**Dubbo**

**学习手册**

**V1.0**

**张镇**

**目 录**

[1 背景和简介 1](#_Toc490063885)

[2 Dubbo是什么 4](#_Toc490063886)

[2.1 Dubbo架构 4](#_Toc490063887)

[2.1.1 总体架构 4](#_Toc490063888)

[2.1.2 服务调用 7](#_Toc490063889)

[2.2 特性 10](#_Toc490063890)

[2.2.1 连通性 10](#_Toc490063891)

[2.2.2 健状性 10](#_Toc490063892)

[2.2.3 伸缩性 10](#_Toc490063893)

[2.2.4 升级性 10](#_Toc490063894)

[2.3 调用方式 11](#_Toc490063895)

[2.3.1 异步调用 11](#_Toc490063896)

[2.3.2 本地调用 12](#_Toc490063897)

[2.4 注册中心 12](#_Toc490063898)

[2.5 支持的远程通信协议 13](#_Toc490063899)

[2.6 支持的远程调用协议 14](#_Toc490063900)

[2.7 优缺点 14](#_Toc490063901)

[3 源代码说明 15](#_Toc490063902)

[3.1 源代码结构 15](#_Toc490063903)

[3.2 内核实现之SPI简单介绍 16](#_Toc490063904)

[4 Dubbo入门Demo 19](#_Toc490063905)

[4.1 Zookeeper介绍与安装 19](#_Toc490063906)

[4.2 创建MAVEN项目 20](#_Toc490063907)

[4.2.1 dubbo-provider服务者项目 20](#_Toc490063908)

[4.2.2 dubbo-consumer消费者项目 24](#_Toc490063909)

[4.2.3 测试 27](#_Toc490063910)

[5 Dubbo进阶 28](#_Toc490063911)

[5.1 集群容错和负载均衡 28](#_Toc490063912)

[5.1.1 集群容错 28](#_Toc490063913)

[5.1.2 负载均衡 31](#_Toc490063914)

[5.2 Dubbo服务集群容错实践 33](#_Toc490063915)

[5.3 Dubbo管理控制台介绍 43](#_Toc490063916)

# 背景和简介

Dubbo开始于电商系统，因此在这里先从电商系统的**演变**讲起。

1. **单一应用框架(ORM)**

当网站流量很小时，只需一个应用，将所有功能如下单支付等都部署在一起，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。

**缺点**：单一的系统[架构](http://lib.csdn.net/base/architecture)，使得在开发过程中，占用的资源越来越多，而且随着流量的增加越来越难以维护

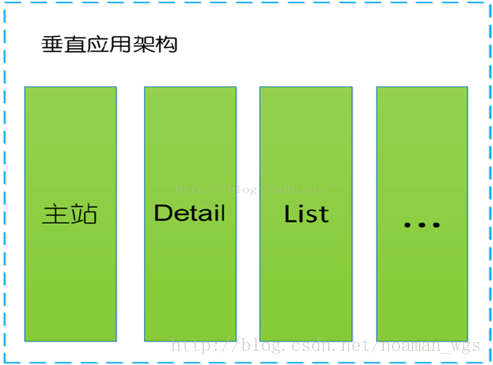


1. **垂直应用框架(MVC)**

垂直应用架构解决了单一应用架构所面临的扩容问题，流量能够分散到各个子系统当中，且系统的体积可控，一定程度上降低了开发人员之间协同以及维护的成本，提升了开发效率。

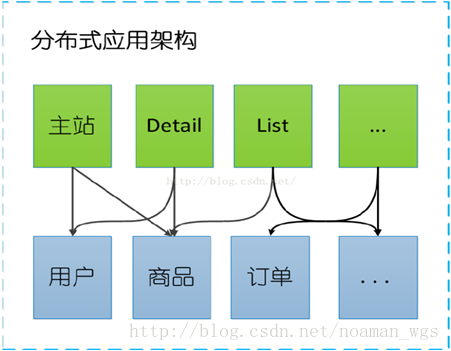
当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，将应用拆成互不相干的几个应用，以提升效率。此时，用于加速前端页面开发的 Web框架(MVC) 是关键。

**缺点**：但是在垂直架构中相同逻辑代码需要不断的复制，不能复用。



1. **分布式应用架构(RPC)**

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。



**小知识：RPC(Remote Procedure Call Protocol)：远程过程调用**：

两台服务器A、B，分别部署不同的应用a,b。当A服务器想要调用B服务器上应用b提供的函数或方法的时候，由于不在一个内存空间，不能直接调用，需要通过网络来表达调用的语义传达调用的数据。

说白了，就是你在你的机器上写了一个程序，我这边是无法直接调用的，这个时候就出现了一个远程服务调用的概念。

RPC是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。RPC协议假定某些传输协议的存在，如TCP或UDP，为通信程序之间携带信息数据。在OSI网络通信模型中，RPC跨越了传输层和应用层。RPC使得开发包括网络分布式多程序在内的应用程序更加容易。

RPC采用客户机/服务器模式。请求程序就是一个客户机，而服务提供程序就是一个服务器。首先，客户机调用进程发送一个有进程参数的调用信息到服务进程，然后等待应答信息。在服务器端，进程保持睡眠状态直到调用信息到达为止。当一个调用信息到达，服务器获得进程参数，计算结果，发送答复信息，然后等待下一个调用信息，最后，客户端调用进程接收答复信息，获得进程结果，然后调用执行继续进行。

**RPC需要解决的问题：**

(可以稍作了解，详情可查看别的博文)

\* **通讯问题** : 主要是通过在客户端和服务器之间建立TCP连接，远程过程调用的所有交换的数据都在这个连接里传输。连接可以是按需连接，调用结束后就断掉，也可以是长连接，多个远程过程调用共享同一个连接。

\* **寻址问题** ： A服务器上的应用怎么告诉底层的RPC框架，如何连接到B服务器（如主机或IP地址）以及特定的端口，方法的名称名称是什么，这样才能完成调用。比如基于Web服务协议栈的RPC，就要提供一个endpoint URI，或者是从UDDI服务上查找。如果是RMI调用的话，还需要一个RMI Registry来注册服务的地址。

\* **序列化 与 反序列化** ： 当A服务器上的应用发起远程过程调用时，方法的参数需要通过底层的网络协议如TCP传递到B服务器，由于网络协议是基于二进制的，内存中的参数的值要序列化成二进制的形式，也就是序列化（Serialize）或编组（marshal），通过寻址和传输将序列化的二进制发送给B服务器。

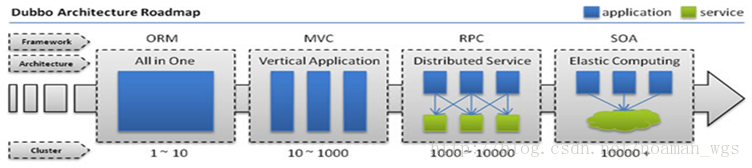
同理，B服务器接收参数要将参数反序列化。B服务器应用调用自己的方法处理后返回的结果也要序列化给A服务器，A服务器接收也要经过反序列化的过程。

1. **流动计算架构(SOA)**

随着服务化的进一步发展，服务越来越多，服务之间的调用和依赖关系也越来越复杂，诞生了面向服务的架构体系(SOA)，也因此衍生出了一系列相应的技术，如对服务提供、服务调用、连接处理、通信协议、序列化方式、服务发现、服务路由、日志输出等行为进行封装的服务框架

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。

从以上是电商系统的演变可以看出架构演变的过程：



# Dubbo是什么

Dubbo是：

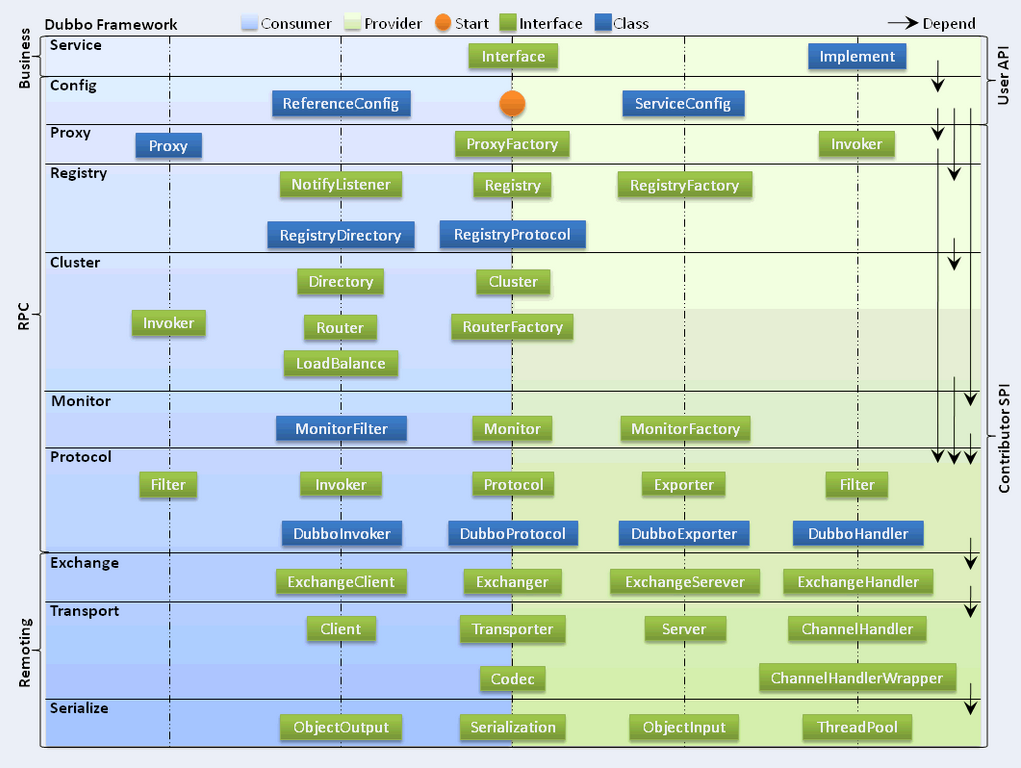
* 一款分布式服务框架
* 高性能和透明化的RPC远程服务调用方案
* SOA服务治理方案

每天为2千多个服务提供大于30亿次访问量支持，并被广泛应用于阿里巴巴集团的各成员站点以及别的公司的业务中。

## Dubbo架构

### 总体架构

Dubbo的总体架构，如图所示：



Dubbo框架设计一共划分了10个层，而最上面的Service层是留给实际想要使用Dubbo开发分布式服务的开发者实现业务逻辑的接口层。图中左边淡蓝背景的为服务消费方使用的接口，右边淡绿色背景的为服务提供方使用的接口， 位于中轴线上的为双方都用到的接口。

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

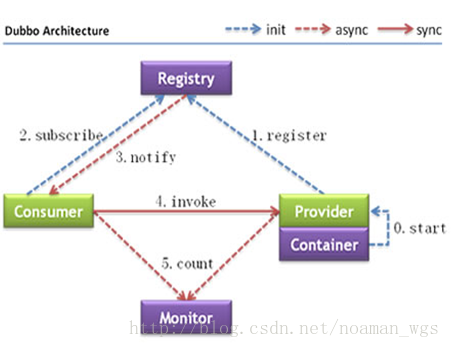
1. 服务接口层（Service）：该层是与实际业务逻辑相关的，根据服务提供方和服务消费方的业务设计对应的接口和实现。
2. 配置层（Config）：对外配置接口，以ServiceConfig和ReferenceConfig为中心，可以直接new配置类，也可以通过spring解析配置生成配置类。
3. 服务代理层（Proxy）：服务接口透明代理，生成服务的客户端Stub和服务器端Skeleton，以ServiceProxy为中心，扩展接口为ProxyFactory。
4. 服务注册层（Registry）：封装服务地址的注册与发现，以服务URL为中心，扩展接口为RegistryFactory、Registry和RegistryService。可能没有服务注册中心，此时服务提供方直接暴露服务。
5. 集群层（Cluster）：封装多个提供者的路由及负载均衡，并桥接注册中心，以Invoker为中心，扩展接口为Cluster、Directory、Router和LoadBalance。将多个服务提供方组合为一个服务提供方，实现对服务消费方来透明，只需要与一个服务提供方进行交互。
6. 监控层（Monitor）：RPC调用次数和调用时间监控，以Statistics为中心，扩展接口为MonitorFactory、Monitor和MonitorService。
7. 远程调用层（Protocol）：封将RPC调用，以Invocation和Result为中心，扩展接口为Protocol、Invoker和Exporter。Protocol是服务域，它是Invoker暴露和引用的主功能入口，它负责Invoker的生命周期管理。Invoker是实体域，它是Dubbo的核心模型，其它模型都向它靠扰，或转换成它，它代表一个可执行体，可向它发起invoke调用，它有可能是一个本地的实现，也可能是一个远程的实现，也可能一个集群实现。
8. 信息交换层（Exchange）：封装请求响应模式，同步转异步，以Request和Response为中心，扩展接口为Exchanger、ExchangeChannel、ExchangeClient和ExchangeServer。
9. 网络传输层（Transport）：抽象mina和netty为统一接口，以Message为中心，扩展接口为Channel、Transporter、Client、Server和Codec。
10. 数据序列化层（Serialize）：可复用的一些工具，扩展接口为Serialization、 ObjectInput、ObjectOutput和ThreadPool。

从上图可以看出，Dubbo对于服务提供方和服务消费方，从框架的10层中分别提供了各自需要关心和扩展的接口，构建整个服务生态系统（服务提供方和服务消费方本身就是一个以服务为中心的）。

根据官方提供的，对于上述各层之间关系的描述，如下所示：

* 在RPC中，Protocol是核心层，也就是只要有Protocol + Invoker + Exporter就可以完成非透明的RPC调用，然后在Invoker的主过程上Filter拦截点。
* 图中的Consumer和Provider是抽象概念，只是想让看图者更直观的了解哪些类分属于客户端与服务器端，不用Client和Server的原因是Dubbo在很多场景下都使用Provider、Consumer、Registry、Monitor划分逻辑拓普节点，保持统一概念。
* 而Cluster是外围概念，所以Cluster的目的是将多个Invoker伪装成一个Invoker，这样其它人只要关注Protocol层Invoker即可，加上Cluster或者去掉Cluster对其它层都不会造成影响，因为只有一个提供者时，是不需要Cluster的。
* Proxy层封装了所有接口的透明化代理，而在其它层都以Invoker为中心，只有到了暴露给用户使用时，才用Proxy将Invoker转成接口，或将接口实现转成Invoker，也就是去掉Proxy层RPC是可以Run的，只是不那么透明，不那么看起来像调本地服务一样调远程服务。
* 而Remoting实现是Dubbo协议的实现，如果你选择RMI协议，整个Remoting都不会用上，Remoting内部再划为Transport传输层和Exchange信息交换层，Transport层只负责单向消息传输，是对Mina、Netty、Grizzly的抽象，它也可以扩展UDP传输，而Exchange层是在传输层之上封装了Request-Response语义。
* Registry和Monitor实际上不算一层，而是一个独立的节点，只是为了全局概览，用层的方式画在一起。

### 服务调用



* **Provider**: 暴露服务的服务提供方。
* **Consumer**: 调用远程服务的服务消费方。
* **Registry**: 服务注册与发现的注册中心。
* **Monitor**: 统计服务的调用次数和调用时间的监控中心。

**调用流程** ：

0.服务容器负责启动，加载，运行服务提供者。

1.服务提供者在启动时，向注册中心注册自己提供的服务。

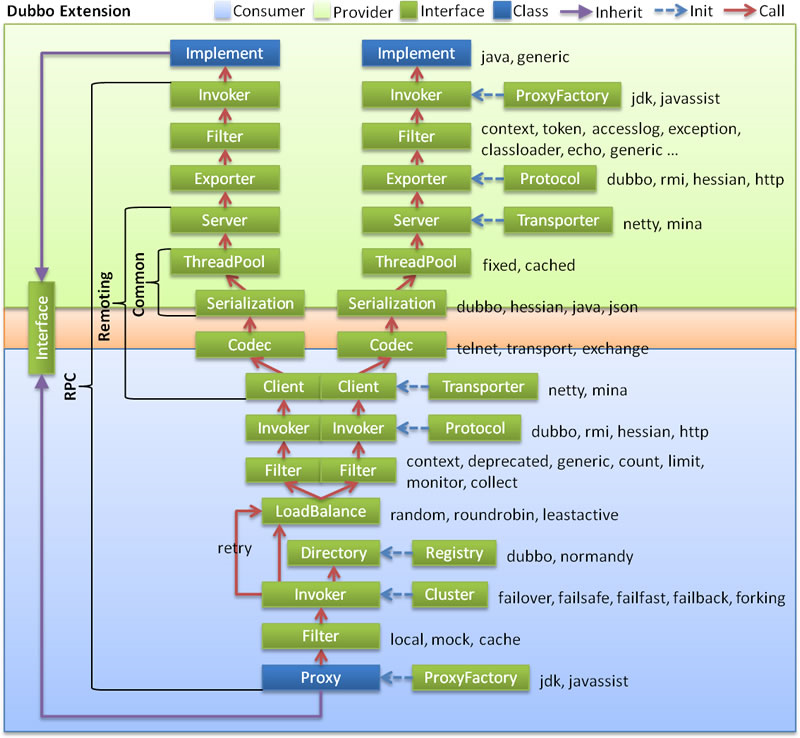
2.服务消费者在启动时，向注册中心订阅自己所需的服务。

3.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。

4.服务消费者，从提供者地址列表中，基于软负载均衡[算法](http://lib.csdn.net/base/datastructure)，选一台提供者进行调用，如果调用失败，再选另一台调用。

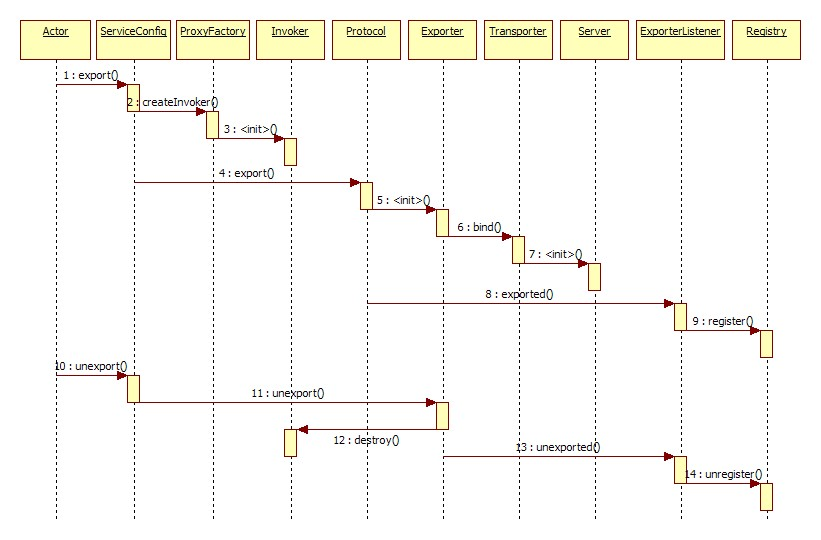
5.服务消费者和提供者，在内存中累计调用次数和调用时间，定时每分钟发送一次统计数据到监控中心

将上面抽象的调用流程图展开，详细如图所示：



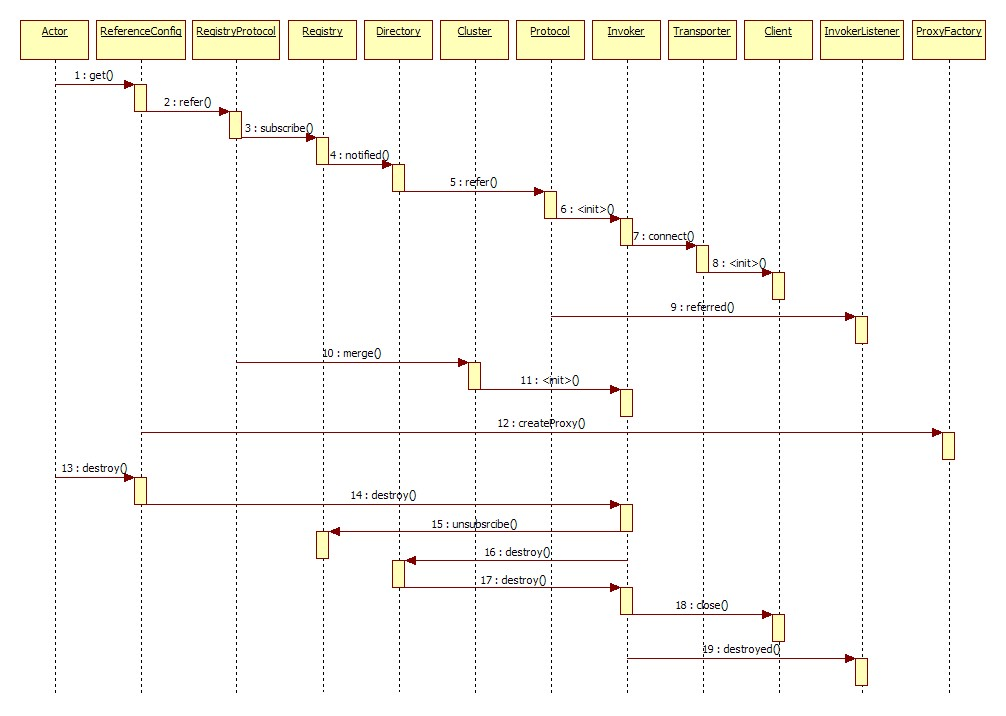
**注册/注销服务**

服务的注册与注销，是对服务提供方角色而言，那么注册服务与注销服务的时序图，如图所示：



**服务订阅/取消**

为了满足应用系统的需求，服务消费方的可能需要从服务注册中心订阅指定的有服务提供方发布的服务，在得到通知可以使用服务时，就可以直接调用服务。反过来，如果不需要某一个服务了，可以取消该服务。下面看一下对应的时序图，如图所示：



## 特性

### 连通性

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

注册中心和监控中心都是可选的，服务消费者可以直连服务提供者

### 健状性

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

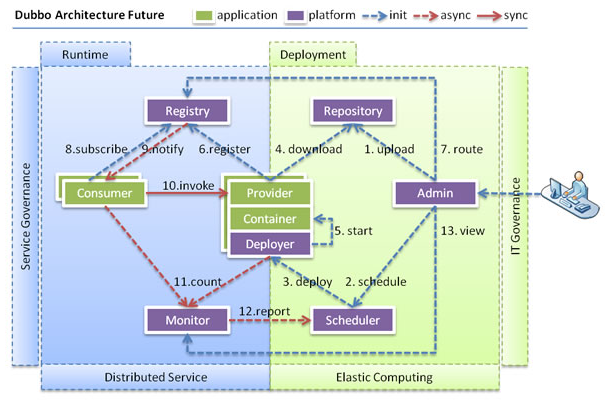
### 伸缩性

注册中心为对等集群，可动态增加机器部署实例，所有客户端将自动发现新的注册中心

服务提供者无状态，可动态增加机器部署实例，注册中心将推送新的服务提供者信息给消费者

### 升级性

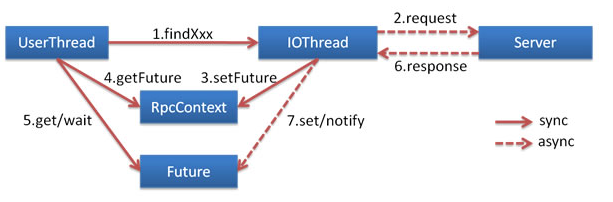
当服务集群规模进一步扩大，带动IT治理结构进一步升级，需要实现动态部署，进行流动计算，现有分布式服务架构不会带来阻力：



## 调用方式

### 异步调用

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



### 本地调用

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

**Define injvm protocol:**

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

**Set default protocol:**

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

**Set service protocol:**

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

**Use injvm first:**

<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”

## 注册中心

对于服务提供方，它需要发布服务，而且由于应用系统的复杂性，服务的数量、类型也不断膨胀；

对于服务消费方，它最关心如何获取到它所需要的服务，而面对复杂的应用系统，需要管理大量的服务调用。

而且，对于服务提供方和服务消费方来说，他们还有可能兼具这两种角色，即既需要提供服务，有需要消费服务。

通过将服务统一管理起来，可以有效地优化内部应用对服务发布/使用的流程和管理。服务注册中心可以通过特定协议来完成服务对外的统一。

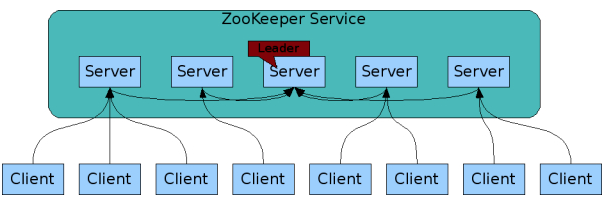
**Dubbo提供的注册中心有如下几种类型可供选择**：

* Multicast注册中心
* Zookeeper注册中心
* [Redis](http://lib.csdn.net/base/redis)注册中心
* Simple注册中心

目前常用ZooKeeper作为服务注册中心。

ZooKeeper是一个开源的分布式服务框架，它是Apache [**Hadoop**](http://lib.csdn.net/base/hadoop)项目的一个子项目，主要用来解决分布式应用场景中存在的一些问题，如：**统一命名服务、状态同步服务、集群管理、分布式应用配置管理等**，它支持Standalone模式和分布式模式，在分布式模式下，能够为分布式应用提供高性能和可靠地协调服务，而且使用ZooKeeper可以大大简化分布式协调服务的实现，为开发分布式应用极大地降低了成本。

ZooKeeper总体架构

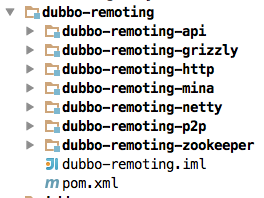


ZooKeeper集群由一组Server节点组成，这一组Server节点中存在一个角色为Leader的节点，其他节点都为Follower。当客户端Client连接到ZooKeeper集群，并且执行写请求时，这些请求会被发送到Leader节点上，然后Leader节点上数据变更会同步到集群中其他的Follower节点。

## 支持的远程通信协议

远程通信需要指定通信双方所约定的协议，在保证通信双方理解协议语义的基础上，还要保证高效、稳定的消息传输。Dubbo继承了当前主流的网络通信框架，主要包括如下几个：

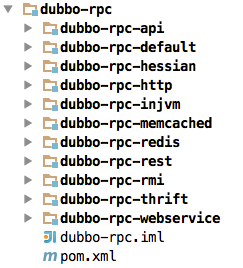
* Mina
* Netty
* Grizzly



## 支持的远程调用协议

Dubbo支持多种协议，如下所示：

* Dubbo协议
* Hessian协议
* HTTP协议
* RMI协议
* WebService协议
* Thrift协议
* Memcached协议
* Redis协议



在通信过程中，不同的服务等级一般对应着不同的服务质量，那么选择合适的协议便是一件非常重要的事情。你可以根据你应用的创建来选择。例如，使用RMI协议，一般会受到防火墙的限制，所以对于外部与内部进行通信的场景，就不要使用RMI协议，而是基于HTTP协议或者Hessian协议。

## 优缺点

**优点：**

1. 透明化的远程方法调用：像调用本地方法一样调用远程方法；只需简单配置，没有任何API侵入。

2. 软负载均衡及容错机制：可在内网替代nginx lvs等硬件负载均衡器。

3. 服务注册中心自动注册 & 配置管理：不需要写死服务提供者地址，注册中心基于接口名自动查询提供者ip。使用类似zookeeper等分布式协调服务作为服务注册中心，可以将绝大部分项目配置移入zookeeper集群。

4. 服务接口监控与治理：Dubbo-admin与Dubbo-monitor提供了完善的服务接口管理与监控功能，针对不同应用的不同接口，可以进行 多版本，多协议，多注册中心管理。

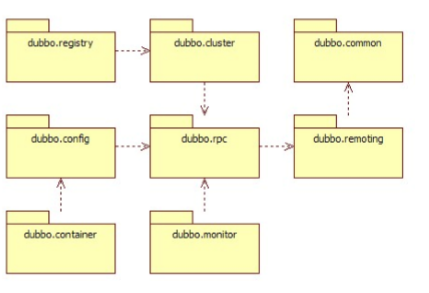
**缺点：**只支持JAVA语言

# 源代码说明

## 源代码结构



Dubbo以包结构来组织各个模块，各个模块及其关系，如图所示：



* dubbo-common 公共逻辑模块，包括Util类和通用模型。
* dubbo-remoting 远程通讯模块，相当于Dubbo协议的实现，如果RPC用RMI协议则不需要使用此包。
* dubbo-rpc 远程调用模块，抽象各种协议，以及动态代理，只包含一对一的调用，不关心集群的管理。
* dubbo-cluster 集群模块，将多个服务提供方伪装为一个提供方，包括：负载均衡、容错、路由等，集群的地址列表可以是静态配置的，也可以是由注册中心下发。
* dubbo-registry 注册中心模块，基于注册中心下发地址的集群方式，以及对各种注册中心的抽象。
* dubbo-monitor 监控模块，统计服务调用次数，调用时间的，调用链跟踪的服务。
* dubbo-config 配置模块，是Dubbo对外的API，用户通过Config使用Dubbo，隐藏Dubbo所有细节。
* dubbo-[**Container**](http://lib.csdn.net/base/docker) 容器模块，是一个Standalone的容器，以简单的Main加载Spring启动，因为服务通常不需要Tomcat/JBoss等Web容器的特性，没必要用Web容器去加载服务。

## 内核实现之SPI简单介绍

Dubbo采用微内核+插件体系，使得设计优雅，扩展性强。那所谓的微内核+插件体系是如何实现的呢！即我们定义了服务接口标准，让厂商去实现（如果不了解spi的请谷歌百度下）, jdk通过ServiceLoader类实现spi机制的服务查找功能。

JDK实现spi服务查找: ServiceLoader

首先定义下示例接口

**package** com.example;

**public** **interface** Spi {

booleanisSupport(String name);

String sayHello();

}

ServiceLoader会遍历所有jar查找META-INF/services/com.example.Spi文件

A厂商提供实现

**package** com.a.example;

**public** **class** SpiAImpl **implements** Spi {

publicboolean isSupport(String name) {

**return**"SPIA".equalsIgnoreCase(name.trim());

}

**public** String syaHello() {

**return** “hello 我是厂商A”;

}

}

在A厂商提供的jar包中的META-INF/services/com.example.Spi文件内容为：

com.a.example.SpiAImpl #厂商A的spi实现全路径类名

B厂商提供实现

**package** com.b.example;

**public** **class** SpiBImpl **implements** Spi {

publicboolean isSupport(String name) {

**return**"SPIB".equalsIgnoreCase(name.trim());

}

**public** String syaHello() {

**return** “hello 我是厂商B”;

}

}

在B厂商提供的jar包中的META-INF/services/com.example.Spi文件内容为：

com.b.example.SpiBImpl #厂商B的spi实现全路径类名

ServiceLoader.load(Spi.class)读取厂商A、B提供jar包中的文件，ServiceLoader实现了Iterable接口可通过while for循环语句遍历出所有实现。

一个接口多种实现，就如策略模式一样提供了策略的实现，但是没有提供策略的选择， 使用方可以根据isSupport方法根据业务传入厂商名来选择具体的厂商。

**public** **class** SpiFactory {

//读取配置获取所有实现

privatestatic ServiceLoader spiLoader = ServiceLoader.load(Spi.**class**);

//根据名字选取对应实现

publicstatic Spi getSpi(String name) {

**for**(Spi spi : spiLoader) {

**if**(spi.isSupport(name) ) {

returnspi;

}

}

returnnull;

}

}

**SPI接口定义**

定义了@SPI注解

**public** **@interface** SPI {

Stringvalue() **default** ""; //指定默认的扩展点

}

只有在接口打了@SPI注解的接口类才会去查找扩展点实现，会依次从这几个文件中读取扩展点

META-INF/dubbo/internal/ //dubbo内部实现的各种扩展都放在了这个目录了

META-INF/dubbo/

META-INF/services/

我们以Protocol接口为例， 接口上打上SPI注解，默认扩展点名字为dubbo

@SPI("dubbo")

**public** **interface** Protocol{

}

具体实现的类有:



所以说：Remoting实现是Dubbo协议的实现

# Dubbo入门Demo

了解了Dubbo以后，自然要搭建一个简单的Demo实现。本文采用Dubbo与Zookeeper、[spring](http://lib.csdn.net/base/javaee)框架的整合。

主要是以下几个步骤：

1. 安装Zookeeper,启动；

2. 创建MAVEN项目，构建Dubbo+Zookeeper+Spring实现的简单Demo；

3. 安装Dubbo-admin，实现监控。

## Zookeeper介绍与安装

本Demo中的Dubbo注册中心采用的是Zookeeper。为什么采用Zookeeper呢？

Zookeeper是一个分布式的服务框架，是树型的目录服务的数据存储，能做到集群管理数据 ，这里能很好的作为Dubbo服务的注册中心。Dubbo能与Zookeeper做到集群部署，当提供者出现断电等异常停机时，Zookeeper注册中心能自动删除提供者信息，当提供者重启时，能自动恢复注册数据，以及订阅请求

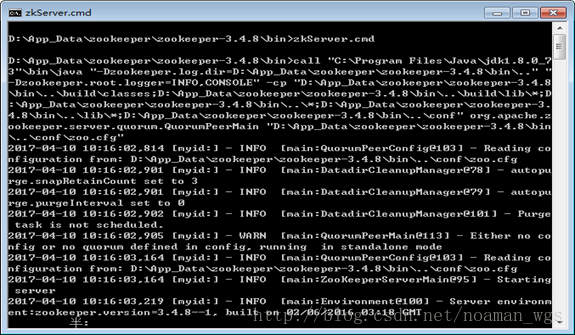
首先安装zookeeper，下载zookeeper-3.3.6.tar安装。

下载地址：http://apache.fayea.com/zookeeper/zookeeper-3.3.6/

注意\zookeeper-3.3.6\conf\下的zoo\_sample.cfg复制一份改成zoo.cfg,目的防止启动zk时闪退。

安装完成后，进入到bin目录，并且启动zkServer.cmd，这个脚本中会启动一个[Java](http://lib.csdn.net/base/java)进程：

(注：需要先启动zookeeper后，后续dubbo demo代码运行才能使用zookeeper注册中心的功能)



## 创建MAVEN项目

### dubbo-provider服务者项目

        1、pox.xml配置

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>dubbo-server</groupId>

<artifactId>dubbo-server</artifactId>

<version>0.0.1-SNAPSHOT</version>

<packaging>jar</packaging>

<name>dubbo-server</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.1.1</version>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>dubbo</artifactId>

<version>2.5.3</version>

</dependency>

<dependency>

<groupId>org.javassist</groupId>

<artifactId>javassist</artifactId>

<version>3.18.1-GA</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.15</version>

<exclusions>

<exclusion>

<groupId>com.sun.jdmk</groupId>

<artifactId>jmxtools</artifactId>

</exclusion>

<exclusion>

<groupId>com.sun.jmx</groupId>

<artifactId>jmxri</artifactId>

</exclusion>

<exclusion>

<artifactId>jms</artifactId>

<groupId>javax.jms</groupId>

</exclusion>

<exclusion>

<artifactId>mail</artifactId>

<groupId>javax.mail</groupId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring</artifactId>

<version>2.5.6.SEC03</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.6</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.6.1</version>

</dependency>

<dependency>

<groupId>org.apache.zookeeper</groupId>

<artifactId>zookeeper</artifactId>

<version>3.4.5</version>

<type>pom</type>

</dependency>

<dependency>

<groupId>com.101tec</groupId>

<artifactId>zkclient</artifactId>

<version>0.4</version>

</dependency>

</dependencies>

</project>

2、服务器service代码

**package** com.demo.server.service;

**public** **interface** DemoTest {

**public** String hello(String name);

}

==========================================================

**package** com.demo.server.service.impl;

**import** com.demo.server.service.DemoTest;

**public** **class** DemoTestImpl **implements** DemoTest{

**public** String hello(String name) {

**return** "hello "+ name;

}

/\*\*

\* 实现公共接口，此实现对消费者隐藏

\* **@param** id

\* **@return**

\*/

**public** List<String> getPermissions(Long id) {

List<String> demo = **new** ArrayList<String>();

demo.add(String.*format*("Permission\_%d", id - 1));

demo.add(String.*format*("Permission\_%d", id));

demo.add(String.*format*("Permission\_%d", id + 1));

**return** demo;

}

}

3、applicationProvider.xml配置    路径：/dubbo-server/src/main/resources

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:dubbo=*"http://code.alibabatech.com/schema/dubbo"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://code.alibabatech.com/schema/dubbo*

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

*"*>

<!-- 具体的实现bean -->

<bean id=*"demoService"*

class=*"com.demo.server.service.impl.DemoTestImpl"* />

<!-- 提供方应用信息，用于计算依赖关系 -->

<dubbo:application name=*"demo\_provider"* />

<!-- 使用multicast广播注册中心暴露服务地址

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

<!-- 使用zookeeper注册中心暴露服务地址 -->

<dubbo:registry address=*"zookeeper://127.0.0.1:2181"* />

<!-- 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.demo.server.service.DemoTest"*

ref=*"demoService"* />

</beans>

4、启动服务代码

**package** com.demo.server;

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

**public** **class** App

{

**public** **static** **void** main( String[] args ) **throws** Exception

{

ClassPathXmlApplicationContext context=

**new** ClassPathXmlApplicationContext(**new** String[] {"applicationProvider.xml"});

context.start();

System.***out***.println("任意键退出");

System.***in***.read();

}

}

### dubbo-consumer消费者项目

1、pom.xml文件配置

<project xmlns=*"http://maven.apache.org/POM/4.0.0"* xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"* xsi:schemaLocation=*"http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd"*>

<modelVersion>4.0.0</modelVersion>

<groupId>dubbo-consumer</groupId>

<artifactId>dubbo-consumer</artifactId>

<version>0.0.1-SNAPSHOT</version>

<name>dubbo-consumer</name>

<url>http://maven.apache.org</url>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

</properties>

<dependencies>

<dependency>

<groupId>junit</groupId>

<artifactId>junit</artifactId>

<version>3.8.1</version>

<scope>test</scope>

</dependency>

<dependency>

<groupId>commons-logging</groupId>

<artifactId>commons-logging</artifactId>

<version>1.1.1</version>

</dependency>

<dependency>

<groupId>com.alibaba</groupId>

<artifactId>dubbo</artifactId>

<version>2.5.3</version>

</dependency>

<dependency>

<groupId>org.javassist</groupId>

<artifactId>javassist</artifactId>

<version>3.18.1-GA</version>

</dependency>

<dependency>

<groupId>log4j</groupId>

<artifactId>log4j</artifactId>

<version>1.2.15</version>

<exclusions>

<exclusion>

<groupId>com.sun.jdmk</groupId>

<artifactId>jmxtools</artifactId>

</exclusion>

<exclusion>

<groupId>com.sun.jmx</groupId>

<artifactId>jmxri</artifactId>

</exclusion>

<exclusion>

<artifactId>jms</artifactId>

<groupId>javax.jms</groupId>

</exclusion>

<exclusion>

<artifactId>mail</artifactId>

<groupId>javax.mail</groupId>

</exclusion>

</exclusions>

</dependency>

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring</artifactId>

<version>2.5.6.SEC03</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-api</artifactId>

<version>1.7.6</version>

</dependency>

<dependency>

<groupId>org.slf4j</groupId>

<artifactId>slf4j-log4j12</artifactId>

<version>1.6.1</version>

</dependency>

<dependency>

<groupId>com.github.adyliu</groupId>

<artifactId>zkclient</artifactId>

<version>2.0</version>

</dependency>

<dependency>

<groupId>org.apache.zookeeper</groupId>

<artifactId>zookeeper</artifactId>

<version>3.4.5</version>

<type>pom</type>

</dependency>

<dependency>

<groupId>dubbo-server</groupId>

<artifactId>dubbo-server</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

</dependencies>

</project>

2、消费者service代码

**package** com.demo.consumer;

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

**import** com.demo.server.service.DemoTest;

**public** **class** ConsumerThd {

**public** **void** sayHello(){

ClassPathXmlApplicationContext context=**new** ClassPathXmlApplicationContext(

**new** String[] {"applicationConsumer.xml"});

context.start();

DemoTest demoService=(DemoTest) context.getBean("demoService");

System.***out***.println(demoService.hello("world"));

}

}

3、消费者applicationConsumer.xml配置

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xmlns:dubbo=*"http://code.alibabatech.com/schema/dubbo"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans.xsd*

*http://code.alibabatech.com/schema/dubbo*

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

*"*>

<!-- consumer application name -->

<dubbo:application name=*"consumer-of-helloworld-app"* />

<!-- registry address, used for consumer to discover services -->

<dubbo:registry address=*"zookeeper://127.0.0.1:2181"* />

<dubbo:consumer timeout=*"5000"* />

<!-- which service to consume? -->

<dubbo:reference id=*"demoService"*

interface=*"com.demo.server.service.DemoTest"* />

</beans>

4、消费服务启动

**package** com.test;

**import** com.demo.consumer.ConsumerThd;

**public** **class** AppTest {

**public** **static** **void** main(String[] args) {

ConsumerThd thd=**new** ConsumerThd();

thd.sayHello();

}

}

### [测试](http://lib.csdn.net/base/softwaretest)

dubbo-provider启动：

1. log4j:WARN No appenders could be found **for** logger (org.springframework.context.support.ClassPathXmlApplicationContext).
2. log4j:WARN Please initialize the log4j system properly.
3. 任意键退出

dubbo-consumer测试启动：

1. log4j:WARN No appenders could be found **for** logger (org.springframework.context.support.ClassPathXmlApplicationContext).
2. log4j:WARN Please initialize the log4j system properly.
3. hello world

运行项目，先确保provider已被运行后再启动consumer模块。找不到provider时会报错：（No provider available for the service com.demo.server.service.DemoTest from the url zookeeper://127.0.0.1:2181）

当然，这只是一个模拟的项目，实际中有多提供者多消费者情况，比这要复杂的多，当然只有这样才能体现dubbo的特性。

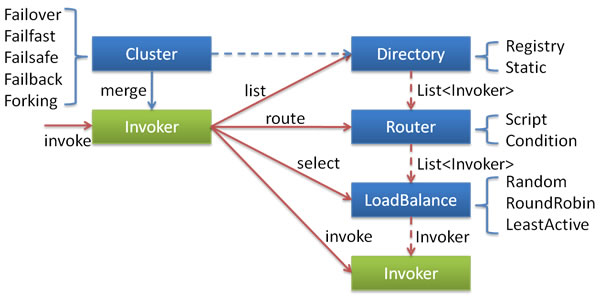
# Dubbo进阶

## 集群容错和负载均衡

### 集群容错

假设我们使用的是单机模式的Dubbo服务，如果在服务提供方（Provider）发布服务以后，服务消费方（Consumer）发出一次调用请求，恰好这次由于网络问题调用失败，那么我们可以配置服务消费方重试策略，可能消费方第二次重试调用是成功的（重试策略只需要配置即可，重试过程是透明的）；但是，如果服务提供方发布服务所在的节点发生故障，那么消费方再怎么重试调用都是失败的，所以我们需要采用集群容错模式，这样如果单个服务节点因故障无法提供服务，还可以根据配置的集群容错模式，调用其他可用的服务节点，这就提高了服务的可用性。

首先，根据Dubbo文档，我们引用文档提供的一个架构图以及各组件关系说明，如下所示：



上述各个组件之间的关系（引自Dubbo文档）说明如下：

* 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。

我们也简单说明目前Dubbo支持的集群容错模式，每种模式适应特定的应用场景，可以根据实际需要进行选择。Dubbo内置支持如下6种集群模式：

* **Failover Cluster模式**

配置值为failover。这种模式是Dubbo集群容错默认的模式选择，调用失败时，会自动切换，重新尝试调用其他节点上可用的服务。对于一些幂等性操作可以使用该模式，如读操作，因为每次调用的副作用是相同的，所以可以选择自动切换并重试调用，对调用者完全透明。可以看到，如果重试调用必然会带来响应端的延迟，如果出现大量的重试调用，可能说明我们的服务提供方发布的服务有问题，如网络延迟严重、硬件设备需要升级、程序算法非常耗时，等等，这就需要仔细检测排查了。

例如，可以这样显式指定Failover模式，或者不配置则默认开启Failover模式，配置示例如下：

<dubbo:service interface=*"org.shirdrn.dubbo.api.ChatRoomOnlineUserCounterService"* version=*"1.0.0"*

cluster=*"failover"* retries=*"2"* timeout=*"100"* ref=*"chatRoomOnlineUserCounterService"* protocol=*"dubbo"* >

<dubbo:method name=*"queryRoomUserCount"* timeout=*"80"* retries=*"2"* />

</dubbo:service>

上述配置使用Failover Cluster模式，如果调用失败一次，可以再次重试2次调用，服务级别调用超时时间为100ms，调用方法queryRoomUserCount的超时时间为80ms，允许重试2次，最坏情况调用花费时间160ms。如果该服务接口org.shirdrn.dubbo.api.ChatRoomOnlineUserCounterService还有其他的方法可供调用，则其他方法没有显式配置则会继承使用dubbo:service配置的属性值。

* **Failfast Cluster模式**

配置值为failfast。这种模式称为快速失败模式，调用只执行一次，失败则立即报错。这种模式适用于非幂等性操作，每次调用的副作用是不同的，如写操作，比如交易系统我们要下订单，如果一次失败就应该让它失败，通常由服务消费方控制是否重新发起下订单操作请求（另一个新的订单）。

* **Failsafe Cluster模式**

配置值为failsafe。失败安全模式，如果调用失败， 则直接忽略失败的调用，而是要记录下失败的调用到日志文件，以便后续审计。

* **Failback Cluster模式**

配置值为failback。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模式**

配置值为forking。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。

* **Broadcast Cluster模式**

配置值为broadcast。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。

上面的6种模式都可以应用于生产环境，我们可以根据实际应用场景选择合适的集群容错模式。如果我们觉得Dubbo内置提供的几种集群容错模式都不能满足应用需要，也可以定制实现自己的集群容错模式，因为Dubbo框架给我提供的扩展的接口，只需要实现接口com.alibaba.dubbo.rpc.cluster.Cluster即可，接口定义如下所示：

@SPI(FailoverCluster.NAME)

**public** **interface** Cluster {

/\*\*

\* Merge the directory invokers to a virtual invoker.

\* **@param** <T>

\* **@param** directory

\* **@return** cluster invoker

\* **@throws** RpcException

\*/

@Adaptive

<T> Invoker<T> join(Directory<T> directory) **throws** RpcException;

}

关于如何实现一个自定义的集群容错模式，可以参考Dubbo源码中内置支持的汲取你容错模式的实现，6种模式对应的实现类如下所示：

com.alibaba.dubbo.rpc.cluster.support.FailoverCluster

com.alibaba.dubbo.rpc.cluster.support.FailfastCluster

com.alibaba.dubbo.rpc.cluster.support.FailsafeCluster

com.alibaba.dubbo.rpc.cluster.support.FailbackCluster

com.alibaba.dubbo.rpc.cluster.support.ForkingCluster

com.alibaba.dubbo.rpc.cluster.support.AvailableCluster

### 负载均衡

Dubbo框架内置提供负载均衡的功能以及扩展接口，我们可以透明地扩展一个服务或服务集群，根据需要非常容易地增加/移除节点，提高服务的可伸缩性。Dubbo框架内置提供了4种负载均衡策略，如下所示：

* Random LoadBalance：随机策略，配置值为random。可以设置权重，有利于充分利用服务器的资源，高配的可以设置权重大一些，低配的可以稍微小一些
* RoundRobin LoadBalance：轮询策略，配置值为roundrobin。
* LeastActive LoadBalance：配置值为leastactive。根据请求调用的次数计数，处理请求更慢的节点会受到更少的请求
* ConsistentHash LoadBalance：一致性Hash策略，具体配置方法可以参考Dubbo文档。相同调用参数的请求会发送到同一个服务提供方节点上，如果某个节点发生故障无法提供服务，则会基于一致性Hash算法映射到虚拟节点上（其他服务提供方）

在实际使用中，只需要选择合适的负载均衡策略值，配置即可，下面是上述四种负载均衡策略配置的示例：

<dubbo:service interface=*"org.shirdrn.dubbo.api.ChatRoomOnlineUserCounterService"* version=*"1.0.0"*

cluster=*"failover"* retries=*"2"* timeout=*"100"* loadbalance=*"random"*

ref=*"chatRoomOnlineUserCounterService"* protocol=*"dubbo"* >

<dubbo:method name=*"queryRoomUserCount"* timeout=*"80"* retries=*"2"* loadbalance=*"leastactive"* />

</dubbo:service>

上述配置，也体现了Dubbo配置的继承性特点，也就是dubbo:service元素配置了loadbalance=”random”，则该元素的子元素dubbo:method如果没有指定负载均衡策略，则默认为loadbalance=”random”，否则如果dubbo:method指定了loadbalance=”leastactive”，则使用子元素配置的负载均衡策略覆盖了父元素指定的策略（这里调用queryRoomUserCount方法使用leastactive负载均衡策略）。

当然，Dubbo框架也提供了实现自定义负载均衡策略的接口，可以实现com.alibaba.dubbo.rpc.cluster.LoadBalance接口，接口定义如下所示：

/\*\*

\* LoadBalance. (SPI, Singleton, ThreadSafe)

\*

\* <a href="http://en.wikipedia.org/wiki/Load\_balancing\_(computing)">Load-Balancing</a>

\*

\* **@see** com.alibaba.dubbo.rpc.cluster.Cluster#join(Directory)

\* **@author** qian.lei

\* **@author** william.liangf

\*/

@SPI(RandomLoadBalance.NAME)

**public** **interface** LoadBalance {

/\*\*

\* select one invoker in list.

\* **@param** invokers invokers.

\* **@param** url refer url

\* **@param** invocation invocation.

\* **@return** selected invoker.

\*/

@Adaptive("loadbalance")

<T> Invoker<T> select(List<Invoker<T>> invokers, URL url, Invocation invocation) **throws** RpcException;

}

如何实现一个自定义负载均衡策略，可以参考Dubbo框架内置的实现，如下所示的3个实现类：

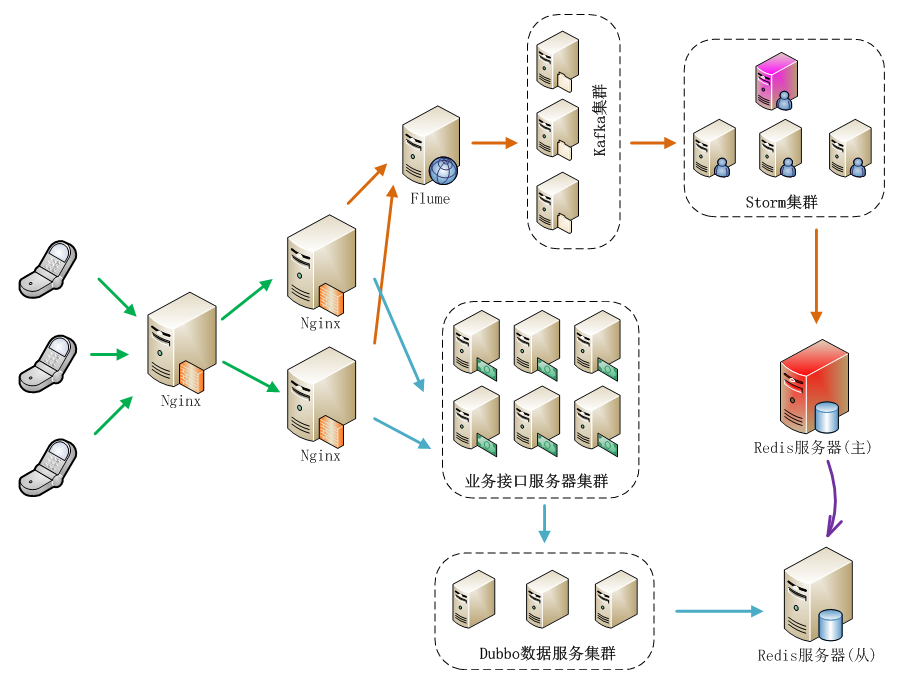
com.alibaba.dubbo.rpc.cluster.loadbalance.RandomLoadBalance

com.alibaba.dubbo.rpc.cluster.loadbalance.RoundRobinLoadBalance

com.alibaba.dubbo.rpc.cluster.loadbalance.LeastActiveLoadBalance

## Dubbo服务集群容错实践

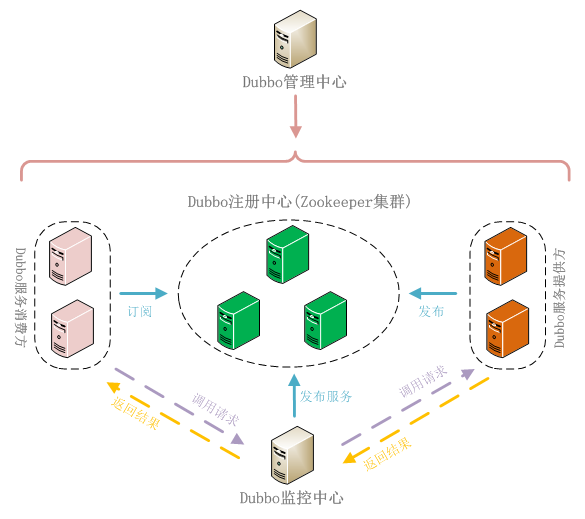
手机应用是以聊天室为基础的，我们需要收集用户的操作行为，然后计算聊天室中在线人数，并实时在手机应用端显示人数，整个系统的架构如图所示：



上图中，主要包括了两大主要流程：日志收集并实时处理流程、调用读取实时计算结果流程，我们使用基于Dubbo框架开发的服务来提供实时计算结果读取聊天人数的功能。上图中，实际上业务接口服务器集群也可以基于Dubbo框架构建服务，就看我们想要构建什么样的系统来满足我们的需要。

如果不使用注册中心，服务消费方也能够直接调用服务提供方发布的服务，这样需要服务提供方将服务地址暴露给服务消费方，而且也无法使用监控中心的功能，这种方式成为直连。

如果我们使用注册中心，服务提供方将服务发布到注册中心，而服务消费方可以通过注册中心订阅服务，接收服务提供方服务变更通知，这种方式可以隐藏服务提供方的细节，包括服务器地址等敏感信息，而服务消费方只能通过注册中心来获取到已注册的提供方服务，而不能直接跨过注册中心与服务提供方直接连接。这种方式的好处是还可以使用监控中心服务，能够对服务的调用情况进行监控分析，还能使用Dubbo服务管理中心，方便管理服务，我们在这里使用的是这种方式，也推荐使用这种方式。使用注册中心的Dubbo分布式服务相关组件结构，如下图所示：



下面，开发部署我们的应用，通过如下4个步骤来完成：

* **服务接口定义**

服务接口将服务提供方（Provider）和服务消费方（Consumer）连接起来，服务提供方实现接口中定义的服务，即给出服务的实现，而服务消费方负责调用服务。我们接口中给出了2个方法，一个是实时查询获取当前聊天室内人数，另一个是查询一天中某个/某些聊天室中在线人数峰值，接口定义如下所示：

**package** org.shirdrn.dubbo.api;

**import** java.util.List;

**public** **interface** ChatRoomOnlineUserCounterService {

String queryRoomUserCount(String rooms);

List<String> getMaxOnlineUserCount(List<String> rooms, String date, String dateFormat);

}

接口是服务提供方和服务消费方公共遵守的协议，一般情况下是服务提供方将接口定义好后提供给服务消费方。

* **服务提供方**

服务提供方实现接口中定义的服务，其实现和普通的服务没什么区别，我们的实现类为ChatRoomOnlineUserCounterServiceImpl，代码如下所示：

**package** org.shirdrn.dubbo.provider.service;

**import** java.util.List;

**import** org.apache.commons.logging.Log;

**import** org.apache.commons.logging.LogFactory;

**import** org.shirdrn.dubbo.api.ChatRoomOnlineUserCounterService;

**import** org.shirdrn.dubbo.common.utils.DateTimeUtils;

**import** redis.clients.jedis.Jedis;

**import** redis.clients.jedis.JedisPool;

**import** com.alibaba.dubbo.common.utils.StringUtils;

**import** com.google.common.base.Strings;

**import** com.google.common.collect.Lists;

**public** **class** ChatRoomOnlineUserCounterServiceImpl **implements** ChatRoomOnlineUserCounterService {

**private** **static** **final** Log LOG = LogFactory.getLog(ChatRoomOnlineUserCounterServiceImpl.**class**);

**private** JedisPool jedisPool;

**private** **static** **final** String KEY\_USER\_COUNT = "chat::room::play::user::cnt";

**private** **static** **final** String KEY\_MAX\_USER\_COUNT\_PREFIX = "chat::room::max::user::cnt::";

**private** **static** **final** String DF\_YYYYMMDD = "yyyyMMdd";

**public** String queryRoomUserCount(String rooms) {

LOG.info("Params[Server|Recv|REQ] rooms=" + rooms);

StringBuffer builder = **new** StringBuffer();

**if**(!Strings.isNullOrEmpty(rooms)) {

Jedis jedis = **null**;

**try** {

jedis = jedisPool.getResource();

String[] fields = rooms.split(",");

List<String> results = jedis.hmget(KEY\_USER\_COUNT, fields);

builder.append(StringUtils.join(results, ","));

} **catch** (Exception e) {

LOG.error("", e);

} **finally** {

**if**(jedis != **null**) {

jedis.close();

}

}

}

LOG.info("Result[Server|Recv|RES] " + builder.toString());

**return** builder.toString();

}

@Override

**public** List<String> getMaxOnlineUserCount(List<String> rooms, String date, String dateFormat) {

// HGETALL chat::room::max::user::cnt::20150326

LOG.info("Params[Server|Recv|REQ] rooms=" + rooms + ",date=" + date + ",dateFormat=" + dateFormat);

String whichDate = DateTimeUtils.format(date, dateFormat, DF\_YYYYMMDD);

String key = KEY\_MAX\_USER\_COUNT\_PREFIX + whichDate;

StringBuffer builder = **new** StringBuffer();

**if**(rooms != **null** && !rooms.isEmpty()) {

Jedis jedis = **null**;

**try** {

jedis = jedisPool.getResource();

**return** jedis.hmget(key, rooms.toArray(**new** String[rooms.size()]));

} **catch** (Exception e) {

LOG.error("", e);

} **finally** {

**if**(jedis != **null**) {

jedis.close();

}

}

}

LOG.info("Result[Server|Recv|RES] " + builder.toString());

**return** Lists.newArrayList();

}

**public** **void** setJedisPool(JedisPool jedisPool) {

**this**.jedisPool = jedisPool;

}

}

代码中通过读取Redis中数据来完成调用，逻辑比较简单。对应的Maven POM依赖配置，如下所示：

<dependencies>

<dependency>

<groupId>org.shirdrn.dubbo</groupId>

<artifactId>dubbo-api</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

<dependency>

<groupId>org.shirdrn.dubbo</groupId>

<artifactId>dubbo-commons</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

<dependency>

<groupId>redis.clients</groupId>

<artifactId>jedis</artifactId>

<version>2.5.2</version>

</dependency>

<dependency>

<groupId>org.apache.commons</groupId>

<artifactId>commons-pool2</artifactId>

<version>2.2</version>

</dependency>

<dependency>

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

<artifactId>netty</artifactId>

<version>3.2.7.Final</version>

</dependency>

</dependencies>

有关对Dubbo框架的一些依赖，我们单独放到一个通用的Maven Module中（详见后面“附录：Dubbo使用Maven构建依赖配置”），这里不再多说。服务提供方实现，最关键的就是服务的配置，因为Dubbo基于Spring来管理配置和实例，所以通过配置可以指定服务是否是分布式服务，以及通过配置增加很多其它特性。我们的配置文件为provider-cluster.xml，内容如下所示：

<?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"*

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

xsi:schemaLocation=*"http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-3.0.xsd*

*http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd"*>

<bean class=*"org.springframework.beans.factory.config.PropertyPlaceholderConfigurer"*>

<property name=*"systemPropertiesModeName"* value=*"SYSTEM\_PROPERTIES\_MODE\_OVERRIDE"* />

<property name=*"ignoreResourceNotFound"* value=*"true"* />

<property name=*"locations"*>

<list>

<value>classpath\*:jedis.properties</value>

</list>

</property>

</bean>

<dubbo:application name=*"chatroom-cluster-provider"* />

<dubbo:registry address=*"zookeeper://zk1:2181?backup=zk2:2181,zk3:2181"* />

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

<dubbo:service interface=*"org.shirdrn.dubbo.api.ChatRoomOnlineUserCounterService"* version=*"1.0.0"*

cluster=*"failover"* retries=*"2"* timeout=*"1000"* loadbalance=*"random"* actives=*"100"* executes=*"200"*

ref=*"chatRoomOnlineUserCounterService"* protocol=*"dubbo"* >

<dubbo:method name=*"queryRoomUserCount"* timeout=*"500"* retries=*"2"* loadbalance=*"roundrobin"* actives=*"50"* />

</dubbo:service>

<bean id=*"chatRoomOnlineUserCounterService"* class=*"org.shirdrn.dubbo.provider.service.ChatRoomOnlineUserCounterServiceImpl"* >

<property name=*"jedisPool"* ref=*"jedisPool"* />

</bean>

<bean id=*"jedisPool"* class=*"redis.clients.jedis.JedisPool"* destroy-method=*"destroy"*>

<constructor-arg index=*"0"*>

<bean class=*"org.apache.commons.pool2.impl.GenericObjectPoolConfig"*>

<property name=*"maxTotal"* value=*"${redis.pool.maxTotal}"* />

<property name=*"maxIdle"* value=*"${redis.pool.maxIdle}"* />

<property name=*"minIdle"* value=*"${redis.pool.minIdle}"* />

<property name=*"maxWaitMillis"* value=*"${redis.pool.maxWaitMillis}"* />

<property name=*"testOnBorrow"* value=*"${redis.pool.testOnBorrow}"* />

<property name=*"testOnReturn"* value=*"${redis.pool.testOnReturn}"* />

<property name=*"testWhileIdle"* value=*"true"* />

</bean>

</constructor-arg>

<constructor-arg index=*"1"* value=*"${redis.host}"* />

<constructor-arg index=*"2"* value=*"${redis.port}"* />

<constructor-arg index=*"3"* value=*"${redis.timeout}"* />

</bean>

</beans>

上面配置中，使用dubbo协议，集群容错模式为failover，服务级别负载均衡策略为random，方法级别负载均衡策略为roundrobin（它覆盖了服务级别的配置内容），其他一些配置内容可以参考Dubbo文档。我们这里是从Redis读取数据，所以使用了Redis连接池。

启动服务示例代码如下所示：

**package** org.shirdrn.dubbo.provider;

**import** org.shirdrn.dubbo.provider.common.DubboServer;

**public** **class** ChatRoomClusterServer {

**public** **static** **void** main(String[] args) **throws** Exception {

DubboServer.startServer("classpath:provider-cluster.xml");

}

}

上面调用了DubboServer类的静态方法startServer，如下所示：

**public** **static** **void** startServer(String config) {

ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext(config);

**try** {

context.start();

System.in.read();

} **catch** (IOException e) {

e.printStackTrace();

} **finally** {

context.close();

}

}

方法中主要是初始化Spring IoC容器，全部对象都交由容器来管理。

* **服务消费方**

服务消费方就容易了，只需要知道注册中心地址，并引用服务提供方提供的接口，消费方调用服务实现如下所示：

**package** org.shirdrn.dubbo.consumer;

**import** java.util.Arrays;

**import** java.util.List;

**import** org.apache.commons.logging.Log;

**import** org.apache.commons.logging.LogFactory;

**import** org.shirdrn.dubbo.api.ChatRoomOnlineUserCounterService;

**import** org.springframework.context.support.AbstractXmlApplicationContext;

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

**public** **class** ChatRoomDubboConsumer {

**private** **static** **final** Log LOG = LogFactory.getLog(ChatRoomDubboConsumer.**class**);

**public** **static** **void** main(String[] args) **throws** Exception {

AbstractXmlApplicationContext context = **new** ClassPathXmlApplicationContext("classpath:consumer.xml");

**try** {

context.start();

ChatRoomOnlineUserCounterService chatRoomOnlineUserCounterService = (ChatRoomOnlineUserCounterService) context.getBean("chatRoomOnlineUserCounterService");

getMaxOnlineUserCount(chatRoomOnlineUserCounterService);

getRealtimeOnlineUserCount(chatRoomOnlineUserCounterService);

System.in.read();

} **finally** {

context.close();

}

}

**private** **static** **void** getMaxOnlineUserCount(ChatRoomOnlineUserCounterService liveRoomOnlineUserCountService) {

List<String> maxUserCounts = liveRoomOnlineUserCountService.getMaxOnlineUserCount(

Arrays.asList(**new** String[] {"1482178010" , "1408492761", "1430546839", "1412517075", "1435861734"}), "20150327", "yyyyMMdd");

LOG.info("After getMaxOnlineUserCount invoked: maxUserCounts= " + maxUserCounts);

}

**private** **static** **void** getRealtimeOnlineUserCount(ChatRoomOnlineUserCounterService liveRoomOnlineUserCountService)

**throws** InterruptedException {

String rooms = "1482178010,1408492761,1430546839,1412517075,1435861734";

String onlineUserCounts = liveRoomOnlineUserCountService.queryRoomUserCount(rooms);

LOG.info("After queryRoomUserCount invoked: onlineUserCounts= " + onlineUserCounts);

}

}

对应的配置文件为consumer.xml，内容如下所示：

<?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-3.0.xsd*

*http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd"*>

<dubbo:application name=*"chatroom-consumer"* />

<dubbo:registry address=*"zookeeper://zk1:2181?backup=zk2:2181,zk3:2181"* />

<dubbo:reference id=*"chatRoomOnlineUserCounterService"* interface=*"org.shirdrn.dubbo.api.ChatRoomOnlineUserCounterService"* version=*"1.0.0"*>

<dubbo:method name=*"queryRoomUserCount"* retries=*"2"* />

</dubbo:reference>

</beans>

也可以根据需要配置dubbo:reference相关的属性值，也可以配置dubbo:method指定调用的方法的配置信息，详细配置属性可以参考Dubbo官方文档。

* **部署与验证**

开发完成提供方服务后，在本地开发调试的时候可以怎么简单怎么做，如果是要部署到生产环境，则需要打包后进行部署，可以参考下面的Maven POM配置：

<build>

<plugins>

<plugin>

<groupId>org.apache.maven.plugins</groupId>

<artifactId>maven-shade-plugin</artifactId>

<version>1.4</version>

<configuration>

<createDependencyReducedPom>true</createDependencyReducedPom>

</configuration>

<executions>

<execution>

<phase>package</phase>

<goals>

<goal>shade</goal>

</goals>

<configuration>

<transformers>

<transformer implementation=*"org.apache.maven.plugins.shade.resource.ServicesResourceTransformer"* />

<transformer implementation=*"org.apache.maven.plugins.shade.resource.ManifestResourceTransformer"*>

<mainClass>org.shirdrn.dubbo.provider.ChatRoomClusterServer</mainClass>

</transformer>

</transformers>

</configuration>

</execution>

</executions>

</plugin>

</plugins>

</build>

这里也给出Maven POM依赖的简单配置：

<dependencies>

<dependency>

<groupId>org.shirdrn.dubbo</groupId>

<artifactId>dubbo-api</artifactId>

<version>0.0.1-SNAPSHOT</version>

</dependency>

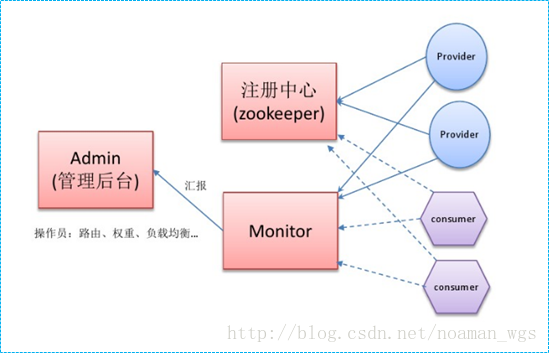
</dependencies>

我们开发的服务应该是分布式的，首先是通过配置内容来决定，例如设置集群模式、设置负载均衡模式等，然后在部署的时候，可以在多个节点上同一个服务，这样多个服务都会注册到Dubbo注册中心，如果某个节点上的服务不可用了，可以根据我们配置的策略来选择其他节点上的可用服务，后面通过Dubbo服务管理中心和监控中心就能更加清楚明了。

## Dubbo管理控制台介绍

管理控制台功能

* 路由规则，动态配置，服务降级
* 访问控制，权重调整
* 负载均衡



安装Dubbo服务管理中心，需要选择一个Web容器，我们使用Tomcat服务器。首先下载Dubbo管理中心安装文件dubbo-admin-2.5.3.war，或者直接从源码构建得到该WAR文件。这里，我们已经构建好对应的WAR文件，然后进行安装，执行如下命令：、

**cd apache-tomcat-6.0.35**

**rm -rf webapps/ROOT**

**unzip ~/dubbo-admin-2.5.3.war -d webapps/ROOT**

修改配置文件~/apache-tomcat-6.0.35/webapps/ROOT/WEB-INF/dubbo.properties，指定我们的注册中心地址以及登录密码，内容如下所示：

**dubbo.registry.address=zookeeper://zk1:2181?backup=zk2:2181,zk3:2181**

**dubbo.admin.root.password=root**

**dubbo.admin.guest.password=guest**

然后，根据需要修改~/apache-tomcat-6.0.35/conf/server.xml配置文件，主要是Tomcat HTTP 端口号（我这里使用8083端口），完成后可以直接启动Tomcat服务器：

**cd ~/apache-tomcat-6.0.35/**

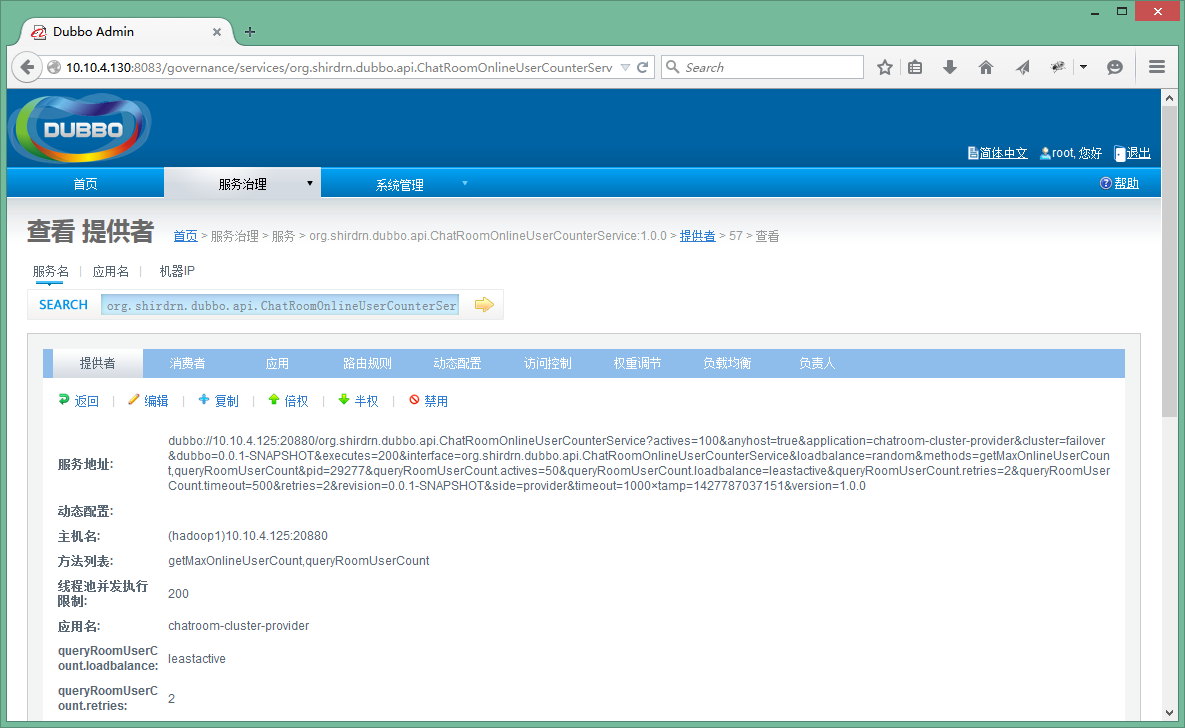
**bin/catalina.sh start**

然后访问地址<http://10.10.4.130:8083/>即可，根据配置文件指定的root用户密码，就可以登录Dubbo管理控制台。

我们将上面开发的服务提供方服务，部署到2个独立的节点上（192.168.14.1和10.10.4.125），然后可以通过Dubbo管理中心查看对应服务的状况，如图所示：



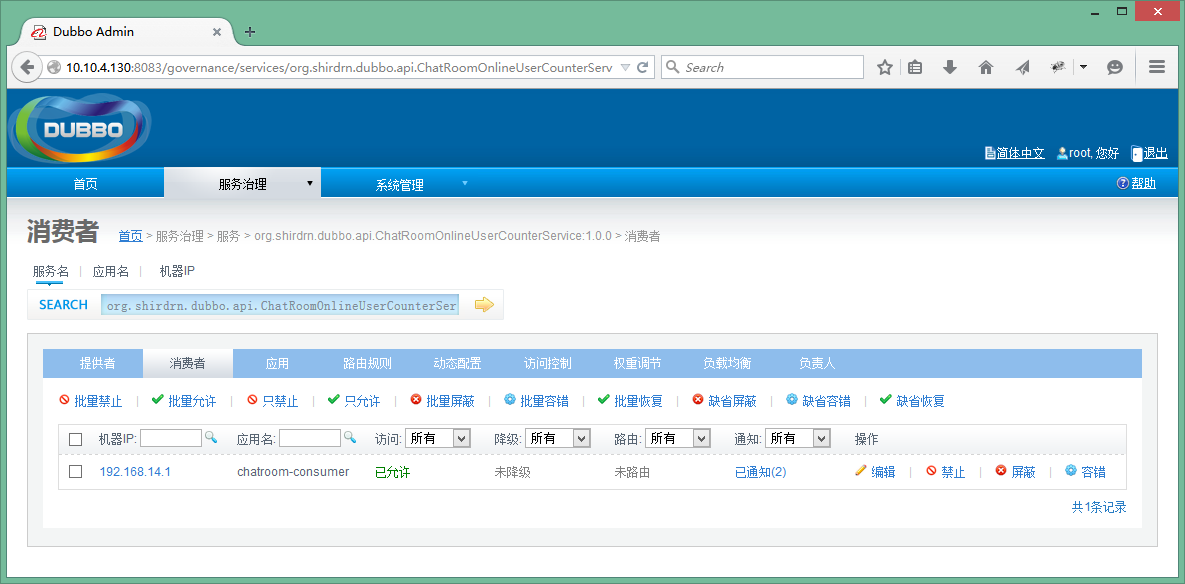
上图中可以看出，该服务有两个独立的节点可以提供，因为配置的集群模式为failover，如果某个节点的服务发生故障无法使用，则会自动透明地重试另一个节点上的服务，这样就不至于出现拒绝服务的情况。如果想要查看提供方某个节点上的服务详情，可以点击对应的IP:Port链接，示例如图所示：



上图可以看到服务地址：

|  |  |
| --- | --- |
| 1 | <dubbo://10.10.4.125:20880/org.shirdrn.dubbo.api.ChatRoomOnlineUserCounterService?actives=100&anyhost=true&application=chatroom-cluster-provider&cluster=failover&dubbo=0.0.1-SNAPSHOT&executes=200&interface=org.shirdrn.dubbo.api.ChatRoomOnlineUserCounterService&loadbalance=random&methods=getMaxOnlineUserCount>,queryRoomUserCount&pid=30942&queryRoomUserCount.actives=50&queryRoomUserCount.loadbalance=leastactive&queryRoomUserCount.retries=2&queryRoomUserCount.timeout=500&retries=2&revision=0.0.1-SNAPSHOT&side=provider&timeout=1000×tamp=1427793652814&version=1.0.0 |

如果我们直接暴露该地址也是可以的，不过这种直连的方式对服务消费方不是透明的，如果以后IP地址更换，也会影响调用方，所以最好是通过注册中心来隐蔽服务地址。同一个服务所部署在的多个节点上，也就对应对应着多个服务地址。另外，也可以对已经发布的服务进行控制，如修改访问控制、负载均衡相关配置内容等，可以通过上图中“消费者”查看服务消费方调用服务的情况，如图所示：



也在管理控制台可以对消费方进行管理控制。

* Dubbo监控中心

Dubbo监控中心是以Dubbo服务的形式发布到注册中心，和普通的服务时一样的。例如，我这里下载了Dubbo自带的简易监控中心文件dubbo-monitor-simple-2.5.3-assembly.tar.gz，可以解压缩以后，修改配置文件~/dubbo-monitor-simple-2.5.3/conf/dubbo.properties的内容，如下所示：

**dubbo.container=log4j,spring,registry,jetty**

**dubbo.application.name=simple-monitor**

**dubbo.application.owner=**

**dubbo.registry.address=zookeeper://zk1:2181?backup=zk2:2181,zk3:2181**

**dubbo.protocol.port=7070**

**dubbo.jetty.port=8087**

**dubbo.jetty.directory=${user.home}/monitor**

**dubbo.charts.directory=${dubbo.jetty.directory}/charts**

**dubbo.statistics.directory=${user.home}/monitor/statistics**

**dubbo.log4j.file=logs/dubbo-monitor-simple.log**

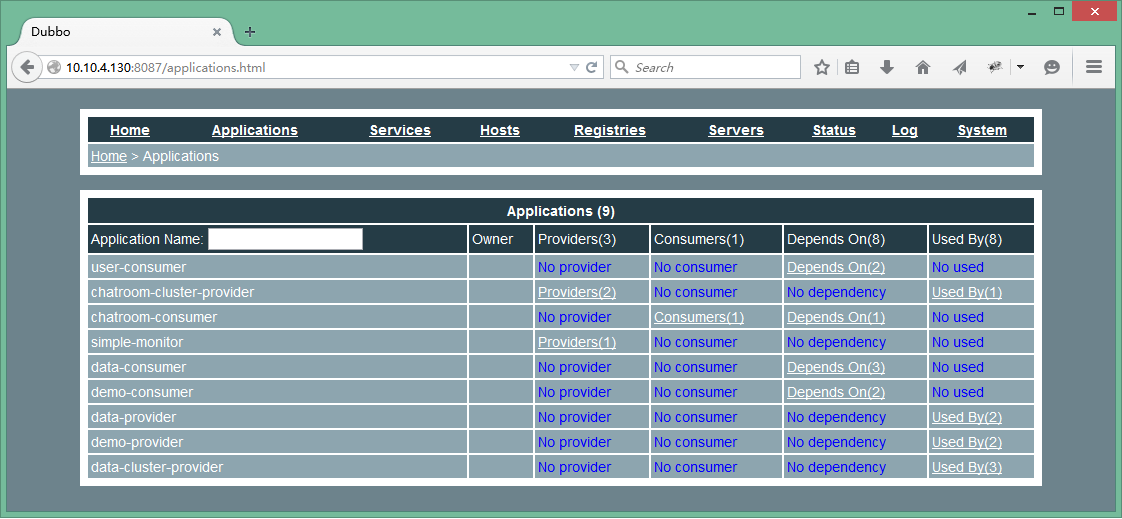
**dubbo.log4j.level=WARN**

然后启动简易监控中心，执行如下命令：

**cd ~/dubbo-monitor-simple-2.5.3**

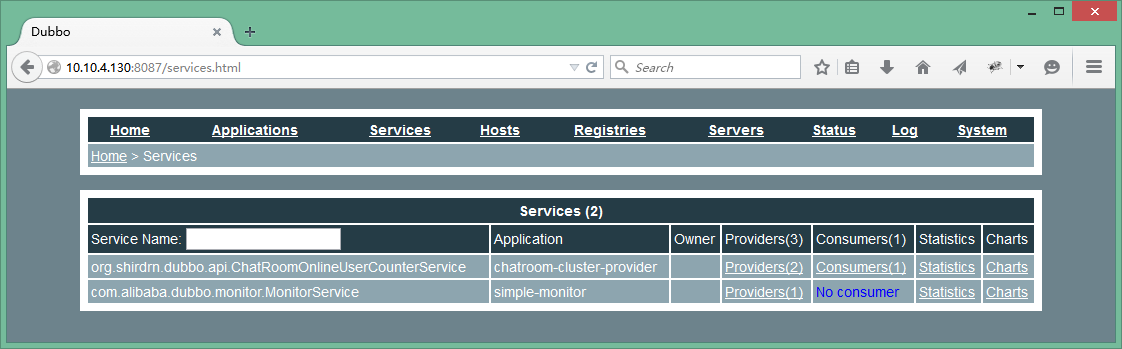
**bin/start.sh**

这里使用了Jetty Web容器，访问地址<http://10.10.4.130:8087/>就可以查看监控中心，Applications选项卡页面包含了服务提供方和消费方的基本信息，如图所示：

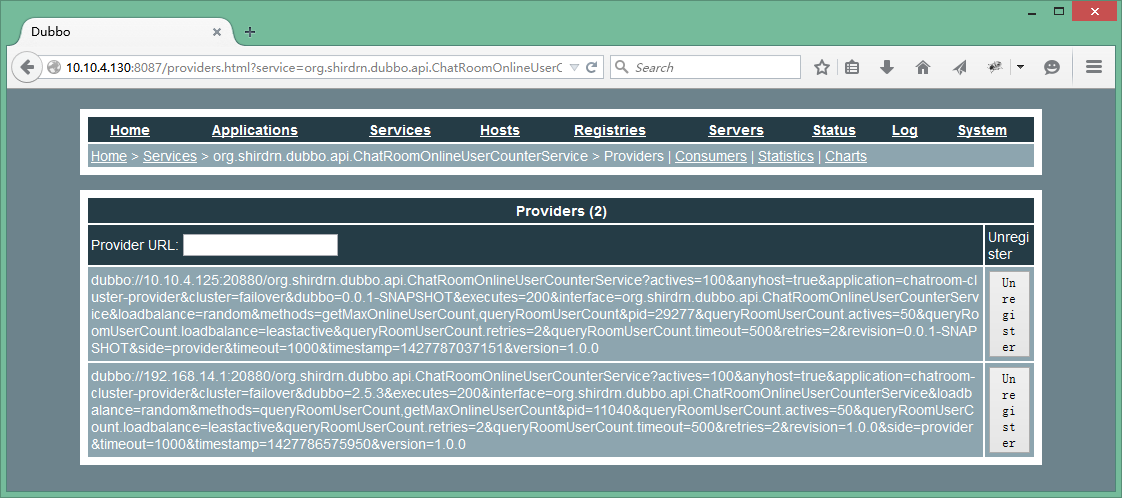


上图主要列出了所有提供方发布的服务、消费方调用、服务依赖关系等内容。

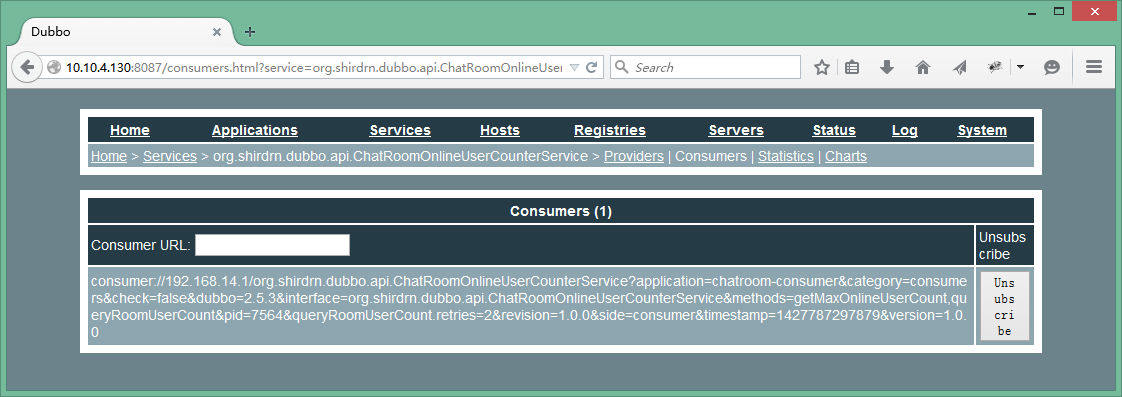
接着，查看Services选项卡页面，包含了服务提供方提供的服务列表，如图所示：



点击上图中Providers链接就能看到服务提供方的基本信息，包括服务地址等，如图所示：



点击上图中Consumers链接就能看到服务消费方的基本信息，包括服务地址等，如图所示：



由于上面是Dubbo自带的一个简易监控中心，可能所展现的内容并不能满足我们的需要，所以可以根据需要开发自己的监控中心。Dubbo也提供了监控中心的扩展接口，如果想要实现自己的监控中心，可以实现接口com.alibaba.dubbo.monitor.MonitorFactory和com.alibaba.dubbo.monitor.Monitor，其中MonitorFactory接口定义如下所示：

/\*\*

\* MonitorFactory. (SPI, Singleton, ThreadSafe)

\*

\* **@author** william.liangf

\*/

@SPI("dubbo")

**public** **interface** MonitorFactory {

/\*\*

\* Create monitor.

\* **@param** url

\* **@return** monitor

\*/

@Adaptive("protocol")

Monitor getMonitor(URL url);

}

Monitor接口定义如下所示：

/\*\*

\* Monitor. (SPI, Prototype, ThreadSafe)

\*

\* **@see** com.alibaba.dubbo.monitor.MonitorFactory#getMonitor(com.alibaba.dubbo.common.URL)

\* **@author** william.liangf

\*/

**public** **interface** Monitor **extends** Node, MonitorService {

}

具体定义内容可以查看MonitorService接口，不再累述。