WASC: WASM 到 RISC-V 的 AOT 编译器

Mohanson, developer @Nervos

Github: https://github.com/mohanson

区块链虚拟机架构

EVM
- 它运行的很好
- 糟糕的设计

WASM
- 快速演进(不一定是优点)
- 广泛被使用

RISC-V
- 硬件规范

JVM
- Oracle😒

…

有人在尝试从 EVM 到 WASM

EVM -> wasm

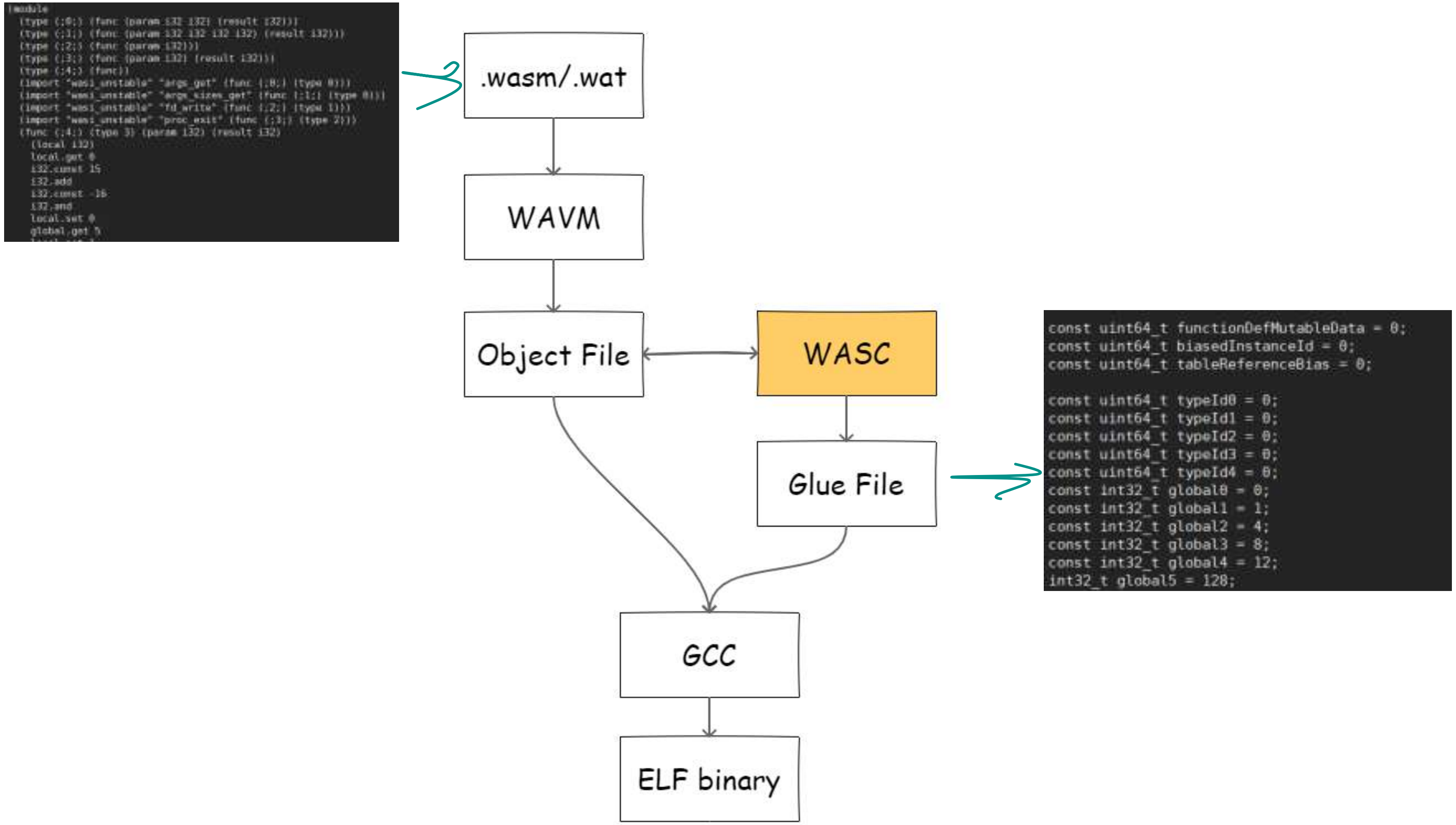
- Evm2wasm
- runevm
- Yevm

Solidity -> wasm

- Solang

WASM 2 RISC-V

WASC



(module
(type (;0;) (func (param i32 i32) (result i32)))
(type (;1;) (func (param i32 i32 i32 i32) (result i32)))
(type (;2;) (func (param i32)))
(type (;3;) (func (param i32) (result i32)))
(type (;4;) (func))
(import "wasi_unstable" "args_get" (func (;0;) (type 0)))
(import "wasi_unstable" "args_sizes_get" (func (;1;) (type 0)))
(import "wasi_unstable" "fd_write" (func (;2;) (type 1)))
(import "wasi_unstable" "proc_exit" (func (;3;) (type 2)))
(func (;4;) (type 3) (param i32) (result i32)
(local i32)
local.get 0
i32.const 15
i32.add
i32.const -16
i32.and
local.set 0
global.get 5
.wasm/.wat
WAVM
Object File
WASC
Glue File
GCC
ELF binary
const uint64_t functionDefMutableData = 0;
const uint64_t biasedInstanceId = 0;
const uint64_t tableReferenceBias = 0;
const uint64_t typeId0 = 0;
const uint64_t typeId1 = 0;
const uint64_t typeId2 = 0;
const uint64_t typeId3 = 0;
const uint64_t typeId4 = 0;
const int32_t global0 = 0;
const int32_t global1 = 1;
const int32_t global2 = 4;
const int32_t global3 = 8;
const int32_t global4 = 12;
int32_t global5 = 128;

WASC

也可以这么认为, WASC 的工作是使用一个极小的运行时替换了原来 WAVM 臃肿(相对的)的运行时

WAVM vs Wasmtime

WAVM

```
*UND*   0000000000000000 functionDefMutableDatas0
*UND*   0000000000000000 functionDefMutableDatas1
*UND*   0000000000000000 functionDefMutableDatas2
*UND*   0000000000000000 functionImport0
*UND*   0000000000000000 functionImport1
*UND*   0000000000000000 functionImport2
*UND*   0000000000000000 functionImport3
*UND*   0000000000000000 global5
*UND*   0000000000000000 memoryOffset0
*UND*   0000000000000000 typeId3
*UND*   0000000000000000 typeId4
```

函数元数据

外部函数

全局变量

线性内存

函数签名

wasmtime

```
*UND*   0000000000000000 .hidden _wasm_function_0
*UND*   0000000000000000 .hidden _wasm_function_1
*UND*   0000000000000000 .hidden _wasm_function_2
*UND*   0000000000000000 .hidden _wasm_function_3
*UND*   0000000000000000 .hidden wasmtime_i64_udiv
*UND*   0000000000000000 .hidden wasmtime_i64_udiv
*UND*   0000000000000000 .hidden wasmtime_i64_sdiv
*UND*   0000000000000000 .hidden wasmtime_i64_urem
*UND*   0000000000000000 .hidden wasmtime_i64_srem
*UND*   0000000000000000 .hidden wasmtime_i64_ishl
*UND*   0000000000000000 .hidden wasmtime_i64_ushr
*UND*   0000000000000000 .hidden wasmtime_i64_sshr
*UND*   0000000000000000 .hidden wasmtime_f32_ceil
*UND*   0000000000000000 .hidden wasmtime_f32_floor
*UND*   0000000000000000 .hidden wasmtime_f32_trunc
*UND*   0000000000000000 .hidden wasmtime_f32_nearest
*UND*   0000000000000000 .hidden wasmtime_f64_ceil
*UND*   0000000000000000 .hidden wasmtime_f64_floor
*UND*   0000000000000000 .hidden wasmtime_f64_trunc
*UND*   0000000000000000 .hidden wasmtime_f64_nearest
```

外部函数

算数运算函数

WASC

```
echo_build/
├── echo
├── echo.c
├── echo_glue.h
├── echo.o
├── echo_precompiled.wasm
└── platform
    ├── common
    │   ├── wasi.h
    │   └── wavm.h
    ├── posix_x86_64_wasi.h
    └── posix_x86_64_wasi_runtime.S
```

Glue File

Object File

Platform based

WASC 胶水文件

```
*UND*   0000000000000000 functionDefMutableDatas0
*UND*   0000000000000000 functionDefMutableDatas1
*UND*   0000000000000000 functionDefMutableDatas2
*UND*   0000000000000000 functionImport0
*UND*   0000000000000000 functionImport1
*UND*   0000000000000000 functionImport2
*UND*   0000000000000000 functionImport3
*UND*   0000000000000000 global5
*UND*   0000000000000000 memoryOffset0
*UND*   0000000000000000 typeId3
*UND*   0000000000000000 typeId4
```

```c
const uint64_t functionDefMutableData = 0;
const uint64_t biasedInstanceId = 0;
const uint64_t tableReferenceBias = 0;

const uint64_t typeId0 = 0;
const uint64_t typeId1 = 0;
const uint64_t typeId2 = 0;
const uint64_t typeId3 = 0;
const uint64_t typeId4 = 0;
const int32_t global0 = 0;
const int32_t global1 = 1;
const int32_t global2 = 4;
const int32_t global3 = 8;
const int32_t global4 = 12;
int32_t global5 = 128;
#define wavm_wasi_args_get functionImport0
extern wavm_ret_int32_t (functionImport0) (void*, int32_t, int32_t);
#define wavm_wasi_args_sizes_get functionImport1
extern wavm_ret_int32_t (functionImport1) (void*, int32_t, int32_t);
#define wavm_wasi_fd_write functionImport2
extern wavm_ret_int32_t (functionImport2) (void*, int32_t, int32_t, int32_t, int32_t);
#define wavm_wasi_proc_exit functionImport3
```

使用 AssemblyScript 进行合约编程

Nervos CKB-VM 是一个 RISC-V

虚拟机，支持解释执行，JIT 或

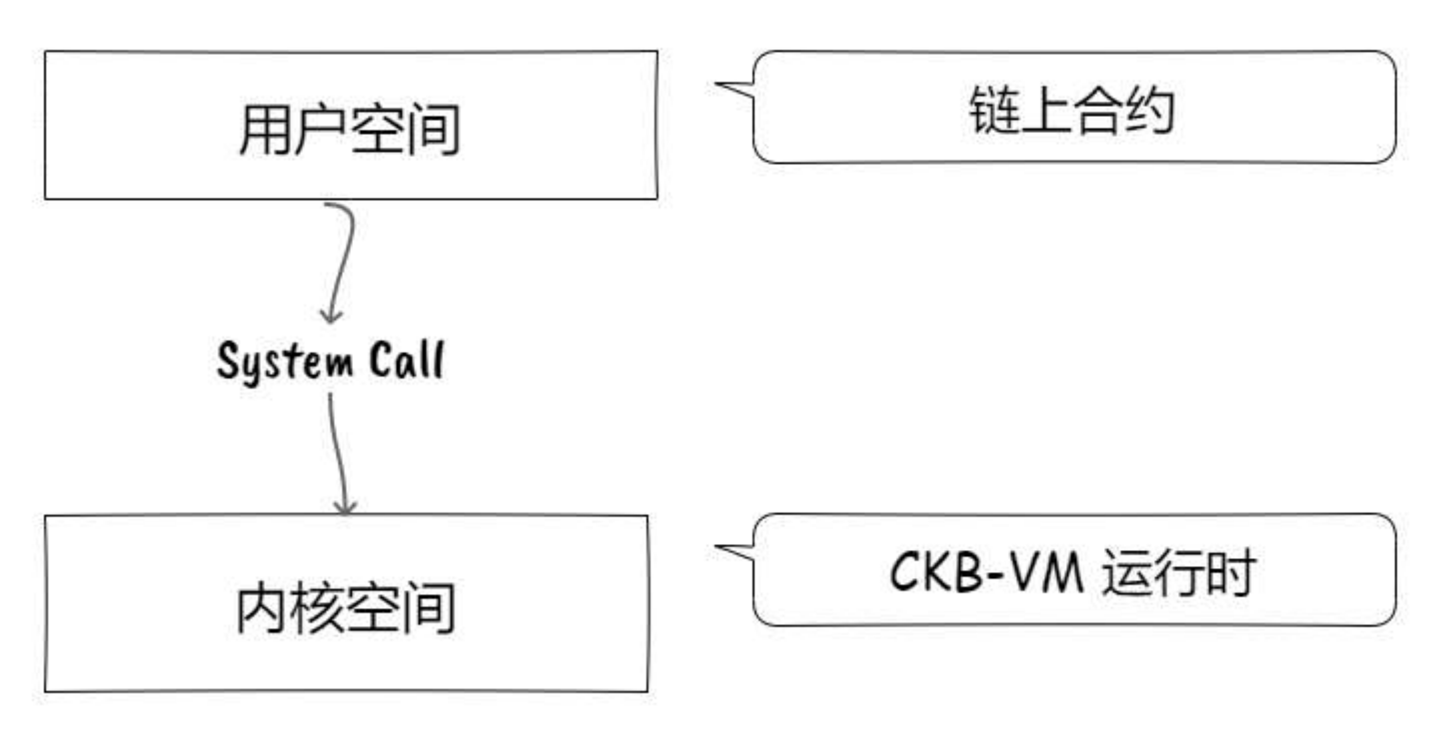
AOT 执行 RISC-V 可执行文件.

使用 AssemblyScript 进行合约编程

2.9 Environment Call and Breakpoints

31 — 20	19 — 15	14 — 12	11 — 7	6 — 0
funct12	rs1	funct3	rd	opcode
12	5	3	5	7
ECALL	0	PRIV	0	SYSTEM
EBREAK	0	PRIV	0	SYSTEM

使用 AssemblyScript 进行合约编程



使用 AssemblyScript 进行合约编程

```c
static inline long __internal_syscall(long n, long _a0, long _a1, long _a2,
                                      long _a3, long _a4, long _a5)
{
    register long a0 asm("a0") = _a0;
    register long a1 asm("a1") = _a1;
    register long a2 asm("a2") = _a2;
    register long a3 asm("a3") = _a3;
    register long a4 asm("a4") = _a4;
    register long a5 asm("a5") = _a5;
    register long syscall_id asm("a7") = n;
    asm volatile("scall"
                 : "+r"(a0)
                 : "r"(a1), "r"(a2), "r"(a3), "r"(a4), "r"(a5), "r"(syscall_id));
    return a0;
}

#define syscall(n, a, b, c, d, e, f)                                         \
    __internal_syscall(n, (long)(a), (long)(b), (long)(c), (long)(d), (long)(e), \
                       (long)(f))
```

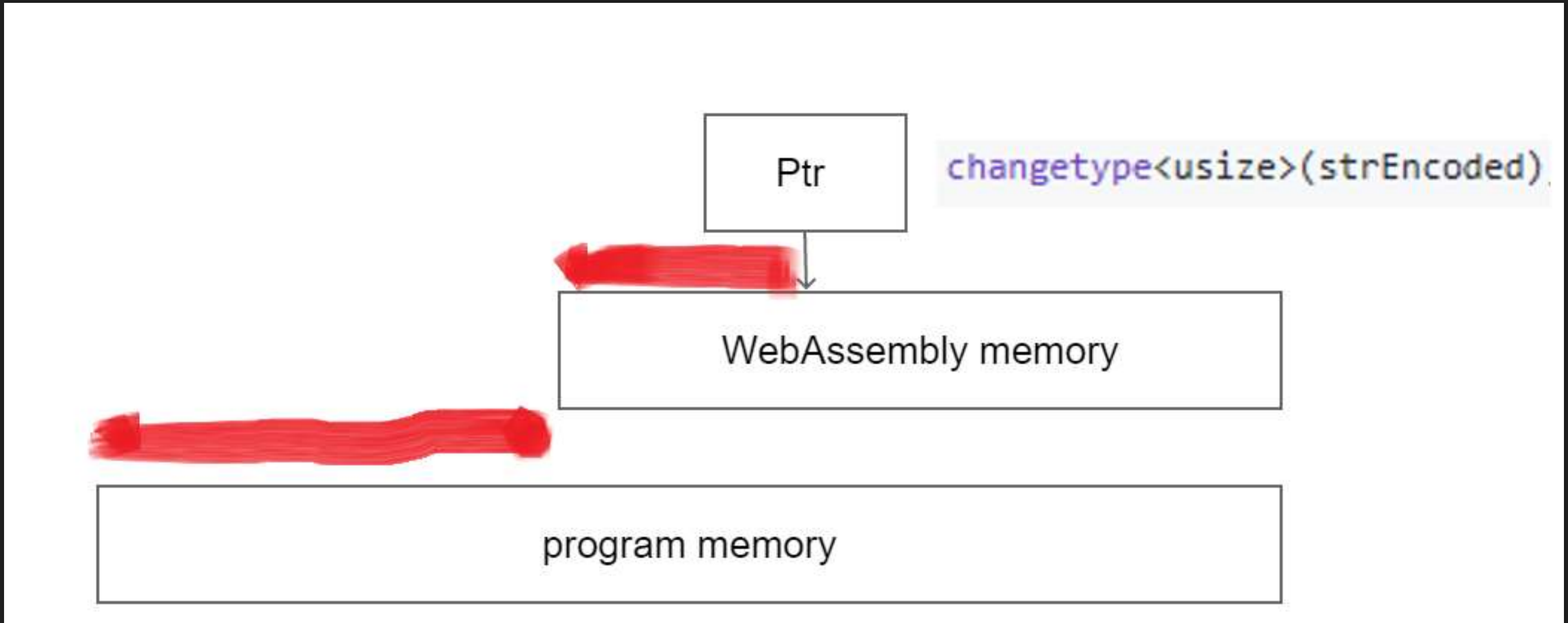
使用 AssemblyScript 进行合约编程

```
char *s = "Hello World!";
syscall(xx, &s[0], strlen(s), 0, 0, 0, 0);
```

但是这在 AssemblyScript 下是不工作的

```
const str = "Hello World!";
let strEncoded = String.UTF8.encode(str);
syscall(xx, changetype<usize>(strEncoded), strEncoded.byteLength, 0, 0, 0);
```

使用 AssemblyScript 进行合约编程



```c
#define syscall(n, a, b, c, d, e, f)                                  \
    __internal_syscall(n, (long)(a), (long)(b), (long)(c), (long)(d), (long)(e), \
                       (long)(f))

#ifdef MEMORY0_DEFINED
wavm_ret_int64_t wavm_env_syscall(void *dummy, int64_t n, int64_t _a0, int64_t _a1, int64_t _a2, int64_t _a3, int64_t _a4, int64_t _a5, int64_t mode)
{
    wavm_ret_int64_t ret;
    ret.dummy = dummy;
    if (mode & 0b100000)
    {
        _a0 = (int64_t)&memoryOffset0.base[0] + _a0;
    }
    if (mode & 0b010000)
    {
        _a1 = (int64_t)&memoryOffset0.base[0] + _a1;
    }
    if (mode & 0b001000)
    {
        _a2 = (int64_t)&memoryOffset0.base[0] + _a2;
    }
    if (mode & 0b000100)
    {
        _a3 = (int64_t)&memoryOffset0.base[0] + _a3;
    }
    if (mode & 0b000010)
    {
        _a4 = (int64_t)&memoryOffset0.base[0] + _a4;
    }
    if (mode & 0b000001)
    {
        _a5 = (int64_t)&memoryOffset0.base[0] + _a5;
    }
    ret.value = syscall(n, _a0, _a1, _a2, _a3, _a4, _a5);
    return ret;
}
```

吐槽一

有许多区块链直接采用 WebAssembly, 比如 Substrate, EOS, 以及未来的以太坊 2.0. 但它们的使用方式存在问题: 比如 EOS, 它只支持使用 C++ 来编写合约代码; Substrate 只支持使用 Rust + 宏的方式来编写合约代码, 以太坊 2.0 则使用预编译的合约来扩展 WebAssembly. 它们的实现是互不兼容的, 抛弃了 WebAssembly 最大的优点即通用性.

WASC 设计的 WebAssembly on RISC-V 方案由于与宿主环境只有 syscall 一种交互方式, 使得可以在 CKB-VM 上运行任何支持 WebAssembly 后端的语言. 另外相比起 C/C++ 与 Rust, AssemblyScript 等语言更加易学和使用.

吐槽 二

WebAssembly 的测试用例真的是一言难尽…

1. 在一个 case 里面测试复数条指令
2. 后一个 case 依赖前面 case 的执行结果