WebAssembly 方案

0 目标

实现 WebAssembly 容器化运行

1 WebAssembly

1.1 wasm 之前

javascript

随着 JavaScript 的快速发展，目前它已然成为最流行的编程语言之一，这背后正是 Web 的发展所推动的。但是随着 JavaScript 被广泛的应用，它也暴露了很多问题：

- 语法太灵活导致开发大型 Web 项目困难；
- 性能不能满足一些场景的需要；

微软 - typescript

MicroSoft 集结了 C# 的首席架构师以及 Delphi 和 Turbo Pascal 的创始人 Anders Hejlsberg 等明星整容，打造了 TypeScript。

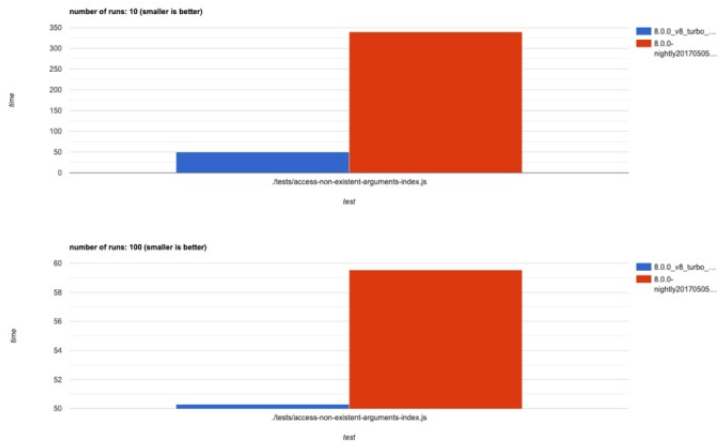
TypeScript 它是 JavaScript 的一个严格超集，并添加了可选的静态类型和使用看起来像基于类的面向对象编程语法操作 Prototype。

但是，由于 TypeScript 最终仍然是被编译成 JavaScript 在浏览器中执行，所以困扰着 JavaScript 开发者的性能问题，仍然没有被解决。

谷歌 - JIT & v8

Google 就推出了自家的 JavaScript 引擎 V8，试图使用 JIT 技术提升 JavaScript 的执行速度，并且它真的做到了。

由于 JIT 技术的引入，V8 使得 Web 性能得到了数十倍的增长！



上图展示了 Chrome 的 v8 与 IE 的 Chakra benchmark 结果。

既然性能得到了如此大的提升，那么 JavaScript 广为诟病的性能问题得到了解决吗？为啥 Web 性能还是被挑战？

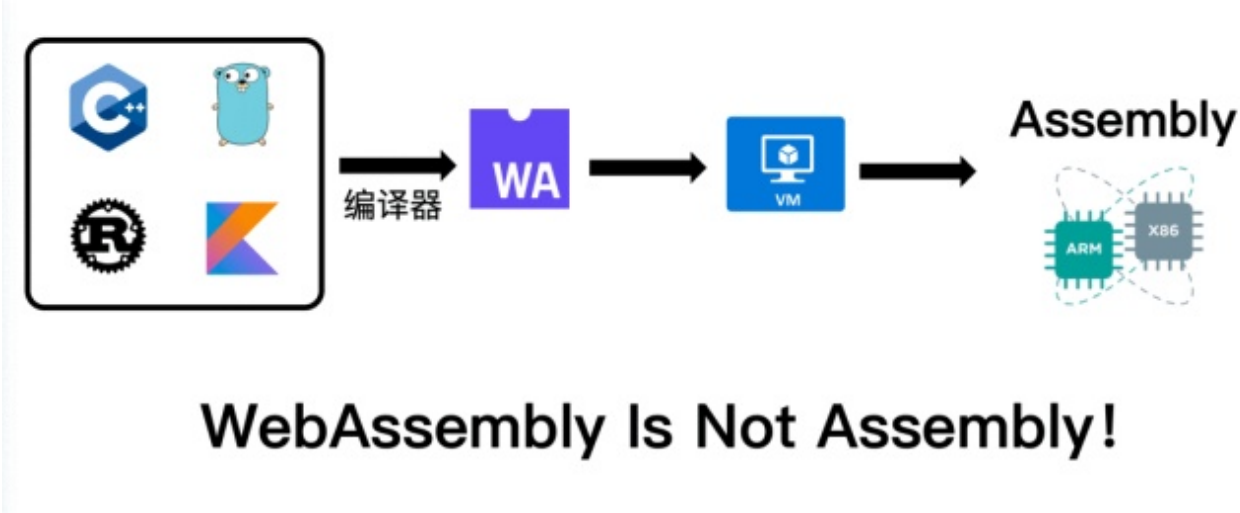
Mozilla - asm.js

和 TypeScript 比较相似的是，asm.js 同样也是强类型的 JavaScript，但是他的语法则是 JavaScript 的子集，是为了 JIT 性能优化而专门打造的。

一段典型的 asm.js 代码如下：

可以看到，asm.js 使用了按位或 0 的操作，来声明 x 为整形。从而确保 JIT 在执行过程中尽快生成相应的二进制代码，不用再去根据上下文判断变量类型。

1.2 wasm 简介



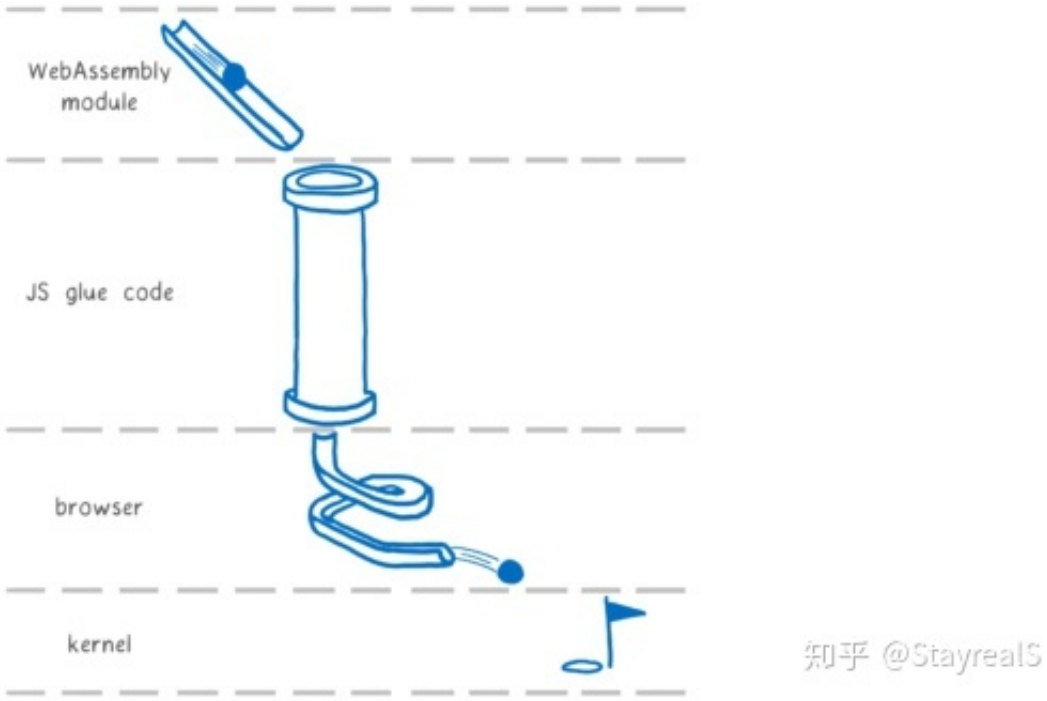
1.3 wasm 工具链

Emscripten

最初是生成asm.js的工具，现被用于生成wasm。

第一个WebAssembly的编译工具是Emscripten。它在网页端模拟了Linux的POSIX接口。这就意味着程序员可以直接使用C标准库（libc）的函数。

为了实现这个模拟，emscripten自己实现了一个版本的libc。这个实现分为两部分，一部分会被编译成WebAssembly的组件（module），另一部分是使用JS胶水代码（glue code）实现的。这个JS胶水代码将会与浏览器沟通，类似于libc与系统调用交互。



绝大多数早期的WebAssembly代码是使用emscripten变异的。所以当人们希望在浏览器外运行WebAssembly的时候，他们就想直接让使用emscripten编译好的代码能够运行（而不是重新设计一个类似标准库的东西）。

使用方式：

```
emcc hello.c # 生成a.out.js胶水，a.out.wasm
emcc hello.c -o hello.html # 生成一个可以运行的网页，可用python -s httpserver 查看
```

支持部分c的标准库，将POSIX调用在浏览器端使用。

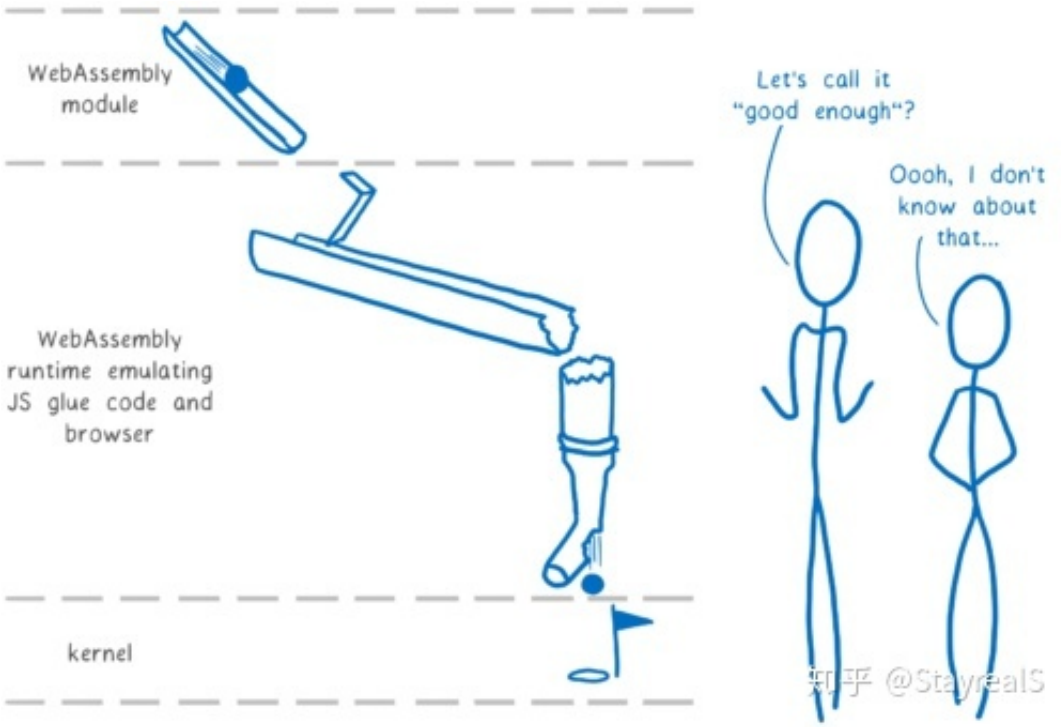
```
printf("hello"); // 没问题
system("echo hello"); // 并不行（其实显然不行）
```

WASI & Wasmtime

在此基础上，WebAssembly想要脱离浏览器运行。

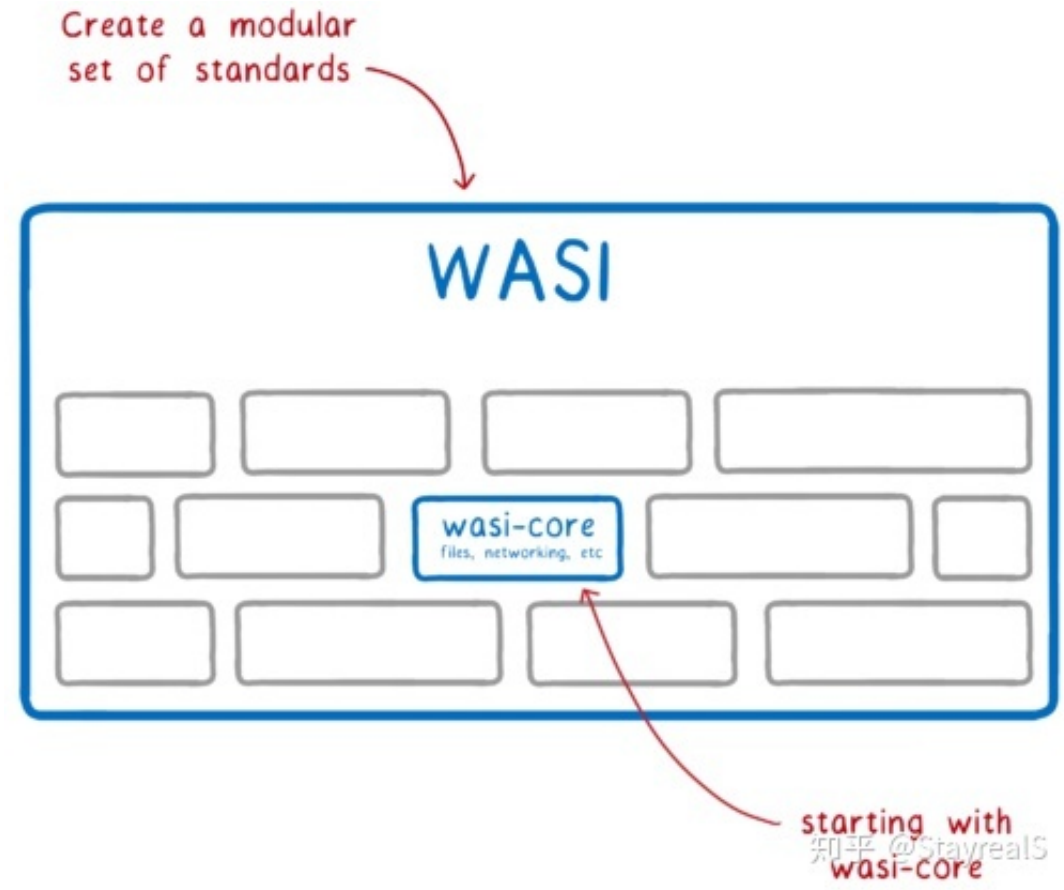
WASI - WebAssembly Stantard Interface - 提供一个标准化的操作系统接口共wasm内部程序使用。

Wasmtime - 以此为基础，wasm的一个脱离浏览器的runtime



这就是我们在标准化过程中所做的事情。并且我们已经有了一个基础的框架：

- 将标准接口分割为一个一个的模块，创建模块集（modular set）
- 从其中一个最基础的模块 `wasi-core` 开始设计



`wasi-core` 中会有什么?

`wasi-core` 将会包含所有程序所需的基础接口。他会覆盖绝大多数的POSIX提供的接口，包括文件、网络、时钟和随机数。

并且他会提供一个类似POSIX的编程方法和环境。例如它将会使用POSIX的基于文件的机制，也即几乎所有的系统调用都是在诸如 `open` , `close` , `read` , `write` 之上的增强和封装。

但是 `wasi-core` 并没有打算覆盖POSIX能做的所有事情。例如进程的概念就无法准确的映射到WebAssembly中。另外，也不是所有的WebAssembly引擎都需要支持像 `fork` 这样的进程操作。尽管如此，我们同样希望标准化fork的实现。。。

使用方式:

```
# 下载标准库后，调用wasi-sdk提供的专门配置的clang编译
./wasi-sdk-8.0/bin/clang hello.c --sysroot ./wasi-sdk-8.0/share/wasi-sysroot -o hello.wasm
# 然后wasmtime运行
wasmtime hello.wasm
```

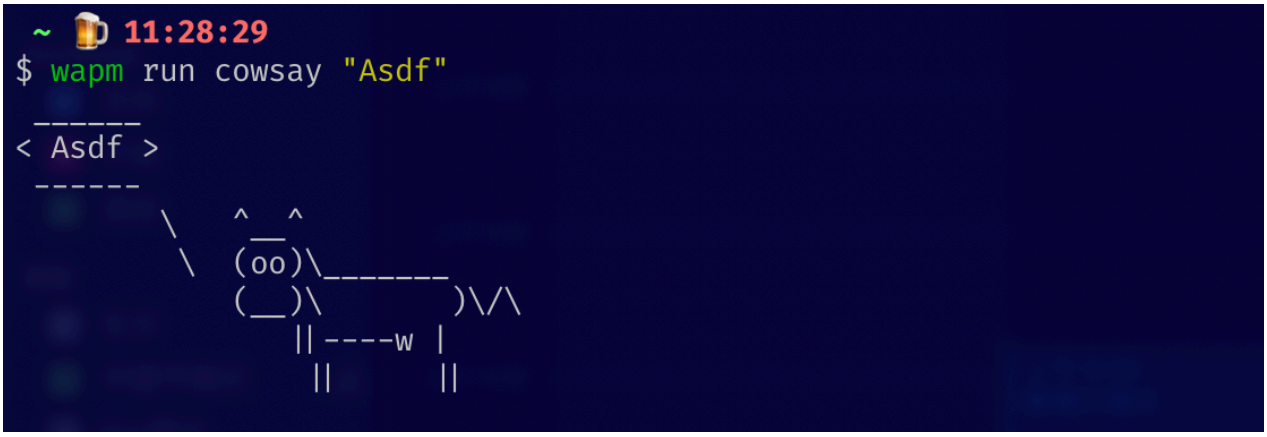
运行测试:

```
printf("Hello"); // 没问题
fprintf(file, "Hello"); // 没问题
system("echo Hello"); // 似乎只有声明没有实现的定义，wasm-ld会报错
```

wasmer & warm

wasmer: **Wasmer** is an open-source runtime for executing WebAssembly on the Server.(The Universal WebAssembly Runtime supporting WASI and Emscripten)

Wapm: **WAPM** is a package manager for WebAssembly modules to be used standalone by any WebAssembly Runtime (such as the Wasmer Runtime).



2 容器虚拟化

2.1 Docker

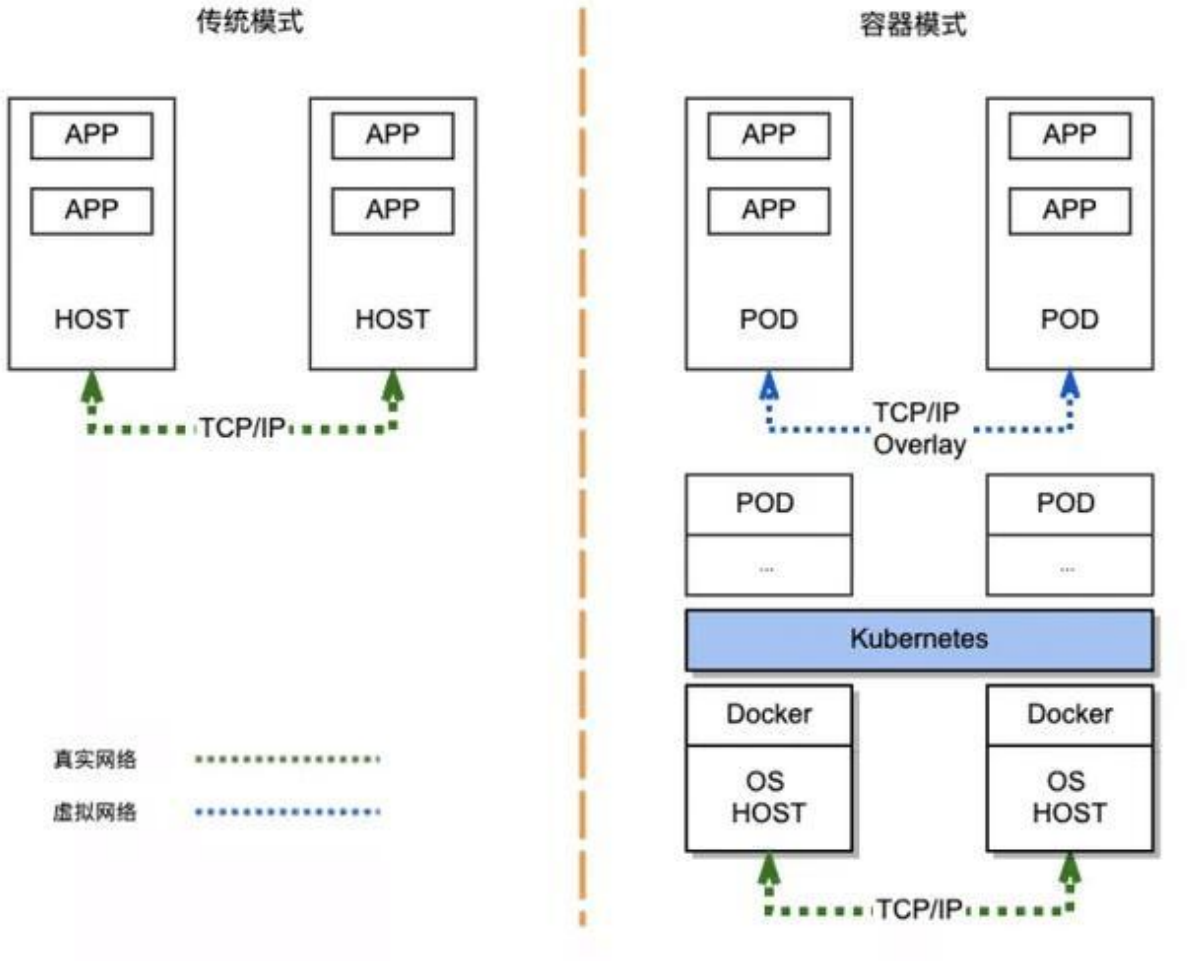
2.2 Kubernetes

Kubernetes 能在实体机或虚拟机集群上调度和运行程序容器。而且，Kubernetes 也能让开发者斩断联系着实体机或虚拟机的“锁链”，从**以主机为中心**的架构跃至**以容器为中心**的架构。该架构最终提供给开发者诸多内在的优势和便利。Kubernetes 提供给基础架构以真正的**以容器为中心**的开发环境。

Kubernetes 满足了一系列产品内运行程序的普通需求，诸如：

- **协调辅助进程**，协助应用程序整合，维护一对一“程序 - 镜像”模型。
- **挂载存储系统**
- **分布式机密信息**
- **检查程序状态**
- **复制应用实例**
- **使用横向荚式自动缩放**
- **命名与发现**
- **负载均衡**
- **滚动更新**
- **资源监控**
- **访问并读取日志**
- **程序调试**
- **提供验证与授权**

以上兼具平台即服务（PaaS）的简化和基础架构即服务（IaaS）的灵活，并促进了在平台服务提供商之间的迁移。



2.3 containerrd项目

美国时间 2016 年 12 月 14 日，Docker 公司宣布将 containerd 从 Docker 中分离，并捐赠到一个新的开源社区独立发展和运营。 containerd 可以作为 daemon 程序运行在 Linux 和 Windows 上，管理机器上所有容器的生命周期。阿里云、AWS、Google、IBM 和 Microsoft 作为初始成员，会为项目提供功能开发和维护。

containerd 对于很多人来说还是很陌生。Docker 公司为什么会大张旗鼓地宣布这个开源项目，并能得到业界巨大的反响呢?

实际上，早在 2016 年 3 月，Docker 1.11 的 Docker 里就包含了 containerd ，而现在则是把 containerd 从 Docker 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 里的 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 unix domain socket 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 完成的。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。 containerd 项目架构图如图 7.5 所示。

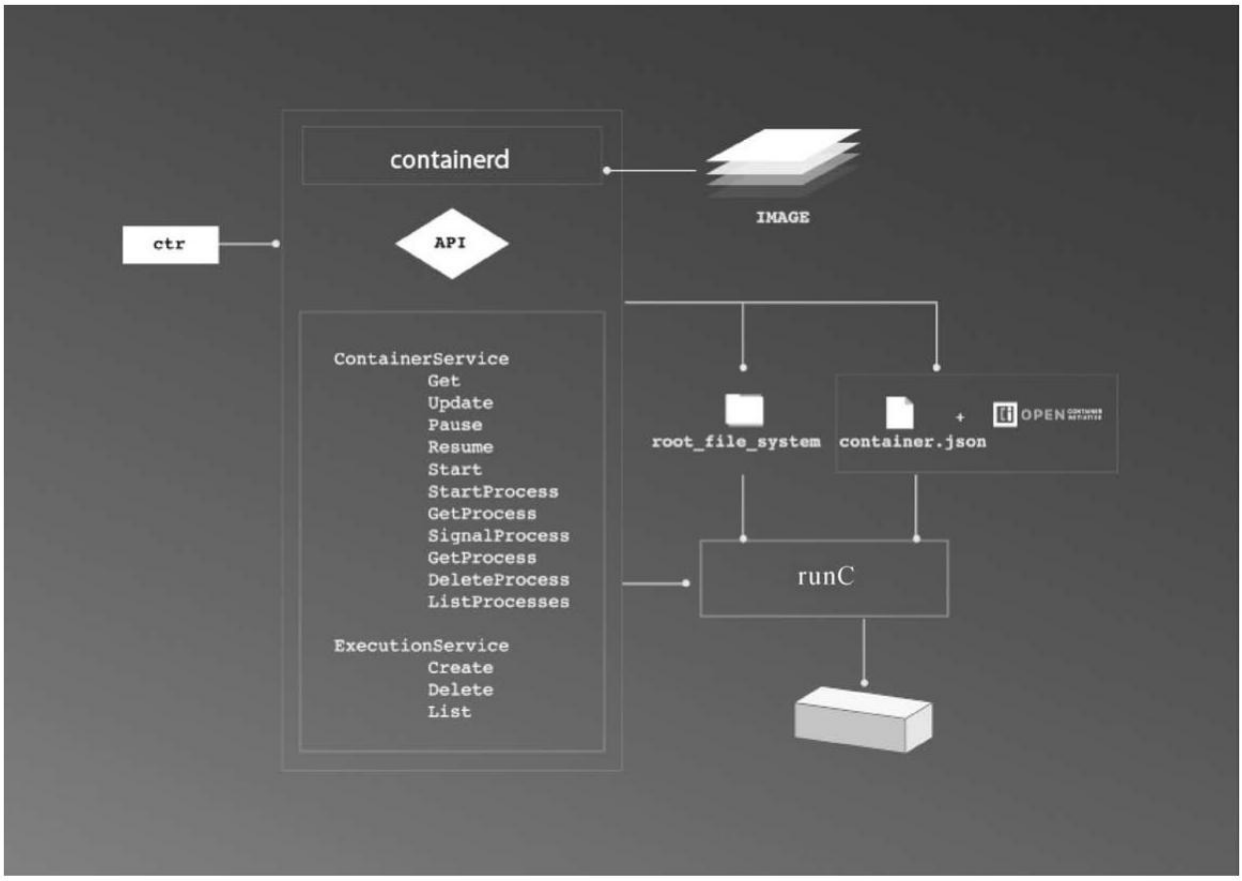


图 7.5

这对于社区和整个 Docker 生态来说是一件好事。对于 Docker 社区的开发者来说，独立的 containerd 更简单清晰，基于 containerd 增加新特性也会比以前容易。

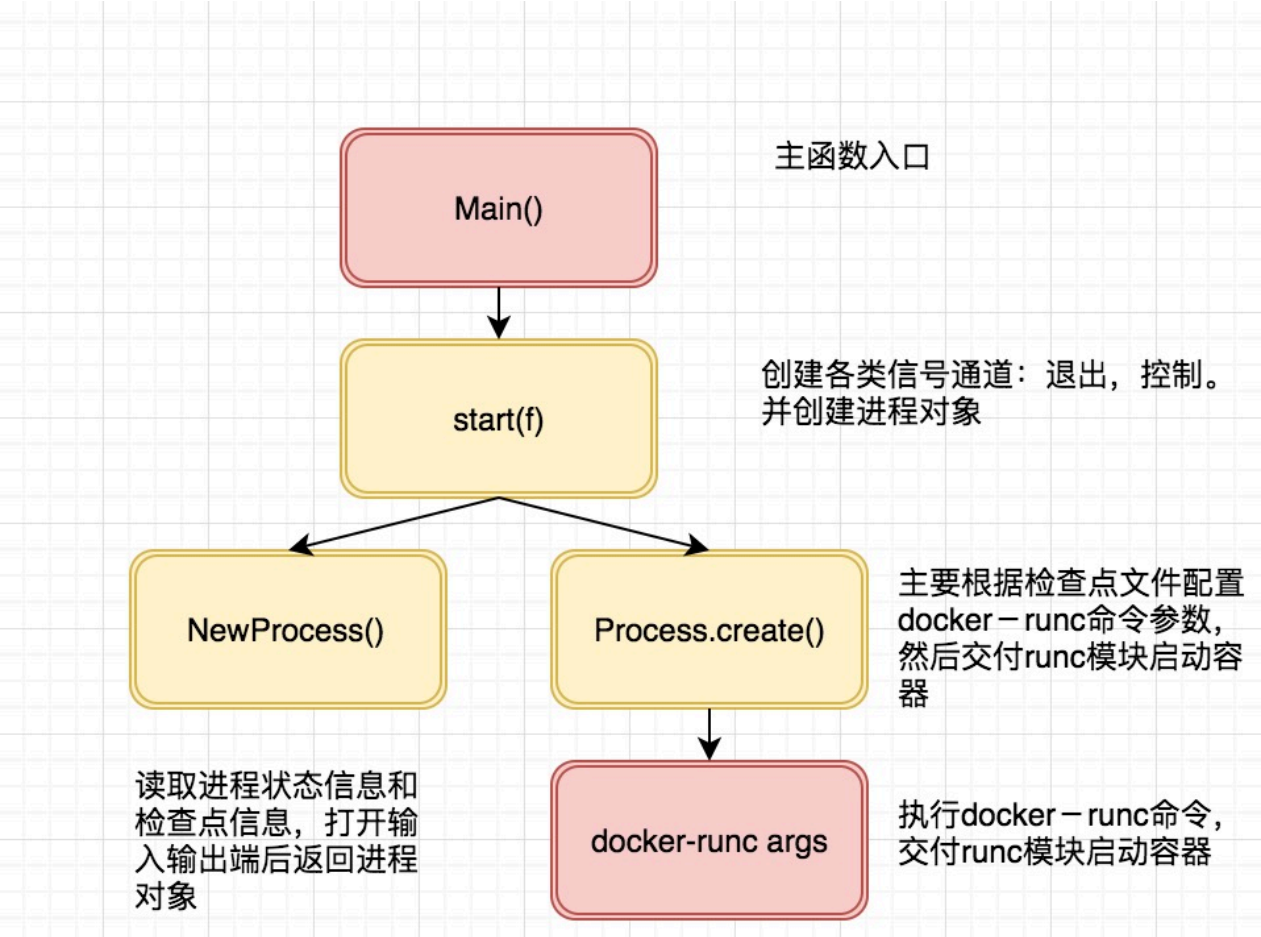
对于容器编排服务来说，运行时只需要使用 containerd +runC，则会更加轻量、容易管理。而独立之后， containerd 的特性演进可以和 Docker 分开，专注容器运行时的管理，这样可以更稳定。在向后兼容上也可以做得更好， containerd 第一个正式版本 1.0 Release 之后将提供一年的支持，包括安全更新和 Bugfix，而每次升级也会向后兼容一个小版本。

Docker 为了表示对于社区和生态的诚意，特意强调了 containerd 中立的地位，符合各方利益。可以预见， containerd 将成为 Docker 平台的一个重要组件。阿里云、AWS、Google、IBM 和 Microsoft 将参与到 containerd 的开发中。

2.4 containerd-shim

containaienrd模块启动容器的过程，在其将一系列配置工作完成后通过调用containerd-shim id /var/run/libcontaienrd/ct_id docker-runc命令交付给containerd-shim子模块来启动容器，所以这一章节将进入containerd-shim模块入口来深入容器的启动过程。

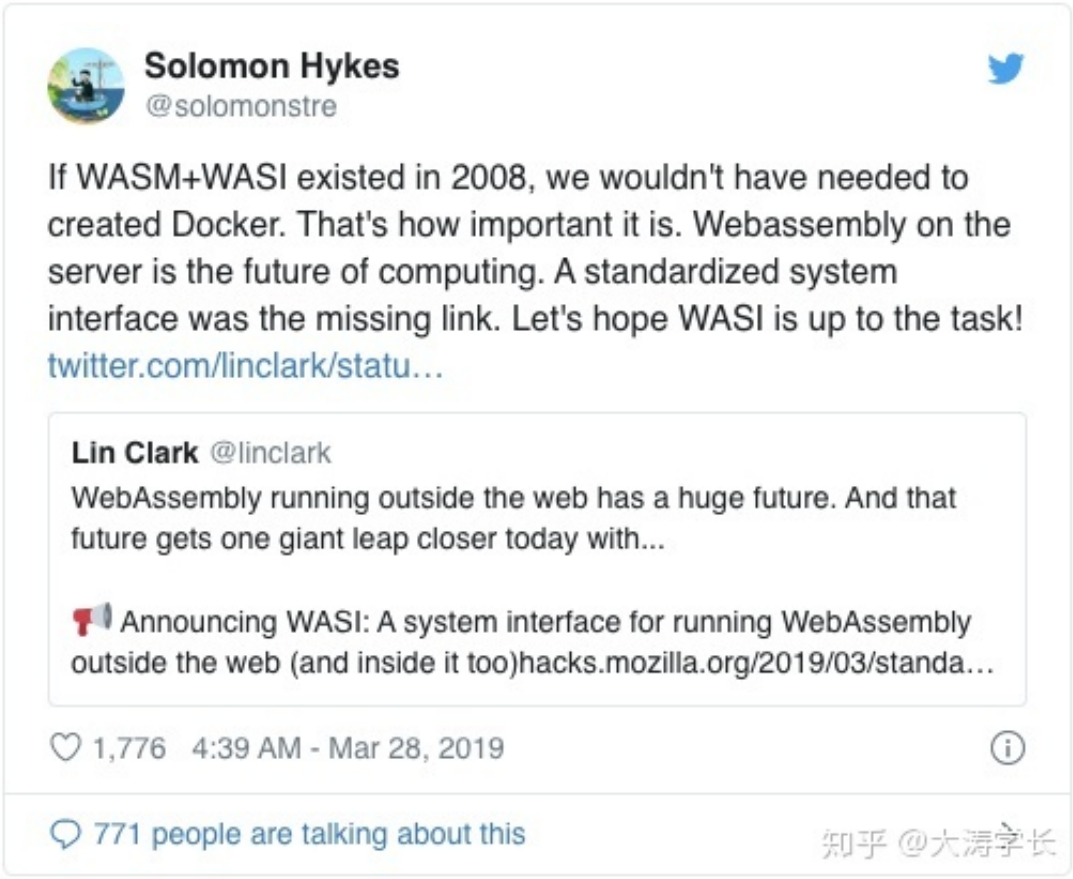
containerd-shim其实相当于containerd到下层runc的一层夹层，即一层接口来适应容器运行环境的实现。



2.5 wasm 与容器技术结合：WebAssembly容器

Mozilla在2019年3月推出了 WebAssembly System Interface（WASI），来标准化WebAssembly应用与系统资源的交互抽象，比如文件系统访问，内存管理，网络连接等，类似POSIX这样的标准API。WASI规范大大拓展了WASM应用的场景，可以让其可以超越浏览器环境，作为一个独立的虚拟机运行各种类型的应用。同时，平台开发商可以针对具体的操作系统和运行环境提供WASI接口不同的实现，可以在不同设备和操作系统上运行跨平台的 WebAssembly 应用。这可以让应用执行与具体平台环境实现解耦。这一切使得“Build Once, Run Anywhere”的理想逐渐形成现实。WASI 的示意图如下所示。2019年月，为了进一步推动模块化 WebAssembly 生态系统，Mozilla、Fastly、英特尔和红帽公司携手成立了字节码联盟（Bytecode Alliance），共同领导 WASI 标准、 WebAssembly 运行时、语言工具等工作。

正因为 WebAssembly 所具备的的安全、可移植、高效率，轻量化的特点，非常适于应用安全沙箱场景。WASM得到了容器、函数计算、IoT/边缘计算等社区的广泛关注。Docker创始人Solomon Hykes在WASI发布之际的一句Twitter，更是成为了去年容器和WebAssembly社区引用频率最高的一句话之一。



Fastly, Cloudflare等CDN厂商基于WebAssembly技术实现了更加轻量化的应用安全沙箱，可以在一个进程内部运行多个独立的用户应用。阿里云CDN团队EdgeRoutine也实现了类似技术。与容器技术相比，WASM可以实现毫秒级冷启动时间和极低的资源消耗。

应用分发

Docker容器的一个重要贡献是其标准化了容器化应用打包规范 Docker Image，而且它已经成为开放容器计划(Open Container Initiative - OCI)的镜像格式标准。Docker镜像提供了自包含、自描述的镜像格式。它可以将应用以及其依赖的环境信息打包在一起，从而实现应用与运行环境解耦，让容器应用可以轻松运行在从本地开发环境到云端生产环境的不同场景中。并且社区围绕Docker镜像构建了繁荣的工具链生态，如Docker Hub可以进行应用分发和CI/CD协同，Nortary/TUF项目可以保障应用可信地分发、交付。

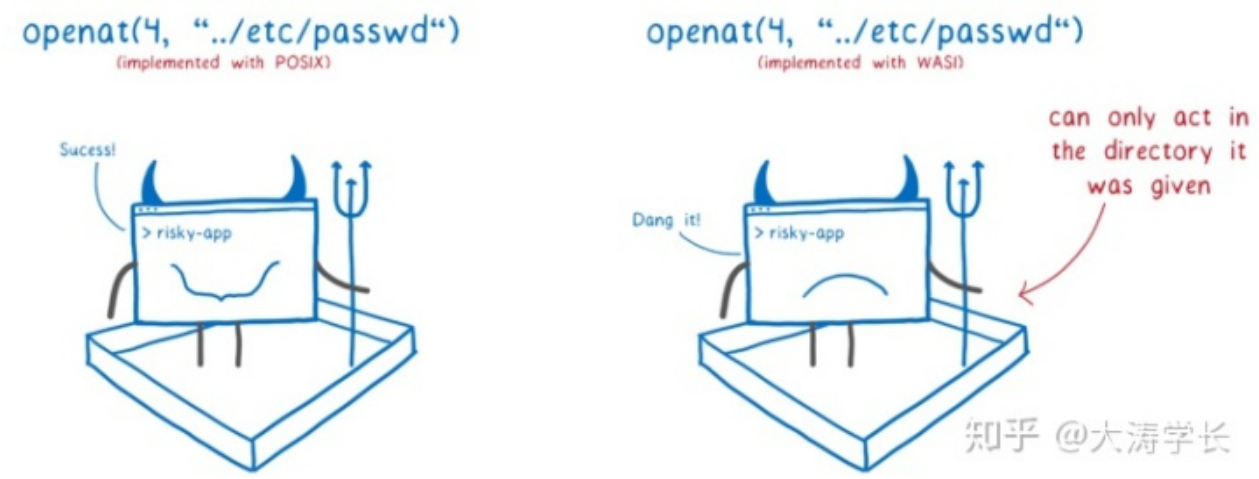
对与WebAssembly，目前社区提供了类似NPM的包管理实现 **WAPM**，可以较好地支持应用的分发。 为WebAssembly应用构建Docker镜像，可以实现双赢的局面。

- WebAssembly开发者可以完全复用Docker/OCI镜像规范和工具链，进一步简化应用分发和交付。比如，我们可以将Nginx的WASM镜像作为基础镜像，基于这个镜像可以构建包含不同Web内容的应用镜像；我们可以利用tag对应用版本进行追踪；利用Docker Registry进行应用分发；在这个过程我们还可以进一步利用数字签名来保障安全的软件供应链。
- Docker镜像规范支持 **Multi-Arch** 镜像，可以简化不同CPU体系架构（如x86, ARM, RISC-V等）的应用镜像的构建与分发。而WebAssembly天生具备可移植性，大大简化了跨平台Docker应用镜像的构建和分发。
- - 参考： **利用Docker加速 ARM 容器应用开发和测试流程**

安全隔离

WebAssembly的最初设计目标是让应用可以安全运行在浏览器中。WASM虚拟机提供的的**沙箱和内存隔离机制**，可以有效减少安全攻击面。而当WebAssembly走出浏览器，面向更加通用的场景。WASM也面对更加复杂的安全挑战。

WASI 提供了**基于能力的安全模型**。WASI应用遵循最小权限原则，应用只能访问其执行所需的确切资源。传统上，如果应用需要打开文件，它会带路径名字符串调用系统操作open。然后系统调用会检查应用是否具有访问该文件的相关权限，比如Linux实现了基于用户/组的权限模型。这样隐式的安全模型，依赖于正确的安全管理配置，比如一旦特权用户执行了一个恶意应用，它就可以访问系统中任意的资源。而对于WASI应用而言，如果它需要需要访问指定文件等系统资源，需要从外部显式传入加有权限的文件描述符引用，而不能访问任何其他未授权资源。这中依赖注入的方式可以避免传统安全模型的潜在风险。一个示意图如下



原图： https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/

我们可以看到WASI的安全模型与传统操作系统安全模型非常不同，而且还在持续演进中。比如字节码联盟提出了 nanoprocess 来解决应用模块间的安全协同和信任传递。

调度与编排

在云时代，Kubernetes已经成为分布式环境下资源调度和应用编排的事实标准。Kubernetes可以屏蔽底层设施的差异性。可以在同一个K8s集群中包含x86、ARM等不同体系架构的节点，可以支持Linux，Windows等不同的操作系统。Kubernetes和WebAssembly相结合可以进一步提升应用的可移植性。

微软的Deis Labs年初发布了一个实验项目， https://github.com/deislabs/krustlet 来利用 Virtual Kubelet 类似的架构调度 WebAssembly 应用。但是这个方式有很多局限，无法借助容器方式进行应用分发，也无法利用 K8s 的语义进行资源编排。

难得有一个春节假期可以宅在家里间，我基于Derek McGowan去年的一个实验性项目 https://github.com/dmcgowan/containerd-wasm，完善了containerd的WASM shim实现。可以让containerd支持WASM container，并且可以利用Kubernetes集群管理和调度 WASM container。 项目的代码实现: https://github.com/denverdino/containerd-wasm

注：这个项目更多是概念验证，进程管理、资源限制，性能优化等的细节并没未完整实现。

3 项目考虑

https://github.com/dmcgowan/containerd-wasm 提供了一个 wasm containe r的containerd实现，我们可以进一步完善这个实现。

https://github.com/denverdino/containerd-wasm 在上一个实现上做了进一步的改进，但是作者本人说“这个项目更多是概念验证，进程管理、资源限制，性能优化等的细节并没未完整实现。”我们可以在这基础上实现相

应的进程管理、资源限制的功能。

我可以考虑与作者联系，了解我们是否能为该项目做一些工作。

可行性：

- 项目采用go语言，对于熟悉C的大家不是问题。
- .....

参考

1. 【译】【图文】标准化中的 WASI：在 web 之外运行 WebAssembly 的系统接口
2. https://leefsmp.github.io/Particle-System/slides/index.html#/1
3. https://github.com/mbasso/awesome-wasm
4. https://zhuanlan.zhihu.com/p/111057726