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]

1、CAP理论的实际应用：参考第13章-两种注册中心的CAP实现区别

2、ProtoBuf简介：14.4.1

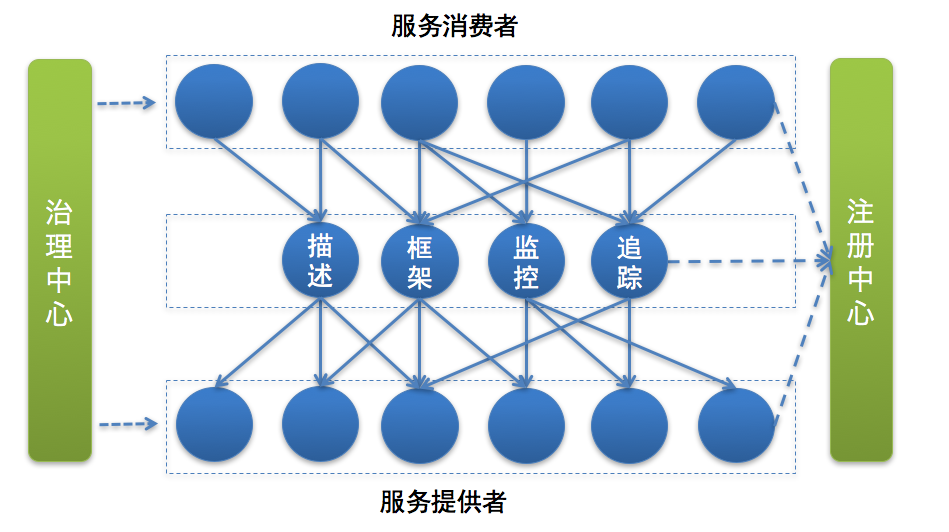
3、Low-Level Transport层的类型（Memory、File、Pipe、TCP/IP、TSL、Unix Domain Socket）—14.4.2 Thrift

4、采用Thrift协议进行编码（序列化）UDP/TCP进行传输 --16.2 Pinpoint调用链

5、负载均衡算法 --18（随机、轮询、加权轮询、最少活跃、一致性Hash算法）

# 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：

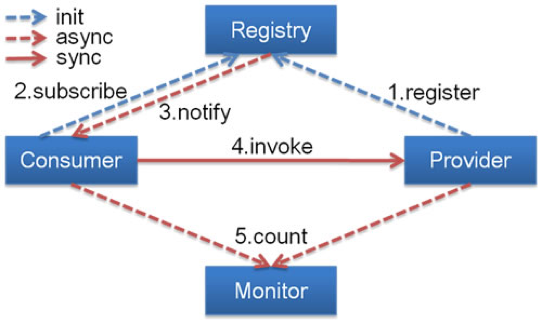
（1）、服务提供者启动时，根据**服务发布文件**中配置的发布信息向注册中心注册自己的服务。

（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

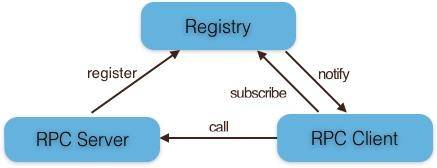
上一讲，我们已经介绍了服务发布和引用常用的三种方式：RESTful API、XML配置和IDL文件。假设你已经使用其中一种方式发布了一个服务，并且已经在一台机器上部署了服务，那**如果我想调用这个服务，我该如何知道你部署的这台机器的地址呢？**

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：

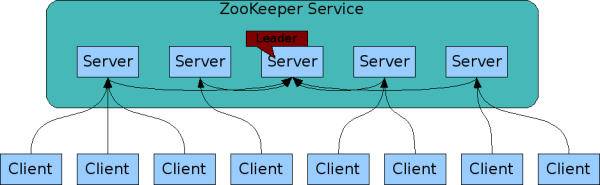
Tip1:每个Server在内存中存储了一份数据，Client的读请求可以请求任意一个Server；

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。

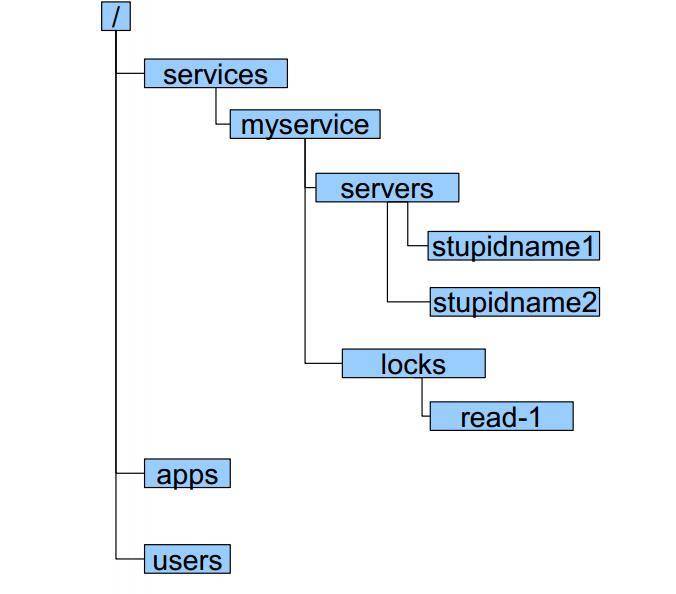


### 5.2.3、目录存储

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？

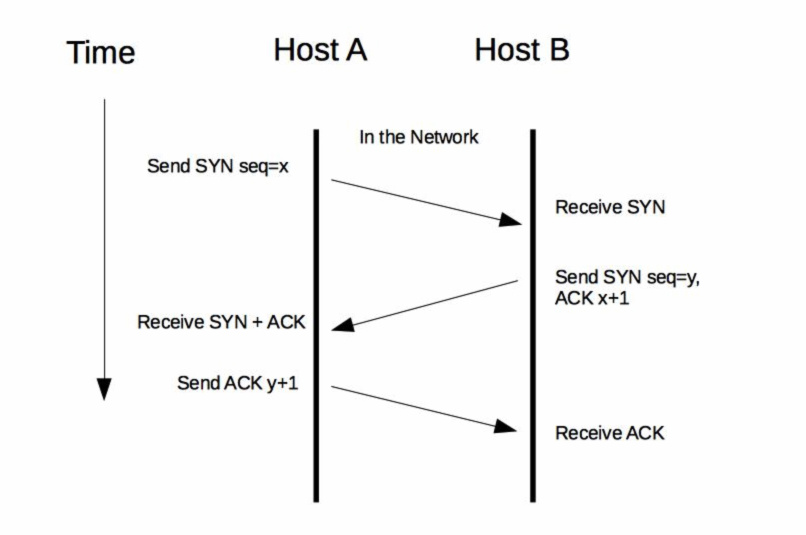
（4）、数据该如何序列化和反序列化？

## 6.1、客户端和服务端建立网络连接

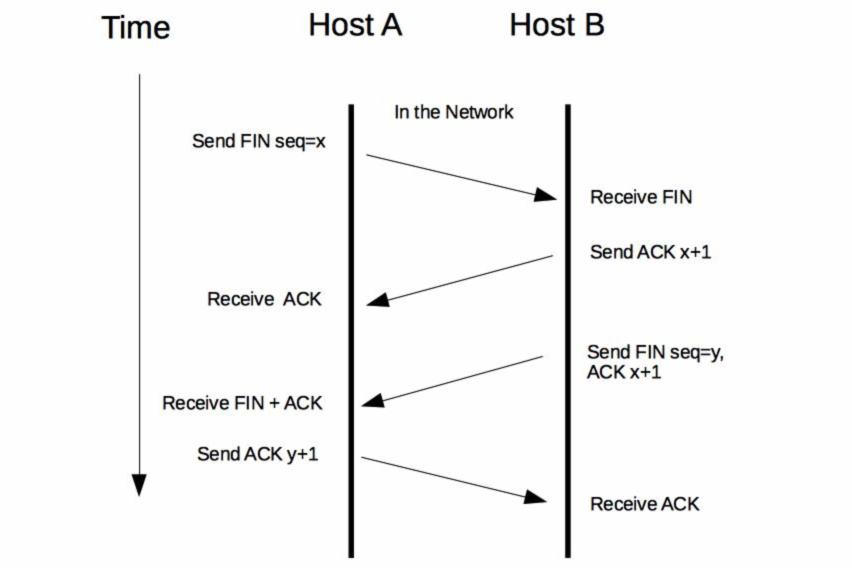
根据作者的实践经验，客户端和服务端之间**基于TCP协议**建立网络连接最常用的途径有两种。

### 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。

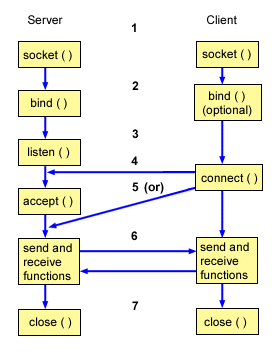


完成请求后，在经历一次“四次挥手”的过程来断开连接。



### 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。

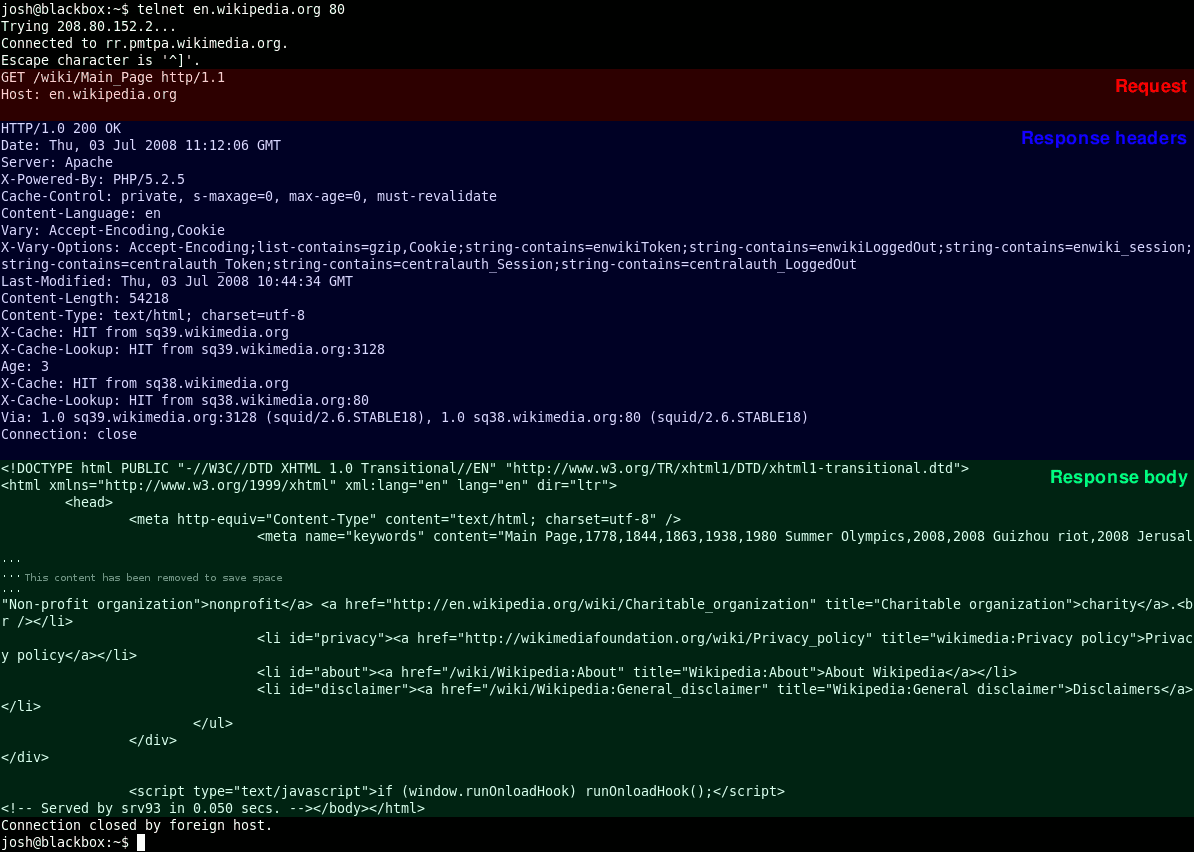
假如客户端和服务端的连接已经建立了，服务端也能正确地处理请求了，接下来完成一次正常的RPC调用还需要解决两个问题，即**数据传输采用什么协议**以及**数据该如何序列化和反序列化**。

## 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。

## 7.4、监控系统原理

显然，我们要对服务调用进行监控，首先要能收集到每一次调用的详细信息，包括调用的响应时间、调用是否成功、调用的发起者和接受者分别是谁，这个过程叫做调用采集。采集到数据之后，要把数据通过一定的方式传输给数据处理中心进行处理，这个过程叫做数据传输。数据传输过来后，数据处理中心再按照服务的维度进行聚合，计算出不同服务的请求量、响应时间以及错误率等信息并存储起来，这个过程叫做数据处理。最后再通过接口或者Dashboard的形式对外展示服务的调用情况，这个过程叫做数据展示。

可见，监控系统主要包括四个环节：数据采集、数据传输、数据处理和数据展示，下面讲解下每个环节的实现原理：

### 7.4.1、数据采集

通常有两种数据收集方式：

**⦁ 服务主动上报：**这种方式通过在业务代码或者服务框架里加入数据收集代码逻辑，在每一次服务调用完成后，主动上报服务的调用信息。

**⦁ 代理收集：**这种方式通过服务调用后把调用的详细信息记录到本地日志文件中，然后通过代理去解析本地日志文件，然后再上报服务的调用信息。

无论哪种数据采集方式，首先要考虑的问题就是采样率，也就是采集数据的频率。采样率决定了监控的实时性与精确度，一般来说，采样率越高，监控的实时性就越高，精确度也越高。但采样对系统本身的性能也会有一定的影响，尤其是采集后的数据需要写到本地磁盘的时候，过高的采样率会导致系统写入磁盘的I/O过高，进而会影响到正常的服务调用。所以设置合理的采样率是数据采集的关键，最好是可以动态控制采样率，在系统比较空闲的时候加大采样率，追求监控的实时性与精确度；在系统负载比较高的时候减小采样率，追求监控的可用性和系统的稳定性。

### 7.4.2、数据传输

数据参数的最常用方式有两种：

**⦁ UDP传输：**这种处理方式是数据处理单元提供服务器的请求地址，数据采集后通过UDP协议与服务器建立连接，然后把数据发送过去。

**⦁ Kafka传输**：这种处理方式是数据采集后发送到指定的Topic，然后数据处理单元订阅对应的Topic，就可以从Kafka消息队列中读取到对应的数据。

无论采用哪种传输方式，数据格式都十分重要，尤其是对带宽敏感以及解析性能要求比较高的场景，一般数据传输时采用的数据格式有两种：

**⦁ 二进制协议：**最常用的就是PB对象，他的优点是高压缩比和高性能，可以减少传输带宽并且序列化和反序列化效率特别高。

**⦁ 文本协议：**最常用的就是JSON字符串，它的优点是可读性好，但相比于PB对象，传输占用带宽高，并且解析性能也要差一些。

### 7.4.2、数据处理

数据处理是对收集来的原始数据进行聚合并存储。数据聚合通常有两个维度：

**⦁ 接口维度聚合：**

这个维度是把实时收到的数据按照接口名维度实时聚合在一起，这样就可以得到每个接口的实时请求量、平均耗时等信息。

**⦁ 机器维度聚合：**

这个维度是把实时收到的数据按照调用的节点维度聚合在一起，这样就可以从单机维度去查看每个接口的实时请求量、平均耗时等信息。

聚合后的数据需要持久化到数据库中存储，所选用的数据库一般分为两种：

**⦁ 索引数据库：**

比如Elasticsearch，以倒排索引的数据结构存储，需要查询的时候，根据索引来查询。

**⦁ 时序数据库：**

比如OpenTSDB，以时序序列数据的方式存储，查询的时候按照时序如1min、5min等维度来查询。

### 7.4.2、数据展示

数据展示是把处理后的数据以Dashboard的方式展示给用户。数据展示有多种方式，比如曲线图、饼状图、格子图展示等。

曲线图：一般用来监控变化趋势的；

饼状图：一般用来监控占比分布的；

格子图：主要做一些粗粒度的监控；

## 7.5、总结

服务监控在微服务改造过程中的重要性不言而喻，没有强大的监控能力，改造成微服务框架后，就五发掌控各个不同服务的情况，在遇到调用失败时，如果不能快速定位发现系统的问题，对于业务来说就是一场灾难。

# 如何追踪服务调用

在微服务框架下，由于进行了服务拆分，一次请求往往需要涉及多个服务，每个服务可能是由不同的团队开发，使用了不同的语言，还可能部署在不同的机器上，分布在不同的数据中心。

如果有一个系统，可以跟踪记录一次用户请求都发起了哪些调用，经过哪些服务处理，并且记录每一次调用所涉及的服务的详细信息，这时候如果发生调用失败，你就可以通过这个日志快速定位是哪个环节处了问题，这个系统就是今天我要讲解的服务追踪系统。

## 8.1、服务追踪的作用

在介绍追踪原理与实现之前，我们先来看看服务追踪的作用。除了刚才说的能够快速定位请求失败的原因之外，我这里再列出四点，它们可以帮你在微服务改造中解决不少问题。

**⦁ 优化系统瓶颈**：

通过记录调用经过的每一条链路上的耗时，我们能快速定位整个系统的瓶颈点在哪里。比如你访问微博首页发现很慢，肯定是由于某种原因造成的，有可能是运营网络延迟，有可能是网关系统异常，有可能是某个服务异常、还可能是缓存或者数据库异常。通过服务追踪，可以从全视角上去观察，找出整个系统的瓶颈点在哪儿，然后做出针对性的优化。

**⦁ 优化链路调用**：

通过服务追踪可以分析调用所经过的路径，然后评估是否合理。比如一个服务调用下游依赖了多个服务，通过调用链分析，可以评估是否每个依赖都是必要的，是否可以通过业务优化来减少服务依赖。

还有就是，一把业务都会在多个数据中心部署服务，以实现异地容灾，这个时候经常会出现一种情况就是服务A调用了另一个数据中心的服务B，而没有调用同处于一个数据中心的服务B。

根据作者的经验，跨数据中心的调用视距离远近都会有一定的网络延时，像北京和广州这种几千公里距离的网络延时可以达到30ms一样，这对于有些业务是不可以接受的。通过对调用链路进行分析，可以找出跨数据中心的服务调用，从而进行优化，尽量规避这种情况出现。

**⦁ 生成网络拓扑**：

通过服务追踪系统中记录的链路信息，可以生成一张系统的网络调用拓扑图，它可以反映系统都依赖了哪些服务，以及服务之间的调用关系是什么样的，可以一目了然。除此之外，在网络拓扑图上还可以把服务调用的详细信息也标出来，也能起到服务监控的作用。

**⦁ 透明传输数据**：

除了服务追踪，业务上经常有一种需求，期望能把一些用户数据，从调用的开始一直往下传递，以便系统中的各个服务都能获取到这个信息。比如业务想做一些A/B测试，这时候就想通过服务追踪系统，把A/B测试的开关逻辑一直往下传递，经过的每一层服务都能获取到这个开关值，就能够统一进行A/B测试。

## 8.2、服务追踪系统原理

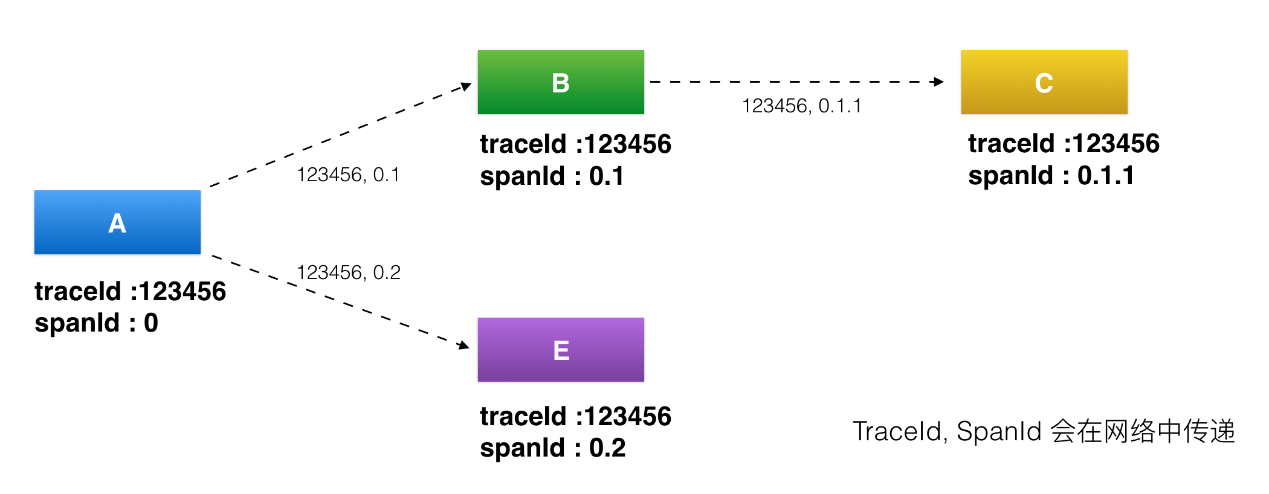
服务追踪系统的鼻祖：Google发布的一篇论文：Dapper,a Large-Scale Distributed Systems Tracing Infrastructure(<http://bigbully.github.io/Dapper-translation/>)，里面详细讲解了服务追踪系统的实现原理。它的核心理念就是调用链：通过一个全局唯一的ID将分布在各个服务节点上的同一次请求串联起来，从而还原原有的调用关系，可以追踪系统问题、分析调用数据并统计各种系统指标。

可以说后面诞生的各种服务追踪系统都是基于Dapper衍生出来的，比较有名的有**Twitter的Zipkin**、**阿里的鹰眼**、**美团的MTrace**等。

要理解服务追踪的原理，首先需要搞懂一些基本概念：traceId、spanId、annotation等。

Dapper这篇论文介绍的比较清楚，但对于初学者来说理解起来可能比较困难，美团的MTrace的原理介绍理解起来相对容易一些，下面我们以MTrace为例，详细讲述服务追踪系统的实现原理。虽然原理有些晦涩，但却是必须掌握的，只有理解了服务追踪的基本概念，才能更好地将其实现出来。

首先看下面这张图，我们来讲解下服务追踪系统中几个最基本的概念:



（图片来源：<https://tech.meituan.com/img/mt-mtrace/mtrace7.png>）

**⦁ traceId**：

用于标识某一次具体的请求ID。

当用户的请求进入系统后，会在RPC调用网络的第一层生成一个全局唯一的traceId，并且会随着每一层的RPC调用，不断往后传递，这样的话通过traceId就可以把一次用户请求在系统中调用的路径串联起来。

**⦁ spanId**：

用于标识一次RPC调用在分布式请求中的位置。

当用户的请求进入系统后，处在RPC调用网络的第一层A时，spanId初始值为0 ，进入下一层RPC调用B的时候，spanId是0.1，继续进入下一层RPC调用C时，spanId是0.1.1，而与B处在同一层的RPC调用E的spanId是0.2，这样的话通过spanId就可以定位某一次RPC请求在系统调用中所处的位置，以及它的上下游依赖分别是谁。

**⦁ annotation**：

用于业务自定义埋点数据，可以是业务感兴趣的想上传到后端的数据。

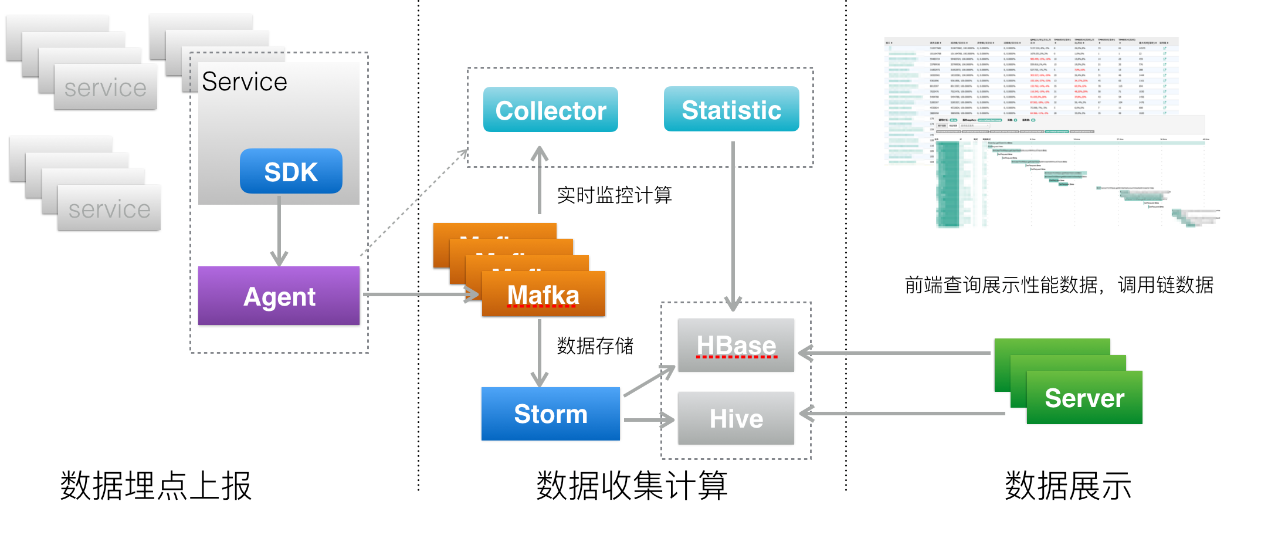
比如一次请求的用户UID。

通俗的讲：

traceId是用于串联某一次请求在系统中经过的所有路径，spanId是用于区分系统不同服务之间调用的先后关系，而annotation是用于业务自定义一些自己感兴趣的数据，在上传traceId和spanId这些基本信息之外，添加一些自己感兴趣的信息。

## 8.3、服务追踪系统实现

我们接下来看看服务追踪系统的架构，了解下服务追踪系统的全貌



（图片来源：<https://tech.meituan.com/img/mt-mtrace/mtrace6.png>）

上面是服务追踪系统的架构图，我们可以看到一个服务追踪系统可以分为三层。

**⦁** 数据采集层：负责数据埋点并上报。

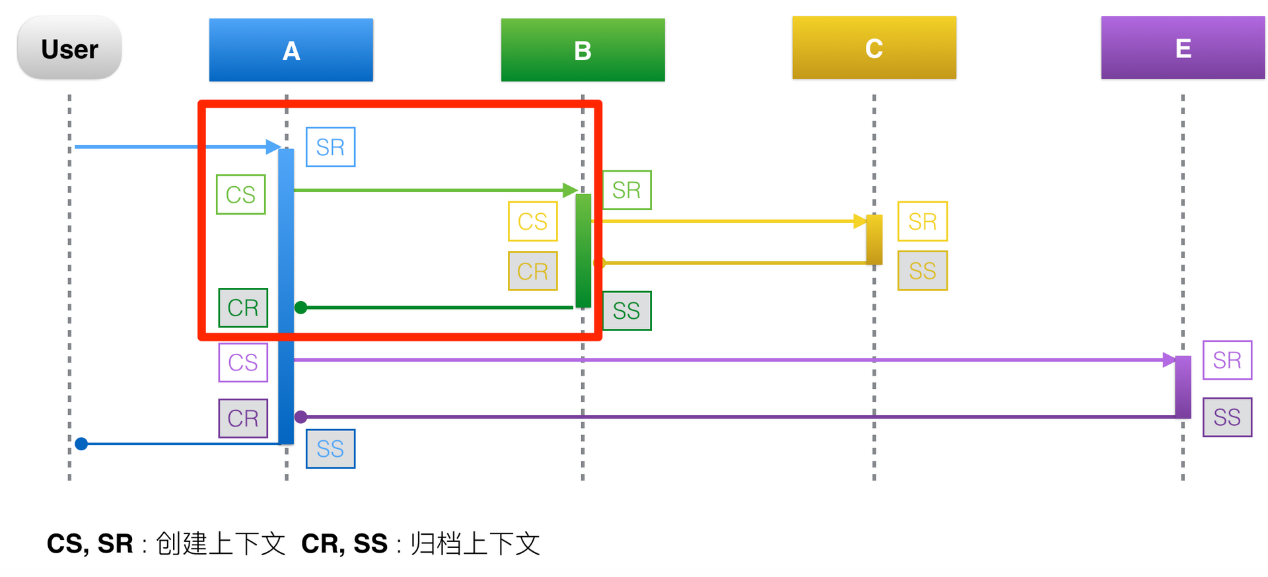
⦁ 数据处理层：服务数据的存储与计算。

⦁ 数据展示层：服务数据的图形化展示。

### 8.3.1数据采集层

数据采集层的作用就是在系统的各个不同模块中进行埋点，采集数据并上报给数据处理层进行处理。

下面，我们结合下面这张图来了解下数据埋点的流程：



（图片来源：<https://tech.meituan.com/img/mt-mtrace/mtrace9.png>）

以红色方框里圈出的A调用B的过程为例，一次RPC请求可以分为四个阶段。

**⦁ CS（Client Send）阶段：**

客户端发起请求，并生成调用的上下文。

**⦁ SR（Server Recieve）阶段：**

服务端接收请求，并生成上下文。

**⦁ SS（Server Send）阶段：**

服务端返回请求，这个阶段会将服务端上下文数据上报。

**CR（Client Recieve）阶段：**

客户端接收返回结果，这个阶段会将客户端上下文数据上报。

### 8.3.2数据处理层

数据处理层的作用就是把数据采集层上报的数据按需处理，然后落地存储供查询使用。

据作者所知，数据处理的需求一般分为两类，一类是实时计算需求，一类是离线计算需求。

实时计算需求对计算效率要求比较高，一般要求对收集的链路数据能够在秒级完成聚合计算，以供实时查询。而离线计算需求对计算效率要求就没那么高了，一般能在小时级别完成链路数据的聚合计算即可，一般用作数据汇总统计。针对这两类不同的数据处理需求，采用的计算方法和存储也不相同。

**⦁ 实时数据处理**

针对实时数据处理，一般采用Storm或者Spark Streaming来对链路数据进行实时聚合加工，存储一般使用OLTP数据仓库，比如HBase，使用traceId作为Rowkey，能天然地把一整条调用链聚合在一起，提高查询效率。

**⦁ 离线数据处理**

针对离线数据处理，一般通过运行MapReduce或者Spark批处理程序来对链路数据进行离线计算，存储一般使用Hive。

### 8.3.3数据展示层

数据展示层的作用就是将处理后的链路信息以图形化的方式展示给用户。

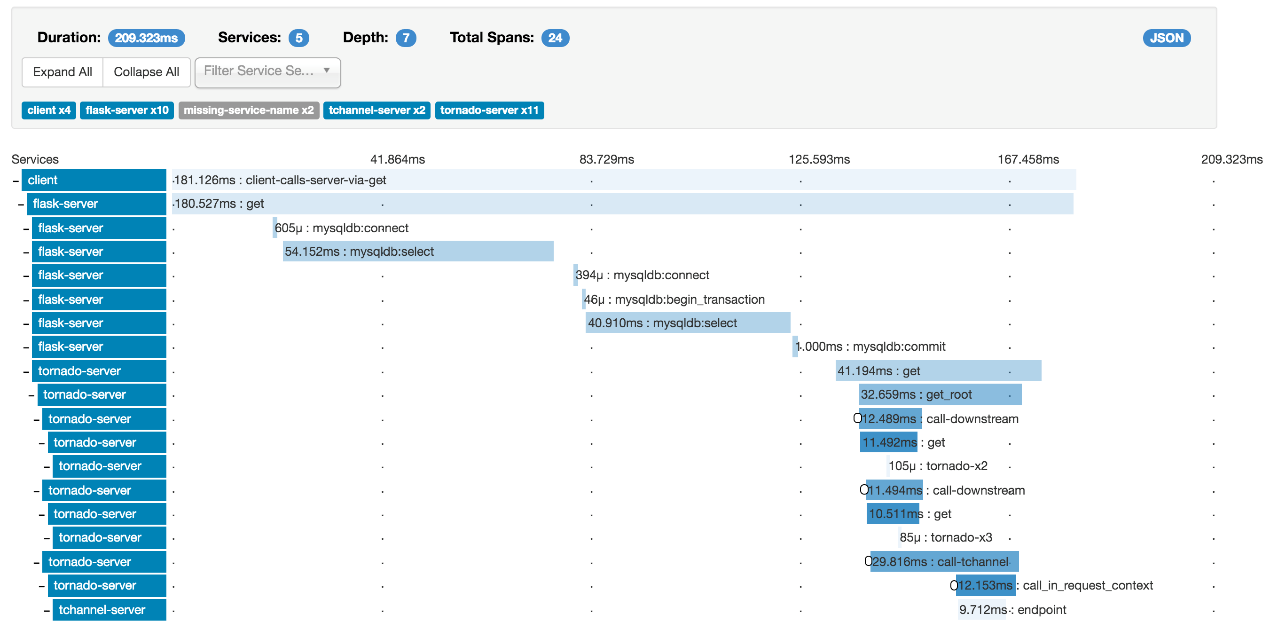
根据作者经验，实际项目中主要用到两种图形展示，一种是调用链路图，一种是调用拓扑图。

**⦁ 调用链路图**

下面以一张Zipkin的调用链路图为例，通过这张图可以看出下面几个信息。

**服务整体情况**：服务总耗时，服务调用的网络深度，每一层经过的系统，以及多少次调用。下图展示的一次调用，总共耗时209ms，经过了5个不同的系统模块，调用深度为7层，共发生了24次系统调用。

**每一层的情况**：每一层发生了几次调用，以及每一层调用的耗时。

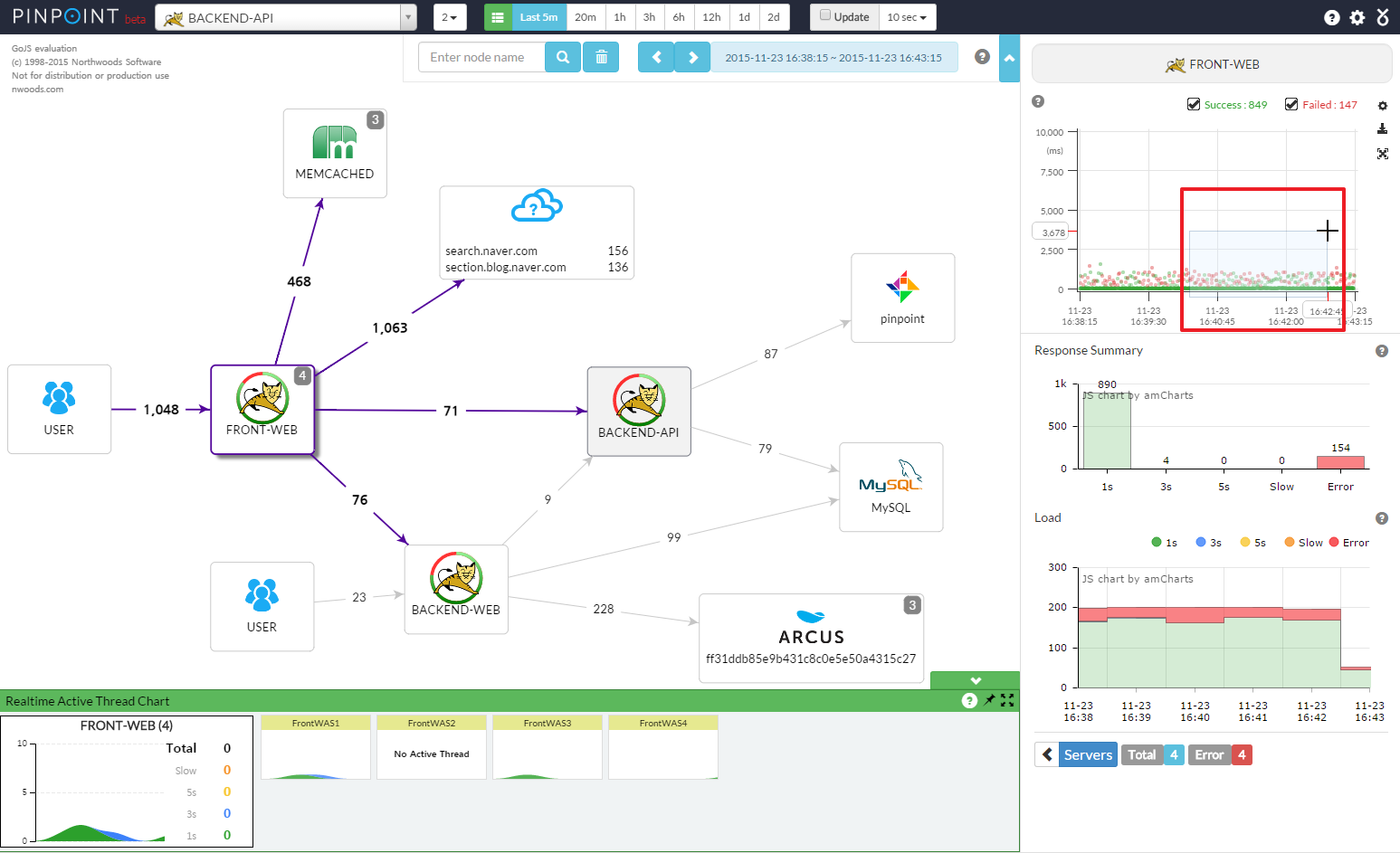
****

（图片来源：https://zipkin.io/public/img/web-screenshot.png）

根据作者经验，调用链路图在实际项目中，主要被用来做故障定位，比如某一次用户调用失败了，可以通过调用链路图查询这次用户调用经过了哪些环节，到底是那一层的调用失败所导致。

**⦁ 调用拓扑图**

下面是一张PinPoint的调用拓扑图，通过这张图可以看出来系统都包含哪些应用，它们之间是什么关系，以及依赖调用的QPS、平均耗时情况。



（图片来源：<https://time.geekbang.org/column/115>）

调用拓扑图是一种全局视野图，在实际项目中，主要用作全局监控，用于发现系统中异常的点，从而快速做出决策。比如，某一个服务突然出现异常，那么在调用链路拓扑图中可以看出对这个服务的调用耗时都变高了，可以用红色的图样标出来，用作监控报警。

## 8.3、总结

实现一个服务追踪系统，涉及数据采集、数据处理和数据展示这三个流程，有多种实现方式，具体采用哪一种要根据自己的业务情况开选择。

## 8.3、思考题

服务追踪系统和服务监控系统的搭建都需要数据采集、处理和展示这三个步骤，它们的相同和不同之处？

# 服务治理的手段

上一期我们讲过服务追踪的基本原理，有了服务追踪系统，在服务出现问题的时候，我们就可以定位服务哪里出现了问题。一般单体应用改造成微服务架构后，还会增加哪些问题呢？又该如何应对呢？

前面我们讲到单体应用改造成微服务架构后，服务调用由本地调用变成远程调用，服务消费者A需要通过注册中心去查询服务提供者B的地址，然后发起调用，这个看似简单的过程就可能会遇到下面几种情况，比如：

**⦁** 注册中心宕机；

**⦁** 服务提供者B有节点宕机；

**⦁** 服务消费者A和注册中心之间的网络不通；

**⦁** 服务提供者B和注册中心之间的网络不通；

**⦁** 服务消费者A和服务提供者B之间的网络不通；

**⦁** 服务提供者B有些节点性能变慢；

**⦁** 服务提供者B短时间内出现问题；

可见，一次服务调用，**服务提供者、注册中心、网络**这三者都可能会出现问题，此时服务消费者应该如何处理才能确保调用成功呢？这就是服务治理要解决的问题。

接下来，我们一起来看看**常用的服务治理手段**

## 9.1节点管理

根据作者的经验，服务调用失败一般是由两类原因引起的，一类是服务提供者自身出现问题，如服务器宕机、进程意外退出等；一类是网络问题，如服务提供者、注册中心、服务消费者这三者任意两者之间的网络出现问题。

无论是服务提供者自身出现问题还是网络发生问题，都有两种节点管理手段。

### 9.1.1注册中心主动摘除机制

这种机制要求服务提供者定时主动向注册中心汇报心跳，注册中心根据服务提供者节点最近一次汇报心跳的时间与上一次汇报心跳时间做对比，如果超出一定时间，就认为服务提供者出现问题，继而把节点从服务列表中摘除，并把最近的可用服务节点列表推送给服务消费者。

### 9.1.1服务消费者摘除机制

虽然注册中心主动摘除机制可以解决服务提供者节点异常的问题，但如果是因为注册中心与服务提供者之间的网络出现异常，最坏的情况是注册中心会把服务节点全部摘除，导致服务消费者没有可用的服务节点调用，但其实这时候服务提供者本身是正常的。所以，将**存活探测机制**用在服务消费者这一端更合理，如果服务消费者调用服务提供者节点失败，就将这个节点从内存中保存的可用服务提供者节点列表中移出。

## 9.2负载均衡

一般情况下，服务提供者节点不是唯一的，多是以集群的方式存在，尤其是对于大规模的服务调用来说，服务提供者节点数目可能有成百上千个。由于机器采购批次不同，不同服务节点本身的配置也可能存在很大的差异，新采购的机器CPU和内存配置可能要高一些，同等请求量情况下，性能要好于旧的机器。对于服务消费者而言，在从服务列表中选取可用节点时，如果能让配置较高的新机器多承担一些流量的话，就能充分利用新机器的性能。这就需要对负载均衡算法做一些调整。

常用的负载均衡算法主要包括下面几种：

### 9.2.1随机算法

顾名思义就是从可用的服务节点中随机选取一个节点。一般情况下，随机算法是均匀的，也就是说后端服务节点无论配置好坏，最终得到的调用量都差不多。

### 9.2.2轮询算法

就是按照固定的权重，对可用服务节点进行轮询。如果所有服务节点的权重都是相同的，则每个节点的调用量也是差不多的。但是可以给某些硬件配置较好的节点的调用权重调大些，这样的话就会得到更大的调用量，从而充分发挥其性能优势，提高整体调用的平均性能。

### 9.2.3最少活跃调用算法

这种算法是在服务消费者这一端的内存里动态维护着同每一个服务节点之间的连接数，当调用某个服务节点时，就给与这个节点之间的连接数加1，调用返回后，就给连接数减1。然后每次选择服务节点时，根据内存里维护的连接数倒序排列，选择连接数最小的节点发起调用，也就是选择了调用量最小的服务节点，性能理论上也是最优的。

### 9.2.4一致Hash算法

指相同参数的请求总是发送到同一服务节点。当某一个服务节点出现故障时，原本发往该节点的请求，基于虚拟节点机制，平摊到其他节点上，不会引起剧烈变动。

这几种算法的实现难度也是逐步提升的，所以选择哪种节点选取的负载均衡算法要根据实际场景而定。如果后端服务节点的配置没有差异，同等调用量下性能也没有差异的话，选择随机或者轮询算法比较合适；如果后端服务节点存在比较明显的配置和性能差异，选择最少活跃调用算法比较合适。

## 9.3服务路由

**对于服务消费者而言，在内存中的可用服务节点列表中选择哪个节点不仅由负载均衡算法决定，还由路由规则确定。**

**所谓路由规则，就是通过一定的规则，如条件表达式或者正则表达式来限定服务节点的选择范围。**

为什么要制定路由规则呢？主要有两个原因。

1. **业务存在灰度发布的需求**

比如，服务提供者做了功能变更，但希望先只让部分人群使用，然后根据这部分人群的使用反馈，再来决定是否做全量发布。这个时候，就可以通过类似按尾号进行灰度的规则限定只有一定比例的人群才会访问新发布的服务节点。

1. **多机房就近访问的需求**

据作者所知，大部分业务规模中等及以上的互联网公司，为了业务的高可用性，都会将自己的业务部署在不止一个IDC中。这个时候就存在一个问题，不同IDC之间的访问由于要跨IDC，通过专线访问，尤其是IDC相距比较远时，延迟就会比较大，比如北京和广州的专线延迟一般在30ms左右，这对于某些延迟敏感性的业务是不可接受的，所以就要一次服务调用尽量选择同一个IDC内部的节点，从而减少网络耗时开销，提高性能。这时一般可以通过IP段规则来控制访问，在选择服务节点时，优先选择同一IP段的节点。

那么，路由规则该如何配置呢？根据作者的实际项目经验，一般有两种配置方式。

1. **静态配置**

就是在服务消费者本地存放服务调用的路由规则，在服务调用期间，路由规则不会发生改变，要想改变就需要修改服务消费者本地配置，上线后才能生效。

1. **动态配置**

这种方式下，路由规则是存在注册中心的，服务消费者定期去请求注册中心来保持同步，要想改变服务消费者的路由配置，可以通过修改注册中心的配置，服务消费者在下一个同步周期之后，就会请求注册中心来更新配置，从而实现动态更新。

## 9.4服务容错

服务调用并不总是一定成功的，可能因为服务提供者节点自身宕机、进程异常退出或者服务消费者与提供者之间的网络出现故障等原因。对于服务调用失败的情况，需要有手段自动恢复，来保证调用成功。

常用的手段主要有如下几种：

### 9.4.1 FailOver-失败自动切换

就是服务消费者发现调用失败或者超时后，自动从可用的服务节点列表中选择下一个节点重新发起调用，也可以设置重试的次数。这种策略要求服务调用的操作必须是**幂等**的，也就是说**无论调用多少次，只要是同一个调用，返回的结果都是相同的。一般适合服务调用是读请求的场景。**

### 9.4.2 FailBack-失败通知

就是服务消费者调用失败或者超时后，不再重试，而是根据失败的详细信息，来决定后续的执行策略。比如对于非幂等的调用场景，如果调用失败后，不能简单地重试，而是应该查询服务端的状态，看调用到底是否实际生效，如果已经生效了就不能再重试了；如果没有生效可以再发起一次调用。

### 9.4.2 FailCache-失败缓存

就是服务消费者调用失败或者超时后，不立即发起重试，而是隔一段时间后再次尝试发起调用。比如后端服务可能一段时间内都有问题，如果立即发起重试，可能会加剧问题，反而不利于后端服务的恢复。如果隔一段时间待后端节点恢复后，再次发起调用效果会更好。

### 9.4.2 FailFast-快速失败

就是服务消费者调用一次失败后，不再重试。实际在业务执行时，一般非核心业务的调用，会采用快速失败策略，调用失败后一般就记录下失败日志就返回了。

上面四中服务容错策略的使用场景大致描述如下，对于幂等的调用，可以选择FailOver或者FailCache，非幂等的调用可以选择FailBack或者FailFast。

## 9.5总结

上面我们讲到的服务治理的手段是最常用的手段，它们从不同的角度来确保服务调用的成功率。节点管理是从**服务节点健康状态角度**来考虑，负载均衡和服务路由是从**服务节点访问优先级角度**来考虑，而服务容错是从**调用的健康状态角度**来考虑，可谓殊途同归。

在实际的微服务家规实践中，上面这些服务治理手段一般都会在服务框架中默认集成了，比如阿里的Dubbo、微博开源的服务框架Motan等，不需要业务代码去实现。如果想自己实现服务治理的手段，可以参考这些开源服务框架的实现。

# Dubbo框架里的微服务组件

通过前面几期的讲解，大家应该对微服务的框架有了初步的了解。简单回顾下，微服务的框架主要包括：**服务描述、服务发现、服务调用、服务监控、服务追踪**以及**服务治理**这几个基本组件。

那么每个基本组件从架构和代码设计该如何实现呢？组件之间又是如何串联来实现一个完整的微服务架构呢？今天我们就以开源的微服务框架Dubbo为例来讲解这些组件的实现。

## 10.1 Dubbo的服务发布与引用

前面我们讲解过服务发布与引用的三种常用方式：RESTtul API、XML配置以及IDL文件，其中Dubbo框架主要是使用XML配置方式。

## 10.2 Dubbo的服务注册与发现

## 10.3 Dubbo的服务调用

前面我们讲过在服务调用的过程中，通常把服务消费者叫做客户端，服务提供者叫做服务端，发起一次服务调用需要解决四个问题:

**⦁** 客户端和服务端如何建立网络连接？

**⦁** 服务端如何处理请求？

**⦁** 数据传输采用什么协议？

**⦁** 数据该如何序列化和反序列化？

其中前两个问题客户端和服务端如何建立连接和服务端如何处理请求是通信框架要解决的问题.

**⦁** Dubbo的通信框架：

Dubbo支持多种通信框架，比如Netty4，需要在服务端和客户端的XML配置中添加下面的配置项。

服务端：<dubbo:protocol server="netty4" />

客户端：<dubbo:consumer client="netty4" />

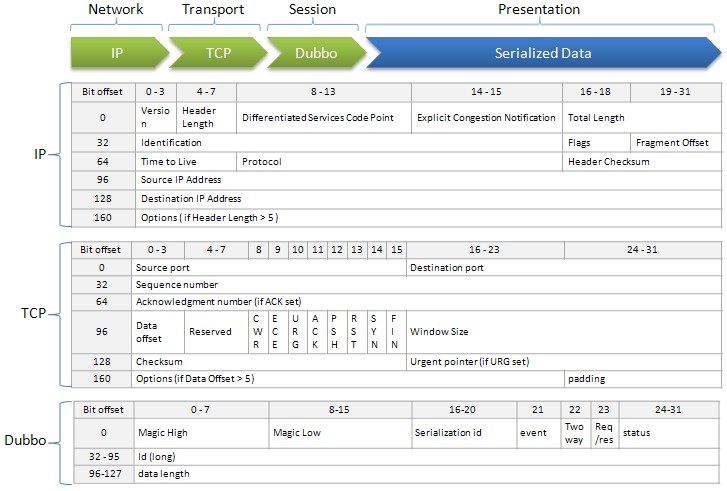
这样基于扩展点自适应机制，客户端和服务端之间的调用会通过Netty 4框架来建立连接

**⦁** Dubbo处理客户端请求的方式：

配置项如上通信框架中的XML配置，配置完后会采用NIO方式处理客户端请求

**⦁** Dubbo的数据传输采用的协议有：

不仅支持私有的Dubbo协议、还支持其他协议，比如Hession、RMI、HTTP、Web Service、Thrift等。下面这张图描述了私有Dubbo协议的协议头约定。



（图片来源：<https://dubbo.incubator.apache.org/docs/zh-cn/dev/sources/images/dubbo_protocol_header.jpg>）

**⦁** Dubbo采用的数据序列化和反序列化方式：

Dubbo支持多种序列化格式，比如：Dubbo、Hession2.0、JSON、Java、Kryo以及FST等

，可以通过在XML配置中添加下面的配置项，例如：

<dubbo:protocol name="dubbo" serialization="kryo"/>

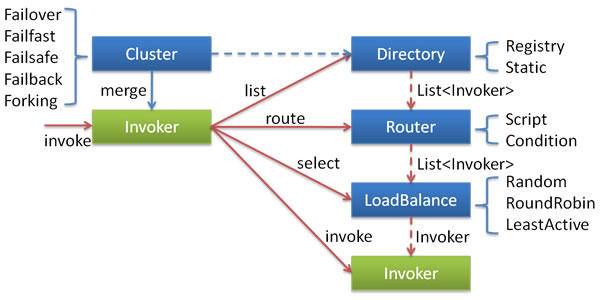
## 10.4 Dubbo的服务监控

服务监控主要包括四个流程：数据采集、数据传输、数据处理和数据展示，其中服务框架的做用是进行埋点数据采集，然后上报给监控系统。

在Dubbo框架中，无论是服务提供者还是服务消费者，在执行服务调用的时候，都会经过Filter调用链拦截，来完成一些特定功能，比如监视数据埋点就是通过在Filter调用链上装备了MonitorFilter来实现的，详细的代码可以参考：https://github.com/apache/incubator-dubbo/blob/7a48fac84b14ac6a21c1bdfc5958705dd8dda84d/dubbo-monitor/dubbo-monitor-api/src/main/java/org/apache/dubbo/monitor/support/MonitorFilter.java

## 10.5 Dubbo的服务治理

服务治理手段包括节点管理、负载均衡、服务路由、服务容错等，下面这张图给出了Dubbo框架服务治理的具体实现



（图片来源：<https://time.geekbang.org/column/article/31644>）

**图中的Invoker是对服务提供者节点的抽象，Invoker封装了服务提供者的地址以及接口信息。**

**⦁** 节点管理：

Directory负责从注册中心获取服务节点列表，并封装成多个Invoker，可以把它看成“List<Invoker>”，它的值可能是动态变化的，比如注册中心推送变更时需要更新。

**⦁** 负载均衡：

LoadBalance负责从多个Invoker中选出某一个用于发起调用，选择时可以采用多种负载均衡算法，比如Random（随机算法）、RoundRobin（轮询算法）、LeastActive（最少活跃调用）等。

**⦁** 服务路由：

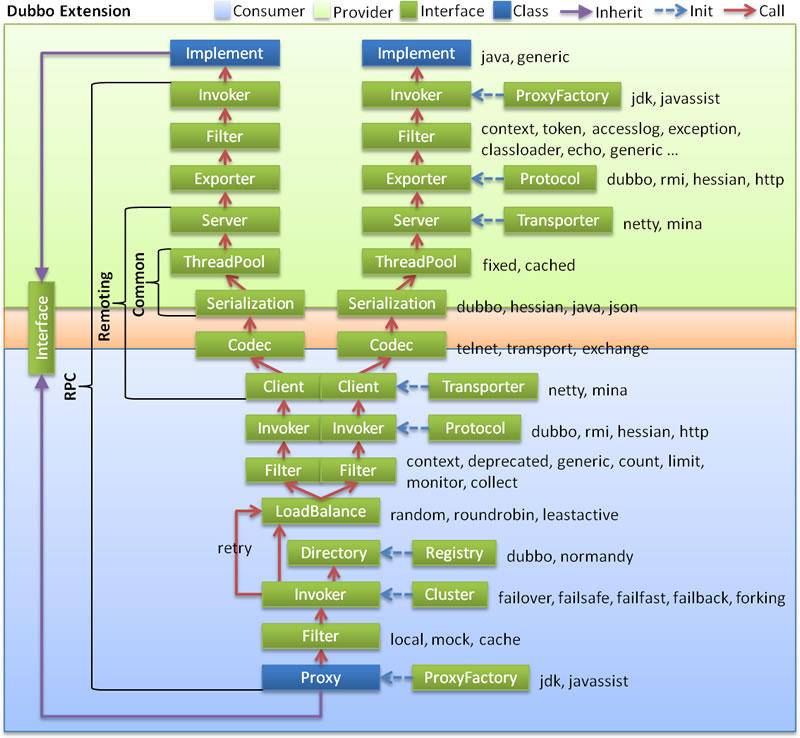
Router负责从多个Invoker中按路由规则选出子集，比如读写分离、机房隔离等。

**⦁** 服务容错：

Cluster将Directory中的多个Invoker伪装成一个Invoker，对上层透明，伪装过程包含了容错逻辑，比如采用Failover策略的话，调用失败后，会选择另一个Invoker，重试请求。

## 10.6 一次服务调用的流程

上面我们讲的是Dubbo下每个基本组件的实现方式，那么Dubbo框架下，一次服务调用的流程是什么样的呢？下面我们结合这张图，来详细讲解一下。



图片来源：

**首先**我来解释微服务架构中各个组件分别对应到上面这张图中是如何实现。

**⦁** 服务发布和引用：

对应实现是图里的**Proxy服务代理层**，Proxy根据客户端和服务端的接口描述，生成接口对应的客户端和服务端的Stub，使得客户端调用服务端就像本地调用一样。

**⦁** 服务注册与发现：

对应实现是图里的**Registry注册中心层**，Registry根据客户端和服务端的接口描述，解析成服务的URL格式，然后调用注册中心的API，完成服务的注册和发现。

**⦁** 服务调用：

对应实现是Protocol远程调用层，Protocol把客户端的本地请求转换成RPC请求。然后通过Transporter层来实现通信，Codec层来实现协议封装，Serialization层来实现数据序列化和反序列化。

**⦁** 服务监控：

对应实现层是Filter调用链层，通过在Filter调用链层中加入MonitorFilter，实现对每一次调用的拦截，在调用前后进行埋点数据采集，上传给监控系统。

**⦁** 服务治理：

对应实现层是Cluster层，负责服务节点管理、负载均衡、服务路由以及服务容错。

**再来**看下微服务架构各个组件是如何串联起来组成一个完整的微服务框架的，以Dubbo框架下一次服务调用的过程为例，先来看下**客户端发起调用的过程**。

**⦁** 首先根据接口定义，通过Proxy层封装好的透明化接口代理，发起调用。

**⦁** 然后再通过Registry层封装好的服务发现功能，获取所有可用的服务提供者节点列表。

**⦁** 再根据Cluster层的负载均衡算法从可用的服务节点列表中选取一个节点发起服务调用，如果调用失败，根据Cluster层提供的服务容错手段进行处理。

**⦁** 同时通过Filter层拦截调用，实现客户端的监控统计。

**⦁** 最后在Protocol层，封装成Dubbo RPC请求，发给服务端节点。

这样的话，客户端的请求就从一个本地调用转化成一个远程RPC调用，经过服务调用框架处理，通过网络传输到达服务端。其中服务调用框架包括通信协框架Transporter？？？、通信协议Codec、序列化Serialization三层处理。

**服务端从网络中接收到请求后的处理过程是这样的：**

**⦁** 首先在Protocol层，把网络上的请求解析成Dubbo RPC请求。

**⦁** 然后通过Filter拦截调用，实现服务端的监控统计。

**⦁** 最后通过Proxy层的处理，把Dubbo RPC请求转化为接口的具体实现，执行调用。

## 10.7 总结

今天我们讲述了Dubbo服务化框架每个基本组件的实现方式，以及一次Dubbo调用流程。

对于学习微服务架构来说，最好的方式是去实际搭建一个微服务的框架。甚至从代码入手做一些二次开发。

你可以按照Dubbo的官方文档（http://dubbo.incubator.apache.org/en-us/）去安装并搭建一个服务化框架。如果想深入了解它的实现的话，可以下载源码（https://github.com/apache/incubator-dubbo）来阅读。

# 服务发布和引用的实践

在专栏第四期，我们讲过服务发布和引用常见的三种方式：RESTful API、XML配置以及IDL文件。今天我将以XML配置为例，给你讲解服务发布和引用的具体实践以及可能会遇到的问题。

首先我们一起来看下XML配置方式，服务发布和引用的具体流程是什么样的。

## 11.1 XML配置方式的服务发布和引用流程

### 11.1.1服务提供者定义接口

服务提供者发布服务之前首先要定义接口、申明接口名、传递参数以及返回值类型，然后把接口打包成JAR包发布出去。

比如下面这段代码，申明了接口UserLastStatusService，包含两个方法getLastStatusId和getLastStatusIds，传递参数一个是long值、一个是long数组，返回值一个是long值、一个是map。

package com.weibo.api.common.status.service;

public interface UserLastStatusService {

\* @param uids

\* @return

\*/

public long getLastStatusId(long uid);

/\*\*

\*

\* @param uids

\* @return

\*/

public Map<Long, Long> getLastStatusIds(long[] uids);

}

### 11.1.2服务提供者发布接口

服务提供者发布的接口是通过在服务发布配置文件中定义接口来实现的。

下面我以一个具体的服务发布配置文件user-last-status.xml来给你讲解，它定义了要发布的接口：userLastStatusLocalService，对外暴露的协议是Motan协议，端口是8882。并且针对两个方法getLastStatusId和getLastStatusIds，通过requestTImeout=”300”单独定义了超时时间是300ms，通过retries=”0”单独定义了调用失败后重试次数为0，也就是不重试。

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-2.5.xsd

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd

">

<motan:service ref="userLastStatusLocalService"

requestTimeout="50" retries="2" interface="com.weibo.api.common.status.service.UserLastStatusService"

basicService="serviceBasicConfig" export="motan:8882">

<motan:method name="getLastStatusId" requestTimeout="300"

retries="0" />

<motan:method name="getLastStatusIds" requestTimeout="300"

retries="0" />

</motan:service>

</beans>

然后服务发布者在进程启动的时候，会加载配置文件user-last-status.xml，把接口对外暴露出去。

### 11.1.3服务消费者引用接口

服务消费者引用接口是通过在服务引用配置文件中定义要引用的接口，并把包含接口定义的JAR包引入到代码依赖中。

下面我再以一个具体的服务引用配置文件user-last-status-client.xml来讲解，它定义服务消费者引用接口commonUserLastStatusService，接口通信协议是Motan。

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xmlns:context="http://www.springframework.org/schema/context"

xmlns:aop="http://www.springframework.org/schema/aop"

xsi:schemaLocation="http://www.springframework.org/schema/context

http://www.springframework.org/schema/context/spring-context-2.5.xsd

http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

http://www.springframework.org/schema/aop http://www.springframework.org/schema/aop/spring-aop-2.5.xsd

">

<motan:protocol name="motan" default="true" loadbalance="${service.loadbalance.name}" />

<motan:basicReferer id="userLastStatusServiceClientBasicConfig"

protocol="motan" />

<!-- 导出接口 -->

<motan:referer id="commonUserLastStatusService" interface="com.weibo.api.common.status.service.UserLastStatusService"

basicReferer="userLastStatusServiceClientBasicConfig" />

</beans>

然后服务消费者在进程启动时，会加载配置文件user-last-status-client.xml来完成服务引用。

上面所讲的服务发布和引用流程看似比较简单，但在实际使用过程中，还是有很多坑的，比如在实际项目中经常会遇到这个问题：一个服务包含了多个接口，可能有上行接口也可能有下行接口，每个接口都有超时控制以及是否重试等配置，如果有多个服务消费者引用这个服务，是不是每个服务消费者都必须在服务引用配置文件中定义？

你可以先思考一下这个问题，联系自己的实践经验，是否有理想的解决方案呢？

## 11.2 服务发布和引用的那些坑

根据作者的项目经验，在一个服务被多个服务消费者引用的情况下，由于业务经验的参差不齐，可能不同的服务消费者对服务的认知水平不一，比如某个服务可能调用超时了，最好可以重试来提高调用成功率。但可能有的服务消费者会忽略这一点，并没有在服务引用配置文件中配置接口调用超时重试的次数，因此最好是可以在服务发布的配置文件中预定义好类似超时重试次数，即使服务消费者没有在服务引用配置文件中定义，也能继承服务提供者的定义。这就是下面要讲的服务发布预定义配置。

### 11.2.1服务发布预定义配置

以下面的服务发布配置文件server.xml为例，它提供了一个服务contentSliceRPCService，并且明确了其中三个方法的调用超时时间为500ms以及超时重试次数为3。

<motan:service ref="contentSliceRPCService" interface="cn.sina.api.data.service.ContentSliceRPCService"

basicService="serviceBasicConfig" export="motan:8882" >

<motan:method name="saveContent" requestTimeout="500"

retries="3" />

<motan:method name="deleteContent" requestTimeout="500"

retries="3" />

<motan:method name="updateContent" requestTimeout="500"

retries="3" />

</motan:service>

假设服务引用的配置文件client.xml的内容如下，那么服务消费者就会默认继承服务发布配置文件中设置的方法调用的超时时间以及超时重传次数。

<motan:referer id="contentSliceRPCService" interface="cn.sina.api.data.service.ContentSliceRPCService" basicReferer="contentSliceClientBasicConfig" >

</motan:referer>

通过服务发布预定义配置可以解决多个服务消费者引用服务可能带来的配置复杂的问题，这样是不是最优的解决方案呢？

**实际上，作者还遇到过另外一种极端情况**，一个服务提供者发布的服务有上百个方法，并且每个方法都有各自的超时时间、重试次数等信息。服务消费者引用服务时，完全继承了服务发布预定义的各项配置。这种情况下，服务提供者所发布服务的详细配置信息都需要存储在注册中心中，这样服务消费者才能在实际引用时从服务发布预定义配置中继承各种配置。

这样就存在一种风险，当服务提供者发生节点变更，尤其是在网络抖动的情况下，所有的服务消费者都会从注册中心拉取最新的服务节点信息，就包括了服务发布配置中预定的各项接口信息，这个信息不加限制的话可以达到1M以上，如果同时有上百个服务消费者从注册中心拉取服务节点信息，在注册中心机器部署为百兆带宽的情况下，很有可能会导致网络带宽打满的情况发生。

面对这种情况，最好的办法就是把服务发布端的详细服务配置信息转移到服务引用端，这样的话注册中心中就不需要存储服务提供者发布的详细服务配置信息了。这就是下面要讲的服务引用定义配置。

### 11.2.2服务引用定义配置

以下面的服务发布配置文件为例，它详细定义了服务userInfoService的各个方法的配置信息，比如超时时间和重传次数等。

<motan:service ref="userInfoService" requestTimeout="50" retries="2" interface="cn.sina.api.user.service.UserInfoService" basicService="serviceBasicConfig">

<motan:method name="addUserInfo" requestTimeout="300" retries="0"/>

<motan:method name="updateUserPortrait" requestTimeout="300" retries="0"/>

<motan:method name="modifyUserInfo" requestTimeout="300" retries="0"/>

<motan:method name="addUserTags" requestTimeout="300" retries="0"/>

<motan:method name="delUserTags" requestTimeout="300" retries="0"/>

<motan:method name="processUserCacheByNewMyTriggerQ" requestTimeout="300" retries="0"/>

<motan:method name="modifyObjectUserInfo" requestTimeout="300" retries="0"/>

<motan:method name="addObjectUserInfo" requestTimeout="300" retries="0"/>

<motan:method name="updateObjectUserPortrait" requestTimeout="300" retries="0"/>

<motan:method name="updateObjectManager" requestTimeout="300" retries="0"/>

<motan:method name="add" requestTimeout="300" retries="0"/>

<motan:method name="deleteObjectManager" requestTimeout="300" retries="0"/>

<motan:method name="getUserAttr" requestTimeout="300" retries="1" />

<motan:method name="getUserAttrList" requestTimeout="300" retries="1" />

<motan:method name="getAllUserAttr" requestTimeout="300" retries="1" />

<motan:method name="getUserAttr2" requestTimeout="300" retries="1" />

</motan:service>

可以像下面一样，把服务userInfoService的详细配置信息转移到服务引用配置文件中。

<motan:referer id="userInfoService" interface="cn.sina.api.user.service.UserInfoService" basicReferer="userClientBasicConfig">

<motan:method name="addUserInfo" requestTimeout="300" retries="0"/>

<motan:method name="updateUserPortrait" requestTimeout="300" retries="0"/>

<motan:method name="modifyUserInfo" requestTimeout="300" retries="0"/>

<motan:method name="addUserTags" requestTimeout="300" retries="0"/>

<motan:method name="delUserTags" requestTimeout="300" retries="0"/>

<motan:method name="processUserCacheByNewMyTriggerQ" requestTimeout="300" retries="0"/>

<motan:method name="modifyObjectUserInfo" requestTimeout="300" retries="0"/>

<motan:method name="addObjectUserInfo" requestTimeout="300" retries="0"/>

<motan:method name="updateObjectUserPortrait" requestTimeout="300" retries="0"/>

<motan:method name="updateObjectManager" requestTimeout="300" retries="0"/>

<motan:method name="add" requestTimeout="300" retries="0"/>

<motan:method name="deleteObjectManager" requestTimeout="300" retries="0"/>

<motan:method name="getUserAttr" requestTimeout="300" retries="1" />

<motan:method name="getUserAttrList" requestTimeout="300" retries="1" />

<motan:method name="getAllUserAttr" requestTimeout="300" retries="1" />

<motan:method name="getUserAttr2" requestTimeout="300" retries="1" />

</motan:referer>

这样的话，服务发布配置文件可以简化为下面这段代码，是不是信息精简了许多。

<motan:service ref="userInfoService" requestTimeout="50" retries="2" interface="cn.sina.api.user.service.UserInfoService" basicService="serviceBasicConfig">

</motan:service>

在进行类似的服务详细信息配置，由服务发布配置文件迁移到服务引用配置文件的过程时，尤其要注意迁移步骤的问题，这就是接下来我要给你讲的服务配置升级问题。

### 11.2.2服务配置升级

实际项目中，作者就经历过一次服务配置升级的过程。由于引用服务的服务消费者众多，并且涉及多个部门，升级步骤就显得异常重要，通常可以按照下面的步骤操作。

**⦁** 各个服务消费者在服务引用配置文件中添加服务详细信息。

**⦁** 服务提供者升级两台服务器，在服务发布配置文件中删除服务详细信息，并观察是否所有的服务消费者引用时都包含服务详细信息。

**⦁** 如果都包含，说明所有服务消费者均完成升级，那么服务提供者就可以删除服务发布配置中的服务详细信息。

**⦁** 如果有不包含服务详细信息的服务消费者，排查出相应的业务放进行升级，直至所有业务完成升级。

## 11.3 总结

今天我给你介绍了XML配置方式的服务发布和引用的具体流程，简单来说就是服务提供者定义好接口，并且在服务发布配置文件中配置要发布的接口名，在进程启动时加载服务发布配置文件就可以对外提供服务了。而服务消费者通过在服务引用配置文件中定义相同的接口名，并且在服务引用配置文件中配置要引用的接口名，在进程启动时加载服务引用配置文件就可以引用服务了。

在业务具体实践过程中可能会遇到引用服务的消费者众多，对业务的敏感度参差不齐的问题，所以在服务发布的时候，最好预定义好接口的各种配置。在服务规模不大，业务比较简单的时候，这样做比较合适。但是对于复杂业务，虽然服务发布时预定义好接口的各种配置，但在引用的服务消费者众多且同时访问的时候，可能会引起网络风暴。这种情况下，比较保险的方法是，把接口的各种配置放在服务引用配置文件里。

在进行服务配置升级过程时，要考虑好步骤，在所有服务消费者完成升级之前，服务提供者还不能把服务的详细信息去掉，否则可能会导致没有升级的服务消费者引用异常。

# 如何将注册中心落地

专栏第5期我们讲了服务注册和发现的原理，这里面的核心是服务提供者、服务消费者和注册中心这三个概念，以及它们之间的交互关系。

掌握了服务注册和发现的原理以后，我们就需要考虑如何把注册中心落地实现。**结合前面所讲的服务注册与发现的流程，在落地注册中心的过程中，我们需要解决一系列的问题，包括如何存储服务信息、如何注册节点、如何反注册、如何查询节点信息以及如何订阅服务变更等。**下面我们一一讲解这些问题的解决方案。

## 12.1 注册中心如何存储服务信息

注册中心既然是用来存储服务信息的，那么服务信息都包含哪些内容呢？

根据作者的实践经验，服务信息除了包含节点信息（IP和端口号）以外，还包括其他一些信息，比如请求失败时重试的次数，请求结果是否压缩等信息。因此服务信息通常用JSON字符串来存储，包含多个字段，每个字段代表不同的含义。

除此之外，服务一般会分成多个不同的分组，每个分组的目的不同。一般来说有下面几种分组方式

**⦁** 核心与非核心，从业务的核心程度来分

**⦁** 机房，从机房的维度来分

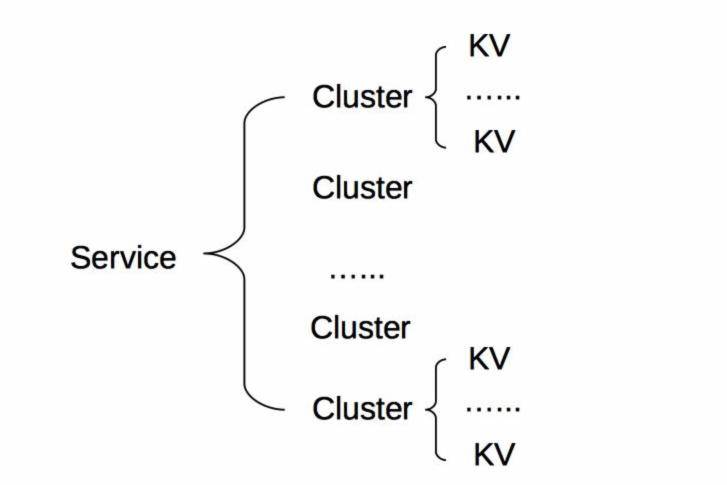
**⦁** 线上环境与测试环境，从业务场景维度来区分

所以**注册中心存储的服务信息一般包含三部分内容：分组、服务名以及节点信息**，节点信息又包括**节点地址**和**节点其他信息**。从注册中心获取的信息结构大致如下图所示。



（图片来源：<https://time.geekbang.org/column/article/39792>）

具体存储的时候，一般是按照“服务-分组-节点信息”三层结构来存储，可以用下图来描述。Service代表服务的具体分组，Cluster代表服务的接口名，节点信息用KV存储。



搞清楚了注册中心存储服务信息的原理后，再来看看注册中心具体是如何工作的，包括四个流程。

**⦁** 服务提供者注册流程

**⦁** 服务提供者反注册流程

**⦁** 服务消费者查询流程

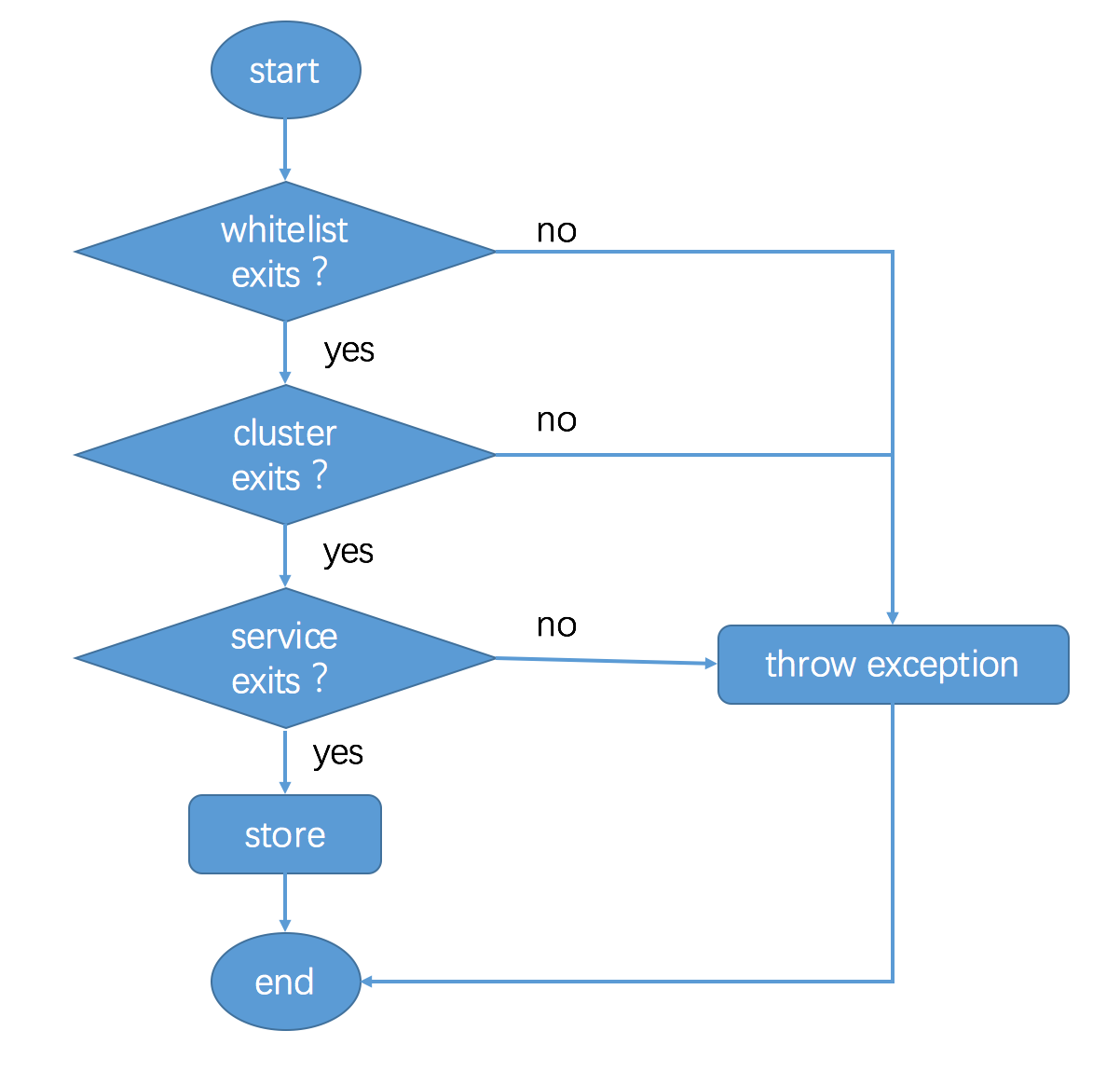
**⦁** 服务消费者订阅变更流程

接下来，我们来看看上面四个流程的实现方式。

## 12.2 注册中心是如何工作的

### 12.2.1 如何注册节点

知道了服务的节点信息如何存储之后，服务注册流程是怎么样的呢？下面这张流程图就描述了服务注册的流程



（图片来源：<https://time.geekbang.org/column/article/39792>）

根据作者的经验，服务注册流程主要有下面几个步骤：

**⦁** 首先查看要注册的节点是否在白名单内？如果不在就抛出异常，在的话继续下一步；

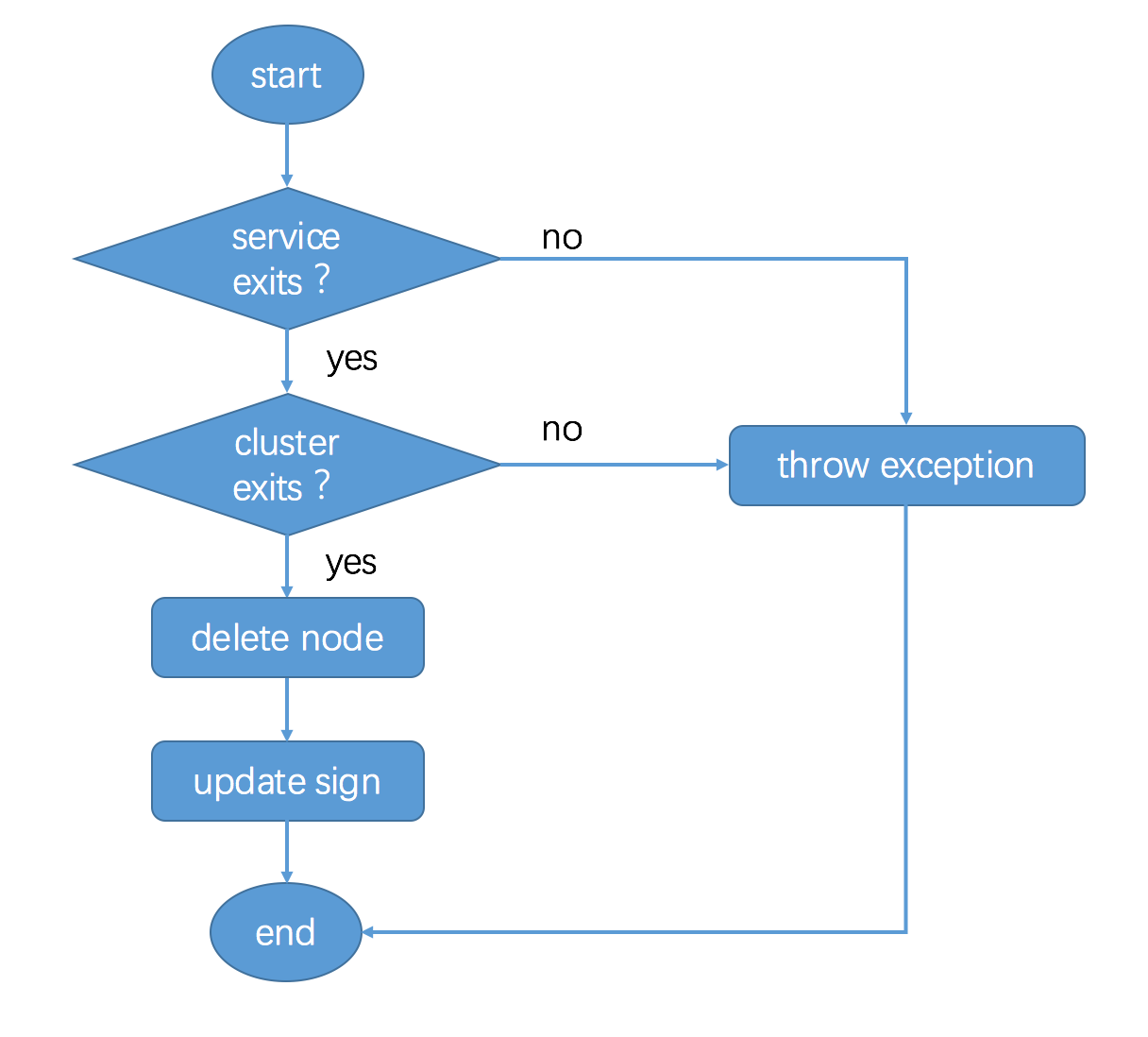
**⦁** 其次要查看注册的Cluster（服务的接口名）是否存在？如果不存在就抛出异常，存在的话继续下一步；

**⦁** 然后要检查Service（服务的分组）是否存在？如果不存在则抛出异常，存在的话继续下一步；

**⦁** 最后将节点信息添加到对应的Service和Cluster下面的存储中。

### 12.2.1 如何反注册

再来看下服务提供者节点反注册的流程，可以用下面这张流程图来描述



根据作者的经验，节点反注册流程主要包括下面几个步骤：

**⦁** 查看Service（服务的分组）是否存在，不存在就抛出异常，存在就继续下一步

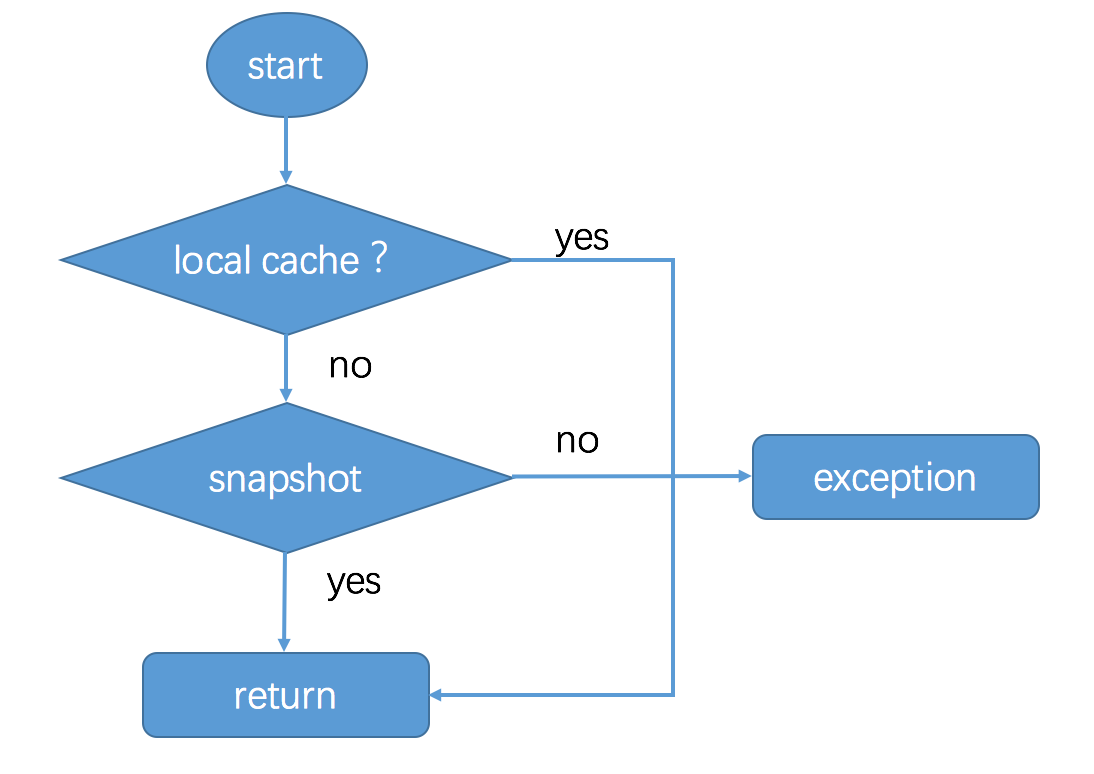
**⦁** 查看Cluster（服务的接口名）是否存在，不存在就抛出异常，存在就继续下一步

**⦁** 删除存储中Service和Cluster下对应的节点信息

**⦁** 更新Cluster的sign值

### 12.2.1 如何查询节点信息

关于服务消费者是如何从注册中心查询服务提供者的节点信息，可以用下面这张流程图来描述



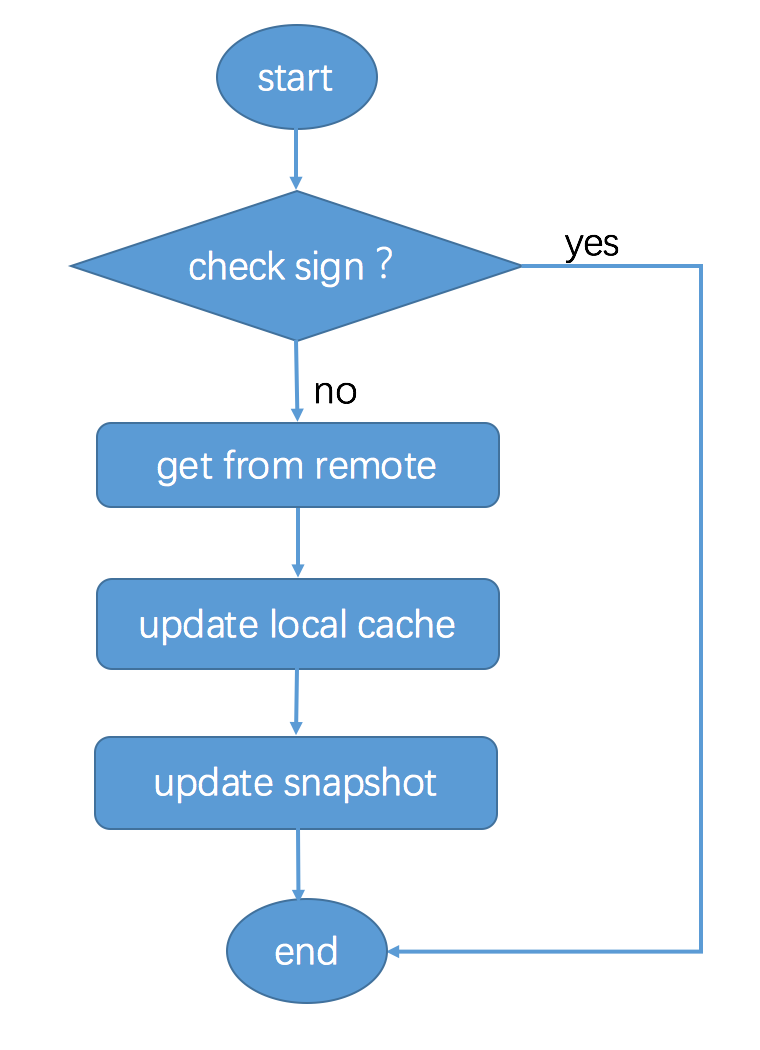
服务消费者查询节点信息主要分为下面几个步骤：

**⦁** 首先从localcache（本机内存）中查找，如果没有就继续下一步。这里为什么服务消费者要把服务信息存在本机内存呢？主要是因为服务节点信息并不总是时刻变化的，并不需要每一次服务调用都要调用注册中心获取最新的节点信息，只需要在本机内存中保留最新的服务提供者的节点列表就可以。

**⦁** 接着从snapshot（本地快照）中查找，如果没有就继续下一步。这里为什么服务消费者要在本地磁盘存储一份服务提供者的节点信息的快照呢？这是因为服务消费者同注册中心之间的网络不一定总是可靠的，服务消费者重启时，本地内存中还不存在服务提供者的节点信息，如果此时调用注册中心失败，那么服务消费者就拿不到服务节点信息了，也就没法调用了。本地快照就是为了防止这种情况的发生，即使服务消费者重启后请求注册中心失败，依然可以读取本地快照，获取到服务节点信息。

### 12.2.1 如何订阅服务变更

最后看下，服务消费者如何订阅服务提供者的变更信息呢？可以用下面这张流程图来描述。



主要分为下面几个步骤：

**⦁** 服务消费者从注册中心获取了服务的信息后，就订阅了服务的变化，会在本地保留Cluster的sign值。

**⦁** 服务消费者每隔一段时间，调用getSign()函数，从注册中心获取服务端该Cluster的sign值，并与本地保留的sign值做对比，如果不一致，就从服务端拉取新的节点信息，并更新localcahe和snapshot。

以上就是服务注册和反注册，服务查询与服务订阅变更的基本流程。除此之外，我再给你讲下我在实际项目实践中，实现服务注册与发现时遇到的几个问题，希望能给你帮助。

## 12.3 注册与发现的几个问题

### 12.3.1 多注册中心

理论上对于一个服务消费者来说，同一个注册中心交互是最简单的。但是不可避免的是，服务消费者可能订阅了多个服务，多个服务可能是由多个业务部门提供的，而且每个业务部门都有自己的注册中心，提供的服务只在自己的注册中心里有记录。这样的话，就要求服务消费者要具备在启动时，能够从多个注册中心订阅服务的能力。

根据作者的经验，还有一种情况是，一个服务提供者提供了某个服务，可能作为静态服务对外提供，有可能又作为动态服务对外提供，这两个服务部署在不同的注册中心，所以要求服务提供者在启动的时候，要能够同时向多个注册中心注册服务。

**也就是说，对于服务消费者来说，要能够同时从多个注册中心订阅服务；对于服务提供者来说，要能够同时向多个注册中心注册服务。**

### 12.3.2 并行订阅服务

通常一个服务消费者订阅了不止一个服务，在我经历的一个项目中，一个服务消费者订阅了几十个不同的服务，每个服务都有自己的方法列表以及节点列表，服务消费者在服务启动时，会加载订阅的服务配置，调用注册中心的订阅接口，获取每个节点列表并初始化连接。

开始我们采用了串行订阅的方式，每订阅一个服务，服务消费者调用一次注册中心的订阅接口，获取这个服务的节点列表并初始化连接，总共要执行几十次这样的过程。在某些服务节点的初始化连接过程中，出现连接超时的情况，后续所有的服务节点的初始化连接都要等待它完成，导致服务消费者启动变慢，最后耗费了将近五分钟时间来完成所有服务节点的初始化连接过程。

后来我们改成了并行订阅的方式。订阅一个服务就单独用一个线程来处理。这样的话即使遇到个别服务节点连接超时，其他服务节点的初始化连接也不受影响，最慢也就是这个服务节点的初始化连接耗费的时间，最终所有服务节点的初始化连接耗时控制在了30秒以内。

### 12.3.3 批量反注册服务

通常一个服务提供者节点提供不止一个服务，所以注册和反注册都需要多次调用注册中心。在与注册中心的多次交互中。可能由于网络抖动、注册中心集群异常等原因。导致个别调用失败。对于注册中心来说。偶发的注册调用失败对服务调用基本没有影响。其结果顶多就是某一个服务少了一个可用的节点。但偶发的反注册调用失败会导致不可用的节点残留在注册中心中。变成“僵尸节节”， 但服务消费者端还会把它当成“活节点”，继续发起调用终致调用失败。

以前我们的业务中经常遇到这个问题，需要定时去清理注册中心中的“僵尸节点”。后来我们通过优化反注册逻辑。对于下线机器、节点销毁的场景，通过调用注册中心提供的批量反注册接口，一次调用就可以把该节上提供的所有服务同时反注册掉，从而避免了“僵尸节点”的出现。

### 12.3.4 服务变更信息增量更新

服务消费者端启动时，除了会查询订阅服务的可用节点列表做初始化连接，还会订阅服务的变更，每隔一段时间从注册中心获取新的服务节信息标记sign，并与本地保存的sign值作比对，如果不一样，就会调用注册中心获取新的服务节信息。

一般情况下，按照这个过程是没问题的，但是在网络频繁抖动时，服务提供者上报给注册中心的心跳可能会一会儿失败一会儿成功，这时候注册中心就会频繁更新服务的可用节信息，导致服务消费者频繁从注册中心拉取新的服务可用节点信息，严重时可能产生网络风暴，导致注册中心带宽被打满。

为了减少服务消费者从注册中心中拉取的服务可用节点信息的数据量，这个时候可以通过增量更新的方式，注册中心只返回变化的那部分节中信息，尤其在只有少数节点信息变更时，此举可以大大减少服务消费者从注册中心拉取的数据量，从而最大程度避免产生网络风暴。

## 12.4 总结

今天我给你讲解了在注册中心实际使用过程中。服务注册、服务反注册、服务订阅和服务变更的实现方式，并列举了几个我在服务注册与发现的过程中遇到的典型问题。

而针对这些异常情况。我都给出了对应的解决方案。这些方案都是经过实际业务验证的。对于大部分中小团队在应用场景面临的问题。应该足以应对。

# 开源服务注册中心如何选型

上一我给你讲了服务注册中心的落地实践，以及在实际应用中可能会遇到的问题和对应的解决方法，关于注册中心，如果你的团队有足够的人才和技术储备，可以选择自己研发注册中心。但对于大多数中小规模团队来说，我的建议是最好使用业界开源的、应用比较成熟的注册中心解决方案，把精力投入到业务架构的改造中，不要自己造轮子。

当下主流的服务注册与发现的解决方案主要有两种：

⦁ 应用内注册与发现：注册中心提供服务端和客户端的SDK，业务应用通过引入注册中心提供的 SDK，通过SDK与注册中心交互，来实现服务的注册和发现。

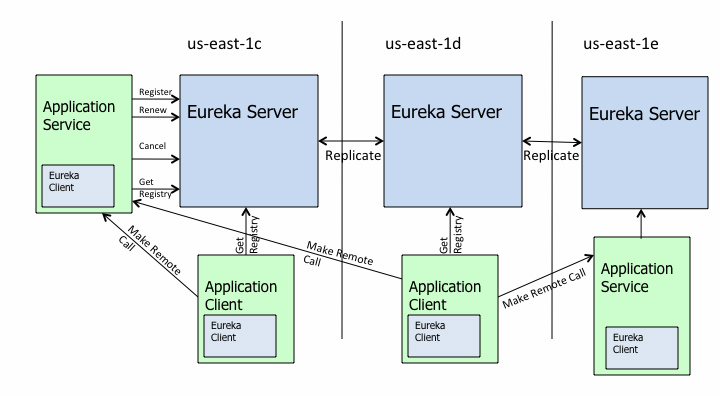
⦁ 应用外注册与发现：业务应用本身不需要通过SDK与注册中心打交道，而是通过其他方式与注册中心交互，间接完成服务注册与发现。

下面我会用两个业界使用比较成熟的注册中心开源实现，来讲解下应用内和应用外两种解决方案的不同之处。

## 13.1 两种典型的注册中心实现

### 13.1.1 应用内

采用应用内注册与发现的方式，最典型的案例要属Netflix开源的Eureka，官方架构图如下。



（图片来源：<https://time.geekbang.org/column/article/39797>）

对这这张图，我来介绍下Eureka的架构，它主要由三个重要的组件组成：

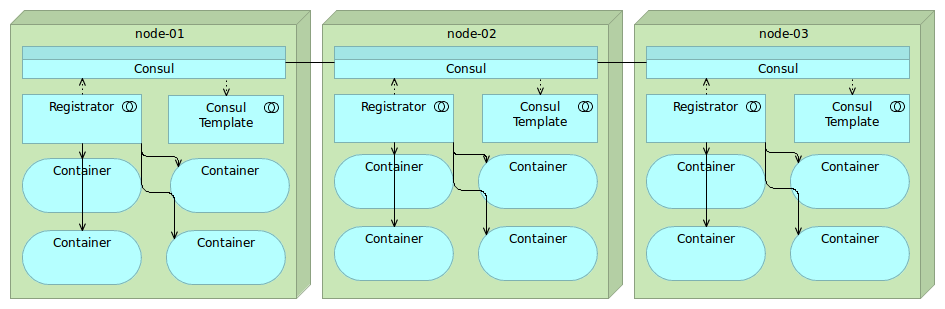
⦁ Eureka Server：注册中心的服务端，实现了服务信息注册、存储以及查询等功能。

⦁ 服务端的Eureka Client：集成在服务端的注册中心SDK，服务提供者通过调用SDK，实现注册、反注册等功能。

⦁ 客户端的Eureka Client：集成在客户端的注册中心SDK，服务消费者通过调用SDK，实现服务订阅、服务更新等功能。

### 13.1.2 应用外

采用应用外方式实现服务注册和发现，最典型的案例是开源注册中心Consul，它的架构图如下。



（图片来源：<https://time.geekbang.org/column/article/39797>）

通过这张架构图，可以看出来使用Consul实现应用外服务注册和发现主要依赖三个重要组件：

⦁ Consul：汪册中心的服务端，实现服务汪册信息的存储，并提供汪册和发现服务。

⦁ Reglstrator：一个开源的第三方服务管理器项目，它通过监听服务部著的Docker实例是否存活，来负责服务提供者的汪册和消毁。

⦁ Consul Template：定时从注册中心服务端获取最新的服务提供者节点列表并刷新LB配置（比如Nginx的upstream）。这样服务消费者就通过访问Nginx就可以获取新的服务提供者信息。

对比小结一下，这两种解决方案的不同之处在于应用场景，应用内的解决方案一般适用于服务提供者和服务消费者同属于一个技术体系；应用外的解决方案一般适用于服务提供者和服务消费者采用了不同技术体系的业务场景，比如服务提供者提供的是C++服务而服务消者是一个Java应用，这时候采用应用外的解决方案就不依赖于具体一个技术体系。同时，对于容器化后的云应用来说。 一般不适合采用应用内SDK的解决方案。因为这样会侵入业务，而应用外的解决方案正好能够解决这个问题。

## 13.2 注册中心选项要考虑的两个问题

在选择注册中心解决方案的时候，除了要考虑是采用应用内注册还是应用外注册的方式以外，还有两个值得关汪的问题，一个是高可用性，一个是数据一致性，下面我来给你详细解释下为什么。

### 13.2.1 高可用

注册中心作为服务提供者和服务消费者之间沟通的纽带，它的高可用性十分重要。试想，如果注册中心不可用了，那么服务提供者就无法对外暴露自己的服务，而服务消费者也无法知道自己想要调用的服务的具体地址，后果将不堪设想。

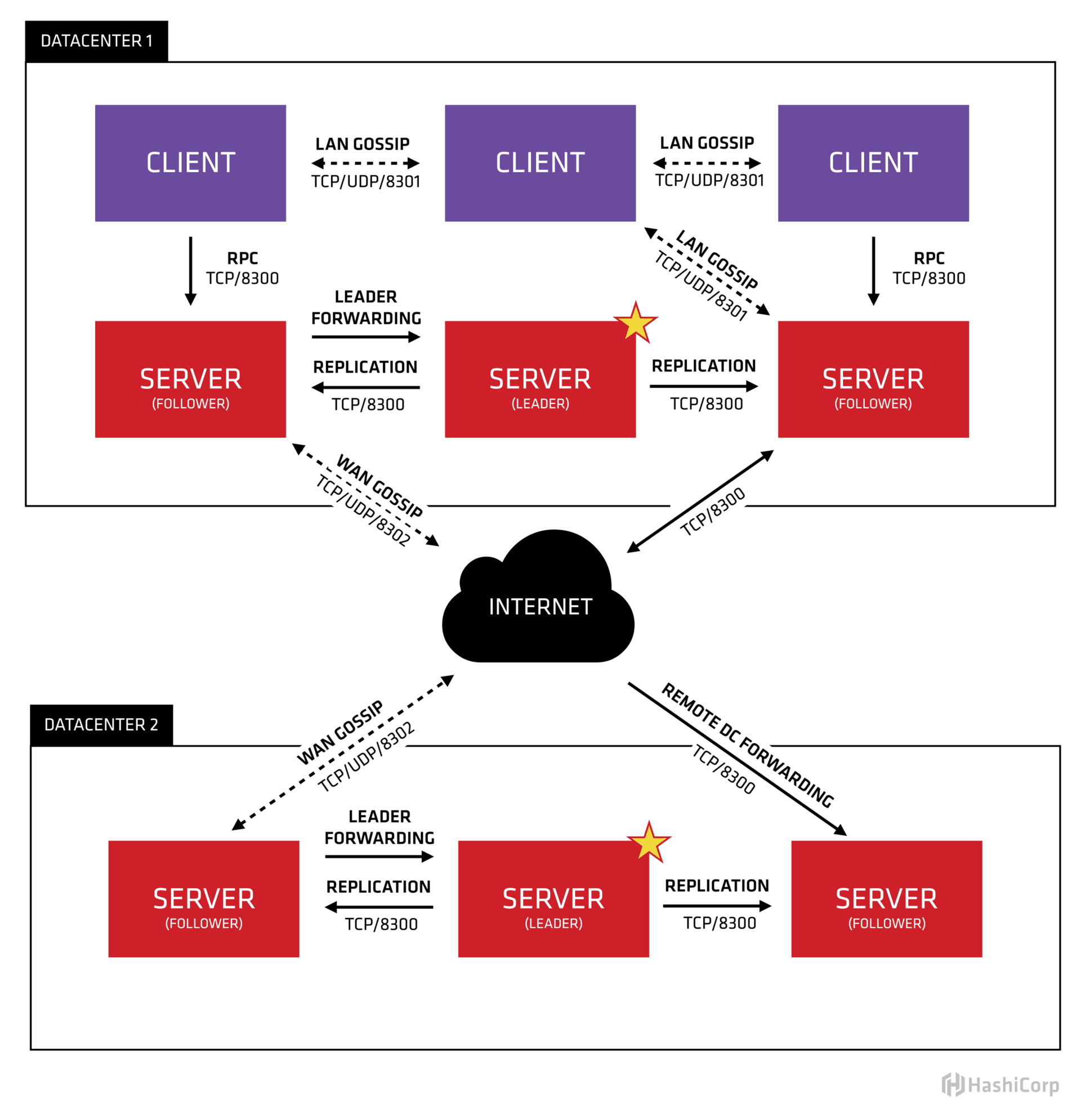
根据我过往的实践经验，实现高可用性的方法主要有两种：

⦁ 集群部署：顾名思义就是通过部著多个实例组成集群来保证高可性，这样的话即使有部分机器宕机，将访问迁移到正常的机器上就可以保证服务的正常访问。

⦁ 多IDC部署：就是部著在不止一个机房，这样能保证即使一个机房因为断电或者光缆被挖断等不可抗力因素时，仍然可以通过把请求迁移到其他机房来保证服务的正常访问。

我们以Consul为例，来看看它是如何通过这两种方法来保证注册中心的高可用性。

从下面的官方架构图中你可以看到，一方面，在每个数据中心（DATACENTER）内都有多个注册中心Server节中可供访问；另一方面还可以部著在多个数据中心来保证多机房高可用性。



（图片来源：<https://time.geekbang.org/column/article/39797>）

### 13.2.1 数据一致性

为了保证注册中心的高可用性，注册中心的部著往往都采用集群部署，并且还通常部著在不止一个数据中心（多IDC部署）。这样的话就会引出另一个问题，多个数据中心之间如何保证数据一致？如何确保访问数据中心中任何一台机器都能得到正确的数据？

这里就涉及分布式系统中著名的CAP理论，即同时满足一致性、可用性、分区容错性这三者是不可能的，其中C〔Consistency）代表一致性，A（Availability〕代表可用性，P（Partition Tolerance〕代表分区容错性。

为什么说CAP三者不能被同时满足的呢？

你可以想在一个分布式系统里面，包含了多个节点，节点之间通过网络连接在一起。正常情况下，通过网络，从一个节点可以访问任何别的节点上的数据。但是有可能出现网络故障，导致整个网络被分成了互不连通的区域，这就叫作分区。一旦出现分区，那么一个区域内的节点就没法访问其他节点上的数据了，最好的办法是把数据复制到其他区域内的节点，这样即使出现分区，也能访问任意区域内节点上的数届，**这就是分区容错性**。

但是把数据复制到多个节点就可能出现数据不一致的情况，**这就是一致性**。

要保证一致，就必须等待所有节点上的数据都更新成功才可用，**这就是可用性**。

**总的来说，就是数据节点越多，分区容错性越高，但数据一致性越难保证。为了保证数据一致性，又会带来可用性的问题。**

而注册中心一般采用分布式集群部署也面临着CAP的问题，根据CAP不能同时满足，所以不同的注册中心解决方案选择的方向也就不同。大致可分为两种。

⦁ CP型注册中心：

牺牲可用性来保证数据强一致性，最典型的例子就是**ZooKeeper、etcd、 Consul**了。ZooKeeper集群内只有一个Leader，而且在Leader无法使用的时候通过Paxos法选举出一个新的Leader。这个Leader的目的就是保证写信息的时候只向这个Leader写入，Leader会同步信息到Followers，这个过程就可以保证数据的强一致性。但如果多个 ZooKeeper之间网络出现问题，造成出现多个Leader，发生脑裂的话，注册中心就不可用了。 而etcd和Consul集群内都是通过raft协议来保证强一致性，如果出现脑裂的话，注册中心也不可用。

⦁ AP型注册中心：

牺牲一致性来保证可用性，典型的例子就是**Eureka**了。对比下 Zookeeper，Eureka不用选举一个Leader，每个Eureka服务器单独保存服务注册地址，因此，有可能出现数据信息不一致的情况。但是当网络出现问题的时候，每台服务器都可以完成独立的服务。

而对于注册中心来说，主要的功能是服务的注册和发现，在网络出现问题的时候，可用性的需求要远远高于数据一致性。即使因为数据不一致。注册中心内引入了不可用的服务节点，也可以通过其他措施来避免，比如客户端的快速失败机制等，只要实现最终一致性，对于注册中心来说就足够了。因此，选择AP型注册中心，一般更加合适。

## 13.3 总结

总的来说，在选择开源注册中心解决方案的时候，要看业务的具体场景。

⦁ 如果你的业务体系都采用Java语言的话，Netflix开源的Eureka是一个不错的选择，并且它作为服务注册与发现解决方案，能够最大程度的保证可用性，即使出现了网络问题导致不同节点间数据不一致，你仍然能够访问Eureka获取数据。

⦁ 如果你的业务体系语言比较复杂，Eureka也提供了Sidecar的解决方案，也可以考虑使用Consul，它支持了多种语言接入。包括Go、Python、PHP、Scala、Java，Erlang、Ruby、 N0de.js、.NET、Perl等。

⦁ 如果你的业务已经是云原生的应用，可以考虑用Consul，搭配Registrator和Consul Template来实现应用外的服务注册与发现。

# 开源RPC框架如何选型

## 14.1 开源框架综述

专栏第6期我给你讲解了RPC远程调用的原理，简单回顾一下一个**完整的RPC框架主要有三部分组成：通信框架、通信协议、序列化和反序列化格式**。根据我的经验，想要开发一个完整的RPC 框架，并且应用到线上生产环，至少要投入三个人力半年以上的时间。这对于大部分中小团队来说，人力成本和时间成本都是不可接受的，所以我建议还是选择开源的RPC框架比较合适。

那么业界应用比较广泛的开源RPC框架有哪些呢？

简单划分的话，主要分为两类：一类是跟某种特定语言平台绑定的，另一类是与语言无关即跨语言平台的。

跟语言平台绑定的开源RPC框架主要有下面几种。

⦁ Dubbo：国内最早开源的RPC框架，由阿里巴巴公司开发并于2011年末对外开源，仅支持Java语言。

⦁ Motan：微博内部使用的的RPC框架，于2016年对外开源，仅支持Java言。

⦁ Tars：腾讯内部使用的RPC框架，于2017年对外开源，仅支持C++语言 。

⦁ SpringCloud：国外Pivotal公司2014年对外开源的RPC框架，仅支持Java语言，最近几年生态发展得比较好，是比较火的RPC框架 。

而跨语言平台的开源RPC框架主要有以下几种。

⦁ gRPC：Google于2015年对外开源的跨语言RPC框架，支持常用的C++、Java、Python、Go、Ruby、PHP、Android Java、Objective-C等多种语言。

⦁ Thrift：最初是由Facebook开发的内部系统跨语言的RPC框架，2007年贡献给了Apache基金，成为Apache开源项目之一。支持常用的C++、Java、PHP、Python、Ruby、Erlang等多种语言。

所以很明显。如果你的业务场景仅仅局限于一种语言的话，可以选择跟语言绑定的RPC框架中的一种；如果涉及多个语言平台之间的相互调用，就应该选择跨语言平台的RPC框架。

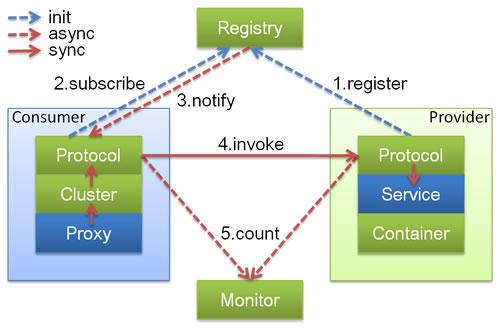
针对每一种RPC框架，它们具体有何区别？该如何择呢？接下来，我就从每个框架的实现角度来具体给你讲解。当你知道了他们的具体实现，也就能知道他们的优缺点以及适用场景了。

限定语言平台的开源RPC框架

## 14.2 限定语言RPC框架

### 14.2.1 Dubbo

先来聊聊Dubbo，Dubbo可以说是国内开源最早的RPC框架了，目前只支持Java语言，它的架构可以用下面这张图展示。



（图片来源：<https://time.geekbang.org/column/article/39809>）

从图中你能看到，Dubbo的架构主要包含四个角色，图中Consumer是服务消费者，Provider是服务提供者，Registry是注册中心，Monitor是监控系统。

具体的**交互流程**是：Consumer一端通过注册中心获取到Provider节点后，通过Dubbo的客户端 SDK与Provider建立连接，并发起调用。Provider一端通过Dubbo的服务端SDK接收到 Consumer的请求，处理后再把结果返回给Consumer。

可以看出服务消費者和服务提供者都要引入Dubbo的SDK才能完成RPC调用，因为Dubbo 本身是采用Java语言实现的，所以要求服务消费者和服务提供者也都必须采用Java语言实现才可以应用。

我们再来看下Dubbo的**调用框架**是如何实现的。

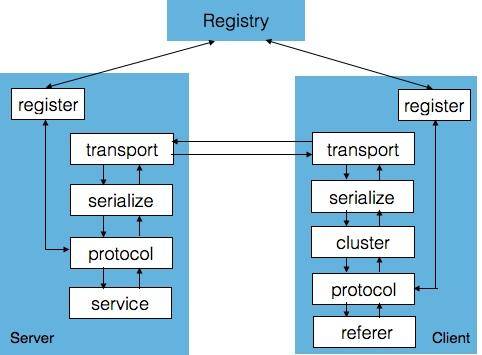
⦁ 通信框架方面，Dubbo默认采用了Netty作为通信框架 。

⦁ 通信协议方面，Dubbo除了支持私有的Dubbo协议外，还支持RMI协议、Hession协议、 HTTP协议、Thrift协议等。

⦁ 序列化格式方面，Dubbo支持多种序列化格式，比如Dubbo、Hession、JSON、Kryo、FST 等。

### 14.2.2 Motan

Motan是国内另外一个比较有名的开源RPC框架，同样也只支持Java语言，它的架构可以用下面这张图描述。



（图片来源：<https://time.geekbang.org/column/article/39809>）

Motan与Dubbo的架构类似，都要在Client端（服务消者）和Server端〔服务提供者〕引 入SDK，其中Motan框架主要包含下面几个功能模块。

⦁ register：用来和注册中心交互，包括注册服务、订阅服务、服务变更通知、服务心跳发送等功能。Server端会在系统初始化时通过register模块注册服务，Client会在系统初始化时通过register模块订阅到具体提供服务的Server列表，当Server列表发生变更时也由register模块通知Client。

⦁ protocol：用来进行RPC服务的描述和RPC服务的配置管理，这一层还可以添加不同功能的 filter用来完成统计、并发限制等功能。

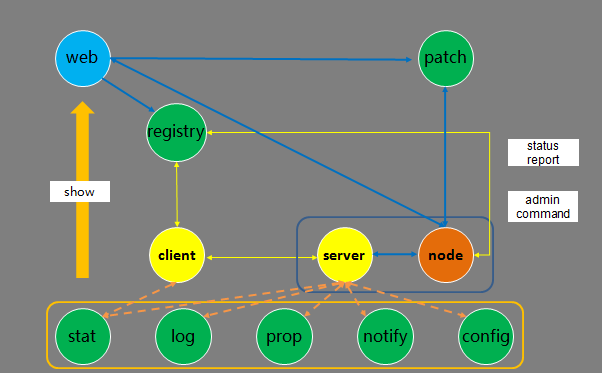
⦁ serialize：将RPC请求中的参数、结果等对象进行序列化与反序列化，即进行对象与字节流的互相转换，默认使用对Java更友好的Hessian2进行序列化。

⦁ transport：用来进行远程通信，默认使用Netty NIO的TCP长链接方式。

⦁ cluster：Client端使用的模块，cluster是一组可用的Server在逻辑上的封装，包含若干可以提供RPC服务的Server，实际请求时会根据不同的高可用与负载均衡策略选择一个可用的 Server发起远程调用。

### 14.2.3 Tars

Tars是腾讯根据内部多年使用微服务框架的实践，总结而成的开源项目，仅支持C++语言，它的架构图如下：



（图片来源：<https://time.geekbang.org/column/article/39809>）

Tars的架构交互主要包括以下几个流程：  
⦁ 服务发布流程：

在web系统上传server的发布包到patch，上传成功后，在web上提交发布server请求，由registry服务传达到node，然后node拉取server的发布包到本地，拉起server服务。

⦁ 管理命令流程：

web系统上的可以提交管理server服务命令请求，由registry服务传达到 node服务然后由node向server发管理命令

⦁ 心跳上报流程：

server服务运行后，会定期上报心跳到node，node然后把服务心跳信息上报到registry服务，由registry进行统一管里。

⦁ 信息上报流程：

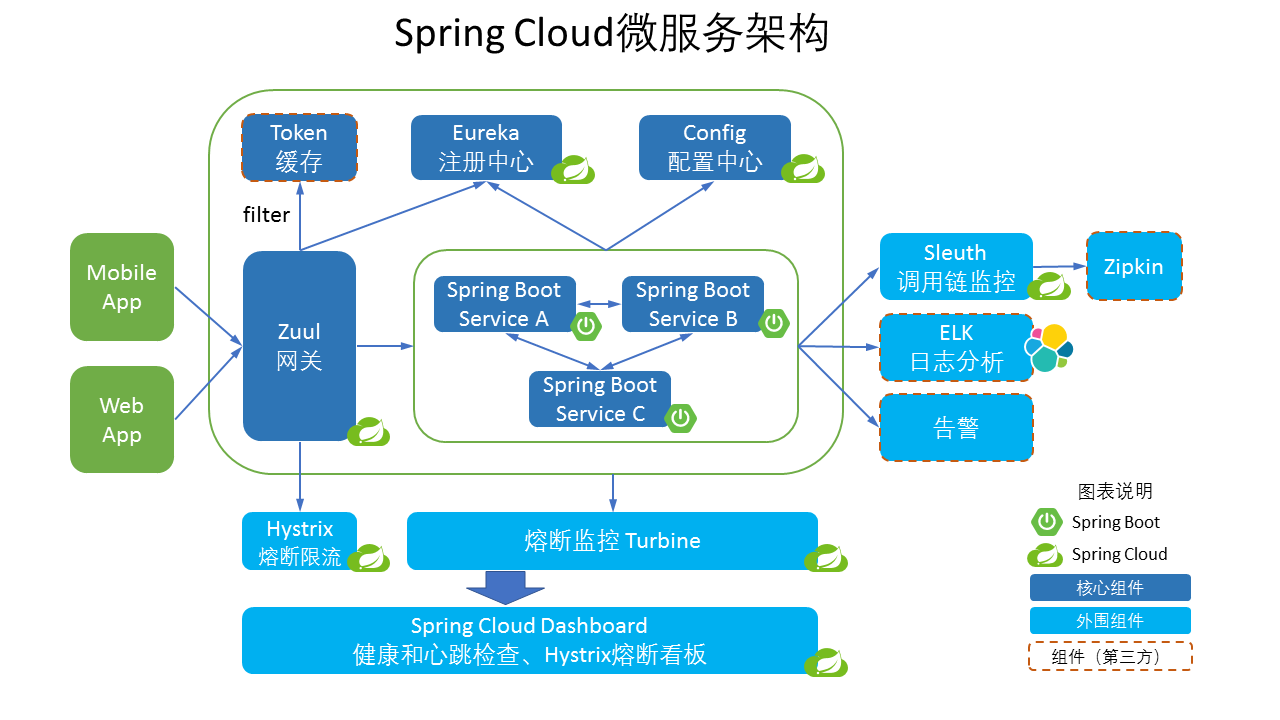
server服务运行后，会定期上报统计信息到stat，打远程日志到log，定期上报属性信息到prop、上报异信息到notify、从config拉取服务配置信息。

⦁ client访问server流程：

client可以通过server的对象名Obj间接访问server，client会从 registry上垃取server的路由信息（如IR、Port,信息），然后根据具体的业务特性（同步或者异步，TCP或者UDP方式〕访问server〔当然client也可以通过IP/Port直接访问 server）。

### 14.2.4 Spring Cloud

Spring Cloud是为了解决微服务架构中服务治理而提供的一系列功能的开发框架，它是完全基于Spring Boot进行开发的，Spring Cloud利用Spring Boot特性整合了开源行业中优秀的组件， 整体对外提供了一套在微服务架构中服务治理的解决方案。因为Spring Boot是用Java语言编写的，所以目前Spring Cloud也只支持Java语言平台，它的架构图可以用下面这张图来描述：



（图片来源：<https://time.geekbang.org/column/article/39809>）

由此可见，Spring Cloud微报务架构是由多个组件一起组成的，各个组件的交互流程如下。

⦁ 请求统一通过API网关在Zuul来访问内部服务，先经过Token进行安全认证。

⦁ 通过安全认证后，网关Zuul从注册中心Eureka获取可用服务节列表。

⦁ 从可用服务节点中选取一个节点，然后把请求分发到这个节点。

⦁ 整个请求过程中，Hystrix组件负责处理服务超时熔断，Turbine组件负责监控服务间的调用和熔断相关指标，Sleuth组件负责调用链监控，ELK负责日志分析。

## 14.3 限定语言RPC框架对比选型

介绍完这4种限定语言的开源RPC框架后我们该如何择呢？

很显然。如果你的语言平台是C++，那么只能选择Tars，而如果是Java的话，可以选择Dubbo、Motan或者SpringCloud。这时你又要问了，它们三个又该如何抉择呢？

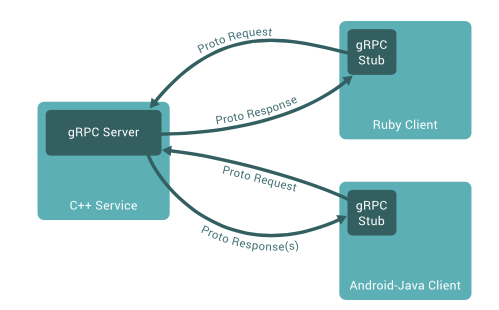
仔细分析，可以看出Spring Cloud不仅提供了基本的RPC框架功能，还提供了服务注册组件、配置中心组件、负载均衡组件、断路器组件、分布式消息追踪组件等一系列组件，也难怪被技术圈的人称之为 “Spring Cloud全家桶”。如果你不想自己实现这些功能，那么Spring Cloud基本可以满足你的全部需求。而Dubbo、Motan基本上只提供了基础的RPC框架的功能，其他微服务组件都要自己去实现。

不过由于Spring Cloud的RPC通信采用了HTTP协议，相比Dubbo和Motan所采用的私有协 议来说，在高并发的通信场景下，性能相对要差一些，所以对性能有苛刻要求的情况下，可以考虑 Dubbo和Motan 。

## 14.4 跨语言RPC框架

### 14.4.1 gRPC

先来看下gRPC。它的原理是通过IDL（lnterface Definition Language）文件定义服务接口的参 数和返回值类型，然后通过代码生成程序生成服务端和客户端的具体实现代码，这样在gRPC里，客户端应用可以像调用本地对象一样调用另一台服务器上对应的方法。



（图片来源：<https://time.geekbang.org/column/article/39809>）

它的主要特性包括三个方面。

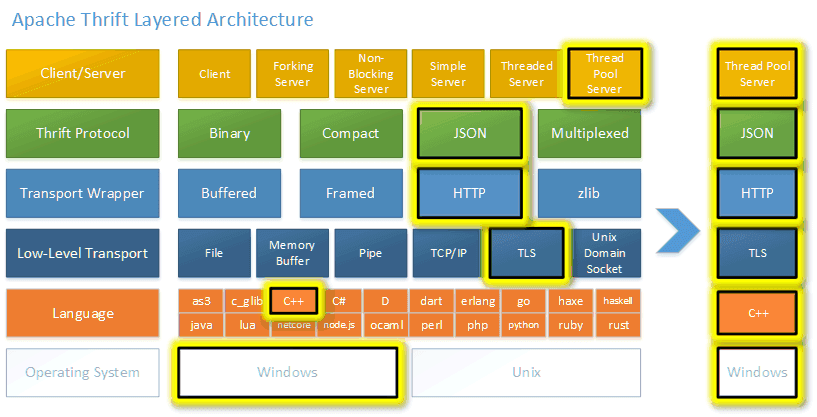
⦁ 通信协议采用了HTTP/2，因为HTTP/2提供了连接复用、双向流、服务器推送、请求优先级、首部压缩等机制，所以在通信过程中可以节省带宽、降低TCP连接次数、节省CPU，尤其对于移动端应用来说，可以帮助延长电池寿命

⦁ IDL使用了ProtoBuf，ProtoBuf是由Google开发的一种数据序列化协议，它的压缩和传输效率极高，语法也简单，所以被广泛应用在数据存储和通信协议上。

⦁ 多语言支持，能够基于多种语言自动生成对应语言的客户端和服务端的代码。

### 14.4.2 Thrift

再来看下Thrift，Thrift是一种轻量级的跨语言RPC通信方案，支持多达25种程语言。为了支持多种语言。跟gRPC一样，Thrift也有一套自己的接口定义语言IDL，可以通过代码生成器，生成各种语言的Client端和Server端的SDK代码，这样就保证了不同语言之间可以相互通信。它的架构图可以用下来描述。



（图片来源：<https://time.geekbang.org/column/article/39809>）

从这张图上可以看出Thrift RPC框架的特性。

⦁ 支持多种序列化格式：如Binary、Compact、JSON、Multiplexed等。

⦁ 支持多种通信方式：如Socket、Framed、File、Memory、zlib等

⦁ 服务端支持多种处理方式：如Simple、ThreadPool、Non-Blocking等。

## 14.5 跨语言RPC框架对比选型

那么涉及跨语言的服务调用场景到底该选择gRPC还是Thrift呢？

从成熟上来讲，Thrift因为诞生的时间要早于gRPC，所以使用的范围要高于gRPC，在 HBase、Hadoop、Scribe、Cassandra等许多开源组件中都得到了广泛的应用。而且**Thrift支持多达25种语言，这要比gRPC支持的语言更多**，所以如果到gRPC不支持的语言场景下，选择 Thrift更合适。

但gRPC作为后起之秀，因为采用了HTTP/2作为通信协议、ProtoBuf作为数据序列化格式，在**移动端设备的应用以及对传输带宽比较敏感**的场景下具有很大的优势，而且开发文档丰富，根据ProtoBuf文件生成的代码要比Thrift更简洁一些，在使用难易程上更占优势，所以如果使用的语言平台gRPC支持的话，建议还是采用gRPC比较好。

## 14.6 总结

以上就是我对几种使用广泛的开源RPC框架的选型建议，也是基于它们目前现状所作出的判断。从长远来看，支持多语言是RPC框架未来的发展趋势。正是基于此判断，各个RPC框架都提供了Sidecar组件来支持多语言平台之间的RPC调用。

⦁ Dubbo在去年年底又重启了维护并且宣称要引入Sidecar组件来构建Dubbo Mesh提供多语言支持

⦁ Motan也在去年对外开源了其部分的Sidecar组件：Motan-go，目前支持PHP,Java语言之间的相互调用。

⦁ Spring Cloud也提供了Sidecar组件：spring-cloud-netflix-sideca，可以让其他语言也可以使用Spring Cloud的组件。

所以未来语言不会成为使用上面这几种RPC框架的约束，而gRPC和Thrift虽然支持跨语言的 RPC调用，但是因为它们只提供了基本的RPC框架功能，缺乏一系列配套的服务化组件和服务治理功能的支撑。所以使用它们作为跨语言调用的RPC框架，就需要自己考虑注册中心、熔断、 限流、监控、分布式追踪等功能的实现。不过好在大多数功能都有开源实现。可以自接采用。

## 14.7 思考题

同样是支持跨语言的RPC调用，你觉得gRPC这类的跨语言服务框架和Motan-go类的 Sidecar方案有什么区别？在使用过程中都要注意什么？

# 如何搭建一个可靠的监控系统

专栏第7我给你讲解了监控系统的实现原理，先来简单回顾一下，一个监控系统的组成主要涉及四个环节：数据收集、数据传输、数据处理和数据展示。不同的监控系统实现方案，在这四个环节所使用的技术方案不同，适合的业务场景也不一样。

目前，比较流行的开源监控系统实现方案主要有两种：以ELK为代表的集中式日志解决方案，以及Graphite、TICK和Prometheus等为代表的时序数据库解决方案。接下来我就以这几个常见的监控系统实现方案，谈谈它们的实现原理，分别适用于什么场景，以及具体该如何做技术选型。

## 15.1 ELK

ELK是Elasticsearch、Logstash、Kibana一个开源软件产品首字母的缩写，它们三个通常配合使用，所以被称为ELK Stack，它的架构可以用下面的图片来描述。



（图片来源：<https://time.geekbang.org/column/article/39907>）

这三个软件的功能也各不相同。

⦁ Logstash负数据收集和传输，它支持动态地从各种数据源收集数据，并对数据进行过滤分析、格式化等，然后存储到指定的位置。

⦁ Elasticsearch负责数据处理，它是一个开源分布式搜索和分析引擎，具有可伸缩、高可靠和易管理等恃点，基于Apache Lucene架构，能对大容量的数据进行接近实时的存储、搜索和分析操作，通常被用作基础搜索引擎。

⦁ Kibana负责数据展示，也是一个开源和免费的工具，通常和Elasticsearch搭配使用。对其中的数据进行搜索、分析并且以表的方式展示。

这种架构因为要在各个服务器上部署Logstash来从不同的数据源收集数据，所以比较消耗CPU 和内存资源。容易造成服务器性能下降。因此后来又在Elasticsearch、Logstash、Kibana之外引入了Beats作为数据收集器。相比于Logstash，Beats所占系统的CPU和内存几乎可以忽略不计。可以装在每台服务器上做轻量型代理。从成百上千或者成千上万台机器向Logstash或者直接向Elasticsearch发送数据。

其中。Beats支持多种数据源，主要包括：

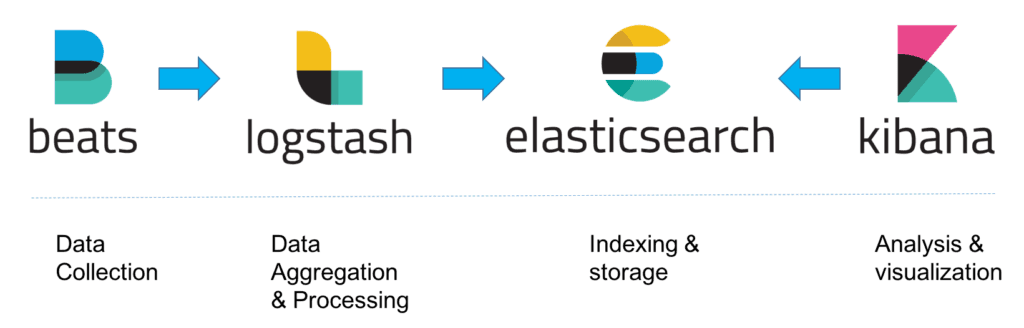
⦁ Packetbeat：用来收集网络流量数据

⦁ Topbeat：用来收集系统进程的CPU和内存使用情况等数据

⦁ Filebeat：用来收集文件数据

⦁ Winlogbeat：用来收集Windows事件日志

Beats将收集到的数据发送到Logstash。经过Logstash解析、过滤后，再将数据发送到 Elasticsearch，最后由Kibana展示，架构就变成下面这张图里描述的了。



（图片来源：<https://time.geekbang.org/column/article/39907>）

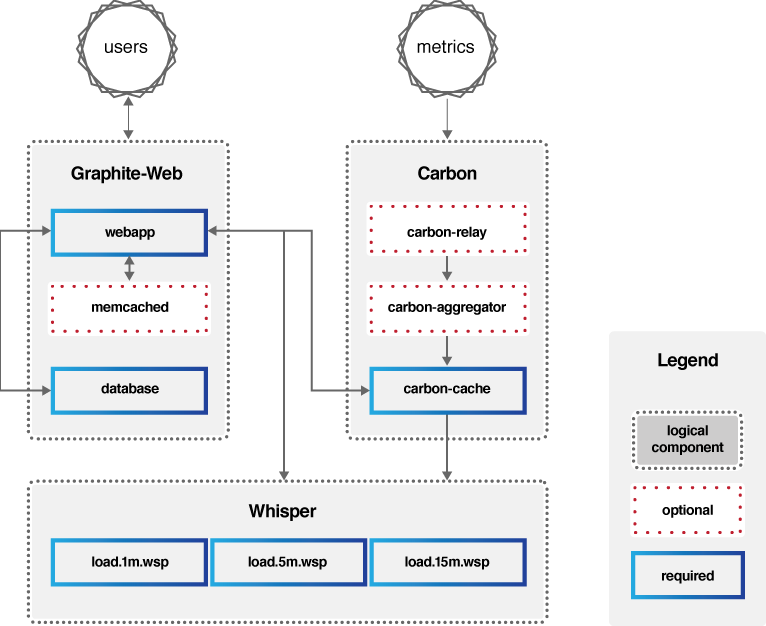
## 15.2 Graphite

Graphite的组成主要包括三部分：Carbon、Whisper、Graphite-Web，它的架构可以用下图来描述。

⦁ Carbon：主要作是接收被监控节点的连接，收集各个指标的数据，将这些数据写入carbon-cache并最终持久化到Whisper存储文件中去。

⦁ Whisper：一个简单的时序数据库，主要作用是存储时间序列数据，可以按照不同的时间粒度来存储数据，比如1分钟1个点、5分钟1个点、15分钟1个点三个精度来存储监控数据。

⦁ Graphite-Web：一个WebApp，其主要功能是绘制报表与展示，即数据展示。为了保证 Graphite-Web能及时绘制出图形，Carbon在将数据写入Whisper存储的同时，会在 carbon-cache中同时写入一份数据，Graphite-Web会先查询carbon-cache，如果没有再查询Whisper存储



（图片来源：https://time.geekbang.org/column/article/39907）

也就是说Carbon负责数据处理，Whisper负责数据存储，Graphite-Web负责数据展示，可见 Graphite自身并不包含数据采集组件，但可以接入StatsD等开源数据采集组件来采集

数据，再传送给Carbom。

其中Carbon对写入的数据格式有一定的要求，比如如下所示：

servers.www01.cpuUsage 42 1286269200

products.snake-oil.salesPerMinute 123 1286269200

[one minute passes]

servers.www01.cpuUsageUser 44 1286269260

products.snake-oil.salesPerMinute 119 1286269260

其中 "serverswww01.cpuUsage 42 1286269200“ 是 "key"+空格分隔符+ "value+时间戳“ 的数据格式，"serverswww01.cpuUsage"是以“ . ”分割的key，代表具体的路径信息，“42”是具体的值，“1286269200是当前的Unix时间戳。

Graphite-Web对外提供了HTTP API可以查间某个key的数据以绘图展示，查询方式如下：

http://graphite.example.com/render?target=servers.www01.cpuUsage&

width=500&height=300&from=-24h

这个HTTP请求意思是查询key "serverswww01.cpuUsage"在过去24小时的数据并且要求 返回50\*300大小的数据图。

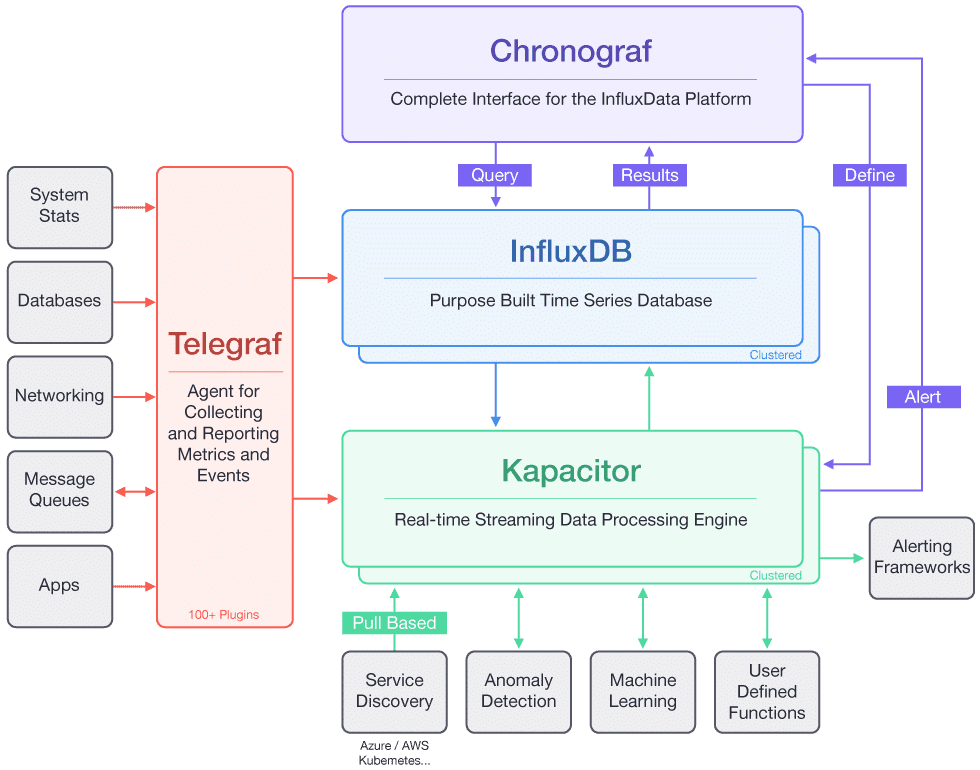
除此之外，Graphite-Web还支持丰富的函数，比如:

target=sumSeries(products.\*.salesPerMinute)

代表了查询匹配规则"products.\*.salesPerMinute"的所有key的数据之和。

## 15.3 TICK

TICK是Telegraf、InfluxDB、Chronograf、Kapacitor四个软件首字母的缩写，是由InfluxData 开发的一套开源监控工具栈。因此也叫作TICK Stack，它的架构可以看用下面这张图来描述。



（图片来源：<https://time.geekbang.org/column/article/39907>）

从这张图可以看出，其中Telegraf负责数据收集，InfluxDB负责数据存储，Chronograf负责数据展示。Kapacitor负责数据告警。

这里面，InfluxDB对写入的数据格式要求如下。

<measurement>[,<tag-key>=<tag-value>...] <field-key>=<field-value>[,<field2-key>=<field2-value>...] [unix-nano-timestamp]

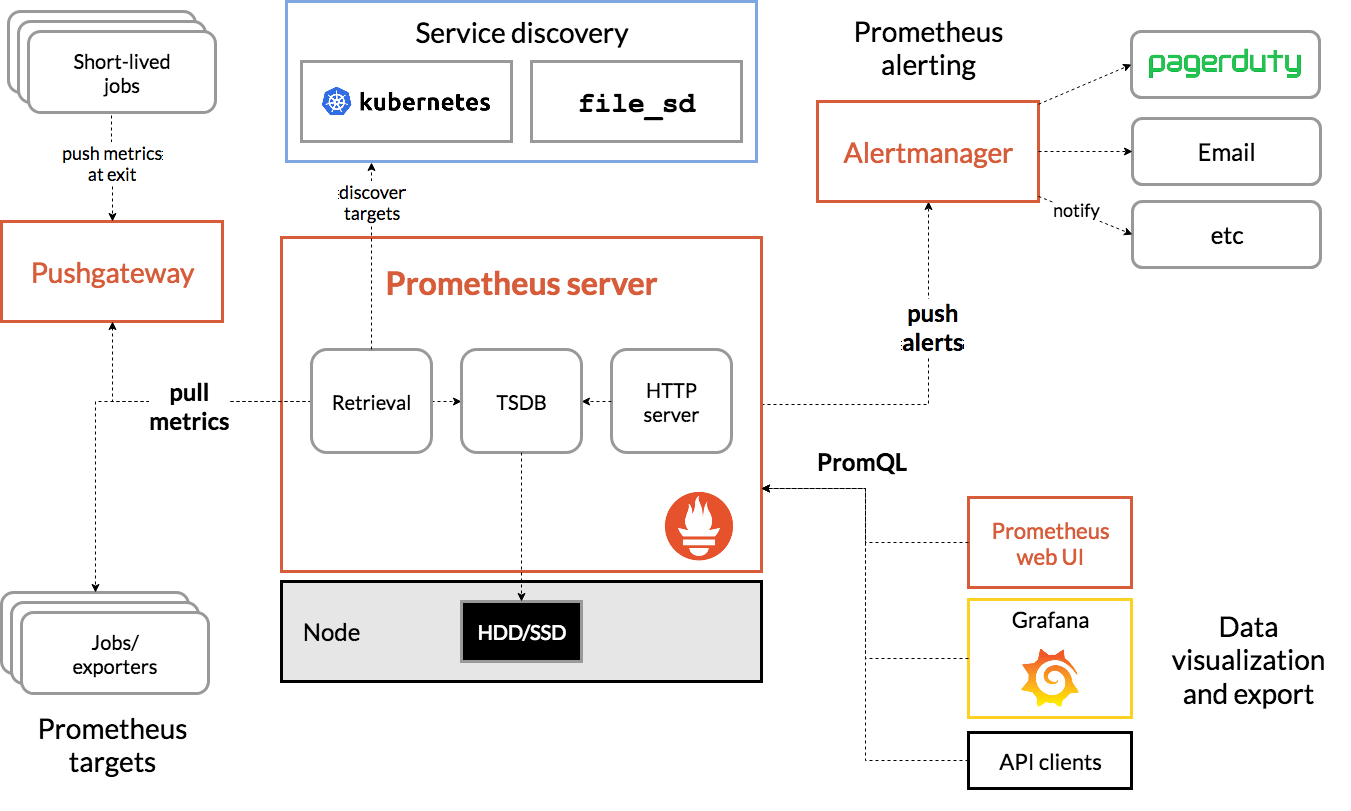
下面我用一个具体示例来说明它的格式：

cpu,host=serverA,region=us\_west value=0.64 1434067467100293230

其中，"cpu,host=serverA,region=us\_west value=0.64 1434067467100293230“代表了 host为serverA、region为us\_west的服务器CPU的值是0.64，时间是 1434067467100293230，时间精确到nano。

## 15.4 Prometheus

还有一种比较有名的时间序数据库解决方案Prometheus。它是一套开源的系统监控报警框架，受 Google的集群监控系统Borgmon启发，由工作在SoundCloud的Google前员工在2012年创建。后来作为社区开源项目进行开发，并于2015年正式发布，2016年正式加入CNCF（Cloud Native Computing Foundation），成为受欢迎程度仅次于Kubernetes的项目，它的架构可以用下图来描述。



（图片来源：<https://time.geekbang.org/column/article/39907>）

从这张图可以看出，Prometheus主要包含下面几个组件：

⦁ Prometheus Server：用于拉取metrics信息并将数据存储在时间序列数据库。

⦁ Jobs/exporters：用于暴露已有的第三方服务的metrics给Prometheus Server。比如 StatsD、Graphite等，负责数据收集 。

⦁ Pushgateway：主要用于短期jobs，由于这类jobs存在时间短，可能在Prometheus Server 来拉取metrics信息之前就消失了。所以这类的jobs可以直接向Prometheus Server推送它们的metrics信息。

⦁ Alertmanager：用于数据报警。

⦁ Prometheus web UI：负责数据展示。

它的工作流程大致是：

⦁ Prometheus Server定期从配置好的jobs或者exporters中拉取metrics信息，或者接收来自 Pushgateway发过来的metrics信息。

⦁ Prometheus Server把收集到的metrics信息存储到时间序列数据库中并运行已经定好的 alert.rules，向Alertmanager推送警报。

⦁ Alertmanager根据配置文件，对接收的警报处理，发出告警

⦁ 通过Prometheus web UI进行可视化展示。

Prometheus存储数据也是用的时间序列数据库。格式如下。

<metric name>{<label name>=<label value>, …}

比如下面这段代码代表了位于集群cluster1上，节点IP为1.1.1.1，端口为80，访问路径 为"/a"的http请求的总数为100。

http\_requests\_total{instance="1.1.1.1:80",job="cluster1",location="/a"} 100

讲到这里。四种监控系统的解决方案都已经介绍完了，接下来我们对比一下这些种方案 看看如何选型。

## 15.5选型对比

我们从监控系统的四个环节来分别对比

### 15.5.1数据收集

ELK是通过在每台服务器上部署Beats代理来采集数据；

Graphite本身没有收据采集组件，需要配合使用开源收据采集组件，比如StatsD；

TICK使用了Telegraf作为数据采集组件；

Prometheus通过Jobs/exporters组件来获取StatsD等采集过来的metrics信息。

### 15.5.2数据传输

ELK是Beats采集的数据传输给Logstash，经过Logstash清洗后再传输给Elasticsearch；

Graphite是通过第三方采集组件采集的数据，传输给Carbon；

TICK是Telegraf采集的数据，传输给InfluxDB；

Prometheus是Prometheus Server隔一段时间定期去从jobs/exporters拉取数据。

可见前三种都是采用“推数据”的方式而Prometheus是采取“拉数据”的方式，因此 Prometheus的解决方案对服务端的侵入最小。不要在服务端部署数据采集代理。

### 15.5.3数据处理

ELK可以对日志的任意字段索引，适合多维度的数据查询，在存储时间序列数据方面与时间序列数据库相比会有额外的性能和存储开销。除此之外，时间序列数据库的几种解决方案都支持多种功能的数据查询处理，功能也更强大。

⦁ Graphite通过Graphite-Web支持正则表达式匹配、sumSeries求和、alias给监控项重新命名等函数功能，同时还支持这些功能的组台，比如下面这个表达式的意思是，要查询所有匹配路径"stats.open.profile.\*.API.\_comments\_flow"的监控项之和，并且把监控项重今名为Total QPS

alias(sumSeries(stats.openapi.profile.\*.API.\_comments\_flow.total\_count,"Total QPS")

⦁ InfluxDB通过类似SQL语言的InfluxQL，能对监控数据进行复杂操作，比如查询一分钟CPU 的使用率，InfluxDB实现的示例是：

SELECT 100 - usage\_idel FROM "autogen"."cpu" WHERE time > now() - 1m and "cpu"='cpu0'

⦁ Prometheus通过私有的PromQL查询语句，如果要和上面InfluxDB实现同样的功能， PromQL语句如下看起来更加简洁。

100 - (node\_cpu{job="node",mode="idle"}[1m])

### 15.5.4数据展示

Graphite、TICK和Prometheus自带的展示功能都比较弱，界面也不好看，不过好在它们都支持 Grafana 来做数据展示。Grafana是一个开源的仪表盘工具，它支持多种数据源比如Graphite、 InfluxDB、Prometheus以及Elasticsearch等。ELK采用Kibana做数据展示，Kibana包含的数据展示功能比较强大，但只支持Elasticsearch，而且界面展示UI效果不如Grafana美观 。

## 15.6总结

以上几种监控系统实现方式，所使用的技术均未开源的，其中：

⦁ ELK的技术栈比较成熟，应用范围也比较广，除了可用作监控系统外，还可以用作日志查询和分析。

⦁ Graphite是基于时间序列数据库存储的监控系统，并且提供了功能强大的各种聚合函数比如 sum、

average、top5等可用于监控分析，而且对外提供了API也可以接入其他图形化监控系统如Grafana。

⦁ TICK的核心在于时间序列数据库InfluxDB的存储功能强大且支持类似SQL语言的复杂数据处理操作。

⦁ Prometheus的独特之处在于它采用了“拉数据”的方式，对业务影响较小，同时也采用了时间序列数据库存储，而且支持独有的PromQL查询语句，功能强大而且简洁。

从对实时性要求考虑，时间序列数据库的实时性要好于ELK，通常可以做到10s级别内的延迟，如果对实时性敏感的话，建议选择时间序列数据库解决方案。

从使用的灵活性角考虑，几种时间序列数据库的监控处理功能都要比ELK更加丰富，使用更灵活也更现代化。

所以如果要搭建一套新的监控系统。我建议可以考虑采用Graphite、TICK或者Prometheus其中之一。不过Graphite还要搭配数据采集系统比如StatsD或者Collectd使用，而且界面展示建议使用Grafana接入Graphite的数据源。它的效果要比Graphite Web本身提供的界面美观很多。TICK提供了完整的监控系统框架，包括从数据采集、数据传输、数据处理再到数据展示，不过在数据展示方面同样也建议用Grafana替换掉默认的数据展示组件Chronograf，这样展示效果更好。Prometheus因为采用 “拉数据”的方式，所以对业务的入侵性最小，比较适合Docker封装好的云原生应用，比如Kubernetes默认就采用了Prometheus作为监控系统。

# 如何搭建一套适合的服务追踪系统

专栏第8讲我给你讲了服务追踪系统的原理以及实现，简单回顾一下服务追踪系统的实现， 主要包括三个部分。

⦁ 埋点数据收集，负责在服务端进行埋点，来收集服务调用的上下文数据。

⦁ 实时数据处理，负责对收集到的链路信息，按照traceld和spanld进行串联和存储。

⦁ 数据链路展示，把处理后的服务调用数据，按照调用链的形式展示出来。

如果要自己从0开始实现一个服务追踪系统，针对以上三个部分你都必须有相应的解决方案。首先你需要在业务代码的框架层开发调用拦截程序，在调用的前后收集相关信息，把信息传输到一个统一的处理中心。然后处理中心要实时处理收集到链路信息，并按照traceld和spanld进行串联，处理完以后再存到合适的存储中。最后还要能把存储中存储的信息，以调用链路图或者调用拓扑图的形式对外展示。

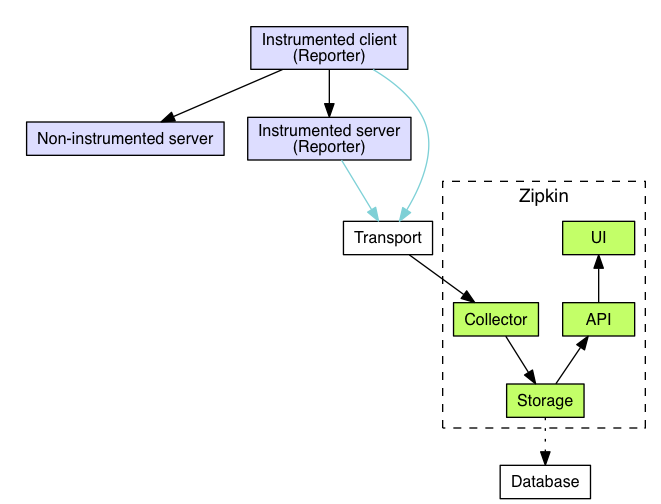
可以想象这个技术难度以及开发工作量都不小，对于大部分中小业务团队来说，都十分具有挑战。不过幸运的是，业界已经有不少开源的服务追踪系统实现，并且应用范围也已经十分广泛，对大部分的中小业务团队来说，足以满足对服务追踪系统的需求。

业界比较有名的服务追踪系统实现有阿里的鹰眼、Twitter开源的OpenZipkin，还有Naver开源的Pinpoint。它们都是受Google发布的Dapper论文启发而实现的。其中阿里的鹰眼解决方案没有开源，而且由于阿里要处理数据量比较大。所以鹰眼的定位相对定制化，不一定适台中小规模的业务团队。感兴趣的同学可以自行拓展学习。

下面我主要来介绍下开源实现方案OpenZipkin和Pinpoint，再看看它们有什么区别。

## 16.1、OpenZipkin

OpenZipkin是Twitter开源的服务踪系统，下面这张展示了它的架构设计。



（图片来源：<https://time.geekbang.org/column/article/40505>）

从图中看，OpenZipkin主要由四个核心部分组成

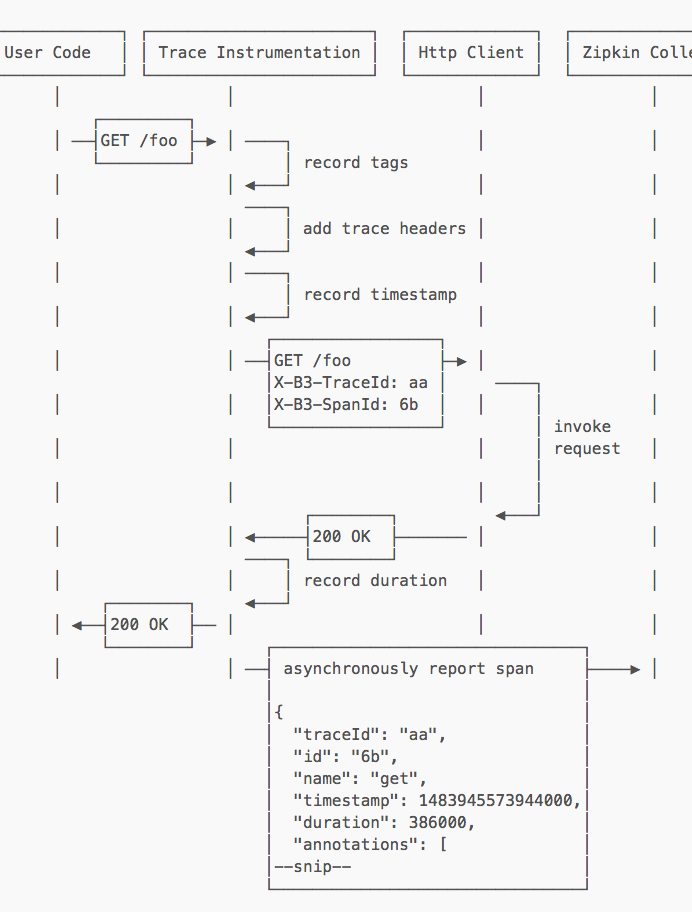
⦁ Collector：负责收集探针Reporter埋点采集的数据，经过验证处理并建立索引。

⦁ Storage：存储服务框架的链路数据，默认使用的是Cassandra，是因为Twitter内部大量使用了Cassandra，你也也可以替换成Elasticsearch或者MySQL。

⦁ API：将格式化和建立索引的链路数据以API的方式对外提供服务，比如被UI调用。

⦁ UI：以图形化的方式展示服务框架的链路数据。

它的工作原理可以用下面这张图来描述。

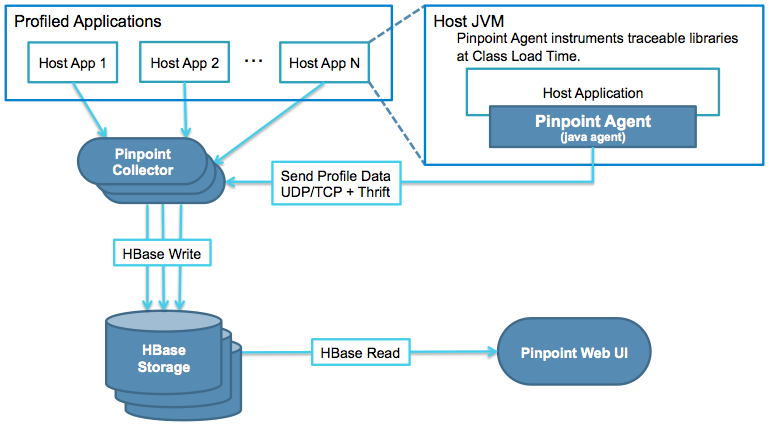


（图片来源：<https://time.geekbang.org/column/article/40505>）

具体流程是，通过在业务的HTTP Client前后引入服务追踪代码，这样在HTTP方法调用"/foo"前，生成trace信息：TraceId：aa、SpanId：6b、annotation：GET/foo，以及当前时刻的 timestamp：1483945573944000，然后调用结果返回后，记录下耗时duration之后，再把这些 trace信息和duration异步上传给Zipkin Collector。

## 16.2、Pinpoint

Pinpoint是Naver开源的一款深度支持Java语言的服务追踪系统，下面这张图是它的架构设计：



（图片来源：<https://time.geekbang.org/column/article/40505>）

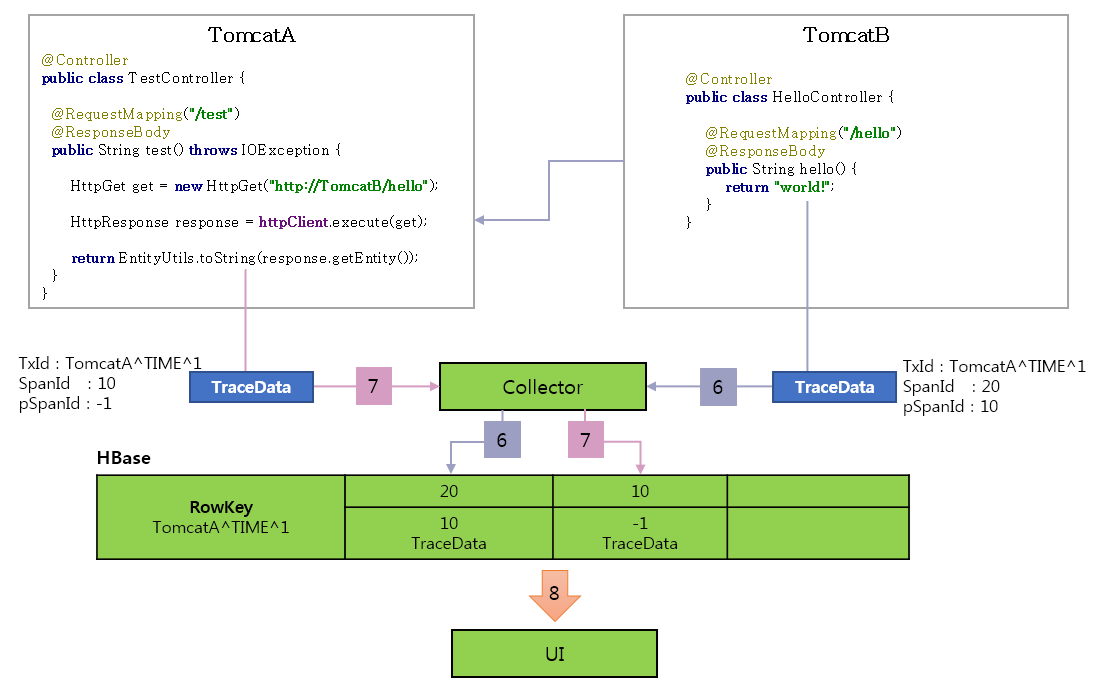
Pinpoint也由四个部分组成：

⦁ Pinpoint Agent：通过Java字节码注入的方式来收集JVM中的调用数据，通过UDP协议传递给Collector，数据采用Thrift协议进行编码。

⦁ Pinpoint Collector：收集Agent传过来的数据，然后写到HBase Storgage。

⦁ HBase Storage： 采用HBase集群存储服务调用的链路信息。

它的工作原理可以看这张图：



（图片来源：<https://time.geekbang.org/column/article/40505>）

具体来看。就是请求进入TomcatA，然后生成TraceId：TomcatA^TIME^1、SpanId：10、 pSpanId：-1〔代表是根请求〕，接着TomatA调用TomcatB的hello方法，TomcatB生成 TraceId：TomcatA^TIME^1、新的SpanId：20、pSpanId：10（代表是TomcatA的请求）。返回调用结果后将trace信息发给Collector，TomcatA收到调用结果后，将trace信息也发给Collector。Collector把trace信息写入到HBase中，Rowkey就是traceld，SpanId和 pSpanId都是列。然后就可以通过UI查询调用链路信息了。

## 16.3、选型对比

根据我的经验。考察服务追踪系统主要从下面这几个方面。

### 16.3.1埋点探针支持平台的广泛性

OpenZipkin和Pinpoint都支持哪些语言平台呢？

OpenZipkin提供了不同语言的Library，不同语言实现时要引入不同版本的Library。

官方提供了C#、Go、Java、JavaScript、Ruby、Scala、PHP等主流语言版本的Library，而且开源社区还提供了更丰富的不同语言版本的Library。详细的可以自行查看；而Pinpoint目前只支持Java语言。所以从探针支持的语言平台广泛性上来看，OpenZipkin比Pinpoint的使用范围要广，而且开源社区很活跃，生命力更强。

### 16.3.2系统集成难易程度

再来看下系统集成的难易程度。

以OpenZipkin的Java探针Brave为例，它只提供了基本的操作API，如果系统要想成 Brave，必须在配置里手动添加相应的配置文件并且加trace业务代码。具体来讲，就是你要先修改工程的POM依赖，以引入Brave相关的JAR包。

<dependencyManagement>

<dependencies>

<dependency>

<groupId>io.zipkin.brave</groupId>

<artifactId>brave-bom</artifactId>

<version>${brave.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

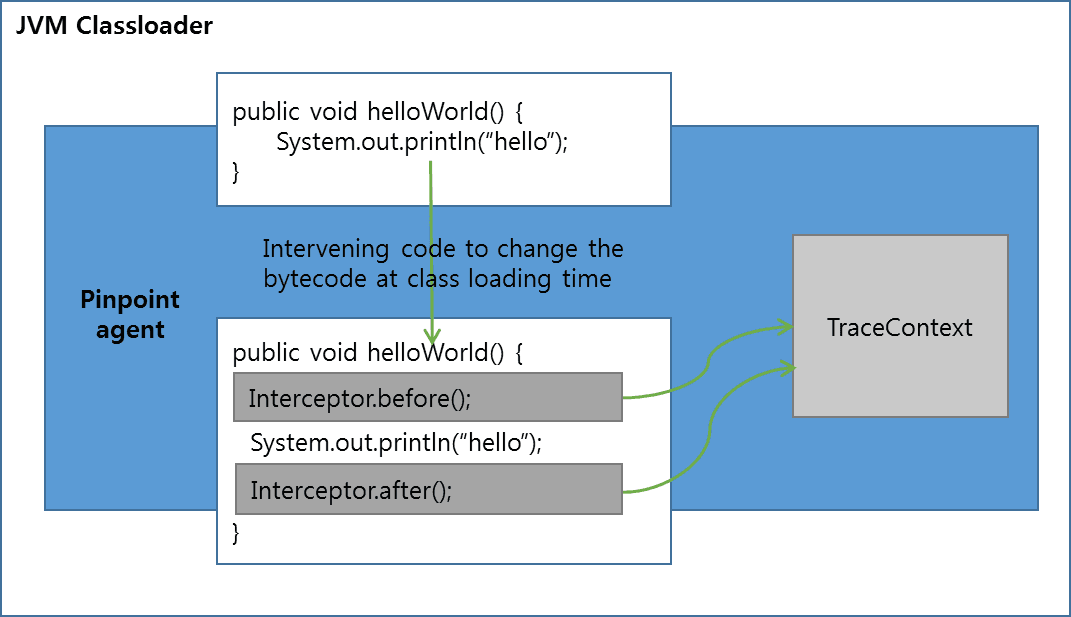
</dependencies>

</dependencyManagement>

然后假如你想收集一次HTTP调用的信息，你就可以使用Brave在Apache Httpclient基础上封装httpClient，它会记录每一次HTTP调用的信息，并上报给OpenZipkin。

httpclient =TracingHttpClientBuilder.create(tracing).build();

而Pinpoint是通过字节码注入的方式来实现拦截服务调用，从而收集trace信息的，所以不要代码做任何改动。Java字节码注入的大致原理你可以参考下图。



（图片来源：<https://time.geekbang.org/column/article/40505>）

我来解释一下。就是JVM在加载class二进制文件时，动态地修改加载的class文件，在方法的前后执行拦截器的before()和after()方法，在before()和after()方法里记录trace()信息。而应用不需要修改业务代码，只要在JVM启动时，添加类似下面的启动参数就可以了。

-javaagent:$AGENT\_PATH/pinpoint-bootstrap-$VERSION.jar

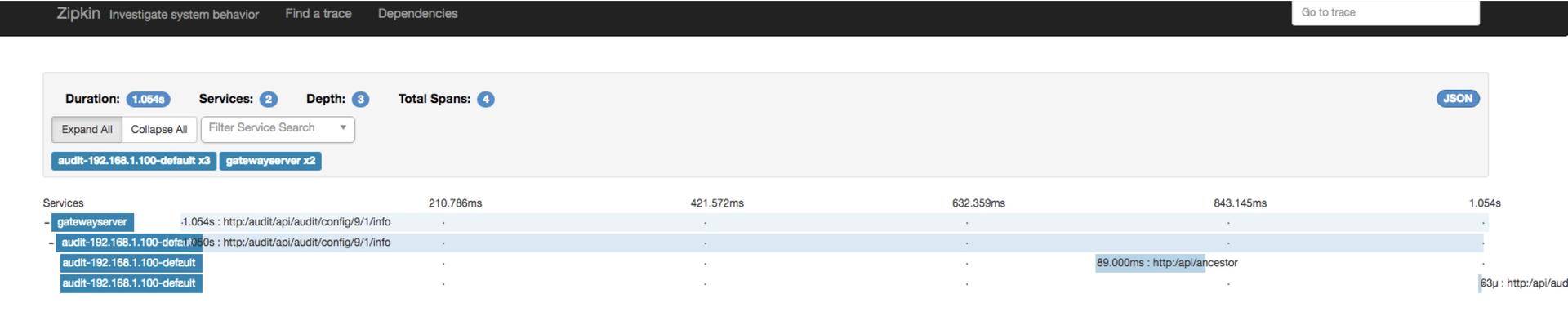
-Dpinpoint.agentId=<Agent's UniqueId>

-Dpinpoint.applicationName=<The name indicating a same service (AgentId collection)

所以从系统集成难易程度上看，Pinpoint要比OpenZipkin简单。

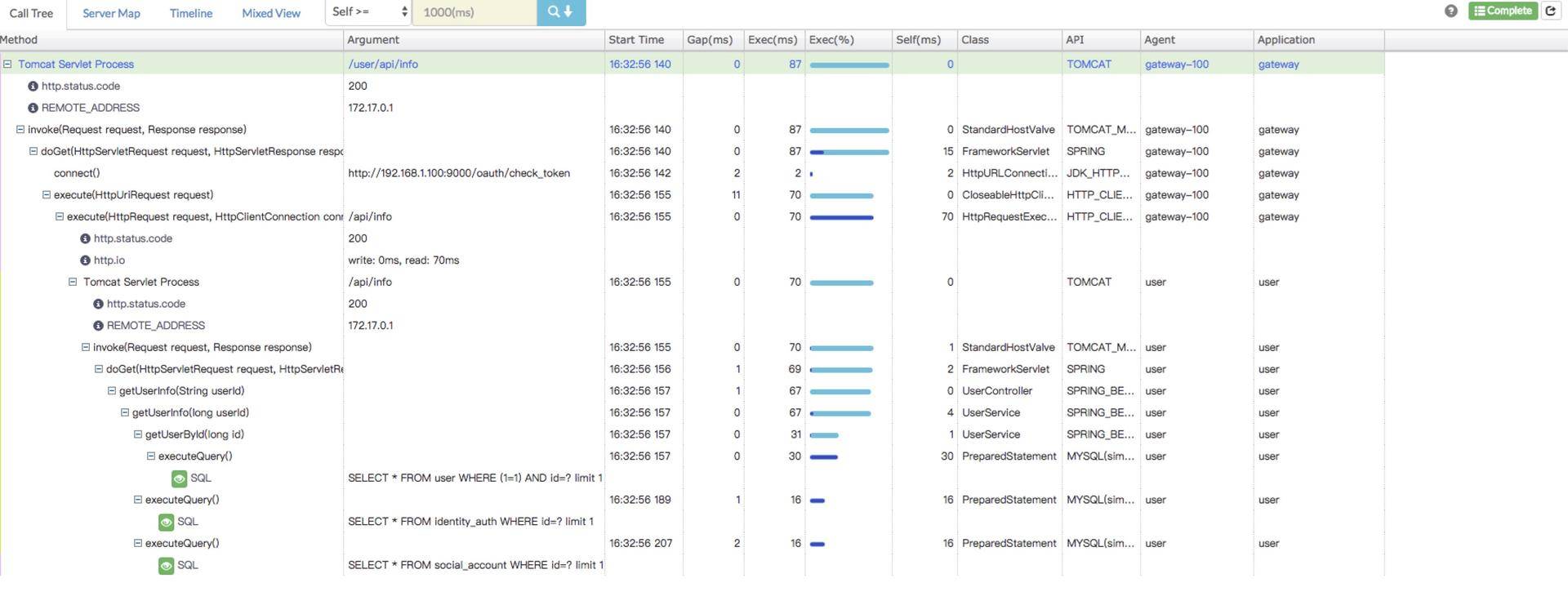
### 16.3.3调用链路数据的精确度

从下面这张OpenZipkin的调用链路图可以看出，OpenZipkin收集到的数据只到接口级别，进一步的信息就没有了。

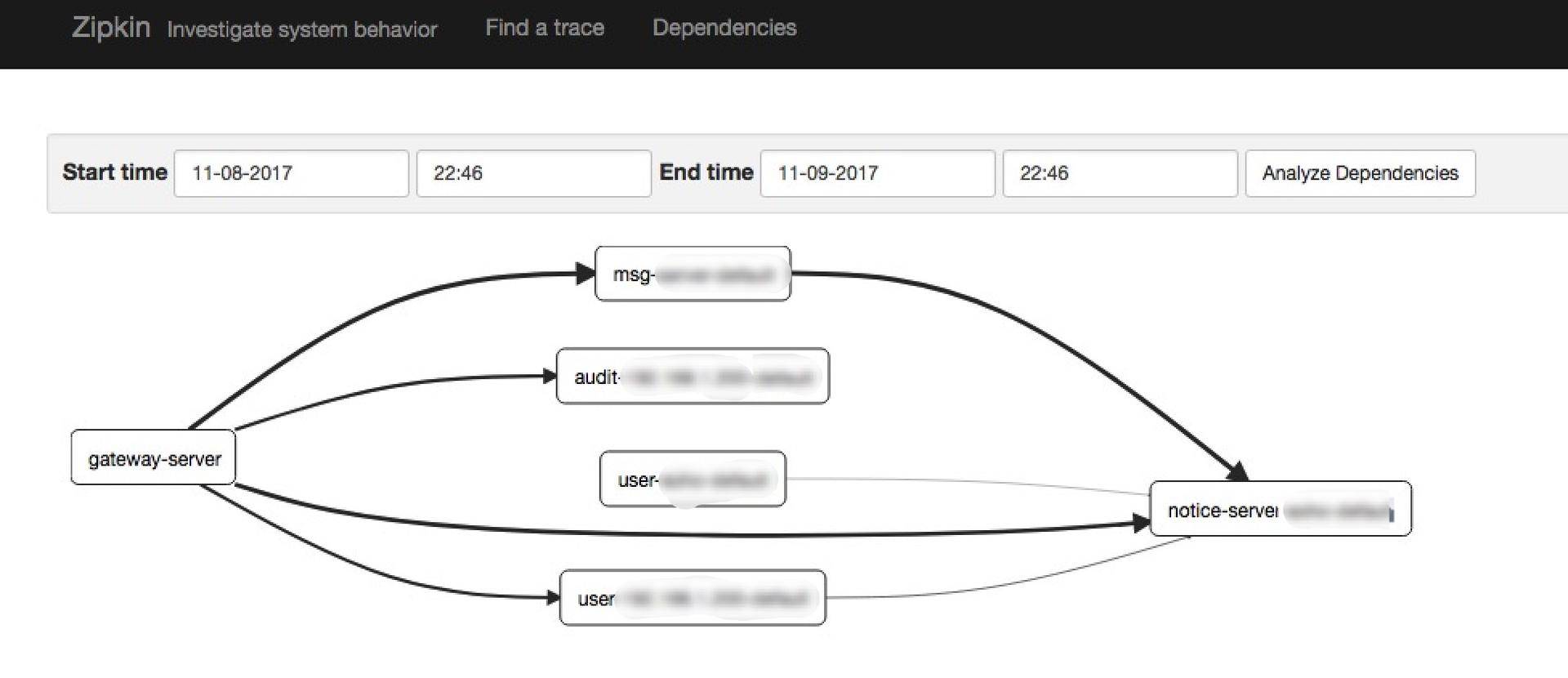


（图片来源：<https://time.geekbang.org/column/article/40505>）

再来看下Pinpoint。因为Pinpoint采用了字节码注入的方式实现trace信息收集，所以它能拿到的信息比OpenZipkin多得多。从下面这张图可以看出，它不仅能够查看接口级别的链路调用信息，还能深入到调用所关联的数据库信息。

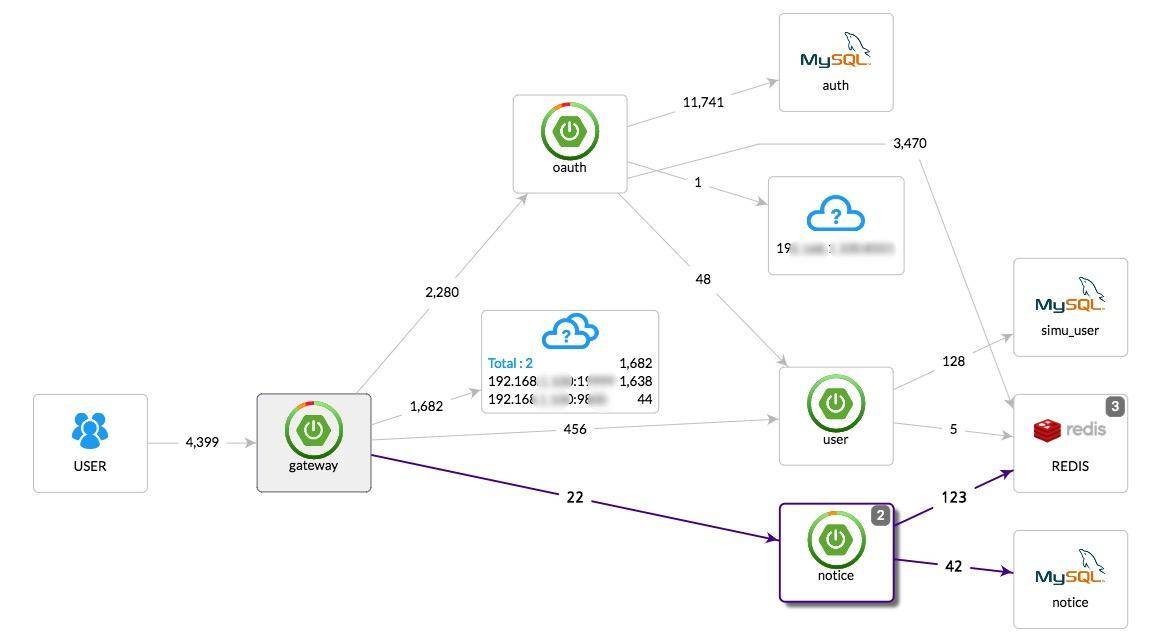


同理在绘制链路拓扑图时。OpenZipkin只能绘制服务与服务之间的调用链路拓扑图，比如下面这张示意图。



（图片来源：<https://time.geekbang.org/column/article/40505>）

而Pinpoint不仅能够绘制服务与服务之间，还能绘制与DB之间的调用链路拓扑图，比如下图：



所以，从调用链路数据的精确上看，Pinpoint要比OpenZipkin精确得多。

## 16.4、总结

今天我给你讲解了两个开源服务追踪系统OpenZipkin和Pinpoint的具体实现，并从埋点探针支持平台广泛性、系统集成难易程度、调用链路数据精确三个方面对它们进行了对比。

从选型的角度来讲，如果你的业务采用的是Java语言，那么采用Pinpoint是个不错的选择，因为它不要业务改动一行代码就可以实现trace信息的收集。除此之外，Pinpoint不仅能看到服务与服务之间的链路调用，还能看到服务内部与资源层的链路调用，功能更为强大，如果你有这方面的需求，Pinpoint正好能满足。

如果你的业务不是Java语言实现，或者采用了多种语言，那毫无疑问应该选择OpenZipkin。并且，由于其开源社区很活跃，基本上各种语言平台都能找到对应的解决方案。不过想要使用OpenZipkin，还要做一些额外的代码开发工作，以引入OpenZipkin提供的Library到你的系统中。

除了OpenZipkin和Pinpoint，业界还有其他开源追踪系统实现，比如Uber开源的Jaeger，以及国内的一款开源服务追踪系统SkyWalking。不过由于目前应用范围不是很广，这里就不详细介绍了，感兴趣的同学可以点击"拓展阅读"自行学习。

## 16.5、拓展阅读

阿里巴巴鹰眼：https://ppt.geekbang.org/slide/download/939/595f4cdcb9d52.pdf/18

Jaeger: https://www.jaegertracing.io/

SkyWalking: <https://github.com/apache/incubator-skywalking>

# 如何识别服务节点是否存活

今天我要与你分享如何识别服务节点是否存活，这在服务治理中是十分重要的。在进入正题之前，你可以先复习一下专栏第5期，我在讲解注册中心原理的时候，以开源注册中心ZooKeeper为例，描述了它是如何管理注册到注册中心的节点的存活的。

其实ZooKeeper判断注册中心节点存活的机制其实就是注册中心摘除机制，服务提供者以注册中心中的数据为准，当服务端节点有变更时注册中心就会把变更通知给服务消费者，服务消费者就会调用注册中心来拉取最新的节点信息。

这种机制在大部分情况下都可以工作得很好，但是在网络频繁抖动时，服务提供者向注册中心汇报心跳信息可能会失败，如果在规定的时间内，注册中心都没有收到服务提供者的心跳信息，就会把这个节点从可用节点列表中移除。更糟糕的是，在服务池有上百个节点的的时候，每个节点都可能会被移除，导致注册中心可用节点的状态一直在变化，这个时候应该如何处理呢?

下面就结合我在实践中的经验，给你讲解几种解决方案。

## 17.1心跳开关保护机制

在网络频繁抖动的情况下，注册中心中可用的节会不断变化，这时候服务消费者会频繁收到服务提供者节点变更的信息于是就不断地请求注册中心来拉取新的可用服务节信息。当有成百上千个服务消费者，同时请求注册中心获取新的服务提供者的节点信息时可能会把注册中心的带宽给占满，尤其是注册中心是百兆网卡的情况下。

所以针对这种情况，需要一种保护机制，即使在网络频繁抖动的时候，服务消费者也不至于同时去请求注册中心获取新的服务节点信息。

我曾经就遇到过这种情况。一个可行的解决方案就是给注册中心设置一个开关，当开关打开时，即使网络频繁抖动，注册中心也不会通知所有的服务消费者有服务节点信息变更，比如只给10％的服务消费者返回变更，这样的话就能将注册中心的请求量减少到原来的1/10。

当然打开这个开关也是有一定代价的，它会导致服务消费者感知新的服务节点信息延迟，原先可能在10s内就能感知到服务提供者节点信息的变更，现在可能会延迟到几分钟，所以在网络正常的情况下，开关并 不适合打开；可以作为一个紧急措施，在网络频繁抖动的时候，才打开这个开关。

## 17.2服务节点摘除保护机制

服务提供者在进程启动时，会注册服务到注册中心，并每隔一段时间，汇报心跳给注册中心，以标识自己的存活状态。如果隔了一段固定时间后，服务提供者仍然没有汇报心跳给注册中心，注册中心就会认为该节中已经处于"dead"状态，于是从服务的可用节点信息中移除出去。

如果遇到网络问题，大批服务提供者节点汇报给注册中心的心跳信息都可能会传达失败，注册中心就会把它们都从可用节点列表中移除出去。造成剩下的可用节点难以承受所有的调用。引起“雪崩”。但是这种情况下，可能大部分服务提供者节点是可用的，仅仅因为网络原因无法汇报心跳给注册中心就被"无情"的摘除了。

这个时候就要根据实际业务的情况，设定一个阈值比例，即使遇到刚才说的这种情况，注册中心也不能摘除超过这个阈值比例的节点。

这个阈值比例可以根据实际业务的冗余来确定，我通常会把这个比例设定在20％，就是说注册中心不能摘除超过20％的节点。因为大部分情况下，节点的变化不会这么频繁，只有在网络抖动或者业务明确要下线大批量节点的情况下才有可能发生。而业务明确要下线大批量节点的情况是可以预知的，这种情况下可以关闭阈值保护；而正常情况下，应该打开阈值保护，以防止网络抖动时 大批量可用的服务节点被摘除。

讲到这里，我们先小结一下。

心跳开关保护机制，是为了防止服务提供者节点频繁变更导致的服务消费者同时去注册中心获取最新服务节点信息。服务节中摘除保护机制，是为了防止服务提供者节中被大量摘除引起服务消费者可以调用的节点不足。

可见，无论是心跳开关保护机制还是服务节摘除保护机制，都是因为注册中心里的节中信息是随时可能发生变化的，所以也可以把注册中心叫作动态注册中心。

那么是不是可以换个路。服务消费者并不严格以注册中心中的服务节中信息为准，而是更多的以服务消费者实际调用信息来判断服务提供者节点是否可用。这就是下面我要讲的静态注册中心。

## 17.3静态注册中心

前面讲过心跳机制能保证在服务提供者出现异常时，注册中心可以及时把不可用的服务提供者从可用节点列表中移除出去，正常情况下这是个很好的机制。但是仔细思考一下，为什么不把这种心跳机制直接用在服务消費者这一端呢 ?

因为服务提供者是向服务消费者提供服务的，是否可用服务消费者应该比注册中心更清楚，因此可以直接在服务消费者端，根据调用服务提供者是否成功来判定服务提供者是否可用。如果服务消费者调用某一个服务提供者节点连续失败超过一定次数，可以在本地内存中将这个节点标记为不可用，并且每隔一段固定时间，服务消费者都要向标记为不可用的节点发起保活探测，如果探测成功了，就将标记为不可用的节点再恢复为可用状态，重新发起调用。

这样的话，服务提供者节点就不要向注册中心汇报心跳信息，注册中心中的服务节点信息也不会动态变化，也可以称之为静态注册中心。

从我的实践经历来看，一开始采用了动态注册中心，后来考虑到网络的复杂性，心跳机制不一定是可靠的，而后开始改为采用服务消费者端的保活机制，事实证明这种机制足以应对网络频繁抖动等复杂的场景。

当然静态注册中心中的服务节点信息并不是一直不变，当在业务上线或者运维人工增加或者删除服务节点这种预先感知的情况下，还是有必要去修改注册中心中的服务节点信息。

比如在业务上线过程中，要把正在部著的服务节点从注册中心中移除，等到服务部署完毕，完全可用的时候，再加入到注册中心。还有就是在业务新增或者下线服务节点的时候，需要调用注册中心提供的接口，添加节点信息或者删除节点。这个时候静态注册中心有点退化到配置中心的意思，只不过这个时候配置中心里存储的不是某一项配置，而是某个服务的可用节点信息。

## 17.3总结

今天我给你讲解了动态注册中心在实际线上业务运行时，如果遇到网络不可靠等因素，可能会带来的两个问题，一个是服务消费者同时并发访问注册中心获取新服务信息导致注册中心带宽被打满；另一个是服务提供者节点被大量摘除导致服务消费者没有足够的节点可以调用。

这两个问题都是我在业务实践过程中遇到过的，我给出的两个解决方案：心跳开关保护机制和服务节点摘除保护机制都是在实践中应用过的，并且被证明是行之有效的。

而静态注册中心的思路，是在斟酌注册中心的本质之后，引入的另外一个解决方案，相比于动态注册中心更加简单，并且基于服务消费者本身调用来判断服务节点是否可用，更加直接也更加准确，尤其在注册中心或者网络出现问题的时候，这种方案基本不受影响。

# 如何使用负载均衡算法

假设你订阅了一个别人的服务，从注册中心查询得到了这个服务的可用节点列表，而这个列表里包含了几十个节点，这个时候你该选择哪个节发起调用呢? 这就是今天我要给你讲解的关于客户端负载均衡算法的问题。

为什么要引入负载均衡法呢？主要有两个原因：一个是要考虑调用的均匀性，也就是要让个节点都接收到调用，发挥所有节点的作用；另一个是要考虑调用的性能，也就是哪个节点响应最快，优先调用哪个节点。

不同的负载均衡算法在这两个方面的考虑不同，下面我就来给你介绍常见的负载均衡算法以及其应用场景。

常见的负载均衡算法

### 18.1、 随机算法

随机算法，顾名思义就是从可用的服务节点中，随机挑选一个节点来访问。

在实现时，随机算法通常是通过生成一个随机数来实现，比如服务有10个节点，那么就每次生成一个1-10之间的随机数，假设生成的是2，那么就访问编号为2的节点。

采用随机算法，在节点数量足够多，并且访问量比较大的情况下，各个节点被访问的概率是基本相同的。一个随机算法的代码实现，可以参考这个示例。

### 18.2、 轮询算法

轮询算法，顾名思义就是按照固定的顺序，把可用的服务节点，挨个访问一次。

在实现时，轮询算法通常是把所有可用节点放到一个数组里，然后按照数组编号，挨个访问。比如服务有10个节点，放到数组里就是一个大小为10的数组，这样的话就可以从序号为0的节点开始访问，访问后序号自动加1，下一次就会访问序号为1的节点，以此类推。

轮询算法能够保证所有节点被访问到的概率是相同的。一个轮询算法的代码实现，可以参考这个示例。

### 18.3、 加权轮询算法

轮询算法能够保证所有节点被访问的概率相同，而加权轮间法是在此基础上，给每个节点赋予一个权重，从而使个节点被访问到的概率不同，权重大的节点被访问的概率就高，权重小的节点被访问的概率就小。

在实现时，加权轮询算法是生成一个节点序列，该序列里有n个节点，n是所有节点的权重之和。在这个序列中，每个节点出现的次数，就是它的杈重值。比如有三个节点：a、b、c，杈重分别是 3、2、1，那么生成的序列就是{a、a、b、c、b、a}，这样的话按照这个序列访问，前6次请求就会分别访问节点a三次，节点b两次，节点c一次。从第7个请求开始，又重新按照这个序列的顺序来访问节点。

在应用加权轮询法的时候，根据我的经验，要尽可能保证生产的序列的均匀，如果生成的不均匀会造成访问失衡，比如刚才的例子，如果生成的序列是{a、a、a、b、b、c}，就会导致前3次访问的节点都是a。一个加权轮询算法的代码实现，可以参考这个示例。

### 18.4、 最少活跃连接算法

最少活跃连接法，顾名思义就是一次访问都选择连接数少的节点。因为不同节点处理请求的速度不同，使得同一个服务消费者同一个节点的连接数都不相同。连接数大的节点，可以认为是处理请求慢，而连接数小的节点，可以认为是处理请求快。所以在挑选节点时，可以以连接数为依据，选择连接数少的节点访问。

在实现时，需要记录跟每一个节点的连接数，这样在选择节点时，才能比较出连接数最小的节点。 一个最少活跃连接算法的代码实现。可以参考这个示例。

### 18.5、 一致性hash算法

一致性hash法，是通过某个hash函数，把同一个来源的请求都映射到同一个节点上。一致性 hash算法最大的特点就是同一个来源的请求，只会映射到同一个节点上，可以说是具有记忆功能。只有当这个节点不可用时，请求才会被分配到相邻的可用节点上。

一个一致性hash算法的代码实现，可以参考这个示例。

### 18.6、 负载均衡算法的使用场景

上面这五种负载均衡算法，具体在业务中该如何选择呢？根据我的经验，它们的各自应用场景如下：

⦁ 随机算法：实现比较简单，在请求量远超可用服务节点数量的情况下，各个服务节点被访问的概率基本相同，主要应用在各个服务节点的性能差异不大的情况下。

⦁ 轮询算法：跟随机算法类似，各个服务节点被访问的概率也基本相同，也主要应用在各个服务节点性能差异不大的情况下。

⦁ 加权轮询法：在轮询算法基础上的改进，可以通过给每个节点设不同的权重来控制访问的概率，因此主要被用在服务节点性能差异比较大的情况。比如经常会出现一种情况，因为采购时间的不同，新的服务节点的性能往往要高于旧的节点，这个时候可以给新的节点设更高的权重，让它承担更多的请求，充分发挥新节点的性能优势。

⦁ 最少活跃连接法：与加权轮询算法预先定义好每个节点的访问权重不同，采用最少活跃连接算法，客户端同服务端节点的连接数是在时刻变化的，理论上连接数越少代表此时服务端节点越空闲，选择空闲的节点发起求，能获取更快的响应速度。尤其在服务端节点性能差异较大，而又不好做到预先定义权重时，采用最少活跃连接法是比较好的选择。

⦁ 一致性hash算法：因为它能够保证同一个客户端的请求始终访问同一个服务节点，所以适合服务端节点处理不同客户端请求差异较大的场景。比如服务端缓存里保存着客户端的求结果，如果一客户端一直访问一个服务节点，那么就可以一直从缓存中获取数据。

这五种负载均衡算法是业界最常用的，不光在RPC调用中被广泛采用，在一些负载均衡组件比如 Nginx中也有应用，所以说是一种通用的负载均衡算法，但是不是所有的业务场景都能很好解决呢？

我曾遇到过这种场景：

⦁ 服务节点数量众多，且性能差异比较大；

⦁ 服务节点列表经常发生变化，增加节点或者减少节点时有发生；

⦁ 客户端和服务节点之间的网络情况比较复杂，有些在一个数据中心，有些不在一个数据中心需要跨网访问，而且网络经常延迟或者抖动。

显然无论是随机算法还是轮询算法，第一个情况就不满足，加权轮询算法要预先配服务节点的权重。在节点列表经常变化的情况下不好维护，所以也不适合。而最少活跃连接法是从客户端自身维度去判断的，在实际应用时，并不能直接反映出服务节点的请求大小，尤其是在网络情况比较复杂的情况下，并不能做到动态的把请求发送给最适合的服务节点。至于一致性hash法，显然不适合这种场景。

针对上面这种场景有一种算法更加适合，这种算法就是自适应最优算法。

### 18.7、 自适应最优选择算法

这种算法的主要思路是在客户端本地维护一份同每一个服务节点的性能统计快照，并且每隔一段时间去更新这个快照。在发起请求时，根据"二八原则"，把服务节点分成两部分。找出20％的那部分响应最慢的节点，然后降低权重。这样的话，客户端就能够实时的根据自身访问每个节点性能的快慢，动态调整访问最慢的那些节点的权重，来减少访问量，从而可以优化长尾请求。

由此可见，自适应最优选择算法是对加权轮询算法的改良，可以看作是一种动态加权轮询算法。它的实现关踺之处就在于两点：第一点是每隔一段时间获取客户端同每个服务节点之间调用的平均性能统计；第二点是按照这个性能统计对服务节点进行排序，对排在性能倒数20％的那部分节点赋予一个较低的权重，其余的节点赋予正常的权重。

在具体实现时，针对第一点，要在内存中开辟一块空间记录客户端同每一个服务节点之间调用的平均性能，并每隔一段固定时间去更新。这个更新的时间间隔不能太短，太短的话很容易受瞬时的性能抖动影响，导致统计变化太快，没有参考性；同时也不能太长，太长的话时效性就会大打折扣，效果不佳。根据我的经验，1分钟的更新时间间隔是个比较合适的值。

针对第二点，关键点是权重值的设定，即使服务节点之间的性能差异较大，也不适合把权重设置得差异太大，这样会致性能较好的节点与性能较差的节点之间调用量相差太大，这样也不是一种合理的状态。在实际设定时，可以设置20％性能较差的节点权重为3，其余节点权重为5。

### 18.8、 总结

今天我给你讲解了最常用的五种客户端负载均衡算法的原理以及适用场景，在业务实践的过程中，究竟采用哪种，要根据实际情况来决定，并不是算法越复杂越好。

比如在一种简单的业务场景下，有10个服务节点，并且配置基本相同，位于同一个数据中心，此时客户端选择随机算法或者轮询算法既简单又高效，并没有必要选择加杈轮询算法或者最少活跃连接算法。

但在遇到前面提到的那种复杂业务场景下，服务节点数量众多，配置差异比较大，而且位于不同的数据中心，客户端与服务节点之间的网络情况也比较复杂，这个时候简单的负载均衡算法通常都难以应对，需要针对实际情况，选择更有针对性的负载均衡算法，比如自适应最优选择算法。

# 如何使用服务路由