以太坊智能合约 OPCODE 逆向之调试器篇

2018年09月04日
区块链 (/category/blockchain/) · 404专栏 (/category/404team/)

作者：Hcamael@知道创宇404区块链安全研究团队
时间：2018/09/04

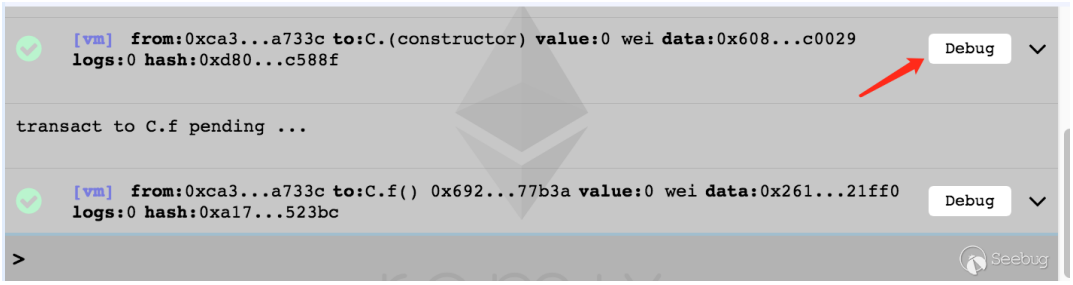
上一篇《以太坊智能合约 OPCODE 逆向之理论基础篇》(https://paper.seebug.org/640/)，对智能合约的OPCODE的基础数据结构进行了研究分析，本篇将继续深入研究OPCODE，编写一个智能合约的调试器。

Remix调试器

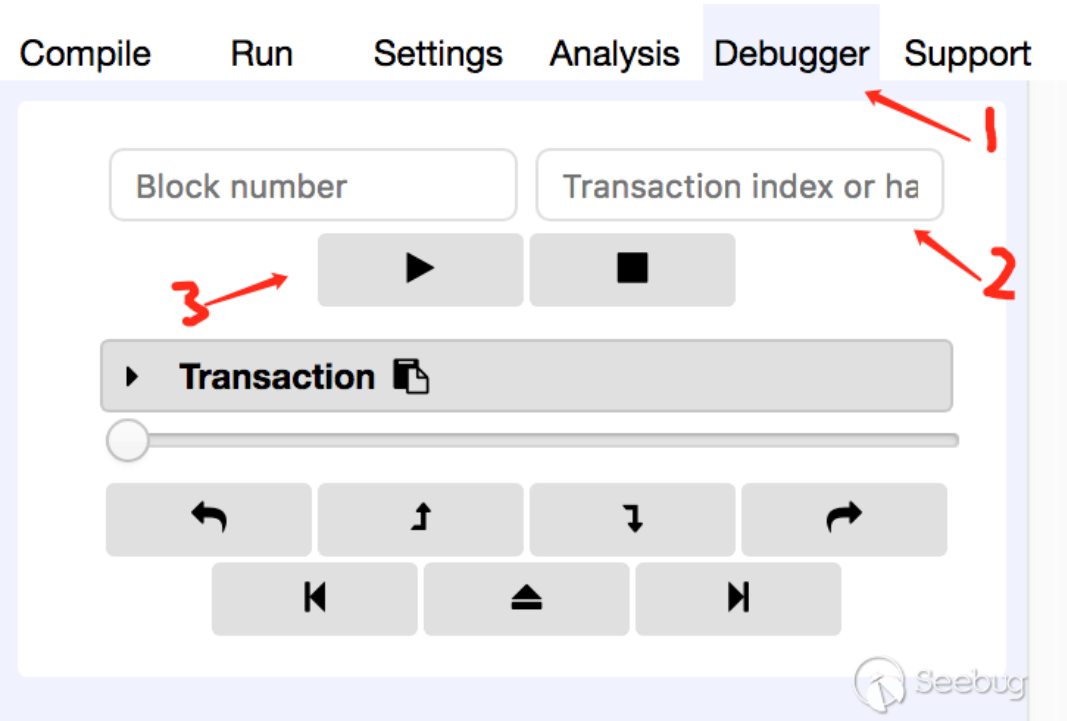
Remix带有一个非常强大的 Debugger ，当我的调试器写到一半的时候，才发现了Remix自带调试器的强大之处，本文首先，对Remix的调试器进行介绍。

能调试的范围：

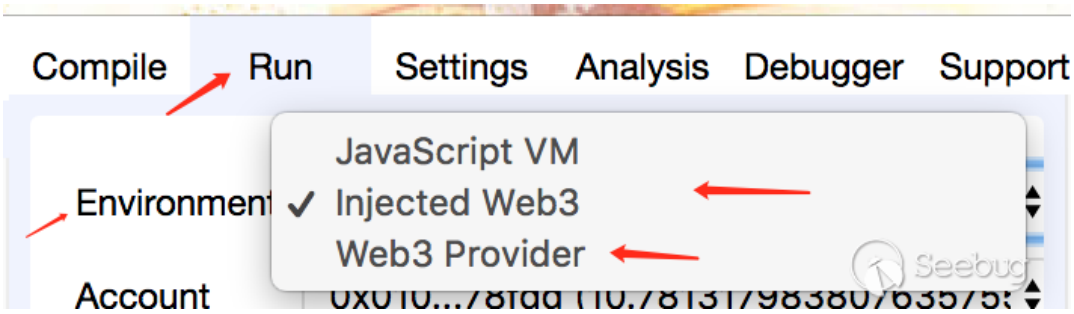
1. 在Remix上进行每一个操作(创建合约/调用合约/获取变量值)时，在执行成功后，都能在下方的控制界面点击 DEBUG 按钮进行调试



2. Debugger能对任意交易进行调试，只需要在调试窗口输入对应交易地址



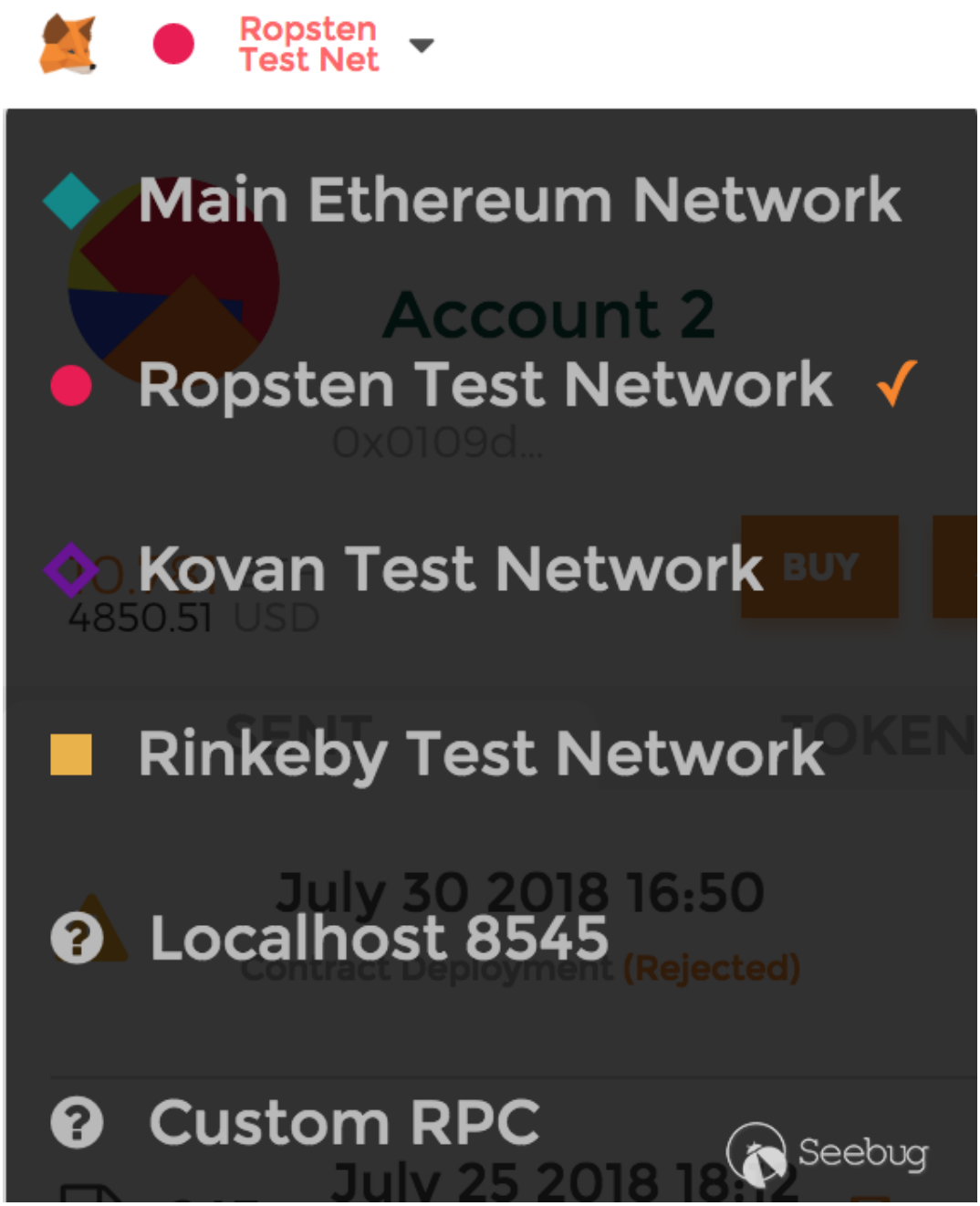
3. 能对公链，测试链，私链上的任意交易进行调试



点击 `Environment` 可以对区块链环境进行设置，选择 `Injected Web3`，环境取决去浏览器安装的插件

比如我，使用的浏览器是 `Chrome`，安装的插件是MetaMask (https://chrome.google.com/webstore/detail/metamask/nkbihfbeogaeaoehlefnkodbefgpgknn?hl=en-US)

通过 `MetaMask` 插件，我能选择环境为公链或者是测试链，或者是私链

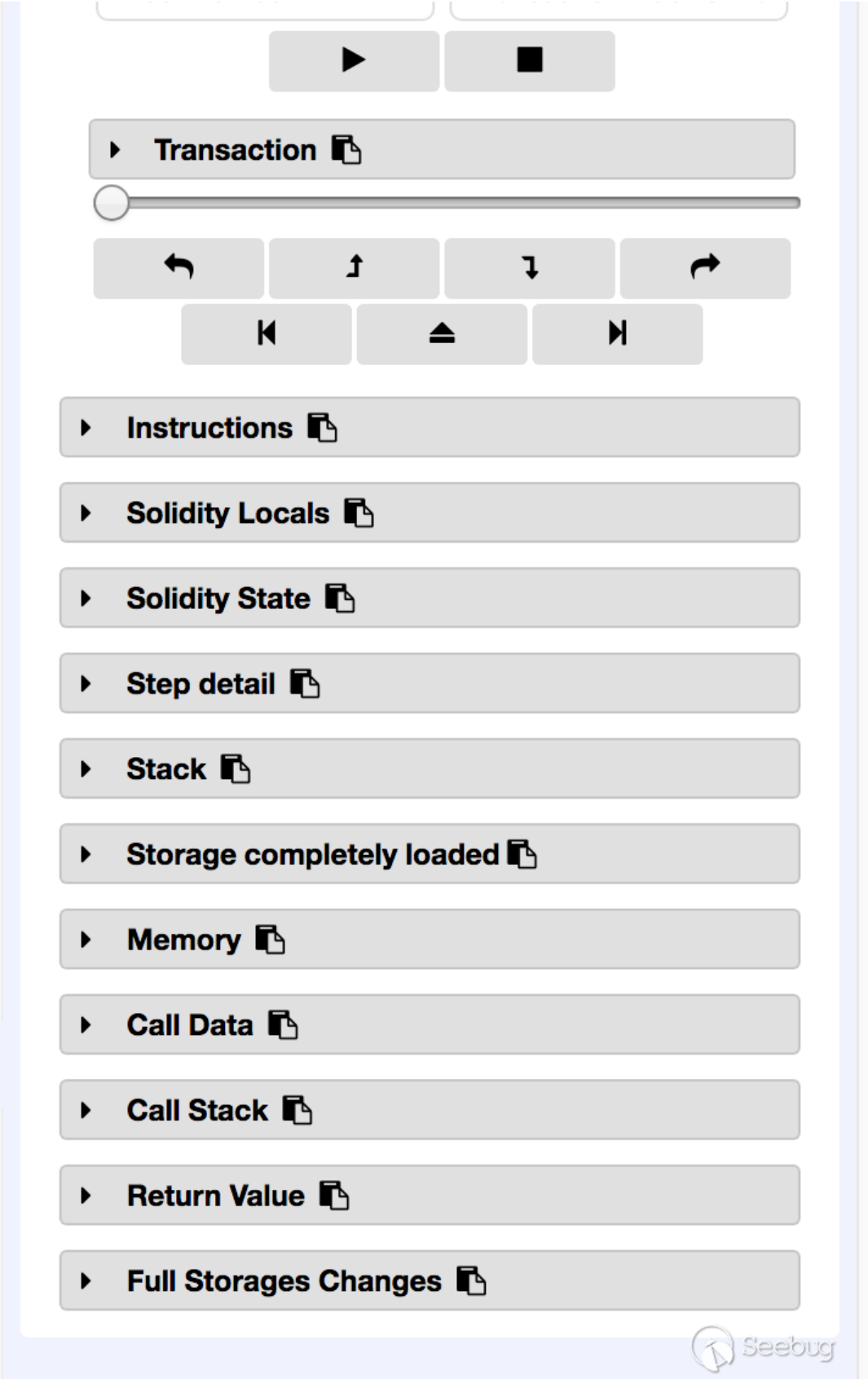


当 `Environment` 设置为 `Web3 Provider` 可以自行添加以太坊区块链的RPC节点，一般是用于设置环境为私链

4. 在JavaScript的EVM环境中进行调试

见3中的图，把 `Environment` 设置为 `JavaScript VM` 则表示使用本地虚拟环境进行调试测试

在调试的过程中能做什么？



Remix的调试器只提供了详细的数据查看功能，没法在特定的指令对 STACK/MEM/STORAGE 进行操作

在了解清楚Remix的调试器的功能后，感觉我进行了一半的工作好像是在重复造轮子。

之后仔细思考了我写调试器的初衷，今天的WCTF有一道以太坊智能合约的题目，因为第一次认真的逆向EVM的OPCODE，不熟练，一个下午还差一个函数没有逆向出来，然后比赛结束了，感觉有点遗憾，如果当时能动态调试，可能逆向的速度能更快。

Remix的调试器只能对已经发生的行为(交易)进行调试，所以并不能满足我打CTF的需求，所以对于我写的调试器，我转换了一下定位：调试没有源码，只有OPCODE的智能合约的逻辑，或者可以称为离线调试。

调试器的编写

智能合约调试器的编写，我认为最核心的部分是实现一个OPCODE解释器，或者说是自己实现一个EVM。

实现OPCODE解释器又分为两部分，1. 设计和实现数据储存器(把STACK/MEM/STORAGE统称为数据储存器)，2. 解析OPCODE指令

数据储存器

STACK

根据OPCODE指令的情况，EVM的栈和计算机的栈数据结构是一个样的，先入先出，都有 PUSH 和 POP 操作。不过EVM的栈还多了 SWAP 和 DUP 操作，栈交换和栈复制，如下所示，是我使用 Python 实现的EVM栈类:

```
class STACK(Base):
    """
    evm stack
    """
    stack: [int]
    max_value: int
    def __init__(self):
        self.stack = []
        self.max_value = 2**256
    def push(self, data: int):
        """
        OPCODE: PUSH
        """
        self.stack.append(data % self.max_value)
    def pop(self) -> (int):
        """
        OPCODE POP
        """
        return self.stack.pop()
    @Base.stackcheck
    def swap(self, n):
        """
        OPCODE: SWAPn(1-16)
        """
        tmp = self.stack[-n-1]
        self.stack[-n-1] = self.stack[-1]
        self.stack[-1] = tmp
    @Base.stackcheck
    def dup(self, n):
        """
        OPCODE: DUPn(1-16)
        """
        self.stack.append(self.stack[-n])
```

和计算机的栈比较，我觉得EVM的栈结构更像Python的List结构

计算机的栈是一个地址储存一个字节的数据，取值可以精确到一个字节，而EVM的栈是分块储存，每次PUSH占用一块，每次POP取出一块，每块最大能储存32字节的数据，也就是 `2^256-1` ，所以上述代码中，对每一个存入栈中的数据进行取余计算，保证栈中的数据小于 `2^256-1`

MEM

EVM的内存的数据结构几乎和计算机内存的一样，一个地址储存一字节的数据。在EVM中，因为栈的结构，每块储存的数据最大为 `256bits` ，所以当OPCODE指令需要的参数长度可以大于 `256bits` 时，将会使用到内存

如下所示，是我使用 `Python` 实现的MEM内存类:

```
class MEM(Base):
    """
    EVM memory
    """
    mem: bytearray

    max_value: int
    length: int
    def __init__(self):
        self.mem = bytearray(0)
        self.max_value = 2**256
        self.length = 0
        self.extend(1)
    @Base.memcheck
    def set(self, key: int, value: int):
        """
        OPCODE: MSTORE
        """
        value %= self.max
        self.mem[key: key+0x20] = value.to_bytes(0x20, "big")
        self.length += 0x20
    @Base.memcheck
    def set_byte(self, key: int, value: int):
        """
        OPCODE: MSTORE8
        """
        self.mem[key] = value  &amp; 0xff
        self.length += length
    @Base.memcheck
    def set_length(self, key: int, value: int, length: int):
        """
        OPCODE: XXXXCOPY
        """
        value %= (2**(8*length))
        data = value.to_bytes(length, "big")
        self.mem[key: key+length] = data
        self.length += length
    @Base.memcheck
    def get(self, key: int) -> (int):
        """
        OPCODE: MLOAD
        return uint256
        """
        return int.from_bytes(self.mem[key: key+0x20], "big", signed=False)
    @Base.memcheck
    def get_bytearray(self, key: int) -> (bytearray):
        """
        OPCODE: MLOAD
        return 32 byte array
        """
        return self.mem[key: key+0x20]
    @Base.memcheck
    def get_bytes(self, key: int) -> (bytes):
        """
        OPCODE: MLOAD
        return 32 bytes
        """
        return bytes(self.mem[key: key+0x20])
    @Base.memcheck
    def get_length(self, key:int , length: int) -> (int):
        """
        return mem int value
        """
        return int.from_bytes(self.mem[key: key+length], "big", signed=False)
    @Base.memcheck
    def get_length_bytes(self, key:int , length: int) -> (bytes):
        """
        return mem bytes value
```

```
        return mem bytes value
        """
        return bytes(self.mem[key: key+length])
    @Base.memcheck
    def get_length_bytearray(self, key:int , length: int) -> (bytearray):
        """

        return mem int value
        """
        return self.mem[key: key+length]
    def extend(self, num: int):
        """
        extend mem space
        """
        self.mem.extend(bytearray(256*num))
```

使用python3中的 `bytearray` 类型作为MEM的结构，默认初始化256B的内存空间，因为有一个OPCODE是 `MSIZE`：

> *Get the size of active memory in bytes.*

所以每次设置内存值时，都要计算 `active memory` 的size

内存相关设置的指令分为三类

1. MSTORE, 储存0x20字节长度的数据到内存中
2. MSTORE8, 储存1字节长度的数据到内存中
3. CALLDATACOPY(或者其他类似指令)，储存指定字节长度的数据到内存中

所以对应的设置了3个不同的储存数据到内存中的函数。获取内存数据的类似。

STORAGE

EVM的STORAGE的数据结构和计算机的磁盘储存结构相差就很大了，STORAGE是用来储存全局变量的，全局变量的数据结构我在上一篇文章中分析过，所以在用Python实现中，我把STORAGE定义为了字典，相关代码如下:

```
class STORAGE(Base):
    """
    EVM storage
    """
    storage: {str: int}
    max: int
    def __init__(self, data):
        self.storage = data
        self.max = 2**256
    @Base.storagecheck
    def set(self, key: str, value: int):
        self.storage[key] = value % self.max
    @Base.storagecheck
    def get(self, key: str) -> (int):
        return self.storage[key]
```

因为EVM中操作STORAGE的相关指令只有 `SSTORE` 和 `SLOAD` ，所以使用python的dict类型作为STORAGE的结构最为合适

解析OPCODE指令

对于OPCODE指令的解析难度不是很大，指令只占一个字节，所以EVM的指令最多也就256个指令(`0x00-0xff`)，但是有很多都是处于`UNUSE`，所以以后智能合约增加新指令后，调试器也要进行更新，因此现在写的代码需要具备可扩展性。虽然解析指令的难度不大，但是仍然是个体力活，下面先来看看OPCODE的分类

OPCODE分类

在以太坊官方黄皮书中，对OPCODE进行了相应的分类:

> *0s: Stop and Arithmetic Operations (从0x00-0x0f的指令类型是STOP指令加上算术指令)*
>
> *10s: Comparison & Bitwise Logic Operations (0x10-0x1f的指令是比较指令和比特位逻辑指令)*
>
> *20s: SHA3 (目前0x20-0x2f只有一个SHA3指令)*
>
> *30s: Environmental Information (0x30-0x3f是获取环境信息的指令)*
>
> *40s: Block Information (0x40-0x4f是获取区块信息的指令)*
>
> *50s: Stack, Memory, Storage and Flow Operations (0x40-0x4f是获取栈、内存、储存信息的指令和流指令(跳转指令))*
>
> *60s & 70s: Push Operations (0x60-0x7f是32个PUSH指令，PUSH1-PUSH32)*
>
> *80s: Duplication Operations (0x80-0x8f属于DUP1-DUP16指令)*
>
> *90s: Exchange Operations (0x90-0x9f属于SWAP1-SWAP16指令)*
>
> *a0s: Logging Operations (0xa0-0xa4属于LOG0-LOG4指令)*
>
> *f0s: System operations (0xf0-0xff属于系统操作指令)*

设计可扩展的解释器

首先，设计一个字节和指令的映射表:

```
import typing

class OpCode(typing.NamedTuple):
    name: str
    removed: int            # 参数个数
    args: int               # PUSH根据该参数获取opcode之后args字节的值作为PUSH的参数

_OPCODES = {
    '00': OpCode(name = 'STOP', removed = 0, args = 0),
    ......
}

for i in range(96, 128):
    _OPCODES[hex(i)[2:]] = OpCode(name='PUSH' + str(i - 95), removed=0, args=i-9
5)
......

# 因为编译器优化的问题，OPCODE中会出现许多执行不到的，UNUSE的指令，为防止解析失败，还要对UNU
SE的进行处理
for i in range(0, 256):
    if not _OPCODES.get(hex(i)[2:].zfill(2)):
            _OPCODES[hex(i)[2:].zfill(2)] = OpCode('UNUSE', 0, 0)
```

然后就是设计一个解释器类:

```
class Interpreter:
    """
    EVM Interpreter
    """
    MAX = 2**256
    over = 1
    store: EVMIO
    #############
    #  0s: Stop and Arithmetic Operations
    #############
    @staticmethod
    def STOP():
        """
        OPCODE: 0x00
        """
        Interpreter.over = 1
        print("========Program STOP=========")
    @staticmethod
    def ADD(x:int, y:int):
        """
        OPCODE: 0x01
        """
        r = (x + y) % Interpreter.MAX
        Interpreter.store.stack.push(r)
......
```

- MAX变量用来控制计算的结果在256bits的范围内
- over变量用来标识程序是否执行结束
- store用来访问runtime变量: STACK, MEM, STORAGE

在这种设计模式下，当解释响应的OPCODE，可以直接使用

```
args = [stack.pop() for _ in OpCode.removed]
getattr(Interpreter, OpCode.name)(*args)
```

特殊指令的处理思路

在OPCODE中有几类特殊的指令:

1. 获取区块信息的指令，比如:

> *NUMBER: Get the block's number*

该指令是获取当前交易打包进的区块的区块数(区块高度)，解决这个指令有几种方案:

- 设置默认值
- 设置一个配置文件，在配置文件中设置该指令的返回值
- 调试者手动利用调试器设置该值
- 设置RPC地址，从区块链中获取该值

文章的开头提过了对我编写的调试器的定位问题，也正是因为遇到该类的指令，才去思考调试器的定位。既然已经打包进了区块，说明是有交易地址的，既然有交易地址，那完全可以使用Remix的调试器进行调试。

所以对我编写的调试器有了离线调试器的定位，采用上述方法中的前三个方法，优先级由高到低分别是，手动设置>配置文件设置>默认设置

2. 获取环境信息指令，比如:

ADDRESS: Get address of currently executing account.

获取当前合约的地址，解决方案如下:

- 设置默认值
- 设置一个配置文件，在配置文件中设置该指令的返回值
- 调试者手动利用调试器设置该值

获取环境信息的指令，因为调试的是OPCODE，没有源码，不需要部署，所以是没法通过RPC获取到的，只能由调试者手动设置

3. 日志指令

LOG0-LOG4: Append log record with no topics.

把日志信息添加到交易的回执单中

```
> eth.getTransactionReceipt("0xe32b3751a3016e6fa5644e59cd3b5072f33f27f10242c7498
0409b637dbb3bdc")
{
  blockHash: "0x04b838576b0c3e44ece7279b3b709e336a58be5786a83a6cf27b4173ce317ad
3",
  blockNumber: 6068600,
  contractAddress: null,
  cumulativeGasUsed: 7171992,
  from: "0x915d631d71efb2b20ad1773728f12f76eeeeee23",
  gasUsed: 81100,
  logs: [],
  logsBloom: "0x00000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000",
  status: "0x1",
  to: "0xd1ceeeefa68a6af0a5f6046132d986066c7f9426",
  transactionHash: "0xe32b3751a3016e6fa5644e59cd3b5072f33f27f10242c74980409b637d
bb3bdc",
  transactionIndex: 150
}
```

上述就是获取一个交易的回执单，其中有一个 `logs` 列表，就是用来储存日志信息

既然是在调试OPCODE，那么记录日志的操作就是没有必要的，因为调试的过程中能看到储存器/参数的情况，所以对于这类指令的操作，完全可以直接输出，或者不做任何处理(直接pass)

4. 系统操作指令

这类指令主要是外部调用相关的指令，比如可以创建合约的 `CREATE`，比如能调用其他合约的 `CALL`，比如销毁自身，并把余额全部转给别人的 `SELFDESTRUCT`

这类的指令我认为的解决办法只有: 调试者手动利用调试器设置该指令的返回值

调用这类函数的时候，我们完全能看到详细的参数值，所以完全可以手动的进行创建合约，调用合约等操作

总结

在完成一个OPCODE的解释器后，一个调试器就算完成了3/4，剩下的工作就是实现自己想实现的调试器功能，比如下断点，查看栈内存储存数据等

下面放一个接近成品的演示gif图:

(/users/a
nicknam

知道创宇404区块链安全研究团队 (/users/author/?nickname=%E7%9F%A5%E9%81%93%E5%88%9B%E5%AE%87404%E5%8C%BA%E5%9D%97%E9%93%BE%E5%AE%89%E5%85%

阅读更多有关该作者 (/users/author/?nickname=%E7%9F%A5%E9%81%93%E5%88%9B%E5%AE%87404%E5%8C%BA%E5%9D%97%E9%93%BE%E5%AE%89%E5%85%A8%E7%A0%94%E7%A9%B6%E5%9B%A2%E9%98%9F)的文章