无服务架构实践手册（Serverless Handbook）

书栈(BookStack.CN)

目　录

致谢

当前文档 《无服务架构实践手册（Serverless Handbook）》 由 进击的皇虫 使用 书栈(BookStack.CN) 进行构建，生成于 2019-11-15。

内容来源：Jimmy Song https://jimmysong.io/serverless-handbook/

文档地址：http://www.bookstack.cn/books/serverless-handbook

书栈官网：http://www.bookstack.cn

书栈开源：https://github.com/TruthHun

目录

前言

什么是 Serverless

Serverless 核心技术

Knative 入门

社区

附录

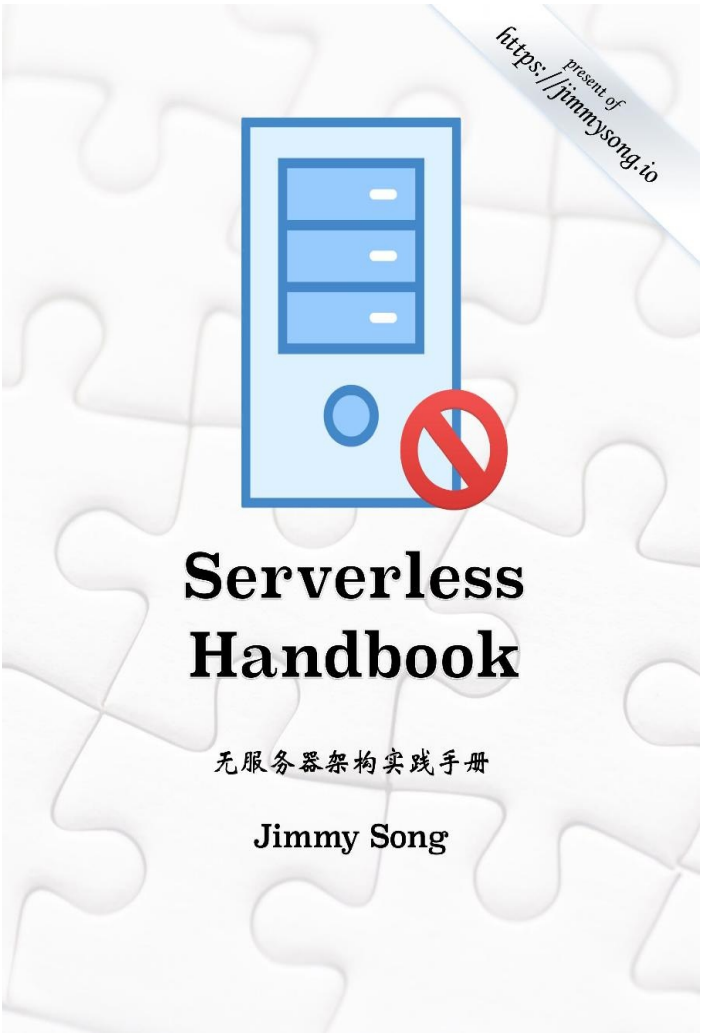
Serverless Handbook—无服务架构实践手册

Serverless（无服务器架构）是指服务端逻辑由开发者实现，应用运行在无状态的计算容器中，由事件触发，完全被第三方管理，其业务层面的状态则存储在数据库或其他介质中。

Serverless 是云原生技术发展的高级阶段，可以使开发者更聚焦在业务逻辑，而减少对基础架构的关注。

关于本书

PASSED



本书是本人学习和实践 Serverless 过程中所整理的资料，主要关注的 Serverless 开源项目是 Knative。

使用方式

您可以通过以下方式使用本书：

- GitHub地址：https://github.com/rootsongjc/serverless-handbook
- GitBook 在线浏览：https://jimmysong.io/serverless-handbook/

License

社区&读者交流

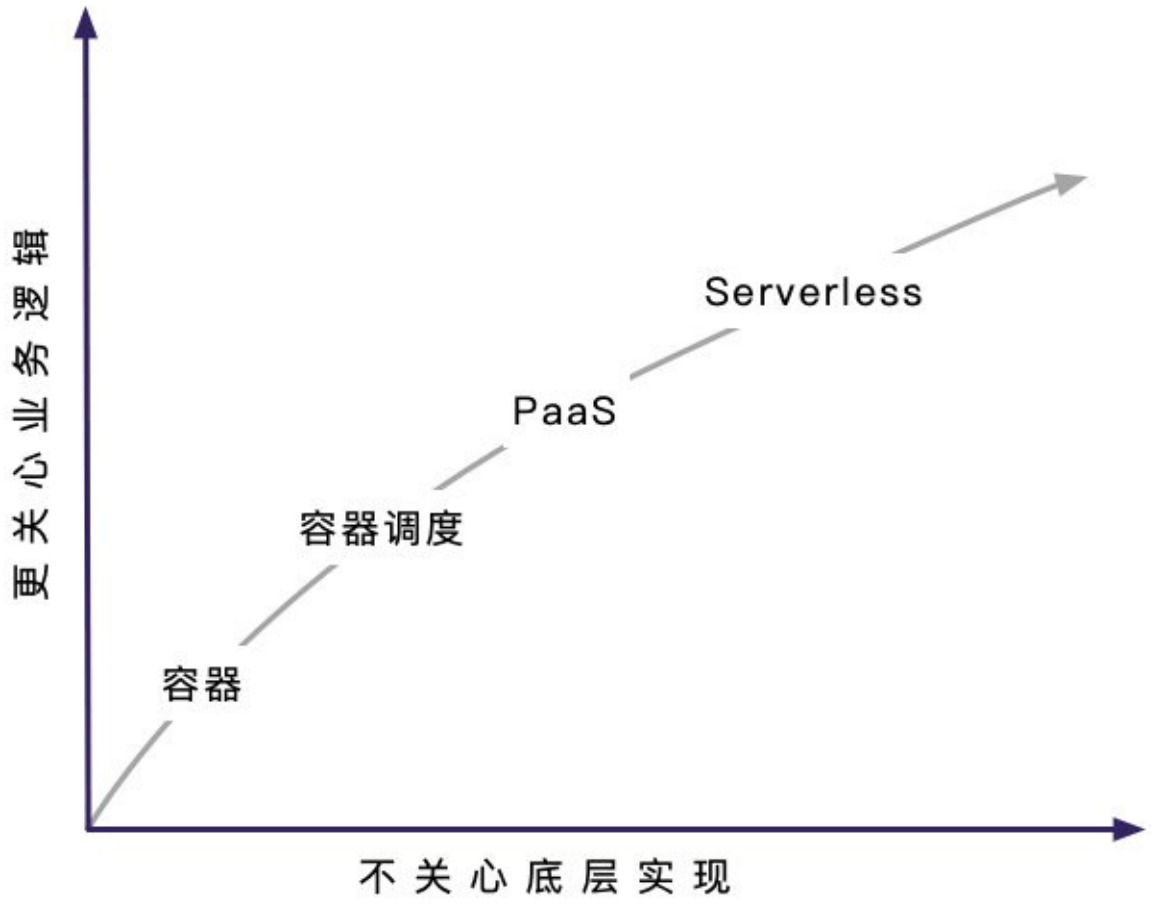
加入 **ServiceMesher** 社区：ServiceMesher 社区是由一群拥有相同价值观和理念的志愿者们共同发起，于 2018 年 4 月正式成立。社区关注领域有：容器、微服务、Service Mesh、Serverless，拥抱开源和云原生。

- Serverless 的定义
- Serverless 的历史
- Serverless 的分类
- Serverless 的使用场景
- Serverless 的优缺点
- 开源的 Serverless 框架

什么是 Serverless？

Serverless（无服务器架构）是指服务端逻辑由开发者实现，运行在无状态的计算容器中，由事件触发，完全被第三方管理，其业务层面的状态则存储在数据库或其他介质中。

Serverless 是云原生技术发展的高级阶段，可以使开发者更聚焦在业务逻辑，而减少对基础设施的关注。



Serverless 的定义

下面将分别从简洁版和进阶版向您展示什么是 Serverless。

简单定义

就像无线互联网实际有的地方也需要用到有线连接一样，无服务器架构仍然在某处有服务器。开发者无需关注服务器，只需关注代码即可。

Serverless（无服务器架构）指的是服务端逻辑由开发者实现，运行在无状态的计算容器中，由事件触发，完全被第三方管理，而业务层面的状态则记录在数据库或存储资源中。

进阶定义

Serverless是由事件（event）驱动（例如 HTTP、pub/sub）的全托管计算服务。用户无需管理服务器等基础设施，只需编写代码和选择触发器（trigger），比如 RPC 请求、定时器等并上传，其余的工作（如实例选择、扩缩容、部署、容灾、监控、日志、安全补丁等）全部由 serverless 系统托管。用户只需要为代码实际运行消耗的资源付费—代码未运行则不产生费用。

Serverless 相对于 serverful，对业务用户强调 noserver（serverless 并不是说没有服务器，只是业务人员无需关注服务器了，代码仍然是运行在真实存在的服务器上）的运维理念，业务人员只需要聚焦业务逻辑代码。

Serverless 相比 serverful，有以下 3 个改变（来自 Berkeley 的总结）：

1. 弱化了存储和计算之间的联系。服务的储存和计算被分开部署和收费，存储不再是服务本身的一部分，而是演变成了独立的云服务，这使得计算变得无状态化，更容易调度和扩缩容，同时也降低了数据丢失的风险。
2. 代码的执行不再需要手动分配资源。不需要为服务的运行指定需要的资源（比如使用几台机器、多大的带宽、多大的磁盘等），只需要提供一份代码，剩下的交由 serverless 平台去处理就行了。当前阶段的实现平台分配资源时还需要用户方提供一些策略，例如单个实例的规格和最大并发数，单实例的最大 CPU 使用率。理想的情况是通过某些学习算法来进行完全自动的自适应分配。
3. 按使用量计费。Serverless按照服务的使用量（调用次数、时长等）计费，而不是像传统的 serverful 服务那样，按照使用的资源（ECS 实例、VM 的规格等）计费。

总结

> No silver bullet. - The Mythical Man-Month

许多年前，我们开发的软件还是 C/S（客户端/服务器）和MVC（模型-试图-控制器）的形式，再后来有了 SOA，最近几年又出现了微服务架构，更新一点的有 Cloud Native（云原生）应用，企业应用从单体架构，到服务化，再到更细粒度的微服务化，应用开发之初就是为了应对互联网的特有的高并发、不间断的特性，需要很高的性能和可扩展性，人们对软件开发的追求孜孜不倦，希望力求在软件开发的复杂度和效率之间达到一个平衡。但可惜的是，No silver bullet！几十年前（1975年）Fred Brooks 就在 *The Mythical Man-Month* 一书中就写到了这句话。那么 Serverlss 会是那颗银弹吗？

云改变了我们对操作系统的认知，原来一个系统的计算资源、存储和网络是可以分离配置的，而且还可以弹性扩展，但是长久以来，我们在开发应用时始终没有摆脱的服务器的束缚（或者说认知），应用必须运行在不论是实体还是虚拟的服务器上，必须经过部署、配置、初始化才可以运行，还需要对服务器和应用进行监控和管理，还需要保证数据的安全性，这些云能够帮我们简化吗？让我们只要关注自己代码的逻辑就好了，其它的东西让云帮我实现就好了。

Serverless 的历史

Serverless 架构是云的自然延伸，为了理解 serverless，我们有必要回顾一下云计算的发展。

IaaS

2006 年 AWS 推出 EC2（Elastic Compute Cloud），作为第一代 IaaS（Infrastructure as a Service），用户可以通过 AWS 快速的申请到计算资源，并在上面部署自己的互联网服务。IaaS 从本质上讲是服务器租赁并提供基础设施外包服务。就比如我们用的水和电一样，我们不会自己去引入自来水和发电，而是直接从自来水公司和电网公司购入，并根据实际使用付费。

EC2 真正对 IT 的改变是硬件的虚拟化（更细粒度的虚拟化），而 EC2 给用户带来了以下五个好处：

- 降低劳动力成本：减少了企业本身雇佣IT人员的成本
- 降低风险：不用再像自己运维物理机那样，担心各种意外风险，EC2 有主机损坏，再申请一个就好了。
- 降低基础设施成本：可以按小时、周、月或者年为周期租用 EC2。
- 扩展性：不必过早的预期基础设施采购，因为通过云厂商可以很快的获取。
- 节约时间成本：快速的获取资源开展业务实验。

以上说了是 IaaS 或者说基础设施外包的好处，当然其中也有弊端，我们将在后面讨论。

以上是 AWS 为代表的公有云 IaaS，还有使用 OpenStack 构建的私有云也能够提供 IaaS 能力。

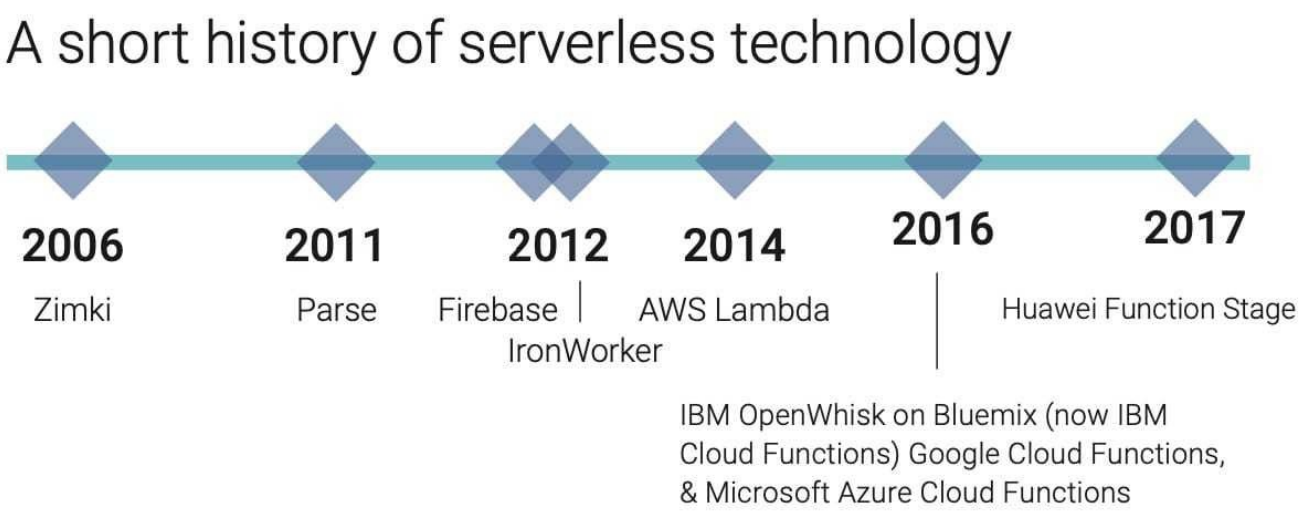
PaaS

PaaS（Platform as a Service）是构建在 IaaS 之上的一种平台服务，提供操作系统安装、监控和服务发现等功能，用户只需要部署自己的应用即可，最早的一代是 Heroku。Heroku 是商业的 PaaS，还有一个开源的 PaaS—Cloud Foundry，用户可以基于它来构建私有 PaaS，如果同时使用公有云和私有云，如果能在两者之间构建一个统一的 PaaS，那就是“混合云”了。

在 PaaS 上最广泛使用的技术就要数 Docker 了，因为使用容器可以很清晰的描述应用程序，并保证环境一致性。管理云上的容器，可以称为是 CaaS（Container as a Service），如 GCE（Google Container Engine）。也可以基于 Kubernetes、Mesos 这类开源软件构件自己的 CaaS，不论是直接在 IaaS 构建还是基于 PaaS。

PaaS 是对软件的一个更高的抽象层次，已经接触到应用程序的运行环境本身，可以由开发者自定义，而不必接触更底层的操作系统。

Serverless



上图来自 CNCF Serverless Whitepaper v1.0。

Serverless 的商业化产品有:

- AWS lambda
- Google Cloud Function
- Microsoft Azure Cloud Functions
- Huawei Function Stage
- 其他

第一个 Serverless 平台

历史上第一个 Serverless 平台可以追溯到 2006 年，名为 Zimki（该公司已倒闭），这个平台提供服务端 JavaScript 应用，虽然他们没有使用Serverless 这个名词，但是他们是第一个“按照实际调用付费”的平台。第一个使用 Serverless 名词的是 iron.io。

总结

Serverless 实际发展已经有 10 年之久，而随着以 Kubernetes 为基础的的云原生应用平台的兴起，serverless 再度成为人民追逐的焦点。

Serverless 的分类

Serverless 不如 IaaS 和 PaaS 那么好理解，因为它通常包含了两个领域 BaaS（Backend as a Service）和 FaaS（Function as a Service）。

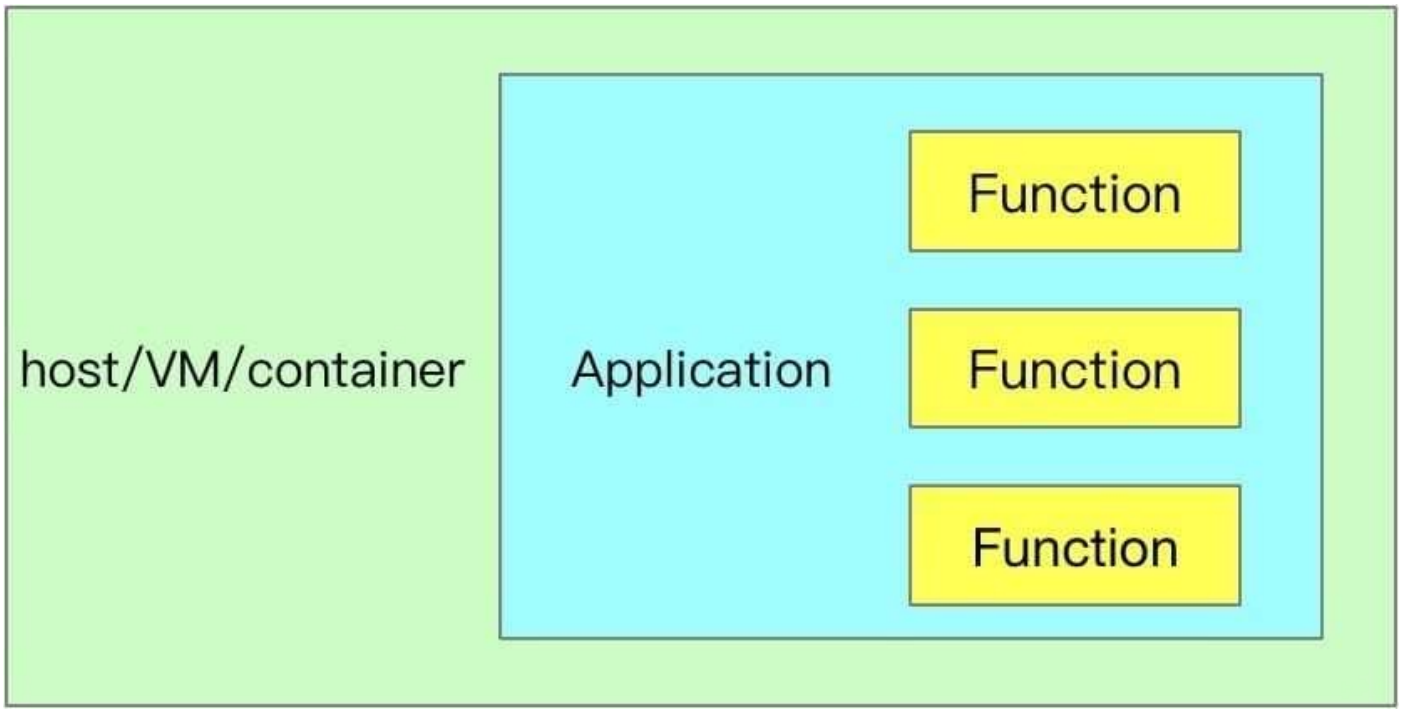
BaaS

BaaS（Backend as a Service）后端即服务，一般是一个个的 API 调用后端或别人已经实现好的程序逻辑，比如身份验证服务 Auth0，这些 BaaS 通常会用来管理数据，还有很多公有云上提供的我们常用的开源软件的商用服务，比如亚马逊的 RDS 可以替代我们自己部署的 MySQL，还有各种其它数据库和存储服务。

FaaS

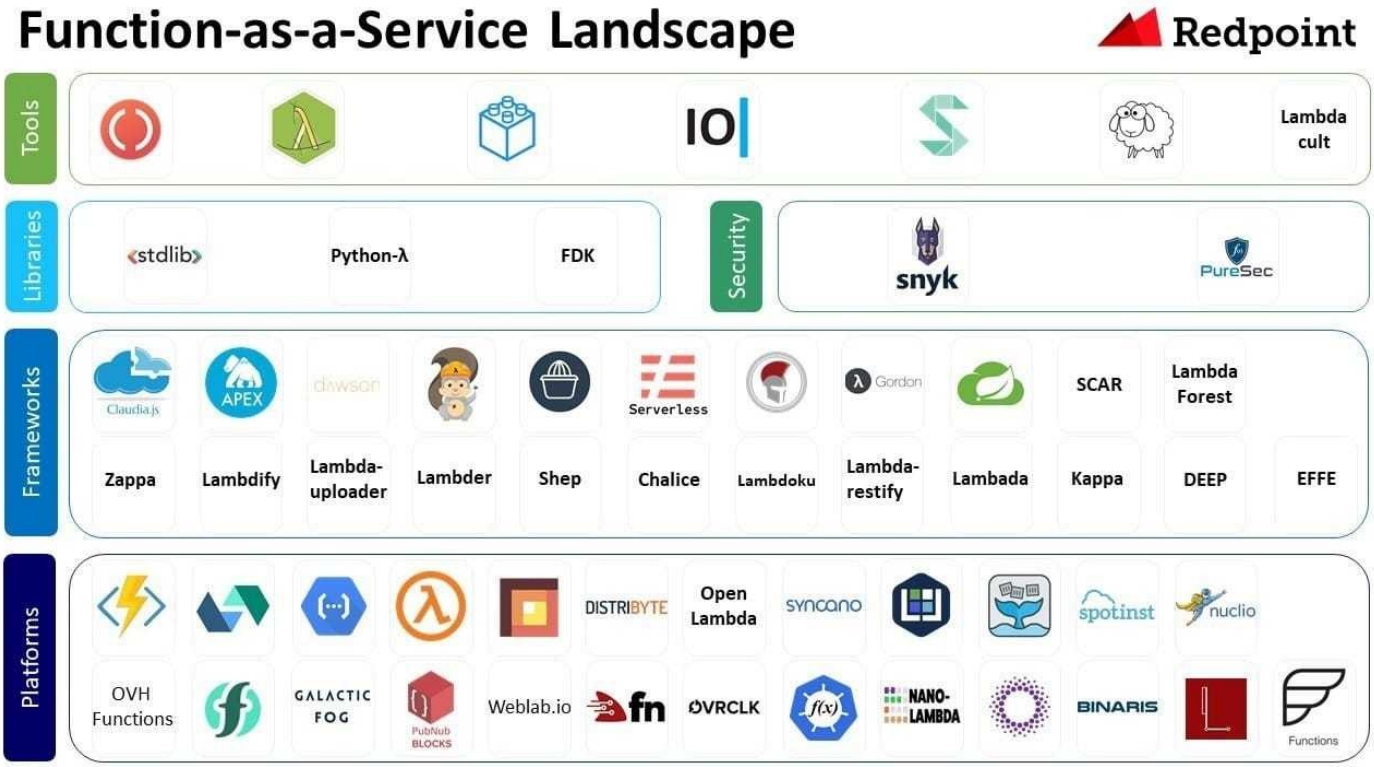
FaaS（Functions as a Service）函数即服务，FaaS 是无服务器计算的一种形式，当前使用最广泛的是 AWS 的 Lambada。

现在当大家讨论 Serverless 的时候首先想到的就是 FaaS，有点甚嚣尘上了。FaaS 本质上是一种事件驱动的由消息触发的服务，FaaS 供应商一般会集成各种同步和异步的事件源，通过订阅这些事件源，可以突发或者定期的触发函数运行。



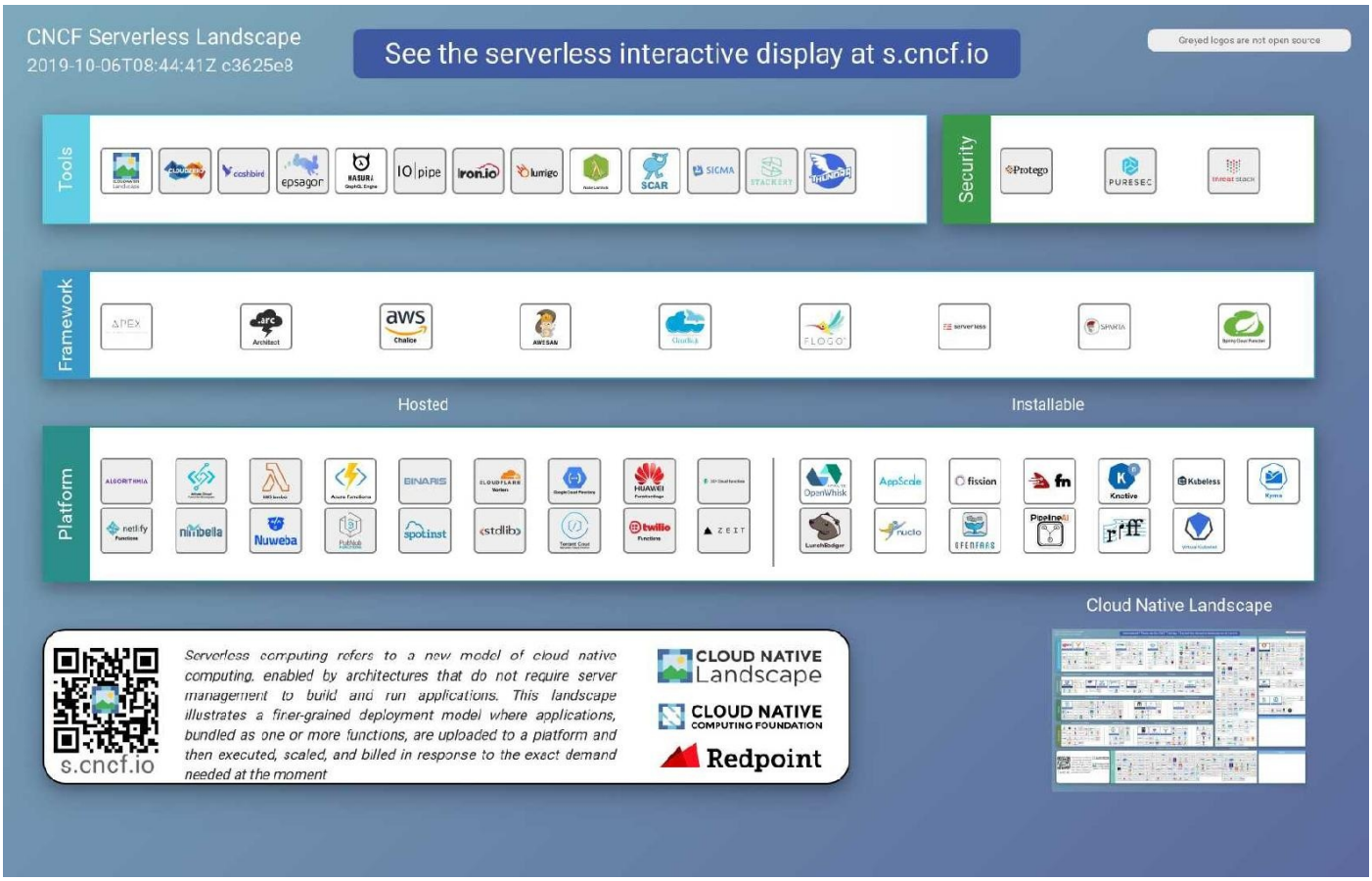
传统的服务器端软件不同是经应用程序部署到拥有操作系统的虚拟机或者容器中，一般需要长时间驻留在操作系统中运行，而 FaaS 是直接将程序部署上到平台上即可，当有事件到来时触发执行，执行完了就可以卸载掉。

下面是 Function-as-a-Service 全景图（图片来自 Github）



Serverless Landscape

CNCF Serverless WG 维护了一份 Serverless Landscape，可以帮助大家快速直观的了解 Serverless 生态系统。



此图的详细信息请访问 https://s.cncf.io 或 https://github.com/cncf/wg-serverless。

Serverless 的使用场景

虽然 Serverless 的应用很广泛，但是其也有局限性，Serverless 比较适合以下场景：

- 异步的并发，组件可独立部署和扩展
- 应对突发或服务使用量不可预测（主要是为了节约成本，因为 Serverless 应用在不运行时不收费）
- 短暂、无状态的应用，对冷启动时间不敏感
- 需要快速开发迭代的业务（因为无需提前申请资源，因此可以加快业务上线速度）

Serverless 的使用场景示例如：

- ETL
- 机器学习及 AI 模型处理
- 图片处理
- IoT 传感器数据分析
- 流处理
- 聊天机器人

CNCF Serverless Whitepaper v1.0 中给出了诸多 Serverless 使用场景的详细描述。

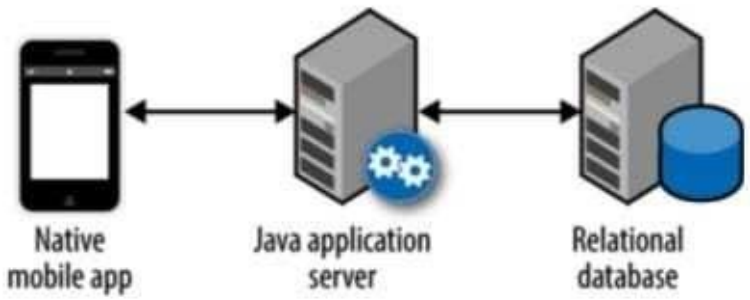
示例

我们以一个游戏应用为例，来说明什么是 serverless 应用。

一款移动端游戏至少包含如下几个特性：

- 移动端友好的用户体验
- 用户管理和权限认证
- 关卡、升级等游戏逻辑，游戏排行，玩家的等级、任务等信息

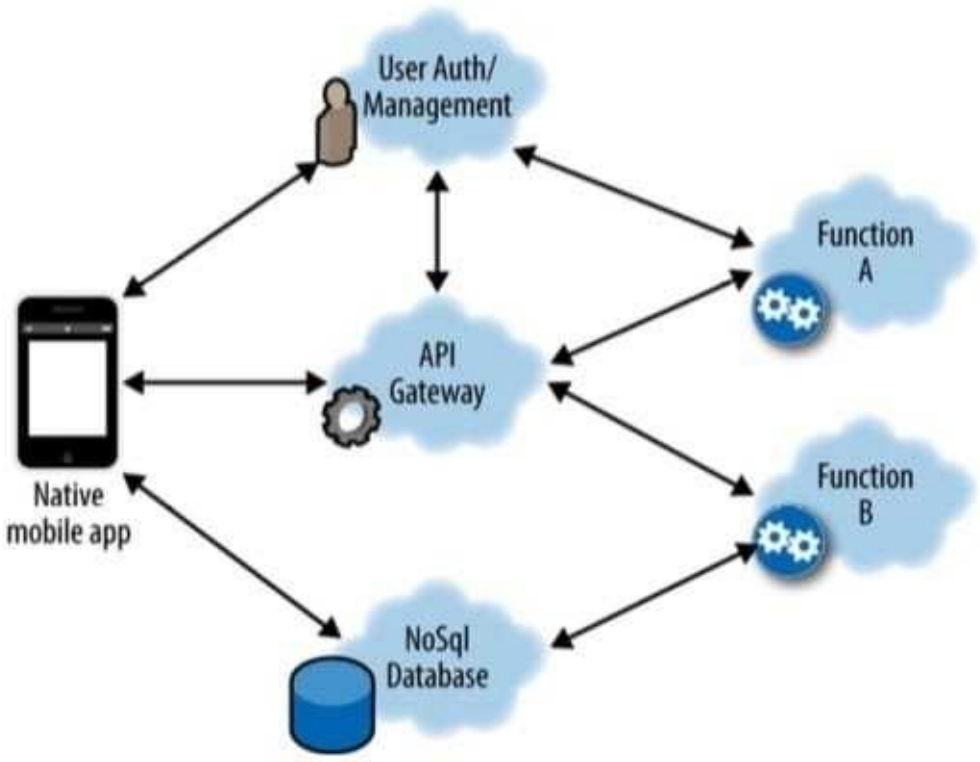
传统的应用程序架构可能是这样的：



- 一个 app 前端，iOS 或者安卓
- 用 Java 写的后端，使用 JBoss 或者 Tomcat 做 server 运行
- 使用关系型数据库存储用户数据，如 MySQL

这样的架构可以让前端十分轻便，不需要做什么应用逻辑，只是负责渲染用户界面，将请求通过 HTTP 发送给后端，而所有的数据操作都是有由后端的 Java 程序来完成的。

这样的架构开发起来比较容易，但是维护起来确十分复杂，前端开发、后端的开发都需要十分专业的人员、环境的配置，还要有人专门维护数据库、应用的更新和升级。



而在 serverless 架构中，我们不再需要在服务器端代码中存储任何会话状态，而是直接将它们存储在 NoSQL 中，这样将使应用程序无状态，有助于弹性扩展。前端可以直接利用 BaaS 而减少后端的编码需求，这样架构的本质上是减少了应用程序开发的人力成本，降低了自己维护基础设施的风险，而且利用云的能力更便于扩展和快速迭代。

Serverless 的优缺点

Serverless 架构不是一劳永逸的，是否使用 Serverless 也需要根据实际需要来选择，请先考虑 Serverless 的优缺点。

Serverless 架构的优点

> 今天大多数公司在开发应用程序并将其部署在服务器上的时候，无论是选择公有云还是私有的数据中心，都需要提前了解究竟需要多少台服务器、多大容量的存储和数据库的功能等。并需要部署运行应用程序和依赖的软件到基础设施之上。假设我们不想在这些细节上花费精力，是否有一种简单的架构模型能够满足我们这种想法？这个答案已经存在，这就是今天软件架构世界中新鲜但是很热门的一个话题——Serverless（无服务器）架构。
>
> ——AWS 费良宏

- 降低运营成本：

Serverless 是非常简单的外包解决方案。它可以让您委托服务提供商管理服务器、数据库和应用程序甚至逻辑，否则您就不得不自己来维护。由于这个服务使用者的数量会非常庞大，于是就会产生规模经济效应。在降低成本上包含了两个方面，即基础设施的成本和人员（运营/开发）的成本。

- 降低开发成本：

IaaS 和 PaaS 存在的前提是，服务器和操作系统管理可以商品化。Serverless 作为另一种服务的结果是整个应用程序组件被商品化。

- 扩展能力：

Serverless 架构一个显而易见的优点即“横向扩展是完全自动的、有弹性的、且由服务提供者所管理”。从基本的基础设施方面受益最大的好处是，您只需支付您所需要的计算能力。

- 更简单的管理：

Serverless 架构明显比其他架构更简单。更少的组件，就意味着您的管理开销会更少。

- “绿色”的计算：

按照《福布斯》杂志的统计，在商业和企业数据中心的典型服务器仅提供 5%～15% 的平均最大处理能力的输出。这无疑是一种资源的巨大浪费。随着Serverless架构的出现，让服务提供商提供我们的计算能力最大限度满足实时需求。这将使我们更有效地利用计算资源。

在上面我们提到了使用 IaaS给 我们带来了五点好处，FaaS 当然也包括了这些好处，但是它给我们带来的最大的好处就是多快好省。减少从概念原型到实施的等待时间，比自己维护服务更省钱。

降低人力成本

不需要再自己维护服务器，操心服务器的各种性能指标和资源利用率，而是关心应用程序本身的状态和逻辑。而且 serverless 应用本身的部署也十分容易，我们只要上传基本的代码但愿，例如 Javascript 或 Python 的源代码的 zip 文件，以及基于JVM的语言的纯 JAR 文件。不需使用 Puppet、Chef、Ansible 或 Docker 来进行配置管理，降低了运维成本。同时，对于运维来说，也不再需要监控那些更底层的如磁盘使用量、CPU 使用率等底层和长期的指标信息，而是监控应用程序本身的度量，这将更加直观和有效。

在此看来有人可能会提出 “NoOps” 的说法，其实这是不存在的，只要有应用存在的一天就会有 Ops，只是人员的角色会有所转变，部署将变得更加自动化，监控将更加面向应用程序本身，更底层的运维依然需要专业的人员去做。

降低风险

对于组件越多越复杂的系统，出故障的风险就越大。我们使用 BaaS 或 FaaS 将它们外包出去，让专业人员来处理这些故障，有时候比我们自己来修复更可靠，利用专业人员的知识来降低停机的风险，缩短故障修复的时间，让我们的系统稳定性更高。

减少资源开销

我们在申请主机资源一般会评估一个峰值最大开销来申请资源，往往导致过度的配置，这意味着即使在主机闲置的状态下也要始终支付峰值容量的开销。对于某些应用来说这是不得已的做法，比如数据库这种很难扩展的应用，而对于普通应用这就显得不太合理了，虽然我们都觉得即使浪费了资源也比当峰值到来时应用程序因为资源不足而挂掉好。

解决这个问题最好的办法就是，不计划到底需要使用多少资源，而是根据实际需要来请求资源，当然前提必须是整个资源池是充足的（公有云显然更适合）。根据使用时间来付费，根据每次申请的计算资源来付费，让计费的粒度更小，将更有利于降低资源的开销。这是对应用程序本身的优化，例如让每次请求耗时更短，让每次消耗的资源更少将能够显著节省成本。

增加缩放的灵活性

以 AWS Lamba 为例，当平台接收到第一个触发函数的事件时，它将启动一个容器来运行你的代码。如果此时收到了新的事件，而第一个容器仍在处理上一个事件，平台将启动第二个代码实例来处理第二个事件。AWS lambad 的这种自动的零管理水平缩放，将持续到有足够的代码实例来处理所有的工作负载。

但是，AWS 仍然只会向您收取代码的执行时间，无论它需要启动多少个容器实例要满足你的负载请求。例如，假设所有事件的总执行时间是相同的，在一个容器中按顺序调用Lambda 100 次与在 100 个不同容器中同时调用 100 次 Lambda 的成本是 一样的。当然 AWS Lambada 也不会无限制的扩展实例个数，如果有人对你发起了 DDos 攻击怎么办，那么不就会产生高昂的成本吗？AWS 是有默认限制的，默认执行 Lambada 函数最大并发数是 1000。

缩短创新周期

小团队的开发人员正可以在几天之内从头开始开发应用程序并部署到生产。使用短而简单的函数和事件来粘合强大的驱动数据存储和服务的 API。完成的应用程序具有高度可用性和可扩展性，利用率高，成本低，部署速度快。

以 Docker 为代表的容器技术仅仅是缩短了应用程序的迭代周期，而 serverless 技术是直接缩短了创新周期，从概念到最小可行性部署的时间，让初级开发人员也能在很短的时间内完成以前通常要经验丰富的工程师才能完成的项目。

Serverless架构的缺点

我们知道没有十全十美的技术，在说了 serverless 的那么多优势之后，我们再来探讨以下 serverless 的劣势，或者说局限性和适用场景。

状态管理

要想实现自由的缩放，无状态是必须的，而对于有状态的服务，使用 serverless 这就丧失了灵活性，有状态服务需要与存储交互就不可避免的增加了延迟和复杂性。

延迟

应用程序中不同组件的访问延迟是一个大问题，我们可以通过使用专有的网络协议、RPC 调用、数据格式来优化，或者是将实例放在同一个机架内或同一个主机实例上来优化以减少延迟。

而 serverless 应用程序是高度分布式、低耦合的，这就意味着延迟将始终是一个问题，单纯使用 serverless 的应用程序是不太现实的。

本地测试

Serverless 应用的本地测试困难是一个很棘手的问题。虽然可以在测试环境下使用各种数据库和消息队列来模拟生产环境，但是对于无服务应用的集成或者端到端测试尤其困难，很难在本地模拟应用程序的各种连接，并与性能和缩放的特性结合起来测试，并且 serverless 应用本身也是分布式的，简单的将无数的 FaaS 和 BaaS 组件粘合起来也是有挑战性的。

总结

Karl Marx说的好，生产力决定生产关系，云计算的概念层出不穷，其本质上还是对生产关系和生产力的配置与优化，生产者抛开场景意味追求高大上的技术将譬如“大炮打蚊子”，小题大做，鼓励大家为了满足大家的好奇心进行折腾，毕竟那么多科学发现和重大发明都是因为折腾出来的，不想要一匹跑的更快的马，而是发明汽车的福特，捣鼓炸药的诺贝尔，种豌豆的孟德尔……同时还是要考虑将技术产业化（或许能改变生产关系），提高生产力。

开源的 Serverless 框架

Kubernetes 的蓬勃发展由催生了一系列以它为基础的 Serverless 应用，目前开源的 Serverless 框架大多以 Kubernetes 为基础。

- dispatch - Dispatch is a framework for deploying and managing serverless style applications.
- faas-netes - Enable Kubernetes as a backend for Functions as a Service (OpenFaaS) https://github.com/alexellis/faas
- firecamp - Serverless Platform for the stateful services
- fission - Fast Serverless Functions for Kubernetes http://fission.io
- fn - The container native, cloud agnostic serverless platform. http://fnproject.io
- funktion - a CLI tool for working with funktion https://funktion.fabric8.io/
- fx - Poor man’s serverless framework based on Docker, Function as a Service with painless.
- gloo - The Function Gateway built on top of Envoy.
- ironfunctions - IronFunctions - the serverless microservices platform. http://iron.io
- knative - Kubernetes-based platform to build, deploy, and manage modern serverless workloads.
- knative-lambda-runtime - Running AWS Lambda Functions on Knative/Kubernetes Clusters https://triggermesh.com
- kubeless - Kubernetes Native Serverless Framework http://kubeless.io
- nuclio - High-Performance Serverless event and data processing platform.
- openfaas - OpenFaaS - Serverless Functions Made Simple for Docker & Kubernetes https://blog.alexellis.io/introducing-functions-as-a-service/
- openwhisk - Apache OpenWhisk (Incubating) is a serverless, open source cloud platform that executes functions in response to events at any scale.
- riff - riff is for functions https://projectriff.io
- serverless - Serverless Framework – Build web, mobile and IoT applications with serverless architectures using AWS Lambda, Azure Functions, Google CloudFunctions & more! – https://serverless.com
- spec - CloudEvents Specification https://cloudevents.io
- thanos - Highly available Prometheus setup with long term storage capabilities.

以上列表来自 https://jimmysong.io/awesome-cloud-native/#serverless。

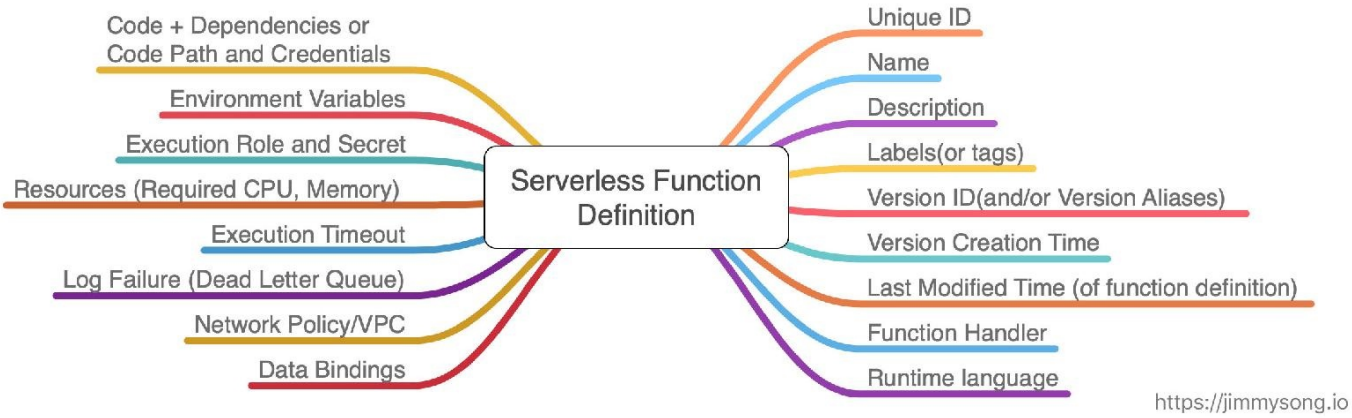
Serverless 核心技术概述

Serverless 是由事件驱动的全托管计算服务，它的核心技术包括：

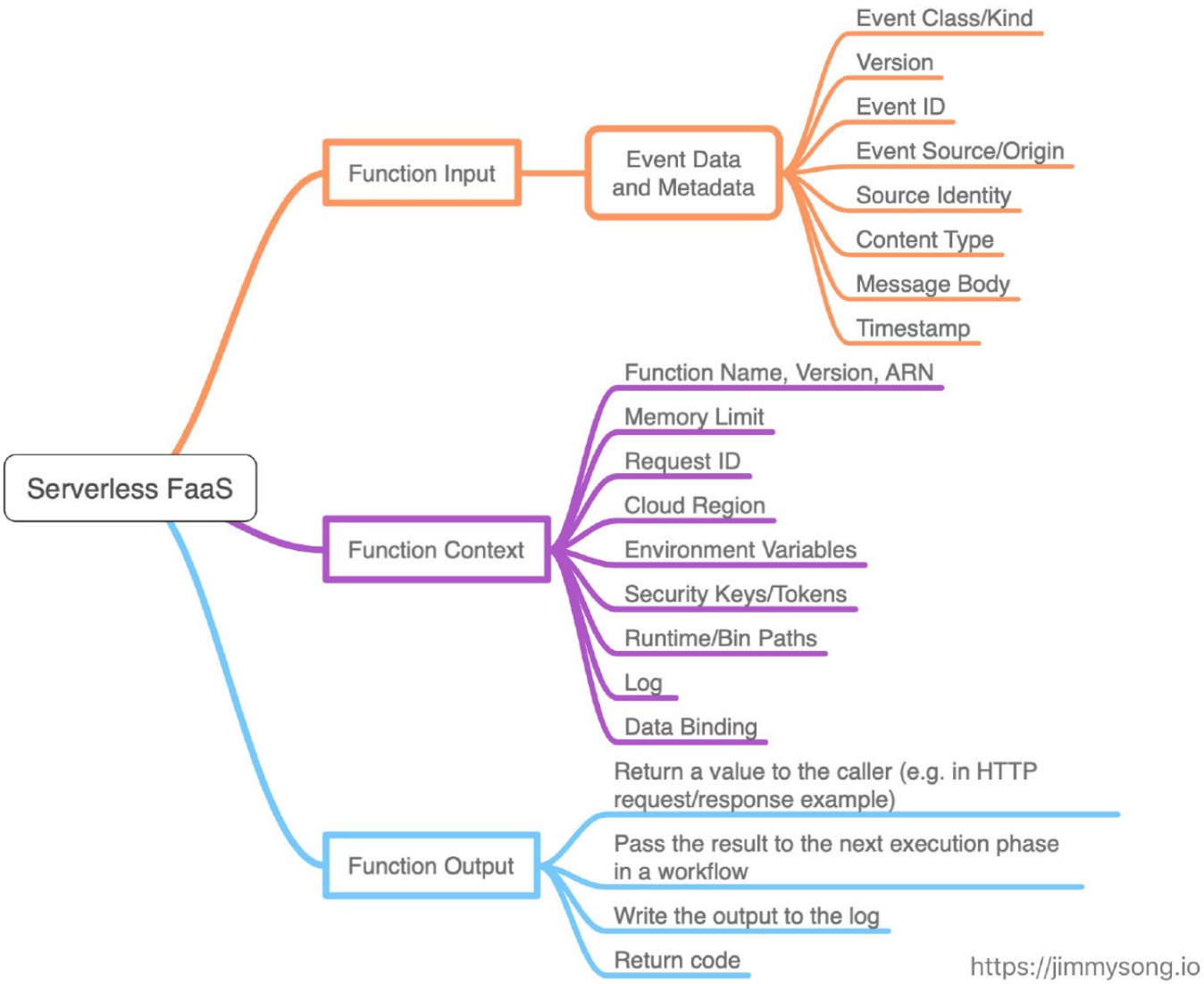
- 函数的规范定义
- 函数部署流水线
- Workflow 设置
- 0-m-n 扩缩容
- 快速冷启动

函数计算

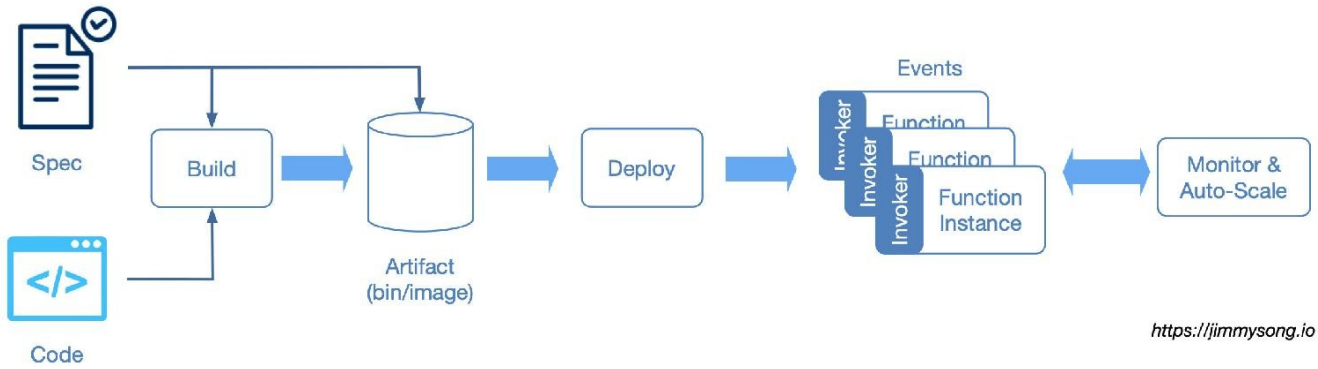
下图是 Serverless 中的（FaaS）函数定义，从图中可以看出与容器、12 要素及 Kubernetes 的运行时设计十分契合。



下图是 FaaS 中函数输入、context 及输出。



当函数创建完成后通过函数部署流水线部署运行。

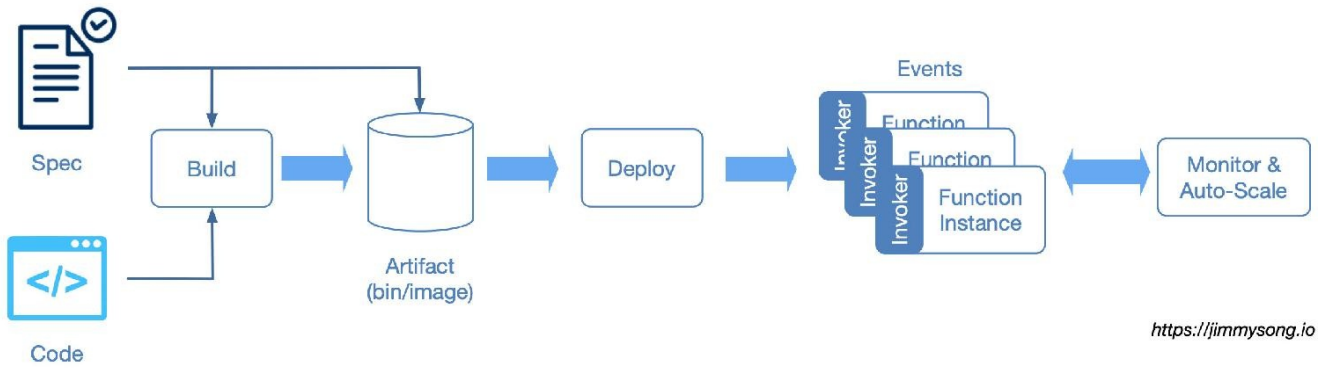


以上图片根据 CNCF Serverless Whitepaper v1.0 绘制。

函数生命周期

本节描述函数生命周期及如何使用 Serverless 框架/运行时来管理函数生命周期。

函数部署流水线

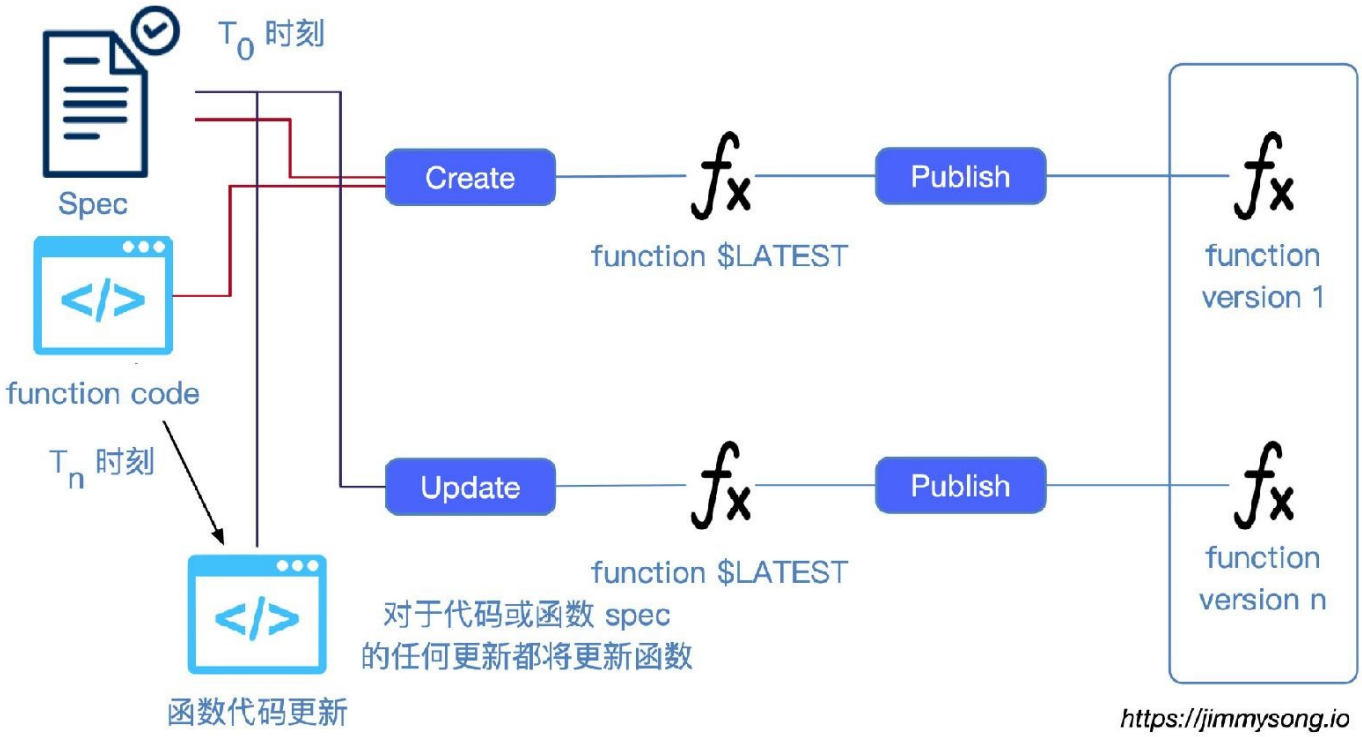


函数的生命周期始于编写代码并提供规范和元数据（请参见下面的函数定义），“builder” 实体将获取代码和规范，进行编译，然后将其转换为工件（代码二进制文件、程序包或容器、镜像）。然后将工件部署到具有控制器实体的集群上，该控制器实体负责根据事件的流量和/或实例上的负载来扩展函数实例的数量。

函数操作

Serverless 框架使用以下动作（Action）控制函数的生命周期：

- 创建：创建新函数，包括其规格（spec）和代码
- 发布：创建可以在集群上部署的函数的新版本
- 更新别名/标签（版本）：更新版本别名
- 执行/调用：不通过事件源调用特定版本
- 事件源关联：将函数的特定版本与事件源连接
- 获取：返回函数元数据和规格
- 更新：修改函数的最新版本
- 删除：删除函数，可以删除特定版本或所有版本的函数
- 列表：显示函数及其元数据的列表
- 获取统计信息：返回有关函数运行时使用情况的统计信息
- 获取日志：返回函数生成的日志



创建函数时需要提供函数的元数据（如稍后在函数规范中所述），创建的函数可能会被编译和发布。稍后可以启动（start）、禁用（disable）和启用（enable）函数。函数部署需要能够支持以下用例：

- 事件流式传输，在这种情况下，可能总是有事件在队列中，但是可能需要通过明确的请求来暂停/恢复处理。
- 热启动（**Warm Startup**）：函数随时都有最少实例数量，由于函数已部署并准备为事件提供服务，因此当收到“第一个”事件时可以热启动（与冷启动相反， 通过“传入”事件在第一次调用时部署函数）。

用户可以发布函数，这将创建一个新版本（“latest” 版本的副本），该发布的版本可能被标记/标签（tag/label）或具有别名（aliases），请参见下文。

用户可能想要直接执行/调用函数（绕过事件源或 API 网关）以进行调试和开发。用户可以指定调用参数，例如所需的版本、同步/异步操作、详细（Verbosity）程度等。

用户可能希望获取函数统计信息（例如，调用次数、平均运行时间、平均延迟、失败、重试等），统计信息可以是当前指标值或值的时间序列（例如，存储在 Prometheus 或云提供商设施中、例如 AWS Cloud Watch）。

用户可能想检索函数日志数据。这可以通过严重性级别和/或时间范围和/或内容来过滤。日志数据是按函数列出的，其中包括以下事件：函数的创建和删除、显式错误、警告或调试消息，以及函数的 Stdout 或 Stderr。最好是每次调用都具有一条日志条目，或者将日志条目与特定调用相关联（以简化对函数执行流的跟踪）。

函数版本控制和别名

一个函数可能具有多个版本，使用户能够运行不同级别的代码，例如 beta/production、A/B测试等。使用版本控制时，默认情况下函数版本为 “latest”，“latest” 版本可以进行更新和修改，可能会在每次更改时触发新的构建过程。

如果用户想要冻结一个版本可以使用发布操作，该操作将创建一个具有潜在标签或别名（例如，“beta”、“production”）的新版本，以配置事件源，事件或 API 调用可以被路由到特定的函数版本。非最新的函

数版本是不可变的（它们的代码以及所有或某些函数规范），并且一旦发布就不能更改。函数不能“未发布”，而应将其删除。

请注意，当前的大多数实现都不允许函数 branch/fork（更新旧版本代码），因为这会使实现和用法变得复杂，但是将来可能需要这样做。

当同一函数有多个版本时，用户必须指定要操作的函数版本以及如何在不同版本之间划分事件流量。例如，用户可以决定路由 90% 的事件流量到稳定版本，10% 的流量到 Beta 版（又称“ canary update” ）。可以通过指定确切版本或通过指定版本别名来实现。版本别名通常将引用特定的函数版本。

用户创建或更新函数时，它可能会根据变更的性质来驱动新的构建和部署。

事件源到函数关联

由于事件源触发事件而调用函数。函数和事件源之间存在一个 n:m 映射。每个事件源都可以用于调用多个函数，而一个函数可以由多个事件源触发。事件源可以映射到函数的特定版本或函数的别名，后者提供了一种用于更改函数并部署新版本的方法，而无需更改事件关联。事件源还可以定义为使用同一函数的不同版本，并定义应为每个函数分配多少流量。

创建函数后或稍后的某个时间，需要关联事件源，该事件源应触发作为该事件的函数调用。这需要一系列动作（action）和方法（method），例如:

- 创建事件源关联
- 更新事件源关联
- 列出事件源关联

事件源

不同类型的事件源包括:

- 事件和消息传递服务，例如: RabbitMQ、MQTT、SES、SNS、Google Pub / Sub
- 存储服务，例如: S3、DynamoDB、Kinesis、Cognito、Google Cloud Storage，Azure Blob、iguazio V3IO（对象/流/数据库）
- 端点服务，例如: 物联网、HTTP网关、移动设备、Alexa、Google Cloud Endpoint
- 配置存储库，例如: Git、CodeCommit
- 使用特定于语言的 SDK 的用户应用程序
- SchEnable 定期调用函数

尽管每个事件提供的数据在不同事件源之间可能会有所不同，但事件结构应该具有通用性，能够封装有关事件源的特定信息（详细信息见事件数据和元数据）。

函数要求

下面的列表根据当前的技术水平描述了函数和 Serverless 运行时应满足的一组通用要求:

- 函数必须与不同事件类的基础实现分离
- 可以从多个事件源调用函数

- 无需为每个调用方法使用不同的函数
- 事件源可以调用多个函数
- 函数可能需要一种与基础平台服务进行持久绑定的机制，可能是跨函数调用。函数的寿命可能很短，但是如果需要在每次调用时都进行引导，那么引导可能会很昂贵，例如在日志记录、连接、安装外部数据源的情况下。
- 同一个应用程序中每个函数可以使用不同的语言编写
- 函数运行时应尽可能减少事件序列化和反序列化的开销（例如，使用本地语言结构或有效的编码方案）

工作流相关要求

- 函数可以作为工作流的一部分被调用，一个函数的结果可以作为另一个函数的触发
- 可以由事件或 “and/or 事件组合”触发函数
- 一个事件可能触发按顺序或并行执行的多个函数
- “and/or 事件组合”可能触发顺序运行、并行运行或分支运行的 m 个函数
- 在工作流的中间，可能会收到不同的事件或函数结果，这将触发分支切换到不同的函数
- 函数的部分或全部结果需要作为输入传递给另一个函数
- 函数可能需要一种与基础平台服务进行持久绑定的机制，这可能是跨函数调用或函数寿命很短

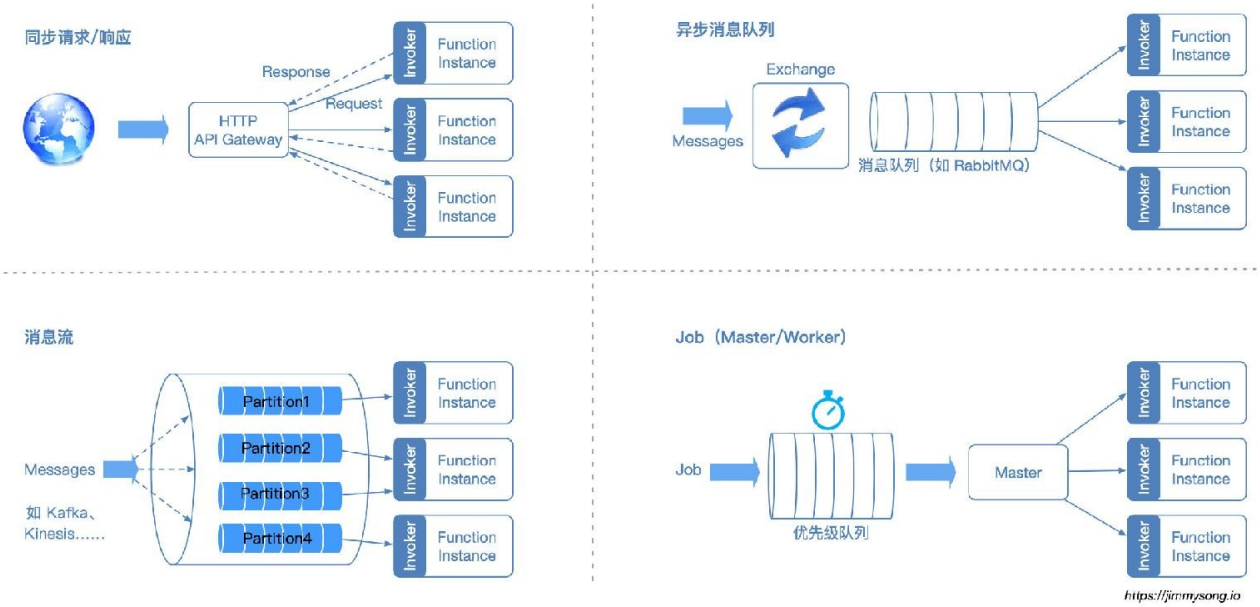
函数调用类型

可以根据不同的用例从不同的事件源调用函数，例如：

1. 同步请求（Req/Rep），例如 HTTP 请求、gRPC调用
    - 客户端发出请求并等待立即响应。这是一个阻塞调用。
2. 异步消息队列请求（Pub/Sub），例如 RabbitMQ、AWS SNS、MQTT、电子邮件，对象（S3）更改、计划事件（如 CRON 作业）
    - 消息发布到交换场所并分发给订阅者
    - 没有严格的消息顺序。恰好一次（Exactly once）处理

3. 消息/记录流：例如 Kafka、AWS Kinesis、AWS DynamoDB 流、数据库 CDC

    - 一组有序的消息/记录（必须按顺序处理）
    - 通常，将流分片到多个分区/分片（分片消费者）每个分片分配给一个 worker
    - 可以从消息，数据库更新（日志）或文件（例如CSV、Json、Parquet）产生流
    - 事件可以推入（Push）到函数运行时或由函数运行时拉取（Pull）
    - 批处理作业，例如 ETL 工作、分布式深度学习、HPC 模拟
    - 作业被调度或提交到到队列中，并在运行时使用多个并行的函数实例进行处理，每个实例处理工作集（一个任务）的一个或多个部分
    - 当所有并行 worker 成功完成所有计算任务时，作业完成


同步请求/响应
Response
Request
HTTP
API Gateway
Invoker
Function
Instance
异步消息队列
Exchange
Messages
消息队列（如 RabbitMQ）
消息流
Partition1
Partition2
Partition3
Partition4
Messages
如 Kafka、
Kinesis.....
Job（Master/Worker）
Job
优先级队列
Master
https://jimmysong.io

函数代码

函数代码、依赖项和/或二进制文件可以驻留在外部存储库（例如 S3 对象存储桶或 Git 存储库）中，或由用户直接提供。如果代码在外部存储库中，则用户需要指定路径和凭据。

Serverless 框架还允许用户 watch 代码存储库中的更改（例如，使 Webhook），并在每次提交时自动构建函数镜像/二进制文件。

函数可能依赖于外部库或二进制文件，这些需要由用户提供，包括描述其构建过程的方式（例如，使用 Dockerfile、Zip）。

另外，可以通过一些二进制打包（例如 OCI 镜像）将函数提供给框架。

函数定义

Serverless 函数定义可能包含以下规范和元数据，该函数定义是特定于版本的：

- 唯一 ID
- 名称
- 说明
- Label（或 tag）
- 版本ID（和/或版本别名）
- 版本创建时间
- 上次修改时间（函数定义）
- 函数处理程序
- 运行时语言
- 代码 + 依赖关系或代码路径和凭据
- 环境变量
- 执行角色和 secret
- 资源（所需的 CPU、内存）
- 执行超时
- 日志记录失败（Dead Letter Queue，）
- 网络策略/VPC
- 数据绑定

> **Dead Letter Queue**
>
> 中文译作“死信队列”，在消息队列中，死信队列是一种服务实现，用于存储满足以下一个或多个条件的消息： 发送到不存在的队列的消息。 超出队列长度限制。 超出了邮件长度限制。 消息被另一个队列交换拒绝。 消息达到阈值读取计数器编号，因为它没有被消耗。有时这被称为“退出队列”。 存储这些消息的死信队列允许开发人员查找常见模式和潜在的软件问题。（摘自维基百科）

元数据详细信息

函数框架可能包括以下函数元数据：

- 版本：每个函数版本应具有唯一的标识符，此外，可以使用一个或多个别名（例

如“latest”、“production”、“beta”）来标记版本。API 网关和事件源会将流量/事件路由到特定的函数版本。

- 环境变量：用户可以指定在运行时将提供给函数的环境变量。环境变量也可以从机密和加密的内容派生，也可以从平台变量派生（例如，像Kubernetes EnvVar定义）。环境变量使开发人员能够控制函数行为和参数，而无需修改代码和/或重建函数，从而获得更好的开发人员体验和函数重用。
- 执行角色：该函数应在特定的用户或角色身份下运行，以授予和审核其对平台资源的访问权限。
- 资源：定义所需或最大的硬件资源，例如函数使用的内存和CPU。
- 超时：指定函数调用在平台终止之前可以运行的最长时间。
- 故障日志（死信队列）：队列或流的路径，它将存储具有适当详细信息的失败函数执行列表。
- 网络策略：分配给函数的网络域和策略（函数与外部服务/资源进行通信）。
- 执行语义：指定应如何执行函数（例如，每个事件至少执行一次，最多执行一次，恰好一次）。

数据绑定

某些 Serverless 框架允许用户指定函数使用的输入/输出数据资源，这使开发变得更简单，性能更高（在执行期间保留数据连接，可以预取数据等）以及更好的安全性（数据资源凭据是上下文的一部分，而不是代码）。

绑定数据可以采用文件、对象、记录、消息等形式，函数说明可以包括一组数据绑定定义，每个定义都指定数据资源、其凭证和使用参数。数据绑定可以引用事件数据（例如，DB 键是从事件 “username” 字段派生的），请参见：https://docs.microsoft.com/zh-CN/azure/azure-functions/functions-triggers-bindings。

函数输入

函数输入包括事件数据和元数据，并且可以包括上下文对象。

事件数据和元数据

事件详细信息应传递给函数处理程序，不同的事件可能具有不同的元数据，因此希望函数能够确定事件的类型并轻松解析公共和特定于事件的元数据。

可能需要将事件类与实现分离，例如：不管流存储是 Kafka 还是 Kinesis，处理消息流的函数都可以运行。在这两种情况下，它将接收消息正文和事件元数据，消息可能在不同框架之间路由。

事件可以包括单个记录（例如，在请求/响应模型中），也可以接受多个记录或微批处理（例如，在流模式中）。

FaaS 解决方案使用的常见事件数据和元数据的示例：

- Event Class/Kind
- 版本
- 事件 ID
- Event Source/Origin
- 来源身份
- 内容类型
- 邮件正文
- 时间戳记

事件/记录特定元数据的示例

- HTTP：Path、Method、Header、查询参数
- 消息队列：Topic、Header
- 记录流（Record Stream）：表、键、操作、修改时间、旧字段、新字段

事件源结构的示例：

- http://docs.aws.amazon.com/lambda/latest/dg/eventsources.html
- https://docs.microsoft.com/zh-cn/azure/azure-functions/functions-triggers-bindings
- https://cloud.google.com/functions/docs/concepts/events-triggers

一些实现将 JSON 视为将事件信息传递给函数的机制。对于高速函数（例如，流处理）或低能耗设备（IoT），这可能会增加大量的序列化/反序列化开销。在这些情况下，可能值得考虑使用本地语言结构或其他序列化机制。

函数上下文

调用函数时，框架可能希望提供对跨多个函数调用的平台资源或常规属性的访问，而不是将所有静态数据放入事件中或强制该函数在每次调用时初始化平台服务。

上下文（Context）可以是一组输入属性、环境变量或全局变量。有的实现将这三者结合使用。

上下文示例：

- 函数名称、版本、ARN
- 内存限制
- 请求 ID
- Cloud Region
- 环境变量
- 安全密钥/令牌
- 运行时/绑定路径
- 日志
- 数据绑定

有的实现初始化日志对象（例如， AWS 中的全局变量或 Azure 中的部分上下文），用户可以使用平台集成的工具查看日志来跟踪函数执行。除了传统的日志记录，未来的实现可能会将计数器/监控和跟踪活动抽象为平台上下文的一部分，以进一步提高函数的可用性。

数据绑定是函数上下文的一部分，平台根据用户配置启动与外部数据资源的连接，并且这些连接可以在多个函数调用之间重用。

函数输出

当函数退出时，它可能：

- 将值返回给调用方（例如，在 HTTP 请求/响应示例中）
- 将结果传递到工作流程中的下一个执行阶段

- 将输出写入日志

应该有确定的方式通过返回的错误值或退出代码来知道函数是成功还是失败。

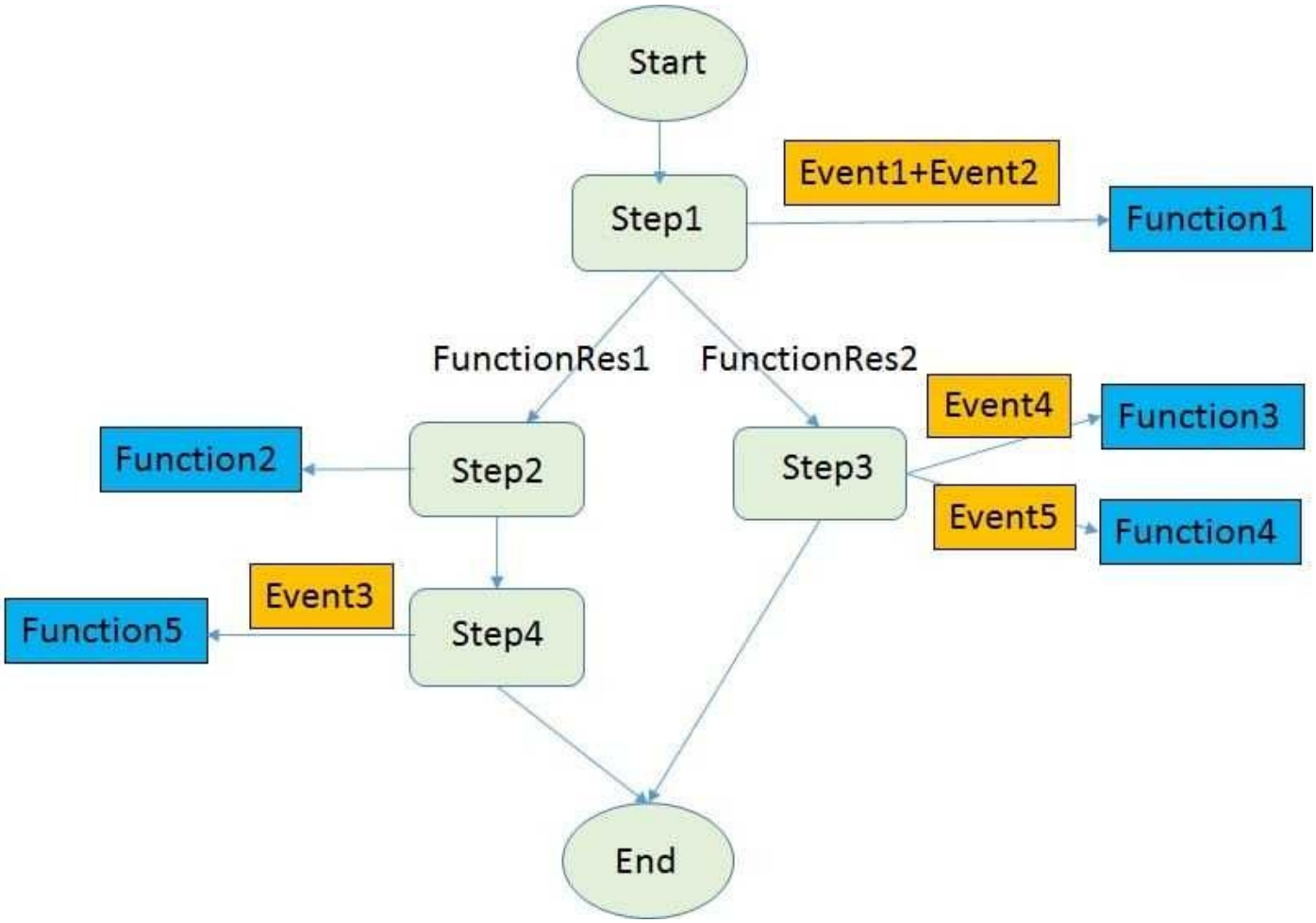
函数输出可以是结构化的（例如 HTTP 响应对象）或非结构化的（例如某些输出字符串）。

事件驱动

Serverless 应用是由事件驱动的，当应用观察的事件源中有情况发生时会触发响应的函数执行，函数执行后会到达某个状态，就像状态机一样，随着一系列的事件发生会触发函数顺序会并行运行，这里就会设计到事件数据结构、传递与工作流的规范。

由 CNCF Serverless 工作组制定的 CloudEvents 是以通用格式描述事件数据的规范，以提供跨服务、平台和系统的互操作性。

Workflow 用于表示一系列事件或函数运行结果触发状态变更并运行相应函数的流程，该流程可能看起来如下图所示。



CNCF Serverless 工作组的 Workflow 子组制定了供应商中立的 Workflow 规范，用于定义用户用于指定或描述其 serverless 应用程序流的格式或原语。

CloudEvents

> 本文翻译并节选自 CloudEvents - Version 1.0-rc1，在原文的基础上有所改动。

CloudEvents 是供应商中立的事件数据定义规范。

概览

事件无处不在。但是，事件产生者倾向于以不同的方式描述事件。

缺少通用的事件描述方式意味着开发人员需要不断重新学习如何使用事件。这也限制了库、工具和基础架构帮助跨环境（例如 SDK、事件路由器或跟踪系统）传递事件数据的潜力。以事件数据实现的可移植性和生产率上受到阻碍。

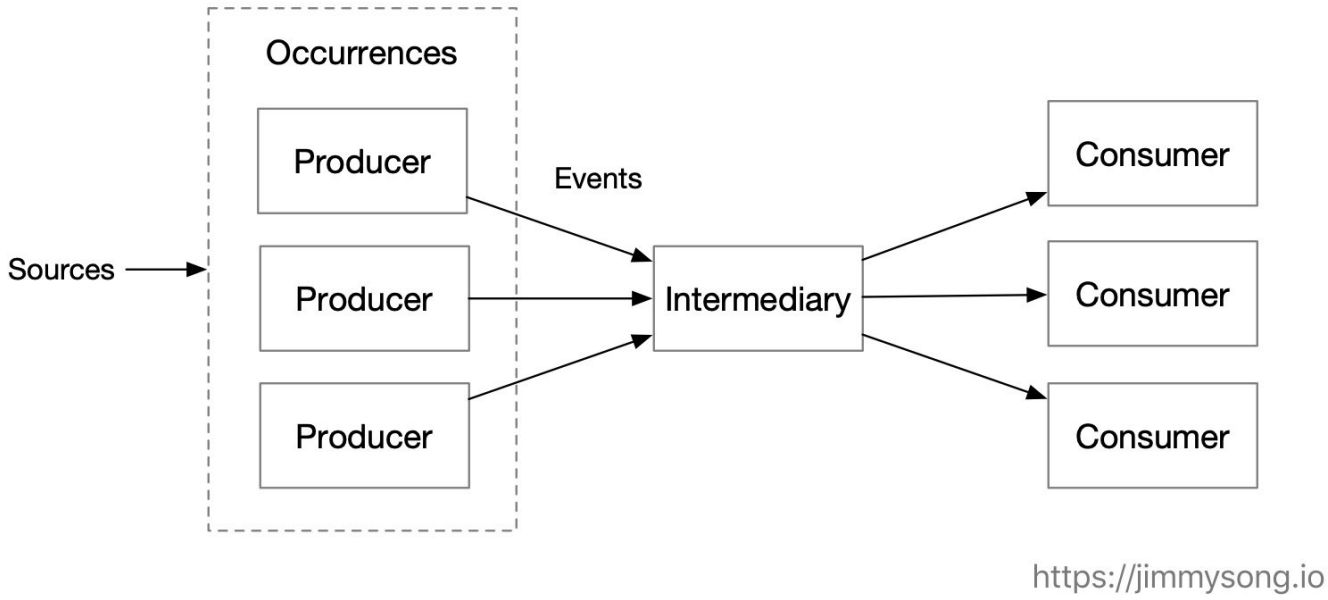
CloudEvents 是以通用格式描述事件数据的规范，以提供跨服务、平台和系统的互操作性。

事件格式指定如何使用某些编码格式序列化 CloudEvent。支持这些编码兼容 CloudEvents 的实现必须遵守相应事件格式中指定的编码规则。所有实现都必须支持 JSON 格式。

有关该规范背后的历史、开发和设计原理的更多信息，请参阅 CloudEvents Primer 文档。

术语

下图是对 CloudEvents 中的部分术语的关系描述。



Occurrence（发生）

“Occurrence（发生）”是在软件系统运行期间捕获的事实陈述。这可能是由于系统发出的信号或系统正在观察的信号，由于状态变化，计时器完成或任何其他值得注意的活动而发生的。 例如，由于电池电量不足或虚拟机将要执行重

启计划，设备可能会进入警报状态。

Event（事件）

“Event（发生）”是表示发生（Occurrence）及其上下文的数据记录。Event 从 Event 生产者（source）路由到感兴趣的 Event 使用者。Event 路由可以基于 Event 中包含的信息，但是 Event 不会标识特定的路由目的地。Event 包含两种类型的信息：表示 Occurrence 的 Event Data 和提供有关 Occurrence 上下文信息的 Context 元数据。一次发生可能会导致多个事件。

Producer（生产者）

“Producer（生产者）”是创建描述 CloudEvent 的数据结构的特定实例、过程或设备。

Source（源）

“Source（源）”是发生事件的上下文。在分布式系统中，Source 可能包含多个 Producer。 如果 Source 不知道 CloudEvent，则外部 Producer 将代表 Source 创建 CloudEvent。

Consumer（消费者）

“消费者”接收事件并对其采取行动。Consumer 使用上下文和数据执行某些逻辑，这可能导致新 Event 的发生。

Intermediary（中介）

“Intermediary（中介）”接收包含 Event 的消息，目的是将 Event 转发到下一个接收者，该接收者可能是另一个 Intermediary 或 Consumer。Intermediary 的典型任务是根据 Context 中的信息将 Event 路由到接收者。

Context（上下文）

Context（上下文）元数据封装在“Context Attributes（上下文属性）”中。 工具和应用程序代码可以使用此信息来标识 Event 与系统各方面或其他 Event 的关系。

Data（数据）

有关事件的特定于域的信息（即有效负载）。这可能包括有关 Occurrence 的信息，有关已更改数据的详细信息或更多信息。

Message（消息）

Event 是通过消息从源传递到目的地。

Protocol（协议）

可以通过各种行业标准协议（例如 HTTP、AMQP、MQTT、SMTP），开源协议（如 Kafka、NATS）或特定于平台/供应商的协议（AWS Kinesis、Azure Event Grid）传递 Message。

Context Attributes（上下文属性）

每个符合此规范的 CloudEvent 必须包含指定为 **REQUIRED** 的上下文属性，并且可以包括一个或多个 **OPTIONAL** 上下文属性。

这些属性虽然描述了事件，但设计为可以独立于事件数据进行序列化。这样就可以在目的地对它们进行检查，而不必对事件数据进行反序列化。

属性命名规范

CloudEvents 规范定义了到各种协议和编码的映射，随附的 CloudEvents SDK 针对各种运行时和语言。其中一些将元数据元素视为区分大小写，而其他元素则不区分大小写，并且单个 CloudEvent 可能会通过涉及协议、编码和运行时混合的多个跃点进行路由。因此，本规范限制了所有属性的可用字符集，以防止区分大小写问题或与通用语言中标识符的允许字符集冲突。

CloudEvents 属性名称必须由 ASCII 字符集的小写字母（“ a”至“ z”）或数字（“ 0”至“ 9”）组成，并且必须以小写字母开头。属性名称应具有描述性和简洁性，长度不得超过 20 个字符。

类型系统

以下抽象数据类型可用于属性。这些类型中的每一个可以通过不同的事件格式和在传输元数据字段中以不同的方式表示。该规范为所有实现必须支持的每种类型定义了规范的字符串编码。

REQUIRED 属性

以下属性必须出现在所有 CloudEvents 中：

- id: String
- source: URI-reference
- specversion: String
- type: String

OPTIONAL 属性

以下属性是可选的，可以出现在 CloudEvents 中。

- datacontenttype: String
- dataschema: URI
- subject: String
- time: Timestamp

扩展 Context Attributes

CloudEvents Producer 可以在 Event 中包含其他 Context Attribute，这些属性可以在与 Event 处理相关的辅助操作中使用。

该规范对扩展属性的语义没有任何限制，但必须使用类型系统中定义的类型。扩展的每个定义都应完全定义属性的所有方面，例如属性的名称，语义含义和可能的值，甚至表明它对其值没有任何限制。新的扩展名定义应该使用具有足够描述性的名称，以减少与其他扩展名发生名称冲突的几率。特别是，扩展作者应该检查扩展文档中的已知扩展集，不仅是可能的名称冲突，还是可能感兴趣的扩展。

每个定义如何序列化 CloudEvent 的规范都将定义扩展属性的显示方式。

扩展属性必须使用与所有 CloudEvents 上下文属性相同的常规模式进行序列化。例如，在二进制 HTTP 中，这意味着它们必须显示为带有 `ce-` 前缀的HTTP标头。属性的规范可以定义一个二级序列化，其中数据在消息中的其他位置重复。

在定义了二级序列化的情况下，扩展规范还必须说明如果两个序列化位置的数据不同，CloudEvent 的接收者将要做什么。另外，发送者需要为 intermediary 和接收者不知道其扩展的情况做好准备，因此专用序列化版本很可能不会作为 CloudEvent 扩展属性进行处理。

许多传输支持发送者包括附加元数据的功能，例如作为 HTTP 标头。虽然未强制要求 CloudEvents 接收器处理和传递它们，但建议通过某种机制来进行处理，以使其清楚地知道它们不是 CloudEvents 的元数据。

这是一个说明需要其他属性的示例。在许多物联网和企业用例中，Event 可以在无服务器应用程序中使用，该应用程序跨多种 Event 类型执行操作。为了支持这种用例，Event Producer 将需要向 “Context Attributes” 添加其他标识属性，Event 使用者可以使用这些属性将这个事件与其他事件相关联。如果此类身份属性恰好是事件 “Data” 的一部分，则 Event 生成器还将身份属性添加到 “Context Attributes”，Event 用者可以轻松访问此信息，而无需解码和检查 Event Data。此类身份属性还可用于帮助中间网关确定如何路由 Event。

Event Data（事件数据）

按照术语 Data（数据）的定义，CloudEvents 可以包含有关事件的特定于域的信息。 如果存在，此信息将封装在数据中。

Size Limits（大小限制）

在许多情况下，CloudEvents 将通过一个或多个通用 intermediary 进行转发，每个 intermediary 都可能会对转发 Event 的大小施加限制。CloudEvents 也可能会路由到受存储或内存限制的 Consumer（如嵌入式设备），因此会遇到大型单一事件。

Event 的 “Size（大小）”是其线路大小，并包括针对该 Event 在线路上传输的每个位：根据所选的 Event 格式和所选的协议绑定，传输 frame-metadata（帧元数据），event metadata（事件元数据）和 event data（事件数据）。

如果应用程序配置要求 Event 跨不同的传输进行路由或 Event 进行重新编码，则应该考虑应用程序使用的效率最低的传输和编码应符合以下大小限制：

- Intermediary 必须转发大小为 64 KB 或更小的事件。
- Consumer 应该接受至少 64 KB 的事件。

实际上，这些规则将允许 Producer 安全地发布最大为 64KB 的 Event。此处的安全是指通常合理的做法是，期望所有 Intermediary 都接受并转发该事件。无论是出于本地考虑，还是要接受或拒绝该大小的事件，它都在任何特定的 Consumer 控制之下。

通常，CloudEvents 发布者应该通过避免将大型数据项嵌入 Event 有效负载中来保持事件紧凑，而是将 Event 有效负载链接到此类数据项。从访问控制的角度来看，此方法还允许 Event 的更广泛分布，因为通过解析链接访问 Event 相关的详细信息可实现差异化的访问控制和选择性公开，而不是将敏感的详细信息直接嵌入事件中。

隐私与安全

互操作性是此规范的主要推动力，要使这种行为成为可能，就需要明确提供一些信息，从而可能导致信息泄漏。

请考虑以下事项，以防止意外泄漏，尤其是在利用第三方平台和通信网络时：

Context Attributes

敏感信息不应携带和表示在上下文属性中。

CloudEvent Producer、Consumer 和 Intermediary 可以内省并记录 Context Attributes。

数据

特定于域的 Event 数据应该被加密以限制对可信方的可见性。用于这种加密的机制是 Producer 和 Consumer 之间的协议，因此不在本规范的范围之内。

传输绑定

应当采用传输级安全性来确保 CloudEvents 的可信和安全交换。

示例

下面时候使用 Json 序列化 CloudEvent 的示例。

```
{
    "specversion" : "1.0-rc1",
    "type" : "com.github.pull.create",
    "source" : "https://github.com/cloudevents/spec/pull",
    "subject" : "123",
    "id" : "A234-1234-1234",
    "time" : "2018-04-05T17:31:00Z",
    "comexampleextension1" : "value",
    "comexampleothervalue" : 5,
    "datacontenttype" : "text/xml",
    "data" : "<much wow=\"xml\"/>"
}
```

参考

- CloudEvents - Version 1.0-rc1 - github.com

Workflow

> 本文翻译并节选自 Workflow - Version 0.1，在原文的基础上有所改动。

Workflow 是供应商中立的规范，用于定义用户用于指定或描述其 serverless 应用程序流的格式或原语。

许多 serverless 应用程序不是由单个事件触发的简单函数，而是由系列函数执行的多个步骤组成，而函数在不同步骤中由不同事件触发。如果某个步骤涉及多个函数，则该步骤中的函数可能会根据不同的事件触发器依次执行、并行执行或在分支中执行。为了使 serverless 平台正确执行 serverless 应用程序的函数工作流程，应用程序开发人员需要提供工作流程规范。

为了给业界提供一种标准方法，CNCF Serverless WG（Working group）成立了 workflow 子组，供用户指定其 serverless 应用程序工作流，以促进 serverless 应用程序在不同供应商平台之间的可移植性。

为此 CNCF Serverless WG workflow 小组制定了一个完整的协议，使给定的事件时间轴和工作流始终产生相同的作用。

相关使用案例请参考 Workflow - Version 0.1 中给出了一系列 use case:

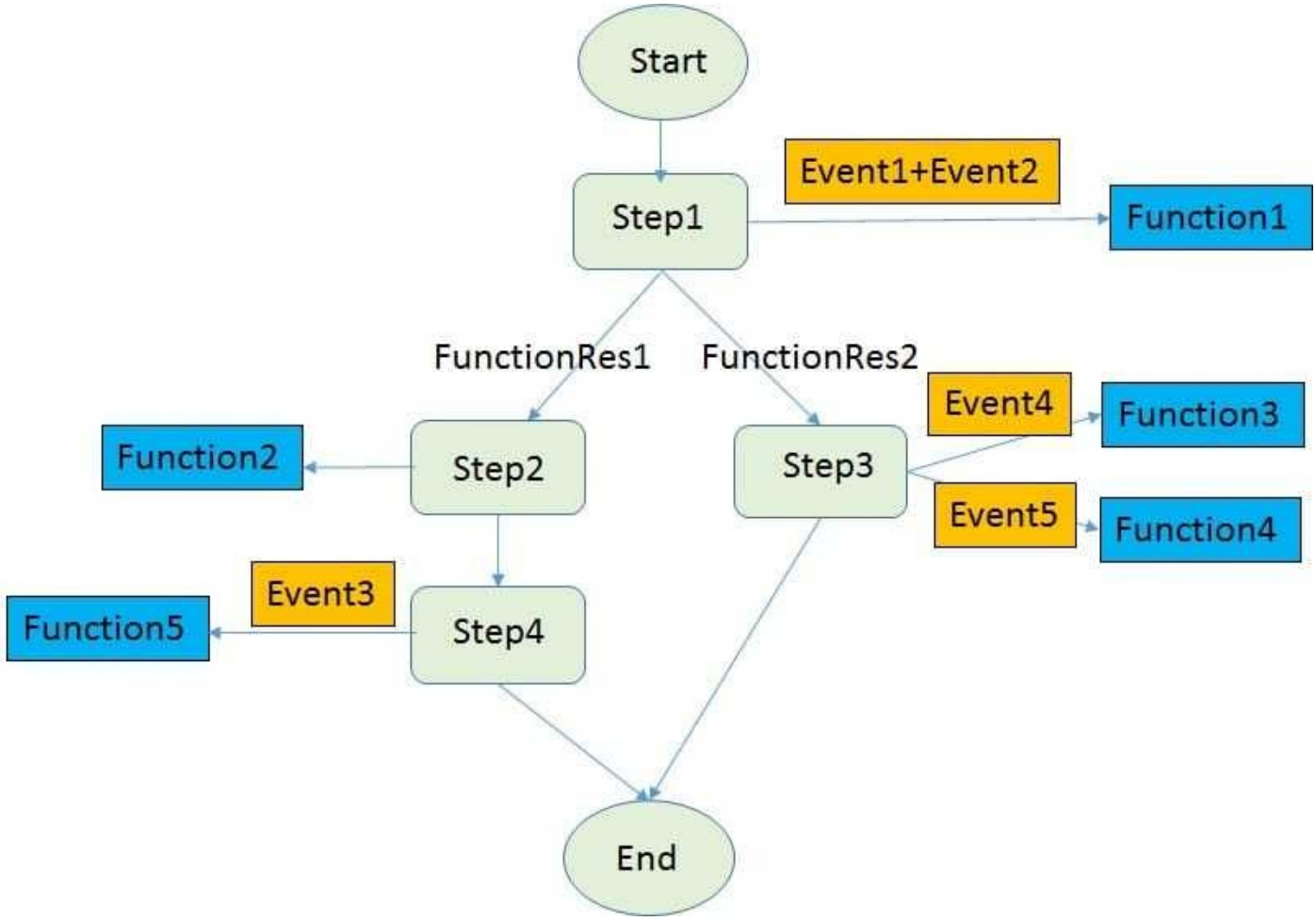
- 家庭监控
- 借贷审批
- 员工旅行预订
- 流媒体视频点播
- 翻译服务评价

功能范围

函数工作流用于将函数编排为协调的微服务应用程序。函数工作流中的每个函数可能由来自各种来源的事件驱动。函数工作流将函数和触发事件分组到一个连贯的单元中，并描述函数的执行和以规定方式传递的信息。具体来说，函数工作流程允许用户:

- 定义 serverless 应用程序中涉及的步骤/状态和工作流程。
- 定义每个步骤中涉及的函数。
- 定义哪个事件或事件组合触发一个或多个函数。
- 定义在触发多个函数时如何安排这些函数依次执行还是并行执行。
- 指定如何在函数或状态之间过滤信息和传递事件。
- 定义在哪种错误状态下需要重试。
- 如果函数是由两个或多个事件触发的，则定义应使用什么标签/键将那些事件与相同的函数工作流实例相关联。

以下是涉及事件和函数的函数工作流的示例。使用这样的函数工作流，用户可以轻松指定事件与函数之间的交互以及如何在工作流程中传递信息。



使用函数工作流，用户可以定义集合点（状态）以等待预定义的事件，然后再执行一个或多个函数并继续执行函数工作流。

Workflow 模型

可以将函数工作流（Function Workflow）视为状态的集合以及这些状态之间的转换和分支，并且每个状态可以具有关联的事件和/或功能。函数工作流可以从 CLI 命令调用，也可以在事件从事件源到达时动态触发。来自事件源的事件也可能与函数工作流中的特定状态相关联。函数工作流中的这些状态将等待一个或多个事件源中的一个或多个事件到达，然后再执行其关联的操作并进入下一个状态。其他工作流程功能包括：

- 函数的结果可用于启动重试操作或确定下一个要执行的函数或要转换到的状态。
- 函数工作流提供了一种在事件处理过程中对 JSON 事件有效负载进行过滤和转换的方法。
- 函数工作流为应用程序开发人员提供了一种在事件中指定唯一字段的方法，该字段可用于将事件源中的事件关联到同一函数工作流实例。

我们可以很自然的将函数工作流建模为状态机。以下是函数工作流的定义/规范提供的状态列表。工作流的规范称为工作流程模板。工作流模板的实例称为工作流实例（workflow instance）。

- **Event State**（事件状态）：用于等待事件源中的事件，然后调用一个或多个函数以顺序或并行运行。
- **Operation State**（操作状态）：允许一个或多个函数按顺序或并行运行，而无需等待任何事件。

- **Switch State**（切换状态）：允许转换到其他多个状态（例如，不同的函数导致前一个状态触发分支/转换到不同的下一个状态）。
- **Delay State**（延迟状态）：使工作流执行延迟指定的持续时间或直到指定的时间/日期。
- **End State**（结束状态）：失败/成功终止工作流。
- **Parallel State**（并行状态）：允许多个状态并行执行。

函数工作流由工作流规范描述。

工作流规范

函数工作流规范定义了函数工作流的行为和操作。函数工作流规范结构应允许用户定义事件到达触发的执行函数。它应具有足够的灵活性，以涵盖从单个函数的简单调用到涉及多个函数和多个事件的复杂应用程序和各种微服务应用程序。

从高层看，函数工作流规范包括两部分：触发器定义（`trigger definition`）和状态定义（`state definition`）。

```
{
  "trigger-defs" : [],
  "states": []
}
```

`trigger-defs` 数组（仅在存在与工作流关联的事件时才需要）是与函数工作流关联的事件触发器的数组。如果应用程序工作流中涉及多个事件，则必须在该事件触发器中指定一个用于将事件与同一工作流实例的其他事件相关联的关联令牌（`correlation-token`）。

状态数组（必需）是与函数工作流相关联的状态数组。

下面是 JSON 格式的函数工作流示例，其中涉及事件状态和该事件状态的触发器：

```
{
   "trigger-defs":[
      {
         "name":"OBS-EVENT",
         "source":"CloudEvent source",
         "eventID":"CloudEvent eventID",
         "correlation-token":"A path string to an identification label field in the event message"
      },
      {
         "name":"TIMER-EVENT",
         "source":"CloudEvent source",
         "eventID":"CloudEvent eventID",
         "correlation-token":"A path string to an identification label field in the event message"
      }
   ],
   "states":[
      {
```

```
         "name":"STATE-OBS",
         "start":true,
         "type":"EVENT",
         "events":[
            {
               "event-expression":"boolean expression 1 of triggering events",
               "action-mode":"Sequential or Parallel",
               "actions":[
                  {
                     "function":"function name 1"
                  },
                  {
                     "function":"function name 2"
                  }
               ],
               "next-state":"STATE-END"
            },
            {
               "event-expression":"boolean expression 2 of triggering events",
               "action-mode":"Sequential or Parallel",
               "actions":[
                  {
                     "function":"function name 3"
                  },
                  {
                     "function":"function name 4"
                  }
               ],
               "next-state":"STATE-END"
            }
         ]
      },
      {
         "name":"SATATE-END",
         "type":"END"
      }
   ]
}
```

以下是带有操作状态的函数工作流的另一个示例：

```
{
   "states":[
      {
         "name":"STATE-ALARM-NOTIFY",
         "start":true,
         "type":"OPERATION",
         "action-mode":"Sequential or Parallel",
         "actions":[
            {
               "function":"function name 1"
            },
```

```
            {
               "function":"function name 2"
            }
         ],
         "next-state":"STATE-END"
      }
   ]
}
```

触发器定义

`trigger-defs` 数组由一个或多个事件触发器组成。

以 JSON 格式定义的事件触发器示例如下：

```
{
   "trigger-defs":[
      {
         "name":"EVENT-NAME",
         "source":"CloudEvent source",
         "eventID":"CloudEvent eventID",
         "correlation-token":"A path string to an identification label field in the event message"
      }
   ]
}
```

动作定义 {#action-definition}

下面是 JSON 格式的定义。

```
{
   "actions":[
      {
         "function":"FUNCTION-NAME",
         "timeout":"TIMEOUT-VALUE",
         "retry":[
            {
               "match":"RESULT-VALUE",
               "retry-interval":"INTERVAL-VALUE",
               "max-retry":"MAX-RETRY",
               "next-state":"STATE-NAME"
            }
         ]
      }
   ]
}
```

- **function:** 指定调用的函数。
- **timeout:** 从请求发送给函数起开始计时，等待函数指定完成的时间，单位秒，必须为正整数。

- **retry:** 重试策略。
- **match:** 匹配的结果值。
- **retry-interval 和 max-retry:** 当出现错误时使用。
- **next-state:** 当超过 `max-retry` 限制后到转移到下一个状态。

状态定义

事件状态（Event State）

```
{
   "states":[
      {
         "name":"STATE-NAME",
         "type":"EVENT",
         "start":true,
         "events":[
            {
               "event-expression":"EVENTS-EXPRESSION",
               "timeout":"TIMEOUT-VALUE",
               "action-mode":"ACTION-MODE",
               "actions":[
               ],
               "next-state":"STATE-NAME"
            }
         ]
      }
   ]
}
```

事件状态必须将 `type` 值指定为 `EVENT` 。

- **start:** 是否为起始状态。可选的字段。默认为 false。
- **events:** 与该事件状态相关的事件数组。
- **event-expression:** 这是一个布尔表达式，由一个或多个事件操作数和布尔运算符组成。`EVENTS-EXPRESSION` 可以是 “Event1 or Event2”。到达并匹配 `EVENTS-EXPRESSION` 的第一个事件将导致执行此状态的所有操作，然后转换到下一个状态。
- **timeout:** 指定在 `EVENTS-EXPRESSION` 中等待事件的时间段。如果事件不在超时时间内发生，则工作流将转换为结束状态。
- **action-mode:** 指定函数是按顺序执行还是并行执行，并且可以是 `SEQUENTIAL` 或 `PARALLEL` 。
- **next-state:** 指定在成功执行所有匹配事件的操作之后要转换到的下一个状态的名称。

操作状态（Operation State）

```
{
   "states":[
      {
         "name":"STATE-NAME",
         "type":"OPERATION",
         "start":true,
```

```
          "action-mode":"ACTION-MODE",
          "actions":[
          ],
          "next-state":"STATE-NAME"
       }
    ]
}
```

- **action-mode:** 指定函数是按顺序执行还是并行执行，并且可以是 `SEQUENTIAL` 或 `PARALLEL` 。
- **actions:** 由一系列动作构成的列表，指定接收到与事件表达式匹配的事件时要执行的函数的列表。
- **next-state:** 指定在成功执行所有匹配事件的操作之后要转换到的下一个状态的名称。

分支状态（Switch State）

```
{
   "states":[
      {
         "name":"STATE-NAME",
         "type":"SWITCH",
         "start":true,
         "choices":[
            {
               "path":"PAYLOAD-PATH",
               "value":"VALUE",
               "operator":"COMPARISON-OPERATOR",
               "next-state":"STATE-NAME"
            },
            {
               "Not":{
                  "path":"PAYLOAD-PATH",
                  "value":"VALUE",
                  "operator":"COMPARISON-OPERATOR"
               },
               "next-state":"STATE-NAME"
            },
            {
               "And":[
                  {
                     "path":"PAYLOAD-PATH",
                     "value":"VALUE",
                     "operator":"COMPARISON-OPERATOR"
                  },
                  {
                     "path":"PAYLOAD-PATH",
                     "value":"VALUE",
                     "operator":"COMPARISON-OPERATOR"
                  }
               ],
               "next-state":"STATE-NAME"
            },
            {
               "Or":[
```

```
                {
                   "path":"PAYLOAD-PATH",
                   "value":"VALUE",
                   "operator":"COMPARISON-OPERATOR"
                },
                {
                   "path":"PAYLOAD-PATH",
                   "value":"VALUE",
                   "operator":"COMPARISON-OPERATOR"
                }
             ],
             "next-state":"STATE-NAME"
          }
       ],
       "default":"STATE-NAME"
    }
  ]
}
```

- **choices:** 针对输入数据定义了一个有序的匹配规则集，以使数据进入此状态，并为每个匹配项转换为下一个状态。
- **path:** JSON Path，用于选择要匹配的输入数据的值。
- **value:** 匹配值。
- **operator:** 指定如何将输入数据与值进行比较，例如 “EQ”、“LT”、“LTEQ”、“GT”、“GTEQ”、“StrEQ”、“StrLT”、“ StrLTEQ”、“ StrGT”、“StrGTEQ”。
- **next-state:** 指定在存在值匹配时要转换到的下一个状态的名称。
- **Not:** 必须是单个匹配规则，且不得包含 `next-state` 字段。
- **And** 和 **Or:** 必须是匹配规则的非空数组，它们本身不能包含 `next-state` 字段。
- **default:** 如果任何选择值都不匹配，则 `default` 字段将指定下一个状态的名称。
- 关于 **next-state:** 评估的顺序是从上到下，如果发生匹配，请转到 `next-state`，并忽略其余条件。

延迟状态（Delay State）

```
{
  "states":[
     {
        "name":"STATE-NAME",
        "type":"DELAY",
        "start":true,
        "time-delay":"TIME-VALUE",
        "next-state":"STATE-NAME"
     }
  ]
}
```

- **time-delay:** 指定时间延迟。 `TIME-VALUE` 是在此状态下延迟的时间（以秒为单位）。 必须是正整数。
- **next-state:** 指定要转换到的下一个状态的名称。 `STATE-NAME` 在函数工作流中必须是有效的 State 名称。

结束状态（End State）

```
{
   "states":[
      {
         "name":"STATE-NAME",
         "type":"END",
         "status": "STATUS"
      }
   ]
}
```

- **status:** 该字段必须为 SUCCESS 或 FAILURE，表示工作流结束。

并行状态（Parallel State）

并行状态由多个并行执行的状态组成。并行状态具有多个同时执行的分支。每个分支都有一个状态列表，其中一个状态为开始状态。每个分支继续执行，直到达到该分支内没有下一个状态的状态为止。当所有分支都执行完成后，并行状态将转换为下一个状态。本质上，这是在并行状态内嵌套一组状态。

并行状态由状态类型 “PARALLEL” 定义，并包括一组并行分支，每个分支都有自己的独立状态。每个分支都接收并行状态的输入数据的副本。除 END 状态外，任何类型的状态都可以在分支中使用。

分支内状态的 ”next-state” 转换只能是到该分支内的其他状态。另外，并行状态之外的状态不能转换到并行状态的分支内的状态。

并行状态会生成一个输出数组，其中每个元素都是分支的输出。输出数组的元素不必是同一类型。

```
{
   "states":[
      {
         "name":"STATE-NAME",
         "type":"PARALLEL",
         "start": true,
         "branches":[
            {
               "name":"BRANCH-NAME1",
               "states":[

               ]
            },
            {
               "name":"BRANCH-NAME2",
               "states":[

               ]
            }
         ],
         "next-state":"STATE-NAME"
      }
   ]
```

```
24. }
```

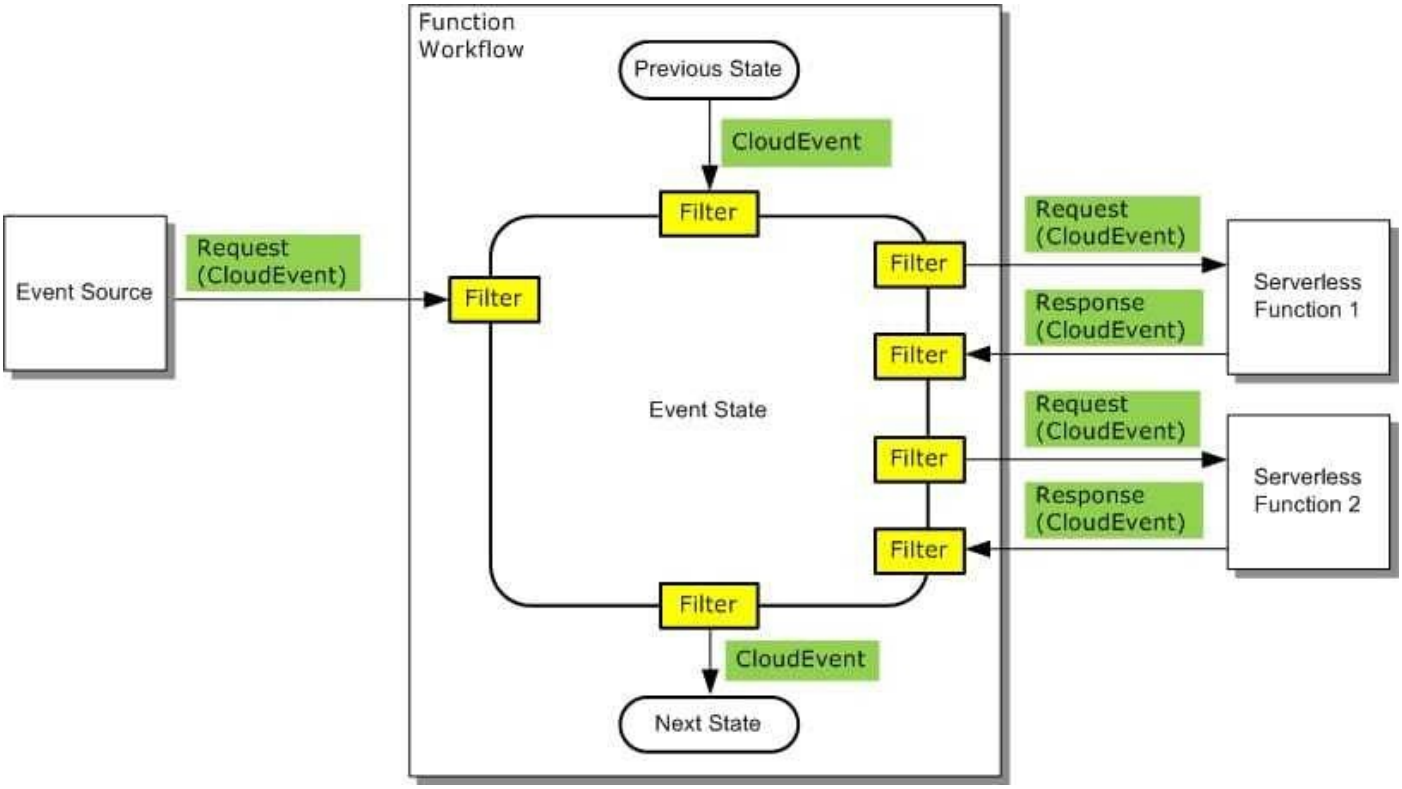
- **branch:** 同时执行的分支的列表。每个命名分支都有一个 “states” 列表。分支内每个状态的 “next-state” 字段必须是该分支内的有效状态名称，或者不存在以指示该状态终止该分支的执行。分支执行从分支内具有 “start”: true 的状态开始。
- **next-state:** 指定在所有分支完成执行之后要转换到的下一个状态的名称。 `STATE-NAME` 在函数工作流中必须是有效的状态名，但不能是并行状态本身中是状态。

信息传递

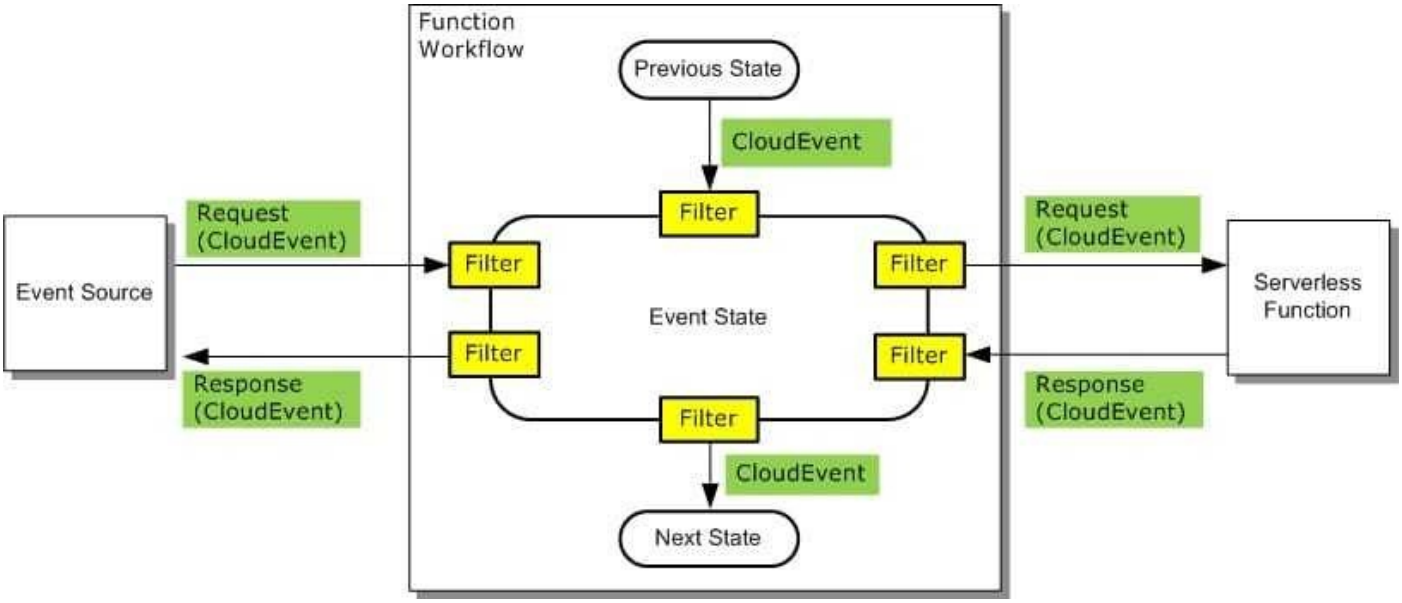
下图显示了通过函数工作流的数据流，该函数工作流包括调用两个 serverless 函数的事件状态。来自一个状态的输出数据作为输入数据传递到下一状态。过滤器用于过滤和转换进入和退出每个状态时的数据。从工作流中的 Operation State 调用时，来自先前状态的输入数据可能会传递到 serverless 函数。

来自 Serverless 函数的响应中包含的数据将作为输出数据发送到下一个状态。如果状态（Operation State 或 Event State）包括一系列顺序操作，则将过滤来自一个 Serverless 函数的响应中包含的数据，然后在请求中将其发送给下一个函数。

在 Event State 下，在将请求从事件源接收到的 CloudEvent 元数据传递到 Serverless 函数之前，可以对其进行转换并将其与从先前状态接收到的数据进行组合。同样，在从 Serverless 函数的响应中接收到的 CloudEvent 元数据可以转换并与从先前状态接收到的数据组合，然后再在发送到事件源的响应中进行传递。



在某些情况下，诸如 API 网关之类的事件源希望收到工作流的响应。在这种情况下，可以将在从 Serverless 函数的响应中接收到的 CloudEvent 元数据进行转换，并与从先前状态接收到的数据进行组合，然后再将其在发送到事件源的响应中进行传递，如下所示。



过滤器机制

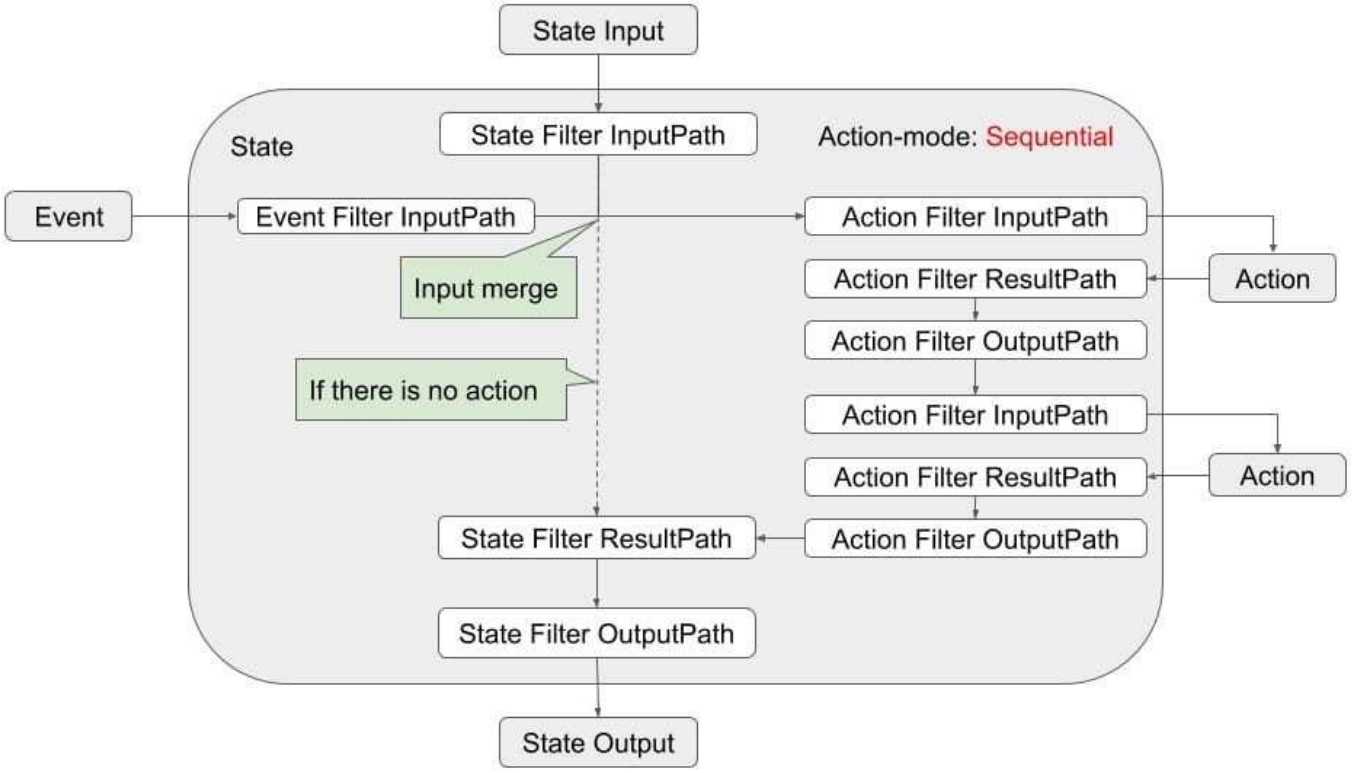
状态机维护一个隐式 JSON 数据，该数据可以从每个过滤器作为 JSONPath 表达式 `$` 进行访问。

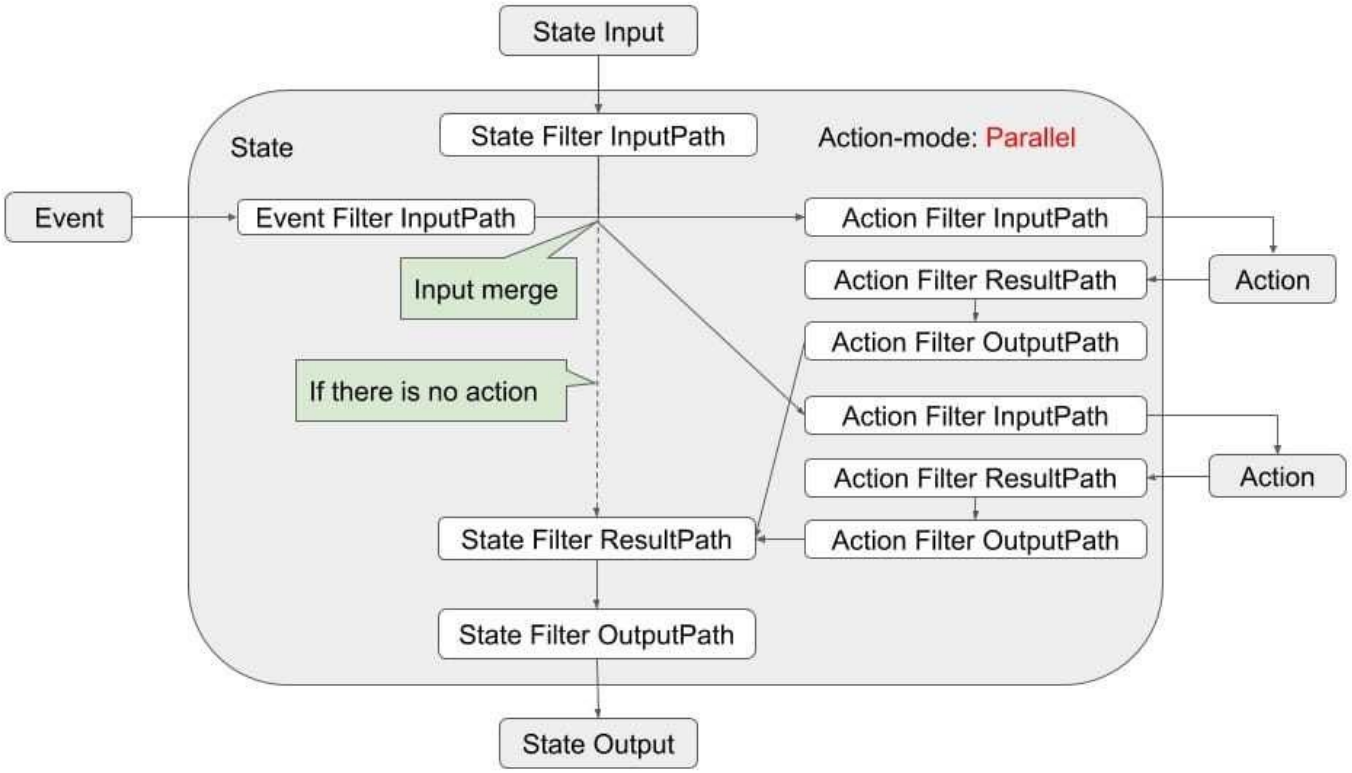
过滤器共有三种：

- 事件过滤器（**Event Filter**）
  - 当数据从事件传递到当前状态时调用。
- 状态过滤器（**State Filter**）
  - 当数据从先前状态传递到当前状态时调用
  - 当数据从当前状态传递到下一个状态时调用
- 动作过滤器（Action 是指定义 Serverless 函数的动作定义）
  - 当数据从当前状态传递到第一个操作时调用
  - 当数据从一个动作传递到另一个动作时调用
  - 当数据从最后一个动作传递到当前状态时调用

每个过滤器都有三种路径过滤器：

- **InputPath**
  - 选择事件、状态或操作的输入数据作为 JSONPath默认值为 `$`
- **ResultPath**
  - 将 Action 输出的结果 JSON 节点指定为 JSONPath
  - 默认值为 $
- **OutputPath**
  - 将 State 或 Action 的输出数据指定为 JSONPath
  - 默认值为 `$`





示例

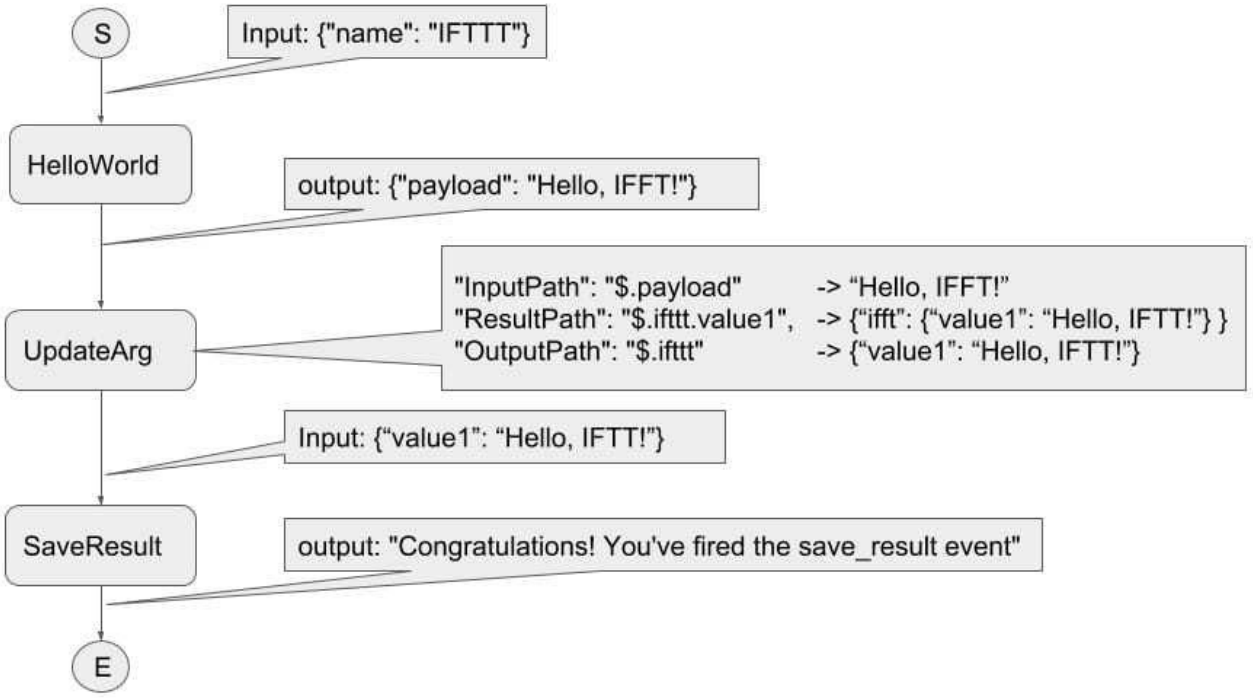
组成以下两个函数 “hello” 和 “save_result”。 第二个函数 “save_result” 不能将第一个函数 “ hello” 输出作为输入。

- Function “hello”:input {“name”: String}output {“payload”: String}
- Function “save_result”:input {“value1: String}output String

CNCF Function Workflow 语言:

```
{
   "states":[
      {
         "name":"HelloWorld",
         "type":"OPERATION",
         "start":true,
         "action-mode":"Sequential",
         "actions":[
            {
               "function":"hello"
            }
         ],
         "next-state":"UpdateArg"
      },
      {
         "name":"UpdateArg",
         "type":"OPERATION",
         "start":false,
         "action-mode":"Sequential",
         "InputPath":"$.payload",
         "ResultPath":"$.ifttt.value1",
         "OutputPath":"$.ifttt",
         "actions":[

         ],
         "next-state":"SaveResult"
      },
      {
         "name":"SaveResult",
         "type":"OPERATION",
         "start":false,
         "action-mode":"Sequential",
         "actions":[
            {
               "function":"save_resut"
            }
         ],
         "next-state":"STATE_END"
      },
      {
         "name":"STATE-END",
         "type":"END"
      }
   ]
}
```



错误

状态机在运行时返回以下预定义的错误代码。通常，它在动作定义的 `retry` 字段中使用。

- `SYS.Timeout`
- `SYS.Fail`
- `SYS.MatchAny`
- `SYS.Permission`
- `SYS.InvalidParameter`
- `SYS.FilterError`

参考

- Workflow - Version 0.1 - github.com

Knative 简介

Knative 开源于 2018 年 8 月，由 Pivotal、Google、IBM 等公司共同发起，从以 K 打头的名字上就可以看出来 Knative 是用以扩展 Kubernetes 的。

组件

Knative 最初包含以下 3 个组件：

- Build*：提供了一种从源代码构建容器的可插拔模型。它以 Google 的容器构建服务为基础。
- Eventing：提供用来使用和生成符合 CloudEvents 规范的事件的构建块。它包括对来自事件源的信息流的抽象，以及通过由可插拔发布/订阅代理服务提供支持的消息传递通道实现交付解耦。
- Serving：可缩放至零、请求驱动的计算运行环境，利用 Istio 在各版本之间路由流量。Serving 的目标是为 Kubernetes 提供扩展功能，用于部署和运行 Serverless 工作负载。

> [warning] 注意
>
> Knative 自 0.8 版本起去掉了 Build 组件，转而使用 tekoncd/pipeline 取代，只保留了 Eventing 和 Serving 组件。

受众

不同受众参与和使用 Knative 的方式不同，如下图所示。

Knative 特性

- 对于常用应用用例的更高级别抽象
- 安全、无状态、可扩展应用的秒级启动
- 功能松耦合，可任意组装
- 组件可拔插，你可以使用自己的日志、监控、网络和服务网格
- 可移植：在任意 Kubernetes 集群上运行，无需担心供应商锁定
- 顺应开发者习惯，支持如 GitOps、DockerOps、ManualOps 等通用模式
- 可以与通用工具及框架一起使用，如 Django、Ruby on Rails、Spring 等

参考

- https://pivotal.io/cn/knative
- https://github.com/knative

Knative 安装

我们在已有的 Kubernetes 集群上安装 Knative，因为 Knative 依赖 Istio，首先我们需要先安装 Istio。

安装 Istio

我们安装的 Istio 的基本情况：

- Kubernetes 1.15
- Istio v1.1.7
- 启动 Sidecar 自动注入
- 使用 Helm 安装
- 不启用 SDS

> [info] 提示
>
> 截止本文发稿时 Istio 已发布 v1.3，但是为了保持兼容性，本文中仍使用 Istio v1.1.7，以保持与 Knative 官方要求的版本统一。

运行以下命令安装 Istio。

```
# Knative v0.9 在 Istio v1.1.7 验证过
export ISTIO_VERSION=1.1.7
curl -L https://git.io/getLatestIstio | sh -
cd istio-${ISTIO_VERSION}

# 安装 Istio CRD
for i in install/kubernetes/helm/istio-init/files/crd*yaml; do kubectl apply -f $i; done

# 创建 istio-system namespace
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Namespace
metadata:
  name: istio-system
  labels:
    istio-injection: disabled
EOF

# 启用 sidecar 注入的模板
helm template --namespace=istio-system \
  --set sidecarInjectorWebhook.enabled=true \
  --set sidecarInjectorWebhook.enableNamespacesByDefault=true \
  --set global.proxy.autoInject=disabled \
  --set global.disablePolicyChecks=true \
  --set prometheus.enabled=false \
  `# 禁用 mixer prometheus adapter, 删除 istio 默认的 metrics` \
  --set mixer.adapters.prometheus.enabled=false \
  `# 禁用 mixer policy check, 我们的模板里不使用 policy` \
  --set global.disablePolicyChecks=true \
```

```
  `# 将 gateway pod 设置为 1 以规避最终一致性/readiness 问题` \
  --set gateways.istio-ingressgateway.autoscaleMin=1 \
  --set gateways.istio-ingressgateway.autoscaleMax=1 \
  --set gateways.istio-ingressgateway.resources.requests.cpu=500m \
  --set gateways.istio-ingressgateway.resources.requests.memory=256Mi \
  `# 多个 pilot replica 便于伸缩` \
  --set pilot.autoscaleMin=2 \
  `# 将 pilot 追踪采样设置为 100%` \
  --set pilot.traceSampling=100 \
  install/kubernetes/helm/istio \
  > ./istio.yaml

# 部署 istio
kubectl apply -f istio.yaml
```

使用 Helm 渲染完成的 `istio.yaml` 文件已保存在 `manifests/istio/v1.17/istio.yaml` 文件中，以后每次可以直接使用该文件执行安装和卸载 Istio。

验证 Istio 安装

执行下面的命令验证 Istio 的安装是否正确。

首先检查 Istio 的 pod 是否完全启动。

```
$ kubectl get pods --namespace istio-system
NAME                                      READY   STATUS      RESTARTS   AGE
istio-citadel-8559955d59-j5xx9            1/1     Running     0          40m
istio-cleanup-secrets-1.1.7-rb9hp         0/1     Completed   0          40m
istio-galley-fd84c8888-8kljq              1/1     Running     0          40m
istio-ingressgateway-8486b5db4f-hhqfg     1/1     Running     0          40m
istio-pilot-7846986bf5-5ql82              2/2     Running     0          40m
istio-pilot-7846986bf5-xgqhk              2/2     Running     0          40m
istio-policy-f7f8c578b-7rz2j              2/2     Running     2          40m
istio-security-post-install-1.1.7-nqm9z   0/1     Completed   0          40m
istio-sidecar-injector-c645cf64-8hfzg     1/1     Running     0          40m
istio-telemetry-575c8d8d66-pb27q          2/2     Running     3          40m
```

部署 bookinfo 示例应用确认 Istio 所有组件可以正常工作。

```
kubectl -n default apply -f manifests/istio/bookinfo-sample
```

参考 Bookinfo Application 确认可以正常访问 productpage 页面。

安装 Knative

我们安装的 Istio 的基本情况：

- Knative v0.9.0

运行下面的命令直接安装 Knative。

```
# 安装 Knative CRD
kubectl apply --selector knative.dev/crd-install=true \
   --filename https://github.com/knative/serving/releases/download/v0.9.0/serving.yaml \
   --filename https://github.com/knative/eventing/releases/download/v0.9.0/release.yaml \
   --filename https://github.com/knative/serving/releases/download/v0.9.0/monitoring.yaml

# 再运行一遍 kubectl apply
kubectl apply --filename https://github.com/knative/serving/releases/download/v0.9.0/serving.yaml \
   --filename https://github.com/knative/eventing/releases/download/v0.9.0/release.yaml \
   --filename https://github.com/knative/serving/releases/download/v0.9.0/monitoring.yaml
```

或者使用本书中提供的 YAML 直接安装。

```
kubectl apply --selector knative.dev/crd-install=true  -f manifests/knative/0.9
kubectl apply -f manifests/knative/v0.9.0
```

> [info] 提示
>
> 因为 quay.io 、 gcr.io 、 k8s.gcr.io 、 docker.elastic.co 等镜像仓库在中国大陆无法访问，位于中国大陆的用户可以使用本书仓库中的 manifests/knative/0.9 目录下的 YAML 文件安装，其中以上镜像已替换为 DockerHub 镜像源。

验证 Knative 安装

运行下面的命令验证 Knative 所有 pod 是否可以正常启动。

```
$ kubectl get pods --namespace knative-serving
$ kubectl get pods --namespace knative-eventing
$ kubectl get pods --namespace knative-monitoring
NAME                                READY   STATUS    RESTARTS   AGE
activator-76f486c78-9k7l7           2/2     Running   3          34m
autoscaler-7495bcbc4b-j58n5         2/2     Running   2          34m
autoscaler-hpa-66b55c9688-w2x58     1/1     Running   0          34m
controller-859b7dbb8f-pdwfp         1/1     Running   0          34m
networking-istio-dc5f9b9f8-8qxfh    1/1     Running   0          34m
webhook-8dcd46846-kz78d             1/1     Running   0          34m
NAME                                    READY   STATUS    RESTARTS   AGE
eventing-controller-8466765fbf-djs8v    1/1     Running   0          34m
eventing-webhook-684467d8f5-klb7s       1/1     Running   0          34m
imc-controller-66dfdb6878-8mppw         1/1     Running   0          34m
imc-dispatcher-7db65bf44b-kwwhs         1/1     Running   0          34m
sources-controller-75c57459cc-xttxr     1/1     Running   0          34m
NAME                                READY   STATUS    RESTARTS   AGE
elasticsearch-logging-0             1/1     Running   0          12m
elasticsearch-logging-1             1/1     Running   0          11m
grafana-85c86fb7b9-8b95g            1/1     Running   0          34m
kibana-logging-85569f954d-fc65f     1/1     Running   0          35m
kube-state-metrics-6fd74d89bf-7kmk6 4/4     Running   0          30m
node-exporter-6z6nk                 2/2     Running   0          34m
```

```
node-exporter-hm2gr                2/2     Running   0          34m
node-exporter-rhc7r                2/2     Running   0          34m
prometheus-system-0                1/1     Running   0          34m
prometheus-system-1                1/1     Running   0          34m
```

参考 Getting Started with App Deployment 部署 helloworld-go 示例验证 Knative 是否正常运行。

```
kubectl -n default apply -f manifests/knative/samples/helloworld-go/service.yaml
```

参考

- Installing Istio for Knative - knative.dev
- Installing Knative - knative.dev
- Performing a Custom Knative Installation - knative.dev
- Bookinfo Application - istio.io

CNCF Serverless WG

CNCF Serverless WG 成立于 2017 年中。当前的主要成果有：

- CNCF Serverless Whitepaper V1.0：阐明 serverless 技术概况、生态系统状态、为 CNCF 的下一步动作做指导。
- Serverless Landscape：展示目前的 serverless 生态系统。
- CloudEvents 规范：定义最小化的事件通用属性，于 2017 年 12 月启动，2018 年 5 月进入 CNCF Sandbox
- Workflow 规范：函数编排

关于该工作组的详细信息请访问 https://github.com/cncf/wg-serverless。

下一步工作

CNCF 技术监督委员会（TOC）会考虑：

- 鼓励更多的 **serverless** 技术供应商和开源开发人员加入 **CNCF**，以共享想法并相互借鉴创新。例如，保持“Serverless Landscape”文档中列出的开源项目更新，并维护功能矩阵。

- 通过建立可互操作的 **API** 来培育开放的生态系统，并通过供应商承诺和开源工具确保可互操作的实施。在平台提供商和第三方开发人员库创建者的帮助下，从事类似于 CSI 和 CNI 新的互操作性和可移植性工作。其中一些可能值得创建自己的 CNCF 工作组，或者可以作为 Serverless 工作组的倡议而继续。例如：

    - 事件：定义通用事件格式和 API 以及元数据。可以在 Serverless WG Github 存储库中找到一些初始建议。
    - 部署：利用既是 Serverless 提供者又是现有 CNCF 成员，成立一个新的工作组，探索可能采取的小步骤，以协调一组通用的功能定义元数据。例如：应用程序定义清单，例如 AWS SAM 和 OpenWhisk Packaging Specification。

- 跨不同提供商的 **Serverless** 平台运行函数 **WorkFlow**。有许多使用场景，它们超出了触发单个函数的单个事件的范围，并且涉及到依次执行或并行执行，并由事件的不同组合 + 函数的返回值触发的多个函数的工作流。在工作流的上一步中。如果我们可以定义一组通用的构造，开发人员可以使用它们来定义其用例工作流，那么他们将是能够创建可在不同的 Serverless 平台上使用的工具。这些构造指定事件和函数之间的关系/交互作用，工作流中的函数之间的关系/交互作用以及如何将信息从一个函数传递给下一步骤的函数等。例如 AWS Step Function Constructs 和 Huawei Function Graph/Workflow Constructs。

- 建立开源工具的生态系统，以加速开发人员采用的速度，探索令人关注的领域，例如：
    - 仪器仪表（Instrumentation）
    - 可调试性（Debugability）

- 教育：提供一组设计模式，参考架构以及通用的新用户词汇表。

    - 术语表：以发布的形式维护术语表，并确保工作组文档始终使用这些术语。
    - 用例：维护用例列表，按通用模式分组，以创建共享的高级词汇表。支持以下目标：
        - 对于不使用 Serverless 平台的开发人员：加深对常见用例的了解，确定良好的切入点。
        - 对于 Serverless 提供者和库/框架作者，促进他们考虑共同需求。

    - CNCF GitHub 存储库中的示例应用程序和开源工具，偏重于强调互操作性方面或链接到每个提供商的外部资源。

- 提供有关如何评估 Serverless 架构相对于 CaaS 或 PaaS 的功能和非功能特性的指南。可以采用决策树的形式，也可以从 CNCF 项目系列中推荐一套工具。

- 开始一个流程，例如来自 Serverless WG 和 Storage WG 的 CNCF 输出（针对上面引用的建议文档），在 GitHub 中以 Markdown 文件保存，可以随着时间的推移对其进行协作维护，鉴于此领域的创新速度这一点尤其重要。

参考资料

本页用于收藏 Serverless 相关的参考资料。

PDF/PPT

- CNCF Serverless WG presentations - github.com
- ibm-opentech-ma - github.com

博客网站

- serverless.com
- thenewstack.io
- servicemesher.com

书籍/教程

- Knative 入门—构建基于 Kubernetes 的现代化 Serverless 应用 - servicemesher.com
- 开源技术 * IBM 微讲堂—Kubernetes 原生无服务器开源项目 Knative - developer.ibm.com

参考文章

- CNCF Serverless Whitepaper v1.0 - github.com
- Serverless Architectures - martinfowler.com
- Knative 是什么？为什么您需要关注它？- ibm.com
- 使用 Tekton Pipelines 部署 Knative 应用程序 - ibm.com