



基于容器云的微服务系统

杨迪

(中国联合网络通信有限公司软件研究院, 北京 100176)

摘要: 现有微服务系统持续交付仅将可执行代码或程序放入运行环境, 缺乏对容器化服务的敏捷开发流程。对于容器化服务的版本测试和配置更新等相关问题也缺乏相应的解决方案。因此, 如何在微服务系统上快速构建容器云并提供相关完整解决方案是急需解决的技术难题。提出了一种基于容器云的微服务系统。以容器为载体灵活快速构建微服务, 为上层应用提供容器云服务。实现了服务容器化持续集成, 支持多个服务的多个版本联调测试, 达到生产不停机、版本无缝切换的效果; 由运维监控平台对这些服务进行统一的监控管理。经过生产环境验证, 系统能够保障多种服务运行不停机, 短时间响应请求, 满足生产的需求。

关键词: 容器云; 微服务; 持续交付; 微服务监控

中图分类号: TP316

文献标识码: A

doi: 10.11959/j.issn.1000-0801.2018168

Microservice system based on container cloud

YANG Di

Software Research Institute of China United Network Co., Ltd., Beijing 100176, China

Abstract: Lack of the procedure of deploying containerized services in microservice systems, the continuous delivery only put executable codes or programs to the production environment. There is no solution for problems of the test version and configuration updates. The microservice system based on container cloud was proposed. Continuous integration for containerized services and the method that supports debugging tests for multi-version of multi-services were designed. Goals of non-stop production and seamless switch between versions were achieved. These services were monitored and managed by an operation and maintenance platform. After production environment verification, the proposed system supports services running without stop and immediate response and satisfies the needs of production.

Key words: container cloud, microservice, continuous delivery, microservice monitoring

1 引言

传统单体服务构建的软件应用内部多个模块紧密耦合, 随着软件系统规模的极具扩张, 结构

复杂度快速增加。任何修改都可能导致软件系统整体运行出现故障, 使得版本升级和快速迭代变得十分困难。而且, 当某个子模块出现性能瓶颈时, 为了缓解负载往往只能按照应用整体重复部



署, 导致其他子模块利用率更低^[1]。

与单体服务相比, 微服务架构具有独立部署、按需扩展、快速迭代、技术栈灵活等诸多优点。微服务是一种简单而轻量的小服务, 职责单一, 代码量少。对这些原子服务进行编排, 可以快速构建一个新的应用, 它们之间通过轻量级的交互机制来通信。另一方面, 容器化技术的出现促进了微服务架构的快速发展。将微服务装入容器进行独立的部署和扩展, 使其具有可移植性, 成为一种必然的趋势^[2]。

美国最大运营商 AT&T 将微服务作为敏捷运营的重要技术之一, 以实现互联网级的速度推进产品, 实现数字化转型。中国移动通信集团浙江有限公司 (以下简称浙江移动) 借助 DC/OS 平台, 以容器化技术解耦了 14 个运营中心, 对客户关系管理系统进行了升级改造^[3]。因此, 基于容器的微服务架构在工业界和学术界得到了广泛关注。

然而, 以容器为载体灵活快速地构建微服务系统, 为上层应用提供容器云服务, 一直是个尚未解决的难题^[4]。目前的微服务系统, 通常直接将微服务打包运行在主机或者虚拟机节点, 这种方式需要在所有主机或虚拟机部署各种运行环境, 而且不能自动在集群中选择运行节点; 在当服务出现过载时, 只能在一台或多台主机上重复启动多个进程, 不能按需进行快速扩展; 运维困难是这种方式的另一个弊病。不同技术栈开发的服务在不同的应用服务器运行, 很难以一个平台一种方法对它们进行统一监控管理。因此, 将微服务容器化是一种势在必行的解决方案^[5-6]。

现有的持续集成流程仅仅将可执行代码或程序放入测试环境, 缺乏容器化的快速部署方案。对于容器化服务的版本测试和配置管理等相关问题也缺乏相应的解决方案。因此, 如何在微服务系统上快速构建容器云并提供相关部署、测试完整解决方案成为急需解决的技术难题^[7]。

针对这些问题, 本文提出了基于容器云的微服务系统, 向不同级别的多种业务应用提供容器化的基础服务, 实现了容器即服务 (container as a service, CaaS)。在系统中, 微服务以容器为载体部署在分布式集群中, 根据业务逻辑重组多个微服务, 构成新的业务应用。新应用仍然以容器化形式部署在集群中, 对更上层的应用提供服务。开发人员只需要提交代码到代码库, 并增加一些必要的配置文件, 系统会自动以容器构建服务, 并一键部署到测试和生产环境, 实现容器化微服务和应用的敏捷开发。利用容器化服务的版本测试方法实现快速迭代。运维人员从统一的监控平台全面检查各个服务的运行状况, 及时处理故障, 保障系统正常运行。

这里选用 Mesos+Marathon 的 DC/OS 架构。Mesos 只分配集群计算资源, 不负责任务调度, 而 Kubernetes (K8S) 设计是面向应用的集群管理工具。基于 Mesos 可以方便地运行不同的分布式大数据处理平台, 如 Spark、Storm、Hadoop、Marathon 等^[4]。以 Mesos 为基础架构有利于高效实时处理运营商内部的大规模数据。Mesos 的商用程度很高, Airbnb、Apple、Uber、Twitter 都在使用。基于 Mesos 构建的主要产品是 DC/OS (datacenter operation system), 而中国联合通信网络有限公司软件研究院已正式加入 DC/OS 社区, 成为主要贡献者。

2 系统架构

系统的总体架构如图 1 所示。每个服务运行在独立的 Docker 中, 由分布式资源管理框架 Mesos^[8]根据集群中主机资源 (CPU、内存和硬盘等) 空闲情况, 自动选择一个合适的节点, 启动容器和其内部服务, 由 Marathon^[9]负责监控并展示容器状态, 实现跨主机的容器管理。服务设计遵循 RESTful (representational state transfer)^[10] 架构, 以 HTTP+REST 协议访问服务。由 F5 地址

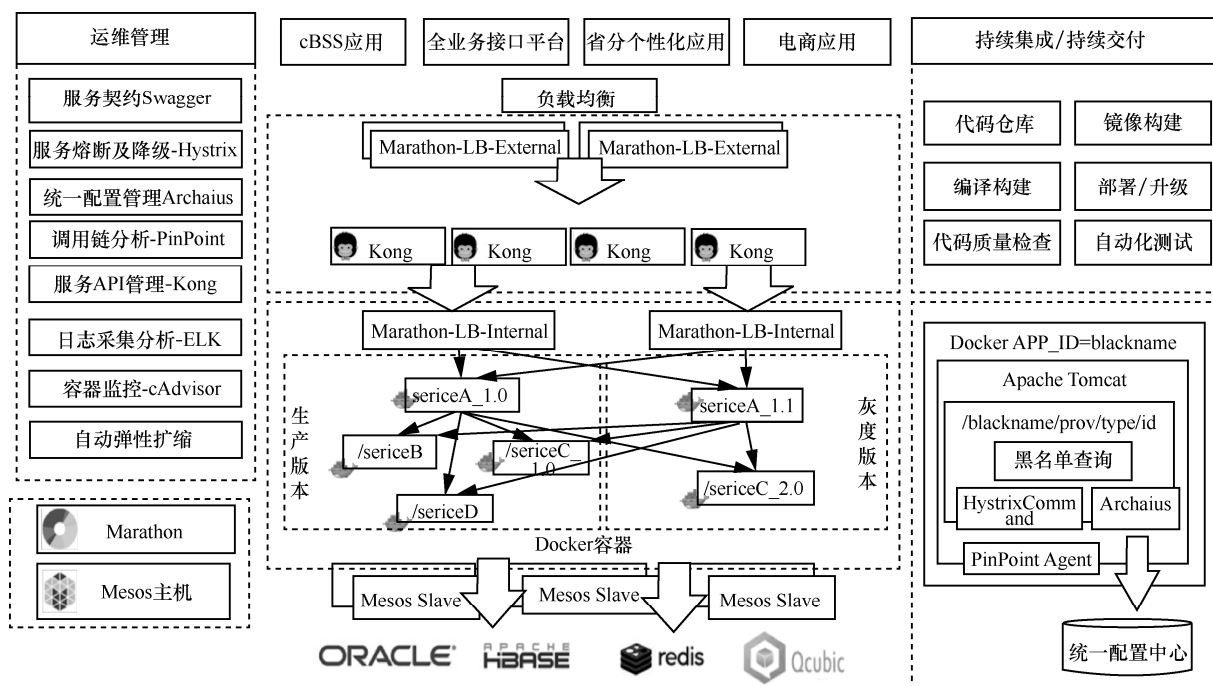


图1 系统框架

作为访问服务的固定 IP 地址，接收访问请求后，经过 Marathon-lb-external 负载将请求转发到 API 网关模块 Kong，经过鉴权，将请求转发到软负载转发器。

Marathon-lb-internal^[11]，由 Marathon-lb-internal 根据预先设定的负载均衡策略及转发规则选择合适的微服务实例响应请求，以 external 和 internal 后缀区分两个 Marathon-lb。当新的服务部署到系统时，服务发现模块 Marathon 会自动将服务容器 IP 地址、端口和容器的配置信息注册到 Marathon-lb 的转发后端 (backend)。当系统中的服务实例 IP 地址和端口更新时，软负载转发器的后端也会随之自动更新。除了 F5 之外，所有组件均以容器方式部署。

Marathon+Mesos 的框架只提供了基本的容器管理功能，展示容器状态，操作容器启停。在这个框架下，开发者每次更新代码都需要手动打包、构建镜像并运行 Docker。这使得部署环节增多，操作更加复杂，不利于版本快速迭代，不符合敏捷开发理念。服务部署完成后，缺乏对调用状态

的监控。在一个服务调用链中，如果有服务调用失败而没有处理，会影响应用正常响应，甚至造成微服务“雪崩”。

基于此，在现有基础架构上，改造一些开源项目，使其能够在微服务容器云的架构中提供持续交付和运维管理等功能。如图 1 所示，持续交付包括代码仓库 Gitlab、编译构建 Jenkins、代码质量检查 Sonar、灰度版本发布和服务契约 Swagger^[12]等。在运维管理方面，要一套完整的监控体系来实时监控容器中服务的调用状态和整个调用链上的服务状态，包括调用链分析 PinPoint^[13]、容器性能监控、日志采集分析 ELK^[14]等。因为每个微服务的部署都会启动多个容器，采用统一配置管理框架，动态更改参数变量而不重启服务。

3 功能模块

开发者提交微服务模块代码到 Gitlab 后，由 CI/CD 的 Jenkins 从 Gitlab 拉取代码并进行编译，Sonar 代码规范检查通过后，依据模板填写部署所

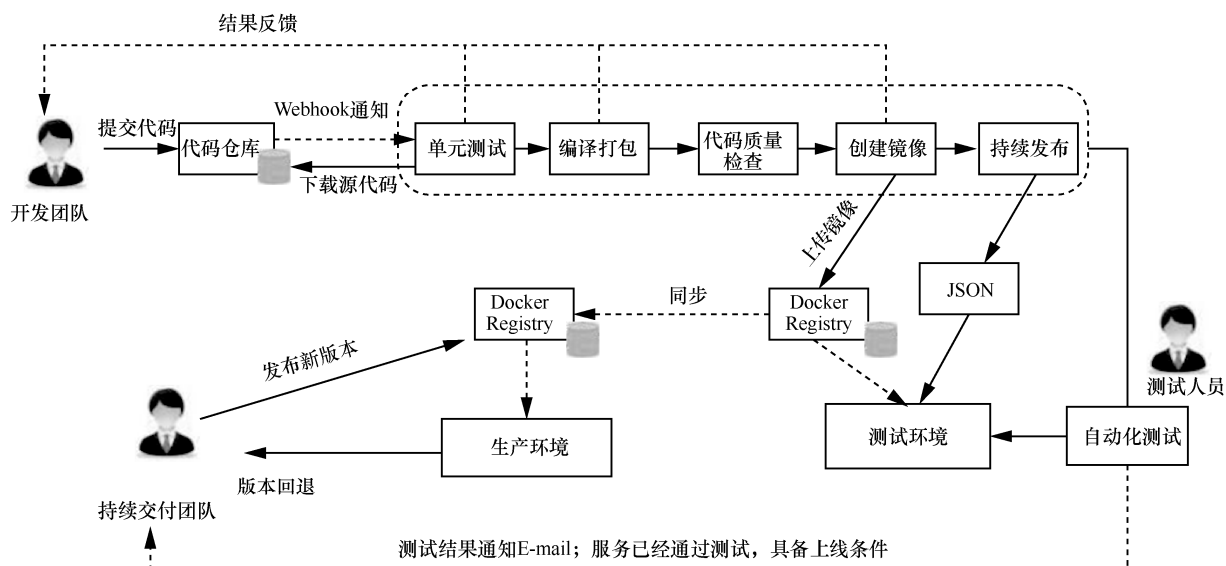


图2 持续集成/持续交付流程

需要的配置参数，将编译文件添加到 Docker 镜像中。由 Marathon 拉取镜像启动容器，完成了一个服务的部署。

3.1 持续集成/持续交付

系统支持在测试环境和生产环境自动实施服务容器化部署，使开发人员更加专注于实现业务逻辑和代码编写，不需要应对异构环境下的部署细节，实现版本快速迭代。如图 2 所示，开发者配置部署所需要的模板文件后，系统将按照模板中配置参数自动生成该服务的 Docker 镜像，将微服务装载到 Docker 中，启动 Docker 和服务。在测试环境下，业务逻辑测试通过后，一键推送到生产环境。服务容器化部署流程，配置参数简单易懂，只需非常必要的配置。具体包含如下步骤。

步骤 1 开发者在工程文件夹 src 下增加 deploy 文件夹，在该文件夹下增加配置文件 deploy.json 和需要在 Docker 中加载的其他文件，配置文件中仅需指定服务名称、版本信息、服务维护者和环境变量等所必需的参数信息，如图 3 所示。将工程提交到代码仓库 Gitlab，由 Sonar 执行代码质量检查，若代码的 bug（漏洞）低于 Sonar 的阈值，编译该工程生成可执行文件或程序。同时，Swagger 根据开发者在代码中配置的访

问路径，生成服务契约，生成调用样例和接口。

```
{
  "description": {
    "service_name": "demo_microservice工程",
    "maintainer_group": "微服务平台",
    "maintainer_name": "张三",
    "maintainer_telephone": "13269625373"
  },
  "env": {
  },
  "labels": {
  }
}
```

图3 配置文件示例

步骤 2 通过运行脚本，将 deploy.json 中配置参数复制到 Dockerfile 模板中，为该服务生成新的 Dockerfile，构建微服务镜像，并上传到测试环境的 DockerRegistry。Mesos 在集群中选择适合的主机节点，拉取镜像后启动 Docker 和其中的微服务，完成服务的容器化部署。

步骤 3 测试通过后，将测试环境服务镜像复制到生产环境的 DockerRegistry，按照步骤 2 中的方法，在生产环境上完成服务的部署。持续部署团队以相同的方法发布新版本，版本更新将在第 3.2 节详细阐述。

3.2 版本更新

系统允许用户灵活配置多个服务的多个版本

联调测试,达到生产不停机、版本无缝切换的效果。只有一个版本的服务可跳过此环节,其他版本均需要对已有服务进行版本管理。本系统允许新旧版本同时部署运行,两种版本容器化服务名称一致,以版本号区分,如新版本服务为 service-1.1,旧版本为 service-1.0。按照设定的规则同时调用新旧版本服务,调用规则在 deploy.json 的 labels 中定义。当容器启动后,Marathon 将新的容器信息推送到 Marathon-lb-internal,它通过解析标签来设置转发规则。当测试通过,将所有调用引入新版本服务,同时停掉旧版本服务。系统支持如下 3 种访问版本规则。

(1) URI 匹配

通过 HTTP 请求的统一资源标识符(uniform resource identifier, URI)设置转发条件,不同的 URI 访问不同的版本。如 URI 为/a/1/2 请求转发到 service-1.1 对应的容器。在服务 service-1.1 的 deploy.json 的 labels 定义路径转发标签 HAPOXY_0_PATH:/a/1/2。在容器启动后,Marathon-lb-internal 自动更新配置文件,将该路径作为转发规则。当接收到的请求 URI 与该转发规则成功匹配后,将请求转发到 service-1.1 对应的服务容器。同理,可将服务 service-1.0 的 URI 设置为其他路径,如/a/2/2 标签设为 HAPOXY_0_PATH:/a/2/2。

(2) Header 匹配

在 HTTP 请求的 Header 和容器标签 HAPOXY_0_Header 中定义 Header 值,作为转发规则。若 HTTP Header 中包含 HAPOXY_0_Header:staffid=1\$departid=1,多个参数以“\$”为分隔符,则转发到 service-1.1,否则转发到 service-1.0。在 deploy.json 中设置转发标签 HAPOXY_0_HEADER:staffid=1\$departid=1,容器启动后,Marathon-lb-internal 自动在配置文件中增加 Header 转发规则。由 Marathon-lb-internal 解析请求 Header 的参数,若匹配成功,将访问

请求转发到版本 service-1.1 的容器。对于未设定 Header 转发规则的 HTTP 请求,默认转发到 service-1.0 的容器。

(3) 多服务多版本访问控制

一个业务应用需要调用多个服务,在做版本访问控制时,通过将同一调用链的服务容器转发规则和 HTTP 请求设置相同的 Header 参数,灵活地配置需要访问的版本。如图 4 所示,业务应用地调用关系为 servA→servB→servC。服务 servB 和 servC 均有测试版本 servB'和 servC',若版本访问的顺序为 servA→servB→servC',访问请求 HTTP Header 和 servB、servC'的容器 labels 都设置 HAPOXY_0_HEADER:staffid=1。在 servA 调用下游服务时,系统取出请求 Header 值,放入调用 servB 和 servC'的 HTTP Header 中。

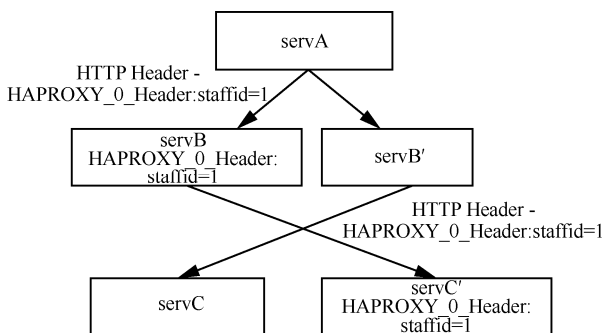


图4 多服务多版本访问

当 servA 接收请求并转发给下游服务 servB 时,Marathon-lb-internal 按照 HAPOXY_0_HEADER:staffid=1 和 URI 规则转发到 servB, servB' 虽然与 servB 的访问 URI 相同,但 servB' 的转发规则不包含 Header,因此,只有 servB 能接收请求, servB' 不会接收请求。同理,下游服务 servC' 接收并响应请求。若访问请求 HTTP Header 没有设置 HAPOXY_0_HEADER,调用顺序为 servA→servB'→servC。

3.3 零分组丢失及配置中心

3.3.1 零分组丢失策略

系统根据业务调用规律自动增加或减少



服务容器实例。例如，在出账期间，出账相关服务调用量是平时的 10 倍，在预先设置的时间自动将实例扩展成设定的实例数，在账期结束后，按照设定将服务实例缩减成设定的实例数。

在扩缩过程中，容易出现丢失请求的问题。原因是改变实例个数后，需要重启软负载转发器重新读取服务实例配置。在重启过程中，Marathon-lb 因无法接受请求而丢弃。这里对 Marathon-lb 进行改进，实现了真正的零停机转发。具体步骤如下。

步骤 1 Marathon-lb 重启之前设置触发点，在重启之前更改触发点的值。

步骤 2 当触发点值改变后，阻塞 Marathon-lb 接收请求。

步骤 3 设置一个队列，解析 HTTP 请求，将源地址为 F5 地址的请求缓存在队列中。

步骤 4 Marathon-lb 重启后，按照顺序释放该队列中的请求。清空缓存队列。

3.3.2 统一配置中心

采用统一配置中心管理容器化服务的运行参数，使服务运行中参数改变，而不需要重启服务。在 Archaius 的基础上进一步封装，为开发者提供可在程序中直接调用的运行分组。

用户通过界面录入服务配置参数，将配置信息存储在数据库，如图 5 所示。配置中心分组自动启动线程定时获取存储在数据库中的配置参数，以 MAP 结构存储在本地的缓存中。开发者调用配置中心分组中的监听函数，监听 MAP 对应配置值的变化情况。当远端配置项发生改变时，更新对应的本地配置项，同时监听函数对配置有关的部分重新操作。

3.4 监控运维

通过增加多种监控组件，实现了对运行中的服务及调用状态进行全面监控并对异常行为发出告警。主要分为对容器状态和服务调用两方面的监控。

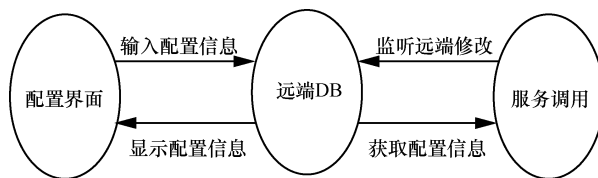


图 5 配置中心流程

3.4.1 服务调用监控

完成一个业务流程往往需要调用多个服务。如果某个服务不可用，将导致整个业务流程被阻塞，最终导致业务崩溃，甚至可能影响整个系统的正常运行，从而造成服务雪崩。为避免雪崩效应，从服务调用和调用监控两方面来预防。

在服务调用环节，主要采用 Hystrix 熔断器机制。在调用远端服务或者数据库时，将请求封装在 HystrixCommand 方法中。在 Callback 方法中，做异常处理，返回远程服务调用异常的错误码。根据设置参数超时时间，若远程服务在超时时间内没有响应，则返回 Callback 中设置错误异常码，保证服务顽健性。

Hystrix 的监控页面如图 6 所示，从左向右依次表示服务的正常、熔断、熔断后半开和恢复正常状态的过程。服务 BlackName 访问数据库 Redis，当多次访问 Redis 失败超过预设值 50%，BlackName 启动熔断状态，暂停对数据库的访问，直接返回错误返回码；当被暂停一段时间（通常是 5 s）之后，进入熔断半开状态，一部分服务访问 Redis，一部分服务直接返回错误码；若请求健康数超过阈值，则恢复正常状态，否则继续熔断。同时，系统引入分布式跟踪工具 PinPoint 来跟踪调用链中有异常的服务。选取的主要原因是代码的零入侵，只需要将 PinPoint 添加到微服务的基础镜像中，当服务被调用时，PinPoint 采用调用链追踪技术，以全局唯一 TraceID，将一个调用链上各个服务关联追踪，以调用树的形式展现出来。如图 7（a）所示，用户调用 ecs_userinfo 服务，

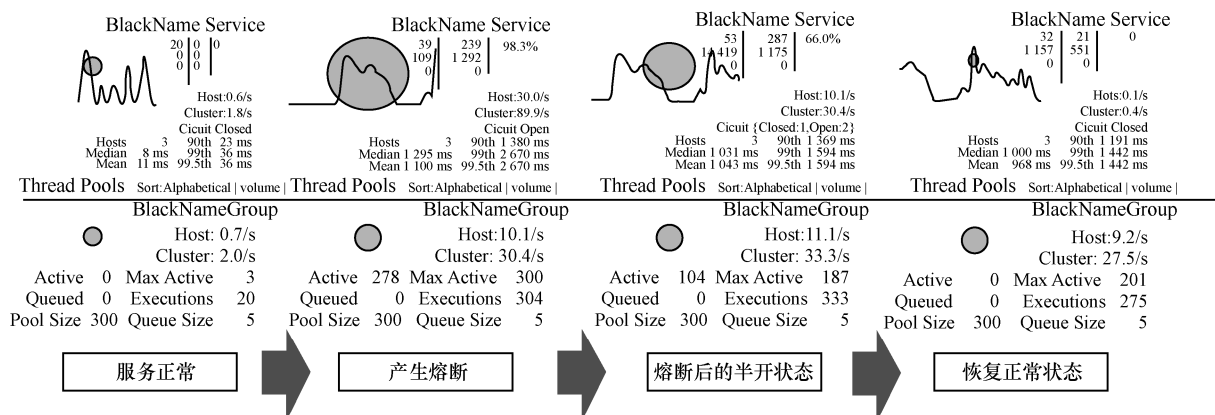
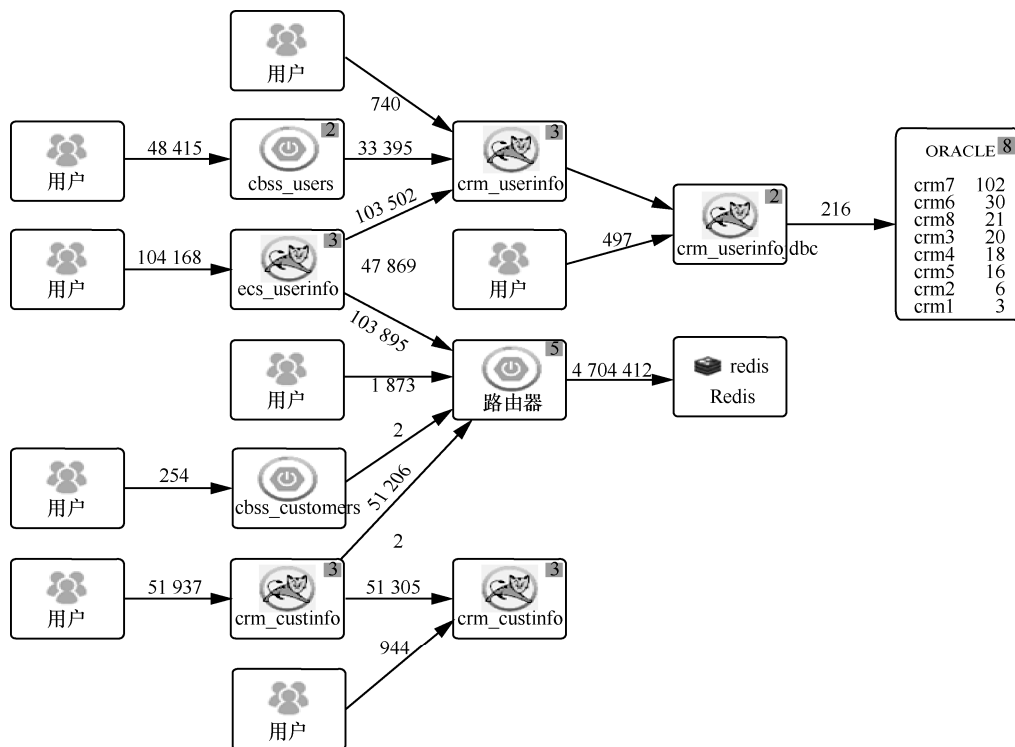
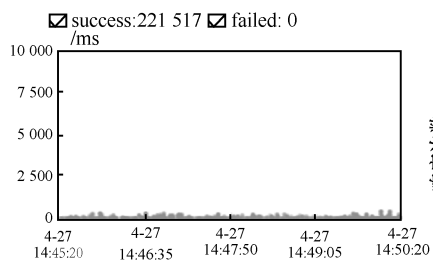


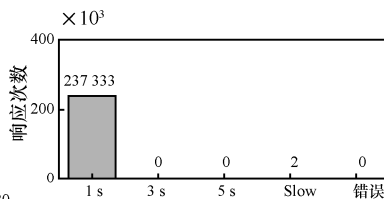
图6 4个熔断级别



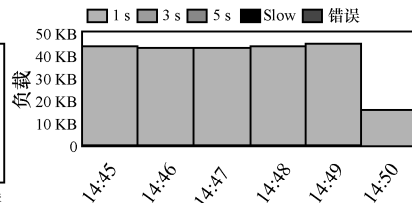
(a) 用户调用 ecs_userinfo 服务



(b) ecs_userinfo调用成功的次数



(c) 服务响应次数



(d) 负载情况

图7 PinPoint 调用链监控



该服务调用 `crm_userinfo` 服务,再调用 `route` 服务,它访问 `Redis` 数据库,箭头上的数字表示调用次数,这就是一条服务调用链。每个服务的右上角表示实例个数,图 7 (b) 表示 `ecs_userinfo` 调用成功的次数,图 7 (c) 表示服务响应次数,图 7 (d) 表示负载情况。

Kong 除了鉴权功能外,支持多种组件的配置。这里启用插件 `StatsD` 来采集各个服务的调用量、时延等情况,并将这些信息发送到 `Graphite`,将其配置为 `Grafana` 数据源,并以图表形式展现给运维人员,当超过预警值时,通过邮件和短信形式通知运维人员。

3.4.2 容器状态监控

通过定时访问 `Docker` 在主机 `Linux` 系统挂载的路径,获取容器 `CPU` 和内存的使用情况。定时访问 `Linux` 系统路径 `/sys/fs/cgroup/cup/docker/dockerID/` 下的 `CPUacct.usage` 获取 `CPU` 使用状态;同理,访问路径 `/sys/fs/cgroup/memory/docker/dockerID/` 下的 `memory.stat` 获取容器当前内存使用状态。

通过在服务中设置 `/health` 路径,并定时访问该路径,获取容器中服务的运行状态,若在指定时间(如 2 s)内没有响应,则服务运行出现问题,自动向运维人员报警。

此外,采用统一的监控页面对容器、服务及服务调用情况做全面的监控。通过实时调用 `Marathon API` 获取服务容器数、分配的 `CPU` 和内存。按照上述方法获得容器当前使用的 `CPU` 和内存。调用 `Hystrix API` 获取服务调用的吞吐量、平均时延、错误率和熔断级别。当这些变量超过预先设置的阈值,采用 `Websocket` 技术,在 `Web` 页面预警区给运维人员实时推送预警信息,并发送邮件和短信。

将容器中的日志按照统一路径映射到各个主机的系统的统一路径下。在各个节点对这些日志进行收集整理,提供统一的检索入口。

4 运行结果

本系统在中国联通生产环境部署了多个服务,为多个业务部门提供平台支撑。目前,生产环境有 14 个节点,每个节点有 32 个 `CPU`、内存 12 GB,共部署了 80 个服务 104 个容器。每个服务的调用频率和占用资源各不相同,经过一年的运行,系统能够稳定地支撑每一种服务,满足不同服务的需求。

4.1 服务调用次数和请求时延

本系统以低时延支撑多个服务的大量调用。这里选取了调用量较大的 4 个服务,分别是 `ecs_userinfo`、`cbss_users`、`blackname`、`cbss_userres`,每个服务分配 1 核 `CPU` 和 2 GB 内存。图 8 展示了 24 h 的调用量统计。图 9 展示了对应时刻响应时延,这两个数据都是每 10 s 统计一次。当 `ecs_userinfo` 在 11:00 左右有突发调用时,时延相应增加。`cbss_userres` 调用量稳定在 75 次左右,相应的时延平稳在 75 ms。其他两个服务的调用量在 5:00—12:00 快速增长,但是时延基本稳定在 25 ms 以下,一些时段有小幅增长。

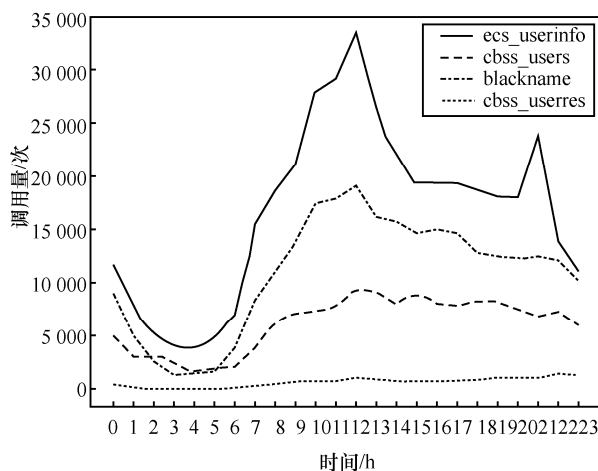


图 8 微服务调用量

为了保证服务能够在调用峰值及时响应请求,需要对服务进行压力测试,考察其吞吐量和内存占用情况。表 1 展示了 `ecs_userinfo` 的压力测

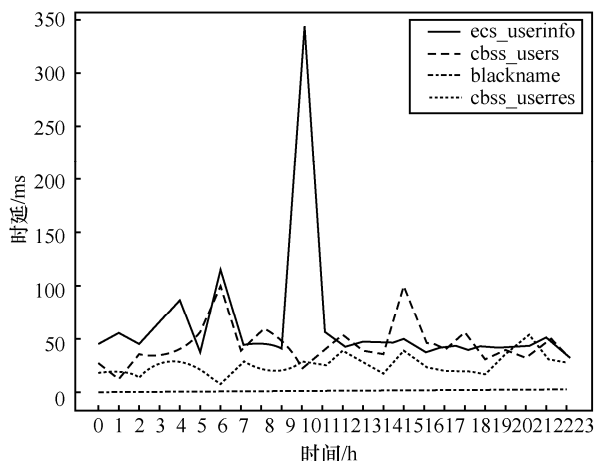


图9 微服务时延响应

试结果, 该服务共有两个容器, 每个容器分配了 10 GB 内存和 1 核 CPU, 表 1 中占用内存的“/”表示两个容器分别占用的内存。当请求数和并发量增加时, 吞吐量随之增加, 见表 1。当调用总量为 100 000、并发量为 10 000 时, 吞吐量达到 7 400 次/s。该服务实际调用量 400 次/s, 压测结果远远超出能够满足实际需要。

表1 ecs_userinfo 的压测结果

请求数	并发量	吞吐量/(次·s ⁻¹)	占用内存/GB	失败率
3 000	3 000	2 300	1.96/2.01	0
9 000	3 000	6 700	1.88/2.01	0
12 000	3 000	3 800	1.67/1.85	0
20 000	5 000	6 800	1.84/2.19	0
50 000	5 000	7 100	2.04/2.46	0
100 000	5 000	7 200	2.45/3.21	0
36 000	6 000	7 000	2.88/3.71	0
72 000	6 000	7 200	3.21/4.64	0
100 000	6 000	7 300	3.93/4.92	0
14 000	7 000	6 900	5.28/5.69	0
100 000	10 000	7 400	7.57/6.11	0

4.2 服务容器状态监控

为了保证容器的正常运行, 需要采集每个容器的监控指标。表 2 展示了 4 个服务容器的使用指标 CPU、内存、入网和出网流量。当调用量很低时, CPU 和内存的使用率很低, 如服务 sdcdr。ecs_userinfo 因为调用量较高, 各个指标高于其他服务。

表2 服务容器运行指标

服务名	CPU 使用率	内存使用率	入网流量/ (bit·s ⁻¹)	出网流量/ (bit·s ⁻¹)
route	30.04%	98%	345 331.11	793 472.64
sdcdr	2.53%	56.31%	1 074.76	1 265.73
blackname	13.83%	99.25%	92 141.56	208 470.4
ecs_userinfo	106.24%	97.26%	349 281.97	684 685.4

5 结束语

实现了在测试和生产环境容器化服务的持续集成和交付, 用户只需将必要的环境变量写入 json 文件, 由本系统自动打包并载入容器。实现一个服务多个版本同时验证, 允许用户灵活配置多个服务的多个版本联调测试, 达到生产不停机、版本无缝切换的效果。实现了根据业务调用规律自动增加或减少服务实例, 并且实现了在扩缩过程中零分组丢失的转发。采用统一配置中心管理容器化服务的配置参数, 支持服务运行中参数改变, 而不需要重启服务。

为了保证每个微服务都能更加稳健地运行, 未来需要在持续交付流程中纳入代码评审环节, 提高开发人员的代码质量。并且通过增加服务模拟模块, 使开发者可以方便地获取所依赖服务变化情况, 能够及时根据依赖的变化调整, 方便进行模拟验证。

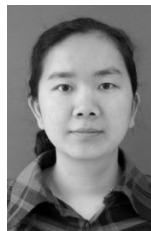
参考文献:

- [1] JARAMILO D, NGUYEN V, SMART R. Leveraging microservices architecture by using Docker technology[J]. SoutheastCon, 2016, 5(16): 1-5.
- [2] 廖建新. 大数据技术的应用现状与展望[J]. 电信科学, 2015, 31(7): 7-18.
LIAO J X. Big data technology: current applications and prospects[J]. Telecommunications Science, 2015, 31(7): 7-18.
- [3] 陈涛, 鲁萌, 陈彦名. 运营商大数据技术应用研究[J]. 电信科学, 2017, 33(1): 130-134.
CHEN T, LU M, CHEN Y M. Research on operators' big data technologies and applications[J]. Telecommunications Science, 2017, 33(1): 130-134.
- [4] 汪保友, 钱晶, 袁时金. 基于 Hadoop 的电信大数据采集方案研究与实现[J]. 电信科学, 2017, 33(1): 135-142.



- WANG B Y, QIAN J, YUAN S J. Research and implementation on acquisition scheme of telecom big data based on Hadoop[J]. Telecommunications Science, 2017, 33(1): 135-142.
- [5] YOUSIF M. Microservices[J]. IEEE Cloud Computing, 2016, 3(5): 4-5.
- [6] 张基, 李大中, 张呈宇, 等. DC/OS 关键技术与应用场景 [J]. 电信科学, 2016, 32(12):13-19.
- ZHANG J, LI D Z, ZHANG C Y, et al. Key technologies and application scenario of DC/OS[J]. Telecommunications Science, 2016, 32(12): 13-19.
- [7] BALALAIE A, HEYDARNOORI A, JAMSHIDI P. Microservices architecture enables DevOps: migration to a cloud-native architecture[J]. IEEE Software, 2016, 33(3): 42-52.
- [8] HINDMAN B, KONWINSKI A, ZAHARIA M, et al. Mesos: a platform for fine-grained resource sharing in the data center [C]// The 8th USENIX conference on Networked systems design and implementation, USENIX Association, March 30-April 1, 2011, Boston, USA. New York: ACM Press, 2011: 429-483.
- [9] SAHA P, GOVINDARAJU M, MARRU S, et al. Integrating Apache Airavata with Docker, Marathon, and Mesos[J]. Concurrency and Computation: Practice and Experience, 2015(28): 1952-1959.
- [10] FIELDING R T. Architectural styles and the design of network-based software architecture[D]. Irvine: University of California, Irvine, 2000.
- [11] PARISA H, YVES L, ABDALLAH S. QoS assurance with light virtualization-a survey[C]//2016 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Dec 12-15, 2016, Luxembourg. Piscataway: IEEE Press, 2016: 1-12.
- [12] MEROÑO-PENUELA A, HOEKSTRA R. grlc makes GitHub taste like linked data APIs[C]//ESWC 2016 Satellite Events, May 29-June 2, 2016, Heraklion, Greece. [S.l.:s.n.], 2016: 342-353.
- [13] AHMED T M, BEZEMER C P, CHEN T H, et al. Studying the effectiveness of application performance management (APM) tools for detecting performance regressions for web applications: an experience report[C]//The 13th International Conference on Mining Software Repositories, May 14-22, 2016, Austin, USA. Piscataway: IEEE Press, 2016: 1-12.
- [14] CLINTON G, ZACHARY T. Elasticsearch: the definitive guide: a distributed real-time search and analytics engine[M]. Sebastopol: O'Reilly Media, 2015.

[作者简介]



杨迪 (1985-), 女, 博士, 中国联通网络有限公司软件研究院工程师, 主要研究方向为人工智能、软件架构。