注：

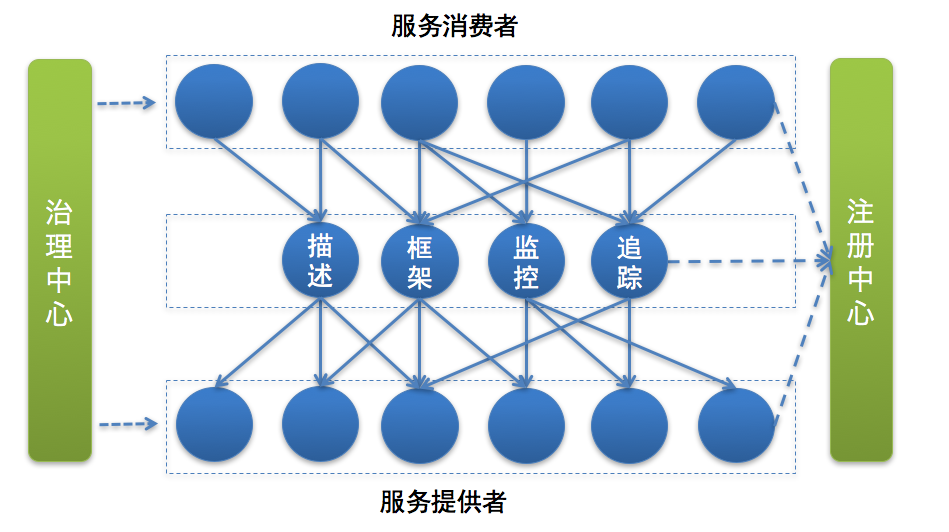
本笔记内容当前主要来源：极客时间专栏-从0开始学微服务[作者：胡忠想]

（如有侵权，请联系我进行处理）

本笔记作者 ：bulingma [微信号：HurricaneMa]

# 3、初探微服务架构

## 3.1、微服务架构模块图



服务提供者（提供服务的一方）：按照一定格式的服务描述，向注册中心注册服务，声明自己能够提供哪些服务以及服务的地址是什么，完成**服务发布**。

服务消费者（调用服务的一方）：请求**注册中心**，查询所需要调用服务的地址，然后以约定的通信协议向服务提供者发起请求，得到请求结果后再按照约定的协议解析结果。

而且在服务的调用过程中，服务的请求耗时、调用量以及成功率等指标都会被记录下来用作**监控**；调用经过的链路信息会被记录下来，用于故障定位和**问题追踪**。在这期间，如果调用失败，可以通过重试等**服务治理**手段来保证成功率。

总结如下，微服务架构中，服务调用主要依赖下面几个基本组件：

**1、服务描述**

**2、注册中心**

**3、服务框架**

**4、服务监控**

**5、服务追踪**

**6、服务治理**

## 3.2、服务描述

服务描述需要解决的问题：服务名、调用服务的入参、服务返回结果格式、如何解析

常用的服务描述方式：

**RESTful API：**通常用于HTTP协议的服务描述，并且常用Wiki或者Swagger来进行管理

**XML配置：**多用做RPC协议的服务描述，通过\*.xml配置文件来定义接口名、参数、返回值类型

**IDL文件：**通常用作**Thrift**和**gRPC**这类跨语言服务调用框架。如gRPC就是通过Protobuf文件来定义服务的接口名、参数以及返回值的数据结构。

## 3.3、注册中心

你提供了一个服务，但是如何让外部想调用你的服务的人知道。这个时候就需要一个类似注册中心的角色，**服务提供者将自己提供的服务以及地址登记到注册中心，服务消费者则从注册中心查询所需要调用的服务的地址，然后发起请求。**

一般的注册中心的工作流程如下：

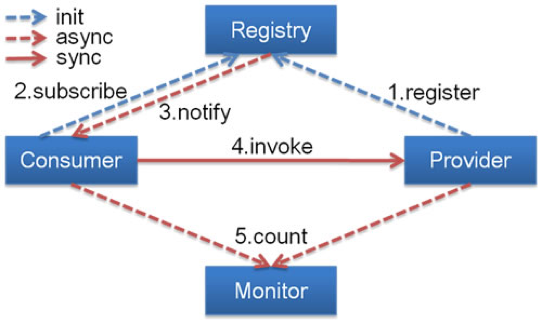
（1）、服务提供者启动时，根据**服务发布文件**中配置的发布信息向注册中心注册自己的服务。

（2）、服务消费者启动时，根据**消费者配置文件**中配置的服务信息向注册中心**订阅**自己所需要的服务。

（3）、注册中心返回服务提供者地址列表给服务消费者。

（4）、当服务提供者发生变化，比如节点新增或销毁，注册中心将变更通知给服务消费者。

注册中心的工作流程



## 3.4、服务框架

通过注册中心，服务消费者就可以获取到服务提供者的地址，有了地址后就可以发起调用。但在发起调用之前还需要解决以下几个问题：

**服务通信采用什么协议？：**

服务提供者和服务消费者之间以什么样的协议进行网络通信，是四层TCP、UDP协议，还是七层的HTTP协议，还是其他协议？

**数据传输采用什么方式？：**

服务调用者和服务消费者之间的数据传输采用**同步还是异步**，**单链接传输还是多路复用**？

**数据压缩采用什么格式？：**

通常数据传输都会对数据进行压缩，来减少网络传输的数据量，从而**减少带宽消耗**和**网络传输时间**，比如常见的**JSON序列化**、**Java对象序列化**以及**Protobuf序列化**等。

## 3.5、服务监控

一旦服务消费者和服务提供者之间能够正常发起服务调用，就需要对调用情况进行监控，以了解服务是否正常。通常来讲，服务监控主要包括三个流程：

**指标收集：**

把每次服务调用的**请求耗时**以及**成功与否**收集起来，并上传到集中的**数据处理中心**。

**数据处理：**

有了每次调用的请求耗时以及成功与否等信息，就可以计算**每秒服务请求量**、**平均耗时**以及**成功率**等指标。

**数据展示：**

数据收集起来，经过处理后，还需要以友好的方式对外展示，才能发挥价值。通常都是将数据展示在**Dashboard面板**上，并且每隔10s等间隔自动刷新，用作**业务监控和报警**等。

## 3.6、服务追踪

除了需要对服务调用情况进行监控之外，还需要记录服务调用经过的每一层链路，以便进行问题追踪和故障定位。

服务追踪的工作原理大致如下：

服务调用者发起调用前，会在本地按照一定的规则生成一个requestid，发起调用时，将requestid当做请求参数的一部分，传递给服务提供者。

服务提供者接收到请求后，记录下这次请求的requestid，然后处理请求。如果服务提供者继续请求其他服务，会在本地再生成一个自己的requestid，然后把这两个requestid都当做参数继续往下传递。

以此类推，通过这种层层往下传递的方式，一次请求，无论最后依赖多少次服务调用、经过多少次服务节点，都可以通过最开始生成的requestid串联所有节点，从而达到服务追踪的目的。

## 3.7、服务治理

服务监控能够发现问题，服务追踪能够定位问题所在，而解决问题就得靠服务治理了。服务治理就是通过**一系列的手段来保证在各种意外情况下，服务调用仍然能够正常进行**。

在生产环境中，应该会遇到下面的几种状况：

**单机故障：**

通常遇到的单机故障，都是靠运维发现并重启服务或者从线上摘除故障节点。当集群规模越大，靠传统的人肉运维显然难以对付。而服务治理可以通过一定的策略，自动摘除故障节点，不需要人为干预，就能保障单机故障不影响业务。

**单IDC故障：**

服务治理可以自动切换故障IDC的流量到其他正常IDC，可以避免因为单IDC故障引起的大批量业务受影响。

**依赖服务不可用：**

比如你的服务依赖了另一个服务，当另一个服务出现问题时，会拖慢甚至拖垮你的服务。而服务治理可以通过限流，在依赖服务异常情况下，一段时期内停止发起调用而直接返回。这样**一方面保证了服务消费者能够不被拖垮，另一方面也给服务提供者减少压力，使其能够尽快恢复。**

以上三种是最常见的需要引入服务治理的场景，还有一些其他服务治理的手段，如：自动扩缩容（解决服务的容量问题）

## 3.8、总结

以上这几个基本组件共同组成了微服务框架，在生产环境中缺一不可，所以在引入微服务架构之前，你的团队必须掌握这些基本组件的原理并具备相应的开发能力。实现方式上，可以引入开源方案，也可以自行研发微服务架构中的每个组件。但是**不管你是采用开源方案还是自行研发，都必须吃透每个组件的工作原理并能在此基础上进行二次开发。**

# 4、服务发布和引用

想要构建微服务，首先要解决的问题是，服务提供者如何发布一个服务，服务消费者如何引用这个服务。具体来说，就是这个服务的接口名是什么？调用这个服务需要传递哪些参数？接口的返回值是什么类型？以及一些其他接口描述信息。

常见的服务发布和引用的方式有三种：RESTful API、XML配置、IDL文件

下面逐个讲解具体使用方法和各自的应用场景。

## 4.1、RESTful API

主要被用作HTTP或HTTPS协议的接口定义，即使在非微服务架构体系下，也被广泛采用。

服务提供者这一端通过部署代码到Tomcat中，并配置Tomcat中web.xml，就可以通过servlet的方式对外提供RESTful API，服务消费者就可以通过HTTP协议调用服务了。

由于HTTP是一个公开的协议，所以**比较适合用作跨业务平台之间的服务协议**。比如你有一个服务，不仅需要在业务部门内部提供服务，还需要向其他业务部门提供服务，甚至开放给外网提供服务，这时候采用HTTP协议就比较适合，也省去了沟通服务协议的成本。

## 4.2、XML配置

XML配置的这种服务发布和引用主要分为三个步骤：

1. 服务提供者定义接口，并实现接口。
2. 服务提供者进程启动时，通过加载server.xml配置文件将接口暴露出去。
3. 服务消费者进程启动时，通过加载client.xml配置文件来引用要调用的接口。

XML配置通过服务提供者和服务消费者之间维持一份对等的XML配置文件，来保证服务消费者按照服务提供者的约定来进行服务调用。在这种方式下，如果服务提供者变更了接口定义，不仅需要更新服务提供者加载的接口描述文件server.xml，还需要同时更新服务消费者加载的接口描述文件client.xml。

一般是**私有RPC框架会选择XML配置这种方式**来描述接口，因为私有RPC协议的性能要比HTTP协议高，**所以在对性能要求比较高的场景下，采用XML配置的方式比较适合**。但这种方式对业务代码侵入性比较高，XML配置有变更的时候，服务消费者和服务提供者都需要更新，所以**适合公司内部联系比较紧密的业务之间采用**。如果要应用到跨部门之间的业务调用，一旦有XML配置变更，需要花费大量精力去协调不同部门做升级工作。

对于XML配置方式的服务描述，一旦应用到多个部门之间的接口格式约定，如果有变更，最好是新增接口，不到万不得已不要对原有的接口格式做变更。

## 4.3、IDL文件

IDL是接口描述文件（interface description language）的缩写，**通过一种中立的方式来描述接口，使得在不同的平台上运行的对象和不同语言编写的程序可以互相通信交流**。也就是说，**IDL主要是用作跨语言平台的服务之间的调用。**

有两种最常用的IDL：**Facebook开源的thrift协议（tcp）、 Google开源的gRPC协议（http2）。**不论是Thrift协议还是gRPC协议，它们的工作原理都是类似的。

下面以gRPC协议为例，讲讲如何使用IDL文件方式来描述接口。

**gRPC协议使用Protobuf简称Proto文件（xxx.proto）来定义接口名、调用参数以及返回值类型**。**然后再使用protoc插件来生成不同语言平台的客户端和服务端代码，从而具备跨语言服务调用能力**。假如服务提供者和服务调用者使用的是java语言，那么利用**protoc插件**即可自动生成Server端、Client端的Java代码。假如服务消费者使用的是PHP语言，那么利用protoc插件即可自动生成Client端的PHP代码。

**在描述接口定义时，IDL文件需要对接口返回值进行详细定义。如果接口返回值的字段比较多，并且经常变化，采用IDL文件方式的接口就不太合适了。一方面可能会造成IDL文件过大难以维护，另一方面只要IDL文件中定义的接口返回值有变更，都需要同步所有的服务消费者都更新，管理成本就太高了。**

## 4.4、总结

具体采用哪种服务描述方式是根据实际情况决定的。通常情况下，参考如下：

XML配置： 企业内部之间的服务调用，并且都是Java语言的

IDL文件： 企业内部存在多个服务，服务采用不同语言

RESTful API：存在对外开放服务调用情况



# 5、如何注册和发现服务

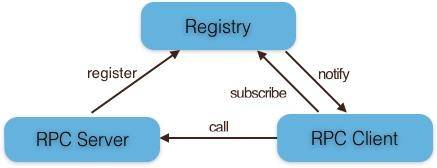
上一讲，我们已经介绍了服务发布和引用常用的三种方式：RESTful API、XML配置和IDL文件。假设你已经使用其中一种方式发布了一个服务，并且已经在一台机器上部署了服务，那**如果我想调用这个服务，我该如何知道你部署的这台机器的地址呢？**

这个问题就像我们现实生活中去肯德基前先使用谷歌地图搜索目的地一样。谷歌地图就扮演了一个类似注册中心的角色，收录了所有肯德基店面的地址。

同理，我想知道这台服务器的地址，那是不是可以去一个类似“谷歌地图”的地方去查询呢？是的，在**分布式系统里，就有这么一个概念，叫注册中心**。它是将服务的机器地址记录到注册中心，服务消费者在有需要的时候，只需要查询注册中心，输入提供的服务名，就可以得到地址，从而发起服务调用。

## 5.1、注册中心原理

在微服务架构下，主要有三种角色：服务提供者（RPC Server）、服务消费者（RPC Client）和服务注册中心（Registry），三者的交互关系见下图



解释一下：

**RPC Server提供服务**：在启动时，根据服务发布文件server.xml中的配置信息，向Registry注册自身服务，并向Registry定期发送心跳汇报存活状态。

**PRC Client调用服务：**在启动时，根据服务应用client.xml中的配置信息，向Registry订阅服务，把Registry返回的服务节点列表缓存在本地内存中，并与RPC Server建立连接。

**节点变更：**当RPC Server节点发生变更时，Registry会同步变更，RPC Client感知后会刷新本地内存中缓存的服务节点列表。

**服务调用：**RPC Client从本地缓存的服务节点列表中，基于负载均衡算法选择一台RPC Server发起调用。

## 5.2、注册中心实现方式

注册中心的实现主要涉及几个问题：

1. 注册中心需要提供哪些接口，该如何部署；
2. 如何存储服务信息；
3. 如何监控服务提供者节点的存活状况；
4. 如果服务提供者节点有变化如何通知服务消费者，以及如何控制注册中心的访问权限；

### 5.2.1、注册中心API

根据注册中心原理的描述，注册中心必须提供以下最基本的API，例如：

1. 服务注册接口：服务提供者通过调用服务注册接口来完成服务注册；
2. 服务反注册接口：服务提供者通过调用服务反注册接口来完成服务注销；
3. 心跳汇报接口：服务提供者通过调用心跳汇报接口，完成节点存活状态上报；
4. 服务订阅接口：服务消费者通过调用服务订阅接口完成服务订阅，获取可用的服务提供者节点列表；
5. 服务变更查询接口：服务消费者通过调用服务变更查询接口，获取最新的可用服务节点列表；

除此之外，为了便于管理，注册中心还必须提供一些后台管理的API，如：

1. 服务查询接口：查询注册中心当前注册了哪些服务信息；
2. 服务修改接口：修改注册中心中某一服务的信息。

### 5.2.2、集群部署

注册中心作为服务提供者和服务消费者之间沟通的桥梁，它的重要性不言而喻。所以注册中心一般都是采用**集群部署**来保证**高可用性**，并**通过分布式一致性协议**来确保集群中不同节点之间的**数据保持一致**。

以开源注册中心ZooKeeper为例，ZooKeeper集群中包含多个节点，服务提供者和服务消费者可以同任意一个节点通信，因为它们的数据一定是相同的。下面介绍下ZooKeeper的工作原理：

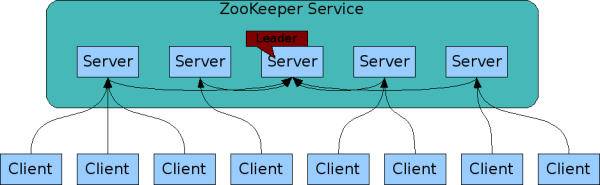
Tip1:每个Server在内存中存储了一份数据，Client的读请求可以请求任意一个Server；

Tip2:ZooKeeper启动时，将从实例中选举一个leader（Paxos协议）；

Tip3:Leader负责处理数据更新等操作（ZAB协议）；

Tip4:一个更新操作成功，当且仅当大多数Server在内存中成功修改；

通过上面这种方式，ZooKeeper保证了高可用性以及数据一致性。

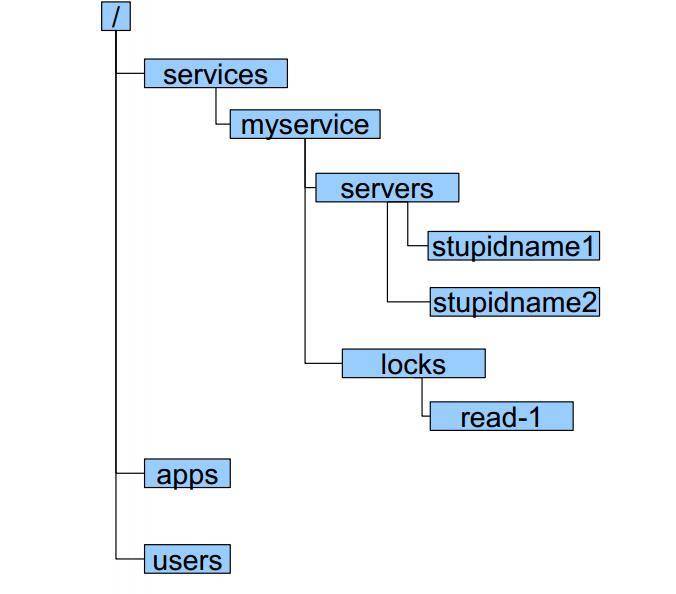


### 5.2.3、目录存储

以ZooKeeper为例，注册中心存储服务信息一般采用层次化的目录结构：

Tip1:每个目录在ZooKeeper中叫做znode，并且其有一个唯一的路径标识；

Tip2:znode可以包含数据和子znode

Tip3:znode中的数据可以有多个版本，比如某一个znode下存有多个数据版本，那么查询这个路径下的数据需带上版本信息。

### 5.2.4、服务健康状态监测

注册中心除了要支持最基本的服务注册和服务订阅功能外，还必须具备对服务提供者节点的健康状态监测功能，这样才能保证注册中心里保存的服务节点都是可用的。

以ZooKeeper为例，它是基于**ZooKeeper客户端和服务端的长链接和会话超时控制机制**，来实现服务健康状态检测的。

在ZooKeeper中，客户端和服务端建立连接后，会话也随之建立，并生成一个全局唯一的Session ID。服务端和客户端维持的是一个长链接，在SESSION\_TIMEOUT周期内，服务端会检测与客户端的链路是否正常，具体方式是通过客户端定时向服务端发送心跳消息（ping消息），服务器重置下次SESSION\_TIMEOUT时间。如果超过SESSION\_TIMEOUT后服务端都没有收到客户端的心跳消息，则服务端认为这个Session就已经结束了，ZooKeeper就会认为这个服务节点已经不可用，将会从注册中心中删除其信息。

### 5.2.5、服务状态变更通知

一旦注册中心探测到有服务提供者节点新加入或者被剔除，就必须立刻通知所有订阅该服务的服务消费者，刷新本地缓存的服务节点信息，确保服务调用不会请求不可用的服务提供者节点。

以ZooKeeper为例，基于ZooKeeper的Watcher机制，来实现服务状态变更通知给服务消费者的。服务消费者在调用ZooKeeper的getData方法订阅服务时，还可以通过监听器Watcher的process方法获取服务的变更，然后调用getData方法来获取变更后的数据，刷新本地缓存的服务节点信息。

### 5.2.6、白名单机制

在实际的微服务测试和部署时，通常包含多套环境，比如生产环境一套、测试环境一套。开发在进行业务自测、测试在进行回归测试时，一般都是用测试环境，部署的RPC Server节点注册到测试的注册中心集群。但经常会出现开发或者测试在部署时，错误的把测试环境下的服务节点注册到了线上注册中心集群，这样的话线上流量就会调用测试环境下的RPC Server节点，可能会造成意想不到的后果。

为了防止这种情况发生，注册中心需要提供一个保护机制，你可以把注册中心想象成一个带有门禁的房间，只有拥有门禁卡的RPC Server才能进入。**在实际应用中，注册中心可以提供一个白名单机制，只有添加到注册中心白名单内的RPC Server，才能够调用注册中心的注册接口，这样的话可以避免测试环境中的节点意外跑到线上环境中去。**

## 5.3、总结

注册中心可以说是实现服务化的关键，因为服务化之后，服务提供者和服务消费者不在同一个进程中运行，**实现了解耦**，这就需要一个纽带去连接服务提供者和服务消费者，而注册中心就正好承担了这一角色。

此外，**服务提供者可以任意伸缩即增加节点或者减少节点**，通过服务健康状态监测，注册中心可以保持最新的服务节点信息，并将变化通知给订阅服务的服务消费者。

注册中心一般采用分布式集群部署，来保证高可用性，并且为实现异步多活，有的注册中心还采用多IDC部署，这就对数据一致性产生了很高的要求，这些都是注册中心在实现时必须要解决的问题。

## 5.4、思考题

采用注册中心来实现服务发现与传统的DNS实现服务发现有什么不同？

1. 注册中心的服务注册和发现都是基于API的，可以自动化注册与发现，DNS则是人工注册，维护管理比较麻烦。这样导致前者实时性、容错性好与后者；
2. 注册中心可以注册http、rpc等各种服务，dns只能注册http服务；
3. 注册中心对已注册的服务会有主动的、自动化的健康检查机制，dns则没有；
4. 注册中心一般是一级分布式的，dns则是多级架构，例如根域名服务器、权威域名服务器等；
5. 注册中心的安全机制相对dns弱一些，毕竟是内部使用。
6. dns更新后生效有延迟
7. dns方式，客户端一般只能同时连接到一个server ip，无法做请求的负载均衡，而注册中心的方式通常客户端会与所有server建立连接形成连接池，从而在调用端实现请求的负载均衡。

# 6、如何实现RPC远程服务调用

有了服务提供者的地址后，服务消费者就可以向这个地址发起请求了，但这时候也产生了一个新的问题。我们知道，在**单体应用**时，一次服务调用发生在同一台机器上的同一个进程内部，也就是说调用发生在本机内部，因此也被叫作**本地方法调用**。在进行服务化拆分后，**服务提供者和服务消费者运行在两台不同物理机上的不同进程**内，它们之间的调用相比于本地方法调用，可称之为远程方法调用，简称**RPC（Romote Procedure Call）**，那么RPC调用是如何实现的呢？

RPC调用中服务消费者（客户端）和服务提供者（服务端）通常位于网络上两个不同的地址，要完成一次RPC调用，就必须先建立网络连接。建立连接后，双方还必须按照某种约定的协议进行网络通信，这个协议就是**通信协议**。双方能够正常通信后，服务端接收到请求时，需要以某种方式进行处理，处理成功后，把请求结果返回给客户端。**为了减少传输的数据大小，还需要对数据进行压缩，也就是对数据进行序列化**。

由此可见，想要完成调用，需要解决四个问题：

（1）、客户端和服务端如何建立网络连接？

（2）、服务端如何处理请求？

（3）、数据传输采用什么协议？

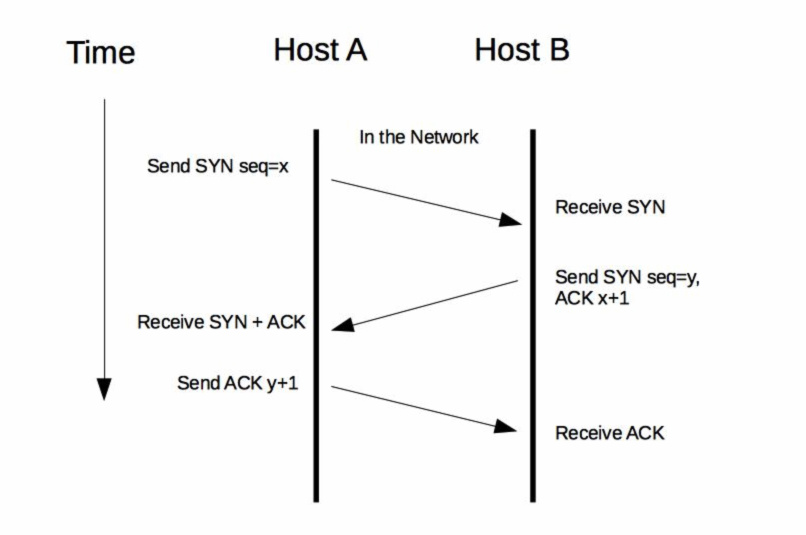
（4）、数据该如何序列化和反序列化？

## 6.1、客户端和服务端建立网络连接

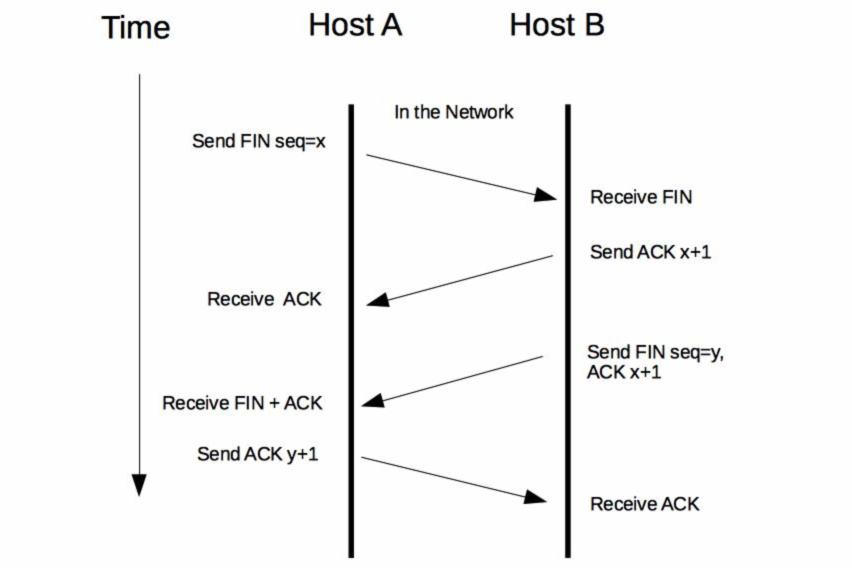
根据作者的实践经验，客户端和服务端之间**基于TCP协议**建立网络连接最常用的途径有两种。

### 6.1.1、HTTP通信

**HTTP通信是基于应用层HTTP协议的，而HTTP协议又是基于传输层TCP协议的**。一次HTTP通信过程就是发起一次HTTP调用，而一次HTTP调用就会建立一个TCP连接，经历一次下图所示的“三次握手”的过程来建立连接。

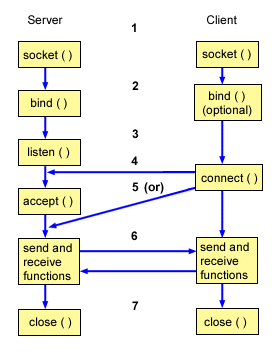


完成请求后，在经历一次“四次挥手”的过程来断开连接。



### 6.1.2、Socket通信

Socket通信是基于TCP/IP协议的封装，建立一次Socket连接至少需要一对套接字，其中一个运行于客户端，称为ClientSocket；另一个运行于服务端，称为ServerSocket。就像下图所描述的，**Socket通信的过程分为四个步骤**：**服务器监听、客户端请求、连接确认、数据传输**。



**A：服务器监听**：

ServerSocket通过调用bind()函数绑定某个具体端口，然后调用listen()函数实时监控网络状态，等待客户端的连接请求；

**B：客户端请求**：

ClientSocket调用connect()函数向ServerSocket绑定的地址和端口发起连接请求；

**C：服务端连接确认**：

当ServerSocket监听到或者接受到ClientSocket的连接请求时，调用accept()函数响应ClientSocket的请求，同客户端建立连接；

**D：数据传输**：

当ClientSocket和ServerSocket建立连接后，ClientSocket调用send()函数，ServerSocket调用receive函数，ServerSocket处理完请求后，调用send()函数，ClientSocket调用receive()函数，就可以得到返回结果。

当客户端和服务端建立网络连接后，就可以发起请求了。但网络不一定总是可靠的，经常会遇到**网络闪断、连接超时、服务端宕机**等各种异常，通常的处理手段有两种。

**Tips1、链路存活检测：**

客户端需要定时地发送心跳检测消息（一般是通过ping请求）给服务端，如果服务端**连续n次心跳检测**或者**超过规定的时间**都没有回复消息，则认为此时链路已经失效，这个时候客户端就需要重新与服务端建立连接。

**Tips2、断连重试：**

通常有多种情况会导致连接断开，比如客户端主动关闭、服务端宕机或者网络故障等。这个时候客户端就需要与服务端重新建立连接，但一般不能立刻完成重连，而是要等待固定的间隔后再发起重连，避免服务端的连接回收不及时，而客户端瞬间重连的请求太多而把服务端的连接数沾满。

## 6.2、服务端处理请求

假设这时候客户端和服务端已经建立了网络连接，服务端又该如何处理客户端的请求呢？通常来讲，有三种处理方式。

1. **同步阻塞方式（BIO:Blocking IO）**：

客户端每发一次请求，服务端就生成一个线程去处理，当客户端同时发起的请求很多时，服务端需要创建很多线程去处理每一个请求，如果达到了系统最大的线程数瓶颈，新来的请求就没法处理了；

1. **同步非阻塞方式（NIO:Non-blocking IO）：**

客户端每发一次请求，服务端并不是每次都创建一个新线程来处理，而是通过**I/O多路复用**技术进行处理。就是**把多个I/O的阻塞复用到同一个select的阻塞上**，从而使系统在单线程的情况下可以同时处理多个客户端请求。这种方式的优势是开销小，不用为每个请求创建一个线程，可以节省系统开销。

1. **异步非阻塞方式（AIO：Asynchronous IO）:**

**客户端只需要发起一个I/O操作然后立即返回**，等I/O操作真正完成以后，客户端会得到I/O操作完成的**通知**，此时客户端只需要对数据进行处理就好了，**不需要进行实际的I/O读写操作**，因为真正的I/O读取或者写入操作已经由内核完成了。这种方式的优势是客户端无需等待，不存在阻塞等待问题。

从前面的描述可以看出，不同的处理方式适用于不同的业务场景，根据作者的经验：

⦁ **BIO适用于连接数比较小的业务场景，**这样的话不至于系统中没有可用线程去处理请求。这种方式写的程序也比较简单直观，易于理解。

⦁  **NIO适用于连接数比较多并且请求消耗比较轻的业务场景**，比如聊天服务器。这种方式相比于BIO，相对来说编程比较复杂。

⦁  **AIO适用于连接数比较多并且请求消耗比较重的业务场景**，比如涉及I/O操作的相册服务器。这种方式相比于前两种，编程难度最大，程序也不易于理解。

上面两个问题是“通信框架”要解决的问题，你可以基于现有的Socket通信，在服务消费者和服务提供者之间建立网络连接，然后在服务提供者一侧基于BIO、NIO、AIO三种方式中的任意一种实现服务端请求处理，最后再花费一些精力去解决服务消费者和服务提供者之间的网络可靠性问题。这种方式对**Socket网络编程、多线程编程知识**都**要求比较高**，感兴趣的话可以尝试自己实现一个通信框架。但是作者建议最为稳妥的方式是使用成熟的开源方案，比如Netty，MINA等，它们都是经过业界大规模应用后，被充分论证是很可靠的方案。

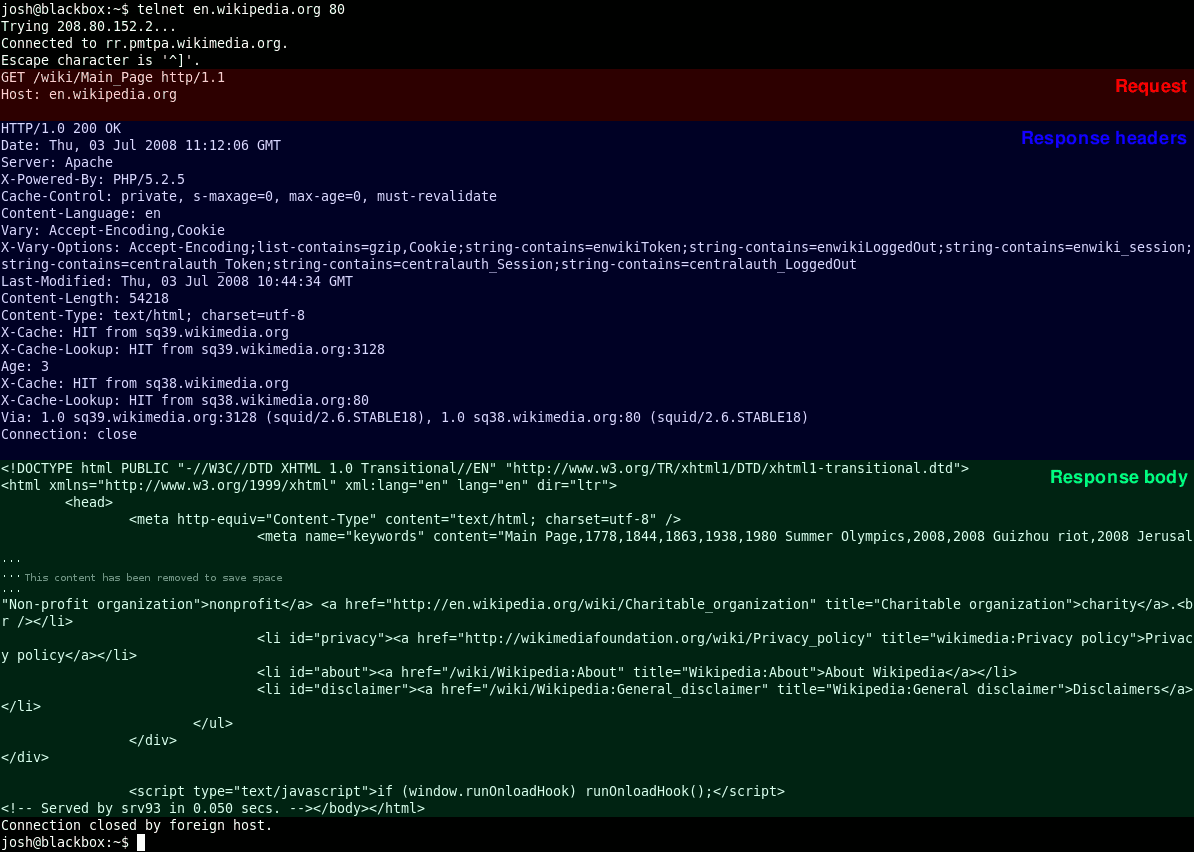
假如客户端和服务端的连接已经建立了，服务端也能正确地处理请求了，接下来完成一次正常的RPC调用还需要解决两个问题，即**数据传输采用什么协议**以及**数据该如何序列化和反序列化**。

## 6.3、数据传输采用什么协议

最常用的有**HTTP协议**，它是一种**开放的协议**，各大网站的服务器和浏览器之间的数据传输大都采用了这种协议。还有一些**定制的私有协议**，比如阿里巴巴开源的**Dubbo协议**，也可以用于服务端和客户端之间的数据传输。无论是开放的还是私有协议，都必须定义一个“契约”，以便服务消费者和服务提供者之间能够达成共识。服务消费者按照契约，对传输的数据进行**编码**，然后通过网络传输过去；服务提供者从网络上接受到数据后，按照契约，对传输的数据进行**解码**，然后处理请求，再把处理后的结果进行编码，通过网络传输返回给服务消费者；服务消费者再对返回的结果进行解码，最终得到服务提供者处理后的返回值。

**通常协议契约包含两个部分**：**消息头和消息体。**其中消息头存放的是**协议的公共字段**以及**用户扩展字段**，消息体存放的是**传输数据的具体内容**。

以HTTP协议为例，下图展示了一段采用HTTP协议传输的数据响应报文，主要分为消息头和消息体两部分，其中消息头存放的是协议的公共字段（比如Server代表是服务端服务器类型、Content-Length代表返回数据的长度、Content-Type代表返回数据的类型），消息体存放的是具体的返回结果（这里就是一段HTML网页代码）。



## 6.4、数据该如何序列化和反序列化

**一般数据在网络中进行传输前，都要先在发送方一端对数据进行编码，经过网络传输到达另一端后，再对数据进行解码，这个过程就是系列化和反序列化。**

为什么要对数据进行序列化和反序列化呢？**我们知道网络传输的耗时，一方面取决于网络带宽的大小，另一方面也取决于数据传输量**。要想加快网络传输，要么提高带宽，要么减小数据传输量，而对数据进行编码的主要目的就是减小数据传输量。比如一部高清电影原始大小为30GB，如果经过特殊编码格式处理，可以减小到3GB，同样是100MB/s的网速，下载时间可以从300s减小到30s。

**常用的序列化方式分为两种：文本类（如XML/JSON等），二进制类（如PB/Thrift等）**，具体采用哪种序列化方式，主要取决于三个方面的因素：

⦁ **支持数据结构类型的丰富度：**

数据结构种类支持的越多越好，这样的话对于使用者来说在编程时更加友好，有些序列化框架如Hessian2.0还支持复杂的数据结构，如Map、List等。

⦁ **跨语言支持：**

序列化方式是否支持跨语言也是一个很重要的因素，否则使用的场景就比较局限，比如Java序列化只支持Java语言，就不能用于跨语言的服务调用了。

⦁ **性能：**

**主要看两点，序列化后的压缩比、序列化的速度。**

以常用的PB序列化和JSON序列化协议为例来对比分析，PB序列化的压缩比和速度都要比JSON序列化高很多，所以对性能和存储空间要求比较高的系统选择PB序列化更合适；而JSON序列化虽然性能要差一些，但可读性更好，更适合对外部提供服务。

## 6.5、总结

本节需要掌握内容：

⦁ **通信框架：**

它主要解决客户端和服务端如何建立连接、管理连接以及服务端如何处理请求的问题。

⦁ **通信协议：**

它主要解决客户端和服务端采用哪种数据传输协议的问题。

⦁ **序列化和反序列化：**

它主要解决客户端和服务端采用哪种数据编解码的问题。

**这三个部分就组成了一个完整的RPC调用框架，通信框架提供了基础的通信能力，通信协议描述了通信契约，而序列化和反序列化则用于数据的编/解码。**

一个同信框架可以适配多种通信协议，也可以采用多种序列化和反序列化的格式，比如服务化框架Dubbo不仅支持Dubbo协议，还支持RMI协议、HTTP协议等，而且还支持多种序列化和反序列化格式，比如JSON、Hession2.0以及Java序列化等。

## 6.5、思考题

gRPC是一个优秀的跨语言RPC调用框架，按照今天讲解的服务调用知识，通过查阅官方文档（https://grpc.io/docs/），给出gRPC调用的实现原理

## 6.6、附

⦁ **IO多路复用（IO Multiplexing）**:

即经典的Reactor设计模式，有时也称为异步阻塞IO，Java中的Selector和Linux中的epoll都是这种模型。高性能并发服务程序使用IO多路复用模型+多线程任务处理的架构。

**老师回答：IO多路复用属于同步非阻塞。**

⦁ **NIO（New I/O）:**

同时支持阻塞和非阻塞模式，但是主要使用同步非阻塞IO。

⦁ **异步IO（Asynchronous IO）**:

即经典的Proactor设计模式，也称为异步非阻塞IO。

# 如何监控微服务调用

与单体应用相比，在微服务架构下，一次用户调用会因为服务化拆分后，变成多个不同服务之间的相互调用，这也就需要对拆分后的每个服务都监控起来。

在讲述如何监控微服务调用前，首先需要搞清楚三个问题：监控的对象是什么？具体监控哪些指标？从哪些维度进行监控？下面就从这三个方面，来看看如何监控微服务调用。

## 7.1、监控对象

根据作者经验，对于微服务来说，监控对象可以分为四个层次，由上到下可以归纳为：

**⦁ 用户端监控**：通常是指业务直接对用户提供的功能的监控。（以微博首页Feed为例，它向用户提供了聚合关注的所有人的微博并按照时间顺序浏览的功能，对首页Feed功能的监控就属于用户端的监控）

**⦁ 接口监控：**通常是指业务提供的功能所依赖的具体RPC接口的监控。（以微博首页Feed为例，这个功能依赖于用户关注了哪些人的关系服务，每个人发过哪些微博的微博列表服务，以及每条微博具体内容是什么的内容服务，对这几个服务的调用情况的监控就属于接口监控）

**⦁ 资源监控：**通常是指某个接口依赖的资源的监控。**（比如用户关注了哪些人的关系服务使用的是Redis来存储关系列表，对Redis的监控就属于资源监控）**

**⦁ 基础监控：**通常是指对服务器本身的健康状况的监控（主要包括**CPU利用率、内存使用量、I/O读写量、网卡带宽**等）对服务器的基本监控也是必不可少的，因为服务器本 身的健康状况也是影响服务本身的一个重要因素，比如服务器本身连接的网络交换机上 联带宽被打满，会影响所有部署在这台服务器上的业务。

## 7.2、监控指标

搞清楚监控对象之后，需要监控具体哪些指标呢？根据作者经验，通常有以下几个业务指标需要重点监控：

**⦁ 请求量：**请求量监控分为两个维度，一个**实时请求量**，一个**统计请求量**。

**实时请求量**：用QPS(Query Per Second)—每秒查询次数 来衡量，它反映了服务调用的实时变化情况。

**统计请求量：**一般用PV(Page View)—一段时间内用户的访问量来衡量，比如一天的PV代表了服务一天的请求量，通常用来统计报表。

**⦁ 响应时间：**大多数情况下，可以用一段时间内所有调用的平均耗时来反映请求的响应时间。但它只代表了请求的平均快慢情况。有时我们更关心慢请求的数量。为此需要把响应时间划分为多个区间，比如**0-10ms、10-50ms、50-100ms、100-500ms、500ms以上这五个区间**，其中500ms以上这个区间内的请求数就代表了慢请求量，正常情况下，这个区间内的请求数应该趋近与0。在出现问题时，这个区间内的请求数会大幅增加，可能平均耗时并不能反映出这一变化。除此之外，还可以从**P90、P95、P99、P999**角度来监控请求的响应时间，比如P99=500ms，意思是99%的请求响应时间在500ms以内，它代表了请求的服务质量，即SLA

**⦁ 错误率：**错误率的监控通常用一段时间内调用失败的次数占调用总次数的比率来衡量，比如对于接口的错误率一般用接口返回错误码为503的比率来表示。

## 7.3、监控维度

一般来说，要从多个维度来对业务进行监控，具体来讲可以包括下面几个维度：

**⦁ 全局维度：**

从整体角度监控对象的请求量、平均耗时以及错误率，全局维度的监控一般是为了让你对监控对象的调用情况有个整体了解。

**⦁ 分机房维度：**

一般为了业务的高可用性，服务通常部署在不止一个机房，因为不同机房地域的不同，同一个监控对象的各种指标可能会相差很大，所以需要深入到机房内部去了解。

**⦁ 单机维度：**

即使是在同一个机房内部，可能由于采购年份，批次的不同，位于不同机器上的同一个监控对象的各种指标也会有很大差异。一般来说，新采购的机器通常由于成本更低，性能更高，在同等请求量的情况下，可能表现出较大的性能差异，因此也需要从单机维度去监控同一个对象。

**⦁ 时间维度：**

同一个监控对象，在每天的同一时刻各种指标通常也不会一样，这种差异要么是由业务变更导致，要么是运营活动导致。为了了解监控对象各种指标的变化，通常需要与一天前，一周前、一个月前，甚至三个月前做对比。

**⦁ 核心维度：**

根据作者经验，业务上一般会依据重要性程度对监控对象进行分级，最简单的是**分成核心业务和非核心业务**。**核心业务和非核心业务在部署上必须隔离，分开监控，这样才能对核心业务做重点保障。**

下面讲解如何搭建一个监控系统，来完成上面这些监控功能。

监控系统原理