阿里云开发者社区 ALIBABA CLOUD DEVELOPER COMMUNITY | 阿里巴巴云原生 | RocketMQ

云原生消息队列 Apache RocketMQ

传统的消息中间件如何持续进化为云原生的消息服务

RocketMQ Prometheus Exporter
RocketMQ Operator
Service Mesh
Serverless

藏经阁

阿里云开发者电子书系列

RocketMQ 中国社区钉钉群
欢迎各位开发者进群交流、勘误

微信扫码关注
“阿里巴巴云原生公众号”

阿里云开发者“藏经阁”
海量电子书免费下载

目录

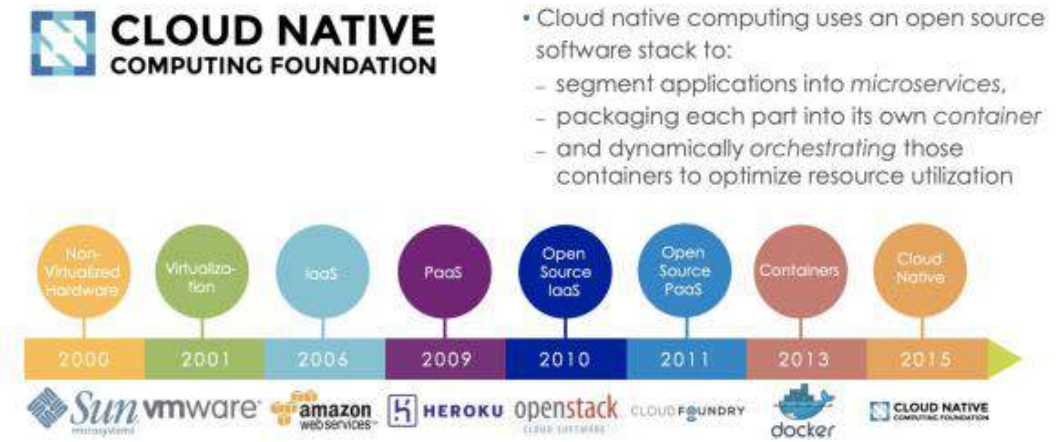
开篇：云原生时代消息中间件的演进路线

作者：尘央 阿里云消息中间件架构师

PC 端登录 start.aliyun.com 即在浏览器中体验 RocketMQ 在线可交互教程

一、引言

本文以一张云进化历史图开场，来谈谈云原生时代消息中间件的演进路线，但本文绝对不是“开局一张图，内容全靠编”。



从虚拟化技术诞生以来，IaaS/PaaS/SaaS 概念陆续被提了出来，各种容器技术层出不穷。到 2015 年，Cloud Native 概念应运而生，一时间，各种云厂商，云服务以及云应用都加上了“云原生”前缀。

我们也一直在思考，什么样的消息中间件才能称得上是“云原生”的，这也是本文探讨的主题：传统的消息中间件如何持续进化为云原生的消息服务。

二、云原生消息服务

1. 什么是云原生

首先来谈谈什么是云原生，云原生是一个天然适用于云计算的架构理念，实践云原生技术理念的应用可以最大化享受云计算的技术红利，包括弹性伸缩、按量付费、无厂商绑定、高 SLA 等。

应用在实践云原生技术理念时一般会遵循四个要素：

- 采取 DevOps 领域的最佳实践来管理研发和运维流程。
- 通过 CICD 工具链做到应用的快速迭代和持续交付。
- 采取微服务架构。
- 采取容器及相关技术进行应用的托管。

消息服务作为应用的通信基础设施，是微服务架构应用的核心依赖，也是实践云原生的核心设计理念的关键技术，通过消息服务能够让用户很容易架构出分布式的、高性能的、弹性的、鲁棒的应用程序。消息服务在云原生的重要性也导致其极可能成为应用实践云原生的阻塞点，所以消息服务的云原生化是至关重要的。

2. 什么是云原生消息服务

先说结论，我们认为云原生消息服务是云原生的通信基础设施。2015 年成立的 CNCF 基金会大范围推广了云原生的技术理念，并提供了一套完整的实践技术工具集，帮助开发者落地云原生理念。这套工具集收录于 CNCF 云原生全景图，其中消息中间件处于应用定义和开发层的 Streaming 和 Messaging 类目。

消息中间件在云原生的应用场景，主要是为微服务和 EDA 架构提供核心的解耦、异步和削峰的能力，在云原生全景图定义的其它层次领域，消息服务还发挥着数据通道、事件驱动、集成与被集成等重要作用。

另外云原生倡导面向性能设计，基于消息队列的异步调用能够显著降低前端业务的响应时间，提高吞吐量；基于消息队列还能实现削峰填谷，把慢服务分离到后置链路，提升整个业务链路的性能。

3. 云原生消息服务演进方向

云原生时代对云服务有着更高的要求，传统的消息服务在云原生这个大背景下如何持续进化为云原生的消息服务，我们认为方向有这么几个：

高 SLA

云原生应用将对消息这种云原生 BaaS 服务有更高的 SLA 要求，应用将假设其依赖的云原生服务具备跟云一样的可用性，从而不需要去建设备份链路来提高应用的可用性，降低架构的复杂度。只有做到与云一样的可用性，云在服务就在，才能称为真正的云原生服务。

低成本

在过去，每家公司自建消息中间件集群，或是自研的、或是开源的，需要投入巨大的研发、运维成本。云原生时代的消息服务借助 Serverless 等弹性技术，无需预先 Book 服务器资源，无需容量规划，采取按量付费这种更经济的模式将大幅度降低成本。

易用性

在云原生时代，消息服务第一步将进化成为一种所见即所得、开箱即用的服务，易用性极大的提高。接下来，消息服务将以网格的形式触达更复杂的部署环境，小到 IoT 设备，大到自建 IDC，都能以跟公有云同样易用的方式接入消息服务，且能轻易地满足云边端一体化、跨 IDC、跨云等互通需求，真正成为应用层的通信基础设施。

多样性

云原生消息服务将致力于建设大而全的消息生态，来涵盖丰富的业务场景，提供各式各样的解决方案，从而满足不同用户的多样性需求。阿里云消息队列目前建设了多个子产品线来支撑丰富的业务需求，比如消息队列 RocketMQ，Kafka，微消息队列等。

标准化

容器镜像这项云原生的核心技术轻易地实现了不可变基础设施，不可变的镜像消除了 IaaS 层的差异，让云原生应用可以在不同的云厂商之间随意迁移。但实际上，很多云服务提供的接入形式并不是标准的，所以依赖这些非标准化云服务的应用形成了事实上的厂商锁定，这些应用在运行时是无法完成真正的按需迁移，所以只能称为某朵云上的原生应用，无法称为真正的云原生应用。因此，消息服务需要做到标准化，消除用户关于厂商锁定的担忧，目前阿里云消息队列采纳了很多社区标准，支持了多种开源的 API 协议，同时也在打造自己标准化接口。

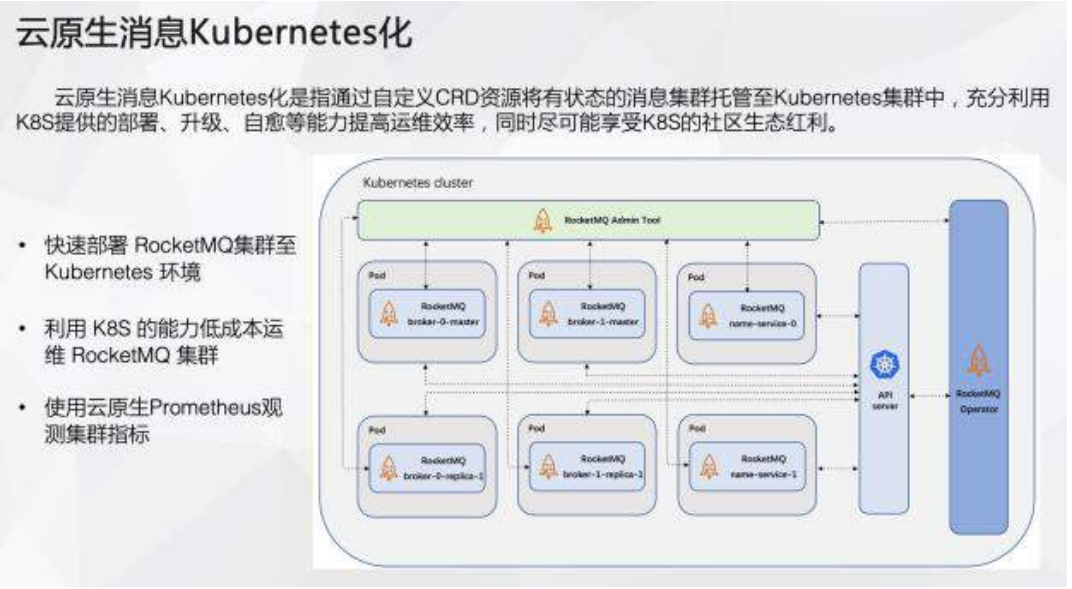
总结一下，传统的消息队列将从高 SLA、低成本、易用性、多样性和标准化几个方向持续进化为云原生的消息服务。

三、云原生消息三化

谈到云原生，离不开 Kubernetes、Serverless 以及 Service Mesh，接下来为大家分享下我们如何利用 K8S 社区的生态红利，如何实践 Serverless 和 Service Mesh 技术理念。

1. 云原生消息 Kubernetes 化

Kubernetes 项目当下绝对是大红大紫，在容器编排和应用托管领域绝对的事实标准，整个社区也是生机盎然。所以，必须将我们的消息服务升级为 K8S 环境开箱即用的服务。



云原生消息 Kubernetes 化是指通过自定义 CRD 资源将有状态的消息集群托管至 Kubernetes 集群中，充分利用 K8S 提供的部署、升级、自愈等能力提高运维效率，同时尽可能享受 K8S 的社区生态红利。

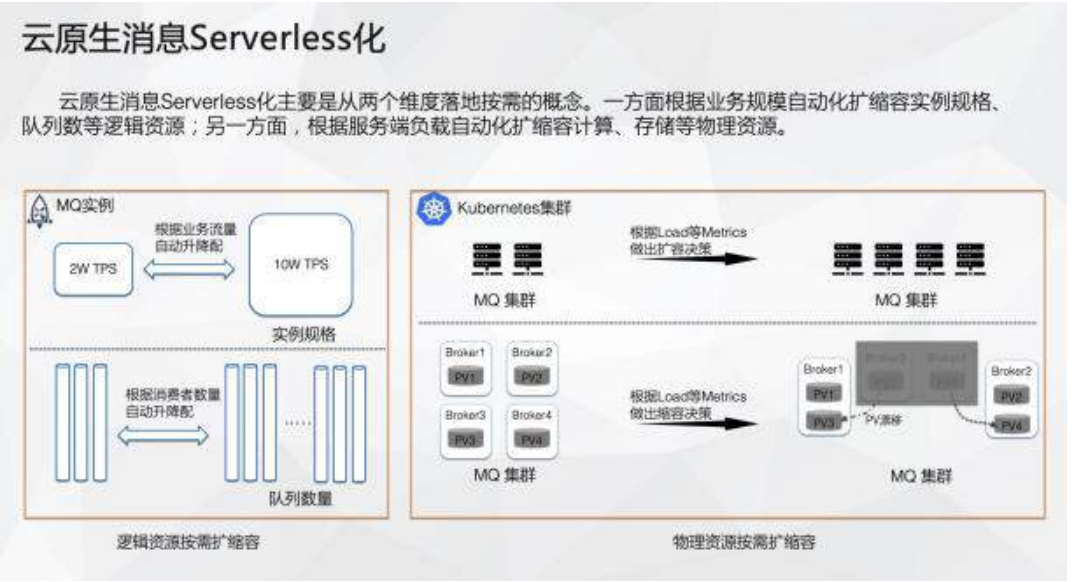
我们在 RocketMQ 开源社区也提供了 CRD 描述文件以及相应的 Operator 实现，通过这套实现，可以快速部署 RocketMQ 集群至 K8S 环境；利用 K8S 的能力低成本运维 RocketMQ 集群；也可以使用云原生的 Prometheus 观察集群指标。

RocketMQ 完成 Kubernetes 化后，就变成了 Kubernetes 环境原生可访问的一个消息服务，将给开发者带来极大的便利性。

同时，在商业化环境，我们也正在依赖 Kubeone 将消息队列系列产品完成 Kubernetes 化。

2. 云原生消息 Serverless 化

Serverless 最核心的理念是“按需”，云原生消息 Serverless 化主要是从两个维度落地按需的概念。一方面根据业务规模自动化扩缩容实例规格、队列数等逻辑资源；另一方面，根据服务端负载自动化扩缩容计算、存储等物理资源。



逻辑资源按需扩缩容：

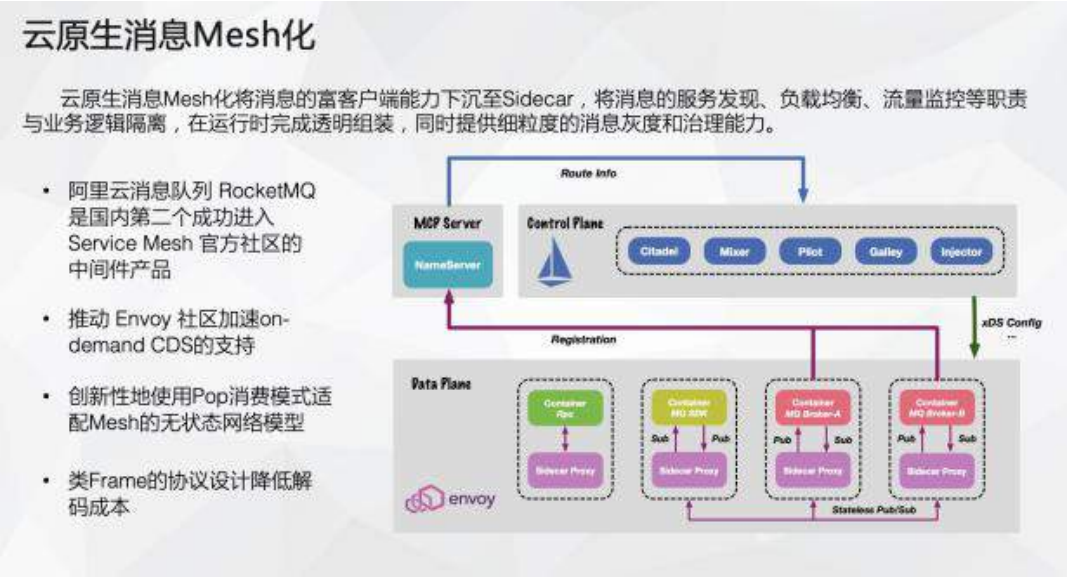
在用户侧，更关心的是消息实例提供的逻辑资源是否充足，比如购买的实例 TPS 规格是否足够，队列数量是否能满足扩展性需求。比如一个商业化的 MQ 实例中，可以根据用户的流量对实例规格进行自动的升降配，从 2W TPS 至 10W TPS 按需调整；也可

以根据用户分布式消费者的数量规模，对逻辑队列数量进行动态调整；用户完全不需要进行容量评估。

物理资源按需扩缩容：

在云服务开发者侧，我们更关心的是如何通过 Serverless 降低运维成本，避免手动的机器购买、VIP 申请、磁盘申请以及集群扩缩容等。在 Kubernetes 化完成后，可以很轻易地根据集群 Load 等指标自动扩容 MQ 物理资源；在集群缩容的处理上，会比较麻烦，因为每个 MQ 节点其实是有状态的，图中的某个 PV 代表了一个 CommitLog，我们在内部通过在 ASI 上支持 PV 漂移，在 RocketMQ 存储层支持多 CommitLog 挂载来完成自动化缩容。

3. 云原生消息 Mesh 化



Service Mesh 出发点是解决微服务架构的网络问题，将服务之间的通信问题从业务进程中进行剥离，让业务方更加专注于自身的业务逻辑。 云原生消息 Mesh 化将消息的富客户端能力下沉至 Sidecar，将消息的服务发现、负载均衡、流量监控等职责与业务逻辑隔离，在运行时完成透明组装，同时提供细粒度的消息灰度和治理能力。

目前阿里云消息队列 RocketMQ 是国内第二个成功进入 Service Mesh 官方社区的中间件产品，在进行 Envoy 适配的过程中推动了 Envoy 社区加速对 on-demand CDS 的支持，创新性地使用 Pop 消费模式来适配 Mesh 的无状态网络模型。

更详细的 Mesh 化介绍参考文章：Apache RocketMQ 的 Service Mesh 开源之旅。

四、云原生消息生态

在云原生消息服务演进方向小节中提到，云原生消息服务需要大而全的消息生态来覆盖业务方丰富的业务场景，本小节介绍我们在生态建设方面做的一些努力。

1. 云原生消息产品矩阵

阿里云消息产品矩阵包含消息队列 RocketMQ、Kafka、AMQP、微消息队列 MQTT、消息通知服务 MNS 以及即将发布的 EventBridge，涵盖互联网、大数据、移动互联网、物联网等领域的业务场景，为云原生客户提供一站式消息解决方案。

- 消息队列 RocketMQ ：阿里巴巴自主研发及双 11 交易核心链路消息产品，阿里云主打品牌，主要面向业务消息处理，打造金融级高可靠消息服务。
- 消息队列 Kafka ： 聚焦大数据生态链，100% 融合 Kafka 开源社区，大数据应用领域中不可或缺的消息产品。
- 微消息队列 MQTT ：基于 MQTT 标准协议自研，拓展消息产品的领域与边界，延伸到移动互联网以及物联网，实现端与云的连接。
- 消息队列 AMQP ：100% 兼容 AMQP 事实标准协议，全面融合 RabbitMQ 开源社区生态。
- 消息服务 MNS ：聚焦云产品生态集成 & 消息通知服务（HTTP Endpoint、Function Compute、事件通知、移动推送等）。

- 事件总线 EventBridge ：作为我们下一代的消息产品形态，原生支持 CloudEvents 标准，提供中心化事件服务能力，加速云原生生态集成，EDA 首选。

2. 云原生消息生态

在生态建设方面，我们在商业化和开源两个生态都取得了不错得成功。

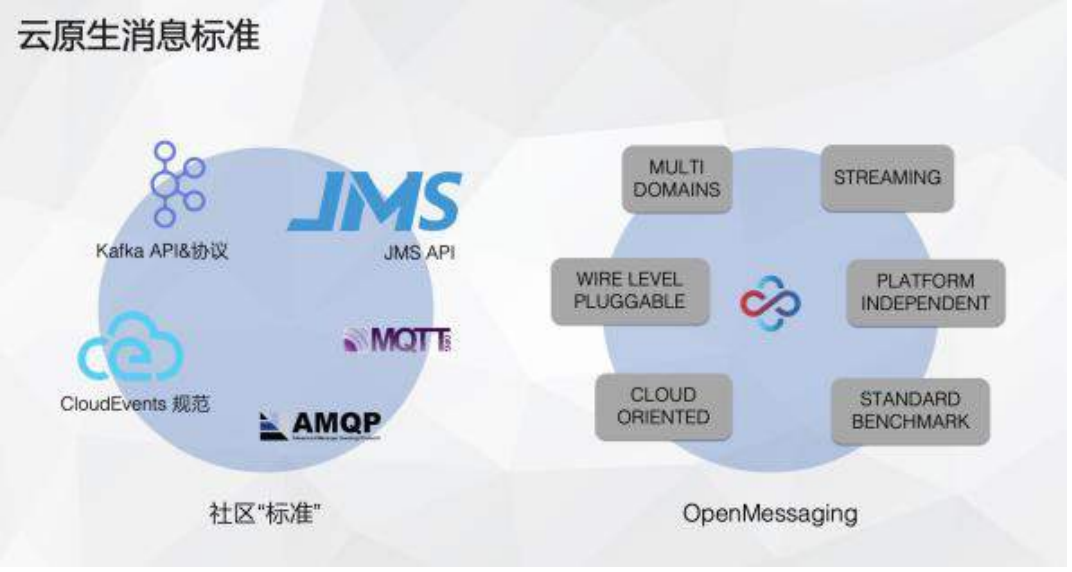
在阿里云消息商业化生态中，消息队列产品线已经支持 11 BU，30+云产品或者解决方案，有些对用户是可见的，有些是不可见的，真正做到了云原生通信基础设施的定位。

在开源方面，开源 RocketMQ 已经完成了云原生技术栈的集成，包括 Knative 中的事件源，Prometheus 的 Exporter，K8S 的 Operator 等；也支持了微服务框架 Dubbo、SpringCloud 以及函数计算框架 OpenWhisk；同时开发了很多 Connector 作为 Sink 或

者 Source 去连接了 ELK、Flume、Flink、Hadoop 等大数据和数据分析领域的优秀开源产品。

3. 云原生消息标准

最开始我们就提到标准化是云原生消息中间件的进化方向之一，我们从两个维度打磨产品的标准化建设。



社区标准 ：

在消息领域，无论是接口还是协议，社区一直有很多事实上的“标准”，比如 Kafka 提供的 API 和协议，JMS API，CloudEvents 规范，MQTT 中的协议和模型，AMQP 的协议和模型等，阿里云消息队列产品线对这些事实标准都提供了相应的接入方式，用户可以低成本完成迁移上云。

自建标准 ：

事实上的“标准”如果太多，其实就没有标准，开源方面一直在推动自建标准 OpenMessaging，OMS 将提供六大核心特性：多领域、流、平台无关、标准的 Benchmark，面向云，线路层可插拔。目前，国内有很多云提供商都接入了 OMS 标准。

五、云原生消息核心竞争力

作为云原生的消息，核心竞争力在哪，特别是开源生态愈发蓬勃，用户可选的解决方案非常多，如何让用户选择我们云原生的消息服务，我们认为核心竞争力主要有这么几个。

1. 领先的消息服务能力

阿里云的消息服务，在多个方面都具备绝对领先的服务能力。

接入&迁移

整个产品线支撑了多协议、多 API、多语言、多终端以及多生态的接入，做到了“0”接入成本，开源或自建用户都可以无缝上云；同时全球消息路由也支持跨地域的消息同步，异构的消息迁移同步等。

多租户

阿里云消息服务支持命名空间隔离、标准的访问控制；支持实例限流、资源隔离、多租户的海量堆积。

消息类型

在业务消息领域，阿里云消息有多年的业务沉淀，消息类型上支持：普通消息、事务消息、定时消息、顺序消息、重试消息以及死信消息等。

消费&治理

在消费和治理领域，云消息服务支持 Pub/Sub 模式，广播/集群消费模式，消费过程中支持 Tag 过滤、SQL 过滤。在运维时，提供了消息轨迹、消息查询以及消息回放等治理能力。

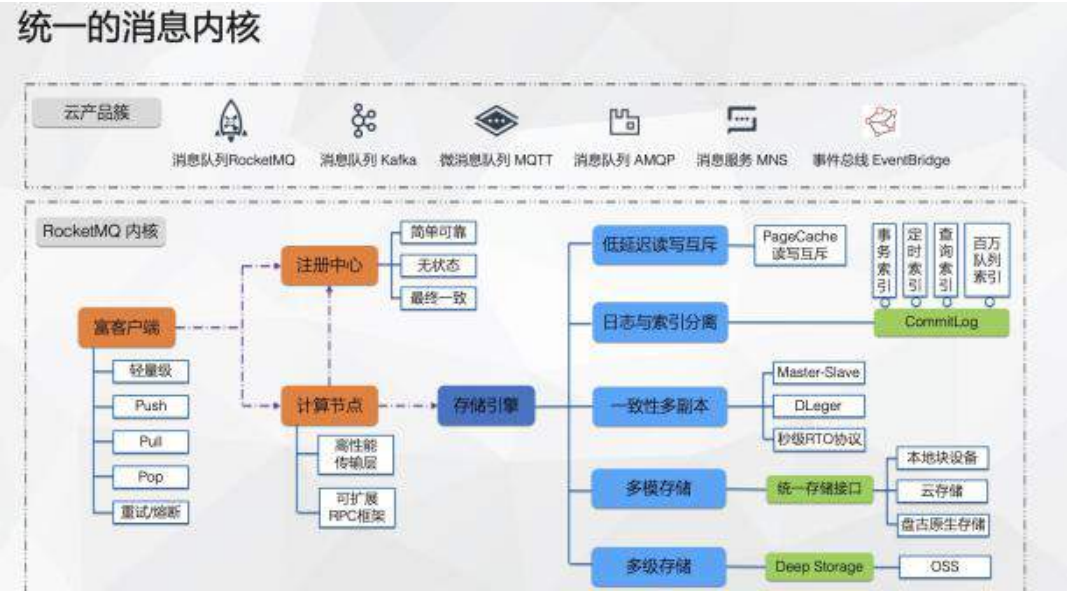
服务能力

阿里云消息的服务能力是经过多年锤炼的：

- 高可用：核心交易链路 12 年，双十一 10 年历程，在云上承诺的可用性 SLA 为 99.95%，可靠性 8 个 9。
- 高性能：双 11 消息收发 TPS 峰值过亿，日消息收发总量 3 万亿；拥有全球最大的业务消息集群之一。
- 低延迟：在双 11 万亿级数据洪峰下，消息发送 99.996%在毫秒级响应；消息发布平均响应时间不超过 3 毫秒。

2. 统一的消息内核

阿里云消息队列的另一核心竞争力为统一的消息内核，整个消息云产品簇都建设在统一的 RocketMQ 内核之上，所有的云产品提供一致的底层能力。



RocketMQ 内核主要包含以下几个模块：

富客户端

RocketMQ 提供一个轻量级的富客户端，暴露 Push、Pull 以及 Pop 三种消费模式，同时内置了重试、熔断等高可用功能，产品簇的众多客户端都是通过对内核的富客户端进行二次开发的。

注册中心

也就是 RocketMQ 开发者熟知的 NameServer，以简单可靠的方式提供集群管理、元数据管理、Topic 路由和发现等功能，节点无状态，最终一致的语义确保 NameServer 具有超高的可用性。

计算节点

Broker 中的计算部分包含一个高性能的传输层，以及一个可扩展的 RPC 框架，以支持各个产品的丰富的业务需求。

存储引擎

Broker 中的核心为存储引擎，经过多年锤炼的存储引擎包含几个核心特点：

- 低延迟读写互斥：通过在 PageCache 层完成消息的读写互斥，来大幅度保障写链路的低延迟。
- 日志与索引分离：整个存储层将消息以 Append-Only 的方式集中式存储在 CommitLog 中，同时以索引派发这种可扩展的方法来支持事务、定时、查询以及百万队列等高级特性。
- 一致性多副本：提供多套一致性多副本实现来满足不同的部署场景和需求，比如 Master-Slave 架构、基于 Raft 的 Dleger 和正在自研的秒级 RTO 多副本协议。
- 多模存储：在未来存储的方式肯定是多样化的，存储引擎抽象来统一的存储接口，并提供了本地块设备、云存储以及盘古原生存储等实现。
- 多级存储：越来越的用户对消息生命周期有了更高的要求，在过去，消息作为应用开发的中间状态，往往只会被存储数天，通过 Deep Storage，将以低成本的方式大幅延长消息的生命周期，将消息转化为用户的数据资产，以挖掘更多的诸如消息分析、消息计算需求。

3. 全方面的稳定性建设

稳定性永远是前面的 1，业务发展和创新是后面的 0。--叔同

稳定性的重要性是不言而喻的，稳定性是用户做技术和产品选型的时候考察第一要素，阿里云消息队列在稳定性方面做了全面的建设。

阿里云消息队列主要从以下几个维度进行稳定性建设：

架构开发

整个系统是面向失败设计的，除了最核心的组件，所有的外部依赖都是弱依赖；在产品迭代阶段，建立了完善的 Code Review、单元测试、集成测试、性能测试以及容灾测试流程。

变更管理

风险往往来自于变更，我们对变更的要求是可灰度、可监控、可回滚以及可降级的。

稳定性防护

限流、降级、容量评估、应急方案、大促保障、故障演练、预案演练、定期风险梳理等都是我们的稳定性防护手段。

体系化巡检

分为黑盒巡检和白盒巡检，黑盒巡检会站在用户视角对产品功能进行全方面扫描，包含了 50+检测项；白盒巡检会自动化检测 JVM 运行时指标、内核系统指标、集群统计指标等，并在指标异常时及时预警。

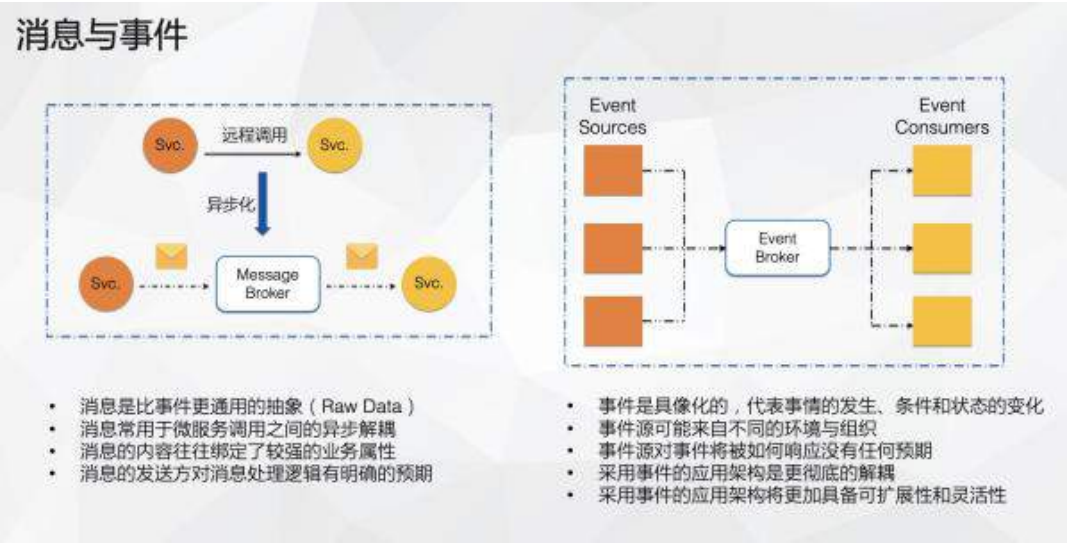
故障应急

我们建设了一套完整的故障应急流程，从监控报警->故障发生->快速止血->排查根因->故障复盘，整个链路都是顺畅的。

六、云原生消息展望

在消息产品矩阵小节中提到，EventBridge 是作为我们下一代的消息产品形态，该产品也即将迎来公测，本章节主要介绍 EventBridge 的产品定位。

1. 消息与事件



消息和事件是两种不同形态的抽象，也意味着满足不同的场景：

消息：消息是比事件更通用的抽象，常用于微服务调用之间的异步解耦，微服务调用之间往往需要等到服务能力不对等时才会去通过消息对服务调用进行异步化改造；消息的内容往往绑定了较强的业务属性，消息的发送方对消息处理逻辑是有明确的预期的。

事件：事件相对于消息更加具像化，代表了事情的发送、条件和状态的变化；事件源来自不同的组织和环境，所以事件总线天然需要跨组织；事件源对事件将被如何响应没有任何预期的，所以采用事件的应用架构是更彻底的解耦，采用事件的应用架构将更加具备可扩展性和灵活性。

2. EventBridge：中心化事件总线

EventBridge 作为我们即将发布的新产品，其核心能力之一便是提供中心化的事件总线能力。

EventBridge 将提供云产品之间、云应用之间、云产品与云应用之间，以及它们与 SaaS 提供商之间的集成与被集成能力。

在中心化事件总线中有几个重要的概念：

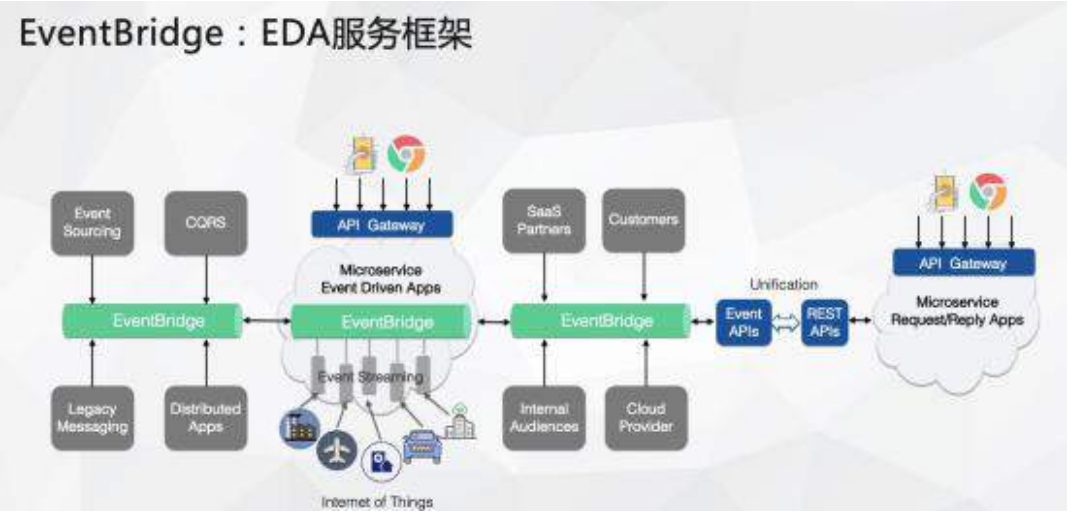
事件源：事件源可以是阿里云服务，比如对象存储、ECS、数据库等，也可以是用户的应用程序或者第三方 SaaS，这些事件源将提供丰富的业务事件、云产品运维事件、事件流等。

资源管理：EventBridge 内部将提供总线管理、规则管理以及 Schema 管理，提供全托管的事件服务。

事件处理：EventBridge 将提供事件传输、事件过滤、事件路由、事件查询、回放、重试、追踪等核心的事件处理能力。

事件目标：事件最终投递的目标服务包罗万象，既可以触发一个 Serverless 的 Function，也可以投递至消息队列其它产品，运维事件还可以通过短信、邮件以及日志服务触达运维人员。

3. EventBridge：EDA 服务框架



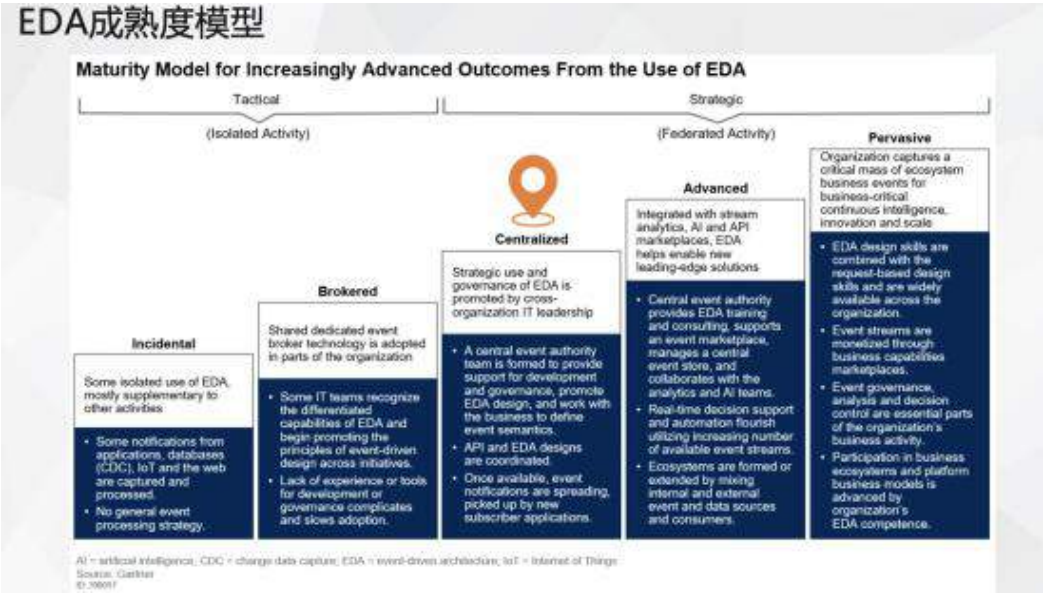
EventBridge 另一个核心能力是 EDA 服务框架。微服务有两种驱动方式：请求驱动和事件驱动。在请求驱动模式下，服务之间调用是同步调用，这种模式优点是延迟低，但是服务之间的耦合是比较重的，相比之下事件驱动有几个优点：

- 异步化，所有的调用通过事件异步化驱动，服务之间没有显示的依赖关系。
- 松耦合，通过事件总线解除所有服务之间的耦合关系。
- 可扩展，可扩展能力非常强，基于总线和事件 Schema 完成业务的扩展非常简单。
- 零改造，请求驱动的微服务，在遇到服务之间能力不对等时，往往需要进行基于消息异步化改造，避免慢调用、超时等异常情况在整个分布式集群触发雪崩效应。基于事件驱动的微服务天然具备力异步、削峰等能力，所以在业务规模扩大时不会带来额外的改造成本。

上图是设想的一个采用 EDA 的应用架构图：

- 基于 EventBridge 可以实践流行的 CQRS 和 Event Souring 范式。
- 可以从 IoT 端设备上接入 Event Streaming。
- 基于事件驱动的微服务程序也可以通过 API 网关暴露传统的 Request 请求方式，供前端访问。
- EventBridge 还可以连接第三方 SaaS 提供商，云提供商，不同组织的观察者，与分布式的事件微服务集成。
- 事件驱动和请求驱动是相辅相成的关系，通过统一 Event APIs 和 REST APIs 打通事件驱动的微服务与请求驱动的微服务，来进行架构上的融合与统一。

4. EDA 成熟度模型



我们通过 Gartner 报告总结的 EDA 成熟度模型，展望以下 EDA 架构的未来：

- Incidental：偶发性地使用事件通知机制来进行一些状态的捕获，没有明确的事件处理策略。
- Brokered：提供托管的事件代理服务，组织中部分应用开始采用基于消息或者事件的异步化架构。
- Centralized：以战略的形式提出中心化的 EDA 解决方案，有专门的组织团队提供 EDA 实现，EDA 架构开始广泛被采用。
- Advanced：EDA 架构开始触达更多的业务领域，比如流计算，数据分析，AI，以及 API 市场等，跨组织的事件生态开始形成并进行扩张。
- Pervasive：事件变得无处不在，庞大的事件生态形成，组织间的隔离被事件彻底打通，企业的关键业务都将采取 EDA 架构；事件驱动与请求驱动两个生态完成融合。

阿里云消息团队在 EDA 领域的探索目前是处于第三个阶段，未来还有很长的路要走。

Apache RocketMQ 的 Service Mesh 开源之旅

作者 | 凌楚　阿里巴巴开发工程师

简介：自 19 年底开始，支持 Apache RocketMQ 的 Network Filter 历时 4 个月的 Code Review（Pull Request），于本月正式合入 CNCF Envoy 官方社区（RocketMQ Proxy Filter 官方文档），这使得 RocketMQ 成为继 Dubbo 之后，国内第二个成功进入 Service Mesh 官方社区的中间件产品。

导读：自 19 年底开始，支持 Apache RocketMQ 的 Network Filter 历时 4 个月的 Code Review（Pull Request），于本月正式合入 CNCF Envoy 官方社区（RocketMQ Proxy Filter 官方文档），这使得 RocketMQ 成为继 Dubbo 之后，国内第二个成功进入 Service Mesh 官方社区的中间件产品。

一、Service Mesh 下的消息收发

主要流程如下图：

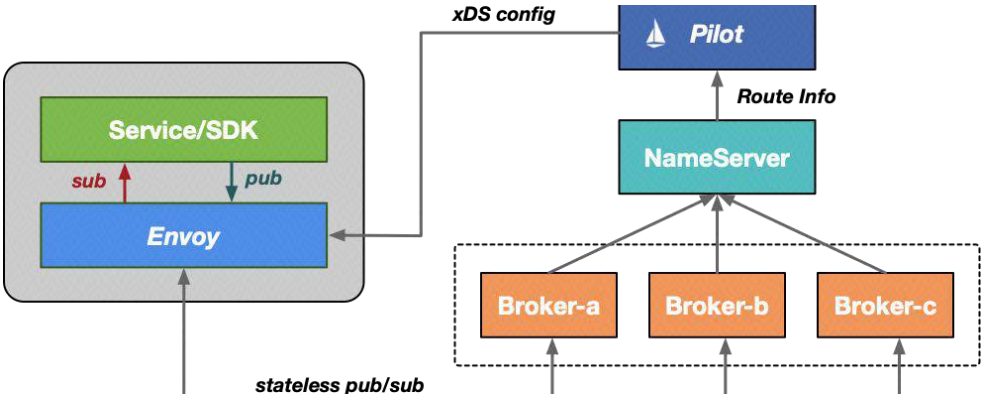


图 1

简述一下 Service Mesh 下 RocketMQ 消息的发送与消费过程：

- Pilot 获取到 Topic 的路由信息并通过 xDS 的形式下发给数据平面/Envoy ，Envoy 会代理 SDK 向 Broker/Nameserver 发送的所有的网络请求；
- 发送时，Envoy 通过 request code 判断出请求为发送，并根据 topic 和 request code 选出对应的 CDS，然后通过 Envoy 提供的负载均衡策略选出对应的 Broker 并发送，这里会使用数据平面的 subset 机制来确保选出的 Broker 是可写的；
- 消费时，Envoy 通过 request code 判断出请求为消费，并根据 topic 和 request code 选出对应的 CDS，然后和发送一样选出对应的 Broker 进行消费（与发送类似，这里也会使用 subset 来确保选出的 Broker 是可读的），并记录相应的元数据，当消息消费 SDK 发出 ACK 请求时会取出相应的元数据信息进行比对，再通过路由来准确将 ACK 请求发往上次消费时所使用的 Broker。

二、RocketMQ Mesh 化所遭遇的难题

Service Mesh 常常被称为下一代微服务，这一方面揭示了在早期 Mesh 化浪潮中微服务是绝对的主力军，另一方面，微服务的 Mesh 化也相对更加便利，而随着消息队列和一些数据库产品也逐渐走向 Service Mesh，各个产品在这个过程中也会有各自的问题亟需解决，RocketMQ 也没有例外。

三、有状态的网络模型

RocketMQ 的网络模型比 RPC 更加复杂，是一套有状态的网络交互，这主要体现在两点：

- RocketMQ 目前的网络调用高度依赖于有状态的 IP；
- 原生 SDK 中消费时的负载均衡使得每个消费者的状态不可以被忽略。

对于前者，使得现有的 SDK 完全无法使用分区顺序消息，因为发送请求和消费请求 RPC 的内容中并不包含 IP/(BrokerName + BrokerId) 等信息，导致使用了 Mesh 之后的 SDK 不能保证发送和消费的 Queue 在同一台 Broker 上，即 Broker 信息本身在 Mesh 化的过程中被抹除了。当然这一点，对于只有一台 Broker 的全局顺序消息而言是不存在的，因为数据平面在负载均衡的时候并没有其他 Broker 的选择，因此在路由层面上，全局顺序消息是不存在问题的。

对于后者，RocketMQ 的 Pull/Push Consumer 中 Queue 是负载均衡的基本单位，原生的 Consumer 中其实是要感知与自己处于同一 ConsumerGroup 下消费同一

Topic 的 Consumer 数目的，每个 Consumer 根据自己的位置来选择相应的 Queue 来进行消费，这些 Queue 在一个 Topic-ConsumerGroup 映射下是被每个 Consumer 独占的，而这一点在现有的数据平面是很难实现的，而且，现有数据平面的负载均衡没法做到 Queue 粒度，这使得 RocketMQ 中的负载均衡策略已经不再适用于 Service Mesh 体系下。

此时我们将目光投向了 RocketMQ 为支持 HTTP 而开发的 Pop 消费接口，在 Pop 接口下，每个 Queue 可以不再是被当前 Topic-ConsumerGroup 的 Consumer 独占的，不同的消费者可以同时消费一个 Queue 里的数据，这为我们使用 Envoy 中原生的负载均衡策略提供了可能。

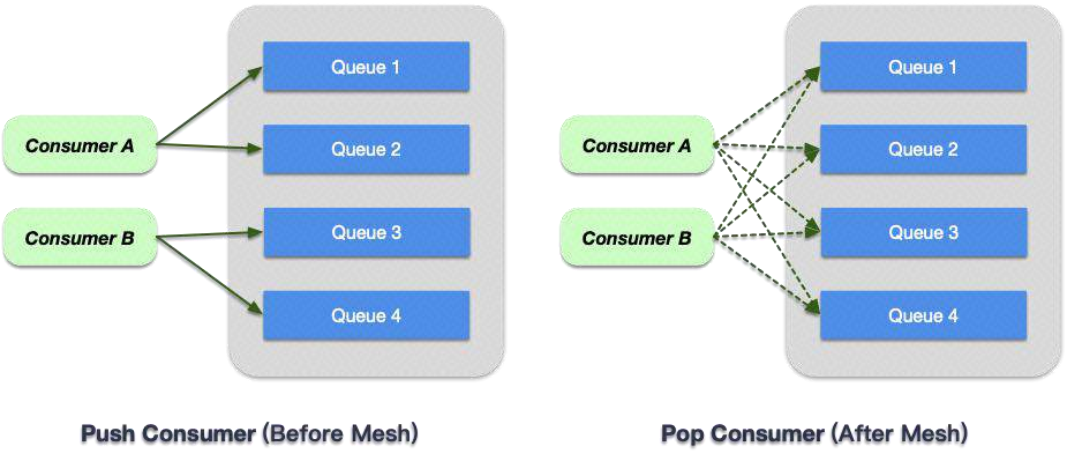


图 2

图 2 右侧即为 Service Mesh 中 Pop Consumer 的消费情况，在 Envoy 中我们会忽略掉 SDK 传来的 Queue 信息。

四、弹内海量的 Topic 路由信息

在集团内部，Nameserver 中保存着上 GB 的 Topic 路由信息，在 Mesh 中，我们将这部分抽象成 CDS，这使得对于无法预先知道应用所使用的 Topic 的情形而言，控制平面只能全量推送 CDS，这无疑会给控制平面带来巨大的稳定性压力。

在 Envoy 更早期，是完全的全量推送，在数据平面刚启动时，控制平面会下发全量的 xDS 信息，之后控制平面则可以主动控制数据的下发频率，但是无疑下发的数据依旧是全量的。后续 Envoy 支持了部分的 delta xDS API，即可以下发增量的 xDS 数据给数据平面，这当然使得对于已有的 sidecar，新下发的数据量大大降低，但是 sidecar 中拥有的 xDS 数据依然是全量的，对应到 RocketMQ ，即全量的 CDS 信息都放在内存中，这是我们不可接受的。于是我们希望能够有 on-demand CDS 的方式使得 sidecar 可以仅仅获取自己想要的 CDS 。而此时正好 Envoy 支持了 delta CDS，并仅支持了这一种 delta xDS。其实此时拥有 delta CDS 的 xDS 协议本身已经提供了 on-demand CDS 的能力，但是无论是控制平面还是数据平面并没有暴露这种能力，于是在这里对 Envoy 进行了修改并暴露了相关接口使得数据平面可以主动向控制平面发起对指定 CDS 的请求，并基于 delta gRPC 的方式实现了一个简单的控制平面。Envoy 会主动发起对指定 CDS 资源的请求，并提供了相应的回调接口供资源返回时进行调用。

对于 on-demand CDS 的叙述对应到 RocketMQ 的流程中是这样的，当 GetTopicRoute 或者 SendMessage 的请求到达 Envoy 时，Envoy 会 hang 住这个流程并发起向控制平面中相应 CDS 资源的请求并直到资源返回后重启这个流程。

关于 on-demand CDS 的修改，之前还向社区发起了 Pull Request ，现在看来当时的想法还是太不成熟了。原因是我们这样的做法完全忽略了 RDS 的存在，而将 CDS 和 Topic 实现了强绑定，甚至名称也一模一样，关于这一点，社区的 Senior Maintainer [@htuch]() 对我们的想法进行了反驳，大意就是实际上的 CDS 资源名可能带上了负载均衡方式，inbound/outbound 等各种 prefix 和 suffix，不能直接等同于 Topic 名，更重要的是社区赋予 CDS 本身的定义是脱离于业务的，而我们这样的做法过于 tricky ，是与社区的初衷背道而驰的。

因此我们就需要加上 RDS 来进行抽象，RDS 通过 topic 和其他信息来定位到具体所需要的 CDS 名，由于作为数据平面，无法预先在代码层面就知道所需要找的 CDS 名，那么如此一来，通过 CDS 名来做 on-demand CDS 就更无从谈起了，因此从这一点出发只能接受全量方案，不过好在这并不会影响代码贡献给社区。

```
route_config:
name: default_route
routes:
  - match:
      topic:
        exact: mesh
      headers:
        - name: code
          exact_match: 105
    route:
      cluster: foo-v145-acme-tau-beta-lambda
```

上面可以看到对于 topic 名为 mesh 的请求会被 RDS 路由到 foo-v145-acme-tau-beta-lambda 这个 CDS 上，事先我们只知道 topic 名，无法知道被匹配到的 CDS 资源名。

如今站在更高的视角，发现这个错误很简单，但是其实这个问题我们直到后续 code review 时才及时纠正，确实可以更早就做得更好。

不过从目前社区的动态来看，on-demand xDS 或许已经是一个 roadmap，起码目前 xDS 已经全系支持 delta ，VHDS 更是首度支持了 on-demand 的特性。

五、Mesh 为 RocketMQ 带来了什么?

A service mesh is a dedicated infrastructure layer for handling service-to-service communication. It’s responsible for the reliable delivery of requests through the complex topology of services that comprise a modern, cloud native application. In practice, the service mesh is typically implemented as an array of lightweight network proxies that are deployed alongside application code, without the application needing to be aware.

这是 Service Mesh 这个词的创造者 William Morgan 对其做出的定义，概括一下：作为网络代理，并对用户透明，承担作为基础设施的职责。

图 3

这里的职责在 RocketMQ 中包括服务发现、负载均衡、流量监控等职责，使得调用方和被代理方的职责大大降低了。

当然目前的 RocketMQ Filter 为了保证兼容性做出了很多让步，比如为了保证 SDK 可以成功获取到路由，将路由信息聚合包装成了 TopicRouteData 返回给 SDK ，但是在理想情况下，SDK 本身已经不需要关心路由了，纯为 Mesh 情景设计的 SDK 是更加精简的，不再会有消费侧 Rebalance，发送和消费的服务发现，甚至在未来像消息体压缩和 schema 校验这些功能 SDK 和 Broker 或许都可以不用再关心，来了就发送/消费，发送/消费完就走或许才是 RocketMQ Mesh 的终极形态。

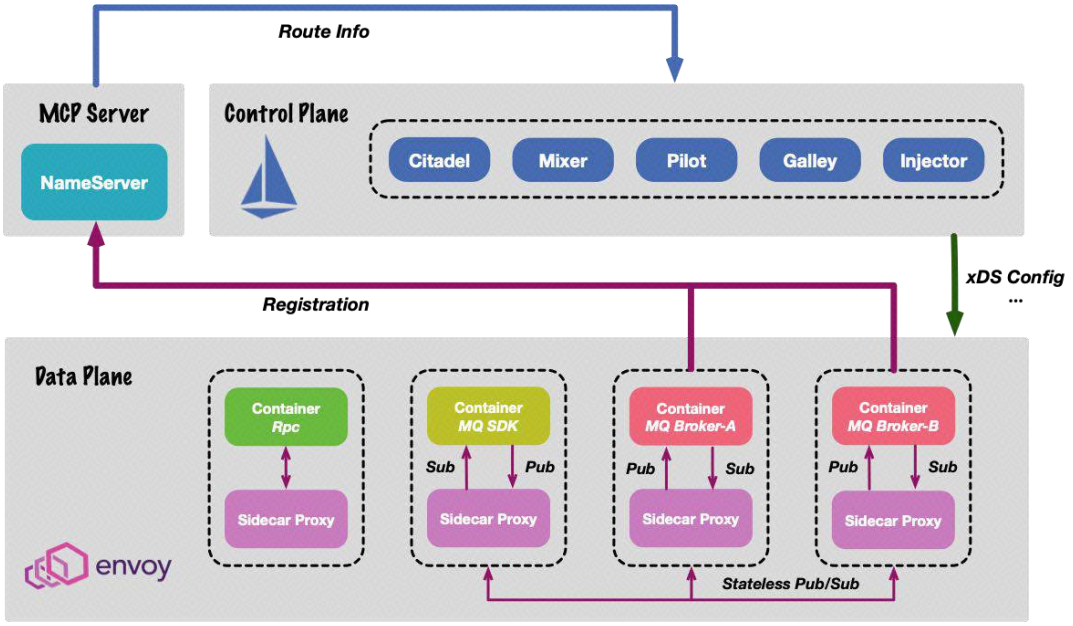


图 4

六、What's Next ?

目前 RocketMQ Filter 具备了普通消息的发送和 Pop 消费能力，但是如果想要具备更加完整的产品形态，功能上还有一些需要补充：

- **支持 Pull 请求：**现在 Envoy Proxy 只接收 Pop 类型的消费请求，之后会考虑支持普通的 Pull 类型，Envoy 会将 Pull 请求转换成 Pop 请求，从而做到让用户无感知；
- **支持全局顺序消息：**目前在 Mesh 体系下，虽然全局顺序消息的路由不存在问题，但是如果多个 Consumer 同时消费全局顺序消息，其中一个消费者突然下线导致消息没有 ACK 而会导致另一个消费者的消息产生乱序，这一点需要在 Envoy 中进行保证；

- **Broker 侧的 Proxy**：对 Broker 侧的请求也进行代理和调度。

七、蜿蜒曲折的社区历程

起初，RocketMQ Filter 的初次 Pull Request 就包含了当前几乎全部的功能，导致了一个超过 8K 行的超大 PR，感谢@天千 在 Code Review 中所做的工作，非常专业，帮助了我们更快地合入社区。

另外，Envoy 社区的 CI 实在太严格了，严格要求 97% 以上的单测行覆盖率，Bazel 源码级依赖，纯静态链接，本身无 cache 编译 24 逻辑核心 CPU 和 load 均打满至少半个小时才能编完，社区的各种 CI 跑完一次则少说两三个小时，多则六七个小时，并对新提交的代码有着极其严苛的语法和 format 要求，这使得在 PR 中修改一小部分代码就可能带来大量的单测变动和 format 需求，不过好的是单测可以很方便地帮助我们发现一些内存 case 。客观上来说，官方社区以这么高的标准来要求 contributors 确实可以很大程度上控制住代码质量，我们在补全单测的过程中，还是发现并解决了不少自身的问题，总得来说还是有一定必要的，毕竟对于 C++ 代码而言，一旦生产环境出问题，调试和追踪起来会困难得多。

最后，RocketMQ Filter 的代码由我和@叔田 共同完成，对于一个没什么开源经验的我来说，为这样的热门社区贡献代码是一次非常宝贵的经历，同时也感谢叔田在此过程中给予的帮助和建议。

八、相关链接

- Official docs for RocketMQ filter
- Pull request of RocketMQ filter
- RocketMQ filter's issue
- On-demand CDS pull request for Envoy
- First version of RocketMQ filter's proposal

阿里的 RocketMQ 如何让双十一峰值之下 0 故障

作者：愈安 阿里云消息中间件架构师

PC 端登录 start.aliyun.com 即在浏览器中体验 RocketMQ 在线可交互教程

2020 年双十一交易峰值达到 58.3 W 笔/秒，消息中间件 RocketMQ 继续数年 0 故障丝般顺滑地完美支持了整个集团大促的各类业务平稳。今年双十一大促中，消息中间件 RocketMQ 发生了以下几个方面的变化：

- 云原生化实践。完成运维层面的云原生化改造，实现 Kubernetes 化。
- 性能优化。消息过滤优化交易集群性能提升 30%。
- 全新的消费模型。对于延迟敏感业务提供新的消费模式，降低因发布、重启等场景下导致的消费延迟。

一、云原生化实践

1. 背景

Kubernetes 作为目前云原生化技术栈实践中重要的一环，其生态已经逐步建立并日益丰富。目前，服务于集团内部的 RocketMQ 集群拥有巨大的规模以及各种历史因素，因此在运维方面存在相当一部分痛点，我们希望能够通过云原生技术栈来尝试找到对应解决方案，并同时实现降本提效，达到无人值守的自动化运维。

消息中间件早在 2016 年，通过内部团队提供的中间件部署平台实现了容器化和自动化发布，整体的运维比 2016 年前已经有了很大的提高，但是作为一个有状态的服务，在运维层面仍然存在较多的问题。

中间件部署平台帮我们完成了资源的申请，容器的创建、初始化、镜像安装等一系列的基础工作，但是因为中间件各个产品都有自己不同的部署逻辑，所以在应用的发布上，就是各应用自己的定制化了。中间件部署平台的开发也不完全了解集团内 RocketMQ 的部署过程是怎样的。

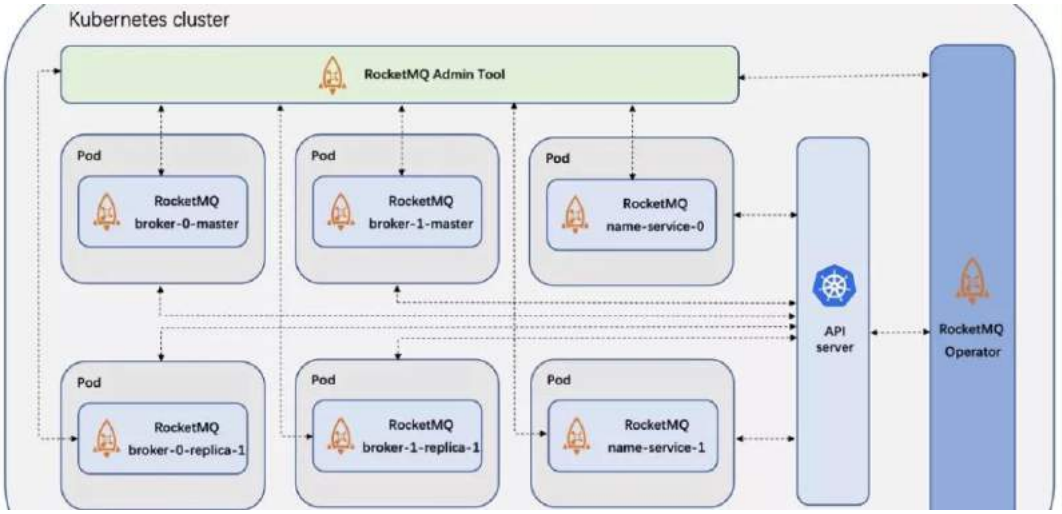
因此在 2016 年的时候，部署平台需要我们去亲自实现消息中间件的应用发布代码。虽然部署平台大大提升了我们的运维效率，甚至还能实现一键发布，但是这样的方案也有不少的问题。比较明显的就是，当我们的发布逻辑有变化的时候，还需要去修改部署平台对应的代码，需要部署平台升级来支持我们，用最近比较流行的一个说法，就是相当不云原生。

同样在故障机替换、集群缩容等操作中，存在部分人工参与的工作，如切流，堆积数据的确认等。我们尝试过在部署平台中集成更多消息中间件自己的运维逻辑，不过在

其他团队的工程里写自己的业务代码，确实也是一个不太友好的实现方案，因此我们希望通过 Kubernetes 来实现消息中间件自己的 operator 。我们同样希望利用云化后云盘的多副本能力来降低我们的机器成本并降低主备运维的复杂程度。

经过一段时间的跟进与探讨，最终再次由内部团队承担了建设云原生应用运维平台的任务，并依托于中间件部署平台的经验，借助云原生技术栈，实现对有状态应用自动化运维的突破。

2. 实现



整体的实现方案如上图所示，通过自定义的 CRD 对消息中间件的业务模型进行抽象，将原有的在中间件部署平台的业务发布部署逻辑下沉到消息中间件自己的 operator 中，托管在内部 Kubernetes 平台上。该平台负责所有的容器生产、初始化以及集团内一切线上环境的基线部署，屏蔽掉 IaaS 层的所有细节。

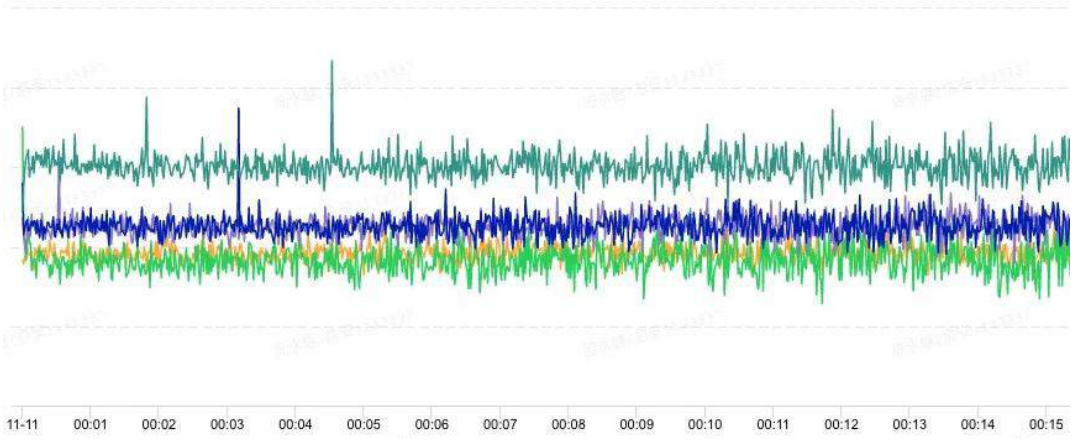
Operator 承担了所有的新建集群、扩容、缩容、迁移的全部逻辑，包括每个 pod 对应的 brokerName 自动生成、配置文件，根据集群不同功能而配置的各种开关，元数据的同步复制等等。同时之前一些人工的相关操作，比如切流时候的流量观察，下线前的堆积数据观察等也全部集成到了 operator 中。当我们有需求重新修改各种运维逻辑的时候，也再也不用去依赖通用的具体实现，修改自己的 operator 即可。

最后线上的实际部署情况去掉了图中的所有的 replica 备机。在 Kubernetes 的理念中，一个集群中每个实例的状态是一致的，没有依赖关系，而如果按照消息中间件原有的主备成对部署的方案，主备之间是有严格的对应关系，并且在上下线发布过程中有严格的顺序要求，这种部署模式在 Kubernetes 的体系下是并不提倡的。若依然采用以上老的架构方式，会导致实例控制的复杂性和不可控性，同时我们也希望能更多的遵循 Kubernetes 的运维理念。

云化后的 ECS 使用的是高速云盘，底层将对数据做了多备份，因此数据的可用性得到了保障。并且高速云盘在性能上完全满足 MQ 同步刷盘，因此，此时就可以把之前的异步刷盘改为同步，保证消息写入时的不丢失问题。云原生模式下，所有的实例环境均是一致性的，依托容器技术和 Kubernetes 的技术，可实现任何实例挂掉（包含宕机引起的挂掉），都能自动自愈，快速恢复。

解决了数据的可靠性和服务的可用性后，整个云原生化后的架构可以变得更加简单，只有 broker 的概念，再无主备之分。

3. 大促验证



上图是 Kubernetes 上线后双十一大促当天的发送 RT 统计，可见大促期间的发送 RT 较为平稳，整体符合预期，云原生化实践完成了关键性的里程碑。

二、性能优化

1. 背景

RocketMQ 至今已经连续七年 0 故障支持集团的双十一大促。自从 RocketMQ 诞生以来，为了能够完全承载包括集团业务中台交易消息等核心链路在内的各类关键业务，复用了原有的上层协议逻辑，使得各类业务方完全无感知的切换到 RocketMQ 上，并同时充分享受了更为稳定和强大的 RocketMQ 消息中间件的各类特性。

当前，申请订阅业务中台的核心交易消息的业务方一直都在不断持续增加，并且随着各类业务复杂度提升，业务方的消息订阅配置也变得更加复杂繁琐，从而使得交易集群的进行过滤的计算逻辑也变得更为复杂。这些业务方部分沿用旧的协议逻辑（Header 过滤），部分使用 RocketMQ 特有的 SQL 过滤。

2. 主要成本

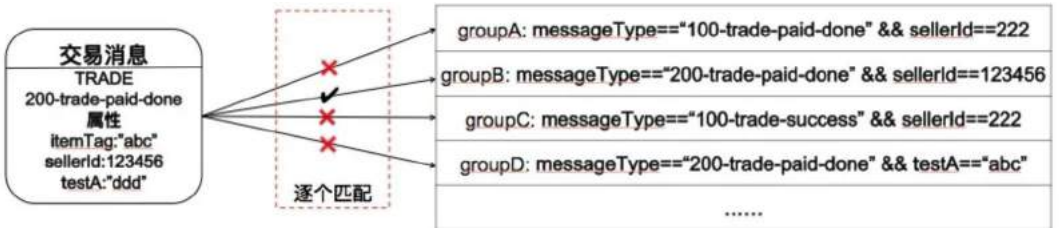
目前集团内部 RocketMQ 的大促机器成本绝大部分都是交易消息相关的集群，在双十一零点峰值期间，交易集群的峰值和交易峰值成正比，叠加每年新增的复杂订阅带来了额外 CPU 过滤计算逻辑，交易集群都是大促中机器成本增长最大的地方。

3. 优化过程

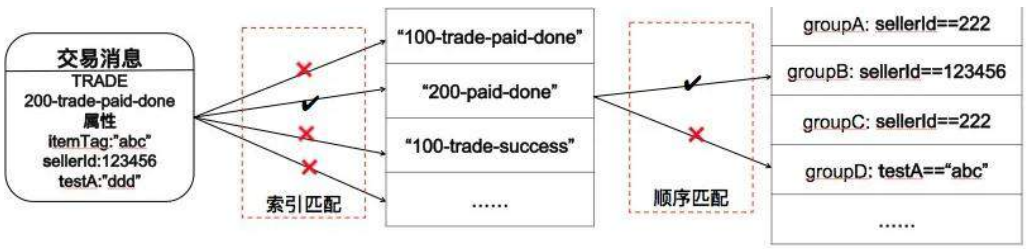
由于历史原因，大部分的业务方主要还是使用 Header 过滤，内部实现其实是 aviator 表达式（ https://github.com/killme2008/aviatorscript ）。仔细观察交易消息集群的业务方过滤表达式，可以发现绝大部分都指定类似 `MessageType == xxxx` 这样的条件。翻看 aviator 的源码可以发现这样的条件最终会调用 Java 的字符串比较 `String.compareTo()`。

由于交易消息包括大量不同业务的 MessageType，光是有记录的起码有几千个，随着交易业务流程复杂化，MessageType 的增长更是繁多。随着交易峰值的提高，交易消息峰值正比增长，叠加这部分更加复杂的过滤，持续增长的将来，交易集群的成本极可能和交易峰值指数增长，因此决心对这部分进行优化。

原有的过滤流程如下，每个交易消息需要逐个匹配不同 group 的订阅关系表达式，如果符合表达式，则选取对应的 group 的机器进行投递。如下图所示：



对此流程进行优化的思路需要一定的灵感，在这里借助数据库索引的思路：原有流程可以把所有订阅方的过滤表达式看作数据库的记录，每次消息过滤就相当于一个带有特定条件的数据库查询，把所有匹配查询（消息）的记录（过滤表达式）选取出来作为结果。为了加快查询结果，可以选择 MessageType 作为一个索引字段进行索引化，每次查询变为先匹配 MessageType 主索引，然后把匹配上主索引的记录再进行其它条件(如下图的 sellerId 和 testA)匹配，优化流程如下图所示：



以上优化流程确定后，要关注的技术点有两个：

1. 如何抽取每个表达式中的 MessageType 字段？
2. 如何对 MessageType 字段进行索引化？

对于技术点 1 ，需要针对 aviator 的编译流程进行 hook ，深入 aviator 源码后，可以发现 aviator 的编译是典型的 Recursive descent ：

http://en.wikipedia.org/wiki/Recursive_descent_parser

同时需要考虑到提取后父表达式的短路问题。

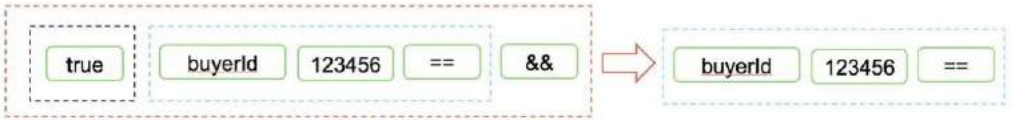
在编译过程中针对 messageType==XXX 这种类型进行提取后，把原有的 message==XXX 转变为 true/false 两种情况，然后针对 true、false 进行表达式的短路即可得出表达式优化提取后的情况。例如：

```
表达式：
messageType=='200-trade-paid-done' && buyerId==123456
提取为两个子表达式：
子表达式 1（messageType==200-trade-paid-done）：buyerId==123456
子表达式 2（messageType!=200-trade-paid-done）：false
```

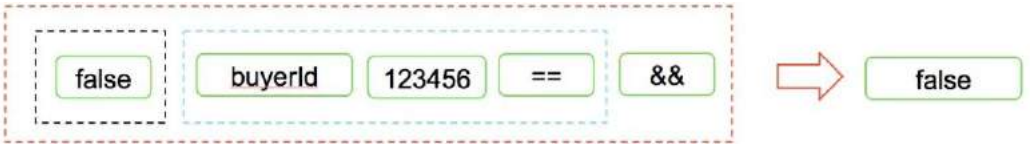
具体到 aviator 的实现里，表达式编译会把每个 token 构建一个 List ，类似如下图所示(为方便理解，绿色方框的是 token ，其它框表示表达式的具体条件组合)：

提取了 messageType ，有两种情况：

情况一：messageType == '200-trade-paid-done'，则把之前 token 的位置合并成 true，然后进行表达式短路计算，最后优化成 buyerId==123456 ，具体如下：



情况二：messageType != '200-trade-paid-done'，则把之前 token 的位置合并成 false ，表达式短路计算后，最后优化成 false ，具体如下：



这样就完成 messageType 的提取。这里可能有人就有一个疑问，为什么要考虑到上面的情况二，messageType != '200-trade-paid-done'，这是因为必须要考虑到多个条件的时候，比如：

(messageType=='200-trade-paid-done' && buyerId==123456) || (messageType =='200-trade-success' && buyerId==3333)

就必须考虑到不等于的情况了。同理，如果考虑到多个表达式嵌套，需要逐步进行短路计算。但整体逻辑是类似的，这里就不再赘述。

说完技术点 1，我们继续关注技术点 2，考虑到高效过滤，直接使用 HashMap 结构进行索引化即可，即把 messageType 的值作为 HashMap 的 key ，把提取后的子表达式作为 HashMap 的 value ，这样每次过滤直接通过一次 hash 计算即可过滤掉绝大部分不适合的表达式，大大提高了过滤效率。

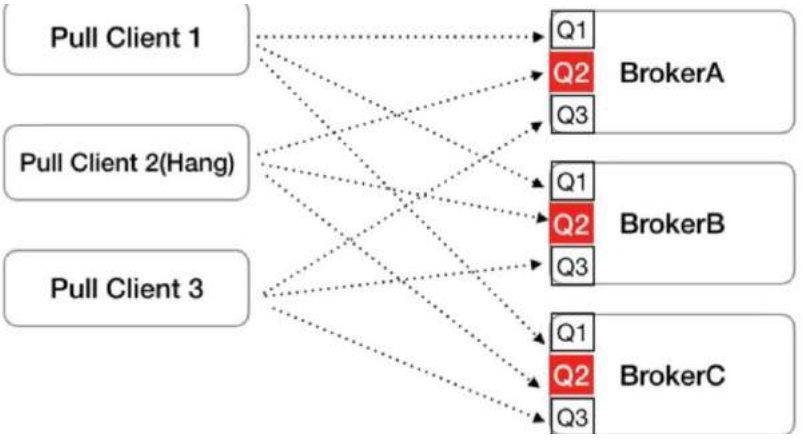
4. 优化效果

该优化最主要降低了 CPU 计算逻辑，根据优化前后的性能情况对比，我们发现不同的交易集群中的订阅方订阅表达式复杂度越高，优化效果越好，这个是符合我们的预期的，其中最大的 CPU 优化有 32% 的提升，大大降低了本年度 RocketMQ 的部署机器成本。

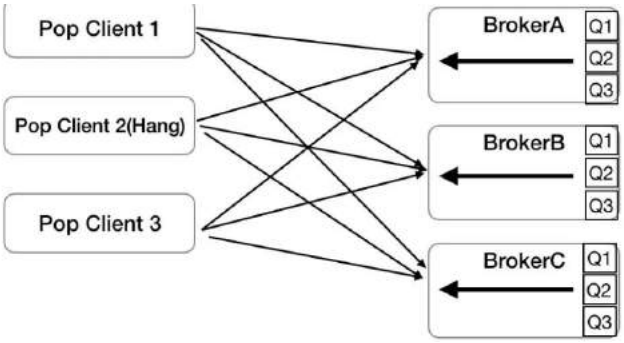
三、全新的消费模型 —— POP 消费

1. 背景

RocketMQ 的 PULL 消费对于机器异常 hang 时并不十分友好。如果遇到客户端机器 hang 住，但处于半死不活的状态，与 broker 的心跳没有断掉的时候，客户端 rebalance 依然会分配消费队列到 hang 机器上，并且 hang 机器消费速度很慢甚至无法消费的时候，这样会导致消费堆积。另外类似还有服务端 Broker 发布时，也会由于客户端多次 rebalance 导致消费延迟影响等无法避免的问题。如下图所示：



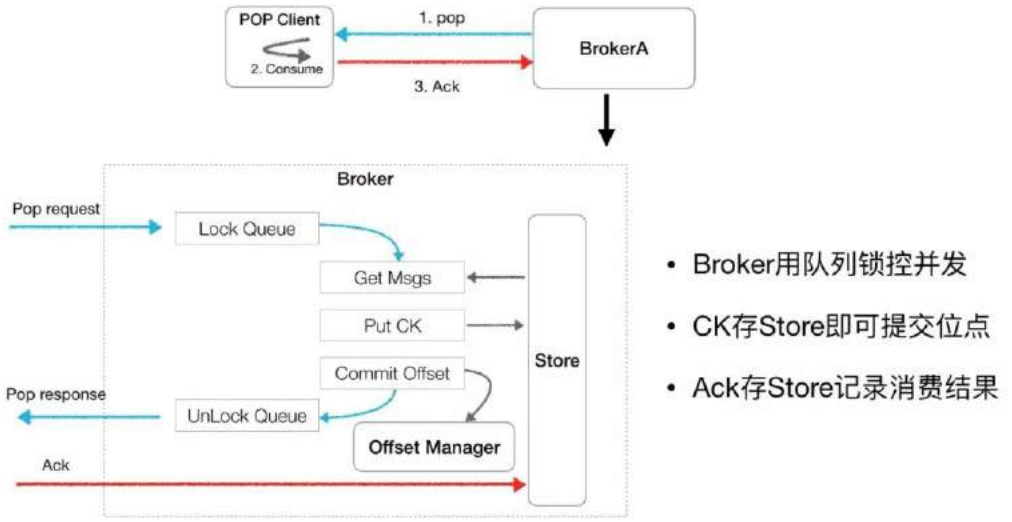
当 Pull Client 2 发生 hang 机器的时候，它所分配到的三个 Broker 上的 Q2 都出现严重的红色堆积。对于此，我们增加了一种新的消费模型——POP 消费，能够解决此类稳定性问题。如下图所示：



POP 消费中，三个客户端并不需要 rebalance 去分配消费队列，取而代之的是，它们都会使用 POP 请求所有的 broker 获取消息进行消费。broker 内部会把自身的三个队列的消息根据一定的算法分配给请求的 POP Client。即使 Pop Client 2 出现 hang，但内部队列的消息也会让 Pop Client1 和 Pop Client2 进行消费。这样就 hang 机器造成的避免了消费堆积。

2. 实现

POP 消费和原来 PULL 消费对比，最大的一点就是弱化了队列这个概念，PULL 消费需要客户端通过 rebalance 把 broker 的队列分配好，从而去消费分配到自己专属的队列，新的 POP 消费中，客户端的机器会直接到每个 broker 的队列进行请求消费，broker 会把消息分配返回给等待的机器。随后客户端消费结束后返回对应的 Ack 结果通知 broker，broker 再标记消息消费结果，如果超时没响应或者消费失败，再会进行重试。



POP 消费的架构图如上图所示。Broker 对于每次 POP 的请求，都会有以下三个操作：

1. 对应的队列进行加锁，然后从 store 层获取该队列的消息；
2. 然后写入 CK 消息，表明获取的消息要被 POP 消费；
3. 最后提交当前位点，并释放锁。

CK 消息实际上是记录了 POP 消息具体位点的定时消息，当客户端超时没响应的时候，CK 消息就会重新被 broker 消费，然后把 CK 消息的位点的消息写入重试队列。如果 broker 收到客户端的消费结果的 Ack ，删除对应的 CK 消息，然后根据具体结果判断是否需要重试。

从整体流程可见，POP 消费并不需要 reblance ，可以避免 rebalance 带来的消费延时，同时客户端可以消费 broker 的所有队列，这样就可以避免机器 hang 而导致堆积的问题。

云原生时代 RocketMQ 运维管控的利器 - RocketMQ Operator

作者 | 刘睿、杜恒

PC 端登录 start.aliyun.com 即在浏览器中体验 RocketMQ 在线可交互教程

导读：RocketMQ Operator 现已加入 OperatorHub，正式进入 Operator 社区。本文将从实践出发，结合案例来说明，如何通过 RocketMQ Operator 在 Kubernetes 上快速搭建一个 RocketMQ 集群，并提供一些 RocketMQ 集群管理功能包括 Broker 扩容等。

本文主要分为三个部分：

- 首先简单介绍一下 RocketMQ Operator 的相关知识；
- 然后结合案例详细介绍 RocketMQ Operator 提供的自定义资源及使用方法；
- 最后介绍 Operator 社区目前的情况并展望 RocketMQ Operator 下一步的发展方向。

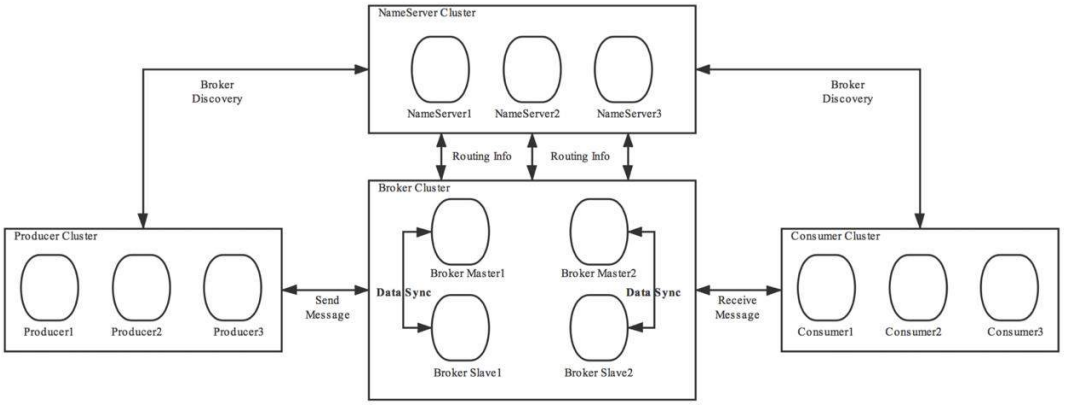
一、相关背景知识

1. RocketMQ

2012~2013 年期间，阿里巴巴中间件团队自主研发并对外开源了第三代分布式消息引擎 RocketMQ，其高性能、低延迟、抗堆积的特性稳定支撑了阿里巴巴 双 11 万亿级数据洪峰业务，其云产品 Aliware MQ 在微服务、流计算、IoT、异步解耦、数据同步等无数工况场景大放异彩。

2016 年，阿里巴巴向 Apache 软件基金会捐赠了 RocketMQ。次年，RocketMQ 顺利从基金会毕业，成为 Apache 顶级开源项目，与 Apache Hadoop，Apache Spark 一起为全球分布式、大数据领域的开发者带来福音。然而，在云原生时代的今天，RocketMQ 作为有状态的分布式服务系统，如何在大规模集群上做到极简运维，则是一个极具挑战和价值的问题。

RocketMQ 支持多种部署方式，以基本的双主双从架构为例，如下图所示。



RocketMQ 双主双从架构

这里面包括了一共 7 个 RocketMQ 服务实例：3 个 name server 实例，2 个 master broker 实例，以及 2 个 slave broker 实例。

传统的部署方式需要手动或编写脚本在每个节点上进行环境和文件配置。此外，随着用户业务的增加，存在对集群进行无缝扩容等需求。传统方式是运维人员访问不同节点，依赖操作手册和脚本按步骤进行操作来完成，耗费人力，且存在误操作的可能。一些公司可能会使用一些平台和工具如 Ansible 来帮助自动化运维，此外越来越多的公司开始集成和使用基于 Kubernetes 的云原生生态。

使用 Kubernetes 提供的 Deployment 和 StatefulSet 等原生资源可以很好地解决无状态应用的管理问题，但对于数据库和 RocketMQ 这类有状态应用，则存在很多局限性。例如对 RocketMQ 来说扩容不仅仅是拉起新的实例 Pod 就完成了，还需要同步复制 Broker 的状态信息包括 Topic 信息和订阅关系这些元数据，同时要正确配置新 Broker 的 config 参数，包括 brokerName 和 NameServer IP List 等，才能使得新扩容的 Broker 可用，而这些仅仅靠用户编写 StatefulSet，修改 size 或 replicas 然后 apply 是无法做到的。

实际上 Kubernetes 开发人员也发现了这些问题，因此引入了自定义资源和控制器的概念，让开发人员可以直接用 Go 语言调用 Kubernetes API，编写自定义资源和对应的控制器逻辑来解决复杂有状态应用的管理问题，提供特定应用相关的自定义资源的这类代码组件称之为 Operator。由具备 RocketMQ 领域知识的专家编写 Operator，屏蔽了应用领域的专业知识，让用户只需要关心和定义希望达到的集群终态，这也是 Kubernetes 声明式 API 的设计哲学。

2. Kubernetes Operator

Operator 是在 Kubernetes 基础上通过扩展 Kubernetes API，用来创建、配置和管理复杂的有状态应用，如分布式数据库等。Operator 基于 Kubernetes 1.7 版本以来引入的自定义控制器的概念，在自定义资源和控制器之上构建，同时又包含了应用程序特定的领域知识。实现一个 Operator 的关键是 CRD（自定义资源）和 Controller（控制器）的设计。

Operator 站在 Kubernetes 内部视角，为应用的云原生化打开了新世界的大门。自定义资源可以让开发人员扩展添加新功能，更新现有的功能，并且可以自动执行一些管理任务，这些自定义的控制器就像 Kubernetes 原生的组件一样，Operator 可以直接使用 Kubernetes API 进行开发，也就是说他们可以根据这些控制器编写的自定义规则来创建和更改 Pods / Services、对正在运行的应用进行扩缩容。

二、快速开始

本文使用 RocketMQ Operator 0.2.1 版本，展示如何使用 RocketMQ Operator 在 Kubernetes 上快速创建部署一个 RocketMQ 服务集群。

- 准备好 K8s 环境，可以使用 docker desktop 自带的 K8s，或者 minikube；
- 克隆 rocketmq-operator 仓库到你的 K8s 节点上；

```
$ git clone rocketmq-operatorhttps://github.com/apache/rocketmq-operator.git$cd
```

- 运行脚本安装 RocketMQ Operator：

```
$ ./install-operator.sh
```

- 检查下 RocketMQ Operator 是否安装成功：

```
$ kubectl get pods
NAME                                READY   STATUS    RESTARTS   AGE
rocketmq-operator-564b5d75d-jllzk   1/1     Running   0          108s
```

成功安装时，rocketmq-operator pod 处于类似上面例子的 running 状态。

- 应用 Broker 和 NameService 自定义资源，创建 RocketMQ 集群；

应用 rocketmq-operator / example 中的 rocketmq_v1alpha1_rocketmq_cluster.yaml 文件，快速部署一个 RocketMQ 集群。

rocketmq_v1alpha1_rocketmq_cluster.yaml 文件内容如下：

```
apiVersion: rocketmq.apache.org/v1alpha1
kind: Broker
metadata:
 # name of broker cluster
  name: broker
spec:
  # size is the number of the broker cluster, each broker cluster contains a master bro
ker and [replicaPerGroup] replica brokers.
  size: 1
  # nameServers is the [ip:port] list of name service
  nameServers: ""
  # replicationMode is the broker replica sync mode, can be ASYNC or SYNC
  replicationMode: ASYNC
  # replicaPerGroup is the number of each broker cluster
  replicaPerGroup: 1
  # brokerImage is the customized docker image repo of the RocketMQ broker
  brokerImage: apacherocketmq/rocketmq-broker:4.5.0-alpine
  # imagePullPolicy is the image pull policy
  imagePullPolicy: Always
  # resources describes the compute resource requirements and limits
  resources:
  requests:
  memory: "2048Mi"
  cpu: "250m"
  limits:
  memory: "12288Mi"
```

```
cpu: "500m"
# allowRestart defines whether allow pod restart
allowRestart: true
# storageMode can be EmptyDir, HostPath, StorageClass
storageMode: EmptyDir
# hostPath is the local path to store data
hostPath: /data/rocketmq/broker
# scalePodName is broker-[broker group number]-master-0
scalePodName: broker-0-master-0
# volumeClaimTemplates defines the storageClass
volumeClaimTemplates:
- metadata:
name: broker-storage
spec:
 accessModes:
 - ReadWriteOnce
storageClassName: rocketmq-storage
resources:
requests:
storage: 8Gi
---
apiVersion: rocketmq.apache.org/v1alpha1
kind: NameService
metadata:
name: name-service
spec:
 # size is the the name service instance number of the name service cluster
size: 1
```

```
    # nameServiceImage is the customized docker image repo of the RocketMQ name service
    nameServiceImage: apacherocketmq/rocketmq-nameserver:4.5.0-alpine
    # imagePullPolicy is the image pull policy
    imagePullPolicy: Always
    # hostNetwork can be true or false
    hostNetwork: true
    #  Set DNS policy for the pod.
    #  Defaults to "ClusterFirst".
    #  Valid values are 'ClusterFirstWithHostNet', 'ClusterFirst', 'Default' or 'None'.
    #  DNS parameters given in DNSConfig will be merged with the policy selected with DNSPolicy.
    #  To have DNS options set along with hostNetwork, you have to specify DNS policy

    #  explicitly to 'ClusterFirstWithHostNet'.
    dnsPolicy: ClusterFirstWithHostNet
    # resources describes the compute resource requirements and limits
    resources:
    requests:
    memory: "512Mi"
    cpu: "250m"
    limits:
    memory: "1024Mi"
    cpu: "500m"
     # storageMode can be EmptyDir, HostPath, StorageClass
    storageMode: EmptyDir
    # hostPath is the local path to store data
    hostPath: /data/rocketmq/nameserver
```

```
  # volumeClaimTemplates defines the storageClass
  volumeClaimTemplates:
  - metadata:
    name: namesrv-storage
    spec:
      accessModes:
      - ReadWriteOnce
      storageClassName: rocketmq-storage
      resources:
        requests:
          storage: 1Gi
```

注意到这个例子中 storageMode: EmptyDir，表示存储使用的是 EmptyDir，数据会随着 Pod 的删除而抹去，因此该方式仅供开发测试时使用。一般使用 HostPath 或 StorageClass 来对数据进行持久化存储。使用 HostPath 时，需要配置 hostPath，声明宿主机上挂载的目录。使用 storageClass 时，需要配置 volumeClaimTemplates，声明 PVC 模版。具体可参考 RocketMQ Operator 文档。

应用上面的 yaml 文件，输入命令：

```
$ kubectl apply -f example/rocketmq_v1alpha1_rocketmq_cluster.yaml
broker.rocketmq.apache.org/broker created
nameservice.rocketmq.apache.org/name-service created
```

查看集群 Pod 状态：

```
$ kubectl get pods -owide

NAME READY STATUS RESTARTS AGE IP NODE NOMINATED NODE
READINESS GATES

broker-0-master-0 1/1 Running 0 27s 10.1.2.27 docker-desktop <none>
<none>

broker-0-replica-1-0 1/1 Running 0 27s 10.1.2.28 docker-desktop <none>
<none>

name-service-0 1/1 Running 0 27s 192.168.65.3 docker-desktop <none>
<none>
rocketmq-operator-76b4b9f4db-x52mz 1/1 Running 0 3h25m 10.1.2.17
docker-desktop <none> <none>
```

使用默认的 rocketmq_v1alpha1_rocketmq_cluster.yaml 文件配置，我们看到集群中拉起了 1 个 name server 服务（name-service-0）和 2 个 broker 服务（1 主 1 从）。

好啦！到这里你已经成功通过 Operator 提供的自定义资源部署了一个 RocketMQ 服务集群。

- 访问这个 RocketMQ 集群中的 Pod 来验证集群是否能正常工作；

使用 RocketMQ 的 tools.sh 脚本运行 Producer example：

```
$ kubectl exec -it broker-0-master-0 bash
bash-4.4# sh ./tools.sh org.apache.rocketmq.example.quickstart.Producer
OpenJDK 64-Bit Server VM warning: ignoring option PermSize=128m; support was removed in 8.0
OpenJDK 64-Bit Server VM warning: ignoring option MaxPermSize=128m; support was removed in 8.0
06:56:29.145 [main] DEBUG i.n.u.i.l.InternalLoggerFactory - Using SLF4J as the default logging framework
SendResult [sendStatus=SEND_OK, msgId=0A0102CF007778308DB1206383920000, offsetMsgId=0A0102CF00002A9F0000000000000000, messageQueue=MessageQueue [topic=TopicTest, brokerName=broker-0, queueId=0], queueOffset=0]
...
06:56:51.120 [NettyClientSelector_1] INFO  RocketmqRemoting - closeChannel: close the connection to remote address[10.1.2.207:10909] result: true
bash-4.4#
```

在另一个节点上运行 Consumer example：

```
$ kubectl exec -it name-service-0 bash
bash-4.4# sh ./tools.sh org.apache.rocketmq.example.quickstart.Consumer
OpenJDK 64-Bit Server VM warning: ignoring option PermSize=128m; support was removed in 8.0
OpenJDK 64-Bit Server VM warning: ignoring option MaxPermSize=128m; support was removed in 8.0
07:01:32.077 [main] DEBUG i.n.u.i.l.InternalLoggerFactory - Using SLF4J as the default logging framework
Consumer Started.
```

```
ConsumeMessageThread_1 Receive New Messages: [MessageExt [queueId=0, storeSize=27
3, queueOffset=19845, sysFlag=0, bornTimestamp=1596768410268, bornHost=/30.4.165.204:534
50, storeTimestamp=1596768410282, storeHost=/100.81.180.84:10911, msgId=6451B45400002A9
F000014F96A0D6C65, commitLogOffset=23061458676837, bodyCRC=532471758, reconsumeTime
s=0, preparedTransactionOffset=0, toString()=Message{topic='TopicTest', flag=0, properties={MI
N_OFFSET=19844, TRACE_ON=true, eagleTraceId=1e04a5cc15967684102641001d0db0, MAX_OFF
SET=19848, MSG_REGION=DefaultRegion, CONSUME_START_TIME=1596783715858, UNIQ_KEY=1
E04A5CC0DB0135FBAA421365A5F0000, WAIT=true, TAGS=TagA, eagleRpcId=9.1}, body=[72, 101,
 108, 108, 111, 32, 77, 101, 116, 97, 81, 32, 48], transactionId='null'}]]

ConsumeMessageThread_4 Receive New Messages: [MessageExt [queueId=1, storeSize=27
3, queueOffset=19637, sysFlag=0, bornTimestamp=1596768410296, bornHost=/30.4.165.204:534
50, storeTimestamp=1596768410298, storeHost=/100.81.180.84:10911, msgId=6451B45400002A9
F000014F96A0D7141, commitLogOffset=23061458678081, bodyCRC=1757146968, reconsumeTim
es=0, preparedTransactionOffset=0, toString()=Message{topic='TopicTest', flag=0, properties={MI
N_OFFSET=19636, TRACE_ON=true, eagleTraceId=1e04a5cc15967684102961002d0db0, MAX_OFF
SET=19638, MSG_REGION=DefaultRegion, CONSUME_START_TIME=1596783715858, UNIQ_KEY=1
E04A5CC0DB0135FBAA421365AB80001, WAIT=true, TAGS=TagA, eagleRpcId=9.1}, body=[72, 101,
 108, 108, 111, 32, 77, 101, 116, 97, 81, 32, 49], transactionId='null'}]]
...
```

- 删除集群，清理环境；

清除 RocketMQ 服务集群实例：

```
$ kubectl delete -f example/rocketmq_v1alpha1_rocketmq_cluster.yaml
```

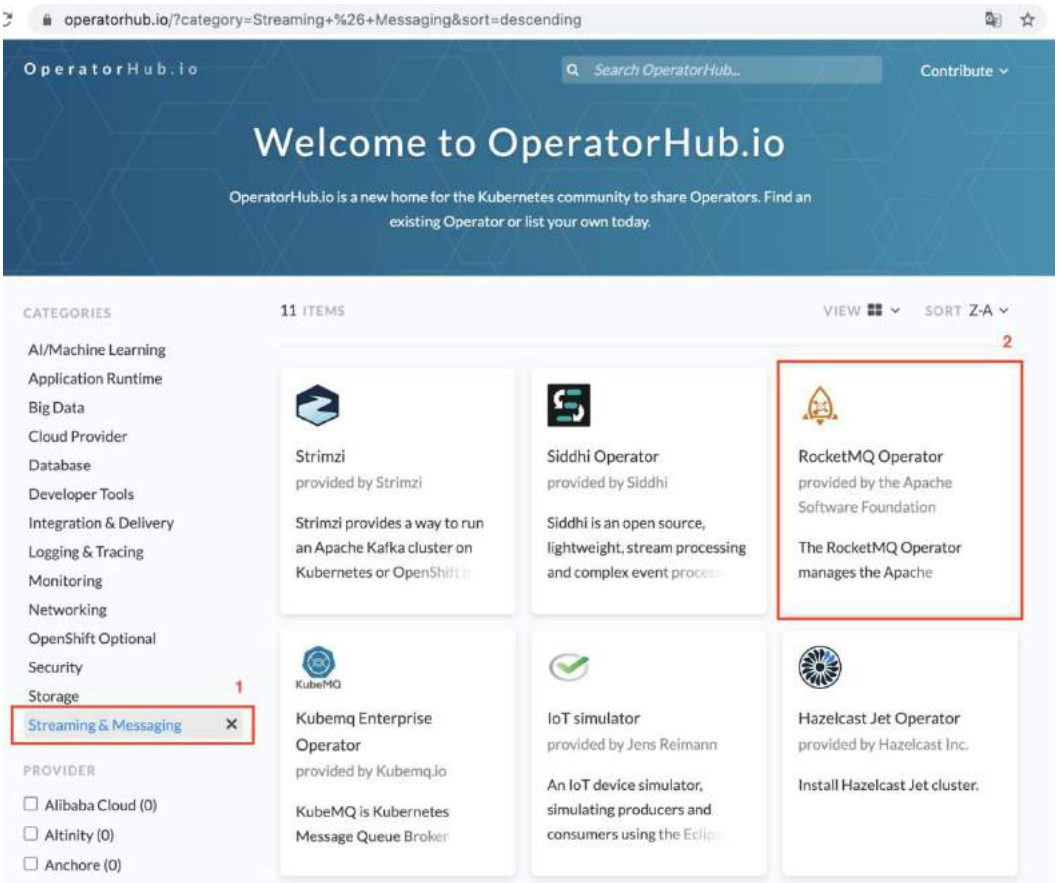
清除 RocketMQ Operator：

```
$ ./purge-operator.sh
```

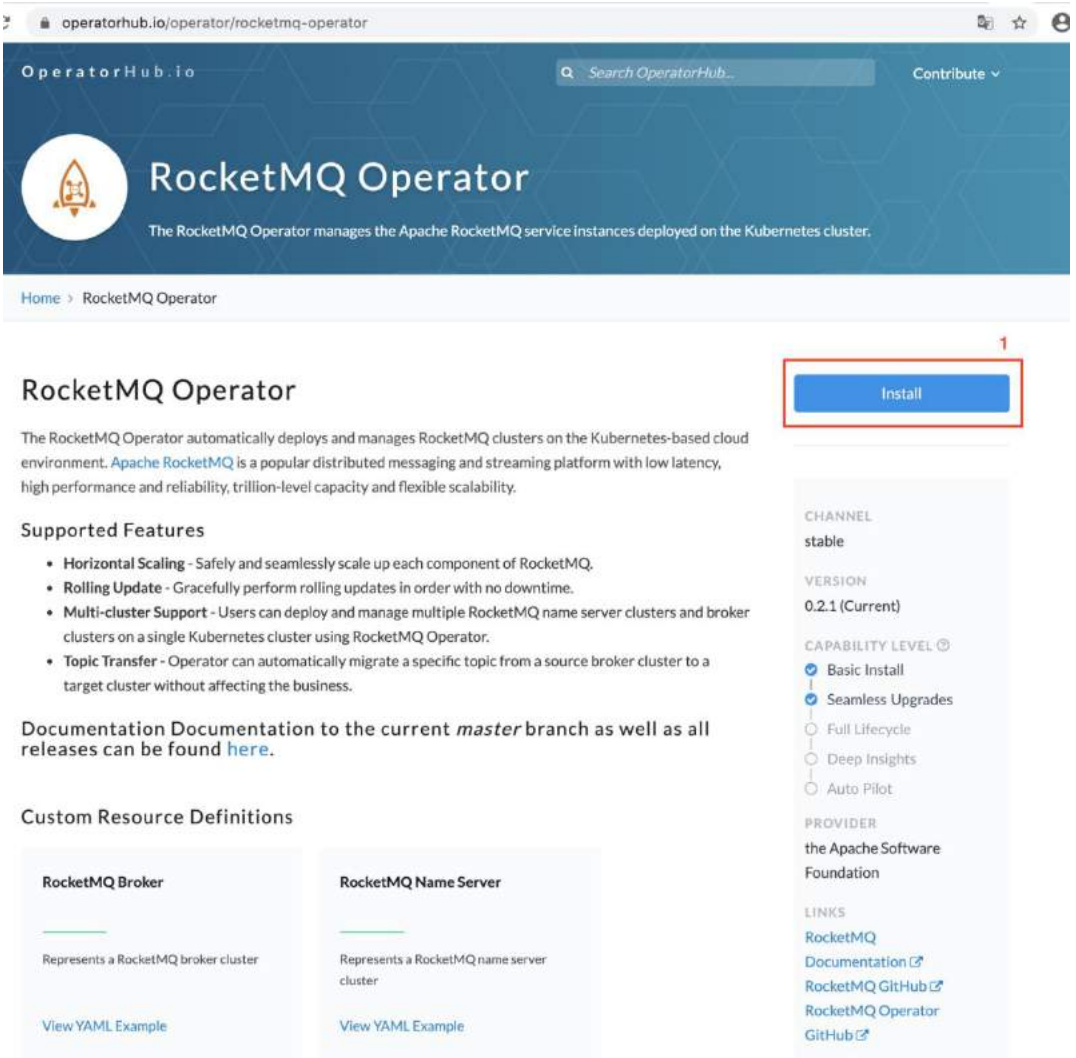
三、按照 OperatorHub 官网指导安装 RocketMQ Operator

- 在 OperatorHub.io 网页搜索 RocketMQ Operator；

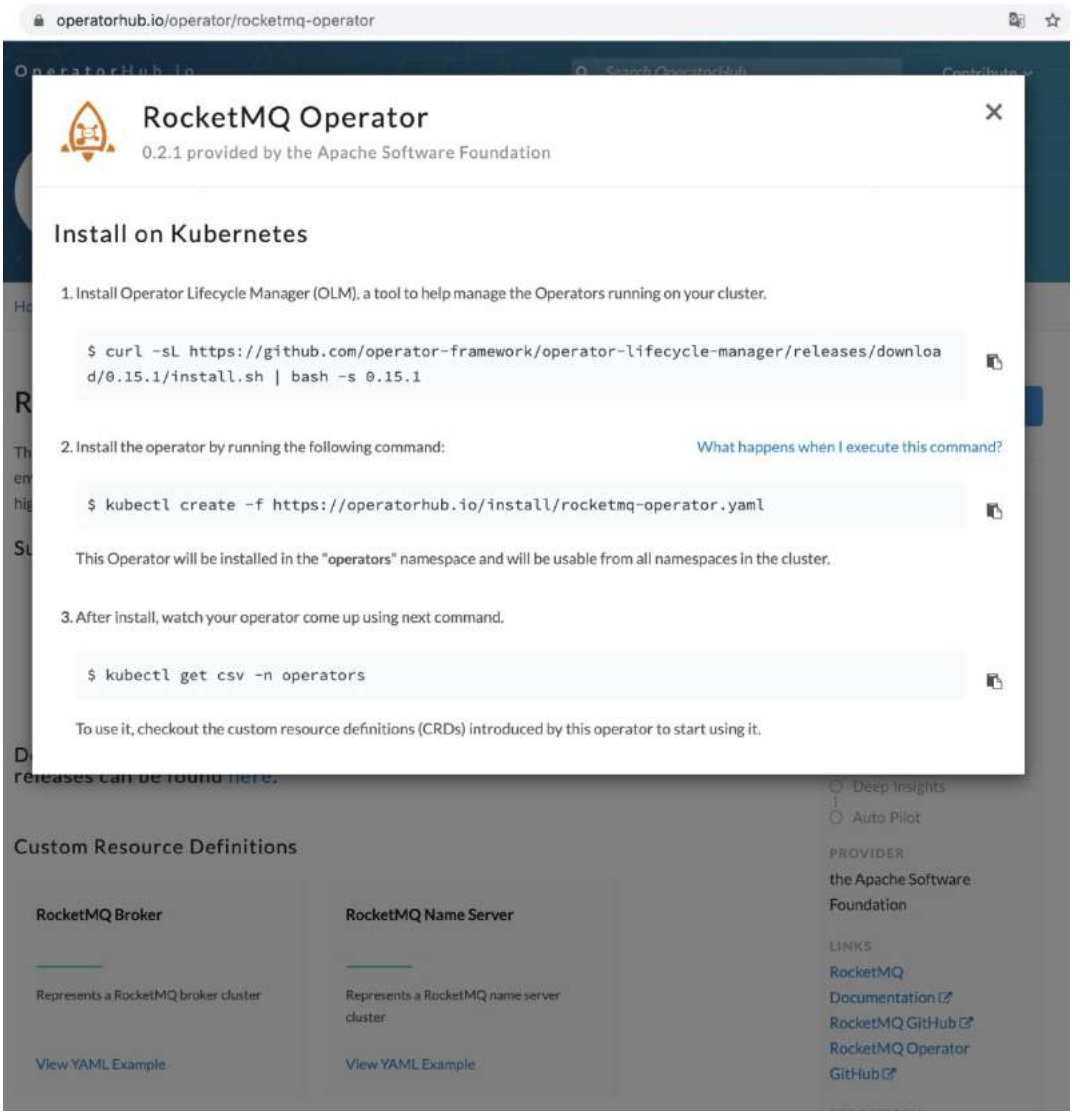
选择 Streaming & Messaging 类别，点击 RocketMQ Operator：



- 进入 RocketMQ Operator 页面，点击 Install 按钮：



- 按照说明安装 OLM 和 RocketMQ Operator：



四、本地安装 OLM 来使用 RocketMQ Operator

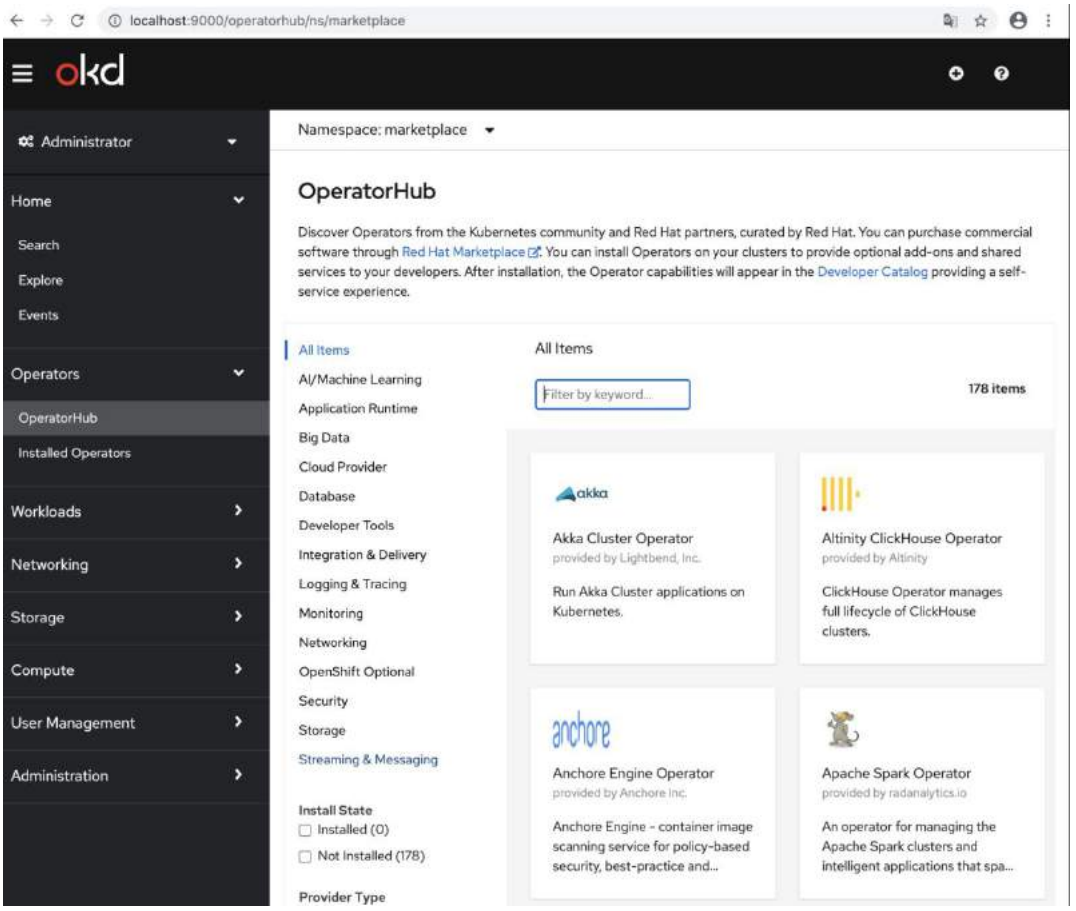
- 本地安装和动 OLM(Operator Lifecycle Manager) console：

参考：OLM 安装文档。

- 本地启动 UI 界面控制台：

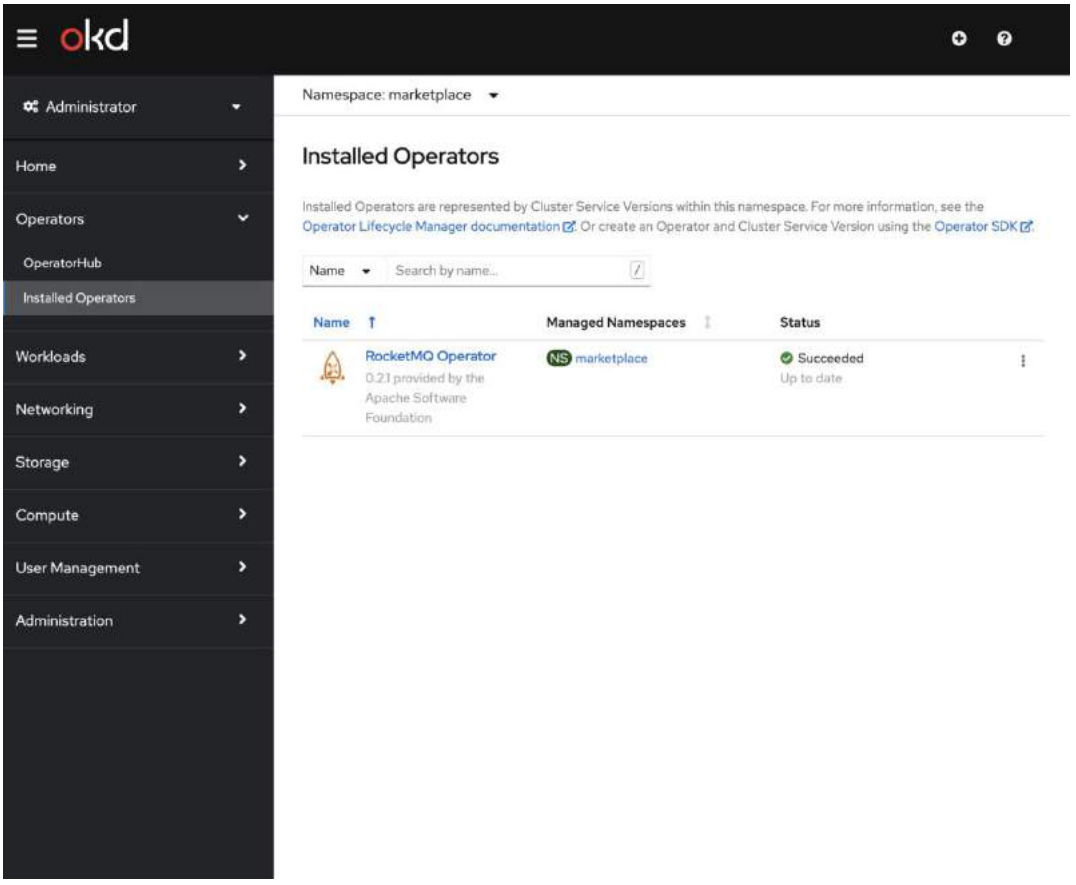
```
$ make run-console-local
```

- 访问 http://localhost:9000 查看控制台：

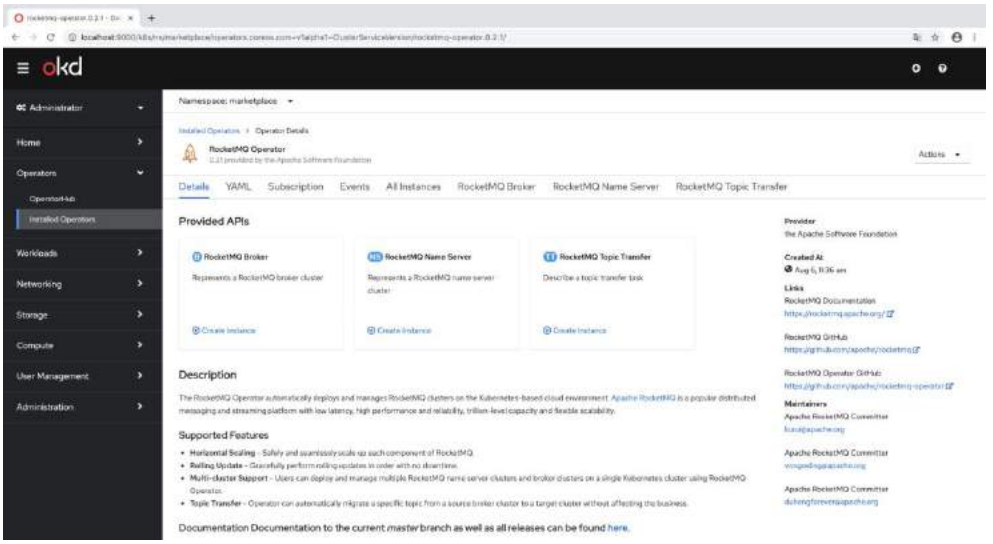


OperatorHub

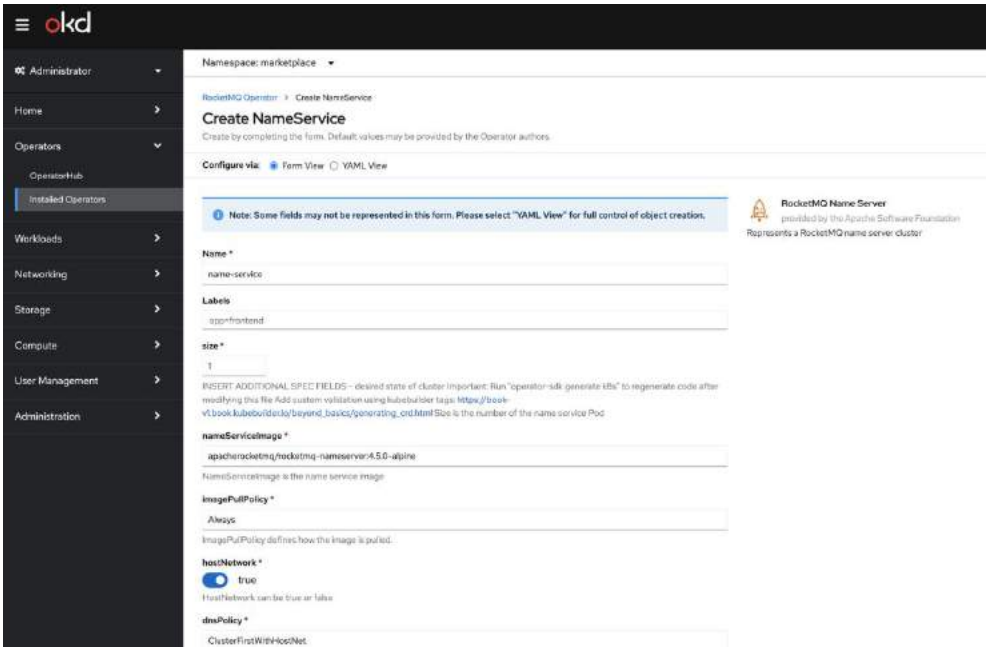
- 搜索 RocketMQ 或点击 All Items 分类中的 Streaming & Messaging，找到 RocketMQ Operator 并进行安装；
- 安装完 RocketMQ Operator 后可以在 Installed Operators 中找到 RocketMQ Operator；



已安装的 Operators 界面



RocketMQ Operator 介绍界面



通过 UI 界面创建 NameService 自定义资源

可以在 UI 中创建指定 Namespace 下的 NameService 和 Broker 实例，并对已创建的实例进行浏览和管理。我们也可以通过命令查看当前 K8s 集群中的 Pod 状态，例如：

```
$ kubectl get pods -A
NAMESPACE        NAME                                READY   STATUS    RESTARTS   AGE
docker    compose-78f95d4f8c-8fr5z            1/1     Running   0          32h
docker    compose-api-6ffb89dc58-nv9rh        1/1     Running   0          32h
kube-system    coredns-5644d7b6d9-hv6r5       1/1     Running   0          32h
kube-system    coredns-5644d7b6d9-mkqb6       1/1     Running   0          32h
kube-system    etcd-docker-desktop            1/1     Running   0          32h
kube-system    kube-apiserver-docker-desktop  1/1     Running   0          32h
kube-system    kube-controller-manager-docker-desktop  1/1  Running  1     32h
kube-system    kube-proxy-snmxh               1/1     Running   0          32h
kube-system    kube-scheduler-docker-desktop  1/1     Running   1          32h
kube-system    storage-provisioner            1/1     Running   1          32h
kube-system    vpnkit-controller              1/1     Running   0          32h
marketplace    broker-0-master-0              1/1     Running   0          5h3m
marketplace    broker-0-replica-1-0           1/1     Running   0          5h3m
marketplace    name-service-0                 1/1     Running   0          5h3m
marketplace    marketplace-operator-69756457d8-42chk   1/1   Running   0   32h
marketplace    rocketmq-operator-0.2.1-c9fffb5f-cztcl  1/1   Running   0    32h
marketplace    rocketmq-operator-84c7bb4ddc-7rvqr  1/1  Running   0        32h
marketplace    upstream-community-operators-5b79db455f-7t47w   1/1   Running   1
 32holm
catalog-operator-7b788c597d-gjz55      1/1     Running   0          32h
olm            olm-operator-946bd977f-dhszg    1/1   Running   0          32h
olm            operatorhubio-catalog-fvxp9     1/1     Running   0          32h
```

```
olm          packageserver-789c7b448b-7ss7m   1/1     Running   0          32h
olm          packageserver-789c7b448b-lfxrw   1/1     Running   0          32h
```

可以看到在 marketplace 这个 namespace 中也成功创建了对应的 name server 和 broker 实例。

以上是基于 OperatorHub 和 OLM 安装使用 RocketMQ Operator 的案例，我们将持续推送和维护新版本的 RocketMQ Operator 至该平台，方便用户获取最新更新或选择合适的 Operator 版本。

五、社区

RocketMQ Operator 是 Apache 社区的开源项目，服务于阿里巴巴 SaaS 类交付专有云，产品私有云环境部署等场景，同时也收到来自爱奇艺等互联网公司开源贡献者的代码提交。欢迎广大用户来社区项目中进行反馈，点击下方链接留下您的信息，让我们更好地完善 RocketMQ Operator。

链接：https://github.com/apache/rocketmq-operator/issues/20

目前，RocketMQ Operator v0.2.1 的 PR 已合并进入 community-operators 仓库，RocketMQ Operator 进入 OperatorHub.io 后，用户可以通过使用 OLM(Operator Lifecycle Manager) 来安装、订阅 RocketMQ Operator，获得持续的服务支持。

六、未来展望

RocketMQ Operator v0.2.1 支持的功能主要包括：Name Server 和 Broker 集群的自动创建，Name Server 集群的无缝扩容（自动通知 Broker 集群更新 Name Server IP 列表），非顺序消息下的 Broker 集群无缝扩容（新 Broker 实例会从 Broker CRD 指定的源 Broker Pod 中同步元数据，包括 Topic 信息和订阅信息），以及 Topic 迁移等。

下一步我们希望和社区一起进一步完善 RocketMQ Operator 项目，包括灰度发布，数据的全生命周期管理，容灾备份恢复，流量等指标监控、自动弹性扩缩容等方面，最终实现通过 Operator 可以覆盖 RocketMQ 服务全生命周期的管理。

欢迎大家使用 RocketMQ Operator，提出宝贵建议。

七、相关链接

- RocketMQ Operator 项目：https://github.com/apache/rocketmq-operator
- OperatorHub：https://operatorhub.io/
- Operator Framework：https://github.com/operator-framework
- RocketMQ 官网：https://rocketmq.apache.org/
- Apache RocketMQ 仓库：https://github.com/apache/rocketmq

基于 RocketMQ Prometheus Exporter 打造定制化 DevOps 平台

作者 | 陈厚道、冯庆

PC 端登录 start.aliyun.com 即在浏览器中体验 RocketMQ 在线可交互教程

作者介绍：陈厚道，曾就职于腾讯、盛大、斗鱼等互联网公司。目前就职于尚德机构，在尚德机构负责基础架构方面的设计和开发工作。对分布式消息队列、微服务架构和落地、DevOps 和监控平台有比较深入的研究。

冯庆，曾就职于华为。目前就职于尚德机构，在尚德机构基础架构团队负责基础组件的开发工作。

本文将对 RocketMQ-Exporter 的设计实现做一个简单的介绍，读者可以通过本文了解到 RocketMQ-Exporter 的实现过程，以及通过 RocketMQ-Exporter 来搭建自己的 RocketMQ 监控系统。

该项目的 git 地址 https://github.com/apache/rocketmq-exporter

文章主要内容包含以下几个方面：

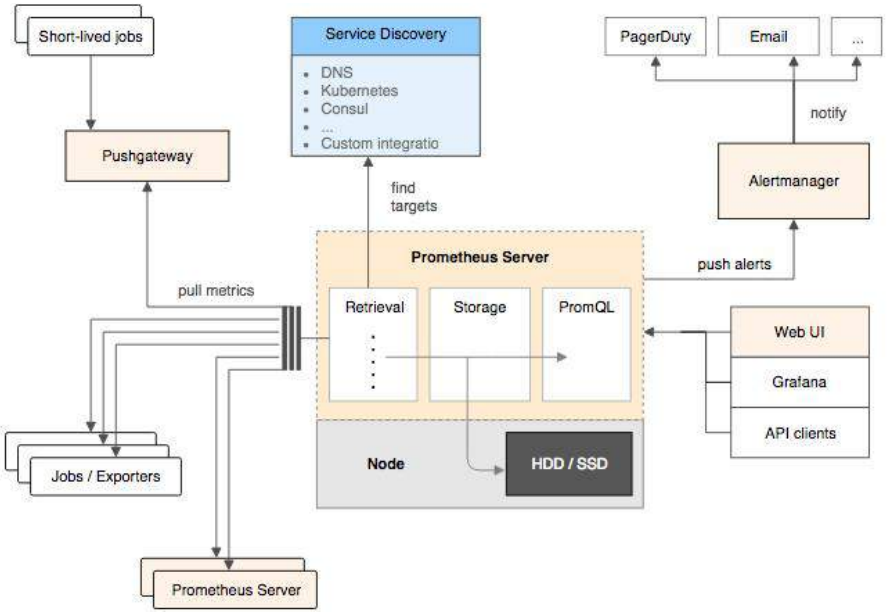
- RocketMQ 介绍
- Prometheus 简介
- RocketMQ-Exporter 的具体实现
- RocketMQ-Exporter 的监控指标和告警指标
- RocketMQ-Exporter 使用示例

一、RocketMQ 介绍

RocketMQ 是一个分布式消息和流数据平台，具有低延迟、高性能、高可靠性、万亿级容量和灵活的可扩展性。简单的来说，它由 Broker 服务器和客户端两部分组成，其中客户端一个是消息发布者客户端 (Producer)，它负责向 Broker 服务器发送消息；另外一个是消息的消费者客户端 (Consumer)，多个消费者可以组成一个消费组，来订阅和拉取消费 Broker 服务器上存储的消息。正由于它具有高性能、高可靠性和高实时性的特点，与其他协议组件在 MQTT 等各种消息场景中的结合也越来越多，应用越来越广泛。而对于这样一个强大的消息中间件平台，在实际使用的时候还缺少一个监控管理平台。而当前在开源界，使用最广泛监控解决方案的就是 Prometheus。与其它传统监控系统相比较，Prometheus 具有易于管理，监控服务的内部运行状态，强大的数据模型，强大的查询语言 PromQL，高效的数据处理，可扩展，易于集成，可视化，开放性等优点。并且借助于 Prometheus 可以很快速的构建出一个能够监控 RocketMQ 的监控平台。

二、Prometheus 简介

下图展示了 Prometheus 的基本架构：

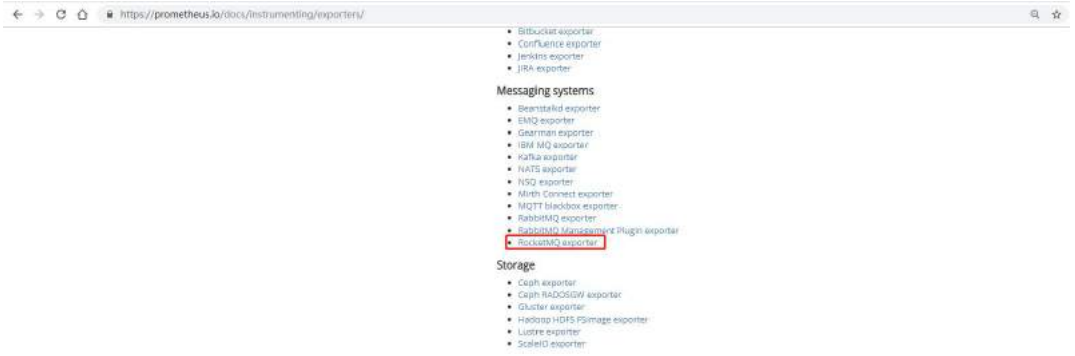


1. Prometheus Server

Prometheus Server 是 Prometheus 组件中的核心部分，负责实现对监控数据的获取，存储以及查询。 Prometheus Server 可以通过静态配置管理监控目标，也可以配合使用 Service Discovery 的方式动态管理监控目标，并从这些监控目标中获取数据。其次 Prometheus Server 需要对采集到的监控数据进行存储，Prometheus Server 本身就是一个时序数据库，将采集到的监控数据按照时间序列的方式存储在本地磁盘当中。最后 Prometheus Server 对外提供了自定义的 PromQL 语言，实现对数据的查询以及分析。

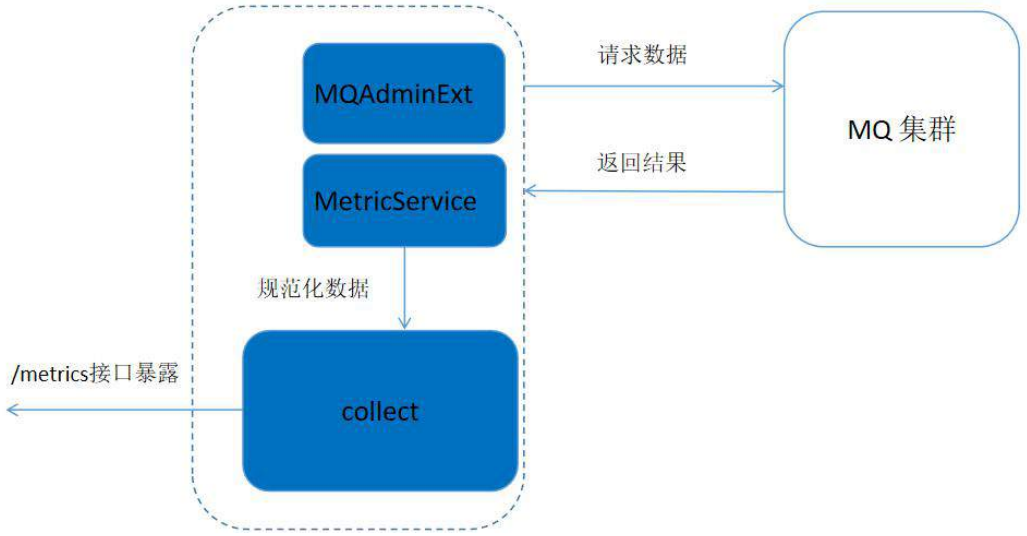
2. Exporters

Exporter 将监控数据采集的端点通过 HTTP 服务的形式暴露给 Prometheus Server，Prometheus Server 通过访问该 Exporter 提供的 Endpoint 端点，即可获取到需要采集的监控数据。RocketMQ-Exporter 就是这样一个 Exporter，它首先从 RocketMQ 集群采集数据，然后借助 Prometheus 提供的第三方客户端库将采集的数据规范化成符合 Prometheus 系统要求的数据，Prometheus 定时去从 Exporter 拉取数据即可。当前 RocketMQ Exporter 已被 Prometheus 官方收录，其地址为 https://github.com/apache/rocketmq-exporter



三、RocketMQ-Exporter 的具体实现

当前在 Exporter 当中，实现原理如下图所示：



整个系统基于 spring boot 框架来实现。由于 MQ 内部本身提供了比较全面的数据统计信息，所以对于 Exporter 而言，只需要将 MQ 集群提供的统计信息取出然后进行加工而已。所以 RocketMQ-Exporter 的基本逻辑是内部启动多个定时任务周期性的从 MQ 集群拉取数据，然后将数据规范化后通过端点暴露给 Prometheus 即可。其中主要包含如下主要的三个功能部分：

- MQAdminExt 模块通过封装 MQ 系统客户端提供的接口来获取 MQ 集群内部的统计信息。
- MetricService 负责将 MQ 集群返回的结果数据进行加工，使其符合 Prometheus 要求的格式化数据。
- Collect 模块负责存储规范化后的数据，最后当 Prometheus 定时从 Exporter 拉取数据的时候，Exporter 就将 Collector 收集的数据通过 HTTP 的形式在 /metrics 端点进行暴露。

四、RocketMQ-Exporter 的监控指标和告警指标

RocketMQ-Exporter 主要是配合 Prometheus 来做监控，下面来看看当前在 Expoter 中定义了哪些监控指标和告警指标。

- 监控指标

监控指标	含义
rocketmq_broker_tps	broker 每秒生产消息数量
rocketmq_broker_qps	broker 每秒消费消息数量
rocketmq_producer_tps	某个 topic 每秒生产的消息数量
rocketmq_producer_put_size	某个 topic 每秒生产的消息大小(字节)
rocketmq_producer_offset	某个 topic 的生产消息的进度
rocketmq_consumer_tps	某个消费组每秒消费的消息数量
rocketmq_consumer_get_size	某个消费组每秒消费的消息大小(字节)
rocketmq_consumer_offset	某个消费组的消费消息的进度
rocketmq_group_get_latency_by_storetime	某个消费组的消费延时时间
rocketmq_message_accumulation（rocketmq_producer_offset-rocketmq_consumer_offset）	消息堆积量（生产进度 - 消费进度）

rocketmq_message_accumulation 是一个聚合指标，需要根据其它上报指标聚合生成。

- 告警指标

告警指标	含义
sum(rocketmq_producer_tps) by (cluster) >= 10	集群发送 tps 太高
sum(rocketmq_producer_tps) by (cluster) < 1	集群发送 tps 太低
sum(rocketmq_consumer_tps) by (cluster) >= 10	集群消费 tps 太高
sum(rocketmq_consumer_tps) by (cluster) < 1	集群消费 tps 太低
rocketmq_group_get_latency_by_storetime > 1000	集群消费延时告警
rocketmq_message_accumulation > value	消费堆积告警

消费者堆积告警指标也是一个聚合指标，它根据消费堆积的聚合指标生成，value 这个阈值对每个消费者是不固定的，当前是根据过去 5 分钟生产者生产的消息数量来定，用户也可以根据实际情况自行设定该阈值。 告警指标设置的值只是个阈值只是象征性的值，用户可根据在实际使用 RocketMQ 的情况下自行设定。这里重点介绍一下消费者堆积告警指标，在以往的监控系统中，由于没有像 Prometheus 那样有强大的 PromQL 语言，在处理消费者告警问题时势必需要为每个消费者设置告警，那这样就需要 RocketMQ 系统的维护人员为每个消费者添加，要么在系统后台检测到有新的消费者创建时自动添加。在 Prometheus 中，这可以通过一条如下的语句来实现：

```
(sum(rocketmq_producer_offset) by (topic) - on(topic) group_right sum(rocketmq_consumer_offset) by (group,topic))
- ignoring(group) group_left sum (avg_over_time(rocketmq_producer_tps[5m])) by (topic) *5*60 > 0
```

借助 PromQL 这一条语句不仅可以实现为任意一个消费者创建消费告警堆积告警，而且还可以使消费堆积的阈值取一个跟生产者发送速度相关的阈值。这样大大增加了消费堆积告警的准确性。

五、RocketMQ-Exporter 使用示例

1. 启动 NameServer 和 Broker

要验证 RocketMQ 的 Spring-Boot 客户端，首先要确保 RocketMQ 服务正确的下载并启动。可以参考 RocketMQ 主站的快速开始来进行操作。确保启动 NameServer 和 Broker 已经正确启动。

2. 编译 RocketMQ-Exporter

用户当前使用，需要自行下载 git 源码编译：

```
git clone https://github.com/apache/rocketmq-exporter
cd rocketmq-exporter
mvn clean install
```

3. 配置和运行

RocketMQ-Exporter 有如下的运行选项：

选项	默认值	含义
rocketmq.config.namesrvAddr	127.0.0.1:9876	MQ 集群的 nameSrv 地址
rocketmq.config.webTelemetryPath	/metrics	指标搜集路径
server.port	5557	HTTP 服务暴露端口

以上的运行选项既可以在下载代码后在配置文件中更改，也可以通过命令行来设置。

编译出来的 jar 包就叫 rocketmq-exporter-0.0.1-SNAPSHOT.jar，可以通过如下的方式来运行。

```
java -jar rocketmq-exporter-0.0.1-SNAPSHOT.jar [--rocketmq.config.namesrvAddr="127.0.0.1:9876" ...]
```

4. 安装 Prometheus

首先到 Prometheus 官方下载地址 去下载 Prometheus 安装包，当前以 linux 系统安装为例，选择的安装包为 prometheus-2.7.0-rc.1.linux-amd64.tar.gz，经过如下的操作步骤就可以启动 prometheus 进程。

```
tar -xzf prometheus-2.7.0-rc.1.linux-amd64.tar.gz
cd prometheus-2.7.0-rc.1.linux-amd64/
./prometheus --config.file=prometheus.yml --web.listen-address=:5555
```

Prometheus 默认监听端口号为 9090，为了不与系统上的其它进程监听端口冲突，我们在启动参数里面重新设置了监听端口号为 5555。然后通过浏览器访问 http://< 服务器 IP 地址 >:5555, 就可以验证 Prometheus 是否已成功安装，显示界面如下：

由于 RocketMQ-Exporter 进程已启动，这个时候可以通过 Prometheus 来抓取 RocketMQ-Exporter 的数据，这个时候只需要更改 Prometheus 启动的配置文件即可

整体配置文件如下：

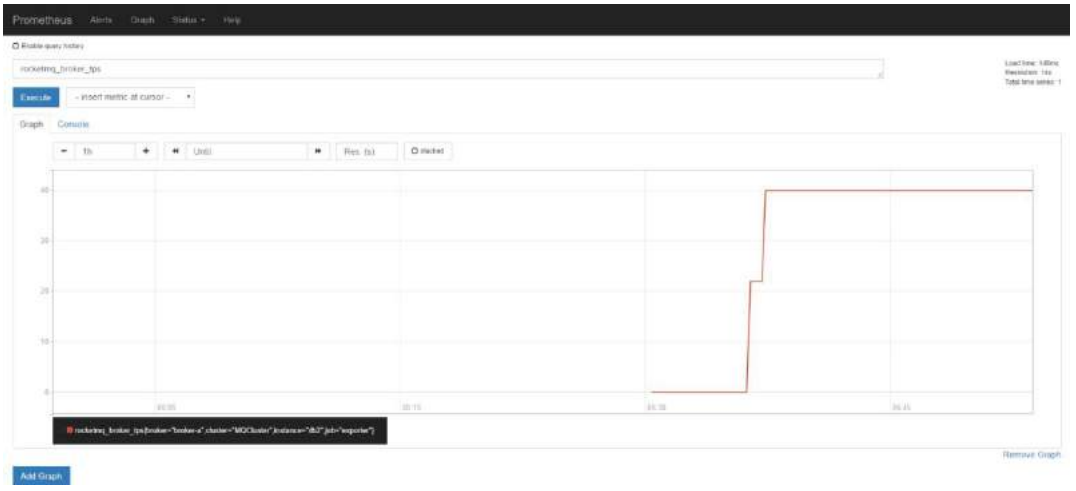
```
# my
global configglobal:
 scrape_interval:     15s # Set the scrape interval to every 15 seconds. Default is every
1 minute.
evaluation_interval: 15s # Evaluate rules every 15 seconds. The default is every 1 minut
e.
```

```
# scrape_timeout is set to the global default (10s).
# Load rules once and periodically evaluate them according to the global 'evaluation_interval'.
rule_files:
# - "first_rules.yml"
# - "second_rules.yml"

 scrape_configs:
- job_name: 'prometheus'
static_configs:
- targets: ['localhost:5555']

- job_name: 'exporter'
static_configs:
- targets: ['localhost:5557']
```

更改配置文件后，重启服务即可。重启后就可以在 Prometheus 界面查询 RocketMQ-Exporter 上报的指标，例如查询 rocketmq_broker_tps 指标，其结果如下



5. 告警规则添加

在 Prometheus 可以展示 RocketMQ-Exporter 的指标后，就可以在 Prometheus 中配置 RocketMQ 的告警指标了。在 Prometheus 的配置文件中添加如下的告警配置项，*.rules 表示可以匹配多个后缀为 rules 的文件。

```
rule_files:
# - "first_rules.yml"
# - "second_rules.yml"
- /home/prometheus/prometheus-2.7.0-rc.1.linux-amd64/rules/*.rules
```

当前设置的告警配置文件为 warn.rules，其文件具体内容如下所示。其中的阈值只起一个示例的作用，具体的阈值还需用户根据实际使用情况来自行设定。

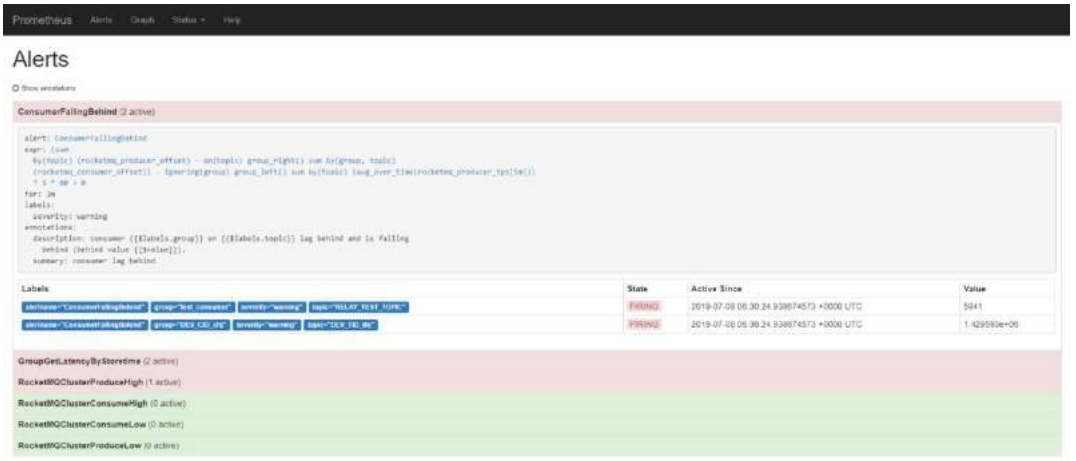
```
###
# Sample prometheus rules/alerts for rocketmq.
#
###
# Galera Alerts

groups:
- name: GaleraAlerts
rules:
- alert: RocketMQClusterProduceHigh
expr: sum(rocketmq_producer_tps) by (cluster) >= 10
for: 3m
labels:
```

```
severity: warning
annotations:
description: '{{$labels.cluster}} Sending tps too high.'
summary: cluster send tps too high
- alert: RocketMQClusterProduceLow
 expr: sum(rocketmq_producer_tps) by (cluster) < 1
for: 3m
labels:
 severity: warning
 annotations:
 description: '{{$labels.cluster}} Sending tps too low.'
summary: cluster send tps too low
- alert: RocketMQClusterConsumeHigh
expr: sum(rocketmq_consumer_tps) by (cluster) >= 10
 for: 3m
labels:
severity: warning
annotations:
description: '{{$labels.cluster}} consuming tps too high.'
summary: cluster consume tps too high
- alert: RocketMQClusterConsumeLow
expr: sum(rocketmq_consumer_tps) by (cluster) < 1
 for: 3m
 labels:
 severity: warning
 annotations:
description: '{{$labels.cluster}} consuming tps too low.'
summary: cluster consume tps too low
- alert: ConsumerFallingBehind
```

```
    expr: (sum(rocketmq_producer_offset) by (topic) - on(topic)  group_right
     sum(rocketmq_consumer_offset) by (group,topic)) - ignoring(group) group_left sum (avg
_over_time(rocketmq_producer_tps[5m])) by (topic)*5*60 > 0
    for: 3m
     labels:
    severity: warning
    annotations:
    description: 'consumer {{$labels.group}} on {{$labels.topic}} lag behind
    and is falling behind (behind value {{$value}}).'
    summary: consumer lag behind
     - alert: GroupGetLatencyByStoretime
    expr: rocketmq_group_get_latency_by_storetime > 1000
     for: 3m
     labels:
    severity: warning
    annotations:
    description: 'consumer {{$labels.group}} on {{$labels.broker}}, {{$labels.topic}} consume ti
me lag behind message store time
    and (behind value is {{$value}}).'
    summary: message consumes time lag behind message store time too much
```

最终，可以在 Prometheus 的看一下告警展示效果，红色表示当前处于告警状态的项，绿色表示正常状态。



6. Grafana dashboard for RocketMQ

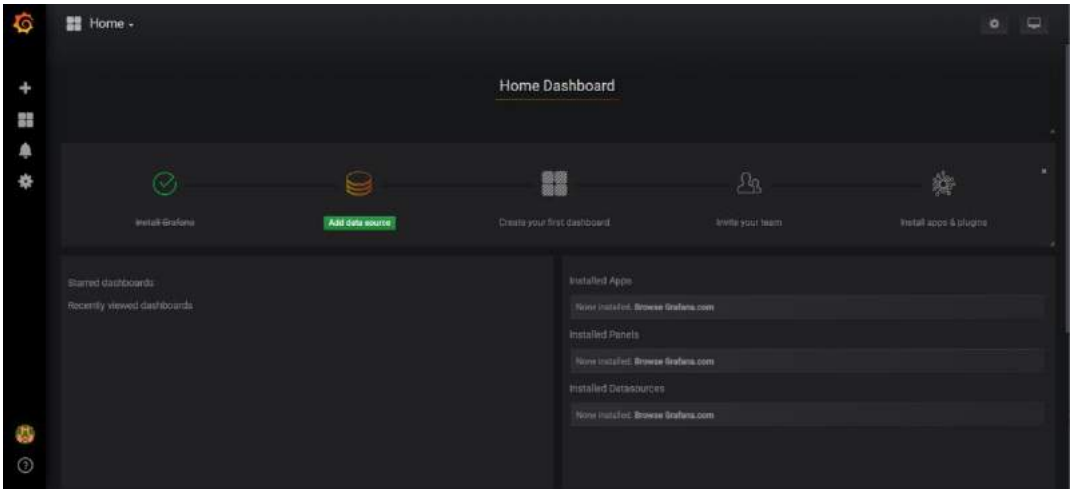
Prometheus 自身的指标展示平台没有当前流行的展示平台 Grafana 好， 为了更好的展示 RocketMQ 的指标，可以使用 Grafana 来展示 Prometheus 获取的指标。首先到官网去下载 https://grafana.com/grafana/download ，这里仍以二进制文件安装为例进行介绍。

```
wget https://dl.grafana.com/oss/release/grafana-6.2.5.linux-amd64.tar.gz
tar -zxvf grafana-6.2.5.linux-amd64.tar.gz
cd grafana-5.4.3/
```

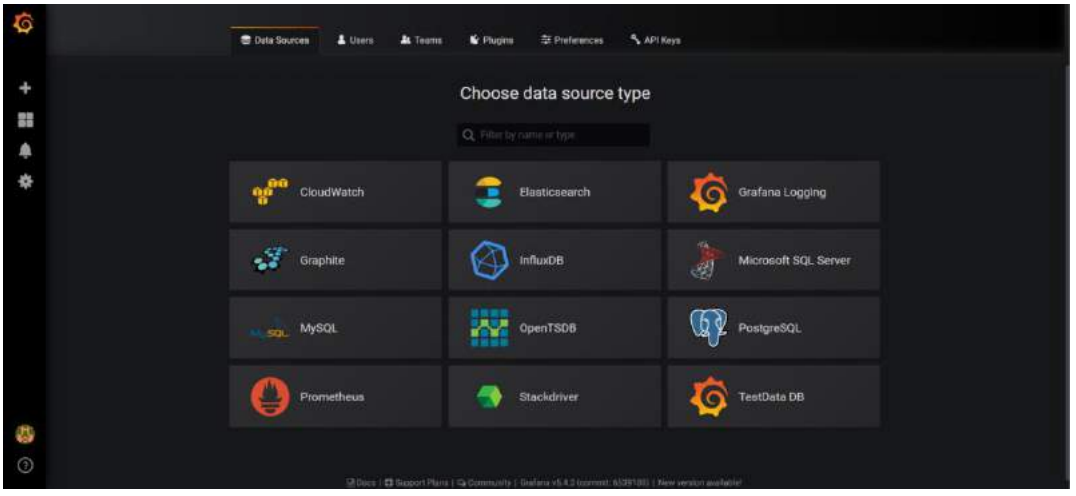
同样为了不与其它进程的使用端口冲突，可以修改 conf 目录下的 defaults.ini 文件的监听端口，当前将 grafana 的监听端口改为 55555，然后使用如下的命令启动即可

```
./bin/grafana-server web
```

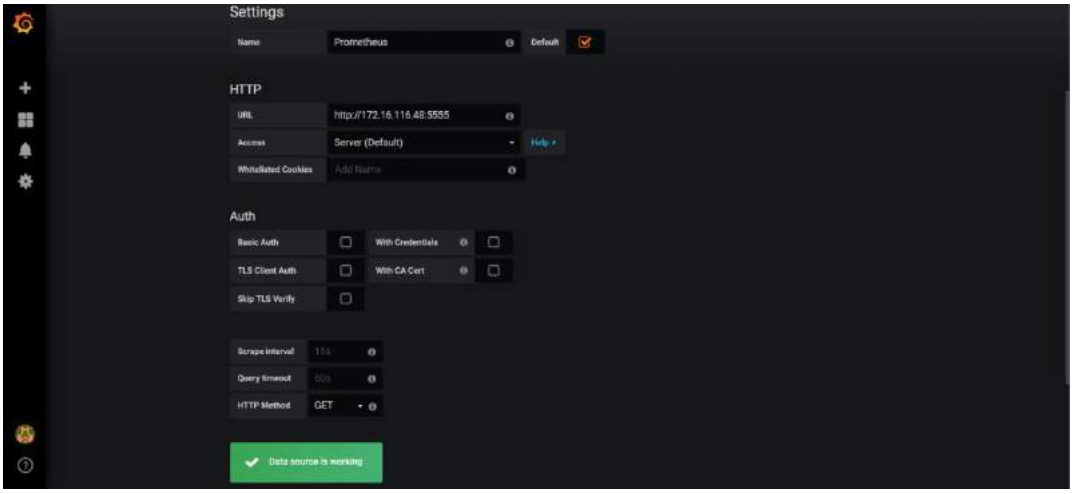
然后通过浏览器访问 http://< 服务器 IP 地址 >:55555，就可以验证 grafana 是否已成功安装。系统默认用户名和密码为 admin/admin，第一次登陆系统会要求修改密码，修改密码后登陆，界面显示如下：



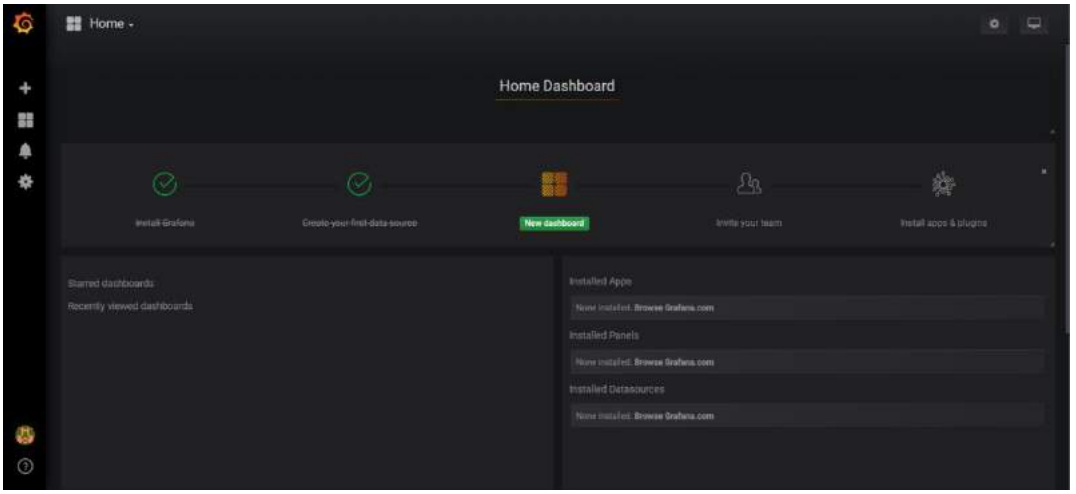
点击 Add data source 按钮，会要求选择数据源。



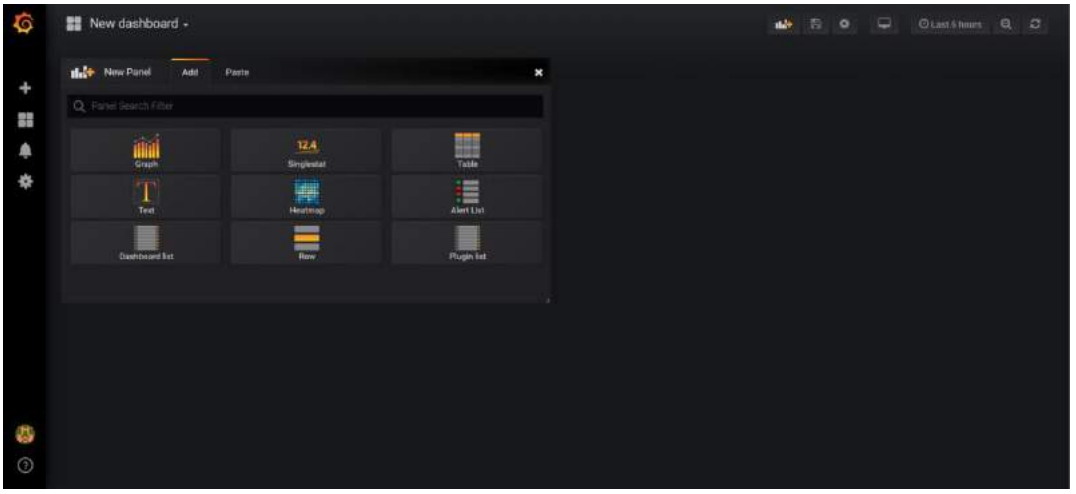
选择数据源为 Prometheus，设置数据源的地址为前面步骤启动的 Prometheus 的地址。



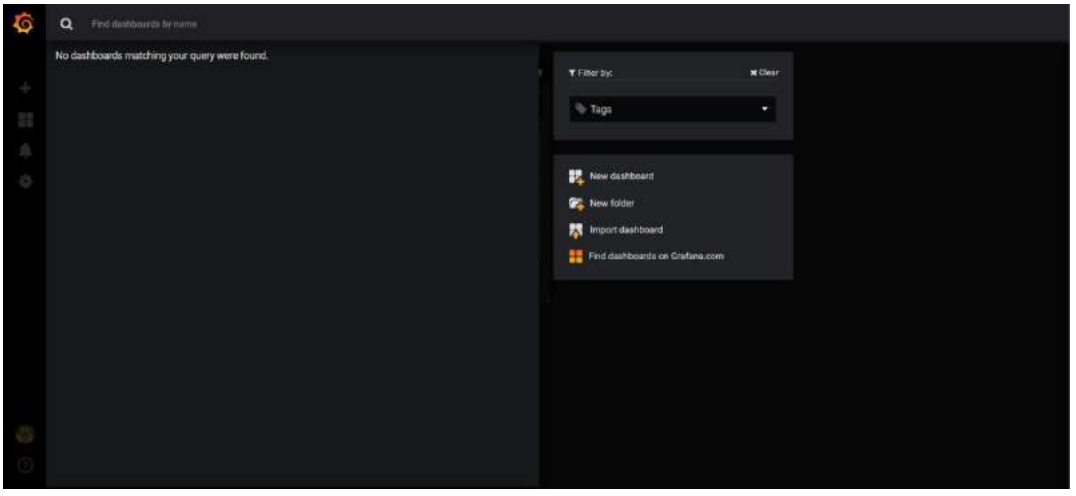
回到主界面会要求创建新的 Dashboard。



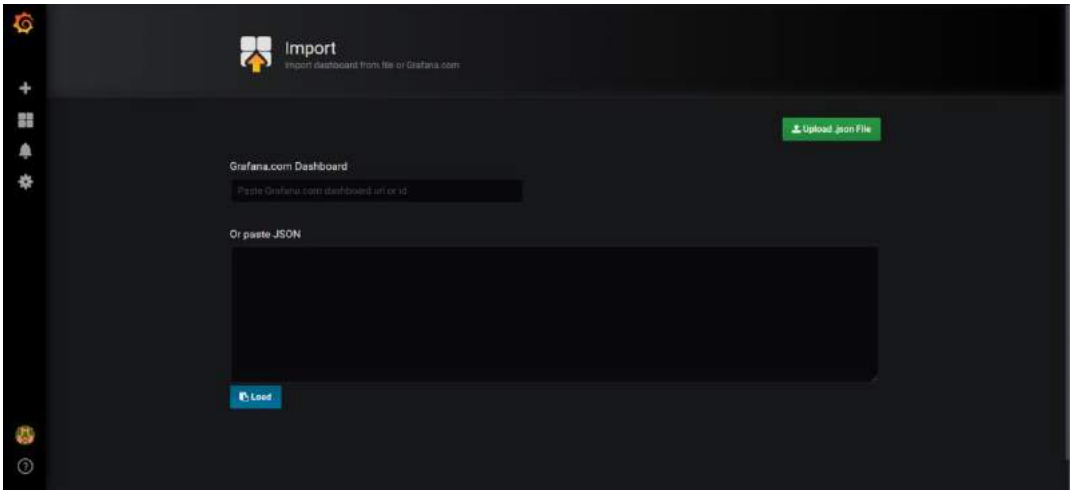
点击创建 dashboard，创建 dashboard 可以自己手动创建，也可以以配置文件导入的方式创建，当前已将 RocketMQ 的 dashboard 配置文件上传到 Grafana 的官网，这里以配置文件导入的方式进行创建。



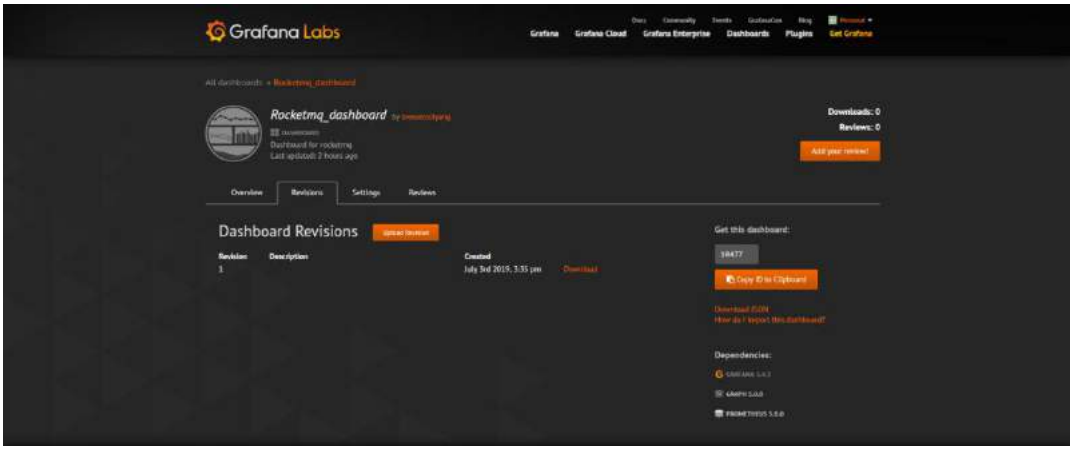
点击 New dashboard 下拉按钮。



选择 import dashboard。

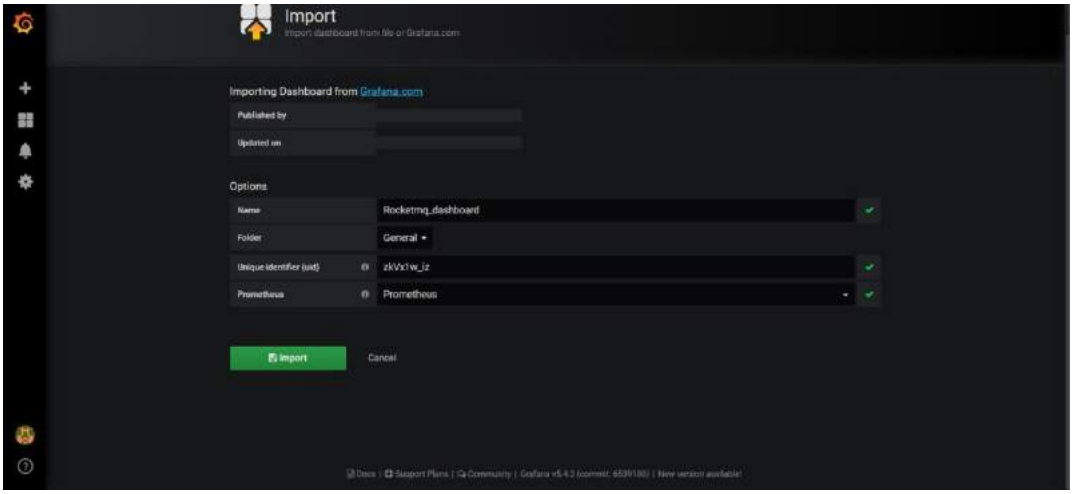


这个时候可以到 Grafana 官网去下载当前已为 RocketMQ 创建好的配置文件，地址为 https://grafana.com/dashboards/10477/revisions ，如下图所示：



点击 download 就可以下载配置文件，下载配置文件然后，复制配置文件中的内容粘贴到上图的粘贴内容处。

最后按上述方式就将配置文件导入到 Grafana 了。



最终的效果如下所示：

当 RocketMQ 遇上 Serverless，会碰撞出怎样的火花

作者 | 元毅 阿里巴巴高级开发工程师

PC 端登录 start.aliyun.com 即在浏览器中体验 RocketMQ 在线可交互教程

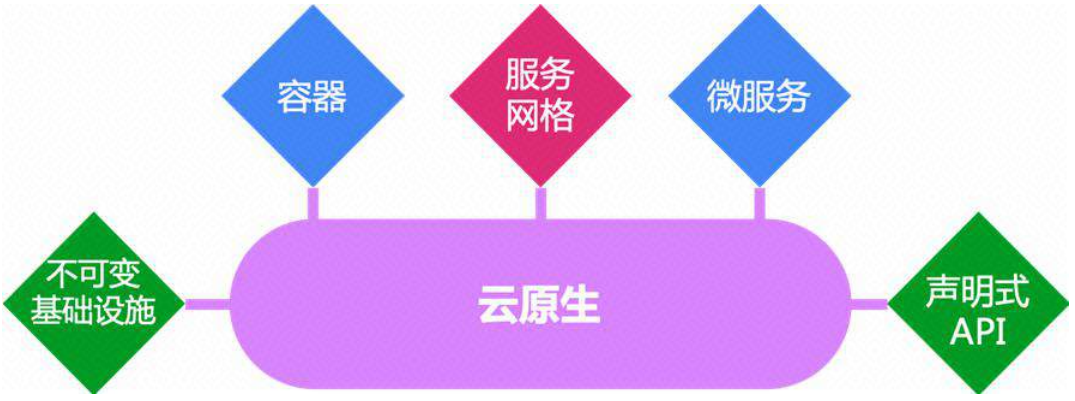
想必大家都比较了解 RocketMQ 消息服务，那么 RocketMQ 与 Serverless 结合会碰撞出怎样的火花呢？我们今天介绍一下如何基于 RocketMQ + Knative 驱动云原生 Serverless 应用 。本文主要从以下几个方面展开介绍：

- 云原生与 Serverless
- Knative 简介
- RocketMQSource
- 餐饮配送场景示例

一、云原生

先看一下 CNCF 对云原生的定义：

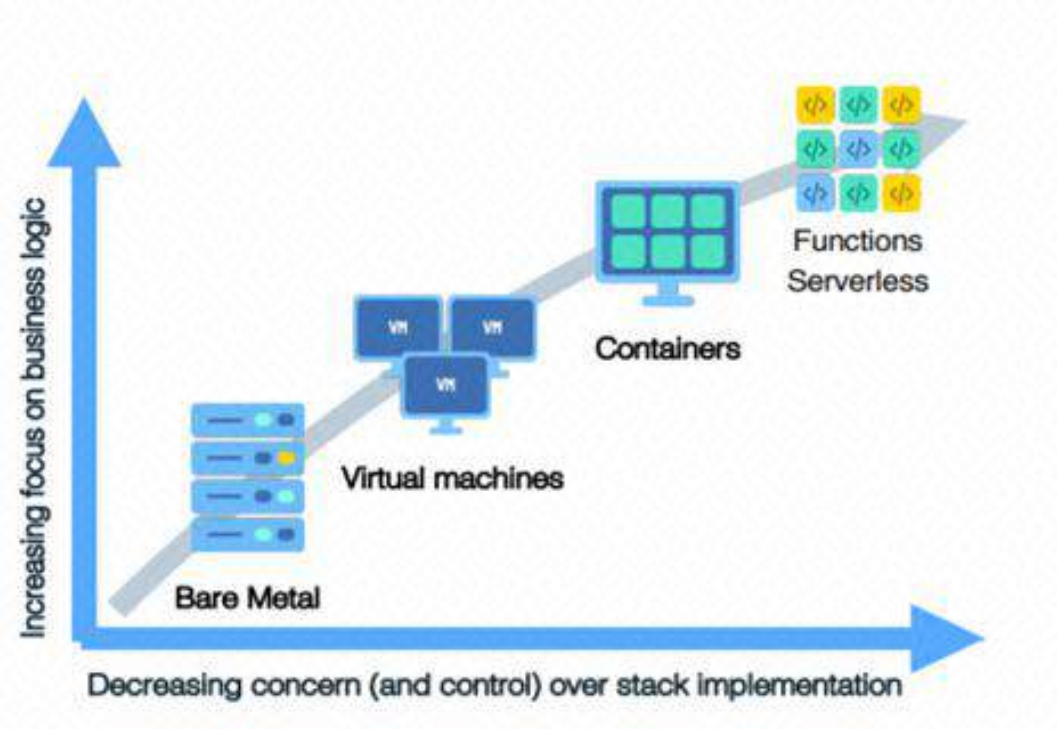
云原生技术有利于各组织在公有云、私有云和混合云等新型动态环境中，构建和运行可弹性扩展的应用。云原生的代表技术包括容器、服务网格、微服务、不可变基础设施和声明式 API。



这些技术能够构建容错性好、易于管理和便于观察的松耦合系统。结合可靠的自动化手段，云原生技术使工程师能够轻松地对系统作出频繁和可预测的重大变更。

其实云原生旨在以标准化云服务的提供方式衔接云厂商和客户。这种方式对于客户而言降低了上云和跨云迁移的成本，让客户始终保有和云厂商议价的能力；对云厂商而言，因为客户跨云迁移的成本低，所以只要能提供性价比更高的云服务，就能很容易的聚集大量用户。

二、Serverless



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

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

这里提到的是 Functions Serverless，其实除了 Functions Serverless， 还有另外一种 Serverless 形态：容器化的 Serverless。相较于 Function Serverless，容器化的

Serverless，可移植性更强，开发者对复杂应用程序能进行更好的掌控。除此之外，对于那些经历过容器时代洗礼的用户，容器化的 serverless 或许是一种更好的选择。

对于 Serverless，有如下几点需要关注一下：

- 事件（event）驱动：Serverless 是由事件（event）驱动（例如 HTTP、pub/sub）的全托管计算服务；
- 自动弹性：按需使用，削峰填谷；
- 按使用量计费：相对于传统服务按照使用的资源（ECS 实例、VM 的规格等）计费，Serverless 场景下更多的是按照服务的使用量（调用次数、时长等）计费；
- 绿色的计算：所谓绿色的计算其实就是最大化的提升资源使用效率，减少资源浪费，做的“节能减排”。

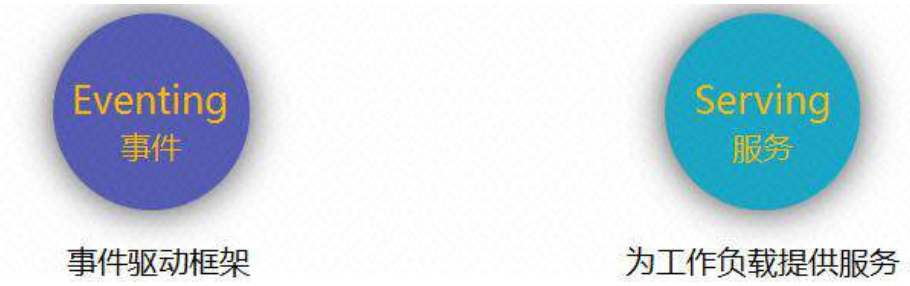
三、Knative

上面提到了容器化的 Serverless，那么有没有这样的 Serveless 平台框架呢？答案就是：Knative。

Knative 是在 2018 的 Google Cloud Next 大会上发布的一款基于 Kubernetes 的 Serverless 编排引擎。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(弹性)以及事件模型这三者来实现的这一 Serverless 标准。Knative 社区的当前主要贡献者有 Google、Pivotal、IBM、RedHat。另外像 CloudFoundry、OpenShift 这些 PaaS 提供商都在积极的参与 Knative 的建设。

1. Knative 核心模块

Knative 核心模块主要包括事件驱动框架 Eventing 和部署工作负载的 Serving。

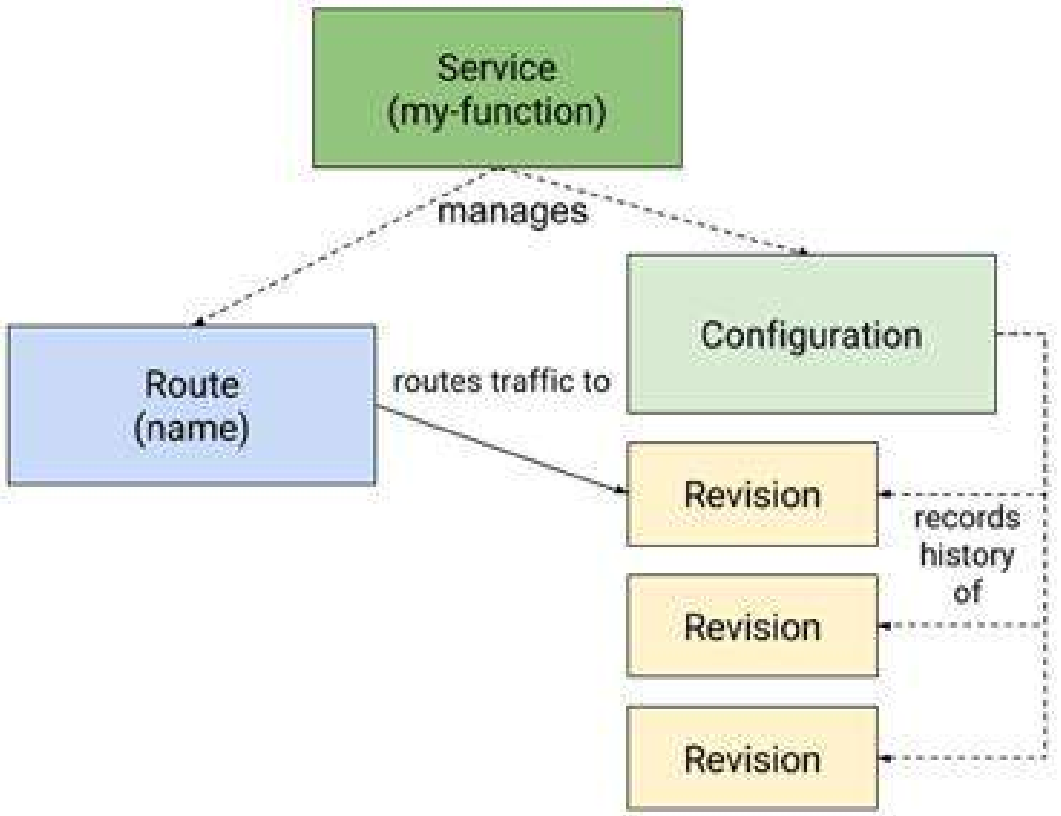


2. Serverless 服务引擎 - Serving

Knative Serving 核心能力就是其简洁、高效的应用托管服务，这也是其支撑 Serverless 能力的基础。Knative 提供的应用托管服务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 Severlesss 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。

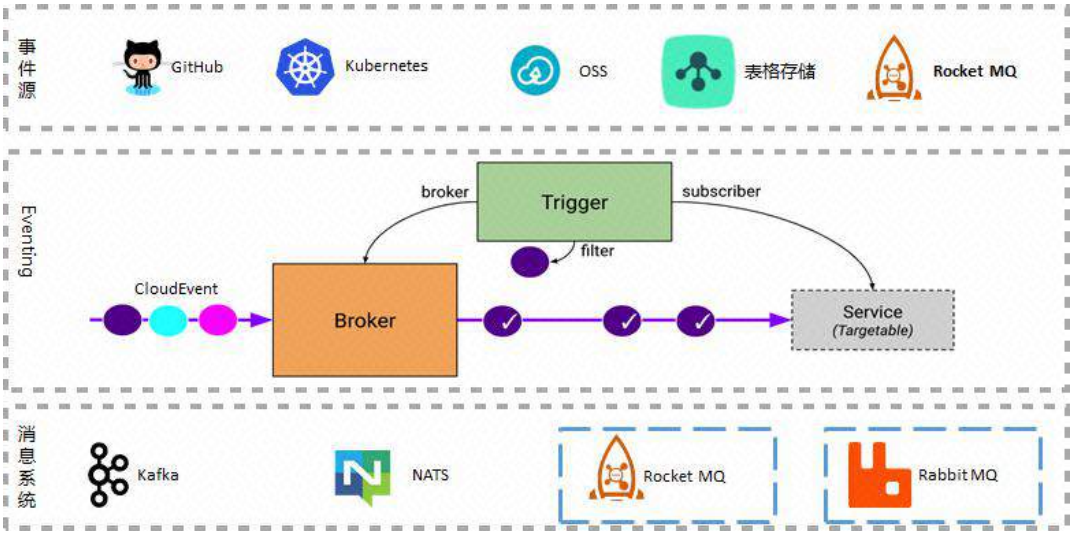
Serving 通过与 Istio 结合还提供了强大的流量管理能力和灵活的灰度发布能力。流量管理能力可以根据百分比切分流量，灰度发布能力可以根据流量百分比进行灰度，同时灰度发布能力还能通过自定义 tag 的方式进行上线前的测试，非常便于和自己的 CICD 系统集成。

Serving 应用模型



- Service: 对应用 Serverless 编排的抽象，通过 Service 管理应用的生命周期；
- Configuration: 当前期望状态的配置。每次更新 Service 就会更新 Configuration；
- Revision: configuration 的每次更新都会创建一个快照，用来做版本管理；
- Route: 将请求路由到 Revision，并可以向不同的 Revision 转发不同比例的流量。

3. 事件驱动框架 - Eventing



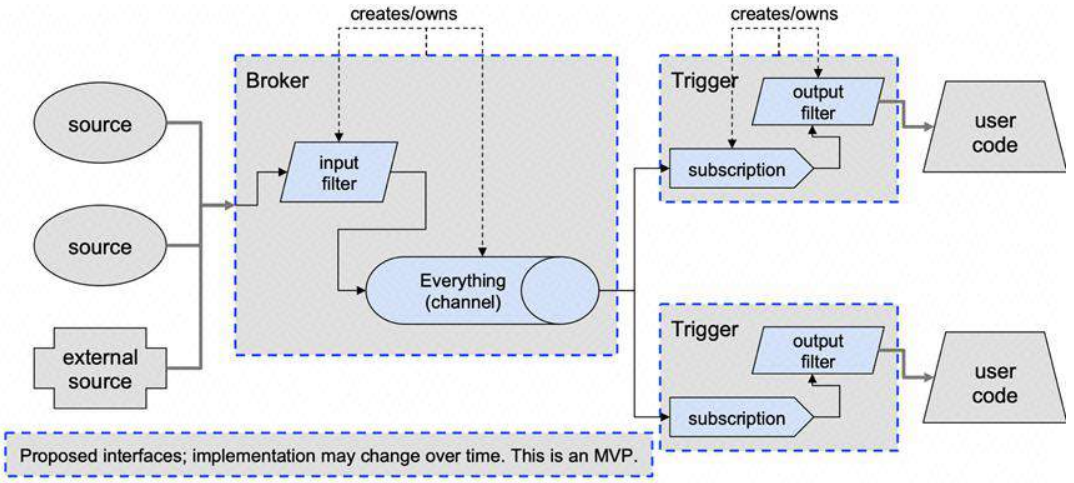
1.采用 CloudEvent 作为事件传输协议：CloudEvent 以通用的格式描述事件数据，提供跨平台的服务交互能力。KnativeEventing 使用 CloudEvent 作为事件传输标准，极大的提升了应用的跨平台可移植性；

2.外部事件源接入和注册：提供 Github、RocketMQ 以及 Kafka 等事件源的支持，当然用户可以自定义事件源；

3.事件的订阅和触发：引入 Broker 和 Trigger 模型意义，不仅将事件复杂的处理实现给用户屏蔽起来，更提供丰富的事件订阅、过滤机制；

4.兼容现有消息系统：KnativeEventing 充分解耦了消息系统的实现，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、NATSStreaming 等消息服务,此外可以方便的对接现有的消息系统。

Eventing 中 Broker/Trigger 模型



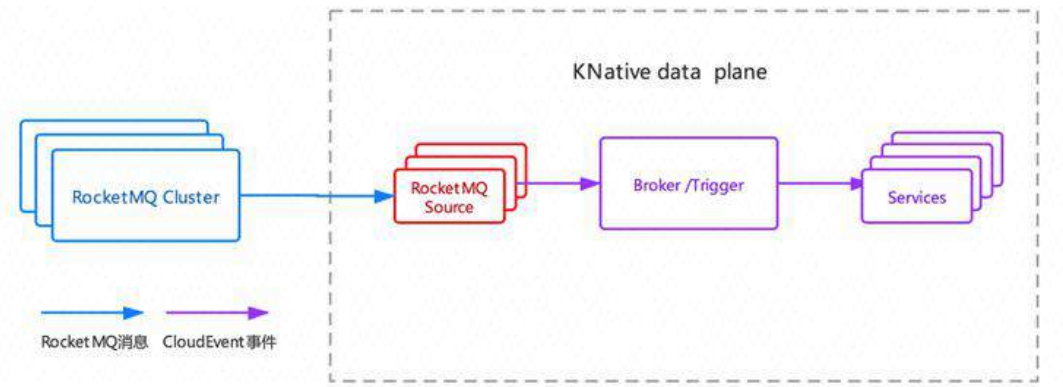
这里介绍一下 Eventing 中 Broker/Trigger 模型， 其实并不复杂。外部事件源将事件发送给 Broker， Broker 接收事件之后发送给对应的 Channel（也就是消息缓存，转发的地方，如 Kafka，InMemoryChannel 等），通过创建 Trigger 订阅 Broker 实现事件的订阅，另外在 Trigger 中定义对应的服务，实现最终的事件驱动服务。

四、消息队列 RocketMQ

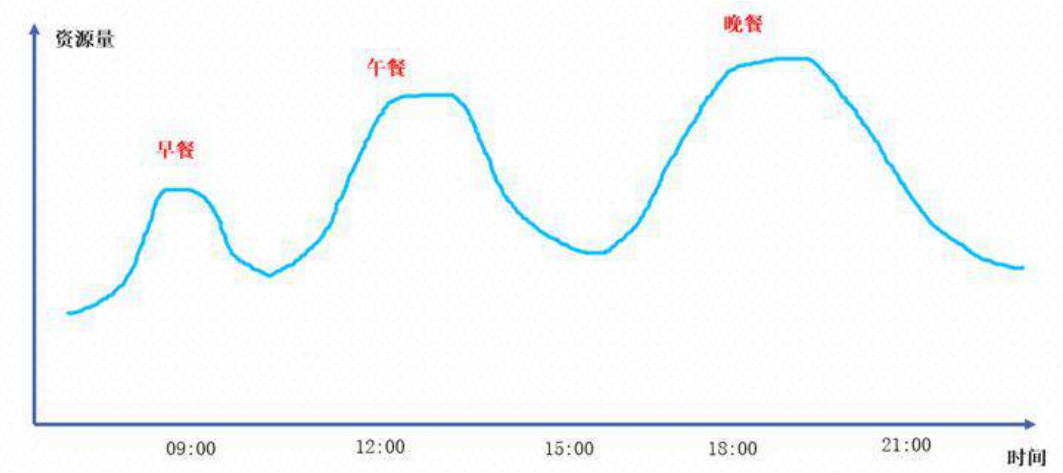
消息队列 RocketMQ 版是阿里云基于 Apache RocketMQ 构建的低延迟、高并发、高可用、高可靠的分布式消息中间件。消息队列 RocketMQ 版既可为分布式应用系统提供异步解耦和削峰填谷的能力，同时也具备互联网应用所需的海量消息堆积、高吞吐、可靠重试等特性。

RocketMQSource

RocketMQSource 是 Knative 平台的 RocketMQ 事件源。其可以将 RocketMQ 集群的消息以 Cloud Event 的格式实时转发到 Knative 平台，是 Apahe RocketMQ 和 Knative 之间的连接器。



五、Knative + RocketMQ 场景示例-餐饮配送场景

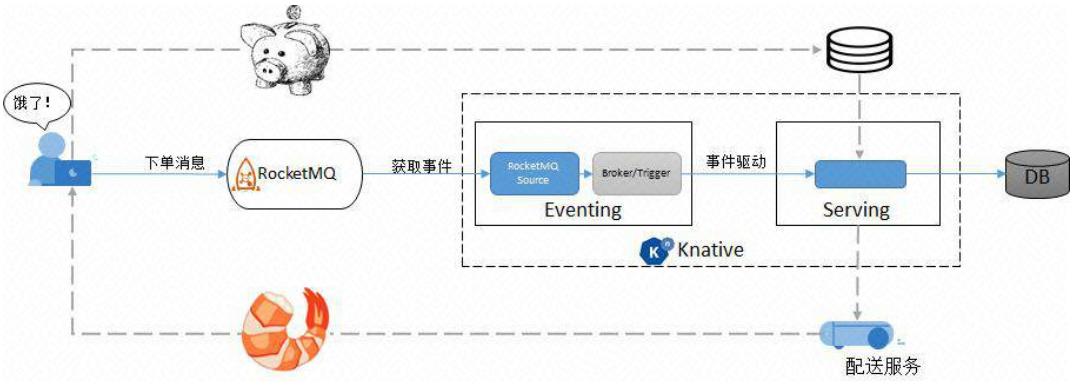


我们接下来以餐饮配送为例进行演示，餐饮配送场景具有以下特征：

- 餐饮配送一天之内存在明显的高峰、低谷；
- 高峰时间下单量很大。

针对这样的情况，我们采用消息驱动 Serverless， 在高峰的时候自动扩容资源，在低谷的时候缩减资源，按需使用能极大的提升资源使用率，从而降低成本。

1. 典型架构



如上图所示，当用餐时间来临，客户点餐生成下单消息发送到 RocketMQ，通过 RocketMQSource 获取下单消息转换成事件发送到 Broker，通过 Trigger 订阅下单事件最终驱动订单服务生成订餐单。采用该方案具有以下优势：

- 通过 Knative 技术以 RocketMQ 为核心将餐饮配送系统 Serverless 化可以极大程度降低服务器运维与成本；
- Knative 的弹性可以帮你轻松应对早、中、晚三餐资源高峰需求；
- 系统以 RocketMQ 做异步解耦，避免长链路调用等问题，提高系统可用性。

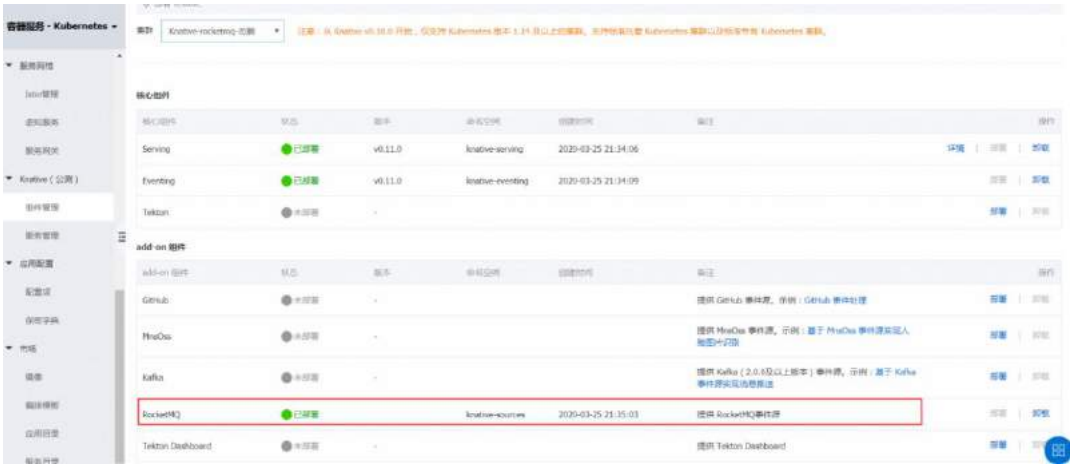
2. 操作

部署 Knative

参见阿里云容器服务部署 Knative。

部署 RocketMQSource

在 Knative 组件管理中，选择 RocketMQSource 点击部署。



部署订单服务

参考示例代码仓库。

一键部署服务命令如下：

```
kubectl apply -f 200-serviceaccount.yaml -f 202-clusterrolebinding.yaml -f 203-secret.yaml -f alirocketmqsource.yaml -f broker.yaml -f ksvc-order-service.yaml -f trigger.yaml
```

模拟高峰订餐下单

通过模拟下单，往 RocketMQ 中并发发送消息即可。消息格式参考：

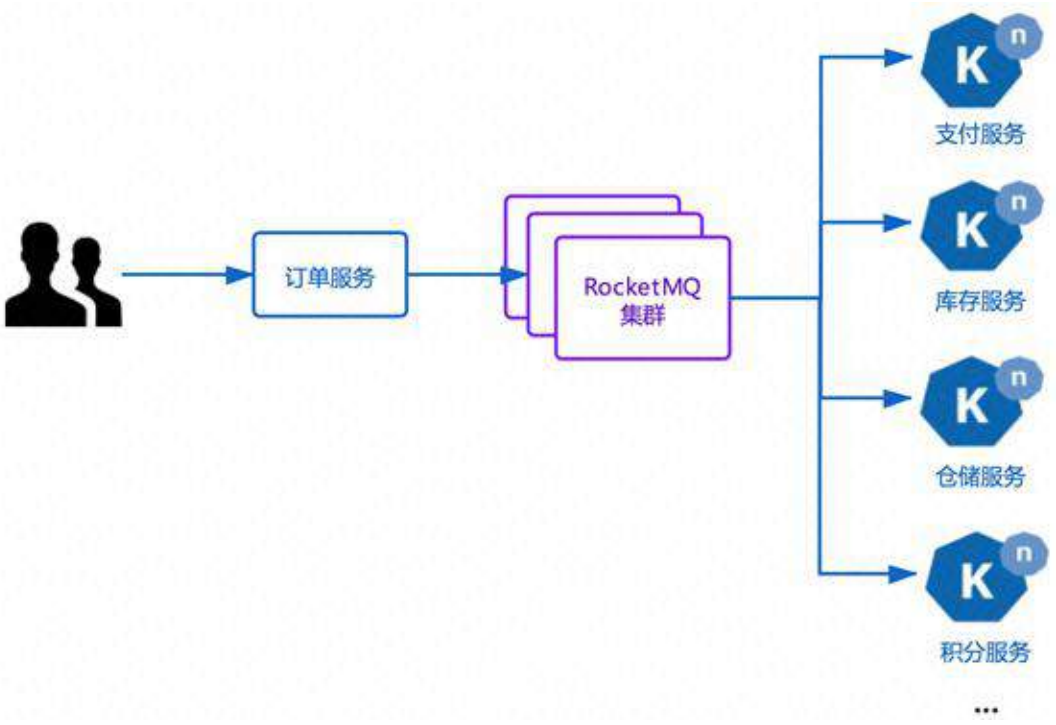
```
{"orderId":"123214342","orderStatus":"completed","userPhoneNo":"152122131323","prodId":"2141412","prodName":"test","chargeMoney":"30.0","chargeTime":"1584932320","finishTime":"1584932320"}
```

3. 演示效果

如下图所示：

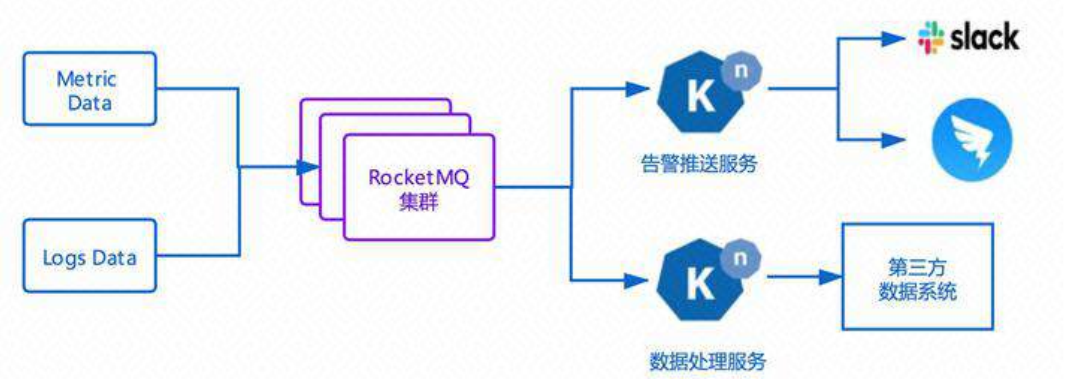
六、其它应用场景

1. Knative + RocketMQ 典型场景 - 构建 Serverless 电商系统



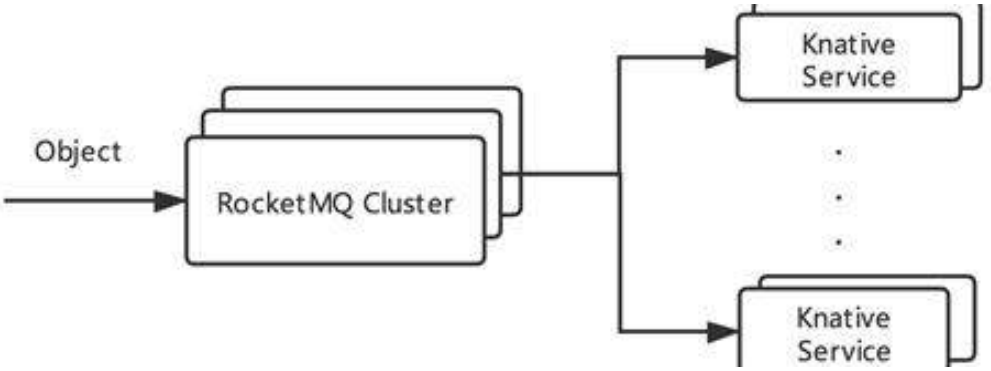
- Knative 弹性可以帮你轻松应对团购、双 11 等电商的大促活动；
- 系统以 RocketMQ 为中心做异步解耦，避免长链路调用等问题，提高系统可用性。

2. Knative + RocketMQ 典型场景- 构建监控告警平台



- Metric、Log 等数据通过 RocketMQ 集群推送到 Knative 服务；
- Knative 服务通过数据分析将告警内容推送钉钉或 slack 等通讯工具；
- Knative 服务可以将 Metric 或 logs 数据进行处理，推送第三方系统。

3. Knative + RocketMQ 典型场景- 多数据格式转换



- 处理数据日志以生成多个结果派生词，这些结果派生词可用于运营，营销，销售等；
- 将内容从一种格式转换为另一种格式，例如，将 Microsoft Word 转换为 PDF；
- 需要转换为多种格式的主媒体文件。

七、总结

通过以上 RocketMQ 事件驱动 Knative Serverless 应用的介绍，是否也给你碰撞出了火花？可以结合自身的应用场景不妨一试，相信会给你带来不一样的体验。

RocketMQ 中国社区钉钉群
欢迎各位开发者进群交流、勘误

微信扫码关注
“阿里巴巴云原生公众号”

阿里云开发者“藏经阁”
海量电子书免费下载