编译原理之以太坊EVM虚拟机源码解析

一、虚拟机外部账户操作流程

虚拟机首先将交易的类型转换成虚拟机能够识别的消息,根据这些消息计算需要花费的gas，然后在创建虚拟机对象，当虚拟机执行完毕后返回收据对象。代码如下

文件：/core/state_processor.go --- Process()

```
for i, tx := range block.Transactions() {
    statedb.Prepare(tx.Hash(), block.Hash(), i)
    receipt, _, err := ApplyTransaction(p.config, p.bc, nil, gp, statedb, header, t
x, totalUsedGas, cfg)
    if err != nil {
        return nil, nil, nil, err
    }
    receipts = append(receipts, receipt)
    allLogs = append(allLogs, receipt.Logs...)
}
```

我们可以看出首先是将block里面所有的tx逐个遍历执行，通过ApplyTransaction函数，每次执行完返回一个收据(Receipt)对象和日志对象（allLogs）。下面我们看一下收据的数据类型

```
type Receipt struct {
    // Consensus fields
    PostState         []byte   `json:"root"`
    Failed            bool     `json:"failed"`
    CumulativeGasUsed *big.Int `json:"cumulativeGasUsed" gencodec:"required"`
    Bloom             Bloom    `json:"logsBloom"         gencodec:"required"`
    Logs              []*Log   `json:"logs"              gencodec:"required"`

// Implementation fields (don't reorder!)
TxHash          common.Hash    `json:"transactionHash" gencodec:"required"`
ContractAddress common.Address `json:"contractAddress"`
GasUsed         *big.Int       `json:"gasUsed" gencodec:"required"`
}
```

可以看出Receipt实际上是一个结构体，结构体中有很多成员变量，下面我们分析一下。
Logs：日志类型数组，其中每一个Log对象记录了交易过程中一小步的操作。所以，每一个交易的执行结果，由一个Receipt对象来表示，用Receipt中的Logs数组保存此交易过程所有操作步

骤。Log数组的成员很重要，比如在不同Ethereum节点的相互同步过程中，待同步区块的Log数组有助于验证同步中收到的block是否正确和完整，所以会被单独同步传输。

PostState：保存了创建该Receipt对象时，将保存区块中所有“帐户”的工作状态。Ethereum 里用stateObject来表示一个账户，这个账户可转帐执行转账, 可执行交易, 账户的用Address类型变量保存地址。这个PostState就是当时所在区块中所有stateObject对象通过RLP编码后的账户状态值。

Bloom: Ethereum内部实现的一个256bit长Bloom Filter。Bloom Filter定义布隆过滤器，它的作用为可用来快速验证一个新收到的对象是否处于一个已知的大量对象集合之中。这里Receipt的Bloom，被用以验证某个给定的Log是否处于Receipt已有的Log数组中。

接下来我们来看一下ApplyTransaction这个函数，学习如何封装EVM对象和Message对象
文件：/core/state_processor.go --- ApplyTransaction()

```
我们可以看出是通过tx.AsMessage将交易的过程封装成消息
msg, err := tx.AsMessage(types.MakeSigner(config, header.Number))
if err != nil { return nil, nil, err }

其次为通过vm.NewEVM函数创建虚拟机对象
context := NewEVMContext(msg, header, bc, author)
vmenv := vm.NewEVM(context, statedb, config, cfg)

最后完成交易的执行，创建一个收据Receipt对象，最后返回该Recetip对象，以及整个tx执行过程所消耗Gas数量
_, gas, failed, err := ApplyMessage(vmenv, msg, gp)
...
```

下面我们来看一下ApplyMessage函数
文件：/core/state_transition.go --- ApplyMessage()

```
发现调用了TransitionDb函数，代码大致为
, _, gasUsed, failed, err := st.TransitionDb()
```

我们来看一下TransitionDb函数
文件：/core/state_transition.go --- TransitionDb()

```
if err = st.preCheck(); err != nil {
    return
}
msg := st.msg
sender := vm.AccountRef(msg.From())
```

```
homestead := st.evm.ChainConfig().IsHomestead(st.evm.BlockNumber)
contractCreation := msg.To() == nil

// Pay intrinsic gas
gas, err := IntrinsicGas(st.data, contractCreation, homestead)
if err != nil {
    return nil, 0, false, err
}
if err = st.useGas(gas); err != nil {
    return nil, 0, false, err
}

var (
    evm = st.evm
    // vm errors do not effect consensus and are therefor
    // not assigned to err, except for insufficient balance
    // error.
    vmerr error
)
if contractCreation {
    ret, _, st.gas, vmerr = evm.Create(sender, st.data, st.gas, st.value)
} else {
    // Increment the nonce for the next transaction
    st.state.SetNonce(msg.From(), st.state.GetNonce(sender.Address())+1)
    ret, st.gas, vmerr = evm.Call(sender, st.to(), st.data, st.gas, st.value)
}
if vmerr != nil {
    log.Debug("VM returned with error", "err", vmerr)
    // The only possible consensus-error would be if there wasn't
    // sufficient balance to make the transfer happen. The first
    // balance transfer may never fail.
    if vmerr == vm.ErrInsufficientBalance {
        return nil, 0, false, vmerr
    }
}
st.refundGas()
st.state.AddBalance(st.evm.Coinbase, new(big.Int).Mul(new(big.Int).SetUint64(st.gas
Used()), st.gasPrice))

return ret, st.gasUsed(), vmerr != nil, err
//该函数主要功能能为
//购买gas
//计算tx固有gas
//EVM执行
//计算本次执行交易的实际gas消耗
//偿退gas
//奖励所属区块的挖掘者
```

二、虚拟机合约账户操作流程

内部合约账户包括执行转帐，和创建合约并执行合约的指令数组的功能，首先我们来看一下EVM的结构体：
文件：/core/vm/evm.go

```
type EVM struct {
    Context
    StateDB StateDB
    depth int
    chainConfig *params.ChainConfig
    chainRules params.Rules
    vmConfig Config
    interpreter *Interpreter
    abort int32
}
Context --携带交易和区块信息，包扩GasPrice，GasLimit,Number,Difficulty
StateDB StateDB --为EVM提供statedb的数据保存相关操作
interpreter *Interpreter --用来编译解释执行EVM中合约的指令，这部分最重要
```

完成转账交易的转帐操作由Context对象中的TransferFunc类型函数来实现，类似的函数类型，还有CanTransferFunc, 和GetHashFunc。
文件：/core/evm.go --Transfer()

```
// Transfer subtracts amount from sender and adds amount to recipient using the giv
en Db
func Transfer(db vm.StateDB, sender, recipient common.Address, amount *big.Int) {
    db.SubBalance(sender, amount)//转出账户减到一定金额以太币
    db.AddBalance(recipient, amount)//转入账户增加一定金额以太币
}
```

注意转出和转入账户的操作不会立即生效，StateDB 并不是真正的数据库，只是一行为类似数据库的结构体它在内部以Trie的数据结构来管理各个基于地址的账户，可以理解成一个cache；当该账户的信息有变化时，变化先存储在Trie中。仅当整个Block要被插入到BlockChain时，StateDB里缓存的所有账户的所有改动，才会被真正的提交到底层数据库。

合约的创建、赋值,我们先来看一下合约contract结构体
文件：/core/vm/contract.go

```
type Contract struct {
    CallerAddress common.Address
    caller        ContractRef
```

```
    self          ContractRef
    jumpdests destinations
    Code     []byte
    CodeHash common.Hash
    CodeAddr *common.Address
    Input    []byte
    Gas   uint64
    value *big.Int
    Args []byte
    DelegateCall bool
}
    caller        ContractRef  //转账转出方地址
    self          ContractRef  //转入方地址
    jumpdests destinations // result of JUMPDEST analysis.
    Code     []byte  //指令数组，其中每一个byte都对应于一个预定义的虚拟机指令
    CodeHash common.Hash
    CodeAddr *common.Address
    Input    []byte  //数据数组，是指令所操作的数据集合
```

创建合约：call(),create() 这两个函数都在StateProcessor的ApplyTransaction()被调用以执行单个交易，并且都有调用转帐函数完成转帐。我们来看一下call()
文件：/core/vm/evm.go

```
var (
    to = AccountRef(addr)
    snapshot = evm.StateDB.Snapshot()
)
if !evm.StateDB.Exist(addr) {
    precompiles := PrecompiledContractsHomestead
    if evm.ChainConfig().IsByzantium(evm.BlockNumber) {
        precompiles = PrecompiledContractsByzantium
    }
    if precompiles[addr] == nil && evm.ChainConfig().IsEIP158(evm.BlockNumber) && v
alue.Sign() == 0 {
        return nil, gas, nil
    }
    evm.StateDB.CreateAccount(addr)
}

//转账
evm.Transfer(evm.StateDB, caller.Address(), to.Address(), value)

//赋值Contract对象
contract := NewContract(caller, to, value, gas)
contract.SetCallCode(&addr, evm.StateDB.GetCodeHash(addr), evm.StateDB.GetCode(addr
))
```

```
//调用run，执行该合约的指令
ret, err = run(evm, snapshot, contract, input)

if err != nil {
    evm.StateDB.RevertToSnapshot(snapshot)
    if err != errExecutionReverted {
        contract.UseGas(contract.Gas)
    }
}
return ret, contract.Gas, err
```

预编译合约，我们来看一下run():
文件：/core/vm/evm.go

```
if contract.CodeAddr != nil {
    precompiles := PrecompiledContractsHomestead
    if evm.ChainConfig().IsByzantium(evm.BlockNumber) {
        precompiles = PrecompiledContractsByzantium
    }
    if p := precompiles[*contract.CodeAddr]; p != nil {
        return RunPrecompiledContract(p, input, contract)
    }
}
return evm.interpreter.Run(snapshot, contract, input)
```

由此可见如果待执行的Contract对象恰好属于一组预编译的合约集合，此时以指令地址CodeAddr为匹配项，那么它可以直接运行，没有经过预编译的Contract，才会由Interpreter解释执行。这里的"预编译"，可理解为不需要编译(解释)指令(Code)。预编译的合约，其逻辑全部固定且已知，所以执行中不再需要Code，仅需Input即可。

解释器执行合约的指令，我们来看一下interpreter.go，可以看到一个Config结构体
文件：/core/vm/.interpreter.go

```
type Config struct {
    Debug bool
    EnableJit bool
    ForceJit bool
    Tracer Tracer
    NoRecursion bool
    DisableGasMetering bool
    EnablePreimageRecording bool
    JumpTable [256]operation  //
}
```

每个operation对象为对应一个已定义的虚拟机指令，含有的四个函数变量execute, gasCost, validateStack, memorySize 提供了这个虚拟机指令所代表的所有操作。每个指令长度1byte，Contract对象的成员变量Code类型为[]byte，也就是这些虚拟机指令的任意集合。operation对象的函数操作，主要会用到Stack，Memory, IntPool这几个自定义的数据结构。