# Dubbo

## dubbo工作原理

第一层：service层，接口层，给服务提供者和消费者来实现的

第二层：config层，配置层，主要是对dubbo进行各种配置的

第三层：proxy层，服务代理层，透明生成客户端的stub和服务单的skeleton

第四层：registry层，服务注册层，负责服务的注册与发现

第五层：cluster层，集群层，封装多个服务提供者的路由以及负载均衡，将多个实例组合成一个服务

第六层：monitor层，监控层，对rpc接口的调用次数和调用时间进行监控

第七层：protocol层，远程调用层，封装rpc调用

第八层：exchange层，信息交换层，封装请求响应模式，同步转异步

第九层：transport层，网络传输层，抽象mina和netty为统一接口

第十层：serialize层，数据序列化层

### 1.1 工作流程

1）第一步，provider向注册中心去注册

2）第二步，consumer从注册中心订阅服务，注册中心会通知consumer注册好的服务

3）第三步，consumer调用provider

4）第四步，consumer和provider都异步的通知监控中心

## 2. dubbo支持的协议

### 2.1 dubbo协议

单一长连接，NIO异步通信，基于hessian作为序列化协议

适用场景：传输数据量很小（每次请求在100kb以内），但是并发量很高

为了要支持高并发场景，一般是服务提供者就几台机器，但是服务消费者有上百台，可能每天调用量达到上亿次！此时用长连接是最合适的，就是跟每个服务消费者维持一个长连接就可以，可能总共就100个连接。然后后面直接基于长连接NIO异步通信，可以支撑高并发请求。

否则如果上亿次请求每次都是短连接的话，服务提供者会扛不住。

而且因为走的是单一长连接，所以传输数据量太大的话，会导致并发能力降低。所以一般建议是传输数据量很小，支撑高并发访问。

## 2.2 hessian协议

走hessian序列化协议，多个短连接，适用于提供者数量比消费者数量还多，适用于文件的传输，一般较少用

## 2.3 http协议

走json序列化

## 2.4 webservice

走SOAP文本序列化

## Dubbo负载均衡及动态代理的策略

### 3.1 负载均衡策略

1. random loadbalance

dubbo默认采用random load balance,即随机调用实现负载均衡，可以对provider不同实例设置不同的权重，会按照权重来负载均衡，权重越大分配流量越高，一般就用默认的即可.

1. roundrobin loadbalance

均匀地将流量打到各个机器上去，但如果各个机器的性能不一样，容易导致性能差的机器负载过高。所以此时需要调整权重，让性能差的机器承载权重小一些，流量少一些。

1. leastactive loadbalance

自动感知一下，如果某个机器性能越差，那么接收的请求越少，越不活跃，此时就会给不活跃的性能差的机器更少的请求

1. consistanthash loadbalance

一致性Hash算法，相同参数的请求一定分发到一个provider上去，provider挂掉的时候，会基于虚拟节点均匀分配剩余的流量，抖动不会太大

如果你需要的不是随机负载均衡，是要一类请求都到一个节点，那就走这个一致性hash策略

### 3.2 集群容错策略

1 failover cluster模式

默认,失败自动切换，自动重试其他机器,常用于读操作

2 failfast cluster模式

一次调用失败就立即失败，常用于写操作

3 failsafe cluster模式

发生异常时忽略掉，常用于不重要的接口调用，如记录日志

4 failbackc cluster模式

失败了后台自动记录请求，然后定时重发，适于写消息队列

5 forking cluster

并行调用多个provider，只要一个成功就立即返回

6 broadcacst cluster

逐个调用所有的provider

## 动态代理策略

默认使用javassist动态字节码生成，创建代理类

但是可以通过spi扩展机制配置自己的动态代理策略

## Dubbo spi机制

SPI 全称为 Service Provider Interface，是一种服务发现机制.

# spring如何加载生成dubbo对象

利用spring提供的可扩展的schema生成。

分为几步：

1. 编写xsd配置文件
2. 编写NamespaceHandler和BeanDefinitionParser完成解析工作
3. 编写spring.handlers和spring.schemas串联起所有部件
4. 在Bean文件中应用

第一步，可扩展就是利用xsd配置文件来进行自定义配置。

第二步，会用到NamespaceHandler和BeanDefinitionParser这两个概念。具体说来NamespaceHandler会根据schema和节点名找到某个BeanDefinitionParser，然后由BeanDefinitionParser完成具体的解析工作。

其中registerBeanDefinitionParser("people", new PeopleBeanDefinitionParser());就是用来把节点名和解析类联系起来，在配置中引用people配置项时，就会用PeopleBeanDefinitionParser来解析配置

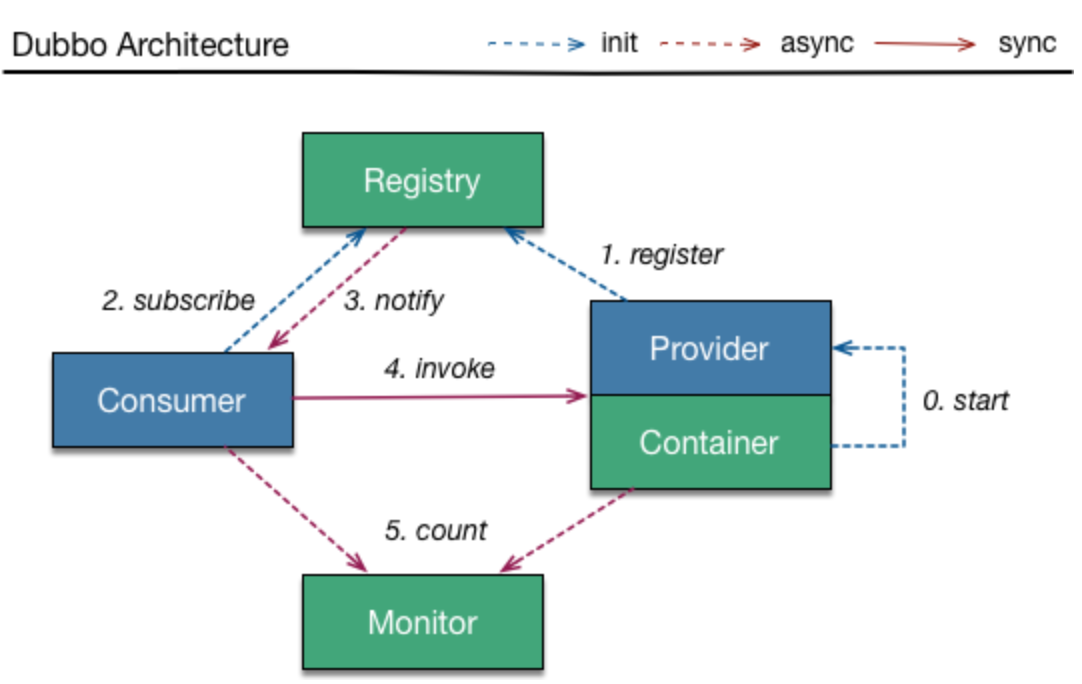
第三步，就是让spring感知到这些配置文件，把这些加载进来的对象交给spring管理。spring提供了spring.handlers和spring.schemas这两个配置文件来完成这项工作。

Spring.handlers里面填写的是handlers的信息。

Spring.schemas里面填写的就是配置文件。

第四部，最后就是正常的使用了。

# Dubbo注册与发现



Provider：暴露服务的服务提供方

Consumer：调用远程服务的服务消费方

Registry：服务注册与发现的注册中心

Monitor：统计服务的调用次调和调用时间的监控中心

Container：服务运行容器

调用关系说明：

1.  服务容器负责启动,加载,运行服务提供者。

2.  服务提供者在启动时,向注册中心注册自己提供的服务。

3.  服务消费者在启动时,向注册中心订阅自己所需的服务。

4.  注册中心返回服务提供者地址列表给消费者,如果有变更,注册中心将基于长连接推送变更数据给消费者。

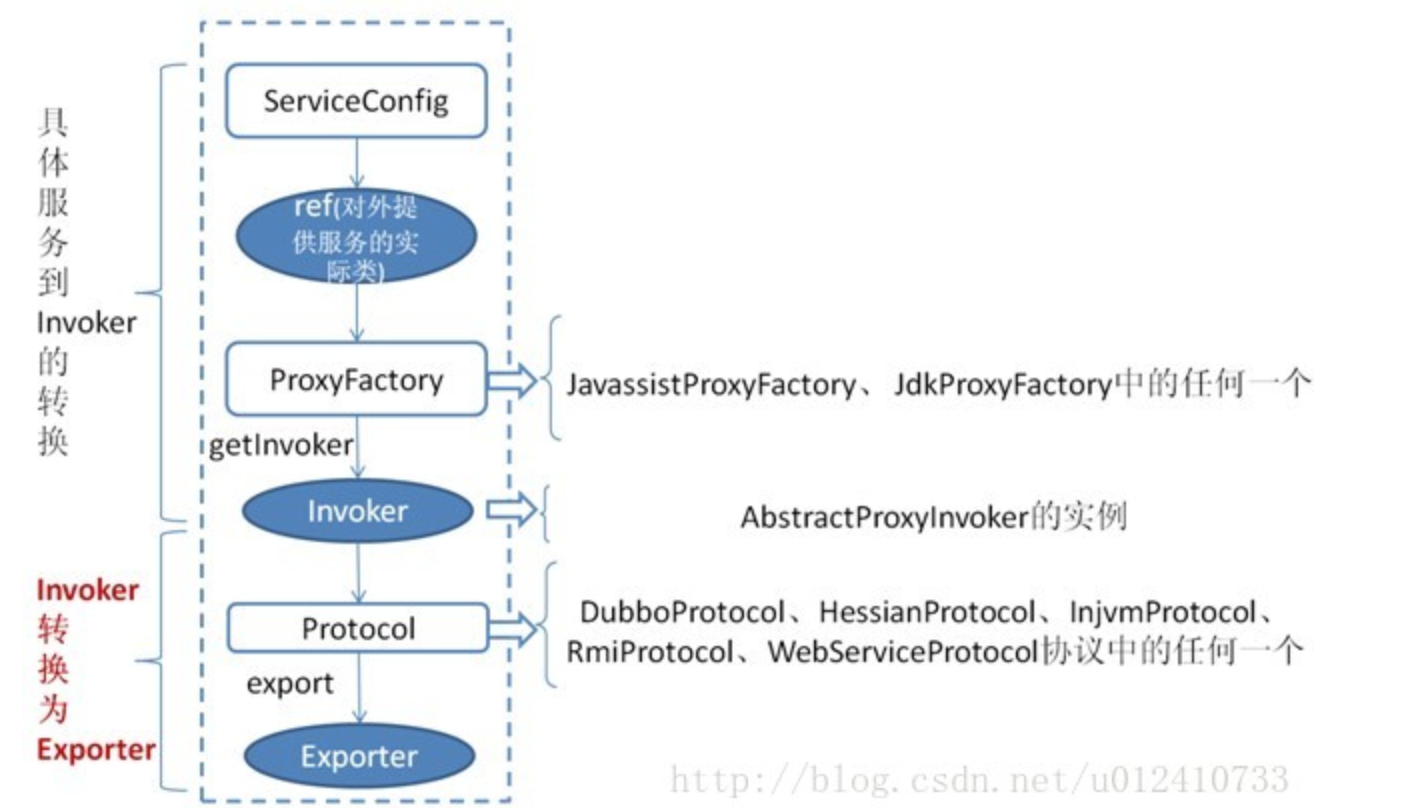
5.  服务消费者,从提供者地址列表中,基于软负载均衡算法,选一台提供者进行调用,如果调用失败,再选另一台调用。

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

dubbo的注册和发现，首先第一部分就是如何加载生成dubbo对象，这个已经写过了。

## 第二部分 服务注册|服务暴露

1. 在第一步里面会根据dubbo的配置文件生成serverBean初始化的bean对象，在serverBean类加载的时候，会调用afterPropertiesSet方法，将配置文件中的属性信息配置到bean里面，最后调用serviverconfig.export方法将服务报漏出去。
2. 首先从配置文件中获取对外提供的实现类Impl（如:DemoServiceImpl）,然后通过AOP代理ProxyFactory把Impl生成一个AbstractProxyInvoker实例，到这里就完成了具体服务到Invoker的转化。
3. 在Invoker中主要包括以下几个属性：
4. 1.前接口的Class实例(DemoService.class)
5. 2.当前接口的实例类对象(DemoServiceImpl.class)
6. 3.URL，配置信息的统一格式，所有扩展点都通过传递 URL 携带配置信息
7. 4.当前接口的代理对象Wrapper实例。



# dubbo服务治理

## 搞清楚dubbo服务链路

服务与服务之间是如何调用的，调用链路是什么，所以对各个服务之间的调用自动记录下来，然后自动将各个服务之间的依赖关系和调用链路生成出来，做成一张图，显示出来，大家才可以看到.

## 服务访问压力以及时长统计

需要自动统计各个接口和服务之间的调用次数以及访问延时，而且要分成两个级别:

接口粒度 每个服务的每个接口每天被调用多少次，TP50/TP90/TP99，三个档次的请求延时分别是多少

从入口开始 一个完整的请求链路经过几十个服务之后，完成一次请求，每天全链路走多少次,全链路请求延时的TP50/TP90/TP99，分别是多少

这些东西都搞定了之后，后面才可以来看当前系统的压力主要在哪里，来确定如何扩容和优化

# 服务降级

服务A调用服务B，如果服务B挂了，需要做服务降级，走一个备用的逻辑，在dubbo中，有一个mock配置，只需要建一个带mock后缀的类，并继承dubbo里面的一个服务就可以了。

# 如何实现了一个类似与dubbo的rpc

给大家说个最简单的回答思路

1.首先服务就得去注册中心注册吧，你是不是得有个注册中心，保留各个服务的信息，可以用zookeeper来做吧

2.然后你的消费者需要去注册中心拿对应的服务信息吧,而且每个服务可能会存在于多台机器上

3.接着你就该发起一次请求了，怎么发起请求？懵逼了吧,当然是基于动态代理了! 你面向接口获取到一个动态代理，这个动态代理就是接口在本地的一个代理，然后这个代理会找到服务对应的机器地址

4.然后找哪个机器发送请求？那肯定得有个负载均衡算法了，比如最简单的可以随机轮询是不是

5.找到一台机器后，就可以跟它发送请求了

5.1.咋发送呢？ 你可以说用netty了，nio方式

5.2.发送什么格式的数据？ 你可以说用hessian序列化协议了，或者是别的，对吧。然后请求过去了

6.服务器那边一样的，需要针对你自己的服务生成一个动态代理，监听某个网络端口，然后代理你本地的服务代码。接收到请求的时候，就调用对应的服务代码.