架构师

ARCHITECT

特刊

Serverless的入门与思考

SPECIAL ISSUE

July,2017

架构师特刊

Geekbang 极客邦科技 | InfoQ

序言

InfoQ 运维技术编辑 木环

Serverless 一词可以追溯到 2012 年，Ken Fromm 撰文讨论了 Why TheFuture Of Software And Apps Is Serverless? 彼时云计算世界依然围绕着“服务器”这个概念，云用户依然需要考虑服务器的数量价格以及何时扩容等问题；Ken 认为 Serverless 并不是没有服务器了，而是说云应用开发者们再也不需要考虑服务器了，云用户可以使用云端资源并且无需操心物理容量或者上限等管理事宜。

大部分的 Serverless 供应商以 FaaS 平台方式提供服务，即 Function as a Service，可直接使用的计算资源按照既定的业务应用逻辑进行运算。其实在 2006 年，Zimki 发布了一款‘pay as you go’理念的代码平台，遗憾的是当时并取得商业化的成功。2014 年 Amazon 发布 AWS Lambda，成为首个提供 serverless 服务的公有云厂商，最初只支持 Node.js，现在支持 Python，Java 和 C#。2016 年，ServerlessConf 在伦敦召开。目前，还有 Google 的 Google Cloud Functions，Azure 的 Azure Functions，IBM 的 OpenWhisk，同时国内公有云厂商阿里云、腾讯云也于今年推出 serverless 服务。

云计算厂商们如此重视的 Serverless 是怎样的？它适用于怎样的业务场景？会多大程度给开发用户们带来便利？又会带来怎样的影响和冲击？我们应该如何看待这种正在被炒作的技术？ InfoQ 精心筛选了若干篇文章整理成册，分享给读者朋友们。

目录

无服务器架构

无服务器架构（一）

作者 Mike Roberts 译者 大愚若智

无服务器架构是指大量依赖第三方服务（也叫做后端即服务，即“BaaS”）或暂存容器中运行的自定义代码（函数即服务，即“FaaS”）的应用程序，AWS Lambda 是目前最著名的供应商。通过采用这样的方式并将大部分行为移至前端，这种架构使得应用程序内部不再需要具备传统的“始终运行”的服务器系统。取决于具体情况，尽管对供应商依赖增加了，并且（目前）配套服务还不成熟，但此类系统依然可以大幅降低运维成本和复杂度。

无服务器是软件架构世界中的热门话题。在这一领域我们已经看到很多书籍、开源框架、大量不同类型的供应商和产品，甚至专门的会议活动。但无服务器到底是什么，为什么值得（或不值得）考虑该技术？通过出版

物的进化这篇文章，希望能针对这些问题向你提供一些启发。

首先我们要介绍无服务器到底是“什么”，在介绍这种技术的优势和劣势时我会尽量保持中立 – 下文将分别探讨这些问题。

无服务器是什么?

与软件行业很多其他趋势类似，关于“无服务器”到底是什么并没有一个清晰直观的看法，而涉及到两个截然不同但又相互重叠的领域后，问题就更突出了：

“无服务器”这个称呼最早被用于描述大量或完全依赖（“位于云中的”）第三方应用程序 / 服务管理服务器端逻辑和状态的应用程序。这些程序通常是使用可通过云访问的数据库（例如 Parse、Firebase）、身份验证服务（例如 Auth0、AWS Cognito）等庞大生态系统的“富客户端”应用程序（例如只包含一个页面的 Web 应用或移动应用）。这类服务以往也被叫做“（移动）后端即服务”，下文会将其简称为“BaaS”。

“无服务器”也可以指部分服务器端逻辑依然由应用程序开发者来编写的应用程序，但与传统架构的不同之处在于，这些逻辑运行在完全由第三方管理，由事件触发的无状态（Stateless）暂存（可能只存在于一次调用的过程中）计算容器内。（感谢 ThoughtWorks 在他们最近的 Tech Radar 中对这种情况的定义。）这种做法可以看作“函数即服务 /FaaS”。AWS Lambda 是目前最流行的 FaaS 实现，但除此之外也有其他实现。下文将使用“FaaS”作为无服务器这种运用方式的简称。

“无服务器”的起源

“无服务器”这个词并不好理解，因为有很多应用程序的服务器硬件与进程运行在不同位置，但与传统方式的差异之处在于，构建和支持无服务器”应用程序的组织并不需要考虑服务器或进程，这些东西可以外包给供应商。

这个词首次出现大概是在 2012 年，包含在 Ken Fromm 撰写的这篇文章中。Badri Janakiraman 说他也在这段时间前后在持续集成和托管服务式

的源代码控制系统领域听到过这个词，但并不是用来代表某个公司自己的服务器。这种用法侧重的是基础结构的开发，并未将其纳入具体产品。

当亚马逊于 2014 年发布 AWS Lambda 后，2015 年这个词开始变得广为人知，亚马逊在 2015 年 7 月发布 API 网关后这个词也变得愈加热门。这里有一个例子，API 网关公布后，Ant Stanley 写了一篇有关无服务器的帖子。2015 年 10 月，亚马逊的 re:Invent 大会中有一场名为“使用 AWS Lambda 搭建的无服务器公司”的讲话，其中介绍了 PlayOn! Sports 这家公司。2015 年底，“Javascript Amazon Web Services（JAWS）”开源项目更名为无服务器框架（Serverless Framework），使得这一势头愈加热烈。

快进到今天（2016 年中期），我们可以看到更多范例，例如最近举办的无服务器大会以及各种无服务器供应商，从产品描述到工作岗位介绍，都在广泛使用这个词。无论怎样，“无服务器”这个词成了“网红”。

本文将主要介绍上文提到的第二个领域，因为这个领域是最新出现的，与我们以往对技术架构的看法有更大不同，并且围绕无服务器这个词在这个领域有更多宣传炒作。

然而这些概念其实是相互关联，实际上甚至会逐渐会聚的。例如 Auth0 就是个很好的例子 – 这个技术最开始属于 BaaS 形式的“身份验证即服务”，但随着 Auth0 Webtask 的发布开始步入 FaaS 领域。

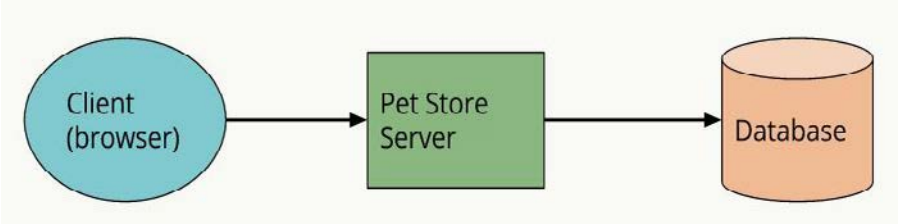
在很多情况下，当开发一款“BaaS 形式”的应用程序时，尤其是在开发“富”Web 应用而非移动应用时，可能依然需要一定数量的自定义服务器端功能。此时 FaaS 函数是一种很好的解决方案，尤其是需要在某种程度上与正在使用的某些 BaaS 服务进行集成时。这类功能的例子包括数据验证（防范仿冒客户端）以及计算密集型处理（例如图片或视频操作）等。

举几个例子

UI 驱动的应用程序

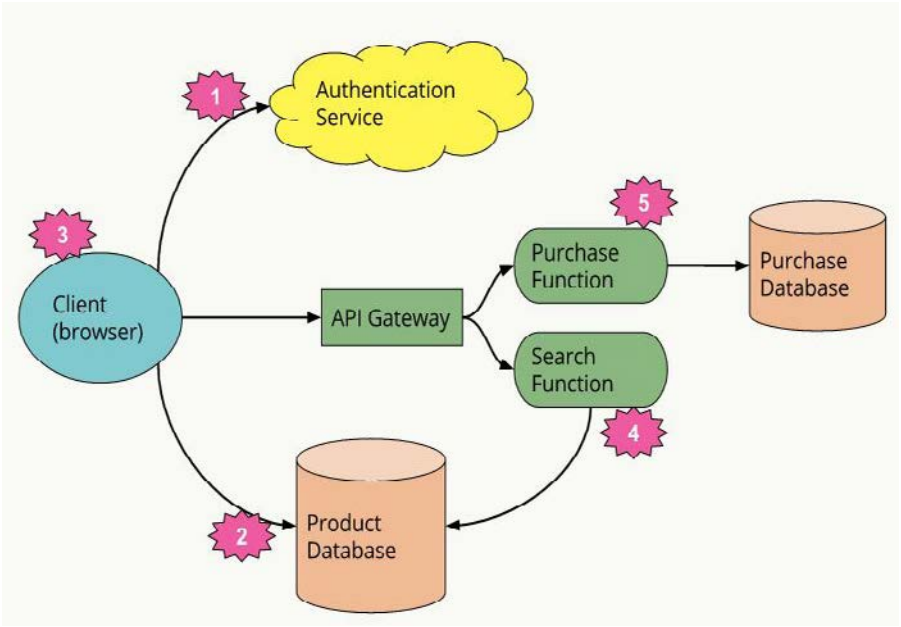
想象一个包含服务器端逻辑，面向客户端的传统三层系统。例如典型的电商应用（可以说在线宠物店吗？）

传统架构看起来类似下图，假设服务器端使用Java实现，并使用HTML/Javascript组件作为客户端：



这种架构下的客户端会显得相当笨重，系统中身份验证、页面导航、搜索、事务等大部分逻辑都是由服务器应用程序实现的。

如果使用无服务器架构，看起来将会是这样：



这是个大幅简化的视图，尽管如此其中也包含大量显著改动。请注意这并非为了向你推荐迁移时所要使用的架构，此处使用的这种架构仅仅为了介绍无服务器相关概念！

我们删掉了原始应用程序中的身份验证逻辑，并使用第三方BaaS服务代替。

作为BaaS的另一个例子，我们让客户端直接访问数据库子集（用于列出产品），而该数据库完全由第三方承载（例如使用AWS Dynamo）。

我们还为访问数据库的客户端提供了不同的安全配置文件，这样即可从任何服务器资源访问数据库。

上两点引出了非常重要的第三点：一些原本位于宠物店服务器上的逻辑，例如追踪用户会话，理解应用程序的 UX 结构（例如页面导航），从数据库读取并转换为可用视图等，现在被放入到客户端中。实际上客户端现在已经变成了一个单页应用程序（Single Page Application）。

我们希望将一些与 UX 有关的功能继续保留在服务器上，例如计算密集型功能或需要访问大量数据的功能。“搜索”就是个很好的例子。搜索功能无需使用持续运行的服务器，而是可以用一个能通过 API 网关（下文将详细介绍）响应 Http 请求的 FaaS 函数来实现。可以通过这样的客户端或服务器函数从同一个数据库中读取产品数据。

由于原本的服务器是通过 Java 实现的，而 AWS Lambda（本例中使用的 FaaS 供应商）支持通过 Java 实现的函数，无须彻底重写即可将搜索功能的代码从宠物店服务器直接移至宠物店搜索函数。

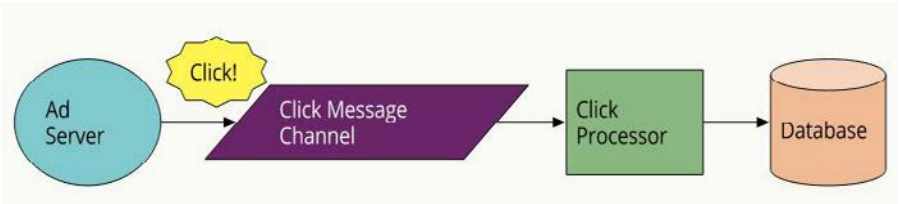
最后还可以使用另一个 FaaS 函数取代原本的“购买”功能，但出于安全考虑依然将其保留在服务器端，而非在客户端重新实现。这也是一种 API 网关的前端。

消息驱动的应用程序

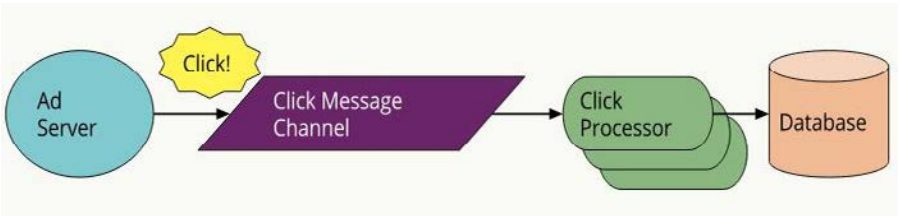
另一个例子是后端数据处理服务。假设编写的某个以用户为中心的应用程序需要快速响应 UI 请求，但你希望随后能记录发生的所有类型的操作。例如在线广告系统，用户点击广告后你希望非常快速地将用户重定向至目标广告，但为了向广告主收费，与此同时你还希望记录已发生点击这件事。（这并不是虚构的例子，我以前在 Intent Media 所属的团队就因为这个原因重新设计了自己的系统。）

传统方式下这种系统的架构可能是类似这样的：“广告服务器”会以同步的方式响应用户（由于只是例子，我们并不需要关心具体的交互），同时需要向渠道发布一条消息并由负责更新数据库的“点击处理器”应用程序

以异步的方式处理，例如扣掉广告主的部分预算。



在无服务器的世界中，这个系统应该是这样的：



和上一个例子相比，本例中两种架构的差异很小。我们将需要长期运行的消费者应用程序替换为一个在供应商提供的事件驱动的上下文中运行的 FaaS* 函数 *。请注意该供应商同时提供了消息代理（Message Broker）和 FaaS 环境，这两个系统非常紧密地相互结合在一起。

通过实例化（Instantiating）函数代码的多个副本，FaaS 环境可以并行处理多个点击，这主要取决于最初流程的编写方式，同时这也是一个需要考虑的新概念。

理解“函数即服务”

上文已多次提到 FaaS 这种想法，但还需要深入考虑一下这到底是什么意思。为此我们一起看看亚马逊 Lambda 产品的公开描述。我将这些描述划分成几点，下文将分别展开介绍。

AWS Lambda 可供你在无须供应或管理服务器的情况下运行自己的代码。[(1)] ... 借助 Lambda，你可以为几乎任何类型的应用程序或后端服务运行所需代码 [(2)] - 所有环境完全无须管理，只需要上传自己的代码，Lambda 将自动完成运行代码所需的一切条件 [(3)] 并可进行伸缩 [(4)] 你的代码将实现高可用。你可以将代码设置为通过其他 AWS 服务自动触发运行 [(5)] 或直接从任何 Web 或移动应用内部调用 [(6)]。

1. 从本质上来说，FaaS意在无须自行管理服务器系统或自己的服务器应用程序，即可直接运行后端代码。其中所指的 -服务器应用程序 - 是该技术与容器和PaaS（平台即服务）等其他现代化架构最大的差异。
2. 回想上文列举的点击处理系统那个例子，可以用FaaS取代点击处理服务器（可能是物理计算机，但绝对需要运行某种应用程序），这样不仅不需要自行供应服务器，也不需要全时运行应用程序。
3. FaaS产品不要求必须使用特定框架或库进行开发。在语言和环境方面，FaaS函数就是常规的应用程序。例如AWS Lambda的函数可以通过‘一等公民’Javascript、Python以及任何JVM语言（Java、Clojure、Scala）等实现。然而Lambda函数也可以执行任何捆绑有所需部署构件的进程，因此可以使用任何语言，只要能编译为Unix进程即可（参阅下文的Apex）。FaaS函数在架构方面确实存在一定的局限，尤其是在状态和执行时间方面，这些会在下文详细介绍。
4. 再次回想上文提到的点击处理范例，在迁往FaaS的过程中，唯一需要修改的代码是“主方法/启动”代码，其中可能需要删除顶级消息处理程序的相关代码（“消息监听器接口”的实现），但这可能只需要更改方法签名即可。在FaaS的世界中，代码的其余所有部分（例如向数据库执行写入的代码）无须任何变化。
5. 由于无须运行任何服务器应用程序，相比传统系统，部署方法会有较大变化 – 将代码上传至FaaS供应商，其他事情均可由供应商完成。目前这种方式通常意味着需要上传代码的全新定义（例如上传zip或JAR文件），随后调用一个专有API发起更新过程。
6. 横向伸缩过程是完全自动化且有弹性的，由供应商负责管理。如果系统需要并行处理100个请求，你自己无须任何额外配置，供应商可全部搞定。负责执行你的函数的‘计算容器’只会短暂存在，

FaaS供应商将完全根据运行时需求自动供应和撤销。

继续回到点击处理器那个例子。假设某天运气好，客户的广告点击量是平时的 10 倍。点击处理应用程序能否顺利处理？例如我们的代码能否同时处理多条消息？就算能处理，只使用一个应用程序实例是否足以应对激增的负载？如果可以运行多个进程，是否要自动伸缩，或者需要手工修改配置？如果使用 FaaS，需要在编写函数的过程中就考虑到这些问题，但随后将由 FaaS 供应商自动完成伸缩相关的任务。

FaaS 中的函数可以通过供应商定义的事件类型触发。对于亚马逊 AWS，此类触发事件可以包括 S3（文件）更新、时间（计划任务），以及加入消息总线的消息（例如 Kinesis）。通常你的函数需要通过参数指定自己需要绑定到的事件源。对于点击处理程序，我们假设已经使用了可被 FaaS 支持的消息代理。如果不支持，则需要换为使用可支持的，并且也可能需要更改消息的创建方。

大部分供应商还允许函数作为对传入 Http 请求的响应来触发，通常这类请求来自某种该类型的 API 网关（例如 AWS API 网关、Webtask）。我们在宠物店例子中为“搜索”和“下单”函数使用了这种触发方式。

状态

在本地（机器 / 实例范围内）状态方面 FaaS 函数会遇到很多局限。简而言之你需要假设对于函数的任何调用，你所创建的任何进程内或主机状态均无法被任何后续调用所使用。包括内存（RAM）中的状态以及写入本地磁盘的状态。换句话说，从部署单位的角度来看，FaaS 函数是无状态的。

虽然影响因素还有很多，但这会对应用程序架构产生巨大影响。“十二要素应用（Twelve-Factor App）”这一概念也存在完全相同的局限。

考虑到这种局限，备选方案有哪些？通常这意味着 FaaS 函数可能是天然无状态的，例如只是单纯对输入内容提供功能的转换，或者可以使用数据库、跨应用程序缓存（例如 Redis）或网络文件存储（例如 S3）跨越

不同请求存储状态，以进一步将其作为其他请求的输入。

执行时间

FaaS 函数通常会对每次调用可以执行的时间长度有所限制。目前 AWS Lambda 函数只能运行不超过 5 分钟，超过这一时间将被终止。

这意味着如果不进行重构，某些类型的“长寿”任务不适合使用 FaaS 函数，例如可能需要创建多个相互协调的 FaaS 函数，而在传统环境中只需要用一个“长寿”的任务同时负责协调和执行。

启动延迟

目前来说，FaaS 函数需要等待多长时间才能响应请求，这取决于多种因素，具体时间可能介于 10 毫秒到 2 分钟之间。听起来很糟糕，但我们一起用 AWS Lambda 作为例子更深入地看看这个问题。

如果函数使用 Javascript 或 Python 实现并且不大（例如不到五百行代码），那么运行该函数的开销通常绝对不会超过 10-100 毫秒。更大的函数偶尔可能需要更长时间。

如果你的 Lambda 函数是通过 JVM 实现的，由于 JVM 需要逐渐启动，响应时间偶尔可能会显得较长（例如超过 10 秒）。然而只有在下列情况下才会存在这样的问题：

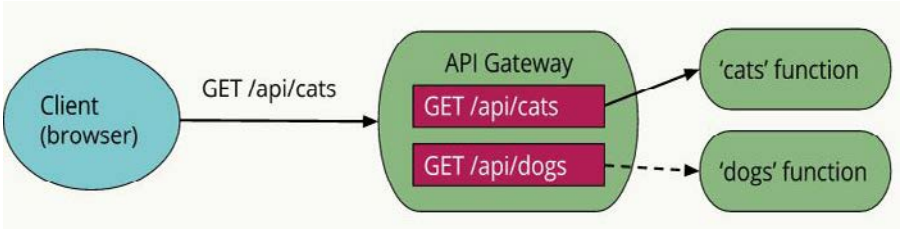
- 不同调用之间偶尔需要以超过10分钟的间隔处理函数处理事件。
- 流量突然激增，例如通常每秒只需处理10个请求，但突然在不到10秒内需要每秒处理100个请求。

通过用没什么技术含量的手段每 5 分钟 Ping 一下函数，使其保活，前一种问题在某些情况下可以避免。

是否需要担心这些问题，主要取决于应用程序的类型和流量模式。我以前加入的一个团队使用 Java 实现了异步消息处理 Lambda 应用，该应用每天需要处理数以百亿计的消息，他们就不需要担心启动延迟的问题。话虽如此，如果你要编写一个低延迟的交易应用程序，无论使用什么语言实现，现阶段可能都不会想使用 FaaS 系统。

无论是否觉得自己的应用会遇到这样的问题，你都需要使用类似生产环境的负载进行测试，看看能获得怎样的性能。如果你的用例目前还无法支持，也许可以过几个月再试试，因为目前 FaaS 供应商都在努力对这一领域进行完善。

API网关



上文还提到过 FaaS 一个比较重要的内容：“API 网关”。API 网关是一种 HTTP 服务器，通过在配置中定义路由 / 端点，可将每个路由与某一 FaaS 函数关联在一起。当 API 网关收到请求后，会查找与该请求匹配的路由配置，随后调用相应的 FaaS 函数。通常来说 API 网关会将 Http 请求的参数与 FaaS 函数的输入参数映射在一起。API 网关会将 FaaS 函数的调用结果转换为 Http 响应，并将其返回给原始调用方。

Amazon Web Services 有自己的 API 网关，其他供应商也提供了类似的能力。

除了最基本的请求路由，API 网关还可以执行身份验证、输入验证、响应代码映射等功能。你可能会从直觉上纳闷这样做是否真是个好办法，先把这个想法暂时放一放，下文还将进一步介绍。

API 网关 +FaaS 的用例之一是在无服务器系统中创建以 Http 为前端的微服务，并通过 FaaS 获得伸缩、管理，以及其他各类收益。

目前与 API 网关有关的工具还非常不成熟，因此使用 API 网关定义应用程序这种做法很可能需要你足够大胆。

工具

上文有关 API 网关工具还不成熟的观点实际上也适用于整个无服务器

FaaS 大环境。不过凡事总有例外，例如 Auth0 Webtask 对于工具开发的重视远远超过开发者 UX 的完善。Tomasz Janczuk 在最近举办的无服务器大会上也很好地阐述了自己对相关工具的重视。

总的来说，无服务器应用的调试和监控都很麻烦，我们也会在下文详细介绍这个问题。

开源

无服务器 FaaS 应用程序的主要收益之一在于可以透明地实现生产运行时供应，因此目前开源对这一方式的意义并不像对于 Docker 和容器等技术的意义那么大。未来我们可能会看到一些主流的 FaaS/API 网关平台可以支持在“本地”，或开发者的工作站上运行。IBM OpenWhisk 就是此类实现的一个例子，我们很期待看到类似这样的实现最终能获得更大范围的采用。

尽管除了运行时实现，已经有开源的工具和框架可以为定义、部署、运行提供帮助。例如无服务器框架使得 API 网关 +Lambda 的使用过程变得比 AWS 更简单，虽然这种方式大量依赖 Javascript，但如果你要编写 JS API 网关应用，绝对值得一试。

另一个例子是 Apex，这个项目可以“更轻松地构建、部署和管理 AWS Lambda 函数”。Apex 尤其有趣的一点在于，该技术可供你用亚马逊不直接支持的语言开发 Lambda 函数，例如 Go 语言。

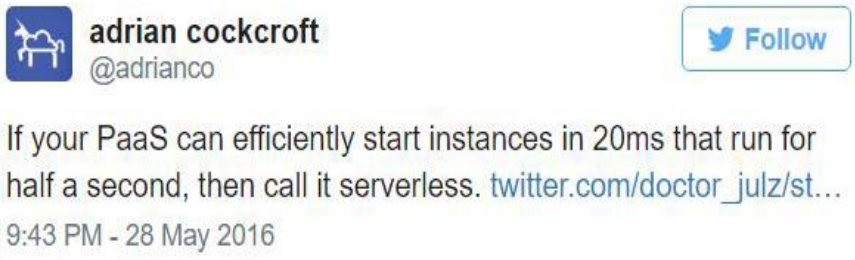
无服务器不是什么?

至此本文已经介绍了“无服务器”是好几个不同概念的结合，例如“后端即服务”和“函数即服务”，并且还对第二个概念进行了详细的介绍。

在开始介绍这一技术最重要的收益和不足之处前，我还想进一步谈谈这一概念的定义，至少要讨论一下无服务器不是什么。我曾发现很多人（包括我自己以前）对这些概念感到困惑，因此有必要进行澄清。

与PaaS相比

考虑到无服务器 FaaS 函数其实非常类似于 12 要素应用，那么无服务器是否像 Heroku 一样仅仅是另一种类型的“平台即服务”（PaaS）？引用 Adrian Cockcroft 的观点作为答案吧。



“如果你的 PaaS 可以非常高效地在 20 毫秒内启动实例，并将该实例运行 0.5 秒，那就将其称之为‘无服务器’吧。 —— @adrianco”

换句话说，大部分 PaaS 应用程序并不适合针对每个请求让整个应用程序上线并下线，而 FaaS 平台就是为实现这一做法而生的。

但那又怎样，如果我精通 12 要素应用的开发，具体如何写代码其实也没什么不同？没错，但应用的运维方式会有翻天覆地的变化。我们都是精通 DevOps 的工程师，对运维和开发的重视程度是相同的，对吧！

FaaS 和 PaaS 在运维方面的最大差异在于伸缩。大部分 PaaS 依然需要考虑伸缩，例如对于 Heroku，到底需要运行几个 Dyno？但 FaaS 应用程序完全无须考虑这些问题。就算配置 PaaS 应用程序进行自动伸缩，但也无法在特定请求层面上做到这一点（除非具备非常有针对性的流量塑形配置文件），因此在成本方面 FaaS 应用程序效率更高。

尽管能获得这样的收益，又为什么还要继续使用 PaaS？原因有很多，工具和 API 网关的成熟度可能是其中最重要的。另外 12 要素应用在 PaaS 中的实现为了进行优化可能需要使用应用内只读缓存，FaaS 函数还无法支持这种技术。

与容器相比

使用无服务器 FaaS 的另一个原因在于避免在操作系统层面或更底层的层面管理计算过程。平台即服务（例如 Heroku）也能实现这一点，但

上文已经介绍过 PaaS 与无服务器 FaaS 的差异。容器是另一种流行的过程抽象方式，例如 Docker 已成为此类技术最明显的例子。我们还发现诸如 Mesos 和 Kubernetes 等容器承载系统正在开始普及，这些技术可以将特定应用程序从操作系统级别的部署中抽象出来。另外还有云端承载的容器平台，例如 Amazon ECS 和 Google Container Engine，这些技术与无服务器 FaaS 类似，可以让用户完全无须管理自己的服务器系统。那么容器技术的发展势头这么迅猛，还需要考虑无服务器 FaaS 吗?

我对于 PaaS 的主要观点依然适用于容器技术，对于无服务器 FaaS 来说，伸缩是自动实现的，是透明的，是非常细化的。容器平台目前还无法满足这些特征。

另外我想说的是，虽然过去几年来容器技术的流行度与日俱增，但这个技术依然尚未成熟。这不是说无服务器 FaaS 更成熟，只不过无论如何你只能在各种都不完善的技术中做出选择。

不过也要承认，随着时间流逝，这些争议终将消失。虽然容器平台目前还做不到无服务器 FaaS 那样程度的无需管理、自动伸缩等特性，但我们也注意到诸如 Kubernetes 的“Horizontal Pod Autoscaling”等技术正在继续向着这个目标努力。我认为这些功能会逐渐包含某些非常智能的流量模式分析，以及与负载更为密切相关的衡量指标。并且 Kubernetes 的快速演化也许很快就能为我们提供一个极为简单稳定的平台。

考虑到无服务器 FaaS 和托管式容器在管理和伸缩方面的差距，选择的余地也将缩小，可能只需要考虑应用程序的风格和类型。例如对于事件驱动的应用，并且每个应用程序组件只需要处理少数类型的事件，此时 FaaS 将是最佳选择；对于包含很多入口点，由同步请求驱动的组件，则更适合使用容器。我预计 5 年内很多应用程序和团队将同时使用这两种架构，很期待尽快看到这样的使用模式日益涌现。

#NoOps

无服务器并不是指无须运维，取决于你在无服务器的道路上能走多远，

可能是指“无须内部系统管理员”。这方面需要考虑两个重要问题。

首先，“运维”不光是服务器本身的管理，还意味着监控、安全、网络，以及大部分情况下必不可少的生产调试和系统伸缩。无服务器应用中依然存在这些问题，需要制定相应的策略。某些情况下无服务器世界中的运维可能更难，主要因为这还是一种比较新的技术。

其次，就算系统本身的管理工作也是存在的，只不过将其外包给了无服务器技术供应商。这不是什么坏事，很多东西都被我们外包了。但取决于你希望实现的精确程度，这种做法是好是坏还有待商榷，无论哪种方式，在某种程度上这样的抽象都有可能渗漏，你必须意识到依然是由人类系统管理员在为你的应用程序提供支持。

Charity Majors 在最近举办的无服务器大会上针对这个话题做了一次非常棒的演讲，建议当本次演讲发布到网上后大家都能去看看。在这之前你可以阅读她写的这篇以及这篇文章。

存储过程即服务

我发现无服务器 FaaS 还有着“存储过程即服务”的特征。我觉得这一特征主要源自很多 FaaS 函数（包括我在本文中用到的那些）都是围绕数据库访问编写的小规模代码片段。如果我们对 FaaS 的全部应用仅限于此，那么这名字还是很贴切的，但由于这只是 FaaS 各种能力中的一部分，对 FaaS 的这种看法实际上是一种无效的约束。

“我担心如果无服务器服务终将变成类似存储过程的技术，一个原本挺好的想法可能很快欠下一大堆技术债。 —— @skamille”

话虽如此，依然有必要考虑 FaaS 是否会遇到一些与存储过程相同的问题，例如在上述引用的推文中 Camille 提到的有关技术债的担忧。使用存储过程的过程中我们学到了很多经验，有必要在 FaaS 的世界中仔细反省一下这些问题，看看是否会出现类似的情况。存储过程所面临的一些问题包括：

1. 通常需要供应商明确指定的语言，或至少需要供应商针对某一语言指定的框架/扩展。
2. 由于需要在数据库的上下文情境中执行因此难以进行测试。
3. 很难像对待“一等公民应用程序”那样实现版本控制/处理。

并非所有存储过程的实现都会遇到上述问题，但我以前的工作中都遇到过。FaaS 是否也存在此类问题：

目前我所接触过的 FaaS 实现都无须担心（1），这一条可以划掉了。

对于（2），由于需要面对的“仅仅是代码”，单元测试过程将与其他任何代码一样易行。集成测试则是另一个截然不同（并且非常合理）的问题，下文将详细介绍。

对于（3），需要再次重申，由于 FaaS 函数面对的“仅仅是代码”，因此版本控制很好实现。但在应用程序打包方面目前还没有出现较为成熟的模式。上文提到的无服务器框架提供了自己的打包模式，AWS 也于 2016 年 5 月在无服务器大会上宣布自己也会开发一种打包技术（“Flourish”），但目前这依然是一个值得我们关注的问题。

无服务器架构（二）

作者 Mike Roberts 译者 大愚若智

收益

上文主要定义并解释了无服务器架构的含义。下文将探讨用这种方式设计和部署的应用程序所能获得的收益和存在的不足。

需要注意的是，其中一些技术还很新。截止撰写本文时，最领先的 FaaS 实现 AWS Lambda 诞生也还没超过 2 年。因此再过两年后，目前我们所获得的一些收益看上去可能也像是炒作，而目前的一些不足之处届时可能也已经妥善解决了。

由于这些结论尚未经过大范围的实践证实，你在决定使用无服务器技

术前一定要慎重考虑。希望本文列出的利弊清单可以帮你顺利做出决策。

首先准备展望一下美好生活，说说无服务器技术能带来的优势。

降低运维成本

从本质上来看，无服务器技术实际上是一种外包解决方案。该技术可以让你雇佣别人代替你管理服务器、数据库，甚至应用程序逻辑。由于使用的都是可以同时被他人共用的预定义服务，这里也存在着很大的规模经济效益，当一个供应商可以同时运行数千个类似的数据库时，每个用户支付的费用自然更低。

对你来说，成本的降低共体现在两方面：基础结构成本和人员（运维 / 开发）成本。虽然部分成本收益可能只来源于与其他用户分享基础结构（硬件、网络），但人们这样做的预期在于大部分情况下相比自行开发和托管的系统，用在外包无服务器系统上的时间会少很多（因此可以降低运维成本）。

然而这种收益和基础结构即服务（IaaS）或平台即服务（PaaS）所提供的收益并无太大区别。但我们可以通过两种主要方式对这些收益进行扩展，分别是无服务器 BaaS 和无服务器 FaaS。

BaaS：降低开发成本

IaaS 和 PaaS 基于这样的一种前提：服务器和操作系统的管理工作可成为一种商品化的服务。然而无服务器后端即服务可以让整个应用程序组件成为商品化的服务。

身份验证就是个很好的例子。很多应用程序需要开发自己的认证功能，其中通常会包含诸如注册、登录、密码管理、与其他认证服务供应商的集成等功能。总的来说大部分应用程序的此类逻辑都是类似的，因此诞生了类似 Auth0 这样的服务，可以让我们将已经创建好的认证功能集成在自己的应用程序中，无须自行开发。

BaaS 数据库也是类似的情况，例如 Firebase 的数据库服务。一些移动应用程序团队发现让自己的客户端直接与服务器端的数据库进行通信，

往往会采用这样的做法。BaaS 数据库避免了大部分数据库管理负担，此外这类服务通常还提供了可满足无服务器应用所需模式的，用于为不同类型用户提供所需认证方法的机制。

取决于你的知识背景，这些方式可能会让你感到坐立不安（别着急，具体原因会在下文不足之处一节详细介绍），但不可否认的是，很多打造出成功产品的成功公司所依赖的绝不仅仅是自己服务器端运行的代码。Joe Emison 在最近举行的无服务器大会上针对这个话题给出了几个例子。

FaaS：伸缩成本

正如上文所述，无服务器 FaaS 的好处之一在于“横向伸缩是完全自动化高弹性的，且将由服务供应商负责管理”。这种做法可以提供诸多收益，但从最基本的基础结构层面来说，最大的收益在于你只用为自己需要的计算能力付费，而在 AWS Lambda 这样的服务中甚至可以将计费粒度细化至 100 毫秒。取决于流量规模和类型，这一特性可能会为你提供巨大的经济效益。

范例：偶发请求

假设你运行的某个服务器应用程序只需要每分钟处理 1 个请求，每个请求的处理需要花费 50 毫秒，而一小时的时间内 CPU 平均使用率为 0.1%。从某种观点来看，这样的使用模式无疑是极为低效的，但如果有其他 1000 个应用程序和你共享 CPU，大家都可以用这一台服务器顺利完成自己的任务。

无服务器 FaaS 就是为了解决这种低效问题，并能帮你降低成本。在这个场景中你每分钟只需要为 100 毫秒的计算时间付费，只占时间总量的 0.15%。

这种方式还能提供下列连锁收益：

对于未来出现的，对负载要求非常低的微服务，可支持将整个组件拆分为不同的逻辑 / 领域，哪怕这种细化程度的运维成本原本根本不支持这

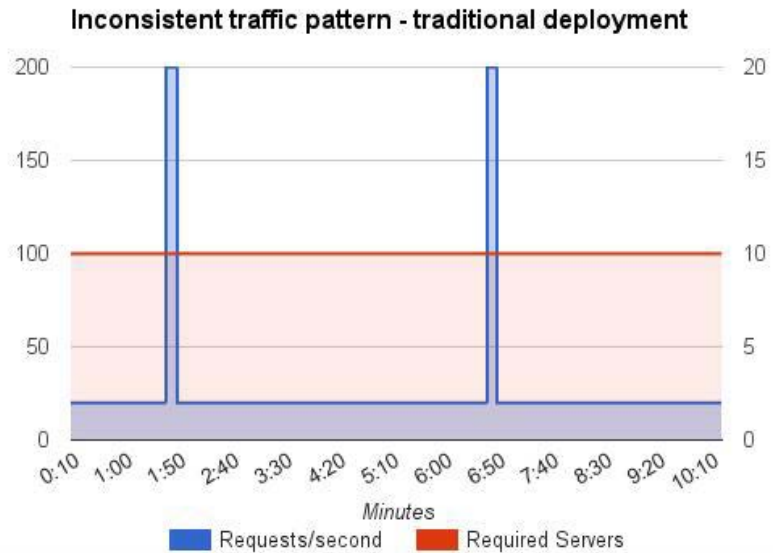
样做。

这样的成本收益还能促进民主化。很多公司或团队很可能想要尝试一些新技术，如果通过 FaaS 满足自己的计算需求，“试水”产生的运维成本将显得微乎其微。实际上如果你的所有工作负载相对较小（但也并非完全微不足道），甚至可能不需要为计算能力支付任何费用，因为某些 FaaS 供应商提供的“免费层”服务就足够了。

范例：不均匀的流量

再看看另一个例子。假设你的流量特征呈现明显的“峰谷”，也许基准流量仅为每秒 20 个请求，但每五分钟会遇到一次每秒 200 个请求（常规数量的 10 倍）并持续 10 秒钟的情况。出于范例的目的，我们假设基准性能就已经让服务器满载运行，但你不希望在峰值时期延长响应时间。如何解决这个问题?

在传统环境中，为了应对峰值需求可能需要将硬件总容量提升 10 倍，但这样的容量只在 4% 的总运行时间内可以得到充分利用。自动伸缩功能在这里可能并不是一种好的做法，因为新的服务器实例需要一定时间“热身”才能正常运转，但当新实例启动完成后，峰值时期已经结束了。



不过在无服务器 FaaS 环境中这一切不再是问题。甚至无须执行任何额外操作，可以直接认为流量是均匀分布的，并且只需要为峰值时期使用

的额外计算资源付费。

很明显，为了凸显无服务器 FaaS 在节约成本方面所提供的巨大收益，这里列举的例子是很有针对性的，目的在于证明除非流量非常均匀一致，并且能充分利用整个服务器系统的全部资源，否则仅从伸缩的角度来看，使用 FaaS 可以帮你节约大量成本。

但上述例子中有个问题需要注意：如果你的流量是均匀的并且始终能够充分利用服务器资源，你可能无法获得这样的成本收益，此时使用 FaaS 反而可能会花更多钱。因此需要针对当前供应商的成本，以及传统方式下同等容量、全时间运行的服务器成本进行一定的权衡，看看自己能否接受这样的成本。

优化是节约某些成本的根本

关于 FaaS 的成本还有个有趣的问题：对自己的代码进行任何性能优化，不仅可以加快应用运行的速度，而且取决于所选供应商计费粒度的细化程度，还会对运维成本的降低立刻产生直接影响。举例来说，如果目前每个操作需要花费 1 秒钟，但将时间缩短至 200 毫秒，无须改动基础结构即可在计算成本方面立即节约 80%。

简化运维管理

本节内容尤其需要注意：运维方面的某些问题对无服务器来说还较为麻烦，目前我们只准备谈谈该领域一些新出现的积极意义…

在无服务器 BaaS 端，运维管理工作比其他架构更简单的原因非常明显：支持的组件数量越少，工作量也就越低。

FaaS 端的特征比较多，下文将深入介绍其中的几个。

FaaS 伸缩能力提供的收益不仅限于成本

从上文的内容来看，伸缩是一种新特征，但也要注意 FaaS 的伸缩功能不仅可以降低计算成本，而且可以降低运维管理成本，因为伸缩工作可以自动完成。

如果伸缩过程是手工进行的，在最理想情况下，例如有专人负责给服

务器阵列添加或删除实例，但如果使用 FaaS 可以完全忽略这一问题，让 FaaS 供应商代替你对自己的应用程序进行伸缩。

就算在非 FaaS 架构中可以使用“自动伸缩”，但这一过程依然需要设置和维护，FaaS 完全无须执行这样的操作。

同理，因为伸缩是由供应商针对每个请求 / 事件分别进行的，你甚至再也不需要考虑自己最多可以处理多少并发请求才不至于耗尽所有内存或导致性能显著下降这样的问题，至少对于通过 FaaS 承载的组件无须考虑。由于负载可能激增，下游数据库和非 FaaS 组件也需要针对这种情况做好充分考虑。

降低程序包和部署的复杂度

虽然 API 网关本身比较复杂，但相比部署整个服务器，打包和部署 FaaS 函数的过程已大幅简化。只需要将代码编译并打包为 zip/jar 格式，随后上传即可。无须 puppet/chef，无须启动 / 停止 Shell 脚本，无须考虑是否要在计算机上部署一个或多个容器。对于新手甚至无须打包代码，可以直接在供应商的控制台内编写代码（当然不推荐用这种方式编写生产代码！）。

这个过程很简单，在一些团队中可以提供巨大的收益：整个无服务器解决方案无须进行任何系统管理。

平台即服务（PaaS）解决方案也能提供类似的部署收益，但正如上文对 FaaS 和 PaaS 对比时提到的，FaaS 的伸缩优势是独一无二的。

上市时间 / 实验

“简化运维管理”是工程师很熟悉的收益，但这对业务意味着什么？

最显著的意义在于成本：运维所需时间更少 = 运维需要的人员更少。但目前我认为最重要的意义在于“上市时间”。随着团队和产品变得愈加精益（Lean）和敏捷，我们会希望持续尝试新事物并快速更新现有系统。虽然直接重新部署即可对稳定状态的项目进行快速迭代，但更出色的新想法

到首次部署能力使得我们能用最小阻力和最低成本完成各种新实验。

FaaS* 新想法到首次部署 * 的特性很适合某些情况，尤其是对于通过供应商生态系统内已经成熟的事件触发简单函数这种情况。举例来说，假设你的组织正在使用类似 Kafka 的消息系统 AWS Kinesis 将不同类型的实时事件广播到整个基础结构。借助 AWS Lambda，可以在几分钟内为这种 Kinesis 流开发并部署新的生产事件监听器，一天之内就可以完成各种不同类型的实验！

对于基于 Web 的 API，由于存在各种不同用例，不能说真能有效简化运维，但各种开源项目和小规模的实现正在朝着这一目标努力。下文将进一步介绍。

“更绿色的”计算?

过去几十年来，全球数据中心的数量和规模都有了爆发式增长，这些数据中心的能耗，以及构建众多服务器、网络交换机等设备所需的物理资源用量也水涨船高。苹果、谷歌，以及其他类似的公司都开始谈到要将自己的数据中心部署在距离可再生能源更近的地区，以降低此类设施对化石燃料的用量。

这种显著增长的部分原因在于有越来越多的服务器大部分时候都是闲置的，但依然需要开机运行。

商用和企业环境中数据中心内典型的服务器在全年时间内只用到了运算能力总量的 5-15%。

—福布斯

这样的效率实在非常低，并会对环境产生巨大影响。

无服务器架构（三）

作者 Mike Roberts 译者 大愚若智

不足之处

亲爱的读者朋友们，希望上面提到的各种显著优势能让你满意，因为随后我们要用现实“打脸”了。

无服务器架构很多方面让人欣喜，如果不是感觉这种技术为我们做出了很多美妙承诺，我根本不愿意花时间介绍，但各种优势和收益都是需要付出代价的。其中一些代价是这种概念与生俱来的，无法通过进一步发展彻底解决，考虑使用无服务器技术时绝对不能忽略这些问题。另外有些代价来自无服务器技术目前的实现，随着进一步完善这些问题有望得到顺利解决。

与生俱来的劣势

供应商控制

与任何外包策略类似，你需要将某些系统的控制权拱手让给第三方供应商。此类控制力的缺乏可能体现为系统停机、非预期的限制、成本变动、功能缺失、强迫的 API 升级等。上文曾经提过的 Charity Majors 在这篇文章中“权衡”一节详细探讨了这个问题：

[供应商的服务] 如果足够智能，会对你使用服务的具体方式施加非常强大的约束，这样供应商才能以符合自己可靠性目标的方式供应这些服务。用户获得灵活性和丰富选项的同时，也会导致混乱和不可靠。如果平台供应商需要在你的幸福和数千个其他客户的幸福之间做出选择，他们绝对会选择“大多数”— 这也是他们应该做的。

— Charity Majors

多租户问题

多租户是指用同一台硬件甚至同一个托管应用程序，为多个客户（或租户）运行多个软件实例的做法。这种策略是实现上文所提到规模经济效益的关键。服务供应商会尽一切努力为自己的客户营造一种感觉，让客户认为自己是这套系统唯一的用户，优秀的服务供应商在这方面通常都做得很好。但凡事不可能完美，有时候多租户解决方案可能会造成一些问题，例如安全（一个客户可以看到另一个客户的数据）、健壮性（一个客户的软件错误导致其他客户的软件出错）、以及性能（高负荷客户导致其他客户的系统运行缓慢）。

无服务器系统也会面临这类问题，很多其他类型的多租户服务也不例外。但因为很多无服务器系统是新出现的，相比其他已经逐渐成熟的系统，无服务器可能会遇到更多此类问题。

供应商锁定

所有涉及第三方的系统都会遇到类似的问题：无服务器供应商锁定。大部分情况下，无论你使用了某一供应商的哪些无服务器功能，在另一个供应商的环境中这些功能都会使用不同的实现方式。如果想要更换供应商，

除了需要更新自己的运维工具（部署、监控等）外，八成还要更改代码（例如为了匹配不同的 FaaS 接口），甚至如果相互竞争的供应商在行为的实现上有差别，可能需要更改自己的决策或架构。

就算可以为生态系统的部分内容更换供应商，在其他架构组件上依然可能存在锁定。例如，假设你正在使用 AWS Lambda 对来自 AWS Kinesis 消息总线的事件做出响应，AWS Lambda、Google Cloud Functions 以及 Microsoft Azure Functions 之间的差异可能非常小，但依然无法将后两个供应商的实现直接挂接到 AWS Kinesis 流。这意味着无法将代码从一个解决方案移动或迁移至另一个解决方案，除非同时移动基础结构中的其他组件。

最终就算能找到某种方法通过其他供应商的服务重新实现整个系统，取决于新供应商所提供的服务，可能依然需要进行迁移。举例来说，如果从某个 BaaS 数据库切换至另一个，源数据库和想要换为使用的目标数据库提供的导出和导入功能可以满足你的需求吗？就算能满足，又需要付出多少成本和精力？

目前新出现了一种也许能缓解此类问题的方法：对多个无服务器供应商的服务进行普适的抽象，下文将详细介绍。

安全顾虑

这个问题本身就足以单独撰文介绍，拥抱全新的无服务器方法会让你面临大量安全问题，例如下文列举了两个例子，除此之外需要考虑很多。

你使用的每个无服务器供应商会增加你自己生态系统内所需安全实现的数量，并会增加面对恶意行为时的攻击面，有可能导致攻击成功率增加。

如果直接通过移动平台使用 BaaS 数据库，将丧失传统应用程序在服务器端提供的防护屏障。虽然这并不会直接造成严重后果，但依然需要在应用程序的设计和开发过程中给予更充分的考虑。

跨客户端平台重复实现相同的逻辑

在使用“完整 BaaS”架构后，不需要在服务器端编写任何自定义逻辑，所有逻辑都位于客户端。对于第一个客户端平台来说这也许不算什么问题，

可一旦需要支持更多平台，就必须重新实现相关逻辑的子集，在传统的架构中是不需要这样做的。举例来说，如果在此类系统中使用了 BaaS 数据库，所有客户端应用（也许有 Web，以及原生 iOS 和原生 Android 应用）都需要与供应商提供的数据库通信，此时就需要了解如何从数据库架构映射至应用程序逻辑。

此外如果任何时候想迁移到新数据库，还需要跨越所有不同客户端重复编写代码并对各种改动进行协调。

丧失优化服务器的能力

对于“完整 BaaS”架构，无法为了提升客户端性能而对服务器端的设计进行优化。“前端的后端（Backend For Frontend）”这种模式会对服务器上整个系统的部分底层内容进行某种程度的抽象，这种做法在一定程度上可让客户端执行速度更快，对于移动应用程序还有助于降低能耗。目前“全面的 BaaS”已经可以使用这样的模式。

需要澄清的是，这里以及上文提到的劣势适用于所有自定义逻辑都位于客户端，仅后端服务需要由供应商提供的“完整 BaaS”架构。为了缓解这些问题可以考虑采用 FaaS 或其他类型的轻量级服务器端模式，以将某些逻辑转移到服务器上。

无服务器 FaaS 不支持服务器内状态

在介绍过几个有关 BaaS 的劣势后，再来谈谈 FaaS 吧。上文我曾提到：

在本地…状态方面 FaaS 函数会遇到很多局限。你需要假设对于函数的任何调用，自己创建的任何进程内或主机状态均无法被任何后续调用所使用……

另外还提过，为了克服这些局限可考虑采用“十二要素应用”中的第六个要素：十二要素进程是无状态并且无共享（Share-nothing）的。任何需要持久保存的数据必须存储在有状态的后端服务中，通常可选择使用数据库。

Heroku 建议考虑这种方法，在使用 PaaS 时这些规则可以放宽，但使

用 FaaS 时往往无法通融。

那么如果无法保存在内存中，FaaS 的状态要存储在哪里？上文的引言中提到可以使用数据库，大部分情况下可以使用速度足够快的 NoSQL 数据库、进程外（Out-of-process）缓存（例如 Redis），或外部文件存储（例如 S3）等选项。但这些方式的速度比内存中或计算机内持久存储的速度慢很多，因此还需要妥善考虑自己的应用程序是否适合这些方式。

这方面另一个不容忽视的问题是内存中缓存。很多需要从外部读取大量数据集的应用会将数据集的部分内容缓存在内存里。你可以通过数据库中的“参考数据”表读取数据并使用诸如 Ehcache 等技术，或者从指定了缓存头的 Http 服务读取，此时内存中的 Http 客户端即可提供本地缓存。对于 FaaS 实现，可以将这些代码保存在应用中，但缓存很少能提供太多收益（可能完全无用）。只要缓存在首次使用时完成“热身”，随着 FaaS 实例被撤销这些缓存将毫无用处。

为了缓解这个问题可以不再假设存在进程中缓存，并使用诸如 Redis 或 Memcached 等低延迟的外部缓存，但这样做 (a) 需要执行额外的工作，并且 (b) 取决于具体用例可能会对性能产生极大影响。

实现方面的劣势

上文提到的劣势很可能会伴随无服务器技术一生。虽然可以通过各种缓解解决方案加以改进，但很可能无非根除。

然而其他劣势纯粹是由于现阶段该技术本身不够完善所致。随着供应商继续重视并不断投入，以及 / 或在社区的帮助下，这些问题有望彻底得到解决。只不过目前这些问题还比较突出……

配置

AWS Lambda 函数没提供任何配置选项。一个都没。甚至环境变量都无法自行配置。如何针对环境的某些特定本质用不同特征运行同一个部署构件？不行。你必须自行调整部署构件，甚至可能需要使用不同的嵌入式配置文件。这种“吃相”实在很丑。无服务器框架可以帮你完成必要的改动，

但这样的改动依然必不可少。

我有理由相信亚马逊正在解决这个问题（可能很快就会搞定），不知道其他供应商是否存在类似问题，但将这个问题放在开头恰好是为了证明这一技术目前确实还比较先进，有各种不足之处。

自己对自己发起的 DoS 攻击

为什么说任何时候使用 FaaS 都需要购者自慎（Caveat Emptor）？有另一个很有趣的例子。目前 AWS Lambda 会对用户可并发执行的 Lambda 数量进行限制，假设上限是 1000，这意味着任何特定时间点下你最多只能同时执行 1000 个函数，如果需要执行更多将开始受到限制，被加入等待队列，并 / 或导致整体执行速度放缓。

这里的问题在于这个限制会应用给你的整个 AWS 帐户。一些组织会为生产和测试环境使用同一个 AWS 帐户，这意味着如果组织内部某人在执行一种全新类型的负载测试，尝试执行超过 1000 个并发 Lambda 函数，这几乎等同于不小心对自己的生产环境发起了 DoS 攻击。晕…

就算为生产和开发环境使用不同的 AWS 帐户，一个超载的 Lambda（例如正在忙于处理客户批量上传操作）将可能导致其他由 Lambda 提供的实时生产 API 变得响应速度缓慢。

其他类型的 AWS 资源可以通过不同的安全和防火墙等手段，针对环境上下文情境和应用范围进行分隔，Lambda 也需要类似的机制，我相信很快就会有的，但目前只能自己小心了。

执行时间

上文提到过 AWS Lambda 函数如果执行时间超过 5 分钟将被终止，预计这个限制很快将被取消，但我更感兴趣的是 AWS 解决这个问题的方法。

启动延迟

上文提到过我的另一个顾虑，FaaS 函数等待多久才能获得响应，如果时不时需要在 AWS 上使用通过 JVM 实现的函数，这一点将尤为重要。如果你用到这样的 Lambda 函数，可能需要十多秒才能启动。

希望以后 AWS 能通过各种缓解措施改善启动速度，目前这一问题可能妨碍到某些用例中 JVM Lambda 的使用。

关于 AWS Lambda 的具体问题就是这些。我相信其他供应商私底下肯定也隐瞒了很多类似的问题。

测试

无服务器应用的单元测试其实相当简单，原因上文已经说过了：你所编写的任何代码“仅仅是代码”，而不是要使用的各种自定义库，也不是需要实施的各种接口。

然而无服务器应用的集成测试过程较为困难。在 BaaS 的世界中只能使用外部提供的系统，而不能使用（举例来说）自己的数据库，那么集成测试也要使用外部系统吗？如果要，这些外部系统与测试场景的匹配程度如何？能否轻松地组建 / 撤销所需状态？供应商能否针对负载测试提供单独的计费策略？

如果想将多个外部系统桩（Stub）在一起用于集成测试，供应商是否提供了本地的桩模拟器？如果有提供，桩的保真度如何？如果供应商没有提供桩又该如何自行实现？

FaaS 领域也存在类似的问题。目前大部分供应商并未向用户提供可用的本地实现，因此你只能使用普通的生产实现。这意味着所有集成 / 接受度测试都必须远程部署并使用远程系统执行。更糟糕的是，上文提到的问题（无法配置，跨帐户执行的限制）也会对测试方式产生影响。

说这个问题很严重，部分原因在于我们在无服务器 FaaS 中使用的集成单位（例如每个函数）远远小于其他架构，因此相比其他类型的架构，此时将更依赖于集成测试。

Tim Wagner（AWS Lambda 总经理）在最近的无服务器大会上简单提到了他们正在解决与测试有关的问题，但听起来测试工作对云的依赖会更高。这也许是一个适合勇敢者的全新世界，但我会怀念在自己笔记本上脱机对整个系统进行完整测试的日子。

部署 / 打包 / 版本控制

这是 FaaS 面临最具体的一个问题。目前我们还缺乏一种将一系列函数打包成应用程序的足够好的模式。造成这个问题的原因在于：

对于整个逻辑应用程序中的每个函数，可能都需要单独部署一个 FaaS 构件。如果（假设）你的应用程序使用 JVM 实现，其中包含 20 个 FaaS 函数，那就需要将 JAR 部署 20 次。

同时这也意味着无法以原子级的方式部署一组函数。你可能需要关闭可能触发该函数的任何事件源，部署整个组，然后重新打开事件源。对零停机应用程序来说这是个很麻烦的问题。

最后这还意味着应用程序缺乏版本控制机制，无法实现原子级回滚。

再次需要提醒，目前有一些意在解决这类问题的开源项目，然而只有在供应商的支持下此类问题才能顺利解决。为了解决部分此类问题，AWS 最近在无服务器大会上公布了一个名为“Flourish”的新技术，但具体细节尚未公布。

发现

与上文提到的有关配置和打包的问题类似，跨越不同 FaaS 函数的发现能力目前也缺乏一种足够好的模式。虽然这个问题不是 FaaS 独有的，但 FaaS 函数细化的本质，以及应用程序 / 版本定义的缺乏会让这个问题变得更严重。

监控 / 调试

目前只能使用供应商提供的监控和调试工具，无法使用第三方工具。某些情况下这样也可以接受，但对 AWS Lambda 来说，自带的工具功能过于简陋，这方面我们更希望有开放的 API 和第三方服务。

API 网关定义和“管太宽”的 API 网关

最近一次 ThoughtWorks Technology Radar 活动讨论了 API 网关“管太宽”的问题。虽然这个链接主要讨论了一般意义上的 API 网关，但正如上文所说，这些内容也适用于 FaaS API 网关。这里的问题在于 API 网关提供了在自己的配置 / 定义等领域内执行大量特定应用程序的能力，通常很难针对这些逻辑执行测试和版本控制等操作，甚至有时逻辑本身的定义也

很难。如果能像应用程序中的其他部分一样将此类逻辑留在程序代码中，这样的方式往往效果更好。

对于亚马逊的 API 网关，就算最简单的应用程序，目前也只能使用该网关特定的概念和配置。也正是因此出现了诸如无服务器框架和 Claudia.js 等开源项目，这类项目可以帮助开发者忽略具体实现的相关概念，更顺利地使用常规代码。

虽然 API 网关很容易会变得异常复杂，但随着技术的进一步完善，我们有望能用上帮助自己完成这些工作，并能通过推荐的使用模式帮我们远离这些陷阱的工具。

行动推迟

上文曾经提到，无服务器并不是指"无须运维"，在监控、架构伸缩、安全、网络等方面依然要做很多工作。但依然有一些人（也许我也算其中之一，这是我的错）会将无服务器描述为"无须运维"，主要原因在于当你开始使用之后，很容易会忽略掉与运维有关的活动："瞧啊，都不用装操作系统！"这种想法的危险之处在于很容易陷入一种虚假的安全感中。也许你的应用可以顺利运行，但在不知情的情况下被 Hacker News 报道之后，突然激增十倍的流量产生了近乎于 DoS 的攻击，然后就没有然后了……

与上文列举的有关 API 网关的其他问题一样，教育是解决这类问题的良药。使用无服务器系统的团队需要提前考虑运维活动，供应商和社区需要通过各种教学内容帮助他们了解这一技术的真正意义。

无服务器技术的未来

这篇无服务器架构的介绍文章已经逐渐到达尾声。作为收尾，我将谈谈未来几个月甚至几年里，无服务器技术可能的发展方向。

弥补劣势

上文已经多次提到，无服务器技术是新事物。尽管上文已经列举了该技术的一些劣势，但写出来的仅仅只是一部分。无服务器后续发展的重要方向之一是弥补固有劣势，消除，或至少改善实施方面的不足。

工具

在我看来，无服务器 FaaS 目前最大的问题在于工具的缺乏。部署 / 应用程序捆绑、配置、监控 / 日志，以及调试，这些工具都存在严重不足。

亚马逊已公布但尚未公开技术细节的 Flourish 项目也许能起到一定帮助。这则消息让人期待的另一个原因在于，该技术将会是开源的，这样就可以对不同供应商的应用程序进行移植。就算没有 Flourish，我们同样期待未来一两年里开源世界中能出现类似的技术。

监控、日志、调试，所有这一切都将由供应商负责实现，但对 BaaS 和 FaaS 领域都是巨大的促进。相比使用 ELK 等技术的传统应用，至少目前 AWS Lambda 的日志功能还很不能让人满意。但我们已经注意到在现在这样的早期阶段，这一领域已经出现了几个第三方的商用和开源工具（例如 IOPipe 和 lltrace-aws-sdk），但是距离类似 New Relic 这样的技术还有很长的路要走。希望 AWS 除了为 FaaS 提供更完善的日志解决方案外，也能像 Heroku 和其他厂商那样让我们更容易地融入第三方日志服务。

API 网关工具还有很大改进空间，其中一些改进可能来自 Flourish，或依然在继续完善的无服务器框架等类似技术。

状态管理

FaaS 缺乏服务器内状态，这一点对很多应用程序来说不是什么大问题，但也会对一些应用程序产生严重影响。例如很多微服务应用程序为了改善延迟会用到一定规模的进程中状态缓存。类似的连接池（连接到数据库，或通过持久的 Http 连接连接到其他服务）则又是另一种形式的状态了。

对于大吞吐率的应用程序，一种解决方法是让供应商延长函数实例的寿命，借此使用常规的进程中缓存方法改善延迟。但这种方法并非总是有效，因为不能为每个请求提供“温”缓存，而且用传统方式部署，并使用了自动伸缩能力的应用也面临类似的困扰。

另一种更好的解决方案是以延迟极低的连接访问进程外（Out-of-process）数据，例如用极低延迟的网络开销查询 Redis 数据库。考虑到亚马逊已经在自家的 Elasticache 产品中提供了托管式的 Redis 解决方案，并

且他们已经使用置放群组（Placement Group）实现 EC2（服务器）实例的相对共置（Relative co-location），这样的做法似乎无法获得足够的延展性。

我觉得更可行的情况是，我们将看到不同类型的应用程序架构会开始考虑非进程中状态的约束问题。对于低延迟应用程序实例，也许可以用普通服务器处理初始请求，通过本地和外部状态收集处理该请求所需的全部上下文，随后将包含完整上下文情境的请求交给自身无须查询外部数据的 FaaS 函数场来处理。

平台的改进

目前，无服务器 FaaS 的某些劣势主要源自平台本身的实现方式。执行时间、启动延迟、无法分隔的执行限制，这是目前最主要的三大劣势。也许可以通过新的解决方案解决这些问题，或可能需要付出额外的成本。例如我觉得启动延迟这个问题可以通过允许客户为 FaaS 函数请求 2 个始终可用，并且延迟足够低的实例的方式加以缓解，只不过客户需要为这样的可用性额外支付一笔费用。

当然，平台自身的继续完善不光是为了解决现有的这些问题，无疑还会提供更多让人激动的新功能。

教育

通过加强教育，也可以缓解不同供应商所提供的无服务器技术固有的一些劣势。每个使用此类平台的人都需要积极考虑将自己的生态系统托管在一个或多个应用程序供应商的平台上，这样的做法到底意味着什么。例如有必要考虑类似这样的问题：“是否有必要考虑使用来自不同供应商的平行解决方案，以防一个供应商的服务故障对我产生较大影响？如果部分服务故障，应用程序又该如何优雅地降级？”

技术运维方面也需要进行教育。目前有很多团队的“系统管理员”数量大幅减少，而无服务器技术还会让这种情况更严峻。但系统管理员的职责不仅仅是配置 Unix 服务器和 Chef 脚本，这些人通常也活跃在技术支持、网络、安全等一线领域。

在无服务器世界中，真正的 DevOps 文化变得愈加重要，因为依然有大量与系统管理无关的活动需要完成，并且通常将由开发者负责。但对大部分开发者和技术领导者来说，这些工作并非他们的本职任务，因此适当的教育以及与运维人员更为密切的合作将变得至关重要。

提高透明度 / 为供应商确定更清晰的预期

最后终于谈到了缓解措施方面，随着对供应商的托管能力愈加依赖，我们对不同供应商的平台会产生怎样的预期，供应商对此必须更加明确。虽然平台迁移工作很难，但至少是可行的，客户会逐渐带着自己的业务逃离不可靠的供应商。

新兴模式

除了半生不熟的底层平台，对于如何以及何时使用无服务器架构这样地问题，我们的理解依然还很浅显。目前很多团队会出于投石问路的考虑将各种想法付诸于无服务器平台，希望这些先驱们好运！

但是用不了多久我们就会开始看到各种推荐的实践模式日益涌现。

某些实践可能专注于应用程序的架构。例如 FaaS 函数在开始显得笨重之前能达到多大规模？假设能以原子级的方式部署一组 FaaS 函数，创建这种分组的最佳方式是什么？这种方式能否与目前我们将逻辑与微服务结合在一起所用的方法严格匹配？架构的差异是否要求我们转向截然不同的方向？

将这些问题进一步扩展，在 FaaS 和传统的“始终在线”持久运行的服务器组件之间创建混合架构的最佳方法是什么？将 BaaS 引入现有生态系统的最佳做法是什么？反过来看，哪些征兆意味着全面或大部分以 BaaS 为主的系统需要开始接受或使用更多自定义服务器端代码？

我们还会看到大量新兴的使用模式。媒体转换是 FaaS 的标准用例之一：“当有比较大的媒体文件存储到 S3 Bucket 后，自动运行进程在另一个 Bucket 中为该文件创建小体积的版本。”但为了确定某一具体用例是否适合无服务器方法，我们还需要进一步分析更多使用模式。

除了应用程序架构，一旦相关工具进一步完善后，我们还将看到各种推荐的运维模式。如何从逻辑上将 FaaS、BaaS，以及传统服务器组成的混合架构中的日志汇总到一起？关于服务的发现有什么好办法？对于以 API 网关作为前端的 FaaS Web 应用程序该如何进行金丝雀发布？如何对 FaaS 函数进行最高效的调试？

超越“FaaS化”

目前我见到的大部分 FaaS 运用场景主要是将现有代码 / 设计想法用“FaaS”的方式实现出来，也就是将其转换为一系列无状态的函数。这种做法挺强大，但我也期待着能实现进一步的抽象甚至形成一种语言，使用 FaaS 作为底层实现为开发者提供 FaaS 收益，而无须实际将自己的应用程序看作一系列相互独立的函数。

例如我不知道谷歌是否在自己的 Dataflow 产品中使用了 FaaS 实现，但我完全可以设想有人创造了能实现类似作用的产品活开源项目，并使用 FaaS 作为实现。这里可以用 Apache Spark 作为类比。Spark 是一种大规模数据处理工具，提供了非常高程度的抽象，可支持使用 Amazon EMR/Hadoop 作为自己的底层平台。

测试

正如上文“劣势”中所述，对无服务器系统来说，在集成和接受度测试方面还有很长的路要走。我们会看到不同供应商分别提出自己的建议，其中一些可能会用到云服务，同时我猜测可能还会有像我这样的“保守派”供应商会提出另一种方案，借助何种方案可以在自己的开发计算机上完成一切测试任务。估计最终我们将看到各种得体的解决方案，分别可用于联机和脱机测试，不过可能还要等几年。

“可移植”的实现

目前所有流行的无服务器实现都以部署到第三方供应商在云中的系统内为前提。这是无服务器的优势之一，可以减少需要我们维护的技术数量。

但这就产生了一个问题，如果有公司希望在自己的系统中运行这些技术，并将其以一种内部服务的方式交付该怎么办?

类似的，目前的所有实现在集成点（Integration point）方面都有自己的偏好：部署、配置、函数接口等。这就会造成上文提到过的供应商锁定的局面。

为了缓解这些顾虑，期待能看到各种可移植的实现，下文将首先谈谈上面的第二点。

针对供应商的实现进行的抽象

我们已经开始看到类似无服务器框架和 Lambada 框架这样的开源项目。这些项目的想法在于让我们忽略实际部署位置和方法，以一种中立的开发方式编写和运维无服务器应用。就算目前仅仅在 AWS API 网关 + Lambda，以及 Auth0 Webtask 之间，如果能根据每个平台的运维能力轻松进行切换，那也是极好的。

我觉得只有在出现大量此类标准化产品之后才能实现这种程度的抽象，但好在这种想法可以通过循序渐进的方式逐步实现。我们可以从一些跨供应商的部署工具着手，甚至可以考虑上文提的 AWS Flourish，并以此为基础逐渐构建出更多功能。

这方面有个比较棘手的问题：在尚未实现标准化的情况下对 FaaS 编码接口的抽象进行建模，预计这样的进展会首先出现在非专有的 FaaS 技术中。例如我认为在 AWS S3 或 Kinesis Lambdas 实现抽象之前，可能首先会完成对 Lambda 的 Web 请求和调度（“Corn”）的实现。

可部署的实现

使用无服务器技术，但不使用第三方供应商的服务，这样的做法听起来有些奇怪，但可以考虑这些情况：

也许有一家大型技术组织，希望开始为所有移动应用程序开发团队提供类似 Firebase 的数据库体验，但想使用现有的数据库架构作为后端。

希望为某些项目使用 FaaS 风格的架构，但出于合规性 / 法律等问题

的考虑，需要在“本地”运行自己的应用程序。

上述任何一个用例都可以在不使用外部供应商托管服务的情况下通过无服务器的方式获得收益。这样的做法是有先例的，例如平台即服务（PaaS）。最初流行的 PaaS 都是基于云平台的（例如 Heroku），但人们很快发现在自己系统中运行 PaaS 环境也能带来大量收益，这就是所谓的“私有 PaaS”（例如 Cloud Foundry）。

可以想象，与私有 PaaS 实现类似，开源和商用的 BaaS 和 FaaS 等概念的实现将愈加流行。Galactic Fog 是在这一领域应用尚处在早期阶段的开源项目的典范，他们就使用了自己的 FaaS 实现。与上文提到的供应商抽象的观点类似，我们可能会看到一些循序渐进的方法。例如 Kong 项目是一种开源的 API 网关实现，虽然在撰写本文时尚未与 AWS Lambda 集成（但据说这个问题正在着手处理中），但实现后就可以为我们提供很多有趣的混合方法。

社区

真心希望无服务器社区能够发展壮大。目前全球范围内围绕该技术已经有一个大会以及大量聚会活动。希望这一领域也能像 Docker 和 Spring 那样在全球发展出更多更大规模的社区。更多大会，更庞大的社区，以及各种在线论坛，借此帮助大家跟上技术的发展。

结论

虽然名称容易让人产生歧义，但“无服务器”这种风格的架构可以帮助我们减少在自己服务器端系统上运行的应用程序代码数量。这是通过两种技术实现的：后端即服务（BaaS），借此可将第三方远程应用程序与我们的前端应用直接进行紧密的集成；以及函数即服务（FaaS），借此可将不间断运行的组件中执行的服务器端代码转移到短暂运行的函数实例中执行。

无服务器技术并不是所有问题的最终答案，如果有人跟你说这种技术将彻底取代现有架构，你需要小心对待。如果想现在就涉足无服务器，尤

其是 FaaS 领域，也需要更加谨慎。虽然这种技术已经产生了累累硕果（伸缩、节约开发工作量等），但依然有（调试、监控等领域的）困难在下个角落等着你。

这些收益并不会被人们快速淡忘，毕竟无服务器架构的积极意义实在是太大了，例如降低运维和开发成本，简化运维管理，降低对环境的影响等。对我来说，影响力最大的收益依然是减少打造全新应用程序组件时的反馈环路（Feedback loop），我是“精益（Lean）”方法的铁粉，主要是因为我觉得尽可能先于最终用户从技术中获得反馈，这种做法可以为我带来大量价值，当无服务器技术能够与这样的理念相符后也可以缩短我将应用投放时长所需的时间。

无服务器系统依然处在襁褓中。未来几年里该技术还有很大的进步空间，我已经迫不及待想看看这个技术将如何融入现有的架构大家族中。

作者简介

Mike Roberts 是一位工程主管，目前居住在纽约市。虽然目前大部分时间都在管理人员和团队，但他也负责管理代码，尤其是 Clojure，他对软件架构有着深入的见解。他对无服务器架构抱有审慎乐观的态度，认为这种技术也许担的上目前所获得的一些炒作。

观点与趋势

2017 年会是 Serverless 爆发之年吗?

作者　麦克周

前言

中小型公司，尤其是互联网行业的创业公司，本身并没有太多的技术人员，如果设计系统时需要考虑诸多的技术问题，例如 Web 应用服务器如何配置、数据库如何配置、消息服务中间件如何搭建等等，那对于他们来说人员成本、系统成本会很高，Serverless 架构的出现，让这种情况可能可以大幅度改善。

初识 Serverless

在目前主流云计算 IaaS（Infrastructure-as-a-Service，基础设施即服务）和 PaaS（Platform-as-a-Service，平台即服务）中，开发人员进行业务开发时，仍然需要关心很多和服务器相关的服务端开发工作，比如缓存、消

息服务、Web 应用服务器、数据库，以及对服务器进行性能优化，还需要考虑存储和计算资源，考虑负载均衡和横向扩展能力，考虑服务器容灾稳定性等非专业逻辑的开发。这些服务器的运维和开发知识、经验极大地限制了开发者进行业务开发的效率。设想一下，如果开发者直接租用服务或者开发服务而无须关注如何在服务器中运行部署服务，是否可以极大地提升开发效率和产品质量？这种去服务器而直接使用服务的架构，我们称之为 Serverless 架构（无服务器架构）。

Serverless 架构的问世

2014 年，云厂商 AWS 推出了“无服务器”的范式服务。

其实，最初“无服务器”意在帮助开发者摆脱运行后端应用程序所需的服务器设备的设置和管理工作。这项技术的目标并不是为了实现真正意义上的“无服务器”，而是指由第三方供应商负责后端基础结构的维护，以服务的方式为开发者提供所需功能，例如数据库、消息，以及身份验证等。这种服务基础结构通常可以叫做后端即服务（Backend-as-a-Service，BaaS），或移动后端即服务（MobileBackend-as-a-service，MBaaS）。

现在，无服务器架构是指大量依赖第三方服务（也叫做后端即服务，即“BaaS”）或暂存容器中运行的自定义代码（函数即服务，即“FaaS”）的应用程序，函数是无服务器架构中抽象语言运行时的最小单位，在这种架构中，我们并不看重运行一个函数需要多少 CPU 或 RAM 或任何其他资源，而是更看重运行函数所需的时间，我们也只为这些函数的运行时间付费。无服务器架构中函数可以多种方式触发，如定期运行函数的定时器、HTTP 请求或某些相关服务中的某个事件。

Serverless 案例

以带有服务功能逻辑的传统面向客户端的三层应用为例（一个典型的电子商务应用网站）。一般来说包含客户端、服务端程序、数据库，服务端用 Java 开发完成，客户端用 JavaScript。

采用这种架构，服务端需要实现诸多系统逻辑，例如认证、页面导航、搜索、交易等都需要在服务端完成。如果采用 Serverless 架构来对该应用进行改造，则架构如图所示：

- Serverless架构相比于传统面向客户端的三层应用架构，有以下几方面的差异：
- 删除认证逻辑，用第三方BaaS服务替代；
- 使用另外一个BaaS，允许客户端直接访问架构与第三方（例如AWS Dynamo）上面的数句子库。通过这种方式提供给客户更安全的访问数据库模式；
- 前两点中包含着很重要的第三点，也就是以前运行在服务端的逻辑转移到客户端中，例如跟踪用户访问。客户端则慢慢转化为单页面应用。

计算敏感或者需要访问大量数据的功能，例如搜索这类应用，我们不需要运行一个专用服务，而是通过 FaaS 模块，通过 API Gateway 对 HTTP 访问提供响应。这样可以使得客户端和服务端都从同一个数据库中读取相关数据。由于原始服务使用 Java 开发，AWS Lambda（FaaS 提供者）支持 Java 功能，因此可以直接从服务端将代码移植到搜索功能，而不用重写代码。

最后，可以将其他功能用另外一个 FaaS 功能取代，因为安全原因放在服务端还不如在客户端重新实现，当然前端还是 API Gateway。

常见的 Serverless 框架介绍

Amazon的Lambda产品

2014 年 11 月 14 日，AWS 发布了 AWS Lambda。AWS Lambda 是市面上最早，也是最为成熟的 Serverless 框架之一。该服务最迟支持 Node.js，现在也支持 Java 和 Python。它与 Alexa Skills Kit（软件开发工具包）紧密集成，亚马逊提供交互式控制台和命令行工具，以便上传和管理代码

片段。

Google Cloud Functions

Google 是为服务架构的最前沿公司，除了推动 Kubernetes，Google 还投资了 Cloud Functions，该架构可以在其公共云基础设施上运行。

Iron.io

Iron.io 最初是为企业级应用提供微服务。Iron.io 是用 Go 语言编写的，用于处理高并发、高性能计算服务，并已经集成 Docker 服务，提供一种完整的微服务平台。

IBM OpenWhisk

2016 年 2 月的 InterConnect 大会，IBM 发布了 OpenWhisk，这种事件驱动型开源计算平台可以用来替代 AWS Lambda。OpenWhisk 平台让广大开发人员能够迅速构建微服务，从而可以响应诸多事件，比如鼠标点击或收到来自传感器的数据，并执行代码。事件发生后，代码会自动执行。

Serverless Framework

Serverless Framework 是无服务器应用框架和生态系统，旨在简化开发和部署 AWS Lambda 应用程序的工作。Serverless Framework 作为 Node.js NPM 模块提供，填补了 AWS Lambda 存在的许多缺口。它提供了多个样本模板，可以迅速启动 AWS Lambda 开发。

Azure WebJobs

Azure Web 的应用功能，可以与 Web、API 应用相同的上下文中运行程序或脚本。可以上传并运行可执行文件，例如 cmd、bat、exe、psl 等等。WebJobs 提供 SDK 用于简化针对 Web 作业可以执行的常见任务，例如图像处理、队列处理、RSS 聚合、文件维护，以及发送电子邮件等等。

Serverless 架构原则

- 按需使用计算服务执行代码

Serverless 架构是 SOA 概念的自然延伸。在 Serverless 架构中，所有自定义代码作为孤立的、独立的、细粒度的函数来编写和执行，这些函数在 AWS Lambda 之类的无状态计算服务中运行。开发人员可以编写函数，执行常见的任务。在比较复杂的情况下，开发人员可以构建更复杂的管道，编排多个函数调用。

- 编写单一用途的无状态函数

单单负责处理某一项任务的函数很容易测试，并稳定运行。通过以一种松散编排的方式将函数和服务组合起来，能够构建易于理解、易于管理的复杂后端系统。

为 lambda 等计算服务编写的代码应该以无状态方式进行构建，这样会让无状态功能很强大，让平台得以迅速扩展，处理数量不断变化的请求或者事件。

- 设计基于推送的、事件驱动的管道

可以构建满足任何用途的服务器架构。系统可以一开始就构建成无服务器，也可以逐步设计现有的单体型应用程序，以便充分发挥这种架构的优势。最灵活、最强大的无服务器设计是事件驱动型的。

构建事件驱动的、基于推送的系统常常有利于降低成本和系统复杂性，但是要注意，并不是任何情况下都是适当的或者容易实现的。

- 创建更强大的前端

由于 Lambda 的定价基于请求数量、执行时间段以及分配的内存量，所以代码执行需要越快越好。数据签名的令牌让前端可以与不同的服务直接通信。相比之下，传统系统中所有通信经由后端服务器来实现。让前端与服务进行通信有助于减少创建环节、尽快获得所需的资源。

- 与第三方服务集成

如果第三方服务能提供价值，并减少自定义代码，那么自然它们就很有价值。开发人员可以通过引入第三方服务来减少自己实现各种业务逻辑的需要，可以减少小型公司的开发成本，避免价格、性能、可用性等要素上的劣势。

未来趋势

随着移动和物联网应用蓬勃发展，伴随着面向服务架构（SOA）以及微服务架构（MSA）的盛行，造就了 Serverless 架构平台的迅猛发展。在 Serverless 架构中，开发者无须考虑服务器的问题，计算资源作为服务而不是服务器的概念出现，这样开发者只需要关注面向客户的客户端业务程序开发，后台服务由第三方服务公司完全或者部分提供，开发者调用相关的服务即可。Serverless 是一种构建和管理基于微服务架构的完整流程，允许我们在服务部署级别而不是服务器部署级别来管理应用部署，甚至可以管理某个具体功能或端口的部署，这就能让开发者快速迭代，更快速地交付软件。

这种新兴的云计算服务交付模式为开发人员和管理人员带了很多好处。它提供了合适的灵活性和控制性级别，因而在 IaaS 和 PaaS 之间找到了一条中间道路。由于服务器端几乎没有什么要管理的，Serverless 架构正在彻底改变软件开发和部署流程，比如推动了 NoOps 模式的发展。

无服务器架构将 DevOps 带入新层次

作者 Mike Roberts 译者 大愚若智

无服务器架构近在眼前

无服务器计算正改变着软件系统构建和运营的方式。尽管它是 IT 行业中一个相对较新的领域，但它可能会大大改变软件行业业务价值的交付方式。它可以使用可用和可扩展的云端负载来以较低的成本运行项目，这对许多产品类型和业务用例来说是一种理想的方式。

但无服务器架构不仅仅只改变了软件交付的方式，它还会改变软件开发组织本身，相信这点对 IT 行业产生的影响将更加深远。在本文中，我们将探讨无服务器架构如何改变那些用其发布软件的组织的文化，以及它是如何影响整个行业的未来的。

DevOps 目前的状态

在过去几年没有太脱离业界环境的人，一定听说过 DevOps。DevOps 运动将敏捷软件开发融入并扩展到 IT 运营领域，旨在通过促进开发和运营团队之间的强力协作并采用新颖的运营实践来提供更高的业务敏捷性，尤其是在基础设施配置、改进发布管理和运营工具方面。

事实上，DevOps 正在成为 IT 行业的新标准，并且已经被业界广泛采纳，常见于云计算和容器技术。同时，许多组织正尽力去理解 DevOps 的全貌，这主要受限于他们专业知识上的缺乏和各种组织结构上的挑战。尽管面对这些挑战，DevOps 正在成为一个主流运动，它正改变着 IT 组织发布软件的方式，这就像敏捷运动在过去十多年中所产生的影响。

但是，无服务器架构是如何适应 DevOps 文化的？它将如何影响常规的 DevOps 实践呢？

为什么要选择无服务器架构?

为了了解无服务器架构是怎样影响那些使用它的组织的，让我们首先来看看用它进行构建和运行软件系统所具备的关键特性。

功能即服务（FaaS）提供了一个托管的运行时，用于执行任何已经上传到服务上的代码。这可能看起来就像将可运行的项目部署到计算机实例或服务器上，并在操作系统上执行它，但实际上这并不相同。FaaS 在保证功能在满足当前需求规模下可用的同时，只以执行次数和运行时间收取费用。同时它会抽象出实际的运行时（如 Java 虚拟机或 NodeJS）和操作系统本身的配置。在其背后，运行时进程、操作系统和计算实例还是在运行着的（不要被“无服务器”这个名字蒙骗了），但开发人员不再需要担心这些因素了。

这正是无服务器架构的优点，整个计算堆栈，包括运行功能代码的操作系统进程，完全由云提供商管理。与传统的基础架构即服务（IaaS）模型相比，这种方式大大简化了运算基础架构的管理，并结合了按使用进行

收费的计费模式，提供了非常灵活且经济的运算选型。

快速开发

除了FaaS计算（如AWS Lambda、Azure Functions以及Google Functions）之外，公共云提供商还提供了一系列其他服务用于组合并创建无服务器架构。从可伸缩的持久化存储和消息中间件，到API网关和内容分发网络，如今想要构建一个完整的系统完全不需要直接摆弄服务器。

每个云提供商的服务都可以通过其提供的软件开发工具包（SDK）进行全方位的配置，可以用其快速地发布产品来提供业务价值，前提是你要熟悉可用的服务及其配置选项。每个功能往往只负责处理简单的事件或请求，因此通常它们不需要大量代码，小而集中的业务逻辑就足够了。例如，一个功能可能只负责根据数据库表中的触发器，将变化信息推送到用户的电子邮件或相应的消息队列上，让其他子系统可以使用这个通知来更新外部系统。

然后，许多这样的功能用于实现业务逻辑及连接服务，从而提供了持久化、消息、集成、内容分发、机器学习等基本功能。这些服务解决了许多复杂的项目问题，并可以用其来创建复杂的解决方案而不会碰到太多困难，进而可以快速进行原型设计并开发。

从一开始就考虑维护性

使用了无服务器架构，就不可能在不考虑代码执行方式以及其他所需资源的情况下开始编写代码，至少这样做毫无意义。毕竟，为了了解代码如何与API网关、数据存储或消息中间件交互，首先必须部署代码，还要配置所有相关资源。虽然，可以使用模拟，而不通过真实的部署来执行代码，但这只提供了有限程度的验证，况且，这样不会运行该功能所需的整个基础架构堆栈。

无服务器架构需要配置好云资源这点可以说有利也有弊。那些习惯于使用自己机器，在本地开发模式下运行应用程序和系统的用户，很可能会因为较长的反馈周期而损失部分生产力。基础设施配置和代码部署确实需

要更多的时间，但并不会像 IaaS 一样多，后者还要算上按需启动计算实例的时间。

从一开始就强制关注基础设施堆栈的主要好处是，能早在编写代码的时候，就考虑基础架构设置和配置机制。这与现在仍常见的传统方法不同，常常开发人员编写代码，并借助于持续集成工具进行打包，然后将其转交给运营团队进行部署，在这个过程中会假设不用考虑网络基础设施的问题。

DevOps 运动促进了开发和运营团队之间的合作，而在无服务器架构中，他们就根本不可能被分开。

在无服务器架构中，即使部署一个简单的功能，也需要对一些运营和财务方面的关键问题作出决定。两个最基本的配置选项就是可用内存和超时时间（即功能调用的预估时间）。这两个设置都会影响调用所需的花费，因为它是按照内存消耗和执行时间来收费的。此外，分配的内存通常与功能运行的计算实例相关联，更多的内存就意味着更多的处理能力。

由于需要这么多次对功能的配置调优，根据可用的预算及期望和观察到的性能特性对设置进行快速地调整就极为重要了。这些特性可以通过云提供商收集并公开的指标进行确定，AWS CloudWatch 就是一个监控服务的例子。实际上，在构建无服务器架构时拥有丰富的 FaaS 和其他服务的指标对于是否可以运营这个架构至关重要。由于在配置资源后立即可以得到这些指标，所以在开发阶段就可以，也应该考虑架构的许多运营问题，如性能优化、容量规划、监控和记录。

安全性是软件交付方面另一个很好的例子，通常它是被放在项目后期来解决的，或被委派给专门的安全团队来处理，在部署到生产环境之前由他们对所有软件组件进行评估和签发。在无服务器架构中，在常规开发活动部署的一开始，就必须考虑安全性。至少每个功能必须有与之相关联的安全策略。由于一个功能可以被同账户下的任何其他资源所访问到，所以花费一些时间来确定并配置正确的基于任务的功能安全策略很有必要。理想情况下，按照最小权限的原则，一个功能应该被赋予它所需的最小权限集。例如，需要查询数据库表的功能只能具有查询相关表的权限。

显然，无服务器架构应该使可维护性（包括安全性）成为正常开发周期的一部分，而不是将这些要素推迟到运营团队参与后再进行，不然就会失去解决问题的最佳时机。

当谈到无服务器架构时，DevOps 的思想并不是用来被逐步接受的（通常这样会代来巨大的痛苦），而是需要刻在其底层的基因上。

按使用收费

与 IaaS 计算模型相比，无服务器架构带来了另一个革命性的变化，即对单个功能调用进行收费的定价模型，而不必为保持服务器运行进行付费。

使用公共云的组织更习惯于将云基础设施成本看作运营支出（OPEX）而不是资本支出（CAPEX），但是在 IaaS 架构中，他们最终往往会进行大量前期投资以降低总成本，例如预留计算实例或购买其他云服务的预留容量。而在无服务器架构中，这就不必要也不经济了，因为只对功能调用进行支付比保持服务器持续运行会便宜许多。

由于用于构建无服务器架构的大部分服务都是按使用进行计费的，这样就可以运行多个环境以支持软件交付涉及的开发、测试和操作活动。毕竟，如果不进行调用，就不会产生很多花费，甚至根本不需要支出。无服务器架构在成本上的影响正在消除 DevOps 在许多公司实践过程中的诸多障碍。

能够拥有尽可能多的环境来满足各种团队或业务利益相关者的需求，会带来一些新的巨大的可能性。例如，每个开发人员可以在云上拥有个人的开发环境，或者正在开发的每个功能都可以部署到专用环境中，从而可以独立于其他任何任务进行演示。这样的独立环境甚至可以在单独的提供者帐户上托管，以提供终极的隔离。

持续部署将成为新常态

持续交付是使 DevOps 可行的关键功能之一，但对许多公司来说，尤

其是在企业领域，这点仍然相当难以实现。虽然持续交付提供了许多好处，并实现了更高的业务敏捷性，但它没能了解到组织的全部潜力。

无服务器架构可以用来实现业务灵活性的最高境界，即持续部署。持续部署让任何合并到主干中的代码更改都自动升级到包括生产在内的所有环境。为了让这种方式在不影响用户的情况下工作，持续部署的系统显然需要从不同的角度进行严格的质量检查。

鉴于诸如基础架构配置和安全性等运营问题可以也应该在功能代码的开发阶段解决，基础设施堆栈就可以从一开始就配置好，或根据源代码仓库中包含的代码和配置进行更新。这些堆栈可以由提供商提供的原生工具（如 AWS 的 Cloud Formation），或其他通用工具（如 Hashicorp Terraform）进行管理。

通过全自动化的基础设施堆栈的配置和代码部署，就可以对任何环境进行应用或取消（回滚）变更，当然这其中也包括生产环境这一环节。为了保证万无一失，在部署或整个流程结束后需要自动在各个相关环境运行那些确保系统质量的测试案例，包括功能性和非功能性的测试。

每个人都是云工程师

无服务器架构模糊了软件交付过程中常涉及的各类技术角色之间的界限。传统的架构师、开发人员、测试人员、数据库管理员、运营和安全工程师将共同合作来发布系统并维护生产环境，在无服务器架构的世界中，这些角色都会被合并为云工程师。

正如许多传统开发过程已被移除或被大大简化一样，如今已经不再需要在项目中引入诸多专家。相反，具有广泛技能且熟悉云提供商平台的工程师就可以完成这些工作，甚至更多，同时也可以做得更快。许多开发和运营过程可以被合并到同一个周期内，并且可以完全消除昂贵的交接或从外部借用资源的成本。

但这并不意味着团队中不再拥有专门从事特定领域的人员，毕竟每个人会自然而然地更偏重于软件交付的某些方面，但理想情况下，团队中的

每个成员都应该能够参与到发布一个功能的所有流程中，包括在生产环境中进行运营。这是激励所有工程师能在一开始就构建一个可维护的高质量软件的最佳方法。

实际上，与那些谈论着要拉近开发和运营距离的团队不同，使用无服务器的团队天生就有着 DevOps 的文化，即软件在开始构建阶段就准备着运行在生产环境中。

适者生存

无服务器架构可以用来实现终极的业务敏捷性。然而，这完全取决于组织理解无服务器架构全貌的能力。虽然许多组织仍在努力建立某种形式的 DevOps 文化和实践，无服务器架构提供了一种全新的方式来创建快速业务价值交付和稳定运营的文化，同时最大限度地降低成本。

并没有多少组织可以接受无服务器架构这个新领域所带来的挑战，因为整个领域仍然非常年轻并且还不成熟，所以要接纳它真的需要很大的勇气，因此需要大量额外的工作来弥补目前初级阶段所带来的差距及挑战。很多健全且愿意采纳无服务器架构的组织，可能会发现自己还在试图套用他们现有的流程和组织结构，并且失去他们已有的敏捷性，或更糟糕的是，还在建立和运行无服务器架构上花费了大量的精力。

那些希望能够充分利用无服务器架构来获得在市场中竞争优势的公司，可能不仅需要调整他们提供软件的方式，还需要改变其产品的创建和销售方式。

结论

无服务器架构不仅补充了 DevOps 的理念，更改进了当前 IT 组织实现更高业务敏捷性的观念。它致力于快速交付商业价值并持续改进和学习，这极有可能会带来文化上大范围的改变，甚至对那些已采用了 DevOps 文化和实践的组织也不例外。

使用无服务器架构不仅可以使组织更快更省地提供新产品和功能。它

还将改变整个流程中的内在文化。

作者简介

Rafal Gancarz 是 OpenCredo 的首席顾问，OpenCredo 是一家位于伦敦的咨询公司，专门帮助客户构建并部署新兴技术以实现客户的业务价值。Rafal 是一位经验丰富的架构和大规模分布式系统发布方面的专家。他也是一位经验丰富的敏捷实践者和认证敏捷专家（Certified Scrum Master），他对改进项目交付抱有极大兴趣。在过去的 18 个月中 Rafal 使用 AWS 平台上的无服务器技术开发大型项目，并拥有使用无服务器堆栈构建企业级分布式系统的第一手经验。

我对无服务器架构的一些看法

作者 Will James 译者 大愚若智

我想通过本文简单谈谈自己对奥斯丁 Serverlessconf 大会上所涉及主要议题的看法。这次活动让我受益匪浅，还见到了不少行业牛人，万分感谢 A Cloud Guru 举办的这次活动。此外我还领到了很多新 T 恤，吃了美味的甜甜圈，拜访了住在奥斯丁的亲友们。

总的感想如下。

什么是“无服务器”？

Serverlessconf 的所有与会者依然还在忙着给“无服务器”下定义，并达成了一些共识：

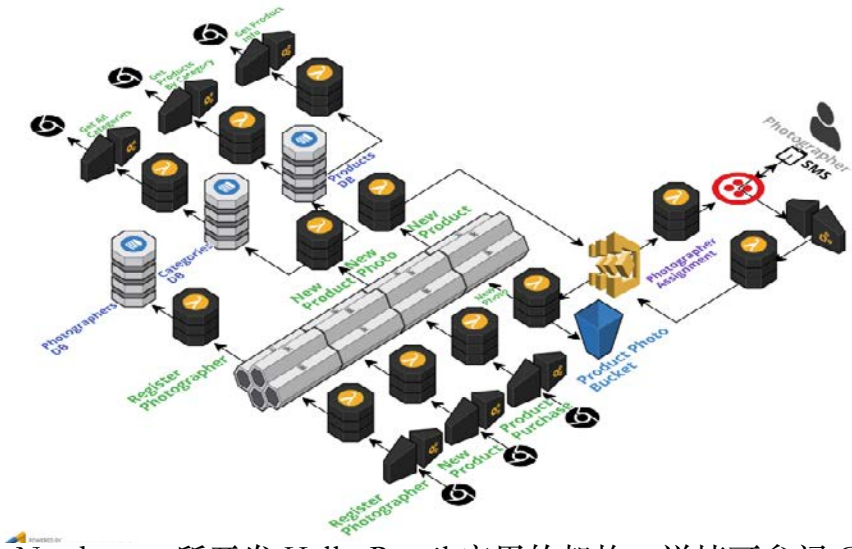
1. 无服务器不仅仅是函数即服务 (FaaS, Functions As A Service)，还包含诸如数据库、身份验证、API网关、编排，甚至具体到某一领域的其他服务，例如视频转码即服务或认知服务。总的来说，所有与这些服务有关的基础架构都不需要我们自行管理。
2. 无服务器意味着（近似于）100%的利用率。如果和PaaS相比的话，PaaS应用程序要么以特定的规模运行，要么以非常慢的速度进行伸缩，但会因为伸缩操作本身造成一定的开销（例如有未使用的实例处于闲置状态，等待接受请求）。作为对比，如果无服务器服务暂不使用，此时不会产生任何成本，但如果有必要可以（几乎）瞬时伸缩至数百万用户，服务的成本直接取决于使用量。

事件驱动，而非数据驱动

会上的另一个重要议题是：无服务器应用促成了一种围绕事件本身，而非围绕数据来设计的架构。将应用程序订阅到某个事件队列，这是管理服务通信的一种好方法，借此我们可以轻松地为现有队列添加新服务，或修改 / 添加功能，而不需要将数据流直接绑定给应用，进而产生强耦合。

Rob Gruhl 介绍了 Nordstrom 公司正在从事的一个有趣项目：他们正在通过一个集中且统一的事件日志管理零售系统内部的数据流动。应用程序可以通过流生成和使用事件，任何需要对当前状态获得高性能视图的应用程序均可订阅至事件流，借此构建自己的状态数据库（一种数据库视图）。随后即可根据需求为请求提供服务。同时这些应用还可生成供其它服务使用的事件。这样的方式彻底避免了为实现某些状态的中心化存储而使用核心数据库系统的做法，可在改善伸缩性的同时实现微服务之间的解耦。

他们提供了一个名为 Hello Retail 的演示应用，摄影师可以借助该应用为新产品拍照。当新产品加入后，系统会从已知摄影师清单中选择一位摄影师，发送手机短信请求对方拍摄新照片。当摄影师（用短信）回复后，系统会开始处理照片，将其加入图库，随后注册为新产品对应的照片。



Nordstrom 所开发 Hello Retail 应用的架构，详情可参阅 GitHub 上的 Hello Retail 项目页面。

来自 Capital One 的 Srini Uppalapati 介绍了 Capital One 的核心金融系统，他们目前正在将这套系统迁移到云端，并且这一过程中大量使用了无服务器相关技术。他们核心交易系统的迁移主要分为两个阶段：

1. 循序渐进撤销通过大型机系统执行的读取操作：逐渐将大型机系统产生的事件以流的形式发送至云端数据库，并供面向用户的应用和数据科学家使用。
2. 消除消费者应用中的写操作，将核心业务逻辑搬入云端。

目前他们已经完成了第一阶段的任务。在 Srini 所介绍的架构中，他们使用 AWS Lambda 对老数据以及来自大型机的实时事件进行批处理，并将数据装入 S3 以便归档，消费者应用使用了 DynamoDB，分析工作使用了 Redshift。

这个系统在架构和应用层面有一些极为有趣的特点，直接使用“一手”事件是一种对逻辑和状态进行解耦的强大方法：单向数据流使得一切变得更简单。在应用层面上，以 ELM 架构和 React/Redux 为例，通过迁入云端，我们可以将云函数与核心事件流配合使用，打造实用的大规模云应用程序。

企业正在快速接受

上文曾经提到，Nordstrom 和 Capital One 正在通过几个重要项目证明无服务器技术在企业领域的运用前景。最令我吃惊的是，Serverlessconf 大会上还提到了很多其他正在这样做的大企业，他们正在快速接受无服务器技术。

我认为，他们能如此快接受这种技术，原因之一在于很多企业已经开始迁往云端，既然云供应商提供了无服务器相关产品，并且这种技术可以进一步降低成本，他们自然会愿意接受。例如 Capital One 的 Srini 介绍说，通过使用云端无服务器技术，他们大幅节约了成本。目前交易中心每年运营成本约为 9.5 万美元（考虑到客户数高达 4500 万，这样的成本已经相当低了）。

为了满足企业的这类需求，不仅 AWS，目前所有云供应商都在无服务器产品方面进行了巨大的投入。其他云供应商（Google、微软，以及 IBM）也介绍了自家的 FaaS 和无服务器编排产品。

无服务器计算造就的现代化敏捷

Mike Roberts 通过一场精彩的演讲介绍了应用开发者如何借助无服务器技术变得更加以客户为中心，而无须过度关注技术问题本身。现代化敏捷（塑造更出色的人员，持续不断地提供价值，让安全成为先决条件，更快速地尝试和学习）在无服务器技术的帮助下变得更易于实现，开发者再也不需要重新解决已被其他开发者解决了无数遍的相同问题（如何进行身份验证，如何伸缩等），这样他们就可以更专注地为自己的客户提供价值。借此，生产发布再也不需要耗费数天甚至数周时间，几小时就够了。

考虑到：“我们的大部分想法其实都很糟糕 —Jeff Patton。”

我们应当尽可能尝试更多想法。感谢无服务器技术，“试错”成本得以大幅降低！

无服务器技术在这方面已经有很多成功先例，例如 Marcia Villalba 介绍了 Toons.tv 是如何迁移至无服务器技术的。他们在面向云环境重新设计架构后，成本大幅降低，而这一切都是由几名不熟悉无服务器技术的工程

师组成的小团队，在几个月的时间里顺利完成的。Marcia 认为他们的成功主要归功于每周一次的研讨会，大家借此交流讨论新技术的学习心得，并通过测试进行概念验证。

相关工具正在逐渐完善

有一种普遍共识认为，相比云供应商提供的服务，周边工具的发展有些落后了。Florian Motlik 详细介绍了 AWS CLI 工具的不足之处。其他云供应商也面临类似情况，他们往往在无服务器运行时方面投入巨大，但总是会忽视部署、监视和本地测试等过程中必不可少的工具。

基本上，这也意味着用户与云供应商之间的任何交互都必须通过第三方工具进行，例如没人会通过 AWS CLI 进行无服务器部署，大家更愿意通过第三方技术将云供应商生硬的接口抽象为简单易用的应用程序部署框架（例如 Serverless Framework、claudiaJS，以及适合新手的 zappa）。

为了解决这种问题，AWS 发布了 Serverless Application Model (SAM)。SAM 是一种 CloudFormation 之上的抽象层，可大幅简化 Serverless Applications 的创建工作，但 AWS CLI 在这方面依然不够成熟（个人观点）。

很多人还认为，无服务器应用的监视和调试方面也缺乏必要的支持，面对基于事件的架构更是如此（“我的函数无法运行但不知道原因！”以及“为什么不能对运行中的无服务器函数进行实时调试？？”）。

我这并不是在抱怨缺乏交互式调试能力，因为交互式调试实际上可能是一种反面模式 (Anti-pattern)，但如果你想要这样做，微软已经支持通过 Visual Studio 或 Visual Studio Code 对 Azure 云函数进行实时调试（虽然演示过程看着有点不靠谱）。如果想避免交互式调试，那就写单元测试吧。

另外，所有云供应商都开始在监视方面发力。Amazon X-Ray 就是一种非常实用的监视技术，通过与 AWS SDK 集成，可提供有关集成点，即实时架构示意图的实时图表 (Live graph) 和分析能力。如果你在自己的服务调用中使用了 AWS SDK，基本上就可以免费使用该技术。

讨论的另一个重点在于，很多团队依然在探寻无服务器应用开发的最

佳模式。传统上，我们很容易理解特定应用所涉及的相关领域，因为所有内容都位于同一个服务器实例中。你可以启动一个本地实例，准备好所有依赖项，开发过程中在本地执行各种探索式测试。然而对于无服务器开发，通常我们会被紧密绑定至云供应商（服务越“微型”，绑定的程度越高），为了测试应用能否端到端运行，通常还需要对应用程序进行实际的部署（可能会涉及多个云服务组件），甚至要在开发过程中进行部署。很多因素会要求我们必须能够在本地进行探索式测试，但如何对测试中的应用进行隔离，目前并没有任何行之有效的模式，通常到最后我们往往会面对一系列同时在本地和云端运行，“拼凑”出来的测试应用。这方面目前已经有了一些解决方案，例如 Atlassian AWS Local Stack，可以在用户的本地计算机上提供一套功能完备的 AWS 栈。然而为何要用它，而不是直接部署到开发环境，这个问题依然存在争议。

无服务器的编排

无服务器计算还面临一个大问题：难以通过编排大量 Lambda 函数进而在云中创建数据管道。事件流是将 Lambda 函数连接在一起的一种方法，然而通常我们还需要更高级的功能，例如等待条件或并行处理能力。

AWS 和 Azure 都曾演示过自己的无服务器编排技术：**AWS Step Functions**，以及 **Azure Logic Apps**，这两种技术看起来都很有吸引力。

Azure Logic Apps 提供了超过 250 种适用于其他 Azure 产品，以及第三方产品的连接器。虽然华丽的用户界面让我觉得它不太靠谱，但该技术以 JSON 形式的脚本 DSL 支撑，演示效果很出彩，他们将实时推文连接到了微软的情绪分析服务，借此围绕特定话题实时进行了推文情绪分析……所有操作均在 45 分钟的演示内完成（当然很多代码是复制粘贴的）。

我还不太确定该如何以足够可靠的方式应用这些工作流编排服务（如何对其进行测试和部署？），不过感觉上它们会成为无服务器技术不可分割的一部分。

我很想亲自见证这些工作流服务如何进一步完善成为功能完备的平

台。如果有更好的语言（例如 JavaScript 或 Swift）可以编译为“AWS 云计算机语言”，并直接在某种抽象的计算层上运行，又何必使用 JSON DSL 来写脚本。随后可由 AWS 管理所有底层服务器及其状态，用户无需考虑任何有关最长运行时间或最大内存数的限制，只需要严格按照用量来付费。

Serverless 先锋实践

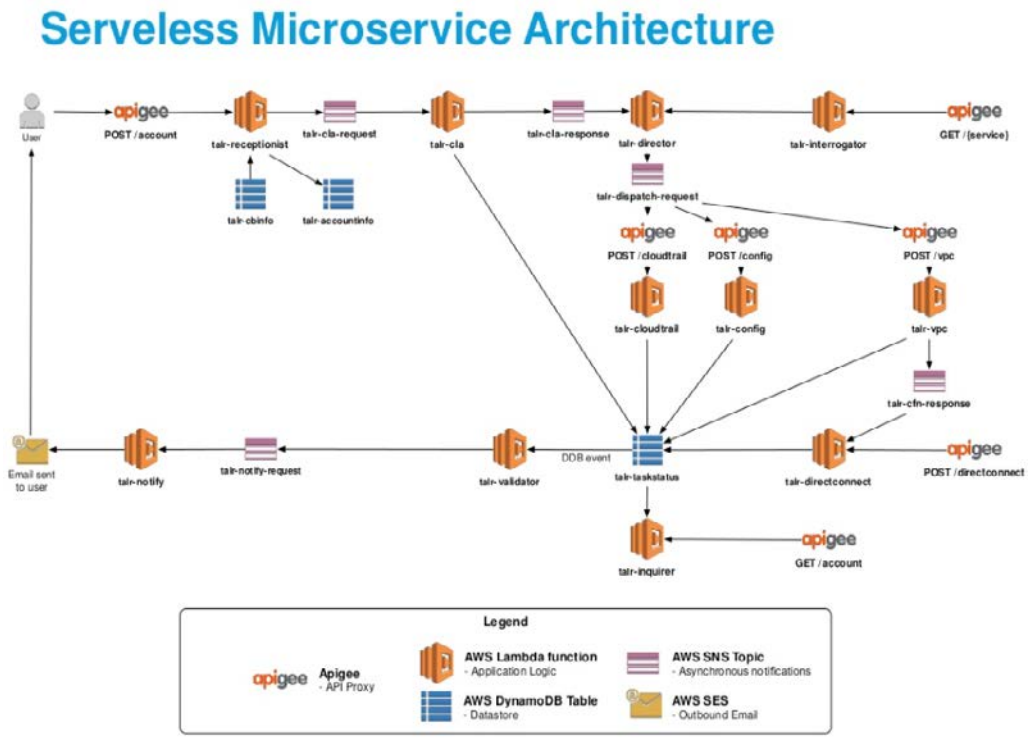
Autodesk 无服务器微服务架构样例

作者 Abel Avram　译者 谢丽

在题为“什么比微服务更好？无服务器微服务”的网络直播中，Alan Williams（Autodesk）、Asha Chakrabarty（Amazon）和 Alan Ho（Apigee）讨论了一个无服务器微服务的架构。其中，该微服务的构建使用了 AWS lambda 函数和运行在 AWS 上的 Apigee 端点。

据 Chakrabarty 介绍，无服务器是一种相对比较新的架构风格，其中的计算单元不是虚拟机，而是一个封装了待执行代码（事件触发）的函数。Williams 指出，无状态计算模型的主要特点是：“代码为主（code focused）”、没有需要管理的服务器、没有需要配置和管理的 EC2 实例、无需人工扩展、没有空闲资源、没有 SSH 或 RDP。

下图简单地描述了一个由 Autodesk 实现的无服务器微服务的架构：



该微服务有多个入口点作为 HTTP 端点（由 Apigee 管理）暴露。用户发起一个 HTTP 调用，并不知道其请求会由一个无服务器微服务提供服务。该服务由多个 Python 编写的 lambda 函数组成，这些函数之间通过 AWS SNS 异步通知进行通信。Lambda 函数是相互独立的，可以使用不同的语言开发，可以由不同的团队维护。

由于 Lambda 不是短期的，所以它们需要将状态在某个地方持久化，其中一个选择是使用 DynamoDB 表。这些表的访问通过 IAM 角色控制，并且仅限于那些需要对它们进行读 / 写访问的函数。这可以避免将不需要的数据暴露给某个 lambda 函数，如果存在安全漏洞的话，这还可以缩小微服务的攻击面。Autodesk 之所以选择使用 DynamoDB 存储状态，是因为它简单，可以将数据作为 JSON 传递，不需要管理一个服务器实例，并且支持自动向上扩展。

上图底部的 DynamoDB 表（talr-taskstatus）将来自多个 lambda 函数的状态持久化，并在表被修改时产生流式事件。这些事件由另一个

lambda 函数监控（talr-validator），它会在必要时采取行动。

对于在 AWS 上实现一个无服务器架构，Williams 列举了如下好处。

- 敏捷性：只需数周就可以实现。
- 不需要管理基础设施，无需EC2或ELB实例，不需要打安全补丁。
- 开发人员只需专注于他们编写的代码。
- 通过无服务器框架管理代码的能力。
- 成本。根据他们的经验，运行lambda解决方案的成本只是传统云解决方案的一小部分（约1%）。由于不需要雇佣运维人员配置和监控EC2和ELB实例，所以成本还会进一步降低。

Williams 还提到，无服务器架构不适合运行长期工作负载或者第三方应用程序。在那些情况下，他认为容器更合适。

本次直播还展示了如何在 AWS 上通过无服务器框架组织代码、部署和运行演示程序。

版权声明

InfoQ 中文站出品

架构师特刊：Serverless 的入门与思考
©2017 北京极客邦科技有限公司

本书版权为北京极客邦科技有限公司所有，未经出版者预先的书面许可，不得以任何方式复制或者抄袭本书的任何部分，本书任何部分不得用于再印刷，存储于可重复使用的系统，或者以任何方式进行电子、机械、复印和录制等形式传播。

本书提到的公司产品或者使用到的商标为产品公司所有。

如果读者要了解具体的商标和注册信息，应该联系相应的公司。

出版：北京极客邦科技有限公司
北京市朝阳区洛娃大厦 C 座 1607
欢迎共同参与 InfoQ 中文站的内容建设工作，包括原创投稿和翻译，请联系 editors@cn.infoq.com。
网 址：www.infoq.com.cn