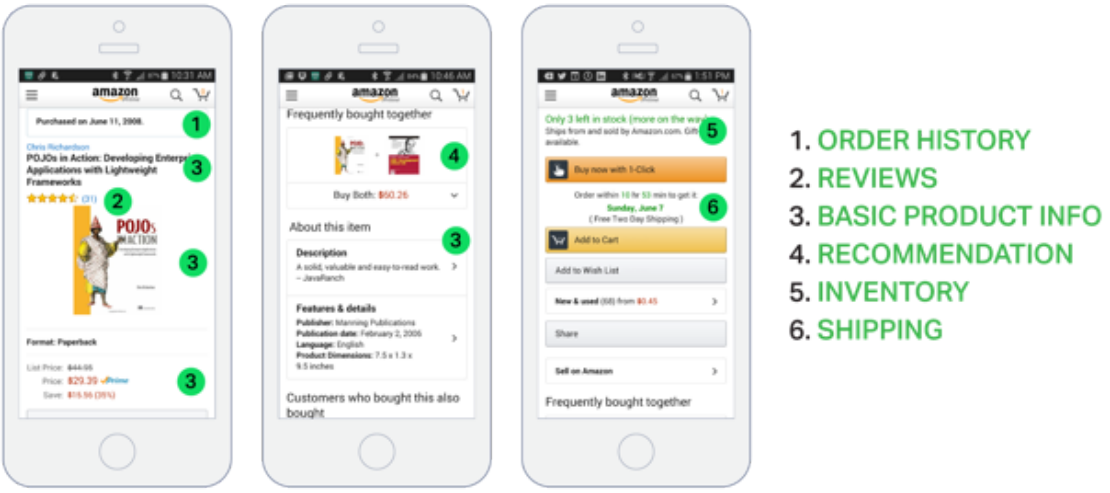
有关微服务的系列文章的第一篇介绍了微服务架构模式，讨论了使用微服务的优缺点，以及为什么它们虽然复杂度高却常常是复杂应用程序的理想选择。

当选择将应用程序构建为一组微服务时，需要确定应用程序客户端如何与微服务交互。在单体应用程序中，只有一组（通常是重复的、负载均衡的）端点。然而，在微服务架构中，每个微服务都会暴露一组通常是细粒度的端点。在本文中，我们将讨论一下这对客户端与应用程序之间的通信有什么影响，并提出一种使用API网关的方法。

开源

让我们想象一下，你要为一个购物应用程序开发一个原生移动客户端。你很可能需要实现一个产品详情页面，上面展示任何指定产品的信息。

例如，下图展示了在Amazon Android移动应用中滚动产品详情时看到的内容。



虽然这是个智能手机应用，产品详情页面也显示了大量的信息。例如，该页面不仅包含基本的产品信息（如名称、描述、价格），而且还显示了如下内容：

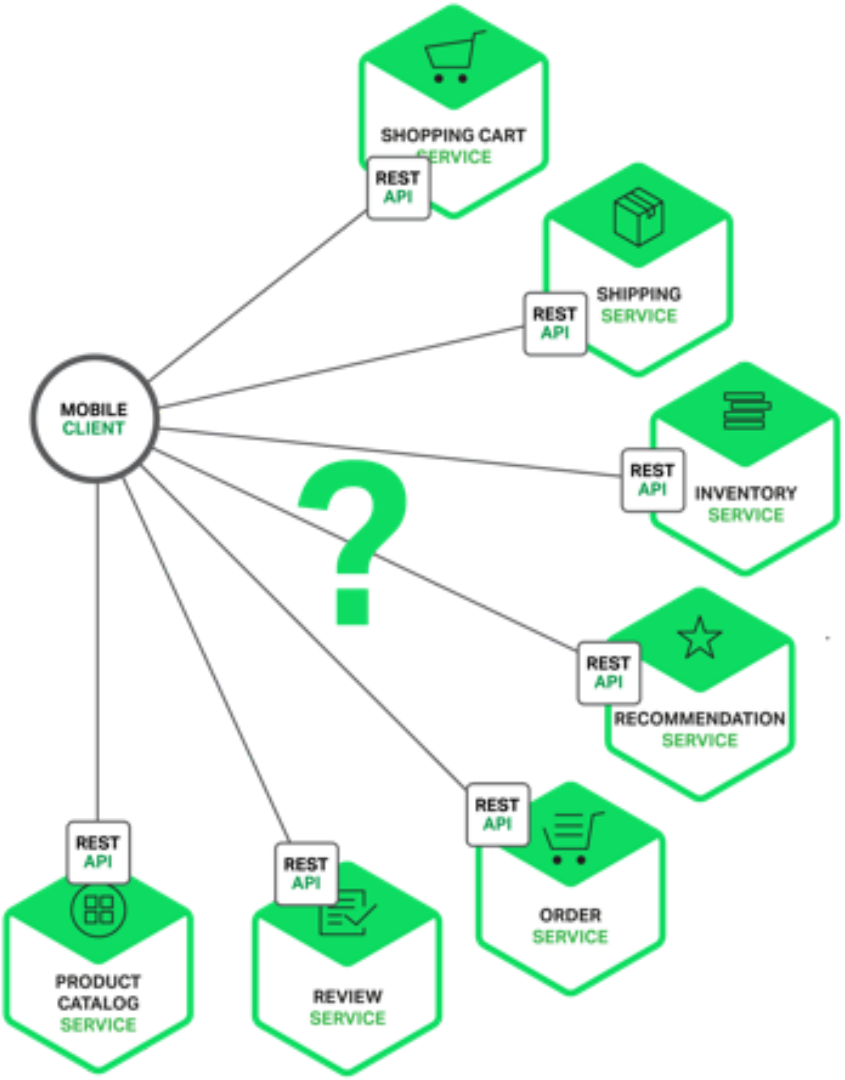
- 购物车中的件数
- 订单历史
- 客户评论
- 低库存预警
- 送货选项
- 各种推荐，包括经常与该产品一起购买的其它产品，购买该产品的客户购买的其它产品，购买该产品的客户看过的其它产品。
- 可选的购买选项。

当使用单体应用程序架构时，移动客户端将通过向应用程序发起一次REST调用（GET api.company.com/productdetails/<productId>）来获取这些数据。负载均衡器将请求路由给N个相同的应用程序实例中的一个。

然后，应用程序会查询各种数据库表，并将响应返回给客户端。

相比之下，当使用微服务架构时，产品详情页面显示的数据归多个微服务所有。下面是部分可能的微服务，它们拥有要显示在示例中产品详情页面上的数据：

- 购物车服务——购物车中的件数
- 订单服务——订单历史
- 目录服务——产品基本信息，如名称、图片和价格
- 评论服务——客户的评论
- 库存服务——低库存预警
- 送货服务——送货选项、期限和费用，这些单独从送货方的API获取
- 推荐服务——建议的产品



我们需要决定移动客户端如何访问这些服务。让我们看看都有哪些选项。

客户端与微服务直接通信

从理论上讲，客户端可以直接向每个微服务发送请求。每个微服务都有一个公开的端点(https://<serviceName>.api.company.name）。该URL将映射到微服务

的负载均衡器，由它负责在可用实例之间分发请求。为了获取产品详情，移动客户端将逐一向上面列出的N个服务发送请求。

遗憾的是，这种方法存在挑战和局限。一个问题是客户端需求和每个微服务暴露的细粒度API不匹配。在这个例子中，客户端需要发送7个独立请求。在更复杂的应用程序中，可能要发送更多的请求。例如，按照Amazon的说法，他们在显示他们的产品页面时就调用了数百个服务。然而，客户端通过LAN发送许多请求，这在公网上可能会很低效，而在移动网络上就根本不可行。这种方法还使得客户端代码非常复杂。

客户端直接调用微服务的另一个问题是，部分服务使用的协议不是Web友好协议。一个服务可能使用Thrift二进制RPC，而另一个服务可能使用AMQP消息传递协议。不管哪种协议都不是浏览器友好或防火墙友好的，最好是内部使用。在防火墙之外，应用程序应该使用诸如HTTP和WebSocket之类的协议。

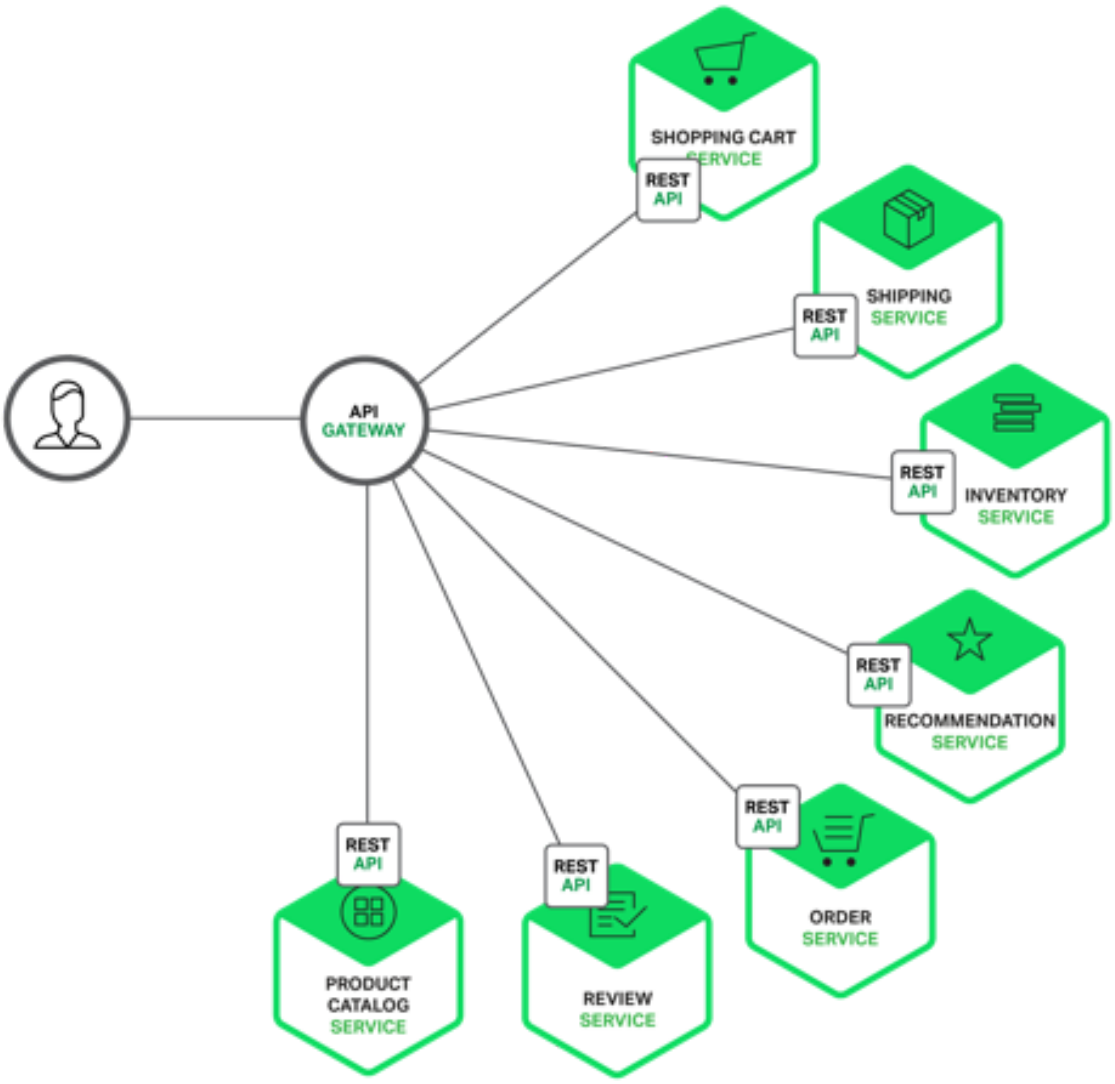
这种方法的另一个缺点是，它会使得微服务难以重构。随着时间推移，我们可能想要更改系统划分成服务的方式。例如，我们可能合并两个服务，或者将一个服务拆分成两个或更多服务。然而，如果客户端与微服务直接通信，那么执行这类重构就非常困难了。

由于这些问题的存在，客户端与微服务直接通信很少是合理的。

使用API网关

通常，一个更好的方法是使用所谓的API网关。API网关是一个服务器，是系统的唯一入口。从面向对象设计的角度看，它与外观模式类似。API网关封装了系统内部架构，为每个客户端提供一个定制的API。它可能还具有其它职责，如身份验证、监控、负载均衡、缓存、“请求整形（request shaping）”与管理、静态响应处理。

下图展示了API网关通常如何融入架构：



API网关负责服务请求路由、组合及协议转换。客户端的所有请求都首先经过API网关，然后由它将请求路由到合适的微服务。API网管经常会通过调用多个微服务并合并结果来处理一个请求。它可以在Web协议（如HTTP与WebSocket）与内部使用的非Web友好协议之间转换。

API网关还能为每个客户端提供一个定制的API。通常，它会向移动客户端暴露一个粗粒度的API。例如，考虑下产品详情的场景。API网关可以提供一个端点（/productdetails?productid=xxx），使移动客户端可以通过一个请求获取所有的产品详情。API网关通过调用各个服务（产品信息、推荐、评论等等）并合并结果来处理请求。

Netflix API网关是一个很好的API网关实例。Netflix流服务提供给数以百计的不同类型的设备使用，包括电视、机顶盒、智能手机、游戏系统、平板电脑等等。最初，Netflix试图为他们的流服务提供一个通用的API。然而他们发现，由于各种各样的设备都有自己独特的需求，这种方式并不能很好地工作。如今，他们使用一个API网关，通过运行特定于设备的适配器代码来为每个设备提供一个定制的API。通常，一个适配器通过调用平均6到7个后端服务来处理每个请求。Netflix API网关每天处理数十亿请求。

API网关的优点和不足

如你所料，使用API网关有优点也有不足。使用API网关的最大优点是，它封装了应用程序的内部结构。客户端只需要同网关交互，而不必调用特定的服务。API网关为每一类客户端提供了特定的API。这减少了客户端与应用程序间的交互次数，还简化了客户端代码。

API网关也有一些不足。它增加了一个我们必须开发、部署和维护的高可用组件。还有一个风险是，API网关变成了开发瓶颈。为了暴露每个微服务的端点，开发人员必须更新API网关。API网关的更新过程要尽可能地简单，这很重要。否则，为了更新网关，开发人员将不得不排队等待。不过，虽然有这些不足，但对于大多数现实世界的应用程序而言，使用API网关是合理的。

实现API网关

到目前为止，我们已经探讨了使用API网关的动机及其优缺点。下面让我们看一下需要考虑的各种设计问题。

性能和可伸缩性

只有少数公司有Netflix的规模，每天需要处理数十亿请求。不管怎样，对于大多数应用程序而言，API网关的性能和可扩展性通常都非常重要。因此，将API网关构建在一个支持异步、I/O非阻塞的平台上是合理的。有多种不同的技术可以用于实现一个可扩展的API网关。在JVM上，可以使用一种基于NIO的框架，比如Netty、Vertx、Spring Reactor或JBoss Undertow中的一种。一个非常流行的非JVM选项是Node.js，它是一个以Chrome JavaScript引擎为基础构建的平台。另一个选项是使用NGINX Plus。NGINX Plus提供了一个成熟的、可扩展的、高性能Web服务器和一个易于部署的、可配置可编程的反向代理。NGINX Plus可以管理身份验证、访问控制、负载均衡请求、缓存响应，并提供应用程序可感知的健康检查和监控。

使用响应式编程模型

API网关通过简单地将请求路由给合适的后端服务来处理部分请求，而通过调用多个后端服务并合并结果来处理其它请求。对于部分请求，比如产品详情相关的多个请求，它们对后端服务的请求是独立于其它请求的。为了最小化响应时间，API网关应该并发执行独立请求。然而，有时候，请求之间存在依赖。在将请求路由到后端服务之前，API网关可能首先需要调用身份验证服务验证请求的合法性。类似地，为了获取客户意愿清单中的产品信息，API网关必须首先获取包含那些信息的客户资料，然后再获取每个产品的信息。关于API组合，另一个有趣的例子是Netflix Video Grid。

使用传统的异步回调方法编写API组合代码会让你迅速坠入回调地狱。代码会变得混乱、难以理解且容易出错。一个更好的方法是使用响应式方法以一种声明式样式编写API网关代码。响应式抽象概念的例子有Scala中的Future、Java 8中的CompletableFuture和JavaScript中的Promise，还有最初是微软为.NET平台开发的Reactive Extensions（RX）。Netflix创建了RxJava for JVM，专门用于他们的API网关。此外，还有RxJS for JavaScript，它既可以在浏览器中运行，也可以在Node.js中运行。使用响应式方

法将使你可以编写简单但高效的API网关代码。

服务调用

基于微服务的应用程序是一个分布式系统，必须使用一种进程间通信机制。有两种类型的进程间通信机制可供选择。一种是使用异步的、基于消息传递的机制。有些实现使用诸如JMS或AMQP那样的消息代理，而其它的实现（如Zeromq）则没有代理，服务间直接通信。另一种进程间通信类型是诸如HTTP或Thrift那样的同步机制。通常，一个系统会同时使用异步和同步两种类型。它甚至还可能使用同一类型的多种实现。总之，API网关需要支持多种通信机制。

服务发现

API网关需要知道它与之通信的每个微服务的位置（IP地址和端口）。在传统的应用程序中，或许可以硬连线这个位置，但在现代的、基于云的微服务应用程序中，这并不是一个容易解决的问题。基础设施服务（如消息代理）通常会有一个静态位置，可以通过OS环境变量指定。但是，确定一个应用程序服务的位置没有这么简单。应用程序服务的位置是动态分配的。而且，单个服务的一组实例也会随着自动扩展或升级而动态变化。总之，像系统中的其它服务客户端一样，API网关需要使用系统的服务发现机制，可以是服务器端发现，也可以是客户及需求。下一篇文章将更详细地描述服务发现。现在，需要注意的是，如果系统使用客户端发现，那么API网关必须能够查询服务注册中心，这是一个包含所有微服务实例及其位置的数据库。

处理局部失败

在实现API网关时，还有一个问题需要处理，就是局部失败的问题。该问题在所有的分布式系统中都会出现，无论什么时候，当一个服务调用另一个响应慢或不可用的服务，就会出现这个问题。API网关永远不能因为无限期地等待下游服务而阻塞。不过，如何处理失败取决于特定的场景以及哪个服务失败。例如，在产品详情场景下，如果推荐服务无响应，那么API网关应该向客户端返回产品详情的其它内容，因为它们对用户依然有用。推荐内容可以为空，也可以，比如说，用一个固定的TOP 10列表取代。不过，如果产品信息服务无响应，那么API网关应该向客户端返回一个错误信息。

如果缓存数据可用，那么API网关还可以返回缓存数据。例如，由于产品价格不经常变化，所以如果价格服务不可用，API网关可以返回缓存的价格数据。数据可以由API网关自己缓存，也可以存储在像Redis或Memcached那样的外部缓存中。通过返回默认数据或者缓存数据，API网关可以确保系统故障不影响用户的体验。

在编写代码调用远程服务方面，Netflix Hystrix是一个异常有用的库。Hystrix会将超出设定阀值的调用超时。它实现了一个“断路器（circuit breaker）”模式，可以防止客户端对无响应的服务进行不必要的等待。如果服务的错误率超出了设定的阀值，那么Hystrix会切断断路器，在一个指定的时间范围内，所有请求都会立即失败。Hystrix允许用户定义一个请求失败后的后援操作，比如从缓存读取数据，或者返回一个默认值。如果你正在使用JVM，那么你绝对应该考虑使用Hystrix。而如果你正在使用一个

非JVM环境，那么你应该使用一个等效的库。

小结

对于大多数基于微服务的应用程序而言，实现一个API网关是有意义的，它可以作为系统的唯一入口。API网关负责服务请求路由、组合及协议转换。它为每个应用程序客户端提供一个定制的API。API网关还可以通过返回缓存数据或默认数据屏蔽后端服务失败。在本系列的下一篇文章中，我们将探讨服务间通信。

感谢郭蕾对本文的审校。