## Dubbo是什么

Dubbo是一个分布式服务框架，致力于提供高性能和透明化的RPC远程服务调用方案，以及SOA服务治理方案。

其核心部分包含:

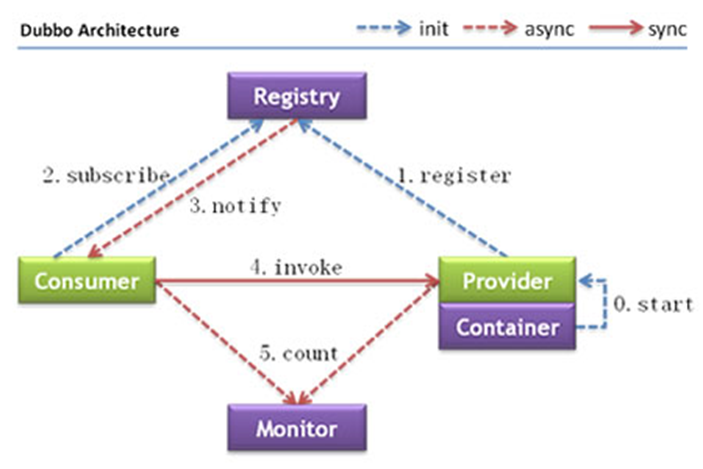
1. 透明化的远程通讯: 提供对多种基于长连接的NIO框架抽象封装，包括多种线程模型，序列化，以及“请求-响应”模式的信息交换方式。透明化的远程方法调用，就像调用本地方法一样调用远程方法，只需简单配置，没有任何API侵入。

2. 软负载均衡及集群容错: 提供基于接口方法的透明远程过程调用，包括多协议支持，以及软负载均衡，失败容错，地址路由，动态配置等集群支持。

3. 服务自动注册与发现: 基于注册中心目录服务，使服务消费方能动态的查找服务提供方，使地址透明，使服务提供方可以平滑增加或减少机器。

## Dubbo的架构

### 2.1节点角色：



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。

### 2.2调用关系：

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。

### 2.5连通关系：

（1）

注册中心负责服务地址的注册与查找，相当于目录服务，服务提供者和消费者只在启动时与注册中心交互，注册中心不转发请求，压力较小。

监控中心负责统计各服务调用次数，调用时间等，统计先在内存汇总后每分钟一次发送到监控中心服务器，并以报表展示。

服务提供者向注册中心注册其提供的服务，并汇报调用时间到监控中心，此时间不包含网络开销。

服务消费者向注册中心获取服务提供者地址列表，并根据负载算法直接调用提供者，同时汇报调用时间到监控中心，此时间包含网络开销。

注册中心，服务提供者，服务消费者三者之间均为长连接，监控中心除外。

注册中心通过长连接感知服务提供者的存在，服务提供者宕机，注册中心将立即推送事件通知消费者。

注册中心和监控中心全部宕机，不影响已运行的提供者和消费者，消费者在本地缓存了提供者列表。

注册中心和监控中心都是可选的，服务消费者可以直连服务提供者。

（2）

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）

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。

### 2.3协议支持

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。

默认使用Dubbo协议

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，尽量不要使用dubbo协议传输大文件或超大字符串

使用场景：常规远程服务方法调用

从上面的适用范围总结，dubbo适合小数据量大并发的服务调用，以及消费者机器远大于生产者机器数的情况，不适合传输大数据量的服务比如文件、视频等，除非请求量很低。

### 2.4负载均衡

dubbo提供4种负载均衡方式:

Random，随机，按权重配置随机概率，调用量越大分布越均匀，默认是这种方式。

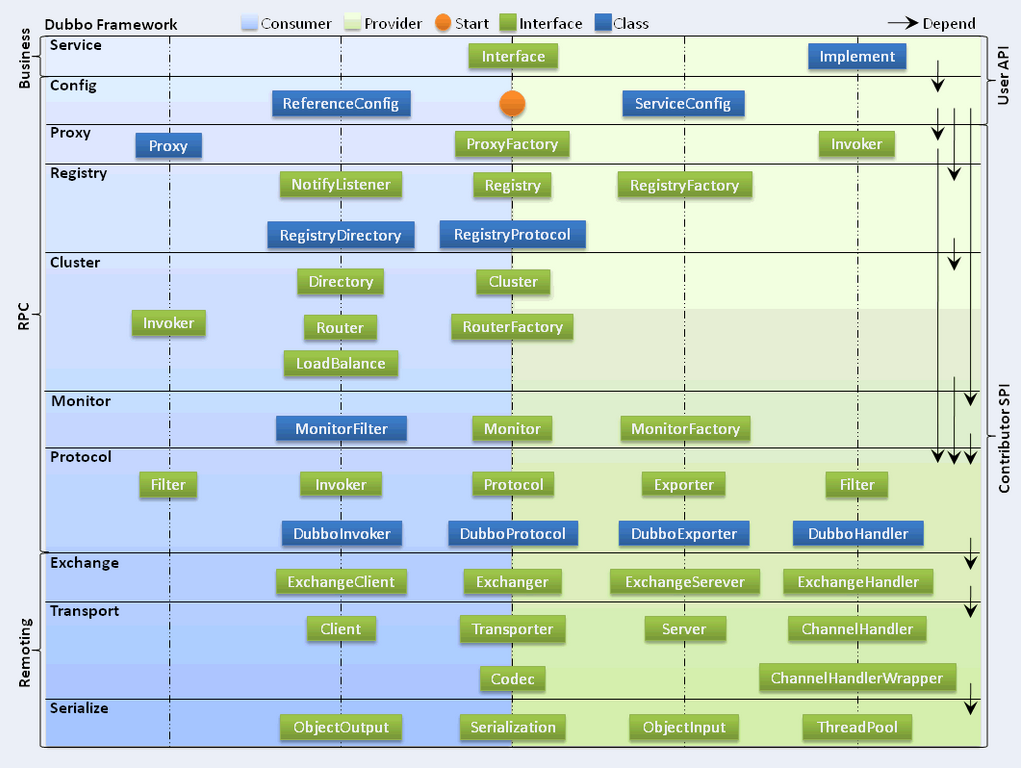
RoundRobin，轮询，按权重设置轮询比例，如果存在比较慢的机器容易在这台机器的请求阻塞较多。

LeastActive，最少活跃调用数，不支持权重，只能根据自动识别的活跃数分配，不能灵活调配。

ConsistentHash，一致性hash，对相同参数的请求路由到一个服务提供者上，如果有类似灰度发布需求可采用。

dubbo的负载均衡机制是在客户端调用时通过内存中的服务方信息及配置的负责均衡策略选择，如果对自己系统没有一个全面认知，建议先采用random方式。

## Dubbo的原理



框架分层架构中，各个层次的设计要点：

**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。

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。

**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。

**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。

**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。

**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。

**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。

**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。

**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。

**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。

### 3.1Dubbo源码模块图

Dubbo以包结构来组织各个模块，各个模块及其关系，如图所示：

可以通过Dubbo的代码（使用Maven管理）组织，与上面的模块进行比较。简单说明各个包的情况：

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 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。

### 3.2Dubbo初始化过程

和Spring中启动过程类似，spring启动时，将bean装载进容器中的时候，首先要解析bean。所以dubbo也是先读配置文件解析服务。

**解析服务：**

1）、基于dubbo.jar内的Meta-inf/spring.handlers配置，spring在遇到dubbo名称空间时，会回调DubboNamespaceHandler类。

2）、所有的dubbo标签，都统一用DubboBeanDefinitionParser进行解析，基于一对一属性映射，将XML标签解析为Bean对象。

源码截图：

在ServiceConfig.export 或者ReferenceConfig.get 初始化时，将Bean对象转会为url格式，将所以Bean属性转成url的参数。

然后将URL传给Protocol扩展点，基于扩展点的Adaptive机制，根据URL的协议头，进行不同协议的服务暴露和引用。

**暴露服务：**

a、 只暴露服务端口

在没有使用注册中心的情况，这种情况一般适用在开发环境下，服务的调用这和提供在同一个IP上，只需要打开服务的端口即可。

即，当配置 or

ServiceConfig解析出的URL的格式为：

Dubbo：//service-host/com.xxx.TxxService?version=1.0.0

基于扩展点的Adaptiver机制，通过URL的“dubbo：//”协议头识别，直接调用DubboProtocol的export（）方法，打开服务端口。

b、向注册中心暴露服务：

和上一种的区别：需要将服务的IP和端口一同暴露给注册中心。

ServiceConfig解析出的url格式为：

registry://registry-host/com.alibaba.dubbo.registry.RegistryService?export=URL.encode(“dubbo://service-host/com.xxx.TxxService?version=1.0.0”)

基于扩展点的Adaptive机制，通过URL的“registry：//”协议头识别，调用RegistryProtocol的export方法，将export参数中的提供者URL先注册到注册中心，再重新传给Protocol扩展点进行暴露：

Dubbo：//service-host/com.xxx.TxxService?version=1.0.0

**引用服务：**

a、直接引用服务：

在没有注册中心的，直连提供者情况下，

ReferenceConfig解析出的URL格式为：

Dubbo：//service-host/com.xxx.TxxService?version=1.0.0

基于扩展点的Adaptive机制，通过url的“dubbo：//”协议头识别，直接调用DubboProtocol的refer方法，返回提供者引用。

b、从注册中心发现引用服务：

此时，ReferenceConfig解析出的URL的格式为： registry://registry-host/com.alibaba.dubbo.registry.RegistryService?refer=URL.encode(“consumer://consumer-host/com.foo.FooService?version=1.0.0”)

基于扩展点的Apaptive机制，通过URL的“registry：//”协议头识别，就会调用RegistryProtocol的refer方法，基于refer参数总的条件，查询提供者URL，如：

Dubbo：//service-host/com.xxx.TxxService?version=1.0.0

### 3.3Dubbo调用过程

1.client一个线程调用远程接口，生成一个唯一的ID（比如一段随机字符串，UUID等），Dubbo是使用AtomicLong从0开始累计数字的

2.将打包的方法调用信息（如调用的接口名称，方法名称，参数值列表等），和处理结果的回调对象callback，全部封装在一起，组成一个对象object

3.向专门存放调用信息的全局ConcurrentHashMap里面put(ID, object)

4.将ID和打包的方法调用信息封装成一对象connRequest，使用IoSession.write(connRequest)异步发送出去

5.当前线程再使用callback的get()方法试图获取远程返回的结果，在get()内部，则使用synchronized获取回调对象callback的锁， 再先检测是否已经获取到结果，如果没有，然后调用callback的wait()方法，释放callback上的锁，让当前线程处于等待状态。

6.服务端接收到请求并处理后，将结果（此结果中包含了前面的ID，即回传）发送给客户端，客户端socket连接上专门监听消息的线程收到消息，分析结果，取到ID，再从前面的ConcurrentHashMap里面get(ID)，从而找到callback，将方法调用结果设置到callback对象里。

7.监听线程接着使用synchronized获取回调对象callback的锁（因为前面调用过wait()，那个线程已释放callback的锁了），再notifyAll()，唤醒前面处于等待状态的线程继续执行（callback的get()方法继续执行就能拿到调用结果了），至此，整个过程结束。