Kubernetes和webwasmly相关资料调研分析

一、Kubernetes介绍

Kubernetes 也称为 K8s，是用于自动部署、扩缩和管理容器化应用程序的开源系统，服务器编排工具。

K8s特点

负载均衡: 如果进入容器的流量很大，K8s可以负载均衡并分配网络流量，从而使部署稳定

自动部署和回滚:可以使用 K8s描述容器的所需状态， 它可以以受控的速率将实际状态更改为期望状态

高可用性和自我修复:K8s 将重新启动失败的容器、替换容器、杀死不响应用户定义的运行状况检查的容器， 并且在准备好服务之前不将其通告给客户端

自动完成装箱计算:可以告诉 K8s 每个容器需要多少 CPU 和内存。 K8s 可以将这些容器按实际情况调度到你的节点上，以最佳方式利用你的资源

K8s资源对象

Node 与 Pod:Node 可以是虚拟机，物理机器，Docker等。Pod 是对 Container 的抽象

Volume：用于数据持久化，重启时数据存储与恢复，可以是本地存储，可以是远端存储

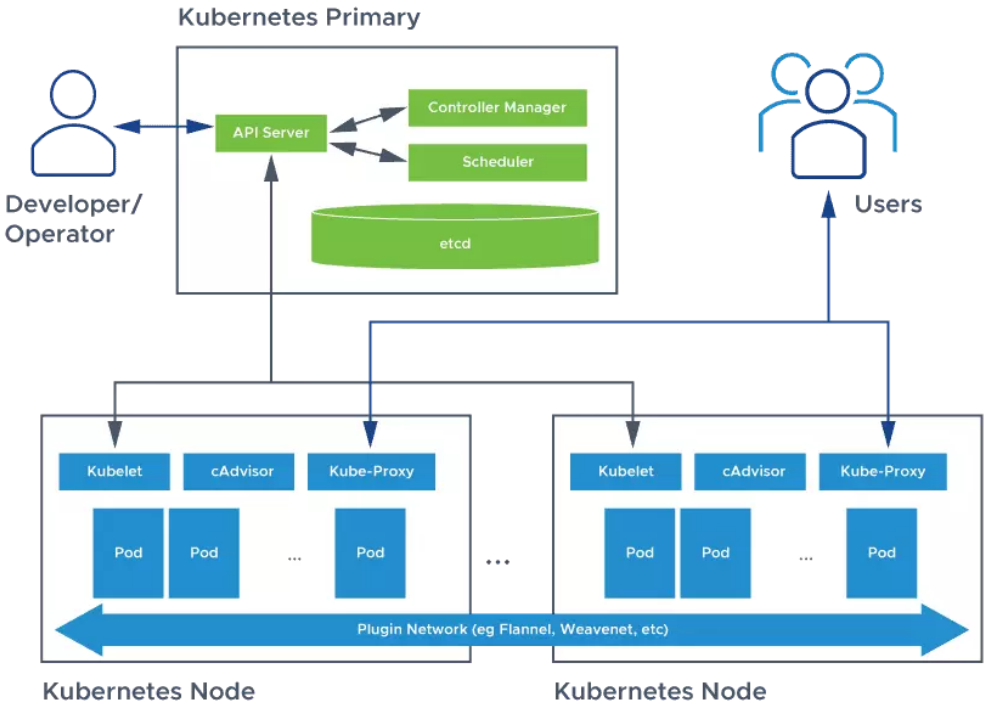
Service 与 Ingress:负责Pods间通讯，相对静止，负载均衡，选择合适的Pod进行服务

Replica：Node 与 Pod 的复制。Kubernetes 高可用性的体现

ConfigMap 与 Secret：外部配置，如url，环境变量。避免小改动带来的更新工作

Deployment：用于指导创建Pod等组件。管理主要的交互介质

K8s架构



Api Server：向控制客户端程序提供接口，处理控制请求，控制Kubernetes

Scheduler：负责调度和创建Pod。合理的选择合理的位置创建Pod。实际创建工作由Kubelet完成

Controller Manager：监听集群的状态。当监听到Pod宕机时调用Scheduler重新创建Pod

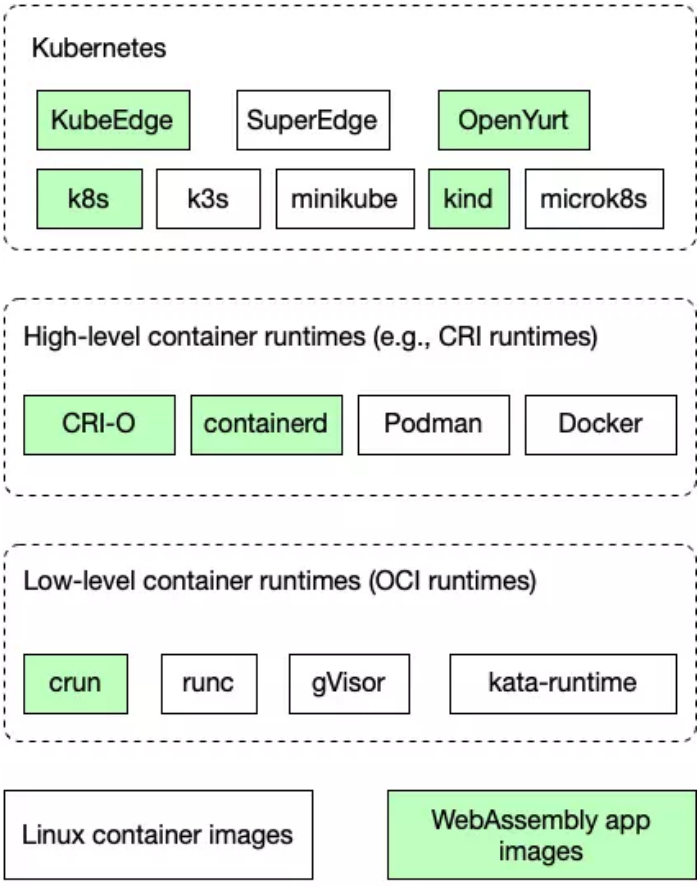
Etcd:一个Key-Value数据库，仅存储和更新集群的状态。如哪些节点可用，集群状态变化等。不存储具体应用的数据

Kubelet:实际调度和管理 Pod 和 Container的程序。同时与 Container 和 Node 交互，负责资源调度等

Kube Proxy：负责定向 Service 的请求到 Pod。具有负载均衡的功能，合理选择服务的Pod

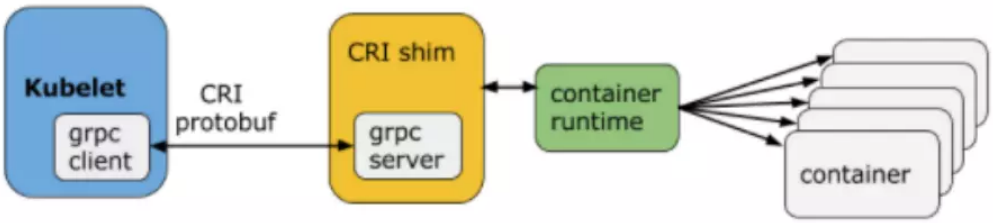
cAdvisor:分析运行中的 Docker 容器的资源占用以及性能

K8s在容器生态中的层次



The container ecosystem

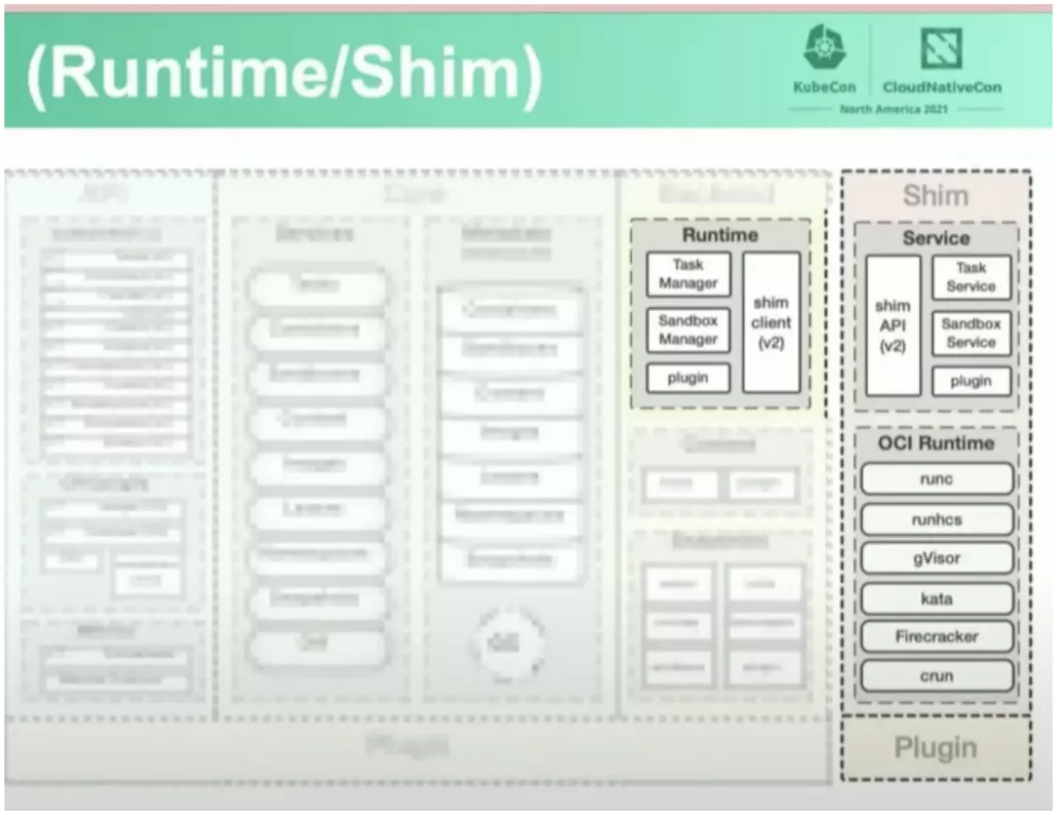
在整个容器生态中，K8s属于最高层，用户通过kubectl启动K8s服务，kubelet象CRI发送服务请求。容器运行时插件（Container Runtime Interface，简称 CRI），是 Kubernetes v1.5 引入的容器运行时接口，将 Kubelet 与容器运行时解，使镜像管理和容器管理分离到不同的服务。CRI调用containerd、CRI-O等高层容器运行时管理容器镜像及底层容器运行时，在containerd中，containerd会调用不同底层容器运行时的shim，这些shim会调用底层容器运行时，如kata、runc，管理容器的实际运行。



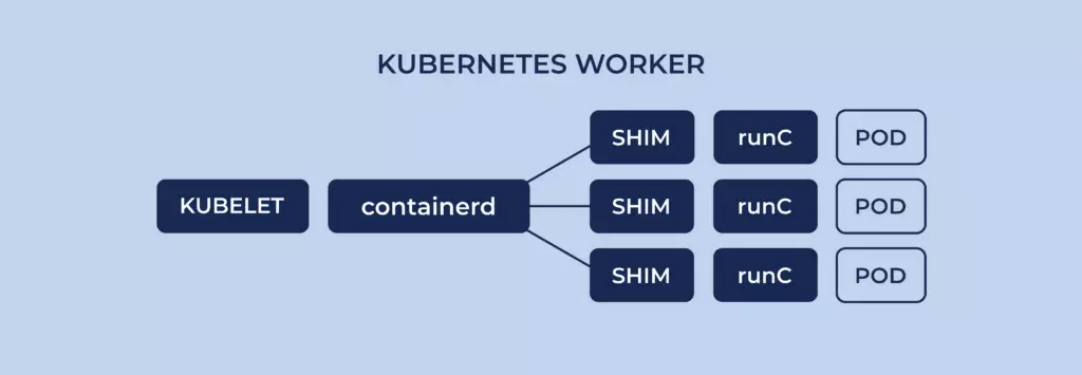
Kubelet通过调用统一的CRI接口，在不同的高层运行时CRI-shim的支持下，实现容器的运行与管理，在containerd中，CRI作为一个plugin(插件)，为kubelet提供image-service和runtime-service

```
$ ctr plugins ls
TYPE                            ID                    PLATFORMS      STATUS
io.containerd.content.v1        content               -              ok
io.containerd.snapshotter.v1    btrfs                 linux/amd64    ok
io.containerd.snapshotter.v1    aufs                  linux/amd64    error
io.containerd.snapshotter.v1    native                linux/amd64    ok
io.containerd.snapshotter.v1    overlayfs             linux/amd64    ok
io.containerd.snapshotter.v1    zfs                   linux/amd64    error
io.containerd.metadata.v1       bolt                  -              ok
io.containerd.differ.v1         walking               linux/amd64    ok
io.containerd.gc.v1             scheduler             -              ok
io.containerd.service.v1        containers-service    -              ok
io.containerd.service.v1        content-service       -              ok
io.containerd.service.v1        diff-service          -              ok
io.containerd.service.v1        images-service        -              ok
io.containerd.service.v1        leases-service        -              ok
io.containerd.service.v1        namespaces-service    -              ok
io.containerd.service.v1        snapshots-service     -              ok
io.containerd.runtime.v1        linux                 linux/amd64    ok
io.containerd.runtime.v2        task                  linux/amd64    ok
io.containerd.monitor.v1        cgroups               linux/amd64    ok
io.containerd.service.v1        tasks-service         -              ok
io.containerd.internal.v1       restart               -              ok
io.containerd.grpc.v1           containers            -              ok
io.containerd.grpc.v1           content               -              ok
io.containerd.grpc.v1           diff                  -              ok
io.containerd.grpc.v1           events                -              ok
io.containerd.grpc.v1           healthcheck           -              ok
io.containerd.grpc.v1           images                -              ok
io.containerd.grpc.v1           leases                -              ok
io.containerd.grpc.v1           namespaces            -              ok
io.containerd.grpc.v1           snapshots             -              ok
io.containerd.grpc.v1           tasks                 -              ok
io.containerd.grpc.v1           version               -              ok
io.containerd.grpc.v1           cri                   linux/amd64    ok
```

Containerd与底层运行时交互



containerd下的shim是充当containerd和OCI Runtime之间的中间件，用来组装OCI Runtime命令的参数，负责容器中进程的启动。



在runc中，真正启动容器是通过 containerd-shim 去调用 runc 来启动容器的，runc 启动完容器后本身会直接退出，containerd-shim 则会成为容器进程的父进程

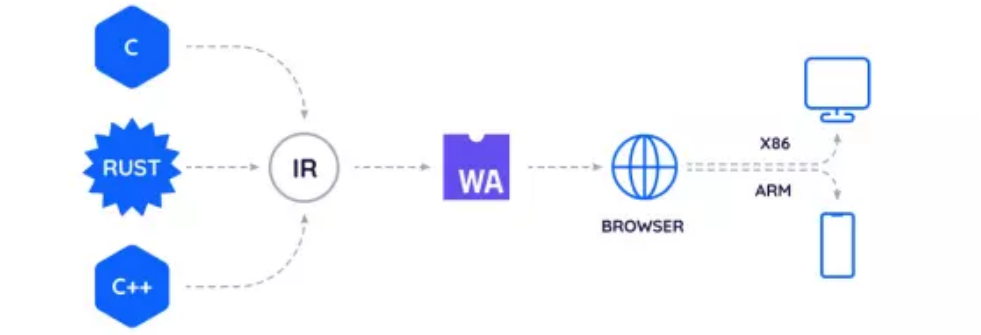
runwasi

该项目致力于通过 containerd （即通过 ctr ） CRI 插件编写,使用 wasmruntime作为底层容器运行时的rust库。

Containerd Wasm Shims是使用runwasi库作实现wasm-shim的一个项目，目前该项目包含spin-shim和slightly-shim。但其shim的功能还不足以支撑wasm容器的正常使用。

二、Webassembly介绍

WebAssembly(简称Wasm）是一个开放标准，其原本的目的是在浏览器中执行二进制代码，这个代码不是手写的，而是作为其它语言的编译目标生成的，其最终在wasm虚拟机中解析执行。



Wasm的优点:

速度快:这个快主要是相对于javascript和其他动态语言，使web应用运行变快，wasm是一个底层的(low-level)、类似汇编的语言，它会产生一个紧凑的二进制格式的文件，这些条件让它有可能提供一个near-native的性能。near-native指达到将近在本地执行的速度，使用静态类型，编译时优化，体积更小，意味着从服务器拉取更快。

- Wasm is 1.15-1.67 times faster than JavaScript on Google Chrome on a desktop.
- Wasm is 1.95-11.71 times faster than JavaScript on Firefox on a desktop.
- Wasm is 1.02-1.38 times faster than JavaScript on Safari on a desktop.
- Wasm is 1.25-2.59 times faster than JavaScript on Chrome on a Moto G5 Plus smartphone.
- Wasm is 1.84-16.11 times faster than JavaScript on Firefox on a Moto G5 Plus smartphone.
- Wasm is 1.07-1.23 times faster than JavaScript on Safari on an iPhone 6s.

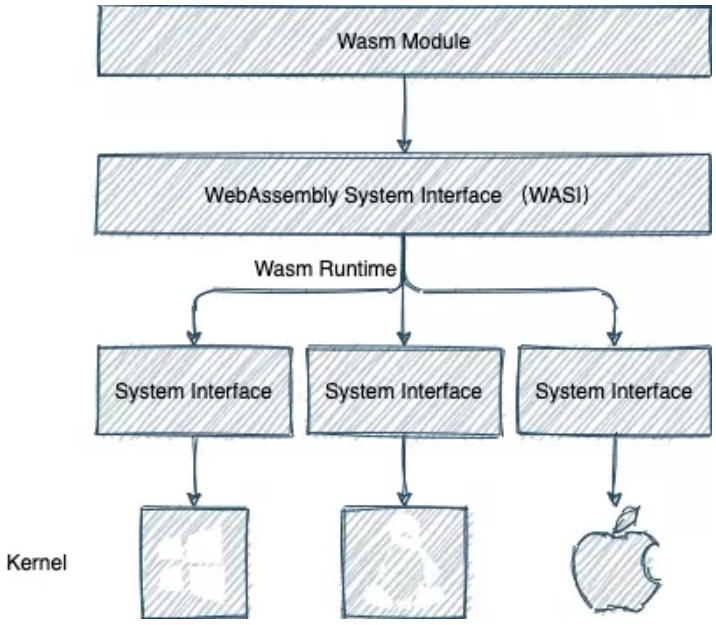
一次编写，到处运行：具有二进制兼容性的代码可以编写大多数应用程序以在本机或浏览器中运行。

安全:WebAssembly 的安全模型是完整沙箱，这表明系统调用可控，内存安全，以及更少的攻击面。

WASI

WASI是一个新的API体系, 由Wasmtime项目设计, 目的是为WASM设计一套引擎无关(engine-indepent),面向非Web系统(non-Web system-oriented)的API标准。

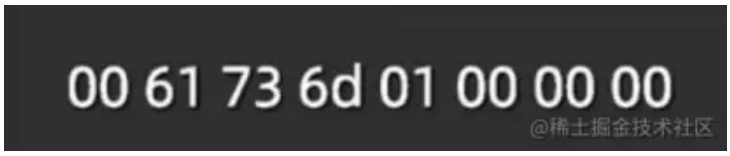
wasm在编译的时候不是面向操作系统，它不知道它将要运行在哪个系统上，WASI作为一个抽象接口层，在系统调用前实现被wasm二进制调用，wasm runtime类似浏览器，wasi类似浏览器提供的api提供虚拟系统调用 / 系统调用的wrapper，wasm runtime将沙盒环境和系统资源隔离开来，并调用系统资源执行程序



在WASI基础上实现的wasmruntime,需要考虑的是可移植性和安全性问题，实现在多平台上.wasm程序的正常执行。

wasm实现原理

wasm是一种新的字节码格式，规定了数据对应的二进制格式。



与字节码格式相对应的人类可读文件格式 wat，方便开发人员阅读，类似S-expression与传统汇编格式的结合。

特点:

wasm被设计成抽象虚拟机的字节码格式

字节码的指令、语法结构不依赖具体虚拟机体系类型（x86、arm），目前有wasm32、wasm64两种目标格式

字节码需要被具体虚拟机程序解析和执行

chrome的v8引擎通过wasm规范来解析字节码，将其转换成宿主平台的机器码，然后执行

wasm的性能

首先，wasm并不是真正的汇编码，wasm code也需要解释器运行

WASM == 汇编级性能?

这显然不对，WASM 里的 Assembly 并不意味着真正的汇编码，而只是种新约定的字节码，也是需要**解释器**运行的。这种解释器肯定比 JS 解释器快得多，但自然也达不到真正的原生机器码水平。一个可供参考的数据指标，是 JS 上了 JIT 后整体性能大致是机器码 1/20 的水平，而 WASM 则可以跑到机器码 1/3 的量级（视场景不同很不好说，仅供参考）。**相当于即便你写的是 C++ 和 Rust 级的语言，得到的其实也只是 Java 和 C# 级的性能。**这也可以解释为什么 WASM 并不能在所有应用场景都显示出压倒性的性能优势：**只要你懂得如何让 JS 引擎走在**[illegible]**在浏览器里，JS 就敢和 Rust 五五开。**

wasm的应用场景

- 迁移成熟的工具
  - 如Google Earth，Unity3D, CAD 等等，包括轻游戏
- 多媒体
  - 视频/直播编解码；
  - 在线图像/视频处理应用；
  - 更轻量、消耗更低的CPU和内存
- 基于边缘计算的机器/深度学习：MXNet.js；
- 高性能 Web 游戏：Unity、Unreal、Ammo.js 等游戏库和引擎；
- 区块链 Ethereum 核心；
- 前端框架：sharpen、asm-dom、yew；
- 微内核
  - 一个完整的集成图形界面、多线程、网络、C标准库的**WASM虚拟机执行层微内核**只有468KB，系统冷启动时间和资源使用率是一个非常亮眼的数据
- IOT：wasmachine；
  - 设备要跑wasm应用，只要一个能解析和运行它字节码的虚拟机即可，而这个虚拟机很轻量

wasmruntime

目前调研的wasmruntime主要有wasmer、wasmedge、wasmtime。
根据之前综合测试的结果，wasmedge的性能较优。

wasm容器

考虑使用wasm技术作为容器时，相较于containerd，wasm的优势体现在:
安全。 即便当前的container在安全领域有很多的防护手段，但是机制上，应用进程是可以直接访问到host主机的内核，因为container是通过cgroup实现的一种轻量级虚拟化技术，并不是完整内核虚拟化。虽然有 security context 机制，但是攻击面还是很大。当前也有一种使用完整内核虚拟化的趋势，包括 gVisor, Firecracker(AWS Lambda, AWS Farget产品都使用此虚拟化技术), Kata。这表明安全是很重要的一个考量因素。 WebAssembly 的安全模型是完整沙箱，这表明系统调用可控，内存安全，以及更少的攻击面。
包大小。 容器镜像包大小一直是一个诟病点，由此出现了很多技术点以及最佳实践用于减少镜像包大小。这是因为机制上，容器镜像要求将内内核之外的、应用程序依赖的lib库都打包到镜像中，其目的是实现可移植性。 WebAssembly的交付包只包含应用程序本身（实际情况还会有15%的压缩）。 包大小减少后可以实现更短时间启动应用程序，如缩短函数计算中的"冷启动"时间。Fastly的Lucent叠加AOT技术，冷启动时间约 50微秒；Cloudflare 使用 V8引擎，冷启动时间 5毫秒。这对Serverless计算场景是非常重要的体验提升。

三、K8s与wasm的交互

Docker的WebAssembly项目

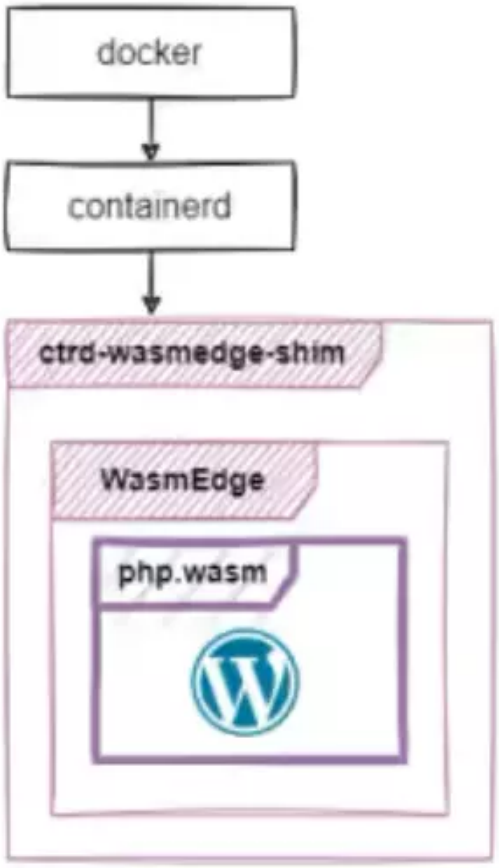
从 Docker Desktop 4.15 开始实现对 WebAssembly 支持。它是通过一个 containerd shim 实现的，该 shim 可以使用WasmEdge的 Wasm 运行时来运行 Wasm 应用程序。使得可以在 WasmEdge 运行时中运行 Wasm 应用程序，而不是典型的 Windows 或 Linux 容器，它们会运行容器映像中二进制文件的单独进程，模拟容器。

当前Docker+Wasm 的全新技术预览版将支持以下三种新的运行时：

- Fermyon 的 spin：https://www.fermyon.com/spin

- Deislabs 的 slight：https://deislabs.github.io/spiderlightning/
- 字节码联盟的 wasmtime：https://wasmtime.dev/

当运行一个wasm应用时，其运行的调用链如图:



在docker中运行wasm容器时，需添加runtime参数

```
docker run --rm -d \  传统
   -p 8083:8080 -v $(pwd)/images/php/docroot:/docroot \
   php:7.4.32-cli \
   -S 0.0.0.0:8080 -t /docroot

docker run --rm -d \  wasm
   --runtime=io.containerd.wasmedge.v1 \
   -p 8082:8080 -v $(pwd)/images/php/docroot:/docroot \
   ghcr.io/vmware-labs/php-wasm:7.4.32-cli-aot
   -S 0.0.0.0:8080 -t /docroot
```

优点:

镜像大小：wasm镜像较传统lxc容器镜像小,对于 Wasm，只需要在容器内添加可执行二进制文件，而对于传统容器，我们仍然需要来自运行二进制文件的操作系统的一些基本库和文件。这种大小差异对于第一次拉取图像的速度以及图像在本地存储库中占用的空间非常有益。

webassembly的天然优势:快速、安全、可移植、高效.

缺点:

1、与docker绑定，不能从k8s直接对containerd进行调用,增加了容器生成环节。

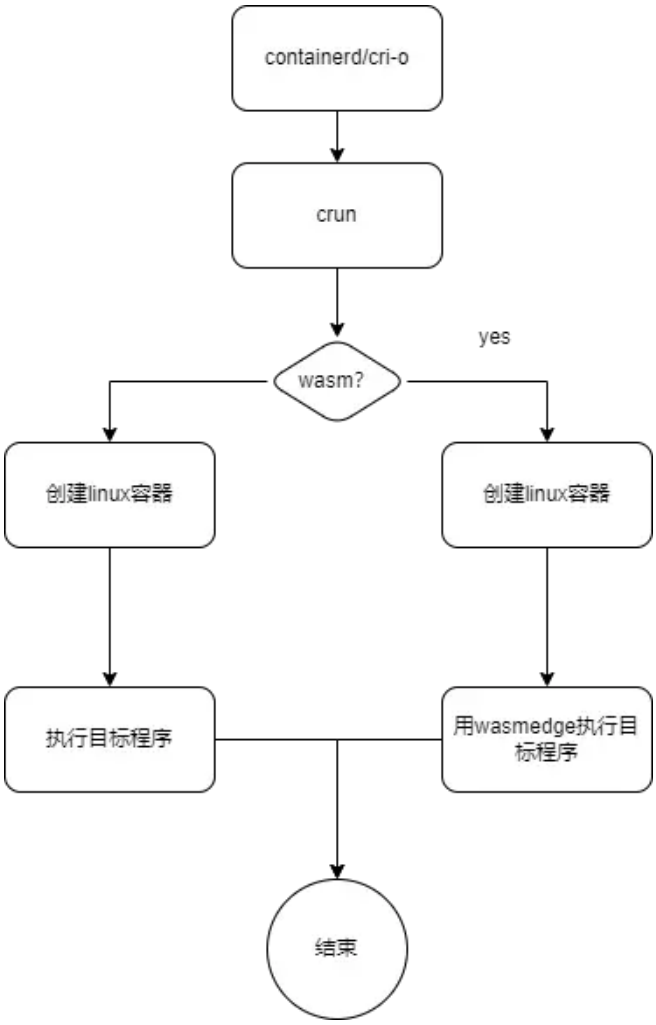
2、wasmruntime目前还只能支持32位应用。

WasmEdge提供的K8s支持方案

wasmedge支持在CRI-O和containerd为高层运行时,crun作为底层运行对WasmEdge提供了官方支持，但wasm应用的执行是在runc的linux容器中完成的。

CRI (high level) runtime	OCI (low level) runtime
CRI-O	crun + WasmEdge
containerd	crun + WasmEdge

其容器的创建流程如下



在创建wasm容器时，其与创建普通linux容器的步骤是一致的，但是使用的wasm镜像中只包含wasm应用文件与配置文件，容器启动时直接通过wasmedge执行wasm文件，管理方式与普通容器一致。

优点:
方便上层运行时对容器的管理，提供了对wasm应用的兼容。
wasm镜像容量较小。
缺点:
在创建wasm容器时，仍需要执行传统linux容器的准备步骤，没有体现出wasm性能上的优势，绑定了crun作为底层运行时，而不是wasmeruntime直接作为底层运行时，提高了冗余的创建时间。

Containerd Wasm Shims项目

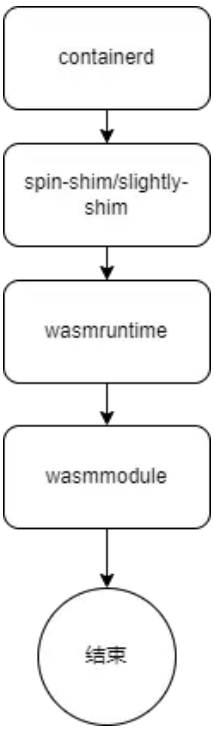
该项目旨提供 Wasm / WASI 容器的 containerd shim 实现。shim的编写使用runwasi作为库。向 Kubernetes 添加运行时类并在节点上部署 Wasm程序。Wasm pod 和deployment可以像传统容器一样运行。

目前该项目提供两个shim:

Spin shim：Spin 是一个开源框架，用于使用 WebAssembly 构建和运行快速、安全且可组合的云微服务。

Slight (SpiderLightning) shim：与 Spin 非常相似，用于使用 WebAssembly 构建和运行快速、安全且可组合的云微服务。

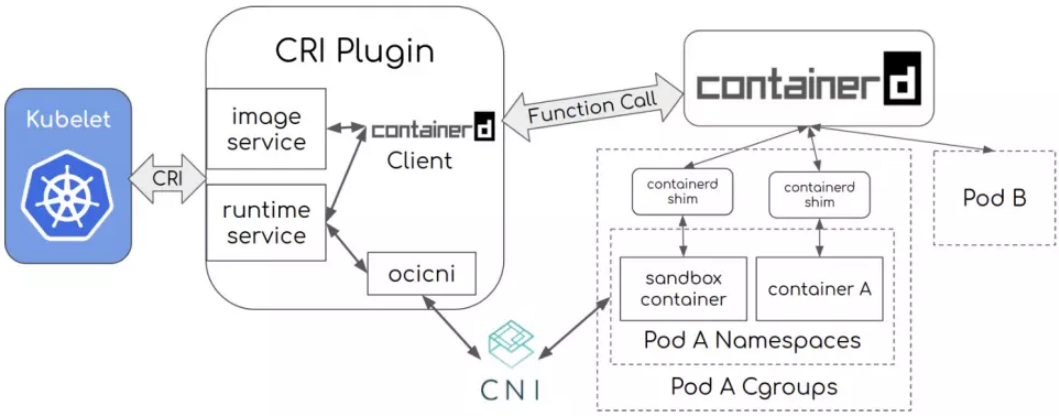
shim直接与wasmruntime进行对接，wasmruntime直接作为底层容器运行时运行wasm应用。
containerd通过cri-plugin直接与k8s进行交互。
其容器创建流程如下:



优点:
容器创建流程短，wasmruntime直接作为底层容器运行时，较能体现出wasm性能的优势。
缺点:
K8s与containerd交互的CRI插件仍保留了较多传统linux容器的准备工作，这对于wasm容器是冗余的。目前只支持spin/slightly微服务框架。

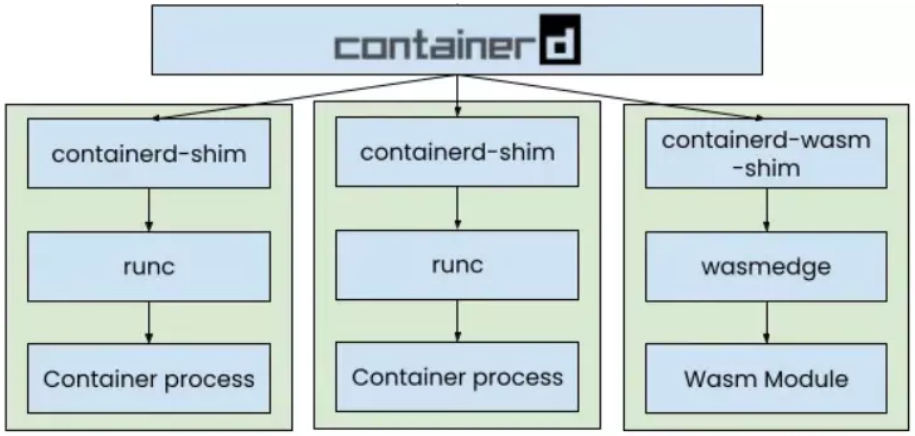
K8s与containerd的改造

以containerd为例:



CRI:Containerd 内置的 CRI 插件实现了 Kubelet CRI 接口中的 Image Service 和 Runtime Service，通过内部接口管理容器和 镜像，并通过 CNI 插件给 Pod 配置网络。

鉴于wasm运行方式与lxc(linux container)容器不同，在CRI Plugin处，可以对runtime-service和image-service进行优化调整，缩短wasm容器部署的准备时间。



wasmruntime与传统容器运行时runc处于类似的层次，但是wasmruntime不具备runc对于容器的管理功能，需要在containerd-wasm-shim中实现wasm容器管理功能并能够管理wasm容器的整个生命周期，将wasmruntime作为容器底层运行时。

将改造后的containerd和上述项目直接进行多方面的比较:

项目	容器创建流程	容器创建速度	底层容器运行时	高层运行时直接对接k8s	使用containerd
传统linux容器	最长	慢	runc、crun等	能	兼容
docker官方	略长	较快	wasmedge	不能	使用
wasmedge官方	较长	较慢	crun	能	兼容
containerd shims	较短	较快	wasmedge wasmtime	能	使用
改造后的k8s和containerd	短	快	wasmedge	能	使用

根据上述项目的分析和比对，我们项目的改造点主要是基于:
1.cri-plugin的优化，缩短wasm容器部署的准备时间。
2.containerd shim的编写，体现wasm性能优势。

综合以上两点，实现wasm容器高效的部署。