我们对于一些事物的不理解或者畏惧，原因都在于这些

事情有意无意带有的绚丽外衣和神秘面纱。

只要揭开这个外衣，就如一个脱了衣服的美女站在你身前，

有什么好神秘的？还有什么好惊艳的？

Surging

刘旭东

[公司名称]

Surging

使用 .Net Core构建微服务



目录

[1介绍微服务 2](#_Toc503675660)

[1.1它是啥玩意？ 2](#_Toc503675661)

[1.2深刻理解它 3](#_Toc503675662)

[1.3为啥使用它 4](#_Toc503675663)

[1.4牢记它的工作原理 4](#_Toc503675664)

[1.5它的优势 6](#_Toc503675665)

[1.6不是你想就能用——先决条件？ 7](#_Toc503675666)

[1.7微服务投入生产使用前的热身 7](#_Toc503675667)

[2实现微服务架构 9](#_Toc503675668)

[2.1一个服务有多大？ 9](#_Toc503675669)

[2.2好的服务的标准是什么 9](#_Toc503675670)

[2.3我的最爱—DDD和它对于微服务的重要性 10](#_Toc503675671)

[2.4The concept of seam 10](#_Toc503675672)

[2.5服务之间的通信 10](#_Toc503675673)

[2.6Revisiting the Flix One case study 10](#_Toc503675674)

[3服务之间的技术集成 10](#_Toc503675675)

[3.1服务之间的通信 10](#_Toc503675676)

[3.2套路之——集成模式 10](#_Toc503675677)

[The API gateway 10](#_Toc503675678)

[The event-driven pattern 11](#_Toc503675679)

[Event sourcing 11](#_Toc503675680)

[Eventual consistency 11](#_Toc503675681)

[Compensating transactions 11](#_Toc503675682)

[Competing consumers 11](#_Toc503675683)

[Azure Service Bus 11](#_Toc503675684)

[Azure queues 11](#_Toc503675685)

[4服务的测试 11](#_Toc503675686)

[4.1如何进行 11](#_Toc503675687)

[4.2测试方法 11](#_Toc503675688)

[4.3Testing pyramid 11](#_Toc503675689)

[4.3测试的分类 11](#_Toc503675690)

[4.3.1单元测试 11](#_Toc503675691)

[4.3.2组件(服务)测试 11](#_Toc503675692)

[4.3.3集成测试 11](#_Toc503675693)

[4.3.4契约测试 12](#_Toc503675694)

[4.3.5性能测试 12](#_Toc503675695)

[4.3.6端对端(前后端)测试 12](#_Toc503675696)

[Sociable versus isolated unit tests 12](#_Toc503675697)

[约定与模拟 12](#_Toc503675698)

[4.4实操演练 12](#_Toc503675699)

[5微服务的部署 12](#_Toc503675700)

[5.1部署实施层面的术语 12](#_Toc503675701)

[5.2成功部署的先决条件 12](#_Toc503675702)

[5.3部署的隔离需求 12](#_Toc503675703)

[5.4部署范例 12](#_Toc503675704)

[5.5容器技术 12](#_Toc503675705)

[5.6Docker介绍 12](#_Toc503675706)

[6系统安全性 12](#_Toc503675707)

[7运行监控 12](#_Toc503675708)

[8系统扩展性 13](#_Toc503675709)

[9Introduction to Reactive Microservices 13](#_Toc503675710)

[10Surging框架的实现解说 13](#_Toc503675711)

**前言**：本书是一个中文翻译，但绝不是一个文字转换而已，而是加入了自己的感悟。

本书是一个技术书，但绝不是一个教程指南，而是加入了改变你思维的过程，骚年接受洗脑吧

最终目的是为了，让**程序员**快速\简单的理解**架构**中的**微服务**的本质与套路，所以有些概念和原理被故意简化了，请大牛们，原谅我这个小菜鸟用来这么通俗的语言描述架构。但是作为码农的你知道那么多干啥？拿起键盘写代码就是了

# 1介绍微服务

## 1.1它是啥玩意？

微服务 (Microservices) 是一种软件架构风格，它是以专注于单一责任与功能的小型功能区块 (Small Building Blocks) 为基础，利用模组化的方式组合出复杂的大型应用程序，各功能区块使用与语言无关 (Language-Independent/Language agnostic) 的 API 集相互通讯

注：上面是一个非常官方的古板的定义，如果没有达到架构师级别,应该很难理解，什么是架构风格，不用想那么多，可以简单的认为架构方式的类别，就是软件设计的最基础的逻辑实现方式。看下面的几个分类，你就会明白，微服务不是什么新玩意，不是从天而降，而是软件设计的一种深入发展的结果。

#### 架构风格——就是架构的分类

**客户端-服务器** （C/S或者B/S）  
将系统分为两个应用，其中客户端向服务器发送服务请求。

**基于组件的架构** (COM技术)  
把应用设计分解为可重用的功能、逻辑组件，这些组件的位置相互透明，只暴露明确定义的通信接口。

**消息总线** （TCP或者MQ）  
指接收、发送消息的软件系统，消息基于一组已知格式，以便系统无需知道实际接收者就能互相通信。

**N层/三层架构** （你现在用的就是这个）  
用与分层风格差不多一样的方式将功能划分为独立的部分，每个部分是一个层，处于完全独立的计算机上。

**面向服务架构（SOA）** （都已经没落了，就是WebService）  
是指那些利用契约和消息将功能暴露为服务、消费功能服务的应用。

看完了吧，是不是很熟悉的感觉？不熟悉？那你至少应该知道B/S架构吧。最原始的B/S架构就是一个aps.net mvc网站，

浏览器作为客户端，网站作为服务端，后面一个数据库。

然后你发现服务端的方法写的太tm长了（一个主页面快要达到2万行代码），无法维护，怎么办？把方法拆分开呗，你把方法分成了多个，然后你把方法写在了一个单独的程序集(DLL动态库)里。这个时候就是在使用N层架构。我擦嘞，系统用了几年，客户不断的提需求，DLL越来越大，我又干不动了，怎么办啊，继续拆分啊，然后你用了webService(Java和C#都有这个技术) ，把DLL放在不同的地方运行，直接而不是放在一起靠程序集引用来使用。系统运行到今天，客户的需求又复杂了，以前不同的业务交叉的地方，都是用数据库的事务来统一处理。以前都是webService为几个系统来服务（大的商业集团都是全国有分公司的，一个网站前端用负载均衡再加上几个独立部署的服务端，然后公用一个集群数据库，） 。现在呢？界面被用户玩的越来越花哨，他们已经恨不得看一眼UI界面就把操作给完成了，导致本来一个简单的往数据库添加一个订单记录而已，现在变成了先调用webservice的库存服务，然后调用webservice的订单服务，然后通过webservice调用银行支付服务，最后还得通过webservice调用快递公司的服务，甚至有得还通过webservice调用会员积分服务。写代码时间长了，你越来越觉得虽然用了webservice但是仍然有一些问题很难解决，终于你在某个加班到凌晨的夜里，抬头仰望星空，发出一声长叹：为什么每次改动很小的代码，总是会引起很多问题来啊！难道没有更好的解决办法，可以让老子快点写完代码、快点测试、快点发布版本、快点下班吗？或许是你的诚心感动了老天爷，天空飘来三个字：微服务

## 1.2深刻理解它

微服务架构是用来开发由服务构成的软件的一种方式，如果不理解这句话，你类比一下：

C/S架构是用来开发由客户端和服务端构成软件的一种方式。而这些服务是彼此完全独立的拥有各自的数据存储方式（绝对不能公用数据库）并运行在各自的进程内，可以说彼此完全不知道对方的存在。那么这样以来，每个服务都能够独立开发和部署（在不同电脑上面的不同进程内，因为他们完全没有依赖啊，就像两个不相干的应用，你甚至可以认为每个服务就是一个独立的软件）。换句话说，我们可以通过多个小服务（微服务）将以前的庞大的一堆webservice再进行细化，完全让不同的团队(比如C#和python)来各自设计、写代码、测试、发布这些小服务。

注：通俗的说，SOA的服务在微服务架构中完全更提升一个档次变成了独立的应用，因为微服务架构中只有服务，再无其他任何顶层概念。到这里，你应该有点恍惚，因为只有服务，那么怎么构成整个应用啊？就不告诉你—继续往下看

微服务中没有SOAP消息（就是没有webservice的那样使用代理类的调用方式），而每个服务又是一个独立的进程，从操作系统角度来看，他们的内存空间都是彼此隔离的，甚至不再一个电脑上面，想要交换数据可就发愁了？想起那句话没有，微服务不是什么新东西，微服务是一种架构风格，是SOA的升级版.其实不用多想，因为计算机最本质的通信方式全世界只有两种TCP和UDP，你TM还能想出第三种？所以理论上虽然SOAP不能用了，但是你可以使用任何构建与TCP之上的通信方式。比如：HTTP技术、RPC技术、消息队列技术，或者你直接使用封装的TCP框架。刚才从进程方面来谈，在服务通信过程中实际执行代码的时候，还是线程在起作用，那么问题来了，关于多线程技术，你至少得了解同步和异步调用的区别。差点忘了，TCP传输的消息格式还没有谈到，理论上来讲，你用啥都行，只要通信双方能理解就OK，但是不要重复造轮子，有很多成熟的技术，比如JSON、二进制、或者你使用的传输方式默认配置的格式

## 1.3为啥使用它

Tremendous patterns and architectures have been explored with some gaining popularity;

there are others, though, that are losing the battle of internet traffic. With each solution

having its own advantages and disadvantages, it has become increasingly important for

companies to quickly respond to fundamental demands, such as scalability, high

performance, and easy deployment. Any single aspect failing to be fulfilled in a cost-

effective manner could easily impact large businesses negatively, making a big difference

between a profitable and non-profitable venture.

This is where we see microservices coming to the rescue of enterprise system architects. They

can ensure their designs against problems mentioned previously, with the help of this

architectural style. It is also important to consider the fact that this objective is met in a cost-

effective manner while respecting the time involved.

注：感觉这些都是废话，所以没有翻译。作为架构师，面对当前的软件行业客户需求，面对采用老技术的各种瓶颈，使用服务化的软件构成方式是不二选择，除非你能创造出比小服务化更高的性能、更高的扩展性、更低的开发成本、更快的部署的架构风格来！

## 1.4牢记它的工作原理

It's programming of the modern era, where we are expected to follow all SOLID

principles. It's object-oriented programming (OOP).

It is the best way is to expose the functionality to other or external components in

a way so that any other programming language will be able to use the

functionality without adhering to any specific user interfaces, that is, services

(web services, APIs, rest services, and so on).

The whole system works as per a type of collaboration that is not interconnected

or interdependent.

Every component is liable for its own responsibilities. In other words,

components are responsible for only one functionality.

It segregates code with a separation concept, and segregated code is reusable.

注：我感觉原著这个章节太差劲了，喊了几句口号，这就算解释了工作原理？

1.单体应用架构

当网站流量很小时，只需要将所有功能部署在一起，以减少部署节点和成本

单体架构业务流程往往在同一个进程内部完成处理，不需要进行分布式协作，它的工作原理如下：

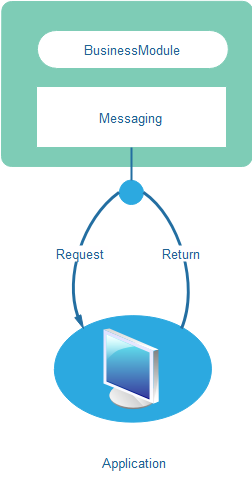
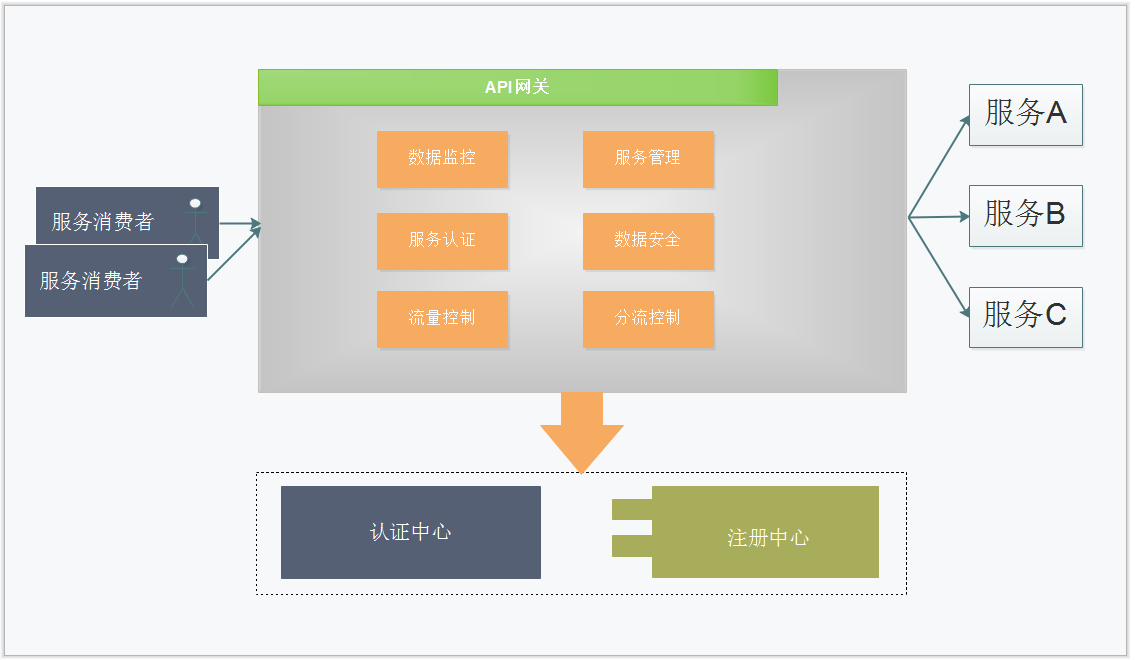


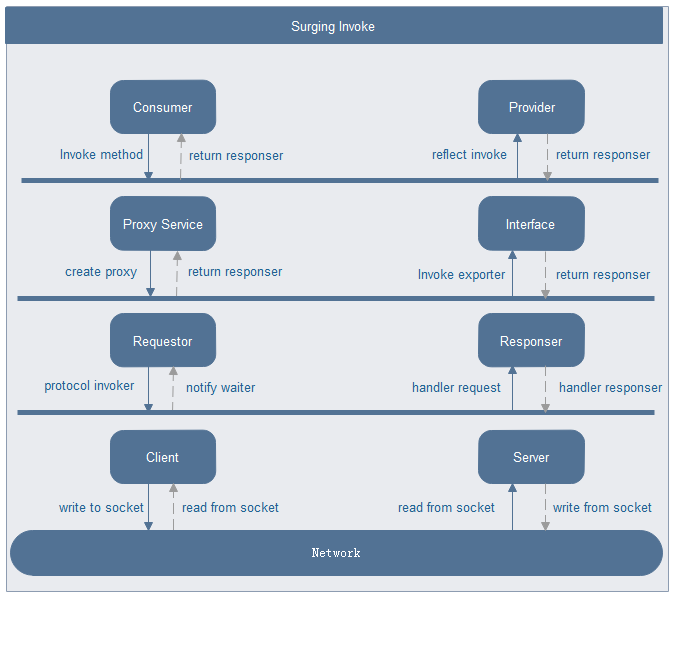
图 1-1 单体架构本地方法调用

2.垂直应用架构

当访问量逐渐增大，单体架构压力越来越大，将架构拆成互不相干的若干应用以提升效率，此时采用MVC、webAPI进行调用

3.分布式微服务架构

当垂直应用越来越多，应用之间交互不可避免，可以将各个独立的业务模块，部署成独立的微服务，逐渐形成稳定的服务中心。而Surging 微服务采用分布式集群部署方式，服务的消费者和提供者通常运行在不同的进程中，进程之间通信采用RPC方式调用，它的工作原理如下：



## 1.5它的优势

现在让我们试着快速了解微服务的先进之处，对比SOA架构和单体架构：

* 扩展的费用更有效的**:**你不需要考虑对整个应用进行扩展的大量投资。就购物车而言，我们可以仅考虑产品目录服务和订单处理服务的负载均衡而不必考虑使用频率很小的服务，例如库存服务、订单取消服务、交付确认服务。
* 清晰的代码边界：边界的划分应该匹配企业的部门层级关系。在大型企业里，由不同部门主导不同软件系统而言，这是一个巨大优势。
* 非常容易进行代码修改：已经完成的代码以一种和其他模块代码没有依赖的方式存在，并且实现了隔离的功能性。如果我们设计合理，那么一个服务中的改变对其他服务的影响非常少。
* 非常容易进行部署：因为整个应用看起来就像是一组彼此隔离的生态系统，如果需要，一次可以部署完成一个服务。这些服务中的任何一个宕机了，都不会影响整个系统的运行。
* 技术实现无关：你可以一夜之间使用不同的技术栈重构任何一个服务或整个系统，而用户根本感觉不到这些，前提是你维护了服务间的接口契约。
* 分布式系统：这是潜在的，需要额外的谨慎在这里。确保您的异步调用使用得当并且同步那些信息并没有真正阻挡整个信息流。使用数据分区好，会在稍后讨论。
* 快速的市场响应：具有竞争力是一个明确的优势；如果在潮流面前，你的系统采用新技术、增加新特性太慢的话，你这些努力还有什么用。

## 1.6不是你想就能用——先决条件？

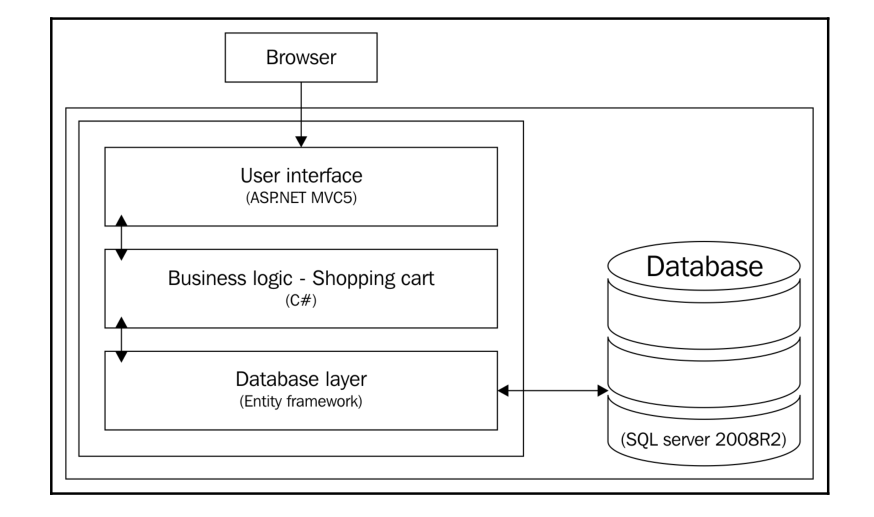
理解采用了微服务架构后导致的开发生态的改变非常重要。微服务的影响不仅仅是在操作之前。所以任何采用了微服务的组织都会有深刻的变化。如果对此没有做好充分的准备，哪怕优势也会变成劣势

采用了微服务后，在适当的时候考虑以下问题是非常明智的

* 部署和QA：需求将变得更加苛刻，更快的发展要求。这将需要你尽快部署和测试。如果只是少量的服务，那么这不会是个问题。但是，如果服务的数量在上升，那么可以很快地挑战现有的基础设施和做法。例如，您的QA和模拟环境可能不再足以支撑测试从开发团队交付产物的数量。
* 开发和运营团队的协作平台：随着应用程序进入公共领域，开发与产品、测试的撕逼大战再次上演并不会很遥远。而这一次的不同之处在于企业的业务将濒临险境。所以，你需要准备好快速反应，如果需要用一种自动化的查找问题根本原因的方式。
* 一个监控框架：随着服务数量的不断增加，你和快就会需要一种方式去检测整个系统的功能性和健康性，以发现任何可能的瓶颈或问题。没有任何手段来监测这些服务的状态和由此产生的业务功能，任何团队都不可能采取积极主动的部署方法

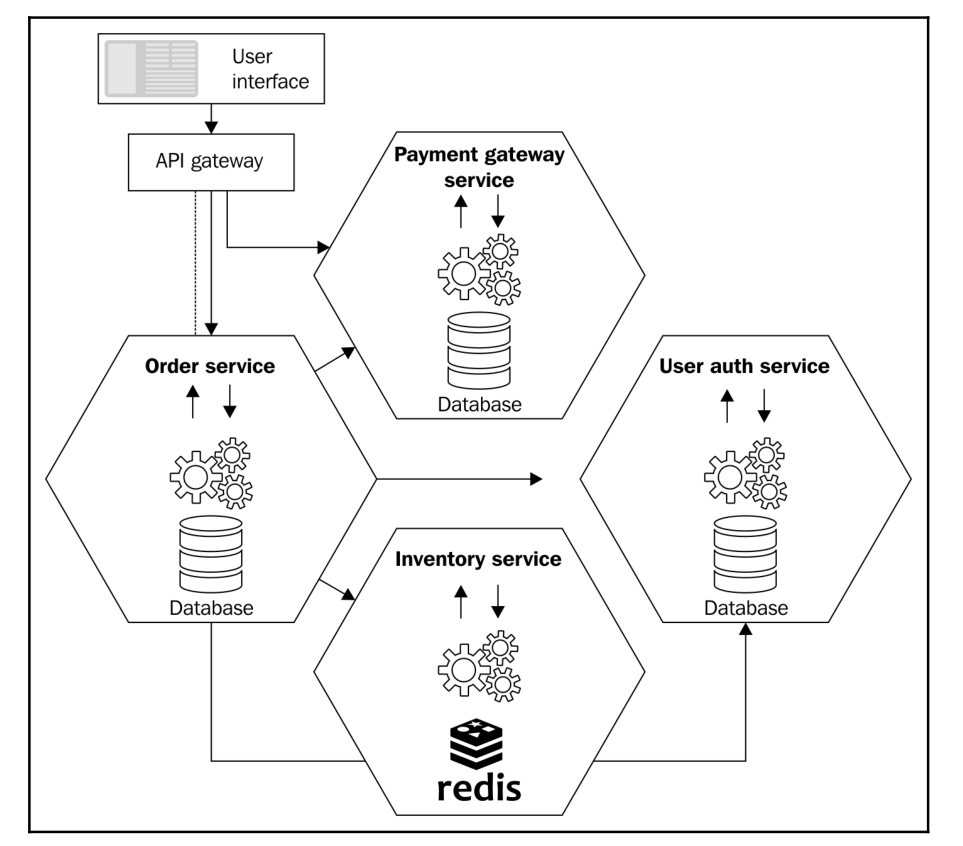
## 1.7微服务投入生产使用前的热身

当你看到这里的时候，想必你的技术能力应该不错，至少应该对多层架构非常熟悉，如果不熟悉，那么你也没有往下看的必要了。

现在而言大多数企业都在使用多层架构，架构图如下：

如果真的要投入生产使用微服务架构，你一定会在心里想，这个架构将要做怎样的修改呢？

结合上面的概念，请你时刻牢记一句话，也就是微服务的精髓——【微服务是一种以多个小服务来构成应用的架构风格，是SOA的升级版】，那么答案就很清晰：



怎么样，和你心里的预期是否吻合呢？此时此刻，你是不是想说，哎呀我擦，微服务就这样？

千真万确，微服务就是这样。你是不是还想问？使用微服务架构后，数据库的事务处理还怎么用？复杂的业务作为服务怎么实现？甚至你足够聪明的话，应该会想到API gateway和服务器以及负载均衡怎么处理啊？数据库库完全拆分开了，那么统计报表和表之间的关系处理怎么写？

那么首先恭喜你，你已经掌握微服务的精髓了，然后这些问题的答案先不告诉你。。。

至于原因嘛，就是这些疑问和微服务没有啥关系。事实上具体的来将，你应该使用过WebService（也就是SOA架构），其内部的服务实现跟你用面向过程和面向对象的代码设计方式没有任何关系，和内部的数据更没有任何关系，架构只解决架构层面的问题，在架构设计下面还有功能模块设计，而这些地方才是设计模式真正发挥作用的地方，才是你的疑问产生和解决的层面，就像你使用HTTP发送一个请求，你会关心底层网络内部的字节传输的问题吗？

若要明白如何解决心理的这些疑问，请继续往下看

# 2实现微服务架构

## 2.1一个服务有多大？

在我们实际构建微服务之前，需要对一些基本方面有清晰的认识。比如定义大小时，哪些因素需要考虑，以及如何保证它与系统其他部分的隔离。

正如它的名字一样，服务应该尽可能的“小”，那么什么是“小”呢？

微服务是关于服务大小和粒度的架构。为了更好理解这一点，让我们用一个传统电商的单体架构拆分为微服务来举例子。

我们首要考虑的是，把系统拆分为多个服务。从微服务顶级的视角来看，需要我们考虑隔离性的因素：

* 来自需求频繁变更的风险：一个服务需求上的变更应该不依赖于其他服务。
* 功能的变更：我们应该隔离极少改变的服务于频繁改变的服务之间的依赖。例如客户管理模块极少改动，但是与其相关的订单模块却在整个生命周期内频繁改动**。**
* 团队的变更：我们也需要考虑隔离模块，用一个团队的工作不会影响其他团队工作的方式。新开发人员处理模块内的任务而不需要团队外部人员的过程，对我们是非常有利的。
* 技术的变更：技术使用的需要在模块内垂直隔离。一个模块在技术和组件的使用上应该不依赖其他模块。我们应该严格隔离使用不同技术开发的模块，或者合并为一个公用部分。

我们最核心的目标并不是机械的让每个服务尽量最小化，而是根据业务需要，把服务隔离到唯一的上下文中并尽量小。

## 2.2好的服务的标准是什么

为了使服务对于开发者和用户更容易使用和维护，服务应该具有以下特征：

* 标准的数据格式：好的服务在与其他服务或系统交互时，应该使用标准的数据格式。例如XML或JSON
* 标准的通信协议：好的服务应该使用标准的通信模式，例如SOAP和REST
* 弱耦合：好的服务重要特征是遵从了强内聚弱耦合的原则，这样服务的变更不会影响其他服务

## 2.3我的最爱—DDD和它对于微服务的重要性

领域驱动设计是一种开发思想体系，它旨在管理为辅助问题领域编写的软件的创建和维护工作。领域驱动设计分为两个阶段：

以一种领域专家、设计人员、开发人员都能理解的通用语言作为相互交流的工具，在交流的过程中发现领域概念，然后将这些概念设计成一个领域模型；

由领域模型驱动软件设计，用代码来实现该领域模型；

由此可见，领域驱动设计的核心是建立正确的领域模型

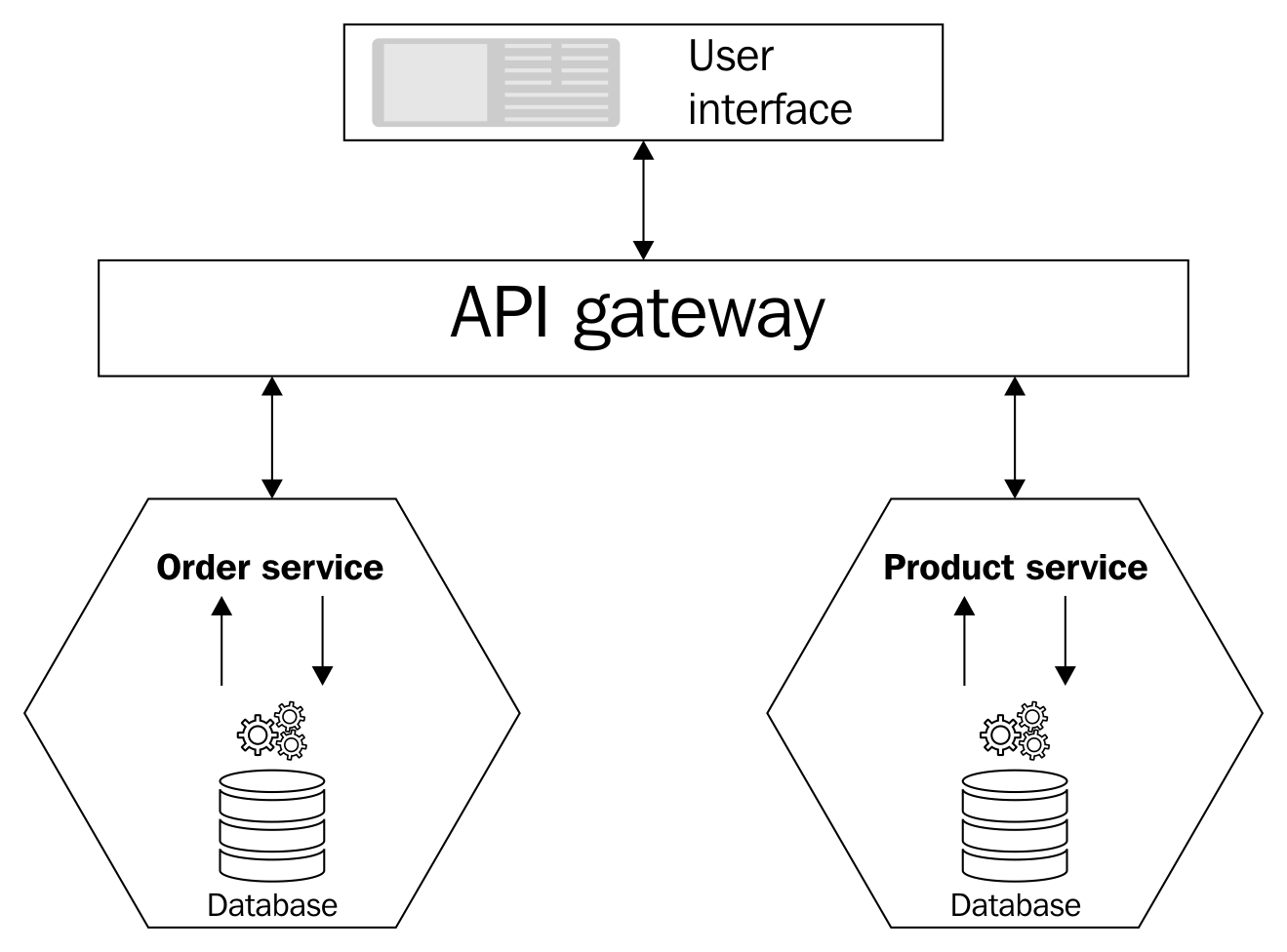
DDD是一个可以被微服务实现的蓝本**。**

## 2.4The concept of seam

## 2.5服务之间的通信

以ORDER 订单 and PRODUCT产品为例，

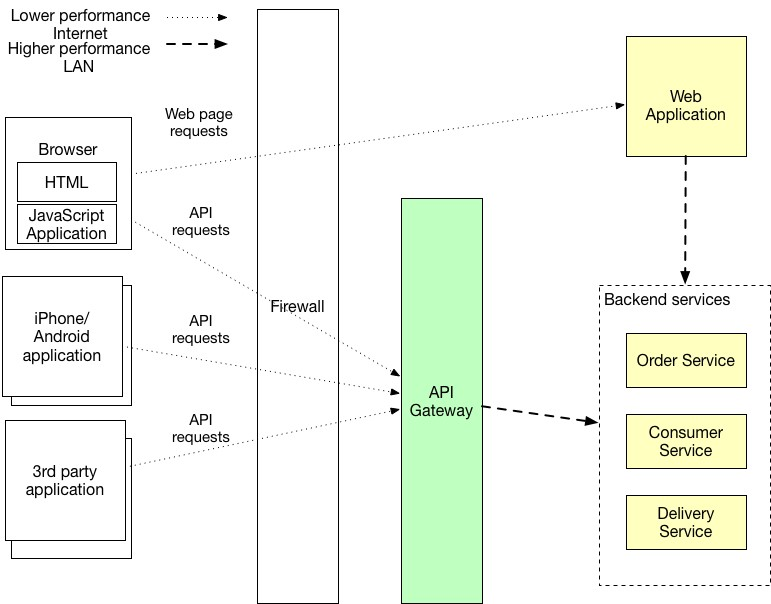
在单体架构下，我们仅使用一个DAL层（或者SQLHelper）来同时处理订单和产品的数据表，而在微服务架构下，我们的Order service and Product service将使用不同的DAL层（比如一个使用ADO，一个使用EF），甚至使用各自独立的数据库（一个Mysql，一个用NoSQL），那么问题来了，以前可以一个DAL层里同时操作两种表，现在怎么办呢？不要怕，肯定有办法，那就是在service层里想办法。又因为每个服务是单独部署的，但是又想统一他们，这个时候就得改变思维，增加一个额外的抽象层，将每个服务组织在一起，这个新加的层，就叫做外观层，由于是以api的形式对外交互，又叫做API网关。如下所示：



具体来说，订单服务你使用一个webapi项目，然后产品服务也使用一个webapi项目，网关使用一个webAPI项目，然后网关仅仅是调用每个服务的接口，然后进行组合数据和交互等等。网关的具体实现技术，有很多框架，比如Ocelot，Kong等，但原理就是这样。

总结：API网关是一个服务，用来转发外部世界的请求到具体的服务。它的职责有路由请求，API组合，授权验证，限流等

完整的示例：



## 2.6Revisiting the Flix One case study

到目前为止，我们讨论了很多理论上的服务分离的技术，是时候开始撸代码了！！！

# 3服务之间的技术集成

## 3.1服务之间的通信

## 3.2套路之——集成模式

### The API gateway

### The event-driven pattern

### Event sourcing

### Eventual consistency

### Compensating transactions

### Competing consumers

### Azure Service Bus

### Azure queues

# 4服务的测试

## 4.1如何进行

## 4.2测试方法

## 4.3Testing pyramid

## 4.3测试的分类

### 4.3.1单元测试

### 4.3.2组件(服务)测试

### 4.3.3集成测试

### 4.3.4契约测试

### 4.3.5性能测试

### 4.3.6端对端(前后端)测试

### Sociable versus isolated unit tests

### 约定与模拟

## 4.4实操演练

# 5微服务的部署

## 5.1部署实施层面的术语

## 5.2成功部署的先决条件

## 5.3部署的隔离需求

## 5.4部署范例

## 5.5容器技术

## 5.6Docker介绍

# 6系统安全性

# 7运行监控

# 8系统扩展性

# 9Introduction to Reactive Microservices

# 10Surging框架的实现解说