云环境中 Web 应用的微服务架构评估①

王纪军, 张 斌, 顾永生, 高沈刚

(江苏电力信息技术有限公司, 南京 210024)

摘 要: 云计算为我们提供了一种全新、高效的方式来部署可扩展的 Web 应用, 这种方式使企业的应用可以按需对计算资源进行分配. 微服务架构用于将庞大复杂的应用系统拆分为一系列可独立开发、测试、部署、运行、升级的服务模块. 微服务架构为大批互联网企业实现云环境中的应用扩展、降低应用开发复杂度、实现敏捷开发提供了更加有效地方法. 本文分析并测试了微服务架构模式, 通过一个具体案例--在云环境中开发和部署的企业级应用系统, 对两种架构模式实现(单体架构模式和微服务架构模式)进行性能测试, 得出评估结果, 这些结果对解决企业级应用微服务化中可能遇到的问题具有一定指导意义.

关键词: 云平台; 微服务; 持续交付; 可扩展应用

Evaluation of Micro-Service Architecture for Web Application in the Cloud

WANG Ji-Jun, ZHANG Bin, GU Yong-Sheng, GAO Shen-Gang

(Jiangsu Electric Power Information Technology Co. Ltd., Nanjing 210024, China)

Abstract: Cloud computing provides a pretty new and efficient way to deploy scalable web application. In this way, it can allocate their computing resource on business requirements for enterprise applications. Micro-service architecture is used to split large applications into a series of small modules that can be developed, tested, deployed, operated and upgraded independently. Micro-service architecture provides a more efficient way for a large number of Internet companies to expand their applications in the cloud, reduce complexity of application development and gain agility. In this paper we analyze and test the micro-service architecture model in a specific case-we develop and deploy the enterprise application system in the cloud to implement performance tests of two architectures (single-mode architecture and micro-service architecture mode), and we can acquire the evaluation result. These results have some guiding significance for solving the problems that may be encountered in the micro-service application of enterprise application.

Key words: cloud platform; micro-service; continuous delivery; scalable applications

云计算[1]提供了一种面向企业应用的可实现按需分配计算资源的模型, 企业可以将应用部署在 IaaS[2]、PaaS[3]或者 SaaS[4]服务平台上. 对于 IaaS 和 PaaS 服务平台的应用部署来说, 很多企业仅仅实现了原有应用单体化架构的迁移部署. 为了充分发挥云计算平台的性能优势, 他们将面对很多挑战, 比如: 弹性扩展、持续交付、热部署、动态监控和高可用性等.

企业选择将系统迁移到 IaaS、PaaS 服务平台的主要原因是为了获取更高的系统执行效率, 提高系统性能, 以及实现按需对应用进行动态扩展. 对于采用三层模型(表现层、业务层、持久层)的 Web 应用来说, 可使用不同技术栈实现整个 Web 应用, 如 J2EE、.NET、PHP 等, 由于所用的技术栈不同, 应用系统不能高效的按需增加或删除服务模块[5], 在实现云环境的部署迁移过程中可能会遇到很多问题[6].

对大部分企业来说, 创新至关重要, 当系统实现了新的创新点与功能点时, 需要完成从系统开发到应用云端部署的快速迭代更新, 以此提升系统应用的用户体验. 这也是大型互联网公司和 SaaS 提供者一直强调持续交付的重要原因[7].

① 收稿时间:2016-08-09;收到修改稿时间:2016-09-23 [doi:10.15888/j.cnki.csa.005746]

在单体化应用中, 所有的服务都由一个统一的代码库实现, 由开发者团队共同维护. 当某个开发者想要修改指定服务时, 必须保证其它所有服务不受影响. 随着应用越来越大, 服务越来越多, 扩展、修改某个服务将变得异常复杂. 除此之外, 单体化应用的更新部署需要重启所有服务(包括系统中未更新的服务), 而且, 单体化应用具有单节点失效性, 即只要一个服务出现问题, 整个应用系统全部宕机.

很多大型互联网公司已经意识到单体化架构的局限性, 他们开始采用一种新的应用系统架构应对出现的问题: 微服务架构[8].

微服务概念由 Martin Fowler 与 James Lewis 于 2014 年提出, 短短几年时间内已有越来越多的公司开始尝试使用微服务架构构建围绕业务、细粒度的分布式系统. 例如: 唯品会通过使用自研的服务化框架, 核心业务已经全面实现微服务化, 同时构建了基于大数据体系的新一代全链路监控系统来支撑微服务化的监控; 今日头条通过将系统从 Mono 架构迁移到微服务架构来应对架构和基础设施在性能及其优化、稳定性、可运维性、可扩展性、开发迭代速度等方面的挑战; 电商 CRM 通过微服务架构的应用重构了原来臃肿低效的 CRM 系统, 让每个服务小团队专注自己的业务快速迭代.

微服务架构可以帮助企业将新的功能点快速迭代到生产环境中, 减少开发、部署的复杂性、降低资源消耗. 对于功能点众多、业务逻辑复杂的大型应用系统来说, 采用微服务架构可以有效提高系统开发效率及容错性, 便于实现系统功能点及集群的扩展. 此外, 应用系统的微服务化可现实服务模块的单独部署, 让持续部署成为可能. 微服务架构模式正在为敏捷部署以及复杂企业应用实施提供着巨大的帮助.

论文的内容组织: 文章的第一部分主要介绍了当前的研究背景及微服务架构的优势. 第二部分通过一个典型应用对比了单体化及微服务架构的特点. 第三部分实现了典型应用在亚马逊的 Web 服务基础设施中的部署, 包括单体化和微服务化两种架构. 第四部分对已部署的单体化及微服务化应用进行测试, 并分析测试结果. 最后一部分对论文工作进行了总结并对进一步工作做了简单介绍.

1 相关研究

一个企业级的单体化应用系统通常由统一的代码库维护所有服务接口, 所有开发人员共享这个代码库. 单体化应用代码的紧耦合特性会增加系统扩展工作的复杂度, 当应用系统需要对某些服务或者功能进行修改和扩展时, 整个过程将变得十分复杂.

为了避免单体化应用的各种问题, 微服务应运而生. 这是一种以 SOA(面向服务架构)为基础的轻量级架构模式, 现在已经被 Amazon[9]、NetFlix[10]、Gilt[11]、LinkedIn[12]、SoundCloud[13]等公司应用到系统中. 微服务架构强调敏捷开发[14]、业务逻辑和技术的简洁性, 允许开发团队在云环境中快速扩展、部署应用, 这是一种面向云计算的持续解决方案.

微服务架构[15]就是把原来的一个提供 n 种服务的应用模块划分为若干个独立的服务模块, 原来应用的所有服务功能都由这些服务模块分别实现. 每一个微服务都是一个相对独立的个体, 由专门的开发团队独立开发、测试、部署, 至于采用哪种技术栈, 完全取决于本微服务模块, 不必考虑其他服务以及后期集成、部署等问题. 在表现层, 所有的微服务都发布成 REST[16]风格的接口.

尽管微服务架构在一些企业中得到了很好的应用, 满足云环境中高效的动态扩展企业应用系统的需求, 减少复杂性、使开发团队管理更加容易, 但对一个没有微服务经验的企业来说, 采用微服务架构具有一定挑战性. 本文用单体化及微服务架构分别实现一个小型应用系统, 比较两种架构的表现.

2 单体化及微服务化架构特点

以一个包含两种业务逻辑的应用系统为例, 对两种架构及系统部署的特点进行说明比较.

应用系统业务流程为: 查询和产生某个贷款用户的还款计划. 为了便于研究, 将其抽象为两个主要服务. 服务 S1: CPU 密集型操作, 负责产生还款计划; 服务 S2: 接收还款计划对应的 ID, 通过数据库查询操作返回这个还款计划的所有相关信息.

2.1 单体化架构

对于单体化架构来说, 整个系统的实现包含在一个应用中, 应用在水平方向采用分层架构, 从上至下依次为表现层、业务层、持久层等. 虽然水平分层降低了业务复杂性, 但由于垂直方向缺乏清晰的边界, 针对不同业务的处理逻辑相互耦合, 因此对于广度上具有复杂业务的系统, 单体化架构会带来一系列问题.

对于本贷款业务系统来说, 服务 S1 和服务 S2 被实现于一个 Web 应用中, 系统实现需要一个统一的代码库, 其架构如图 1 所示.

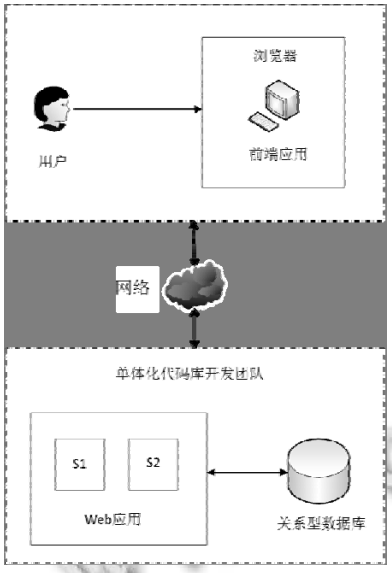

图 1 单体化架构

单体化架构实现此贷款业务系统具有如下特点:

1) 所有服务实现于一个应用中, 易于部署;

2) 代码依赖关系复杂, 不易于管理维护;

3) 局部修改影响不可知, 例如, 开发人员修改服务 S1 可能造成系统整体出现问题;

4) 不易于实现功能调整及增加, 例如, 开发人员增加功能点后, 可能会对 S1 及 S2 服务造成影响, 需要对系统进行全覆盖测试并重新部署;

5) 当需要增加硬件资源以应对更大的业务量时, 无法做到针对指定服务的硬件调整, 由于不同模块对资源需求差异大, 整体的硬件增加会造成资源浪费.

单化架构实现的应用系统提供两个 REST 风格的接口(分别对应 S1 和 S2)供 Web 前端调用. Web 前端应用是一个轻量级应用, 负责接收终端用户的请求(来自浏览器), 得到服务端单体化应用对请求的处理结果并将其返回终端. 前端应用和终端(浏览器)之间通过自定义的消息协议交换消息(JSON 格式).

此外, 为了提高系统响应效率, 在 Web 前端和服务端间增加负载均衡服务器, 将单体化应用部署在多台 Web 服务器中. 对单体化架构来说, 所有部署了应用系统的节点均包含对数据库的查询操作实现, 关系型数据库存储了用户的贷款信息.

部署到云环境中的系统架构图如图 2 所示.

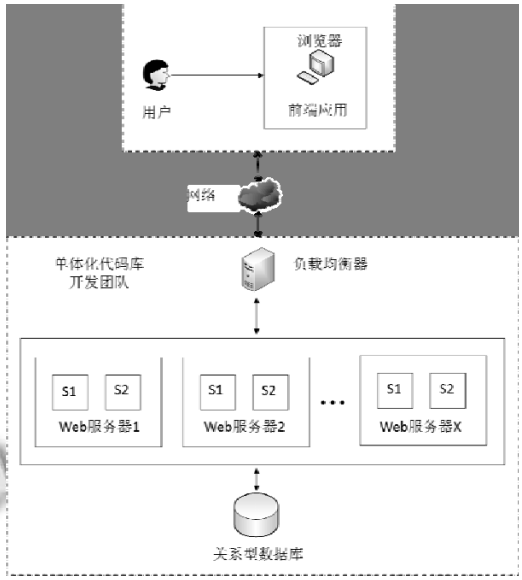

图 2 单体化系统部署架构(示意图)

2.2 微服务架构

对于微服务架构来说, 根据不同的业务实现, 系统被拆分为多个独立的服务模块, 合理划分系统的功能模块很重要, 由于本贷款系统业务逻辑简单, 即可将其划分为两个微服务 uS1 和 uS2, 分别对应单体化架构中的 S1 和 S2. 微服务间通过有限的 API 接口相互关联, 通讯协议一般使用 HTTP, 数据格式为 JSON, 系统架构如图 3 所示.

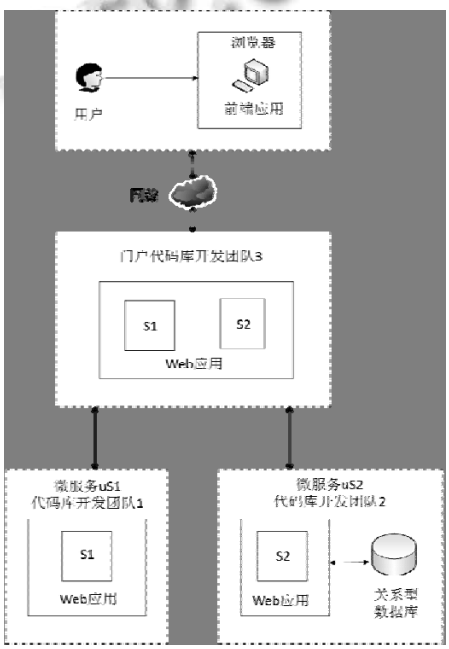

图 3 微服务架构

当客户端向应用系统发起请求时, 由于微服务架构包含多个服务模块, 客户端完成一次业务逻辑往往需要发送多个 HTTP 请求, 这会造成应用系统的吞吐量下降, 在高延迟的网络环境下更会影响用户体验. 因此, 不同于单体化应用, 微服务需要通过加入门户应用程序(Gateways application)提高系统性能.

门户应用负责为不同类型的终端用户提供相应的服务接口, 它可以将客户端请求转换为一组内部服务的调用. 此外, 门户应用可减小客户端与微服务的关联性, 服务器升级将不再影响客户端.

门户应用程序和微服务一样, 由一个单独的开发团队负责开发、测试、部署、维护、扩展和升级. 微服务通过门户应用程序向外界提供服务.

微服务架构实现此贷款业务系统具有如下特点:

1) 每个微服务足够内聚(只包含 S1 或 S2), 易于代码的开发与理解;

2) 不同服务部署相对独立, 例如, 对 uS2 微服务进行更新只需重部署此模块, uS1 在此期间保持正常对外提供服务;

3) 易于实现功能和集群扩展, 可将不同服务部署于合适的节点, 如将 uS1 部署于高 CPU 性能节点, uS2 部署于大内存节点;

4) 提高系统容错性, 例如, uS2 发生内存泄漏不会致使整个系统瘫痪;

5) 能够随着业务需求的变化灵活调整、增加系统功能, 不受所用技术的限制.

微服务在云环境中的部署架构如图 4 所示. 为了提高系统的可扩展性, 每一个微服务均可独立扩展. 微服务 uS1 使用一个负载均衡服务器及若干 Web 服务器完成部署. 微服务 uS2 除了上述服务器外还需要一个关系型数据库存储相关数据.

不同于单体架构, 微服务化的贷款业务系统只有 uS2 所部署的节点才会与数据库交互, 其他节点不包含数据库访问客户端.

此外, 为了提高系统执行效率, 门户应用同样采用负载均衡机制.

3　单体及微服务化应用的云化部署

以第二部分提到的贷款业务系统为例实现两种架构的云化部署. 系统中包含的两个服务的业务需求如表 1 所示.

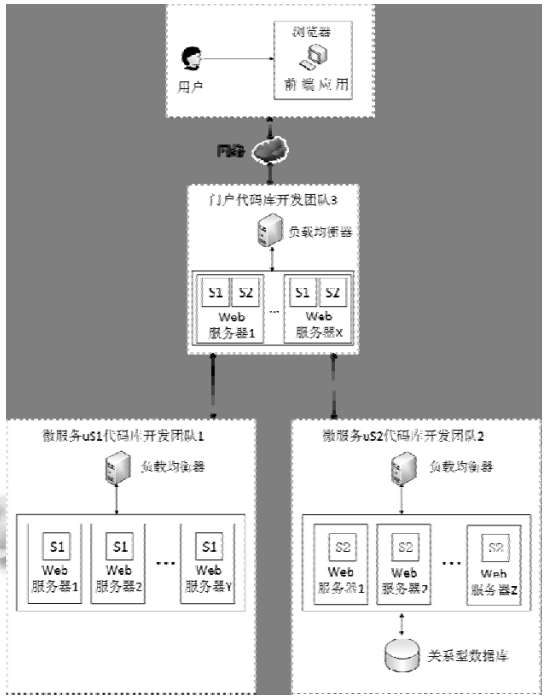

图 4　微服务部署架构(示意图)

表 1　两个服务的业务需求

服务类型	每分钟请求数	平均响应时间(毫秒)	最大响应时间(毫秒)
S1-产生还款计划	30	3000	6000
S2-获取指定还款计划	1100	300	1500

两种架构均采用 Play Web 框架[17]实现. Play 框架是一种轻量级的、无状态的、异步的云友好框架. 采用 Play 框架的应用程序可以部署在 Netty 服务器上, 这是一种启动速度快并且节约计算资源的服务器.

系统包含关系型数据库, 数据操作通过 EBeans 实现. EBeans ORM 是一种快速、简单的数据访问结构, 也是 Play 默认的对象关系映射模型, 在浏览器端执行的应用程序用 Angular.js 实现, 因为它能很好的支持 Google.

单体化架构系统由两个相对独立的应用模块组成, 使用技术栈及框架如下:

① Web 应用: 该模块的开发采用 Play2.2.2, Scala2.10.2 和 Java1.7.0, 数据库用 PostgreSQL9.3.6.

② 前端应用: 该模块的开发采用 Angular.js 1.3.14(HTML5, CSS 和 JQuery).

微服务架构系统由四个相对独立的应用模块组成, 使用技术栈及框架如下:

① 微服务 uS1: 该模块的开发采用 Play2.2.2,

Scala2.10.2 和 Java1.7.0.

② 微服务 uS2: 该模块的开发采用 Play2.2.2, Scala2.10.2 和 Java1.7.0, 数据库用 PostgreSQL9.3.6.

③ 门户应用(Gateway application): 该模块的开发采用 Play2.2.2, Scala2.10.2 和 Java1.7.0.

④ 前端应用: 该模块的开发采用 Angular.js 1.3.14(HTML5, CSS 和 JQuery).

为了比较部署在云环境中的不同架构系统的表现, 将两种架构系统均部署在 AWS(Amazon Web Service)上, 并通过对系统的性能测试选出满足表 1 所示业务需求的 AWS 实例. 例如, 对单体化 Play 架构系统的性能测试以 C4 簇(c4.large、c4.xlarge、c4.2xlarge)实例为测试用例, 其中 c4.2xlarge 能很好地满足表 1 中的业务需求, 即采用 c4.2xlarge 实例.

3.1 单体化架构云端部署

单体化架构云端部署采用图 1 所示结构, 其中的 Web 应用及前端应用所用 AWS 实例如下所示:

① Web 应用: Web 应用部署在 EC2 的 c4.2xlarge 型实例上, 配置参数: 8 vCPUs,, 31 ECUs 和 15GB RAM, 服务器: Netty3.7.0, 数据库用 AWS 提供的 db.m3.medium 型实例(1 vCPU 和 3.75GB RAM).

② 前端应用: Angular.js 的一些静态文件存储在 Play 的 Web 服务器上, 通过本地缓存方式响应请求.

3.2 微服务架构云端部署

微服务架构云端部署采用图 3 所示结构, 其中的微服务 uS1、uS2、门户应用及前端应用所用 AWS 实例如下所示:

① 微服务 uS1: Web 应用部署在 EC2 的 c4.xlarge 型实例上, 配置参数: 4 vCPUs, 16 ECUs 和 7.5GB RAM, 服务器: Netty3.7.0.

② 微服务 uS2: Web 应用部署在 EC2 的 m3.medium 型实例上, 配置参数: 1 vCPUs, 3 ECUs 和 7.5GB RAM, 服务器: Netty3.7.0. 数据库用 AWS 提供的 db.m3.medium 型实例(1 vCPU 和 3.75GB RAM).

③ 门户应用: Web 应用部署在 EC2 的 m3.medium 型实例上, 配置参数: 1 vCPUs, 3 ECUs 和 7.5GB RAM, 服务器: Netty3.7.0.

④ 前端应用: 与单体化应用架构前端部署情况类似.

两种架构的系统运行开销如表 2 和表 3 所示, 与开销相关的带宽、存储等因素在两种架构中保持一致.

表 2 单体化架构部署的基础设施开销

服务名称	每小时开销(美元)	每月运行小时数	每月开销(美元)
Web 应用. 1 个 EC2 云服务实例 c4.2xlarge.	$0.464	720	$334.08
Web 应用. 1 个关系型数据库实例 db.m3.medium.	$0.090	720	$64.80
每月系统总开销(美元)			$398.99

表 3 微服务架构部署的基础设施开销

服务名称	每小时开销(美元)	每月运行小时数	每月开销(美元)
微服务 μS1. 1 个 EC2 云服务实例 c4.xlarge.	$0.232	720	$167.04
微服务 μS2. 1 个关系型数据库实例 db.m3.medium	$0.090	720	$64.80
微服务 μS2. 1 个 EC2 云服务实例 m3.medium.	$0.067	720	$48.24
门户应用. 1 个 EC2 云服务实例 m3.medium.	$0.067	720	$48.24
每月系统总开销(美元)			$328.32

4 实验结果分析

对比单体化架构和微服务架构测试数据, 从应用系统的性能、部署、开发等方面分析实验结果, 得出详细结论.

4.1 性能

在满足表 1 中业务需求的前提下分析两种架构性能表现, 我们采用 AWS 上相同的实例 JMter[18]2.13 进行比较. 在测试中, 通过配置 JMter 模拟一个稳定的工作负载, 对服务 1 每分钟执行 30 个请求, 服务 2 每分钟执行 1100 个请求.

两种框架每个服务的平均响应时间和 90%响应时间线如图 5 和图 6 所示. 从图中可以看出, 这两种框架都满足了表 1 提出的业务需求, 并验证了如下结论:

1) 微服务不会因为使用服务主机数量过多而导致响应时间的延迟;

2) 微服务允许更大粒度的实例类型减少系统开销. 在本例中微服务架构减少 17%的基础设施服务开销(根据表 2 和表 3).

4.2 开发方法

不同于单体化架构系统开发中使用的统一代码库, 微服务架构的每个开发团队都是相对独立的, 他们无需关心其他微服务开发团队的工作细节, 只负责自己的微服务模块, 每个开发团队都可以根据各自的技术

特点和业务需求使用不同的技术栈实现微服务.

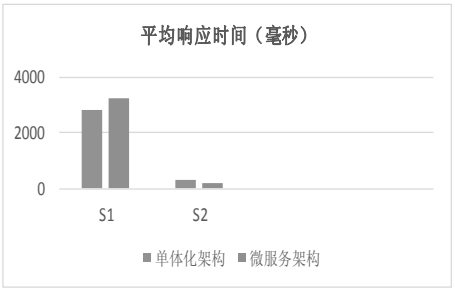

图 5 单体化架构和微服务架构的平均响应时间

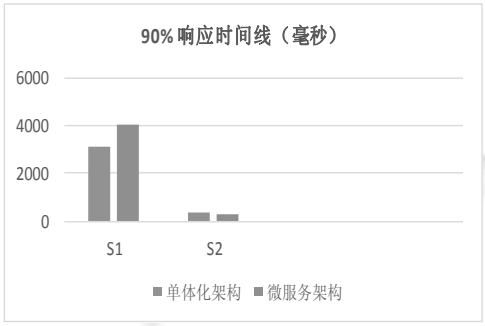

图 6 单体化架构和微服务架构 90%响应时间线

微服务架构系统开发过程中应遵循以下原则:

1) 为了避免使用技术过多不便管理, 通过制定相关技术准则规范所有团队可以使用的技术集;

2) 通过制定详细的规范文档、合理设计每个微服务模块及对外发布的 REST API 接口, 保证微服务提供的功能模块能很好的被调用;

3) 保证门户应用提供的服务便于前端应用(浏览器、安卓、IOS)调用.

4.3 部署、扩展和持续交付

对微服务架构应用系统而言, 新版本的发布容易破坏其外部原本依赖的一些服务, 通过引入服务版本控制, 可避免新服务加入或旧服务过期造成的系统依赖异常问题.

在微服务中, 持续交付耗时耗力, 因为持续交付要求每一次部署工作的重复性手动任务被正确执行通过, 当然, 通过一些自动化工具可以节省时间, 提高交付效率. 微服务的部署工作也涉及 Devops, 开发部署一体化, 提供云环境下的检测和运行机制. 对于部署在 AWS 上的服务, 我们可以通过 New Relic 很方便的检测其运行状态, 但是不得不面对一个重要问题, 即在部署的过程中无法很好的跟踪从终端用户到微服务的请求流程.

5 结论和进一步工作

通过本文的实验结果分析, 我们看到了微服务架构的一些优势和不足, 其最突出的优势是可以把一个复杂应用拆分为一系列功能明确的服务模块, 每个服务模块由专门的开发团队负责, 独立于其他服务模块, 可以单独开发、测试、部署和扩展升级. 同时, 不同于单体化架构的统一代码库模式, 微服务架构应用系统中的每一个微服务都有各自的代码库, 各自维护.

下一步我们将对微服务架构的各方面性能做进一步评估, 比如更大粒度的可扩展性和更低的资源开销, 如何划分微服务也是我们需要重点考虑的一个问题. 微服务和门户应用的自动化部署采用不同的策略、工具和云服务管理器进行评估. 后续工作还包括评估微服务的一些其他特性, 如容错能力、分布式事务、异构数据分布、服务版本控制及微服务的可靠性保障等.

参考文献

1 Buyya R. Cloud computing: The next revolution in information technology. The 1st International Conference on Parallel Distributed and Grid Computing (PDGC). Solan. 2010. 2–3.

2 Vosshall P. Web scale computing: The power of infrastructure as a service. Athman Bouguettaya, Ingolf Krueger, and Tiziana Margaria, Service-Oriented Computing–ICSOC 2008. Berlin. Springer. 2008, 1.

3 Beimborn D, Miletzki T, Wenzel S. Platform as a Service (PaaS). Business & Information Systems Engineering, 2011, 3(6): 381–384.

4 Schütz SW, Kude T, Popp KM. The impact of software-as-a-service on software ecosystems. Georg Herzwurm and Tiziana Margaria, Eds. Software Business. From Physical Products to Software Services and Solutions. Berlin. Springer. 2013. 130–140.

5 Lorido-Botran T, Miguel-Alonso J, Lozano JA. A review of auto-scaling techniques for elastic applications in cloud environments. Journal of Grid Computing, 2014, 12(4): 559–592.

6 Cretella G, Di MB. An overview of approaches for the migration of applications to the cloud. Caporarello L, Di

Martino B, Martinez M, eds. Smart Organizations and Smart Artifacts. Berlin. Springer. 2014. 67–75.

7 Rossberg J, Olausson M. Continuous Delivery. Proc Application Lifecycle Management with Visual Studio 2012. Berkeley Apress. 2012. 425–432.

8 Lewis J, Fowler M. Microservices. http://martinfowler.com/articles/microservices.html. [2014-03].

9 Kramer S. GIGAOM. The biggest thing Amazon got right: The Platform. https://gigaom.com/2011/10/12/ 419-the-biggest-thing-amazon-got-right-the-platform/. [2011-10-12].

10 Mauro T. Nginx. Adopting microservices at Netflix: Lessons for architectural design. http://nginx.com/blog/microservices-at-netflix-architectural-best-practices/. [2015-02].

11 Goldberg Y. InfoQ. Scaling Gilt: From monolithic ruby application to distributed scala micro-services architecture. http://www.infoq.com/ presentations/ scale-gilt. [2014-10].

12 Ihde S. InfoQ. From a monolith to microservices+REST: The evolution of linkedIn's service architecture. http://www.infoq.com/presentations/linkedin-microservices-urn. [2015-03].

13 Calçado P. SoundCloud. Building products at SoundCloud-Part I: Dealing with the Monolith. https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith. [2014-06].

14 Lawton G. TechTarget. How microservices bring agility to SOA. http://searchcloudapplications.techtarget.com/feature/How-microservices-bring-agility-to-SOA. [2015-01].

15 Richardson C. Microservices. Pattern: Microservices architecture. http://microservices.io/patterns/microservices.html. [2014-03].

16 Vinoski S. REST eye for the SOA guy. IEEE Internet Computing, 2007, 11(1): 82–84.

17 Hunt J. Play Framework. A Beginner's Guide to Scala, Object Orientation and Functional Programming. Berlin: Springer, 2014: 413–428.

18 Rahmel D. Testing a Site with ApacheBench, JMeter, and Selenium. Advanced Joomla! Berkeley: Apress, 2013: 211–247.