# 1 入门

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

## 背景

# 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。

## 1.2 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 | 统计服务的调用次数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

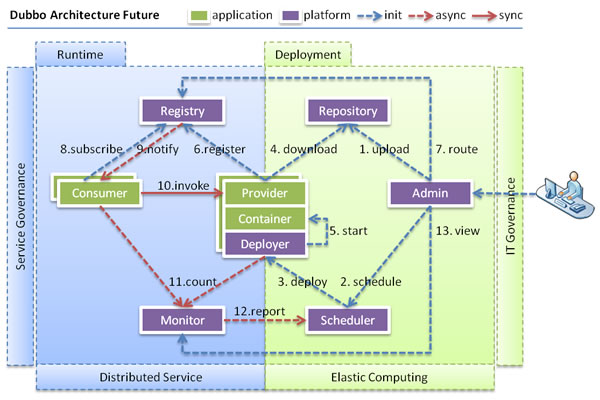
* 监控中心宕掉不影响使用，只是丢失部分采样数据
* 数据库宕掉后，注册中心仍能通过缓存提供服务列表查询，但不能注册新服务
* 注册中心对等集群，任意一台宕掉后，将自动切换到另一台
* 注册中心全部宕掉后，服务提供者和服务消费者仍能通过本地缓存通讯
* 服务提供者无状态，任意一台宕掉后，不影响使用
* 服务提供者全部宕掉后，服务消费者应用将无法使用，并无限次重连等待服务提供者恢复

## 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 | 统计服务的调用次数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](http://dubbo.io/books/configuration/api.md) 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](https://dubbo.gitbooks.io/dubbo-admin-book/install/provider-demo.html)

### 定义服务接口

1. DemoService.java [1](http://dubbo.io/books/dubbo-user-book/quick-start.html#fn_1)：
2. package com.alibaba.dubbo.demo;
3. public interface DemoService {
4. String sayHello(String name);
5. }

### 在服务提供方实现接口

1. DemoServiceImpl.java [2](http://dubbo.io/books/dubbo-user-book/quick-start.html#fn_2)：
2. package com.alibaba.dubbo.demo.provider;
3. import com.alibaba.dubbo.demo.DemoService;
4. public class DemoServiceImpl implements DemoService {
5. public String sayHello(String name) {
6. return "Hello " + name;
7. }
8. }

### 用 Spring 配置声明暴露服务

1. provider.xml：
2. <?xml version="1.0" encoding="UTF-8"?>
3. <beans xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd">
7. <!-- 提供方应用信息，用于计算依赖关系 -->
8. <dubbo:application name="hello-world-app" />
9. <!-- 使用multicast广播注册中心暴露服务地址 -->
10. <dubbo:registry address="multicast://224.5.6.7:1234" />
11. <!-- 用dubbo协议在20880端口暴露服务 -->
12. <dubbo:protocol name="dubbo" port="20880" />
13. <!-- 声明需要暴露的服务接口 -->
14. <dubbo:service interface="com.alibaba.dubbo.demo.DemoService" ref="demoService" />
15. <!-- 和本地bean一样实现服务 -->
16. <bean id="demoService" class="com.alibaba.dubbo.demo.provider.DemoServiceImpl" />
17. </beans>

### 加载 Spring 配置

1. Provider.java：
2. import org.springframework.context.support.ClassPathXmlApplicationContext;
3. public class Provider {
4. public static void main(String[] args) throws Exception {
5. ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(new String[] {"http://10.20.160.198/wiki/display/dubbo/provider.xml"});
6. context.start();
7. System.in.read(); // 按任意键退出
8. }
9. }

## 服务消费者

完整安装步骤，请参见：[示例消费者安装](https://dubbo.gitbooks.io/dubbo-admin-book/install/consumer-demo.html)

### 通过 Spring 配置引用远程服务

1. consumer.xml：
2. <?xml version="1.0" encoding="UTF-8"?>
3. <beans xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd">
7. <!-- 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 -->
8. <dubbo:application name="consumer-of-helloworld-app" />
9. <!-- 使用multicast广播注册中心暴露发现服务地址 -->
10. <dubbo:registry address="multicast://224.5.6.7:1234" />
11. <!-- 生成远程服务代理，可以和本地bean一样使用demoService -->
12. <dubbo:reference id="demoService" interface="com.alibaba.dubbo.demo.DemoService" />
13. </beans>

### 加载Spring配置，并调用远程服务

Consumer.java [3](http://dubbo.io/books/dubbo-user-book/quick-start.html#fn_3)：

1. import org.springframework.context.support.ClassPathXmlApplicationContext;
2. import com.alibaba.dubbo.demo.DemoService;
3. public class Consumer {
4. public static void main(String[] args) throws Exception {
5. ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(new String[] {"http://10.20.160.198/wiki/display/dubbo/consumer.xml"});
6. context.start();
7. DemoService demoService = (DemoService)context.getBean("demoService"); // 获取远程服务代理
8. String hello = demoService.sayHello("world"); // 执行远程方法
9. System.out.println( hello ); // 显示调用结果
10. }
11. }

1. 该接口需单独打包，在服务提供方和消费方共享[↩](http://dubbo.io/books/dubbo-user-book/quick-start.html#reffn_1)

2. 对服务消费方隐藏实现[↩](http://dubbo.io/books/dubbo-user-book/quick-start.html#reffn_2)

3. 也可以使用 IoC 注入[↩](http://dubbo.io/books/dubbo-user-book/quick-start.html#reffn_3)

# 3 依赖

## 必须依赖

JDK 1.5+ [1](http://dubbo.io/books/dubbo-user-book/dependencies.html#fn_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

这里所有依赖都是换照 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](http://dubbo.io/books/dubbo-user-book/dependencies.html#fn_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](http://dubbo.io/books/dubbo-user-book/dependencies.html#fn_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](http://dubbo.io/books/dubbo-user-book/dependencies.html#fn_4): 如果用 ServiceConfig和 ReferenceConfig 的 API 调用，则不需要。
* netty.jar [5](http://dubbo.io/books/dubbo-user-book/dependencies.html#fn_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](http://dubbo.io/books/dubbo-user-book/dependencies.html#fn_6)
* bsf: 3.1 [6](http://dubbo.io/books/dubbo-user-book/dependencies.html#fn_6)
* validation-api: 1.0.0.GA [6](http://dubbo.io/books/dubbo-user-book/dependencies.html#fn_6)
* jcache: 0.4 [6](http://dubbo.io/books/dubbo-user-book/dependencies.html#fn_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[↩](http://dubbo.io/books/dubbo-user-book/dependencies.html#reffn_1)

2. 日志输出包[↩](http://dubbo.io/books/dubbo-user-book/dependencies.html#reffn_2)

3. 字节码生成[↩](http://dubbo.io/books/dubbo-user-book/dependencies.html#reffn_3)

4. 配置解析[↩](http://dubbo.io/books/dubbo-user-book/dependencies.html#reffn_4)

5. 网络传输[↩](http://dubbo.io/books/dubbo-user-book/dependencies.html#reffn_5)

6. JEE[↩](http://dubbo.io/books/dubbo-user-book/dependencies.html#reffn_6)

# 4 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** | **User** |
| --- | --- | --- | --- | --- | --- |
| 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 | Alibaba |
| 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 | Alibaba |
| 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 | Alibaba |
| 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 | Alibaba |
| 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（推荐使用） | Hessian的各版本兼容性不好，可能和应用使用的Hessian冲突，Dubbo内嵌了hessian3.2.1的源码 | 可用于生产环境 | Alibaba |
| 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 | Alibaba |
| 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 | Alibaba |
| 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 | Alibaba |
| 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 | Monitor |
| Failback Cluster | Tested | 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 | 不可靠，重启丢失 | 可用于生产环境 | Registry |
| 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 | Alibaba |
| 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，功能简单易用 | 有些复杂多分支条件情况，规则很难描述 | 可用于生产环境 | Alibaba |
| 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 | Alibaba |
| 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 | Alibaba |
| 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 | Alibaba |

# 5 配置

## 5.1 XML 配置

有关 XML 的详细配置项，请参见：[配置参考手册](http://dubbo.io/books/dubbo-user-book/references/xml/introduction.html)。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](http://dubbo.io/books/dubbo-user-book/configuration/api.html)。想知道如何使用配置，请参见：[快速启动](http://dubbo.io/books/dubbo-user-book/quick-start.html)。

## 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.springframework.org/schema/beans/spring-beans.xsd http://code.alibabatech.com/schema/dubbo http://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="demoServiceLocal" />

<dubbo:reference id="demoServiceRemote" interface="com.alibaba.dubbo.demo.DemoService" />

</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](http://dubbo.io/books/dubbo-user-book/configuration/xml.html#fn_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"

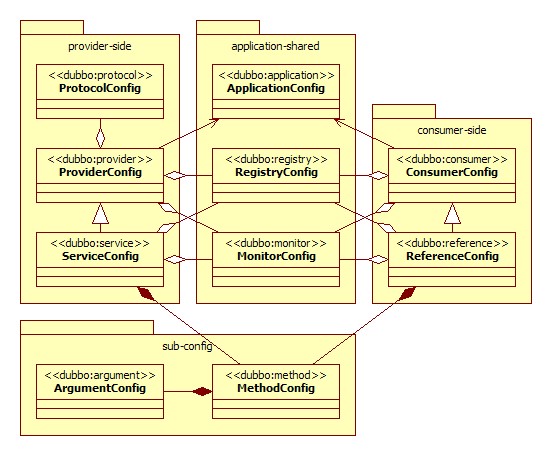
xmlns:p="http://www.springframework.org/schema/p"

xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd">

<dubbo:protocol name="jms" p:queue="your\_queue" />

</beans>

## 配置之间的关系



| **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](http://dubbo.io/books/dubbo-user-book/configuration/xml.html#fn_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"[↩](http://dubbo.io/books/dubbo-user-book/configuration/xml.html#reffn_1)

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" />[↩](http://dubbo.io/books/dubbo-user-book/configuration/xml.html#reffn_2)

## 5.2 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](http://dubbo.io/books/dubbo-user-book/configuration/properties.html#fn_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](http://dubbo.io/books/dubbo-user-book/configuration/properties.html#fn_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。[↩](http://dubbo.io/books/dubbo-user-book/configuration/properties.html#reffn_1)

2. 协议的 id 没配时，缺省使用协议名作为 id[↩](http://dubbo.io/books/dubbo-user-book/configuration/properties.html#reffn_2)

## 5.3 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](http://dubbo.io/books/dubbo-user-book/references/xml/introduction.html)，比如：ApplicationConfig.setName("xxx") 对应 <dubbo:application name="xxx" /> [1](http://dubbo.io/books/dubbo-user-book/configuration/api.html#fn_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](http://dubbo.io/books/dubbo-user-book/configuration/xml.html)方式使用 Dubbo[↩](http://dubbo.io/books/dubbo-user-book/configuration/api.html#reffn_1)

## 5.3 注解配置

需要 2.5.7 及以上版本支持

## 服务提供方

### Service注解暴露服务

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

@Service(timeout = 5000)

public class AnnotateServiceImpl implements AnnotateService {

// ...

}

### javaconfig形式配置公共模块

@Configuration

public class DubboConfiguration {

@Bean

public ApplicationConfig applicationConfig() {

ApplicationConfig applicationConfig = new ApplicationConfig();

applicationConfig.setName("provider-test");

return applicationConfig;

}

@Bean

public RegistryConfig registryConfig() {

RegistryConfig registryConfig = new RegistryConfig();

registryConfig.setAddress("zookeeper://127.0.0.1:2181");

registryConfig.setClient("curator");

return registryConfig;

}

}

### 指定dubbo扫描路径

@SpringBootApplication

@DubboComponentScan(basePackages = "com.alibaba.dubbo.test.service.impl")

public class ProviderTestApp {

// ...

}

## 服务消费方

### Reference注解引用服务

public class AnnotationConsumeService {

@com.alibaba.dubbo.config.annotation.Reference

public AnnotateService annotateService;

// ...

}

### javaconfig形式配置公共模块

@Configuration

public class DubboConfiguration {

@Bean

public ApplicationConfig applicationConfig() {

ApplicationConfig applicationConfig = new ApplicationConfig();

applicationConfig.setName("consumer-test");

return applicationConfig;

}

@Bean

public ConsumerConfig consumerConfig() {

ConsumerConfig consumerConfig = new ConsumerConfig();

consumerConfig.setTimeout(3000);

return consumerConfig;

}

@Bean

public RegistryConfig registryConfig() {

RegistryConfig registryConfig = new RegistryConfig();

registryConfig.setAddress("zookeeper://127.0.0.1:2181");

registryConfig.setClient("curator");

return registryConfig;

}

}

### 指定dubbo扫描路径

@SpringBootApplication

@DubboComponentScan(basePackages = "com.alibaba.dubbo.test.service")

public class ConsumerTestApp {

// ...

}

## 注意

如果你曾使用旧版annotation配置，请删除所有相关配置，我们将在下个版本删除所有旧版配置项。

<dubbo:annotation package="com.alibaba.dubbo.test.service" />

# 6 示例

## 6.1 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

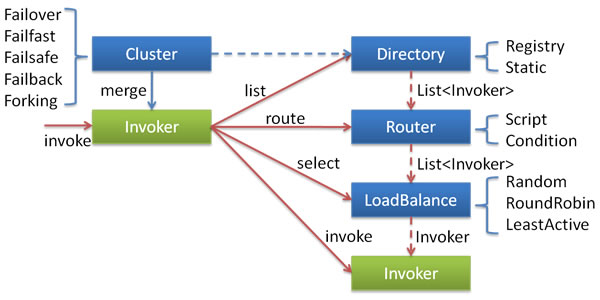
dubbo.reference.check=false，强制改变所有 reference 的 check 值，就算配置中有声明，也会被覆盖。

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。

## 6.2 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](https://dubbo.gitbooks.io/dubbo-dev-book/impls/cluster.html)

### Failover Cluster

失败自动切换，当出现失败，重试其它服务器 [1](http://dubbo.io/books/dubbo-user-book/demos/fault-tolerent-strategy.html#fn_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](http://dubbo.io/books/dubbo-user-book/demos/fault-tolerent-strategy.html#fn_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[↩](http://dubbo.io/books/dubbo-user-book/demos/fault-tolerent-strategy.html#reffn_1)

2. 2.1.0 开始支持[↩](http://dubbo.io/books/dubbo-user-book/demos/fault-tolerent-strategy.html#reffn_2)

## 6.3 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](https://dubbo.gitbooks.io/dubbo-dev-book/content/impls/load-balance.html)

## 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"/>

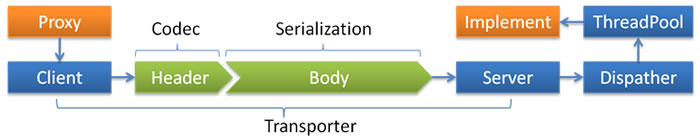
</dubbo:reference>

## 6.4 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

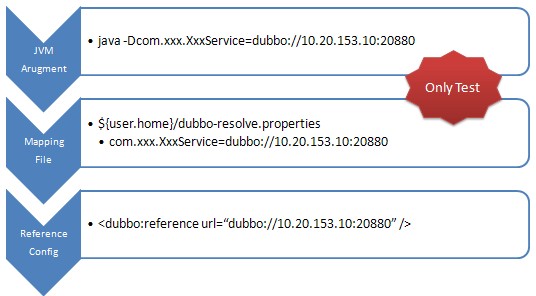
* all 所有消息都派发到线程池，包括请求，响应，连接事件，断开事件，心跳等。
* direct 所有消息都不派发到线程池，全部在 IO 线程上直接执行。
* message 只有请求响应消息派发到线程池，其它连接断开事件，心跳等消息，直接在 IO 线程上执行。
* execution 只请求消息派发到线程池，不含响应，响应和其它连接断开事件，心跳等消息，直接在 IO 线程上执行。
* connection 在 IO 线程上，将连接断开事件放入队列，有序逐个执行，其它消息派发到线程池。

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。

## 6.5 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](http://dubbo.io/books/dubbo-user-book/demos/explicit-target.html#fn_1)：

<dubbo:reference id="xxxService" interface="com.alibaba.xxx.XxxService" url="dubbo://localhost: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](http://dubbo.io/books/dubbo-user-book/demos/explicit-target.html#fn_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](http://dubbo.io/books/dubbo-user-book/demos/explicit-target.html#fn_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[↩](http://dubbo.io/books/dubbo-user-book/demos/explicit-target.html#reffn_1)

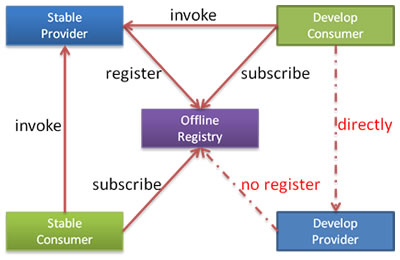
2. key 为服务名，value 为服务提供者 url，此配置优先级最高，1.0.15 及以上版本支持[↩](http://dubbo.io/books/dubbo-user-book/demos/explicit-target.html#reffn_2)

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[↩](http://dubbo.io/books/dubbo-user-book/demos/explicit-target.html#reffn_3)

## 6.6 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" />

## 6.7 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" />

## 6.8 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](http://dubbo.io/books/dubbo-user-book/demos/static-service.html#fn_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[↩](http://dubbo.io/books/dubbo-user-book/demos/static-service.html#reffn_1)

## 6.9 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.springframework.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" password="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.springframework.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" password="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" version="1.0.0" protocol="dubbo,hessian" />

</beans>

1. 可以自行扩展协议，参见：[协议扩展](https://dubbo.gitbooks.io/dubbo-dev-book/content/impls/protocol.html)[↩](http://dubbo.io/books/dubbo-user-book/demos/multi-protocols.html#reffn_1)

## 6.10 多注册中心

ne# 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](http://dubbo.io/books/dubbo-user-book/demos/multi-registry.html#fn_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.springframework.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.springframework.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.springframework.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.HelloService" version="1.0.0" registry="chinaRegistry" />

<!-- 引用国际站站服务 -->

<dubbo:reference id="intlHelloService" interface="com.alibaba.hello.api.HelloService" 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.springframework.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](https://dubbo.gitbooks.io/dubbo-dev-book/content/impls/registry.html)[↩](http://dubbo.io/books/dubbo-user-book/demos/multi-registry.html#reffn_1)

## 6.11 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.IndexService" />

<dubbo:reference id="memberIndexService" group="member" interface="com.xxx.IndexService" />

任意组 [1](http://dubbo.io/books/dubbo-user-book/demos/service-group.html#fn_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[↩](http://dubbo.io/books/dubbo-user-book/demos/service-group.html#reffn_1)

## 6.12 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](http://dubbo.io/books/dubbo-user-book/demos/multi-versions.html#fn_1)：

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

1. 2.2.0 以上版本支持[↩](http://dubbo.io/books/dubbo-user-book/demos/multi-versions.html#reffn_1)

## 6.13 分组聚合

按组合并返回结果 [1](http://dubbo.io/books/dubbo-user-book/demos/group-merger.html#fn_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](https://github.com/alibaba/dubbo/tree/master/dubbo-test/dubbo-test-examples/src/main/java/com/alibaba/dubbo/examples/merge)

## 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](http://dubbo.io/books/dubbo-user-book/demos/group-merger.html#fn_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[↩](http://dubbo.io/books/dubbo-user-book/demos/group-merger.html#reffn_1)

2. 参见：[合并结果扩展](https://dubbo.gitbooks.io/dubbo-dev-book/content/impls/merger.html)[↩](http://dubbo.io/books/dubbo-user-book/demos/group-merger.html#reffn_2)

## 6.14 参数验证

参数验证功能 [1](http://dubbo.io/books/dubbo-user-book/demos/parameter-validation.html#fn_1) 是基于 [JSR303](https://jcp.org/en/jsr/detail?id=303) 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](http://dubbo.io/books/dubbo-user-book/demos/parameter-validation.html#fn_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(config);

context.start();

ValidationService validationService = (ValidationService)context.getBean("validationService");

// 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.getCause(); // 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](https://github.com/alibaba/dubbo/tree/master/dubbo-test/dubbo-test-examples/src/main/java/com/alibaba/dubbo/examples/validation)[↩](http://dubbo.io/books/dubbo-user-book/demos/parameter-validation.html#reffn_1)

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](https://dubbo.gitbooks.io/dubbo-dev-book/content/impls/validation.html)[↩](http://dubbo.io/books/dubbo-user-book/demos/parameter-validation.html#reffn_2)

## 6.15 结果缓存

结果缓存 [1](http://dubbo.io/books/dubbo-user-book/demos/result-cache.html#fn_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](http://dubbo.io/books/dubbo-user-book/demos/result-cache.html#fn_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](http://jcp.org/en/jsr/detail?id=107%27) 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](http://dubbo.io/developer-guide/impls/cache.html)

## 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[↩](http://dubbo.io/books/dubbo-user-book/demos/result-cache.html#reffn_1)

2. [示例代码](https://github.com/alibaba/dubbo/tree/master/dubbo-test/dubbo-test-examples/src/main/java/com/alibaba/dubbo/examples/cache)[↩](http://dubbo.io/books/dubbo-user-book/demos/result-cache.html#reffn_2)

## 6.16 泛化引用

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");

## 6.17 泛化实现

泛接口实现方式主要用于服务器端没有API接口及模型类元的情况，参数及返回值中的所有POJO均用Map表示，通常用于框架集成，比如：实现一个通用的远程服务Mock框架，可通过实现GenericService接口处理所有服务请求。

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) throws 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();

## 6.18 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"));

## 6.19 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](http://dubbo.io/books/dubbo-user-book/references/xml/introduction.html) 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("application");

// 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

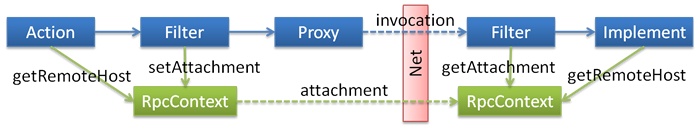
boolean isProviderSide = RpcContext.getContext().isProviderSide();

}

}

## 6.20 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](http://dubbo.io/books/dubbo-user-book/demos/attachment.html#fn_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");

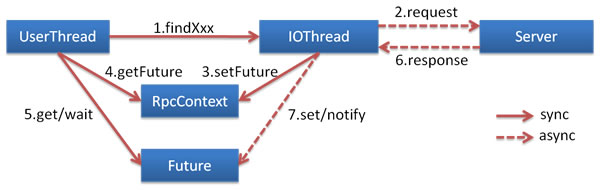
}

}

1. 注意：path, group, version, dubbo, token, timeout 几个 key 是保留字段，请使用其它值。[↩](http://dubbo.io/books/dubbo-user-book/demos/attachment.html#reffn_1)

## 6.21 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](http://dubbo.io/books/dubbo-user-book/demos/async-call.html#fn_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](http://dubbo.io/books/dubbo-user-book/demos/async-call.html#fn_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[↩](http://dubbo.io/books/dubbo-user-book/demos/async-call.html#reffn_1)

2. 异步总是不等待返回[↩](http://dubbo.io/books/dubbo-user-book/demos/async-call.html#reffn_2)

## 6.22 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" />

## 6.23 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](http://dubbo.io/books/dubbo-user-book/demos/callback-parameter.html#fn_1)。可以参考 [dubbo 项目中的示例代码](https://github.com/alibaba/dubbo/tree/master/dubbo-test/dubbo-test-examples/src/main/java/com/alibaba/dubbo/examples/callback)。

#### 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<String, CallbackListener>();

public CallbackServiceImpl() {

Thread t = new Thread(new Runnable() {

public void run() {

while(true) {

try {

for(Map.Entry<String, CallbackListener> entry : listeners.entrySet()){

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 Date());

}

}

#### 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" connections="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 CallbackListener(){

public void changed(String msg) {

System.out.println("callback1:" + msg);

}

});

1. 2.0.6 及其以上版本支持[↩](http://dubbo.io/books/dubbo-user-book/demos/callback-parameter.html#reffn_1)

## 6.24 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](http://dubbo.io/books/dubbo-user-book/demos/events-notify.html#fn_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="demoService" 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.IDemoService" 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](http://dubbo.io/books/dubbo-user-book/demos/events-notify.html#fn_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;

}

}

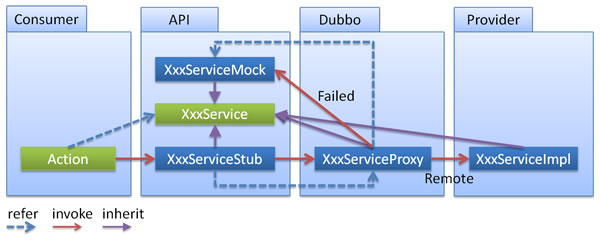
Assert.assertEquals(requestId, notify.ret.get(requestId).getId());

1. 支持版本：2.0.7 之后[↩](http://dubbo.io/books/dubbo-user-book/demos/events-notify.html#reffn_1)

2. async=false 默认[↩](http://dubbo.io/books/dubbo-user-book/demos/events-notify.html#reffn_2)

## 6.25 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](http://dubbo.io/books/dubbo-user-book/demos/local-stub.html#fn_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](http://dubbo.io/books/dubbo-user-book/demos/local-stub.html#fn_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。[↩](http://dubbo.io/books/dubbo-user-book/demos/local-stub.html#reffn_1)

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[↩](http://dubbo.io/books/dubbo-user-book/demos/local-stub.html#reffn_2)

## 6.26 本地伪装

本地伪装 [1](http://dubbo.io/books/dubbo-user-book/demos/local-mock.html#fn_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](http://dubbo.io/books/dubbo-user-book/demos/local-mock.html#fn_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。[↩](http://dubbo.io/books/dubbo-user-book/demos/local-mock.html#reffn_1)

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[↩](http://dubbo.io/books/dubbo-user-book/demos/local-mock.html#reffn_2)

## 6.27 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](http://dubbo.io/books/dubbo-user-book/demos/delay-publish.html#fn_1)

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

### Spring 2.x 初始化死锁问题

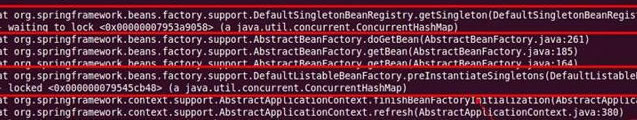
#### 触发条件

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。



1. 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[↩](http://dubbo.io/books/dubbo-user-book/demos/delay-publish.html#reffn_1)

## 6.28 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](http://dubbo.io/books/dubbo-user-book/demos/user-guide-configuration#%E9%85%8D%E7%BD%AE%E8%A6%86%E7%9B%96)。

## 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" />

## 6.29 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](http://dubbo.io/books/dubbo-user-book/demos/config-connections.html#fn_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](http://dubbo.io/books/dubbo-user-book/demos/config-connections.html#fn_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](http://dubbo.io/books/dubbo-user-book/configuration/xml.html)

1. 因为连接在 Server上，所以配置在 Provider 上[↩](http://dubbo.io/books/dubbo-user-book/demos/config-connections.html#reffn_1)

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[↩](http://dubbo.io/books/dubbo-user-book/demos/config-connections.html#reffn_2)

## 6.30 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](http://dubbo.io/books/dubbo-user-book/demos/lazy-connect.html#fn_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。[↩](http://dubbo.io/books/dubbo-user-book/demos/lazy-connect.html#reffn_1)

## 6.31 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](http://dubbo.io/books/dubbo-user-book/demos/lazy-connect.html)，以减少长连接数。

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

## 6.32 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" />

## 6.33 路由规则

路由规则 [1](http://dubbo.io/books/dubbo-user-book/demos/routing-rule.html#fn_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](http://dubbo.io/books/dubbo-user-book/demos/routing-rule.html#fn_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：
2. => host != 172.22.3.91
3. 白名单 [3](http://dubbo.io/books/dubbo-user-book/demos/routing-rule.html#fn_3)：
4. host != 10.20.153.10,10.20.153.11 =>
5. 黑名单：
6. host = 10.20.153.10,10.20.153.11 =>
7. 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：
8. => host = 172.22.3.1\*,172.22.3.2\*
9. 为重要应用提供额外的机器：
10. application != kylin => host != 172.22.3.95,172.22.3.96
11. 读写分离：
12. method = find\*,list\*,get\*,is\* => host = 172.22.3.94,172.22.3.95,172.22.3.96
13. method != find\*,list\*,get\*,is\* => host = 172.22.3.97,172.22.3.98
14. 前后台分离：
15. application = bops => host = 172.22.3.91,172.22.3.92,172.22.3.93
16. application != bops => host = 172.22.3.94,172.22.3.95,172.22.3.96
17. 隔离不同机房网段：
18. host != 172.22.3.\* => host != 172.22.3.\*
19. 提供者与消费者部署在同集群内，本机只访问本机的服务：
20. => host = $host

## 脚本路由规则

脚本路由规则 [4](http://dubbo.io/books/dubbo-user-book/demos/routing-rule.html#fn_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.encode("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[↩](http://dubbo.io/books/dubbo-user-book/demos/routing-rule.html#reffn_1)

2. 路由规则扩展点：[路由扩展](http://dubbo.io/developer-guide/impls/router.html)[↩](http://dubbo.io/books/dubbo-user-book/demos/routing-rule.html#reffn_2)

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[↩](http://dubbo.io/books/dubbo-user-book/demos/routing-rule.html#reffn_3)

4. 注意：脚本没有沙箱约束，可执行任意代码，存在后门风险[↩](http://dubbo.io/books/dubbo-user-book/demos/routing-rule.html#reffn_4)

## 6.34 配置规划

向注册中心写入动态配置覆盖规则 [1](http://dubbo.io/books/dubbo-user-book/demos/config-rule.html#fn_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)
2. override://10.20.153.10/com.foo.BarService?category=configurators&dynamic=false&disbaled=true
3. 调整权重：(通常用于容量评估，缺省权重为 100)
4. override://10.20.153.10/com.foo.BarService?category=configurators&dynamic=false&weight=200
5. 调整负载均衡策略：(缺省负载均衡策略为 random)
6. override://10.20.153.10/com.foo.BarService?category=configurators&dynamic=false&loadbalance=leastactive
7. 服务降级：(通常用于临时屏蔽某个出错的非关键服务)
8. 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[↩](http://dubbo.io/books/dubbo-user-book/demos/config-rule.html#reffn_1)

## 6.35 服务降级

可以通过服务降级功能 [1](http://dubbo.io/books/dubbo-user-book/demos/service-downgrade.html#fn_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[↩](http://dubbo.io/books/dubbo-user-book/demos/service-downgrade.html#reffn_1)

## 6.36 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.properties

dubbo.service.shutdown.wait=15000

如果 ShutdownHook 不能生效，可以自行调用，**使用tomcat等容器部署的場景，建议通过扩展ContextListener等自行调用以下代码实现优雅停机**：

ProtocolConfig.destroyAll();

## 6.37 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：
2. test1 205.182.23.201
3. 在 dubbo.xml 中加入主机地址的配置：
4. <dubbo:protocol host="205.182.23.201">
5. 或在 dubbo.properties 中加入主机地址的配置：
6. 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：
2. <dubbo:protocol name="dubbo" port="20880">
3. 或在 dubbo.properties 中加入主机地址的配置：

dubbo.protocol.dubbo.port=20880

## 6.38 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](http://dubbo.io/books/dubbo-user-book/demos/logger-strategy.html#fn_1)，也可以通过以下方式显示配置日志输出策略：

1. 命令行
2. java -Ddubbo.application.logger=log4j
3. 在 dubbo.properties 中指定
4. dubbo.application.logger=log4j
5. 在 dubbo.xml 中配置
6. <dubbo:application logger="log4j" />

1. 自定义扩展可以参考[日志适配扩展](https://dubbo.gitbooks.io/dubbo-dev-book/content/impls/logger-adapter.html)[↩](http://dubbo.io/books/dubbo-user-book/demos/logger-strategy.html#reffn_1)

## 6.39 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" />

## 6.40 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](https://dubbo.gitbooks.io/dubbo-dev-book/content/impls/container.html)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

## 6.41 Reference Config 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。

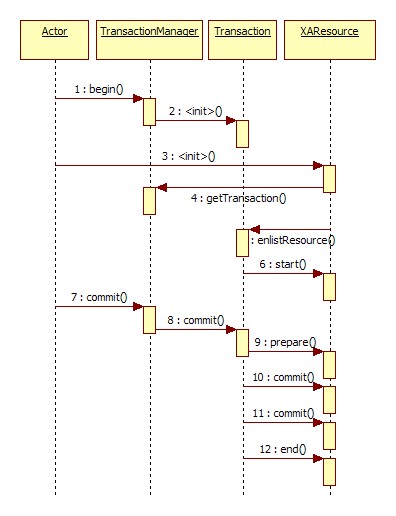
KeyGenerator keyGenerator = new ...

ReferenceConfigCache cache = ReferenceConfigCache.getCache(keyGenerator );

## 6.42 分布式缓存

分布式事务基于 JTA/XA 规范实现 [1](http://dubbo.io/books/dubbo-user-book/demos/distributed-transaction.html#fn_1)。

两阶段提交：



1. 本功能暂未实现[↩](http://dubbo.io/books/dubbo-user-book/demos/distributed-transaction.html#reffn_1)

## 6.43 线程栈自动dump

当业务线程池满时，我们需要知道线程都在等待哪些资源、条件，以找到系统的瓶颈点或异常点。dubbo通过Jstack自动导出线程堆栈来保留现场，方便排查问题

默认策略:

* 导出路径，user.home标识的用户主目录
* 导出间隔，最短间隔允许每隔10分钟导出一次

指定导出路径：

# dubbo.properties

dubbo.application.dump.directory=/tmp

<dubbo:application ...>

<dubbo:parameter key="dump.directory" value="/tmp" />

</dubbo:application>

## 6.44 Netty 4

dubbo 2.5.6版本新增了对netty4通信模块的支持，启用方式如下

provider端：

<dubbo:protocol server="netty4" />

或

<dubbo:provider server="netty4" />

consumer端：

<dubbo:consumer client="netty4" />

**注意**

1. provider端如需不同的协议使用不同的通信层框架，请配置多个protocol分别设置
2. consumer端请使用如下形式：
3. <dubbo:consumer client="netty">
4. <dubbo:reference />
5. </dubbo:consumer>
6. <dubbo:consumer client="netty4">
7. <dubbo:reference />
8. </dubbo:consumer>

接下来我们会继续完善：

1. 性能测试指标及与netty3版本的性能测试对比，我们会提供一份参考数据

# 7 API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](http://dubbo.io/books/dubbo-user-book/references/api.html#fn_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](http://dubbo.io/books/dubbo-user-book/configuration/api.html)

## 注解 API

com.alibaba.dubbo.config.annotation.Service

com.alibaba.dubbo.config.annotation.Reference

详细参见：[注解配置](http://dubbo.io/books/dubbo-user-book/configuration/annotation.html)

## 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](http://dubbo.io/books/dubbo-user-book/demos/context.html) & [隐式传参](http://dubbo.io/books/dubbo-user-book/demos/attachment.html) & [异步调用](http://dubbo.io/books/dubbo-user-book/demos/async-call.html)

## 服务API

com.alibaba.dubbo.rpc.service.GenericService

com.alibaba.dubbo.rpc.service.GenericException

详细参见：[泛化引用](http://dubbo.io/books/dubbo-user-book/demos/generic-reference.html) & [泛化实现](http://dubbo.io/books/dubbo-user-book/demos/generic-service.html)

com.alibaba.dubbo.rpc.service.EchoService

详细参见：[回声测试](http://dubbo.io/books/dubbo-user-book/demos/echo-service.html)

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。[↩](http://dubbo.io/books/dubbo-user-book/references/api.html#reffn_1)

# 8 Schema 配置参考手册

这里以 XML Config [1](http://dubbo.io/books/dubbo-user-book/references/xml/introduction.html#fn_1) 为准，列举所有配置项 [2](http://dubbo.io/books/dubbo-user-book/references/xml/introduction.html#fn_2)。其它配置方式，请参见相应转换关系：[属性配置](http://dubbo.io/books/dubbo-user-book/configuration/properties.html)，[注解配置](http://dubbo.io/books/dubbo-user-book/configuration/annotation.html)，[API 配置](http://dubbo.io/books/dubbo-user-book/configuration/api.html)。

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](http://dubbo.io/books/dubbo-user-book/references/xml/introduction.html#fn_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>[↩](http://dubbo.io/books/dubbo-user-book/references/xml/introduction.html#reffn_1)

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。[↩](http://dubbo.io/books/dubbo-user-book/references/xml/introduction.html#reffn_2)

3. URL 格式：protocol://username:password@host:port/path?key=value&key=value[↩](http://dubbo.io/books/dubbo-user-book/references/xml/introduction.html#reffn_3)