45-基于Kubernetes的云原生架构设计

你好，我是孔令飞。

前面两讲，我们一起看了云技术的演进之路。软件架构已经进入了云原生时代，云原生架构是当下最流行的软件部署架构。那么这一讲，我就和你聊聊什么是云原生，以及如何设计一种基于Kubernetes的云原生部署架构。

云原生简介

云原生包含的概念很多，对于一个应用开发者来说，主要关注点是如何开发应用，以及如何部署应用。所以，这里我在介绍云原生架构的时候，会主要介绍应用层的云原生架构设计和系统资源层的云原生架构设计。

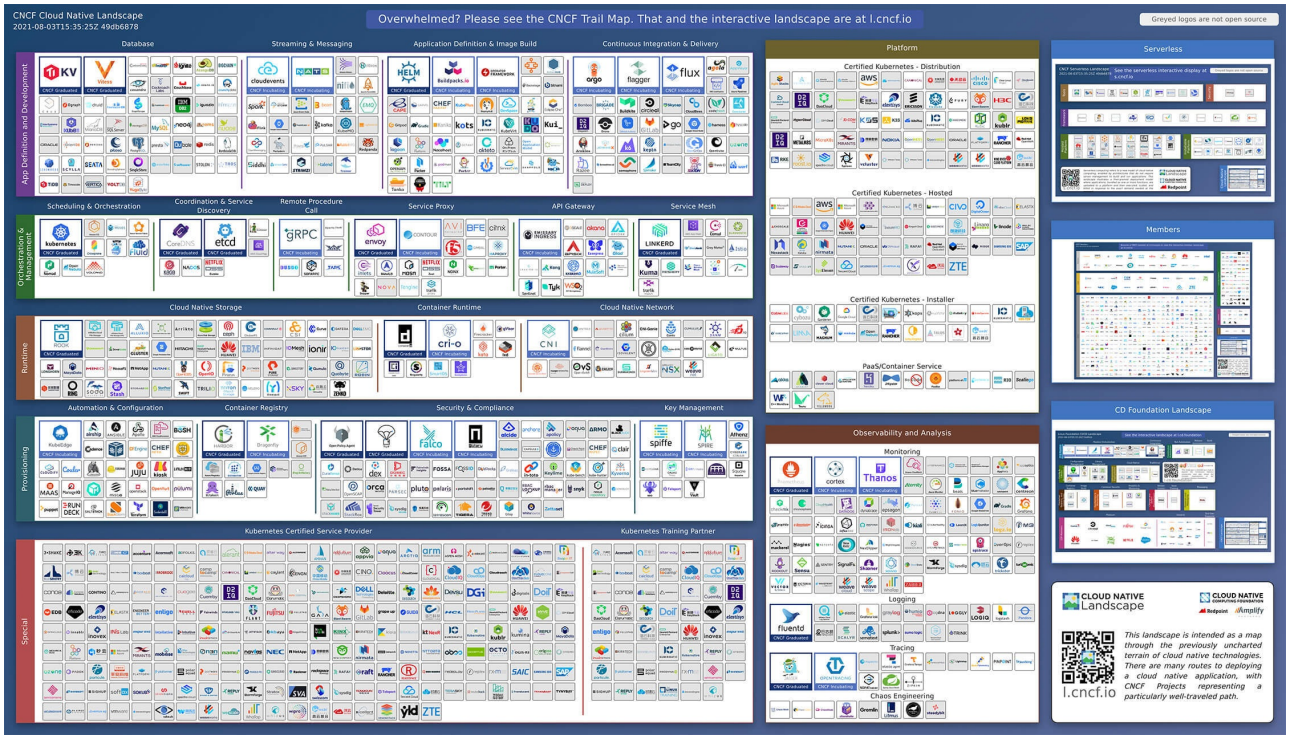
在设计云原生架构时，应用生命周期管理层的云原生技术，我们主要侧重在使用层面，所以这里我就不详细介绍应用生命周期管理层的云原生架构了。后面的云原生架构鸟瞰图中会提到它，你可以看看。

另外，在介绍云原生时，也总是绕不开云原生计算基金会。接下来，我们就先来简单了解下CNCF基金会。

CNCF（云原生计算基金会）简介

CNCF（Cloud Native Computing Foundation，云原生计算基金会），2015年由谷歌牵头成立，目前已有一百多个企业与机构作为成员，包括亚马逊、微软、思科、红帽等巨头。CNCF致力于培育和维护一个厂商中立的开源社区生态，用以推广云原生技术。

CNCF目前托管了非常多的开源项目，其中有很多我们耳熟能详的项目，例如 Kubernetes、Prometheus、Envoy、Istio、etcd等。更多的项目，你可以参考CNCF公布的Cloud Native Landscape，它给出了云原生生态的参考体系，如下图所示：

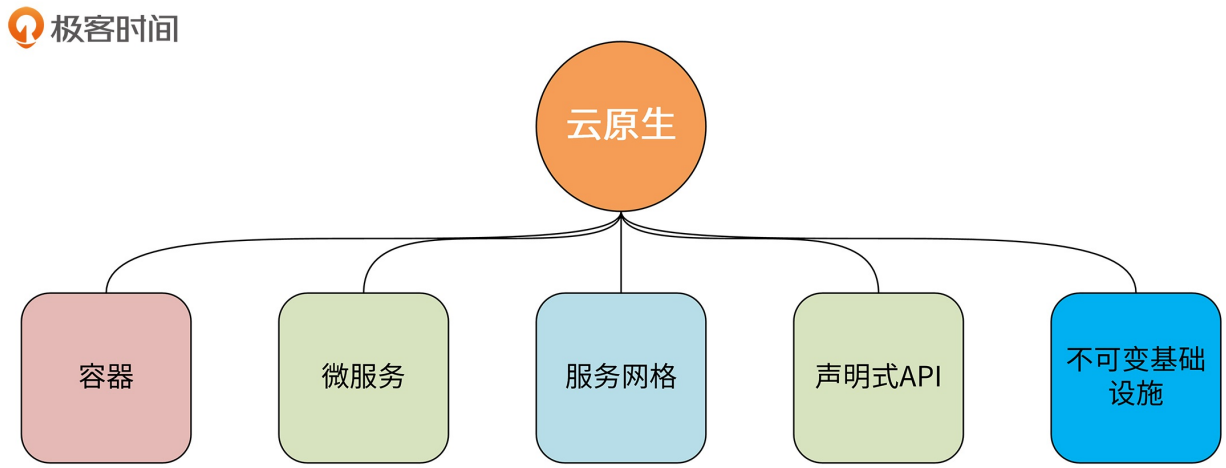


什么是云原生？

CNCF官方在2018年发布了云原生v1.0，并给出了定义：

> “云原生技术有利于各组织在公有云、私有云和混合云等新型动态环境中，构建和运行可弹性扩展的应用。云原生的代表技术包括容器、服务网格、微服务、不可变基础设施和声明式API。 这些技术能够构建容错性好、易于管理和便于观察的松耦合系统。结合可靠的自动化手段，云原生技术使工程师能够轻松地对系统作出频繁和可预测的重大变更。”

简单点说，云原生（Cloud Native）是一种构建和运行应用程序的方法，是一套技术体系和方法论。云原生中包含了3个概念，分别是技术体系、方法论和云原生应用。整个云原生技术栈是围绕着Kubernetes来构建的，具体包括了以下核心技术栈：



这里来介绍下这些核心技术栈的基本内容。

- **容器**：Kubernetes的底层计算引擎，提供容器化的计算资源。
- **微服务**：一种软件架构思想，用来构建云原生应用。
- **服务网格**：建立在Kubernetes之上，作为服务间通信的底座，提供强大的服务治理功能。
- **声明式 API**：一种新的软件开发模式，通过描述期望的应用状态，来使系统更加健壮。
- **不可变基础设施**：一种新的软件部署模式，应用实例一旦被创建，便只能重建不能更新，是现代运维的基础。

在 43讲 和 44讲 中，我介绍了容器、服务网格和微服务，这里再补充介绍下不可变基础设施和声明式API。

不可变基础设施（Immutable Infrastructure）的构想，是由Chad Fowler 于 2013 年提出的。具体来说就是：一个应用程序的实例，一旦被创建，就会进入只读的状态，后面如果想变更这个应用程序的实例，只能重新创建一个新的实例。通过这种模式，可以确保应用程序实例的一致性，这使得落地DevOps更加容易，并可以有效减少运维人员管理配置的负担。

声明式API是指我们通过工具描述期望的应用状态，并由工具保障应用一直处在我们期望的状态。

Kubernetes的API设计，就是一种典型的声明式API。例如，我们在创建Deployment时，在Kubernetes YAML文件中声明应用的副本数为2，即设置`replicas: 2`，Deployment Controller就会确保应用的副本数一直为2。也就是说，如果当前副本数大于2，Deployment Controller会删除多余的副本；如果当前副本数小于2，会创建新的副本。

声明式设计是一种设计理念，同时也是一种工作模式，它使得你的系统更加健壮。分布式系统环境可能会出现各种不确定的故障，面对这些组件故障，如果使用声明式 API ，你只需要查看对应组件的 API 服务器状态，再确定需要执行的操作即可。

什么是云原生应用？

上面，我介绍了什么是云原生，接下来再介绍下什么是云原生应用。

整体来看，云原生应用是指生而为云的应用，应用程序从设计之初就考虑到了云的环境，可以在云上以最佳姿势运行，充分利用和发挥云平台提供的各种能力。具体来看，云原生应用具有以下三大特点：

- 从应用生命周期管理维度来看，使用DevOps和CI/CD的方式，进行开发和交付。
- 从应用维度来看，以微服务原则进行划分设计。
- 从系统资源维度来看，采用Docker + Kubernetes的方式来部署。

看完上面的介绍，你应该已经对云原生和云原生应用有了一定的理解，接下来我就介绍一种云原生架构实现。因为云原生内容很多，所以这里的介绍只是起到抛砖引玉的作用，让你对云原生架构有初步的理解。至于在具体业务中如何设计云原生架构，你还需要根据业务、团队和技术栈等因素综合考虑。

云原生架构包含很多内容，如何学习？

云原生架构中包含了很多概念、技术，那么我们到底如何学习呢？在前面的两讲中，我分别从系统资源层、应用层、应用生命周期管理层介绍了云技术。这3个层次基本上构成了整个云计算的技术栈。

今天，我仍然会从这三个层次入手，来对整个云原生架构设计进行相对完整的介绍。每个层次涉及到的技术很多，这一讲我只介绍每一层的核心技术，通过这些核心技术来看每一层的构建方法。

另外，因为应用生命周期管理层涉及到的技术栈非常多，所以今天不会详细讲解每种生命周期管理技术的实现原理，但会介绍它们提供的能力。

除了功能层面的架构设计之外，我们还要考虑部署层面的架构设计。对于云原生架构的部署，通常我们需要关注以下两点：

- 容灾能力：容灾能力是指应用程序遇到故障时的恢复能力。在互联网时代，对应用的容灾能力有比较高的要求。理想情况是系统在出现故障时，能够无缝切换到另外一个可用的实例上，继续提供服务，并做到用户无感知。但在实际开发中，无缝切换在技术上比较难以实现，所以也可以退而求其次，允许系统在一定时间内不可用。通常这个时间需要控制在秒级，例如5s。容灾能力可以通过负载均衡、健康检查来实现。
- 扩缩容能力：扩缩容能力指的是系统能够根据需要扩缩容，可以手动扩缩容，也可以自动扩缩容。互联网时代对扩缩容能力的要求也比较高，需要实现自动扩缩容。我们可以基于一些自定义指标，例如CPU使用率、内存使用率等来自动扩缩容。扩容也意味着能够承载更多的请求，提高系统的吞吐量；缩容，意味着能够节省成本。扩缩容能力的实现，需要借助于负载均衡和监控告警能力。

容灾能力和扩缩容能力都属于高可用能力。也就是说，在部署层面，需要我们的架构具备高可用能力。

接下来，我就重点介绍下系统资源层和应用层的云原生架构设计，并简单介绍下应用生命周期管理层的核心功能构建。在介绍完架构设计之后，我还会介绍下这些层面的高可用架构设计。

系统资源层的云原生架构设计

先来看系统资源层面的云原生架构设计。对于一个系统来说，系统资源的架构是需要优先考虑的。在云原生架构中，当前的业界标准是通过Docker提供系统资源（例如CPU、内存等），通过Kubernetes来编排Docker容器。Docker和Kubernetes的架构，我在43讲中介绍过，这里我主要介绍下系统资源层面的高可用架构设计。

基于Docker+Kubernetes的方案，高可用架构是通过Kubernetes高可用架构来实现的。要实现整个Kubernetes集群的高可用，我们需要分别实现以下两类高可用：

- Kubernetes集群的高可用。
- Kubernetes集群中所部署应用的高可用。

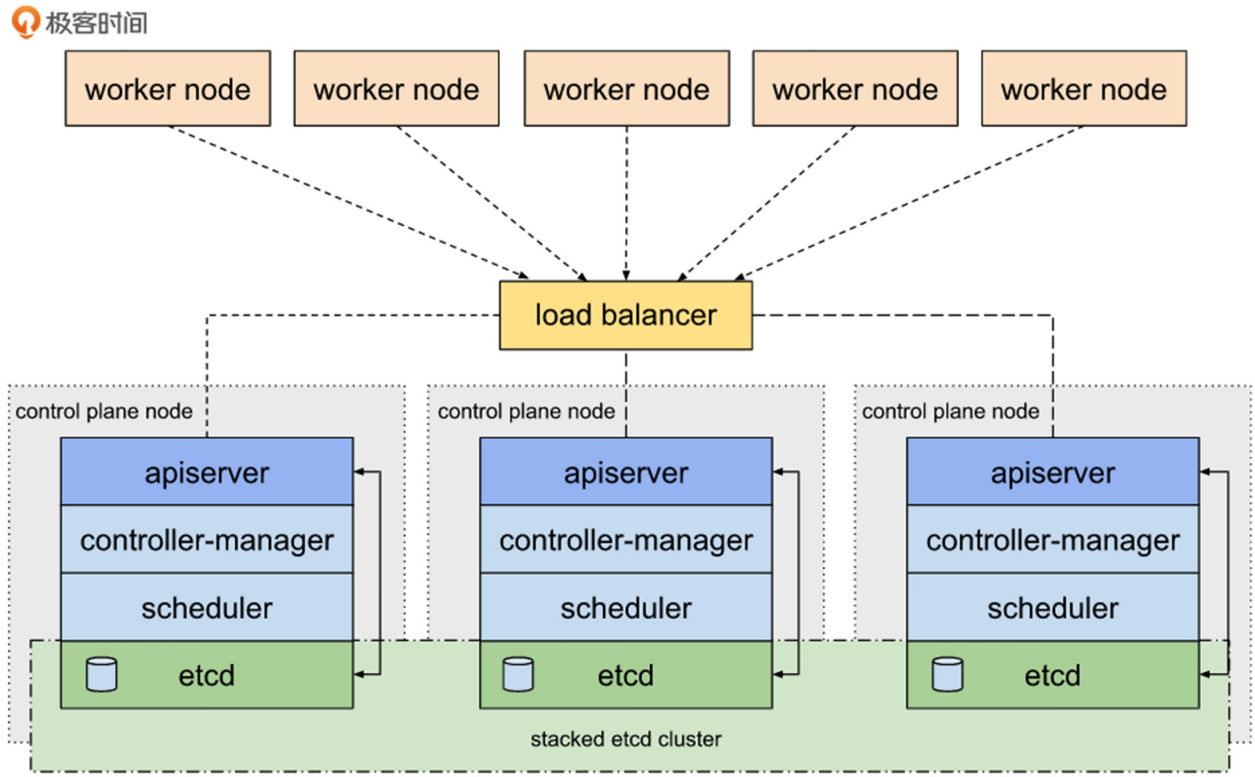
我们来分别看下这两个高可用方案。

Kubernetes集群高可用方案设计

通过43讲的学习，我们知道Kubernetes由kube-apiserver、kube-controller-manager、kube-scheduler、cloud-controller-manager、etcd、kubelet、kube-proxy、container runtime 8大核心组件组成。

其中，kube-apiserver、kube-controller-manager、kube-scheduler、cloud-controller-manager、etcd通常部署在master节点，kubelet、kube-proxy、container runtime 部署在Node节点上。实现Kubernetes集群的高可用，需要分别实现这8大核心组件的高可用。

Kubernetes集群的高可用架构图如下：



上面图片展示的方案中，所有管理节点都部署了kube-apiserver、kube-controller-manager、kube-

scheduler、etcd等组件。kube-apiserver均与本地的etcd进行通信，etcd在三个节点间同步数据；而kube-controller-manager、kube-scheduler和cloud-controller-manager，也只与本地的kube-apiserver进行通信，或者通过负载均衡访问。

一个Kubernetes集群中有多个**Node节点**，当一个Node节点故障时，Kubernetes的调度组件kube-controller-manager会将Pod调度到其他节点上，并将故障节点的Pod在其他可用节点上重建。也就是说，只要集群中有两个以上的节点，当其中一个Node故障时，整个集群仍然能够正常提供服务。换句话说，集群的**kubelet、kube-proxy、container runtime组件**可以是单点的，不用实现这些组件的高可用。

接下来，我们来看下**Master节点**各组件是如何实现高可用的。先来说下**kube-apiserver组件的高可用方案设计**。

因为kube-apiserver是一个无状态的服务，所以可以通过部署多个kube-apiserver实例，其上挂载负载均衡的方式来实现。其他所有需要访问kube-apiserver的组件，都通过负载均衡来访问，以此实现kube-apiserver的高可用。

kube-controller-manager、cloud-controller-manager和kube-scheduler因为是有状态的服务，所以它们的高可用能力不能通过负载均衡来实现。kube-controller-manager/kube-scheduler/cloud-controller-manager通过–leader-elect=true参数开启分布式锁机制，来进行leader election。

你可以创建多个kube-controller-manager/kube-scheduler/cloud-controller-manager实例，同一时刻只有一个实例能够获取到锁，成为leader，提供服务。如果当前leader故障，其他实例感知到leader故障之后会自动抢锁，成为leader继续提供服务。通过这种方式，我们实现了kube-controller-manager/kube-scheduler/cloud-controller-manager组件的高可用。

当kube-apiserver、kube-controller-manager、kube-scheduler、cloud-controller-manager故障时，我们期望这些组件能够自动恢复，这时候可以将这些组件以Static Pod的方式进行部署，这样当Pod故障时，上述实例就能够自动被拉起。

etcd的高可用方案有下面这3种思路：

- 使用独立的etcd集群，独立的etcd集群自带高可用能力。
- 在每个Master节点上，使用Static Pod来部署etcd，多个节点上的etcd实例数据相互同步。每个kube-apiserver只与本Master节点的etcd通信。
- 使用CoreOS提出的self-hosted方案，将etcd集群部署在kubernetes集群中，通过kubernetes集群自身的容灾能力来实现etcd的高可用。

这三种思路，需要你根据实际需要进行选择，在实际生产环境中，第二种思路用得最多。

到这里，我们就实现了整个Kubernetes集群的高可用。接下来，我们来看下Kubernetes集群中，应用的高可用是如何实现的。

Kubernetes应用的高可用

Kubernetes自带了应用高可用能力。在Kubernetes中，应用以Pod的形式运行。你可以通过Deployment/StatefulSet来创建应用，并在Deployment/StatefulSet中指定多副本和Pod的健康检查方式。

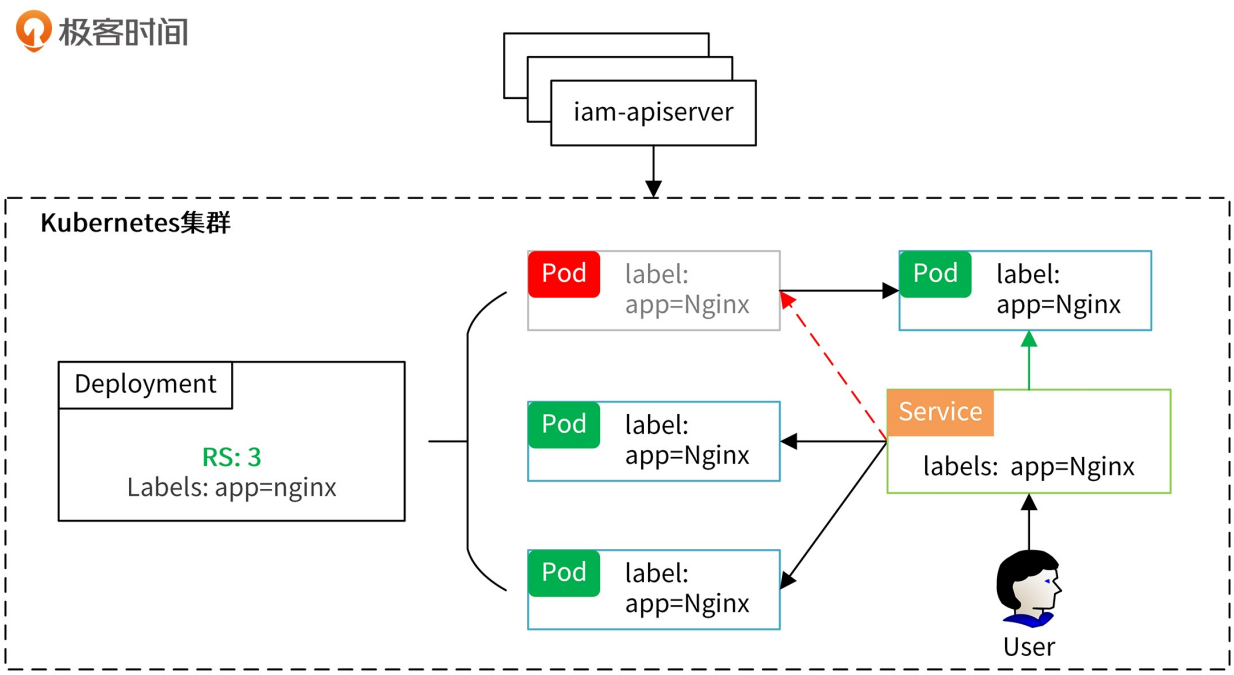
当Pod健康检查失败时，Deployment/StatefulSet的控制器（ReplicaSet）会自动销毁故障Pod，并创建一个新的Pod，替换故障的Pod。

你可能会问：当Pod故障时，怎么才能避免请求被调度到已故障的Pod上，造成请求失败？这里我也详细介绍下。

在Kubernetes中，我们可以通过Kubernetes Service或者负载均衡来访问这些Pod。当通过负载均衡来访问Pod时，负载均衡后端的RS（Real Server）实例其实就是Pod。我们创建了多个Pod，负载均衡可以自动根据Pod的健康状况来进行负载。

接下来，我们主要看下这个问题：当通过Kubernetes Service访问Pod时，如何实现高可用？

高可用原理如下图所示：



在Kubernetes中，我们可以给每个Pod打上标签（Label），标签是一个key-value对，例如`label: app=Nginx`。当我们访问Service时，Service会根据它配置的Label Selector，匹配具有相同Label的Pod，并将这些Pod的endpoint地址作为其后端RS。

举个例子，你可以看看上面的图片：Service的Label Selector是Labelsapp=Nginx，这样就会选择我们创建的具有`label: app=Nginx`的3个Pod实例。这时候，Service会根据其负载均衡策略，选取一个Pod将请求流量转发过去。当其中一个Pod故障时，Kubernetes会自动将故障Pod的endpoint从Service后端对应的RS列表中剔除。

由Deployment创建的ReplicaSet，这时候也会发现有一个Pod故障，健康的Pod实例数变为2，这时候跟其期望的值3不匹配，就会自动创建一个新的健康Pod，替换掉故障的Pod。因为新Pod满足Service的Label Selector，所以新Pod的endpoint会被Kubernetes自动添加到Service对应的endpoint列表中。

通过上面这些操作，Service后端的RS中，故障的Pod IP被新的、健康的Pod IP所替换，通过Service访问的后端Pod就都是健康的。这样，就通过Service实现了应用的高可用。

从上面的原理分析中，我们也可以发现，Service本质上是一个负载均衡器。

Kubernetes还提供了滚动更新（RollingUpdate）机制，来确保在发布时服务正常可用。这个机制的大致原理是：在更新时，先创建一个Pod，再销毁一个Pod，依次循环，直到所有的Pod都更新完成。在更新时，我们还可以控制每次更新的Pod数，以及最小可用的Pod数。

接下来，我们再来看下应用层的云原生架构设计和高可用设计。

应用层的云原生架构设计

在云原生架构中，我们采用微服务架构来构建应用。所以，这里我主要围绕着微服务架构的构建方式来介绍。先和你谈谈我对微服务的理解。

从本质上来说，微服务是一个轻量级的Web服务，只不过在微服务场景下，我们通常考虑的不是单个微服务，而是更多地考虑由多个微服务组成的应用。也就是说，一个应用由多个微服务组成，多个微服务就带来了一些单个Web服务不会面临的问题，例如部署复杂、排障困难、服务依赖复杂、通信链路长，等等。

在微服务场景下，除了编写单个微服务（轻量级的Web服务）之外，我们更多是要专注于解决应用微服务化所带来的挑战。所以，在我看来，微服务架构设计包括两个重要内容：

- 单个微服务的构建方式；
- 解决应用微服务化带来的挑战。

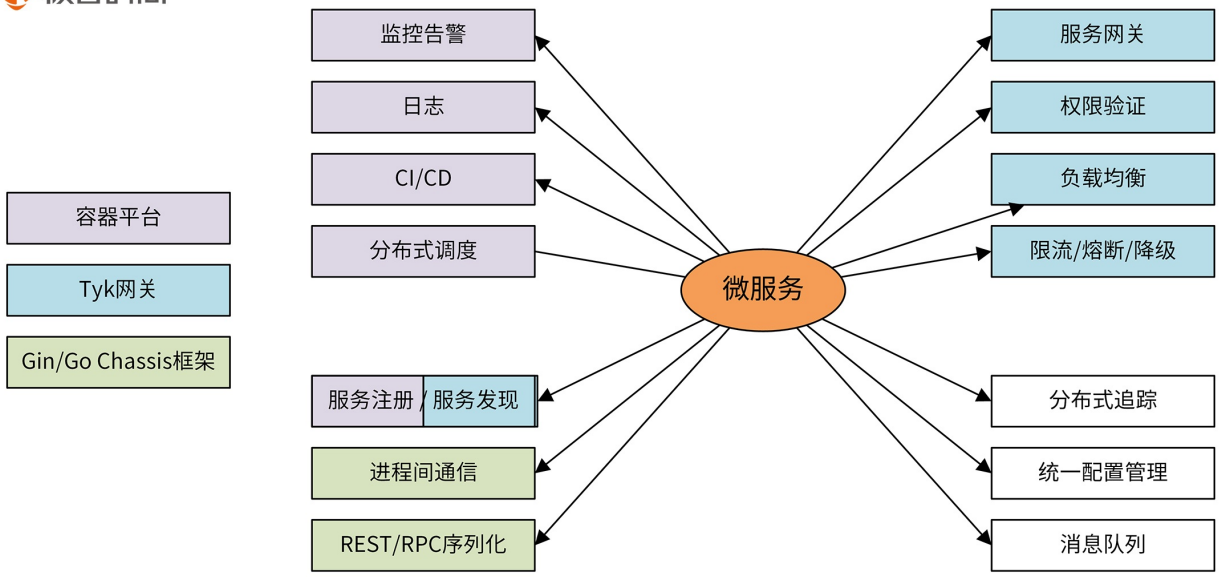
微服务实现

我们可以通过两种方式来构建微服务：

- 采用Gin、Echo等轻量级Web框架。
- 采用微服务框架，例如 go-chassis、go-micro、go-kit等。

如果要解决应用微服务化带来的挑战，我们需要采用多种技术和手段，每种技术和手段会解决一个或一部分挑战。

综上，在我看来，微服务本质上是一个轻量级的Web服务，但又包含一系列的技术和手段，用来解决应用微服务化带来的挑战。微服务的技术栈如下图所示：



不同的技术栈可以由不同的方式来实现，并解决不同的问题：

- 监控告警、日志、CI/CD、分布式调度，可以由Kubernetes平台提供的能力来实现。
- 服务网关、权限验证、负载均衡、限流/熔断/降级，可以由网关来实现，例如Tyk网关。
- 进程间通信、REST/RPC序列化，可以借助Web框架来实现，例如Gin、Go Chassis、gRPC、Sprint Cloud。
- 分布式追踪可以由Jaeger来实现。
- 统一配置管理可以由Apollo配置中心来实现。
- 消息队列可以由NSQ、Kafka、RabbitMQ来实现。

上面的服务注册/服务发现，有3种实现方式：

- 通过Kubernetes Service来进行服务注册/服务发现，Kubernetes自带服务注册/服务发现功能。使用此方式，我们不需要额外的开发。
- 通过服务中心来实现服务注册/服务发现功能。采用这种方式，需要我们开发并部署服务中心，服务中心通常可以使用etcd/consul/mgmet来实现，使用etcd的较多。
- 通过网关，来进行服务注册/服务发现。这种情况下，可以将服务信息直接上报给网关服务，也可以将服务信息上报到一个服务中心，例如etcd中，再由网关从服务中心中获取。

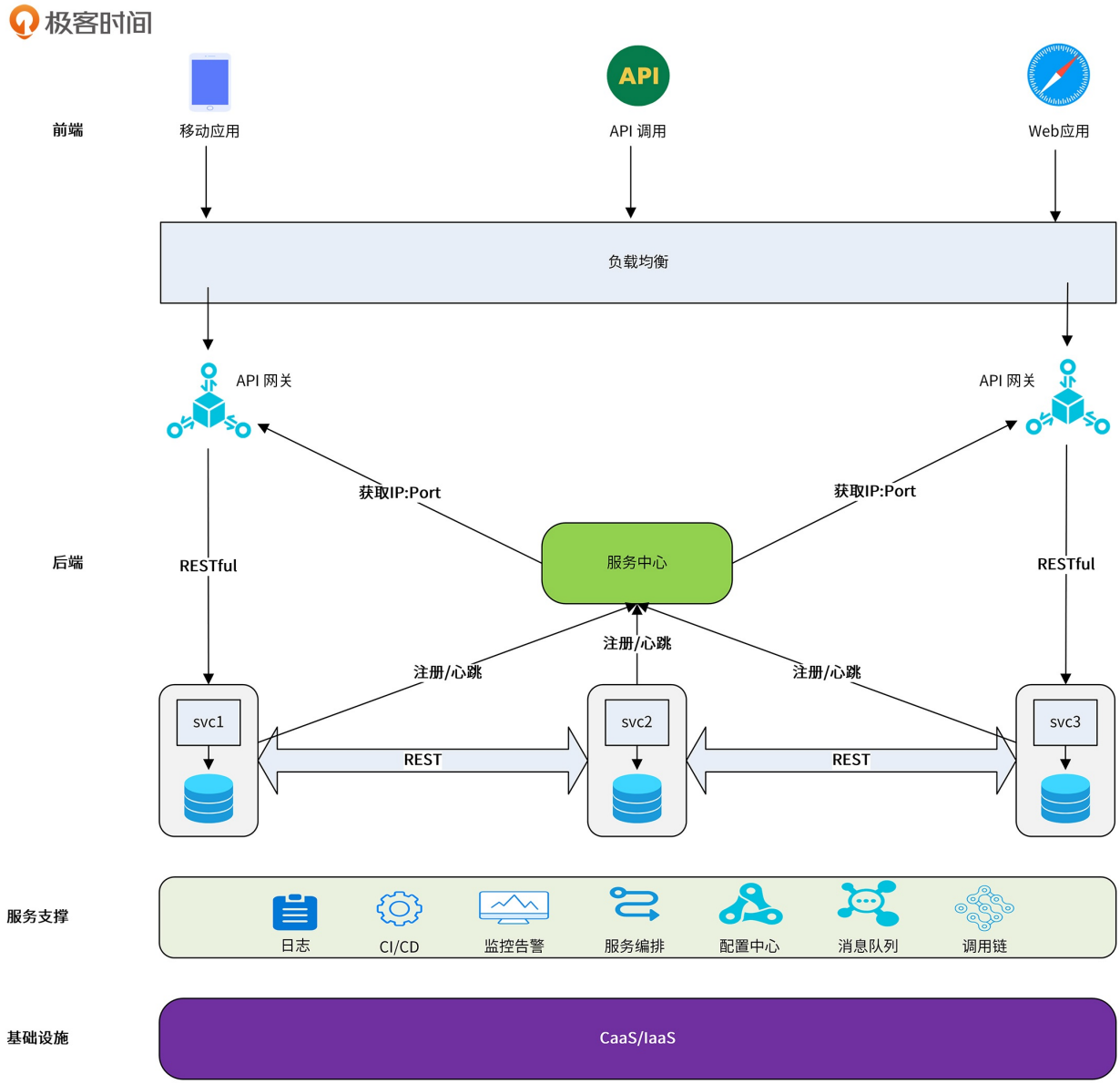
这里要注意，原生的Kubernetes集群是不支持监控告警、日志、CI/CD等功能的。我们在使用Kubernetes集群时，通常会使用一个基于Kubernetes开发而来的Kubernetes平台，例如腾讯云容器服务TKE。

在Kubernetes平台中，通常会基于一些优秀的开源项目，进行二次开发，来实现平台的监控告警、日志、CI/CD等功能。

- 监控告警：基于Prometheus来实现。
- 日志：基于EFK日志解决方案来实现。
- CI/CD：可以自己研发，也可以基于优秀的开源项目来实现，例如 drone。

微服务架构设计

上面我介绍了如何实现微服务，这里我再来具体讲讲，上面提到的各个组件/功能是如何有机组合在一起，共同构建一个微服务应用的。下面是微服务的架构图：



在上图中，我们将微服务应用层部署在Kubernetes集群中，在Kubernetes集群之上，可以构建微服务需要的其他功能，例如监控告警、CI/CD、日志、调用链等。这些功能共同完成应用的生命周期管理。

我们在微服务的最上面挂载负载均衡。客户端，例如移动端应用、Web应用、API调用等，都通过负载均衡来访问微服务。

微服务在启动时会将自己的endpoint信息（通常是`ip:port`格式）上报到服务中心。微服务也会定时上报自己的心跳到服务中心。在服务中心中，我们可以监控微服务的状态，剔除不健康的微服务，获取微服务之间的访问数据，等等。如果要通过网关调用微服务，或者需要使用网关做负载均衡，那我们还需要网关从服务中心中获取微服务的endpoint信息。

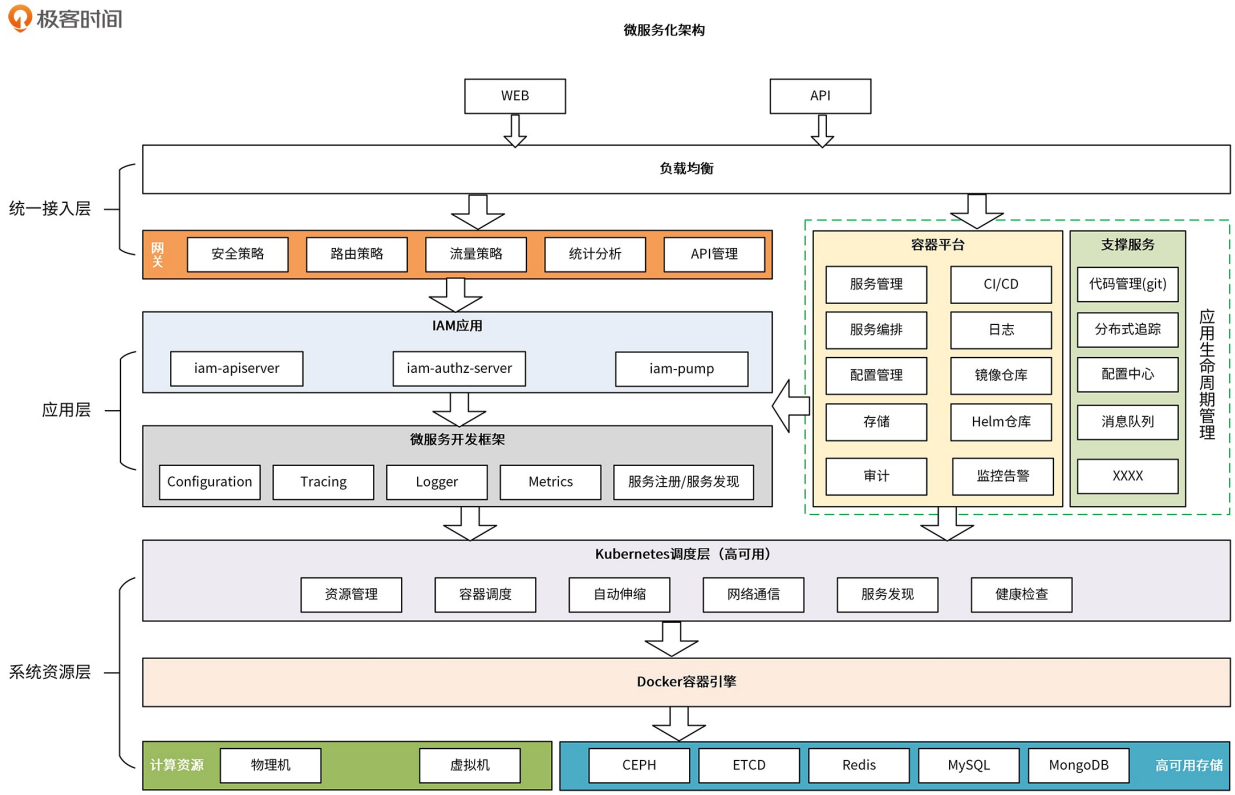
微服务高可用架构设计

我们再来看下如何设计微服务应用的高可用能力。

我们可以把所有微服务组件以Deployment/StatefulSet的形式部署在Kubernetes集群中，副本数至少设置为两个，更新方式为滚动更新，设置服务的监控检查，并通过Kubernetes Service或者负载均衡的方式访问服务。这样，我们就可以不用做任何改造，直接使用Kubernetes自有的容灾能力，实现微服务架构的高可用。

云原生架构鸟瞰图

上面，我介绍了系统资源层和应用层的云原生架构设计，但还不能构成整个云原生架构设计。这里，我通过一张云原生架构鸟瞰图，来整体介绍下云原生架构的设计方案。



上图的云原生架构分为4层，除了前面提到的系统资源层、应用层、应用生命周期管理层之外，又加了统一接入层。接下来，我来介绍下这些层在云原生架构中的作用。

在最下面的**系统资源层**，我们除了提供传统的计算资源（物理机、虚拟机）之外，还可以提供容器化的计算资源和高可用的存储服务。其中，容器化的计算资源是基于传统的物理机/虚拟机来构建的。

在云原生架构中，我们更应该使用容器化的计算资源，通过Docker容器技术来隔离并对外提供计算资源，通过Kubernetes来编排容器。Docker + Kubernetes的组合使用，可以构建出一个非常优秀的系统资源层。这个系统资源层，自带了资源管理、容器调度、自动伸缩、网络通信、服务发现、健康检查等企业应用需要的核心能力。

在云原生时代，这些系统资源除了具有容器化、轻量化的特点之外，还越来越倾向于朝着Serverless化的方向去构建：系统资源免申请、免运维，按需计费，具备极致的弹性伸缩能力，并能够缩容到0。Serverless化的系统资源，可以使开发者只聚焦在应用层的应用功能开发上，而不用再把时间浪费在系统层的运维工作上。

在系统资源层之上，就可以构建我们的**应用层**了。云原生架构中，应用的构建方式，基本上都是采用的微服务架构。开发一个微服务应用，我们可以使用微服务框架，也可以不使用。二者的区别是，微服务框架替我

们完成了服务治理相关功能，让我们不需要再开发这些功能。

在我看来，这一点有利有弊。好处当然是节省了开发工作量。至于坏处，主要有两方面：一方面，在实现方式和实现思路上，微服务框架所集成的服务治理功能并不一定是最适合我们的方案。另一方面，使用微服务框架还意味着我们的应用会跟微服务框架耦合，不能自由选择服务治理技术和方式。所以，在实际开发中，你应该根据需要，自行选择微服务的构建方式。

一般来说，一个微服务框架中，至少集成了这些服务治理功能：配置中心、调用链追踪、日志系统、监控、服务注册/服务发现。

再往上，我们就实现了**统一接入层**。统一接入层中包含了负载均衡和网关两个组件，其中负载均衡作为服务的唯一入口，供API、Web浏览器、手机终端等客户端访问。通过负载均衡，可以使我们的应用在故障时，能够自动切换实例，在负载过高时能够水平扩容。负载均衡下面还对接了网关，网关提供了一些通用能力，例如安全策略、路由策略、流量策略、统计分析、API管理等能力。

最后，我们还可以构建一系列的应用生命周期管理技术，例如服务编排、配置管理、日志、存储、审计、监控告警、消息队列、分布式链路追踪。这些技术中，一些可以基于Kubernetes，集成在我们的Kubernetes平台中，另一些则可以单独构建，供所有产品接入。

公有云版云原生架构

上面我们提到，云原生架构涉及到很多的技术栈。如果公司有能力，可以选择自己开发；如果觉得人力不够、成本太高，也可以使用公有云厂商已经开发好的云原生基础设施。使用云厂商的云原生基础设施，好处很明显：这些基础设施专业、稳定、免开发、免运维。

为了补全云原生架构设计版图，这里我也介绍一个公用云版的云原生架构设计。那么，公有云厂商会提供哪些云原生基础设施呢？这里我介绍下腾讯云提供的云原生解决方案。解决方案全景如下图所示：

可以看到，**腾讯云提供了全栈的云原生能力。**

腾讯云基于底层的云原生能力，提供了一系列的云原生解决方案。这些解决方案，是已经设计好的云原生架构构建方案，可以帮助企业快速落地云原生架构，例如混合云解决方案、AI解决方案、IoT解决方案等。

那么，腾讯云底层提供了哪些云原生能力呢？我们一起来看下。

在应用层，通过TSF微服务平台，我们可以实现微服务的构建，以及微服务的服务治理能力。另外，还提供了更多的应用构建架构，例如：

- Serverless Framework，可以构建Serverless应用。
- CloudBase，云原生一体化应用开发平台，可以快速构建小程序、Web、移动应用。
- …

在系统资源层，腾讯云提供了多种计算资源提供形态。例如：通过TKE，可以创建原生的Kubernetes集群；通过EKS，可以创建Serverless化的Kubernetes集群；通过TKE-Edge，可以创建能够纳管边缘节点的Kubernetes集群。此外，还提供了开源容器服务平台TKEStack，TKEStack是一个非常优秀的容器云平台，在代码质量、稳定性、平台功能等方面，都在开源的容器云平台中处于龙头地位，也欢迎你Star。

在应用生命周期管理这一层，提供了云原生的etcd、Prometheus服务。此外，还提供了CLS日志系统，供你保存并查询应用日志；提供了云监控，供你监控自己的应用程序；提供了容器镜像服务（TCR），用来保存Docker镜像；提供了CODING DevOps平台，用来支持应用的CI/CD；提供了调用链跟踪服务（TDW），用来展示微服务的调用链。

在统一接入层，腾讯云提供了功能强大的API网关。此外，还提供了多种Serverless化的中间件服务，例如消息队列TDMQ、云原生数据库TDSQL等。

所有这些云原生基础设施，都有共同的特点，就是免部署、免运维。换句话说，在腾讯云，你可以只专注于使用编程语言编写你的业务逻辑，其他的一切都交给腾讯云来搞定。

总结

云原生架构设计，包含了系统资源层、应用层、统一接入层和应用生命周期管理层4层。

在系统资源层，可以采用Docker + Kubernetes的方式来提供计算资源。我们所有的应用和应用生命周期管理相关的服务，都可以部署在Kubernetes集群中，利用Kubernetes集群的能力实现服务发现/服务注册、弹性伸缩、资源调度等核心能力。

在应用层，可以采用微服务架构，来构建我们的应用。具体构建时，我们可以根据需要，采用类似Gin这种轻量级的Web框架来构建应用，然后再实现旁路的服务治理功能；也可以采用集成了很多服务治理功能的微服务框架，例如 go-chassis、go-micro等。

因为我们采用了微服务架构，为了能够将微服务的一些功能，例如：认证授权、限流等功能最大化的复用，我们又提供了统一接入层。可以通过API网关、负载均衡、服务网格等技术来构建统一接入层。

在应用生命周期管理这一层，我们可以实现一些云原生的管理平台，例如 DevOps、监控告警、日志、配置中心等，并使我们的应用以云原生化的方式接入这些平台，使用这些平台提供的能力。

最后，我还介绍了腾讯云的云原生基础设施。通过腾讯云提供的云原生能力，你可以专注于使用编程语言编写你的业务逻辑，其他的各种云原生能力，都可以交给云厂商来帮你实现。

课后练习

1. 思考下，服务注册/服务发现的3种实现方式中，哪种方法适用于你的项目，为什么？
2. 思考下，在设计云原生架构时，还需要考虑哪些重要的点？欢迎你在留言区分享。

欢迎你在留言区与我交流讨论，我们下一讲见。