https://www.bilibili.com/video/BV1jd4y1P7th?p=2&spm_id_from=pageDriver&vd_source=6d1166f6ebf24940077bce753ef61d75

1、云原生简介

云原生的代表技术如下：docker和微服务比较熟悉了

1.1、如何理解云原生

云：阿里云，腾讯云这类的云基础设施（以前叫it基础设施）
原生：比如原生家庭
云原生：企业开发的应用能够在云计算平台里原生开发和运行的（就是基于这个云 开发和运行的）

1.2、云原生概念的由来（2013年提出）

有一个背景先了解：

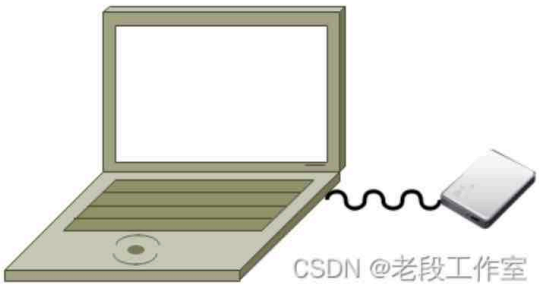
疑问：k8s最新版本不支持docker了，为什么

https://blog.csdn.net/lduan_001/article/details/125301335

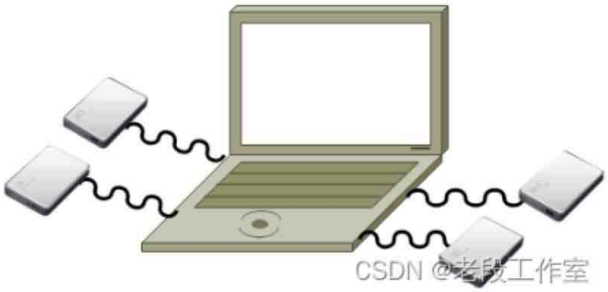
先看下这个非常通俗易懂的例子：

开天辟地时，天地间就有了一宝贝：金士顿移动硬盘。

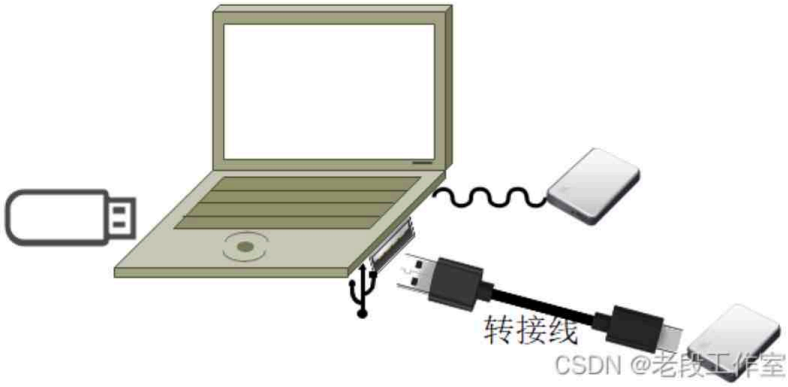
又过了几万年之后，天地间产生了一个新的产品叫做笔记本电脑，但是这个笔记本却缺少存储设备。然后这个笔记本一看，诶？世界上还有这样的一款外接设备叫做移动硬盘，这个笔记本电脑上就自带一根线头连接着这个金士顿硬盘，如下图。



后来又过了很多年，又产生了一些其他品牌的移动硬盘比如闪迪和西数，他们也想让这个笔记本使用他们作为外接设备，当然这个笔记本留了更多的线头连接各种移动硬盘来满足了他们，如下图。

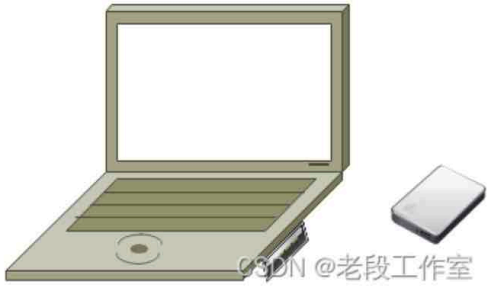


这样笔记本上就裸露出很多的线头以连接了不同品牌的移动硬盘。但是笔记本一看，这样不行啊，我都变成“大蜈蚣”了，咋有那么多线头呢，真丑啊，这个不行，绝对不行。
于是呢，这个笔记本就开发出了一个接口叫做USB接口，然后跟这些移动硬盘说：“同志们，我这开发了一个接口，叫做USB，你们呢要想连接到我这台电脑上作为外接设备，你们就遵从这个USB标准就可以了，如果你们自身接口不满足USB标准的话，你们就造一个满足USB标准的转接头。”



所以呢，除了金士顿之外其他的移动硬盘要不带有USB接口，要不就用一个带有USB接口的数据线来连接到笔记本。
但是这个金士顿移动硬盘偏不，他说我没这个接口，也研究不出来。那咋办呢，没办法，因为这个金士顿太强大了，所以笔记本上仍然留了一个线头连接着金士顿移动硬盘。

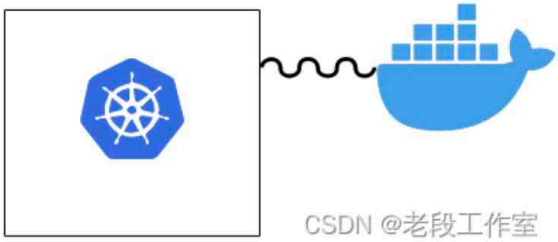
又多了很久，这个笔记本发展的已经很强大了，而且也已经有了很多其他牌子的移动硬盘可用，没必要非金士顿不可了，所以这个笔记本就开始把连接金士顿的那个线头给切掉了。



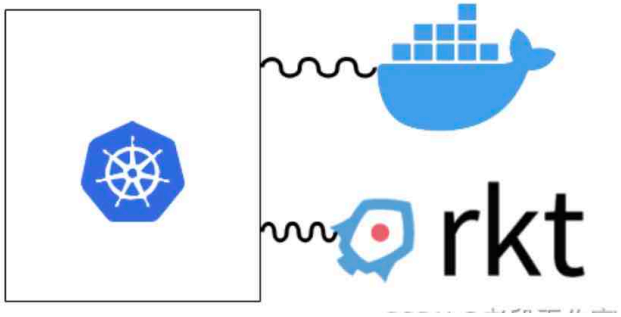
这下完了，金士顿硬盘没法连接到笔记本上了，用不了了。

同理去理解CRI

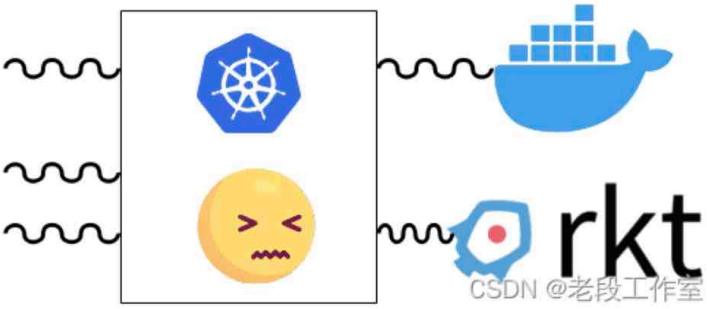
在k8s刚出来的时候，除了 docker 之外并没有几个runtime（运行时，就是管理容器的东西），所以呢在k8s里就内置了代码来适配docker，如下图。



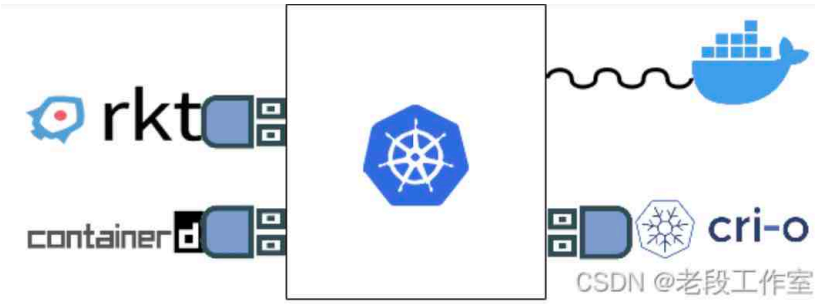
后来除了docker之外又有了其他的runtime，比如rkt。然后呢在 kubernetes 代码里也内置了代码来适配rkt。



后来又有了越来越多的runtime，比如cri-o、containerd等，他们也想集成到kubernetes里让kubernetes以他们作为runtime。



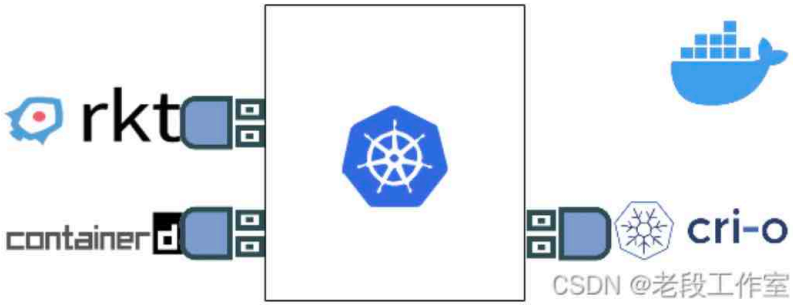
这样一来，kubernetes发现，这样不行啊，这样没完没了的，我这代码太臃肿了不好维护。干脆我就开发一个叫做CRI（容器运行时接口）的标准，并对这些运行时说：“你们也别都想着让把你们整合到我的代码里了，我这开发了一个接口叫做CRI，你们这些runtime只要符合CRI标准就能连接到我了。当然了，如果你们不符合CRI标准的话，你们自己就弄一个转接口转接一下就行了。这个转接口就称之为shim（垫片）吧”。
然后呢有些runtime就开发了符合CRI标准的转接线就是一个shim，叫做CRI-shim。当然有些runtime本身就符合CRI接口了，那么这个runtime就是CRI-shim。



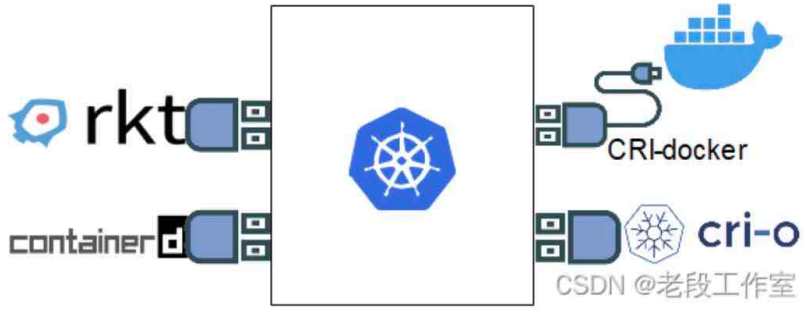
但是呢docker说，我没有CRI接口也开发不出来符合CRI接口的shim。那咋办呢？没办法，因为docker太强大了，所以kubernetes（其实是kubelet）里仍然内置了连接docker的代码叫做（dockershim）。

除了dockershim是内置的，其他的shim一律称之为remote。所以在使用除docker之外的runtime，kubelet都会有一个选项叫做KUBELET_EXTRA_ARGS=–container-runtime=remote。

又过了几年，这时kubernetes已经足够强大了，而且也有了很多其他runtime可选，也不是非docker不可了，所以他必须要一视同仁，所以呢从kubernetes 1.24版本里就把dockershim从kubernetes（其实是kubelet）的代码中移除了。



这下完了，docker不能再作为kubernetes的runtime了。如果要非想让docker作为kubernetes的runtime的话，那么就必须要找一个符合CRI标准的转接口了，这个就是cri-docker，他是一个符合CRI标准的shim（转接口），他可以作为一个转接口连接kubernetes和docker。



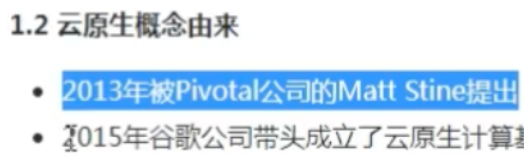
当然了，这也不能说docker不行了，docker仍然是非常有用的，比如构建镜像、搭建仓库等。

视频里的说法：

默认支持docker的垫片shim技术之前是k8s在维护，但是现在k8s把它包出去，不再自己维护了

这么做是为了能和containerd紧密结合，为什么呢，就需要了解containerd的出

现缘由，因为2003年谷歌搞了个容器化应用，到了2013年被docker泄露了，所以就分为了docker和k8s两大阵营，而docker显然没有k8s发展的好



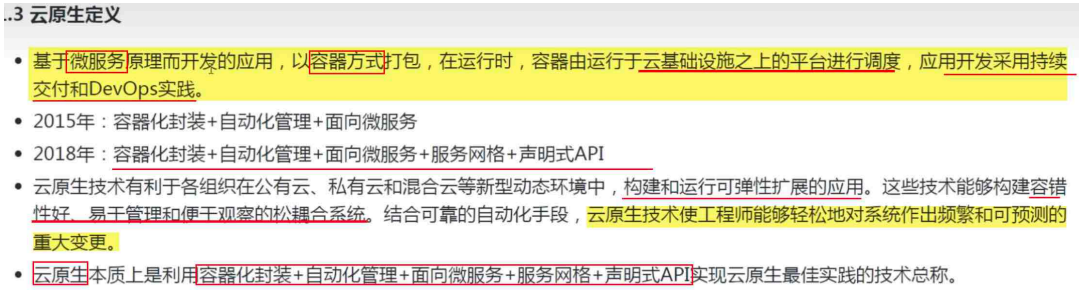
而cncf的目的就是为了能够在容器化应用中一统天下，如果有想抗衡的，肯定就会遭到打压，所以目前docker根本就不在cncf的孵化项目当中，而containerd是在的。

只要是能够进入cncf基金会的项目，都能有300w美金，让他们把产品开发出来，能够给更多人使用，这样逐渐就会类似操作系统windows一样一统天下。**所以k8s就会成为云原生的操作系统，在哪儿都会被使用，像windows，linux一样。**

据统计2023年75%的企业都在使用云原生相关技术

1.3、云原生的定义（如何落地云原生）

就像Devops，不能只是一个概念，要有具体的落地方案



如果早期项目就是微服务思想，上云就很简单了（ps：就像我之前写的微服务demo，最开始并没有涉及docker和k8s，直接是生产者和消费者的jar包，但是是微服务思想，所以后面就很容易制作镜像上云，部署到k8s），而如果不是基于微服务思想开发的，服务之间的关联，服务拆分和制作镜像等就会非常麻烦，我们公司的微服务也是完全从头开始重新开发的，不是原有基础上的

只不过我的demo不是运行在公有云上，而是在本地虚拟机上，ps：其实差不多，只不过公有云更有保障，有更多其他功能，视频里也是用的物理机上的虚拟机，视频里也说了，或者购买阿里云主机，然后再在上面去部署docker，k8s都是可以的

不过目前还没有用到jenkins持续部署。

容器化：就是容器化封装，打包成镜像

自动化管理：指的就是持续交付和DevOps的运用

面向微服务：基于微服务原理开发的应用

服务网格（如Istio）：由于微服务是被拆解成了很多份，那这些服务如何让它们启停，如何对它们之间的调用进行管理（包括熔断限流等），这些都是云原生中遇到的很大的麻烦，所以在2018年后为了让微服务之间的调用更好的管理，用了服务网格技术（ps：之前的微服务demo实现方式是代码里加相关注解）

声明式API：为了在云原生环境中更好的将应用部署起来，并且它的资源能按照我们声明的方式去执行，所以还提到了声明式API，服务发布时如果没有声明就发布了，如果出错了怎么排错呢，它就给了我们一个很好的机会（ps：不是很理解）

其实就是用命令式或资源清单文件来创建应用：

应用部署大体上分为两种执行方式：命令式和声明式。

- 命令式

```
# kubectl run -it busyboxapp --image=busyboxapp:1.28.4
```

- 声明式
  - 使用YAML资源清单文件
  - 在YAML文件中声明要做的操作、需要的配置信息有哪些、用户期
  - 使用声明式API，任何对生产环境、配置都不是操作一条命令来完
    配置管理中进行集中管理，这样有利于在生产环境出现问题时，能
    版本回退、回滚等操作。

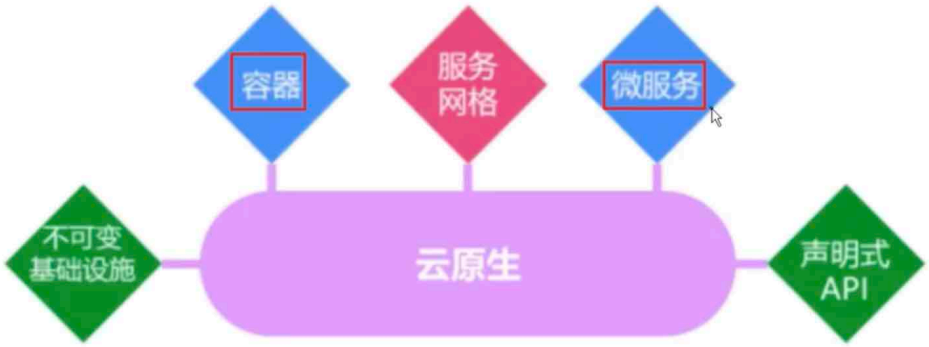
```
# cat nginx.yaml
---
apiVersion: v1
kind: Service
metadata:
```

上面说的排错，就是如果部署出问题了，可以查看yaml和进行修改

云原生就是上面几个技术所实现的最佳实践的总称

就像大数据，并不是数据量很大，海量数据，而应该是收集，处理，得出结果的一种技术总称

下图还有个自动化管理没加进去，就是通过devops相关技术进行自动化管理



容器化技术是微服务最好的载体，并不是物理机和虚拟机是最好的载体

1.3.1、微服务

1.3.1 微服务

微服务的定义：原有单体应用拆分为多个独立自治的组件，每个组件都可以独立设计、开发、测试、部署和运维，这个组件可以单独提供对外服务，我们称之为微服务。

例如：早期的LNMT WEB部署架构，使用微服务后，每一个组件都可以独立自治、运行、扩容、缩容等

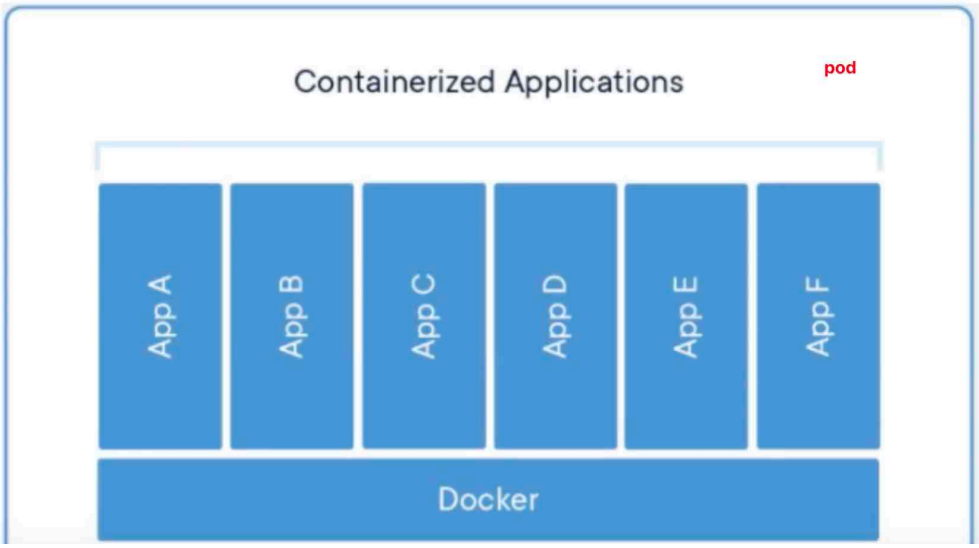
各组件之间可通过轻量的Restful风格接口进行交互和协同

上图说的就跟我的微服务demo一样，消费者和生产者独立部署后，都能分别运行，分别进行扩容缩容

1.3.2、容器化

1.3.2 容器化

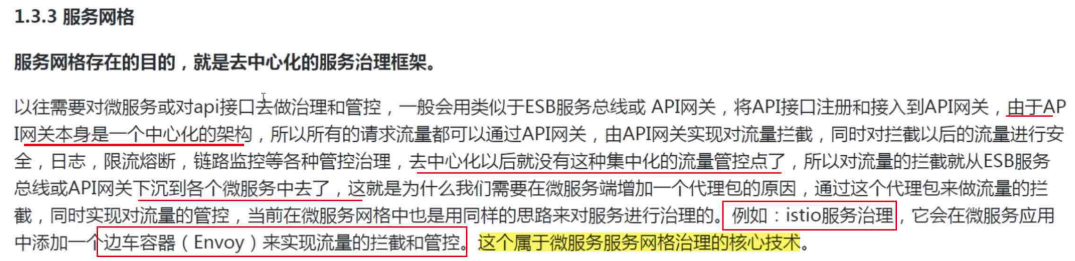
- Docker容器，容器属于it基础设施层概念，是比虚拟机更轻量化的隔离工具，是微服务最佳载体。
- 使用kubernetes的资源调度与容器编排，可以实现Docker容器更优管理，进一步实现其PaaS层能力。



1个pod--1个或多个docker（封装的可以是多个app，就是我们的微服务）

1.3.3、服务网格（框架Istio，核心技术：边车容器Envoy）

如果能理解nginx正反代理，haproxy等就能理解服务网格，因为服务网格本质上也是服务的一种管理方式，代理方式，现在云原生中非常火的就是Istio



1.3.3 服务网格

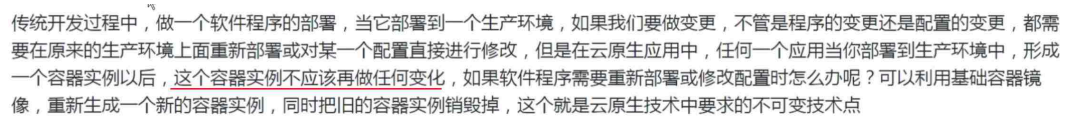
服务网格存在的目的，就是去中心化的服务治理框架。

以往需要对微服务或对api接口去做治理和管控，一般会用类似于ESB服务总线或 API网关，将API接口注册和接入到API网关，由于API网关本身是一个中心化的架构，所以所有的请求流量都可以通过API网关，由API网关实现对流量拦截，同时对拦截以后的流量进行安全，日志，限流熔断，链路监控等各种管控治理，去中心化以后就没有这种集中化的流量管控点了，所以对流量的拦截就从ESB服务总线或API网关下沉到各个微服务中去了，这就是为什么我们需要在微服务端增加一个代理包的原因，通过这个代理包来做流量的拦截，同时实现对流量的管控，当前在微服务网格中也是用同样的思路来对服务进行治理的。例如：istio服务治理，它会在微服务应用中添加一个边车容器（Envoy）来实现流量的拦截和管控。这个属于微服务服务网格治理的核心技术。

带有中心节点架构的服务的问题就是如果崩了，整个应用就没法用了，比如云原生中的存储ceph，如果中心节点被干掉，整个集群就没法用了，但是在学习k8s过程中也有不带中心节点的，比如glusterfs，彼此之间是伙伴关系，挂掉一个不影响其他使用。所以在微服务治理中是不需要中心节点的，要去中心化。
所以会用到istio，把服务进行拆解，拆解成数据平面和控制平面，控制平面出现问题不影响数据平面的使用。

1.3.4、不可变的基础设施（容器化的镜像）

不可变基础设施就是容器化的镜像（为了减少对其他人的干扰，也就是不要改变发布后的版本，最好也就是每次发布版本都用新的，而不是在原有的镜像上进行修改）



传统开发过程中，做一个软件程序的部署，当它部署到一个生产环境，如果我们要做变更，不管是程序的变更还是配置的变更，都需要在原来的生产环境上面重新部署或对某一个配置直接进行修改，但是在云原生应用中，任何一个应用当你部署到生产环境中，形成一个容器实例以后，这个容器实例不应该再做任何变化，如果软件程序需要重新部署或修改配置时怎么办呢？可以利用基础容器镜像，重新生成一个新的容器实例，同时把旧的容器实例销毁掉，这个就是云原生技术中要求的不可变技术点

1.3.5、声明式API（命令式和声明式）

其实就是用命令式或资源清单文件来创建应用：

应用部署大体上分为两种执行方式：命令式和声明式。

- 命令式

```
# kubectl run -it busyboxapp --image=busyboxapp:1.28.4
```

- 声明式
  - 使用YAML资源清单文件
  - 在YAML文件中声明要做的操作、需要的配置信息有哪些、用户期
  - 使用声明式API，任何对生产环境、配置都不是操作一条命令来完
    配置管理中进行集中管理，这样有利于在生产环境出现问题时，能
    版本回退、回滚等操作。

```
# cat nginx.yaml
---
apiVersion: v1
kind: Service
metadata:
```

上面说的排错，就是如果部署出问题了，可以查看yaml和进行修改

对于主机物理资源，一般都需要准备的比较充分，避免我之前经常出现的资源不足问题，视频里说他之前应用时物理机都是32核，64核的，内存可以达到1TB，这个确实厉害。不过一般刚开始上云的时候，头两年都会有75%左右的浪费，随着经验增加，会慢慢好起来，也可以利用hpa，vpa（ps：还没听过）技术来动态扩容缩容或者更牛的**serveless（超微服务，函数级，有用户访问应用时，应用才开启，不访问时就，更加节约资源）**。

补充：云计算下一个十年的技术--超微服务serveless（框架--Knative）

比如Knative（Knative是一款基于Kubernetes的Serverless框架）

这个技术如果用起来就很棒，特别是对开发人员，你每天的关注点不应该在于底层资源调用的问题（特别是多年开发经验的人），这个不应该是开发人员去关注的，虽然让它运行的更快更流畅，更节省资源，确实应该关注底层调用，但是绝大多数公司是不允许有这种非分之想的，没时间让你去研究底层，而是能够快速盈利，上线越快越好，所以你只需要关注你的业务功能实现，不用去关注其他的。

运维也不用担心失业了，它的计算资源，存储等都由你来提供，所以很有必要掌握。

1.3.6、自动化管理（借助于云原生技术，DevOps时代才算真正到来）

1.3.6 DevOps

借助于云原生相关技术，DevOps时代才真正地到来了。

- 实现开发、运维、测试协同合作
- 构建自动化发布管道，实现代码快速部署(测试环境、预发布环境、生产环境等)
- 频繁发布、快速交付、快速反馈、降低发布风险

1.4、云原生最佳实践三个层面

- 服务编排要实现计算资源弹性化
- 服务构建和部署要实现高度自动化
- 事件驱动基础设施标准化

1.4.1、服务编排要实现计算资源弹性化（HPA，VPA，底层计算资源的提供）

已学

1.4.2、服务构建和部署要实现高度自动化（DevOps）

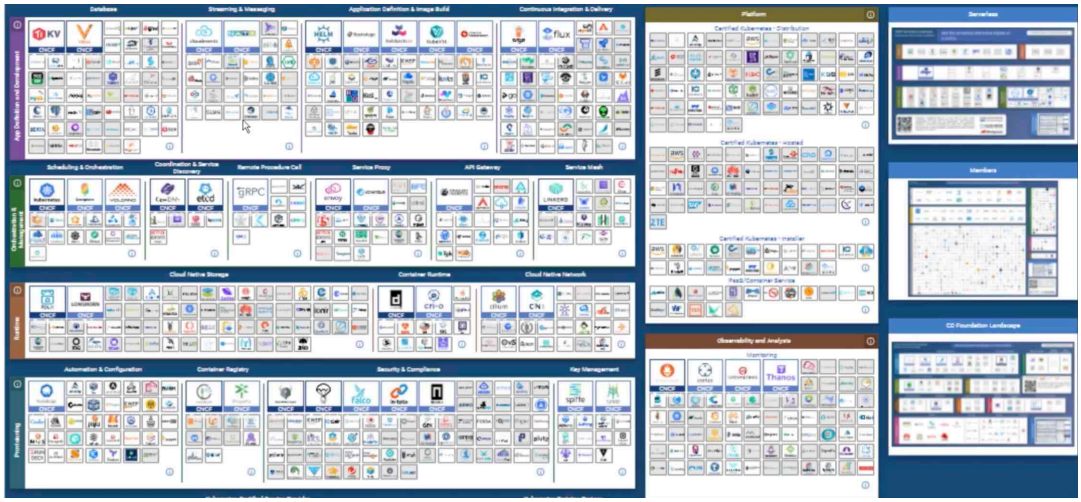
提交代码后，git仓库就能拿到代码，就能在自动化发布平台中发邮件钉钉等通知到你，然后你就可以去审核代码，审核完后发布

ps：这个应该没说完整，还有自动化测试的过程，而且也没说到镜像发布到harbor怎么集成

1.4.3、事件驱动基础设施标准化（serveless）

比如当有用户来访问，这就是一个驱动，一个事件，就会让k8s把相关应用运行起来

1.5、CNCF云原生全景图



学习相关技术即可，不用纠结学完。

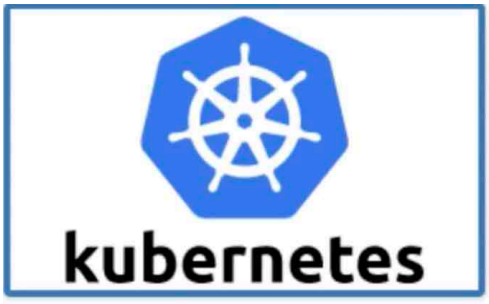
2、云原生应用领域

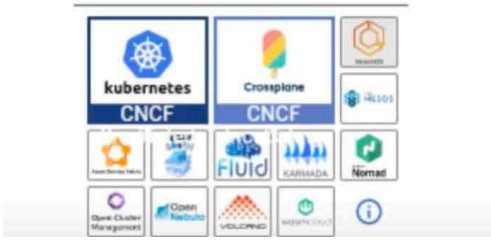
2.1、云原生应用编排及管理

1.6 云原生应用领域

云原生应用生态已覆盖到:大数据、人工智能、边缘计算、区块链等领域。

2.1.1、编排与调度





2.1.2、远程调用

2.1.3、服务代理

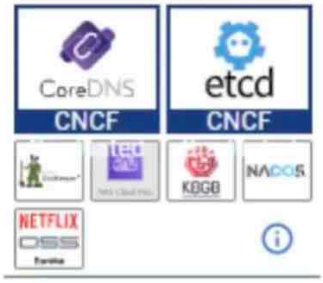
比如nginx，haproxy这类的

envoy是istio的底层，istio是控制平面的东西，envoy是数据平面的东西

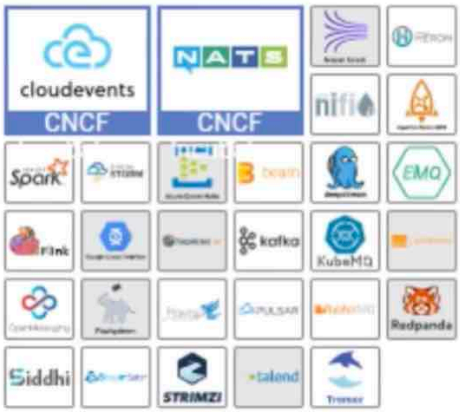
2.1.4、API网关

2.1.5、服务网格

2.1.6、服务发现



2.1.7、消息和流式处理



2.1.8、Serveless

并不是没有服务器，而是让你更少的去了解服务器是什么，不用去关心服务器有几个cpu，内存多少，这不是你关心的事

只需要关心把服务开发出来，运行起来，有用户访问就启动，没用户就停止。

2.1.9、CI/CD

2.1.10、自动化配置

携程的阿波罗也在里面

2.1.11、数据库（MariaDB）

2.1.12、容器镜像仓库（Harbor）

2.1.13、应用定义及镜像制作（HELM）

2.1.14、密钥管理

2.2、云原生底层技术

2.2.1、容器技术（containerd）

这种轻量级或工业级的工具containerd并不适合个人使用，它的功能没有docker完善，而且使用上也没有docker方便，它只是作为一个守护进程在k8s中，比docker更加轻量，节省资源

ps：就是代替docker引擎，但并不直接使用它

3.1 Containerd由来

- 早在2016年3月，Docker 1.11的Docker Engine里就包含了containerd，而现在则是把containerd从Docker Engine里彻底剥离出来，作为一个独立的开源项目独立发展，目标是提供一个更加开放、稳定的容器运行基础设施。和原先包含在Docker Engine里containerd相比，独立的containerd将具有更多的功能，可以涵盖整个容器运行时管理的所有需求。
- containerd并不是直接面向最终用户的，而是主要用于集成到更上层的系统里，比如Swarm, Kubernetes, Mesos等容器编排系统。
- containerd以Daemon的形式运行在系统上，通过暴露底层的gRPC API，上层系统可以通过这些API管理机器上的容器。
- 每个containerd只负责一台机器，Pull镜像，对容器的操作（启动、停止等），网络，存储都是由containerd完成。具体运行容器由runC负责，实际上只要是符合OCI规范的容器都可以支持。
- 对于容器编排服务来说，运行时只需要使用containerd+runC，更加轻量，容易管理。
- 独立之后containerd的特性演进可以和Docker Engine分开，专注容器运行时管理，可以更稳定。

视频里给出的建议是，如果你部署的k8s用的是docker，就不要切换成containerd了，如果你已经部署了containerd，就要忍受在使用k8s过程中，对于底层容器运行时的验证，我理解就是比如使用docker images这样的命令去验证，containerd命令是不一样的。

2 Containerd应用之难

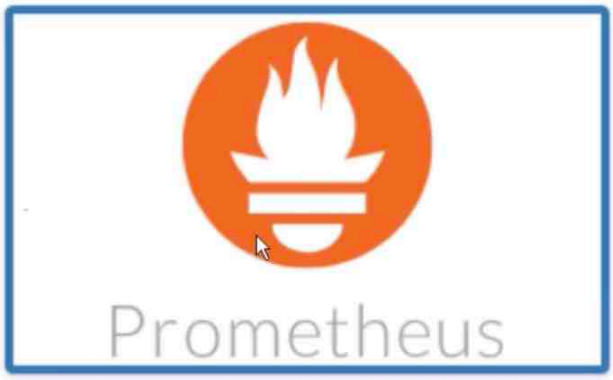
- docker使用docker images命令管理镜像
- 单机containerd使用ctr images命令管理镜像,containerd本身的CLI
- k8s中containerd使用crictl images命令管理镜像,Kubernetes社区的专用CLI工具

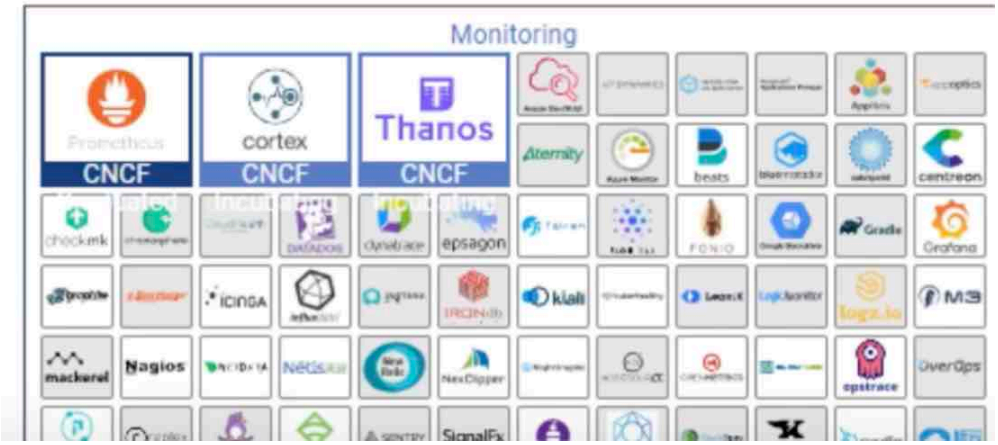
2.2.2、存储技术（ceph，glusterfs）

2.2.3、网络技术

2.3、云原生监控分析

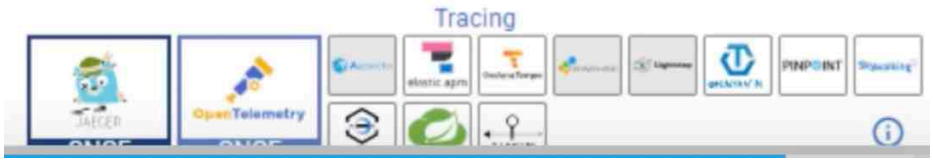
2.3.1、主机状态及服务状态监控





2.3.2、日志收集

2.3.3、全链路状态跟踪



从主机状态，日志，链路，打通任督二脉

2.3.4、云原生安全技术

- 基础设施安全
  - 存储安全、网络安全、计算安全
- 应用安全
  - 应用数据安全、应用配置安全、应用环境安全
- 云原生研发安全
  - 代码托管、代码审计、软件管理、可信测试、可信构建
- 容器生命周期安全
  - 运行时安全、容器构建
- 安全管理
  - 身份认证、访问授权、账号管理、审计日志、密钥管理、监控告警

2、相较于HPA的另外2个概念，VPA,CA

补充一下之前没听过的2个概念，vpa和ca

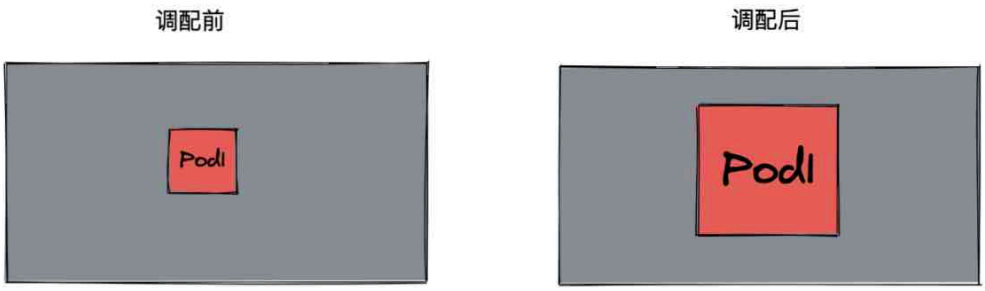
https://www.cnblogs.com/dai-zhe/p/14995444.html#121vpa-%E7%AE%80%E4%BB%8B

- HPA：Pod 水平缩放器
- VPA：Pod 垂直缩放器
- CA：集群自动缩放器

VPA 全称 Vertical Pod Autoscaler，即垂直 Pod 自动扩缩容，它根据容器资源使用率自动设置 CPU 和 内存 的requests，从而允许在节点上进行适

当的调度，以便为每个 Pod 提供适当的资源。
它既可以缩小过度请求资源的容器，也可以根据其使用情况随时提升资源不足的容量。
ps：这个思路确实可以，之前没有接触过，hpa是着重于cpu内存来更改pod数量，vpa反过来了，我猜应该是根据固定pod数量，动态调整内存cpu资源
比如我只有1个pod，但是这个pod占用cpu和内存太高了，比如daemonset，只能有1个pod，就可以通过vpa来调整

- 有些时候无法通过增加 Pod 数来扩容，比如数据库。这时候可以通过 VPA 增加 Pod 的大小，比如调整 Pod 的 CPU 和内存：



- 3、创建VPA
  - 这里先使用updateMode: "Off"模式，这种模式仅获取资源推荐不更新Pod

```
# cat   nginx-vpa-demo.yaml
apiVersion: autoscaling.k8s.io/v1beta2
kind: VerticalPodAutoscaler
metadata:
  name: nginx-vpa
  namespace: vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: nginx
  updatePolicy:
    updateMode: "Off"
  resourcePolicy:
    containerPolicies:
    - containerName: "nginx"
      minAllowed:
        cpu: "250m"
        memory: "100Mi"
      maxAllowed:
        cpu: "2000m"
        memory: "2048Mi"
```

4、查看部署结果

```
[root@k8s-node001 examples]# kubectl  get vpa -n vpa
NAME        AGE
nginx-vpa   2m34s
5、使用describe查看vpa详情，主要关注Container Recommendations

[root@k8s-node001 examples]# kubectl  describe  vpa nginx-vpa   -n vpa
Name:         nginx-vpa
Namespace:    vpa
....略去10000字 哈哈......
  Update Policy:
    Update Mode:  Off
Status:
  Conditions:
    Last Transition Time:  2020-09-28T04:04:25Z
    Status:                True
    Type:                  RecommendationProvided
  Recommendation:
    Container Recommendations:
      Container Name:  nginx
      Lower Bound:
        Cpu:     250m
        Memory:  262144k
      Target:
        Cpu:     250m
        Memory:  262144k
      Uncapped Target:
        Cpu:     25m
        Memory:  262144k
      Upper Bound:
        Cpu:     803m
        Memory:  840190575
Events:          <none>
```

```
Lower Bound:                下限值
Target:                     推荐值
Upper Bound:                上限值
Uncapped Target:            如果没有为VPA提供最小或最大边界，则表示目标利用率
上述结果表明，推荐的 Pod 的 CPU 请求为 25m，推荐的内存请求为 262144k 字节。
```

补充：Linux之ab命令

ab是apachebench命令的缩写，ab是apache自带的压力测试工具。ab非常实用，它不仅可以对apache服务器进行网站访问压力测试，也可

以对或其它类型的服务器进行压力测试。比如nginx、tomcat、IIS等。

ab的原理：ab命令会创建多个并发访问线程，模拟多个访问者同时对某一URL地址进行访问。它的测试目标是基于URL的，因此，它既可以用来测试apache的负载压力，也可以测试nginx、lighthttp、tomcat、IIS等其它Web服务器的压力。

ab命令对发出负载的计算机要求很低，它既不会占用很高CPU，也不会占用很多内存。但却会给目标服务器造成巨大的负载，其原理类似CC攻击。自己测试使用也需要注意，否则一次上太多的负载。可能造成目标服务器资源耗完，严重时甚至导致死机。

语法：

> `ab -n 1000 -c 10 https://json.im`

- -n 1000表示请求总数为1000
- -c 10表示并发用户数为10

- 4、对nginx进行压测，执行压测命令

```
# ab -c 100 -n 10000000 http://192.168.127.124:32621/
  This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
  Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
  Licensed to The Apache Software Foundation, http://www.apache.org/

  Benchmarking 192.168.127.124 (be patient)
  Completed 1000000 requests
  Completed 2000000 requests
  Completed 3000000 requests
```

- 5、稍后再观察VPA Recommendation变化

```
# kubectl  describe  vpa nginx-vpa   -n vpa |tail -n 20
  Conditions:
    Last Transition Time:  2021-06-28T04:04:25Z
    Status:                True
    Type:                  RecommendationProvided
  Recommendation:
    Container Recommendations:
      Container Name:  nginx
      Lower Bound:
        Cpu:     250m
        Memory:  262144k
      Target:
        Cpu:     476m
        Memory:  262144k
      Uncapped Target:
        Cpu:     476m
        Memory:  262144k
      Upper Bound:
        Cpu:     2
        Memory:  387578728
Events:          <none>
```

- 从输出信息可以看出，VPA对Pod给出了推荐值：Cpu: 476m，因为我们这里设置了updateMode: "Off"，所以不会更新Pod;

1.2.2.4、updateMode: "Auto"（此模式当目前运行的pod的资源达不到VPA的推荐值，就会执行pod驱逐，重新部署新的足够资源的服务）

- 1、把updateMode: "Auto"，看看VPA会有什么动作
  - 并且把resources改为：memory: 50Mi，cpu: 100m

```
# kubectl  apply -f nginx-vpa.yaml
  deployment.apps/nginx created

# cat nginx-vpa.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app: nginx
  name: nginx
  namespace: vpa
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - image: nginx
        name: nginx
        resources:
          requests:
            cpu: 100m
            memory: 50Mi
```

```
# kubectl  get po  -n vpa
  NAME                     READY   STATUS    RESTARTS   AGE
  nginx-7ff65f974c-f4vgl   1/1     Running   0          114s
  nginx-7ff65f974c-v9ccx   1/1     Running   0          114s
```

- 2、再次部署vpa,这里VPA部署文件nginx-vpa-demo.yaml只改了 `updateMode: "Auto"` 和 `name: nginx-vpa-2`

```
# cat  nginx-vpa-demo.yaml
apiVersion: autoscaling.k8s.io/v1beta2
kind: VerticalPodAutoscaler
metadata:
  name: nginx-vpa-2
  namespace: vpa
spec:
  targetRef:
    apiVersion: "apps/v1"
    kind: Deployment
    name: nginx
  updatePolicy:
    updateMode: "Auto"
  resourcePolicy:
    containerPolicies:
    - containerName: "nginx"
      minAllowed:
        cpu: "250m"
        memory: "100Mi"
      maxAllowed:
        cpu: "2000m"
        memory: "2048Mi"

# kubectl apply -f nginx-vpa-demo.yaml
  verticalpodautoscaler.autoscaling.k8s.io/nginx-vpa created

# kubectl  get vpa -n vpa
  NAME        AGE
  nginx-vpa-2   9s
```

- 3、再次压测

```
# ab -c 1000 -n 100000000 http://192.168.127.124:32621/
```

- 4、稍后使用describe查看vpa详情，同样只关注Container Recommendations

```
# kubectl  describe  vpa nginx-vpa-2    -n vpa |tail -n 30
      Min Allowed:
        Cpu:     250m
        Memory:  100Mi
  Target Ref:
    API Version:  apps/v1
    Kind:         Deployment
    Name:         nginx
  Update Policy:
    Update Mode:  Auto
Status:
  Conditions:
    Last Transition Time:  2021-06-28T04:48:25Z
    Status:                True
    Type:                  RecommendationProvided
  Recommendation:
    Container Recommendations:
      Container Name:  nginx
      Lower Bound:
        Cpu:     250m
        Memory:  262144k
      Target:
        Cpu:     476m
        Memory:  262144k
      Uncapped Target:
        Cpu:     476m
        Memory:  262144k
      Upper Bound:
        Cpu:     2
        Memory:  262144k
Events:          <none>
```

- Target变成了Cpu: 587m ， Memory: 262144k

补充：什么是VPA？HPA?

另外一个文档：https://www.shouxicto.com/article/897.html

为了解决业务服务负载时刻存在的巨大波动和资源实际使用与预估之间差距，就有了针对业务本身的“扩缩容”解决方案： Horizontal Pod Autoscaler（HPA）和 Vertical Pod Autoscaler（VPA）。

为了充分利用集群现有资源优化成本，当一个资源占用已经很大的业务需

要扩容时，其实可以先尝试优化**业务负载自身的资源需求配置**（request与实际的差距），只有当集群的资源池确实已经无法满足负载的实际的资源需求时，再调整资源池的总量保证资源的可用性，这样可以将资源用到极致。

所以总的来说弹性伸缩应该包括：

1. Cluster-Autoscale: 集群容量（node数量）自动伸缩，跟自动化部署相关的，依赖iaas的弹性伸缩，主要用于虚拟机容器集群

2. Vertical Pod Autoscaler: 工作负载Pod垂直（资源配置）自动伸缩，如自动计算或调整deployment的Pod模板limit/request，**依赖业务历史负载指标**

3. Horizontal-Pod-Autoscaler: 工作负载Pod水平自动伸缩，如自动scale deployment的replicas，**依赖业务实时负载指标**

VPA和HPA都是从业务负载角度从发的优化

VPA是解决资源配额（Pod的CPU、内存的limit/request）评估不准的问题。

HPA则要解决的是业务负载压力波动很大，需要人工根据监控报警来不断调整副本数的问题。

有了HPA后，被关联上HPA的deployment，后续副本数修改就不用人工管理，HPA controller将会根据业务忙闲情况自动帮你动态调整。当然还有一种固定策略的特殊HPA: cronHPA，也就是直接按照cron的格式设定扩容和缩容时间及对应副本数，可以简单理解为定时伸缩，这种不需要动态识别业务繁忙度，属于静态HPA，适用于业务流量变化有固定时间周期规律的情况。

疑问：为什么需要vpa的方式扩容?

其实也很好理解，因为deamonset没法通过增加副本数的方式来扩容，只能有1个，所以才有了vpa的方式

补充：cadvisor与cgroup

cadvisor 获取指标时，实际上也只是个转发者，它的数据来自于 **cgroup**

文件。

在Linux里，对进程进程分组，比如Session group、process group等，后来需要追踪一组进程的内存和IO使用情况，出现了cgroup，用来统一对进程进行分组，并在分组的基础上对进程进行监控和资源控制管理等。

Cgroup可以指整个cgrop技术，也可以指一个具体的进程组。

Cgroup是Linux下一种将进程按组管理的机制，在用户层面看，Cgroup技术就是可以把系统中的所有进程组织成一颗独立的树，每颗树都包含系统所有进程，树的每个节点就是一个进程组，而每颗树又和一个或者多个subsystem关联，树的作用是对进程分组，而subsystem作用是对这些组进行操作。

3、云原生最佳实践3个方面

- 服务编排要实现计算资源弹性化
- 服务构建和部署要实现高度自动化
- 事件驱动基础设施标准化

学习docker和k8s有一个终极诀窍：学完即忘

如果你已经学会了k8s，就不会再想k8s怎么部署了。