面向服务的架构（SOA） 微服务架构

SOA和微服务架构的区别？

添加评论 分享

7 个回答 按投票排序

知乎用户 ，大型电信行业SOA实施经验，SOA规划咨询，…

46 人赞同

谢多人邀请，其实前面几位的回答已经差不多了，在这里仅谈下自己的简单总结。

微服务架构强调的第一个重点就是业务系统需要彻底的组件化和服务化，原有的单个业务系统会拆分为多个可以独立开发，设计，运行和运维的小应用。这些小应用之间通过服务完成交互和集成。每个小应用从前端web ui，到控制层，逻辑层，数据库访问，数据库都完全是独立的一套。在这里我们不用组件而用小应用这个词更加合适，每个小应用除了完成自身本身的业务功能外，重点就是还需要消费外部其它应用暴露的服务，同时自身也将自身的能力朝外部发布为服务。

如果一句话来谈**SOA**和微服务的区别，即微服务不再强调传统**SOA**架构里比较重的**ESB**企业服务总线，同时**SOA**的思想进入到单个业务系统内部实现真正的组件化。

把这个核心搞清楚后，再来看下网上找到的对微服务架构的一些定义和阐述：

> 微服务可以在“自己的程序”中运行，并通过“轻量级设备与**HTTP**型**API**进行沟通”。关键在于该服务可以在自己的程序中运行。通过这一点我们就可以将服务公开与微服务架构（在现有系统中分布一个API）区分开来。在服务公开中，许多服务都可以被内部独立进程所限制。如果其中任何一个服务需要增加某种功能，那么就必须缩小进程范围。在微服务架构中，只需要在特定的某种服务中增加所需功能，而不影响整体进程。
>
> 微服务不需要像普通服务那样成为一种独立的功能或者独立的资源。定义中称，微服务是需要与业务能力相匹配，这种说法完全正确。不幸的是，仍然意味着，如果能力模型粒度的设计是错误的，那么，我们就必须付出很多代价。如果你阅读了Fowler的整篇文章，你会发现，其中的指导建议是非常实用的。在决定将所有组件组合到一起时，开发人员需要非常确信这些组件都会有所改变，并且规模也会发生变化。服务粒度越粗，就越难以符合规定原则。服务粒度越细，就越能够灵活地降低变化和负载所带来的影响。然而，利弊之间的权衡过程是非常复杂的，我们要在配置和资金模型的基础上考虑到基础设施的成本问题。

再强调下即：

首先对于应用本身暴露出来的服务，是和应用一起部署的，即服务本身并不单独部署，服务本身就是业务组件已有的接口能力发布和暴露出来的。了解到这点我们就看到一个关键，即我们在进行单个应用组件设计的时候，本身在组件内部就会有很大接口的设计和定义，那么这些接口我们可以根据和外部其它组件协同的需要将其发布为微服务，而如果不需要对外协同我们完全可以走内部API接口访问模式提高效率。

其次，微服务架构本身来源于互联网的思路，因此组件对外发布的服务强调了采用**HTTP Rest API**的方式来进行。这个也可以看到在互联网开放能力服务平台基本都采用了Http API的方式进行服务的发布和管理。从这个角度来说，组件超外部暴露的能力才需要发布为微服务，其本身也是一种封装后的粗粒度服务。而不是将组件内部的所有业务规则和逻辑，组件本身的底层数据库CRUD操作全部朝外部发布。否则将极大的增加服务的梳理而难以进行整体服务管控和治理。

微服务的基本思想在于考虑围绕着业务领域组件来创建应用，这些就应用可独立地进行开发、管理和加速。在分散的组件中使用微服务云架构和平台使部署、管理和服务功能交付变得更加简单。

对于互联网谈到微服务架构一定会谈到Devops即开发测试和部署运维的一体化。当我们的单体应用以及拆分为多个小应用后，虽然整体架构可以松耦合和可扩展，但是如果拆分的组件越多，这些组件之间本身的集成和部署运维就越复杂。即任何一个组件，当他依赖的外部其它应用组件越多的时候，整个集成，部署和联调测试的过程就越复杂。这些如果完全靠我们手工去完成一是增加工作量，一是增加出错概率。

原来谈组件化开发谈的最多的是单个组件的持续集成，包括配置环境集成，自动打包部署，自动化的冒烟测试等。对于微服务架构下首先仍然是要做好单个组件本身的持续集成，其次在这个基础上增加了多个组件的打包部署和组件间的集成。里面的核心思想就是Devops的思路，希望能够实现开发设计到部署运维的一体化。

由于微服务架构里面强调了单个组件本身是可以在独立的进程里面运行，各个组件之间在部署的时候就能够做到进程级别的隔离。那么一台服务器我们可能需要初始化几十个甚至更多的进程来进行应用组件的部署。为了保持进程的隔离性，我们可以用虚拟机，但是当几十个进程都完全用独立的虚拟机就不现实的，而这个问题的解决刚好就是利用PaaS平台里面的轻量Docker容器来做这个事情，每个Docker是独立的 46 刚好又完全做到进程级别的隔离，资源占用率又最小，这些特点刚好满足微服务架构的开发测试和自动化部署。

前面这些问题思考清楚后就是考虑所有暴露的微服务是否需要一个统一的服务管控和治理平台，按照当前微服务架构的整体思路，虽然单个服务的实现和发布仍然是在组件内部完成的，但是这些组件暴露的服务本身的调用情况，服务本身的安全，日志和流量控制等仍然需要一个统一的**SOA**服务管理平台来完成。

由于微服务尽量都是通过HTTP API的方式暴露出去的，因此这种服务管理平台不需要像传统企业内部的ESB服务总线这么重。但是最基本的服务注册，服务代理，服务发布，服务简单的路由，安全访问和授权，服务调用消息和日志记录这些功能还是需要具备。类似淘宝的Dubbo架构，即可以做为微服务架构下的服务管控平台。

对于这种服务管控平台，核心需要讨论的就是服务每次调用本身的消息传递，输入和输出日志是否需要记录，当前就有两种做法，一种是不记录，管理平台只负责服务注册和目录发布，安全授权，实际的服务访问仍然是两个组件之间的点对点连接完成，这种方式下整个架构下获取更高的性能，同时服务管理平台也不容易成为大并发服务访问下的单点瓶颈；另外一种方式就是完全记录，在这种方式下就需要考虑服务管理平台本身的集群化不是，高并发下的性能问题。而个人建议最好的方式还是SOA服务管理平台应该提供两种管理能力，同时仅仅对核心的需要Log日志的服务进行日志记录，而其它服务只提供服务目录和访问控制即可。

==========2016.6.8日更新，增加Chris Richardson微服务系列读书笔记

本文为阅读《Chris Richardson 微服务系列》的阅读笔记，具体原文参考：「Chris Richardson 微服务系列」服务发现的可行方案以及实践案例 ， 里面有另外四篇的链接，当前daocloud已经更新到第5篇事件驱动架构。

第一篇 微服务架构的优势和不足

文中强调的单体应用的场景，我在前面很多谈组件化和服务化的文章里面已经都谈到过了，即一个应用系统里面的

模块没有办法做到彻底解耦，如果要实现按组件单独部署是不可能的，相互之间仍然有大量内部不可见依赖而导致了模块间无法拆分。

那么单体应用本身带来的问题主要有哪些？

1.系统复杂：内部多个模块紧耦合，关联依赖复杂，牵一发而动全身。
2.运维困难：变更或升级的影响分析困难，任何一个小修改都可能导致单体应用整体运行出现故障。
3.无法扩展：无法拆分部署，出现性能瓶颈后往往只能够增加服务器或增加集群节点，但是DB问题难解决

正是由于这些原因需要考虑引入微服务架构（实质仍然是单个应用本身的组件化和服务化），对于微服务文章里面有一个详细说明如下：一个微服务一般完成某个特定的功能，比如订单管理、客户管理等。每个微服务都是一个微型应用，有着自己六边形架构，包括商业逻辑和各种接口。有的微服务通过暴露 **API** 被别的微服务或者应用客户端所用；有的微服务则通过网页 **UI** 实现。在运行时，每个实例通常是一个云虚拟机或者 **Docker** 容器。

从这个定义和说明仍然需要做一些关键理解，即在我前面谈微服务的文章里面谈到过的，即核心的几点包括了，其一足够构成一个独立小应用（从**DB**到**UI**），其二微服务应用之间只能通过**Service API**进行交互，其三一般运行在云虚拟机或更轻的**Docker**容器上。

API Gateway，这实际上微服务架构里面的很重要的内容，其作用类似于传统企业内部的ESB服务总线，只是更加轻量和高性能来解决微服务的管控和治理问题。而对于负载均衡，缓存，路由，访问控制，服务代理，监控，日志等都属于基本的服务管控内容，也是API Gateway需要考虑的核心能力。

Scale Cube的3D模型，描述的相当好，即通过微服务架构实施后扩展性的[illegible]

1. Y轴：本质是应用的分解，即将传统的单体应用分解为多个微服务应用。
2. X轴：水平弹性扩展能力，即通过负载均衡来实现水平弹性扩展，但是DB问题无法解决，引入3
3. Z轴：当单个微服务应用引入了DB弹性扩展能力要解决的时候，我们引入了对数据库进行拆分和DaaS

对于微服务架构的好处前面在讲单体应用的问题的时候已经谈到了，微服务架构正好是解决这些问题。而对于微服务架构的不足，简单总结如下：

1. CAP原则：由于服务无状态和引入了分布式，较难解决事务一致性问题。
2. 集成复杂：任何彻底的分解都将带来集成的复杂度，即模块在集成时候需要外部微服务模块更多的配合。
3. 部署问题：稍大项目都涉及到上100个服务节点部署，还涉及到部署后的配置，扩展和监控问题。

第二篇 使用**API**网关构建微服务

首先说下这篇文章的引入场景，以一个亚马逊购物网站的手机APP订单查看界面来举例，如果是一个单体应用，那么所有的界面需要获取信息都通过单体应用统一一个地址提供的多个Service API获取。但是转变为微服务架构后可以看到对于会员管理，商品管理，订单管理，财务结算管理等都已经拆分为了不同的微服务模块，需要从不同的服务提供地址调用不同的微服务模块提供的Service API来返回数据。

在原文里面我们看到对于客户端和微服务模块间点对点直接通讯提了三个问题，如下：

1. 问题一：客户端需求和每个微服务暴露的细粒度 API 不匹配
2. 问题二：部分服务使用的协议对 web 并不友好，如二进制RPC或AMQP消息等。
3. 问题三：会使得微服务难以重构，如服务拆分或服务组合的场景。

那么我们从传统的ESB能力来对上面三个问题进行一个说明，第一个问题即可能涉及到细粒度的API组合，类似组合服务无法做；其二是可能存在协议转换的问 题要解决；其三即服务透明的问题，即需要对客户端提供一个统一的服务目录以使底层服务透明。由于以上问题，引入了API服务网关的概念，再次强调，对于**API**服务网关即使微服务架构里面的轻量服务总线，解决服务管控和治理相关问题。文中对API Gateway给出如下说明：

API 网关是一个服务器，也可以说是进入系统的唯一节点。这与面向对象设计模式中的 Facade 模式很像。API 网关封装内部系统的架构，并且提供 API 给各个客户端。它还可能还具备授权、监控、负载均衡、缓存、请求分片和管理、静态响应处理等功能。

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 网关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。以产品详情的场景为例，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。

API网关的优点和缺点

对于API网关的优点，其实是类似传统ESB企业服务总线的优点，即实现服务透明，同时对于服务运行过程中的日志，安全，路由，缓存等问题进行统一配置和处理，而不需要每个微服务分别时都去考虑。如开源的Dubbo服务总线即可以看作是一个API网关的实现。

API网关和ESB的一些重要区别点在于API网关更加轻量和高性能，它不需要去考虑太多遗留系统和诸多协议的适配，其次也不需要考虑服务集成过程中的大 量数据转换和映射。同时为了提升服务网关的性能，一般API网关在实现过程中不会去记录详细的数据传输日志，或者类似Dubbo架构数据传输根本就不会通 过API网关。

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。

简单来说，在我们期望的去中心化和全分布式架构中，网关又成了一个中心点或瓶颈点，正是由于这个原因我们在网关设计的时候必须考虑即使**API Gateway**宕机也不要影响到服务的调用和运行。

API网关的设计和实现

对于大多数应用程序而言，API 网关的性能和可扩展性都非常重要。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，它是一个基于 Chrome JavaScript 引擎构建的平台。

另一个方法是使用 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。

对于**API**网关需要实现底层多个细粒度的**API**组合的场景，文章推荐采用响应式编程模型进行而不是传统的异步回

调方法组合代码。其原因除了采用回调方式导致的代码混乱外，还有就是对于**API**组合本身可能存在并行或先后调用，对于采用回调方式往往很难控制。

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。

注：如果服务是同步调用可以看到微服务模块之间本身是没有彻底解耦的，即如果**A**依赖**B**提供的**API**，如果**B**提供的服务不可用将直接影响到**A**不可用。除非同步服务调用在**API**网关层或客户端做了相应的缓存。因此为了彻底解耦，在微服务调用上更建议选择异步方式进行。

对于大多数基于微服务的应用程序而言，实现 API 网关，将其作为系统的唯一入口很有必要。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。

第三篇 微服务架构中的进程间通信

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。

对于微服务架构的交互模式，文章从两个维度进行了描述，即

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。

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。

对于分为这两个维度进行描述意义不太大，对于同步模式往往只能是1对1，而且还需要同步等待容易引起阻塞，而对于异步模块往往采用消息机制来实现，同时配合消息中间件可以进一步实现消息的发布订阅。而对于**EDA**事件驱动架构要看到其本质也是伊布　消息中间件和消息的发布订阅。

异步消息机制可以做到最大化的解耦，对于数据CUD的场景可以看到是比较容易通过异步消息机制实现的，但是会进一步引入事务一致性问题，即在采用异步消息 机制后往往通过BASE事务最终一致性来解决事务层面的问题。而对于查询功能可以看到是比较难通过异步消息API实现的，在引入这个之前可以看到需要考虑 两方面的问题并解决。

其一是服务网关需要有数据缓存能力，以解决无法从源端获取数据的场景。其二是前端开发框架本身需要支持异步调用和数据装载模式，特别是对于数据查询功能对于用户来讲，在前端的感受仍然需要时同步的。即通过异步方式返回了查询数据后可以动态刷新前端展示界面。

服务版本的问题：这是不可避免要遇到的问题，特别是对于RestAPI调用，由于Json格式本身无Schema返回更加容易忽视了对服务 版本的管理和控制。要知道对于Json数据格式变化仍然会导致RestAPI调用后处理失败。因此服务版本仍然采用大小版本管理机制比较好，对于小版本变 更则直接对原有服务进行覆盖同时对所有受影响的服务消费端进行升级；而对于大版本升级则本质是新增加了一个新服务，而对于旧版本服务逐步迁移和替代。

处理局部失败：文中提到了Netfilix的服务解决方案，对于失败问题的解决要注意常用的仍然是服务超时设置，断路器机制，流量控制，缓存数据或默认值返回等。不论采用哪种失败处理策略，都需要考虑应该尽量减少服务调用失败或超时对最终用户造成的影响。

基于请求/响应的同步 **IPC**

使用同步的、基于请求/响应的 IPC 机制的时候，客户端向服务端发送请求，服务端处理请求并返回响应。一些客户端会由于等待服务端响应而被阻塞，而另外一些客户端可能使用异步的、基于事件驱动的客户端代码，这些代码可能通过 Future 或者 Rx Observable 封装。然而，与使用消息机制不同，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**。

Thrift 也能够让你选择传输协议，包括原始 TCP 和 HTTP。原始 TCP 比 HTTP 更高效，然而 HTTP 对于防火墙、浏览器和使用者来说更友好。文中对于两种实现方式已经描述的相当详细，可以看到当前互联网**OpenAPI**平台和微服务架构实现中仍然是大量以采用**Rest API**接口为主。

而对于消息格式的选择，可以看到在使用**RestAPI**接口的时候，更多的是采用了**Json**消息格式而非**XML**，对于**SOAP WebService**则更多采用了**XML**消息格式。如果采用**Thrift**则还可以采用二进制消息格式以提升性能。

第四篇 服务发现的可行方案以及实践案例

首先还是先说场景，看似简单的服务注册和服务目录库管理为何会变复杂 46 要的原因还是在结合了云端PaaS和Docker容器部署后，对于微服务模块部 署完成后提供出来的IP地址是动态 化的，包括模块在进行动态集群扩展的时候也需要动态接入新的服务提供IP地址。正是由于这个原因引入了服 和管 理的困难度。

在文章中提到了两种服务发现模式，即客户端发现模式和服务端发现模式，分开描述如下：

服务客户端发现模式

使用客户端发现模式时，客户端决定相应服务实例的网络位置，并且对请求实现负载均衡。客户端查询服务注册表，后者是一个可用服务实例的数据库；然后使用负 载均衡算法从中选择一个实例，并发出请求。客户端从服务注册服务中查询，其中是所有可用服务实例的库。客户端使用负载均衡算法从多个服务实例中选择出一 个，然后发出请求。

注：这是类似Dubbo实现机制一样的两阶段模式，即任何一个服务的消费都需要分两个步骤进行，第一步首先是访问服务注册库（更多是**API GateWay**提供的一个能力）返回一个已经动态均衡后的服务可用地址，第二步即客户端和该地址直接建立连接进行服务消费和访问。

在这种模式的实现中有两个重点，其一是动态负载均衡算法，其二是服务网关需要能够对原始服务提供点进行实时的心跳检测以确定服务提供的可用性。

Netflix OSS 是客户端发现模式的绝佳范例。Netflix Eureka 是一个服务注册表，为服务实例注册管理和查询可用实例提供了 REST API 接口。Netflix Ribbon 是 IPC 客户端，与 Eureka 一起实现对请求的负载均衡。

缺点：底层的**IP**虽然动态提供出去了，但是最终仍然暴露给了服务消费方，再需要进一步做安全和防火墙隔离的场景下显然是不能满足要求的。

服务端发现模式

客户端通过负载均衡器向某个服务提出请求，负载均衡器查询服务注册表，并将请求转发到可用的服务实例。如同客户端发现，服务实例在服务注册表中注册或注销。在原文中有图示，基本看图就清楚了，即在服务注册库前新增加了一个Load Balancer节点。注：这两个节点感觉是可以合并到**API GateWay**的能力中去的。

服务端发现模式兼具优缺点。它最大的优点是客户端无需关注发现的细节，只需要简单地向负载均衡器发送请求，这减少了编程语言框架需要完成的发现逻辑。并且 如上文所述，某些部署环境免费提供这一功能。这种模式也有缺点。除非负载均衡器由部署环境提供，否则会成为一个需要配置和管理的高可用系统组件。

服务注册表

服务注册表需要高可用而且随时更新。客户端能够缓存从服务注册表中获取的网络地址，然而，这些信息最终会过时，客户端也就无法发现服务实例。因此，服务注册表会包含若干服务端，使用复制协议保持一致性。

首先可以看到服务注册表本身不能是单点，否则存在单点故障，当服务注册表有多台服务器的时候同时需要考虑服务注册库信息在多台机器上的实时同步和一致。我们操作和配置服务注册信息的时候往往只会在一个统一的服务管控端完成。

其次如果服务注册服务器宕机是否一定影响到服务本身的消费和调用，如果考虑更高的整体架构可用性，还可以设计对于服务注册库信息在客户端本地进行缓存，当服务注册表无法访问的时候可以临时读取本地缓存的服务注册库信息并发起服务访问请求。

46

对于服务注册表，文章提供了三种选择，感觉最常用的实现仍然是基于Zo er进行的。

Etcd – 高可用、分布式、一致性的键值存储，用于共享配置和服务发现。
Consul – 发现和配置的服务，提供 API 实现客户端注册和发现服务。
Apache ZooKeeper – 被分布式应用广泛使用的高性能协调服务。

如前所述，服务实例必须在注册表中注册和注销。注册和注销有两种不同的方法。方法一是服务实例自己注册，也叫自注册模式（self-registration pattern）；另一种是采用管理服务实例注册的其它系统组件，即第三方注册模式。（原文有详细机制描述，不再累述）

虽然方法一把服务实例和服务注册表耦合，必须在每个编程语言和框架内实现注册代码。但是在自己实现完整微服务架构中，考虑到PaaS平台下微服务模块的动 态部署和扩展，采用方法1相当来说更加容易实现。但是方法1仍然不能代替服务注册库本身应该具备的服务节点的心跳检测能力。

编辑于 2016-06-08　5 条评论　感谢　分享　收藏 • 没有帮助 • 举报 • 作者保留权利

扬尘 ，没有it的智商，却强行与it有关！dota 羽…

16

16 人赞同

潜水知乎多年，今天把我知道的说出来，希望高手多多提意见。。。。。。。。。。。。。。。。
这半年的实习，接触了分布式集群处理和自动化运维监控，我已从传统的单机处理的理念转变到分布式处理，我们处理了很多实际的问题，发现虽然分布式非常吃硬件，但是运行的效率和时间却是大大的缩减了，例如我们分析处理无锡市九个卡口一年的交通数据，这个数据量大家估计都是懂得，对这些数据的处理仅仅就用了两天！因此我当时坚定不移的确定，以后的程序处理和业务的处理必然是往大集群、分布式方向发展的！扯了这么多没用的，关于微服务这个，也是这几天刚刚开始接触（老大出差一回来跟我们说的）。
SOA：面向服务架构，java级企业开发的首选。

微服务：采用一组服务的方式来构建一个应用，服务独立部署在不同的进程中，不同服务通过一些轻量级交互机制来通信，例如 RPC、HTTP 等，服务可独立扩展伸缩，每个服务定义了明确的边界，不同的服务甚至可以采用不同的编程语言来实现，由独立的团队来维护。简单的来说，一个系统的不同模块转变成不同的服务！而且服务可以使用不同的技术加以实现！

SOA我想我就不用细说了，使用java EE开发过的都知道其中的具体流程和步骤，J2EE的确在统一开发中有着很好的使用，但是随着模块功能的不断扩展或者变动，对于其中一点的维护可能会影响到整体的项目。

所以有了微服务，彻底的将耦合性再次的降低了。为什么要讲我实习呢，因为微服务的使用会随着单一进程的传统应用被拆分为一系列的多进程服务后，意味着开发、调试、测试、集成、监控和发布的复杂度都会相应增大。 必须要有合适的自动化基础设施来支持微服务架构模式，否则开发、运维成本将大大增加。所以自动化运维是十分必要的。

总的说来，微服务有以下几个特征：**1.** 通过服务实现组件化；**2.** 按业务能力来划分服务与组织团队；**3.** 服务即产品；**4.** 智能终端与哑管道；**5.** 去中心统一化；**6.** 基础设施自动化；**7. Design for failure**；**8.** 进化设计

具体大家可以参考：**blog.csdn.net/mindfloat...**

发布于 2015-12-24　2 条评论　感谢　分享　收藏 • 没有帮助 • 举报 • 作者保留权利

用心阁 ，软件工程师

25

25 人赞同

SOA

SOA的提出是在企业计算领域，就是要将紧耦合的系统，划分为面向业务的，粗粒度，松耦合，无状态的服务。服务发布出来供其他服务调用，一组互相依赖的服务就构成了SOA架构下的系统。

基于这些基础的服务，可以将业务过程用类似BPEL流程的方式编排起来，BPEL反映的是业务处理的过程，这些过程对于业务人员更为直观，调整也比hardcode的代码更容易。

46

当然企业还需要对服务治理，比如服务注册库，监控管理等。

我们知道企业计算领域，如果不是交易系统的话，并发量都不是很大的，所以大多数情况下，一台服务器就容纳将许许多多的服务，这些服务采用统一的基础设施，可能都运行在一个应用服务器的进程中。虽然说是面向服务了，但还是单一的系统。

微服务

而微服务架构大体是从互联网企业兴起的，由于大规模用户，对分布式系统的要求很高，如果像企业计算那样的系统，伸缩就需要多个容纳续续多多的服务的系统实例，前面通过负载均衡使得多个系统成为一个集群。

但这是很不方便的，互联网企业迭代的周期很短，一周可能发布一个版本，甚至可能每天一个版本，而不同的子系统的发布周期是不一样的。

而且，不同的子系统也不像原来企业计算那样采用集中式的存储，使用昂贵的Oracle存储整个系统的数据，二是使用MongoDB，HBase，Cassandra等NOSQL数据库和Redis，memcache等分布式缓存。

那么就倾向采用以子系统为分割，不同的子系统采用自己的架构，那么各个服务运行自己的Web容器中，当需要增加计算能力的时候，只需要增加这个子系统或服务的实例就好了，当升级的时候，可以不影响别的子系统。这种组织方式大体上就被称作微服务架构。

微服务与SOA相比，更强调分布式系统的特性，比如横向伸缩性，服务发现，负载均衡，故障转移，高可用。互联网开发对服务治理提出了更多的要求，比如多版本，比如灰度升级，比如服务降级，比如分布式跟踪，这些都是在SOA实践中重视不够的。

Docker容器技术的出现，为微服务提供了更便利的条件，比如更小的部署单元，每个服务可以通过类似Node.js或Spring Boot的技术跑在自己的进程中。可能在几十台计算机中运行成千上万个Docker容器，每个容器都运行着服务的一个实例。随时可以增加某个服务的实例数，或者某个实例崩溃后，在其他的计算机上再创建该服务的新的实例。

这就是我对SOA和微服务架构区别的一点理解。

发布于 2016-01-12　3 条评论　感谢　分享　收藏 • 没有帮助 • 举报 • 作者保留权利

徐兵元 ，IT民工，关注SOA、云计算、电力企业信息...

9 人赞同

首先，可以肯定的是SOA和微服务的确是一脉相承的，大神Martin Fowler提出来这一概念可以说把SOA的理念继续升华，精进了一步。其核心思想是在应用开发领域，使用一系列微小服务来实现单个应用的方式途径，或者说微服务的目的是有效的拆分应用，实现敏捷开发和部署 ，可以是使用不同的编程语言编写。而SOA可能包含的意义更泛一些，更不准确一些。

其次，从实现方式上，两者都是中立性，语言无关，协议跨平台，相比SOA，微服务框架将能够带来更大的敏捷性，并为你构建应用提供更轻量级、更高效率的开发。而SOA更适合大型企业中的业务过程编排、应用集成。另外还有微服务甚至是去**ESB**、去中心化、分布式的，而SOA还是以ESB为核心，大量的WS标准实现。

再次，从服务粒度上，既然是微，必然微服务更倡导服务的细粒度，重用组合，甚至是每个操作（或方法）都是独立开发的服务，足够小到不能再进行拆分。而SOA没有这么极致的要求，只需要接口契约的规范化，内部实现可以更粗粒度，微服务更多为了可扩充性、负载均衡以及提高吞吐量而去分解 但同时也引发了打破数据模型以及维护一致性的问题。

最后，从部署方式上，这个是最大的不同，对比Monolithic（有人翻译为单体）的Java EE部署架构，通过展现层打包WARs，业务层划分到JARs最后部署为EAR一个大包，而微服务则打开了这个黑盒子，把应用拆分成为一个一个的单个服务，应用**Docker**技术，不依赖任何服务器和数据模型，是一个 全栈应用，可以通过自动化方式独立部署，每个服务运行在自己的进程中，通过轻量的通讯机制联系，经常是基于HTTP资源API，这些服务基于业务能力构建，能实现集中化管理（因为服务太多啦，不集中管理就无法DevOps啦）。

发布于 2016-03-16　2 条评论　感谢　分享　收藏 • 没有帮助 • 举报 • 作者保留权利

---

waiting hello ，IT

6 人赞同

先说明观点：SOA与微服务不存在冲突问题，他们是相辅相承的！ SOA不是针对某一个系统的架构设计，他更多是针对整个企业的架构设计，他所解决的是更高层的整个企业各业务领域的整合问题。其实微服 务的概念非常像早期的Cobra，微服务是针对某单一业务功能的原子化架构设计，是解决下层具体业务实现的架构设计方法。微服务的颗粒度要求是一个funtion（）这样的级别，一个单独的funtion是不能匹配业务的，那么肯定需要在其上层有一个实现调度的东西，Api Gateway就被提出来了。Api Gateway的主要作用是管理api,调度，转换，拼装，这这些是不是很熟悉？没错，它就是SOA架构中的核心:企业服务总线的作用。因此，微服务负责访问数据库封装成服务，服务被注册到esb，在esb中被转换和拼装，然后再给界面层使用。

手机码字太累，先说那么多。

发布于 2016-01-19　1 条评论　感谢　分享　收藏 • 没有帮助 • 举报 • 作者保留权利

知乎用户 ，zhangwenbo.net

1 人赞同

segmentfault.com/a/1190...

发布于 2016-06-17　添加评论　感谢　分享　收藏 • 没有帮助 • 举报 • 作者保留权利

0

张振 ，将装逼进行到底~~~~

以一个公司为例：有基层员工 有管理层 有老板，最初大家都听老板指挥，谁干什么谁干什么，根据需要，你可能今天干A事情，明天干B事情，后来人越来越多了，事情也越来越多了，做事情的效率越来越多，管理也很混乱，就开始做部门划分（服务化），专门部门做专门事情的，IT部门只做研发，人事部门只做招聘； 这个时候就无法避免的发生跨部门协作（服务器调用）， 但是你怎么知道有这样一个部门可以做这个事情呢，就要依赖行政部门（注册中心），新成立的部门要在行政哪里做一个备案（服务注册），然后公布一下，让其他部门知道了（服务发布），大家就可以在新的工作秩序里面嗨皮的上班了，这个时候依然是在公司的组织架构中运转；
上述就是我理解的SOA的概念

微服务没有具体的实施过，通过自己的一些理解尝试解释一下，勿喷！
微服务有一定SOA的概念在里面，只是在粒度中，微服务更加下一点，比如说用户业务服务：登录 注册 个人中心 包含3个业务，都有userService 提供的，但是在微服务中，登录会被独立出来一个服务，注册也会被独立出来，相对SOA的粒度更新，业务场景耦合更低；

另外微服务强调一个去中心化，上述的公司的组织架构会被打散，没有老板，没有管理层，每一个人都是一个服务，做着自己的事情，虽然没有完全想明白，把自己的理解放出来，大家可以探讨一下

发布于 2016-07-01 添加评论 感谢 分享 收藏 • 没有帮助 • 举报 • 作者保留权利

我来回答这个问题

46

写回答…

我要回答

关注问题 304 人关注该问题

相关问题 换一换

云计算在传统企业级应用及政府信息化中的应用思路？ 6 个回答

国内SOA和BPM的结合产品，哪家强？ 7 个回答

如何判断一个软件是否是建立在真正意义上的SOA架构风格上的？（是架构风格而不是架构） 7 个回答

没有大规模分布式高负载并发系统的工作经验如何社招进入阿里或京东？ 12 个回答

可否推荐一些SOA方面的经典技术书籍？ 3 个回答

46