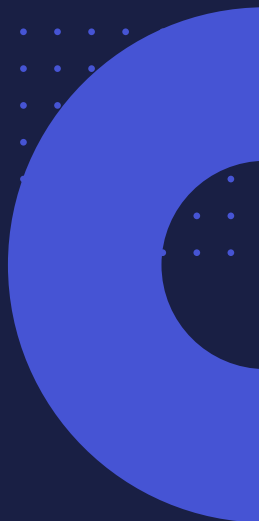




# 阿里巴巴 云原生实践 15 讲



始于 2011 年，阿里巴巴云原生规模化落地之路全景揭秘





微信扫一扫关注  
阿里巴巴云原生公众号



微信扫一扫关注  
阿里技术公众号



阿里云开发者社区  
开发者一站式平台



钉钉扫描二维码，  
立即加入 K8s 社区大群



# 目录

<b>开篇</b>	<b>iv</b>
容器十年——一部软件交付编年史	iv
<b>阿里云原生实践</b>	<b>1</b>
坚持探索与落地并重，阿里巴巴云原生之路全景揭秘	1
1-5-10：如何快速恢复大规模容器故障	6
阿里巴巴利用 K8S、Kata 容器和裸金属服务器构建无服务器平台	23
CafeDeployment：为互联网金融关键任务场景扩展的 Kubernetes 资源	35
Serverless 市场观察和落地挑战	45
有效可靠地管理大规模 Kubernetes 集群	51
<b>阿里新技术方案</b>	<b>59</b>
云原生应用 Kubernetes 监控与弹性实践	59
了解 Kubernetes Master 的可扩展性和性能	66
云原生时代加速镜像分发的三种方法	81
在 Web 级集群中动态调整 Pod 资源限制	93
大规模 k8s 集群下的巡检	108
使用 Istio 管理跨地域多集群的服务	122
<b>阿里云开源贡献</b>	<b>135</b>
首个普惠社区的平民化方案：GPU 共享调度	135
容器运行时管理引擎 Containerd	148
基于 P2P 原理的高可用高性能大规模镜像分发系统：Dragonfly	155

## 开篇

# 容器十年——一部软件交付编年史

阿里云容器平台高级技术专家 张磊

2019年，全世界的开发人员都开始习惯用容器测试自己的软件，用容器做线上发布，开始对容器化的软件构建和交付流程习以为常。全世界的架构师们都在对“云原生”侃侃而谈，描绘多云时代的应用治理方式，不经意间就把“sidecar”这种容器组织方式当做了默认选项。在“云”已经成为了大众基础设施的今天，我们已经习惯了把“容器”当做现代软件基础设施的基本依赖。这就像我们每天打开 Eclipse 编写 Java 代码一样自然。

但往回倒数两年，整个容器生态都还在围着 Docker 公司争得不可开交，看起来完全没有定数。当时的国内很多公有云厂商，甚至都没有正式的 Kubernetes 服务。在那个时候，要依托容器技术在云上托管完整的软件生命周期，可以说是相当前沿的探索。谁能想到短短两年之后，容器这个站在巨人肩膀上的设计，就真的成为技术人员日常工作的一部分呢？

伴随着容器技术普及到“千家万户”，我们在这两年期间所经历的，是现代软件交付形式的一次重要变革。

## 源起：属于 Jails 们的时代

虚拟化容器技术 (virtualized container) 的历史，其实可以一直追溯上世纪 70 年代末。时间回到 1979 年，贝尔实验室 (Bell Laboratories) 正在为 Unix V7 (Version 7 Unix) 操作系统的发布进行最后的开发和测试工作。



Ken Thompson(sitting) and Dennis Ritchie at PDP-11 ©wikipedia

在那个时候，Unix 操作系统还是贝尔实验室的内部项目，而运行 Unix 的机器则是长得像音响一样的、名叫 PDP 系列的巨型盒子。在那个“软件危机(The Software Crisis)”横行的末期，开发和维护 Unix 这样一套操作系统项目，即使对贝尔实验室来说也绝非易事。更何况，那里的程序员们还得一边开发 Unix，一边开发 C 语言本身呢。

而在 Unix V7 的开发过程中，系统级别软件构建 (Build) 和测试 (Test) 的效率其实是其中一个最为棘手的难题。这里的原因也容易理解：当一个系统软件编译和安装完成后，整个测试环境其实就被“污染”了。如果要进行下一次构建、安装和测试，就必须重新搭建和配置整改测试环境。在有云计算能力的今天，我们或许可以通过虚拟机等方法来完整的复现一个集群。但在那个一块 64K 的内存条要卖 419 美元的年代，“快速销毁和重建基础设施”的想法还是有点“科幻”了。

所以，贝尔实验室的聪明脑袋们开始构思一种在现有操作系统环境下“隔离”出一个可供软件进行构建和测试的环境。更确切的说，就是我能否简单的执行一些指令，就改变一个程序的“视图”，让它把当前目录当做自己的根目录呢？这样，我每次只要在当前目录里放置一个完整操作系统文件系统部分，该软件运行所需的所有依赖就完备了。

更为重要的是，有了这个能力，开发者实际上就间接拥有了应用基础设施“快速销毁和重现”的能力，而不需要在环境搭好之后进入到环境里去进行应用所需的依赖安装和配置。这当然是因为，现在我的整个软件运行的依赖，是以一个操作系统文件目录的形式事先准备好的，而开发者只需要构建和测试应用的时候，切换应用“眼中”的根目录到这个文件目录即可。

**于是，一个叫做 chroot (Change Root) 的系统调用就此诞生了。**

顾名思义，chroot 的作用是“重定向进程及其子进程的根目录到一个文件系统上的新位置”，使得该进程再也看不到也没法接触到这个位置上层的“世界”。所以这个被隔离出来的新环境就有一个非常形象的名字，叫做 Chroot Jail。

值得一提的是，这款孕育了 chroot 的 Unix V7 操作系统，成为了贝尔实验室 Unix 内部发行版的绝唱。在这一年末尾，Unix 操作系统正式被 AT&T 公司商业化，并被允许授权给外部使用，自此开启了一代经典操作系统的传奇之旅。

**而 Chroot 的诞生，也第一次为世人打开了“进程隔离”的大门。**

伴随着这种机制被更广泛的用户接触到，chroot 也逐渐成为了开发测试环境配置和应用依赖管理的一个重要工具。而“Jail”这个用来描述进程被隔离环境的概念，也开始激发出这个技术领域更多的灵感。

**在 2000 年，同属 Unix 家族的 FreeBSD 操作系统发布了“jail”命令，宣布了 FreeBSD Jails 隔离环境的正式发布。相比于 Chroot Jail，FreeBSD Jails 把”隔离“这个概念扩展到了进程的完整视图，隔离出了独立进程环境和用户体系，并为 Jails 环境分配了独立的 IP 地址。所以确切的说，尽管 chroot 开创了进程环境隔离的思想，但 FreeBSD Jails，其实才真正实现了进程的沙箱化。而这里的关键在于，这种沙箱的实现，依靠的是操作系统级别的隔离与限制能力而非硬件虚拟化技术。**



不过，无论是 FreeBSD Jails (Unix 平台)，还是紧接着出现的 Oracle Solaris Containers (Solaris 平台)，都没有能在更广泛的软件开发和交付场景中扮演到更重要的角色。在这段属于 Jails 们的时代，进程沙箱技术因为“云”的概念尚未普及，始终被局限在了小众而有限的世界里。

## 发展：云，与应用容器

事实上，在 Jails 大行其道的这几年间，同样在迅速发展的 Linux 阵营上也陆续出现多个类似的沙箱技术比如 Linux VServer 和 Open VZ (未进入内核主干)。但跟 Jails 的发展路径比较类似，这些沙箱技术最后也都沦为了小众功能。我们再次看到了，进程沙箱技术的发展过程中“云”的角色缺失所带来的影响，其实还是非常巨大的。而既然说到“云”，我们就不得不提到基础设施领域的翘楚：Google。

Google 在云计算背后最核心的基础设施领域的强大影响力，是被业界公认的一个事实，无论是当初震惊世界的三大论文，还是像 Borg/Omega 等领先业界多年的内部基础设施项目，都在这个领域扮演者重要的启迪者角色。然而，放眼当今的云计算市场，仅仅比 Google 云计算发布早一年多的 AWS 却是“云”这个产业毫无争议

的领导者。而每每谈到这里的原因，大家都会提到一个充满了争议的项目：GAE。



GAE 对于中国的某几代技术人员来说，可以说是一个挥之不去的记忆。然而，即使是这批 GAE 的忠实用户，恐怕也很难理解这个服务竟然就是 Google 当年决定用来对抗 AWS 的核心云产品了。事实上，GAE 本身与其说是 PaaS，倒不如说是简化版的 Serverless。在 2008 年，在绝大多数人还完全不知道云计算为何物的情况下，这样的产品要想取得成功拿下企业用户，确实有些困难。

**不过，在这里我们想要讨论的并不是 Google 的云战略，而是为什么从技术的角度上，Google 也认为 GAE 这样的应用托管服务就是云计算呢？**

这里的一个重要原因可能很多人都了解过，那就是 Google 的基础设施技术栈其实是一个标准容器技术栈，而不是一个虚拟机技术栈。更为重要的是，在 Google 的这套体系下，得益于 Borg 项目提供的独有的应用编排与管理能力，容器不再是一个简单的隔离进程的沙箱，而成为了对应用本身的一种封装方式。依托于这种轻量级的应用封装，Google 的基础设施可以说是一个天然以应用为中心的托管架构和编程框架，这也是很多前 Googler 调侃“离开了 Borg 都不知道怎么写代码”的真实含义。这样一种架构和形态，映射到外部的云服务成为 GAE 这样的一个 PaaS/Serverless 产品，也就容易理解了。

**而 Google 这套容器化基础设施的规模化应用与成熟，可以追溯到 2004~2007 年之间，而这其中一个最为关键的节点，正是一种名叫 Process Container 技术的发布。**

Process Container 的目的非常直白，它希望能够像虚拟化技术那样给进程提



供操作系统级别的资源限制、优先级控制、资源审计能力和进程控制能力，这与前面提到的沙箱技术的目标其实是一致的。这种能力，是前面提到的 Google 内部基础设施得以实现的基本诉求和基础依赖，同时也成为了 Google 眼中“容器”技术的雏形。带着这样的设思路，Process Container 在 2006 年由 Google 的工程师正式推出后，第二年就进入了 Linux 内核主干。

不过，由于 Container 这个术语在 Linux 内核中另有它用，Process Container 在 Linux 中被正式改名叫作：Cgroups。Cgroups 技术的出现和成熟，标志在 Linux 阵营中“容器”的概念开始被重新审视和实现。更重要的是，这一次“容器”这个概念的倡导者变成了 Google：一个正在大规模使用容器技术定义世界级基础设施的开拓者。

在 2008 年，通过将 Cgroups 的资源管理能力和 Linux Namespace 的视图隔离能力组合在一起，LXC (Linux Container) 这样的完整的容器技术出现在了 Linux 内核当中。尽管 LXC 提供给用户的能力跟前面提到的各种 Jails 以及 OpenVZ 等早期 Linux 沙箱技术是非常相似的，但伴随着 Linux 操作系统开始迅速占领商用服务器市场的契机，LXC 的境遇比前面的这些前辈要好上不少。

而更为重要的是，2008 年之后 AWS，Microsoft 等巨头们持续不断的开始在公有云市场上进行发力，很快就催生出了一个全新的、名叫 PaaS 的新兴产业。

老牌云计算厂商在 IaaS 层的先发优势以及这一部分的技术壁垒，使得越来越多受到公有云影响的技术厂商以及云计算的后来者，开始思考如何在 IaaS 之上构建新的技术与商业价值，同时避免走入 GAE 当年的歧途。在这样的背景之下，一批以开源和开放为主要特点的平台级项目应运而生，将“PaaS”这个原本虚无缥缈的概念第一次实现和落地。这些 PaaS 项目的定位是应用托管服务，而不同于 GAE 等公有云托管服务，这些开放 PaaS 项目希望构建的则是完全独立于 IaaS 层的一套应用管理生态，目标是借助 PaaS 离开发者足够近的优势锁定云乃至所有数据中心的更上层入口。这样的定位，实际上就意味着 PaaS 项目必须能够不依赖 IaaS 层虚拟机技术，就能够将用户提交的应用进行封装，然后快速的部署到下层基础设施上。而这其中，开源、中立，同时又轻量、敏捷的 Linux 容器技术，自然就成为了 PaaS 进行托

管和部署应用的最佳选择。

在 2009 年，VMware 公司在收购了 SpringSource 公司（Spring 框架的创始公司）之后，将 SpringSource 内部的一个 Java PaaS 项目的名字，套在了自己的一个内部 PaaS 头上，然后于 2011 年宣布了这个项目的开源。这个名字，就叫做：Cloud Foundry。2009 年，Cloud Foundry 项目的诞生，第一次对 PaaS 的概念完成了清晰而完整的定义。



这其中，“PaaS 项目通过对应用的直接管理、编排和调度让开发者专注于业务逻辑而非基础设施”，以及“PaaS 项目通过容器技术来封装和启动应用”等理念，也第一次出现在云计算产业当中并得到认可。值得一提的是，Cloud Foundry 用来操作和启动容器的项目叫做：Warden，它最开始是一个 LXC 的封装，后来重构成了直接对 Cgroups 以及 Linux Namespace 操作的架构。

Cloud Foundry 等 PaaS 项目的逐渐流行，与当初 Google 发布 GAE 的初衷是完全一样的。说到底，这些服务都认为应用的开发者不应该关注于虚拟机等底层基础设施，而应该专注在编写业务逻辑这件最有价值的事情上。这个理念，在越来越多的人得以通过云的普及开始感受到管理基础设施的复杂性和高成本之后，才变得越来越有说服力。在这幅蓝图中，Linux 容器已经跳出了进程沙箱的局限性，开始扮演着“应用容器”的角色。在这个新的舞台上，容器和应用终于画上了等号，这才最终使得平台层系统能够实现应用的全生命周期托管。

按照这个剧本，容器技术以及云计算的发展，理应向着 PaaS 和以应用为中心的方向继续演进下去。如果不是有一家叫做 Docker 的公司出现的话。

## 容器：改写的软件交付的历程

如果不是亲历者的话，你很难想象 PaaS 乃至云计算产业的发展，会如何因为

2013 年一个创业公司开源项目的发布而被彻底改变。但这件事情本身，确实是过去 5 年间整个云计算产业变革的真实缩影。



Docker 项目的发布，以及它与 PaaS 的关系，想必我们已经无需再做赘述。一个“降维打击”，就足以给当初业界的争论不休画上一个干净利落的句号。

我们都知道，Docker 项目发布时，无非也是 LXC 的一个使用者，它创建和使用应用容器的逻辑跟 Warden 等没有本质不同。不过，我们现在也知道，真正让 PaaS 项目无所适从的，是 Docker 项目最厉害的杀手锏：容器镜像。

关于如何封装应用，这本身不是开发者所关心的事情，所以 PaaS 项目有着无数的发挥空间。但到这如何定义应用这个问题，就是跟每一位技术人员息息相关了。在那个时候，Cloud Foundry 给出的方法是 Buildpack，它是一个应用可运行文件（比如 WAR 包）的封装，然后在里面内置了 Cloud Foundry 可以识别的启动和停止脚本，以及配置信息。

然而，Docker 项目通过容器镜像，直接将一个应用运行所需的完整环境，即：整个操作系统的文件系统也打包了进去。这种思路，可算是解决了困扰 PaaS 用户已久的一致性问题，制作一个“一次发布、随处运行”的 Docker 镜像的意义，一下子就比制作一个连开发和测试环境都无法统一的 Buildpack 高明了太多。

更为重要的是，Docker 项目还在容器镜像的制作上引入了“层”的概念，这种基于“层”（也就是“commit”）进行 build, push, update 的思路，显然是借鉴了 Git 的思想。所以这个做法的好处也跟 Github 如出一辙：制作 Docker 镜像不再是一个枯燥而乏味的事情，因为通过 DockerHub 这样的镜像托管仓库，你和你的软件立刻就可以参与到全世界软件分发的流程当中。

至此，你就应该明白，**Docker 项目实际上解决的确实是一个更高维度的问题：软件究竟应该通过什么样的方式进行交付？**更重要的是，一旦当软件的交付方式定义的如此清晰并且完备的时候，利用这个定义在去做一个托管软件的平台比如 PaaS，就变得非常简单而明了了。这也是为什么 Docker 项目会多次表示自己只是“站在巨人肩膀上”的根本原因：没有最近十年 Linux 容器等技术的提出与完善，要通过一个开源项目来定义并且统一软件的交付历程，恐怕如痴人说梦。

## 云，应用，与云原生

时至今日，容器镜像已经成为了现代软件交付与分发的事实标准。然而，Docker 公司却并没有在这个领域取得同样的领导地位。这里的原因相比大家已经了然于心：在容器技术取得巨大的成功之后，Docker 公司在接下来的“编排之争”中犯下了错误。事实上，Docker 公司凭借“容器镜像”这个巧妙的创新已经成功解决了“应用交付”所面临的最关键的技术问题。但在如何定义和管理应用这个更为上层的问题上，容器技术并不是“银弹”。在“应用”这个与开发者息息相关的领域里，从来就少不了复杂性和灵活性的诉求，而容器技术又天然要求应用的“微服务化”和“单一职责化”，这对于绝大多数真实企业用户来说都是非常困难的。而这部分用户，又偏偏是云计算产业的关键所在。

而相比于 Docker 体系以“单一容器”为核心的应用定义方式，Kubernetes 项目则提出了一整套容器化设计模式和对应的控制模型，从而明确了如何真正以容器为核心构建能够真正跟开发者对接起来的应用交付和开发范式。而 Docker 公司、Mesosphere 公司以及 Kubernetes 项目在“应用”这一层上的不同理解和顶层设计，其实就是所谓“编排之争”的核心所在。

2017 年末，Google 在过去十年编织全世界最先进的容器化基础设施的经验，最终帮助 Kubernetes 项目取得到了关键的领导地位，并将 CNCF 这个以“云原生”为关键词的组织和生态推向了巅峰。

而最为有意思的是，Google 公司在 Kubernetes 项目倾其全力注入的“灵魂”，既不是 Borg/Omega 多年来积累下来的大规模调度与资源管理能力，也不是“三大

论文”这样让当年业界望尘莫及的领先科技。Kubernetes 项目里最能体现 Google 容器理念的设计，是“源自 Borg/Omega 体系的应用编排与管理能力”。

我们知道，Kubernetes 是一个“重 API 层”的项目，但我们还应该理解的是，Kubernetes 是一个“以 API 为中心”的项目。围绕着这套声明式 API，Kubernetes 的容器设计模式，控制器模型，以及异常复杂的 apiserver 实现与扩展机制才有了意义。而这些看似繁杂的设计与实现背后，实际上只服务于一个目的，那就是：如何让用户在管理应用的时候能最大程度的发挥容器和云的价值。

本着这个目的，Kubernetes 才会把容器进行组合，用 Pod 的概念来模拟进程组的行为。才会坚持用声明式 API 加控制器模型来进行应用编排，用 API 资源对象的创建与更新 (PATCH) 来驱动整个系统的持续运转。更确切的说，有了 Pod 和容器设计模式，我们的应用基础设施才能够与应用（而不是容器）进行交互和响应的能力，实现了“云”与“应用”的直接对接。而有了声明式 API，我们的应用基础而设施才能真正同下层资源、调度、编排、网络、存储等云的细节与逻辑解耦。我们现在，可以把这些设计称为“云原生的应用管理思想”，这是我们“让开发者专注于业务逻辑”、“最大程度发挥云的价值”的关键路径。

所以说，Kubernetes 项目一直在做的，其实是在进一步清晰和明确“应用交付”这个亘古不变的话题。只不过，相比于交付一个容器和容器镜像，Kubernetes 项目正在尝试明确的定义云时代“应用”的概念。在这里，应用是一组容器的有机组合，同时也包括了应用运行所需的网络、存储的需求的描述。而像这样一个“描述”应用的 YAML 文件，放在 etcd 里存起来，然后通过控制器模型驱动整个基础设施的状态不断地向用户声明的状态逼近，就是 Kubernetes 的核心工作原理了。

## 未来：应用交付的革命不会停止

说到这里，我们已经回到了 2019 年这个软件交付已经被 Kubernetes 和容器重新定义的时间点。

在这个时间点上，Kubernetes 项目正在继续尝试将应用的定义、管理和交付推向一个全新的高度。我们其实已经看到了现有模型的一些问题与不足之处，尤其是声

明式 API 如何更好的与用户的体验达成一致。在这个事情上，Kubernetes 项目还有不少路要走，但也在快速前行。

我们也能够看到，整个云计算生态正在尝试重新思考 PaaS 的故事。Google Cloud Next 2019 上发布的 Cloud Run，其实已经间接宣告了 GAE 正凭借 Kubernetes 和 Knative 的标准 API “浴火重生”。而另一个典型的例子，则是越来越多很多应用被更“极端”的抽象成了 Function，以便完全托管于与基础设施无关的环境 (FaaS) 中。如果说容器是完整的应用环境封装从而将应用交付的自由交还给开发者，那么 Function 则是剥离了应用与环境的关系，将应用交付的权利交给了 FaaS 平台。我们不难看到，云计算在向 PaaS 的发展过程中被 Docker “搅局”之后，又开始带着“容器”这个全新的思路向“PaaS”不断收敛。只不过这一次，PaaS 可能会换一个新的名字叫做：Serverless。

我们还能够看到，云的边界正在被技术和开源迅速的抹平。越来越多的软件和框架从设计上就不会再会跟某云产生直接绑定。毕竟，你没办法抚平用户对商业竞争担忧和焦虑，也不可能阻止越来越多的用户把 Kubernetes 部署在全世界的所有云上和数据中心里。我们常常把云比作“水、电、煤”，并劝诫开发者不应该关心“发电”和“烧煤”的事情。但实际上，开发者不仅不关心这些事情，他们恐怕连“水、电、煤”是哪来的都不想知道。在未来的云的世界里，开发者完全无差别的交付自己的应用到世界任何一个地方，很有可能会像现在我们会把电脑插头插在房间里任何一个插孔里那样自然。这也是为什么，我们看到越来越多的开发者在讨论“云原生”。

我们无法预见未来，但代码与技术演进的正在告诉我们这样一个事实：未来的软件一定是生长于云上的。这将会是“软件交付”这个本质问题的不断自我革命的终极走向，也是“云原生”理念的最核心假设。而所谓“云原生”，实际上就是在定义一条能够让应用最大程度利用云的能力、发挥云的价值最佳路径。在这条路径上，脱离了“应用”这个载体，“云原生”就无从谈起；容器技术，则是将这个理念落地、将软件交付的革命持续进行下去的重要手段之一。

而至于 Kubernetes 项目，它的确是整个“云原生”理念落地的核心与关键所在。但更为重要的是，在这次关于“软件”的技术革命中，Kubernetes 并不需要尝

试在 PaaS 领域里分到一杯羹：它会成为连通“云”与“应用”的高速公路，以标准、高效的方式将“应用”快速交付到世界上任何一个位置。而这里的交付目的地，既可以是最终用户，也可以是 PaaS/Serverless 从而催生出更加多样化的应用托管生态。“云”的价值，一定会回归到应用本身。

## 一起进入云原生架构时代

在云的趋势下，越来越多的企业开始将业务与技术向“云原生”演进。这个过程中最为艰难的挑战其实来自于如何将应用和软件向 Kubernetes 体系进行迁移、交付和持续发布。而在这次技术变革的浪潮中，“云原生应用交付”，已经成为了 2019 年云计算市场上最热门的技术关键词之一。

阿里巴巴从 2011 年开始通过容器实践云原生技术体系，在整个业界都还没有任何范例可供参考的大背景下，逐渐摸索出了一套比肩全球一线技术公司并且服务于整个阿里集团的容器化基础设施架构。在这些数以万计的集群管理经验当中，阿里容器平台团队探索并总结了四个让交付更加智能与标准化的方法：Everything on Kubernetes，利用 K8s 来管理一切，包括 K8s 自身；应用发布回滚策略，按规则进行灰度发布，当然也包括发布 kubelet 本身；将环境进行镜像切分，分为模拟环境和生产环境；并且在监控侧下足功夫，将 Kubernetes 变得更白盒化和透明化，及早发现问题、预防问题、解决问题。

而近期，阿里云容器平台团队也官宣了两个社区项目：Cloud Native App Hub——面向所有开发者的 Kubernetes 应用管理中心，OpenKruise —— 源自全球顶级互联网场景的 Kubernetes 自动化开源项目集。

### OpenKruise 开源地址

<https://github.com/openkruise/kruise>

### Cloud Native App Hub

<https://developer.aliyun.com/hub>

云原生应用中心 (Cloud Native App Hub)，它首先为国内开发者提供了一个

Helm 应用中国镜像站，方便用户获得云原生应用资源，同时推进标准化应用打包格式，并可以一键将应用交付到 K8s 集群当中，大大简化了面向多集群交付云原生应用的步骤；而 OpenKruise/Kruise 项目致力于成为“云原生应用自动化引擎”，解决大规模应用场景下的诸多运维痛点。OpenKruise 项目源自于阿里巴巴经济体过去多年的大规模应用部署、发布与管理的最佳实践，从不同维度解决了 Kubernetes 之上应用的自动化问题，包括部署、升级、弹性扩缩容、QoS 调节、健康检查，迁移修复等等。

在接下来的时间，阿里云容器平台团队还会以此为基础，继续同整个生态一起推进云原生应用定义、K8s CRD 与 Operator 编程范式、增强型 K8s 自动化插件等一系列能够让云原生应用在规模化多集群场景下实现快速交付、更新和部署等技术的标准化与社区化。

我们期待与您一起共建，共同体迎接云原生时代的来临！



# 阿里云原生实践

## 坚持探索与落地并重， 阿里巴巴云原生之路全景揭秘

阿里云容器平台资深技术专家 李响

为什么要做云原生？云原生究竟能带来什么价值？从最初的独自摸索到如今拥抱开源回馈社区，阿里巴巴走过了怎样的云原生旅程？又有哪些技术心得？今天，将全部分享出来。

### 多年沉淀，坚持探索与落地并重

阿里巴巴从 2011 年开始通过容器实践云原生技术体系，在整个业界都还没有任何范例可供参考的大背景下，逐渐摸索出了一套比肩全球一线技术公司并且服务于整个阿里集团的容器化基础设施架构。这个探索历程虽然孤独，但却被始终如一的坚持至今。正是在这个孤注一掷的技术探索与奋进的过程中，阿里巴巴的技术团队完整的经历了云原生技术浪潮里的所有关键节点，不仅成为了这次技术革命的重要见证者，也逐渐成为中国云原生技术体系当之无愧的推动者与引领者之一。

阿里的体量大、业务复杂，推动云原生要找到合适的切入点。在双十一成本压力的推动下，资源成本与效率优化成了阿里云原生的起点。

阿里从容器入手，研究低成本虚拟化与调度技术：提供灵活、标准的部署单元；将静态资源分配更换为动态按需调度，进一步提升部署效率，解决资源碎片化问题，提高部署密度；通过存储网络虚拟化和存储计算分离等技术，增强任务的可迁移性，进一步提高了资源的可靠性，降低了资源成本。

在资源成本的推动下，阿里完成了全面容器化，资源分配也被高效调度平台接管。阿里的云原生并未止步于此。提高研发效率与加快迭代周期是推动阿里业务增强

的秘密武器。阿里希望通过云原生让开发者效率更高。

为了降低应用部署难度，提高部署自动化程度，阿里开始采用 Kubernetes 作为容器编排平台，并且持续推动 Kubernetes 的性能与可扩展性。具体 Kubernetes，阿里持续对研发、部署流程进行改进。为了构建更云原生化 CI/CD，进一步做到标准化和自动化，从研发到上线流程，阿里引入了诸如 Helm 的应用标准化管理，也尝试了 GitOps 这样的部署流程，还推动了 PaaS 层的面向终态自动化改造。于此同时，阿里也开始探索服务网格，致力于进一步提高服务治理的普适性与标准性，降低开发者采用门槛，进一步推动微服务在多语言和多环境下的普及。

今年，阿里也展开了全站上云。经过云原生的探索与改造，阿里基础架构体系是现代化和标准化的。利用容器技术，应用与宿主机运行时完成了解耦；利用 Kubernetes 对 Pod 与 Volume 等的抽象，完成了对多种资源实现的统一化；通过智能调度与 PaaS 平台，让自动迁移应用，修复不稳定因素成为了可能，阿里通过云原生技术大大降低了上云的难度。

在这个提高资源和人员效率的过程中，阿里巴巴的整个基础设施也变得更加开放，连通开源生态，在交流互动中不断吸收和贡献好的理念、技术、思想。如今，阿里云不仅支撑着中国最大的云原生应用双 11，而且拥有国内最大的公共云集群和镜像仓库。作为唯一入选 Gartner 的公有云容器服务竞争格局的厂商，阿里云也积累了最为丰富和宝贵的客户实践。

云原生技术与应用领先者

全方位贡献开源，致力成为云原生标准制定者和引领者之一

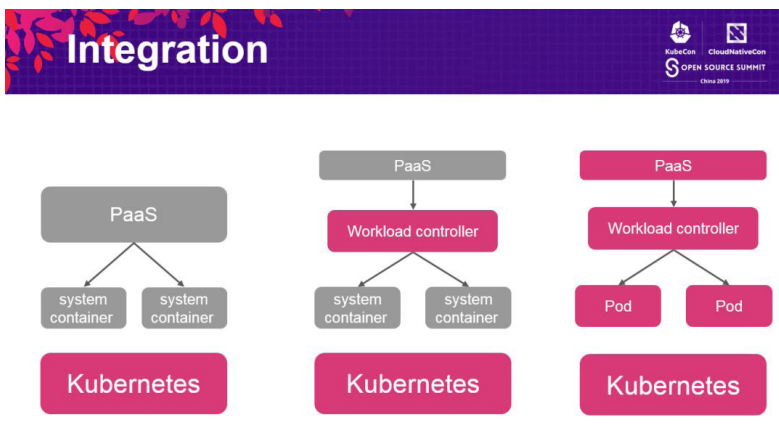
<b>作业管理</b>	Streaming & Messaging	<b>应用定义与镜像构建</b>	CI/CD
<ul style="list-style-type: none"> <li>• Kruse 作业管理</li> <li>• KubeFlow/Arena AI</li> <li>• Argo pipeline 共建</li> </ul>	<ul style="list-style-type: none"> <li>• 云原生 RocketMQ</li> </ul>	<ul style="list-style-type: none"> <li>• Helm 中国站 (App Hub)</li> <li>• 应用 YAML 托管</li> <li>• 云原生应用管理工具</li> </ul>	<ul style="list-style-type: none"> <li>• 云原生持续交付 (CD) 插件</li> </ul>
<b>调度编排</b>	<b>云原生存储</b>	<b>服务发现与RPC</b>	Service Mesh + Gateway
<ul style="list-style-type: none"> <li>• Kubernetes 核心上游投入</li> <li>• Kubernetes 多租户插件</li> <li>• Kubernetes 阿里云集成</li> <li>• GPU Sharing 共享调度</li> </ul>	<ul style="list-style-type: none"> <li>• etcd + etcd operator</li> </ul>	<ul style="list-style-type: none"> <li>• Dubbo</li> <li>• Nacos</li> </ul>	<ul style="list-style-type: none"> <li>• Envoy 增强插件</li> <li>• Tengine</li> <li>• Istio 上游共建</li> </ul>
<b>多云</b>	<b>无服务器框架</b>	<b>容器运行时</b>	<b>工具</b>
<ul style="list-style-type: none"> <li>• Kubernetes 多云插件</li> </ul>	<ul style="list-style-type: none"> <li>• X-Ray Serverless 框架</li> <li>• Virtual Kuberet</li> </ul>	<ul style="list-style-type: none"> <li>• Containerd 核心上游投入</li> <li>• PouchContainer</li> </ul>	<ul style="list-style-type: none"> <li>• ChaosBlade</li> <li>• Dragonfly</li> </ul>
<b>云集成</b>			
<ul style="list-style-type: none"> <li>• Alibaba Cloud controller manager</li> </ul>	<ul style="list-style-type: none"> <li>• CNI Terway</li> <li>• CSI</li> </ul>	<ul style="list-style-type: none"> <li>• Cluster Autoscaler</li> <li>• Cluster API</li> </ul>	<ul style="list-style-type: none"> <li>• 安全集成 RAM Authenticator</li> <li>• KMS plugin</li> </ul>

## 追求极致，优化扩展性和规模性

弹性和规模性，这是支撑阿里巴巴各种类型的复杂场景以及流量高峰的关键因素。

经过不断打磨，阿里巴巴在 Kubernetes 规模与性能上取得了显著的成果：将存储 object 的数量提升 25 倍，支持的节点数从 5000 提升到上万，在端到端调度延迟从 5s 变为 100ms 等等。其中有不少工作在阿里巴巴和社区中共同开展，而这些研发成果都已经贡献给社区，我们期望其他企业及开发者也可以享受阿里巴巴规模所带来的技术红利。

阿里巴巴持续优化性能，可以分为以下四个维度：工作负载追踪、性能分析、定制化调度、大规模镜像分发。首先对工作负载调度有完整的追踪、重放机制，其次将所有性能问题的进行细致分析，逐一攻克技术瓶颈。Kubernetes 本身的可定制性很强，阿里巴巴针对自身业务场景沉淀了定制化的调度能力和镜像分发系统。开源 Dragonfly 项目脱胎于双十一，具备极强的镜像分发能力。数十个超级集群，每个超级集群具有数万节点，数百万的容器。



阿里巴巴落地 Kubernetes 可以分为三个阶段：首先通过 Kubernetes 提供资源供给，但是不过多干扰运维流程，这系统容器是富容器，将镜像标准化与轻量级虚拟化能力带到了上面的 PaaS 平台。第二步，通过 Kubernetes controller 的形式改造

PaaS 平台的运维流程，给 PaaS 带来更强的面向终态的自动化能力。最后把运行环境等传统重模式改成原生容器与 pod 的轻量模式，同时将 PaaS 能力完全移交给 Kubernetes controller，从而形成一个完全云原生的架构体系。

## 如何解决云原生的关键难点

阿里巴巴云原生的探索，起步于自研容器和调度系统，到如今拥抱开源的标准化技术。对于当下开发者的建议是：如果想构建云原生架构，建议直接从 Kubernetes 入手即可。一方面，Kubernetes 为平台建设者而生，已经成为云原生生态的中流砥柱，它不仅向下屏蔽了底层细节，而且向上支撑各种周边业务生态；另一方面，更重要的是社区中有着越来越多围绕 Kubernetes 构建的开源项目，比如 Service Mesh、Kubeflow。

那么作为过来人，阿里有哪些“避坑指南”呢？



云原生技术架构演进中最为艰难的挑战，其实来自于 Kubernetes 本身的管理。因为 Kubernetes 相对年轻，其自身的运维管理系统生态尚不完善。对于阿里而言，数以万计的集群管理至关重要，我们探索并总结了四个方法：Kubernetes on Kubernetes，利用 K8s 来管理 K8s 自身；节点发布回滚策略，按规则要求灰度发

布；将环境进行镜像切分，分为模拟环境和生产环境；并且在监控侧下足功夫，将 Kubernetes 变得更白盒化和透明化，及早发现问题、预防问题、解决问题。

另外一个关键技术问题是 Kubernetes 的多租户管理。相比于 namespace 扩展性差和命名冲突等限制，可以在 Kubernetes 之上建立虚拟集群。在提高扩展性的同时，能够做到 API 层面的强隔离，通过 syncer 链接虚拟集群和真实集群，在 node 添加 agent，达到更好的多租管理和更好的利用。

此次 KubeCon 大会上，阿里云重磅公布了两个项目：Cloud Native App Hub——面向所有开发者的 Kubernetes 应用管理中心，OpenKruise——源自全球顶级互联网场景的 Kubernetes 自动化开源项目集。

**OpenKruise 开源地址：**<https://github.com/openkruise/kruise>

**Cloud Native App Hub：**<https://developer.aliyun.com/hub>

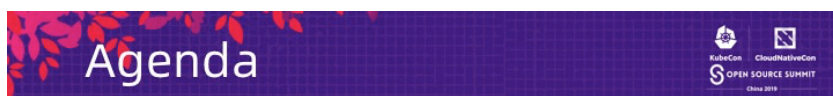
云原生应用中心 (Cloud Native App Hub)，可以简单理解为 Helm 应用中国镜像站，方便用户获得应用资源，并大大简化了 Kubernetes 部署安装一个应用的步骤；OpenKruise/Kruise 项目致力于成为“云原生应用自动化引擎”，解决大规模应用场景下的诸多运维痛点。这次沙龙首秀，开发者们体验了从云原生应用中心快速下载应用，并通过带状态 pod 原地升级、sidecar 容器注入、节点镜像预热等三个场景，实际体验了 Kruise 强大的自动化运维能力。

值得一提的是，OpenKruise 项目源自于阿里巴巴经济体过去多年的大规模应用部署、发布与管理的最佳实践；源于容器平台团队对集团应用规模化运维，规模化建站的能力；源于阿里云 Kubernetes 服务数千客户的需求沉淀。从不同维度解决了 Kubernetes 之上应用的自动化问题，包括部署、升级、弹性扩缩容、QoS 调节、健康检查，迁移修复等等。

## 1-5-10: 如何快速恢复大规模容器故障

阿里云容器平台技术专家 宁拙

### Agenda



- ① Definition of 1-5-10
- ② An incident replay
- ③ How to 1-5-10--offline
- ④ How to 1-5-10--online

在云计算时代，基于容器的应用规模越来越大，随之而来的容器故障也越来越多，故障原因有许多，比如人工误操作，硬件故障等等。如何在不提升成本的前提下提高稳定性，成为大家都要面临的挑战。在阿里巴巴，运行着数百万的容器，积累了大量素材，经过分析提炼之后，提出 1-5-10 的目标。

今天的话题就是 1-5-10，将分为 4 个阶段来讲述：

第一是介绍一下 1-5-10 的定义；

第二是我们来复盘 review 一次故障；

第三和第四就是如何达成 1-5-10: 分别是通过线下和线上两个维度的改进来达成 1-5-10 ；

## 1-5-10



1. 1 分钟发现问题
2. 5 分钟定位问题
3. 10 分钟恢复

为什么是 1-5-10 呢？基于我们的一些历史数据分析，部分故障是能做到 1-5-10，但依然还有不少故障是超出了 1-5-10。

所以呢，我们制定了 1-5-10 的目标，牵引大家一起提升故障发现，定位以及恢复的能力，实现缩短故障时长、减少故障影响，从而达到让集团没有重大故障的目的。现在定义和目标明确了，接下来我们就看一次故障的时间线，看看是否做到了 1-5-10。



- ① Definition of 1-5-10
- ② An incident replay
- ③ How to 1-5-10--offline
- ④ How to 1-5-10--online

## 某模拟故障的时间线



- 15:17 Starts agent upgrade, 3 batches, first batch x nodes.
- 16:30 Starts to upgrade second batch xx nodes.
- 16:55 The new feature of agent is activated.
- 17:30 Internal monitoring systems began generating alerts.
- 17:39 Starts to upgrade last batch xxx nodes.
- 17:47 xxxx pods are impacted by agent.
- 17:50 Stops upgrade job of agent.
- 18:15 Difficult to identify the root cause.
- 18:19 changes load balancing to increase traffic to other clusters.
- 18:28 Roll back agent.
- 19:03 Finish to roll back agent, the status was updated to green.

1-5-10?

在分析这次故障时间线之前，我想先分享一个很好的理念，这个理念来自谷歌 SRE，说的就是事后分析，通过事后分析，我们可以从失败中吸取教训，并且提炼出一系列的方法，可以指导我们在未来避免犯类似的错误。我们现在就用事后分析的方法，来看一下这个故障的时间线：

15 点 17 开始发布，17 点 30 才收到报警，18 点 15 还未能止血，直到 19 点

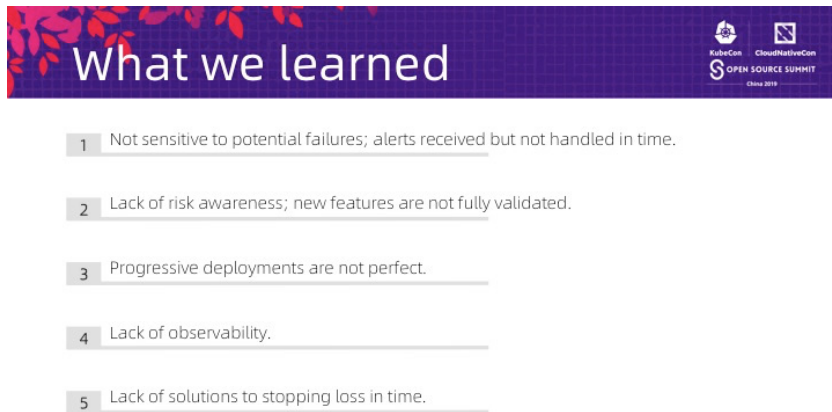


才通过切流 + 回滚的方式恢复。

这里顺便提一句，作为底层基础设施，一个很小的问题，到上层可能就被放大成一个很大的问题，所以我们做底层基础设施的团队，稳定性更是重中之重。稳定性是 1，特性，功能，性能，都是后面的 0，如果 1 不存在了，后面的 0 将一直都是 0。

我们继续讲这个故障，这是一个很典型的因变更发布导致的问题，更明显的是，没有达成 1-5-10 的目标。故障是 15 点发生，到 19 点才完成止血，花了将近 4 小时，离 1-5-10 的目标相去甚远。接下来我们看看在哪些地方可以改进，可以做的更好，可以帮助我们达成 1-5-10。

## 暴露的问题



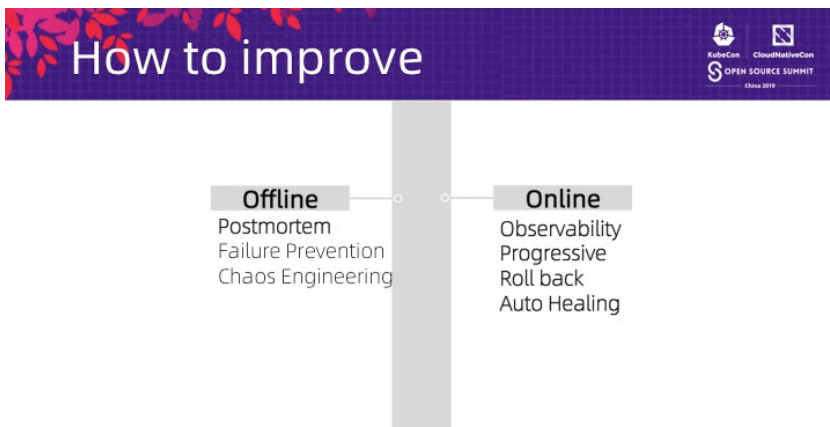
通过对这次故障的分析，我们学到了什么呢？暴露了哪些问题？我们该如何去改进呢？

1. 对故障不敏感，收到报警后，本应该立即停止，但是还继续进行第三批的操作。如果及时停止，问题的影响就不会这么大。
2. 风险意识不足，对新特性的评估不谨慎，没有做全场景的验证就发布。如果在开发、发布的过程中，有足够的风险意识，可能问题就提早暴露了，而不是流入到线上。

3. 灰度没做好，没有严格遵守灰度流程，没有留够观察时间。理论上来说，我们如果留了足够的观察时间，应该是可以发现问题的，问题的规模就不会这么大。
4. 可观测性差，15:17 开始操作，17:30 才收到报警。如果观测性足够好，异常的指标尽早暴露出来，那么应该可以更早的发现问题。
5. 缺乏快速止血的手段，17:30 收到报警，到 18:19 都未能止血，直到 19 点才恢复，这样的恢复速度，实在是难以接受。

针对这些问题，我们将从线下和线上这两个方向来改进。其实用线下和线上这两个词并不准确，暂时没有想到合适的词来描述，我们先看下这两者都包含哪些内容。

## 如何改进



线下：

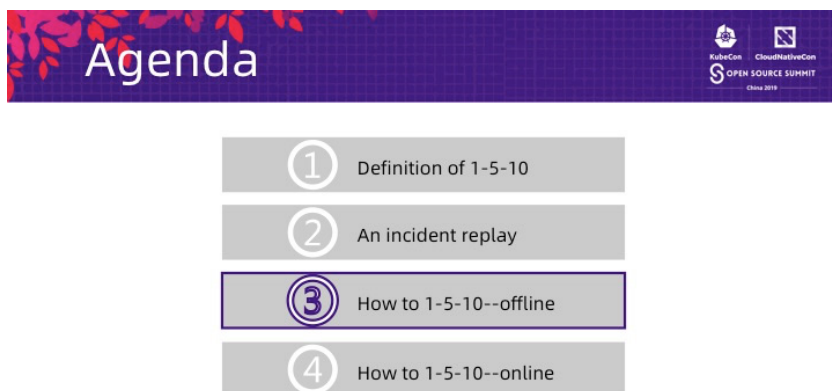
1. 故障复盘，也就是事后分析，从失败中学习，避免再犯。
2. 故障预防，在问题发生之前，就将稳定性作为第一要素考虑进去。
3. 故障演练，通过故障演练，一方面可以发现平时注意不到的问题；另一方面，可以形成正确的应急流程，总结出一套应急预案。这样大家在遇到真正的问题的时候，就不会手忙脚乱。

线上：

主要是通过技术手段来解决稳定性的问题，有四个方面：

1. 可灰度。有正确的流程，正确的工具，来进行变更操作，如果有问题，可以及时发现，避免范围扩大，引发更大的故障。
2. 可止血。光是避免范围扩大还是不够，我们还需要能及时止血，避免问题继续持续下去，尽早恢复。
3. 可观测。可以及时观测到问题，如果问题发生，就能及时发现并通知到相关人员。
4. 自动修复。当集群规模足够大，有一些问题是不可避免的，并且是每时每刻都在发生，对于这样的问题，如果不及时处理，也可能会积少成多形成故障，所以我们还需要一种自动修复的机制，尽快地解决掉这类问题。

## 目录



接下来我们进入第三节，详细看一下，如何通过线下手段，帮助我们做到 1-5-10。

## 故障复盘



阶段Phase	关键时间点Time	要点KeyPoint
发生Occur	故障注入的时间点 故障发生的时间点	故障的根因？以后能不能避免同类问题？
发现Detect	故障发现的时间点	发现途径是什么？自动还是人工？ 是谁发现的？业务方还是责任方？ 能不能快速发现？ 能不能提前发现？
定位Identify	故障相关人接手的时间点 定位出根因的时间点 找出止血方法的时间点	相关人是否及时接手？ 定位根因的方法？看监控，查日志？ 是否通过人工排查才找出止血方法？
恢复Recover	系统恢复的时间点 业务恢复的时间点	恢复的方式是什么？回滚或是其他？ 执行恢复操作花了多长时间？ 未来能不能快速恢复？

既然我们的目标是 1-5-10，那么我们可以针对几个关键的时间点来进行复盘，找出问题所在，制定相应的措施。

我们将故障分为 4 个阶段来分析。

第一个阶段，故障发生，有两个关键的时间点，故障注入和故障发生的时间，review 的时候要重点关注，故障的根因是什么？以及以后能不能避免同类的问题？这也是我们做复盘的重要目的之一。

第二个阶段，故障发现。这个阶段的关键就是故障发现的时间点，review 的时候要关注：

1. 故障发现的途径是什么？是人工发现还是自动发现？能不能做到自动发现？
2. 故障是由谁发现的？是业务方，还是责任方？如果是业务方发现的，之后能不能责任方自己发现？而不是等业务反馈才知道出了问题。等到业务方反馈，那么就远远超出 1 分钟了。
3. 故障能不能快速发现？能不到做到 1 分钟之内发现？
4. 故障能不能提前发现？在故障真正爆发之前，就发现，对于整个业务的稳定性的收益，是巨大的。

第三个阶段，故障定位。相对来说，这个阶段，耗时是最多的，我们需要特别重视。需要关注 3 个时间点，故障相关人接手的时间点，定位出根因的时间点，找出止血方法的时间点。review 的要点：

1. 相关人是否及时接手？没有接手的原因是什么？节假日？通知不及时？责任不明确？
2. 定位根因的方法是什么？是通过监控数据发现，还是通过查日志等需要耗费大量时间的操作发现的？以后可不可以做到自动发现？只有做到自动发现，5 分钟这个目标才有可能达成。
3. 是否通过人工排查才找出止血方法？有没有不需要人工排查分析，就能更快速的止血方法？

第四个阶段，故障恢复。需要关注两个时间点，系统恢复的时间，业务恢复的时间。这里需要关注

1. 恢复的方法是什么？回滚，切流，重启，隔离，新发布？
2. 执行恢复的操作花了多久时间？有没有更快的方式？
3. 未来可不可以快速的回复？

其实就是要求我们，要提前准备好低成本，可快速完成的应急预案。

## 故障预防



Stability is vital in the process of design, development, validation and deployment.

先说结论：稳定性要贯穿产品设计，产品研发，产品验证和产品准入各个环节。

具体表现在：

1. 软件工程，从代码质量入手，做好测试，验证。
2. 稳定性是什么？不是完全不发生问题，而是能够将问题影响降到最小。所以，这就要求我们，所有的线上异常，都要能自动化发现，自动化处理，才能对客户影响减少到最小。在产品研发阶段我们不仅需要适配性能，功能，我们也需要做到可监控，充分考虑发布的场景，在研发阶段就需要考虑清楚每一个异常应该怎样隔离。对于做不到的产品，SRE 有权拒绝上线。

## 故障演练



By introducing faults to the system while it's running and interrupting its normal workloads, we can discover some potential bugs and improve the stability and resiliency of our system.

不演练，永远不知道哪里会出问题。

质量不是测出来的，故障演练本身并不能够直接提高系统的健壮性，但是可以有效衡量和推动系统健壮性提高。

合理地、科学地逐步进行故障注入和混沌工程建设，有助于了解当前系统的弱点做到知己知彼，有助于帮助大家认识到系统的薄弱点反推设计，也有助于团队工程素质、设计能力和应急能力的整体提高，是一个很好的练兵场。

## 目录



- ① Definition of 1-5-10
- ② An incident replay
- ③ How to 1-5-10--offline
- ④ How to 1-5-10--online

接下来我们将一起看下，如何通过线上的技术手段来解决稳定性问题。

## 可灰度



每一次规模化的操作，都必须经过灰度。在灰度的过程中，做好观测，如果发现问题，要快速解决。那么，什么样的灰度方案比较好呢？当然并没有完美的灰度方案，只有最适合的。

我们是这样做的：将机房分为两大类——日常机房和线上机房。从机器重要性来看，我们分为核心业务和非核心业务。比如说一些线下新零售的业务，直接与客户在

线下打交道，这个就是非常重要的业务，容不得半点差池，所以我们在灰度的时候，一般都是先避开这些核心业务。

具体的灰度节奏是：从日常机房开始，先对非核心机器进行变更，按照 1-10-50-100 的节奏，留够观测时间；然后是线上机房，同样的节奏。在灰度的过程中，发布系统会从监控系统取数据，判断发布是否成功，如果出现了异常，则会通知到操作人来决策，是要继续灰度，或是回滚。

## 可止血



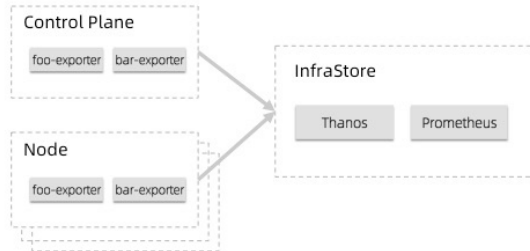
**X** Make sure the strategy of stopping loss is ready before enforcing any changes.

有个原则：在变更之前，就要确定止血策略，未虑胜先虑败。没有止血方案，没有应急预案的变更，一律不许上线。

其实在故障发生的时候，快速止血的方案会有很多。一般我们想到的都是回滚，在除了回滚以外，还有哪些方案？如果故障处理经验丰富的人一定知道，除了回滚，其实还有隔离，降级等多种方案。所以呢，我们在制定止血方案的时候，要充分考虑到止血的可操作性以及耗时，尽量选择一个成本低的止血方案，这样一方面可以加快止血速度，尽快恢复业务，另一方面也容易操作，避免二次故障，线上的操作，一定要最简化。



## 可观测



先简单介绍一下我们当前的监控系统，我们的监控系统是叫 Infra Store，是基于 Prometheus 和 Thanos 来做的。

监控数据分为两大类：

1. 控制面，主要是 api-server，scheduler，etcd 等控制组件的指标和事件。
2. 节点，包括节点，pod，容器，pouch daemon，containerd 等节点上组件的资源类指标（比如 CPU 内存使用情况）等指标。

这两大类的指标，都会都会被 Prometheus 抓取。

但这里边有个比较关键的缺失，我们对宿主机上的异常，检测不够充分，比如说 systemd 有问题，我们需要通过 containerd kubelet 的一些错误才能发现，但这个时候可能为时已晚。

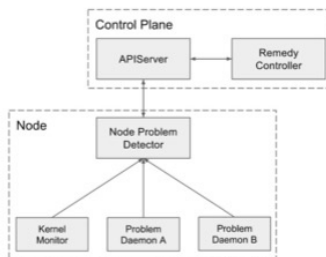
因此我们基于社区的 Node Problem Detector（简称 NPD）做了增强，接下来我们看一下 NPD。

## NPD



**Node Problem Detector:** A DaemonSet detects node problems and reports them to APIServer.

- Enhancement of NPD
- Adapter of Prometheus
  - External monitoring system

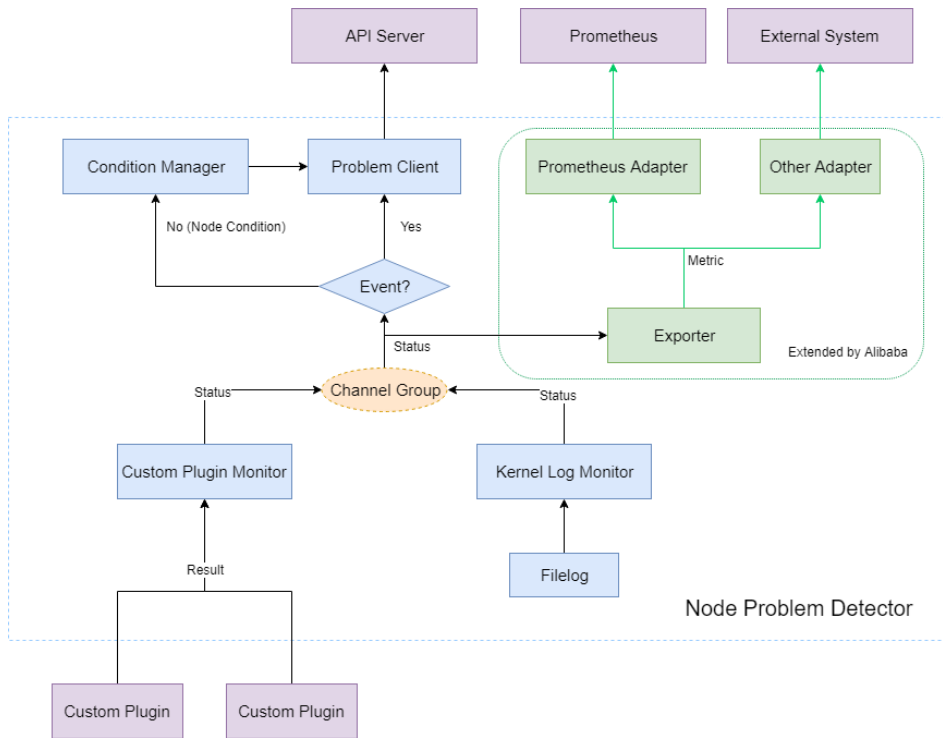


首先看一下社区 NPD 的介绍：发现节点的问题，并上报到 api-server。

NPD 支持多种检测模式，比如说支持自定义脚本，支持 log 分析，将检测的结果，转化成 condition 或 event，上报到 api-server。

前面说到，我们的监控系统是基于 Prometheus 来做的，社区的 NPD 只是将异常事件发送到 API-server，所以上报到 api-server 的方案不满足我们需求的。

因此，我们在社区的基础上，增加了 Exporter 层，收集检测结果，然后将检测结果通过不同的 Adapter，上报到不同的系统。我们现在用的比较多是 Prometheus。



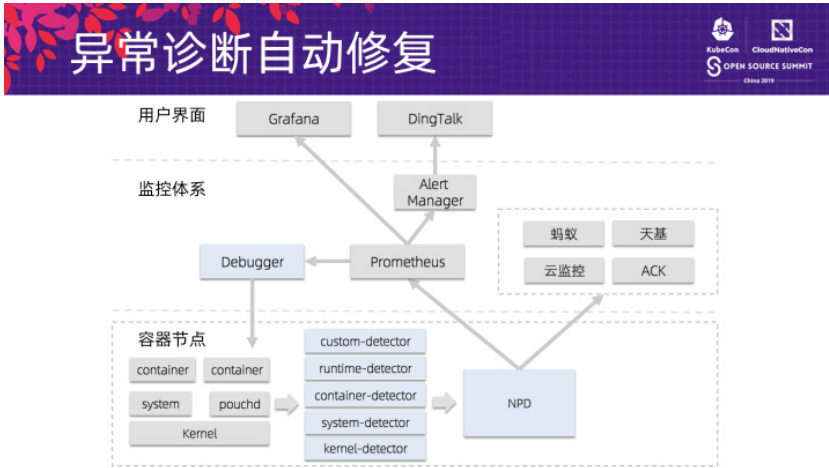
看一下我们的方案，其实比较简单，只是在传递检测结果那里，增加了一个通道。

在后来，我们和 NPD maintainer 交流之后，社区也开始在做类似的事，具体可以看 PR。<https://github.com/kubernetes/node-problem-detector/pull/275>

上层的发布系统，监控报警系统，通过 NPD 上报的异常信息，已有的节点 agent 和控制面的指标，得到了更多的输入，可以发现更多的问题，可以更快的发现问题。

但仅仅发现和上报问题，其实是不够的，还有一些比较小的问题，是完全可以自动修复的，还有一些潜在的问题，是可以通过一定的分析提前预测出来的。接下来我们看看 debugger 的自动修复和异常预测。

## 异常诊断自动修复



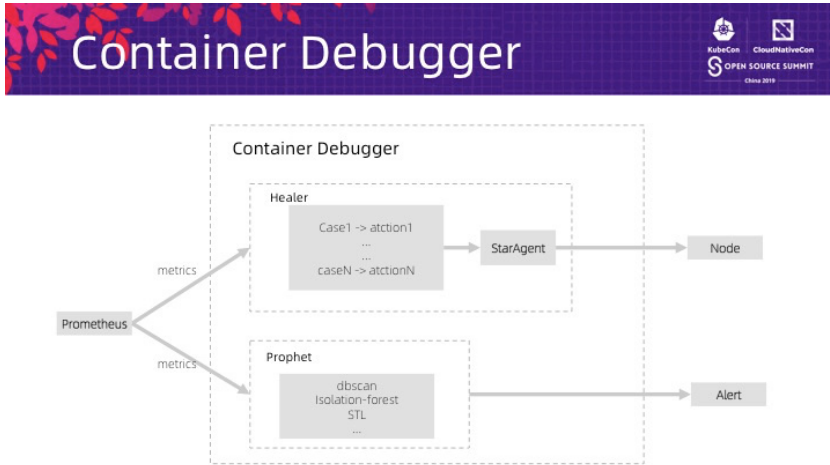
通过 NPD，我们增加了许多异常检测项，可以获取到节点更多的状态，拿到了很多异常的信息。

如何利用这些信息呢？我们开发了一个叫 container debugger 的组件，debugger 基于这些异常信息，以及我们构建的知识库，每个异常都有对应的一个修复方案，发现异常就执行自动修复。

比如修复节点上的网卡问题。简单描述一下这个问题的症状，大家都知道，容器的网络离不开 veth pair，有这么一种情况，veth 在宿主机节点的这一端，并没有挂到网桥上，这样容器的网络就有问题，所以呢，我们增加了一个检测项，用于发现这类异常，然后增加了一个 case，如果发现这个问题，就将 veth 再挂回去。

像这类比较简单的问题，都可以通过自动修复来解决，但如果遇到实在无法解决的问题，就只能将容器迁移到别的节点。

## 自动修复与异常预测

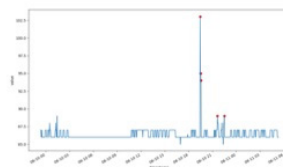


debugger 是如何运行的呢？可以分为两部分：自动修复和异常预测。

第一部分，自动修复。前面我们说到，每个检测项都被转换成了指标，汇总到 Prometheus。debugger 有个 healer 模块，会自动去 Prometheus 取指标，如果某个指标异常，并且符合 case 的定义，healer 就会调用 staragent 到节点上去执行修复任务。

第二部分，就是我们才开始做的，是关于异常预测。大概的原理，就是基于一些无法完全通过设置阈值来做告警的指标，我们通过一些算法，来做分析，提前发现一些潜在的问题。如果确定是风险比较高，则会使用报警通道进行报警，提前人工排查和干预。

## Prophet



现在还不是特别准确，只能简单看一下预测的一些结果。

# 阿里巴巴利用 K8S、Kata 容器和裸金属服务器构建无服务器平台

阿里云容器平台技术专家 张翼飞 (悟鹏)

阿里云容器平台高级开发工程师 唐华敏 (华敏)

## 前言

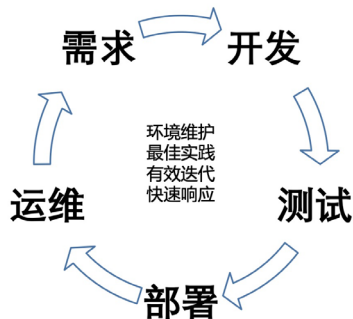
Serverless 是近一年来比较火的概念，在理解这个概念和使用相应的服务之前，需要理解：

- 现状中的痛点是什么
- Serverless 解决的问题域是什么
- Serverless 的解决方案是什么
- 如何做的更好

本次分享将探讨上述问题，并针对解决方案中核心的安全容器 Kata Containers 做深入分析。

## 现状中的痛点

先简单回顾下开发的常见流程：



从上图可看出，服务的开发、运维是个持续迭代的过程，在这几个阶段，通常都会有如下问题需要处理：

- 开发、测试、预发、线上的环境维护，包括准备、调整等
- 最佳实践的应用
- 有效进行迭代
- 快速进行响应

上述这些问题很多都是重复性的，且在每个迭代中都会重复进行。纵览一次迭代中不同阶段和事项的耗时，往往投入到业务逻辑开发的耗时占比会很小，大部分的时间是在处理环境维护、运维管理工作，其次是在开发阶段应用最佳实践。

## Serverless 解决的问题域

抽象下上述描述的问题域：

- 运行 / 运维成本高
- 开发成本高

Serverless 主要解决的是上述两个问题：





## Serverless 的解决方案

Serverless 是如何解决上述问题的？

先看来自维基百科中对 Serverless 的定义：

Serverless computing is a cloud-computing execution model in which the cloud provider runs the server, and dynamically manages the allocation of machine resources. Pricing is based on the actual amount of resources consumed by an application, rather than on pre-purchased units of capacity.

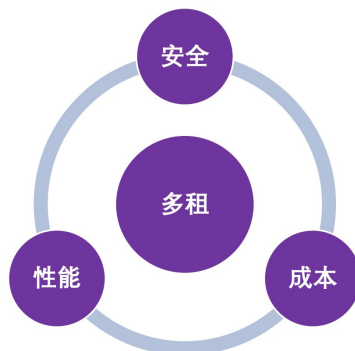
平台方维护资源池，用户可以动态申请资源，解决用户的环境准备和维护问题，降低用户的运维成本。平台方针对用户使用的资源进行计费，降低用户服务的运行成本。平台方提供诸如无侵入性的日志、监控、告警等常规运维服务，进一步提供服务副本数保障、网络管理、扩缩容等确保服务高可用的服务，进一步降低用户服务的运行和运维成本。

平台还可以针对服务开发过程中的通用逻辑进行抽象，使得用户更集中精力在业务逻辑的实现方面，降低服务的开发成本。

## Serverless 平台的搭建

那么如何搭建 Serverless 平台呢？

先思考下 Serverless 平台的挑战，关键有这几方面：



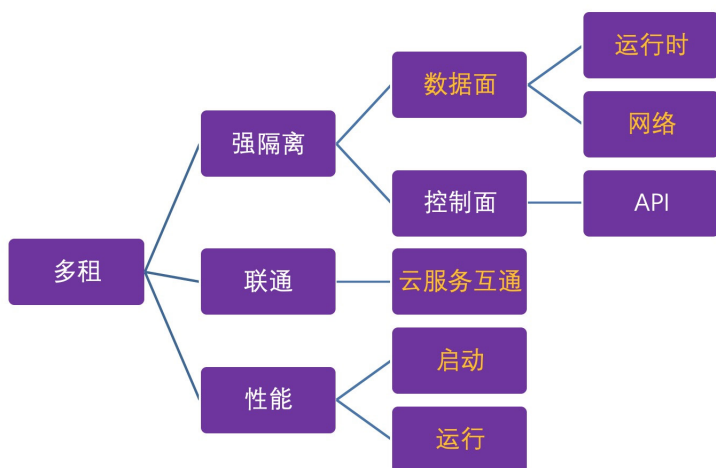
**安全** 强调的是 Serverless 平台上的租户的服务安全，不同租户间不能通过非法方式进行访问。

**成本** 强调是平台方需要尽可能降低运维平台的成本，虽然单个应用对应单个虚拟机的模型可以很大程度上解决**安全**问题，但平台上需要承担很大的成本。

**性能** 强调的是运行在 Serverless 平台上的租户服务，调度和运行性能需要满足租户的需求。

解决上述挑战的关键在于如何支持**多租**，即不同租户的服务会运行在同一个节点上，需要实现**硬多租**。

细化下**多租**相关的维度：

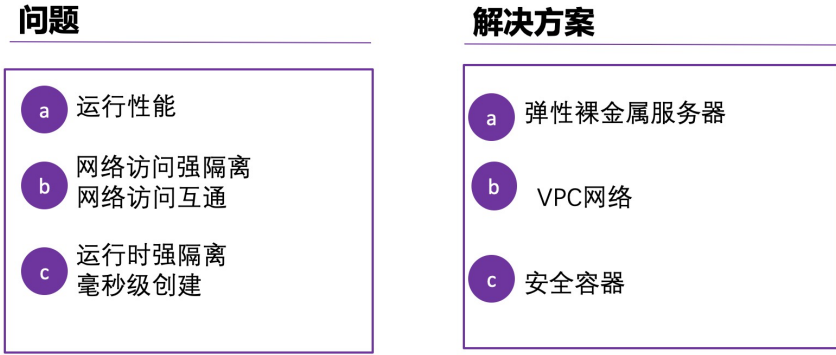


**强隔离**是**多租**需要满足的核心需求，包括**数据面**和**控制面**两个层面，其中**数据面**的强隔离更为关键。不同租户的服务在运行时不能通过非法手段访问到，网络方面要从二层网络隔离。

服务运行时一般会依赖其他云产品服务，如 RDS、MQ 等，为了提升云上同一租户的云服务之间相互访问的性能，提升安全性，也需要满足**联通**的需求。

除此之外，**启动性能**和**运行性能**是服务运行时的核心诉求，在多租场景下满足**安全**和**成本**的同时，不能降低服务运行时的性能。

针对上述问题，解决方案：

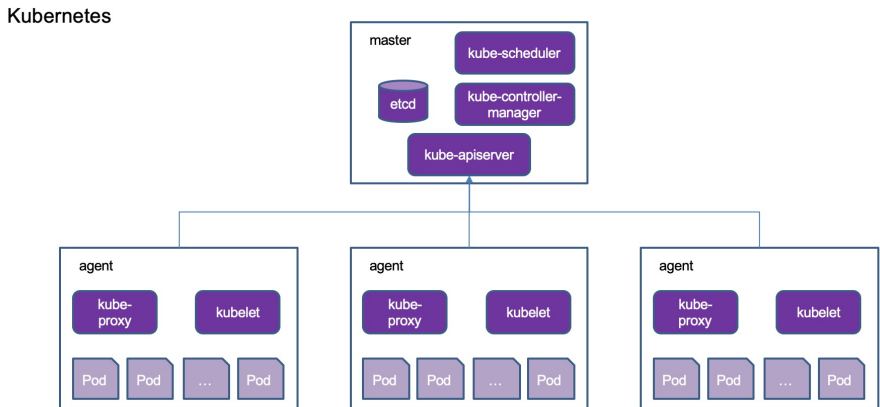


不同租户的服务运行时的强隔离，通过安全容器来解决。每个服务实例运行在轻量级的 VM 中，独占内核，降低共享内核带来的风险。

不同租户的服务网络之间，通过云上的 VPC 强隔离，不同 VPC 内的服务天然不能互通。

弹性裸金属服务器是一款同时兼具虚拟机弹性和物理机性能及特性的新型计算类产品，既保留了普通云服务器的资源弹性，又借助嵌套虚拟化技术保留了物理机的体验。基于弹性裸金属服务器，提升服务运行时对性能的诉求。

我们基于 Kubernetes 做资源管理和服务编排，按照用户的申请进行资源的调度：

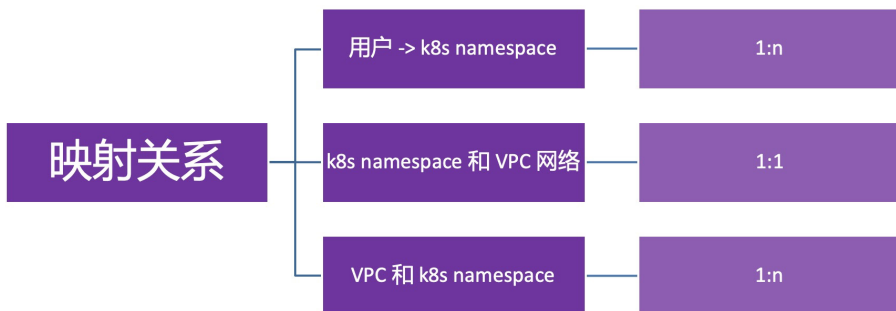


Kubernetes 中有 namespace 的概念，从逻辑上集合一组资源。而在上述多租的设计中，有租户、VPC 的概念，一个租户可以有多个 VPC。为了管理上的方便，需要将此 3 个概念进行映射。

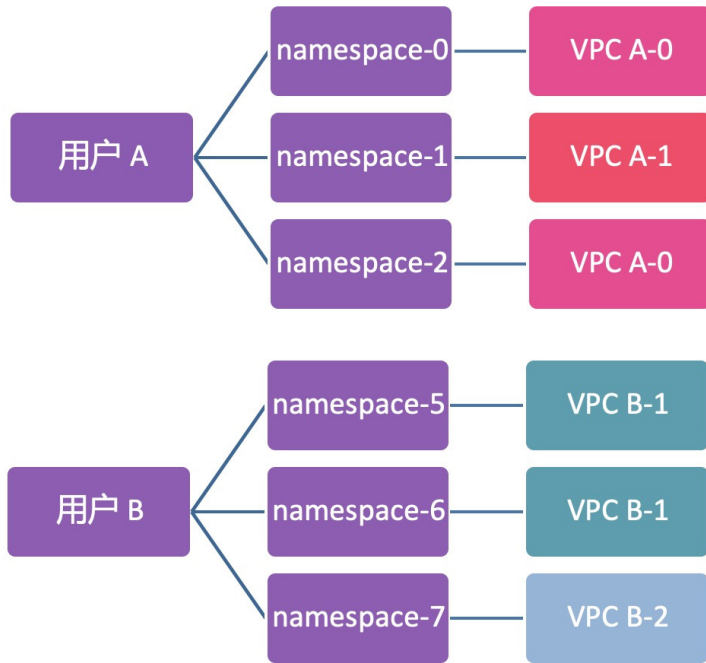
有这样的关系：

- 每个租户有多个 Kubernetes namespace
- 每个 Kubernetes namespace 与一个租户对应
- 每个租户有多个 VPC
- 每个 Kubernetes namespace 与一个 VPC 对应
- 每个 VPC 可以对应多个 Kubernetes namespace

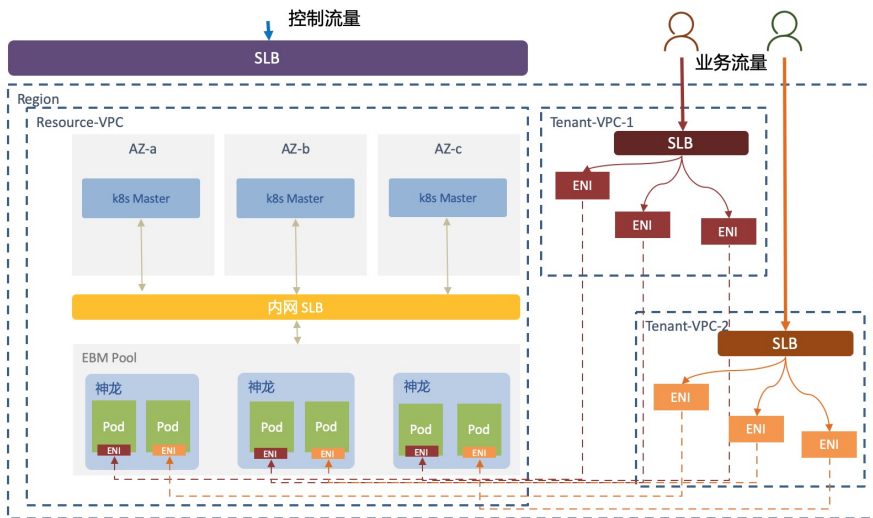
如下图所示：



以两个租户为例，有这样的关系：



这样，完整的 Serverless 平台的架构图如下：



平台层面，基于 Kubernetes 提供资源管理和调度的能力。

Kubernetes 的 master 节点部署在 3 个可用区，提升 master 节点的可用性。

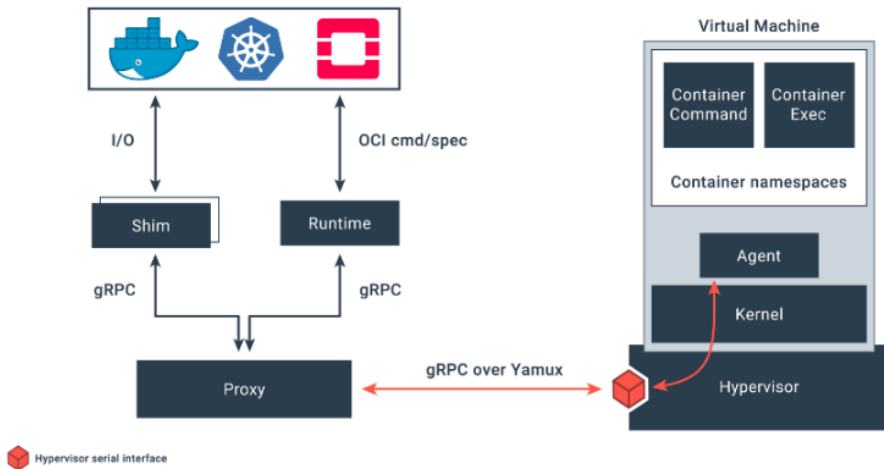
平台维护高性能的弹性裸金属服务器，不同租户的服务混部在节点上，通过安全容器实现不同租户服务间运行时的强隔离，通过 VPC 网络实现网络层面的强隔离和同一租户内服务间的联通性。

用户通过控制链路部署服务，业务流量可通过 SLB 进行访问。

## Kata Containers on serverless 平台

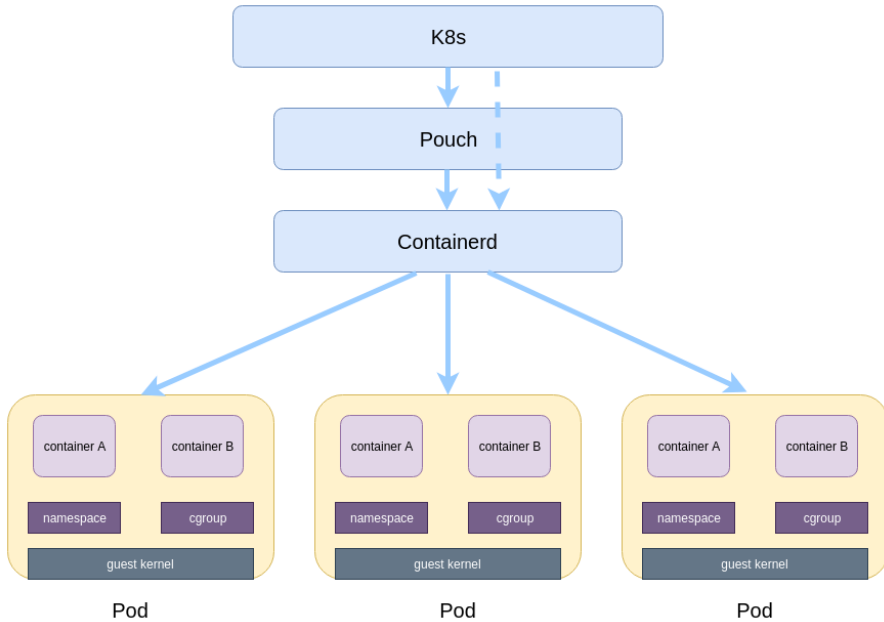
多租户的 serverless 平台上，kata 给容器提供了与 vm 相同的安全性，为用户提供了更好的隔离性。同时有着容器的各种优化，1) 兼容 oci 容器镜像，2) 与 runc 相同的启动速度，3) 比 vm 更轻量的虚拟化

来源于 kata 官网的架构图：



## serverless 平台的 Kata 架构

架构: K8s + pouch cri + containerd + kata-runtime + qemu hypervisor



## 实践中解决的问题

1) kata-runtime 的架构升级，从 shim v1 切换到 shim v2

一方面解决 v1 架构下 kata 组件太多导致的泄漏问题，另一方面可以解决容器优雅退出的问题。

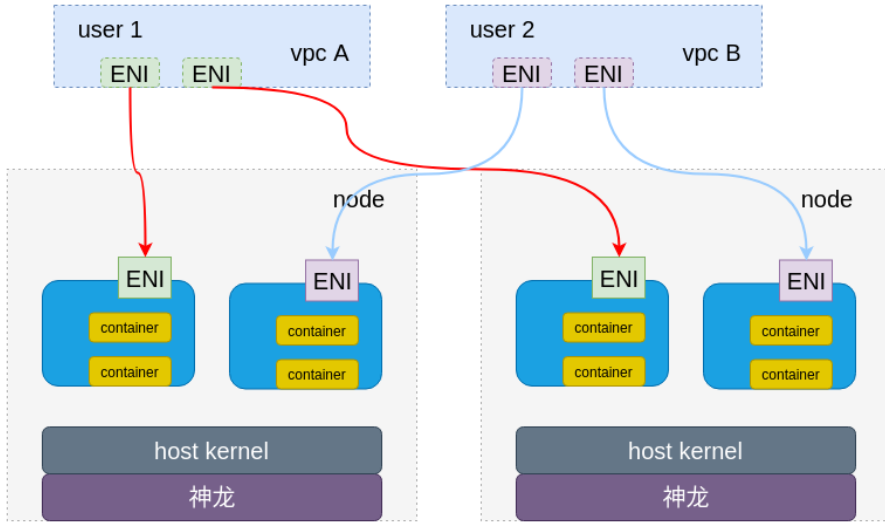
2) 9pfs 的替代方案

kata-runtime 默认的 9pfs 文件系统有几个问题，不兼容 posix 接口，文件操作可能存在问题，io 性能差，会造成 vm hang 住，自研了 qcow2 graph driver 替换 9pfs，并且同步 containerd 社区的 devicemapper 的解决方案。

## 网络和 IO 的优化

kata 容器是集成了虚拟机优势的技术，自然也带来了一些虚机的劣势，在网络和 io 上是比不上 runc 容器的，我们相应的做了优化，使 kata 容器达到生产水平。

## 网络优化



阿里云神龙机型提供了 ENI 网卡，基于这个技术，在 kata 容器中，使用了直通网络模式，性能持平 runc 容器。

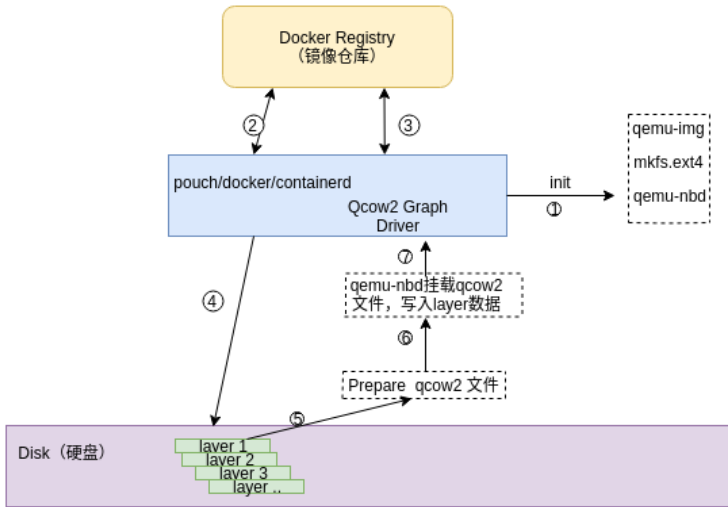
ENI 网卡还有以下优势：Kata 容器的 ENI 属于用户自己的 VPC，不同用户的 VPC 网络二层隔离，同用户 VPC 内不同云服务的网络可以互通

## IO 优化

kata 默认的 9pfs IO 性能差，并且不兼容 posix 接口，文件操作可能会有问题  
使用了 2 种优化手段，用块设备替代掉 9pfs

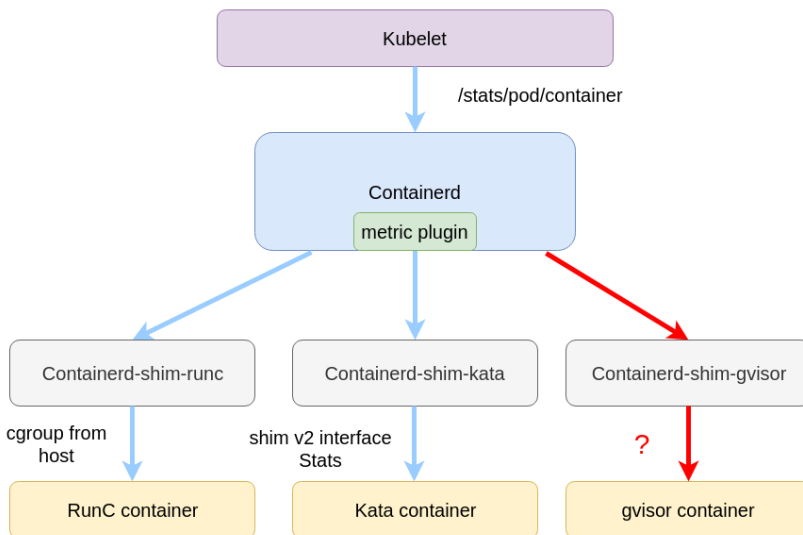
1. qcow2 graphdriver
2. device mapper graphdriver





上图是 qcow2 graphdriver 的设计图，把 pull 下来的镜像保存为 qcow2 格式，启动容器时，通过 qcow2 graphdriver，把 docker 镜像以 qcow2 的形式热插到虚拟机中。

## Kata 容器的监控方案



设计的通用的监控方案，在 containerd 中放一个 metric plugin，不仅适用于 kata 容器的监控，也适用于 runc 或其他使用 containerd shim v2 的运行时的监控。kata 容器的链路中，通过 shim v2 的 Stat 接口获取 kata 容器的监控数据。

## 如何做的更好

对于平台方，需要针对**安全、成本、性能**方面持续迭代，进一步增强安全性，降低平台维护、服务运行的成本，提升服务的调度和运行性能，给租户带来更好的体验，如采用更精细化的资源隔离手段、更低成本高新性能的安全容器。除了极大降低运行和运维成本，还需要针对开发流程做进一步抽象，提供针对函数、程序包等的运行环境服务和运维最佳实践服务，使得租户可以进一步将精力集中在满足业务逻辑方面。

# CafeDeployment: 为互联网金融关键任务场景扩展的 Kubernetes 资源

蚂蚁金服技术专家 昊天

蚂蚁金服高级开发工程师 枫晟

## 背景介绍

Kubernetes 原生社区 Deployment 和 StatefulSet 解决了“服务节点版本一致性”的问题，并且通过 rolling update 实现了滚动升级，提供了基本的回滚策略。对于高可用建设要求不高的“年轻”业务，是一个不错的选择。

但是，在金融场景下，要解决的场景复杂得多，因此我们在金融分布式架构 - 云应用引擎 (SOFAStack-CAFE) 中提出了 CafeDeployment 的云原生模型，致力于解决以下问题：

### 1. IP 不可变

对于很多运维体系建设较为早期的用户，使用的服务框架、监控、安全策略，大量依赖 IP 作为唯一标识而被广泛使用。迁移到 Kubernetes 最大的改变就是 IP 会飘，而这对于他们来说，无异于运维、服务框架的推倒重来。

### 2. 金融体系下的高可用

Deployment/StatefulSet 无法根据特定属性进行差异化部署。而在以同城双活为建设基础的金融领域，为了强管控 Pod 的部署结构（即保证每个机房 / 部署单元都有副本运行，若通过原生组件进行部署，我们不得不维护多个几乎一模一样的 Deployment/StatefulSet），来保证 Pod 一定会飘到指定机房 / 部署单元的 node 上。在规模达到一定程度后，这疑加大了运维管控的复杂度和成本。

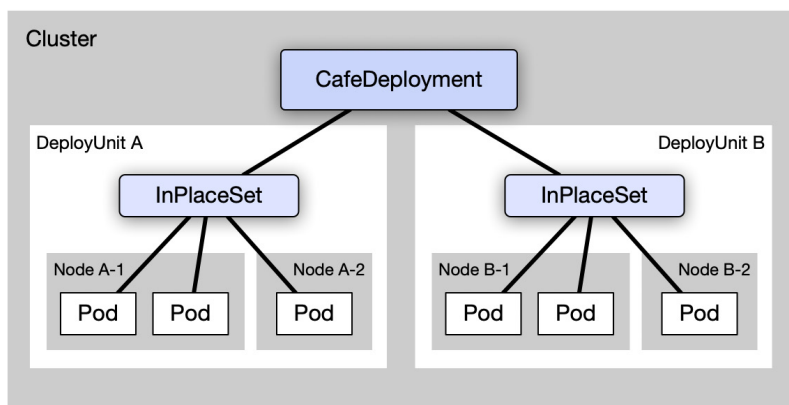
### 3. 灵活的部署策略

Deployment 无法控制发布步长，StatefulSet 虽然可以控制步长，但是每次都

需要人工计算最新版本需要的副本数并修改 Partition，在多机房 / 部署单元的情况下，光想想发布要做的操作都脑袋炸裂。

在面对以上这些问题的时候，我们思考：能不能有一个类似 Deployment 的东西，不仅可以实现副本保持，而且还能协助用户管控应用节点部署结构、做 Beta 验证、分批发布，减少用户干预流程，实现最大限度减少发布风险的目标，做到快速止损，并进行修正干预。这就是我们为什么选择定义了自己的 CRD——CafeDeployment。

## 模型定义



CafeDeployment 主要提供跨部署单元的管理功能，其下管理多个 InPlaceSet。每个 InPlaceSet 对应一个部署单元。部署单元是逻辑概念，它通过 Node 上的 label 来划分集群中的节点，而 InPlaceSet 则通过 NodeAffinity 能力，将其下的 Pod 部署到同一个部署单元的机器上，由此实现 CafeDeployment 跨部署单元的管理。

CafeDeployment 作为多个部署单元的上层，除了提供副本保持，历史版本维护等基本功能，还提供了跨部署单元的分组扩容，分组发布，Pod 调度等功能。模型定义如下：

```

apiVersion: apps.cafe.cloud.alipay.com/v1alpha1
kind: CafeDeployment
metadata:
  .....
spec:
  historyLimit: 20
  podSetType: InPlaceSet # 目前支持底层 PodSet: InPlaceSet, ReplicaSet, StatefulSet
  replicas: 10
  selector:
  matchLabels:
    instance: productpage
    name: bookinfo
  strategy:
    batchSize: 4 # 分组发布时, 每组更新的 Pod 数目
    minReadySeconds: 30
    needWaitingForConfirm: true # 分组发布中, 每组结束时是否需要等待确认
    upgradeType: Beta # 目前支持发布策略: Beta 发布, 分组发布
    pause: false
  template:
    .....
  volumeClaimTemplates: # 用于支持 statefulSet
  serviceName: # 用于支持 statefulSet
  topology:
    autoReschedule:
      enable: true # 是否启动 Pod 自动重调度
      initialDelaySeconds: 10
      unitType: Cell # 部署单元类型: Cell, Zone, None
      unitReplicas:
        CellA: 4 # 固定某部署单元的 Pod 数目
      values: # 部署单元
        - CellA
        - CellB

```

因为我们将大部分的控制逻辑都抽取到上层 CafeDeployment 中, 所以我们重新设计了 InPlaceSet, 将它做得足够简单, 只关注于 “InPlace” 相关的功能, 即副本保持和原地升级, 保持 IP 不变的能力, 模型定义如下:

```

spec:
  minReadySeconds: 30
  replicas: 6
  selector:
    matchLabels:
      instance: productpage
      name: bookinfo
      deployUnit: CellB

```

```
strategy:
  partition: 6    # 控制发布时更新 Pod 的进度
template:
  .....
```

## 功能特性

### 灵活的分组定义

CafeDeployment 支持跨部署单元的分组扩容、Pod 调度、分组发布。分组策略主要分为两种，Beta 分组和 Batch 分组：

- Batch 分组

即根据 BatchSize 将 Pod 分为多个批次，每批中的 Pod 会同时发布。待用户确认（needWaitingForConfirm=true 时）无误时，或当前批次所有 Pod 都 ready 后（needWaitingForConfirm=false 时），则会开始进行下一组的发布。

在分组暂停时，CafeDeployment 会被打上 Annotation: cafe.sofastack.io/upgrade-confirmed=false，用户可通过将 Annotation 的值改为 true，确认当前分组。

- Beta 分组

相比 Batch 发布，会在第一组发布之前多一步 Beta 分组。此组会在每个部署单元内选择一个 Pod 进行发布，以减小错误配置带来的影响。若用户确认无误，可以确认继续，以进入正常的 Batch 发布流程。

## 安全的分组扩容和发布能力

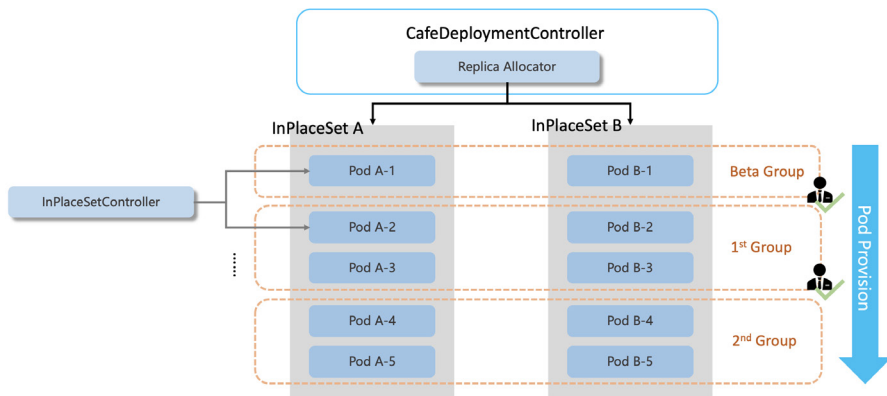
### 分组扩容

为预防不正确的配置造成大量错误 Pod 同时创建、占用大量资源等意外情况出现，CafeDeployment 支持分组扩容，以降低风险。

在有如下配置时，CafeDeployment 会创建两个 InPlaceSet 实例，并开始分组创建（扩容）Pod。

```
spec:
  .....
  replicas: 10                # 副本数为 10
  strategy:
    upgradeType: Beta        # Beta 发布
    batchSize: 4             # 每组 Pod 数为 4
    needWaitingForConfirm: true # 分组暂停
  topology:
    .....
    values: # 两个部署单元, CellA 和 CellB
      - CellA
      - CellB
```

初始时，InPlaceSet 的 replicas 和 partition 都为 0，CafeDeployment 会在之后默认将 10 个 Pod 均分到两个部署单元中，并参考 Beta 发布和 BatchSize 配置，分成 3 组进行，如下图所示：



第一组，为 Beta 分组，会在两个部署单元中各创建一个 Pod。待 Pod 都 ready 后，会要求用户进行确认，方可继续第二组。

第二组，为普通发布，因为 BatchSize=4，所以会在两个部署单元中各创建 2 个 Pod。之后同样需要经过确认才会继续进入第三组。若 CafeDeployment 中的配置 needWaitingForConfirm=false，则在当前批次的所有 Pod 都 ready 后，会自

动进入下一组的发布。

第三组，为最后一组发布，当所有 Pod 都 ready 后，则会结束当前发布。

发布过程中若出现问题，可通过修改 CafeDeployment 的 replicas 值等方式结束当前发布。

## 分组发布

当修改 CafeDeployment 中的 PodTemplate 里的相关配置后，就会触发发布流程。目前同样支持 Beta 发布和 Batch 发布两种类型。当 CafeDeployment 有如下配置时，CafeDeployment 的 10 个 Pod 会被分到三组中进行发布。

```
spec:
  .....
  replicas: 10                # 副本数为 10
  strategy:
    upgradeType: Beta        # Beta 发布
    batchSize: 4             # 每组 Pod 数为 4
    needWaitingForConfirm: true # 分组暂停
  topology:
    .....
    values: # 两个部署单元, CellA 和 CellB
      - CellA
      - CellB
```

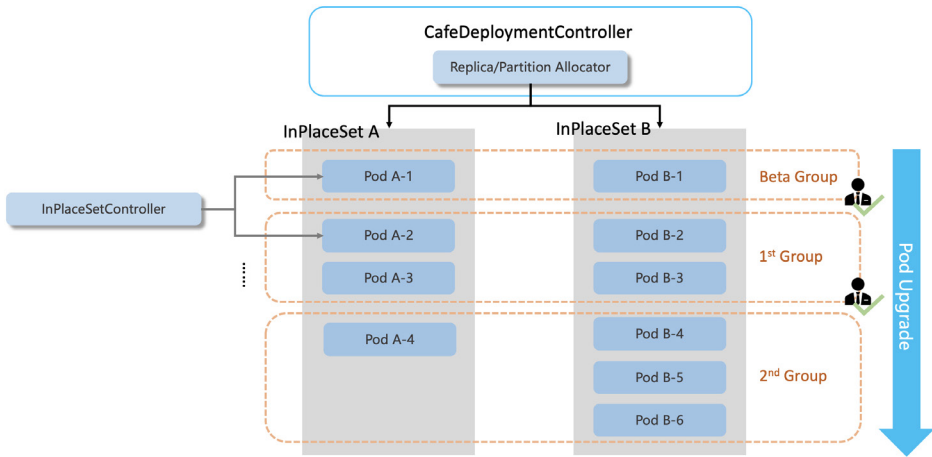
第一组为 Beta 分组，每个部署单元会选择 1 个 Pod 进行发布。

第二组和第三组各有 4 个 Pod。

若当前 Pod 在部署单元的分配不均匀，如下图所示，CafeDeploymentController 也会负责计算并分配对应的配额给两个部署单元，来对发布进度进行统一的调度。

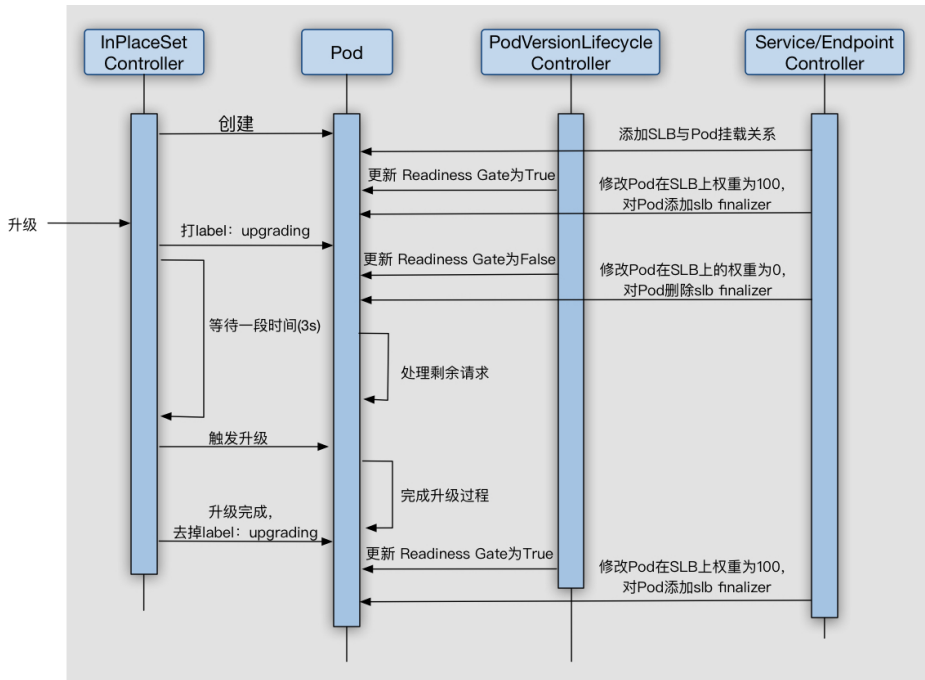
如果配置了分组暂停，则每组结束后都会需要用户进行确认。





## Pod 发布与外部的通信机制

使用 Readiness Gate 作为 Pod 是否可以承载外部流量的标识。在发布前，通过将 Readiness Gate 设置为 False，使得当前 Pod IP 在 Endpoint 上从 addresses 列表转移到 notReadyAddresses 列表；在发布完成后，将 Readiness Gate 设置为 True，Pod IP 在 Endpoint 上又会从 notReadyAddresses 转移到 addresses。相关流量组件通过 Watch Endpoint 上地址变化完成切流和引流，并通过 finalizer 对 Pod 进行打标保护。

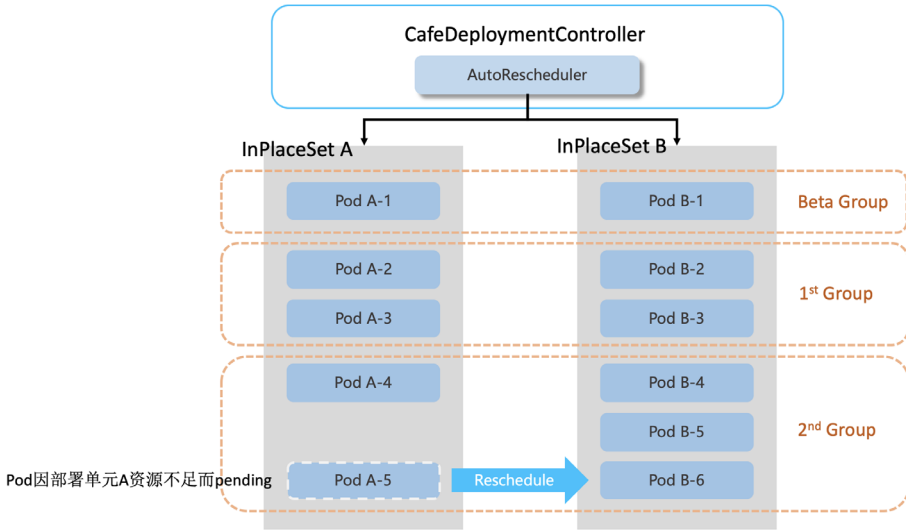


## 自适应的 Pod 重调度

在创建 Pod 的过程中，可能会遇到某个部署单元资源不足的情况，新的 Pod 会一直 Pending。这时如果打开自动重调度功能（如下所示），则 CafeDeployment-Controller 会尝试将 Pod 分配到其他未出现资源紧张的部署单元上。

```
spec:
  topology:
    autoReschedule:
      enable: true           # 是否启动 Pod 自动重调度
      initialDelaySeconds: 10 # Pod 由于资源不足，10 秒后会被尝试重调度
```

在 Pod 部署过程中，如下图所示：



如果在最后一批分组发布的时候，出现 Pod 因资源不足而无法启动的情况，则 CafeDeploymentController 会自动将此 Pod 的配额调度到其他的资源充足的部署单元中。

当然，CafeDeployment 也支持手动指定 Pod 的分配方案，通过修改如下相关配置可以进行精确的指定：

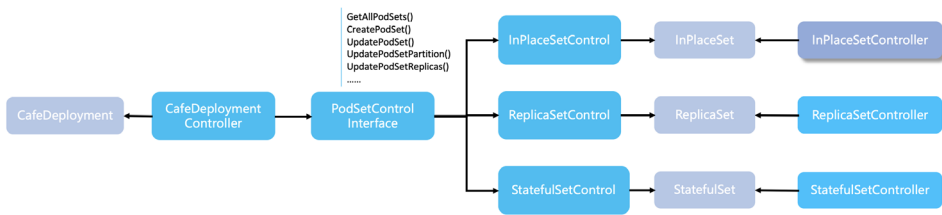
```
spec:
  topology:
    .....
  unitReplicas:
    CellA: 4 # 固定某部署单元的 Pod 数目
    CellB: 10% # 通过百分比指定
  values:
    - CellA
    - CellB
    - CellC
```

这时，CafeDeploymentController 会优先根据指定方案进行 Pod 分配。

## 可适配多种社区工作负载

CafeDeploymentController 本身只提供了发布策略和跨部署单元管理的一个

抽象实现，它对底层的 Pod 集合是通过 PodSetControlInterface 接口来控制。因此，通过对此接口的不同实现，可以保证对接多种 workload。目前已经实现了与 InPlaceSet 和 ReplicaSet 的对接，对 StatefulSet 的对接也在进行中。



因为 CafeDeployment 只负责各种策略的实现，所以并不会对 Kubernetes 原生的功能有任何入侵。

ReplicaSetController、StatefulSetController 会继续履行他们之前的职责，保证各自的特性。

## 总结

CafeDeployment 的设计与实现，并非一日之功，我们走过弯路，也受到过质疑。但我们仍然坚信，在金融场景下需要这样的一种工作负载，因为无论是 Deployment、StatefulSet 还是 InPlaceSet，为了实现高可用和无损发布，都无疑需要付出比 apply yaml 更多的精力，而这些往往都不是一个业务开发所关心的。

目前，CafeDeployment 所提供的各种发布策略，灵活的分组发布，高可用和无损升级的能力已成为了金融云应用发布的重要一环，为产品层提供容器云原生的部署能力，并给我们用户的生产力和效率带来极大提升。后续我们将会继续增强 CafeDeployment 的能力，比如提供更灵活的自定义拓扑结构、机房 / 部署单元内更灵活的部署策略以满足更多的高可用发布场景的需求等。

最后，感谢一直以来紧密合作的 Sigma 团队同学，在相互的交流中迸发出了更多的灵感和想法。

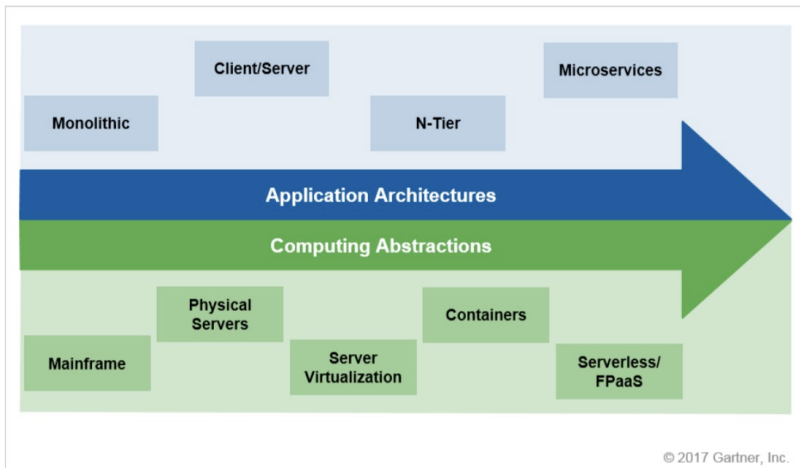
# Serverless 市场观察和落地挑战

蚂蚁金服高级产品经理 隐秀

## 市场观察

当我们回顾云计算的发展历程，会看到基础架构经历了从物理机到虚拟机，从虚拟机再到容器的演进过程。在这大势之下，应用架构也在同步演进，从单体过渡到多层，再到当下的微服务。在变化的背后，有一股持续的动力，它来自于三个不变的追求：**提高资源利用率，优化开发运维体验，以及更好地支持业务发展。**

目前，**Serverless** 已成为云原生社区关注的重点之一，它的发展也不例外。相比容器技术，Serverless 可以将资源管理的粒度更加细化，使开发者更快上手云原生，并且倡导事件驱动模型支持业务发展。从而帮助用户解决了资源管理复杂、低频业务资源占用等问题；实现面向资源使用，以取代面向资源分配的模式。根据 CNCF 在 2018 年底基于 2400 人的一份统计报告，已有 38% 的组织正在使用 Serverless 技术，相比 2017 同期增长了 22%。（数据来源：[CNCFSurvey](#)）



图片来源: Gartner Report: China Summary Translation Evolution of Server Computing – VMs to Containers to Serverless – Which to Use When



同时，对于在企业级应用的生产环境落地 Serverless，各方也有了很多探索和突破。在本周刚结束的 KubeCon China 2019 大会上，Serverless 工作组会议也以此为话题展开了讨论。目前的核心挑战可归纳为：

### 平台可迁移

目前众多平台都推出了自己的 Serverless 标准，包括代码格式、框架和运维工具等，用户既面临较高的学习成本和选择压力，也担心无法在平台之间灵活迁移 Serverless 应用。

### 0-M-N 性能

线上应用对控制请求延迟有严格的要求，因此，用户需要谨慎地验证 Serverless 0-1 冷启动速度、M-N 扩容速度以及稳定性都达到了生产要求。

### 调试和监控

用户对底层资源无感知，只能借助平台能力对应用进行调试和监控，用户需要平台提供强大的日志功能进行排错，和多维度的监控功能时刻了解应用状态。

### 事件源集成

采用 Serverless 架构后，应用往往进行更细粒度的拆分，并通过事件串联。因此用户希望平台能集成大多数通用的事件源，并支持自定义事件，使得触发机制更加灵活。

### 工作流支持

完成某个业务，往往涉及多个 Serverless 应用之间的配合，当数目较多时，用户希望可以用工作流工具来进行统一编排和状态查看，提高效率。

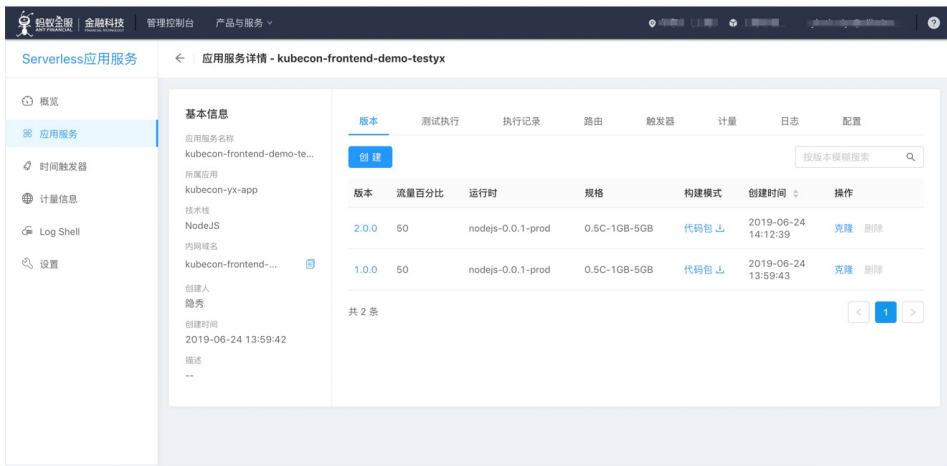
## 蚂蚁实践

**SOFAShield** 致力于通过产品技术解决云上客户实际痛点，沉淀蚂蚁技术实践，帮助用户以高效、低成本的方式迁移到云原生架构。**Serverless 应用服务** (Serverless Application Service, 简称 SOFA SAS) 是一款源自蚂蚁实践的一站式 Serverless 平台。SAS 基于 SOFAShield CAFE 云应用引擎 (Cloud Application Fabric Engine 简称 CAFE)，CAFE 的容器服务已经通过了 CNCF 的

一致性认证，是一个标准的 Kubernetes。



Serverless 应用服务产品在兼容标准 Knative 同时，融入了源自蚂蚁实践的应用全生命周期管理能力，提供了 Serverless 引擎管理、应用与服务管理、版本管理与流控、根据业务请求或事件触发较快的 0-M-N-0 自动伸缩、计量、日志及监控等配套能力。同时结合金融云上客户实际痛点，产品独居匠心的提供独占版与共享版两种形态，以及传统代码包、容器镜像与纯函数三种研发模式，以解决用户的不同需求，降低客户准入门槛。

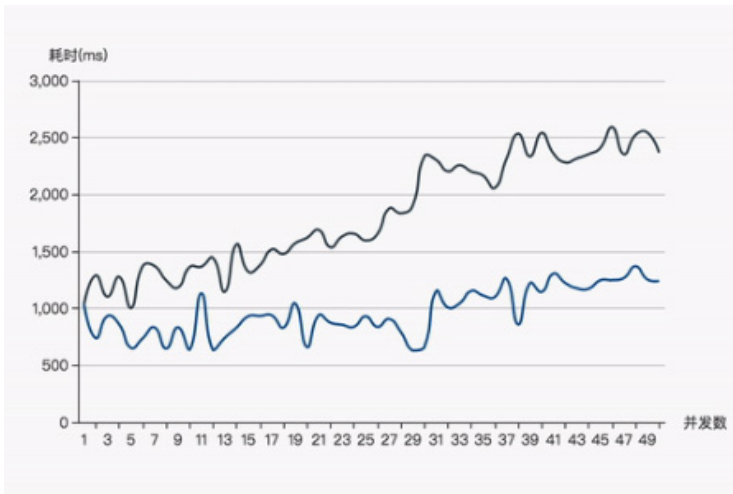


- **一键部署**：用户可以通过代码包或容器镜像的方式一键部署应用并在任意时刻测试执行。
- **引擎管理**：SAS 提供了丰富的引擎全生命周期管理、诊断、监测等能力，为独

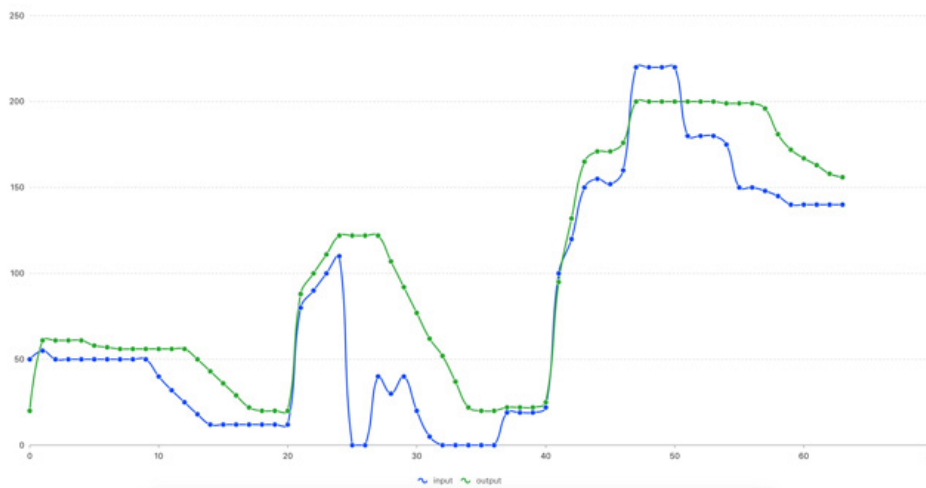


占版客户赋能 Serverless 引擎数据面的全方位管理与运维运营能力。

- **服务及版本**: SAS 提供应用管理、应用服务管理以及版本管理。版本可以采用容器镜像方式部署也可以采用传统 VM 发布模式下的代码包部署, 很多情况下用户代码无需修改也无需编写维护 Dockerfile 即可迁移。
- **0-M-N**: SAS 提供 0-M-N-M-0 的 Serverless 快速伸缩能力, 支持事件触发或流量触发的 0-M, 多种指标的 M-N (如 QPS、CPU、MEM 等等)
- **日志监控计量**: 产品内置了日志、监控、计量等配套设施能力, 帮助用户进行调试和应用状态监控。
- **流量控制**: 基于 SOFAMesh, SAS 提供基本流控能力, 后续会与服务网格进一步深度集成提供大规模多维跨地域及混合云的流控能力。
- **触发器管理**: 产品支持基于常见周期以及秒级精度的 cron 表达式触发器, 可关联并触发无服务器应用, 后续将支持更多 IaaS、PaaS 管控型与数据型事件。



- **性能简析**: 横轴为完全在同一时刻触发冷启的 Java 应用个数, 纵轴为冷启应用的平均与最小耗时。随着压力增大, 50 个 Java 应用同一时刻调度冷启平均耗时 2.5 秒左右, 100 个 Java 应用同一时刻调度冷启平均耗时 3-4 秒, 最短耗时 1.5 到 2 秒。



性能简析：Pooling 快弹慢缩时序算法，池容量和实际单位时间申请量关系可做到如图所示（蓝色为实际申请量，绿色为池容量）

目前产品已顺利支撑**生产环境**小程序 Serverless 模式。同时通过 O-M-N-M-O 的能力在很大程度上降低了小程序的运营成本。在行业客户领域，某保险公司决定近期迁移部分日结前置和长尾应用到 Serverless 产品平台，这也是我们产品又一个重要突破。未来，我们致力于将 SAS 打造成为一个金融级的 Serverless 平台。

# 有效可靠地管理大规模 Kubernetes 集群

蚂蚁金服高级技术工程师 沧漠

## 前言

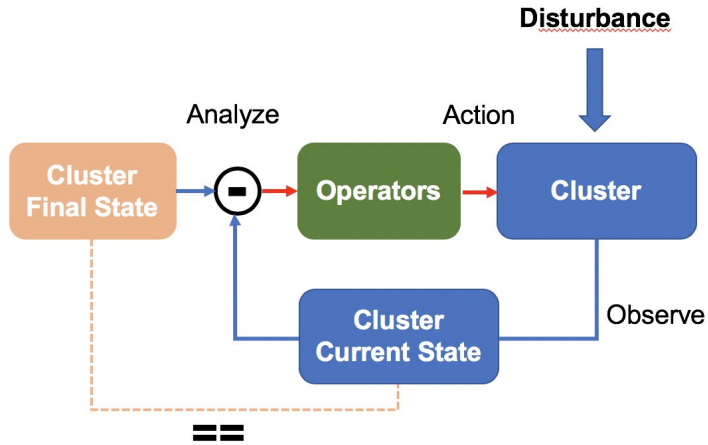
Kubernetes 以其超前的设计理念和优秀的技术架构，在容器编排领域拔得头筹。越来越多的公司开始在生产环境部署实践 Kubernetes，在阿里巴巴和蚂蚁金服 Kubernetes 已被大规模用于生产环境。Kubernetes 的出现使得广大开发同学也能运维复杂的分布式系统，它大幅降低了容器化应用部署的门槛，但运维和管理一个生产级的高可用 Kubernetes 集群仍十分困难。本文将分享蚂蚁金服是如何有效可靠地管理大规模 Kubernetes 集群的，并会详细介绍集群管理系统核心组件的设计。

## 系统概览

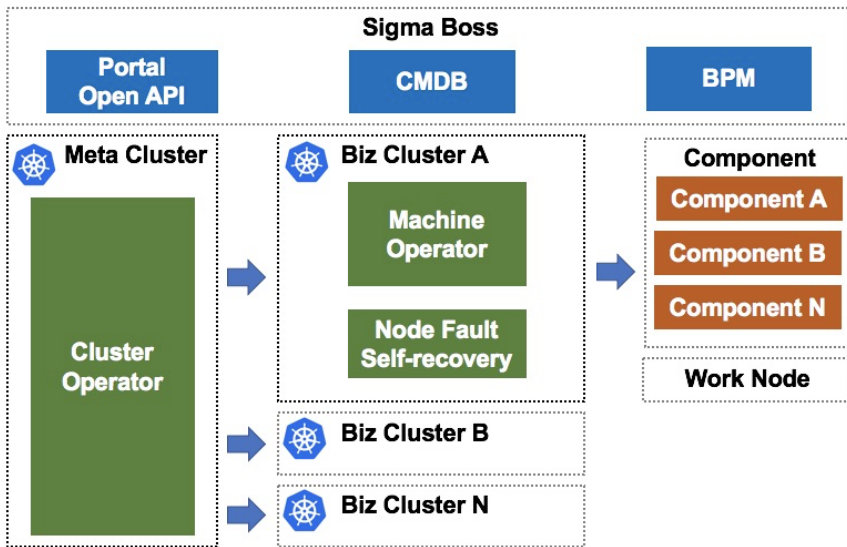
Kubernetes 集群管理系统需要具备便捷的集群生命周期管理能力，完成集群的创建、升级和工作节点的管理。在大规模场景下，集群变更的可控性直接关系到集群的稳定性，因此管理系统可监控、可灰度、可回滚的能力是系统设计的重点之一。除此之外，超大规模集群中，节点数量已经达到 10K 量级，节点硬件故障、组件异常等问题会常态出现。面向大规模集群的管理系统设计之初就需要充分考虑这些异常场景，并能够从这些异常场景中自恢复。

## 设计模式

基于这些背景，我们设计了一个面向终态的集群管理系统。系统定时检测集群当前状态，判断是否与目标状态一致，出现不一致时，Operators 会发起一系列操作，驱动集群达到目标状态。这一设计参考控制理论中常见的负反馈闭环控制系统，系统实现闭环，可以有效抵御系统外部的干扰，在我们的场景下，干扰对应于节点软硬件故障。



## 架构设计



如上图，元集群是一个高可用的 Kubernetes 集群，用于管理 N 个业务集群的 Master 节点。业务集群是一个服务生产业务的 Kubernetes 集群。SigmaBoss 是集群管理入口，为用户提供便捷的交互界面和可控的变更流程。

元集群中部署的 Cluster-Operator 提供了业务集群创建、删除和升级能力，Cluster-Operator 面向终态设计，当业务集群 Master 节点或组件异常时，会自动隔离并进行修复，以保证业务集群 Master 节点达到稳定的终态。这种采用 Kubernetes 管理 Kubernetes 的方案，我们称作 Kube on Kube 方案，简称 KOK 方案。

业务集群中部署有 Machine-Operator 和节点故障自愈组件用于管理业务集群的工作节点，提供节点新增、删除、升级和故障处理能力。在 Machine-Operator 提供的单节点终态保持的能力上，SigmaBoss 上构建了集群维度灰度变更和回滚能力。

## 核心组件

### 集群终态保持器

基于 K8S CRD，在元集群中定义了 Cluster CRD 来描述业务集群终态，每个业务集群对应一个 Cluster 资源，创建、删除、更新 Cluster 资源对应于实现业务集群创建、删除和升级。Cluster-Operator watch Cluster 资源，驱动业务集群 Master 组件达到 Cluster 资源描述的终态。

业务集群 Master 组件版本集中维护在 ClusterPackageVersion CRD 中，ClusterPackageVersion 资源记录了 Master 组件（如：api-server、controller-manager、scheduler、operators 等）的镜像、默认启动参数等信息。Cluster 资源唯一关联一个 ClusterPackageVersion，修改 Cluster CRD 中记录的 ClusterPackageVersion 版本即可完成业务集群 Master 组件发布和回滚。

### 节点终态保持器

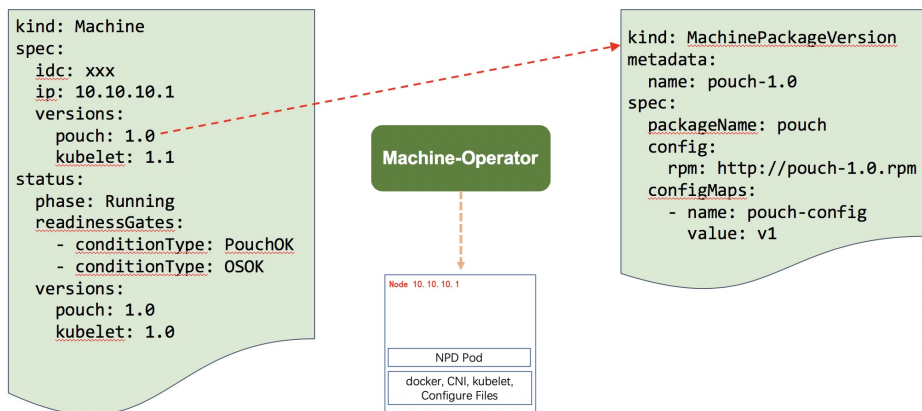
Kubernetes 集群工作节点的管理任务主要有：

- 节点系统配置、内核补丁管理
- docker / kubelet 等组件安装、升级、卸载

- 节点终态和可调度状态管理 (如关键 DaemonSet 部署完成后才允许开启调度)
- 节点故障自愈

为实现上述管理任务，在业务集群中定义了 Machine CRD 来描述工作节点终态，每一个工作节点对应一个 Machine 资源，通过修改 Machine 资源来管理工作节点。

Machine CRD 定义如下图所示，spec 中描述了节点需要安装的组件名和版本，status 中记录有当前这个工作节点各组件安装运行状态。除此之外，Machine CRD 还提供了插件式终态管理能力，用于与其它节点管理 Operators 协作，这部分会在后文详细介绍。

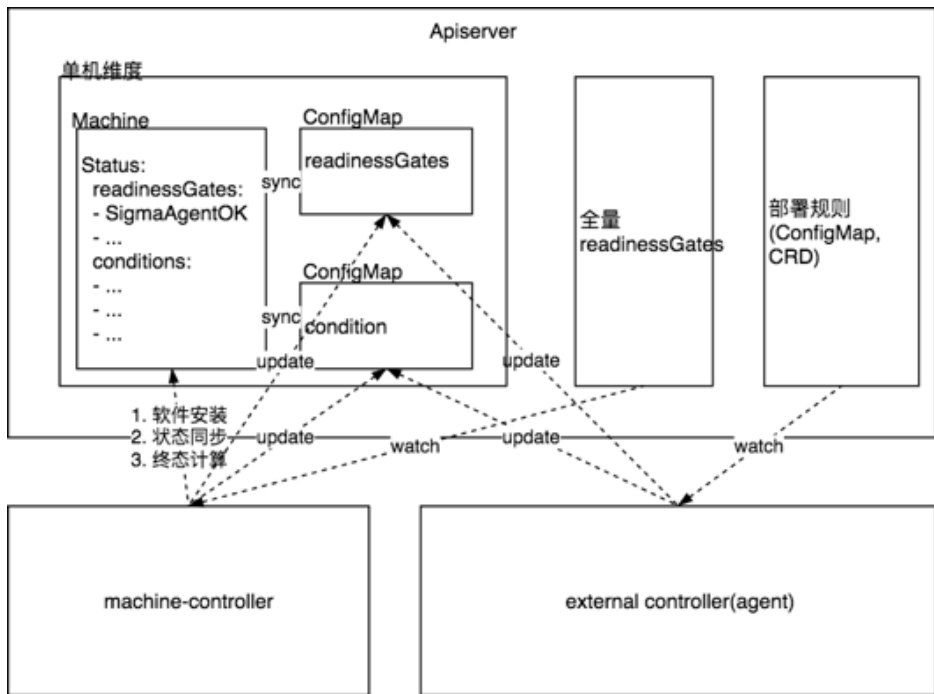


工作节点上的组件版本管理由 MachinePackageVersion CRD 完成。MachinePackageVersion 维护了每个组件的 rpm 版本、配置和安装方法等信息。一个 Machine 资源会关联 N 个不同的 MachinePackageVersion，用来实现安装多个组件。

在 Machine、MachinePackageVersion CRD 基础上，设计实现了节点终态控制器 Machine-Operator。Machine-Operator watch Machine 资源，解析 MachinePackageVersion，在节点上执行运维操作来驱动节点达到终态，并持续守护终态。

## 节点终态管理

随着业务诉求的变化，节点管理已不再局限于安装 docker / kubelet 等组件，我们需要实现如等待日志采集 DaemonSet 部署完成才可以开启调度的需求，而且这类需求变得越来越多。如果将终态统一交由 Machine-Operator 管理，势必会增加 Machine-Operator 与其它组件的耦合性，而且系统的扩展性会受到影响。因此，我们设计了一套节点终态管理的机制，来协调 Machine-Operator 和其它节点运维 Operators。设计如下图所示：



全量 ReadinessGates：记录节点可调度需要检查的 Condition 列表

Condition ConfigMap：各节点运维 Operators 终态状态上报 ConfigMap 协作关系：

1. 外部节点运维 Operators 检测并上报与自己相关的子终态数据至对应的 Condition ConfigMap；

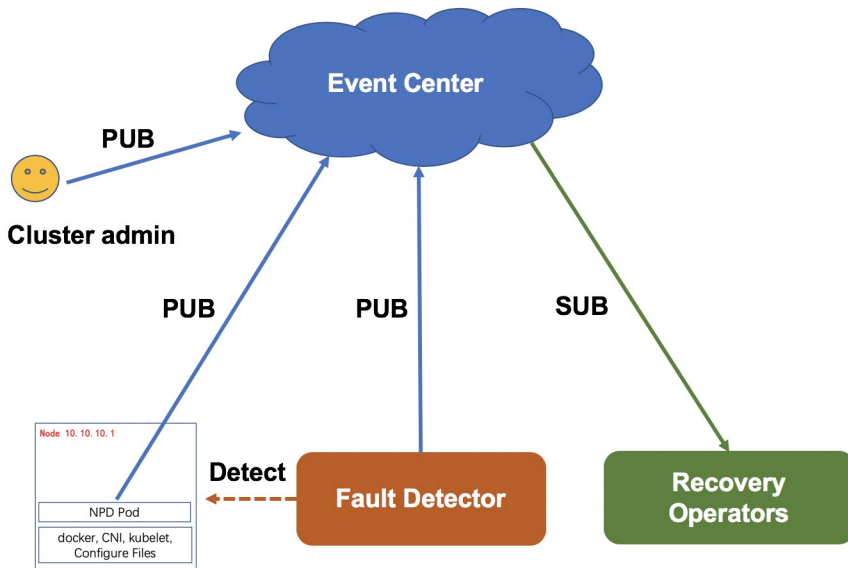
2. Machine-Operator 根据标签获取节点相关的所有子终态 Condition ConfigMap，并同步至 Machine status 的 conditions 中
3. Machine-Operator 根据全量 ReadinessGates 中记录的 Condition 列表，检查节点是否达到终态，未达到终态的节点不开启调度

## 节点故障自愈

我们都知道物理机硬件存在一定的故障概率，随着集群节点规模的增加，集群中会常态出现故障节点，如果不及时修复上线，这部分物理机的资源将会被闲置。

为解决这一问题，我们设计了一套故障发现、隔离、修复的闭环自愈系统。

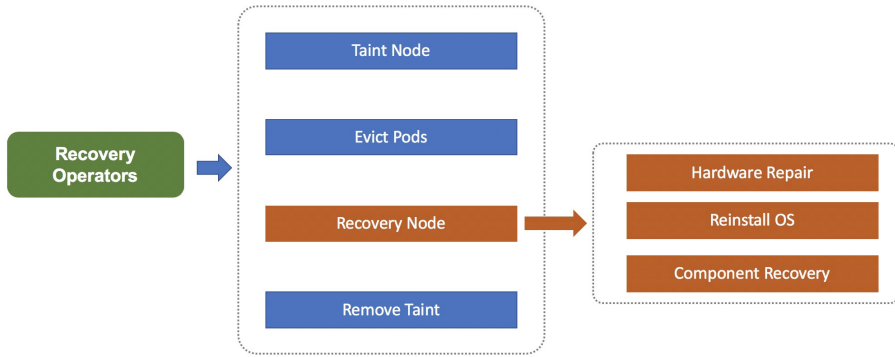
如下图所示，故障发现方面，采取 Agent 上报和监控系统主动探测相结合的方式，保证了故障发现的实时性和可靠性（Agent 上报实时性比较好，监控系统主动探测可以覆盖 Agent 异常未上报场景）。故障信息统一存储于事件中心，关注集群故障的组件或系统都可以订阅事件中心事件拿到这些故障信息。



节点故障自愈系统会根据故障类型创建不同的维修流程，例如：硬件维系流程、系统重装流程等。维修流程中优先会隔离故障节点（暂停节点调度），然后将节点上

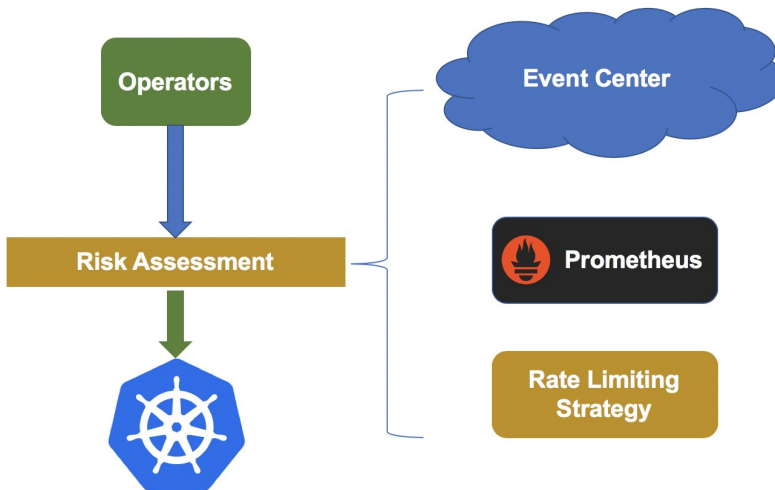


Pod 打上待迁移标签来通知 PAAS 或 MigrateController 进行 Pod 迁移，完成这些前置操作后，会尝试恢复节点（硬件维修、重装操作系统等），修复成功的节点会重新开启调度，长期未自动修复的节点由人工介入排查处理。



## 风险防范

在 Machine-Operator 提供的原子能力基础上，系统中设计实现了集群维度的灰度变更和回滚能力。此外，为了进一步降低变更风险，Operators 在发起真实变更时都会进行风险评估，架构示意图如下。



高风险变更操作(如:删除节点、重装系统)接入统一限流中心,限流中心维护了不同类型操作的限流策略,若触发限流,则熔断变更。

为了评估变更过程是否正常,我们会在变更前后,对各组件进行健康检查,组件的健康检查虽然能够发现大部分异常,但不能覆盖所有异常场景。所以,风险评估过程中,系统会从事件中心、监控系统中获取集群业务指标(如:Pod 创建成功率),如果出现异常指标,则自动熔断变更。

## 结束语

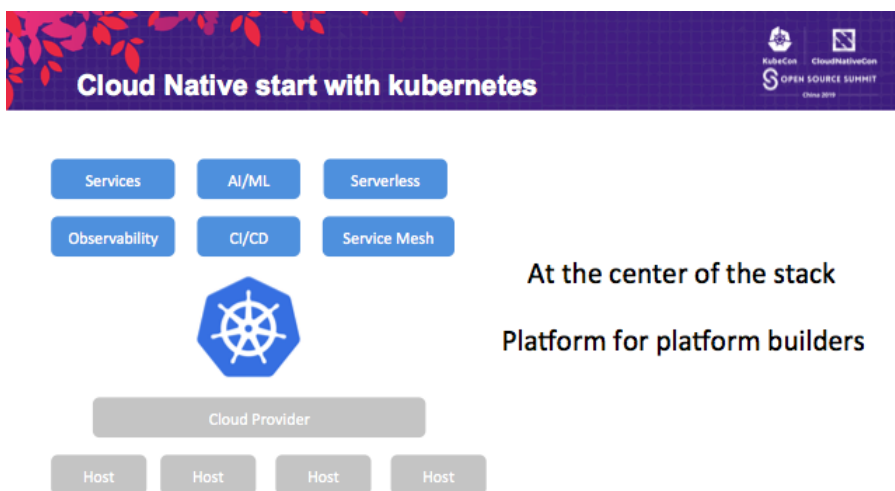
本文主要和大家分享了现阶段蚂蚁金服 Kubernetes 集群管理系统的核心设计,核心组件大量使用 Operator 面向终态设计模式。未来我们会尝试将集群规模变更切换为 Operator 面向终态设计模式,探索如何在面向终态的模式下,做到变更的可监控、可灰度和可回滚,实现变更的无人值守。

# 阿里新技术方案

## 云原生应用 Kubernetes 监控与弹性实践

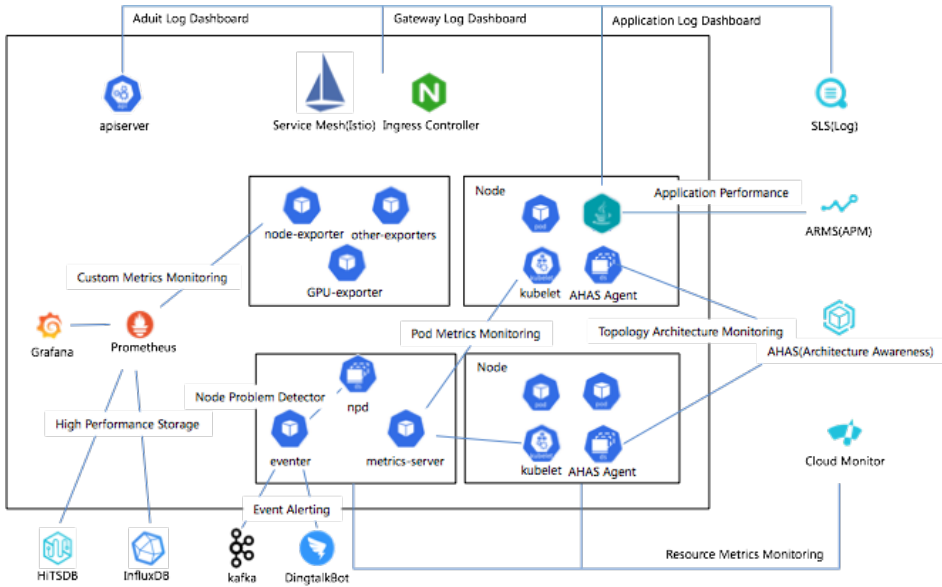
阿里云容器服务团队技术专家 莫源

### 前言



云原生应用的设计理念已经被越来越多的开发者接受与认可，而 Kubernetes 作为云原生的标准接口实现，已经成为了整个 stack 的中心，云服务的能力可以通过 Cloud Provider、CRD Controller、Operator 等等的方式从 Kubernetes 的标准接口向业务层透出。开发者可以基于 Kubernetes 来构建自己的云原生应用与平台，Kubernetes 成为了构建平台的平台。今天我们会向大家介绍一个云原生应用该如何在 Kubernetes 中无缝集成监控和弹性能力。

## 阿里云容器服务 Kubernetes 的监控总览



### 云服务集成

阿里云容器服务 Kubernetes 目前已经和四款监控云服务进行了打通，分别是 SLS (日志服务)、ARMS (应用性能监控)、AHAS (架构感知监控服务)、Cloud Monitor (云监控)。

SLS 主要负责日志的采集、分析。在阿里云容器服务 Kubernetes 中，SLS 可以采集三种不同类型的日志：

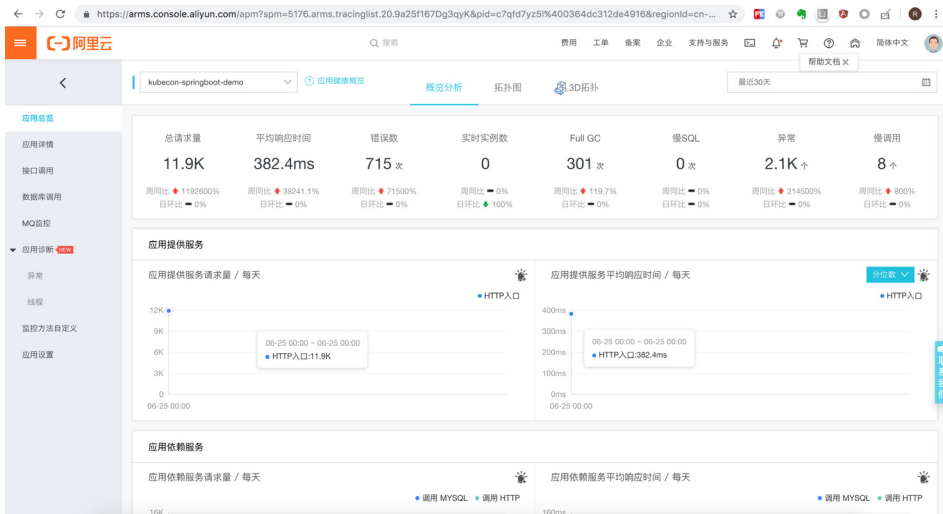
- APIServer 等核心组件的日志
- Service Mesh/Ingress 等接入层的日志
- 应用的标准日志

除了采集日志的标准链路外，SLS 还提供了上层的日志分析能力，默认提供了基于 APIServer 的审计分析能力、接入层的可观测性展现、应用层的日志分析。在阿里云容器服务 Kubernetes 中，日志组件已经默认安装，开发者只需要通过在集群

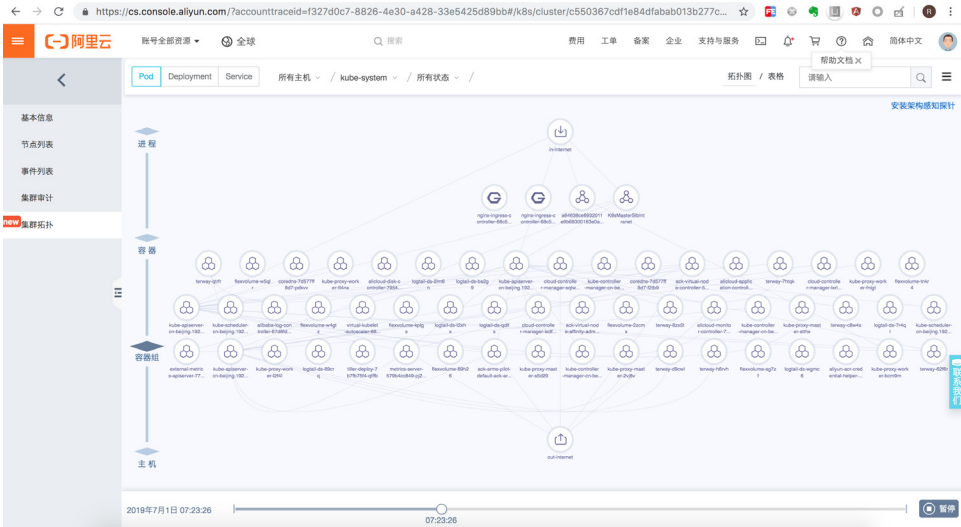
创建时勾选即可。



ARMS 主要负责采集、分析、展现应用的性能指标。目前主要支持 Java 与 PHP 两种语言的集成，可以采集虚拟机 ( JVM) 层的指标，例如 GC 的次数、应用的慢 SQL 、调用栈等等。对于后期性能调优可以起到非常重要的作用。



AHAS 是架构感知监控，通常在 Kubernetes 集群中负载的类型大部分为微服务，微服务的调用拓扑也会比较复杂，因此当集群的网络链路出现问题时，如何快速定位问题、发现问题、诊断问题则成为了最大的难题。AHAS 通过网络的流量和走向，将集群的拓扑进行展现，提供更高层次的问题诊断方式。



## 开源方案集成

开源方案的兼容和集成也是阿里云容器服务 Kubernetes 监控能力的一部分。主要包含如下两个部分：

### Kubernetes 内置监控组件的增强与集成

在 kubernetes 社区中，heapster/metrics-server 是内置的监控方案，而且例如 Dashboard、HPA 等核心组件会依赖于这些内置监控能力提供的 metrics。由于 Kubernetes 生态中组件的发布周期和 Kubernetes 的 release 不一定保证完整的同步，这就造成了部分监控能力的消费者在 Kubernetes 中存在监控问题。因此阿里云就这个问题做了 metrics-server 的增强，实现版本的兼容。此外针对节点的诊断能力，阿里云容器服务增强了 NPD 的覆盖场景，支持了 FD 文件句柄的监测、NTP 时间同步的校验、出入网能力的校验等等，并开源了 eventer，支持离线 Kubernetes

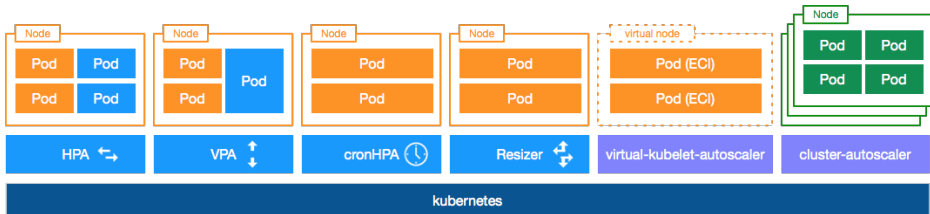
的事件数据到 SLS、kafka 以及钉钉，实现 ChatOps。

### Prometheus 生态的增强与集成

Prometheus 作为 Kubernetes 生态中三方监控的标准，阿里云容器服务也提供了集成的 Chart 供开发者一键集成。此外，我们还在如下三个层次作了增强：

- 存储、性能增强：支持了产品级的存储能力支持（TSDB、InfluxDB），提供更持久、更高效的监控存储与查询。
- 采集指标的增强：修复了部分由于 Prometheus 自身设计缺欠造成的监控不准的问题，提供了 GPU 单卡、多卡、共享分片的 exporter。
- 提供上层可观测性的增强：支持场景化的 CRD 监控指标集成，例如 argo、spark、tensorflow 等云原生的监控能力，支持多租户可观测性。

## 阿里云容器服务 Kubernetes 的弹性总览



阿里云容器服务 Kubernetes 主要包含如下两大类弹性组件：调度层弹性组件与资源层弹性组件。

### 调度层弹性组件

调度层弹性组件是指所有的弹性动作都是和 Pod 相关的，并不关心具体的资源情况。

- HPA

HPA 是 Pod 水平伸缩的组件，除了社区支持的 Resource Metrics 和 Custom Metrics，阿里云容器服务 Kubernetes 还提供了 external-metrics-adapter，支持

云服务的指标作为弹性伸缩的判断条件。目前已经支持例如：Ingress 的 QPS、RT，RMS 中应用的 GC 次数、慢 SQL 次数等等多个产品不同维度的监控指标。

- VPA

VPA 是 Pod 的纵向伸缩的组件，主要面向有状态服务的扩容和升级场景。

- cronHPA

cronHPA 是定时伸缩组件，主要面向的是周期性负载，通过资源画像可以预测有规律的负载周期，并通过周期性伸缩，实现资源成本的节约。

- Resizer

Resizer 是集群核心组件的伸缩控制器，可以根据集群的 CPU 核数、节点的个数，实现线性和梯度两种不同的伸缩，目前主要面对的场景是核心组件的伸缩，例如：CoreDNS。

## 资源层弹性组件

资源层弹性组件是指弹性的操作都是针对于 Pod 和具体资源关系的。

- Cluster-Autoscaler

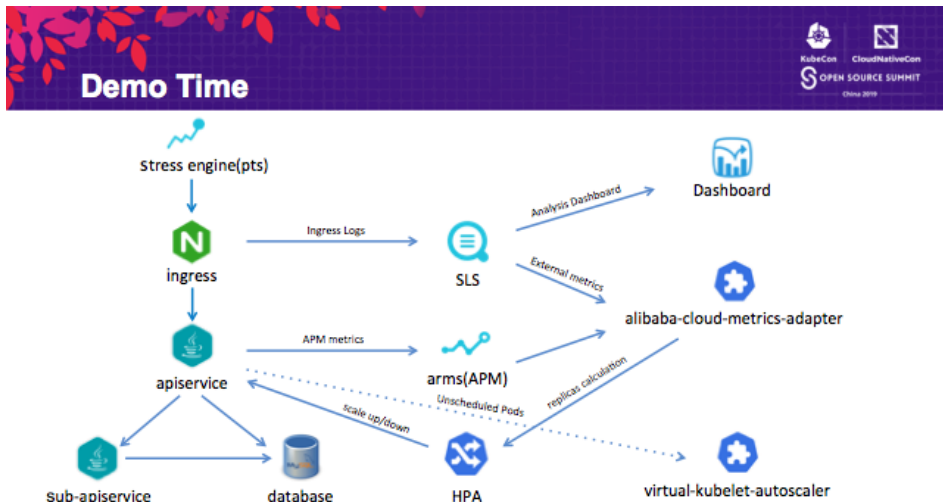
Cluster-Autoscaler 是目前比较成熟的节点伸缩组件，主要面向的场景是当 Pod 资源不足时，进行节点的伸缩，并将无法调度的 Pod 调度到新弹出的节点上。

- virtual-kubelet-autoscaler

virtual-kubelet-autoscaler 是阿里云容器服务 Kubernetes 开源的组件，和 Cluster-Autoscaler 的原理类似，当 Pod 由于资源问题无法调度时，此时弹出的不是节点，而是将 Pod 绑定到虚拟节点上，并通过 ECI 的方式将 Pod 进行启动。



## Demo Show Case



最后给大家进行一个简单的 Demo 演示：应用主体是 apiservice，apiservice 会通 sub-apiservice 调用 database，接入层通过 ingress 进行管理。我们通过 PTS 模拟上层产生的流量，并通过 SLS 采集接入层的日志，ARMS 采集应用的性能指标，并通过 alibaba-cloud-metrics-adapster 暴露 external metrics 触发 HPA 重新计算工作负载的副本，当伸缩的 Pod 占满集群资源时，触发 virtual-kubelet-autoscaler 生成 ECI 承载超过集群容量规划的负载。

## 总结

在阿里云容器服务 Kubernetes 上使用监控和弹性的能力是非常简单的，开发者只需一键安装相应的组件 Chart 即可完成接入，通过多维度的监控、弹性能力，可以让云原生应用在最低的成本下获得更高的稳定性和鲁棒性。

## 了解 Kubernetes Master 的可扩展性和性能

阿里云容器平台高级软件工程师 陈星宇 (宇慕)

阿里云容器平台高级技术专家 曾凡松 (逐灵)

### Background



今年的 Kubecon China 在上海黄浦江边的世博中心举办，能在梅雨时节的上海遇到三天的蓝天、白云与阳光，可以说是运气相当不错。大会的参与人员多各大云厂商和解决方案提供方，在应用 kubernetes 作为公司的 IaaS 基础设施时，或多或少都遇到了规模化相关的问题。阿里巴巴分享的在大规模应用 kubernetes 上的经验这个 topic 吸引了很多的听众，也得到了一些客户的反馈，这里将本次分享的内容做一个总结。

从阿里巴巴最早期的 AI 系统 (2013) 开始，集群管理系统经历了多轮的架构演进，到 2018 年全面的应用 kubernetes，这中间的故事挺多。鉴于篇幅的问题，这里不讨论为什么 kubernetes 能够在社区和公司内部全面的胜出，我们将焦点关注到应用 kubernetes 中会遇到什么样的问题，以及我们做了哪些关键的优化。

## Kubernetes in Alibaba

- Production environment
  - 10, 000s of applications
  - 1, 000, 000s of containers
  - 10s of clusters
  - 100, 000s of nodes
  - 10, 000 nodes / largest cluster

在阿里巴巴的生产环境中，容器化的应用超过了 10k 个，全网的容器在百万的级别，运行在十几万台宿主机上。支撑阿里巴巴核心电商业务的集群有十几个，最大的集群有几万的节点。在落地 kubernetes 的过程中，在规模上面临了很大的挑战，如何将 kubernetes 应用到超大规模的生产级别。

罗马不是一天就建成的，为了了解 Kubernetes 的性能瓶颈，我们结合阿里的生产集群现状，估算了在 10k 个节点的集群中，预计会达到的规模：

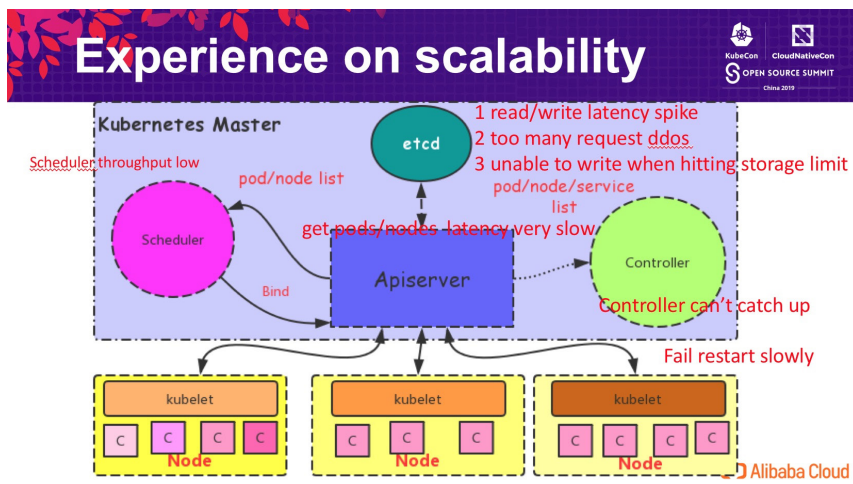
- 20w pods
- 100w objects

## Experience on scalability

A 10k nodes cluster

# Pods	~200k
# Objects	~1000k
# Latency	~10s

我们基于 Kubemark 搭建了大规模集群模拟的平台，通过一个容器启动多个 (50 个) kubemark 进程的方式，使用了 200 个 4c 的容器模拟了 10k 节点的 kubelet。在模拟集群中运行常见的负载时，我们发现一些基本的操作比如 pod 调度延迟非常高，达到了惊人的 10s 这一级别，并且集群处在非常不稳定的状态。

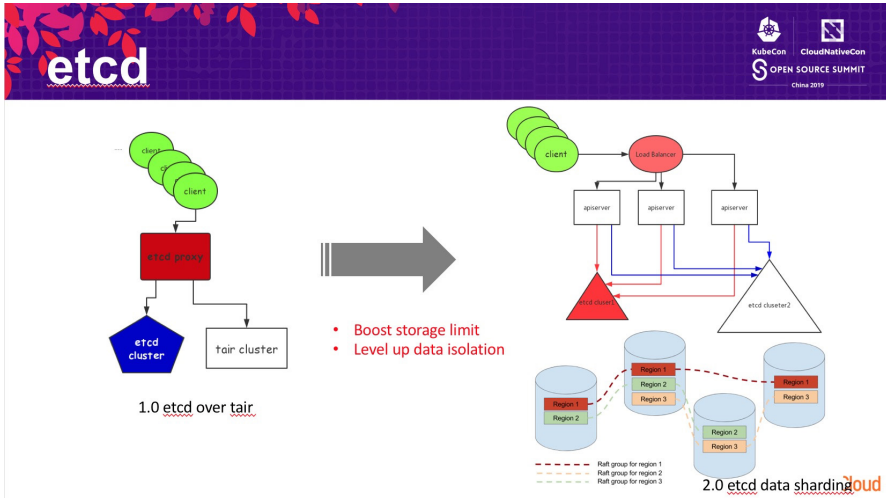


当 kubernetes 集群规模达到 10k 节点时，系统的各个组件均出现相应的性能问题，比如：

1. etcd 中出现了大量的读写延迟，并且产生了拒绝服务的情形，同时因其空间的限制也无法承载 kubernetes 存储大量的对象
2. API Server 查询 pods/nodes 延迟非常的高，并发查询请求可能地址后端 etcd oom
3. Controller 不能及时从 API Server 感知到在最新的变化，处理的延时较高；当发生异常重启时，服务的恢复时间需要几分钟
4. Scheduler 延迟高、吞吐低，无法适应阿里业务日常运维的需求，更无法支持大促态的极端场景

## etcd improvements

为了解决这些问题，阿里云容器平台在各方面都做了很大的努力，改进 kubernetes 在大规模场景下的性能。首先是 etcd 层面，作为 kubernetes 存储对象的数据库，其对 kubernetes 集群的性能影响至关重要。



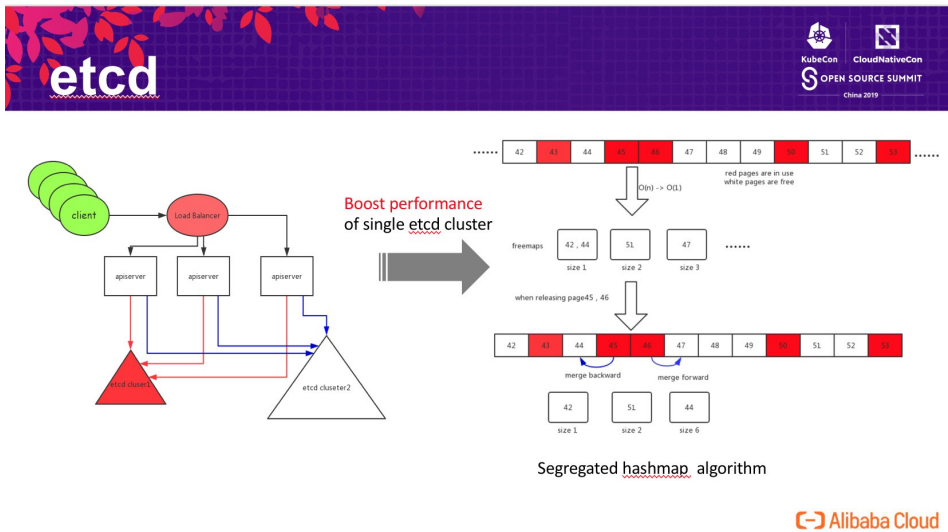
第一版本的改进，我们通过将 etcd 的数据转存到 tair 集群中，提高了 etcd 存储的数据总量。但这个方式有一个显著的弊端是额外增加的 tair 集群，增加的运维复杂性对集群中的数据安全性带来了很大的挑战，同时其数据一致性模型也并非基于 raft 复制组，牺牲了数据的安全性。

第二版本的改进，我们通过将 API Server 中不同类型的对象存储到不同的 etcd 集群中。从 etcd 内部看，也就对应了不同的数据目录，通过将不同目录的数据路由到不同的后端 etcd 中，从而降低了单个 etcd 集群中存储的数据总量，提高了扩展性。

第三版本的改进，我们深入研究了 etcd 内部的实现原理，并发现了影响 etcd 扩展性的一个关键问题在底层 bbolt db 的 page 页面分配算法上：随着 etcd 中存储的数据量的增长，bbolt db 中线性查找“连续长度为 n 的 page 存储页面”的性能显著下降。为了解决该问题，我们设计了基于 segregated hashmap 的空闲页面管理算法，hashmap 以连续 page 大小为 key，连续页面起始 page id 为 value。通过查这

个 segregated hashmap 实现  $O(1)$  的空闲 page 查找，极大地提高了性能。在释放块时，新算法尝试和地址相邻的 page 合并，并更新 segregated hashmap。更详细的算法分析可以见已发表在 cncf 博客的博文：

<https://www.cncf.io/blog/2019/05/09/performance-optimization-of-etcd-in-web-scale-data-scenario/>



通过这个算法改进，我们可以将 etcd 的存储空间从推荐的 2GB 扩展到 100GB，极大的提高了 etcd 存储数据的规模，并且读写无显著延迟增长。除此之外，我们也和谷歌工程师协作开发了 etcd raft learner (类 zookeeper observer) / fully concurrent read 等特性，在数据的安全性和读写性能上进行增强。这些改进已贡献开源，将在社区 etcd 3.4 版本中发布。

## API Server improvements

### Efficient node heartbeats

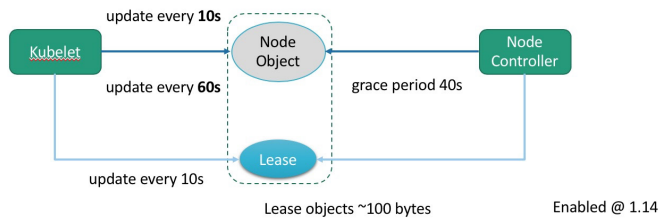
在 kubernetes 集群中，影响其扩展到更大规模的一个核心问题是如何有效的处理节点的心跳。在一个典型的生产环境中 (non-trivial)，kubelet 每 10s 汇报一次心

跳，每次心跳请求的内容达到 15kb (包含节点上数十计的镜像，和若干的卷信息)，这会带来两大问题：

1. 心跳请求触发 etcd 中 node 对象的更新，在 10k nodes 的集群中，这些更新将产生近 1GB/min 的 transaction logs (etcd 会记录变更历史)
2. API Server 很高的 CPU 消耗，node 节点非常庞大，序列化 / 反序列化开销很大，处理心跳请求的 CPU 开销超过 API Server CPU 时间占用的 80%



● Add a new `Lease` build-in API



Alibaba Cloud

为了解决这个问题，kubernetes 引入了一个新的 build-in Lease API，将与心跳密切相关的信息从 node 对象中剥离出来，也就是上图中的 Lease。原本 kubelet 每 10s 更新一次 node 对象升级为：

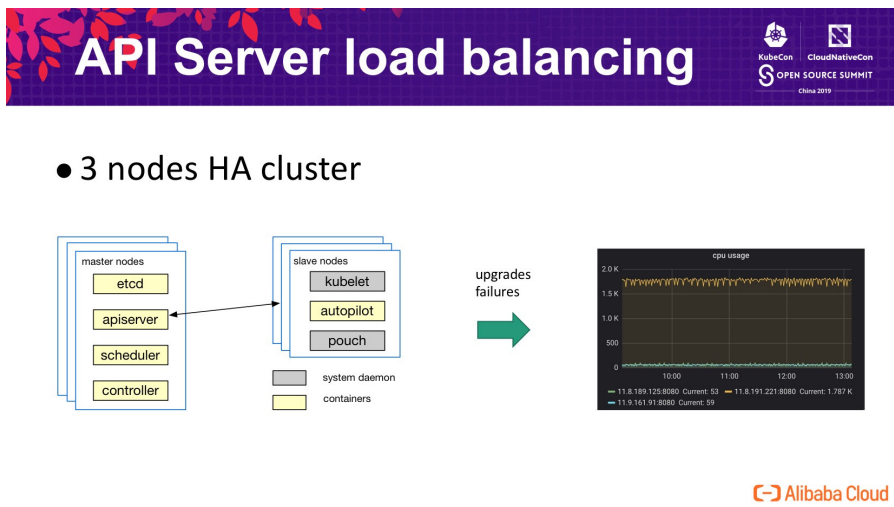
1. 每 10s 更新一次 Lease 对象，表明该节点的存活状态，Node Controller 根据该 Lease 对象的状态来判断节点是否存活。
2. 处于兼容性的考虑，降低为每 60s 更新一次 node 对象，使得 Eviction Manager 等可以继续按照原有的逻辑工作。

因为 Lease 对象非常小，因此其更新的代价远小于更新 node 对象。kubernetes 通过这个机制，显著的降低了 API Server 的 CPU 开销，同时也大幅减小了

etcd 中大量的 transaction logs，成功将其规模从 1000 扩展到了几千个节点的规模，该功能在社区 kubernetes-1.14 中已经默认启用。

## API Server load balancing

在生产集群中，出于性能和可用性的考虑，通常会部署多个节点组成高可用 kubernetes 集群。但在高可用集群实际的运行中，可能会出现多个 API Server 之间的负载不均衡，尤其是在集群升级或部分节点发生故障重启的时候。这给集群的稳定性带来了很大的压力，原本计划通过高可用的方式分摊 API Server 面临的压力，但在极端情况下所有压力又回到了一个节点，导致系统响应时间变长，甚至击垮该节点继而导致雪崩。下图为压测集群中模拟的一个 case，在三个节点的集群，API Server 升级后所有的压力均打到了其中一个 API Server 上，其 CPU 开销远高于其他两个节点。



解决负载均衡问题，一个自然的思路就是增加 load balancer。前文的描述中提到，集群中主要的负载是处理节点的心跳，那我们就在 API Server 与 kubelet 中间增加 lb，有两个典型的思路：

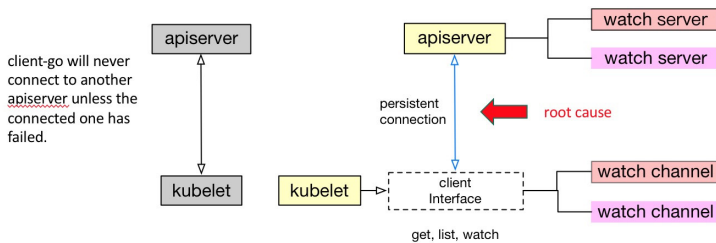


1. API Server 测增加 lb，所有的 kubelets 连接 lb，典型的云厂商交付的 kubernetes 集群，就是这一模式
2. kubelet 测增加 lb，由 lb 来选择 API Server

## API Server load balancing

KubeCon CloudNativeCon  
OPEN SOURCE SUMMIT  
China 2019

### ● try hard to reuse tls connection

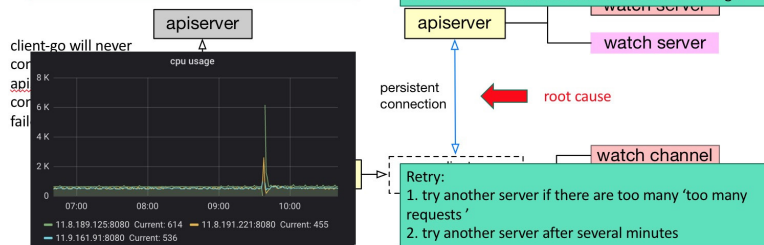


Alibaba Cloud

## API Server load balancing

KubeCon CloudNativeCon  
OPEN SOURCE SUMMIT  
China 2019

### ● Upgrade with {maxSurge=3} needed



Alibaba Cloud

通过压测环境验证发现，增加 lb 并不能很好的解决上面提到的问题，我们必须  
要深入理解 kubernetes 内部的通信机制。深入到 kubernetes 中研究发现，为了解  
决 tls 连接认证的开销，kubernetes 客户端做了很多的努力确保 “尽量复用同样的

tls 连接”，大多数情况下客户端 watcher 均工作在下层的同一个 tls 连接上，仅当这个连接发生异常时，才可能会触发重连继而发生 API Server 的切换。其结果就是我们看到的，当 kubelet 连接到其中一个 API Server 后，基本上是不会发生负载切换。为了解决这个问题，我们进行了三个方面的优化：

1. API Server：认为客户端是不可信的，需要保护自己不被过载的请求击溃。当自身负载超过一个阈值时，发送 409 - too many requests 提醒客户端退避；当自身负载超过一个更高的阈值时，通过关闭客户端连接拒绝请求
2. Client：在一个时间段内频繁的收到 409 时，尝试重建连接切换 API Server；定期的重建连接切换 API Server 完成洗牌
3. 运维层面，我们通过设置 maxSurge=3 的方式升级 API Server，避免升级过程带来的性能抖动

如上图左下角监控图所示，增强后的版本可以做到 API Server 负载基本均衡，同时在显示重启两个节点（图中抖动）时，能够快速自动恢复到均衡状态。

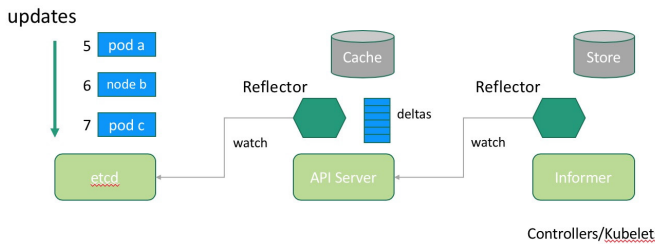
## List-Watch & Cacher

List-Watch 是 Kubernetes 中 Server 与 Client 通信最核心一个机制，etcd 中所有对象及其更新的信息，API Server 内部通过 Reflector 去 watch etcd 的数据变化并存储到内存中，controller/kubelets 中的客户端也通过类似的机制去订阅数据的变化。

# List-Watch & Cacher



- Key communication mechanisms between client and server

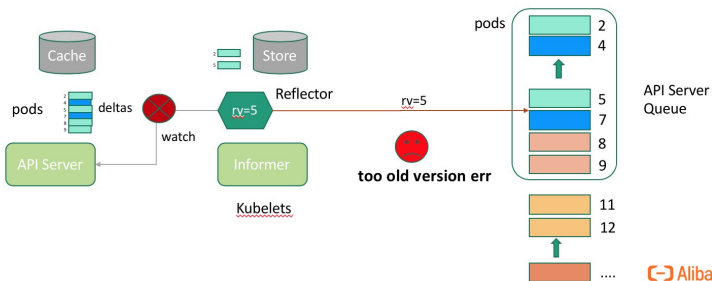


在 List-Watch 机制中面临的一个核心问题是，当 Client 与 Server 之间的通信断开时，如何确保重连期间的数据不丢，这在 kubernetes 中通过了一个全局递增的版本号 resourceVersion 来实现。如下图所示 Reflector 中保存这当前已经同步到的数据版本，重连时 Reflector 告知 Server 自己当前的版本 (5)，Server 根据内存中记录的最近变更历史计算客户端需要的数据起始位置 (7)。这一切看起来十分简单可靠，但是 ...

# List-Watch & Cacher



- What happens if the connection is broken ?

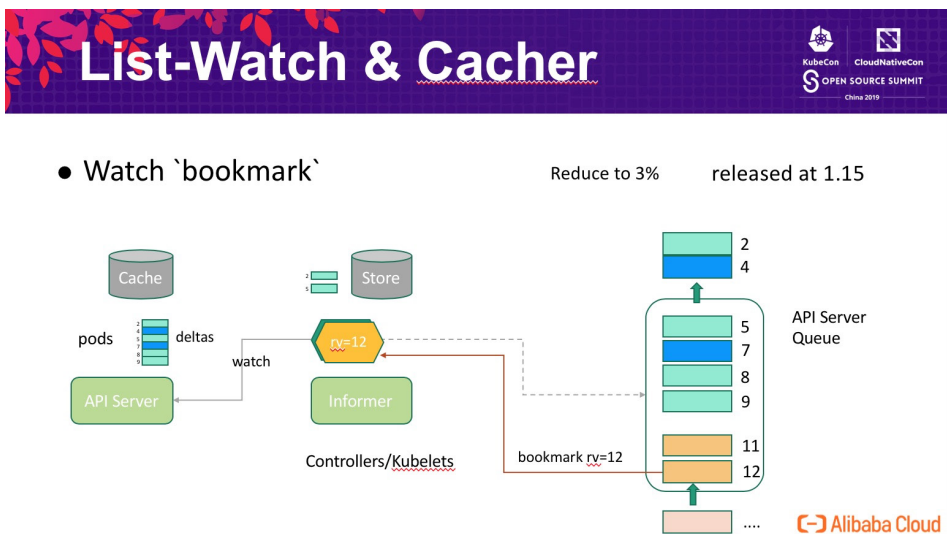


在 API Server 内部，每个类型的对象会存储在一个叫做 storage 的对象中，比如会有：

1. Pod Storage
2. Node Storage
3. Configmap Storage
4. ...

每个类型的 storage 会有一个有限的队列，存储对象最近的变更，用于支持 watcher 一定的滞后（重试等场景）。一般来说，所有类型的类型共享一个递增版本号空间（1, 2, 3, ..., n），也就是如上图所示，pod 对象的版本号仅保证递增不保证连续。Client 使用 List-Watch 机制同步数据时，可能仅关注 pods 中的一部分，最典型的 kubelet 仅关注和自己节点相关的 pods，如上图所示，某个 kubelet 仅关注绿色的 pods（2, 5）。

因为 storage 队列是有限的（FIFO），当 pods 的更新时队列，旧的变更就会从队列中淘汰。如上图所示，当队列中的更新与某个 Client 无关时，Client 进度仍然保持在  $rv=5$ ，如果 Client 在 5 被淘汰后重连，这时候 API Server 无法判断 5 与当前队列最小值（7）之间是否存在客户端需要感知的变更，因此返回 Client too old version err 触发 Client 重新 list 所有的数据。为了解决这个问题，kubernetes 引入 Watch bookmark 机制：



bookmark 的核心思想概括起来就是在 Client 与 Server 之间保持一个“心跳”，即使队列中无 Client 需要感知的更新，Reflector 内部的版本号也需要及时的更新。如上图所示，Server 会在合适的适合推送当前最新的 rv=12 版本号给 Client，使得 Client 版本号跟上 Server 的进展。bookmark 可以将 API Server 重启时需要重新同步的事件降低为原来的 3% (性能提高了几十倍)，该功能有阿里云容器平台开发，已经发布到社区 kubernetes-1.15 版本中。

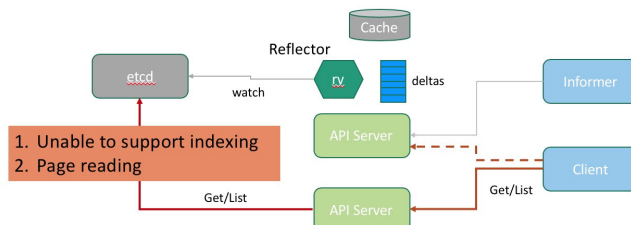
## Cacher & Indexing

除 List-Watch 之外，另外一种客户端的访问模式是直接查询 API Server，如下图所示。为了保证客户端在多个 API Server 节点间读到一致的数据，API Server 会通过获取 etcd 中的数据来支持 Client 的查询请求。从性能角度看，这带来了几个问题：

1. 无法支持索引，查询节点的 pod 需要先获取集群中所有的 pod，这个开销是巨大的
2. 因为 etcd 的 request-response 模型，单次请求查询过大的数据会消耗大量的内存，通常情况下 API Server 与 etcd 之间的查询会限制请求的数据量，并通过分页的方式来完成大量的数据查询，分页带来的多次的 round trip 显著降低了性能
3. 为了确保一致性，API Server 查询 etcd 均采用了 Quorum read，这个查询开销是集群级别，无法扩展的

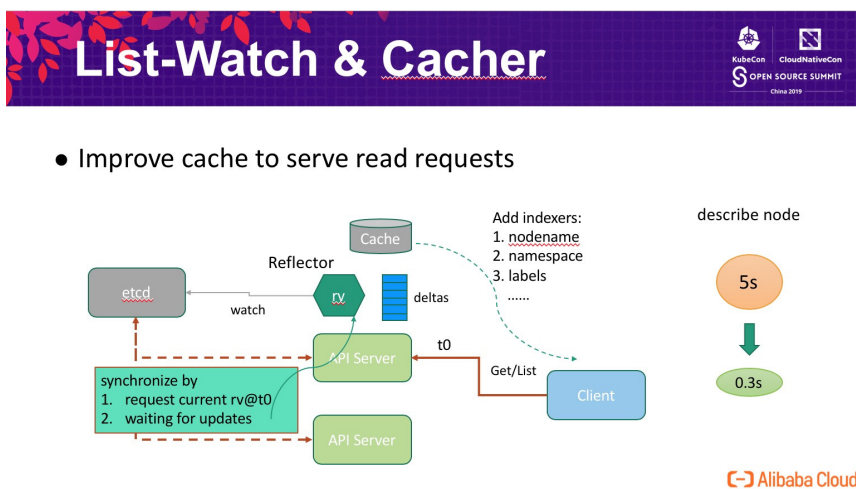


- Improve cache to serve read requests



为了解决这个问题，我们设计了 API Server 与 etcd 的数据协同机制，确保 Client 能够通过 API Server 的 cache 获取到一致的数据，其原理如下图所示，整体工作流程如下：

1. t0 时刻 Client 查询 API Server
2. API Server 请求 etcd 获取当前的数据版本 rv@t0
3. API Server 请求进度的更新，并等待 Reflector 数据版本达到 rv@t0
4. 通过 cache 响应用户的请求



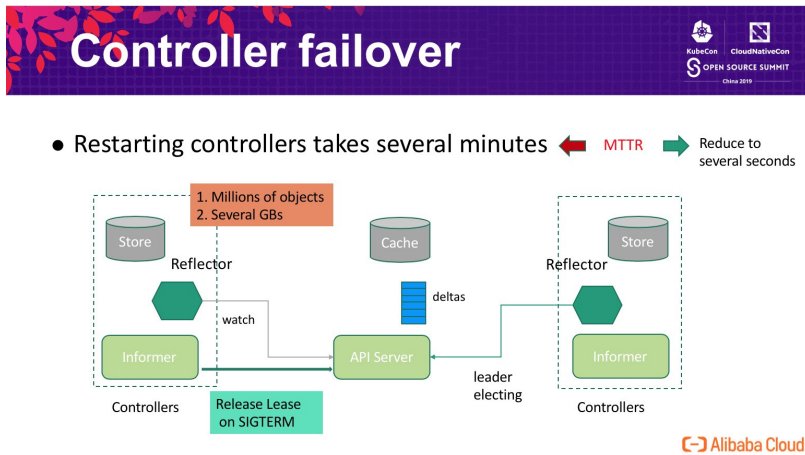
这个方式并未打破 Client 的一致性模型（感兴趣的可以自己论证一下），同时通过 cache 响应用户请求时我们可以灵活的增强查询能力，比如支持 namespace/nodename/labels 索引。该增强大幅提高了 API Server 的读请求处理能力，在万台规模集群中典型的 describe node 的时间从原来的 5s 降低到 0.3s（触发了 node name 索引），其他如 get nodes 等查询操作的效率也获得了成倍的增长。

## Controller failover

在 10k node 的生产集群中，Controller 中存储着近百万的对象，从 API Server 获取这些对象并反序列化的开销是无法忽略的，重启 Controller 恢复时可能

需要花费几分钟才能完成这项工作，这对于阿里巴巴规模的企业来说是不可接受的。为了减小组件升级对系统可用性的影响，我们需要尽量的减小 controller 单次升级对系统的中断时间，这里通过如下图所示的方案来解决这个问题：

1. 预启动备 controller informer，提前加载 controller 需要的数据
2. 主 controller 升级时，会主动释放 Leader Lease，触发备立即接管工作



通过这个方案，我们将 controller 中断时间降低到秒级别（升级时 < 2s），即使在异常宕机时，备仅需等待 leader lease 的过期（默认 15s），无需要花费几分钟重新同步数据。通过这个增强，显著的降低了 controller MTTR，同时降低了 controller 恢复时对 API Server 的性能冲击。

## Customized scheduler

由于历史原因，阿里巴巴的调度器采用了自研的架构，因时间的关系本次分享并未展开调度器部分的增强。这里仅分享两个基本的思路，如下图所示：

1. Equivalence classes：典型的用户扩容请求为一次扩容多个容器，因此我们通过将 pending 队列中的请求划分等价类的方式，实现批处理，显著的降低 Predicates/Priorities 的次数

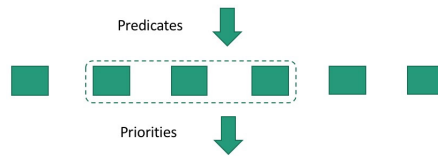
2. Relaxed randomization: 对于单次的调度请求，当集群中的候选节点非常多时，我们并不需要评估集群中全部节点，在挑选到足够的节点后即可进入调度的后续处理（通过牺牲求解的精确性来提高调度性能）



- Equivalence classes



- Relaxed randomization



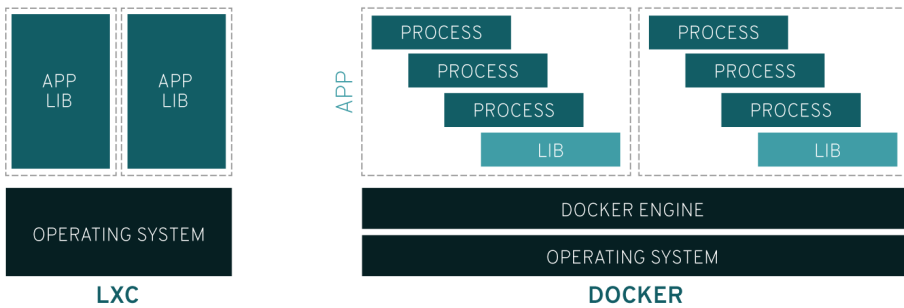


# 云原生时代加速镜像分发的三种方法

阿里云容器平台技术专家 江勇 (益方)

## 背景介绍

### Traditional Linux containers vs. Docker



在业务很小的时候，可以单台机器部署单个容器；但当业务量大了，并且分为多个应用时，就需要将多个应用部署在同一台机器上；此时多个应用的依赖是一个比较难解决的问题。同时多个应用同时运行，相互之间也有所干扰。

一开始，可以使用虚拟机的方式来实现，这样每个应用一个虚拟机，基本不存在依赖冲突，但是虚拟机还是会有启动慢，空间占用大等一系列问题。

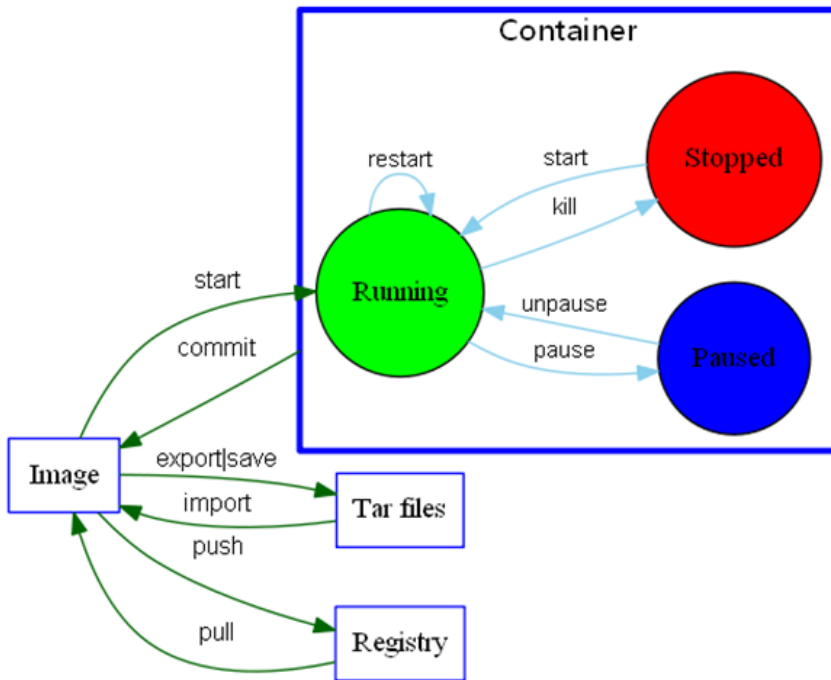
后面随着技术的发展，有了 lxc 技术，它能让应用跑在一个单独的命名空间上，实现了应用的运行态隔离。

这里，可以看到虚拟机解决了静态隔离的问题，而 lxc 解决了运行态的问题，而 docker 则同时解决了运行态和静态隔离两个问题，因此得到青睐。

## Docker 核心

Docker 的三大核心：镜像、容器、仓库。容器在 Docker 之前已经有了不同的

实现，比如 lxc。镜像和仓库是 Docker 的很大创新。镜像是一个实体，仓库是它的集中存储中心，而容器则是镜像的运行时，Docker 通过镜像，可以很神奇地实现比如，开发的同学在自己喜欢的平台上面做开发测试，比如：ubuntu；而真正部署的节点，可能是 redhat/centos 的服务器。

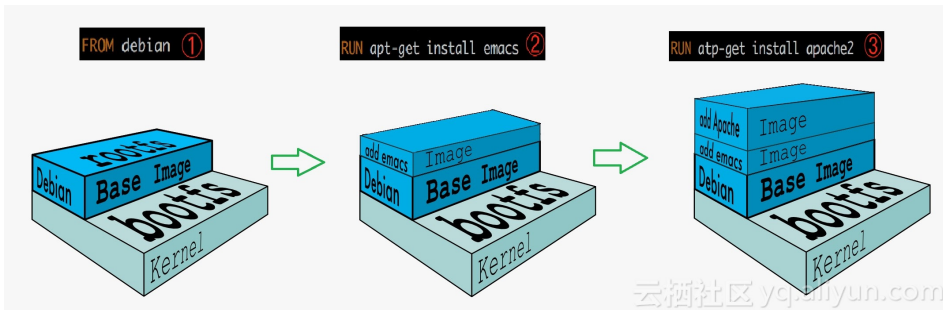


Docker 是以镜像为基础，所以容器运行是必要有镜像，而容器又可以通过 docker commit 生成一个镜像，但是第一个镜像从哪里来呢？可以通过 docker import/load 的方式导入。并且可以通过 docker push/pull 的方向将镜像上传到镜像中心，或从镜像中心下载镜像。

## 镜像制作

```

1. From Debian
2. RUN apt-get install emacs
3. RUN apt-get install apache2
4. CMD [ "/bin/bash" ]
    
```

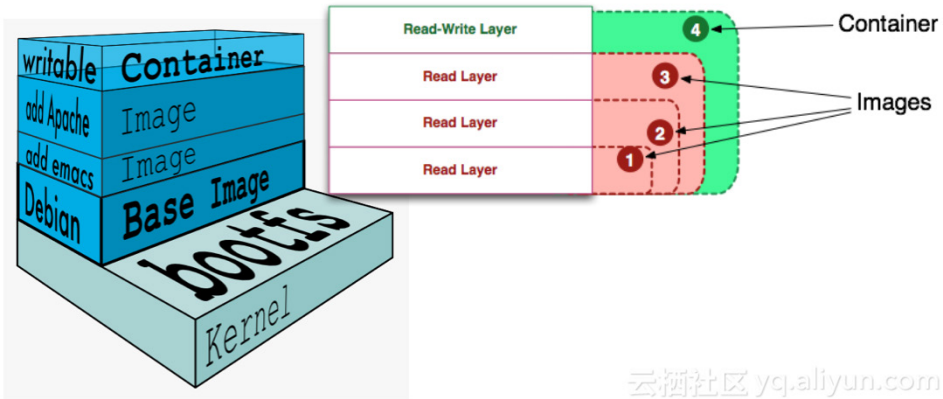


Docker 提供了一套 docker build 机制，方便来制作镜像，想像一下，没有 docker build 机制，制作镜像的步骤是如下：

- 从一个基础镜像启动一个容器
- 在容器中执行对应的命令，比如安装一个软件包，添加一个文件等；
- 使用 docker commit 命令，把刚才的容器 commit 成一个镜像；
- 如果还有其它步骤要做，就要从新镜像中启动一个容器，然后再 commit，直到完成所有操作。

有了 docker build 机制，只需要一个 dockerfile，然后 docker 就会把上述步骤自动化了，最后生成一个目标镜像。

## 容器



详细看一下是如何使用镜像的，镜像是一层一层的，设计上，镜像所有层是只读的，linux 通过 aufs/overlayfs 的方式将一个镜像 mount 到主机上后，从上往下看，最上面层是容器的可读写层，容器运行时的所有写操作都写到这一层上面，而下面的层都是镜像的只读层。

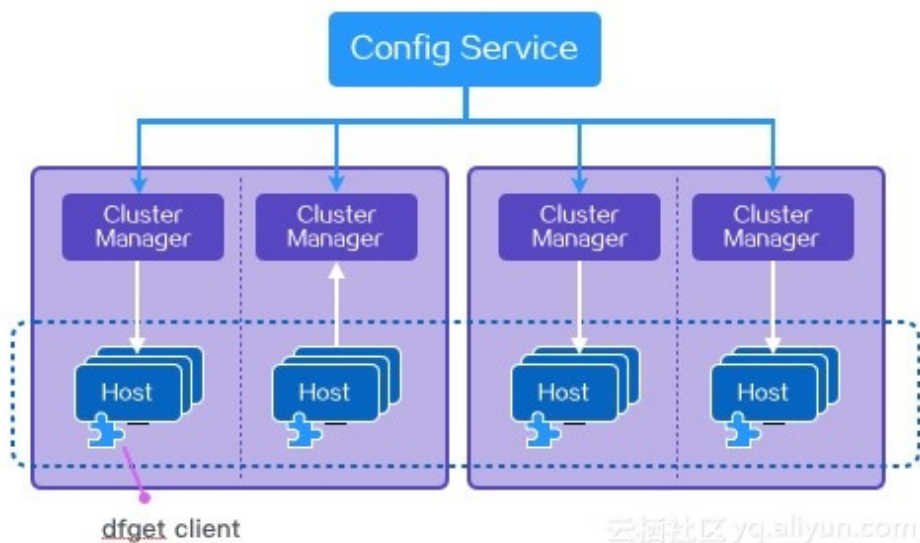
## 大规模集群镜像下载慢

在 docker 的 registry 或其它网站上看到，像 busybox，nginx，mysql 等镜像都不大，也就是几十 M 到几百 M 的样子。但是阿里巴巴的镜像普遍有 3~4G，镜像这么大原因有许多，比如集团使用了富容器的模式，这是一种类似虚拟机的方式，对开发较为友好，开发可以在以前的经验上做开发，不需要太多的学习成本，这就导致集团的基础镜像有 2~3G；第二集团的业务都采用分布式的方式开发，依赖许多中间件，而中间件因版本的原因，没有完全统一起来，每个中间件都有几百 M；再加上本身 java 业务包也不小，所以普遍 3~4G 是很正常的。

另外一个情况是，集团有上十万的物理机，上面运行着上百万的容器，每天有着上千次发布，可以想像一下，就单单把镜像从镜像中心下载到物理机上面，每天消耗的带宽是巨大的。

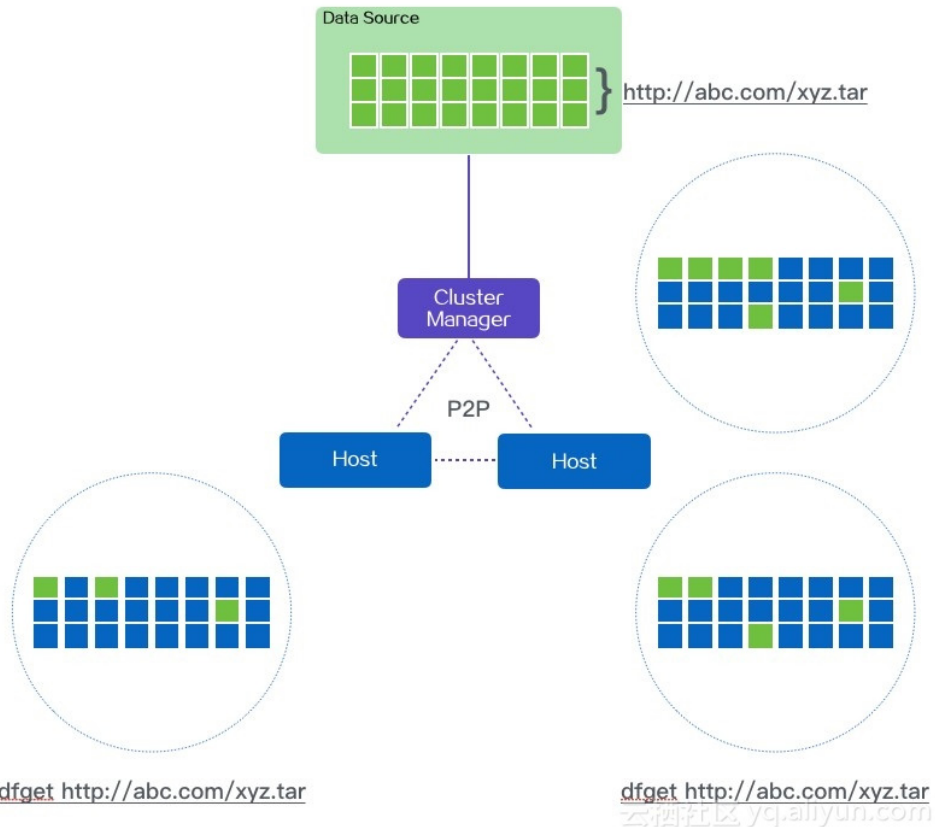
这就导致一个问题，镜像中心很有可能被打爆，这种情况在集团出现多次，在打爆的情况下，当前的镜像拉取被阻塞，同时又有更多的镜像拉取请求上来，导致镜像中心严重超负荷运行，并且导致问题越来越严重，可能会导致整个系统奔溃。因此，必须解决镜像下载导致的问题。

## 蜻蜓架构



首先，集团开发了蜻蜓这套 P2P 下载系统，上面是蜻蜓的架构图，蜻蜓可以分为三层，最上层是一个配置中心，中间是超级节点，下面就是运行容器的物理节点。

配置中心管理整个蜻蜓集群，当物理节点上有镜像要下载时，它通过 dfget 命令向超级节点发出请求，而向哪些超级节点发出请求是由配置中心配置的。以下图为例：



当下载一个文件时，物理节点发送了一个 `dfget` 请求到超级节点，超级节点会检查自身是否有这部分数据，如果有，则直接把数据传送给物理节点；否则会向镜像中心发出下载请求，这个下载请求是分块的，这样，只有下载失败的块需要重新下载，而下载成功的块，则不需要。

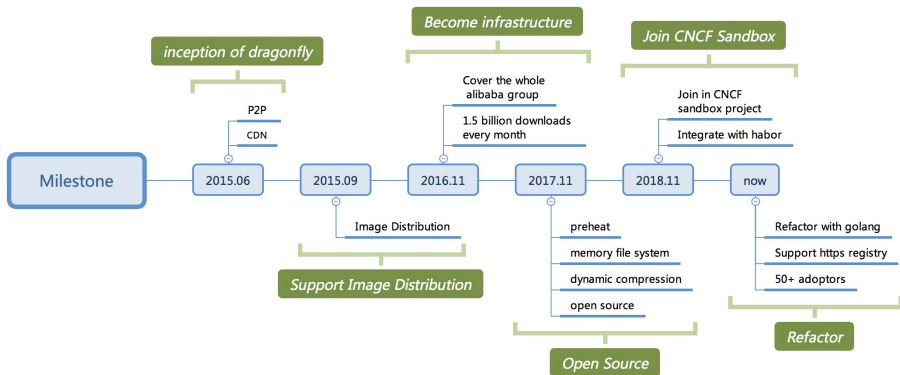
更为重要的是，如果是两个或更多物理节点请求同一份数据，那么蜻蜓收到请求后，只会往镜像中心发一个请求，可极大地降低了镜像中心的压力；

关于 P2P，也就是说如果一个节点有了另一个节点的数据，为了减少超级节点的压力，两个节点可以完成数据的互传。这样更高效地利用了网络带宽。

## 蜻蜓效果

从上面的蜻蜓效果图可以看到，随着镜像拉取的并发量越来越大，传统方式所消耗的时间会越来越多，后面的线条越来越陡；但是蜻蜓模式下，虽然耗时有增加，但只是轻微增加。效果还是非常明显的。

## 蜻蜓里程碑



应该说，集团内是先有蜻蜓，后面才有 docker 的，原因是集团在全面 docker 之前，使用了 t4 等技术，不管哪种技术，都需要在物理机上面下载应用的包，然后把应用跑起来。

所以，从里程碑上看，2015 年 6 月就已经有了蜻蜓 p2p 了，随着集团全面 docker 化，在 2015 年 9 月，蜻蜓提供了对 docker 镜像的支持，后面的几个时间点上，可以看到蜻蜓经受了数次双 11 的考验，并且经过软件迭代，在 2017 年 11 月的时候实现了开源。

现在蜻蜓还在不断演化，比如说全部使用 golang 重写，解决 bug，提供更多能力集，相信蜻蜓的明天更美好。

## 蜻蜓也不能完美解决镜像下载的问题

蜻蜓在集团取得了极好的效果，但是并不能解决所有问题，比如我们就遇到了以下比较突出的问题：

- 镜像下载导致业务抖动

在实践中，上线的业务运行经常有抖动的现象，分析其中原因，有部分业务抖动时，都在做镜像下载。

分析原因发现，镜像层采用的是 gzip 压缩算法，这一古老的算法，是单线程运行，并且独占一个 cpu，更要命的是解压速度也很慢，如果几个镜像层同时展开的话，就要占用几个 CPU，这抢占了业务的 CPU 资源，导致了业务的抖动。

- 应用扩容要更好的时效性

平时应用发布、升级时，只需要选择在业务低峰，并且通过一定的算法，比如只让 10% 左右的容器做发布、升级操作，其它 90% 的容器还给用户提供服务，这种不影响业务、不影响用户体验的操作，对于时效性要求不高，只要在业务高峰来临前完成操作即可，所以耗时一两个小时也无所谓。

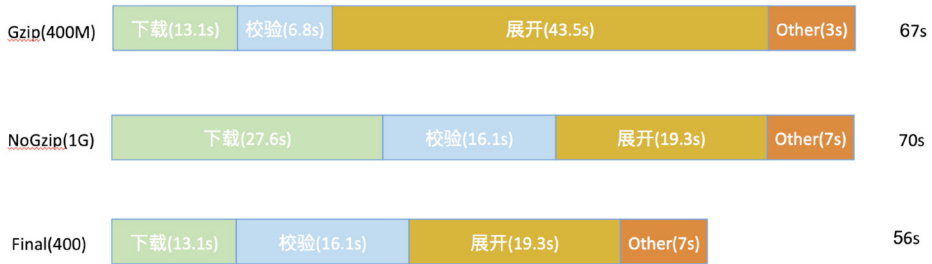
假设遇到大促场景，原计划 10 个容器可以服务 100 万用户，但是，突然来了 300 万用户，这时所有用户的体验将会下降，这时就需要通过扩容手段来增加服务器数量，这时对容器扩出来有着极高的时效要求。

- 如何而对云环境

现在所有公司都在上云，集团也在上云阶段，但云上服务器的一些特性与物理机还是有些差别，对镜像来讲感受最深的就是磁盘 IO 了，原来物理机的 SSD 磁盘，IO 可以达到 1G 以上，而使用 ECS 后，标准速度是 140M，速度只有原来的十分之一。



## Gzip 优化

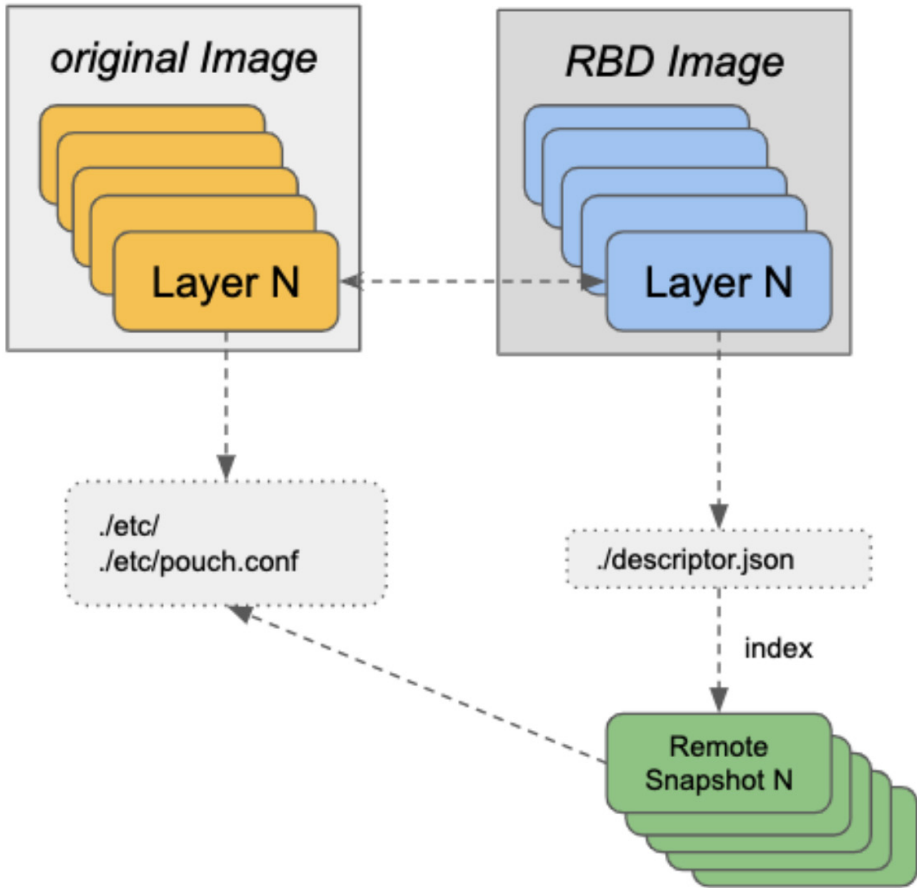


对于 gzip 的问题，通过实测及数据分析，如上图，1 与 2 下载的是同一个镜像层，只是 1 下载的是一个 gzip 格式的，而 2 下载的是一个没有压缩的镜像层，可以看到 2 因为下载的数据量要大很多，所以下载的时间要长许多，但是 1 中将 400M 的 gzip 还原成 1G 的原始数据却又消耗了太多时间，所以结果是两者总体时效是差不多的。

并且由于去掉 gzip，镜像下载不会抢占业务的 CPU 资源。自从上了这一方案后，业务抖动的次数明显减少了许多。

在蜻蜓的章节，镜像下载是镜像层从超级节点到物理机，在这一过程中，蜻蜓有一个动态压缩能力，比如使用更好的 lz4 算法，即达到了数据压缩的效果，又不会造成业务抖动。这就是图 3，整体上这一点在集团的应用效果很不错。

## 远程盘架构



解决了镜像下载对业务的干扰后，扩容、云环境的问题还没有解决。这时远程盘就派上用场了。

从上面的架构图看，集团有一套镜像转换机制，将原始的镜像层放在远端服务器上，第一个层都有一个唯一的远程盘与之对应。然后镜像中保存的是这个远程盘的id，这样做下来，远程盘的镜像可以做到kB级别。对于kB级别的镜像，下载耗时在1~2秒之间。

通过远程盘，解决了镜像下载的问题，同时由于远程盘放在物理机同一个机房，

容器运行时读取镜像数据，相当于从远程盘上面读取数据，因为在同一个机房，当然不能跟本地盘比，但是效果可以与云环境的云盘性能相媲美。

## 远程盘的问题

远程盘虽然解决了镜像下载的问题，但是所有镜像的数据都是从远程盘上读取，消耗比较大的网络带宽。当物理机上环境比较复杂时，远程盘的数据又不能缓存在内存时，所有数据都要从远端读取，当规模上来后，就会给网络带来不小的压力。

另外一个是，如果远程盘出现问题，导致 IO hang，对于容器进程来讲，就是僵尸进程，而僵尸进程是无法杀死的。对于应用来讲，一个容器要么是死，要么是活，都好办，死的了容器上面的流量分给活着的容器即可，但对于一个僵尸容器，流量没法摘除，导致这部分业务就受损了。

最后，远程盘因为要服务多台物理机，必然要在磁盘 IO 上面有比较好的性能，这就造成了成本较高。

## DADI

- 使用P2P技术
- 高效压缩算法
- 本机缓存



针对上述问题，集团采用的 DADI 这一项技术，都是对症下药。

- 使用 P2P 技术

远程盘是物理机所有数据都要从远程盘上读，这样会导致对远程盘机器的配置较高的要求，并且压力也很大。

而通过 P2P 手段，可以将一部分压力分担掉，当一台机器已经有另一台需要的数据时，它俩之间可以完成数据互传。

- 高效压缩算法

同样是解决网络带宽的问题，远程盘对应的数据都是没有压缩的，传输会占用比较多的带宽。

而 DADI 则在传输过程中，跟蜻蜓一样，对数据进行压缩，减少网络压力。

- 本机缓存

这个是核心买点了，当容器启动后，DADI 会把整个镜像数据拉取到本地，这样即使网络有问题，也不会导致容器进程僵尸，解决了业务受损的问题。

## 在 Web 级集群中动态调整 Pod 资源限制

阿里云容器平台技术专家 王程

阿里云容器平台技术专家 张晓宇(袁源)

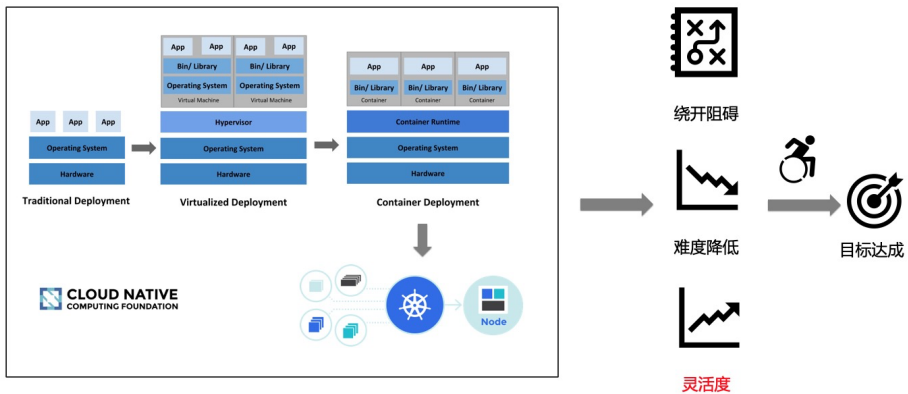
### 引子

不知道大家有没有过这样的经历，当我们拥有了一套 Kubernetes 集群，然后开始部署应用的时候，我们应该给容器分配多少资源呢？很难说。由于 Kubernetes 自己的机制，我们可以理解容器的资源实质上是一个静态的配置。如果发现资源不足，为了分配给容器更多资源，我们需要重建 Pod。如果分配冗余的资源，那么我们的 worker node 节点似乎又部署不了多少容器。试问，我们能做到容器资源的按需分配吗？这个问题的答案，我们可以在本次分享中和大家一起进行探讨。

首先允许我们根据我们的实际情况抛出我们实际生产环境的挑战。或许大家还记的，2018 的天猫双 11，一天的总成交额达到了 2135 亿。由此一斑可窥全豹，能够支撑如此大规模的交易量背后的系统，其应用种类和数量应该是怎样的一种规模。在这种规模下，我们常常听到的容器调度，如：容器编排，负载均衡，集群扩缩容，集群升级，应用发布，应用灰度等等这些词，在被超大规模集群这个词修饰后，都不再是件容易处理的事情。规模本身也就是我们最大的挑战。如何运营和管理好这么一个庞大的系统，并遵循业界 dev-ops 宣传的那样效果，犹如让大象去跳舞。但是马老师说过，大象就该干大象该干的事情，为什么要去跳舞呢。

## Kubernetes 的帮助

### Kubernetes 的帮助



大象是否可以跳舞，带着这个问题，我们需要从淘宝天猫等 APP 背后系统说起。这套互联网系统应用部署大致可分为三个阶段，传统部署，虚拟机部署和容器部署。相比于传统部署，虚拟机部署有了更好的隔离性和安全性，但是由于性能少，不可避免的产生了大量损耗。而容器部署又在虚拟机部署实现隔离和安全的背景下，提出了更轻量化的解决方案。我们的系统也是沿着这么一条主航道上运行的。假设底层系统好比一艘巨轮，面对巨量的集装箱——容器，我们需要一个优秀的船长，对它们进行调度编排，让系统这艘大船可以避开层层险阻，操作难度降低，且具备更多灵活性，最终达成航行的目的。

### 理想与现实

在开始之初，想到容器化和 Kubernetes 的各种美好场景，我们理想中的容器编排效果应该是这样的：

- 从容：我们的工程师脸上更加从容的面对复杂的挑战，不再眉头紧锁而是更多笑容和自信。

- 优雅：每一次线上变更操作都可以像品着红酒一样气定神闲，优雅地按下执行的回车键。
- 有序：从开发到测试，再到灰度发布，一气呵成，行云流水。
- 稳定：系统健壮性良好，任尔东西南北风，我们系统岿然不动。全年系统可用性 N 多个 9。
- 高效：节约出更多人力，实现“快乐工作，认真生活”。

然而理想很丰满，现实很骨感。迎接我们的是杂乱和形态各异的窘迫。杂乱是因为：作为一个异军突起的新兴技术栈，很多配套工具和工作流的建设处于初级阶段。Demo 版本中运行良好的工具，在真实场景下大规模铺开，各种隐藏的问题就会暴露无遗，层出不穷。从开发到运维，所有的工作人员都在各种被动地疲于奔命。另外，“大规模铺开”还意味着，要直接面对形态各异的生产环境：异构配置的机器，复杂的需求，甚至是适配用户的既往的使用习惯等等。

## 有时候，应用crash无处不在...



内存OOM?  
CPU throttle?  
关键应用交易量  
断崖式下跌...

除了让人心力交瘁的混乱，系统还面临着应用容器的各种崩溃问题：内存不足导致的 OOM，CPU quota 分配太少导致的，进程被 throttle，还有带宽不足，响应时延大幅上升 ... 甚至是交易量在面对访问高峰时候由于系统不给力导致的断崖式下跌等等。这些都使我们在大规模商用 Kubernetes 场景中积累非常多的经验。

## 直面问题

### 稳定性

问题总要进行面对的。正如某位高人说过：如果感觉哪里不太对，那么肯定有些地方出问题了。于是我们就要剖析，问题究竟出在哪里。针对于内存的 OOM，CPU 资源被 throttle，我们可以推断我们给与容器分配的初始资源不足。

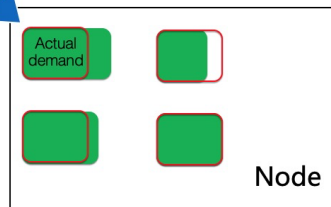
### 稳定性Stability



#### 不合理的资源初始分配



```
apiVersion: v1
kind: Pod
metadata:
  name: example
spec:
  containers:
  - name: example
    image: example-image
    resources:
      limits:
        cpu: "1"
        memory: "200Mi"
```



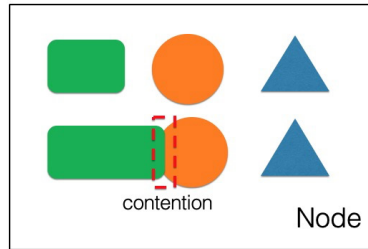
资源不足就势必造成整个应用服务稳定性下降的问题。例如上图的场景：虽然是同一种应用的副本，或许是由于负载均衡不够强大，或者是由于应用自身的原因，甚至是由于机器本身是异构的，相同数值的资源，可能对于同一种应用的不同副本并具有相等的价值和意义。在数值上他们看似分配了相同的资源，然而在实际负载工作时，极有可能出现的现象是肥瘦不均的。



## 稳定性Stability



### 应用容器资源竞争



而在资源 overcommit 的场景下，应用在整个节点资源不足，或是在所在的 CPU share pool 资源不足时，也会出现严重的资源竞争关系。资源竞争是对应用稳定性最大的威胁之一。所以我们要尽力在生产环境中清除所有的威胁。

我们都知道稳定性是件很重要的事情，尤其对于掌控上百万容器生杀大权的一线研发人员。或许不经心的一个操作就有可能造成影响面巨大的生产事故。因此，我们也按照一般流程也做了系统预防和兜底工作。在预防维度，我们可以进行全链路的压力测试，并且提前通过科学的手段预判应用需要的副本数和资源量。如果没法准确预算资源，那就只采用冗余分配资源的方式了。在兜底维度，我们可以在大规模访问量抵达后，对不紧要的业务做服务降级并同时主要应用进行临时扩容。但是对于陡然增加几分钟的突增流量，这么多组合拳的花费不菲，似乎有些不划算。或许我们可以提出一些解决方案，达到我们的预期。

### 资源利用率

回顾一下我们的应用部署情况：节点上的容器一般分属多种应用，这些应用本身不一定，也一般不会同时处于访问的高峰。对于混合部署应用的宿主机，如果能错峰分配上面运行容器的资源或许更科学。

## 资源利用率

### 潮汐应用：分时复用



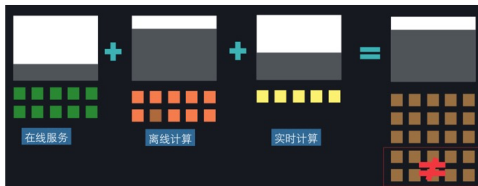
容器的资源需求并非一成不变，恰好满足才最“节能，环保”，但是怎么做



应用的资源需求可能就像月亮一样有阴晴圆缺，有周期变化。例如在线业务，尤其是交易业务，它们在资源使用上呈现一定的周期性，例如：在凌晨、上午时，它的使用量并不是很高，而在午间、下午时会比较高。打个比方：对于 A 应用的重要时刻，对于 B 应用可能不那么重要，适当打压 B 应用，腾挪出资源给 A 应用，这是一个不错的选择。这听起来有点像是分时复用的感觉。但是如果按照流量峰值时的需求配置资源就会产生大量的浪费。

## 资源利用率

### Resource packing: 减少资源碎片



在线 v.s. 离线

- 在线优先级高，延时敏感
- 离线优先级低，延时不敏感
- 低优先级牺牲
- 优先级互补性

在离线资源竞争



除了对于实时性要求很高的在线应用外，我们还有离线应用和实时计算应用等：离线计算对于 CPU、Memory 或网络资源的使用以及时间不那么敏感，所以在任何时间段它都可以运行。实时计算，可能对于时间敏感性就会很高。早期，我们业务是在不同的节点按照应用的类型独立进行部署。从上面这张图来看，如果它们进行分时复用资源，针对实时性这个需求层面，我们会发现它实际的最大使用量不是  $2+2+1=5$ ，而是某一时刻重要紧急应用需求量的最大值，也就是 3。如果我们能够数据监测到每个应用的真实使用量，给它分配合理值，那么就能产生资源利用率提升的实际效果。

对于电商应用，对于采用了重量级 java 框架和相关技术栈的 web 应用，短时间内 HPA 或者 VPA 都不是件容易的事情。先说 HPA，我们或许可以秒级拉起了 Pod，创建新的容器，然而拉起的容器是否真的可用呢。从创建到可用，可能需要比较久的时间，对于大促和抢购秒杀 - 这种访问量“洪峰”可能仅维持几分钟或者十几分钟的实际场景，如果我们等到 HPA 的副本全部可用，可能市场活动早已经结束了。至于社区目前的 VPA 场景，删掉旧 Pod，创建新 Pod，这样的逻辑更难接受。所以综合考虑，我们需要一个更实际的解决方案弥补 HPA 和 VPA 的在这一单机资源调度的空缺。

## 解决方案

### 交付标准

我们首先要对解决方案设定一个可以交付的标准：那就是“既要稳定性，也要利用率，还要自动化实施，当然如果能够智能化那就更好”。

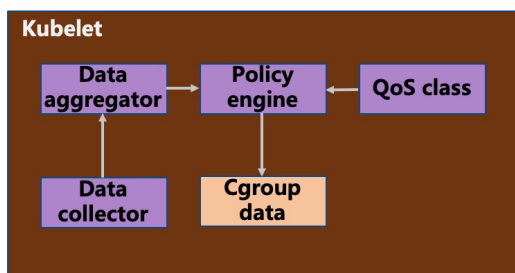
然后再交付标准进行细化：

- 安全稳定：工具本身高可用。所用的算法和实施手段必须做到可控。
- 业务容器按需分配资源：可以及时根据业务实时资源消耗对不太久远的将来进行资源消耗预测，让用户明白业务接下来对于资源的真实需求。
- 工具本身资源开销小：工具本身资源的消耗要尽可能小，不要成为运维的负担。

- 操作方便，扩展性强：能做到无需接受培训即可玩转这个工具，当然工具还要具有良好扩展性，供用户 DIY。
- 快速发现 & 及时响应：实时性，也就是最重要的特质，这也是和 HPA 或者 VPA 在解决资源调度问题方式不同的地方。

## 设计与实现

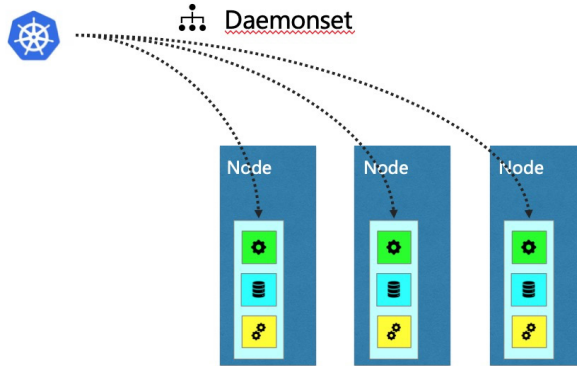
### 最初的设计



- Data collector – 数据采集
- Data aggregator – 容器画像
- Policy engine – 资源调整

上图是我们最初的工具流程设计：当一个应用面临很高的业务访问需求时，体现在 CPU、Memory 或其他资源类型需求量变大，我们根据 Data Collector 采集的实时基础数据，利用 Data Aggregator 生成某个容器或整个应用的画像，再将画像反馈给 Policy engine。Policy engine 会瞬时快速修改容器 Cgroup 文件目录下的的参数。我们最早的架构和我们的想法一样朴实，在 kubelet 进行了侵入式的修改。虽然我们只是加了几个接口，但是这种方式确实不够优雅。每次 kubernetes 升级，对于 Policy engine 相关组件升级也有一定的挑战。

## 演进 & 现状：如何快速迭代和交付？

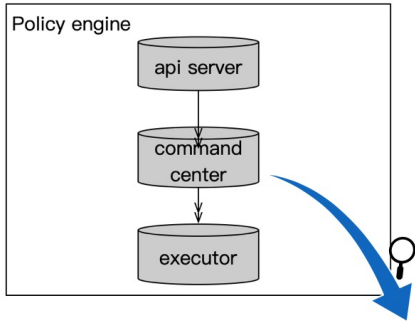


为了做到快速迭代并和 Kubelet 解耦，我们对于实现方式进行了新的演进。那就是将关键应用容器化。这样可以达到以下功效：

- 不侵入修改 K8s 核心组件
- 方便迭代 & 发布
- 借助于 Kubernetes 相关的 QoS Class 机制，容器的资源配置，资源开销可控。

当然在后续演进中，我们也在尝试和 HPA，VPA 进行打通，毕竟这些和 Policy engine 是存在着互补的关系。因此我们架构进一步演进成如下情形。当 Policy engine 在处理一些更多复杂场景搞到无力时，上报事件让中心端做出更全局的决策。水平扩容或是垂直增加资源。

## Policy engine 框架设计



### 设计原则

- 插件化
- 稳定：控制器、触发规则、不主动驱逐
- 自愈
- 不依赖先验知识



下面我们具体讨论一下 Policy engine 的设计。Policy engine 是单机节点上进行智能调度并执行 Pod 资源调整的核心组件。它主要包括 api server，指挥中心 command center 和执行层 executor。其中 api server 用于服务外界对于 policy engine 运行状态的查询和设置的请求；command center 根据实时的容器画像和物理机本身的负载以及资源使用情况，作出 Pod 资源调整的决策。Executor 再根据 command center 的决策，对容器的资源限制进行调整。同时，executor 也把每次调整的 revision info 持久化，以便发生故障时可以回滚。

指挥中心定期从 data aggregator 获取容器的实时画像，包括聚合的统计数据 and 预测数据，首先判断节点状态，例如节点磁盘异常，或者网络不通，表示该节点已经发生异常，需要保护现场，不再对 Pod 进行资源调整，以免造成系统震荡，影响运维和调试。如果节点状态正常，指挥中心会策略规则，对容器数据进行再次过滤。比如容器 cpu 率飙高，或者容器的响应时间超过安全阈值。如果条件满足，则对满足条件的容器集合给出资源调整建议，传递给 executor。

在架构设计上，我们遵循了以下原则：

- 插件化：所有的规则和策略被设计为可以通过配置文件来修改，尽量与核心控

制流程的代码解耦，与 data collector 和 data aggregator 等其他组件的更新和发布解耦，提升可扩展性。

- 稳定，这包括以下几个方面：
  - 控制器稳定性。指挥中心的决策以不影响单机乃至全局稳定性为前提，包括容器的性能稳定和资源分配稳定。例如，目前每个控制器仅负责一种 cgroup 资源的控制，即在同一时间窗口内，Policy engine 不同时调整多种资源，以免造成资源分配震荡，干扰调整效果。
  - 触发规则稳定性。例如，某一条规则的原始触发条件为容器的性能指标超出安全阈值，但是为避免控制动作被某一突发峰值触发而导致震荡，我们把触发规则定制为，过去一段时间窗口内性能指标的低百分位超出安全阈值；如果规则满足，说明这段时间内绝大部分的性能指标值都已经超出了安全阈值，就需要触发控制动作了。
  - 另外，与社区版 Vertical-Pod-Autoscaler 不同，Policy engine 不主动驱逐腾挪容器，而是直接修改容器的 cgroup 文件。
- 自愈：资源调整等动作的执行可能会产生一些异常，我们在每个控制器内都加入了自愈回滚机制，保证整个系统的稳定性。
- 不依赖应用先验知识：为所有不同的应用分别进行压测、定制策略，或者提前对可能排部在一起的应用进行压测，会导致巨大开销，可扩展性降低。我们的策略在设计上尽可能通用，尽量采用不依赖于具体平台、操作系统、应用的指标和控制策略。

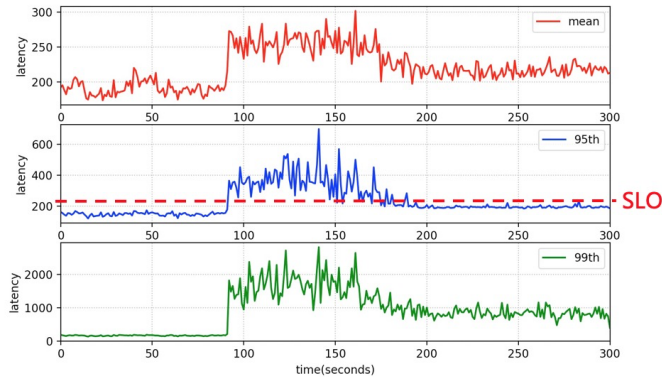
在资源调整方面，Cgroup 支持我们对各个容器的 CPU、内存、网络和磁盘 IO 带宽资源进行，目前我们主要对容器的 CPU 资源进行调整，同时在测试中探索在时分复用的场景下动态调整 memory limit 和 swap usage 而避免 OOM 的可行性；在未来我们将支持对容器的网络和磁盘 IO 的动态调整。

## 调整效果

### 调整效果



Latency 95th (95百分位)  $\leq$  250ms



上图展示了我们在测试集群得到的一些实验结果。我们把高优先级的在线应用和低优先级的离线应用混合部署在测试集群里。SLO 是 250ms，我们希望在线应用的 latency 的 95 百分位值低于阈值 250ms。在实验结果中可以看到，在大约 90s 前，在线应用的负载很低；latency 的均值和百分位都在 250ms 以下。到了 90s 后，我们给在线应用加压，流量增加，负载也升高，导致在线应用 latency 的 95 百分位值超过了 SLO。在大约 150s 左右，我们的小步快跑控制策略被触发，渐进式地 throttle 与在线应用发生资源竞争的离线应用。到了大约 200s 左右，在线应用的性能恢复正常，latency 的 95 百分位回落到 SLO 以下。这说明了我们的控制策略的有效性。

## 经验和教训

下面我们总结一下在整个项目的进行过程中，我们收获的一些经验和教训，希望这些经验教训能够对遇到类似问题和场景的人有所帮助。

- 避开硬编码，组件微服务化，不仅便于快速演进和迭代，还有利于熔断异常服务。



- 尽可能不要调用类库中还是 alpha 或者 beta 特性的接口。例如我们曾经直接调用 CRI 接口读取容器的一些信息，或者做一些更新操作，但是随着接口字段或者方法的修改，共建有些功能就会变得不可用，或许有时候，调用不稳定的接口还不如直接获取某个应用的打印信息可能更靠谱。
- 基于 QoS 的资源动态调整方面：如我们之前所讲，阿里集团内部有上万个应用，应用之间的调用链相当复杂。应用 A 的容器性能发生异常，不一定都是在单机节点上的资源不足或者资源竞争导致，而很有可能是它下游的应用 B、应用 C，或者数据库、cache 的访问延迟导致的。由于单机节点上这种信息的局限性，基于单机节点信息的资源调整，只能采用“尽力而为”，也就是 best effort 的策略了。在未来，我们计划打通单机节点和中心端的资源调控链路，由中心端综合单机节点上报的性能信息和资源调整请求，统一进行资源的重新分配，或者容器的重新编排，或者触发 HPA，从而形成一个集群级别的闭环的智能资源调控链路，这将会大大提高整个集群维度的稳定性和综合资源利用率。
- 资源 v.s. 性能模型：可能有人已经注意到，我们的调整策略里，并没有明显地提出为容器建立“资源 v.s. 性能”的模型。这种模型在学术论文里非常常见，一般是对被测的几种应用进行了离线压测或者在线压测，改变应用的资源分配，测量应用的性能指标，得到性能随资源变化的曲线，最终用在实时的资源调控算法中。在应用数量比较少，调用链比较简单，集群里的物理机硬件配置也比较少的情况下，这种基于压测的方法可以穷举到所有可能的情况，找到最优或者次优的资源调整方案，从而得到比较好的性能。但是在阿里集团的场景下，我们有上万个应用，很多重点应用的版本发布也非常频繁，往往新版本发布后，旧的压测数据，或者说资源性能模型，就不适用了。另外，我们的集群很多是异构集群，在某一种物理机上测试得到的性能数据，在另一台不同型号的物理机上就不会复现。这些都对我们直接应用学术论文里的资源调控算法带来了障碍。所以，针对阿里集团内部的场景，我们采用了这样的策略：不对应用进行离线压测，获取显示的资源性能模型。而是建立实时的动态容器画像，

用过去一段时间窗口内容器资源使用情况的统计数据作为对未来一小段时间内的预测，并且动态更新；最后基于这个动态的容器画像，执行小步快跑的资源调整策略，边走边看，尽力而为。

## 总结与展望

总结起来，我们的工作主要实现了以下几方面的收益：

- 通过分时复用以及将不同优先级的容器（也就是在线和离线任务）混合部署，并且通过对容器资源限制的动态调整，保证了在线应用在不同负载情况下都能得到足够的资源，从而提高集群的综合资源利用率。
- 通过对单机节点上的容器资源的智能动态调整，降低了应用之间的性能干扰，保障高优先级应用的性能稳定性
- 各种资源调整策略通过 Daemonset 部署，可以自动地、智能地在节点上运行，减少人工干预，降低了运维的人力成本。

展望未来，我们希望在以下几个方面加强和扩展我们的工作：

- 闭环控制链路：前面已经提到，单机节点上由于缺乏全局信息，对于资源的调整有其局限性，只能尽力而为。未来，我们希望能够打通与 HPA 和 VPA 的通路，使单机节点和中心端联动进行资源调整，最大化弹性伸缩的收益。
- 容器重新编排：即使是同一个应用，不同容器的负载和所处的物理环境也是动态变化的，单机上调整 pod 的资源，不一定能够满足动态的需求。我们希望单机上实时容器画像，能够为中心端提供更多的有效信息，帮助中心端的调度器作出更加智能的容器重编排决策。
- 策略智能化：我们现在的资源调整策略仍然比较粗粒度，可以调整的资源也比较有限；后续我们希望让资源调整策略更加智能化，并且考虑到更多的资源，比如对磁盘和网络 IO 带宽的调整，提高资源调整的有效性。
- 容器画像精细化：目前的容器画像也比较粗糙，仅仅依靠统计数据和线性预测；刻画容器性能的指标种类也比较局限。我们希望找到更加精确的、通用的、反

映容器性能的指标，以便更加精细地刻画容器当前的状态和对不同资源的需求程度。

- 查找干扰源：我们希望能找到在单机节点上找到行之有效的方案，来精准定位应用性能受损时的干扰源；这对策略智能化也有很大意义。

## 开源计划

如果大家对于我们的项目代码感兴趣的话，预计 2019 年 9 月份，我们的工作也将出现在阿里巴巴开源项目 OpenKruise (<https://github.com/openkruise>) 中，敬请期待！

## 大规模 k8s 集群下的巡检

阿里巴巴技术专家 陈杰（韩堂）

蚂蚁金服高级开发工程师 马金晶（楚贤）

### 议程

先看一下我们这次分享的主要内容，首先我们会讨论一下，在 K8S 中，我们为什么需要巡检，然后会介绍一下，一个简单的巡检服务应该怎么做比较合适，再之后，将会深入介绍一下如果打造一下完整的巡检系统以及自愈生态系统，最后我们会展望一下巡检系统的未来。

### 大规模 K8S 的现状

那下面我们进入正题，首先我们看一下目前一个较大规模 K8S 集群的现状，第一，存在数以万计的节点，这些节点也会存在较大差异，比如，在硬件上的差异（CPU，内存大小的差异，有的节点还会有 GPU），同时在操作系统和软件方面也会存在差异，比如，有不同版本的 Kubelet，这是节点的情况。我们再看看 Pod 情况，在这上万台节点之上，可能会有数以十万计的 Pod，这些 Pod 也会存在较大的差异，包括，Pod 的用途，有的是用于跑长期的服务的，有的是 Job 类型，有的是实时的，有的是离线计算的，等等。再看看我们的自定义资源，随着业务的不断增长，我们的自定义资源数量会达到百这个级别，对应的 Controller，也会达到上百的级别，最典型的例子就是集群在大量部署了 Service Mesh (Istio) 以及 Serverless (Knative) 服务之后，繁多的自定义资源数量就会上百了。可以看出，随着集群规模和业务的不断扩大，集群中的各类资源都会不断的增长。

那下面这张图就很形象的表达了大规模的 K8S 集群的现状



## 我们怎么样确保集群状态的一致性？

我们知道 K8S 的关键词就是最终一致性，所有的 Controller 都会朝着最终一致性不断 sync，但是一个 K8S 集群一定能达到理想的最终一致性么？一个较小的集群做到这一点还是比较容易的，但是像我们前面介绍的如此之大规模的集群，还是有一点难度的，于此同时，我们还注意到目前一个 Controller 只会关注一种资源的同步，那其实在集群之外，我们考虑的是，更综合更全面的情况。

这里我们看一下一些典型的集群状态不一致的（不符合预期）的例子：

- IP 冲突

这个为用户最能感知到的不符合预期的了，一旦 IP 冲突就回导致服务时断时续，如果是关键服务，还可能会造成各种故障，这是挺严重的一个异常，现有 K8S 是没有针对这种情况的检查和的。

- 元数据未同步

我们知道，绝大部分情况，在 Pod 创建完之后，还会有元数据同步到其他服务上，在 Pod 交付之后，还会有外围的服务进行操作（部署、导流等等），这些外部服务会依赖某些元数据，一旦这些元数据未及时同步完整，会导致后续操作的失败。

基于上面的这两个例子，我们可以看出我们的 K8S 集群中存在一些不一致，影响整体健康度的问题，再加上前面介绍的集群规模不断扩大，一些不一致和异常会被不断的放大，所以，为了达到我们所期望的一致性，我们需要一种方法去深入我们的 K8S 内部，我们称之为“巡检”。

## 基于巡检服务的一些考量

那我们看一下，我们的巡检服务需要什么的特性：

第一点，高效性，我们的巡检需要处理的资源是很多的，上万节点，十万级的 pod，再加上上百的自定义资源，这些自定义资源很可能每个 pod 都会有一份，那就更多了。

第二点，扩展性，这么大的 K8S 集群不仅仅是一个小团队维护，大家都会有巡检的需求，每个小团队又可能关注的点是不同的，综合起来，巡检对应的关注点就会非常多。

第三点，可观测，找出了这些不一致，怎么查看，怎么针对性的区分出紧急程度来做监控也是非常重要的。

## 我们构建的巡检服务 V1

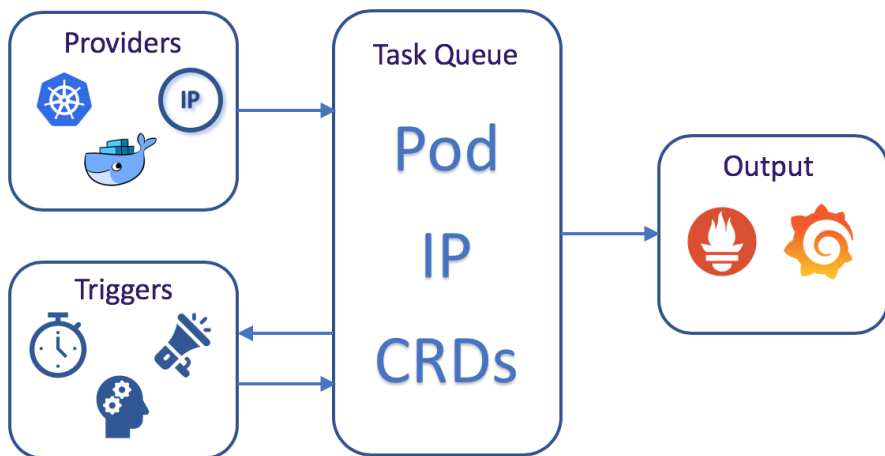
那我们看一下，一个巡检可能长什么样的：

**Providers:** 巡检所使用的数据源，例如：K8S 集群中的所有资源，节点上的容器信息

**Triggers:** 巡检的起点，有定时的触发，人工手动触发，还有用户根据事件进行触发

**Task Queue:** 被触发的所有任务都会进入任务队列进行运行，得出的结果会输出，我们的最初的实践是由 Prometheus 直接采集汇总

这里用一句话来描述一下就是：以集中化的 Providers 方式提供集群当前所有的信息以及资源状态，再通过适当的手段触发对应的巡检任务，由巡检任务进行计算是否有异常，并最终将结果导出到统一的存储中。



## 巡检服务 V1 存在的问题

前面介绍的这个巡检模型是我们刚开始在实践中使用的，但是随着任务数以及集群规模的增长，我们慢慢的发现了一些不足，主要的问题有：

- 性能问题

我们的巡检是每个集群中以单个 Pod 的方式跑的，上面提到的所有操作都是打包到一个二进制文件中运行的，这种方式有下面这些不足，所有的巡检跑在一起，最终会被单机性能锁限制，这个是显而易见的。同时在大量的巡检在跑的时候，由于任务队列是有限的，会出现问题争抢的问题，总的来说，不具有横向的伸缩性。

- 扩展性

另外一个不足就是缺乏扩展性，例如 Trigger 比较单一，可快速接入的手段有限，K8S 中会有各种事件，这些其实都可以作为 Trigger 接入到巡检服务中，再比如巡检结果的输出，结果单一，属于定制化的，如果想支持多种输出，需要再次进行定制，耗时费力。

为了解决上面这些问题，我们综合考虑之后决定引入 Knative。

## 使用 Knative 可以解决的一些问题

为什么要使用 Knative 呢?

第一点，是使用 Serverless 之后，我们的巡检任务在跑的时候，有很大的横向伸缩性，同时也不会有巡检任务争抢的问题的。

第二点，我们知道，Knative 原生支持 CloudEvent 的，结合巡检触发的特征，CloudEvent 非常适合来触发我们的巡检，可以将我们的巡检入口无限的扩展。

第三点，基于 Knative，我们的巡检可以做成一个框架服务，而不仅仅是内部使用的特有服务。

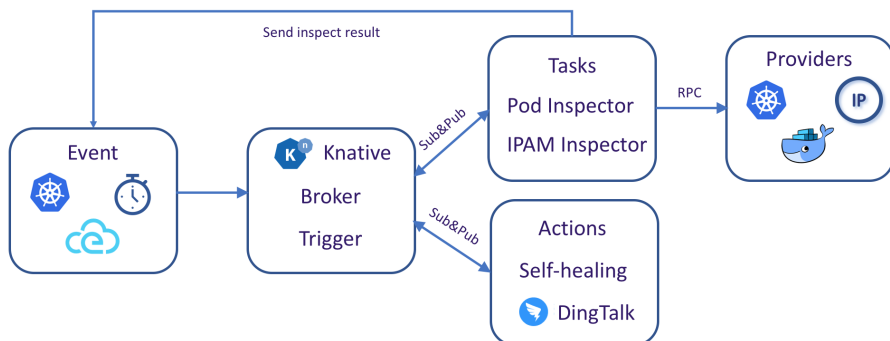
## 基于 Knative 重构的巡检服务 V2

基本上的思路和 V1 是一致的，我们这里将 V1 中的 Task 改造成了 Knative 中的 Function，Providers 以独立服务的方式部署在外部，Tasks 使用 RPC 的方式调用 Providers，这里相比 V1 主要将 Output 替换成了 Actions，Actions 其实和 Tasks 比较类似，这里描述一下一个正常的巡检任务流程：

1. Task 向 Knative 订阅自己关心的 Event，例如 K8S Pod Event
2. 有新的 K8S Pod Event 发送到 Knative
3. Knative 将事件转发给对应的 Task
4. Task 根据自己的巡检逻辑配合 Providers 提供的数据进行核对
5. 将结果继续以 Event 的形式返回给 Knative
6. Action 订阅对应的巡检结果 Event
7. Knative 将巡检结果发送给 Action
8. Action 根据自己的逻辑进行下一步

我们可以看到整个一套巡检服务是一个完整的闭环，同时又有很大扩展性和兼容性。我们这里的 Actions 可以根据自己的需求进行任意的定制，例如，将结果发送到钉钉，或者触发自愈系统工作等等，我们这里的 Event 是可以根据 CloudEvent 进行无限的扩展的。





到这里为止，我们都是在描述怎么样去做一个巡检服务，仅仅是简单的将 K8S 中的一些异常以及不符预期的资源找出来，那么这些结果怎么使用，怎么形成一套体系化的巡检生态呢？我们下面就会继续深入的讨论。

## 如何体系化打造我们的巡检生态

刚才我们蚂蚁的同事马金晶介绍了在一个实际的大规模 k8s 生产集群当中，他发现存在大量的异常，他介绍了过去我们巡检服务的一个演进过程，以及演进到今天，我们说拥抱 serviceless，用了更 cloud native 的方式来解决巡检服务的性能和扩展性问题。巡检服务只是其中的一部分，或者一个开始，一个起点，接下来我们是如何基于这个起点，来体系化的去打造我们的整个巡检生态。

首先我们总结一下我同事看到的那些问题，你会发现，他说的，其实是基础设施（服务器、网络、存储）和各业务之间的稳定性工作还存在一个很大的 gap，那我们用 k8s 去做资源编排之后，光靠监控系统是很难发现这么多异常，并解决这些异常，我们思考的是说，要用巡检的方式去串联整个基础设施，和各个业务之间的一个稳定性，去弥补这个大 gap，为 k8s 上面跑的各各业务，去提供全方位的一个稳定性保障。

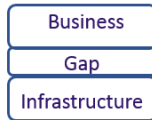
除了监控系统本身解决不了这么多异常，另外监控系统自己也会存在数据本身失真一个问题，那么监控系统本身的问题如何去发现，我们说是也可以通过巡检来做到，并且除了监控系统，我们整个 k8s 生产集群当中，会存在大量的这种依赖系统，

比如我们的 cmdb, ipam 等等，这些系统都会存在自己的一份数据，这些数据跟 k8s 集群的数据都会有一定的关联，比如这个 ip，在几乎所有的系统里面都有，到底谁的数据是真的，谁的数据是假的，需要去分辨，需要有方法去保证一致，最终保证数据是正确的，是没有异常的。

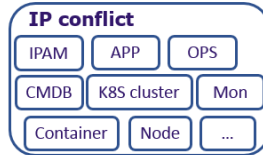
那么我们其实是说从 k8s 本身以及周边这样一个更大的系统闭环，去考虑问题，去设计系统。

---

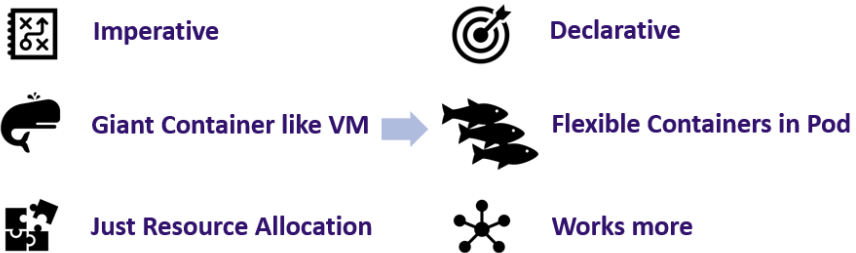
### 1) A huge gap between the stability of infrastructure and business



### 2) Lack of global perspective of status inspection



那么我们说希望从 k8s 本身以及周边这样一个更大的系统闭环，去考虑问题，去设计系统；去做一套覆盖 k8s 核心、周边配套、托管的业务，做到一个全视角的巡检平台，从而实现我们提升 k8s 集群的一个稳定性目标，进而保障 k8s 集群上面跑的业务的一个稳定性。



同时我们在基于 k8s 这样的云原生的架构演进当中，整个 k8s 编排相比以前，

发生了太多的变化。比如，我们说 api 从命令式变成声明式。我们的容器的运维方式发生了变化，从富容器到具有更扩展的云原生方式。另外更多的动作沉淀到我们这一层。比如服务发现。因此我们的巡检的内容相比以前发生了变化。

因此我们有必要在新的云原生架构演进当中，去看看 k8s 的巡检要解决的问题到底有哪些，或者说看看我们 k8s 编排系统的巡检的价值点在哪里，以及它的边界在哪里？从而去怎么更好的设计我们整个的巡检平台或生态。

## 如何确定调度系统中巡检的边界

首先我们看看它和基础设施的一个边界，这里的基础设施是指我们的服务器硬件、我们的网络、我们的 IDC 等等，如果你跑在云上，那云就是你的基础设施。那 k8s 的稳定肯定是要依赖于底层基础设施的稳定，那我们可以把底层基础设施的监控和巡检结果作为我们 k8s 巡检的一个输入；

然后我们再看看与上层应用的一个边界，我们可以针对 k8s 中应用的 pod 或者 container 的状态做巡检，但这些巡检都不会涉及到应用自身业务逻辑的巡检，比如电商交易系统里面的业务对账，资损检查和控制等。

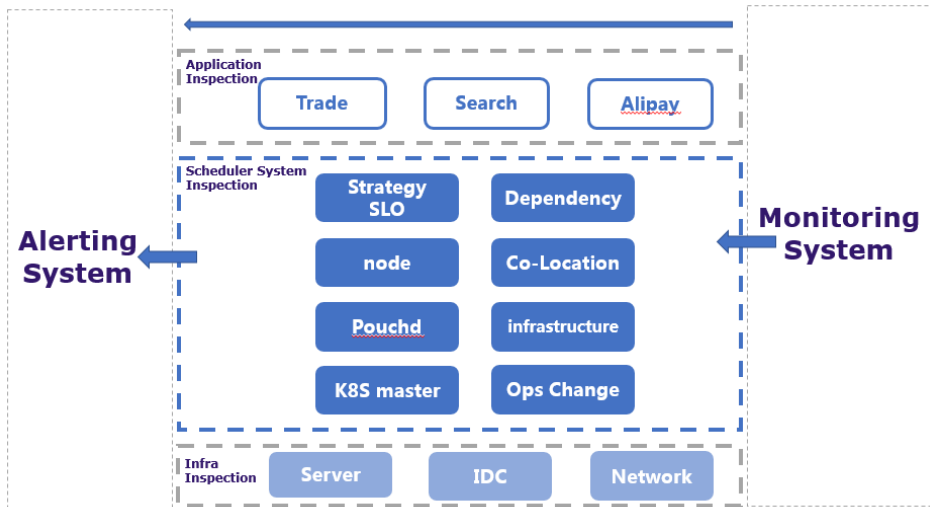
再看看和监控告警等系统的区别：

首先监控，大家如果做运维或 sre, 都应该非常熟悉，它可以通过 metrics 对系统进行监测，从而判断系统是否运行良好。同时监控系统往往是一个独立的闭环，它一般有自己的数据通道，有自己独立的 agent 去采集数据，往往数据源也是单一的。

那巡检呢，我们说往往是根据既定的一些预期，或者按照预期的目标或规则，定期的对系统进行检查，从而判断系统是否符合预期，是否良好、是否需要维修。这里需要强调的是说，我们巡检更多的是通过多种渠道拿到的数据，而不是单一的数据通道，并且需要针对的场景进行计算，从某种意义上来说，巡检系统是在监控系统之上，巡检更要面向用户，面向业务，面向目标。

最后我们再来看看告警系统，大家都知道告警是在系统运行的过程当中，出现异常的时候，有一种途径去通知到人，通知到我们的运维同学，包括我们说有电话、短信、钉钉、Dashboard 等不同的通知方式，但总体来说，告警主要是通知，要保障

它产生的指标是必须是时效、必须是精准的、也必须是可读的。



那一个实际的 k8s 生产集群当中到底有哪些价值的事情呢？

1) 首先一个呢，是这个 node/pouchd 的巡检 (pouchd 是我们内部的一个类似 docker 的容器技术) node 和 pouchd(dockerd) 是 k8s 集群当中的两个运行平台，node 和 pouchd 平台的稳定是容器健康运行、比如 k8s 调度成功的关键基础，node 和 pouchd 平台的环境巡检是 k8s 巡检平台中最核心的部分。

2) 第二个是 k8s 调度核心链路的巡检

核心链路，我们说包括扩缩容和发布，这里面要涉及到中心 Master，单机 kublet，还有持续存储 Etcd 的一个运行状态巡检，以及说通过我们创建的容器，它的状态是否符合我们的预期、比如说内存是否存在超卖，cpu 是否存在堆叠，ip 是否存在冲突等。

3) 第三个是外部依赖系统的巡检

在实际的 k8s 生产集群当中，外部依赖非常多，比如我们的 cmdb、ip 分配系统 ipam，还有很多容器运行时的外部依赖，比如存储、网络 vpc 等等，是否按照预期去设置。

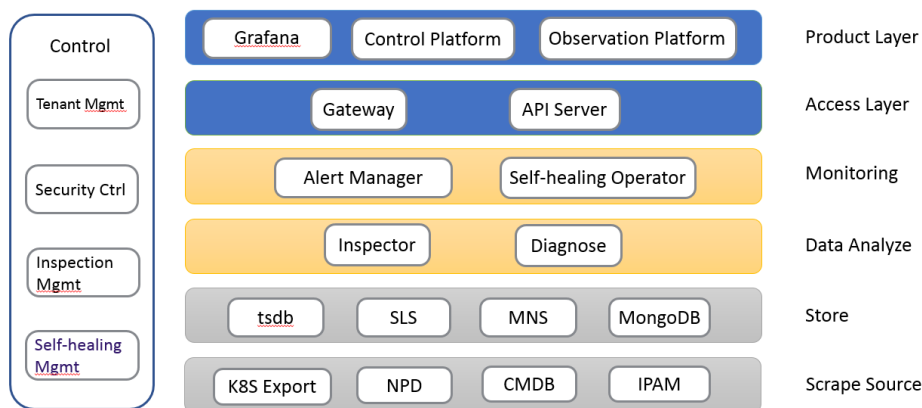
#### 4) 最后一个事件的巡检

k8s 内部有很多事件，除了这个外，还有各种外围的运维操作系统触发的变更事件，多个变更同时发生。需要做好关联以及统计分析，去发现集群是否按照预期进行，去发现潜在的异常，去发现真正有价值的事件。

## 如何做巡检自愈闭环

当然我们说巡检系统的目的不是为了巡检而去巡检，更多的是对巡检之后的结果进行分析和决策，发现 k8s 集群中潜在的稳定性风险，并且有能力提前排除，或者在故障发生时，或者我们通过巡检的结果去快速定位问题，去找到导致故障发生的原因。

因此这里面很自然想到的一个事情是做一个修复，当然我们首先第一步是说要发现问题后去手动修复，手动修复是我们的第一步，在一个大规模集群当中，这样是很费人力的，我们希望做成自动化的，一个自愈的闭环。如何打造这个闭环呢？



在整个巡检的自愈闭环系统里面，它首先要解决的是各种数据的采集，我们通过开源的 k8s export，还有 kube-state-metrics，同时在单机上采用兼容社区 npd 的方式，增加 export 机制，可以更好的对接社区生态中的 prometheus)，基于这些方式来采集数据。

有了数据采集之后，首先想到的是如何存储，我们用了多种方式，大部分 metrics 存储到基于 prometheus 的数据平台上，还有我们的事件，我们可以存储到 es，消息中间件上，还有我们的 pod,node 数据我们可以存在 mysql 中，也可以存储到 mongodb 中，做到 freeschema。

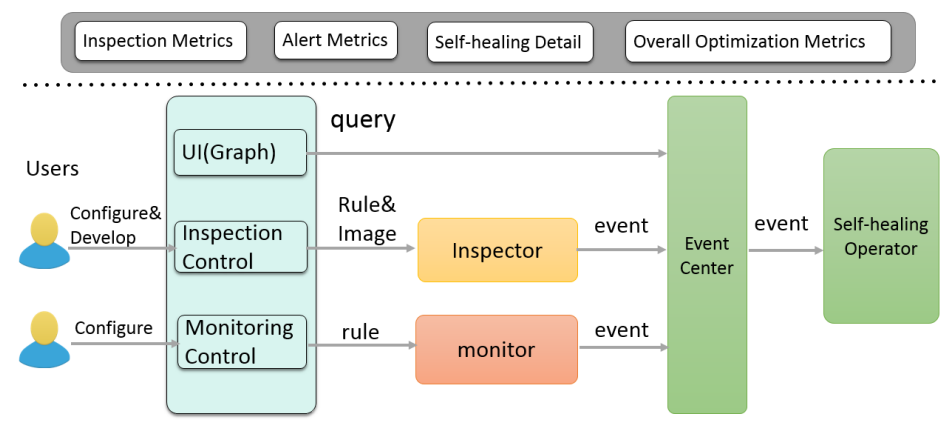
数据持久化之后呢，我们就可以通过对各路数据做巡检的计算了。前面我们的码金晶同事也讲了，我们在这里做了一套 serverless 的巡检计算框架，能够很方便的去让 k8s 系统的各方人员去定制自己的巡检逻辑。并通过格式化后的巡检结果，去对接告警，以及自愈引擎去完成修复。

最后通过 api 开放底层的能力，去对接上层的各种产品。比如这种可观察性产品。同时我们从下到上，无论是数据的采集，数据的存储，数据的分析，还是整个监控自愈的引擎，都需要去做控制和管理。比如我们的自愈引擎这一块，它是一种写操作，盲目的自动化会是一种毁灭性的打击，如果是大批量的自愈，如果存在 bug，整个集群的业务都存在被摧毁的可能，因此我们需要通过一定的安全控制策略，如我们常见的这种限流，熔断，预案开关，以及灰度策略机制，我们叫抽样自愈机制。去更安全的完成修复。

## 用户如何交互

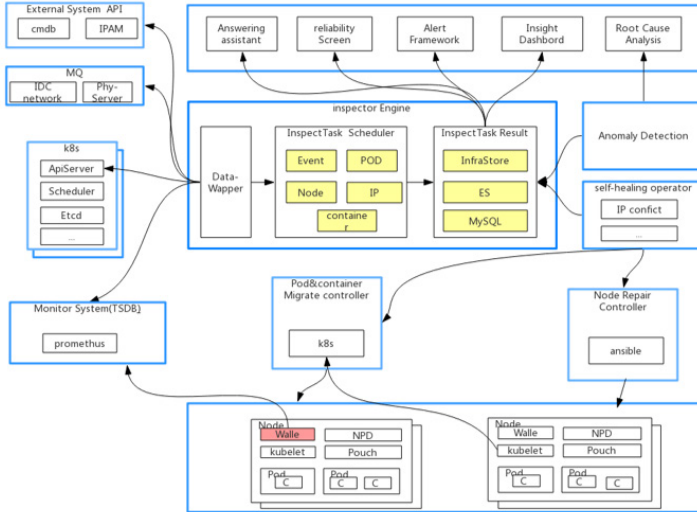
我们来看看从用户的视角，是如何和这个巡检平台进行交互的，我们还是以传统的监控系统的交互为例，我们实际的生产当中也是用了监控系统，比如基于 prometheus 改造的监控系统，我们早期也是简单的基于 grafana，或者 prometheus 提供的 promql 或 UI 去配置监控。但集群规模和用户数量上去后呢，我们开始去不断完善一套更好的产品或统一的入口对接用户，在这里可以更友好的配置监控。同样的对于巡检来说，也需要这样类似的东西，我们需要对巡检进行配置，在不同的集群里面，需要去开启不同的巡检，哪些巡检需要报警，报警的阈值是怎么样的，这些都需要有更好的管理。同时我们新一代的巡检系统也在支持更好的代码接入，比如用户提交一个 image，可以更好的支持多语言。未来实际上更有想象空间，用户可以写 yarm 或者 UDF 的形式来定制巡检。

另外整个平台需要提供一个管控，可以让用户看到自己优化的指标，驱动他去不断的去配置和调试优化自己的监控和巡检。比如说 sre 需要去关注真个巡检的异常数量，它的这个准确率，覆盖度，以及收敛的情况。另外这里面要强调的是自愈的一些指标和详情，刚才也一直强调自愈是非常危险的，你需要关注整个修复的一个进度和风险。比如你在自愈的过程当中腾挪了一个容器，有可能这个时候业务发生了故障，需要通过审计去更好的确定是否是因为我们腾挪导致的故障。等等这样一些自证清白或者说追溯和跟踪的能力都需要建设好。



## 巡检引擎的设计和实现

接下来，我们再看一眼这个巡检和自愈引擎的工程实现细节，从这个图中，我们看，单机中，我们通过一个类似 NPD 的单机采集程序采集单机 metrics 存储到数据平台中，同时我们把 k8s 里面的 pod,node,event 也导出来，并且我们把基础设施的事件以及外部系统的数据导入进来，通过不同的 provider 接入我们的巡检引擎中，在这个巡检引擎中把一个个独立的巡检任务进行调度计算，得到巡检结果，最后通过存储把巡检的结果存储到不同的存储中。同时自愈 operator 可以去 watch 到巡检事件，通过调用迁移腾挪 controller 完成容器的自愈。这是一条线。另外一条是上面，我们也把巡检的结果通过 api 进行输出和展示，比如我们的观察系统。



## 巡检结果的处理

我们稍微看一下我们的巡检观察页面长什么样。

从我们这个页面看，我们对巡检进行了归类展示，主要是为了将更重要的巡检提醒到 sre 等运维同学，同时我们也可以看到一种异常巡检的详情，这样让运维同学可以快速的洞察到集群的异常在哪里。同时这些异常也会关联展示在其他页面里面。

**ASI Insight** 用户手册 / Aone 提需求 / 直接反馈问题 / 阿里云智能事业群 运营平台

集群列表

业务健康

集群SLO

资源 Insight

节点与 Pod

通用搜索

应用视图

资源对象

发布与扩容

应用 Insight

应用观察

**平台组件可用性问题**

记录(条) 30

超卖 10 / 资源规则 5 / 脏数据 15

**外部依赖可用性问题**

记录(条) 30

镜像类 10 / 网络Overlay-ENI 5 / CMDB(Skyline) 15

**调度质量SLO问题**

记录(条) 25

组件异常 5 / 组件版本一致性 5 / 异常处理 15

**其他**

记录(条) 10

潜在风险 10

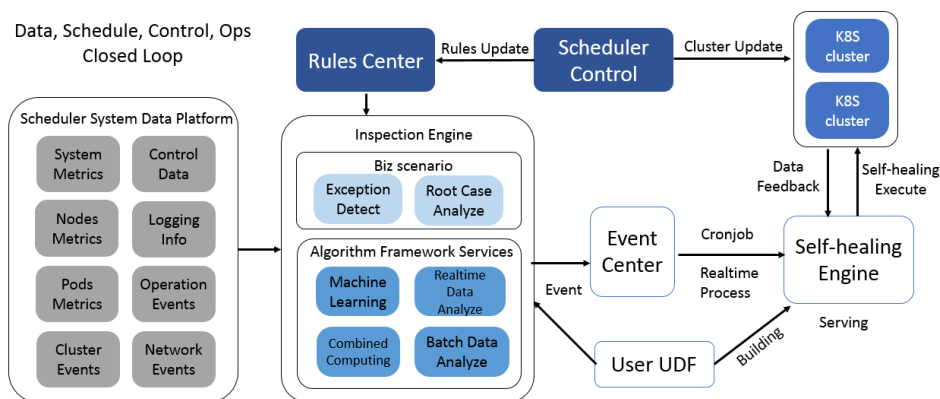
分类	指标项	是否符合SLO	SLO承诺	现状概览	异常详情
平台组件可用性	超卖	不正常	10000	2000	<a href="#">查看详情</a>
平台组件可用性	资源规则	正常	10000	2000	<a href="#">查看详情</a>
平台组件可用性	脏数据	正常	10000	2000	<a href="#">查看详情</a>
外部依赖可用性	拉镜像成功率	正常	100	90	<a href="#">查看详情</a>
外部依赖可用性	拉镜像成功率	正常	100	90	<a href="#">查看详情</a>
外部依赖可用性	网络Overlay-ENI	正常	100	90	<a href="#">查看详情</a>
外部依赖可用性	CMDB(Skyline)	正常	100	90	<a href="#">查看详情</a>
调度质量SLO	组件异常	正常	200	80	<a href="#">查看详情</a>



## k8s 集群运维如何通过巡检走向 AIOPS

最后，我们看看巡检他在整个 k8s 运维平台里面处于一个什么样的位置，或者说它未来可以发展成什么样子。这里我想把巡检引擎升级成巡检洞察引擎。更多的是说，我们引入运维经验，引入专家知识，同时我们基于数据驱动，引入机器学习等进行巡检计算。

具体怎么做呢，我想当我们把 k8s 编排系统里面的各种数据收集到我们的数据平台后，比如 prometheus，我们就可以扩展它的架构，通过它去对接 kafaka,blink 做实时计算，当然也可以阿里云的 MaxComputer 做批量计算，利用 tensorflow 等做机器学习。并且基于数据平台和这些计算框架，再沉淀我们的算法框架或算法服务，提供更多的通用能力给这种业务场景去复用，比如异常检测，根因分析；同样，还是把这些计算出来的异常通过事件的方式给到我们的自愈引擎，去发起自愈操作，完成自动修复。在这里，无论是巡检洞察，还是自愈执行引擎，都可以提供给业务方更好的方式来接入，比如脚本式的 UDF。最终通过这样一种方式，走向 AIOPS。



## 使用 Istio 管理跨地域多集群的服务

阿里云容器平台高级技术专家 王夕宁 (贝叮)

Backend Architect UniCareer 刘晓忠

看到这个主题，大家可能第一个反应是为什么需要跨地域的多集群？背后的诉求是什么？首先，第一点是业务一致性、灾难恢复的诉求，不把鸡蛋放在一个篮子里，避免由于一个集群的不稳定导致业务出现问题；

第二点，由于业务应用可能是面向分布在全球的终端用户，那么为了减少服务之间的访问延时，这些业务应用服务就需要被部署在全球相应的地域；与此同时，有些特定业务，由于它的特殊性，需要满足本地数据合规要求，也需要将这一些应用服务部署在对应的地域中。

第三点，当前的网络连接能力，特别是一些云厂商提供的云企业网、高速通道等，使得打通跨地域跨集群之间的网络连接的门槛大大降低，成本也大大降低，这一点就加速了跨地域跨集群的应用更加平民化。

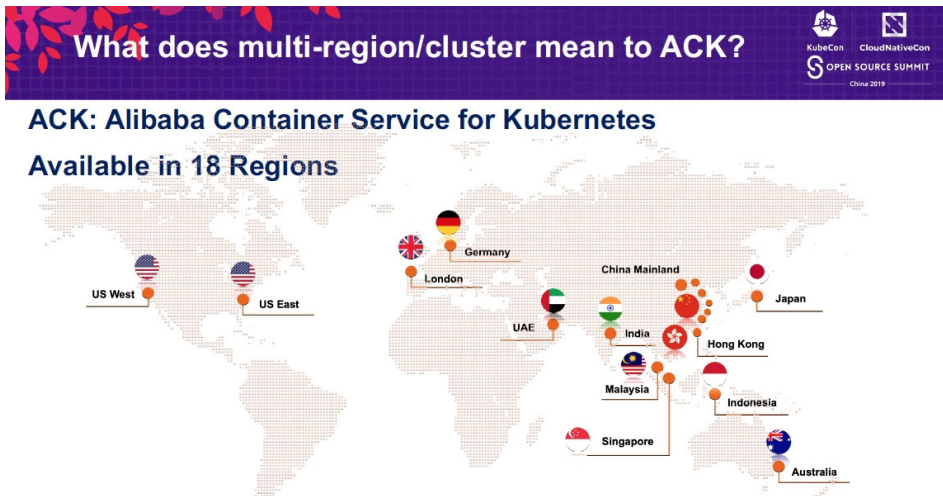
### Why need the multiple regions/clusters?

KubeCon CloudNativeCon  
OPEN SOURCE SUMMIT  
China 2019

1. Business Continuity/Disaster Recovery
2. Geographically distributed end users
  - Reduce the access latency
  - Meet local legal and data regulatory compliance
3. Lower barrier to setup the network connectivity b/w regions & clusters

而跨地域多集群对于 ACK ( 阿里云容器服务 Kubernetes 的简称 ) 来说意味着

什么？截至目前为止，ACK 服务已经在全球多个国家和地区 18 个地域开通。这对于一些在海外有业务的国内客户或者在国内有业务的国外客户来说，他们的业务应用就可以很容易地在 ACK 上进行跨地域多集群的管理。



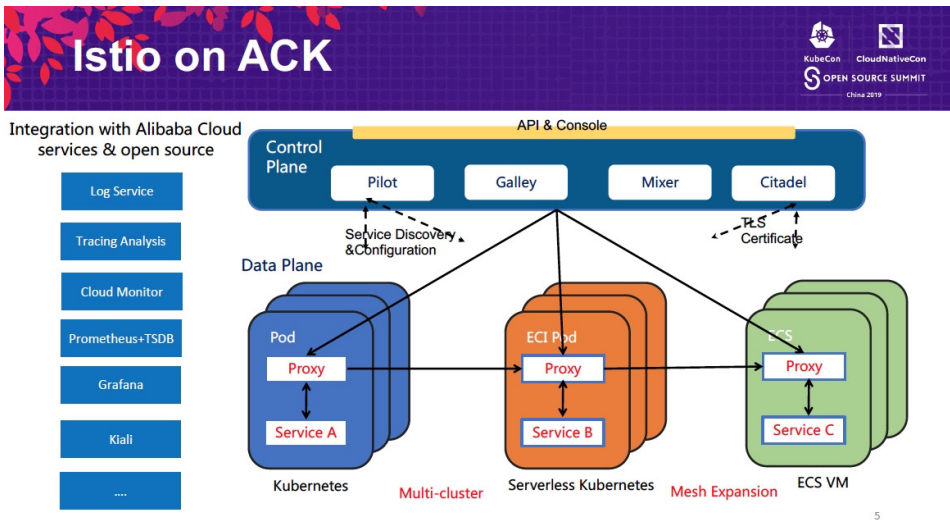
利用 Istio 服务网格，我们可以管理 Kubernetes 集群上面的不同服务之间的流量，基于社区 Istio，阿里云容器服务 ACK 又提供了哪些扩展与集成能力。

重点介绍 4 个方面：

1. 我们在开源社区基础之上，优化集成了阿里云的其他服务。大家知道 Istio 社区版本中自带的一些开源组件在生产环境中直接使用并不可靠，譬如分布式跟踪组件 Jaeger 社区版本中并没有整合数据存储部分，Prometheus 的数据存储等，而 ACK Istio 中通过优化整合阿里云服务，提供了更加稳定易用的服务能力。
2. 我们一直帮助用户更容易地去使用 ACK 上的 Istio，我们根据实际中得到的一些最佳实践，通过 UI 控制台或者 API 方式输出，包括简化 Istio 从安装部署到升级的生命周期管理、配置参数的校验分析和组件的自我恢复能力等。
3. 我们提供了混合部署能力，除了 Kubernetes 集群，我们还可以把 ECS 虚拟机集成到一个 Istio 服务网格中。这样可以帮助现有虚拟机上的应用，逐渐

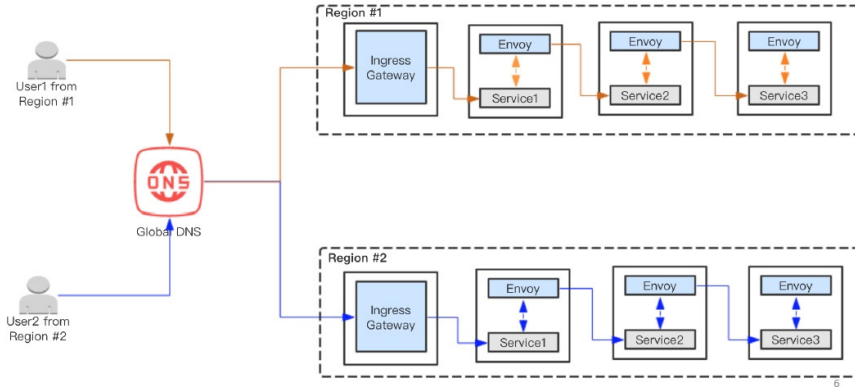
地迁移到容器平台上。此外，Istio 和 Serverless Kubernetes 是天作之和，虚拟节点中 ECI 实现的 POD，可以通过 Istio 和托管服务节点中的 Pod 实现互联互通和统一管理。

4. 基于阿里云的跨集群的网络能力，我们实现了多集群下的 Istio 管理。通过 Istio 将多集群下的应用服务实现互联互通和统一管理。



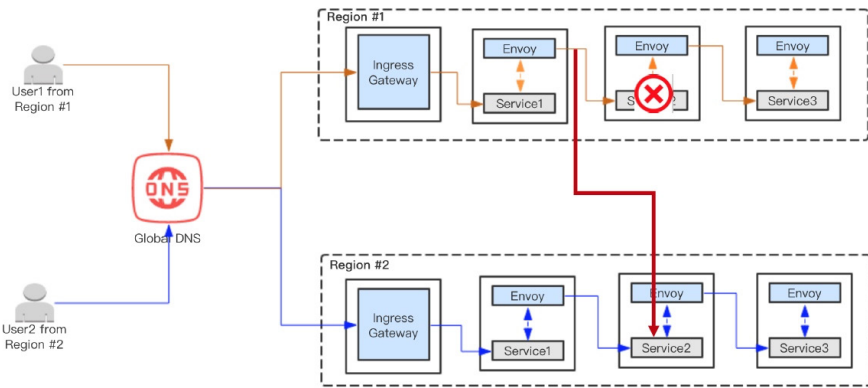
对于跨地域多集群来说，一个重要的诉求是要支持服务就近访问原则。应用服务分别部署在不同地域的 Kubernetes 集群上，例如图中杭州和北京的集群上分别运行了 service1、service2 以及 service3，按照 Locality based service routing 原则，本地域集群上的 service1 会优先调用同一地域上的 service2，同样地，service2 也会优先访问同一地域上的 service3。基于 Envoy 提供的 zone aware 能力，在 Istio 中，会根据服务调用方的 envoy 所在的地域信息，去匹配最合适的目标服务，并把流量优先转发过去。

## Feature Request for multi-region/cluster - Locality based service routing



而当某一地域的服务出现故障时，Istio on ACK 能够将服务请求故障转移到健康检查正常的地域。如图所示，随着地域 #2 的 service2 的请求量增大，在 ACK 中可以通过设置启用 HPA(Horizontal Pod Autoscaling)，使用定制的进行 pod 的自动伸缩。

## Feature Request for multi-region/cluster - Automatic Failover



总结来说，Istio 1.0 开始就提供了多集群管理能力，通过一个控制平面来管理

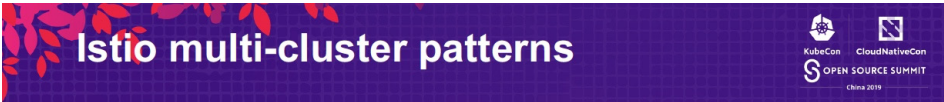
多个集群上的服务，前提是这些集群采用 flat network，Pod 网络互通并且 Pod/Service 的 CIDR 网段不能重叠。

到了 Istio 1.1，这种单一控制平面的模式继续支持，同时为了支持更多其他的网络模式，例如集群之间无法网络互通，Pod/Service 的 CIDR 网段有重叠等，这种情况下集群之间的交互只能通过对外暴露的公网端口，所以 Istio 1.1 提出了通过 Istio Gateway 来实现多集群的工作负载统一管理。

在单一控制平面下，无论是上述哪种方式，对于 Istio 的用户来说，部署管理 Kubernetes 的应用负载、定义 Istio 的 Virtual Service、Destination Rule 等都保持一致，就像操作单个集群一样。

此外，Istio 1.1 开始也支持 Mesh Federation 能力，也就是可以在每个集群上各自部署一个 Istio 的控制平面，并且每个控制平面只会管理自己集群内的服务端点。通过使用 Istio 网关、公共根证书颁发机构 (CA) 以及服务条目 ServiceEntry，可以将多个集群配置组成一个逻辑上的单一服务网格。

Istio on ACK 已经通过使用 Operator 机制支持了单一控制平面方式，并且近期也会发布对 Mesh Federation 的支持。

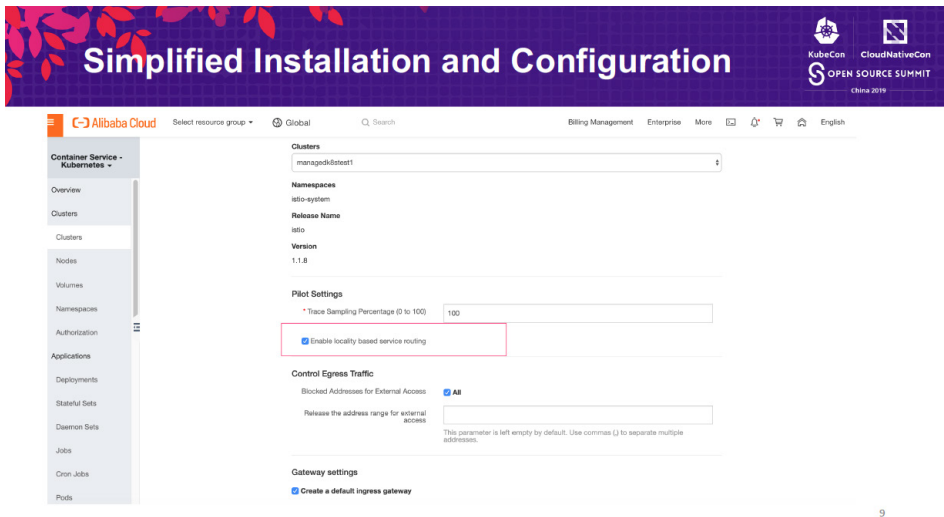


The slide features a purple header with the title "Istio multi-cluster patterns" in white. To the right of the title are logos for KubeCon, CloudNativeCon, and the Open Source Summit China 2019.

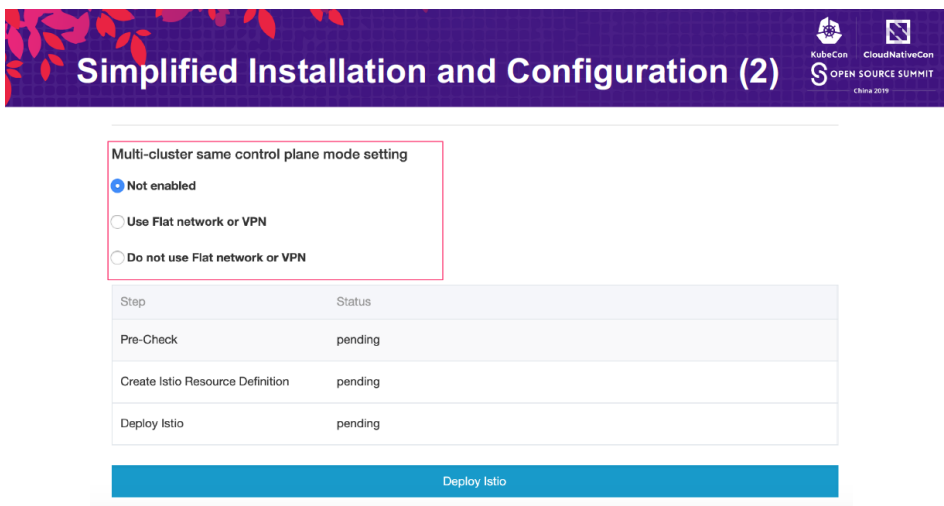
	Single Mesh		Mesh Federation
Istio 1.0 * Support for Istio 1.0 ends on June 19th, 2019	<ul style="list-style-type: none"> <li>Single Control Plane</li> <li>Flat network</li> <li>All pod CIDRs in every cluster must be routable to each other</li> <li>Pod/Service CIDR ranges must be unique</li> </ul>		
Istio 1.1	<ul style="list-style-type: none"> <li>Single Control Plane</li> <li>Flat network</li> <li>All pod CIDRs in every cluster must be routable to each other</li> <li>Pod/Service CIDR ranges must be unique</li> </ul>	<ul style="list-style-type: none"> <li>Single Control Plane</li> <li>No VPN network</li> <li>Inter-connectivity through gateways</li> <li>Pod/Service CIDR ranges may be overlapped</li> </ul>	<ul style="list-style-type: none"> <li>Multiple Control Planes</li> <li>No VPN network</li> <li>Inter-connectivity through gateways</li> <li>Pod/Service CIDR ranges may be overlapped</li> </ul>

在 ACK 控制台上，根据我们的一些最佳实践以及云上客户的使用反馈，我们把一些常用的配置参数化，以 UI 方式提供简单的配置，这样用户就可以快速部署一套

Istio，通过勾选，就可以启用或者禁止服务就近访问能力。



在多集群管理方面，通过配置项，可以决定是否启用多集群模式，并指定相应的网络拓扑方式。



以上这些能力，在后台是如何实现的呢？那就是 Istio on ACK 上提供的 Istio Operator 组件。Istio Operator 定义了一个描述 Istio 部署所需的自定义资源，当前定义了两个自定义资源 CRD：Istio 与 RemoteIstio。通过 CRD 来抽象 Istio 常用的使用场景，包括服务治理的配置、多集群管理等。

每一个运行 Istio 控制平面的 Kubernetes 集群上都需要定义一个 Istio 类型的自定义资源，其中里面定义了一些配置参数，如是否启用服务就近访问、是否启用多集群模式、采用哪种多集群部署方式、是否启用 TLS 认证等。

而在多集群单一控制平面模式下，为每一个远程加入的集群，都会定义一个 RemoteIstio 类型的自定义资源，里面的定义相对比较简单，例如指定启用自动注入 sidecar 的命名空间等。

Custom Resource for Istio Operator


  
OPEN SOURCE SUMMIT  
China 2019

Each K8s cluster running Istio control plane should have one Istio CR(custom resource)

```

1 apiVersion: istio.alibabacloud.com/v1beta1
2 kind: Istio
3 metadata:
4   labels:
5     controller-tools.k8s.io: "1.0"
6   heritage: Tiller
7   release: istio
8   name: istio-config
9 spec:
10  hub: registry-vpc.cn-hangzhou.aliyuncs.com/aliacs-app
11  version: 1.1.8
12  controlPlaneSecurityEnabled: false
13  mtls:
14    enabled: false
15  security:
16    selfSigned: true
17  meshExpansions:
18    enabled: false
19  includeIPRanges: "*"
20  excludeIPRanges: ""
21  outboundTrafficPolicy:
22    mode: "REGISTRY_ONLY"
23  logging:
24    level: "default:info"
25  dockerImage:
26    region: ""
27  gateways:
28    enabled: true
          
```

One RemoteIstio custom resource is created for each remote cluster

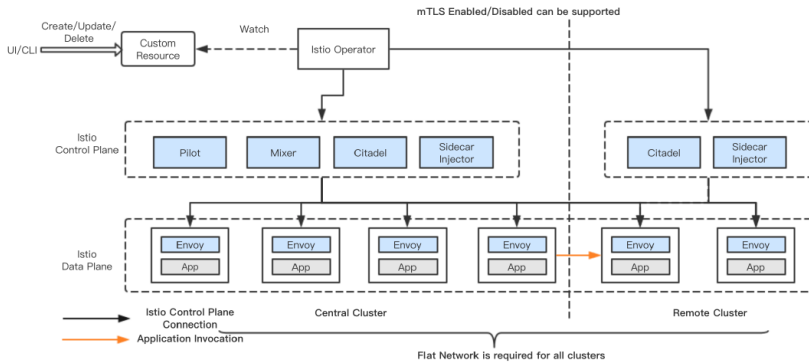
```

1 apiVersion: istio.alibabacloud.com/v1beta1
2 kind: RemoteIstio
3 metadata:
4   name: localityaware-bj
5   namespace: istio-system
6 spec:
7   autoInjectionNamespaces:
8     - default
9   dockerImage:
10    region: cn-beijing
11   gateways:
12     enabled: true
13     ingress:
14       - enabled: true
15         k8sIngress: {}
16   hub: registry-cn-beijing.aliyuncs.com/aliacs-app-catalog
17   includeIPRanges: "*"
18   proxy: {}
19   security: {}
20   sidecarInjector:
21     enabled: true
22     replicaCount: 1
          
```

在 Istio on ACK 上，通过使用 Istio Operator，监听定义的自定义资源，如果其中一个或多个发生更改，该 Operator 会自动协调组件的状态以匹配其新的所需状态。如下图所示，Istio Operator 除了管理部署本地集群的 Istio 组件之外，还会处理远程集群的 Istio 组件部署，并提供同步机制，从远程集群提供对 Istio 控制平面组件的访问。



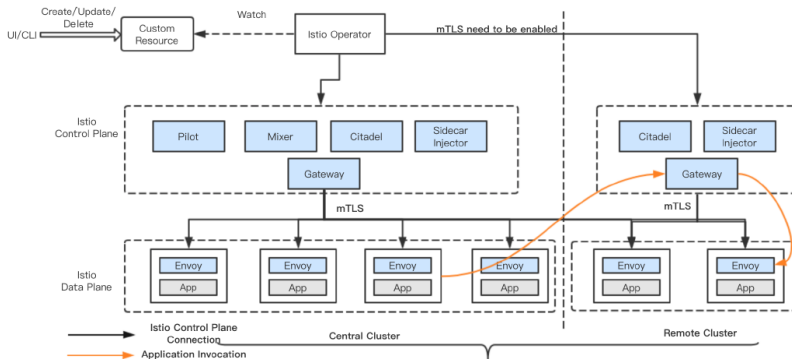
## Multi-cluster support of Istio on ACK - Flat network



11

除了 Flat network 模式之外，Istio on ACK 也支持在单一控制平面下通过网关进行连接的方式。在社区 Istio 中，提供了通过 Helm Chart 方式进行部署，但整个过程的参数配置比较复杂，根据我们公有云用户的一些反馈，这种方式容易出错，并且后期运维升级管理中会比较麻烦。而通过 Istio Operator 简化了 Istio 组件的部署和生命周期管理，支持无缝升级，支持高效的多集群管理。

## Multi-cluster support of Istio on ACK - Gateway Connected

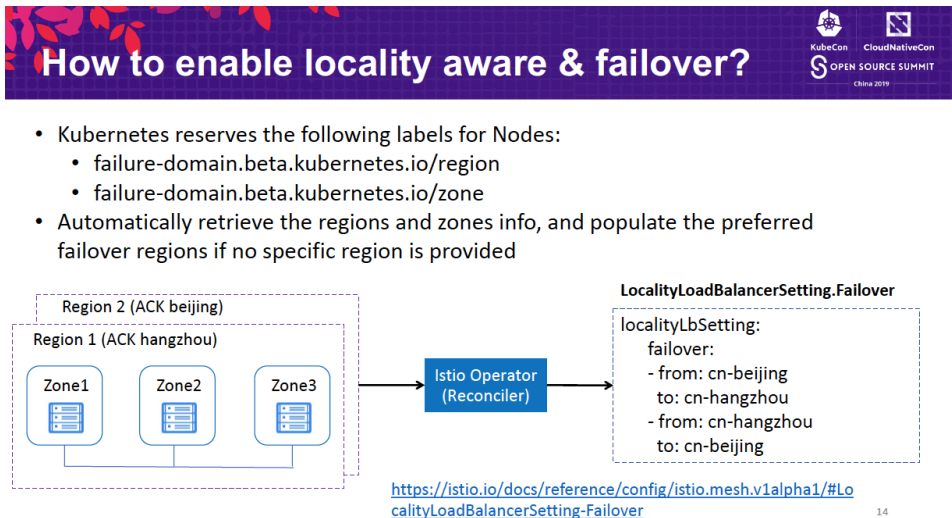


1. Flat Network is NOT required for all clusters;
2. Cross cluster mTLS requires one shared Root CA;

12

在跨地域多集群情况下，应用服务被部署到不同的地域上，Istio on ACK 是怎么支持服务就近访问与故障自动转移呢？我们知道，Kubernetes 提供了 2 个标签 `failure-domain.beta.kubernetes.io/region` 和 `failure-domain.beta.kubernetes.io/zone`，用于标示节点所在的地域信息。

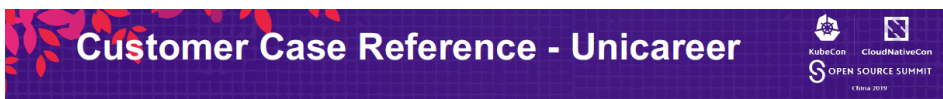
Istio Operator 通过 Reconciler 组件会同步到所有服务所在的地域信息。对于每一个服务请求来说，如果没有特别指定服务接收方的地域，Operator 就会结合阿里云上的地域网络情况，推荐出最适合的地域上的服务。



接下来是我们的客户 UniCareer 分享，看看他们在使用 Istio on ACK 做跨地域多集群微服务治理的经验。首先会简单介绍一下客户的业务场景，其次是这些场景下碰到的主要问题，然后会讲一下针对这些问题采用 Istio on ACK 的方案和一个简单 Demo。

先来看一下 UniCareer 的业务场景：UniCareer 是一家面向全球学生和在职专业人士的各种需求的 E-Learning 职业发展平台。它的应用将会同时服务来自亚洲、美洲、欧洲、澳洲等各地域的用户，对用户体验有相当高的要求。同时，因为业务的特殊性，开发迭代非常快，不同的子项目使用了不同的编程语言来各自的微服务。为了更好地支持服务运行，同时考虑到没有专职的运维人员，Unicareer 研发团队最开

始使用了阿里云的容器服务 Kubernetes，在此基础之上，特别需要一套比较成熟又简单易用的微服务治理方案，最好同时能与 Kubernetes 原生集成。



### Business Scenario

- Serving global users including China Mainland, America, Europe and Australia, etc.
- No dedicated operations team.
- Need one easy-to-use microservice management solution for our full chain of web apps

### Key Problems

- Stable high latency across regions.
- Occasional extreme latency due to cross regional network fluctuation.
- Existing K8s Ingress is weak and inconvenient.
- Multiple programming languages support.
- Decouple application logic from service infrastructure.
- Need to integrate with K8s ecosystem.

15

在这些情况下，UniCareer 遇到了以下一些关键问题：

- 单地域部署应用导致跨地域访问有一定的高延迟；
- 跨地域网络不稳定导致应用访问偶尔延迟非常高；
- 之前使用的 Kubernetes Nginx Ingress 功能薄弱，基于 Annotation 的配置比较繁琐，而云服务商的扩展不通用；
- 业务所用技术没有相应成熟的微服务治理方案，即便像 Java 的成熟方案如 Dubbo、Spring Cloud 等对开发人员要求较高且没有很好的办法与 Kubernetes 原生集成；

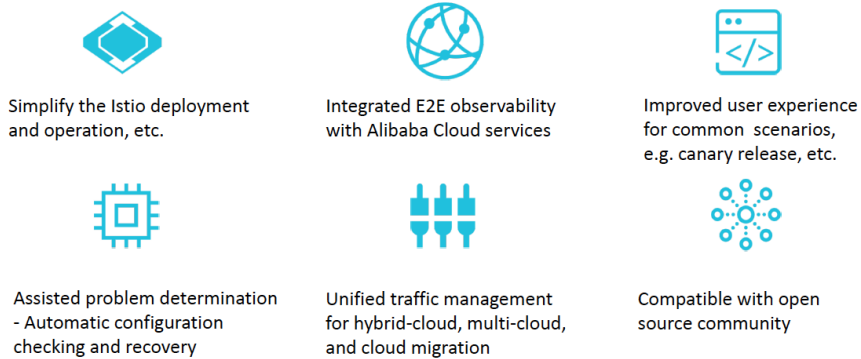
UniCareer 在 2018 就开始针对 Istio on ACK 提供的能力进行了技术评估，最终认为可以从以下 6 个维度的功能获得帮助：

- 简化 Istio 的部署运维以及生命周期的管理
- 端到端的可观测性能力的整合；

- 对一些常用的场景提供了良好的用户体验；
- 提供了问题诊断模块，自动检查配置项是否合理并给修复建议；
- **多集群下的统一流量管理，适合于多云混合云，可以保持云厂商中立；**
- 兼容社区开源版本，以便可以及时获得社区最新功能；

## Benefit from Istio on ACK

KubeCon CloudNativeCon  
OPEN SOURCE SUMMIT  
China 2019

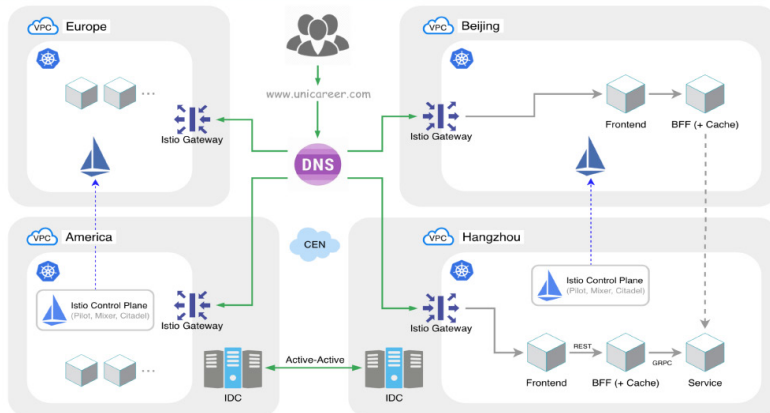


18

下面通过这个拓扑图来讲解一下 UniCareer 采用的方案：

## Services running on cross-regional K8s clusters

KubeCon CloudNativeCon  
OPEN SOURCE SUMMIT  
China 2019



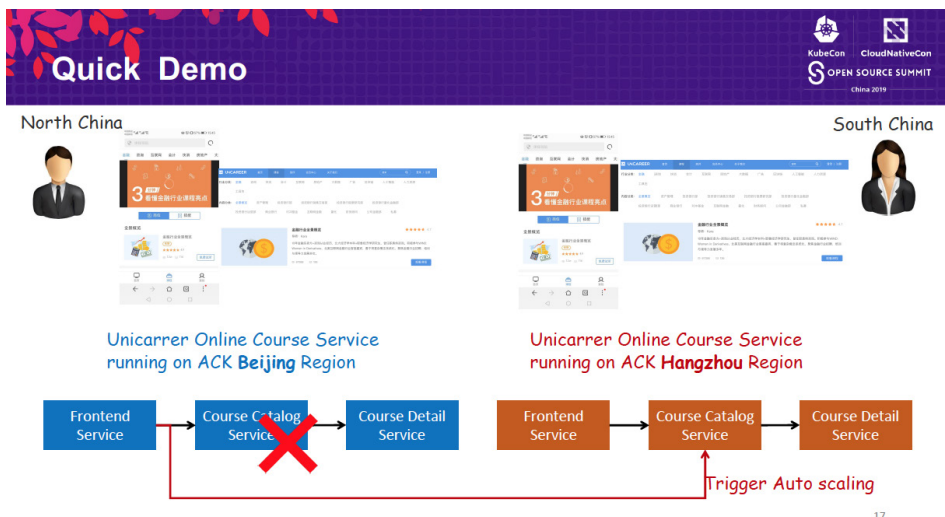
16

图上的 4 个灰块代表 4 个地域，地域内部的白色区域代表 Kubernetes 集群。左边两个地域，右边两个地域，各自就近组成了一个 mesh，是类似的拓扑。以右边的 beijing 和 hangzhou 地域组成的 mesh 为例。

- 通过专线 ( 阿里云企业网 ) 将多集群网络拉到同一平面；
- 采用 Istio 并打通跨地域多集群的单一控制平面拓扑；
- 策略配置由控制平面统一管理并下发到 remote 集群；
- 用户请求按地域 DNS 分流到最近地域的负载均衡及 Kubernetes 集群

请求进到集群之后经过 Istio 的 Locality based Routing 功能路由到本地 endpoints，并且在一个地域的实例全部 down 掉之后可以 failover 到其他地域。现在 Istio 1.1 原生支持了这项功能: Frontend + BFF(+Cache) 在各地域集群同步部署以便就近访问，GRPC Service 与 DB 交互频繁，仅在 IDC 所在地域集群部署。目前业务按地域单元化，IDC 全球双活，数据走阿里云的专用通道进行实时同步。

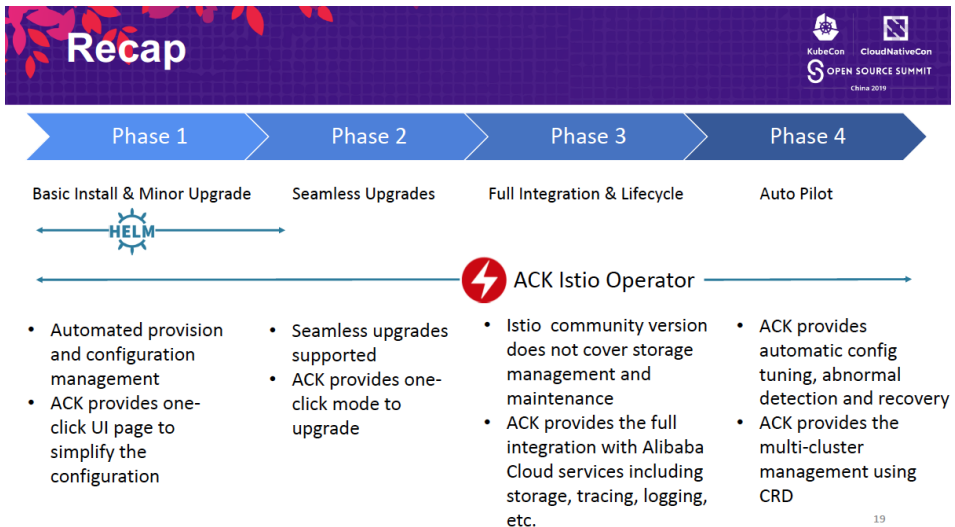
我们简单看一下一个脱敏之后的 failover 的 demo:



同样以 beijing 和 hangzhou 地域为例，中国北部和南部的用户会就近访问 UniCareer 的应用，前端应用也会请求本地的 API 服务。当本地的 API 服务 down

掉之后,后续请求将被 failover 到互备地域的 API 服务。同时切换之后因为承受了更多流量,如果服务器压力比较大,由于开启了 HPA, API 服务也会自动扩容。

最后,总结一下 Istio on ACK。从去年 0.8 版本开始,基于社区版本,在 ACK 上提供了基于 Helm Chart 的部署运维管理,但在实际客户使用过程中发现存在诸多问题,譬如说如何做到无缝平滑升级、如何有效地管理 Istio 与其他服务的整合及其整个生命周期。在此基础上,Istio on ACK 推出了 Istio Operator,简化 Istio 组件的部署和生命周期管理、在实际客户生产环境做到无缝升级、优化参数配置及异常检测与自动回复、并且很好地支持多集群模式。



# 阿里云开源贡献

---

## 首个普惠社区的平民化方案：GPU 共享调度

阿里云容器平台高级技术专家 张凯

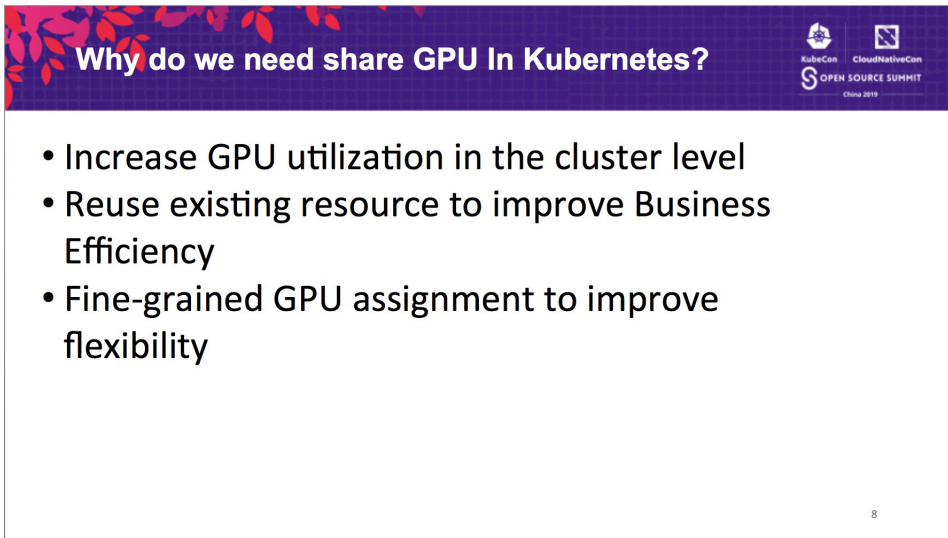
阿里云容器平台技术专家 车漾（必燃）

### GPU 共享调度 – 首个普惠社区的平民化方案

#### 我们为什么需要共享 GPU

Gartner 对全球 CIO 的调查显示人工智能将成为 2019 年组织革命的颠覆性力量，而利用 Docker 和 Kubernetes 代表云原生技术为 AI 提供了一种新的工作模式，将 GPU 机器放到统一的资源池，进行调度和管理，这避免了 GPU 资源利用率低下和人工管理的成本。

因此全球主要的 Kubernetes 容器服务厂商都提供了 Nvidia GPU 容器集群调度能力，但是通常都是将一个 GPU 卡分配给一个容器。这可以实现比较好的隔离性，确保使用 GPU 的应用不会被其他应用影响；但是由于 GPU 的自身成本很高，大量用户都希望通过共享 GPU 提升集群 GPU 使用率。



**Why do we need share GPU In Kubernetes?**

- Increase GPU utilization in the cluster level
- Reuse existing resource to improve Business Efficiency
- Fine-grained GPU assignment to improve flexibility

8

KubeCon CloudNativeCon  
OPEN SOURCE SUMMIT  
China 2019

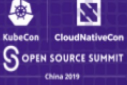
大家就有了共享 GPU 的集群调度需求：能够让更多的模型开发和预测服务共享同一个 GPU 卡上，进而提高集群中 Nvidia GPU 的利用率。而这就需要提供 GPU 资源的划分，这里 GPU 资源划分的维度指的就是 GPU 显存和 Cuda Kernel 线程的划分。在集群级别谈支持共享 GPU，通常是两件事情：

1. 调度，将可以共享 GPU 的应用调度到同一个 GPU 设备上。
2. 隔离，我们这里主要讨论的是调度，隔离的方案目前需要用户通过应用限制（比如使用 tensorflow 的 `perprocessgpumemoryfraction` 来控制），未来会基于 Nvidia 的 MPS, 也会考虑 vGPU。




## 共享 GPU 的挑战

The Challenge of Sharing GPU in Kubernetes



- Scheduling
  - Kubernetes exclusive GPU assignment, can't be shared
  - Device Plugin and Scheduler make decision independently
- Isolation
  - NVIDIA GRID is for the Hypervisor, not for Kubernetes whose `runc` is default container runtime
  - MPS is only for Volta and is not ready for the production



9

而细粒度的 GPU 卡调度，目前 Kubernetes 社区并没有很好的方案，这是由于 Kubernetes 对于 GPU 这类扩展资源的定义仅仅支持整数粒度的加加减减，无法支持复杂资源的分配。比如用户希望使用 Pod A 占用半张 GPU 卡，这在目前 Kubernetes 的架构设计中无法实现资源分配的记录和调用。

### GPUSharing: 业界唯一开源的 GPU 共享调度方案

针对此问题，我们设计了一个 Out Of Tree 的共享 GPU 调度方案，该方案依赖于 Kubernetes 的现有的插件工作机制：

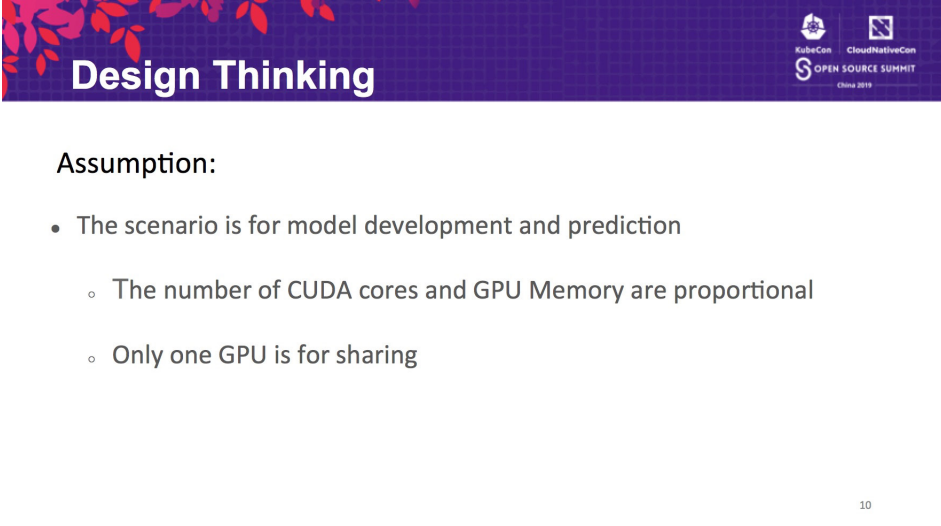
- Extended Resource 定义
- Scheduler Extender 机制
- Device Plugin 机制
- Kubectl 的扩展机制

这个 GPU 共享调度扩展的好处是：利用 Kubernetes 的扩展和插件机制实现，对于 API Server，Scheduler，Controller Manager 以及 Kubelet 等核心组件没有侵入性。这就方便了使用者可以在不同 Kubernetes 版本上应用这个方案，无需 rebase 代码和重新构建 Kubernetes 二进制包。

而且这个方案已经贡献到开源社区，欢迎大家使用和贡献。项目链接：<https://github.com/AliyunContainerService/gpushare-scheduler-extender>

## 设计思考

### 前提假设



The slide features a dark blue header with a red leaf pattern on the left. The text 'Design Thinking' is in white. On the right, there are logos for KubeCon, CloudNativeCon, and Open Source Summit China 2019. The main content area is white with a list of assumptions. The footer shows the number 10.

**Design Thinking**


**Assumption:**

- The scenario is for model development and prediction
  - The number of CUDA cores and GPU Memory are proportional
  - Only one GPU is for sharing

10

1. 依旧沿用 Kubernetes Extended Resource 定义，但是衡量维度最小单位从 1 个 GPU 卡变为 GPU 显存的 GiB。如果所节点使用的 GPU 为单卡 8GiB 显存，它对应的资源就是 8
2. 由于用户对于共享 GPU 的诉求在于模型开发和模型预测场景，在此场景下，用户申请的 GPU 资源上限不会超过一张卡，也就是申请的资源上限为单卡

## 设计原则



# Design Thinking

Principal:

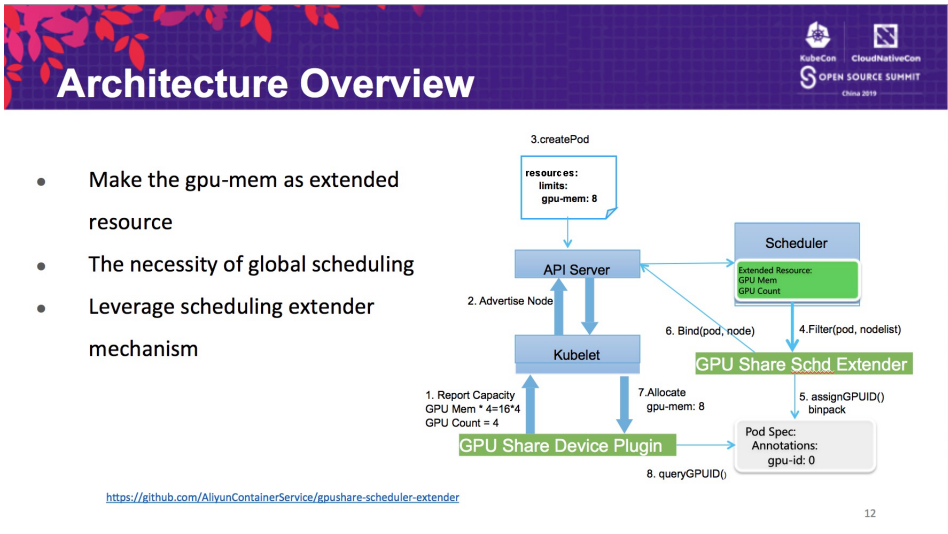
- Easy for user to understand
- Only for scheduling, not for isolation
- Don't change any Kubernetes core code

11

3. 明确问题简化设计，第一步只负责调度和部署，后续再实现运行时显存管控。有很多的客户明确的诉求是首先可以支持多 AI 应用可以调度到同一个 GPU 上，他们可以接受从应用级别控制显存的大小，利用类似 `gpu_options.per_process_gpu_memory_fraction` 控制应用的显存使用量。那我们要解决的问题就先简化到以显存为调度标尺，并且把显存使用的大小以参数的方式传递给容器内部。
4. 不做侵入式修改  
本设计中不会修改 Kubernetes 核心的 Extended Resource 的设计，Scheduler 的实现，Device Plugin 的机制以及 Kubelet 的相关设计。重用 Extended Resource 描述共享资源的申请 API。这样的好处在于提供一个可以移植的方案，用户可以在原生 Kubernetes 上使用这个方案。
5. 按显存和按卡调度的方式可以在集群内并存，但是同一个节点内是互斥的，不支持二者并存；要么是按卡数目，要么是按显存分配。

## 设计方案

而我们的工作首先是定义了两个新的 Extended Resource：第一个是 `gpu-mem`，对应的是 GPU 显存；第二个是 `gpu-count`，对应的是 GPU 卡数。结合这两个资源，提供支持共享 GPU 的工作机制。下面是基本的架构图：



### 核心功能模块：

- **GPU Share Scheduler Extender**: 利用 Kubernetes 的调度器扩展机制，负责在全局调度器 Filter 和 Bind 的时候判断节点上单个 GPU 卡是否能够提供足够的 GPU Mem，并且在 Bind 的时刻将 GPU 的分配结果通过 annotation 记录到 Pod Spec 以供后续 Filter 检查分配结果。
- **GPU Share Device Plugin**: 利用 Device Plugin 机制，在节点上被 Kubelet 调用负责 GPU 卡的分配，依赖 scheduler Extender 分配结果执行。

### 具体流程：

#### 1. 资源上报

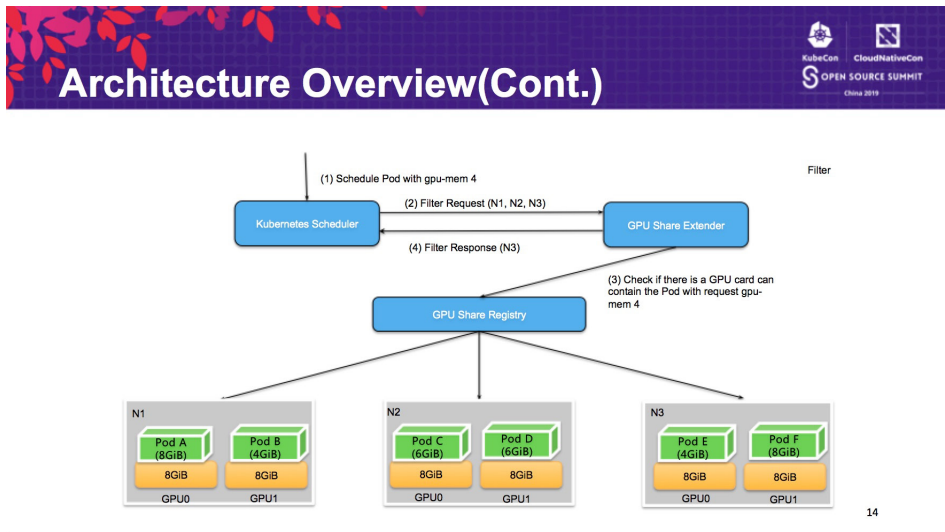
GPU Share Device Plugin 利用 `nvidia-smi` 库查询到 GPU 卡的数量和每张 GPU

卡的显存，通过 `ListAndWatch()` 将节点的 GPU 总显存(数量\_显存)作为另外 Extended Resource 汇报给 Kubelet; Kubelet 进一步汇报给 Kubernetes API Server。举例说明，如果节点含有两块 GPU 卡，并且每块卡包含 8GiB，从用户的角度来看：该节点的 GPU 资源为  $8 \times 2 = 16$ ; 同时也会将节点上的 GPU 卡数量 2 作为另外一个 Extended Resource 上报。

## 2. 扩展调度

GPU Share Scheduler Extender 可以在分配 gpu-mem 给 Pod 的同时将分配信息以 annotation 的形式保留在 Pod spec 中，并且在过滤时刻根据此信息判断每张卡是否包含足够可用的 gpu-mem 分配。

2.1 Kubernetes 默认调度器在进行完所有过滤(filter)行为后会通过 http 方式调用 GPU Share Scheduler Extender 的 filter 方法，这是由于默认调度器计算 Extended Resource 时，只能判断资源总量是否有满足需求的空闲资源，无法具体判断单张卡上是否满足需求；所以需要由 GPU Share Scheduler Extender 检查单张卡上是否含有可用资源。



以上图为例，在由 3 个包含两块 GPU 卡的节点组成的 Kubernetes 集群中，当

用户申请 `gpu-mem=4` 时，默认调度器会扫描所有节点，发现 N1 所剩的资源为  $(8 * 2 - 8 - 4 = 4)$  不满足资源需求，N1 节点被过滤掉。而 N2 和 N3 节点所剩资源都为 4GiB，从整体调度的角度看，都符合默认调度器的条件；此时默认调度器会委托 GPU Share Scheduler Extender 进行二次过滤，在二次过滤中，GPU Share Scheduler Extender 需要判断单张卡是否满足调度需求，在查看 N2 节点时发现该节点虽然有 4GiB 可用资源，但是落到每张卡上看，GPU0 和分别 GPU1 只有 4GiB 的可用资源，无法满足单卡 4GiB 的诉求。而 N3 节点虽然也是总共有 4GiB 可用资源，但是这些可用资源都属于 GPU0，满足单卡可调度的需求。由此，通过 GPU Share Scheduler Extender 的筛选就可以实现精准的条件筛选。

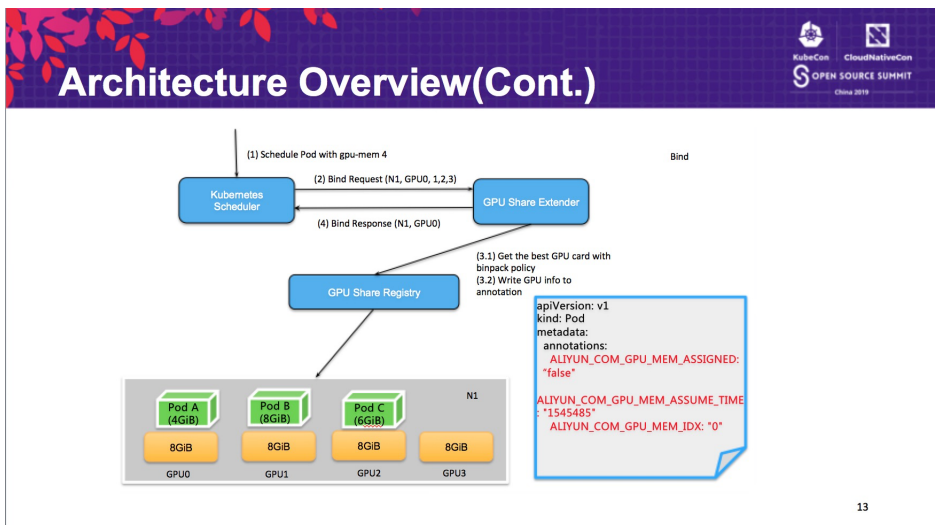
2.2 当调度器找到满足条件的节点，就会委托 GPU Share Scheduler Extender 的 `bind` 方法进行节点和 Pod 的绑定，这里 Extender 需要做的是两件事情

- 以 `binpack` 的规则找到节点中最优选择的 GPU 卡 id，此处的最优含义是对于同一个节点不同的 GPU 卡，以 `binpack` 的原则作为判断条件，优先选择空闲资源满足条件但同时又是所剩资源最少的 GPU 卡，并且将其作为 `ALIYUN_COM_GPU_MEM_IDX` 保存到 Pod 的 `anotation` 中；同时也保存该 Pod 申请的 GPU Memory 作为 `ALIYUN_COM_GPU_MEM_POD` 和 `ALIYUN_COM_GPU_MEM_ASSUME_TIME` 保存至 Pod 的 `annotation` 中，并且在此时进行 Pod 和所选节点的绑定。

注意：这时还会保存 `ALIYUN_COM_GPU_MEM_ASSIGNED` 的 Pod `annotation`，它被初始化为“false”。它表示该 Pod 在调度时刻被指定到了某块 GPU 卡，但是并没有真正在节点上创建该 Pod。`ALIYUN_COM_GPU_MEM_ASSUME_TIME` 代表了指定时间。

如果此时发现分配节点上没有 GPU 资源符合条件，此时不进行绑定，直接不报错退出，默认调度器会在 `assume` 超时后重新调度。

- 调用 Kubernetes API 执行节点和 Pod 的绑定



以上图为例，当 GPU Share Scheduler Extender 要把 `gpu-mem: 4` 的 Pod 和经过筛选出来的节点 N1 绑定，首先会比较不同 GPU 的可用资源，分别为 GPU0(4),GPU1(0),GPU2(2),GPU3(8)，其中 GPU2 所剩资源不满足需求，被舍弃掉；而另外三个满足条件的 GPU 中，GPU0 恰恰是符合空闲资源满足条件但同时又是所剩资源最少的 GPU 卡，因此 GPU0 被选出。

### 3. 节点上运行

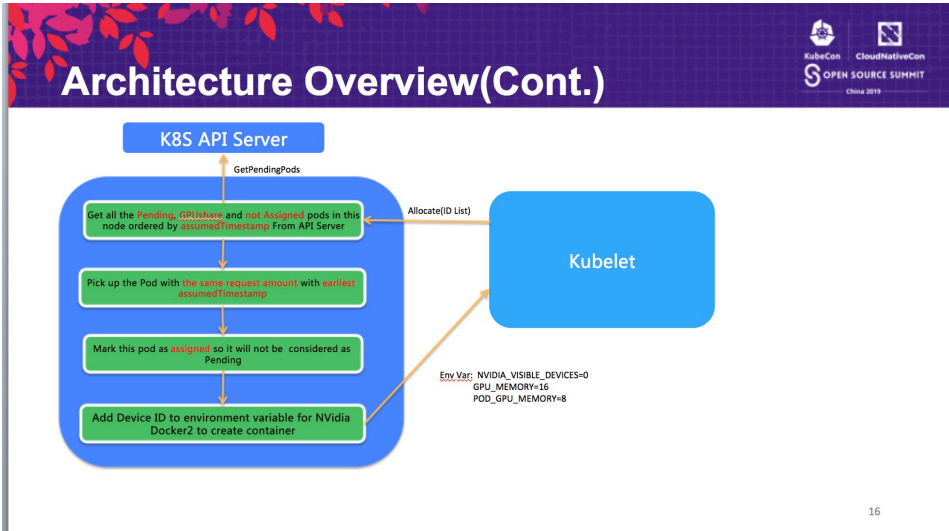
当 Pod 和节点绑定的事件被 Kubelet 接收到后，Kubelet 就会在节点上创建真正的 Pod 实体，在这个过程中，Kubelet 会调用 GPU Share Device Plugin 的 `Allocate` 方法，`Allocate` 方法的参数是 Pod 申请的 `gpu-mem`。而在 `Allocate` 方法中，会根据 GPU Share Scheduler Extender 的调度决策运行对应的 Pod

3.1 会列出该节点中所有状态为 Pending 并且 `ALIYUN_COM_GPU_MEM_ASSIGNED` 为 `false` 的 GPU Share Pod

3.2 选择出其中 Pod Annotation 的 `ALIYUN_COM_GPU_MEM_POD` 的数量与 `Allocate` 申请数量一致的 Pod。如果有多个符合这种条件的 Pod，就会选择其中 `ALIYUN_COM_GPU_MEM_ASSUME_TIME` 最早的 Pod。

3.3 将该 Pod 的 annotation `ALIYUN_COM_GPU_MEM_ASSIGNED` 设置为 `true`,

并且将 Pod annotation 中的 GPU 信息转化为环境变量返回给 Kubelet 用以真正的创建 Pod。



## 操作步骤

### 环境搭建

**Deploy GPU Sharing Capabilities in Kubernetes**

1. Install with Helm

```
# git clone https://github.com/AliyunContainerService/gpushare-scheduler-extender.git
# cd gpushare-scheduler-extender/deployer/chart
# helm install --name gpushare --namespace kube-system --set kubeVersion=1.12.6 --set masterCount=3
gpushare-installer
```

2. Add node labels for GPU sharing

```
# kubectl label node <target_node> gpushare=true
```

3. Download and install the kubectl extension

```
# cd /usr/bin/
# wget https://github.com/AliyunContainerService/gpushare-device-plugin/releases/download/v0.3.0/
kubectl-inspect-gpushare
# chmod u+x /usr/bin/kubectl-inspect-gpushare
```

17



- 用 Helm 安装 gpushare 需要依赖的基础服务, 包括 `gpushare scheduler extender` 和 `gpushare device plugin`

```
# git clone https://github.com/AliyunContainerService/gpushare-scheduler-extender.git
# cd gpushare-scheduler-extender/deployer/chart
# helm install --name gpushare --namespace kube-system --set kubeVersion=1.12.6
--set masterCount=3 gpushare-installer
```

- 为需要共享 GPU 的节点添加节点标签

```
# kubectl label node <target_node> gpushare=true
```

- 安装 kubectl 扩展

```
# cd /usr/bin/
# wget https://github.com/AliyunContainerService/gpushare-device-plugin/
releases/download/v0.3.0/kubectl-inspect-gpushare
# chmod u+x /usr/bin/kubectl-inspect-gpushare
```

## 使用步骤

# Use GPU Sharing in Kubernetes

OPEN SOURCE SUMMIT  
China 2019

1. Query the allocation status of the shared GPU
 

```
# kubectl inspect gpushare
NAME          IPADDRESS      GPU0(Allocated/Total) GPU Memory(GiB)
cn-shanghai.i-uf61h64dz1tmlob9hmtb 192.168.0.71 0/15      0/15
cn-shanghai.i-uf61h64dz1tmlob9hmtc 192.168.0.70 0/15      0/15
```

-----

Allocated/Total GPU Memory In Cluster:  
0/30 (0%)
2. Add node labels for GPU sharing
 

```
# kubectl apply -f binpack.yaml
```

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: binpack-1
  labels:
    app: binpack-1
spec:
  replicas: 3
  serviceName: "binpack-1"
  podManagementPolicy: "Parallel"
  selector: # define how the deployment finds the pods it manages
    matchLabels:
      app: binpack-1
  template: # define the pods specifications
    metadata:
      labels:
        app: binpack-1
    spec:
      containers:
        - name: binpack-1
          image: cheyong/gpu-player:v2
          resources:
            limits:
              # GiB
              aliyun.com/gpu-mem: 3
```

18

### 1. 查看 GPU 共享调度的状态

```
# kubectl inspect gpushare
NAME                                IPADDRESS      GPU0(Allocated/Total)  GPU Memory(GiB)
cn-shanghai.192.168.0.71           192.168.0.71   6/15                   6/15
cn-shanghai.192.168.0.70           192.168.0.70   3/15                   3/15
-----
Allocated/Total GPU Memory In Cluster:
9/30 (30%)
```

### 2. 创建一个使用 `aliyun.com/gpu-mem` 的应用

```
apiVersion: apps/v1beta1
kind: StatefulSet
metadata:
  name: binpack-1
  labels:
    app: binpack-1
spec:
  replicas: 3
  serviceName: "binpack-1"
  podManagementPolicy: "Parallel"
  selector: # define how the deployment finds the pods it manages
    matchLabels:
      app: binpack-1
  template: # define the pods specifications
    metadata:
      labels:
        app: binpack-1
    spec:
      containers:
        - name: binpack-1
          image: cheyang/gpu-player:v2
          resources:
            limits:
              # GiB
              aliyun.com/gpu-mem: 3
```

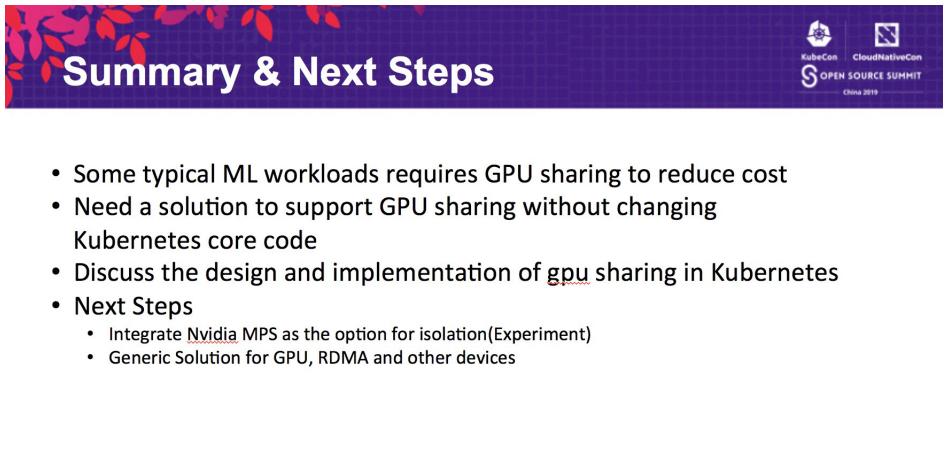
### 3. 调度框架会传递环境变量到运行容器

```
# The total amount of GPU memory on the current device (GiB)
ALIYUN_COM_GPU_MEM_DEV=15
# The GPU Memory of the container (GiB)
ALIYUN_COM_GPU_MEM_CONTAINER=3
```

4. 用户可以在应用层限制 GPU 显存，比如通过 TensorFlow 的 API 可以达到限制的效果

```
fraction = round(3 / 15 , 1 )
config = tf.ConfigProto()
config.gpu_options.per_process_gpu_memory_fraction = fraction
sess = tf.Session(config=config)
# Runs the op.
while True:
    sess.run(c)
```

## 总结和展望



The slide features a dark blue background with a red leaf pattern on the left. The title 'Summary & Next Steps' is in white. Logos for KubeCon, CloudNativeCon, and Open Source Summit China 2019 are in the top right.

- Some typical ML workloads requires GPU sharing to reduce cost
- Need a solution to support GPU sharing without changing Kubernetes core code
- Discuss the design and implementation of gpu sharing in Kubernetes
- Next Steps
  - Integrate Nvidia MPS as the option for isolation(Experiment)
  - Generic Solution for GPU, RDMA and other devices

我们会在 Device Plugin 中提供 Nvidia MPS 的可选支持；支持该方案可以在由 kubeadm 初始化的 Kubernetes 集群自动化部署；Scheduler Extender 的高可用性；以及为 GPU, RDMA 和弹性网卡提供通用方案。

## 相关项目

GPU 共享调度器: [gpushare-scheduler-extender](#)

GPU 共享插件: [gpushare-device-plugin](#)

## 容器运行时管理引擎 Containerd

阿里云容器平台高级研发工程师 傅伟

Software Engineer from Google Kubernetes Team 刘澜涛

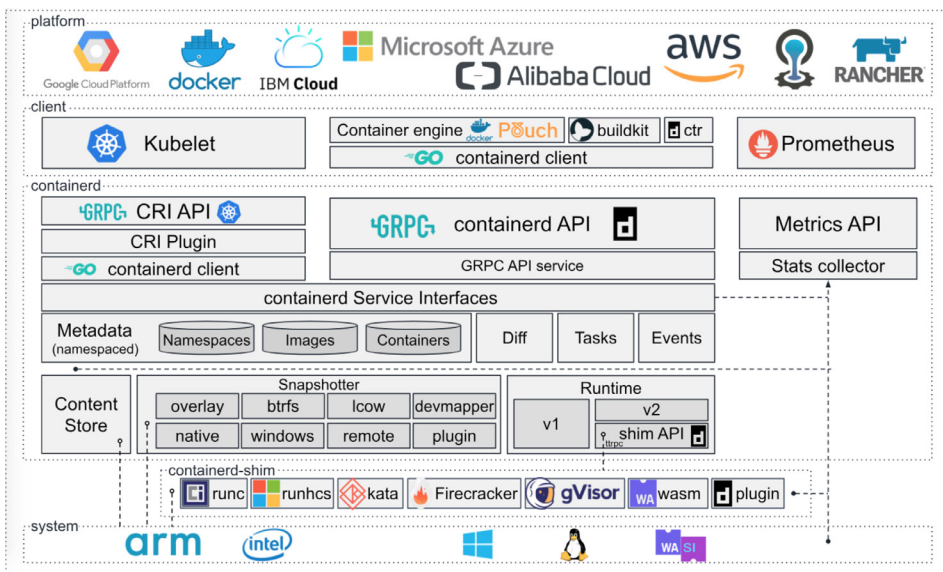
### No.5 project to graduate from CNCF

containerd 项目起源于 docker 公司。在项目之初，containerd 仅负责管理容器的生命周期。由于 Kubernetes Container Runtime Interface 理念的推行，containerd 项目改变其策略来适应生态变化，提出了「被集成」的理念，并在 2017 年初以 Incubating 项目的身份加入 CNCF。除了 moby 项目而外，社区还有 cri-containerd，buildkit 以及 PouchContainer 等开源项目也都集成了 containerd。在很多公司的贡献和支持下，containerd 变得越来越成熟和稳定。也就在今年年初，containerd 以第五名的身份从 CNCF 中毕业。



### Be designed to be embedded into a larger system

containerd 是一个 production-ready 的容器运行时管理引擎，它全面兼容 Open Containers Initiative 协议标准。在 containerd 架构设计里，把容器配置、镜像元数据、镜像原始数据以及镜像本地存储等看成是「资源」，并围绕着资源来构建服务能力。



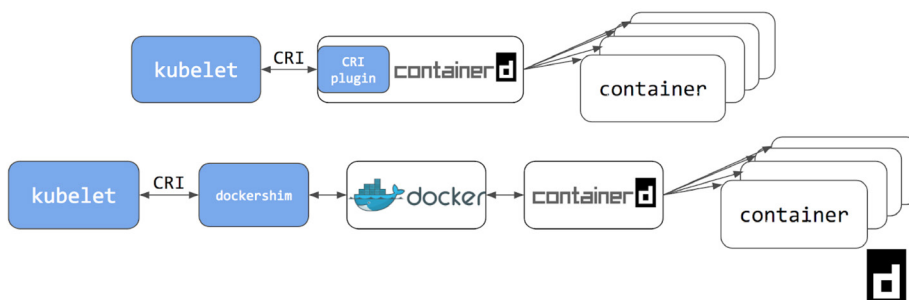
containerd 向开发者开放了 gRPC 服务 API 接口：

- Content Store: 它用来管理镜像原始数据，基本上是 Tarball File ；
- Snapshotter Service: 它用来管理不同文件系统下的镜像本地存储内容，比如 overlays 和 devicemapper ；
- Image Service: 它用来管理镜像元数据与镜像数据之间的映射，方便查询和管理镜像数据；
- Container Service: 它用来管理容器配置信息；
- Metrics API: 它用来和 Prometheus 集成，监控 containerd 以及运行容器的健康状态。

开发者可以根据这些 gRPC API 接口来访问和操作「资源」。为了提升它的可用性，containerd 还引入了 namespace 的概念，它将允许不同用户操作同一个 containerd 服务，实现「多租户」概念。当然它还原生支持 Container Runtime Interface，可以直接同 Kubernetes 集成。

## CRI Plugin

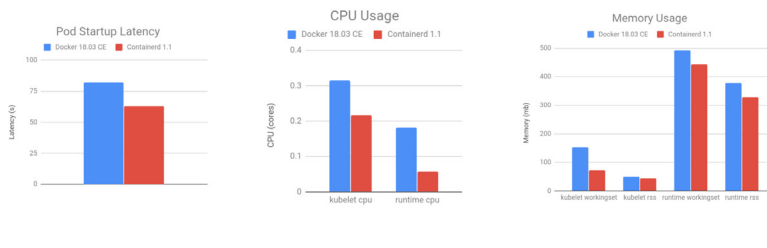
**cri plugin:** A containerd plugin implementation of CRI.



## Performance

Dockershim (Docker CE 18.03) vs. CRI Plugin (Containerd 1.1):

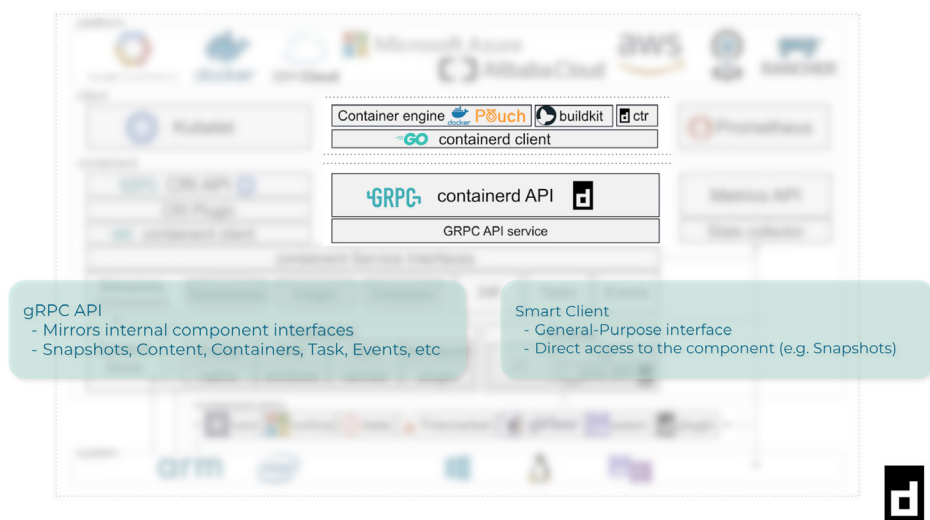
- 105 pods batch startup benchmark
- 105 pods management overhead benchmark.



Container Runtime Interface 功能作为 containerd 默认模块，从 containerd.io 上下载 v1.1 版本以后的 containerd 都会默认支持 CRI，可直接对接 kubelet。相对于 docker-shim，containerd 原生支持 CRI 接口，目前也是 kubernetes 社区所推荐的方式。我们可以看到上图，containerd 性能和架构上都很大程度上优化。

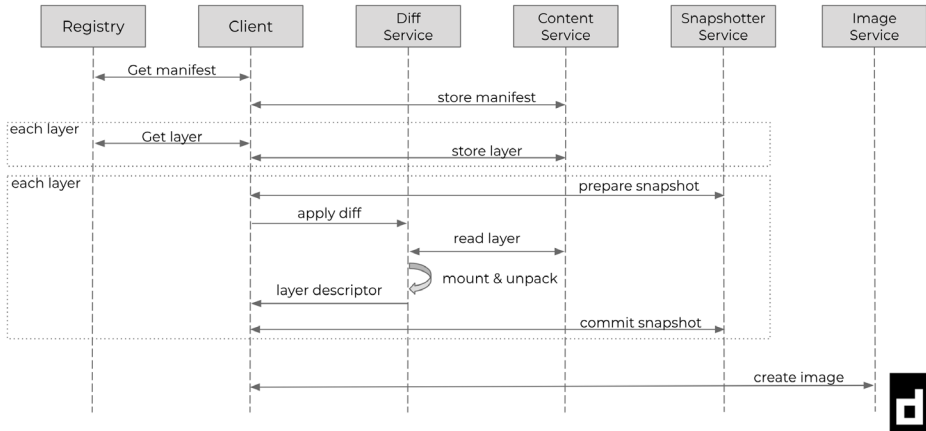
## Smart Client Model –「搭积木」

你会发现 containerd 并不会提供 client-facing 的功能，比如镜像构建。containerd 提供的 gRPC 服务接口，更多是映射内部模块的接口，所以开发者可以很方便地组装「资源」服务来完成各式各样的能力。



为了方便开发者做集成工作，containerd 提供了 Smart Client，这个客户端包含了常用的「组装」功能，比如下镜像和上传镜像等等。

## Pull Image



Smart Client 这种设计理念可以允许开发者在客户端做各式各样的定制化功能，无需改变后端逻辑就能完成集成功能。

### 「插件」化组件

除了客户端可定制化之外，根据特殊场景需求，开发者也需要在 containerd 服务端做改造。针对不同的镜像存储场景，开发者需要 containerd 能支持 qcow2 文件系统；再者，当下有全新的容器运行时，比如 Katacontainer，gVisor 等等，containerd 需要做到通过配置的方式来做集成。因此 containerd 将内部的模块都进行插件化。

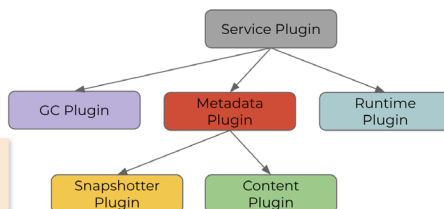
针对目前现有的集成方案，Snapshotter 和 Runtime 插件需要定制化的频率比较高。



## Plugin Registration

- loose coupling and clear boundaries
- dependency Graph

```
plugin.Register(&plugin.Registration{
    Type: plugin.MetadataPlugin,
    ID: "bolt",
    Requires: []plugin.Type{
        plugin.ContentPlugin,
        plugin.SnapshotPlugin,
    },
    Config: &srvconfig.BoltConfig{
        ContentSharingPolicy: srvconfig.SharingPolicyShared,
    },
    InitFn: func(ic *plugin.InitContext) (interface{}, error) {
    },
})
```



### Snapshotter 代理服务

开发者可以在外部编写属于自己 Snapshotter，然后把自定义的 Snapshotter 访问方式添加到 containerd 的启动配置里即可。containerd 在启动时会创建 Proxy Snapshotter 服务，它会将请求转发到自定义的 Snapshotter。

## Remote snapshotter service

- Configure with **proxy\_plugins**
- Build as an external plugin

```
[proxy_plugins]
[proxy_plugins.customsnapshot]
type = "snapshot"
address = "/var/run/mysnapshotter.sock"
```

```
package main

import(
    "net"
    "log"

    "github.com/containerd/containerd/api/services/snapshots/v1"
    "github.com/containerd/containerd/contrib/snapshotservice"
)

func main() {
    rpc := grpc.NewServer()
    sn := CustomSnapshotter()
    service := snapshotservice.FromSnapshotter(sn)
    snapshots.RegisterSnapshotsServer(rpc, service)

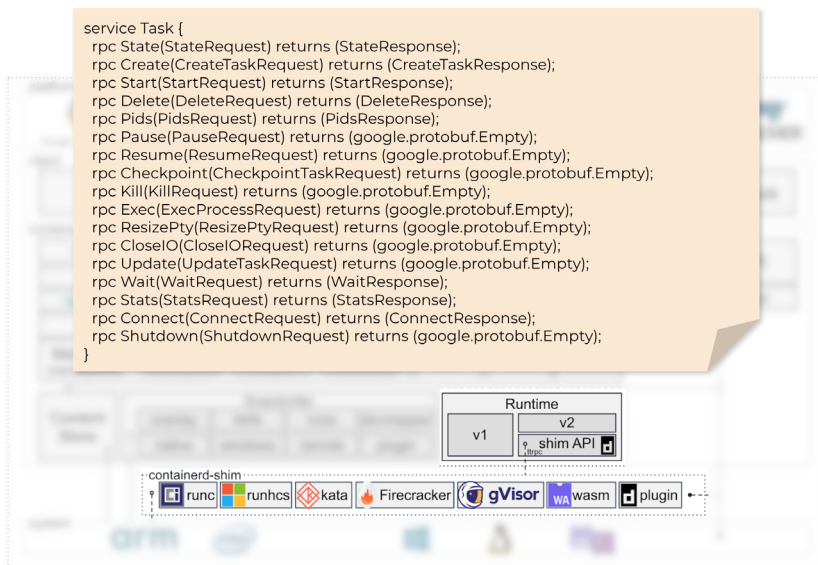
    // Listen and serve
    l, err := net.Listen("unix", "/var/run/mysnapshotter.sock")
    if err != nil {
        log.Fatalf("error: %v\n", err)
    }

    if err := rpc.Serve(l); err != nil {
        log.Fatalf("error: %v\n", err)
    }
}
```



## VM-Like container Runtime 集成

早期 containerd 同其他非 runC 运行时集成时，通过脚本封装的形式来将其他运行的命令行转化成 runC 类型。这样对集成方不友好，而且也加大了配置的难度。因此 containerd 提出了 [ Shim V2 API ]，开发者仅需要在运行时上实现这些 API 即可，containerd 会通过命令系统来寻找该运行时服务二进制并启动它。



目前 Windows 容器、KataContainer、Firecracker、gVisor 等流行的容器运行时都有实现 Shim V2 API，可以直接对接 containerd。可以说 Shim V2 API 已经成为一种标准。

## containerd v1.3

最后，containerd v1.3 版本即将发布，它将会带来更好的插件生态，帮助开发者更好地集成 containerd。

更多信息，可以查看 containerd 项目：<https://github.com/containerd/containerd>

# 基于 P2P 原理的高可用高性能大规模镜像分发系统: Dragonfly

阿里云应用运维平台技术专家 胡作政 (正希)

本文主要分为 4 个部分，首先介绍 Dragonfly (下文称为“蜻蜓”) 的一些历史背景以及过往取得的一些里程碑，然后介绍蜻蜓目前的现状及主要架构原理，接下来介绍蜻蜓的一些使用案例，最后简单介绍了蜻蜓后续的一些规划。

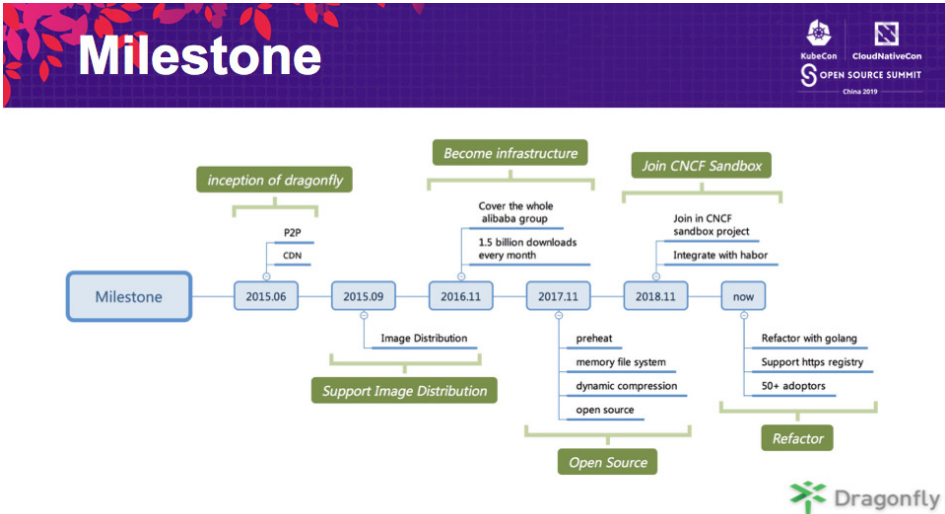
## History



- P2P based, high reliable, image distribution system at large scale
- Bottleneck in pulling images in DCs of Alibaba Group

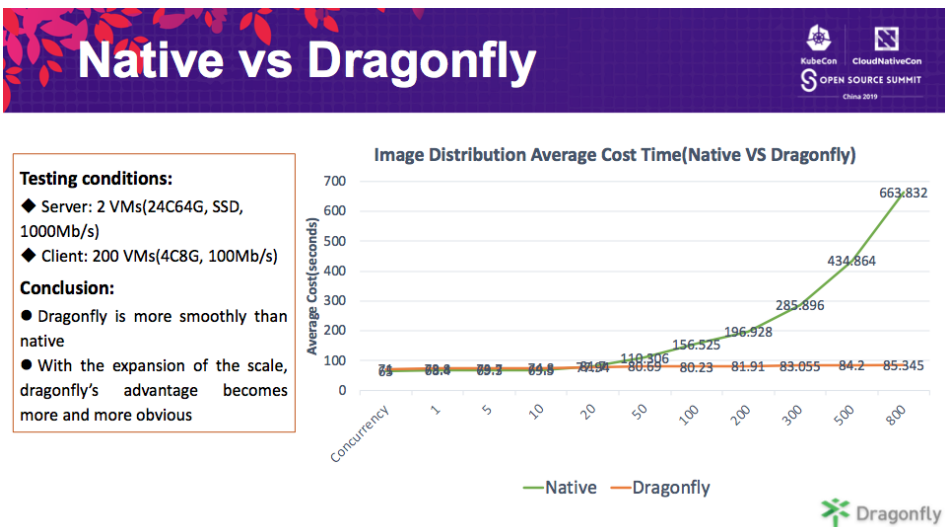


蜻蜓是一款基于 P2P 原理的高可用高性能大规模镜像分发系统，主要用于解决镜像分发的效率以及成本等问题；产生的背景是伴随着阿里巴巴集团业务规模的不断扩大，应用部署的规模也越来越大，从而导致文件下载规模急剧膨胀，通过直接 Wget 下载文件的方式越来越慢，严重影响了应用的部署效率，甚至因为文件下载问题发生了好几次重大故障，可以预见再过不久文件服务器将会被打爆，同时扩容也无济于事，因此为了彻底解决该问题，我们决定开发一套专门的文件分发系统，蜻蜓就这样诞生了！



这是蜻蜓在过往取得的一些主要里程碑，其中 2015 年 6 月份正式上线，2016 年 11 月份成为阿里巴巴集团的基础设施，目前蜻蜓每月承载了 30 亿次以上的文件下载任务，同时也为每年的双十一活动提供数据分发服务。

## Status



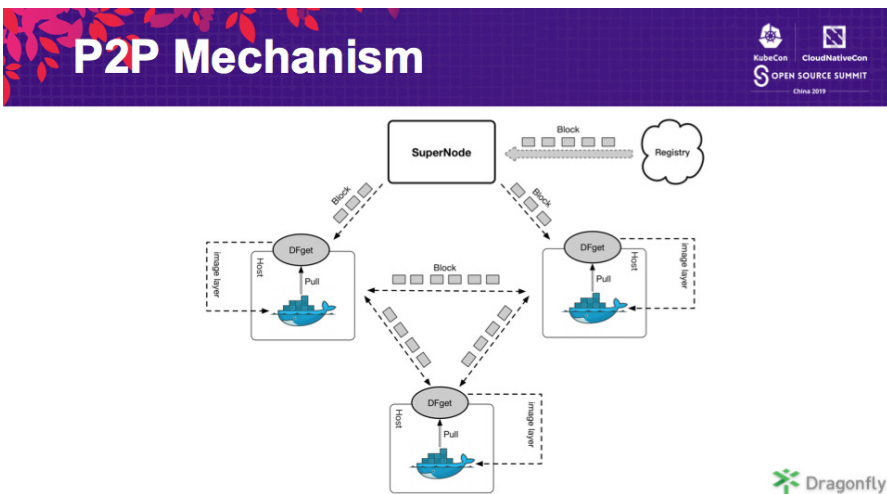
这是使用蜻蜓和使用原生的镜像下载方式的数据对比图，从中可以很明显的看出，使用蜻蜓下载镜像的平均耗时非常平滑，同时随着下载规模的扩大，蜻蜓的优势对比原生方式愈加明显。

## Community Situation

Stars: 3700+  
Adoptors: 50+  
Maintainers: 6  
    Outside: 1 ebay, 1 meitu  
    Internal: 4 alibaba group  
Contributors: 36  
Discussion Group:  
    DingTalk: 23304666  
    Gitter: please refer to Github

Dragonfly

这是蜻蜓在社区目前的现状，在镜像分发领域，蜻蜓的市场是第一位 (Stars 和 Adoptors 等)，蜻蜓广泛应用在各个行业，包括电商类公司、云计算公司、金融科技公司以及本地生活和直播公司等等。



针对上图有几个概念需要先解释：

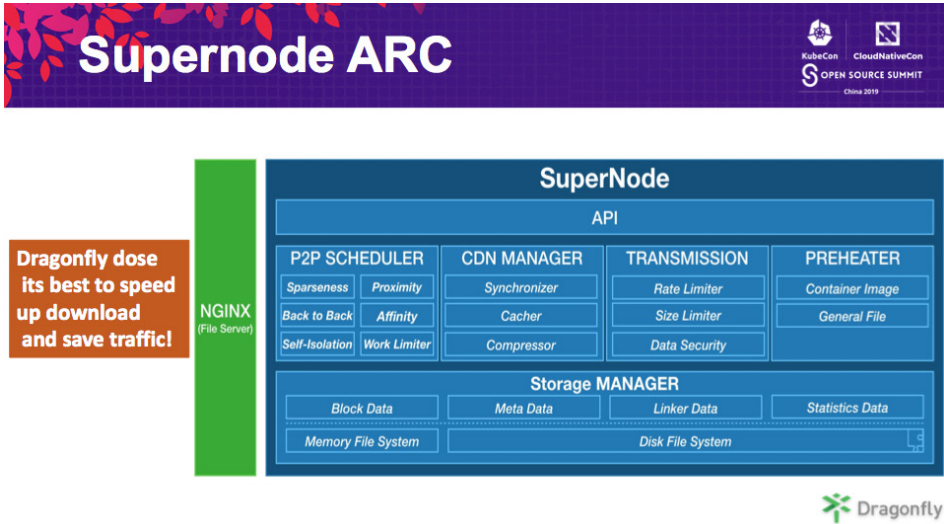
- Registry：容器镜像的存储仓库，每个镜像由多个镜像层组成，而每个镜像层又表现为一个普通文件
- Block：当通过蜻蜓下载某层镜像文件时，蜻蜓的 SuperNode 会把整个文件拆分成一个个的块，SuperNode 中的分块称为种子块，种子块由若干初始客户端下载并迅速在所有客户端之间传播，其中分块大小通过动态计算而来
- SuperNode：蜻蜓的服务端，它主要负责种子块的生命周期管理以及构造 P2P 网络并调度客户端互传指定分块
- DFget：蜻蜓的客户端，安装在每台主机上，主要负责分块的上传与下载以及与容器 Daemon 的命令交互
- Peer：下载同一个文件的 Host 彼此之间称为 Peer

下载过程大致如下：

1. 首先由 Docker 发起 Pull 镜像命令，该命令会被 DFget 代理截获
2. 然后由 DFget 向 SuperNode 发送调度请求
3. SuperNode 在收到请求后会检查对应的文件是否已经被缓存到本地，如果没有被缓存，则会从 Registry 中下载对应的文件并生成种子块数据（种子块一旦生成就可以立即传播，而并不需要等到 SuperNode 下载完成整个文件后才开始分发），如果已经被缓存，则直接生成分块任务
4. 客户端解析相应的任务并从其他 Peer 或者 SuperNode 中下载分块数据，当某个 Layer 的所有分块下载完成后，一个 Layer 也就下载完毕，此时会传递给容器引擎使用，而当所有的 Layer 下载完成后，整个镜像也就下载完成了

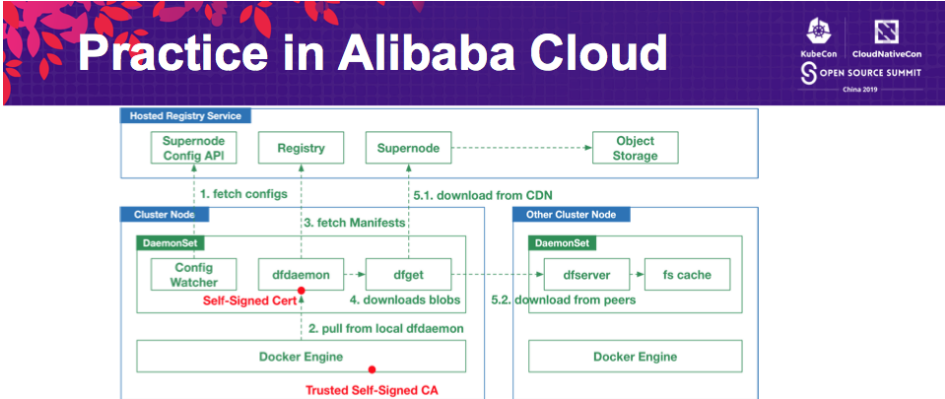
通过上述 P2P 技术，可以彻底解决镜像仓库的带宽瓶颈问题，充分利用各个 Peer 的硬件资源和网络传输能力，达到规模越大传输越快的效果。蜻蜓的系统架构不涉及对容器技术体系的任何改动，完全可以无缝支持容器使其拥有 P2P 镜像分发

能力，以大幅提升文件分发效率！



以上是蜻蜓的核心架构，也是 SuperNode 的架构，一共分为五大模块：P2P Scheduler、CDN Manager、Transmission、Preheat、Storage Manager；其中 P2P Scheduler 负责调度分片任务，确定请求的客户端当前应该去哪些 Peer 下载哪些 Block；CDN Manager 负责种子生成以及缓存管理和动态压缩等等；Transmission 负责控制传输过程中的限速、分块大小以及数据安全等；PreHeat 负责文件和镜像的自动化预热；Storage Manager 负责存储分块种子以及对应的元信息等，同时不仅支持普通的磁盘文件系统，而且还支持内存文件系统，用于解决大文件的分发效率问题

## Adoption

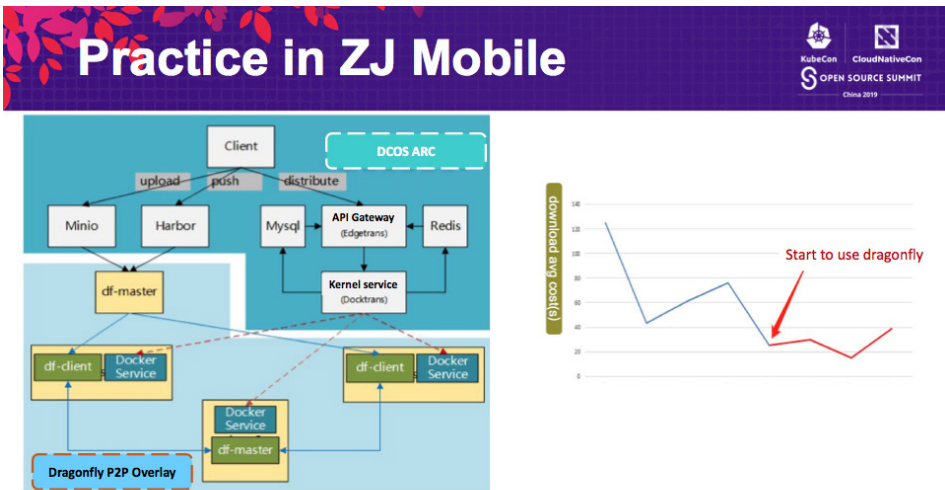


ACR(Alibaba Container Registry Service) provides two registry domains, one for normal cases, the other is for P2P distributed pulls, which always points to localhost.

ACR's P2P image distribution asks users to deploy a DaemonSet in their Kubernetes cluster, and copy a Self-Signed CA to Docker's configuration path. After this, every node will have a pod running as a registry proxy, then Docker can pull new images using the P2P domain, without any interruption or restart of any containers.



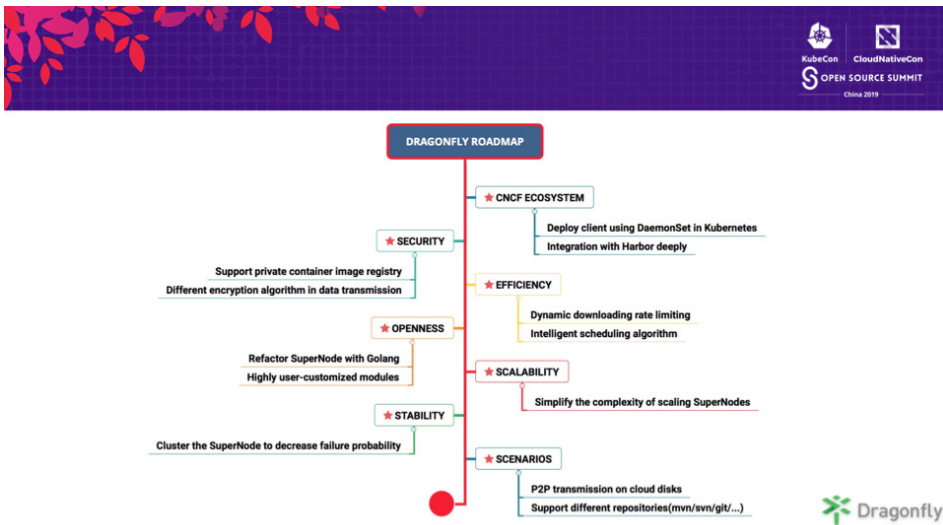
以上是阿里云的镜像仓库服务整合使用蜻蜓的方案。



以上是浙江移动整合使用蜻蜓的方案，从中可以很明显的看出（红色箭头表示使用蜻蜓后的应用部署耗时），通过使用蜻蜓可以很大程度上（3 倍以上）提升应用的部署效率。



## Roadmap



以上是蜻蜓未来的一些规划，主要会从安全性、稳定性、性能以及与 CNCF 生态的融合等方面做更加深入的投入。

## 结束



## 首个为中国开发者量身打造的云原生课程 《CNCF x Alibaba 云原生技术公开课》

由阿里云与 CNCF (Cloud Native Computing Foundation) 共同开发的《CNCF x Alibaba 云原生技术公开课》与 2019 年 4 月正式上线。来自全球“云原生”技术社区的亲历者和领军人物，齐聚“课堂”为每一位中国开发者讲解和剖析关于“云原生”的方方面面，一步步揭示这次云计算变革背后的技术思想和本质。

《CNCF x Alibaba 云原生技术公开课》，也是 CNCF 旗下首个为中国开发者量身打造的云原生课程。这门课程完全免费且无需注册，旨在让广大中国开发者可以近距离聆听世界级技术专家解析云原生技术，让“云原生”技术真正触手可及。



微信或钉钉扫描二维码  
直接开始学习

### 适合人群

- 计算机科学、软件工程等领域的软件工程师和大学生
- 使用 / 尝试使用容器和 Kubernetes 技术的应用程序开发者
- 具有基本服务器端知识、正在探索容器技术的软件开发者和技术管理者
- 希望理解云原生技术栈基本原理的技术管理者和开发者

### 你可以收获什么？

- 完善的知识体系，打造属于自己的云原生技能树
- 理解云原生技术背后的思想与本质
- 与知识体系相辅相成的动手实践
- 一线技术团队云原生技术最佳实践
- CNCF 与阿里云联合颁发课程结业证书

后续，《CNCF x Alibaba 云原生技术公开课》还会推出涵盖 Serverless, ServiceMesh 等更多云原生技术的高品质原创系列课程，敬请期待。



微信扫一扫关注  
阿里巴巴云原生公众号



微信扫一扫关注  
阿里技术公众号



阿里云开发者社区  
开发者一站式平台



钉钉扫描二维码，  
立即加入 K8s 社区大群

