# 1什么是RPC

## 1.1概念

RPC（Remote Procedure Call）是一种远程调用协议，简单地说就是能使应用像调用本地方法一样的调用远程的过程或服务，可以应用在分布式服务、分布式计算、远程服务调用等许多场景。

这样空洞的讲概念大家可能会难以理解，下面换一种比较通俗的方式来讲讲。

早期单机时代，一台电脑上运行多个进程，大家各干各的，老死不相往来。假如A进程需要一个画图的功能，B进程也需要一个画图的功能，程序员就必须为两个进程都写一个画图的功能。这不是整人么？于是就出现了IPC（Inter-process communication，单机中运行的进程之间的相互通信）。OK，现在A既然有了画图的功能，B就调用A进程上的画图功能好了，程序员终于可以偷下懒了。

到了网络时代，大家的电脑都连起来了。以前程序只能调用自己电脑上的进程，能不能调用其他机器上的进程呢？于是就程序员就把IPC扩展到网络上，这就是RPC（远程过程调用）了。现在不仅单机上的进程可以相互通信，多机器中的进程也可以相互通信了。

要知道实现RPC很麻烦呀，什么多线程、什么Socket、什么I/O，都是让咱们普通程序员很头疼的事情。于是就有牛人开发出RPC框架（比如，CORBA、RMI、Web Services、RESTful Web Services等等）。

OK，现在可以定义RPC框架的概念了。简单点讲，RPC框架就是可以让程序员来调用远程进程上的代码一套工具。有了RPC框架，咱程序员就轻松很多了，终于可以逃离多线程、Socket、I/O的苦海了。

## 1.2 RPC 与其它远程调用方式比较

RPC 与 HTTP、RMI、Web Service 都能完成远程调用，但是实现方式和侧重点各有不同。

### **HTTP**

HTTP（HyperText Transfer Protocol）是应用层通信协议，使用标准语义访问指定资源（图片、接口等），网络中的中转服务器能识别协议内容。HTTP 协议是一种资源访问协议，通过 HTTP 协议可以完成远程请求并返回请求结果。

HTTP 的**优点是简单、易用、可理解性强且语言无关**，在远程服务调用中包括微博有着广泛应用。HTTP 的缺点是协议头较重，一般请求到具体服务器的链路较长，可能会有 DNS 解析、Nginx 代理等。

RPC 是一种协议规范，可以把 HTTP 看作是一种 RPC 的实现，也可以把 HTTP 作为 RPC 的传输协议来应用。RPC 服务的自动化程度比较高，能够实现强大的服务治理功能，和语言结合更友好，性能也十分优秀。与 HTTP 相比，RPC 的缺点就是相对复杂，学习成本稍高。

### RMI

RMI（Remote Method Invocation）是指 Java 语言中的远程方法调用，RMI 中的每个方法都具有方法签名，RMI 客户端和服务器端通过方法签名进行远程方法调用。RMI 只能在 Java 语言中使用，**可以把 RMI 看作面向对象的 Java RPC**。

### Web Service

Web Service 是一种基于 Web 进行服务发布、查询、调用的架构方式，重点在于服务的管理与使用。Web Service 一般通过 WSDL 描述服务，使用 SOAP通过 HTTP 调用服务。

RPC 是一种远程访问协议，而 Web Service 是一种体系结构，Web Service 也可以通过 RPC 来进行服务调用，因此 Web Service 更适合同一个 RPC 框架进行比较。当 RPC 框架提供了服务的发现与管理，并使用 HTTP 作为传输协议时，其实就是 Web Service。

相对 Web Service，RPC 框架可以对服务进行更细粒度的治理，包括流量控制、SLA 管理等，在微服务化、分布式计算方面有更大的优势。

## **RPC 框架介绍**

RPC 协议只规定了 Client 与 Server 之间的点对点调用流程，包括 stub、通信协议、RPC 消息解析等部分，在实际应用中，还需要考虑服务的高可用、负载均衡等问题，所以这里的 RPC 框架指的是能够完成 RPC 调用的解决方案，除了点对点的 RPC 协议的具体实现外，还可以包括服务的发现与注销、提供服务的多台 Server 的负载均衡、服务的高可用等更多的功能。**目前的 RPC 框架大致有两种不同的侧重方向，一种偏重于服务治理，另一种偏重于跨语言调用**。

### **服务治理型 RPC 框架**

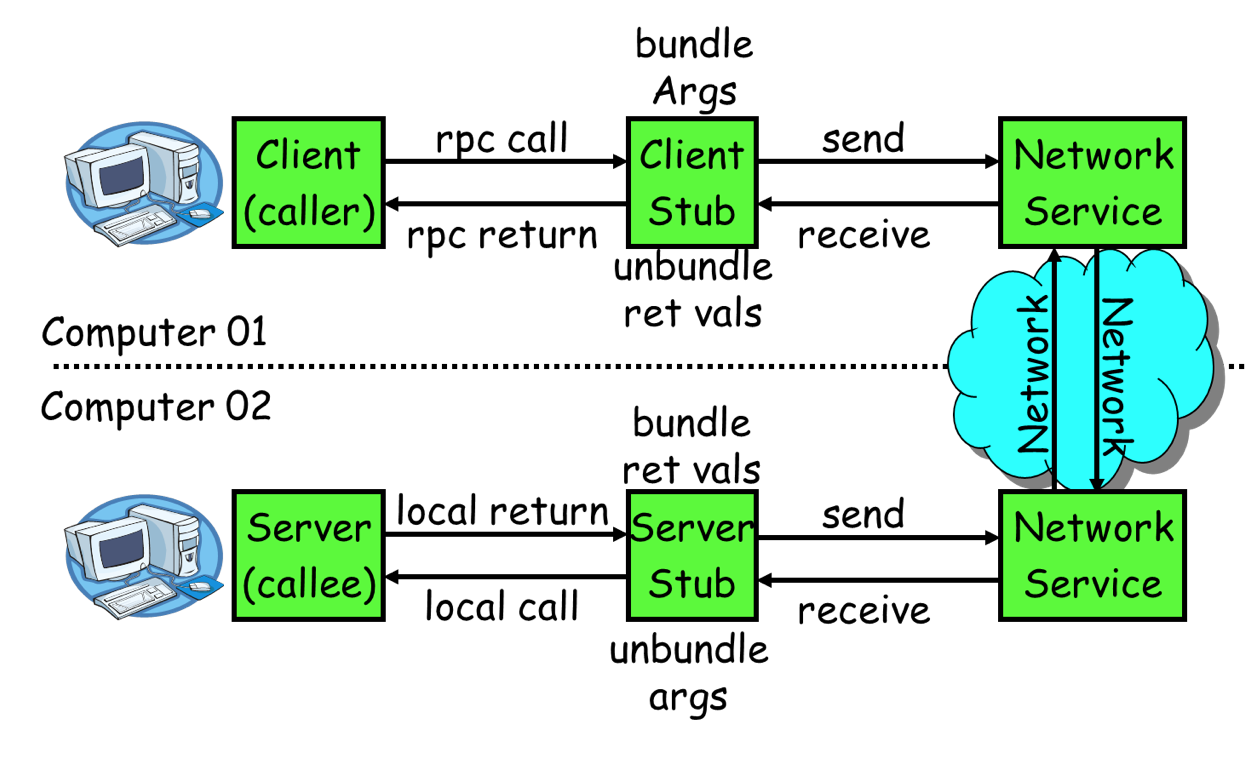
服务治理型的 RPC 框架有 Dubbo、DubboX 等，Dubbo 是阿里开源的分布式服务框架，能够实现高性能 RPC 调用，并且提供了丰富的管理功能，是十分优秀的 RPC 框架。DubboX 是基于 Dubbo 框架开发的 RPC 框架，支持 REST 风格远程调用，并增加了一些新的 feature。

这类的 RPC 框架的特点是功能丰富，提供高性能的远程调用以及服务发现及治理功能，**适用于大型服务的微服务化拆分以及管理**，对于特定语言（Java）的项目可以十分友好的透明化接入。但缺点是语言耦合度较高，跨语言支持难度较大。

### **跨语言调用型**

跨语言调用型的 RPC 框架有 **Thrift**、gRPC、Hessian、Hprose 等，这一类的 RPC 框架重点关注于服务的跨语言调用，能够支持大部分的语言进行语言无关的调用，非常适合于为不同语言提供通用远程服务的场景。**但这类框架没有服务发现相关机制**，实际使用时一般需要代理层进行请求转发和负载均衡策略控制。

# 如何调用他人的远程服务？



* 1）服务消费方（client）调用以本地调用方式调用服务；
* 2）client stub接收到调用后负责将方法、参数等组装成能够进行网络传输的消息体；
* 3）client stub找到服务地址，并将消息发送到服务端；
* 4）server stub收到消息后进行解码；
* 5）server stub根据解码结果调用本地的服务；
* 6）本地服务执行并将结果返回给server stub；
* 7）server stub将返回结果打包成消息并发送至消费方；
* 8）client stub接收到消息，并进行解码；
* 9）服务消费方得到最终结果。

RPC的目标就是要2~8这些步骤都封装起来，让用户对这些细节透明。

## 2.1怎么做到透明化远程服务调用？

怎么封装通信细节才能让用户像以本地调用方式调用远程服务呢？对java来说就是使用代理！java代理有两种方式：1） jdk 动态代理；2）字节码生成。尽管字节码生成方式实现的代理更为强大和高效，但代码不易维护，大部分公司实现RPC框架时还是选择动态代理方式。

下面简单介绍下动态代理怎么实现我们的需求。我们需要实现RPCProxyClient代理类，代理类的invoke方法中封装了与远端服务通信的细节，消费方首先从RPCProxyClient获得服务提供方的接口，当执行helloWorldService.sayHello(“test”)方法时就会调用invoke方法。

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | public class RPCProxyClient implements java.lang.reflect.InvocationHandler{      private Object obj;      public RPCProxyClient(Object obj){          this.obj=obj;      }      /\*\*       \* 得到被代理对象;       \*/      public static Object getProxy(Object obj){          return java.lang.reflect.Proxy.newProxyInstance(obj.getClass()  .getClassLoader(),                  obj.getClass().getInterfaces(), new RPCProxyClient(obj));      }        /\*\*       \* 调用此方法执行       \*/      public Object invoke(Object proxy, Method method, Object[] args)              throws Throwable {          //结果参数;          Object result = new Object();          // ...执行通信相关逻辑          // ...          return result;      }  } |

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class Test {      public static void main(String[] args) {          HelloWorldService helloWorldService = (HelloWorldService)RPCProxyClient  .getProxy(HelloWorldService.class);          helloWorldService.sayHello("test");      }  } |

## 2.2  怎么对消息进行编码和解码？

### 2.2.1 确定消息数据结构

通信的第一步就是要确定客户端和服务端相互通信的消息结构。客户端的请求消息结构一般需要包括以下内容：

1）接口名称

在我们的例子里接口名是“HelloWorldService”，如果不传，服务端就不知道调用哪个接口了；

2）方法名

一个接口内可能有很多方法，如果不传方法名服务端也就不知道调用哪个方法；

3）参数类型&参数值

参数类型有很多，比如有bool、int、long、double、string、map、list，甚至如struct（class）；

以及相应的参数值；

4）超时时间

5）requestID，标识唯一请求id，在下面一节会详细描述requestID的用处。

同理服务端返回的消息结构一般包括以下内容。

1）返回值

2）状态code

3）requestID

### 2.2.2 序列化

一旦确定了消息的数据结构后，下一步就是要考虑序列化与反序列化了。

什么是序列化？序列化就是将数据结构或对象转换成二进制串的过程，也就是编码的过程。

什么是反序列化？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。

为什么需要序列化？转换为二进制串后才好进行网络传输嘛！为什么需要反序列化？将二进制转换为对象才好进行后续处理！

现如今序列化的方案越来越多，每种序列化方案都有优点和缺点，它们在设计之初有自己独特的应用场景，那到底选择哪种呢？从RPC的角度上看，主要看三点：1）通用性，比如是否能支持Map等复杂的数据结构；2）性能，包括时间复杂度和空间复杂度，由于RPC框架将会被公司几乎所有服务使用，如果序列化上能节约一点时间，对整个公司的收益都将非常可观，同理如果序列化上能节约一点内存，网络带宽也能省下不少；3）可扩展性，对互联网公司而言，业务变化快，如果序列化协议具有良好的可扩展性，支持自动增加新的业务字段，删除老的字段，而不影响老的服务，这将大大提供系统的健壮性。

目前国内各大互联网公司广泛使用hessian、protobuf、**thrift**、avro等成熟的序列化解决方案来搭建RPC框架，这些都是久经考验的解决方案。

## 2.3  通信

消息数据结构被序列化为二进制串后，下一步就要进行网络通信了。目前有两种IO通信模型：1）BIO；2）NIO。

如何实现RPC的IO通信框架？1）使用java nio方式自研，这种方式较为复杂，而且很有可能出现隐藏bug，见过一些互联网公司使用这种方式；2）基于mina，mina在早几年比较火热，不过这些年版本更新缓慢；3）基于netty，现在很多RPC框架都直接基于netty这一IO通信框架，比如阿里巴巴的HSF、dubbo，Twitter的finagle等。

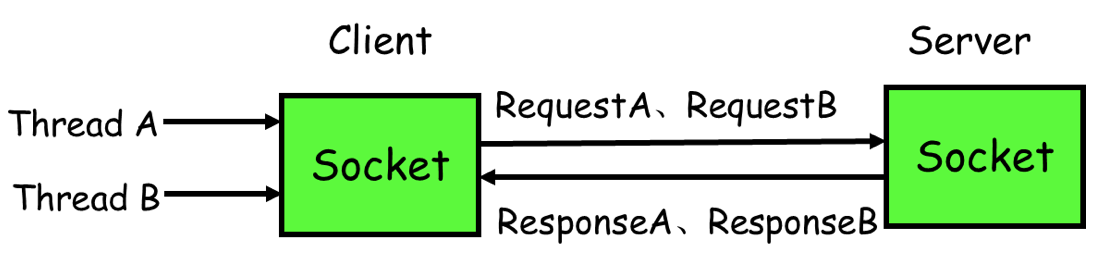
## 2.4  消息里为什么要带有requestID？

如果使用netty的话，一般会用channel.writeAndFlush()方法来发送消息二进制串，这个方法调用后对于整个远程调用(从发出请求到接收到结果)来说是一个异步的，即对于当前线程来说，将请求发送出来后，线程就可以往后执行了，至于服务端的结果，是服务端处理完成后，再以消息的形式发送给客户端的。于是这里出现以下两个问题：

1）怎么让当前线程“暂停”，等结果回来后，再向后执行？

2）如果有多个线程同时进行远程方法调用，这时建立在client server之间的socket连接上会有很多双方发送的消息传递，前后顺序也可能是随机的，server处理完结果后，将结果消息发送给client，client收到很多消息，怎么知道哪个消息结果是原先哪个线程调用的？

如下图所示，线程A和线程B同时向client socket发送请求requestA和requestB，socket先后将requestB和requestA发送至server，而server可能将responseA先返回，尽管requestA请求到达时间更晚。我们需要一种机制保证responseA丢给ThreadA，responseB丢给ThreadB。



怎么解决呢？

1）client线程每次通过socket调用一次远程接口前，生成一个唯一的ID，即requestID（requestID必需保证在一个Socket连接里面是唯一的），一般常常使用AtomicLong从0开始累计数字生成唯一ID；

2）将处理结果的回调对象callback，存放到全局ConcurrentHashMap里面put(requestID, callback)；

3）当线程调用channel.writeAndFlush()发送消息后，紧接着执行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。

4）服务端接收到请求并处理后，将response结果（此结果中包含了前面的requestID）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requestID，再从前面的ConcurrentHashMap里面get(requestID)，从而找到callback对象，再用synchronized获取callback上的锁，将方法调用结果设置到callback对象里，再调用callback.notifyAll()唤醒前面处于等待状态的线程。

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | public Object get() {          synchronized (this) {  // 旋锁              while (!isDone) {  // 是否有结果了                  wait(); //没结果是释放锁，让当前线程处于等待状态              }          }      } |

Java

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | private void setDone(Response res) {          this.res = res;          isDone = true;          synchronized (this) { //获取锁，因为前面wait()已经释放了callback的锁了              notifyAll(); // 唤醒处于等待的线程          }      } |

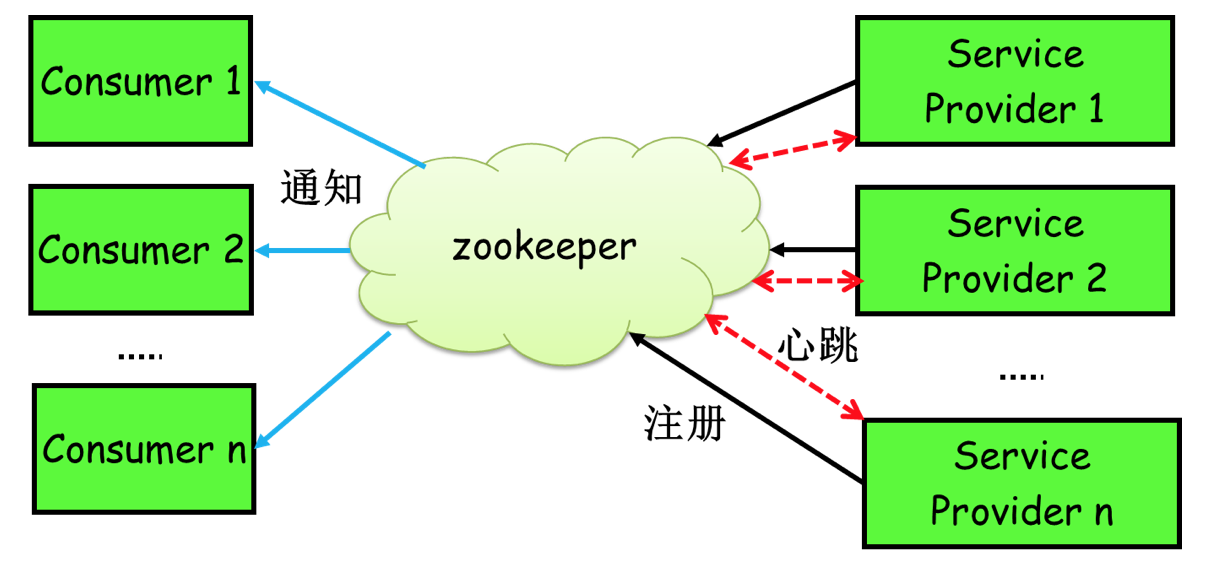
# 3 如何发布自己的服务？

如何让别人使用我们的服务呢？有同学说很简单嘛，告诉使用者服务的IP以及端口就可以了啊。确实是这样，这里问题的关键在于是自动告知还是人肉告知。

人肉告知的方式：如果你发现你的服务一台机器不够，要再添加一台，这个时候就要告诉调用者我现在有两个ip了，你们要轮询调用来实现负载均衡；调用者咬咬牙改了，结果某天一台机器挂了，调用者发现服务有一半不可用，他又只能手动修改代码来删除挂掉那台机器的ip。现实生产环境当然不会使用人肉方式。

有没有一种方法能实现自动告知，即机器的增添、剔除对调用方透明，调用者不再需要写死服务提供方地址？当然可以，现如今zookeeper被广泛用于实现服务自动注册与发现功能！

简单来讲，zookeeper可以充当一个服务注册表（Service Registry），让多个服务提供者形成一个集群，让服务消费者通过服务注册表获取具体的服务访问地址（ip+端口）去访问具体的服务提供者。如下图所示：



具体来说，zookeeper就是个分布式文件系统，每当一个服务提供者部署后都要将自己的服务注册到zookeeper的某一路径上: /{service}/{version}/{ip:port}, 比如我们的HelloWorldService部署到两台机器，那么zookeeper上就会创建两条目录：分别为/HelloWorldService/1.0.0/100.19.20.01:16888  /HelloWorldService/1.0.0/100.19.20.02:16888。

zookeeper提供了“心跳检测”功能，它会定时向各个服务提供者发送一个请求（实际上建立的是一个 socket 长连接），如果长期没有响应，服务中心就认为该服务提供者已经“挂了”，并将其剔除，比如100.19.20.02这台机器如果宕机了，那么zookeeper上的路径就会只剩/HelloWorldService/1.0.0/100.19.20.01:16888。

服务消费者会去监听相应路径（/HelloWorldService/1.0.0），一旦路径上的数据有任务变化（增加或减少），zookeeper都会通知服务消费方服务提供者地址列表已经发生改变，从而进行更新。

更为重要的是zookeeper 与生俱来的容错容灾能力（比如leader选举），可以确保服务注册表的高可用性。

# Dubbo实现RPC调用使用入门（阿里云例子）

使用Dubbo进行远程调用实现服务交互，它支持多种协议，如Hessian、HTTP、RMI、Memcached、Redis、Thrift等等。由于Dubbo将这些协议的实现进行了封装了，无论是服务端（开发服务）还是客户端（调用服务），都不需要关心协议的细节，只需要在配置中指定使用的协议即可，从而保证了服务提供方与服务消费方之间的透明。

另外，如果我们使用Dubbo的服务注册中心组件，这样服务提供方将服务发布到注册的中心，只是将服务的名称暴露给外部，而服务消费方只需要知道注册中心和服务提供方提供的服务名称，就能够透明地调用服务，后面我们会看到具体提供服务和消费服务的配置内容，使得双方之间交互的透明化。

**示例场景**

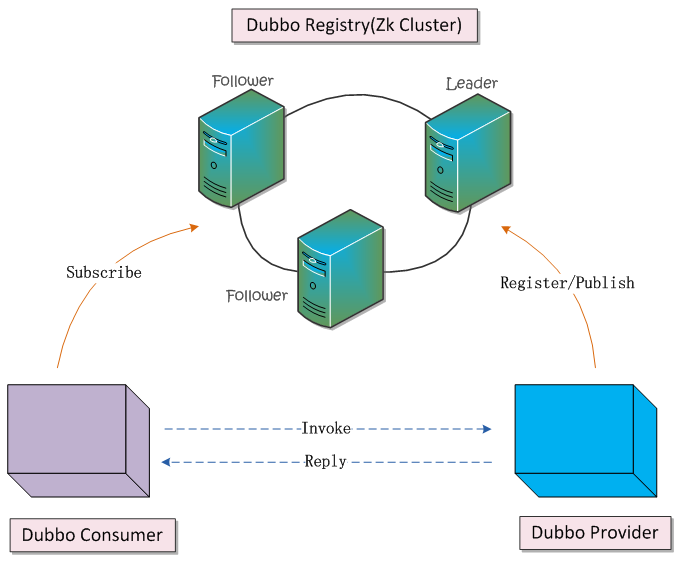
我们给出一个示例的应用场景：

服务方提供一个搜索服务，对服务方来说，它基于SolrCloud构建了搜索服务，包含两个集群，ZooKeeper集群和Solr集群，然后在前端通过Nginx来进行反向代理，达到负载均衡的目的。

服务消费方就是调用服务进行查询，给出查询条件（满足Solr的REST-like接口）。

**应用设计**

基于上面的示例场景，我们打算使用ZooKeeper集群作为服务注册中心。注册中心会暴露给服务提供方和服务消费方，所以注册服务的时候，服务先提供方只需要提供Nginx的地址给注册中心，但是注册中心并不会把这个地址暴露给服务消费方，如图所示：



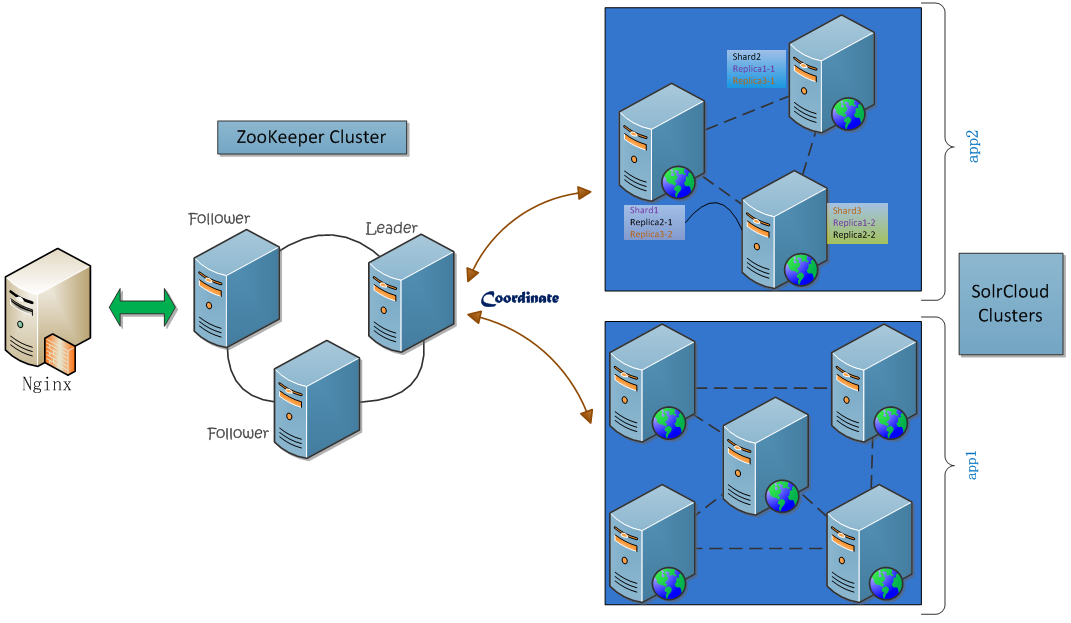
我们先定义一下，通信双方需要使用的接口，如下所示：

|  |  |
| --- | --- |
| 01 | **package** org.shirdrn.platform.dubbo.service.rpc.api; |
| 02 | |  |
| 03 | **public** **interface** SolrSearchService { |
| 04 |  |
| 05 | String search(String collection, String q, ResponseType type, **int** start, **int**rows); |
| 06 |  |
| 07 | **public** **enum** ResponseType { |
| 08 | JSON, |
| 09 | XML |
| 10 | } |
| 11 | } |

基于上图中的设计，下面我们分别详细说明Provider和Consumer的设计及实现。

* **Provider服务设计**

Provider所发布的服务组件，包含了一个SolrCloud集群，在SolrCloud集群前端又加了一个反向代理层，使用Nginx来均衡负载。Provider的搜索服务系统，设计如下图所示：



上图中，实际Nginx中将请求直接转发内部的Web Servers上，在这个过程中，使用ZooKeeper来进行协调：从多个分片（Shard）服务器上并行搜索，最后合并结果。我们看一下Nginx配置的内容片段：

|  |  |
| --- | --- |
| 01 | user nginx; |
| 02 | worker\_processes 4; |
| 03 |  |
| 04 | error\_log /var/log/nginx/error.log warn; |
| 05 | pid /var/run/nginx.pid; |
| 06 |  |
| 07 |  |
| 08 | events { |
| 09 | worker\_connections 1024; |
| 10 | } |
| 11 |  |
| 12 |  |
| 13 | http { |
| 14 | include /etc/nginx/mime.types; |
| 15 | default\_type application/octet-stream; |
| 16 |  |
| 17 | log\_format main '$remote\_addr - $remote\_user [$time\_local] "$request" ' |
| 18 | '$status $body\_bytes\_sent "$http\_referer" ' |
| 19 | '"$http\_user\_agent" "$http\_x\_forwarded\_for"'; |
| 20 |  |
| 21 | access\_log /var/log/nginx/access.log main; |
| 22 |  |
| 23 | sendfile on; |
| 24 | #tcp\_nopush on; |
| 25 |  |
| 26 | keepalive\_timeout 65; |
| 27 |  |
| 28 | #gzip on; |
| 29 |  |
| 30 | upstream master { |
| 31 | server slave1:8888 weight=1; |
| 32 | server slave4:8888 weight=1; |
| 33 | server slave6:8888 weight=1; |
| 34 | } |
| 35 |  |
| 36 | server { |
| 37 | listen 80; |
| 38 | server\_name master; |
| 39 | location / { |
| 40 | root /usr/share/nginx/html/solr-cloud; |
| 41 | index index.html index.htm; |
| 42 | proxy\_pass [http://master;](https://yq.aliyun.com/) |
| 43 | include /home/hadoop/servers/nginx/conf/proxy.conf; |
| 44 | } |
| 45 | } |
| 46 | } |

一共配置了3台Solr服务器，因为SolrCloud集群中每一个节点都可以接收搜索请求，然后由整个集群去并行搜索。最后，我们要通过Dubbo服务框架来基于已有的系统来开发搜索服务，并通过Dubbo的注册中心来发布服务。

首先需要实现服务接口，实现代码如下所示：

|  |  |
| --- | --- |
| 01 | **package** org.shirdrn.platform.dubbo.service.rpc.server; |
| 02 |  |
| 03 | **import** java.io.IOException; |
| 04 | **import** java.util.HashMap; |
| 05 | **import** java.util.Map; |
| 06 |  |
| 07 | **import** org.apache.commons.logging.Log; |
| 08 | **import** org.apache.commons.logging.LogFactory; |
| 09 | **import** org.shirdrn.platform.dubbo.service.rpc.api.SolrSearchService; |
| 10 | **import** org.shirdrn.platform.dubbo.service.rpc.utils.QueryPostClient; |
| 11 | **import** org.springframework.context.support.ClassPathXmlApplicationContext; |
| 12 |  |
| 13 | **public** **class** SolrSearchServer **implements** SolrSearchService { |
| 14 |  |
| 15 | **private** **static** **final** Log LOG = LogFactory.getLog(SolrSearchServer.**class**); |
| 16 | **private** String baseUrl; |
| 17 | **private** **final** QueryPostClient postClient; |
| 18 | **private** **static** **final** Map<ResponseType, FormatHandler> handlers = **new**HashMap<ResponseType, FormatHandler>(0); |
| 19 | **static** { |
| 20 | handlers.put(ResponseType.XML, **new** FormatHandler() { |
| 21 | **public** String format() { |
| 22 | **return** "&wt=xml"; |
| 23 | } |
| 24 | }); |
| 25 | handlers.put(ResponseType.JSON, **new** FormatHandler() { |
| 26 | **public** String format() { |
| 27 | **return** "&wt=json"; |
| 28 | } |
| 29 | }); |
| 30 | } |
| 31 |  |
| 32 | **public** SolrSearchServer() { |
| 33 | **super**(); |
| 34 | postClient = QueryPostClient.newIndexingClient(**null**); |
| 35 | } |
| 36 |  |
| 37 | **public** **void** setBaseUrl(String baseUrl) { |
| 38 | **this**.baseUrl = baseUrl; |
| 39 | } |
| 40 |  |
| 41 | **public** String search(String collection, String q, ResponseType type, |
| 42 | **int** start, **int** rows) { |
| 43 | StringBuffer url = **new** StringBuffer(); |
| 44 | url.append(baseUrl).append(collection).append("/select?").append(q); |
| 45 | url.append("&start=").append(start).append("&rows=").append(rows); |
| 46 | url.append(handlers.get(type).format()); |
| 47 | LOG.info("[REQ] " + url.toString()); |
| 48 | **return** postClient.request(url.toString()); |
| 49 | } |
| 50 |  |
| 51 | **interface** FormatHandler { |
| 52 | String format(); |
| 53 | } |
| 54 |  |
| 55 | **public** **static** **void** main(String[] args) **throws** IOException { |
| 56 | String config = SolrSearchServer.**class**.getPackage().getName().replace('.','/') + "/search-provider.xml"; |
| 57 | ClassPathXmlApplicationContext context = **new**ClassPathXmlApplicationContext(config); |
| 58 | context.start(); |
| 59 | System.in.read(); |
| 60 | } |
| 61 |  |
| 62 | } |

对应的Dubbo配置文件为search-provider.xml，内容如下所示：

|  |  |
| --- | --- |
| 01 | <?**xml** version="1.0" encoding="UTF-8"?> |
| 02 |  |
| 03 | <**beans** xmlns="<http://www.springframework.org/schema/beans>" |
| 04 | xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"xmlns:dubbo="<http://code.alibabatech.com/schema/dubbo>" |
| 05 | xsi:schemaLocation="<http://www.springframework.org/schema/beans><http://www.springframework.org/schema/beans/spring-beans-2.5.xsd> |
| 06 | <http://code.alibabatech.com/schema/dubbo><http://code.alibabatech.com/schema/dubbo/dubbo.xsd>"> |
| 07 |  |
| 08 | <**dubbo:application** name="search-provider" /> |
| 09 | <**dubbo:registry** address="zookeeper://slave1:2188?backup=slave3:2188,slave4:2188"/> |
| 10 | <**dubbo:protocol** name="dubbo" port="20880" /> |
| 11 | <**bean** id="searchService"class="org.shirdrn.platform.dubbo.service.rpc.server.SolrSearchServer"> |
| 12 | <**property** name="baseUrl" value="<http://nginx-lbserver/solr-cloud/>" /> |
| 13 | </**bean**> |
| 14 | <**dubbo:service**interface="org.shirdrn.platform.dubbo.service.rpc.api.SolrSearchService"ref="searchService" /> |
| 15 |  |
| 16 | </**beans**> |

上面，Dubbo服务注册中心指定ZooKeeper的地址：[zookeeper://slave1:2188?backup=slave3:2188,slave4:2188](http://shiyanjun.cn/archives/2188)，使用Dubbo协议。配置服务接口的时候，可以按照Spring的Bean的配置方式来配置，注入需要的内容，我们这里指定了搜索集群的Nginx反向代理地址<http://nginx-lbserver/solr-cloud/>。

* **Consumer调用服务设计**

这个就比较简单了，拷贝服务接口，同时要配置一下Dubbo的配置文件，写个简单的客户端调用就可以实现。客户端实现的Java代码如下所示：

|  |  |
| --- | --- |
| 01 | **package** org.shirdrn.platform.dubbo.service.rpc.client; |
| 02 |  |
| 03 | **import** java.util.concurrent.Callable; |
| 04 | **import** java.util.concurrent.Future; |
| 05 |  |
| 06 | **import** org.shirdrn.platform.dubbo.service.rpc.api.SolrSearchService; |
| 07 | **import** org.shirdrn.platform.dubbo.service.rpc.api.SolrSearchService.ResponseType; |
| 08 | **import** org.springframework.beans.BeansException; |
| 09 | **import** org.springframework.context.support.AbstractXmlApplicationContext; |
| 10 | **import** org.springframework.context.support.ClassPathXmlApplicationContext; |
| 11 |  |
| 12 | **import** com.alibaba.dubbo.rpc.RpcContext; |
| 13 |  |
| 14 | **public** **class** SearchConsumer { |
| 15 |  |
| 16 | **private** **final** String collection; |
| 17 | **private** AbstractXmlApplicationContext context; |
| 18 | **private** SolrSearchService searchService; |
| 19 |  |
| 20 | **public** SearchConsumer(String collection, Callable<AbstractXmlApplicationContext> call) { |
| 21 | **super**(); |
| 22 | **this**.collection = collection; |
| 23 | **try** { |
| 24 | context = call.call(); |
| 25 | context.start(); |
| 26 | searchService = (SolrSearchService) context.getBean("searchService"); |
| 27 | } **catch** (BeansException e) { |
| 28 | e.printStackTrace(); |
| 29 | } **catch** (Exception e) { |
| 30 | e.printStackTrace(); |
| 31 | } |
| 32 | } |
| 33 |  |
| 34 | **public** Future<String> asyncCall(**final** String q, **final** ResponseType type, **final** **int**start, **final** **int** rows) { |
| 35 | Future<String> future = RpcContext.getContext().asyncCall(**new** Callable<String>() { |
| 36 | **public** String call() **throws** Exception { |
| 37 | **return** search(q, type, start, rows); |
| 38 | } |
| 39 | }); |
| 40 | **return** future; |
| 41 | } |
| 42 |  |
| 43 | **public** String syncCall(**final** String q, **final** ResponseType type, **final** **int** start,**final** **int** rows) { |
| 44 | **return** search(q, type, start, rows); |
| 45 | } |
| 46 |  |
| 47 | **private** String search(**final** String q, **final** ResponseType type, **final** **int** start,**final** **int** rows) { |
| 48 | **return** searchService.search(collection, q, type, start, rows); |
| 49 | } |
| 50 |  |
| 51 | **public** **static** **void** main(String[] args) **throws** Exception { |
| 52 | **final** String collection = "tinycollection"; |
| 53 | **final** String beanXML = "search-consumer.xml"; |
| 54 | **final** String config = SearchConsumer.**class**.getPackage().getName().replace('.','/') + "/" + beanXML; |
| 55 | SearchConsumer consumer = **new** SearchConsumer(collection, **new**Callable<AbstractXmlApplicationContext>() { |
| 56 | **public** AbstractXmlApplicationContext call() **throws** Exception { |
| 57 | **final** AbstractXmlApplicationContext context = **new**ClassPathXmlApplicationContext(config); |
| 58 | **return** context; |
| 59 | } |
| 60 | }); |
| 61 |  |
| 62 | String q = "q=上海&fl=\*&fq=building\_type:1"; |
| 63 | **int** start = 0; |
| 64 | **int** rows = 10; |
| 65 | ResponseType type = ResponseType.XML; |
| 66 | **for** (**int** k = 0; k < 10; k++) { |
| 67 | **for** (**int** i = 0; i < 10; i++) { |
| 68 | start = 1 \* 10 \* i; |
| 69 | **if**(i % 2 == 0) { |
| 70 | type = ResponseType.XML; |
| 71 | } **else** { |
| 72 | type = ResponseType.JSON; |
| 73 | } |
| 74 | // String result = consumer.syncCall(q, type, start, rows); |
| 75 | // System.out.println(result); |
| 76 | Future<String> future = consumer.asyncCall(q, type, start, rows); |
| 77 | // System.out.println(future.get()); |
| 78 | } |
| 79 | } |
| 80 | } |
| 81 | } |

查询的时候，需要提供查询字符串，符合Solr语法，例如“q=上海&fl=\*&fq=building\_type:1”。配置文件，我们使用search-consumer.xml，内容如下所示：

|  |  |
| --- | --- |
| 01 | <?**xml** version="1.0" encoding="UTF-8"?> |
| 02 |  |
| 03 | <**beans** xmlns="<http://www.springframework.org/schema/beans>" |
| 04 | xmlns:xsi="<http://www.w3.org/2001/XMLSchema-instance>"xmlns:dubbo="<http://code.alibabatech.com/schema/dubbo>" |
| 05 | xsi:schemaLocation="<http://www.springframework.org/schema/beans><http://www.springframework.org/schema/beans/spring-beans-2.5.xsd> |
| 06 | <http://code.alibabatech.com/schema/dubbo><http://code.alibabatech.com/schema/dubbo/dubbo.xsd>"> |
| 07 |  |
| 08 | <**dubbo:application** name="search-consumer" /> |
| 09 | <**dubbo:registry** address="zookeeper://slave1:2188?backup=slave3:2188,slave4:2188"/> |
| 10 | <**dubbo:reference** id="searchService"interface="org.shirdrn.platform.dubbo.service.rpc.api.SolrSearchService" /> |
| 11 |  |
| 12 | </**beans**> |

**运行说明**

首先保证服务注册中心的ZooKeeper集群正常运行，然后启动SolrSearchServer，启动的时候直接将服务注册到ZooKeeper集群存储中，可以通过ZooKeeper的客户端脚本来查看注册的服务数据。一切正常以后，可以启动运行客户端SearchConsumer，调用SolrSearchServer所实现的远程搜索服务