



Knative 云原生应用 开发指南



开启云原生时代 Serverless 之门





微信扫一扫关注
阿里巴巴云原生公众号



钉钉扫描二维码立即加入
Knative 交流群



钉钉微信扫一扫关注
云原生技术圈



目录

引言	v
快速入门	1
初识 Knative: 跨平台的 Serverless 编排框架	1
在阿里云上一键安装 Knative	10
手动安装 Knative	13
Serving Hello World	18
Eventing Hello World	25
Tekton Hello World	32
Serving 进阶	45
自动扩缩容 – Autoscaler	45
Serving 健康检查机制分析	54
流量灰度和版本管理	63
服务路由管理	75
WebSocket 和 gRPC 服务	83
Serving Client 介绍	88
Eventing 进阶	96
定义无处不在的事件 – CloudEvent	96
关于 Broker/Trigger 事件模型	101
事件注册机制 – Registry	110
Sequeue 解析	115
Parallel 解析	132

云原生开发实战	140
日志和监控告警	140
调用链管理	148
使用 GitHub 事件源	152
基于 Kafka 实现消息推送	158
基于 MNS 与 OSS 实现人脸图片识别	164
基于 APIGateway 打造生产级别的 Knative 服务	173
三步走！基于 Knative Serverless 技术实现一个短网址服务	185
基于 Knative Serverless 技术实现天气服务	195

引言



牛秋霖
花名：冬岛

阿里云容器平台技术专家，2014 年加入阿里，深度参与了阿里巴巴全面容器化、连续多年支持双十一容器化链路。专注于容器、Kubernetes、Service Mesh 和 Serverless 等云原生领域，致力于构建新一代 Serverless 平台。当前主导阿里云容器服务 Knative 相关工作。



李鹏
花名：元毅

阿里云容器平台高级开发工程师，2016 年加入阿里，深度参与了阿里巴巴全面容器化、连续多年支持双十一容器化链路。专注于容器、Kubernetes、Service Mesh 和 Serverless 等云原生领域，致力于构建新一代 Serverless 平台。当前负责阿里云容器服务 Knative 相关工作。

随着 Kubernetes 的崛起和 CNCF 的壮大，云原生的概念已经越来越深入人心。几乎到了如果不和云原生扯上点儿关系似乎就不好意思和别人对话的地步。那么到底什么是云原生？要搞清楚什么是云原生，我们首先要弄清楚云到底是怎么回事儿。接下来我们就从 IaaS 的演进和服务化的演进这两个脉络来看看云的发展趋势和云原生的含义。

IaaS 的演进

应用想要运行首先需要足够的计算资源，而物理机的性能在过去的十几年里始终保持着摩尔定律的速度在增长。这就导致单个应用根本无法充分利用整个物理机的资源。所以需要有一种技术解决资源利用率的问题。简单的想如果一个应用占不满整个物理机的资源，就多部署几个。但在同一个物理机下混合部署多个应用会有下面这些问题：

- 应用之间的端口冲突

如果一个应用占用了 80 端口，那么所有其他的应用就都不能使用 80 端口了，在同一台物理机上面每一个端口都是唯一的，不能被重复使用

- 应用的依赖冲突

比如 Python、Nodejs、Perl、PHP 等等这些脚本语言都需要提前安装好脚本解析器才能运行。而在同一台物理机上面如果对而写软件维护多个版本是一件非常困难的事情，一旦出错就可能导致所有应用崩溃。

- 资源隔离

不同应用使用的出资源不能有效的进行隔离，就容易相互影响。如果一个应用有 BUG 占光了内存会导致其他应用出问题。

- 运维操作相互影响

一个应用的管理员误操作可能会导致本机器上面的所有应用崩溃

虚拟机技术的出现就完美的解决了上述问题，通过虚拟机技术既可以在同一个物理机上面部署多个应用，又能保证应用之间的松耦合。

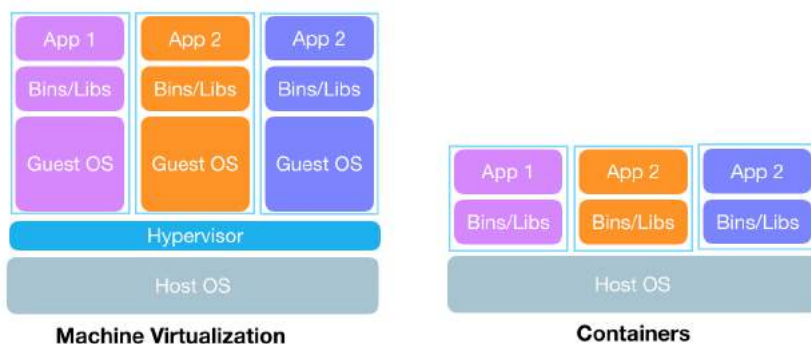
虚拟机技术的出现除了提升资源的使用效率还带来了另外一个变革：使得不可变基础设施成为了可能。

不可变基础设施 (Immutable Infrastructure) 是由 Chad Fowler 于 2013 年提出的一个很有前瞻性的构想：在这种模式中，任何基础设施的实例（包括服务器、容器等各种软硬件）一旦创建之后便成为一种只读状态，不可对其进行任何更改。如果需要修改或升级某些实例，唯一的方式就是创建一批新的实例以替换。

这意味着配置工作可以被低成本的重复使用，这就大大减少了配置管理工作的负担，让持续集成与持续部署过程变得更流畅。同时它也更易于应对部署环境间的差异及版本管理，包括在部署出错时可进行快速回滚 —— 只要旧版本的镜像文件还有备份，就可以快速地生成旧版本的实例进行替换。

随着虚拟机技术的普及，人们越来越发现虽然虚拟机可以提升物理机的使用效率，但是虚拟机本身也会占用巨大的开销。2013 年 Docker 的出现又在虚拟机的基

础上增进了一步。Docker 首先提出了镜像的概念，通过 Docker 镜像可以很容易的创建两个环境一样的运行时、这对开发测试、软件交付已经线上升级等都带来了巨大的便利性。另外 Docker 使用了内核的隔离技术，相比于虚拟机会占用的系统资源更少，所以效率更高。下图是基于虚拟机部署应用和基于容器化部署应用的对比关系。



从架构上来看 Docker 比虚拟化少了两层，取消了 hypervisor 层和 GuestOS 层，使用 Docker Engine 进行调度和隔离，所有应用共用主机操作系统，因此在体量上 Docker 较虚拟机更轻量级，在性能上优于虚拟化，接近裸机性能。

Docker 之后紧跟着 Kubernetes 就出现了，Kubernetes 之后的故事大家都知道了，现在 Kubernetes 已经成了云原生的标配了，并且现在各大云厂商都提供了 Kubernetes 服务。在 Kubernetes 模式下有一个明显的特征就是你无需关心你的应用实例 (Pod) 运行在哪里，因为这都是系统自动调度的。

从物理机到 VM 就已经无需关心真正的物理机了，应用程序只需要面向逻辑的虚拟机进行管理。到了容器时代还是有很多人直接在 VM 里面使用容器的。比如现在的公有云厂售卖的 VM 都可以跑 Docker。再次 Docker 到 Kubernetes 会发现又对底层的计算资源做了一层抽象。到了 Kubernetes 这一代就完全是根据需要自动分配资源，不需要提前占用物理机资源、或者 VM 资源，都是按需申请、按需调度的。

从 IaaS 虚拟化的这个维度我们发现云的发展趋势是在这两个维度上进行演化：

- 按需分配计算资源

对应用程序运行所在的节点越来越弱化，从虚拟化开始就都是逻辑层面的。到了人 Kubernetes 时代完全是按需申请和按需分配的。

- 不可变基础设施

应用程序只需要先设置好运行的基础环境，做一个镜像，然后用这个镜像到处运行。

服务化的演进

在单体应用时代，每一个单体应用都是一个大而全的功能集合。每个服务器运行的都是这个应用的完整服务。但随着业务的增长单体应用面对的问题也越来越多，例如：

- 开发效率变低
- 部署影响变大
- 可扩展性较差
- 技术选型成本高

所以出现了服务的拆分，也就是微服务。微服务可以实现：

- 每个微服务易于开发与维护，便于沟通与协作，很适合小团队敏捷开发与持续交付
- 每个微服务职责单一，高内聚、低耦合。
- 每个微服务能够独立开发、独立运行、独立部署
- 每个微服务之间是独立的，如果某个服务部署或者宕机，只会影响到当前服务，而不会对整个业务系统产生影响
- 每个微服务可以随着系统规模的不断扩大，面对海量用户和高并发，独立做水平扩展与垂直扩展

微服务虽好，但也不是银弹。微服务提升效率的同时也对设计和运维难度提出了

更高的要求，同时也带来了技术的复杂度。我们必须思考与解决分布式的复杂性、数据的一致性、服务的管理与运维、服务的自动化部署等解决方案。

我们知道应用交互的复杂性不会消失，只会换一种方式存在。这个原理也同样适用于软件架构。引入微服务并不会减少原来业务的开发的复杂性，只是把复杂性换一个方式存在而已。事实上，微服务通过拆分单体应用使其成为多个体积更小的服务来降低单个服务的复杂性，但从整体来看，这种方式有造成了存在大量的服务，而服务之间的相互调用也会增多，从而导致整个系统的复杂性增加不少。

繁多的微服务之间的关系需要自动化的方式维护：

- 服务注册和发现
- 限流、熔断
- 失败重试、超时时间等
- 负载均衡
- trace 等

Dubbo、Spring Cloud 这些微服务框架的出现很好的解决了微服务调用之间的复杂度问题。微服务框架能够解决服务通信和服务治理的复杂性，比如服务发现、熔断、限流、全链路追踪等挑战。

因为复杂性不会消失，只可能换一种形式存在。所以现代软件架构的核心任务之一就是定义基础设施与应用的边界，合理切分复杂性，尽量减少应用开发者直接面对的复杂性。也就是让开发者专注在核心价值创新上，而把一些与核心业务不相关的问题交给更合适的人或系统来解决。而这些人或系统就是中间件团队、IaaS 团队和运维团队等。随着微服务框架和基础平台自动化能力的提升，业务团队越来越容易的面向微服务框架编程，基于微服务框架管理自己的业务应用，这也是 DevOps 出现的前奏。

服务通信和服务治理本质是横向的系统级关系，是与业务逻辑正交的。但微服务框架如 HSF/Dubbo 或 Spring Cloud 以代码库的方式来封装这些能力。这些代码库被构建在应用程序本身中，随着应用一起发布和维护，其实现方式和生命周期与业务

逻辑耦合在一起的。微服务框架的升级会导致整个服务应用的重新构建和部署。所以不同的微服务框架是不能相互结合的。

如果是在一家公司内部还可以通过行政命令强制公司内使用一样的技术栈、使用一样的微服务框架进行编程。但如果提供的是云服务就不可能强制用户必须使用同一个微服务框架。更何况很多语言其实是没有微服务框架的。即便是那些有微服务框架的语言之间不同的微服务框架的设计也是完全不一样的。所以微服务框架在公有云面向大众提供服务的时候就显得很受限制。此时微服务框架代码级别内置集成的方式就显得很繁琐了，因此我们需要一个框架无关、语言无关甚至是协议无关的通用的微服务解决方案。为了解决上述挑战，社区提出了 Service Mesh (服务网格) 架构。它重新将服务治理能力下沉到基础设施，在服务的消费者和提供者两侧以独立进程的方式部署。这样既达到了去中心化的目的，保障了系统的可伸缩性；也实现了服务治理和业务逻辑的解耦，二者可以独立演进不相互干扰，提升了整体架构演进的灵活性；同时服务网格架构减少了对业务逻辑的侵入性，降低了多语言支持的复杂性。

Istio 通过 Sidecar 的方式把微服务治理的能力下沉到业务进程之外，从而解除了微服务治理和业务进程生命周期的强耦合关系。更重要的是 ServiceMesh 这一层抽象除了一套通用的操作标准，不会因不同的语言、不同的微服务框架而不同。

服务化的发展历程，是伴随着业务一起发展起来的，随着业务本身越来越复杂导致软件架构的复杂。而服务化解决这个问题的方法就是让复杂性下沉，通过层层抽象把复杂性下沉到底层系统。始终都要保持业务开发尽量聚焦在业务本身。

微服务发展历程的背后力量

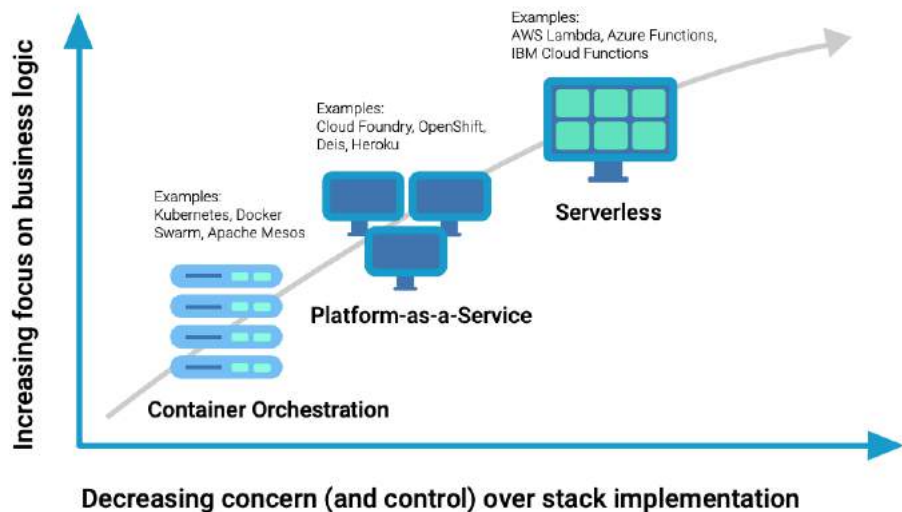
- 业务需求越来越多，业务复杂度不断增加
- 业务需求越来越多，人员角色越来越多，每一个角色都需要有自己独立决策的空间
- 激烈的竞争，需要业务更快速的迭代。也希望更尽可能的复用已有的模块

从应用服务化发展的这个维度我们发现云的发展有这样一个趋势：**业务开发的复杂度在不断的向基础平台下沉。**

Serverless

IaaS 的演进路线一个明显的趋势就是无服务器，这其实也是 Serverless 这个单词可以看到的字面含义。无服务器是 Serverless 的一个维度。

从服务化的演进路线可以看到一个明显的趋势是业务开发的复杂度在不断的向基础平台下沉。并且下沉的同时业务和基础平台的交互界面是越来越清晰、越来越标准了。在微服务框架的模式中各种微服务框架是各自为政的，但到了 ServiceMesh 的时代就都统一到 ServiceMesh 的标准。业务复杂度向基础平台下沉这是 Serverless 的另一个维度。



借用 [CNCF Serverless 白皮书](#)里面的一张图来说明一下这个关系。

纵轴表示业务应用越来越关注自己业务逻辑，业务应用只需要使用基础平台提供的能力就可以了，至于这些能力是怎么实现的不应该让业务应用关系细节。这是业务复杂度在不断的向基础平台下沉的过程，业务开发和基础平台的界面也在不断的上涨。

横轴表示业务应用对基础平台的资源占有模式的变化，从最初的独占模式到现在的按需分配。这是无服务器的发展过程，业务系统只需要在需要的时候拿到资源使用

即可，占有资源不是业务系统的核心诉求。

到这里我们看到云服务的三个趋势，这三个演进方向也就是云原生释放云红利的方式。

- 无服务器的趋势
- 业务逻辑不断的向基础平台下沉的趋势

Knative 作为一个 Serverless 编排框架其 Autoscaler 的自动扩缩容能力可以让应用在没有服务的时候缩容到零，完美的展示了无服务器的威力。Knative 是建立在 Istio 之上的，基于 Istio 可以提供完整的 ServiceMesh 能力。可以看到 Knative 完美的结合无服务器和服务复杂度下沉这两个维度。

Knative 作为最通用的 Serverless Framework，其中一个核心能力就是其简洁、高效的应用托管服务 (MicroPaaS)，这也是其支撑 Serverless 能力的基础。Knative 提供的应用托管服务 (MicroPaaS) 可以大大降低直接操作 Kubernetes 资源的复杂度和风险，提升应用的迭代和服务交付效率。当然作为 Serverless Framework 就离不开按需分配资源的能力，阿里云容器服务 Knative 可以根据您应用的请求量在高峰时期自动扩容实例数，当请求量减少以后自动缩容实例数，可以非常自动化的帮助您节省成本。

快速入门

初识 Knative: 跨平台的 Serverless 编排框架

Knative 是什么

Knative 是 Google 在 2018 的 Google Cloud Next 大会上发布的一款基于 Kubernetes 的 Serverless 框架。Knative 一个很重要的目标就是制定云原生、跨平台的 Serverless 编排标准。Knative 是通过整合容器构建 (或者函数)、工作负载管理 (和动态扩缩) 以及事件模型这三者来实现的这一 Serverless 标准。Knative 社区的主要贡献者有 Google、Pivotal、IBM、Red Hat。可见其阵容强大，CloudFoundry、OpenShift 这些 PAAS 提供商都在积极的参与 Knative 的建设。

Knative 出现的背景

在 Knative 之前社区已经有很多 Serverless 解决方案，如下所示这些：

- kubeless
- Fission
- OpenFaaS
- Apache OpenWhisk
- ...

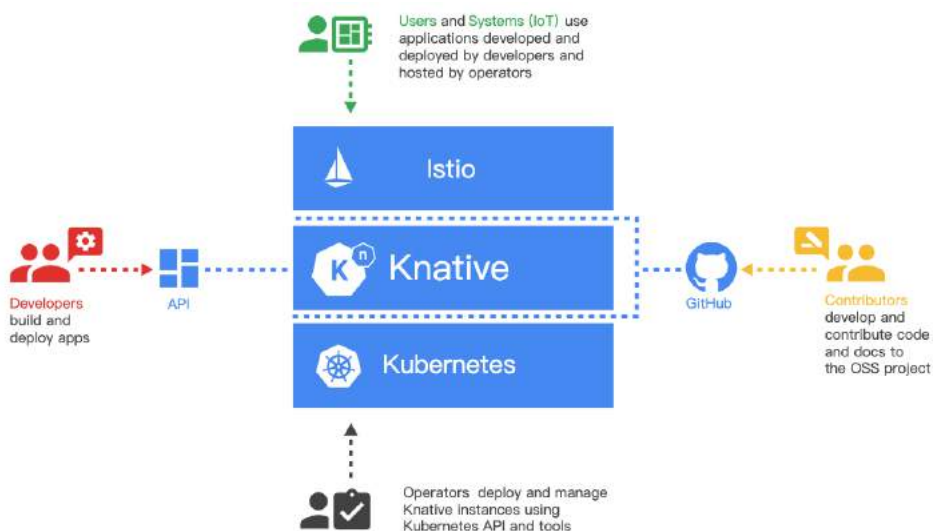
除了上面这些社区的开源解决方案以外各大云厂商也都有各自的 FAAS 产品的实现比如：

- AWS Lambda
- Google Cloud Functions
- Microsoft Azure Functions
- 阿里云的函数计算

业务代码部署到 Serverless 平台上就离不开源码的编译、部署和事件的管理。然而无论是开源的解决方案还是各公有云的 FAAS 产品大家的实现方式大家都各不相同，缺乏统一的标准导致市场呈现碎片化。因此无论选择哪一个方案都面临供应商绑定的风险。

没有统一的标准、市场的碎片化这对云厂商来说用户 Serverless 上云就比较困难，对于 PAAS 提供商来说很难做一个通用的 PAAS 平台给用户使用。基于这样的背景 Google 牵头联合 Pivotal、IBM、Red Hat 等发起了 Knative 项目。

我们看一下在 Knative 体系下各个角色的协作关系：



- Developers

Serverless 服务的开发人员可以直接使用原生的 Kubernetes API 基于 Knative 部署 Serverless 服务

- Contributors

主要是指社区的贡献者

- Operators

Knative 可以被集成到任何支持的环境中，比如：云厂商、或者企业内部。目前 Knative 是基于 Kubernetes 来实现的，有 Kubernetes 的地方就可以部署 Knative

- Users

终端用户通过 Istio 网关访问服务，或者通过事件系统触发 Knative 中的 Serverless 服务

核心组件

Knative 主要由 Serving 和 Eventing 核心组件构成。除此之外使用 Tekton 作为 CI/CD 构建工具。下面让我们来分别介绍一下这三个核心组件。

Serving

Knative 作为 Serverless 框架最终是用来提供服务的，那么 Knative Serving 应运而生。

Knative Serving 构建于 Kubernetes 和 Istio 之上，为 Serverless 应用提供部署和服务支持。其特性如下：

- 快速部署 Serverless 容器
- 支持自动扩缩容和缩到 0 实例
- 基于 Istio 组件，提供路由和网络编程
- 支持部署快照

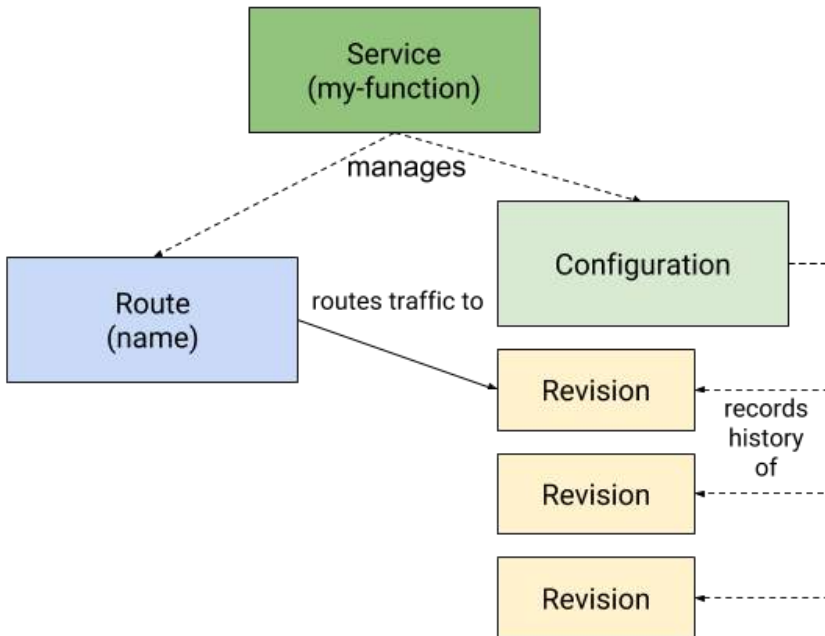
Knative Serving 中定义了以下 CRD 资源：

- Service: 自动管理工作负载整个生命周期。负责创建 Route、Configuration

以及 Revision 资源。通过 Service 可以指定路由流量使用最新的 Revision 还是固定的 Revision

- Route: 负责映射网络端点到一个或多个 Revision。可以通过多种方式管理流量。包括灰度流量和重命名路由
- Configuration: 负责保持 Deployment 的期望状态，提供了代码和配置之间清晰的分离，并遵循应用开发的 12 要素。修改一次 Configuration 产生一个 Revision
- Revision: Revision 资源是对工作负载进行的每个修改的代码和配置的时间点快照。Revision 是不可变对象，可以长期保留

资源关系图:

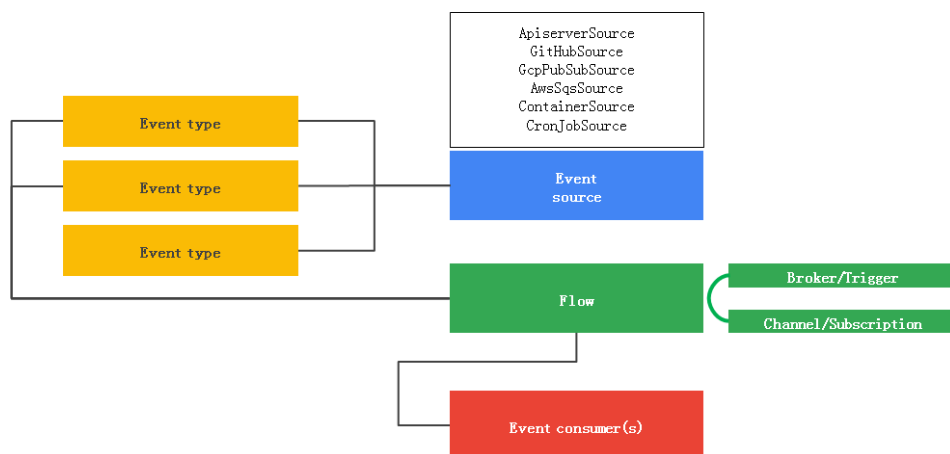


Eventing

Knative Eventing 旨在满足云原生开发中通用需求，以提供可组合的方式绑定事件源和事件消费者。其设计目标:

- Knative Eventing 提供的服务是松散耦合，可独立开发和部署。服务可跨平台使用（如 Kubernetes, VMs, SaaS 或者 FaaS）
- 事件的生产者和事件的消费者是相互独立的。任何事件的生产者（事件源）可以先于事件的消费者监听之前产生事件，同样事件的消费者可以先于事件产生之前监听事件
- 支持第三方的服务对接该 Eventing 系统
- 确保跨服务的互操作性

事件处理示意图：



如上图所示，Eventing 主要由事件源（Event Source）、事件处理（Flow）以及事件消费者（Event Consumer）三部分构成。

事件源（Event Source）

当前支持以下几种类型的事件源：

- ApiserverSource：每次创建或更新 Kubernetes 资源时，ApiserverSource 都会触发一个新事件

- `GitHubSource`: GitHub 操作时, `GitHubSource` 会触发一个新事件
- `GcpPubSubSource`: GCP 云平台 Pub/Sub 服务会触发一个新事件
- `AwsSqsSource`: Aws 云平台 SQS 服务会触发一个新事件
- `ContainerSource`: `ContainerSource` 将实例化一个容器, 通过该容器产生事件
- `CronJobSource`: 通过 `CronJob` 产生事件
- `KafkaSource`: 接收 Kafka 事件并触发一个新事件
- `CamelSource`: 接收 Camel 相关组件事件并触发一个新事件

事件接收 / 转发 (Flow)

当前 Knative 支持如下事件接收处理:

- 直接事件接收

通过事件源直接转发到单一事件消费者。支持直接调用 `Knative Service` 或者 `Kubernetes Service` 进行消费处理。这样的场景下, 如果调用的服务不可用, 事件源负责重试机制处理。

- 通过事件通道 (Channel) 以及事件订阅 (Subscriptions) 转发事件处理

这样的情况下, 可以通过 Channel 保证事件不丢失并进行缓冲处理, 通过 Subscriptions 订阅事件以满足多个消费端处理。

- 通过 `brokers` 和 `triggers` 支持事件消费及过滤机制

从 v0.5 开始, `Knative Eventing` 定义 `Broker` 和 `Trigger` 对象, 实现了对事件进行过滤 (亦如通过 `ingress` 和 `ingress controller` 对网络流量的过滤一样)。

通过定义 `Broker` 创建 Channel, 通过 `Trigger` 创建 Channel 的订阅 (subscription), 并产生事件过滤规则。

事件消费者 (Event Consumer)

为了满足将事件发送到不同类型的服务进行消费, `Knative Eventing` 通过多个 k8s 资源定义了两个通用的接口:

- Addressable 接口提供可用于事件接收和发送的 HTTP 请求地址，并通过 status.address.hostname 字段定义。作为一种特殊情况，Kubernetes Service 对象也可以实现 Addressable 接口
- Callable 接口接收通过 HTTP 传递的事件并转换事件。可以按照处理来自外部事件源事件的相同方式，对这些返回的事件做进一步处理

当前 Knative 支持通过 Knative Service 或者 Kubernetes Service 进行消费事件。

另外针对事件消费者，如何事先知道哪些事件可以被消费？Knative Eventing 在最新的 0.6 版本中提供 Registry 事件注册机制，这样事件消费者就可以事先通过 Registry 获得哪些 Broker 中的事件类型可以被消费。

Tekton

Tekton 是一个功能强大且灵活的 Kubernetes 原生 CI/CD 开源框架。通过抽象底层实现细节，用户可以跨多云平台和本地系统进行构建、测试和部署。具有组件化、声明式、可复用及云原生的特点。



- Tekton 是云原生的 – Cloud Native
 - 在 Kubernetes 上运行

- 将 Kubernetes 集群作为第一选择
- 基于容器进行构建
- Tekton 是解耦的 – Decoupled
 - Pipeline 可用于部署到任何 k8s 集群
 - 组成 Pipeline 的 Tasks 可以轻松地独立运行
 - 像 git repos 这样的 Resources 可以在运行期间轻松切换
- Tekton 是类型化的 – Typed
 - 类型化资源的概念意味着对于诸如的资源 Image，可以轻松地将实现切换（例如，使用 kaniko vs buildkit 进行构建）

Knative 的优势

Knative 一方面基于 Kubernetes 实现 Serverless 编排，另外一方面 Knative 还基于 Istio 实现服务的接入、服务路由的管理以及度发布等功能。Knative 是在已有的云原生基础之上构建的，有很好的社区基础。Knative 一经开源就得到了大家的追捧，其中的主要缘由有：

- Knative 的定位不是 PAAS 而是一个通用的 Serverless 编排框架，大家可以基于此框架构建自己的 Serverless PAAS
- Knative 对于 Serverless 架构的设计是标准先行。比如之前的 FAAS 解决方案每一个都有一套自己的事件标准，相互之间无法通用。而 Knative 首先提出了 CloudEvent 这一标准，然后基于此标准进行设计
- 完备的社区生态：Kubernetes、ServiceMesh 等社区都是 Knative 的支持者
- 跨平台：因为 Knative 是构建在 Kubernetes 之上的，而 Kubernetes 在不同的云厂商之间几乎可以提供通用的标准。所以 Knative 也就实现了跨平台的能力，没有供应商绑定的风险
- 完备的 Serverless 模型设计：和之前的 Serverless 解决方案相比 Knative 各方面的设计更加完备。比如：

- 事件模型非常完备，有注册、订阅、以及外部事件系统的接入等等一整套的设计
- 比如从源码到镜像的构建
- 比如 Serving 可以做到按比例的比例发布

总结

相信通过上面的介绍，大家对 Knative 有了初步的认识。Knative 使用 Tekton 提供云原生“从源代码到容器”的镜像构建能力，通过 Serving 部署容器并提供通用的服务模型，同时以 Eventing 提供事件全局订阅、传递和管理能力，实现事件驱动。这就是 Knative 呈现给我们的标准 Serverless 编排框架。

在阿里云上一键安装 Knative

本文介绍一下如何在阿里云容器服务中一键安装 Knative。

前提条件

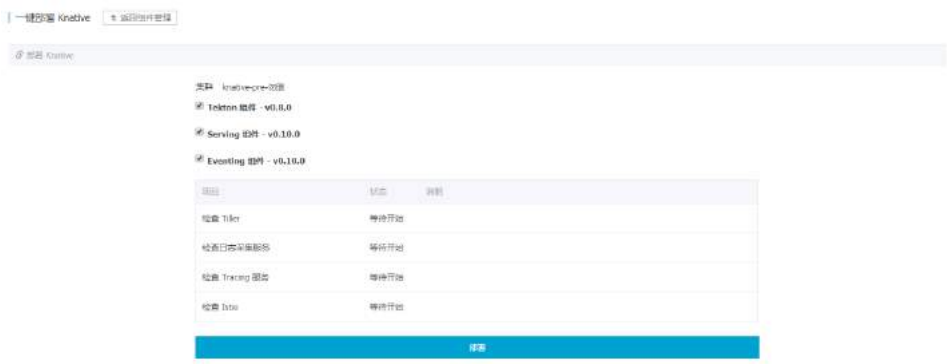
- 您已经成功创建一个 Kubernetes 集群，参见[创建 Kubernetes 集群](#)
- 部署成功 Istio，参见[部署 Istio](#)
- 仅支持 Kubernetes 版本 1.14 及以上的集群。只支持**标准托管**以及**标准专有** Kubernetes 集群
- 一个集群中，worker 节点数量需要大于等于 3 个，保证资源充足可用

操作步骤

1. 登录 [容器服务管理控制台](#)
2. 单击左侧导航栏中的 **Knative> 组件管理**，进入**组件管理**页面
3. 点击**一键部署**按钮



4. 选择需要安装的 Knative 组件 (Tekton 组件、Serving 组件和 Eventing 组件)，点击**部署**

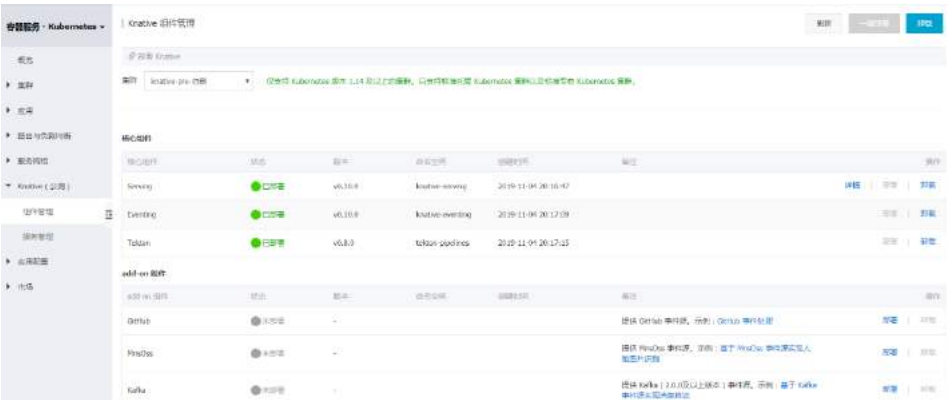


执行结果

部署完成之后，可以看到部署结果信息。



点击进入组件管理，查看组件信息。



总结

在阿里云容器服务中提供了自定义安装 Knative 组件以及 addon 组件，让你轻松安装 Knative。

手动安装 Knative

本章主要介绍如何在已有 Kubernetes 集群上执行 Knative 的自定义安装。Knative 的模块化组件可以允许您安装所需的组件。

准备工作

- 本安装操作中的步骤 bash 适用于 MacOS 或 Linux 环境。对于 Windows，某些命令可能需要调整。
- 本安装操作假定您具有现有的 Kubernetes 集群，可以在其上轻松安装和运行 Alpha 级软件。
- Knative 需要 Kubernetes 集群 v1.14 或更高版本，以及可兼容 kubectl。

安装 Istio

Knative 依赖 Istio 进行流量路由和入口。您可以选择注入 Istio sidecar 并启用 Istio 服务网格，但是并非所有 Knative 组件都需要它。

如果您的云平台提供了托管的 Istio 安装，则建议您以这种方式安装 Istio，除非您需要自定义安装功能。

如果您希望手动安装 Istio，或者云提供商不提供托管的 Istio 安装，或者您要使用 Minkube 或类似的本地安装 Knative，请参阅[《安装 Istio for Knative》指南](#)。

注意：可以使用 [Ambassador](#) 和 [Gloo](#) 替代 Istio。

安装 Knative 组件

每个 Knative 组件必须单独安装。您可以根据需要决定安装哪些组件和内置监控插件。

注意：如果首次尝试安装失败，请尝试重新运行命令。很可能在第二次尝试中成功。

选择 Knative 安装文件

可以使用以下 Knative 安装文件：

- Serving 部分
 - <https://github.com/knative/serving/releases/download/{{ < 版本 > }}/serving.yaml>
 - <https://github.com/knative/serving/releases/download/{{ < 版本 > }}/serving-cert-manager.yaml>
- 内置监控插件
 - <https://github.com/knative/serving/releases/download/{{ < 版本 > }}/monitoring.yaml>
 - <https://github.com/knative/serving/releases/download/{{ < 版本 > }}/monitoring-logs-elasticsearch.yaml>
 - <https://github.com/knative/serving/releases/download/{{ < 版本 > }}/monitoring-metrics-prometheus.yaml>
 - <https://github.com/knative/serving/releases/download/{{ < 版本 > }}/monitoring-tracing-jaeger.yaml>
 - <https://github.com/knative/serving/releases/download/{{ < 版本 > }}/monitoring-tracing-jaeger-in-mem.yaml>
 - <https://github.com/knative/serving/releases/download/{{ < 版本 > }}/monitoring-tracing-zipkin.yaml>
 - <https://github.com/knative/serving/releases/download/{{ < 版本 > }}/monitoring-tracing-zipkin-in-mem.yaml>
- Eventing 组件
 - <https://github.com/knative/eventing/releases/download/{{ < 版本 > }}/release.yaml>
 - <https://github.com/knative/eventing/releases/download/{{ < 版本 > }}>

eventing.yaml

- [https://github.com/knative/eventing/releases/download/{{ < 版本 > }}/](https://github.com/knative/eventing/releases/download/{{ < 版本 > }}/in-memory-channel.yaml)

in-memory-channel.yaml

- Eventing 事件源

- <https://github.com/knative/eventing-contrib/releases/download/{{ < 版本 > }}/github.yaml>

- <https://github.com/knative/eventing-contrib/releases/download/{{ < 版本 > }}/camel.yaml>

- <https://github.com/knative/eventing-contrib/releases/download/{{ < 版本 > }}/gcppubsub.yaml>

- <https://github.com/knative/eventing-contrib/releases/download/{{ < 版本 > }}/kafka.yaml>

- <https://github.com/knative/eventing-contrib/releases/download/{{ < 版本 > }}/kafka-channel.yaml>

安装 Knative

1. 如果要从 Knative 0.3.x 升级，请执行以下操作：将 domain 和静态 IP 地址更新为与 istio-ingressgateway 关联，而不是 knative-ingressgateway。然后运行以下命令清理剩余的资源：

```
kubectl delete svc knative-ingressgateway -n istio-system
kubectl delete deploy knative-ingressgateway -n istio-system
```

如果安装了 Knative Eventing Sources 组件，则还需要在升级之前删除以下资源：

```
kubectl delete statefulset/controller-manager -n knative-sources
```

2. 要安装 Knative 组件或插件，请在 kubectl apply 命令中指定文件名。为了防止由于资源安装顺序导致安装失败，请首先运行带有该 -l knative.dev/

crd-install=true 标志的安装命令，然后再次运行没有 --selector 标志的安装命令。

示例安装命令：

- 如果需要使用内置监控插件，安装 Knative Serving 组件，请运行以下命令：
 - 仅安装 CRD：

```
kubectl apply --selector knative.dev/crd-install=true \
  --filename https://github.com/knative/serving/releases/download/{{ < 版本 >
  }}/serving.yaml \
  --filename https://github.com/knative/serving/releases/download/{{ < 版本 >
  }}/monitoring.yaml
```

- 删除 --selector knative.dev/crd-install=true 标志，然后运行命令以安装 Serving 组件和监控插件：

```
kubectl apply --filename https://github.com/knative/serving/releases/download/
  {{ < 版本 > }}/serving.yaml \
  --filename https://github.com/knative/serving/releases/download/{{ < 版本 >
  }}/monitoring.yaml
```

- 如果不使用内置监控插件的情况下安装所有 Knative 组件，请运行以下命令。
 - 仅安装 CRD：

```
kubectl apply --selector knative.dev/crd-install=true \
  --filename https://github.com/knative/serving/releases/download/{{ < version
  > }}/serving.yaml \
  --filename https://github.com/knative/eventing/releases/download/{{ <
  version > }}/release.yaml
```

- 删除 --selector knative.dev/crd-install=true 标志，然后运行命令以安装所有 Knative 组件，包括 Eventing 资源：

```
kubectl apply --filename https://github.com/knative/serving/releases/download/
  {{ < 版本 > }}/serving.yaml \
  --filename https://github.com/knative/eventing/releases/download/{{ < 版本 >
  }}/release.yaml
```

3. 根据选择安装的内容，通过运行以下一个或多个命令来查看安装状态：

```
kubectl get pods --namespace knative-serving  
kubectl get pods --namespace knative-eventing
```

4. 如果安装了内置监控插件，请运行以下命令：

```
kubectl get pods --namespace knative-monitoring
```

其它

考虑到国内用户有可能拉取不断外部镜像，安装文件可参考：<https://github.com/knative-sample/knative-release/tree/v0.10.0>

Serving Hello World

通过前面两章的学习你已经掌握了很多 Knative 的理论知识，基于这些知识你应该对 Knative 是谁、它来自哪里以及它要做什么有了一定的认识。可是即便如此你可能还是会有一种犹抱琵琶半遮面，看不清真容的感觉，这就好比红娘拿姑娘的 100 张生活照给你看也不如你亲自去见一面。按常理出牌，一般到这个阶段就该 Hello World 出场了。本篇文章就通过一个 Hello World 和 Knative 来一个“约会”，让你一睹 Knative 这位白富美的真容。

Serverless 一个核心思想就是按需分配，那么 Knative 是如何实现按需分配的呢？另外在前面的文章中你已经了解到 Knative Serving 在没有流量的时候是可以把 Pod 缩容到零的。接下来就通过一些例子体验一下 Knative 缩容到零和按需自动扩缩容的能力。

部署 helloworld-go 示例

Knative 官方给出了好几种语言的 [Helloworld 示例](#)，这些不同的语言其实只是编译镜像的 Dockerfile 有所不同，做好镜像之后的使用方式没什么差异。本例以 go 的 Hello World 为例进行演示。官方给出的例子都是源码，需要编译长镜像才能使用。为了你验证方便我已经提前编译好了一份镜像 `registry.cn-hangzhou.aliyuncs.com/knative-sample/simple-app:07`，你可以直接使用。

首先编写一个 Knative Service 的 yaml 文件 `helloworld-go.yaml`，内容如下：

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: helloworld-go
spec:
  template:
    metadata:
      labels:
```

```

    app: helloworld-go
    annotations:
      autoscaling.knative.dev/target: "10"
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/helloworld-go:160e4dc8
          ports:
            - name: http1
              containerPort: 8080
          env:
            - name: TARGET
              value: "World"

```

注意其中 `autoscaling.knative.dev/target: "10"` 这个 Annotation 是设置每一个 Pod 的可处理并发请求数 10，Knative KPA 自动伸缩的时候会根据当前总请求的并发数和 `autoscaling.knative.dev/target` 自动调整 Pod 的数量，从而达到自动扩缩的目的。更多的策略信息我会在后续的文章中一一介绍。

现在使用 `kubectl` 命令把 yamI 提交到 Kubernetes 中：

- 部署 `helloworld-go`

```

└─# kubectl apply -f helloworld-go.yaml
service.serving.knative.dev/helloworld-go created

```

- 查看 `helloworld-go` pod

```

└─# kubectl get pod
NAME
READY   STATUS      RESTARTS   AGE
helloworld-go-7n4dm-deployment-65cb6d9bf-9njqx
2/2     Running    0          8s

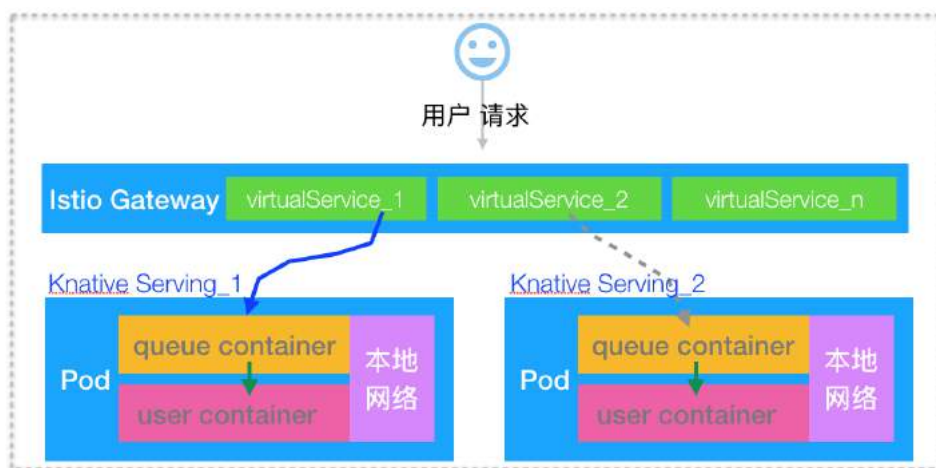
```

到此 `helloworld-go` 已经运行起来了，接下来访问一下 `helloworld-go` 这个服务吧。

访问 `helloworld-go` 示例

在访问 `helloworld-go` 之前我要先来介绍一下在 Knative 模型中流量是怎么进

来的。Knative Service 和 Kubernetes 原生的 Deployment 不一样，Knative 不会创建 Loadbalance 的 Service，也不能创建 NodePort 类型的 Service，所以不能通过 SLB 或者 NodePort 访问。只能通过 ClusterIP 访问。而 ClusterIP 是不能直接对外暴露的，所以必须经过 Gateway 才能把用户的流量接入进来。本例就是使用 Istio 的 Gateway 承接 Knative 的南北流量（进和出）。如下图所示是 Knative 模型中流量的转发路径。用户发起的请求首先会打到 Gateway 上面，然后 Istio 通过 VirtualService 再把请求转发到具体的 Revision 上面。当然用户的流量还会经过 Knative 的 queue 容器才能真正转发到业务容器，关于这方面的细节我在后续的文章再进行详细的介绍。



所以想要访问 Knative 的服务首先要获取 Gateway 的 IP 地址，可以通过如下方式获取 Gateway 的 IP：

```
# kubectl get svc istio-ingressgateway --namespace istio-system --output
jsonpath="{.status.loadBalancer.ingress[*].ip}"
39.106.232.122
```

前面也介绍了 Gateway 是通过 VirtualService 来进行流量转发的，这就要求访问者要知道目标服务的名字才行（域名），所以要先获取 helloworld-go 的域名，注

意下面这条命令中的 `${SVC_NAME}` 需要替换成 `helloworld-go`，这个名字必须要和 Knative Service 的名字一致，因为每一个 Service 都有一个唯一的名字。

```
└─# kubectl get route ${SVC_NAME} --output jsonpath="{.status.domain}"
helloworld-go.default.knative.kuberun.com
```

至此你已经拿到 IP 地址和 Hostname，可以通过 `curl` 直接发起请求：

```
└─# curl -H "Host: helloworld-go.default.knative.kuberun.com"
"http://39.106.232.122"
Hello World!
```

缩容到零

刚刚部署完 Service 的时候 Knative 默认会创建出一个 Pod 提供服务，如果你超过 90 秒没有访问 `helloworld-go` 这个服务那么这个 Pod 就会自动删除，此时就是缩容到零了。现在看一下 Pod 情况，你可能会发现没有 Pod。

```
└─# kubectl get pod -o wide
No resources found.
```

```
└─# time curl -H "Host: helloworld-go.default.knative.kuberun.com"
"http://39.106.232.122"
Hello World!
real    0m2.775s
user    0m0.007s
sys 0m0.007s
```

注意结果中，这面这一段：

```
real    0m2.775s
user    0m0.007s
sys 0m0.007s
```

`real 0m2.775s` 意思意思是 `curl` 请求执行一共消耗了 2.775s，也就是说 Knative 从零到 1 扩容 + 启动容器再到服务响应请求总共消耗了 2.775s（我之前的测试导致在 Node 上面缓存了镜像，所以没有拉镜像的时间）。可以看出来这个速度

还是很快的。

再看一下 pod 数量，你会发现此时 Pod 自动扩容出来了。并且 Pod 数量为零时发起的请求并没有拒绝链接。

```
└─ # kubectl get pod
NAME
READY   STATUS    RESTARTS   AGE
helloworld-go-p9w6c-deployment-5dfdb6bccb-gjfxj   2/2
Running   0           31s
```

按需分配，自动扩缩

接下来再测试一下 Knative 按需扩容的功能。使用社区提供的 hey 进行测试。hey 有 Windows、Linux 和 Mac 的二进制可以在这里下载。

使用这个命令测试之前需要在本机进行 Host 绑定，对于 helloworld-go 来说要把 helloworld-go 的域名绑定到 Istio Gateway 的 IP 上，/etc/hosts 添加如下配置：

```
39.106.232.122 helloworld-go.default.knative.kuberun.com
```

如下所示 这条命令的意思是：

- -z 30s 持续测试 30s
- -c 50 保持每秒 50 个请求

测试结果如下：

```
└─ # hey -z 30s -c 50 "http://helloworld-go.default.knative.kuberun.com/" &&
kubectl get pods
Summary:
  Total:      30.0388 secs
  Slowest:   2.6178 secs
  Fastest:   0.0276 secs
  Average:   0.0434 secs
  Requests/sec: 1152.3116
  Total data: 449982 bytes
  Size/request: 13 bytes
  Response time histogram:
    0.028 [1] |
```


NAME			READY
STATUS	RESTARTS	AGE	
helloworld-go-7n4dm-deployment-65cb6d9bf-8x6b5			2/2
Running	0	29s	
helloworld-go-7n4dm-deployment-65cb6d9bf-dvrn5			2/2
Running	0	27s	
helloworld-go-7n4dm-deployment-65cb6d9bf-f7db8			2/2
Running	0	30s	
helloworld-go-7n4dm-deployment-65cb6d9bf-fsn2z			2/2
Running	0	29s	
helloworld-go-7n4dm-deployment-65cb6d9bf-rmrh2			2/2
Running	0	29s	

可以看到此时 Knative 自动扩容出来了 5 个 Pod 处理请求。

总结

至此你已经完成了和 Knative Serving 的首次约会，也看到了这位白富美的真容。通过本篇文章你应该掌握以下几点：

- 理解 Knative 从零到一的含义，并且能够基于 helloworld-go 例子演示这个过程
- 理解 Knative 按需扩缩容的含义，并且能够基于 autoscale-go 例子演示这个过程
- 理解 Knative KPA 按需扩容的原理

参考文档

示例代码：<https://github.com/knative-sample/helloworld-go/tree/b1.0>

Eventing Hello World

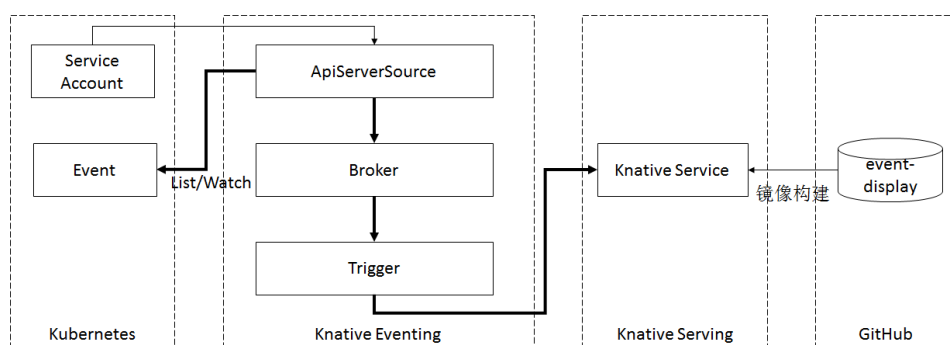
基于事件驱动是 Serverless 的核心功能之一，通过事件驱动服务，满足了用户按需付费 (Pay-as-you-go) 的需求。在之前的文章中我们介绍过 Knative Eventing 由事件源、事件处理模型和事件消费 3 个主要部分构成，那么事件如何通过这 3 个组件产生、处理以及消费呢？本文通过 Hello World 示例带你初探 Eventing。

前置准备

- Knative 版本 ≥ 0.5
- 已安装 Serving 组件
- 已安装 Eventing 组件

操作步骤

先看一下 Kubernetes Event Source 示例处理流程，如图所示：



接下来介绍一下各个阶段如何进行操作处理。

创建 Service Account

为 ApiServerSource 创建 Service Account, 用于授权 ApiServerSource 获取 Kubernetes Events。

serviceaccount.yaml 如下:

```
apiVersion: v1
kind: ServiceAccount
metadata:
  name: events-sa
  namespace: default
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: event-watcher
rules:
- apiGroups:
  - ""
  resources:
  - events
  verbs:
  - get
  - list
  - watch
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: k8s-ra-event-watcher
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: event-watcher
subjects:
- kind: ServiceAccount
  name: events-sa
  namespace: default
```

执行如下操作:

```
kubectl apply --filename serviceaccount.yaml
```

创建 Event Source

Knative Eventing 中 通过 Event Source 对接第三方系统产生统一的事件类型。当前支持 ApiServerSource, GitHub 等多种数据源。这里我们创建一个 ApiServerSource 事件源用于接收 Kubernetes Events 事件并进行转发。k8s-events.yaml 如下:

```
apiVersion: sources.eventing.knative.dev/v1alpha1
kind: ApiServerSource
metadata:
  name: testevents
  namespace: default
spec:
  serviceAccountName: events-sa
  mode: Resource
  resources:
  - apiVersion: v1
    kind: Event
sink:
  apiVersion: eventing.knative.dev/v1alpha1
  kind: Broker
  name: default
```

这里通过 sink 参数指定事件接收方, 支持 Broker 和 k8s service。

执行命令:

```
kubectl apply --filename k8s-events.yaml
```

创建 Knative Service

首先构建你的事件处理服务, 可以参考 [knative-sample/event-display](https://github.com/knative-samples/event-display) 开源项目。

这里的 Service 服务仅把接收到的事件打印出来, 处理逻辑如下:

```
package main
import (
    "context"
    "fmt"
    "log"
    cloudevents "github.com/cloudevents/sdk-go"
```

```

    "github.com/knative-sample/event-display/pkg/kncloudevents"
)
/*
Example Output:
* cloudevents.Event:
Validation: valid
Context Attributes,
  SpecVersion: 0.2
  Type: dev.knative.eventing.samples.heartbeat
  Source: https://github.com/knative/eventing-sources/cmd/heartbeats/#local/demo
  ID: 3d2b5a1f-10ca-437b-a374-9c49e43c02fb
  Time: 2019-03-14T21:21:29.366002Z
  ContentType: application/json
  Extensions:
    the: 42
    beats: true
    heart: yes
Transport Context,
  URI: /
  Host: localhost:8080
  Method: POST
Data
{
  {
    "id":162,
    "label":""
  }
}
*/
func display(event cloudevents.Event) {
    fmt.Printf("Hello World: \n")
    fmt.Printf("cloudevents.Event\n%s", event.String())
}
func main() {
    c, err := kncloudevents.NewDefaultClient()
    if err != nil {
        log.Fatal("Failed to create client, ", err)
    }
    log.Fatal(c.StartReceiver(context.Background(), display))
}

```

通过上面的代码，可以轻松构建你自己的镜像。镜像构建完成之后，接下来可以创建一个简单的 Knative Service，用于消费 ApiServerSource 产生的事件。

service.yaml 示例如下：

```

apiVersion: serving.knative.dev/v1alpha1
kind: Service

```



```

metadata:
  name: event-display
  namespace: default
spec:
  template:
    spec:
      containers:
        - image: {yourrepo}/{yournamespace}/event-display:latest

```

执行命令：

```
kubectl apply --filename service.yaml
```

创建 Broker

在所选命名空间下，创建 default Broker。假如选择 default 命名空间，执行操作如下。

```
kubectl label namespace default knative-eventing-injection=enabled
```

这里 Eventing Controller 会根据设置 knative-eventing-injection=enabled 标签的 namespace，自动创建 Broker。并且使用在 webhook 中默认配置的 ClusterChannelProvisioner (in-memory)。

创建 Trigger

Trigger 可以理解为 Broker 和 Service 之间的过滤器，可以设置一些事件的过滤规则。这里为默认的 Broker 创建一个最简单的 Trigger，并且使用 Service 进行订阅。trigger.yaml 示例如下：

```

apiVersion: eventing.knative.dev/v1alpha1
kind: Trigger
metadata:
  name: testevents-trigger
  namespace: default
spec:
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1alpha1

```

```
kind: Service
name: event-display
```

执行命令：

```
kubectl apply --filename trigger.yaml
```

注意：如果没有使用默认的 Broker，在 Trigger 中可以通过 spec.broker 指定 Broker 名称。

验证

执行如下命令，生成 k8s events。

```
kubectl run busybox --image=busybox --restart=Never -- ls
kubectl delete pod busybox
```

可以通过下述方式查看 Knative Service 是否接收到事件。

```
kubectl get pods
kubectl logs -l serving.knative.dev/service=event-display -c user-container
```

日志输出类似下面，说明已经成功接收事件。

```
Hello World:
* ☐ CloudEvent: valid  ✓
Context Attributes,
  SpecVersion: 0.2
  Type: dev.knative.apiserver.resource.add
  Source: https://10.39.240.1:443
  ID: 716d4536-3b92-4fbb-98d9-14bfcf94683f
  Time: 2019-05-10T23:27:06.695575294Z
  ContentType: application/json
  Extensions:
    knativehistory: default-broker-b7k2p-channel-z7mqg.default.svc.cluster.local
    subject: /apis/v1/namespaces/default/events/busybox.159d7608e3a3572c
Transport Context,
  URI: /
  Host: auto-event-display.default.svc.cluster.local
  Method: POST
Data,
{
  "apiVersion": "v1",
```

```

    "count": 1,
    "eventTime": null,
    "firstTimestamp": "2019-05-10T23:27:06Z",
    "involvedObject": {
      "apiVersion": "v1",
      "fieldPath": "spec.containers{busybox}",
      "kind": "Pod",
      "name": "busybox",
      "namespace": "default",
      "resourceVersion": "28987493",
      "uid": "1efb342a-737b-11e9-a6c5-42010a8a00ed"
    },
    "kind": "Event",
    "lastTimestamp": "2019-05-10T23:27:06Z",
    "message": "Started container",
    "metadata": {
      "creationTimestamp": "2019-05-10T23:27:06Z",
      "name": "busybox.159d7608e3a3572c",
      "namespace": "default",
      "resourceVersion": "506088",
      "selfLink": "/api/v1/namespaces/default/events/busybox.159d7608e3a3572c",
      "uid": "2005af47-737b-11e9-a6c5-42010a8a00ed"
    },
    "reason": "Started",
    "reportingComponent": "",
    "reportingInstance": "",
    "source": {
      "component": "kubelet",
      "host": "gke-knative-auto-cluster-default-pool-23c23c4f-xdj0"
    },
    "type": "Normal"
  }
}

```

总结

相信通过上面的例子你已经了解了 Knative Eventing 如何产生事件、处理事件以及消费事件。当然你可以自己定义一个事件消费服务，来处理事件。

| Tekton Hello World



Tekton 作为 Knative Build 模块的升级版，提供了更丰富的功能，可以适用更多的场景。如果你知道 Knative Build 是什么相信你理解起 Tekton 就是很容易的一件事了。

- Knative Build 对自己的一句话概述是: A Kubernetes-native Build resource.
- Tekton 对自己的一句话概述是: A K8s-native Pipeline resource. <https://tekton.dev>

可以看到两者的定位非常相近，而且在功能上 Tekton 的设计更加的丰富、完整，这也是社区最终采用 Tekton 替代 Build 的原因。接下来我们就看一下 Tekton 的核心概念。

Tekton 极速入门

Tekton 主要由如下五个核心概念组成：

- Task
- TaskRun
- Pipeline
- PipelineRun

- PipelineResource

这五个概念每一个都是以 CRD 的形式提供服务的，下面分别简述一下这五个概念的含义。

Task

Task 就是一个任务执行模板，之所以说 Task 是一个模板是因为 Task 定义中可以包含变量，Task 在真正执行的时候需要给定变量的具体值。如果把 Tekton 的 Task 有点儿类似于定义一个函数，Task 通过 inputs.params 定义需要哪些入参，并且每一个入参还可以指定默认值。Task 的 steps 字段表示当前 Task 是有哪些步骤组成的，每一个步骤具体就是基于镜像启动一个 container 执行一些操作，container 的启动参数可以通过 Task 的入参使用模板语法进行配置。

```
apiVersion: tekton.dev/v1alpha1
kind: Task
metadata:
  name: task-with-parameters
spec:
  inputs:
    params:
      - name: flags
        type: array
      - name: someURL
        type: string
  steps:
    - name: build
      image: registry.cn-hangzhou.aliyuncs.com/knative-sample/alpine:3.9
      command: ["sh", "-c"]
      args: [ "echo ${inputs.params.flags} ; echo ${someURL}" ]
```

TaskRun

Task 定义好以后是不能执行的，就像一个函数定义好以后需要调用才能执行一样。所以需要再定义一个 TaskRun 去执行 Task。TaskRun 主要是负责设置 Task 需要的参数，并通过 taskRef 字段引用要执行的 Task。

```
apiVersion: tekton.dev/v1alpha1
kind: TaskRun
metadata:
  name: run-with-parameters
spec:
  taskRef:
    name: task-with-parameters
  inputs:
    params:
      - name: flags
        value: "--set"
      - name: someURL
        value: "https://github.com/knative-sample"
```

Pipeline

一个 TaskRun 只能执行一个 Task，当需要编排多个 Task 的时候就需要 Pipeline 出马了。Pipeline 是一个编排 Task 的模板。Pipeline 的 params 声明了执行时需要的入参。Pipeline 的 spec.tasks 定义了需要编排的 Task。Tasks 是一个数组，数组中的 task 并不是通过数组声明的顺序去执行的，而是通过 runAfter 来声明 task 执行的顺序。Tekton controller 在解析 CRD 的时候会解析 Task 的顺序，然后根据 runAfter 设置生成的依次树依次去执行。Pipeline 在编排 Task 的时候需要给每一个 Task 传入必须的参数，这些参数的值可以来自 Pipeline 自身的 params 设置。

```
apiVersion: tekton.dev/v1alpha1
kind: Pipeline
metadata:
  name: pipeline-with-parameters
spec:
  params:
    - name: context
      type: string
      description: Path to context
      default: /some/where/or/other
  tasks:
    - name: task-1
      taskRef:
        name: build
      params:
        - name: pathToDockerFile
```

```

        value: Dockerfile
      - name: pathToContext
        value: "${params.context}"
    - name: task-2
      taskRef:
        name: build-push
      runAfter:
        - source-to-image
      params:
        - name: pathToDockerFile
          value: Dockerfile
        - name: pathToContext
          value: "${params.context}"

```

PipelineRun

和 Task 一样 Pipeline 定义完成以后也是不能直接执行的，需要 PipelineRun 才能执行 Pipeline。PipelineRun 的主要作用是给 Pipeline 传入必要的入参，并执行 Pipeline。

```

apiVersion: tekton.dev/v1alpha1
kind: PipelineRun
metadata:
  name: pipelinerun-with-parameters
spec:
  pipelineRef:
    name: pipeline-with-parameters
  params:
    - name: "context"
      value: "/workspace/examples/microservices/leeroy-web"

```

PipelineResource

前面已经介绍了 Tekton 的四个核心概念。现在我们已经知道怎么定义 Task、执行 Task 以及编排 Task 了。但可能你还想在 Task 之间共享资源，这就是 PipelineResource 的作用。比如我们可以把 git 仓库信息放在 PipelineResource 中。这样所有 Task 就可以共享这些信息了。

```

piVersion: tekton.dev/v1alpha1
kind: PipelineResource
metadata:

```

```

name: wizzbang-git
namespace: default
spec:
  type: git
  params:
    - name: url
      value: https://github.com/wizzbangcorp/wizzbang.git
    - name: revision
      value: master

```

授权信息

git 仓库、镜像仓库这些都是需要鉴权才能使用的。所以还需要一种设定鉴权信息的机制。Tekton 本身是 Kubernetes 原生的编排系统。所以可以直接使用 Kubernetes 的 ServiceAccount 机制实现鉴权。

实例如下：

- 定义一个保存镜像仓库鉴权信息的 secret

```

apiVersion: v1
kind: Secret
metadata:
  name: ack-cr-push-secret
  annotations:
    tekton.dev/docker-0: https://registry.cn-hangzhou.aliyuncs.com
type: kubernetes.io/basic-auth
stringData:
  username: <cleartext non-encoded>
  password: <cleartext non-encoded>

```

- 定义 ServiceAccount，并且使用上面的 secret

```

apiVersion: v1
kind: ServiceAccount
metadata:
  name: pipeline-account
secrets:
  - name: ack-cr-push-secret

```

- PipelineRun 中引用 ServiceAccount


```

apiVersion: tekton.dev/v1alpha1
kind: PipelineRun
metadata:
  generateName: tekton-kn-sample-
spec:
  pipelineRef:
    name: build-and-deploy-pipeline
  ... ..
  serviceAccount: pipeline-account

```

Hello World

<https://github.com/knative-sample/tekton-knative/tree/b1.0> 这是一个完整的 Tekton 的 Hello World。下面我们一起体验一下这个 Hello World。

在开始实战之前你需要有一个 Kubernetes 集群，并还需要安装 Knative 和 Tekton，本文是基于 Tekton 最新的 0.8.0 版本写的 Demo。下面我们开始体验使用 Tekton 从源码到构建再到部署的自动化过程。

clone 代码

clone 代码到本地，切换到 b1.0 分支，到 tekton-cicd 目录进行后面的操作。

```

git clone https://github.com/knative-sample/tekton-knative
git checkout b1.0

```

创建 PipelineResource

主要内容在 resources/picalc-git.yaml 文件中。如下所示主要是把 <https://github.com/knative-sample/tekton-knative/tree/b1.0> 保存在 resource 中给其他资源使用。

```

apiVersion: tekton.dev/v1alpha1
kind: PipelineResource
metadata:
  name: tekton-knative-git
spec:
  type: git

```

```

params:
  - name: revision
    value: b1.0
  - name: url
    value: https://github.com/knative-sample/tekton-knative

```

创建 task

创建 task，这个例子中我们创建两个 task: source-to-image 和 deploy-using-kubectl。

- source-to-image

主要内容在 tasks/source-to-image.yaml 文件中。此 task 的主要功能是把源代码编译成镜像。

主要是使用 kaniko 实现容器内编译 Docker 镜像的能力。此 Task 的参数主要是设置编译上下文的一些信息，比如: Dockerfile、ContextPath 以及目标镜像 tag 等。

```

apiVersion: tekton.dev/v1alpha1
kind: Task
metadata:
  name: source-to-image
spec:
  inputs:
    resources:
      - name: git-source
        type: git
    params:
      - name: pathToContext
        description: The path to the build context, used by Kaniko - within
the workspace
        default: .
      - name: pathToDockerFile
        description: The path to the dockerfile to build (relative to the
context)
        default: Dockerfile
      - name: imageUrl
        description: Url of image repository
      - name: imageTag
        description: Tag to apply to the built image

```

```

    default: "latest"
  steps:
    - name: build-and-push
      image: registry.cn-hangzhou.aliyuncs.com/knative-sample/kaniko-project-
executor:v0.10.0
      command:
        - /kaniko/executor
      args:
        - --dockerfile=$(inputs.params.pathToDockerFile)
        - --destination=$(inputs.params.imageUrl):$(inputs.params.imageTag)
        - --context=/workspace/git-source/$(inputs.params.pathToContext)
      env:
        - name: DOCKER_CONFIG
          value: /builder/home/.docker

```

- deploy-using-kubectl

主要内容在 tasks/deploy-using-kubectl.yaml 文件中。

如下所示这个 Task 主要的作用是通过参数获取到目标镜像的信息，然后执行一条 sed 命令把 Knative Service yaml 中的 __IMAGE__ 替换成目标镜像。再通过 kubectl 发布到 Kubernetes 中。

```

apiVersion: tekton.dev/v1alpha1
kind: Task
metadata:
  name: deploy-using-kubectl
spec:
  inputs:
    resources:
      - name: git-source
        type: git
    params:
      - name: pathToYamlFile
        description: The path to the yaml file to deploy within the git source
      - name: imageUrl
        description: Url of image repository
      - name: imageTag
        description: Tag of the images to be used.
        default: "latest"
  steps:
    - name: update-yaml
      image: alpine
      command: ["sed"]
      args:
        - "-i"

```

```

- "-e"
- "s;__IMAGE__;$(inputs.params.imageUrl):$(inputs.params.imageTag);g"
- "/workspace/git-source/$(inputs.params.pathToYamlFile)"
- name: run-kubect1
  image: registry.cn-hangzhou.aliyuncs.com/knative-sample/kubect1:v0.5.0
  command: ["kubect1"]
  args:
    - "apply"
    - "-f"
    - "/workspace/git-source/$(inputs.params.pathToYamlFile)"

```

定义 Pipeline

现在我们已经有两个 Task 了，现在我们就用一个 Pipeline 来编排这两个 Task:

```

apiVersion: tekton.dev/v1alpha1
kind: Pipeline
metadata:
  name: build-and-deploy-pipeline
spec:
  resources:
    - name: git-source
      type: git
  params:
    - name: pathToContext
      description: The path to the build context, used by Kaniko - within the
workspace
      default: src
    - name: pathToYamlFile
      description: The path to the yaml file to deploy within the git source
    - name: imageUrl
      description: Url of image repository
    - name: imageTag
      description: Tag to apply to the built image
  tasks:
    - name: source-to-image
      taskRef:
        name: source-to-image
      params:
        - name: pathToContext
          value: "$(params.pathToContext)"
        - name: imageUrl
          value: "$(params.imageUrl)"
        - name: imageTag

```

```

        value: "$(params.imageTag)"
resources:
  inputs:
    - name: git-source
      resource: git-source
- name: deploy-to-cluster
  taskRef:
    name: deploy-using-kubect1
  runAfter:
    - source-to-image
params:
  - name: pathToYamlFile
    value: "$(params.pathToYamlFile)"
  - name: imageUrl
    value: "$(params.imageUrl)"
  - name: imageTag
    value: "$(params.imageTag)"
resources:
  inputs:
    - name: git-source
      resource: git-source

```

鉴权信息

如下所示，定义一个 Secret 和 ServiceAccount。并且给 ServiceAccount 绑定执行 Knative Service 的权限。

首先创建一个 Secret 保存镜像仓库的鉴权信息，如下所示：

- tekton.dev/docker-0 换成你要推送的镜像仓库的地址
- username 换成镜像仓库鉴权的用户名
- password 换成镜像仓库鉴权的密码

```

apiVersion: v1
kind: Secret
metadata:
  name: ack-cr-push-secret
  annotations:
    tekton.dev/docker-0: https://registry.cn-hangzhou.aliyuncs.com
type: kubernetes.io/basic-auth
stringData:
  username: <cleartext non-encoded>
  password: <cleartext non-encoded>

```

下面这些信息保存到文件中，然后使用 `kubectl apply -f` 指令提交到 Kubernetes。

```

---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: pipeline-account
secrets:
- name: ack-cr-push-secret
---
apiVersion: v1
kind: Secret
metadata:
  name: kube-api-secret
  annotations:
    kubernetes.io/service-account.name: pipeline-account
type: kubernetes.io/service-account-token
---
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: pipeline-role
rules:
- apiGroups: ["serving.knative.dev"]
  resources: ["services"]
  verbs: ["get", "create", "update", "patch"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pipeline-role-binding
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: pipeline-role
subjects:
- kind: ServiceAccount
  name: pipeline-account

```

定义 PipelineRun

ServiceAccount 对应的鉴权信息是通过和 PipelineRun 绑定的方式执行的。参见 `run/picalc-pipeline-run.yaml` 文件。

```

apiVersion: tekton.dev/v1alpha1
kind: PipelineRun
metadata:
  generateName: tekton-kn-sample-
spec:
  pipelineRef:
    name: build-and-deploy-pipeline
  resources:
    - name: git-source
      resourceRef:
        name: tekton-knative-git
  params:
    - name: pathToContext
      value: "src"
    - name: pathToYamlFile
      value: "knative/helloworld-go.yaml"
    - name: imageUrl
      value: "registry.cn-hangzhou.aliyuncs.com/knative-sample/tekton-
knative-helloworld"
    - name: imageTag
      value: "1.0"
  serviceAccount: pipeline-account

```

运行 Tekton HelloWorld

准备 Pipeline 的资源。

```

kubectl apply -f tasks/source-to-image.yaml -f tasks/deploy-using-kubectl.
yaml -f resources/picalc-git.yaml -f image-secret.yaml -f pipeline-account.
yaml -f pipeline/build-and-deploy-pipeline.yaml

```

执行 create 把 pipelineRun 提交到 Kubernetes 集群。之所以这里使用 create 而不是使用 apply 是因为 PipelineRun 每次都会创建一个新的，kubectl 的 create 指令会基于 generateName 创建新的 PipelineRun 资源。

```

kubectl create -f run/picalc-pipeline-run.yaml

```

查看一下 pod 信息可能是下面这样：

```

└─ # kubectl get pod
NAME                                READY    STATUS

```

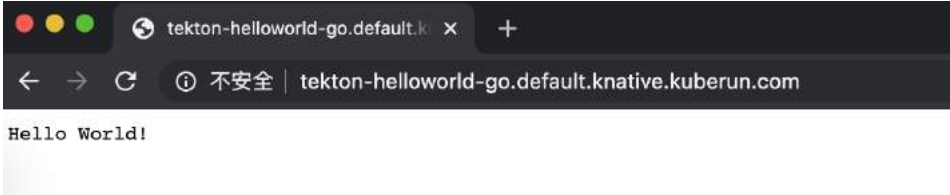
RESTARTS	AGE			
tekton-kn-sample-45d84-deploy-to-cluster-wfrzx-pod-f093ef	0	8h	0/3	Completed
tekton-kn-sample-45d84-source-to-image-7zpqn-pod-c2d20c	0	8h	0/2	Completed

此时查看 Knative service 的配置：

```
└─ # kubectl get ksvc
```

NAME	URL
LATESTCREATED	LATESTREADY
READY	REASON
tekton-helloworld-go	http://tekton-helloworld-go.default.knative.kuberun.com
tekton-helloworld-go-ntksb	tekton-helloworld-go-ntksb
True	True

通过浏览器访问 <http://tekton-helloworld-go.default.knative.kuberun.com> 可以看到 hello World。



参考资料

[knative-sample/tekton-knative https://github.com/knative-sample/tekton-knative/tree/b1.0](https://github.com/knative-sample/tekton-knative/tree/b1.0)

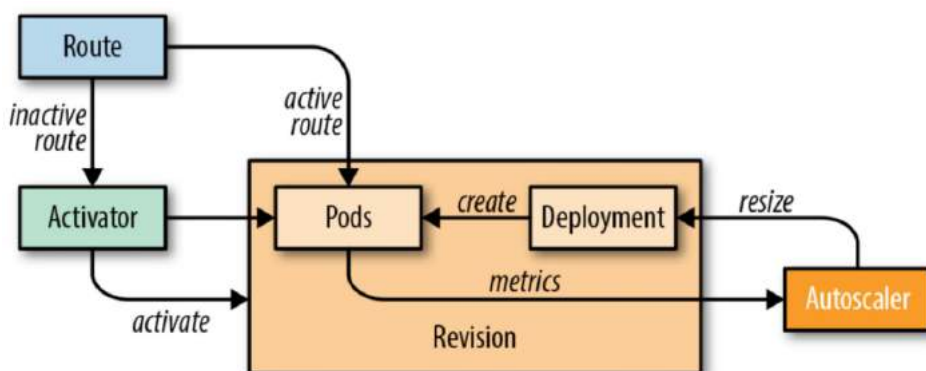
Serving 进阶

自动扩缩容 – Autoscaler

Knative Serving 默认情况下，提供了开箱即用的快速、基于请求的自动扩缩容功能 – Knative Pod Autoscaler (KPA)。下面带你体验如何在 Knative 中配置 Autoscaler。

Autoscaler 机制

Knative Serving 为每个 POD 注入 QUEUE 代理容器 (queue-proxy)，该容器负责向 Autoscaler 报告用户容器并发指标。Autoscaler 接收到这些指标之后，会根据并发请求数及相应的算法，调整 Deployment 的 POD 数量，从而实现自动扩缩容。



算法

Autoscaler 基于每个 POD 的平均请求数 (并发数)。默认并发数为 100。POD

数 = 并发请求总数 / 容器并发数。

如果服务中并发数设置了 10，这时候如果加载了 50 个并发请求的服务，Autoscaler 就会创建了 5 个 POD (50 个并发请求 / 10 = POD)。

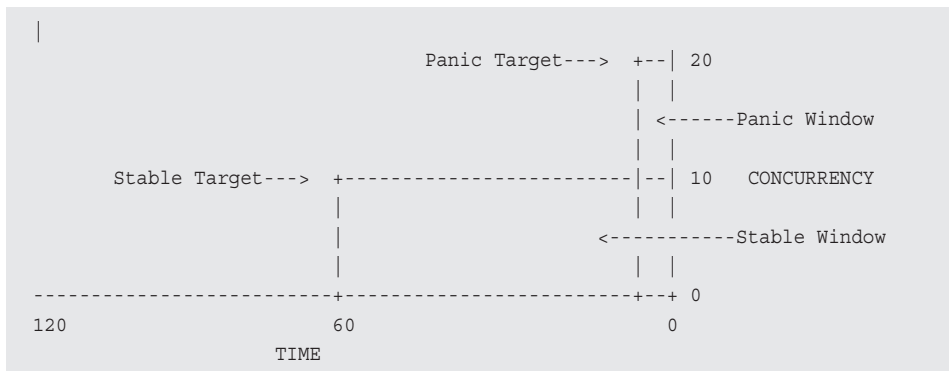
Autoscaler 实现了两种操作模式的缩放算法：Stable/ 稳定模式和 Panic/ 恐慌模式。

稳定模式

在稳定模式下，Autoscaler 调整 Deployment 的大小，以实现每个 POD 所需的平均并发数。POD 的并发数是根据 60 秒窗口内接收所有数据请求的平均数来计算得出。

恐慌模式

Autoscaler 计算 60 秒窗口内的平均并发数，系统需要 1 分钟稳定在所需的并发级别。但是，Autoscaler 也会计算 6 秒的恐慌窗口，如果该窗口达到目标并发的 2 倍，则会进入恐慌模式。在恐慌模式下，Autoscaler 在更短、更敏感的紧急窗口上工作。一旦紧急情况持续 60 秒后，Autoscaler 将返回初始的 60 秒稳定窗口。



配置 KPA

通过上面的介绍，我们对 Knative Pod Autoscaler 工作机制有了初步的了解，那么接下来介绍如何配置 KPA。在 Knative 中配置 KPA 信息，需要修改 k8s 中的 ConfigMap: config-autoscaler，该 ConfigMap 在 knative-serving 命名空间下。

查看 config-autoscaler 使用如下命令：

```
kubectl -n knative-serving get cm config-autoscaler
```

默认的 ConfigMap 如下：

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: config-autoscaler
  namespace: knative-serving
data:
  container-concurrency-target-percentage: "70"
  container-concurrency-target-default: "100"
  requests-per-second-target-default: "200"
  target-burst-capacity: "200"
  stable-window: "60s"
  panic-window-percentage: "10.0"
  panic-threshold-percentage: "200.0"
  max-scale-up-rate: "1000.0"
  max-scale-down-rate: "2.0"
  enable-scale-to-zero: "true"
  tick-interval: "2s"
  scale-to-zero-grace-period: "30s"
```

- container-concurrency-target-percentage: 容器并发请求数比例。容器实际最大并发数 = 容器最大并发请求数 * 容器并发请求数比例。例如，Revision 设置的容器最大并发请求数为：10，容器并发请求数比例为：70%，那么在稳定状态下，实际容器的最大并发请求数为：7。
- container-concurrency-target-default: 容器并发请求默认值。当 Revision 中未设置容器最大并发请求数时，使用该默认值作为容器最大并发请求数
- requests-per-second-target-default: 每秒请求并发 (RPS) 默认值。当使用 RPS 进行度量时，autoscaler 会依据此值进行扩缩容判断。
- target-burst-capacity: 突发请求容量。在突发流量场景下，切换到 Activator 模式进行流量控制。取值范围为 $[-1, +\infty)$ 。-1 表示一直使用 Activator 模式；0 表示不使用突发流量功能。

- `stable-window`: 稳定窗口期。稳定模式窗口期。
- `panic-window-percentage`: 恐慌窗口比例。通过恐慌窗口比例, 计算恐慌窗口期。恐慌窗口期 = 恐慌窗口比例 * 稳定窗口期 / 100。
- `panic-threshold-percentage`: 恐慌模式比例阈值。当前并发请求数大于容器最大并发请求数 * 恐慌比例阈值, 并且达到恐慌窗口期, 则进入恐慌模式。
- `max-scale-up-rate`: 最大扩容比例。每次扩容允许的最大速率。当前最大扩容数 = 最大扩容比例 * Ready 的 Pod 数量。
- `max-scale-down-rate`: 最大缩容比例。
- `enable-scale-to-zero`: 允许缩容至 0。
- `tick-interval`: 扩缩容计算间隔。
- `scale-to-zero-grace-period`: 缩容至 0 优雅下线时间。

为 KPA 配置缩容至 0

为了正确配置使 Revision 缩容为 0, 需要修改 ConfigMap 中的如下参数。

`scale-to-zero-grace-period`

`scale-to-zero-grace-period` 表示在缩为 0 之前, inactive revision 保留的运行时间 (最小是 30s)。

```
scale-to-zero-grace-period: 30s
```

`stable-window`

当在 `stable mode` 模式运行中, autoscaler 在稳定窗口期下平均并发数下的操作。

```
stable-window: 60s
```

`stable-window` 同样可以配置在 Revision 注释中。

```
autoscaling.knative.dev/window: 60s
```

enable-scale-to-zero

保证 enable-scale-to-zero 参数设置为 true。

Termination period

Termination period (终止时间) 是 POD 在最后一个请求完成后关闭的时间。POD 的终止周期等于稳定窗口值和缩放至零宽限期参数的总和。在本例中, Termination period 为 90 秒。

配置并发数

可以使用以下方法配置 Autoscaler 的并发数。

target

target 定义在给定时间 (软限制) 需要多少并发请求, 是 Knative 中 Autoscaler 的推荐配置。

在 ConfigMap 中默认配置的并发 target 为 100。

```
`container-concurrency-target-default: 100`
```

这个值可以通过 Revision 中的 autoscaling.knative.dev/target 注释进行修改:

```
autoscaling.knative.dev/target: 50
```

containerConcurrency

注意: 只有在明确需要限制在给定时间有多少请求到达应用程序时, 才应该使用 containerConcurrency (容器并发)。只有当应用程序需要强制的并发约束时, 才建议使用 containerConcurrency。

containerConcurrency 限制在给定时间允许并发请求的数量 (硬限制), 并在 Revision 模板中配置。

```
containerConcurrency: 0 | 1 | 2-N
```

- 1: 将确保一次只有一个请求由 Revision 给定的容器实例处理。
- 2-N: 请求的并发值限制为 2 或更多
- 0: 表示不作限制，有系统自身决定

配置扩缩容边界 (minScale 和 maxScale)

通过 minScale 和 maxScale 可以配置应用程序提供服务的最小和最大 Pod 数量。通过这两个参数配置可以控制服务冷启动或者控制计算成本。

minScale 和 maxScale 可以在 Revision 模板中按照以下方式进行配置：

```
spec:
  template:
    metadata:
      autoscaling.knative.dev/minScale: "2"
      autoscaling.knative.dev/maxScale: "10"
```

通过在 Revision 模板中修改这些参数，将会影响到 PodAutoscaler 对象，这也表明在无需修改 Knative Serving 系统配置的情况下，PodAutoscaler 对象是可被修改的。

```
edit podautoscaler <revision-name>
```

注意：这些注释适用于 Revision 的整个生命周期。即使 Revision 没有被任何 route 引用，minscale 指定的最小 POD 计数仍将提供。请记住，不可路由的 Revision 可能被垃圾收集掉。

默认情况

如果未设置 minscale 注释，pods 将缩放为零（如果根据上面提到的 config-map, enable-scale-to-zero 为 false，则缩放为 1）。

如果未设置 maxscale 注释，则创建的 Pod 数量将没有上限。

下面我们看一下基于 KPA 配置的示例

Knative 0.10.0 版本部署安装可以参考：[阿里云部署 Knative](#)

我们使用官方提供的 autoscale-go 示例来进行演示，示例 service.yaml 如下：

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: autoscale-go
  namespace: default
spec:
  template:
    metadata:
      labels:
        app: autoscale-go
      annotations:
        autoscaling.knative.dev/target: "10"
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/autoscale-go:0.1
```

获取访问网关：

```
$ kubectl get svc istio-ingressgateway --namespace istio-system --output
jsonpath="{.status.loadBalancer.ingress[*]['ip']}"
121.199.194.150
```

Knative 0.10.0 版本中获取域名信息：

```
$ kubectl get route autoscale-go --output jsonpath="{.status.url}" | awk -F/
'{print $3}'
autoscale-go.default.example.com
```

场景 1：并发请求示例

如上配置，当前最大并发请求数 10。我们执行 30s 内保持 50 个并发请求，看一下执行情况：

```
hey -z 30s -c 50 -host "autoscale-go.default.example.com" "http://121.199
.194.150?sleep=100&prime=10000&bloat=5"
```



```
hey -z 30s -c 50 -host "autoscale-go.default.example.com" "http://121.199.194.150?sleep=100&prime=10000&bloat=5"
```

```

[roo@i2p1k5d8a/g07f0tk2 autoscale-go]$ kubectl apply -f autoscale-go.yaml
service.serving.knative.dev/autoscale-go configured
[roo@i2p1k5d8a/g07f0tk2 autoscale-go]$

[roo@i2p1k5d8a/g07f0tk2 ~]$ kubectl get pods -w
NAME                                READY   STATUS    RESTARTS   AGE
autoscale-go-c6mk-deployment-6684bf54c9-p8tf2   3/3     Running   0           45s
default-broker-filter-776d4d5dd-wcsqw           2/2     Running   1           12s
default-broker-ingress-5f4f48888-qgdbb          2/2     Running   1           12s
ens-mssos-face-h5vhq-f444c57f-zktwp            2/2     Running   2           12s
    
```

结果如我们所预期：最多扩容出来了 3 个 POD，并且即使在无访问请求流量的情况下，保持了 1 个运行的 POD。

结论

看了上面的介绍，是不是感觉在 Knative 中配置应用扩缩容是如此简单。其实 Knative 中除了支持 KPA 之外，也支持 K8s HPA。你可以通过如下配置基于 CPU 的 Horizontal POD Autoscaler (HPA)：

通过在修订模板中添加或修改 `autoscaling.knative.dev/class` 和 `autoscaling.knative.dev/metric` 值作为注释，可以将 Knative 配置为使用基于 CPU 的自动缩放，而不是默认的基于请求的度量。配置如下：

```
spec:
  template:
    metadata:
      autoscaling.knative.dev/metric: concurrency
      autoscaling.knative.dev/class: hpa.autoscaling.knative.dev
```

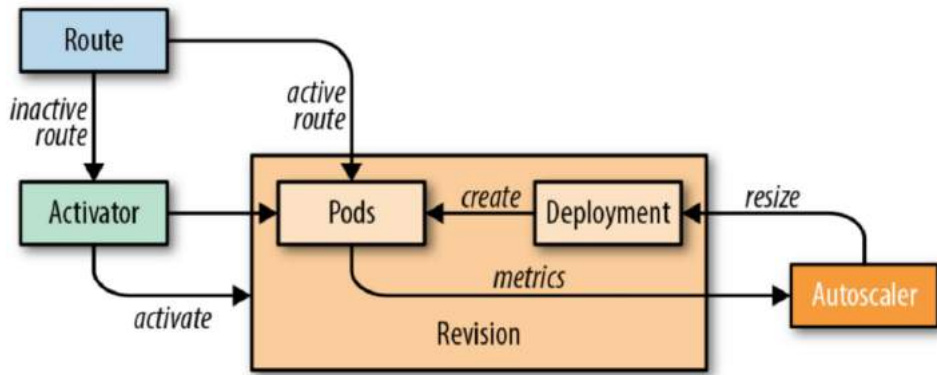
你可以自由的将 Knative Autoscaling 配置为使用默认的 KPA 或 Horizontal POD Autoscaler (HPA)。

Serving 健康检查机制分析

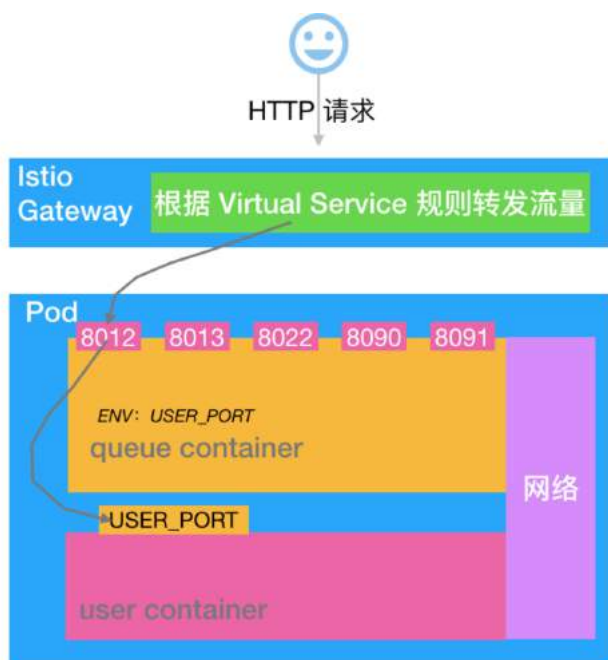
从头开发一个 Serverless 引擎并不是一件容易的事情，今天咱们就从 Knative 的健康检查说起。通过健康检查这一点来看看 Serverless 模式和传统的模式都有哪些不同以及 Knative 针对 Serverless 场景都做了什么思考。

Knative Serving 模块的核心原理如下图所示。下图中的 Route 可以理解成是 Istio Gateway 的角色。

- 当缩容到零时进来的流量就会指到 Activator 上面
- 当 Pod 数不为零时流量就会指到对应的 Pod 上面，此时流量不经过 Activator
- 其中 Autoscaler 模块根据请求的 Metrics 信息实时动态的扩缩容



Knative 的 Pod 是由两个 Container 组成的：Queue-Proxy 和业务容器 user-container。架构如下：



咱们以 http1 为例进行说明。业务流量首先进入 Istio Gateway，然后会转发到 Queue-Proxy 的 8012 端口，Queue-Proxy 8012 再把请求转发到 user-container 的监听端口，至此一个业务请求的服务就算完成了。

粗略的介绍原理基本就是上面这样，现在咱们对几个细节进行深入的剖析看看其内部机制：

- 为什么要引入 Queue-Proxy ？
- Pod 缩容到零的时候流量会转发到 Activator 上面，那么 Activator 是怎么处理这些请求的？
- Knative 中的业务 Pod 有 Queue-Proxy 和 user-container，那么 Pod 的 readinessProber 和 LivenessProber 分别是怎么做的？Pod 的 readinessProber、LivenessProber 和业务的健康状态是什么样的关系？
- Istio Gateway 向 Pod 转发流量的时候是怎么选择 Pod 进行转发的？

为什么要引入 Queue-Proxy

Serverless 的一个核心诉求就是把业务的复杂度下沉到基础平台，让业务代码快速的迭代并且按需使用资源。不过现在更多的还是聚焦在按需使用资源层面。

如果想要按需使用资源我们就需要收集相关的 Metrics，并根据这些 Metrics 信息来指导资源的伸缩。Knative 首先实现的就是 KPA 策略，这个策略是根据请求数来判断是否需要扩容的。所以 Knative 需要有一个机制收集业务请求数量。除了业务请求数还有如下信息也是需要统一处理：

- 访问日志的管理
- Tracing
- Pod 健康检查机制
- 需要实现 Pod 和 Activator 的交互，当 Pod 缩容到零的时候如何接收 Activator 转发过来的流量
- 其他诸如判断 Ingress 是否 Ready 的逻辑也是基于 Queue-Proxy 实现的

为了保持和业务的低耦合关系，还需要实现上述这些功能所以就引入了 Queue-Proxy 负责这些事情。这样可以在业务无感知的情况下把 Serverless 的功能实现。

从零到一的过程

当 Pod 缩容到零的时候流量会指到 Activator 上面，Activator 接收到流量以后会主动“通知”Autoscaler 做一个扩容的操作。扩容完成以后 Activator 会探测 Pod 的健康状态，需要等待第一个 Pod ready 之后才能把流量转发过来。所以这里就出现了第一个健康检查的逻辑：**Activator 检查第一个 Pod 是否 ready**。

这个健康检查是调用的 Pod 8012 端口完成的，Activator 会发起 HTTP 的健康检查，并且设置 K-Network-Probe=queue Header，所以 Queue Container 中会根据 K-Network-Probe=queue 来判断这是来自 Activator 的检查，然后执行相应的逻辑。

参考阅读

- Activator to perform health checks before forwarding real requests
<https://github.com/knative/serving/issues/2856>
- Activator: Retry on Get Revision error
<https://github.com/knative/serving/issues/1573>
- Retry on Get Revision error? #1558
<https://github.com/knative/serving/issues/1558>
- Always pass Healthy dests to the throttler #5466
<https://github.com/knative/serving/issues/5466>
- Consolidate queue-proxy probe handlers #5465
<https://github.com/knative/serving/issues/5465>
- Queue proxy logging, metrics and end to end traces #1286
<https://github.com/knative/serving/issues/1286>
- End to end traces from queue proxy #3898
<https://github.com/knative/serving/issues/3898>

VirtualService 的健康检查

Knative Revision 部署完成以后就会自动创建一个 Ingress(以前叫做 ClusterIngress), 这个 Ingress 最终会被 Ingress Controller 解析成 Istio 的 VirtualService 配置, 然后 Istio Gateway 才能把相应的流量转发给相关的 Revision。

所以每添加一个新的 Revision 都需要同步创建 Ingress 和 Istio 的 VirtualService, 而 VirtualService 是没有状态表示 Istio 的管理的 Envoy 是否配置生效的能力的。所以 Ingress Controller 需要发起一个 http 请求来监测 VirtualService 是否 ready。这个 http 的检查最终也会打到 Pod 的 8012 端口上。标识 Header 是 K-Network-Probe=probe。Queue-Proxy 需要基于此来判断, 然后执行相应的逻辑。

相关代码如下所示:

https://github.com/knative/serving/blob/master/pkg/network/probe_handler.go#L37

```

36 // ServeHTTP handles probing requests
37 func (h *handler) ServeHTTP(w http.ResponseWriter, r *http.Request) {
38     if ph := r.Header.Get(ProbeHeaderName); ph != ProbeHeaderValue {
39         r.Header.Del(HashHeaderName)
40         h.next.ServeHTTP(w, r)
41         return
42     }
43
44     hh := r.Header.Get(HashHeaderName)
45     if hh == "" {
46         http.Error(w, fmt.Sprintf("a probe request must contain a non-empty %q header", HashHeaderName), http.StatusBadRequest)
47         return
48     }
49
50     w.Header().Set(HashHeaderName, hh)
51     w.WriteHeader(200)
52 }

```

<https://github.com/knative/serving/blob/master/pkg/reconciler/ingress/status.go#L348>

```

348 ok, err := prober.Do(
349     item.podState.context,
350     transport,
351     item.url,
352     prober.WithHeader(network.ProbeHeaderName, network.ProbeHeaderValue),
353     prober.ExpectsStatusCodes([]int{http.StatusOK}),
354     prober.ExpectsHeader(network.HashHeaderName, item.ingressState.hash))
355
356 // In case of cancellation, drop the work item

```

参考阅读

Gateway 通过这个健康检查来判断 Pod 是否可以提供服务

- New probe handling in Queue-Proxy & Activator #5159
<https://github.com/knative/serving/pull/5159>
- Extend VirtualService/Gateway probing to HTTPS #5156
<https://github.com/knative/serving/issues/5156>
- Probe Envoy pods to determine when a ClusterIngress is actually deployed #4734
<https://github.com/knative/serving/pull/4734>

- ClusterIngress Status

<https://docs.google.com/document/d/1mXDrRhVOF48qRR7-4fZMTk-MHkoOGZJtrRGavGloVjGs/edit>

- Consolidate queue-proxy probe handlers #5465

<https://github.com/knative/serving/issues/5465>

Kubelet 的健康检查

Knative 最终生成的 Pod 是需要落实到 Kubernetes 集群的，Kubernetes 中 Pod 有两个健康检查的机制 ReadinessProber 和 LivenessProber。其中 LivenessProber 是判断 Pod 是否活着，如果检查失败 Kubelet 就会尝试重启 Container，ReadinessProber 是用来判断业务是否 Ready，只有业务 Ready 的情况下才会把 Pod 挂载到 Kubernetes Service 的 EndPoint 中，这样可以保证 Pod 故障时对业务无损。

那么问题来了，Knative 的 Pod 中默认会有两个 Container: Queue-Proxy 和 user-container。前面两个健康检查机制你应该也发现了，流量的“前半路径”需要通过 Queue-Proxy 来判断是否可以转发流量到当前 Pod，而在 Kubernetes 的机制中 Pod 是否加入 Kubernetes Service EndPoint 中完全是由 Readiness-Prober 的结果决定的。而这两个机制是独立的，所以我们需要有一种方案来把这两个机制协调一致。这也是 Knative 作为一个 Serverless 编排引擎是需要对流量做更精细的控制要解决的问题。所以 Knative 最终是把 user-container 的 Readiness-Prober 收敛到 Queue-Proxy 中，通过 Queue-Proxy 的结果来决定 Pod 的状态。

另外 <https://github.com/knative/serving/issues/2912> 这个 Issue 中也提到在启动 istio 的情况下，kubelet 发起的 tcp 检查可能会被 Envoy 拦截，所以给 user-container 配置 TCP 探测器判断 user-container 是否 ready 也是不准的。这也是需要把 Readiness 收敛到 Queue-Proxy 的一个动机。

Knative 收敛 user-container 健康检查能力的方法是：

- 置空 user-container 的 ReadinessProber
- 把 user-container 的 ReadinessProber 配置的 json String 配置到 Queue-Proxy 的 env 中
- Queue-Proxy 的 Readinessprober 命令里面解析 user-container 的 ReadinessProber 的 json String 然后实现健康检查逻辑。并且这个检查的机制和前面提到的 Activator 的健康检查机制合并到了一起。这样做也保证了 Activator 向 Pod 转发流量时 user-container 一定是 Ready 状态

参考阅读

- Consolidate queue-proxy probe handlers #5465
<https://github.com/knative/serving/issues/5465>
- Use user-defined readinessProbe in queue-proxy #4731
<https://github.com/knative/serving/pull/4731>
- Apply default livenessProbe and readinessProbe to the user container #4014
<https://github.com/knative/serving/issues/4014>
- Good gRPC deployment pods frequently fail at least one health check #3308
<https://github.com/knative/serving/issues/3308>
- Fix invalid helloworld example
<https://github.com/knative/serving/pull/4780>

这里面有比较详细的方案讨论，最终社区选择的方案也是在这里介绍的

- Allow probes to run on a more granular timer. #76951
<https://github.com/kubernetes/kubernetes/issues/76951>
- Merge 8022/health to 8012/8013 #5524
<https://github.com/knative/serving/pull/5524>
- TCP probe the user-container from the queue-proxy before marking the pod ready. #2915

<https://github.com/knative/serving/pull/2915>

- [WIP] Use user-defined readiness probes through queue-proxy #4600

<https://github.com/knative/serving/pull/4600>

- queue-proxy /health to perform TCP connect to user container #2912

<https://github.com/knative/serving/issues/2912>

使用方法

如下所示可以在 Knative Service 中定义 Readiness

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: readiness-prober
spec:
  template:
    metadata:
      labels:
        app: helloworld-go
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/helloworld-go:160e4db7
          readinessProbe:
            httpGet:
              path: /
            initialDelaySeconds: 3
```

但是需要说明两点：

1. 和原生的 Kubernetes Pod Readiness 配置相比，Knative 中 timeoutSeconds、failureThreshold、periodSeconds 和 successThreshold 如果要配置就要一起配置，并且不能为零，否则 Knative webhook 校验无法通过。并且如果设置了 periodSeconds 那么一旦出现一次 Success，就再也不会去探测 user-container(不建议设置 periodSeconds，应该让系统自动处理)
2. 如果 periodSeconds 没有配置那么就会使用默认的探测策略，默认配置如下：

```
timeoutSeconds: 60
  failureThreshold: 3
  periodSeconds: 10
  successThreshold: 1
```

从这个使用方式上来看其实 Knative 是在逐渐收敛 user-container 配置，因为在 Serverless 模式中需要系统自动化处理很多逻辑，这些“系统行为”就不需要麻烦用户了。

小结

前面提到的三种健康检查机制的对比关系：

Probe Request Source	Path	Extra features	comment
Activator probe requests	:8012	With header K-Net-work-Probe=-queue. Expected queue as response body.	Probe requests from Activator before it proxies external requests
VirtualService/Gateway probe requests	:8012	With header K-Net-work-Probe=probe and non-empty K-Network-Hash header	This is used to detect which version of a VirtualService an Envoy Pod is currently serving. They are proxied from VirtualService to activator/queue-proxy.
Kubelet probe requests	:8012	With non-empty K-Kubelet-Probe header or with header user-agent=kube-probe/*	I don't think currently kubectl sends probe requests to this path. We can delete it. Correct me if I was wrong.

流量灰度和版本管理

本篇主要介绍 Knative Serving 的流量灰度，通过一个 rest-api 的例子演示如何创建多个 Revision、并在不同的 Revision 之间按照流量比例灰度。

部署 rest-api v1

咱们先部署第一个版本。

- 代码

测试之前我们需要写一段 rest-api 的代码，并且还要能够区分不同的版本。下面我基于官方的[例子](#)进行了修改，为了方便去掉了 github.com/gorilla/mux 依赖，直接使用 Golang 系统包 `net/http` 替代。这段代码可以通过 `RESOURCE` 环境变量来区分不同的版本。

```
package main

import (
    "fmt"
    "io/ioutil"
    "log"
    "net/http"
    "net/url"
    "os"

    "flag"
)

var resource string

func main() {
    flag.Parse()
    //router := mux.NewRouter().StrictSlash(true)

    resource = os.Getenv("RESOURCE")
    if resource == "" {
```

```

    resource = "NOT SPECIFIED"
}

root := "/" + resource
path := root +("/{stockId}"

http.HandleFunc("/", Index)
http.HandleFunc(root, StockIndex)
http.HandleFunc(path, StockPrice)

if err := http.ListenAndServe(fmt.Sprintf(":%s", "8080"), nil); err !=
nil {
    log.Fatalf("ListenAndServe error:%s ", err.Error())
}
}

func Index(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Welcome to the %s app! \n", resource)
}

func StockIndex(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "%s ticker not found!, require /%s/{ticker}\n", resource,
resource)
}

func StockPrice(w http.ResponseWriter, r *http.Request) {
    stockId := r.URL.Query().Get("stockId")

    url := url.URL{
        Scheme: "https",
        Host:   "api.iextrading.com",
        Path:   "/1.0/stock/" + stockId + "/price",
    }

    log.Print(url)

    resp, err := http.Get(url.String())
    if err != nil {
        fmt.Fprintf(w, "%s not found for ticker : %s \n", resource, stockId)
        return
    }

    defer resp.Body.Close()

    body, err := ioutil.ReadAll(resp.Body)

    fmt.Fprintf(w, "%s price for ticker %s is %s\n", resource, stockId,
string(body))
}

```

- Dockerfile

创建一个叫做 Dockerfile 的文件，把下面这些内容复制到文件中。执行 `docker build --tag registry.cn-hangzhou.aliyuncs.com/knative-sample/rest-api-go:v1 --file ./Dockerfile .` 命令即可完成镜像的编译。

你在测试的时候请把 `registry.cn-hangzhou.aliyuncs.com/knative-sample/rest-api-go:v1` 换成你自己的镜像仓库地址。

编译好镜像以后执行 `docker push registry.cn-hangzhou.aliyuncs.com/knative-sample/rest-api-go:v1` 把镜像推送到镜像仓库。

```
FROM registry.cn-hangzhou.aliyuncs.com/knative-sample/golang:1.12 as builder

WORKDIR /go/src/github.com/knative-sample/rest-api-go
COPY . .

RUN CGO_ENABLED=0 GOOS=linux go build -v -o rest-api-go
FROM registry.cn-hangzhou.aliyuncs.com/knative-sample/alpine-sh:3.9
COPY --from=builder /go/src/github.com/knative-sample/rest-api-go/rest-api-go
/rest-api-go

CMD ["/rest-api-go"]
```

- Service 配置

镜像已经有了，我们开始部署 Knative Service。把下面的内容保存到 `revision-v1.yaml` 中，然后执行 `kubectl apply -f revision-v1.yaml` 即可完成 Knative Service 的部署。

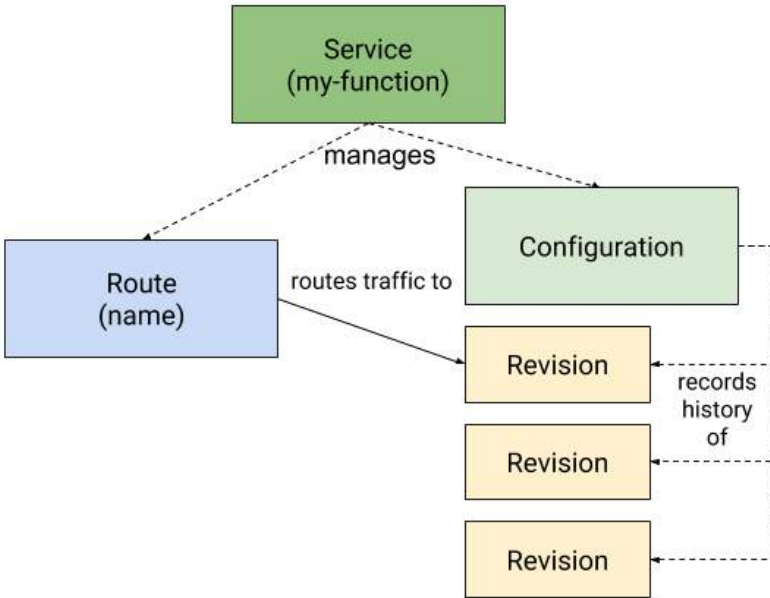
```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: traffic-example
  namespace: default
spec:
  template:
    metadata:
      name: traffic-example-v1
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/rest-api-go:v1
```

```
env:
  - name: RESOURCE
    value: v1
readinessProbe:
  httpGet:
    path: /
```

首次安装会创建出一个叫做 traffic-example-v1 的 Revision，并且是把 100% 的流量都打到 traffic-example-v1 上。

验证 Serving 的各个资源

如下图所示，我们先回顾一下 Serving 涉及到的各种资源。接下来我们分别看一下刚才部署的 revision-v1.yaml 各个资源配置。



- Knative Service

```
kubectl get ksvc traffic-example --output yaml
```

- Knative Configuration

```
kubectl get configuration -l \
"serving.knative.dev/service=traffic-example" --output yaml
```

- Knative Revision

```
kubectl get revision -l \
"serving.knative.dev/service=traffic-example" --output yaml
```

- Knative Route

```
kubectl get route -l \
"serving.knative.dev/service=traffic-example" --output yaml
```

访问 rest-api 服务

我们部署的 Service 名称是: traffic-example。访问这个 Service 需要获取 Istio Gateway 的 IP, 然后使用 traffic-example.default.knative.kuberun.com 这个 Domain 绑定 Host 的方式发起 curl 请求。为了方便测试我写成了一个脚本。创建一个 run-test.sh 文件, 把下面这些内容复制到文件内, 然后赋予文件可执行权限。执行执行此脚本就能得到测试结果。

```
#!/bin/bash
#*****#
# Create Date: 2019-11-06 14:38
#*****#

SVC_NAME="traffic-example"
export INGRESSGATEWAY=istio-ingressgateway
export GATEWAY_IP=`kubectl get svc $INGRESSGATEWAY --namespace istio-system
--output jsonpath="{.status.loadBalancer.ingress[*]['ip']}"`
export DOMAIN_NAME=`kubectl get route ${SVC_NAME} --output jsonpath="{.
status.url}"| awk -F/ '{print $3}'`

curl -H "Host: ${DOMAIN_NAME}" http://${GATEWAY_IP}
```

测试结果:

从下面的命令输出结果可以看到现在返回的是 v1 的信息, 说明请求打到 v1 上面了。

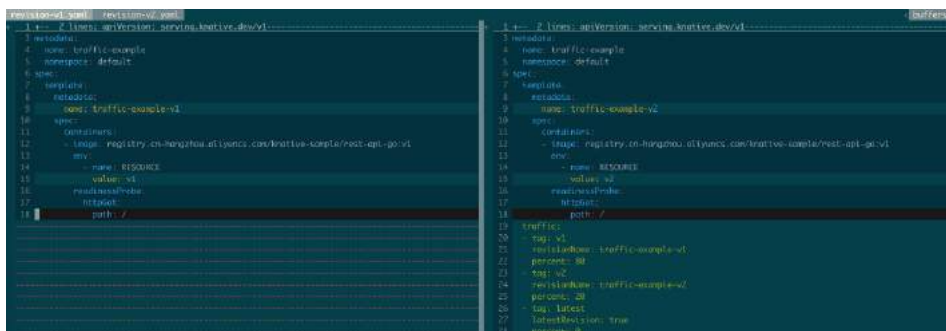
```
└─ # ./run-test.sh  
Welcome to the v1 app!
```

灰度 20% 的流量到 v2

创建 v2 revision revision-v2.yaml 文件，内容如下：

```
apiVersion: serving.knative.dev/v1  
kind: Service  
metadata:  
  name: traffic-example  
  namespace: default  
spec:  
  template:  
    metadata:  
      name: traffic-example-v2  
    spec:  
      containers:  
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/rest-api-  
go:v1  
        env:  
          - name: RESOURCE  
            value: v2  
        readinessProbe:  
          httpGet:  
            path: /  
      traffic:  
        - tag: v1  
          revisionName: traffic-example-v1  
          percent: 80  
        - tag: v2  
          revisionName: traffic-example-v2  
          percent: 20  
        - tag: latest  
          latestRevision: true  
          percent: 0
```

我们对比一下 v1 版本和 v2 版本可以发现，v2 版本的 Service 中增加了 traffic: 的配置。在 traffic 中指定了每一个 Revision。



执行 `kubectl apply -f revision-v2.yaml` 安装 v2 版本的配置。然后多次执行 `for ((i=1; i<=10; i++)); do ./run-test.sh; done` 这条命令就能看到现在返回的结果中 v1 和 v2 的比例基本是 8:2 的比例。下面这是我真实测试的结果。

```
└─# for ((i=1; i<=10; i++)); do ./run-test.sh; done
Welcome to the v1 app!
Welcome to the v2 app!
Welcome to the v1 app!
Welcome to the v1 app!
Welcome to the v1 app!
Welcome to the v1 app!
Welcome to the v2 app!
Welcome to the v1 app!
Welcome to the v2 app!
Welcome to the v2 app!
```

提前验证 Revision

上面展示的 v2 的例子，在创建 v2 的时候直接就把流量分发到 v2，如果此时 v2 有问题就会导致有 20% 的流量异常。下面我们就展示一下如何在转发流量之前验证新的 revision 服务是否正常。我们再创建一个 v3 版本。

创建一个 `revision-v3.yaml` 的文件，内容如下：

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: traffic-example
  namespace: default
```

```
spec:
  template:
    metadata:
      name: traffic-example-v3
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/rest-api-
go:v1
      env:
        - name: RESOURCE
          value: v2
      readinessProbe:
        httpGet:
          path: /
  traffic:
    - tag: v1
      revisionName: traffic-example-v1
      percent: 80
    - tag: v2
      revisionName: traffic-example-v2
      percent: 20
    - tag: latest
      latestRevision: true
      percent: 0
```

执行 `kubectl apply -f revision-v3.yaml` 部署 v3 版本。然后查看一下 Revision 情况:

```
└─ # kubectl get revision -l "serving.knative.dev/service=traffic-example"
NAME          CONFIG NAME      K8S SERVICE NAME  GENERATION
READY  REASON
traffic-example-v1  traffic-example  traffic-example-v1  1          True
traffic-example-v2  traffic-example  traffic-example-v2  2          True
traffic-example-v3  traffic-example  traffic-example-v3  3          True
```

可以看到现在已经创建出来了三个 Revision。

此时我们再看一下 `stock-service-example` 的真实生效:

```
└─ # kubectl get ksvc traffic-example -o yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  annotations:
  ...
```

```
status:
...

traffic:
- latestRevision: false
  percent: 80
  revisionName: traffic-example-v1
  tag: v1
  url: http://v1-traffic-example.default.knative.kuberun.com
- latestRevision: false
  percent: 20
  revisionName: traffic-example-v2
  tag: v2
  url: http://v2-traffic-example.default.knative.kuberun.com
- latestRevision: true
  percent: 0
  revisionName: traffic-example-v3
  tag: latest
  url: http://latest-traffic-example.default.knative.kuberun.com
url: http://traffic-example.default.knative.kuberun.com
```

可以看到 v3 Revision 虽然创建出来了，但是因为没有设置 traffic，所以并不会
有流量转发。此时你执行多少次 `./run-test.sh` 都不会得到 v3 的输出。

在 Service 的 `status.traffic` 配置中可以看到 latest Revision 的配置：

```
- latestRevision: true
  percent: 0
  revisionName: traffic-example-v3
  tag: latest
  url: http://latest-traffic-example.default.knative.kuberun.com
```

每一个 Revision 都有一个自己的 URL，所以只需要基于 v3 Revision 的 URL
发起请求就能开始测试了。

我已经写好了一个测试脚本，你可以把下面这段脚本保存在 `latest-run-test.sh`
文件中，然后执行这个脚本就能直接发起到 latest 版本的请求：

```
#!/bin/bash
export INGRESSGATEWAY=istio-ingressgateway
export GATEWAY_IP=`kubectl get svc $INGRESSGATEWAY --namespace istio-system
--output jsonpath="{.status.loadBalancer.ingress[*]['ip']}"`
export DOMAIN_NAME=`kubectl get route ${SVC_NAME} --output jsonpath="{.
```

```
status.url}"| awk -F/ '{print $3}'`

export LAST_DOMAIN=`kubectl get ksvc traffic-example --output jsonpath="{.
status.traffic[?(@.tag=='latest')].url}"| cut -d '/' -f 3`

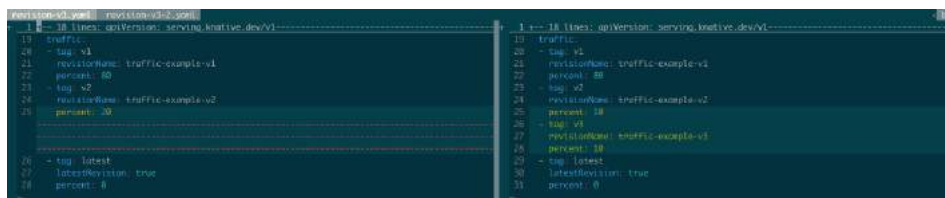
curl -H "Host: ${LAST_DOMAIN}" http://${GATEWAY_IP}
```

测试 v3 版本如果没问题就可以把流量分发到 v3 版本了。

下面我们再创建一个文件 revision-v3-2.yaml，内容如下：

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: traffic-example
  namespace: default
spec:
  template:
    metadata:
      name: traffic-example-v3
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/rest-api-
go:v1
      env:
        - name: RESOURCE
          value: v3
      readinessProbe:
        httpGet:
          path: /
  traffic:
    - tag: v1
      revisionName: traffic-example-v1
      percent: 80
    - tag: v2
      revisionName: traffic-example-v2
      percent: 10
    - tag: v3
      revisionName: traffic-example-v3
      percent: 10
    - tag: latest
      latestRevision: true
      percent: 0
```

用 vimdiff 看一下 revision-v3.yaml 和 revision-v3-2.yaml 的区别：



```

1 18 lines: apiVersion: serving.knative.dev/v1
19 traffic
20 - tag: v1
21 revisionName: traffic-example-v1
22 percent: 80
23 - tag: v2
24 revisionName: traffic-example-v2
25 percent: 10
26
27
28 - tag: latest
29 latestRevision: true
30 percent: 0
31

```

```

1 18 lines: apiVersion: serving.knative.dev/v1
19 traffic
20 - tag: v1
21 revisionName: traffic-example-v1
22 percent: 80
23 - tag: v2
24 revisionName: traffic-example-v2
25 percent: 10
26 - tag: v3
27 revisionName: traffic-example-v3
28 percent: 10
29
30 - tag: latest
31 latestRevision: true
32 percent: 0
33

```

revision-v3-2.yaml 增加了到 v3 的流量转发。此时执行 `for ((i=1; i<=10; i++)); do ./run-test.sh; done` 可以看到 v1、v2 和 v3 的比例基本是：8:1:1

```

└─ # for ((i=1; i<=10; i++)); do ./run-test.sh; done
Welcome to the v3 app!
Welcome to the v1 app!
Welcome to the v1 app!
Welcome to the v1 app!
Welcome to the v2 app!
Welcome to the v1 app!
Welcome to the v1 app!
Welcome to the v1 app!
Welcome to the v1 app!
Welcome to the v1 app!
Welcome to the v1 app!

```

版本回滚

Knative Service 的 Revision 是不能修改的，每次 Service Spec 的更新创建的 Revision 都会保留在 kube-apiserver 中。如果应用发布到某个新版本发现问题想要回滚到老版本的时候只需要指定相应的 Revision，然后把流量转发过去就行了。

小结

Knative Service 的灰度、回滚都是基于流量的。Workload(Pod) 是根据过来的流量自动创建出来的。所以在 Knative Serving 模型中流量是核心驱动。这和传统的应用发布、灰度模型是有区别的。

假设有一个应用 app1，传统的做法首先是设置应用的实例个数 (Kubernetes 体系中就是 Pod)，我们假设实例个数是 10 个。如果要进行灰度发布，那么传统的做法就是先发布一个 Pod，此时 v1 和 v2 的分布方式是：v1 的 Pod 9 个，v2 的

Pod 1 个。如果要继续扩大灰度范围的话那就是 v2 的 Pod 数量变多，v1 的 Pod 数量变少，但总的 Pod 数量维持 10 个不变。

在 Knative Serving 模型中 Pod 数量永远都是根据流量自适应的，不需要提前指定。在灰度的时候只需要指定流量在不同版本之间的灰度比例即可。每一个 Revision 的实例数都是根据流量的大小自适应，不需要提前指定。

从上面的对比中可以发现 Knative Serving 模型是可以精准的控制灰度影响的范围的，保证只灰度一部分流量。而传统的模型中 Pod 灰度的比例并不能真实的代表流量的比例，是一个间接的灰度方法。

服务路由管理

Knative 默认会为每一个 Service 生成一个域名，并且 Istio Gateway 要根据域名判断当前的请求应该转发给哪个 Knative Service。Knative 默认使用的主域名是 example.com，这个域名是不能作为线上服务的。本文我首先介绍一下如何修改默认主域名，然后再深入一层介绍如何添加自定义域名以及如何根据 path 关联到不同的 Knative Service。

Knative Serving 的默认域名 example.com

首先需要部署一个 Knative Service。如果你已经有了一个 Knative 集群，那么直接把下面的内容保存到 login-service.yaml 文件中。然后执行一下 `kubectl apply -f login-service.yaml` 即可把 login-service 服务部署到 default namespace 中。

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: login-service
spec:
  template:
    metadata:
      labels:
        app: login-service
      annotations:
        autoscaling.knative.dev/target: "10"
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/helloworld-go:73fbdd56
          env:
            - name: TARGET
              value: "Login Service"
```

现在来看一下 Knative Service 自动生成的域名配置：

```
└─ # kubectl -n default get ksvc
```

NAME	URL	LATESTCREATED	LATESTREADY
READY	REASON		
login-service	http://login-service.default.example.com	login-service-	
wsnvc	login-service-wsnvc	True	

现在使用 curl 指定 Host 就能访问服务了。

- 首先获取到 Istio Gateway IP

```
└─ # kubectl get svc istio-ingressgateway --namespace istio-system --output  
    jsonpath="{.status.loadBalancer.ingress[*]['ip']}"  
    47.95.191.136
```

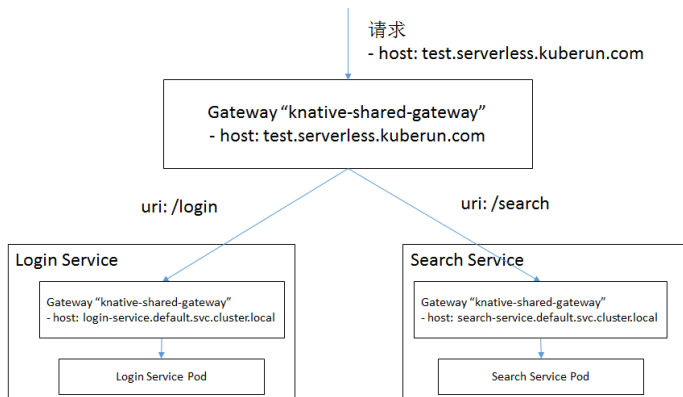
- 访问 login-service 服务

```
└─ # curl -H "Host: login-service.default.example.com" http://47.95.191.136/  
    Hello Login Service
```

如果想要在浏览器中访问 login-service 服务需要先做 host 绑定，把域名 login-service.default.example.com 指向 47.95.191.136 才行。这种方式还不能对外提供服务。下面接着介绍一下如何把默认的 example.com 改成我们自己的域名。

使用自定义主域名

在阿里云 Knative 用户可以通过控制台配置自定义域名，并基于 Path 和 Header 进行路由转发设置。如图所示：



假设我们自己的域名是: knative.kuberun.com, 现在执行 `kubectl edit cm config-domain --namespace knative-serving` 如下图所示, 添加 knative.kuberun.com 到 ConfigMap 中, 然后保存退出就完成了自定义主域名的配置。

```
# This block is not actually functional configuration,
# but serves to illustrate the available configuration
# options and document them in a way that is accessible
# to users that `kubectl edit` this config map.
#
# These sample configuration options may be copied out of
# this example block and unindented to be in the data block
# to actually change the configuration.

# Default value for domain.
# Although it will match all routes, it is the least-specific rule so it
# will only be used if no other domain matches.
example.com: |

# These are example settings of domain.
# example.org will be used for routes having app=nonprofit.
example.org: |
  selector:
    app: nonprofit

# Routes having domain suffix of 'svc.cluster.local' will not be exposed
# through Ingress. You can define your own label selector to assign that
# domain suffix to your Route here, or you can set the label
# "serving.knative.dev/visibility=cluster-local"
# to achieve the same effect. This shows how to make routes having
# the label app=secret only exposed to the local cluster.
svc.cluster.local: |
  selector:
    app: secret
  knative.kuberun.com: ""
kind: ConfigMap
metadata:
  creationTimestamp: "2019-11-04T09:39:08Z"
  labels:
    serving.knative.dev/release: v0.10.0
  name: config-domain
  namespace: knative-serving
  resourceVersion: "263861887"
  selfLink: /api/v1/namespaces/knative-serving/configmaps/config-domain
  uid: f32b169e-fee6-11e9-a166-befc39544974
```

再来看一下 Knative Service 的域名, 如下所示已经生效了。

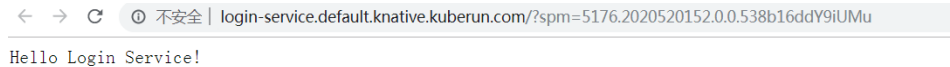
```
└─# kubectl -n default get ksvc
NAME      URL                                                                                                     LATESTCREATED
LATESTREADY READY  REASON
login-service http://login-service.default.knative.kuberun.com  login-
service-wsnvc login-service-wsnvc  True
```

泛域名解析

Knative Service 默认生成域名的规则是 `servicename.namespace.use-domain`。所以不同的 namespace 会生成不同的子域名，每一个 Knative Service 也会生成一个唯一的子域名。为了保证所有的 Service 服务都能在公网上面访问到，需要做一个泛域名解析。把 `*.knative.kuberun.com` 解析到 Istio Gateway 47.95.191.136 上面去。如果你是在阿里云（万网）上面购买的域名，你可以通过如下方式配置域名解析：

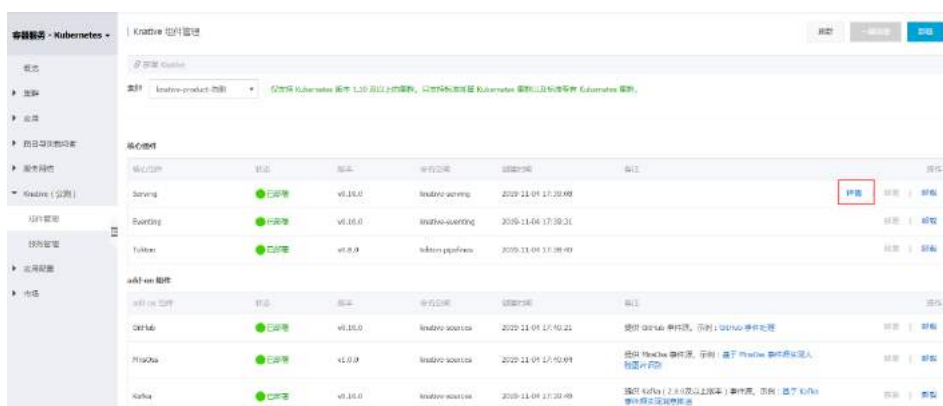


现在直接通过浏览器访问 <http://login-service.default.knative.kuberun.com/> 就可以直接看到 login-service 服务了：

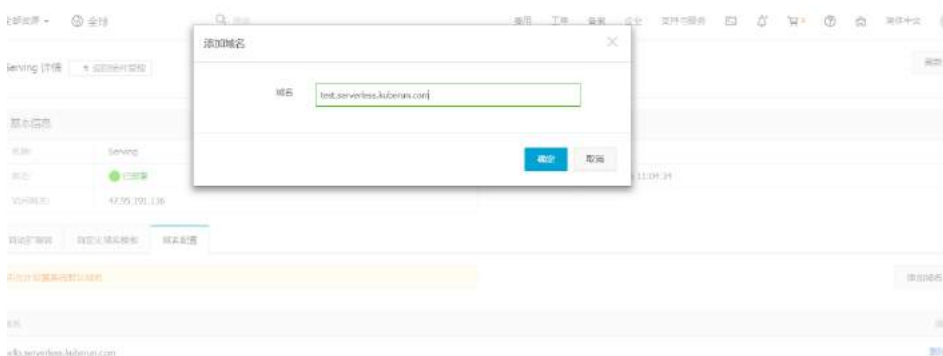


自定义域名

登录阿里云容器服务控制台，进入【Knative】-【组件管理】，点击 Serving 组件【详情】。



进入详情之后，选择域名配置，添加自定义域名：test.knative.kuberun.com。点击【确定】进行保存。

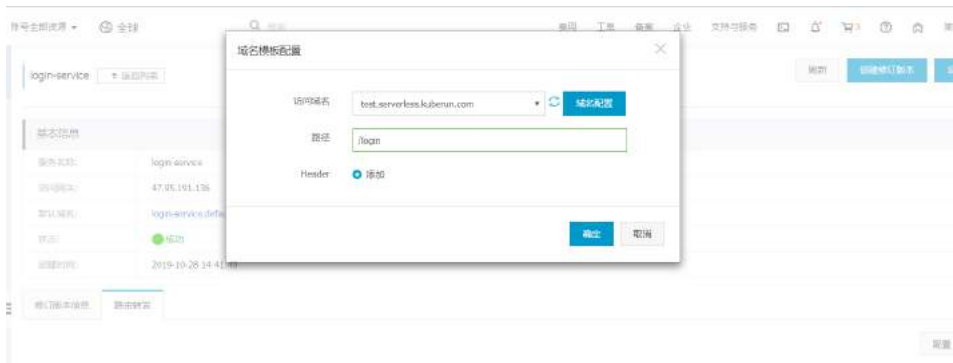


配置路由转发

进入【Knative】-【服务管理】控制台，选择对应的服务。这里我们对 Log-in-Service 服务 以及 Search-Service 服务分别设置不同的 Path 进行访问。

Login-Service 服务路由转发配置

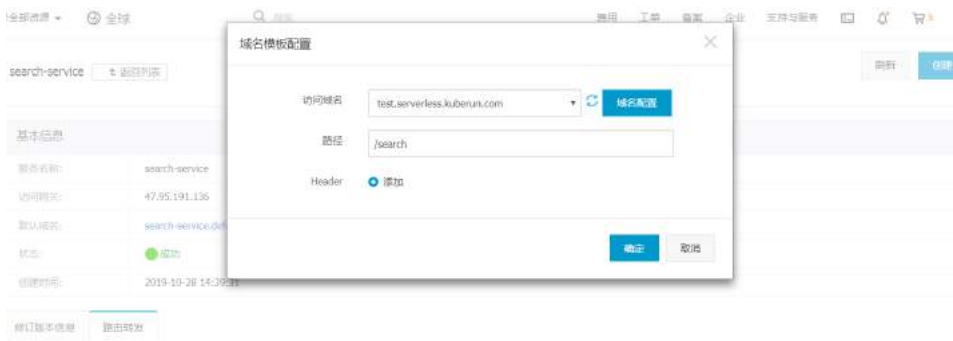
选择 Login-Service 服务，选择 路由转发 页签，点击 配置，选择 test.knative.kuberun.com 域名，配置路径: /login。点击 确定 进行保存。



接下来继续配置 Search-Service 服务路由规则。

Search-Service 服务路由转发配置

选择 Search-Service 服务，选择 路由转发 页签，点击 配置，选择 test.knative.kuberun.com 域名，配置路径: /search。点击 确定 进行保存。



服务访问

以上路由转发配置完成之后，我们开始测试一下服务访问：

在浏览器中输入：<http://test.knative.kuberun.com/login> 可以看到输出：Hello Login Service!

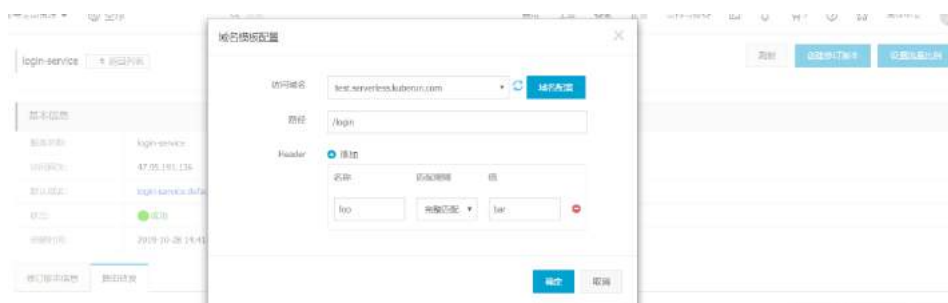
← → ↻ ① 不安全 | test.serverless.kuberun.com/login
Hello Login Service!

在浏览器中输入: <http://test.knative.kuberun.com/search> 可以看到输出:
Hello Search Service!

← → ↻ ① 不安全 | test.serverless.kuberun.com/search
Hello Search Service!

基于 Path + Header 进行路由转发

选择 Login-Service 服务, 选择 路由转发 页签, 点击 配置, 这里我们加上 Header 配置: foo=bar。点击 确定 进行保存。



访问 <http://test.knative.kuberun.com/login> 发现服务 404 不可访问。

← → ↻ ① test.serverless.kuberun.com/login



找不到 test.serverless.kuberun.com 的网页

找不到与以下网址对应的网页:

<http://test.serverless.kuberun.com/login>

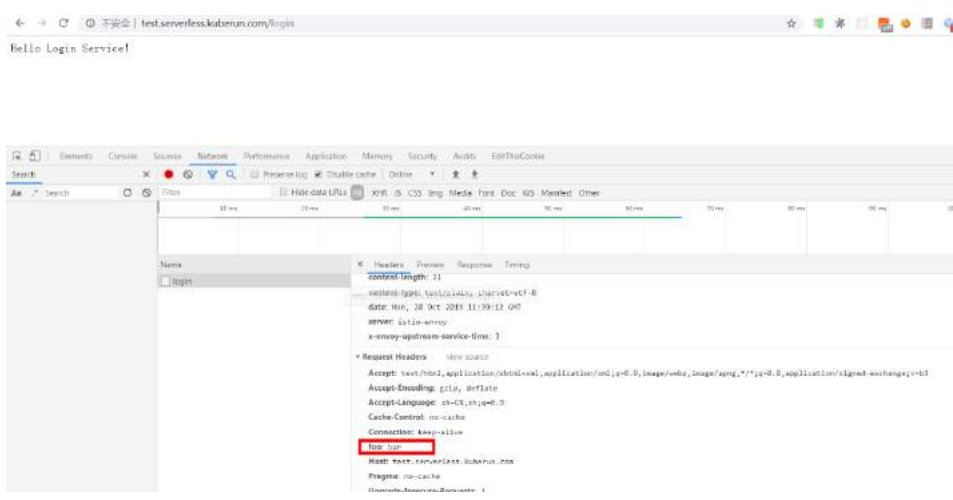
请在 Google 中搜索“test serverless kube run log”

HTTP ERROR 404

说明基于 Header 是生效的，下面我们在访问请求中通过 ModHeader 插件配置上 Header: foo=bar.

Name	Value
Request Headers	
foo	bar

配置完成之后，我们再一次访问服务：<http://test.knative.kuberun.com/login>。



服务访问 OK。这样我们就完成了基于 Path + Header 路由转发配置

总结

以上主要围绕 Knative Service 域名配置展开介绍了 Knative Serving 的路由管理，并且通过阿里云 Knative 控制台让你更轻松、快捷的实现自定义域名及路由规则，以打造生产可用的服务访问。

| WebSocket 和 gRPC 服务

虽然说 Knative 默认就支持 WebSocket 和 gRPC，但在使用中发现有些时候想要把自己的 WebSocket 或 gRPC 部署到 Knative 中还是会有各种不顺利的地方，尽管最后排查发现大多都是自己的程序问题或者是配置错误导致的。为了方便大家做验证，这里就分别给出一个 WebSocket 的例子和一个 gRPC 的例子。当我们需要在生产或者测试环境部署相关服务的时候可以使用本文给出的示例进行 Knative 服务的测试。

WebSocket

如果自己手动的配置 Istio Gateway 支持 WebSocket 就需要开启 websocket-Upgrade 功能。但使用 Knative Serving 部署其实就自带了这个能力。本示例的完整代码放在 <https://github.com/knative-sample/websocket-chat/tree/b1.0>，这是一个基于 WebSocket 实现的群聊的例子。使用浏览器连接到部署的服务中就可以看到一个接收信息的窗口和发送信息的窗口。当你发出一条信息以后所有连接进来的用户都能收到你的消息。所以你可以使用两个浏览器窗口分别连接到服务中，一个窗口发送消息一个窗口接收消息，以此来验证 WebSocket 服务是否正常。

本示例是在 [gorilla/websocket](#) 基础之上进行了一些优化：

- 代码中添加了 vendor 依赖，你下载下来就可以直接使用
- 添加了 Dockerfile 和 Makefile 可以直接编译二进制和制作镜像
- 添加了 Knative Service 的 yaml 文件 (service.yaml)，你可以直接提交到 Knative 集群中使用
- 也可以直接使用编译好的镜像 `registry.cn-hangzhou.aliyuncs.com/knative-sample/websocket-chat:2019-10-15`

Knative Service 配置

```

apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: websocket-chat
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/websocket-
chat:2019-10-15
          ports:
            - name: http1
              containerPort: 8080

```

代码 clone 下来以后执行 `kubectl apply -f service.yaml` 把服务部署到 Knative 中，然后直接访问服务地址即可使用。

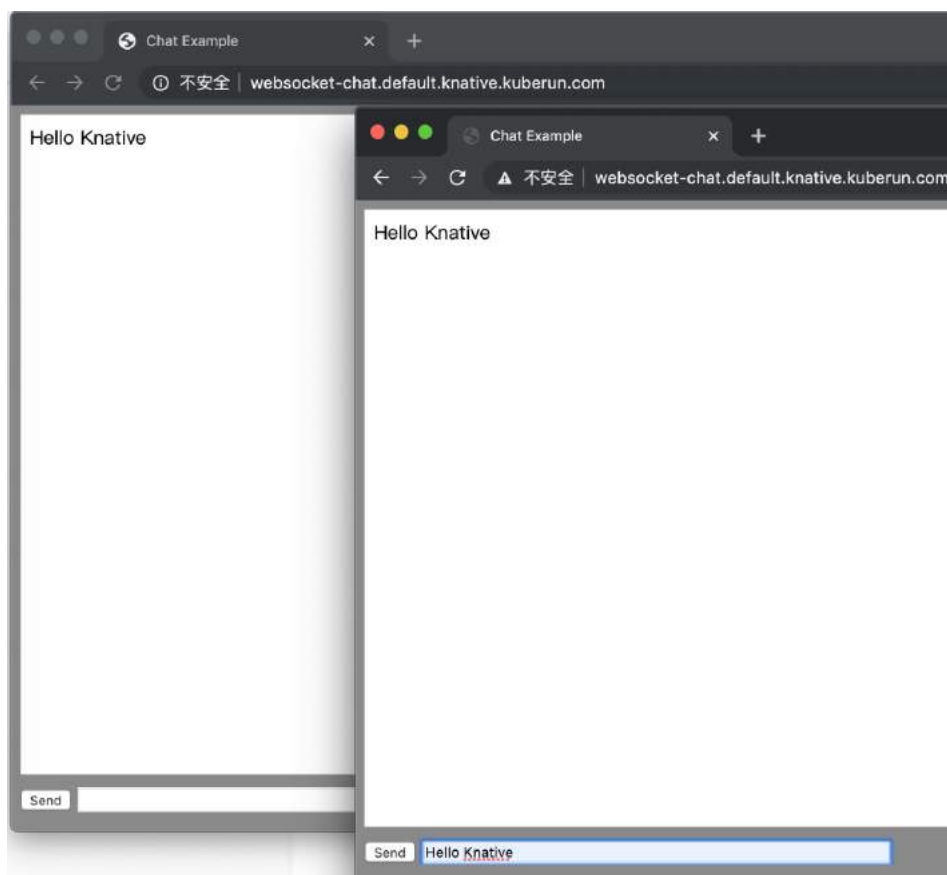
查看 `ksvc` 列表，并获取访问域名

```

└─ # kubectl get ksvc
NAME                                URL
LATESTCREATED                      LATESTREADY
READY   REASON
websocket-chat                      http://websocket-chat.default.knative.
kuberun.com                         websocket-chat-tp4ph                websocket-
chat-tp4ph                          True

```

现在使用浏览器打开 <http://websocket-chat.default.knative.kuberun.com> 即可看到群聊窗口



打开两个窗口，在其中一个窗口发送一条消息，另外一个窗口通过 WebSocket 也收到了这条消息。

gRPC

gRPC 不能通过浏览器直接访问，需要通过 client 端和 server 端进行交互。本示例的完整代码放在 <https://github.com/knative-sample/grpc-ping-go/tree/b1.0>，本示例会给一个可以直接使用的镜像，测试 gRPC 服务。

Knative Service 配置

```
apiVersion: serving.knative.dev/v1
```

```

kind: Service
metadata:
  name: grpc-ping
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/grpc-ping-
          go:2019-10-15
          ports:
            - name: h2c
              containerPort: 8080

```

代码 clone 下来以后执行 `kubectl apply -f service.yaml` 把服务部署到 Knative 中。

获取 ksvc 列表和访问域名：

```

└─# kubectl get ksvc
NAME                                URL
LATESTCREATED                      LATESTREADY
READY    REASON
grpc-ping                                http://grpc-ping.default.knative.kuberun.
com                                grpc-ping-plzrv                                grpc-ping-
plzrv                                True

```

现在我们已经知道 gRPC server 的地址是 `grpc-ping.default.knative.kuberun.com` 端口是 80 那么我们可以发起测试请求：

```

└─# docker run --rm registry.cn-hangzhou.aliyuncs.com/knative-sample/grpc-
ping-go:2019-10-15 /client -server_addr="grpc-ping.default.knative.kuberun.
com:80" -insecure
2019/11/06 13:45:46 Ping got hello - pong
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.43385349 +0800 CST
m=+52.762218971
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.433885075 +0800 CST
m=+52.762250507
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.433894386 +0800 CST
m=+52.762259840
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.433902205 +0800 CST
m=+52.762267652
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.433909964 +0800 CST
m=+52.762275418
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.433926773 +0800 CST

```

```
m=+52.762292207
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.43393548 +0800 CST
m=+52.762300916
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.433940721 +0800 CST
m=+52.762306150
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.433954408 +0800 CST
m=+52.762319841
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.433959768 +0800 CST
m=+52.762325212
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.433964951 +0800 CST
m=+52.762330381
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.433973361 +0800 CST
m=+52.762338796
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.433986818 +0800 CST
m=+52.762352250
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.433994339 +0800 CST
m=+52.762359790
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.434001938 +0800 CST
m=+52.762367370
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.434016244 +0800 CST
m=+52.762381708
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.434024768 +0800 CST
m=+52.762390227
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.434037418 +0800 CST
m=+52.762402862
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.434045691 +0800 CST
m=+52.762411130
2019/11/06 13:45:46 Got pong 2019-11-06 13:45:46.434053459 +0800 CST
m=+52.762418894
```

小结

本文通过两个例子分别展示了 WebSocket 和 gRPC 的部署方法：

- WebSocket 示例通过一个 chat 的方式展示发送和接受消息
- gRPC 通过启动一个 client 的方式展示 gRPC 远程调用的过程

参考资料

- WebSocket 示例代码 <https://github.com/knative-sample/websocket-chat/tree/b1.0>
- gRPC 示例代码 <https://github.com/knative-sample/grpc-ping-go/tree/b1.0>

Serving Client 介绍

通过前面的一系列文章你已经知道如何基于 `kubectl` 来操作 Knative 的各种资源。但是如果想要在项目中集成 Knative 仅仅使用 `kubectl` 这种命令的方式是不够的。是需要代码中基于 Knative Serving SDK 进行集成开发。本文就从 Knative Serving SDK 入手，介绍如何基于 Knative SDK 进行 serverless 开发。

Golang Context

在正式开始介绍 Knative Serving SDK 之前我们先简单的介绍一下 Golang Context 的机理，因为在 Knative Serving 中 client、Informer 的初始化和信息传递完全是基于 Golang Context 实现的。

Golang 是从 1.7 版本开始引入的 Context，Golang 的 Context 可以很好的简化多个 goroutine 之间以及请求域间的数据传递、取消信号和截至时间等相关操作。Context 主要有两个作用：

1. 传输必要的的数据
2. 进行协调控制，比如终止 goroutine、设置超时时间等

Context 定义

Context 本身是一个接口

```
type Context interface {  
    Deadline() (deadline time.Time, ok bool)  
    Done() <-chan struct{}  
    Err() error  
    Value(key interface{}) interface{}  
}
```

这个接口中定义了四个方法，下面分别介绍如下：

- Deadline 方法是获取设置的截止时间的意思，到了这个时间点，Context 会自动发起取消请求
- Done 方法返回一个只读的 chan，如果该方法返回的 chan 可以读取，则意味着 parent Context 已经发起了取消请求，此时应该做清理操作，然后退出 goroutine 并释放资源
- Err 方法返回取消的错误原因
- Value 方法获取该 Context 上绑定的值，是一个键值对。所以要通过一个 Key 才可以获取对应的值，这个值是线程安全的

关于 Context 主要记住一点：可以通过 Value 传递数据，Value 是一个键值对结构。更多详细的介绍可以参见下面这些文章：

- [Concurrency Patterns in Go](#)
- [How to correctly use context.Context in Go 1.7](#)
- [Using context cancellation in Go](#)
- [Go Context](#)

Knative Serving client 源码浅析

在 Context 的这些特性中，Knative Serving 中重度依赖的是 Value 功能。以 Service 的 Informer 初始化为例进行说明，[这里可以看到源码](#)。

Informer “构造函数”是在 init 函数中自动注册到 injection.Default 中的。当 Informer “构造函数”被调用之后会自动把生成的 Informer 注入到 Context 中 context.WithValue(ctx, Key{}, inf), inf.Informer()。

```

18
19 package service
20
21 import (
22     "context"
23
24     controller "knative.dev/pkg/controller"
25     injection "knative.dev/pkg/injection"
26     logging "knative.dev/pkg/logging"
27     v1alpha1 "knative.dev/serving/pkg/client/informers/externalversions/serving/v1alpha1"
28     factory "knative.dev/serving/pkg/client/injection/informers/serving/factory"
29 )
30
31 func init() {
32     injection.Default.RegisterInformer(withInformer)
33 }
34
35 // Key is used for associating the Informer inside the context.Context.
36 type Key struct{}
37
38 func withInformer(ctx context.Context) (context.Context, controller.Informer) {
39     f := factory.Get(ctx)
40     inf := f.Serving().V1alpha1().Services()
41     return context.WithValue(ctx, Key{}, inf), inf.Informer()
42 }
43
44 // Get extracts the typed informer from the context.
45 func Get(ctx context.Context) v1alpha1.ServiceInformer {
46     untyped := ctx.Value(Key{})
47     if untyped == nil {
48         logging.FromContext(ctx).FatalF(
49             "Unable to fetch %T from context.", (v1alpha1.ServiceInformer)(nil))
50     }
51     return untyped.(v1alpha1.ServiceInformer)
52 }

```

从上图中可以看到，Informer 初始化的时候需要调用 factory，而 factory 本身是从 Context 中获取的。下面我发再看看 factory 是怎么初始化的。

[factory 的初始化](#)

```

21 import {
22     "context"
23
24     controller "knative.dev/pkg/controller"
25     injection "knative.dev/pkg/injection"
26     logging "knative.dev/pkg/logging"
27     externalversions "knative.dev/serving/pkg/client/informers/externalversions"
28     client "knative.dev/serving/pkg/client/injection/client"
29 }
30
31 func init() {
32     injection.Default.RegisterInformerFactory(withInformerFactory)
33 }
34
35 // Key is used as the key for associating information with a context.Context.
36 type Key struct{}
37
38 func withInformerFactory(ctx context.Context) context.Context {
39     c := client.Get(ctx)
40     return context.WithValue(ctx, Key{},
41         externalversions.NewSharedInformerFactory(c, controller.GetResyncPeriod(ctx)))
42 }

```

可以发现 factory 也是把“构造函数”注册到 injection.Default 中，并且会把生成的 SharedInformerFactory 注入到 Context 中。而且 factory 中使用的 client(链接 kube-apiserver 使用的对象) 也是从 Context 获取到的。

可以说 Knative Serving SDK 初始化的过程是面向 Context 编程的。关键对象是自动注入到 Context，在使用的时候从 Context 中取出。

顺带提一点，Knative Serving 的日志对象也是在 Context 保存的，当需要打印日志的时候先通过 logger := logging.FromContext(ctx) 从 Context 中拿到 logger，然后就可以使用了。这样做的好处是通过管理 logger 对象，比如做 trace 功能。如下所示是基于 logger 打印出来的日志，可以看到对于同一个请求的处理是可以通过 traceID 进行追踪的。下面这段日志都是对 577f8de5-cec9-4c17-84f7-f08d39f40127 这个 trace 的处理。

```

{"level":"info","ts":"2019-08-28T20:24:39.871+0800","caller":"controller/service.go:67","msg":"Reconcile: default/helloworld-go","knative.dev/traceid":"be5ec711-6ca3-493c-80ed-dddfa21fd576","knative.dev/key":"default/helloworld-go"}
{"level":"info","ts":"2019-08-28T20:24:39.871+0800","caller":"controller/controller.go:339","msg":"Reconcile succeeded. Time taken: 487.347µs.","knative.dev/traceid":"90653eda-644b-4b1e-8bdb-4a1a7a7ff0d8","knative.dev/key":"eci-test/helloworld-go"}
{"level":"info","ts":"2019-08-28T20:24:39.871+0800","caller":"controller/service.go:106","msg":"service: default/helloworld-go route:

```

```
default/helloworld-go ", "knative.dev/traceid": "be5ec711-6ca3-493c-80ed-
dddfa21fd576", "knative.dev/key": "default/helloworld-go"}
{"level": "info", "ts": "2019-08-28T20:24:39.872+0800", "caller": "controller/
service.go:67", "msg": "Reconcile: eci-test/helloworld-go", "knative.dev/
traceid": "22f6c77d-8365-4773-bd78-e011ccb2fa3d", "knative.dev/key": "eci-test/
helloworld-go"}
{"level": "info", "ts": "2019-08-28T20:24:39.872+0800", "caller": "controll
er/service.go:116", "msg": "service: default/helloworld-go revisions: 1
", "knative.dev/traceid": "be5ec711-6ca3-493c-80ed-dddfa21fd576", "knative.dev/
key": "default/helloworld-go"}
{"level": "info", "ts": "2019-08-28T20:24:39.872+0800", "caller": "controller/
service.go:118", "msg": "service: default/helloworld-go revision: default/
helloworld-go-cgt65 ", "knative.dev/traceid": "be5ec711-6ca3-493c-80ed-
dddfa21fd576", "knative.dev/key": "default/helloworld-go"}
{"level": "info", "ts": "2019-08-28T20:24:39.872+0800", "caller": "
controller/controller.go:339", "msg": "Reconcile succeeded. Time
taken: 416.527µs.", "knative.dev/traceid": "be5ec711-6ca3-493c-80ed-
dddfa21fd576", "knative.dev/key": "default/helloworld-go"}
{"level": "info", "ts": "2019-08-28T20:24:39.872+0800", "caller": "controller/
service.go:106", "msg": "Reconcile succeeded. Time taken: 416.527µs."}
```

使用 Knative Serving SDK

介绍完 Knative Serving client 的初始化过程，下面我们看一下应该如何在代码中用 Knative Serving SDK 进行编码。

示例参见: <https://github.com/knative-sample/serving-controller/blob/b1.0/cmd/app/app.go>

这个示例中首先使用配置初始化 `*zap.SugaredLogger` 对象，然后基于 `ctx := signals.NewContext()` 生成一个 Context。`signals.NewContext()` 作用是监听 SIGINT 信号，也就是处理 CTRL+c 指令。这里用到了 Context 接口的 Done 函数机制。

构造 Informer

接着使用 `ctx, informers := injection.Default.SetupInformers(ctx, cfg)` 构造出所有的 informer，然后调用下面这段代码执行注入，把 informer 注入到 Context 中。

```
// Start all of the informers and wait for them to sync.
logger.Info("Starting informers.")
if err := controller.StartInformers(ctx.Done(), informers...); err != nil {
    logger.Fatalw("Failed to start informers", err)
}
```



```
// StartInformers kicks off all of the passed informers and then waits for all
// of them to synchronize.
func StartInformers(stopCh <-chan struct{}, informers ...Informer) error {
    for _, informer := range informers {
        informer := informer
        go informer.Run(stopCh)
    }

    for i, informer := range informers {
        if ok := cache.WaitForCacheSync(stopCh, informer.HasSynced); !ok {
            return fmt.Errorf("failed to wait for cache at index %d to sync", i)
        }
    }
    return nil
}
```

从 Context 中获取 Informer

实例代码: [https://github.com/knative-sample/serving-controller/blob/v0.1/](https://github.com/knative-sample/serving-controller/blob/v0.1/pkg/controller/controller.go)

[pkg/controller/controller.go](#)

```
35 // Registers eventhandlers to enqueue events
36 func NewController(
37     ctx context.Context,
38 ) *controller.Impl {
39     logger := logging.FromContext(ctx)
40     serviceInformer := kserviceinformer.Get(ctx)
41     routeInformer := routeinformer.Get(ctx)
42     configurationInformer := configurationinformer.Get(ctx)
43     revisionInformer := revisioninformer.Get(ctx)
44
45     c := &Reconciler{
46         serviceLister:    serviceInformer.Lister(),
47         revisionLister:   revisionInformer.Lister(),
48         revisionClientSet: servingclient.Get(ctx),
49         routeLister:      routeInformer.Lister(),
50     }
51     impl := controller.NewImpl(c, logger, ReconcilerName)
52
53     logger.Info("Setting up event handlers")
54     serviceInformer.Informer().AddEventHandler(controller.HandleAll(impl.Enqueue))
55
56     configurationInformer.Informer().AddEventHandler(cache.FilteringResourceEventHandler{
57         FilterFunc: controller.Filter(v1alpha1.SchemeGroupVersion.WithKind("Service")),
58         Handler:    controller.HandleAll(impl.EnqueueControllerOf),
59     })
60
61     routeInformer.Informer().AddEventHandler(cache.FilteringResourceEventHandler{
62         FilterFunc: controller.Filter(v1alpha1.SchemeGroupVersion.WithKind("Service")),
63         Handler:    controller.HandleAll(impl.EnqueueControllerOf),
64     })
65
66     return impl
67 }
```

如上所示，所有的 informer 都是从 Context 中获取的。

最后 Controller 初始化一个 Reconciler 接口，接口的定义如下，里面只有一个 Reconcile 函数。这个使用方式和 sigs.k8s.io/controller-runtime 使用的逻辑是一样的。如果你之前写过 Operator 之类的功能，对这个操作应该不会陌生。

```
// Reconciler is the interface that controller implementations are expected
// to implement, so that the shared controller.Impl can drive work through
// it.
type Reconciler interface {
    Reconcile(ctx context.Context, key string) error
}
```

在 Reconcile 中调用 Knative API

代码示例: <https://github.com/knative-sample/serving-controller/blob/v0.1/pkg/controller/service.go>

```
55 // Reconcile compares the actual state with the desired, and attempts to
56 // converge the two. It then updates the Status block of the Service resource
57 // with the current status of the resource.
58 func (c *Reconciler) Reconcile(ctx context.Context, key string) error {
59     // Convert the namespace/name string into a distinct namespace and name
60     namespace, name, err := cache.SplitMetaNamespaceKey(key)
61     if err != nil {
62         c.logger.Errorf("invalid resource key: %s", key)
63         return nil
64     }
65     logger := logging.FromContext(ctx)
66
67     logger.Infof("Reconcile: %s/%s", namespace, name)
68
69     // Get the Service resource with this namespace/name
70     original, err := c.servicelister.Services(namespace).Get(name)
71     if apierrs.IsNotFound(err) {
72         // The resource may no longer exist, in which case we stop processing.
73         logger.Errorf("service %q in work queue no longer exists", key)
74         return nil
75     } else if err != nil {
76         return err
77     }
78
79     if original.GetDeletionTimestamp() != nil {
80         return nil
81     }
82
83     // Don't modify the informers copy
84     service := original.DeepCopy()
85
86     // Reconcile this copy of the service and then write back any status
```

```

87:         // updates regardless of whether the reconciliation errored out.
88:         if reconcileErr := c.reconcile(ctx, service); reconcileErr != nil {
89:             c.Recorder.Event(service, corev1.EventTypeWarning, "InternalError", reconcileErr.Error())
90:             logger.Errorf("Reconcile service: %s/%s error: %s ", service.Namespace, service.Name, reconcileErr)
91:             return reconcileErr

```

现在就可以在 Reconcile 中通过 `c.serviceLister.Services(namespace).Get(name)` 这种方式直接操作 Serving 资源了。

至此已经把基于 Knative Serving 开发 Serverless 应用的关键脉梳理了一遍。更详细的代码示例请参见：<https://github.com/knative-sample/serving-controller/tree/b1.0>，这里面有完整可以运行的代码。

本文提到的示例代码是基于 Knative 0.10 版本开发的，这个版本的 SDK 需要使用 Golang 1.13 才行。

另外除了文章中提到的示例代码，提供另外一个例子 (<https://github.com/knative-sample/serving-sdk-demo/tree/b1.0>) 这个例子是直接使用 Knative SDK 创建一个 ksvc，这两个例子的侧重点有所不同。可以参考着看。

小结

本文从 Knative Serving client 的初始化过程开始展开，介绍了 Knative informer 的设计以及使用方法。通过本文你可以了解到：

- Knative Serving client 的设计思路
- 如何基于 Knative Serving SDK 进行二次开发
- 通过 Knative Serving 学习到如何面向 Context 编程
- Knative Serving 集成开发示例：<https://github.com/knative-sample/serving-controller>

参考资料

- <https://github.com/knative-sample/serving-sdk-demo/tree/b1.0>
- <https://github.com/knative-sample/serving-controller/tree/b1.0>

Eventing 进阶

定义无处不在的事件 – CloudEvent

背景

Event 事件无处不在，然而每个事件提供者产生的事件各部相同。由于缺乏事件的统一描述，对于事件的开发者来说需要不断的重复学习如何消费不同类型的事件。这也限制了类库、工具和基础设施在跨环境（如 SDK、事件路由或跟踪系统）提供事件数据方面的潜力。从事件数据本身实现的可移植性和生产力上受到了阻碍。

什么是 CloudEvents

CloudEvents 是一种规范，用于以通用格式描述事件数据，以提供跨服务、平台和系统的交互能力。

事件格式指定了如何使用某些编码格式来序列化 CloudEvent。支持这些编码的兼容 CloudEvents 实现必须遵循在相应的事件格式中指定的编码规则。所有实现都必须支持 JSON 格式。

协议规范

命名规范

CloudEvents 属性名称必须由 ASCII 字符集的小写字母（“a”到“z”）或数字（“0”到“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，长度不应超过 20 个字符。

术语定义

本规范定义如下术语：

- Occurrence: “Occurrence” 是指在软件系统运行期间捕获描述信息。
- Event: “Event” 是表示事件及其上下文的数据记录。
- Context: Context 表示上下文，元数据将封装在 Context 属性中。应用程序代码可以使用这些信息来标识事件与系统或其他事件之间的关系。
- Data: 实际事件中有效信息载荷。
- Message: 事件通过 Message 从数据源传输到目标地址。
- Protocol: 消息可以通过各种行业标准协议（如 http、amqp、mqtt、smtp）、开源协议（如 kafka、nats）或平台 / 供应商特定协议（aws-kinesis、azure-event-grid）进行传递。

上下文属性 (Context Attributes)

符合本规范的每个 CloudEvent 必须包括根据需要指定的上下文属性，并且可以包括一个或多个可选的上下文属性。

参考示例：

```
specversion: 0.2
type: dev.knative.k8s.event
source: /apis/serving.knative.dev/v1alpha1/namespaces/default/routes/sls-
cloudevent
id: 269345ff-7d0a-11e9-b1f1-00163f005e02
time: 2019-05-23T03:23:36Z
contenttype: application/json
```

- type: 事件类型，通常此属性用于路由、监控、安全策略等。
- specversion: 表示 CloudEvents 规范的版本。引用 0.2 版本的规范时，事件生产者必须使用 0.2 设置此值。
- source: 表示事件的产生者，也就是事件源。
- id: 事件的 id。

- time: 事件的产生时间。
- contentType: 表示 Data 的数据内容格式。

扩展属性 (Extension Attributes)

CloudEvents 生产者可以在事件中包含其他上下文属性，这些属性可能用于与事件处理相关的辅助操作。

Data

正如术语 Data 所定义的，CloudEvents 产生具体事件的内容信息封装在数据属性中。例如，KubernetesEventSource 所产生的 CloudEvent 的 Data 信息如下：

```
data:
{
  "metadata": {
    "name": "event-display.15a0a2b54007189b",
    "namespace": "default",
    "selfLink": "/api/v1/namespaces/default/events/event-
display.15a0a2b54007189b",
    "uid": "9195ff11-7b9b-11e9-b1f1-00163f005e02",
    "resourceVersion": "18070551",
    "creationTimestamp": "2019-05-21T07:39:30Z"
  },
  "involvedObject": {
    "kind": "Route",
    "namespace": "default",
    "name": "event-display",
    "uid": "31c68419-675b-11e9-a087-00163e08f3bc",
    "apiVersion": "serving.knative.dev/v1alpha1",
    "resourceVersion": "9242540"
  },
  "reason": "InternalServerError",
  "message": "Operation cannot be fulfilled on clusteringresses.networking.
internal.knative.dev \"route-31c68419-675b-11e9-a087-00163e08f3bc\": the
object has been modified; please apply your changes to the latest version and
try again",
  "source": {
    "component": "route-controller"
  },
  "firstTimestamp": "2019-05-21T07:39:30Z",
  "lastTimestamp": "2019-05-26T07:10:51Z",
```

```

    "count": 5636,
    "type": "Warning",
    "eventTime": null,
    "reportingComponent": "",
    "reportingInstance": ""
  }

```

实现

以 Go SDK 实现 CloudEvent 0.2 规范为例：

事件接收服务

- 导入 cloudevents

```
import "github.com/cloudevents/sdk-go"
```

– 通过 HTTP 协议接收 CloudEvent 事件

```

func Receive(event cloudevents.Event) {
    fmt.Printf("cloudevents.Event\n%s", event.String())
}

func main() {
    c, err := cloudevents.NewDefaultClient()
    if err != nil {
        log.Fatalf("failed to create client, %v", err)
    }
    log.Fatal(c.StartReceiver(context.Background(), Receive));
}

```

事件发送服务

- 创建一个基于 0.2 协议的 CloudEvent 事件

```

event := cloudevents.NewEvent()
event.SetID("ABC-123")
event.SetType("com.cloudevents.readme.sent")
event.SetSource("http://localhost:8080/")
event.SetData(data)

```

- 通过 HTTP 协议发送这个 CloudEvent

```
t, err := cloudevents.NewHTTPTransport(  
    cloudevents.WithTarget("http://localhost:8080/"),  
    cloudevents.WithEncoding(cloudevents.HTTPBinaryV02),  
)  
if err != nil {  
    panic("failed to create transport, " + err.Error())  
}  
  
c, err := cloudevents.NewClient(t)  
if err != nil {  
    panic("unable to create cloudevent client: " + err.Error())  
}  
  
if err := c.Send(ctx, event); err != nil {  
    panic("failed to send cloudevent: " + err.Error())  
}
```

这样我们就通过 Go SDK 方式实现了 CloudEvent 事件的发送和接收

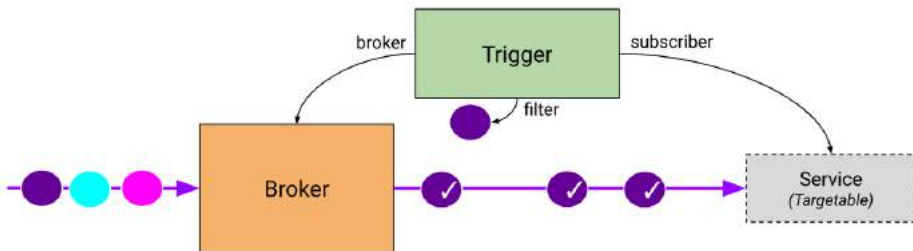
关于 Broker/Trigger 事件模型

Broker 和 Trigger

从 v0.5 开始, Knative Eventing 定义 Broker 和 Trigger 对象, 从而能方便的对事件进行过滤 (亦如通过 ingress 和 ingress controller 对网络流量的过滤一样)。

- Broker 提供一个事件集, 可以通过属性选择该事件集。它负责接收事件并将其转发给由一个或多个匹配 Trigger 定义的订阅者。
- Trigger 描述基于事件属性的过滤器。同时可以根据需要创建多个 Trigger。

如图:



当前实现方式

Namespace

通过 Namespace Reconciler (代码: eventing/pkg/reconciler/v1alpha1/namespace/namespace.go) 创建 broker。Namespace Reconciler 会查询所有带 knative-eventing-injection: enabled 标签的 namespace。如果存在这样标签的 namespace, 那么 Namespace Reconciler 将会进行如下处理操作:

1. 创建 Broker 过滤器的 ServiceAccount: eventing-broker-filter
2. 通过 RoleBinding 确保 ServiceAccount 的 RBAC 权限
3. 创建名称为 default 的 Broker

```
// newBroker creates a placeholder default Broker object for Namespace 'ns'.
func newBroker(ns *corev1.Namespace) *v1alpha1.Broker {
    return &v1alpha1.Broker{
        ObjectMeta: metav1.ObjectMeta{
            Namespace: ns.Name,
            Name:       defaultBroker,
            Labels:     injectedLabels(),
        },
    }
}
```

Broker (事件代理)

通过 Broker Reconciler 进行处理 broker, 对于每一个 broker, 会进行一下处理操作:

1. 创建 'trigger'Channel。所有在 Broker 中的 event 事件都会发送到这个 Channel, 所有的 Trigger 会订阅这个 Channel。
2. 创建 'filter'Deployment。这个 Deployment 会运行 cmd/broker/filter。其目的是处理与此 Broker 相关的所有 Trigger 的数据平面。说白了其实就做了两件事情, 从 Channel 中接收事件, 然后转发给事件的订阅者。
3. 创建 'filter' Kubernetes Service。通过该 Service 提供 'filter' Deployment 的服务访问。
4. 创建 'ingress' Deployment。这个 Deployment 会运行 cmd/broker/ingress。其目的是检查进入 Broker 的所有事件
5. 创建 'ingress' Kubernetes Service。通过该 Service 提供 'Ingress' Deployment 的服务访问。
6. 创建 'ingress' Channel。这是一个 Trigger 应答的 Channel。目的是将 Trigger 中返回的事件通过 Ingress Deployment 回写到 Broker。理想情况

下，其实不需要这个，可以直接将 Trigger 的响应发送给 k8s Service。但是作为订阅的场景，只允许我们向 Channel 发送响应信息，所以我们需要这个作为中介。

7. 创建 'ingress' Subscription。它通过 'ingress' Channel 来订阅 'ingress' Service

代码如下：

```
func (r *reconciler) reconcile(ctx context.Context, b *v1alpha1.Broker)
(reconcile.Result, error) {
    // 1. Trigger Channel is created for all events. Triggers will Subscribe
    to this Channel.
    // 2. Filter Deployment.
    // 3. Ingress Deployment.
    // 4. K8s Services that point at the Deployments.
    // 5. Ingress Channel is created to get events from Triggers back into
    this Broker via the
    //     Ingress Deployment.
    //     - Ideally this wouldn't exist and we would point the Trigger's reply
    directly to the K8s
    //     Service. However, Subscriptions only allow us to send replies to
    Channels, so we need
    //     this as an intermediary.
    // 6. Subscription from the Ingress Channel to the Ingress Service.

    if b.DeletionTimestamp != nil {
        // Everything is cleaned up by the garbage collector.
        return nil
    }

    if b.Spec.ChannelTemplate == nil {
        r.Logger.Error("Broker.Spec.ChannelTemplate is nil",
            zap.String("namespace", b.Namespace), zap.String("name", b.Name))
        return nil
    }

    gvr, _ := meta.UnsafeGuessKindToResource(b.Spec.ChannelTemplate.
GetObjectKind().GroupVersionKind())
    channelResourceInterface := r.DynamicClientSet.Resource(gvr).Namespace(b.
Namespace)
    if channelResourceInterface == nil {
        return fmt.Errorf("unable to create dynamic client for: %v", b.Spec.
ChannelTemplate)
    }
}
```

```

track := r.channelableTracker.TrackInNamespace(b)

triggerChannelName := resources.BrokerChannelName(b.Name, "trigger")
triggerChannelObjRef := corev1.ObjectReference{
    Kind:      b.Spec.ChannelTemplate.Kind,
    APIVersion: b.Spec.ChannelTemplate.APIVersion,
    Name:      triggerChannelName,
    Namespace: b.Namespace,
}
// Start tracking the trigger channel.
if err := track(triggerChannelObjRef); err != nil {
    return fmt.Errorf("unable to track changes to the trigger Channel:
    %v", err)
}

logging.FromContext(ctx).Info("Reconciling the trigger channel")
triggerChan, err := r.reconcileTriggerChannel(ctx,
channelResourceInterface, triggerChannelObjRef, b)
if err != nil {
    logging.FromContext(ctx).Error("Problem reconciling the trigger
channel", zap.Error(err))
    b.Status.MarkTriggerChannelFailed("ChannelFailure", "%v", err)
    return err
}
.....

filterDeployment, err := r.reconcileFilterDeployment(ctx, b)
if err != nil {
    logging.FromContext(ctx).Error("Problem reconciling filter
Deployment", zap.Error(err))
    b.Status.MarkFilterFailed("DeploymentFailure", "%v", err)
    return err
}
_, err = r.reconcileFilterService(ctx, b)
if err != nil {
    logging.FromContext(ctx).Error("Problem reconciling filter Service",
zap.Error(err))
    b.Status.MarkFilterFailed("ServiceFailure", "%v", err)
    return err
}
b.Status.PropagateFilterDeploymentAvailability(filterDeployment)

ingressDeployment, err := r.reconcileIngressDeployment(ctx, b,
triggerChan)
if err != nil {
    logging.FromContext(ctx).Error("Problem reconciling ingress
Deployment", zap.Error(err))
    b.Status.MarkIngressFailed("DeploymentFailure", "%v", err)

```

```

        return err
    }

    svc, err := r.reconcileIngressService(ctx, b)
    if err != nil {
        logging.FromContext(ctx).Error("Problem reconciling ingress Service",
zap.Error(err))
        b.Status.MarkIngressFailed("ServiceFailure", "%v", err)
        return err
    }
    b.Status.PropagateIngressDeploymentAvailability(ingressDeployment)
    b.Status.SetAddress(&apis.URL{
        Scheme: "http",
        Host:   names.ServiceHostName(svc.Name, svc.Namespace),
    })

    ingressChannelName := resources.BrokerChannelName(b.Name, "ingress")
    ingressChannelObjRef := corev1.ObjectReference{
        Kind:      b.Spec.ChannelTemplate.Kind,
        APIVersion: b.Spec.ChannelTemplate.APIVersion,
        Name:      ingressChannelName,
        Namespace: b.Namespace,
    }

    // Start tracking the ingress channel.
    if err = track(ingressChannelObjRef); err != nil {
        return fmt.Errorf("unable to track changes to the ingress Channel:
%v", err)
    }

    ingressChan, err := r.reconcileIngressChannel(ctx,
channelResourceInterface, ingressChannelObjRef, b)
    if err != nil {
        logging.FromContext(ctx).Error("Problem reconciling the ingress
channel", zap.Error(err))
        b.Status.MarkIngressChannelFailed("ChannelFailure", "%v", err)
        return err
    }
    b.Status.IngressChannel = &ingressChannelObjRef
    b.Status.PropagateIngressChannelReadiness(&ingressChan.Status)

    ingressSub, err := r.reconcileIngressSubscription(ctx, b, ingressChan,
svc)
    if err != nil {
        logging.FromContext(ctx).Error("Problem reconciling the ingress
subscription", zap.Error(err))
        b.Status.MarkIngressSubscriptionFailed("SubscriptionFailure", "%v",
err)
        return err
    }

```

```

    }
    b.Status.PropagateIngressSubscriptionReadiness(&ingressSub.Status)

    return nil
}

```

Broker 示例:

```

apiVersion: eventing.knative.dev/v1alpha1
kind: Broker
metadata:
  name: default
spec:
  channelTemplateSpec:
    apiVersion: messaging.knative.dev/v1alpha1
    kind: InMemoryChannel

```

Trigger (触发器)

通过 Trigger Reconciler 进行处理 trigger, 对于每一个 trigger, 会进行一下处理操作:

1. 验证 Broker 是否存在
2. 获取 对应 Broker 的 Trigger Channel、Ingress Channel 以及 Filter Service
3. 确定订阅者的 URI
4. 创建一个从 Broker 特定的 Channel 到这个 Trigger 的 kubernetes Service 的订阅。reply 被发送到 Broker 的 ingress Channel。
5. 检查是否包含 knative.dev/dependency 的注释。

代码如下:

```

func (r *reconciler) reconcile(ctx context.Context, t *v1alpha1.Trigger)
error {
    .....
    // 1. Verify the Broker exists.
    // 2. Get the Broker's:
    //    - Trigger Channel

```

```

// - Ingress Channel
// - Filter Service
// 3. Find the Subscriber's URI.
// 4. Creates a Subscription from the Broker's Trigger Channel to this
Trigger via the Broker's
// Filter Service with a specific path, and reply set to the Broker's
Ingress Channel.
// 5. Find whether there is annotation with key "knative.dev/dependency".
// If not, mark Dependency to be succeeded, else figure out whether the
dependency is ready and mark Dependency correspondingly

if t.DeletionTimestamp != nil {
    // Everything is cleaned up by the garbage collector.
    return nil
}
// Tell tracker to reconcile this Trigger whenever the Broker changes.
brokerObjRef := corev1.ObjectReference{
    Kind:      brokerGVK.Kind,
    APIVersion: brokerGVK.GroupVersion().String(),
    Name:      t.Spec.Broker,
    Namespace: t.Namespace,
}

if err := r.tracker.Track(brokerObjRef, t); err != nil {
    logging.FromContext(ctx).Error("Unable to track changes to Broker",
zap.Error(err))
    return err
}

b, err := r.brokerLister.Brokers(t.Namespace).Get(t.Spec.Broker)
if err != nil {
    logging.FromContext(ctx).Error("Unable to get the Broker", zap.
Error(err))
    if apierrs.IsNotFound(err) {
        t.Status.MarkBrokerFailed("DoesNotExist", "Broker does not
exist")
        _, needDefaultBroker := t.GetAnnotations()[v1alpha1.
InjectionAnnotation]
        if t.Spec.Broker == "default" && needDefaultBroker {
            if e := r.labelNamespace(ctx, t); e != nil {
                logging.FromContext(ctx).Error("Unable to label the
namespace", zap.Error(e))
            }
        }
    } else {
        t.Status.MarkBrokerFailed("BrokerGetFailed", "Failed to get
broker")
    }
    return err
}

```

```

    }
    t.Status.PropagateBrokerStatus(&b.Status)

    brokerTrigger := b.Status.TriggerChannel
    if brokerTrigger == nil {
        logging.FromContext(ctx).Error("Broker TriggerChannel not populated")
        r.Recorder.Eventf(t, corev1.EventTypeWarning, triggerChannelFailed,
"Broker's Trigger channel not found")
        return errors.New("failed to find Broker's Trigger channel")
    }

    brokerIngress := b.Status.IngressChannel
    if brokerIngress == nil {
        logging.FromContext(ctx).Error("Broker IngressChannel not populated")
        r.Recorder.Eventf(t, corev1.EventTypeWarning, ingressChannelFailed,
"Broker's Ingress channel not found")
        return errors.New("failed to find Broker's Ingress channel")
    }

    // Get Broker filter service.
    filterSvc, err := r.getBrokerFilterService(ctx, b)
    .....

    subscriberURI, err := r.uriResolver.URIFromDestination(*t.Spec.
Subscriber, t)
    if err != nil {
        logging.FromContext(ctx).Error("Unable to get the Subscriber's URI",
zap.Error(err))
        return err
    }
    t.Status.SubscriberURI = subscriberURI

    sub, err := r.subscribeToBrokerChannel(ctx, t, brokerTrigger,
brokerIngress, filterSvc)
    if err != nil {
        logging.FromContext(ctx).Error("Unable to Subscribe", zap.Error(err))
        t.Status.MarkNotSubscribed("NotSubscribed", "%v", err)
        return err
    }
    t.Status.PropagateSubscriptionStatus(&sub.Status)

    if err := r.checkDependencyAnnotation(ctx, t); err != nil {
        return err
    }

    return nil
}

```


Trigger 示例:

```
apiVersion: eventing.knative.dev/v1alpha1
kind: Trigger
metadata:
  name: my-service-trigger
spec:
  broker: default
  filter:
    attributes:
      type: dev.knative.foo.bar
      myextension: my-extension-value
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1
      kind: Service
      name: my-service
```

总结

Broker/Trigger 模型出现的意义不仅在于其提供了消息过滤机制，更充分解耦了消息通道的实现，目前除了系统自身支持的基于内存的消息通道 InMemoryChannel 之外，还支持 Kafka、NATS Streaming 等消息服务。

此外结合 CloudEvent 进行事件统一标准传输，无论对于客户端接入事件源，还是消费端提供的消费事件服务，都能极大的提升了应用的跨平台可移植性。

事件注册机制 – Registry

背景

作为事件消费者，之前是无法事先知道哪些事件可以被消费，如果能通过某种方式获得哪些 Broker 提供哪些事件，那么事件消费者就能很方便通过这些 Broker 消费事件。Registry 就是在这样的背景下被提出的，通过 Registry 机制，消费者能针对特定的 Broker 的事件通过 Trigger 进行事件订阅消费。这里需要说明一下，Registry 设计与实现目前是针对 Broker/Trigger 事件处理模型。

诉求

- 每个 Registry 对应一个 Namespace 作为资源隔离的边界
- Registry 中包括事件类型列表，以提供事件发现机制，通过事件列表，我们可以决定对哪些 Ready 的事件进行消费
- 由于事件最终需要通过 Trigger 进行订阅，因此事件类型信息中需要包括创建 Trigger 所需要的信息。

实现

定义 EventType CRD 资源

定义 EventType 类型的 CRD 资源，示例 yaml:

```
apiVersion: eventing.knative.dev/v1alpha1
kind: EventType
metadata:
  name: com.github.pullrequest
  namespace: default
spec:
  type: com.github.pull_request
  source: github.com
```

```
schema: //github.com/schemas/pull_request
description: "GitHub pull request"
broker: default
```

- name: 设置时需要符合 k8s 命名规范
- type: 遵循 CloudEvent 类型设置。
- source: 遵循 CloudEvent source 设置。
- schema: 可以是 JSON schema, protobuf schema 等。可选项。
- description: 事件描述, 可选项。
- broker: 所提供 EventType 的 Broker。

事件注册与发现

1. 创建 EventType

一般我们通过以下两种方式创建 EventType 实现事件的注册。

1.1 通过 Event 事件源实例自动注册

基于事件源实例, 通过事件源控制器创建 EventType 进行注册:

```
apiVersion: sources.eventing.knative.dev/v1alpha1
kind: GitHubSource
metadata:
  name: github-source-sample
  namespace: default
spec:
  eventTypes:
    - push
    - pull_request
  ownerAndRepository: my-other-user/my-other-repo
  accessToken:
    secretKeyRef:
      name: github-secret
      key: accessToken
  secretToken:
    secretKeyRef:
      name: github-secret
      key: secretToken
  sink:
    apiVersion: eventing.knative.dev/v1alpha1
    kind: Broker
    name: default
```

通过运行上面的 yaml 信息，通过 GitHubSource 事件源控制器可以创建事件类型 `dev.knative.source.github.push` 以及事件类型 `dev.knative.source.github.pull_request` 这两个 EventType 进行注册，其中 source 为 `github.com` 以及名称为 `default` 的 Broker。具体如下：

```
apiVersion: eventing.knative.dev/v1alpha1
kind: EventType
metadata:
  generateName: dev.knative.source.github.push-
  namespace: default
  owner: # Owned by github-source-sample
spec:
  type: dev.knative.source.github.push
  source: github.com
  broker: default
---
apiVersion: eventing.knative.dev/v1alpha1
kind: EventType
metadata:
  generateName: dev.knative.source.github.pullrequest-
  namespace: default
  owner: # Owned by github-source-sample
spec:
  type: dev.knative.source.github.pull_request
  source: github.com
  broker: default
```

这里有两点需要注意：

- 通过自动生成默认名称，避免名称冲突。对于是否应该在代码（在本例中是 GithubSource 控制器）创建事件类型时生成，需要进一步讨论。
- 我们给 `spec.type` 加上了 `dev.knative.source.github.` 前缀，这个也需要进一步讨论确定是否合理。

1.2 手动注册

直接通过创建 EventType CR 资源实现注册，如：

```
apiVersion: eventing.knative.dev/v1alpha1
kind: EventType
```

```

metadata:
  name: org.bitbucket.repo:fork
  namespace: default
spec:
  type: org.bitbucket.repo:fork
  source: bitbucket.org
  broker: dev
  description: "BitBucket fork"

```

通过这种方式可以注册名称为 `org.bitbucket.repo:fork`, `type` 为 `org.bitbucket.repo:fork`, `source` 为 `bitbucket.org` 以及属于 `dev` Broker 的 `EventType`。

2. 获取 Registry 的事件注册

事件消费者可以通过如下方式获取 Registry 的事件注册列表：

```
$ kubectl get eventtypes -n default
```

NAME	TYPE
SOURCE SCHEMA	BROKER DESCRIPTION
READY REASON	
org.bitbucket.repo:fork	org.bitbucket.repo:fork
bitbucket.org	dev BitBucket fork
False BrokerIsNotReady	
com.github.pullrequest	com.github.pull_request
github.com //github.com/schemas/pull_request	default GitHub pull request
request True	
dev.knative.source.github.push-34cnb	dev.knative.source.github.
push github.com	default
True	
dev.knative.source.github.pullrequest-86jvh	dev.knative.source.github.
pull_request github.com	default
True	

3. Trigger 订阅事件

最后基于事件注册列表信息，事件消费者创建对应的 Trigger 对 Registry 中的 `EventType` 进行监听消费

Trigger 示例：

```

apiVersion: eventing.knative.dev/v1alpha1
kind: Trigger
metadata:

```

```
name: my-service-trigger
namespace: default
spec:
  filter:
    sourceAndType:
      type: dev.knative.source.github.push
      source: github.com
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1alpha1
      kind: Service
      name: my-service
```

总结

Registry 的设计主要是针对 Broker/Trigger 事件处理模型。如果没有发送事件到 Broker，就不会创建 EventType 注册到 Registry。在实现方面，我们可以检查 Event Source 的 Sink 类型是否是 Broker，如果是，则对其注册 EventType。

| Sequeue 解析

在实际的开发中我们会经常遇到将一条数据需要经过多次处理的场景，称为 Pipeline。那么在 Knative 中是否也提供这样的能力呢？其实从 Knative Eventing 0.7 版本开始，提供了 Sequence CRD 资源，用于事件处理 Pipeline。下面我们介绍一下 Sequence。

Sequence 定义

首先我们看一下 Sequence Spec 定义：

```
apiVersion: messaging.knative.dev/v1alpha1
kind: Sequence
metadata:
  name: test
spec:
  channelTemplate:
    apiVersion: messaging.knative.dev/v1alpha1
    kind: InMemoryChannel
  steps:
    - ref:
        apiVersion: serving.knative.dev/v1alpha1
        kind: Service
        name: test
  reply:
    kind: Broker
    apiVersion: eventing.knative.dev/v1alpha1
    name: test
```

Sequence Spec 包括 3 个部分：

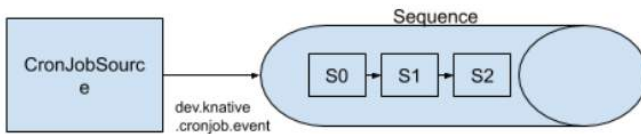
1. steps: 在 step 中定义了按照顺序执行的服务，每个服务会对应创建 Subscription
2. channelTemplate: 指定了使用具体的那个 Channel
3. reply: (可选) 定义了将最后一个 step 服务结果转发到的目标服务

Sequence 都是适合哪些具体应用场景呢？我们上面也提到了事件处理的 Pipeline。那么在实际场景应用中究竟以什么样的形式体现呢？现在我们揭晓一下 Sequence 在 Knative Eventing 中提供的如下 4 种使用场景：

- 直接访问 Service
- 面向事件处理
- 级联 Sequence
- 面向 Broker/Trigger

直接访问 Service 场景

事件源产生的事件直接发送给 Sequence 服务，Sequence 接收到事件之后顺序调用 Service 服务对事件进行处理：



创建 Knative Service

这里我们创建 3 个 Knative Service 用于事件处理。每个 Service 接收到事件之后会打印当前的事件处理信息。

```

apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: first
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/probable-summer:2656f39a7fcb6afd9fc79e7a4e215d14d651dc674f38020d1d18c6f04b220700
          env:
            - name: STEP
  
```



```

        value: "0"

---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: second
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/probable-summer:2656f39a7fcb6afd9fc79e7a4e215d14d651dc674f38020d1d18c6f04b220700
          env:
            - name: STEP
              value: "1"
---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: third
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/probable-summer:2656f39a7fcb6afd9fc79e7a4e215d14d651dc674f38020d1d18c6f04b220700
          env:
            - name: STEP
              value: "2"
---

```

创建 Sequence

创建顺序调用 first->second->third Service 的 Sequence。

```

apiVersion: messaging.knative.dev/v1alpha1
kind: Sequence
metadata:
  name: sequence
spec:
  channelTemplate:
    apiVersion: messaging.knative.dev/v1alpha1
    kind: InMemoryChannel
  steps:
    - ref:
        apiVersion: serving.knative.dev/v1alpha1

```

```

    kind: Service
    name: first
  - ref:
    apiVersion: serving.knative.dev/v1alpha1
    kind: Service
    name: second
  - ref:
    apiVersion: serving.knative.dev/v1alpha1
    kind: Service
    name: third

```

创建数据源

创建 CronJobSource 数据源，每隔 1 分钟发送一条事件消息 { “message”: “Hello world!” } 到 Sequence 服务。

```

apiVersion: sources.eventing.knative.dev/v1alpha1
kind: CronJobSource
metadata:
  name: cronjob-source
spec:
  schedule: "*/1 * * * *"
  data: '{"message": "Hello world!"}'
  sink:
    apiVersion: messaging.knative.dev/v1alpha1
    kind: Sequence
    name: sequence

```

示例结果

```

[root@iZ2ze7hmlpupjzuxllp5v8Z demo1]# kubectl get pods

```

NAME	READY	STATUS	RESTARTS	AGE
cronjobsource-cronjob-sour-61b02eea-c47c-11e9-8893-6a59193xsq69	1/1	Running	0	15m
first-n44mx-deployment-788ff8d9f6-rrtzj	2/2	Running	0	2m7s
second-2pvm5-deployment-b954b9c7-76d2z	2/2	Running	0	4m4s
third-wnl9n-deployment-fb4cfbc6c-bqcvm	2/2	Running	0	90s

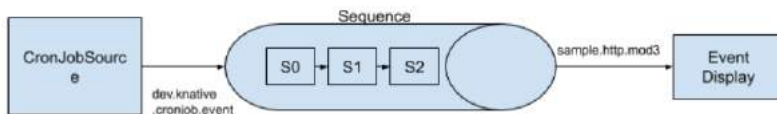
```
[root@iz2ze7hmlpupjzuxllp5v8Z demo1]# kubectl logs first-n44mx-deployment-788ff8d9f6-rrtzj user-container
Got Event Context: Context Attributes,
  specversion: 0.3
  type: dev.knative.cronjob.event
  source: /apis/v1/namespaces/default/cronjobsources/cronjob-source
  id: 31a7a955-a59d-45fa-ba69-bf5dcc3ce546
  time: 2019-08-22T01:44:00.000386589Z
  datacontenttype: application/json
Extensions,
  knativehistory: sequence-kn-sequence-0-kn-channel.default.svc.cluster.local
Got Data: &{Sequence:0 Message:Hello world!}
Got Transport Context: Transport Context,
```

```
[root@iz2ze7hmlpupjzuxllp5v8Z demo1]# kubectl logs second-2pvm5-deployment-b954b9c7-76d2z user-container
2019/08/22 01:44:37 http: superfluous response.WriteHeader call from github.com/vaikas-google/transformer/vendor/github.com:446)
Got Event Context: Context Attributes,
  cloudEventsVersion: 0.1
  eventType: samples.http.mod3
  source: /transformer/0
  eventID: 8d3a6807-b122-4d33-8adb-dbf5b242ba64
  eventTime: 2019-08-22T01:42:03.236259557Z
  contentType: application/json
Got Data: &{Sequence:0 Message:Hello world! - Handled by 0}
Got Transport Context: Transport Context,
```

```
[root@iz2ze7hmlpupjzuxllp5v8Z demo1]# kubectl logs third-wnl9n-deployment-fb4cfbc6c-bgcvm user-container
Got Event Context: Context Attributes,
  cloudEventsVersion: 0.1
  eventType: samples.http.mod3
  source: /transformer/1
  eventID: 31a7a955-a59d-45fa-ba69-bf5dcc3ce546
  eventTime: 2019-08-22T01:44:38.088712289Z
  contentType: application/json
Got Data: &{Sequence:0 Message:Hello world! - Handled by 0 - Handled by 1}
Got Transport Context: Transport Context,
```

面向事件处理场景

事件源产生的事件直接发送给 Sequence 服务，Sequence 接收到事件之后顺序调用 Service 服务对事件进行处理，处理之后的最终结果会调用 event-display Service 显示：



创建 Knative Service

同上创建 3 个 Knative Service 用于事件处理：

```

apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: first
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/probable-summer:2656f39a7fcb6afd9fc79e7a4e215d14d651dc674f38020d1d18c6f04b220700
          env:
            - name: STEP
              value: "0"
---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: second
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/probable-summer:2656f39a7fcb6afd9fc79e7a4e215d14d651dc674f38020d1d18c6f04b220700
          env:
            - name: STEP
              value: "1"
---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: third
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/probable-summer:2656f39a7fcb6afd9fc79e7a4e215d14d651dc674f38020d1d18c6f04b220700
          env:
            - name: STEP
              value: "2"
---

```

创建 Sequence

创建顺序调用 first->second->third Service 的 Sequence，将处理结果通过 reply 发送给 event-display。

```

apiVersion: messaging.knative.dev/v1alpha1
kind: Sequence
metadata:
  name: sequence
spec:
  channelTemplate:
    apiVersion: messaging.knative.dev/v1alpha1
    kind: InMemoryChannel
  steps:
    - ref:
        apiVersion: serving.knative.dev/v1alpha1
        kind: Service
        name: first
    - ref:
        apiVersion: serving.knative.dev/v1alpha1
        kind: Service
        name: second
    - ref:
        apiVersion: serving.knative.dev/v1alpha1
        kind: Service
        name: third
  reply:
    kind: Service
    apiVersion: serving.knative.dev/v1alpha1
    name: event-display

```

创建结果显示 Service

创建 event-display Service, 用于接收最终的结果信息。

```

apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: event-display
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-release/event_
          display: bf45b3eb1e7fc4cb63d6a5a6416cf696295484a7662e0cf9ccdf5c080542c21d

```

创建数据源

创建 CronJobSource 数据源, 每隔 1 分钟发送一条事件消息 { “message”:

“Hello world!”} 到 Sequence 服务。

```
apiVersion: sources.eventing.knative.dev/v1alpha1
kind: CronJobSource
metadata:
  name: cronjob-source
spec:
  schedule: "*/1 * * * *"
  data: '{"message": "Hello world!"}'
  sink:
    apiVersion: messaging.knative.dev/v1alpha1
    kind: Sequence
    name: sequence
```

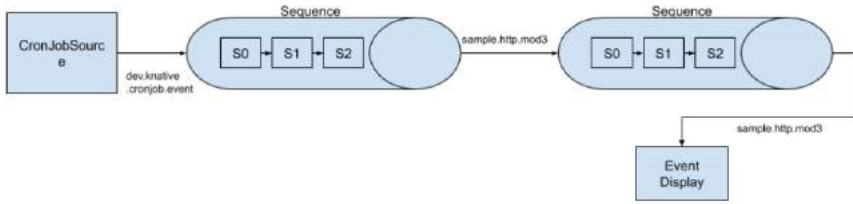
示例结果

```
[root@iZ2ze7hmlpupjzuxllp5v8Z demo2]# kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
cronjobsource-cronjob-sour-61b02eea-c47c-11e9-8893-6a59193xsq69    1/1     Running   0           44m
event-display-j9rzc-deployment-7645948d87-bmck7                    2/2     Running   0           10s
first-n44mx-deployment-788ff8d9f6-l74kk                            2/2     Running   0           51s
second-2pvm5-deployment-b954b9c7-k9r22                            2/2     Running   0           2m48s
third-wm19n-deployment-fb4c4bc6c-w2j4m                            2/2     Running   0           14s
```

```
[root@iZ2ze7hmlpupjzuxllp5v8Z demo2]# kubectl logs event-display-j9rzc-deployment-7645948d87-bmck7 user-container
- CloudEvent: valid ☐
Context Attributes,
  CloudEventsVersion: 0.1
  EventType: samples.http.mod3
  Source: /transformer/2
  EventID: 857a5ee0-9834-4a4b-90bb-4065377157b7
  EventTime: 2019-08-22T02:14:41.105177262Z
  ContentType: application/json
Transport Context,
  URI: /
  Host: event-display.default.svc.cluster.local
  Method: POST
Data,
{
  "id": 0,
  "message": "Hello world! - Handled by 0 - Handled by 1 - Handled by 2"
}
```

级联 Sequence 场景

Sequence 更高级的地方还在于支持级联处理：Sequence By Sequence，这可以进行多次 Sequence 处理，满足复杂事件处理场景需求：



创建 Knative Service

创建 6 个 Knative Service 用于事件处理，前 3 个用于第 1 个 Sequence，后 3 个用于第 2 个 Sequence。

```

apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: first
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/probable-sunmer:2656f39a7fcb6afd9fc79e7a4e215d14d651dc674f38020d1d18c6f04b220700
          env:
            - name: STEP
              value: "0"

---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: second
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/probable-sunmer:2656f39a7fcb6afd9fc79e7a4e215d14d651dc674f38020d1d18c6f04b220700
          env:
            - name: STEP
              value: "1"

---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:

```

```

    name: third
  spec:
    template:
      spec:
        containers:
          - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/probable-su
mmmer:2656f39a7fcb6afd9fc79e7a4e215d14d651dc674f38020d1d18c6f04b220700
            env:
              - name: STEP
                value: "2"
---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: fourth
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/probable-su
mmmer:2656f39a7fcb6afd9fc79e7a4e215d14d651dc674f38020d1d18c6f04b220700
            env:
              - name: STEP
                value: "3"
---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: fifth
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/probable-su
mmmer:2656f39a7fcb6afd9fc79e7a4e215d14d651dc674f38020d1d18c6f04b220700
            env:
              - name: STEP
                value: "4"
---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: sixth
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/probable-su

```



```

mmer:2656f39a7fcb6afd9fc79e7a4e215d14d651dc674f38020d1d18c6f04b220700
  env:
    - name: STEP
      value: "5"
---
```

创建第 1 个 Sequence

使用 first→second→third Service 用于第 1 个 Sequence 调用处理，将执行结果发送给第 2 个 Sequence。

```

apiVersion: messaging.knative.dev/v1alpha1
kind: Sequence
metadata:
  name: first-sequence
spec:
  channelTemplate:
    apiVersion: messaging.knative.dev/v1alpha1
    kind: InMemoryChannel
  steps:
    - ref:
        apiVersion: serving.knative.dev/v1alpha1
        kind: Service
        name: first
    - ref:
        apiVersion: serving.knative.dev/v1alpha1
        kind: Service
        name: second
    - ref:
        apiVersion: serving.knative.dev/v1alpha1
        kind: Service
        name: third
  reply:
    kind: Sequence
    apiVersion: messaging.knative.dev/v1alpha1
    name: second-sequence
```

创建第 2 个 Sequence

使用 fourth→fifth→sixth Service 用于第 2 个 Sequence 调用处理，将执行结果发送给 event-display。

```

apiVersion: messaging.knative.dev/v1alpha1
```

```

kind: Sequence
metadata:
  name: second-sequence
spec:
  channelTemplate:
    apiVersion: messaging.knative.dev/v1alpha1
    kind: InMemoryChannel
  steps:
    - ref:
        apiVersion: serving.knative.dev/v1alpha1
        kind: Service
        name: fourth
    - ref:
        apiVersion: serving.knative.dev/v1alpha1
        kind: Service
        name: fifth
    - ref:
        apiVersion: serving.knative.dev/v1alpha1
        kind: Service
        name: sixth
  reply:
    kind: Service
    apiVersion: serving.knative.dev/v1alpha1
    name: event-display

```

创建结果显示 Service

```

apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: event-display
spec:
  template:
    spec:
      containerizers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-release/event_
display:bf45b3eb1e7fc4cb63d6a5a6416cf696295484a7662e0cf9ccdf5c080542c21d

```

创建数据源指向第 1 个 Sequence

```

apiVersion: sources.eventing.knative.dev/v1alpha1
kind: CronJobSource
metadata:
  name: cronjob-source
spec:

```

```

schedule: "*/1 * * * *"
data: '{"message": "Hello world!"}'
sink:
  apiVersion: messaging.knative.dev/v1alpha1
  kind: Sequence
  name: first-sequence

```

示例结果

```

[root@i22ze7hmlpupjzuxllp5v8z demo3]# kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
cronjobsource-cronjob-sour-9fd14416-c484-11e9-8ada-3e0b0afqrgvm  1/1     Running   0           78s
event-display-j9rzc-deployment-7645948d87-nkx7k                 2/2     Running   0           18s
fifth-q2pbj-deployment-654856668d-jgw2m                        2/2     Running   0           19s
first-n44mx-deployment-788ff8d9f6-9wtgf                         2/2     Running   0           28s
fourth-p8jr4-deployment-fc855fbfc-jdrx4                       2/2     Running   0           21s
second-2pvm5-deployment-b954b9c7-8m8jd                         2/2     Running   0           2m25s
sixth-pdds4-deployment-77b5dcc77f-7mg5p                       2/2     Running   0           16s
third-wml9n-deployment-fb4cfbc6c-9xcbd                         2/2     Running   0           24s

```

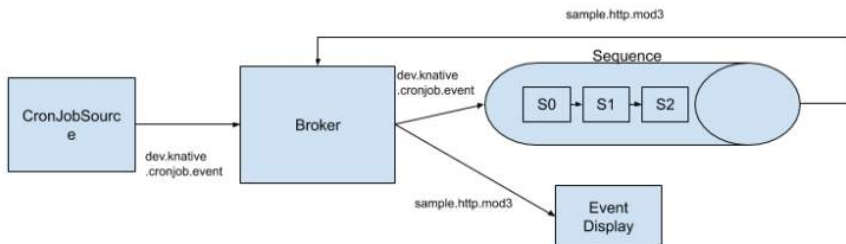
```

[root@i22ze7hmlpupjzuxllp5v8z demo3]# kubectl logs event-display-j9rzc-deployment-7645948d87-nkx7k user-container
- CloudEvent: valid {}
Context Attributes,
  CloudEventsVersion: 0.1
  EventType: samples.http.mod3
  Source: /transformer/5
  EventID: f5cc37de-d3e9-4942-824b-29fe41cb42d9
  EventTime: 2019-08-22T02:30:18.626429389Z
  ContentType: application/json
Transport Context,
  URI: /
  Host: event-display.default.svc.cluster.local
  Method: POST
Data,
{
  "id": 0,
  "message": "Hello world! - Handled by 0 - Handled by 1 - Handled by 2 - Handled by 3 - Handled by 4 - Handled by 5"
}

```

Broker/Trigger 场景

事件源 cronjobsource 向 Broker 发送事件，通过 Trigger 将这些事件发送到由 3 个 Service 调用的 Sequence 中。Sequence 处理完之后将结果事件发送给 Broker，并最终由另一个 Trigger 发送给 event-display Service 显示事件结果：



创建 Knative Service

同上创建 3 个 Knative Service，用于 Sequence 中服务处理。

```

apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: first
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/probable-summer:2656f39a7fcb6afd9fc79e7a4e215d14d651dc674f38020d1d18c6f04b220700
          env:
            - name: STEP
              value: "0"

---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: second
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/probable-summer:2656f39a7fcb6afd9fc79e7a4e215d14d651dc674f38020d1d18c6f04b220700
          env:
            - name: STEP
              value: "1"

---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: third
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/probable-summer:2656f39a7fcb6afd9fc79e7a4e215d14d651dc674f38020d1d18c6f04b220700
          env:
            - name: STEP
              value: "2"

---
```

创建 Sequence

创建 Sequence，这里依次顺序执行 first→second→third 这 3 个服务。将最终处理的结果发送到 broker-test 中。

```
apiVersion: messaging.knative.dev/v1alpha1
kind: Sequence
metadata:
  name: sequence
spec:
  channelTemplate:
    apiVersion: messaging.knative.dev/v1alpha1
    kind: InMemoryChannel
  steps:
    - ref:
        apiVersion: serving.knative.dev/v1alpha1
        kind: Service
        name: first
    - ref:
        apiVersion: serving.knative.dev/v1alpha1
        kind: Service
        name: second
    - ref:
        apiVersion: serving.knative.dev/v1alpha1
        kind: Service
        name: third
  reply:
    kind: Broker
    apiVersion: eventing.knative.dev/v1alpha1
    name: default
```

创建事件源指向 Broker

创建 CronjobSource，它将每隔 1 分钟发送一条 { "message": "Hello world!" } 消息到 broker-test 中。

```
apiVersion: sources.eventing.knative.dev/v1alpha1
kind: CronJobSource
metadata:
  name: cronjob-source
spec:
  schedule: "*/1 * * * *"
  data: '{"message": "Hello world!"}'
  sink:
    apiVersion: eventing.knative.dev/v1alpha1
```

```
kind: Broker
name: default
```

创建 Broker

创建默认 Broker

```
kubectl label namespace default knative-eventing-injection=enabled
```

创建 Trigger 指向 Sequence

创建订阅事件类型为 `dev.knative.cronjob.event` 的 Trigger, 用于 Sequence 进行消费处理。

```
apiVersion: eventing.knative.dev/v1alpha1
kind: Trigger
metadata:
  name: sequence-trigger
spec:
  filter:
    sourceAndType:
      type: dev.knative.cronjob.event
  subscriber:
    ref:
      apiVersion: messaging.knative.dev/v1alpha1
      kind: Sequence
      name: sequence
```

创建结果订阅 Trigger

创建订阅 `samples.http.mod3` 的事件类型 Trigger, 将 Sequence 执行的结果发送给 `event-display Service` 进行显示。

```
apiVersion: eventing.knative.dev/v1alpha1
kind: Trigger
metadata:
  name: display-trigger
spec:
  filter:
    sourceAndType:
      type: samples.http.mod3
  subscriber:
```

```

ref:
  apiVersion: serving.knative.dev/v1alpha1
  kind: Service
  name: event-display
---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: event-display
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-release/event_
          display:bf45b3eb1e7fc4cb63d6a5a6416cf696295484a7662e0cf9ccdf5c080542c21d
---

```

示例结果

```

[root@iZ2ze7hmlpupjzuxllp5v8Z demo4]# kubectl get pod
NAME                                READY   STATUS    RESTARTS   AGE
cronjobsource-cronjob-sour-f25a7f48-c487-11e9-8ada-3e0b0afpc2sw  1/1     Running   0           10m
default-broker-filter-9c5dbdfbd-2w9r4  1/1     Running   0           13m
default-broker-ingress-7b6765c54d-qq794  1/1     Running   0           13m
event-display-j9rzc-deployment-7645948d87-r4rdk  2/2     Running   0           76s
first-zsth-deployment-549b5d7866-gtrnk  2/2     Running   0           6m
second-m9kmr-deployment-7bfff5498b8-4fjmt  2/2     Running   0           82s
third-69qv8-deployment-698db6f666-jfgbm  2/2     Running   0           79s

```

```

[root@iZ2ze7hmlpupjzuxllp5v8Z demo4]# kubectl logs event-display-j9rzc-deployment-7645948d87-r4rdk user-container
- CloudEvent: valid ☐
Context Attributes,
  SpecVersion: 0.3
  Type: samples.http.mod3
  Source: /transformer/2
  ID: 6c6dc096-ce6d-427f-b2c7-e075e26533fe
  Time: 2019-08-22T03:14:44.280781273Z
  DataContentType: application/json
Extensions:
  kn00timeinflight: 2019-08-22T03:14:44.293301239Z
Transport Context,
  URI: /
  Host: event-display.default.svc.cluster.local
  Method: POST
Data,
{
  "id": 0,
  "message": "Hello world! - Handled by 0 - Handled by 1 - Handled by 2"
}

```

总结

以上介绍了什么是 Sequence，以及基于 Sequence 的 4 种使用场景，我们可以根据实际需求选择不同的使用场景，从而实现事件处理 Pipeline。这对于需要多步骤处理事件的场景尤为适合。

Parallel 解析

从 Knative Eventing 0.8 开始，支持根据不同的过滤条件对事件进行选择处理。通过 Parallel 提供了这样的能力。本文就给大家介绍一下这个特性。

资源定义

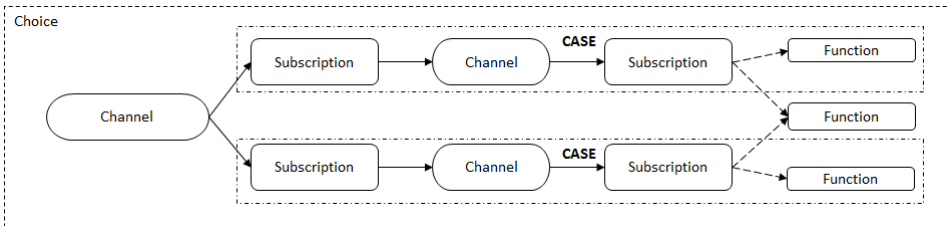
我们先看一下 Parallel 资源定义，典型的 Parallel Spec 描述如下：

```
apiVersion: messaging.knative.dev/v1alpha1
kind: Parallel
metadata:
  name: me-odd-even-parallel
spec:
  channelTemplate:
    apiVersion: messaging.knative.dev/v1alpha1
    kind: InMemoryChannel
  cases:
    - filter:
        uri: "http://me-even-odd-switcher.default.svc.cluster.local/0"
      subscriber:
        ref:
          apiVersion: serving.knative.dev/v1alpha1
          kind: Service
          name: me-even-transformer
    - filter:
        uri: "http://me-even-odd-switcher.default.svc.cluster.local/1"
      subscriber:
        ref:
          apiVersion: serving.knative.dev/v1alpha1
          kind: Service
          name: me-odd-transformer
  reply:
    apiVersion: serving.knative.dev/v1alpha1
    kind: Service
    name: me-event-display
```

主要包括如下 3 部分：

- cases 定义了一系列 filter 和 subscriber。对于每个条件分支：
 - 首先判断 filter，当返回事件时，调用 subscriber。filter 和 subscriber 要求都是可访问的。
 - subscriber 执行返回的事件会发生到 reply。如果 reply 为空，则发送到 spec.reply
- channelTemplate 定义了当前 Parallel 中使用的 Channel 类型
- reply 定义了全局响应的目标函数。

逻辑架构如图所示：



代码实现

关键代码实现如下：

1. 首先为 Parallel 创建一个全局的 Channel。然后为每一个 case 创建一个过滤 Channel
2. 在每个 case 中做了如下处理：
 - 为全局的 Channel 创建一个 Subscription，订阅条件为 filter 信息，并且把 reply 响应发送给当前 case 中的过滤 Channel
 - 为过滤 Channel 创建一个 Subscription，将订阅信息发送给每个 case 中的 Reply。如果当前 case 中没有设置 Reply，则发送的全局 Reply。

```
func (r *Reconciler) reconcile(ctx context.Context, p *v1alpha1.Parallel)
error {
    p.Status.InitializeConditions()
```

```

    // Reconciling parallel is pretty straightforward, it does the following
    things:
    // 1. Create a channel fronting the whole parallel and one filter channel
    per branch.
    // 2. For each of the Branches:
    //     2.1 create a Subscription to the fronting Channel, subscribe the
    filter and send reply to the filter Channel
    //     2.2 create a Subscription to the filter Channel, subscribe the
    subscriber and send reply to
    //         either the branch Reply. If not present, send reply to the
    global Reply. If not present, do not send reply.
    // 3. Rinse and repeat step #2 above for each branch in the list
    if p.DeletionTimestamp != nil {
        // Everything is cleaned up by the garbage collector.
        return nil
    }

    channelResourceInterface := r.DynamicClientSet.Resource(duckroot.
    KindToResource(p.Spec.ChannelTemplate.GetObjectKind().GroupVersionKind())).
    Namespace(p.Namespace)

    if channelResourceInterface == nil {
        msg := fmt.Sprintf("Unable to create dynamic client for: %v",
        p.Spec.ChannelTemplate)
        logging.FromContext(ctx).Error(msg)
        return errors.New(msg)
    }

    // Tell tracker to reconcile this Parallel whenever my channels change.
    track := r.resourceTracker.TrackInNamespace(p)

    var ingressChannel *duckv1alpha1.Channelable
    channels := make([]*duckv1alpha1.Channelable, 0, len(p.Spec.Branches))
    for i := -1; i < len(p.Spec.Branches); i++ {
        var channelName string
        if i == -1 {
            channelName = resources.ParallelChannelName(p.Name)
        } else {
            channelName = resources.ParallelBranchChannelName(p.Name, i)
        }

        c, err := r.reconcileChannel(ctx, channelName,
        channelResourceInterface, p)
        if err != nil {
            logging.FromContext(ctx).Error(fmt.Sprintf("Failed to reconcile
            Channel Object: %s/%s", p.Namespace, channelName), zap.Error(err))
            return err
        }
    }

```

```

        // Convert to Channel duck so that we can treat all Channels the
        same.
        channelable := &duckv1alpha1.Channelable{}
        err = duckapis.FromUnstructured(c, channelable)
        if err != nil {
            logging.FromContext(ctx).Error(fmt.Sprintf("Failed to convert to
            Channelable Object: %s/%s", p.Namespace, channelName), zap.Error(err))
            return err
        }
        // Track channels and enqueue parallel when they change.
        if err = track(utils.ObjectRef(channelable, channelable.
        GroupVersionKind())); err != nil {
            logging.FromContext(ctx).Error("Unable to track changes to
            Channel", zap.Error(err))
            return err
        }
        logging.FromContext(ctx).Info(fmt.Sprintf("Reconciled Channel Object:
        %s/%s %v", p.Namespace, channelName, c))

        if i == -1 {
            ingressChannel = channelable
        } else {
            channels = append(channels, channelable)
        }
    }
    p.Status.PropagateChannelStatuses(ingressChannel, channels)

    filterSubs := make([]*v1alpha1.Subscription, 0, len(p.Spec.Branches))
    subs := make([]*v1alpha1.Subscription, 0, len(p.Spec.Branches))
    for i := 0; i < len(p.Spec.Branches); i++ {
        filterSub, sub, err := r.reconcileBranch(ctx, i, p)
        if err != nil {
            return fmt.Errorf("Failed to reconcile Subscription Objects for
            branch: %d : %s", i, err)
        }
        subs = append(subs, sub)
        filterSubs = append(filterSubs, filterSub)
        logging.FromContext(ctx).Debug(fmt.Sprintf("Reconciled Subscription
        Objects for branch: %d: %v, %v", i, filterSub, sub))
    }
    p.Status.PropagateSubscriptionStatuses(filterSubs, subs)

    return nil
}

```

示例演示

接下来让我们通过一个实例具体了解一下 Parallel。通过 CronJobSource 产生事件发送给 me-odd-even-parallel Parallel，Parallel 会将事件发送给每个 case，Case 中通过 filter 不同的参数访问 me-even-odd-switcher 服务，me-even-odd-switcher 服务会根据当前事件的创建时间随机计算 0 或 1 的值，如果计算值和请求参数值相匹配，则返回事件，否则不返回事件。

- 若 `http://me-even-odd-switcher.default.svc.cluster.local/0` 匹配成功，返回事件到 me-even-transformer 服务进行处理
- 若 `http://me-even-odd-switcher.default.svc.cluster.local/1` 匹配成功，返回事件到 odd-transformer 服务进行处理

不管哪个 case 处理完之后，将最终的事件发送给 me-event-display 服务进行事件显示。

具体操作步骤如下：

创建 Knative Service

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: me-even-odd-switcher
spec:
  template:
    spec:
      containers:
        - image: villardl/switcher-nodejs:0.1
          env:
            - name: EXPRESSION
              value: Math.round(Date.parse(event.time) / 60000) % 2
            - name: CASES
              value: '[0, 1]'
---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: even-transformer
```

```
spec:
  template:
    spec:
      containers:
      - image: villardl/transformer-nodejs:0.1
        env:
        - name: TRANSFORMER
          value: |
            ({"message": "we are even!"})

---
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: odd-transformer
spec:
  template:
    spec:
      containers:
      - image: villardl/transformer-nodejs:0.1
        env:
        - name: TRANSFORMER
          value: |
            ({"message": "this is odd!"})
.
```

创建 Parallel

```
apiVersion: messaging.knative.dev/v1alpha1
kind: Parallel
metadata:
  name: me-odd-even-parallel
spec:
  channelTemplate:
    apiVersion: messaging.knative.dev/v1alpha1
    kind: InMemoryChannel
  cases:
  - filter:
      uri: "http://me-even-odd-switcher.default.svc.cluster.local/0"
    subscriber:
      ref:
        apiVersion: serving.knative.dev/v1alpha1
        kind: Service
        name: me-even-transformer
  - filter:
      uri: "http://me-even-odd-switcher.default.svc.cluster.local/1"
    subscriber:
```

```

      ref:
        apiVersion: serving.knative.dev/v1alpha1
        kind: Service
        name: me-odd-transformer
    reply:
      apiVersion: serving.knative.dev/v1alpha1
      kind: Service
      name: me-event-display

```

创建 CronJobSource 数据源

```

apiVersion: sources.eventing.knative.dev/v1alpha1
kind: CronJobSource
metadata:
  name: me-cronjob-source
spec:
  schedule: "*/1 * * * *"
  data: '{"message": "Even or odd?"}'
  sink:
    apiVersion: messaging.knative.dev/v1alpha1
    kind: Parallel
    name: me-odd-even-parallel

```

查看结果

运行之后可以看到类似如下结果：

```

kubectl logs -l serving.knative.dev/service=me-event-display --tail=30 -c
user-container

* cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 0.3
  type: dev.knative.cronjob.event
  source: /apis/v1/namespaces/default/cronjobsources/me-cronjob-source
  id: 48eea348-8cfd-4aba-9ead-cb024ce16a48
  time: 2019-07-31T20:56:00.000477587Z
  datacontenttype: application/json; charset=utf-8
Extensions,
  knativehistory: me-odd-even-parallel-kn-parallel-kn-channel.default.svc.
cluster.local, me-odd-even-parallel-kn-parallel-0-kn-channel.default.svc.
cluster.local
Data,
{

```

```

    "message": "we are even!"
  }
* cloudevents.Event
Validation: valid
Context Attributes,
  specversion: 0.3
  type: dev.knative.cronjob.event
  source: /apis/v1/namespaces/default/cronjobsources/me-cronjob-source
  id: 42717dcf-b194-4b36-a094-3ea20e565ad5
  time: 2019-07-31T20:57:00.000312243Z
  datacontenttype: application/json; charset=utf-8
Extensions,
  knativehistory: me-odd-even-parallel-kn-parallel-1-kn-channel.default.
svc.cluster.local, me-odd-even-parallel-kn-parallel-kn-channel.default.svc.
cluster.local
Data,
  {
    "message": "this is odd!"
  }

```

结论

通过上面的介绍，相信大家对 Parallel 如何进行事件条件处理有了更多的了解，对于并行处理事件的场景下，不妨试试 Parallel。

云原生开发实战

日志和监报告警

阿里云日志服务 (Log Service, 简称 LOG) 是针对日志类数据的一站式服务。您无需开发就能快速完成日志数据采集、消费、投递、查询分析以及监报告警等功能。在 Knative 中结合日志服务, 能有效提升对 Serverless 应用的运维能力。

前提条件

- 您已经成功创建一个 Kubernetes 集群, 参见[创建 Kubernetes 集群](#)
- 部署[日志服务](#)
- 部署成功 Knative, 参见[部署 Knative](#)
- 部署成功 Serving 组件

日志

1. 部署 Knative Service 服务。参考[部署 Serving Hello World 应用示例](#)
2. 选择[日志库](#), 创建 Logstore。这里以创建 helloworld 为例:

Logstore列表						学习路径	查看Logstore	创建
请输入Logstore名称进行模糊查询						搜索		
Logstore名称	数据输入来源	监控	日志采集模式	日志消费模式			操作	
				日志消费	日志投递	日志分析		
audit-c908f4dcf7cab0f5658c68c67e3e23a16d	🔍	🔍	Logtail配置 (帮助) 诊断 更多 +	预览 更多 +	MaxCompute OSS	查询	修改 删除	
config operation log	🔍	🔍	Logtail配置 (帮助) 诊断 更多 +	预览 更多 +	MaxCompute OSS	查询	修改 删除	
helloworld	🔍	🔍	Logtail配置 (帮助) 诊断 更多 +	预览 更多 +	MaxCompute OSS	查询	修改 删除	

共3页, 当前显示: 10 / 页

3. 数据源接入，选择 Docker 标准输出，参见[日志服务容器标准输出](#)

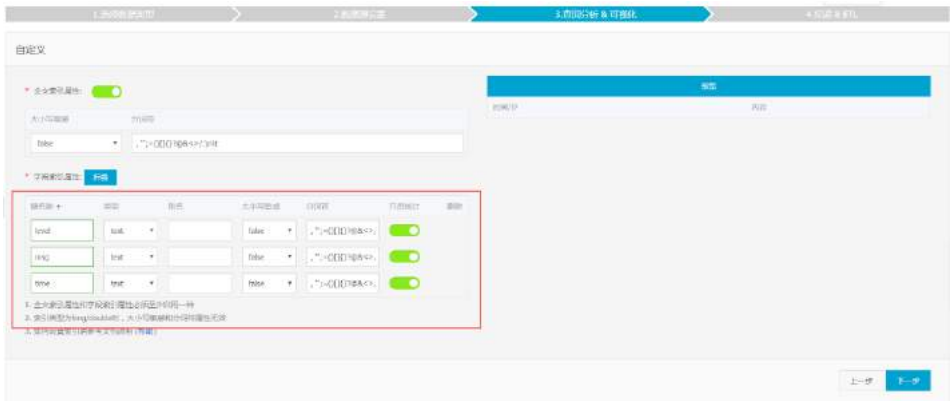


4. 插件配置这里我们针对 helloworld-go Service，设置采集的环境变量为：“K_SERVICE”：“helloworld-go”。并且通过 processors 分割日志信息，如这里”Keys”:[“time”,“level”,“msg”]。

```
{
  "inputs": [
    {
      "detail": {
        "IncludeEnv": {
          "K_SERVICE": "helloworld-go"
        },
        "IncludeLabel": {},
        "ExcludeLabel": {}
      },
      "type": "service_docker_stdout"
    }
  ],
  "processors": [
    {
      "detail": {
        "KeepSource": false,
        "NoMatchError": true,
        "Keys": [
          "time",
          "level",
          "msg"
        ]
      }
    }
  ]
}
```

```
    "msg"
  ],
  "NoKeyError": true,
  "Regex": "(\\d+-\\d+-\\d+\\s+\\d+:\\d+:\\d+)\\s+(\\w+)\\s+(.*)",
  "SourceKey": "content"
},
"type": "processor_regex"
}
]
```

4. 开启全文索引，设置查询展示列



5. 访问 Hello World 示例服务。

```
$ curl -H "Host: helloworld-go.default.example.com" http://112.124.XX.XX
Hello Go Sample v1!
```

6. 登录[日志服务控制台](#)，进入对应的 Project，选择 helloworld Logstore，点击[查询](#)


```
* | select 'ERROR' , count(1) as total group by 'ERROR'
```

点击【查询 / 分析】，结果如图所示：



2. 告警设置。点击【另存为告警】。



3. 设置告警名称、添加到仪表盘（这里可以新建，输入名称即可）等。其中告警触发条件输入判断告警是否触发的条件表达式，可以参考[告警条件表达式语法](#)。我们这里设置“查询区间：1 分钟，执行间隔：1 分组，触发条件：total > 3”表示间隔 1 分钟检查，如果 1 分钟内出现 3 次 ERROR 信息，则触发告警。

创建告警



告警配置

通知

* 告警名称

hello

5/64

* 添加到仪表盘

新建

helloworld

10/64

* 图表名称

hello

5/64

查询语句

* | select 'ERROR' , count(1) as total group by 'ERROR'

* 查询区间

1分钟 (相对)

* 检查频率

固定间隔

1

分钟

* 触发条件

total>3

支持加(+)减(-)乘(*)除(/)取模(%)运算和>,>=,<,<=,=,!=,=~比较运算。[帮助文档](#)

高级选项

下一步

取消

4. 告警通知

当前支持如图所示告警通知：

创建告警

告警配置

通知

通知列表

邮件 ×

✕ 邮件

* 收件人

a@a.com7/256

多个收件人请用逗号(,)分隔

主题

日志服务告警6/128

* 发送内容

error test

支持使用模版变量：\${Project}, \${Condition}, \${AlertName}, \${AlertID}, \${Dashboard}, \${FireTime}, \${Results} [查看全部变量](#)

上一步

提交

取消

5. 访问 Hello World 示例服务。执行多次以下命令，就会触发告警通知

```
$ curl -H "Host: helloworld-go.default.example.com" http://112.124.XX.XX
Hello Go Sample v1!
```

如果是设置的邮件通知，告警信息如下图所示：

尊敬的日志服务用户，您好：

您在[日志服务](#)配置的告警规则 **hello** 已经触发，请及时处理。触发详情如下：

所属项目	k8s-log-c83121c1311314426a7da633caf6bf411
触发时间	2019-11-07 11:01:13
触发条件	[11] > 3
告警日志	[_col0:ERROR,total:11]
仪表盘	dashboard-1573094687-489620
详情	点此查看详情
自定义通知内容	error test
告警历史	点此查看告警历史

总结

通过上面的介绍相信你已经了解如何在 Knative 中使用日志服务收集 Serverless 应用容器日志，并进行告警设置。在 Knative 中采用日志服务收集、分析业务日志，并结合设置监控告警，满足了生产级别的 Serverless 应用运维的诉求。

调用链管理

阿里云链路追踪 Tracing Analysis 为分布式应用的开发者提供了完整的调用链路还原、调用请求量统计、链路拓扑、应用依赖分析等工具。本文介绍了如何在 Knative 上实现 Tracing 分布式追踪,以帮助开发者快速分析和诊断 Knative 中部署的应用服务。

前提条件

- 您已经成功创建一个 Kubernetes 集群, 参见[创建 Kubernetes 集群](#)
- 部署 Istio, 参见[部署 Istio](#), 这里需要注意以下几点:
- Pilot 设置跟踪采样百分比, 推荐设置大一些 (如: 100), 便于数据采样



- 启用[阿里云链路追踪服务](#), 查看 token 点击 on, 客户端采集工具选择 **zipkin** (目前仅支持使用 zipkin 的 /api/v1/spans 接口访问), 选择集群所在 Region 的接入点 (推荐使用内网接入地址), 以华南 1 Region 为例如下:

您已开通链路追踪, 请按以下步骤进行监控接入

1 开通相关服务

开通日志服务已开通

开通访问控制RAM已开通

2 授权链路追踪读与您的日志服务已授权

3 接入链路追踪监控

下载SDK并查看接入指南

Java Go C++ Python PHP .Net Nodejs

Region对应信息 查看token on

客户端采集工具: jaeger zipkin skywalking

region	相关信息
华南1(深圳)	公网接入点: http://tracing-analysis-dc-sz.aliyuncs.com/adapt_a92srsbtktl@xxxxxxxx_xxxfd1@53df7ad2afxxxx/api/v1/spans 公网接入点: http://tracing-analysis-dc-sz.aliyuncs.com/adapt_a92srsbtktl@xxxxxxxx_xx@xxxxx/api/v2/spans 内网接入点: http://tracing-analysis-dc-sz-internal.aliyuncs.com/adapt_a92srsxxx@xxxxfss_xxx@xxx/api/v1/spans 内网接入点: http://tracing-analysis-dc-sz-internal.aliyuncs.com/adapt_a92srsbtktl@dfsfsfb_xxd1@53df7ad2sfsfssfs/api/v2/spans

- 选择启用链路追踪。设置链路追踪服务地址

链路追踪设置

不启用

启用分布式追踪 Jaeger (需要开通阿里云日志服务, 立即开通)

启用链路追踪 (需要开通阿里云链路追踪服务, 立即开通)

将自动创建名称为 istio-tracing-{ClusterID} 的 Project

* 接入点地址 http://tracing-analysis-dc-sz-internal.aliyuncs.com/adapt_a92srsbtktl@6580e31a949f4eb_a'

- 部署成功 Knative , 参见[部署 Knative](#)
- 部署成功 Serving 组件

调用链追踪

- 选择命名空间设置如下标签启用 Sidecar 自动注入: istio-injection=enabled。通过这种方式就注入了 Istio 的 envoy 代理(proxy)容器, Istio 的

envoy 代理拦截流量后会主动上报 trace 系统。以设置 default 命名空间为例：

```
kubectl label namespace default istio-injection=enabled
```

- 2. 部署 Knative Service 服务。参考 Serving Hello World 章节
- 3. 访问 Hello World 示例服务。

```
$ curl -H "Host: helloworld-go.default.example.com" http://112.124.XX.XX
Hello Go Sample v1!
```

4. 登录[阿里云链路追踪服务控制台](#)，选择应用列表，可以查看对应应用的 tracing 信息。



5. 选择应用，点击查看应用详情，可以看到服务调用的平均响应时间。



总结

在 Knative 中，推荐开启阿里云链路追踪服务。通过 Tracing 不仅有助于复杂应用服务之间的问题排查及效率优化，同时也是 Knative 的最佳实践方式。

使用 GitHub 事件源

前提条件

- 您已经成功创建一个 Kubernetes 集群
- 部署成功 Knative
- 部署成功 Serving 组件，并且完成域名配置。
- 部署成功 Eventing 组件

操作步骤

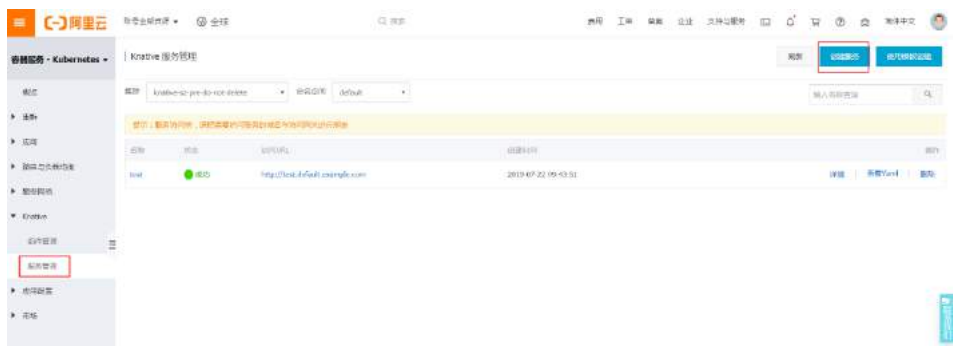
部署 GitHub 事件源。

1. 登录[容器服务管理控制台](#)。
2. 在 Kubernetes 菜单下，单击左侧导航栏的 **Knative > 组件管理**，选择安装 GitHub addon 组件，如图所示

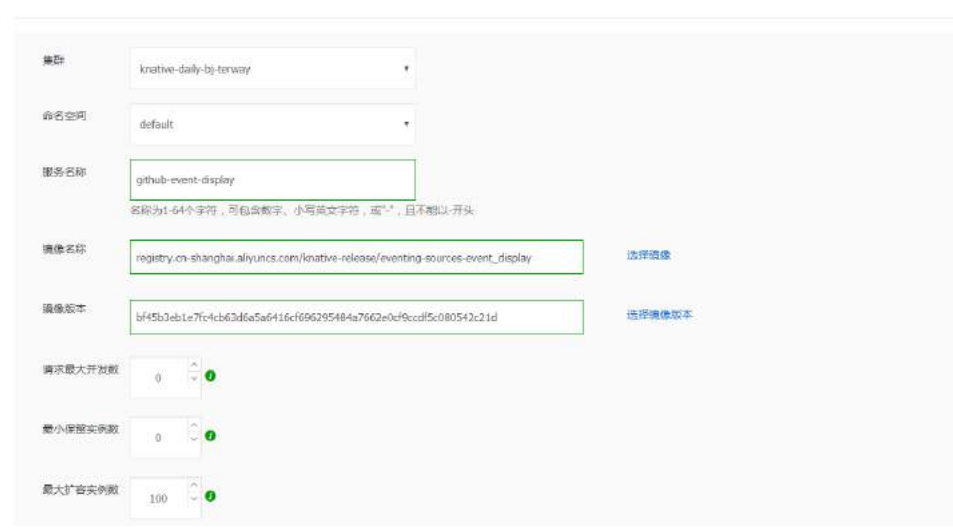


创建服务

1. 单击左侧导航栏的 **Knative > 服务管理**，进入服务管理页面。
2. 右上角单击**创建服务**。



3. 设置部署集群、命名空间、服务名称。选择所要使用的镜像和镜像的版本，如图：



- 镜像名称: registry.cn-shanghai.aliyuncs.com/knative-release/eventing-sources-event_display
- 镜像版本: bf45b3eb1e7fc4cb63d6a5a6416cf696295484a7662e0cf9cdf5c080542c21d

4. 点击**创建**按钮。

创建 GitHub Token

创建 [Personal access tokens](#), 用于访问 GitHub API。token 的名称可以任意设置。Source 需要开启 repo:public_repo 和 admin:repo_hook , 以便通过公共仓库触发 Event 事件, 并为这些公共仓库创建 webhooks 。

下面是设置一个 “GitHubSource Sample” token 的示例。

Token description

GitHubSource Sample

What's this token for?

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

<input type="checkbox"/> repo	Full control of private repositories
<input type="checkbox"/> repo:status	Access commit status
<input type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input type="checkbox"/> repo:invite	Access repository invitations
<input type="checkbox"/> admin:org	Full control of orgs and teams
<input type="checkbox"/> write:org	Read and write org and team membership
<input type="checkbox"/> read:org	Read org and team membership
<input type="checkbox"/> admin:public_key	Full control of user public keys
<input type="checkbox"/> write:public_key	Write user public keys
<input type="checkbox"/> read:public_key	Read user public keys
<input checked="" type="checkbox"/> admin:repo_hook	Full control of repository hooks
<input checked="" type="checkbox"/> write:repo_hook	Write repository hooks
<input checked="" type="checkbox"/> read:repo_hook	Read repository hooks

secretToken 内容可以通过下述方式生成随机字符串:

```
head -c 8 /dev/urandom | base64
```

更新 githubsecret.yaml 内容。如果生成的是 personal_access_token_value token, 则需要设置 secretToken 如下:

```
apiVersion: v1
kind: Secret
metadata:
  name: githubsecret
type: Opaque
stringData:
  accessToken: personal_access_token_value
  secretToken: asdfasfdsaf
```

执行命令使其生效:

```
kubectl --namespace default apply --filename githubsecret.yaml
```

创建 GitHub 事件源

为了接收 GitHub 产生的事件, 需要创建 GitHubSource 用于接收事件。

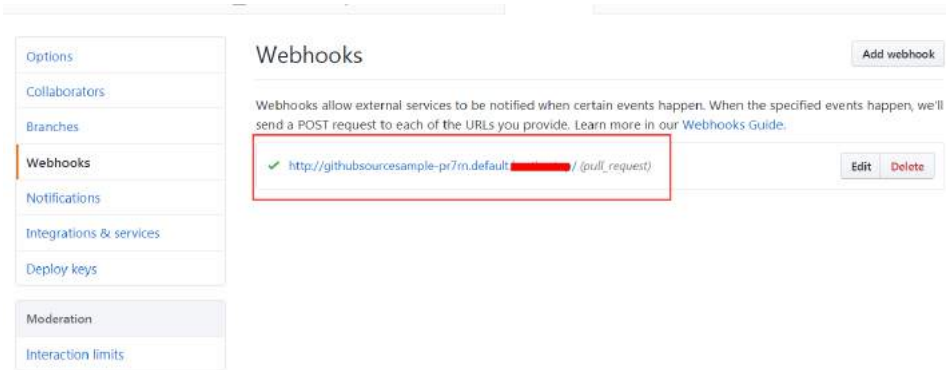
```
apiVersion: sources.eventing.knative.dev/v1alpha1
kind: GitHubSource
metadata:
  name: githubsourcesample
spec:
  eventTypes:
    - pull_request
  ownerAndRepository: <YOUR USER>/<YOUR REPO>
  accessToken:
    secretKeyRef:
      name: githubsecret
      key: accessToken
  secretToken:
    secretKeyRef:
      name: githubsecret
      key: secretToken
  sink:
    apiVersion: serving.knative.dev/v1alpha1
    kind: Service
    name: github-event-display
```

执行 kubectl 命令：

```
kubectl --namespace default apply --filename github-source.yaml
```

验证结果

在 GitHub repository 中 Settings->Webhooks 查看会有一个验证成功的 Hook URL。注意：域名需要备案，否则无法进行访问。



在 GitHub repository 中 创建一个 pull request，会产生 Event 事件，然后在 Knative Eventing 可以查看：

```
kubectl --namespace default get pods
kubectl --namespace default logs github-event-display-XXXX user-container
```

我们可以看到类似下面的事件结果：

```
2018/11/08 18:25:34 Message Dumper received a message: POST / HTTP/1.1
Host: github-event-display.knative-demo.svc.cluster.local
Accept-Encoding: gzip
Ce-Cloudeventsversion: 0.1
Ce-Eventid: a8d4cf20-e383-11e8-8069-46e3c8ad2b4d
Ce-Eventtime: 2018-11-08T18:25:32.819548012Z
Ce-Eventtype: dev.knative.source.github.pull_request
Ce-Source: https://github.com/someuser/somerepo/pull/1
Content-Length: 21060
Content-Type: application/json
User-Agent: Go-http-client/1.1
```



```
X-B3-Parentspanid: b2e514c3dbe94c03
X-B3-Sampled: 1
X-B3-Spanid: c85e346d89c8be4e
X-B3-Traceid: abf6292d458fb8e7
X-Envoy-Expected-Rq-Timeout-Ms: 60000
X-Envoy-Internal: true
X-Forwarded-For: 127.0.0.1, 127.0.0.1
X-Forwarded-Proto: http
X-Request-Id: 8a2201af-5075-9447-b593-ec3a243aff52

{"action": "opened", "number": 1, "pull_request": ...}
```

总结

通过 Knative Eventing 对接 GitHub 事件源，可以感知代码 Merge、Issue 提交等事件，进而针对这些事件触发对于的服务进行处理。

基于 Kafka 实现消息推送

当前在 Knative 中已经提供了对 Kafka 事件源的支持，那么如何基于 Kafka 实现消息推送，本文以阿里云 Kafka 产品为例，给大家解锁这一新的姿势。

背景

消息队列 for Apache Kafka 是阿里云提供的分布式、高吞吐、可扩展的消息队列服务。消息队列 for Apache Kafka 广泛用于日志收集、监控数据聚合、流式数据处理、在线和离线分析等大数据领域，已成为大数据生态中不可或缺的部分。

结合 Knative 中提供了 KafkaSource 事件源的支持，可以方便的对接 Kafka 消息服务。

另外也可以安装社区 Kafka 2.0.0 及以上版本使用。

在阿里云上创建 Kafka 实例

创建 Kafka 实例

登录[消息队列 Kafka 控制台](#)，选择【购买实例】。由于当前 Knative 中 Kafka 事件源支持 2.0.0 及以上版本，在阿里云上创建 Kafka 实例需要选择包年包月、专业版本进行购买，购买之后升级到 2.0.0 即可。

消息队列 Kafka (包年包月)

消息队列 Kafka (包年包月)

消息队列 Kafka (按量后付费)

规格类型

标准版

专业版

Kafka 支持 0.10.x ~ 1.0.x 版本，专业版仅支持 0.10.x 系列的 0.10.x 版本，支持无缝升级到 0.10.x ~ 2.x 系列版本。

地域

华东1 (杭州)

华北2 (北京)

华北1 (青岛)

华东2 (上海)

华南1 (深圳)

中国 (香港)

华北3 (张家口)

华北5 (呼和浩特)

新加坡

印度 (孟买)

印度尼西亚 (雅加达)

实例类型

VPC实例

公网/VPC实例

流量规格

20MB/s

30MB/s

60MB/s

90MB/s

120MB/s

160MB/s

200MB/s

250MB/s

300MB/s

600MB/s

800MB/s

1000MB/s

1200MB/s

1500MB/s

1800MB/s

2000MB/s

流量规格指的是业务流量，非集群流量。

业务流量规格指的是业务上实际流量产生的流量，集群流量指的是默认 Kafka 3 备份产生的包括业务和复制产生的总体流量。

举例，若自建计算的流量是 300MB/s，迁移到云上的消息流量可以购买 120MB/s。

建议，为了业务的稳定性，建议购买大于业务流量 30% 左右的余量作为 buffer。

如有疑问或报错，请提交工单咨询。

部署实例并绑定 VPC

购买完成之后，进行部署，部署时设置 Knative 集群所在的 VPC 即可：

部署

×

* VPC ID

vpcdfs-dsacac

* VSwitch ID

xxd-xssfadsl

* 可用区

请选择

▼

重新设置用户名密码 ☒ 否 ☐ 是

部署

取消

创建 Topic 和 Consumer Group

接下来我们创建 Topic 和消费组。

进入【Topic 管理】，点击创建 Topic，这里我们创建名称为 demo 的 topic：

创建Topic ×

* Topic

demo 4/64

1. Topic 名称只能包含字母，数字，下划线(_)和短横线(-)

2. 名称长度限制在 3-64 字节之间，长于 64 字节将被自动截取

3. 一旦创建后不能再修改 Topic 名称

* 实例

alibaba pre-cn-4591bo6u6008(alibaba pre-cn-4591bo6u6008) ▾

* 备注

demo 4/64

分区数

12

建议分区数是6的倍数，减少数据倾斜风险，分区数限制（1~48），特殊需求请提交工单咨询

高级设置 ▾

创建

取消

进入【Consumer Group 管理】，点击创建 Consumer Group，这里我们创建名称为 demo-consumer 的消费组：

创建Consumer Group ×

* Consumer Group

demo-consumer 13/64 ✓

1. 名称只能包含字母，数字，短横线(-)，下划线(_)

2. 名称长度限制在 3-64 字节之间，长于 64 字节将被自动截取

3. 一旦创建后不能再修改 Consumer Group 名称

* 实例

alibaba pre-cn-4591bo6u6008(alibaba pre-cn-4591bo6u6008) ▾

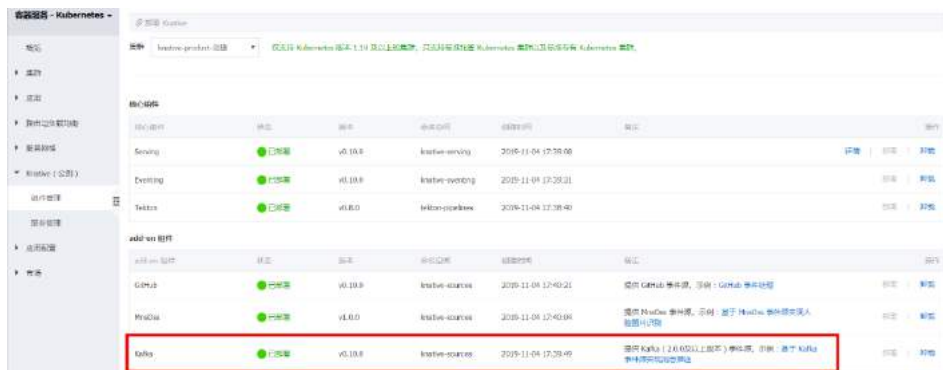
创建

取消

部署 Kafka 数据源

部署 Kafka addon 组件

登录容器服务控制台，进入【Knative 组件管理】，部署 Kafka addon 组件。



创建 KafkaSource 实例

首先创建用于接收事件的服务 event-display:

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: event-display
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/eventing-sources-cmd-event_display:bf45b3eb1e7fc4cb63d6a5a6416cf696295484a7662e0cf9ccdf5c080542c21d
```

接下来创建 KafkaSource

```
apiVersion: sources.eventing.knative.dev/v1alpha1
kind: KafkaSource
metadata:
  name: alikafka-source
spec:
  consumerGroup: demo-consumer
```

```
# Broker URL. Replace this with the URLs for your kafka cluster,
# which is in the format of my-cluster-kafka-bootstrap.my-kafka-
namespace:9092.
bootstrapServers: 192.168.0.6x:9092,192.168.0.7x:9092,192.168.0.8x:9092
topics: demo
sink:
  apiVersion: serving.knative.dev/v1alpha1
  kind: Service
  name: event-display
```

说明:

- bootstrapServers: Kafka VPC 访问地址
- consumerGroup: 设置消费组
- topics: 设置 Topic

创建完成之后，我们可以查看对应的实例已经运行：

```
[root@iZ2zeae8wzyq0ypgjowzq2Z ~]# kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
alikafka-source-k22vz-db44cc7f8-879pj 1/1     Running   0          8h
```

验证

在 Kafka 控制台，选择 topic 发送消息，注意这里的消息格式必须是 json 格式：

发送消息 ×

Topic: demo

分区

* Message Key

* Message Value

发送

取消

我们可以看到已经接收到了发送过来的 Kafka 消息：

```
[root@iZ2zeae8wzyq0ypgjowzq2Z ~]# kubectl logs event-display-zl6m5-
deployment-6bf9596b4f-8psx4 user-container

* CloudEvent: valid ✓
Context Attributes,
  SpecVersion: 0.2
  Type: dev.knative.kafka.event
  Source: /apis/v1/namespaces/default/kafkasources/alikafka-source#demo
  ID: partition:7/offset:1
  Time: 2019-10-18T08:50:32.492Z
  ContentType: application/json
  Extensions:
    key: demo
Transport Context,
  URI: /
  Host: event-display.default.svc.cluster.local
  Method: POST
Data,
  {
    "key": "test"
  }
```

总结

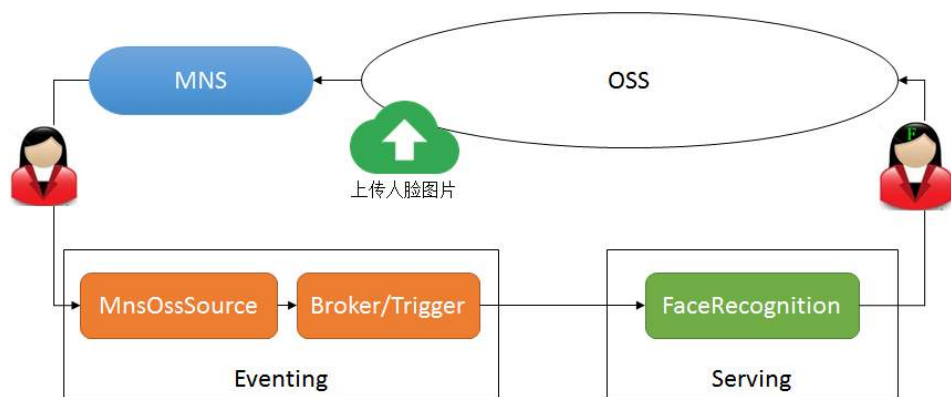
结合阿里云 Kafka 产品，通过事件驱动触发服务（函数）执行，是不是简单又高效。这样我们利用 Knative 得以把云原生的能力充分释放出来，带给我们更多的想象空间。欢迎对 Knative 感兴趣的一起交流。

基于 MNS 与 OSS 实现人脸图片识别

标准 Serverless 框架和人脸识别服务结合会产生怎样的火花？本文介绍如何通过 Knative 实现人脸识别服务，看看能否给你带来不一样的体验。

场景

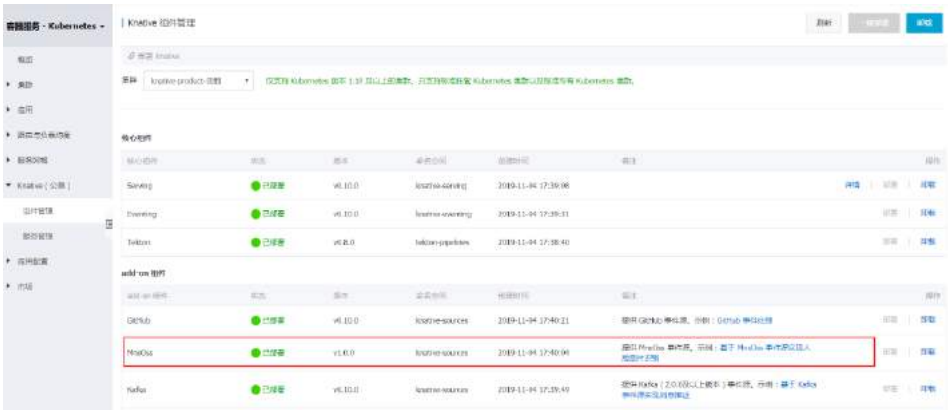
通过 OSS 控制台上传照片，MnsOss 事件源接收图片上传的事件信息，发送到 Knative Eventing，通过 Broker/Trigger 事件处理模型之后，接着触发 Knative Serving 中的人脸识别服务进行分析。最后把分析之后的图片回传到 OSS。



准备

- 安装 Knative Serving 和 Eventing
- 安装 Knative MnsOssSource 事件源服务

容器服务控制台 -> Knative -> 组件管理，选择 MnsOss 安装



The screenshot shows the Kubernetes dashboard with a list of pods. The 'NewOs' pod is highlighted with a red box. The table contains the following data:

Pod Name	Status	Image	Container Image	Creation Time	Owner	Actions
Server	Running	v1.10.0	k8s276a-40d4e4e1	2019-11-04 17:39:06		View Refresh
Server	Running	v1.10.0	k8s276a-40d4e4e1	2019-11-04 17:39:06		View Refresh
Server	Running	v1.10.0	k8s276a-40d4e4e1	2019-11-04 17:39:06		View Refresh
NewOs	Running	v1.10.0	k8s276a-40d4e4e1	2019-11-04 17:40:06		View Refresh
Server	Running	v1.10.0	k8s276a-40d4e4e1	2019-11-04 17:39:06		View Refresh

- 在 OSS 控制台创建 Bucket，参见[创建存储空间](#)
- 已开通 MNS 服务，参见[开通 MNS 服务](#)
- 已开通人脸识别服务，参见[开通人脸识别服务](#)

操作

创建 OSS 事件通知

选择 Bucket, 点击事件通知页签



The screenshot shows the OSS console with the 'Event Notification' tab selected. The table contains the following data:

Bucket Name	Event Type	Event Description	Receiver	Topic Name	Actions
face-recognition	PostObject / PutObject	Object Created / Object Deleted	face-recognition	mns-queue-oss-face-recognition-3618540660232	View

获取公网 Topic 访问连接:



这里我们选择公网访问连接: <https://xxxx.mns.cn-shanghai.aliyuncs.com/>

创建 Mns Token

获取上面的公网 Topic 访问连接以及 ak, sk 信息。按照下面的格式进行 base64 进行编码处理, 生成访问 Token。

```
# echo '{ "url":"https://xxxx.mns.cn-shanghai.aliyuncs.com/",  
"accessKeyId":"xxx","accessKeySecret":"xx" }' | base64
```

设置 mnsoss-secret.yaml 内容。则需要设置 mns 如下:

```
apiVersion: v1  
kind: Secret  
metadata:  
  name: mnsoss-secret  
type: Opaque  
data:  
  mns:  
eyAidXJsIjoiaHR0cHM6Ly94eHh4Lm1ucy5jbilzaGFuZ2hhaS5hbG15dW5jcy5jb20vIiwgImFjY  
2Vzc0tleUlkIjoieHh4IiwiaWYWNjZXRzS2V5U2VjcmV0IjoieHgiIH0K
```

执行命令使其生效:

```
kubectl apply -f mnsoss-secret.yaml
```

创建 Service Account 及角色绑定

设置 mnsoss-sa.yaml 内容。

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: eventing-sources-mnsoss
subjects:
- kind: ServiceAccount
  name: mnsoss-sa
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: eventing-sources-mnsoss-controller

---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: mnsoss-sa
```

执行命令使其生效：

```
kubectl apply -f mnsoss-sa.yaml
```

设置 istio egress (若当前命名空间下启用了 istio 注入)

```
apiVersion: networking.istio.io/v1alpha3
kind: ServiceEntry
metadata:
  name: alimns-ext
spec:
  hosts:
  - "*.aliyuncs.com"
  ports:
  - number: 443
    name: https
    protocol: HTTPS
```

创建 Broker

```
kubectl label namespace default knative-eventing-injection=enabled
```

创建 MnsOss 事件源

为了接收 MnsOss 产生的事件，需要创建 MnsOssSource 用于接收事件。

mnsoss-source.yaml 如下：

```
apiVersion: sources.eventing.knative.dev/v1alpha1
kind: MnsOssSource
metadata:
  labels:
    controller-tools.k8s.io: "1.0"
  name: mnsoss-face
spec:
  # Add fields here
  serviceAccountName: mnsoss-sa
  accessToken:
    secretKeyRef:
      name: mnsoss-secret
      key: mns
  sink:
    apiVersion: eventing.knative.dev/v1alpha1
    kind: Broker
    name: default
    topic: mns-en-topics-oss-face-image-2381221888dds9129
```

参数说明：

topic: 表示 MNS 主题名称

执行 kubectl 命令：

```
kubectl apply -f mnsoss-source.yaml
```

创建 Knative Service

为了验证 MnsOssSource 是否可以正常工作，可以这里使用人脸识别的的 Knative Service 示例。service.yaml 如下：

```

apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: face-recognition
  namespace: default
spec:
  template:
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/face-
recognition:v0.2.7
          command:
            - '/app/face-recognition'
            - '--configpath=/app/config'
          env:
            - name: ACCESSKEY_ID
              value: "xxx"
            - name: ACCESSKEY_SECRET
              value: "xxx"
            - name: UPLOAD_OSS_PATH
              value: "face-image/target"

```

env 参数说明:

UPLOAD_OSS_PATH: 表示目标图片的存放位置
 ACCESSKEY_ID: 用户 ak 信息
 ACCESSKEY_SECRET: 用户 sk 信息

执行以下命令创建 Service。

```
kubectl apply -f service.yaml
```

创建 Trigger

创建 Trigger, 订阅事件信息。trigger.yaml 如下:

```

apiVersion: eventing.knative.dev/v1alpha1
kind: Trigger
metadata:
  name: face-trigger
  namespace: default
spec:
  subscriber:

```

```
ref:
  apiVersion: serving.knative.dev/v1alpha1
  kind: Service
  name: face-recognition
```

执行 kubectl 命令:

```
kubectl apply -f trigger.yaml
```

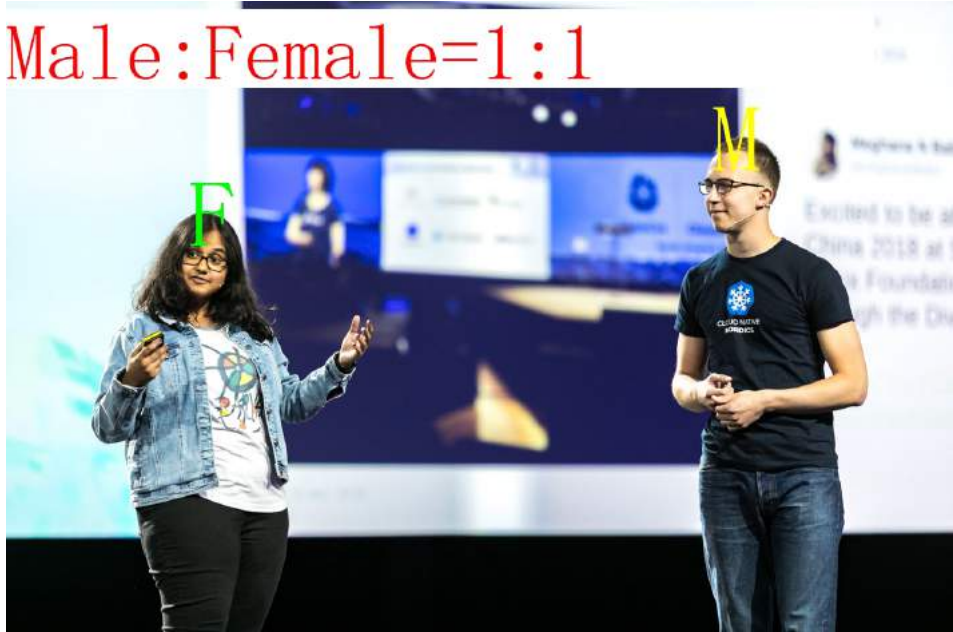
验证

通过 OSS 上传人脸图片。会在目标图片的存放位置生成人脸识别结果图片。

识别前:



识别结果:



总结

从事件源到 Eventing，再到 Serving 进行服务处理，通过 Knative 实现方式是不是给你带来了不一样的体验。你可以结合实际场景使用 Knative 打造属于自己的识别系统。

参考文档

mnsoss 代码示例: <https://github.com/knative-sample/mnsoss/tree/b1.0>

代码示例: <https://github.com/knative-sample/face-recognition/tree/b1.0>

基于 APIGateway 打造生产级别的 Knative 服务

在实际应用中，通过 APIGateway（即 API 网关），可以为内部服务提供保护，提供统一的鉴权管理，限流、监控等能力，开发人员只需要关注内部服务的业务逻辑即可。本文就会介绍一下如何通过阿里云 API 网关结合内网 SLB，将 Knative 服务对外发布，以打造生产级别的 Knative 服务。

关于阿里云 API 网关

阿里云 API 网关为您提供完整的 API 托管服务，辅助用户将能力、服务、数据以 API 的形式开放给合作伙伴，也可以发布到 API 市场供更多的开发者采购使用。

- 提供防攻击、防重放、请求加密、身份认证、权限管理、流量控制等多重手段保证 API 安全，降低 API 开放风险
- 提供 API 定义、测试、发布、下线等全生命周期管理，并生成 SDK、API 说明文档，提升 API 管理、迭代的效率
- 提供便捷的监控、报警、分析、API 市场等运维、运营工具，降低 API 运营、维护成本

基于阿里云 API 网关发布服务

绑定 Istio 网关到内网 SLB

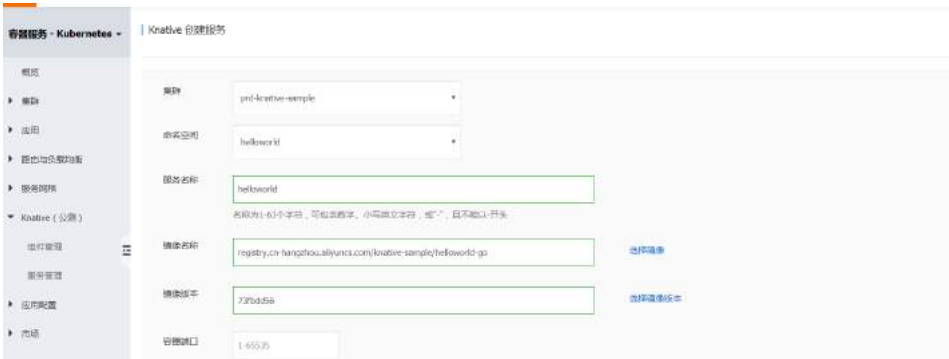
创建内网 SLB，绑定 Istio 网关应用。可以直接通过下面的 yaml 创建内网 SLB：

```
apiVersion: v1
kind: Service
metadata:
  annotations:
```


创建 Knative 服务

登录阿里云容器服务控制台，[创建 Knative 服务](#)。

这里我们创建 helloworld 服务，如图所示：



验证一下服务是否可以访问：

```
[root@izbp1c1wa320d487jdm78aZ ~]# curl -H "Host:helloworld.default.example.com" http://192.168.0.23
Hello World!
```

配置 API 网关

接下来进入重头戏，如何配置 API 网关与 Knative Service 进行访问。

创建分组

由于 API 需要归属分组，我们首先创建分组。登录[阿里云 API 网关控制台](#)，开放 API-> 分组管理：



点击【创建分组】，选择共享实例（VPC）

创建分组

✕

地域：

华东 1（杭州）（每个用户只能创建50个分组）

*实例：

共享实例（VPC）

▼

实例类型与选择指南

*分组名称：

knative_test

分组名称必须唯一，支持汉字、英文字母、数字、英文格式的下划线，必须以英文字母或汉字开头，4~50个字符

描述：

不超过180个字符

⌵

确定

取消

创建完成之后，我们需要在分组详情中开启公网域名，以进行公网服务访问：可以通过 1 开启公网二级域名进行测试，或者通过 2 设置独立域名。

[illegible]

这里我们开启公网二级域名进行测试访问，开启后如图所示：

基本信息

修改基本信息

地域：华东 1（杭州）	名称：knative-test	API 分组 ID：673d606b3820495ae7018a19c71a3069	
二级域名	公网二级域名：6714696b3820495ae7018a19c71a3069-cn-hangzhou.aliyuncs.com (该二级域名仅做测试使用，有购买 10000 次访问限制。请改用您立架架开使保等)		关闭公网二级域名
关联类型： 共享实例（VPC）	分租 ID 专 上租：500 (如需提升租额，请提交工单申请，或 购买专享实例)	变更分租实例	关闭类型与选择加解
网络访问策略	HTTPS 安全策略： HTTPS2_TLS1_0	查看 https 安全策略 https 安全策略说明	
连接状态：正常			
描述：			

独立域名

绑定域名

独立域名：	WebSocket 连接状态：	连接状态：	SSL 证书：	操作：
-------	-----------------	-------	---------	-----

创建 VPC 授权

由于我们是访问 K8s VPC 内的服务，需要创建 VPC 授权。选择 开放 API->VPC 授权：

API 网关

授权列表

全部（刷新）

华东 2（上海）

华东 1（杭州）

华东 2（北京）

华北 3（张家口）

华北 2（深圳）

华南 1（广州）

西南 1（成都）

中南 1（香港）

新加坡

亚太 1（悉尼）

亚太 2（墨尔本）

印度 1（孟买）

日本（东京）

印度（孟买）

德国（法兰克福）

美国（硅谷）

美国（弗吉尼亚）

俄罗斯（莫斯科）

全部（选择）

最近云控上库

创建授权

授权名称	VPC ID	公网 ID	端口号	创建时间	操作
knative-test	vpc-bp1gnrtm4pdkoqymr1	ip-5p113r4pdkv796q2hy	80	2018-06-05 10:01:44	删除

共 1 条，每页显示 10 条

点击【创建授权】，设置 VPC Id 以及内网 SLB 实例 Id。这里创建 knative-test VPC 授权：

创建VPC授权

×

地域：华东 1 (杭州)

*VPC授权名称：

knative-test

支持汉字、英文字母、数字、英文格式的下划线、中横线，必须以英文字母或汉字开头，4~50个字符

*VPC Id：

vpc-bp1pmlhh0jip8ncgvlmrl

VPC控制台

请输入您的VPC ID，例如：vpc-uf657xxxxxxx

*实例Id或地址：

lb-bp1l1hejtpzvq796fp2hy

请输入您的ECS或者SLB的实例ID（例如：i-uf1dfwexxxxxx 或lb-j1wb2342xxxxxx），或者对应实例的私网IP

*端口号：

80

必须是数字，2~6个字符，例如：8080

确定

取消

创建应用

创建应用用于阿里云 APP 身份认证。该认证要求请求者调用该 API 时，需通过对 APP 的身份认证。这里我们创建 knative 应用。



创建 API

登录阿里云 API 网关控制台，开放 API->API 列表，选择【创建 API】。关于创建 API，详细可参考：[创建 API](#)。



接下来我们输入【基本信息】，选择安全认证：阿里云 APP，AppCode 认证可以选择：允许 AppCode 认证 (Header & Query)。具体 AppCode 认证方式可以参考：[使用简单认证 \(AppCode\) 方式调用 API](#)



点击下一步，定义 API 请求。协议可以选择 HTTP 和 HTTPS，请求 Path 可设置 /。

请求基础定义

请求类型: ☒ 普通请求 ☐ 注册请求(双向通信) ☐ 注销请求(双向通信) ☐ 下行通知请求(双向通信)

协议: ☒ HTTP ☐ HTTPS ☐ WEBSOCKET

自定义域名:

二级域名:

请求Path: ☒ 匹配所有子路径

请求Path必须包含请求参数中的Parameter Path, 包含在[]中, 比如getUserInfo[userId]

HTTP Method:

入参请求模式:

请求中的所有参数, 包括Path中的动态参数, Headers参数, Query参数, Body参数(通过Form表单传输的参数), 参数名称保证唯一。

入参定义

代码语言	参数名	参数位置	类型	必填	默认值	示例	描述	操作
+ 添加一行								

点击下一步, 定义 API 后端服务。后端服务类型我们设置为 VPC, 设置 VPC 授权名称等。

后端基础定义

后端服务类型: ☐ HTTP(s)服务 ☒ VPC ☐ 函数计算 ☐ Mock

VPC授权名称: ☒ 使用HTTPS协议 [添加/删除VPC授权](#)

此处填写VPC授权中已经授权的VPC的授权名称, 授权名称支持系统环境变量, 如可使用VPC_ID, 如使用环境变量?

后端请求Path: ☒ 匹配所有子路径

后端请求Path必须包含后端服务参数中的Parameter Path, 包含在[]中, 比如getUserInfo[userId]

HTTP Method:

后端超时: ms

设置常量参数, 其中后端参数名称: Host, 参数值: helloworld.default.example.com, 参数位置: Header。

后端服务参数配置

修改请求

回调参数名称

回调参数位置

对传入参数名称

对传入参数位置

对传入参数类型

常量参数

回调参数名称	参数值	参数位置	描述	操作
Host	hello-world.example.com	Header		删除

新增一行

系统参数

系统参数名称	回调参数名称	参数位置	描述	操作
--------	--------	------	----	----

新增一行

上一步

下一步

点击下一步，完成创建。



发布 API

创建完成之后，可直接进行发布。

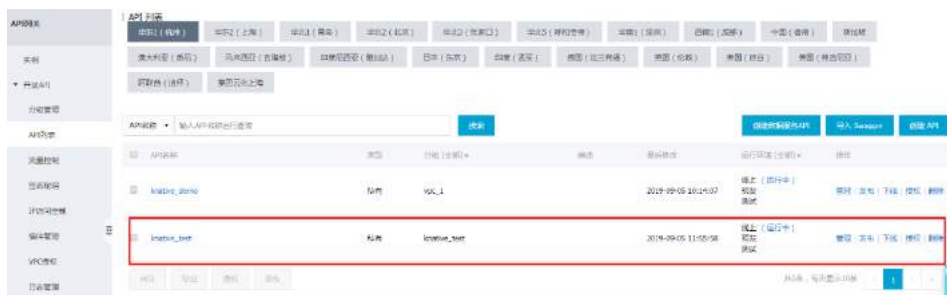


选择线上，点击【发布】



验证 API

发布完成之后，我们可以在【API 列表】中看到当前 API：线上（运行中）



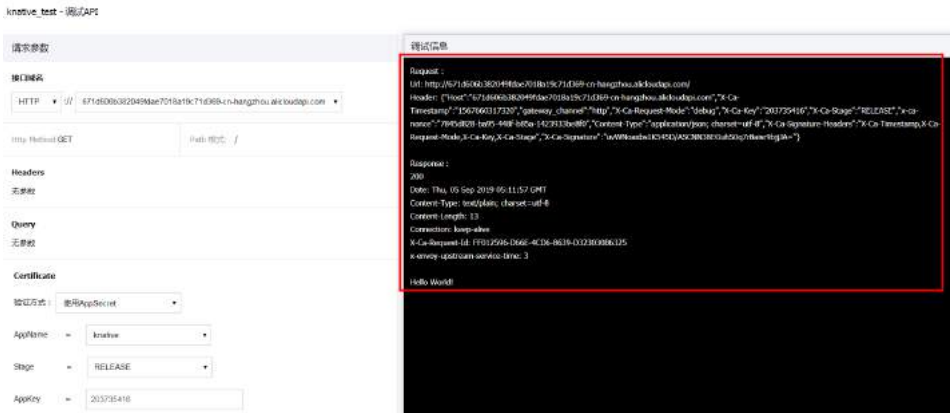
在调用 API 测试之前，我们需要对该 API 进行应用授权，进入 API 详情，选择【授权信息】



点击【添加授权】，这里我们选择上面创建的 knative 应用进行授权



接下来我们进行验证 API，点击在 API 详情中，选择【调试 API】，点击【发送请求】，可以看到测试结果信息：



至此，我们通过阿里云 API 网关将 Knative 服务发布完成。

小结

通过上面的介绍，相信大家对如何通过阿里云 API 网关将 Knative 服务对外发布有了初步的了解。在实际生产中我们对 Serverless 服务的访问安全、流控、监控运维等要求是不必可少的，而通过阿里云 API 网关恰好可以对 Knative 服务提供保驾护航能力。通过阿里云 API 网关可以对 API 服务配置：

- 流量控制
- 访问鉴权
- 日志监控
- API 全生命周期管理：测试、发布、下线

正是通过这些能力，阿里云 API 网关为 Knative 提供生产级别的服务。欢迎有兴趣的同学一起交流。

三步走！基于 Knative Serverless 技术实现一个短网址服务

短网址顾名思义就是使用比较短的网址代替很长的网址。维基百科上面的解释是这样的：

短网址又称网址缩短、缩短网址、URL 缩短等，指的是一种互联网上的技术与服务，此服务可以提供一个非常短小的 URL 以代替原来的可能较长的 URL，将长的 URL 位址缩短。用户访问缩短后的 URL 时通常将会重定向到原来的长 URL

起源

虽然现在互联网已经非常发达了，但还是有很多场景会对用户输入的内容有长度限制。比如：

- 微薄、Twitter 长度不能超过 140 个字
- 一些早期的 BBS 文章单行的长度不能超过 78 字符等场景
- 运营商短信的长度不能超过 70 个字

而现在很多媒体、电商平台的内容大多都是多人协作通过比较复杂的系统、框架生成的，链接长度几十个甚至上百字符都是很平常的事情，所以如果在上述的几个场景中传播链接使用短网址服务就是一个必然的结果。比如下面这些短信截图你应该不会陌生：

【SoulAPP】你是不是恋爱了，不然为什么这么久都不来？有人联系你，你都不理，来看看 <http://1t.click/gZy> 回T退订

【雅漾官方旗舰店】进淘宝群抢150元大额券，明星喷雾买2赠2，券后仅55.5/瓶。快入群拼手速吧 c.tb.cn/c.05jPKE，回T退

【盒马】周六到！小盒送您5元券，梭子蟹券后95元/3斤，金枕榴莲17.9/斤，更多买一赠一→ m.tb.cn/x.Val3OX 回td退订

【阿里云】【到期提醒】尊敬的lar****rd：您有2台云服务器ECS将于2019-10-10 00:00:00正式到期，截至目前仅剩30天。点击 <http://tb.cn/HJahPkw> 立即续费

【中国联通】尊敬的用户您好，07月通信账单已出，请登录中国联通手机营业厅APP查看 u.10010.cn/gACyC 通信消费一目了然，尽在掌握。还有一份大礼送您，点击 [我的] > [互动参与] > [新人百元礼包] 即可免费领取1GB流量包和最高百元话费礼包。

【丰巢】您的 3.00 元寄件红包即将在 2019-08-29 过期,请尽快使用。到货不满意,退货用丰巢,下班后也能寄。点击链接直接下单 <https://dwz.cn/Jtk36RwZ> ,退订回复TD

【滴滴快车】5张7折滴滴优惠券送到，每单最高减15元。9月2-4日限杭州地区APP: <https://z.dididi.cn/2xYeG> 退订TD

【自然堂旗舰店】99大促，抢100元无门槛优惠券，每天1场，18点准时开抢 → c.tb.cn/c.05hkEG 可回T退

应用场景

短网址服务的最初本意就是缩短长 url，方便传播。但其实短网址服务还能做很多其他的事情。比如下面这些：

- 访问次数的限制，比如只能访问 1 次，第二次访问的时候就拒绝服务
- 时间的限制，比如只能在一周内提供访问服务，超过一周就拒绝服务
- 根据访问者的地域的限制
- 通过密码访问
- 访问量统计
- 高峰访问时间统计等等
- 统计访问者的一些信息，比如：
 - 来源城市
 - 访问时间
 - 使用的终端设备、浏览器
 - 访问来源 IP
- 在营销活动中其实还可以对不同的渠道生成不通的短网址，这样通过统计这些短网址还能判断不同渠道的访问量等信息

基于 Knative Serverless 技术实现一个短网址服务

在 Knative 模式下可以实现按需分配，没有流量的时候实例缩容到零，当有流量进来的时候再自动扩容实例提供服务。

现在我们就基于阿里云容器服务的 Knative 来实现一个 serverless 模式的短网址服务。本示例会给出一个完整的 demo，你可以自己在阿里云容器服务上面创建一个 Knative 集群，使用本示例提供服务。本示例中实现一个最简单的功能。

- 通过接口实现长网址到短网址的映射服务
- 当用户通过浏览器访问短网址的时候通过 301 跳转到长网址

下面我们一步一步实现这个功能

数据库

既然要实现短网址到长网址的映射，那么就需要保存长网址的信息到数据库，并且生成一个短的 ID 作为短网址的一部分。所以我们首先需要选型使用什么数据库。在本示例中我们选择使用**阿里云的表格存储**，**表格存储最大的优势就是按量服务**，你只需要为你使用的量付费，而且价格也很实惠。如下所示的按量计费价格表。1G 的数据保存一年的费用是 **3.65292 元 / 年** ($0.000417 \times 24 \times 365 = 3.65292$)，是不是很划算。

按量计费

计量项	容量型实例	高性能实例
数据存储	0.000417元/GB/小时	0.0015元/GB/小时
预留读吞吐量	不支持	0.00056元/CU/小时
按量读吞吐量	0.004元/万CU	0.01元/万CU
预留写吞吐量	不支持	0.00139元/CU/小时
按量写吞吐量	0.002元/万CU	0.02元/万CU
外网下行流量	0.8元/GB	0.8元/GB

短网址生成 API

我们需要有一个 API 生成短网址

/new?origin-url=\${长网址}

- origin-url 访问地址

返回结果

```
vEzm6v
```

假设我们服务的域名是 short-url.default.knative.kuberun.com，那么现在访问 <http://short-url.default.knative.kuberun.com/vEzm6v> 就可以跳转到长网址了。

代码实现

```
package main

import (
    "crypto/md5"
    "encoding/hex"
    "fmt"
    "log"
    "net/http"
    "os"
    "strconv"
    "time"

    "strings"

    "github.com/aliyun/aliyun-tablestore-go-sdk/tablestore"
)

var (
    alphabet = []
    byte("abcdefghijklmnopqrstuvwxyz0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZ")
    l          = &Log{}
)

func ShortUrl(url string) string {
    md5Str := getMd5Str(url)
    var tempVal int64
```



```

var result [4]string
for i := 0; i < 4; i++ {
    tempSubStr := md5Str[i*8 : (i+1)*8]
    hexVal, _ := strconv.ParseInt(tempSubStr, 16, 64)
    tempVal = 0x3FFFFFFF & hexVal
    var index int64
    tempUri := []byte{}
    for i := 0; i < 6; i++ {
        index = 0x0000003D & tempVal
        tempUri = append(tempUri, alphabet[index])
        tempVal = tempVal >> 5
    }
    result[i] = string(tempUri)
}
return result[0]
}

func getMd5Str(str string) string {
    m := md5.New()
    m.Write([]byte(str))
    c := m.Sum(nil)
    return hex.EncodeToString(c)
}

type Log struct {
}

func (log *Log) Infof(format string, a ...interface{}) {
    log.log("INFO", format, a...)
}

func (log *Log) Info(msg string) {
    log.log("INFO", "%s", msg)
}

func (log *Log) Errorf(format string, a ...interface{}) {
    log.log("ERROR", format, a...)
}

func (log *Log) Error(msg string) {
    log.log("ERROR", "%s", msg)
}

func (log *Log) FataIf(format string, a ...interface{}) {
    log.log("FATAL", format, a...)
}

func (log *Log) Fatal(msg string) {
    log.log("FATAL", "%s", msg)
}

```

```

}

func (log *Log) log(level, format string, a ...interface{}) {
    var cstSh, _ = time.LoadLocation("Asia/Shanghai")
    ft := fmt.Sprintf("%s %s %s\n", time.Now().In(cstSh).Format("2006-01-02
15:04:05"), level, format)
    fmt.Printf(ft, a...)
}

func handler(w http.ResponseWriter, r *http.Request) {
    l := &Log{}
    l.Infof("Hello world received a request, url: %s", r.URL.Path)
    l.Infof("url: %s ", r.URL)
    //if r.URL.Path == "/favicon.ico" {
    //    http.NotFound(w, r)
    //    return
    //}

    urls := strings.Split(r.URL.Path, "/")
    originUrl := getOriginUrl(urls[len(urls)-1])
    http.Redirect(w, r, originUrl, http.StatusMovedPermanently)
}

func new(w http.ResponseWriter, r *http.Request) {
    l.Infof("Hello world received a request, url: %s", r.URL)
    l.Infof("url: %s ", r.URL)
    originUrl, ok := r.URL.Query()["origin-url"]
    if !ok {
        l.Errorf("no origin-url params found")
        w.WriteHeader(http.StatusBadRequest)
        w.Write([]byte("Bad request!"))
        return
    }

    surl := ShortUrl(originUrl[0])
    save(surl, originUrl[0])
    fmt.Fprint(w, surl)
}

func getOriginUrl(surl string) string {
    endpoint := os.Getenv("OTS_TEST_ENDPOINT")
    tableName := os.Getenv("TABLE_NAME")
    instanceName := os.Getenv("OTS_TEST_INSTANCENAME")
    accessKeyId := os.Getenv("OTS_TEST_KEYID")
    accessKeySecret := os.Getenv("OTS_TEST_SECRET")
    client := tablestore.NewClient(endpoint, instanceName, accessKeyId,
accessKeySecret)

    getRowRequest := &tablestore.GetRowRequest{}

```

```

criteria := &tablestore.SingleRowQueryCriteria{}

putPk := &tablestore.PrimaryKey{}
putPk.AddPrimaryKeyColumn("id", surl)
criteria.PrimaryKey = putPk

getRowRequest.SingleRowQueryCriteria = criteria
getRowRequest.SingleRowQueryCriteria.TableName = tableName
getRowRequest.SingleRowQueryCriteria.MaxVersion = 1

getResp, _ := client.GetRow(getRowRequest)
colmap := getResp.GetColumnMap()
return fmt.Sprintf("%s", colmap.Columns["originUrl"][0].Value)
}

func save(surl, originUrl string) {
    endpoint := os.Getenv("OTS_TEST_ENDPOINT")
    tableName := os.Getenv("TABLE_NAME")
    instanceName := os.Getenv("OTS_TEST_INSTANCENAME")
    accessKeyId := os.Getenv("OTS_TEST_KEYID")
    accessKeySecret := os.Getenv("OTS_TEST_SECRET")
    client := tablestore.NewClient(endpoint, instanceName, accessKeyId,
        accessKeySecret)

    putRowRequest := &tablestore.PutRowRequest{}
    putRowChange := &tablestore.PutRowChange{}
    putRowChange.TableName = tableName

    putPk := &tablestore.PrimaryKey{}
    putPk.AddPrimaryKeyColumn("id", surl)
    putRowChange.PrimaryKey = putPk

    putRowChange.AddColumn("originUrl", originUrl)
    putRowChange.SetCondition(tablestore.RowExistenceExpectation_IGNORE)
    putRowRequest.PutRowChange = putRowChange

    if _, err := client.PutRow(putRowRequest); err != nil {
        l.Errorf("putrow failed with error: %s", err)
    }
}

func main() {
    http.HandleFunc("/", handler)
    http.HandleFunc("/new", new)
    port := os.Getenv("PORT")
    if port == "" {
        port = "9090"
    }

    if err := http.ListenAndServe(fmt.Sprintf(":%s", port), nil); err != nil

```

```
{
    log.Fatalf("ListenAndServe error:%s ", err.Error())
}
```

代码我已经编译成镜像，你可以直接使用 registry.cn-hangzhou.aliyuncs.com/knative-sample/shorturl:v1 此镜像编部署服务。

三步走起!!!

第一步 准备数据库

首先到[阿里云开通表格存储服务](#)，然后创建一个实例和表。我们需要的结构比较简单，只需要短 URL ID 到长 URL 的映射即可，存储表结构设计如下：

名称	描述
id	短网址 ID
originUrl	长网址

第二步 获取 access key

登陆到阿里云以后鼠标浮动在页面的右上角头像，然后点击 accesskeys 跳转到 accesskeys 管理页面



点击显示即可显示 Access Key Secret



第三步 部署服务

Knative Service 的配置如下，使用前两步的配置信息填充 Knative Service 的环境变量。然后部署到 Knative 集群即可

```
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: short-url
  namespace: default
spec:
  template:
    metadata:
      labels:
        app: short-url
      annotations:
        autoscaling.knative.dev/maxScale: "20"
        autoscaling.knative.dev/minScale: "0"
        autoscaling.knative.dev/target: "100"
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/shorturl:v1
          ports:
            - name: http1
              containerPort: 8080
          env:
            - name: OTS_TEST_ENDPOINT
              value: http://t.cn-hangzhou.ots.aliyuncs.com
            - name: TABLE_NAME
              value: ${TABLE_NAME}
            - name: OTS_TEST_INSTANCENAME
              value: ${OTS_TEST_INSTANCENAME}
            - name: OTS_TEST_KEYID
              value: ${OTS_TEST_KEYID}
            - name: OTS_TEST_SECRET
```

```
value: ${OTS_TEST_SECRET}
```

把上面的这个 service 中 `${TABLE_NAME}` 这些变量替换成具体的值，然后创建 knative service:

```
└─ # kubectl get ksvc
short-url      http://short-url.default.knative.kuberun.com      short-url-
456q9          short-url-456q9          True
```

现在可以开始测试

- 生成一个短网址

```
└─ # curl 'http://short-url.default.knative.kuberun.com/
new?origin-url=https://help.aliyun.com/document_detail/121534.
html?spm=a2c4g.11186623.6.786.41e074d9oHpb02'
vEzm6v
```

curl 命令输出的结果 `VR7baa` 就是短网址的 ID

- 访问短网址

在浏览器中打开 <http://short-url.default.knative.kuberun.com/vEzm6v> 就能跳转到长 url 网址了。

小结

本实战我们只需三步就基于 Knative 实现了一个 Serverless 的短网址服务，此短网址服务在没有请求的时候可以缩容到零节省计算资源，在有很多请求的时候可以自动扩容。并且使用了阿里云表格存储，这样数据库也是按需付费。基于 Knative + TableStore 实现了短网址服务的 Serverless 化。

基于 Knative Serverless 技术实现天气服务

提到天气预报服务，我们第一反应是很简单的一个服务啊，目前网上有大把的天气预报 API 可以直接使用，有必要去使用 Knative 搞一套吗？杀鸡用牛刀？先不要着急，我们先看一下实际的几个场景需求：

- 场景需求 1：根据当地历年的天气信息，预测明年大致的高温到来的时间
- 场景需求 2：近来天气多变，如果明天下雨，能否在早上上班前，给我一个带伞提醒通知
- 场景需求 3：领导发话：最近经济不景气，公司财务紧张，那个服务器，你们提供天气、路况等服务的那几个小程序一起用吧，但要保证正常提供服务。

从上面的需求，我们其实发现，要做好一个天气预报的服务，也面临内忧（资源紧缺）外患（需求增加），并不是那么简单的。不过现在更不要着急，我们可以使用 Knative 帮你解决上面的问题。

关键词：天气查询、表格存储、通道服务，事件通知

场景需求

首先我们来描述一下我们要做的天气服务场景需求：

1. 提供对外的天气预报 RESTful API

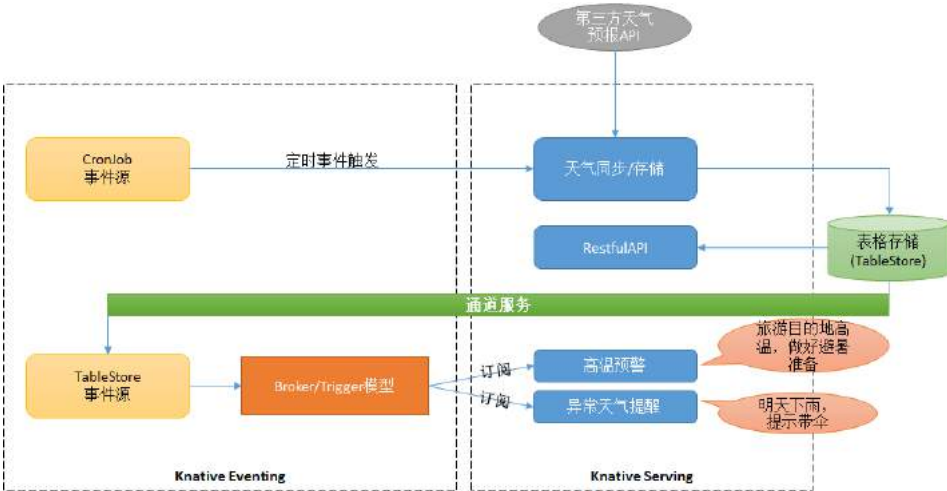
- 根据城市、日期查询（支持未来 3 天）国内城市天气信息
- 不限制查询次数，支持较大并发查询（1000）

2. 天气订阅提醒

- 订阅国内城市天气信息，根据实际订阅城市区域，提醒明天下雨带伞
- 使用钉钉进行通知

整体架构

有了需求，那我们就开始如何基于 Knative 实现天气服务。我们先看一下整体架构：



- 通过 CronJob 事件源，每隔 3 个小时定时发送定时事件，将国内城市未来 3 天的天气信息，存储更新到表格存储
- 提供 RESTful API 查询天气信息
- 通过表格存储提供的通道服务，实现 TableStore 事件源
- 通过 Borker/Trigger 事件驱动模型，订阅目标城市天气信息
- 根据订阅收到的天气信息进行钉钉消息通知。如明天下雨，提示带伞等

提供对外的天气预报 RESTful API

对接高德开放平台天气预报 API

查询天气的 API 有很多，这里我们选择高德开放平台提供的天气查询 API，使用简单、服务稳定，并且该天气预报 API 每天提供 100000 免费的调用量，支持国内 3500 多个区域的天气信息查询。另外高德开放平台，除了天气预报，还可以提供 ip 定位、搜索服务、路径规划等，感兴趣的也可以研究一下玩法。

定时同步并更新天气信息

同步并更新天气信息

该功能主要实现对接高德开放平台天气预报 API，获取天气预报信息，同时对接阿里云表格存储服务 (TableStore)，用于天气预报数据存储。具体操作如下：

- 接收 CloudEvent 定时事件
- 查询各个区域天气信息
- 将天气信息存储或者更新到表格存储

在 Knative 中，我们可以直接创建服务如下：

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: weather-store
  namespace: default
spec:
  template:
    metadata:
      labels:
        app: weather-store
      annotations:
        autoscaling.knative.dev/maxScale: "20"
        autoscaling.knative.dev/target: "100"
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/weather-
store:1.2
      ports:
        - name: http1
          containerPort: 8080
      env:
        - name: OTS_TEST_ENDPOINT
          value: http://xxx.cn-hangzhou.ots.aliyuncs.com
        - name: TABLE_NAME
          value: weather
        - name: OTS_TEST_INSTANCENAME
          value: ${xxx}
        - name: OTS_TEST_KEYID
          value: ${yyy}
        - name: OTS_TEST_SECRET
          value: ${Pxxx}
```

```
- name: WEATHER_API_KEY
  value: xxx
```

关于服务具体实现参见 GitHub 源代码: <https://github.com/knative-sample/weather-store>

创建定时事件

这里或许有疑问: 为什么不在服务中直接进行定时轮询, 非要通过 Knative Eventing 搞一个定时事件触发执行调用? 那我们要说明一下, Serverless 时代下就该这样玩 – 按需使用。千万不要在服务中按照传统的方式空跑这些定时任务, 亲, 这是在持续浪费计算资源。

言归正传, 下面我们使用 Knative Eventing 自带的定时任务数据源 (CronJobSource), 触发定时同步事件。

创建 CronJobSource 资源, 实现每 3 个小时定时触发同步天气服务 (weather-store), WeatherCronJob.yaml 如下:

```
apiVersion: sources.eventing.knative.dev/v1alpha1
kind: CronJobSource
metadata:
  name: weather-cronjob
spec:
  schedule: "0 */3 * * *"
  data: '{"message": "sync"}'
  sink:
    apiVersion: serving.knative.dev/v1alpha1
    kind: Service
    name: weather-store
```

执行命令:

```
kubectl apply -f WeatherCronJob.yaml
```

现在我们登录阿里云表格存储服务, 可以看到天气预报数据已经按照城市、日期的格式同步进来了。



id	date(日期)	city	direction	distance	direction2	direction3	id2	temperature	wind
110000	2019-09-24	北京市	北	11	南	南	07a5a7c3-8472-492a-8...	℃3	1
110000	2019-09-25	北京市	北	11	南	南	24ba825c-08ba-42db-8...	℃3	1
110000	2019-09-26	北京市	北	11	南	南	15c95128-4483-4725-9...	℃3	1
110000	2019-09-27	北京市	北	11	南	南	ee902365-8d98-4711-9...	℃3	1
110000	2019-09-28	北京市	北	11	南	南	05ea0812-35f0-82da-8...	℃3	1
109000	2019-09-24	深圳市	北	11	南	南	c7a0a100-070e-42db-8...	℃3	1

提供天气预报查询 RESTful API

有了这些天气数据，可以随心所欲的提供属于我们自己的天气预报服务了（感觉像是承包了一块地，我们来当地主），这里没什么难点，从表格存储中查询对应的天气数据，按照返回的数据格式进行封装即可。

在 Knative 中，我们可以部署 RESTful API 服务如下：

```
apiVersion: serving.knative.dev/v1alpha1
kind: Service
metadata:
  name: weather-service
  namespace: default
spec:
  template:
    metadata:
      labels:
        app: weather-service
    annotations:
      autoscaling.knative.dev/maxScale: "20"
      autoscaling.knative.dev/target: "100"
    spec:
      containers:
        - image: registry.cn-hangzhou.aliyuncs.com/knative-sample/weather-service:1.1
      ports:
        - name: http1
          containerPort: 8080
      env:
        - name: OTS_TEST_ENDPOINT
          value: http://xxx.cn-hangzhou.ots.aliyuncs.com
        - name: TABLE_NAME
```

```

    value: weather
  - name: OTS_TEST_INSTANCENAME
    value: ${xxx}
  - name: OTS_TEST_KEYID
    value: ${yyy}
  - name: OTS_TEST_SECRET
    value: ${Pxxx}

```

具体实现源代码 GitHub 地址: <https://github.com/knative-sample/weather-service>

查询天气 RESTful API:

- 请求 URL

GET /api/weather/query

参数:

cityCode: 城市区域代码。如北京市区域代码: 110000

date: 查询日期。如格式: 2019-09-26

- 返回结果

```

{
  "code": 200,
  "message": "",
  "data": {
    "adcode": "110000",
    "city": "北京市",
    "date": "2019-09-26",
    "daypower": "≤ 3",
    "daytemp": "30",
    "dayweather": "晴",
    "daywind": "东南",
    "nightpower": "≤ 3",
    "nighttemp": "15",
    "nightweather": "晴",
    "nightwind": "东南",
    "province": "北京",
    "reporttime": "2019-09-25 14:50:46",
    "week": "4"
  }
}

```

查询：杭州，2019-09-26 天气预报信息示例

测试地址：<http://weather-service.default.knative.kuberun.com/api/weather/query?cityCode=330100&date=2019-11-06>

另外城市区域代码表可以在上面提供的源代码 GitHub 中可以查看，也可以到高德开放平台中下载：<https://lbs.amap.com/api/webservice/download>

天气订阅提醒

首先我们介绍一下表格存储提供的通道服务。通道服务 (Tunnel Service) 是基于表格存储数据接口之上的全增量一体化服务。通道服务为您提供了增量、全量、增量加全量三种类型的分布式数据实时消费通道。通过为数据表建立数据通道，您可以简单地实现对表中历史存量和新增数据的消费处理。通过数据通道可以进行数据同步、事件驱动、流式数据处理以及数据搬迁。这里事件驱动正好契合我们的场景。

自定义 TableStore 事件源

在 Knative 中自定义事件源其实很容易，可以参考官方提供的自定义事件源的实例：<https://github.com/knative/docs/tree/master/docs/eventing/samples/writing-a-source>。

我们这里定义数据源为 AliTablestoreSource。代码实现主要分为两部分：

1. 资源控制器 -Controller：接收 AliTablestoreSource 资源，在通道服务中创建 Tunnel。
2. 事件接收器 -Receiver：通过 Tunnel Client 监听事件，并将接收到的事件发送到目标服务 (Broker)

关于自定义 TableStore 事件源实现参见 GitHub 源代码：<https://github.com/knative-sample/tablestore-source>

部署自定义事件源服务如下：

从 <https://github.com/knative-sample/tablestore-source/tree/master/>

[config](#) 中可以获取事件源部署文件，执行下面的操作

```
kubectl apply -f 200-serviceaccount.yaml -f 201-clusterrole.yaml
-f 202-clusterrolebinding.yaml -f 300-alitablestoresource.yaml -f
400-controller-service.yaml -f 500-controller.yaml -f 600-istioegress.yaml
```

部署完成之后，我们可以看资源控制器已经开始运行：

```
[root@iZ8vb5wa3qvlgrgb3lxqpZ config]# kubectl -n knative-sources get pods
NAME                                READY    STATUS    RESTARTS   AGE
alitablestore-controller-manager-0  1/1      Running   0           4h12m
```

创建事件源

由于我们是通过 Knative Eventing 中 Broker/Trigger 事件驱动模型对天气事件进行处理。首先我们创建用于数据接收的 Broker 服务。

创建 Broker

```
apiVersion: eventing.knative.dev/v1alpha1
kind: Broker
metadata:
  name: weather
spec:
  channelTemplateSpec:
    apiVersion: messaging.knative.dev/v1alpha1
    kind: InMemoryChannel
```

创建事件源实例

这里需要说明一下，创建事件源实例其实就是在表格存储中创建通道服务，那么就需要配置访问通道服务的地址、accessKeyId 和 accessKeySecret，这里参照格式：{"url":"https://xxx.cn-beijing.ots.aliyuncs.com/", "accessKeyId":"xxxx","accessKeySecret":"xxxx"} 设置并进行 base64 编码。将结果设置到如下 Secret 配置文件 alitablestore 属性中：

```
apiVersion: v1
kind: Secret
metadata:
```

```

name: alitablestore-secret
type: Opaque
data:
  # { "url": "https://xxx.cn-beijing.ots.aliyuncs.com/",
  "accessKeyId": "xxxx", "accessKeySecret": "xxxx" }
  alitablestore: "<base64>"

```

创建 RBAC 权限

```

apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: eventing-sources-alitablestore
subjects:
- kind: ServiceAccount
  name: alitablestore-sa
  namespace: default
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: eventing-sources-alitablestore-controller

---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: alitablestore-sa
secrets:
- name: alitablestore-secret

```

创建 AliTablestoreSource 实例，这里我们设置接收事件的 sink 为上面创建的 Broker 服务。

```

---
apiVersion: sources.eventing.knative.dev/v1alpha1
kind: AliTablestoreSource
metadata:
  labels:
    controller-tools.k8s.io: "1.0"
  name: alitablestoresource
spec:
  # Add fields here
  serviceAccountName: alitablestore-sa
  accessToken:
    secretKeyRef:

```



```
name: alitablestore-secret
key: alitablestore
tableName: weather
instance: knative-weather
sink:
  apiVersion: eventing.knative.dev/v1alpha1
  kind: Broker
  name: weather
```

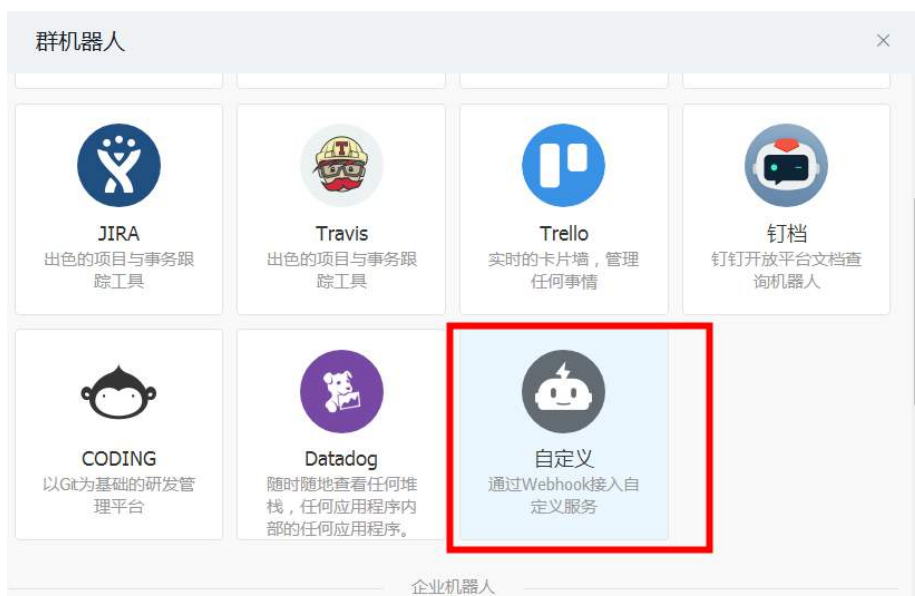
创建完成之后，我们可以看到运行中的事件源：

```
[root@iz8vb5wa3qvlgrgb3lxqpZ config]# kubectl get pods
NAME                                     READY
STATUS      RESTARTS   AGE
tablestore-alitablestoresource-9sjqx-656c5bf84b-pbhvw  1/1
Running      0          4h9m
```

订阅事件和通知提醒

创建天气提醒服务

如何进行钉钉通知呢，我们可以创建一个钉钉的群组（可以把家里人组成一个钉钉群，天气异常时，给家人一个提醒），添加群机器人：




```
image: registry.cn-hangzhou.aliyuncs.com/knative-sample/dingtalk-weather-service:1.2
```

关于钉钉提醒服务具体实现参见 GitHub 源代码: <https://github.com/knative-sample/dingtalk-weather-service>

创建订阅

最后我们创建 Trigger 订阅天气事件, 并且触发天气提醒服务:

```
apiVersion: eventing.knative.dev/v1alpha1
kind: Trigger
metadata:
  name: weather-trigger
spec:
  broker: weather
  subscriber:
    ref:
      apiVersion: serving.knative.dev/v1alpha1
      kind: Service
      name: day-weather
```

订阅之后, 如果北京 (110000), 日期: 2019-10-13, 天气有雨, 会收到如下的钉钉提醒:



这里其实还有待完善的地方：

- 是否可以基于城市进行订阅（只订阅目标城市）？
- 是否可以指定时间发送消息提醒（当天晚上 8 点准时推送第 2 天的天气提醒信息）？

有兴趣的可以继续完善当前的天气服务功能。

总结

通过上面的介绍，大家对如何通过 Knative 提供天气查询、订阅天气信息，钉钉推送通知提醒应该有了更多的体感，其实类似的场景我们有理由相信通过 Knative Serverless 可以帮你做到资源利用游刃有余。欢迎持续关注。



微信扫一扫关注
阿里巴巴云原生公众号



钉钉扫描二维码立即加入
Knative 交流群



钉钉微信扫一扫关注
云原生技术圈

