

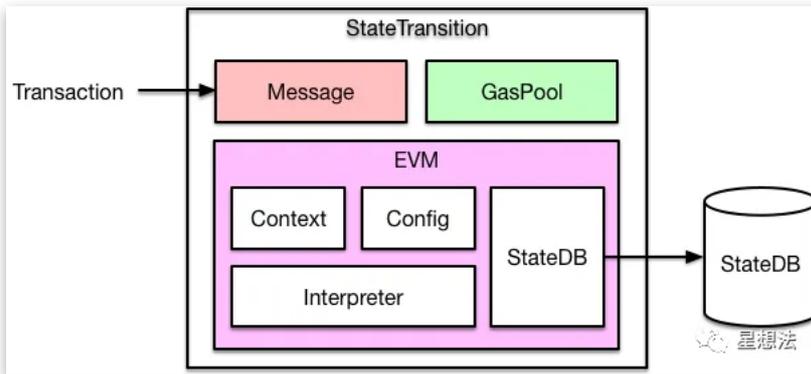
以太坊EVM源码分析之执行流程

2020-05-05

以太坊EVM源码分析之执行流程

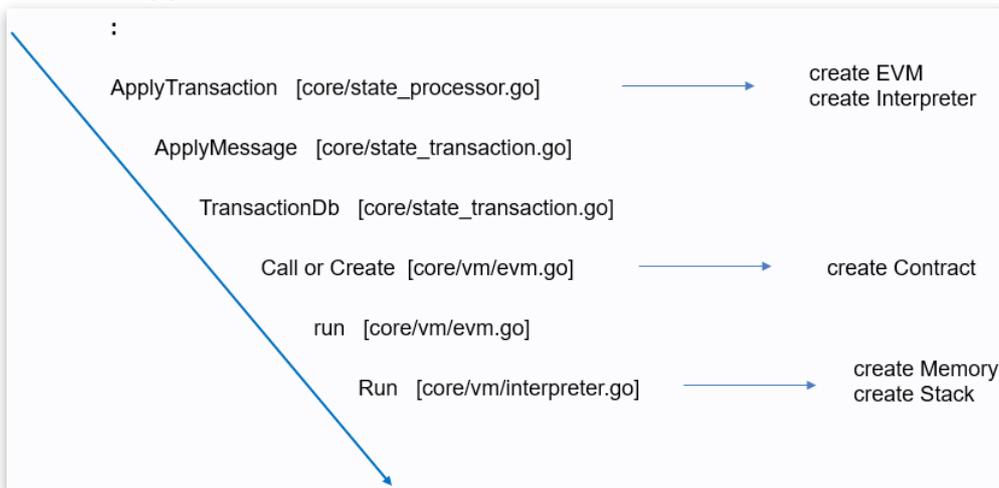
业务流程概述

EVM是用来执行智能合约的。输入一笔交易，内部会将其转换成一个 **Message** 对象，传入 EVM 执行。在合约中，msg 全局变量记录了附带当前合约的交易的信息，可能是为了一致，这里也将 **transaction** 转换成 **Message** 传给 EVM 对象。



如果是普通转账交易，执行时完全不需要EVM的操作(EVM进行的是空操作)，直接修改 StateDB 中对应的账户余额即可。

如果是智能合约的创建或者调用，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。[6]



接下来我们按照这个顺序分析源码。

Create EVM

core/state_processor.go

```

1 // ApplyTransaction attempts to apply a transaction to the given state database
2 // and uses the input parameters for its environment. It returns the receipt
3 // for the transaction, gas used and an error if the transaction failed,
4 // indicating the block was invalid.
5 // ApplyTransaction尝试将事务应用于给定的状态数据库，并为其环境使用输入参数。
6 // 它返回事务的receipt、使用的gas，如果事务失败，返回一个错误指示块无效。
7 // 将交易的信息记录到以太坊状态数据库 (state.StateDB) 中，这其中包括转账信息和执行合约信息
8 func ApplyTransaction(config *params.ChainConfig, bc ChainContext, author *common.Address, g
  
```

码农家园

```

12     return nil, err
13 }
14 // Create a new context to be used in the EVM environment
15 // 创建一个要在EVM环境中使用的上下文
16 context := NewEVMContext(msg, header, bc, author)
17 // Create a new environment which holds all relevant information
18 // about the transaction and calling mechanisms.
19 // 创建一个新环境，其中包含关于事务和调用机制的所有相关信息。
20 vmenv := vm.NewEVM(context, statedb, config, cfg)
21 // Apply the transaction to the current state (included in the env)
22 // 将事务应用于当前状态
23 _, gas, failed, err := ApplyMessage(vmenv, msg, gp)
24 if err != nil {
25     return nil, err
26 }
27 // Update the state with pending changes
28 // 使用挂起的更改更新状态
29 var root []byte
30 // 根据版本采用不同的状态更新方法
31 if config.IsByzantium(header.Number) {
32     statedb.Finalise(true)
33 } else {
34     root = statedb.IntermediateRoot(config.IsEIP158(header.Number)).Bytes()
35 }
36 *usedGas += gas
37
38 // Create a new receipt for the transaction, storing the intermediate root and gas used by the tx
39 // based on the eip phase, we're passing whether the root accounts.
40 // 为交易创建一个新的receipt，存储中间根和基于eip阶段交易使用的gas，我们正在传递是否根到
41 receipt := types.NewReceipt(root, failed, *usedGas)
42 receipt.TxHash = tx.Hash()
43 receipt.GasUsed = gas
44 // if the transaction created a contract, store the creation address in the receipt.
45 // 如果事务创建了一个合约，则将创建地址存储在receipt中。
46 if msg.To() == nil {
47     receipt.ContractAddress = crypto.CreateAddress(vmenv.Context.Origin, tx.Nonce())
48 }
49 // Set the receipt logs and create a bloom for filtering
50 // 设置receipt日志并创建用于过滤的bloom
51 receipt.Logs = statedb.GetLogs(tx.Hash())
52 receipt.Bloom = types.CreateBloom(types.Receipts{receipt})
53 receipt.BlockHash = statedb.BlockHash()
54 receipt.BlockNumber = header.Number
55 receipt.TransactionIndex = uint(statedb.TxIndex())
56
57 return receipt, err
58 }

```

这个函数首先将 `transaction` 转换成了 `Message`，然后创建了一个 `Context`，接下来调用 `vm.NewEVM` 创建了新的EVM，通过 `ApplyMessage` 执行相关功能。也就是说每处理一笔交易，就要创建一个 EVM 来执行交易中的数据。执行完成后，该函数更新状态、创建 `receipt` 以及进行日志记录。`ApplyMessage` 只有一行代码，调用了 `StateTransition.TransitionDb` 函数。

状态转换模型

`core/state_tansaction.go`

```

1 /*
2 The State Transitioning Model 状态转换模型
3

```

码农家园

```

7
8 1) Nonce handling 处理Nonce
9 2) Pre pay gas 提前支付gas
10 3) Create a new state object if the recipient is \0*32 如果接收方是\0*32, 则创建一个新的stateObjc
11 4) Value transfer 价值转移
12 == If contract creation 如果是合约创建 ==
13 4a) Attempt to run transaction data 尝试运行事务数据
14 4b) If valid, use result as code for the new state object 如果有效, 则使用result作为新stateObject的
15 == end ==
16 5) Run Script section 运行脚本部分
17 6) Derive new state root 导出新状态根
18 */
19 // 记录了在处理一笔交易过程中的状态数据, 比如 gas 的花费等
20 type StateTransition struct {
21     gp    *GasPool
22     msg    Message
23     gas    uint64 // 此交易过程当前剩余的gas
24     gasPrice *big.Int
25     initialGas uint64 // 此交易初始的gas, 即消息发送者指定用于此交易的gas量
26     value    *big.Int
27     data    []byte
28     state    vm.StateDB
29     evm     *vm.EVM
30 }
31
32 // TransitionDb will transition the state by applying the current message and
33 // returning the result including the used gas. It returns an error if failed.
34 // An error indicates a consensus issue.
35 // TransitionDb将通过应用当前消息来转换状态并返回结果(包括使用的gas)。如果失败, 它将返回一
36 func (st *StateTransition) TransitionDb() (ret []byte, usedGas uint64, failed bool, err error) {
37     if err = st.preCheck(); err != nil {
38         return
39     }
40     msg := st.msg
41     sender := vm.AccountRef(msg.From())
42     homestead := st.evm.ChainConfig().IsHomestead(st.evm.BlockNumber)
43     istanbul := st.evm.ChainConfig().IsIstanbul(st.evm.BlockNumber)
44     // 判断是否是创建新合约
45     contractCreation := msg.To() == nil
46
47     // Pay intrinsic gas
48     // 支付固定gas
49     gas, err := IntrinsicGas(st.data, contractCreation, homestead, istanbul)
50     if err != nil {
51         return nil, 0, false, err
52     }
53     if err = st.useGas(gas); err != nil {
54         return nil, 0, false, err
55     }
56
57     var (
58         evm = st.evm
59         // vm errors do not effect consensus and are therefor
60         // not assigned to err, except for insufficient balance
61         // error.
62         // vm错误不会影响共识, 因此不会被分配为err, 除非余额不足。
63         vmerr error
64     )
65     if contractCreation {

```

码农家园

```

69 // Increment the nonce for the next transaction
70 // 为下一个事务增加nonce
71 st.state.SetNonce(msg.From(), st.state.GetNonce(sender.Address()+1)
72 // 不是创建合约, 则调用 evm.Call调用合约
73 ret, st.gas, vmerr = evm.Call(sender, st.to(), st.data, st.gas, st.value)
74 }
75 if vmerr != nil {
76     log.Debug("VM returned with error", "err", vmerr)
77     // The only possible consensus-error would be if there wasn't
78     // sufficient balance to make the transfer happen. The first
79     // balance transfer may never fail.
80     // 唯一可能的共识错误是, 如果没有足够的余额进行交易。
81     if vmerr == vm.ErrInsufficientBalance {
82         return nil, 0, false, vmerr
83     }
84 }
85 // 返还gas, 并将已消耗的 gas 计入矿工账户中
86 st.refundGas()
87 st.state.AddBalance(st.evm.Coinbase, new(big.Int).Mul(new(big.Int).SetUint64(st.gasUsed()), st.
88
89 return ret, st.gasUsed(), vmerr != nil, err
90 }

```

`StateTransition.TransitionDb()` 方法首先调用 `StateTransaction.preCheck` 验证交易的 `Nonce` 值, 并从交易的发送者账户余额扣除 `gasLimit*gasPrice`, 用来「购买」交易执行需要的 `gas`。具体可参考`gas`的详细介绍。

然后先将交易的固有成本扣除。发送一笔交易的`gas`包含两部分: 固有成本和执行成本。

执行成本根据该交易需要使用多少EVM的资源来运算而定, 执行一笔交易所需的操作越多, 则它的执行成本就越高。

固有成本(`intrinsic gas`)由交易的基础成本(`base fee`)和负载(`payload`)决定, 每个零字节4 `gas`, 非零字节68 `gas`。交易负载分为以下三种负载:

- 若是创建智能合约, 则负载就是创建智能合约的 EVM 代码
- 若是调用智能合约的函数, 则负载就是执行消息的输入数据
- 若只是单纯在两个账户间转账, 则负载为空

接下来判断当前的交易是否是创建合约, 根据交易的接收者是否为空来判断。如果需要创建合约, 则调用 `EVM.Create` 进行创建; 如果不是, 则调用 `EVM.Call` 执行合约代码。(如果是转账交易, 接收者肯定不为空, 那么就会调用 `EVM.Call` 实现转账功能)[5]。

EVM 对象执行完相关功能后, 调用 `StateTransaction.refundGas` 将未用完的 `gas` 还给交易的发送者。然后将消耗的 `gas` 计入矿工账户中。

创建合约

`/core/vm/evm.go`

```

1 // create creates a new contract using code as deployment code.
2 // create使用code作为部署代码创建一个新合约
3 func (evm *EVM) create(caller ContractRef, codeAndHash *codeAndHash, gas uint64, value *big.
4 // Depth check execution. Fail if we're trying to execute above the
5 // limit.
6 // 检查合约创建的递归调用次数。若超过深度限制(1024)执行代码, 则失败。
7 if evm.depth > int(params.CallCreateDepth) {
8     return nil, common.Address{}, gas, ErrDepth
9 }
10 // 检查余额
11 if !evm.CanTransfer(evm.StateDB, caller.Address(), value) {
12     return nil, common.Address{}, gas, ErrInsufficientBalance

```

码农家园

```

16  evm.StateDB.SetNonce(caller.Address(), nonce+1)
17
18  // Ensure there's no existing contract already at the designated address
19  // 确保特定的地址没有合约存在
20  contractHash := evm.StateDB.GetCodeHash(address)
21  if evm.StateDB.GetNonce(address) != 0 || (contractHash != (common.Hash{})) && contractHash
22      return nil, common.Address{}, 0, ErrContractAddressCollision
23  }
24  // Create a new account on the state
25  // 在现有状态上创建一个新合约
26  snapshot := evm.StateDB.Snapshot() // 创建一个StateDB的快照，以便回滚
27  evm.StateDB.CreateAccount(address) // 创建账户
28  if evm.chainRules.IsEIP158 {
29      evm.StateDB.SetNonce(address, 1) // 设置nonce，从1开始
30  }
31  // 执行交易
32  evm.Transfer(evm.StateDB, caller.Address(), address, value)
33
34  // Initialise a new contract and set the code that is to be used by the EVM.
35  // The contract is a scoped environment for this execution context only.
36  contract := NewContract(caller, AccountRef(address), value, gas)
37  contract.SetCodeOptionalHash(&address, codeAndHash)
38
39  // NoRecursion禁用call, callcode, delegate call 和 create。
40  // 以太坊虚拟机被配置成不可递归创建合约，而当前创建合约的过程正是在递归过程中
41  if evm.vmConfig.NoRecursion && evm.depth > 0 {
42      return nil, address, gas, nil
43  }
44
45  // Debug模式，捕获跟踪程序启动事件
46  if evm.vmConfig.Debug && evm.depth == 0 {
47      evm.vmConfig.Tracer.CaptureStart(caller.Address(), address, true, codeAndHash.code, gas,
48  }
49  start := time.Now() // 当前时间
50
51  // ret所存的就是新合约的代码
52  ret, err := run(evm, contract, nil, false)
53
54  // check whether the max code size has been exceeded
55  // 判断返回的合约代码是否超过大小限制
56  maxCodeSizeExceeded := evm.chainRules.IsEIP158 && len(ret) > params.MaxCodeSize
57  // if the contract creation ran successfully and no errors were returned
58  // calculate the gas required to store the code. If the code could not
59  // be stored due to not enough gas set an error and let it be handled
60  // by the error checking condition below.
61  // 如果合约创建操作执行成功并且没有返回错误。计算存储代码所需gas。
62  // 如果因为gas不足而不能存储代码，设置一个错误，然后让下面的错误检查条件来处理它。
63  if err == nil && !maxCodeSizeExceeded {
64      // 合约创建成功，将合约保存到 StateDB 之前，先 useGas
65      createDataGas := uint64(len(ret)) * params.CreateDataGas // 计算gas
66      if contract.UseGas(createDataGas) { // gas充足，
67          evm.StateDB.SetCode(address, ret) // 往StateDB中存代码
68      } else {
69          err = ErrCodeStoreOutOfGas // 否则设置错误
70      }
71  }
72
73  // When an error was returned by the EVM or when setting the creation code
74  // above we revert to the snapshot and consume any gas remaining. Additionally

```

码农家园

```

78     evm.StateDB.RevertToSnapshot(snapshot)
79     if err != errExecutionReverted {
80         contract.UseGas(contract.Gas)
81     }
82 }
83 // Assign err if contract code size exceeds the max while the err is still empty.
84 // 如果合约代码大小超过限制并且err仍未空, 设置err = errMaxCodeSizeExceeded
85 if maxCodeSizeExceeded && err == nil {
86     err = errMaxCodeSizeExceeded
87 }
88 // Debug模式, 捕获跟踪程序结束事件
89 if evm.vmConfig.Debug && evm.depth == 0 {
90     evm.vmConfig.Tracer.CaptureEnd(ret, gas-contract.Gas, time.Since(start), err)
91 }
92 // 返回结果
93 return ret, address, contract.Gas, err
94
95 }
96
97 // Create creates a new contract using code as deployment code.
98 func (evm *EVM) Create(caller ContractRef, code []byte, gas uint64, value *big.Int) (ret []byte, con
99     // CreateAddress根据给定的地址和nonce生成一个以太坊地址
100    contractAddr = crypto.CreateAddress(caller.Address(), evm.StateDB.GetNonce(caller.Address(
101    return evm.create(caller, &codeAndHash{code: code}, gas, value, contractAddr)
102 }
103
104 // Create2 creates a new contract using code as deployment code.
105 //
106 // The different between Create2 with Create is Create2 uses sha3(0xff ++ msg.sender ++ salt ++
107 // instead of the usual sender-and-nonce-hash as the address where the contract is initialized at.
108 // Create2和Create的区别就是, Create2使用sha3(0xff ++ msg.sender ++ salt ++ sha3(init_code))
109 // 而不是通常的msg.sender + nonce的hash
110 func (evm *EVM) Create2(caller ContractRef, code []byte, gas uint64, endowment *big.Int, salt *bi
111    codeAndHash := &codeAndHash{code: code}
112    // CreateAddress2根据给定的sender地址、初始代码的hash、salt生成一个以太坊地址
113    contractAddr = crypto.CreateAddress2(caller.Address(), common.BigToHash(salt), codeAndHa
114    return evm.create(caller, codeAndHash, gas, endowment, contractAddr)
115 }

```

如果交易的接收者为空, 则代表此条交易的目的是要创建一条合约。此方法中存储的合约代码是合约运行后的返回码, 而不是原来交易中的数据 (即 `Transaction.data.Payload`, 或者说 `EVM.Create` 方法的 `code` 参数)。这是因为合约源代码在编译成二进制数据时, 除了合约原有的代码外, 编译器还另外插入了一些代码, 以便执行有关功能。对于创建来说, 编译器插入了执行合约构造函数 (即合约的 `constructor` 方法) 的代码。在将编译器编译后的二进制字节码提交到以太坊节点创建合约时, EVM 执行这段二进制代码, 实际上主要执行了合约的 `constructor` 方法, 然后将合约的其它代码返回, 所以才会有这里的 `ret` 变量作为合约的真正代码存储到状态数据库 (`StateDB`) 中。[5]

调用合约

`/core/vm/evm.go`

```

1 // 无论转账或者是执行合约代码都会调用到Call方法, 同时合约里面的call指令也会执行到这里。
2
3 // Call executes the contract associated with the addr with the given input as
4 // parameters. It also handles any necessary value transfer required and takes
5 // the necessary steps to create accounts and reverses the state in case of an
6 // execution error or failed value transfer.
7 // Call 执行与addr相关联的合约, 以给定的input作为参数。
8 // 它还处理所需的任何必要的转账操作, 并采取必要的步骤来创建帐户

```

码农家园

```

12     if evm.vmConfig.NoRecursion && evm.depth > 0 {
13         return nil, gas, nil
14     }
15
16     // Fail if we're trying to execute above the call depth limit
17     // 如果我们试图超过调用深度限制执行代码，则会失败
18     // 调用深度最大1024
19     if evm.depth > int(params.CallCreateDepth) {
20         return nil, gas, ErrDepth
21     }
22     // Fail if we're trying to transfer more than the available balance
23     // 如果我们试图交易比可用余额多的钱，失败
24     // 查看余额是否充足
25     if !evm.Context.CanTransfer(evm.StateDB, caller.Address(), value) {
26         return nil, gas, ErrInsufficientBalance
27     }
28
29     var (
30         // 接收方
31         to = AccountRef(addr)
32         // 系统当前状态快照
33         snapshot = evm.StateDB.Snapshot()
34     )
35     // Exist报告给定帐户是否存在。值得注意的是，对于自杀账户也应该返回true。
36     if !evm.StateDB.Exist(addr) { //地址不存在
37         // 检查是否是预编译的合约
38         precompiles := PrecompiledContractsHomestead
39         if evm.chainRules.IsByzantium {
40             precompiles = PrecompiledContractsByzantium
41         }
42         if evm.chainRules.IsIstanbul {
43             precompiles = PrecompiledContractsIstanbul
44         }
45         // 不是预编译合约，IsEIP158，并且value的值为0
46         if precompiles[addr] == nil && evm.chainRules.IsEIP158 && value.Sign() == 0 {
47             // Calling a non existing account, don't do anything, but ping the tracer
48             // 调用不存在的账户，什么都不做，但触发跟踪程序。不消耗Gas
49             // Debug模式
50             if evm.vmConfig.Debug && evm.depth == 0 {
51                 evm.vmConfig.Tracer.CaptureStart(caller.Address(), addr, false, input, gas, value)
52                 evm.vmConfig.Tracer.CaptureEnd(ret, 0, 0, nil)
53             }
54             return nil, gas, nil
55         }
56         // 在本地状态中创建指定地址addr的状态
57         evm.StateDB.CreateAccount(addr)
58     }
59     // 执行交易
60     evm.Transfer(evm.StateDB, caller.Address(), to.Address(), value)
61     // Initialise a new contract and set the code that is to be used by the EVM.
62     // The contract is a scoped environment for this execution context only.
63     // 初始化一个新的合约，并设置EVM要使用的代码。
64     // 这个合约只是此执行上下文作用域内的环境。
65     contract := NewContract(caller, to, value, gas)
66     contract.SetCallCode(&addr, evm.StateDB.GetCodeHash(addr), evm.StateDB.GetCode(addr))
67
68     // Even if the account has no code, we need to continue because it might be a precompile
69     // 即使该账户没有代码，也需要继续执行，因为它可能是预编译的
70     start := time.Now() //开始时间

```

码农家园

```

74     if evm.vmConfig.Debug && evm.depth == 0 {
75         evm.vmConfig.Tracer.CaptureStart(caller.Address(), addr, false, input, gas, value)
76
77         defer func() { // Lazy evaluation of the parameters 参数的延迟计算
78             evm.vmConfig.Tracer.CaptureEnd(ret, gas-contract.Gas, time.Since(start), err)
79         }()
80     }
81     // 运行该合约
82     ret, err = run(evm, contract, input, false)
83
84     // When an error was returned by the EVM or when setting the creation code
85     // above we revert to the snapshot and consume any gas remaining. Additionally
86     // when we're in homestead this also counts for code storage gas errors.
87     // 当EVM返回一个错误时，或者在设置上面的创建代码时，我们将恢复系统状态到快照并用完剩
88     // 此外，当我们在homestead版本时，代码存储gas错误也会触发上述操作。
89     if err != nil {
90         evm.StateDB.RevertToSnapshot(snapshot) // 回滚状态到快照
91
92         if err != errExecutionReverted {
93             // 如果是由revert指令触发的错误，因为ICO一般设置了人数限制或者资金限制
94             // 在大家抢购的时候很可能会触发这些限制条件，导致被抽走不少钱。这个时候
95             // 又不能设置比较低的GasPrice和GasLimit。因为要速度快。
96             // 那么不会使用剩下的全部Gas，而是只会使用代码执行的Gas
97             // 不然会被抽走 GasLimit * GasPrice的钱，那可不少。
98             contract.UseGas(contract.Gas)
99         }
100     }
101     return ret, contract.Gas, err
102 }

```

该函数首先判断递归层次和合约调用者是否有足够的余额。需要注意的是 **input** 参数，这是调用合约的 **public** 方法的参数。判断合约地址是否存在，使用当前的信息生成一个合约对象，从状态数据库中获取合约的代码，填充到合约对象中。

一般情况下，被调用的合约地址应该存在于以太坊状态数据库中，也就是合约已经创建。否则就返回失败。但有一种例外，就是被调用的合约地址是预编译的情况，此时即使地址不在状态数据库中，也要立即创建一个。

最后主要是对 `run` 函数的调用，然后处理其返回值并返回。

Call与Create异同

Call方法和create方法的逻辑大体相同，通过执行不同的合约指令，达到创建或调用的目的。这里分析一下他们的不同之处：

1.

call调用的是一个已经存在合约账户的合约，create是新建一个合约账户。这个区别在于：合约编译器在编译时，会插入一些代码。在合约创建时，编译器插入的是创建合约的代码，解释器执行这些代码，就可以将合约的真正代码返回；在调用合约时，编译器会插入一些调用合约的代码，只要使用正确的参数执行这些代码，就可以「调用」到我们想调用的合约的 **public** 方法。[5]

2.

call里 **evm.Transfer** 发生在合约的发送方和接收方之间

```
1 evm.Transfer(evm.StateDB, caller.Address(), to.Address(), value)
```

create里则是创建合约的账户和该合约之间

```

1 contractAddr = crypto.CreateAddress(caller.Address(), evm.StateDB.GetNonce(caller.Address()))
2 evm.Transfer(evm.StateDB, caller.Address(), contractAddr, value)

```

码农家园

是为了实现合约的库的特性。它们修改了被调用合约的上下文环境，可以让被调用的合约代码就像自己写的代码一样，从而达到库合约的目的。

`StaticCall` 不允许执行会修改 `storage` 的指令。如果执行过程中遇到这样的指令，就会失败。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，`pure` 类型的函数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，就可以完全在运行阶段来保证这一点了。

这三个特殊的消息调用只能由指令触发，不像 `Call` 可以由外部调用。

1. `CallCode` 与 `Call` 方法的不同在于，它在调用者的上下文中执行给定地址的代码。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`。

```

1 // CallCode executes the contract associated with the addr with the given input
2 // as parameters. It also handles any necessary value transfer required and takes
3 // the necessary steps to create accounts and reverses the state in case of an
4 // execution error or failed value transfer.
5 // 同Call方法
6 //
7 // CallCode differs from Call in the sense that it executes the given address'
8 // code with the caller as context.
9 // CallCode与Call方法的不同在于，它在调用者的上下文中执行给定地址的代码
10 // 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 storag
11 func (evm *EVM) CallCode(caller ContractRef, addr common.Address, input []byte, gas uini
12 // 同Call
13 if evm.vmConfig.NoRecursion && evm.depth > 0 {
14     return nil, gas, nil
15 }
16
17 // Fail if we're trying to execute above the call depth limit
18 if evm.depth > int(params.CallCreateDepth) {
19     return nil, gas, ErrDepth
20 }
21 // Fail if we're trying to transfer more than the available balance
22 if !evm.CanTransfer(evm.StateDB, caller.Address(), value) {
23     return nil, gas, ErrInsufficientBalance
24 }
25
26 var (
27     snapshot = evm.StateDB.Snapshot()
28     // 此处接收方是调用者，与Call不同
29     to = AccountRef(caller.Address())
30 )
31 // Initialise a new contract and set the code that is to be used by the EVM.
32 // The contract is a scoped environment for this execution context only.
33 contract := NewContract(caller, to, value, gas)
34 contract.SetCallCode(&addr, evm.StateDB.GetCodeHash(addr), evm.StateDB.GetCode
35
36 // 运行合约
37 ret, err = run(evm, contract, input, false)
38 if err != nil {

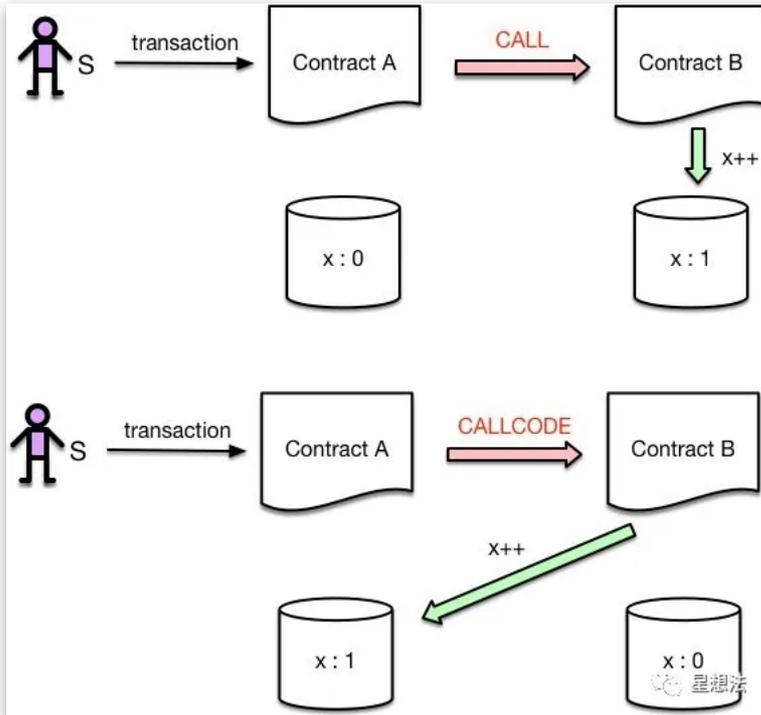
```

码农家园

```

41     contract.UseGas(contract.Gas)
42     }
43     }
44     return ret, contract.Gas, err
45 }

```



2. DelegateCall与CallCode的不同在于，它的调用者被设置为调用者的调用者。具体来说，CallCode 和 DelegateCall 的区别在于：msg.sender 不同。DelegateCall 会一直使用原始调用者的地址，而 CallCode 不会。可以认为 DelegateCall 是 CallCode 的一个 bugfix 版本，官方已经不建议使用 CallCode 了。

```

1 // DelegateCall executes the contract associated with the addr with the given input
2 // as parameters. It reverses the state in case of an execution error.
3 // DelegateCall执行与addr相关联的合约，以给定的input作为参数。
4 // 它在执行错误的情况下回滚状态。
5 //
6 // DelegateCall differs from CallCode in the sense that it executes the given address'
7 // code with the caller as context and the caller is set to the caller of the caller.
8 // DelegateCall与CallCode的不同在于，它在调用者的上下文中执行给定地址的代码并且调用者被设
9 // 具体来说，CALLCODE 和 DELEGATECALL 的区别在于：msg.sender 不同。
10 // DELEGATECALL 会一直使用原始调用者的地址，而 CALLCODE 不会。
11 // 可以认为 DELEGATECALL 是 CALLCODE 的一个 bugfix 版本，官方已经不建议使用 CALLCODE
12 func (evm *EVM) DelegateCall(caller ContractRef, addr common.Address, input []byte, gas uint64)
13     if evm.vmConfig.NoRecursion && evm.depth > 0 {
14         return nil, gas, nil
15     }
16     // Fail if we're trying to execute above the call depth limit
17     if evm.depth > int(params.CallCreateDepth) {
18         return nil, gas, ErrDepth
19     }
20
21     var (
22         snapshot = evm.StateDB.Snapshot()
23         to       = AccountRef(caller.Address())
24     )
25

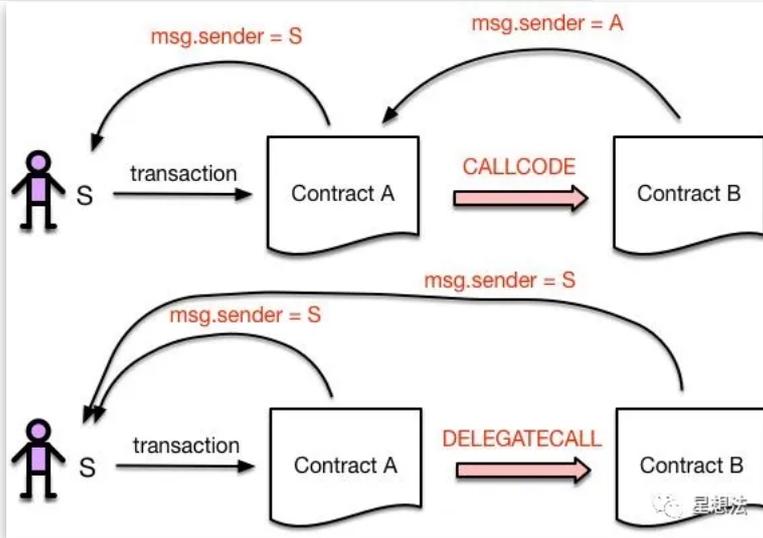
```

码农家园

```

29 contract.SetCallCode(&addr, evm.StateDB.GetCodeHash(addr), evm.StateDB.GetCode(addr))
30
31 ret, err = run(evm, contract, input, false)
32 if err != nil {
33     evm.StateDB.RevertToSnapshot(snapshot)
34     if err != errExecutionReverted {
35         contract.UseGas(contract.Gas)
36     }
37 }
38 return ret, contract.Gas, err
39 }

```



3. staticCall在调用期间不允许执行任何修改状态的操作,试图修改状态的指令会引发异常。

```

1 // StaticCall executes the contract associated with the addr with the given input
2 // as parameters while disallowing any modifications to the state during the call.
3 // Opcodes that attempt to perform such modifications will result in exceptions
4 // instead of performing the modifications.
5 // StaticCall执行与addr相关联的合约, 以给定的input作为参数并且在调用期间不允许执行任
6 // 试图修改状态的指令会引发异常, 而不是执行修改操作。
7 func (evm *EVM) StaticCall(caller ContractRef, addr common.Address, input []byte, gas uir
8     if evm.vmConfig.NoRecursion && evm.depth > 0 {
9         return nil, gas, nil
10    }
11    // Fail if we're trying to execute above the call depth limit
12    if evm.depth > int(params.CallCreateDepth) {
13        return nil, gas, ErrDepth
14    }
15
16    var (
17        to = AccountRef(addr)
18        snapshot = evm.StateDB.Snapshot()
19    )
20    // Initialise a new contract and set the code that is to be used by the EVM.
21    // The contract is a scoped environment for this execution context only.
22    contract := NewContract(caller, to, new(big.Int), gas)
23    contract.SetCallCode(&addr, evm.StateDB.GetCodeHash(addr), evm.StateDB.GetCode
24

```

码农家园

```

27 // but is the correct thing to do and matters on other networks, in tests, and potential
28 // future scenarios
29 // 我们在这里做一个参数为0的AddBalance操作，只是为了触发一个touch操作。
30 // 这对于主网来说并不重要，Byzantium版本中所有的空值都不见了。
31 // 但这对于其他网络，测试中的或者潜在的应用场景来说，是一件正确而且重要的操作。
32 evm.StateDB.AddBalance(addr, bigZero)
33
34 // When an error was returned by the EVM or when setting the creation code
35 // above we revert to the snapshot and consume any gas remaining. Additionally
36 // when we're in Homestead this also counts for code storage gas errors.
37 ret, err = run(evm, contract, input, true) // readOnly=true, 只读，不允许任何更新状态的:
38 if err != nil {
39     evm.StateDB.RevertToSnapshot(snapshot)
40     if err != errExecutionReverted {
41         contract.UseGas(contract.Gas)
42     }
43 }
44 return ret, contract.Gas, err
45 }

```

evm 运行环境

`/core/vm/evm.go`

```

1 // run runs the given contract and takes care of running precompiles with a fallback to the byte code
2 // run运行给定的合约并通过回退到字节码解释器来运行预编译合约
3 // evm 运行环境 contract 要运行的合约 input 输入 readOnly 只读标志，若为true，不允许进行写入和
4 func run(evm *EVM, contract *Contract, input []byte, readOnly bool) ([]byte, error) {
5     if contract.CodeAddr != nil { //如果合约代码地址不为空
6         precompiles := PrecompiledContractsHomestead // HomeStead预编译合约集
7         // 根据当前采用的链规则采用不同版本的预编译合约集
8         if evm.chainRules.IsByzantium {
9             precompiles = PrecompiledContractsByzantium
10        }
11        if evm.chainRules.IsIstanbul {
12            precompiles = PrecompiledContractsIstanbul
13        }
14        // 如果合约代码地址对应的是预编译合约，则调用RunPrecompiledContract
15        if p := precompiles[*contract.CodeAddr]; p != nil {
16            return RunPrecompiledContract(p, input, contract)
17        }
18    }
19    for _, interpreter := range evm.interpreters {
20        // CanRun告诉当前解释器是否可以运行当前合约，合约作为参数传递。
21        if interpreter.CanRun(contract.Code) {
22            if evm.interpreter != interpreter { // 如果evm的解释器不是当前解释器
23                // Ensure that the interpreter pointer is set back
24                // to its current value upon return.
25                // 确保解释器指针在返回时被设置回当前值。
26                defer func(i Interpreter) {
27                    evm.interpreter = i
28                }(evm.interpreter)
29                evm.interpreter = interpreter // 那就设置为当前解释器
30            }
31            // 解释器运行并返回结果
32            return interpreter.Run(contract, input, readOnly)

```

码农家园

```

36     return nil, ErrNoCompatibleInterpreter
37 }

```

函数前半部分判断合约的地址是否是一些特殊地址，如果是则执行其对应的对象的 Run 方法。这些特殊的地址都是一些预编译合约。以下是一个预编译合约集：

`core/vm/contracts.go`

```

1 // PrecompiledContract is the basic interface for native Go contracts. The implementation
2 // requires a deterministic gas count based on the input size of the Run method of the
3 // contract.
4 // PrecompiledContract是本地Go合约的基本接口。
5 // 该接口的实现需要基于合约的运行方法的输入大小来确定gas。
6 type PrecompiledContract interface {
7     // RequiredGas 负责计算合约的gas使用量
8     RequiredGas(input []byte) uint64 // RequiredPrice calculates the contract gas use
9     // Run负责运行预编译好的合约
10    Run(input []byte) ([]byte, error) // Run runs the precompiled contract
11 }
12
13 // homestead byzantium Istanbul三种预编译的合约集
14
15
16 // PrecompiledContractsIstanbul contains the default set of pre-compiled Ethereum
17 // contracts used in the Istanbul release.
18 // Istanbul版本的预编译合约集
19 var PrecompiledContractsIstanbul = map[common.Address]PrecompiledContract{
20     common.BytesToAddress([]byte{1}): &ecrecover{},
21     common.BytesToAddress([]byte{2}): &sha256hash{},
22     common.BytesToAddress([]byte{3}): &ripemd160hash{},
23     common.BytesToAddress([]byte{4}): &dataCopy{},
24     common.BytesToAddress([]byte{5}): &bigModExp{},
25     common.BytesToAddress([]byte{6}): &bn256AddIstanbul{},
26     common.BytesToAddress([]byte{7}): &bn256ScalarMulIstanbul{},
27     common.BytesToAddress([]byte{8}): &bn256PairingIstanbul{},
28     common.BytesToAddress([]byte{9}): &blake2F{},
29 }

```

run 函数的后半部分代码是一个 for 循环，从当前 EVM 对象中选择一个可以运行的解释器，运行当前的合约并返回。当前源代码中只有一个版本的解释器，就是 `EVMInterpreter`。下一代解释器好像叫做 `EWASMInterpreter`。

解释器执行

`core/vm/interpreter.go`

```

1 // Run loops and evaluates the contract's code with the given input data and returns
2 // the return byte-slice and an error if one occurred.
3 // 用给定的入参循环执行合约的代码，并返回返回结果的字节切片，如果出现错误的话返回错误。
4 //
5 // It's important to note that any errors returned by the interpreter should be
6 // considered a revert-and-consume-all-gas operation except for
7 // errExecutionReverted which means revert-and-keep-gas-left.
8 // 应该注意的是，除了errExecutionReverted错误表示回滚状态但保留gas以外，解释器返回的任何
9 func (in *EVMInterpreter) Run(contract *Contract, input []byte, readOnly bool) (ret []byte, err error)
10     if in.intPool == nil { // 初始化intPool
11         in.intPool = poolOfIntPools.get()
12         defer func() { // 用完再放回去
13             poolOfIntPools.put(in.intPool)
14             in.intPool = nil
15         }()

```

码农家园

```

19 // 调用深度自增，最大调用深度为1024.
20 in.evm.depth++
21 defer func() { in.evm.depth-- }() // 合约执行完，调用深度减一。
22
23 // Make sure the readOnly is only set if we aren't in readOnly yet.
24 // This makes also sure that the readOnly flag isn't removed for child calls.
25 // 确保只有在还没有设置readOnly时才设置readOnly。
26 // 这也确保了readOnly标志不会被子调用移除。
27 if readOnly && !in.readOnly {
28     in.readOnly = true
29     defer func() { in.readOnly = false }()
30 }
31
32 // Reset the previous call's return data. It's unimportant to preserve the old buffer
33 // as every returning call will return new data anyway.
34 // 重置前一个调用的返回数据。保留旧的缓冲区并不重要，因为每次返回调用都会返回新的数据
35 in.returnData = nil
36
37 // Don't bother with the execution if there's no code.
38 // 如果没有代码，就不用执行。直接返回
39 if len(contract.Code) == 0 {
40     return nil, nil
41 }
42
43 var (
44     op OpCode // current opcode 当前指令
45     mem = NewMemory() // bound memory 绑定内存
46     stack = newstack() // local stack 本地堆栈
47     // For optimisation reason we're using uint64 as the program counter.
48     // It's theoretically possible to go above 2^64. The YP defines the PC
49     // to be uint256. Practically much less so feasible.
50     // 出于优化的原因，我们使用uint64作为程序计数器。理论上有可能超过2^64。
51     // YP将PC定义为uint256。实际上不太可行。
52     pc = uint64(0) // program counter 程序计算器
53     cost uint64
54     // copies used by tracer
55     // 跟踪程序使用的拷贝
56     pcCopy uint64 // needed for the deferred Tracer 延迟的跟踪程序需要此字段
57     gasCopy uint64 // for Tracer to log gas remaining before execution 用于跟踪程序记录执行前的
58     logged bool // deferred Tracer should ignore already logged steps 延迟跟踪程序应忽略已记
59     res []byte // result of the opcode execution function 指令执行函数的结果
60 )
61 contract.Input = input
62
63 // Reclaim the stack as an int pool when the execution stops
64 // 当执行停止时，将堆栈作为int池回收
65 defer func() { in.intPool.put(stack.data...) }()
66
67 // 若处于调试模式，执行结束记录状态
68 if in.cfg.Debug {
69     defer func() {
70         if err != nil {
71             if !logged {
72                 in.cfg.Tracer.CaptureState(in.evm, pcCopy, op, gasCopy, cost, mem, stack, contract,
73             } else {
74                 in.cfg.Tracer.CaptureFault(in.evm, pcCopy, op, gasCopy, cost, mem, stack, contract,
75             }
76         }
77     }()

```

码农家园

```

81 // the execution of one of the operations or until the done flag is set by the
82 // parent context.
83 // 解释器的主循环。解释器的主要循环，直到遇到STOP，RETURN，SELFDESTRUCT指令被
84 // 或者是遇到任意错误，或者父上下文设置了done标志。
85 for atomic.LoadInt32(&in.evm.abort) == 0 { // 程序没被终止
86     if in.cfg.Debug {
87         // Capture pre-execution values for tracing.
88         // 记录执行前的值以进行跟踪。
89         logged, pcCopy, gasCopy = false, pc, contract.Gas
90     }
91
92     // Get the operation from the jump table and validate the stack to ensure there are
93     // enough stack items available to perform the operation.
94     // 从指令表拿到对应的operation，并验证堆栈，以确保有足够的堆栈项可用于执行操作。
95     op = contract.GetOp(pc)
96     operation := in.cfg.JumpTable[op]
97     if !operation.valid {
98         return nil, fmt.Errorf("invalid opcode 0x%x", int(op))
99     }
100    // Validate stack
101    // 验证堆栈
102    if sLen := stack.len(); sLen < operation.minStack {
103        return nil, fmt.Errorf("stack underflow (%d <=> %d)", sLen, operation.minStack)
104    } else if sLen > operation.maxStack {
105        return nil, fmt.Errorf("stack limit reached %d (%d)", sLen, operation.maxStack)
106    }
107    // If the operation is valid, enforce and write restrictions
108    // 如果operation有效，检查是否有写入限制
109    if in.readOnly && in.evm.chainRules.IsByzantium {
110        // If the interpreter is operating in readonly mode, make sure no
111        // state-modifying operation is performed. The 3rd stack item
112        // for a call operation is the value. Transferring value from one
113        // account to the others means the state is modified and should also
114        // return with an error.
115        // 如果解释器在只读模式下运行，请确保不执行任何状态修改操作。
116        // 第三个堆栈项是call指令携带的值。将值从一个帐户转移到其他帐户意味着状态被修改，
117        if operation.writes || (op == CALL && stack.Back(2).Sign() != 0) {
118            return nil, errWriteProtection
119        }
120    }
121    // Static portion of gas
122    // gas静态部分
123    cost = operation.constantGas // For tracing 用于跟踪程序
124    if !contract.UseGas(operation.constantGas) { // 先扣掉固定gas
125        return nil, ErrOutOfGas
126    }
127
128    // 如果将要执行的指令需要用到内存存储空间，则计算所需要的空间大小
129    var memorySize uint64
130    // calculate the new memory size and expand the memory to fit
131    // the operation
132    // Memory check needs to be done prior to evaluating the dynamic gas portion,
133    // to detect calculation overflows
134    // 计算新的内存大小并扩展内存以适应该指令。在评估动态气体部分之前，需要进行内存检查
135    // 计算内存使用量
136    if operation.memorySize != nil {
137        memSize, overflow := operation.memorySize(stack)
138        if overflow {
139            return nil, errGasUintOverflow

```

码农家园

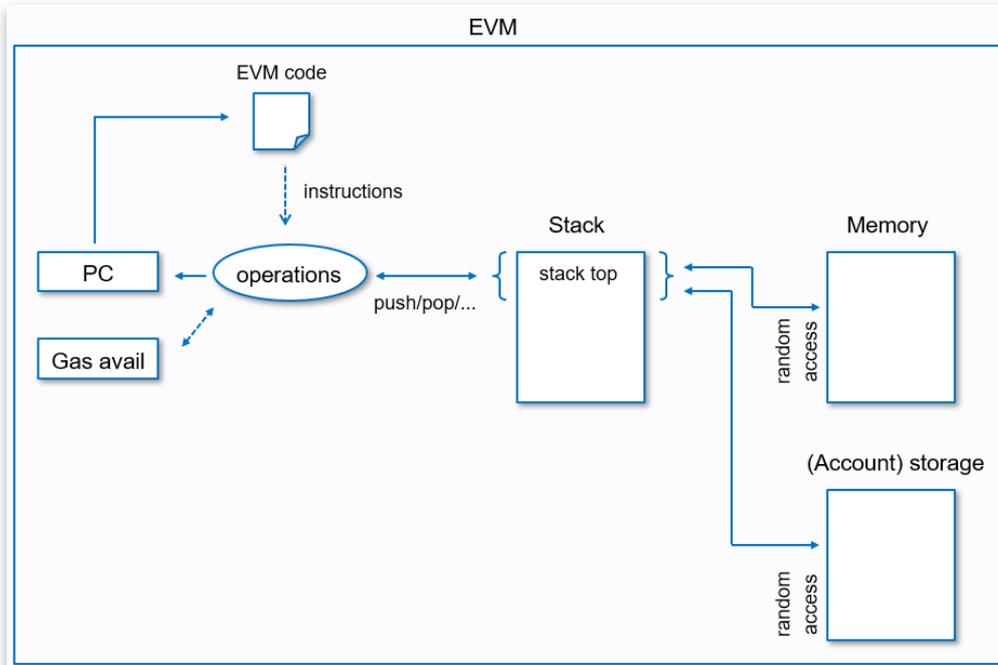
```

143 // 内存扩展为32字节的字。Gas也以字为单位进行计算。
144 if memorySize, overflow = math.SafeMul(toWordSize(memSize), 32); overflow {
145     return nil, errGasUintOverflow
146 }
147 }
148 // Dynamic portion of gas
149 // consume the gas and return an error if not enough gas is available.
150 // cost is explicitly set so that the capture state defer method can get the proper cost
151 // gas的动态部分
152 // 用掉gas, 如果gas不足返回一个错误。
153 // cost是显式设置的, 这样随后捕获状态的方法可以获得正确的cost。
154 if operation.dynamicGas != nil {
155     var dynamicCost uint64
156     // 调用指令对应的gas计算函数
157     dynamicCost, err = operation.dynamicGas(in.evm, contract, stack, mem, memorySize)
158     // gas 花费
159     cost += dynamicCost // total cost, for debug tracing
160     if err != nil || !contract.UseGas(dynamicCost) { // 用掉gas
161         return nil, ErrOutOfGas
162     }
163 }
164 if memorySize > 0 {
165     mem.Resize(memorySize) // 扩展内存
166 }
167
168 // 记录状态
169 if in.cfg.Debug {
170     in.cfg.Tracer.CaptureState(in.evm, pc, op, gasCopy, cost, mem, stack, contract, in.evm.de
171     logged = true
172 }
173
174 // execute the operation
175 // 执行指令, 调用指令对应的执行函数
176 res, err = operation.execute(&pc, in, contract, mem, stack)
177 // verifyPool is a build flag. Pool verification makes sure the integrity
178 // of the integer pool by comparing values to a default value.
179 // verifyPool是一个生成标志。池的验证函数通过将值与默认值进行比较来确保整数池的完整性
180 if verifyPool {
181     verifyIntegerPool(in.intPool)
182 }
183 // if the operation clears the return data (e.g. it has returning data)
184 // set the last return to the result of the operation.
185 // 如果有返回值, 那么就设置返回值。只有最后一个返回有效。
186 if operation.returns {
187     in.returnData = res
188 }
189
190 switch {
191 case err != nil:
192     return nil, err
193 case operation.reverts:
194     return res, errExecutionReverted
195 case operation.halts:
196     return res, nil
197 case loperation.jumps: // 若不跳转, 程序计数器向前移动一个指令。
198     pc++
199 }
200 }

```

码农家园

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。总的来说该方法主循环就是从给定的代码的第 0 个字节开始执行，直到退出。



首先 **PC** 会从合约字节码中读取一个 **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gas，如果gas用光则执行失败，返回 **ErrOutOfGas** 错误。如果油费充足，则调用相应的 **execute** 函数执行该指令，根据指令类型的不同，会分别对 **Stack**、**Memory** 或者 **StatedB** 进行读写操作。

可能出现的错误(errors)

在 `core/vm/errors.go` 中列举了执行时错误：

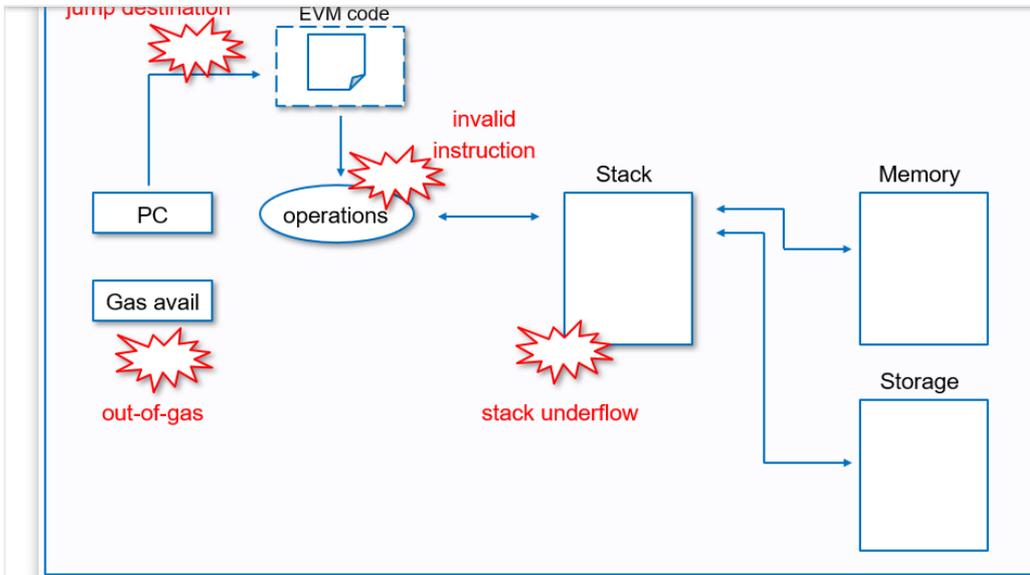
```

1 // List execution errors
2 // 列出执行时错误
3 var (
4     // gas不足
5     ErrOutOfGas = errors.New("out of gas")
6     // 合约创建代码存储gas不足
7     ErrCodeStoreOutOfGas = errors.New("contract creation code storage out of gas")
8     // 超过了最大调用深度
9     ErrDepth = errors.New("max call depth exceeded")
10    // 日志的数量达到了指定的限制
11    ErrTraceLimitReached = errors.New("the number of logs reached the specified limit")
12    // 交易余额不足
13    ErrInsufficientBalance = errors.New("insufficient balance for transfer")
14    // 合约地址冲突
15    ErrContractAddressCollision = errors.New("contract address collision")
16    // 没有兼容的解释器
17    ErrNoCompatibleInterpreter = errors.New("no compatible interpreter")
18 )

```

下图也描述了一些异常发生的场景：

码农家园



参考文献

1. Ethereum Yellow Paper
ETHEREUM: A SECURE DECENTRALISED GENERALISED TRANSACTION LEDGER
<https://ethereum.github.io/yellowpaper/paper.pdf>
2. Ethereum White Paper
A Next-Generation Smart Contract and Decentralized Application Platform
<https://github.com/ethereum/wiki/wiki/White-Paper>
3. Ethereum EVM Illustrated
<https://github.com/takenobu-hs/ethereum-evm-illustrated>
4. Go Ethereum Code Analysis
<https://github.com/ZtesoftCS/go-ethereum-code-analysis>
5. 以太坊源码解析: evm
<https://yangzhe.me/2019/08/12/ethereum-evm/>
6. 以太坊 - 深入浅出虚拟机
<https://learnblockchain.cn/2019/04/09/easy-evm/>