陈卷毛 Lv1
2020年02月15日 阅读 1500

关注

Wasm介绍之2：指令集和栈

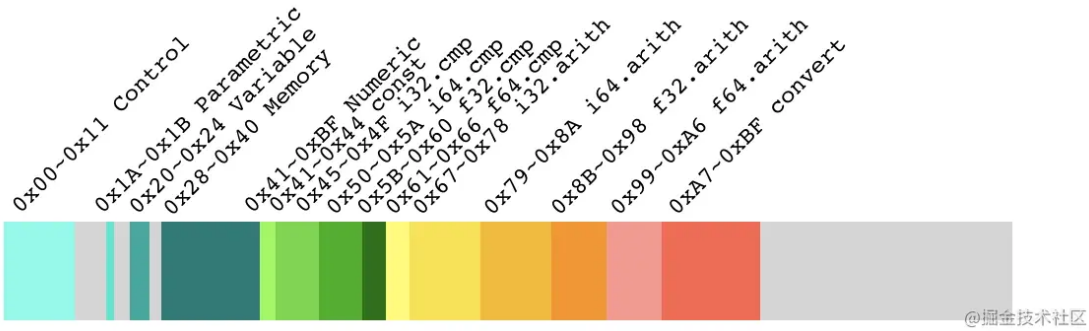
上一篇文章介绍了WebAssembly（后文简称Wasm）二进制格式，这一篇文章将介绍Wasm指令集、操作数栈和部分指令。

Wasm指令集

和真实的机器码一样，Wasm二进制文件中的代码也由一条一条的**指令**构成。同样，Wasm指令也包含两部分信息：**操作码**（Opcode）和**操作数**（Operands）。Wasm操作码固定为一个字节，因此最多能表示256条指令，这一点和Java字节码一样。Wasm1.0规范一共定义了172条指令，这些指令按功能可以分为5大类，分别是：

- **控制指令**（Control Instructions），共13条。
- **参数指令**（Parametric Instructions），共2条。
- **变量指令**（Variable Instructions），共5条。

可以看到，已经定义的指令中，有超过2/3属于数值指令。为了方便人类书写和理解，Wasm规范给也给每个操作码定义了**助记符**（Mnemonic），比如说操作码 `0x41` 的助记符是 `i32.const` 。下面是已定义指令的操作码分布示意图：



有一部分指令需要携带一些信息，这些信息编码后紧跟在指令操作数的后面，叫做**静态立即参数**（Static Immediate Arguments，后文简称**立即数**）。 以 `i32.const` 指令为例，操作码 `0x41` 后面要跟一个编码后的32位整数。在后面的例子中，我们将用类似下面这样的示意图来表示编码后的指令：

复制代码

```
bytecode:
...][ i32.const ][ 123 ][...
```

和JVM等栈式虚拟机一样，大部分Wasm指令也会用到**操作数栈**（Operand Stack，后文简称**栈**）。这些指令从栈顶**弹出**一个或多个数，进行计算，然后把结果**推入**栈顶。被指令操作的这些栈顶元素叫做指令的**动态操作数**（Dynamic Operands，后文简称**操作数**）。在后面的例子中，我们将用类似下面这样的示意图来表示指令执行前后栈的状态（小箭头表示弹出或推入操作）：

复制代码

```
stack:
|          |          |          |
|    e     |↗         |          |
|    d     |↗        ↘|   d+e    |
|    c     |          |    c     |
|    b     |          |    b     |
|    a     |          |    a     |
└──────────┘          └──────────┘
   before                 after
```

参数指令和数值指令仅仅对栈进行操作，行为比较简单，由本文进行介绍。其他指令将在后续文章中介绍。

参数指令有两条： `drop` （操作码是 `0x1A` ）和 `select` （操作码是 `0x1B` ）。

drop

`drop` 指令，从栈顶弹出一个任意类型的操作数。 `drop` 指令没有立即数，下面是它的示意图：

复制代码

```
bytecode:
...][ drop ][...

stack:
|           |          |           |
|           |          |           |
|     d     |↗         |           |
|     c     |          |     c     |
|     b     |          |     b     |
|     a     |          |     a     |
└───────────┘          └───────────┘
```

select

`select` 指令先后从栈顶弹出3个操作数，如果最先弹出的操作数等于0则将第二个弹出的操作数推入栈，否则将第三个弹出的操作数推入栈。 `select` 指令也没有立即数，下面是它的示意图：

复制代码

```
bytecode:
...][ select ][...

stack:
|           |          |           |
|   e(i32)  |↗         |           |
|     d     |↗         |           |
|     c     |↗        ↘| e!=0?c:d  |
|     b     |          |     b     |
|     a     |          |     a     |
└───────────┘          └───────────┘
```

注意位于栈顶的操作数必须是 `i32` 类型，其余两个操作数必须有相同类型。当需要强调操作数的具体类型时，我们会在示意图中用圆括号标出类型。 `drop` 和 `select` 是比较特殊的两条指令，因为只有这两条指令没有将操作数的类型完全限定。对于其他的指令，所有操作数的类型都是完全限定的。

数值指令可以按操作数类型分成 `i32` 、 `i64` 、 `f32` 、 `f64` 四组，每一组指令又可以按照操作进一步分为：

- 常量指令（Constant Instructions）
- 测试指令（Test Instructions）
- 比较指令（Comparison Instructions）
- 算术指令（Arithmetic Instructions）
  - 一元（Unary）算术指令
  - 二元（Binary）算术指令
- 转换指令（Conversion Instructions）

除常量指令外，其余数值指令都没有立即数。

常量指令

常量指令将立即数推入栈顶，以 `i32.const` 指令（操作码 `0x41` ）为例，下面是它的示意图：

复制代码

```
bytecode:
...][ i32.const ][ 123 ][...

stack:
|           |          |           |
|           |         ↘|  123(i32) |
|     d     |          |     d     |
|     c     |          |     c     |
|     b     |          |     b     |
|     a     |          |     a     |
└───────────┘          └───────────┘
```

常量指令一共有四条，另外三条是： `i64.const` （操作码 `0x42` ）、 `f32.const` （操作码 `0x43` ）、 `f64.const` （操作码 `0x44` ）。不难发现，Wasm操作码助记符的命名规则是：如果指令执行后栈顶元素的类型是 `t` ，那么助记符就以 `t.` 开头。

测试指令

测试指令从栈顶弹出一个操作数，测试它是否是0，如果是则将 `i32` 类型1推入栈，否则将 `i32` 类型0推入栈。测试指令只有两条： `i32.eqz` （操作码 `0x45` ）和 `i64.eqz` （操作码 `0x50` ）。以 `i64.eqz` 指令为例，下面是它的示意图：

```
stack:
|           |          |           |
|           |          |           |
|   d(i64)  |↗        ↘| d==0(i32) |
|     c     |          |     c     |
|     b     |          |     b     |
|     a     |          |     a     |
└───────────┘          └───────────┘
```

可以看到，测试指令的结果其实是**布尔值**，只不过Wasm没有定义 `bool` 类型，而是用 `i32` 类型来表示（1表示 `ture` ，0表示 `false` ）。

比较指令

比较指令从栈顶弹出两个相同类型的操作数，进行比较，然后将结果压栈。和测试指令一样，比较指令的结果也是布尔值（也就是 `i32` 类型）。以 `i64.lt_s` 指令（操作码 `0x53` ）为例，下面是它的示意图：

复制代码

```
bytecode:
...][ i64.lt_s ][...

stack:
|           |          |           |
|   e(i64)  |↗         |           |
|   d(i64)  |↗        ↘|  d<e(i32) |
|     c     |          |     c     |
|     b     |          |     b     |
|     a     |          |     a     |
└───────────┘          └───────────┘
```

除了**等于**（eq），还有进行**不等于**（ne）、**小于**（le）、**大于**（gt）、**小于等于**（le）、**大于等于**（ge）比较的指令，这里就不一一介绍了。需要说明的是，对于有些整数类型的指令，需要明确指出如何解释操作数：将其当成**有符号数**（Signed，助记符带 `_s` 后缀）还是**无符号数**（Unsigned，助记符带 `_u` 后缀）。这类指令一般是成对儿出现，比如上面例子中的 `i64.lt_s` 指令，与之对应的还有 `i64.lt_u` 指令（操作码 `0x54` ）。

一元算术运算

复制代码

```
bytecode:
...][ f32.neg ][...

stack:
|           |          |           |
|           |          |           |
|   d(f32)  |↗        ↘|  -d(f32)  |
|     c     |          |     c     |
|     b     |          |     b     |
|     a     |          |     a     |
└───────────┘          └───────────┘
```

二元算术运算

二元算术指令从栈顶弹出两个操作数，进行计算，然后将结果推入栈顶。以 `f32.sub` 指令（操作码 `0x93` ）为例，下面是它的示意图：

复制代码

```
bytecode:
...][ f32.sub ][...

stack:
|           |          |           |
|   e(i64)  |↗         |           |
|   d(i64)  |↗        ↘|  d-e(i64) |
|     c     |          |     c     |
|     b     |          |     b     |
|     a     |          |     a     |
└───────────┘          └───────────┘
```

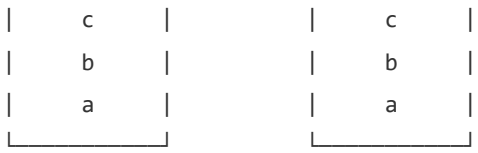
类型转换

类型转换指令从栈顶弹出一个操作数，进行类型转换，然后把结果推入栈顶。如果操作数在类型转换之前的类型是 `t` ，之后的类型是 `t'` ，转换操作是 `conv` ，则指令的助记符是 `t'.conv_t` 。以 `i32.wrap_i64` （操作码 `0xA7` ）指令为例，下面是它的示意图：

复制代码

```
bytecode:
...][ i32.wrap_i64 ][...

stack:
```



比较、算 术、转换指令数量比较多，本文无法一一介绍，请读者参考Wasm规范。

文章分类 阅读 文章标签 WebAssembly

陈卷毛 Lv1
获得点赞 5 · 获得阅读 8,639

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

吕小鸣 2天前 前端 WebAssembly

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

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...

201 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。

452 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...

979 11 评论

doodlewind 9月前 JavaScript WebAssembly

我把世界上第一个 JS 引擎编译回了 JS

1995 年，在我刚满周岁的时候，大洋彼岸有个叫 Brendan Eich 的人在十天内创造了一门...

4498 79 13

jordiwang 1年前 WebAssembly

7072 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...

823 12 7

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...

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