**文档修订记录**

| **日期** | **版本号** | **描述** | **著者** | **审阅者** | **审阅日期** |
| --- | --- | --- | --- | --- | --- |
| 2018/4/11 | 1.0 | 创建模板 | MTKunlun |  |  |
| 2018/4/25 | 1.1 | 修改技术亮点、增加源码解读 | MTKunlun |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |
|  |  |  |  |  |  |

目 录

目录

[1. 引言 3](#_Toc490127982)

[1.1. 文档说明 3](#_Toc490127983)

[1.2. 阅读对象 3](#_Toc490127984)

[1.3. 选用原因 3](#_Toc490127985)

[2. 技术简介 4](#_Toc490127986)

[2.1. 应用场景说明 4](#_Toc490127987)

[2.2. 基本组成要素 4](#_Toc490127988)

[2.3. 工作原理说明 4](#_Toc490127989)

[2.4. 使用指导 4](#_Toc490127990)

[2.5. 同类技术优劣势对比 4](#_Toc490127991)

[2.6. 相关技术说明 4](#_Toc490127992)

[3. 程序DEMO 5](#_Toc490127993)

[4. 常见问题汇总 6](#_Toc490127994)

# 引言

*<必填：简略说明此项技术的来源和技术特点。>*

## 文档说明

此文档主要用于描述Dubbo技术的应用场景，基本要素，工作原理，技术优劣势，Demo及工作中常见问题，让读者能够快速的了解此技术的相关组成要素和工作原理，同时能够为后续的工作提供参考指引。

## 阅读对象

该文档主要用于以下人员使用：

平台的设计人员，开发人员，测试人员，维护人员。

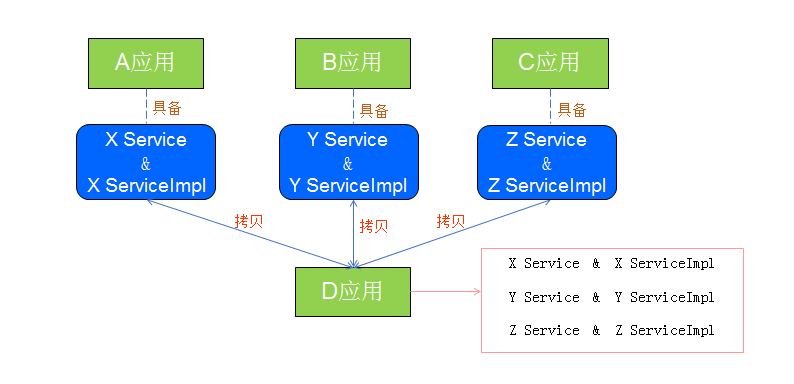
## 选用原因

Dubbo是Alibaba开源的分布式服务框架，它最大的特点是按照分层的方式来架构，使用这种方式可以使各个层之间解耦合（或者最大限度地松耦合）。从服务模型的角度来看，Dubbo采用的是一种非常简单的模型，要么是提供方提供服务，要么是消费方消费服务，所以基于这一点可以抽象出服务提供方（Provider）和服务消费方（Consumer）两个角色。关于注册中心、协议支持、服务监控等内容，详见后面描述。

# 技术简介

## 应用场景说明

### 传统服务调用

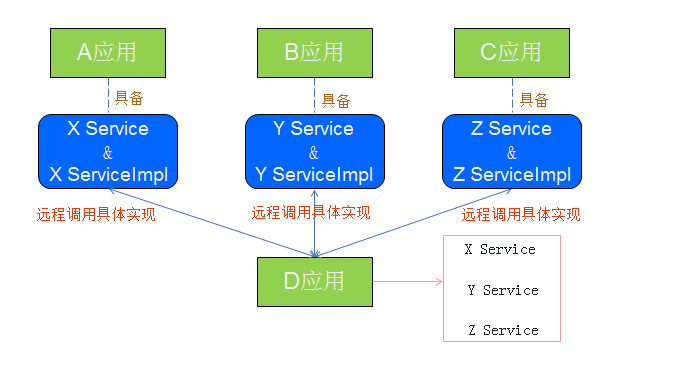


#### 传统服务调用局限性

通过上图，我们可以知道，随着A、B、C应用版本的不断更新，D应用也要重复再重复的将X、Y、Z的应用的实现拷贝到自身的应用当中。并且随着D应用的需求的不断更新，只会有更多的E应用、F应用、G应用的服务亦或者是A、B、C应用的其他服务需要添加进来，那么此时D应用就会随着需求的不断增加而使得本身十分的臃肿庞大。

因此，为了避免这些问题，远程调用服务应运而生。就好比如说我想要上网，我只需购买订阅运营商的宽带服务买个光猫即可上网，而不需要自己还要在家里搭个基站才能上网，内部怎么实现是运营商（服务提供者）的问题，而不是服务使用者的问题。

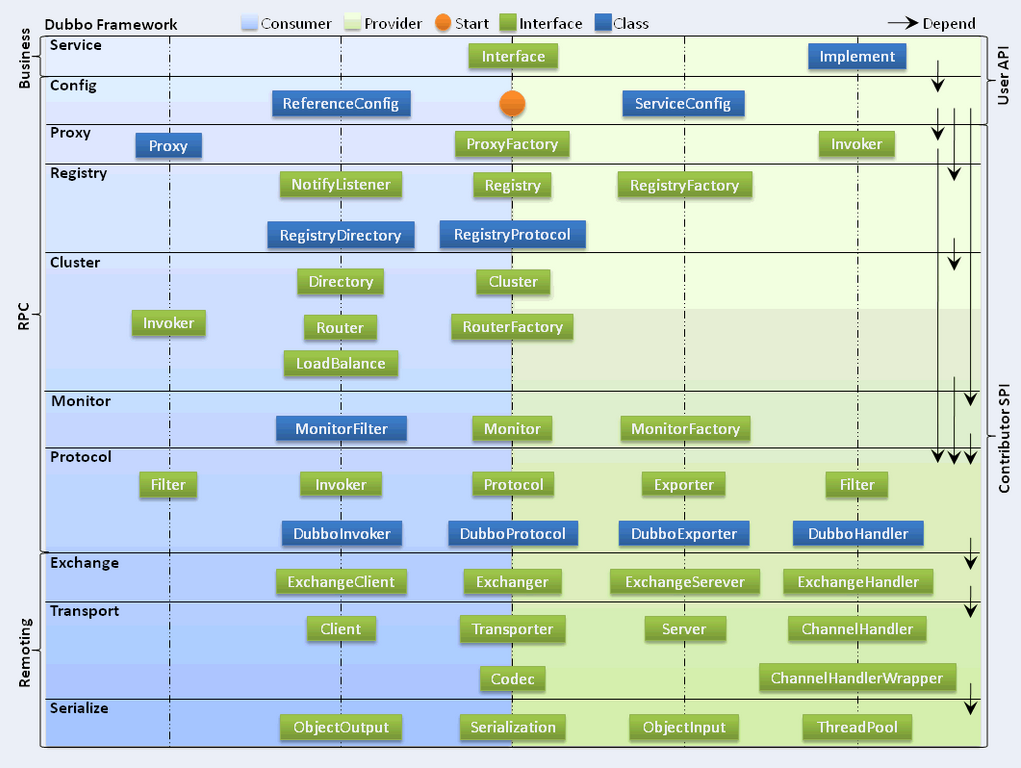
### 远程服务调用



从上图中，我们不难看到与之前的传统服务调用不同的是，D应用中只将X、Y、Z的接口拷贝了过来，但是并没有把对应的实现拷贝过来，那么这样当A、B、C应用的Service在更新的时候，D应用也能够直接获取到对应的实现更新，而不需重新的把A、B、C应用的更新实现拷贝到D应用中来。这样就解决了传统服务调用方法的局限性问题。

而Dubbo就是为了解决远程调用服务应运而生的RPC框架。

## 基本组成要素



Dubbo框架设计一共划分了10个层，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。

下面，结合Dubbo官方文档，我们分别理解一下框架分层架构中，各个层次的设计要点：

* 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，实现对服务消费方来透明，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，其它模型都向它靠扰，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）。

根据官方提供的，对于上述各层之间关系的描述，如下所示：

在RPC中，Protocol是核心层，也就是只要有Protocol + Invoker + Exporter就可以完成非透明的RPC调用，然后在Invoker的主过程上Filter拦截点。

图中的Consumer和Provider是抽象概念，只是想让看图者更直观的了解哪些类分属于客户端与服务器端，不用Client和Server的原因是Dubbo在很多场景下都使用Provider、Consumer、Registry、Monitor划分逻辑拓普节点，保持统一概念。

而Cluster是外围概念，所以Cluster的目的是将多个Invoker伪装成一个Invoker，这样其它人只要关注Protocol层Invoker即可，加上Cluster或者去掉Cluster对其它层都不会造成影响，因为只有一个提供者时，是不需要Cluster的。

Proxy层封装了所有接口的透明化代理，而在其它层都以Invoker为中心，只有到了暴露给用户使用时，才用Proxy将Invoker转成接口，或将接口实现转成Invoker，也就是去掉Proxy层RPC是可以Run的，只是不那么透明，不那么看起来像调本地服务一样调远程服务。

而Remoting实现是Dubbo协议的实现，如果你选择RMI协议，整个Remoting都不会用上，Remoting内部再划为Transport传输层和Exchange信息交换层，Transport层只负责单向消息传输，是对Mina、Netty、Grizzly的抽象，它也可以扩展UDP传输，而Exchange层是在传输层之上封装了Request-Response语义。

Registry和Monitor实际上不算一层，而是一个独立的节点，只是为了全局概览，用层的方式画在一起。

## 工作原理说明

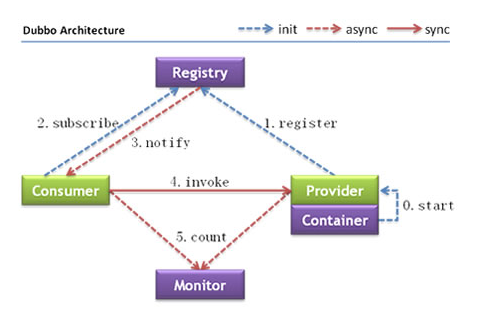
#### Dubbo主要核心部件

1、Remoting：网络通信框架，实现了sync-over-async和request-response消息机制。

2、RPC：一个远程过程调用的抽象，支持负载均衡、容灾和集群功能。

3、Registry：服务目录框架用于服务的注册和服务事件发布和订阅。

#### Dubbo架构



调用关系

0、服务器负责启动，加载，运行提供者（例如在tomcat容器中，启动dubbo服务端）

1、提供者在启动时，向注册中心注册自己提供的服务。

2、消费者启动时，向注册中心订阅自己所需的服务。

3、注册中心返回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、消费者，从远程接口列表中，调用远程接口，dubbo会基于负载均衡算法，选一台提供者进行调用，如果调用失败则选择另一台。

5、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。（可以在dubbo的可视化界面看到）

#### Dubbo原理 —— 解析服务

1、基于dubbo.jar内的Meta-inf/spring.handlers配置，spring在遇到dubbo名称空间时，会回调DubboNamespaceHandler类。

2、所有的dubbo标签，都统一用DubboBeanDefinitionParser进行解析，基于一对一属性映射，将XML标签解析为Bean对象。

#### Dubbo原理 —— 暴露服务

1、只暴露服务端口

使用场景：适用于开发环境下，当Provider与Consumer的IP一致；

实现方式：基于Adaptive机制，通过URL的”dubbo :// ”协议头识别，调用DubboProtocol的export()方法，直接打开服务端口；

2、向注册中心暴露服务端口

使用场景：适用于测试环境或正式部署环境，当Provider与Consumer的IP不一致

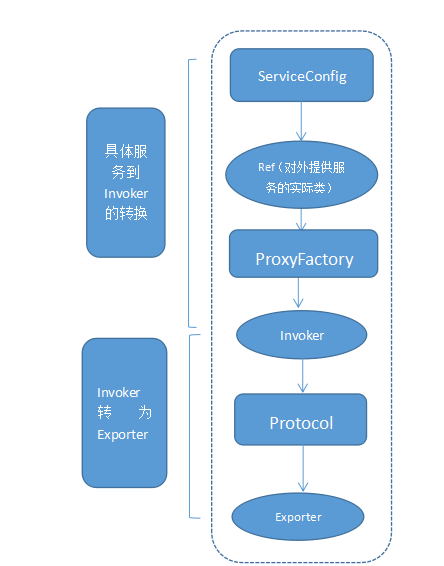
实现方式：基于Adaptive机制，通过URL的”registry://”协议头识别，调用RegistryProtocol的export()方法，先注册URL到注册中心，再打开服务端口。

#### Dubbo原理 —— 引用服务

1、直接引用服务；

2、从注册中心发现服务并引用。

#### 服务提供者暴露服务流程



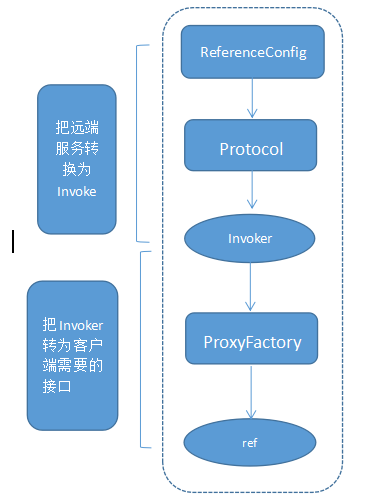
Dubbo处理服务暴露的关键就在Invoker转换到Exporter的过程，下面我们以Dubbo和rmi这两种

典型协议的实现来进行说明。

Dubbo的实现：   
Dubbo协议的Invoker转为Exporter发生在DubboProtocol类的export方法，它主要是打开socket侦听服务，并接收客户端发来的各种请求，通讯细节由dubbo自己实现。

Rmi的实现：   
RMI协议的Invoker转为Exporter发生在RmiProtocol类的export方法，他通过Spring或Dubbo或JDK来实现服务，通讯细节由JDK底层来实现。

#### 服务消费者消费服务流程



## 技术亮点

#### 服务治理方案（SOA）

#### 应用架构增长带来的影响

1、当网站变大后，不可避免的需要拆分应用进行服务化，以提高开发效率，调优性能，节省关键竞争资源等。

2、当服务越来越多时，服务的URL地址信息就会爆炸式增长，配置管理变得非常困难，F5硬件负载均衡器的单点压力也越来越大。

3、当进一步发展，服务间依赖关系变得错踪复杂，甚至分不清哪个应用要在哪个应用之前启动，架构师都不能完整的描述应用的架构关系。

4、接着，服务的调用量越来越大，服务的容量问题就暴露出来，这个服务需要多少机器支撑？什么时候该加机器？

针对以上的问题以及当应用架构越发庞大时可能出现的问题，Dubbo提供了一套较为完整的服务治理方案，这也是Dubbo这个框架比较突出的一个技术特点。

#### 服务降级

使用Dubbo在进行服务调用时，可能由于各种原因（服务器宕机/网络超时/并发数太高等），调用中就会出现RpcException，调用失败。服务降级就是指在由于非业务异常导致的服务不可用时可以返回默认值，避免异常影响主业务的处理。

Dubbo提供了Mock机制来实现服务降级，通过设置mock的参数值来实现服务降级。目前来说，mock的配置支持两种，一种为boolean值，默认的为false。如果配置为true，则缺省使用mock类名，即类名+Mock后缀；另外一种则是配置”return null”，可以很简单的忽略掉异常。

#### 软负载均衡

当一个服务有多个提供者时，Dubbo会进行正确的提供者选择。Dubbo提供一系列的软负载均衡策略：

1. Random: 随机，按权重设置随机概率；在一个截面上碰撞的概率高，但调用量越大分布越均匀，而且按概率使用权重后也比较均匀，有利于动态调整提供者权重；
2. RoundRobin：轮循，按公约后的权重设置轮循比率，轮询选择提供者；存在慢的提供者累积请求问题，比如：第二台机器很慢，但没挂，当请求调到第二台时就卡在那，久而久之，所有请求都卡在调到第二台上，可以通过结合权重，把第二台机(性能低的)的权重设置低一点；
3. LeastActive：最少活跃调用数，相同活跃数的随机，活跃数指调用前后计数差；使慢的提供者收到更少请求，因为越慢的提供者的调用前后计数差会越大；
4. ConsistentHash：一致性Hash，相同参数的请求总是发到同一提供者；当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。

#### 集群容错

集群容错：是指在一个服务下去暴露多个provider（即同一个服务有多个服务提供者支持），可以有任何一个provider宕机，不影响服务的正常提供。Dubbo提供多种容错策略：

1. failover：失败时自动切换，当出现失败，重试其他服务器（缺省是这种模式）常用于读操作，但重试会带来更长延迟，可以通过reties属性来设置重试次数，不含第一次。
2. failfast：快速失败，只发起一次调用，失败立即报错。通常用于非幂等性的写操作，比如新增记录
3. failsafe：出现错误，直接忽略，不重试也不报错，不影响后面的操作
4. failback：失败后不报错，会将该失败请求，定时重发，适合消息通知类型的服务
5. forking：并行调用多个服务器，只要在某一台提供者上面成功，那么方法返回，适合实时性要求较高的查询服务，但要牺牲性能。因为每台服务器会做同一个操作
6. broadcast：广播调用所有服务提供者，逐个调用，任意一台报错则报错。  适合与更新每台提供者上面的缓存这类型的服务

#### 调度中心

Dubbo官方出品了一个调度中心平台，在这个调度中心平台上可以实时对注册的服务以及服务调用者进行调度处理，并且该调度中心提供了路由规则、动态配置等功能配置，较为完整的提供了一套调度方案。官方调度中心主界面如下：



具体的搭建方式请参看我的博客：

或者是在网上搜索，有很多搭建教程资源。

#### 监控中心

Dubbo官方出品了一个监控中心平台，在这个监控中心平台可以实时查看服务的调用次数跟调用情况，以及服务的调用所耗时间等，并且该监控平台还会对服务调用情况生成一个图表实现可视化视图，更加清晰明了的让用户去了解服务的状态和调用情况。官方监控中心主界面如下：



具体的搭建方式请参看我的博客：

或者是在网上搜索，有很多搭建教程资源。

#### 服务治理解决

1、当服务越来越多时，服务的URL地址信息就会爆炸式增长，配置管理变得非常困难，F5硬件负载均衡器的单点压力也越来越大；

解决方式：此时需要一个服务注册中心，动态的注册和发现服务，使服务的位置透明。并通过在消费方获取服务提供方地址列表，实现软负载均衡和Failover，降低对F5硬件负载均衡器的依赖，也能减少部分成本。

2、当进一步发展，服务间依赖关系变得错踪复杂，甚至分不清哪个应用要在哪个应用之前启动，架构师都不能完整的描述应用的架构关系；

解决方式：这时，需要自动画出应用间的依赖关系图，以帮助架构师理清关系。（通过dubbo-keeper <https://github.com/dubboclub/dubbokeeper>）

3、接着，服务的调用量越来越大，服务的容量问题就暴露出来，这个服务需要多少机器支撑？什么时候该加机器？

解决方式：为了解决这些问题，第一步，要将服务现在每天的调用量，响应时间，都统计出来，作为容量规划的参考指标。 其次，要可以动态调整权重，在线上，将某台机器的权重一直加大，并在加大的过程中记录响应时间的变化，直到响应时间到达阀值，记录此时的访问量，再以此访问量乘以机器数反推总容量。

#### 服务注册中心

1. Multicast
2. Zookeeper（官方推荐）
3. Redis
4. Simple

……

#### 多协议

不同服务在性能上适用不同协议进行传输，比如大数据用短连接协议，小数据大并发用长连接协议。

1. dubbo缺省协议。Dubbo缺省协议采用单一长连接和NIO异步通讯，适合于小数据量大并发的服务调用，以及服务消费者机器数远大于服务提供者机器数的情况。Dubbo缺省协议不适合传送大数据量的服务，比如传文件，传视频等，除非请求量很低。
2. RMI协议。采用JDK标准的java.rmi.\*实现，采用阻塞式短连接和JDK标准序列化方式。
3. Hessian协议。用于集成Hessian的服务，Hessian底层采用Http通讯，采用Servlet暴露服务，Dubbo缺省内嵌Jetty作为服务器实现。
4. http协议。当前 dubbo 支持的 thrift 协议是对 thrift 原生协议的扩展，在原生协议的基础上添加了一些额外的头信息，比如service name，magic number等。使用dubbo thrift协议同样需要使用thrift的idl compiler编译生成相应的java代码。
5. thrift协议。当前 dubbo 支持的 thrift 协议是对 thrift 原生协议的扩展，在原生协议的基础上添加了一些额外的头信息，比如service name，magic number等。使用dubbo thrift协议同样需要使用thrift的idl compiler编译生成相应的java代码

……

## 使用指导

## 同类技术优劣势对比

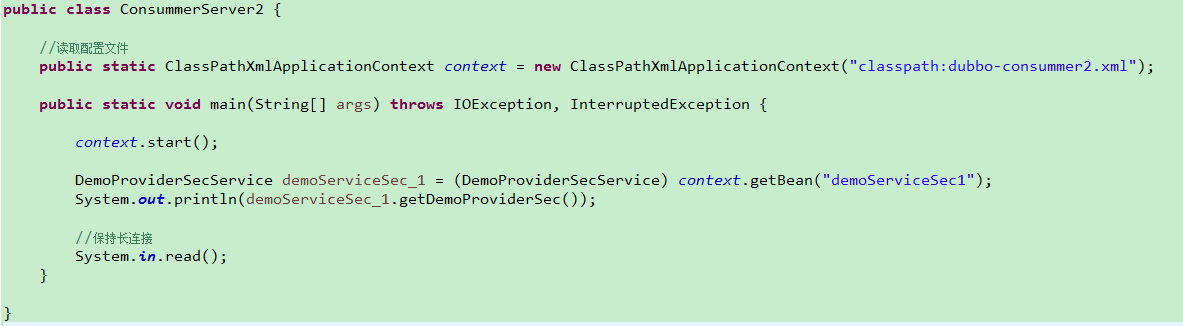
## 相关技术说明

1. zookeeper
2. spring机制,自动扫描注解，自动启动服务器
3. 反射
4. 动态代理
5. socket
6. nio-netty
7. 注解

# 源码解读

## Consumer代理对象生成流程

Dubbo号称通过spring的方式可以透明化接入应用，对应用没有任何api侵入。我们通过分析Consumer的配置获取方式来感受一下：





从上面的Demo截图我们可以看到，代码方面确实是零侵入，而在配置方面，则是增加了一些服务的声明，环境配置之类的（不可缺少）。对于开发者来说非常友好。那么dubbo是如何做到这点的呢？具体做法如下：

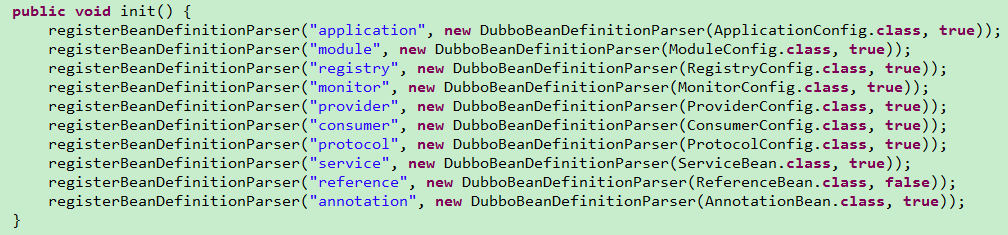
1、通过Spring的FactoryBean进行反射实现具体Consumer对象的实例化创建；

2、通过Spring的schema扩展机制实现Spring对dubbo标签的解析。

刚刚所展示的Demo代码中，里面并没有看到哪个类是实现了FactoryBean这个接口，这里我们则需要引入Spring的schema扩展机制。

在dubbo中，所有namespace=dubbo的标签将通过com.alibaba.dubbo.config.spring.schema.DubboNamespaceHandler被dubbo自己解析。而这个类中有一个init()方法，在该方法中通过调用了org.springframework.beans.factory.xml.NamespaceHandlerSupport类的registerBeanDefinitionParser()来进行xml标签的解析。

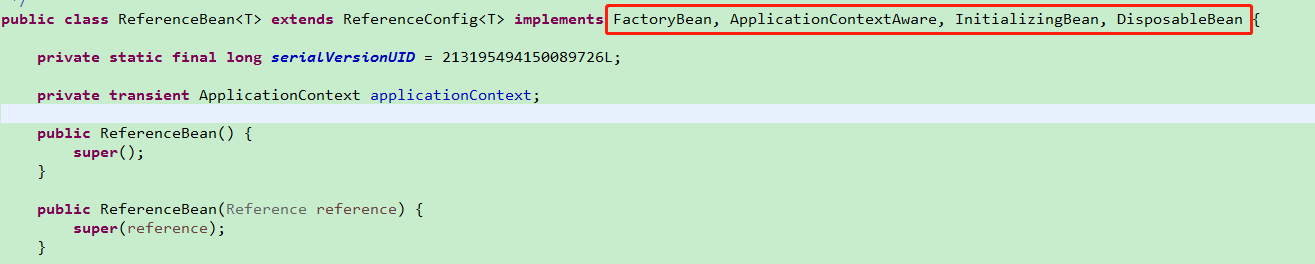
在com.alibaba.dubbo.config.spring.schema.DubboNamespaceHandler的init()方法中，我们可以看到在dubbo配置文件中的几个标签配置：



根据配置文件中的配置，我们很清楚的知道reference标签正是consumer的配置标签，因此，consumer的代理对象生成正是由此开始。

之前我们也提到了代理对象的生成是通过Spring的FactoryBean进行反射实现具体Consumer对象的实例化创建,那么在dubbo中是哪一个类去实现了FactoryBean这个接口完成了对象的实例化操作呢？

我们通过观察registerBeanDefinitionParser()方法中的参数，很容易就发现根据字面上的意思ReferenceBean就是消费者的实例对象，那么当我们点击进去的时候，发现正好ReferenceBean是实现了FactoryBean这个接口的，因此我们得出结论在Dubbo中是对应的标签实例对象实现了FactoryBean这个接口。

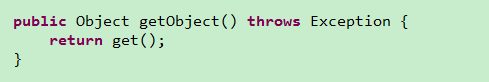


根据ReferenceBean的实现接口，我们可以看到其实现了FactoryBean,ApplicationContextAware, InitializingBean, DisposableBean这四个接口。

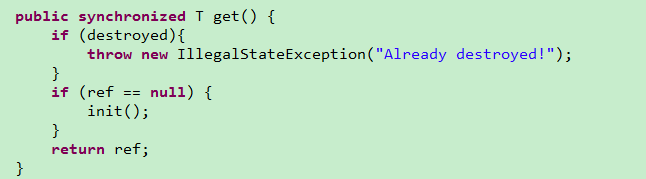
由于实现了InitializingBean这个接口，因此在初始化各个属性后会调用afterPropertiesSet()， 实现比较简单，主要是对没有初始化的几个属性尝试用公共的默认配置进行初始化，由于afterPropertiesSet()该方法方法体太长，我对这个方法进行了总结缩减，总结如下：



由于实现了FactoryBean，当需要初始化或者应用中需要用到时，会调用getObject()方法获取实际的对象：



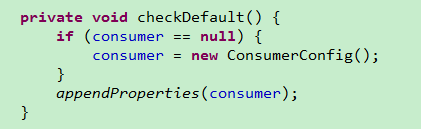
getObject()方法中调用了父类ReferenceConfig中的get方法：



可以看到在get()方法中，如果对象还未被创建，则进行init()初始化操作：



在init()方法中调用ReferenceConfig类中的checkDefault()方法：



此方法中调用了appendProperties()方法进行ConsumerConfig中的默认信息配置，appendProperties从System.getProperty中获取配置，如果有相应值则替换配置文件中的值。

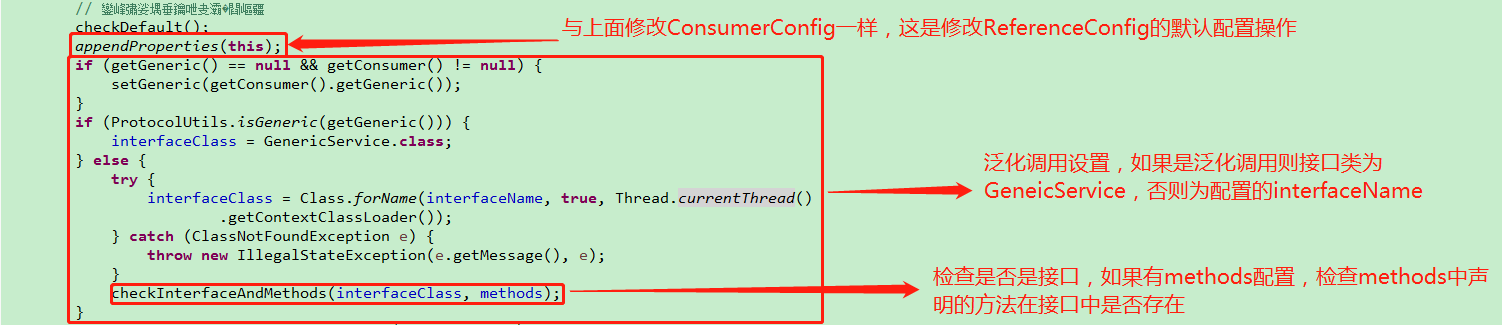
比如对于consumer来说，如果配置了timeout=5000, 可以通过：

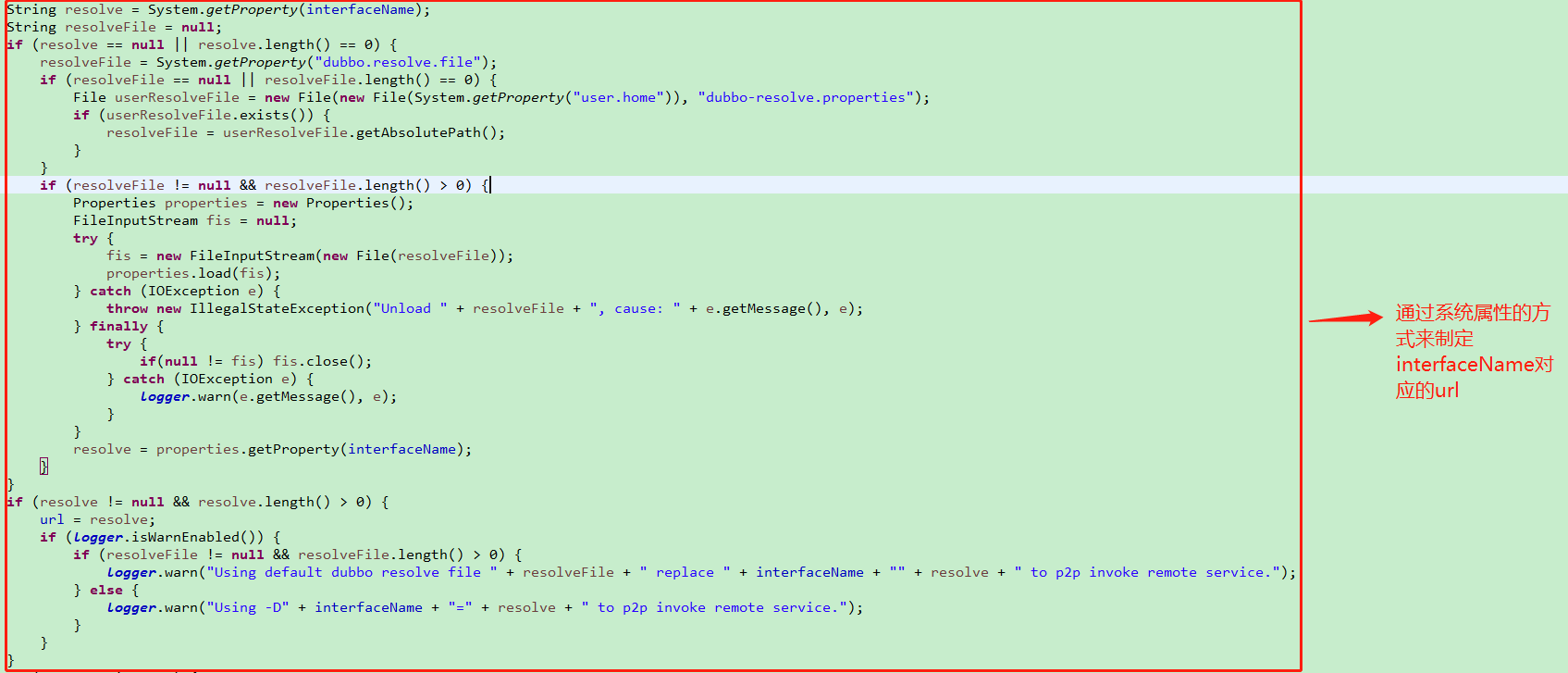
1、在启动参数中设置dubbo.consumer.timeout=3000来修改这个值；

2、在启动参数中设置dubbo.consumer.com.alibaba.dubbo.config.ConsumerConfig.timeout=3000来修改这个值。

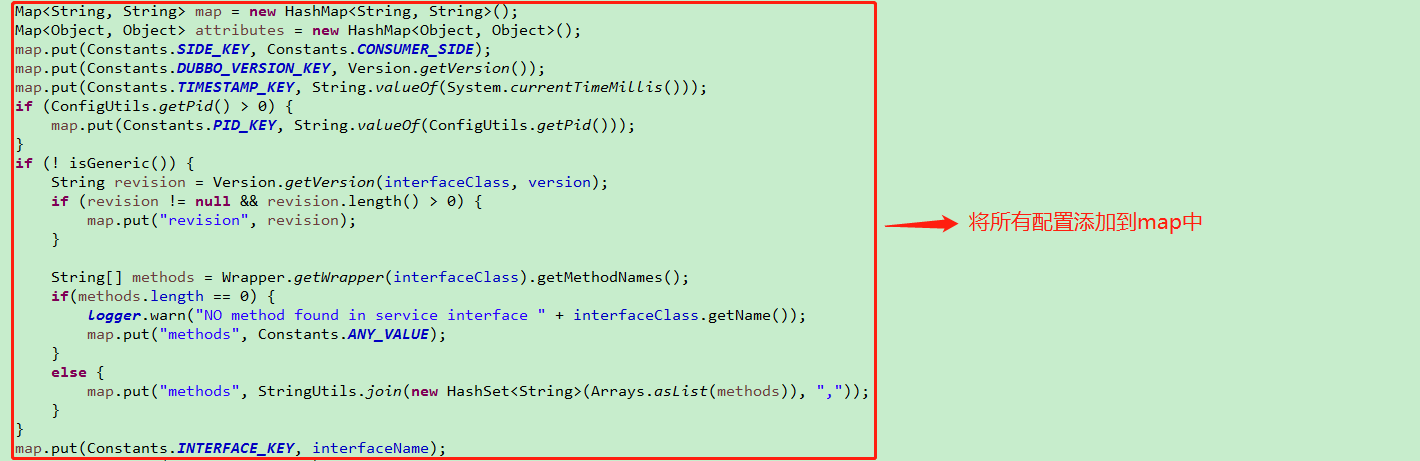
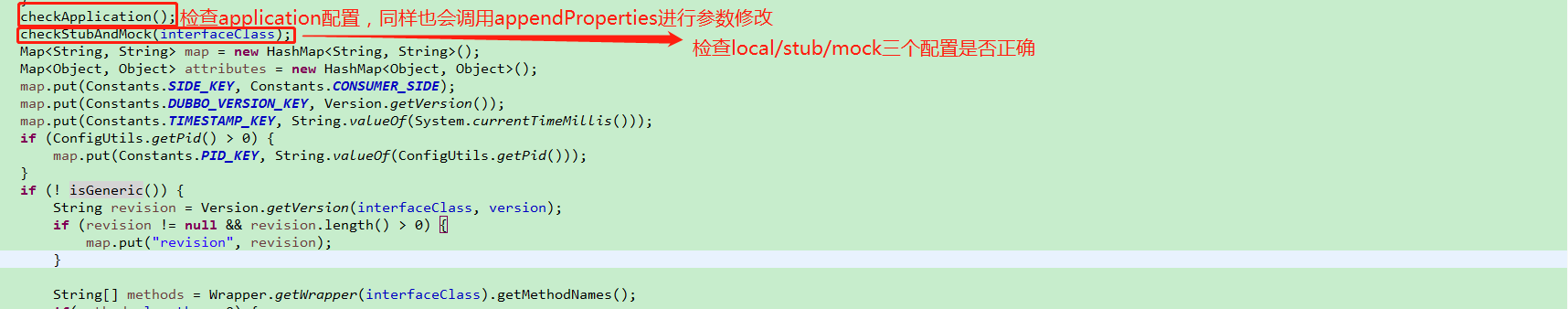
其中第二种方式优先级高于第一种，修改其他属性只需要替换"timeout"。

接着init()方法继续走下去，来到了appendProperties(this)这个方法，这个方法其实跟之前appendProperties(consumer)方法的作用一样的，只是传入的参数不一样，前者是传入this,即ReferenceConfig，后者是传入consumer，即ConsumerConfig，因此可以从字面上知道一个是对ReferenceConfig进行默认配置设置，另一个是对ConsumerConfig进行默认配置设置。









即检查远端和本地服务接口是否真实存在以及是否可实现加载

包括：

SIDE\_KEY(是属于提供者还是消费者)；

DUBBO\_VERSION\_KEY(dubbo的当前版本)；

TIMESTAMP\_KEY(当前时间戳)；

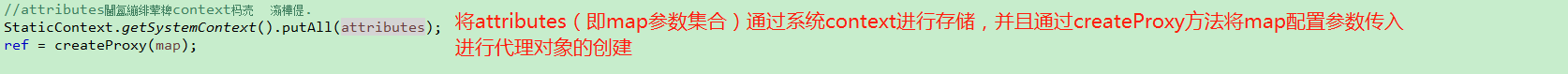
PID\_KEY(当前的进程号)；

revision(这是在provider提供的服务地址中的revision字段, 不一定会提供，如果有的话则放进map集合中)；

methods(这是在provider提供的服务地址中的methods字段)；

INTERFACE\_KEY(即interfaceName)





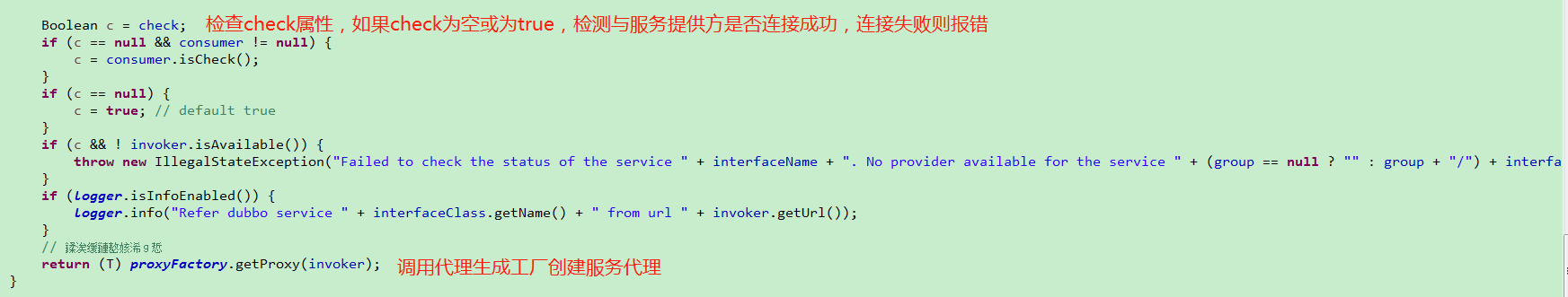
可以看到init方法会先做大量的配置初始化和检查工作，并将生成的配置放入map中，通过map创建代理。





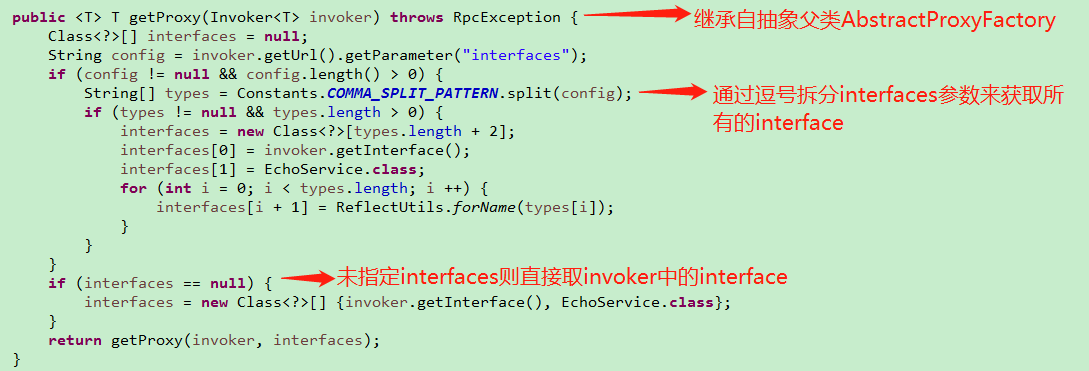
在loadRegistries方法中最重要的就是设置了protocol为Constants.REGISTRY\_PROTOCOL,因此我们可以认为此处使用的是RegistryProtocol

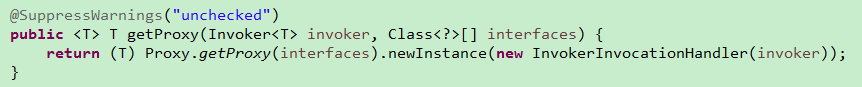




proxyFactory.getProxy()方法默认的实现为

com.alibaba.dubbo.rpc.proxy.javassist.JavassistProxyFactory(可通过ProxyFactory的SPI标签查看默认的实现), 它的getProxy方法实现如下：



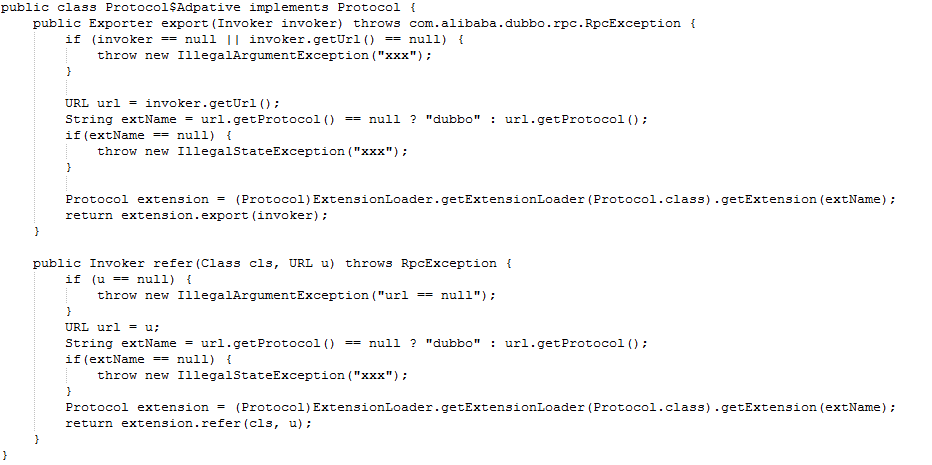


从刚刚解释createProxy()方法创建代理对象我们可以看到，实际代理对象的创建是通过proxyFactory.getProxy()传进一个invoker参数进行代理的创建。

而创建代理之前需要先加载注册中心的配置，并生成monitor对应的key。

接下来就是根据上面加载到的信息来创建invoker,invoker是dubbo中非常重要的一个概念，它代表一个可执行体，通过调用它的invoke方法来进行本地/远程调用，并获取结果。

它的生成有两个分支，第一个分支是urls.size() == 1，从前面的代码可以知道当注册中心只有一个（单个或集群）时进入此分支，此处的refprotocal是由ExtensionLoader生成的代理类，大致实现代码如下：



这个代理类的实现比较简单，我们知道一个接口的实现者可能有多个，此时到底注入哪一个呢？这个代理类就是根据URL中的protocol加载实际的实现；

从上面的代码中可以看到，Protocol$Adpative是根据URL参数中protocol属性的值来选择具体的实现类的。

如值为dubbo，则从ExtensionLoader(Protocol)中获取dubbo对应的实例，即DubboProtocol实例。

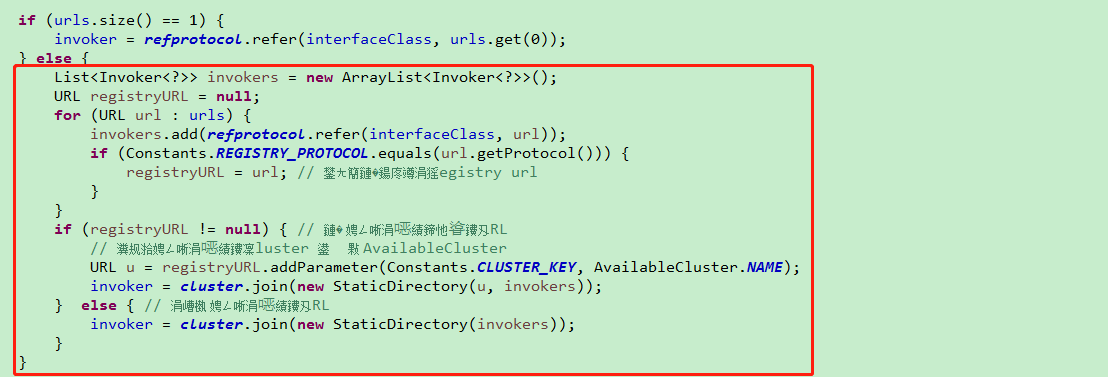
refprotocol.refer方法中urls.get(0)的protocol为registry，因此最终调用的是com.alibaba.dubbo.registry.integration.RegistryProtocol的refer方法：



refprotocol.refer方法中最后调用的是doRefer()方法:



现在我们来看看第二个分支，即urls.size() != 1的情况，此时针对每个url都会生成一个ClusterInvoker，然后将其放入StaticDirectory管理。对于StaticDirectory来说，其下的任意一个注册中心可用则其可用：



到此，Consumer的代理生成完成，总结如下：

1、spring加载时拦截namespace=dubbo的标签进行解析，生成dubbo中的config；

2、consumer对应的直接配置为ReferenceConfig， reference的config加载完成后，分别使用ConsumerConfig、ApplicationConfig、ModuleConfig、RegistryConfig、MonitorConfig等的默认值来初始化ReferenceConfig;

3、在创建bean的对象时，如果已经创建过则不重复创建，否则进入创建流程，并将是否创建的标识设为true；

4、使用系统参数、配置文件覆盖api/xml中设置的配置，将所有配置项存入map;

5、获取注册中心配置，根据配置连接注册中心，并注册和订阅url的变动提醒；

6、生成cluster invoker；

7、根据接口生成代理类，并创建对象，创建时传入InvokerInvocationHandler，该handler封装了上面生成的invoker，最终的接口调用都是通过invoker.invoke（）。

# 常见问题汇总

无