03 微服务实战：从架构到发布（一）

MAR 2016

IN 后端编程,架构

引言：“微服务”是当前软件架构领域非常热门的词汇，能找到很多关于微服务的定义、准则，以及如何从微服务中获益的文章，在企业的实践中去应用“微服务”的资源却很少。本篇文章中，会介绍微服务架构（Microservices Architecture）的基础概念，以及如何在实践中具体应用。

单体架构(Monolithic Architecture)

企业级的应用一般都会面临各种各样的业务需求，而常见的方式是把大量功能堆积到同一个单体架构中去。比如：常见的ERP、CRM等系统都以单体架构的方式运行，同时由于提供了大量的业务功能，随着功能的升级，整个研发、发布、定位问题，扩展，升级这样一个“怪物”系统会变得越来越困难。

单体架构的初期效率很高，应用会随着时间推移逐渐变大。在每次的迭代中，开发团队都会面对新功能，然后开发许多新代码，随着时间推移，这个简单的应用会变成了一个巨大的怪物。

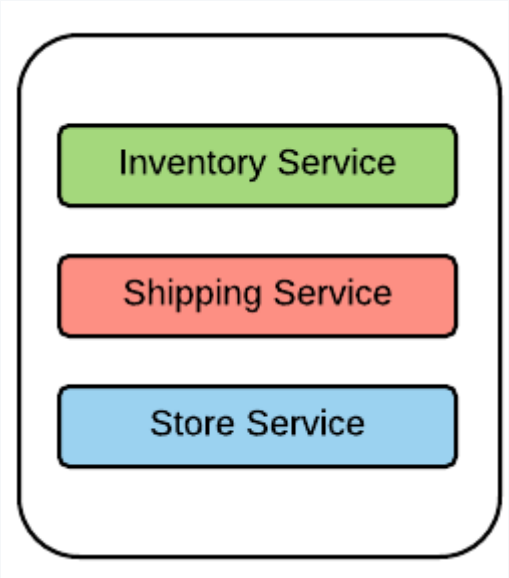

图1：单体架构

大部分企业通过SOA来解决上述问题，SOA的思路是把应用中相近的功能聚合到一起，以服务的形式提供出去。因此基于SOA架构的应用可以理解为一批服务的组合。SOA带来的问题是，引入了大量的服务、消息格式定义和规范。

多数情况下，SOA的服务直接相互独立，但是部署在同一个运行环境中（类似于一个Tomcat实例下，运行了很多web应用）。和单体架构类似，随着业务功能的增多SOA的服务会变得越来越复杂，本质上看没有因为使用SOA而变的更好。图1，是一个包含多种服务的在线零售网站，所有的服务部署在一个运行环境中，是一个典型的单体架构。

单体架构的应用一般有以下特点：

- 设计、开发、部署为一个单独的单元。
- 会变得越来越复杂，最后导致维护、升级、新增功能变得异常困难
- 很难以敏捷研发模式进行开发和发布
- 部分更新，都需要重新部署整个应用

- 水平扩展：必须以应用为单位进行扩展，在资源需求有冲突时扩展变得比较困难（部分服务需要更多的计算资源，部分需要更多内存资源）
- 可用性：一个服务的不稳定会导致整个应用出问题
- 创新困难：很难引入新的技术和框架，所有的功能都构建在同质的框架之上

微服务架构（Microservices Architecture）

微服务架构的核心思想是，一个应用是由多个小的、相互独立的、微服务组成，这些服务运行在自己的进程中，开发和发布都没有依赖。

多数人对于微服务的定义是，把本来运行在单体架构中的服务拆分成相互独立的服务，并运行在各自的进程中。在我看来，不仅如此。最关键的地方在于，不同的服务能依据不同的业务需求，构建的不同的技术架构之上，并且聚焦在有限的业务功能之上。

因此，在线零售网站可以用图2的微服务架构来简单概括。基于业务需求，需要增加一个账户服务微服务，因此构建微服务绝不是在单体架构中把服务拆分开这么简单。

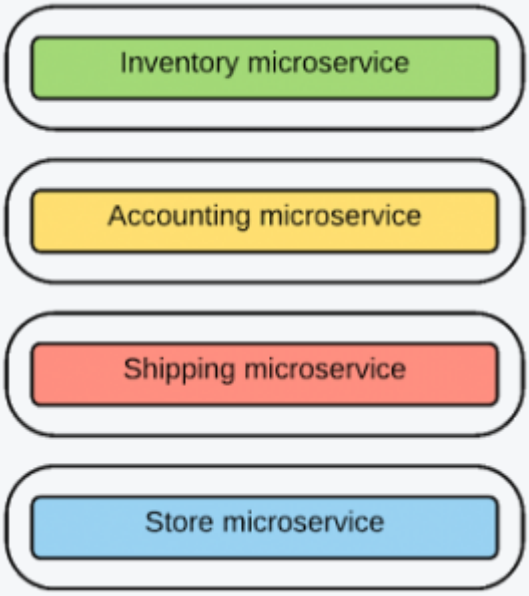

图2：微服务架构

微服务设计：规模、范围、业务功能

你可能从零开始用微服务来构建应用，也可能重构现有系统，确定微服务的规模，范围和功能都特别重要。让我们讨论一些有关微服务设计的关键问题和对它的误解：

- “微”很容易被误解：很多开发者会倾向于把服务往尽量小的颗粒度去做
- 在SOA方式下，服务都还是以单体架构在运行，用于支持不同的功能。如果依旧采用SAO类似的服务，仅仅是名义上叫做微服务，并不能带来任何微服务的优势。

那我们在微服务中应该怎样设计呢。以下是微服务的设计指南：

- 职责单一原则（Single Responsibility Principle）：把某一个微服务的功能聚焦在特定业务或者有限的范围内会有助于敏捷开发和服务的发布。
- 设计阶段就需要把业务范围进行界定。
- 需要关心微服务的业务范围，而不是服务的数量和规模尽量小。数量和规模需要依照业务功能而定。

- 于SOA不同，某个微服务的功能、操作和消息协议尽量简单。
- 项目初期把服务的范围制定相对宽泛，随着深入，进一步重构服务，细分微服务是个很好的做法。

微服务消息

在单体架构中，不同功能之间通信通过方法调用，或者跨语言通信。SOA降低了这种语言直接的耦合度，采用基于SOAP协议的web服务。这种web服务的功能和消息体定义都十分复杂，微服务需要更轻量的机制。

同步消息 – REST, Thrift

同步消息就是客户端需要保持等待，直到服务器返回应答。REST是微服务中默认的同步消息方式，它提供了基于HTTP协议和资源API风格的简单消息格式，多数微服务都采用这种方式（每个功能代表了一个资源和对应的操作）。

Thrift是另外一个可选的方案。它采用接口描述语言定义并创建服务，支持可扩展的跨语言服务开发，所包含的代码生成引擎可以在多种语言中，如 C++, Java, Python, PHP, Ruby, Erlang, Perl, Haskell, C#, Cocoa, Smalltalk 等创建高效的、无缝的服务，其传输数据采用二进制格式，相对 XML 和 JSON 体积更小，对于高并发、大数据量和多语言的环境更有优势。

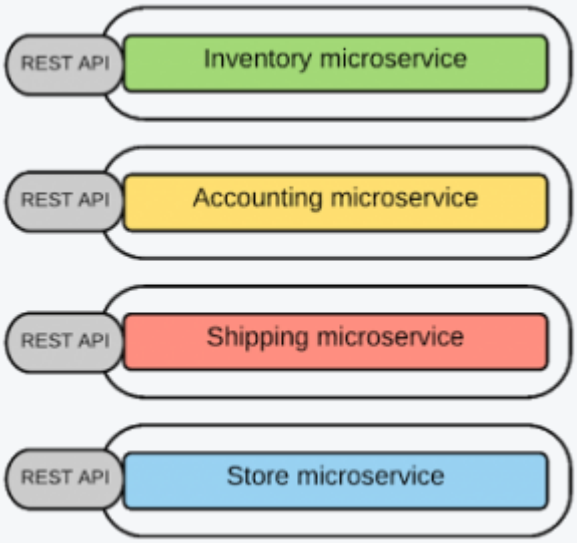

图3：REST接口，对外微服务

异步消息 – AMQP, STOMP, MQTT

异步消息就是客户端不需要一直等待服务应答，有应到后会得到通知。某些微服务需要用到异步消息，一般采用AMQP, STOMP, MQTT。

消息格式 – JSON, XML, Thrift, ProtoBuf, Avro

消息格式是微服务中另外一个很重要的因素。SOA的web服务一般采用文本消息，基于复杂的消息格式（SOAP）和消息定义（xsd）。微服务采用简单的文本协议JSON和XML，基于HTTP的资源API风格。如果需要二进制，通过用到Thrift, ProtoBuf, Avro。

服务约定 – 定义接口 – Swagger, RAML, Thrift IDL

如果把功能实现为服务，并发布，需要定义一套约定。单体架构中，SOA采用WSDL，WSDL过于复杂并且和SOAP紧耦合，不适合微服务。

REST设计的微服务，通常采用Swagger和RAML定义约定。

对于不是基于REST设计的微服务，比如Thrift，通常采用IDL（Interface Definition Languages），比如Thrift IDL。

微服务集成 (服务间通信)

微服务架构下，应用的服务直接相互独立。在一个具体的商业应用中，需要有些机制支持微服务之间通信。因此服务间的通信机制特别重要。

SOA体系下，服务之间通过企业服务总线（Enterprise Service Bus）通信，许多业务逻辑在中间层（消息的路由、转换和组织）。微服务架构倾向于降低中心消息总线（类似于ESB）的依赖，将业务逻辑分布在每个具体的服务终端。

大部分微服务基于HTTP、JSON这样的标准协议，集成不同标准和格式变的不再重要。另外一个选择是采用轻量级的消息总线或者网关，有路由功能，没有复杂的业务逻辑。下面就介绍几种常见的架构方式。

点对点方式 – 直接调用服务

点对点方式中，服务之间直接用。每个微服务都开放REST API，并且调用其它微服务的接口。

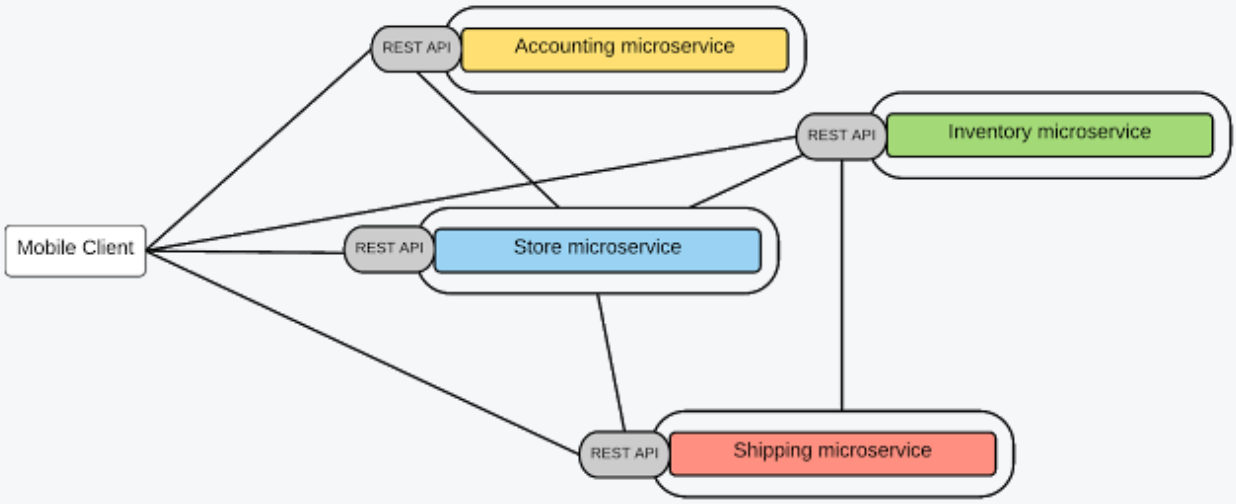

图4：通过点对点方式通信

很明显，在比较简单的微服务应用场景下，这种方式还可行，随着应用复杂度的提升，会变得越来越不可维护。这点有些类似SOA的ESB，尽量不采用点对点的集成方式。

点对点有下面几个缺点:

- 非功能的需求，比如用户授权、限制、监控，需要在每个微服务中进行实现
- 随着功能的演进，服务会变得越来越复杂。
- 不同的服务直接，客户端和服务直接没有控制功能（监控、跟踪、过滤）
- 直接通信在大型系统设计中，一般是反面典型。

因此，如果设计一个大型的微服务系统，尽量避免点对点的通信方式，也不能像ESB这样重量级的总线。而是一个轻量级的总线，能够提供非业务功能的抽象。这就是API网关方式。

API-网关方式

API网关方式的核心要点是，所有的客户端和消费端都通过统一的网关接入微服务，在网关层处理所有的非业务功能个。通常，网关也是提供REST/HTTP的访问API。服务端通过API-GW注册和管理服务。

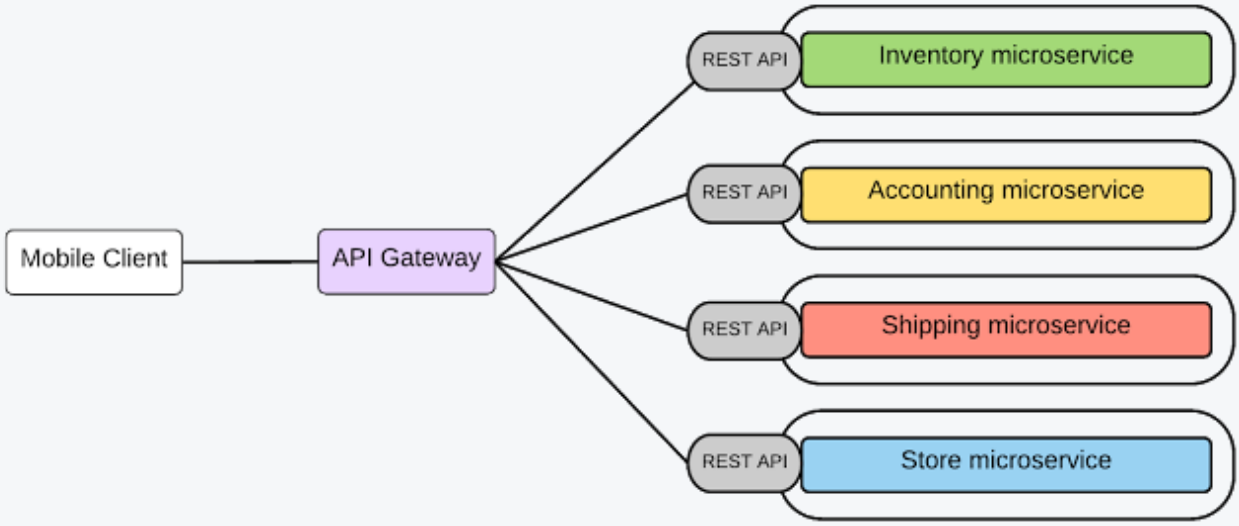

图5：通过API-网关暴露微服务

用我们网上商店的例子，在图5中，所有的业务接口通过API网关暴露，是所有客户端接口的唯一入口。微服务之间的通信也通过API网关。

采用网关方式有如下优势：

- 有能力为微服务接口提供网关层次的抽象。比如：微服务的接口可以各种各样，在网关层，可以对外暴露统一的规范接口。
- 轻量的消息路由、格式转换。
- 统一控制安全、监控、限流等非业务功能。
- 每个微服务会变得更加轻量，非业务功能个都在网关层统一处理，微服务只需要关注业务逻辑

目前，API网关方式应该是微服务架构中应用最广泛的设计模式。

消息代理方式

微服务也可以集成在异步的场景下，通过队列和订阅主题，实现消息的发布和订阅。一个微服务可以是消息的发布者，把消息通过异步的方式发送到队列或者订阅主题下。作为消费者的微服务可以从队列或者主题共获取消息。通过消息中间件把服务之间的直接调用解耦。

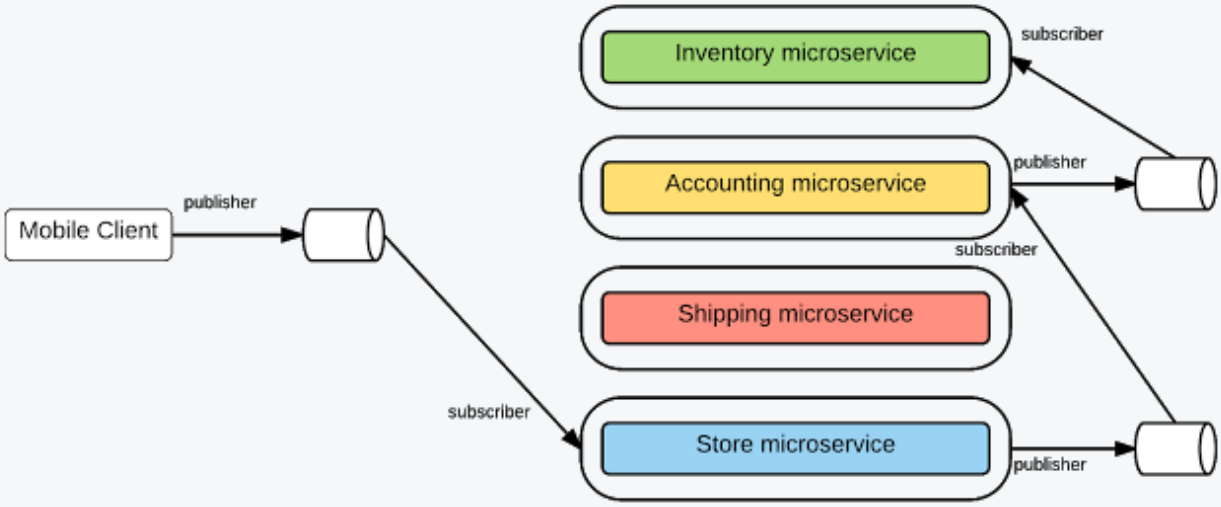

图6：异步通信方式

通常异步的生产者/消费者模式，通过AMQP、MQTT等异步消息规范。

数据的去中心化

单体架构中，不同功能的服务模块都把数据存储在某个中心数据库中。

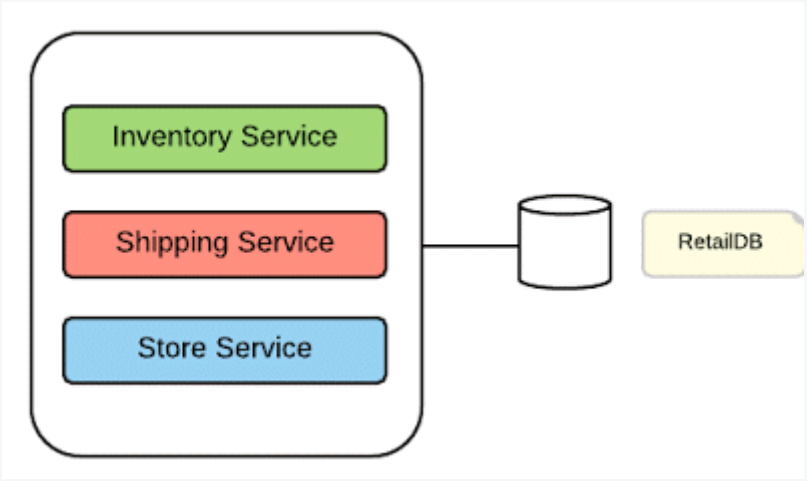

图7：单体架构，用一个数据库存储所有数据

微服务方式，多个服务之间的设计相互独立，数据也应该相互独立（比如，某个微服务的数据库结构定义方式改变，可能会中断其它服务）。因此，每个微服务都应该有自己的数据库。

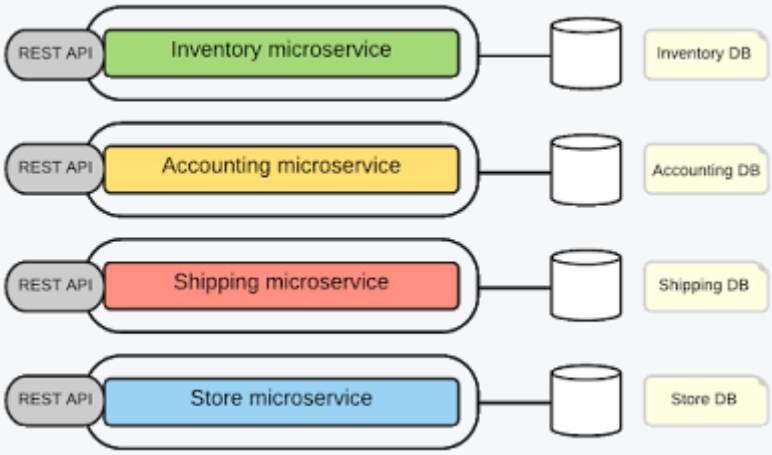

图8：每个微服务有自己私有的数据库，其它微服务不能直接访问。

数据去中心话的核心要点：

- 每个微服务有自己私有的数据库持久化业务数据
- 每个微服务只能访问自己的数据库，而不能访问其它服务的数据库
- 某些业务场景下，需要在一个事务中更新多个数据库。这种情况也不能直接访问其它微服务的数据库，而是通过对于微服务进行操作。

数据的去中心化，进一步降低了微服务之间的耦合度，不同服务可以采用不同的数据库技术（SQL、NoSQL等）。在复杂的业务场景下，如果包含多个微服务，通常在客户端或者中间层（网关）处理。

下篇文章会介绍微服务实战的其它内容：管理去中心化、服务的注册和发现、安全、事务、失败的设计、其它。

原文作者：Kasun Indrasiri，软件架构师，WSO2
原文链接：https://dzone.com/articles/microservices-in-practice-1
翻译自MaxLeap团队_云服务研发成员：Frank Qin

关于MaxLeap
MaxLeap移动云服务平台为企业提供一站式的移动研发和运营云服务，帮助企业快速研发和上线移动应用，平台提供数据云存储，云引擎，支付管理，IM，数据分析和营销自动化等服务。
MaxLeap官网链接： https://maxleap.cn

如果您正在学习移动研发和云服务等方面的讯息，不妨关注我们的微信服务号MaxLeapSvc，我们将不定期推送相关干货。敬请期待！

DOCKER JAVA SOA 分布式 微服务 架构