

asm.js 原理和应用

EtherDream 2015

分享话题

- 为什么要有 asm.js
- 实现原理
- 相关工具
- 实际应用

JS 性能问题

- ✦ 动态类型
- ✦ 解释执行
- ✦ 沙箱限制

业务需求：灵活 > 性能



编译型语言

- ✦ ActiveX 需安装
- ✦ Applet 需安装
- ✦ NPAPI 需安装
- ✦ Silverlight 需安装
- ✦ NaCl 只支持 Chrome (商店应用)
- ✦ Flash 即将淘汰

提升 JavaScript 性能 ...

脑筋急转弯

1 + 1 = ?

- 2
- 11

动态判断类型，开销很大。

优化

```
var a = 0;
```

```
for (var i = 0; i < 10000; i++) {
```

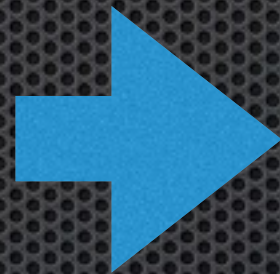
```
    a = a + 1;
```

```
}
```

```
.....
```

```
a = "abc";
```

```
a = a + 1
```



该片段内，
a 始终是数字型，
无需判断类型

数字范围

- ✦ 传统语言

`int8, int16, int32, ...`

- ✦ JavaScript

`number` (内存布局 `double` 类型)

再简单的计算，也使用双精度浮点。

范围约束

信息越充足，优化越容易

例如，明天有人来做客：

- 确定十个人，则准备 10 个座位
- 来十几个人，只能准备 19 个座位

数字范围

- $i = i + 1;$ 结果不确定，只能用浮点容纳
- $i = (i + 1) | 0;$ 结果整数，并且范围 $[-2^{31}, 2^{31})$

JS 中 位运算 会先取整，并且结果保留 32 位

因此 $n | 0$ 的结果可用 `int32_t` 容纳

数字范围

.....

```
for (i = 0; i < n | 0; i = (i + 1) | 0) {  
    sum = (sum + i) | 0  
}
```

JS 引擎优化成 `int32_t`, 取代浮点型

数字范围

- $n \mid 0$ int32_t
 - $n >>> 0$ uint32_t
 - $n \& 0\text{xffff}$ int16_t
 -
- 

asm.js

一种严格的 JavaScript 书写规范

规范

- 只能数值计算（不支持字符串）
- 所有数值必须限定范围（模拟“强类型”）
- 不能使用全局变量
- 不支持闭包、动态参数等任何 JS 特征
-

违反任何一条，都会导致编译失败

模块定义

常用函数 外部函数 内存

```
function AsmModule(stdlib, foreign, heap) {
```

```
    "use asm"; 开启标记
```

```
    程序内容 ...
```

```
    return { 导出接口
```

```
        method_1: fn_1,
```

```
        method_2: fn_2,
```

```
    };
```

```
}
```


函数定义

```
function add( a, b ) {  
    a = a | 0;           // 参数类型声明  
    b = b | 0;  
    return (a + b) | 0;  
}
```



```
int32_t add(int32_t a, int32_t b) {  
    return a + b;  
}
```


演示

```
function AsmTest() {  
    "use asm";  
    function add(a, b) {  
        a = a | 0;  
        b = b | 0;  
        return (a + b) | 0;  
    }  
  
    return {  
        add: add  
    };  
}
```


内存访问

传统 Array

- 可以存储任意 JS 类型
- 长度可变，可存在空元素

HTML5 ArrayBuffer

- 存储二进制数据
- 长度固定，线性存储

内存访问

```
var buf = new ArrayBuffer(1000);  
var ptr8 = new Int8Array(buf);  
var ptr16 = new Int16Array(buf);
```



```
char* buf = new char[1000];  
int8_t* ptr8 = (int8_t*) buf;  
int16_t* ptr16 = (int16_t*) buf;
```


模拟指针

```
ptr8[0] = 0x11;
```

```
ptr8[1] = 0x22;
```

```
console.log( ptr16[0] );    // 0x2211
```


asm.js 总结

- 使用 JS 语法，描述一种底层语言
- 强类型、静态特征、线性内存，性能接近本地程序
- 不支持 asm.js 的引擎仍能正常运行，只是性能较低

缺点：书写复杂，需借助工具生成

相关工具

能将 C/C++ 转换成 asm.js 的工具

- Cheerp 不流行
- **emscripten** Mozilla 发布, 免费, 开源, 持续更新

转换方式

不同于 CoffeeScript、TypeScript 那种 1:1 转换

emscripten:



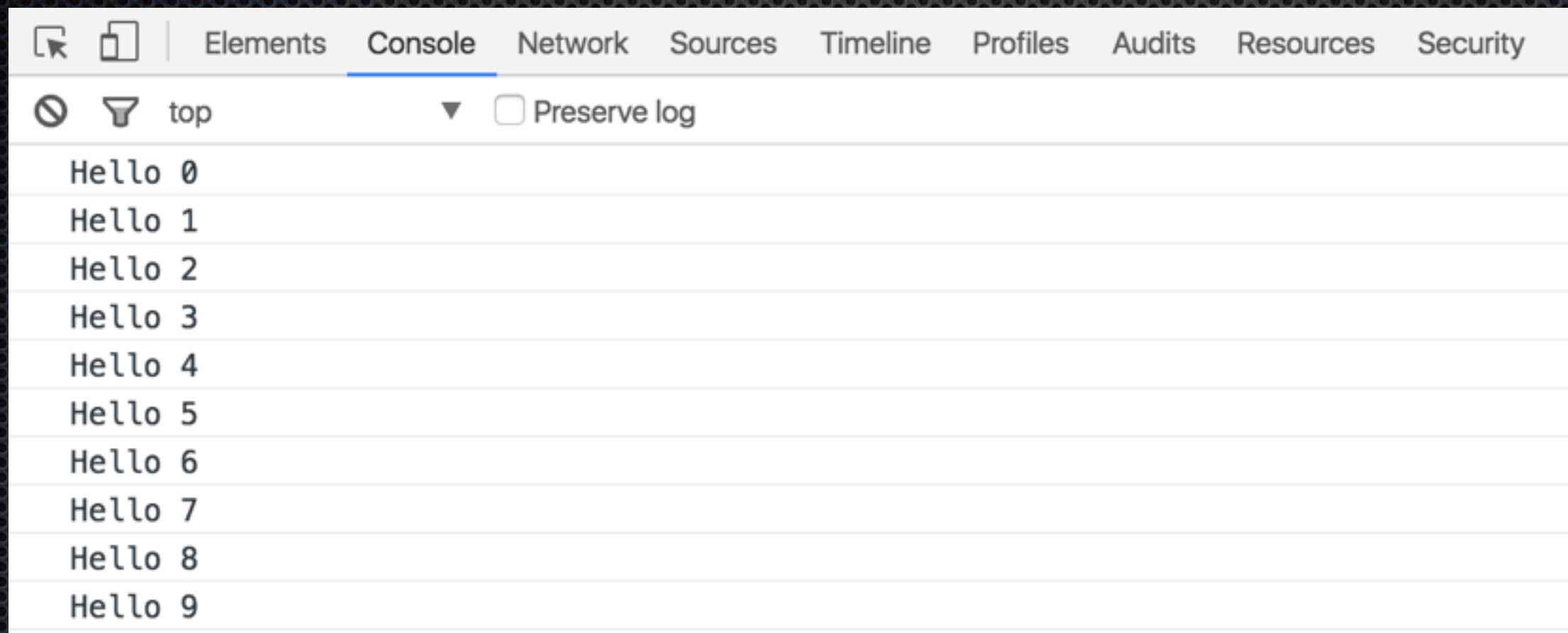
编译期间深度优化，性能再次提升

演示

```
main.c x
1  #include <stdio.h>
2
3  int main() {
4      for (int i = 0; i < 10; i++) {
5          printf("Hello %d\n", i);
6      }
7      return 1;
8  }
```

emcc main.c -o main.html

演示



细节

1. 字符串如何实现

2. 程序流程转换

3. 管理内存

4. API 模拟

字符串

C 字符串本质为数字（char*），所以内部都是 数值计算

只有输出展示时，才转换成 JS 字符串（utf-8）

程序流程

JS 和 C 都支持的流程符：

if / for / do / while / switch

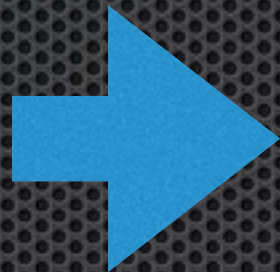
JS 不支持的流程符：

goto

reloop

模拟语块之间跳转

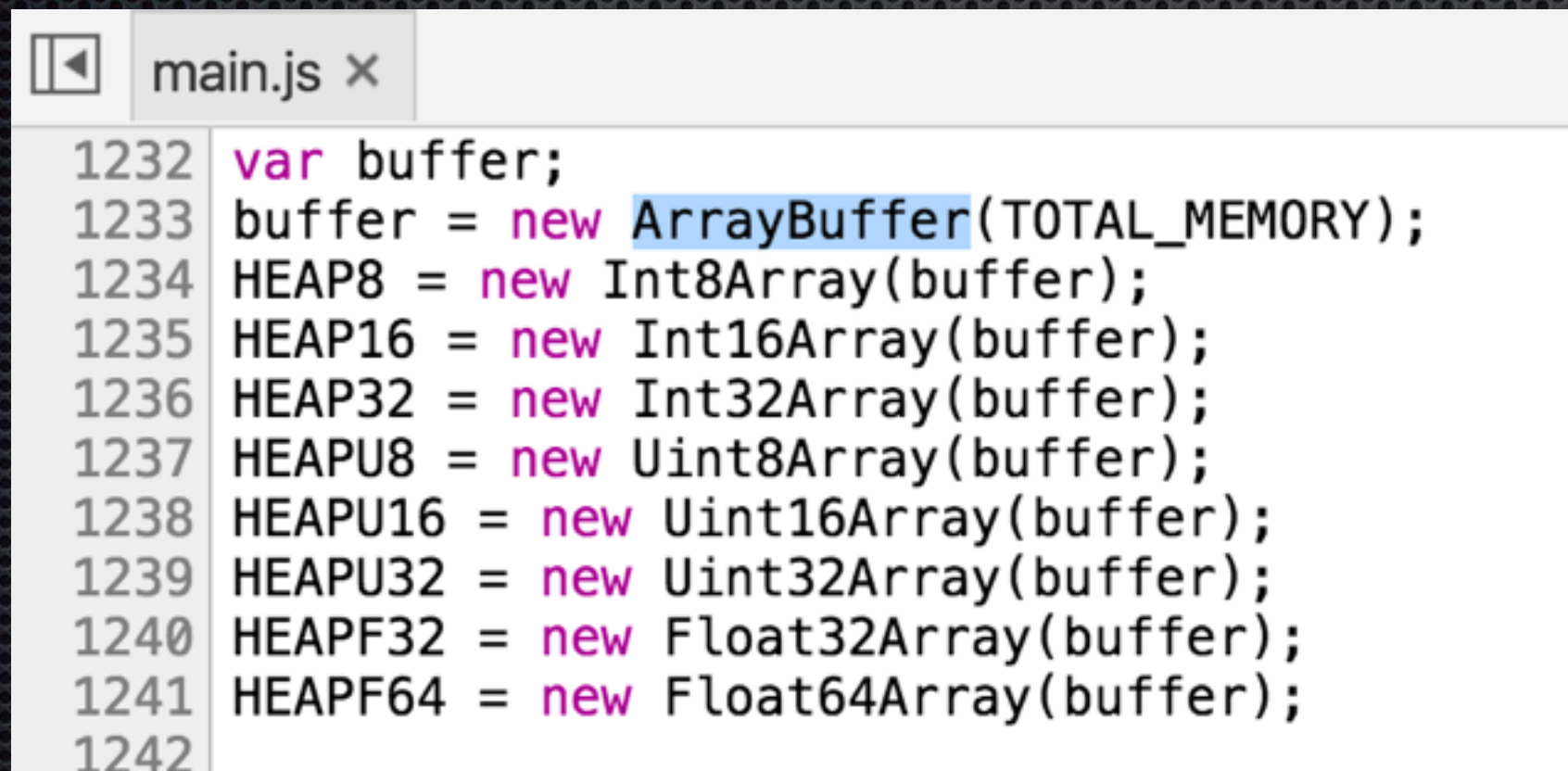
```
a:
.....;
goto c;
b:
.....;
goto a;
c:
.....;
goto b;
```



```
while (true) {
  switch (label) {
    case 1:
      .....;
      label = 3; break;
    case 2:
      .....;
      label = 1; break;
    case 3:
      .....;
      label = 2; break;
  }
}
```


内存管理

申请大块 ArrayBuffer, 模拟地址空间



The screenshot shows a code editor window with a tab labeled 'main.js'. The code is as follows:

```
1232 var buffer;  
1233 buffer = new ArrayBuffer(TOTAL_MEMORY);  
1234 HEAP8 = new Int8Array(buffer);  
1235 HEAP16 = new Int16Array(buffer);  
1236 HEAP32 = new Int32Array(buffer);  
1237 HEAPU8 = new Uint8Array(buffer);  
1238 HEAPU16 = new Uint16Array(buffer);  
1239 HEAPU32 = new Uint32Array(buffer);  
1240 HEAPF32 = new Float32Array(buffer);  
1241 HEAPF64 = new Float64Array(buffer);  
1242
```


内存管理

ArrayBuffer 的分配

- 全局数据（全局变量、字符串常量等）
- 栈（函数中的局部变量）
- 堆（动态申请的内存）

内存管理

```
uint8_t arr[100];
```

全局变量

```
int main() {
```

```
    int i = 123;
```

栈数据

```
    malloc(100);
```

堆数据

```
    printf("hello world");
```

常量数据

```
    return 0;
```

```
}
```


栈的细节

```
function A() {  
    var a = 0;           // 直接用 JS 层面的栈  
    var b = 0;  
  
    buffer[SP + 0] = 1;  // 使用虚拟的栈  
    buffer[SP + 1] = 2;  
}
```

C 中简单的局部变量，大多可编译成 JS 变量

栈的细节

```
int func() {
```

```
    int i = 123;
```

独立的 C 栈变量 -> 独立的 JS 栈变量

```
    i = 456;
```

```
    int x = 123;
```

该变量存在引用，只能使用模拟栈

```
    int* y = &x;
```

```
    .....
```

```
}
```


堆内存

```
6495 function _malloc($bytes) {  
8919 }  
8920 function _free($mem) {  
8921     $mem = $mem|0;  
8922     var $$lcssa = 0, $$pre = 0, $$pre$p  
8923     var $$sum25 = 0, $$sum26 = 0, $$sum  
8924     var $105 = 0, $106 = 0, $107 = 0, $  
8925     var $123 = 0, $124 = 0, $125 = 0, $  
8926     var $141 = 0, $142 = 0, $143 = 0, $  
8927     var $16 = 0, $160 = 0, $161 = 0, $1  
8928     var $178 = 0, $179 = 0, $18 = 0, $1
```

使用 malloc 和 free 函数，动态管理 ArrayBuffer 中的内存

API 模拟

- 文件系统
- 终端 (TTY)
- 图像渲染
- 音频播放
- 网络接口

演示

<https://github.com/kripken/emscripten/wiki/Porting-Examples-and-Demos>

<https://tsone.kapsi.fi/em-fceux/>

asm.js 缺陷

本质仍是 JS 代码，初始化开销很大（语法分析、优化等）

解决方案：Emterpreter

C -> 字节码，由虚拟机解释执行（运行变慢，启动加快）

原理：<https://github.com/kripken/emscripten/wiki/Emterpreter>

对比：<https://research.mozilla.org/2015/02/23/the-emterpreter-run-code-before-it-can-be-parsed/>

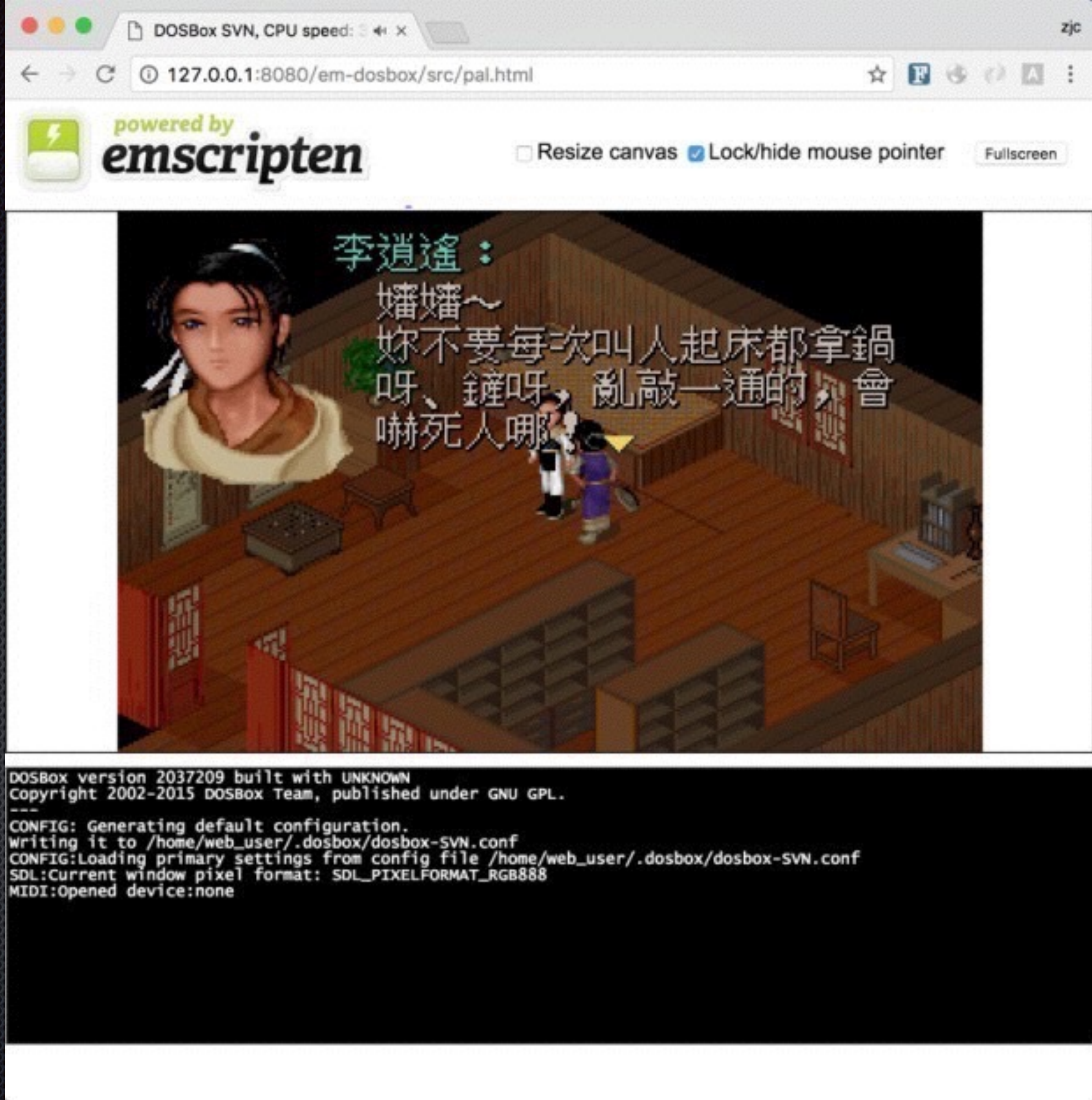
Interpreter 案例

网页版 DOSBOX

<https://github.com/dreamlayers/em-dosbox>

在线演示

<http://playdosgamesonline.com/>



EM-DOSBOX

虚拟机 (JS 引擎) 里的

虚拟机 (emterpreter) 里的

虚拟机 (X86 模拟)

盗梦空间的感觉~

不过仍然很流畅。。。~



☐ Resize canvas



DOSBox version 2037209 built with UNKNOWN
Copyright 2002-2015 DOSBox Team, published under GNU GPL.

CONFIG:Loading primary settings from config file dosbox.conf
SDL:Current window pixel format: SDL_PIXELFORMAT_RGB888
MIDI:Opened device:none
Using partition 0 on drive; skipping 63 sectors
Mounted FAT volume is FAT16 with 64110 clusters
Loaded disk 0 from swaplist position 0 - "drive-c.img"
Loaded disk 1 from swaplist position 0 - "drive-c.img"

原生虚拟机

WebAssembly

- 二进制字节码
- 支持 64 bit 运算
- 标准规范，未来兼容性更好

拭目以待...

EOF