5以上版本 |
| <dubbo:reference> | retries | retries | int | 可选 | 缺省使用<dubbo:consumer>的retries | 性能调优 | 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 | 2.0.0以上版本 |
| <dubbo:reference> | connections | connections | int | 可选 | 缺省使用<dubbo:consumer>的connections | 性能调优 | 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 | 2.0.0以上版本 |
| <dubbo:reference> | loadbalance | loadbalance | string | 可选 | 缺省使用<dubbo:consumer>的loadbalance | 性能调优 | 负载均衡策略，可选值：random,roundrobin,leastactive，分别表示：随机，轮循，最少活跃调用 | 2.0.0以上版本 |
| <dubbo:reference> | async | async | boolean | 可选 | 缺省使用<dubbo:consumer>的async | 性能调优 | 是否异步执行，不可靠异步，只是忽略返回值，不阻塞执行线程 | 2.0.0以上版本 |
| <dubbo:reference> | generic | generic | boolean | 可选 | 缺省使用<dubbo:consumer>的generic | 服务治理 | 是否缺省泛化接口，如果为泛化接口，将返回GenericService | 2.0.0以上版本 |
| <dubbo:reference> | check | check | boolean | 可选 | 缺省使用<dubbo:consumer>的check | 服务治理 | 启动时检查提供者是否存在，true报错，false忽略 | 2.0.0以上版本 |
| <dubbo:reference> | url | <url> | string | 可选 |  | 服务治理 | 点对点直连服务提供者地址，将绕过注册中心 | 1.0.6以上版本 |
| <dubbo:reference> | stub | stub | class/boolean | 可选 |  | 服务治理 | 服务接口客户端本地代理类名，用于在客户端执行本地逻辑，如本地缓存等，该本地代理类的构造函数必须允许传入远程代理对象，构造函数如：public XxxServiceLocal(XxxService xxxService) | 2.0.0以上版本 |
| <dubbo:reference> | mock | mock | class/boolean | 可选 |  | 服务治理 | 服务接口调用失败Mock实现类名，该Mock类必须有一个无参构造函数，与Local的区别在于，Local总是被执行，而Mock只在出现非业务异常(比如超时，网络异常等)时执行，Local在远程调用之前执行，Mock在远程调用后执行。 | Dubbo1.0.13及其以上版本支持 |
| <dubbo:reference> | cache | cache | string/boolean | 可选 |  | 服务治理 | 以调用参数为key，缓存返回结果，可选：lru, threadlocal, jcache等 | Dubbo2.1.0及其以上版本支持 |
| <dubbo:reference> | validation | validation | boolean | 可选 |  | 服务治理 | 是否启用JSR303标准注解验证，如果启用，将对方法参数上的注解进行校验 | Dubbo2.1.0及其以上版本支持 |
| <dubbo:reference> | proxy | proxy | boolean | 可选 | javassist | 性能调优 | 选择动态代理实现策略，可选：javassist, jdk | 2.0.2以上版本 |
| <dubbo:reference> | client | client | string | 可选 |  | 性能调优 | 客户端传输类型设置，如Dubbo协议的netty或mina。 | Dubbo2.0.0以上版本支持 |
| <dubbo:reference> | registry |  | string | 可选 | 缺省将从所有注册中心获服务列表后合并结果 | 配置关联 | 从指定注册中心注册获取服务列表，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 | 2.0.0以上版本 |
| <dubbo:reference> | owner | owner | string | 可选 |  | 服务治理 | 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 | 2.0.5以上版本 |
| <dubbo:reference> | actives | actives | int | 可选 | 0 | 性能调优 | 每服务消费者每服务每方法最大并发调用数 | 2.0.5以上版本 |
| <dubbo:reference> | cluster | cluster | string | 可选 | failover | 性能调优 | 集群方式，可选：failover/failfast/failsafe/failback/forking | 2.0.5以上版本 |
| <dubbo:reference> | filter | reference.filter | string | 可选 | default | 性能调优 | 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 | 2.0.5以上版本 |
| <dubbo:reference> | listener | invoker.listener | string | 可选 | default | 性能调优 | 服务消费方引用服务监听器名称，多个名称用逗号分隔 | 2.0.5以上版本 |
| <dubbo:reference> | layer | layer | string | 可选 |  | 服务治理 | 服务调用者所在的分层。如：biz、dao、intl:web、china:acton。 | 2.0.7以上版本 |
| <dubbo:reference> | init | init | boolean | 可选 | false | 性能调优 | 是否在afterPropertiesSet()时饥饿初始化引用，否则等到有人注入或引用该实例时再初始化。 | 2.0.10以上版本 |
| <dubbo:reference> | protocol | protocol | string | 可选 |  | 服力治理 | 只调用指定协议的服务提供方，其它协议忽略。 | 2.2.0以上版本 |

#### <dubbo:protocol/>

服务提供者协议配置：

配置类：com.alibaba.dubbo.config.ProtocolConfig  
说明：如果需要支持多协议，可以声明多个<dubbo:protocol>标签，并在<dubbo:service>中通过protocol属性指定使用的协议。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **标签** | **属性** | **对应URL参数** | **类型** | **是否必填** | **缺省值** | **作用** | **描述** | **兼容性** |
| <dubbo:protocol> | id |  | string | 可选 | dubbo | 配置关联 | 协议BeanId，可以在<dubbo:service protocol="">中引用此ID，如果ID不填，缺省和name属性值一样，重复则在name后加序号。 | 2.0.5以上版本 |
| <dubbo:protocol> | name | <protocol> | string | 必填 | dubbo | 性能调优 | 协议名称 | 2.0.5以上版本 |
| <dubbo:protocol> | port | <port> | int | 可选 | dubbo协议缺省端口为20880，rmi协议缺省端口为1099，http和hessian协议缺省端口为80  如果配置为-1 或者 没有配置port，则会分配一个没有被占用的端口。Dubbo 2.4.0+，分配的端口在协议缺省端口的基础上增长，确保端口段可控。 | 服务发现 | 服务端口 | 2.0.5以上版本 |
| <dubbo:protocol> | host | <host> | string | 可选 | 自动查找本机IP | 服务发现 | -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 | 2.0.5以上版本 |
| <dubbo:protocol> | threadpool | threadpool | string | 可选 | fixed | 性能调优 | 线程池类型，可选：fixed/cached | 2.0.5以上版本 |
| <dubbo:protocol> | threads | threads | int | 可选 | 100 | 性能调优 | 服务线程池大小(固定大小) | 2.0.5以上版本 |
| <dubbo:protocol> | iothreads | threads | int | 可选 | cpu个数+1 | 性能调优 | io线程池大小(固定大小) | 2.0.5以上版本 |
| <dubbo:protocol> | accepts | accepts | int | 可选 | 0 | 性能调优 | 服务提供方最大可接受连接数 | 2.0.5以上版本 |
| <dubbo:protocol> | payload | payload | int | 可选 | 88388608(=8M) | 性能调优 | 请求及响应数据包大小限制，单位：字节 | 2.0.5以上版本 |
| <dubbo:protocol> | codec | codec | string | 可选 | dubbo | 性能调优 | 协议编码方式 | 2.0.5以上版本 |
| <dubbo:protocol> | serialization | serialization | string | 可选 | dubbo协议缺省为hessian2，rmi协议缺省为java，http协议缺省为json | 性能调优 | 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等 | 2.0.5以上版本 |
| <dubbo:protocol> | accesslog | accesslog | string/boolean | 可选 |  | 服务治理 | 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 | 2.0.5以上版本 |
| <dubbo:protocol> | path | <path> | string | 可选 |  | 服务发现 | 提供者上下文路径，为服务path的前缀 | 2.0.5以上版本 |
| <dubbo:protocol> | transporter | transporter | string | 可选 | dubbo协议缺省为netty | 性能调优 | 协议的服务端和客户端实现类型，比如：dubbo协议的mina,netty等，可以分拆为server和client配置 | 2.0.5以上版本 |
| <dubbo:protocol> | server | server | string | 可选 | dubbo协议缺省为netty，http协议缺省为servlet | 性能调优 | 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 | 2.0.5以上版本 |
| <dubbo:protocol> | client | client | string | 可选 | dubbo协议缺省为netty | 性能调优 | 协议的客户端实现类型，比如：dubbo协议的mina,netty等 | 2.0.5以上版本 |
| <dubbo:protocol> | dispatcher | dispatcher | string | 可选 | dubbo协议缺省为all | 性能调优 | 协议的消息派发方式，用于指定线程模型，比如：dubbo协议的all, direct, message, execution, connection等 | 2.1.0以上版本 |
| <dubbo:protocol> | queues | queues | int | 可选 | 0 | 性能调优 | 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，当线程程池时应立即失败，重试其它服务提供机器，而不是排队，除非有特殊需求。 | 2.0.5以上版本 |
| <dubbo:protocol> | charset | charset | string | 可选 | UTF-8 | 性能调优 | 序列化编码 | 2.0.5以上版本 |
| <dubbo:protocol> | buffer | buffer | int | 可选 | 8192 | 性能调优 | 网络读写缓冲区大小 | 2.0.5以上版本 |
| <dubbo:protocol> | heartbeat | heartbeat | int | 可选 | 0 | 性能调优 | 心跳间隔，对于长连接，当物理层断开时，比如拔网线，TCP的FIN消息来不及发送，对方收不到断开事件，此时需要心跳来帮助检查连接是否已断开 | 2.0.10以上版本 |
| <dubbo:protocol> | telnet | telnet | string | 可选 |  | 服务治理 | 所支持的telnet命令，多个命令用逗号分隔 | 2.0.5以上版本 |
| <dubbo:protocol> | register | register | boolean | 可选 | true | 服务治理 | 该协议的服务是否注册到注册中心 | 2.0.8以上版本 |
| <dubbo:protocol> | contextpath | contextpath | String | 可选 | 缺省为空串 | 服务治理 |  | 2.0.6以上版本 |

#### <dubbo:registry/>

注册中心配置：

配置类：com.alibaba.dubbo.config.RegistryConfig  
说明：如果有多个不同的注册中心，可以声明多个<dubbo:registry>标签，并在<dubbo:service>或<dubbo:reference>的registry属性指定使用的注册中心。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **标签** | **属性** | **对应URL参数** | **类型** | **是否必填** | **缺省值** | **作用** | **描述** | **兼容性** |
| <dubbo:registry> | id |  | string | 可选 |  | 配置关联 | 注册中心引用BeanId，可以在<dubbo:service registry="">或<dubbo:reference registry="">中引用此ID | 1.0.16以上版本 |
| <dubbo:registry> | address | <host:port> | string | 必填 |  | 服务发现 | 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 | 1.0.16以上版本 |
| <dubbo:registry> | protocol | <protocol> | string | 可选 | dubbo | 服务发现 | 注同中心地址协议，支持dubbo, http, local三种协议，分别表示，dubbo地址，http地址，本地注册中心 | 2.0.0以上版本 |
| <dubbo:registry> | port | <port> | int | 可选 | 9090 | 服务发现 | 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 | 2.0.0以上版本 |
| <dubbo:registry> | username | <username> | string | 可选 |  | 服务治理 | 登录注册中心用户名，如果注册中心不需要验证可不填 | 2.0.0以上版本 |
| <dubbo:registry> | password | <password> | string | 可选 |  | 服务治理 | 登录注册中心密码，如果注册中心不需要验证可不填 | 2.0.0以上版本 |
| <dubbo:registry> | transport | registry.transporter | string | 可选 | netty | 性能调优 | 网络传输方式，可选mina,netty | 2.0.0以上版本 |
| <dubbo:registry> | timeout | registry.timeout | int | 可选 | 5000 | 性能调优 | 注册中心请求超时时间(毫秒) | 2.0.0以上版本 |
| <dubbo:registry> | session | registry.session | int | 可选 | 60000 | 性能调优 | 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。 | 2.1.0以上版本 |
| <dubbo:registry> | file | registry.file | string | 可选 |  | 服务治理 | 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 | 2.0.0以上版本 |
| <dubbo:registry> | wait | registry.wait | int | 可选 | 0 | 性能调优 | 停止时等待通知完成时间(毫秒) | 2.0.0以上版本 |
| <dubbo:registry> | check | check | boolean | 可选 | true | 服务治理 | 注册中心不存在时，是否报错 | 2.0.0以上版本 |
| <dubbo:registry> | register | register | boolean | 可选 | true | 服务治理 | 是否向此注册中心注册服务，如果设为false，将只订阅，不注册 | 2.0.5以上版本 |
| <dubbo:registry> | subscribe | subscribe | boolean | 可选 | true | 服务治理 | 是否向此注册中心订阅服务，如果设为false，将只注册，不订阅 | 2.0.5以上版本 |
| <dubbo:registry> | dynamic | dynamic | boolean | 可选 | true | 服务治理 | 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。 | 2.0.5以上版本 |

#### <dubbo:monitor/>

监控中心配置：

配置类：com.alibaba.dubbo.config.MonitorConfig

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **标签** | **属性** | **对应URL参数** | **类型** | **是否必填** | **缺省值** | **作用** | **描述** | **兼容性** |
| <dubbo:monitor> | protocol | protocol | string | 可选 | dubbo | 服务治理 | 监控中心协议，如果为protocol="registry"，表示从注册中心发现监控中心地址，否则直连监控中心。 | 2.0.9以上版本 |
| <dubbo:monitor> | address | <url> | string | 可选 | N/A | 服务治理 | 直连监控中心服务器地址，address="10.20.130.230:12080" | 1.0.16以上版本 |

#### <dubbo:application/>

应用信息配置：

配置类：com.alibaba.dubbo.config.ApplicationConfig

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **标签** | **属性** | **对应URL参数** | **类型** | **是否必填** | **缺省值** | **作用** | **描述** | **兼容性** |
| <dubbo:application> | name | application | string | 必填 |  | 服务治理 | 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 | 1.0.16以上版本 |
| <dubbo:application> | version | application.version | string | 可选 |  | 服务治理 | 当前应用的版本 | 2.2.0以上版本 |
| <dubbo:application> | owner | owner | string | 可选 |  | 服务治理 | 应用负责人，用于服务治理，请填写负责人公司邮箱前缀 | 2.0.5以上版本 |
| <dubbo:application> | organization | organization | string | 可选 |  | 服务治理 | 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，比如china,intl,itu,crm,asc,dw,aliexpress等 | 2.0.0以上版本 |
| <dubbo:application> | architecture | architecture | string | 可选 |  | 服务治理 | 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。 | 2.0.7以上版本 |
| <dubbo:application> | environment | environment | string | 可选 |  | 服务治理 | 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 | 2.0.0以上版本 |
| <dubbo:application> | compiler | compiler | string | 可选 | javassist | 性能优化 | Java字节码编译器，用于动态类的生成，可选：jdk或javassist | 2.1.0以上版本 |
| <dubbo:application> | logger | logger | string | 可选 | slf4j | 性能优化 | 日志输出方式，可选：slf4j,jcl,log4j,jdk | 2.2.0以上版本 |

#### <dubbo:module/>

模块信息配置：

配置类：com.alibaba.dubbo.config.ModuleConfig

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **标签** | **属性** | **对应URL参数** | **类型** | **是否必填** | **缺省值** | **作用** | **描述** | **兼容性** |
| <dubbo:module> | name | module | string | 必填 |  | 服务治理 | 当前模块名称，用于注册中心计算模块间依赖关系 | 2.2.0以上版本 |
| <dubbo:module> | version | module.version | string | 可选 |  | 服务治理 | 当前模块的版本 | 2.2.0以上版本 |
| <dubbo:module> | owner | owner | string | 可选 |  | 服务治理 | 模块负责人，用于服务治理，请填写负责人公司邮箱前缀 | 2.2.0以上版本 |
| <dubbo:module> | organization | organization | string | 可选 |  | 服务治理 | 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，比如china,intl,itu,crm,asc,dw,aliexpress等 | 2.2.0以上版本 |

#### <dubbo:provider/>

服务提供者缺省值配置：

配置类：com.alibaba.dubbo.config.ProviderConfig  
说明：该标签为<dubbo:service>和<dubbo:protocol>标签的缺省值设置。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **标签** | **属性** | **对应URL参数** | **类型** | **是否必填** | **缺省值** | **作用** | **描述** | **兼容性** |
| <dubbo:provider> | id |  | string | 可选 | dubbo | 配置关联 | 协议BeanId，可以在<dubbo:service proivder="">中引用此ID | 1.0.16以上版本 |
| <dubbo:provider> | protocol | <protocol> | string | 可选 | dubbo | 性能调优 | 协议名称 | 1.0.16以上版本 |
| <dubbo:provider> | host | <host> | string | 可选 | 自动查找本机IP | 服务发现 | 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 | 1.0.16以上版本 |
| <dubbo:provider> | threads | threads | int | 可选 | 100 | 性能调优 | 服务线程池大小(固定大小) | 1.0.16以上版本 |
| <dubbo:provider> | payload | payload | int | 可选 | 88388608(=8M) | 性能调优 | 请求及响应数据包大小限制，单位：字节 | 2.0.0以上版本 |
| <dubbo:provider> | path | <path> | string | 可选 |  | 服务发现 | 提供者上下文路径，为服务path的前缀 | 2.0.0以上版本 |
| <dubbo:provider> | server | server | string | 可选 | dubbo协议缺省为netty，http协议缺省为servlet | 性能调优 | 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 | 2.0.0以上版本 |
| <dubbo:provider> | client | client | string | 可选 | dubbo协议缺省为netty | 性能调优 | 协议的客户端实现类型，比如：dubbo协议的mina,netty等 | 2.0.0以上版本 |
| <dubbo:provider> | codec | codec | string | 可选 | dubbo | 性能调优 | 协议编码方式 | 2.0.0以上版本 |
| <dubbo:provider> | serialization | serialization | string | 可选 | dubbo协议缺省为hessian2，rmi协议缺省为java，http协议缺省为json | 性能调优 | 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 | 2.0.5以上版本 |
| <dubbo:provider> | default |  | boolean | 可选 | false | 配置关联 | 是否为缺省协议，用于多协议 | 1.0.16以上版本 |
| <dubbo:provider> | filter | service.filter | string | 可选 |  | 性能调优 | 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 | 2.0.5以上版本 |
| <dubbo:provider> | listener | exporter.listener | string | 可选 |  | 性能调优 | 服务提供方导出服务监听器名称，多个名称用逗号分隔 | 2.0.5以上版本 |
| <dubbo:provider> | threadpool | threadpool | string | 可选 | fixed | 性能调优 | 线程池类型，可选：fixed/cached | 2.0.5以上版本 |
| <dubbo:provider> | accepts | accepts | int | 可选 | 0 | 性能调优 | 服务提供者最大可接受连接数 | 2.0.5以上版本 |
| <dubbo:provider> | version | version | string | 可选 | 0.0.0 | 服务发现 | 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 | 2.0.5以上版本 |
| <dubbo:provider> | group | group | string | 可选 |  | 服务发现 | 服务分组，当一个接口有多个实现，可以用分组区分 | 2.0.5以上版本 |
| <dubbo:provider> | delay | delay | int | 可选 | 0 | 性能调优 | 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 | 2.0.5以上版本 |
| <dubbo:provider> | timeout | default.timeout | int | 可选 | 1000 | 性能调优 | 远程服务调用超时时间(毫秒) | 2.0.5以上版本 |
| <dubbo:provider> | retries | default.retries | int | 可选 | 2 | 性能调优 | 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 | 2.0.5以上版本 |
| <dubbo:provider> | connections | default.connections | int | 可选 | 0 | 性能调优 | 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 | 2.0.5以上版本 |
| <dubbo:provider> | loadbalance | default.loadbalance | string | 可选 | random | 性能调优 | 负载均衡策略，可选值：random,roundrobin,leastactive，分别表示：随机，轮循，最少活跃调用 | 2.0.5以上版本 |
| <dubbo:provider> | async | default.async | boolean | 可选 | false | 性能调优 | 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 | 2.0.5以上版本 |
| <dubbo:provider> | stub | stub | boolean | 可选 | false | 服务治理 | 设为true，表示使用缺省代理类名，即：接口名 + Local后缀。 | 2.0.5以上版本 |
| <dubbo:provider> | mock | mock | boolean | 可选 | false | 服务治理 | 设为true，表示使用缺省Mock类名，即：接口名 + Mock后缀。 | 2.0.5以上版本 |
| <dubbo:provider> | token | token | boolean | 可选 | false | 服务治理 | 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 | 2.0.5以上版本 |
| <dubbo:provider> | registry | registry | string | 可选 | 缺省向所有registry注册 | 配置关联 | 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 | 2.0.5以上版本 |
| <dubbo:provider> | dynamic | dynamic | boolean | 可选 | true | 服务治理 | 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。 | 2.0.5以上版本 |
| <dubbo:provider> | accesslog | accesslog | string/boolean | 可选 | false | 服务治理 | 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 | 2.0.5以上版本 |
| <dubbo:provider> | owner | owner | string | 可选 |  | 服务治理 | 服务负责人，用于服务治理，请填写负责人公司邮箱前缀 | 2.0.5以上版本 |
| <dubbo:provider> | document | document | string | 可选 |  | 服务治理 | 服务文档URL | 2.0.5以上版本 |
| <dubbo:provider> | weight | weight | int | 可选 |  | 性能调优 | 服务权重 | 2.0.5以上版本 |
| <dubbo:provider> | executes | executes | int | 可选 | 0 | 性能调优 | 服务提供者每服务每方法最大可并行执行请求数 | 2.0.5以上版本 |
| <dubbo:provider> | actives | default.actives | int | 可选 | 0 | 性能调优 | 每服务消费者每服务每方法最大并发调用数 | 2.0.5以上版本 |
| <dubbo:provider> | proxy | proxy | string | 可选 | javassist | 性能调优 | 生成动态代理方式，可选：jdk/javassist | 2.0.5以上版本 |
| <dubbo:provider> | cluster | default.cluster | string | 可选 | failover | 性能调优 | 集群方式，可选：failover/failfast/failsafe/failback/forking | 2.0.5以上版本 |
| <dubbo:provider> | deprecated | deprecated | boolean | 可选 | false | 服务治理 | 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 | 2.0.5以上版本 |
| <dubbo:provider> | queues | queues | int | 可选 | 0 | 性能调优 | 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，当线程程池时应立即失败，重试其它服务提供机器，而不是排队，除非有特殊需求。 | 2.0.5以上版本 |
| <dubbo:provider> | charset | charset | string | 可选 | UTF-8 | 性能调优 | 序列化编码 | 2.0.5以上版本 |
| <dubbo:provider> | buffer | buffer | int | 可选 | 8192 | 性能调优 | 网络读写缓冲区大小 | 2.0.5以上版本 |
| <dubbo:provider> | iothreads | iothreads | int | 可选 | CPU + 1 | 性能调优 | 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。 | 2.0.5以上版本 |
| <dubbo:provider> | telnet | telnet | string | 可选 |  | 服务治理 | 所支持的telnet命令，多个命令用逗号分隔 | 2.0.5以上版本 |
| <dubbo:service> | contextpath | contextpath | String | 可选 | 缺省为空串 | 服务治理 |  | 2.0.6以上版本 |
| <dubbo:provider> | layer | layer | string | 可选 |  | 服务治理 | 服务提供者所在的分层。如：biz、dao、intl:web、china:acton。 | 2.0.7以上版本 |

#### <dubbo:consumer/>

服务消费者缺省值配置：

配置类：com.alibaba.dubbo.config.ConsumerConfig  
说明：该标签为<dubbo:reference>标签的缺省值设置。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **标签** | **属性** | **对应URL参数** | **类型** | **是否必填** | **缺省值** | **作用** | **描述** | **兼容性** |
| <dubbo:consumer> | timeout | default.timeout | int | 可选 | 1000 | 性能调优 | 远程服务调用超时时间(毫秒) | 1.0.16以上版本 |
| <dubbo:consumer> | retries | default.retries | int | 可选 | 2 | 性能调优 | 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 | 1.0.16以上版本 |
| <dubbo:consumer> | loadbalance | default.loadbalance | string | 可选 | random | 性能调优 | 负载均衡策略，可选值：random,roundrobin,leastactive，分别表示：随机，轮循，最少活跃调用 | 1.0.16以上版本 |
| <dubbo:consumer> | async | default.async | boolean | 可选 | false | 性能调优 | 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 | 2.0.0以上版本 |
| <dubbo:consumer> | connections | default.connections | int | 可选 | 100 | 性能调优 | 每个服务对每个提供者的最大连接数，rmi、http、hessian等短连接协议支持此配置，dubbo协议长连接不支持此配置 | 1.0.16以上版本 |
| <dubbo:consumer> | generic | generic | boolean | 可选 | false | 服务治理 | 是否缺省泛化接口，如果为泛化接口，将返回GenericService | 2.0.0以上版本 |
| <dubbo:consumer> | check | check | boolean | 可选 | true | 服务治理 | 启动时检查提供者是否存在，true报错，false忽略 | 1.0.16以上版本 |
| <dubbo:consumer> | proxy | proxy | string | 可选 | javassist | 性能调优 | 生成动态代理方式，可选：jdk/javassist | 2.0.5以上版本 |
| <dubbo:consumer> | owner | owner | string | 可选 |  | 服务治理 | 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 | 2.0.5以上版本 |
| <dubbo:consumer> | actives | default.actives | int | 可选 | 0 | 性能调优 | 每服务消费者每服务每方法最大并发调用数 | 2.0.5以上版本 |
| <dubbo:consumer> | cluster | default.cluster | string | 可选 | failover | 性能调优 | 集群方式，可选：failover/failfast/failsafe/failback/forking | 2.0.5以上版本 |
| <dubbo:consumer> | filter | reference.filter | string | 可选 |  | 性能调优 | 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 | 2.0.5以上版本 |
| <dubbo:consumer> | listener | invoker.listener | string | 可选 |  | 性能调优 | 服务消费方引用服务监听器名称，多个名称用逗号分隔 | 2.0.5以上版本 |
| <dubbo:consumer> | registry |  | string | 可选 | 缺省向所有registry注册 | 配置关联 | 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 | 2.0.5以上版本 |
| <dubbo:consumer> | layer | layer | string | 可选 |  | 服务治理 | 服务调用者所在的分层。如：biz、dao、intl:web、china:acton。 | 2.0.7以上版本 |
| <dubbo:consumer> | init | init | boolean | 可选 | false | 性能调优 | 是否在afterPropertiesSet()时饥饿初始化引用，否则等到有人注入或引用该实例时再初始化。 | 2.0.10以上版本 |
| <dubbo:consumer> | cache | cache | string/boolean | 可选 |  | 服务治理 | 以调用参数为key，缓存返回结果，可选：lru, threadlocal, jcache等 | Dubbo2.1.0及其以上版本支持 |
| <dubbo:consumer> | validation | validation | boolean | 可选 |  | 服务治理 | 是否启用JSR303标准注解验证，如果启用，将对方法参数上的注解进行校验 | Dubbo2.1.0及其以上版本支持 |

#### <dubbo:method/>

方法级配置：

配置类：com.alibaba.dubbo.config.MethodConfig  
说明：该标签为<dubbo:service>或<dubbo:reference>的子标签，用于控制到方法级，

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **标签** | **属性** | **对应URL参数** | **类型** | **是否必填** | **缺省值** | **作用** | **描述** | **兼容性** |
| <dubbo:method> | name |  | string | 必填 |  | 标识 | 方法名 | 1.0.8以上版本 |
| <dubbo:method> | timeout | <metodName>.timeout | int | 可选 | 缺省为的timeout | 性能调优 | 方法调用超时时间(毫秒) | 1.0.8以上版本 |
| <dubbo:method> | retries | <metodName>.retries | int | 可选 | 缺省为<dubbo:reference>的retries | 性能调优 | 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 | 2.0.0以上版本 |
| <dubbo:method> | loadbalance | <metodName>.loadbalance | string | 可选 | 缺省为的loadbalance | 性能调优 | 负载均衡策略，可选值：random,roundrobin,leastactive，分别表示：随机，轮循，最少活跃调用 | 2.0.0以上版本 |
| <dubbo:method> | async | <metodName>.async | boolean | 可选 | 缺省为<dubbo:reference>的async | 性能调优 | 是否异步执行，不可靠异步，只是忽略返回值，不阻塞执行线程 | 1.0.9以上版本 |
| <dubbo:method> | sent | <methodName>.sent | boolean | 可选 | true | 性能调优 | 异步调用时，标记sent=true时，表示网络已发出数据 | 2.0.6以上版本 |
| <dubbo:method> | actives | <metodName>.actives | int | 可选 | 0 | 性能调优 | 每服务消费者最大并发调用限制 | 2.0.5以上版本 |
| <dubbo:method> | executes | <metodName>.executes | int | 可选 | 0 | 性能调优 | 每服务每方法最大使用线程数限制- -，此属性只在<dubbo:method>作为<dubbo:service>子标签时有效 | 2.0.5以上版本 |
| <dubbo:method> | deprecated | <methodName>.deprecated | boolean | 可选 | false | 服务治理 | 服务方法是否过时，此属性只在<dubbo:method>作为<dubbo:service>子标签时有效 | 2.0.5以上版本 |
| <dubbo:method> | sticky | <methodName>.sticky | boolean | 可选 | false | 服务治理 | 设置true 该接口上的所有方法使用同一个provider.如果需要更复杂的规则，请使用用路由 | 2.0.6以上版本 |
| <dubbo:method> | return | <methodName>.return | boolean | 可选 | true | 性能调优 | 方法调用是否需要返回值,async设置为true时才生效，如果设置为true，则返回future，或回调onreturn等方法，如果设置为false，则请求发送成功后直接返回Null | 2.0.6以上版本 |
| <dubbo:method> | oninvoke | attribute属性，不在URL中体现 | String | 可选 |  | 性能调优 | 方法执行前拦截 | 2.0.6以上版本 |
| <dubbo:method> | onreturn | attribute属性，不在URL中体现 | String | 可选 |  | 性能调优 | 方法执行返回后拦截 | 2.0.6以上版本 |
| <dubbo:method> | onthrow | attribute属性，不在URL中体现 | String | 可选 |  | 性能调优 | 方法执行有异常拦截 | 2.0.6以上版本 |
| <dubbo:method> | cache | <methodName>.cache | string/boolean | 可选 |  | 服务治理 | 以调用参数为key，缓存返回结果，可选：lru, threadlocal, jcache等 | Dubbo2.1.0及其以上版本支持 |
| <dubbo:method> | validation | <methodName>.validation | boolean | 可选 |  | 服务治理 | 是否启用JSR303标准注解验证，如果启用，将对方法参数上的注解进行校验 | Dubbo2.1.0及其以上版本支持 |

比如：

<dubbo:reference interface="com.xxx.XxxService">

    <dubbo:method name="findXxx" timeout="3000" retries="2" />

</dubbo:reference>

#### <dubbo:argument/>

方法参数配置：

配置类：com.alibaba.dubbo.config.ArgumentConfig  
说明：该标签为<dubbo:method>的子标签，用于方法参数的特征描述，比如：

<dubbo:method name="findXxx" timeout="3000" retries="2">

    <dubbo:argument index="0" callback="true" />

<dubbo:method>

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **标签** | **属性** | **对应URL参数** | **类型** | **是否必填** | **缺省值** | **作用** | **描述** | **兼容性** |
| <dubbo:argument> | index |  | int | 必填 |  | 标识 | 方法名 | 2.0.6以上版本 |
| <dubbo:argument> | type |  | String | 与index二选一 |  | 标识 | 通过参数类型查找参数的index | 2.0.6以上版本 |
| <dubbo:argument> | callback | <metodName><index>.retries | boolean | 可选 |  | 服务治理 | 参数是否为callback接口，如果为callback，服务提供方将生成反向代理，可以从服务提供方反向调用消费方，通常用于事件推送. | 2.0.6以上版本 |

#### <dubbo:parameter/>

选项参数配置：

配置类：java.util.Map  
说明：该标签为<dubbo:protocol>或<dubbo:service>或<dubbo:provider>或<dubbo:reference>或<dubbo:consumer>的子标签，用于配置自定义参数，该配置项将作为扩展点设置自定义参数使用。

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **标签** | **属性** | **对应URL参数** | **类型** | **是否必填** | **缺省值** | **作用** | **描述** | **兼容性** |
| <dubbo:parameter> | key | key | string | 必填 |  | 服务治理 | 路由参数键 | 2.0.0以上版本 |
| <dubbo:parameter> | value | value | string | 必填 |  | 服务治理 | 路由参数值 | 2.0.0以上版本 |

### xml配置使用样例

xml的配置使用样例，以及与Springmvc的集成demo 。可参见源码程序包busi-xml-server-client 和 busi-mvc项目

## 注解方式

注解方式的底层与XML一致，只是表现形式上的不同。目标都是将Dubbo基础信息配入，主要涉及以下五个必不可少的信息：**ApplicationConfig、ProtocolConfig 、 RegistryConfig、service、reference**

### EnableDubbo开启服务

**@EnableDubbo：**开启注解Dubbo功能 ，其中可以加入scanBasePackages属性配置包扫描的路径，用于扫描并注册bean。其中 封装了组件 @DubboComponentScan，来扫描Dubbo @Service 注解暴露 Dubbo 服务，以及扫描Dubbo  @Reference字段或者方法注入 Dubbo 服务代理。

其它Dubbo三种公共信息的配置，有两种方式，根据自己喜好选用

### Configuration方式配置公共信息

@Configuration方式：分别将**ApplicationConfig、ProtocolConfig 、 RegistryConfig**创建到IOC容器中即可，如下：

@Configuration  
@EnableDubbo(scanBasePackages = "com.enjoy.service")  
class ProviderConfiguration {  
 @Bean  
 public ApplicationConfig applicationConfig() {  
 ApplicationConfig applicationConfig = new ApplicationConfig();  
 applicationConfig.setName("busi-provider");  
 return applicationConfig;  
 }  
 @Bean  
 public RegistryConfig registryConfig() {  
 RegistryConfig registryConfig = new RegistryConfig();  
 registryConfig.setProtocol("zookeeper");  
 registryConfig.setAddress("192.168.0.128");  
 registryConfig.setPort(2181);  
 return registryConfig;  
 }  
 @Bean  
 public ProtocolConfig protocolConfig() {  
 ProtocolConfig protocolConfig = new ProtocolConfig();  
 protocolConfig.setName("dubbo");  
 protocolConfig.setPort(20880);  
 return protocolConfig;  
 }  
}

### Property方式自动装配公共信息

Property方式：使用Springboot属性文件方式，由Dubbo自动将文件信息配置入容器，示例如下：

@Configuration  
@EnableDubbo(scanBasePackages = "com.enjoy.service")  
@PropertySource("classpath:/dubbo-provider.properties")  
static class ProviderConfiguration {  
}

**dubbo-provider.properties文件内容**

dubbo.application.name=busi-provider  
dubbo.registry.address=zookeeper://192.168.0.128:2181  
dubbo.protocol.name=dubbo  
dubbo.protocol.port=20880

完整的项目示例，参见busi-xml-server-client源码包

# Dubbo高级特性

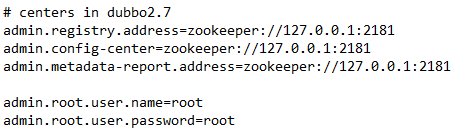
## Dubbo控制台部署

从2.6版本之后，dubbo控制台已单独版本管理（目前只到0.1版本），使用了前后端分离的模式。前端使用Vue和Vuetify分别作为Javascript框架和UI框架，后端采用Spring Boot框架。考虑到后端人员操作的简易性，本文只介绍Maven部署方案（即将前端vue产出的静态内容集成到springboot包内）

* 下载：

git clone https://github.com/apache/dubbo-admin.git

在dubbo-admin-server子项目的属性文件中，设置zk地址及登陆帐户/密码



* 编译：

cd dubbo-admin

mvn clean package ##编译时间稍长，请耐心等待（前提是配置好你的maven环境）

* 启动：

cd dubbo-admin-distribution/target

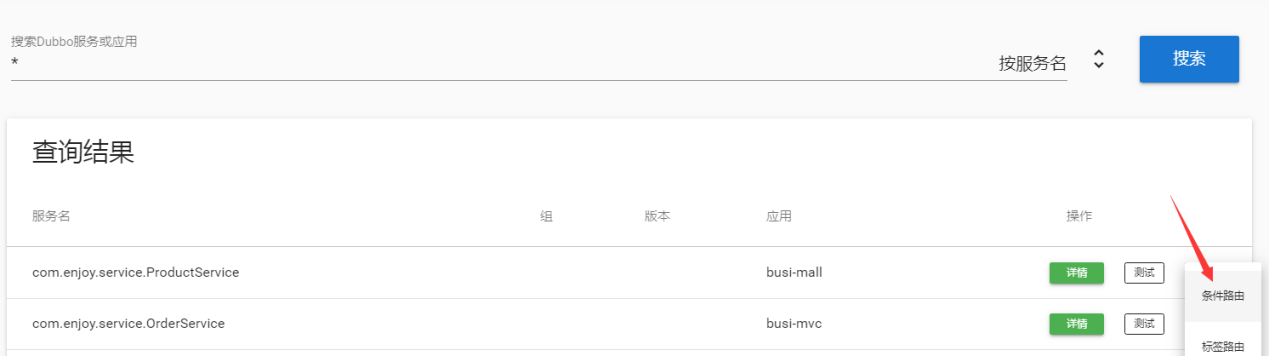
java -jar dubbo-admin-0.1.jar

* 访问：

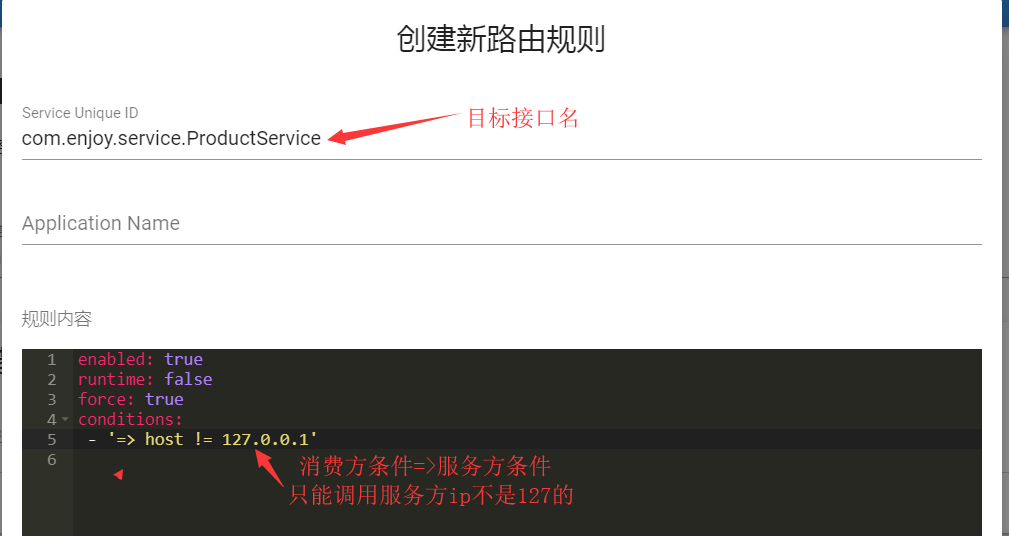
[http://localhost:8080](http://localhost:8080/)

* 条件路由的使用示例：

查询服务列表后，对目标服务添加路由



点击创建按钮后，如下填写条件信息



保存后测试，你会发现消费方已无法调用此服务

## 启动时检查

Dubbo缺省会在启动时检查依赖的服务是否可用，不可用时会抛出异常，阻止Spring初始化完成，以便上线时，能及早发现问题，默认check="true"

可以通过check="false"关闭检查，比如，测试时，有些服务不关心，或者出现了循环依赖，必须有一方先启动

另外，如果你的Spring容器是懒加载的，或者通过API编程延迟引用服务，请关闭check，否则服务临时不可用时，会抛出异常，拿到null引用，如果check="false"，总是会返回引用，当服务恢复时，能自动连上

* 关闭某个服务的启动时检查（没有提供者时报错）：

<dubbo:reference id="xxxService" check="false" interface="com.xxx.XxxService"/>

* 关闭所有服务的启动时检查（没有提供者时报错）：

<dubbo:consumer check="false" />

* 关闭注册中心启动时检查（注册订阅失败时报错）：

<dubbo:registry check="false" />

## Dubbo超时重连

### 超时

Dubbo消费端在发出请求后，需要有一个临界时间界限来判断服务端是否正常。这样消费端达到超时时间，那么Dubbo会进行重试机制，不合理的重试在一些特殊的业务场景下可能会引发很多问题，需要合理设置接口超时时间

Dubbo超时和重试配置示例：

<!-- 服务调用超时设置为5秒,超时不重试-->

<dubbo:reference id="xxxService" interface="com.xxx.XxxService" retries="0" timeout="5000"/>

### 重连

Dubbo在调用服务不成功时，默认会重试2次

Dubbo的路由机制，会把超时的请求路由到其他机器上，而不是本机尝试，所以Dubbo的重试机制也能一定程度的保证服务的质量

## **集群容错**

当消费端某次调用失败是一些环境偶然因素造成的（如网络抖动），dubbo还给予了容错补救机会。补救方式存在以下几种

<dubbo:consumer cluster="failover” retries="2" forks="2" />

1、 Failover ：当出现失败，重试其它服务器。 retries=“2” 来设置重试次数(不含第一次)。

幂等性操作使用，如读操作

2、 Failfast ：快速失败，只发起一次调用，失败立即报错

非幂等性操作，如写操作

3、 Failsafe ：出现异常时，直接忽略

无关紧要的旁支操作，如打日志

4、 Failback ：后台记录失败请求，定时重发

后续专业处理

5、 Forking ：并行调用多个服务器，只要一个成功即返回

forks=“2” 来设置最大并行数

## 负载均衡配置

在集群负载均衡时，Dubbo提供了多种均衡策略，缺省为random随机调用。可以自行扩展负载均衡策略

<dubbo:consumer loadbalance="random"/>

1、 Random ：按权重随机------根据weight值（服务方设置）来随机--

2、 RoundRobin ：轮询

3、 LeastActive ：最少活跃数（正在处理的数），慢的机器，收到的请求少

## 结果缓存

dubbo对方法调用的结果，还有缓存功能。在服务消费方和提供方都可以配置使用缓存。  
以消费方为例，可以配置全局缓存策略，这样所有服务引用都启动缓存  
<dubbo:consumer cache="lru"/>  
还可以仅对某个服务引用配置缓存策略  
<dubbo:reference id="xxxService"   interface="com.xxx.XxxService" cache="lru"  >  
还支持对单个方法启用缓存策略  
<dubbo:reference id="xxxService"    interface="com.xxx.XxxService" >  
   <dubbo:method name="sayHello" cache="lru"> </dubbo:method>  
</dubbo:reference>  
服务方配置方法与消费端完全一样。

## 服务分组

比如，想在同一个注册中心中，分隔测试和开发环境（省一套注册服务）

<dubbo:consumer group ="dev"/>

<dubbo:provider group ="dev"/>

只要group按开发组和测试组对应，同一个注册中心里的两套服务就互不干扰

## 多版本

服务端提供接口的实现升级时，可由dubbo的版本号操作进行过渡。如果上线上测试新版本接口有缺陷，为了不影响业务，要迅速切回原版本接口，最大程度减少损失。

服务方：

<!--版本1接口-->

<dubbo:service interface="com.xxx.XxxServices" ref="xxxService" protocol="hessian" version="1.0"/>

<!--版本2接口-->

<dubbo:service interface="com.xxx.XxxServices" ref="xxxService2" protocol="hessian" version="2.0"/>

消费方：

<dubbo:reference id="xxxService1.0"

interface="com.xxx.XxxServices"

protocol="hessian"

version="2.0"/>

## 只订阅/只注册

### 只订阅

场景：我们在本地开发的时候，不能把自己机器的未开发好的服务注册到开发环境，但是又需要使用注册中心的其他服务

服务提供者配置禁止注册 register="false"

<dubbo:registry protocol="zookeeper" register="false"/>

### 只注册

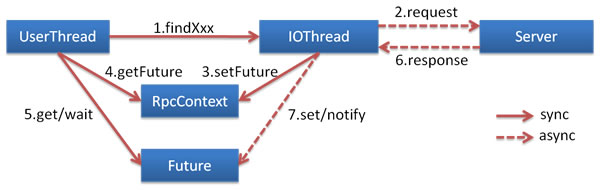
比如开发环境为了省机器，没有部署某个服务，如短信/邮件功能。但整个系统又必须要调用它。此时我们可以借用一下测试环境的此服务（不能影响测试环境原本的服务闭环）。将测试环境的短信/邮件服务也向开发环境注册一份，只注册（其依赖的服务必须还是测试环境的）

服务提供者配置禁止订阅 subscribe="false"

<dubbo:registry protocol="zookeeper"  subscribe="false"/>

## **异步调用**

Dubbo的异步调用是非阻塞的NIO调用，一个线程可同时并发调用多个远程服务，每个服务的调用都是非阻塞的，线程立即返回。就是对java中Futrue模式的扩展支持



如上图，userThread发出调用后，IOThread会立即返回，并在RPC上下文RpcContext中设置Future。userThread后续可以从RpcContext中取得此Future，然后wait这个Future其它的事情都由IOThread完成。

总之，userThread发出调用后IOThread会立刻返回，而不是等调用在服务端执行完代码、返回结果后返回。用户返回后可以去做点其它事情，比如调用另外一个服务，然后回头等待前一个调用完成。从上图可以看出，异步调用完全是Consumer端的行为。

配置：

<dubbo:reference id="xxxService" interface="com.xxx.XxxService">

<dubbo:method name="doSomething" async="true" />

</dubbo:reference>

async=true表示异步调用，可以看到配置发生在Consumer端，能精确到方法。

方法异步调用后，Consumer端的代码写法：

// 此方法异步后不再有返回值，会立刻返回NULL

xxxService.doSomething(xxx);

// 立刻得到当前调用的Future实例，当发生新的调用时这个东西将会被覆盖

Future<XXX> xxxFuture = RpcContext.getContext().getFuture();

// 等待调用完成，线程会进入Sleep状态，当调用完成后被唤醒。

Foo foo = fooFuture.get();

配置是否等待IOThread发送完Request后再返回：

* sent="true" ，等待请求发送出去后再返回，如果发送失败直接抛出异常。
* sent="false" ，将调用交给IOThread后立即返回。实际这个时候请求进入到IOThread的队列，排除等着被发送出去。

<dubbo:method name="xxx" async="true" sent="true" />

如果对返回结果没有兴趣：

<dubbo:method name="xxx" async="true" return="false" />

## 事件通知

对于一次远程方法调用，有oninvoke、onreturn、onthrow三个事件，分别为调用之前、返回之后，抛出异常三个事件。在Consumer端，可以为三个事件指定事件处理方法。

首先，需要在SpringIOC容器中，创建一个实现了回调接口的Bean,假设id=callback

interface Callback {

public void onreturn(Person msg, Integer id);

public void onthrow(Throwable ex, Integer id);

}

方法中，第一个参数是远程方法的返回值，其它是方法的参数。

然后在Dubbo中配置：

<dubbo:reference interface="com.xxx.XxxService" >

<dubbo:method name="doSomeThing" async="true" onreturn = "callback.onreturn" onthrow="callback.onthrow" />

</dubbo:reference>

配置中async与onreturn、onthrow是成对配置，组合不同，功能也不同：

* 异步回调模式：async=true onreturn="xxx"
* 同步回调模式：async=false onreturn="xxx"
* 异步无回调 ：async=true
* 同步无回调 ：async=false

## Provider 端应尽量配置的属性

Dubbo的属性配置优先度上，遵循如下顺序：

reference属性 》 service属性 》Consumer 属性

其中reference和Consumer 是消费端配置，service是服务端配置

而对于服务调用的超时时间、重试次数等属性，服务的提供方比消费方更了解服务性能，因此我们应该在 Provider 端尽量多配置 Consumer 端属性，让其漫游到消费端发挥作用。

Provider 端尽量多配置 Consumer 端的属性，也让 Provider 的实现者一开始就思考 Provider 端的服务特点和服务质量等问题。

建议在 Provider 端配置的 Consumer 端属性有：

timeout：方法调用的超时时间

retries：失败重试次数

loadbalance：负载均衡算法，缺省是随机 random。

actives：消费者端的最大并发调用限制，即当 Consumer 对一个服务的并发调用到上限后，新调用会阻塞直到超时，在方法上配置 dubbo:method 则针对该方法进行并发限制，在接口上配置 dubbo:service，则针对该服务进行并发限制

<dubbo:service interface="com.xxx.xxxService" timeout="300" retries="2" loadbalance="random" actives="0" >

<dubbo:method name="xxx" timeout="10000" retries="9" loadbalance="leastactive" actives="5" />

<dubbo:service/>

建议在 Provider 端配置的 Provider 端属性有：

threads：服务线程池大小

executes：一个服务提供者并行执行请求上限，即当 Provider 对一个服务的并发调用达到上限后，新调用会阻塞，此时 Consumer 可能会超时。在方法上配置 dubbo:method 则针对该方法进行并发限制，在接口上配置 dubbo:service，则针对该服务进行并发限制

<dubbo:protocol threads="200" />

<dubbo:service interface="com.xxx.xxxService" version="1.0.0" ref="xxxService" executes="200" >

<dubbo:method name="xxx" executes="50" />

</dubbo:service>

## 服务拆分最佳实现

**分包**

建议将服务接口、服务模型、服务异常等均放在 API 包中，因为服务模型和异常也是 API 的一部分，这样做也符合分包原则：重用发布等价原则(REP：复用的粒度即是发布的粒度，我们所重用的任何东西必须同时被发布和跟踪)，共同重用原则(CRP：如果你重用了一个组件中的一个类，那么就要重用包中的所有类)。

**粒度**

服务接口尽可能大粒度，每个服务方法应代表一个功能，而不是某功能的一个步骤，否则将面临分布式事务问题，Dubbo 暂未提供分布式事务支持。

不建议使用过于抽象的通用接口，如：Map query(Map)，这样的接口没有明确语义，会给后期维护带来不便。

**版本**

每个接口都应定义版本号，为后续不兼容升级提供可能，如： <dubbo:service interface="com.xxx.XxxService" version="1.0" />。

建议使用两位版本号，因为第三位版本号通常表示兼容升级，只有不兼容时才需要变更服务版本。

当不兼容时，先升级一半提供者为新版本，再将消费者全部升为新版本，然后将剩下的一半提供者升为新版本。

**异常**

建议使用异常汇报错误，而不是返回错误码，异常信息能携带更多信息，并且语义更友好。

## Dubbo的线程模型

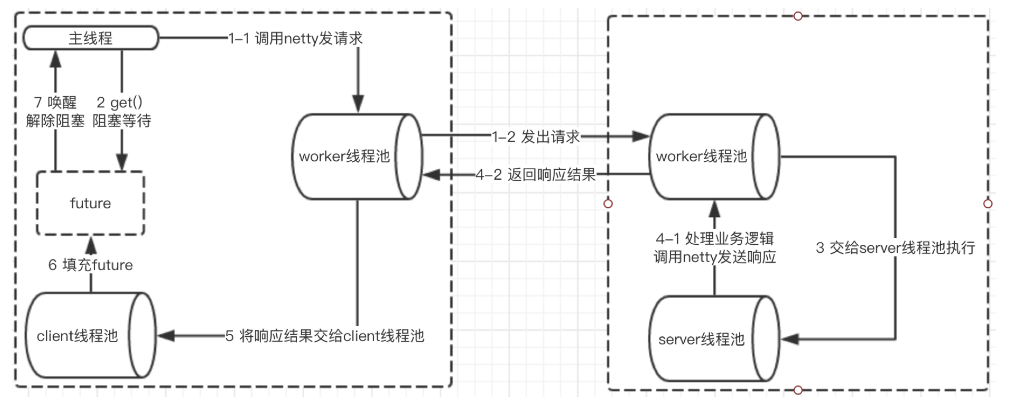
首先明确一个基本概念：IO线程和业务线程的区别

* IO线程：配置在netty连接点的用于处理网络数据的线程，主要处理编解码等直接与网络数据打交道的事件。
* 业务线程：用于处理具体业务逻辑的线程，可以理解为自己在provider上写的代码所执行的线程环境。

Dubbo默认采用的是长链接的方式，即默认情况下一个consumer和一个provider之间只会建立一条链接，这种情况下IO线程的工作就是编码和解码数据，监听具体的数据请求，直接通过Channel发布数据等等；

业务线程就是处理IO线程处理之后的数据，业务线程并不知道任何跟网络相关的内容，只是纯粹的处理业务逻辑，在业务处理逻辑的时候往往存在复杂的逻辑，所以业务线程池的配置往往都要比IO线程池的配置大很多。

IO线程部分，Netty服务提供方NettyServer又使用了两级线程池，master主要用来接受客户端的链接请求，并把接受的请求分发给worker 来处理。整个过程如下图：



IO线程与业务线程的交互如下，：

**IO线程的派发策略**：

* 默认是all：所有消息都派发到线程池，包括请求，响应，连接事件，断开事件，心跳等。 即worker线程接收到事件后，将该事件提交到业务线程池中，自己再去处理其他IO事件。
* direct：worker线程接收到事件后，由worker执行到底。
* message：只有请求响应消息派发到线程池，其它连接断开事件，心跳等消息，直接在 IO线程上执行
* execution：只有请求消息派发到线程池，不含响应（客户端线程池），响应和其它连接断开事件，心跳等消息，直接在 IO 线程上执行
* connection：在 IO 线程上，将连接断开事件放入队列，有序逐个执行，其它消息派发到线程池。

**业务线程池设置**：

* fixed：固定大小线程池，启动时建立线程，不关闭，一直持有。(缺省)
  + coresize：200
  + maxsize：200
  + 队列：SynchronousQueue
  + 回绝策略：AbortPolicyWithReport - 打印线程信息jstack，之后抛出异常
* cached：缓存线程池，空闲一分钟自动删除，需要时重建。
* limited：可伸缩线程池，但池中的线程数只会增长不会收缩。只增长不收缩的目的是为了避免收缩时突然来了大流量引起的性能问题。

**配置示例：**

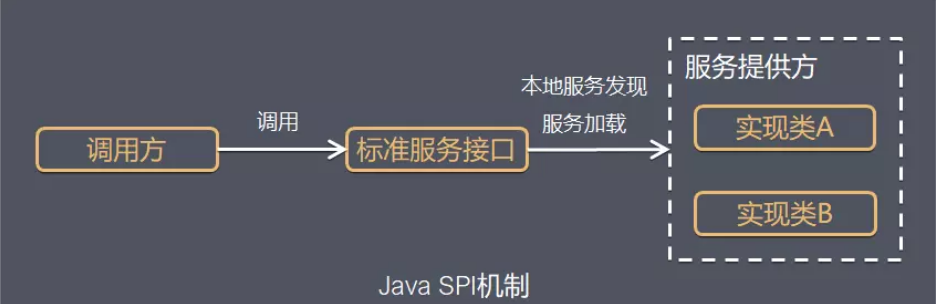
|  |
| --- |
| <dubbo:protocol name="dubbo"dispatcher="all"threadpool="fixed"threads="100"/> |

# SPI机制原理

因dubbo框架是建立的SPI机制上，因此在探寻dubbo框架源码前，我们需要先把SPI机制了解透彻

## java spi机制

SPI全称Service Provider Interface，是Java提供的一套用来被第三方实现或者扩展的API，它可以用来启用框架扩展和替换组件。



可以看到，SPI的本质，其实是帮助程序，为某个特定的接口寻找它的实现类。而且哪些实现类的会加载，是个动态过程（不是提前预定好的）。

有点类似IOC的思想，就是将装配的控制权移到程序之外，在模块化设计中这个机制尤其重要。所以SPI的核心思想就是**解耦**。

比较常见的例子：

* 数据库驱动加载接口实现类的加载  
  JDBC加载不同类型数据库的驱动
* 日志门面接口实现类加载  
  SLF4J加载不同提供商的日志实现类
* Spring  
  Spring中大量使用了SPI,比如：对servlet3.0规范对ServletContainerInitializer的实现、自动类型转换Type Conversion SPI(Converter SPI、Formatter SPI)等

### 使用介绍

要使用Java SPI，需要遵循如下约定：

* 1、当服务提供者提供了接口的一种具体实现后，在jar包的META-INF/services目录下创建一个以“接口全限定名”为命名的文件，内容为实现类的全限定名；
* 2、接口实现类所在的jar包放在主程序的classpath中；
* 3、主程序通过java.util.ServiceLoder动态装载实现模块，它通过扫描META-INF/services目录下的配置文件找到实现类的全限定名，把类加载到JVM；
* 4、SPI的实现类必须携带一个不带参数的构造方法；

**示例**

先定义一个接口（代码示例见源码包）

public interface InfoService {  
 Object sayHello(String name) ;  
}

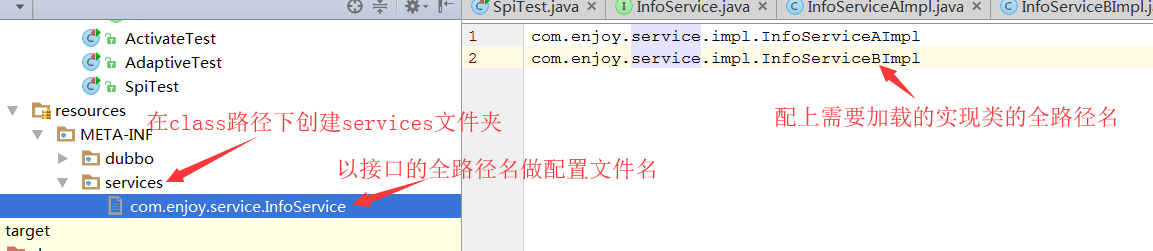
再定义一系列它的实现

public class InfoServiceAImpl implements InfoService {  
 @Override  
 public Object sayHello(String name) {  
 System.out.println(name+",你好，调通了A实现！");  
 return name+",你好，调通了A实现！";  
 }  
}

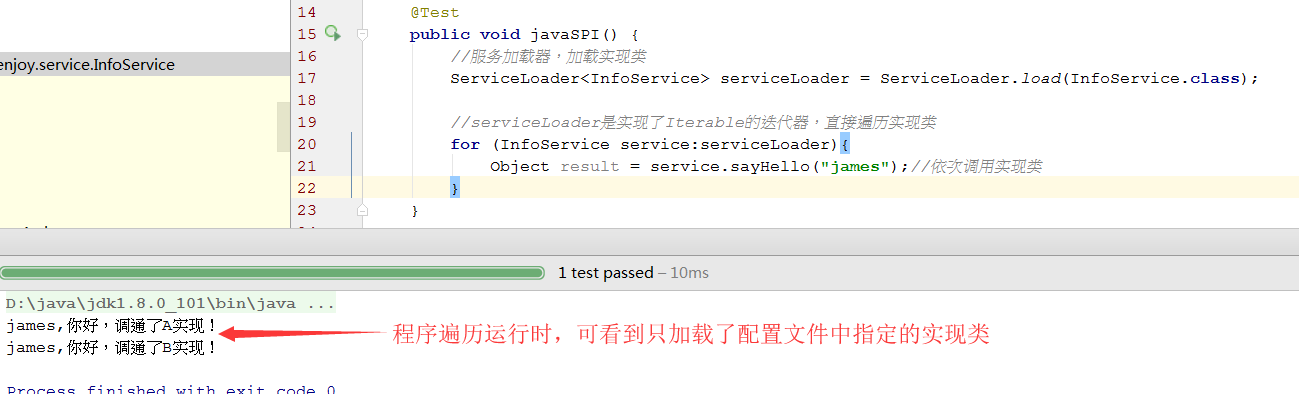
第二个实现

public class InfoServiceBImpl implements InfoService {  
 @Override  
 public Object sayHello(String name) {  
 System.out.println(name+",你好，调通了B实现！");  
 return name+",你好，调通了B实现！";  
 }  
}

...等等实现



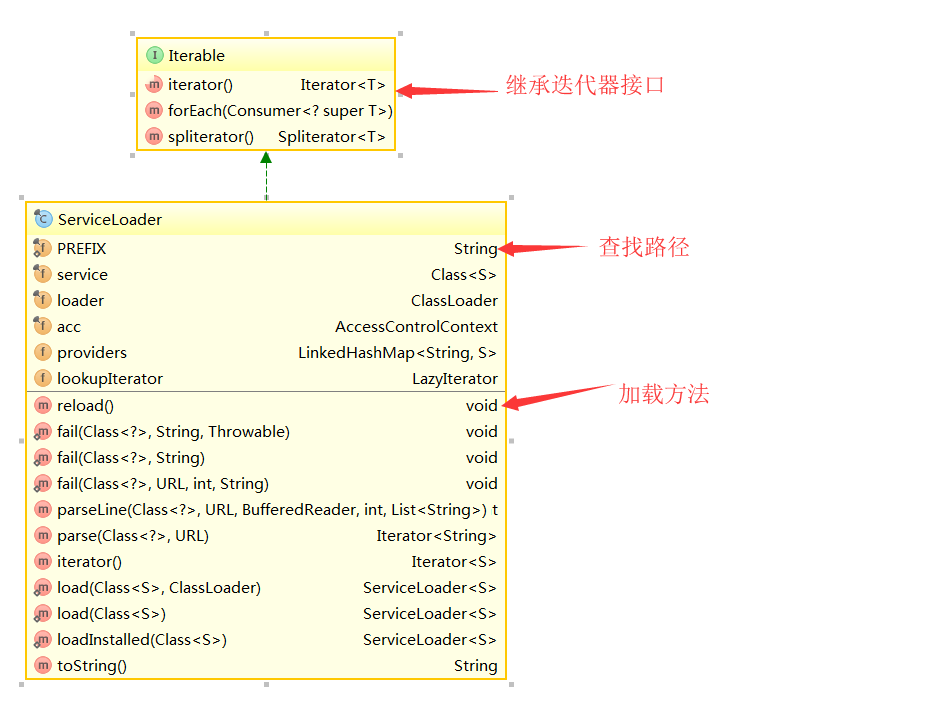
**测试**



至此，整个java SPI的机制使用介绍完毕。

### 核心功能类

需要指出的是，java之所以能够顺利根据配置加载这个实现类，完全依赖于jdk内的一个核心类：



## Dubbo SPI机制

在上一节中，可以看到，java spi机制非常简单，就是读取指定的配置文件，将所有的类都加载到程序中。而这种机制，存在很多缺陷，比如：

1. 所有实现类无论是否使用，直接被加载，可能存在浪费

2. 不能够灵活控制什么时候什么时机，匹配什么实现，功能太弱

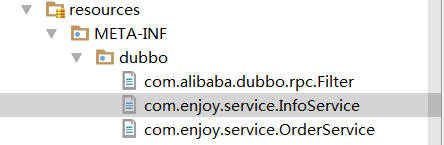
Dubbo基于自己的需要，增强了这套SPI机制，下面介绍Dubbo中的SPI用法。

### 标签@SPI用法

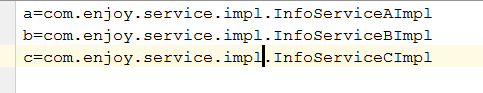
与 Java SPI 实现类配置不同，Dubbo SPI 是通过键值对的方式进行配置，这样我们就可以按需加载指定的实现类。另外，需要在接口上标注 @SPI 注解。表明此接口是SPI的扩展点：

@SPI("b")  
public interface InfoService {  
 Object sayHello(String name) ;  
 Object passInfo(String msg, URL url) ;  
}

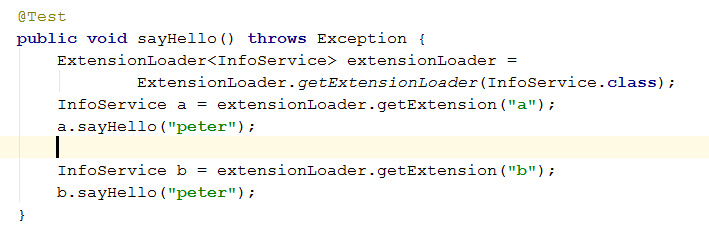
dubbo的配置文件夹路径：



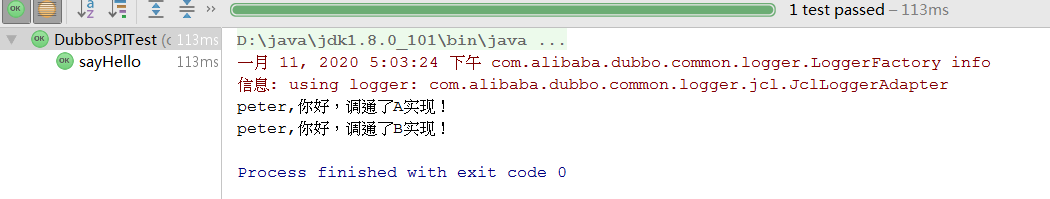
配置文件内容：



测试功能，指定加载某个实现



测试结果



### 标签@Activate用法

Dubbo的Spi机制虽然对原生SPI有了增强，但功能还远远不够。

在工作中，某种时候存在这样的情形，需要同时启用某个接口的多个实现类，如Filter过滤器。我们希望某种条件下启用这一批实现，而另一种情况下启用那一批实现，比如：希望的RPC调用的消费端和服务端，分别启用不同的两批Filter，怎么处理呢？

这时我们的条件激活注解@Activate，就派上了用场

Activate注解表示一个扩展是否被激活(使用),可以放在类定义和方法上，dubbo用它在spi扩展类定义上，表示这个扩展实现激活条件和时机。它有两个设置过滤条件的字段，group，value 都是字符数组。 用来指定这个扩展类在什么条件下激活。

下面以com.alibaba.dubbo.rpc.filter接口的几个扩展来说明。

@Activate(group = {Constants.PROVIDER, Constants.CONSUMER})

public class testActivate1 implements Filter {

}  
//表示如果过滤器使用方（通过group指定）属于Constants.PROVIDER（服务提供方）或者Constants.CONSUMER（服务消费方）就激活使用这个过滤器

//再看这个扩展

@Activate(group = Constants.PROVIDER, value = Constants.TOKEN\_KEY)

public class testActivate2 implements Filter {

}  
//表示如果过滤器使用方（通过group指定）属于Constants.PROVIDER（服务提供方）并且 URL中有参数 Constants.TOKEN\_KEY（token）时就激活使用这个过滤器

详细用法见源码demo

### javassist动态编译

在SPI寻找实现类的过程中，getAdaptiveExtension方法得到的对象，只是个接口代理对象，此代理对象是由临时编译的类来实现的。在此，先说明一个javassist动态编译类的两种用法：

通过创建class模型对象设置class属性，然后生成Class：

1. CtClass-->CtField-->CtMethod

2. Class<?> clazz = ctClass.toClass()

直接编译拼凑好的定义class的字符串，来生成class：

JavassistCompiler.compile(**"public class DemoImpl implements DemoService {...}"**,ClassLoader());

### 标签@Adaptive用法

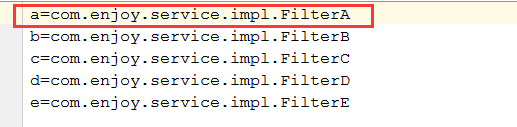
我们在前面演示了dubbo SPI的使用，但是有一个问题，扩展点对应的实现类不能在程序运行时动态指定，就是extensionLoader.getExtension方法写死了扩展点对应的实现类，不能在程序运行期间根据运行时参数进行动态改变。

而我们希望在程序使用时，对实现类进行懒加载，并且能根据运行时情况来决定，应该启用哪个扩展类。为了解决这个问题，dubbo引入了Adaptive注解，也就是dubbo 的自适应机制。

先看下面的示例：

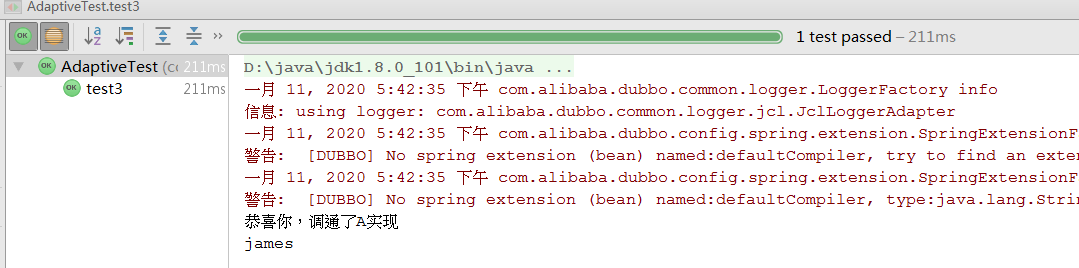
**public void** test3(){  
 ExtensionLoader<InfoService> loader = ExtensionLoader.*getExtensionLoader*(InfoService.**class**);  
 InfoService adaptiveExtension = loader.getAdaptiveExtension();  
 URL url = URL.*valueOf*(**"test://localhost/test?info.service=a"**);  
 System.***out***.println(adaptiveExtension.passInfo(**"james"**, url));  
}

我们的InfoService 的passInfo方法参数内，有一个URL的参数。URL中附带了信息**info.service=a，**希望调用a实现。a实现的配置如下：



这初看起来非常矛盾，都已经在调用InfoService对象了，怎么还有机会来选择调用哪个InfoService对象呢？

其实重点就在于，现在的InfoService 的调用对象adaptiveExtension ，在当前，还只是个代理类，因此我们还有在代理内选择哪个目标实现的机会。



我们运行代码，会发现还真就是调用的A实现类

使用重点，URL的格式：

**info.service=a的参数名格式，是接口类InfoService的驼峰大小写拆分**

### Dubbo SPI的依赖注入

Dubbo SPI的核心实现类为ExtensionLoader，此类的使用几乎遍及Dubbo的整个源码体系。是大家以传统方式读源码的严重障碍。

ExtensionLoader有三个重要的入口方法，分别与@SPI、@Activate、@Adaptive注解对应。

getExtension方法，对应加载所有的实现

getActivateExtension方法，对应解析加载@Activate注解对应的实现

getAdaptiveExtension方法，对应解析加载@Adaptive注解对应的实现

其中，@Adaptive注解作的自适应功能，还涉及到了代理对象（而Dubbo的代理机制，有两种选择，jdk动态代理和javassist动态编译类）。我们将后后续篇章对此进行说明。

Dubbo的SPI机制，除上以上三种注解的用法外，还有一个重要的功能依赖注入

对于spring这个强大的IOC工具，依赖注入大家一定都很了妥！在Dubbo自动生成SPI的扩展实例的时候也会发生依赖注入的场景，举一个具体的例子。

现有一个接口扩展点

@SPI(**"peter"**)  
**public interface** OrderService {  
 String getDetail(String id, URL url);  
}

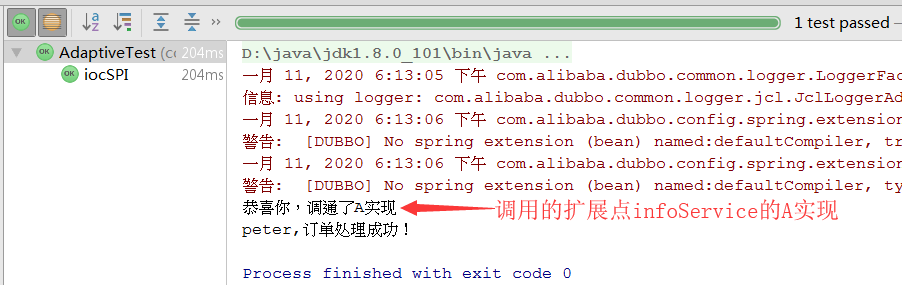
而其实现类中，引入了另一个扩展点接口对象**infoService**

**public class** OrderServiceImpl **implements** OrderService {  
 **private** InfoService **infoService**;  
  
 **public void** setInfoService(InfoService infoService) {  
 **this**.**infoService** = infoService;  
 }  
  
 @Override  
 **public** String getDetail(String name, URL url) {  
 **infoService**.passInfo(name,url);  
 System.***out***.println(name+**",订单处理成功！"**);  
 **return** name+**",你好，订单处理成功！"**;  
 }  
}

此时，假如我们的OrderService 在加载时会发生什么呢？

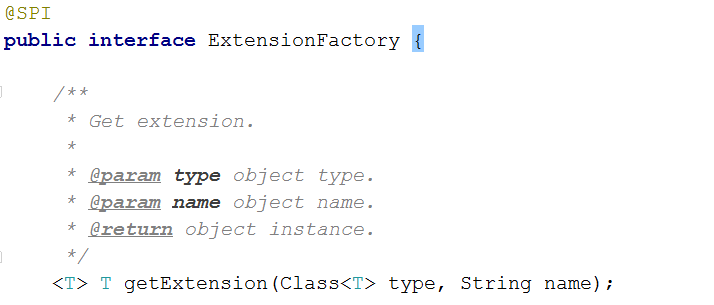
@Test  
**public void** iocSPI() {  
 *//获取OrderService的 Loader 实例* ExtensionLoader<OrderService> extensionLoader = ExtensionLoader.*getExtensionLoader*(OrderService.**class**);  
 *//取得默认拓展类* OrderService orderService = extensionLoader.getDefaultExtension();  
 URL url = URL.*valueOf*(**"test://localhost/test?info.service=a"**);  
 orderService.getDetail(**"peter"**,url);  
}

结果如下：

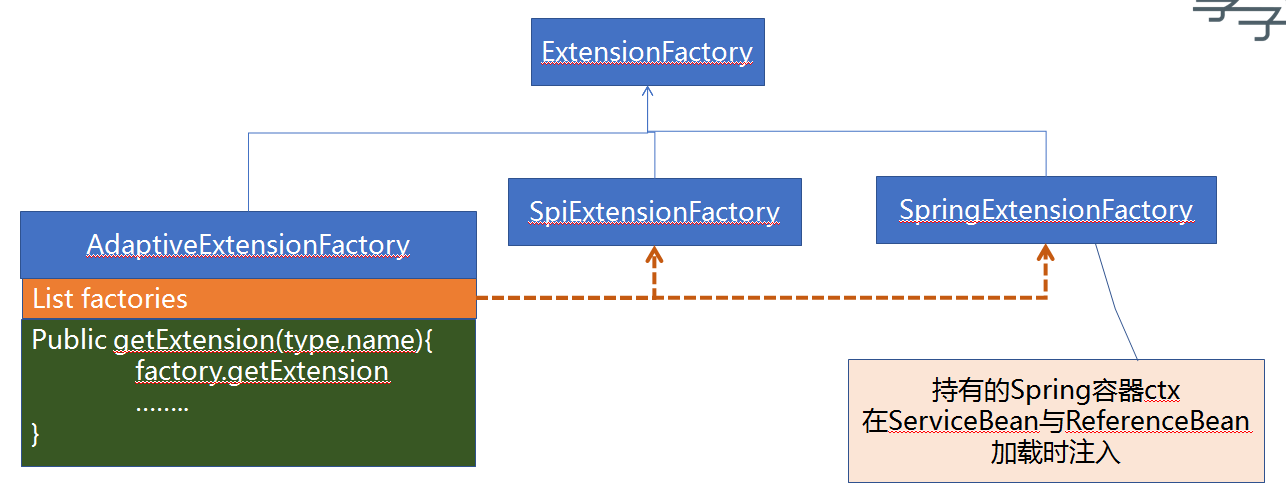


可以看到，dubbo自动生成了实例，并注入了依赖之中。

这是什么机理呢，看这个扩展接口：



这是dubbo的一个扩展点工厂接口，只有一个方法，根据class和name查找实现。这个接口，是一个扩展点，接下来看看此接口的实现类



可以看到，有两个实现类，一个适配类（adaptive，接口的默认实现）

AdaptiveExtensionFactory在内部持有了所有的factory实现工厂，即后两个实现类。一个为SPI工厂（依赖类是扩展接口时发挥作用），一个为Spring工厂（依赖的是springbean时发挥作用）。于是，当我们需要为某个生成的对象注入依赖时，直接调用此对象即可。

至此，整套DubboSPI的IOC功能圆满了。

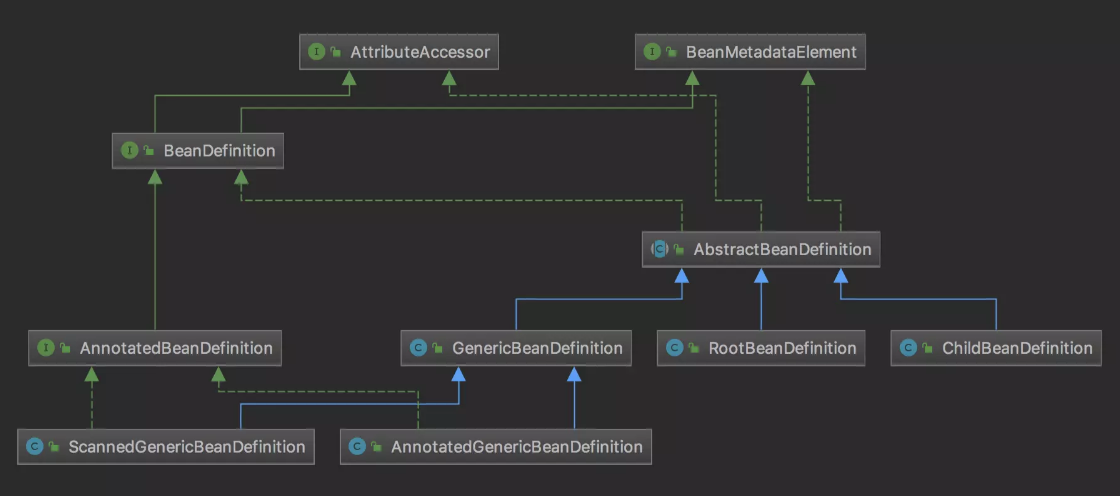
# Dubbo启动过程探析

## Spring简介BeanDefinition

在Java中，一切皆对象。在JDK中使用java.lang.Class来描述类这个对象。

而在Spring中，bean对象是操作核心。那么Spring也需要一个东西来描述bean这个对象，它就是BeanDefinition。

有兴趣的同学，可以详细追一下spring的基础流程，其整体类图如下：



详细的spring原理不属于本课的内容，这里只回顾下BeanDefinition的基础用法：

|  |
| --- |
| RootBeanDefinition beanDef = **new** RootBeanDefinition();  beanDef.setBeanClass(DemoServiceImpl.**class**); beanDef.setBeanClassName(DemoServiceImpl.**class**.getName()); beanDef.setScope(BeanDefinition.***SCOPE\_SINGLETON***);*//单例  //设置属性* MutablePropertyValues propertyValues = beanDef.getPropertyValues(); propertyValues.addPropertyValue(**"type"**,**"runtime"**);  *//注册bean* applicationContext.registerBeanDefinition(**"demoService"**, beanDef); |

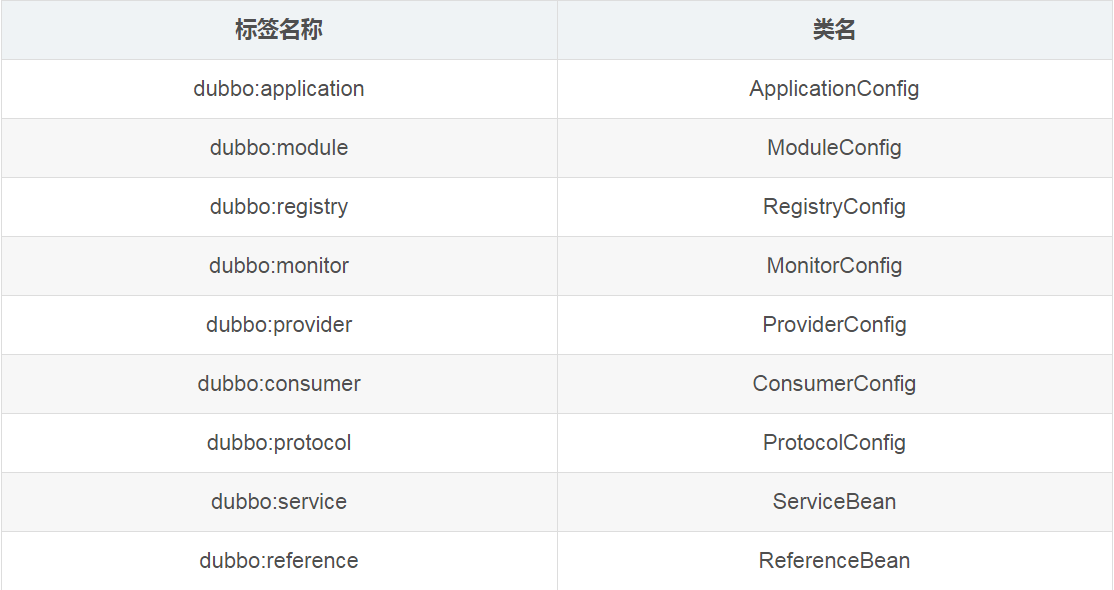
可以看到，最后的spring动作applicationContext.registerBeanDefinition会在IOC容器内创建描述的bean对象。具体验证可参见demo代码

后续Dubbo的所有对象创建，皆以此形式委托给Spring来创建

## Dubbo的配置解析过程

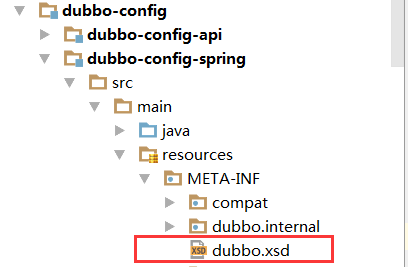
dubbo的配置解析，不论是xml方式的配置，还是注解的配置，目标都是把配置的属性值提取出来，变成dubbo的组件bean（先由BeanDefinition描述，然后委托spring生成组件bean）。

下面以xml的标签为例，列出每种配置对应要解析成为的目标组件bean

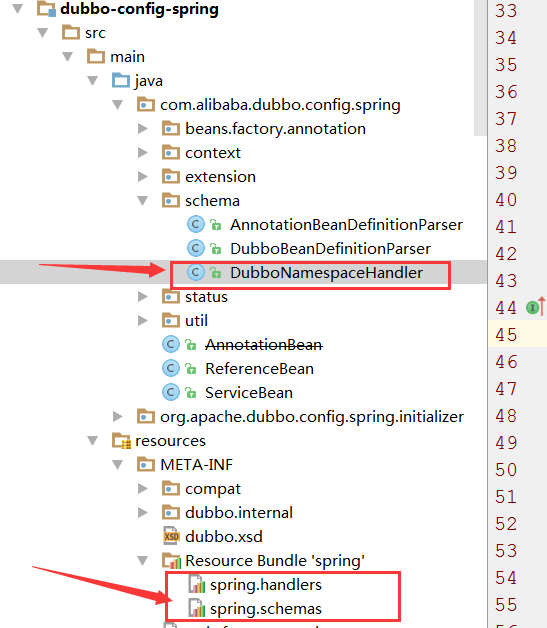


### xml配置的解析过程

首先，dubbo自定义了spring标签描述dubbo.xsd。在dubbo-config-spring模块下的 src/main/resouce/META-INF下



其次，在spring.handlers、spring.schemas中指定解析类，将标签引入spring中管理。

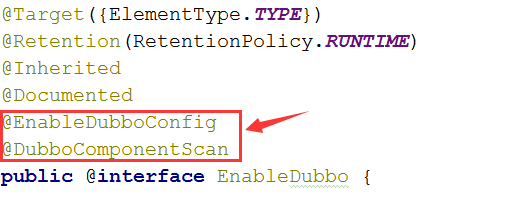


DubboBeanDefinitionParser继承了spring的BeanDefinitionParser接口，spring会调用parse方法来读取每个标签配置，将属性值装入对应的BeanDefinition定义中，后续spring会根据此BeanDefinition定义生成dubbo的组件bean。

### 注解的解析过程

Dubbo注解的目标，与xml一致，都是将配置信息变为BeanDefinition定义，交由spring生成组件bean。

注解的源头是@EnableDubbo用于启用dubbo配置。而此注解又引入EnableDubboConfig和DubboComponentScan注解。

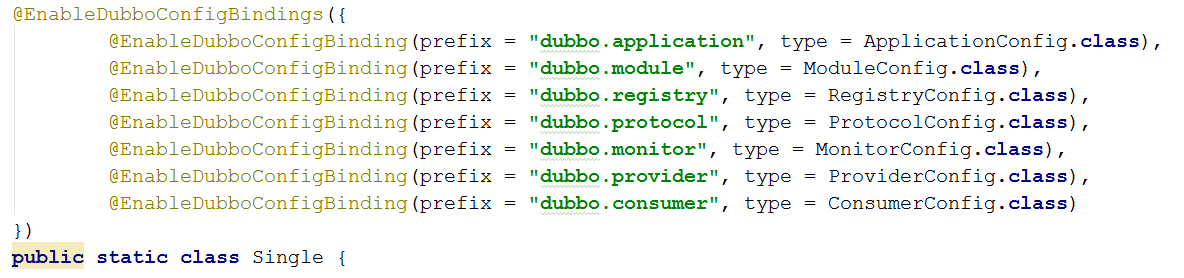


其中，@EnableDubboConfig主要用于处理dubbo中全局性的组件配置，一般在.properties文件中的配置项，如Application/Registry/Protocol/Provider/consumer ，而@DubboComponentScan负责扫描项目源代码，处理业务类上的Reference/Service 注解

#### @EnableDubboConfig的过程

@EnableDubboConfig主要用来解析属性文件中的配置，一般在springboot项目中比较常用，过程如下：

通过DubboConfigConfigurationSelector类，连接到DubboConfigConfiguration类，此处配置了它支持解析的所有注解组件，如下：



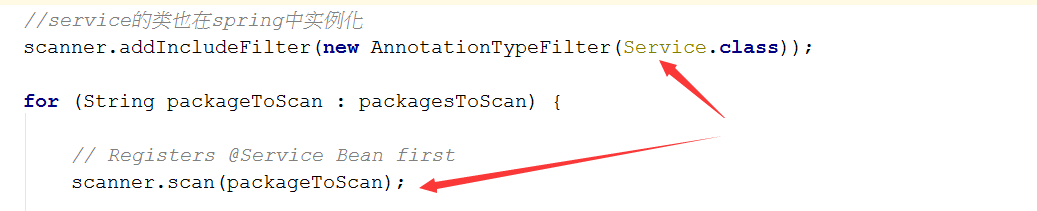
此处为每个dubbo组件绑定了属性文件的前缀值。具体的处理过程在@EnableDubboConfigBinding中，最终引入到DubboConfigBindingRegistrar类来完成组件bean注册。

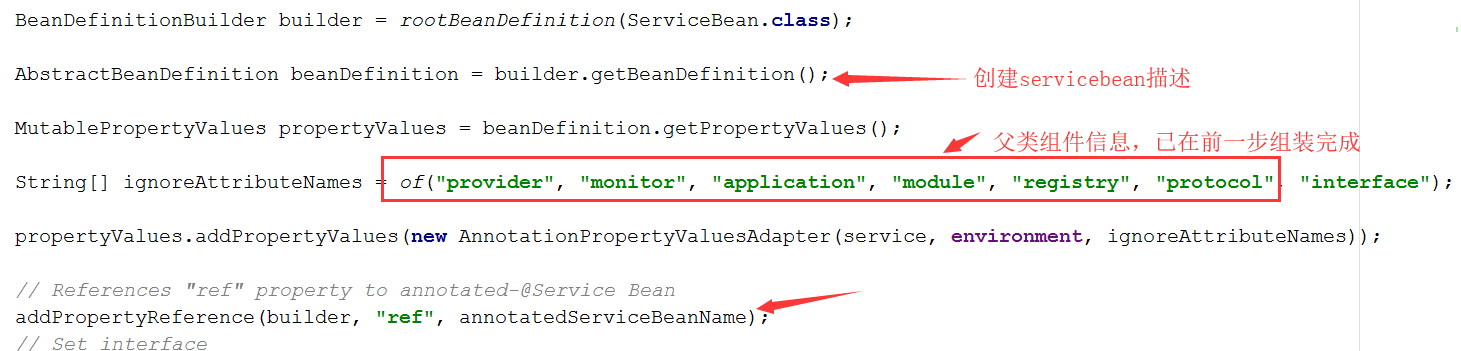
#### @DubboComponentScan的过程

DubboComponentScan用来解析，标注在业务Service实现上的注解，主要是暴露业务服务的@Service 和引入服务的@Reference。总入口在DubboComponentScanRegistrar上。可以看到，两个注解的处理是分开处理的。

##### service注解

service注解的处理，最终由serviceAnnotationBeanPostProcessor来处理。整个过程比较简单，dubbo先调用spring扫描包处理



此处，先由spring将暴露的业务服务实例化。然后再创建dubbo的组件对象

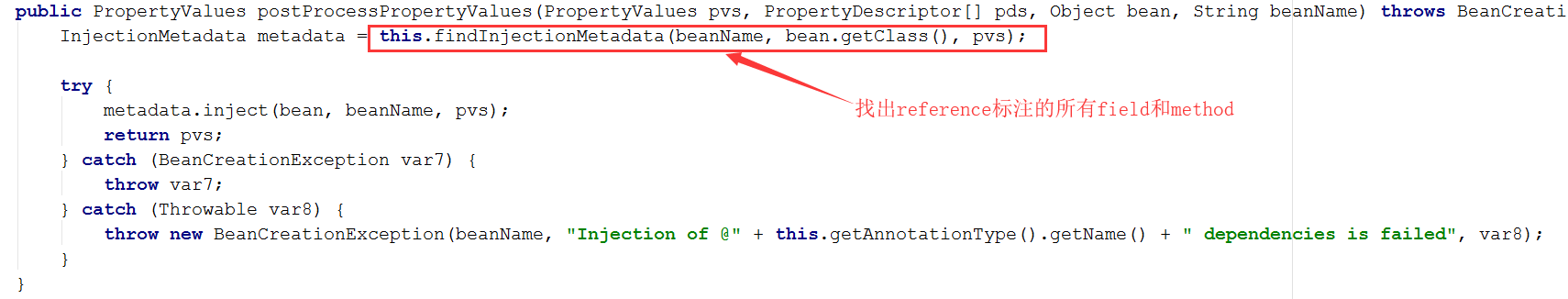
servicebean有很多父级组件信息引入，这些组件已经在属性文件处理部分完成。

servicebean中将ref指向业务bean

##### reference注解

reference注解的处理，最终由ReferenceAnnotationBeanPostProcessor来处理。

ReferenceAnnotationBeanPostProcessor继承于AnnotationInjectedBeanPostProcessor实现了MergedBeanDefinitionPostProcessor接口，方法postProcessMergedBeanDefinition在创建bean实例前会被调用（用来找出bean中含有@Reference注解的Field和Method）



然后使用这一句metadata.inject(bean, beanName, pvs)对字段和方案进行反射绑定。

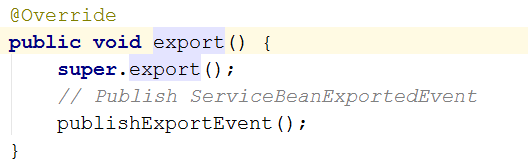
当Spring完成bean的创建后会调用AbstractAutowireCapableBeanFactory#populateBean方法完成属性的填充

## Dubbo的服务暴露过程

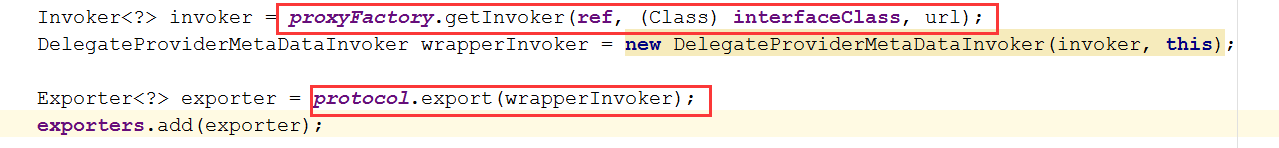
前面已经看到，dubbo的组件中，servicebean和referencebean是比较特殊的。这两个组件，将完成dubbo服务的远程rpc过程。

servicebean作为服务端，会在bean创建成功后，发起服务暴露流程。其过程如下：

在实现的InitializingBean接口中，spring调用afterPropertiesSet方法，发起服务的暴露动作。



父类中最终执行此动作



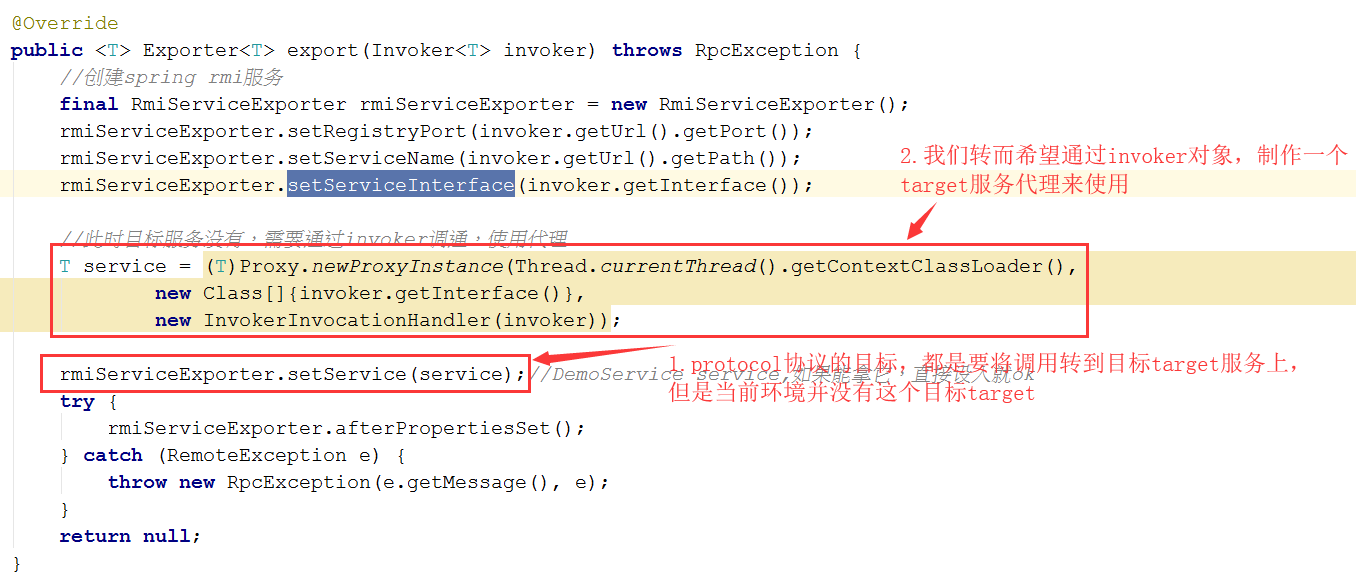
先将本地服务ref包装成invoker，然后由protocol网络协议将invoker连通到网络上。其核心，即是一旦有protocol网络传输过来的请求，则拉起invoker动作，并将动作传递到ref服务上进行响应。在这个整个过程中，dubbo不希望每一个具体的协议ptotocol去关心目标服务是谁（耦合性太强），于是中间插入了一个invoker概念实体来解耦双方的绑定关系（重点）

为了详细说明白这个过程，我们细细探究看一下invoker的机理：

* 首先，明确目标，invoker的本义，是invoke方法会转嫁对象到目标ref上（具体怎么转嫁，后续会演示），接口定义如下



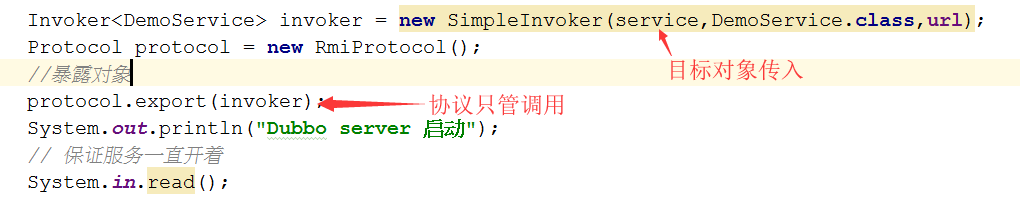
* 协议使用方，希望如此调用invoker：



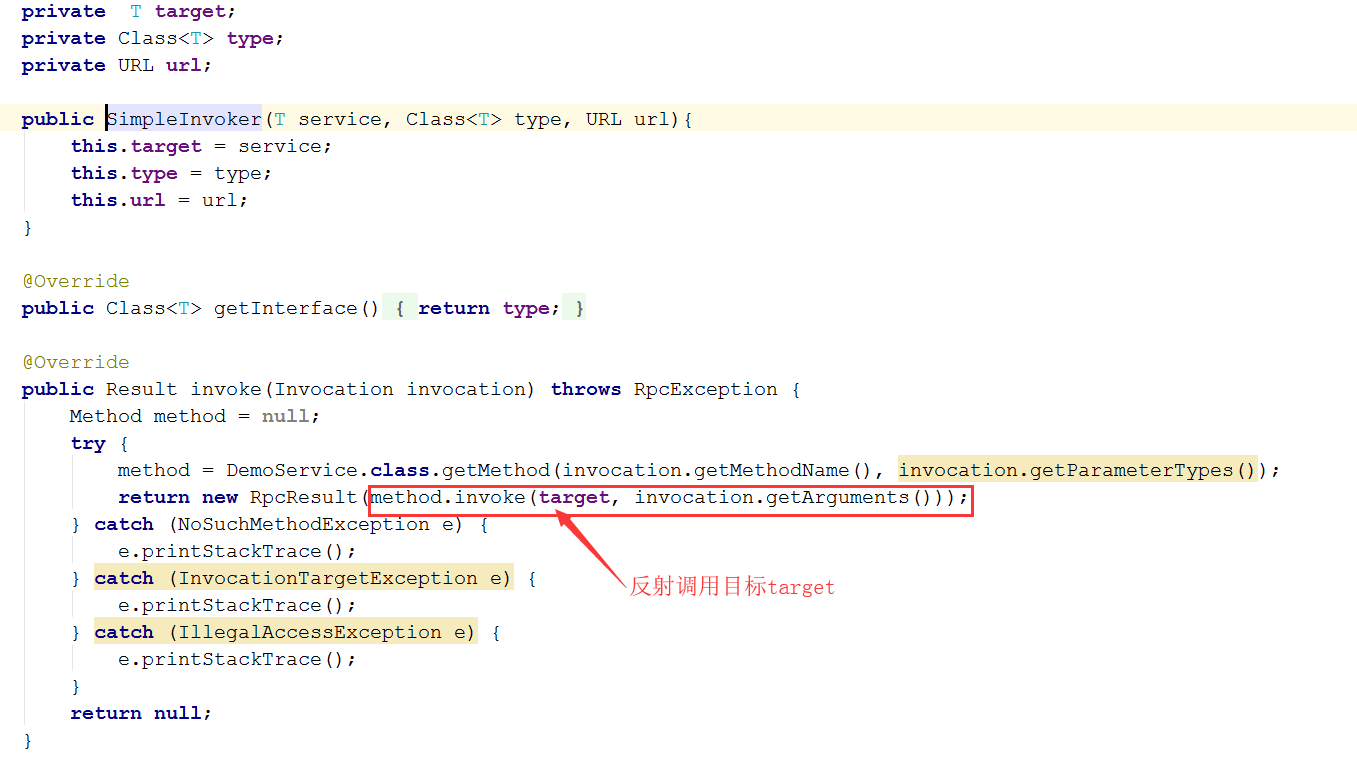
意思是，希望dubbo中的所有协议都按此模式，将网络请求转给invoker对象即可，剩下的protocol协议不要去关心。

* Invoker如何转请求到target目标服务呢？

交给Invoker的实现类去具体实现，下面是个例子，invoker中本身持有目标target服务



后续的调用不言自明，肯定是通过java反射将方法转给target服务



* 总结上述过程：

1、protocol组件收到网络请求到来时，它需要将请求发向target目标服务（如果当前环境中有此服务就好了）。

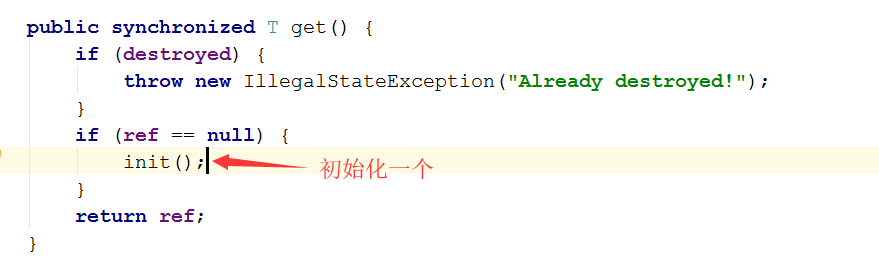
2、因为当前环境中没有target对象，于是它创建了一个target的代理对象proxy，将请求转给了此代理proxy对象，而此proxy对象只会干一件事，将调用甩锅给了invoker对象的invoke方法

3、invoker对象发现自己内部有target对象，调用so easy，于是它使用java反射，将请求发向了target服务

PS：我们可以想到，如果让protocol中持有target服务，直接转向请求到target要简单得多，但这样一来，每一个ptotocol服务要对接千千万万的业务service接口，耦合性太强。于是，dubbo专门设计了invoker实体来解开两者间的直接耦合（工作中可否借鉴？）

## Dubbo的服务引入过程

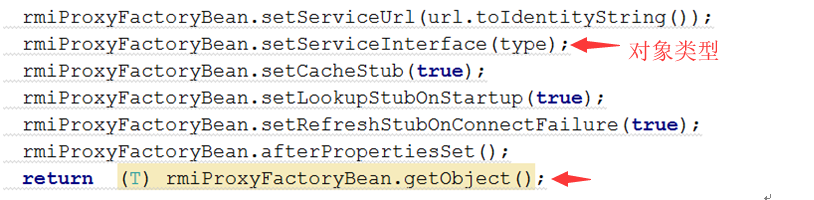
dubbo服务的引入过程，是在referencebean的实例化过程中实现的。当dubbo启动过程中，遇到@reference，即会创建一个referencebean的实例。此实例一样实现了InitializingBean接口，在其调用的afterPropertiesSet方法中，会为服务调用方创建一个远程代理对象



ref是通过interface和url信息生成的代理



意思即是说，protocol协议制作了一个invoker对象，你可以通过invoker对象，向protocol协议发送信息（网络传输）。借用springrmi协议来说明一个这个过程：

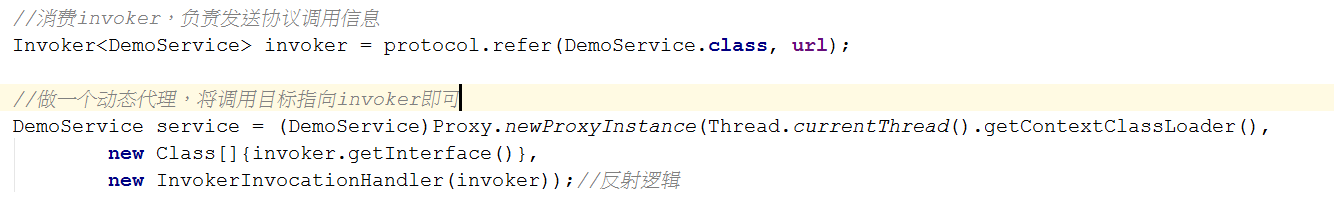


一个protocol协议建立后，会得到一个object输出对象，输出到网络信息的动作，全仗此对象进行。

本来，此对象类型已经是我们的业务接口类型，我们可以直接使用此对象进行通信了。但是，考虑到protocol本身不应该跟具体的业务接口耦合，于是，我们再次插入了invoker实体来解耦双方

1、将protocol生成的输出对象object，包装成invoker对象

2、在业务操作端，为了方便操作，再做一个代理对象，来转请求到invoker上



PS：Dubbo中的invoker概念，作用不仅仅于此，它统一了dubbo中各组件间相互交流的规范，大家统一都用invoker进行粘合（书同文、车同轴）。我们后续将继续展示invoker更高一层的作用