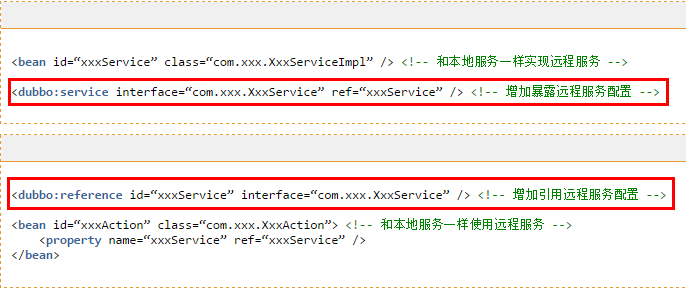
# 用法

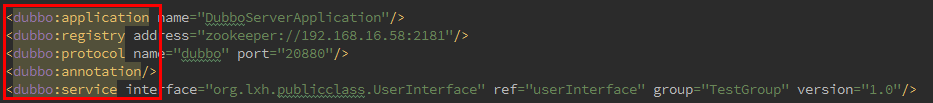
最基本的配置（使用Spring方式进行配置，Dubbo的使用可以使用Spring配置方式和API调用方式进行设置，但不推荐API使用，因为其比较复杂且不易维护）：



# 快速启动

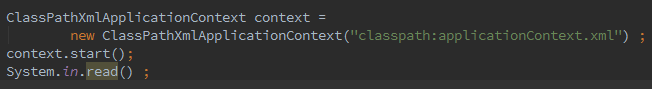
## 服务生产者

### Spring XML Config



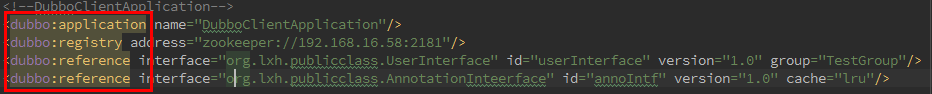
### 启动：

可以通过Main函数启动，也可以通过Web容器启动，Main函数启动较为常见（同Schedule工程一样）



## 服务消费者

### Spring XML Config



### 应用

同上，可以实例化一个Spring的上下文对象获取Bean来进行使用；更多的是作为一个Bean实例，依赖注入所需要的其他对象的实例（使用方法与一般对象实例无异）

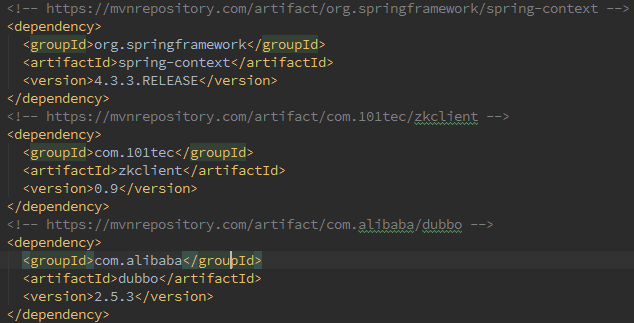
# 依赖

## JDK

官网说明JDK版本应该1.5+，目前公司JDK版本为1.7，客户端与服务端工作正常

## JAR包依赖

Dubbo本身的基础功能很简单，便是RPC远程服务调用，但是其的高度可扩展性使得它成为了一套非常成熟的微服务架构，针对不同的配置与扩展（指的是功能与协议配置类的改变），对应的则会有不同的JAR依赖，如果只是简单的使用，少许的依赖便可满足



# XML配置

所有标签支持自定义参数，用于不同扩展点实现特殊配置：



如上，当Dubbo协议设置为jms时，需要设置“queue”参数才能实例化相应的对象

## 常用配置

### <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:module/>

模块配置，用于配置当前模块信息，可选。

### <dubbo:registry/>

注册中心配置，用于配置连接注册中心相关信息。

### <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。

## 配置优先级

方法级别>接口级别>全局配置级别；其中客户端配置>服务端配置；

此外在XML中的相应配置项也可以在dubbo.properties属性文件中封装为通用的配置文件，当Dubbo在XML中找不到相应的配置项时，则会从对应项目的classpath下加载dubbo.properties文件（如果存在多个相同的配置文件，会导致报Error），此时的优先级为：XML配置>属性文件配置

## 注解配置

@Service(version="1.0.0") ----- 注册服务

@Reference(version="1.0.0") ----- 引用服务

<dubbo:annotation/> ----- 开启Dubbo注解功能

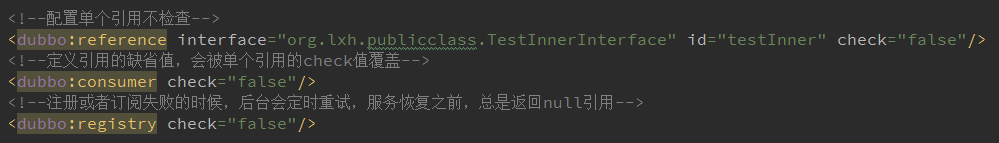
# API配置

API仅用于OpenAPI, ESB, Test, Mock等系统集成，普通服务提供方或消费方，请采用配置方式使用Dubbo。如有特别的需求需要使用API方式的话，请参阅相关文档。

# 功能

## 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；引用缺省是延迟初始化的，只有引用被注入到其它Bean，或被getBean()获取，才会初始化*】根据上述结论，当我们在客户端采用默认配置的时候，默认都是延迟加载且带检查约束的，也就是说当我们在Java程序中引用配置的服务实例的时候（直接调用服务方法或者通过依赖注入等方式）便会抛出错误信息，阻止Spring容器初始化，错误信息为提示没有对应的Provider提供相应服务；但是若项目中没有引用这类服务的时候是不报错的，因为默认所有的服务都是延迟加载的；如果我们将其设置为“饿汉式”加载模式的话，则项目在启动的时候便会提示错误信息。最佳实践为尽量保持原本的检查模式，只有当某些服务不关心或者存在循环依赖的情况下需要关闭检查约束，这种条件下总是会返回一个null指针，服务恢复的时候可以自动重连。示例配置如下：



## 集群容错

【*在集群调用失败时，Dubbo提供了多种容错方案，缺省为failover重试*】其包含以下几种集群容错模式：

### Failover Cluster

失败自动切换，当出现失败，重试其它服务器。(缺省)

通常用于读操作，但重试会带来更长延迟。

可通过retries="2"来设置重试次数(不含第一次)。

### Failfast Cluster

快速失败，只发起一次调用，失败立即报错。

通常用于非幂等性的写操作，比如新增记录。

### Failsafe Cluster

失败安全，出现异常时，直接忽略。

通常用于写入审计日志等操作。

### Failback Cluster

失败自动恢复，后台记录失败请求，定时重发。

通常用于消息通知操作。

### Forking Cluster

并行调用多个服务器，只要一个成功即返回。

通常用于实时性要求较高的读操作，但需要浪费更多服务资源。

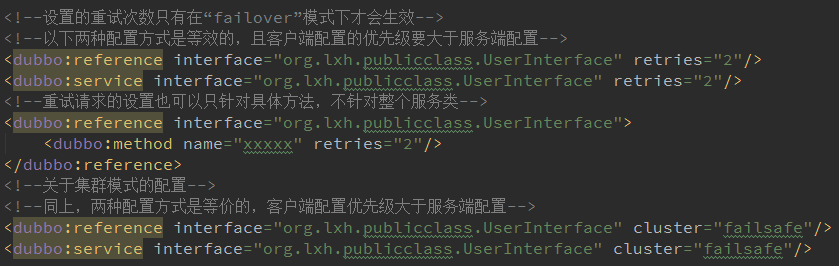
可通过forks="2"来设置最大并行数。

### Broadcast Cluster

广播调用所有提供者，逐个调用，任意一台报错则报错。(2.1.0开始支持)

通常用于通知所有提供者更新缓存或日志等本地资源信息。

以下为示例配置：



## 负载均衡

【*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*】Dubbo提供了多种不同的负载均衡策略模式，有如下几种：

### Random LoadBalance

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。

### RoundRobin LoadBalance

轮循，按公约后的权重设置轮循比率。

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。

### LeastActive LoadBalance

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。

### ConsistentHash LoadBalance

一致性Hash，相同参数的请求总是发到同一提供者。

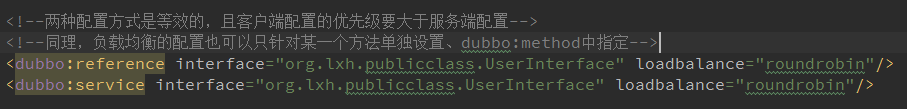
当某一台提供者挂时，原本发往该提供者的请求，基于虚拟节点，平摊到其它提供者，不会引起剧烈变动。

算法参见：http://en.wikipedia.org/wiki/Consistent\_hashing。

缺省只对第一个参数Hash，如果要修改，请配置<dubbo:parameter key="hash.arguments" value="0,1" />

缺省用160份虚拟节点，如果要修改，请配置<dubbo:parameter key="hash.nodes" value="320" />

关于负载均衡示例配置如下：



## 线程模型

【*如果事件处理的逻辑能迅速完成，并且不会发起新的IO请求，比如只是在内存中记个标识，则直接在IO线程上处理更快，因为减少了线程池调度；但如果事件处理逻辑较慢，或者需要发起新的IO请求，比如需要查询数据库，则必须派发到线程池，否则IO线程阻塞，将导致不能接收其它请求；如果用IO线程处理事件，又在事件处理过程中发起新的IO请求，比如在连接事件中发起登录请求，会报“可能引发死锁”异常，但不会真死锁。*】【比如，在Android中，耗时的任务是不能再UI主线程运行的，因为会阻断其他任务的执行，其必须在单独的线程中执行然后再返回结果】

### Dispatcher（分发的任务类型）

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。

direct 所有消息都不派发到线程池，全部在IO线程上直接执行。

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 只请求消息派发到线程池，不含响应，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。

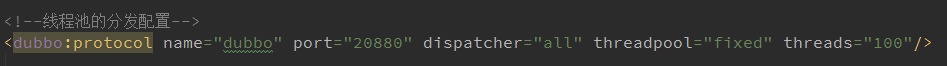
### ThreadPool（线程池种类）

fixed 固定大小线程池，启动时建立线程，不关闭，一直持有。(缺省)

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。(为避免收缩时突然来了大流量引起的性能问题)。

示例配置如下：

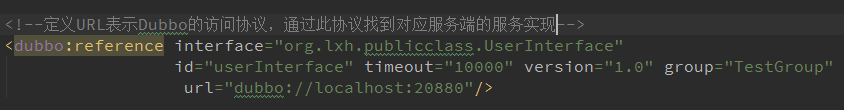


## 直连提供者

【*在开发及测试环境下，经常需要绕过注册中心，只测试指定服务提供者，这时候可能需要点对点直连，点对点直联方式，将以服务接口为单位，忽略注册中心的提供者列表，*

*A接口配置点对点，不影响B接口从注册中心获取列表。*】

示例如下（该配置对应的还可以修改属性文件的内容，不过修改XML配置文件的方式依然是最佳的方式）：



## 只订阅

使用场景【*为方便开发测试，经常会在线下共用一个所有服务可用的注册中心，这时，如果一个正在开发中的服务提供者注册，可能会影响消费者不能正常运行；可以让服务提供者开发方，只订阅服务(开发的服务可能依赖其它服务)，而不注册正在开发的服务，通过直连测试正在开发的服务*】

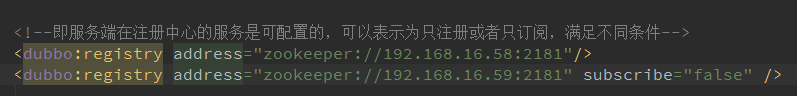
示例代码如下：



## 只注册

使用场景：【*如果有两个镜像环境，两个注册中心，有一个服务只在其中一个注册中心有部署，另一个注册中心还没来得及部署，而两个注册中心的其它应用都需要依赖此服务，所以需要将服务同时注册到两个注册中心，但却不能让此服务同时依赖两个注册中心的其它服务；可以让服务提供者方，只注册服务到另一注册中心，而不从另一注册中心订阅服务*】

示例配置如下：



## 静态服务

【有时候希望人工管理服务提供者的上线和下线，此时需将注册中心标识为非动态管理模式；即当服务端启动的时候，需要人工启用；当服务断掉的时候也需要人工禁用】

示例配置如下：



## 多协议

### 不同服务不同协议

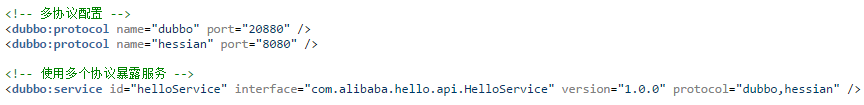
不同服务在性能上适用不同协议进行传输，比如大数据用短连接协议，小数据大并发用长连接协议

Dubbo执行一次RPC的请求过程：消费者（Consumer）向注册中心（Registry）执行RPC请求，注册中心分配服务URL并路由到具体服务提供方（Provider），消费者和服务提供方建立网络连接，服务提供方在本地创建连接池对象并提供远程服务，对于长连接类型协议（如dubbo协议）将保持连接，减少握手认证，调用过程中可以避免频繁建立和断开连接导致的性能开销，保持长连接需要有心跳包的发送，所以对于非频繁调用的服务保持连接同样会有消耗，故在项目的实际开发中，针对不同业务处理的服务，要适当选择不同的协议支持。



### 多协议暴露服务

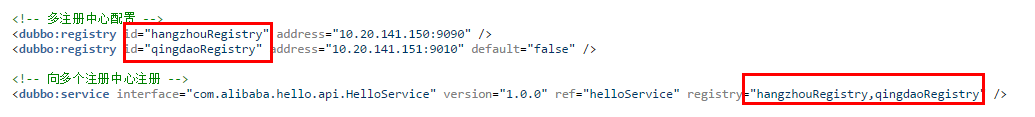
定义服务默认情况下，如果不显示指明使用哪种协议定义服务，则对应的服务会使用配置文件所配置的所有协议去定义，在引用服务的时候，默认引用最后一种协议，如需引用指定协议的服务，同样需要protocol去指定



## 多注册中心

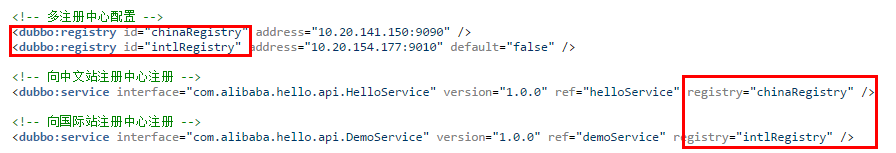
### 多注册中心注册

注册中心的配置与协议的配置相类似，如果在定义服务的时候不特别声明，则默认会注册到配置文件中的所有注册中心，据此有参数“default”选项来配置此行为；另外还有一点不同的是：registry配置项此时配置的应该是注册中心的ID了，这样才能正确引用（每一个Dubbo标签都会对应一个实例）。



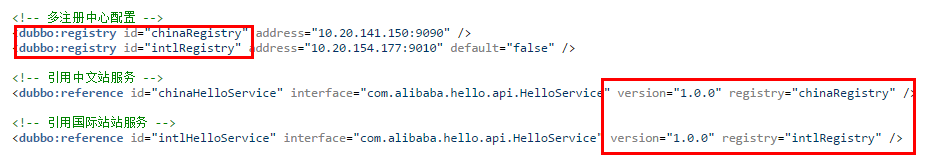
### 不同服务使用不同注册中心

应该会有这样一个需求，开发不同的服务针对性不同：比如地域、语言、区域等等，不同地域的RPC请求便会请求不同的注册中心



### 多注册中心引用

同上，可能某一个服务需要联合使用不同注册中心注册的服务（服务引用要求版本号，组名等元数据信息要保持一致，配置不同的注册中心获取不同的实例服务），多注册中心配置，竖号分割表示连接不同的注册中心，同一注册中心的多个集群地址用逗号来进行分割



## 服务分组【划分服务】

当一个接口有多种实现时，可以用group区分



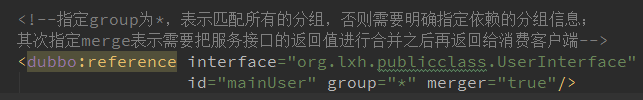
## 多版本【划分服务】

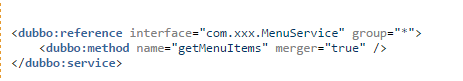
当一个接口实现，出现不兼容升级时，可以用版本号过渡，版本号不同的服务相互间不引用。（有一个服务升级规则：在低压力时间段，先升级一半的提供者为新版本，再将所有消费者升级为新版本，最后再将剩下的一般提供者升级为新版本）



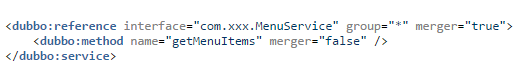
## 分组聚合【特性】

按组合并返回结果，比如菜单服务，接口一样，但有多种实现，用group区分，现在消费方需从每种group中调用一次返回结果，合并结果返回，这样就可以实现合并菜单项

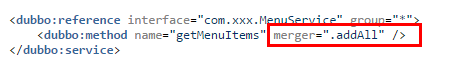




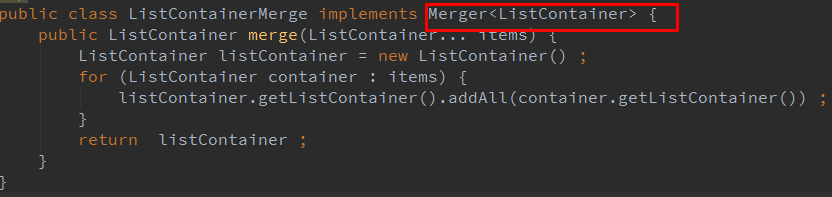
指定某个具体的方法返回结果需要进行合并，其余的方法不执行合并操作

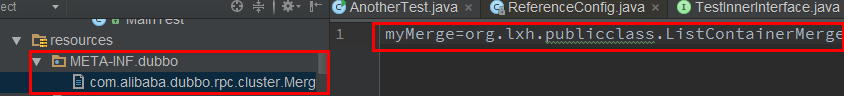


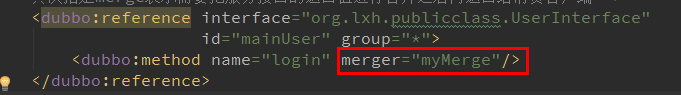
指定某个具体的方法不执行合并操作，其余方法都执行合并操作



这种配置合并结果的方式可能更为常用，适合于一些已经封装好的类





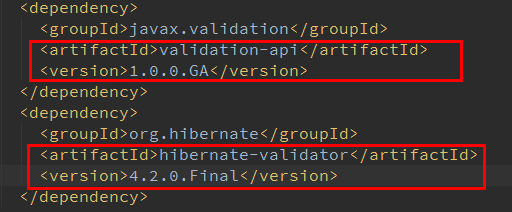


以上的使用方式更为灵活，方便的自定义扩展合并器，需要指定Key为合并器名称，Value为合并器完整类名，且配置文件名的位置和格式也是固定的，Dubbo会默认读取该配置文件

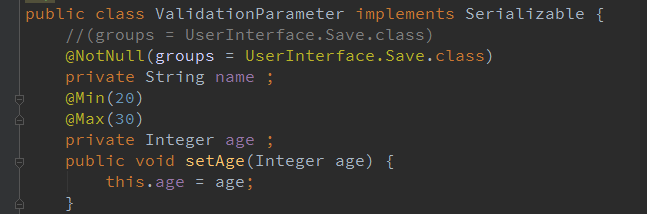
## 参数验证

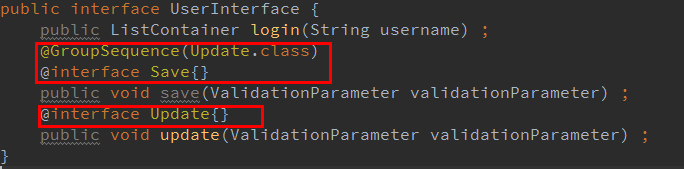
参数验证功能是基于***JSR303***实现的，用户只需标识JSR303标准的验证Annotation，并通过声明filter来实现验证

首先配置基本的依赖库，经测试采用最新版本的依赖可能会报错，按照文档中给出的即可



先参考如下示例截图：







***先复习确定一个概念：在Dubbo传输的过程中，所使用到的全部实体类都要实现序列化接口***；参数验证的低一点，在Bean中加入特定验证的Annotation标记，进一步可以配置组，即groups，表示这一条验证规则只在改组下生效，若无表示应用到所有场景；下一步组的定义还是依赖于接口方法的Annotation标记，且命名为首字母大写，比如上面的接口Save对应方法save，在此基础上，还可以继续定义GroupSequence配置，表示同时还需要满足的组条件（这一点测试的时候发现好像有bug，即只会验证Sequence配置的验证器，本身配置的不会执行），最后在服务的引用或者声明处开启验证选项（位置的不同表示验证是在服务端还是客户端进行）。



可以利用这种方式进行多个组的联合设置

## 结果缓存

用于加速热门数据的访问速度，Dubbo提供声明式缓存，以减少用户加缓存的工作量

***lru*** 基于最近最少使用原则删除多余缓存，保持最热的数据被缓存（是最常用的）。

***threadlocal*** 当前线程缓存，比如一个页面渲染，用到很多portal，每个portal都要去查用户信息，通过线程缓存，可以减少这种多余访问。

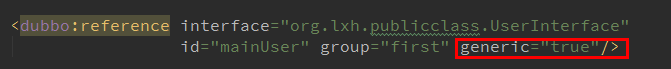
***jcache*** 与JSR107集成，可以桥接各种缓存实现。

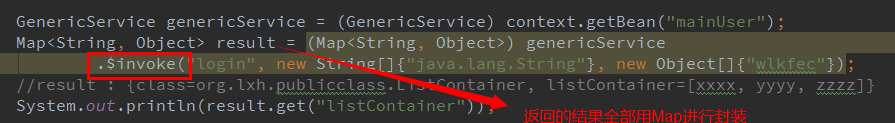


缓存设置，会对热门数据基于lru算法缓存，若服务器端数据频繁更新，则不建议缓存的使用

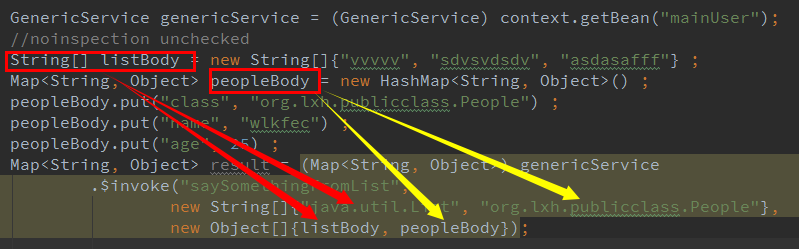
## 泛化引用

泛接口调用方式主要用于***客户端没有API接口及模型类元***的情况，即客户端只知道所调用服务的完整包名，方法名，参数类型等相关信息，参数及返回值中的所有***POJO均用Map表示***，通常用于框架集成，比如：实现一个通用的服务测试框架，可通过***GenericService***调用所有服务实现。示例调用如下：



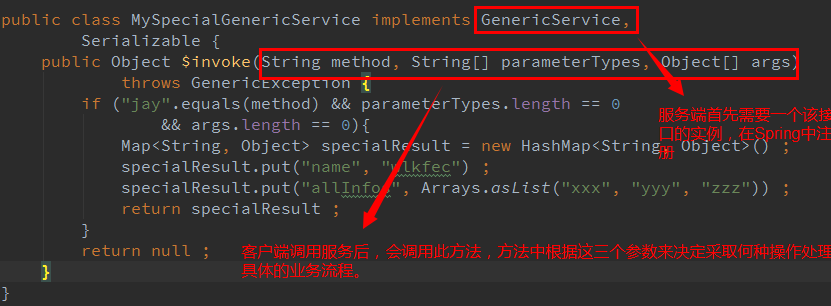


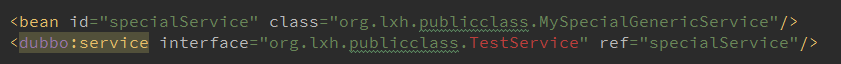
如果服务调用参数为POJO类对象，则可能封装的参数依然都为Map对象，参照如下：



## 泛化实现

泛接口实现方式主要用于***服务器端***没有API接口及模型类元的情况，参数及返回值中的所有POJO均用Map表示，通常用于框架集成，比如：实现一个通用的远程服务Mock框架，可通过实现***GenericService***接口处理所有服务请求，即后端服务采用了一种通用匹配的模式来处理客户端发送而来的服务请求，参考示例如下：

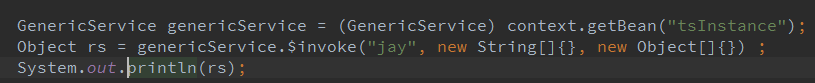




如上，服务端在Spring容器中注册该实例；Dubbo同时声明服务，这里的接口名代表服务，是需要注册到注册中心的，之所以这里报错，是因为这是在Spring的配置环境中，没有定义此接口，可以关闭文件验证，对程序无影响。



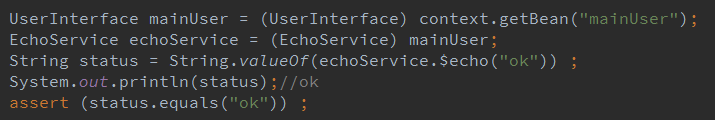
客户端同样声明此引用



根据后端业务逻辑，传入适合的参数。

## 回声测试

回声测试用于检测服务是否可用，回声测试按照正常请求流程执行，能够测试整个调用是否通畅，可用于监控；所有服务自动实现EchoService接口，只需将任意服务引用强制转型为EchoService，即可使用

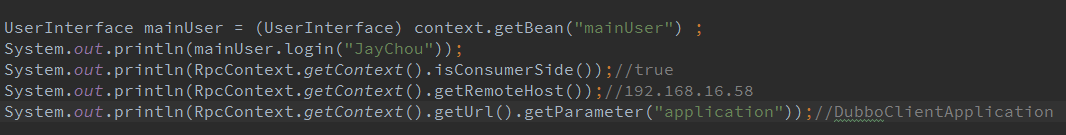


如上例，如果服务正常调用完毕，最后便会输出传入的参数，利用断言来进行正确性测试

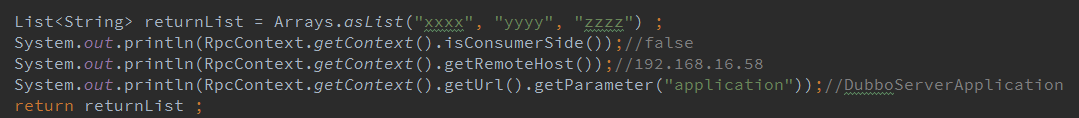
## 上下文信息

首先，上下文中存放的是当前调用过程中所需的环境信息；其次，在调用的过程中，所有配置信息都将转换为URL的参数进行传递。

***RpcContext是一个ThreadLocal的临时状态记录器（不同线程有不同的状态）***，当接收到RPC请求，或发起RPC请求时，RpcContext的状态都会变化，可能这一刻为消费者，下一秒就变为服务提供者了。比如：A调B，B再调C，则B机器上，在B调C之前，RpcContext记录的是A调B的信息，在B调C之后，RpcContext记录的是B调C的信息。使用参考如下：



客户端环境：获取IP、应用名称等信息



客户端环境：获取IP、应用名称等信息

## 隐式传参

首先，隐式传参的过程类似于一个cookie的传递过程，但是官方不建议在常规业务上使用，只应用于框架集成；同时还有一点需要注意，***使用setAttachment设置的隐式参数KV，在完成一次远程调用会被清空***。即多次远程调用要多次设置



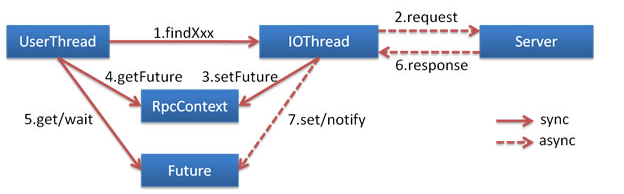
客户端设置参数



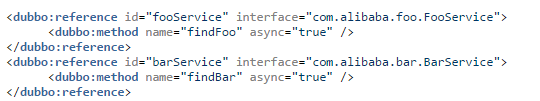
服务端接收参数

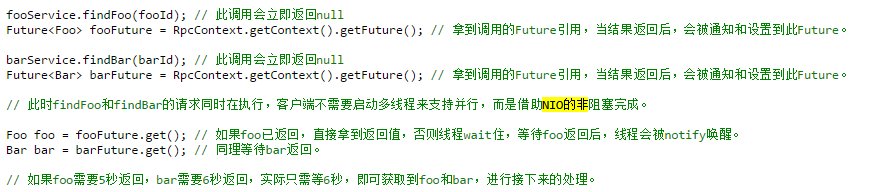
## 异步调用

附上官方设计图，可以参考，借鉴了***NIO的非阻塞***实现，不是很熟悉，需要学习一下



经测试，目前版本的异步调用模式有bug，在测试的过程中发现会报空指向异常以及调用多个不同服务返回的却是一个服务的返回结果，官方示例代码如下：





还有一点，我们一般使用异步调用的时候，总会等待异步请求返回结果继续处理后续请求；那么如果，有如下这种场景，当我们的调用不需要结果的返回时，便可以加一个参数“return=false”,表示只发送了请求，客户端不进行等待处理。

## 本地调用

本地调用，使用了Injvm协议，是一个伪协议，它不开启端口，不发起远程调用，只在JVM内直接关联，但执行Dubbo的Filter链

从 dubbo 2.2.0 开始，每个服务默认都会在本地暴露；在引用服务的时候，默认优先引用本地服务；如果希望引用远程服务可以使用一下配置强制引用远程服务

<dubbo:reference ... scope="remote" /><!--其表示远程服务引用-->

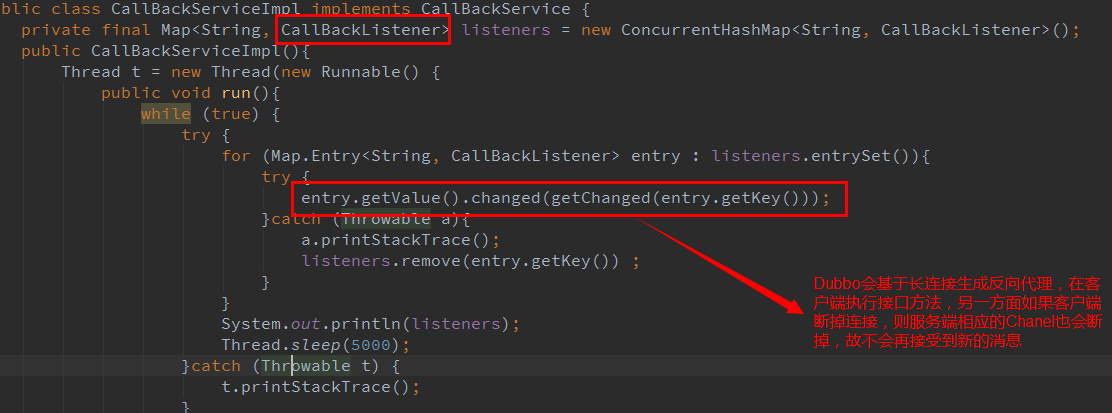
<dubbo:reference injvm="true" .../>

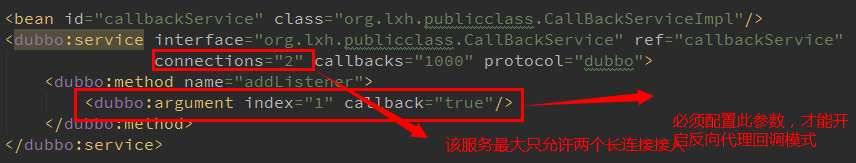
<dubbo:service injvm="true" .../>

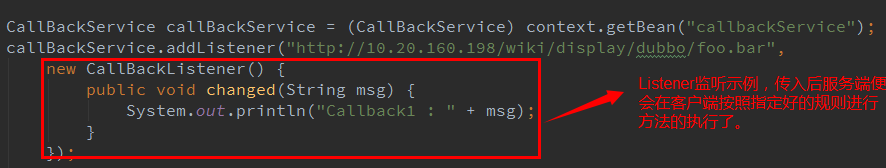
<!--这两行配置为基本配置，且在服务端和客户端二者都需要此参数的配置-->

## 参数回调

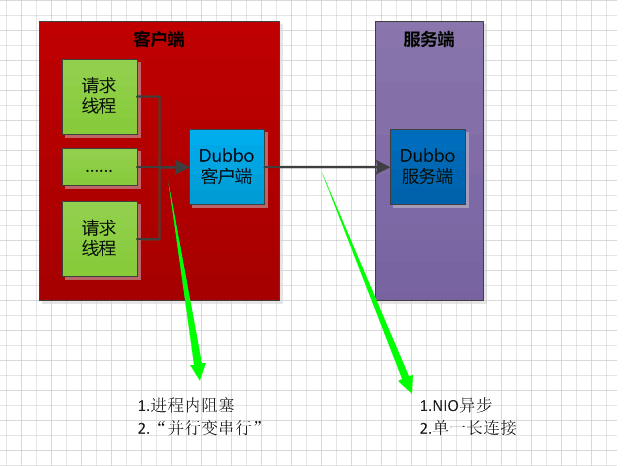
参数回调方式与调用本地callback或listener相同，只需要在Spring的配置文件中声明哪个参数是callback类型即可，Dubbo将基于长连接生成反向代理，这样就可以从***服务器端调用客户端***逻辑，即服务端在此种模式下可以完成只能再客户端完成的工作，参考示例如下：





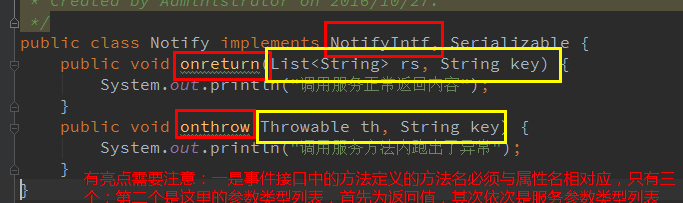


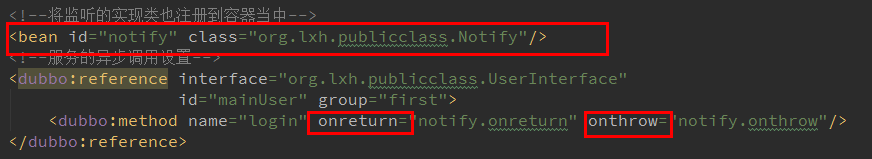
再附一张关系图



## 事件通知机制

在调用之前，调用之后，出现异常时，会触发oninvoke, onreturn, onthrow三个事件，可以配置当事件发生时，通知哪个类的哪个方法，严格按照约定格式来编码，否则程序会意外退出，示例参考如下：



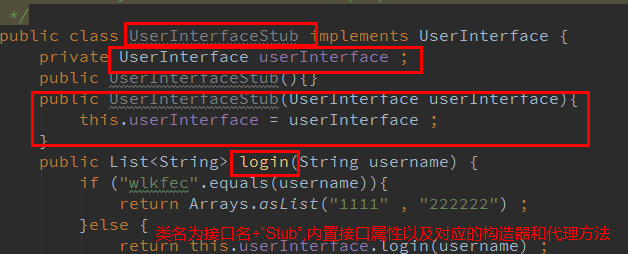


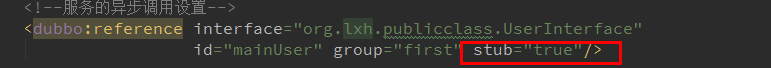
如上先将事件监听实例注册到容器中，之后在方法的声明处进行配置

此外：callback与async功能正交分解：async=true，表示结果是否马上返回.onreturn 表示是否需要回调.组合情况：(async=false 默认)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

## 本地存根

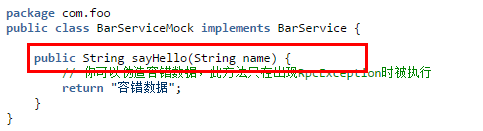
远程服务后，客户端通常只剩下接口，而实现全在服务器端，但提供方有些时候想在客户端也执行部分逻辑，比如：做ThreadLocal缓存，提前验证参数，调用失败后伪造容错数据等等，此时就需要在API中带上Stub，客户端生成Proxy实，会把Proxy通过构造函数传给Stub，然后把***Stub***暴露组给用户，Stub可以决定要不要去调Proxy【Stub必须有可传入Proxy的构造函数】，使用示例如下（客户端调用和普通方法无异）：





## 本地伪装

功能定义：Mock是Stub的一个子集，便于服务提供方在客户端执行容错逻辑，因经常需要在出现RpcException(比如网络失败，超时等)时进行容错，而在出现业务异常(比如登录用户名密码错误)时不需要容错，如果用Stub，可能就需要捕获并依赖RpcException类，而用Mock就可以不依赖RpcException，因为它的约定就是***只有出现RpcException***时才执行。示例参考如下（与本地存根部分的使用和配置相似）：（测试过程中，发现这好像是一个bug，测试了好多遍，当异常被抛出的时候，对应方法从不执行）



## 延迟暴露

如果你的服务需要Warmup时间，比如初始化缓存，等待相关资源就位等，可以使用delay进行延迟暴露，而且默认条件下所有的服务都是懒加载模式，需要时才进行实例化，这样做可以节省资源

<dubbo:service delay="5000" />

<dubbo:service delay="-1" /> 该配置表示服务暴露在Spring容器启动完成之后进行

最佳实践：

1. 强烈建议不要在服务的实现类中有applicationContext.getBean()的调用，全部采用IoC注入的方式使用Spring的Bean。

2. 如果实在要调getBean()，可以将Dubbo的配置放在Spring的最后加载。

3. 如果不想依赖配置顺序，可以使用<dubbo:provider deplay=”-1” />，使Dubbo在Spring容器初始化完后，再暴露服务。

4. 如果大量使用getBean()，相当于已经把Spring退化为工厂模式在用，可以将Dubbo的服务隔离单独的Spring容器。

## 并发控制

示例使用配置如下：





分别配置在服务端和客户端上的最大并发连接数，并发控制

## 连接控制

<dubbo:protocol name="dubbo" accepts="10" />

限制客户端连接服务端的连接个数

## 延迟连接

延迟连接，用于减少长连接数，当有调用发起时，再创建长连接

<dubbo:protocol name="dubbo" lazy="true" />

配置lazy属性，只有当请求调用的时候才会建立长连接。

## 粘滞连接

粘滞连接用于有状态服务，尽可能让客户端总是向同一提供者发起调用，除非该提供者挂了，再连另一台；粘滞连接将自动开启延迟连接，以减少长连接数

<dubbo:protocol name="dubbo" sticky="true" />

## 令牌验证

一种安全限制规则：防止消费者绕过注册中心访问提供者；在注册中心控制权限，以决定要不要下发令牌给消费者；注册中心可灵活改变授权方式，而不需修改或升级提供者

<dubbo:protocol name="dubbo" token="123456" />

在访问协议的级别上设置令牌验证

# 最佳实践

## 版本控制

<dubbo:provider version="1.0"/>

<dubbo:consumer version="1.0"/>

虽然在配置中可以针对每一个服务进行版本号的定义，但是为了管理和升级服务的方便，采用这种全局配置的方式更为方便。

## 对服务进行调优

***timeout***，调用超时时间，默认为1000毫秒，即超过1000毫秒没有返回数据，就会执行重试机制

***retries***，失败重试次数，默认为2，即失败（超时）之后的重试次数

***connections***，对每个提供者的最大链接数，默认为100，建议根据服务器配置进行调整

***loadbalance***，负载均衡策略，默认为random

***async***, 是否异步执行，默认为false

***delay***, 延迟注册服务时间，默认为0，建议不同的接口把暴露服务时间错开，避免dubbo爆端口被占用错误

## 若某一个接口服务执行时间很长

忽略返回值，配置方法的“return”参数值为“false”，这样接口方法即刻返回null，继续向下执行；将指定方法配置为异步执行，配置“async”参数为“true”,后续程序可以通过上下文对象获取Future获取返回数据；配置回调的方式，客户端向服务端传递一个需要在本地执行的回调函数，当服务端达到指定条件，便可以调用此回调函数执行（按照约定好的规则来编码）

## 对参数进行校验

默认是不开启校验模式的，需要在客户端或者服务端加入配置，“validation”参数配置为“true”，这样在需要校验的POJO类上加上验证注解（可以自定义验证规则），之后的接口调用就会对指定的类进行参数校验

# 常见的问题

## dubbo.xsd

若在XML配置相应的dubbo应用信息时，发现很多不支持的标签，需要手工在XML Catalog中添加与namespace同名的KV值，此文件在jar文件中存在，即可解决

## dubbo与spring、netty版本冲突

解决办法是移除掉低版本的Spring等库依赖

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>dubbo</artifactId>

<version>2.5.3</version>

<exclusions>

<exclusion>

<groupId>org.springframework</groupId>

<artifactId>spring</artifactId>

</exclusion>

<exclusion>

<artifactId>netty</artifactId>

<groupId>org.jboss.netty</groupId>

</exclusion>

</exclusions>

</dependency>

## 端口20880被绑定

Dubbo协议的默认端口为20880，但是在某些特定的情况下，该端口会被别的程序异常占用，解决此问题的办法为将此协议的port设置为-1，表示自动寻找可用端口号，这样就不会出现端口被异常占用的问题了

## dubbo 注解配置的不足之处

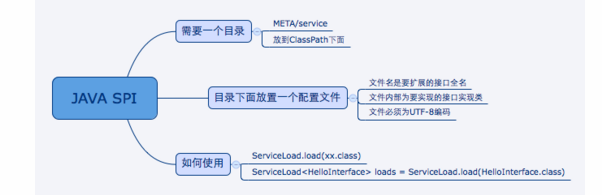
目前来说，dubbo的配置几乎完全集中在了XML配置方式，而对于Annotation的配置方式，有很多的不足，若有需要只能通过扩展源代码开发新的Annotation标记

# 源码解读

## SPI

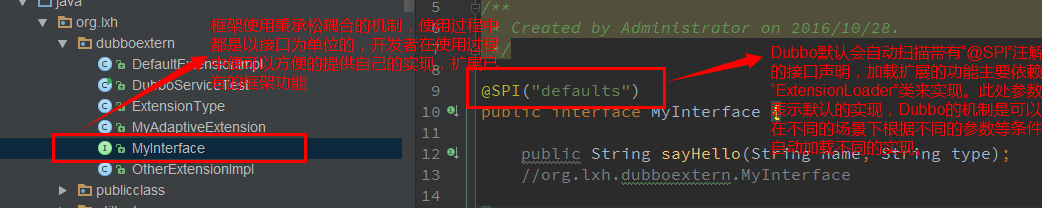
Dubbo内核的实现就是依靠SPI的机制来完成的，可以说SPI是Dubbo最核心的（没有之一）的机制

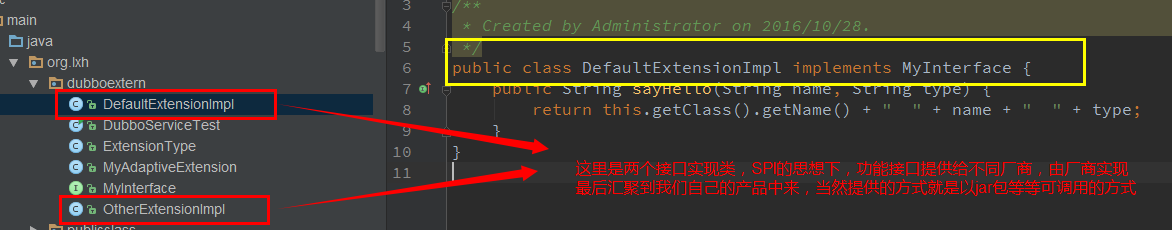
以下是Java SPI的示例：

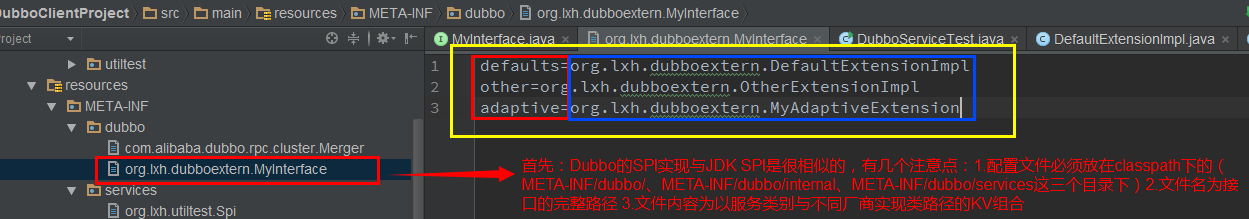


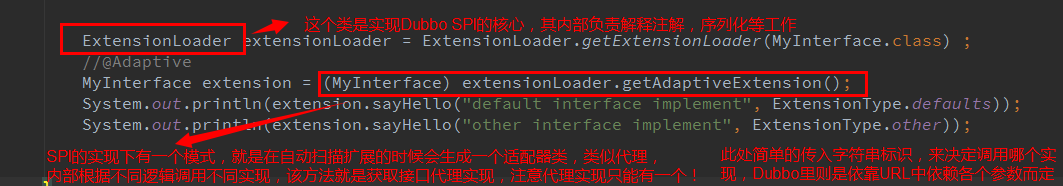
Dubbo SPI是Java SPI的升级版本，增加了若干新的特性

以下是依据SPI机制的小例子：

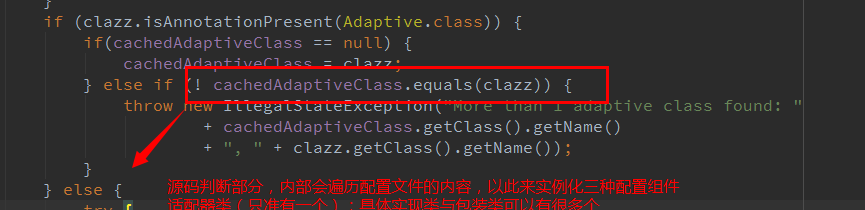












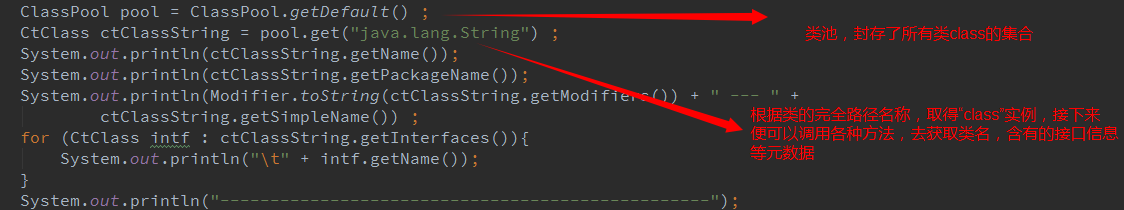


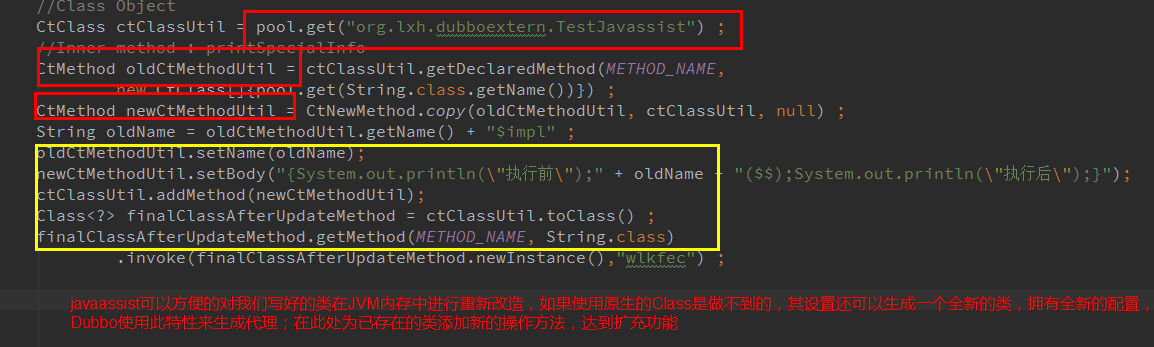
## Javassist动态编译

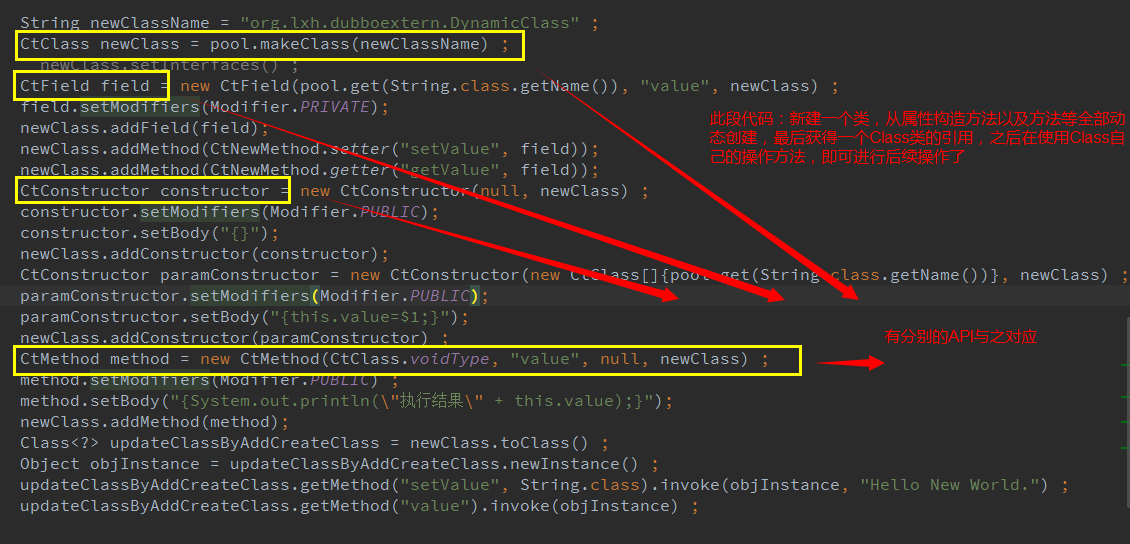
Javassist（Javassist是一个处理***Java字节码***的类库，可以直接便捷和生成Java字节码，已达到对.class文件进行动态修改的效果）（动态代理）

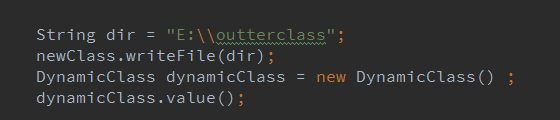
这一部分的代码感觉有点绕，没看完全，不过它的一个主要作用便是：利用Javassist工具生成SPI机制的设配用来根据统一数据模型URL中获取协议来选择使用何种策略。

以下是一些相关的示例：









如上图，当我们制定好一个类的规则时，还可以使用writeFile方法将class文件写到指定位置，默认如果不加参数，会将文件保存到根目录为项目目录的路径下，如果别的地方使用，加入classpath即可正常使用了。



# 参考资料

【官方文档】

# 重要技术

## Javassist

分析、编辑和创建Java字节码，即在Java应用程序运行时，按照要求动态生成class并且完成实例化（示例）（***主要用在RPC模块的代理层***）。

ClassPool classPool = ClassPool.getDefault() ;

CtClass Foo = classPool.makeClass("Foo") ;

CtMethod method = CtNewMethod.make("public Integer getInteger() { return null; }", Foo) ;

Foo.addMethod(method);

CtField field = new CtField(CtClass.intType, "count", Foo) ;

Foo.addField(field);

Class<?> cClass = Foo.toClass() ;

//根据指定的class字节码实例化指定的类对象。

Object instance = cClass.newInstance() ;

## 设计模式：工厂方法模式

设计模式分为三大类：创建模型、结构模型和行为模型

Dubbo广泛应用了此模型，如果后续需要引入一种新的RPC协议支持的话，只要创建该协议的具体实现和对应的创建工厂，已有的RPC协议不会受到任何影响

【使用Spring的动态代理结束可以解决，代码的解藕特性】

## RPC名词

在RPC框架中，应该会有管理RPC接口的注册，判断客户端的请求权限、控制接口实现类的执行

在代理层，一起的对消息管理等的操作，还是依赖于客户端模块中的RPC通信

RPC的消息管理层，是专门用来处理所传递消息进行添加编号以及解码操作

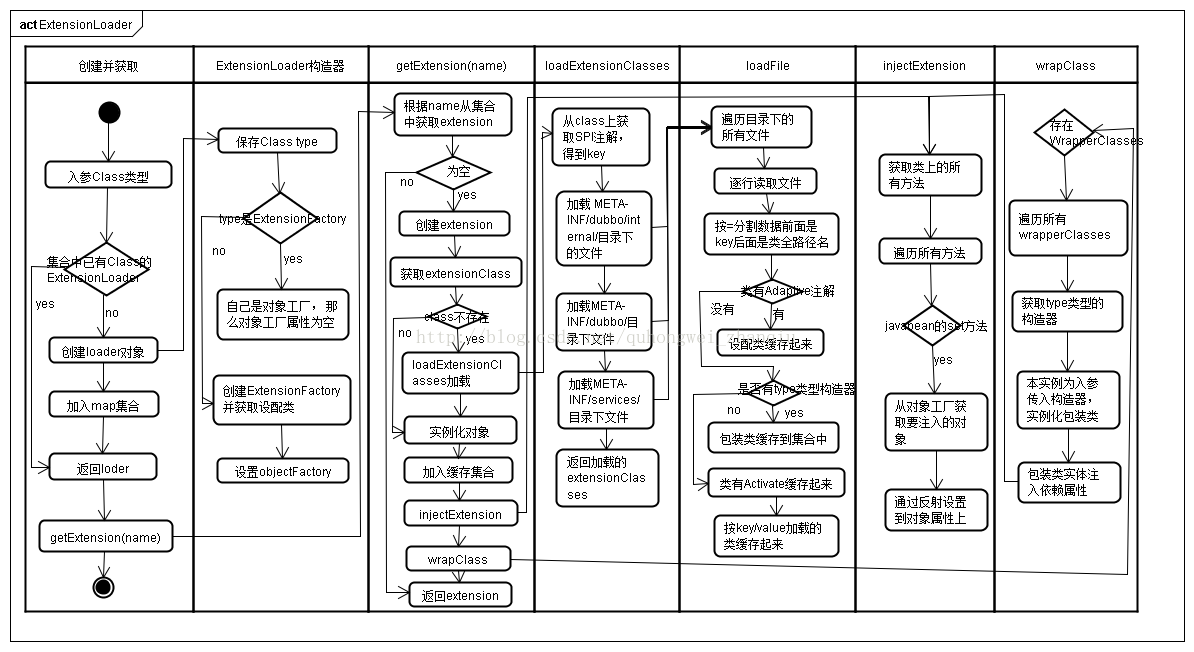
传输层需要统一RPC客户端与服务端所使用的网络IO模型

IDL是为了跨平台通信的一种描述形式

此部分涉及的网络概念与应用比较多

Java代理与动态代理

## ExtensionLoader流程图



# 框架整体设计

## 层次整体说明（大量使用了代理、Annotation以及反射等技术）：

config，配置层，对外配置接口，以ServiceConfig, 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

proxy，服务代理层，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

registry，注册中心层，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, RegistryService

cluster，路由层，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, LoadBalance

monitor，监控层，RPC调用次数和调用时间监控，以Statistics为中心，扩展接口为MonitorFactory, Monitor, MonitorService

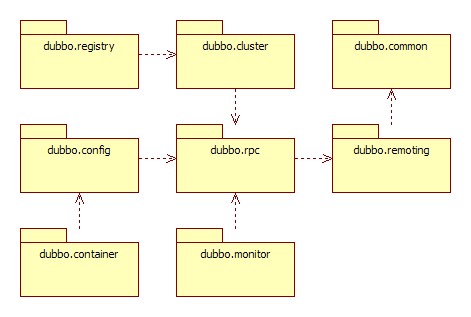
protocol，远程调用层，封将RPC调用，以Invocation, Result为中心，扩展接口为Protocol, Invoker, Exporter

exchange，信息交换层，封装请求响应模式，同步转异步，以Request, Response为中心，扩展接口为Exchanger, ExchangeChannel, ExchangeClient, ExchangeServer

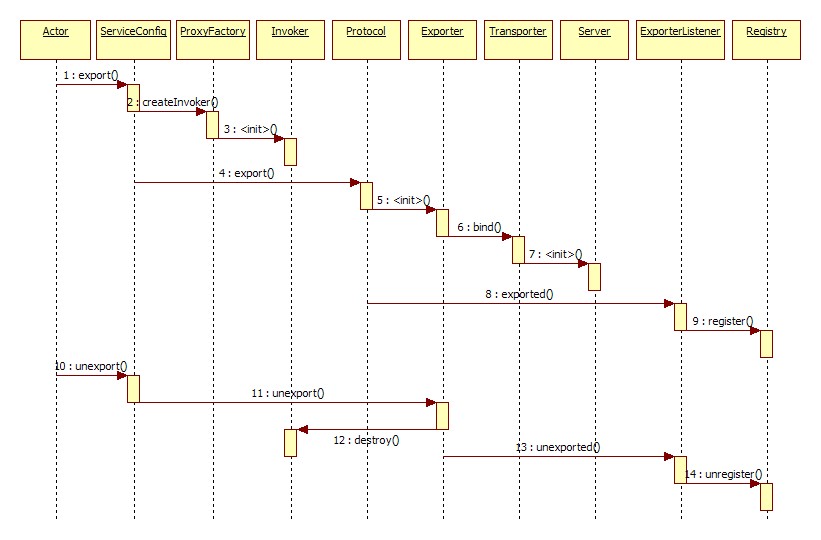
transport，网络传输层，抽象mina和netty为统一接口，以Message为中心，扩展接口为Channel, Transporter, Client, Server, Codec

serialize，数据序列化层，可复用的一些工具，扩展接口为Serialization, ObjectInput, ObjectOutput, ThreadPool

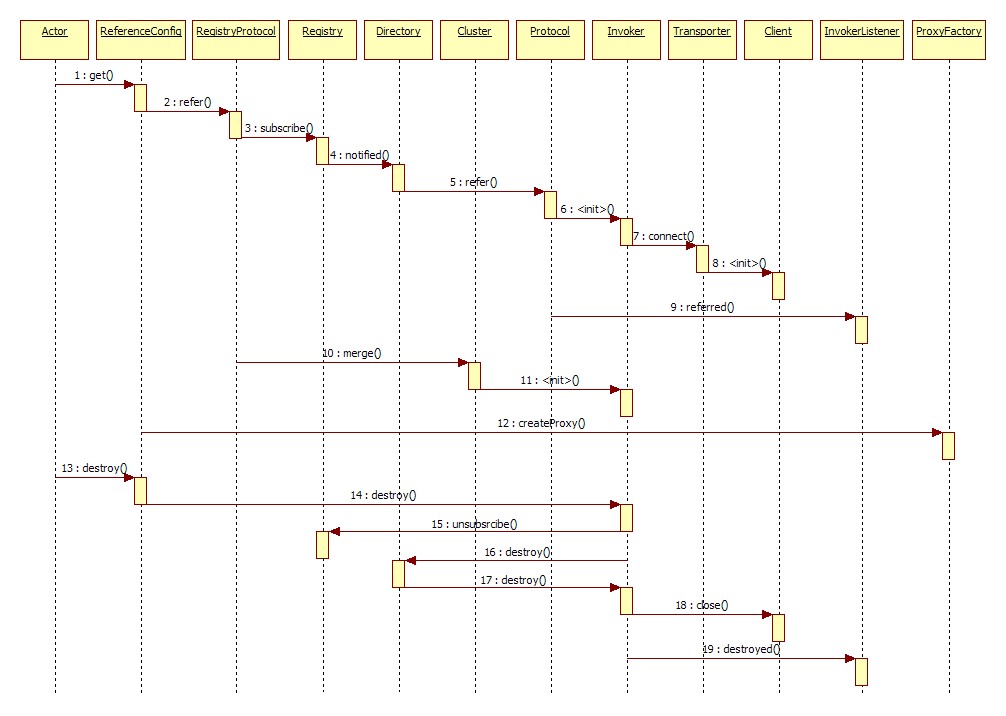
## 模块分包结构图



## 暴露服务时序图

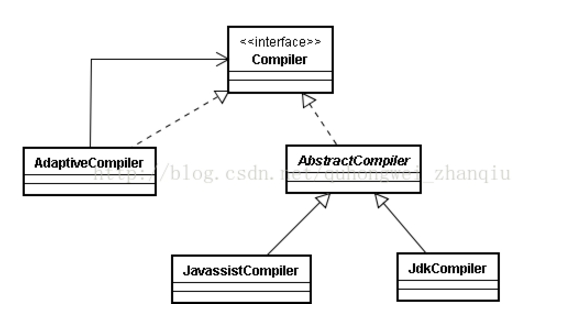


## 引用服务时序图

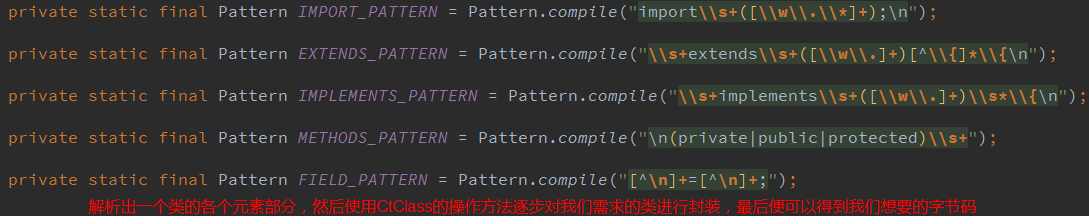


# 动态编译

首先参考以下类图:



此模式在Dubbo整个系统当中，被大量应用；首先定义SPI接口Compiler，表明它将加载扩展点的实现，左侧为一个适配器类，负责按照不同的策略调用不同的实现；右侧，引用一个抽象类来作为接口和具体实现类之间的桥梁，这样做的目的是，封装通用的一类处理方法，最后加入不同的实现类，供不同的选择，Dubbo默认使用Javassist来进行代码的动态编译，这样便可以动态生成适配类。







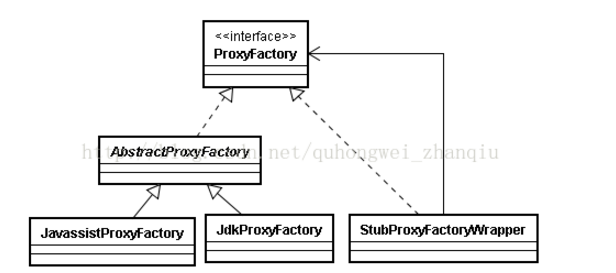






# 代理之接口定义

使用场景：Dubbo服务的消费端获取到的就是对远程服务的一个代理，代理对象由ProxyFactory创建完成（这部分没研究很明白，Java的代理机制也算是一个比较成熟的生态了），可以参考如下类图：



# 与Spring融合

在实际使用过程中，我们可以使用API或者XML配置的方式来使用Dubbo服务，不过利用XML配置方式才是我们更为推荐的方式，缘由XML配置具有诸多优点，对应的，在dubbo的命名空间下，相应的DubboNameSpaceHander会将配置文件中的不同配置项，最终都会转化为相应的API实例。

