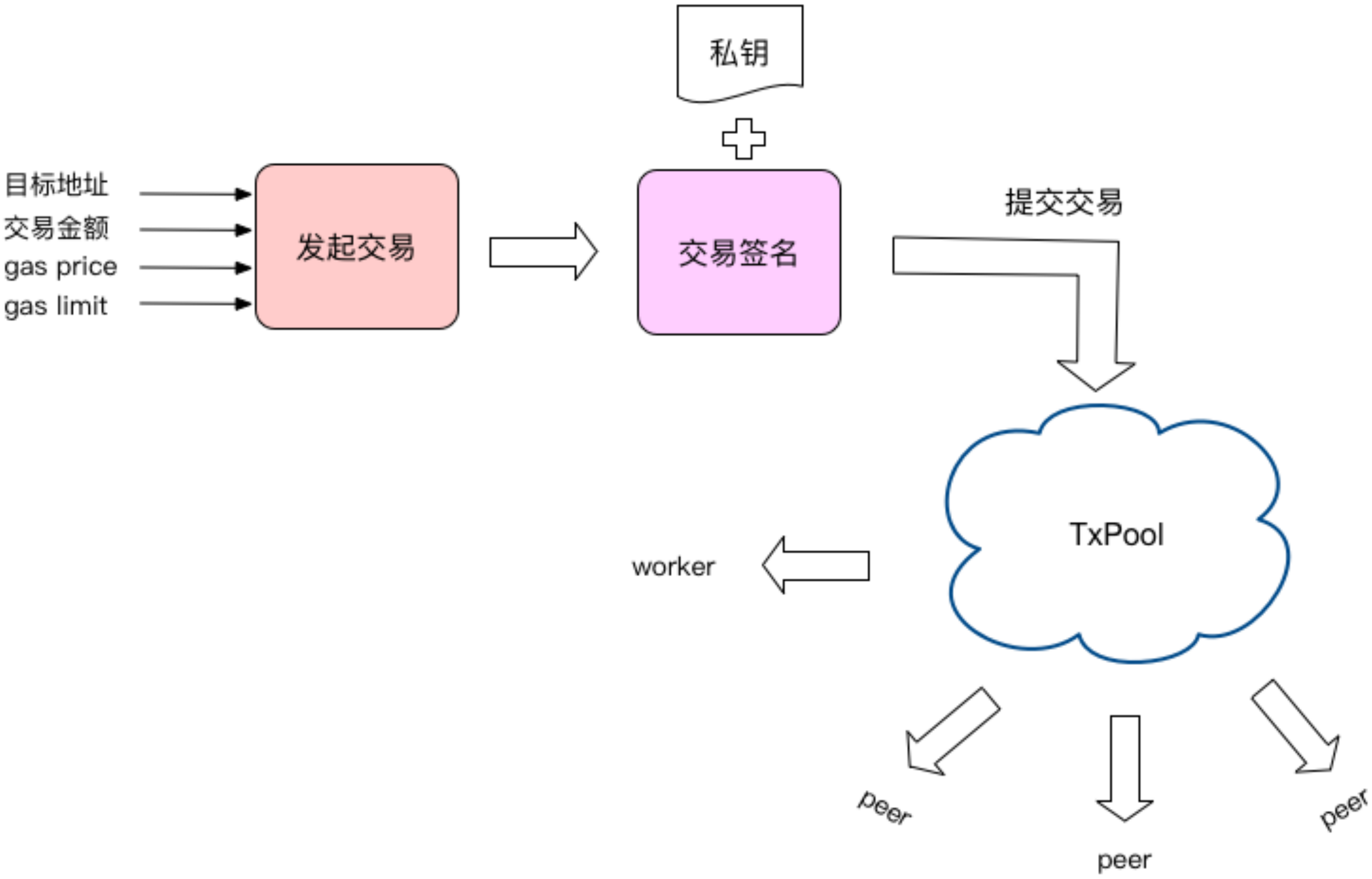
以太坊最重要角色之以太坊交易源码解析1

以太坊交易基本流程：



完整流程分为以下几个步骤：

- 发起交易：指定目标地址和交易金额，以及需要的gas/gaslimit
- 交易签名：使用账户私钥对交易进行签名
- 提交交易：把交易加入到交易缓冲池txpool中（会先对交易签名进行验证）
- 广播交易：通知EVM执行，同时把交易信息广播给其他结点

一、发起交易

用户通过JSON RPC发起 `eth_sendTransaction` 请求，最终会调用 `PublicTransactionPoolAPI` 的 `SendTransaction` 实现，
首先根据from地址查找到对应的wallet，检查一下参数值，
* 通过SendTxArgs.toTransaction()创建交易

* 通过Wallet.SignTx()对交易进行签名
* 通过submitTransaction()提交交易

```
//代码位于 `internal/ethapi/api.go`

func (s *PrivateAccountAPI) SendTransaction(ctx context.Context, args SendTxArgs, passwd string) (common.Hash, error) {
    if args.Nonce == nil {
        // Hold the addresse's mutex around signing to prevent concurrent assignment of
        // the same nonce to multiple accounts.
        s.nonceLock.LockAddr(args.From)
        defer s.nonceLock.UnlockAddr(args.From)
    }
    signed, err := s.signTransaction(ctx, args, passwd)
    if err != nil {
        return common.Hash{}, err
    }
    return submitTransaction(ctx, s.b, signed)
}
```

交易签名主要实现在 `signTransaction`，主要功能：

`toTransaction()`：创建交易
`wallet.SignTxWithPassphrase(account, passwd, tx, chainID)`：对交易进行签名

```
func (s *PrivateAccountAPI) signTransaction(ctx context.Context, args SendTxArgs, passwd string) (*types.Transaction, error) {
    // Look up the wallet containing the requested signer
    account := accounts.Account{Address: args.From}
    wallet, err := s.am.Find(account)
    if err != nil {
        return nil, err
    }
    // Set some sanity defaults and terminate on failure
    if err := args.setDefaults(ctx, s.b); err != nil {
        return nil, err
    }
    // Assemble the transaction and sign with the wallet
    tx := args.toTransaction()

    var chainID *big.Int
    if config := s.b.ChainConfig(); config.IsEIP155(s.b.CurrentBlock().Number()) {
        chainID = config.ChainId
    }
    return wallet.SignTxWithPassphrase(account, passwd, tx, chainID)
}
```

二、创建交易

tx := args.toTransaction() 创建交易

先看一下SendTxArgs类型的定义：

```
// 代码 internal/ethapi/api.go

// SendTxArgs represents the arguments to sumbit a new transaction into the transaction pool.
type SendTxArgs struct {
    From     common.Address  `json:"from"`
    To       *common.Address `json:"to"`
    Gas      *hexutil.Uint64 `json:"gas"`
    GasPrice *hexutil.Big    `json:"gasPrice"`
    Value    *hexutil.Big    `json:"value"`
    Nonce    *hexutil.Uint64 `json:"nonce"`
    // We accept "data" and "input" for backwards-compatibility reasons. "input" is the
    // newer name and should be preferred by clients.
    Data  *hexutil.Bytes `json:"data"`
    Input *hexutil.Bytes `json:"input"`
}
```

可以看到是和JSON字段相应的，包括了地址、gas、金额这些交易信息，nonce是一个随账户交易次数自增的数字，一般会自动填充。交易还可以携带一些额外数据，存放在data或者input字段中，推荐用input，data是为了向后兼容。

toTransaction()函数：

```
// 代码 internal/ethapi/api.go

func (args *SendTxArgs) toTransaction() *types.Transaction {
    var input []byte
    if args.Data != nil {
        input = *args.Data
    } else if args.Input != nil {
        input = *args.Input
    }
    if args.To == nil {
        return types.NewContractCreation(uint64(*args.Nonce), (*big.Int)(args.Value
), uint64(*args.Gas), (*big.Int)(args.GasPrice), input)
    }
    return types.NewTransaction(uint64(*args.Nonce), *args.To, (*big.Int)(args.Valu
e), uint64(*args.Gas), (*big.Int)(args.GasPrice), input)
}
```

可以看到，如果目标地址为空的话，表示这是一个创建智能合约的交易，调用NewContractCreation()。否则说明这是一个普通交易，调用NewTransaction()。不管调用哪个，最终都会生成一个Transaction实例，我们看一下Transaction类型的定义：

```
// 代码位于core/types/transaction.go

type Transaction struct {
    data txdata
    // caches
    hash atomic.Value
    size atomic.Value
    from atomic.Value
}

type txdata struct {
    AccountNonce uint64          `json:"nonce"    gencodec:"required"`
    Price        *big.Int        `json:"gasPrice" gencodec:"required"`
    GasLimit     uint64          `json:"gas"      gencodec:"required"`
    Recipient    *common.Address `json:"to"       rlp:"nil"` // nil means contract
creation
    Amount       *big.Int        `json:"value"    gencodec:"required"`
    Payload      []byte          `json:"input"    gencodec:"required"`

    // Signature values
    V *big.Int `json:"v" gencodec:"required"`
    R *big.Int `json:"r" gencodec:"required"`
    S *big.Int `json:"s" gencodec:"required"`

    // This is only used when marshaling to JSON.
    Hash *common.Hash `json:"hash" rlp:"-"`
}
```

三、交易签名

wallet.SignTxWithPassphrase 代码

```
// accounts/keystore/keystore_wallet.go
// SignTxWithPassphrase implements accounts.Wallet, attempting to sign the given
// transaction with the given account using passphrase as extra authentication.
func (w *keystoreWallet) SignTxWithPassphrase(account accounts.Account, passphrase
string, tx *types.Transaction, chainID *big.Int) (*types.Transaction, error) {
    // Make sure the requested account is contained within
    if account.Address != w.account.Address {
        return nil, accounts.ErrUnknownAccount
    }
    if account.URL != (accounts.URL{}) && account.URL != w.account.URL {
        return nil, accounts.ErrUnknownAccount
    }
    // Account seems valid, request the keystore to sign
    return w.keystore.SignTxWithPassphrase(account, passphrase, tx, chainID)
}
```

w.keystore.SignTxWithPassphrase(account, passphrase, tx, chainID) 代码：
主要就是通过 `SignTx` 进行签名。

```
// 代码 accounts/keystore/keystore.go

func (ks *KeyStore) SignTxWithPassphrase(a accounts.Account, passphrase string, tx
*types.Transaction, chainID *big.Int) (*types.Transaction, error) {
    _, key, err := ks.getDecryptedKey(a, passphrase)
    if err != nil {
        return nil, err
    }
    defer zeroKey(key.PrivateKey)

    // Depending on the presence of the chain ID, sign with EIP155 or homestead
    if chainID != nil {
        return types.SignTx(tx, types.NewEIP155Signer(chainID), key.PrivateKey)
    }
    return types.SignTx(tx, types.HomesteadSigner{}, key.PrivateKey)
}
```

这里会首先判断账户是否已经解锁，如果已经解锁的话就可以获取它的私钥。

然后创建签名器，如果要符合EIP155规范的话，需要把chainID传进去，也就是我们的“--networkid”命令行参数。

最后调用一个全局函数SignTx()完成签名：

```
代码位于core/types/transaction_signing.go：

// SignTx signs the transaction using the given signer and private key
```

```
func SignTx(tx *Transaction, s Signer, prv *ecdsa.PrivateKey) (*Transaction, error)
 {
    h := s.Hash(tx)
    sig, err := crypto.Sign(h[:], prv)
    if err != nil {
        return nil, err
    }
    return tx.WithSignature(s, sig)
}
```

主要分为3个步骤：

- 生成交易的hash值
- 根据hash值和私钥生成签名
- 把签名数据填充到Transaction实例中

生成交易的hash值

以EIP155Signer为例，代码如下：

```
func (s EIP155Signer) Hash(tx *Transaction) common.Hash {
    return rlpHash([]interface{}{
        tx.data.AccountNonce,
        tx.data.Price,
        tx.data.GasLimit,
        tx.data.Recipient,
        tx.data.Amount,
        tx.data.Payload,
        s.chainId, uint(0), uint(0),
    })
}

func rlpHash(x interface{}) (h common.Hash) {
    hw := sha3.NewKeccak256()
    rlp.Encode(hw, x)
    hw.Sum(h[:0])
    return h
}
```

可以看到，先用SHA3-256生成hash值，然后再进行RLP编码。RLP是一种数据序列化方法。

根据hash值和私钥生成签名-crypto.Sign()

```
// 代码位于crypto/signature_cgo.go：
```

```
func Sign(hash []byte, prv *ecdsa.PrivateKey) (sig []byte, err error) {
    if len(hash) != 32 {
        return nil, fmt.Errorf("hash is required to be exactly 32 bytes (%d)", len(
hash))
    }
    seckey := math.PaddedBigBytes(prv.D, prv.Params().BitSize/8)
    defer zeroBytes(seckey)
    return secp256k1.Sign(hash, seckey)
}
```

这里是通过ECDSA算法生成签名数据。最终会返回的签名是一个字节数组，按R / S / V的顺序排列。

填充签名数据 - WithSignature

```
//代码位于 core/types/transaction.go

func (tx *Transaction) WithSignature(signer Signer, sig []byte) (*Transaction, erro
r) {
    r, s, v, err := signer.SignatureValues(tx, sig)
    if err != nil {
        return nil, err
    }
    cpy := &Transaction{data: tx.data}
    cpy.data.R, cpy.data.S, cpy.data.V = r, s, v
    return cpy, nil
}
```

生成的签名数据是字节数组类型，需要通过signer.SignatureValues()函数转换成3个big.Int类型的数据，然后填充到Transaction结构的R / S / V字段上

四、提交交易

签名完成以后，就需要调用 `submitTransaction()` 函数提交到交易缓冲池txpool中。

先看下TxPool中的几个重要字段：

```
// 代码 core/tx_pool.go

type TxPool struct {
    config       TxPoolConfig
    chainconfig  *params.ChainConfig
    chain        blockChain
    gasPrice     *big.Int
    txFeed       event.Feed
    scope        event.SubscriptionScope
    chainHeadCh  chan ChainHeadEvent
    chainHeadSub event.Subscription
    signer       types.Signer
    mu           sync.RWMutex

    currentState  *state.StateDB      // Current state in the blockchain head
    pendingState  *state.ManagedState // Pending state tracking virtual nonces
    currentMaxGas uint64              // Current gas limit for transaction caps

    locals  *accountSet // Set of local transaction to exempt from eviction rules
    journal *txJournal  // Journal of local transaction to back up to disk

    pending map[common.Address]*txList   // All currently processable transactions
    queue   map[common.Address]*txList   // Queued but non-processable transactions
    beats   map[common.Address]time.Time // Last heartbeat from each known account
    all     *txLookup                    // All transactions to allow lookups
    priced  *txPricedList                // All transactions sorted by price

    wg sync.WaitGroup // for shutdown sync

    homestead bool
}
```

pending字段中包含了当前所有可被处理的交易列表，而**queue字段**中包含了所有不可被处理、也就是新加入进来的交易。下面查看一下**pending字段** 的txList的结构：

```
type txList struct {
    strict bool         // Whether nonces are strictly continuous or not
    txs    *txSortedMap // Heap indexed sorted hash map of the transactions

    costcap *big.Int // Price of the highest costing transaction (reset only if exceeds balance)
    gascap  uint64   // Gas limit of the highest spending transaction (reset only if exceeds block limit)
}
```

txList内部包含一个txSortedMap结构，实现按nonce排序，其内部维护了两张表：

- 一张是包含了所有Transaction的map，key是Transaction的nonce值。之前提到过，这个nonce是随着账户的交易次数自增的一个数字，所以越新的交易，nonce值越高。
- 还有一张表是一个数组，包含了所有nonce值，其内部是进行过堆排序的（小顶堆），nonce值按照从大到小排列。每次调用heap.Pop()时会取出最小的nonce值，也就是最老的

交易。

all字段 中包含了所有的交易列表，以交易的hash作为key。

priced字段 则是把all中的交易列表按照gas price从大到小排列，如果gas price一样，则按照交易的nonce值从小到大排列。最终的目标是每次取出gas price最大、nonce最小的交易。

我们提交交易的目标是：先把交易放入queue中记录在案，然后再从queue中选一部分放入pending中进行处理。如果发现txpool满了，则依据priced中的排序，剔除低油价的交易。

txpool的默认配置：

```
var DefaultTxPoolConfig = TxPoolConfig{
    Journal:   "transactions.rlp",
    Rejournal: time.Hour,

    PriceLimit: 1,
    PriceBump:  10,

    AccountSlots: 16,
    GlobalSlots:  4096,
    AccountQueue: 64,
    GlobalQueue:  1024,

    Lifetime: 3 * time.Hour,
}
```

- GlobalSlots：pending列表的最大长度，默认4096笔
- AccountSlots：pending中每个账户存储的交易数的阈值，超过这个数量可能会被认为是垃圾交易或者是攻击者，多余交易可能被丢弃
- GlobalQueue：queue列表的最大长度，默认1024笔
- AccountQueue：queue中每个账户允许存储的最大交易数，超过会被丢弃，默认64笔
- PriceLimit：允许进入txpool的最低gas price，默认1 Gwei
- PriceBump：如果出现两个nonce相同的交易，gas price的差值超过该阈值则用新交易替换老交易

现在我们分析submitTransaction()函数：

```
//代码位于 `internal/ethapi/api.go`

func submitTransaction(ctx context.Context, b Backend, tx *types.Transaction) (common.Hash, error) {
    if err := b.SendTx(ctx, tx); err != nil {
        return common.Hash{}, err
    }
    if tx.To() == nil {
        signer := types.MakeSigner(b.ChainConfig(), b.CurrentBlock().Number())
        from, err := types.Sender(signer, tx)
        if err != nil {
            return common.Hash{}, err
        }
        addr := crypto.CreateAddress(from, tx.Nonce())
        log.Info("Submitted contract creation", "fullhash", tx.Hash().Hex(), "contract", addr.Hex())
    } else {
        log.Info("Submitted transaction", "fullhash", tx.Hash().Hex(), "recipient", tx.To())
    }
    return tx.Hash(), nil
}
```

这里有一个Backend参数，是在eth Service初始化时创建的，具体实现在EthApiBackend中，代码位于eth/api_backend.go。可以看到，这里先调用了SendTx()函数提交交易，然后如果发现目标地址为空，表明这是一个创建智能合约的交易，会创建合约地址。

提交交易到txpool

```
//代码 eth/api_backend.go

func (b *EthAPIBackend) SendTx(ctx context.Context, signedTx *types.Transaction) error {
    return b.eth.txPool.AddLocal(signedTx)
}
```

继续跟踪TxPool的AddLocal()函数：

```
// 代码位于 core/tx_pool.go

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()

    // Try to inject the transaction and update any state
    replace, err := pool.add(tx, local)
    if err != nil {
        return err
    }
    // 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
}
```

这里有两个主要函数：add()和promoteExecuteables()。
add()会判断是否应该把当前交易加入到queue列表中，promoteExecuteables()则会从queue中选取一些交易放入pending列表中等待执行。下面分别讨论这两个函数。

TxPool.add()

```
// 代码位于 core/tx_pool.go

func (pool *TxPool) add(tx *types.Transaction, local bool) (bool, error) {
    // If the transaction is already known, discard it
    hash := tx.Hash()
    if pool.all.Get(hash) != nil {
        log.Trace("Discarding already known transaction", "hash", hash)
        return false, fmt.Errorf("known transaction: %x", hash)
    }
    // If the transaction fails basic validation, discard it
    if err := pool.validateTx(tx, local); err != nil {
        log.Trace("Discarding invalid transaction", "hash", hash, "err", err)
        invalidTxCounter.Inc(1)
        return false, err
    }
    // If the transaction pool is full, discard underpriced transactions
    if uint64(pool.all.Count()) >= pool.config.GlobalSlots+pool.config.GlobalQueue
{
        // If the new transaction is underpriced, don't accept it
        if !local && pool.priced.Underpriced(tx, pool.locals) {
            log.Trace("Discarding underpriced transaction", "hash", hash, "price",
tx.GasPrice())
            underpricedTxCounter.Inc(1)
            return false, ErrUnderpriced
        }
        // New transaction is better than our worse ones, make room for it
        drop := pool.priced.Discard(pool.all.Count()-int(pool.config.GlobalSlots+po
ol.config.GlobalQueue-1), pool.locals)
```

```
        for _, tx := range drop {
            log.Trace("Discarding freshly underpriced transaction", "hash", tx.Hash(), "price", tx.GasPrice())
            underpricedTxCounter.Inc(1)
            pool.removeTx(tx.Hash(), false)
        }
    }
    // If the transaction is replacing an already pending one, do directly
    from, _ := types.Sender(pool.signer, tx) // already validated
    if list := pool.pending[from]; list != nil && list.Overlaps(tx) {
        // Nonce already pending, check if required price bump is met
        inserted, old := list.Add(tx, pool.config.PriceBump)
        if !inserted {
            pendingDiscardCounter.Inc(1)
            return false, ErrReplaceUnderpriced
        }
        // New transaction is better, replace old one
        if old != nil {
            pool.all.Remove(old.Hash())
            pool.priced.Removed()
            pendingReplaceCounter.Inc(1)
        }
        pool.all.Add(tx)
        pool.priced.Put(tx)
        pool.journalTx(from, tx)

        log.Trace("Pooled new executable transaction", "hash", hash, "from", from, "to", tx.To())

        // We've directly injected a replacement transaction, notify subsystems
        go pool.txFeed.Send(NewTxsEvent{types.Transactions{tx}})

        return old != nil, nil
    }
    // New transaction isn't replacing a pending one, push into queue
    replace, err := pool.enqueueTx(hash, tx)
    if err != nil {
        return false, err
    }
    // Mark local addresses and journal local transactions
    if local {
        pool.locals.add(from)
    }
    pool.journalTx(from, tx)

    log.Trace("Pooled new future transaction", "hash", hash, "from", from, "to", tx.To())
    return replace, nil
```

```
}
```

我们分成一段一段的来分析：

```
hash := tx.Hash()
    if pool.all.Get(hash) != nil {
        log.Trace("Discarding already known transaction", "hash", hash)
        return false, fmt.Errorf("known transaction: %x", hash)
    }
```

这一段是先计算交易的hash值，然后判断是不是已经在txpool 中，在的话就直接退出。

```
// If the transaction fails basic validation, discard it
    if err := pool.validateTx(tx, local); err != nil {
        log.Trace("Discarding invalid transaction", "hash", hash, "err", err)
        invalidTxCounter.Inc(1)
        return false, err
    }
```

查看 `pool.validateTx(tx, local)` 代码

```
// 代码位于 core/tx_pool.go

func (pool *TxPool) validateTx(tx *types.Transaction, local bool) error {
    // Heuristic limit, reject transactions over 32KB to prevent DOS attacks
    if tx.Size() > 32*1024 {
        return ErrOversizedData
    }
    // Transactions can't be negative. This may never happen using RLP decoded
    // transactions but may occur if you create a transaction using the RPC.
    if tx.Value().Sign() < 0 {
        return ErrNegativeValue
    }
    // Ensure the transaction doesn't exceed the current block limit gas.
    if pool.currentMaxGas < tx.Gas() {
        return ErrGasLimit
    }
    // Make sure the transaction is signed properly
    from, err := types.Sender(pool.signer, tx)
    if err != nil {
        return ErrInvalidSender
    }
    // Drop non-local transactions under our own minimal accepted gas price
    local = local || pool.locals.contains(from) // account may be local even if the
```

```
 transaction arrived from the network
    if !local && pool.gasPrice.Cmp(tx.GasPrice()) > 0 {
        return ErrUnderpriced
    }
    // Ensure the transaction adheres to nonce ordering
    if pool.currentState.GetNonce(from) > tx.Nonce() {
        return ErrNonceTooLow
    }
    // Transactor should have enough funds to cover the costs
    // cost == V + GP * GL
    if pool.currentState.GetBalance(from).Cmp(tx.Cost()) < 0 {
        return ErrInsufficientFunds
    }
    intrGas, err := IntrinsicGas(tx.Data(), tx.To() == nil, pool.homestead)
    if err != nil {
        return err
    }
    if tx.Gas() < intrGas {
        return ErrIntrinsicGas
    }
    return nil
}
```

这一段是验证交易的有效性，主要进行以下几个方面的检查：

- 数据量必须<32KB
- 交易金额必须非负（>=0）
- 交易的gas limit必须低于block的gas limit
- 签名数据必须有效，能够解析出发送者地址
- 交易的gas price必须高于pool设定的最低gas price（除非是本地交易）
- 交易的nonce值必须高于当前链上该账户的nonce值（低于则说明这笔交易已经被打包过了）
- 当前账户余额必须大于`“交易金额 + gasprice * gaslimit”`
- 交易的gas limit必须大于对应数据量所需的最低gas水平

```
    if uint64(len(pool.all)) >= pool.config.GlobalSlots+pool.config.GlobalQueue {
        // If the new transaction is underpriced, don't accept it
        if !local && pool.priced.Underpriced(tx, pool.locals) {
            log.Trace("Discarding underpriced transaction", "hash", hash, "price",
tx.GasPrice())
            underpricedTxCounter.Inc(1)
            return false, ErrUnderpriced
        }
        // New transaction is better than our worse ones, make room for it
        drop := pool.priced.Discard(len(pool.all)-int(pool.config.GlobalSlots+pool.
config.GlobalQueue-1), pool.locals)
        for _, tx := range drop {
            log.Trace("Discarding freshly underpriced transaction", "hash", tx.Hash
(), "price", tx.GasPrice())
            underpricedTxCounter.Inc(1)
            pool.removeTx(tx.Hash(), false)
        }
    }
```

这一段是在当前txpool已满的情况下，剔除掉低油价的交易。还记得之前有个priced字段存储了按gas price以及nonce排序的交易列表吗？这里会先把当前交易的gas price和当前池中的最低价进行比较：

- 如果低于最低价，直接丢弃该交易返回
- 如果高于最低价，则从txpool中剔除一些低价的交易

```
// New transaction isn't replacing a pending one, push into queue
    replace, err := pool.enqueueTx(hash, tx)
    if err != nil {
        return false, err
    }
```

如果之前的那些检查都没有问题，就真正调用enqueueTx()函数把交易加入到queue列表中了。

```
// Mark local addresses and journal local transactions
    if local {
        pool.locals.add(from)
    }
    pool.journalTx(from, tx)
```

最后，如果发现这个账户是本地的，就把它加到一个白名单里，默认会保证本地交易优先被加到txpool中。