Duboo框架原理

分层结构

- Service：服务层 — 定义服务接口
- Config：配置层 — 对外配置接口，ServiceConfig，ReferenceConfig
- Proxy：服务代理层 — 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
- Registry：注册中心层 — 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, RegistryService
- Cluster：路由层 — 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, LoadBalance
- Monitor：监控层 — RPC 调用次数和调用时间监控，以 Statistics 为中心，扩展接口为 MonitorFactory, Monitor, MonitorService
- Protocol：远程调用层 — 封装 RPC 调用，以 Invocation, Result 为中心，扩展接口为 Protocol, Invoker, Exporter
- Exchange：信息交换层 — 封装请求响应模式，同步转异步，以 Request, Response 为中心，扩展接口为 Exchanger, ExchangeChannel, ExchangeClient, ExchangeServer
- Transport：网络传输层 — 抽象 mina 和 netty 为统一接口，以 Message 为中心，扩展接口为 Channel, Transporter, Client, Server, Codec
- Serialize：数据序列化层 — 可复用的一些工具，扩展接口为 Serialization, ObjectInput, ObjectOutput, ThreadPool

Dubbo SPI机制

- Dubbo SPI 与 JDK SPI 不同点
  - JDK SPI 通过ServiceLoader加载路径 META-INF/service/ 下的配置文件
  - dubbo SPI 配置文件目录：/META-INF/dubbo/
  - 文件内容由 com.foo.xxxProtocol 改为键值对形式 xxx = com.foo.xxxProtocol
  - Dubbo SPI机制增加了 AOP IOC的支持，一个扩展点可以通过setter注入到其他扩展点
- 改进了JDK SPI扩展机制，好处
  - JDK SPI只能通过遍历来查找扩展点和实例化，有可能一次性把扩展点全部加载出来了，如果有部分扩展点用不到就会导致资源浪费，Dubbo SPI可以做到按需指定加载
  - 为了更容易的定位问题，如果第三方库不存在，无法初始化，导致无法加载扩展点"A"，当用户使用"A"时，dubbo就会报无法加载扩展名的错误，而不是报哪些扩展名的实现加载失败
- 通过注解 **@SPI** 标识这个接口是可扩展的
- **@Adaptive**：dubbo会动态生成代理类，方法参数必须有 URL
  - @Adaptive({"server", "transporter"}) RemotingServer bind(URL url, ChannelHandler handler) throws RemotingException;
  - 会从URL中，先去取server配置作为SPI的key，取不到的话再取transporter配置作为SPI的key 如果还是空，则会用SPI默认配置去调用实现类，如果没有默认值，则抛出异常
- **@Activate**：扩展点自动激活加载
- 通过 ExtensionLoader 类去加载扩展点
  - 1. 从缓存中获取扩展Class对象 — 第一次加载，会去读配置文件，通过Class.forName，解析内容，得到class对象，然后缓存到本地
  - 2.通过反射实例化对象，放入缓存 — 第一次通过反射实例化对象，后面的都从缓存里拿
  - 3.向实例中注入依赖 — 基于setter方法注入
  - 4.增强扩展功能，类似AOP，通过@Wrapper 注解，实例化包装类
  - 5.实例初始化，如果继承了Lifecycle,执行initialize方法

负载均衡策略

- **Random**：默认，加权随机，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：加权轮询，按公约后的权重设置轮询比率 — 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：最少活跃调用数
  - 通过一个active来表示机器上活跃请求数量，请求到了active+1，请求结束active-1
  - active 越小，表示机器性能好，处理请求快，则给他分配更多的请求
- ConsistentHash：一致哈希，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。
  - 通过参数 hash.nodes 配置虚拟节点，默认 160 个虚拟节点
- ShortestResponse：最短响应时间 — 预估出来每个处理完请求的提供者所需时间，然后又选择最少最短时间的提供者进行调用

集群容错策略

- **Failover**：默认，失败自动切换 — 当出现失败，重试其它服务器。通常用于读操作，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：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：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：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：并行调用多个请求，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

通信协议

- **dubbo** 协议：默认
  - 缺省协议采用**单一长连接**和 **NIO 异步通讯**
  - 特性：单连接，长连接，TCP协议，NIO异步通信，Hessian 二进制序列化
  - 适用范围：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
  - 适用场景：常规远程服务方法调用
- http 协议
  - 基于 HTTP 表单的远程调用协议，采用 Spring 的 HttpInvoker 实现
  - 特性：多连接，短连接，HTTP协议，同步传输，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
- 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
  - 特性：多连接，短连接，HTTP协议，同步传输，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
- rmi 协议
  - 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
  - 特性：多连接，短连接，TCP协议，同步传输，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
- rest 协议 — 基于标准的Java REST API——JAX-RS 2.0 实现的REST调用支持
- gRPC 协议 — 计划使用 HTTP/2 通信，或者想利用 gRPC 带来的 Stream、反压、Reactive 编程等能力的开发者来说， 都可以考虑启用 gRPC 协议
- thirft 协议 — 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
- redis 协议 — 基于 Redis 实现的 RPC 协议。
- webservice 协议
  - 基于 WebService 的远程调用协议
  - 特性：多连接，短连接，HTTP协议，同步传输，SOAP 文本序列化

动态代理

- 基于JDK动态代理 — 调用 Proxy.newProxyInstance 创建代理
- 基于 javassist 字节码实现动态代理，**默认** — 字节码引擎工具,能够在运行时编译、生成Java Class
- 创建 InvokerInvocationHandler 的代理，动态代理类必须实现 InvocationHandler接口

服务降级

- 通过服务降级功能，临时屏蔽某个出错的非关键服务，定义降级后的返回策略
- mock=force:return+null： 表示消费方对该服务的方法调用**都直接返回 null 值**，不发起远程调用。用来屏蔽不重要服务不可用时对调用方的影响。
- mock=fail:return+null：表示消费方对该服务的方法调用在失败后，再返回 null 值，不抛异常。用来容忍不重要服务不稳定时对调用方的影响

工作原理

消费端流程

1. 服务端向注册中心进行服务注册
2. 消费端从注册中心拉取服务列表，缓存到本地
3. 消费端通过ProxyFactory创建代理对象
4. 根据路由规则，从服务列表中筛选服务地址
5. 根据负载均衡策略，挑选一个合适的服务
6. Protocol层组装协议，Exchange层封装请求，Transporter层通过网络框架，数据序列化后，发送请求

服务端流程

7. 服务从监听端口接收到请求，反序列化
8. Exchange层解析请求，Protocol层根据协议解析请求
9. 服务端通过ProxyFactory创建代理对象，调用具体的impl实现类，执行服务端业务逻辑

服务暴露和引用过程

Spring容器加载完成后，发布事件 DubboBootstrapApplicationListener 实现了 ApplicationListener，接收到事件通知通过 DubboBootStrap.start() 启动服务暴露和引用流程

服务暴露 ServiceConfig

1. 获取服务配置，检查并更新配置信息
2. 初始化元数据参数，找到 IP 和 端口 构造URL
3. 通过代理工厂，获取invoker代理对象
4. 调用RegistryProtocol里的export，进行服务暴露和服务注册
5. 发布服务暴露事件

如果是延迟暴露，则通过定时器延迟指定时间后，再进行服务暴露

根据scope判断是否暴露到本地

遍历注册中心（多注册中心）

服务引用 ReferenceConfig

服务引用默认采用懒加载的方式进行，在容器启动时并不会去执行这个过程

1. 获取引用配置，检查并更新配置
2. 初始化元数据参数

调用RegistryProtocol里的refer

3. 如果consumer需要注册的话，注册服务消费者
4. 创建路由规则链
5. 订阅providers,configurators,routes等节点
6. 获取集群配置，将多个服务提供者合并成一个invoker

7. 通过代理工厂，创建消费端服务代理

线程模型

派发策略：Dispatcher

- **all**：所有消息都派发到线程池，包括请求，响应，连接事件，断开连接事件，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，其他消息在IO线程上执行
- **execution**：只有请求消息派发到线程池，不包含响应，其他消息在IO线程上执行
- **connection**：在IO线程上，将连接，断开连接事件放入队列有序逐个执行，其他消息派发到线程池

线程池配置：ThreadPool

- **fixed**（默认）：**固定大小线程池**，启动时建立线程，不关闭，一直持有
- **cached**：缓存线程池，空闲1分钟自动删除
- **limited**：可伸缩线程池，但池中的线程数只会增长不会收缩。只增长不收缩的目的是为了避免收缩时突然来了大流量引起的性能问题
- **eager**：优先创建Worker线程池。在任务数量大于corePoolSize但是小于maximumPoolSize时，优先创建Worker来处理任务。当任务数量大于maximumPoolSize时，将任务放入阻塞队列中。阻塞队列充满时抛出RejectedExecutionException

如何实现优雅停机

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

服务消费方：
1、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。
2、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

服务提供方：
1、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
2、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。

路由规则

作用：用于过滤目标服务器地址

规则用途示例

- 黑名单
- 白名单
- 排除预发布机
- 读写分离
- 为重要应用提供额外的机器
- 隔离不同机房网段
- 提供者与消费者部署在同集群内，本机只访问本机的服务