TxPool代码分析

1、首先先介绍一下相关数据结构

```
// TxPoolConfig are the configuration parameters of the transaction pool.
//1--交易池配置
type TxPoolConfig struct {
    Locals    []common.Address // Addresses that should be treated by default as local 本地账户地址存放
    NoLocals  bool             // Whether local transaction handling should be disabled 是否开启本地交易机制
    Journal   string           // Journal of local transactions to survive node restarts 本地交易存放路径
    Rejournal time.Duration    // Time interval to regenerate the local transaction journal 持久化本地交易的间隔

    PriceLimit uint64 // Minimum gas price to enforce for acceptance into the pool 最小gasPrice限制
    PriceBump  uint64 // Minimum price bump percentage to replace an already existing transaction (nonce) 价格超出比例，若想覆盖一笔交易的时候，若价格上涨比例达不到要求，那么不能覆盖
    // pending 和 queue 槽位限制
    AccountSlots uint64 // Number of executable transaction slots guaranteed per account 每个账户的可执行交易槽位限制
    GlobalSlots  uint64 // Maximum number of executable transaction slots for all accounts 全部账户最大可执行交易槽位
    AccountQueue uint64 // Maximum number of non-executable transaction slots permitted per account 单个账户不可执行的槽位限制
    GlobalQueue  uint64 // Maximum number of non-executable transaction slots for all accounts 全部账户最大非执行交易槽位限制
    //一个账户在queue中的交易可以存活的时间
    Lifetime time.Duration // Maximum amount of time non-executable transaction are queued
}

// 2--默认交易池配置
// DefaultTxPoolConfig contains the default configurations for the transaction
// pool.
var DefaultTxPoolConfig = TxPoolConfig{
    Journal:   "transactions.rlp", // 本地交易的存放文件
    Rejournal: time.Hour, //一般一小时进行一次重新存储

    PriceLimit: 1, //最小price限制默认值
    PriceBump:  10, //

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

    Lifetime: 3 * time.Hour, //3 小时生命周期
}
// 3--交易池数据结构
// TxPool contains all currently known transactions. Transactions
// enter the pool when they are received from the network or submitted
// locally. They exit the pool when they are included in the blockchain.
//
// The pool separates processable transactions (which can be applied to the
// current state) and future transactions. Transactions move between those
// two states over time as they are received and processed.
type TxPool struct {
    config       TxPoolConfig //交易池配置
    chainconfig  *params.ChainConfig // 区块链配置
    chain        blockChain // 定义的blockChain接口 对链的一些操作
    gasPrice     *big.Int
    txFeed       event.Feed // 时间流
    scope        event.SubscriptionScope //订阅范围
    chainHeadCh  chan ChainHeadEvent // chainHead事件通道
    chainHeadSub event.Subscription
    signer       types.Signer // 签名
    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
}
//4--交易列表
// txList is a "list" of transactions belonging to an account, sorted by account
// nonce. The same type can be used both for storing contiguous transactions for
// the executable/pending queue; and for storing gapped transactions for the non-
// executable/future queue, with minor behavioral changes.
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)
}

//5--
// txSortedMap is a nonce->transaction hash map with a heap based index to allow
// iterating over the contents in a nonce-incrementing way.
type txSortedMap struct {
    items map[uint64]*types.Transaction // Hash map storing the transaction data
    index *nonceHeap                    // Heap of nonces of all the stored transactions (non-
strict mode) nonceHeap是自己实现的一个最小堆
    cache types.Transactions            // Cache of the transactions already sorted
}
```

2、开始代码分析

2.1下面开始第一部分代码，介绍txPool的初始化代码

```go
// NewTxPool creates a new transaction pool to gather, sort and filter inbound
// transactions from the network.
func NewTxPool(config TxPoolConfig, chainconfig *params.ChainConfig, chain blockChain) *TxPool {
    // 检查配置  配置有问题则用默认值填充
    // Sanitize the input to ensure no vulnerable gas prices are set
    config = (&config).sanitize()

    // Create the transaction pool with its initial settings
    pool := &TxPool{
        config:      config,
        chainconfig: chainconfig,
        chain:       chain,
        signer:      types.NewEIP155Signer(chainconfig.ChainID),
        pending:     make(map[common.Address]*txList), // pending队列 也叫可执行队列
        queue:       make(map[common.Address]*txList),// queue队列  称为非可执行队列
        beats:       make(map[common.Address]time.Time), // beats 主要用来管理queue的生命周期
        all:         newTxLookup(), // 一个存放所有tx的遍历
        chainHeadCh: make(chan ChainHeadEvent, chainHeadChanSize), //chainHead事件通知通道
        gasPrice:    new(big.Int).SetUint64(config.PriceLimit), // 最小gas限制
    }
    // 初始化local account set
    pool.locals = newAccountSet(pool.signer)
    // 将配置的本地节点加到交易池
    for _, addr := range config.Locals {
        log.Info("Setting new local account", "address", addr)
        pool.locals.add(addr)
    }
    //创建所有交易存储的列表，所有交易的价格用最小堆存放
    pool.priced = newTxPricedList(pool.all)
    // 对交易池执行更新的主要逻辑
    pool.reset(nil, chain.CurrentBlock().Header())
    // 如果本地交易开启 那么从本地磁盘加载本地交易
    // If local transactions and journaling is enabled, load from disk
    if !config.NoLocals && config.Journal != "" {
        pool.journal = newTxJournal(config.Journal)

        if err := pool.journal.load(pool.AddLocals); err != nil {
            log.Warn("Failed to load transaction journal", "err", err)
        }
        if err := pool.journal.rotate(pool.local()); err != nil {
            log.Warn("Failed to rotate transaction journal", "err", err)
        }
    }
    //订阅区块头时间
    // Subscribe events from blockchain
    pool.chainHeadSub = pool.chain.SubscribeChainHeadEvent(pool.chainHeadCh)

    // Start the event loop and return
    pool.wg.Add(1)
     //开启主循环
    go pool.loop()
    return pool
}
```

上面主要是txPool的构造方法，主要做了如下几件事：

- 检查配置，若配置有问题则使用默认值填充
- 初始化txPool各个变量
- 调用reset方法对交易池进行一次刷新
- 如果开启了本地交易，那么查看本地交易的存放文件是否有交易，若有则加载本地交易进交易池
- 订阅区块头事件
- 开启主循环loop()

2.2loop()代码部分

```
// loop()主循环
// loop is the transaction pool's main event loop, waiting for and reacting to
// outside blockchain events as well as for various reporting and transaction
// eviction events.
func (pool *TxPool) loop() {
    defer pool.wg.Done()
    //状态报告变量
    // Start the stats reporting and transaction eviction tickers
    var prevPending, prevQueued, prevStales int
    //定时报告状态
    report := time.NewTicker(statsReportInterval)
    defer report.Stop()
    //  定时清理queue交易的触发器
    evict := time.NewTicker(evictionInterval)
    defer evict.Stop()
    // 定时将本地交易持久化到磁盘
    journal := time.NewTicker(pool.config.Rejournal)
    defer journal.Stop()
    // 获取区块头
    // Track the previous head headers for transaction reorgs
    head := pool.chain.CurrentBlock()

    // Keep waiting for and reacting to the various events
    for {
        select {
         // 处理区块头事件
        // Handle ChainHeadEvent
        case ev := <-pool.chainHeadCh:
            if ev.Block != nil {
                pool.mu.Lock()
                if pool.chainconfig.IsHomestead(ev.Block.Number()) {
                    pool.homestead = true
                }
                // 如果有新的区块进来  则进行一次pool reset 将最新的区块头 和上次记录的区块头传进去
                pool.reset(head.Header(), ev.Block.Header())
                  // 记录本次区块头 作为下一次的老区块头
                head = ev.Block

                pool.mu.Unlock()
            }
        // Be unsubscribed due to system stopped
        case <-pool.chainHeadSub.Err():
            return
        // 定时打印报告数据
        // Handle stats reporting ticks
        case <-report.C:
            pool.mu.RLock()
            pending, queued := pool.stats()
            stales := pool.priced.stales
            pool.mu.RUnlock()

            if pending != prevPending || queued != prevQueued || stales != prevStales {
                log.Debug("Transaction pool status report", "executable", pending, "queued",
queued, "stales", stales)
                prevPending, prevQueued, prevStales = pending, queued, stales
            }

        // Handle inactive account transaction eviction
        // 放逐机制 只清理queue里面的交易
        case <-evict.C:
            pool.mu.Lock()
            for addr := range pool.queue {
                // 跳过本地节点交易放逐机制
                // Skip local transactions from the eviction mechanism
                if pool.locals.contains(addr) {
                    continue
                }
                // Any non-locals old enough should be removed
                // 如果一个账户的心跳时间逾期大于 3小时  那么从本地删除关于此账户的所有交易
                if time.Since(pool.beats[addr]) > pool.config.Lifetime {
                    for _, tx := range pool.queue[addr].Flatten() {
                        pool.removeTx(tx.Hash(), true)
                    }
                }
            }
            pool.mu.Unlock()

        // Handle local transaction journal rotation
        //  定期将local地址的交易持久化到本地
        case <-journal.C:
            if pool.journal != nil {
                pool.mu.Lock()
                  // rotate regenerates the transaction journal based on the current contents of
the transaction pool.
                if err := pool.journal.rotate(pool.local()); err != nil {
                    log.Warn("Failed to rotate local tx journal", "err", err)
                }
                pool.mu.Unlock()
            }
        }
    }
}
```

主循环里面主要做了四件事情

- 从区块头监听通道读取数据，若有数据则拿到最新区块头，把上次拿到的区块头和当前最新的区块头传到reset方法，执行一次reset
- 定时出发报告事件，打印出pending、queue、stale的信息
- 触发放逐事件，将所有非本地账户且心跳超过三小时时长的账户的所有交易 remove掉
- 定时将交易池中本地交易持久化到磁盘

2.3 pool.reset()部分代码

```
// reset retrieves the current state of the blockchain and ensures the content
// of the transaction pool is valid with regard to the chain state.
func (pool *TxPool) reset(oldHead, newHead *types.Header) {
    // If we're reorging an old state, reinject all dropped transactions
    // 如果老的区块头和新的区块头不符合一定条件，那么需要将部分交易重新注入交易池
    var reinject types.Transactions
    // 如果老区块头不为空 且老区块头不是新区块的父区块
    if oldHead != nil && oldHead.Hash() != newHead.ParentHash {
        // If the reorg is too deep, avoid doing it (will happen during fast sync)
        oldNum := oldHead.Number.Uint64()
        newNum := newHead.Number.Uint64()
        // 如果老区块 和 新区块 相差大于64  那么不进行重组
        if depth := uint64(math.Abs(float64(oldNum) - float64(newNum))); depth > 64 {
            log.Debug("Skipping deep transaction reorg", "depth", depth)
        } else {

            // 重组看起来足够浅  所以可以将所有交易放入内存
            // Reorg seems shallow enough to pull in all transactions into memory
            //  定义了丢弃  已经包含的交易集
            var discarded, included types.Transactions

            var (
                rem = pool.chain.GetBlock(oldHead.Hash(), oldHead.Number.Uint64())
                add = pool.chain.GetBlock(newHead.Hash(), newHead.Number.Uint64())
            )
//--------------------------以下两个for循环，每次reset调用时候只能执行其中一个-----------------
            //  如果老区块的区块号大于新区块的区块号，那么逐个将老的区块向前推直到区块号不大于新区块的区块号为止，期间将所有老区块的交易放入discarded中
            for rem.NumberU64() > add.NumberU64() {
                discarded = append(discarded, rem.Transactions()...)
                if rem = pool.chain.GetBlock(rem.ParentHash(), rem.NumberU64()-1); rem == nil {
                    log.Error("Unrooted old chain seen by tx pool", "block", oldHead.Number,
"hash", oldHead.Hash())
                    return
                }
            }
            // 如果新区块的区块号大于老区块的区块号，那么逐个将新区块向前推直到区块号不大于老区块为止，期间将每一个新区块的所有交易加入included中
            for add.NumberU64() > rem.NumberU64() {
                included = append(included, add.Transactions()...)
                if add = pool.chain.GetBlock(add.ParentHash(), add.NumberU64()-1); add == nil {
                    log.Error("Unrooted new chain seen by tx pool", "block", newHead.Number,
"hash", newHead.Hash())
                    return
                }
            }
//-------------------------------------------------------------------------------------------
            //经过上面的循环以后按理来说，新老区块号已经相等
            // 那么此时若新老区块的hash还是不一样，说明不是同一个区块，那么同时将新区块和老区块往前逐个推，直到两个区块的hash相等，期间将每一个新区块的交易加入included,将每一个老区块的交易放入discarded
            for rem.Hash() != add.Hash() {
                discarded = append(discarded, rem.Transactions()...)
                if rem = pool.chain.GetBlock(rem.ParentHash(), rem.NumberU64()-1); rem == nil {
                    log.Error("Unrooted old chain seen by tx pool", "block", oldHead.Number, "hash", oldHead.Hash())
                    return
                }
                included = append(included, add.Transactions()...)
                if add = pool.chain.GetBlock(add.ParentHash(), add.NumberU64()-1); add == nil {
                    log.Error("Unrooted new chain seen by tx pool", "block", newHead.Number, "hash", newHead.Hash())
                    return
                }
            }
             // 所有在included中的交易说明已经存在于区块链上，所有在discarded中的交易是即将要丢弃的交易，将discarded中所有已经在included的交易排除掉，剩下的就是需要重新注入交易池的交易
            reinject = types.TxDifference(discarded, included)
        }
    }
    // Initialize the internal state to the current head
    if newHead == nil {
        newHead = pool.chain.CurrentBlock().Header() // Special case during testing
    }
    // 获取当前最新区块的状态
    statedb, err := pool.chain.StateAt(newHead.Root)
    if err != nil {
        log.Error("Failed to reset txpool state", "err", err)
        return
    }
    pool.currentState = statedb
    pool.pendingState = state.ManageState(statedb)
    pool.currentMaxGas = newHead.GasLimit
    // 注入所有交易

    // Inject any transactions discarded due to reorgs
    log.Debug("Reinjecting stale transactions", "count", len(reinject))
    senderCacher.recover(pool.signer, reinject)
    // 将所有需要重新注入的交易以非本地交易注入到交易池
    pool.addTxsLocked(reinject, false)

    // validate the pool of pending transactions, this will remove
    // any transactions that have been included in the block or
    // have been invalidated because of another transaction (e.g.
    // higher gas price)
    // 降级未执行的交易
    pool.demoteUnexecutables()

    // 执行完降级以后 更新所有账户 的 nonce
    // Update all accounts to the latest known pending nonce
    for addr, list := range pool.pending {
        txs := list.Flatten() // Heavy but will be cached and is needed by the miner anyway
        pool.pendingState.SetNonce(addr, txs[len(txs)-1].Nonce()+1)
    }

    // 下面开始执行交易升级 将所有账户队列中的交易进行升级或者移除
    // Check the queue and move transactions over to the pending if possible
    // or remove those that have become invalid
    pool.promoteExecutables(nil)
}
```

reset()方法主要做了以下几件事：

- 判断是否有需要重新注入的交易，若有则将相关交易重新注入交易池
- 下来对交易进行降级，验证交易池pending的所有交易，移除已经上链的和非法的交易
- 做完交易降级以后开始将所有账户的nonce更新一次
- 开始执行交易升级，将queue中所有可以升级的交易放入pending中，如果可能的话移除非法交易

2.4 add()方法分析

```
// 添加一个交易
// add validates a transaction and inserts it into the non-executable queue for
// later pending promotion and execution. If the transaction is a replacement for
// an already pending or queued one, it overwrites the previous and returns this
// so outer code doesn't uselessly call promote.
//
// If a newly added transaction is marked as local, its sending account will be
// whitelisted, preventing any associated transaction from being dropped out of
// the pool due to pricing constraints.(约束，限制)
func (pool *TxPool) add(tx *types.Transaction, local bool) (bool, error) {
    // If the transaction is already known, discard it
    hash := tx.Hash()
    // 看交易池所有交易是否已经存在此交易，如果存在则返回错误
    if pool.all.Get(hash) != nil {
        log.Trace("Discarding already known transaction", "hash", hash)
        return false, fmt.Errorf("known transaction: %x", hash)
    }
    //  开始对交易进行基础校验，基础校验代码后面分析
    // 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
    }
    // 如果交易所有的交易数量大于队列和pend的交易总数，那么执行如下逻辑
    // If the transaction pool is full, discard underpriced transactions
    if uint64(pool.all.Count()) >= pool.config.GlobalSlots+pool.config.GlobalQueue {
        // 如果交易非本地交易 此交易的价格比交易池所有交易里面最小gasPrice还小 那么直接返回错误
        // 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
        }
        // 将价格列表中最低的n+1个交易移除 n为交易池超出部分 多减去1是为了给新交易腾出位置
         //  将gasPrice最低的n+1个非本地交易删除，
        // New transaction is better than our worse ones, make room for it
        drop := pool.priced.Discard(pool.all.Count()-
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)
        }
    }
    // 回复本交易的发送者地址
    // If the transaction is replacing an already pending one, do directly
    from, _ := types.Sender(pool.signer, tx) // already validated
    // 如果账户的pending交易不为空，且交易依附的nonce已经存在在pending中 执行以下逻辑
    if list := pool.pending[from]; list != nil && list.Overlaps(tx) {
         // 一般情况下gasPrice大的交易会替换pending里面gasPrice小的交易
        // Nonce already pending, check if required price bump is met
        inserted, old := list.Add(tx, pool.config.PriceBump)
         // 如果没插入成功则丢弃交易
        if !inserted {
            pendingDiscardCounter.Inc(1)
            return false, ErrReplaceUnderpriced
        }
        // 新交易被插入  在所有已知交易里面删除老的交易
        // New transaction is better, replace old one
        if old != nil {
            pool.all.Remove(old.Hash())
            pool.priced.Removed()
            pendingReplaceCounter.Inc(1)
        }
        // 新交易添加到池中已知交易中
        pool.all.Add(tx)
        // 价格列表中放入此交易
        pool.priced.Put(tx)
        // 是否是本地交易  若是则放入本地交易中
        pool.journalTx(from, tx)

        log.Trace("Pooled new executable transaction", "hash", hash, "from", from, "to", tx.To())
        // 广播交易
        // We've directly injected a replacement transaction, notify subsystems
        go pool.txFeed.Send(NewTxsEvent{types.Transactions{tx}})

        return old != nil, nil
    }
    //  新交易没有不存在与发送账户的pending的列表  直接放入queue中
    // New transaction isn't replacing a pending one, push into queue
    replace, err := pool.enqueueTx(hash, tx)
    if err != nil {
        return false, err
    }
    // 如果交易被标记为本地交易且账户地址不在本地交易账户集中，那么将新的账户加入本地账户集
    // Mark local addresses and journal local transactions
    if local {
        if !pool.locals.contains(from) {
            log.Info("Setting new local account", "address", from)
            pool.locals.add(from)
        }
    }
    // 最后若交易为本地交易，那么将交易加入本地交易中
    pool.journalTx(from, tx)
```

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

```

add()方法主要将一个交易加入交易池中，主要做了以下事情：

- 首先先判断交易是否已经存在all中，若存在那么直接返回
- 对交易进行基础校验，如tx大小必须小于32KB、签名非负、gas不能超过限制、验证签名者、非本地交易要确保gasPrice大于gas最低限制、nonce需要大于账户当前nonce、账户余额必须大于当前交易的花费、gas数量必须大于固有gas消耗等
- 判断池中all交易是否大于pending+queue的总数，如果大于则进行交易drop，若当前交易gasPrice小于池中所有交易，那么直接返回，要不然将池中超出限制的gasprice倒叙排列的n个交易再加上1个交易drop掉
- 如果账户pending列表存在，而且此次tx的nonce再pending中已经存在，那么尝试替换已有的（gasPrice大于已有的交易情况下可以替换成功），替换成功后将老的交易从all里面删除，重新构建priced结构，在all里面添加新的交易，在priced里面放入新的交易，若是本地交易，将此交易插入到本地交易池中方便下次持久化到数据库，最后广播交易。
- 若账户不存在pending列表，那么直接将此交易放入queue里面

2.5 demoteUnexecutables()方法分析

```
// demoteUnexecutables removes invalid and processed transactions from the pools
// executable/pending queue and any subsequent transactions that become unexecutable
// are moved back into the future queue.
func (pool *TxPool) demoteUnexecutables() {
    // 遍历所有账户pending状态的交易
    // Iterate over all accounts and demote any non-executable transactions
    for addr, list := range pool.pending {
        //1、获取账户当前状态下的nonce值 并在pending list里面获取nonce值低于当前状态nonce的tx从all里面
删除
        nonce := pool.currentState.GetNonce(addr)
        // Drop all transactions that are deemed too old (low nonce)
        for _, tx := range list.Forward(nonce) {
            hash := tx.Hash()
            log.Trace("Removed old pending transaction", "hash", hash)
            // 从all交易中移除nonce过低的交易
            pool.all.Remove(hash)
             // 告知价格列表有交易被删除
            pool.priced.Removed()
        }
        // 2、从all中删除每个账户的花费大于账户余额的交易 ， 并将验证不通过的交易移入发送账户的queue中
        // Drop all transactions that are too costly (low balance or out of gas), and queue any
invalids back for later
        drops, invalids := list.Filter(pool.currentState.GetBalance(addr), pool.currentMaxGas)
        for _, tx := range drops {
            hash := tx.Hash()
            log.Trace("Removed unpayable pending transaction", "hash", hash)
            pool.all.Remove(hash)
            pool.priced.Removed()
            pendingNofundsCounter.Inc(1)
        }
        for _, tx := range invalids {
            hash := tx.Hash()
            log.Trace("Demoting pending transaction", "hash", hash)
            pool.enqueueTx(hash, tx)
        }
        // 经过排除nonce低的交易，花费较大的交易以后，判断账户pending是否还大于0且且pending列表中不存在跟当前nonce值相同的交易 那么将此地址的pending列表的交易全部取出来放入queue中
        // If there's a gap in front, alert (should never happen) and postpone all transactions
        if list.Len() > 0 && list.txs.Get(nonce) == nil {
            for _, tx := range list.Cap(0) {
                hash := tx.Hash()
                log.Error("Demoting invalidated transaction", "hash", hash)
                pool.enqueueTx(hash, tx)
            }
        }
        // 如果此账户的pending的list为空  那么从所有pending中删除此账户的list   从心跳列表里也删除此地址的
        // Delete the entire queue entry if it became empty.
        if list.Empty() {
            delete(pool.pending, addr)
            delete(pool.beats, addr)
        }
    }
}
```

上面主要是交易降级部分逻辑，主要做了以下事情：

- 遍历所有账户的pending列表，包括本地账户的交易
  - 将账户pending列表里面小于账户当前状态下的nonce值的交易全部删除
  - 将花费大于账户余额的交易和gas超出限制的交易drop掉 ，若严格模式开启，那么将那些小于将要删除的交易的最小nonce的正常交易拿出来放入queue
  - 如果账户经过如上两步以后pending列表长度大于0，且pending列表中不存在nonce值和账户当前状态下nonce值相等的交易，那么将账户所有pending列表的交易取出来放入queue

做完以上步骤后回到reset方法内部，程序将所有账户的nonce值做了一次更新，紧接着开始执行交易升级

2.6 promoteExecutables()方法分析

```
// promoteExecutables moves transactions that have become processable from the
// future queue to the set of pending transactions. During this process, all
// invalidated transactions (low nonce, low balance) are deleted.
func (pool *TxPool) promoteExecutables(accounts []common.Address) {
    // Track the promoted transactions to broadcast them at once
    var promoted []*types.Transaction
    // 从queue中收集所有账户地址
    // Gather all the accounts potentially needing updates
    if accounts == nil {
        accounts = make([]common.Address, 0, len(pool.queue))
        for addr := range pool.queue {
            accounts = append(accounts, addr)
        }
    }
    // 以下for循环遍历所有账户的queue做了如下事情 和降级交易部分逻辑有点相似
    // 1--将所有queue中nonce低于账户当前nonce的交易从all里面删除
    // 2--将所有花费大于账户余额 或者gas大于限制的交易从all里面删除
    // 3--将所有账户准备好的交易从queue里面移到pending里面
    // 4--判断非本地账户的其他账户的queue长度是不是超过限制 若超过限制 则将账户的queue里面“nonce较大”(待验证)的去除掉
    // 5--如果此账户的queue为空了 那么从pool.queue中删除此账户
    // Iterate over all accounts and promote any executable transactions
    for _, addr := range accounts {
        list := pool.queue[addr]
        if list == nil {
            continue // Just in case someone calls with a non existing account
        }

        // 拿到账户中nonce过低的交易 从all中删除
        // Drop all transactions that are deemed too old (low nonce)
        for _, tx := range list.Forward(pool.currentState.GetNonce(addr)) {
            hash := tx.Hash()
            log.Trace("Removed old queued transaction", "hash", hash)
            pool.all.Remove(hash)
            pool.priced.Removed()
        }
        //  过滤账户中花费大于余额或者gas超过限制的交易从all中删除
        // Drop all transactions that are too costly (low balance or out of gas)
        drops, _ := list.Filter(pool.currentState.GetBalance(addr), pool.currentMaxGas)
        for _, tx := range drops {
            hash := tx.Hash()
            log.Trace("Removed unpayable queued transaction", "hash", hash)
            pool.all.Remove(hash)
            pool.priced.Removed()
            queuedNofundsCounter.Inc(1)
        }
        // 将账户所有queue里面交易列表里面nonce准备好的交易取出来进行升级
         // 准备好的交易可以理解为将pending里面nonce值大于等于账户当前状态nonce的且nonce连续的几笔交易作
为准备好的交易
        // Gather all executable transactions and promote them
        for _, tx := range list.Ready(pool.pendingState.GetNonce(addr)) {
            hash := tx.Hash()
            // 此步骤主要将tx加入到账户的pending list里面
            if pool.promoteTx(addr, hash, tx) {
                log.Trace("Promoting queued transaction", "hash", hash)
                promoted = append(promoted, tx)
            }
        }
        //  drop掉所有超出账户队列限制的tx
        // Drop all transactions over the allowed limit
        if !pool.locals.contains(addr) {
            // 如果账户queue 大于给定限制的部分  从最后取出nonce较大的交易进行removed
            for _, tx := range list.Cap(int(pool.config.AccountQueue)) {
                hash := tx.Hash()
                pool.all.Remove(hash)
                pool.priced.Removed()
                queuedRateLimitCounter.Inc(1)
                log.Trace("Removed cap-exceeding queued transaction", "hash", hash)
            }
        }
        // 如果队列中此账户的交易为空则删除此账户
        // Delete the entire queue entry if it became empty.
        if list.Empty() {
            delete(pool.queue, addr)
        }
    }
    // 广播 进入pending的交易
    // Notify subsystem for new promoted transactions.
    if len(promoted) > 0 {
        go pool.txFeed.Send(NewTxsEvent{promoted})
    }
    // 将pending的所有账户的交易做统计，总数为pending
    // If the pending limit is overflown, start equalizing allowances
    pending := uint64(0)
    for _, list := range pool.pending {
        pending += uint64(list.Len())
    }
    // 如果pending大于全局限制的槽位 一般是4096
    if pending > pool.config.GlobalSlots {
        // 记录pending总数
        pendingBeforeCap := pending
        // 创建个优先级队列
        // Assemble a spam order to penalize(处罚) large transactors first
        spammers := prque.New(nil)
        // 遍历pending队列 将所有非本地账户的大于单账户pending list大小限制的账户放入一个优先级队列
        for addr, list := range pool.pending {
            // Only evict transactions from high rollers
            // 如果非本地地址 且账户pending list > AccountsSolt(16) 此处对本地账户进行绿灯保护
            if !pool.locals.contains(addr) && uint64(list.Len()) > pool.config.AccountSlots {
                // 将账户地址 和账户的pending list 长度作为优先级 放入队列
                 // 此队列将按照交易数量长度从大到小排列账户
                spammers.Push(addr, int64(list.Len()))
            }
        }
        // Gradually drop transactions from offenders
        offenders := []common.Address{}
        // 如果pending总数大于 全局槽位  且要处理掉的队列不为空
        for pending > pool.config.GlobalSlots && !spammers.Empty() {
            // 在此循环内部 要求offenders 最少存在两个惩罚者 且是pending tx最多的两个
            // Retrieve the next offender if not local address
            offender, _ := spammers.Pop()
            offenders = append(offenders, offender.(common.Address))
            // 如果惩罚者计数大于1
            // Equalize balances until all the same or below threshold
            if len(offenders) > 1 {
                // 设最后取出的offender的交易为当前阈值（阈值一直在变小）
                // Calculate the equalization threshold for all current offenders
                threshold := pool.pending[offender.(common.Address)].Len()
                // 如果pending tx总数还大于全局限制 且 第一个账户的pending 长度大于 后面取到的(按照优先
级队列来说 第一个长度肯定大于后面的)
                // Iteratively reduce all offenders until below limit or threshold reached
                for pending > pool.config.GlobalSlots &&
pool.pending[offenders[len(offenders)-2]].Len() > threshold {
                    // 将前面账户的pending list减少一个交易 如果pending一直大于全局限制那么一直减到比阈
值小
                    for i := 0; i < len(offenders)-1; i++ {
                        list := pool.pending[offenders[i]]
                        // nonce较大的会被删除
                        for _, tx := range list.Cap(list.Len() - 1) {
                            // Drop the transaction from the global pools too
                            hash := tx.Hash()
                            pool.all.Remove(hash)
                            pool.priced.Removed()

                            // Update the account nonce to the dropped transaction
                            if nonce := tx.Nonce(); pool.pendingState.GetNonce(offenders[i]) > 
nonce {
                                pool.pendingState.SetNonce(offenders[i], nonce)
                            }
                            log.Trace("Removed fairness-exceeding pending transaction", "hash", 
hash)
                        }
                        // pending 总计数减少1
                        pending--
                    }
                }
            }
        }
        // 如果经过上面操作以后pending总数还是大于全局设置的值 且 惩罚者数量大于0 将所有账户的pending的
tx数降低到 AccountSlots 一般为16
        // If still above threshold, reduce to limit or min allowance
        if pending > pool.config.GlobalSlots && len(offenders) > 0 {
            // 如果 pending总数大于全局阈值  且一个惩罚者的交易数大于单个账户的限制
            for pending > pool.config.GlobalSlots && 
uint64(pool.pending[offenders[len(offenders)-1]].Len()) > pool.config.AccountSlots {
                // 开始遍历惩罚者
                for _, addr := range offenders {
                    // 拿到惩罚者的pending list
                    list := pool.pending[addr]
                    // 将 此账户的 所有pending交易缩小一个
                    for _, tx := range list.Cap(list.Len() - 1) {
                        // Drop the transaction from the global pools too
                        hash := tx.Hash()
                        pool.all.Remove(hash)
                        pool.priced.Removed()
                        // 删除pending 交易以后 更新账户nonce
                        // Update the account nonce to the dropped transaction
                        if nonce := tx.Nonce(); pool.pendingState.GetNonce(addr) > nonce {
                            pool.pendingState.SetNonce(addr, nonce)
                        }
                        log.Trace("Removed fairness-exceeding pending transaction", "hash", hash)
                    }
                    // pending 交易总计数减一
                    pending--
                }
            }
        }
        // pending减少数量计数
        pendingRateLimitCounter.Inc(int64(pendingBeforeCap - pending))
    }
    // -----------------------------------------------------------------------------------------
----------------------
    // If we've queued more transactions than the hard limit, drop oldest ones
    queued := uint64(0)
    // 统计所有账户queue中tx的总数
    for _, list := range pool.queue {
        queued += uint64(list.Len())
    }
    // 如果queue中tx 总数大于全局总数限制
    if queued > pool.config.GlobalQueue {
        // Sort all accounts with queued transactions by heartbeat
        addresses := make(addressesByHeartbeat, 0, len(pool.queue))
        for addr := range pool.queue {
            // 将队列中所有非本地账户的地址和心跳统计出来
            if !pool.locals.contains(addr) { // don't drop locals
                addresses = append(addresses, addressByHeartbeat{addr, pool.beats[addr]})
            }
        }
        sort.Sort(addresses)
        // 计算drop数量等于queue总数减去全局queue限制，如果drop大于0且地址数量大于0进行循环
        // Drop transactions until the total is below the limit or only locals remain
        for drop := queued - pool.config.GlobalQueue; drop > 0 && len(addresses) > 0; {
            // 取出heart beat时间最大的 queue list
            addr := addresses[len(addresses)-1]
            list := pool.queue[addr.address]
            //将addresses 缩短一个位置
            addresses = addresses[:len(addresses)-1]
            // 如果一个heart beats时间长的账户的所有交易小于要drop掉的数量  那么remove掉此账户的所有tx 
跳过本次循环
            // Drop all transactions if they are less than the overflow
            if size := uint64(list.Len()); size <= drop {
                for _, tx := range list.Flatten() {
                    pool.removeTx(tx.Hash(), true)
                }
                drop -= size
                queuedRateLimitCounter.Inc(int64(size))
                continue
            }
            // 要不然从此账户的txs集合里面最后删除掉一部分tx 以符合要求
            // Otherwise drop only last few transactions
            txs := list.Flatten()
            for i := len(txs) - 1; i >= 0 && drop > 0; i-- {
                pool.removeTx(txs[i].Hash(), true)
                drop--
                queuedRateLimitCounter.Inc(1)
            }
        }
    }
}
```

上面就是交易升级的代码分析，主要做了如下几件事情：

- 先遍历queue里面的所有账户
    - 将每个账户的queue交易列表中所有nonce低于当前状态nonce的交易从all里面删除
    - 将交易费用大于当前余额的交易或者gas大于当前交易池最大gas限制的交易从all里面删除
    - 将准备好进入pending队列的交易拿出来放入pending队列
    - 最后将非本地账户中大于单个账户queue大小限制的交易（一般是nonce较大的）拿出来删除
- 将进入pending队列的交易进行事件广播
- 下来主要整理pending队列，若在执行了上面交易从queue进入pending后，pending交易总是超出全局pending限制，那么进行处理
- 若queue中总交易数超出全局限制，一般为1024个，那么进行相关删除queue里面交易的操作

以上就是所有交易池主循环loop()中的逻辑函数的分析。

3、额外部分分析

3.1 types.TxDifference()

```
// TxDifference returns a new set which is the difference between a and b.
func TxDifference(a, b Transactions) Transactions {
    // types.TxDifference(discarded, included)
    keep := make(Transactions, 0, len(a))

    remove := make(map[common.Hash]struct{})
    // 把已经存在的所有txHash放在一个map的key里面
    for _, tx := range b {
        remove[tx.Hash()] = struct{}{}
    }
    // 要丢弃的交易没在链上的 那么得保留下来
    for _, tx := range a {
        if _, ok := remove[tx.Hash()]; !ok {
            keep = append(keep, tx)
        }
    }
    return keep
}
```

3.2 pool.validateTx()

```
// validateTx checks whether a transaction is valid according to the consensus
// rules and adheres to some heuristic limits of the local node (price and size).
func (pool *TxPool) validateTx(tx *types.Transaction, local bool) error {
    // Heuristic limit, reject transactions over 32KB to prevent DOS attacks
    // 交易最大大小为32KB
    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.
    // 对交易的签名结果做非负判断
    if tx.Value().Sign() < 0 {
        return ErrNegativeValue
    }
    // Ensure the transaction doesn't exceed the current block limit gas.
    // 交易的gas值不能大于交易池当前最大gas限制
    if pool.currentMaxGas < tx.Gas() {
        return ErrGasLimit
    }
    // Make sure the transaction is signed properly
    // 确保交易被正确签名了
    from, err := types.Sender(pool.signer, tx)
    if err != nil {
        return ErrInvalidSender
    }
    // Drop non-local transactions under our own minimal accepted gas price
    // 判断是不是local交易 如果不是local交易且gasPrice低于交易池最低gasPrice限制则返回错误 如果是local
交易则跳过此验证
    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
    }
    // 如果当前状态下发送方的nonce值大于此交易的nonce 那么返回错误
    // 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
    // 如果发送方账户的余额低于此交易的花费 返回余额不足错误
    if pool.currentState.GetBalance(from).Cmp(tx.Cost()) < 0 {
        return ErrInsufficientFunds
    }
    // 判断gas数量是否小于固有gas数量  如果小于则返回gas数量不足
    intrGas, err := IntrinsicGas(tx.Data(), tx.To() == nil, pool.homestead)
    if err != nil {
        return err
    }
    if tx.Gas() < intrGas {
        return ErrIntrinsicGas
    }
    return nil
}
```

3.3 list.Forward()

```
// Forward removes all transactions from the map with a nonce lower than the
// provided threshold. Every removed transaction is returned for any post-removal
// maintenance.
func (m *txSortedMap) Forward(threshold uint64) types.Transactions {
    var removed types.Transactions
    // 从最小堆里面拿出nonce小于给定值的tx放入removed
    // Pop off heap items until the threshold is reached
    for m.index.Len() > 0 && (*m.index)[0] < threshold {
        nonce := heap.Pop(m.index).(uint64)
        removed = append(removed, m.items[nonce])
        delete(m.items, nonce)
    }
    // If we had a cached order, shift the front
    if m.cache != nil {
        m.cache = m.cache[len(removed):]
    }
    return removed
}
```

3.4 list.Cap()

```
// Cap places a hard limit on the number of items, returning all transactions
// exceeding that limit.
func (m *txSortedMap) Cap(threshold int) types.Transactions {
    // Short circuit if the number of items is under the limit
    //如果元素个数小于给定阈值，那么直接返回
    if len(m.items) <= threshold {
        return nil
    }
    // Otherwise gather and drop the highest nonce'd transactions
    var drops types.Transactions
    //要不然将list的nonce从小到大排序
    sort.Sort(*m.index)
     // 将最后几个超出限制的nonce值较大的交易返回
    for size := len(m.items); size > threshold; size-- {
        drops = append(drops, m.items[(*m.index)[size-1]])
        delete(m.items, (*m.index)[size-1])
    }
    //重建nonce堆
    *m.index = (*m.index)[:threshold]
    heap.Init(m.index)

    // If we had a cache, shift the back
    if m.cache != nil {
        m.cache = m.cache[:len(m.cache)-len(drops)]
    }
    return drops
}
```

3.5 pool.priced.Removed()

```
// Removed notifies the prices transaction list that an old transaction dropped
// from the pool. The list will just keep a counter of stale objects and update
// the heap if a large enough ratio of transactions go stale.
func (l *txPricedList) Removed() {
    // Bump the stale counter, but exit if still too low (< 25%)
    // 当有一个tx从all中删除了 那么l.stales（陈旧交易计数）进行自增，
    l.stales++
     //如果陈旧交易小于所有交易总数的25%，那么直接返回
    if l.stales <= len(*l.items)/4 {
        return
    }
    // 要不然将堆所有交易进行一次重新按价格建堆操作
    // Seems we've reached a critical number of stale transactions, reheap
    reheap := make(priceHeap, 0, l.all.Count())

    l.stales, l.items = 0, &reheap
    l.all.Range(func(hash common.Hash, tx *types.Transaction) bool {
        *l.items = append(*l.items, tx)
        return true
    })
    heap.Init(l.items)
}
```