无服务器云计算综述

黄邵兴[1)]

[1)](华中科技大学, 计算机科学与技术学院, 武汉 中国 085400)

摘 要 云基础设施的发展激发了云原生计算的出现。作为最有前途的微服务部署架构，无服务器计算最近越来越受到业界和学术界的关注。由于其固有的可扩展性和灵活性，无服务器计算对于不断增长的网络服务变得越来越有吸引力和普遍。尽管云原生社区势头强劲，但现有的挑战和妥协仍在等待更先进的研究和解决方案来进一步探索无服务器计算模型的潜力。有学者将架构解耦为四个堆栈层：虚拟化层、封装层、系统编排层和系统协调层。其中，容器作为无服务器计算过程中虚拟化的关键技术, 与其他虚拟化技术：包括 VM，安全容器，unikernel 也将进行对比。由于云原生测试基准的缺乏，也已经有学者做了相关的工作来补充。本文将叙述关于无服务器计算的定义，阐述无服务器计算的动机，优势与挑战。

关键词 无服务器计算，云计算，容器

Overview of Serverless Cloud Computing

Shaoxing Huang[1)]

[1)]((Department of computer science and technology, HuaZhong University of science and technology,Wuhan, China)

Abstract The development of cloud infrastructure has inspired the emergence of cloud-native computing. As the most promising microservice deployment architecture, serverless computing has recently gained increasing attention from industry and academia. Due to its inherent scalability and flexibility, serverless computing is becoming increasingly attractive and commonplace for growing web services. Despite the momentum in the cloud-native community, existing challenges and compromises await more advanced research and solutions to further explore the potential of serverless computing models. Some scholars decouple the architecture into four stack layers: virtualization layer, encapsulation layer, system orchestration layer, and system coordination layer. Among them, containers, as the key technology of virtualization in the process of serverless computing, will also be compared with other virtualization technologies including VM, secure container, and unikernel. Due to the lack of cloud native test benchmarks, scholars have done related work to supplement them. This article will describe the definition of serverless computing, describe the motivations, advantages.

Key words serverless computing; cloud computing; container

1 引言

早在 2009 年, The Berkeley View on Cloud Computing [1] 就确定了云计算的六个潜在优势：

1. 无限计算资源随需应变的出现。

2. 取消云用户的预先承诺。

3. 根据需要支付短期使用计算资源的能力。

4. 由于许多非常大的数据中心，规模经济显着降低了成本。

5、通过资源虚拟化，简化操作，提高利用率。

6. 通过多路复用来自不同组织的工作负载来提高硬件利用率。

在过去十年，这些优势已经基本实现，但云用户承受着复杂操作的负担，许多工作负载仍然无法从高效的多路复用中受益。这些不足主要对应于未能实现最后两个潜在优势。云计算减轻了用户物理基础设施管理的负担，但留给他们对激增的虚拟资源的管理。多路复用适用于批量样式的工作负载，例如 MapReduce 或高性能计算，可以充分利用到它们分配的实例。但它不适用于有状态服务，例如将企业软件（如数据库管理系统）移植到云端时。

2009 年，有两种相互竞争的云虚拟化方法。市场最终接受了亚马逊用于云计算的低级虚拟机方法，因此谷歌、微软和其他云公司提供了类似的界面。低级虚拟机成功的主要原因是在云计算的早期，用户希望在云中重新创建与本地计算机上相同的计算环境，以简化将工作负载移植到云的过程。显然，这种实际需求优先于仅为云编写新程序，尤其是在尚不清楚云的前景的情况下。

这种选择的缺点是开发人员必须自己管理虚拟机，基本上说，要么成为系统管理员，要么与他们一起设置环境。一长串低级虚拟机管理职责激发

了拥有更简单应用程序的客户要求为新应用程序提供更简单的云路径。例如，假设该应用程序希望将图像从手机应用程序发送到云，后者应创建缩略图，然后将它们放置在网络上。完成这些任务的代码可能是几十行 JavaScript，与在适当的环境中设置服务器来运行代码相比，开发量将微不足道的。

认识到这些需求后，亚马逊在 2015 年推出了一项名为 AWS Lambda 服务的新选项。Lambda 提供了云函数，并引起了对无服务器计算的广泛关注。云用户只需编写代码并将所有服务器配置和管理任务留给云提供商。云函数打包为函数即服务（FaaS），代表了无服务器计算的核心，云平台还提供专门的无服务器框架，以满足特定应用程序的需求，如 BaaS（后端即服务）产品，简而言之，无服务器计算 = FaaS + BaaS[2]。一个谬论是“无服务器”可以与 FaaS 互换。准确地说，它们都是无服务器计算必不可少的。Faas 模型实现了函数隔离和调用，而 Baas 提供了对在线服务的整体后端支持。

在 Faas 模型（Lambda 范式）中，应用程序被分割为函数或函数级微服务。函数标识符、语言运行时、一个实例的内存限制和函数代码 blob URI（统一资源标识符）共同定义了一个函数的存在。

Baas 涵盖了广泛的服务，任何应用程序所依赖的服务都可以归入其中。例如，云存储（Amazon S3 和 DynamoDB）、消息传递系统（谷歌云发布/订阅）、消息通知服务（Amazon SNS）和 DevOps 工具（Microsoft Azure DevOps）。

传统的基础设施即服务 (IaaS) 部署模式需要长期运行的服务器来实现可持续的服务交付。但是，无论用户应用程序是否正在运行，这种独占分配都需要保留资源。因此，它导致当前数据中心的资源利用率平均只有 10% 左右，特别是对于具有昼夜模式的在线服务。这种促进了平台化管理的按需开发的服务模式，即无服务器计算，以获得更高的资源利用率和更低的云计算成本。亚马逊、谷歌、微软、IBM、阿里巴巴等大多数大型云厂商都已经提供了这种弹性计算服务。

在任何无服务器平台中，用户只需用高级语言编写云函数，选择应该触发函数运行的事件，例如将图像加载到云存储或将图像缩略图添加到数据库中，并且让无服务器系统处理其他所有事情：实例选择、扩展、部署、容错、监控、日志记录、安全补丁等。表 1 总结了无服务器和传统方法之间的区别，这两种方法代表了基于函数/以服务器为中心的计算平台的连续统一体的端点，容器化编排框架（如 Kubernetes）代表中间件。

	Characteristic	*AWS Serverless Cloud*	*AWS Serverful Cloud*
PROGRAMMER	When the program is run	On event selected by Cloud user	Continuously until explicitly stopped
	Programming Language	JavaScript, Python, Java, Go, C#, etc.[4]	Any
	Program State	Kept in storage (stateless)	Anywhere (stateful or stateless)
	Maximum Memory Size	0.125 - 3 GiB (Cloud user selects)	0.5 - 1952 GiB (Cloud user selects)
	Maximum Local Storage	0.5 GiB	0 - 3600 GiB (Cloud user selects)
	Maximum Run Time	900 seconds	None
	Minimum Accounting Unit	0.1 seconds	60 seconds
	Price per Accounting Unit	$0.0000002 (assuming 0.125 GiB)	$0.0000867 - $0.4080000
	Operating System & Libraries	Cloud provider selects[5]	Cloud user selects
SYSADMIN	Server Instance	Cloud provider selects	Cloud user selects
	Scaling[6]	Cloud provider responsible	Cloud user responsible
	Deployment	Cloud provider responsible	Cloud user responsible
	Fault Tolerance	Cloud provider responsible	Cloud user responsible
	Monitoring	Cloud provider responsible	Cloud user responsible
	Logging	Cloud provider responsible	Cloud user responsible

表 1

在云环境中，服务器式计算就像用低级汇编语言编程，而无服务器计算就像用 Python 等高级语言编程。计算简单表达式（例如 c = a + b）的汇编语言，程序员必须选择一个或多个要使用的寄存器，将值加载到这些寄存器中，执行算术运算，然后存储结果。这反映了服务器式云编程的几个步骤，首先提供资源或识别可用资源，然后使用必要的代码和数据加载这些资源，执行计算，返回或存储结果，并最终管理资源释放。无服务器计算的目标和机会是为云程序员提供类似于向高级编程语言过渡的遍历。高级语言编程环境的其他特性在无服务器计算中也有自然的相似之处。自动化内存管理使程序员无需管理内存资源，而无服务器计算则使程序员无需管理服务器资源。

准确地说，无服务器和有服务器计算之间存在三个关键区别：

1. 计算和存储解耦。存储和计算单独扩展，独立配置和定价。通常，存储由单独的云服务提供，计算是无状态的。

2. 执行代码而不管理资源分配。用户提供一段代码，而不是请求资源，云会自动提供资源来执行该代码。

3. 按使用资源的比例付费，而不是按分配的资源付费。计费是根据与执行相关的某个维度（例如执行时间），而不是根据基础云平台的维度（例如分配的 VM 的大小和数量）进行的。

2 原理与优势

2.1 原理

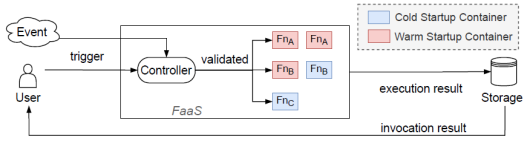

图 1

为了描述无服务器计算模型，我们以图 1 中的异步调用为例。无服务器系统从用户那里接收触发的 API 查询，验证它们，并通过创建新的沙箱（也称为冷启动）或重用运行的热启动（也称为热启动）来调用函数。隔离确保每个函数调用在单个容器或从访问控制控制器分配的虚拟机中运行。由于事件驱动和单事件处理的特性，可以触发无服务器系统以提供按需隔离的实例，并根据实际应用程序工作负载水平扩展它们。之后，每个执行工作者访问后端数据库以保存执行结果。通过进一步配置触发器和桥接交互，用户可以自定义复杂应用程序的执行（例如，在 $\{F_{n_A}, F_{n_B}, F_{n_C}\}$ 管道中构建内部事件调用）。

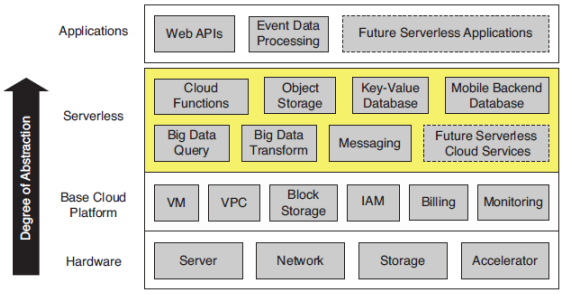

图 2

无服务器云的架构如图 2。无服务器层位于应用程序和基础云平台之间，简化了云编程。云功能（即 FaaS）提供通用计算，并辅以专门的后端即服务 (BaaS) 产品生态系统，例如对象存储、数据库或消息传递。具体来说,AWS 上的无服务器应用程序可能使用 Lambda 和对象存储和 DynamoDB（键值数据库），而谷歌云上的应用程序可能使用 Cloud Functions 和 Cloud Firestore（移动后端数据库和 Cloud Pub/Sub（消息传递）。无服务器计算还包括某些大数据服务，例如 AWS Athena 和 Google BigQuery（大数据查询），以及 Google Cloud Dataflow 和 AWS Glu（大数据转换）。基础底层基础云平台包括虚拟机（VM）、私有网络（VPC）、虚拟化块存储、身份和访问管理（IAM）以及计费和监控。

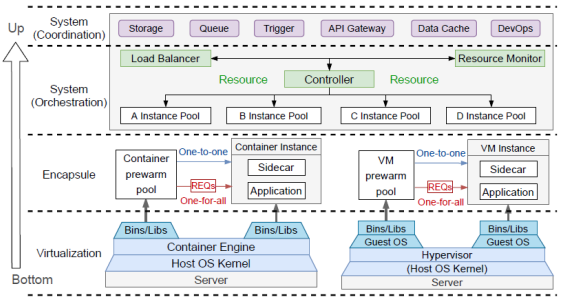

图 3

在 The Serverless Computing Survey: A Technical Primer for Design Architecture[3] 中，作者用自底向上的逻辑分析其设计架构，将无服务器计算架构解耦为四个堆栈层：虚拟化、封装、系统编排和系统协调。

虚拟化层：虚拟化层在性能和功能安全的沙箱中实现函数隔离。沙箱为应用服务代码运行时提供运行时环境、依赖项和系统库。为了防止多应用或多租户场景下的资源访问，云厂商通常采用容器/虚拟机来实现隔离。目前，流行的沙盒技术有 Docker、gVi-sor 、Kata 、Firecracker 和 Unikernel。

封装层：封装层中的各种中间件可以实现自定义的函数触发器和执行，并且它们还提供用于通信和监控的数据度量收集。这些额外的中间件称为 sidecar。它分离了服务的业务逻辑，并实现了函数和底层平台之间的松散耦合。同时，为了加快实例启动和初始化，封装层通常使用预热池。此外无服务器系统可以通过分析负载模式来使用预测，以一对一的方式预热每个函数，或者为所有函数构建一个模板，以通过一对一的方法根据运行时特性动态安装需求 (REQ)。

系统编排层：系统编排层允许用户配置触发器和绑定规则，通过随着负载变化动态调整来保证用户应用的高可用性和稳定性。通过云编排，线上线下调度相结合，可以避免资源争用，回收闲置资源，缓解共置功能的性能下降。上述实现通常也集成到容器编排服务中（例如，Google Kubernetes 和 Docker Swarm）。而在无服务器系统中，资源监视器、控制器和负载均衡器被整合以解决调度挑战。它们使无服务器系统能够在三个不同级别上实现调度优化：分别是资源级、实例级和应用程序级。

系统协调层：系统协调层由一系列后端即服务 (BaaS) 组件组成，这些组件使用统一的 API 和 SDK

将后端服务集成到函数中。显然，它不同于使用云之外的本地物理服务的传统中间件。这些 BaaS 服务提供存储、队列服务、触发器绑定、API 网关、数据缓存、DevOps 工具等定制组件以更好地满足系统编排层的灵活性要求。

2.2 优势

对于云提供商而言，无服务器计算可促进业务增长，因为使云更易于编程有助于吸引新客户并帮助现有客户更多地使用云产品。例如，最近的调查发现，大约 24% 的无服务器用户是云计算的新手，30% 的现有服务器云客户也使用无服务器计算 [2]。此外，短运行时间、小内存占用和无状态特性通过使云提供商更容易找到运行这些任务的未使用资源来改善统计多路复用。云提供商还可以使用不太流行的计算机，因为实例类型取决于云提供商，例如可能对多服务器云客户没有吸引力的旧服务器。这两个好处都增加了现有资源的收入。

客户受益于提高的编程效率，并且在许多情况下还可以节省成本，这是底层服务器利用率更高的结果。即使无服务器计算让客户更加高效，Jevons 悖论表明他们将增加对云的使用，而不是减少使用，因为更高的效率将通过增加用户来增加需求。

无服务器还将云部署级别从 x86 机器代码（99% 的云计算机使用 x86 指令集）提升到高级编程语言，从而实现架构创新。如果 ARM 或 RISC-V 提供比 x86 更好的性能，那么无服务器计算可以更轻松地更改指令集。云提供商甚至可以开展面向语言的优化和特定领域架构的研究，专门旨在加速用 Python 等语言编写的程序。

无服务器计算对云用户非常友好，因为新手可以在不了解云基础架构的情况下部署功能，因为专家可以节省开发时间并专注于其应用程序特有的问题。无服务器用户可以节省资金，因为这些函数仅在事件发生时执行，而细粒度计费（今天通常为 100 毫秒）意味着他们只需为使用的内容付费，而不是为保留的内容付费。

研究人员一直被无服务器计算所吸引，尤其是云函数，因为它是一种新的通用计算抽象，有望成为云计算的未来，并且有很多机会可以提高当前的性能并克服其当前的局限性。

3 研究进展

3.1 无服务器计算的特征

根据一些学者的观点，无服务器计算应有以下特征 [1-4]：

自动缩放: 自动可扩展性不应只限于 FaaS 模型（例如，容器黑匣子作为 OpenWhisk 中的调度单元）。识别无服务器系统不可或缺的因素是在适应工作负载动态时执行水平和垂直扩展。允许应用程序将实例数量缩放到零也引入了一个挑战——冷启动。当一个函数遇到冷启动时，实例需要从头开始，初始化软件环境，并加载特定于应用程序的代码。这些步骤会明显拖累服务响应，导致影响 QoS（服务质量）。

灵活的调度: 由于应用程序不再绑定到特定的服务器，无服务器控制器根据集群中的资源使用情况动态调度应用程序，同时确保负载平衡和性能保证。此外，无服务器平台还考虑了多区域协作。对于更健壮和可用的无服务器系统，灵活的调度允许工作负载查询分布在更广泛的区域。它避免了在节点不可用或崩溃的情况下严重的性能下降或对服务连续性的破坏。

事件驱动: 无服务器应用程序由事件触发，例如 REST-ful HTTP 查询的到达、消息队列的更新或存储服务的新数据。通过使用触发器和规则将事件绑定到函数，控制器和函数可以使用封装在上下文属性中的元数据。它使事件和系统之间的关系变得可检测，从而实现对不同事件的不同协作响应。云原生计算基金会 (CNCF) 无服务器小组还发布了 CloudEvents 规范，用于通常描述事件元数据以提供互操作性。

透明发展：一方面，管理底层主机资源将不再是应用维护者的烦恼。这是因为他们不知道执行环境。同时，云供应商应确保可用的物理节点、隔离的沙箱、软件运行时和计算能力，同时使它们对维护人员透明。另一方面，无服务器计算还应该集成 DevOps 工具，以帮助实现更有效地部署和迭代。

现收现付: 无服务器计费模型将计算能力的成本从资本支出转变为运营支出。这种模式消除了用户根据峰值负载购买专属服务器的需求。即用即付模型通过共享网络、磁盘、CPU、内存等资源，只指示应用实际使用的资源，无论实例是运行还是空闲。

3.2 实现技术与遇到的困难

下面将阐述上面提到的虚拟化层、封装层、系统编排层和系统协调层的实现技术和遇到的问题。

虚拟化层：每当在无服务器计算中调用用户函数时，它都会在虚拟化沙箱中加载和执行。一个函数可以重用一个沙箱或创建一个新的沙箱，但通常不会与不同的用户函数共同运行。在此前提下，虚拟化中的大多数问题是隔离性、灵活性和低启动延迟。这种隔离保证了每个应用进程都在划定的资源空间中运行，运行中的进程可以避免受到其他进程的干扰。测试和调试能力以及对系统扩展的额外支持证明了灵活性。低启动延迟需要对沙箱创建和初始化的快速响应。目前虚拟化层的沙箱机制分为四大类：传统 VM（虚拟机）、容器、安全容器和 Unikernel。

Virtualization	Startup latency	Isolation level	OSkernel	Hotplug	Hypervisor	OCI supported	Backed by
Traditional VM	>1000ms	Strong	unsharing		✓	✓	/
Docker [41]	50ms-500ms	Weak	host-sharing	✓		✓	Docker
SOCK [101]	10ms-50ms	Weak	host-sharing	✓		✓	/
Hyper-V [58]	>1000ms	Strong	unsharing	✓	✓	✓	Microstft
gVisor [49]	50ms-500ms	Strong	unsharing		✓	✓	Google
Kata [67]	50ms-500ms	Strong	unsharing	✓	✓	✓	OpenStack
FireCracker [3]	50ms-500ms	Strong	unsharing		✓	✓	Amazon
Unikernel [86]	10ms-50ms	Strong	Built-in		✓		Docker

表 2

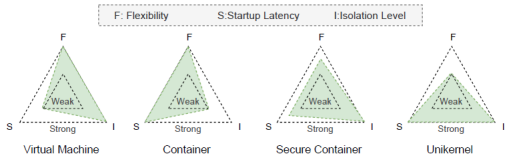

图 4

表 2 在几个方面比较了这些主流方法。表中“启动延迟”表示冷启动的响应延迟。“隔离级别”表示函数在不受他人干扰的情况下运行的能力。“OSkernel”显示 GuestOS 中的内核是否共享。“热插拔”允许函数实例以最少的资源（CPU、内存、virtio 块）启动，并在运行时添加额外的资源。“OCI 支持”是指它是否提供开放容器倡议 (OCI)，一种用于表达容器格式和运行时的开放治理结构。

传统的基于 VM 的隔离采用 VMM（virtual machine manager，e.g.，hypervisor），为 GuestOS 提供虚拟化能力。它还可以通过提供的接口（或使用 Qemu/KVM）调解对所有共享资源的访问。借助快照，当对每个 VM 实例中的应用程序执行补丁时，VM 在快速故障保护方面表现出高度的灵活性。虽然 VM 提供了强大的隔离机制和灵活性，但它缺乏用户应用程序启动延迟较低（通常 >1000 毫秒）的好处。这种权衡是无服务器计算的基础，其中函数很小，而 VMM 和客户内核的相对开销很高。图 4 对 4 种技术的灵活性，隔离级，启动延迟作了对比。

Unikernel 对应用程序天生缺乏灵活性，更不用说糟糕的 DevOps 环境了。此外，在异构集群中，底层硬件的异构性迫使 Unikernel 随着驱动程序的变化而更新，使其与无服务器计算特性的要求不相符。因此，将分析最主要的两个虚拟化技术：VM 和容器。

根据 Docker 官方网站给出的定义 [6]：容器是打包代码及其所有依赖项的标准软件单元，因此应用程序从一个计算环境快速可靠地运行到另一个计算环境。Docker 容器映像是一个轻量级、独立、可执行的软件包，其中包含运行应用程序所需的一切：代码、运行时、系统工具、系统库和设置。

云使用虚拟化技术来实现大规模共享资源的弹性 [3]。VM 通常是基础架构层的主干。相比之下，容器化允许轻量级虚拟化，通过定制构建容器作为来自单个镜像（通常从镜像存储库中检索）的应用程序包，消耗更少的资源和时间。它们还支持云中可移植、可互操作的软件应用程序所需的更具互操作性的应用程序包。容器化基于开发、测试和部署应用程序到大量服务器以及互连这些容器的能力。因此，容器解决了云 PaaS 级别的问题。

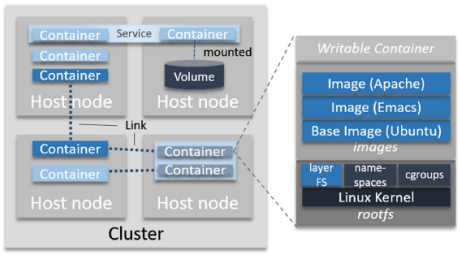

图 5

许多容器解决方案都基 Linux LXC 技术。最近的 Linux 发行版——Linux 容器项目 LXC 的一部分——提供内核机制，如命名空间和 cgroups，以隔离共享操作系统上的进程。Docker 是目前最流行的容器解决方案。Docker 镜像由相互分层的文件系统组成，类似于 Linux 虚拟化堆栈，使用 LXC 机制，见图 5 右侧。Docker 使用联合挂载在只读文件系统之上添加可写文件系统。这允许多个只读文件系统相互堆叠。此属性可用于通过在基本镜像之上构建来创建新镜像。只有顶层是可写的，也就是容器本身。

容器化促进了从容器中的单个应用程序到可

以跨集群主机运行容器化应用程序的容器主机集群。后者受益于容器的内置互操作性。单个容器主机被分组为相互连接的集群，如图 5 左侧所示。每个集群由几个（主机）节点组成。应用程序服务是来自同一镜像的逻辑容器组。应用程序服务允许跨不同的主机节点扩展应用程序。卷是用于需要数据持久性的应用程序的机制。容器可以挂载这些卷进行存储。链接允许两个或多个容器连接和通信。这些容器集群的设置和管理需要对容器间通信、链接和服务组件的编排支持。

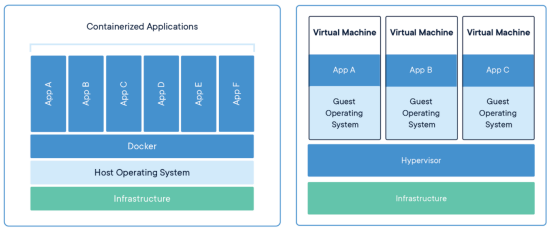

图 6

容器和 VM 具有类似的资源隔离和分配优势，但功能不同，因为容器虚拟化操作系统而不是硬件。容器更便携、更高效，其区别见图 6[6]。一般将容器与 VM 混合使用，容器面向部署频繁变动，生命周期短的服务，而 VM 面向相对长期，比如在一周或一个月内在服务器上部署。在具体的部署时，也需要对图 4 中的灵活性，启动时间，隔离级以及安全性进行权衡。

封装层：无服务器计算中的冷启动可能发生在函数无法捕获热运行的容器或遇到突发负载时。在前者中，函数被第一次调用，或者以比实例生命周期更长的调用间隔进行调度。典型的特征是实例（或 Pod）必须从头开始。在后一种突发负载情况下，实例需要在用户工作负载激增期间执行水平扩展。函数实例将随着负载变化自动缩放以确保足够的资源分配。包括用不到一秒的时间在虚拟化层准备一个沙箱，软件环境的初始化，例如加载 Python 库和特定于应用程序的用户代码。虽然可以提供更轻量级的沙箱机制来减少虚拟化层的冷启动延迟，但是当迁移到现有的无服务器架构时，最先进的沙箱机制可能无法证明对容器或 VM 的完美兼容性。针对性能和兼容性之间的权衡，一个有效的解决方案是在封装层预热实例。这种方法被称为预热启动，已被广泛研究。

有两种常见的预热启动方式：一对一预热启动和一对所有预热启动。在一对一的预热启动中，每个函数实例都是从一个固定大小的池中预热，或者根据历史工作负载轨迹进行动态预测。而在一对所有的预热启动中，所有功能的实例都是从缓存沙箱中预热的，这些沙箱是根据一个通用配置文件预先生成的。当发生冷启动时，该函数只需要通过导入特定于函数的代码 blob URI 和设置来专门化这些预初始化的沙箱。为了获得更高的可扩展性和更低的实例初始化延迟，C/R（检查点/恢复）还与无服务器系统中的预热实例相结合。C/R 是一种技术，可以冻结一个正在运行的实例，在一个文件列表中创建一个检查点，然后在冻结点恢复实例的运行状态。无服务器实现中的一个常见模式是在空闲时暂停实例以节省资源，然后在调用时恢复它以供重用。

对于一对一的预热启动和一对所有的预热启动，两者都可以有利于在无服务器架构的封装层中优化冷启动。它们各自的缺陷也很明显。一对一预热启动侧重于通过交换内存资源来显着减少初始化延迟。它在确保内存资源的合理分配的同时，带来了一个通常难以衡量或预测的预热时间的挑战。一方面，当历史数据足以构建准确的模型时，基于预测和基于启发式的方法特别有效，但在缺乏跟踪分析时效果不佳。另一方面，当大量应用程序和函数链共存时，预测和迭代操作会引入高 CPU 开销。

采用一对所有预热启动模板缓解了函数从头开始冷启动的高成本。此外，与一对一预热启动相比，维护全局预热池引入的额外内存资源消耗更少。然而，它仍然面临一些挑战，包括巨大的模板镜像大小和各种预导入库的冲突。它还可能会揭示具有相似特征的应用程序被广泛部署的区域。对于不同场景下的冷启动，“因地制宜”非常重要。例如，在第一次调用该函数时，或者在跟踪分析期间预测不佳时，通过一对多的预热启动生成模板的效率要高得多。一对一预热启动对于具有通用规则或昼夜模式的功能表现更好，反之亦然。

系统编排层: 如何单独和集群协调容器的构建和部署已成为一个中心问题 [4]。系统编排层的主要挑战是对不同服务的友好和弹性支持。尽管当前的无服务器编排器的实现方式不同，但它们在系统中遇到的挑战却大同小异。由于数百个函数共存于一个无服务器节点上，这对调度具有不可分割依赖关系的大量函数提出了挑战。与传统解决方案类似，无服务器模型也关注预测按需计算资源的能力，以及高效的服务调度策略。如图 7 所示，研究人员通

常建议在控制器中引入负载均衡器和资源监视器组件，以解决供应和调度挑战。负载均衡器旨在协调资源使用以避免任何单个资源过载。同时，资源监控器持续关注各个节点的资源利用率，并将更新的信息传递给负载均衡器。通过资源监控器和负载均衡器，无服务器控制器可以在三个方面执行更好的调度策略：资源级、实例级和应用程序级。

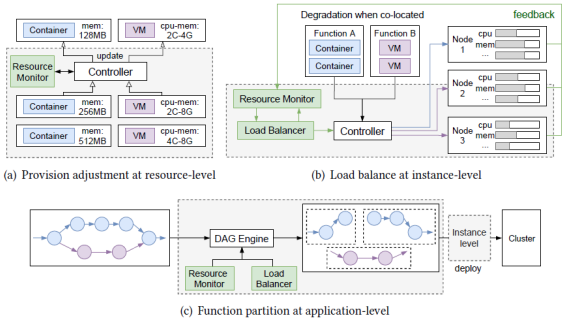

图 7

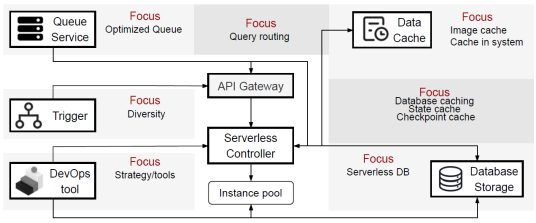

图 8

系统协调层; 系统协调层中的一些其他组件来支持或增强无服务器系统。如图 8 所示。在实现方面，无服务器系统需要集成六个重要组件或服务：存储、队列、API 网关、触发器、数据缓存和 DevOps 工具。

3.3 无服务器计算遇到的挑战

除了在各层实现中遇到的困难，无服务器计算还有其他几个方面的挑战：

1. 封装层内的无状态

无服务器计算的一个基本特性是服务按需加载和执行，而不是部署在长期运行的实例中。为了防止大量实例占用内存资源，无服务器计算控制器设置了实例生命周期来自动回收它们。应用程序的状态不能也不会保留在恢复的实例上。无状态特性削弱了无服务器架构的通用性，将其范围限制在无状态应用程序，例如 Web 应用程序、IoT（物联网）、媒体处理等。毫无疑问，有状态无服务器架构的扩展，如通过在对象存储或键值存储中保存状态无法同时提供低延迟和高吞吐量，使其不如 IaaS 或 PaaS 的常规持久型会话。

2. 编排层内存碎片化

在多个租户共存的无服务器架构中，并发调用要么在多个容器中处理并在每个容器中经历意外的冷启动，要么在一个容器中并发执行（例如 OpenFaaS 和 OpenLambda）。在前者中，为了性能隔离，一个容器一次只能执行一次调用。在这种情况下，海量 sidecar 的内存占用阻碍了无服务器容器实现高密度部署和提高资源利用率。这一挑战的关键是通过 VMM 和客户内核中的重复数据删除来精简和压缩容器运行时，例如在主机上的不同实例之间共享页面缓存。在后者中，内存碎片成为重中之重。分配碎片通常是由于 microVM 的不当提供造成的。函数执行器不能充分利用分配的内存。调度碎片是不可避免的，通常是在随着工作负载变化自动扩展时由实例级负载平衡策略引起的。自从无服务器计算出现以来，在指导如何搜索高密度容器部署解决方案方面，进一步实现一种有效的方法仍然存在挑战。

3. 协调层内的 API 和基准锁定

当人们谈论无服务器供应商锁定时，他们关心的是功能的可移植性。然而，这个问题的真正点取决于来自其他服务的 API 而不是函数本身。尽管 Apex 和 Sparta 等一些努力允许用户使用本机不支持的语言将功能部署到无服务器平台，但来自不同平台 BaaS 服务及其 API 定义仍然不同。API 锁定的挑战源于用户功能与其他 BaaS 组件之间的紧密耦合，这会增加不同 FaaS 平台之间的代码迁移难度。

过度简化的基准测试是 API 锁定的另一个问题。易于构建的微基准被过分强调并在当前工作的 75% 中使用。除了科学的工作流，我们还呼吁建立和开源跨平台的现实应用程序基准。然而，当将一个大型服务分解为不同的函数，然后构建细粒度的节点互连时，应用架构的复杂性使得功能的分级难以指导和确定。

不过针对基准测试这一问题，在 2021 年的 FAST 会议上，有学者提出了一种更接近现实负载的基础测试应用程序 CNSbench [5]。以往的基准测试程序忽略了控制操作，CNSbench 填补了这一空缺，并且它允许将传统存储基准测试工作负载与用户定义的控制操作工作负载轻松结合。由于 CNSBench 本身是一个云原生应用程序，因此它本身就支持大规模不同控制和 I/O 工作负载组合的编

排。作者为 Kubernetes 构建了 CNSBench 的原型，利用几个现有的容器化存储基准来生成数据和元数据 I/O。通过 Ceph 和 OpenEBS（Kubernetes 的两个流行存储提供商）的案例研究证明了 CNSBench 的有用性。

4 总结与展望

云原生概念的快速发展激发了开发人员将云应用重组为微服务。弹性无服务器计算成为这些微服务的最佳方案。通过提供简化的编程环境，无服务器计算使云更易于使用，从而吸引更多能够并且将会使用它的人。无服务器计算包括 FaaS 和 BaaS 产品，标志着云编程的重要成熟。它消除了当今服务器计算强加给应用程序开发人员的手动资源管理和优化的需要，这种成熟类似于四十多年前从汇编语言到高级语言的转变。

无服务器计算的使用将越来越广泛。混合云本地应用程序将随着时间的推移而减少。无服务器计算仍处于起步阶段，未来几年潜力仍然存在。

无服务器计算的未来面临的两个挑战是提高安全性和适应可能来自专用处理器的成本性能优化。在这两种情况下，无服务器计算都具有可能有助于解决这些挑战的特性。物理共驻是旁道攻击的要求，但在无服务器计算中确认要困难得多，并且可以轻松地采取步骤来随机化云功能放置。使用 JavaScript、Python 或 TensorFlow 等高级语言对云函数进行编程，提高了编程抽象级别，并使其更容易创新，从而使底层硬件能够提供更高的性价比。

参考文献

[1] ARMBRUST M, FOX A, GRIFFITH R, et al. Above the clouds: A berkeley view of cloud computing[R]. [S.l.]: Technical Report UCB/EECS-2009-28, EECS Department, University of California ⋯, 2009.

[2] JONAS E, SCHLEIER-SMITH J, SREEKANTI V, et al. Cloud programming simplified: A berkeley view on serverless computing[J]. arXiv preprint arXiv:1902.03383, 2019.

[3] LI Z, GUO L, CHENG J, et al. The serverless computing survey: A technical primer for design architecture[J]. arXiv preprint arXiv:2112.12921, 2021.

[4] PAHL C, BROGI A, SOLDANI J, et al. Cloud container technologies: a state-of-the-art review[J]. IEEE Transactions on Cloud Computing, 2017, 7(3):677-692.

[5] MERENSTEIN A, TARASOV V, ANWAR A, et al. Cnsbench: A cloud native storage benchmark[C]//19th {USENIX} Conference on File and Storage Technologies ({FAST} 21). [S.l.: s.n.], 2021: 263-276.

[6] [C/OL]//https://www.docker.com/resources/what-container.