Nginx (http://dockone.io/topic/Nginx)

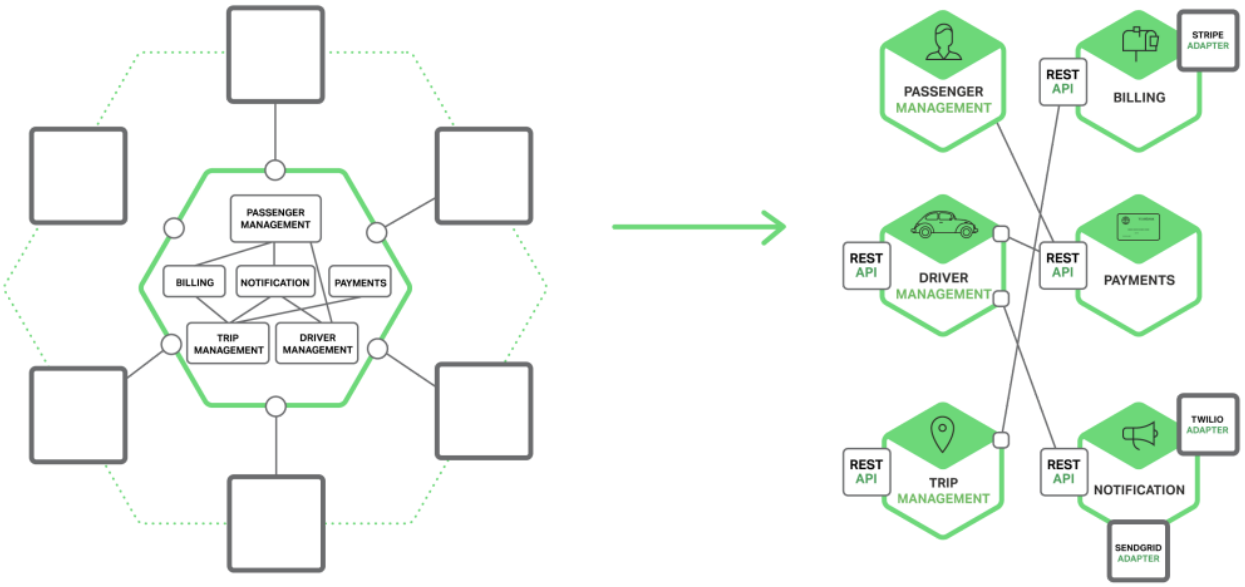
微服务 (http://dockone.io/topic/%E5%BE%AE%E6%9C%8D%E5%8A%A1)

微服务实战（三）：深入微服务架构的进程间通信

【编者的话】这是采用微服务架构创建自己应用系列第三篇文章。第一篇 (http://dockone.io/article/394)介绍了微服务架构模式，和单体式模式进行了比较，并且讨论了使用微服务架构的优缺点。第二篇 (http://dockone.io/article/482)描述了采用微服务架构应用客户端之间如何采用API Gateway方式进行通信。在这篇文章中，我们将讨论系统服务之间如何通信。

简介

在单体式应用中，各个模块之间的调用是通过编程语言级别的方法或者函数来实现的。但是一个基于微服务的分布式应用是运行在多台机器上的。一般来说，每个服务实例都是一个进程。因此，如下图所示，服务之间的交互必须通过进程间通信（IPC）来实现。



(http://dockerone.com/uploads/article/20150802/1fc9e63825109680c7138b786c6e1c24.png)

后面我们将会详细介绍IPC技术，现在我们先来看下设计相关的问题。

交互模式

当为某一个服务选择IPC时，首先需要考虑服务之间如何交互。客户端和服务器之间有很多的交互

模式，我们可以从两个维度进行归类。第一个维度是一对一还是一对多：

• **一对一**：每个客户端请求有一个服务实例来响应。
• **一对多**：每个客户端请求有多个服务实例来响应

第二个维度是这些交互式同步还是异步：

• 同步模式：客户端请求需要服务端即时响应，甚至可能由于等待而阻塞。
• 异步模式：客户端请求不会阻塞进程，服务端的响应可以是非即时的。

下表显示了不同交互模式：

	One-to-One	**One-to-Many**
Synchronous	Request/response	—
Asynchronous	Notification	Publish/subscribe
	Request/async response	Publish/async responses

(http://dockerone.com/uploads/article/20150802/ab63cad76d0743e8a77c423eda919a24.jpg)

一对一的交互模式有以下几种方式：

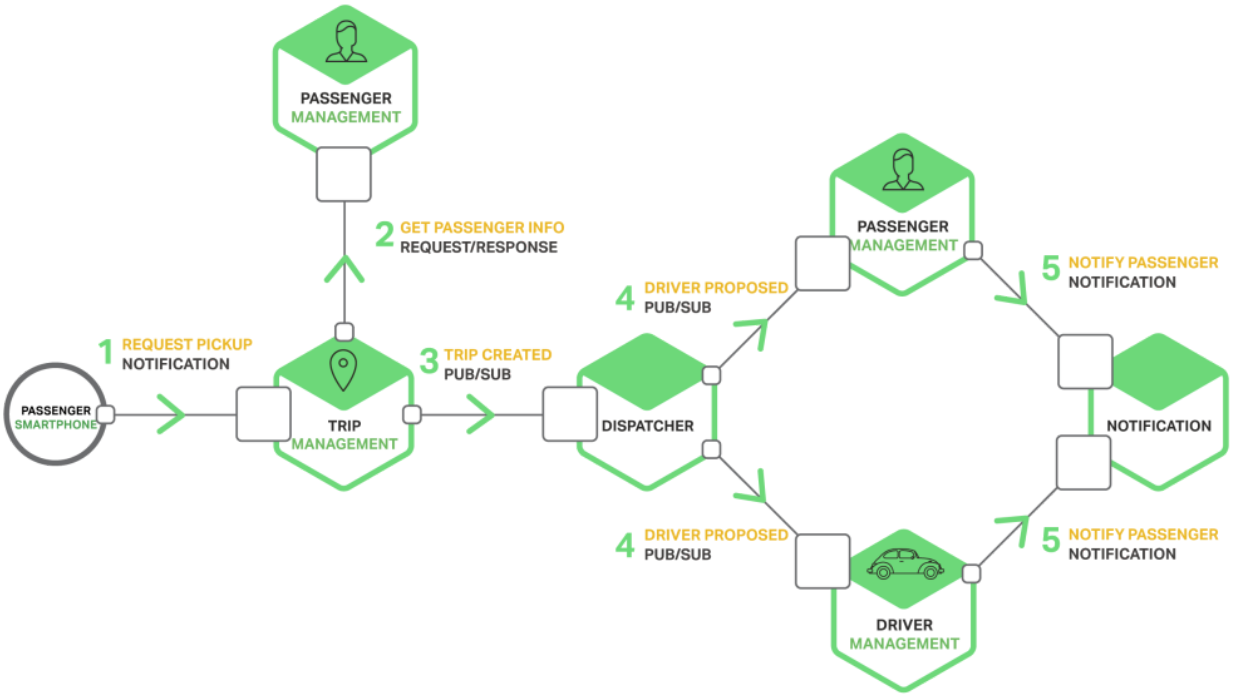
• 请求/响应：一个客户端向服务器端发起请求，等待响应。客户端期望此响应即时到达。在一个基于线程的应用中，等待过程可能造成线程阻塞。
• 通知（也就是常说的单向请求）：一个客户端请求发送到服务端，但是并不期望服务端响应。
• 请求/异步响应：客户端发送请求到服务端，服务端异步响应请求。客户端不会阻塞，而且被设计成默认响应不会立刻到达。

一对多的交互模式有以下几种方式：

• 发布/ 订阅模式：客户端发布通知消息，被零个或者多个感兴趣的服务消费。

• 发布/异步响应模式：客户端发布请求消息，然后等待从感兴趣服务发回的响应。

每个服务都是以上这些模式的组合，对某些服务，一个IPC机制就足够了；而对另外一些服务则需要多种IPC机制组合。下图展示了在一个打车服务请求中服务之间是如何通信的。



(http://dockerone.com/uploads/article/20150802/6c6cc8293d2a8334327902edcd32167d.png)

上图中的服务通信使用了通知、请求/响应、发布/订阅等方式。例如，乘客通过移动端给『行程管理服务』发送通知，希望申请一次出租服务。『行程管理服务』发送请求/响应消息给『乘客服务』以确认乘客账号是有效的。紧接着创建此次行程，并用发布/订阅交互模式通知其他服务，包括定位可用司机的调度服务。

现在我们了解了交互模式，接下来我们一起来看看如何定义API。

定义API

API是服务端和客户端之间的契约。不管选择了什么样的IPC机制，重要的是使用某种交互式定义语言（IDL）来精确定义一个服务的API。甚至有一些关于使用API first的方法(http://www.programmableweb.com/news/how-to-design-great-apis-api-first-design-and-raml/how-to/2015/07/10)（API-first approach）来定义服务的很好的理由。在开发之前，你需要先定义服务的接口，并与客户端开发者详细讨论确认。这样的讨论和设计会大幅度提到API的可用度以及满意度。

在本文后半部分你将会看到，API定义实质上依赖于选择哪种IPC。如果使用消息机制，API则由消息频道（channel）和消息类型构成；如果选择使用HTTP机制，API则由URL和请求、响应格式构成。后面将会详细描述IDL。

API的演化

服务端API会不断变化。在一个单体式应用中经常会直接修改API，然后更新给所有的调用者。而在基于微服务架构应用中，这很困难，即使只有一个服务使用这个API，不可能强迫用户跟服务端保持同步更新。另外，开发者可能会尝试性的部署新版本的服务(http://techblog.netflix.com/2013/08/deploying-netflix-api.html)，这个时候，新旧服务就会同事运行。你需要知道如何处理这些问题。

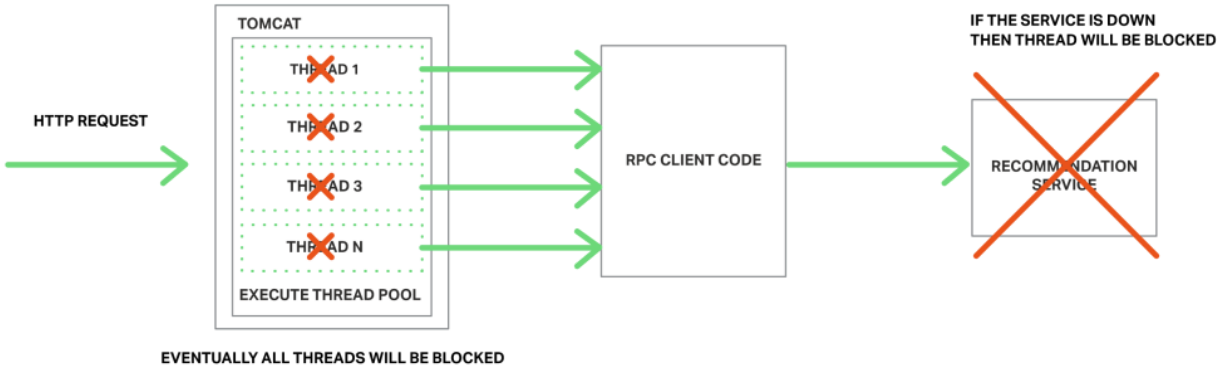
你如何处理API变化，这依赖于这些变化有多大。某些改变是微小的，并且可以和之前版本兼容。比如，你可能只是为某个请求和响应添加了一个属性。设计客户端和服务端时候应该遵循健壮性原理 (https://en.wikipedia.org/wiki/Robustness_principle)，这很重要。客户端使用旧版API应该也能和新版本一起工作。服务端仍然提供默认响应值，客户端忽略此版本不需要的响应。使用IPC机制和消息格式对于API演化很有帮助。

但是有时候，API需要进行大规模的改动，并且可能与之前版本不兼容。因为你不可能强制让所有的客户端立即升级，所以支持老版本客户端的服务还需要再运行一段时间。如果你正在使用基于基于HTTP机制的IPC，例如REST，一种解决方案是把版本号嵌入到URL中。每个服务都可能同时处理多个版本的API。或者，你可以部署多个实例，每个实例负责处理一个版本的请求。

处理部分失败

在上一篇关于API gateway (http://dockone.io/article/482)的文章中，我们了解到分布式系统中部分失败是普遍存在的问题。因为客户端和服务端是都是独立的进程，一个服务端有可能因为故障或者维护而停止服务，或者此服务因为过载停止或者反应很慢。

考虑这篇文章中描述的部分失败的场景 (http://dockone.io/article/482)。假设推荐服务无法响应请求，那客户端就会由于等待响应而阻塞，这不仅会给客户带来很差的体验，而且在很多应用中还会占用很多资源，比如线程，以至于到最后由于等待响应被阻塞的客户端越来越多，线程资源被耗费完了。如下图所示：



(http://dockerone.com/uploads/article/20150802/4eb9c81e3bb9606f75eccb4f71cc84cb.png)

为了预防这种问题，设计服务时候必须要考虑部分失败的问题。

Netfilix提供 (http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html)了一个比较好的解决方案，具体的应对措施包括：

• 网络超时：当等待响应时，不要无限期的阻塞，而是采用超时策略。使用超时策略可以确保资源不会无限期的占用。
• 限制请求的次数：可以为客户端对某特定服务的请求设置一个访问上限。如果请求已达上限，就要立刻终止请求服务。
• 断路器模式（Circuit Breaker Pattern）(http://martinfowler.com/bliki/CircuitBreaker.html)：记录成功和失败请求的数量。如果失效率超过一个阈值，触发断路器使得后续的请求立刻失败。如果大量的请求失败，就可能是这个服务不可用，再发请求也无意义。在一个失效期后，客户端可以再试，如果成功，关闭此断路器。
• 提供回滚：当一个请求失败后可以进行回滚逻辑。例如，返回缓存数据或者一个系统默认值。

Netflix Hystrix (https://github.com/Netflix/Hystrix)是一个实现相关模式的开源库。如果使用JVM，推荐考虑使用Hystrix。而如果使用非JVM环境，你可以使用类似功能的库。

IPC技术

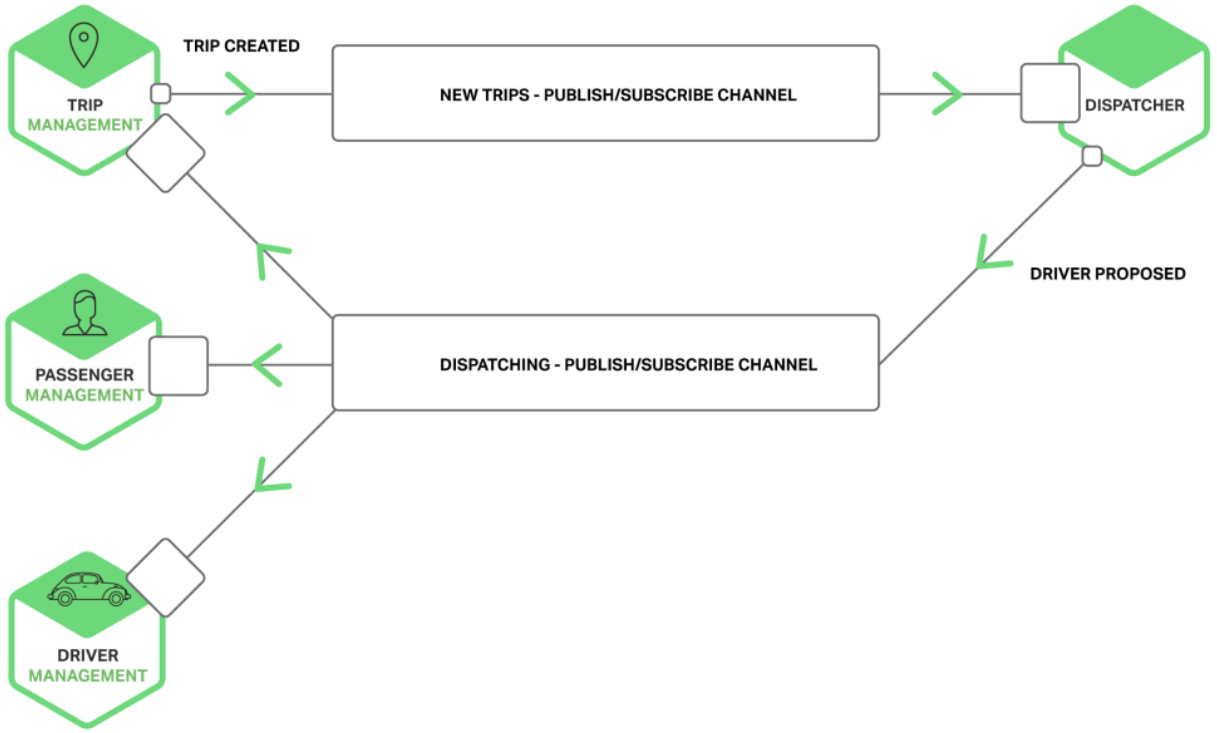
现在有很多不同的IPC技术。服务之间的通信可以使用同步的请求/响应模式，比如基于HTTP的REST或者Thrift。另外，也可以选择异步的、基于消息的通信模式，比如AMQP或者STOMP。除以之外，还有其它的消息格式供选择，比如JSON和XML，它们都是可读的，基于文本的消息格式。当然，也还有二进制格式（效率更高）的，比如Avro和Protocol Buffer。接下来我们将会讨论异步的IPC模式和同步的IPC模式，首先来看异步的。
异步的，基于消息通信

当使用基于异步交换消息的进程通信方式时，一个客户端通过向服务端发送消息提交请求。如果服务端需要回复，则会发送另外一个独立的消息给客户端。因为通信是异步的，客户端不会因为等待而阻塞，相反，客户端理所当然的认为响应不会立刻接收到。

一个消息 (http://www.enterpriseintegrationpatterns.com/Message.html)由头部（元数据例如发送方）和消息体构成。消息通过channel (http://www.enterpriseintegrationpatterns.com/MessageChannel.html)发送，任何数量的生产者都可以发送消息到channel，同样的，任何数量的消费者都可以从渠道中接受数据。有两类channel，*点对点 (http://www.enterpriseintegrationpatterns.com/PointToPointChannel.html)*和*发布/订阅 (http://www.enterpriseintegrationpatterns.com/PublishSubscribeChannel.html)*。点对点channel会把消息准确的发送到某个从channel读取消息的消费者，服务端使用点对点来实现之前提到的一对一交互模式；而发布/订阅则把消息投送到所有从channel读取数据的消费者，服务端使

用发布/订阅channel来实现上面提到的一对多交互模式。

下图展示了打车软件如何使用发布/订阅：



(http://dockerone.com/uploads/article/20150802/0855ed5ec97708bac54acd80af43641e.png)

行程管理服务在发布-订阅channel内创建一个行程消息，并通知调度服务有一个新的行程请求，调度服务发现一个可用的司机然后向发布-订阅channel写入司机建议消息（Driver Proposed message）来通知其他服务。

有很多消息系统可以选择，最好选择一种支持多编程语言的。一些消息系统支持标准协议，例如AMQP和STOMP。其他消息系统则使用独有的协议，有大量开源消息系统可选，比如RabbitMQ (https://www.rabbitmq.com/)、Apache Kafka (http://kafka.apache.org/)、Apache ActiveMQ (http://activemq.apache.org/)和NSQ (https://github.com/bitly/nsq)。它们都支持某种形式的消息和channel，并且都是可靠的、高性能和可扩展的；然而，它们的消息模型完全不同。

使用消息机制有很多优点：

• **解耦客户端和服务端：**客户端只需要将消息发送到正确的channel。客户端完全不需要了解具体的服务实例，更不需要一个发现机制来确定服务实例的位置。

• **Message Buffering：**在一个同步请求/响应协议中，例如HTTP，所有的客户端和服务端必须在

交互期间保持可用。而在消息模式中，消息broker将所有写入channel的消息按照队列方式管理，直到被消费者处理。也就是说，在线商店可以接受客户订单，即使下单系统很慢或者不可用，只要保持下单消息进入队列就好了。

• 弹性客户端-服务端交互：消息机制支持以上说的所有交互模式。

• **直接进程间通信：**基于RPC机制，试图唤醒远程服务看起来跟唤醒本地服务一样。然而，因为物理定律和部分失败可能性，他们实际上非常不同。消息使得这些不同非常明确，开发者不会出现问题。

然而，消息机制也有自己的缺点：

• **额外的操作复杂性：**消息系统需要单独安装、配置和部署。消息broker（代理）必须高可用，否则系统可靠性将会受到影响。

• **实现基于请求/响应交互模式的复杂性：**请求/响应交互模式需要完成额外的工作。每个请求消息必须包含一个回复渠道ID和相关ID。服务端发送一个包含相关ID的响应消息到channel中，使用相关ID来将响应对应到发出请求的客户端。也许这个时候，使用一个直接支持请求/响应的IPC机制会更容易些。

现在我们已经了解了基于消息的IPC，接下来我们来看看基于请求/响应模式的IPC。

同步的，基于请求/响应的IPC

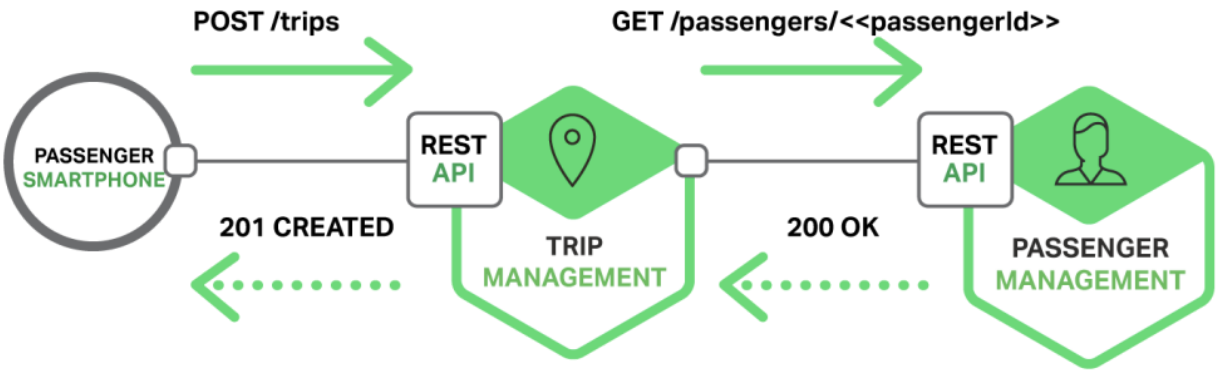
当使用一个同步的，基于请求/响应的IPC机制，客户端向服务端发送一个请求，服务端处理请求，返回响应。一些客户端会由于等待服务端响应而被阻塞，而另外一些客户端也可能使用异步的、基于事件驱动的客户端代码（Future或者Rx Observable的封装）。然而，不像使用消息机制，客户端需要响应及时返回。这个模式中有很多可选的协议，但最常见的两个协议是REST和Thrift。首先我们来看下REST。

REST

现在很流行使用RESTful (https://en.wikipedia.org/wiki/Representational_state_transfer)风格的API。REST是基于HTTP协议的。另外，一个需要理解的比较重要的概念是，REST是一个资源，一般代表一个业务对象，比如一个客户或者一个产品，或者一组商业对象。REST使用HTTP语法协议来修改资源，一般通过URL来实现。举个例子，GET请求返回一个资源的简单信息，响应格式通常是XML或者JSON对象格式。POST请求会创建一个新资源，PUT请求更新一个资源。这里引用下REST之父Roy Fielding说的：

> 当需要一个整体的、重视模块交互可扩展性、接口概括性、组件部署独立性和减小延迟、提供安全性和封装性的系统时，REST可以提供这样一组满足需求的架构。

下图展示了打车软件是如何使用REST的。



(http://dockerone.com/uploads/article/20150802/188f16abf54a6fc3446d51dc3625e571.png)

乘客通过移动端向行程管理服务的 `/trips` 资源提交了一个POST请求。行程管理服务收到请求之后，会发送一个GET请求到乘客管理服务以获取乘客信息。当确认乘客信息之后，紧接着会创建一个行程，并向移动端返回201（译者注：状态码）响应。

很多开发者都表示他们基于HTTP的API是RESTful的。但是，如同Fielding在他的博客(http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven)中所说，这些API可能并不都是RESTful的。Leonard Richardson为REST定义了一个成熟度模型(http://martinfowler.com/articles/richardsonMaturityModel.html)，具体包含以下4个层次（摘自IBM (http://www.ibm.com/developerworks/cn/java/j-lo-SpringHATEOAS/)）：

- 第一个层次（Level 0）的 Web 服务只是使用 HTTP 作为传输方式，实际上只是远程方法调用（RPC）的一种具体形式。SOAP 和 XML-RPC 都属于此类。
- 第二个层次（Level 1）的 Web 服务引入了资源的概念。每个资源有对应的标识符和表达。
- 第三个层次（Level 2）的 Web 服务使用不同的 HTTP 方法来进行不同的操作，并且使用 HTTP 状态码来表示不同的结果。如 HTTP GET 方法来获取资源，HTTP DELETE 方法来删除资源。
- 第四个层次（Level 3）的 Web 服务使用 HATEOAS。在资源的表达中包含了链接信息。客户端可以根据链接来发现可以执行的动作。

使用基于HTTP的协议有如下好处：

• HTTP非常简单并且大家都很熟悉。

• 可以使用浏览器扩展（比如Postman (https://www.getpostman.com/)）或者curl之类的命令行来测试API。
• 内置支持请求/响应模式的通信。
• HTTP对防火墙友好的。
• 不需要中间代理，简化了系统架构。

不足之处包括：

• 只支持请求/响应模式交互。可以使用HTTP通知，但是服务端必须一直发送HTTP响应才行。
• 因为客户端和服务端直接通信（没有代理或者buffer机制），在交互期间必须都在线。
• 客户端必须知道每个服务实例的URL。如之前那篇关于API Gateway的文章(http://dockone.io/article/482)所述，这也是个烦人的问题。客户端必须使用服务实例发现机制。

开发者社区最近重新发现了RESTful API接口定义语言的价值。于是就有了一些RESTful风格的服务框架，包括RAML (http://raml.org/)和Swagger (http://swagger.io/)。一些IDL，例如Swagger允许定义请求和响应消息的格式。其它的，例如RAML，需要使用另外的标识，例如JSON Schema (http://json-schema.org/)。对于描述API，IDL一般都有工具来定义客户端和服务端骨架接口。

Thrift

Apache Thrift (https://thrift.apache.org/)是一个很有趣的REST的替代品。它是Facebook实现的一种高效的、支持多种编程语言的远程服务调用的框架。Thrift提供了一个C风格的IDL定义API。使用Thrift编译器可以生成客户端和服务器端代码框架。编译器可以生成多种语言的代码，包括C++、Java、Python、PHP、Ruby, Erlang和Node.js。

Thrift接口包括一个或者多个服务。服务定义类似于一个JAVA接口，是一组方法。Thrift方法可以返回响应，也可以被定义为单向的。返回值的方法其实就是请求/响应类型交互模式的实现。客户端等待响应，并可能抛出异常。单向方法对应于通知类型的交互模式，服务端并不返回响应。

Thrift支持多种消息格式：JSON、二进制和压缩二进制。二进制比JSON更高效，因为二进制解码更快。同样原因，压缩二进制格式可以提供更高级别的压缩效率。JSON，是易读的。Thrift也可以在裸TCP和HTTP中间选择，裸TCP看起来比HTTP更加有效。然而，HTTP对防火墙，浏览器和人来说更加友好。
消息格式

了解完HTTP和Thrift后，我们来看下消息格式方面的问题。如果使用消息系统或者REST，就可以选择消息格式。其它的IPC机制，例如Thrift可能只支持部分消息格式，也许只有一种。无论哪种方式，我们必须使用一个跨语言的消息格式，这非常重要。因为指不定哪天你会使用其它语言。

有两类消息格式：文本和二进制。文本格式的例子包括JSON和XML。这种格式的优点在于不仅可读，而且是自描述的。在JSON中，一个对象就是一组键值对。类似的，在XML中，属性是由名字和值构成。消费者可以从中选择感兴趣的元素而忽略其它部分。同时，小幅度的格式修改可以很容器向后兼容。

XML文档结构是由XML schema定义的。随着时间发展，开发者社区意识到JSON也需要一个类似的机制。一个选择是使用JSON Schema，要么是独立的，要么是例如Swagger的IDL。

基于文本的消息格式最大的缺点是消息会变得冗长，特别是XML。因为消息是自描述的，所以每个消息都包含属性和值。另外一个缺点是解析文本的负担过大。所以，你可能需要考虑使用二进制格式。

二进制的格式也有很多。如果使用的是Thrift RPC，那可以使用二进制Thrift。如果选择消息格式，常用的还包括Protocol Buffers (https://developers.google.com/protocol-buffers/docs/overview)和Apache Avro (https://avro.apache.org/)。它们都提供典型的IDL来定义消息架构。一个不同点在于Protocol Buffers使用的是加标记（tag）的字段，而Avro消费者需要知道模式（schema）来解析消息。因此，使用前者，API更容易演进。这篇博客(http://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html)很好的比较了Thrift、Protocol Buffers、Avro三者的区别。

总结

微服务必须使用进程间通信机制来交互。当设计服务的通信模式时，你需要考虑几个问题：服务如何交互，每个服务如何标识API，如何升级API，以及如何处理部分失败。微服务架构有两类IPC机制可选，异步消息机制和同步请求/响应机制。在下一篇文章中，我们将会讨论微服务架构中的服务发现问题。

原文链接：Building Microservices: Inter-Process Communication in a Microservices Architecture (https://www.nginx.com/blog/building-microservices-inter-process-communication/)（翻译：杨峰 校对：李颖杰）

 (http://dockone.io/people/%E9%9A%BE%E6%98%93) 

(http://dockone.io/people/%E6%80%9D%E8%80%83zhe)

1 个评论

beiluo (http://dockone.io/people/beiluo)
赞
(http://dockone.io/people/beiluo)

2015-08-03 09:24

要回复文章请先登录 (http://dockone.io/login/)或注册 (http://dockone.io/account/register/)
