Table of Contents

Introduction	1.1
1 入门	1.2
1.1 背景	1.2.1
1.2 需求	1.2.2
1.3 架构	1.2.3
1.4 用法	1.2.4
2 快速启动	1.3
2.1 依赖	1.3.1
2.2 成熟度	1.3.2
3配置	1.4
3.1 XML配置	1.4.1
3.2 属性配置	1.4.2
3.3 API配置	1.4.3
3.4 注解配置	1.4.4
4 示例	1.5
4.1 启动时检查	1.5.1
4.2 集群容错	1.5.2
4.3 负载均衡	1.5.3
4.4 线程模型	1.5.4
4.5 直连提供者	1.5.5
4.6 只订阅	1.5.6
4.7 只注册	1.5.7
4.8 静态服务	1.5.8
4.9 多协议	1.5.9
4.10 多注册中心	1.5.10
4.11 服务分组	1.5.11
4.13 多版本	1.5.12
4.14 分组聚合	1.5.13
4.15 参数验证	1.5.14
4.16 结果缓存	1.5.15

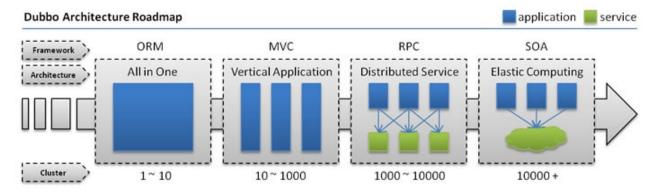
	4.17 泛化引用	1.5.16
	4.18 泛化实现	1.5.17
	4.19 回声测试	1.5.18
	4.20 上下文信息	1.5.19
	4.21 隐式参数	1.5.20
	4.22 异步调用	1.5.21
	4.23 本地调用	1.5.22
	4.24 参数回调	1.5.23
	4.25 事件通知	1.5.24
	4.26 本地存根	1.5.25
	4.27 本地伪装	1.5.26
	4.28 延迟暴露	1.5.27
	4.29 并发控制	1.5.28
	4.30 连接控制	1.5.29
	4.31 延迟连接	1.5.30
	4.32 粘滞连接	1.5.31
	4.33 令牌验证	1.5.32
	4.34 路由规则	1.5.33
	4.35 配置规则	1.5.34
	4.36 服务降级	1.5.35
	4.37 优雅停机	1.5.36
	4.38 主机绑定	1.5.37
	4.39 日志适配	1.5.38
	4.40 访问日志	1.5.39
	4.41 服务容器	1.5.40
	4.42 Reference Config 缓存	1.5.41
	4.43 分布式事务	1.5.42
5 A	PI参考手册	1.6
6 s	chema配置参考手册	1.7
	6.1 dubbo:service	1.7.1
	6.2 dubbo:reference	1.7.2
	6.3 dubbo:protocol	1.7.3
	6.4 dubbo:registry	1.7.4
	6.5 dubbo:monitor	1.7.5

6.6 dubbo:application	1.7.6
6.7 dubbo:module	1.7.7
6.8 dubbo:provider	1.7.8
6.9 dubbo:consumer	1.7.9
6.10 dubbo:method	1.7.10
6.11 dubbo:argument	1.7.11
6.12 dubbo:parameter	1.7.12
7协议参考手册	1.8
7.1 dubbo://	1.8.1
7.2 rmi//	1.8.2
7.3 hessian://	1.8.3
7.4 http://	1.8.4
7.5 webservice://	1.8.5
7.6 thrift://	1.8.6
7.7 memcached://	1.8.7
7.8 redis://	1.8.8
8 注册中心参考手册	1.9
8.1 Multicast注册中心	1.9.1
8.2 Zookeeper注册中心	1.9.2
8.3 Redis注册中心	1.9.3
8.4 Simple注册中心	1.9.4
9 telnet命令参考手册	1.10
10 maven插件参考手册	1.11
11 服务化最佳实践	1.12
12 推荐用法	1.13
13 容量规划	1.14
14 基准测试工具包	1.15
15 性能测试报告	1.16
16 测试覆盖率报告	1.17

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

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"

o 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。

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。

(1) 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。

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

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。

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

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,更多服务治理问题参见:

http://code.alibabatech.com/blog/experience_1402/service-governance-process.html

• 节点角色说明:

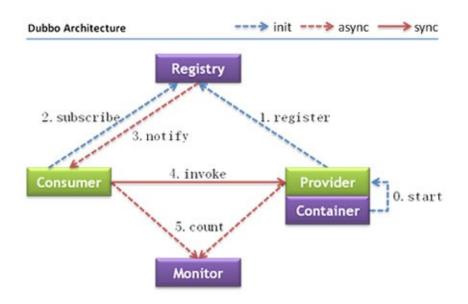
• Provider: 暴露服务的服务提供方

• 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。

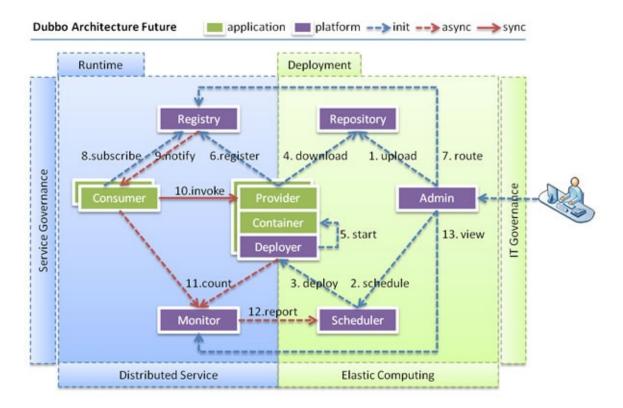
• Monitor: 统计服务的调用次调和调用时间的监控中心。

• Container: 服务运行容器。



- 调用关系说明:
- 服务容器负责启动,加载,运行服务提供者。
- 服务提供者在启动时,向注册中心注册自己提供的服务。
- 服务消费者在启动时,向注册中心订阅自己所需的服务。
- 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。
- 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。
- 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。
- (1) 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,服务提供者名机,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

- (2) 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
- (3) 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
- (4) 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:



- 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。

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,而希望通过API的方式进行调用(不推荐),请参见:API配置

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

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

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

该接口需单独打包,在服务提供方和消费方共享

在服务提供方实现接口

DemoServiceImpl.java

```
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;
    }
}
```

对服务消费方隐藏实现

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"</pre>
            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"///
xsi:schemaLocation="http://www.springframework.org/schema/beans"////
xsi:schemaLocation="http://www.springframework.org/schema/beans"////
xsi:schemaLocation="http://www.springframework.org/schema/beans"////
xsi:schemaLocation="http://www.springframework.org/schema/beans"///
xsi:schemaLocation="http://www.springframework.org/schema/beans"///
xsi:schemaLocation="http://www.springframework.org/schema/beans"//
xsi:schemaLocation="ht
                                                                                                                                                                                                                                           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"</pre>
   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

也可以使用loC注入

必须依赖

JDK1.5+

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。

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日志输出包。
 - o 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。
 - o 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 字节码生成。
 - o 如果或,以及,则不需要。
- spring.jar 配置解析。
 - o 如果用ServiceConfig和ReferenceConfig的API调用,则不需要。
- netty.jar 网络传输。
 - o 如果或,则换成mina.jar或grizzly.jar。
 - o 如果,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: 1.1.7grizzly: 2.1.4httpclient: 4.1.2

hessian_lite: 3.2.1-fixed

xstream: 1.4.1fastjson: 1.1.8zookeeper: 3.3.3

• jedis: 2.0.0

xmemcached: 1.3.6jfreechart: 1.0.13

hessian: 4.0.7jetty: 6.1.26

• hibernate-validator: 4.2.0.Final

zkclient: 0.1curator: 1.1.10

cxf: 2.6.1thrift: 0.8.0

JEE:

• servlet: 2.5

• bsf: 3.1

• validation-api: 1.0.0.GA

• jcache: 0.4

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
		伪造返回结果, 可在失		可用于	

本地伪 装	Stable	败时执行,或直接执 行,用于服务降级	需注册中心 支持	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	Us
Zookeeper注册 中心	Stable	支持基于 有产品 有产品 使用 dubbo- 2.3.3 (推 本用)	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 ANIO 其 其 其 其 其 其 其 其 其 其 其 其 其	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 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	Aliba
		通过不传送 POJO的类元			

Serialization	Tested	信息,在大量 POJO传输 时,性能较好	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	失败自动切现失败自动切现失重强,,重强,身上,重强,是是一个,是一个,是一个,是一个,是一个,是一个,是一个,是一个,是一个,是一	重试会带来更长延迟	可用于生产环境	Aliba
Failfast Cluster	Stable	快速失败,明是一个人,只是一个人,是一个人,是一个人,是一个人,是一个人,是一个人,是一个人,是一个人,	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, 现安全时, 理忽下,通常 用于等操作 日志等操作	调用信息丢失	可用于 生产环境	Moni
Failback Cluster	Tested	失败,后, 自 一 会 以 后 后 , 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。	不可靠,重启丢失	可用于 生产环 境	Regi
		并行调用多个			

Forking Cluster	Tested	服务器,只要 一个成功即可 可,通常用于 实时性要求 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	广提调台错更地 播者,任则常用 所逐意报用方 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。 。	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	最数数跃后慢更少相随相,用数据数数跃后,用,的数数数机器,用,收入,用,收入,,并不够,不够,不够,不够,不够,不够,不够,不够。	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Hash,提的时该求节其不变致sh的到,供原供基,提引相,请同当者本者于平供起同,是是对自求,是引用,是是不够。	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	Aliba
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 进程启动时之 自动给日志分目 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

❷配置项说明

详细配置项,请参见:配置参考手册

❷API使用说明

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

❷配置使用说明

想知道如何使用配置,请参见:tutorials

示例:

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/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"
1" />
    <dubbo:reference id="demoServiceRemote" interface="com.alibaba.dubbo.demo.DemoService" />
    </beans>
```

必所有标签者支持自定义参数,用于不同扩展点实现的特殊配置。

如:

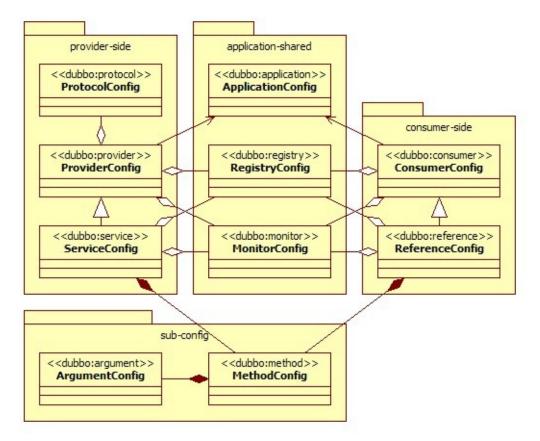
```
<dubbo:protocol name="jms">
     <dubbo:parameter key="queue" value="your_queue" />
</dubbo:protocol>
```

或:(2.1.0开始支持)

▲ 注意声明:xmlns:p="http://www.springframework.org/schema/p"

```
<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 http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd">
        <dubbo:protocol name="jms" p:queue="your_queue" />
        </beans>
```

Configuration Relation:



- 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。
- 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。
- 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。
- 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。
- 模块配置,用于配置当前模块信息,可选。
- 注册中心配置,用于配置连接注册中心相关信息。
- 监控中心配置,用于配置连接监控中心相关信息,可选。
- 提供方的缺省值,当ProtocolConfig和ServiceConfig某属性没有配置时,采用此缺省值,可选。
- 消费方缺省配置,当ReferenceConfig某属性没有配置时,采用此缺省值,可选。
- 方法配置,用于ServiceConfig和ReferenceConfig指定方法级的配置信息。

• 用于指定方法参数配置。

Configuration Override:

•<dubbo:reference interface="com.xxx.XxxService"> <dubbo:method name="findXxx" timeout="1000" /> reference </dubbo:reference> method •<dubbo:service interface="com.xxx.XxxService"> <dubbo:method name="findXxx" timeout="2000" /> service -</dubbo:service> •<dubbo:reference interface="com.xxx.XxxService" timeout="3000" /> reference •<dubbo:service interface="com.xxx.XxxService" timeout="4000" /> service •<dubbo:consumer timeout="5000" /> consumer •<dubbo:provider timeout="6000" /> provider

- 上图中以timeout为例,显示了配置的查找顺序,其它retries, loadbalance, actives等类似。
 - o 方法级优先,接口级次之,全局配置再次之。
 - o 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。

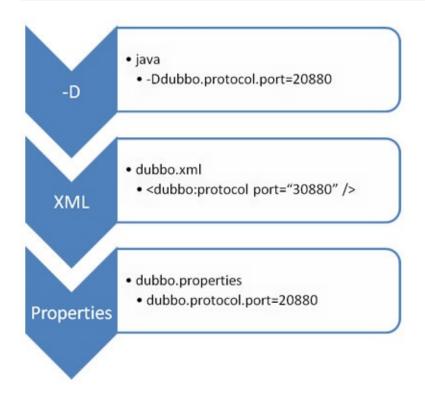
- ☑如果公共配置很简单,没有多注册中心,多协议等情况,或者想多个Spring容器想共享配置,可以使用dubbo.properties作为缺省配置。
- ●Dubbo将自动加载classpath根目录下的dubbo.properties,可以通过JVM启动参数:-Ddubbo.properties.file=xxx.properties 改变缺省配置位置。
- ⚠如果classpath根目录下存在多个dubbo.properties,比如多个jar包中有dubbo.properties,Dubbo会任意加载,并打印Error日志,后续可能改为抛异常。

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:
 - o 比如: dubbo.application.name=foo 等价于 <dubbo:application name="foo" />
 - o 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:
 - o 比如: dubbo.protocol.rmi.port=1234 等价于 <dubbo:protocol id="rmi" name="rmi" port="1099" /> (协议的id没配时,缺省使用协议名作为id)
 - o 比如: dubbo.registry.china.address=10.20.153.10:9090 等价于 <dubbo:registry id="china" address="10.20.153.10:9090" />

典型配置如:

dubbo.properties
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。

▲ API使用范围

API仅用于OpenAPI, ESB, Test, Mock等系统集成,普通服务提供方或消费方,请采用配置方式使用Dubbo,请参见:

❷ API属性含义参考

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

(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();
```

(2). 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
```

(3). 特殊场景

注:下面只列出不同的地方,其它参见上面的写法

(3.1). 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<XxxXService> reference = new ReferenceConfig<XxxXService>(); // 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
```

(3.2). 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");
```

▲ 2.2.1以上版本支持

服务提供方注解:

服务提供方配置:

```
<!-- 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;
}
```

服务消费方配置:

```
<!-- 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" />
```

也可以使用:(等价于前面的:)

```
<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>
```

▲ Spring2.5及以后版本支持component-scan,如果用的是Spring2.0及以前版本,需配置:

```
<!-- 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" />
```

●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"。

▲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。

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。

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" />
```

也可以用dubbo.properties配置

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

也可以用-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
```

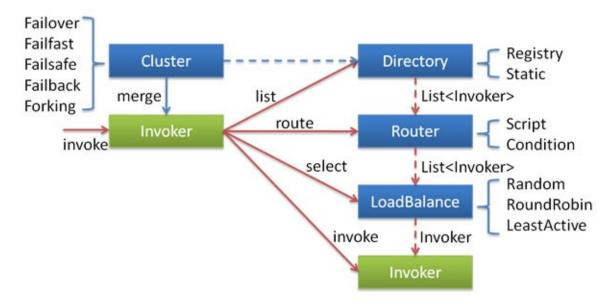
▲注意区别

- 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。

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 interface="com.foo.BarService" init="true" />

❷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,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。(缺省)
- 通常用于读操作,但重试会带来更长延迟。
- 可通过 retries="2" 来设置重试次数(不含第一次)。

Failfast Cluster

- 快速失败,只发起一次调用,失败立即报错。
- 通常用于非幂等性的写操作,比如新增记录。

Failsafe Cluster

- 失败安全,出现异常时,直接忽略。
- 通常用于写入审计日志等操作。

Failback Cluster

- 失败自动恢复,后台记录失败请求,定时重发。
- 通常用于消息通知操作。

Forking Cluster

- 并行调用多个服务器,只要一个成功即返回。
- 通常用于实时性要求较高的读操作,但需要浪费更多服务资源。
- 可通过 forks="2" 来设置最大并行数。

Broadcast Cluster

- 广播调用所有提供者,逐个调用,任意一台报错则报错。(2.1.0开始支持)
- 通常用于通知所有提供者更新缓存或日志等本地资源信息。

重试次数配置如: (failover集群模式生效)

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

或

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

或

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

集群模式配置如:

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

或

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

◆ 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

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" />

配置如:

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

或

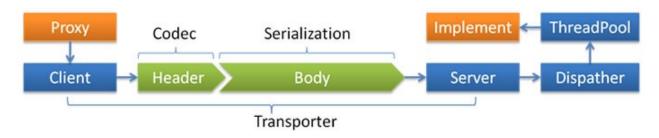
```
<dubbo:reference interface="..." loadbalance="roundrobin" />
```

或

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



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



●事件处理线程说明

- 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|O请求,比如需要查询数据库,则必 须派发到线程池,否则|O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。

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。(为避免收缩时突然来了大流量引起的性能问题)。

配置如下:

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

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。



• 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:

1.0.6及以上版本支持

<dubbo:reference id="xxxService" interface="com.alibaba.xxx.XxxService" url="dub
bo://localhost:20890" />

• 在JVM启动参数中加入-D参数映射服务地址,如:

kev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

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

• 如果服务比较多,也可以用文件映射,如:

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,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

java -Ddubbo.resolve.file=xxx.properties

然后在映射文件xxx.properties中加入:(key为服务名,value为服务提供者url)

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

❷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。

▲问题

如果有两个镜像环境,两个注册中心,有一个服务只在其中一个注册中心有部署,另一个注册中心还没来得及部署,而两个注册中心的其它应用都需要依赖此服务,所以需要将服务同时注册到两个注册中心,但却不能让此服务同时依赖两个注册中心的其它服务。

②解决方案

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

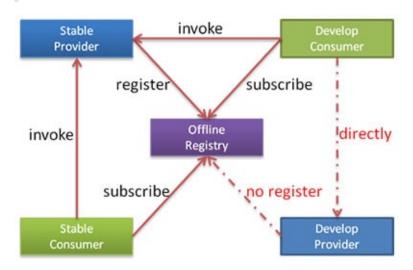
```
<dubbo:registry id="hzRegistry" address="10.20.153.10:9090" />
<dubbo:registry id="qdRegistry" address="10.20.141.150:9090?subscribe=false" />
```

▲问题

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。

②解决方案

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" />

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等,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:

通常由脚本监控中心页面等调用

```
RegistryFactory registryFactory = ExtensionLoader.getExtensionLoader(RegistryFactory.c
lass).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=pr
oviders&dynamic=false&application=foo"));
```

可以自行扩展协议,参见:协议扩展

(1) 不同服务不同协议

比如: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

consumer.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"</pre>
   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</pre>
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=</pre>
"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="</pre>
demoService" protocol="rmi" />
</beans>
```

(2) 多协议暴露服务

比如: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"</pre>
   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</pre>
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</pre>
rsion="1.0.0" protocol="dubbo, hessian" />
</beans>
```

可以自行扩展注册中心,参见:注册中心扩展

(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。

consumer.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"</pre>
   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"</pre>
/>
   <!-- 向多个注册中心注册 -->
   <dubbo:service interface="com.alibaba.hello.api.HelloService" version="1.0.0" ref=</pre>
"helloService" registry="hangzhouRegistry, qingdaoRegistry" />
</beans>
```

(2) 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"</pre>
   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=</pre>
"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="</pre>
demoService" registry="intlRegistry" />
</beans>
```

(3) 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"</pre>
   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</pre>
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</pre>
ce" version="1.0.0" registry="intlRegistry" />
</heans>
```

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/dubbo/dubbo.xsd">
    </dubbo:application name="world" />
    <!-- 多注册中心配置,坚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>
```

❷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.IndexService" />
```

任意组:

2.2.0以上版本支持,总是只调一个可用组的实现

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

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

老版本服务

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

新版本服务

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

引用老版本

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

引用新版本

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

不区分版本:(2.2.0以上版本支持)

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

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

▲从2.1.0版本开始支持,

代码参见:https://github.com/alibaba/dubbo/tree/master/dubbo-test/dubbo-test-examples/src/main/java/com/alibaba/dubbo/examples/merge

配置如:

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 参见:合并结果扩展

```
<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>
```

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

▲2.1.0以上版本支持, 完整示例代码参

见:https://github.com/alibaba/dubbo/tree/master/dubbo-test/dubbo-test-examples/src/main/java/com/alibaba/dubbo/examples/validation

验证方式可扩展,参见:验证扩展

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^{-1}}[-1]^{-1}) = [a-zA-Z0-9]+(?:[-.][a-zA-Z0-9]+)^{-1}
\\.[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
}
```

在客户端验证参数

```
<dubbo:reference id="validationService" interface="com.alibaba.dubbo.examples.validati
on.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);
       }
   }
}
```

需要加入依赖

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

▲2.1.0以上版本支持

示例代码:https://github.com/alibaba/dubbo/tree/master/dubbo-test/dubbo-test-examples/src/main/java/com/alibaba/dubbo/examples/cache

- 1ru 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: CacheFactory扩展点

配置如:

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

或:

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

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

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

```
import com.alibaba.dubbo.rpc.service.GenericService;
. . .
// 引用远程服务
ReferenceConfig<GenericService> reference = new ReferenceConfig<GenericService>(); //
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
reference.setInterface("com.xxx.XxxService"); // 弱类型接口名
reference.setVersion("1.0.0");
reference.setGeneric(true); // 声明为泛化接口
GenericService genericService = reference.get(); // 用com.alibaba.dubbo.rpc.service.Gen
ericService可以替代所有接口引用
// 基本类型以及Date, List, Map等不需要转换,直接调用
Object result = genericService.$invoke("sayHello", new String[] {"java.lang.String"},
new Object[] {"world"});
// 用Map表示POJO参数,如果返回值为POJO也将自动转成Map
Map<String, Object> person = new HashMap<String, Object>();
person.put("name", "xxx");
person.put("password", "yyy");
Object result = genericService.$invoke("findPerson", new String[]{"com.xxx.Person"}, n
ew Object[]{person}); // 如果返回POJO将自动转成Map
```

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>();
map.put("class", "com.xxx.PersonImpl"); // 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("name", "xxx");
map.put("password", "yyy");
```

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

```
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];
        }
    }
}
```

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

ServiceConfig<GenericService> service = new ServiceConfig<GenericService>(); // 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 service.setInterface("com.xxx.XxxService"); // 弱类型接口名 service.setVersion("1.0.0"); service.setRef(xxxService); // 指向一个通用服务实现

// 暴露及注册服务 service.export();
```

- ●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。

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

```
MemberService memberService = ctx.getBean("memberService"); // 远程服务引用
EchoService echoService = (EchoService) memberService; // 强制转型为EchoService
String status = echoService.$echo("OK"); // 回声测试可用性
assert(status.equals("OK"));
```

- **◎上下文中存放的是当前调用过程中所需的环境信息。**
- ❷所有配置信息都将转换为URL的参数,参见配置项一览表中的"对应URL参数"一列。

注意 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。

(1) 服务消费方

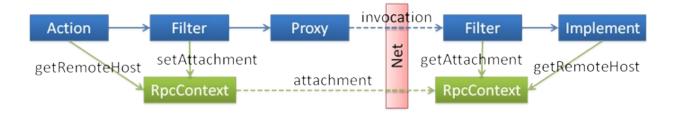
```
xxxService.xxx(); // 远程调用
boolean isConsumerSide = RpcContext.getContext().isConsumerSide(); // 本端是否为消费端,
这里会返回true
String serverIP = RpcContext.getContext().getRemoteHost(); // 获取最后一次调用的提供方IP地址
String application = RpcContext.getContext().getUrl().getParameter("application"); //
获取当前服务配置信息,所有配置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

(2) 服务提供方

```
public class XxxServiceImpl implements XxxService {

public void xxx() { // 服务方法实现
    boolean isProviderSide = RpcContext.getContext().isProviderSide(); // 本端是否为
提供端,这里会返回true
    String clientIP = RpcContext.getContext().getUrl().getParameter("application"); // 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
    // ...
    yyyService.yyy(); // 注意:每发起RPC调用,上下文状态会变化
    boolean isProviderSide = RpcContext.getContext().isProviderSide(); // 此时本端变
成消费端,这里会返回false
    // ...
}
```

▲ 注:path,group,version,dubbo,token,timeout几个key有特殊处理,请使用其它key 值。



(1) 服务消费方

```
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 // ...
```

注: setAttachment设置的KV,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。

(2) 服务提供方

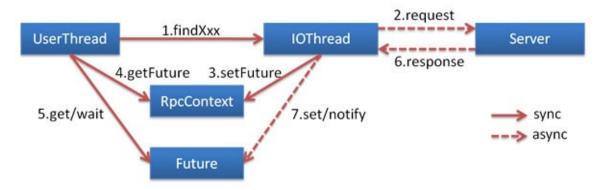
```
public class XxxServiceImpl implements XxxService {

public void xxx() { // 服务方法实现

String index = RpcContext.getContext().getAttachment("index"); // 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

// ...
}
```

▲2.0.6及其以上版本支持



配置声明:

consumer.xml

调用代码:

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

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

// 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。

Foo foo = fooFuture.get(); // 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。
Bar bar = barFuture.get(); // 同理等待bar返回。

// 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:(异步总是不等待返回)

• 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" />
```

▽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。

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

```
<dubbo:reference ... scope="remote" />
```

●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。

▲2.0.6及其以上版本支持

代码参见:https://github.com/alibaba/dubbo/tree/master/dubbo-test/dubbo-test-examples/src/main/java/com/alibaba/dubbo/examples/callback

(1) 共享服务接口

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);
}
```

(2) 服务提供者

服务提供者接口实现示例

CallbackServiceImpl.java

```
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</pre>
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

(3) 服务消费者

服务消费者配置示例

consumer.xml

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

服务消费者调用示例

CallbackServiceTest.java

```
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);
    }
});
```

②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。

▲ 支持版本: 2.0.7之后

(1) 服务提供者与消费者共享服务接口

IDemoService.java

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

(2) 服务提供者实现

DemoServiceImpl.java

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

(3) 服务提供者配置

provider.xml

```
<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"/>
```

(4) 服务消费者Callback接口及实现

Notify.java

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

NotifyImpl.java

(5) 服务消费者Callback接口及实现

consumer.xml

⊘callback与async功能正交分解:

- async=true,表示结果是否马上返回.
- onreturn 表示是否需要回调.

组合情况:(async=false 默认)

- 异步回调模式:async=true onreturn="xxx"
- 同步回调模式:async=false onreturn="xxx"
- 异步无回调:async=true
- 同步无回调:async=false

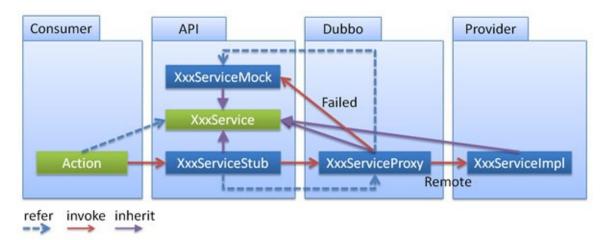
(6) TEST CASE

Test.java

```
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());
```

●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,然后把Stub暴露组给用户,Stub可以决定要不要去调Proxy。

△Stub必须有可传入Proxy的构造函数。



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

或

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

api.jar:

com.foo.BarService

com.foo.BarServiceStub // 在API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

```
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
          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";
       }
   }
}
```

- ☑Mock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。
- ☑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。

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

或

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

api.jar:

```
com.foo.BarService
com.foo.BarServiceMock // 在API旁边放一个Mock实现,它实现BarService接口,并有一个无参构造函数
```

```
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" />
```

☑如果你的服务需要warmup时间,比如初始化缓存,等待相关资源就位等,可以使用
delav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:(基于Spring的ContextRefreshedEvent事件触发暴露)

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

△Spring2.x初始化死锁问题

!在Spring解析到时,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。

```
org.springframework.beans.factory.support.DefaultListableBeanFactory.getBeanDefinitionNames(DefaultListableBeanFactory waiting to lock <0x0000000079545cb40> (a java.util.concurrent.ConcurrentHashMap)
org.springframework.beans.factory.support.DefaultListableBeanFactory.getBeanNamesForType(DefaultListableBeanFactory.javarg.springframework.beans.factory.support.DefaultListableBeanFactory.findAutowireCandidates(DefaultListableBeanFactory.org.springframework.beans.factory.support.DefaultListableBeanFactory.findAutowireCandidates(DefaultListableBeanFactory.org.springframework.beans.factory.support.DefaultListableBeanFactory.findAutowireCandidates(DefaultListableBeanFactory.grspringframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessorsAutowiredFieldElement.inject(Autowir.org.springframework.beans.factory.annotation.InjectionMetadata.injectFields(InjectionMetadata.java:105)
org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor.postProcessAfterInstantiation(Autowirog.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(AbstractAutowireCapableBeanForg.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.doCreateBean(AbstractAutowireCapableBeanForg.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.createBean(AbstractAutowireCapableBeanFactory.java.security.AccessController.doPrivileged(Native Method)
org.springframework.beans.factory.support.BeanDefinitionValueResolver.resolveInnerBean(BeanDefinitionValueResolver.jav.org.springframework.beans.factory.support.BeanDefinitionValueResolver.resolveAutoMireCapableBeanFinitionValueResolver.jav.org.springframework.beans.factory.support.BeanDefinitionValueResolver.resolveValueIfNecessary(BeanDefinitionValueResolver.jav.org.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.populateBean(AbstractAutowireCapableBeanForg.springframework.beans.factory.support.AbstractAutowireCapableBeanFactory.jopulateBean(AbstractAutowireCapableBeanForg.spring
```

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。

```
org.springframework.beans.factory.support.DefaultSingletonBeanRegistry.getSingleton(DefaultSingletonBeanRegistry.getSingleton(DefaultSingletonBeanRegistry.getSingleton(DefaultSingletonBeanRegistry.getSingleton(DefaultSingletonBeanRegistry))

org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:185)

org.springframework.beans.factory.support.AbstractBeanFactory.getBean(AbstractBeanFactory.java:184)

org.springframework.beans.factory.support.AbstractBeanFactory.petBean(AbstractBeanFactory.java:164)

org.springframework.beans.factory.support.DefaultListableBeanFactory.preInstantiateSingletons(DefaultListable locked <0x0000000079545cb48> (a java.util.concurrent.ConcurrentHashMap)

org.springframework.context.support.AbstractApplicationContext.finishBeanFactoryInitialization(AbstractApplicatorg.springframework.context.support.AbstractApplicationContext.refresh(AbstractApplicationContext.java:380)
```

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。

❷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 loC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。

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" />
```

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>
```

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" />
```

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" />



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

限制服务器端接受的连接不能超过10个: (因为连接在Server上,所以配置在Provider上)

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

限制客户端服务使用连接连接数:(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)

```
<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

- ●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。
- ⚠只对使用长连接的dubbo协议生效。

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

- ◆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。
- ❷粘滞连接将自动开启延迟连接,以减少长连接数,参见:延迟连接

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



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

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" />
<!--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" />
<!--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" />
```

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

▲2.2.0以上版本支持.

❷路由规则扩展点:路由扩展

向注册中心写入路由规则:(通常由监控中心或治理中心的页面完成)

 $\label{eq:registryFactory} \textit{RegistryFactory} = \textit{ExtensionLoader.getExtensionLoader}(\textit{RegistryFactory.c} \\ \textit{lass}).\textit{getAdaptiveExtension()};$

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("http://10.20.160.198/wiki/display/dubbo/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:
 - o 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.*
 - o 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: 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)

```
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

●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。

△脚本没有沙箱约束,可执行任意代码,存在后门风险

```
"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 ("http://10.20.160.198/wiki/display/dubbo/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</pre>
```

▲2.2.0 以上版本支持

向注册中心写入动态配置覆盖规则:(通常由监控中心或治理中心的页面完成)

RegistryFactory registryFactory = ExtensionLoader.getExtensionLoader(RegistryFactory.c
lass).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=configur
ators&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:

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. Bar Service? 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. Bar Service? category = configurators & dynamic = false & load balance = least active

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. Bar Service? category = configurators & dynamic = false & application = foo & mock = force: return + null

▲2.2.0以上版本支持,

❷参见:配置规则

向注册中心写入动态配置覆盖规则:(通过由监控中心或治理中心的页面完成)

RegistryFactory registryFactory = ExtensionLoader.getExtensionLoader(RegistryFactory.c lass).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=configur ators&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。
- 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。
- 容忍不重要服务不稳定时对调用方的影响。

▲ 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。

设置优雅停机超时时间,缺省超时时间是10秒:(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();

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。

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="http://10.20.160.198/wiki/display/dubbo/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
```

缺省主机端口与协议相关:

• dubbo: 20880

• rmi: 1099

• http: 80

• hessian: 80

• webservice: 80

• memcached: 11211

• redis: 6379

主机端口配置:

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
```

▲2.2.1以上版本支持

❷扩展点:日志适配扩展

缺省自动查找:

- log4j
- slf4j
- jcl
- jdk

可以通过以下方式配置日志输出策略

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

dubbo.properties

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

dubbo.xml

<dubbo:application logger="log4j" />

❷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。

▲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。
- ❷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。
- ❷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Container扩展点进行扩展,参见:Container。配置配在java命令-D参数或者dubbo.properties中

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
 - o 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loq4j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
 - o 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

如:(缺省只加载spring)

java com.alibaba.dubbo.container.Main

或:(通过main函数参数传入要加载的容器)

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

或:(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

或:(通过classpath下的dubbo.properties配置传入要加载的容器)

dubbo.properties

dubbo.container=spring,jetty,log4j

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。API方式编程时,容易忽略此问题。

Dubbo 2.4.0+版本,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();
XxxService xxxService = cache.get(reference); // cache.get方法中会Cache Reference对象,并且调用ReferenceConfig.get方法启动ReferenceConfig
// 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,销毁ReferenceConfig并释放对应的资源。

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

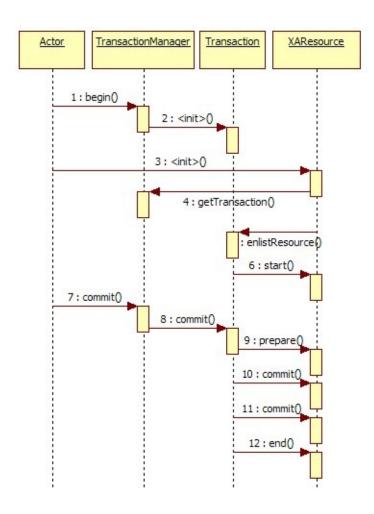
缺省ReferenceConfigCache把相同服务Group、接口、版本的ReferenceConfig认为是相同, 缓存一份。即以服务Group、接口、版本为缓存的Key。

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 );
```

- ❷基于JTA/XA规范实现。
- ▲暂未实现。

两阶段提交:



- ❷ 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。
- ⚠ 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。

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

 $\verb|com.alibaba.dubbo.rpc.RpcContext| \\$

详细参见:上下文信息&对方地址&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

这里以Xml Config为准,列举所有配置项,其它配置方式,请参见相应转换关	
系:Properties Config,Annotation Config,API Config	
注意:只有group, interface, version是服务的匹配条件, 三者决定是不是同	_
个服务,其它配置项均为调优和治理参数。	

所有配置项分为三大类,参见下表中的"作用"一列。

- 服务发现:表示该配置项用于服务的注册与发现,目的是让消费方找到提供方。
- 服务治理:表示该配置项用于治理服务间的关系,或为开发测试提供便利条件。
- 性能调优:表示该配置项用于调优性能,不同的选项对性能会产生影响。
- 所有配置最终都将转换为URL表示,并由服务提供方生成,经注册中心传递给消费方,各属性对应URL的参数,参见配置项一览表中的"对应URL参数"列。

URL格式:

protocol://username:password@host:port/path?key=value&key=value

Schema: http://code.alibabatech.com/schema/dubbo/dubbo.xsd

服务提供者暴露服务配置:

配置类:com.alibaba.dubbo.config.ServiceConfig

标签	属性	对应URL参数	类型	是否必填	缺
<dubbo:service></dubbo:service>	interface		class	必塡	
<dubbo:service></dubbo:service>	ref		object	必塡	
<dubbo:service></dubbo:service>	version	version	string	可选	0.0
<dubbo:service></dubbo:service>	group	group	string	可选	
<dubbo:service></dubbo:service>	path	<path></path>	string	可选	缺 ² 接 ¹
<dubbo:service></dubbo:service>	delay	delay	int	可选	0
<dubbo:service></dubbo:service>	timeout	timeout	int	可选	10(
<dubbo:service></dubbo:service>	retries	retries	int	可选	2
<dubbo:service></dubbo:service>	connections	connections	int	可选	100
<dubbo:service></dubbo:service>	loadbalance	loadbalance	string	可选	ran
<dubbo:service></dubbo:service>	async	async	boolean	可选	fals
<dubbo:service></dubbo:service>	stub	stub	class/boolean	可选	fals

<dubbo:service></dubbo:service>	mock	mock	class/boolean	可选	fals
<dubbo:service></dubbo:service>	token	token	string/boolean	可选	fals
<dubbo:service></dubbo:service>	registry		string	可选	缺; 所; reg
<dubbo:service></dubbo:service>	provider		string	可选	缺f 第- pro 配i
<dubbo:service></dubbo:service>	deprecated	deprecated	boolean	可选	fals
<dubbo:service></dubbo:service>	dynamic	dynamic	boolean	可选	true
<dubbo:service></dubbo:service>	accesslog	accesslog	string/boolean	可选	fals
<dubbo:service></dubbo:service>	owner	owner	string	可选	
<dubbo:service></dubbo:service>	document	document	string	可选	
<dubbo:service></dubbo:service>	weight	weight	int	可选	

<dubbo:service></dubbo:service>	executes	executes	int	可选	0
<dubbo:service></dubbo:service>	proxy	proxy	string	可选	jav
<dubbo:service></dubbo:service>	cluster	cluster	string	可选	fail
<dubbo:service></dubbo:service>	filter	service.filter	string	可选	def
<dubbo:service></dubbo:service>	listener	exporter.listener	string	可选	def
<dubbo:service></dubbo:service>	protocol		string	可选	
<dubbo:service></dubbo:service>	layer	layer	string	可选	
<dubbo:service></dubbo:service>	register	register	boolean	可选	true

服务消费者引用服务配置:

配置类:com.alibaba.dubbo.config.ReferenceConfig

标签	属性	对应URL参数	类型	是否必填	
<dubbo:reference></dubbo:reference>	id		string	必塡	
<dubbo:reference></dubbo:reference>	interface		class	必塡	
<dubbo:reference></dubbo:reference>	version	version	string	可选	
<dubbo:reference></dubbo:reference>	group	group	string	可选	
<dubbo:reference></dubbo:reference>	timeout	timeout	long	可选	缺~的
<dubbo:reference></dubbo:reference>	retries	retries	int	可选	缺 Vi 的
<dubbo:reference></dubbo:reference>	connections	connections	int	可选	缺 < 的
<dubbo:reference></dubbo:reference>	loadbalance	loadbalance	string	可选	缺 V 的
<dubbo:reference></dubbo:reference>	async	async	boolean	可选	缺 Vi Ki
<dubbo:reference></dubbo:reference>	generic	generic	boolean	可选	缺 Vi 的
<dubbo:reference></dubbo:reference>	check	check	boolean	可选	缺 Vi Ki
<dubbo:reference></dubbo:reference>	url	url	string	可选	
<dubbo:reference></dubbo:reference>	stub	stub	class/boolean	可选	

<dubbo:reference></dubbo:reference>	mock	mock	class/boolean	可选	
<dubbo:reference></dubbo:reference>	cache	cache	string/boolean	可选	
<dubbo:reference></dubbo:reference>	validation	validation	boolean	可选	
<dubbo:reference></dubbo:reference>	proxy	proxy	boolean	可选	ja
<dubbo:reference></dubbo:reference>	client	client	string	可选	
<dubbo:reference></dubbo:reference>	registry		string	可选	
<dubbo:reference></dubbo:reference>	owner	owner	string	可选	
<dubbo:reference></dubbo:reference>	actives	actives	int	可选	0
<dubbo:reference></dubbo:reference>	cluster	cluster	string	可选	fa
<dubbo:reference></dubbo:reference>	filter	reference.filter	string	可选	dı
<dubbo:reference></dubbo:reference>	listener	invoker.listener	string	可选	de
<dubbo:reference></dubbo:reference>	layer	layer	string	可选	
<dubbo:reference></dubbo:reference>	init	init	boolean	可选	fa
<dubbo:reference></dubbo:reference>	protocol	protocol	string	可选	

服务提供者协议配置:

配置类:com.alibaba.dubbo.config.ProtocolConfig

说明:如果需要支持多协议,可以声明多个<dubbo:protocol>标签,并在<dubbo:service>中通过protocol属性指定使用的协议。

标签	属性	对应URL参数	类型	是否必填	
<dubbo:protocol></dubbo:protocol>	id		string	可选	dubb
<dubbo:protocol></dubbo:protocol>	name	<pre><pre><pre><pre><pre><pre><pre><pre></pre></pre></pre></pre></pre></pre></pre></pre>	string	必填	dubb
<dubbo:protocol></dubbo:protocol>	port	<port></port>	int	可选	du端m口和缺如或 po一的 2端端长可的口协为 he省果者 rt 个端 .0口口,控
<dubbo:protocol></dubbo:protocol>	host	<host></host>	string	可选	自动
<dubbo:protocol></dubbo:protocol>	threadpool	threadpool	string	可选	fixed
<dubbo:protocol></dubbo:protocol>	threads	threads	int	可选	100
<dubbo:protocol></dubbo:protocol>	iothreads	threads	int	可选	cpu/
<dubbo:protocol></dubbo:protocol>	accepts	accepts	int	可选	0

<dubbo:protocol></dubbo:protocol>	payload	payload	int	可选	8838
<dubbo:protocol></dubbo:protocol>	codec	codec	string	可选	dubb
<dubbo:protocol></dubbo:protocol>	serialization	serialization	string	可选	dubb 为he rmi协 java 缺省
<dubbo:protocol></dubbo:protocol>	accesslog	accesslog	string/boolean	可选	
<dubbo:protocol></dubbo:protocol>	path	<path></path>	string	可选	
<dubbo:protocol></dubbo:protocol>	transporter	transporter	string	可选	dubb 为ne
<dubbo:protocol></dubbo:protocol>	server	server	string	可选	dubb 为ne 议缺
<dubbo:protocol></dubbo:protocol>	client	client	string	可选	dubb 为ne
<dubbo:protocol></dubbo:protocol>	dispatcher	dispatcher	string	可选	dubb 为all
<dubbo:protocol></dubbo:protocol>	queues	queues	int	可选	0
<dubbo:protocol></dubbo:protocol>	charset	charset	string	可选	UTF-
<dubbo:protocol></dubbo:protocol>	buffer	buffer	int	可选	8192
<dubbo:protocol></dubbo:protocol>	heartbeat	heartbeat	int	可选	0

<dubbo:protocol></dubbo:protocol>	telnet	telnet	string	可选	
<dubbo:protocol></dubbo:protocol>	register	register	boolean	可选	true
<dubbo:protocol></dubbo:protocol>	contextpath	contextpath	String	可选	缺省

注册中心配置:

配置类:com.alibaba.dubbo.config.RegistryConfig

说明:如果有多个不同的注册中心,可以声明多个<dubbo:registry>标签,并在<dubbo:service>或<dubbo:reference>的registry属性指定使用的注册中心。

标签	属性	对应URL参数	类型	是否必填	缺省值
<dubbo:registry></dubbo:registry>	id		string	可选	
<dubbo:registry></dubbo:registry>	address	<host:port></host:port>	string	必塡	
<dubbo:registry></dubbo:registry>	protocol	<pre><pre><pre><pre><pre><pre><pre><pre></pre></pre></pre></pre></pre></pre></pre></pre>	string	可选	dubbo
<dubbo:registry></dubbo:registry>	port	<port></port>	int	可选	9090
<dubbo:registry></dubbo:registry>	username	<username></username>	string	可选	
<dubbo:registry></dubbo:registry>	password	<password></password>	string	可选	
<dubbo:registry></dubbo:registry>	transport	registry.transporter	string	可选	netty

<dubbo:registry></dubbo:registry>	timeout	registry.timeout	int	可选	5000
<dubbo:registry></dubbo:registry>	session	registry.session	int	可选	60000
<dubbo:registry></dubbo:registry>	file	registry.file	string	可选	
<dubbo:registry></dubbo:registry>	wait	registry.wait	int	可选	0
<dubbo:registry></dubbo:registry>	check	check	boolean	可选	true
<dubbo:registry></dubbo:registry>	register	register	boolean	可选	true
<dubbo:registry></dubbo:registry>	subscribe	subscribe	boolean	可选	true
<dubbo:registry></dubbo:registry>	dynamic	dynamic	boolean	可选	true

监控中心配置:

配置类:com.alibaba.dubbo.config.MonitorConfig

标签	属性	对应URL参数	类型	是否必填	缺省值	作用
<dubbo:monitor></dubbo:monitor>	protocol	protocol	string	可选	dubbo	服务治理
<dubbo:monitor></dubbo:monitor>	address	<url></url>	string	可选	N/A	服务治理

应用信息配置:

配置类:com.alibaba.dubbo.config.ApplicationConfig

标签	属性	对应 URL 参数	类型	是否必填	缺省
<dubbo:application></dubbo:application>	name	application	string	必塡	
<dubbo:application></dubbo:application>	version	application.version	string	可选	
<dubbo:application></dubbo:application>	owner	owner	string	可选	
<dubbo:application></dubbo:application>	organization	organization	string	可选	
<dubbo:application></dubbo:application>	architecture	architecture	string	可选	
<dubbo:application></dubbo:application>	environment	environment	string	可选	
<dubbo:application></dubbo:application>	compiler	compiler	string	可选	javas
<dubbo:application></dubbo:application>	logger	logger	string	可选	slf4j

模块信息配置: 配置类:com.alibaba.dubbo.config.ModuleConfig

标签	属性	对应URL参数	类型	是否必填	缺省值	1
<dubbo:module></dubbo:module>	name	module	string	必塡		那
<dubbo:module></dubbo:module>	version	module.version	string	可选		那
<dubbo:module></dubbo:module>	owner	owner	string	可选		那
<dubbo:module></dubbo:module>	organization	organization	string	可选		用注

服务提供者缺省值配置:

配置类:com.alibaba.dubbo.config.ProviderConfig

说明:该标签为<dubbo:service>和<dubbo:protocol>标签的缺省值设置。

标签	属性	对应URL参数	类型	是否必填
<dubbo:provider></dubbo:provider>	id		string	可选
<dubbo:provider></dubbo:provider>	protocol	<pre><pre><pre><pre><pre><pre><pre><pre></pre></pre></pre></pre></pre></pre></pre></pre>	string	可选
<dubbo:provider></dubbo:provider>	host	<host></host>	string	可选
<dubbo:provider></dubbo:provider>	threads	threads	int	可选
<dubbo:provider></dubbo:provider>	payload	payload	int	可选
<dubbo:provider></dubbo:provider>	path	<path></path>	string	可选
<dubbo:provider></dubbo:provider>	server	server	string	可选
<dubbo:provider></dubbo:provider>	client	client	string	可选
<dubbo:provider></dubbo:provider>	codec	codec	string	可选
<dubbo:provider></dubbo:provider>	serialization	serialization	string	可选
<dubbo:provider></dubbo:provider>	default		boolean	可选
<dubbo:provider></dubbo:provider>	filter	service.filter	string	可选

<dubbo:provider></dubbo:provider>	listener	exporter.listener	string	可选
<dubbo:provider></dubbo:provider>	threadpool	threadpool	string	可选
<dubbo:provider></dubbo:provider>	accepts	accepts	int	可选
<dubbo:provider></dubbo:provider>	version	version	string	可选
<dubbo:provider></dubbo:provider>	group	group	string	可选
<dubbo:provider></dubbo:provider>	delay	delay	int	可选
<dubbo:provider></dubbo:provider>	timeout	default.timeout	int	可选
<dubbo:provider></dubbo:provider>	retries	default.retries	int	可选
<dubbo:provider></dubbo:provider>	connections	default.connections	int	可选
<dubbo:provider></dubbo:provider>	loadbalance	default.loadbalance	string	可选
<dubbo:provider></dubbo:provider>	async	default.async	boolean	可选
<dubbo:provider></dubbo:provider>	stub	stub	boolean	可选
<dubbo:provider></dubbo:provider>	mock	mock	boolean	可选
<dubbo:provider></dubbo:provider>	token	token	boolean	可选

<dubbo:provider></dubbo:provider>	registry	registry	string	可选
<dubbo:provider></dubbo:provider>	dynamic	dynamic	boolean	可选
<dubbo:provider></dubbo:provider>	accesslog	accesslog	string/boolean	可选
<dubbo:provider></dubbo:provider>	owner	owner	string	可选
<dubbo:provider></dubbo:provider>	document	document	string	可选
<dubbo:provider></dubbo:provider>	weight	weight	int	可选
<dubbo:provider></dubbo:provider>	executes	executes	int	可选
<dubbo:provider></dubbo:provider>	actives	default.actives	int	可选
<dubbo:provider></dubbo:provider>	proxy	proxy	string	可选
<dubbo:provider></dubbo:provider>	cluster	default.cluster	string	可选
<dubbo:provider></dubbo:provider>	deprecated	deprecated	boolean	可选
<dubbo:provider></dubbo:provider>	queues	queues	int	可选
<dubbo:provider></dubbo:provider>	charset	charset	string	可选

<dubbo:provider></dubbo:provider>	buffer	buffer	int	可选
<dubbo:provider></dubbo:provider>	iothreads	iothreads	int	可选
<dubbo:provider></dubbo:provider>	telnet	telnet	string	可选
<dubbo:service></dubbo:service>	contextpath	contextpath	String	可选
<dubbo:provider></dubbo:provider>	layer	layer	string	可选

服务消费者缺省值配置:

配置类:com.alibaba.dubbo.config.ConsumerConfig 说明:该标签为<dubbo:reference>标签的缺省值设置。

标签	属性	对应URL参数	类型	是否必填
<dubbo:consumer></dubbo:consumer>	timeout	default.timeout	int	可选
<dubbo:consumer></dubbo:consumer>	retries	default.retries	int	可选
<dubbo:consumer></dubbo:consumer>	loadbalance	default.loadbalance	string	可选
<dubbo:consumer></dubbo:consumer>	async	default.async	boolean	可选
<dubbo:consumer></dubbo:consumer>	connections	default.connections	int	可选
<dubbo:consumer></dubbo:consumer>	generic	generic	boolean	可选
<dubbo:consumer></dubbo:consumer>	check	check	boolean	可选
<dubbo:consumer></dubbo:consumer>	proxy	proxy	string	可选
<dubbo:consumer></dubbo:consumer>	owner	owner	string	可选
<dubbo:consumer></dubbo:consumer>	actives	default.actives	int	可选
<dubbo:consumer></dubbo:consumer>	cluster	default.cluster	string	可选
<dubbo:consumer></dubbo:consumer>	filter	reference.filter	string	可选
<dubbo:consumer></dubbo:consumer>	listener	invoker.listener	string	可选
<dubbo:consumer></dubbo:consumer>	registry		string	可选
<dubbo:consumer></dubbo:consumer>	layer	layer	string	可选

<dubbo:consumer></dubbo:consumer>	init	init	boolean	可选
<dubbo:consumer></dubbo:consumer>	cache	cache	string/boolean	可选
<dubbo:consumer></dubbo:consumer>	validation	validation	boolean	可选

方法级配置:

配置类:com.alibaba.dubbo.config.MethodConfig

说明:该标签为<dubbo:service>或<dubbo:reference>的子标签,用于控制到方法级,

标签	属性	对应URL参数	类型	是不
<dubbo:method></dubbo:method>	name		string	必均
<dubbo:method></dubbo:method>	timeout	<metodname>.timeout</metodname>	int	可並
<dubbo:method></dubbo:method>	retries	<metodname>.retries</metodname>	int	可设
<dubbo:method></dubbo:method>	loadbalance	<metodname>.loadbalance</metodname>	string	可並
<dubbo:method></dubbo:method>	async	<metodname>.async</metodname>	boolean	可並
<dubbo:method></dubbo:method>	sent	<methodname>.sent</methodname>	boolean	可追
<dubbo:method></dubbo:method>	actives	<metodname>.actives</metodname>	int	可並
<dubbo:method></dubbo:method>	executes	<metodname>.executes</metodname>	int	可並
<dubbo:method></dubbo:method>	deprecated	<methodname>.deprecated</methodname>	boolean	可並
<dubbo:method></dubbo:method>	sticky	<methodname>.sticky</methodname>	boolean	可並
<dubbo:method></dubbo:method>	return	<methodname>.return</methodname>	boolean	可兹
<dubbo:method></dubbo:method>	oninvoke	attribute属性,不在URL中 体现	String	可追
<dubbo:method></dubbo:method>	onreturn	attribute属性,不在URL中 体现	String	可並

<dubbo:method></dubbo:method>	onthrow	attribute属性,不在URL中 体现	String	可並
<dubbo:method></dubbo:method>	cache	<methodname>.cache</methodname>	string/boolean	可並
<dubbo:method></dubbo:method>	validation	<methodname>.validation</methodname>	boolean	可並

比如:

方法参数配置:

配置类:com.alibaba.dubbo.config.ArgumentConfig

说明:该标签为<dubbo:method>的子标签,用于方法参数的特征描述,比如:

```
<dubbo:method name="findXxx" timeout="3000" retries="2">
     <dubbo:argument index="0" callback="true" />
</dubbo:method>
```

标签	属性	对应URL参数	类型	是否必填	缺省值	竹
<dubbo:argument></dubbo:argument>	index		int	必塡		析
<dubbo:argument></dubbo:argument>	type		String	与index 二选一		杤
<dubbo:argument></dubbo:argument>	callback	<metodname> <index>.retries</index></metodname>	boolean	可选		那

选项参数配置:

配置类:java.util.Map

说明:该标签

为 <dubbo:protocol> 或 <dubbo:service> 或 <dubbo:provider> 或 <dubbo:reference> 或 <dubbo:consumer> 的子标签,用于配置自定义参数,该配置项将作为扩展点设置自定义参数使用。

标签	属性	对应URL参数	类型	是否必填	缺省值	作用
<dubbo:parameter></dubbo:parameter>	key	key	string	必填		服务治理
<dubbo:parameter></dubbo:parameter>	value	value	string	必塡		服务治理

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" />
```

推荐使用Dubbo协议
性能测试报告
各协议的性能情况,请参见:性能测试报告

Dubbo缺省协议采用单一长连接和NIO异步通讯,适合于小数据量大并发的服务调用,以及服务消费者机器数远大于服务提供者机器数的情况。

Dubbo缺省协议不适合传送大数据量的服务,比如传文件,传视频等,除非请求量很低。

```
<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 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" />

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

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"> 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" />

缺省协议,使用基于mina1.1.7+hessian3.2.1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

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(1024Mbit=128MByte),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(1024Mbit=128MByte),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) 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:(由吴亚军提供)

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 | serialld 不相同 | 正常传输

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

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。

(2) 配置:

dubbo.properties

dubbo.service.protocol=dubbo

	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。	

如果正在使用RMI提供服务给外部访问(公司内网环境应该不会有攻击风险),同时应用里依赖了老的common-collections包(dubbo不会依赖这个包,请排查自己的应用有没有使用)的情况下,存在反序列化安全风险。

请检查应用:

- 将commons-collections3 请升级到 3.2.2版本
- 将commons-collections4 请升级到 4.1版本 新版本的commons-collections解决了该问题

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。

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" />
```

Java标准的远程调用协议

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

(1) 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

(2) 配置:

dubbo.properties

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

(3) RMI配置:

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

更多RMI优化参数请查看:

http://download.oracle.com/docs/cd/E17409_01/javase/6/docs/technotes/guides/rmi/sunrmip roperties.html

Hessian协议用于集成Hessian的服务,Hessian底层采用Http通讯,采用Servlet 暴露服务,Dubbo缺省内嵌Jetty作为服务器实现。

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。

依赖:

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

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。

基于Hessian的远程调用协议。

- 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

(1) 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。

(2) 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" />
```

采用Spring的HttpInvoker实现
2.3.0以上版本支持

基于http表单的远程调用协议。参见:HTTP协议使用说明

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。

(1) 约束:

• 参数及返回值需符合Bean规范

(2) 配置: dubbo.xml:

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

h4. Jetty Server: (default)

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

h4. Servlet Bridge Server: (recommend)

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

web.xml:

注意,如果使用servlet派发请求:

- 协议的端口必须与servlet容器的端口相同,
- 协议的上下文路径必须与servlet应用的上下文路径相同。

```
2.3.0以上版本支持。

基于CXF的frontend-simple和transports-http实现。

CXF是Apache开源的一个RPC框架:http://cxf.apache.org,由Xfire和Celtix合并而来。
```

依赖:

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。

基于WebService的远程调用协议:

• 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。

(1) 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。

(2) 配置:

- Define hessian protocol:
- Set default protocol:

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

Set service protocol:

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

• Multi port:

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

• Directly provider:

```
<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
```

h4. Jetty Server: (default)

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

h4. Servlet Bridge Server: (recommend)

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

web.xml:

注意,如果使用servlet派发请求:

- 协议的端口必须与servlet容器的端口相同,
- 协议的上下文路径必须与servlet应用的上下文路径相同。

	2.3.0以上版本支持。
	Thrift 是Facebook捐给Apache的一个RPC框架
[当前 dubbo 支持的 thrift 协议是对 thrift 原生协议的扩展,在原生协议的基础上添
)	加了一些额外的头信息,比如service name,magic number等。使用dubbo thrift协议同
,	样需要使用thrift的idl compiler编译生成相应的java代码,后续版本中会在这方面做一些
	增强。

示例:https://github.com/alibaba/dubbo/tree/master/dubbo-rpc/dubbo-rpc-thrift/src/test/java/com/alibaba/dubbo/rpc/protocol/thrift/examples

依赖:

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

所有服务共用一个端口:(与原生Thrift不兼容)

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

Thrift不支持数据类型:null值 (不能在协议中传递null值)

2.3.0以上版本支持。

Memcached 是一个高效的KV缓存服务器。

可以通过脚本或监控中心手工填写表单注册memcached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("memcached://10.20.153.11/com.foo.BarService?category=pr
oviders&dynamic=false&application=foo&group=member&loadbalance=consistenthash"));

然后在客户端使用时,不需要感知Memcached的地址:

<dubbo:reference id="cache" interface="http://10.20.160.198/wiki/display/dubbo/java.ut
il.Map" group="member" />

或者,点对点直连:

<dubbo:reference id="cache" interface="http://10.20.160.198/wiki/display/dubbo/java.ut
il.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://10.20.15
3.10:11211" />

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:(其中"p:xxx"为spring的标准p标签)

<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" />

2.3.0以上版本支持。

Redis 是一个高效的KV存储服务器。

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=providers&dynamic=false&application=foo&group=member&loadbalance=consistenthash"));

然后在客户端使用时,不需要感知Redis的地址:

 $\verb| <dubbo:reference id="store" interface="http://10.20.160.198/wiki/display/dubbo/java.util.Map" group="member" /> \\$

或者,点对点直连:

<dubbo:reference id="store" interface="http://10.20.160.198/wiki/display/dubbo/java.ut il.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:(其中"p:xxx"为spring的标准p标签)

<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" />

推荐使用Zookeeper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。
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

- 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。

```
<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

建议使用dubbo-2.3.3以上版本的zookeeper注册中心客户端。
Zookeeper说明
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。
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。
Z/Zookeeper.// 127.0.0. 1.2 То Гир 1 1/2/11 °
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。

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。

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:http://repo1.maven.org/maven2/org/apache/zookeeper/zookeeper

支持zkclient和curator两种Zookeeper客户端实现:

ZKClient Zookeeper Registry

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,开源比较早,参

见: https://github.com/sgroschupf/zkclient

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:

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

或直接下载:http://repo1.maven.org/maven2/com/github/sgroschupf/zkclient

Curator Zookeeper Registry

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,比较活跃。

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:

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

或直接下载:http://repo1.maven.org/maven2/com/netflix/curator/curator-framework

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" />
```

Redis 是一个高效的KV存储服务器。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。

Redis过期数据

通过心跳的方式检测脏数据,服务器时间必须相同,并且对服务器有一定压力。

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。

从2.1.0版本开始支持

数据结构:

使用Redis的Key/Map结构存储数据

- 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。(注意:服务器时间必需同步,否则过期检测会不准确)

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。

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。
 - o 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。
 - o 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。

配置 redis registry

```
<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="10.20.153.10:6379,10.20.153.11:6379,10.20.15
3.12:6379" />

Dogfooding
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。
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。

Export simple registry service:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"</pre>
    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"</pre>
yService" registry="N/A" ondisconnect="disconnect" callbacks="1000">
        <dubbo:method name="subscribe"><dubbo:argument index="1" callback="true" /></d</pre>
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</pre>
Service" />
</beans>
```

Reference the simple registry service:

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

Or:

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

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

⚠ Dubbo2.0.5以上版本服务提供端口支持telnet命令,

使用如:

telnet localhost 20880

或者:

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

telnet命令可以扩展,参见:扩展参考手册。 status命令所检查的资源也可以扩展,参见:扩展参考手册。

Is

(list services and methods)

- Is: 显示服务列表。
- IS-I: 显示服务详细信息列表。
- Is XxxService: 显示服务的方法列表。
- Is -I XxxService: 显示服务的方法详细信息列表。

ps

(print server ports and connections)

- ps: 显示服务端口列表。
- ps-I: 显示服务地址列表。
- ps 20880: 显示端口上的连接信息。
- ps-I 20880: 显示端口上的连接详细信息。

cd

(change default service)

- 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。
- cd /: 取消缺省服务。

pwd

(print working default service)

• pwd: 显示当前缺省服务。

trace

- trace XxxService: 跟踪1次服务任意方法的调用情况。
- trace XxxService 10: 跟踪10次服务任意方法的调用情况。
- trace XxxService xxxMethod: 跟踪1次服务方法的调用情况
- trace XxxService xxxMethod 10: 跟踪10次服务方法的调用情况。

count

- count XxxService: 统计1次服务任意方法的调用情况。
- count XxxService 10: 统计10次服务任意方法的调用情况。
- count XxxService xxxMethod: 统计1次服务方法的调用情况。
- count XxxService xxxMethod 10: 统计10次服务方法的调用情况。

invoke

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

status

- 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。
- status -l: 显示状态列表。

log

2.0.6以上版本支持

- log debug: 修改dubbo logger的日志级别
- log 100: 查看file logger的最后100字符的日志

help

- help: 显示telnet命帮助信息。
- help xxx: 显示xxx命令的详细帮助信息。

clear

- clear: 清除屏幕上的内容。
- 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。

mvn dubbo:registry

以缺省的9090端口启动一个简易注册中心

样例:以指定的9099端口启动一个简易注册中心

mvn dubbo:registry -Dport=9099

mvn dubbo:create

(尚未发布)

生成demo服务提供者应用

样例: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

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set,get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byRef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。

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。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。

PS: 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 Hard Code的配置值(见Dubbo配置参考手册)

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(表示加上第一次调用,会调用3次)
- 3. loadbalance 负载均衡算法(有多个Provider时,如何挑选Provider调用),缺省是随机(random)。还可以有轮询(roundrobin)、最不活跃优先(leastactive,指从Consumer 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)
- 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(Consumer可能到超时)。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。

配置上管理信息

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。

示例:

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。

配置上Dubbo缓存文件

配置方法如下:

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:

- 1. 注册中心的列表
- 2. 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

Dubbo2中所有的配置项都可以Spring配置中,并且可以针对单个服务配置。

如完全不配置使用Dubbo缺省值,参见 Dubbo配置参考手册 中的说明。

在Dubbo1中需要在dubbo.properties文件中的配置项,Dubbo2中配置示例如下:

1. 应用名

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

对应dubbo.properties中的Key名 dubbo.application.name

2. 注册中心地址

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

对应**dubbo**.**properties**中的**Key**名 dubbo.registry.address

3. 调用超时

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

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 全局配置项值,对应dubbo.properties中的Key名 dubbo.service.invoke.timeout

4. 服务提供者协议、服务的监听端口

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

对应dubbo.properties中的Key名 dubbo.service.protocol、dubbo.service.server.port

5. 服务线程池大小

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

对应 dubbo.properties 中的Key名 dubbo.service.max.thread.threads.size

6. 消费者启动时,没有提供者是否抛异常Fast-Fail

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

对应 dubbo.properties 中的Key名 alibaba.intl.commons.dubbo.service.allow.no.provider

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)

基准测试工具包

下载 benchmark压缩包,并解压

阅读ReadMe.txt (内容如下,请以压缩包内的为准)

- 新建一个benchmark工程,如demo.benchmark
- 导入自己服务的接口api包和dubbo.benchmark.jar(解压dubbo.benchmark.tar.gz, 在lib目录下)
- 新建一个类,实现AbstractClientRunnable
 - o 实现父类的构造函数
 - 。 实现invoke方法,通过serviceFactory创建本地接口代理,并实现自己的业务逻辑, 如下

```
public Object invoke(ServiceFactory serviceFactory) {
    DemoService demoService = (DemoService) serviceFactory.get(DemoService.
class);
    return demoService.sendRequest("hello");
}
```

- 将自己的benchmark工程打成jar包,如demo.benchmark.jar
- 将demo.benchmark.jar 和服务的api包放到dubbo.benchmark/lib目录下
- 配置duubo.properties
- 运行run.bat(windows)或run.sh(linux)

如想测试dubbo的不同版本,直接替换lib下的dubbo的jar包即可。

测试说明

- 1. 本次性能测试,测试了dubbo2.0所有支持的协议在不同大小和数据类型下的表现,并与dubbo1.0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dubbo2.0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。(可以确认为mina在高并发下的设计缺陷)
- 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	足

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1kString,不做任何处理,原样返回
 - 。 传入50kString,不做任何处理,原样返回
 - 。 传入200kString,不做任何处理,原样返回
 - o 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dubbo1.0\dubbo2.0(hessian2序列化)\dubbo2.0(dubbo序列化)\rmi\hessian3.2.0\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。

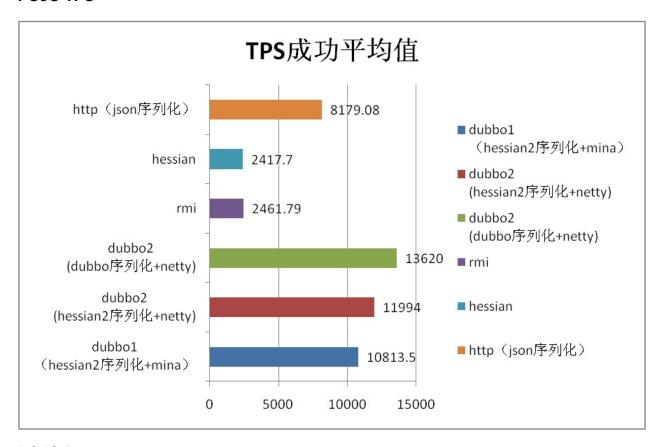
- 2. 并发场景(20并发) 传入1kString,在服务器段循环1w次,每次重新生成一个随机数然后进行拼装。考察业务线程是否能够分配到每个CPU上。
- 3. 稳定性场景(20并发) 同时调用1个参数为String(5k)方法,1个参数为person对象的方法,1个参数为map(值为3个person)的方法,持续运行50小时。
- 4. 高压力场景(20并发) 在稳定性场景的基础上,将提供者和消费者布置成均为2台(一台机器2个实例),且String的参数从20byte到200k,每隔10分钟随机变换。

测试结果

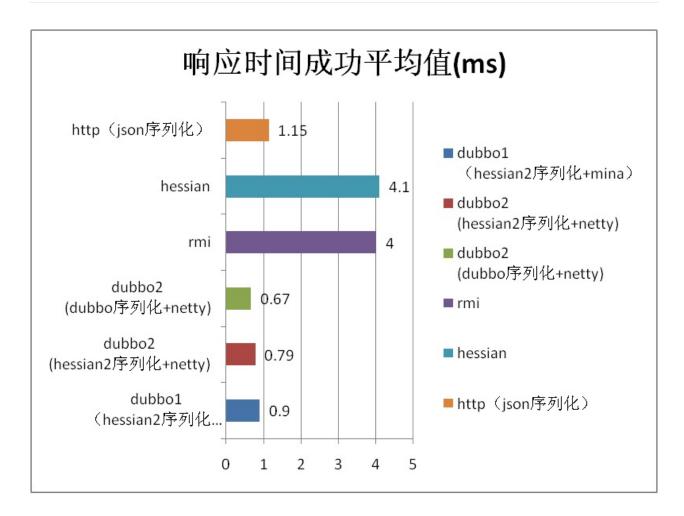
场景名称:POJO 场景

	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



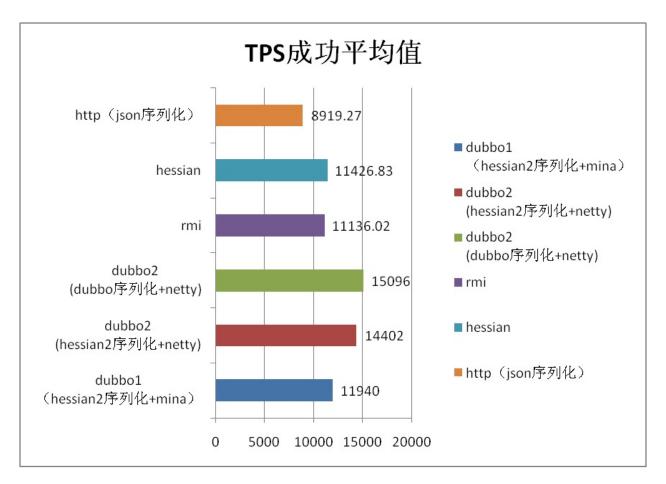
POJO Response



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



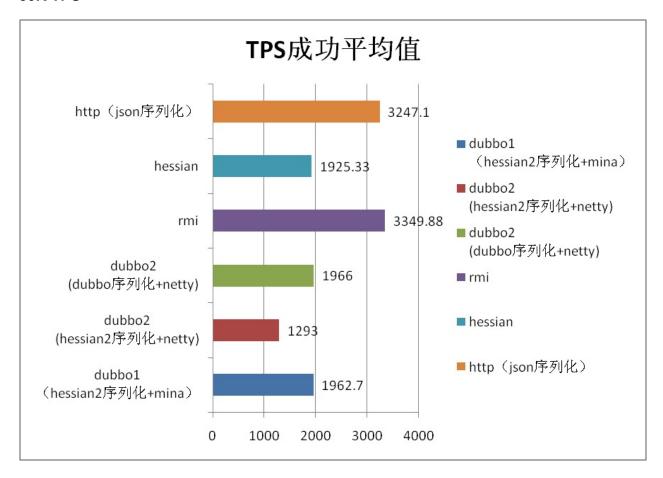
1k Response



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



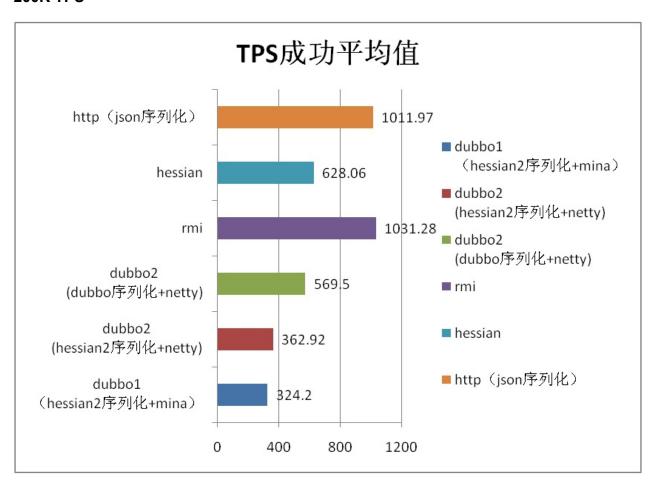
50K Response

/user-guide/images/50kres.png

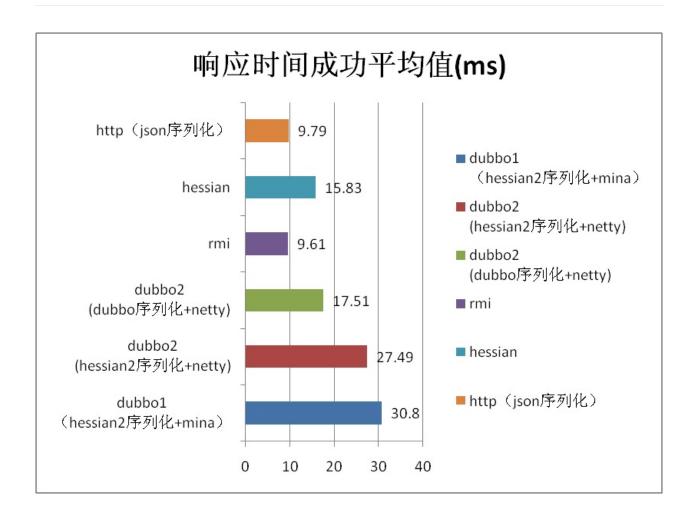
场景名称: 200k string 场景

	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

Dubbo2.0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。如下图:

性能对比分析(新旧环境、不同数据量级等)

Dubbo2.0相比较Dubbo1.0(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。

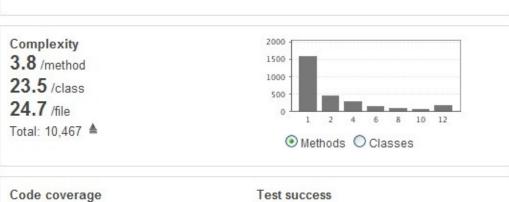
由于邮件篇幅所限没有列出所有的监控图,如需获得可在大力神平台上查询。

测试覆盖率报告

统计于 2012-02-03 (2.0.12)



Comments Duplications 9.2% 3.8% 3,575 lines ▲ 2,077 lines ▲ 32.1% docu. API 99 blocks 1,839 undocu. API 47 files 52 commented LOCs



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

0 errors
817 tests
+102 skipped
55.9 sec

Test success
0 failures
0 errors

Lines of code

35,406 ≜

54,469 lines ≜

17,315 statements ≜

423 files

Classes

445

90 packages

2,762 methods A

+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 ≜

2000 1500 1000 500 1 2 4 6 8 10 12

Methods O Classes

Code coverage

40.0%

43.4% line coverage

34.1% branch coverage

100.0% 0 failures

0 errors

817 tests

+102 skipped

Test success

55.9 sec ≜

Lines of code

35,406 ≜

54,469 lines ≜

17,315 statements ≜

423 files

Classes

445

90 packages

2,762 methods A

+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 ≜

2000 1500 1000 500 0 1 2 4 6 8 10 12 • Methods • Classes

Code coverage

40.0%

43.4% line coverage

34.1% branch coverage

100.0%

0 failures

0 errors

817 tests

+102 skipped

Test success

55.9 sec ≜

Lines of code

35,406 ≜

54,469 lines ≜

17,315 statements ≜

423 files

Classes

445

90 packages

2,762 methods A

+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 ≜

2000 1500 1000 500 0 1 2 4 6 8 10 12 • Methods • Classes

Code coverage

40.0%

43.4% line coverage

34.1% branch coverage

100.0%

0 failures

0 errors

817 tests

+102 skipped

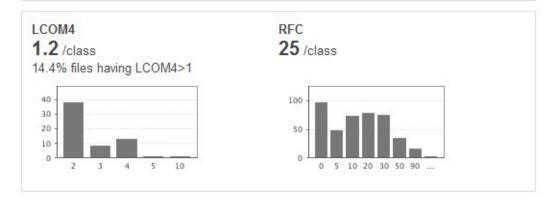
Test success

55.9 sec ≜



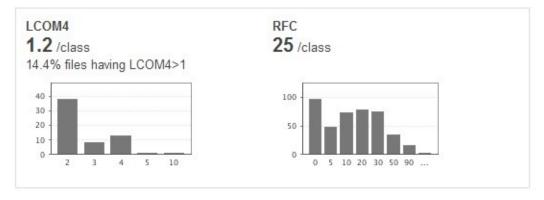


Dependencies to cut
7 between packages
14 between files



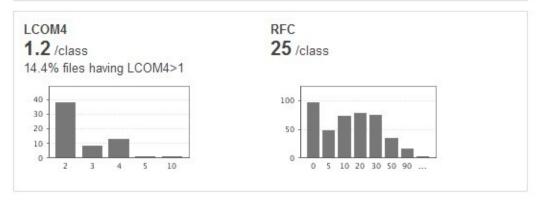












	Name	Rules compliance	Line coverage	Branch coverage
Q	Dubbo Common Module	69.9%	58.1%	43.3%
Q	Dubbo Remoting Module	82.6%	25.3%	20.7%
Q	Dubbo Netty Remoting Module	79.9%	75.7%	53.6%
Q	Dubbo Mina Remoting Module	81.2%	69.1%	42.7%
Q	Dubbo Grizzly Remoting Module	79.6%	0.0%	0.0%
Q	Dubbo P2P Remoting Module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

