我们对于一些事物的不理解或者畏惧，原因都在于这些

事情所有意无意带有的绚丽外衣和神秘面纱。

只要揭开这个外衣，就如一个裸体美女站在你身前，还有

什么好神秘的？还有什么好惊奇的？

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](#_Toc503209681)

[1.1它是啥玩意？ 2](#_Toc503209682)

[1.2深刻理解它 2](#_Toc503209683)

[1.3为啥使用它 2](#_Toc503209684)

[1.4牢记它的工作原理 2](#_Toc503209685)

[1.5它的优势 2](#_Toc503209686)

[1.6不是你想就能用——先决条件？ 3](#_Toc503209687)

[1.7 3](#_Toc503209688)

[2实现微服务架构 3](#_Toc503209689)

[2.1一个服务有多大？ 3](#_Toc503209690)

[2.2好的服务的标准是什么 3](#_Toc503209691)

[2.3我的最爱—DDD和它对于微服务的重要性 3](#_Toc503209692)

[2.4The concept of seam 3](#_Toc503209693)

[2.5服务之间的通信 3](#_Toc503209694)

[2.6Revisiting the Flix One case study 3](#_Toc503209695)

[3服务之间的技术集成 3](#_Toc503209696)

[3.1服务之间的通信 3](#_Toc503209697)

[3.2套路之——集成模式 3](#_Toc503209698)

[The API gateway 3](#_Toc503209699)

[The event-driven pattern 4](#_Toc503209700)

[Event sourcing 4](#_Toc503209701)

[Eventual consistency 4](#_Toc503209702)

[Compensating transactions 4](#_Toc503209703)

[Competing consumers 4](#_Toc503209704)

[Azure Service Bus 4](#_Toc503209705)

[Azure queues 4](#_Toc503209706)

[4服务的测试 4](#_Toc503209707)

[4.1如何进行 4](#_Toc503209708)

[4.2测试方法 4](#_Toc503209709)

[4.3Testing pyramid 4](#_Toc503209710)

[4.3测试的分类 4](#_Toc503209711)

[4.3.1单元测试 4](#_Toc503209712)

[4.3.2组件(服务)测试 4](#_Toc503209713)

[4.3.3集成测试 4](#_Toc503209714)

[4.3.4契约测试 5](#_Toc503209715)

[4.3.5性能测试 5](#_Toc503209716)

[4.3.6端对端(前后端)测试 5](#_Toc503209717)

[Sociable versus isolated unit tests 5](#_Toc503209718)

[约定与模拟 5](#_Toc503209719)

[4.4实操演练 5](#_Toc503209720)

[5微服务的部署 5](#_Toc503209721)

[5.1部署实施层面的术语 5](#_Toc503209722)

[5.2成功部署的先决条件 5](#_Toc503209723)

[5.3部署的隔离需求 5](#_Toc503209724)

[5.4部署范例 5](#_Toc503209725)

[5.5容器技术 5](#_Toc503209726)

[5.6Docker介绍 5](#_Toc503209727)

[6系统安全性 5](#_Toc503209728)

[7运行监控 5](#_Toc503209729)

[8系统扩展性 6](#_Toc503209730)

[9Introduction to Reactive Microservices 6](#_Toc503209731)

[10Surging框架的实现解说 6](#_Toc503209732)

**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？

我用图片来解释**。。。**

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：

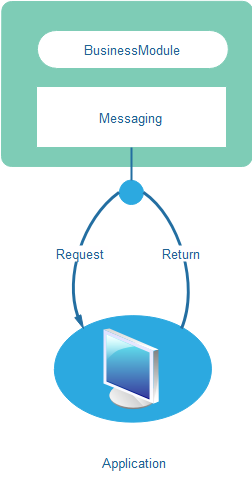


图 1-1 单体架构本地方法调用

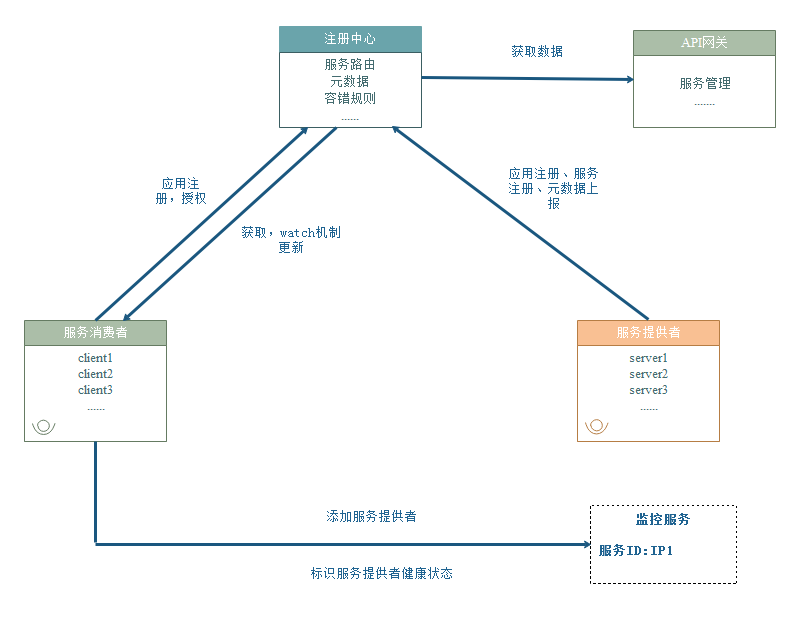
2.垂直应用架构

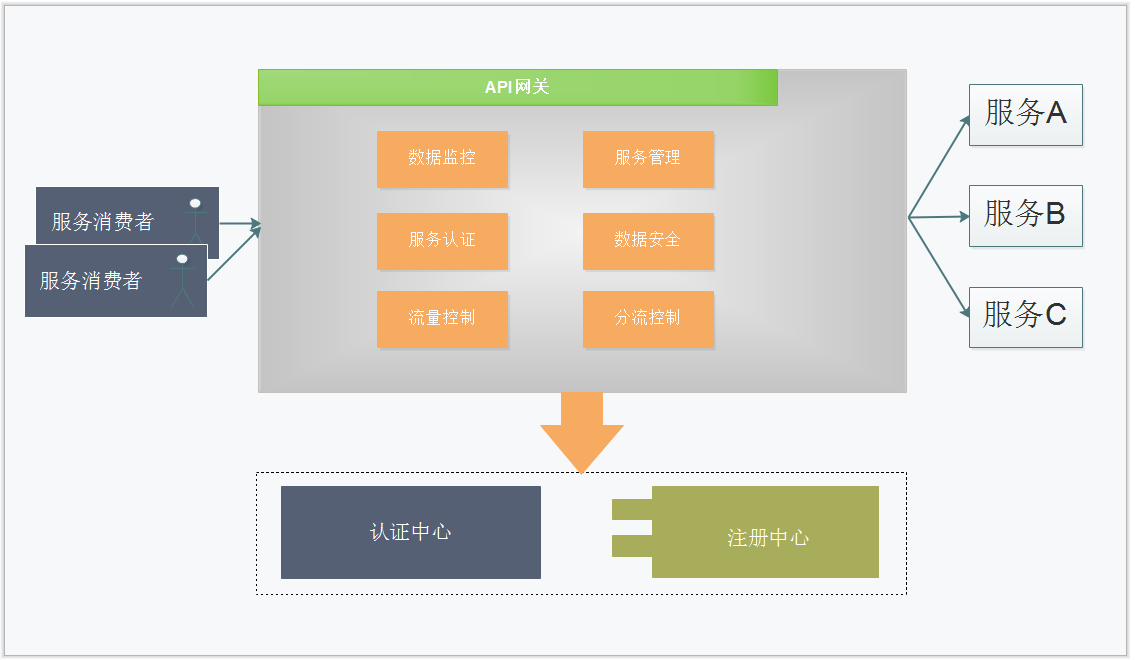
当访问量逐渐增大，单体架构压力越来越大，将架构拆成互不相干的若干应用以提升效率，此时采用MVC、webAPI进行调用

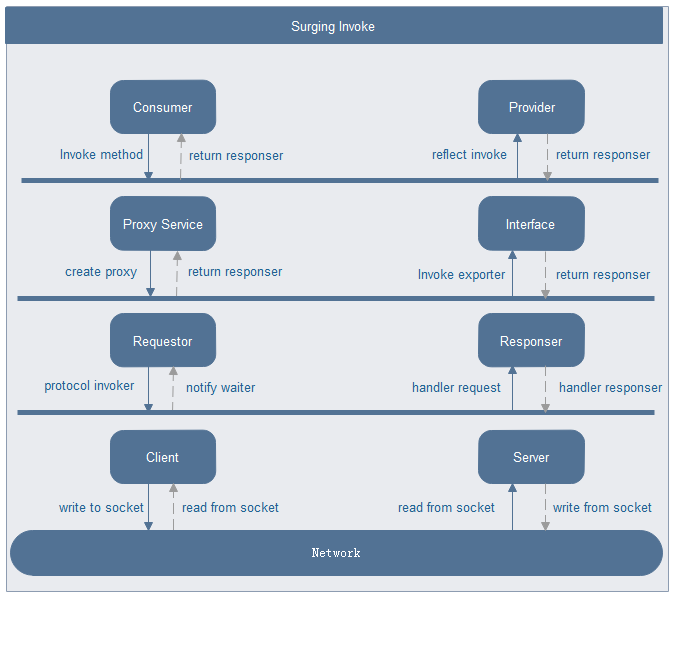
3.分布式微服务架构

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

## 1.6不是你想就能用——先决条件？

## 1.7

# 2实现微服务架构

## 2.1一个服务有多大？

## 2.2好的服务的标准是什么

## 2.3我的最爱—DDD和它对于微服务的重要性

## 2.4The concept of seam

## 2.5服务之间的通信

## 2.6Revisiting the Flix One case study

# 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