wasm运行时和容器调研与对比评测

一、WebAssembly调研

1.1 WebAssembly介绍及其特性

1.1.1 WebAssembly简介

WebAssembly(简称Wasm）是一个高效、安全、开放、标准的字节码格式.这种全新格式可移植、体积小、加载快并且兼容 Web，通常作为可作为其它语言的编译目标,而不是用于书写程序。

高效。Wasm有一套完整的语义，是体积小、加载快的二进制格式， 其目标就是充分发挥硬件能力以达到原生执行效率。

安全。Wasm运行在一个沙箱化的执行环境中，甚至可以在现有的 JavaScript 虚拟机中实现。在web环境中，WebAssembly将会严格遵守同源策略以及浏览器安全策略。

开放。Wasm设计了一个非常规整的文本格式用来、调试、测试、实验、优化、学习、教学或者编写程序。可以以这种文本格式在web页面上查看wasm模块的源码。

标准。WebAssembly 在 web 中被设计成无版本、特性可测试、向后兼容的。WebAssembly 可以被 JavaScript 调用，进入 JavaScript 上下文，也可以像 Web API 一样调用浏览器的功能。除此之外，也可以运行在非web环境下。

鉴于Wasm 轻量、高性能、沙箱安全、高可移植等特点，可以在以下场景发挥作用：

1. 迁移成熟的工具，如： Google Earth，Unity3D, CAD 等等，包括轻游戏
2. 多媒体： 更轻量、消耗更低的CPU和内存
    a. 视频/直播编解码；
    b. 在线图像/视频处理应用；
3. 基于边缘计算的机器/深度学习：MXNet.js；
4. 高性能 Web 游戏：Unity、Unreal、Ammo.js 等游戏库和引擎；
5. 区块链 Ethereum 核心；
6. 前端框架：sharpen、asm-dom、yew；
7. IOT：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

Wasm的轻量体现在两点，一是其本身小：格式紧凑，运行时体积小。二是不需要额外的依赖。性能方面下文将在性能测评部分来展示这一点。沙箱安全和可移植特性将单独一个小节来说明

1.1.2 Wasm的安全性

WebAssembly 运行在一个沙箱化的执行环境中，甚至可以在现有的 JavaScript 虚拟机中实现。在web环境中，WebAssembly将会严格遵守同源策略以及浏览器安全策略。不同于分级保护，Wasm的安全机制使用的是 capability-based security模型。

当使用分级保护时，如果程序需要打开一个文件，它会调用open系统调用并传入文件位置，然后由操作系统检查这个程序是否有权限打开（基于程序的用户权限）。

在具有capability概念的系统中，只要用户具有capability，他就具有足够的权限去访问资源。Wasm使用一种叫open-at形式的系统调用。如果调用一个需要打开某个文件的程序，那么必须要传入对应文件的file descriptor（它有权

限），即指定程序执行可以访问的文件夹。文件实例是由操作系统打开，然后WASI把该实例传递给Wasm沙箱。因为该实例能打开的文件范围仅仅是文件夹圈定的范围，所以Wasm程序能访问的文件系统范围也被圈定在文件夹以内。

以open为例，分级保护的函数是 `fd = sys_open(filename, flags, 0666);` Wasi的capability-based的open-at形式则是 `wasmtime --dir=filepath` （以 Wasmtime为例），它通过 `dir` 参数指定可访问的文件系统范围。

Wasm的 capability-based security模型更适合现今"基于众多不知来源的第三方库进行代码开发"的现状。

1.1.2 Wasm的可移植性

Wasm的可移植性体现在两点，一个是Web环境下基于浏览器的可移植性，一个是非web环境下基于WASI的可移植性。

基于浏览器的可移植性:

WebAssembly 目前通过 JavaScript 来加载和编译。基础的加载只需要3步：

1. 获取 .wasm 二进制文件，将它转换成类型数组或者 ArrayBuffer
2. 将二进制数据编译成一个 WebAssembly.Module
3. 使用 imports 实例化这个 WebAssembly.Module，然后获取 exports

通过JS加载以后，Wasm字节码即可在浏览器上运行。Wasm设计之初就是可以被浏览器解析执行的，由浏览器的可移植特性而获得可移植性。

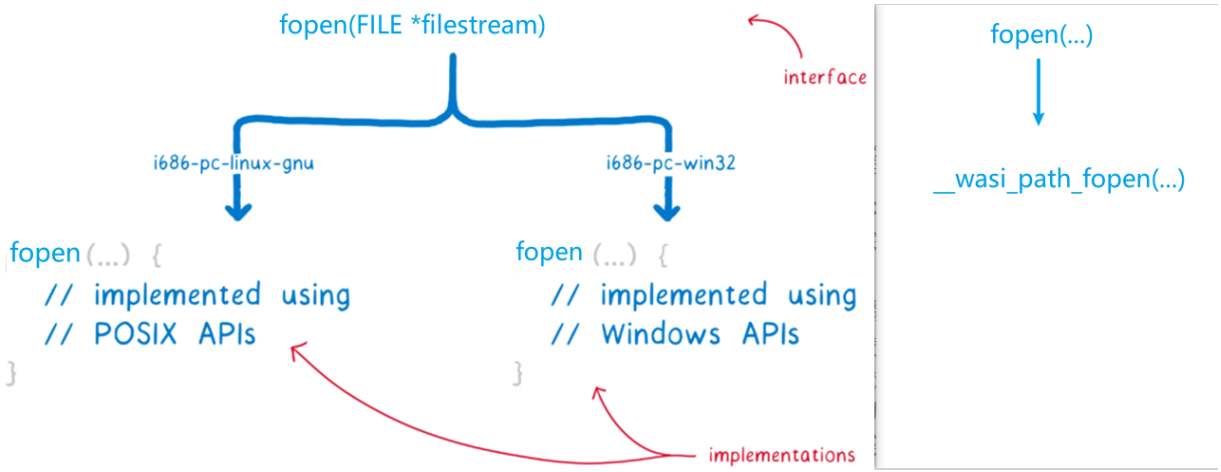
非Web环境下的可移植性:

目前已经有许多组织（如字节码联盟）为Wasm开发了许多可独立运行的运行时（如：wasmtime，wasmEdge，wasmer等等），这些运行时都可以不依赖浏览器来执行Wasm程序的计算任务。计算任务可以完全在沙箱内执行，但是系统调用的执行会涉及到操作系统的敏感资源，这些函数的具体执行往往和具体的平台有关，而Wasm要求的可移植性不允许这样的平台相关性。为了解决这个问题产生了WASI。

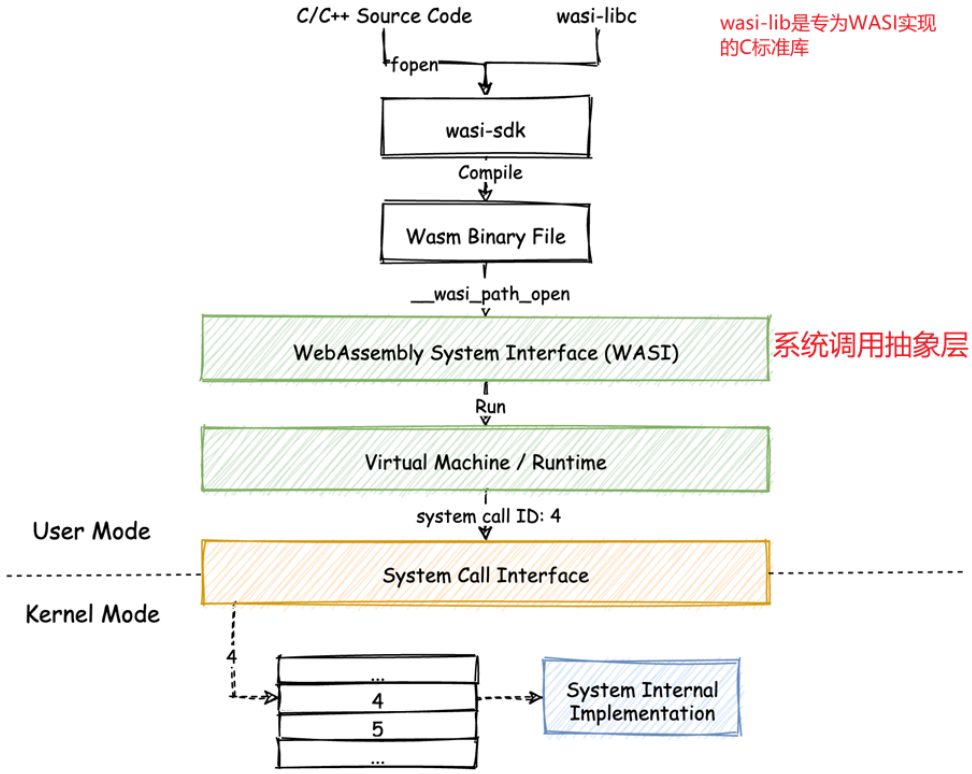
WASI是WebAssembly的模块化系统接口，专注于安全性和可移植性，提供类似posix(portable operating system interface)的系统调用函数。WASI向上提供统一的抽象的系统调用函数，向下把抽象系统调用映射到具体的系统调用。由运行时实现对WASI的支持。

以C语言为例，当系统调用函数被编译时，会根据要运行的平台选择对应平台的系统调用。然而，由于Wasm的可移植性要求代码在任何未知的平台都可以运行，所以系统调用函数编译为Wasm字节码时无法确定的应当选用哪个平台的系统调用函数实现。WASI把这些具体平台的系统调用抽象成一个抽象的系统调用，这样C代码的系统调用函数在编译时可以编译为抽象的系统调用，之后在Wasm运行时执行该系统调用时再根据所处的平台执行正确的系统调用。其他语言的代码也是相似的。

更具体的，以C中的 `fopen()` 为例（如下图）。编译时需要确定代码是在Linux平台执行还是Windows平台执行，以此选择i686-pc-linux-gnu的 `fopen()` 还是i686-pc-win32的 `fopen()` 。WASI则把具体平台的多个 `fopen()` 抽象为一个抽象系统调用 `__wasi_path_fopen()` ，这个抽象的系统调用定义在c标准库*wasi-libc（*一个专为WASI实现的C标准库*）*中。至此，C代码成功的编译为wasm字节码。



Wasm字节码由具体的运行时运行，当运行时执行到 `__wasi_path_fopen()` 时，根据所处的平台选用正确的系统调用来执行（如下图）。



1.2 主流的wasm运行时及其特点

为了在非Web环境下独立运行Wasm程序，包括字节码联盟在内的许多组织开发了Wasm的运行时。出现过的运行时有wasmtime、wasmer、wasmEdge、WAVM、WAMR、Wasmi、Wasm3、Lucet等等，在近年的发展中，有几个运行时由于自身优秀的特性脱颖而出成为主流的运行时，这里着重关注3个运行时：wasmtime、wasmEdge和wasmer。

Wasmtime是由字节码联盟开发和维护的快速、安全的Wasm运行时，可以独立执行Wasm程序或者嵌入到其他程序当中。Wasmtime的一个重要特点是标准，通过了Wasm的官方测试，是一个标准的实现。Wasmtime基于Cranelifit产生高质量的机器码，执行速度优秀。Wasmtime还具有可配置的特点，可以根据需要进行细粒度的配置，即使在物联网设备上也可以运行。

WasmEdge是由云计算基金会（CNCF）开发和维护的一个快速的Wasm运行时。WasmEdge的目标是将云原生和serverless应用带到边缘计算，和容器相比拥有较快的启动速度，在执行速度方面也优于容器。WasmEdge使用类似WASI的扩展，支持网络sockets、异步进程、键值对存储、TensorFlow等拓展功能。此外还有JS的ES6 module支持、容易嵌入到宿主应用等优点。WasmEdge最具有特色的一点是具有云原生工具链的支持，如docker和K8s的支持。

Wasmer 是开源社区wasmerio开发和维护的一个运行时。其鲜明的特点是支持最多高级语言，支持三种编译后端（Singlepass、Cranelift和LLVM），支持所有平台，拥有运行时无关的包管理器WAPM。可增强区块链、Faas和机器学习领域的能力。

三个运行时的主要特点总结为下表：

	wasmtime	**wasmer**	**wasmEdge (SSVM)**
关键词	标准的 可配置的	最多语言、平台支持 包管理器	云原生工具链支持 边缘计算
组织	字节码联盟	wasmerio	字节码联盟
支持语言	Rust 、C /C++ 、Python 、.NET 、Go	所有主流语言	Rust、C/Cpp、 Node.js、 Go
编译后端	Cranelift	Cranelift、Singlepass、LLVM	LLVM
WASI	支持	支持	支持
执行方式	JIT AOT	JIT AOT	interpret（很慢） AOT
特色	标准实现 性能优秀	语言、平台等支持最广 性能很好 适用于区块链/Faas/机器学习	云原生工具链支持 适用于云计算 接近原生程序的AOT执行速度
其他特性	优秀的JIT执行速度	优秀的JIT执行速度 支持交叉编译	Interpret模式执行速度很慢 不支持交叉编译
直接执行性能	Wasmer = Wasmtime > WasmEdge （Wasmer和Wasmtime接近，WasmEdge明显比其他两个慢）		
AOT执行性能	WasmEdge> Wasmer = Wasmtime		

1.3 Wasm在x86和ARM的性能和成本建议

通过调研Wasm在x86平台和ARM平台的性能表现，找到一篇名为《 WebAssembly as a Common Layer for the Cloud-edge Continuum 》的论文，作者使用WAMR作为Wasm的运行时，在x86和ARM平台进行了性能测试。该测试使用的一台配置了Intel Xeon E3-1275 v6 (3.8GHz)的服务器和一台配备了Arm Cortex-A53 (1.5GHz) 的机器。

为了两台机器之间可以进行比较，都以各自的原生程序执行时间为单位1， 即默认了原生程序的执行性能不受平台的影响。下图是测试的结果，可以看出在测试集 Polybench上，x86平台的执行性能要优于ARM平台（ARM上需要更多的时间）。

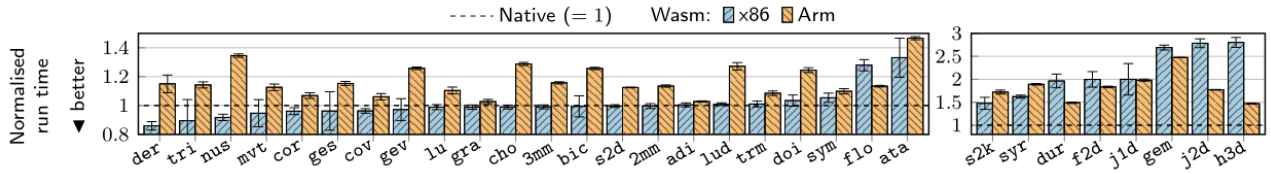


Figure 2: Relative performance of `Polybench/C` benchmarks.

在测试集SQLite Speedtest benchmarks 上则是ARM平台的Wasm程序执行性能要明显优于在x86平台时的执行性能（x86平台上的执行时间是原生程序的2.7倍，ARM平台上的执行时间则是原生程序的2.1倍）。

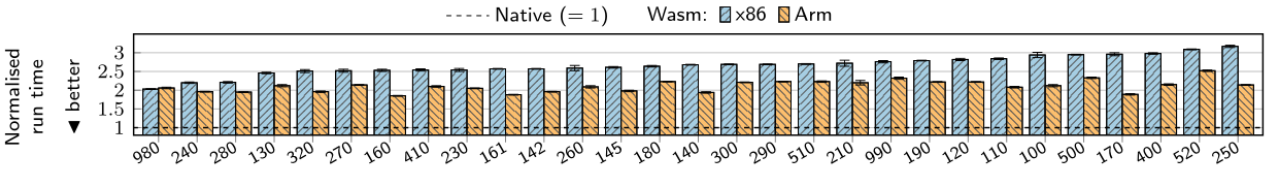


Figure 3: Relative performance of `SQLite Speedtest1` benchmarks.

上面的调研可以看出，x86和ARM平台各自在一些测试集上更优优势，总体上性能相当，但是使用ARM平台需要的成本更低，因此使用ARM平台更节约成本。

1.4 Wasm开发及调试工具调研

经过调研发现VS code支持Wasm的开发，同时试用VS code 编写和调试一个demo程序如下。

前置依赖

CMake：编写CMakeLists.txt文件用于项目管理

Ninja：对项目进行快速构建

Wasm runtime：wasm文件需要运行时进行解释执行

WASI-SDK编译器：借助LLVM工具链将C/C++代码编译为.wasm目标格式，通过clang编译目标文件，选定编译参数，存储调试信息

VScode拓展

CodeLLDB：用于对C/C++代码的调试

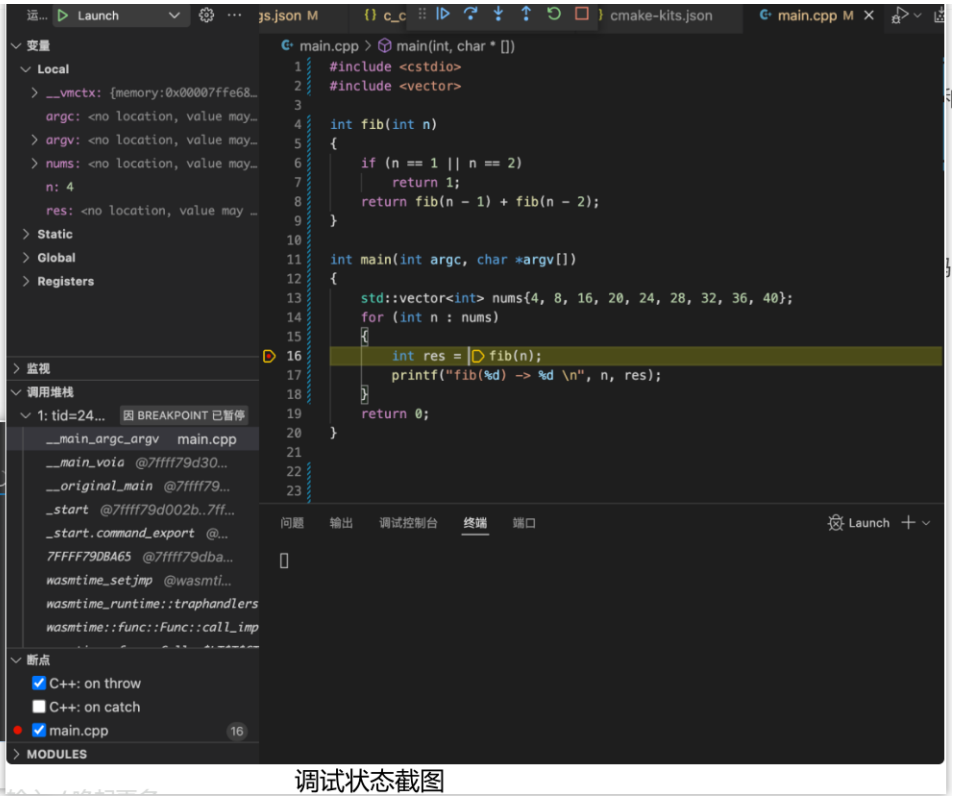
C/C++ Extension Pack：C/C++辅助工具

要使用VS code进行Wasm的开发，需要经过以下步骤：

1. 配置工具链: 在 .vscode 目录中创建两个文件 cmake-kits.json 和 c_cpp_properties.json
    a. 其中cmake-kits.json用于配置 WASM 编译器工具链，指定使用 wasi-sdk。

2. 配置启动：由于编译的结果是 wasm 文件，无法直接执行，我们需要使用 wasmruntime 加载和启动编译后的 wasm 文件，所以要配置运行时等信息（在launch.json）
    a. 另外，由于 wasi-sdk 不支持 C++ 异常，需要关闭异常功能
3. 运行：Build Variant 选择 Debug 或 RelWithDebInfo
4. 调试：wasmtime 加载 wasm 文件后，会在 JIT 模式下运行，LLDB 支持调试 JIT 后的脚本代码。我们可以直接在 C++ 源码中设置断点。同理，在wasmedge或其他运行时执行时，也可以参照此方式进行调试

示例项目结构



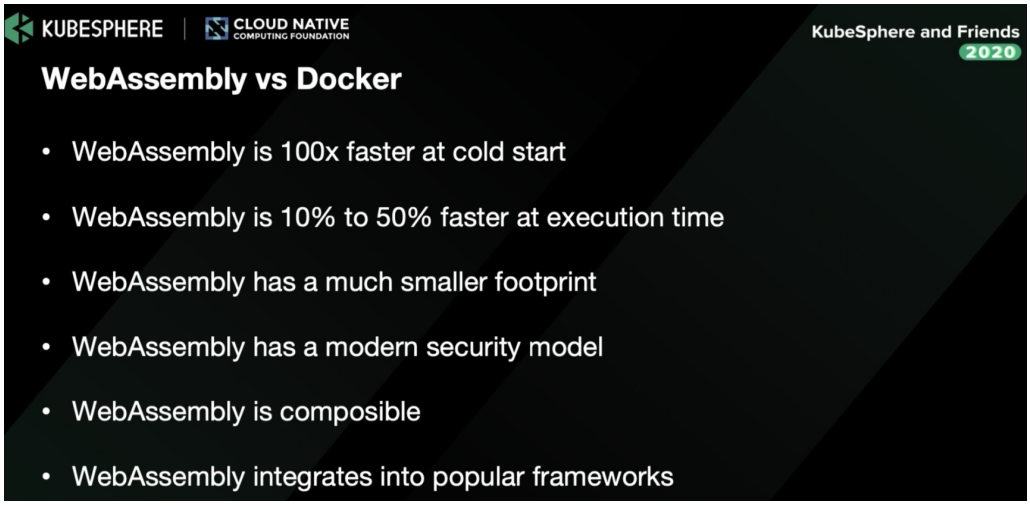
调试状态截图

二、wasm运行时和容器性能对比

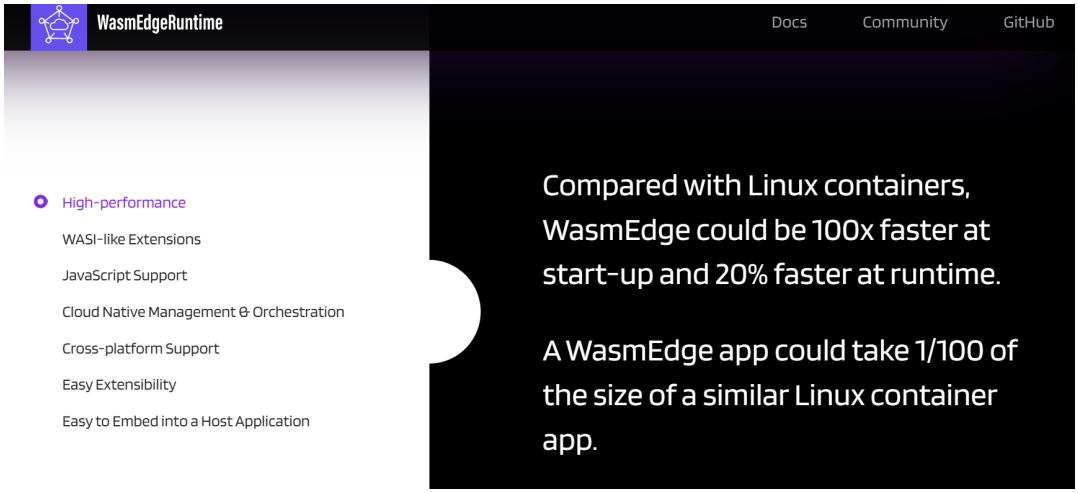
2.1 wasm运行时和容器的性能对比

2.1.1 Wasm运行时和容器性能对比调研

SSVM 项目 Maintainer Michael Yuan 博士在 KubeSphere 2020年度 meetup 的分享中给出的对比：Wasm和 Docker 相比，在启动速度上快100倍，在运行速度上快10%–50%， 在占用的内存上要小得多。



WasmEdge的官网介绍中表示：就和Linux容器相比，WasmEdge在启动速度上要快100倍，在运行速度上要快20%，在空间占用上只需要Linux容器的1/100。

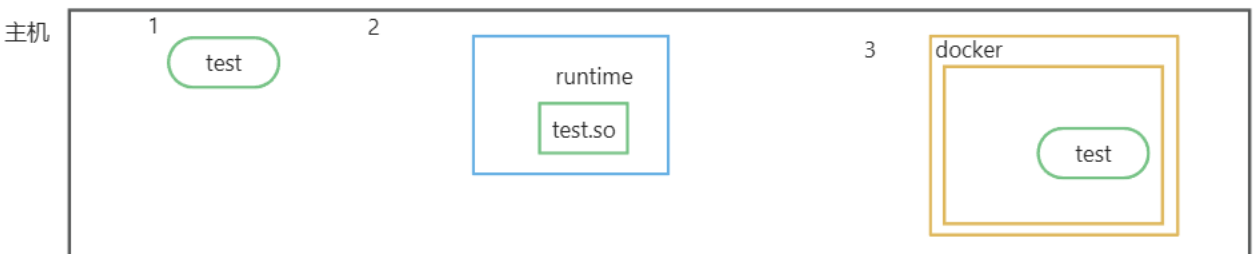


2.1.2 Wasm运行时和容器性能对比测评

从前面的调研可以知道：WasmEdge的执行速度比Linux容器快20%左右，接下来的测试的目的有：

1. Wasm的性能损失：Wasm的执行速度和原生的程序相比慢多少?
2. Wasm相对容器的性能优势：
    a. Wasm的执行速度和容器相比快多少?
    b. Wasm的启动速度和容器相比快多少?

测试方案：



1. 在主机上执行Linux程序: 作为比较的参照
2. 在主机上, 使用runtime运行wasm程序: 包括runtime启动的时间
3. 在主机上,使用docker–linux运行linux程序: 包括启动docker的时间
4. 测试指标:
    a. 执行时间：测试程序启动到结束的时间，用于比较执行速度
    b. 启动时间：程序启动到执行第一行代码的时间，通过执行空程序测试启动时间。用于比较启动速度

内存的使用：测试native、运行时执行、容器执行所需的运行内存大小，用于比较内存消耗。

测试对象：

native：原生程序

wasm aot： AOT编译后的Wasm程序

docker native： 使用docker容器执行和native一样的Linux程序

测试目的：Wasm运行时的执行速度和容器执行速度比较

测试程序： binarytree.c （不同深度的满二叉树遍历）

运行时： WasmEdge

序号	项目	测试命令	执行时间(s)	native=1的执行时间	内存占用(MB)	容器内存占用(MB)
1	native	\time –v –o tmp1 . /binarytree 20	6.56	1	128.8	/
2	wasm aot	\time –v –o tmp3 wasmedge ./biwasm.wasm 20	7.07	1.078	79.79	/
3	docker native	\time –v –o tmp5 docker run –i conttest:v1 CMD [” /binarytree”,”20”]	8.54	1.302	128.57	35.25

以原生程序执行时间6.56s作为标准1，在WasmEdge上经过AOT编译后的执行时间为1.078，使用docker容器执行的时间为1.302，由此可以得出：

1. WasmEdge上经过AOT编译后的执行速度接近原生程序的速度，此例中是原生的92.79%
2. 此例中容器运行的速度是原生的76.82%， 由此得到WasmEdge在经过AOT编译后的执行速度比容器快15.94% , 基本吻合官方宣称的20%的优势
3. 内存上，使用容器执行程序占用的内存和原生程序相当，但是使用容器时容器本身需要额外的内存。使用运行时的内存占用在后文有更多测试，将在后文讨论。

测试对象：

native：原生程序

wasm aot： AOT编译后的Wasm程序

docker native： 使用docker容器执行和native一样的Linux程序

测试目的：Wasm运行时的启动速度和容器启动速度比较

测试程序： binarytree.c （深度为0二叉树遍历）

运行时： WasmEdge

序号	项目	测试命令	执行时间(s) 单次执行	内存占用(MB)	容器内存占用(MB)
1	wasm aot	\time –v –o tmp3 wasmedge ./biwasm.wasm 0	0 （小于10ms）	17.62	/
2	docker image only	\time –v – docker run –i conkara:v1 CMD [”/binarytree”,”0”]	0.58	/	35.56

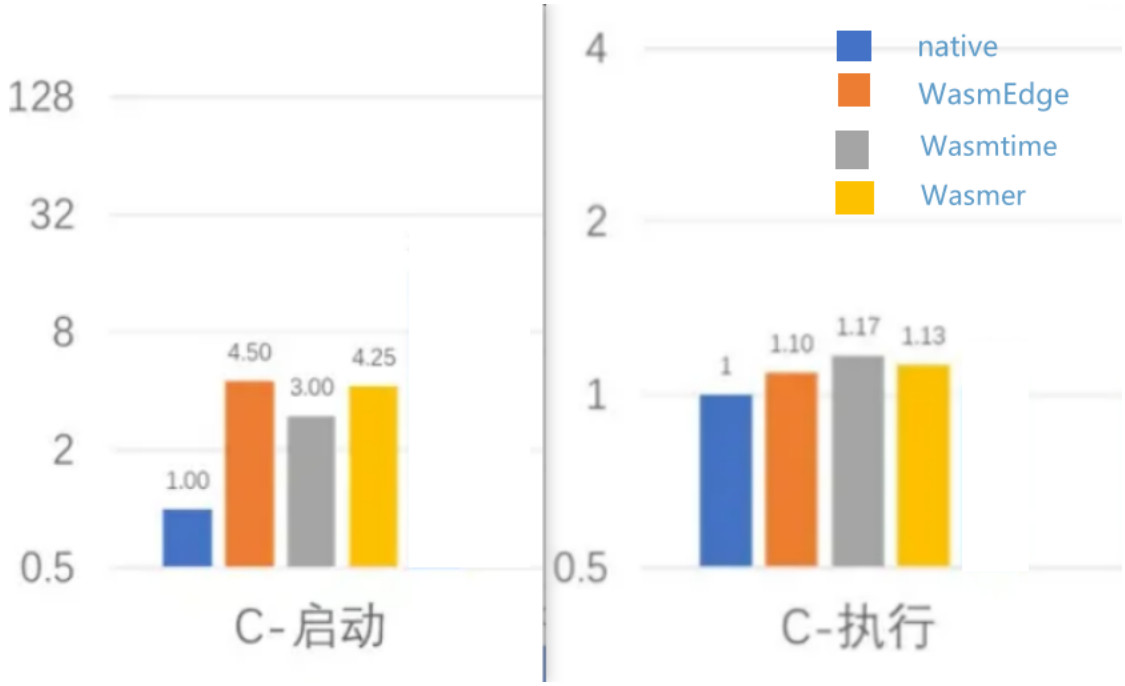
上面的测试可以看出，容器的启动时间大致在580ms，而WasmEdge的启动时间小于10ms（因此显示为0），由此WasmEdge在启动速度方面比容器快100倍的结论可以取信。

2.2 主流wasm运行时之间性能对比

2.2.1 主流Wasm运行时性能调研

调研的结果是在执行速度上，WasmEdge的AOT编译以后的速度是最快的， 其次是Wasmer，最后是Wasmtime。Wasmer和Wasmtime的执行速度多数情况都是比较接近的，但是Wasmtime的表现要比Wasmer要稳定。此外调用到的比较的数据多是AOT编译以后的，对于直接执行的情况是缺失的。因此后面的评测会关注到直接执行方式下的比较。

通过调研，找到已有的测评文章“4种主流WASM Runtime的性能比较”，结果如下图。可以看出使用运行时的启动时间相对于原生程序要更多，是原生程序的3到4倍。WasmEdge和Wasmer启动时间接近，但二者比Wasmtime慢，启动时间是Wasmtime的1.5倍左右。执行时间方面，使用运行时的速度比较接近原生，运行时之间的执行速度差异也不大，但是该文章的作者使用的测试程序太少，后续的本文的测评将使用更多的测试程序以避免偶然性。



2.2.2 主流Wasm运行时性能测评–x86

执行速度

平台： x86

运行时： Wasmtime、WasmEdge 和 Wasmer

测试程序：binarytree （深度为20的满二叉树遍历）

序号	项目	执行时间(s) 单次执行
1	native	6.39
2	WasmEdge 直接执行	>100
3	Wasmer 直接执行	7.33
4	Wasmtime 直接执行	9.00
5	WasmEdge AOT后执行	6.83
6	Wasmer AOT后执行	7.38
7	Wasmtime AOT后执行	9.08

平台： x86

运行时： Wasmtime、WasmEdge 和 Wasmer

测试程序：n体模型程序 n_body

序号	项目	执行时间(s) 单次执行	内存占用(MB)
1	native	3.72	0.81
2	WasmEdge 直接执行	>100	/
3	Wasmer 直接执行	6.79	21.89
4	Wasmtime 直接执行	4.88	10.61
5	WasmEdge AOT后执行	4.41	17.19
6	Wasmer AOT后执行	6.83	21.57
7	Wasmtime AOT后执行	4.92	9.86

通过以上的测试数据，可以看出： AOT编译后的执行速度是WasmEdge 最快，接近原生的执行速度，而Wasmer和Wasmtime 二者性能在不同测试程序下表现不一，需要考虑更多的测试程序，但都比WasmEdge的性能要稍弱。

在直接执行的情况下，WasmEdge是Interpret执行，而Wasmtime和Wasmer都是JIT的方式执行。以JIT方式执行时先以解释执行的方式执行，然后对热点代码进行编译执行以提高执行速度。因此正如测试的数据所示，WasmEdge没有JIT优化，执行速度非常慢，而Wasmtime和Wasmer直接执行时由于使用JIT的执行方式，执行速度仅仅比AOT编译后的速度稍慢，总体性能优秀。

为了比较Wasmtime和Wasmer的性能，我们使用更多的测试测试程序进行测试：

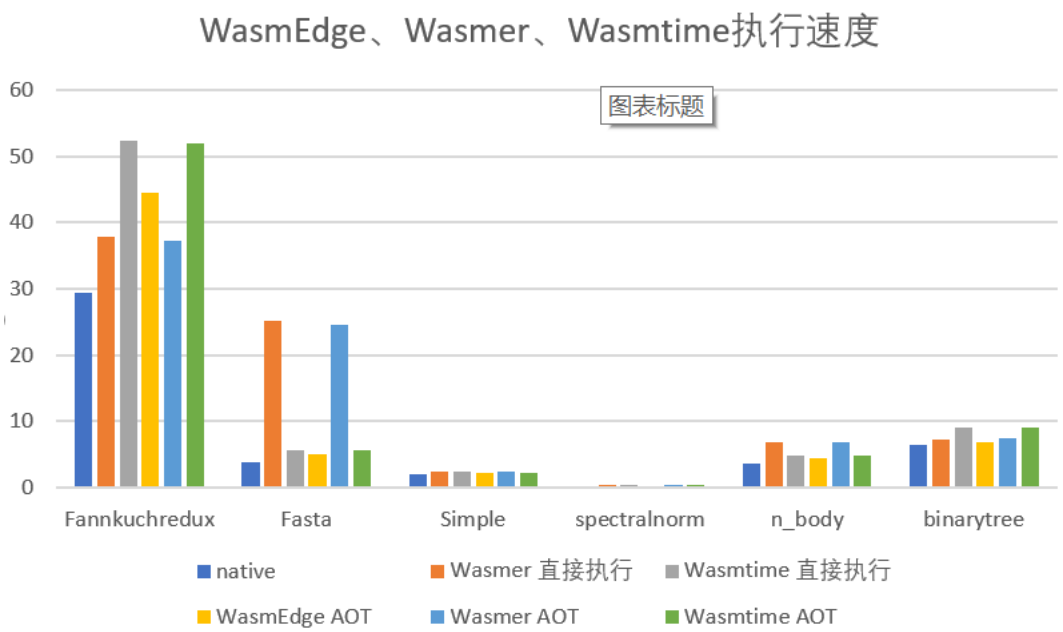
测试运行时： Wasmtime、Wasmer、WasmEdge

测试指标： 执行时间（s）

注： 由于WasmEdge 直接执行的速度很慢，超过100s，不放入表中

测试程序	native	Wasmer 直接执行	Wasmtime 直接执行	WasmEdge AOT	Wasmer AOT	Wasmtime AOT
Fannkuchredux	29.38	37.84	52.27	44.40	37.27	52.02
Fasta	3.76	25.09	5.72	4.99	24.51	5.65
Simple	2.13	2.50	2.34	2.16	2.52	2.30
spectralnorm	0.27	0.34	0.40	0.27	0.33	0.39
n_body	3.72	6.79	4.88	4.41	6.83	4.92
binarytree	6.39	7.33	9.00	6.83	7.38	9.08

上表的柱状图如下：



通过上面的几个测试程序，我们可以看出：

1. WasmEdge的AOT性能最好，在多数的测试程序上都拥有接近原生程序的速度。
2. 相比WasmEdge AOT，Wasmer和Wasmtime在AOT下的速度稍慢一点，WasmEdge的速度优势并不明显。
3. WasmEdge的直接执行速度特别慢，远超过其他运行时
4. Wasmer和Wasmtime在AOT和直接执行这两种方式上性能都相近
5. Wasmer在部分测试程序如Fasta，执行时间是其他运行时的10倍，出奇的慢

运行内存

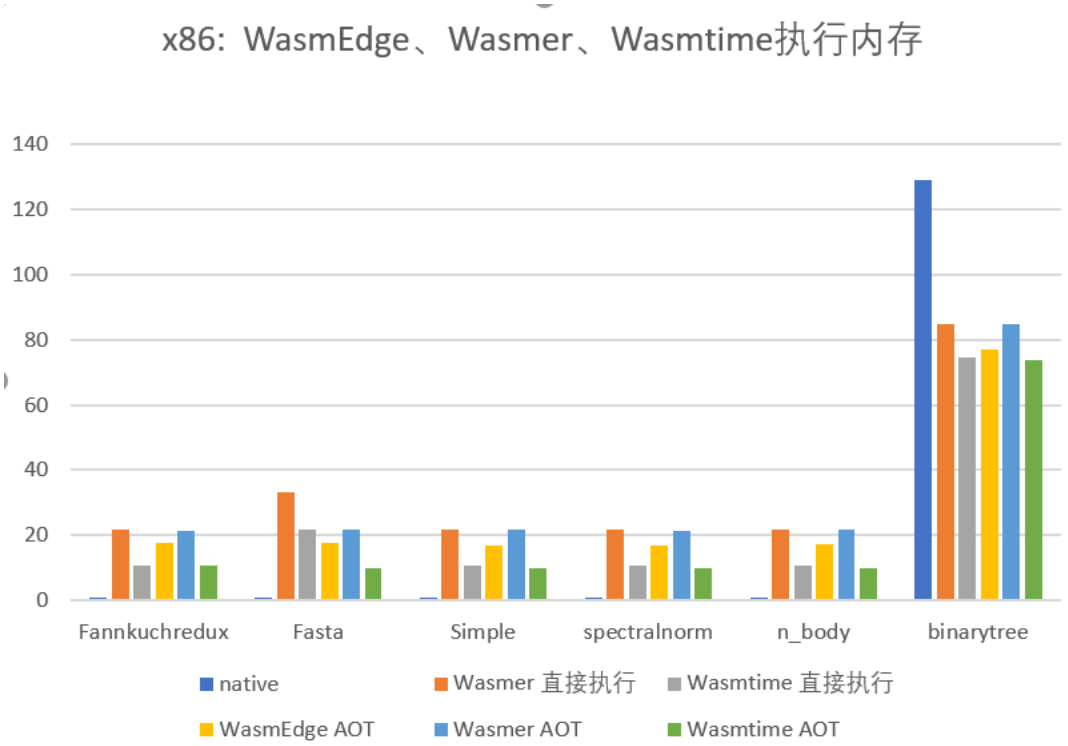
测试运行时： Wasmtime、Wasmer、WasmEdge

测试指标： 运行内存（MB）

注： 由于WasmEdge 直接执行的速度很慢，超过100s，不放入表中

测试程序	native	Wasmer 直接执行	Wasmtime 直接执行	WasmEdge AOT	Wasmer AOT	Wasmtime AOT
Fannkuchredux	0.81	21.83	10.73	17.50	21.53	10.73
Fasta	0.82	33.15	21.76	17.52	21.75	10.02
Simple	0.80	21.79	10.59	16.73	21.79	9.81
spectralnorm	0.86	21.79	10.60	16.97	21.51	9.97
n_body	0.81	21.89	10.61	17.19	21.57	9.86

binarytree	128.8	84.89	74.41	76.93	84.61	73.88



1. Wasmer和Wasmtime经过AOT编译以后，运行内存都会减少
2. 三个运行时的运行内存排序: Wasmtime < WasmEdge < Wasmer
3. 大多数时候，原生程序占用的运行内存都要比使用运行时要少，但是存在部分测试程序里，如binarytree，使用运行时的占用的运行内存要比其他原生程序少。

2.2.3 主流Wasm运行时性能测评–ARM

ARM 虚拟机配置: qemu–system–aarch64

内存： 4096MB

模拟CPU： cortex–a57

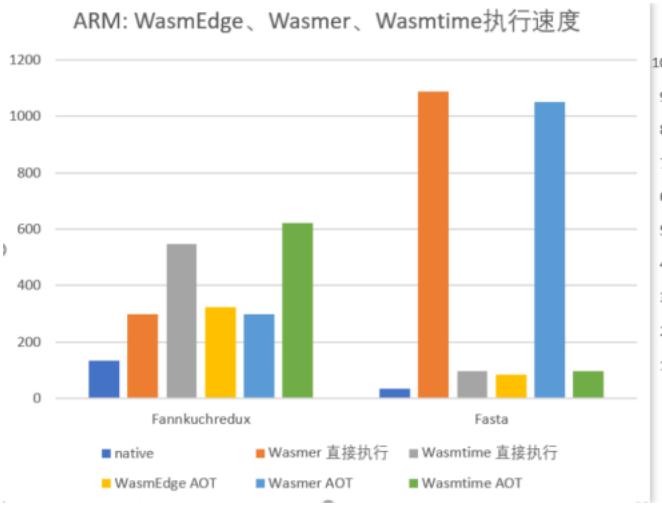
平台： ARM

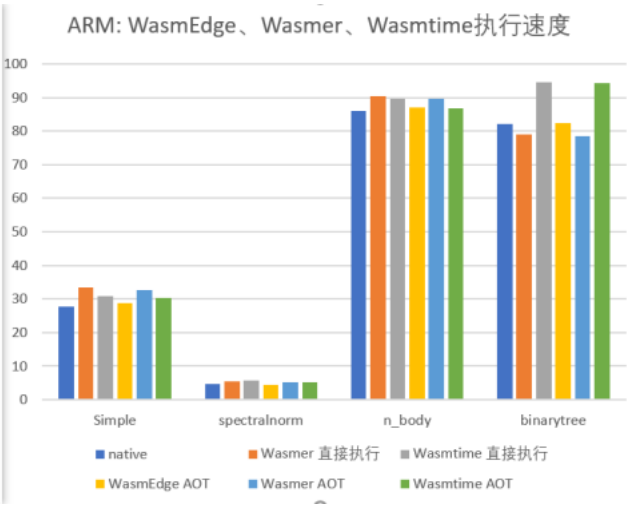
运行时： Wasmtime、WasmEdge 和 Wasmer

测试目标：执行速度（s）

测试程序	native	Wasmer **直接执行**	Wasmtime **直接执行**	WasmEdge AOT	Wasmer AOT	Wasmtime AOT
Fannkuchredux	133.79	300.02	545.92	324.97	300.12	620.76
Fasta	34.91	1087.65	97.55	84.62	1049.35	96.82

Simple	27.81	33.42	30.72	28.71	32.75	30.31
spectralnorm	4.64	5.56	5.73	4.53	5.08	5.27
n_body	86.01	90.38	89.70	86.91	89.67	86.68
binarytree	82.07	78.98	94.43	82.48	78.39	94.35





1. 在ARM虚拟机上进行的测试，时间明显比在x86虚拟机上的时间长，但是由于配置的不同，我们无法说明在ARM上的Wasm程序性能要比x86平台差
2. 和x86平台上一样，WasmEdge的性能仍然接近原生程序，Wasmer和Wasmtime性能仍是相当的
3. 三个运行时经过AOT编译后速度都比AOT编译前要快
    a. WasmEdge的直接执行方式速度远比AOT后的速度要慢10倍以上
    b. Wasmer和Wasmtime的直接执行（JIT）速度比较快，但经过AOT编译以后速度提升不明显
4. Wasmer的性能在部分程序如：Fasta，会出现比其他运行时慢10倍以上的情况，相对于Wasmtime而言性能不稳定。

运行内存

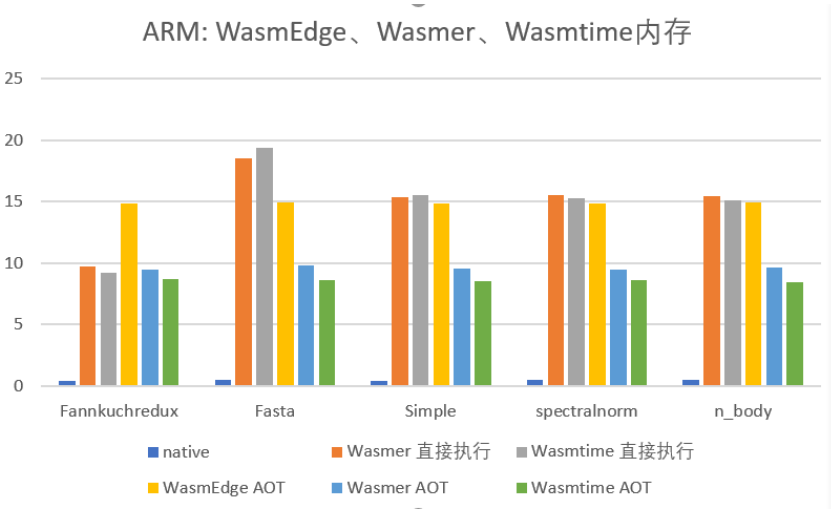
测试运行时： Wasmtime、Wasmer、WasmEdge

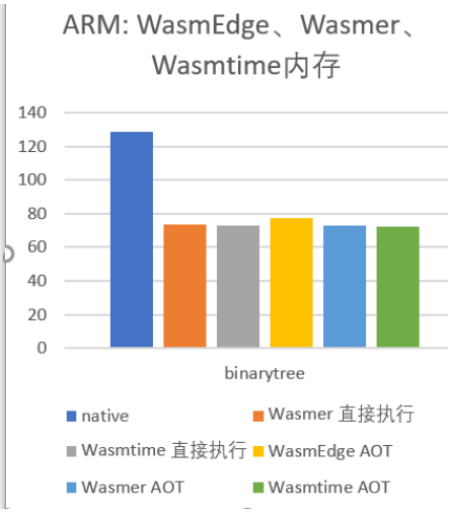
测试指标： 运行内存（MB）

注： 由于WasmEdge 直接执行的速度很慢，超过100s，不放入表中

测试程序	**native**	**Wasmer 直接执行**	**Wasmtime 直接执行**	**WasmEdge AOT**	**Wasmer AOT**	**Wasmtime AOT**
Fannkuchredux	0.47	9.69	9.21	14.86	9.51	8.67
Fasta	0.54	18.52	19.36	14.95	9.79	8.59
Simple	0.47	15.36	15.57	14.86	9.56	8.53
spectralnorm	0.52	15.52	15.29	14.86	9.50	8.58

n_body	0.48	15.47	15.13	14.93	9.61	8.47
binarytree	128.48	73.33	72.91	77.62	73.17	72.45





1. 和原生程序相比，使用运行时明显需要更多的内存，大致是原生程序的15倍左右
2. Wasmer和Wasmtime经过AOT编译后，占用内存减少约30%
3. 在直接执行情况下，Wasmer和Wasmtime占用的内存接近。
4. 在AOT情况下，WasmEdge要比其他两个运行时的1.5倍左右。

三、总结

第一章中我们先介绍了wasm轻量、高性能、沙箱安全和高可移植性的主要特点，通过例子深入阐述了wasm基于open-at的系统调用方式和通过WASI实现的高可移植性。接着我们讨论了wasm在x86平台和ARM平台上的性能表现，得出ARM在成本上更具性价比的结论。为了顺利使用wasm进行开发，我们调研和试用了VS code进行wasm开发，确定VS code具备调试所需的工具。

第二章围绕wasm运行时的性能进行，分成了wasm与容器的性能对比和wasm运行时之间的性能对比两个主题。调研和测试的结果都表明wasm和容器相比，wasm在启动速度上有100倍的优势，运行速度则有20%左右的优势。运行内存方面，wasm运行时相比原生程序需要更多的内存空间，但仍比容器更有优势。wasm运行时之间的对比，通过调研和测试我们得出wasmEdge的性能比其他两个主流运行时要更优。最后，我们在ARM和x86两个平台都执行相同的测试，得到了基本一致的结论。

为弥补前面测试的不足，后续计划进行的更多测试。对于测试数据中表现出来的特殊现象进行调研，如：Wasmer在部分测试程序中执行时间是其他运行时的10倍。