ethereum-evm代码分析(v1.8.24)

1、引导

本篇为分析以太坊虚拟机部分的代码，但是为了大家方便理解，本次打算讲解一笔交易发送以太坊客户端，再到上链作为一个完整链路来分析，并把以太坊evm作为重点来讲解。

1.1 sendTx()

第一部分主要涉及交易池部分逻辑代码，不了解的部分请翻阅tx_pool代码分析部分

```
# go-ethereum/eth/api_backend.go
//当用户需要发送一笔交易时候，实际上是调用了以太坊客户端的sendTx方法
func (b *EthAPIBackend) SendTx(ctx context.Context, signedTx *types.Transaction) error {
    // 调用以太坊交易池，将tx添加到本地交易列表中
    return b.eth.txPool.AddLocal(signedTx)
}
// AddLocal enqueues a single transaction into the pool if it is valid, marking
// the sender as a local one in the mean time, ensuring it goes around the local
// pricing constraints.
func (pool *TxPool) AddLocal(tx *types.Transaction) error {
    return pool.addTx(tx, !pool.config.NoLocals)
}
// addTx enqueues a single transaction into the pool if it is valid.
func (pool *TxPool) addTx(tx *types.Transaction, local bool) error {
    pool.mu.Lock()
    defer pool.mu.Unlock()
    // 在此将交易放入交易池，若相同nonce的交易存在，那么尝试覆盖并广播交易
    // 若是新交易则放入账户的queue中，如果账户queue中存在则覆盖，
    // Try to inject the transaction and update any state
    replace, err := pool.add(tx, local)
    if err != nil {
        return err
    }
    // 如果我们添加的是一笔新的交易，那么执行以下交易升级
    // If we added a new transaction, run promotion checks and return
    if !replace {
        from, _ := types.Sender(pool.signer, tx) // already validated
        pool.promoteExecutables([]common.Address{from})
    }
    return nil
}
```

从上面代码可以看出，用户发送交易到以太坊客户端，以太坊拿到客户端以后如何放入交易池的逻辑。

在上面逻辑执行以后若交易能进入pending列表，那么交易就会被广播，通过p2p模块广播给若干接点。

如交易没进入pending列表，进入了账户queue中，那么随后若交易正常，那么也会进入账户的pending列表中，并且会被广播。

若本地节点是挖矿节点则将交易打包成区块进行上链，那么下面转到挖矿部分代码，看看如何拿到交易进行执行

1.2 commitNewWork()

此部分为挖矿代码部分的主要代码部分，在这个期间，做了将pending里面的交易拿出来执行后打包到一个新区块中的工作。那么下面开始分析交易如何被拿出来执行的

```
// commitNewWork generates several new sealing tasks based on the parent block.
func (w *worker) commitNewWork(interrupt *int32, noempty bool, timestamp int64) {
    ..........................
    // 以上部分与本次分析相关性不大，忽略处理

    // 下面从交易池拿出所有pending的交易
    // Fill the block with all available pending transactions.
    pending, err := w.eth.TxPool().Pending()
    if err != nil {
        log.Error("Failed to fetch pending transactions", "err", err)
        return
    }
    // Short circuit if there is no available pending transactions
    if len(pending) == 0 {
        w.updateSnapshot()
        return
    }
    // 将交易分为远程的和本地的交易
    // Split the pending transactions into locals and remotes
    localTxs, remoteTxs := make(map[common.Address]types.Transactions), pending
    for _, account := range w.eth.TxPool().Locals() {
        if txs := remoteTxs[account]; len(txs) > 0 {
            delete(remoteTxs, account)
            localTxs[account] = txs
        }
    }
    if len(localTxs) > 0 {
        txs := types.NewTransactionsByPriceAndNonce(w.current.signer, localTxs)
         // 下面这一步主要是开始将交易进行执行了
        if w.commitTransactions(txs, w.coinbase, interrupt) {
            return
        }
    }
    if len(remoteTxs) > 0 {
        txs := types.NewTransactionsByPriceAndNonce(w.current.signer, remoteTxs)
        if w.commitTransactions(txs, w.coinbase, interrupt) {
            return
        }
    }
    // 最后提交一个 区块
    w.commit(uncles, w.fullTaskHook, true, tstart)
}
```

下面部分代码主要将需要打包在区块中的交易逐个执行，逻辑如下

```
func (w *worker) commitTransactions(txs *types.TransactionsByPriceAndNonce, coinbase
common.Address, interrupt *int32) bool {
    // Short circuit if current is nil
    if w.current == nil {
        return true
```

```
    }

    if w.current.gasPool == nil {
        w.current.gasPool = new(core.GasPool).AddGas(w.current.header.GasLimit)
    }
    // 合并logs
    var coalescedLogs []*types.Log

    for {
        // In the following three cases, we will interrupt the execution of the transaction.
        // (1) new head block event arrival, the interrupt signal is 1
        // (2) worker start or restart, the interrupt signal is 1
        // (3) worker recreate the mining block with any newly arrived transactions, the interrupt
signal is 2.
        // For the first two cases, the semi-finished work will be discarded.
        // For the third case, the semi-finished work will be submitted to the consensus engine.
        if interrupt != nil && atomic.LoadInt32(interrupt) != commitInterruptNone {
            // Notify resubmit loop to increase resubmitting interval due to too frequent commits.
            if atomic.LoadInt32(interrupt) == commitInterruptResubmit {
                ratio := float64(w.current.header.GasLimit-w.current.gasPool.Gas()) /
float64(w.current.header.GasLimit)
                if ratio < 0.1 {
                    ratio = 0.1
                }
                w.resubmitAdjustCh <- &intervalAdjust{
                    ratio: ratio,
                    inc:   true,
                }
            }
            return atomic.LoadInt32(interrupt) == commitInterruptNewHead
        }
        // 不能低于执行交易的最小gas  21000
        // If we don't have enough gas for any further transactions then we're done
        if w.current.gasPool.Gas() < params.TxGas {
            log.Trace("Not enough gas for further transactions", "have", w.current.gasPool,
"want", params.TxGas)
            break
        }
         // 从txs里面拿出顶端的 gasPrice最小的
        // Retrieve the next transaction and abort if all done
        tx := txs.Peek()
        if tx == nil {
            break
        }
        // Error may be ignored here. The error has already been checked
        // during transaction acceptance is the transaction pool.
        //
        // We use the eip155 signer regardless of the current hf.
        from, _ := types.Sender(w.current.signer, tx)
        // Check whether the tx is replay protected. If we're not in the EIP155 hf
        // phase, start ignoring the sender until we do.
        if tx.Protected() && !w.config.IsEIP155(w.current.header.Number) {
            log.Trace("Ignoring reply protected transaction", "hash", tx.Hash(), "eip155",
w.config.EIP155Block)

            txs.Pop()
            continue
```

```
        }
        // 准备开始执行交易
        // Start executing the transaction
        w.current.state.Prepare(tx.Hash(), common.Hash{}, w.current.tcount)
        // 此段代码将涉及到交易执行，离虚拟机部分很近了
        logs, err := w.commitTransaction(tx, coinbase)
         // 以下部分与本次分析关系不大 ，忽略处理
        ..........................................
}
```

commitTransaction()实际将交易传到了core.ApplyTransaction()，并将tx转换为一个message，并且在ApplyTransaction()构造了虚拟机的context，随后创建一个虚拟机，并调用ApplyMessage后调用TransitionDb()实际将交易执行，具体逻辑如下

```
// ApplyTransaction attempts to apply a transaction to the given state database
// and uses the input parameters for its environment. It returns the receipt
// for the transaction, gas used and an error if the transaction failed,
// indicating the block was invalid.
func ApplyTransaction(config *params.ChainConfig, bc ChainContext, author *common.Address, gp
*GasPool, statedb *state.StateDB, header *types.Header, tx *types.Transaction, usedGas *uint64,
cfg vm.Config) (*types.Receipt, uint64, error) {
    msg, err := tx.AsMessage(types.MakeSigner(config, header.Number))
    if err != nil {
        return nil, 0, err
    }
    // Create a new context to be used in the EVM environment
    context := NewEVMContext(msg, header, bc, author)
    // Create a new environment which holds all relevant information
    // about the transaction and calling mechanisms.
    vmenv := vm.NewEVM(context, statedb, config, cfg)
    // Apply the transaction to the current state (included in the env)
    _, gas, failed, err := ApplyMessage(vmenv, msg, gp)
    ..........................................
}
// ApplyMessage computes the new state by applying the given message
// against the old state within the environment.
//
// ApplyMessage returns the bytes returned by any EVM execution (if it took place),
// the gas used (which includes gas refunds) and an error if it failed. An error always
// indicates a core error meaning that the message would always fail for that particular
// state and would never be accepted within a block.
func ApplyMessage(evm *vm.EVM, msg Message, gp *GasPool) ([]byte, uint64, bool, error) {
    return NewStateTransition(evm, msg, gp).TransitionDb()
}

// TransitionDb will transition the state by applying the current message and
// returning the result including the used gas. It returns an error if failed.
// An error indicates a consensus issue.
func (st *StateTransition) TransitionDb() (ret []byte, usedGas uint64, failed bool, err error) {
    // 做一次预检，主要做了nonce检查和账户余额检查
    if err = st.preCheck(); err != nil {
        return
    }
    msg := st.msg
    sender := vm.AccountRef(msg.From())
    homestead := st.evm.ChainConfig().IsHomestead(st.evm.BlockNumber)
```

```
     // 如果本次tx的to地址为空则为合约创建交易
    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 {
        // 如果是合约创建交易，则调用evm.create()方法
        ret, _, st.gas, vmerr = evm.Create(sender, st.data, st.gas, st.value)
    } else {
        // 要不然调用合约的Call()方法执行交易,在这里先把evm当成黑盒，先知道这里是实际调用evm执行字节码即
可
        // 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
        }
    }
    // 计算回退的gas
    st.refundGas()
    st.state.AddBalance(st.evm.Coinbase, new(big.Int).Mul(new(big.Int).SetUint64(st.gasUsed()),
st.gasPrice))

    return ret, st.gasUsed(), vmerr != nil, err
}
```

1.3 第一小节总结

经过上面主流程的分析以后，现在已经可以知道一笔交易发送到以太坊客户端以后经历了一系列操作后一直到需要虚拟机实际执行交易的部分。

2、以太坊虚拟机部分分析

下面按照惯例先介绍一下以太坊相关数据结构

```
# go-ethereum/core/vm/evm.go
// context是在执行一笔交易时候的附加信息，上下文环境
// Context provides the EVM with auxiliary information. Once provided
// it shouldn't be modified.
type Context struct {
    // canTransfer函数用来判断是否可以执行转账，很简单
    // CanTransfer returns whether the account contains
    // sufficient ether to transfer the value
    CanTransfer CanTransferFunc
    // transfer函数用来执行转账的函数，很简单
    // Transfer transfers ether from one account to the other
    Transfer TransferFunc
    // GetHash returns the hash corresponding to n
    // 获取hash值函数
    GetHash GetHashFunc
    // 下面是交易执行的一些简要附加信息
    // Message information
    Origin   common.Address // Provides information for ORIGIN
    GasPrice *big.Int       // Provides information for GASPRICE

    // Block information
    Coinbase    common.Address // Provides information for COINBASE
    GasLimit    uint64         // Provides information for GASLIMIT
    BlockNumber *big.Int       // Provides information for NUMBER
    Time        *big.Int       // Provides information for TIME
    Difficulty  *big.Int       // Provides information for DIFFICULTY
}
//
// EVM is the Ethereum Virtual Machine base object and provides
// the necessary tools to run a contract on the given state with
// the provided context. It should be noted that any error
// generated through any of the calls should be considered a
// revert-state-and-consume-all-gas operation, no checks on
// specific errors should ever be performed. The interpreter makes
// sure that any errors generated are to be considered faulty code.
// Evm不能被重用且不是线程安全的，每一笔交易都会创建一个新的context和evm
// The EVM should never be reused and is not thread safe.
type EVM struct {
    // Context provides auxiliary(辅助) blockchain related information
    Context
    // StateDB gives access to the underlying state
    StateDB StateDB
    // 当前调用栈的深度
    // Depth is the current call stack
    depth int

    // chainConfig contains information about the current chain
    chainConfig *params.ChainConfig
    // chain rules contains the chain rules for the current epoch
    chainRules params.Rules
    // virtual machine configuration options used to initialise the
    // evm.
    vmConfig Config
    // 解释器
    // global (to this context) ethereum virtual machine
    // used throughout the execution of the tx.
    interpreters []Interpreter
```

```
    interpreter  Interpreter
    // 原子操作
    // abort is used to abort the EVM calling operations
    // NOTE: must be set atomically
    abort int32
    // callGasTemp holds the gas available for the current call. This is needed because the
    // available gas is calculated in gasCall* according to the 63/64 rule and later
    // applied in opCall*.
    callGasTemp uint64
}
// 虚拟机解释器
// EVMInterpreter represents an EVM interpreter
type EVMInterpreter struct {
    evm      *EVM
    cfg      Config
    gasTable params.GasTable

    intPool *intPool

    hasher    keccakState // Keccak256 hasher instance shared across opcodes
    hasherBuf common.Hash // Keccak256 hasher result array shared aross opcodes

    readOnly   bool   // Whether to throw on stateful modifications
    returnData []byte // Last CALL's return data for subsequent reuse
}
// 以太坊操作结构体。以太坊定义了一个长度为256的operation数组来存放虚拟机操作，每一个操作有4个相关函数和6个标记位
type operation struct {
    //实际执行函数
    // execute is the operation function
    execute executionFunc
    // 计算执行花费的gas
    // gasCost is the gas function and returns the gas required for execution
    gasCost gasFunc
    // 验证栈的函数，栈必须符合一定条件比如深度小于1024
    // validateStack validates the stack (size) for the operation
    validateStack stackValidationFunc
    // 返回操作需要的内存大小
    // memorySize returns the memory size required for the operation
    memorySize memorySizeFunc
    //操作标记位
    halts   bool // indicates whether the operation should halt further execution
    jumps   bool // indicates whether the program counter should not increment
    writes  bool // determines whether this a state modifying operation
    valid   bool // indication whether the retrieved operation is valid and known
    reverts bool // determines whether the operation reverts state (implicitly halts)
    returns bool // determines whether the operations sets the return data content
}
```

以上就是以太坊虚拟机相关的主要数据结构，下面开始介绍一些逻辑实现

2.1 go-ethereum/core/evm.go分析

```
# go-ethereum/core/evm.go
// 先介绍一个主要用来创建一些环境信息的逻辑代码
```

```
// 此函数主要用来为以太坊创建一个上下文环境
// NewEVMContext creates a new context for use in the EVM.
func NewEVMContext(msg Message, header *types.Header, chain ChainContext, author *common.Address)
vm.Context {
    // If we don't have an explicit author (i.e. not mining), extract from the header
    // 受益人，也就是挖矿奖励所得者
    var beneficiary common.Address
    if author == nil {
        beneficiary, _ = chain.Engine().Author(header) // Ignore error, we're past header
validation
    } else {
        beneficiary = *author
    }
    // 实际就初始化了以太坊虚拟机执行的上下文环境
    return vm.Context{
        CanTransfer: CanTransfer,
        Transfer:    Transfer,
        GetHash:     GetHashFn(header, chain),
        Origin:      msg.From(),
        Coinbase:    beneficiary,
        BlockNumber: new(big.Int).Set(header.Number),
        Time:        new(big.Int).SetUint64(header.Time),
        Difficulty:  new(big.Int).Set(header.Difficulty),
        GasLimit:    header.GasLimit,
        GasPrice:    new(big.Int).Set(msg.GasPrice()),
    }
}
// 返回一个函数，返回的函数主要用区块号来取区块的hash值
// GetHashFn returns a GetHashFunc which retrieves header hashes by number
func GetHashFn(ref *types.Header, chain ChainContext) func(n uint64) common.Hash {
    var cache map[uint64]common.Hash

    return func(n uint64) common.Hash {
        // 如果没有缓存，那么创建一个缓存，并将传来的区块头的父区块信息放入缓存
        // If there's no hash cache yet, make one
        if cache == nil {
            cache = map[uint64]common.Hash{
                ref.Number.Uint64() - 1: ref.ParentHash,
            }
        }
        // 如果从缓存中找到相关区块号对应的区块hash，那么直接返回
        // Try to fulfill the request from the cache
        if hash, ok := cache[n]; ok {
            return hash
        }
        // 要不然从给的区块往前推，逐个缓存区块号和区块hash，并且在找到要查询的区块号对应的区块hash时候返
回
        // Not cached, iterate the blocks and cache the hashes
        for header := chain.GetHeader(ref.ParentHash, ref.Number.Uint64()-1); header != nil;
header = chain.GetHeader(header.ParentHash, header.Number.Uint64()-1) {
            cache[header.Number.Uint64()-1] = header.ParentHash
            if n == header.Number.Uint64()-1 {
                return header.ParentHash
            }
        }
        return common.Hash{}
    }
```

```
}
// 主要判断一下账户余额够不够转账
// CanTransfer checks whether there are enough funds in the address' account to make a transfer.
// This does not take the necessary gas in to account to make the transfer valid.
func CanTransfer(db vm.StateDB, addr common.Address, amount *big.Int) bool {
    return db.GetBalance(addr).Cmp(amount) >= 0
}
// transfer主要执行了实际的转账操作，给发送者/接受者 减去/加上相应金额，此处的加减不是立即生效
// Transfer subtracts amount from sender and adds amount to recipient using the given Db
func Transfer(db vm.StateDB, sender, recipient common.Address, amount *big.Int) {
    db.SubBalance(sender, amount)
    db.AddBalance(recipient, amount)
}
```

以上主要是以太坊源代码core下面evm.go的主要逻辑分析，其实做了给以太坊虚拟机每次执行创建上下文环境一件事。

2.2 go-ethereum/core/vm/evm.go

下面开始分析以太坊虚拟机的核心文件，vm包下面的evm.go

2.2.1 NewEVM()

```
// 这是创建虚拟机的逻辑，执行交易时每次使用虚拟机都新创建一个
// NewEVM returns a new EVM. The returned EVM is not thread safe and should
// only ever be used *once*.
func NewEVM(ctx Context, statedb StateDB, chainConfig *params.ChainConfig, vmConfig Config) *EVM {
    // 给evm属性初始化变量
    evm := &EVM{
        Context:      ctx,
        StateDB:      statedb,
        vmConfig:     vmConfig,
        chainConfig:  chainConfig,
        chainRules:   chainConfig.Rules(ctx.BlockNumber),
        interpreters: make([]Interpreter, 0, 1),
    }

    if chainConfig.IsEWASM(ctx.BlockNumber) {
        // to be implemented by EVM-C and Wagon PRs.
        // if vmConfig.EWASMInterpreter != "" {
        //  extIntOpts := strings.Split(vmConfig.EWASMInterpreter, ":")
        //  path := extIntOpts[0]
        //  options := []string{}
        //  if len(extIntOpts) > 1 {
        //    options = extIntOpts[1..]
        //  }
        //  evm.interpreters = append(evm.interpreters, NewEVMVCInterpreter(evm, vmConfig,
options))
        // } else {
        //  evm.interpreters = append(evm.interpreters, NewEWASMInterpreter(evm, vmConfig))
        // }
        panic("No supported ewasm interpreter yet.")
    }
    // 创建虚拟机解释器
    // vmConfig.EVMInterpreter will be used by EVM-C, it won't be checked here
    // as we always want to have the built-in EVM as the failover option.
```

```
    evm.interpreters = append(evm.interpreters, NewEVMInterpreter(evm, vmConfig))
    evm.interpreter = evm.interpreters[0]

    return evm
}
```

上面主要是emv在创建时候的简单逻辑，由第一节分析的tx执行过程可知，实际上每次交易调用虚拟机执行时候主要调用了虚拟机的两个方法`evm.Call()`和`evm.Create()`两个方法，那么下面的分析主要由这两个方法展开

2.2.2 evm.Call()

```
// 这个方法是除合约创建以外所tx实际执行都要使用的方法
// Call executes the contract associated with the addr with the given input as
// parameters. It also handles any necessary value transfer required and takes
// the necessary steps to create accounts and reverses the state in case of an
// execution error or failed value transfer.
func (evm *EVM) Call(caller ContractRef, addr common.Address, input []byte, gas uint64, value
*big.Int) (ret []byte, leftOverGas uint64, err error) {
    // 先判断若开启了非递归模式且栈深度大于0 则直接返回
    if evm.vmConfig.NoRecursion && evm.depth > 0 {
        return nil, gas, nil
    }
    // 如果evm的调用栈大于Call/Create方法的最大深度限制1024 那么返回错误
    // Fail if we're trying to execute above the call depth limit
    if evm.depth > int(params.CallCreateDepth) {
        return nil, gas, ErrDepth
    }
    // 如果value字段大于调用者的余额那么返回余额不足错误
    // Fail if we're trying to transfer more than the available balance
    if !evm.Context.CanTransfer(evm.StateDB, caller.Address(), value) {
        return nil, gas, ErrInsufficientBalance
    }
    // 将addr（被调方）地址转为AccountRef类型，用snapshot记录当前状态快照
    var (
        to       = AccountRef(addr)
        snapshot = evm.StateDB.Snapshot()
    )
     // 若addr不存在于statedb中
    if !evm.StateDB.Exist(addr) {
        precompiles := PrecompiledContractsHomestead
        if evm.ChainConfig().IsByzantium(evm.BlockNumber) {
            precompiles = PrecompiledContractsByzantium
        }
        // 判断此地址若不是预编译合约地址 且区块号符合eip158提议 且 value值为0 ，那么简单返回
        if precompiles[addr] == nil && evm.ChainConfig().IsEIP158(evm.BlockNumber) && value.Sign()
== 0 {
            // Calling a non existing account, don't do anything, but ping the tracer
            if evm.vmConfig.Debug && evm.depth == 0 {
                evm.vmConfig.Tracer.CaptureStart(caller.Address(), addr, false, input, gas, value)
                evm.vmConfig.Tracer.CaptureEnd(ret, 0, 0, nil)
            }
            return nil, gas, nil
        }
        // 若上面条件不成立，那么创建此账户
        evm.StateDB.CreateAccount(addr)
```

```
    }
    // 先将以太币转账操作执行 call --> to ( addr )
    evm.Transfer(evm.StateDB, caller.Address(), to.Address(), value)
    // Initialise a new contract and set the code that is to be used by the EVM.
    // The contract is a scoped environment for this execution context only.
    // 创建contract结构体对call、to、value、gas等信息包装成contract结构体方便虚拟机执行
    contract := NewContract(caller, to, value, gas)
    // 下面将addr、codehash、code赋值给contract的成员属性
    contract.SetCallCode(&addr, evm.StateDB.GetCodeHash(addr), evm.StateDB.GetCode(addr))

    // Even if the account has no code, we need to continue because it might be a precompile
    start := time.Now()
    // debug模式下捕获一次evm执行交易的开始结束时间
    // Capture the tracer start/end events in debug mode
    if evm.vmConfig.Debug && evm.depth == 0 {
        evm.vmConfig.Tracer.CaptureStart(caller.Address(), addr, false, input, gas, value)

        defer func() { // Lazy evaluation of the parameters
            evm.vmConfig.Tracer.CaptureEnd(ret, gas-contract.Gas, time.Since(start), err)
        }()
    }
    // 下面调用run方法执行本次tx
    ret, err = run(evm, contract, input, false)

    // When an error was returned by the EVM or when setting the creation code
    // above we revert to the snapshot and consume any gas remaining. Additionally
    // when we're in homestead this also counts for code storage gas errors.
     // 若错误不为空，那么进行revert回滚
    if err != nil {
        evm.StateDB.RevertToSnapshot(snapshot)
        if err != errExecutionReverted {
            contract.UseGas(contract.Gas)
        }
    }
    // 返回执行结果，剩余gas，错误信息
    return ret, contract.Gas, err
}
// run()方法分析
// run runs the given contract and takes care of running precompiles with a fallback to the byte
code interpreter.
func run(evm *EVM, contract *Contract, input []byte, readOnly bool) ([]byte, error) {
    // 若contract的addr不为空，那么判断此addr是不是预编译合约，若是那么执行RunPrecompileContract方法
    if contract.CodeAddr != nil {
        precompiles := PrecompiledContractsHomestead
        if evm.ChainConfig().IsByzantium(evm.BlockNumber) {
            // 此处为预编译合约
            precompiles = PrecompiledContractsByzantium
        }
        if p := precompiles[*contract.CodeAddr]; p != nil {
            return RunPrecompiledContract(p, input, contract)
        }
    }
    // 调用evm的解释器执行contract
    for _, interpreter := range evm.interpreters {
        // CanRun()直接返回true
        if interpreter.CanRun(contract.Code) {
            if evm.interpreter != interpreter {
```

```
                // Ensure that the interpreter pointer is set back
                // to its current value upon return.
                defer func(i Interpreter) {
                    evm.interpreter = i
                }(evm.interpreter)
                evm.interpreter = interpreter
            }
            return interpreter.Run(contract, input, readOnly)
        }
    }
    return nil, ErrNoCompatibleInterpreter
}

//interpreter.Run()方法分析
// Run loops and evaluates the contract's code with the given input data and returns
// the return byte-slice and an error if one occurred.
// 下面短话主要说出了errExecutionReverted错误以外，其他错误将消耗所有gas
// It's important to note that any errors returned by the interpreter should be
// considered a revert-and-consume-all-gas operation except for
// errExecutionReverted which means revert-and-keep-gas-left.
func (in *EVMInterpreter) Run(contract *Contract, input []byte, readOnly bool) (ret []byte, err
error) {

    if in.intPool == nil {
        // 若intPool为空，那么尝试从intPool的pool的顶端面拿一个出来用，intPool的pool为空的情况下创建一
个新的intPool，intPool底层是一个栈结构且长度不超过256
        // 在本次运行结束以后将intPool放入intPool的pool中且将解释器的intPool置空
        // intPool的pool容量为25
        in.intPool = poolOfIntPools.get()
        defer func() {
            poolOfIntPools.put(in.intPool)
            in.intPool = nil
        }()
    }
    // 以太坊虚拟机调用深度计数
    // Increment the call depth which is restricted to 1024
    in.evm.depth++
    defer func() { in.evm.depth-- }()
    // 如果此次执行被标记为只读模式那么将解释器的模式也改成只读模式，在本次执行完成以后恢复解释器的执行模式
为非只读模式
    // Make sure the readOnly is only set if we aren't in readOnly yet.
    // This makes also sure that the readOnly flag isn't removed for child calls.
    if readOnly && !in.readOnly {
        in.readOnly = true
        defer func() { in.readOnly = false }()
    }
    // 重设返回数据为nil
    // Reset the previous call's return data. It's unimportant to preserve the old buffer
    // as every returning call will return new data anyway.
    in.returnData = nil
    // 若contract.Code长度为0，比如调用了普通转以太币的操作而不是合约，那么直接返回
    // Don't bother with the execution if there's no code.
    if len(contract.Code) == 0 {
        return nil, nil
    }

    var (
```

```
        op    OpCode        // current opcode 当前操作码
        mem   = NewMemory() // bound memory 创建内存，用代码实现的内存操作
        stack = newstack()  // local stack 创建一个栈，深度为1024
        // For optimisation reason we're using uint64 as the program counter.
        // It's theoretically possible to go above 2^64. The YP defines the PC
        // to be uint256. Practically much less so feasible.
        pc   = uint64(0) // program counter 程序计数器
        cost uint64 // 花费计数
        // 以下主要为了日志方面的记录
        // copies used by tracer
        pcCopy  uint64 // needed for the deferred Tracer
        gasCopy uint64 // for Tracer to log gas remaining before execution
        logged  bool   // deferred Tracer should ignore already logged steps
    )
     // 将input赋值给contract.input
    contract.Input = input
    // 执行结束以后将栈的数据放入intPool中
    // Reclaim the stack as an int pool when the execution stops
    defer func() { in.intPool.put(stack.data...) }()
    // 如果是debug模式，那么对相应的需要log的数据做记录
    if in.cfg.Debug {
        defer func() {
            if err != nil {
                if !logged {
                    in.cfg.Tracer.CaptureState(in.evm, pcCopy, op, gasCopy, cost, mem, stack,
contract, in.evm.depth, err)
                } else {
                    in.cfg.Tracer.CaptureFault(in.evm, pcCopy, op, gasCopy, cost, mem, stack,
contract, in.evm.depth, err)
                }
            }
        }()
    }
    // The Interpreter main run loop (contextual). This loop runs until either an
    // explicit STOP, RETURN or SELFDESTRUCT is executed, an error occurred during
    // the execution of one of the operations or until the done flag is set by the
    // parent context.
    // 虚拟机中断标志位为0的时候继续进行for循环
    for atomic.LoadInt32(&in.evm.abort) == 0 {
        if in.cfg.Debug {
            // Capture pre-execution values for tracing.
            logged, pcCopy, gasCopy = false, pc, contract.Gas
        }

        // Get the operation from the jump table and validate the stack to ensure there are
        // enough stack items available to perform the operation.
        // 从code里面取出字节码
        op = contract.GetOp(pc)
        // 从jumpTable中获取字节码对应的operation
        operation := in.cfg.JumpTable[op]
         // 如果此operation非法，返回错误
        if !operation.valid {
            return nil, fmt.Errorf("invalid opcode 0x%x", int(op))
        }
        // 判断栈的最大深度是否超过1024，超过则返回
        if err := operation.validateStack(stack); err != nil {
            return nil, err
```

```
        }
        // 对本字节码做一些验证，主要防止evm在readOnly模式下执行了修改状态的字节码
        // If the operation is valid, enforce and write restrictions
        if err := in.enforceRestrictions(op, operation, stack); err != nil {
            return nil, err
        }

        var memorySize uint64
        // calculate the new memory size and expand the memory to fit
        // the operation
        if operation.memorySize != nil {
            memSize, overflow := bigUint64(operation.memorySize(stack))
            if overflow {
                return nil, errGasUintOverflow
            }
            // memory is expanded in words of 32 bytes. Gas
            // is also calculated in words.
            if memorySize, overflow = math.SafeMul(toWordSize(memSize), 32); overflow {
                return nil, errGasUintOverflow
            }
        }
        // consume the gas and return an error if not enough gas is available.
        // cost is explicitly set so that the capture state defer method can get the proper cost
        // 开始计算本次opCode所消耗的gas
        cost, err = operation.gasCost(in.gasTable, in.evm, contract, stack, mem, memorySize)
         // UseGas会将c.gas - cost  判断剩余gas是不是能够执行此次操作
        if err != nil || !contract.UseGas(cost) {
            return nil, ErrOutOfGas
        }
        // 分配内存
        if memorySize > 0 {
            mem.Resize(memorySize)
        }

        if in.cfg.Debug {
            in.cfg.Tracer.CaptureState(in.evm, pc, op, gasCopy, cost, mem, stack, contract,
in.evm.depth, err)
            logged = true
        }
        // 在此实际执行了相关操作码，为了方便在这里用ADD字节码举例
        //func opAdd(pc *uint64, interpreter *EVMInterpreter, contract *Contract, memory *Memory,
stack *Stack) ([]byte, error) {
        //x, y := stack.pop(), stack.peek()
        //math.U256(y.Add(x, y))

        //interpreter.intPool.put(x)
        //return nil, nil
        //}
        // execute the operation
        res, err := operation.execute(&pc, in, contract, mem, stack)
        // verifyPool is a build flag. Pool verification makes sure the integrity
        // of the integer pool by comparing values to a default value.
        if verifyPool {
            verifyIntegerPool(in.intPool)
        }
         // 如果字节码中的returns标志位为true那么将字节码执行的结果放在解释器的returnData中
        // if the operation clears the return data (e.g. it has returning data)
```

```
        // set the last return to the result of the operation.
        if operation.returns {
            in.returnData = res
        }

        switch {
         // 若执行完操作码后出现错误，那么返回错误
        case err != nil:
            return nil, err
         // 若操作码回滚标记为true，那么返回回滚错误
        case operation.reverts:
            return res, errExecutionReverted
         // 若操作码停止标记位为true，那么返回执行结果
        case operation.halts:
            return res, nil
         // 若操作码跳转标记位为false，那么程序计数器自增，进行下次循环
        case !operation.jumps:
            pc++
        }
    }
    return nil, nil
}
```

以上主要是call()函数执行时的逻辑，其中[256]operation个操作字节码的生成在go-ethereum/core/vm/jump_table.go中，其中每个operation的execute函数的实现在go-ethereum/core/vm/instructions.go

下面分析create()函数的逻辑

2.2.3 evm.Create()

```
// Create函数主要用来创建合约，Create2是支持了新的eip的函数，这里暂时不分析
// Create creates a new contract using code as deployment code.
func (evm *EVM) Create(caller ContractRef, code []byte, gas uint64, value *big.Int) (ret []byte,
contractAddr common.Address, leftOverGas uint64, err error) {
    // 计算合约地址，使用调用者地址和nonce创建合约地址
    contractAddr = crypto.CreateAddress(caller.Address(), evm.StateDB.GetNonce(caller.Address()))
    return evm.create(caller, &codeAndHash{code: code}, gas, value, contractAddr)
}

// create creates a new contract using code as deployment code.
func (evm *EVM) create(caller ContractRef, codeAndHash *codeAndHash, gas uint64, value *big.Int,
address common.Address) ([]byte, common.Address, uint64, error) {
    // Depth check execution. Fail if we're trying to execute above the
    // limit.
    //判断evm调用的深度是否大于最大调用限制
    if evm.depth > int(params.CallCreateDepth) {
        return nil, common.Address{}, gas, ErrDepth
    }
    // 若value大于0 则判断账户的余额是否充足
    if !evm.CanTransfer(evm.StateDB, caller.Address(), value) {
        return nil, common.Address{}, gas, ErrInsufficientBalance
    }
    // 获取调用者的nonce值并将调用者的nonce更新为+1以后的值
    nonce := evm.StateDB.GetNonce(caller.Address())
    evm.StateDB.SetNonce(caller.Address(), nonce+1)
    // 确保此地址没有合约已经存在，一般来说一个stateObject只能有一个code存在
```

```
    // Ensure there's no existing contract already at the designated address
    contractHash := evm.StateDB.GetCodeHash(address)
     // 如果新生成的合约地址的Nonce不为0或者使用此地址获取的codehash值不为初始值，返回合约地址冲突错误
     // 下面主要检查新生成的合约地址是不是个新地址，是不是被使用过
    if evm.StateDB.GetNonce(address) != 0 || (contractHash != (common.Hash{}) && contractHash !=
emptyCodeHash) {
        return nil, common.Address{}, 0, ErrContractAddressCollision
    }
    // 获取当前statedb的快照
    // Create a new account on the state
    snapshot := evm.StateDB.Snapshot()
    // 在statedb中创建此账户
    evm.StateDB.CreateAccount(address)
     // 如果区块号的范围是在eip158的范围内，那么将合约地址的nonce设置为1
    if evm.ChainConfig().IsEIP158(evm.BlockNumber) {
        evm.StateDB.SetNonce(address, 1)
    }
    // 先执行以太币转账
    evm.Transfer(evm.StateDB, caller.Address(), address, value)

    // initialise a new contract and set the code that is to be used by the
    // EVM. The contract is a scoped environment for this execution context
    // only.
    //和调用call一样，将数据包装在contract里面
    contract := NewContract(caller, AccountRef(address), value, gas)
    contract.SetCodeOptionalHash(&address, codeAndHash)
    // 如果配置了非递归且evm.depth大于0，那么说明递归调用了，直接返回
    if evm.vmConfig.NoRecursion && evm.depth > 0 {
        return nil, address, gas, nil
    }

    if evm.vmConfig.Debug && evm.depth == 0 {
        evm.vmConfig.Tracer.CaptureStart(caller.Address(), address, true, codeAndHash.code, gas,
value)
    }
    start := time.Now()
    // 具体执行
    ret, err := run(evm, contract, nil, false)
    // 判断是否超出最大代码大小限制
    // check whether the max code size has been exceeded
    maxCodeSizeExceeded := evm.ChainConfig().IsEIP158(evm.BlockNumber) && len(ret) >
params.MaxCodeSize
    // if the contract creation ran successfully and no errors were returned
    // calculate the gas required to store the code. If the code could not
    // be stored due to not enough gas set an error and let it be handled
    // by the error checking condition below.
     // 如果执行成功了，且创建数据花费的gas数量没超过账户限制，那么在statedb中存入此账户的代码
    if err == nil && !maxCodeSizeExceeded {
        createDataGas := uint64(len(ret)) * params.CreateDataGas
        if contract.UseGas(createDataGas) {
            evm.StateDB.SetCode(address, ret)
        } else {
            err = ErrCodeStoreOutOfGas
        }
    }

    // When an error was returned by the EVM or when setting the creation code
```

```
    // above we revert to the snapshot and consume any gas remaining. Additionally
    // when we're in homestead this also counts for code storage gas errors.
    // 如果代码大小超出限制 或者 错误不为空且区块号是homestead版本或者错误不等于code存储费用超出账户gas错误 那么回滚到前面的状态 且消耗完本次账户提供的gas
    if maxCodeSizeExceeded || (err != nil && (evm.ChainConfig().IsHomestead(evm.BlockNumber) || err != ErrCodeStoreOutOfGas)) {
        evm.StateDB.RevertToSnapshot(snapshot)
        if err != errExecutionReverted {
            contract.UseGas(contract.Gas)
        }
    }
    // Assign err if contract code size exceeds the max while the err is still empty.
    if maxCodeSizeExceeded && err == nil {
        err = errMaxCodeSizeExceeded
    }
    if evm.vmConfig.Debug && evm.depth == 0 {
        evm.vmConfig.Tracer.CaptureEnd(ret, gas-contract.Gas, time.Since(start), err)
    }
    return ret, address, contract.Gas, err

}
```

以上部分粗略的分析了以太坊交易如何在以太坊虚拟机中执行的过程。期间还有许多细节需要仔细去理解。

另外 jump_table.go和instractions.go是虚拟机字节码的具体实现，请自行翻阅

在vm/evm.go下面还有CallCode、DelegateCall、StaticCall方法不允许直接调用，而是在字节码层面间接调用，需要了解的请自行翻阅。逻辑和call类似