**如何选型分布式架构**

提问：实现一个分布示框架最核心功能是什么?

**RPC远程调用技术：**

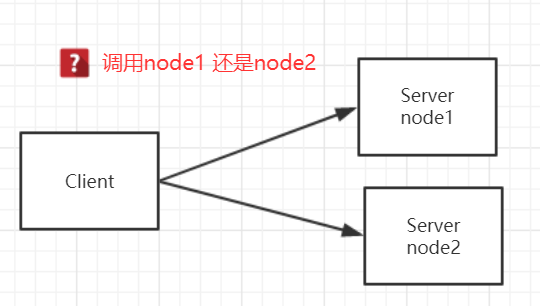


大家知道的 有哪些远程调用的 方式？拿几个大家比较熟悉的来举例：RMI 、Web Service、Http

|  |  |  |  |
| --- | --- | --- | --- |
| **协议** | **描述** | **优点** | **缺点** |
| RMI | JAVA 远程方法调用、使用原生二进制方式进行序列化 | 简单易用、SDK支持，提高开发效率 | 不支持跨语言 |
| Web Service | 比较早系统调用解决方案 ，跨语言, 其基于WSDL 生成 SOAP 进行消息的传递。 | SDK支持、跨语言 | 实现较重，发布繁琐 |
| Http | 采用htpp +json 实现 | 简单、轻量、跨语言 | 不支持SDK |

基于比较上述比较，大家会选择哪个方案，综合考虑 RMI是比较合适的方案，基本没有学习成本。而跨语言问题基本可以勿略。

如果服务端不是单个的话，这个方案差点我就用了。实际上服务端是多个的 ，好了新的问题又来了。



1. **负载均衡：**这么多个机器调用哪一台?
2. **服务发现：**怎样发现新的服务地址呢？
3. **健康检测：**服务宕机或恢复后怎么办？
4. **容错：**如果调用其中一台调用出错了怎么办？

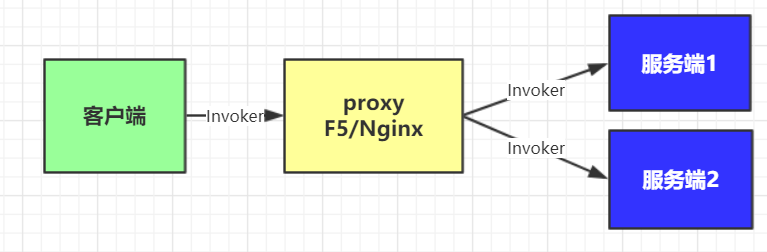
这些功能怎么解决呢？一个一个的去编码实现么？。有没有现成的方案可以直接借鉴呢？

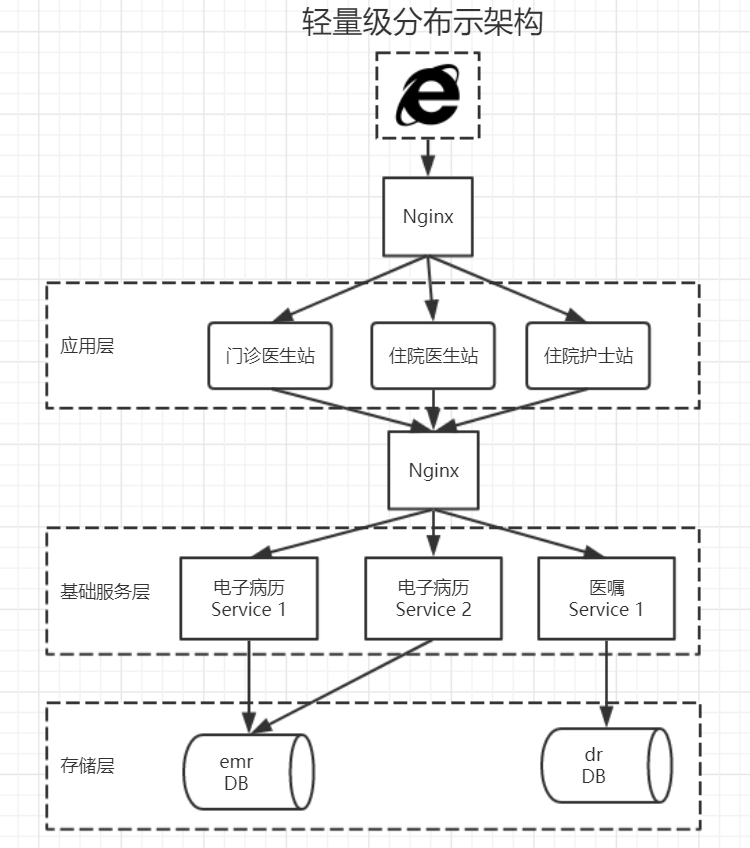
**分布式架构的三种解决方案：**

1. 基于反向代理的中心化架构
2. 嵌入应用内部的去中心化架构
3. 基于独立代理进程的Service Mesh架构

基于反向代理的集中式分布式架构

这是最简单和传统做法，在服务消费者和生产者之间，代理作为独立一层集中部署，由独立团队(一般是运维或框架)负责治理和运维。常用的集中式代理有硬件负载均衡器(如F5)，或者软件负载均衡器(如Nginx)，这种软硬结合两层代理也是业内常见做法，兼顾配置的灵活性(Nginx比F5易于配置)。





**Http+Nginx  方案总结：**

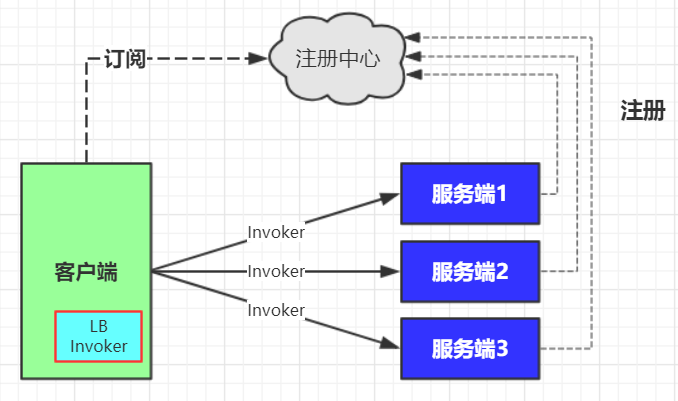
**优点：**简单快速、几乎没有学习成本

**适用场景：**轻量级分布式系统、局部分布式架构。

**瓶颈：**Nginx中心负载、Http传输、JSON序列化、开发效率、运维效率。

嵌入应用内部的去中心化架构

这是很多互联网公司比较流行的一种做法，代理(包括服务发现和负载均衡逻辑)以客户库的形式嵌入在应用程序中。这种模式一般需要独立的服务注册中心组件配合，服务启动时自动注册到注册中心并定期报心跳，客户端代理则发现服务并做负载均衡。我们所熟悉的 duboo 和spring cloud Eureka +Ribbon/'rɪbən/ 都是这种方式实现。

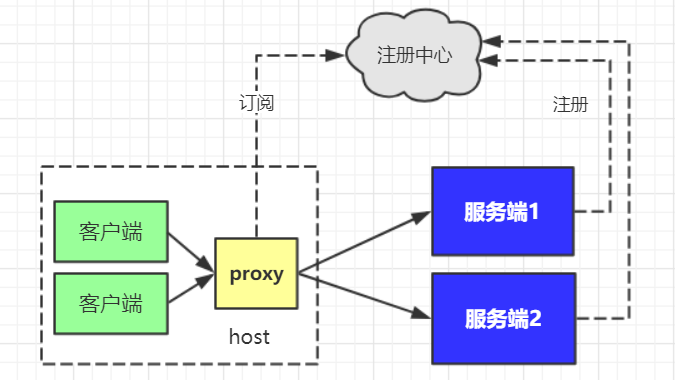


相比第一代架构它有以下特点几点：

* 去中心化，客户端直连服务端
* 动态注册和发现服务
* 高效稳定的网络传输
* 高效可容错的序列化

基于独立代理进程的架构(Service Mesh)

这种做法是上面两种模式的一个折中，代理既不是独立集中部署，也不嵌入在客户应用程序中，而是作为独立进程部署在每一个主机上，一个主机上的多个消费者应用可以共用这个代理，实现服务发现和负载均衡，如下图所示。这个模式一般也需要独立的服务注册中心组件配合，作用同第二代架构。



美团笔试：Service Mesh为了解决传统微服务框架"胖客户端"方式，引入的如下问题：

与业务无关的服务治理逻辑与业务代码强耦合，框架、SDK的升级与业务代码强绑定，多语言的胖客户端支持起来性价比极低。

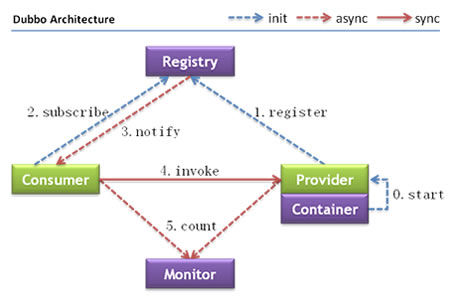
三种架构的比较

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **模式** | **优点** | **缺点** | **适应场景** | **案例** |
| 集中式负载架构 | 简单  集中式治理  与语言无关 | 配置维护成本高  多了一层IO  单点问题 | 大部分公司都适用，对运维有要求 | 亿贝、携程、早期互联网公司 |
| 客户端嵌入式架构 | 无单点  性能更好 | 客户端复杂  语言栈要求 | 中大规模公司、语言栈统一 | Dubbo 、  Twitter finagle、  Spring Cloud Ribbon |
| 独立进程代理架构 | 无单点  性能更好  与语言无关 | 运维部署复杂  开发联调复杂 | 中大规模公司  对运维有要求 | Smart Stack  Service Mesh |

**Dubbo 架构与设计说明**

**dubbo架构简要讲解**

        架构图



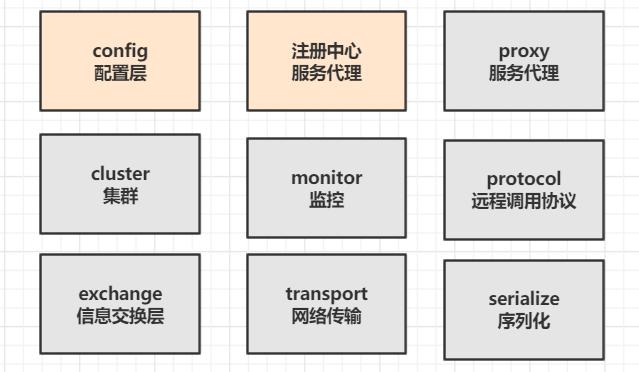
**流程说明：**

1. Provider(提供者)绑定指定端口并启动服务
2. 指供者连接注册中心，并发本机IP、端口、应用信息和提供服务信息发送至注册中心存储
3. Consumer(消费者），连接注册中心 ，并发送应用信息、所求服务信息至注册中心
4. 注册中心根据 消费 者所求服务信息匹配对应的提供者列表发送至Consumer 应用缓存。
5. Consumer 在发起远程调用时基于缓存的消费者列表择其一发起调用。
6. Provider 状态变更会实时通知注册中心、在由注册中心实时推送至Consumer

**这么设计的意义：**

1. Consumer 与Provider 解偶，双方都可以横向增减节点数。
2. 注册中心对本身可做对等集群，可动态增减节点，并且任意一台宕掉后，将自动切换到另一台
3. 去中心化，双方不直接依懒注册中心，即使注册中心全部宕机短时间内也不会影响服务的调用
4. 服务提供者无状态，任意一台宕掉后，不影响使用

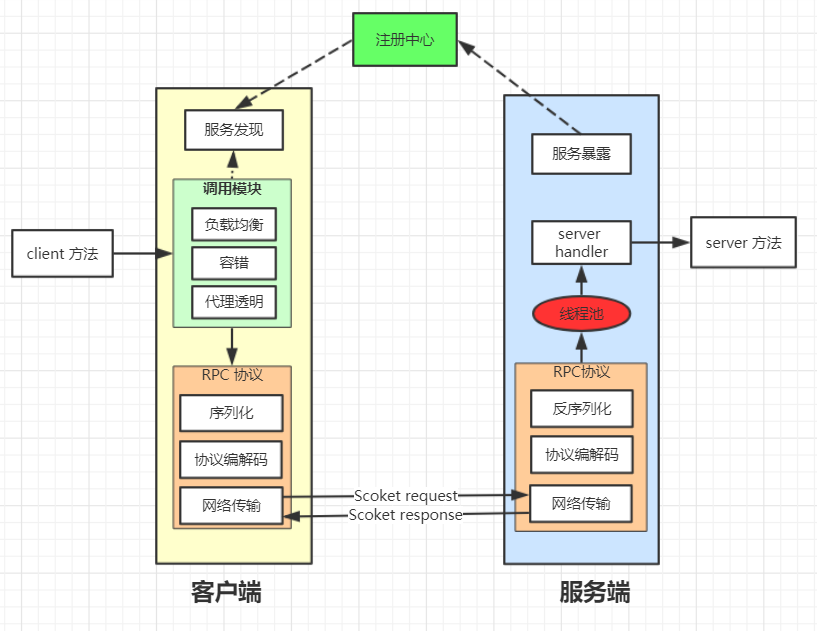
**Dubbo 整体设计**





* config **配置层**：对外配置接口，以 ServiceConfig, ReferenceConfig 为中心，可以直接初始化配置类，也可以通过 spring 解析配置生成配置类
* proxy **服务代理层**：服务接口透明代理，生成动态代理 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

其协作流程如下：



**Dubbo 中的SPI机制**

 在了解Dubbo的spi之前 先来了解一下 JAVA自带的SPI

 java spi的具体约定为:

当我们对某个接口进行实现后，在程序的META-INF/services/目录里同时创建一个以服务接口（全名）命名的文件。该文件里的内容是实现该服务接口的具体实现类（全名）。

当外部程序装配这个模块的时候，就能通过该jar包META-INF/services/里的配置文件找到具体的实现类名，并装载实例化，完成模块的注入。

基于这样一个约定就能很好的找到服务接口的实现类，而不需要再代码里制定（Class.forName(“xxx.xxx.xximpl”)，就不用这种在代码写死的方法）。

jdk提供服务实现查找的一个工具类java.util.ServiceLoader

**演示JAVA SPI机制**

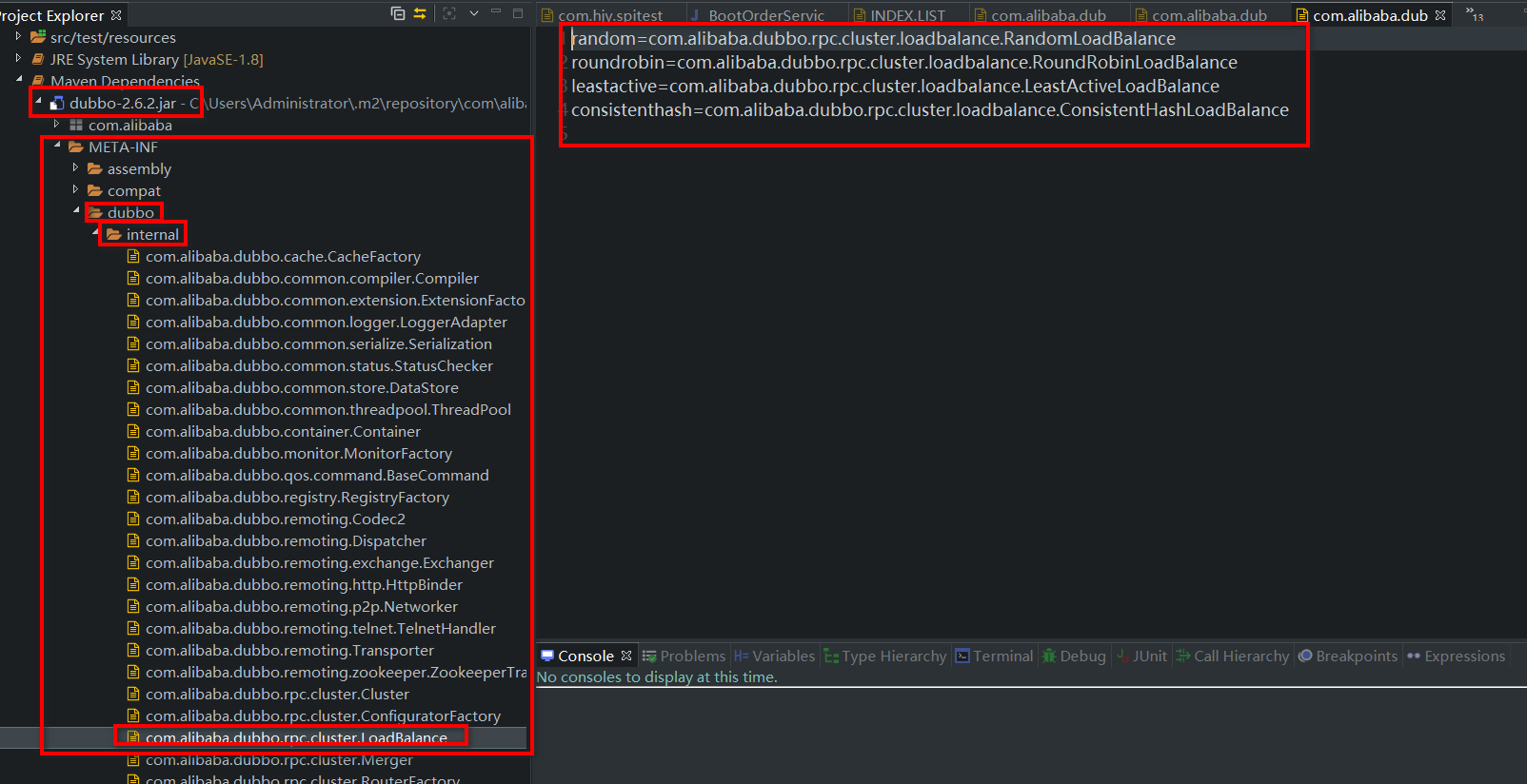
* 编写接口
* 编写实现类
* 编辑META-INF/services/xxx 文件
* 演示spi 实现

|  |
| --- |
| package com.hjy.spitest;  public interface TestInterface {  public void get();  } |
| package com.hjy.spitest;  public class TestImpl implements TestInterface{  @Override  public void get() {  System.out.println("invoke");  }  } |
|  |
| @Test  public void test1()  {  ServiceLoader<TestInterface> load = ServiceLoader.load(TestInterface.class);  System.out.println(load.iterator().next());  } |

如果在接口文件中写了多个实现类，那么在Test中load.iterator()，就会有多个实现对象。

**Dubbo的SPI机制：**

dubbo spi 在JAVA自带的SPI基础上加入了扩展点的功能，即每个实现类都会对应至一个扩展点名称，其目的是 应用可基于此名称进行相应的装配。



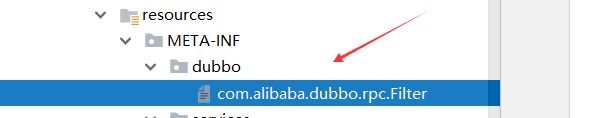
扩展点其实就是在配置文件中，以key value的形式存储。Key就是扩展点。

演示Dubbo SPI机制：

* 编写Filter 过滤器
* 编写 dubbo spi 配置文件
* 装配自定义Filter

dubbo spi 目录文件

（注意，这里是在META-INF下的dubbo，而不是services下。）



dubbo spi 文件内容：

（注意，这里的内容是以key，value的形式）

luban=tuling.dubbo.server.LubanFilter

装配自定义Filter

