

02 | 历史篇：为什么会有 WebAssembly 这样一门技术？

2020-09-07 于航 来自北京

《WebAssembly入门课》



你好，我是于航。

在开始真正学习 Wasm 这门技术之前，我想先来问你一个问题：你有没有思考过，在 Web 技术的历史发展长河中，为什么会出现 Wasm 这样一门技术？现有的这些 Web 技术，又存在着哪些无法解决的问题？

要知道，所有新兴技术的诞生都一定有它存在的意义，或者要去解决的问题。比如 jQuery 之于浏览器的兼容性、Vue.js / React.js 之于 Web 应用的构建模式。

虽然用前端框架和库来类比 Wasm 不算十分合适，但我想阐述的是，Wasm 的出现也并非偶然。在这节课的内容中，我们就来一起看看 Wasm 诞生背后的那些故事。相信在学习完本课程后，你会对 Wasm 有了一些新的了解。而这些了解有时可能比一项技术本身更加重要。

JavaScript 的发展和困境

1995 年末，Brendan Eich 仅用了 10 天时间便发明出了 JavaScript 编程语言，而在随后的二十多年中，JavaScript 已经成为了不可动摇的，用于开发 Web 前端应用的必备编程语言之一。

不仅如此，随着后来诸如 React Native、Electron 以及 Vue.js 等各类框架的不断涌现，JavaScript 曾经一度成为 GitHub 语言排行榜的年度冠军。JavaScript 也因此被广泛应用到了各行各业、各个领域的各类项目中。

虽说 JavaScript 的应用场景如此广泛，但也会有它自己的烦恼。下面我们就从 Web 应用层面以及 JavaScript 语言本身，来看看它究竟在愁些什么。

Web 应用规模的急速增长

随着移动互联网的发展和各种形式经济活动的不断展开，运行在浏览器中的各类 Web 应用，它们的体积与复杂性随着时间的推移在不断发展。为了能够在浏览器中高效运行这些不断“变大”的 Web 应用，浏览器厂商们也在不断地寻求着各种“黑科技”来优化浏览器的性能。

但与日益庞大和复杂化的 Web 应用相比，浏览器对自身性能优化可谓举步维艰。不难预见，当“复杂化”与“性能优化”的速度之比不断变大时，迟早有一天，浏览器会再也无法支撑起这些庞大 Web 应用的运行。

据相关数据统计，截止 2019 年底，全世界一共有约 16 亿个可索引网页，而其中的 95% 都在使用 JavaScript。在这些网页中，大约有 45% 的网页创建于最近 5 年。而 2015 年，ECMAScript 2015 (ES6) 诞生，也标志着 JavaScript 开始进入了标准一年一更新的节奏中。

现代的大多数网页，都会使用较新的 JavaScript 语法标准进行开发，然后在发布时使用诸如 Babel 等工具，将这些新的 JavaScript 语法转换为对应的 ES5 旧版本语法，来兼容旧版本浏览器。但这样做，产生的各类 Polyfill 代码，会极大地增加整个 Web 应用的体积。

同时，在 Web 应用的实际运行过程中，大量的 JavaScript 代码也会降低应用的整体运行效率。Twitter 曾尝试直接以 ES6+ 版本代码的形式，来发布整个 Web 应用。通过这种方式所减少的 Polyfill Bundle 文件的大小，竟然可以达到应用所使用的全部 JavaScript 代码的 83%。

JavaScript 的弱类型之殇

除了上面我们讲到的，浏览器性能优化与 Web 应用规模日益增大，这两者行进速度的“不协调”所可能带来的问题之外，JavaScript 语言本身也有着其自身的“弱点”。而由于这些“弱点”所带来的妥协，使得浏览器在面对庞大的 Web 应用时，也会显得力不从心。

可以说，JavaScript 是一个“动态类型”的编程语言。在实际编码过程中，我们不需要为每一个变量指定对应类型。变量具体类型的推导过程，会被推迟到代码的实际运行时再进行。JavaScript 这种动态类型语言所独有的特性，在某种程度上相较于静态类型语言而言，会带来额外的运行时性能开销。

下面我们来一起想象一下，JavaScript 引擎在执行表达式 “ $x + y$ ” 时的具体流程。这里 x 与 y 分别是在一段 JavaScript 代码中定义的两个变量，当引擎执行到 “ $x + y$ ” 时，对于运算符 “+” 来说，位于其左右两侧的操作数可以是 JavaScript 中任何有效类型的组合，比如 “ $\{ \} + []$ ”、“ $[] + \text{null}$ ”、“ $1 + 2$ ” 等等。因此，引擎在对 “+” 运算符表达式进行求值时，会根据 ECMAScript 标准中规定的 “+” 运算符的语义，来对表达式进行求值。

通过下图你可以看到，在 ECMAScript 标准中定义的，“+” 运算符的运行时求值流程，实际上十分复杂和繁琐。这也是相对于静态语言来说，JavaScript 很少能够进行优化的地方。

在现代的 JavaScript 引擎中，尽管可以使用诸如 JIT 等技术来提高代码的执行效率，但在实际使用中，如果代码执行没有遵守 JIT 优化路径中特定 Guard 的要求，“去优化”的过程，也同样会影响引擎的整体执行效率。而这些影响都是由于 JavaScript 的“动态性”导致的。

12.8.3 The Addition Operator (+)

NOTE The addition operator either performs string concatenation or numeric addition.

12.8.3.1 Runtime Semantics: Evaluation

AdditiveExpression : *AdditiveExpression* + *MultiplicativeExpression*

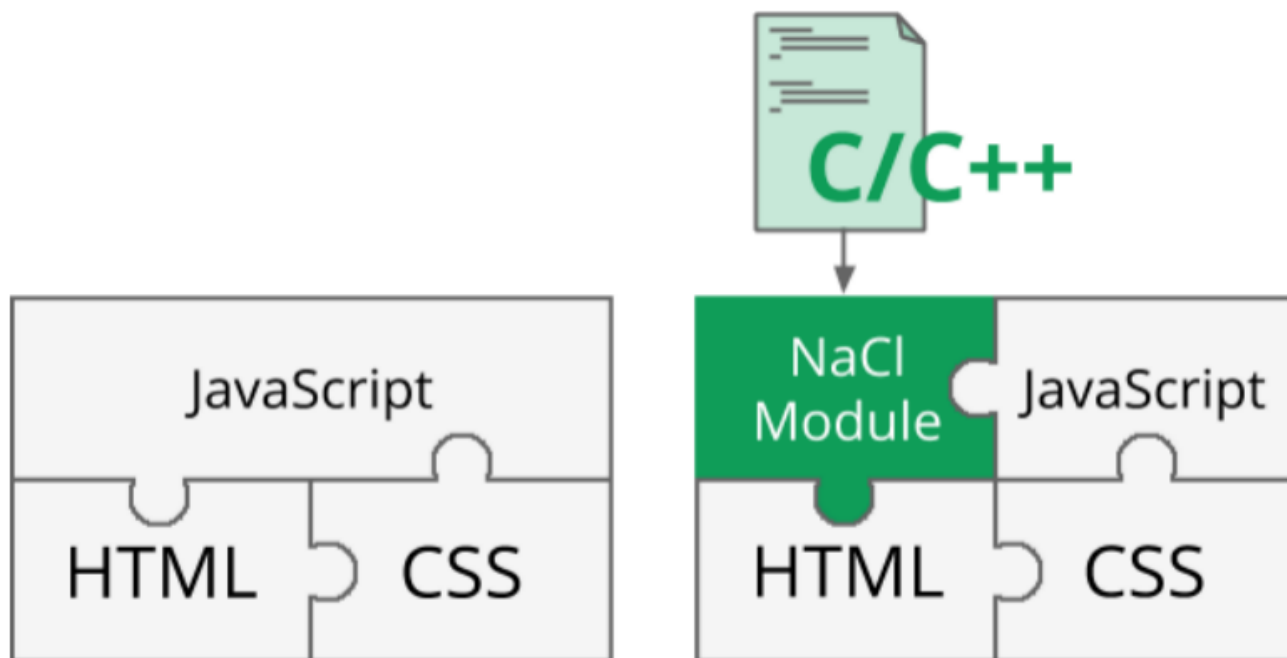
1. Let *lref* be the result of evaluating *AdditiveExpression*.
2. Let *lval* be ? *GetValue*(*lref*).
3. Let *rref* be the result of evaluating *MultiplicativeExpression*.
4. Let *rval* be ? *GetValue*(*rref*).
5. Let *lprim* be ? *ToPrimitive*(*lval*).
6. Let *rprim* be ? *ToPrimitive*(*rval*).
7. If *Type*(*lprim*) is String or *Type*(*rprim*) is String, then
 - a. Let *lstr* be ? *ToString*(*lprim*).
 - b. Let *rstr* be ? *ToString*(*rprim*).
 - c. Return the string-concatenation of *lstr* and *rstr*.
8. Let *lnum* be ? *ToNumeric*(*lprim*).
9. Let *rnum* be ? *ToNumeric*(*rprim*).
10. If *Type*(*lnum*) is different from *Type*(*rnum*), throw a **TypeError** exception.
11. Let *T* be *Type*(*lnum*).
12. Return *T*::*add*(*lnum*, *rnum*).

图片来自 ECMAScript@2020 官方标准文档

最初的尝试 —— NaCl 与 PNaCl

JavaScript 的发展困境在逐渐显现，人们对 Web 性能的担忧也在与日俱增，人们永远没有停下优化的脚步。NaCl 是由 Google 在 2011 年于 Chrome 浏览器中发布的一项技术，该技术旨在提供一个沙盒环境，可以让基于 C/C++ 语言编写的 Native 应用，安全地运行在浏览器中。NaCl 的全称 “Native Client” 也暗示了这一点。

如下图所示，一个标准 NaCl 应用的组成结构，与普通的 JavaScript Web 应用十分类似。NaCl 模块作为应用的一部分，主要用来进行复杂的数据处理和运算，JavaScript 则负责处理应用与外部用户的交互逻辑。NaCl 实例与 JavaScript 代码之间可以通过 “订阅 / 发布” 模型，来互相传递消息。



图片来自 Chrome 官方相关文档

理想虽好，但现实却存在着很多问题。通常，一个 NaCl 模块文件需要在开发者本地进行编译，然后才能够在浏览器中使用。而本地编译的模块文件通常仅含有架构相关（architecture-dependent）的代码，因此没有办法直接在其他类型的系统中使用。

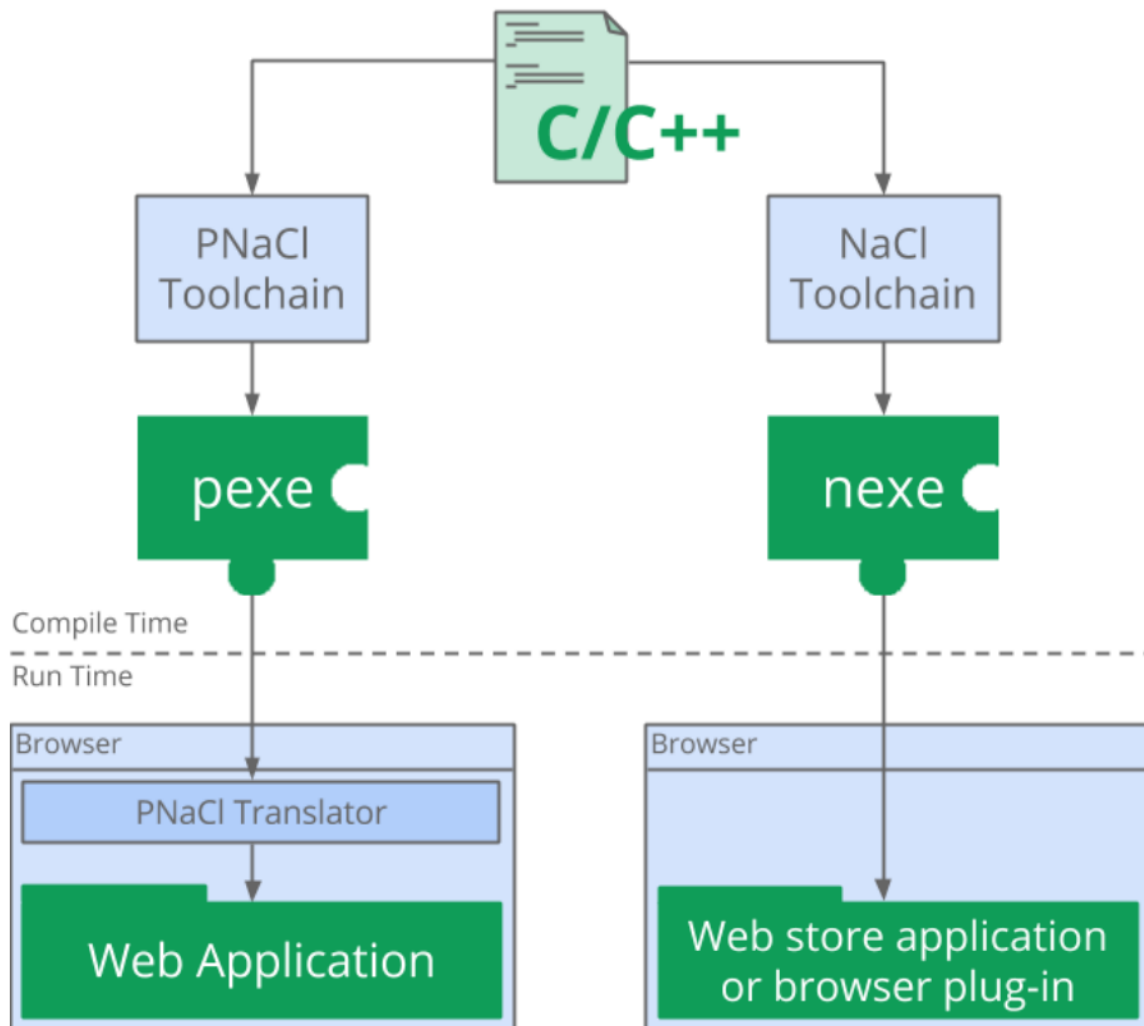
一个完整的 NaCl 应用，在分发时需要提供支持多个架构平台（X86_32 / X86_64 / ARM 等）的模块文件。浏览器在实际使用时，会根据当前系统的具体架构类型，来动态地选择，对应合适的模块文件进行使用。

不仅如此，由于 NaCl 模块“平台依赖”的特殊性，因此 NaCl 模块进行分发的过程，仅能够在 Chrome Web Store 中进行。另一方面，如果你想要将已经存在的 C/C++ 代码库编译至 NaCl，并在浏览器中使用，你还需要通过名为 Pepper 的库来对这些代码进行重写。

Pepper 提供了很多包装类型，以及用于和浏览器进行交互的 API，比如“PP_Bool”等。这些 API 和特殊类型可以便于整合传统 C/C++ 代码与 Web 浏览器的沙盒环境。

鉴于 NaCl 存在的“平台依赖”问题，Google 在后期又推出了名为 PNaCl 的技术。这里名字中多出来的“P”代表着“Portable”，也就是“可移植”的意思。

PNaCl 采用了不一样的生命周期，参考下图我们可以看到，相较于 NaCl 模块直接包含有平台架构相关的代码，PNaCl 将源 C/C++ 代码编译到一种中间代码。这些中间代码会在浏览器实际加载这个 PNaCl 模块时，再被转换为对应的平台相关代码。因此，对于 PNaCl 模块而言，分发的过程变得更加简单，且不用担心移植性的问题。



图片来自 Chrome 官方相关文档

不过，即使是对于 PNaCl 这类“可移植性”已经不再成为问题的技术而言，它们的面前还有很多“大山”难以逾越。比如：“需要使用 Pepper 重写 C/C++ 代码，标准较为封闭、仅 Chrome 浏览器支持”等等。

总而言之，无论是 NaCl 还是 PNaCl，它们都已经成为过去。现在，如果你再次回到 NaCl / PNaCl 在 Google 的官方文档网站，你会发现如下这样一段声明。Wasm 将会作为新一代的技术，接替并继续传承 Google 赋予给 NaCl / PNaCl 的使命。

WebAssembly Migration Guide

(P)NaCl Deprecation Announcements

Given the momentum of cross-browser WebAssembly support, we plan to focus our native code efforts on WebAssembly going forward and plan to remove support for PNaCl in Q4 2019 (except for Chrome Apps). We believe that the vibrant ecosystem around **WebAssembly** makes it a better fit for new and existing high-performance web apps and that usage of PNaCl is sufficiently low to warrant deprecation.

图片来自 Chrome 官方相关文档

Wasm 的前身 —— ASM.js

除了 NaCl 与 PNaCl，另一个不可不提的技术便是 Mozilla 于 2013 提出的 ASM.js。同前两者一样，ASM.js 的设计目标也是为了能够在 JavaScript 语言之外，为“构建更高性能的 Web 应用”这个目标，提供另外一种实现的可能。

“ASM.js 是 JavaScript 的一个严格子集。它是一种可用于编译器的目标语言，低层次且高效。该目标语言有效地为内存不安全语言（如 C/C++），描述了一个沙盒虚拟机运行环境。静态和动态验证相结合的方式，使得 JavaScript 引擎能够使用 AOT 等优化编译策略来验证 ASM.js 代码”。这是 Mozilla 官方给出的关于“ASM.js 是什么？”这个问题的解答。


乍一看这段解释，可能会有点抽象和复杂。但实际上，我们只需要知道两件事情。

第一，ASM.js 是 JavaScript 的严格子集。这也就意味着，对于一段 ASM.js 代码，JavaScript 引擎可以将它视作普通的 JavaScript 代码来执行，这便保障了 ASM.js 在旧版本浏览器上的可移植性。

第二，ASM.js 使用了 “Annotation (注解)” 的方式来标记代码中包括：函数参数、局部 / 全局变量，以及函数返回值在内的各类值的实际类型。

当 JavaScript 引擎满足一定条件后，便会通过 AOT 静态编译的方式，将这些被 Annotation 标记的 ASM.js 代码，编译成对应的机器码并加以保存。当 JavaScript 引擎再次执行（甚至在第一次执行）这段 ASM.js 代码时，便会直接使用先前已经存储好的机器码版本。因此，引擎的性能会得到大幅的提升。

对于一段标准 ASM.js 代码的具体组成形式，你可以参考下面给出的这段代码，以便有一个更加直观的印象。

 复制代码

```
1 function asm (stdin, foreign, heap) {
2     "use asm";
3
4     function add (x, y) {
5         x = x|0; // 变量 x 存储了 int 类型值;
6         y = y|0; // 变量 y 存储了 int 类型值;
7         var addend = 1.0, sum = 0.0; // 变量 addend 和 sum 默认存放了"双精度浮点"类型值;
8         sum = sum + x + y;
9         return +sum; // 函数返回值为"双精度浮点"类型;
10    }
11    return { add: add };
12 }
13
```

在这段 JavaScript 代码中，最为重要的是函数 “asm” 在其函数体定义开头处使用的 “use asm” 指令。这个指令将会在代码执行过程中 “告诉” JavaScript 引擎，当前这个函数体内的代码可以按照 ASM.js 代码，来进行相应的优化和处理。

实际上，上述这样的一个 JavaScript 函数，便定义了一个标准的 ASM.js 模块。模块内部可以通过 return 的方式，导出包含有若干内联方法的对象。这些方法可以在外部的 JavaScript 代码中进行调用。

在上述 asm 模块内定义的内联函数 add 中，我们在其开头的前两行代码通过 “x|0” 和 “y|0” 的方式，分别对变量 x 与 y 的值类型进行了标记。而这种方式便是我们之前提到的

ASM.js 所使用的 Annotation。

当 JavaScript 引擎在编译这段 ASM.js 代码时，便会将这里的变量 x 与 y 的类型视为 int 整型。同样的，还有我们对函数返回值的处理 “+sum”。通过这样的 Annotation，引擎会将变量 sum 的值视为双精度浮点类型。类似的，ASM.js 在标准中还规定了其他的诸多 Annotation 形式，可以将变量值标记为不同的类型，甚至对值类型进行转换。

为了确保上述的这样一个 JavaScript 函数，能够被当做一个标准的 ASM.js 模块进行必要的优化处理，JavaScript 引擎通常会在实际编译加载这些模块前，进行很多必要的检查验证工作。

因此，并不是说只要为函数添加了 “use asm” 指令，并且为使用到的变量添加 Annotation 之后，JavaScript 引擎就会通过 AOT 的方式来优化代码的执行。所以这也是为什么我们先前提到的，ASM.js 通常被作为一种可用于编译器的，低层次且高效的目标语言，而不是用于手写。

从过去到未来

时间来到 2015 年 5 月。Chrome 团队的 Ben 正在为 V8 设计一种新的 Prototype（原型），而另一位团队成员 Rosbery，正在为这种 Prototype 设计对应的字节码格式。实际上，这个 Prototype 和对应的字节码格式，便是如今 Wasm 所分别对应的 WAT 可读文本格式与二进制字节码格式。在当时的谷歌内部，这两部分暂时被称为 ml-proto 与 v8-native-prototype。

随着 V8 团队对 ml-proto 与 v8-native-prototype 的不断修改和优化，它们最终便成为了 Wasm 早期标准的一部分。与此同期出现的，还有一个名为 “sexpr-wasm” 的内部工具，在当时这个工具用于对这两种格式进行相互转换。随着 Wasm 的标准化，它也同样成为了 Wasm 常用调试工具的一部分，这也就是我们所熟知的 —— WABT。

Chrome V8 团队作为参与过 PNaCL 与 ASM.js 这两个标准制定的团队，在设计和实现 Wasm 时也同样参考了很多从这两种技术中总结下来的优缺点。而这些经验也将会帮助 Wasm 做好准备，避开那些曾经走过的坑。最后，这些经验使得 Wasm 能够以一种更好的方式，展现在人们的面前。

总结

好了，讲到这，今天的内容也就基本结束了。最后我来给你总结一下。

实际上在 Wasm 真正出现之前，人们就已经开始尝试探索各类新型技术以赋予 Web 应用更高的运行效率。

从 NaCl、PNaCl 到 ASM.js，它们主要有三点共同特征：

1. 源码中都使用了类型明确的变量；
2. 应用都拥有独立的运行时环境，并且与原有的 JavaScript 运行时环境分离；
3. 支持将原有的 C/C++ 应用通过某种方式转换到基于这些技术的实现，并可以直接运行在 Web 浏览器中。

Wasm 这项技术的设计与实现，离不开从这些“前辈”们身上学习到的经验。从表面上来看，互联网技术迭代飞快。但实际上，当稍微深入和总结之后，你就会发现其实它们都有着基本相同的，想要去解决的目标问题，比如对于性能的执著要求。以及十分类似的技术解决方案，比如尽最大可能去确定那些能够确定、不会发生变化的部分（比如类型），然后再以此为基础进行优化。Wasm 也不例外。

课后思考

最后，我们来做一个思考题吧。

你觉得就目前的 Web 技术领域而言，存在着哪些困境？或者说需要去解决和优化的地方？

今天的课程就结束了，希望可以帮助到你，也希望你在下方的留言区和我参与讨论，同时欢迎你把这节课分享给你的朋友或者同事，一起交流一下。

精选留言 (14)



Jupiter

2020-09-08

老师好，Typescript 可以设置参数类型，现在的项目中也经常使用，但是最后Typescript的代码也会被解释成Javascript, 所以Typescripts是不是只是规范程序员写代码，对于应用的性能其实没有什么帮助？

作者回复: 你可以参考一个名为 AssemblyScript 的项目。这个项目就是基于 TS 语法设计的，可以将以 TS 语法编写的代码编译为 Wasm 格式。通常来讲，在日常的 Web 开发中，由于 TS 需要被编译为 JS，所以对于最终的代码执行性能是没有什么影响的。但 TS 的类型信息在对于编译到 Wasm 来说，还是可以被参考的。



👍 9



Cryhard

2020-09-08

ES6等后续版本如果解决了浏览器兼容性问题，后续不再需要“编译”回老版脚本语言，从而获得了性能保障。这样的情况是否会成为wasm的重大阻碍呢？

作者回复: 直接运行 ES6 代码的尝试其实有很多企业很早就在尝试，但大家都相对比较保守，还是会去兼容老一代的浏览器。就性能提升来讲，就算是直接运行 ES6 的代码，JavaScript 引擎也同样还是需要经历生成 AST, Profiling, Lowering, 以及 Optimization 和 De-optimization 的过程，因此相较于 Wasm 在整个引擎 Pipeline 中更加靠后的位置，性能上也还是无法比拟的。况且性能也只是 Wasm 的一方面特性，另一个“历史代码可复用性”的特征我们也同样需要关注。因此其实两者不会有太多的重叠，更不会阻碍 Wasm 的发展。



👍 8



arch

2020-09-08

wasm纯cpu计算和native差别不大，但是一旦涉及到访存就麻烦了，因为wasm只能访问vm内的内存。这就和应用逻辑非常相关了。典型就是protobuf message操作导致读写放大。

共 1 条评论 >

👍 4



Vfeelit

2021-01-13

JS虽然动态，在TS把持下，jit 不进行或几乎不进行反优化，性能已经相当不错了吧。wasm不也通过vm转一圈最终产生机器码？

作者回复：实际上还是有很大差别的，借助于 TS 确实可以解决“去优化”的问题，但对于 JS 还是要通过：生成 AST、生成 IR、Lowering、优化编译器、Profiling、生成机器码这些步骤的。但对于 Wasm，仅需要“生成机器码”这一步基本就够了，大部分优化在静态编译时就已经完成了，引擎只需要对照 Wasm 的 ISA 翻译成目标机器的 ISA 就可以了，基本上只是一个汇编器的作用，而且目前也不需要静态链接。



👍 3



fa

2020-09-11

阻碍在于苹果允不允许我在iOS上使用，目前有人在做基于wasm的热更新，别像jspatch一样给封了



👍 2



Vfeelit

2021-01-13

实现wasm的是虚拟机，任何语言，编译为某个中间语言并不是很困难的事。类似基于jvm，它之上除了java还有其他语言，虽然都是基于jvm，但生态各不同。wasm 相比下不过就是一套vm规范，没有那么大想象力啊

作者回复：就本质而言的话确实是这样，类似 JVM 只是一套新的 V-ISA 标准。但目前来看其不同在于：

- 1、Wasm 可以用于 Web（起源）；
- 2、WASI 提供了一套 out-of-web 的操作系统接口标准，并且基于 capability-based security 可以提供更加安全上层应用实现（本身基于 Wasm）。无论是安全的代码复用，还是基于 VM 层的轻量级沙盒环境，都是很有价值和发挥空间的。



👍 1



小炭

2020-10-19

我靠，既然没看懂，看来要多读几遍。



👍 1



别力的灯有点弱
2020-09-08

这个技术足以改变世界 + + +



1



donglin

2022-10-11 来自北京

文中说ASM.js会被编译成机器码保存，下次再次执行时便会直接使用已保存的机器码，从而提高性能。

这个不就是JIT机制嘛，但在iOS上是禁止的啊，所以ASM.js在iOS上也无法提高性能？



Carpe

2022-09-28 来自四川

“据相关数据统计，截止 2019 年底，全世界一共有约 16 亿个可索引网页，而其中的 95% 都在使用 JavaScript。”数据来源于哪？



地球外地人

2020-09-17

老师好，感觉ts代码虽然被编译成了js，但是转换后的js代码内部也避免了很多类型转换的问题，js引擎对于这一部分没有什么正反馈吗

作者回复: 会有一部分。由于 TS 的类型信息并没有给到 JavaScript 引擎来使用，所以即便整个程序是 well-organized 的，但是 JavaScript 引擎仍然还是需要通过运行时推导来推测出变量的具体类型。但也有的好处是，JIT 在假设某个变量的类型时，基本上都可以保证是正确的。这样引擎的去优化过程就会减少很多，性能会相对提高。



郭纯

2020-09-14

在 web vr 领域下 场景中会设计到很多模型和特效如果大规模进行平移 旋转 缩放我感觉很适合 wasm 场景.

作者回复: 嗯嗯是的，不过现在有 WebXR 这个 W3C 的草案了。如果浏览器能够 ship，就会有专门的 Native API 来用于 Web VR 应用的开发。不过目前看还为时尚早，Wasm 可以作为次级兼容的一个优

秀备选方案。



浩明啦

2020-09-11

评论里老师说AssemblyScript这个，我刚刚扫了一眼，我们现在ts的项目也能用这个来编译用到生产环境上么？不过很多的那些react 和vue框架打包后为啥不直接编译wasm呢？

作者回复: AssemblyScript 其实只是基于 TS 的语法而已，实际在应用时需要进行一些改动以适配 Wasm 标准以及浏览器上的运行时环境，所以直接编译前端的 TS 项目应该是不行的。现代前端开发框架由于需要与 Web API 打交道，因此实际上在编译到 Wasm 后，对框架的性能提升暂时还看不到可观的收益。关于前端框架与 Wasm 之间的关系，可以关注课程后面的文章哈。



余文郁

2020-09-07

老师，ecma标准文档和Chrome官方文档地址可以分享一下吗

作者回复: EMCA: <https://www.ecma-international.org/ecma-262/>

NaCl/PNaCl: <https://developer.chrome.com/native-client/nacl-and-pnacl>

