Dubbo 序列化协议 5 连问，你接得住不？

Java后端 2019-12-08

编辑 | Java之间

来源 | www.toutiao.com/i6745361206137061895/

1）dubbo 支持哪些通信协议？

2）支持哪些序列化协议？

3）说一下 Hessian 的数据结构？

4）PB 知道吗？

5）为什么 PB 的效率是最高的？

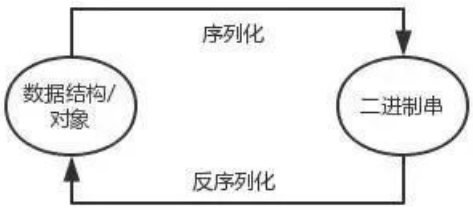
面试官心理分析

上一个问题，说说 dubbo 的基本工作原理，那是你必须知道的，至少要知道 dubbo 分成哪些层，然后平时怎么发起 rpc 请求的，注册、发现、调用，这些是基本的。

接着就可以针对底层进行深入的问问了，比如第一步就可以先问问序列化协议这块，就是平时 RPC 的时候怎么走的？

面试题剖析

序列化，就是把数据结构或者是一些对象，转换为二进制串的过程，而反序列化是将在序列化过程中所生成的二进制串转换成数据结构或者对象的过程



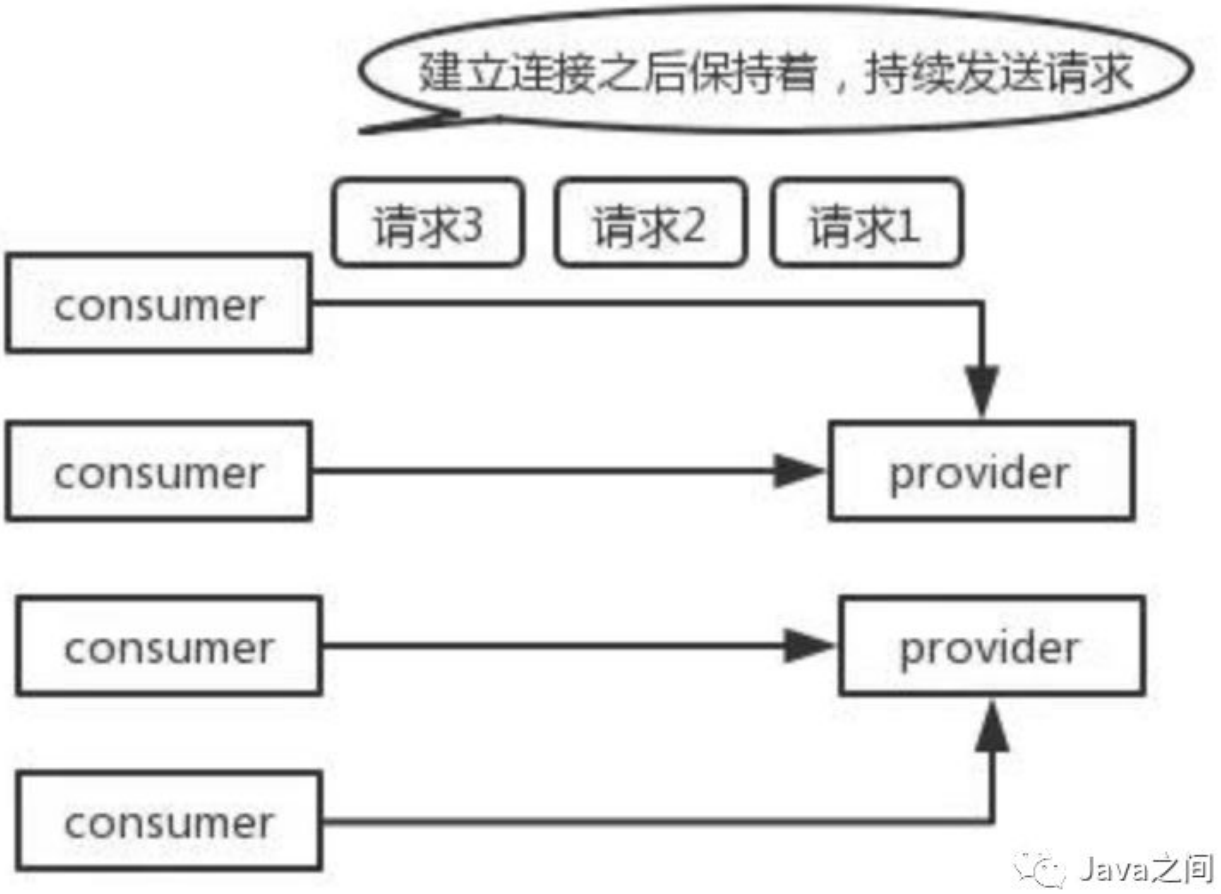
dubbo 支持不同的通信协议

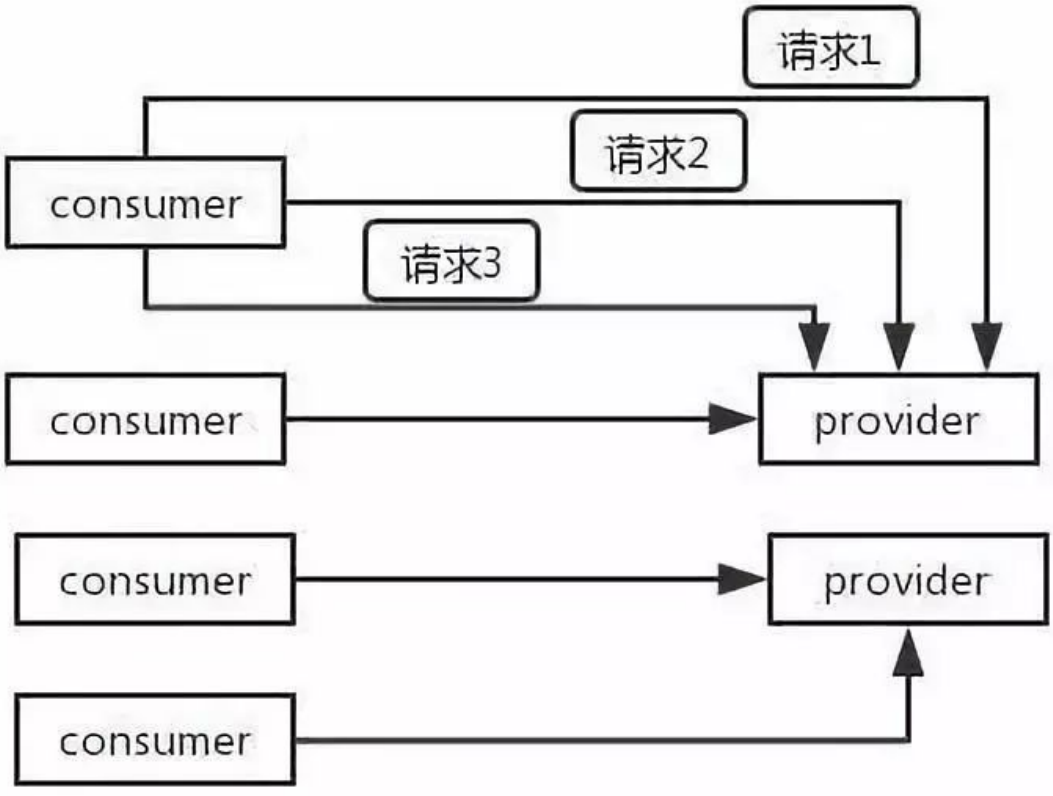
1、dubbo 协议

默认就是走 dubbo 协议，单一长连接，进行的是 NIO 异步通信，基于 hessian 作为序列化协议。使用的场景是：传输数据量小（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。

长连接，通俗点说，就是建立连接过后可以持续发送请求，无须再建立连接。





2、rmi 协议

走 Java 二进制序列化，多个短连接，适合消费者和提供者数量差不多的情况，适用于文件的传输，一般较少用。

3、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的情况，适用于文件的传输，一般较少用。

4、http 协议

走 json 序列化。

5、webservice

走 SOAP 文本

dubbo 支持的序列化协议?

dubbo 支持 hession、Java 二进制序列化、json、SOAP 文本序列化多种序列化协议。但是 hessian 是其默认的序列化协议。

说一下 Hessian 的数据结构?

Hessian 的对象序列化机制有 8 种原始类型：

- 原始二进制数据
- boolean
- 64-bit date（64 位毫秒值的日期）
- 64-bit double
- 32-bit int
- 64-bit long
- null
- UTF-8 编码的 string

另外还包括 3 种递归类型：

- list for lists and arrays
- map for maps and dictionaries
- object for objects

还有一种特殊的类型：

- ref：用来表示对共享对象的引用。

为什么 PB 的效率是最高的?

可能有一些同学比较习惯于 JSON or XML 数据存储格式，对于 Protocol Buffer 还比较陌生。

Protocol Buffer 其实是 Google 出品的一种轻量并且高效的结构化数据存储格式，性能比 JSON、XML 要高很多。

其实 PB 之所以性能如此好，主要得益于两个：

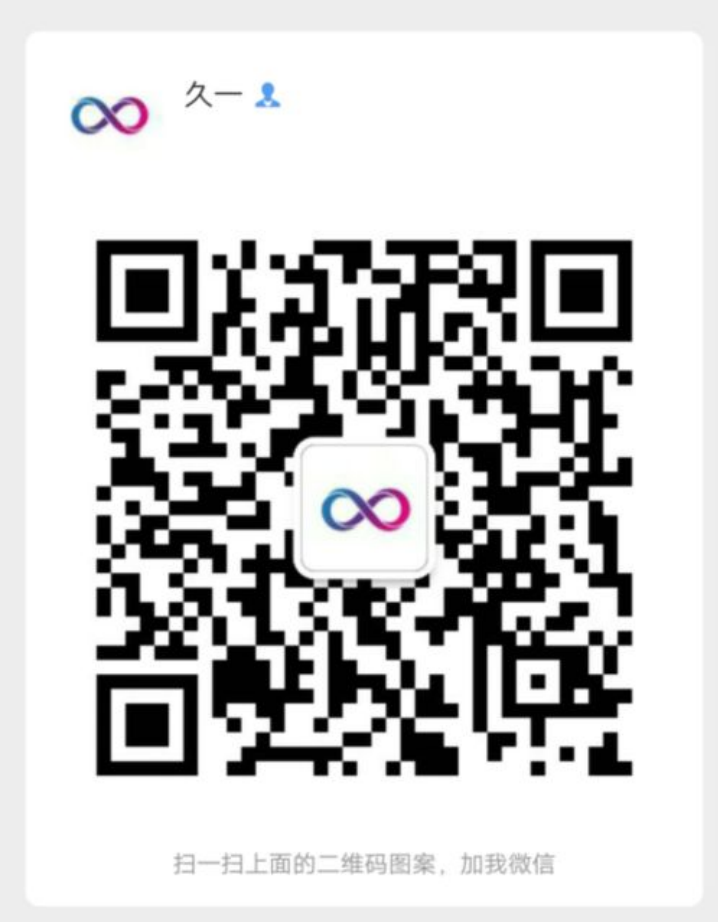
第一，它使用 proto 编译器，自动进行序列化和反序列化，速度非常快，应该比 XML 和 JSON 快上了 20~100 倍；

第二，它的数据压缩效果好，就是说它序列化后的数据量体积小。因为体积小，传输起来带宽和速度上会有优化。

【END】

如果看到这里，说明你喜欢这篇文章，请**转发、点赞**。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓扫描二维码进群↓



推荐阅读

1. Spring Boot：启动原理解析
2. 一键下载 Pornhub 视频！
3. Spring Boot 多模块项目实践（附打包方法）
4. 一个女生不主动联系你还有机会吗？
5. 团队开发中 Git 最佳实践

喜欢文章，点个在看

阅读原文

Dubbo 爆出严重漏洞! 可远程执行恶意代码！（附解决方案）

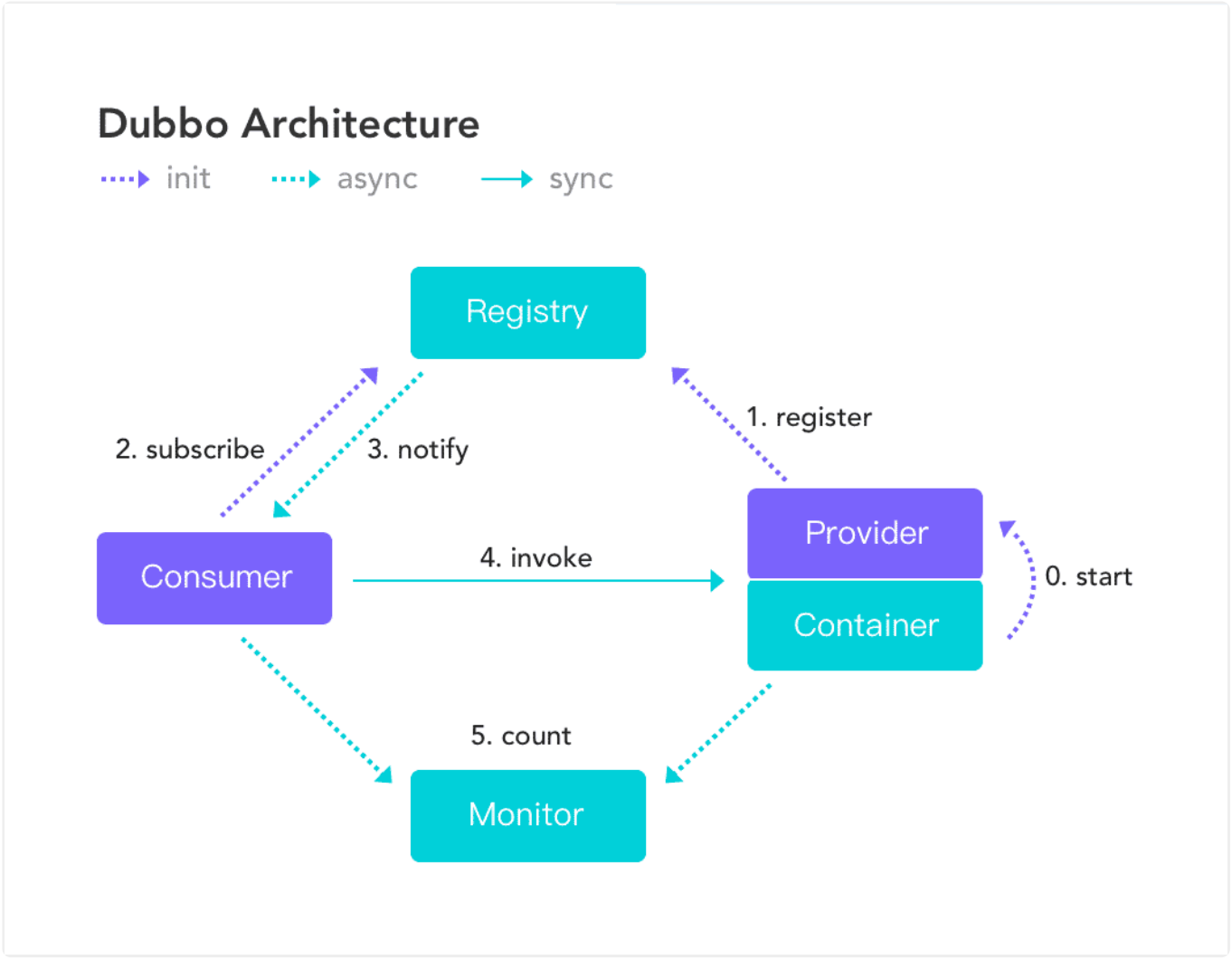
安全客 Java后端 2月17日

链接：https | www.anquanke.com/post/id/198747

近日检测到Apache Dubbo官方发布了CVE-2019-17564漏洞通告，360灵腾安全实验室判断漏洞等级为高，利用难度低，威胁程度高，影响面大。建议使用用户及时安装最新补丁，以免遭受黑客攻击。

0x00 漏洞概述

Apche Dubbo是一款高性能、轻量级的开源Java RPC框架。它提供了三大核心能力：面向接口的远程方法调用，智能容错和负载均衡以及服务自动注册和发现。



Apache Dubbo支持多种协议，当用户选择http协议进行通信时，Apache Dubbo 在接受远程调用的POST请求的时候会执行一个反序列化的操作，当项目包中存在可用的 gadgets时，由于安全校验不当会导致反序列化执行任意代码。

0x01 漏洞详情

漏洞分析，开始跟踪

```
    public DispatcherServlet() { INSTANCE = this; }

    public static void addHttpHandler(int port, HttpHandler processor) { handlers.put(port, processor); }

    public static void removeHttpHandler(int port) { handlers.remove(port); }

    public static DispatcherServlet getInstance() { return INSTANCE; }

    protected void service(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException
        HttpHandler handler = (HttpHandler)handlers.get(request.getLocalPort());
        if (handler == null) {
            response.sendError( i: 404,  s: "Service not found.");
        } else {
            handler.handle(request, response);
        }

    }
}
```

请求传入 org.apache.dubbo.rpc.protocol.http.HttpProtocol中的 handle

```
162         }
163
164         private class InternalHandler implements HttpHandler {
165     @       private InternalHandler() {
166             }
167
168     @       public void handle(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException {
169                 String uri = request.getRequestURI();
170                 HttpInvokerServiceExporter skeleton = (HttpInvokerServiceExporter)HttpProtocol.this.skeletonMap.get(uri);
171                 if (!request.getMethod().equalsIgnoreCase( anotherString: "POST")) {
172                     response.setStatus(500);
173                 } else {
174                     RpcContext.getContext().setRemoteAddress(request.getRemoteAddr(), request.getRemotePort());
175
176                     try {
177                         skeleton.handleRequest(request, response);
178                     } catch (Throwable var6) {
179                         throw new ServletException(var6);
180                     }
181                 }
182
183             }
184         }
185     }
186
```

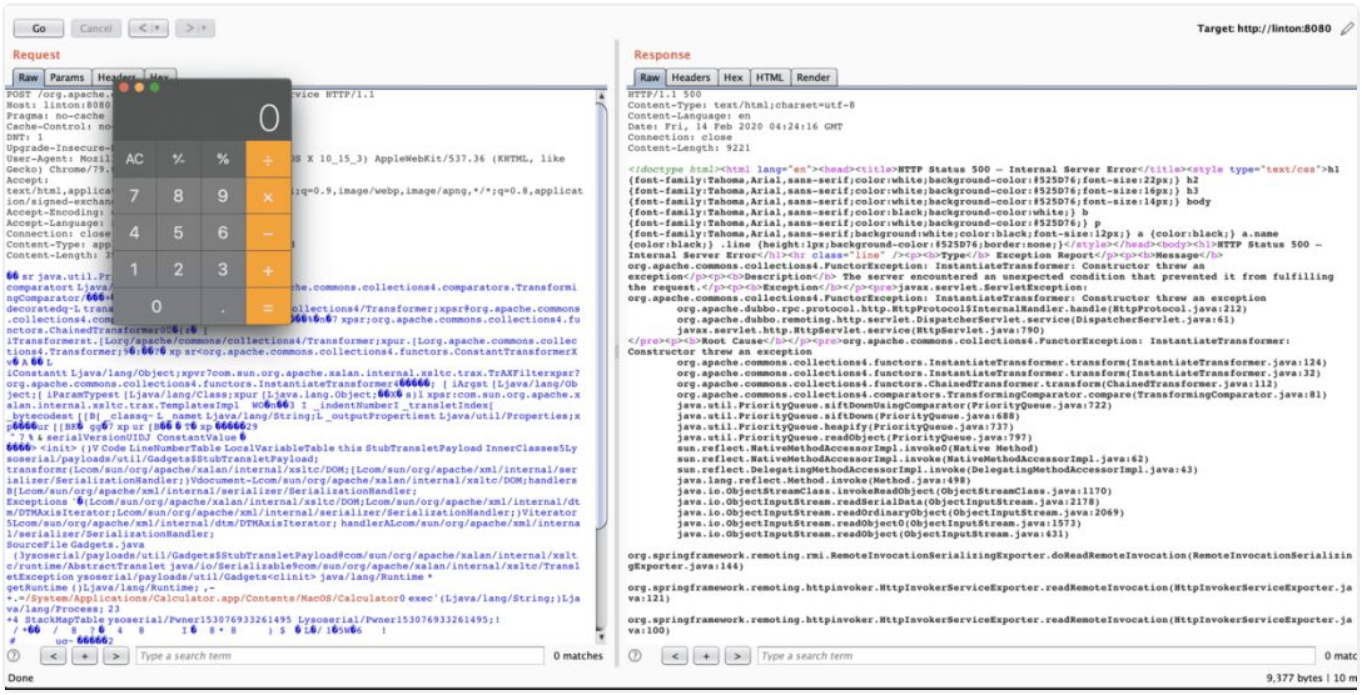
通过进一步跟踪发现其传入 org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter的 readRemoteInvocation

```
34          }
35      }
36
37  @   protected RemoteInvocation readRemoteInvocation(HttpServletRequest request) throws IOException, ClassNotFoundException {
38          return this.readRemoteInvocation(request, request.getInputStream());
39      }
40
41      protected RemoteInvocation readRemoteInvocation(HttpServletRequest request, InputStream is) throws IOException, ClassNotFoundException {
42          ObjectInputStream ois = this.createObjectInputStream(this.decorateInputStream(request, is));
43
44          RemoteInvocation var4;
45          try {
46              var4 = this.doReadRemoteInvocation(ois);
47          } finally {
48              ois.close();
49          }
50
51          return var4;
52      }
53
```

在 org.springframework.remoting.rmi.RemoteInvocationSerializingExporter中，报文中post data部分为ois，全程并没有做任何安全过滤和检查，直接进行 readObject方法

```
        return new CodebaseAwareObjectInputStream(is, this.getBeanClassLoader(), this.isAcceptProxyClasses());
    }

    protected RemoteInvocation doReadRemoteInvocation(ObjectInputStream ois) throws IOException, ClassNotFoundException {
        Object obj = ois.readObject();
        if (!(obj instanceof RemoteInvocation)) {
            throw new RemoteException("Deserialized object needs to be assignable to type [" + RemoteInvocation.class.getName() + "]: '
        } else {
            return (RemoteInvocation)obj;
        }
    }

    protected ObjectOutputStream createObjectOutputStream(OutputStream os) throws IOException {
```

最终导致命令执行



0x02 影响版本

2.7.0 <= Apache Dubbo <= 2.7.4

2.6.0 <= Apache Dubbo <= 2.6.7

Apache Dubbo = 2.5.x

0x03 漏洞检测

仅影响在漏洞版本内启用http协议的用户：<dubbo：protocolname=“http”/>

0x03 处置建议

1、 建议用户升级到2.7.5以上：https://github.com/apache/dubbo/releases/tag/dubbo-2.7.5

2、升级方法

Maven dependency

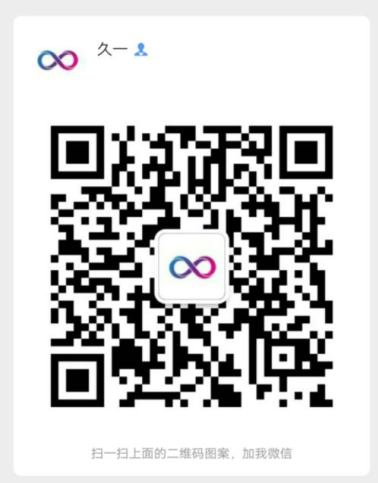
```
<properties>
<dubbo.version>2.7.5</dubbo.version>
</properties>

<dependencies>
<dependency>
<groupId>org.apache.dubbo</groupId>
<artifactId>dubbo</artifactId>
<version>${dubbo.version}</version>
</dependency>

<dependency>
<groupId>org.apache.dubbo</groupId>
<artifactId>dubbo-dependencies-zookeeper</artifactId>
<version>${dubbo.version}</version>
<type>pom</type>
</dependency>

</dependencies>
```

详细升级过程可参考官方的文档:https://github.com/apache/dubbo

-END-

如果看到这里，说明你喜欢这篇文章，请**转发、点赞**。微信搜索「web_resource」，欢迎添加小编微信「focusoncode」，每日朋友圈更新一篇高质量技术博文（无广告）。

↓扫描二维码添加小编↓



推荐阅读

1. 揭秘阿里、腾讯、字节跳动在家办公的区别
2. 前后端分离开发，RESTful 接口应该这样设计
3. Spring Boot + Vue 如此强大?
4. 如何获取靠谱的新型冠状病毒疫情

Dubbo 面试 18 问！ 这些你都会吗?

Dean Wang Java后端 2019-09-10

作者:Dean Wang

dubbo是什么

> dubbo是一个分布式框架，远程服务调用的分布式框架，其核心部分包含：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。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。自动发现：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能做什么

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。软负载均衡及容错机制，可在内网替代F5等硬件负载均衡器，降低成本，减少单点。服务自动注册与发现，不再需要写死服务提供方地址，注册中心基于接口名查询服务提供者的IP地址，并且能够平滑添加或删除服务提供者。

1、默认使用的是什么通信框架，还有别的选择吗?

答：默认也推荐使用 netty 框架，还有 mina。

2、服务调用是阻塞的吗?

答：默认是阻塞的，可以异步调用，没有返回值的可以这么做。

3、一般使用什么注册中心？还有别的选择吗?

答：推荐使用 zookeeper 注册中心，还有 Multicast注册中心, Redis注册中心, Simple注册中心.

ZooKeeper的节点是通过像树一样的结构来进行维护的，并且每一个节点通过路径来标示以及访问。除此之外，每一个节点还拥有自身的一些信息，包括：数据、数据长度、创建时间、修改时间等等。

4、默认使用什么序列化框架，你知道的还有哪些?

答：默认使用 Hessian 序列化，还有 Duddo、FastJson、Java 自带序列化。 hessian是一个采用二进制格式传输的服务框架，相对传统soap web service，更轻量，更快速。

Hessian原理与协议简析：

http的协议约定了数据传输的方式，hessian也无法改变太多：

1) hessian中client与server的交互，基于http-post方式。

2) hessian将辅助信息，封装在http header中，比如“授权token”等，我们可以基于http-header来封装关于“安全校验”“meta数据”等。hessian提供了简单的”校验”机制。

3) 对于hessian的交互核心数据，比如“调用的方法”和参数列表信息，将通过post请求的body体直接发送，格式为字节流。

4) 对于hessian的server端响应数据，将在response中通过字节流的方式直接输出。

hessian的协议本身并不复杂，在此不再赘言；所谓协议(protocol)就是约束数据的格式，client按照协议将请求信息序列化成字节序列发送给server端，server端根据协议，将数据反序列化成“对象”，然后执行指定的方法，并将方法的返回值再次按照协议序列化成字节流，响应给client，client按照协议将字节流反序列话成”对象”。

5、服务提供者能实现失效踢出是什么原理?

答：服务失效踢出基于 zookeeper 的临时节点原理。

6、服务上线怎么不影响旧版本?

答：采用多版本开发，不影响旧版本。在配置中添加version来作为版本区分

7、如何解决服务调用链过长的问题?

答：可以结合 zipkin 实现分布式服务追踪。

8、说说核心的配置有哪些?

核心配置有：

1) dubbo:service/

2) dubbo:reference/

3) dubbo:protocol/

4) dubbo:registry/

5) dubbo:application/

6) dubbo:provider/

7) dubbo:consumer/

8) dubbo:method/

9、dubbo 推荐用什么协议?

答：默认使用 dubbo 协议。

10、同一个服务多个注册的情况下可以直连某一个服务吗?

答：可以直连，修改配置即可，也可以通过 telnet 直接某个服务。

11、dubbo 在安全机制方面如何解决的?

dubbo 通过 token 令牌防止用户绕过注册中心直连，然后在注册中心管理授权，dubbo 提供了黑白名单，控制服务所允许的调用方。

12、集群容错怎么做?

答：读操作建议使用 Failover 失败自动切换，默认重试两次其他服务器。写操作建议使用 Failfast 快速失败，发一次调用失败就立即报错。

13、在使用过程中都遇到了些什么问题？如何解决的?

1) 同时配置了 XML 和 properties 文件，则 properties 中的配置无效

只有 XML 没有配置时，properties 才生效。

2) dubbo 缺省会在启动时检查依赖是否可用，不可用就抛出异常，阻止 spring 初始化完成，check 属性默认为 true。

测试时有些服务不关心或者出现了循环依赖，将 check 设置为 false

3) 为了方便开发测试，线下有一个所有服务可用的注册中心，这时，如果有一个正在开发中的服务提供者注册，可能会影响消费者不能正常运行。

解决：让服务提供者开发方，只订阅服务，而不注册正在开发的服务，通过直连测试正在开发的服务。设置 dubbo:registry 标签的 register 属性为 false。

4) spring 2.x 初始化死锁问题。

在 spring 解析到 dubbo:service 时，就已经向外暴露了服务，而 spring 还在接着初始化其他 bean，如果这时有请求进来，并且服务的实现类里有调用 applicationContext.getBean() 的用法。getBean 线程和 spring 初始化线程的锁的顺序不一样，导致了线程死锁，不能提供服务，启动不了。

解决：不要在服务的实现类中使用 applicationContext.getBean(); 如果不想依赖配置顺序，可以将 dubbo:provider 的 deplay 属性设置为 -1，使 dubbo 在容器初始化完成后再暴露服务。

5) 服务注册不上

检查 dubbo 的 jar 包有没有在 classpath 中，以及有没有重复的 jar 包

检查暴露服务的 spring 配置有没有加载

在服务提供者机器上测试与注册中心的网络是否通

6) 出现 RpcException: No provider available for remote service 异常

表示没有可用的服务提供者，

a. 检查连接的注册中心是否正确

b. 到注册中心查看相应的服务提供者是否存在

c. 检查服务提供者是否正常运行

7) 出现” 消息发送失败” 异常

通常是接口方法的传入传出参数未实现 Serializable 接口。

14、dubbo 和 dubbox 之间的区别?

答：dubbox 是当当网基于 dubbo 上做了一些扩展，如加了服务可 restful 调用，更新了开源组件等。

15、你还了解别的分布式框架吗?

答：别的还有 spring 的 spring cloud，facebook 的 thrift，twitter 的 finagle 等。

16、Dubbo 支持哪些协议，每种协议的应用场景，优缺点?

dubbo：单一长连接和 NIO 异步通讯，适合大并发小数据量的服务调用，以及消费者远大于提供者。 传输协议 TCP，异步，Hessian 序列化；

rmi：采用 JDK 标准的 rmi 协议实现，传输参数和返回参数对象需要实现 Serializable 接口，使用 java 标准序列化机制，使用阻塞式短连接，传输数据包大小混合，消费者和提供者个数差不多，可传文件，传输协议 TCP。 多个短连接，TCP 协议传输，同步传输，适用常规的远程服务调用和 rmi 互操作。在依赖低版本的 Common-Collections 包，java 序列化存在安全漏洞； 关注微信公众号「web_resourc」,回复Java 领取2019最新资源。

webservice:基于 WebService 的远程调用协议，集成 CXF 实现，提供和原生 WebService 的互操作。 多个短连接，基于 HTTP 传输，同步传输，适用系统集成和跨语言调用；http：基于 Http 表单提交的远程调用协议，使用 Spring 的 HttpInvoke 实现。 多个短连接，传输协议 HTTP，传入参数大小混合，提供者个数多于消费者，需要给应用程序和浏览器 JS 调用； hessian：集成 Hessian 服务，基于 HTTP 通讯，采用 Servlet 暴露服务，Dubbo 内嵌 Jetty 作为服务器时默认实现，提供与 Hession 服务互操作。 多个短连接，同步 HTTP 传输，Hessian 序列化，传入参数较大，提供者大于消费者，提供者压力较大，可传文件；

memcache：基于 memcached 实现的 RPC 协议 redis：基于 redis 实现的 RPC 协议

17、Dubbo 集群的负载均衡有哪些策略

Dubbo 提供了常见的集群策略实现，并预扩展点予以自行实现。

Random LoadBalance: 随机选取提供者策略，有利于动态调整提供者权重。 截面碰撞率高，调用次数越多，分布越均匀；关注微信公众号「web_resourc」,回复 Java 领取2019最新资源。

RoundRobin LoadBalance: 轮循选取提供者策略，平均分布，但是存在请求累积的问题；

LeastActive LoadBalance: 最少活跃调用策略，解决慢提供者接收更少的请求； ConstantHash LoadBalance: 一致性 Hash 策略，使相同参数请求总是发到同一提供者，一台机器宕机，可以基于虚拟节点，分摊至其他提供者，避免引起提供者的剧烈变动；

18、服务调用超时问题怎么解决

dubbo在调用服务不成功时，默认是会重试两次的。这样在服务端的处理时间超过了设定的超时时间时，就会有重复请求，比如在发邮件时，可能就会发出多份重复邮件，执行注册请求时，就会插入多条重复的注册数据，那么怎么解决超时问题呢？如下

对于核心的服务中心，去除dubbo超时重试机制，并重新评估设置超时时间。业务处理代码必须放在服务端，客户端只做参数验证和服务调用，不涉及业务流程处理 全局配置实例

```
<dubbo:provider delay="-1" timeout="6000" retries="0"/>
```

当然Dubbo的重试机制其实是非常好的QOS保证，它的路由机制，是会帮你把超时的请求路由到其他机器上，而不是本机尝试，所以 dubbo 的重试机器也能一定程度的保证服务的质量。但是请一定要综合线上的访问情况，给出综合的评估。

> **原文链接：**
> https://deanwang1943.github.io/bugs/2018/10/05/面试/饿了么/dubbo 面试题/

如果喜欢本篇文章，欢迎转发、点赞。关注订阅号「Web项目聚集地」，回复「进群」即可进入无广告技术交流。

推荐阅读

1. 史上最烂的项目：苦撑12年，600 多万行代码 ...
2. 请给 Spring Boot 多一些内存
3. 如何从零搭建百亿流量系统？
4. 惊了！原来 Web 发展历史是这样的

Web项目聚集地

微信扫描二维码，关注我的公众号

喜欢文章，点个在看

阅读原文

Dubbo 面试题

Java后端 2019-11-23

dubbo是什么

dubbo是一个分布式框架，远程服务调用的分布式框架，其核心部分包含：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。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。自动发现：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能做什么

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。软负载均衡及容错机制，可在内网替代F5等硬件负载均衡器，降低成本，减少单点。服务自动注册与发现，不再需要写死服务提供方地址，注册中心基于接口名查询服务提供者的IP地址，并且能够平滑添加或删除服务提供者。

1、默认使用的是什么通信框架，还有别的选择吗？

答：默认也推荐使用 netty 框架，还有 mina。

2、服务调用是阻塞的吗？

答：默认是阻塞的，可以异步调用，没有返回值的可以这么做。

3、一般使用什么注册中心？还有别的选择吗？

答：推荐使用 zookeeper 注册中心，还有 Multicast注册中心，Redis注册中心，Simple注册中心

ZooKeeper的节点是通过像树一样的结构来进行维护的，并且每一个节点通过路径来标示以及访问。除此之外，每一个节点还拥有自身的一些信息，包括：数据、数据长度、创建时间、修改时间等等。

4、默认使用什么序列化框架，你知道的还有哪些？

答：默认使用 Hessian 序列化，还有 Duddo、FastJson、Java 自带序列化。hessian是一个采用二进制格式传输的服务框架，相对传统soap web service，更轻量，更快速。

Hessian原理与协议简析：

http的协议约定了数据传输的方式，hessian也无法改变太多：

1) hessian中client与server的交互，基于http-post方式。

2) hessian将辅助信息，封装在http header中，比如“授权token”等，我们可以基于http-header来封装关于“安全校验”“meta数据”等。hessian提供了简单的”校验”机制。

3) 对于hessian的交互核心数据，比如“调用的方法”和参数列表信息，将通过post请求的body体直接发送，格式为字节流。

4) 对于hessian的server端响应数据，将在response中通过字节流的方式直接输出。

hessian的协议本身并不复杂，在此不再赘言；所谓协议(protocol)就是约束数据的格式，client按照协议将请求信息序列化成字节序列发送给server端，server端根据协议，将数据反序列化成“对象”，然后执行指定的方法，并将方法的返回值再次按照协议序列化

成字节流，响应给client，client按照协议将字节流反序列话成”对象”。

5、服务提供者能实现失效踢出是什么原理?

答：服务失效踢出基于 zookeeper 的临时节点原理。

6、服务上线怎么不影响旧版本?

答：采用多版本开发，不影响旧版本。在配置中添加version来作为版本区分

7、如何解决服务调用链过长的问题?

答：可以结合 zipkin 实现分布式服务追踪。

8、说说核心的配置有哪些?

核心配置有：

1) dubbo:service/

2) dubbo:reference/

3) dubbo:protocol/

4) dubbo:registry/

5) dubbo:application/

6) dubbo:provider/

7) dubbo:consumer/

8) dubbo:method/

9、dubbo 推荐用什么协议?

答：默认使用 dubbo 协议。

10、同一个服务多个注册的情况下可以直连某一个服务吗?

答：可以直连，修改配置即可，也可以通过 telnet 直接某个服务。

11、dubbo 在安全机制方面如何解决的?

dubbo 通过 token 令牌防止用户绕过注册中心直连，然后在注册中心管理授权，dubbo 提供了黑白名单，控制服务所允许的调用方。

12、集群容错怎么做?

答：读操作建议使用 Failover 失败自动切换，默认重试两次其他服务器。写操作建议使用 Failfast 快速失败，发一次调用失败就立即报错

13、在使用过程中都遇到了些什么问题?如何解决的?

1) 同时配置了 XML 和 properties 文件，则 properties 中的配置无效

只有 XML 没有配置时，properties 才生效。

2) dubbo 缺省会在启动时检查依赖是否可用，不可用就抛出异常，阻止 spring 初始化完成，check 属性默认为 true。

测试时有些服务不关心或者出现了循环依赖，将 check 设置为 false

3) 为了方便开发测试，线下有一个所有服务可用的注册中心，这时，如果有一个正在开发中的服务提供者注册，可能会影响消费者不能正常运行。

解决：让服务提供者开发方，只订阅服务，而不注册正在开发的服务，通过直连测试正在开发的服务。设置 dubbo:registry 标签的 register 属性为 false。

4) spring 2.x 初始化死锁问题。

在 spring 解析到 dubbo:service 时，就已经向外暴露了服务，而 spring 还在接着初始化其他 bean，如果这时有请求进来，并且服务的实现类里有调用 applicationContext.getBean() 的用法。getBean 线程和 spring 初始化线程的锁的顺序不一样，导致了线程死锁，不能提供服务，启动不了。

解决：不要在服务的实现类中使用 applicationContext.getBean(); 如果不想依赖配置顺序，可以将 dubbo:provider 的 deplay 属性设置为 - 1，使 dubbo 在容器初始化完成后再暴露服务。

5) 服务注册不上

检查 dubbo 的 jar 包有没有在 classpath 中，以及有没有重复的 jar 包

检查暴露服务的 spring 配置有没有加载

在服务提供者机器上测试与注册中心的网络是否通

6) 出现 RpcException: No provider available for remote service 异常

表示没有可用的服务提供者，

a. 检查连接的注册中心是否正确

b. 到注册中心查看相应的服务提供者是否存在

c. 检查服务提供者是否正常运行

7) 出现” 消息发送失败” 异常

通常是接口方法的传入传出参数未实现 Serializable 接口。

14、dubbo 和 dubbox 之间的区别?

答：dubbox 是当当网基于 dubbo 上做了一些扩展，如加了服务可 restful 调用，更新了开源组件等。

15、你还了解别的分布式框架吗?

答：别的还有 spring 的 spring cloud，facebook 的 thrift，twitter 的 finagle 等。

16、Dubbo 支持哪些协议，每种协议的应用场景，优缺点?

dubbo：单一长连接和 NIO 异步通讯，适合大并发小数据量的服务调用，以及消费者远大于提供者。传输协议 TCP，异步，Hessian 序列化；

rmi：采用 JDK 标准的 rmi 协议实现，传输参数和返回参数对象需要实现 Serializable 接口，使用 java 标准序列化机制，使用阻塞式短连接，传输数据包大小混合，消费者和提供者个数差不多，可传文件，传输协议 TCP。多个短连接，TCP 协议传输，同步传输，适用常规的远程服务调用和 rmi 互操作。在依赖低版本的 Common-Collections 包，java 序列化存在安全漏洞；

webservice:基于 WebService 的远程调用协议，集成 CXF 实现，提供和原生 WebService 的互操作。多个短连接，基于 HTTP 传输，同步传输，适用系统集成和跨语言调用；http：基于 Http 表单提交的远程调用协议，使用 Spring 的 HttpInvoke 实现。多个短连接，传输协议 HTTP，传入参数大小混合，提供者个数多于消费者，需要给应用程序和浏览器 JS 调用；hessian：集成 Hessian 服务，基于 HTTP 通讯，采用 Servlet 暴露服务，Dubbo 内嵌 Jetty 作为服务器时默认实现，提供与 Hession 服务互操作。多个短连接，同步 HTTP 传输，Hessian 序列化，传入参数较大，提供者大于消费者，提供者压力较大，可传文件；

memcache：基于 memcached 实现的 RPC 协议 redis：基于 redis 实现的 RPC 协

17、Dubbo 集群的负载均衡有哪些策略

Dubbo 提供了常见的集群策略实现，并预扩展点予以自行实现。

Random LoadBalance: 随机选取提供者策略，有利于动态调整提供者权重。截面碰撞率高，调用次数越多，分布越均匀；

RoundRobin LoadBalance: 轮循选取提供者策略，平均分布，但是存在请求累积的问题；

LeastActive LoadBalance: 最少活跃调用策略，解决慢提供者接收更少的请求；ConstantHash LoadBalance: 一致性 Hash 策略，使相同参数请求总是发到同一提供者，一台机器宕机，可以基于虚拟节点，分摊至其他提供者，避免引起提供者的剧烈变动；

18、服务调用超时问题怎么解决

dubbo在调用服务不成功时，默认是会重试两次的。这样在服务端的处理时间超过了设定的超时时间时，就会有重复请求，比如在发邮件时，可能就会发出多份重复邮件，执行注册请求时，就会插入多条重复的注册数据，那么怎么解决超时问题呢？如下对于核心的服务中心，去除dubbo超时重试机制，并重新评估设置超时时间。业务处理代码必须放在服务端，客户端只做参数验证和服务调用，不涉及业务流程处理 全局配置实例

```
<dubbo:provider delay="-1" timeout="6000" retries="0"/>
```

当然Dubbo的重试机制其实是非常好的QOS保证，它的路由机制，是会帮你把超时的请求路由到其他机器上，而不是本机尝试，所以 dubbo的重试机器也能一定程度的保证服务的质量。但是请一定要综合线上的访问情况，给出综合的评估。

【END】

推荐阅读

1. 我采访了一位 Pornhub 工程师，聊了这些纯纯的话题
2. 常见排序算法总结 - Java 实现
3. Java：如何更优雅的处理空值？
4. MySQL：数据库优化，可以看看这篇文章
5. 团队开发中 Git 最佳实践

喜欢文章，点个在看

「附源码」Dubbo+Zookeeper 的 RPC 远程调用框架

码农云帆哥 Java后端 2019-10-11

作者 | 码农云帆哥

链接 | blog.csdn.net/sinat_27933301

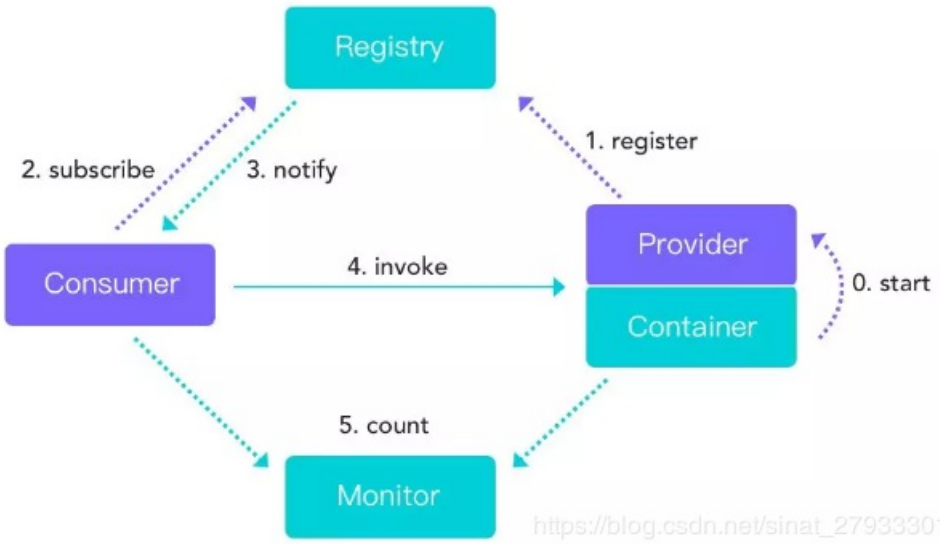
上一篇:从零搭建创业公司后台技术栈

这是一个基于Dubbo+Zookeeper 的 RPC 远程调用框架 demo，希望读者可以通过这篇文章大概能看懂这一个简单的框架搭建。

Demo 源码获取方式：关注微信公众号「Java后端」，回复「DZ」获取

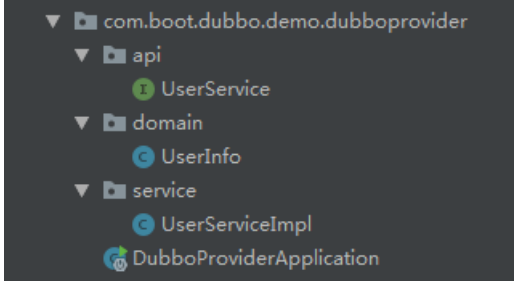
Dubbo是阿里巴巴公司开源的一个高性能优秀的服务框架，使得应用可通过高性能的 RPC 实现服务的输出和输入功能，可以和Spring框架无缝集成。

Dubbo是一款高性能、轻量级的开源Java RPC框架，它提供了三大核心能力：面向接口的远程方法调用，智能容错和负载均衡，以及服务自动注册和发现。微信搜索 web_resource 关注获取更多推送



- 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：服务注册于发现的注册中心
- monitor：统计服务调用次数和调用时间的监控中心
- container：服务运行容器

一、dubbo-provider（服务提供方）



1、Java Bean

为什么要实现Serializable接口？当我们需要把对象的状态信息通过网络进行传输，或者需要将对象的状态信息持久化，以便将来使用时都需要把对象进行序列化。

使用了Lombok，它能通过注解的方式，在编译时自动为属性生成构造器、getter/setter、equals、hashcode、toString方法。

```
@Data
public class UserInfo implements Serializable {
    private String account;
    private String password;
}
```

2、UserService

服务提供方　暴露的服务，将注册到zookeeper上。

```
public interface UserService {
    // 定义用户登录的api
    UserInfo login(UserInfo user)
;
}
```

3、UserServiceImpl

服务提供方暴露的服务对应的实现类。

```
@Component
@Service(interfaceClass = UserService.class)
public class UserServiceImpl implements UserService {
    public UserInfo login(UserInfo user)
{
        UserInfo reUser = new UserInfo()
;
        reUser.setAccount("登录的账号为:"+user.getAccount());
        reUser.setPassword("登录的密码为:"+user.getPassword());

        return reUser;
    }
}
```

4、DubboProviderApplication

```
@SpringBootApplication
@EnableDubboConfiguration // 启用dubbo自动配置
public class DubboProviderApplication {

    public static void main(String[] args)
{
        SpringApplication.run(DubboProviderApplication.class, args);
    }
}
```

5、pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-ins
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4
>
    <modelVersion>4.0.0</modelVersion
>

    <groupId>com.boot.dubbo.demo</groupId
>
    <artifactId>dubbo-provider</artifactId
>
    <version>0.0.1-SNAPSHOT</version
>
    <packaging>jar</packaging
>

    <name>dubbo-provider</name
>
    <description>Demo project for Spring Boot</description
>

    <parent
>
        <groupId>org.springframework.boot</groupId
>
        <artifactId>spring-boot-starter-parent</artifactId
>
        <version>2.0.6.RELEASE</version
>
        <relativePath/> <!-- lookup parent from repository -->
    </parent
>

    <properties
>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding
>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding
```

```
        <java.version>1.8</java.version
    >
    </properties
    >

    <dependencies
    >
        <dependency
        >
            <groupId>org.springframework.boot</groupId
            >
            <artifactId>spring-boot-starter-web</artifactId
            >
        </dependency
        >

        <!--Dubbo依赖-->
        <dependency
        >
            <groupId>com.alibaba.spring.boot</groupId
            >
            <artifactId>dubbo-spring-boot-starter</artifactId
            >
            <version>2.0.0</version
            >
        </dependency
        >

        <!--自动生成getter，setter，equals，hashCode和toString等等-->
        <dependency
        >
            <groupId>org.projectlombok</groupId
            >
            <artifactId>lombok</artifactId
            >
            <version>1.16.20</version
            >
            <scope>provided</scope
            >
        </dependency
        >

        <!--Zookeeper客户端-->
        <dependency
        >
            <groupId>com.101tec</groupId
            >
            <artifactId>zkclient</artifactId
            >
            <version>0.10</version
            >
        </dependency
        >
```

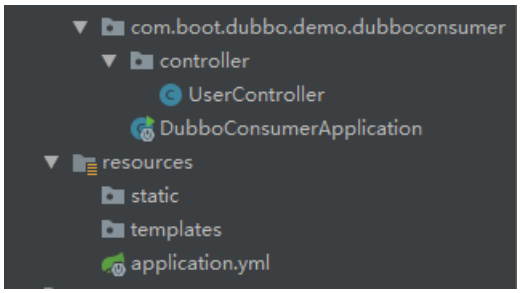
```
        <!--Zookeeper依赖，排除log4j避免依赖冲突-->
        <dependency
>
            <groupId>org.apache.zookeeper</groupId
>
            <artifactId>zookeeper</artifactId
>
            <version>3.4.10</version
>
            <exclusions
>
                <exclusion
>
                    <groupId>org.slf4j</groupId
>
                    <artifactId>slf4j-log4j12</artifactId
>
                </exclusion
>
                <exclusion
>
                    <groupId>log4j</groupId
>
                    <artifactId>log4j</artifactId
>
                </exclusion
>
            </exclusions
>
        </dependency
>

        <dependency
>
            <groupId>org.springframework.boot</groupId
>
            <artifactId>spring-boot-starter-test</artifactId
>
            <scope>test</scope
>
        </dependency
>
    </dependencies
>

    <build
>
        <plugins
>
            <plugin
>
                <groupId>org.springframework.boot</groupId
>
```

```
                <artifactId>spring-boot-maven-plugin</artifactId
>
            </plugin
>
        </plugins
>
    </build
>
</project>
```

6、application.yml

```
spring:
  dubbo
:
    application
:
      name: dubbo-provider
    protocol
:
      name: dubb
o
      port: 2088
0
    registry
:
      address: zookeeper://127.0.0.1:2181
```

二、dubbo-consumer（服务消费方）



1、UserController

```
@RestController
public class UserController {

    @Reference // 引用dubbo服务器提供服务器接口
    private UserService userService;

    @GetMapping("/login"
)
    public UserInfo login(UserInfo userInfo) {
        return userService.login(userInfo);
    }
}
```

2、DubboConsumerApplication

```
@SpringBootApplication
@EnableDubboConfiguration // 启用dubbo自动配置
public class DubboConsumerApplication {

    public static void main(String[] args)
{
        SpringApplication.run(DubboConsumerApplication.class, args);
    }
}
```

3、pom.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-ins
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4
>
    <modelVersion>4.0.0</modelVersion
>

    <groupId>com.boot.dubbo.demo</groupId
>
    <artifactId>dubbo-consumer</artifactId
>
    <version>0.0.1-SNAPSHOT</version
>
    <packaging>jar</packaging
>

    <name>dubbo-consumer</name
>
    <description>Demo project for Spring Boot</description
>

    <parent
>
        <groupId>org.springframework.boot</groupId
>
        <artifactId>spring-boot-starter-parent</artifactId
>
        <version>2.0.6.RELEASE</version
>
        <relativePath/> <!-- lookup parent from repository -->
    </parent
>

    <properties
>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding
>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding
```

```
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding
>
        <java.version>1.8</java.version
>
    </properties
>

    <dependencies
>
        <dependency
>
            <groupId>org.springframework.boot</groupId
>
            <artifactId>spring-boot-starter-web</artifactId
>
        </dependency
>

        <!--依赖dubbo-provider服务提供者-->
        <dependency
>
            <groupId>com.boot.dubbo.demo</groupId
>
            <artifactId>dubbo-provider</artifactId
>
            <version>0.0.1-SNAPSHOT</version
>
        </dependency
>

        <!--Dubbo依赖-->
        <dependency
>
            <groupId>com.alibaba.spring.boot</groupId
>
            <artifactId>dubbo-spring-boot-starter</artifactId
>
            <version>2.0.0</version
>
        </dependency
>

        <!--Zookeeper客户端-->
        <dependency
>
            <groupId>com.101tec</groupId
>
            <artifactId>zkclient</artifactId
>
            <version>0.10</version
>
        </dependency
>

        <!--Zookeeper依赖，排除log4j避免依赖冲突-->
```

```
        <!--zookeeper依赖，排除log4j避免依赖冲突-->
        <dependency
>
            <groupId>org.apache.zookeeper</groupId
>
            <artifactId>zookeeper</artifactId
>
            <version>3.4.10</version
>
            <exclusions
>
                <exclusion
>
                    <groupId>org.slf4j</groupId
>
                    <artifactId>slf4j-log4j12</artifactId
>
                </exclusion
>
                <exclusion
>
                    <groupId>log4j</groupId
>
                    <artifactId>log4j</artifactId
>
                </exclusion
>
            </exclusions
>
        </dependency
>

        <dependency
>
            <groupId>org.springframework.boot</groupId
>
            <artifactId>spring-boot-starter-test</artifactId
>
            <scope>test</scope
>
        </dependency
>
    </dependencies
>

    <build
>
        <plugins
>
            <plugin
>
                <groupId>org.springframework.boot</groupId
>
                <artifactId>spring-boot-maven-plugin</artifactId
```

```
>
            </plugin
>
        </plugins
>
    </build
>
</project>
```

4、application.yml

```
spring:
  dubbo
:
    application
:
      name: dubbo-consumer
    protocol
:
      name: dubb
o
      port: 2088
0
    registry
:
      address: zookeeper://127.0.0.1:2181
server:
  port: 8081
```

三、Zookeeper安装配置

Zookeeper是什么:

https://blog.csdn.net/sinat_27933301/article/details/80101970

Zookeeper的安装配置:

https://blog.csdn.net/sinat_27933301/article/details/84001530

Zookeeper集群环境搭建:

https://blog.csdn.net/sinat_27933301/article/details/84351404

四、项目启动

五、查看Zookeeper服务注册情况

```
WATCHER::

WatchedEvent state:SyncConnected type:None path:null
[zk: localhost:2181(CONNECTED) 0] ls /
[dubbo, zookeeper]
[zk: localhost:2181(CONNECTED) 1] ls /dubbo
[com.boot.dubbo.demo.api.UserService]
[zk: localhost:2181(CONNECTED) 2]
```

六、通过浏览器发送请求（dubbo-provider的服务注册到Zookeeper上，dubbo-consumer消费者可以调用注册的服务）。

到此，rpc远程调用服务通了！感兴趣的话可以去学习下。

Demo 源码获取方式：关注微信公众号「Java后端」，回复「DZ」获取

-END-

如果看到这里，说明你喜欢这篇文章，请**转发、点赞**。微信搜索「web_resource」，关注后回复「进群」或者扫描下方二维码即可进入无广告交流群。

↓扫描二维码进群↓

推荐阅读

1. 从零搭建创业公司后台技术栈
2. 如何阅读 Java 源码？
3. 某小公司RESTful、前后端分离的实践
4. 该如何弥补 GitHub 功能缺陷？
5. 团队开发中 Git 最佳实践

喜欢文章，点个在看

阅读原文

一场近乎完美基于 Dubbo 的微服务改造实践

Java后端 1月3日

以下文章来源于阿里巴巴中间件，作者网易考拉 陶杨

作者：网易考拉 陶杨

微信公众号：阿里巴巴中间件（ID：Aliware_2018）

导读：

网易考拉（以下简称考拉）是网易旗下以跨境业务为主的综合型电商，自2015年1月9日上线公测后，业务保持了高速增长，这背后离不开其技术团队的支撑。微服务化是电商IT架构演化的必然趋势，网易考拉的服务架构演进也经历了从单体应用走向微服务化的整个过程，以下整理自网易考拉陶杨在近期Apache Dubbo Meetup上的分享，通过该文，您将了解到：

- 考拉架构的演进过程
- 考拉在服务化改造方面的实践
- 考拉在解决注册中心性能瓶颈方面的实践
- 考拉未来的规划

本文相关PPT等资源下载地址：

https://github.com/dubbo/awesome-dubbo/tree/master/slides/meetup/201812%40hangzhou

考拉架构的演进过程

考拉在2015年初上线的时候，线上只有七个工程，商品详情页、购物车下单页等都耦合在中间这个online的工程里面。

在上线之初的时候，这种架构还是比较有优势的，因为当时考拉的开发人员也不是很多，把所有的功能都耦合在一个进程里面，利于集中开发、测试和上线，是一种比较高效和节省成本的方式。

但是随着业务的不断发展，包括需求的逐步增多，开发团队的不断扩容，这时候，单体架构的一些劣势就逐渐的暴露出来了，例如**开发效率低**：功能之间的相互耦合，不同需求的不同分支也经常会修改同一块代码，导致合代码的过程非常痛苦，而且经常会出

问题。

再例如**上线成本高**：几乎所有的发布需求都会涉及到这些应用的上线，同时不断增长的业务需求，也会使得我们的代码越来越臃肿，造成维护困难、可用性差，功能之间相互耦合，都耦合在一个进程里面，导致一旦某一个业务需求涉及的代码或者资源出现问题，那么就会影响其他的业务。比如说我们曾经在online工程里面，因为优惠券兑换热点的问题，影响了核心的下单服务。

这个架构在考拉运行的4到5个月的时间里，从开发到测试再到上线，大家都特别痛苦。所以我们就开始进行了服务化拆分的工作。

这个是考拉现在的分布式服务架构。伴随着服务化的拆分，我们的组织架构也进行了很多调整，出现了商品中心、用户中心和订单中心等等。拆分其实是由业务驱动的，通过业务来进行一些横向拆分或者纵向拆分，同时，拆分也会面对一个拆分粒度的问题，比如怎么才算一个服务，或者说服务拆的过细，是不是会导致我们管理成本过高，又或者说是否会带来架构上的新问题。

考拉的拆分由粗到细是一个逐步演进的过程。随着服务化的拆分，使得服务架构越来越复杂，随之而来产生了各种各样的公共技术，比如说服务治理、平台配置中心、分布式事务和分布式定时任务等等。

考拉的服务化实践

微服务框架在服务化中起到了很重要的作用，是服务化改造的基石，经过严格的技术选型流程后，我们选用了Dubbo来作为考拉服务改造的一个重要支柱。Dubbo可以解决服务化过程中服务的定义、服务的注册与发现、服务的调用和路由等问题，此外，Dubbo也具有一些服务治理的功能和服务监控的功能。下面我将介绍考拉基于Dubbo做的一些服务化实践。

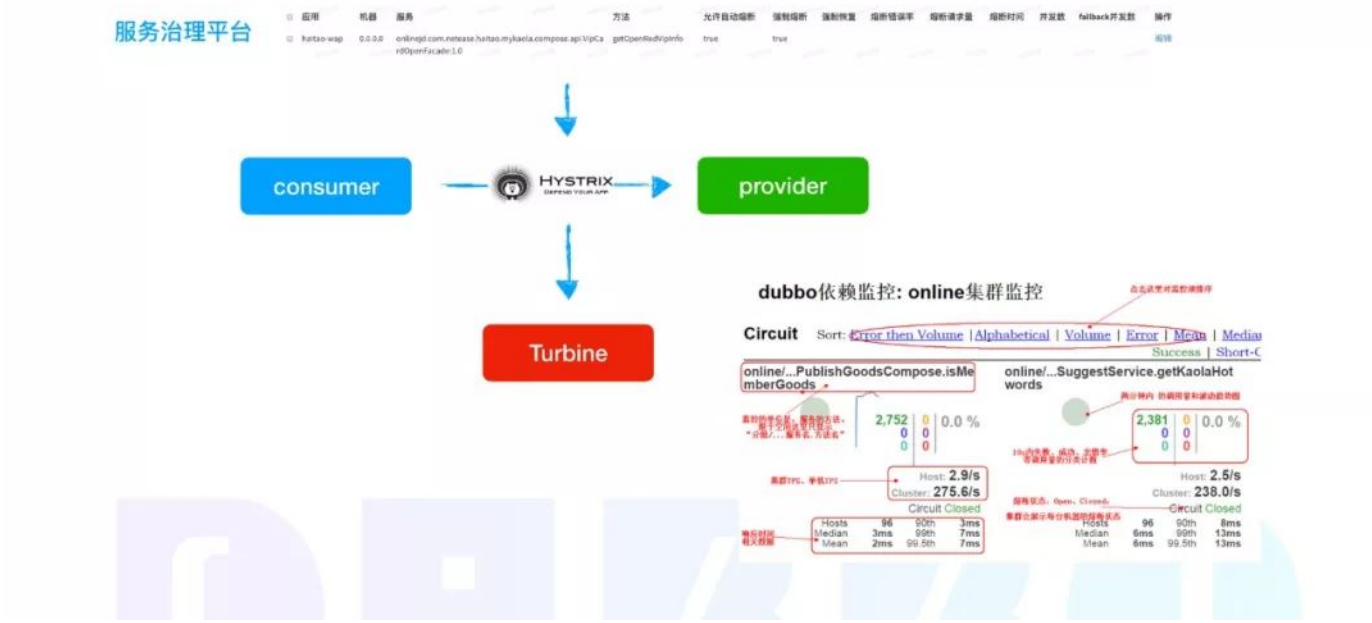
首先来说一下 熔断。

在进行服务化拆分之后，应用中原有的本地调用就会变成远程调用，这样就引入了更多的复杂性。比如说服务A依赖于服务B，这个过程中可能会出现网络抖动、网络异常，或者说服务B变得不可用或者不好用时，也会影响到A的服务性能，甚至可能会使得服务A占满整个线程池，导致这个应用上其它的服务也受影响，从而引发更严重的雪崩效应。

因此，服务之间有这样一种依赖关系之后，需要意识到服务的依赖其实是不稳定的。此时，需要通过采取一些服务治理的措施，例

如熔断、降级、限流、隔离和超时等，来保障应用不被外部的异常拖垮。Dubbo提供了降级的特性，比如可以通过mock参数来配置一些服务的失败降级或者强制降级，但是Dubbo缺少自动熔断的特性，所以我们在Dubbo上引入了Hystrix。

熔断



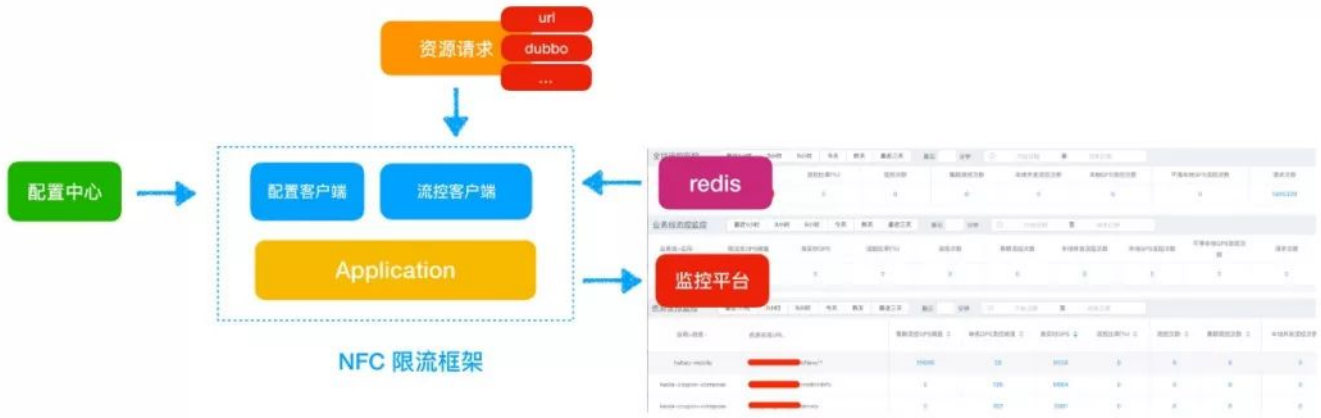
消费者在进行服务调用的时候会经过熔断器，当服务提供者出现异常的时候，比如暂时性的不可用，熔断器就会打开，对消费端进行调用短路，此时，消费端就不会再发起远程调用，而是直接走向降级逻辑。与此同时，消费端会持续的探测服务的可用性，一旦服务恢复，熔断器就会关闭，重新恢复调用。在Dubbo的服务治理平台上，可以对Hystrix上运行的各种动态参数进行动态的配置，包括是否允许自动熔断，是否要强制熔断，熔断的失败率和时间窗口等等。

下面再说一下 限流。

当用户的请求量，调用超过系统可承受的并发时系统QPS会降低、出现不可用甚至存在宕机的风险。这就需要一个机制来保护我们的系统，当预期并发超过系统可承受的范围时，进行快速失败、直接返回，以保护系统。

Dubbo提供了一些基础的限流特性，例如可以通过信号量的配置来限制我们消费者的调用并发，或者限制提供者的执行并发。但是这些是远远不够的，考拉自研了限流框架NFC，并基于Dubbo filter 的形式，实现了对 Dubbo的支持，同时也支持对URL等其他资源的限流。通过配置中心动态获取流控规则，对于资源的请求，比如Dubbo调用会经过流控客户端，进行处理并判断是否触发限流，一旦请求超出定义的阈值，就会快速失败。

限流



同时，这些限流的结果会上报到监控平台。上图中的页面就是考拉流控平台的一个监控页面，我们在页面上可以对每一个资源（URL、Dubbo接口）进行一个阈值的配置，并对限流进行准实时监控，包括流控比率、限流次数和当前的QPS等。限流框架除了实现基本的并发限流之外，也基于令牌桶和漏桶算法实现了QPS限流，并基于Redis实现了集群级别的限流。这些措施保障系统在高流量的情况下不会被打垮。

考拉在监控服务方面的改造

在服务化的过程中，系统变得越来越复杂，服务数量变得越来越多，此时需要引入更多维度的监控功能，帮助快速的去定位并解决系统中的各类问题。监控主要分为这四个方面，日志、Metrics、Trace和HealthCheck。

监控

在应用程序、操作系统运行的时候，都会产生各种各样的日志，通过日志平台对这些日志进行采集、分析和展示，并支持查询和操作。Metrics反映的是系统运行的基本状态，包括瞬时值或者聚合值，例如系统的CPU使用率、磁盘使用率，以及服务调用过程中的平均延时等。Trace是对服务调用链的一个监控，例如调用过程中的耗时分析、瓶颈分析、依赖分析和异常分析等。Healthcheck可以探测应用是否准备就绪，是否健康，或者是否还存活。

接下来，围绕Dubbo来介绍一下考拉在监控方面的改造实践。

第一个是服务监控。

Dubbo提供了服务监控功能，支持定期上报服务监控数据，通过代码增强的方式，采集Dubbo调用数据，存储到时序数据库里面，将Dubbo的调用监控功能接入到考拉自己的监控平台。

上图中的页面是对Dubbo提供者的服务监控，包括对服务接口、源集群等不同维度的监控，除了全局的调用监控，还包括不同维度的监控，例如监控项里的调用次数。有时候我们更关心慢请求的情况，所以会将响应时间分为多个范围，比如说从0到10毫秒，或是从10到50毫秒等，这样就可以看到在各个范围内请求的数量，从而更好地了解服务质量。

同时，也可以通过各种报警规则，对报警进行定义，当服务调用出现异常时，通过邮件、短信和电话的形式通知相关人员。监控平台也会对异常堆栈进行采集，例如说这次服务调用的异常的原因，是超时还是线程满了的，可以在监控平台上直接看到。同时生成一些监控报表，帮助我们更好地了解服务的性能，推进开发去改进。

第二个是Trace。

Trace

耗时分析

分析调用链中各环节的耗时分布

瓶颈分析

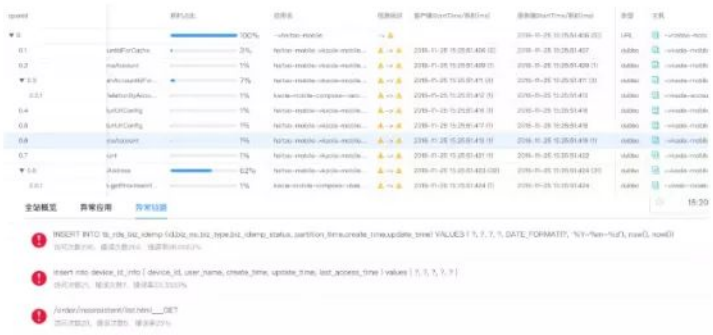
分析调用链中的瓶颈点

依赖分析

分析调用链中的依赖关系是否合理

异常分析

异常链路报警以及排查

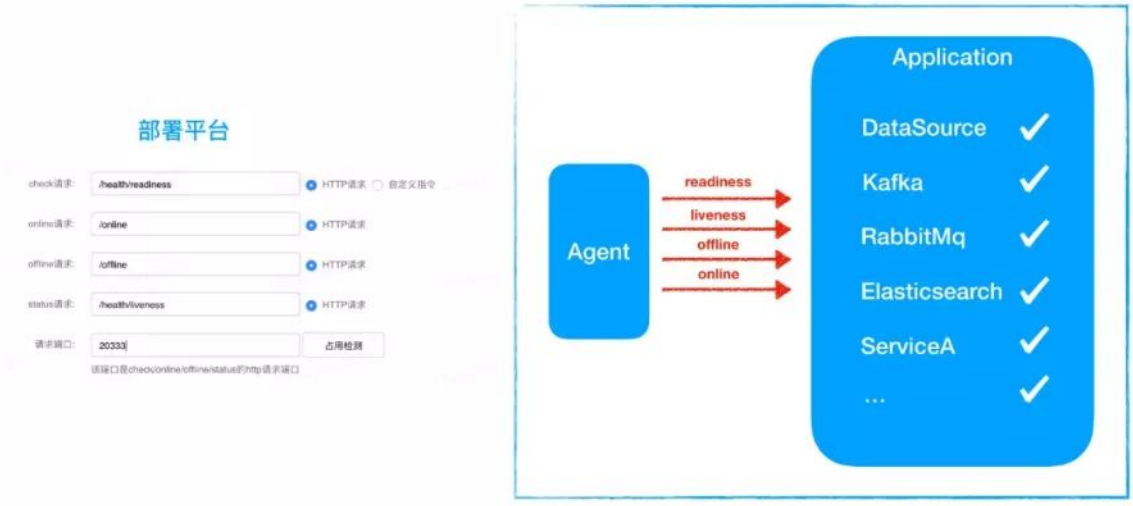


我们参考了Dapper，自研了Trace平台，并通过代码增强的方式，实现了对Dubbo调用链路的采集。相关调用链参数如TarceID，SpanID　等是通过Dubbo的隐式传参来传递的。Trace可以了解在服务调用链路中的一个耗时分析和瓶颈分析等。Trace平台上可以展示一次服务调用，经历了哪些节点，最耗时的那个节点是在哪里，从而可以有针对性的去进行性能优化。Trace还可以进行依赖分析，这些依赖是否合理，能否通过一些业务手段或者其它手段去减少一些不合理的依赖。

Trace对异常链路进行监控报警，及时的探测到系统异常并帮助我们快速的定位问题，同时和日志平台做了打通，通过TraceId可以很快的获取到关联的异常日志。

第三个是健康检查。

健康检查



健康检查也是监控中很重要的一个方面，以更优雅的方式上线应用实例。我们和自动部署平台结合，实现应用的健康检查。服务启动的时候可以通过Readiness接口判断应用依赖的各种资源，包括数据库、消息队列等等是否已经准备就绪。只有健康检查成功的

时候才会触发出注册操作。同时Agent也会在程序运行的过程中定时的检查服务的运行状态。

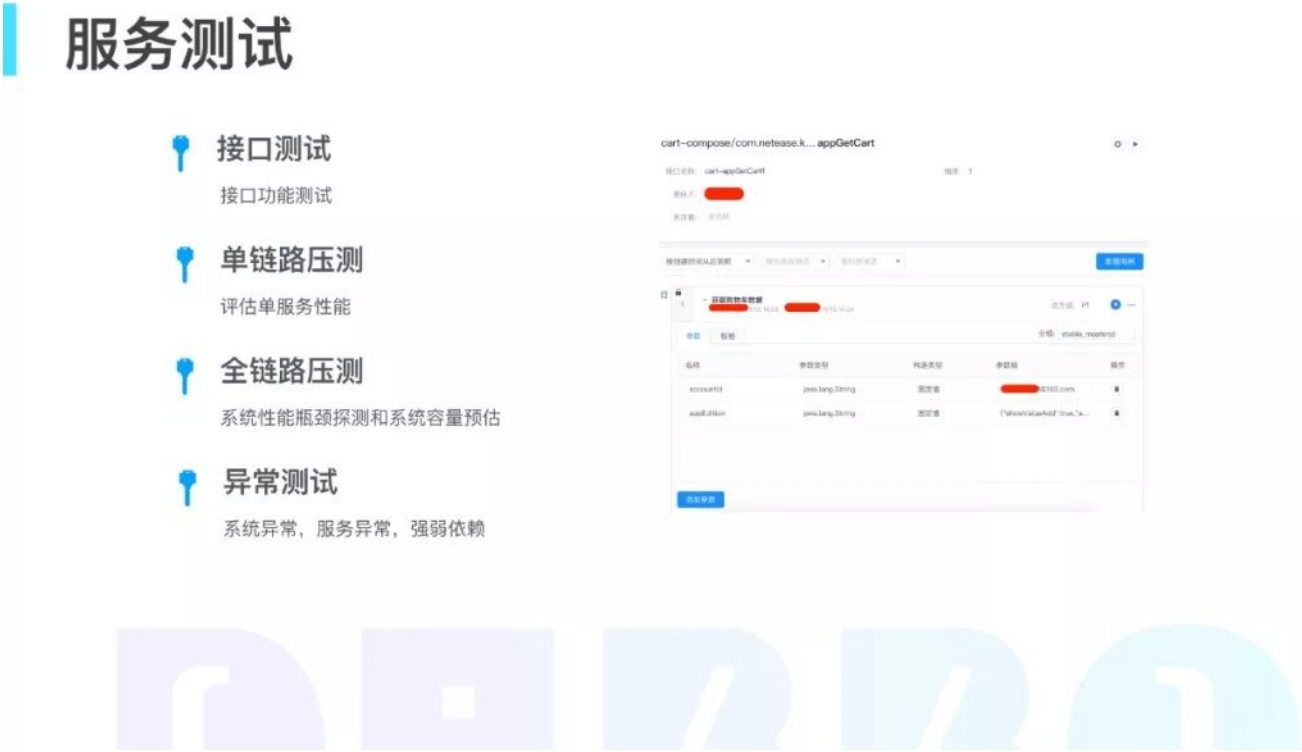
同时，也通过这些接口实现更优雅的停机，仅依赖shutdownhook，在某些情况下不一定靠谱，比如会有shutdownhook执行先后顺序的问题。应用发布的时候，首先调用offline接口，将注册服务全部从注册中心反注册，这时不再有新的流量进来，等到一段时间后，再执行停机发布操作，可以实现更加优雅的停机。

考拉在服务测试方面的改造

下面来介绍一下考拉在服务测试方面的实践。服务测试分为接口测试、单链路压测、全链路压测和异常测试四个维度。

接口测试

通过接口测试，可以来验证对外提供的Dubbo服务是否正确，因此我们也有接口测试平台，帮助QA更好的进行接口测试，包括对接口的编辑（入参、出参），用例的编辑和测试场景的执行等，



单链路压测

单链路的压测，主要面对单个功能的压测，比如要上线一个重要功能或者比较重要的接口之前，必须通过性能测试的指标才可以上线。

全链路压测

考拉作为电商平台，在大促前都会做全链路压测，用以探测系统的性能瓶颈，和对系统容量的预估。例如，探测系统的各类服务的容量是否够，需要扩容多少，以及限流的阈值要定多少合适，都可以通过全链路压测来给出一些合理的值。

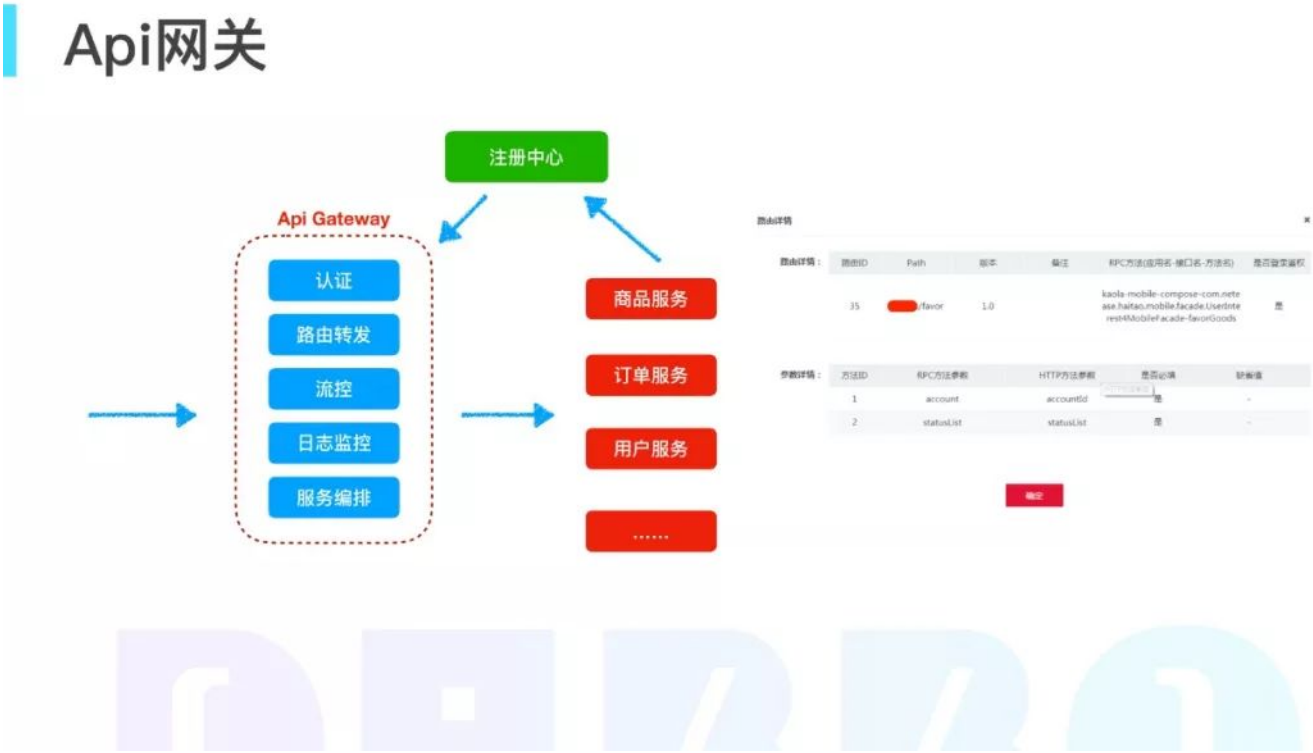
异常测试

对服务调用链路中的一些节点进行系统异常和服务异常的注入，也可以获取他们的强度依赖关系。比如一个非常重要的接口，可以

从Trace获取的调用链路，然后对调用链的依赖的各个服务节点进行异常注入。通过接口的表现，系统就会判断这个接口的强度依赖关系，以改善这些不合理的强依赖关系。

考拉在API网关方面的改造

随着考拉服务化的发展，我们自研了API网关，API网关可以作为外部流量的统一接口，提供了包括路由转发、流控和日志监控等一些公共的功能。

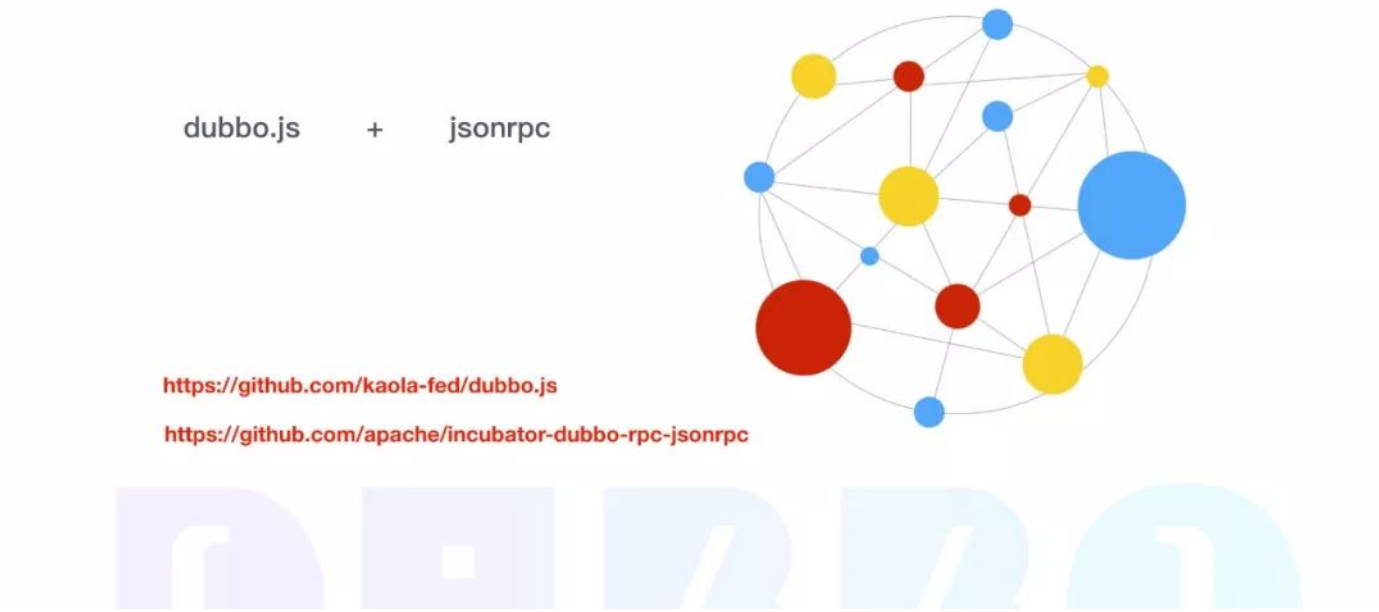


考拉的API网关是通过泛化调用的方式来调用后台Dubbo的服务的。Dubbo原生的泛化调用的性能比普通Api调用要差一些，所以我们也对泛化调用性能做了一些优化，也就是去掉了泛化调用在返回结果时的一次对象转换。最终压测的结果泛化的性能甚至比正常的调用性能还要好些。

考拉在多语言方面的改造

考拉在业务发展的过程中产生了不少多语言的需求，例如，我们的前端团队希望可以用Node应用调用Dubbo服务。对比了易用性，选用了开源的jsonrpc 方案，然后在后端的Dubbo服务上暴露了双协议，包括Dubbo协议和json rpc协议。

多语言



但在实施的过程中，也遇到了一些小问题，比如说，对于Dubbo消费者来说，不管是什么样的协议提供者，都是invoker。通过一个负载均衡策略，选取一个invoker进行调用，这个时候就会导致原来的Java客户端选用一个jsonrpc协议的提供者。这样如果他们的API版本不一致，就有可能导致序列化异常，出现调用失败的情况。所以，我们对Dubbo的一些调用逻辑做了改造，例如在Java客户端的消费者进行调用的时候，除非显示的配置，否则默认只用Dubbo协议去调用。另外，考拉也为社区的jsonrpc扩展了隐式传参的功能，因为可以用Dubbo隐式传参的功能来传递一些全链路参数。

考拉在解决注册中心性能瓶颈方面的实践

注册中心瓶颈可能是大部分电商企业都会遇到的问题，考拉也不例外。我们现在线上的Dubbo服务实例大概有4000多个，但是在ZooKeeper中注册的节点有一百多万个，包括服务注册的URL和消费者订阅的URL。

注册瓶颈

Dubbo应用发布时的惊群效应、重复通知和消费者拉取带来的瞬时流量一下就把ZooKeeper集群的网卡打满，ZooKeeper还有另外一个问题，他的强一致性模型导致CPU的利用率不高。

就算扩容，也解决不了ZooKeeper写性能的问题，ZooKeeper写是不可扩展的，并且应用发布时有大量的请求排队，从而使得接口性能急剧下降，表现出来的现象就是应用启动十分缓慢。

因此，在今年年初的时候就我们决定把ZooKeeper注册中心给替换掉，对比了现有的一些开源的注册中心，包括Consul、Eruka、etcd等，觉得他们并不适合Dubbo这种单进程多服务的注册模型，同时容量能否应对未来考拉的发展，也是一个问号。于是，我们决定自研注册中心，目前正在注册中心的迁移过程当中，采用的是双注册中心的迁移方案，即服务会同时注册ZooKeeper注册中心，还有新的注册中心，这样对原有的架构不会产生太大的影响。

考拉新的注册中心改造方案和现在社区的差不多，比如说也做了一个注册数据的拆分，往注册中心注册的数据只包含IP，Port 等关键数据，其它的数据都写到了Redis里面，注册中心实现使用了去中心化的一个架构，包括使用最终一致性来换取我们接口性能的一个提升。后面如果接入Dubbo，会考虑使用Nacos而不是ZooKeeper作为注册中心。

未来规划

考拉最近也在进行第二机房的建设，通过两个机房独立部署相同的一套系统，以实现同城双活。针对双机房的场景，Dubbo会做一定的改造，例如同机房优先调用，类似于即将发布的Dubbo2.7.0中的路由特性。在Dubbo在服务注册的时候，读取系统环境变量的环境标或者机房标，再将这些机房标注册到注册中心，然后消费端会做一个优先级路由，优先进行同机房的服务调用。

未来展望

容器化也是我们在规划的一个方向。随着服务化进程的演进，服务数也变得越来越多，通过容器化、DevOps可以提升测试、部署和运维效率。

Service Mesh在今年非常火，通过Service Mesh将服务框架的的能力比如注册发，路由和负载均衡，服务治理等下沉到Sidecar，

使用独立进程的方式来运行。对于业务工程的一个解耦，帮助我们实现一个异构系统，对多语言支持，也可以解决中间件升级推动困难以及各种依赖的冲突，业务方也可以更好的关注于业务开发，这也会是未来探索的一个方向。

以上就是我们团队在服务化进程中的一些实践和思考，谢谢大家。

- END -

推荐阅读

1. 全双工通信的 WebSocket
2. 还没抢到票吗？
3. 为什么年终奖是一个彻头彻尾的职场圈套？
4. 什么是一致性 Hash 算法？
5. 团队开发中 Git 最佳实践

喜欢文章，点个在看

大白话带你梳理一下 Dubbo 的那些事儿

Java后端 2月14日

以下文章来源于Java知音 ，作者idea

首先声明，本文并不是什么代码实战类型的文章，适合于想对dubbo有更加全面认识的读者阅读，文章不会过于深奥，只是将一系列的知识点串通起来，帮助读者温故而知新。

RPC服务的介绍

相信有过一些分布式开发经历的读者都有用过一些RPC框架，通过框架包装好之后提供的API接口调用远程服务，体验感觉起来就和调用本地服务一样轻松。**这么方便好用的技术框架，在实际的开发过程中是如何包装的呢？**

很早的时候，国外的工程师设计了一种能够通过A计算机调用B计算机上边应用程序的技术，这种技术不需要开发人员对于网络通讯了解过多，并且调用其他机器上边程序的时候和调用本地的程序一样方便好用。

A机器发起请求去调用B机器程序的时候会被挂起，B机器接收到A机器发起的请求参数之后会做一定的参数转换，最后将对应的程序结果返回给A，这就是最原始的RPC服务调用了。

RPC调用的优势

简单

不需要开发者对于网络通信做过多的设置，例如我们在使用http协议进行远程接口调用的时候，总是会需要编写较多的http协议参数（header，context，Accept-Language,Accept-Encode等等），这些处理对于开发人员来说，实际上都并不是特别友好。但是RPC服务调用框架通常都将这类解析进行了对应的封装，大大降低了开发人员的使用难度。

高效

在网络传输方面，RPC更多是处于应用层和传输层之间。这里我们需要先理清楚一个问题，网络分层。RPC是处于会话层的部分，相比处于应用层的HTTP而言，**RPC要比Rest服务调用更加轻便。**

常见的远程调用技术

rmi

利用java.rmi包实现，**基于Java远程方法协议(Java Remote Method Protocol) 和java的原生序列化**。

Hessian

是一个轻量级的remoting onhttp工具，使用简单的方法提供了RMI的功能。基于HTTP协议，采用二进制编解码。

protobuf-rpc-pro

是一个Java类库，提供了基于 Google 的 Protocol Buffers 协议的远程方法调用的框架。基于 Netty 底层的 NIO 技术。支持 TCP 重用/ keep-alive、SSL加密、RPC 调用取消操作、嵌入式日志等功能。

Thrift

是一种可伸缩的跨语言服务的软件框架。它拥有功能强大的代码生成引擎，无缝地支持C + +，C#，Java，Python和PHP和Ruby。thrift允许你定义一个描述文件，描述数据类型和服务接口。依据该文件，编译器方便地生成RPC客户端和服务器通信代码。

最初由facebook开发用做系统内部语言之间的RPC通信，2007年由facebook贡献到apache基金 ，现在是apache下的opensource之一 。支持多种语言之间的RPC方式的通信：php语言client可以构造一个对象，调用相应的服务方法来调用java语言的服务，跨越语言的C/S RPC调用。底层通讯基于SOCKET。

Avro

出自Hadoop之父Doug Cutting, 在Thrift已经相当流行的情况下推出Avro的目标不仅是提供一套类似Thrift的通讯中间件,更是要建立一个新的，标准性的云计算的数据交换和存储的Protocol。支持HTTP，TCP两种协议。

Dubbo

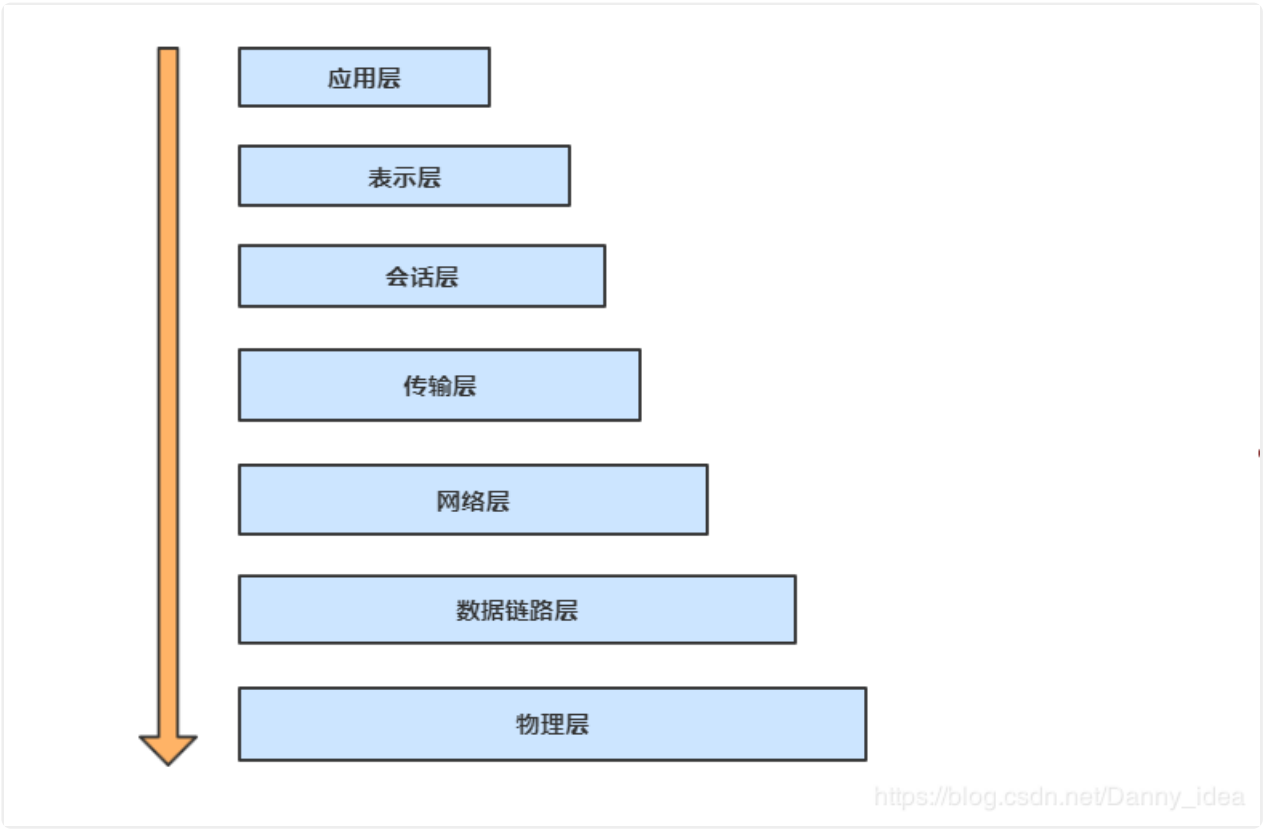
Dubbo是 阿里巴巴公司开源的一个高性能优秀的服务框架，使得应用可通过高性能的 RPC 实现服务的输出和输入功能，可以和Spring框架无缝集成。

上边我们说到了RPC的远程调用发展历史，那么下边我们一起来深入探讨一下RPC的服务。

首先我们来看看OSI的网络协议内容。

OSI的七层网络模型

对于OSI的七层网络模型我绘制了下边的这么一张图：



下边是我个人对于这七层协议的理解：

- **应用层** 主要是对于服务接口的格式多定义，例如提供一定的终端接口暴露给外部应用调用。
- **表示层** 处理一些数据传输的格式转换，例如说编码的统一，加密和解密处理。
- **会话层** 管理用户的会话和对话，建立不同机器之间的会话连接。
- **传输层** 向网络层提供可靠有序的数据包信息。
- **网络层** 真正发送数据包信息的层面，提供流和拥塞控制，从而降低网络的资源损耗。
- **数据链路层** 封装对应的数据包，检测和纠正数据包传输信息。
- **物理层** 通过网络通讯设备发送数据

HTTP & RPC

HTTP主要是位于TCP/IP协议栈的应用层部分，首先需要构建三次握手的链接，接着才能进行数据信息的请求发送，最后进行四次挥手断开链接。

RPC在请求的过程中跨越了传输层和应用层，这是因为它本身是依赖于Socket的原因。（再深入的原因我也不知道）。减少了上边几层的封装，RPC的请求效率自然是要比HTTP高效很多。

那么一个完整的RPC调用应该包含哪些部分呢?

通常我们将一个完整的RPC架构分为了以下几个核心组件：

- Server
- Client
- Server Stub
- Client Stub

这四个模块中我稍微说下stub吧。这个单词翻译过来称之为存根。

Client Stub 就是将客户端请求的参数，服务名称，服务地址进行打包，统一发送给server方。

Server Stub 我用通俗易懂的语言来解释就是服务端接收到Client发送的数据之后进行消息解包，调用本地方法。（看过netty拆包机制应该会对这块比较了解）。

Dubbo的核心属性

其实Dubbo配置里面的核心内容就是 ***服务暴露，服务发现，服务治理***。

什么是服务暴露，服务发现，服务治理?

下边我们用一段xml的配置来进行讲解：

```xml
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:dubbo="http://code.alibabatech.com/schema/dubbo"
    xmlns="http://www.springframework.org/schema/beans"
    xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spr
  http://code.alibabatech.com/schema/dubbo http://code.alibabatech.com/schema/dubbo/dubbo.xsd">
  <dubbo:application name="dubbo-invoker-provider">
    <dubbo:parameter key="qos.port" value="22222"/>
  </dubbo:application>
  <dubbo:registry address="zookeeper://127.0.0.1:2181"/>
  <dubbo:protocol name="dubbo" port="20880"/>
  <bean id="userService" class="com.sise.user.service.UserServiceImpl" />
  <dubbo:service interface="com.sise.user.service.UserService" ref="userService" />
</beans>
```

在dubbo的配置文件里面，通常我们所说的**dubbo:service** 可以理解为服务暴露，**dubbo:refernce** 为服务发现，**mock**是服务治理，**timeout**属于服务治理的一种（性能调优）.

假设dubbo里面希望将一些公共的配置抽取出来，我们可以通过properties文件进行配置，dubbo在加载配置文件的优先顺序如下：

1. 优先会读取JVM -D启动参数后边的内容
2. 读取xml配置文件
3. 读取properties配置文件内容

dubbo默认会读取dubbo.properties配置文件的信息，例如下边这种配置：

```
dubbo.application.name=dubbo-user-service
dubbo.registry.address=zookeeper://127.0.0.1:2181
```

假设我们的dubbo配置文件不命名为dubbo.properties（假设命名为了my-dubbo.properties）的时候，可以在启动参数的后边加上这么一段指令：

```
-Ddubbo.properties.file=my-dubbo.properties
```

那么在应用程序启动之后，对应的工程就会读取指定的配置文件，这样就可以将一些共用的dubbo配置给抽取了出来。

XML和配置类的映射

在工作中，我们通常都会通过配置xml的方式来设定一个服务端暴露的服务接口和消费端需要调用的服务信息，这些配置的xml实际上在dubbo的源码中都会被解析为对应的实体类对象。

例如说我们常用到的reference配置类，下边我贴出一段代码：

```java
package com.sise.user.config;
import com.sise.user.service.UserService;
import com.sise.user.service.UserServiceImpl;
import org.apache.dubbo.config.*;
import java.io.IOException;
import java.util.concurrent.CountDownLatch;
/**
 * dubbo里面的自定义配置类
 *
 * @author idea
 * @data 2019/12/29
 */
public class DubboSelfDefConfig {
    /**
     * dubbo的服务暴露
     */
    public void server() {
        ApplicationConfig applicationConfig = new ApplicationConfig();
        applicationConfig.setName("dubbo-server-config");
        RegistryConfig registryConfig = new RegistryConfig();
        registryConfig.setAddress("zookeeper://127.0.0.1:2181");
        ProtocolConfig protocolConfig = new ProtocolConfig();
        protocolConfig.setName("dubbo");
        protocolConfig.setPort(20880);
        protocolConfig.setThreads(200);
        UserService userService = new UserServiceImpl();
        ServiceConfig<UserService> serviceConfig = new ServiceConfig<>();
        serviceConfig.setApplication(applicationConfig);
        serviceConfig.setRegistry(registryConfig);
        serviceConfig.setProtocol(protocolConfig);
        serviceConfig.setInterface(UserService.class);
        serviceConfig.setRef(userService);
        serviceConfig.export();
    }

    public void consumer() {
        ApplicationConfig applicationConfig = new ApplicationConfig();
        applicationConfig.setName("dubbo-client-config");
        RegistryConfig registryConfig = new RegistryConfig();
        registryConfig.setAddress("zookeeper://127.0.0.1:2181");
        ReferenceConfig<UserService> referenceConfig = new ReferenceConfig<>();
        referenceConfig.setApplication(applicationConfig);
        referenceConfig.setRegistry(registryConfig);
        referenceConfig.setInterface(UserService.class);
        UserService localRef = referenceConfig.get();
        localRef.echo("idea");
    }
    public static void main(String[] args) throws InterruptedException, IOException {
        DubboSelfDefConfig d = new DubboSelfDefConfig();
        d.consumer();
        CountDownLatch countDownLatch = new CountDownLatch(1);
        countDownLatch.await();
    }
}
```

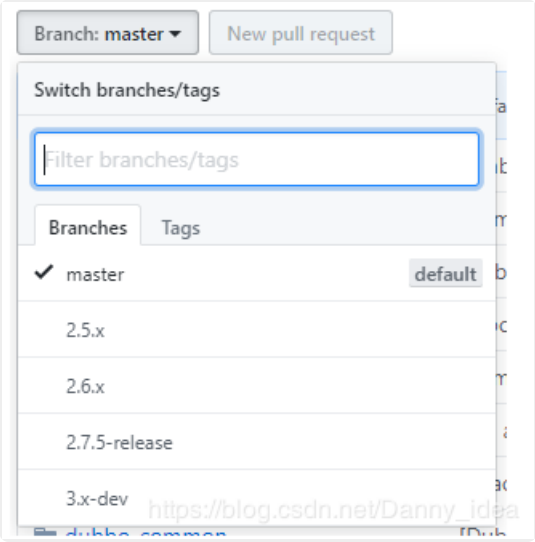
在这段代码里面，通过案例可以发现有这些信息内容：

```
UserService localRef = referenceConfig.get();
  localRef.echo("idea");
```

这两行语句是获取具体服务的核心之处，由于我在别处定义了一个叫做UserService 的公共服务接口，因此在服务引用的过程中可以进行转换。

Dubbo2.7的三大新特新

Dubbo的github官方地址为 https://github.com/apache/dubbo



Dubbo 目前有如图所示的 5 个分支，其中 2.7.1-release 只是一个临时分支，忽略不计，对其他 4 个分支而言，我归纳了一下，分别有如下信息：

- 2.5.x 近期已经通过投票，Dubbo 社区即将停止对其的维护。
- 2.6.x 为长期支持的版本，也是 Dubbo 贡献给 Apache 之前的版本，其包名前缀为：com.alibaba，JDK 版本对应 1.6。
- 3.x-dev 是前瞻性的版本，对 Dubbo 进行一些高级特性的补充，如支持 rx 特性。
- master 为长期支持的版本，版本号为 2.7.x，也是 Dubbo 贡献给 Apache 的开发版本，其包名前缀为：org.apache，JDK 版本对应 1.8。

Dubbo 2.7 新特性

Dubbo 2.7.x 作为 Apache 的孵化版本，除了代码优化之外，还新增了许多重磅的新特性，本文将会介绍其中最典型的2个新特性：

- **异步化改造**
- **三大中心改造**

异步化改造

1.异步化调用的方式，在Dubbo2.7版本里面提供了异步化调用的功能，相关案例代码如下所示：

```
@RestController
@RequestMapping(value = "/test")
public class TestController {
   @Reference(async = true)
   private UserService userService;

   @GetMapping("/testStr")
   public String testStr(String param){
      return userService.testEcho(param);
   }
}
```

但是通过这种异步发送的方式我们通常都是获取不到响应值的，所以这里的return为null。

如果在低于2.7版本的dubbo框架中希望获取到异步返回的响应值还是需要通过RPC上下文来提取信息。

代码案例如下所示：

```
@GetMapping("/futureGet")
   public String futureGet(String param) throws ExecutionException, InterruptedException {
      userService.testEcho(param);
      Future<String> future= RpcContext.getContext().getFuture();
      String result = future.get();
      System.out.println("this is :"+result);
      return result;
   }
```

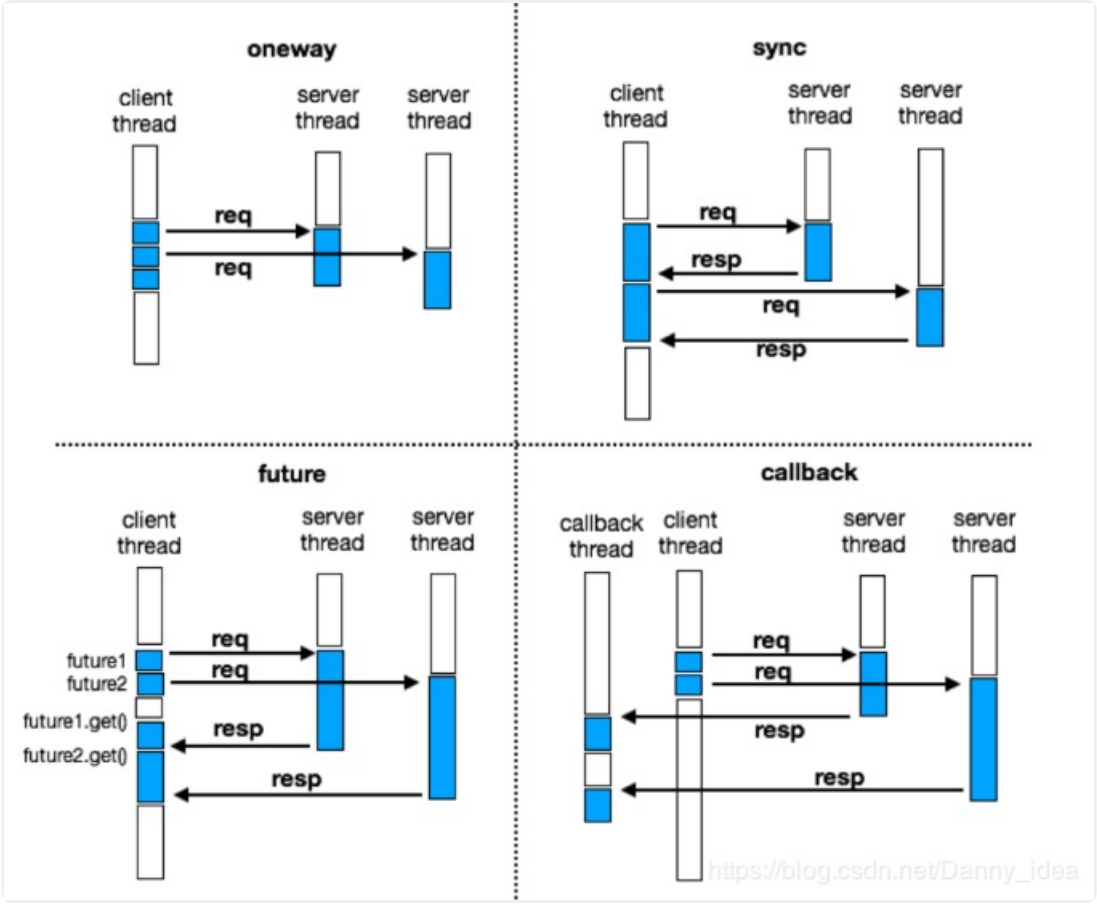
通过RPC上下文的方式可以取到对应的响应值,但是这种方式需要有所等待，因此此时的效率会有所降低。假设我们将dubbo的版本提升到了2.7.1之后，通过使用CompletableFuture来进行接口优化的话，这部分的代码实现就会有所变化：

```
/**
 * @author idea
 * @date 2019/12/31
 * @Version V1.0
 */
public interface DemoService {
    String sayHello(String name) ;
    default CompletableFuture<String> sayAsyncHello(String name){
        return CompletableFuture.completedFuture(sayHello(name));
    }
}
```

调用方代码：

```java
package com.sise.consumer.controller;

import com.sise.dubbo.service.DemoService;
import org.apache.dubbo.config.annotation.Reference;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.atomic.AtomicReference;
/**
 * @author idea
 * @date 2019/12/31
 * @Version V1.0
 */
@RestController
@RequestMapping(value = "/demo")
public class DemoController {

  @Reference
  private DemoService demoService;

  @RequestMapping(value = "/testDemo")
  public String testDemo(String name){
    System.out.println("【testDemo】 this is :"+name);
    return demoService.sayHello(name);
  }.

  @RequestMapping(value = "/testAsyncDemo")
  public String testAsyncDemo(String name){
    System.out.println("【testAsyncDemo】 this is :"+name);
    CompletableFuture<String> future = demoService.sayAsyncHello(name);
    AtomicReference<String> result = null;
    //通过一条callback线程来处理响应的数据信息
    future.whenComplete((retValue,exception)->{
      if(exception==null){
        System.out.println(retValue);
        result.set(retValue);
      } else {
        exception.printStackTrace();
      }
    });
    return "通过一条callback线程来处理响应的数据信息,所以这个时候获取不到信息响应";
  }
}
```

这样的调用是借助了callback线程来帮我们处理原先的数据内容，关于dubbo里面的异步化调用，我借用了官方的一张图来进行展示：



我们上边讲解的众多方法都只是针对于dubbo的客户端异步化，并没有讲解关于服务端的异步化处理，这是因为结合dubbo的业务线程池模型来思考，服务端的异步化处理比较鸡肋（因为dubbo内部服务端的线程池本身就是异步化调用的了）。

当然dubbo 2.6 里面对于接口异步化调用的配置到了2.7版本依旧有效。

三大中心的改造

注册中心

在dubbo2.7之前，dubbo主要还是由consumer，provider ，register组成，然而在2.7版本之后，dubbo的注册中心被拆解为了三个中心，分别是原先的**注册中心**和**元数据中心**以及**配置中心**。

元数据配置

在dubbo2.7版本中，将原先注册在zk上边的过多数据进行了注册拆分，这样能够保证减少对于zk端的压力。具体配置如下：

```
<dubbo:registry address=“zookeeper://127.0.0.1:2181” simplified="true"/>
```

简化了相应配置之后，dubbo也只会上传一些必要的服务治理数据了，简化版本的服务数据只剩下下边这些信息：

```
dubbo://30.5.120.185:20880/com.sise.TestService?
application=test-provider&
dubbo=2.0.2&
release=2.7.0&
timestamp=1554982201973
```

对于其他的元数据信息将会被存储到一些元数据中心里面，例如说redis，nacos，zk等

元数据配置改造主要解决的问题是：推送量大 -> 存储数据量大 -> 网络传输量大 -> 延迟严重

配置中心

dubbo2.7开始支持多种分布式配置中心的组件。例如说：zk，Spring Cloud Config, Apollo, Nacos，关于这部分的配置网上的资料也比较多，我就不在这里细说了。

推荐阅读

1. 如何轻松阅读 GitHub 上的项目源码？

2. IntelliJ IDEA新增禅模式和LightEdit模式

3. 安利一款 IDEA 中强大的代码生成利器

4. 如何获取靠谱的新型冠状病毒疫情

面试挂在 Dubbo RPC？我把常问面试题整理好了，来拿！

hu1991die Java后端 2019-09-26

作者 | hu1991die

链接 | www.jianshu.com/p/78f72ccf0377

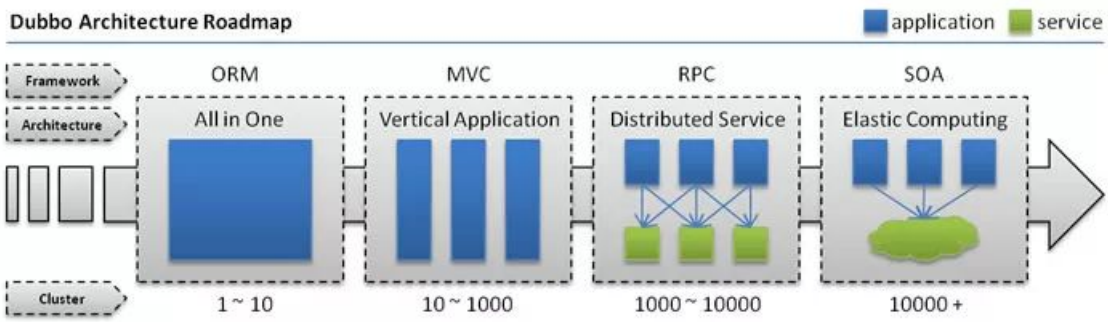
上一篇：计算机专业的学生也太太太太太惨了吧?

RPC非常重要，很多人面试的时候都挂在了这个地方！你要是还不懂RPC是什么？他的基本原理是什么？你一定要把下边的内容记起来！好好研究一下！特别是文中给出的一张关于RPC的基本流程图，重点中的重点，Dubbo RPC的基本执行流程就是他，RPC框架的基本原理也是他，别说我没告诉你！看了下边的内容你要掌握的内容如下，当然还有很多：

- RPC的由来，是怎样一步步演进出来的；
- RPC的基本架构是什么；
- RPC的基本实现原理，就是下边的这张图，重点中的重点；
- REST 和 SOAP、RPC 有何区别呢?
- 整个调用的过程经历了哪几步和Spring MVC的执行流程一样，相当重要；

一、为什么要有RPC

随着互联网的发展，网站应用的规模不断扩大，常规的垂直应用架构已无法应对，分布式服务架构以及流动计算架构势在必行，亟需一个治理系统确保架构有条不紊的演进。



1、单一应用架构

当网站流量很小时，只需一个应用，将所有功能都部署在一起，以减少部署节点和成本。此时，用于简化增删改查工作量的数据访问框架(ORM) 是关键。

2、垂直应用架构

当访问量逐渐增大，单一应用增加机器带来的加速度越来越小，将应用拆成互不相干的几个应用，以提升效率。此时，用于加速前端页面开发的 Web框架(MVC) 是关键。

3、分布式服务架构

当垂直应用越来越多，应用之间交互不可避免，将核心业务抽取出来，作为独立的服务，逐渐形成稳定的服务中心，使前端应用能更快速的响应多变的市场需求。

此时，用于提高业务复用及整合的分布式服务框架(RPC)，提供统一的服务是关键。

例如：各个团队的服务提供方就不要各自实现一套序列化、反序列化、网络框架、连接池、收发线程、超时处理、状态机等“业务之外”的重复技术劳动，造成整体的低效。

PS：其实上述三个原因也是为什么要有Dubbo的原因！不信你去Dubbo官网去看！

流动计算架构

PS：这个属于扩展内容，摘自Dubbo官网，属于架构演进的一个过程

当服务越来越多，容量的评估，小服务资源的浪费等问题逐渐显现，此时需增加一个调度中心基于访问压力实时管理集群容量，提高集群利用率。此时，用于提高机器利用率的资源调度和治理中心(SOA)是关键。

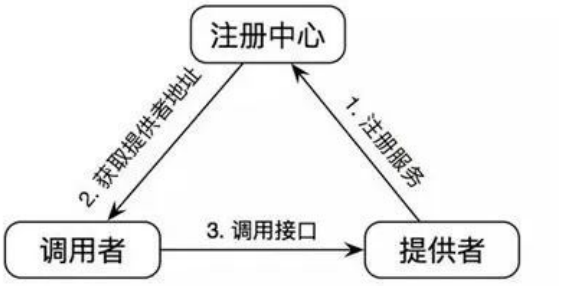
4、另外一个原因

就是因为在几个进程内（应用分布在不同的机器上），无法共用内存空间，或者在一台机器内通过本地调用无法完成相关的需求，比如不同的系统之间的通讯，甚至不同组织之间的通讯。此外由于机器的横向扩展，需要在多台机器组成的集群上部署应用等等。

所以，统一RPC框架来解决提供统一的服务。

二、什么是RPC

RPC（Remote Procedure Call Protocol）远程过程调用协议，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。简言之，RPC使得程序能够像访问本地系统资源一样，去访问远端系统资源。比较关键的一些方面包括：通讯协议、序列化、资源（接口）描述、服务框架、性能、语言支持等。



简单的说，RPC就是从一台机器(客户端)上通过参数传递的方式调用另一台机器(服务器)上的一个函数或方法(可以统称为服务)并得到返回的结果。

三、PRC架构组件

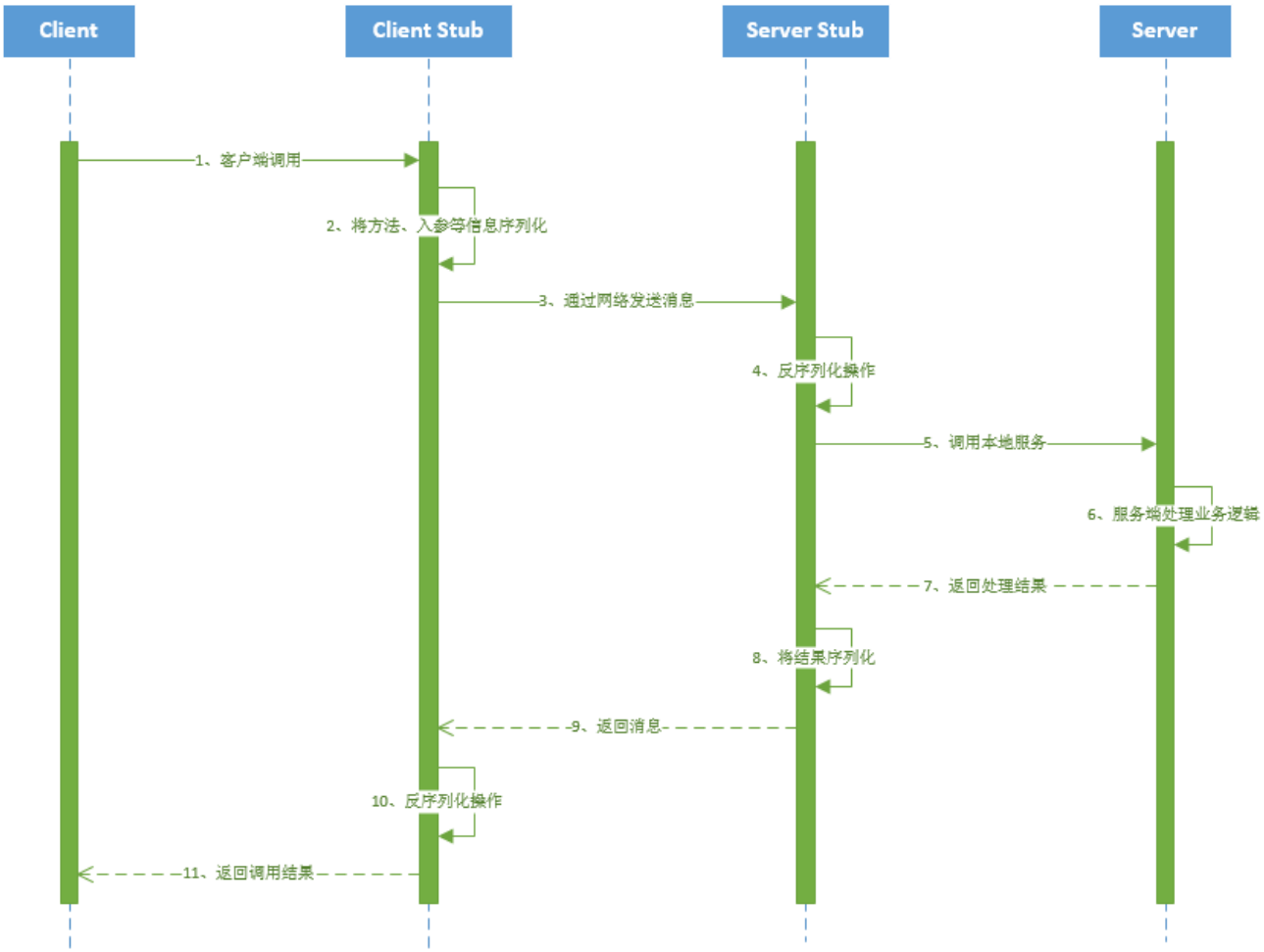
一个基本的RPC架构里面应该至少包含以下4个组件：

1、**客户端**（Client）:服务调用方（服务消费者）

2、**客户端存根**（Client Stub）:存放服务端地址信息，将客户端的请求参数数据信息打包成网络消息，再通过网络传输发送给服务端

3、**服务端存根**（Server Stub）:接收客户端发送过来的请求消息并进行解包，然后再调用本地服务进行处理

4、**服务端**（Server）:服务的真正提供者



具体调用过程：

- 服务消费者（client客户端）通过调用本地服务的方式调用需要消费的服务；
- 客户端存根（client stub）接收到调用请求后负责将方法、入参等信息序列化（组装）成能够进行网络传输的消息体；
- 客户端存根（client stub）找到远程的服务地址，并且将消息通过网络发送给服务端；
- 服务端存根（server stub）收到消息后进行解码（反序列化操作）；
- 服务端存根（server stub）根据解码结果调用本地的服务进行相关处理；
- 本地服务执行具体业务逻辑并将处理结果返回给服务端存根（server stub）；
- 服务端存根（server stub）将返回结果重新打包成消息（序列化）并通过网络发送至消费方；
- 客户端存根（client stub）接收到消息，并进行解码（反序列化）；
- 服务消费方得到最终结果；

而RPC框架的实现目标则是将上面的第2-10步完好地封装起来，也就是把调用、编码/解码的过程给封装起来，让用户感觉上像调用本地服务一样的调用远程服务。

四、RPC和SOA、SOAP、REST的区别

1、REST

可以看着是HTTP协议的一种直接应用，默认基于JSON作为传输格式,使用简单,学习成本低效率高,但是安全性较低。

2、SOAP

SOAP是一种数据交换协议规范,是一种轻量的、简单的、基于XML的协议的规范。而SOAP可以看着是一个重量级的协议，基于XML、SOAP在安全方面是通过使用XML-Security和XML-Signature两个规范组成了WS-Security来实现安全控制的,当前已经得到了各个厂商的支持。

它有什么优点？简单总结为：易用、灵活、跨语言、跨平台。

3、SOA

面向服务架构，它可以根据需求通过网络对松散耦合的粗粒度应用组件进行分布式部署、组合和使用。服务层是SOA的基础，可以直接被应用调用，从而有效控制系统中与软件代理交互的人为依赖性。

SOA是一种粗粒度、松耦合服务架构，服务之间通过简单、精确定义接口进行通讯，不涉及底层编程接口和通讯模型。SOA可以看作是B/S模型、XML（标准通用标记语言的子集）/Web Service技术之后的自然延伸。

4、REST 和 SOAP、RPC 有何区别呢?

没什么太大区别，他们的本质都是提供可支持分布式的基础服务，最大的区别在于他们各自的的特点所带来的不同应用场景 。

五、RPC框架需要解决的问题?

1、如何确定客户端和服务端之间的通信协议?

2、如何更高效地进行网络通信?

3、服务端提供的服务如何暴露给客户端?

4、客户端如何发现这些暴露的服务?

5、如何更高效地对请求对象和响应结果进行序列化和反序列化操作?

六、RPC的实现基础?

1、需要有非常高效的网络通信，比如一般选择Netty作为网络通信框架;

2、需要有比较高效的序列化框架，比如谷歌的Protobuf序列化框架;

3、可靠的寻址方式（主要是提供服务的发现），比如可以使用Zookeeper来注册服务等等;

4、如果是带会话（状态）的RPC调用，还需要有会话和状态保持的功能;

七、RPC使用了哪些关键技术?

1、动态代理

生成Client Stub（客户端存根）和Server Stub（服务端存根）的时候需要用到Java动态代理技术，可以使用JDK提供的原生的动态代理机制，也可以使用开源的：CGLib代理，Javassist字节码生成技术。

2、序列化和反序列化

在网络中，所有的数据都将会被转化为字节进行传送，所以为了能够使参数对象在网络中进行传输，需要对这些参数进行序列化和反序列化操作。

序列化：把对象转换为字节序列的过程称为对象的序列化，也就是编码的过程。

反序列化：把字节序列恢复为对象的过程称为对象的反序列化，也就是解码的过程。

目前比较高效的开源序列化框架：如Kryo、FastJson和Protobuf等。

3、NIO通信

出于并发性能的考虑，传统的阻塞式 IO 显然不太合适，因此我们需要异步的 IO，即 NIO。Java 提供了 NIO 的解决方案，Java 7 也提供了更优秀的 NIO.2 支持。可以选择Netty或者MINA来解决NIO数据传输的问题。

4、服务注册中心

可选：Redis、Zookeeper、Consul 、Etcd。一般使用ZooKeeper提供服务注册与发现功能，解决单点故障以及分布式部署的问题(注册中心)。

八、主流RPC框架有哪些

1、RMI

利用java.rmi包实现，基于Java远程方法协议(Java Remote Method Protocol) 和java的原生序列化。

2、Hessian

是一个轻量级的remoting onhttp工具，使用简单的方法提供了RMI的功能。基于HTTP协议，采用二进制编解码。

3、protobuf-rpc-pro

是一个Java类库，提供了基于 Google 的 Protocol Buffers 协议的远程方法调用的框架。基于 Netty 底层的 NIO 技术。支持 TCP 重用/keep-alive、SSL加密、RPC 调用取消操作、嵌入式日志等功能。

4、Thrift

是一种可伸缩的跨语言服务的软件框架。它拥有功能强大的代码生成引擎，无缝地支持C ++ +，C#，Java，Python和PHP和Ruby。thrift允许你定义一个描述文件，描述数据类型和服务接口。依据该文件，编译器方便地生成RPC客户端和服务器通信代码。

最初由facebook开发用做系统内个语言之间的RPC通信，2007年由facebook贡献到apache基金 ，现在是apache下的opensource之一 。支持多种语言之间的RPC方式的通信：php语言client可以构造一个对象，调用相应的服务方法来调用java语言的服务，跨越语言的C/S RPC调用。底层通讯基于SOCKET。

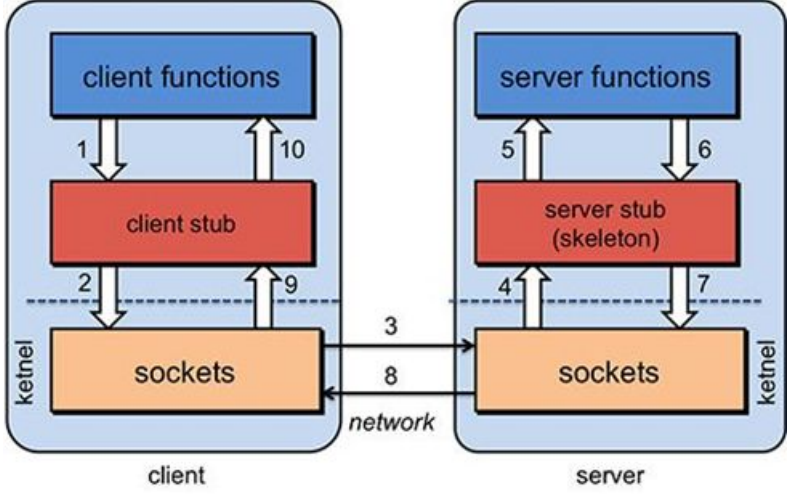
5、Avro

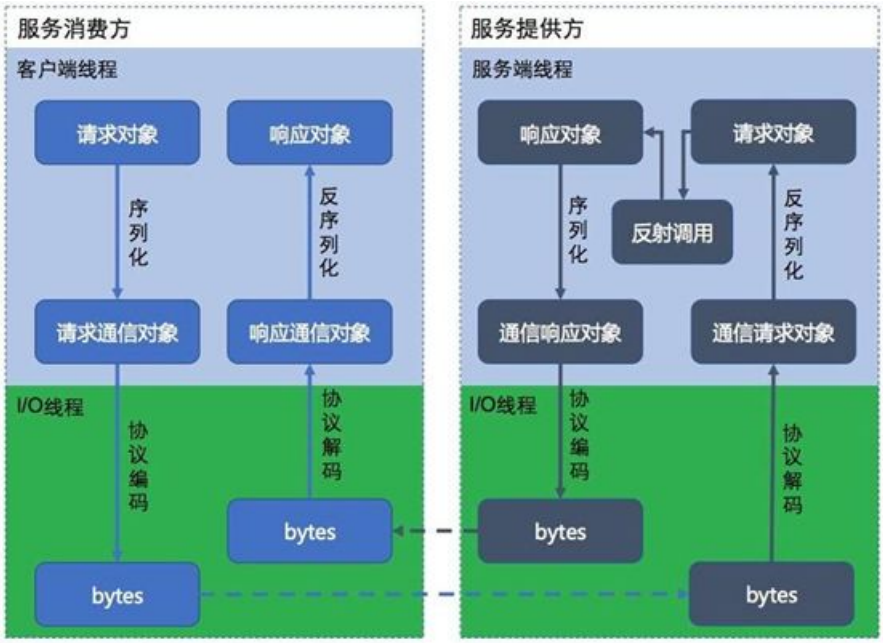
出自Hadoop之父Doug Cutting，在Thrift已经相当流行的情况下推出Avro的目标不仅是提供一套类似Thrift的通讯中间件，更是要建立一个新的，标准性的云计算的数据交换和存储的Protocol。支持HTTP，TCP两种协议。

6、Dubbo

Dubbo是 阿里巴巴公司开源的一个高性能优秀的服务框架，使得应用可通过高性能的 RPC 实现服务的输出和输入功能，可以和 Spring框架无缝集成。

九、RPC的实现原理架构图





PS：这张图非常重点，是PRC的基本原理，请大家一定记住！

也就是说两台服务器A，B，一个应用部署在A服务器上，想要调用B服务器上应用提供的函数/方法，由于不在一个内存空间，不能直接调用，需要通过网络来表达调用的语义和传达调用的数据。

比如说，A服务器想调用B服务器上的一个方法：

User getUserByName(String userName)

1、建立通信

首先要解决通讯的问题：即A机器想要调用B机器，首先得建立起通信连接。

主要是通过在客户端和服务器之间建立TCP连接，远程过程调用的所有交换的数据都在这个连接里传输。连接可以是按需连接，调用结束后就断掉，也可以是长连接，多个远程过程调用共享同一个连接。

通常这个连接可以是按需连接（需要调用的时候就先建立连接，调用结束后就立马断掉），也可以是长连接（客户端和服务器建立起连接之后保持长期持有，不管此时有无数据包的发送，可以配合心跳检测机制定期检测建立的连接是否存活有效），多个远程过程调用共享同一个连接。

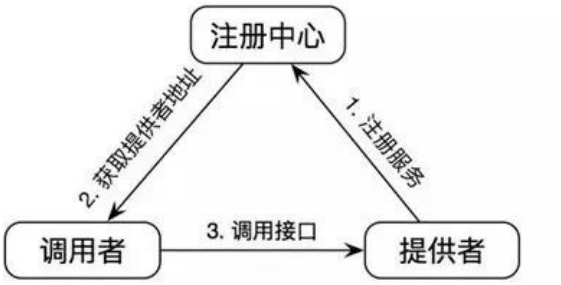
2、服务寻址

要解决寻址的问题，也就是说，A服务器上的应用怎么告诉底层的RPC框架，如何连接到B服务器（如主机或IP地址）以及特定的端口，方法

的名称名称是什么。

通常情况下我们需要提供B机器（主机名或IP地址）以及特定的端口，然后指定调用的方法或者函数的名称以及入参出参等信息，这样才能完成服务的一个调用。

可靠的寻址方式（主要是提供服务的发现）是RPC的实现基石，比如可以采用Redis或者Zookeeper来注册服务等等。



2.1、从服务提供者的角度看：

- 当服务提供者启动的时候，需要将自己提供的服务注册到指定的注册中心，以便服务消费者能够通过服务注册中心进行查找；
- 当服务提供者由于各种原因致使提供的服务停止时，需要向注册中心注销停止的服务；
- 服务的提供者需要定期向服务注册中心发送心跳检测，服务注册中心如果一段时间未收到来自服务提供者的心跳后，认为该服务提供者已经停止服务，则将该服务从注册中心上去掉

2.2、从调用者的角度看：

- 服务的调用者启动的时候根据自己订阅的服务向服务注册中心查找服务提供者的地址等信息；
- 当服务调用者消费的服务上线或者下线的时候，注册中心会告知该服务的调用者；
- 服务调用者下线的时候，则取消订阅。

3、网络传输

3.1、序列化

当A机器上的应用发起一个RPC调用时，调用方法和其入参等信息需要通过底层的网络协议如TCP传输到B机器，由于网络协议是基于二进制的，所有我们传输的参数数据都需要先进行序列化（Serialize）或者编组（marshal）成二进制的形式才能在网络中进行传输。然后通过寻址操作和网络传输将序列化或者编组之后的二进制数据发送给B机器。

3.2、反序列化

当B机器接收到A机器的应用发来的请求之后，又需要对接收到的参数等信息进行反序列化操作（序列化的逆操作），即将二进制信息恢复为内存中的表达方式，然后再找到对应的方法（寻址的一部分）进行本地调用（一般是通过生成代理Proxy去调用，
通常会有JDK动态代理、CGLIB动态代理、Javassist生成字节码技术等），之后得到调用的返回值。

4、服务调用

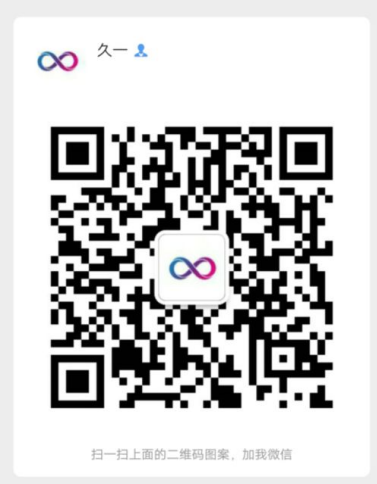
B机器进行本地调用（通过代理Proxy和反射调用）之后得到了返回值，此时还需要再把返回值发送回A机器，同样也需要经过序列化操作，然后再经过网络传输将二进制数据发送回A机器，而当A机器接收到这些返回值之后，则再次进行反序列化操作，恢复为内存中的表达方式，最后再交给A机器上的应用进行相关处理，一般是业务逻辑处理操作。

通常，经过以上四个步骤之后，一次完整的RPC调用算是完成了，另外可能因为网络抖动等原因需要重试等。

-END-

广告交流群。

↓扫描二维码进群↓



推荐阅读

1. Java后端优质文章整理
2. 计算机专业的学生也太太太太太惨了吧?
3. 面试官:说一说 Spring Boot 自动配置原理
4. 在浏览器输入 URL 回车之后发生了什么?
5. 接私活必备的 10 个开源项目

Java后端

长按识别二维码，关注我的公众号

喜欢文章，点个在看