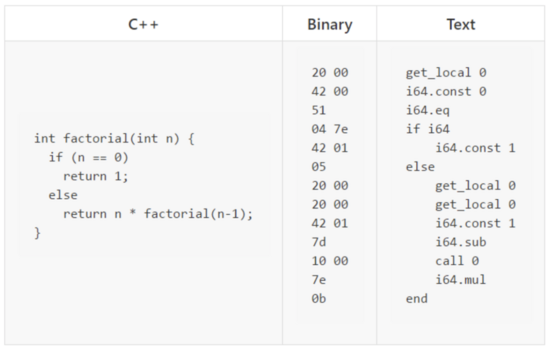
WASM技术浅析

概述

--------------------------------------------------------------------------------------------------------------------------------

WebAssembly（WASM）是由主流浏览器厂商组成的[W3C社区团体](https://www.w3.org/community/webassembly/)制定的一个二进制指令格式。

WASM的目标是定义一个可移植、体积小、加载快并且兼容 Web 的全新格式，此二进制格式文件将可以在各种平台（包括移动设备和物联网设备）上被编译，然后发挥通用的硬件性能以原生应用的速度运行。



上图分别是c++,wasm,wast(wasm的可读形式)

关键词：

可移植：WebAssembly是一种可移植的二进制格式，它不依赖于具体的浏览器平台。

高效：WebAssembly被设计为针对Size和Load Time进行优化的格式，可以在各个硬件平台上以native speed运行。

安全：WebAssembly是运行在沙盒内的，甚至可以和当前的JavaScript虚拟机共享一套环境，并且也遵守浏览器各种跨域不跨域的规章制度。

开放：WebAssembly是开放标准，不受某一家厂商控制。并且被设计为可以和JavaScript API和Context交互。

诞生背景及机制介绍

--------------------------------------------------------------------------------------------------------------------------------

JS语言原本只是作为辅助java的脚本语言设计出来的，本身存在很多问题。

首先JS作为一种解释型语言，存在较大的性能瓶颈。JIT(Just-In-Time Compilation,即时编译)技术的出现，将JS运行速度提升了几十倍（Google的V8引擎）。

然后是JS作为一种动态类型语言，语法太过灵活，一是用来开发大型web项目比较困难，另一方面也导致JIT编译出的机器码复用率低，反复重编产生很大开销，或是难以编译如for in这样的代码，使Web代码的运行速度和原生代码相比仍然有较大差距。

于是产生了给JS进行类型标注的思路，比如微软的TypeScript，Google的Dart，FireFox的asm.js。前二者是可以转换为JS的强类型语言，后者是JS的子集，要求对JS变量进行类型标注使其完全确定。

asm.js的标注方式如下所示：

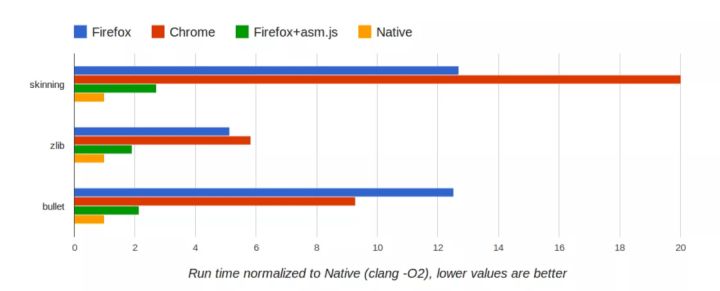
var a = 1;

var x = a | 0; // x 是32位整数

var y = +a; // y 是64位浮点数

由于asm.js是强数据类型，可以AOT（Ahead-Of-Time Compilation，编译在执行之前，执行时不用做任何编译处理），比JIT性能提升空间更大，得到了各大浏览器开发商的认可。

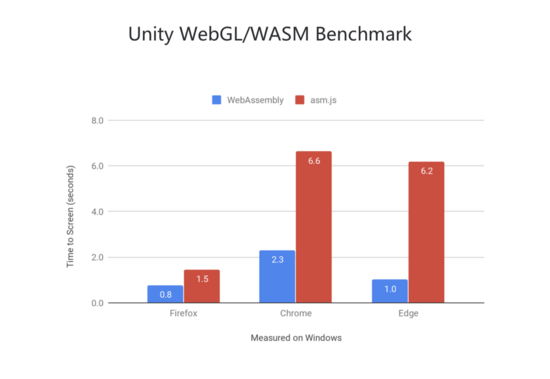
下图是Mozilla给出的asm.js的benchmark:



在asm.js基础上，Google、Microsoft、Mozilla、Apple等主流浏览器开发商决定更进一步，建立一个统一的标准，就是WASM。

WASM的机制与asm.js类似，但是更为激进。它不再处理JS变量标注，而是直接给浏览器提供AST（Abstract Syntax Tree抽象语法树，可以快速编译为机器码）。只有对不支持WASM的浏览器，才通过polyfill技术转换为JS代码执行。以此解决JS语言层面的固有问题，进一步提升Web性能，同时也避免了各大开发商自成体系，互不兼容的问题。

下面的图是Unity WebGL使用和不使用WASM的benchmark对比:



当然，以上benchmark都是针对一些特定的计算密集型的代码跑出的结果，对于普通的Web页面提升并没有那么明显，因为性能瓶颈在其他地方，并且在JS和WASM相互调用也需要一定开销，下面说明一下WASM如何运行。

实践

--------------------------------------------------------------------------------------------------------------------------------

如前所述，WASM目标是一种通用的二进制格式，它是可以由LLVM IR（Intermedia Representation，中间表示）直接生成的（LLVM是一个非常有名的编译器框架，关于LLVM的内容本文不做详述，有兴趣可以去自行了解，官网：<http://www.llvm.org/>）。因此理论上所有支持被编译为LLVM IR的各种语言代码，都可以被编译为WASM，例如C/C++,Rust,Swift等，都已经有不少应用。如下图所示：



以C++为例，可以使用工具Emscripten将C++代码生成WASM，具体的操作流程网上教程很多，参考以下这篇:

[**https://mp.weixin.qq.com/s?\_\_biz=MzAxNDEwNjk5OQ==&mid=2650404876&idx=1&sn=713633ee025f1a12a0ea657d28af1e91&chksm=83953014b4e2b902388d2f144ae348bae5ba1700ea4c7c71bb867a4a143b399340ede018db7d&scene=27#wechat\_redirect**](https://mp.weixin.qq.com/s?__biz=MzAxNDEwNjk5OQ==&mid=2650404876&idx=1&sn=713633ee025f1a12a0ea657d28af1e91&chksm=83953014b4e2b902388d2f144ae348bae5ba1700ea4c7c71bb867a4a143b399340ede018db7d&scene=27#wechat_redirect)

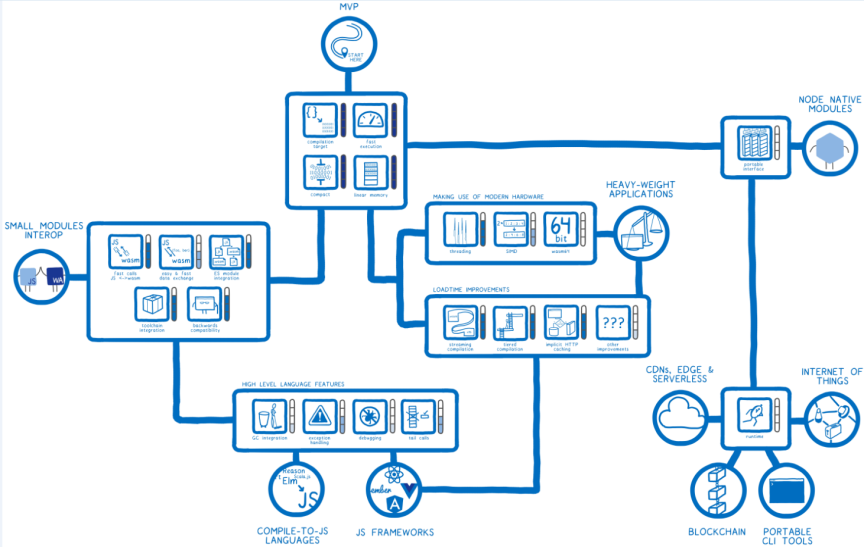
另外值得一提的Rust语言，它最初就是由Mozilla研究院设计的，开发者在编写浏览器内核过程中积累了很多经验，最终帮助他创造了Rust。Rust的定位就是“现代化的C++”，同样高性能无额外代价，并且有更安全的内存管理，十分适合开发一些对性能和安全性都有较高要求的系统模块。而它也是高级语言中对WASM支持最好的语言，自带的工具链非常完善,将Rust用于开发一些高性能Web应用模块可以说是顺理成章。目前市面上用到了WASM技术的，使用Rust编写的占了很大比例。Rust编译为WASM可以参考这篇文章：

<https://developer.mozilla.org/zh-CN/docs/WebAssembly/Rust_to_wasm>

发展现状

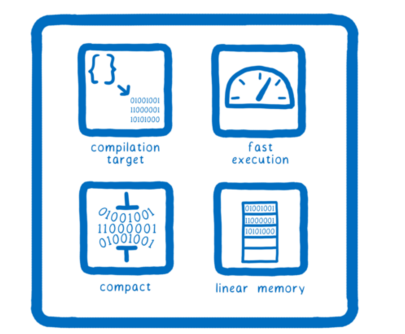
--------------------------------------------------------------------------------------------------------------------------------

官方发布的WASM进度图，对WASM已经实现的功能和未来的规划有详细描述，强烈建议阅读原文，不管是对了解WASM的机制，还是这项技术的目标和远景都有很大帮助：<https://hacks.mozilla.org/2018/10/webassemblys-post-mvp-future/>



上图中每个图标右侧为该项技能的开发进度，由于此文是18年发布的，实际上某些功能已经有了新的进展。下面对此图做个分块简介。

MVP(Mininum Viable Product)，最小可用产品，1.0版本已经实现的功能：



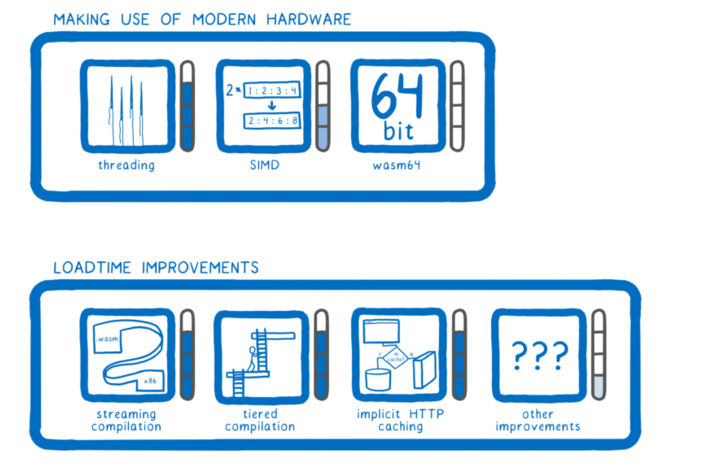
编译到二进制目标文件

快速运行

较小的尺寸:减少模块加载时间

线性内存:在javascript中运行时，获取一块线性内存供WASM模块自行管理。

在Web上使用重量级桌面应用：



多线程支持

SIMD: Single instruction multiple data，单指令计算多组数据

64位地址

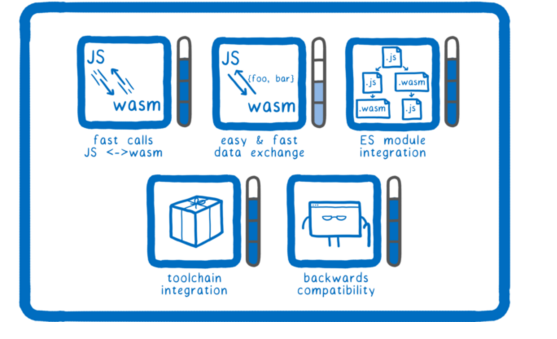
流式编译:一边加载一边编译

阶梯式编译：先快速编译一个可运行版本，再编译一个性能更高的完整版本

内部Http缓存：缓存编译结果进行复用

其他

与JS交互：



JS和WASM快速地相互调用

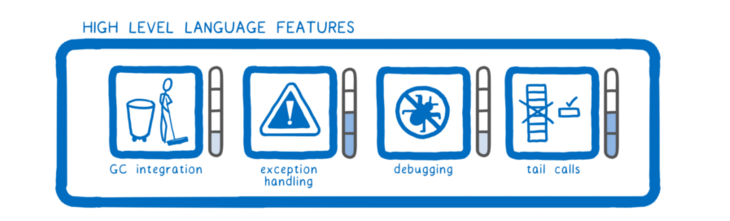
JS和WASM快速地进行数据交换：目前两者互相传输数据还不够方便，导致WASM更适合实现一些独立的计算模块，这块还有优化空间。

ES Module集成

工具链集成

向后兼容：兼容之前版本的浏览器

JS框架以及编译到JS：



GC：JS和WASM共享对象并对其进行管理

异常处理

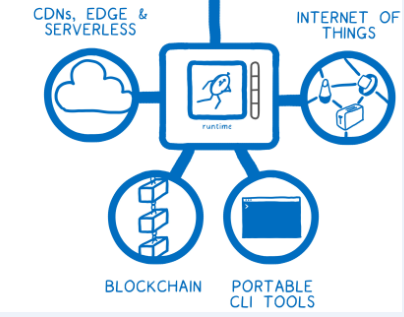
调试：已有一些浏览器工具可调试，还需完善

tail calls

此外，WASM还可用于浏览器之外的应用，首先需要实现一套可移植的通用接口和WASM的运行时。

如此，则可以展望WASM作为一套跨平台的通用二进制实现规范，应用于这些场景：



CDN，边缘计算，无服务器：将计算单元用WASM模块的方式直接上传使用

智能手表，嵌入式设备等等

区块链应用

跨操作系统的通用命令行工具

应用现状

--------------------------------------------------------------------------------------------------------------------------------

目前WASM最常见的使用场景是这两种情况：

1.对性能要求较高的Web模块，比如音视频信息处理，加解密计算，区块链，网页游戏等

2.将原生应用搬上浏览器，比如PhotoShop，甚至windows (https://bellard.org/jslinux/vm.html?url=https://bellard.org/jslinux/win2k.cfg&mem=192&graphic=1&w=1024&h=768)

白鹭引擎使用WASM的经验总结：

<https://my.oschina.net/u/3695009/blog/1610888>

