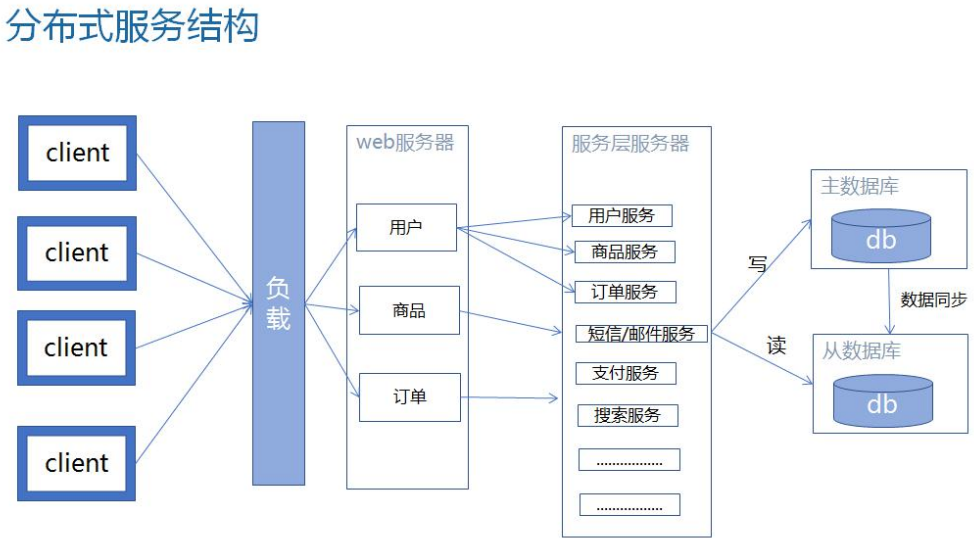
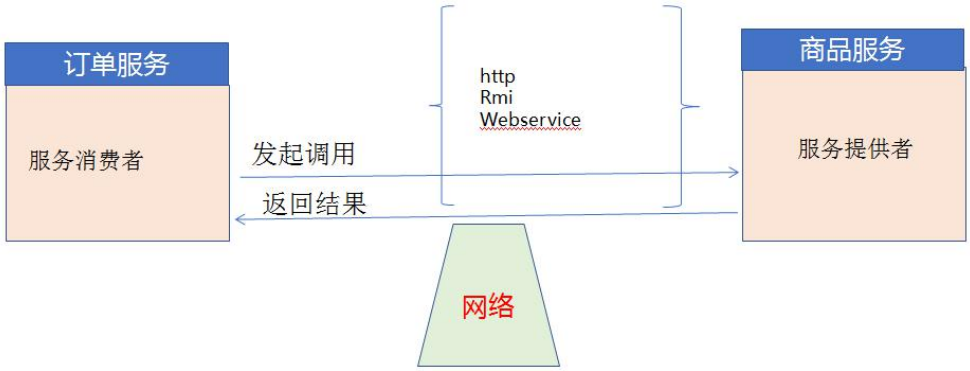
1 RPC 场景和过程

1.1RPC 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：



1.2RPC 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

1.3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：

1.3.1 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;
    }
}
```

1.3.2 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();
//注册通讯端口
LocateRegistry.createRegistry(9080);
//注册通讯路径
Naming.bind("rmi://127.0.0.1:9080/InfoService
", infoService);
```

1.3.3 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

1.4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

1.5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：

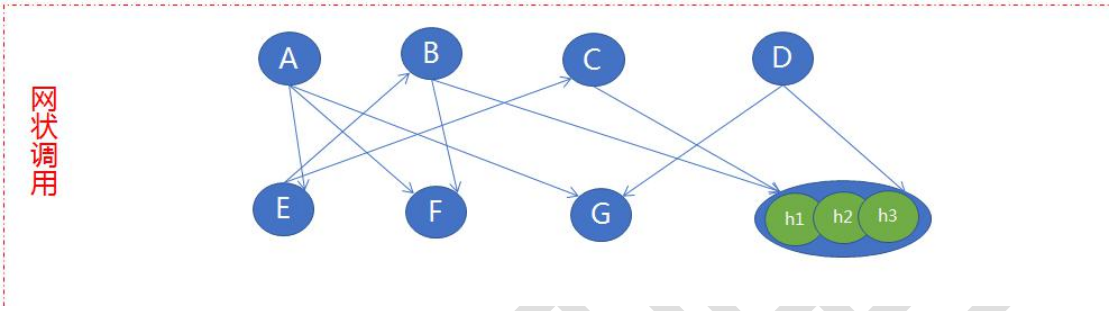
```
OrderService service = new OrderService(){
    @Override
    public OrderEntiry getDetail(String id) {
        Map<String,String> info = new HashMap();
        //写死了反射的目标，静态代理
        info.put("target","orderService");//对象
        info.put("methodName","getDetail");//方法
        info.put("arg",id);//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

1.6Dubbo 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



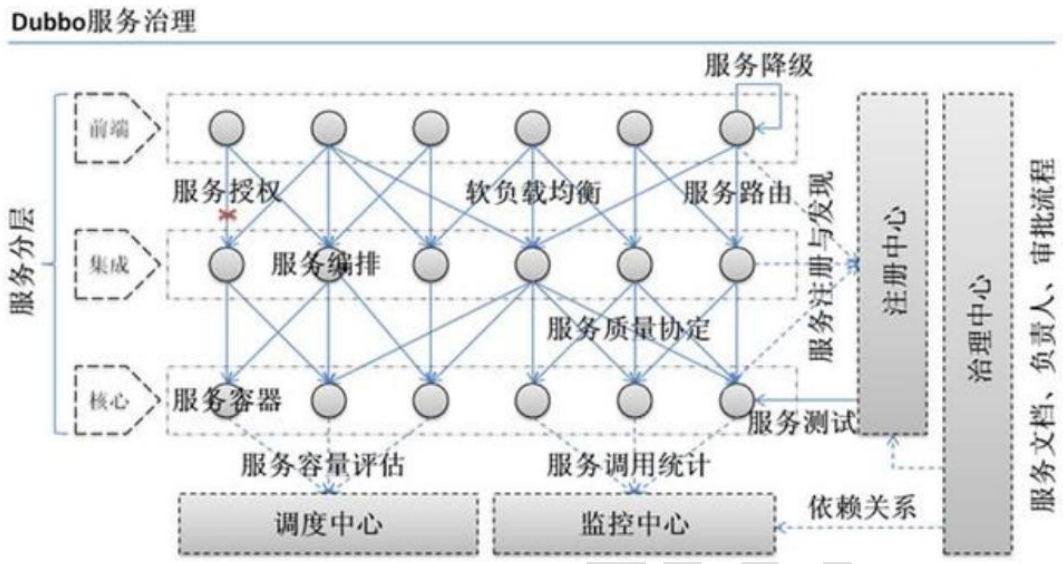
我们除了要关心 RPC 的过程实现，还需要考虑：
1.服务方是集群时，如何挑选一台机器来响应客户端？
2.因网络抖动引起的调用失败，如何重试来弥补？
3.服务方机器的动态增减，如何能够让客户端及时了解到并做出调整？
.....

Dubbo 的使命，即是解决上述围绕 RPC 过程的一览子问题

2 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。

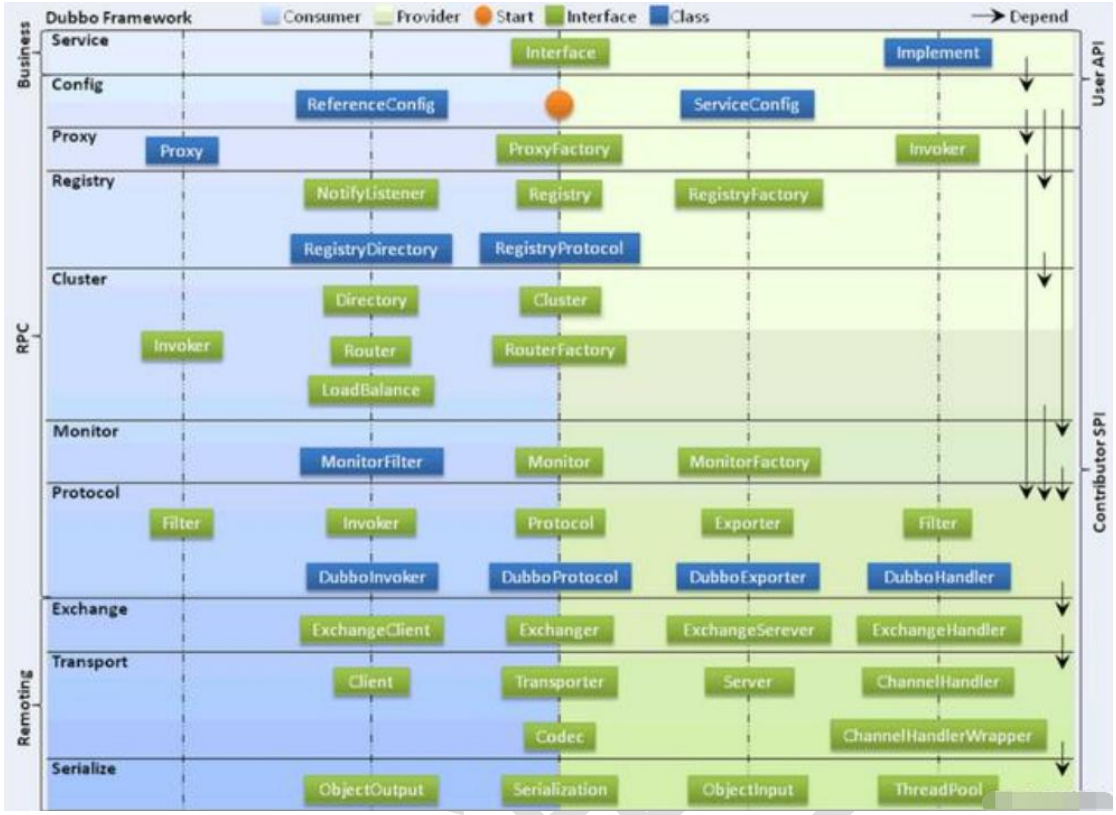


简单的说，Dubbo 就是个服务调用的框架，如果没有分布式的需求，其实是不需要用的，只有在分布式的时候，才有使用 Dubbo 这样的分布式服务框架的需求，并且本质上是个服务调用的东东。
其核心部分包含:

- 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。

2.1 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，

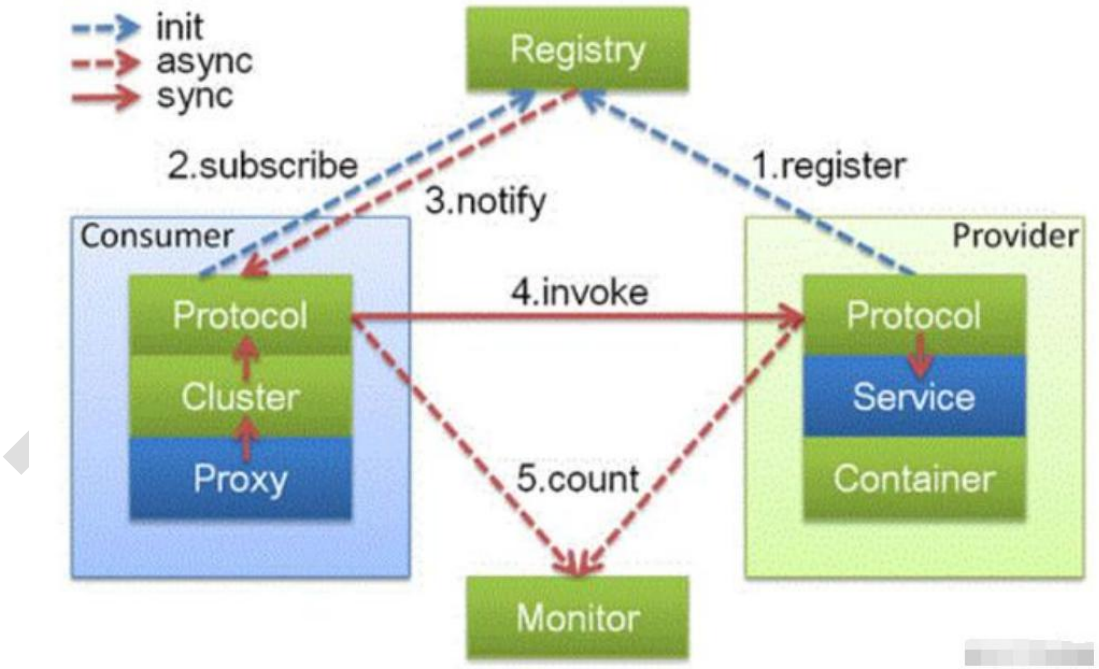
它代表一个可执行体，可向它发起 invoke 调用。它有可能是一个本地的实现，也可能是一个远程的实现，也可能是一个集群实现。

- 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)。

2.2Dubbo 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:

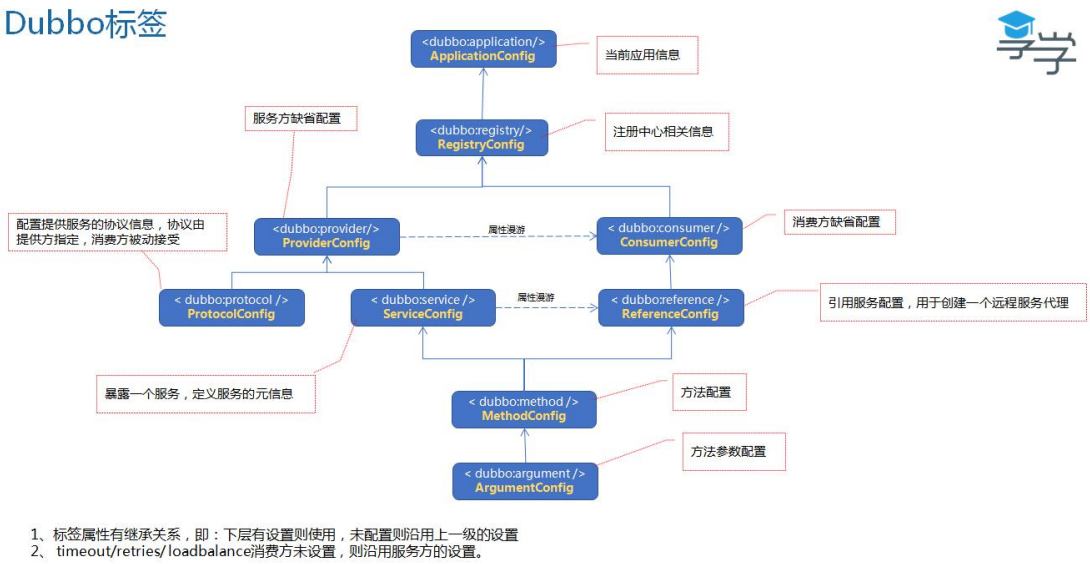
节点	角色说明
Provider	暴露服务的服务提供方
Consumer	调用远程服务的服务消费方
Registry	服务注册与发现的注册中心
Monitor	统计服务的调用次数和调用时间的监控中心
Container	服务运行容器

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。

3 Dubbo 的基础配置使用

3.1 xml 配置方式

3.1.1 dubbo 功能标签集



1、标签属性有继承关系，即：下层有设置则使用，未配置则沿用上一级的设置
2、 timeout/retries/loadbalance消费方未设置，则沿用服务方的设置。

<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

3.1.2 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	2.0.0 以上版本

							对象，构造函数如：public XxxServiceLocal(XxxService xxxService)	
<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	2.0.0 以上版本

							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)	
<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	io 线程池大小(固定大小)	2.0.5 以上版本

						优		
<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	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	2.0.5 以上版本

						优	机器，而不是排队，除非有特殊需求。	
<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	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	1.0.16 以上版本

						现	<dubbo:registry>标签	
<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	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

						理		
<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		服	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

				选		务治理		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	性能	协议的服务器端实现类型，比如：dubbo 协议的 mina,netty 等，http 协议的	2.0.0 以上

					议缺省为 servlet	调优	jetty,servlet 等	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	0	性	延迟注册服务时间(毫秒)- ，设为-1 时，	2.0.5

				选		能调优	表示延迟到 Spring 容器初始化完成时暴露服务	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	序列化编码	2.0.5 以上版本

						优		
<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，	2.0.5 以上版本

							可将值设为 N/A	
<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>	性能	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

					的 async	调优		
<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

3.1.3 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

3.2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**

3.2.1 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

3.2.2 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;
    }
}
```

3.2.3 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

4 Dubbo 高级特性

4.1 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

```
# centers in dubbo2.7
admin.registry.address=zookeeper://127.0.0.1:2181
admin.config-center=zookeeper://127.0.0.1:2181
admin.metadata-report.address=zookeeper://127.0.0.1:2181

admin.root.user.name=root
admin.root.user.password=root
```

- 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
```

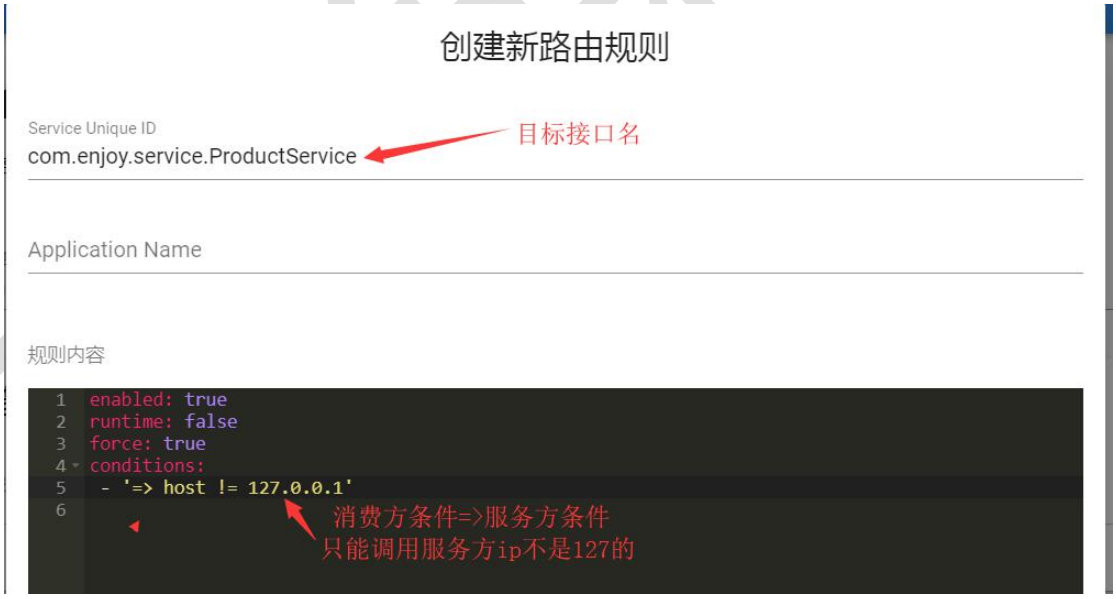
- 访问：

http://localhost:8080

- 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

4.2 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" />
```

4.3Dubbo 超时重连

4.3.1 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"/>
```

4.3.2 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

4.4 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
```

4.5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
```

4.6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。

4.7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

4.8 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"/>
```

4.9 只订阅/只注册

4.9.1 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"/>
```

4.9.2 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）

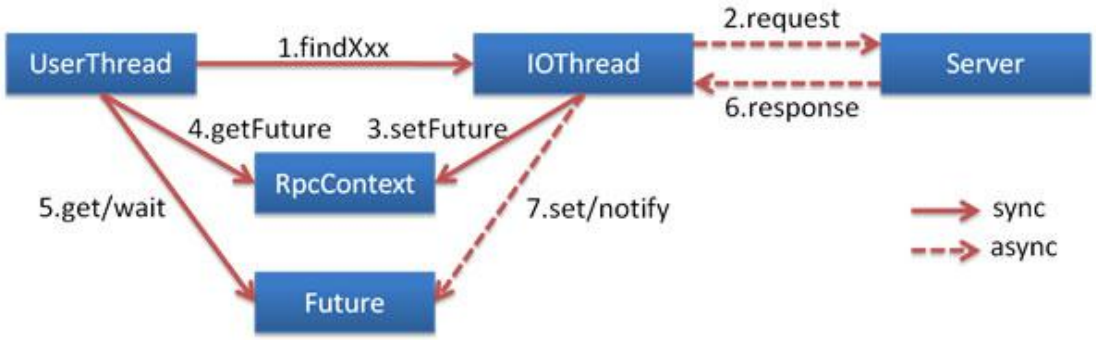
服务提供者配置禁止订阅 subscribe="false"

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

4.10 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" />

4.11 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

4.12 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>
```

4.13 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。

5 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

5.1 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

5.1.1 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！";

    }

}
```

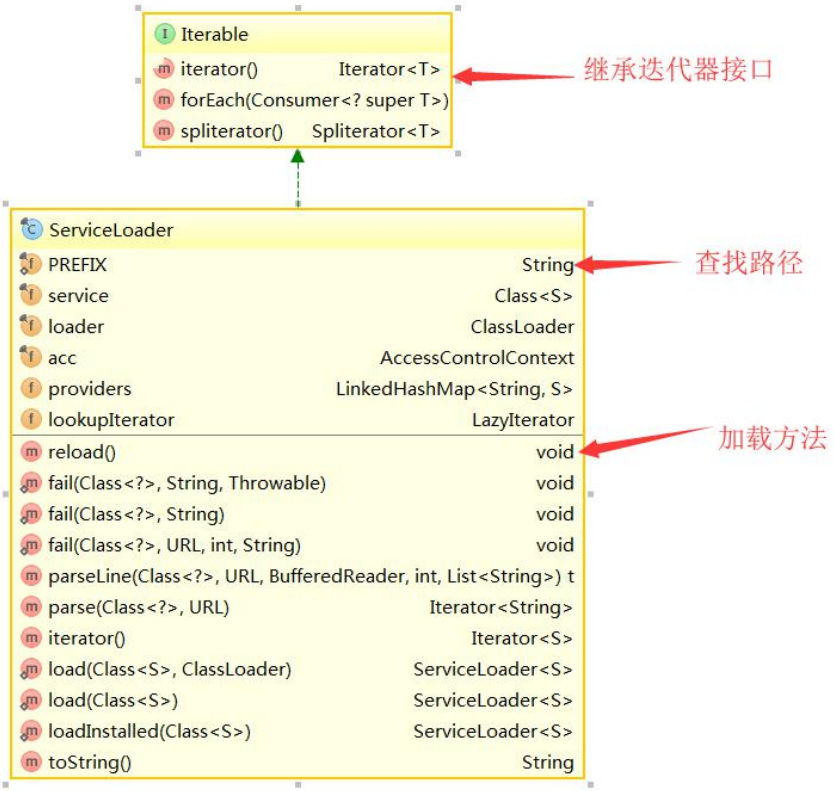
...等等实现

测试

至此，整个 java SPI 的机制使用介绍完毕。

5.1.2 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：



5.2Dubbo 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

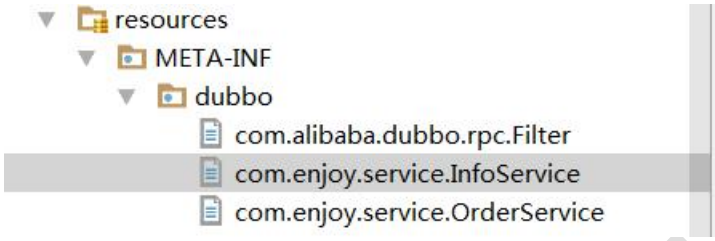
Dubbo 基于自己的需要，增强了这套 SPI 机制，下面介绍 Dubbo 中的 SPI 用法。

5.2.1 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：

```
a=com.enjoy.service.impl.InfoServiceAImpl
b=com.enjoy.service.impl.InfoServiceBImpl
c=com.enjoy.service.impl.InfoServiceCImpl
```

测试功能，指定加载某个实现

```
@Test
public void sayHello() throws Exception {
    ExtensionLoader<InfoService> extensionLoader =
            ExtensionLoader.getExtensionLoader(InfoService.class);
    InfoService a = extensionLoader.getExtension("a");
    a.sayHello("peter");

    InfoService b = extensionLoader.getExtension("b");
    b.sayHello("peter");
}
```

测试结果

5.2.2 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 testActivatel 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

5.2.1 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());
```

5.2.2 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：

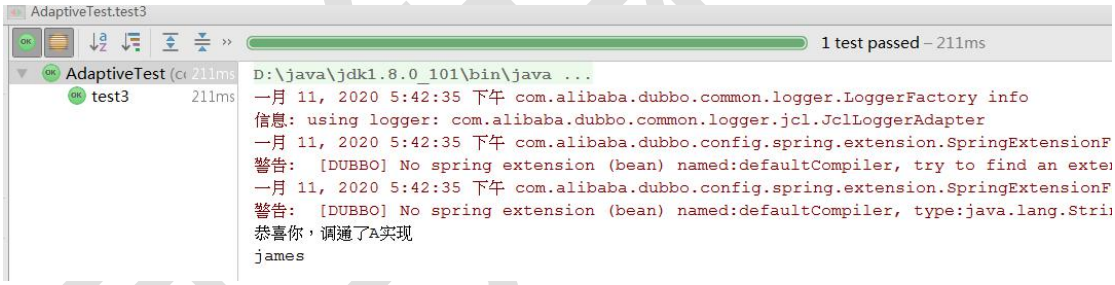
```
public void test3(){
    ExtensionLoader<InfoService> loader =
ExtensionLoader.getExtensionLoader(InfoService.class);
    InfoService adaptiveExtension = loader.getAdaptiveExtension();
    URL url = URL.valueOf("test://localhost/test?info.service=a");
    System.out.println(adaptiveExtension.passInfo("james",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

```
a=com.enjoy.service.impl.FilterA
b=com.enjoy.service.impl.FilterB
c=com.enjoy.service.impl.FilterC
d=com.enjoy.service.impl.FilterD
e=com.enjoy.service.impl.FilterE
```

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

5.2.3 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);
}
```

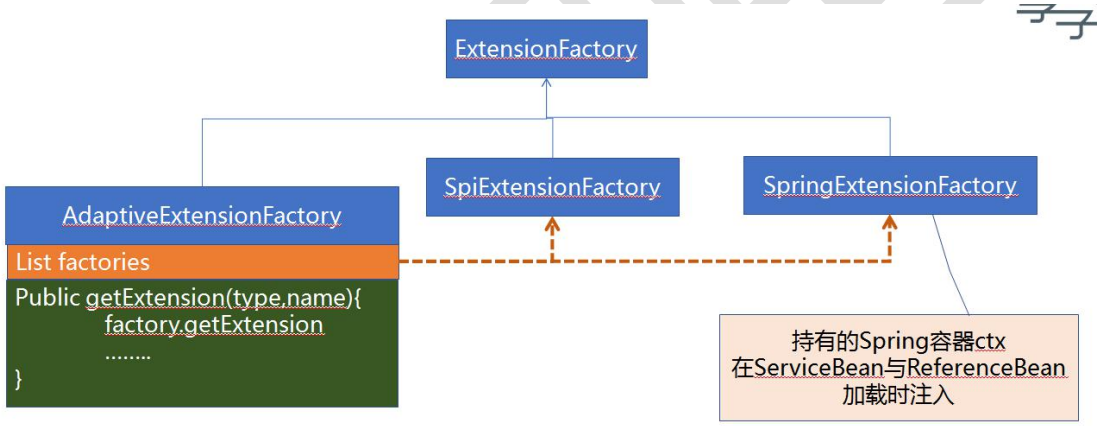
结果如下：

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：

```
@SPI
public interface ExtensionFactory {

    /**
     * Get extension.
     *
     * @param type object type.
     * @param name object name.
     * @return object instance.
     */
    <T> T getExtension(Class<T> type, String name);
```

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）

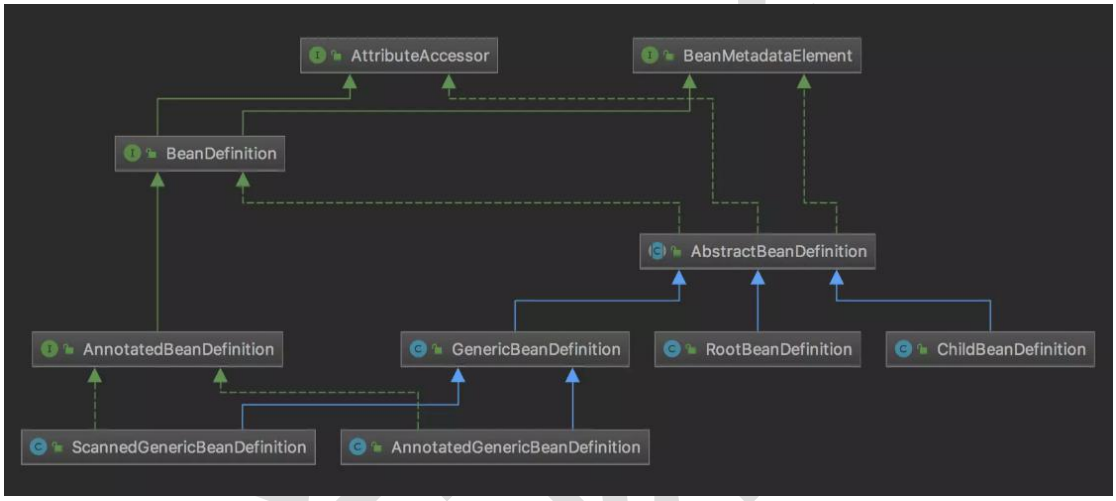
AdaptiveExtensionFactory 在内部持有了所有的 factory 实现工厂，即后两个实现类。一个为 SPI 工厂（依赖类是扩展接口时发挥作用），一个为 Spring 工厂（依赖的是 springbean 时发挥作用）。于是，当我们需要为某个生成的对象注入依赖时，直接调用此对象即可。

至此，整套 DubboSPI 的 IOC 功能圆满了。

6 Dubbo 启动过程探析

6.1Spring 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

6.2Dubbo 的配置解析过程

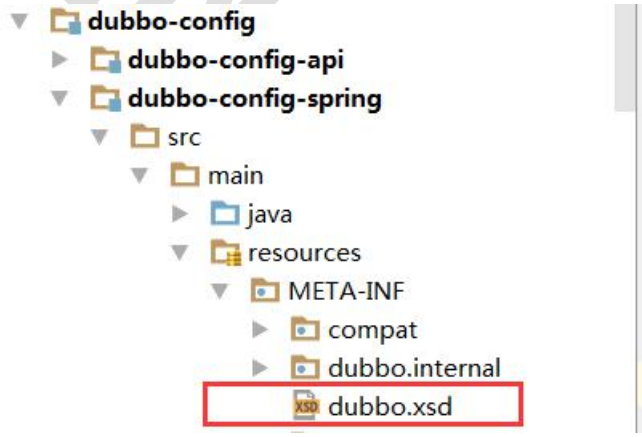
dubbo 的配置解析，不论是 xml 方式的配置，还是注解的配置，目标都是把配置的属性值提取出来，变成 dubbo 的组件 bean（先由 `BeanDefinition` 描述，然后委托 `spring` 生成组件 bean）。

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

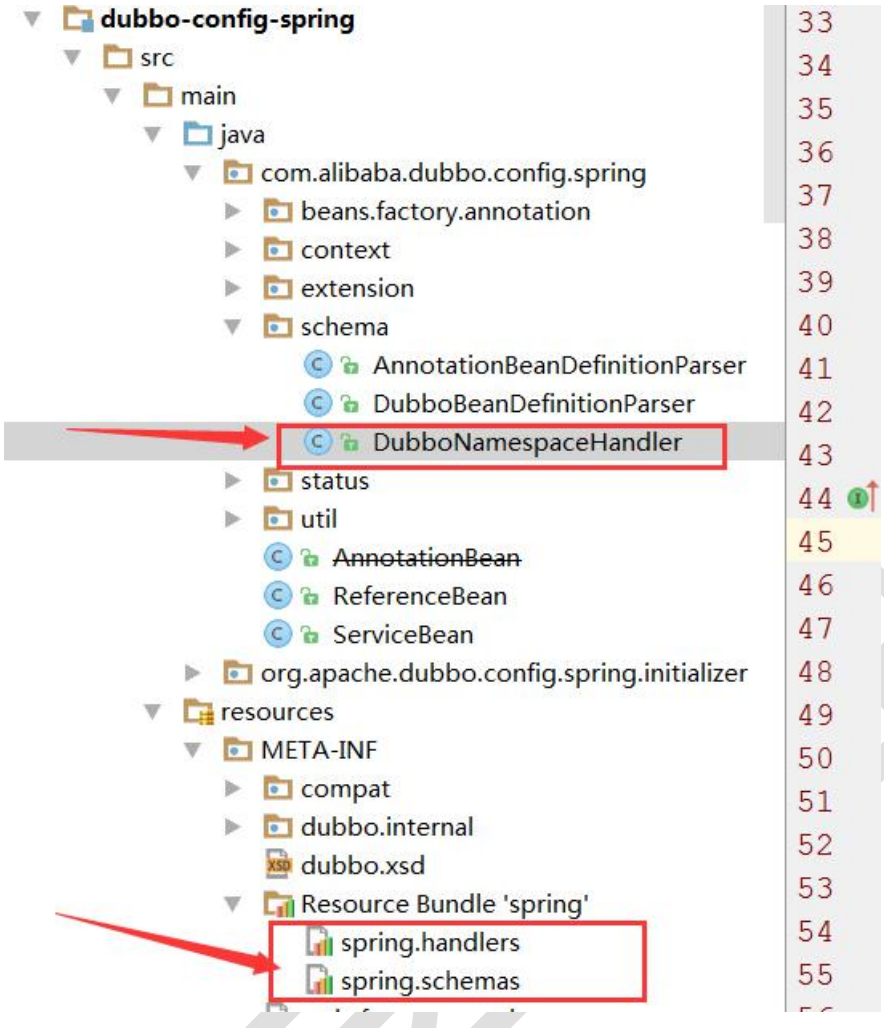
标签名称	类名
dubbo:application	ApplicationConfig
dubbo:module	ModuleConfig
dubbo:registry	RegistryConfig
dubbo:monitor	MonitorConfig
dubbo:provider	ProviderConfig
dubbo:consumer	ConsumerConfig
dubbo:protocol	ProtocolConfig
dubbo:service	ServiceBean
dubbo:reference	ReferenceBean

6.2.1 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。

6.2.2 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。

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
@EnableDubboConfig
@DubboComponentScan
public @interface EnableDubbo {
```

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

6.2.2.1@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：

```
@EnableDubboConfigBindings({
        @EnableDubboConfigBinding(prefix = "dubbo.application", type = ApplicationConfig.class),
        @EnableDubboConfigBinding(prefix = "dubbo.module", type = ModuleConfig.class),
        @EnableDubboConfigBinding(prefix = "dubbo.registry", type = RegistryConfig.class),
        @EnableDubboConfigBinding(prefix = "dubbo.protocol", type = ProtocolConfig.class),
        @EnableDubboConfigBinding(prefix = "dubbo.monitor", type = MonitorConfig.class),
        @EnableDubboConfigBinding(prefix = "dubbo.provider", type = ProviderConfig.class),
        @EnableDubboConfigBinding(prefix = "dubbo.consumer", type = ConsumerConfig.class)
})
public static class Single {
```

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。

6.2.2.2@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。

6.2.2.2.1 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

```
//service的类也在spring中实例化
scanner.addIncludeFilter(new AnnotationTypeFilter(Service.class));

for (String packageToScan : packagesToScan) {

    // Registers @Service Bean first
    scanner.scan(packageToScan);
```

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

```
BeanDefinitionBuilder builder = rootBeanDefinition(ServiceBean.class);

AbstractBeanDefinition beanDefinition = builder.getBeanDefinition();        创建servicebean描述

MutablePropertyValues propertyValues = beanDefinition.getPropertyValues();
                                                                            父类组件信息，已在前一步组装完成
String[] ignoreAttributeNames = of("provider", "monitor", "application", "module", "registry", "protocol", "interface");

propertyValues.addPropertyValues(new AnnotationPropertyValuesAdapter(service, environment, ignoreAttributeNames));

// References "ref" property to annotated-@Service Bean
addPropertyReference(builder, "ref", annotatedServiceBeanName);
// Set interface
```

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

6.2.2.2.2 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）

```
public PropertyValues postProcessPropertyValues(PropertyValues pvs, PropertyDescriptor[] pds, Object bean, String beanName) throws BeanCreati
    InjectionMetadata metadata = this.findInjectionMetadata(beanName, bean.getClass(), pvs);

    try {
        metadata.inject(bean, beanName, pvs);
        return pvs;
    } catch (BeanCreationException var7) {
        throw var7;
    } catch (Throwable var8) {
        throw new BeanCreationException(beanName, "Injection of @" + this.getAnnotationType().getName() + " dependencies is failed", var8);
    }
}
```

找出reference标注的所有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

6.3Dubbo 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。

```
@Override
public void export() {
    super.export();
    // Publish ServiceBeanExportedEvent
    publishExportEvent();
}
```

父类中最终执行此动作

```
Invoker<?> invoker = proxyFactory.getInvoker(ref, (Class) interfaceClass, url);
DelegateProviderMetaDataInvoker wrapperInvoker = new DelegateProviderMetaDataInvoker(invoker, this);

Exporter<?> exporter = protocol.export(wrapperInvoker);
exporters.add(exporter);
```

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

```
public interface Invoker<T> extends Node {

    /**
     * get service interface.
     *
     * @return service interface.
     */
    Class<T> getInterface();

    /**
     * invoke.
     *
     * @param invocation
     * @return result
     * @throws RpcException
     */
    Result invoke(Invocation invocation) throws RpcException;

}
```

实现此动作，最终将动作传递到目标对象ref上

- 协议使用方，希望如此调用 invoker：

```java
@Override
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    //创建spring rmi服务
    final RmiServiceExporter rmiServiceExporter = new RmiServiceExporter();
    rmiServiceExporter.setRegistryPort(invoker.getUrl().getPort());
    rmiServiceExporter.setServiceName(invoker.getUrl().getPath());
    rmiServiceExporter.setServiceInterface(invoker.getInterface());

    //此时目标服务没有，需要通过invoker调通，使用代理
    T service = (T)Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(),
            new Class[]{invoker.getInterface()},
            new InvokerInvocationHandler(invoker));

    rmiServiceExporter.setService(service);//DemoService service,就不能多态了，直接这入就ok
    try {
        rmiServiceExporter.afterPropertiesSet();
    } catch (RemoteException e) {
        throw new RpcException(e.getMessage(), e);
    }
    return null;
}
```

2. 我们转而希望通过invoker对象，制作一个target服务代理来使用

1. protocol协议的目标，都是要将调用转到目标target服务上，但是当前环境并没有这个目标target

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

```java
Invoker<DemoService> invoker = new SimpleInvoker(service,DemoService.class,url);
Protocol protocol = new RmiProtocol();
//暴露对象
protocol.export(invoker);
System.out.println("Dubbo server 启动");
// 保证服务一直开着
System.in.read();
```

目标对象传入

协议只管调用

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

```java
private  T target;
private Class<T> type;
private URL url;

public SimpleInvoker(T service, Class<T> type, URL url){
    this.target = service;
    this.type = type;
    this.url = url;
}

@Override
public Class<T> getInterface() { return type; }

@Override
public Result invoke(Invocation invocation) throws RpcException {
    Method method = null;
    try {
        method = DemoService.class.getMethod(invocation.getMethodName(), invocation.getParameterTypes());
        return new RpcResult(method.invoke(target, invocation.getArguments()));
    } catch (NoSuchMethodException e) {
        e.printStackTrace();
    } catch (InvocationTargetException e) {
        e.printStackTrace();
    } catch (IllegalAccessException e) {
        e.printStackTrace();
    }
    return null;
}
```

反射调用目标target

➢ 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？）

6.4Dubbo 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

```
public synchronized T get() {
    if (destroyed) {
        throw new IllegalStateException("Already destroyed!");
    }
    if (ref == null) {
        init();
    }
    return ref;
}
```

初始化一个

ref 是通过 interface 和 url 信息生成的代理

```
invoker = refprotocol.refer(interfaceClass, urls.get(0));
```

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：

```
rmiProxyFactoryBean.setServiceUrl(url.toIdentityString());
rmiProxyFactoryBean.setServiceInterface(type);
rmiProxyFactoryBean.setCacheStub(true);
rmiProxyFactoryBean.setLookupStubOnStartup(true);
rmiProxyFactoryBean.setRefreshStubOnConnectFailure(true);
rmiProxyFactoryBean.afterPropertiesSet();
return (T) rmiProxyFactoryBean.getObject();
```

对象类型

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

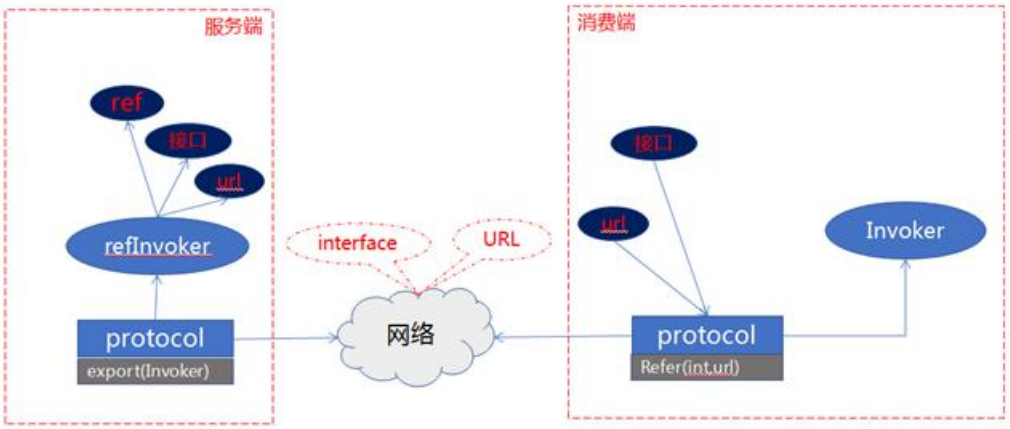
2、在业务操作端，为了方便操作，再做一个代理对象，来转请求到 invoker 上

```
//消费invoker，负责发送协议调用信息
Invoker<DemoService> invoker = protocol.refer(DemoService.class, url);

//做一个动态代理，将调用目标指向invoker即可
DemoService service = (DemoService)Proxy.newProxyInstance(Thread.currentThread().getContextClassLoader(),
        new Class[]{invoker.getInterface()},
        new InvokerInvocationHandler(invoker));//反射逻辑
```

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

6.5rpc 过程概述

我们再归纳一个服务暴露与服务引入的过程，如下图



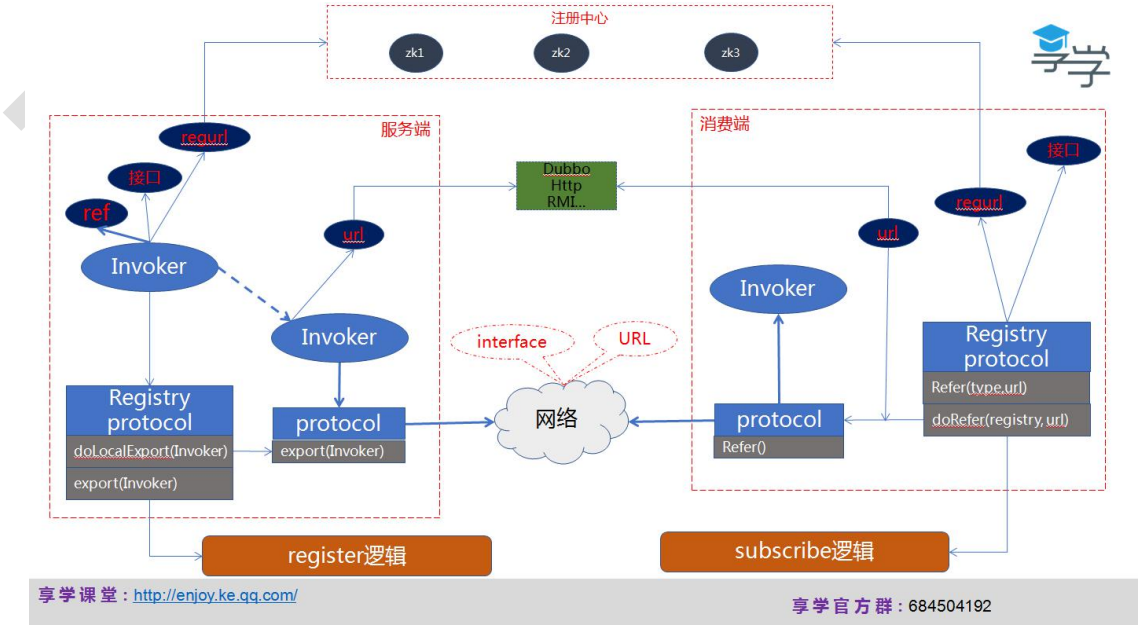
1、网络数据的传输的过程，由 protocol 组件负责。无论你是什么协议实现，主要目标，就是将消费端的调用信息（interface 接口描述），传递到服务端（java 反射使用）。
2、服务端使用 protocol 来监听网络，当数据信息到来时，需要按信息指示（interface 描述），调用对应的 service 服务来响应。
3、消费端发起远程调用。它需要将本地代理对象动作，转换成调用信息（interface 描述），通过 protocol 发送到网络上。
4、整个过程自始至终涉及到的信息有两种，一种为 url 信息（协议头://ip:port/path）,一种 interface 参数。为了简化三个部分的协作，Dubbo 提出一个实体域对象 invoker（内部封装了 url 和 interface）
5、invoker 的思想：万事万物只有一个调用入口，invoker 的 invoke 方法。因此，服务端 protocol 对本地 service 的调用，被封装成了 invoker;消费端发起远程调用到网络的动作，也被封装成了 invoker。
6、从此，服务端监听到的网络请求，将自动触发服务端绑定 invoker.invoke,消费端直接调用消费端的 invoker.invoke 也自动将信息发送到了 protocol 网络上。

6.6Dubbo 的服务注册与发现机制

在上面的 dubbo 服务暴露和引入过程中，核心动作是两个，通过 protocol.export(serviceInvoker)将服务暴露出去，通过 protocol.refer(DemoService.**class**, **registryUrl**)将网络上的服务引入进来。整个 rpc 链条已成闭环。那么 dubbo 的动态服务发布，注册中心又是如何集成上去的呢？

按照开闭原则，dubbo 希望只增加代码来完成功能的增强，杜绝更改代码的，如何做呢？我们前面的 SPI 思想此时就展示出它的威力了

我们的 protocol 对象，本身只是个 SPI 适配对象，它可以根据 URL 的不同，选择不同的 protocol 实现类。于是，dubbo 在此基础上，进行一层嵌套：将服务的注册/发现功能，抽象成一个 protocol 协议，即 RegistryProtocol（并不是个真的协议实现，是来做服务发现注册使用），并在 RegistryProtocol 对象内，再次嵌套一个真实的 protocol，整个结构如下图：



6.6.1 服务暴露时注册服务

ServiceConfig.doExportUrls 执行服务 export

而 doExportUrls 方法分成两部分

1、获取所有注册中心的 URL

2、遍历所有协议 ProtocolConfig，将每个 protocol 发布到所有注册中心上

```
private void doExportUrls() {
          // 1.获取注册中心 URL
   List<URL> registryURLs = loadRegistries(true);
   // 2.遍历所有协议，export protocol 并注册到所有注册中心
   for (ProtocolConfig protocolConfig : protocols) {
      doExportUrlsFor1Protocol(protocolConfig, registryURLs);
   }
      }
```

其中，loadRegistries 获取配置的注册中心 URL
首先执行 checkRegistry，判断是否有配置注册中心，如果没有，则从默认配置文件 dubbo.properties 中读取 dubbo.registry.address 组装成 RegistryConfig。

```
      AbstractInterfaceConfig.checkRegistry()
if (registries == null || registries.isEmpty()) {
   String address = ConfigUtils.getProperty("dubbo.registry.address");
   if (address != null && address.length() > 0) {
      registries = new ArrayList<RegistryConfig>();
      String[] as = address.split("\\s*[|]+\\s*");
      for (String a : as) {
         RegistryConfig registryConfig = new RegistryConfig();
         registryConfig.setAddress(a);
         registries.add(registryConfig);
      }
   }
}
```

然后根据 RegistryConfig 的配置，组装 registryURL，形成的 URL 格式如下

```
registry://127.0.0.1:2181/com.alibaba.dubbo.registry.RegistryService?application=demo-provid
er&registry=zookeeper
```

这个 URL 表示它是一个 registry 协议(RegistryProtocol)，地址是注册中心的 ip:port，服务接口是 RegistryService，registry 的类型为 zookeeper。

doExportUrlsFor1Protocol 发布服务和注册因为 dubbo 支持多协议配置，对于每个 ProtocolConfig 配置，组装 protocolURL，注册到每个注册中心上。

首先根据 ProtocolConfig 构建协议的 URL

1. 设置 side=provider,dubbo={dubboVersion},timestamp=时间戳,pid=进程 id
2. 从 application，module，provider，protocol 配置中添加 URL 的 parameter
3. 遍历 dubbo:service 下的 dubbo:method 及 dubbo:argument 添加 URL 的 parameter
4. 判断是否泛型暴露，设置 generic，及 methods=*，否则获取服务接口的所有 method
5. 获取 host 及 port，host 及 port 都是通过多种方式获取，保证最终不为空

```
// 获取绑定的 ip，1 从系统配置获取 2 从 protocolConfig 获取 3 从 providerConfig 获取
// 4 获取 localhost 对应得 ipv45 连接注册中心获取 6 直接获取 localhost 对应的 ip(127.0.0.1)
String host = this.findConfigedHosts(protocolConfig, registryURLs, map);
// 获取绑定的 port，1 从系统配置 2 从 protocolConfig3 从 providerConfig4 从 defaultPort 之上随机取可用的
Integer port = this.findConfigedPorts(protocolConfig, name, map);
```

最终构建 URL 对象

```
// 创建 protocol export url
URL url = new URL(name, host, port, (contextPath == null || contextPath.length() == 0 ? "" :
contextPath + "/") + path, map);
```

构建出的 protocolURL 格式如下

```
dubbo://192.168.199.180:20880/com.alibaba.dubbo.demo.DemoService?anyhost=true&application=de
mo-provider&bind.ip=192.168.199.180&bind.port=20880&dubbo=2.0.0&generic=false&interface=com.
alibaba.dubbo.demo.DemoService&methods=sayHello&pid=5744&qos.port=22222&side=provider&timest
amp=1530746052546
```

这个 URL 表示它是一个 dubbo 协议(DubboProtocol)，地址是当前服务器的 ip，端口是要暴露的服务的端口号，可以从 dubbo:protocol 配置，服务接口为 dubbo:service 配置发布的接口。

遍历所有的 registryURL，执行以下操作:

1. 给 registryURL 设置 EXPORT_KEY 为上面构建的 protocolURL
2. 根据实现对象，服务接口 Class 和 registryuRL 通过 ProxyFactory 获取代理 Invoker(继承于 AbstractProxyInvoker)
3. 将 Invoker 对象和 ServiceConfig 组装成 MetaDataInvoker，通过 protocol.export(invoker)暴露出去。

```
for (URL registryURL : registryURLs) {
    url = url.addParameterIfAbsent("dynamic", registryURL.getParameter("dynamic"));
    // 组装监控 URL
    URL monitorUrl = loadMonitor(registryURL);
    if (monitorUrl != null) {
        url = url.addParameterAndEncoded(Constants.MONITOR_KEY, monitorUrl.toFullString());
    }
    // 以 registryUrl 创建 Invoker
    Invoker<?> invoker = proxyFactory.getInvoker(ref, (Class) interfaceClass,
registryURL.addParameterAndEncoded(Constants.EXPORT_KEY, url.toFullString()));
```

```
    // 包装 Invoker 和 ServiceConfig
    DelegateProviderMetaDataInvoker wrapperInvoker = new
DelegateProviderMetaDataInvoker(invoker, this);

    // 以 RegistryProtocol 为主，注册和订阅注册中心，并暴露本地服务端口
    Exporter<?> exporter = protocol.export(wrapperInvoker);
    exporters.add(exporter);
}
```

这里的 protocol 是一个适配代理对象，根据 SPI 机制，这里的 procotol.export 执行时，会根据 Invoker 的 URL 的 protocol 来选择具体的实现类，此处 URL 的协议头为 registry,因此方法会交由 RegistryProtocol 处理 export 过程。

RegistryProtocol.export 暴露服务，doLocalExport 内执行服务的暴露逻辑，后续执行注册中心信息注册逻辑

```
public <T> Exporter<T> export(final Invoker<T> originInvoker) throws RpcException {
    // 发布本地 invoker，暴露本地服务，打开服务器端口
    final ExporterChangeableWrapper<T> exporter = doLocalExport(originInvoker);
    // 根据 url 从 registryFactory 中获取对应的 registry
    final Registry registry = getRegistry(originInvoker);
    //......(省略部分代码)
    if (register) {
            // 向注册中心注册 providerUrl
        register(registryUrl, registedProviderUrl);
        // 本地注册表设置此 provider 注册完成
        ProviderConsumerRegTable.getProviderWrapper(originInvoker).setReg(true);
    }
    //......(省略部分代码)
    // 向注册中心订阅提供者 url 节点
    registry.subscribe(overrideSubscribeUrl, overrideSubscribeListener);
    //......(省略部分代码)
}
```

从 Invoker 中获取 providerURL，同传入的 Invoker 对象组装成 InvokerDelegete，通过 protocol.export 根据 providerURL(还是 SPI 自适配逻辑)暴露服务，打开服务器端口，获得 Exporter 缓存到本地。

```
private <T> ExporterChangeableWrapper<T> doLocalExport(final Invoker<T> originInvoker) {
            // 获取 cache key
    String key = getCacheKey(originInvoker);
    // 是否存在已绑定的 exporter
    ExporterChangeableWrapper<T> exporter = (ExporterChangeableWrapper<T>) bounds.get(key);
    if (exporter == null) {
        synchronized (bounds) {
            exporter = (ExporterChangeableWrapper<T>) bounds.get(key);
            if (exporter == null) {
            // 封装 invoker 和 providerUrl
```

```
            final Invoker<?> invokerDelegete = new InvokerDelegete<T>(originInvoker,
getProviderUrl(originInvoker));
            // export provider invoker，Protocol 的具体实现是由 Url 中的 protocol 属性决定的
            // 封装创建出的 exporter 和 origin invoker
            exporter = new ExporterChangeableWrapper<T>((Exporter<T>)
         protocol.export(invokerDelegete), originInvoker);
            bounds.put(key, exporter);
         }
      }
   }
   return exporter;
}
```

6.6.2 服务引入时订阅服务地址

ReferenceConfig.init()方法，最后会来创建代理对象

ref = createProxy(map);

```
private T createProxy(Map map) {
   //......(省略部分代码)
if (urls.size() == 1) {
   invoker = refprotocol.refer(interfaceClass, urls.get(0));
}
//......(省略部分代码)
   // 创建服务代理
   return (T) proxyFactory.getProxy(invoker);
      }
```

看这一段代码，前面部分，主要用来找到并校验配置的 urls，此 url 一般只是一个注册中心的 url。值类似下面这样：

```
registry://127.0.0.1:2181/com.alibaba.dubbo.registry.RegistryService?registry=zookeeper
```

所以当 refprotocol.refer 调用时，直接还是适配到 RegistryProtocol。

RegistryProtocol 的 refer 动作，会转发到 doRefer 方法：

```
private <T> Invoker<T> doRefer(Cluster cluster, Registry registry, Class<T> type, URL url) {
   RegistryDirectory<T> directory = new RegistryDirectory<T>(type, url);
   directory.setRegistry(registry);
   directory.setProtocol(protocol);
   // all attributes of REFER_KEY
   Map<String, String> parameters = new HashMap<String,
String>(directory.getUrl().getParameters());
   // 初始化订阅 URL
   URL subscribeUrl = new URL(Constants.CONSUMER_PROTOCOL,
```

```
parameters.remove(Constants.REGISTER_IP_KEY), 0, type.getName(), parameters);
    if (!Constants.ANY_VALUE.equals(url.getServiceInterface())
            && url.getParameter(Constants.REGISTER_KEY, true)) {
        registry.register(subscribeUrl.addParameters(Constants.CATEGORY_KEY,
Constants.CONSUMERS_CATEGORY,
                Constants.CHECK_KEY, String.valueOf(false)));
    }
    // 注册监听节点，即向注册中心订阅服务
    directory.subscribe(subscribeUrl.addParameter(Constants.CATEGORY_KEY,
            Constants.PROVIDERS_CATEGORY
                    + "," + Constants.CONFIGURATORS_CATEGORY
                    + "," + Constants.ROUTERS_CATEGORY));

    // 包装一个 invoker 集群返回
    Invoker invoker = cluster.join(directory);
    ProviderConsumerRegTable.registerConsumer(invoker, url, subscribeUrl, directory);
    return invoker;
}
```

这一段代码逻辑非常清晰，就是组装注册中心 URL，订阅服务

6.6.3 注册/订阅逻辑

dubbo 的注册/订阅动作，主要涉及以下接口：

- org.apache.dubbo.registry.Registry

其中 Registry 继承自 RegistryService，负责注册/订阅动作：

```
public interface RegistryService { // Registry extends RegistryService
    /**
     * 注册服务.
     * @param url 注册信息，不允许为空，如：
dubbo://10.20.153.10/com.alibaba.foo.BarService?version=1.0.0&application=kylin
     */
    void register(URL url);

    /**
     * 取消注册服务.
     * @param url 注册信息，不允许为空，如：
dubbo://10.20.153.10/com.alibaba.foo.BarService?version=1.0.0&application=kylin
     */
    void unregister(URL url);

    /**
     * 订阅服务.     *
```

```
    * @param url 订阅条件，不允许为空，如：
consumer://10.20.153.10/com.alibaba.foo.BarService?version=1.0.0&application=kylin
    * @param listener 变更事件监听器，不允许为空
    */
   void subscribe(URL url, NotifyListener listener);

   /**
    * 取消订阅服务.
    *  @param url 订阅条件，不允许为空，如：
consumer://10.20.153.10/com.alibaba.foo.BarService?version=1.0.0&application=kylin
    * @param listener 变更事件监听器，不允许为空
    */
   void unsubscribe(URL url, NotifyListener listener);

   /**
    * 查询注册列表，与订阅的推模式相对应，这里为拉模式，只返回一次结果。
    *  @param url 查询条件，不允许为空，如：
consumer://10.20.153.10/com.alibaba.foo.BarService?version=1.0.0&application=kylin
    * @return 已注册信息列表，可能为空，
    */
   List<URL> lookup(URL url);

}
```

订阅方法需要一个监听器参数：

NotifyListener.java：

```
public interface NotifyListener {
   /**
    * 当收到服务变更通知时触发。
    *  @param urls 已注册信息列表，总不为空
    */
   void notify(List<URL> urls);
}
```

方法动作很好理解，register 方法就是将 url 写入注册中心，subscribe 则将监听器注册到 url 上，当服务 url 有变化时，则触发 notify 方法。其每个服务最终注册的信息结构，示例如下（以消费端在 zookeeper 为例）

当某个服务发生变动，notify 触发回来的 urls 信息也同样包含这些信息

当然，Dubbo 此处定义的 `Registry` 服务，是个扩展点，有很多实现，可

心通过 SPI 机制适配实现类。

- 扩展点接口：org.apache.dubbo.registry.RegistryFactory

此处的扩展，与这个配置相关联，SPI 的机制不再展开细说

```
<!-- 定义注册中心 -->
<dubbo:registry id="xxx1" address="xxx://ip:port" />
<!-- 引用注册中心，如果没有配置 registry 属性，将在 ApplicationContext 中自动扫描 registry 配置 -->
<dubbo:service registry="xxx1" />
<!-- 引用注册中心缺省值，当<dubbo:service>没有配置 registry 属性时，使用此配置 -->
<dubbo:provider registry="xxx1" />
```

有兴趣的同学，可以去看下 Dubbo 的监听实现逻辑类 RegistryDirectory，它的包装了 Dubbo 的发布订阅逻辑并且其本身也是看监听器。监听触发逻辑在 notify 方法中，主要职责便是监听到的 url 信息转化为 invoker 实体，提供给 Dubbo 使用。

为了性能，在 RegistryDirectory 中，可以看到有很多的缓存容器，urlInvokerMap/**methodInvokerMap/**cachedInvokerUrls 等用来缓存服务的信息。也就是说，notify 的作用是更改这些缓存信息，而 Dubbo 在 rpc 过程中，则是直接使用缓存中的信息。

这里要强调一下，在 Dubbo 中，URL 是整个服务发布和调用流程的串联信息，它包含了服务的基本信息（服务名、服务方法、版本、分组），注册中心配置，应用配置等等信息，还包括在 dubbo 的消费端发挥作用的各种组件信息如：filter、loadbalance、cluster 等等。

在消费端 notify 中收到这些 url 信息时，意味着这个组件信息也已经得到了。Dubbo 此时便扩展逻辑，来加入这些组件功能了。

最后，完整描述下服务注册与发现机制：

基于注册 中心的事件通知（订阅与发布），一切支持事件订阅与发布的框架都可以作为 Dubbo 注册中心的选型。

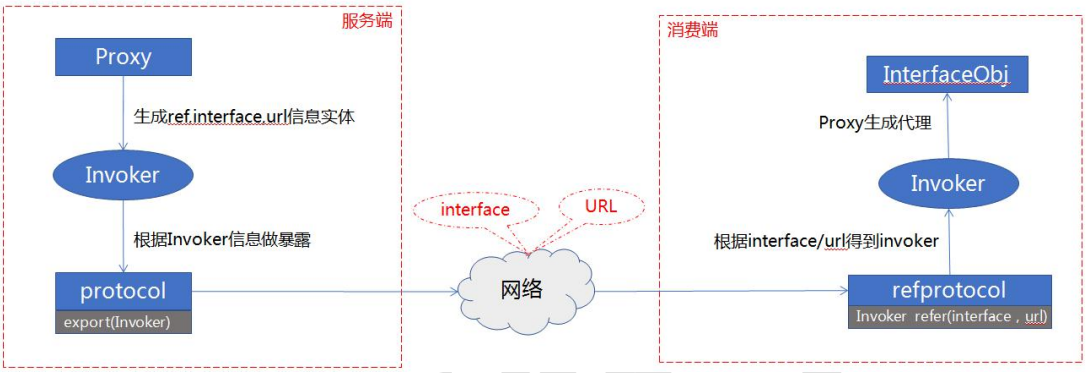
服务提供者在暴露服务时，会向注册中心注册自己，具体就是在${service interface}/providers 目录下添加 一个节点（临时），服务提供者需要与注册中心保持长连接，一旦连接断掉（重试连接）会话信息失效后，注册中心会认为该服务提供者不可用（提供者节点会被删除）。

消费者在启动时，首先也会向注册中心注册自己，具体在${interface interface}/consumers 目录下创建一个节点。

消费者订阅${service interface}/ [providers、configurators、routers]三个目录，这些目录下的节点删除、新增事件都会通知消费者，根据通知，重构服务调用器(Invoker)。

6.7再谈 Invoker

在前面的服务注册与发现中，我们发现，服务在订阅过程中，把 notify 过来的 urls 都转成了 invoker，不知道大家是否还记得前面的 rpc 过程，protocol 也是在服务端和消费端各连接子一个 invoker，如下图：



这张图主要展示 rpc 主流程，消费端想要远程调用时，他是调用 invoker.invoke 方法;服务端收到网络请求时，也是直接触发 invoker.invoke 方法

对的，你没有猜错，Dubbo 设计 invoker 实体的初衷，就是想要统一操作，无论你要做什么方法调用，都请使用 invoker 来包装后，使用 invoker.invoke 来调用这个动作（你内部如何转我不管），简化来看，rpc 过程即是如此：

消费端 invoker.invoke ------>网络---->服务端 invoker.invoke--->ref 服务

上面的链路是个简化的路径，但在实际的 dubbo 调用中，此链条可能会有局部的多层嵌套，如：

消费端 invoker.invoke ------>容错策略--->网络---->服务端 invoker.invoke--->ref 服务

那么此时要重新定义链条吗？那不是个好主意。

Dubbo 的做法是这样，将容错策略也包装成 invoker 对象：

FailfastClusterInvoker.invoke--->protocolInvoker.invoke-->**网络---->服务端 invoker.invoke--->ref 服务**

依次类推，dubbo 内部有非常多的 invoker 包装类，它们层层嵌套，但 rpc 流程不关心细节，只傻瓜式地调用其 invoke 方法，剩下的逻辑自会传递到最后一个 invoker 进行网络调用。

下面我们来研究一下 Cluster、Cluster Invoker、Directory、Router 和 LoadBalance 组件是如何嵌入到 rpc 调用过程中来的

6.7.1 集群容错

在消费端，当 protocol.refer 得到 invoker 对象后，按调用关系，会把 invoker 对象作为目标 target，做一个动态代理生成 service 接口代理。

Dubbo 在这里玩了个心眼。真正的过程走得百绕千回，看这段代码，

RegistryProtocol.doRefer 方法：

```
private <T> Invoker<T> doRefer(Cluster cluster, Registry registry, Class<T> type, URL url) {
    RegistryDirectory<T> directory = new RegistryDirectory<T>(type, url);
    directory.setRegistry(registry);
    directory.setProtocol(protocol);
    //......(省略部分代码)
    // 包装一个 invoker 集群返回
    Invoker invoker = cluster.join(directory);
    //......(省略部分代码)
    return invoker;
}
```

原来的 protocol 被转进了 RegistryDirectory 类中去了，doRefer 返回的 invoker 对象，是 cluster.join(directory)返回的 invoker

而 cluster 是一个扩展接口，因此，这个接口方法最终执行的对象，是根据容错策略自适配出来的对象，如果 url 中不指定则默认是 **failover**

再看 failover 实现类的逻辑，非常简单，只是返回一个 FailoverClusterInvoker 对象

```
public class FailoverCluster implements Cluster {
    public final static String NAME = "failover";
    @Override
    public <T> Invoker<T> join(Directory<T> directory) throws RpcException {
        return new FailoverClusterInvoker<T>(directory);
    }
}
```

再看下具体实现逻辑：

```
public class FailoverClusterInvoker<T> extends AbstractClusterInvoker<T> {
    //......(省略部分代码)
    public Result doInvoke(Invocation invocation, final List<Invoker<T>> invokers, LoadBalance
loadbalance) throws RpcException {
        //......(省略部分代码)
            //容错次数
        int len = getUrl().getMethodParameter(invocation.getMethodName(), Constants.RETRIES_KEY,
Constants.DEFAULT_RETRIES) + 1;
        if (len <= 0) {
            len = 1;
        }
```

```
        //......(省略部分代码)
        for (int i = 0; i < len; i++) {
            //......(省略部分代码)
            Invoker<T> invoker = select(loadbalance, invocation, copyinvokers, invoked);
            //......(省略部分代码)
           Result result = invoker.invoke(invocation);
           return result;
        }
    }
}
```

此 invoker 的逻辑：

1、按重新次数 for 循环，只要不是正常返回，则再试一次

2、调用 select 方法，取得一个 loadbalance 策略的 invoker 对象，然后执行此 invoker 对象得到结果。要注意的是，此 select 方法，是从一组 invoker 中选择一个 invoker 出来

```
Invoker<T> invoker = loadbalance.select(invokers, getUrl(), invocation);
```

3、继续往下追，在 FailoverClusterInvoker 对象的 invoke 方法中（父类）可看到，最终 invokers 来自这一句

```
List<Invoker<T>> invokers = directory.list(invocation);
```

此就是从这一句传入的

```
Invoker invoker = cluster.join(directory);
```

最终追溯到 RegistryDirectory 对象的缓存 methodInvokerMap 上来。如果你有印象，则知道，RegistryDirectory 正是我们上一步，注册/订阅逻辑的核心类，它将订阅得到的 urls 信息缓存在 RegistryDirectory 中。

6.7.2 Dubbo 的过滤器链

Filter（过滤器）在很多框架中都有使用过这个概念，基本上的作用都是类似的，在请求处理前或者处理后做一些通用的逻辑，而且 Filter 可以有多个，支持层层嵌套。

Dubbo 的 Filter 实现入口是在 ProtocolFilterWrapper，因为 ProtocolFilterWrapper 是 Protocol 的包装类，所以会在 SPI 加载的 Extension 的时候被自动包装进来。当然 filter 要发挥作用，必定还是要

在嵌入到 RPC 的调用线中（你马上应该反应过来，嵌入的办法就是包装成 invoker）

ProtocolFilterWrapper 作为包装类，会成为其它 protocol 的修饰加强外层。因此，protocol 的 `export` 和 `refer` 方法，首先是调用 ProtocolFilterWrapper 类的。

暴露服务代码：

```
public <T> Exporter<T> export(Invoker<T> invoker) throws RpcException {
    //注册协议，并不是真的协议，所以用不上 filter
    if (Constants.REGISTRY_PROTOCOL.equals(invoker.getUrl().getProtocol())) {
        return protocol.export(invoker);
    }
    return protocol.export(buildInvokerChain(invoker, Constants.SERVICE_FILTER_KEY,
Constants.PROVIDER));
}
```

引入服务的代码：

```
public <T> Invoker<T> refer(Class<T> type, URL url) throws RpcException {
    if (Constants.REGISTRY_PROTOCOL.equals(url.getProtocol())) {
        return protocol.refer(type, url);
    }
    return buildInvokerChain(protocol.refer(type, url), Constants.REFERENCE_FILTER_KEY,
Constants.CONSUMER);
}
```

可以看到，两者原来的 invoker 对象，都由 buildInvokerChain 做了一层包装

来看一下 filterChain 的逻辑

```
private static <T> Invoker<T> buildInvokerChain(final Invoker<T> invoker, String key, String group
{
    Invoker<T> last = invoker;
    List<Filter> filters =
ExtensionLoader.getExtensionLoader(Filter.class).getActivateExtension(invoker.getUrl(), key,
group);
    if (!filters.isEmpty()) {
        for (int i = filters.size() - 1; i >= 0; i--) {
            final Filter filter = filters.get(i);
            final Invoker<T> next = last;
            last = new Invoker<T>() {
                //......(省略部分代码)
                @Override
                public Result invoke(Invocation invocation) throws RpcException {
                    return filter.invoke(next, invocation);
                }
```

```
            //......(省略部分代码)
        };
      }
    }
    return last;
}
```

逻辑较为简单：

1、所有 filter 包装进 invoker 对象中，invoke 方法直接调对应的 filter.invoke

2、filter 对象首尾相联，前一个 filter.invoke 参数，传入后一个 filter 的 invoker 对象

3、最后一个 filter.invoke 参数中，直接传原始的 invoker 对象

4、filter 的所有获取，按扩展点方式得到

```
ExtensionLoader.getExtensionLoader(Filter.class).getActivateExtension(invoker.getUrl(), key, group);
```

6.8Dubbo 的线程模型

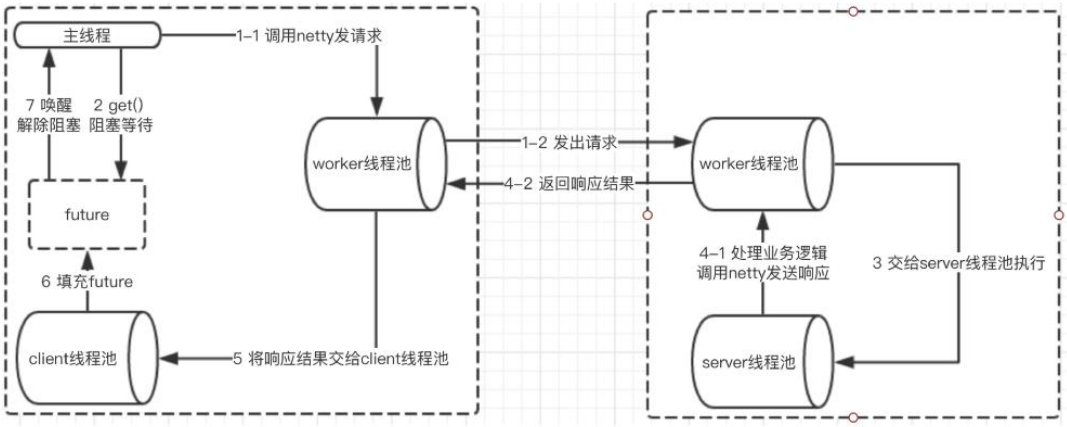
首先明确一个基本概念：IO 线程和业务线程的区别

- 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"/>
```