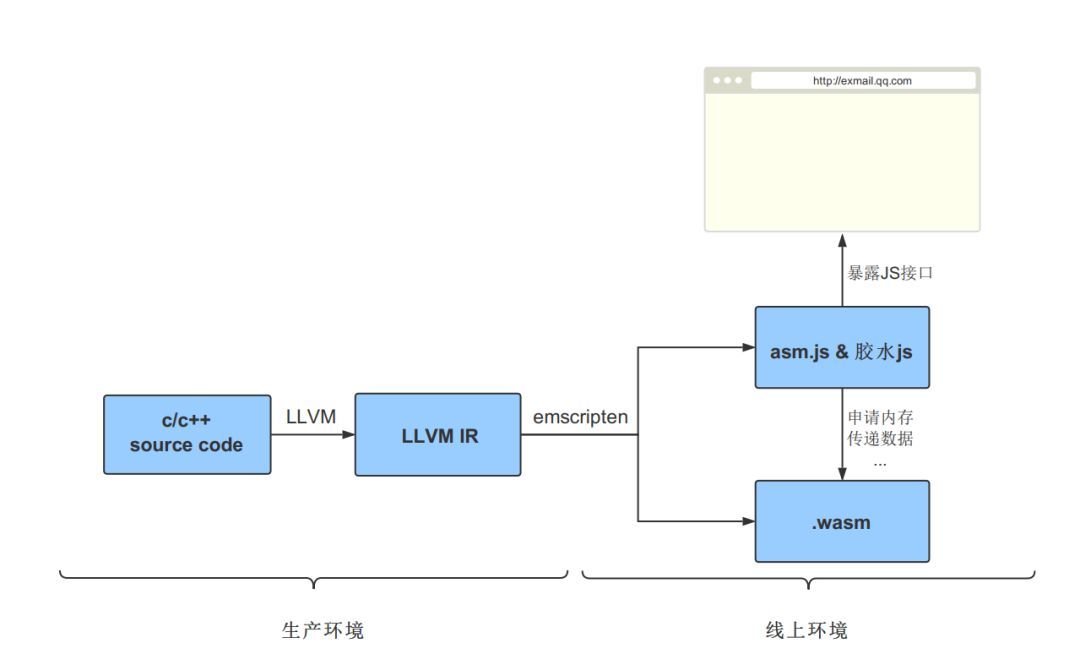
# 对WebAssembly的补充

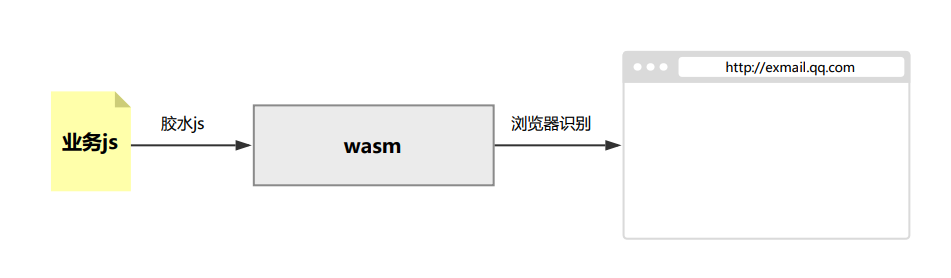
*Ending's law: "Any application that can be compiled to WebAssembly, will be compiled to WebAssembly eventually."*

*Ending定律也称为终结者定律，它是Ending在2016年Emscripten技术交流会上给出的断言：一切可编译为WebAssembly的，终将会被编译为WebAssembly。*（重要性）

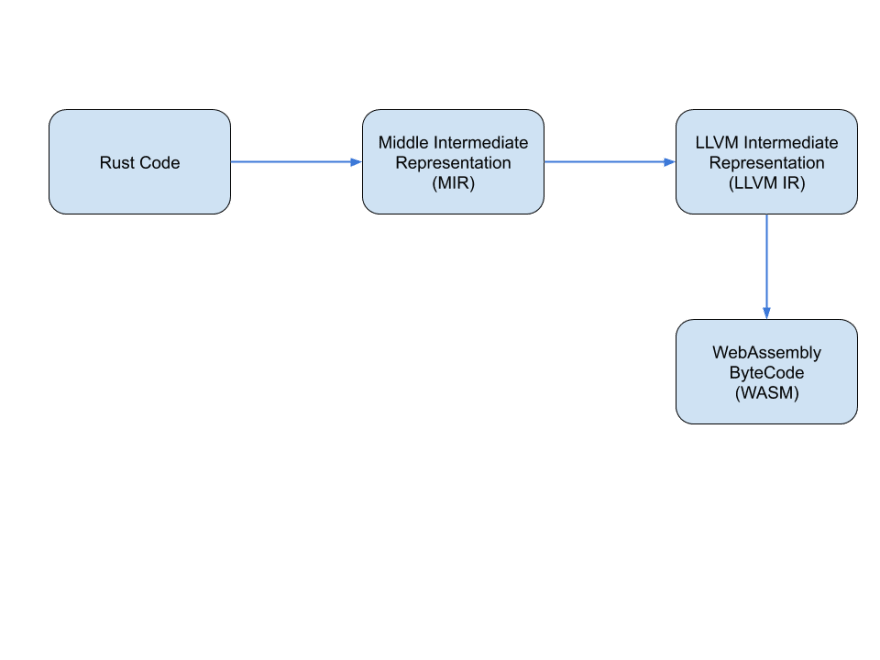
## 应用方面

* bilibili web投稿页面（更加快速地在网页端解析视频）
  + 当你的视频还在上传中，已经可以自由选择AI推荐的封面。这里采用了webassembly+AI的前端整合。
  + 
  + webassembly 负责读取本地视频，生成图片；
  + tensorflow.js 负责加载AI训练过的model，读取图片并打分。
  + 从完全的服务端架构 => 前端架构 && 服务端兜底。
  + webassembly支持解析99%以上的视频编码格式，速度提升体验惠及约50%的web投稿用户
* 邮箱上传和扫描文件方面的应用
  + 核心：LLVM，emscripten，js/wasem/浏览器通信
  + 流程：编译前端 LLVM / Emscripten 流程可以获得 wasm 文件和胶水 js。然后，通过胶水 js 来加载 wasm 并转为 arrayBuffer 格式。紧接着进行编译和实例化后，即可用 JavaScript 与 WebAssembly 通信。
  + 
  + LLVM的简单解释
    - LLVM本质上是一系列帮助开发编译器、解释器的SDK(软件开发工具包)集合，按照传统编译器三段式的结构来说，更接近于优化层（Optimizer）和编译后端（Backend），而不是一个完整的编译器
    - 
    - LLVM在优化层，对中间代码IR来执行代码的优化，它独特的地方在于IR的可读性很高
    - 作为SDK集合的LLVM还提供了一些工具，用来支持代码复用、多语言、JIT，文档也比较友善
  + emscripten
    - emen的编译平台fastcomp负责将LLVM IR转化为特定的机器码或者其他目标语言（包括wasm）。在这里，emen其实扮演了编译器后端的角色（LLVM Backend）
    - js/wasm/浏览器的调用关系
      * 理解：
        + 业务js：实现需要的功能
        + 胶水js：提供和暴露接口，用于代码间的交互

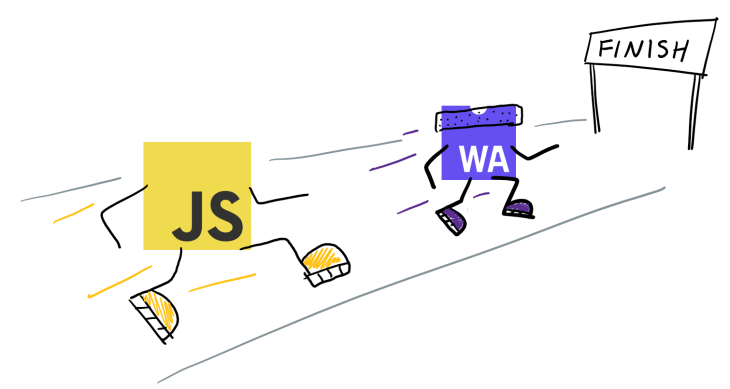
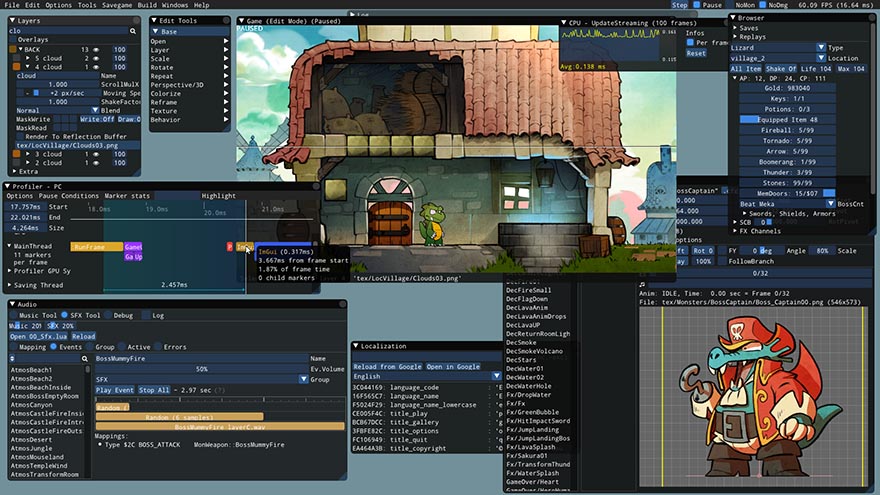
JavaScript胶接代码（glue code），用于连接相互不兼容的软件组件

* + - 
    - 与JS相比的优势：比js更直接的映射为机器码，这是由它所处在IR和机器码之间决定的
    - 达到的效果：扫描1.9G文件耗时约12.1秒，扫描速度可以到160M/s。速度达到了原有速度（75M/s）的2.1倍左右
* 保护核心代码不被破解
  + 在许多需要密码保护的页面，采用了ActiveX，代码逻辑的安全性是有保障了，但缺点也是显而易见的，即通用性太差，只能用于Win系统的特定浏览器，Mac用户，甚至是手机用户都跟这个功能绝缘了
  + 一些大的厂家如Google的reCAPTCHA，就是自己用js实现了一套VM，来解析自定义的字节码，这样即兼顾了代码逻辑保护，也不丢失跨平台通用性。缺点：一般公司没这样的技术跟时间来投入到这样的工作上
  + 这就体现出了WebAssembly的优势：可以对大量C/C++代码进行复用，这样即使是小的创业公司，也能以一个相对低的成本来实现类Google的VM方案来对自己的核心代码进行保护

## 特点

* WebAssembly适合完成CPU密集的操作
* 开源
* 安全性更高了。相比传统的浏览器插件（Plug-in），WebAssembly 的权限设计更好。传统的插件权力太大，它在后台执行了什么，用户根本不知道。而 WebAssembly 再使用某一项权限时都会得到用户的同意才能进行
* **Rust提供了first class对WebAssembly 的支持**（sym)
* **Rust在标准rust编译器（rustc）中构建了自己的WebAssembly工具链。这使Rust开发人员能够将Rust代码编译为WebAssembly模块。**(sym)
* 
* 

## 发展趋势、需求

* WebAssembly 只是允许了更多的语言能编译运行而 JavaScript 作为快速发展的语言, 很可能会被一直支持下去的，两种语言很可能会被长期共用下去, 比如性能要求高的部分用 WebAssembly，而对性能要求不高的部分用JavaScript
* 可以比当前的JavaScript引擎更一致地工作
* Wasm也可以在JavaScript虚拟机中运行，但是它表现得更好。两者可以自由交互、互不排斥，这样你就同时拥有了两者最大的优势——JavaScript巨大的生态系统和有好的语法，WebAssembly接近原生的表现性能。如今市场上可用的每种浏览器都使用不同的JavaScript引擎。引擎基本上将要运行的代码解析为一个抽象语法树，该语法树会生成字节码。不同的JavaScript引擎有时无法以相同的方式解析目标代码，因为它们的实现方式不同，并且可以进行不同的优化。而WebAssembly的实现需要较少的优化和类型假设，因此它将比JavaScript引擎更加一致。
* 
* **RUST对WASM的支持可能是所有高级语言中最完整的**(sym)
* **Rust使用户有机会管理内存，同时也提供了内存安全性**（sym）
* 提供了一个很好的Web编译目标，因此人们可以选择将其网站编码为哪种语言
* WebAssembly支持可管理的线性内存，连续范围的无类型字节
* WebGUI：GUI由许多控件组成，需要准确、快速的放置它们，而基于WebAssembly的C、C++、Rust程序快速地在屏幕上放置大量字节（ImGUI【C++无膨胀图形用户界面】的[WebAssembly实现](https://pbrfrat.com/post/imgui_in_browser.html)）
* 
* 某些平台提供对最大16GiB的内存页面的支持，在某些情况下可以提高内存管理的效率。而WebAssembly可以为程序提供指定比[默认](https://webassembly.org/docs/semantics/#resizing)大小更大的页面大小的选项
* 现在的 WebAssembly 还并不完美，但是线程的支持，异常处理，垃圾收集，尾调用优化等，都已经加入 WebAssembly 的计划列表中了，列表如下

|  |  |  |  |
| --- | --- | --- | --- |
| * 特征 | * 追踪问题 | * 状态 | * 阶段 |
| * 规格 | * 15 | * 进行中 | * 建议的规范文本可用 |
| * 线程数 | * 14 | * 进行中 | * 功能提案 |
| * 固定宽度的SIMD | * 1 | * 进行中 | * 功能提案 |
| * 异常处理 | * 4 | * 进行中 | * 功能提案 |
| * 参考类型 | * 10 | * 进行中 | * 实施阶段 |
| * 垃圾收集 | * 16 | * 进行中 | * 功能提案 |
| * 大容量内存操作 | * 18 | * 进行中 | * 功能提案 |
| * Web内容安全政策 | * 3 | * 进行中 | * 提案前 |
| * ECMAScript模块集成 | * 12 | * 进行中 | * 功能提案 |
| * 尾叫 | * 17 | * 进行中 | * 功能提案 |
| * 非陷阱浮点到整数转换 | * 11 | * 进行中 | * 标准化功能 |
| * 多值 | * 19 | * 进行中 | * 实施阶段 |
| * 主机绑定 | * 8 | * 进行中 | * 功能提案 |
| * 标志扩展符 | * 9 | * 进行中 | * 标准化功能 |
| * 导入/导出可变全局变量 | * 5 | * 进行中 | * 标准化功能 |
| * WebAssembly JavaScript API的类型反射 | * 2 | * 进行中 | * 功能提案 |
| * 非托管关闭 | * 6 | * 进行中 | * 提案前 |
| * JavaScript BigInt与WebAssembly i64集成 | * 7 | * 进行中 | * 提议的规范文本可用 |
| * 文本格式的自定义注释语法 | * 13 | * 进行中 | * 功能提案 |

## 面临的难题、局限

* WebAssembly不适合重逻辑的情况，因为这会增加额外的调用消耗
* JS也在不断的改进，弥补它一些天生的不足，发扬其轻快便捷的语言优势，其现行有很多框架，如React/Angular/Vue，这些框架都是有惯性的，不是说不用就不用
* 至少在当下，在很多场景下(数据量和运算复杂度不是很高)，wasm对比js(jit优化后的)，并没有多大优势【jit-即时编译器（Just In Time Compiler）】
* **WebAssembly 的性能优于js，但是低于nodejs的原生模块或者是c的原生模块。所以如果不是运行在浏览器环境中，不用特意的转化成WebAssembly**
* WebAssembly能做到的事情js都能做到，至少目前还没有发现什么是wasm能做，而js不能的
* WebAssembly具有一些重要的设计约束，这些约束会影响其当前的使用方式，其中包括：
  + 仅支持4种类型-全数字
  + 模块无法直接访问DOM（文档对象类型）或WebAPI
  + 模块可以导入/导出函数（但只能接收/返回wasm数字类型）
  + 没有垃圾收集器，故垃圾收集语言不适用于WebAssembly

### 参考

* https://www.zhihu.com/question/265700379
* https://www.sohu.com/a/283054773\_495695 WebAssembly在邮箱的应用
* https://www.zhuanzhi.ai/document/647669c1fc82b6599ddafb510be3d381
* https://blog.csdn.net/cpongo1/article/details/89548574?depth*1-utm*source=distribute.pc*relevant.none-task&utm*source=distribute.pc\_relevant.none-task WASM发展状况更新，以及LLVM-wasm编译环境搭建过程
* **https://rustwasm.github.io/book/ Rust and WebAssembly Book**（sym）
* https://blog.scottlogic.com/2018/07/20/wasm-future.html 一个关于WebAssembly的一些具体的细节方面的介绍（英文）
* https://medium.com/@sahiljadon/webassembly-the-future-of-web-development-708a25bef57d
* https://webassembly.org/docs/future-features/ future about WebAssembly