欢迎学习模块 13：构建微服务和无服务器架构。

模块概览

章节

1. 架构需求
2. 介绍微服务
3. 使用 AWS 容器服务构建微服务应用程序
4. 介绍无服务器架构
5. 使用 AWS Lambda 构建无服务器架构
6. 使用 Amazon API Gateway 扩展无服务器架构
7. 使用 AWS Step Functions 编排微服务

演示

- 创建 AWS Lambda 函数
- 结合使用 AWS Lambda 和 Amazon S3

实验

- （可选）指导实验 1：将整体式 Node.js 应用程序拆分为微服务
- 指导实验 2：在 AWS 上实施无服务器架构
- 挑战实验：为咖啡馆实施无服务器架构

知识考核

本模块包括以下章节：
架构需求
1.介绍微服务
2.使用 AWS 容器服务构建微服务应用程序
3.介绍无服务器架构
4.使用 AWS Lambda 构建无服务器架构
5.使用 Amazon API Gateway 扩展无服务器架构
6.使用 AWS Step Functions 编排微服务

本模块还包括：

- 两个 AWS Lambda 演示
- 一个可选的指导实验，您可以将整体式应用程序重构为微服务
- 一个指导实验，您可以使用 Amazon S3、AWS Lambda、Amazon DynamoDB 和 Amazon SNS 在 AWS 上实施无服务器架构
- 一个挑战实验，您可以使用 AWS Lambda 和 Amazon Simple Notification Service (Amazon SNS) 生成并发送咖啡馆的每日销售报告。

最后，您需要完成一个知识考核，以测试您对本模块中涵盖的关键概念的理解程度。

模块目标

学完本模块后，您应该能够：

- 指出微服务的特性
- 将整体式应用程序重构为微服务，然后使用 Amazon ECS 部署容器化微服务
- 解释无服务器架构的含义
- 使用 AWS Lambda 实施无服务器架构
- 描述 Amazon API Gateway 的通用架构
- 描述 AWS Step Functions 支持的工作流类型

学完本模块后，您应该能够：
- 指出微服务的特性
- 将整体式应用程序重构为微服务，然后使用 Amazon ECS 部署容器化微服务
- 解释无服务器架构的含义
- 使用 AWS Lambda 实施无服务器架构
- 描述 Amazon API Gateway 的通用架构
- 描述 AWS Step Functions 支持的工作流类型


第 1 节：架构需求
模块 13：构建微服务和无服务器架构
aws
© 2023, Amazon Web Services, Inc. 或其联属公司。保留所有权利。

介绍第 1 节：架构需求。

咖啡馆业务需求

咖啡馆希望通过电子邮件获得有关网站上所有订单的每日报告。他们想要获取这些信息，以便能够预测需求，然后烘焙正确数量的甜点（减少浪费）。他们还希望识别业务中的任何模式（分析）。

Frank 和 Martha 希望通过电子邮件获得有关网站上所有订单的每日报告。Frank 希望能够预测需求，以便烘焙正确数量的甜点（减少浪费）。Martha 希望识别咖啡馆业务中的任何模式（分析）。目前，Sofía 已经在 Web 服务器实例上设置了 cron 作业，将这些每日订单报告电子邮件消息发送给 Frank 和 Martha。但是，cron 作业为资源密集型作业，会降低 Web 服务器的性能。

Olivia 建议 Sofía 和 Nikhil 将非业务关键型报告任务分离开。Sofía 和 Nikhil 希望进一步解耦架构，并将 cron 作业移动到能够很好地扩展且能够降低成本的托管式无服务器环境中。


第 2 节：介绍微服务
模块 13：构建微服务和无服务器架构
aws
© 2023, Amazon Web Services, Inc. 或其联属公司。保留所有权利。

介绍第 2 节：介绍微服务。

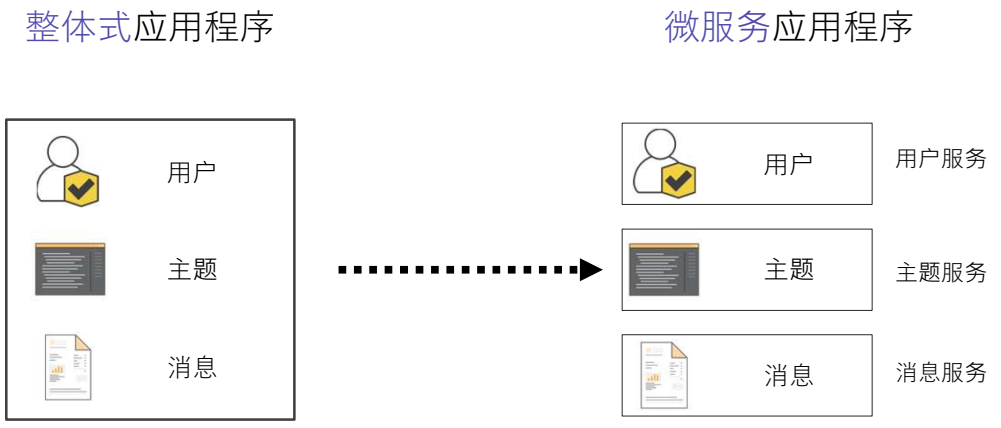
什么是微服务？

应用程序由独立的服务组成，这些服务通过明确定义的 API 通信

微服务是一种开发软件的架构和组织方法，采用这种方法开发出的应用程序由独立的服务组成，这些服务通过明确定义的 Application Programming Interface (API) 进行通信。这种方法旨在加快部署周期。

微服务方法促进了创新并明确了所有权，还提高了软件应用程序的可维护性和可扩展性。

整体式应用程序与微服务应用程序的对比



要了解微服务的益处，首先要了解整体式应用程序。

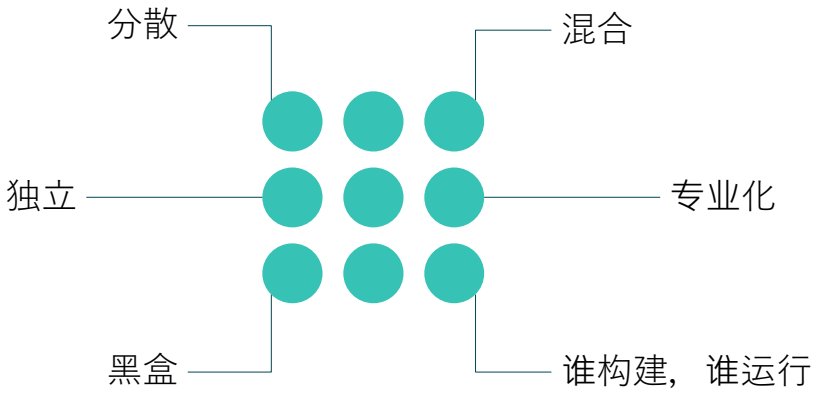
在左侧的示例中，整体式论坛应用程序的三个进程（用户、主题和消息）紧耦合。它们一起作为单一服务运行。如果应用程序的一个进程遇到需求峰值，则必须扩展整个架构。随着代码库的增长，添加或改进功能变得更加复杂，这不仅限制了试验，还让实现新创意变得困难。整体式应用程序的可用性也会有风险，因为许多相互依赖、紧耦合的进程会放大单个进程故障带来的影响。

现在，假设同一个应用程序在微服务架构中运行。应用程序的每个进程都构建为独立的组件，作为服务运行。服务通过使用轻量级 API 操作进行通信。每个服务执行一项可支持多个应用程序的功能。由于服务独立运行，因此可以更新、部署和扩展这些服务，以满足应用程序特定功能的需求。

微服务架构提供更快的迭代、自动化和整体敏捷性。支持快速启动、快速失败和快速恢复。

有关 AWS 上微服务的概览，请参阅什么是微服务？

微服务的特性



微服务具有一些共同的特性：

- 分散 – 微服务架构是采用分散式数据管理的分布式系统。它们不依赖中央数据库中的统一架构。每个微服务都有自己对数据模型的看法。微服务的开发、部署、管理和运行方式也是分散式的。
- 独立 – 微服务架构中的每个组件服务都可以独立进行更改、升级或替换，而不会影响其他服务的功能。这些服务不需要与其他服务共享任何代码或实施。同样，负责不同微服务的团队可以彼此独立行事。
- 专业化 – 每项组件服务都专为实现一组功能而设计，并专注于特定领域。如果特定组件服务的代码达到一定的复杂程度，则该服务可以拆分为两个或更多服务。
- 混合 – 微服务并不遵循单一的方法。团队可以自由选择最佳工具来解决他们的具体问题。因此，微服务架构在操作系统、编程语言、数据存储和工具方面采用了异构方法。这种方法被称为混合持久性和编程。
- 黑盒 – 单个组件服务被设计为黑盒，这意味着其复杂性的细节对其他组件隐藏不见。服务之间的任何通信都通过明确定义的 API 进行，以防止隐式和隐藏的依赖项。
- 谁构建，谁运行 – DevOps 是微服务的关键组织原则，负责构建服务的团队还负责在生产中运行和维护服务。

第 2 节要点

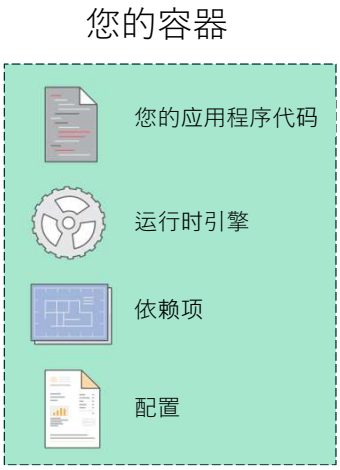
- 微服务应用程序由独立的服务组成，这些服务通过明确定义的 API 通信
- 微服务具有以下特性 –
  - 分散
  - 独立
  - 专业化
  - 混合
  - 黑盒
  - 谁构建，谁运行

本模块中这节内容的要点包括：

- 微服务应用程序由独立的服务组成，这些服务通过明确定义的 API 通信
- 微服务具有以下特性 –
  - 分散：微服务的开发、部署、管理和运行方式是分散式的
  - 独立：微服务架构中的每个组件服务都可以在不影响其他服务功能的情况下进行开发、部署、运行和扩展
  - 专业化：每项组件服务都专为实现一组功能而设计，侧重于解决特定问题
  - 混合：微服务架构在操作系统、编程语言、数据存储和工具方面采用异构方法
  - 黑盒：微服务组件复杂性的细节对其他组件隐藏不见
  - 谁构建，谁运行：DevOps 是微服务的关键组织原则

介绍第 3 节：使用 AWS 容器服务构建微服务应用程序。

什么是容器？



构建微服务架构时，可以使用容器来提供处理能力。

*容器*是实现操作系统虚拟化的一种途径，让您可以在资源受到隔离的进程中运行应用程序及其依赖项。容器是一个轻量级的独立软件包。它包含软件应用程序运行所需的一切，例如应用程序代码、运行时引擎、系统工具、系统库和配置。

容器解决的问题

让软件在不同工作环境中可靠运行

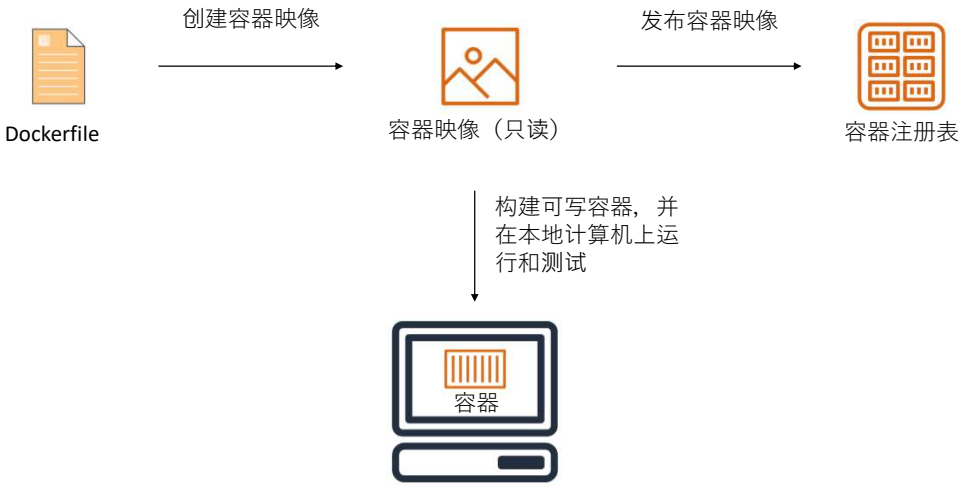
开发人员的工作站

生产环境

测试环境

容器有助于确保快速、可靠且一致地部署应用程序，不受部署环境的影响。容器还可以让您更精细地控制资源，提高您的基础设施效率。

容器术语



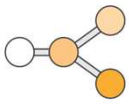
容器是通过一种称为*映像*的只读模板创建的。映像通常是利用 Dockerfile 构建出来的，后者是一个纯文本文件，指定容器中包含的所有组件。您既可以从头开始创建映像，也可以使用其他人创建并发布到公有或私有容器注册表的映像。

容器映像是容器可用的文件系统的快照。例如，您可能使用 Debian 操作系统作为容器映像。当您运行这个容器时，它就可以使用 Debian 操作系统。您还可以将所有代码依赖项打包到容器映像中，并将其用作代码构件。

容器映像存储在*注册表中*。您可以从注册表下载映像并在集群上运行它们。注册表可以存在于您的 AWS 基础设施内部或外部。

Amazon ECS

编排容器的运行时间



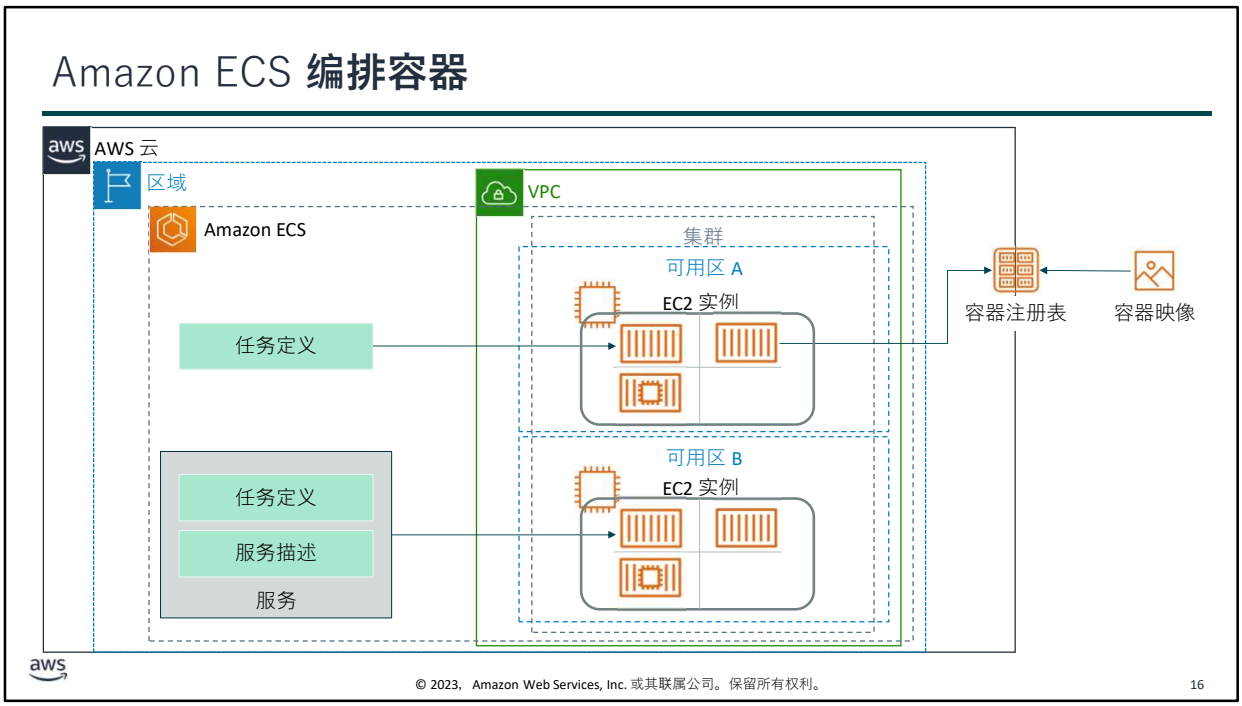
维护和扩展运行容器的实例队列

消除构建基础设施的复杂性

您可以在 Amazon Elastic Container Service (Amazon ECS) 上运行容器。Amazon ECS 是一项高度可扩展的高性能容器管理服务。它支持 Docker 容器，使您能够在 Amazon Elastic Compute Cloud (Amazon EC2) 实例的托管集群上轻松运行应用程序。

Amazon ECS 是一种可扩展的集群服务，用于托管容器：
- 可以在数秒内扩展到数以千计的 Docker 容器
- 监控容器部署
- 管理运行容器的集群的状态
- 使用内置的调度器或第三方调度器（Apache Mesos、Blox）对容器进行调度
- 可使用 API 进行扩展
- 可使用 AWS Fargate 或 Amazon EC2 启动类型启动

通过将 Spot 实例与按需实例和预留实例混合使用，可以大规模运行 ECS 集群。



Amazon ECS 是一项区域服务，可在区域内的多个可用区中以高度可用的方式简化运行应用程序容器。您可以在新的或现有的 Virtual Private Cloud (VPC) 中创建 ECS 集群。集群是资源的逻辑分组。

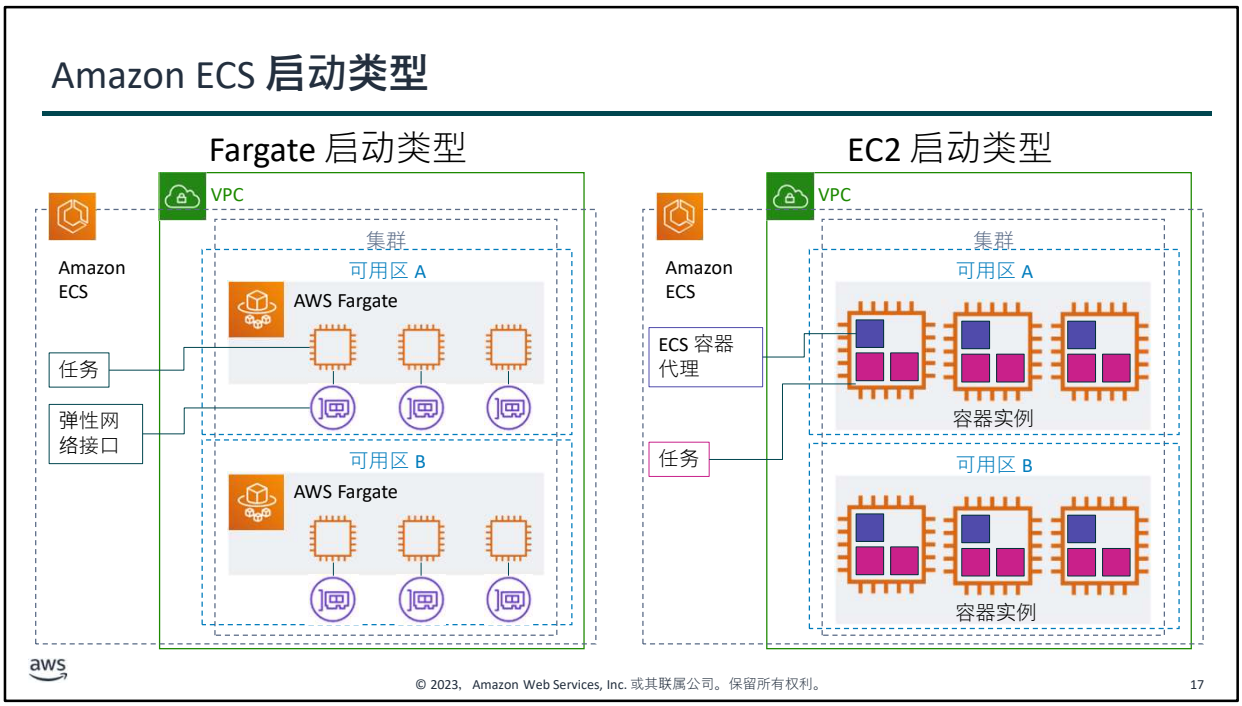
集群启动并运行后，您可以定义任务定义和服务，指定在集群中运行哪些 Docker 容器映像。

任务定义是 JavaScript 对象表示法 (JSON) 格式的文本文件。它描述构成应用程序的一个或多个容器，最多 10 个。您可以将它视为应用程序的蓝图。任务定义指定应用程序的参数 – 例如，要使用的容器和启动类型。其他参数包括应为应用程序打开哪些端口，以及任务中的容器应使用哪些数据卷。

服务让您能够指定要在集群中运行并维护多少个任务定义副本。您可以选择使用弹性负载均衡负载均衡器将传入流量分配给服务中的容器。Amazon ECS 保持该数量的任务，并使用负载均衡器来协调任务调度。

在为应用程序创建任务定义后，您可以指定将在集群上运行的任务的数量。任务是集群内的任务定义的实例化。当您使用 ECS 运行任务时，可以将它们放在集群中。

Amazon ECS 从您指定的注册表中下载容器映像，并在集群内运行这些映像。



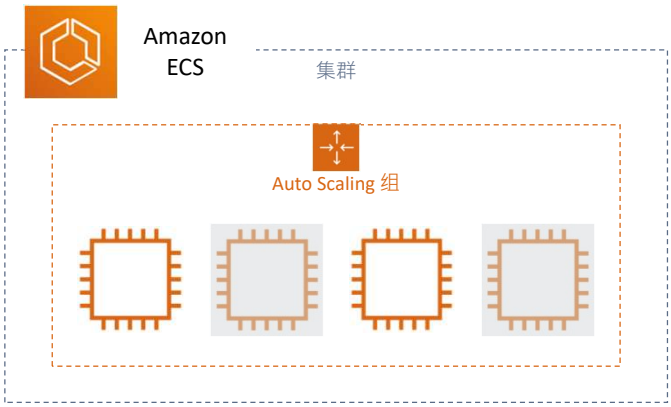
Amazon ECS 提供两种启动类型，用于托管容器化应用程序。

您可以使用 Fargate 启动类型在 Amazon ECS 管理的无服务器基础设施上托管集群。您只需将应用程序打包到容器中，指定 CPU 和内存要求，定义联网和 AWS Identity and Access Management (IAM) 策略，然后启动应用程序即可。

或者，如果您想要更多的控制权，可以使用 EC2 启动类型在*您*管理的 EC2 容器实例集群上托管任务。*容器实例*是运行 *Amazon ECS 容器代理*的 EC2 实例。您可以使用 Amazon ECS，根据资源需求、隔离策略和可用性要求安排集群中的容器放置。有关不同调度选项的信息，请参阅调度 Amazon ECS 任务。Amazon ECS 会跟踪集群中的所有 CPU、内存和其他资源。它还会根据您指定的资源需求为容器找到最佳的运行服务器。

有关 Fargate 和 EC2 启动类型的更多信息，请参阅 Amazon ECS 启动类型。

Amazon ECS 集群弹性伸缩



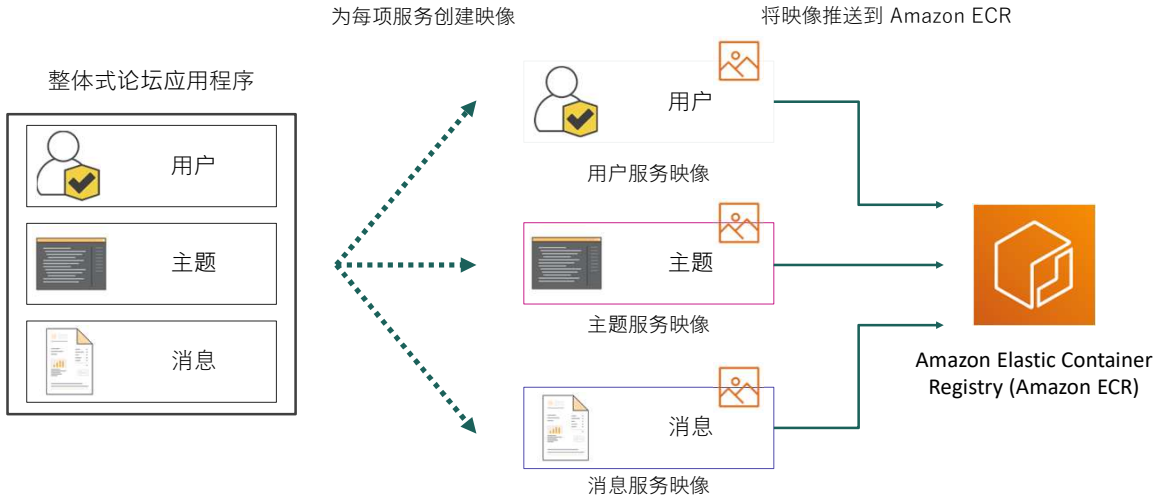
您可以为 Amazon ECS 集群创建 Auto Scaling 组。Auto Scaling 组包含您可以使用 Amazon CloudWatch 警报扩展（和缩减）的容器实例。如果您将 Auto Scaling 组配置为删除容器实例，则任何正在已删除容器实例上运行的任务都将停止。如果您的任务作为服务的一部分运行，则 Amazon ECS 会在所需资源可用的情况下在另一个实例上重新启动这些任务。此类所需资源的示例包括 CPU、内存和端口。不过，手动启动的任务不会自动重新启动。

您还可以利用 Amazon ECS 集群弹性伸缩，这使您可以更好地控制集群中的任务扩展方式。它提高了集群扩展的速度和可靠性。它使您能够控制集群中维护的空闲容量，并在缩减时自动管理实例终止。

通过集群弹性伸缩，您可以配置 Amazon ECS 以自动缩减和扩展 Auto Scaling 组。集群弹性伸缩依赖于容量提供程序，这些提供程序将 ECS 集群链接到您想要使用的 Auto Scaling 组。每个 Auto Scaling 组都与一个容量提供程序关联，并且每个容量提供程序只有一个 Auto Scaling 组。但是，许多容量提供程序可以与一个 ECS 集群相关联。为了自动扩展整个集群，每个容量提供程序都会管理其关联的 Auto Scaling 组的扩展。

有关集群弹性伸缩的更多信息，请参阅 Amazon ECS 集群弹性伸缩 AWS 新闻博客文章。

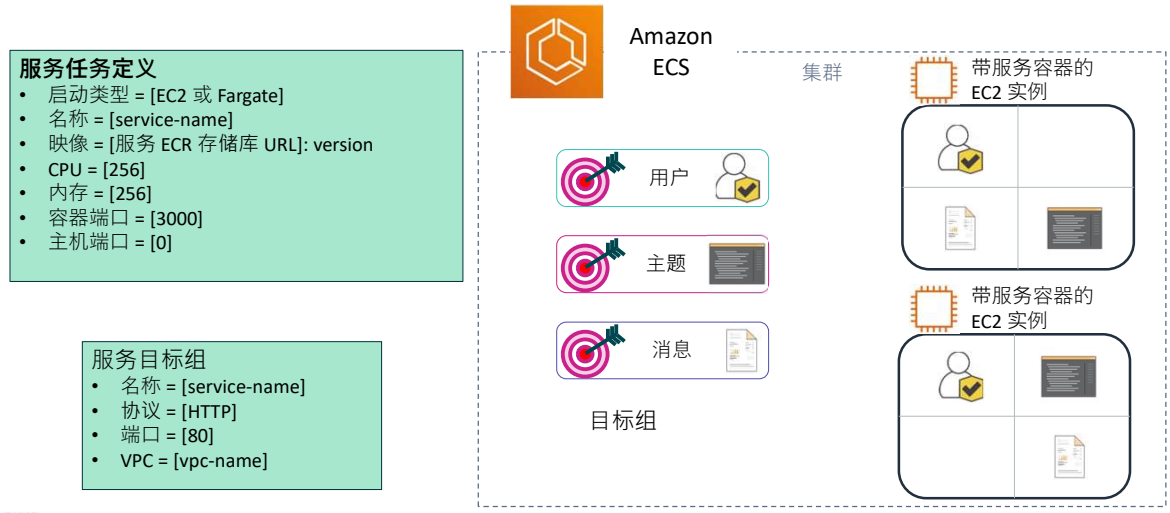
分解整体式应用程序 – 步骤 1：创建容器映像



再次考虑一下您之前看到的整体式论坛应用程序，其中整个应用程序作为单个服务运行。要使用微服务架构重新设计此应用程序的架构，您可以将每个应用程序进程作为单独的服务在其自己的容器中运行。借助微服务架构，这些服务可以独立于其他服务进行扩展和更新。

要将整体式应用程序部署为微服务应用程序，首先要为每项服务构建并标记映像。然后，在 Amazon Elastic Container Registry (Amazon ECR) 中注册映像。

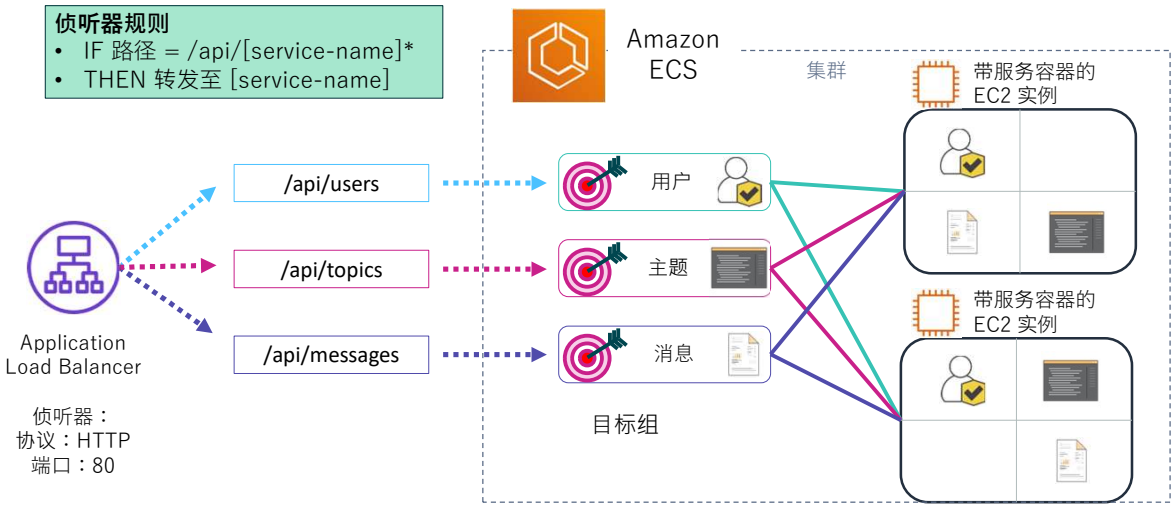
分解整体式应用程序 – 步骤 2：创建服务任务定义和目标组



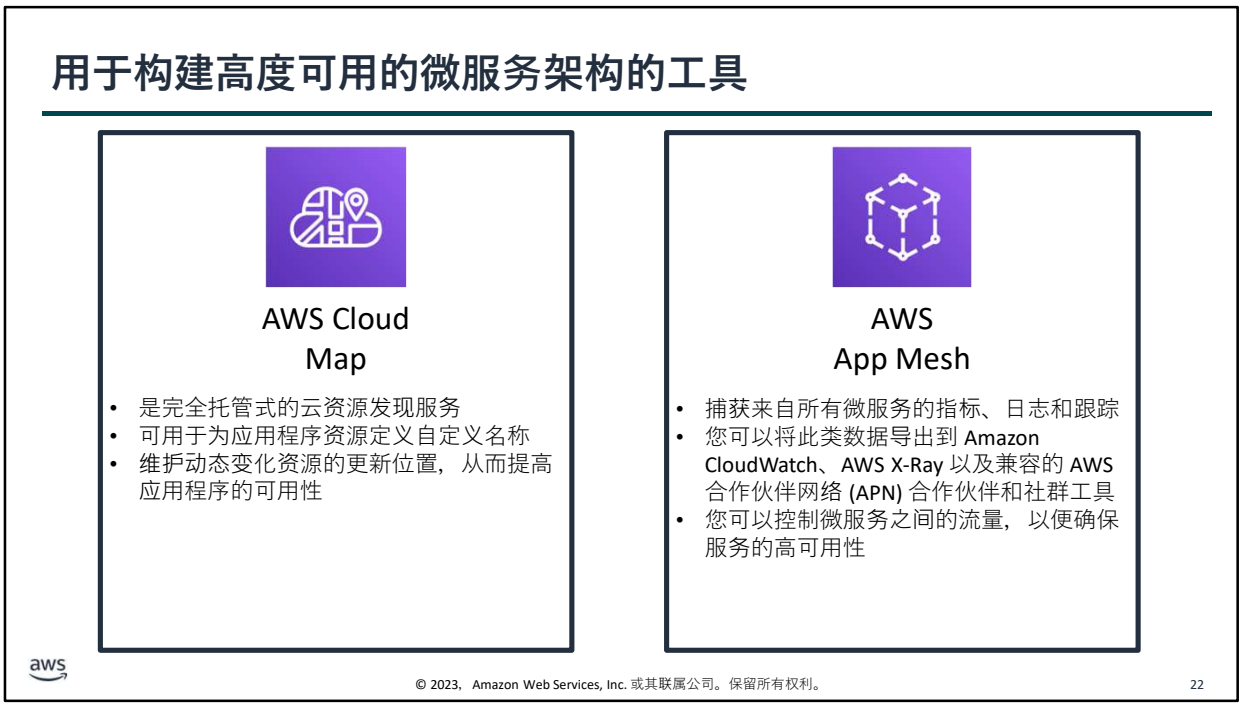
接下来，选择启动类型，然后为原始整体式应用程序的每个部分创建新服务。Amazon ECS 会在 ECS 集群上将每项服务部署到自己的容器中。

然后，为每项服务创建一个目标组。目标组可跟踪为该服务运行的每个容器的实例和端口。

分解整体式应用程序 – 步骤 3：将负载均衡器连接到服务



最后，创建 Application Load Balancer 并配置侦听器规则以连接到服务。侦听器会检查传入负载均衡器的连接请求，并使用规则对流量进行适当路由。在本示例中，Application Load Balancer 的侦听器会侦听端口 80 上的 HTTP 服务请求并将其路由到相应的服务。



AWS Cloud Map 和 AWS App Mesh 这两个工具，有助于您构建高度可用的微服务架构。

AWS Cloud Map 是一种完全托管式的云资源发现服务。您可以使用此服务为应用程序资源（如数据库、队列、微服务和其他云资源）定义自定义名称。AWS Cloud Map 维护这些动态变化的资源的更新位置。这种位置维护可提高应用程序的可用性，因为 Web 服务总能发现其资源的最新位置。您只需对映射进行最少的人工干预，即可添加和注册任何资源。AWS Cloud Map 可在微服务和应用程序的服务发现、持续集成和运行状况监控方面提供帮助。

有关 AWS Cloud Map 的更多信息，请阅读此 AWS 开源博客文章。要进一步了解如何使用 AWS Cloud Map 启用容器化服务以发现彼此并相互连接，请阅读 AWS Fargate、Amazon EKS 和 Amazon ECS 现已与 AWS Cloud Map 集成。

创建任务定义时，可以启用 App Mesh 集成。AWS App Mesh 可以捕获所有微服务的指标、日志和跟踪。您可以将此类数据导出到 Amazon CloudWatch、AWS X-Ray 以及兼容的 AWS 合作伙伴网络 (APN) 合作伙伴和社区工具，以进行监控和跟踪。AWS App Mesh 还能让您控制微服务之间的流量流向，确保每项服务在部署期间、故障后以及应用程序扩展时都高度可用。

App Mesh 可让您配置微服务，通过代理直接相互连接，而无需在应用程序内编写代码或使用负载均衡器。App Mesh 使用开源服务网格代理 Envoy，它与微服务容器一起部署。

有关 AWS Cloud Map 和 AWS App Mesh 的更多信息，请参阅此 AWS YouTube 视频。

AWS Fargate

- 是完全托管式容器服务
- 可与 Amazon Elastic Container Service (Amazon ECS) 和 Amazon Elastic Kubernetes Service (Amazon EKS) 配合使用
- 预置、管理和扩展容器集群
- 管理运行时环境
- 提供弹性伸缩

在本节中，您了解到 Amazon ECS 提供两种启动类型：EC2 和 Fargate。

AWS Fargate 是一项完全托管式的容器服务，可与 Amazon ECS 和 Amazon Elastic Kubernetes Service (Amazon EKS) 配合使用。它使您无需管理服务器或集群即可运行容器。借助 AWS Fargate，您不再需要预置、配置和扩展虚拟机集群就可以运行容器。因此，您就不需要选择服务器类型，不需要决定在什么时候扩展集群，也不需要优化集群包装。AWS Fargate 让您省去了考虑服务器和集群以及与之交互的麻烦。Fargate 使您能够专注于设计和构建应用程序，而不是管理运行应用程序的基础设施。

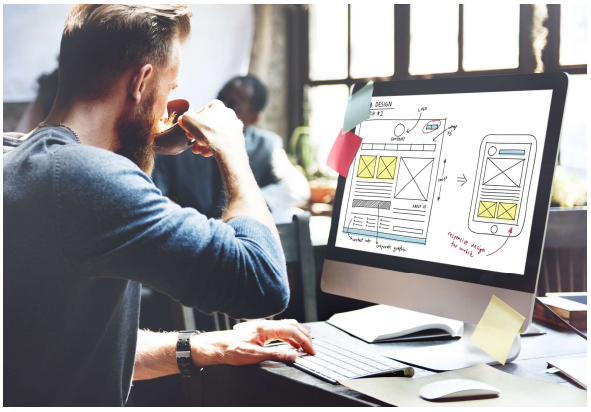
第 3 节要点

- Amazon ECS 是一项高度可扩展的高性能容器管理服务。它支持 Docker 容器，使您能够在 Amazon EC2 实例的托管集群上轻松运行应用程序。
- 集群弹性伸缩让您能够更好地控制集群内的任务扩展方式。
- AWS Cloud Map 使您能够为应用程序资源定义自定义名称。它维护这些动态变化的资源的更新位置。
- AWS App Mesh 是一种提供应用程序级联网的服务网格。它使您的服务能够在多种类型的计算基础设施之间轻松地相互通信。
- AWS Fargate 是一项完全托管式容器服务，使您无需管理服务器或集群即可运行容器。

本模块中这节内容的要点包括：

- Amazon ECS 是一项高度可扩展的高性能容器管理服务。它支持 Docker 容器，使您能够在 Amazon EC2 实例的托管集群上轻松运行应用程序。
- 集群弹性伸缩让您能够更好地控制集群内的任务扩展方式。
- AWS Cloud Map 使您能够为应用程序资源定义自定义名称。它维护这些动态变化的资源的更新位置。
- AWS App Mesh 是一种服务网格，可提供应用程序级联网，让您的服务轻松跨多种类型的计算基础设施相互通信。
- AWS App Mesh 是一种提供应用程序级联网的服务网格。它使您的服务能够在多种类型的计算基础设施之间轻松地相互通信。
- AWS Fargate 是一项完全托管式容器服务，使您无需管理服务器或集群即可运行容器。

模块 13 – 指导实验 1：将整体式 Node.js 应用程序拆分为微服务（可选实验）



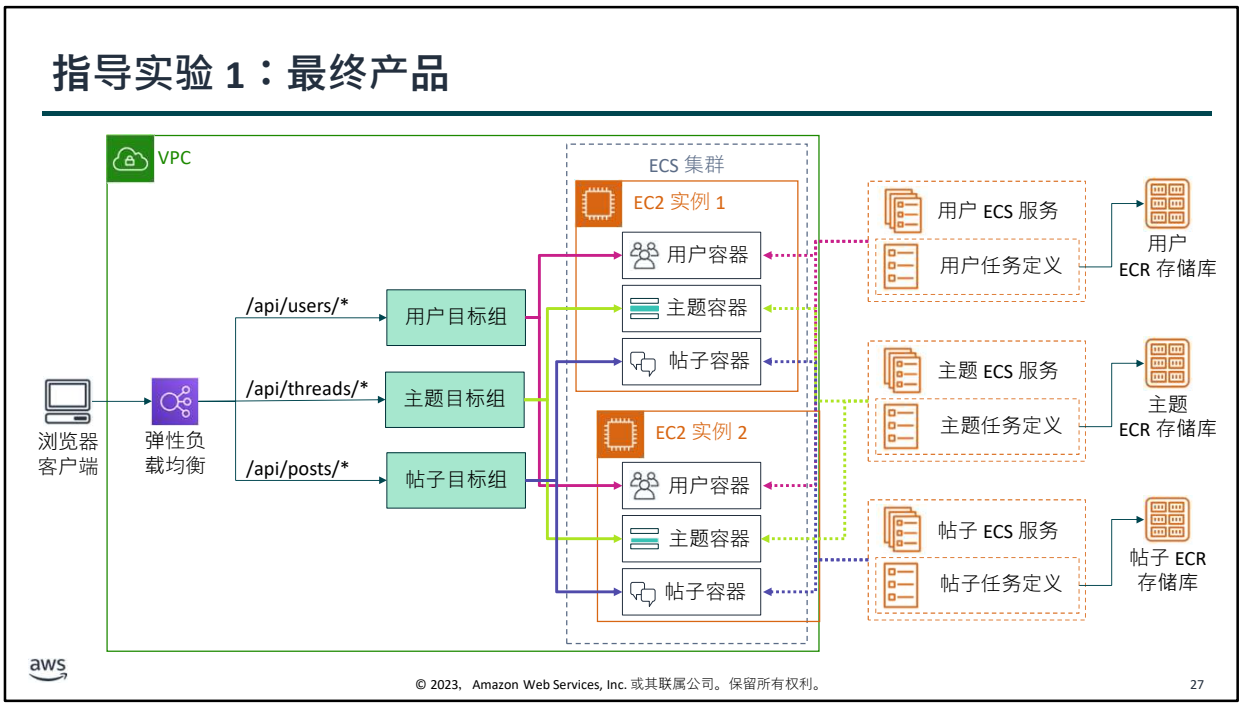
您可以选择完成模块 13 – 指导实验 1：将整体式 Node.js 应用程序拆分为微服务。本实验为可选实验。

指导实验 1：任务

1. 准备 AWS Cloud9 开发环境
2. 在基本 Node.js 服务器上运行整体式应用程序
3. 为 Amazon ECS 容器化整体式应用程序
4. 将整体式应用程序部署至 Amazon ECS
5. 将整体式应用程序重构为容器化微服务

在本指导实验中，您将完成以下任务：

1. 准备 AWS Cloud9 开发环境
2. 在基本 Node.js 服务器上运行整体式应用程序
3. 为 Amazon ECS 容器化整体式应用程序
4. 将整体式应用程序部署至 Amazon ECS
5. 将整体式应用程序重构为容器化微服务



该图总结了您完成实验后将构建的内容。

现在可以开始可选的指导实验了。

指导实验 1 总结：要点

完成这个指导实验之后，您的讲师可能会带您讨论此指导实验的要点。

介绍第 4 节：介绍无服务器架构。

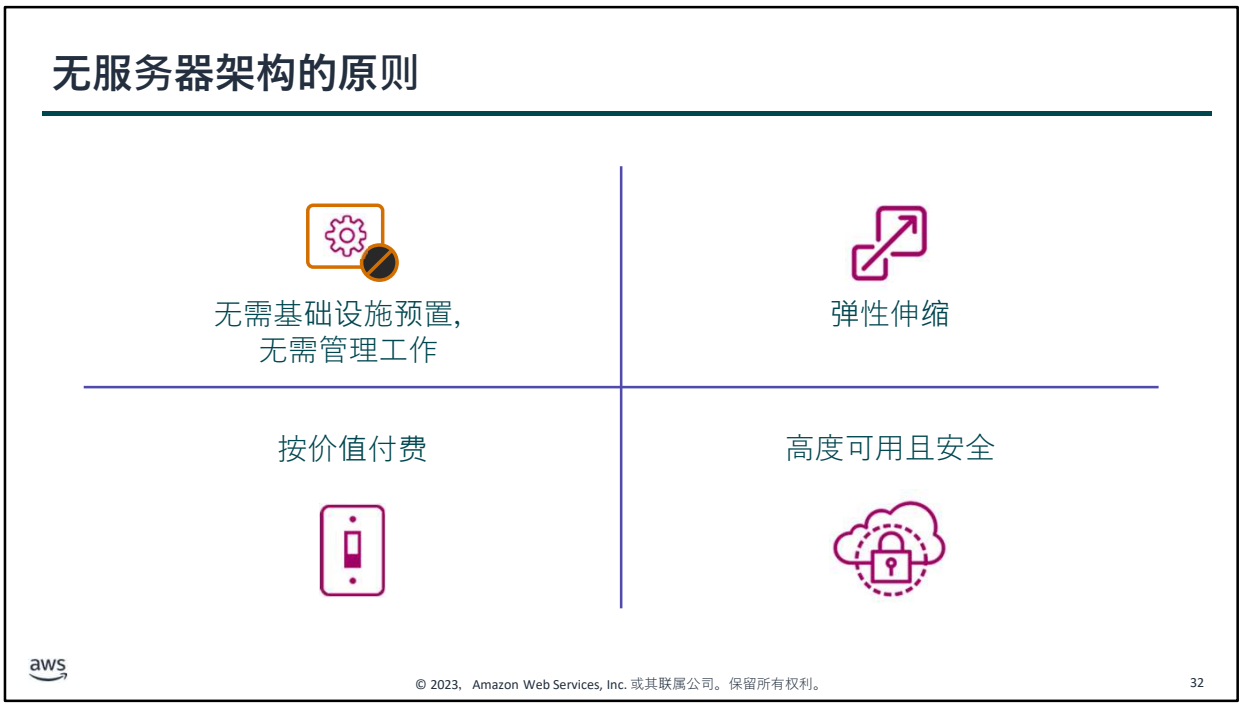
无服务器意味着什么？

使您可以在不考虑服务器的情况下构建并运行应用程序和服务的方法

到目前为止，您已经了解到可以使用容器通过 Amazon ECS 构建微服务应用程序。Amazon ECS 是一项容器编排服务，您可以在其中管理应用程序代码、数据源集成、安全配置、更新、网络配置、防火墙和管理任务。您还了解到可以使用 Fargate 启动类型在 Amazon ECS 管理的*无服务器*基础设施上托管集群。

但是，无服务器意味着什么？

*无服务器*是云原生架构，使您能够将更多的运营职责转移到 AWS，从而提高敏捷性和创新能力。无服务器使您可以在不考虑服务器的情况下构建并运行应用程序和服务。应用程序仍在服务器上运行。但是，AWS 会执行所有服务器管理任务，例如服务器或集群预置、修补、操作系统维护和容量预置。



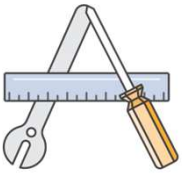
将无服务器定义为运营模型的原则包括：

- 无需预置或管理基础设施（无需预置、运行或修补服务器）
- 按消费单位自动扩展（按工作单位或消费单位进行扩展，而不是按服务器单位进行扩展）
- *按价值付费*定价模型（您只需为资源运行的持续时间付费，而不是按服务器单位付费）
- 内置的可用性和容错能力（因为可用性内置在服务中，因此无需设计可用性架构）

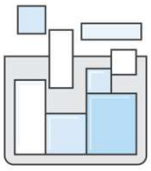
有关无服务器介绍的更多信息，请参阅此 AWS 网站。

无服务器的益处

更低的总体拥有成本

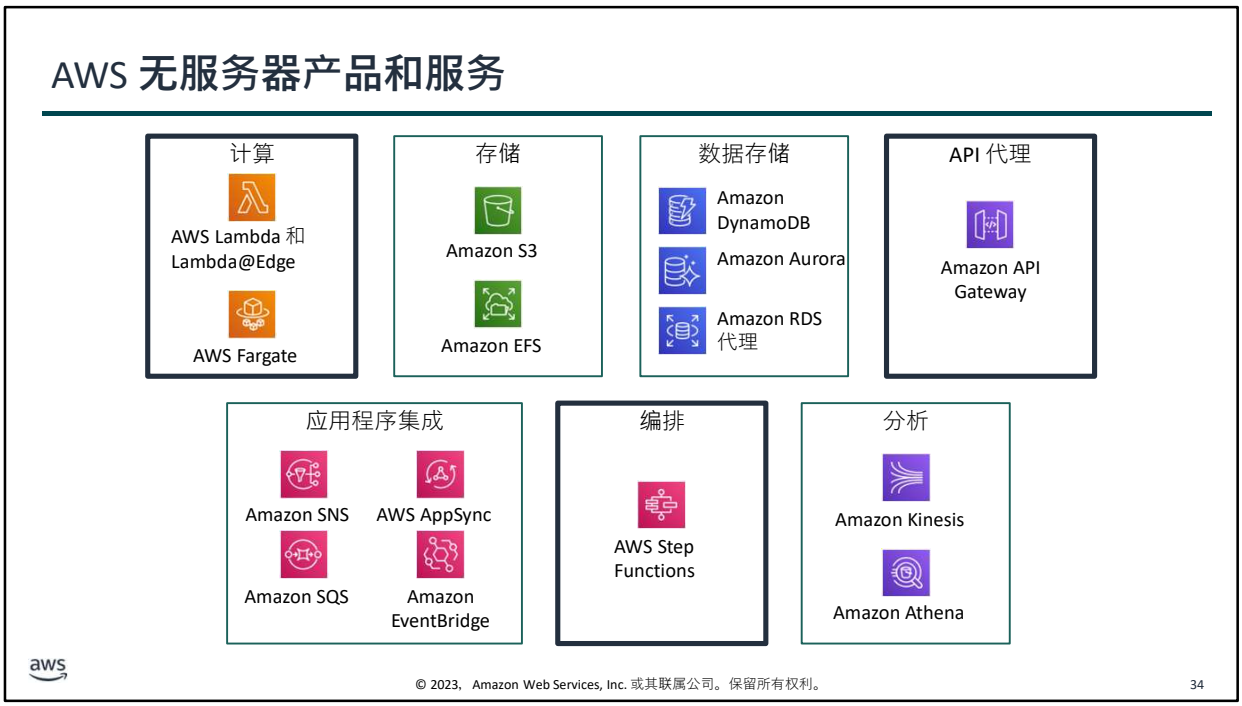


专注于应用程序
而不是配置



构建微服务应用程序

无服务器计算让您能够以更高的敏捷性和更低的总体拥有成本 (TCO) 构建现代化应用程序。通过使用无服务器架构，您可以专注于核心产品。不管是在云中还是在本地部署，您都无需担心服务器或运行时的管理和运行。这样可以减少开销，让您能够将时间和精力投入到可扩展并且可靠的产品的开发上。最后，无服务器架构使您能够构建微服务应用程序。



AWS 提供了许多产品，您可以使用它们在 AWS 上构建无服务器架构。到目前为止，在本课程中，您已经学习了其中几个产品。

本模块的其余部分重点介绍如何使用 AWS Lambda、Amazon API Gateway 和 AWS Step Functions 构建无服务器架构。

第 4 节要点

- 无服务器计算使您可以构建并运行应用程序和服务，而无需预置或管理服务器
- 无服务器架构提供以下益处 –
  - 更低的总体拥有成本 (TCO)
  - 您可以专注于应用程序
  - 您可以使用它们构建微服务应用程序

本模块中这节内容的要点包括:

- 无服务器计算使您可以构建并运行应用程序和服务，而无需预置或管理服务器
- 无服务器架构提供以下益处 –
  - TCO 更低
  - 您可以专注于应用程序
  - 您可以使用它们构建微服务应用程序

介绍第 5 节：使用 AWS Lambda 构建无服务器架构。

AWS Lambda

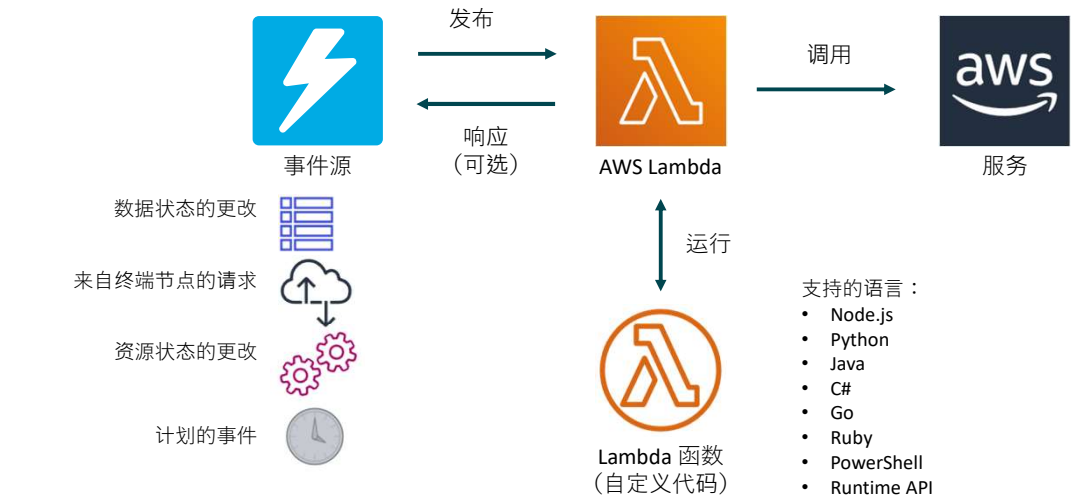
- 是完全托管式计算服务
- 按计划或根据事件（例如 Amazon S3 存储桶或 Amazon DynamoDB 表的更改）运行代码
- 支持 Java, Go, PowerShell, Node.js, C#, Python, Ruby 和 Runtime API
- 可以在靠近用户的边缘站点运行

AWS Lambda 是一项完全托管式的计算服务，可运行代码来响应事件并为您自动管理底层计算资源。 Lambda 在高可用性计算基础设施上运行您的代码，执行计算资源的所有管理工作，包括服务器和操作系统维护、容量预置、弹性伸缩、代码监控和日志记录。

AWS Lambda 原生支持 Java、Go、PowerShell、Node.js、C#、Python 和 Ruby 代码，还提供 Runtime API，让您能够使用任何其他编程语言来编写函数。

Lambda@Edge 是 Amazon CloudFront 的一项功能，它可让您在靠近应用程序用户的地方运行代码，从而提高性能、降低延迟。Lambda@Edge 根据 Amazon CloudFront 内容分发网络 (CDN) 生成的事件运行代码。利用 Lambda@Edge，您可以运行 Node.js 和 Python Lambda 函数，以自定义 Amazon CloudFront 分发的内容。有关如何添加 HTTP 安全响应标头的信息，请阅读这篇 AWS 联网和内容分发博客文章。

AWS Lambda 的工作原理

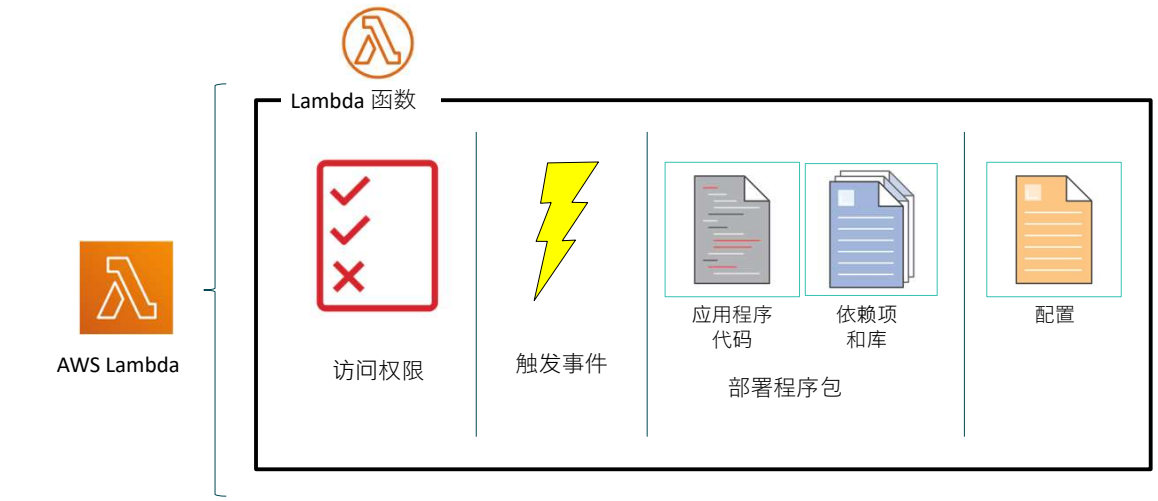


AWS Lambda 与其他 AWS 服务集成以调用 Lambda 函数。*Lambda 函数*是您使用 Lambda 支持的其中一种语言编写的自定义代码。您可以配置触发器来调用函数以响应资源生命周期事件、响应传入的 HTTP 请求、使用队列中的事件或按计划运行。

*事件源*是将事件发布到 Lambda 的实体。您的 Lambda 函数会处理事件，Lambda 代表您运行您的 Lambda 函数。

Lambda 函数是*无状态的*，这意味着它们与底层基础设施没有密切关系。Lambda 可以根据需要快速启动足够多的函数副本，以根据传入事件的速率进行扩展。

Lambda 函数



当您创建 Lambda 函数时，您将定义该函数的权限并指定触发该函数的事件。您还将创建一个部署程序包，其中包括您的应用程序代码以及运行您的代码所需的所有依赖项和库。最后，您将配置内存、超时和并发性等运行时参数。调用函数时，Lambda 将基于您选择的运行时和配置选项运行环境。

有关 AWS Lambda 运行方式的更多信息，请参阅 AWS 文档。

Lambda 函数剖析

Handler()
调用时要运行的函数

事件对象
Lambda 函数调用期间发送的数据

上下文对象
可用于与运行时信息交互的方法（请求 ID, 日志组等）

```
import json

def lambda_handler(event, context):
    # TODO implement
    return {
        'statusCode': 200,
        'body': json.dumps('Hello World')
    }
```


调用 Lambda 函数时，代码开始在处理程序处运行。处理程序是您创建并包含在程序包中的特定代码方法或函数。创建 Lambda 函数时指定处理程序。对于如何在程序包中定义和引用函数处理程序，所支持的每种语言都有各自不同的要求。在 Lambda 函数内成功调用处理程序后，运行时环境属于您编写的代码。

处理程序始终采用两个对象：事件对象和上下文对象。

事件对象提供有关触发 Lambda 函数的事件的信息。此类事件可以是 AWS 服务生成的预定义对象，也可以是可序列化字符串形式的用户自定义对象。此类字符串的示例可能是普通的旧 Java 对象 (POJO) 或 JSON 流。

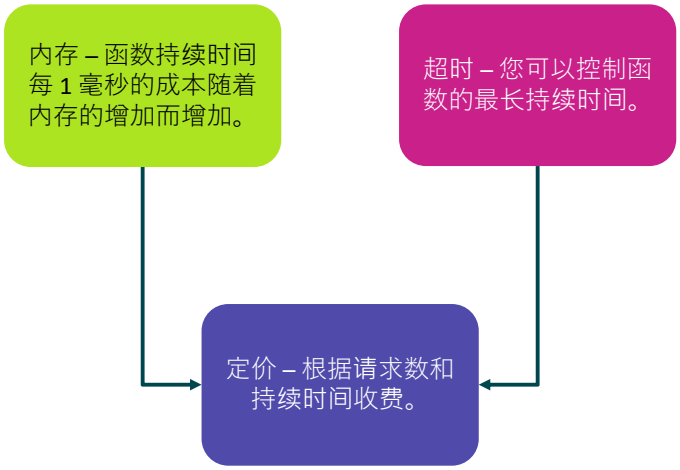
事件对象的内容包括 Lambda 函数驱动其逻辑所需的所有数据和元数据。事件对象的内容和结构各不相同，具体取决于由哪个事件源创建。例如，API Gateway 创建的事件包含与 API 客户端发出的 HTTPS 请求相关的详细信息 – 例如路径、查询字符串和请求体。而 Amazon 创建的事件包括有关存储桶和新对象的详细信息。

上下文对象由 AWS 生成，提供有关运行时环境的元数据。上下文对象使您的函数代码能与 Lambda 运行时环境进行交互。上下文对象的内容和结构因 Lambda 函数使用的语言运行时而异。

不过，上下文对象至少包含：

- awsRequestId – 此属性用于跟踪 Lambda 函数的特定调用（对于错误报告或联系 AWS Support 时很重要）
- logStreamName – 您的日志语句将被发送到的 CloudWatch 日志流
- getRemainingTimeInMillis() – 该方法返回函数运行超时前剩余的毫秒数

Lambda 函数配置和计费



内存和超时是确定 Lambda 函数执行方式的配置。这些配置会影响您的账单。如果您使用 AWS Lambda，我们将根据您函数的请求数量（所有函数的请求总数）和持续时间（代码运行花费的时间）向您收费。价格取决于您为函数分配的内存量。

内存 – 您可以指定要分配给 Lambda 函数的内存量。然后，Lambda 分配与内存成比例的 CPU 处理能力。Lambda 的定价方式是，每 1 毫秒函数持续时间的成本随着内存配置的增加而增加。例如，假设您有一个内存为 256 MB 的 Lambda 函数，运行时间为 110 毫秒。与运行时间相同、内存容量为 128 MB 的 Lambda 函数相比，该函数的成本将增加一倍。

超时 – 您可以使用超时配置来控制函数的最长持续时间。您可以将函数的超时值设置为不超过 15 分钟的任意值。当达到指定超时后，AWS Lambda 会停止 Lambda 函数的运行。使用超时可以避免因长时间运行函数而产生更高的成本。您必须在不让函数运行太久和正常情况下能够完成函数之间找到适当的平衡。

请遵循以下最佳实践：

- 测试 Lambda 函数的性能，确保选择了最佳内存大小配置。您可以在 Amazon CloudWatch Logs 中查看函数的内存使用情况。

- 对 Lambda 函数进行负载测试，分析函数的运行时间，确定最佳超时值。当您的 Lambda 函数对可能无法处理 Lambda 函数扩展的资源进行网络调用时，这一点非常重要。

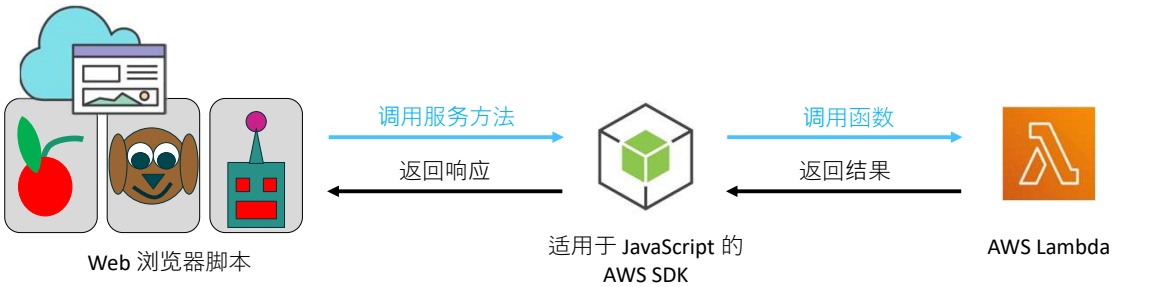
有关更多信息，请参阅以下资源：

- AWS Lambda 限制
- AWS Lambda 定价

演示：创建 AWS Lambda 函数

现在，讲师可能会选择演示如何创建 AWS Lambda 函数。

AWS Lambda 示例：
模拟老虎机浏览器游戏

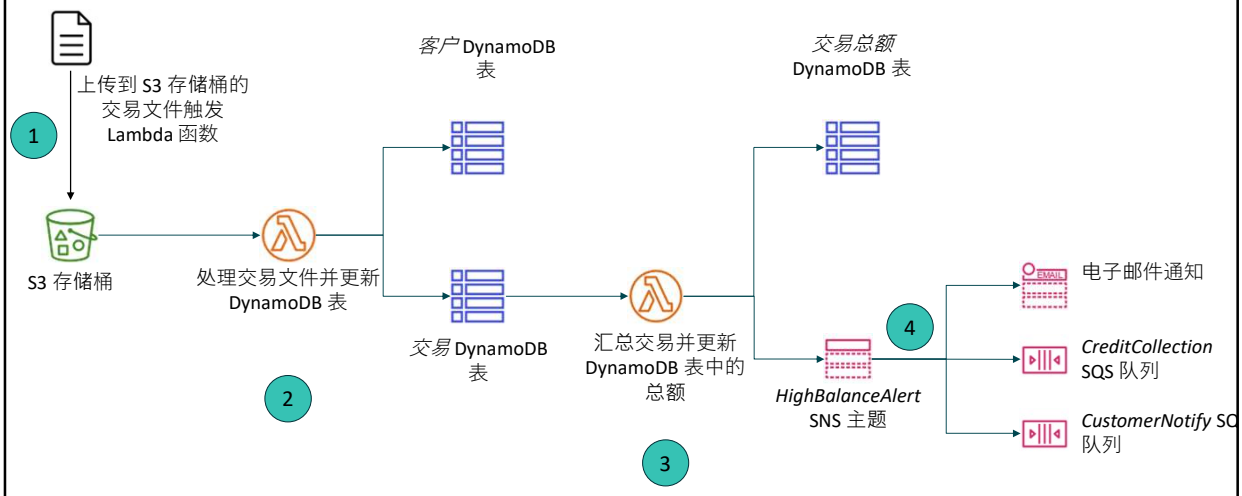



```
lambda.invoke(pullParams, function(error, data) {
 if (error) {
 prompt(error);
 } else {
 pullResults = JSON.parse(data.Payload);
 }
});
```

```
{
 isWinner: false,
 leftWheelImage : {S : 'cherry.png'},
 midWheelImage : {S : 'puppy.png'},
 rightWheelImage : {S : 'robot.png'}
}
```


您可以创建 Lambda 函数来执行各种任务。此示例使用的是一款模拟老虎机的基于浏览器的游戏。该游戏会调用一个 Lambda 函数，生成每次拉动老虎机的随机结果。函数返回这些结果，并使用相应的图像文件名来显示结果。图像存储在 Amazon S3 存储桶中，而存储桶配置为静态 Web 主机，用于托管提供应用程序体验所需的 HTML、CSS 和其他资产。

基于事件的 Lambda 函数示例：订单处理



此示例展示了如何在订单处理解决方案中使用 Lambda。

在本架构中：

1. 客户将交易文件上传到 S3 存储桶，从而触发 Lambda 函数的运行。
2. Lambda 函数会处理交易文件并更新*客户*和*交易* DynamoDB 表。
3. 对*交易* DynamoDB 表的更改将触发第二个 Lambda 函数来聚合事务，并更新*交易总额* DynamoDB 表中的总数。它还向 *HighBalancerAlert* SNS 主题推送一条消息。
4. *HighBalancerAlert* SNS 主题向客户发送电子邮件通知，并更新 *CreditCollection* 和 *CustomerNotify* SQS 队列，以便进行付款处理。

Lambda 层

- 使函数能够轻松共享代码 – 您可以一次上传一层并在任何函数中引用它
- 促进责任分离 – 开发人员可以更快地迭代编写业务逻辑
- 使您能够保持较小的部署程序包
- 限制 –
  - 最多五层
  - 250 MB

在构建无服务器应用程序时，通常会在 Lambda 函数之间共享代码。可以是两个或更多函数使用的自定义代码，也可以是为简化业务逻辑实施而添加的标准库。

以前，您会将这些共享代码与使用这些代码的所有函数一起打包和部署。现在，您可以将 Lambda 函数配置为以层的形式包括其他代码和内容。层是一个包含库、自定义运行时或其他依赖项的 .zip 归档文件。

使用 Lambda 层，函数可以共享代码。开发人员使用层一次性上传代码，然后多次重复使用。利用层，您可以在函数中使用库，而不必将库包含在部署程序包中。

这样共享代码有助于促进责任分工。一个人可以负责管理核心库。另一个人可以负责使用库代码并在库代码的基础上构建应用程序逻辑。

通过使用层，您可以将部署程序包保持较小，从而使开发变得更轻松。

一个函数一次最多可以使用五个层。函数和所有层的解压后总大小不能超出 250 MB 的解压后部署程序包大小限制。

有关层的更多信息，请参阅 AWS Lambda 层。

演示：结合使用 AWS Lambda 和 Amazon S3

现在，讲师可能会选择演示如何配置 Amazon S3 事件以触发 Lambda 函数。

Demonstration diagram

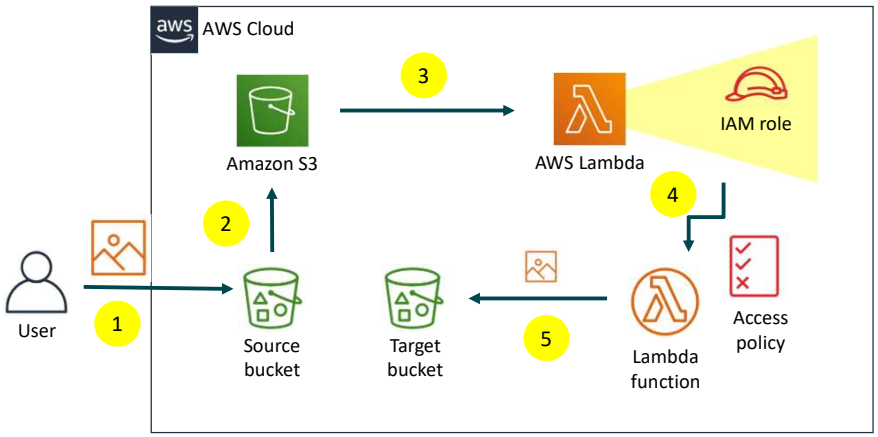
1. User uploads image to source S3 bucket.

2. Amazon S3 detects the object-created event.

3. Amazon S3 publishes the event to Lambda.

4. Lambda runs the Lambda function.

5. The Lambda function resizes the original image and saves the thumbnail to the target S3 bucket.



This demonstration covers the following steps:

1. A user uploads an image to a source S3 bucket.
2. Amazon S3 detects the object-created event.
3. Amazon S3 publishes the event to Lambda by invoking the Lambda function and passing event data as a function parameter.
4. Lambda assumes an IAM role that allows it to run the function, and then runs the Lambda function.
5. From the event data it receives, the Lambda function knows the source bucket and the object key name. The Lambda function reads the object, creates a thumbnail of the original image, and saves the thumbnail to the target S3 bucket.

```
import boto3
import os
import sys
import uuid
from PIL import Image
import PIL.Image

s3_client = boto3.client('s3')

def resize_image(image_path, resized_path):
    with Image.open(image_path) as image:
        image.thumbnail((128, 128))
        image.save(resized_path)

def handler(event, context):
    for record in event['Records']:
        bucket = record['s3']['bucket']['name']
        key = record['s3']['object']['key']
        download_path = '/tmp/{}{}'.format(uuid.uuid4(), key)
        upload_path = '/tmp/resized-{}'.format(key)

        s3_client.download_file(bucket, key, download_path)
        resize_image(download_path, upload_path)
        s3_client.upload_file(upload_path, '{}-resized'.format(bucket), key)
```

Receives S3 event and downloads the image to local storage

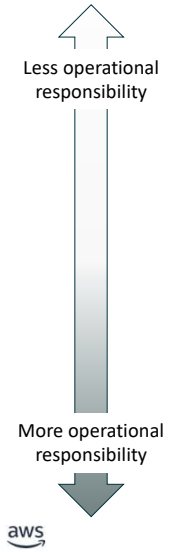
Resizes the image

Uploads resized image to the *–resized* bucket

The Lambda function code performs the following steps:

- Receives an S3 event, which contains the name of the incoming object (Bucket, Key)
- Downloads the image to local storage
- Resizes the image using the *Pillow* library
- Uploads the resized image to the *–resized* S3 bucket

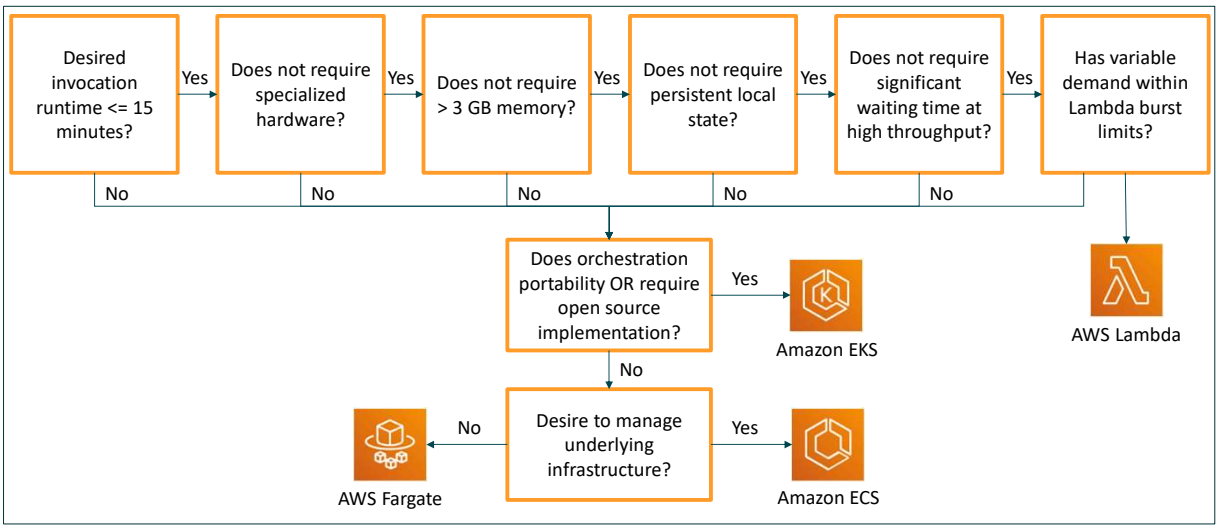
Comparison of operational responsibility for container and serverless architectures



	AWS Manages	Customer Manages
AWS Lambda Serverless Functions	• Data source integrations • Physical hardware, software, networking, and facilities • Provisioning	• Application code
AWS Fargate Serverless Containers	• Container orchestration and provisioning • Cluster scaling • Physical hardware, host OS/kernel, networking, and facilities	• Application code • Data source integrations • Security configuration and updates, network configuration, and management tasks
Amazon ECS and Amazon EKS Container Management as a Service	• Container orchestration control plane • Physical hardware software, networking, and facilities	• Application code • Data source integrations • Work clusters • Security configuration and updates, network configuration, firewall, and management tasks

Which compute service you use to build your application depends on the level of control that you want. You share a spectrum of operational responsibility with AWS over your compute options for container and serverless architectures.

Choosing a compute platform: Containers versus AWS Lambda



This slide displays a rough guideline for selecting your compute platform. The recommendation is based on AWS Lambda limits (memory limitation, time limitation).

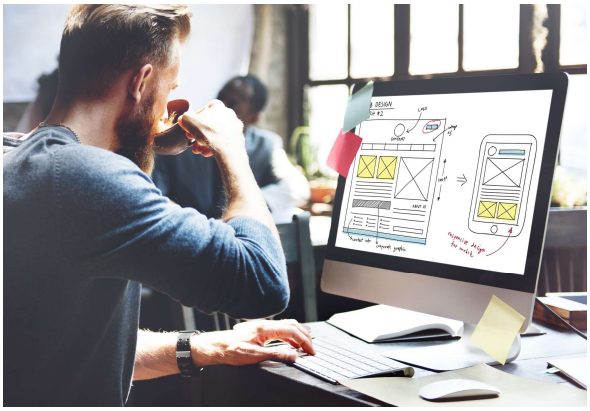
第 5 节要点

- Lambda 是一种无服务器计算服务，提供内置的容错能力和弹性伸缩功能
- Lambda 函数是您编写的用于处理事件的自定义代码
- Lambda 函数由处理程序调用，处理程序使用事件对象和上下文对象作为参数
- 事件源是触发 Lambda 函数运行的 AWS 服务或开发人员创建的应用程序
- Lambda 层使函数能够共享代码并能够保持部署程序包较小

本模块中这节内容的要点包括：

- Lambda 是一种无服务器计算服务，提供内置的容错能力和弹性伸缩功能。
- Lambda 函数是您编写的用于处理事件的自定义代码。
- Lambda 函数由处理程序调用，处理程序使用事件对象和上下文对象作为参数。
- 事件源是触发 Lambda 函数运行的 AWS 服务或开发人员创建的应用程序。
- Lambda 层使函数能够共享代码并能够保持部署程序包较小。

模块 13 – 指导实验 2：在 AWS 上实施无服务器架构



现在您将完成模块 13 – 指导实验 2：在 AWS 上实施无服务器架构。

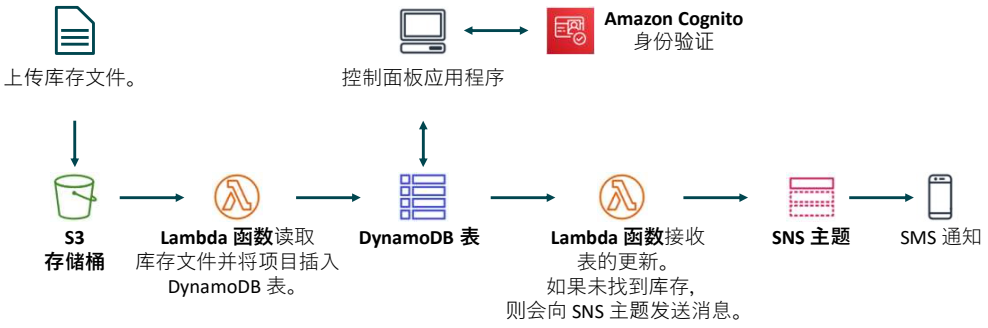
指导实验 2：任务

1. 创建 Lambda 函数以加载数据
2. 配置 Amazon S3 事件
3. 测试加载过程
4. 配置通知
5. 创建 Lambda 函数以发送通知
6. 测试系统

在本指导实验中，您将完成以下任务：

1. 创建 Lambda 函数以加载数据
2. 配置 Amazon S3 事件
3. 测试加载过程
4. 配置通知
5. 创建 Lambda 函数以发送通知
6. 测试系统

指导实验 2：最终产品



该图总结了您完成实验后将构建的内容。

大约 40 分钟

开始模块 13 – 指导实验 2：在 AWS 上实施无服务器架构

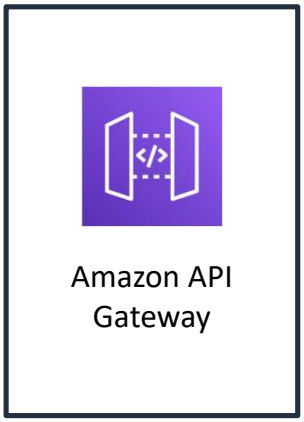
现在可以开始指导实验了。

指导实验 2 总结：要点

完成这个指导实验之后，您的讲师可能会带您讨论此指导实验的要点。

介绍第 6 节：使用 Amazon API Gateway 扩展无服务器架构。

Amazon API Gateway



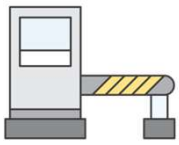
- 使您能够创建、发布、维护、监控和保护作为应用程序后端资源入口点的 API
- 处理高达数十万个并发 API 调用
- 可以处理在以下工具上运行的工作负载 –
  - Amazon EC2
  - Lambda
  - 任何 Web 应用程序
  - 实时通信应用程序
- 可以托管和使用多个版本和多个阶段的 API

Amazon API Gateway 是一项完全托管式服务，使您可以创建、发布、维护、监控和保护任意规模的 API。您可以使用此服务创建表述性状态转移 (RESTful) 和 WebSocket API，这些 API 充当应用程序的入口点，以便它们可以访问后端资源。然后应用程序可从后端服务访问数据、业务逻辑或功能。此类服务包括在 Amazon EC2 上运行的应用程序、在 Lambda 上运行的代码、任何 Web 应用程序或实时通信应用程序。

API Gateway 可处理接受和处理多达数十万次并发 API 调用所涉及的所有任务。这些调用可能包括流量管理、授权和访问控制、监控以及 API 版本管理。API Gateway 没有最低费用，也没有启动成本。您只需为收到的 API 调用以及传出的数据量付费。利用 API Gateway 分级定价模式，您可以随着 API 使用量的增加而降低成本。

您可以使用 API Gateway 托管多个版本和阶段的 API。

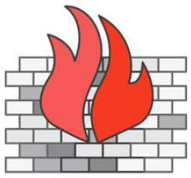
Amazon API Gateway 安全



需要授权

应用资源策略

限流设置



防止分布式拒绝服务 (DDoS) 和注入攻击

当您将 API 设为公开可用时，您就暴露在攻击者面前，他们会试图利用您的服务。借助 Amazon API Gateway，您可以通过多种方式保护 API。

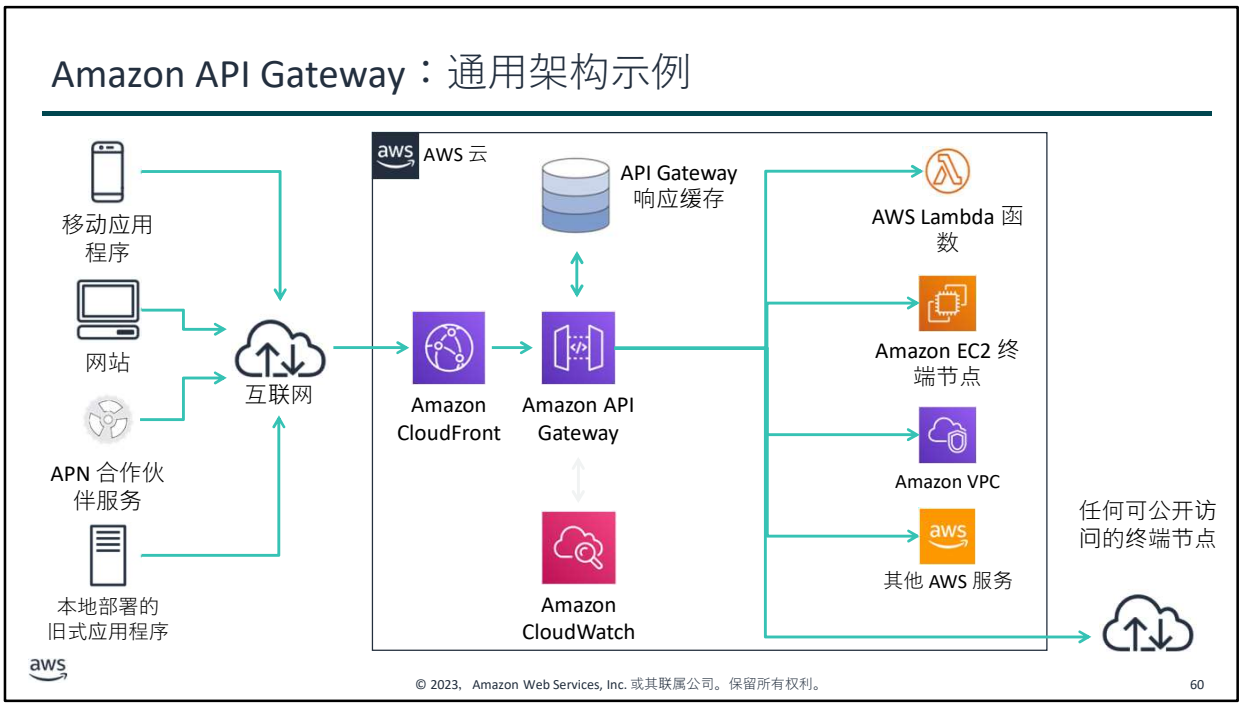
借助 Amazon API Gateway，您可以选择性地将 API 方法设置为需要授权。在设置需要授权的方法时，可以使用 AWS 签名版本 4 或 Lambda 授权方来支持自己的不记名令牌身份验证策略。AWS 签名版本 4 是为通过 HTTP 发送的 AWS 请求添加身份验证信息的过程。出于安全考虑，大多数发送到 AWS 的请求都必须使用访问密钥（包括访问密钥 ID 和秘密访问密钥）进行签名。与其他 AWS 服务一样，您可以使用这些 AWS 凭证来签署对您服务的请求并授权访问。您可以使用 Amazon Cognito 检索与 AWS 账户中的角色相关联的临时凭证。Lambda 授权方是一种 Lambda 函数，它通过使用像 OAuth 这样的持有者令牌身份验证策略来授权对 API 的访问。

您还可以将资源策略应用到 API，以限制对特定 Amazon VPC 或 VPC 终端节点的访问。您可以通过使用资源策略，让不同账户的 Amazon VPC 或 VPC 终端节点访问私有 API。

Amazon API Gateway 支持对 API 中的每个方法或路由进行限流设置。您可以为 REST API 中的每个方法和 WebSocket API 中的每个路由设置标准速率限制和每秒突发速率限制。

此外，您可以使用 AWS WAF 来保护 API Gateway API 的安全。AWS WAF 是一款 Web 应用程序防火墙，有助于保护您的 Web 应用程序免受常见 Web 攻击，这些攻击可能会影响可用性、危及安全性或消耗过多资源。

有关如何使用 Amazon API Gateway 保护 API 的更多信息，请参阅 Amazon API Gateway 常见问题中的安全和授权部分。



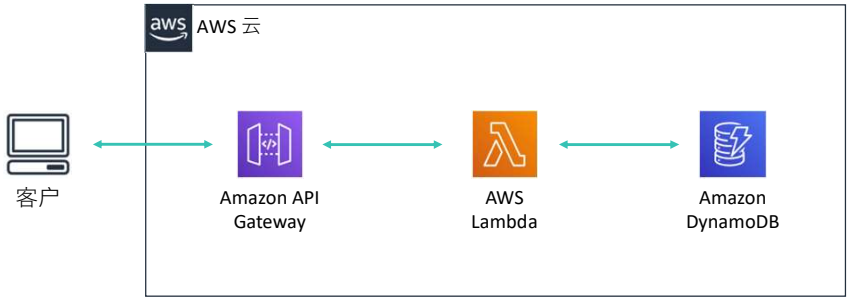
您可以使用 Amazon API Gateway 为应用程序提供 API 层。以下是使用 Amazon API Gateway 的通用架构示例。

在此示例中，前端客户端应用程序和应用程序服务通过互联网向 API Gateway 发送流量。通常，Amazon CloudFront 用于缓存静态内容。API Gateway 抽象并公开可以调用各种后端应用程序的 API。这些应用程序包括 Lambda 函数、在 EC2 实例上运行的 Docker 容器、Virtual Private Cloud (VPC) 或任何可公开访问的终端节点。如有必要，API Gateway 可以缓存响应。最后，所有 API 调用都可以使用 Amazon CloudWatch 进行监控。

您可以将 API Gateway 与其他 AWS 托管服务结合使用，为应用程序构建无服务器后端。例如，API Gateway 可以将请求代理到运行代码并生成响应的 Lambda 函数。您甚至可以创建将请求代理到其他 AWS 服务（例如 Amazon Simple Storage Service (Amazon S3)）的 API，而不必编写任何代码。您可以更快地运行生产规模的后端，因为您只需编写更少的代码，而且还可以将操作负担转移到 AWS 服务。

有关如何将 API Gateway 与 AWS Lambda 结合使用的示例，请参阅此 AWS 博客文章。

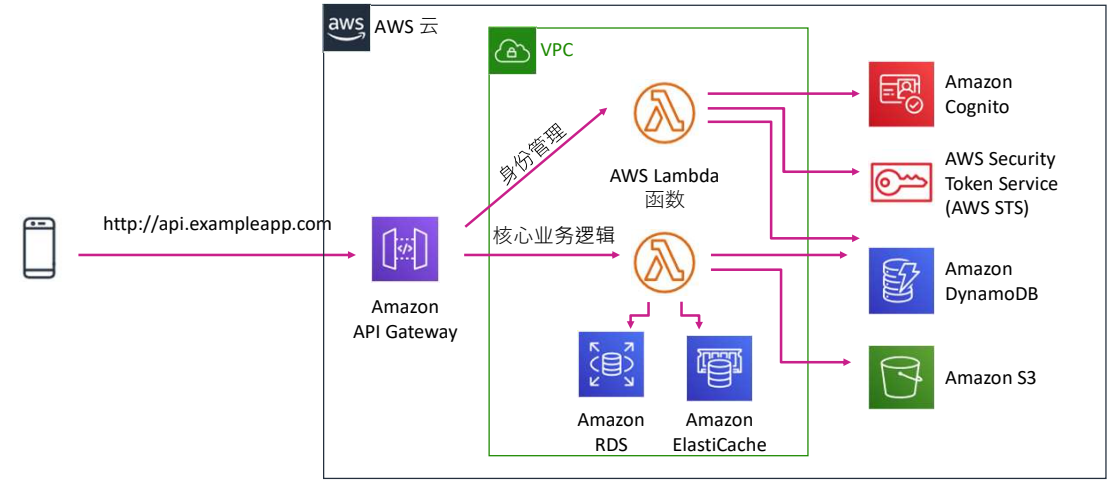
示例：RESTful 微服务



Amazon API Gateway 与 AWS Lambda 深度集成。此示例展示了如何在 RESTful 微服务应用程序的架构中将这两种服务结合使用。

在此示例中，客户可以通过进行 HTTP API 调用来使用微服务。Amazon API Gateway 托管来自客户的 RESTful HTTP 请求和对客户的响应。在这种情况下，API Gateway 提供内置的授权、限流、安全性、容错能力、请求/响应映射和性能优化。AWS Lambda 包含处理传入 API 调用的业务逻辑，并使用 DynamoDB 作为持久性存储。Amazon DynamoDB 可持久存储微服务数据，并根据需求进行扩展。由于微服务的设计目的是做好一件事，因此会定期采用无架构 NoSQL 数据存储。

示例：无服务器移动后端



再看一个例子，说明如何在无服务器架构中将 Amazon API Gateway 与 Lambda 结合使用，作为移动应用程序的后端网关。人们期望移动应用程序能够提供快速、一致、功能丰富的用户体验，而且这些应用程序通常会覆盖全球。此外，移动用户模式是动态的，其使用高峰无法预测。移动应用程序需要一套丰富的移动服务，这些服务可以无缝协作，同时又不影响后端基础设施的控制和灵活度。

在此示例中，移动应用程序向 Amazon API Gateway 发送请求，后者将请求转发给一个 Lambda 函数，该函数调用 Amazon Cognito 和 AWS Security Token Service (AWS STS) 来管理身份。Amazon Cognito 通过支持安全断言标记语言 (SAML) 或 OpenID Connect 的外部身份提供商 (IdP)、社交 IdP（如 Facebook、Twitter、Amazon）和自定义 IdP 对移动应用程序的用户进行身份验证。确认移动用户身份后，另一个 Lambda 函数将运行应用程序的核心业务逻辑。

第 6 节要点

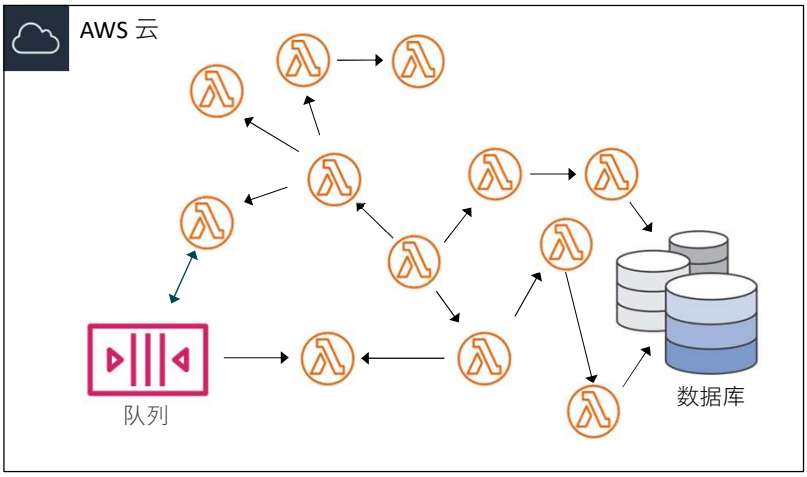
- Amazon API Gateway 是一项完全托管式服务，使您可以创建、发布、维护、监控和保护任意规模的 API。
- Amazon API Gateway 可充当应用程序后端资源的入口点。它抽象并公开可以调用各种后端应用程序的 API。这些应用程序包括 Lambda 函数、在 EC2 实例上运行的 Docker 容器、VPC 或任何可公开访问的终端节点。
- Amazon API Gateway 与 Lambda 深度集成。

本模块中这节内容的要点包括：

- Amazon API Gateway 是一项完全托管式服务，使您可以创建、发布、维护、监控和保护任意规模的 API。
- Amazon API Gateway 可充当应用程序后端资源的入口点。它抽象并公开可以调用各种后端应用程序的 API。这些应用程序包括 Lambda 函数、在 EC2 实例上运行的 Docker 容器、VPC 或任何可公开访问的终端节点。
- Amazon API Gateway 与 Lambda 深度集成。

介绍第 7 节：使用 AWS Step Functions 编排微服务。

微服务应用程序的挑战



随着应用程序规模的扩大，组件的数量也会增加，因此可能会出现多种运行任务的模式和顺序。例如，假设有一个使用 Lambda 函数构建的微服务应用程序。您可能希望当且仅当另一个函数成功运行后，立即调用 Lambda 函数。您可能希望并行调用两个函数，然后将合并后的结果提供给第三个函数。或者，您可能希望根据另一个函数的输出来选择调用两个函数中的哪一个。

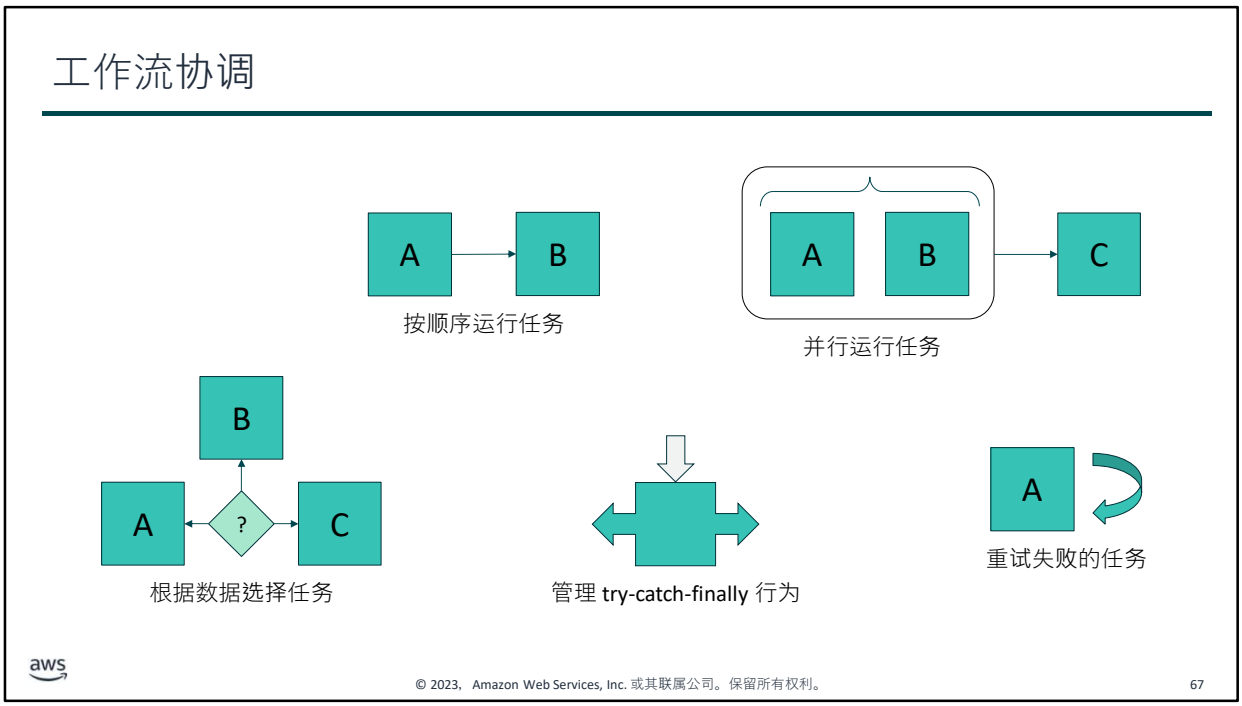
出于多种原因，函数调用可能会导致错误。您的代码可能引发异常、超时或用尽内存。运行您的代码的运行时可能会遇到错误并因此停止。当发生错误时，您的代码可能已经完全运行、部分运行，或者完全未运行。在大多数情况下，调用您函数的客户端或服务会在遇到错误时重试，因此您的代码必须能够重复处理同一事件，而不会产生不必要的影响。如果您的函数管理资源或向数据库写入内容，您必须处理多次提出相同请求的情况。

您需要一种方法来协调应用程序的各个组件。该协调层必须能够根据不断变化的工作负载自动扩展，并处理错误和超时。它还必须在应用程序运行时维护其状态，例如跟踪当前正在运行的步骤，并存储在工作流各步骤之间移动的数据。这些功能有助于您构建和运行应用程序。您还希望应用程序具有可视性，以便可以排查错误和跟踪性能。

AWS Step Functions

- 通过使用可视化工作流协调微服务
- 让您可以逐步执行应用程序的功能
- 自动触发和跟踪每个步骤
- 在步骤失败时提供简单的错误捕获和日志记录

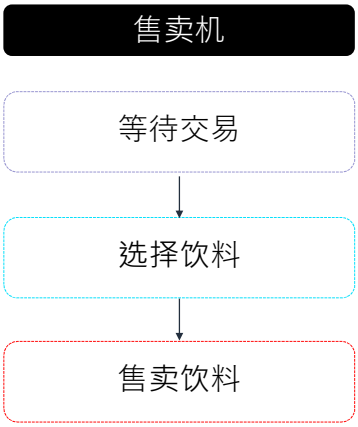
AWS Step Functions 是一项 Web 服务，让您能够使用可视化工作流来协调分布式应用程序和微服务的组件。Step Functions 是协调组件和逐步执行应用程序函数的可靠方法。Step Functions 提供图形控制台，以按照一系列步骤实现应用程序组件的可视化。它能自动触发和跟踪每个步骤，还能在出现错误时重试，从而使应用程序按预期顺序运行。Step Functions 记录每个步骤的状态，因此您可以快速诊断并调试问题。



AWS Step Functions 为您管理应用程序的逻辑。它实施了基本的基元，如按顺序或并行运行任务、分支和超时。这种技术可以删除微服务和函数中可能重复的额外代码。AWS Step Functions 利用内置的 try-catch 和重试功能自动处理错误和异常，无论任务需要几秒钟还是几个月才能完成均如此。您可以自动重试失败或超时的任务。您可以对不同类型的错误做出不同的响应，并通过回退到指定的清理和恢复代码来从容恢复。

状态机

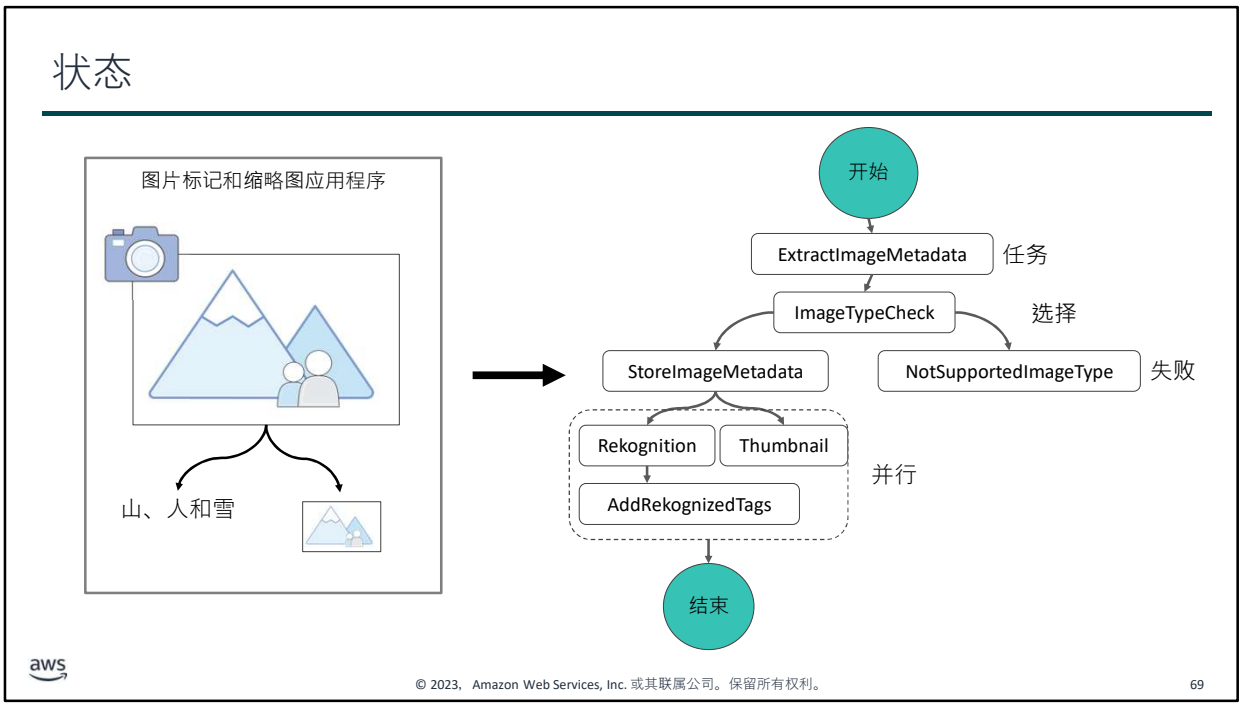
状态机是可以执行工作的状态集合。



借助 AWS Step Functions，您可以使用基于 JSON 的 Amazon States Language 在 AWS 环境中创建自己的状态机并让其自动运行。该语言包含由各种状态、任务、选项、错误处理等组成的结构。

*状态机*是可以执行工作的状态集合。

饮料售卖机就是一种常见的状态机。机器在开始时处于运行状态（等待交易），然后在放入钱时转到饮料选择状态。然后进入售卖状态，将饮料提供给客户。完成后，售卖机返回运行状态。



有限状态机可以将算法表示为许多状态、其关系，及其输入和输出。*状态*是您状态机中的元素。各个状态可以根据其输入作出决定，执行操作并将输出传递给其他状态。

状态类型

任务	由状态机执行的一个工作单元
选择	向状态机添加分支逻辑
失败	停止运行中的状态机并将其标记为失败
成功	成功停止运行中的状态机
传递	将其输入传递到其输出，而不执行任何工作
等待	延迟指定时间，然后再继续
并行	创建在状态机中运行的并行分支
映射	动态迭代步骤

状态可以在状态机中执行各种函数：
- 在您的状态机中执行一些工作（*任务*状态）
- 在状态机分支之间进行选择（*选择*状态）
- 停止正在运行的状态机，返回失败或成功（*失败*状态或*成功*状态）
- 将其输入传递到其输出或者注入一些固定数据（*传递*状态）
- 提供一定时间量的延迟或直至指定时间和日期（*等待*状态）
- 开始创建在状态机中运行的并行分支（*并行*状态）
- 动态地迭代步骤（*映射*状态）

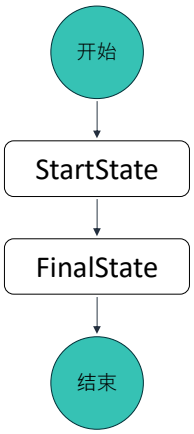
有关状态类型的更多信息，请参阅 AWS 文档中的状态。

Amazon States Language

使用 Amazon States Language
以 JSON 格式定义工作流

```
{
 "Comment": "An example of the ASL.",
 "StartAt": "StartState",
 "States": {
 "StartState": {
 "Type": "Task",
 "Resource": "arn:aws:lambda:us-east…,
 "Next": "FinalState"
 }
 "FinalState": {
 "Type": "Task",
 "Resource": "arn:aws:lambda:us-east…,
 "End": true
 }
 }
}
```

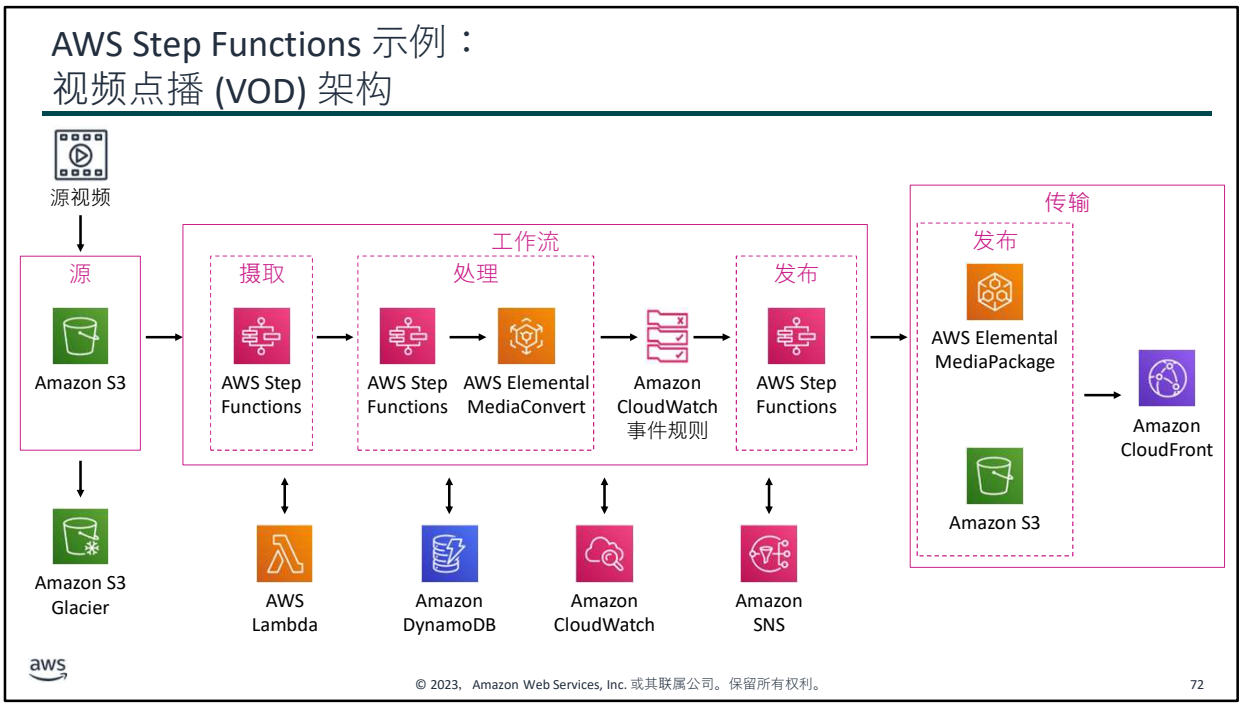
在 Step Functions 控制台
中可视化工作流



您可以使用 Amazon States Language 来定义状态机。Amazon States Language 是一种基于 JSON 的结构化语言。然后，AWS Step Functions 在实时图形视图中表示 JSON 结构。通过这种方式，您可以直接在 Step Functions 控制台中可视化状态机。

在这里，您可以看到具有两种任务状态类型的状态机的示例。

有关 Amazon States Language 的更多信息，请参阅 AWS 文档。



此架构图表显示了视频点播 (VOD) 解决方案中 AWS Step Functions 的示例使用案例。此架构中的另一个关键组件是 AWS Elemental MediaConvert，这是一款具有广播级功能的基于文件的视频转码服务。它使您能够创建视频点播 (VOD) 内容，实现大规模的广播和多屏幕传输。

在该解决方案中，源视频和元数据文件经过摄取和处理，可在各种设备上播放。

- AWS Step Functions 可创建**摄取**、**处理**和**发布**阶跃函数。
- AWS Lambda 函数执行每个步骤的工作并处理错误消息。
- Amazon S3 存储桶可存储源媒体文件和目标媒体文件。
- Amazon CloudWatch 用于日志记录。
- AWS Elemental MediaConvert 通知的 Amazon CloudWatch 事件规则。
- Amazon DynamoDB 表存储通过工作流捕获的数据。
- Amazon SNS 主题发送编码、发布和错误通知。
- 转码的媒体文件已存储，以便通过 Amazon CloudFront 按需交付给用户。

有关此架构的更多信息，请参阅架构概览。

第 7 节要点

- AWS Step Functions 是一项 Web 服务，让您能够使用可视化工作流来协调分布式应用程序和微服务的组件
- 借助 AWS Step Functions，您可以在 AWS 环境中创建自己的状态机并让其自动运行
- AWS Step Functions 为您管理应用程序的逻辑，并实施基本基元，例如顺序或并行分支和超时
- 您可以使用 Amazon States Language 来定义状态机

本模块中这节内容的要点包括：

- AWS Step Functions 是一项 Web 服务，让您能够使用可视化工作流来协调分布式应用程序和微服务的组件
- 借助 AWS Step Functions，您可以在 AWS 环境中创建自己的状态机并让其自动运行
- AWS Step Functions 为您管理应用程序的逻辑，并实施基本基元，例如顺序或并行分支和超时
- 您可以使用 Amazon States Language 来定义状态机

模块 13 – 挑战实验：为咖啡馆实施无服务器架构

现在您将完成模块 13 – 挑战实验：为咖啡馆实施无服务器架构。

业务需求：无服务器环境

Frank 和 Martha 希望获得有关网站上所有订单的每日电子邮件报告。Olivia 建议 Sofía 和 Nikhil 将非业务关键型报告任务与生产 Web 服务器实例分离开。

Sofía 和 Nikhil 希望进一步解耦架构，并将 cron 作业移动到能够很好地扩展且能够降低成本的托管式无服务器环境中。

Frank 和 Martha 希望通过电子邮件获得有关网站上所有订单的每日报告。Frank 希望能够预测需求，以便烘焙正确数量的甜点（减少浪费）。Martha 希望识别咖啡馆业务中的任何模式（分析）。目前，Sofía 已经在 Web 服务器实例上设置了 cron 作业，将这些每日订单报告电子邮件消息发送给 Frank 和 Martha。但是，cron 作业为资源密集型作业，会降低 Web 服务器的性能。

Olivia 建议 Sofía 和 Nikhil 将非业务关键型报告任务分离开。Sofía 和 Nikhil 希望进一步解耦架构，并将 cron 作业移动到能够很好地扩展且能够降低成本的托管式无服务器环境中。

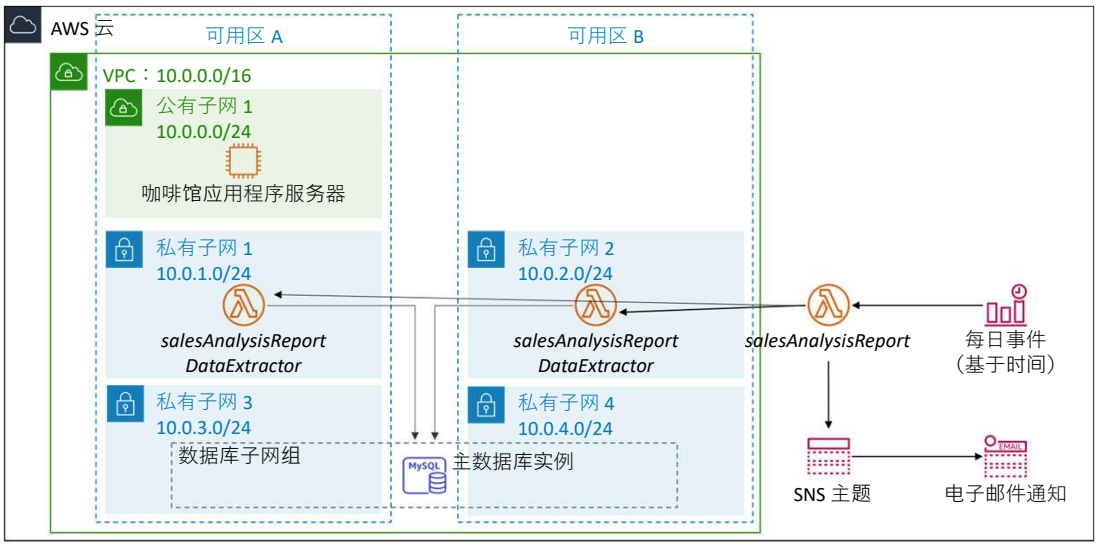
挑战实验：任务

1. 下载源代码
2. 在 VPC 中创建 *DataExtractor* Lambda 函数
3. 创建 *salesAnalysisReport* Lambda 函数
4. 创建 SNS 主题
5. 创建 SNS 主题的电子邮件订阅
6. 测试 *salesAnalysisReport* Lambda 函数
7. 将 Amazon EventBridge 事件设置为每天触发 Lambda 函数

在本挑战实验中，您将完成以下任务：

1. 下载源代码
2. 在 VPC 中创建 *DataExtractor* Lambda 函数
3. 创建 *salesAnalysisReport* Lambda 函数
4. 创建 SNS 主题
5. 创建 SNS 主题的电子邮件订阅
6. 测试 *salesAnalysisReport* Lambda 函数
7. 将 Amazon EventBridge 事件设置为每天触发 Lambda 函数

挑战实验：最终产品



该图总结了您完成实验后将构建的内容。

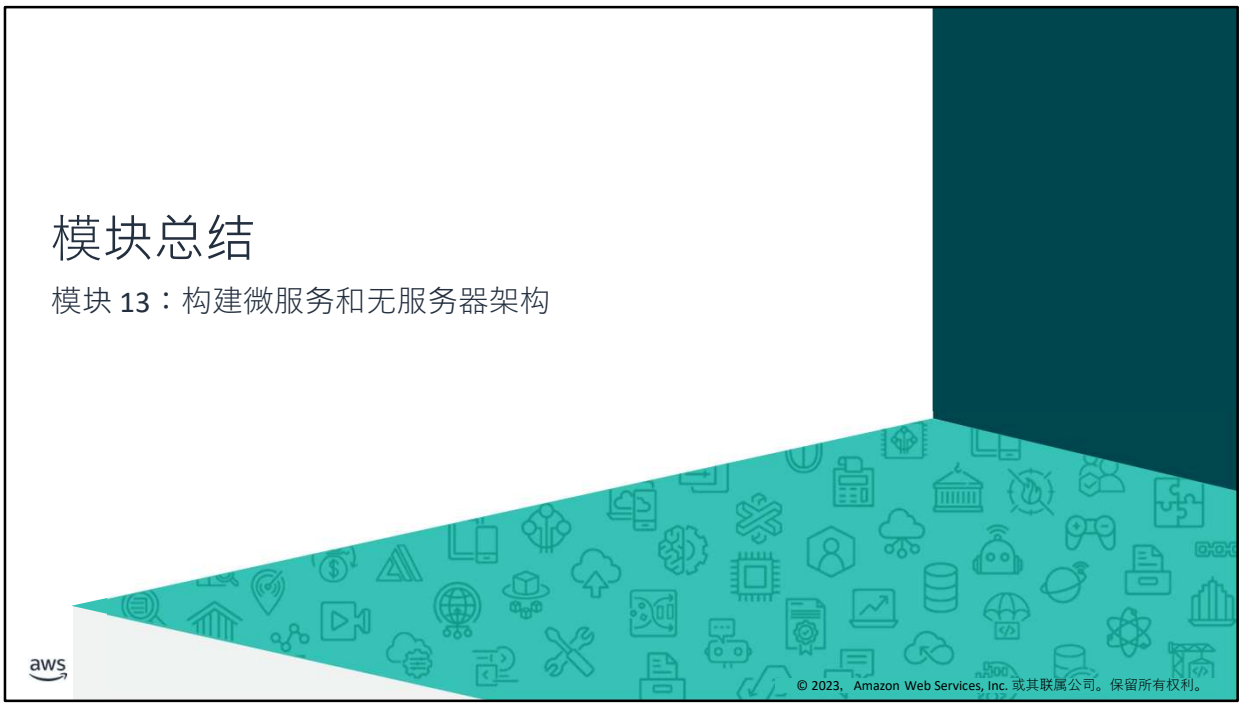
内容说明：三层架构图，其中咖啡馆应用程序服务器位于公有子网 1，销售分析报告数据提取器 Lambda 函数位于私有子网 1 和 2，主数据库实例位于横跨私有子网 3 和 4 的数据库子网组。每日事件会触发销售分析报告 Lambda 函数，该函数会触发数据提取器 Lambda 函数，还会从 SNS 主题发送电子邮件通知。**内容说明结束。**

大约 90 分钟

现在可以开始挑战实验了。

挑战实验总结：要点

完成这个挑战实验之后，您的讲师可能会带您讨论此挑战实验的要点。



现在该复习本模块，并完成最后的知识考核和对实践认证考试问题的讨论了。

模块总结

总的来说，在本模块中，您学习了如何：

- 指出微服务的特性
- 将整体式应用程序重构为微服务，然后使用 Amazon ECS 部署容器化微服务
- 解释无服务器架构的含义
- 使用 AWS Lambda 实施无服务器架构
- 描述 Amazon API Gateway 的通用架构
- 描述 AWS Step Functions 支持的工作流类型

总的来说，在本模块中，您学习了如何:
- 指出微服务的特性
- 将整体式应用程序重构为微服务，然后使用 Amazon ECS 部署容器化微服务
- 解释无服务器架构的含义
- 使用 AWS Lambda 实施无服务器架构
- 描述 Amazon API Gateway 的通用架构
- 描述 AWS Step Functions 支持的工作流类型

完成知识考核

现在该完成本模块的知识考核了。

考试样题

企业托管了 10 项微服务，每项微服务都位于单个 Classic Load Balancer 后面的 Auto Scaling 组中。每个 EC2 实例均以最佳负载运行。

以下哪些措施可以让企业在不影响性能的情况下降低成本？

选项	答案
A	减少每个 Classic Load Balancer 后面的 EC2 实例数量。
B	在 Auto Scaling 组启动配置中更改实例类型。
C	更改 Auto Scaling 组的最大大小，但保留所需容量。
D	将 Classic Load Balancer 替换为单个 Application Load Balancer。

思考答案选项，并根据关键词排除错误选项。

考试样题答案

企业托管了 10 项微服务，每项微服务都位于单个 Classic Load Balancer 后面的 Auto Scaling 组中。每个 EC2 实例均以最佳负载运行。

以下哪些措施可以让企业在不影响性能的情况下降低成本？

正确答案是 D。

问题的要点是在不影响性能的情况下降低成本，并将 Classic Load Balancer 替换为单个 Application Load Balancer。

以下是要识别的关键词：在不影响性能的情况下降低成本，并将 Classic Load Balancer 替换为单个 Application Load Balancer。

正确答案是 D：“将 Classic Load Balancer 替换为单个 Application Load Balancer。”通过排除法，选项 D 是正确的。您可以使用单个 Application Load Balancer 将请求路由到应用程序的所有服务，而不是为每个微服务使用一个 Classic Load Balancer。

错误的答案：

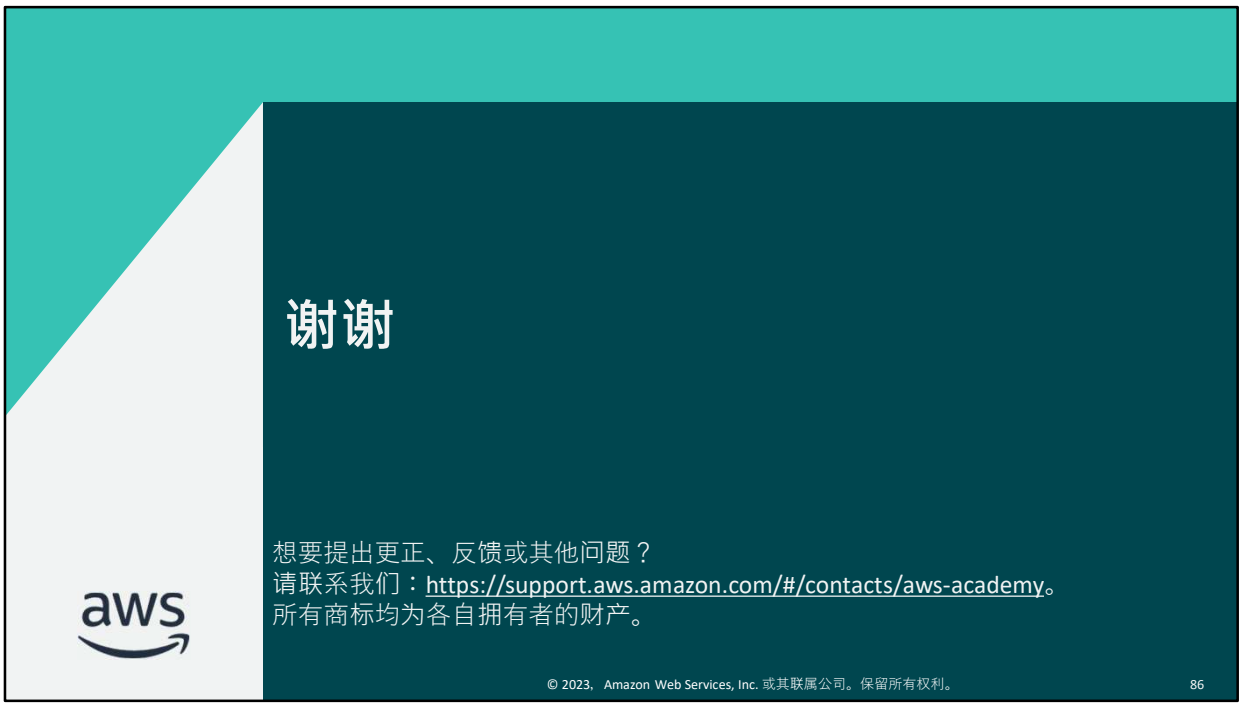
- 选项 A 可以排除 – 因为 EC2 实例以最佳负载运行，因此如果您减少实例的数量，它们将变得超载。
- 选项 B 可以排除 – 选择较大的实例大小并不会降低成本。而如果减小实例大小，EC2 实例就会超载，因为它们已经以最佳负载运行。
- 选项 C 也可以排除 – 如果增加最大大小，而保留所需容量，成本将保持不变或增加。

其他资源

- 将整体式应用程序拆分为微服务项目
- 使用 AWS Lambda 的无服务器架构白皮书
- 使用 Amazon API Gateway 和 AWS Lambda 的 AWS 无服务器多层架构白皮书
- AWS Well-Architected Framework：无服务器应用程序详解白皮书
- 创建和使用 Lambda 函数教程

如果您想进一步了解本模块中涵盖的主题，以下额外资源可能会对您有所帮助：

- 将整体式应用程序拆分为微服务项目
- 使用 AWS Lambda 的无服务器架构白皮书
- 使用 Amazon API Gateway 和 AWS Lambda 的 AWS 无服务器多层架构白皮书
- AWS Well-Architected Framework：无服务器应用程序详解白皮书
- 创建和使用 Lambda 函数教程



感谢您完成本模块的学习。