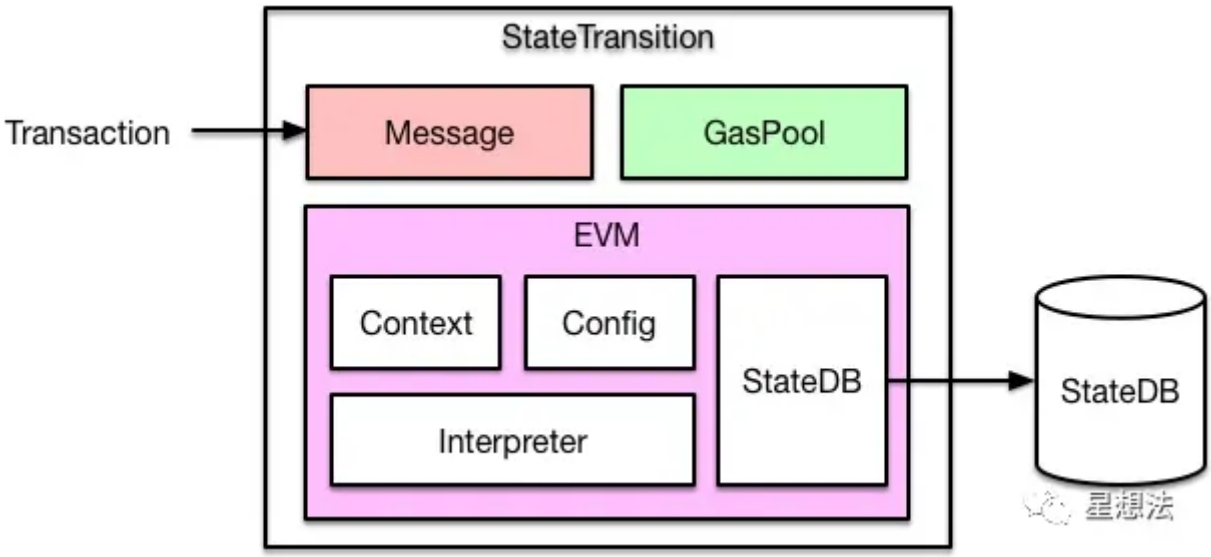
以太坊 – 深入浅出虚拟机

EVM (https://learnblockchain.cn/tags/EVM)

虚拟机用来执行以太坊上的交易，更改以太坊状态。交易分两种：普通交易和智能合约交易。在执行交易时需要支付油费。智能合约之间的调用有四种方式

以太坊虚拟机

以太坊虚拟机，简称EVM，是用来执行以太坊上的交易的。业务流程如下图：

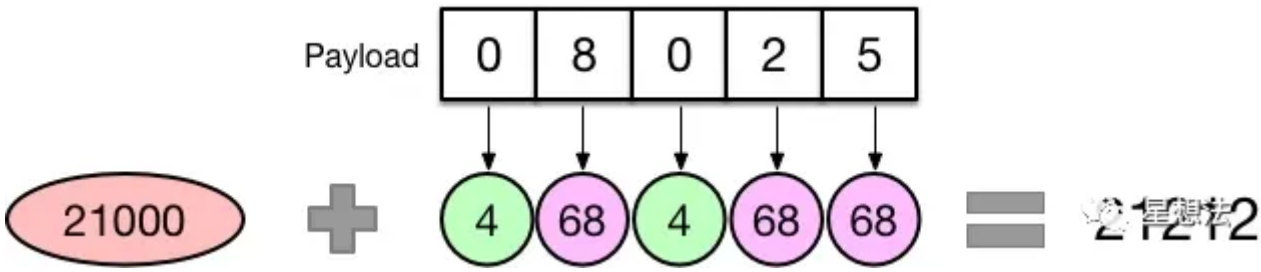


输入一笔交易，内部会转换成一个Message对象，传入EVM执行。

如果是一笔普通转账交易，那么直接修改 `StateDB` 中对应的账户余额即可。如果是智能合约(https://learnblockchain.cn/2018/01/04/understanding-smart-contracts/)的创建或者调用，则通过EVM中的解释器加载和执行字节码，执行过程中可能会查询或者修改StateDB。

固定油费（Intrinsic Gas）

每笔交易过来，不管三七二十一先需要收取一笔固定油费，计算方法如下：

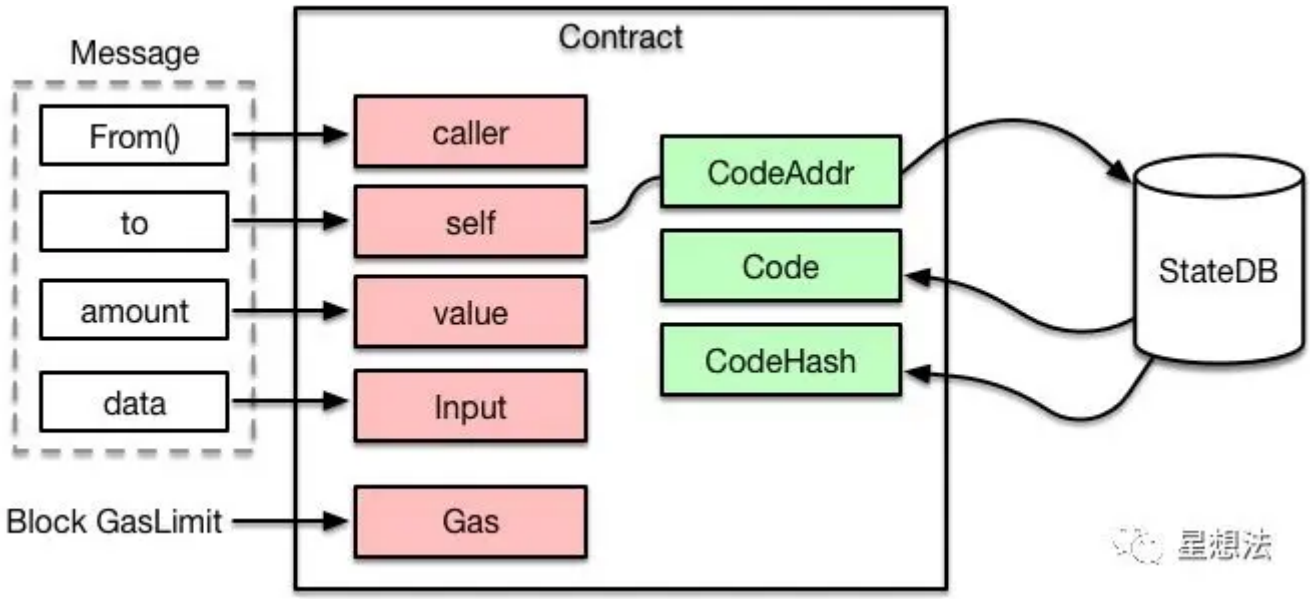


如果你的交易不带额外数据（Payload），比如普通转账，那么需要收取21000的油费。

如果你的交易携带额外数据，那么这部分数据也是需要收费的，具体来说是按字节收费：字节为0的收4块，字节不为0收68块，所以你会看到很多做合约优化的，目的就是减少数据中不为0的字节数量，从而降低油费gas (https://learnblockchain.cn/2019/06/11/gas-mean/)消耗。

生成Contract对象

交易会被转换成一个Message对象传入EVM，而EVM则会根据Message生成一个Contract对象以便后续执行：



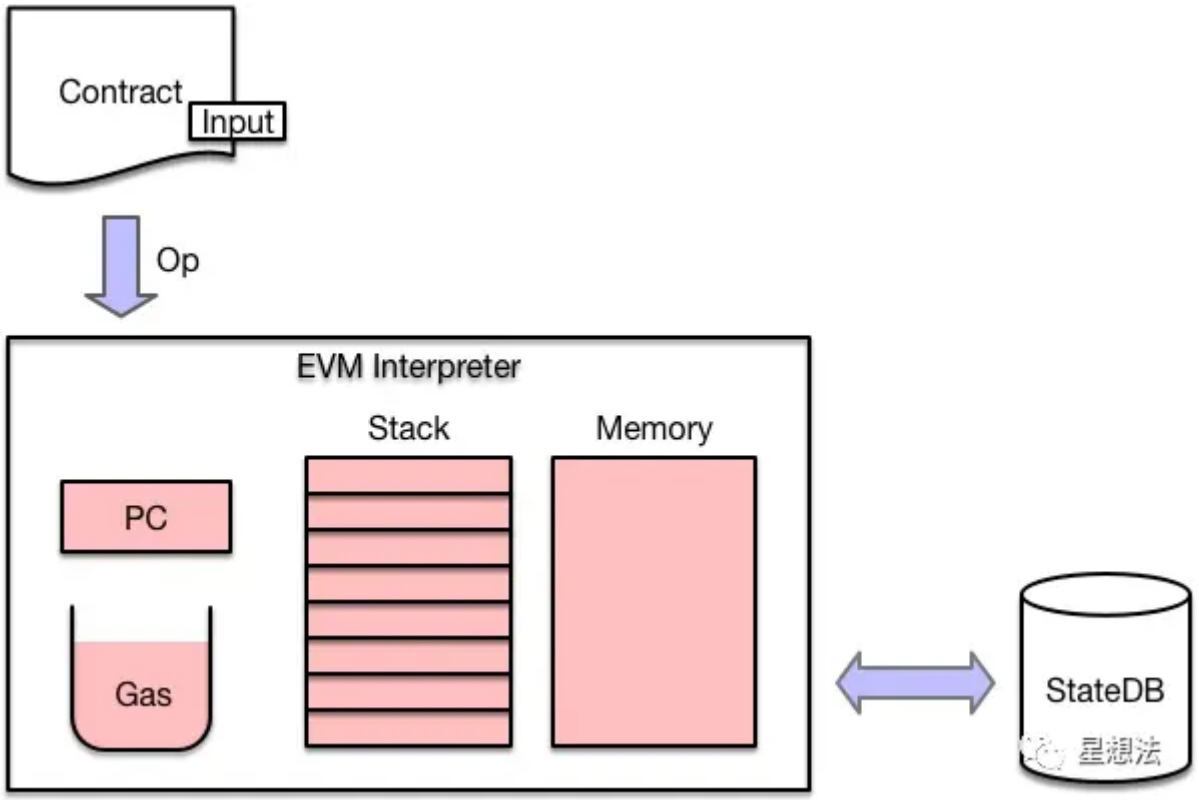
可以看到，Contract中会根据合约地址，从 `StateDB` 中加载对应的代码，后面就可以送入解释器执行了。

另外，执行合约能够消耗的油费有一个上限，就是节点配置的每个区块能够容纳的 `GasLimit` 。

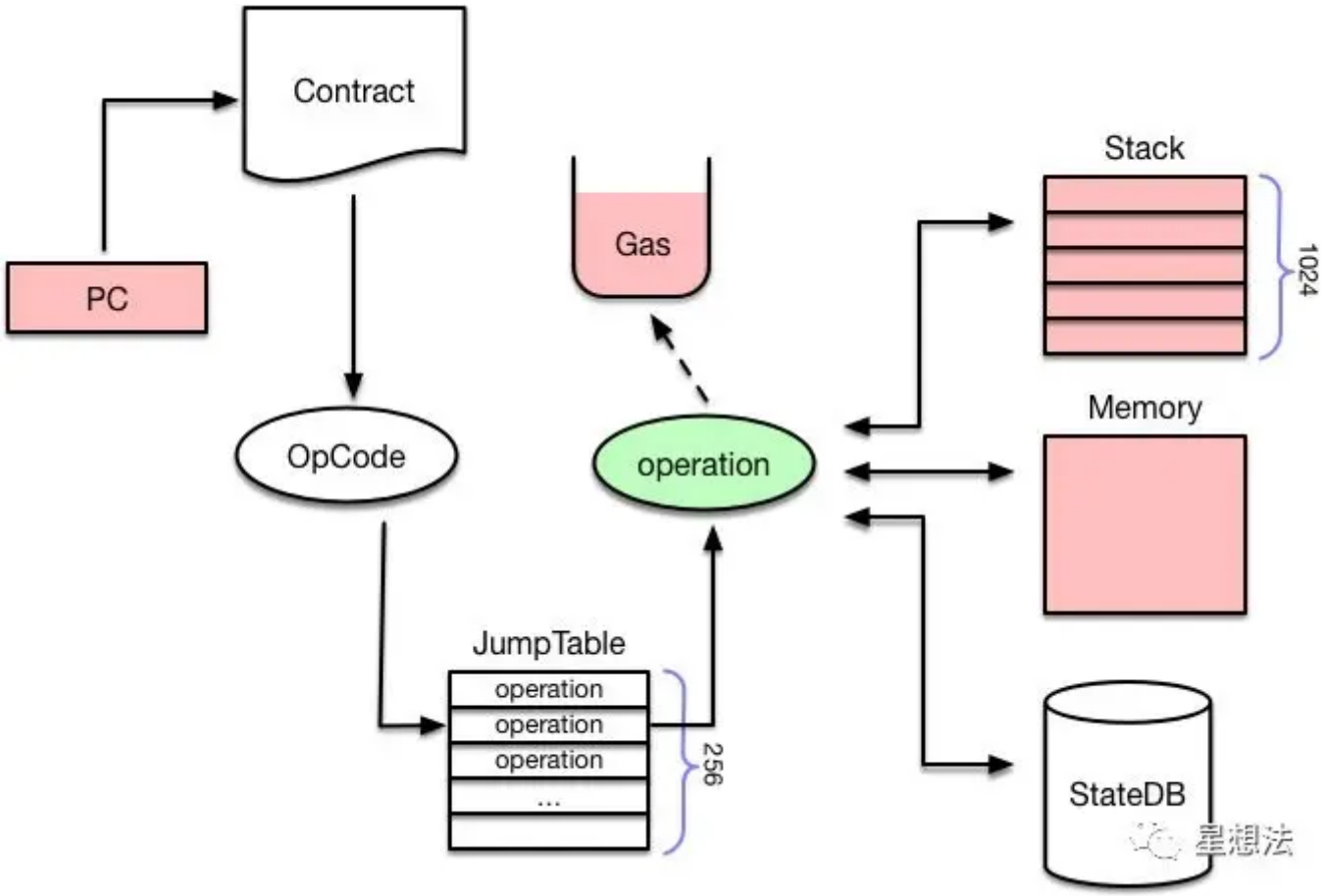
送入解释器执行

代码跟输入都有了，就可以送入解释器执行了。EVM是基于栈的虚拟机，解释器中需要操作四大组件：

- PC：类似于CPU中的PC寄存器，指向当前执行的指令
- Stack：执行堆栈，位宽为256 bits，最大深度为1024
- Memory：内存空间
- Gas：油费池，耗光邮费则交易执行失败



具体**解释执行的流程**参见下图:

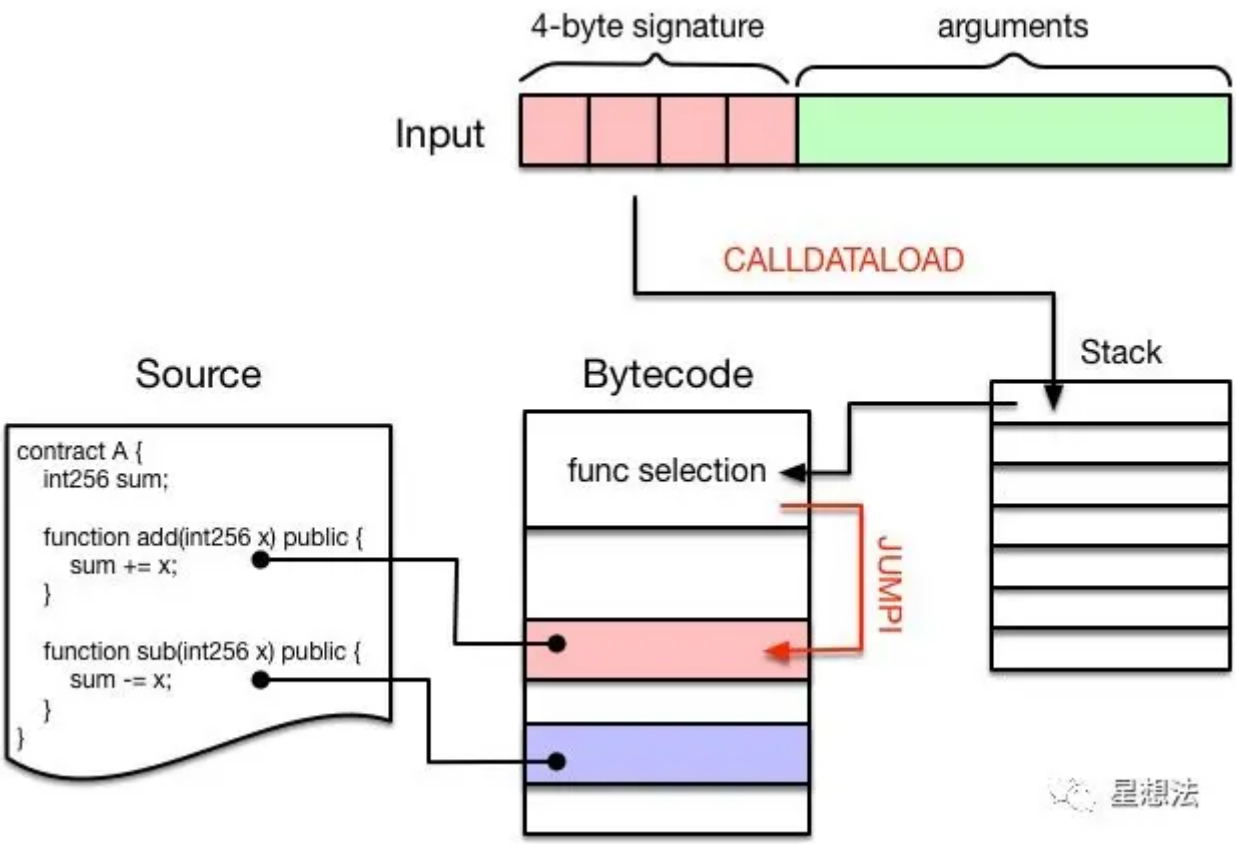


EVM的每条指令称为一个OpCode，占用一个字节，所以指令集最多不超过256，具体描述参见：https://ethervm.io (https://ethervm.io) 。比如下图就是一个示例（PUSH1=0x60, MSTORE=0x52）：

首先PC会从合约代码中读取一个OpCode，然后从一个JumpTable中检索出对应的operation，也就是与其相关联的函数集合。接下来会计算该操作需要消耗的油费，如果油费耗光则执行失败，返回ErrOutOfGas错误。如果油费充足，则调用execute()执行该指令，根据指令类型的不同，会分别对Stack、Memory或者StateDB进行读写操作。

调用合约函数

前面分析完了EVM解释执行的主要流程，可能有些同学会问：那么EVM怎么知道交易想调用的是合约里的哪个函数呢？别急，前面提到跟合约代码一起送到解释器里的还有一个Input，而这个Input数据是由交易提供的。



Input数据通常分为两个部分：

- 前面4个字节被称为“4-byte signature”，是某个函数签名的Keccak哈希值的前4个字节，作为该函数的唯一标识。（可以在该网站查询目前所有的函数签名(https://www.4byte.directory)）
- 后面跟的就是调用该函数需要提供的参数了，长度不定。

举个例子：我在部署完A合约后，调用add(1)对应的Input数据是

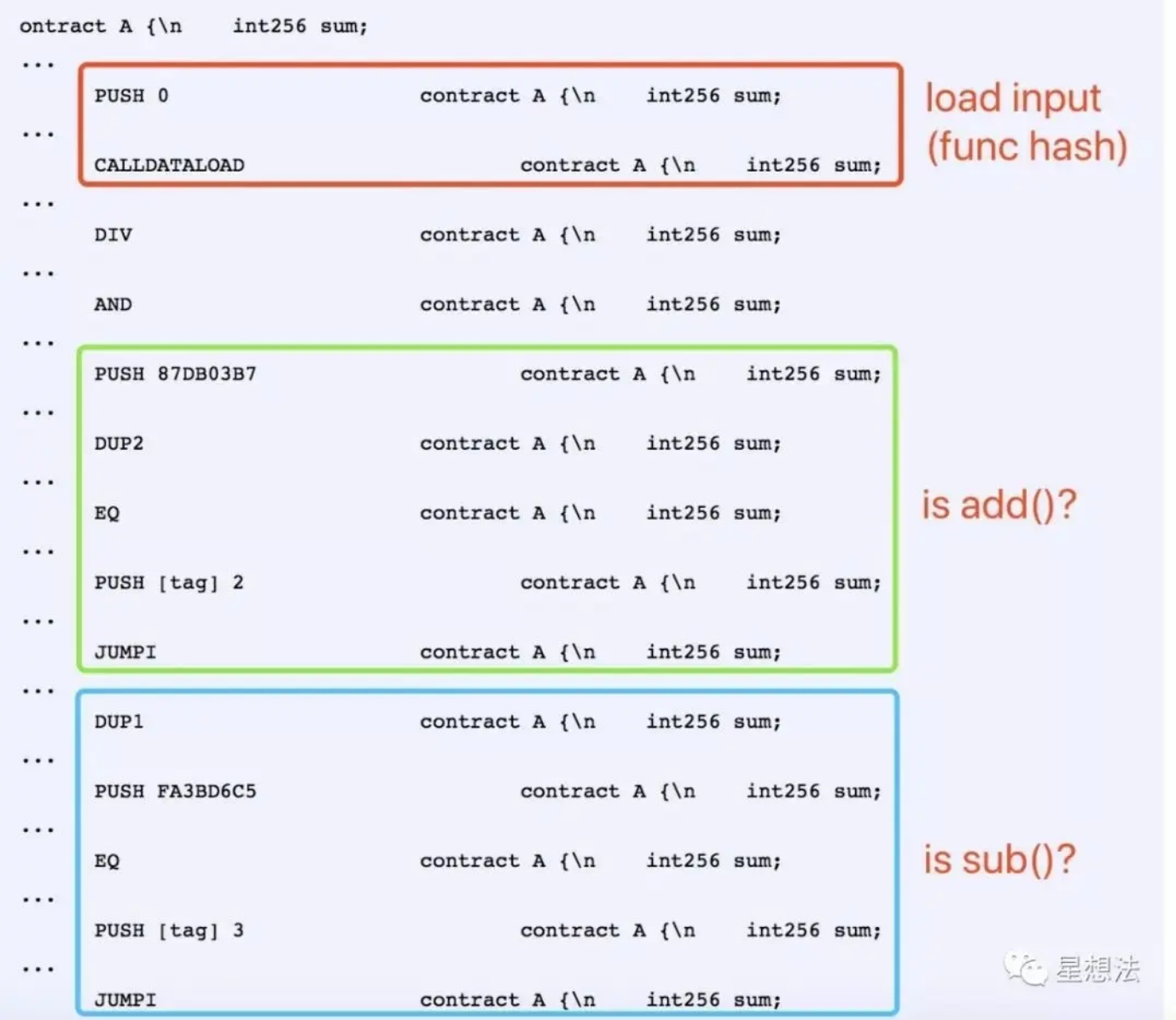
```
0x87db03b70000000000000000000000000000000000000000000000000000000000000001
```

而在我们编译智能合约的时候，编译器会自动在生成的字节码的最前面增加一段函数选择逻辑：

首先通过 `CALLDATALOAD` 指令将“4-byte signature”压入堆栈中，然后依次跟该合约中包含的函数进行比对，如果匹配则调用JUMPI指令跳入该段代码继续执行。

这么讲可能有点抽象，我们可以看一看上图中的合约对应的反汇编代码就一目了然了：

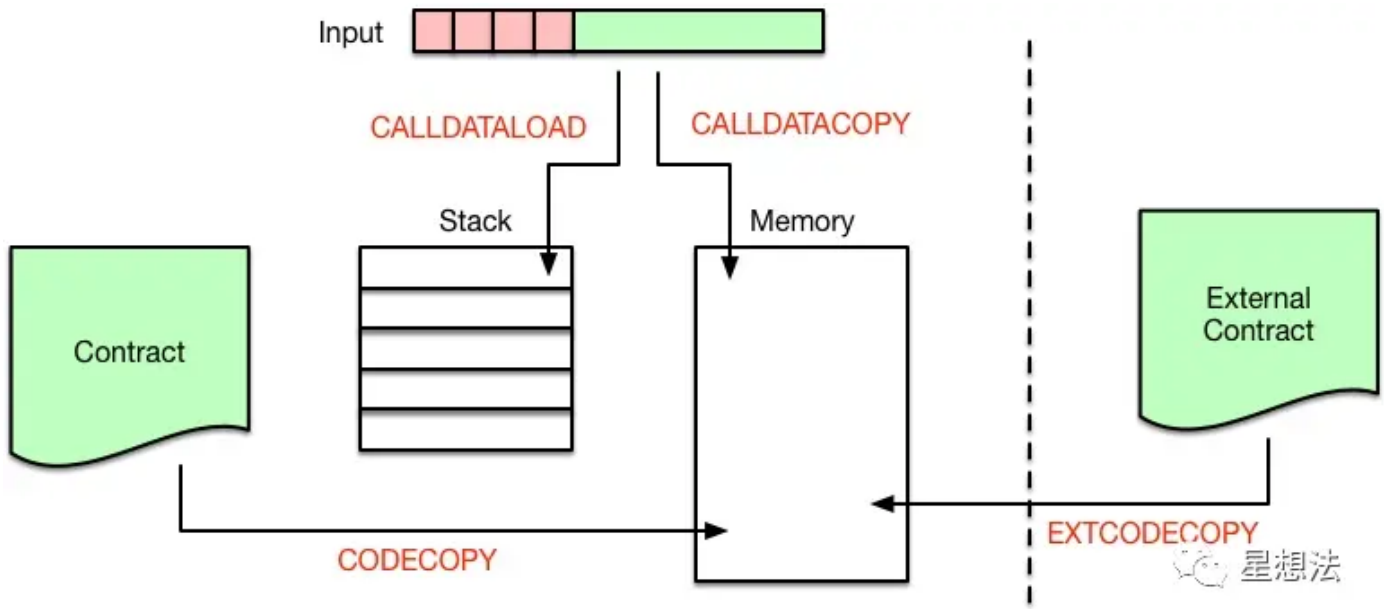
```
FUNCTIONHASHES

{
    "87db03b7": "add(int256)",
    "fa3bd6c5": "sub(int256)"
}
```



这里提到了 `CALLDATALOAD`，就顺便讲一下数据加载相关的指令，一共有4种：

- CALLDATALOAD：把输入数据加载到Stack中
- CALLDATACOPY：把输入数据加载到Memory中
- CODECOPY：把当前合约代码拷贝到Memory中
- EXTCODECOPY：把外部合约代码拷贝到Memory中

最后一个EXTCODECOPY不太常用，一般是为了审计第三方合约的字节码是否符合规范，消耗的gas一般也比较多。这些指令对应的操作如下图所示：

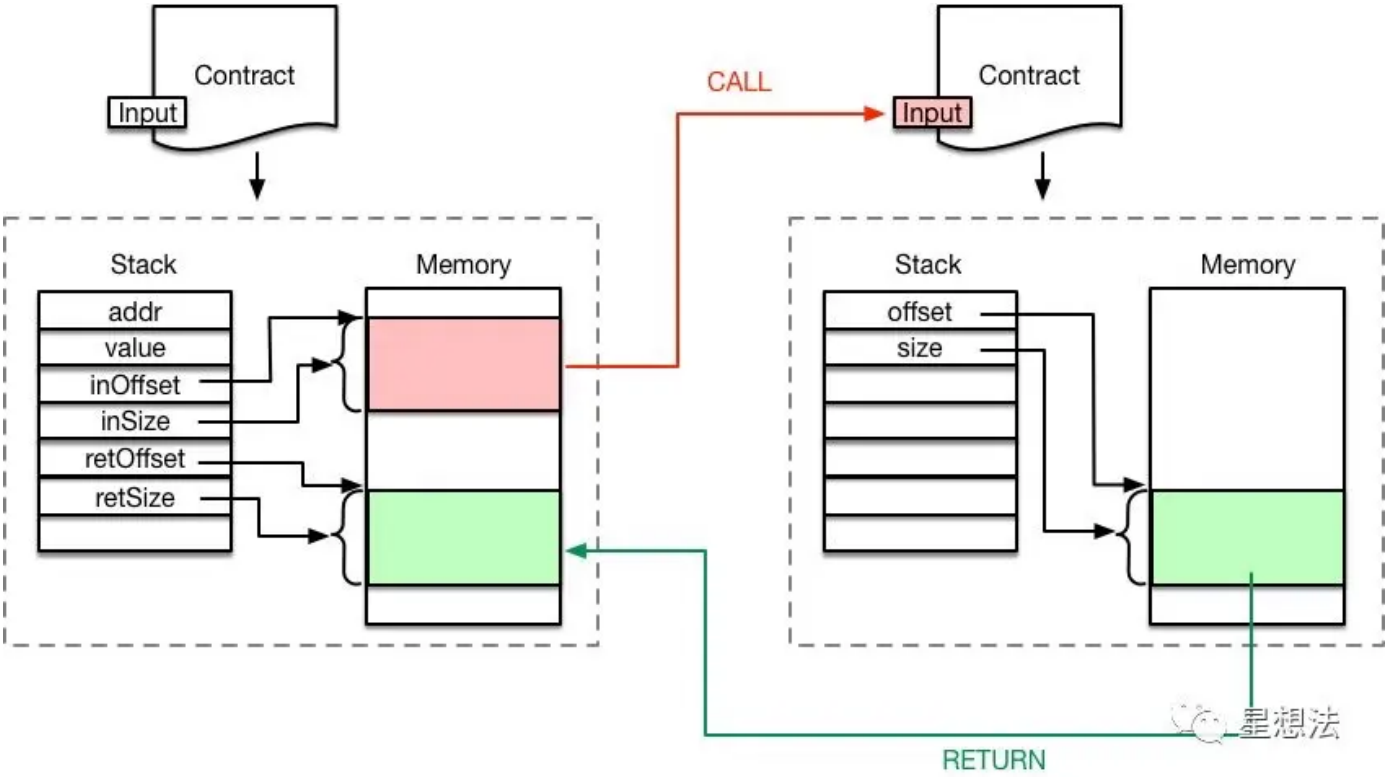


合约调用合约

合约内部调用另外一个合约，有4种调用方式：

- CALL
- CALLCODE
- DELEGATECALL
- STATICALL

后面会专门写篇文章比较它们的异同，这里先以最简单的CALL为例，调用流程如下图所示：



可以看到，调用者把调用参数存储在内存中，然后执行CALL指令。

CALL指令执行时会创建新的Contract对象，并以内存中的调用参数作为其Input。

解释器会为新合约的执行创建新的 `Stack` 和 `Memory` ，从而不会破坏原合约的执行环境。

新合约执行完成后，通过RETURN指令把执行结果写入之前指定的内存地址，然后原合约继续向后执行。

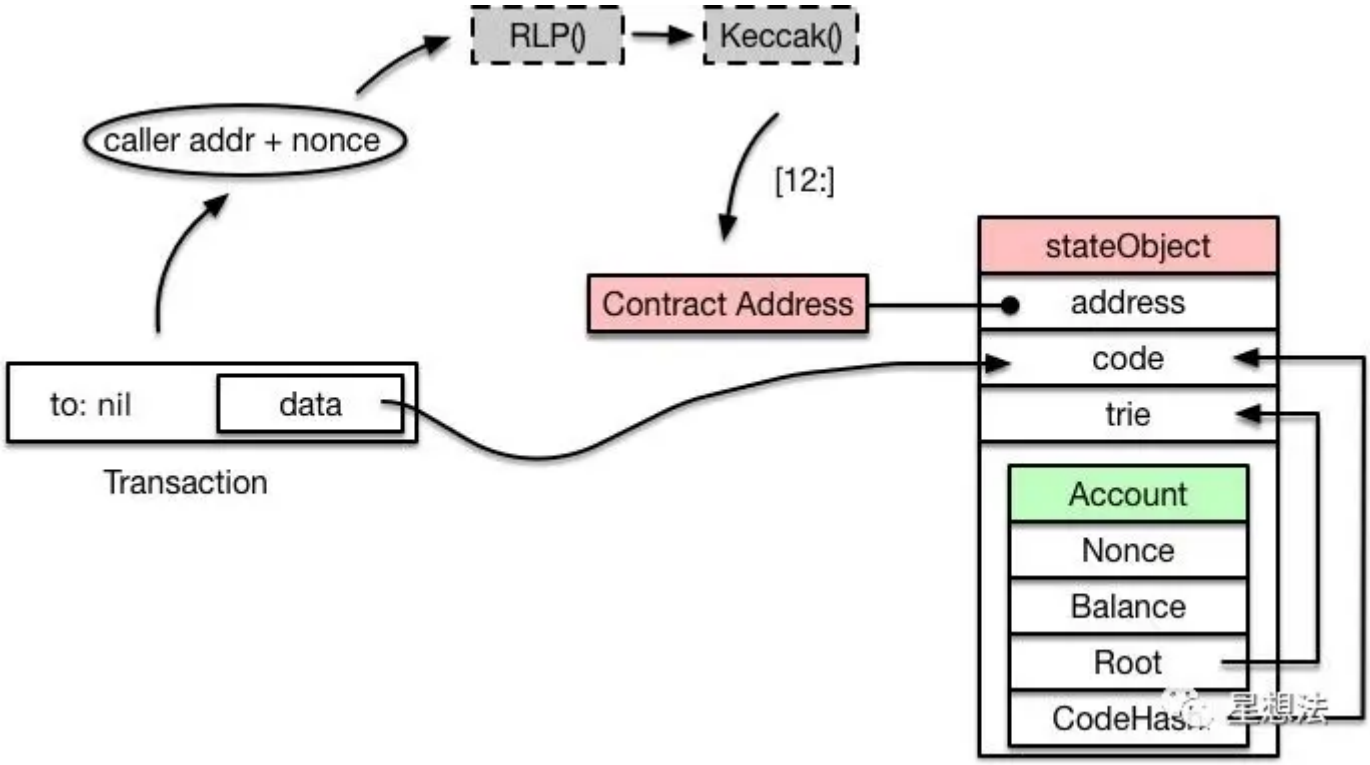
创建合约

前面都是讨论的合约调用，那么创建合约的流程时怎么样的呢？

如果某一笔交易的to地址为nil，则表明该交易是用于创建智能合约的。

首先需要创建合约地址，采用下面的计算公式： `Keccak(RLP(call_addr, nonce))[:12]` 。也就是说，对交易发起人的地址和nonce进行RLP编码，再算出Keccak哈希值，取后20个字节作为该合约的地址。

下一步就是根据合约地址创建对应的 `stateObject` ，然后存储交易中包含的合约代码。该合约的所有状态变化会存储在一个 `storage trie` 中，最终以 `Key-Value` 的形式存储到StateDB中。代码一经存储则无法改变，而 `storage trie` 中的内容则是可以通过调用合约进行修改的，比如通过SSTORE指令。


RLP()
Keccak()
caller addr + nonce
[12:]
stateObject
Contract Address
address
code
trie
to: nil
data
Transaction
Account
Nonce
Balance
Root
CodeHas
星想法

油费计算

最后啰嗦一下油费的计算，计算公式基本上是根据以太坊黄皮书 (http://gavwood.com/paper.pdf) 中的定义。

(220)

$$C(\boldsymbol{\sigma},\boldsymbol{\mu},I) \equiv C_{mem}(\boldsymbol{\mu}'_i) - C_{mem}(\boldsymbol{\mu}_i) + \begin{cases} C_{\text{SSTORE}}(\boldsymbol{\sigma},\boldsymbol{\mu}) & \text{if} \quad w = \text{SSTORE} \\ G_{exp} & \text{if} \quad w = \text{EXP} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] = 0 \\ G_{exp} + G_{expbyte} \times (1 + \lfloor \log_{256}(\boldsymbol{\mu}_{\mathbf{s}}[1]) \rfloor) & \text{if} \quad w = \text{EXP} \wedge \boldsymbol{\mu}_{\mathbf{s}}[1] > 0 \\ G_{verylow} + G_{copy} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[2] \div 32 \rceil & \text{if} \quad w = \text{CALLDATACOPY} \vee \text{CODECOPY} \\ G_{extcode} + G_{copy} \times \lceil \boldsymbol{\mu}_{\mathbf{s}}[3] \div 32 \rceil & \text{if} \quad w = \text{EXTCODECOPY} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] & \text{if} \quad w = \text{LOG0} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + G_{logtopic} & \text{if} \quad w = \text{LOG1} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 2G_{logtopic} & \text{if} \quad w = \text{LOG2} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 3G_{logtopic} & \text{if} \quad w = \text{LOG3} \\ G_{log} + G_{logdata} \times \boldsymbol{\mu}_{\mathbf{s}}[1] + 4G_{logtopic} & \text{if} \quad w = \text{LOG4} \\ C_{\text{CALL}}(\boldsymbol{\sigma},\boldsymbol{\mu}) & \text{if} \quad w = \text{CALL} \vee \text{CALLCODE} \vee \text{DELEGATECALL} \\ C_{\text{SUICIDE}}(\boldsymbol{\sigma},\boldsymbol{\mu}) & \text{if} \quad w = \text{SUICIDE} \\ G_{create} & \text{if} \quad w = \text{CREATE} \\ G_{sha3} + G_{sha3word} \lceil \mathbf{s}[1] \div 32 \rceil & \text{if} \quad w = \text{SHA3} \\ G_{jumpdest} & \text{if} \quad w = \text{JUMPDEST} \\ G_{sload} & \text{if} \quad w = \text{SLOAD} \\ G_{zero} & \text{if} \quad w \in W_{zero} \\ G_{base} & \text{if} \quad w \in W_{base} \\ G_{verylow} & \text{if} \quad w \in W_{verylow} \\ G_{low} & \text{if} \quad w \in W_{low} \\ G_{mid} & \text{if} \quad w \in W_{mid} \\ G_{high} & \text{if} \quad w \in W_{high} \\ G_{extcode} & \text{if} \quad w \in W_{extcode} \\ G_{balance} & \text{if} \quad w = \text{BALANCE} \\ G_{blockhash} & \text{if} \quad w = \text{BLOCKHASH} \end{cases}$$

(221)

$$w \equiv \begin{cases} I_{\mathbf{b}}[\boldsymbol{\mu}_{pc}] & \text{if} \quad \boldsymbol{\mu}_{pc} < \|I_{\mathbf{b}}\| \\ \text{STOP} & \text{otherwise} \end{cases}$$

where:

(222)

$$C_{mem}(a) \equiv G_{memory} \cdot a + \left\lfloor \frac{a^2}{512} \right\rfloor$$

with C_{CALL}, C_{SUICIDE} and C_{SSTORE} as specified in the appropriate section below. We define the following subsets of instructions:

W_{zero} = {STOP, RETURN}
W_{base} = {ADDRESS, ORIGIN, CALLER, CALLVALUE, CALLDATASIZE, CODESIZE, GASPRICE, COINBASE, TIMESTAMP, NUMBER, DIFFICULTY, GASLIMIT, POP, PC, MSIZE, GAS}
$W_{verylow}$ = {ADD, SUB, NOT, LT, GT, SLT, SGT, EQ, ISZERO, AND, OR, XOR, BYTE, CALLDATALOAD, MLOAD, MSTORE, MSTORE8, PUSH*, DUP*, SWAP*}
W_{low} = {MUL, DIV, SDIV, MOD, SMOD, SIGNEXTEND}
W_{mid} = {ADDMOD, MULMOD, JUMP}
W_{high} = {JUMPI}
$W_{extcode}$ = {EXTCODESIZE}

当然你可以直接read the fucking code，代码位于core/vm/gas.go和core/vm/gas_table.go中。

合约的四种调用方式

在中大型的项目中，我们不可能在一个智能合约中实现所有的功能，而且这样也不利于分工合作。一般情况下，我们会把代码按功能划分到不同的库或者合约中，然后提供接口互相调用。

在 `Solidity` 中，如果只是为了代码复用，我们会把公共代码抽出来，部署到一个library中，后面就可以像调用C库、Java库一样使用了。但是library中不允许定义任何storage类型的变量，这就意味着library不能修改合约的状态。如果需要修改合约状态，我们需要部署一个新的合约，这就涉及到合约调用合约的情况。

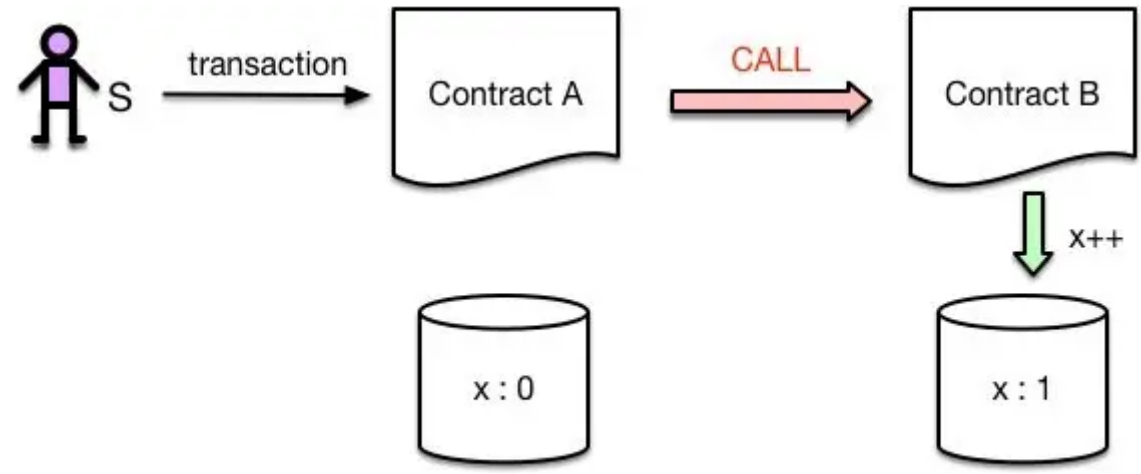
合约调用合约有下面4种方式：

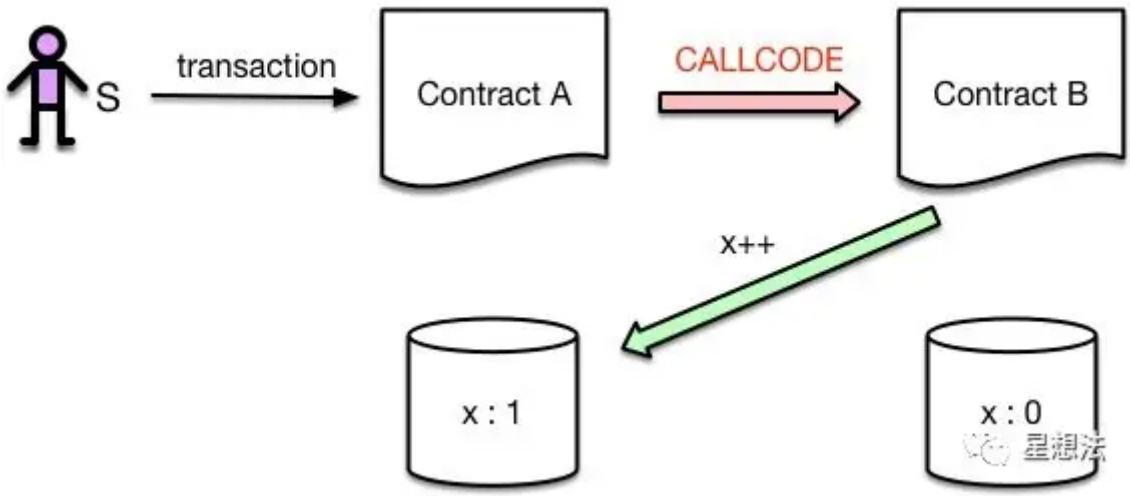
- CALL
- CALLCODE
- DELEGATECALL
- STATICCALL

CALL vs. CALLCODE

CALL和CALLCODE的区别在于：代码执行的上下文环境不同。

具体来说，CALL修改的是**被调用者**的storage，而CALLCODE修改的是**调用者**的storage。

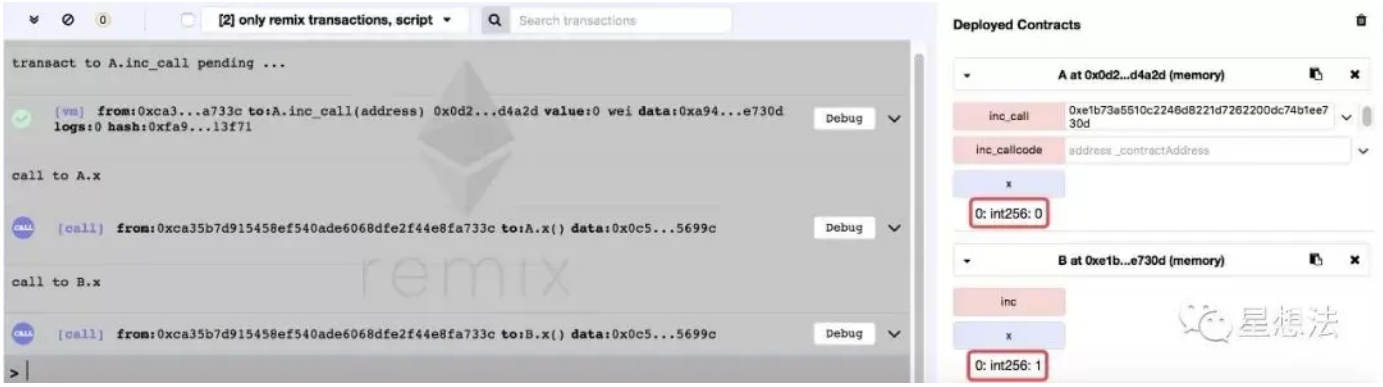




我们写个合约验证一下我们的理解：

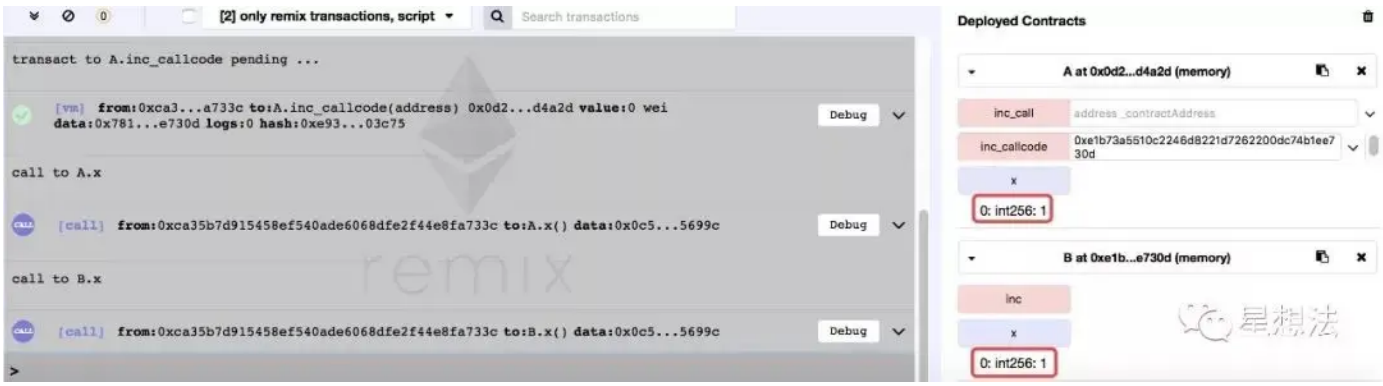
```
pragma solidity ^0.4.25;

contract A {
  int public x;

  function inc_call(address _contractAddress) public {
      _contractAddress.call(bytes4(keccak256("inc()")));
  }
  function inc_callcode(address _contractAddress) public {
      _contractAddress.callcode(bytes4(keccak256("inc()")));
  }
}

contract B {
  int public x;

  function inc() public {
      x++;
  }
}
```

我们先调用一下 `inc_call()` ，然后查询合约A和B中x的值有什么变化：



可以发现，合约B中的x被修改了，而合约A中的x还等于0。

我们再调用一下 `inc_callcode()` 试试：



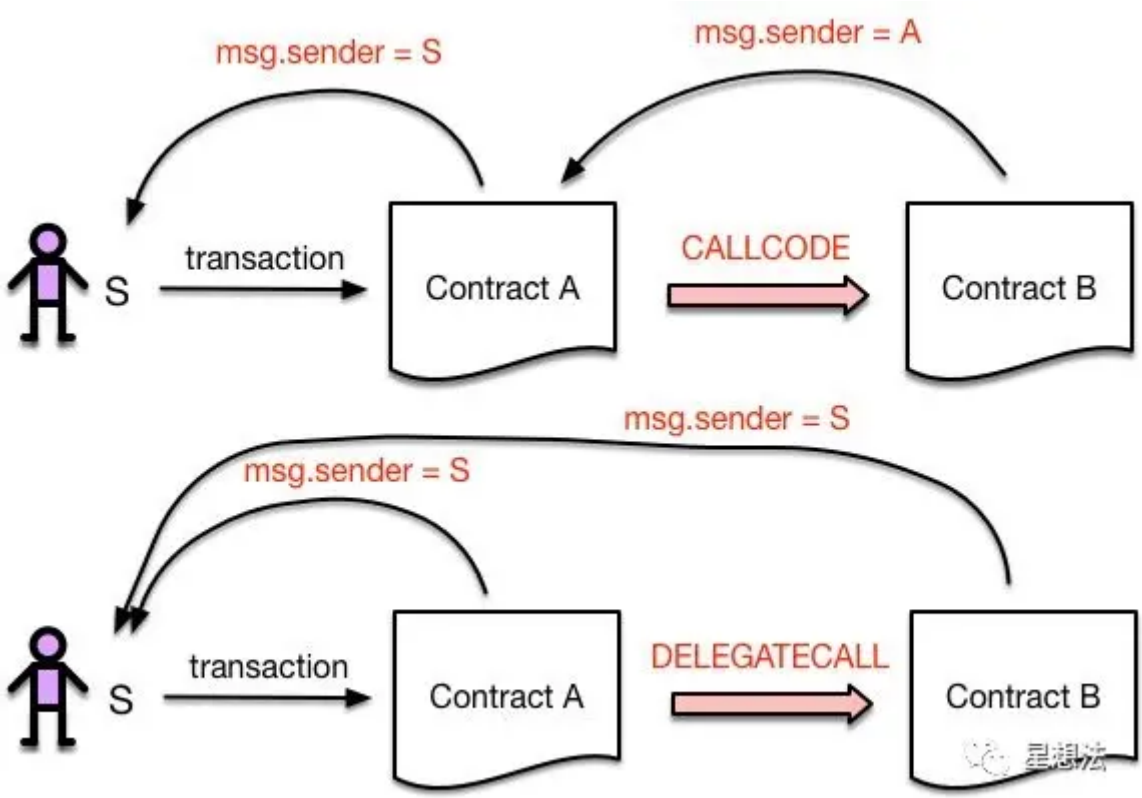
可以发现，这次修改的是合约A中x，合约B中的x保持不变。

CALLCODE vs. DELEGATECALL

实际上，可以认为DELEGATECALL是CALLCODE的一个bugfix版本，官方已经不建议使用CALLCODE了。

CALLCODE和DELEGATECALL的区别在于： `msg.sender` 不同。

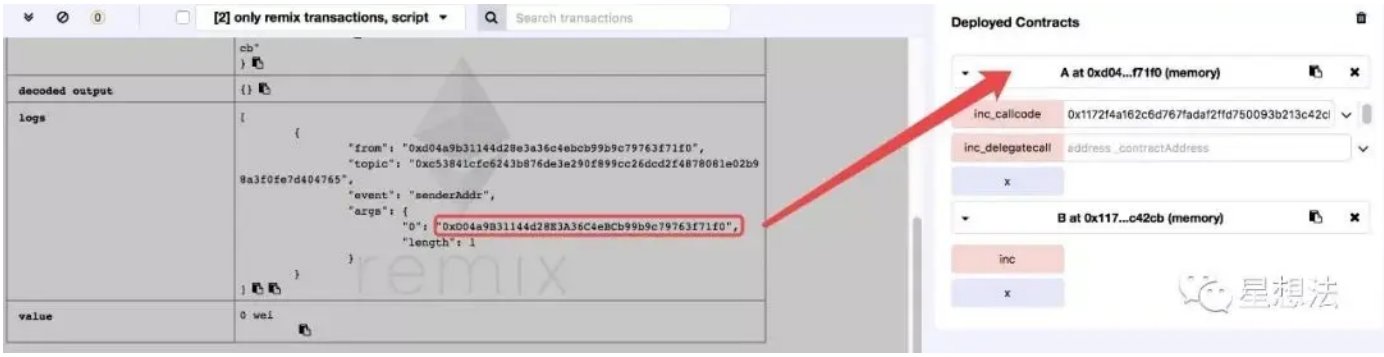
具体来说，DELEGATECALL会一直使用原始调用者的地址，而CALLCODE不会。



我们还是写一段代码来验证我们的理解：

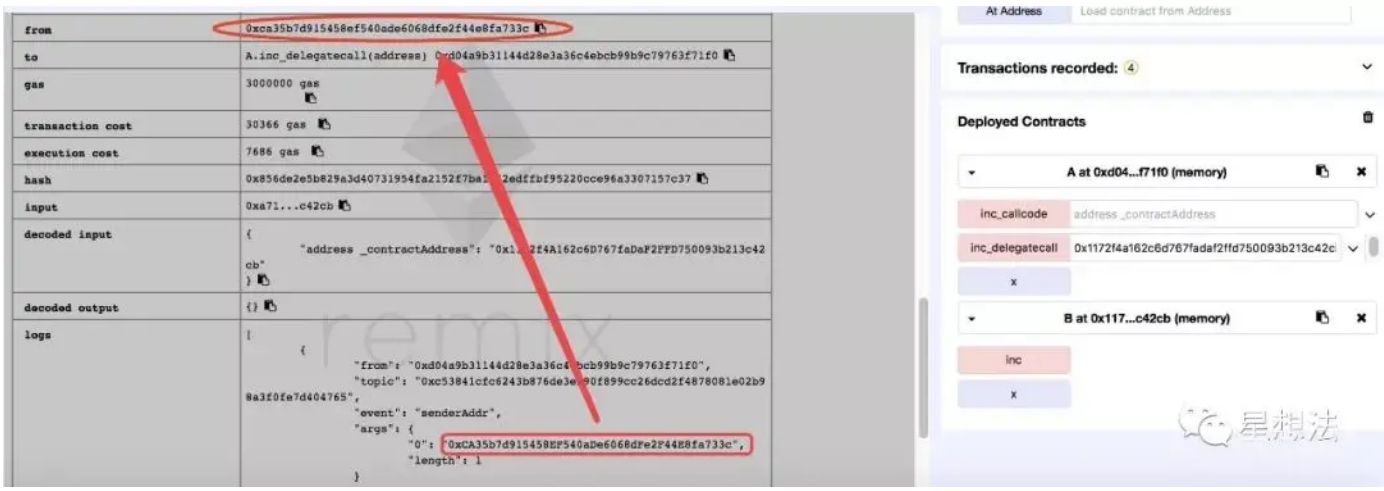
```
pragma solidity ^0.4.25;

contract A {
  int public x;

  function inc_callcode(address _contractAddress) public {
      _contractAddress.callcode(bytes4(keccak256("inc()")));
  }
  function inc_delegatecall(address _contractAddress) public {
      _contractAddress.delegatecall(bytes4(keccak256("inc()")));
  }
}

contract B {
  int public x;

  event senderAddr(address);
  function inc() public {
      x++;
      emit senderAddr(msg.sender);
  }
}
```

我们首先调用一下inc_callcode()，观察一下log输出：



可以发现，msg.sender指向合约A的地址，而非交易发起者的地址。

我们再调用一下inc_delegatecall()，观察一下log输出：



可以发现，msg.sender指向的是交易的发起者。

STATICCALL

STATICCALL放在这里似乎有滥竽充数之嫌，因为目前Solidity中并没有一个low level API可以直接调用它，仅仅是计划将来在编译器层面把调用view和pure类型的函数编译成STATICCALL指令。

view类型的函数表明其不能修改状态变量，而pure类型的函数则更加严格，连读取状态变量都不允许。

目前是在编译阶段来检查这一点的，如果不符合规定则会出现编译错误。如果将来换成STATICCALL指令，就可以完全在运行时阶段来保证这一点了，你可能会看到一个执行失败的交易。

话不多说，我们就先看看STATICCALL的实现代码吧：

```go
func (in *Interpreter) enforceRestrictions(op OpCode, operation operation, stack *Stack)
    if in.evm.chainRules.IsByzantium {
        if in.readOnly {
            // If the interpreter is operating in readonly mode, make sure no
            // state-modifying operation is performed. The 3rd stack item
            // for a call operation is the value. Transferring value from one
            // account to the others means the state is modified and should also
            // return with an error.
            if operation.writes || (op == CALL && stack.Back(2).BitLen() > 0) {
                return errWriteProtection
            }
        }
    }
    return nil
}
```


可以看到，解释器增加了一个readOnly属性，STATICCALL会把该属性置为true，如果出现状态变量的写操作，则会返回一个errWriteProtection错误。

总结：以太坊 (https://learnblockchain.cn/2017/11/20/whatiseth/)虚拟机用来执行以太坊上的交易，更改以太坊状态。交易分两种：普通交易和智能合约交易。在执行交易时需要支付油费。智能合约之间的调用有四种方式。

作者Star Li，公众号星想法 (https://mp.weixin.qq.com/mp/profile_ext?action=home&__biz=MzU5MzMxNTk2Nw==&scene=124#wechat_redirect)有很多原创高质量文章，欢迎大家关注。

深入浅出区块链 (https://learnblockchain.cn/) - 系统学习区块链，学区块链都在这里，打造最好的区块链技术博客。

你可能感兴趣的文章

相关问题

evm相关问题 (https://learnblockchain.cn/question/2886) 1 回答

2 条评论

(https://learnblockchain.cn/people/3276)

bxex (https://learnblockchain.cn/people/3276)

固定油费章节里，我的理解有些不同，想请教一下。我认为21000的gas是初始gas或者是最高gas，然后根据交易字节长度计算实际花费，从21000里面扣除实际花费的gas给矿工，剩余gas退还给发起交易者。而不是文章里所说的21000再加上实际gas消耗。

2021-05-22 16:05 0

(https://learnblockchain.cn/people/3206)

尾张大 (https://learnblockchain.cn/people/3206)

大佬您好，我在测试的转账交易，非0字节收费是16，我在ganache上测试的，和您的68有些不一样，可以解释一下吗

2021-12-31 11:36 0

请先 登录 (https://learnblockchain.cn/login) 后评论

(https://learnblockchain.cn/people/28)

Star Li (https://learnblockchain.cn/people/28)

70 篇文章, 4732 学分