> 点赞再看，养成习惯，微信搜一搜【**三太子敖丙**】关注这个喜欢写情怀的程序员。
>
> 本文 **GitHub** https://github.com/JavaFamily 已收录，有一线大厂面试完整考点、资料以及我的系列文章。

前言

接下来一段时间敖丙将带大家开启紧张刺激的 Dubbo 之旅！是的要开始写 Dubbo 系列的文章了，之前我已经写过一篇架构演进的文章，也说明了微服务的普及化以及重要性，服务化场景下随之而来的就是服务之间的通信问题，那服务间的通信脑海中想到的就是 RPC，说到 RPC 就离不开咱们的 Dubbo。

这篇文章敖丙先带着大家来**总览全局**，一般而言熟悉一个框架你要先知道这玩意是做什么的，能解决什么痛点，核心的模块是什么，大致运转流程是怎样的。

你要一来就扎入细节之中无法自拔，一波 DFS 直接被劝退的可能性高达99.99%，所以本暖男敖丙将带大家先过一遍 **Dubbo 的简介、总体分层、核心组件以及大致调用流程**。

不仅如此我还会带着大家过一遍如果要让你**设计一个 RPC 框架你看看都需要什么功能**？这波操作之后你会发现嘿嘿 Dubbo 怎么设计的和我想的一样呢？真是英雄所见略同啊！

而且我还会写一个简单版 RPC 框架实现，让大家明白 RPC 到底是如何工作的。

如果看了这篇文章你要还是不知道 Dubbo 是啥，我可以要劝退了。

我们先来谈一谈什么叫 RPC ，我发现有很多同学不太了解这个概念，还有人把 RPC 和 HTTP 来进行对比。所以咱们先来说说什么是 RPC。

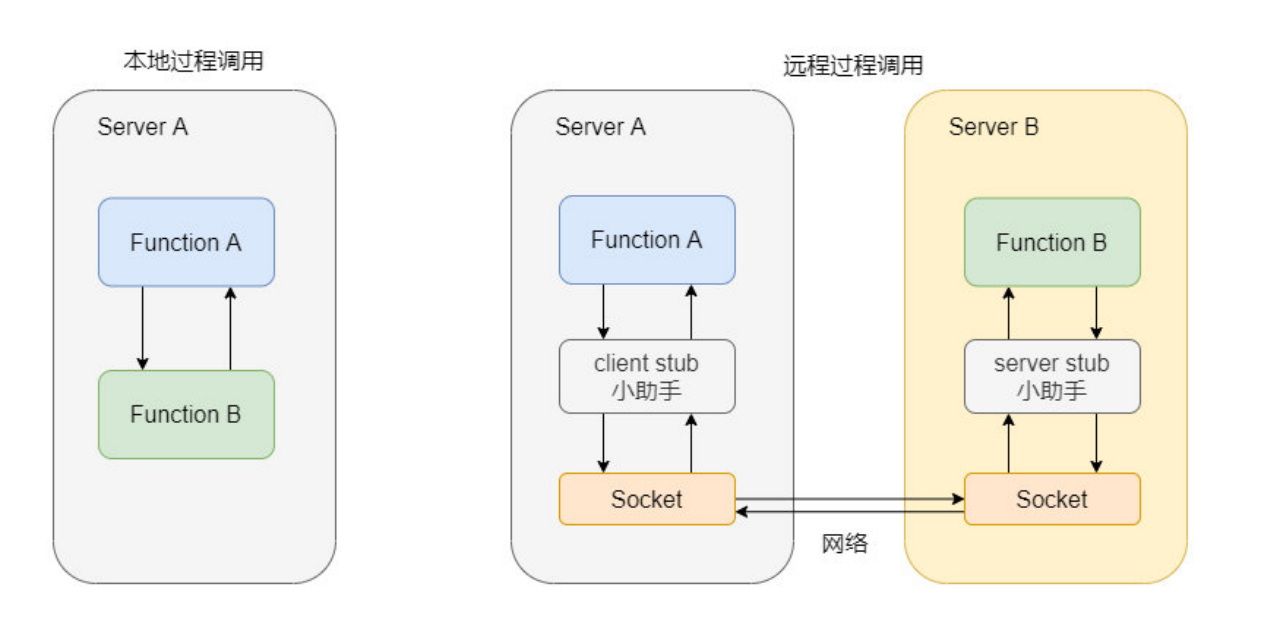
什么是 RPC

RPC，Remote Procedure Call 即远程过程调用，远程过程调用其实对标的是本地过程调用，本地过程调用你熟悉吧?

想想那青葱岁月，你在大学赶着期末大作业，正在攻克图书管理系统，你奋笔疾书疯狂地敲击键盘，实现了图书借阅、图书归还等等模块，你实现的一个个方法之间的调用就叫本地过程调用。

你要是和我说你实现图书馆里系统已经用了服务化，搞了远程调用了，我只能和你说你有点东西。

简单的说本机上内部的方法调用都可以称为本地过程调用，而远程过程调用实际上就指的是你本地调用了远程机子上的某个方法，这就是远程过程调用。



所以说 RPC 对标的是本地过程调用，至于 RPC 要如何调用远程的方法可以走 HTTP、也可以是基于 TCP 自定义协议。

所以说你讨论 RPC 和 HTTP 就不是一个层级的东西。

而 **RPC 框架就是要实现像那小助手一样的东西，目的就是让我们使用远程调用像本地调用一样简单方便，并且解决一些远程调用会发生的一些问题**，使用户用的无感知、舒心、放心、顺心，它好我也好，快乐没烦恼。

如何设计一个 RPC 框架

在明确了什么是 RPC，以及 RPC 框架的目的之后，咱们想想如果让你做一款 RPC 框架你该如何设计?

服务消费者

我们先从消费者方(也就是调用方)来看需要些什么，首先消费者面向接口编程，所以需要得知有哪些接口可以调用，可以通过**公用 jar 包**的方式来维护接口。

现在知道有哪些接口可以调用了，但是只有接口啊，具体的实现怎么来？这事必须框架给处理了！所以还**需要来个代理类**，让消费者只管调，啥事都别管了，我**代理帮你搞定**。

对了，还需要告诉代理，你调用的是哪个方法，并且参数的值是什么。

虽说代理帮你搞定但是代理也需要知道它到底要调哪个机子上的远程方法，所以**需要有个注册中心**，这样调用方从注册中心可以知晓可以调用哪些服务提供方，一般而言提供方不止一个，毕竟只有一个挂了那不就没了。

所以提供方一般都是集群部署，那调用方需要通过**负载均衡**来选择一个调用，可以通过**某些策略**例如同机房优先调用啊啥的。

当然还需要有**容错机制**，毕竟这是远程调用，网络是不可靠的，所以可能需要重试什么的。

还要和服务提供方**约定一个协议**，例如我们就用 HTTP 来通信就好啦，也就是大家要讲一样的话，不然可能听不懂了。

当然序列化必不可少，毕竟我们本地的结构是“立体”的，需要序列化之后才能传输，因此还需要**约定序列化格式**。

并且这过程中间可能还需要掺入一些 Filter，来作一波统一的处理，例如调用计数啊等等。

这些都是框架需要做的，让消费者像在调用本地方法一样，无感知。

服务提供者

服务提供者肯定要**实现对应的接口**这是毋庸置疑的。

然后需要把自己的接口暴露出去，向**注册中心注册自己**，暴露自己所能提供的服务。

然后有消费者请求过来需要处理，提供者需要用和消费者**协商好的协议**来处理这个请求，然后做**反序列化**。

序列化完的请求应该**扔到线程池里面做处理**，某个线程接受到这个请求之后找到对应的实现调用，然后再**将结果原路返回**。

注册中心

上面其实我们都提到了注册中心，这东西就相当于一个平台，大家在上面暴露自己的服务，也在上面得知自己能调用哪些服务。

当然还能做配置中心，将配置集中化处理，动态变更通知订阅者。

监控运维

面对众多的服务，精细化的监控和方便的运维必不可少。

这点很多开发者在开发的时候察觉不到，到你真正上线开始运行维护的时候，如果没有良好的监控措施，快速的运维手段，到时候就是睁眼瞎！手足无措，等着挨批把！

那种痛苦不要问我为什么知道，我就是知道！

小结一下

让我们小结一下，大致上一个 RPC 框架需要做的就是约定要通信协议，序列化的格式、一些容错机制、负载均衡策略、监控运维和一个注册中心！

简单实现一个 RPC 框架

没错就是简单的实现，上面我们在思考如何设计一个 RPC 框架的时候想了很多，那算是生产环境使用级别的功能需求了，我们这是 Demo，目的是突出 RPC框架重点功能 - **实现远程调用**。

所以啥七七八八的都没，并且我用伪代码来展示，其实也就是删除了一些保护性和约束性的代码，因为看起来太多了不太直观，需要一堆 try-catch 啥的，因此我删减了一些，直击重点。

Let's Do It!

首先我们定义一个接口和一个简单实现。

```
public interface AobingService {
    String hello(String name);
}

public class AobingServiceImpl implements AobingService {
    public String hello(String name) {
        return "Yo man Hello, I am" + name;
    }
}
```

然后我们再来实现服务提供者暴露服务的功能。

```
public class AobingRpcFramework {
     public static void export(Object service, int port) throws Exception {
          ServerSocket server = new ServerSocket(port);
          while(true) {
              Socket socket = server.accept();
              new Thread(new Runnable() {
                  //反序列化
                  ObjectInputStream input = new
ObjectInputStream(socket.getInputStream());
                  String methodName = input.read(); //读取方法名
                  Class<?>[] parameterTypes = (Class<?>[])
input.readObject(); //参数类型
                  Object[] arguments = (Object[]) input.readObject(); //参
数
                  Method method = service.getClass().getMethod(methodName,
parameterTypes);  //找到方法
                  Object result = method.invoke(service, arguments); //调用
方法
                  // 返回结果
                  ObjectOutputStream output = new
ObjectOutputStream(socket.getOutputStream());
                  output.writeObject(result);
              }).start();
          }
     }
    public static <T> T refer (Class<T> interfaceClass, String host, int
port) throws Exception {
       return  (T) Proxy.newProxyInstance(interfaceClass.getClassLoader(),
new Class<?>[] {interfaceClass},
            new InvocationHandler() {
                public Object invoke(Object proxy, Method method, Object[]
arguments) throws Throwable {
                    Socket socket = new Socket(host, port);  //指定
provider 的 ip 和端口
                    ObjectOutputStream output = new
ObjectOutputStream(socket.getOutputStream());
                    output.write(method.getName());  //传方法名
                    output.writeObject(method.getParameterTypes());  //传参
数类型
                    output.writeObject(arguments);  //传参数值
                    ObjectInputStream input = new
ObjectInputStream(socket.getInputStream());
                    Object result = input.readObject();  //读取结果
                    return result;
```

```
                }
            });
        }
    }
```

好了，这个 RPC 框架就这样好了，是不是很简单？就是**调用者传递了方法名、参数类型和参数值，提供者接收到这样参数之后调用对于的方法返回结果就好了**！这就是远程过程调用。

我们来看看如何使用

```
        //服务提供者只需要暴露出接口
        AobingService service = new AobingServiceImpl ();
        AobingRpcFramework.export(service, 2333);

        //服务调用者只需要设置依赖
        AobingService service =
AobingRpcFramework.refer(AobingService.class, "127.0.0.1", 2333);
        service.hello();
```

看起来好像好不错哟，不过这很是简陋，用作 demo 有助理解还是极好的！

接下来就来看看 Dubbo 吧！上正菜！

Dubbo 简介

Dubbo 是阿里巴巴 2011年开源的一个基于 Java 的 RPC 框架，中间沉寂了一段时间，不过其他一些企业还在用 Dubbo 并自己做了扩展，比如当当网的 Dubbox，还有网易考拉的 Dubbok。

但是在 2017 年阿里巴巴又重启了对 Dubbo 维护。在 2017 年荣获了开源中国 2017 最受欢迎的中国开源软件 Top 3。

在 2018 年和 Dubbox 进行了合并，并且进入 Apache 孵化器，在 2019 年毕业正式成为 Apache 顶级项目。

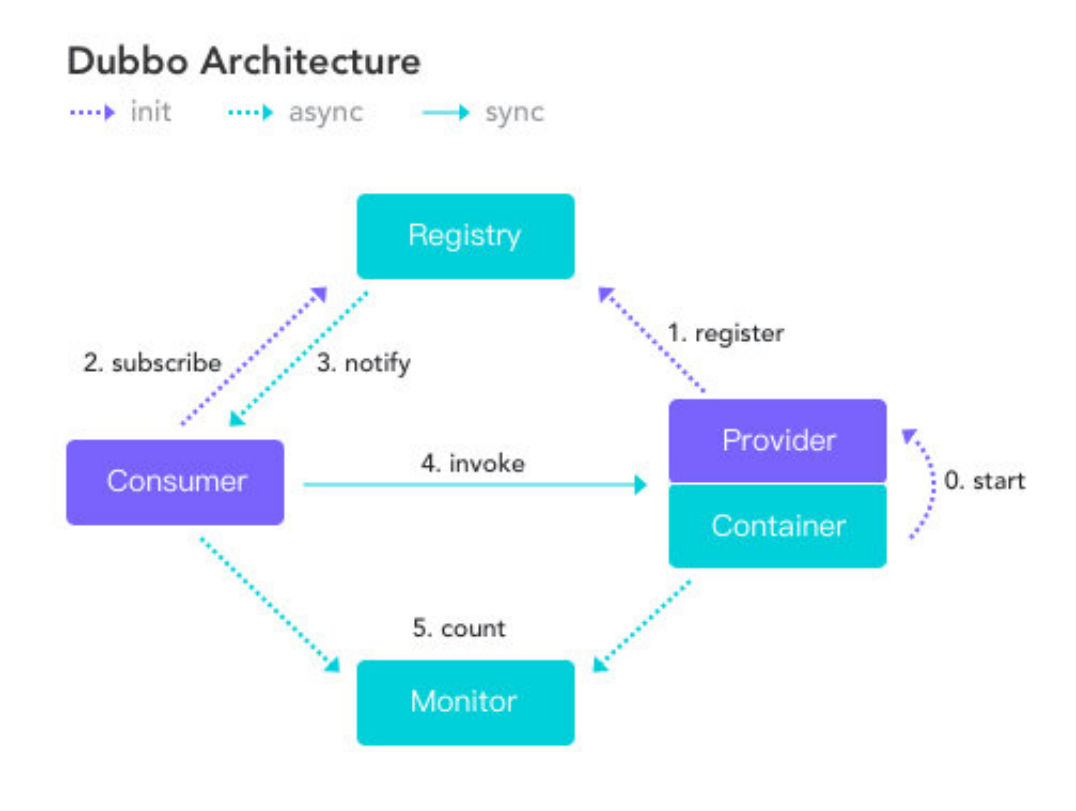
目前 Dubbo 社区主力维护的是 2.6.x 和 2.7.x 两大版本，2.6.x 版本主要是 bug 修复和少量功能增强为准，是稳定版本。

而 2.7.x 是主要开发版本，更新和新增新的 feature 和优化，并且 2.7.5 版本的发布被 Dubbo 认为是里程碑式的版本发布，之后我们再做分析。

它实现了面向接口的代理 RPC 调用，并且可以配合 ZooKeeper 等组件实现服务注册和发现功能，并且拥有负载均衡、容错机制等。

Dubbo 总体架构

我们先来看下官网的一张图。



本丙再暖心的给上图内每个节点的角色说明一下。

节点	角色说明
Consumer	需要调用远程服务的服务消费方
Registry	注册中心
Provider	服务提供方
Container	服务运行的容器
Monitor	监控中心

我再来大致说一下整体的流程，首先服务提供者 **Provider 启动然后向注册中心注册**自己所能提供的服务。

服务消费者 **Consumer 启动向注册中心订阅**自己所需的服务。然后注册中心将提供者元信息通知给 Consumer， 之后 Consumer 因为已经从注册中心获取提供者的地址，因此可以**通过负载均衡选择一个 Provider 直接调用** 。

之后服务提供方元数据变更的话**注册中心会把变更推送给服务消费者**。

服务提供者和消费者都会在内存中记录着调用的次数和时间，然后**定时的发送统计数据到监控中心**。

一些注意点

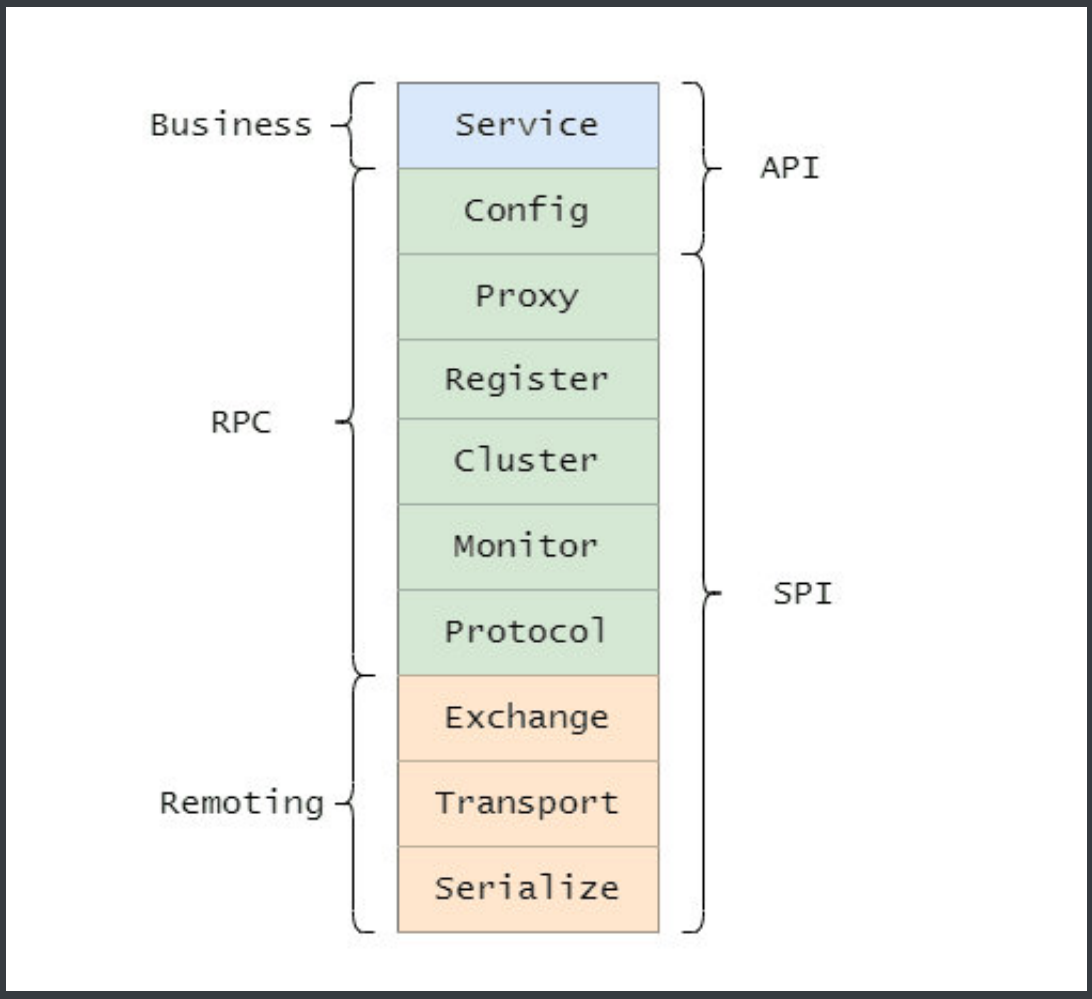
首先**注册中心和监控中心是可选的**，你可以不要监控，也不要注册中心，直接在配置文件里面写然后提供方和消费方直连。

然后注册中心、提供方和消费方之间都是长连接，和监控方不是长连接，并且**消费方是直接调用提供方，不经过注册中心**。

就算**注册中心和监控中心宕机了也不会影响到已经正常运行的提供者和消费者**，因为消费者有本地缓存提供者的信息。

Dubbo 分层架构

总的而言 Dubbo 分为三层，如果每一层再细分下去，一共有十层。别怕也就十层，本丙带大家过一遍，大家先有个大致的印象，之后的文章丙会带着大家再深入。



大的三层分别为 Business（业务层）、RPC 层、Remoting，并且还分为 API 层和 SPI 层。

分为大三层其实就是和我们知道的网络分层一样的意思，**只有层次分明，职责边界清晰才能更好的扩展**。

而分 API 层和 SPI 层这是 Dubbo 成功的一点，**采用微内核设计+SPI扩展**，使得有特殊需求的接入方可以自定义扩展，做定制的二次开发。

接下来咱们再来看看每一层都是干嘛的。

- Service，业务层，就是咱们开发的业务逻辑层。
- Config，配置层，主要围绕 ServiceConfig 和 ReferenceConfig，初始化配置信息。
- Proxy，代理层，服务提供者还是消费者都会生成一个代理类，使得服务接口透明化，代理层做远程调用和返回结果。
- Register，注册层，封装了服务注册和发现。
- Cluster，路由和集群容错层，负责选取具体调用的节点，处理特殊的调用要求和负责远程调用失败的容错措施。
- Monitor，监控层，负责监控统计调用时间和次数。
- Portocol，远程调用层，主要是封装 RPC 调用，主要负责管理 Invoker，Invoker代表一个抽象封装了的执行体，之后再做详解。
- Exchange，信息交换层，用来封装请求响应模型，同步转异步。
- Transport，网络传输层，抽象了网络传输的统一接口，这样用户想用 Netty 就用 Netty，想用 Mina

就用 Mina。
- Serialize，序列化层，将数据序列化成二进制流，当然也做反序列化。

SPI

我再稍微提一下 SPI（Service Provider Interface），是 JDK 内置的一个服务发现机制，**它使得接口和具体实现完全解耦**。我们只声明接口，具体的实现类在配置中选择。

具体的就是你定义了一个接口，然后在 `META-INF/services` 目录下**放置一个与接口同名的文本文件**，文件的内容为**接口的实现类**，多个实现类用换行符分隔。

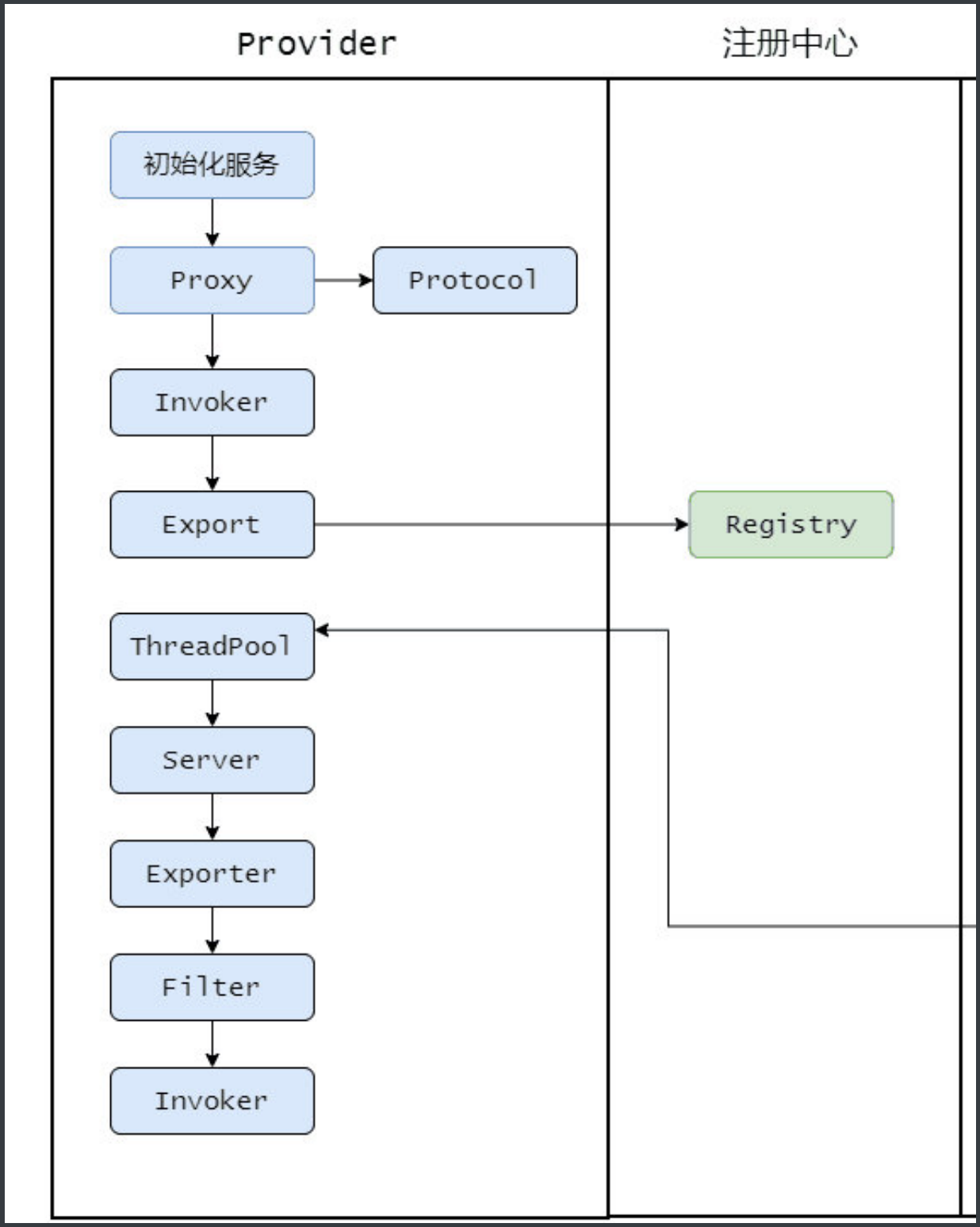
这样就通过配置来决定具体用哪个实现！

而 Dubbo SPI 还做了一些改进，篇幅有限留在之后再谈。

Dubbo 调用过程

上面我已经介绍了每个层到底是干嘛的，我们现在再来串起来走一遍调用的过程，加深你对 Dubbo 的理解，让知识点串起来，由点及面来一波连连看。

我们先从服务提供者开始，看看它是如何工作的。



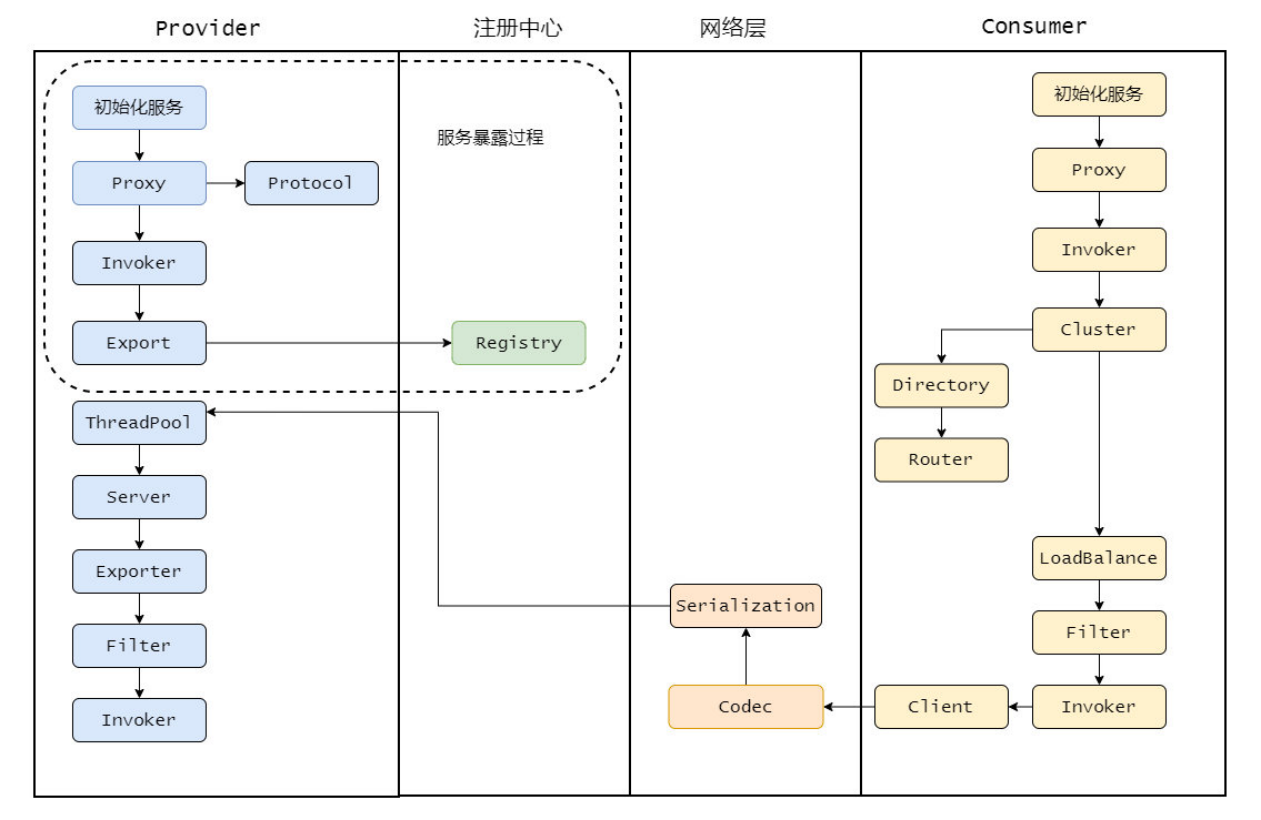
服务暴露过程

首先 Provider 启动，通过 Proxy 组件根据具体的协议 Protocol 将需要暴露出去的接口封装成 Invoker，Invoker 是 Dubbo 一个很核心的组件，代表一个可执行体。

然后再通过 Exporter 包装一下，这是为了在注册中心暴露自己套的一层，然后将 Exporter 通过 Registry 注册到注册中心。 这就是整体服务暴露过程。

消费过程

接着我们来看消费者调用流程（把服务者暴露的过程也在图里展示出来了，这个图其实算一个挺完整的流程图了）。



首先消费者启动会向注册中心拉取服务提供者的元信息，然后调用流程也是从 Proxy 开始，毕竟都需要代理才能无感知。

Proxy 持有一个 Invoker 对象，调用 invoke 之后需要通过 Cluster 先从 Directory 获取所有可调用的远程服务的 Invoker 列表，如果配置了某些路由规则，比如某个接口只能调用某个节点的那就再过滤一遍 Invoker 列表。

剩下的 Invoker 再通过 LoadBalance 做负载均衡选取一个。然后再经过 Filter 做一些统计什么的，再通过 Client 做数据传输，比如用 Netty 来传输。

传输需要经过 Codec 接口做协议构造，再序列化。最终发往对应的服务提供者。

服务提供者接收到之后也会进行 Codec 协议处理，然后反序列化后将请求扔到线程池处理。某个线程会根据请求找到对应的 Exporter ，而找到 Exporter 其实就是找到了 Invoker，但是还会有一层层 Filter，经过一层层过滤链之后最终调用实现类然后原路返回结果。

完成整个调用过程！

总结

这次敖丙带着大家先了解了下什么是 RPC，然后规划了一波 RPC 框架需要哪些组件，然后再用代码实现了一个简单的 RPC 框架。

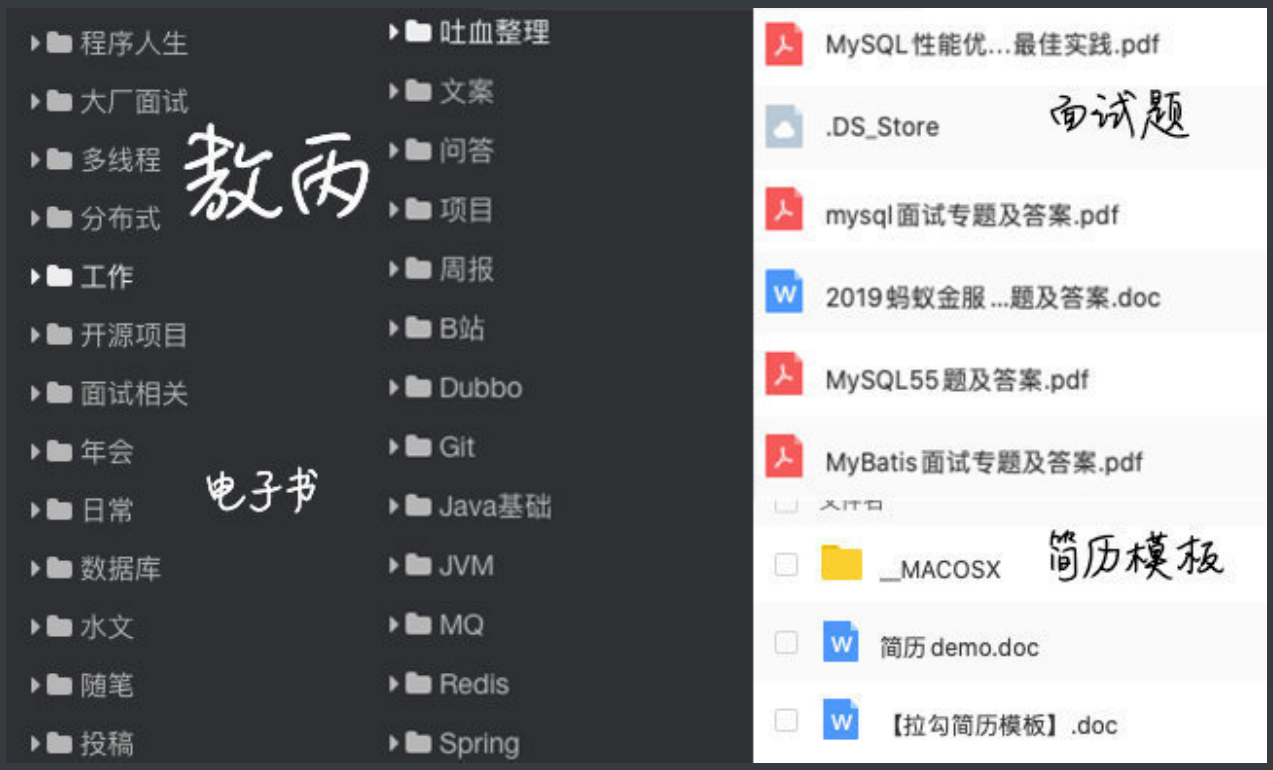
然后带着大家了解了下 Dubbo 的发展历史、总体架构、分层设计架构以及每个组件是干嘛的，再带着大伙走了一遍整体调用过程。

我真的是太暖了啊！

dubbo近期我会安排几个章节继续展开，最后会出一个面试版本的dubbo，我们拭目以待吧。

絮叨

另外，敖丙把自己的面试文章整理成了一本电子书，共 1630页！目录如下，还有我复习时总结的面试题以及简历模板



现在免费送给大家，点赞后在我的公众号**三太子敖丙**回复 **【资料】** 即可获取。

我是敖丙，你知道的越多，你不知道的越多，我们下期见！

人才们的 **【三连】** 就是敖丙创作的最大动力，如果本篇博客有任何错误和建议，欢迎人才们留言！

> 文章持续更新，可以微信搜一搜「 **三太子敖丙** 」第一时间阅读，回复**【资料】**有我准备的一线大厂面试资料和简历模板，本文 **GitHub** https://github.com/JavaFamily 已经收录，有大厂面试完整考点，欢迎Star。