陈卷毛 Lv1
2020年02月10日 阅读 928

关注

WebAssembly介绍之1：二进制格式

关于WebAssembly（下文简称Wasm）的介绍已经有很多了，本文不打算再多啰嗦。本文介绍的重点是Wasm二进制格式，我们会把一个最简单的Rust程序（没错，就是 `Hello，World！` 程序）编译成Wasm二进制格式，然后以Go伪代码结合 `xxd` 命令的形式来剖析Wasm二进制格式 。下面是这个Rust程序的完整代码（如果不了解如何将Rust编译成Wasm，请看这篇文章）：

复制代码

```
#![no_std]
#![no_main]

extern "C" {
    fn print_str(ptr: *const u8, len: usize);
}

#[panic_handler]
fn panic(_info: &core::panic::PanicInfo) -> ! {
    loop {}
}

#[no_mangle]
pub extern "C" fn main() {
    unsafe {
        let s = "Hello, World!\n";
        print_str(s.as_ptr(), s.len());
    }
}
```

Module

Wasm的顶层结构是**模块**（Module），每一个Wasm二进制文件对应一个模块。模块以4字节魔数开始，接着是4字节版本号，其余是模块的数据。具体的模块数据被分门别类的放在不同的**段**（Section）中，每个段都由唯一的段ID来标识。除了**自定义段**（后文会介绍）以外，其他所有的段都

复制代码

```
type Module struct {
  Magic    uint32 // `\0asm`
  Version  uint32 // 0x00000001
  Sections []Section
}
```

用 `xxd` 命令观察hw.wasm文件可以发现，Wasm二进制格式的魔数是 `\0asm` ，当前的版本号是1（整数按照小端在前的方式存储）：

复制代码

```
$ xxd -u -g 1 target/wasm32-unknown-unknown/release/hw.wasm
00000000: 00 61 73 6D 01 00 00 00 01 0F 03 60 02 7F 7F 00  .asm.......`....
          ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^
00000010: 60 02 7F 7F 01 7F 60 00 00 02 11 01 03 65 6E 76  `.....`......env
00000020: 09 70 72 69 6E 74 5F 73 74 72 00 00 03 05 04 01  .print_str......
...
```

Section

Wasm规范一共定义了12种段，ID从0到11，见下面的Go常量定义：

复制代码

```
const (
    SecCustomID = 0
    SecTypeID   = 1
    SecImportID = 2
    SecFuncID   = 3
    SecTableID  = 4
    SecMemID    = 5
    SecGlobalID = 6
    SecExportID = 7
    SecStartID  = 8
    SecElemID   = 9
    SecCodeID   = 10
    SecDataID   = 11
)
```

跳过不认识的自定义段。为了让wasm二进制文件更加紧凑，整数是按LEB128格式编码后存储的。下面的伪代码给出了段的抽象结构（`varU32` 类型表示LEB128编码的32位无符号整数）：

复制代码

```
type Section struct {
  ID   byte
  Size varU32 // uint32
  Cont []byte
}
```

Type Section

在Wasm二进制文件里，函数的信息分布在三个段里（先忽略导入的外部函数以及调试信息）。类型（Type）段用来存放函数的类型信息（或者签名），代码（Code）段存放函数的局部变量信息和字节码，函数（Function）段将代码段和类型段关联起来。由于需要存储多个函数类型，所以需要把函数类型的数量先存下来，这是Wasm二进制格式存储Vector数据的标准做法。为了简单起见，我们在后面的伪代码中省略数量，直接用Go的slice类型表示vec。类型段的ID是1，结构由下面的伪代码给出：

复制代码

```
type TypeSec struct {
  ID    byte       // 0x01
  Size  varU32     // uint32
  Types []FuncType // vec
}
```

函数类型由 `0x60` 开头，然后是参数类型和返回值类型。

复制代码

```
type FuncType struct {
  Tag     byte // 0x60
    Params  []ValType
  Results []ValType
}

// 0x7F: i32, 0x7E: i64, 0x7D: f32, 0x7C: f64
type ValType = byte
```

复制代码

```
$ xxd -u -g 1 target/wasm32-unknown-unknown/release/hw.wasm
00000000: 00 61 73 6D 01 00 00 00 01 0F 03 60 02 7F 7F 00  .asm.......`....
                                  ^^ ^^ ^^
00000010: 60 02 7F 7F 01 7F 60 00 00 02 11 01 03 65 6E 76  `.....`......env
00000020: 09 70 72 69 6E 74 5F 73 74 72 00 00 03 05 04 01  .print_str......
...
```

再来观察第一个函数类型，可以看到的确由 `0x60` 开头，参数有两个，都是 `i32` 类型（ `0x7F` ），没有返回值：

复制代码

```
$ xxd -u -g 1 target/wasm32-unknown-unknown/release/hw.wasm
00000000: 00 61 73 6D 01 00 00 00 01 0F 03 60 02 7F 7F 00  .asm.......`....
                                        ^^ ^^ ^^ ^^ ^^
00000010: 60 02 7F 7F 01 7F 60 00 00 02 11 01 03 65 6E 76  `.....`......env
00000020: 09 70 72 69 6E 74 5F 73 74 72 00 00 03 05 04 01  .print_str......
...
```

Import Section

导入段存储导入信息，段ID是2。导入信息又包括导入模块名、导入名和导入描述。Wasm支持四种类型的导入：函数、表、内存和全局变量。为了区分具体是哪种导入，导入描述需要以一字节的tag开头，具体的描述信息因tag而异。下面的伪代码描述了导入段的整体结构：

复制代码

```
type ImportSec struct {
  ID      byte // 0x02
  Size    varU32
  Imports []Import
}

type Import struct {
    Module  string
  Name    string
  DescTag byte // func: 0, table: 1, mem: 2, global: 3
  Desc    ...
}
```

（ 0x11 ）个字节，仅有1条导入信息：

复制代码

```
$ xxd -u -g 1 target/wasm32-unknown-unknown/release/hw.wasm
00000000: 00 61 73 6D 01 00 00 00 01 0F 03 60 02 7F 7F 00  .asm.......`....
00000010: 60 02 7F 7F 01 7F 60 00 00 02 11 01 03 65 6E 76  `.....`......env
                                  ^^ ^^ ^^
00000020: 09 70 72 69 6E 74 5F 73 74 72 00 00 03 05 04 01  .print_str......
...
```

在唯一的一条导入信息中，模块名是 `env` （这是由Rust编译器默认生成的），导入名是 `print_str` ：

复制代码

```
$ xxd -u -g 1 target/wasm32-unknown-unknown/release/hw.wasm
00000000: 00 61 73 6D 01 00 00 00 01 0F 03 60 02 7F 7F 00  .asm.......`....
00000010: 60 02 7F 7F 01 7F 60 00 00 02 11 01 03 65 6E 76  `.....`......env
                                        ^^ ^^ ^^ ^^              ^^^^
00000020: 09 70 72 69 6E 74 5F 73 74 72 00 00 03 05 04 01  .print_str......
          ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^                   ^^^^^^^^^^
...
```

不难发现，在Wasm二进制文件里，字符串也是按照vector方式（长度+内容）存储的，内容是UTF-8编码后的字节数组。继续观察可以看到，tag是 `0x00` ，表示这是一条函数导入，函数的类型信息存储在类型段的第0号位置：

复制代码

```
$ xxd -u -g 1 target/wasm32-unknown-unknown/release/hw.wasm
00000000: 00 61 73 6D 01 00 00 00 01 0F 03 60 02 7F 7F 00  .asm.......`....
00000010: 60 02 7F 7F 01 7F 60 00 00 02 11 01 03 65 6E 76  `.....`......env
00000020: 09 70 72 69 6E 74 5F 73 74 72 00 00 03 05 04 01  .print_str......
                                        ^^ ^^
...
```

Function Section

函数段相对来说比较简单，按顺序存储了每一个内部函数的签名在类型段中的索引。比如说某个模块有5个内部函数，那么查两次表就可以找到该函数的签名： `TypeSec.Types[FuncSec.Types[n]]` 。函数段的ID是3，整体结构由下面的伪代码给出：

```
  Size  varU32
  Types []TypeIdx // []varU32
}
```

用 xxd 命令观察hw.wasm文件可知，函数段跟在导入段后面，内容是5个字节，包含4个函数类型索引（1、1、2、1）：

复制代码

```
$ xxd -u -g 1 target/wasm32-unknown-unknown/release/hw.wasm
00000000: 00 61 73 6D 01 00 00 00 01 0F 03 60 02 7F 7F 00  .asm.......`....
00000010: 60 02 7F 7F 01 7F 60 00 00 02 11 01 03 65 6E 76  `.....`......env
00000020: 09 70 72 69 6E 74 5F 73 74 72 00 00 03 05 04 01  .print_str......
                                        ^^ ^^ ^^ ^^
00000030: 01 02 01 04 05 01 70 01 01 01 05 03 01 00 11 06  ......p.........
          ^^ ^^ ^^
...
```

Table&Element Section

表段（ID是4）和元素段（ID是9）与间接函数调用指令有关，本文不做介绍，请读者参考Wasm规范。

Memory Section

内存段的ID是5，存放内存信息。内存信息给出该模块运行所需内存页数（一页是64K）的下限（必须指定）和上限（可选）。虽然内存段是支持多个内存的，但是Wasm1.0规范规定内存不能超过一个，这个限制可能会在后续版本中放开。下面的伪代码描述了内存段的整体结构：

复制代码

```
type MemSec struct {
  ID    byte // 0x05
  Size  varU32
  Mems  []Limits
}

type Limits struct {
  HasMax byte // 0: min, 1: min+max
  Min    varU32
```

用 `xxd` 命令观察hw.wasm文件可知，内存段的内容是3个字节，包含1份内存信息。这份内存信息以0开头，所以仅指定了页数下限（ `0x11` ，也就是17），没有指定页数上限：

复制代码

```
$ xxd -u -g 1 target/wasm32-unknown-unknown/release/hw.wasm
00000000: 00 61 73 6D 01 00 00 00 01 0F 03 60 02 7F 7F 00  .asm.......`....
00000010: 60 02 7F 7F 01 7F 60 00 00 02 11 01 03 65 6E 76  `.....`......env
00000020: 09 70 72 69 6E 74 5F 73 74 72 00 00 03 05 04 01  .print_str......
00000030: 01 02 01 04 05 01 70 01 01 01 05 03 01 00 11 06  ......p.........
                                    ^^ ^^ ^^ ^^ ^^

...
```

Globlal Section

全局变量段存储内部（非导入）的全局变量信息，段ID是6。全局变量信息包括变量的类型、是否只读以及用于初始化该全局变量的字节码。下面的伪代码描述了全局变量段的整体结构：

复制代码

```
type GlobalSec struct {
  ID      byte // 0x06
  Size    varU32
  Globals []Global
}

type Global struct {
  Type  ValType
  Mut   byte // const: 0, var: 1
  Expr  []Instruction
}
```

用 `xxd` 命令观察hw.wasm文件可知，全局变量段跟在内存段后面，内容是25（ `0x19` ）个字节，包含3个全局变量：

复制代码

```
...
00000030: 01 02 01 04 05 01 70 01 01 01 05 03 01 00 11 06  ......p.........
                                                  ^^
00000040: 19 03 7F 01 41 80 80 C0 00 0B 7F 00 41 8E 80 C0  ....A.......A...
          ^^ ^^
```

其中第1个全局变量的类型是 `i32` （ `0x7F` ），可变（ `0x01` ），初始化字节码一共6个字节（本文不介绍指令编码，请读者参考Wasm规范）：

复制代码

```
...
00000030: 01 02 01 04 05 01 70 01 01 01 05 03 01 00 11 06  ......p.........
00000040: 19 03 7F 01 41 80 80 C0 00 0B 7F 00 41 8E 80 C0  ....A.......A...
                ^^ ^^ ~~ ~~ ~~ ~~ ~~ ~~
00000050: 00 0B 7F 00 41 8E 80 C0 00 0B 07 2C 04 06 6D 65  ....A......,..me
...
```

Export Section

和导入段相对应的是导出段，ID是7。和导入信息一样，导出信息也分为4种：函数、表、内存和全局变量。不过导出信息要稍微简单一些，不包含模块 名，且无论导出的是什么，都只需给出相应的索引即可。下面的伪代码描述了导出段的整体结构：

复制代码

```
type ExportSec struct {
  ID      byte // 0x07
  Size    varU32
  Exports []Export
}

type Export struct {
  Name  string
  Tag   byte // func: 0, table: 1, mem: 2, global: 3
  Idx   varU32
}
```

用 `xxd` 命令观察hw.wasm文件可知，导出段跟在全局变量段后面，内容是44（ `0x2C` ）个字节，包含4个全局变量：

复制代码

```
...
00000030: 01 02 01 04 05 01 70 01 01 01 05 03 01 00 11 06  ......p.........
00000040: 19 03 7F 01 41 80 80 C0 00 0B 7F 00 41 8E 80 C0  ....A.......A...
00000050: 00 0B 7F 00 41 8E 80 C0 00 0B 07 2C 04 06 6D 65  ....A......,..me
```

其中第4个导出的名字是 `main` ，类型是函数（ `0x00` ），索引是3（导入的函数和内部函数一起构成了函数索引空间）：

复制代码

```
...
00000050: 00 0B 7F 00 41 8E 80 C0 00 0B 07 2C 04 06 6D 65  ....A......,..me
00000060: 6D 6F 72 79 02 00 0A 5F 5F 64 61 74 61 5F 65 6E  mory..__data_en
00000070: 64 03 01 0B 5F 5F 68 65 61 70 5F 62 61 73 65 03  d...__heap_base.
00000080: 02 04 6D 61 69 6E 00 03 0A DD 01 04 5D 01 08 7F  ..main......]...
             ^^ ^^ ^^ ^^ ^^ ^^ ^^                          ^^^^^
...
```

Start Section

和其他段不同，起始段只包含一个起始函数索引。指定的函数相当于C/C++/Java等语言里的 `main` 函数，当Wasm实现将模块实例化后，需要执行这个 函数。起始段的ID是8，结构可以用下面的伪代码描述：

复制代码

```
type StartSec struct {
  ID      byte // 0x08
  Size    varU32
  FuncIdx varU32
}
```

由于Rust编译器没有给模块生成起始段，所以也就没办法用hw.wasm文件进行观察了。

Code Section

如前所述，内部函数的局部变量信息和字节码存放在代码段中，段ID是10。为了便于并行处理（比如验证、分析、AOT编译等），代码段中的每一项都自带了字节数。这样就可以先把每一个内部函数的局部变量信息和字节码数据提取出来，然后同时处理多个函数。下面的伪代码描述了代码段的整体结构：

```
  Size  varU32
  Codes []Code
}

type Code struct {
  Size   varU32
    Locals []Locals
    Expr   []Instruction
}
type Locals struct {
    N    uint32
    Type ValType
}
```

用 `xxd` 命令观察hw.wasm文件可知，代码段跟在导出段后面，内容是221（ `0xDD` 、 `0x01` ）个字节，包含4个内部函数：

复制代码

```
...
00000080: 02 04 6D 61 69 6E 00 03 0A DD 01 04 5D 01 08 7F  ..main......]...
                                ^^ ^^ ^^ ^^
00000090: 23 80 80 80 80 00 21 02 41 10 21 03 20 02 20 03  #.....!.A.!. . .
000000a0: 6B 21 04 20 04 24 80 80 80 80 00 20 04 20 00 36  k!. .$..... . .6
000000b0: 02 08 20 04 20 01 36 02 0C 20 04 28 02 08 21 05  .. . .6.. .(..!.
000000c0: 20 04 28 02 0C 21 06 20 05 20 06 10 84 80 80 80   .(..!. . ......
...
```

第一个内部函数有93个字节的数据（ `0x5D` ）。有一个局部变量信息，表示该函数需要8个 `i32` （ `0x7F` ）类型的参数。其余90个字节是该函数的字节码（本文不介绍指令编码，请读者参考Wasm规范）：

复制代码

```
...
00000080: 02 04 6D 61 69 6E 00 03 0A DD 01 04 5D 01 08 7F  ..main......]...
                                        ^^ ^^ ^^ ^^
00000090: 23 80 80 80 80 00 21 02 41 10 21 03 20 02 20 03  #.....!.A.!. . .
000000a0: 6B 21 04 20 04 24 80 80 80 80 00 20 04 20 00 36  k!. .$..... . .6
000000b0: 02 08 20 04 20 01 36 02 0C 20 04 28 02 08 21 05  .. . .6.. .(..!.
000000c0: 20 04 28 02 0C 21 06 20 05 20 06 10 84 80 80 80   .(..!. . ......
000000d0: 00 21 07 41 10 21 08 20 04 20 08 6A 21 09 20 09  .!.A.!. . .j!. .
000000e0: 24 80 80 80 80 00 20 07 0F 0B 05 00 20 00 0F 0B  $..... ..... ...
```

Data Section

数据段存放内存初始化信息，ID是11。每条内存初始化信息包括内存索引（目前只可能是0）、起始位置（字节码，且必须是是常量表达式）、初始数据。下面的伪代码描述了数据段的整体结构：

复制代码

```
type DataSec struct {
  ID    byte // 0x0B
  Size  varU32
  Datas []Data
}

type Data struct {
  MemIdx varU32 // 0
  Offset []Instruction
  Init   []byte
}
```

用 `xxd` 命令观察hw.wasm文件可知，数据段跟在代码段后面，内容是23（ `0x17` ）个字节，包含1份内存初始化信息：

复制代码

```
...
00000160: 02 0C 21 05 20 05 0F 0B 0B 17 01 00 41 80 80 C0  ..!. .......A...
                                  ^^ ^^ ^^
00000170: 00 0B 0E 48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21  ...Hello, World!
00000180: 0A 00 DE 01 04 6E 61 6D 65 01 D6 01 05 00 09 70  .....name......p
...
```

这份内存初始化信息的内存索引是0，内存起始位置由一条i32.const指令（操作码是 `0x41` ，参数是 `0x100000` ）给出。由指该令可以算出，内存起始位置在第17页的最开始（ `0x100000 / (64 * 1024)` ）：

复制代码

```
...
00000160: 02 0C 21 05 20 05 0F 0B 0B 17 01 00 41 80 80 C0  ..!. .......A...
                                        ^^ ^^ ^^ ^^ ^^
00000170: 00 0B 0E 48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21  ...Hello, World!
          ^^
```

初始化数据是14（ `0x0E` ）个字节，内容就是Rust代码中的字符串字面量 `Hello, World!\n` ：

复制代码

```
...
00000160: 02 0C 21 05 20 05 0F 0B 0B 17 01 00 41 80 80 C0  ..!. .......A...
00000170: 00 0B 0E 48 65 6C 6C 6F 2C 20 57 6F 72 6C 64 21  ...Hello, World!
                ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^    ^^^^^^^^^^^^^
00000180: 0A 00 DE 01 04 6E 61 6D 65 01 D6 01 05 00 09 70  .....name......p
          ^^                                              ^
...
```

Custom Section

自定义段可以存放调试信息、第三方扩展信息等等，这些信息不是Wasm执行所必须的，可以完全忽略。另外，与其他段不同（必须严格按顺序出现，且最多只能出现一次），自定义段可以自由出现在其他段前后，且可以连续出现多次。自定义段的内容以字符串开始，这样就进一步区分自定义段。下面的伪代码描述了自定义段的抽象结构：

复制代码

```
type CustomSec struct {
  ID    byte // 0
  Size  varU32
  Name  string
  Cont  ...
}
```

Wasm规范只定义了一种自定义段，name是字符串 `name` 。本文不介绍name段的具体格式，请读者参考Wasm规范。用 `xxd` 命令观察hw.wasm文件可知，在数据段后面跟了一个自定义段，内容是222（ `0xDE` 、 `0x01` ）个字节，它的name正好就是 `name` （ `0x6E` 、 `0x61` 、 `0x6D` 、 `0x65` ）：

复制代码

```
...
00000180: 0A 00 DE 01 04 6E 61 6D 65 01 D6 01 05 00 09 70  .....name......p
             ^^ ^^ ^^ ^^ ^^ ^^ ^^ ^^
00000190: 72 69 6E 74 5F 73 74 72 01 3B 5F 5A 4E 34 63 6F  rint_str.;_ZN4co
000001a0: 72 65 33 73 74 72 32 31 5F 24 4C 54 24 69 6D 70  re3str21_$LT$imp
000001b0: 6C 24 75 32 30 24 73 74 72 24 47 54 24 33 6C 65  l$u20$str$GT$3le
000001c0: 6E 31 37 68 63 65 63 35 39 61 61 36 36 64 36 34  n17hcec59aa66d64
```

```
00000200: 72 31 37 68 31 30 64 38 36 64 62 35 37 30 32 38  r17h10d86db57028
00000210: 32 33 30 64 45 03 04 6D 61 69 6E 04 45 5F 5A 4E  230dE..main.E_ZN
00000220: 34 63 6F 72 65 35 73 6C 69 63 65 32 39 5F 24 4C  4core5slice29_$L
00000230: 54 24 69 6D 70 6C 24 75 32 30 24 24 75 35 62 24  T$impl$u20$u5b$
00000240: 54 24 75 35 64 24 24 47 54 24 33 6C 65 6E 31 37  T$u5d$GT$3len17
00000250: 68 64 36 36 37 30 30 66 65 31 35 30 61 66 38 62  hd66700fe150af8b
00000260: 63 45
```

文章分类 代码人生 文章标签 WebAssembly

陈卷毛 Lv1
获得点赞 5 · 获得阅读 8,642

关注

输入评论（Enter换行，Ctrl + Enter发送）

发表评论

相关推荐

天行无忌 4天前 WebAssembly Unity3D 前端

WebAssembly影响未来WEB技术

WebAssembly（缩写为 Wasm）是一种用于基于堆栈的虚拟机的二进制指令格式， Was...

556 9 评论

WebAssembly是什么

2019 年 12 月 5 日，WebAssembly正式加入 HTML、CSS 和 JavaScript 的 Web 标准大家庭。

255 2 评论

seminelee小青龙 11天前 WebAssembly

前端webassembly+ffmpeg+web worker视频抽帧

自定义编译ffmpeg，优化wasm文件大小，编译出不带`sharedarraybuffer`的...

202 7 6

Michael_Yuan 15天前 Serverless WebAssembly 前端

Netlify 中的 Rust 与 WebAssembly Serverless 函数

如何使用 Rust 编写的 WasmEdge 函数来支持 Netlify 应用程序后端，构建高性能的函数。

453 4 评论

EvalStudio 1月前 WebAssembly 前端

WebAssembly漫谈

WebAssembly是什么 —— https://webassembly.org/ 从官网释义来看，Wasm是基于栈...

641 31 评论

Michael_Yuan 1月前 Rust WebAssembly 前端

目前大火的 Jamstack 到底是什么?

这篇文章将带你了解 Jamstack 的概念以及开发范式。我们也将讨论 Rust 与 WebAssembl...

1135 11 评论

星星不懂前端啊o_o 23天前 WebAssembly 前端

初识 WASM

Wasm 是什么？ Wasm（WebAssembly） 是一种底层的汇编语言，能够在所有当代桌面浏览器及很多移动浏览器...

331 7 评论

声网Agora 1月前 算法 WebAssembly 前端

实践解析 | 如何通过 WebAssembly 在 Web 进行实时视频人像分割

5 月 15 日，声网Agora 高级架构师高纯参加了 WebAssambly 社区举办的第一场线下活...

980 11 评论

doodlewind 9月前 JavaScript WebAssembly

4498 79 13

jordiwang 1年前 WebAssembly

前端视频帧提取 ffmpeg + Webassembly

现有的前端视频帧提取主要是基于 canvas + video 标签的方式，在用户本地选取视频文件...

7076 148 38

Michael_Yuan 1月前 Go WebAssembly 后端

通过 WasmEdge 嵌入WebAssembly 函数扩展 Golang 应用

通过 WasmEdge ，用 Rust 扩展 Golang 应用。WebAssembly 提供了一种强大、灵活、...

607 3 评论

Michael_Yuan 23天前 Serverless WebAssembly 后端

在腾讯云上部署基于 WebAssembly 的高性能 serverless 函数

WebAssembly 拥有 Docker 的安全隔离、跨平台可移植、可编排等优点，从应用的颗粒度...

96 1 评论

Rust_Magazine 1月前 WebAssembly Rust 前端

华为 | WebAssembly 安全性调研

作者：华为可信软件工程和开源2012实验室 2015年4月，W3C成立WebAssembly工作组，用于监督与规范...

468 1 评论

SH的全栈笔记 2年前 JavaScript 前端 编译器

WebAssembly完全入门——了解wasm的前世今身

接触WebAssembly之后，在google上看了很多资料。感觉对WebAssembly的使用、介绍...

2.3w 188 20

老錢 2年前 Go JavaScript 前端

不安分的 Go 语言开始入侵 Web 前端领域了

从 Go 语言诞生以来，它就开始不断侵蚀 Java 、C、C++ 语言的领地。今年下半年 Go 语言发布了 1.11 版本，引...

2.2w 179 49

Col0ring 3月前 WebAssembly 前端

WebAssembly 与 AssemblyScript 初探

WebAssembly 是一种在Web上运行字节码的编译型语言。 相对于 Javascript，WebAssembly 提供了可预测的性...

阿里巴巴业务中台前端 2月前 WebAssembly 前端

十年磨一剑，WebAssembly是如何诞生的?

创造一个编程语言最好的时间是10年前，其次是现在。 从Emscripten到asm.js再到WebAssembly，从一个业余项...

476 11 1

腾讯IVWEB团队 2年前 前端 WebAssembly

WebAssembly 不完全指北

随着JavaScript的快速发展，目前它已然成为最流行的编程语言之一，这背后正是 Web 的发展所推动的。但是随着...

9892 183 17

lencx 6月前 WebAssembly 前端 后端

Vite与Rust邂逅

WebAssembly: WebAssembly（缩写为Wasm）是基于堆栈的虚拟机的二进制指令格式。...

1705 28 评论

吃凹吃凹 1月前 Rust WebAssembly

Yew 实现路由

使用 Rust 进行 Webassembly 开发，并通过 Yew 框架实现路由。Yew 是一个使用 WebAssembly 创建多线程前端...

185 2 评论