Table of Contents

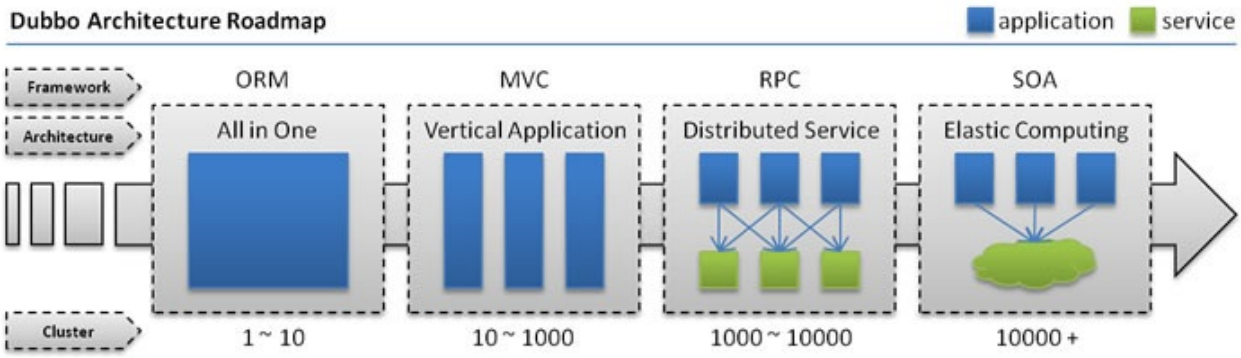
这篇文档详细讲解了 `dubbo` 的使用，基本涵盖 `dubbo` 的所有功能特性。

如果你正依赖 `dubbo` 作为你业务工程的RPC通信框架，这里可以作为你的参考手册

入门

背景

随着互联网的发展，网站应用的规模不断扩大，常规的垂直应用架构已无法应对，分布式服务架构以及流动计算架构势在必行，亟需一个治理系统确保架构有条不紊的演进。



单一应用架构

当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本。此时，用于简化增删改查工作量的数据访问框架(ORM)是关键。

垂直应用架构

当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，将应用拆成互不相干的几个应用，以提升效率。此时，用于加速前端页面开发的Web框架(MVC)是关键。

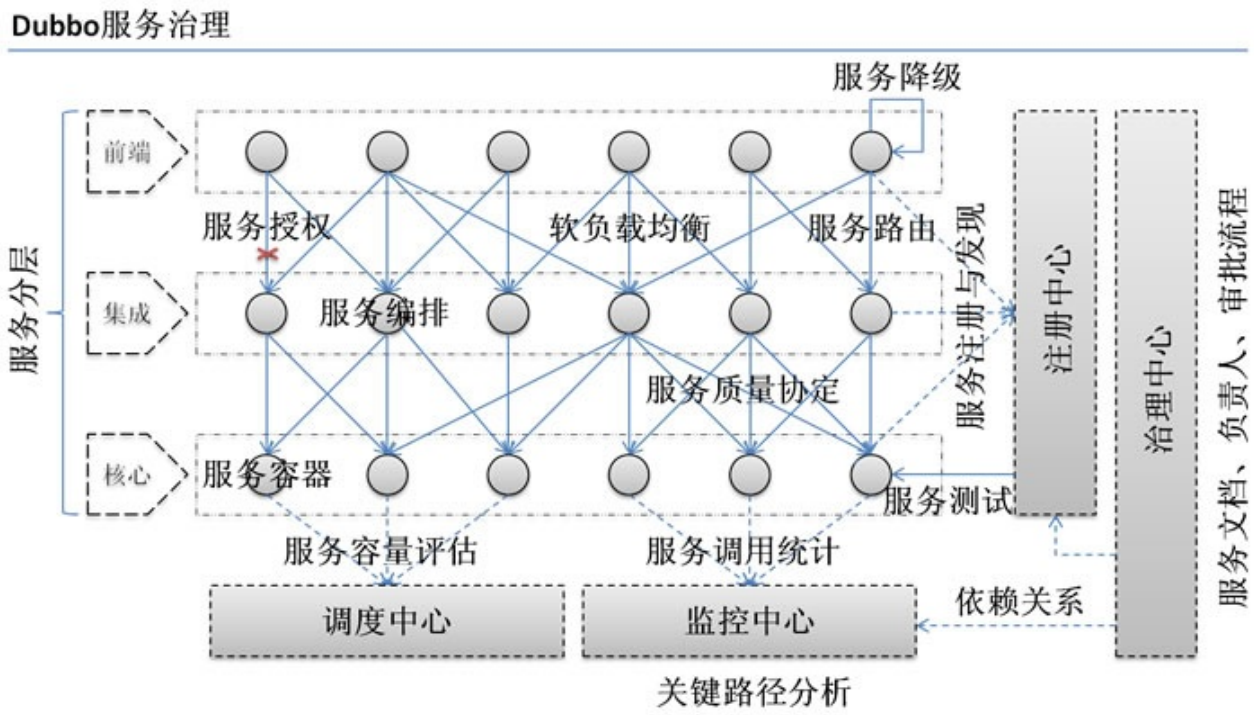
分布式服务架构

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。此时，用于提高业务复用及整合的分布式服务框架(RPC)是关键。

流动计算架构

当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。此时，用于提高机器利用率的资源调度和治理中心(SOA)是关键。

需求



在大规模服务化之前，应用可能只是通过 RMI 或 Hessian 等工具，简单的暴露和引用远程服务，通过配置服务的URL地址进行调用，通过 F5 等硬件进行负载均衡。

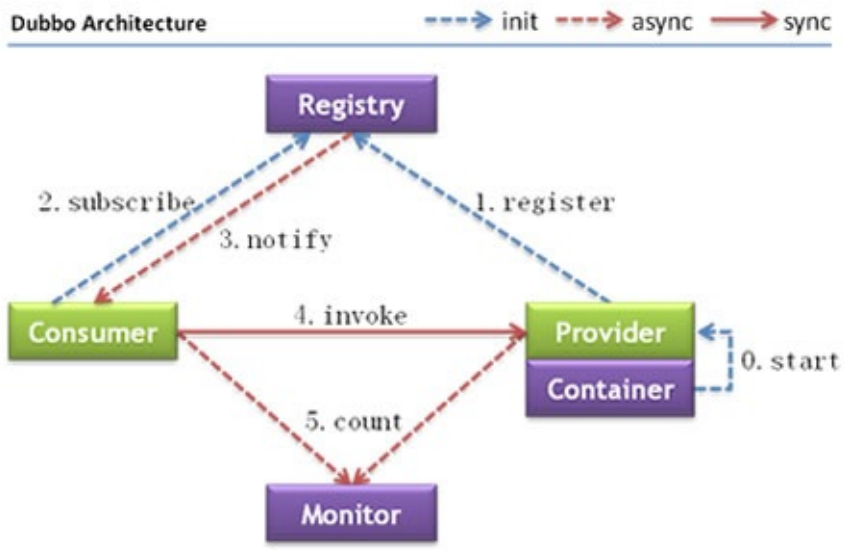
当服务越来越多时，服务 **URL** 配置管理变得非常困难，**F5** 硬件负载均衡器的单点压力也越来越大。 此时需要一个服务注册中心，动态的注册和发现服务，使服务的位置透明。并通过在消费方获取服务提供方地址列表，实现软负载均衡和 Failover，降低对 F5 硬件负载均衡器的依赖，也能减少部分成本。

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。 这时，需要自动画出应用间的依赖关系图，以帮助架构师理清理关系。

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？ 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。

以上是 Dubbo 最基本的几个需求。

架构



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`	统计服务的调用次调和调用时间的监控中心
`Container`	服务运行容器

调用关系说明

1. 服务容器负责启动，加载，运行服务提供者。
2. 服务提供者在启动时，向注册中心注册自己提供的服务。
3. 服务消费者在启动时，向注册中心订阅自己所需的服务。
4. 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。
5. 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。
6. 服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心。

Dubbo 架构具有以下几个特点，分别是连通性、健壮性、伸缩性、以及向未来架构的升级性。

连通性

- 注册中心负责服务地址的注册与查找，相当于目录服务，服务提供者和消费者只在启动时与注册中心交互，注册中心不转发请求，压力较小
- 监控中心负责统计各服务调用次数，调用时间等，统计先在内存汇总后每分钟一次发送到监控中心服务器，并以报表展示
- 服务提供者向注册中心注册其提供的服务，并汇报调用时间到监控中心，此时间不包含网络开销
- 服务消费者向注册中心获取服务提供者地址列表，并根据负载算法直接调用提供者，同时汇报调用时间到监控中心，此时间包含网络开销
- 注册中心，服务提供者，服务消费者三者之间均为长连接，监控中心除外
- 注册中心通过长连接感知服务提供者的存在，服务提供者宕机，注册中心将立即推送事件通知消费者
- 注册中心和监控中心全部宕机，不影响已运行的提供者和消费者，消费者在本地缓存了提供者列表
- 注册中心和监控中心都是可选的，服务消费者可以直连服务提供者

健状性

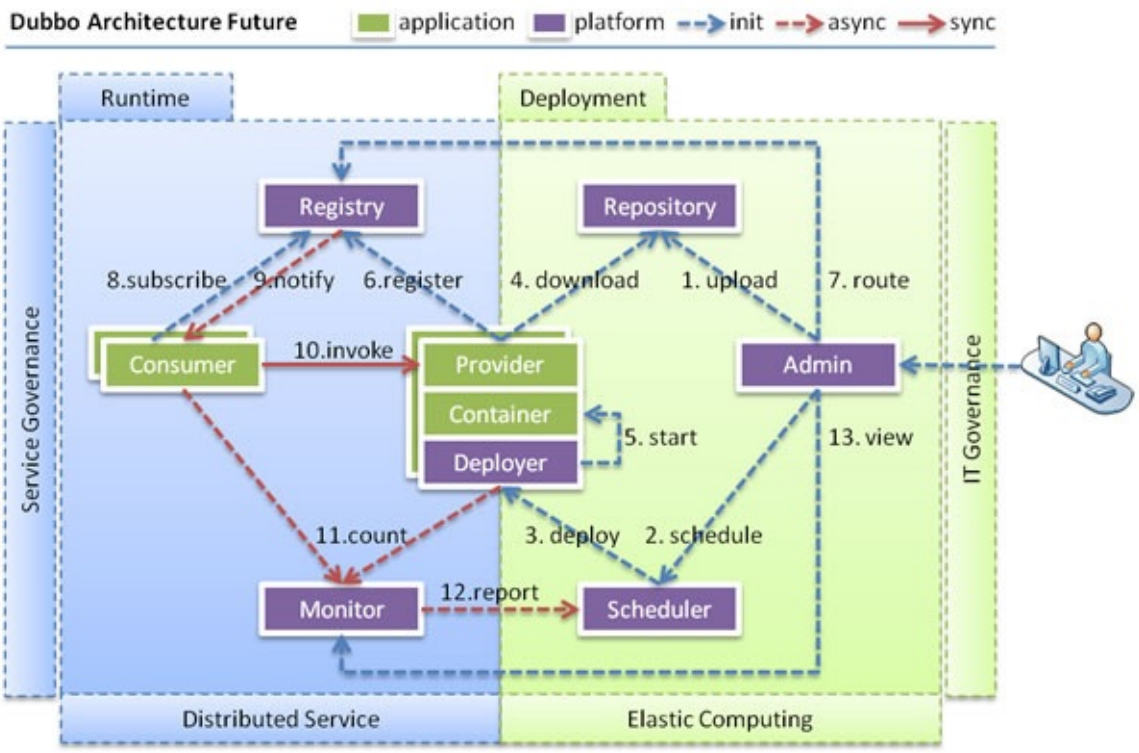
- 监控中心宕掉不影响使用，只是丢失部分采样数据
- 数据库宕掉后，注册中心仍能通过缓存提供服务列表查询，但不能注册新服务
- 注册中心对等集群，任意一台宕掉后，将自动切换到另一台
- 注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存通讯
- 服务提供者无状态，任意一台宕掉后，不影响使用
- 服务提供者全部宕掉后，服务消费者应用将无法使用，并无限次重连等待服务提供者恢复

伸缩性

- 注册中心为对等集群，可动态增加机器部署实例，所有客户端将自动发现新的注册中心
- 服务提供者无状态，可动态增加机器部署实例，注册中心将推送新的服务提供者信息给消费者

升级性

当服务集群规模进一步扩大，带动IT治理结构进一步升级，需要实现动态部署，进行流动计算，现有分布式服务架构不会带来阻力。下图是未来可能的一种架构：



节点角色说明

节点	角色说明
`Deployer`	自动部署服务的本地代理
`Repository`	仓库用于存储服务应用发布包
`Scheduler`	调度中心基于访问压力自动增减服务提供者
`Admin`	统一管理控制台
`Registry`	服务注册与发现的注册中心
`Monitor`	统计服务的调用次调和调用时间的监控中心

用法

本地服务 Spring 配置

local.xml:

```
<bean id="xxxService" class="com.xxx.XxxServiceImpl" />
<bean id="xxxAction" class="com.xxx.XxxAction">
    <property name="xxxService" ref="xxxService" />
</bean>
```

远程服务 Spring 配置

在本地服务的基础上，只需做简单配置，即可完成远程化：

- 将上面的 `local.xml` 配置拆分成两份，将服务定义部分放在服务提供方 `remote-provider.xml` ，将服务引用部分放在服务消费方 `remote-consumer.xml` 。
- 并在提供方增加暴露服务配置 `<dubbo:service>` ，在消费方增加引用服务配置 `<dubbo:reference>` 。

remote-provider.xml:

```
<!-- 和本地服务一样实现远程服务 -->
<bean id="xxxService" class="com.xxx.XxxServiceImpl" />
<!-- 增加暴露远程服务配置 -->
<dubbo:service interface="com.xxx.XxxService" ref="xxxService" />
```

remote-consumer.xml:

```
<!-- 增加引用远程服务配置 -->
<dubbo:reference id="xxxService" interface="com.xxx.XxxService" />
<!-- 和本地服务一样使用远程服务 -->
<bean id="xxxAction" class="com.xxx.XxxAction">
    <property name="xxxService" ref="xxxService" />
</bean>
```

快速启动

Dubbo 采用全 Spring 配置方式，透明化接入应用，对应用没有任何 API 侵入，只需用 Spring 加载 Dubbo 的配置即可，Dubbo 基于 Spring 的 Schema 扩展进行加载。

如果不想使用 Spring 配置，可以通过 API 的方式 进行调用。

服务提供者

完整安装步骤，请参见：示例提供者安装

定义服务接口

DemoService.java [1]：

```
package com.alibaba.dubbo.demo;

public interface DemoService {
    String sayHello(String name);
}
```

在服务提供方实现接口

DemoServiceImpl.java [2]：

```


package com.alibaba.dubbo.demo.provider;

import com.alibaba.dubbo.demo.DemoService;

public class DemoServiceImpl implements DemoService {
    public String sayHello(String name) {
        return "Hello " + name;
    }
}
```

用 **Spring** 配置声明暴露服务

provider.xml：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
    xsi:schemaLocation="http://www.springframework.org/schema/beans        http://www.
springframework.org/schema/beans/spring-beans.xsd        http://code.alibabatech.com/s
chema/dubbo        http://code.alibabatech.com/schema/dubbo/dubbo.xsd">

    <!-- 提供方应用信息，用于计算依赖关系 -->
    <dubbo:application name="hello-world-app"  />

    <!-- 使用multicast广播注册中心暴露服务地址 -->
    <dubbo:registry address="multicast://224.5.6.7:1234" />

    <!-- 用dubbo协议在20880端口暴露服务 -->
    <dubbo:protocol name="dubbo" port="20880" />

    <!-- 声明需要暴露的服务接口 -->
    <dubbo:service interface="com.alibaba.dubbo.demo.DemoService" ref="demoService" />

    <!-- 和本地bean一样实现服务 -->
    <bean id="demoService" class="com.alibaba.dubbo.demo.provider.DemoServiceImpl" />
</beans>
```

加载 Spring 配置

Provider.java：

```
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Provider {
    public static void main(String[] args) throws Exception {
        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(new
 String[] {"http://10.20.160.198/wiki/display/dubbo/provider.xml"});
        context.start();
        System.in.read(); // 按任意键退出
    }
}
```

服务消费者

完整安装步骤，请参见：示例消费者安装

通过 Spring 配置引用远程服务

consumer.xml：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
    xsi:schemaLocation="http://www.springframework.org/schema/beans        http://www.
springframework.org/schema/beans/spring-beans.xsd        http://code.alibabatech.com/s
chema/dubbo        http://code.alibabatech.com/schema/dubbo/dubbo.xsd">

    <!-- 消费方应用名，用于计算依赖关系，不是匹配条件，不要与提供方一样 -->
    <dubbo:application name="consumer-of-helloworld-app"  />

    <!-- 使用multicast广播注册中心暴露发现服务地址 -->
    <dubbo:registry address="multicast://224.5.6.7:1234" />

    <!-- 生成远程服务代理，可以和本地bean一样使用demoService -->
    <dubbo:reference id="demoService" interface="com.alibaba.dubbo.demo.DemoService" />

</beans>
```

加载Spring配置，并调用远程服务

Consumer.java [3]：

```
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.alibaba.dubbo.demo.DemoService;

public class Consumer {
    public static void main(String[] args) throws Exception {
        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(new
 String[] {"http://10.20.160.198/wiki/display/dubbo/consumer.xml"});
        context.start();
        DemoService demoService = (DemoService)context.getBean("demoService"); // 获取
远程服务代理
        String hello = demoService.sayHello("world"); // 执行远程方法
        System.out.println( hello ); // 显示调用结果
    }
}
```

1. 该接口需单独打包，在服务提供方和消费方共享 ↩

2. 对服务消费方隐藏实现 ↩

3. 也可以使用 IoC 注入 ↩

依赖

必须依赖

JDK 1.5+ [1]

缺省依赖

通过 `mvn dependency:tree > dep.log` 命令分析，Dubbo 缺省依赖以下三方库：

```
[INFO] +- com.alibaba:dubbo:jar:2.1.2:compile
[INFO] |  +- log4j:log4j:jar:1.2.16:compile
[INFO] |  +- org.javassist:javassist:jar:3.15.0-GA:compile
[INFO] |  +- org.springframework:spring:jar:2.5.6.SEC03:compile
[INFO] |  +- commons-logging:commons-logging:jar:1.1.1:compile
[INFO] |  \- org.jboss.netty:netty:jar:3.2.5.Final:compile
```

这里所有依赖都是按照 Dubbo 缺省配置选的，这些缺省值是基于稳定性和性能考虑的。

- log4j.jar 和 commons-logging.jar [2]: 可以直接去掉，dubbo 本身的日志会自动切换为 JDK 的 java.util.logging 输出。但如果其它三方库比如 spring.jar 间接依赖 commons-logging，则不能去掉。
- javassist.jar [3]: 如果 `<dubbo:provider proxy="jdk" />` 或 `<dubbo:consumer proxy="jdk" />`，以及 `<dubbo:application compiler="jdk" />`，则不需要。
- spring.jar [4]: 如果用 `ServiceConfig` 和 `ReferenceConfig` 的 API 调用，则不需要。
- netty.jar [5]: 如果 `<dubbo:protocol server="mina"/>` 或 `<dubbo:protocol server="grizzly"/>`，则换成 mina.jar 或 grizzly.jar。如果 `<protocol name="rmi"/>`，则不需要。

可选依赖

以下依赖，在主动配置使用相应实现策略时用到，需自行加入依赖。

- mina: 1.1.7
- grizzly: 2.1.4
- httpclient: 4.1.2
- hessian_lite: 3.2.1-fixed
- xstream: 1.4.1

- fastjson: 1.1.8
- zookeeper: 3.3.3
- jedis: 2.0.0
- xmemcached: 1.3.6
- jfreechart: 1.0.13
- hessian: 4.0.7
- jetty: 6.1.26
- hibernate-validator: 4.2.0.Final
- zkclient: 0.1
- curator: 1.1.10
- cxf: 2.6.1
- thrift: 0.8.0
- servlet: 2.5 [6]
- bsf: 3.1 [6]
- validation-api: 1.0.0.GA [6]
- jcache: 0.4 [6]

> 1. 理论上 Dubbo 可以只依赖 JDK，不依赖于任何三方库运行，只需配置使用 JDK 相关实现策略 ↩
>
> 2. 日志输出包 ↩
>
> 3. 字节码生成 ↩
>
> 4. 配置解析 ↩
>
> 5. 网络传输 ↩
>
> 6. JEE ↩

成熟度

功能成熟度

Feature	Maturity	Strength	Problem	Advise	User
并发控制	Tested	并发控制		试用	
连接控制	Tested	连接数控制		试用	
直连提供者	Tested	点对点直连服务提供方，用于测试		测试环境使用	Alibaba
分组聚合	Tested	分组聚合返回值，用于菜单聚合等服务	特殊场景使用	可用于生产环境	
参数验证	Tested	参数验证，JSR303验证框架集成	对性能有影响	试用	LaiWang
结果缓存	Tested	结果缓存，用于加速请求		试用	
泛化引用	Stable	泛化调用，无需业务接口类进行远程调用，用于测试平台，开放网关桥接等		可用于生产环境	Alibaba
泛化实现	Stable	泛化实现，无需业务接口类实现任意接口，用于Mock平台		可用于生产环境	Alibaba
回声测试	Tested	回声测试		试用	
隐式传参	Stable	附加参数		可用于生产环境	
异步调用	Tested	不可靠异步调用		试用	
本地调用	Tested	本地调用		试用	
参数回调	Tested	参数回调	特殊场景使用	试用	Registry
事件通知	Tested	事件通知，在远程调用执行前后触发		试用	

本地存根	Stable	在客户端执行部分逻辑		可用于生产环境	Alibaba
本地伪装	Stable	伪造返回结果，可在失败时执行，或直接执行，用于服务降级	需注册中心支持	可用于生产环境	Alibaba
延迟暴露	Stable	延迟暴露服务，用于等待应用加载warmup数据，或等待spring加载完成		可用于生产环境	Alibaba
延迟连接	Tested	延迟建立连接，调用时建立		试用	Registry
粘滞连接	Tested	粘滞连接，总是向同一个提供方发起请求，除非此提供方挂掉，再切换到另一台		试用	Registry
令牌验证	Tested	令牌验证，用于服务授权	需注册中心支持	试用	
路由规则	Tested	动态决定调用关系	需注册中心支持	试用	
配置规则	Tested	动态下发配置，实现功能的开关	需注册中心支持	试用	
访问日志	Tested	访问日志，用于记录调用信息	本地存储，影响性能，受磁盘大小限制	试用	
分布式事务	Research	JTA/XA三阶段提交事务	不稳定	不可用	

策略成熟度

Feature	Maturity	Strength	Problem	Advise	Use
Zookeeper注册中心	Stable	支持基于网络的集群方式，有广泛周边开源产品，建议使用dubbo-2.3.3以上版本（推荐使用）	依赖于Zookeeper的稳定性	可用于生产环境	
Redis注册中心	Stable	支持基于客户端双写的集群方式，性能高	要求服务器时间同步，用于检查心跳过期脏数据	可用于生产环境	

Multicast注册中心	Tested	去中心化，不需要安装注册中心	依赖于网络拓普和路由，跨机房有风险	小规模应用或开发测试环境	
Simple注册中心	Tested	Dogfooding，注册中心本身也是一个标准的RPC服务	没有集群支持，可能单点故障	试用	
Feature	Maturity	Strength	Problem	Advise	User
Simple监控中心	Stable	支持JFreeChart统计报表	没有集群支持，可能单点故障，但故障后不影响RPC运行	可用于生产环境	
Feature	Maturity	Strength	Problem	Advise	User
Dubbo协议	Stable	采用NIO复用单一长连接，并使用线程池并发处理请求，减少握手和加大并发效率，性能较好（推荐使用）	在大文件传输时，单一连接会成为瓶颈	可用于生产环境	Aliba
Rmi协议	Stable	可与原生RMI互操作，基于TCP协议	偶尔会连接失败，需重建Stub	可用于生产环境	Aliba
Hessian协议	Stable	可与原生Hessian互操作，基于HTTP协议	需hessian.jar支持，http短连接的开销大	可用于生产环境	
Feature	Maturity	Strength	Problem	Advise	User
Netty Transporter	Stable	JBoss的NIO框架，性能较好（推荐使用）	一次请求派发两种事件，需屏蔽无用事件	可用于生产环境	Aliba
Mina Transporter	Stable	老牌NIO框架，稳定	待发送消息队列派发不及时，大压力下，会出现FullGC	可用于生产环境	Aliba
Grizzly Transporter	Tested	Sun的NIO框架，应用于GlassFish服务器中	线程池不可扩展，Filter不能拦截下一Filter	试用	
Feature	Maturity	Strength	Problem	Advise	User
Hessian Serialization	Stable	性能较好，多语言支持（推	Hessian的各版本兼容性不好，可能和应用使用的Hessian冲	可用于生产环	Aliba

Serialization		荐使用）	突，Dubbo内嵌了hessian3.2.1的源码	境	
Dubbo Serialization	Tested	通过不传送POJO的类元信息，在大量POJO传输时，性能较好	当参数对象增加字段时，需外部文件声明	试用	
Json Serialization	Tested	纯文本，可跨语言解析，缺省采用FastJson解析	性能较差	试用	
Java Serialization	Stable	Java原生支持	性能较差	可用于生产环境	
Feature	Maturity	Strength	Problem	Advise	User
Javassist ProxyFactory	Stable	通过字节码生成代替反射，性能比较好（推荐使用）	依赖于javassist.jar包，占用JVM的Perm内存，Perm可能要设大一些：java -XX:PermSize=128m	可用于生产环境	Aliba
Jdk ProxyFactory	Stable	JDK原生支持	性能较差	可用于生产环境	
Feature	Maturity	Strength	Problem	Advise	User
Failover Cluster	Stable	失败自动切换，当出现失败，重试其它服务器，通常用于读操作（推荐使用）	重试会带来更长延迟	可用于生产环境	Aliba
Failfast Cluster	Stable	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	如果有机器正在重启，可能会出现调用失败	可用于生产环境	Aliba
Failsafe Cluster	Stable	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	调用信息丢失	可用于生产环境	Moni
Failback		失败自动恢复，后台记录失败请求，定		可用于	

Cluster		时重发，通常用于消息通知操作		境	
Forking Cluster	Tested	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	需要浪费更多服务资源	可用于生产环境	
Broadcast Cluster	Tested	广播调用所有提供者，逐个调用，任意一台报错则报错，通常用于更新提供方本地状态	速度慢，任意一台报错则报错	可用于生产环境	
Feature	Maturity	Strength	Problem	Advise	User
Random LoadBalance	Stable	随机，按权重设置随机概率（推荐使用）	在一个截面上碰撞的概率高，重试时，可能出现瞬间压力不均	可用于生产环境	Aliba
RoundRobin LoadBalance	Stable	轮循，按公约后的权重设置轮循比率	存在慢的机器累积请求问题，极端情况可能产生雪崩	可用于生产环境	
LeastActive LoadBalance	Stable	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，使慢的机器收到更少请求	不支持权重，在容量规划时，不能通过权重把压力导向一台机器压测容量	可用于生产环境	
ConsistentHash LoadBalance	Stable	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	压力分摊不均	可用于生产环境	
Feature	Maturity	Strength	Problem	Advise	User
条件路由规则	Stable	基于条件表达式的路由规则，功能简单	有些复杂多分支条件情况，规则很难描述	可用于生产环	Aliba

		易用			
脚本路由规则	Tested	基于脚本引擎的路由规则，功能强大	没有运行沙箱，脚本能力过于强大，可能成为后门	试用	
Feature	Maturity	Strength	Problem	Advise	User
Spring Container	Stable	自动加载META-INF/spring目录下的所有Spring配置		可用于生产环境	Aliba
Jetty Container	Stable	启动一个内嵌Jetty，用于汇报状态	大量访问页面时，会影响服务器的线程和内存	可用于生产环境	Aliba
Log4j Container	Stable	自动配置log4j的配置，在多进程启动时，自动给日志文件按进程分目录	用户不能控制log4j的配置，不灵活	可用于生产环境	Aliba

配置

XML 配置

有关 XML 的详细配置项，请参见：配置参考手册。如果不想使用 Spring 配置，而希望通过 API 的方式进行调用，请参见：API配置。想知道如何使用配置，请参见：快速启动。

provider.xml 示例

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springf
ramework.org/schema/beans/spring-beans.xsd http://code.alibabatech.com/schema/dubbo ht
tp://code.alibabatech.com/schema/dubbo/dubbo.xsd">
    <dubbo:application name="hello-world-app"  />
    <dubbo:registry address="multicast://224.5.6.7:1234" />
    <dubbo:protocol name="dubbo" port="20880" />
    <dubbo:service interface="com.alibaba.dubbo.demo.DemoService" ref="demoServiceLoca
l" />
    <dubbo:reference id="demoServiceRemote" interface="com.alibaba.dubbo.demo.DemoServ
ice" />
</beans>
```

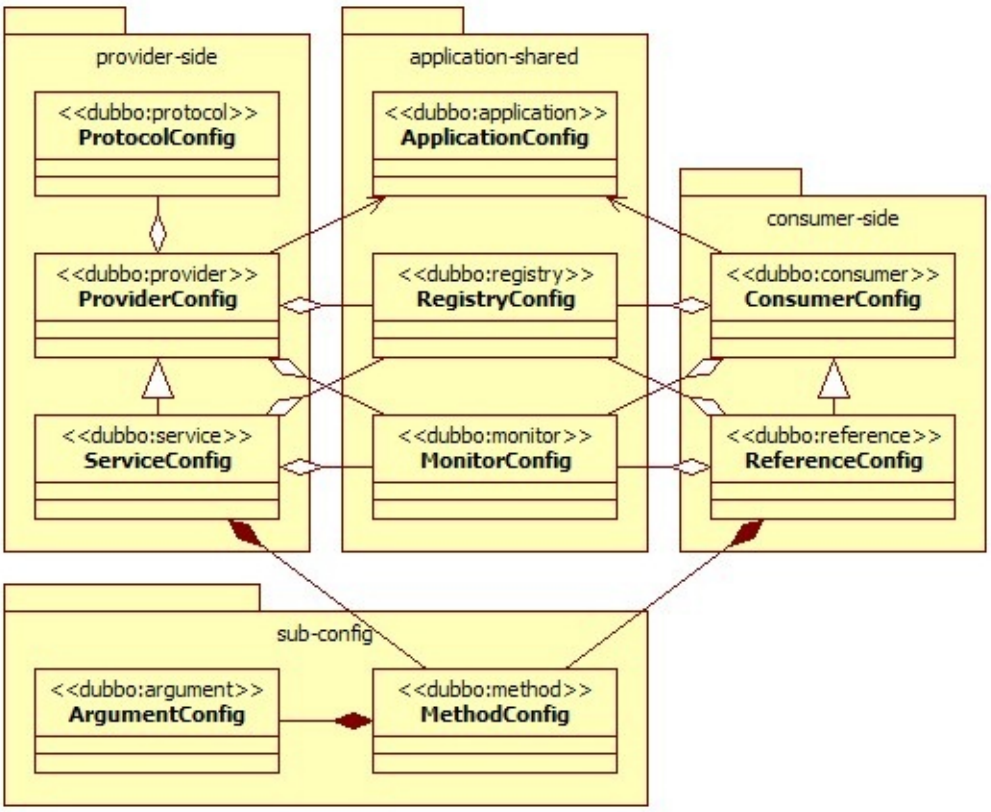
所有标签都支持自定义参数，用于不同扩展点实现的特殊配置，如：

```
<dubbo:protocol name="jms">
    <dubbo:parameter key="queue" value="your_queue" />
</dubbo:protocol>
```

或：[1]

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
    xmlns:p="http://www.springframework.org/schema/p"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springf
ramework.org/schema/beans/spring-beans.xsd http://code.alibabatech.com/schema/dubbo ht
tp://code.alibabatech.com/schema/dubbo/dubbo.xsd">
    <dubbo:protocol name="jms" p:queue="your_queue" />
</beans>
```

配置之间的关系


provider-side
<<dubbo:protocol>>
ProtocolConfig
<<dubbo:provider>>
ProviderConfig
<<dubbo:service>>
ServiceConfig
application-shared
<<dubbo:application>>
ApplicationConfig
<<dubbo:registry>>
RegistryConfig
<<dubbo:monitor>>
MonitorConfig
consumer-side
<<dubbo:consumer>>
ConsumerConfig
<<dubbo:reference>>
ReferenceConfig
sub-config
<<dubbo:argument>>
ArgumentConfig
<<dubbo:method>>
MethodConfig

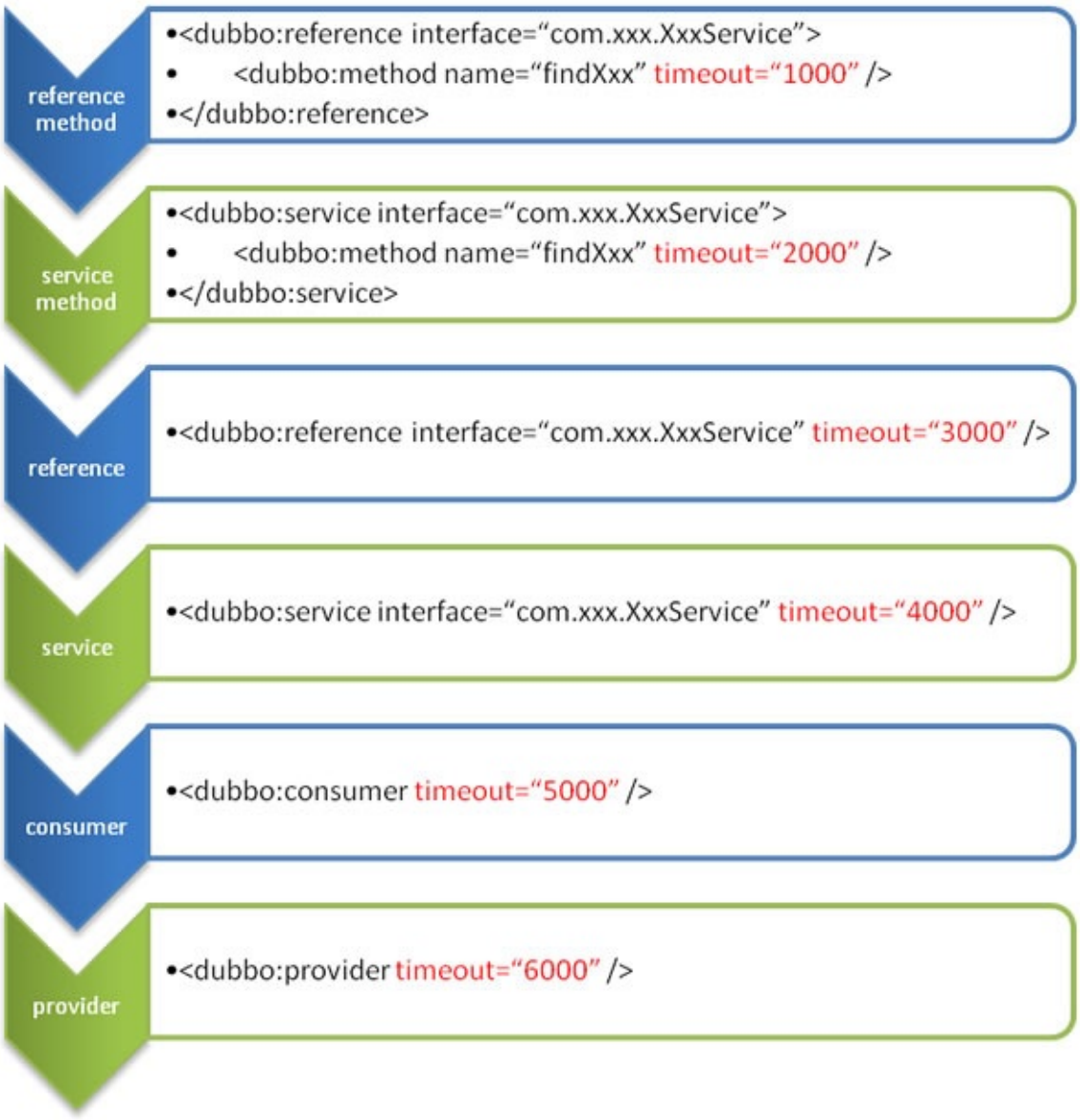
标签	用途	解释
`<dubbo:service/>`	服务配置	用于暴露一个服务，定义服务的元信息，一个服务可以用多个协议暴露，一个服务也可以注册到多个注册中心
`<dubbo:reference/>` 2	引用配置	用于创建一个远程服务代理，一个引用可以指向多个注册中心
`<dubbo:protocol/>`	协议配置	用于配置提供服务的协议信息，协议由提供方指定，消费方被动接受
`<dubbo:application/>`	应用配置	用于配置当前应用信息，不管该应用是提供者还是消费者
`<dubbo:module/>`	模块配置	用于配置当前模块信息，可选
`<dubbo:registry/>`	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/>`	提供方配置	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/>`	参数配置	用于指定方法参数配置

配置覆盖关系

以 timeout 为例，显示了配置的查找顺序，其它 retries, loadbalance, actives 等类似：

- 方法级优先，接口级次之，全局配置再次之。
- 如果级别一样，则消费方优先，提供方次之。

其中，服务提供方配置，通过 URL 经由注册中心传递给消费方。



建议由服务提供方设置超时，因为一个方法需要执行多长时间，服务提供方更清楚，如果一个消费方同时引用多个服务，就不需要关心每个服务的超时设置。

理论上 ReferenceConfig 的非服务标识配置，在 ConsumerConfig，ServiceConfig, ProviderConfig 均可以缺省配置。

> 1. `2.1.0` 开始支持，注意声明：`xmlns:p="http://www.springframework.org/schema/p"` ↩
>
> 2. 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。如果需要饥饿加载，即没有人引用也立即生成动态代理，可以配置：`<dubbo:reference ... init="true" />` ↩

属性配置

如果公共配置很简单，没有多注册中心，多协议等情况，或者想多个 Spring 容器想共享配置，可以使用 dubbo.properties 作为缺省配置。

Dubbo 将自动加载 classpath 根目录下的 dubbo.properties，可以通过JVM启动参数 `-Ddubbo.properties.file=xxx.properties` 改变缺省配置位置。[1]

映射规则

将 XML 配置的标签名，加属性名，用点分隔，多个属性拆成多行

- 比如：`dubbo.application.name=foo` 等价于 `<dubbo:application name="foo" />`
- 比如：`dubbo.registry.address=10.20.153.10:9090` 等价于 `<dubbo:registry address="10.20.153.10:9090" />`

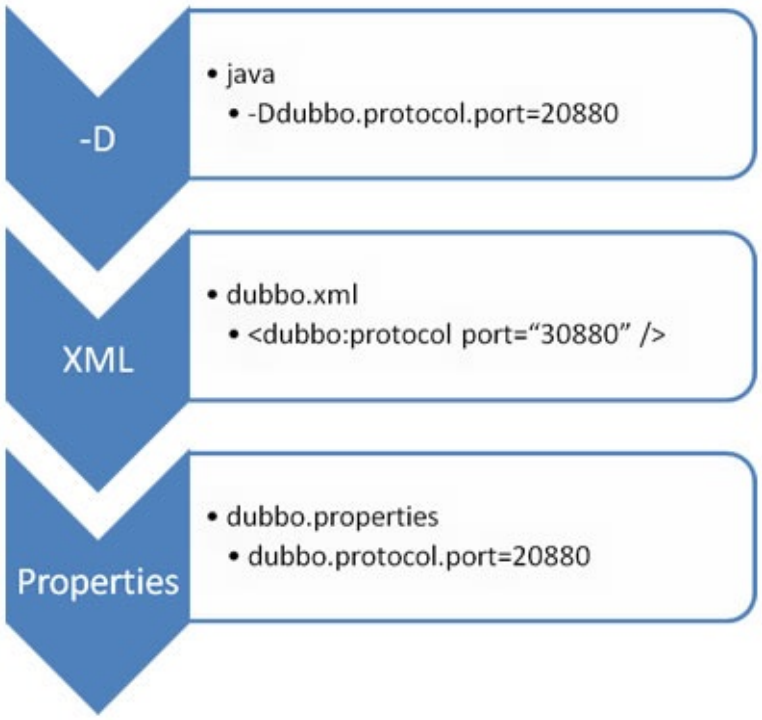
如果 XML 有多行同名标签配置，可用 id 号区分，如果没有 id 号将对所有同名标签生效

- 比如：`dubbo.protocol.rmi.port=1234` 等价于 `<dubbo:protocol id="rmi" name="rmi" port="1099" />` [2]
- 比如：`dubbo.registry.china.address=10.20.153.10:9090` 等价于 `<dubbo:registry id="china" address="10.20.153.10:9090" />`

下面是 dubbo.properties 的一个典型配置：

```
dubbo.application.name=foo
dubbo.application.owner=bar
dubbo.registry.address=10.20.153.10:9090
```

覆盖策略



JVM 启动 -D 参数优先，这样可以使用户在部署和启动时进行参数重写，比如在启动时需改变协议的端口。

XML 次之，如果在 XML 中有配置，则 dubbo.properties 中的相应配置项无效。

Properties 最后，相当于缺省值，只有 XML 没有配置时，dubbo.properties 的相应配置项才会生效，通常用于共享公共配置，比如应用名。

> [1]. 如果 classpath 根目录下存在多个 dubbo.properties，比如多个 jar 包中有 dubbo.properties，Dubbo 会任意加载，并打印 Error 日志，后续可能改为抛异常。 ↩

> [2]. 协议的 id 没配时，缺省使用协议名作为 id ↩

API 配置

API 属性与配置项一对一，各属性含义，请参见：配置参考手册，比如：`ApplicationConfig.setName("xxx")` 对应 `<dubbo:application name="xxx" />`[1]

服务提供者

```
import com.alibaba.dubbo.rpc.config.ApplicationConfig;
import com.alibaba.dubbo.rpc.config.RegistryConfig;
import com.alibaba.dubbo.rpc.config.ProviderConfig;
import com.alibaba.dubbo.rpc.config.ServiceConfig;
import com.xxx.XxxService;
import com.xxx.XxxServiceImpl;

// 服务实现
XxxService xxxService = new XxxServiceImpl();

// 当前应用配置
ApplicationConfig application = new ApplicationConfig();
application.setName("xxx");

// 连接注册中心配置
RegistryConfig registry = new RegistryConfig();
registry.setAddress("10.20.130.230:9090");
registry.setUsername("aaa");
registry.setPassword("bbb");

// 服务提供者协议配置
ProtocolConfig protocol = new ProtocolConfig();
protocol.setName("dubbo");
protocol.setPort(12345);
protocol.setThreads(200);

// 注意：ServiceConfig为重对象，内部封装了与注册中心的连接，以及开启服务端口

// 服务提供者暴露服务配置
ServiceConfig<XxxService> service = new ServiceConfig<XxxService>(); // 此实例很重，封装了与注册中心的连接，请自行缓存，否则可能造成内存和连接泄漏
service.setApplication(application);
service.setRegistry(registry); // 多个注册中心可以用setRegistries()
service.setProtocol(protocol); // 多个协议可以用setProtocols()
service.setInterface(XxxService.class);
service.setRef(xxxService);
service.setVersion("1.0.0");

// 暴露及注册服务
service.export();
```

服务消费者

```
import com.alibaba.dubbo.rpc.config.ApplicationConfig;
import com.alibaba.dubbo.rpc.config.RegistryConfig;
import com.alibaba.dubbo.rpc.config.ConsumerConfig;
import com.alibaba.dubbo.rpc.config.ReferenceConfig;
import com.xxx.XxxService;

// 当前应用配置
ApplicationConfig application = new ApplicationConfig();
application.setName("yyy");

// 连接注册中心配置
RegistryConfig registry = new RegistryConfig();
registry.setAddress("10.20.130.230:9090");
registry.setUsername("aaa");
registry.setPassword("bbb");

// 注意：ReferenceConfig为重对象，内部封装了与注册中心的连接，以及与服务提供方的连接

// 引用远程服务
ReferenceConfig<XxxService> reference = new ReferenceConfig<XxxService>(); // 此实例很重，封装了与注册中心的连接以及与提供者的连接，请自行缓存，否则可能造成内存和连接泄漏
reference.setApplication(application);
reference.setRegistry(registry); // 多个注册中心可以用setRegistries()
reference.setInterface(XxxService.class);
reference.setVersion("1.0.0");

// 和本地bean一样使用xxxService
XxxService xxxService = reference.get(); // 注意：此代理对象内部封装了所有通讯细节，对象较重，请缓存复用
```

特殊场景

下面只列出不同的地方，其它参见上面的写法

方法级设置

```
...

// 方法级配置
List<MethodConfig> methods = new ArrayList<MethodConfig>();
MethodConfig method = new MethodConfig();
method.setName("createXxx");
method.setTimeout(10000);
method.setRetries(0);
methods.add(method);

// 引用远程服务
ReferenceConfig<XxxService> reference = new ReferenceConfig<XxxService>(); // 此实例很重
，封装了与注册中心的连接以及与提供者的连接，请自行缓存，否则可能造成内存和连接泄漏
...
reference.setMethods(methods); // 设置方法级配置

...
```

点对点直连

```
...

ReferenceConfig<XxxService> reference = new ReferenceConfig<XxxService>(); // 此实例很重
，封装了与注册中心的连接以及与提供者的连接，请自行缓存，否则可能造成内存和连接泄漏
// 如果点对点直连，可以用reference.setUrl()指定目标地址，设置url后将绕过注册中心，
// 其中，协议对应provider.setProtocol()的值，端口对应provider.setPort()的值，
// 路径对应service.setPath()的值，如果未设置path，缺省path为接口名
reference.setUrl("dubbo://10.20.130.230:20880/com.xxx.XxxService");

...
```

> [1]. API使用范围说明：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XML 配置方式使用 Dubbo ↩

注解配置

需要 `2.2.1` 以上版本支持

服务提供方

注解

```
import com.alibaba.dubbo.config.annotation.Service;

@Service(version="1.0.0")
public class FooServiceImpl implements FooService {
    // ...
}
```

XML 配置

```
<!-- 公共信息，也可以用dubbo.properties配置 -->
<dubbo:application name="annotation-provider" />
<dubbo:registry address="127.0.0.1:4548" />

<!-- 扫描注解包路径，多个包用逗号分隔，不填pacakge表示扫描当前ApplicationContext中所有的类 -->
<dubbo:annotation package="com.foo.bar.service" />
```

服务消费方

注解

```
import com.alibaba.dubbo.config.annotation.Reference;
import org.springframework.stereotype.Component;

@Component
public class BarAction {
    @Reference(version="1.0.0")
    private FooService fooService;
}
```

XML 配置

```
<!-- 公共信息，也可以用dubbo.properties配置 -->
<dubbo:application name="annotation-consumer" />
<dubbo:registry address="127.0.0.1:4548" />

<!-- 扫描注解包路径，多个包用逗号分隔，不填pacakge表示扫描当前ApplicationContext中所有的类 -->
<dubbo:annotation package="com.foo.bar.action" />
```

关于 XML 配置的一些说明

也可以使用以下的 XML 配置完成 annotation 扫描，与前面的 `<dubbo:annotation package="..." />` 等价

```
<dubbo:annotation />
<context:component-scan base-package="com.foo.bar.service">
    <context:include-filter type="annotation" expression="com.alibaba.dubbo.config.ann
otation.Service" />
</context:component-scan>
```

Spring 2.5 及以后版本支持 `<component-scan>`，对于 Spring 2.0 及以前版本，需要按照下面的方式配置：

```
<!-- Spring2.0支持@Service注解配置，但不支持package属性自动加载bean的实例，需人工定义bean的实例
。-->
<dubbo:annotation />
<bean id="barService" class="com.foo.BarServiceImpl" />
```

示例

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。

示例

通过 spring 配置文件

关闭某个服务的启动时检查 (没有提供者时报错)：

```
<dubbo:reference interface="com.foo.BarService" check="false" />
```

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" />
```

通过 dubbo.properties

```
dubbo.reference.com.foo.BarService.check=false
dubbo.reference.check=false
dubbo.consumer.check=false
dubbo.registry.check=false
```

通过 -D 参数

```
java -Ddubbo.reference.com.foo.BarService.check=false
java -Ddubbo.reference.check=false
java -Ddubbo.consumer.check=false
java -Ddubbo.registry.check=false
```

配置的含义

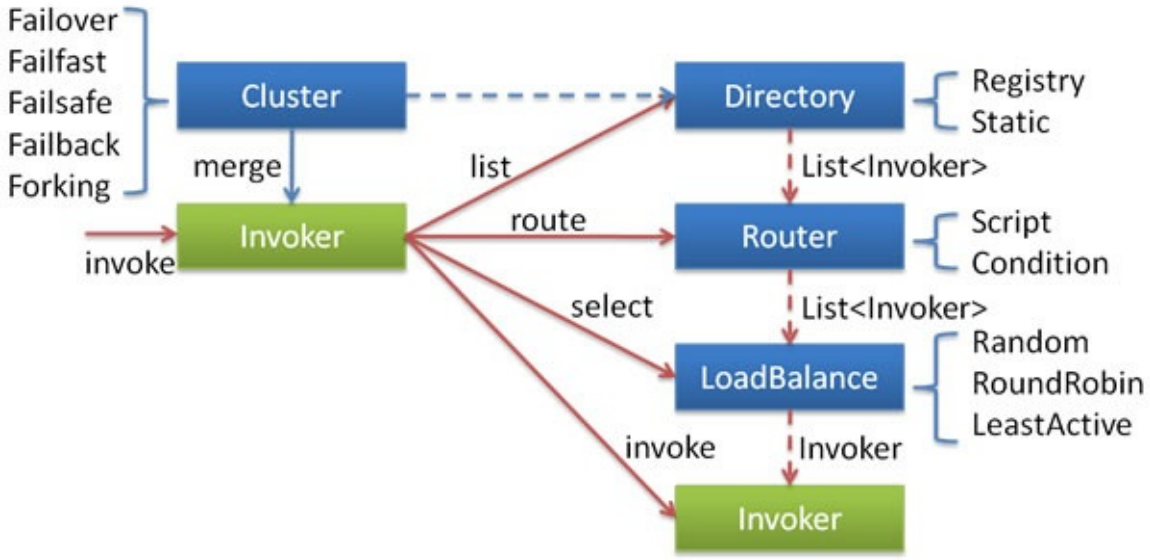
`dubbo.reference.check=false`，强制改变所有 reference 的 check 值，就算配置中有声明，也会被覆盖。

`dubbo.consumer.check=false`，是设置 check 的缺省值，如果配置中有显式的声明，如：`<dubbo:reference check="true"/>`，不会受影响。

`dubbo.registry.check=false`，前面两个都是指订阅成功，但提供者列表是否为空是否报错，如果注册订阅失败时，也允许启动，需使用此选项，将在后台定时重试。

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。



各节点关系：

- 这里的 `Invoker` 是 `Provider` 的一个可调用 `Service` 的抽象，`Invoker` 封装了 `Provider` 地址及 `Service` 接口信息
- `Directory` 代表多个 `Invoker`，可以把它看成 `List<Invoker>` ，但与 `List` 不同的是，它的值可能是动态变化的，比如注册中心推送变更
- `Cluster` 将 `Directory` 中的多个 `Invoker` 伪装成一个 `Invoker`，对上层透明，伪装过程包含了容错逻辑，调用失败后，重试另一个
- `Router` 负责从多个 `Invoker` 中按路由规则选出子集，比如读写分离，应用隔离等
- `LoadBalance` 负责从多个 `Invoker` 中选出具体的一个用于本次调用，选的过程包含了负载均衡算法，调用失败后，需要重选

集群容错模式

可以自行扩展集群容错策略，参见：集群扩展

Failover Cluster

失败自动切换，当出现失败，重试其它服务器 [1]。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)。

重试次数配置如下：

```
<dubbo:service retries="2" />
```

或

```
<dubbo:reference retries="2" />
```

或

```
<dubbo:reference>
    <dubbo:method name="findFoo" retries="2" />
</dubbo:reference>
```

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]。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。

集群模式配置

按照以下示例在服务提供方和消费方配置集群模式

```
<dubbo:service cluster="failsafe" />
```

或

```
<dubbo:reference cluster="failsafe" />
```

1. 该配置为缺省配置 ↩

2. `2.1.0` 开始支持 ↩

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。

可以自行扩展负载均衡策略，参见：负载均衡扩展

负载均衡策略

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的问题，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" />`

配置

服务端服务级别

```
<dubbo:service interface="..." loadbalance="roundrobin" />
```

客户端服务级别

```
<dubbo:reference interface="..." loadbalance="roundrobin" />
```

服务端方法级别

```
<dubbo:service interface="...">
    <dubbo:method name="..." loadbalance="roundrobin"/>
</dubbo:service>
```

客户端方法级别

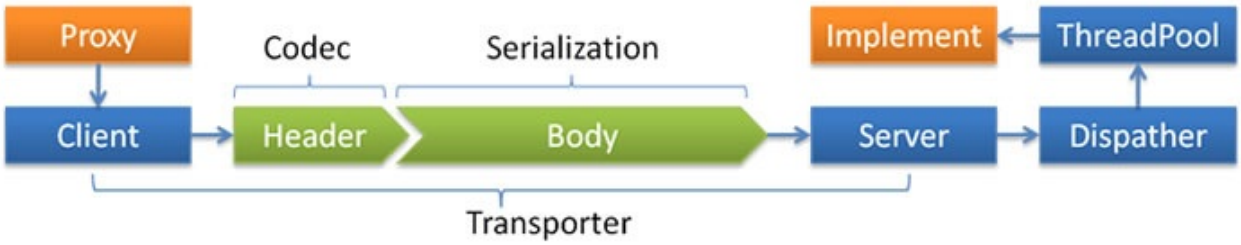
```
<dubbo:reference interface="...">
    <dubbo:method name="..." loadbalance="roundrobin"/>
</dubbo:reference>
```

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。



因此，需要通过不同的派发策略和不同的线程池配置的组合来应对不同的场景:

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

Dispatcher

- `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

- `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。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。

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。

通过 XML 配置

如果是线上需求需要点对点，可在 `<dubbo:reference>` 中配置 url 指向提供者，将绕过注册中心，多个地址用分号隔开，配置如下[1]：

```
<dubbo:reference id="xxxService" interface="com.alibaba.xxx.XxxService" url="dubbo://l
ocalhost:20890" />
```

通过 -D 参数指定

在 JVM 启动参数中加入-D参数映射服务地址[2]，如：

```
java -Dcom.alibaba.xxx.XxxService=dubbo://localhost:20890
```

通过文件映射

如果服务比较多，也可以用文件映射，用 `-Ddubbo.resolve.file` 指定映射文件路径，此配置优先级高于 `<dubbo:reference>` 中的配置[3]，如：

```
java -Ddubbo.resolve.file=xxx.properties
```

然后在映射文件 `xxx.properties` 中加入配置，其中 key 为服务名，value 为服务提供者 URL：

```
com.alibaba.xxx.XxxService=dubbo://localhost:20890
```

注意 为了避免复杂化线上环境，不要在线上使用这个功能，只应在测试阶段使用。

1. `1.0.6` 及以上版本支持 ↩

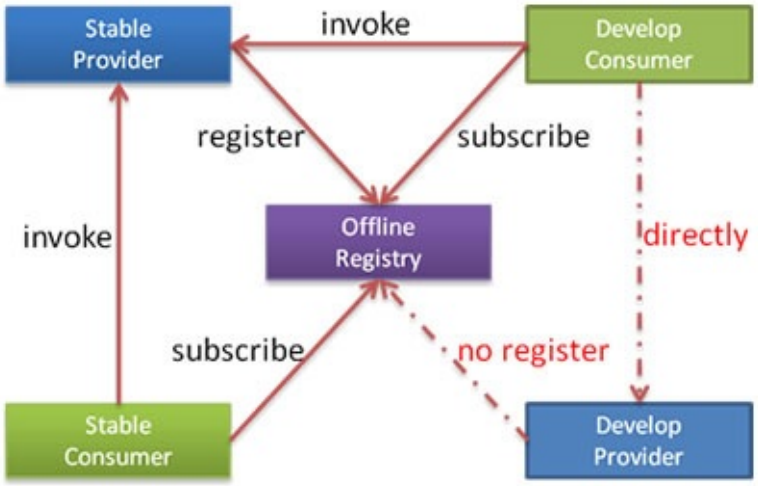
2. key 为服务名，value 为服务提供者 url，此配置优先级最高，`1.0.15` 及以上版本支持 ↩

3. `1.0.15` 及以上版本支持，`2.0` 以上版本自动加载 ${user.home}/dubbo-resolve.properties文件，不需要配置 ↩

只订阅

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。



禁用注册配置

```
<dubbo:registry address="10.20.153.10:9090" register="false" />
```

或者

```
<dubbo:registry address="10.20.153.10:9090?register=false" />
```

只注册

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。这个时候，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。

禁用订阅配置

```
<dubbo:registry id="hzRegistry" address="10.20.153.10:9090" />
<dubbo:registry id="qdRegistry" address="10.20.141.150:9090" subscribe="false" />
```

或者

```
<dubbo:registry id="hzRegistry" address="10.20.153.10:9090" />
<dubbo:registry id="qdRegistry" address="10.20.141.150:9090?subscribe=false" />
```

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。

```
<dubbo:registry address="10.20.141.150:9090" dynamic="false" />
```

或者

```
<dubbo:registry address="10.20.141.150:9090?dynamic=false" />
```

服务提供者初次注册时为禁用状态，需人工启用。断线时，将不会被自动删除，需人工禁用。

如果是一个第三方独立提供者，比如 memcached，可以直接向注册中心写入提供者地址信息，消费者正常使用 [1]：

```
RegistryFactory registryFactory = ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptiveExtension();
Registry registry = registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:2181"));
registry.register(URL.valueOf("memcached://10.20.153.11/com.foo.BarService?category=providers&dynamic=false&application=foo"));
```

1. 通常由脚本监控中心页面等调用 ↩

多协议

Dubbo 允许配置多协议，在不同服务上支持不同协议或者同一服务上同时支持多种协议。

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

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
    xsi:schemaLocation="http://www.springframework.org/schema/beanshttp://www.springfr
amework.org/schema/beans/spring-beans.xsdhttp://code.alibabatech.com/schema/dubbohttp:
//code.alibabatech.com/schema/dubbo/dubbo.xsd">
    <dubbo:application name="world"  />
    <dubbo:registry id="registry" address="10.20.141.150:9090" username="admin" passwo
rd="hello1234" />
    <!-- 多协议配置 -->
    <dubbo:protocol name="dubbo" port="20880" />
    <dubbo:protocol name="rmi" port="1099" />
    <!-- 使用dubbo协议暴露服务 -->
    <dubbo:service interface="com.alibaba.hello.api.HelloService" version="1.0.0" ref=
"helloService" protocol="dubbo" />
    <!-- 使用rmi协议暴露服务 -->
    <dubbo:service interface="com.alibaba.hello.api.DemoService" version="1.0.0" ref="
demoService" protocol="rmi" />
</beans>
```

多协议暴露服务

需要与 http 客户端互操作

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
    xsi:schemaLocation="http://www.springframework.org/schema/beanshttp://www.springfr
amework.org/schema/beans/spring-beans.xsdhttp://code.alibabatech.com/schema/dubbohttp:
//code.alibabatech.com/schema/dubbo/dubbo.xsd">
    <dubbo:application name="world"  />
    <dubbo:registry id="registry" address="10.20.141.150:9090" username="admin" passwo
rd="hello1234" />
    <!-- 多协议配置 -->
    <dubbo:protocol name="dubbo" port="20880" />
    <dubbo:protocol name="hessian" port="8080" />
    <!-- 使用多个协议暴露服务 -->
    <dubbo:service id="helloService" interface="com.alibaba.hello.api.HelloService" ve
rsion="1.0.0" protocol="dubbo,hessian" />
</beans>
```

1. 可以自行扩展协议，参见：协议扩展 ↩

多注册中心

Dubbo 支持同一服务向多注册中心同时注册，或者不同服务分别注册到不同的注册中心上去，甚至可以同时引用注册在不同注册中心上的同名服务。另外，注册中心是支持自定义扩展的[1]。

多注册中心注册

比如：中文站有些服务来不及在青岛部署，只在杭州部署，而青岛的其它应用需要引用此服务，就可以将服务同时注册到两个注册中心。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
    xsi:schemaLocation="http://www.springframework.org/schema/beanshttp://www.springfr
amework.org/schema/beans/spring-beans.xsdhttp://code.alibabatech.com/schema/dubbohttp:
//code.alibabatech.com/schema/dubbo/dubbo.xsd">
    <dubbo:application name="world"  />
    <!-- 多注册中心配置 -->
    <dubbo:registry id="hangzhouRegistry" address="10.20.141.150:9090" />
    <dubbo:registry id="qingdaoRegistry" address="10.20.141.151:9010" default="false"
/>
    <!-- 向多个注册中心注册 -->
    <dubbo:service interface="com.alibaba.hello.api.HelloService" version="1.0.0" ref=
"helloService" registry="hangzhouRegistry,qingdaoRegistry" />
</beans>
```

不同服务使用不同注册中心

比如：CRM 有些服务是专门为国际站设计的，有些服务是专门为中文站设计的。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
    xsi:schemaLocation="http://www.springframework.org/schema/beanshttp://www.springfr
amework.org/schema/beans/spring-beans.xsdhttp://code.alibabatech.com/schema/dubbohttp:
//code.alibabatech.com/schema/dubbo/dubbo.xsd">
    <dubbo:application name="world"  />
    <!-- 多注册中心配置 -->
    <dubbo:registry id="chinaRegistry" address="10.20.141.150:9090" />
    <dubbo:registry id="intlRegistry" address="10.20.154.177:9010" default="false" />
    <!-- 向中文站注册中心注册 -->
    <dubbo:service interface="com.alibaba.hello.api.HelloService" version="1.0.0" ref=
"helloService" registry="chinaRegistry" />
    <!-- 向国际站注册中心注册 -->
    <dubbo:service interface="com.alibaba.hello.api.DemoService" version="1.0.0" ref="
demoService" registry="intlRegistry" />
</beans>
```

多注册中心引用

比如：CRM 需同时调用中文站和国际站的 PC2 服务，PC2 在中文站和国际站均有部署，接口及版本号都一样，但连的数据库不一样。

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
    xsi:schemaLocation="http://www.springframework.org/schema/beanshttp://www.springfr
amework.org/schema/beans/spring-beans.xsdhttp://code.alibabatech.com/schema/dubbohttp:
//code.alibabatech.com/schema/dubbo/dubbo.xsd">
    <dubbo:application name="world"  />
    <!-- 多注册中心配置 -->
    <dubbo:registry id="chinaRegistry" address="10.20.141.150:9090" />
    <dubbo:registry id="intlRegistry" address="10.20.154.177:9010" default="false" />
    <!-- 引用中文站服务 -->
    <dubbo:reference id="chinaHelloService" interface="com.alibaba.hello.api.HelloServ
ice" version="1.0.0" registry="chinaRegistry" />
    <!-- 引用国际站站服务 -->
    <dubbo:reference id="intlHelloService" interface="com.alibaba.hello.api.HelloServi
ce" version="1.0.0" registry="intlRegistry" />
</beans>
```

如果只是测试环境临时需要连接两个不同注册中心，使用竖号分隔多个不同注册中心地址：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
    xsi:schemaLocation="http://www.springframework.org/schema/beanshttp://www.springfr
amework.org/schema/beans/spring-beans.xsdhttp://code.alibabatech.com/schema/dubbohttp:
//code.alibabatech.com/schema/dubbo/dubbo.xsd">
    <dubbo:application name="world"  />
    <!-- 多注册中心配置，竖号分隔表示同时连接多个不同注册中心，同一注册中心的多个集群地址用逗号分隔 -
->
    <dubbo:registry address="10.20.141.150:9090|10.20.154.177:9010" />
    <!-- 引用服务 -->
    <dubbo:reference id="helloService" interface="com.alibaba.hello.api.HelloService"
version="1.0.0" />
</beans>
```

1. 可以自行扩展注册中心，参见：注册中心扩展 ↩

服务分组

当一个接口有多种实现时，可以用 group 区分。

服务

```
<dubbo:service group="feedback" interface="com.xxx.IndexService" />
<dubbo:service group="member" interface="com.xxx.IndexService" />
```

引用

```
<dubbo:reference id="feedbackIndexService" group="feedback" interface="com.xxx.IndexSe
rvice" />
<dubbo:reference id="memberIndexService" group="member" interface="com.xxx.IndewxServi
ce" />
```

任意组[1]：

```
<dubbo:reference id="barService" interface="com.foo.BarService" group="*" />
```

1. `2.2.0` 以上版本支持，总是只调一个可用组的实现 ↩

多版本

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。

可以按照以下的步骤进行版本迁移：

1. 在低压力时间段，先升级一半提供者为新版本
2. 再将所有消费者升级为新版本
3. 然后将剩下的一半提供者升级为新版本

老版本服务提供者配置：

```
<dubbo:service interface="com.foo.BarService" version="1.0.0" />
```

新版本服务提供者配置：

```
<dubbo:service interface="com.foo.BarService" version="2.0.0" />
```

老版本服务消费者配置：

```
<dubbo:reference id="barService" interface="com.foo.BarService" version="1.0.0" />
```

新版本服务消费者配置：

```
<dubbo:reference id="barService" interface="com.foo.BarService" version="2.0.0" />
```

如果不需要区分版本，可以按照以下的方式配置[1]：

```
<dubbo:reference id="barService" interface="com.foo.BarService" version="*" />
```

1. `2.2.0` 以上版本支持 ↩

分组聚合

按组合并返回结果[1]，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，这样就可以实现聚合菜单项。

相关代码可以参考 dubbo 项目中的示例

配置

搜索所有分组

```
<dubbo:reference interface="com.xxx.MenuService" group="*" merger="true" />
```

合并指定分组

```
<dubbo:reference interface="com.xxx.MenuService" group="aaa,bbb" merger="true" />
```

指定方法合并结果，其它未指定的方法，将只调用一个 Group

```
<dubbo:reference interface="com.xxx.MenuService" group="*">
    <dubbo:method name="getMenuItems" merger="true" />
</dubbo:service>
```

某个方法不合并结果，其它都合并结果

```
<dubbo:reference interface="com.xxx.MenuService" group="*" merger="true">
    <dubbo:method name="getMenuItems" merger="false" />
</dubbo:service>
```

指定合并策略，缺省根据返回值类型自动匹配，如果同一类型有两个合并器时，需指定合并器的名称[2]

```
<dubbo:reference interface="com.xxx.MenuService" group="*">
    <dubbo:method name="getMenuItems" merger="mymerge" />
</dubbo:service>
```

指定合并方法，将调用返回结果的指定方法进行合并，合并方法的参数类型必须是返回结果类型本身

```
<dubbo:reference interface="com.xxx.MenuService" group="*">
    <dubbo:method name="getMenuItems" merger=".addAll" />
</dubbo:service>
```

1. 从 `2.1.0` 版本开始支持 ↩

2. 参见：合并结果扩展 ↩

参数验证

参数验证功能[1]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[2]。

Maven 依赖

```
<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>1.0.0.GA</version>
</dependency>
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>4.2.0.Final</version>
</dependency>
```

示例

参数标注示例

```
import java.io.Serializable;
import java.util.Date;

import javax.validation.constraints.Future;
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Past;
import javax.validation.constraints.Pattern;
import javax.validation.constraints.Size;

public class ValidationParameter implements Serializable {
    private static final long serialVersionUID = 7158911668568000392L;

    @NotNull // 不允许为空
    @Size(min = 1, max = 20) // 长度或大小范围
    private String name;

    @NotNull(groups = ValidationService.Save.class) // 保存时不允许为空，更新时允许为空 ，表示不更新该字段
```

```
    @Pattern(regexp = "^\\s*\\w+(?:\\.{0,1}[\\w-]+)*@[a-zA-Z0-9]+(?:[-.][a-zA-Z0-9]+)*
\\.[a-zA-Z]+\\s*$")
    private String email;

    @Min(18) // 最小值
    @Max(100) // 最大值
    private int age;

    @Past // 必须为一个过去的时间
    private Date loginDate;

    @Future // 必须为一个未来的时间
    private Date expiryDate;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public Date getLoginDate() {
        return loginDate;
    }

    public void setLoginDate(Date loginDate) {
        this.loginDate = loginDate;
    }

    public Date getExpiryDate() {
        return expiryDate;
    }

    public void setExpiryDate(Date expiryDate) {
        this.expiryDate = expiryDate;
    }
```

```
}
```

分组验证示例

```
public interface ValidationService { // 缺省可按服务接口区分验证场景，如：@NotNull(groups = ValidationService.class)
    @interface Save{} // 与方法同名接口，首字母大写，用于区分验证场景，如：@NotNull(groups = ValidationService.Save.class)，可选
    void save(ValidationParameter parameter);
    void update(ValidationParameter parameter);
}
```

关联验证示例

```
import javax.validation.GroupSequence;

public interface ValidationService {
    @GroupSequence(Update.class) // 同时验证Update组规则
    @interface Save{}
    void save(ValidationParameter parameter);

    @interface Update{}
    void update(ValidationParameter parameter);
}
```

参数验证示例

```
import javax.validation.constraints.Min;
import javax.validation.constraints.NotNull;

public interface ValidationService {
    void save(@NotNull ValidationParameter parameter); // 验证参数不为空
    void delete(@Min(1) int id); // 直接对基本类型参数验证
}
```

配置

在客户端验证参数

```
<dubbo:reference id="validationService" interface="com.alibaba.dubbo.examples.validation.api.ValidationService" validation="true" />
```

在服务器端验证参数

```
<dubbo:service interface="com.alibaba.dubbo.examples.validation.api.ValidationService"
ref="validationService" validation="true" />
```

验证异常信息

```
import javax.validation.ConstraintViolationException;
import javax.validation.ConstraintViolationException;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.alibaba.dubbo.examples.validation.api.ValidationParameter;
import com.alibaba.dubbo.examples.validation.api.ValidationService;
import com.alibaba.dubbo.rpc.RpcException;

public class ValidationConsumer {
    public static void main(String[] args) throws Exception {
        String config = ValidationConsumer.class.getPackage().getName().replace('.', '
/') + "/validation-consumer.xml";
        ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(co
nfig);
        context.start();
        ValidationService validationService = (ValidationService)context.getBean("vali
dationService");
        // Error
        try {
            parameter = new ValidationParameter();
            validationService.save(parameter);
            System.out.println("Validation ERROR");
        } catch (RpcException e) { // 抛出的是RpcException
            ConstraintViolationException ve = (ConstraintViolationException) e.getCaus
e(); // 里面嵌了一个ConstraintViolationException
            Set<ConstraintViolation<?>> violations = ve.getConstraintViolations(); //
可以拿到一个验证错误详细信息的集合
            System.out.println(violations);
        }
    }
}
```

1. 自 `2.1.0` 版本开始支持, 如何使用可以参考 dubbo 项目中的示例代码 ↩

2. 验证方式可扩展，扩展方式参见开发者手册中的验证扩展 ↩

结果缓存

结果缓存[1]，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[2]。

缓存类型

- `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。
- `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。

缓存类型可扩展，参见：缓存扩展

配置

```
<dubbo:reference interface="com.foo.BarService" cache="lru" />
```

或：

```
<dubbo:reference interface="com.foo.BarService">
    <dubbo:method name="findBar" cache="lru" />
</dubbo:reference>
```

1. `2.1.0` 以上版本支持 ↩
2. 示例代码 ↩

使用泛化调用

泛化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，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。

通过 Spring 使用泛化调用

在 Spring 配置申明 `generic="true"`：

```
<dubbo:reference id="barService" interface="com.foo.BarService" generic="true" />
```

在 Java 代码获取 barService 并开始泛化调用：

```
GenericService barService = (GenericService) applicationContext.getBean("barService");
Object result = barService.$invoke("sayHello", new String[] { "java.lang.String" }, new
 Object[] { "World" });
```

通过 API 方式使用泛化调用

```
import com.alibaba.dubbo.rpc.service.GenericService;
...

// 引用远程服务
// 该实例很重量，里面封装了所有与注册中心及服务提供方连接，请缓存
ReferenceConfig<GenericService> reference = new ReferenceConfig<GenericService>();
// 弱类型接口名
reference.setInterface("com.xxx.XxxService");
reference.setVersion("1.0.0");
// 声明为泛化接口
reference.setGeneric(true);

// 用com.alibaba.dubbo.rpc.service.GenericService可以替代所有接口引用
GenericService genericService = reference.get();

// 基本类型以及Date,List,Map等不需要转换，直接调用
Object result = genericService.$invoke("sayHello", new String[] {"java.lang.String"},
new Object[] {"world"});

// 用Map表示POJO参数，如果返回值为POJO也将自动转成Map
Map<String, Object> person = new HashMap<String, Object>();
person.put("name", "xxx");
person.put("password", "yyy");
// 如果返回POJO将自动转成Map
Object result = genericService.$invoke("findPerson", new String[]
{"com.xxx.Person"}, new Object[]{person});

...
```

有关泛化类型的进一步解释

假设存在 POJO 如：

```
package com.xxx;

public class PersonImpl implements Person {
    private String name;
    private String password;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

则 POJO 数据：

```
Person person = new PersonImpl();
person.setName("xxx");
person.setPassword("yyy");
```

可用下面 Map 表示：

```
Map<String, Object> map = new HashMap<String, Object>();
// 注意：如果参数类型是接口，或者List等丢失泛型，可通过class属性指定类型。
map.put("class", "com.xxx.PersonImpl");
map.put("name", "xxx");
map.put("password", "yyy");
```

实现泛化调用

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。

在 Java 代码中实现 `GenericService` 接口：

```
package com.foo;
public class MyGenericService implements GenericService {

    public Object $invoke(String methodName, String[] parameterTypes, Object[] args) t
hrows GenericException {
        if ("sayHello".equals(methodName)) {
            return "Welcome " + args[0];
        }
    }
}
```

通过 Spring 暴露泛化实现

在 Spring 配置申明服务的实现：

```
<bean id="genericService" class="com.foo.MyGenericService" />
<dubbo:service interface="com.foo.BarService" ref="genericService" />
```

通过 API 方式暴露泛化实现

```
...
// 用com.alibaba.dubbo.rpc.service.GenericService可以替代所有接口实现
GenericService xxxService = new XxxGenericService();

// 该实例很重量，里面封装了所有与注册中心及服务提供方连接，请缓存
ServiceConfig<GenericService> service = new ServiceConfig<GenericService>();
// 弱类型接口名
service.setInterface("com.xxx.XxxService");
service.setVersion("1.0.0");
// 指向一个通用服务实现
service.setRef(xxxService);

// 暴露及注册服务
service.export();
```

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。

Spring 配置：

```
<dubbo:reference id="memberService" interface="com.xxx.MemberService" />
```

代码：

```
// 远程服务引用
MemberService memberService = ctx.getBean("memberService");

EchoService echoService = (EchoService) memberService; // 强制转型为EchoService

// 回声测试可用性
String status = echoService.$echo("OK");

assert(status.equals("OK"));
```

上下文信息

上下文中存放的是当前调用过程中所需的环境信息。所有配置信息都将转换为 URL 的参数，参见 schema 配置参考手册 中的对应**URL**参数一列。

RpcContext 是一个 ThreadLocal 的临时状态记录器，当接收到 RPC 请求，或发起 RPC 请求时，RpcContext 的状态都会变化。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。

服务消费方

```
// 远程调用
xxxService.xxx();
// 本端是否为消费端，这里会返回true
boolean isConsumerSide = RpcContext.getContext().isConsumerSide();
// 获取最后一次调用的提供方IP地址
String serverIP = RpcContext.getContext().getRemoteHost();
// 获取当前服务配置信息，所有配置信息都将转换为URL的参数
String application = RpcContext.getContext().getUrl().getParameter("application");
// 注意：每发起RPC调用，上下文状态会变化
yyyService.yyy();
```

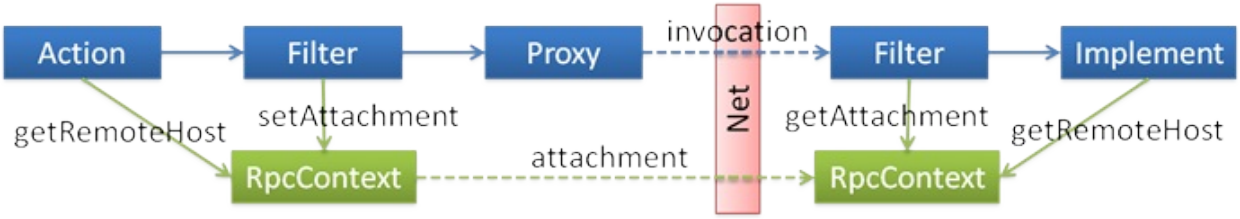
服务提供方

```
public class XxxServiceImpl implements XxxService {

    public void xxx() {
        // 本端是否为提供端，这里会返回true
        boolean isProviderSide = RpcContext.getContext().isProviderSide();
        // 获取调用方IP地址
        String clientIP = RpcContext.getContext().getRemoteHost();
        // 获取当前服务配置信息，所有配置信息都将转换为URL的参数
        String application = RpcContext.getContext().getUrl().getParameter("applicatio
n");
        // 注意：每发起RPC调用，上下文状态会变化
        yyyService.yyy();
        // 此时本端变成消费端，这里会返回false
        boolean isProviderSide = RpcContext.getContext().isProviderSide();
    }
}
```

隐式参数

可以通过 `RpcContext` 上的 `setAttachment` 和 `getAttachment` 在服务消费方和提供方之间进行参数的隐式传递。[1]



在服务消费方端设置隐式参数

`setAttachment` 设置的 KV 对，在完成下面一次远程调用会被清空，即多次远程调用要多次设置。

```
RpcContext.getContext().setAttachment("index", "1"); // 隐式传参，后面的远程调用都会隐式将这些参数发送到服务器端，类似cookie，用于框架集成，不建议常规业务使用
xxxService.xxx(); // 远程调用
// ...
```

在服务提供方端获取隐式参数

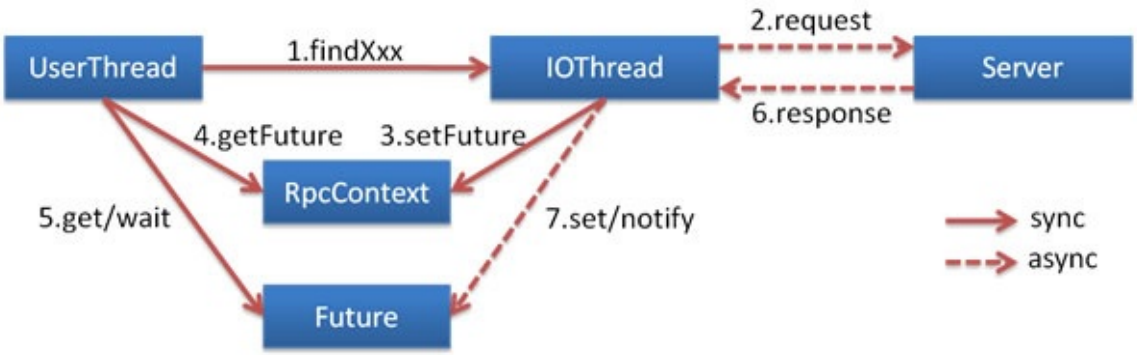
```
public class XxxServiceImpl implements XxxService {

    public void xxx() {
        // 获取客户端隐式传入的参数，用于框架集成，不建议常规业务使用
        String index = RpcContext.getContext().getAttachment("index");
    }
}
```

1. 注意：path, group, version, dubbo, token, timeout 几个 key 是保留字段，请使用其它值。 ↩

异步调用

基于 NIO 的非阻塞实现并行调用，客户端不需要启动多线程即可完成并行调用多个远程服务，相对多线程开销较小。[1]



在 consumer.xml 中配置：

```
<dubbo:reference id="fooService" interface="com.alibaba.foo.FooService">
      <dubbo:method name="findFoo" async="true" />
</dubbo:reference>
<dubbo:reference id="barService" interface="com.alibaba.bar.BarService">
      <dubbo:method name="findBar" async="true" />
</dubbo:reference>
```

调用代码:

```
// 此调用会立即返回null
fooService.findFoo(fooId);
// 拿到调用的Future引用，当结果返回后，会被通知和设置到此Future
Future<Foo> fooFuture = RpcContext.getContext().getFuture();

// 此调用会立即返回null
barService.findBar(barId);
// 拿到调用的Future引用，当结果返回后，会被通知和设置到此Future
Future<Bar> barFuture = RpcContext.getContext().getFuture();

// 此时findFoo和findBar的请求同时在执行，客户端不需要启动多线程来支持并行，而是借助NIO的非阻塞完成

// 如果foo已返回，直接拿到返回值，否则线程wait住，等待foo返回后，线程会被notify唤醒
Foo foo = fooFuture.get();
// 同理等待bar返回
Bar bar = barFuture.get();

// 如果foo需要5秒返回，bar需要6秒返回，实际只需等6秒，即可获取到foo和bar，进行接下来的处理。
```

你也可以设置是否等待消息发出：[2]

- `sent="true"` 等待消息发出，消息发送失败将抛出异常。
- `sent="false"` 不等待消息发出，将消息放入 IO 队列，即刻返回。

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

如果你只是想异步，完全忽略返回值，可以配置 `return="false"`，以减少 Future 对象的创建和管理成本：

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

1. `2.0.6` 及其以上版本支持 ↩

2. 异步总是不等待返回 ↩

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。

配置

定义 injvm 协议

```
<dubbo:protocol name="injvm" />
```

设置默认协议

```
<dubbo:provider protocol="injvm" />
```

设置服务协议

```
<dubbo:service protocol="injvm" />
```

优先使用 injvm

```
<dubbo:consumer injvm="true" .../>
<dubbo:provider injvm="true" .../>
```

或

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

注意：服务暴露与服务引用都需要声明 `injvm="true"`

自动暴露、引用本地服务

从 `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" />
```

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[1]。可以参考 dubbo 项目中的示例代码。

服务接口示例

CallbackService.java

```
package com.callback;

public interface CallbackService {
    void addListener(String key, CallbackListener listener);
}
```

CallbackListener.java

```
package com.callback;

public interface CallbackListener {
    void changed(String msg);
}
```

服务提供者接口实现示例

```
package com.callback.impl;

import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Map;
import java.util.concurrent.ConcurrentHashMap;

import com.callback.CallbackListener;
import com.callback.CallbackService;

public class CallbackServiceImpl implements CallbackService {

    private final Map<String, CallbackListener> listeners = new ConcurrentHashMap<Stri
ng, CallbackListener>();

    public CallbackServiceImpl() {
        Thread t = new Thread(new Runnable() {
            public void run() {
                while(true) {
                    try {
                        for(Map.Entry<String, CallbackListener> entry : listeners.entr
ySet()){
                            try {
                                entry.getValue().changed(getChanged(entry.getKey()));
                            } catch (Throwable t) {
                                listeners.remove(entry.getKey());
                            }
                        }
                        Thread.sleep(5000); // 定时触发变更通知
                    } catch (Throwable t) { // 防御容错
                        t.printStackTrace();
                    }
                }
            }
        });
        t.setDaemon(true);
        t.start();
    }

    public void addListener(String key, CallbackListener listener) {
        listeners.put(key, listener);
        listener.changed(getChanged(key)); // 发送变更通知
    }

    private String getChanged(String key) {
        return "Changed: " + new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(new Da
te());
    }
}
```

服务提供者配置示例

```
<bean id="callbackService" class="com.callback.impl.CallbackServiceImpl" />
<dubbo:service interface="com.callback.CallbackService" ref="callbackService" connecti
ons="1" callbacks="1000">
    <dubbo:method name="addListener">
        <dubbo:argument index="1" callback="true" />
        <!--也可以通过指定类型的方式-->
        <!--<dubbo:argument type="com.demo.CallbackListener" callback="true" />-->
    </dubbo:method>
</dubbo:service>
```

服务消费者配置示例

```
<dubbo:reference id="callbackService" interface="com.callback.CallbackService" />
```

服务消费者调用示例

```
ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext("classpath
:consumer.xml");
context.start();

CallbackService callbackService = (CallbackService) context.getBean("callbackService")
;

callbackService.addListener("http://10.20.160.198/wiki/display/dubbo/foo.bar", new Cal
lbackListener(){
    public void changed(String msg) {
        System.out.println("callback1:" + msg);
    }
});
```

1. `2.0.6` 及其以上版本支持 ↩

事件通知

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[1]。

服务提供者与消费者共享服务接口

```
interface IDemoService {
    public Person get(int id);
}
```

服务提供者实现

```
class NormalDemoService implements IDemoService {
    public Person get(int id) {
        return new Person(id, "charles`son", 4);
    }
}
```

服务提供者配置

```
<dubbo:application name="rpc-callback-demo" />
<dubbo:registry address="http://10.20.160.198/wiki/display/dubbo/10.20.153.186" />
<bean id="demoService" class="com.alibaba.dubbo.callback.implicit.NormalDemoService" />

<dubbo:service interface="com.alibaba.dubbo.callback.implicit.IDemoService" ref="demoS
ervice" version="1.0.0" group="cn"/>
```

服务消费者 Callback 接口

```
interface Notify {
    public void onreturn(Person msg, Integer id);
    public void onthrow(Throwable ex, Integer id);
}
```

服务消费者 Callback 实现

```
class NotifyImpl implements Notify {
    public Map<Integer, Person>    ret    = new HashMap<Integer, Person>();
    public Map<Integer, Throwable> errors = new HashMap<Integer, Throwable>();

    public void onreturn(Person msg, Integer id) {
        System.out.println("onreturn:" + msg);
        ret.put(id, msg);
    }

    public void onthrow(Throwable ex, Integer id) {
        errors.put(id, ex);
    }
}
```

服务消费者 Callback 配置

```
<bean id ="demoCallback" class = "com.alibaba.dubbo.callback.implicit.NofifyImpl" />
<dubbo:reference id="demoService" interface="com.alibaba.dubbo.callback.implicit.IDemo
Service" version="1.0.0" group="cn" >
      <dubbo:method name="get" async="true" onreturn = "demoCallback.onreturn" onthrow=
"demoCallback.onthrow" />
</dubbo:reference>
```

`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。

两者叠加存在以下几种组合情况[2]：

- 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`

测试代码

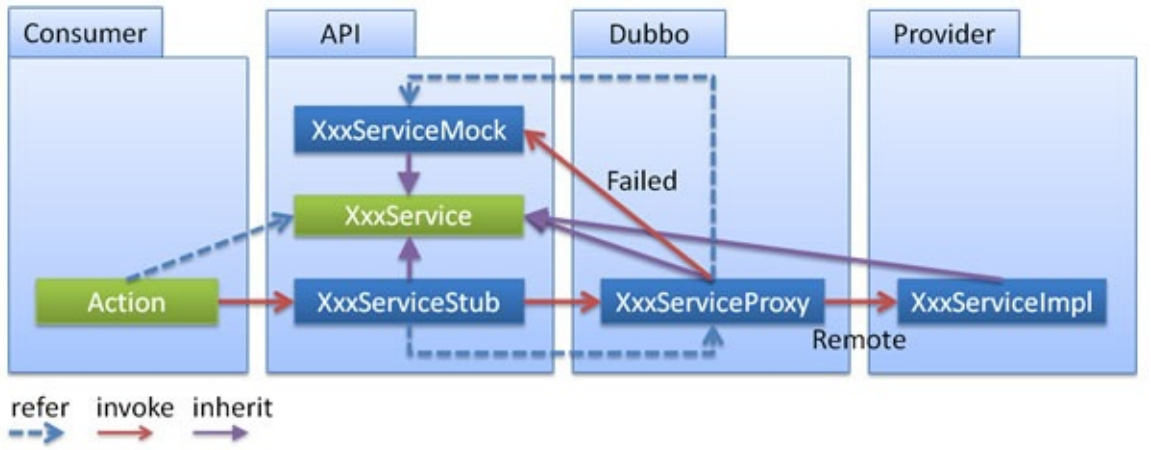
```
IDemoService demoService = (IDemoService) context.getBean("demoService");
NofifyImpl notify = (NofifyImpl) context.getBean("demoCallback");
int requestId = 2;
Person ret = demoService.get(requestId);
Assert.assertEquals(null, ret);
//for Test：只是用来说明callback正常被调用，业务具体实现自行决定.
for (int i = 0; i < 10; i++) {
    if (!notify.ret.containsKey(requestId)) {
        Thread.sleep(200);
    } else {
        break;
    }
}
Assert.assertEquals(requestId, notify.ret.get(requestId).getId());
```

1. 支持版本：`2.0.7` 之后 ↩

2. `async=false` 默认 ↩

本地存根

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



在 spring 配置文件中按以下方式配置：

```
<dubbo:service interface="com.foo.BarService" stub="true" />
```

或

```
<dubbo:service interface="com.foo.BarService" stub="com.foo.BarServiceStub" />
```

提供 Stub 的实现 [2]：

```
package com.foo;
public class BarServiceStub implements BarService {
    private final BarService barService;

    // 构造函数传入真正的远程代理对象
    public (BarService barService) {
        this.barService = barService;
    }

    public String sayHello(String name) {
        // 此代码在客户端执行, 你可以在客户端做ThreadLocal本地缓存，或预先验证参数是否合法，等等
        try {
            return barService.sayHello(name);
        } catch (Exception e) {
            // 你可以容错，可以做任何AOP拦截事项
            return "容错数据";
        }
    }
}
```

1. Stub 必须有可传入 Proxy 的构造函数。 ↩

2. 在 interface 旁边放一个 Stub 实现，它实现 BarService 接口，并有一个传入远程 BarService 实例的构造函数 ↩

本地伪装

本地伪装[1]通常用于服务降级，比如某验权服务，当服务提供方全部挂掉后，客户端不抛出异常，而是通过 Mock 数据返回授权失败。

在 spring 配置文件中按以下方式配置：

```
<dubbo:service interface="com.foo.BarService" mock="true" />
```

或

```
<dubbo:service interface="com.foo.BarService" mock="com.foo.BarServiceMock" />
```

在工程中提供 Mock 实现[2]：

```
package com.foo;
public class BarServiceMock implements BarService {
    public String sayHello(String name) {
        // 你可以伪造容错数据，此方法只在出现RpcException时被执行
        return "容错数据";
    }
}
```

如果服务的消费方经常需要 try-catch 捕获异常，如：

```
Offer offer = null;
try {
    offer = offerService.findOffer(offerId);
} catch (RpcException e) {
   logger.error(e);
}
```

请考虑改为 Mock 实现，并在 Mock 实现中 return null。如果只是想简单的忽略异常，在 `2.0.11` 以上版本可用：

```
<dubbo:service interface="com.foo.BarService" mock="return null" />
```

[1]. 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。 ↩

[2]. 在 interface 旁放一个 Mock 实现，它实现 BarService 接口，并有一个无参构造函数 ↩

延迟暴露

如果你的服务需要预热时间，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。

延迟 5 秒暴露服务

```
<dubbo:service delay="5000" />
```

延迟到 Spring 初始化完成后，再暴露服务 [1]

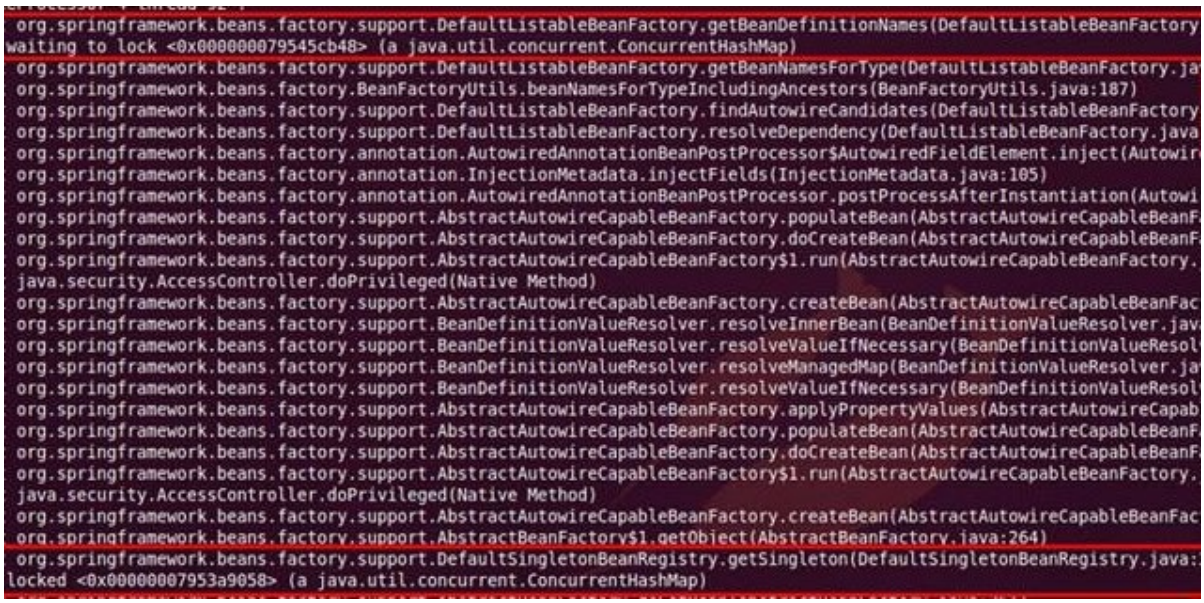
```
<dubbo:service delay="-1" />
```

Spring 2.x 初始化死锁问题

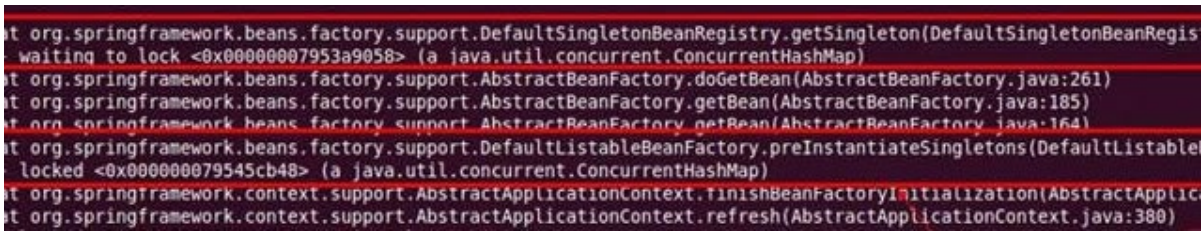
触发条件

在 Spring 解析到 `<dubbo:service />` 时，就已经向外暴露了服务，而 Spring 还在接着初始化其它 Bean。如果这时有请求进来，并且服务的实现类里有调用 `applicationContext.getBean()` 的用法。

1. 请求线程的 applicationContext.getBean() 调用，先同步 singletonObjects 判断 Bean 是否存在，不存在就同步 beanDefinitionMap 进行初始化，并再次同步 singletonObjects 写入 Bean 实例缓存。



2. 而 Spring 初始化线程，因不需要判断 Bean 的存在，直接同步 beanDefinitionMap 进行初始化，并同步 singletonObjects 写入 Bean 实例缓存。



这样就导致 getBean 线程，先锁 singletonObjects，再锁 beanDefinitionMap，再次锁 singletonObjects。
而 Spring 初始化线程，先锁 beanDefinitionMap，再锁 singletonObjects。反向锁导致线程死锁，不能提供服务，启动不了。

规避办法

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。

[1]. 基于 Spring 的 ContextRefreshedEvent 事件触发暴露 ↩

并发控制

配置样例

样例 1

限制 `com.foo.BarService` 的每个方法，服务器端并发执行（或占用线程池线程数）不能超过 10 个：

```
<dubbo:service interface="com.foo.BarService" executes="10" />
```

样例 2

限制 `com.foo.BarService` 的 `sayHello` 方法，服务器端并发执行（或占用线程池线程数）不能超过 10 个：

```
<dubbo:service interface="com.foo.BarService">
    <dubbo:method name="sayHello" executes="10" />
</dubbo:service>
```

样例 3

限制 `com.foo.BarService` 的每个方法，每客户端并发执行（或占用连接的请求数）不能超过 10 个：

```
<dubbo:service interface="com.foo.BarService" actives="10" />
```

或

```
<dubbo:reference interface="com.foo.BarService" actives="10" />
```

样例 4

限制 `com.foo.BarService` 的 `sayHello` 方法，每客户端并发执行（或占用连接的请求数）不能超过 10 个：

```
<dubbo:service interface="com.foo.BarService">
    <dubbo:method name="sayHello" actives="10" />
</dubbo:service>
```

或

```
<dubbo:reference interface="com.foo.BarService">
    <dubbo:method name="sayHello" actives="10" />
</dubbo:service>
```

如果 `<dubbo:service>` 和 `<dubbo:reference>` 都配了actives，`<dubbo:reference>` 优先，参见：配置的覆盖策略。

Load Balance 均衡

配置服务的客户端的 `loadbalance` 属性为 `leastactive`，此 Loadbalance 会调用并发数最小的 Provider（Consumer端并发数）。

```
<dubbo:reference interface="com.foo.BarService" loadbalance="leastactive" />
```

或

```
<dubbo:service interface="com.foo.BarService" loadbalance="leastactive" />
```

连接控制

服务端连接控制

限制服务器端接受的连接不能超过 10 个[1]：

```
<dubbo:provider protocol="dubbo" accepts="10" />
```

或

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

客户端连接控制

限制客户端服务使用连接不能超过 10 个[2]：

```
<dubbo:reference interface="com.foo.BarService" connections="10" />
```

或

```
<dubbo:service interface="com.foo.BarService" connections="10" />
```

如果 `<dubbo:service>` 和 `<dubbo:reference>` 都配了 connections，`<dubbo:reference>` 优先，参见：配置的覆盖策略

1. 因为连接在 Server上，所以配置在 Provider 上 ↩

2. 如果是长连接，比如 Dubbo 协议，connections 表示该服务对每个提供者建立的长连接数 ↩

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。[1]

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

1. 注意：该配置只对使用长连接的 dubbo 协议生效。 ↩

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

通过令牌验证在注册中心控制权限，以决定要不要下发令牌给消费者，可以防止消费者绕过注册中心访问提供者，另外通过注册中心可灵活改变授权方式，而不需修改或升级提供者

可以全局设置开启令牌验证：

```
<!--随机token令牌，使用UUID生成-->
<dubbo:provider interface="com.foo.BarService" token="true" />
```

或

```
<!--固定token令牌，相当于密码-->
<dubbo:provider interface="com.foo.BarService" token="123456" />
```

也可在服务级别设置：

```
<!--随机token令牌，使用UUID生成-->
<dubbo:service interface="com.foo.BarService" token="true" />
```

或

```
<!--固定token令牌，相当于密码-->
<dubbo:service interface="com.foo.BarService" token="123456" />
```

还可在协议级别设置：

```
<!--随机token令牌，使用UUID生成-->
<dubbo:protocol name="dubbo" token="true" />
```

或

```
<!--固定token令牌，相当于密码-->
<dubbo:protocol name="dubbo" token="123456" />
```

路由规则

路由规则[1]决定一次 dubbo 服务调用的目标服务器，分为条件路由规则和脚本路由规则，并且支持可扩展[2]。

写入路由规则

向注册中心写入路由规则的操作通常由监控中心或治理中心的页面完成

```
RegistryFactory registryFactory = ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptiveExtension();
Registry registry = registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:2181"));
registry.register(URL.valueOf("condition://0.0.0.0/com.foo.BarService?category=routers&dynamic=false&rule=" + URL.encode("host = 10.20.153.10 => host = 10.20.153.11") + "));
```

其中：

- `condition://` 表示路由规则的类型，支持条件路由规则和脚本路由规则，可扩展，必填。
- `0.0.0.0` 表示对所有 IP 地址生效，如果只想对某个 IP 的生效，请填入具体 IP，必填。
- `com.foo.BarService` 表示只对指定服务生效，必填。
- `category=routers` 表示该数据为动态配置类型，必填。
- `dynamic=false` 表示该数据为持久数据，当注册方退出时，数据依然保存在注册中心，必填。
- `enabled=true` 覆盖规则是否生效，可不填，缺省生效。
- `force=false` 当路由结果为空时，是否强制执行，如果不强制执行，路由结果为空的路由规则将自动失效，可不填，缺省为 `flase`。
- `runtime=false` 是否在每次调用时执行路由规则，否则只在提供者地址列表变更时预先执行并缓存结果，调用时直接从缓存中获取路由结果。如果用了参数路由，必须设为 `true`，需要注意设置会影响调用的性能，可不填，缺省为 `flase`。
- `priority=1` 路由规则的优先级，用于排序，优先级越大越靠前执行，可不填，缺省为 `0`。
- `rule=URL.encode("host = 10.20.153.10 => host = 10.20.153.11")` 表示路由规则的内容，必填。

条件路由规则

基于条件表达式的路由规则，如：`host = 10.20.153.10 => host = 10.20.153.11`

规则：

- `=>` 之前的为消费者匹配条件，所有参数和消费者的 URL 进行对比，当消费者满足匹配条件时，对该消费者执行后面的过滤规则。
- `=>` 之后为提供者地址列表的过滤条件，所有参数和提供者的 URL 进行对比，消费者最终只拿到过滤后的地址列表。
- 如果匹配条件为空，表示对所有消费方应用，如：`=> host != 10.20.153.11`
- 如果过滤条件为空，表示禁止访问，如：`host = 10.20.153.10 =>`

表达式：

参数支持：

- 服务调用信息，如：method, argument 等，暂不支持参数路由
- URL 本身的字段，如：protocol, host, port 等
- 以及 URL 上的所有参数，如：application, organization 等

条件支持：

- 等号 `=` 表示"匹配"，如：`host = 10.20.153.10`
- 不等号 `!=` 表示"不匹配"，如：`host != 10.20.153.10`

值支持：

- 以逗号 `,` 分隔多个值，如：`host != 10.20.153.10,10.20.153.11`
- 以星号 `*` 结尾，表示通配，如：`host != 10.20.*`
- 以美元符 `$` 开头，表示引用消费者参数，如：`host = $host`

示例：

1. 排除预发布机：

   ```
   => host != 172.22.3.91
   ```

2. 白名单[3]：

   ```
   host != 10.20.153.10,10.20.153.11 =>
   ```

3. 黑名单：

```
host = 10.20.153.10,10.20.153.11 =>
```

4. 服务寄宿在应用上，只暴露一部分的机器，防止整个集群挂掉：

```
=> host = 172.22.3.1*,172.22.3.2*
```

5. 为重要应用提供额外的机器：

```
application != kylin => host != 172.22.3.95,172.22.3.96
```

6. 读写分离：

```
method = find*,list*,get*,is* => host = 172.22.3.94,172.22.3.95,172.22.3.96
method != find*,list*,get*,is* => host = 172.22.3.97,172.22.3.98
```

7. 前后台分离：

```
application = bops => host = 172.22.3.91,172.22.3.92,172.22.3.93
application != bops => host = 172.22.3.94,172.22.3.95,172.22.3.96
```

8. 隔离不同机房网段：

```
host != 172.22.3.* => host != 172.22.3.*
```

9. 提供者与消费者部署在同集群内，本机只访问本机的服务：

```
=> host = $host
```

脚本路由规则

脚本路由规则[4]支持 JDK 脚本引擎的所有脚本，比如：javascript, jruby, groovy 等，通过 `type=javascript` 参数设置脚本类型，缺省为 javascript。

```
"script://0.0.0.0/com.foo.BarService?category=routers&dynamic=false&rule=" + URL.encod
e("function route(invokers) { ... } (invokers)")
```

基于脚本引擎的路由规则，如：

```
function route(invokers) {
    var result = new java.util.ArrayList(invokers.size());
    for (i = 0; i < invokers.size(); i ++) {
        if ("10.20.153.10".equals(invokers.get(i).getUrl().getHost())) {
            result.add(invokers.get(i));
        }
    }
    return result;
} (invokers); // 表示立即执行方法
```

1. `2.2.0` 以上版本支持 ↩
2. 路由规则扩展点：路由扩展 ↩
3. 注意：一个服务只能有一条白名单规则，否则两条规则交叉，就都被筛选掉了 ↩
4. 注意：脚本没有沙箱约束，可执行任意代码，存在后门风险 ↩

配置规则

向注册中心写入动态配置覆盖规则[1]。该功能通常由监控中心或治理中心的页面完成。

```
RegistryFactory registryFactory = ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptiveExtension();
Registry registry = registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:2181"));
registry.register(URL.valueOf("override://0.0.0.0/com.foo.BarService?category=configurators&dynamic=false&application=foo&timeout=1000"));
```

其中：

- `override://` 表示数据采用覆盖方式，支持 `override` 和 `absent`，可扩展，必填。
- `0.0.0.0` 表示对所有 IP 地址生效，如果只想覆盖某个 IP 的数据，请填入具体 IP，必填。
- `com.foo.BarService` 表示只对指定服务生效，必填。
- `category=configurators` 表示该数据为动态配置类型，必填。
- `dynamic=false` 表示该数据为持久数据，当注册方退出时，数据依然保存在注册中心，必填。
- `enabled=true` 覆盖规则是否生效，可不填，缺省生效。
- `application=foo` 表示只对指定应用生效，可不填，表示对所有应用生效。
- `timeout=1000` 表示将满足以上条件的 `timeout` 参数的值覆盖为 1000。如果想覆盖其它参数，直接加在 `override` 的 URL 参数上。

示例：

1. 禁用提供者：(通常用于临时踢除某台提供者机器，相似的，禁止消费者访问请使用路由规则)

   ```
   override://10.20.153.10/com.foo.BarService?category=configurators&dynamic=false&disbaled=true
   ```

2. 调整权重：(通常用于容量评估，缺省权重为 100)

   ```
   override://10.20.153.10/com.foo.BarService?category=configurators&dynamic=false&weight=200
   ```

3. 调整负载均衡策略：(缺省负载均衡策略为 random)

```
override://10.20.153.10/com.foo.BarService?category=configurators&dynamic=false&loadbalance=leastactive
```

4. 服务降级：(通常用于临时屏蔽某个出错的非关键服务)

```
override://0.0.0.0/com.foo.BarService?category=configurators&dynamic=false&application=foo&mock=force:return+null
```

1. `2.2.0` 以上版本支持 ↩

服务降级

可以通过服务降级功能[1]临时屏蔽某个出错的非关键服务，并定义降级后的返回策略。

向注册中心写入动态配置覆盖规则：

```
RegistryFactory registryFactory = ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptiveExtension();
Registry registry = registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:2181"));
registry.register(URL.valueOf("override://0.0.0.0/com.foo.BarService?category=configurators&dynamic=false&application=foo&mock=force:return+null"));
```

其中：

- `mock=force:return+null` 表示消费方对该服务的方法调用都直接返回 null 值，不发起远程调用。用来屏蔽不重要服务不可用时对调用方的影响。
- 还可以改为 `mock=fail:return+null` 表示消费方对该服务的方法调用在失败后，再返回 null 值，不抛异常。用来容忍不重要服务不稳定时对调用方的影响。

1. `2.2.0` 以上版本支持 ↩

优雅停机

Dubbo 是通过 JDK 的 ShutdownHook 来完成优雅停机的，所以如果用户使用 `kill -9 PID` 等强制关闭指令，是不会执行优雅停机的，只有通过 `kill PID` 时，才会执行。

原理

服务提供方

- 停止时，先标记为不接收新请求，新请求过来时直接报错，让客户端重试其它机器。
- 然后，检测线程池中的线程是否正在运行，如果有，等待所有线程执行完成，除非超时，则强制关闭。

服务消费方

- 停止时，不再发起新的调用请求，所有新的调用在客户端即报错。
- 然后，检测有没有请求的响应还没有返回，等待响应返回，除非超时，则强制关闭。

设置方式

设置优雅停机超时时间，缺省超时时间是 10 秒，如果超时则强制关闭。

```
<dubbo:application ...>
    <dubbo:parameter key="shutdown.timeout" value="60000" /> <!-- 单位毫秒 -->
</dubbo:application>
```

如果 ShutdownHook 不能生效，可以自行调用：

```
ProtocolConfig.destroyAll();
```

主机绑定

查找顺序

缺省主机 IP 查找顺序：

- 通过 `LocalHost.getLocalHost()` 获取本机地址。
- 如果是 `127.*` 等 loopback 地址，则扫描各网卡，获取网卡 IP。

主机配置

注册的地址如果获取不正确，比如需要注册公网地址，可以：

1. 可以在 `/etc/hosts` 中加入：机器名 公网 IP，比如：

```
test1 205.182.23.201
```

2. 在 `dubbo.xml` 中加入主机地址的配置：

```
<dubbo:protocol host="205.182.23.201">
```

3. 或在 `dubbo.properties` 中加入主机地址的配置：

```
dubbo.protocol.host=205.182.23.201
```

端口配置

缺省主机端口与协议相关：

协议	端口
dubbo	20880
rmi	1099
http	80
hessian	80
webservice	80
memcached	11211
redis	6379

可以按照下面的方式配置端口：

1. 在 `dubbo.xml` 中加入主机地址的配置：

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

2. 或在 `dubbo.properties` 中加入主机地址的配置：

```
dubbo.protocol.dubbo.port=20880
```

日志适配

自 `2.2.1` 开始，dubbo 开始内置 log4j、slf4j、jcl、jdk 这些日志框架的适配[1]，也可以通过以下方式显示配置日志输出策略：

1. 命令行

```
java -Ddubbo.application.logger=log4j
```

2. 在 `dubbo.properties` 中指定

```
dubbo.application.logger=log4j
```

3. 在 `dubbo.xml` 中配置

```
<dubbo:application logger="log4j" />
```

1. 自定义扩展可以参考日志适配扩展 ↩

访问日志

如果你想记录每一次请求信息，可开启访问日志，类似于apache的访问日志。注意：此日志量比较大，请注意磁盘容量。

将访问日志输出到当前应用的log4j日志：

```
<dubbo:protocol accesslog="true" />
```

将访问日志输出到指定文件：

```
<dubbo:protocol accesslog="http://10.20.160.198/wiki/display/dubbo/foo/bar.log" />
```

服务容器

服务容器是一个 standalone 的启动程序，因为后台服务不需要 Tomcat 或 JBoss 等 Web 容器的功能，如果硬要用 Web 容器去加载服务提供方，增加复杂性，也浪费资源。

服务容器只是一个简单的 Main 方法，并加载一个简单的 Spring 容器，用于暴露服务。

服务容器的加载内容可以扩展，内置了 spring, jetty, log4j 等加载，可通过容器扩展点进行扩展。配置配在 java 命令的 -D 参数或者 `dubbo.properties` 中。

容器类型

Spring Container

- 自动加载 `META-INF/spring` 目录下的所有 Spring 配置。
- 配置 spring 配置加载位置：

```
dubbo.spring.config=classpath*:META-INF/spring/*.xml
```

Jetty Container

- 启动一个内嵌 Jetty，用于汇报状态。
- 配置：
  - `dubbo.jetty.port=8080` ：配置 jetty 启动端口
  - `dubbo.jetty.directory=/foo/bar` ：配置可通过 jetty 直接访问的目录，用于存放静态文件
  - `dubbo.jetty.page=log,status,system` ：配置显示的页面，缺省加载所有页面

Log4j Container

- 自动配置 log4j 的配置，在多进程启动时，自动给日志文件按进程分目录。
- 配置：
  - `dubbo.log4j.file=/foo/bar.log` ：配置日志文件路径
  - `dubbo.log4j.level=WARN` ：配置日志级别
  - `dubbo.log4j.subdirectory=20880` ：配置日志子目录，用于多进程启动，避免冲突

容器启动

缺省只加载 spring

```
java com.alibaba.dubbo.container.Main
```

通过 main 函数参数传入要加载的容器

```
java com.alibaba.dubbo.container.Main spring jetty log4j
```

通过 JVM 启动参数传入要加载的容器

```
java com.alibaba.dubbo.container.Main -Ddubbo.container=spring,jetty,log4j
```

通过 classpath 下的 `dubbo.properties` 配置传入要加载的容器

```
dubbo.container=spring,jetty,log4j
```

ReferenceConfig 缓存

`ReferenceConfig` 实例很重，封装了与注册中心的连接以及与提供者的连接，需要缓存。否则重复生成 `ReferenceConfig` 可能造成性能问题并且会有内存和连接泄漏。在 API 方式编程时，容易忽略此问题。

因此，自 `2.4.0` 版本开始，dubbo 提供了简单的工具类 `ReferenceConfigCache` 用于缓存 `ReferenceConfig` 实例。

使用方式如下：

```
ReferenceConfig<XxxService> reference = new ReferenceConfig<XxxService>();
reference.setInterface(XxxService.class);
reference.setVersion("1.0.0");
......
ReferenceConfigCache cache = ReferenceConfigCache.getCache();
// cache.get方法中会缓存 Reference对象，并且调用ReferenceConfig.get方法启动ReferenceConfig
XxxService xxxService = cache.get(reference);
// 注意！ Cache会持有ReferenceConfig，不要在外部再调用ReferenceConfig的destroy方法，导致Cache
内的ReferenceConfig失效！
// 使用xxxService对象
xxxService.sayHello();
```

消除 Cache 中的 `ReferenceConfig`，将销毁 `ReferenceConfig` 并释放对应的资源。

```
ReferenceConfigCache cache = ReferenceConfigCache.getCache();
cache.destroy(reference);
```

缺省 `ReferenceConfigCache` 把相同服务 Group、接口、版本的 `ReferenceConfig` 认为是相同，缓存一份。即以服务 Group、接口、版本为缓存的 Key。

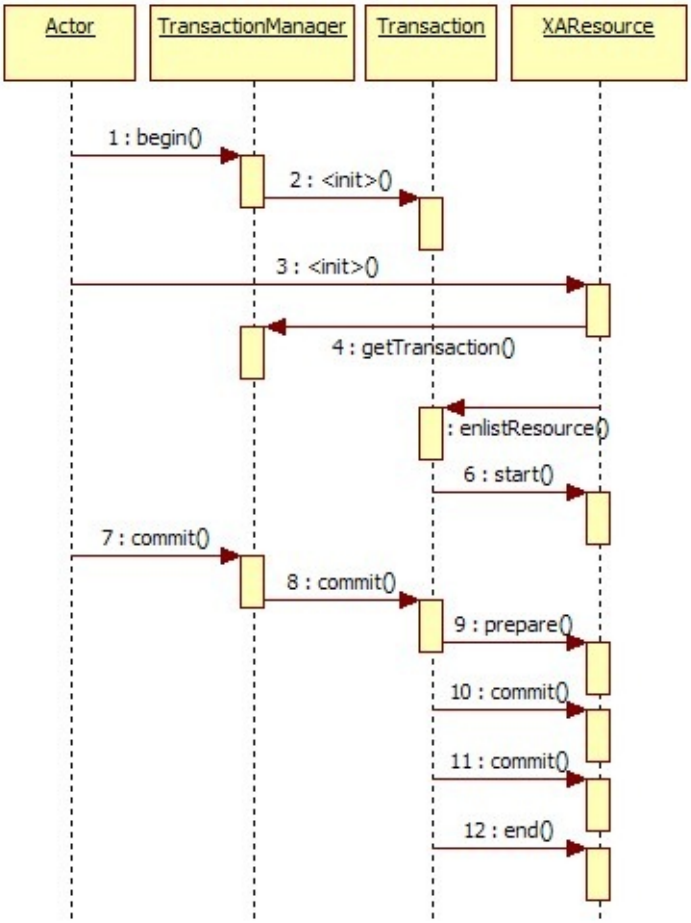
可以修改这个策略，在 `ReferenceConfigCache.getCache` 时，传一个 `KeyGenerator`。详见 `ReferenceConfigCache` 类的方法。

```
KeyGenerator keyGenerator = new ...
ReferenceConfigCache cache = ReferenceConfigCache.getCache(keyGenerator );
```

分布式事务

分布式事务基于 JTA/XA 规范实现[1]。

两阶段提交：



1. 本功能暂未实现 ↩

API 参考手册

Dubbo 的常规功能，都保持零侵入，但有些功能不得不用 API 侵入才能实现[1]。

API 汇总如下：

配置 API

```
com.alibaba.dubbo.config.ServiceConfig
com.alibaba.dubbo.config.ReferenceConfig
com.alibaba.dubbo.config.ProtocolConfig
com.alibaba.dubbo.config.RegistryConfig
com.alibaba.dubbo.config.MonitorConfig
com.alibaba.dubbo.config.ApplicationConfig
com.alibaba.dubbo.config.ModuleConfig
com.alibaba.dubbo.config.ProviderConfig
com.alibaba.dubbo.config.ConsumerConfig
com.alibaba.dubbo.config.MethodConfig
com.alibaba.dubbo.config.ArgumentConfig
```

详细参见：API配置

注解 API

```
com.alibaba.dubbo.config.annotation.Service
com.alibaba.dubbo.config.annotation.Reference
```

详细参见：注解配置

模型 API

```
com.alibaba.dubbo.common.URL
com.alibaba.dubbo.rpc.RpcException
```

上下文 API

```
com.alibaba.dubbo.rpc.RpcContext
```

详细参见：上下文信息 & 隐式传参 & 异步调用

服务API

```
com.alibaba.dubbo.rpc.service.GenericService
com.alibaba.dubbo.rpc.service.GenericException
```

详细参见：泛化引用 & 泛化实现

```
com.alibaba.dubbo.rpc.service.EchoService
```

详细参见：回声测试

> 1. 注意：Dubbo 中除这里声明以外的接口或类，都是内部接口或扩展接口，普通用户请不要直接依赖，否则升级版本可能出现不兼容。 ↩

schema 配置参考手册

这里以 XML Config [1] 为准，列举所有配置项 [2]。其它配置方式，请参见相应转换关系：属性配置，注解配置，API 配置。

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 [3] 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。

> 1. XML Schema: http://code.alibabatech.com/schema/dubbo/dubbo.xsd ↩

> 2. 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。 ↩

> 3. URL 格式：`protocol://username:password@host:port/path?key=value&key=value` ↩

dubbo:service

服务提供者暴露服务配置。对应的配置类：`com.alibaba.dubbo.config.ServiceConfig`

属性	对应URL参数	类型	是否必填	缺省值	作用	
interface		class	必填		服务发现	服务接口名
ref		object	必填		服务发现	服务对象实
version	version	string	可选	0.0.0	服务发现	服务版本，如：1.0，通才需要升级
group	group	string	可选		服务发现	服务分组，可以用分组
path	<path>	string	可选	缺省为接口名	服务发现	服务路径 (径，总是使2.0，配置服
delay	delay	int	可选	0	性能调优	延迟注册服时，表示延成时暴露服
timeout	timeout	int	可选	1000	性能调优	远程服务调
retries	retries	int	可选	2	性能调优	远程服务调次调用，不
connections	connections	int	可选	100	性能调	对每个提供http、hessia连接数，du

					优	长连接个数
loadbalance	loadbalance	string	可选	random	性能调优	负载均衡策略 random,roun 表示：随机
async	async	boolean	可选	false	性能调优	是否缺省异 是忽略返回
stub	stub	class/boolean	可选	false	服务治理	设为true，表 即：接口名 户端本地代 行本地逻辑 代理类的构 代理对象，XxxServiceL xxxService)
mock	mock	class/boolean	可选	false	服务治理	设为true，表 即：接口名 用失败Mock 一个无参构 于，Local总 现非业务异 时执行，Lo Mock在远程
token	token	string/boolean	可选	false	服务治理	令牌验证，为 true，表示随 用静态令牌 者绕过注册 中心的授权 点调用，需
registry		string	可选	缺省向所有registry注册	配置关联	向指定注册 心时使用，为 属性，多个 如果不想将 registry，可
provider		string	可选	缺使用第一个provider配置	配置关联	指定provide 的id属性
deprecated	deprecated	boolean	可选	false	服务治理	服务是否过时 引用时将打
dynamic	dynamic	boolean	可选	true	服务治	服务是否动态 注册后将显 启用，并且

dynamic	dynamic	boolean	选	true	治理	启用，并且 会自动取消
accesslog	accesslog	string/boolean	可选	false	服务治理	设为true，将 志，也可填 接把访问日
owner	owner	string	可选		服务治理	服务负责人 负责人公司
document	document	string	可选		服务治理	服务文档UR
weight	weight	int	可选		性能调优	服务权重
executes	executes	int	可选	0	性能调优	服务提供者 执行请求数
proxy	proxy	string	可选	javassist	性能调优	生成动态代 jdk/javassis
cluster	cluster	string	可选	failover	性能调优	集群方式， failover/failf
			可		性能	服务提供方

					优	
listener	exporter.listener	string	可选	default	性能调优	服务提供方 个名称用逗
protocol		string	可选		配置关联	使用指定的 时使用，值 性，多个协
layer	layer	string	可选		服务治理	服务提供者 dao、intl:we
register	register	boolean	可选	true	服务治理	该协议的服

dubbo:reference

服务消费者引用服务配置。对应的配置类： `com.alibaba.dubbo.config.ReferenceConfig`

属性	对应URL参数	类型	是否必填	缺省值	作用	
id		string	必填		配置关联	服
interface		class	必填		服务发现	服
version	version	string	可选		服务发现	服
group	group	string	可选		服务发现	服可一
timeout	timeout	long	可选	缺省使用<dubbo:consumer>的timeout	性能调优	服
retries	retries	int	可选	缺省使用<dubbo:consumer>的retries	性能调优	远次
connections	connections	int	可选	缺省使用<dubbo:consumer>的connections	性能调优	对ht连长
loadbalance	loadbalance	string	可选	缺省使用<dubbo:consumer>的loadbalance	性能调优	负ra表
async	async	boolean	可选	缺省使用<dubbo:consumer>	性能调	是略

					优	
generic	generic	boolean	可选	缺省使用<dubbo:consumer>的generic	服务治理	是口
check	check	boolean	可选	缺省使用<dubbo:consumer>的check	服务治理	启错
url	url	string	可选		服务治理	点注
stub	stub	class/boolean	可选		服务治理	服在等许pu xx
mock	mock	class/boolean	可选		服务治理	服M Lo而时调行
cache	cache	string/boolean	可选		服务治理	以选
validation	validation	boolean	可选		服务治理	是启验
proxy	proxy	boolean	可选	javassist	性能调优	选ja
client	client	string	可选		性能调优	客的
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	从在<d中

					联	中
owner	owner	string	可选		服务治理	调填
actives	actives	int	可选	0	性能调优	每调
cluster	cluster	string	可选	failover	性能调优	集fa
filter	reference.filter	string	可选	default	性能调优	服称
listener	invoker.listener	string	可选	default	性能调优	服个
layer	layer	string	可选		服务治理	服da
init	init	boolean	可选	false	性能调优	是化实
protocol	protocol	string	可选		服力治理	只协

dubbo:protocol

服务提供者协议配置。对应的配置类： `com.alibaba.dubbo.config.ProtocolConfig` 。同时，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。

属性	对应URL参数	类型	是否必填	缺省值	作用	
id		string	可选	dubbo	配置关联	协议Bean protocol= 填，缺省 在name
name	<protocol>	string	必填	dubbo	性能调优	协议名称
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
host	<host>	string	可选	自动查找本机IP	服务发现	-服务主机 域名时使 IP，-建议 取本机IP
threadpool	threadpool	string	可选	fixed	性能调优	线程池类
threads	threads	int	可	100	性能	服务线程

			选		调优	
iothreads	threads	int	可选	cpu个数+1	性能调优	io线程池
accepts	accepts	int	可选	0	性能调优	服务提供
payload	payload	int	可选	88388608(=8M)	性能调优	请求及响节
codec	codec	string	可选	dubbo	性能调优	协议编码
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	协议序列化方式时dubbo,he以及http
accesslog	accesslog	string/boolean	可选		服务治理	设为true志，也可把访问日
path	<path>	string	可选		服务发现	提供者上
transporter	transporter	string	可选	dubbo协议缺省为netty	性能调优	协议的服如：dubb分拆为se
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dubbo协jetty,serv
client	client	string	可选	dubbo协议缺省为netty	性能调优	协议的客协议的m
dispatcher	dispatcher	string	可	dubbo协议缺省	性能	协议的消型，比如

dispatcher	dispatcher	string	选	为all	调优	message
queues	queues	int	可选	0	性能调优	线程池队等待执行当线程程务提供机需求。
charset	charset	string	可选	UTF-8	性能调优	序列化编
buffer	buffer	int	可选	8192	性能调优	网络读写
heartbeat	heartbeat	int	可选	0	性能调优	心跳间隔时，比如及发送，要心跳来
telnet	telnet	string	可选		服务治理	所支持的分隔
register	register	boolean	可选	true	服务治理	该协议的
contextpath	contextpath	String	可选	缺省为空串	服务治理	

dubbo:registry

注册中心配置。对应的配置类： `com.alibaba.dubbo.config.RegistryConfig` 。同时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。

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

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
transport	registry.transporter	string	可选	netty	性能调优	网络传输方式，可选mina,netty
timeout	registry.timeout	int	可选	5000	性能调优	注册中心请求超时时间(毫秒)
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。
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
wait	registry.wait	int	可选	0	性能调优	停止时等待通知完成时间(毫秒)
check	check	boolean	可选	true	服务治理	注册中心不存在时，是否报错
register	register	boolean	可选	true	服务治理	是否向此注册中心注册服务，如果设为false，将只订阅，不注册
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
						服务是否动态注

dynamic	dynamic	boolean	可选	true	服务治理	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。

dubbo:monitor

监控中心配置。对应的配置类： `com.alibaba.dubbo.config.MonitorConfig`

属性	对应URL参数	类型	是否必填	缺省值	作用	描述
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。
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"

dubbo:application

应用信息配置。对应的配置类：`com.alibaba.dubbo.config.ApplicationConfig`

属性	对应URL参数	类型	是否必填	缺省值	作用	描
name	application	string	必填		服务治理	当前应用名称，用应用间依赖关系，提供者应用名不要是匹配条件，你当字就填什么，和提无关，比如：kylir morgan应用的服务配成kylin，morga morgan，可能kyli 务给别人使用，但成kylin，这样注册依赖于morgan
version	application.version	string	可选		服务治理	当前应用的版本
owner	owner	string	可选		服务治理	应用负责人，用于写负责人公司邮箱
organization	organization	string	可选		服务治理	组织名称(BU或部心区分服务来源，要使用autoconfig置中，比如 china,intl,itu,crm,a 等
architecture	architecture	string	可选		服务治理	用于服务分层对应intl、china。不同的分层。
environment	environment	string	可选		服务治理	应用环境，如：develop/test/prod 用不同的缺省值，开发测试功能的限
compiler	compiler	string	可选	javassist	性能优化	Java字节码编译器生成，可选：jdk或
logger	logger	string	可选	slf4j	性能优化	日志输出方式，可slf4j,jcl,log4j,jdk

dubbo:module

模块信息配置。对应的配置类 `com.alibaba.dubbo.config.ModuleConfig`

属性	对应URL参数	类型	是否必填	缺省值	作用	描述
name	module	string	必填		服务治理	当前模块名称，用于注册中 模块间依赖关系
version	module.version	string	可选		服务治理	当前模块的版本
owner	owner	string	可选		服务治理	模块负责人，用于服务治理 写负责人公司邮箱前缀
organization	organization	string	可选		服务治理	组织名称(BU或部门)，用于 心区分服务来源，此配置项 要使用autoconfig，直接写死 置中，比如 china,intl,itu,crm,asc,dw,alie 等

dubbo:provider

服务提供者缺省值配置。对应的配置类： `com.alibaba.dubbo.config.ProviderConfig` 。同时该标签为 `<dubbo:service>` 和 `<dubbo:protocol>` 标签的缺省值设置。

属性	对应URL参数	类型	是否必填	缺省值
id		string	可选	dubbo
protocol	<protocol>	string	可选	dubbo
host	<host>	string	可选	自动查找本机IP
threads	threads	int	可选	100
payload	payload	int	可选	88388608(=8M)
path	<path>	string	可选	
server	server	string	可选	dubbo协议缺省为netty，http协议缺省为servlet
client	client	string	可选	dubbo协议缺省为netty
codec	codec	string	可选	dubbo

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
default		boolean	可选	false
filter	service.filter	string	可选	
listener	exporter.listener	string	可选	
threadpool	threadpool	string	可选	fixed
accepts	accepts	int	可选	0
version	version	string	可选	0.0.0
group	group	string	可选	
delay	delay	int	可选	0
timeout	default.timeout	int	可选	1000
retries	default.retries	int	可选	2

connections	default.connections	int	可选	0
loadbalance	default.loadbalance	string	可选	random
async	default.async	boolean	可选	false
stub	stub	boolean	可选	false
mock	mock	boolean	可选	false
token	token	boolean	可选	false
registry	registry	string	可选	缺省向所有registry注册
dynamic	dynamic	boolean	可选	true
accesslog	accesslog	string/boolean	可选	false
owner	owner	string	可选	
document	document	string	可选	

weight	weight	int	可选	
executes	executes	int	可选	0
actives	default.actives	int	可选	0
proxy	proxy	string	可选	javassist
cluster	default.cluster	string	可选	failover
deprecated	deprecated	boolean	可选	false
queues	queues	int	可选	0
charset	charset	string	可选	UTF-8
buffer	buffer	int	可选	8192
iothreads	iothreads	int	可选	CPU + 1
telnet	telnet	string	可选	

<dubbo:service>	contextpath	contextpath	String	可选
layer	layer	string	可选	

dubbo:consumer

服务消费者缺省值配置。配置类： `com.alibaba.dubbo.config.ConsumerConfig` 。同时该标签为 `<dubbo:reference>` 标签的缺省值设置。

属性	对应URL参数	类型	是否必填	缺省值	作用	
timeout	default.timeout	int	可选	1000	性能调优	远程服务
retries	default.retries	int	可选	2	性能调优	远程服务次调用，
loadbalance	default.loadbalance	string	可选	random	性能调优	负载均衡 random, 表示：随
async	default.async	boolean	可选	false	性能调优	是否缺省是忽略返
connections	default.connections	int	可选	100	性能调优	每个服务数，rmi 支持此配此配置
generic	generic	boolean	可选	false	服务治理	是否缺省口，将返
check	check	boolean	可选	true	服务治理	启动时检错，fals
proxy	proxy	string	可选	javassist	性能调优	生成动态 jdk/javas
			可		服务	调用服务

owner	owner	string	选		治理	填写负责
actives	default.actives	int	可选	0	性能调优	每服务消调用数
cluster	default.cluster	string	可选	failover	性能调优	集群方式failover/f
filter	reference.filter	string	可选		性能调优	服务消费称，多个
listener	invoker.listener	string	可选		性能调优	服务消费个名称用
registry		string	可选	缺省向所有registry注册	配置关联	向指定注心时使用属性，多如果不想registry
layer	layer	string	可选		服务治理	服务调用dao、int
init	init	boolean	可选	false	性能调优	是否在a化引用，实例时再
cache	cache	string/boolean	可选		服务治理	以调用参选：lru,
validation	validation	boolean	可选		服务治理	是否启用启用，将验

dubbo:method

方法级配置。对应的配置类： `com.alibaba.dubbo.config.MethodConfig` 。同时该标签为 `<dubbo:service>` 或 `<dubbo:reference>` 的子标签，用于控制到方法级。

属性	对应URL参数	类型	是否必填	缺省值
name		string	必填	
timeout	<metodName>.timeout	int	可选	缺省为的timeout
retries	<metodName>.retries	int	可选	缺省为<dubbo:reference>的retries
loadbalance	<metodName>.loadbalance	string	可选	缺省为的loadbalance
async	<metodName>.async	boolean	可选	缺省为<dubbo:reference>的async
sent	<methodName>.sent	boolean	可选	true
actives	<metodName>.actives	int	可选	0
executes	<metodName>.executes	int	可选	0
deprecated	<methodName>.deprecated	boolean	可选	false

sticky	<methodName>.sticky	boolean	可选	false
return	<methodName>.return	boolean	可选	true
oninvoke	attribute属性，不在URL中体现	String	可选	
onreturn	attribute属性，不在URL中体现	String	可选	
onthrow	attribute属性，不在URL中体现	String	可选	
cache	<methodName>.cache	string/boolean	可选	
validation	<methodName>.validation	boolean	可选	

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。对应的配置类： `com.alibaba.dubbo.config.ArgumentConfig` 。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>
```

属性	对应URL参数	类型	是否必填	缺省值	作用	描述	兼容性
index		int	必填		标识	方法名	2.0.6以上版本
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
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。对应的配置类：`java.util.Map`。同时该标签为`<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。

属性	对应URL参数	类型	是否必填	缺省值	作用	描述	兼容性
key	key	string	必填		服务治理	路由参数键	2.0.0以上版本
value	value	string	必填		服务治理	路由参数值	2.0.0以上版本

比如：

```
<dubbo:protocol name="napoli">
    <dubbo:parameter key="http://10.20.160.198/wiki/display/dubbo/napoli.queue.name" v
alue="xxx" />
</dubbo:protocol>
```

也可以：

```
<dubbo:protocol name="jms" p:queue="xxx" />
```

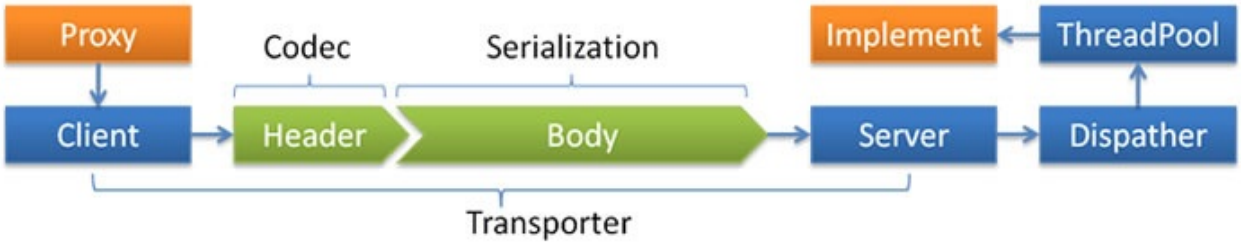
协议参考手册

推荐使用 Dubbo 协议。各协议的性能情况，请参见：性能测试报告

dubbo://

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。

反之，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。



- Transporter: mina, netty, grizzy
- Serialization: dubbo, hessian2, java, json
- Dispatcher: all, direct, message, execution, connection
- ThreadPool: fixed, cached

特性

缺省协议，使用基于 mina `1.1.7` 和 hessian `3.2.1` 的 tbremoting 交互。

- 连接个数：单连接
- 连接方式：长连接
- 传输协议：TCP
- 传输方式：NIO 异步传输
- 序列化：Hessian 二进制序列化
- 适用范围：传入传出参数数据包较小（建议小于100K），消费者比提供者个数多，单一消费者无法压满提供者，尽量不要用 dubbo 协议传输大文件或超大字符串。
- 适用场景：常规远程服务方法调用

约束

- 参数及返回值需实现 `Serializable` 接口
- 参数及返回值不能自定义实现 `List`，`Map`，`Number`，`Date`，`Calendar` 等接口，只能用 JDK 自带的实现，因为 hessian 会做特殊处理，自定义实现类中的属性值都会丢失。
- Hessian 序列化，只传成员属性值和值的类型，不传方法或静态变量，兼容情况[12]：

数据通讯	情况	结果
A->B	类A多一种 属性（或者说类B少一种 属性）	不抛异常，A多的那 个属性的值，B没有， 其他正常
A->B	枚举A多一种 枚举（或者说B少一种 枚举），A使用多 出来的枚举进行传输	抛异常
A->B	枚举A多一种 枚举（或者说B少一种 枚举），A不使用 多出来的枚举进行传输	不抛异常，B正常接 收数据
A->B	A和B的属性 名相同，但类型不相同	抛异常
A->B	serialId 不相同	正常传输

接口增加方法，对客户端无影响，如果该方法不是客户端需要的，客户端不需要重新部署。输入参数和结果集中增加属性，对客户端无影响，如果客户端并不需要新属性，不用重新部署。

输入参数和结果集属性名变化，对客户端序列化无影响，但是如果客户端不重新部署，不管输入还是输出，属性名变化的属性值是获取不到的。

总结：服务器端和客户端对领域对象并不需要完全一致，而是按照最大匹配原则。

配置

配置协议：

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

设置默认协议：

```
<dubbo:provider protocol="dubbo" />
```

设置服务协议：

```
<dubbo:service protocol="dubbo" />
```

多端口：

```
<dubbo:protocol id="dubbo1" name="dubbo" port="20880" />
<dubbo:protocol id="dubbo2" name="dubbo" port="20881" />
```

配置协议选项：

```
<dubbo:protocol name="dubbo" port="9090" server="netty" client="netty" codec="dubbo" s
erialization="hessian2" charset="UTF-8" threadpool="fixed" threads="100" queues="0" io
threads="9" buffer="8192" accepts="1000" payload="8388608" />
```

多连接配置：

Dubbo 协议缺省每服务每提供者每消费者使用单一长连接，如果数据量较大，可以使用多个连接。

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

- `<dubbo:service connections="0">` 或 `<dubbo:reference connections="0">` 表示该服务使用 JVM 共享长连接。缺省
- `<dubbo:service connections="1">` 或 `<dubbo:reference connections="1">` 表示该服务使用独立长连接。
- `<dubbo:service connections="2">` 或 `<dubbo:reference connections="2">` 表示该服务使用独立两条长连接。

为防止被大量连接撑挂，可在服务提供方限制大接收连接数，以实现服务提供方自我保护。

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

`dubbo.properties` 配置：

```
dubbo.service.protocol=dubbo
```

常见问题

为什么要消费者比提供者个数多?

因 dubbo 协议采用单一长连接，假设网络为千兆网卡[3]，根据测试经验数据每条连接最多只能压满 7MByte(不同的环境可能不一样，供参考)，理论上 1 个服务提供者需要 20 个服务消费者才能压满网卡。

为什么不能传大包?

因 dubbo 协议采用单一长连接，如果每次请求的数据包大小为 500KByte，假设网络为千兆网卡[3]，每条连接最大 7MByte(不同的环境可能不一样，供参考)，单个服务提供者的 TPS(每秒处理事务数)最大为：128MByte / 500KByte = 262。单个消费者调用单个服务提供者的 TPS(每秒处理事务数)最大为：7MByte / 500KByte = 14。如果能接受，可以考虑使用，否则网络将成为瓶颈。

为什么采用异步单一长连接?

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

> [1]. 由吴亚军提供 ↩
>
> [2]. 总结：会抛异常的情况：枚举值一边多一种，一边少一种，正好使用了差别的那种，或者属性名相同，类型不同 ↩
>
> [3]. 1024Mbit=128MByte ↩

rmi://

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。

注意：如果正在使用 RMI 提供服务给外部访问 [1]，同时应用里依赖了老的 common-collections 包 [2] 的情况下，存在反序列化安全风险 [3]。

特性

- 连接个数：多连接
- 连接方式：短连接
- 传输协议：TCP
- 传输方式：同步传输
- 序列化：Java 标准二进制序列化
- 适用范围：传入传出参数数据包大小混合，消费者与提供者个数差不多，可传文件。
- 适用场景：常规远程服务方法调用，与原生RMI服务互操作

约束

- 参数及返回值需实现 `Serializable` 接口
- dubbo 配置中的超时时间对 RMI 无效，需使用 java 启动参数设置：`-Dsun.rmi.transport.tcp.responseTimeout=3000`，参见下面的 RMI 配置

dubbo.properties 配置

```
dubbo.service.protocol=rmi
```

RMI配置

```
java -Dsun.rmi.transport.tcp.responseTimeout=3000
```

更多 RMI 优化参数请查看 JDK 文档

接口

如果服务接口继承了 `java.rmi.Remote` 接口，可以和原生 RMI 互操作，即：

- 提供者用 Dubbo 的 RMI 协议暴露服务，消费者直接用标准 RMI 接口调用，
- 或者提供方用标准 RMI 暴露服务，消费方用 Dubbo 的 RMI 协议调用。

如果服务接口没有继承 `java.rmi.Remote` 接口：

- 缺省 Dubbo 将自动生成一个 `com.xxx.XxxService$Remote` 的接口，并继承 `java.rmi.Remote` 接口，并以此接口暴露服务，
- 但如果设置了 `<dubbo:protocol name="rmi" codec="spring" />`，将不生成 `$Remote` 接口，而使用 Spring 的 `RmiInvocationHandler` 接口暴露服务，和 Spring 兼容。

配置

定义 RMI 协议：

```
<dubbo:protocol name="rmi" port="1099" />
```

设置默认协议：

```
<dubbo:provider protocol="rmi" />
```

设置服务协议：

```
<dubbo:service protocol="rmi" />
```

多端口：

```
<dubbo:protocol id="rmi1" name="rmi" port="1099" />
<dubbo:protocol id="rmi2" name="rmi" port="2099" />

<dubbo:service protocol="rmi1" />
```

Spring 兼容性：

```
<dubbo:protocol name="rmi" codec="spring" />
```

1. 公司内网环境应该不会有攻击风险 ↩

2

[2]. dubbo 不会依赖这个包，请排查自己的应用有没有使用 ↩

[3]. 请检查应用：将 commons-collections3 请升级到 3.2.2；将 commons-collections4 请升级到 4.1。新版本的 commons-collections 解决了该问题 ↩

hessian://

Hessian [1] 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。

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

- 提供者用 Dubbo 的 Hessian 协议暴露服务，消费者直接用标准 Hessian 接口调用
- 或者提供方用标准 Hessian 暴露服务，消费方用 Dubbo 的 Hessian 协议调用。

特性

- 连接个数：多连接
- 连接方式：短连接
- 传输协议：HTTP
- 传输方式：同步传输
- 序列化：Hessian二进制序列化
- 适用范围：传入传出参数数据包较大，提供者比消费者个数多，提供者压力较大，可传文件。
- 适用场景：页面传输，文件传输，或与原生hessian服务互操作

依赖

```
<dependency>
    <groupId>com.caucho</groupId>
    <artifactId>hessian</artifactId>
    <version>4.0.7</version>
</dependency>
```

约束

- 参数及返回值需实现 `Serializable` 接口
- 参数及返回值不能自定义实现 `List` , `Map` , `Number` , `Date` , `Calendar` 等接口，只能用 JDK 自带的实现，因为 hessian 会做特殊处理，自定义实现类中的属性值都会丢失。

配置

定义 hessian 协议：

```
<dubbo:protocol name="hessian" port="8080" server="jetty" />
```

设置默认协议：

```
<dubbo:provider protocol="hessian" />
```

设置 service 协议：

```
<dubbo:service protocol="hessian" />
```

多端口：

```
<dubbo:protocol id="hessian1" name="hessian" port="8080" />
<dubbo:protocol id="hessian2" name="hessian" port="8081" />
```

直连：

```
<dubbo:reference id="helloService" interface="HelloWorld" url="hessian://10.20.153.10:
8080/helloWorld" />
```

> [1]. Hessian 是 Caucho 开源的一个 RPC 框架，其通讯效率高于 WebService 和 Java 自带的序列化。 ↩

http://

基于 HTTP 表单的远程调用协议，采用 Spring 的 HttpInvoker 实现 [1]

特性

- 连接个数：多连接
- 连接方式：短连接
- 传输协议：HTTP
- 传输方式：同步传输
- 序列化：表单序列化
- 适用范围：传入传出参数数据包大小混合，提供者比消费者个数多，可用浏览器查看，可用表单或URL传入参数，暂不支持传文件。
- 适用场景：需同时给应用程序和浏览器 JS 使用的服务。

约束

- 参数及返回值需符合 Bean 规范

配置

配置协议：

```
<dubbo:protocol name="http" port="8080" />
```

配置 Jetty Server (默认)：

```
<dubbo:protocol ... server="jetty" />
```

配置 Servlet Bridge Server (推荐使用)：

```
<dubbo:protocol ... server="servlet" />
```

配置 DispatcherServlet：

```
<servlet>
        <servlet-name>dubbo</servlet-name>
        <servlet-class>com.alibaba.dubbo.remoting.http.servlet.DispatcherServlet</ser
vlet-class>
        <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
        <servlet-name>dubbo</servlet-name>
        <url-pattern>/*</url-pattern>
</servlet-mapping>
```

注意，如果使用 servlet 派发请求：

- 协议的端口 `<dubbo:protocol port="8080" />` 必须与 servlet 容器的端口相同，
- 协议的上下文路径 `<dubbo:protocol contextpath="foo" />` 必须与 servlet 应用的上下文路径相同。

1. `2.3.0` 以上版本支持 ↩

webservice://

基于 WebService 的远程调用协议，基于 Apache CXF [1] 的 `frontend-simple` 和 `transports-http` 实现 [2]。

可以和原生 WebService 服务互操作，即：

- 提供者用 Dubbo 的 WebService 协议暴露服务，消费者直接用标准 WebService 接口调用，
- 或者提供方用标准 WebService 暴露服务，消费方用 Dubbo 的 WebService 协议调用。

依赖

```
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-frontend-simple</artifactId>
    <version>2.6.1</version>
</dependency>
<dependency>
    <groupId>org.apache.cxf</groupId>
    <artifactId>cxf-rt-transports-http</artifactId>
    <version>2.6.1</version>
</dependency>
```

特性

- 连接个数：多连接
- 连接方式：短连接
- 传输协议：HTTP
- 传输方式：同步传输
- 序列化：SOAP 文本序列化
- 适用场景：系统集成，跨语言调用

约束

- 参数及返回值需实现 `Serializable` 接口
- 参数尽量使用基本类型和 POJO

配置

配置协议：

```
<dubbo:protocol name="webservice" port="8080" server="jetty" />
```

配置默认协议：

```
<dubbo:provider protocol="webservice" />
```

配置服务协议：

```
<dubbo:service protocol="webservice" />
```

多端口：

```
<dubbo:protocol id="webservice1" name="webservice" port="8080" />
<dubbo:protocol id="webservice2" name="webservice" port="8081" />
```

直连：

```
<dubbo:reference id="helloService" interface="HelloWorld" url="webservice://10.20.153.
10:8080/com.foo.HelloWorld" />
```

WSDL：

```
http://10.20.153.10:8080/com.foo.HelloWorld?wsdl
```

Jetty Server (默认)：

```
<dubbo:protocol ... server="jetty" />
```

Servlet Bridge Server (推荐)：

```
<dubbo:protocol ... server="servlet" />
```

配置 DispatcherServlet：

```
<servlet>
        <servlet-name>dubbo</servlet-name>
        <servlet-class>com.alibaba.dubbo.remoting.http.servlet.DispatcherServlet</ser
vlet-class>
        <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
        <servlet-name>dubbo</servlet-name>
        <url-pattern>/*</url-pattern>
</servlet-mapping>
```

注意，如果使用 servlet 派发请求：

- 协议的端口 `<dubbo:protocol port="8080" />` 必须与 servlet 容器的端口相同，
- 协议的上下文路径 `<dubbo:protocol contextpath="foo" />` 必须与 servlet 应用的上下文路径相同。

1. CXF 是 Apache 开源的一个 RPC 框架，由 Xfire 和 Celtix 合并而来 ↩

2. `2.3.0` 以上版本支持 ↩

thrift://

当前 dubbo 支持[1]的 thrift 协议是对 thrift 原生协议[2]的扩展，在原生协议的基础上添加了一些额外的头信息，比如 service name，magic number 等。

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

依赖

```
<dependency>
    <groupId>org.apache.thrift</groupId>
    <artifactId>libthrift</artifactId>
    <version>0.8.0</version>
</dependency>
```

配置

所有服务共用一个端口[3]：

```
<dubbo:protocol name="thrift" port="3030" />
```

使用

可以参考 dubbo 项目中的示例代码

常见问题

- Thrift 不支持 null 值，即：不能在协议中传递 null 值

1. `2.3.0` 以上版本支持 ↩
2. Thrift 是 Facebook 捐给 Apache 的一个 RPC 框架 ↩
3. 与原生Thrift不兼容 ↩

memcached://

基于 memcached [1] 实现的 RPC 协议 [2]。

注册 memcached 服务的地址

```
RegistryFactory registryFactory = ExtensionLoader.getExtensionLoader(RegistryFactory.class).getAdaptiveExtension();
Registry registry = registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:2181"));
registry.register(URL.valueOf("memcached://10.20.153.11/com.foo.BarService?category=providers&dynamic=false&application=foo&group=member&loadbalance=consistenthash"));
```

在客户端引用

在客户端使用 [3]：

```
<dubbo:reference id="cache" interface="java.util.Map" group="member" />
```

或者，点对点直连：

```
<dubbo:reference id="cache" interface="java.util.Map" url="memcached://10.20.153.10:11211" />
```

也可以使用自定义接口：

```
<dubbo:reference id="cache" interface="com.foo.CacheService" url="memcached://10.20.153.10:11211" />
```

方法名建议和 memcached 的标准方法名相同，即：get(key), set(key, value), delete(key)。

如果方法名和 memcached 的标准方法名不相同，则需要配置映射关系 [4]：

```
<dubbo:reference id="cache" interface="com.foo.CacheService" url="memcached://10.20.153.10:11211" p:set="putFoo" p:get="getFoo" p:delete="removeFoo" />
```

1. Memcached 是一个高效的 KV 缓存服务器 ↩

2. `2.3.0` 以上版本支持 ↩

3. 不需要感知 Memcached 的地址 ↩

4. 其中 "p:xxx" 为 spring 的标准 p 标签 ↩

redis://

基于 Redis [1] 实现的 RPC 协议 [2]。

注册 redis 服务的地址

```
RegistryFactory registryFactory = ExtensionLoader.getExtensionLoader(RegistryFactory.c
lass).getAdaptiveExtension();
Registry registry = registryFactory.getRegistry(URL.valueOf("zookeeper://10.20.153.10:
2181"));
registry.register(URL.valueOf("redis://10.20.153.11/com.foo.BarService?category=provid
ers&dynamic=false&application=foo&group=member&loadbalance=consistenthash"));
```

在客户端引用

在客户端使用 [3]：

```
<dubbo:reference id="store" interface="java.util.Map" group="member" />
```

或者，点对点直连：

```
<dubbo:reference id="store" interface="java.util.Map" url="redis://10.20.153.10:6379"
/>
```

也可以使用自定义接口：

```
<dubbo:reference id="store" interface="com.foo.StoreService" url="redis://10.20.153.10
:6379" />
```

方法名建议和 redis 的标准方法名相同，即：get(key), set(key, value), delet(key)。

如果方法名和 redis 的标准方法名不相同，则需要配置映射关系 [4]：

```
<dubbo:reference id="cache" interface="com.foo.CacheService" url="memcached://10.20.15
3.10:11211" p:set="putFoo" p:get="getFoo" p:delete="removeFoo" />
```

> 1. Redis 是一个高效的 KV 存储服务器 ↩

2. `2.3.0` 以上版本支持 ↩

3. 不需要感知 Redis 的地址 ↩

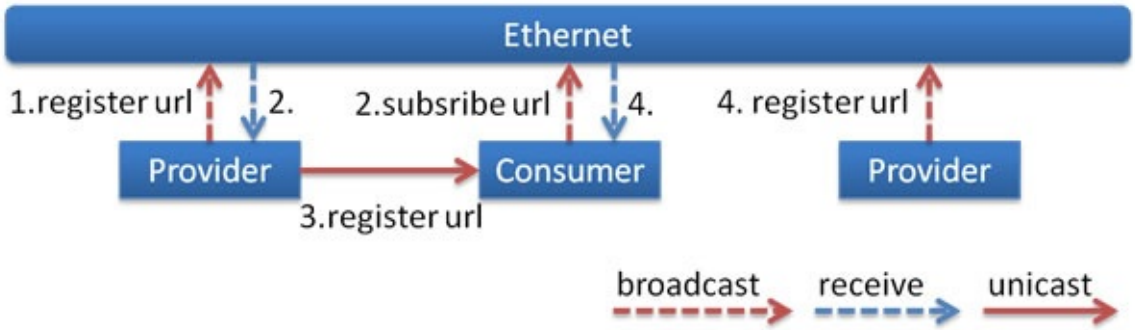
4. 其中 "p:xxx" 为 spring 的标准 p 标签 ↩

注册中心参考手册

推荐使用 Zookeeper 注册中心

Multicast 注册中心

Multicast 注册中心不需要启动任何中心节点，只要广播地址一样，就可以互相发现。



1. 提供方启动时广播自己的地址
2. 消费方启动时广播订阅请求
3. 提供方收到订阅请求时，单播自己的地址给订阅者，如果设置了 `unicast=false` ，则广播给订阅者
4. 消费方收到提供方地址时，连接该地址进行 RPC 调用。

组播受网络结构限制，只适合小规模应用或开发阶段使用。组播地址段: 224.0.0.0 - 239.255.255.255

配置

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

或

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

为了减少广播量，Dubbo 缺省使用单播发送提供者地址信息给消费者，如果一个机器上同时启了多个消费者进程，消费者需声明 `unicast=false` ，否则只会有一个消费者能收到消息：

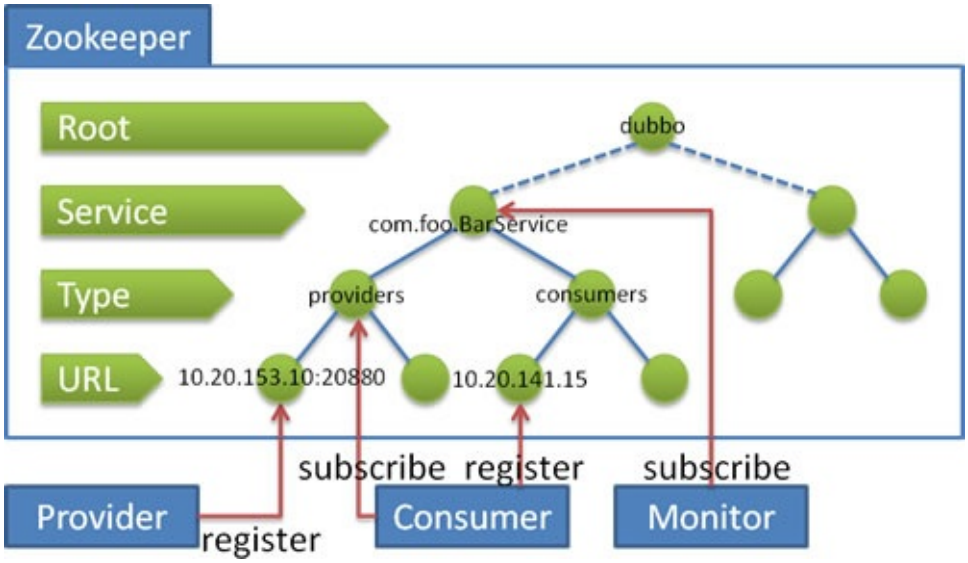
```
<dubbo:registry address="multicast://224.5.6.7:1234?unicast=false" />
```

或

```
<dubbo:registry protocol="multicast" address="224.5.6.7:1234">
    <dubbo:parameter key="unicast" value="false" />
</dubbo:registry>
```

zookeeper 注册中心

Zookeeper 是 Apacahe Hadoop 的子项目，是一个树型的目录服务，支持变更推送，适合作为 Dubbo 服务的注册中心，工业强度较高，可用于生产环境，并推荐使用 [1]。



流程说明：

- 服务提供者启动时: 向 `/dubbo/com.foo.BarService/providers` 目录下写入自己的 URL 地址
- 服务消费者启动时: 订阅 `/dubbo/com.foo.BarService/providers` 目录下的提供者 URL 地址。并向 `/dubbo/com.foo.BarService/consumers` 目录下写入自己的 URL 地址
- 监控中心启动时: 订阅 `/dubbo/com.foo.BarService` 目录下的所有提供者和消费者 URL 地址。

支持以下功能：

- 当提供者出现断电等异常停机时，注册中心能自动删除提供者信息
- 当注册中心重启时，能自动恢复注册数据，以及订阅请求
- 当会话过期时，能自动恢复注册数据，以及订阅请求
- 当设置 `<dubbo:registry check="false" />` 时，记录失败注册和订阅请求，后台定时重试
- 可通过 `<dubbo:registry username="admin" password="1234" />` 设置 zookeeper 登录信息
- 可通过 `<dubbo:registry group="dubbo" />` 设置 zookeeper 的根节点，不设置将使用无根树
- 支持 `*` 号通配符 `<dubbo:reference group="*" version="*" />`，可订阅服务的所有分组和所有版本的提供者

使用

在 provider 和 consumer 中增加 zookeeper 客户端 jar 包依赖：

```
<dependency>
    <groupId>org.apache.zookeeper</groupId>
    <artifactId>zookeeper</artifactId>
    <version>3.3.3</version>
</dependency>
```

或直接下载。

Dubbo 支持 zkclient 和 curator 两种 Zookeeper 客户端实现：

使用 zkclient 客户端

从 `2.2.0` 版本开始缺省为 zkclient 实现，以提升 zookeeper 客户端的健状性。zkclient 是 Datameer 开源的一个 Zookeeper 客户端实现。

缺省配置：

```
<dubbo:registry ... client="zkclient" />
```

或：

```
dubbo.registry.client=zkclient
```

或：

```
zookeeper://10.20.153.10:2181?client=zkclient
```

需依赖或直接下载：

```
<dependency>
    <groupId>com.github.sgroschupf</groupId>
    <artifactId>zkclient</artifactId>
    <version>0.1</version>
</dependency>
```

使用 curator 客户端

从 `2.3.0` 版本开始支持可选 curator 实现。Curator 是 Netflix 开源的一个 Zookeeper 客户端实现。

如果需要改为 curator 实现，请配置：

```
<dubbo:registry ... client="curator" />
```

或：

```
dubbo.registry.client=curator
```

或：

```
zookeeper://10.20.153.10:2181?client=curator
```

需依赖或直接下载：

```
<dependency>
    <groupId>com.netflix.curator</groupId>
    <artifactId>curator-framework</artifactId>
    <version>1.1.10</version>
</dependency>
```

Zookeeper 单机配置:

```
<dubbo:registry address="zookeeper://10.20.153.10:2181" />
```

或：

```
<dubbo:registry protocol="zookeeper" address="10.20.153.10:2181" />
```

Zookeeper 集群配置：

```
<dubbo:registry address="zookeeper://10.20.153.10:2181?backup=10.20.153.11:2181,10.20.
153.12:2181" />
```

或：

```
<dubbo:registry protocol="zookeeper" address="10.20.153.10:2181,10.20.153.11:2181,10.2
0.153.12:2181" />
```

同一 Zookeeper，分成多组注册中心:

```
<dubbo:registry id="chinaRegistry" protocol="zookeeper" address="10.20.153.10:2181" gr
oup="china" />
<dubbo:registry id="intlRegistry" protocol="zookeeper" address="10.20.153.10:2181" gro
up="intl" />
```

zookeeper 安装

安装方式参见: Zookeeper安装手册，只需搭一个原生的 Zookeeper 服务器，并将 Quick Start 中 Provider 和 Consumer 里的 `conf/dubbo.properties` 中的 `dubbo.registry.addrss` 的值改为 `zookeeper://127.0.0.1:2181` 即可使用。

可靠性声明

阿里内部并没有采用 Zookeeper 做为注册中心，而是使用自己实现的基于数据库的注册中心，即：Zookeeper 注册中心并没有在阿里内部长时间运行的可靠性保障，此 Zookeeper 桥接实现只为开源版本提供，其可靠性依赖于 Zookeeper 本身的可靠性。

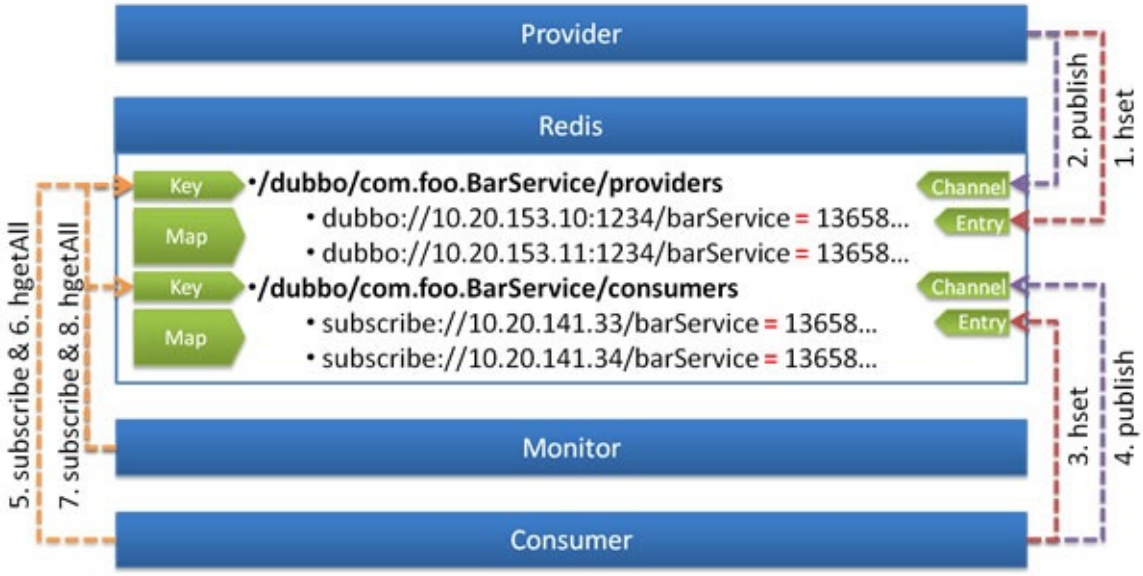
兼容性声明

因 `2.0.8` 最初设计的 zookeeper 存储结构不能扩充不同类型的数据，`2.0.9` 版本做了调整，所以不兼容，需全部改用 `2.0.9` 版本才行，以后的版本会保持兼容 `2.0.9`。`2.2.0` 版本改为基于 zkclient 实现，需增加 zkclient 的依赖包，`2.3.0` 版本增加了基于 curator 的实现，作为可选实现策略。

> [1]. 建议使用 `2.3.3` 以上版本的 zookeeper 注册中心客户端 ↩

Redis 注册中心

基于 Redis [1] 实现的注册中心 [2]。



使用 Redis 的 Key/Map 结构存储数据结构：

- 主 Key 为服务名和类型
- Map 中的 Key 为 URL 地址
- Map 中的 Value 为过期时间，用于判断脏数据，脏数据由监控中心删除 [3]

使用 Redis 的 Publish/Subscribe 事件通知数据变更：

- 通过事件的值区分事件类型：`register`，`unregister`，`subscribe`，`unsubscribe`
- 普通消费者直接订阅指定服务提供者的 Key，只会收到指定服务的 `register`，`unregister` 事件
- 监控中心通过 `psubscribe` 功能订阅 `/dubbo/*`，会收到所有服务的所有变更事件

调用过程：

1. 服务提供方启动时，向 `Key:/dubbo/com.foo.BarService/providers` 下，添加当前提供者的地址
2. 并向 `Channel:/dubbo/com.foo.BarService/providers` 发送 `register` 事件
3. 服务消费方启动时，从 `Channel:/dubbo/com.foo.BarService/providers` 订阅 `register` 和 `unregister` 事件
4. 并向 `Key:/dubbo/com.foo.BarService/providers` 下，添加当前消费者的地址
5. 服务消费方收到 `register` 和 `unregister` 事件后，从 `Key:/dubbo/com.foo.BarService/providers` 下获取提供者地址列表
6. 服务监控中心启动时，从 `Channel:/dubbo/*` 订阅 `register` 和 `unregister`，以及

`subscribe` 和 `unsubsribe` 事件
7. 服务监控中心收到 `register` 和 `unregister` 事件后，从 `Key:/dubbo/com.foo.BarService/providers` 下获取提供者地址列表
8. 服务监控中心收到 `subscribe` 和 `unsubsribe` 事件后，从 `Key:/dubbo/com.foo.BarService/consumers` 下获取消费者地址列表

配置

```
<dubbo:registry address="redis://10.20.153.10:6379" />
```

或

```
<dubbo:registry address="redis://10.20.153.10:6379?backup=10.20.153.11:6379,10.20.153.12:6379" />
```

或

```
<dubbo:registry protocol="redis" address="10.20.153.10:6379" />
```

或

```
<dubbo:registry protocol="redis" address="10.20.153.10:6379,10.20.153.11:6379,10.20.153.12:6379" />
```

选项

- 可通过 `<dubbo:registry group="dubbo" />` 设置 redis 中 key 的前缀，缺省为 `dubbo`。
- 可通过 `<dubbo:registry cluster="replicate" />` 设置 redis 集群策略，缺省为 `failover`：
  - `failover`：只写入和读取任意一台，失败时重试另一台，需要服务器端自行配置数据同步
  - `replicate`：在客户端同时写入所有服务器，只读取单台，服务器端不需要同步，注册中心集群增大，性能压力也会更大

可靠性声明

阿里内部并没有采用 Redis 做为注册中心，而是使用自己实现的基于数据库的注册中心，即：Redis 注册中心并没有在阿里内部长时间运行的可靠性保障，此 Redis 桥接实现只为开源版本提供，其可靠性依赖于 Redis 本身的可靠性。

安装

安装方式参见: Redis安装手册，只需搭一个原生的 Redis 服务器，并将 Quick Start 中 Provider 和 Consumer 里的 `conf/dubbo.properties` 中的 `dubbo.registry.addrss` 的值改为 `redis://127.0.0.1:6379` 即可使用。

> 1. Redis 是一个高效的 KV 存储服务器 ↩
>
> 2. 从 `2.1.0` 版本开始支持 ↩
>
> 3. Redis 过期数据通过心跳的方式检测脏数据，服务器时间必须同步，并且对服务器有一定压力，否则过期检测会不准确 ↩

Simple 注册中心

Simple 注册中心本身就是一个普通的 Dubbo 服务，可以减少第三方依赖，使整体通讯方式一致。

配置

将 Simple 注册中心暴露成 Dubbo 服务：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springf
ramework.org/schema/beans/spring-beans-2.5.xsdhttp://code.alibabatech.com/schema/dubbo
 http://code.alibabatech.com/schema/dubbo/dubbo.xsd">
    <!-- 当前应用信息配置 -->
    <dubbo:application name="simple-registry" />
    <!-- 暴露服务协议配置 -->
    <dubbo:protocol port="9090" />
    <!-- 暴露服务配置 -->
    <dubbo:service interface="com.alibaba.dubbo.registry.RegistryService" ref="registr
yService" registry="N/A" ondisconnect="disconnect" callbacks="1000">
        <dubbo:method name="subscribe"><dubbo:argument index="1" callback="true" /></d
ubbo:method>
        <dubbo:method name="unsubscribe"><dubbo:argument index="1" callback="false" />
</dubbo:method>
    </dubbo:service>
    <!-- 简单注册中心实现，可自行扩展实现集群和状态同步 -->
    <bean id="registryService" class="com.alibaba.dubbo.registry.simple.SimpleRegistry
Service" />
</beans>
```

引用 Simple Registry 服务：

```
<dubbo:registry address="127.0.0.1:9090" />
```

或者：

```
<dubbo:service interface="com.alibaba.dubbo.registry.RegistryService" group="simple" v
ersion="1.0.0" ... >
```

或者：

```
<dubbo:registry address="127.0.0.1:9090" group="simple" version="1.0.0" />
```

适用性说明

此 `SimpleRegistryService` 只是简单实现，不支持集群，可作为自定义注册中心的参考，但不适合直接用于生产环境。

Telnet 命令参考手册

从 `2.0.5` 版本开始，dubbo 开始支持通过 telnet 命令来镜像服务治理。

使用

```
telnet localhost 20880
```

或者：

```
echo status | nc -i 1 localhost 20880
```

status命令所检查的资源也可以扩展，参见：扩展参考手册。

命令

以下展示了 dubbo 内建的 telnet 命令的说明和用法，此外，telnet 命令还支持用户自行扩展，参见：Telnet 命令扩展。

ls

1. `ls`：显示服务列表
2. `ls -l`：显示服务详细信息列表
3. `ls XxxService`：显示服务的方法列表
4. `ls -l XxxService`：显示服务的方法详细信息列表

ps

1. `ps`：显示服务端口列表
2. `ps -l`：显示服务地址列表
3. `ps 20880`：显示端口上的连接信息
4. `ps -l 20880`：显示端口上的连接详细信息

cd

1. `cd XxxService`：改变缺省服务，当设置了缺省服务，凡是需要输入服务名作为参数的命

令，都可以省略服务参数
2. `cd /`：取消缺省服务

pwd

`pwd`：显示当前缺省服务

trace

1. `trace XxxService`：跟踪 1 次服务任意方法的调用情况
2. `trace XxxService 10`：跟踪 10 次服务任意方法的调用情况
3. `trace XxxService xxxMethod`：跟踪 1 次服务方法的调用情况
4. `trace XxxService xxxMethod 10`：跟踪 10 次服务方法的调用情况

count

1. `count XxxService`：统计 1 次服务任意方法的调用情况
2. `count XxxService 10`：统计 10 次服务任意方法的调用情况
3. `count XxxService xxxMethod`：统计 1 次服务方法的调用情况
4. `count XxxService xxxMethod 10`：统计 10 次服务方法的调用情况

invoke

1. `invoke XxxService.xxxMethod({"prop": "value"})`：调用服务的方法
2. `invoke xxxMethod({"prop": "value"})`：调用服务的方法(自动查找包含此方法的服务)

status

1. `status`：显示汇总状态，该状态将汇总所有资源的状态，当全部 OK 时则显示 OK，只要有一个 ERROR 则显示 ERROR，只要有一个 WARN 则显示 WARN
2. `status -l`：显示状态列表

log [1]

1. `log debug`：修改 dubbo logger 的日志级别
2. `log 100`：查看 file logger 的最后 100 字符的日志

help

1. `help`：显示 telnet 命帮助信息
2. `help xxx`：显示xxx命令的详细帮助信息

clear

1. `clear`：清除屏幕上的内容
2. `clear 100`：清除屏幕上的指定行数的内容

exit

`exit`：退出当前 telnet 命令行

> 1. `2.0.6` 以上版本支持 ↩

Maven 插件参考手册

启动一个简易注册中心

以指定的9099端口启动一个简易注册中心[1]：

```
mvn dubbo:registry -Dport=9099
```

生成demo服务提供者应用

生成指定接口和版本的服务提供者应用：

```
mvn dubbo:create -Dapplication=xxx -Dpackage=com.alibaba.xxx -Dservice=XxxService,YyyS
ervice -Dversion=1.0.0
```

> [1]. 如果端口不指定，默认端口为 9090 ↩

服务化最佳实践

分包

建议将服务接口，服务模型，服务异常等均放在 API 包中，因为服务模型及异常也是 API 的一部分，同时，这样做也符合分包原则：重用发布等价原则(REP)，共同重用原则(CRP)。

如果需要，也可以考虑在 API 包中放置一份 spring 的引用配置，这样使用方，只需在 spring 加载过程中引用此配置即可，配置建议放在模块的包目录下，以免冲突，
如：`com/alibaba/china/xxx/dubbo-reference.xml`。

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。

服务接口建议以业务场景为单位划分，并对相近业务做抽象，防止接口数量爆炸。

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。

兼容性

服务接口增加方法，或服务模型增加字段，可向后兼容，删除方法或删除字段，将不兼容，枚举类型新增字段也不兼容，需通过变更版本号升级。

各协议的兼容性不同，参见：服务协议

枚举值

如果是完备集，可以用 `Enum`，比如：`ENABLE`, `DISABLE`。

如果是业务种类，以后明显会有类型增加，不建议用 `Enum`，可以用 `String` 代替。

如果是在返回值中用了 `Enum`，并新增了 `Enum` 值，建议先升级服务消费方，这样服务提供方不会返回新值。

如果是在传入参数中用了 `Enum`，并新增了 `Enum` 值，建议先升级服务提供方，这样服务消费方不会传入新值。

序列化

服务参数及返回值建议使用 POJO 对象，即通过 `setter`, `getter` 方法表示属性的对象。

服务参数及返回值不建议使用接口，因为数据模型抽象的意义不大，并且序列化需要接口实现类的元信息，并不能起到隐藏实现的意图。

服务参数及返回值都必需是 byValue 的，而不能是 byReference 的，消费方和提供方的参数或返回值引用并不是同一个，只是值相同，Dubbo 不支持引用远程对象。

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，以及语义更友好。

如果担心性能问题，在必要时，可以通过 override 掉异常类的 `fillInStackTrace()` 方法为空方法，使其不拷贝栈信息。

查询方法不建议抛出 checked 异常，否则调用方在查询时将过多的 `try...catch`，并且不能进行有效处理。

服务提供方不应将 DAO 或 SQL 等异常抛给消费方，应在服务实现中对消费方不关心的异常进行包装，否则可能出现消费方无法反序列化相应异常。

调用

不要只是因为是 Dubbo 调用，而把调用 `try...catch` 起来。`try...catch` 应该加上合适的回滚边界上。

对于输入参数的校验逻辑在 Provider 端要有。如有性能上的考虑，服务实现者可以考虑在 API 包上加上服务 Stub 类来完成检验。

推荐用法

在 Provider 上尽量多配置 Consumer 端属性

原因如下：

- 作服务的提供者，比服务使用方更清楚服务性能参数，如调用的超时时间，合理的重试次数，等等
- 在 Provider 配置后，Consumer 不配置则会使用 Provider 的配置值，即 Provider 配置可以作为 Consumer 的缺省值[1]。否则，Consumer 会使用 Consumer 端的全局设置，这对于 Provider 不可控的，并且往往是不合理的

Provider 上尽量多配置 Consumer 端的属性，让 Provider 实现者一开始就思考 Provider 服务特点、服务质量的问题。

示例：

```
<dubbo:service interface="com.alibaba.hello.api.HelloService" version="1.0.0" ref="hel
loService"
    timeout="300" retry="2" loadbalance="random" actives="0"
/>

<dubbo:service interface="com.alibaba.hello.api.WorldService" version="1.0.0" ref="hel
loService"
    timeout="300" retry="2" loadbalance="random" actives="0" >
    <dubbo:method name="findAllPerson" timeout="10000" retries="9" loadbalance="leasta
ctive" actives="5" />
<dubbo:service/>
```

在 Provider 上可以配置的 Consumer 端属性有：

1. `timeout` 方法调用超时
2. `retries` 失败重试次数，缺省是 2[2]
3. `loadbalance` 负载均衡算法[3]，缺省是随机 `random`。还可以有轮询 `roundrobin`、最不活跃优先[4] `leastactive`
4. `actives` 消费者端，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 Wait 直到超时 在方法上配置 `dubbo:method` 则并发限制针对方法，在接口上配置 `dubbo:service`，则并发限制针对服务

详细配置说明参见：Dubbo配置参考手册

Provider 上配置合理的 Provider 端属性

```
<dubbo:protocol threads="200" />
<dubbo:service interface="com.alibaba.hello.api.HelloService" version="1.0.0" ref="hel
loService"
    executes="200" >
    <dubbo:method name="findAllPerson" executes="50" />
</dubbo:service>
```

Provider 上可以配置的 Provider 端属性有：

1. `threads` 服务线程池大小
2. `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到上限后，新调用会 Wait，这个时候 Consumer可能会超时。在方法上配置 `dubbo:method` 则并发限制针对方法，在接口上配置 `dubbo:service`，则并发限制针对服务

配置管理信息

目前有负责人信息和组织信息用于区分站点。有问题时便于的找到服务的负责人，至少写两个人以便备份。负责人和组织的信息可以在注册中心的上看到。

应用配置负责人、组织：

```
<dubbo:application owner="ding.lid,william.liangf" organization="intl" />
```

service 配置负责人：

```
<dubbo:service owner="ding.lid,william.liangf" />
```

reference 配置负责人：

```
<dubbo:reference owner="ding.lid,william.liangf" />
```

`dubbo:service`、`dubbo:reference` 没有配置负责人，则使用 `dubbo:application` 设置的负责人。

配置 Dubbo 缓存文件

提供者列表缓存文件：

```
<dubbo:registry file="${user.home}/output/dubbo.cache" />
```

注意：

1. 文件的路径，应用可以根据需要调整，保证这个文件不会在发布过程中被清除。
2. 如果有多个应用进程注意不要使用同一个文件，避免内容被覆盖。

这个文件会缓存注册中心的列表和服务提供者列表。有了这项配置后，当应用重启过程中，Dubbo 注册中心不可用时则应用会从这个缓存文件读取服务提供者列表的信息，进一步保证应用可靠性。

监控配置

1. 使用固定端口暴露服务，而不要使用随机端口

   这样在注册中心推送有延迟的情况下，消费者通过缓存列表也能调用到原地址，保证调用成功。

2. 使用 Dragoon 的 http 监控项监控注册中心上服务提供方

   Dragoon 监控服务在注册中心上的状态：http://dubbo-reg1.hst.xyi.cn.alidc.net:8080/status/com.alibaba.morgan.member.MemberService:1.0.5 确保注册中心上有该服务的存在。

3. 服务提供方，使用 Dragoon 的 telnet 或 shell 监控项

   监控服务提供者端口状态：`echo status | nc -i 1 20880 | grep OK | wc -l`，其中的 20880 为服务端口

4. 服务消费方，通过将服务强制转型为 EchoService，并调用 `$echo()` 测试该服务的提供者是可用

   如 `assertEqauls("OK", ((EchoService)memberService).$echo("OK"));`

不要使用 dubbo.properties 文件配置，推荐使用对应 XML 配置

Dubbo 中所有的配置项都可以配置在 Spring 配置文件中，并且可以针对单个服务配置。

如完全不配置则使用 Dubbo 缺省值，参见 Dubbo配置参考手册 中的说明。

dubbo.properties 中属性名与 XML 的对应关系

1. 应用名 `dubbo.application.name`

```
<dubbo:application name="myalibaba" >
```

2. 注册中心地址 `dubbo.registry.address`

```
<dubbo:registry address="11.22.33.44:9090" >
```

3. 调用超时 `dubbo.service.*.timeout`

   可以在多个配置项设置超时 `timeout` ，由上至下覆盖（即上面的优先）[5]，其它的参数（ `retries` 、 `loadbalance` 、 `actives` 等）的覆盖策略也一样示例如下：

   提供者端特定方法的配置

```
<dubbo:service interface="com.alibaba.xxx.XxxService" >
    <dubbo:method name="findPerson" timeout="1000" />
</dubbo:service>
```

   提供者端特定接口的配置

```
<dubbo:service interface="com.alibaba.xxx.XxxService" timeout="200" />
```

4. 服务提供者协议 `dubbo.service.protocol` 、服务的监听端口 `dubbo.service.server.port`

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

5. 服务线程池大小 `dubbo.service.max.thread.threads.size`

```
<dubbo:protocol threads="100" />
```

6. 消费者启动时，没有提供者是否抛异常 Fast-Fail
   `alibaba.intl.commons.dubbo.service.allow.no.provider`

```
<dubbo:reference interface="com.alibaba.xxx.XxxService" check="false" />
```

[1]. 配置的覆盖规则：1) 方法级配置别优于接口级别，即小 Scope 优先 2) Consumer 端配置优于 Provider 配置，优于全局配置，最后是Dubbo 硬编码的配置值（Dubbo 配置参考手册） ↩

[2]. 表示加上第一次调用，会调用 3 次 ↩

3

> 3. 有多个 Provider 时，如何挑选 Provider 调用 ↩
>
> 4. 指从 Consume r端并发调用最好的 Provider，可以减少的反应慢的 Provider 的调用，因为反应更容易累积并发的调用 ↩
>
> 5. `timeout` 可以在多处设置，配置项及覆盖规则详见： Dubbo 配置参考手册 ↩

容量规划

以下数据供参考：

使用 **Dubbo** 的会员服务项目

- 每天接收 4 亿次远程调用
- 使用 12 台网站标配机器提供服务(8 核 CPU, 8G 内存)
- 平均负载在 1 以下(对于 8 核 CPU 负载很低)
- 平均响应时间 2.3 到 2.5 毫秒，网络开销约占 1.5 到 1.6 毫秒(和数据包大小有关)

使用 **Dubbo** 的产品授权服务项目

- 每天接收 3 亿次远程调用
- 使用 8 台网站标配机器提供服务(8 核CPU,8G 内存)
- 平均负载在 1 以下(对于 8 核 CPU 负载很低)
- 平均响应时间 1.4 到 2.8 毫秒，网络开销约占 1.0 到 1.1 毫秒(和数据包大小有关)

性能测试报告

测试说明

1. 本次性能测试，测试了 dubbo 2.0 所有支持的协议在不同大小和数据类型下的表现，并与 dubbo 1.0 进行了对比。
2. 整体性能相比 1.0 有了提升，平均提升 10%，使用 dubbo 2.0 新增的 dubbo 序列化还能获得 10%~50% 的性能提升，详见下面的性能数据。
3. 稳定性测试中由于将底层通信框架从 mina 换成 netty，old 区对象的增长大大减少，50 小时运行，增长不到 200m，无 fullgc。
4. 存在的问题：在 50k 数据的时候 2.0 性能不如 1.0，怀疑可能是缓冲区设置的问题，下版本会进一步确认。

测试环境

硬件部署与参数调整

机型	**CPU**	内存	网络	磁盘	内核
Tecal BH620	model name : Intel(R) Xeon(R) CPU E5520 @ 2.27GHz cache size : 8192 KB processor_count : 16	Total System Memory: 6G Hardware Memory Info: Size: 4096MB	eth0: Link is up at 1000 Mbps, full duplex. peth0: Link is up at 1000 Mbps, full duplex.	/dev/sda: 597.9 GB	2.6.18-128.el5xen x86_64

软件架构

软件名称及版本	关键参数
java version "1.6.0_18" Java(TM) SE Runtime Environment (build 1.6.0_18-b07) Java HotSpot(TM) 64-Bit Server VM (build 16.0-b13, mixed mode)	-server -Xmx2g -Xms2g -Xmn256m -XX:PermSize=128m -Xss256k -XX:+DisableExplicitGC -XX:+UseConcMarkSweepGC -XX:+CMSParallelRemarkEnabled -XX:+UseCMSCompactAtFullCollection -XX:LargePageSizeInBytes=128m -XX:+UseFastAccessorMethods -XX:+UseCMSInitiatingOccupancyOnly -XX:CMSInitiatingOccupancyFraction=70
jboss-4.0.5.GA	
httpd-2.0.61	KeepAlive On MaxKeepAliveRequests 100000 KeepAliveTimeout 180 MaxRequestsPerChild 1000000 StartServers 5 MaxClients 1024 MinSpareThreads 25 MaxSpareThreads 75 ThreadsPerChild 64 ThreadLimit 128 ServerLimit 16

测试目的

期望性能指标(量化)

场景名称	对应指标名称	期望值范围	实际值	是否满足期望(是/否)
1k数据	响应时间	0.9ms	0.79ms	是
1k数据	TPS	10000	11994	是

期望运行状况(非量化，可选)

- 2.0 性能不低于 1.0, 2.0 和 1.0 互调用的性能无明显下降。 除了 50k string 其余皆通过
- JVM 内存运行稳定，无 OOM，堆内存中无不合理的大对象的占用。通过
- CPU、内存、网络、磁盘、文件句柄占用平稳。通过
- 无频繁线程锁，线程数平稳。通过
- 业务线程负载均衡。通过

测试脚本

1. 性能测试场景（10 并发）
    - 传入 1k String，不做任何处理，原样返回
    - 传入 50k String，不做任何处理，原样返回
    - 传入 200k String，不做任何处理，原样返回

- 传入 1k POJO（嵌套的复杂 person 对象），不做任何处理，原样返回

  上述场景在 dubbo 1.0, dubbo 2.0(hessian2序列化), dubbo 2.0(dubbo序列化), rmi, hessian 3.2.0, http(json序列化) 进行 10 分钟的性能测试。主要考察序列化和网络 IO 的性能，因此服务端无任何业务逻辑。取 10 并发是考虑到 http 协议在高并发下对 CPU 的使用率较高可能会先打到瓶颈。

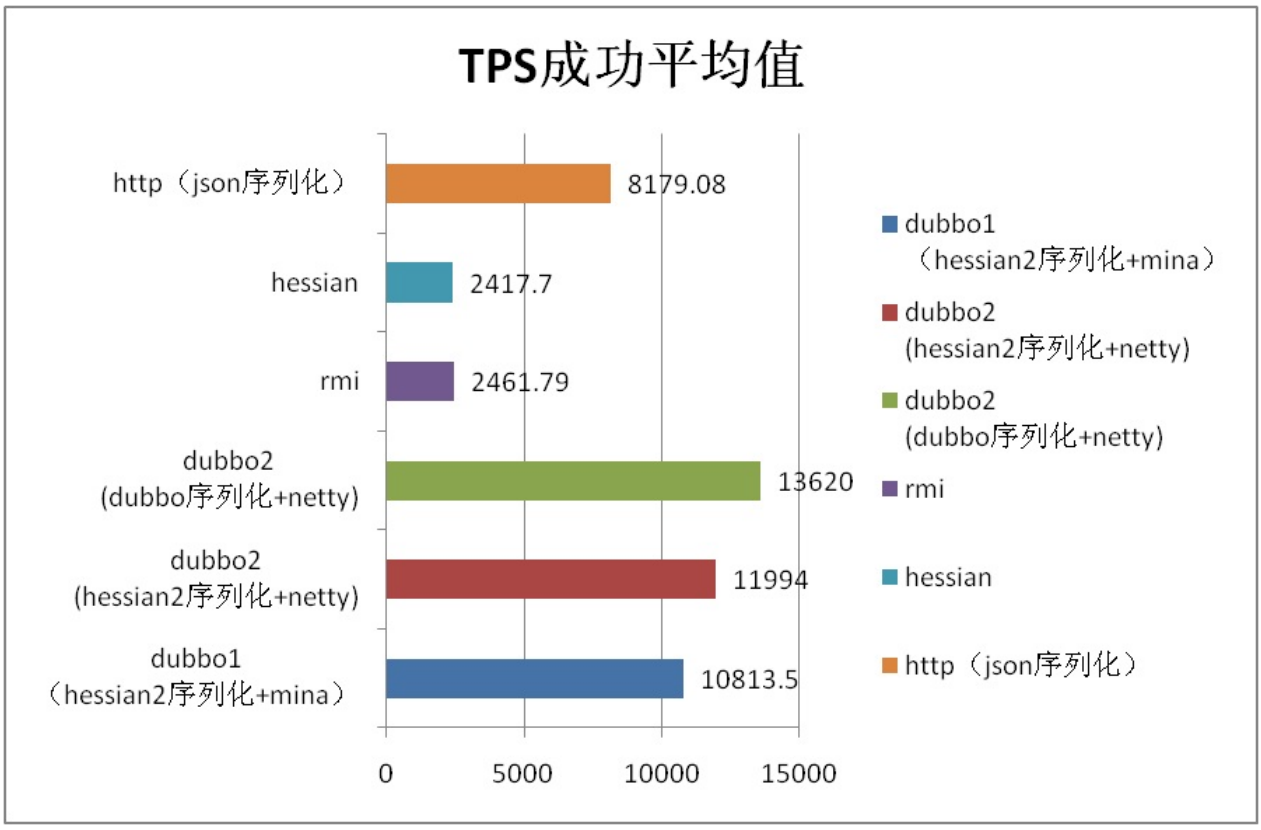
2. 并发场景（20 并发） 传入 1k String，在服务器段循环 1w 次，每次重新生成一个随机数然后进行拼装。考察业务线程是否能够分配到每个 CPU 上。
3. 稳定性场景（20 并发） 同时调用 1 个参数为 String（5k）方法，1 个参数为 person 对象的方法，1 个参数为 map（值为 3 个 person）的方法，持续运行 50 小时。
4. 高压力场景（20 并发） 在稳定性场景的基础上，将提供者和消费者布置成均为 2 台（一台机器 2 个实例），且 String 的参数从 20byte 到 200k，每隔 10 分钟随机变换。

测试结果

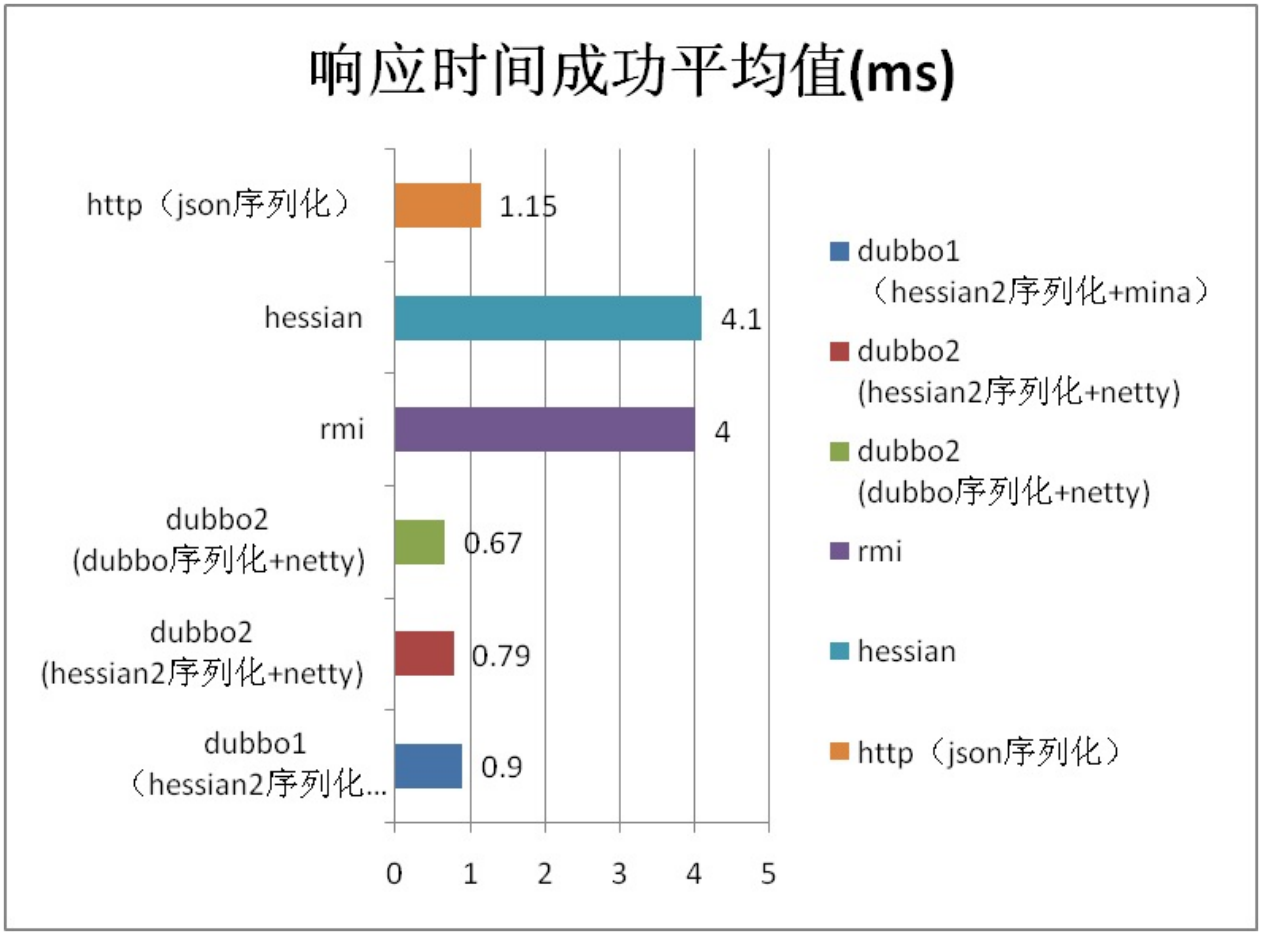
场景名称：**POJO** 场景

	TPS成功平均值	响应时间成功平均值**(ms)**
dubbo1 (hessian2序列化+mina)	10813.5	0.9
dubbo2 (hessian2序列化+netty)	11994	0.79
dubbo2 (dubbo序列化+netty)	13620	0.67
rmi	2461.79	4
hessian	2417.7	4.1
http（json序列化）	8179.08	1.15
2.0和1.0默认对比百分比	10.92	-12.22
dubbo序列化相比hessian2序列化百分比	13.56	-15.19

POJO TPS



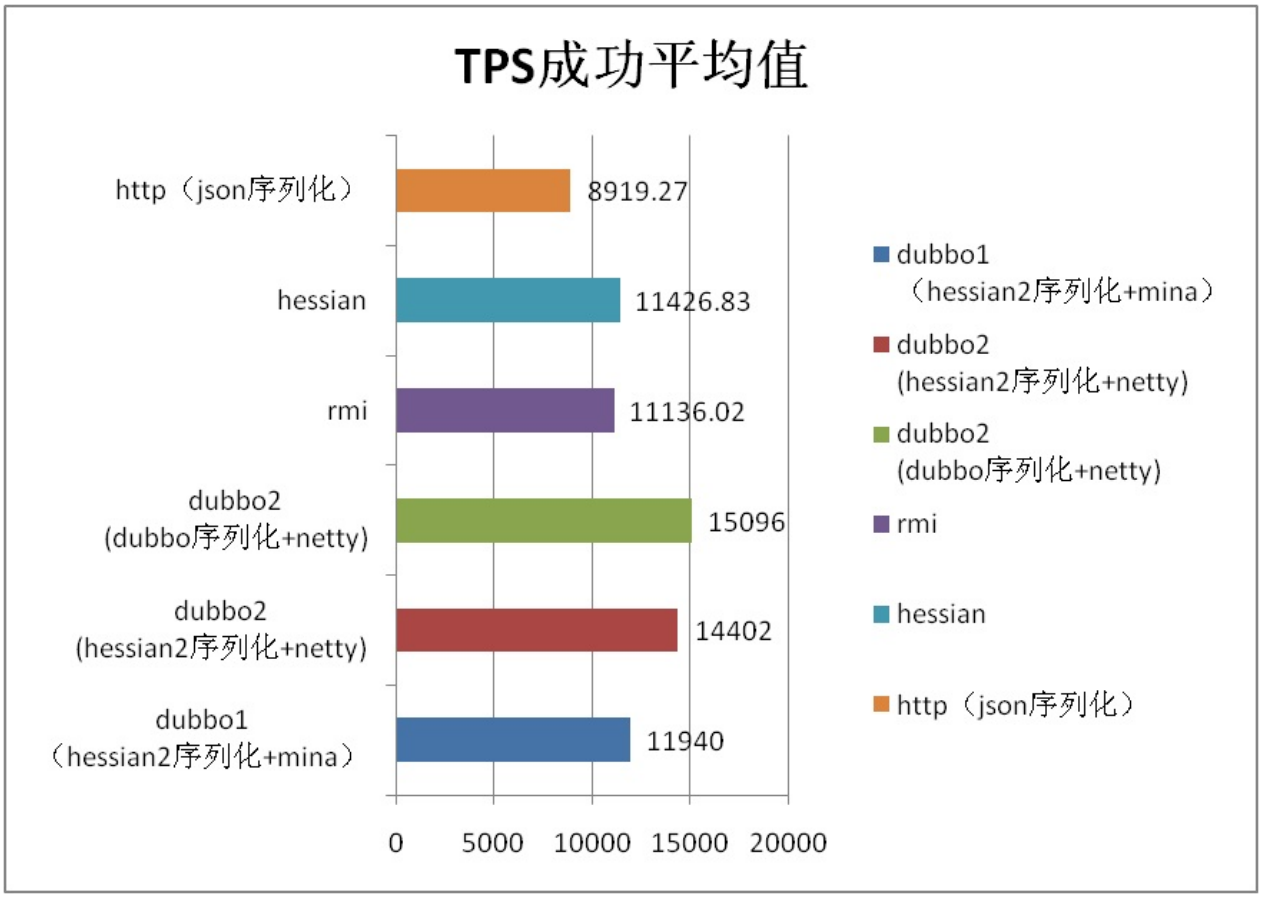
POJO Response



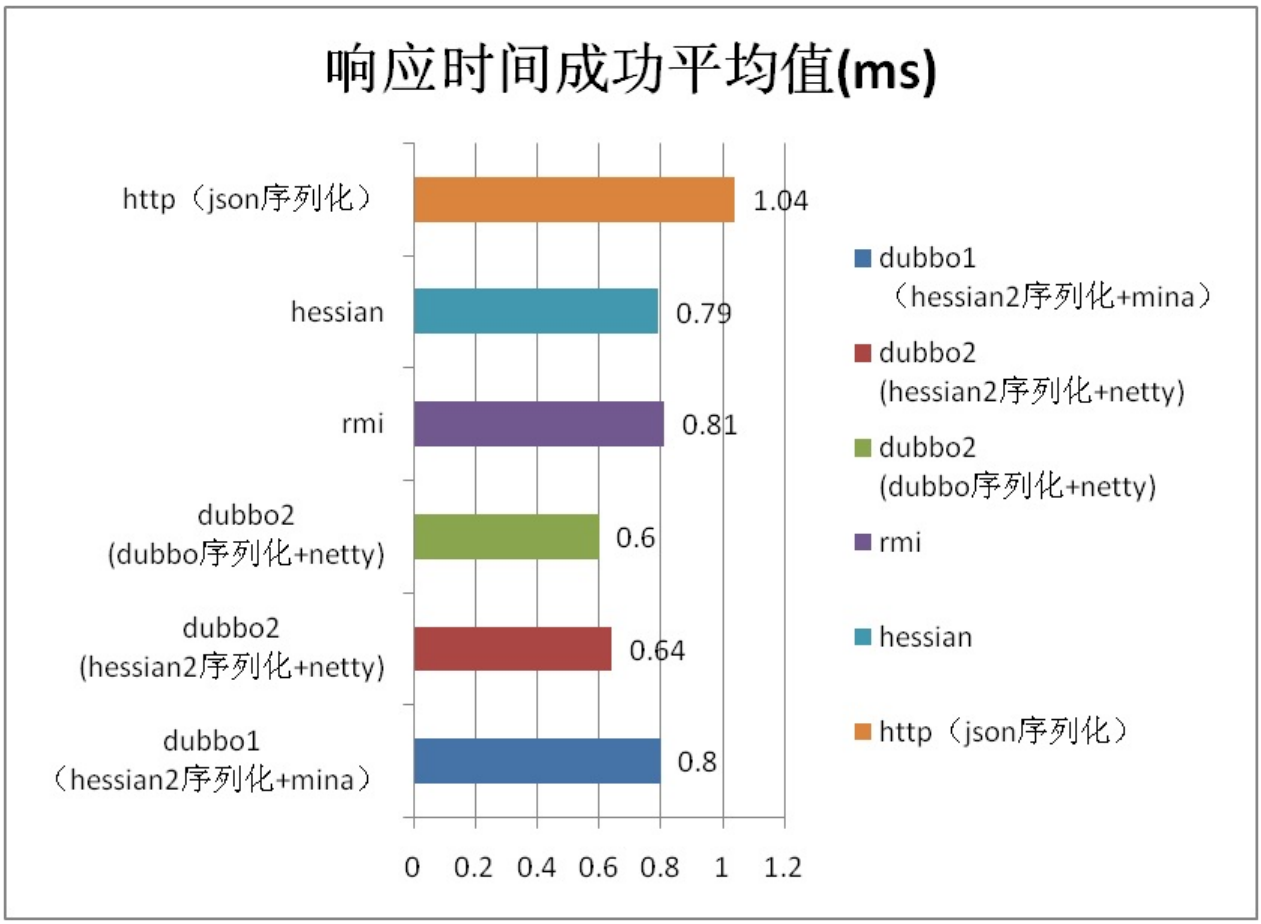
场景名称：**1k string** 场景

	TPS成功平均值	响应时间成功平均值(ms)
dubbo1（hessian2序列化+mina）	11940	0.8
dubbo2 (hessian2序列化+netty)	14402	0.64
dubbo2 (dubbo序列化+netty)	15096	0.6
rmi	11136.02	0.81
hessian	11426.83	0.79
http（json序列化）	8919.27	1.04
2.0和1.0默认对比百分比	20.62	-20.00
dubbo序列化相比hessian2序列化百分比	4.82	-6.25

1k TPS



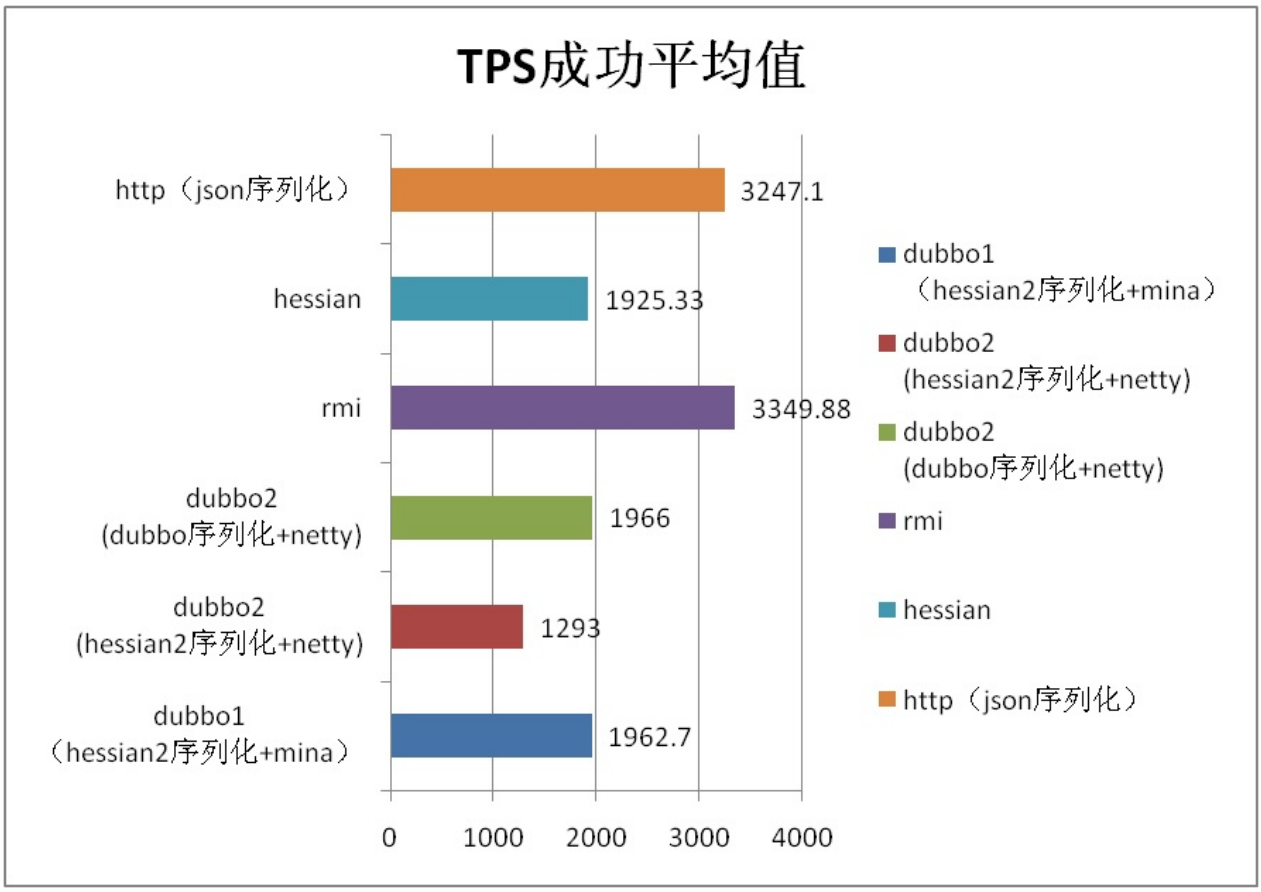
1k Response



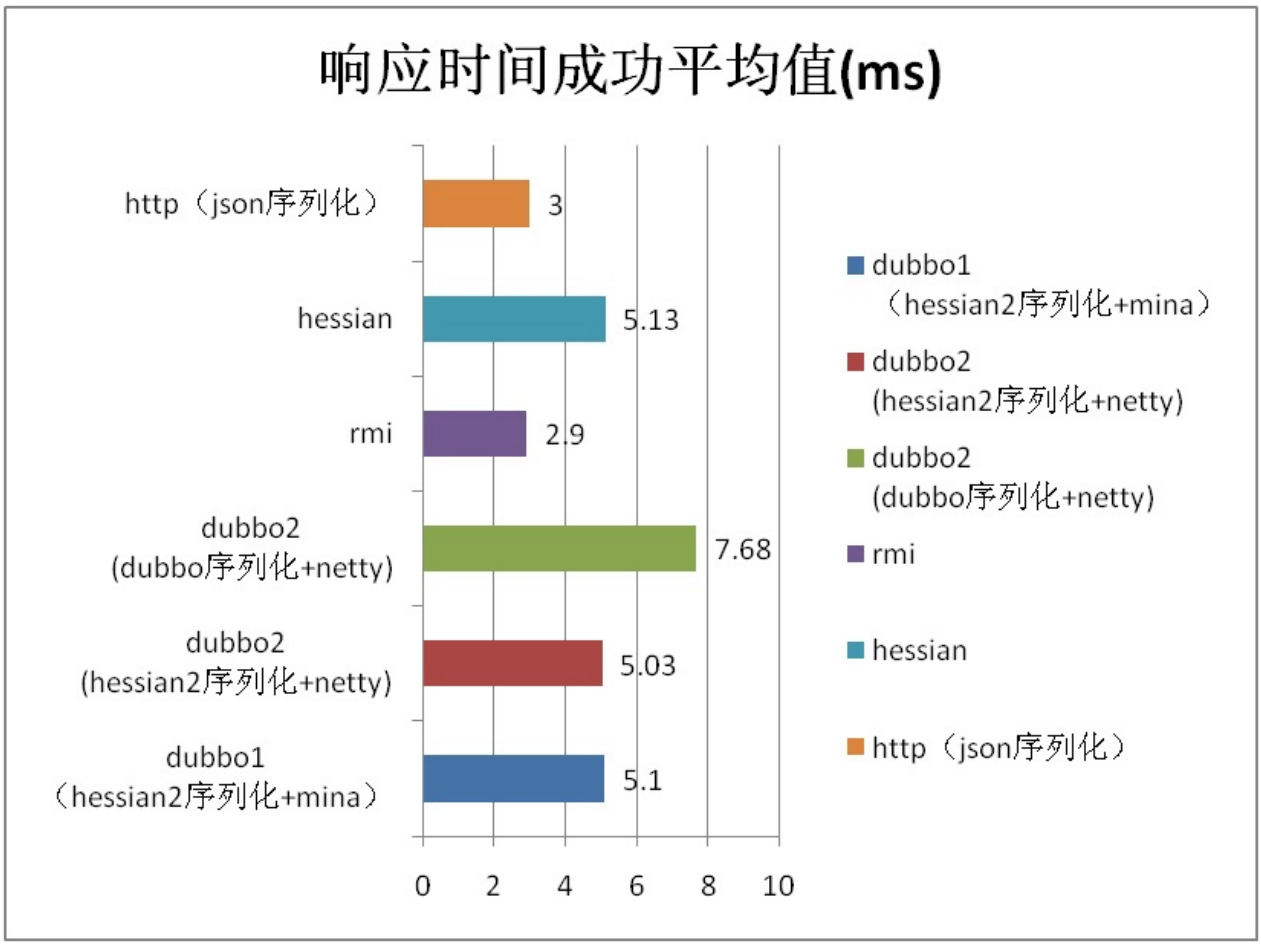
场景名称：**50k string** 场景

	TPS成功平均值	响应时间成功平均值**(ms)**
dubbo1（hessian2序列化+mina	1962.7	5.1
dubbo2 (hessian2序列化+netty)	1293	5.03
dubbo2 (dubbo序列化+netty)	1966	7.68
rmi	3349.88	2.9
hessian	1925.33	5.13
http（json序列化）	3247.1	3
2.0和1.0默认对比百分比	-34.12	-1.37
dubbo序列化相比hessian2序列化百分比	52.05	52.68

50K TPS



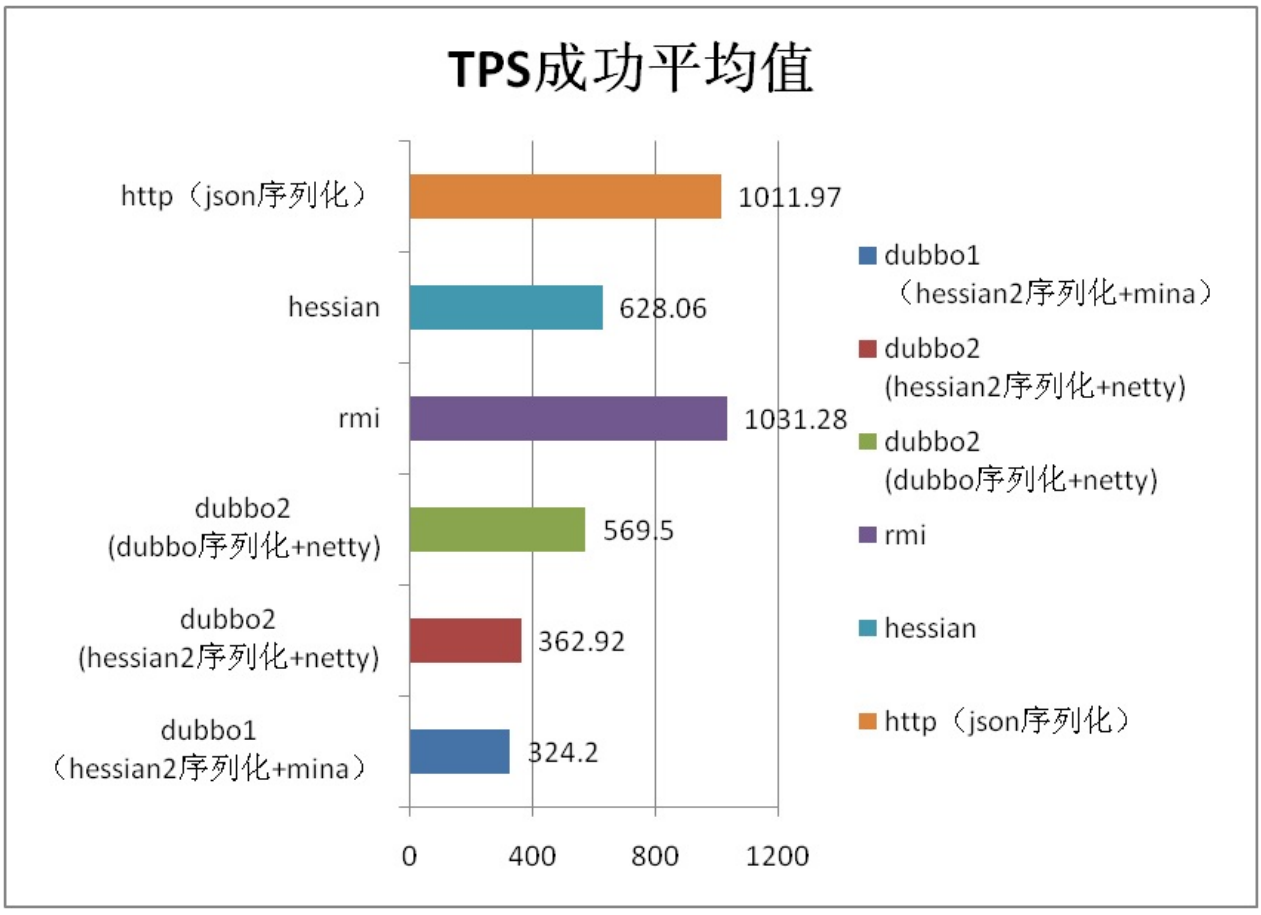
50K Response



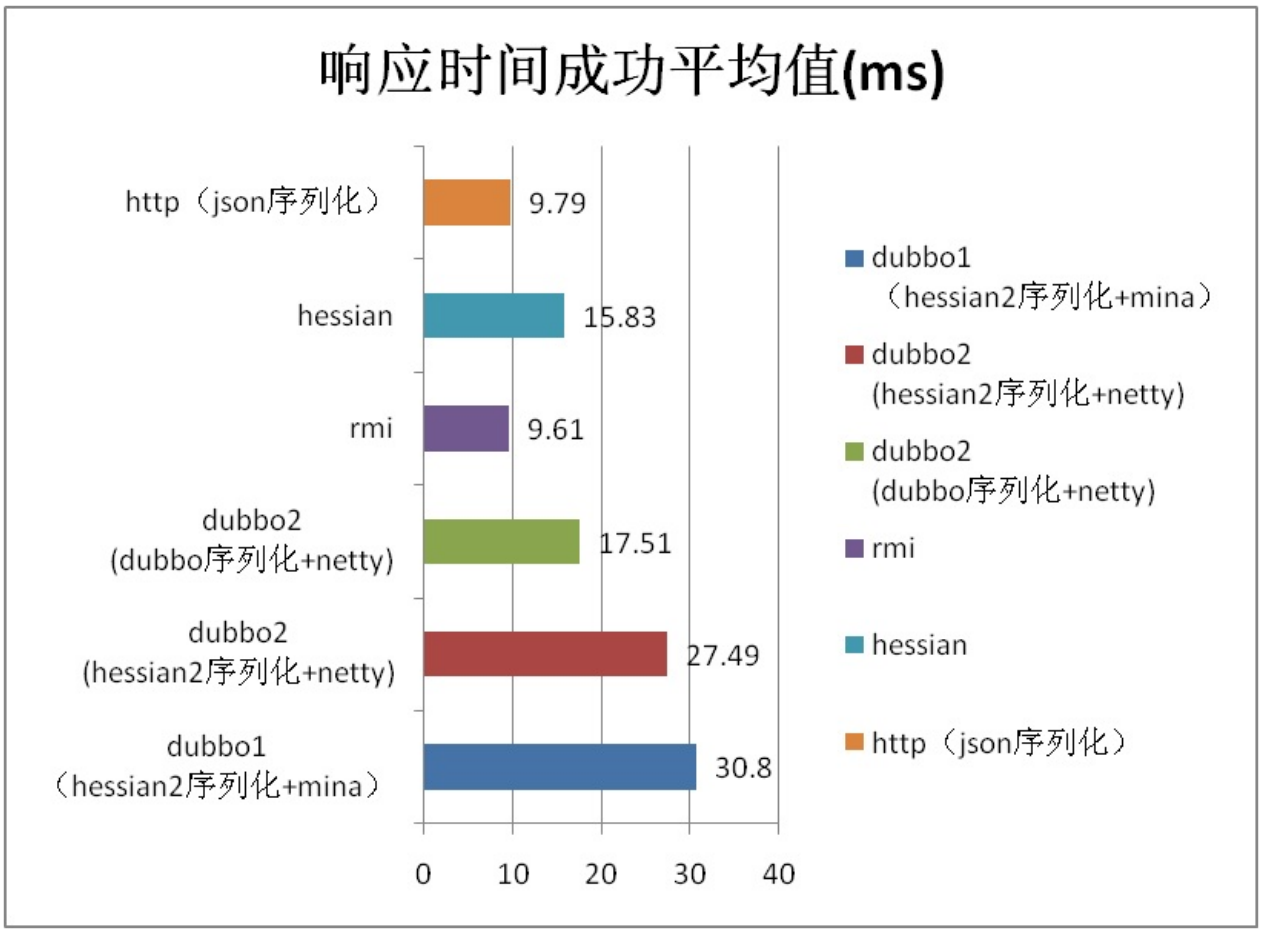
场景名称：**200k string** 场景

	TPS成功平均值	响应时间成功平均值**(ms)**
dubbo1（hessian2序列化+mina）	324.2	30.8
dubbo2 (hessian2序列化+netty)	362.92	27.49
dubbo2 (dubbo序列化+netty)	569.5	17.51
rmi	1031.28	9.61
hessian	628.06	15.83
http（json序列化）	1011.97	9.79
2.0和1.0默认对比百分比	11.94	-10.75
dubbo序列化相比hessian2序列化百分比	56.92	-36.30

200K TPS



200K Response



测试分析

性能分析评估

Dubbo 2.0 的性能测试结论为通过，从性能、内存占用和稳定性上都有了提高和改进。由其是内存管理由于将 mina 换成netty，大大减少了 1.0 版本在高并发大数据下的内存大锯齿。

性能对比分析(新旧环境、不同数据量级等)

Dubbo 2.0 相比较Dubbo 1.0（默认使用的都是 hessian2 序列化）性能均有提升（除了50k String），详见第五章的性能数据。

出于兼容性考虑默认的序列化方式和 1.0 保持一致使用 hessian2，如对性能有更高要求可以使用 dubbo 序列化，由其是在处理复杂对象时，在大数据量下能获得 50% 的提升（但此时已不建议使用 Dubbo 协议）。

Dubbo 的设计目的是为了满足高并发小数据量的 rpc 调用，在大数据量下的性能表现并不好，建议使用 rmi 或 http 协议。

测试局限性分析（可选）

本次性能测试考察的是 dubbo 本身的性能，实际使用过程中的性能有待应用来验证。

由于 dubbo 本身的性能占用都在毫秒级，占的基数很小，性能提升可能对应用整体的性能变化不大。

由于邮件篇幅所限没有列出所有的监控图，如需获得可在大力神平台上查询。

测试覆盖率报告

基于 `2.0.12` 版本，统计于 2012-02-03

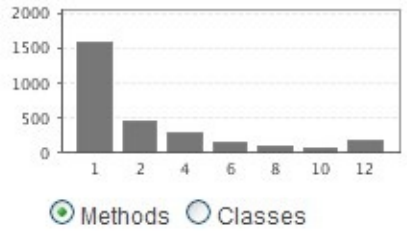
Lines of code
35,406
54,469 lines
17,315 statements
423 files

Classes
445
90 packages
2,762 methods
+347 accessors

Comments
9.2%
3,575 lines
32.1% docu. API
1,839 undocu. API
52 commented LOCs

Duplications
3.8%
2,077 lines
99 blocks
47 files

Complexity
3.8 /method
23.5 /class
24.7 /file
Total: 10,467



Code coverage
40.0%
43.4% line coverage
34.1% branch coverage

Test success
100.0%
0 failures
0 errors
817 tests
+102 skipped
55.9 sec

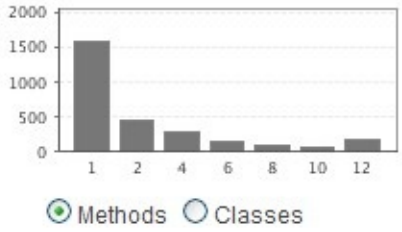
Lines of code
35,406
54,469 lines
17,315 statements
423 files

Classes
445
90 packages
2,762 methods
+347 accessors

Comments
9.2%
3,575 lines
32.1% docu. API
1,839 undocu. API
52 commented LOCs

Duplications
3.8%
2,077 lines
99 blocks
47 files

Complexity
3.8 /method
23.5 /class
24.7 /file
Total: 10,467



Code coverage
40.0%
43.4% line coverage
34.1% branch coverage

Test success
100.0%
0 failures
0 errors
817 tests
+102 skipped
55.9 sec

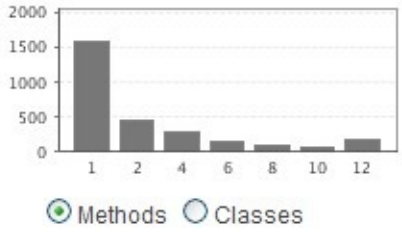
Lines of code
35,406
54,469 lines
17,315 statements
423 files

Classes
445
90 packages
2,762 methods
+347 accessors

Comments
9.2%
3,575 lines
32.1% docu. API
1,839 undocu. API
52 commented LOCs

Duplications
3.8%
2,077 lines
99 blocks
47 files

Complexity
3.8 /method
23.5 /class
24.7 /file
Total: 10,467



Code coverage
40.0%
43.4% line coverage
34.1% branch coverage

Test success
100.0%
0 failures
0 errors
817 tests
+102 skipped
55.9 sec

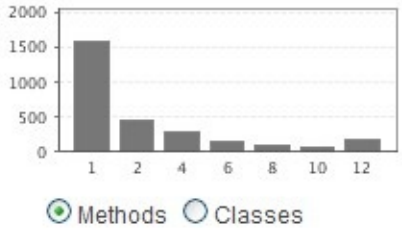
Lines of code
35,406
54,469 lines
17,315 statements
423 files

Classes
445
90 packages
2,762 methods
+347 accessors

Comments
9.2%
3,575 lines
32.1% docu. API
1,839 undocu. API
52 commented LOCs

Duplications
3.8%
2,077 lines
99 blocks
47 files

Complexity
3.8 /method
23.5 /class
24.7 /file
Total: 10,467



Code coverage
40.0%
43.4% line coverage
34.1% branch coverage

Test success
100.0%
0 failures
0 errors
817 tests
+102 skipped
55.9 sec

Violations
2,996 ▲

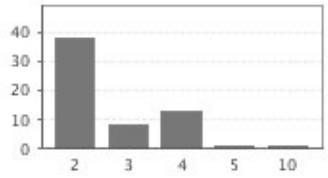
Rules compliance
80.0%

Blocker	0	
Critical	232	
Major	1,590	
Minor	1,145	▲
Info	29	

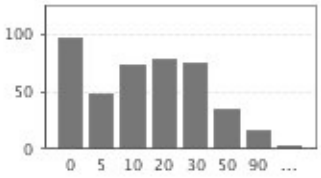
Package tangle index
4.1%
> 11 cycles

Dependencies to cut
7 between packages
14 between files

LCOM4
1.2 /class
14.4% files having LCOM4>1



RFC
25 /class



Violations
2,996 ▲

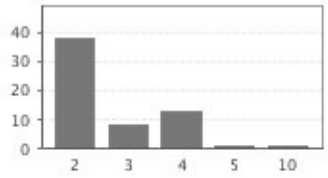
Rules compliance
80.0%

Blocker	0	
Critical	232	
Major	1,590	
Minor	1,145	▲
Info	29	

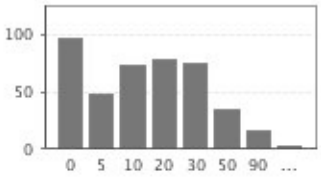
Package tangle index
4.1%
> 11 cycles

Dependencies to cut
7 between packages
14 between files

LCOM4
1.2 /class
14.4% files having LCOM4>1



RFC
25 /class



Violations
2,996

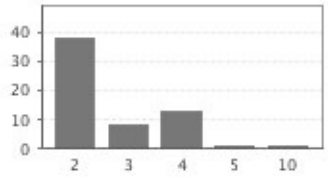
Rules compliance
80.0%

Blocker	0	
Critical	232	
Major	1,590	
Minor	1,145	
Info	29	

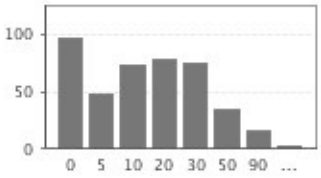
Package tangle index
4.1%
> 11 cycles

Dependencies to cut
7 between packages
14 between files

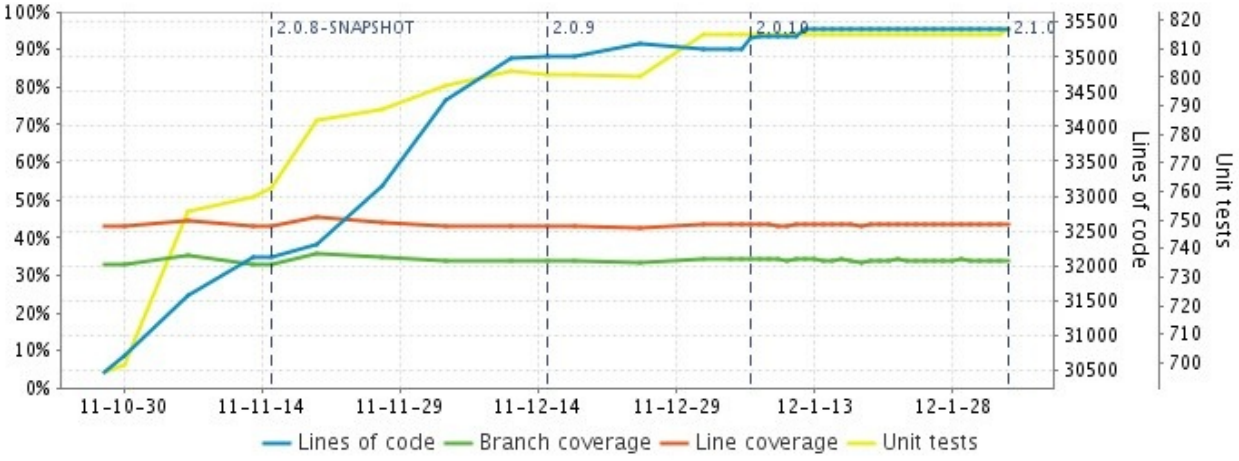
LCOM4
1.2 /class
14.4% files having LCOM4>1

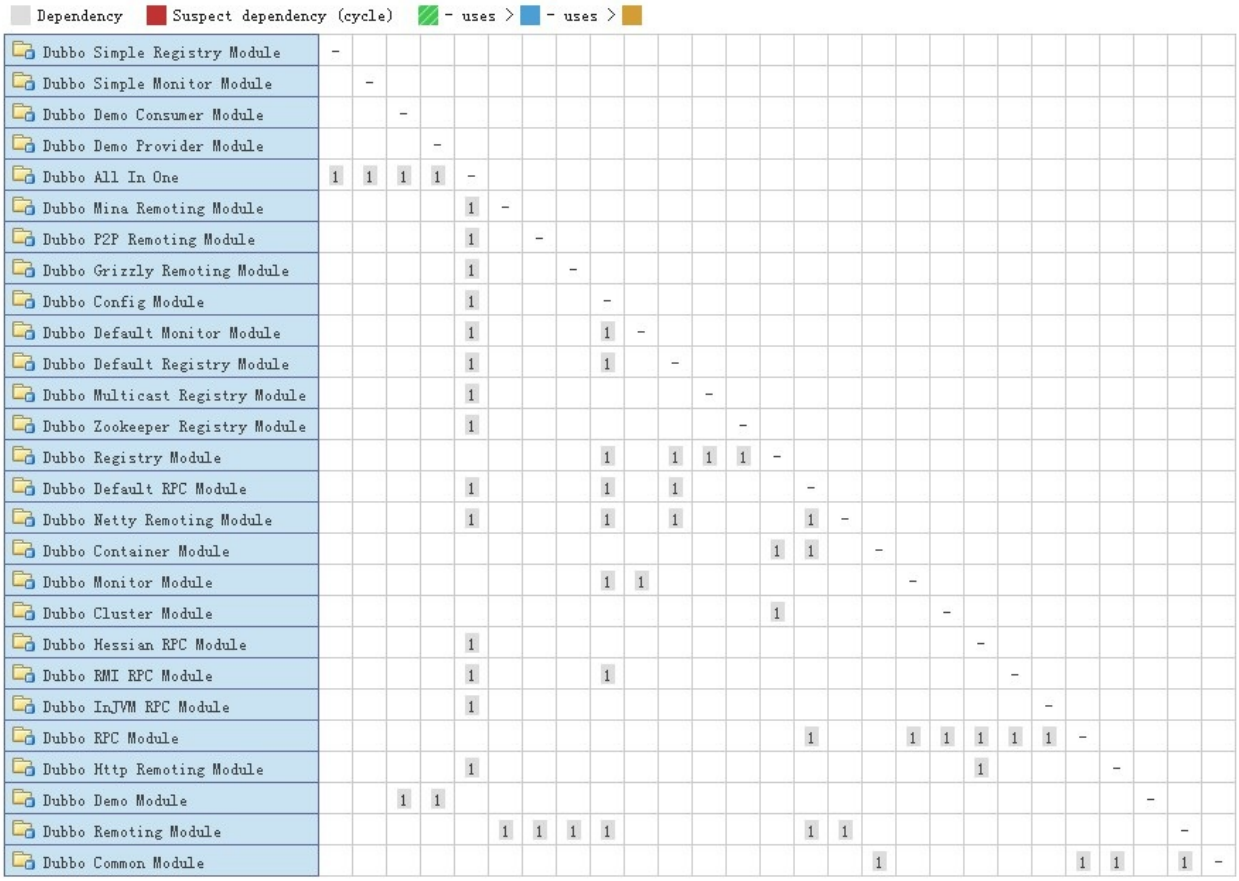


RFC
25 /class



Name	Rules compliance	Line coverage	Branch coverage
Dubbo Common Module	69.9%	58.1%	43.3%
Dubbo Remoting Module	82.6%	25.3%	20.7%
Dubbo Netty Remoting Module	79.9%	75.7%	53.6%
Dubbo Mina Remoting Module	81.2%	69.1%	42.7%
Dubbo Grizzly Remoting Module	79.6%	0.0%	0.0%
Dubbo P2P Remoting Module	85.1%	0.0%	0.0%
Dubbo Http Remoting Module	99.0%	0.0%	0.0%
Dubbo RPC Module	89.8%	23.9%	17.1%
Dubbo Default RPC Module	89.0%	61.3%	47.7%
Dubbo InJVM RPC Module	92.6%	76.2%	50.0%
Dubbo RMI RPC Module	88.3%	48.5%	29.8%
Dubbo Hessian RPC Module	87.9%	72.2%	39.1%
Dubbo Cluster Module	85.1%	66.3%	51.9%
Dubbo Registry Module	85.6%	1.7%	0.6%
Dubbo Default Registry Module	84.2%	48.1%	10.0%
Dubbo Multicast Registry Module	93.5%	65.8%	50.0%
Dubbo Zookeeper Registry Module	81.3%	12.9%	6.7%
Dubbo Monitor Module	93.9%	60.7%	37.5%
Dubbo Default Monitor Module	63.4%	47.1%	17.7%
Dubbo Config Module	88.6%	55.3%	46.5%
Dubbo Container Module	86.7%	15.5%	8.7%
Dubbo All In One			
Dubbo Simple Registry Module	98.7%	5.2%	0.0%
Dubbo Simple Monitor Module	87.2%	3.0%	0.6%
Dubbo Demo Module	100.0%		
Dubbo Demo Consumer Module	66.7%	0.0%	0.0%
Dubbo Demo Provider Module	72.7%	0.0%	


100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%
2.0.8-SNAPSHOT
2.0.9
2.0.10
2.1.0
35500
35000
34500
34000
33500
33000
32500
32000
31500
31000
30500
820
810
800
790
780
770
760
750
740
730
720
710
700
Lines of code
Unit tests
11-10-30
11-11-14
11-11-29
11-12-14
11-12-29
12-1-13
12-1-28
Lines of code
Branch coverage
Line coverage
Unit tests


Dependency
Suspect dependency (cycle)
- uses >
- uses >
Dubbo Simple Registry Module
Dubbo Simple Monitor Module
Dubbo Demo Consumer Module
Dubbo Demo Provider Module
Dubbo All In One
Dubbo Mina Remoting Module
Dubbo P2P Remoting Module
Dubbo Grizzly Remoting Module
Dubbo Config Module
Dubbo Default Monitor Module
Dubbo Default Registry Module
Dubbo Multicast Registry Module
Dubbo Zookeeper Registry Module
Dubbo Registry Module
Dubbo Default RPC Module
Dubbo Netty Remoting Module
Dubbo Container Module
Dubbo Monitor Module
Dubbo Cluster Module
Dubbo Hessian RPC Module
Dubbo RMI RPC Module
Dubbo InJVM RPC Module
Dubbo RPC Module
Dubbo Http Remoting Module
Dubbo Demo Module
Dubbo Remoting Module
Dubbo Common Module