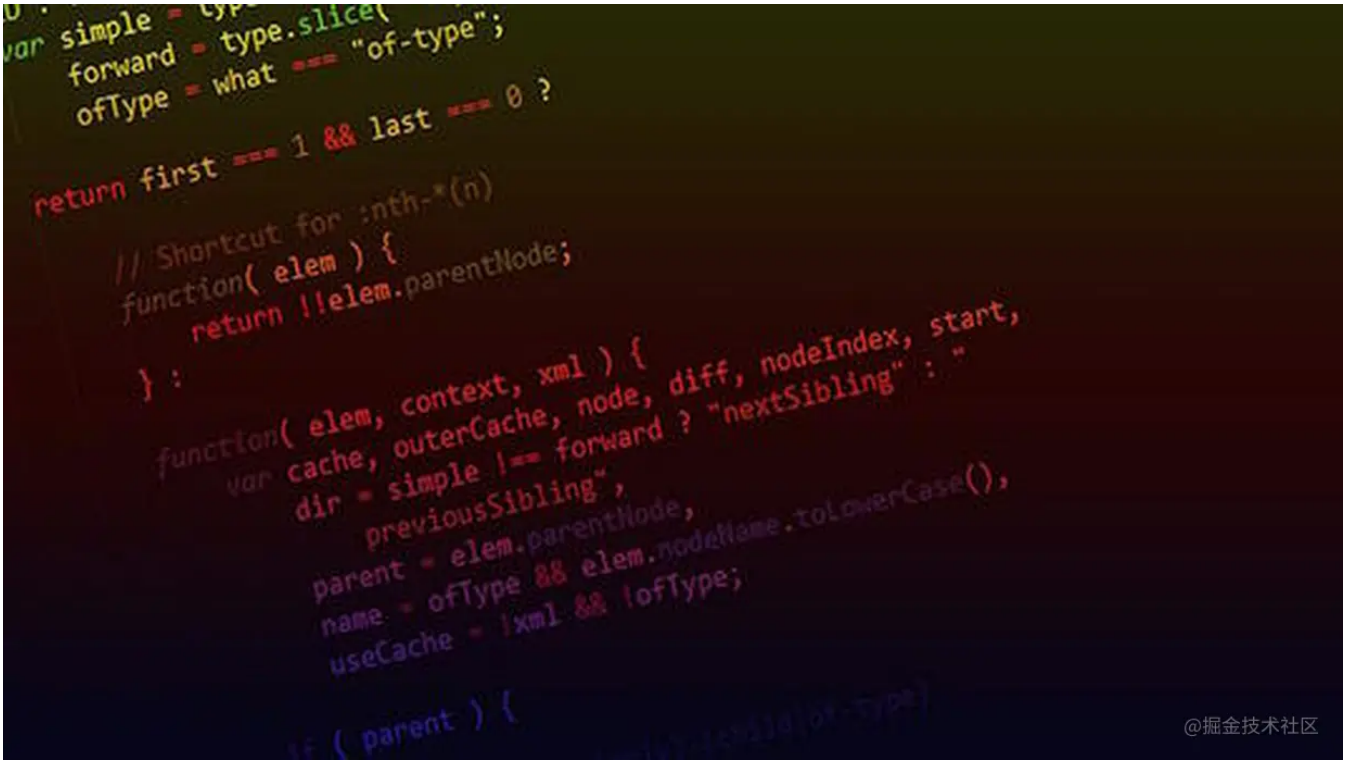
陈卷毛 Lv1
2020年03月05日 阅读 251

关注



Wasm介绍之5：控制指令

WebAssembly（简称Wasm）控制指令一共有11条，其中 `unreachable` 指令（操作码 `0x00` ）和 `nop` 指令（操作码 `0x01` ）比较简单，不介绍。 `call` 指令（操作码 `0x10` ）已经在上一篇文章里介绍， `call_indirect` 指令（操作码 `0x11` ）将在下一篇文章里介绍。本文重点讨论 `block` （操作码 `0x02` ）、 `loop` （操作码 `0x03` ）、 `if` （操作码 `0x04` ）、 `br` （操作码 `0x0C` ）、 `br_if` （操作码 `0x0D` ）、 `br_table` （操作码 `0x0E` ）和 `return` （操作码 `0x0F` ）这7条指令。

block

`block` 指令相当于一个无参的内联（inline）函数调用。函数的返回值类型，也就是 `block` 指令的结果类型（Result Type，在后面的示意图中简称 `rt` ）编码后存储在指令的第一个立即数参数里。函数的指令（可能有很多条）编码后存储在第二个立即数参数里。 `block` 指令必须以 `end` 指令（操作码

Wasm1.0规范规定 `block` 指令的结果不能超过一个，所以 `rt` 可以用一个字节表示： `0x40` 表示没有结果、 `0x7F` 表示 `i32` 类型、 `0x7E` 表示 `i64` 类型、 `0x7D` 表示 `f32` 类型、 `0x7C` 表示 `f64` 类型。根据讨论可知， `block` 指令在执行时不会使用栈上已经存在的任何操作数，执行完毕后可能会在栈顶留下一个操作数，下面是它的示意图：

复制代码

```
bytecode:
...][ block ][ rt ][ instrs ][ end ][...

stack:
|           |          |           |
|           |         ↘|   ?(rt)   | # exec(instrs)
|     d     |          |     d     |
|     c     |          |     c     |
|     b     |          |     b     |
|     a     |          |     a     |
└───────────┘          └───────────┘
```

下面是一个非常简单的WAT例子，展示了 `block` 指令的用法：

复制代码

```
(module
  (func (result i32)
    ;; ... other instructions
    (block (result i32)
      (i32.const 100)
    )
  )
)
```

loop

`loop` 指令和 `block` 指令非常相似，区别仅在于如何跳出控制块。后文介绍 `br` 指令时会进一步讨论这个区别，下面先给出一个跳转示意图：

复制代码

```
...][ block ][ rt ]...[ br ]...[ end ][...
                        |        ↑
                        └────────┘

...][ loop ][ rt ]...[ br ]...[ end ][...
       ↑               |
       └───────────────┘
```

if

和 block 指令类似， if 指令也类似于一个内联函数。区别主要有两点。第一， if 内联函数带一个 i32 类型的参数。第二， if 内联函数带有两份代码（两条分支），中间用 else 指令隔开。 if 指令执行时，会先从栈顶弹出这个 i32 类型的参数，如果参数值不等于0，则执行分支1代码，否则执行分支2代码。下面是 if 指令的示意图：

复制代码

```
bytecode:
...][ if ][ rt ][ instrs1 ][ else ][ instrs2 ][ end ][...

stack:
|           |          |           |
|   e(i32)  |↗        ↘|   ?(rt)   | # exec(e!=0?instrs1:instrs2)
|     d     |          |     d     |
|     c     |          |     c     |
|     b     |          |     b     |
|     a     |          |     a     |
└───────────┘          └───────────┘
```

也可以把 if 指令的 else 分支省略，但是这种情况下 if 指令不能有任何结果。下面是省略 else 分支时 if 指令的示意图：

复制代码

```
bytecode:
...][ if ][ 0x40 ][ instrs1 ][ end ][...

stack:
|           |          |           |
|   e(i32)  |↗         |           | # e==1?exec(instrs1)
|     d     |          |     d     |
|     c     |          |     c     |
|     b     |          |     b     |
|     a     |          |     a     |
└───────────┘          └───────────┘
```

下面这个WAT例子展示了如何用 if 指令实现 max() 函数：

复制代码

```
(module
  (func $max (param $a i32) (param $b i32) (result i32)
    (i32.gt_s (local.get $a) (local.get $b))
    (if (result i32)
```

```
    )
  )
```

br

br 指令（可以理解为break，或者branch）可以进行无条件跳转。和传统汇编语言里的 JUMP 指令不同， br 指令并不能跳转到任意位置，而只能跳出（相对于 block 和 if 指令而言是跳出，相对于 loop 指令而言是重新开始，后面不再强调）其他控制指令产生的控制块。 br 指令带有一个 u32 类型（32位无符号整数）的立即数参数，指定跳转的层数：0表示跳出当前循环，1代表跳出2层循环，以此类推。下面是一个WAT例子，展示了嵌套的 block ，以及 br 指令的用法：

复制代码

```
(module
  (func (export "main") (result i32)
    (i32.const 100) (block (result i32)
      (i32.const 200) (block (result i32)
        (i32.const 300) (block (result i32)
          (i32.const 123) (br n)
        ) (i32.add)
      ) (i32.add)
    ) (i32.add)
  )
)
```

假设上面例子中的 main() 函数已经执行到了 br 指令这里，下图展示了跳转层数分别为0、1、2、3时操作数栈的变化情况：

复制代码

```
bytecode:
...][ br ][ depth ][...

stack:
|           |  |           |  |           |  |           |  |           |
|           |  |           |  |           |  |           |  |           |
|    123    |  |    123    |  |           |  |           |  |           |
|    300    |  |    300    |  |    123    |  |           |  |           |
|    200    |  |    200    |  |    200    |  |    123    |  |           |
|    100    |  |    100    |  |    100    |  |    100    |  |    123    |
└───────────┘  └───────────┘  └───────────┘  └───────────┘  └───────────┘
 (before br)       n==0          n==1          n==2          n==3
```

`br_if` 指令从栈顶弹出一个 `i32` 类型的操作数，如果操作数的值为0，则不跳转，否则执行 `br` 逻辑。下面是 `br_if` 指令的示意图：

复制代码

```
bytecode:
...][ br_if ][ depth ][...

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

下面是一个稍微复杂一些的WAT例子，展示了如何用 `loop` 和 `br_if` 指令实现 `sum()` 函数：

复制代码

```
(module
  (func $sum (param $from i32) (param $to i32) (result i32)
    (local $n i32)

    (loop $l
      ;; $n += $from
      (local.set $n (i32.add (local.get $n) (local.get $from)))
      ;; $from++
      (local.set $from (i32.add (local.get $from) (i32.const 1)))
      ;; if $from <= $to { continue }
      (br_if $l (i32.le_s (local.get $from) (local.get $to)))
    )

    ;; return $n
    (local.get $n)
  )
)
```

请注意 `loop` 指令并不会自动循环，必须和 `br` 等跳转指令配合使用。

br_table

不管是 `br` 还是 `br_if` 指令，都只有一个立即数参数，能指定一个跳转深度。 `br_table` 指令打破了这种限制，可以带N+1个立即数参数，指定N+1个跳转深度。 `br table` 指令执行时，会从栈顶弹出一

复制代码

```
bytecode:
...][ br_table ][ labels... ][ default ][...

stack:
|           |          |           |
|   n(i32)  |↗         |           |
|     d     |          |     ?     | # d<len(labels)?br(labels[n]):br(default)
|     c     |          |           |
|     b     |          |           |
|     a     |          |           |
└───────────┘          └───────────┘
```

return

`return` 指令可以认为是 `br` 指令的特殊形式：直接跳出最外层循环（也就是整个函数）。 `return` 指令没有立即数参数，不画示意图了。下面的WAT例子展示了如何用 `block` 、 `br_table` 和 `return` 指令实现Go等高级语言中的 `switch-case` 语句：

复制代码

```
(module
  (func $select3 (param $n i32) (param i32 i32 i32) (result i32)
    (block
      (block
        (block
          (local.get $n)
          (br_table 0 1 2)
        )
        (return (local.get 1))
      )
      (return (local.get 2))
    )
    (return (local.get 3))
  )
)
```

如果把上面例子中定义的 `select3()` 函数翻译成Go语言代码的话，应该是下面这样：

复制代码

```
func select3(n, a, b, c int32) int32 {
  switch n {
    case 0 : return a
    case 1 : return b
    default: return c
```

*本文由CoinEx Chain开发团队成员Chase撰写。CoinEx Chain是全球首条基于Tendermint共识协议和Cosmos SDK开发的DEX专用公链，借助IBC来实现DEX公链、智能合约链、隐私链三条链合一的方式去解决可扩展性（Scalability）、去中心化（Decentralization）、安全性（security）区块链不可能三角的问题，能够高性能的支持数字资产的交易以及基于智能合约的Defi应用。

文章分类 代码人生 文章标签 WebAssembly

[Dart翻译]用Dart和Wasm做实验

将Dart编译为Wasm，并从Dart调用Wasm模块 WebAssembly（通常缩写为Wasm）是 "一种基于堆栈的虚拟机的...

173 5 评论

方石剑 1月前 Rust

WebAssembly会取代JavaScript吗？WASM 介绍和性能比较

WebAssembly是过去几十年来互联网最强大的创新之一，它是一个开放的标准，为可执行...

287 1 评论

sealyun 1月前 Go 前端 后端

rust+wasm写前端真香之路由

[sealer](https://github.com/alibaba/sealer)是阿里巴巴开源的基于kuberentes的集群镜像开源技术，可以把整个...

1674 7 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...

1136 11 评论

sealyun 1月前 Go 后端

rust+wasm写前端真香之嵌套与循环

[sealer](https://github.com/alibaba/sealer)是阿里巴巴开源的基于kuberentes的集群镜像开源技术，可以把整个...

111 2 评论

飞书技术 1月前 SQLite Rust 前端

飞书WASM实践——SQLite篇

SQLite是一个跨平台的关系型数据库，广泛使用于客户端开发，飞书也使用SQLite作为数据持久化存储；同时为了方...

3842 30 4

Michael_Yuan 1月前 Go WebAssembly

用 WasmEdge 和 YoMo 对实时数据流进行 AI 推理

我们将向你展示如何为基于 Tensorflow 的图片识别创建 Rust 函数，将其编译为...

209 2 评论

码上出彩 1月前 SVN

285 1 评论

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 2月前 云原生

使用 Docker 工具管理WasmEdge 中的 WebAssembly 程序 |

开发者可以利用 DockerHub 和 CRI-O 等 Docker 工具在 WasmEdge 中部署、管理和运...

391 4 评论

阿宝哥 2月前 前端 JavaScript

JavaScript 中如何实现并发控制?

本文介绍了在 JavaScript 中，如何基于 ES7 或 ES6 实现并发控制的两种不同方案，同时介...

1.4w 327 33

橙某人 3月前 Vue.js 前端

Vue3自定义指令-10个常见的实用指令，带详细讲解，快拿去收藏！！！

Vue 在除了提供默认的十几个内置的指令外，还允许开发人员根据实际情况自定义指令，那...

4792 164 36

耗子君QAQ 3月前 Vue.js 前端

Vue组件设计 | 实现水波涟漪效果的点击反馈指令

不知道小伙伴们有没有注意过这样一个细节，有的应用按钮，链接，可交互的卡片点击起来...

5240 111 30

Houtaroy 3月前 Vue.js 前端

Vue中的动态菜单和权限控制指令

需求可具体描述为如下内容: 根据权限动态筛选路由 根据权限控制组件的是否展示 我们默认后端权限接口可用, 且返...

2788 52 2

Huup_We 3月前 前端

VUE 自定义指令合集

Vue 自定义指令合集 在 Vue2.0 中，代码复用和抽象的主要形式是组件。然而，有的情况下，你仍然需要对普通

只要我E得够快 3月前 前端

前端展望-WebAssembly（wasm）技术入门

wasm技术帮助前端解决性能瓶颈，VR/图像视频编辑/3D游戏的希望。本文旨在了解其作用和使用方式。

496 4 评论

Rust_Magazine 4月前 Rust 后端

华为 | 基于 TVM Rust Runtime 和 WASM 沙箱运行 AI 模型 【Rust 中文精选 3 月刊】

基于TVM Rust Runtime和WASM沙箱运行AI模型 作者： 王辉 / 后期编辑： 张汉东 说明 本文介绍了一种WASM与...

956 3 评论

Patract 5月前 智能合约

访谈|探索 Wasm 合约的无限可能

**Wasm，即 WebAssembly，是一种用来补充 JS 在运行上不足的“低级”语言——基于二...

88 点赞 评论
