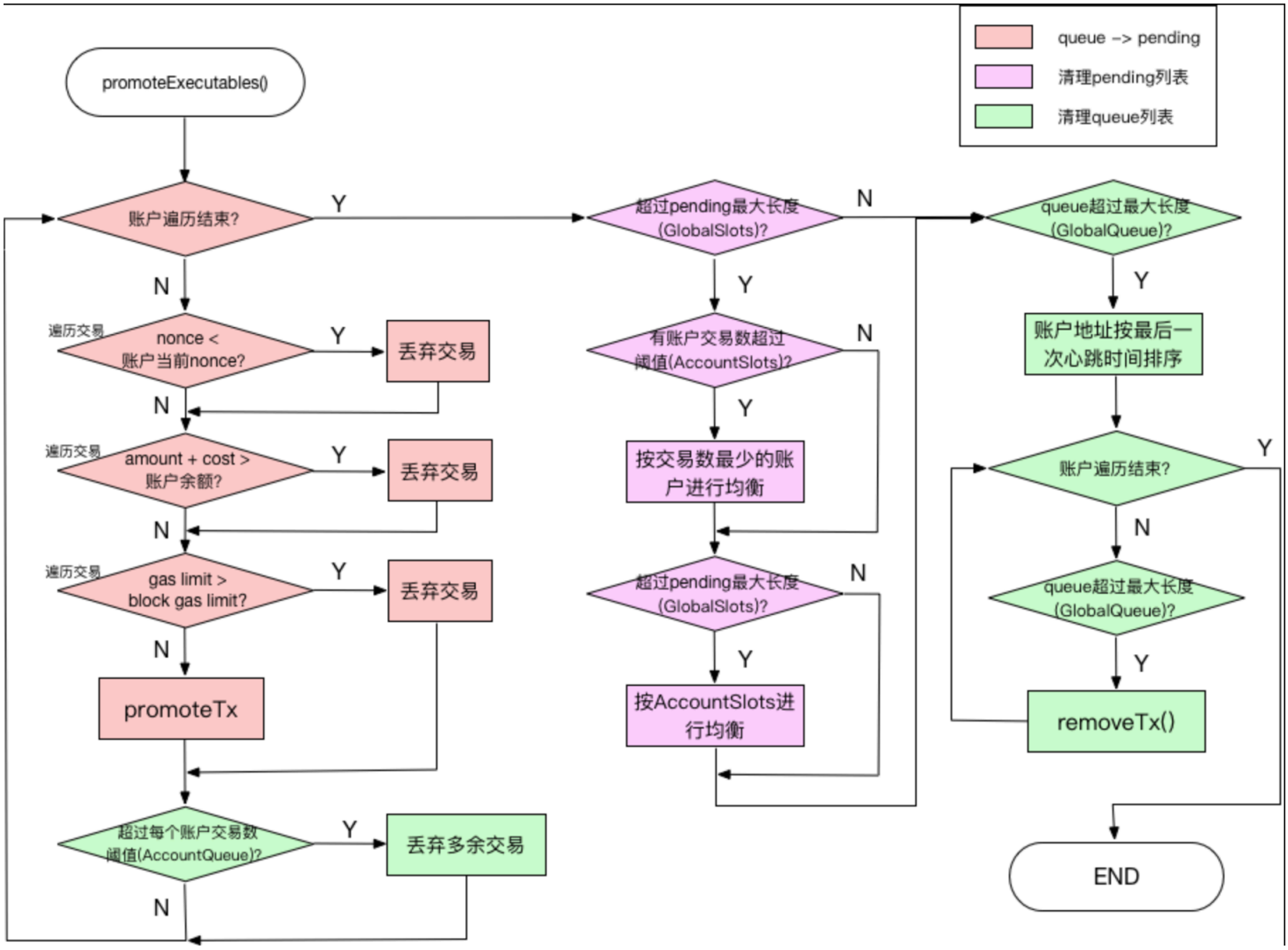
# 以太坊最重要角色之以太坊交易源码解析2

## TxPool.promoteExecuteables()

主要目的是把交易从queue列表“提拔”到pending列表，代码逻辑比较清楚，具体可以参见下面这张图：




根据不同的目的可以分为3块，分别以粉色、紫色、绿色标识。

粉色部分主要是为了把queue中的交易“提拔”到pending中。当然在这之前需要先要进行一番检查：

- 丢弃nonce < 账户当前nonce的交易，也就是已经被打包过的交易
- 丢弃转账金额 + gas消耗 > 账户余额的交易，也就是会out-of-gas的交易
- 丢弃gas limit > block gas limit的交易，这部分交易可能会导致区块生成失败

紫色部分主要是为了清理pending列表，使其满足GlobalSlots和AccountSlots的限制条件：

- 如果有些账户的交易数超过了AccountSlots，则先按交易数最少的账户进行均衡。举例来说，如果有10个账户交易数超过了AccountSlots（默认16），其中交易数最少的账户包含20笔交易，那么先把其他9个账户的交易数量削减到20。
- 如果经过上面的步骤，pending的长度还是超过了GlobalSlots，那就严格按照AccountSlots进行均衡，也就是把上面的10个账户的交易数进一步削减到16。

绿色部分主要是为了清理queue列表，使其满足GlobalQueue和AccountQueue的限制条件：

- 如果每个账户的交易数超过了AccountQueue，丢弃多余交易
- 如果queue的长度超过了GlobalQueue，则把账户按最后一次心跳时间排序，然后依次去除账户中的交易，直到满足限制条件位置。

```
// 代码位于 core/tx_pool.go

func (pool *TxPool) promoteExecutables(accounts []common.Address) {
    // Track the promoted transactions to broadcast them at once
    var promoted []*types.Transaction

    // 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)
        }
    }
    // 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
        }
        // 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()
        }
        // Drop all transactions that are too costly (low balance or out of gas)
        drops, _ := list.Filter(pool.currentState.GetBalance(addr), pool.currentMax
Gas)
        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)
        }
        // Gather all executable transactions and promote them
        for _, tx := range list.Ready(pool.pendingState.GetNonce(addr)) {
            hash := tx.Hash()
            if pool.promoteTx(addr, hash, tx) {
                log.Trace("Promoting queued transaction", "hash", hash)
                promoted = append(promoted, tx)
            }
        }
        // Drop all transactions over the allowed limit
        if !pool.locals.contains(addr) {
            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)
            }
        }
        // Delete the entire queue entry if it became empty.
        if list.Empty() {
            delete(pool.queue, addr)
        }
    }
    // Notify subsystem for new promoted transactions.
    if len(promoted) > 0 {
        pool.txFeed.Send(NewTxsEvent{promoted})
    }
    // If the pending limit is overflown, start equalizing allowances
    pending := uint64(0)
    for _, list := range pool.pending {
        pending += uint64(list.Len())
    }
    if pending > pool.config.GlobalSlots {
        pendingBeforeCap := pending
        // Assemble a spam order to penalize large transactors first
        spammers := prque.New()
        for addr, list := range pool.pending {
            // Only evict transactions from high rollers
            if !pool.locals.contains(addr) && uint64(list.Len()) > pool.config.Acco
untSlots {
                spammers.Push(addr, float32(list.Len()))
            }
        }
        // Gradually drop transactions from offenders
```

```
        offenders := []common.Address{}
        for pending > pool.config.GlobalSlots && !spammers.Empty() {
            // Retrieve the next offender if not local address
            offender, _ := spammers.Pop()
            offenders = append(offenders, offender.(common.Address))

            // Equalize balances until all the same or below threshold
            if len(offenders) > 1 {
                // Calculate the equalization threshold for all current offenders
                threshold := pool.pending[offender.(common.Address)].Len()

                // Iteratively reduce all offenders until below limit or threshold
reached
                for pending > pool.config.GlobalSlots && pool.pending[offenders[len
(offenders)-2]].Len() > threshold {
                    for i := 0; i < len(offenders)-1; i++ {
                        list := pool.pending[offenders[i]]
                        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(offe
nders[i]) > nonce {
                                pool.pendingState.SetNonce(offenders[i], nonce)
                            }
                            log.Trace("Removed fairness-exceeding pending transacti
on", "hash", hash)
                        }
                        pending--
                    }
                }
            }
        }
        // If still above threshold, reduce to limit or min allowance
        if pending > pool.config.GlobalSlots && len(offenders) > 0 {
            for pending > pool.config.GlobalSlots && uint64(pool.pending[offenders[
len(offenders)-1]].Len()) > pool.config.AccountSlots {
                for _, addr := range offenders {
                    list := pool.pending[addr]
                    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(addr) >
nonce {
                            pool.pendingState.SetNonce(addr, nonce)
                        }
                        log.Trace("Removed fairness-exceeding pending transaction",
 "hash", hash)
                    }
                    pending--
                }
            }
        }
        pendingRateLimitCounter.Inc(int64(pendingBeforeCap - pending))
    }
    // If we've queued more transactions than the hard limit, drop oldest ones
    queued := uint64(0)
    for _, list := range pool.queue {
        queued += uint64(list.Len())
    }
    if queued > pool.config.GlobalQueue {
        // Sort all accounts with queued transactions by heartbeat
        addresses := make(addresssByHeartbeat, 0, len(pool.queue))
        for addr := range pool.queue {
            if !pool.locals.contains(addr) { // don't drop locals
                addresses = append(addresses, addressByHeartbeat{addr, pool.beats[a
ddr]})
            }
        }
        sort.Sort(addresses)

        // Drop transactions until the total is below the limit or only locals rema
in
        for drop := queued - pool.config.GlobalQueue; drop > 0 && len(addresses) >
0; {
            addr := addresses[len(addresses)-1]
            list := pool.queue[addr.address]

            addresses = addresses[:len(addresses)-1]

            // 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
            }
```

```
            // 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)
            }
        }
    }
}
```

# 广播交易

交易提交到txpool中后，还需要广播出去，一方面通知EVM执行该交易，另一方面要把交易信息广播给其他结点。具体调用在 `promoteExecutables` 提到的promoteTx()函数中：

```
// 代码位于 core/tx_pool.go

func (pool *TxPool) promoteExecutables(accounts []common.Address) {

...
 // Gather all executable transactions and promote them
        for _, tx := range list.Ready(pool.pendingState.GetNonce(addr)) {
            hash := tx.Hash()
            if pool.promoteTx(addr, hash, tx) {
                log.Trace("Promoting queued transaction", "hash", hash)
                promoted = append(promoted, tx)
            }
        }
    }

    ...
    // Notify subsystem for new promoted transactions.
    if len(promoted) > 0 {
        pool.txFeed.Send(NewTxsEvent{promoted})
    }
```

promoteTx 详细代码：

```
// 代码 crypto/tx_pool.go
func (pool *TxPool) promoteTx(addr common.Address, hash common.Hash, tx *types.Tran
saction) bool {
    // Try to insert the transaction into the pending queue
    if pool.pending[addr] == nil {
```

```
        pool.pending[addr] = newTxList(true)
    }
    list := pool.pending[addr]

    inserted, old := list.Add(tx, pool.config.PriceBump)
    if !inserted {
        // An older transaction was better, discard this
        pool.all.Remove(hash)
        pool.priced.Removed()

        pendingDiscardCounter.Inc(1)
        return false
    }
    // Otherwise discard any previous transaction and mark this
    if old != nil {
        pool.all.Remove(old.Hash())
        pool.priced.Removed()

        pendingReplaceCounter.Inc(1)
    }
    // Failsafe to work around direct pending inserts (tests)
    if pool.all.Get(hash) == nil {
        pool.all.Add(tx)
        pool.priced.Put(tx)
    }
    // Set the potentially new pending nonce and notify any subsystems of the new t
x
    pool.beats[addr] = time.Now()
    pool.pendingState.SetNonce(addr, tx.Nonce()+1)

    return true
}
```

先更新了最后一次心跳时间，然后更新账户的nonce值。

**pool.txFeed.Send** 发送一个TxPreEvent事件，外部可以通过SubscribeNewTxsEvent()函数订阅该事件：

```
func (pool *TxPool) SubscribeNewTxsEvent(ch chan<- core.NewTxsEvent) event.Subscrip
tion {
    return pool.scope.Track(pool.txFeed.Subscribe(ch))
}
```

我们只要搜索一下这个函数，就可以知道哪些组件订阅了该事件了。

# 六、执行交易

第一个订阅的地方位于miner/worker.go：

```
func newWorker(config *params.ChainConfig, engine consensus.Engine, coinbase common
.Address, eth Backend, mux *event.TypeMux) *worker {
    ....

    // Subscribe NewTxsEvent for tx pool
    worker.txsSub = eth.TxPool().SubscribeNewTxsEvent(worker.txsCh)
    ....
}
```

开启了一个goroutine来接收TxPreEvent，看一下update()函数：

```
func (self *worker) update() {
    defer self.txsSub.Unsubscribe()
    defer self.chainHeadSub.Unsubscribe()
    defer self.chainSideSub.Unsubscribe()

    for {
        ...

        // Handle NewTxsEvent
        case ev := <-self.txsCh:
            // Apply transactions to the pending state if we're not mining.
            //
            // Note all transactions received may not be continuous with transactio
ns
            // already included in the current mining block. These transactions wil
l
            // be automatically eliminated.
            if atomic.LoadInt32(&self.mining) == 0 {
                self.currentMu.Lock()
                txs := make(map[common.Address]types.Transactions)
                for _, tx := range ev.Txs {
                    acc, _ := types.Sender(self.current.signer, tx)
                    txs[acc] = append(txs[acc], tx)
                }
                txset := types.NewTransactionsByPriceAndNonce(self.current.signer,
txs)
                self.current.commitTransactions(self.mux, txset, self.chain, self.c
oinbase)
                self.updateSnapshot()
                self.currentMu.Unlock()
```

```
            } else {
                // If we're mining, but nothing is being processed, wake on new tra
nsactions
                if self.config.Clique != nil && self.config.Clique.Period == 0 {
                    self.commitNewWork()
                }
            }

        ...
        }
    }
}
```

可以看到，如果结点不挖矿的话，这里会立即调用commitTransactions()提交给EVM执行，获得本地回执。

如果结点挖矿的话，miner会调用commitNewWork()，内部也会调用commitTransactions()执行交易。

## 七、广播给其他结点

另一个订阅的地方位于eth/handler.go：

```
func (pm *ProtocolManager) Start(maxPeers int) {
...

    // broadcast transactions
    pm.txsCh = make(chan core.NewTxsEvent, txChanSize)
    pm.txsSub = pm.txpool.SubscribeNewTxsEvent(pm.txsCh)
    go pm.txBroadcastLoop()

...
}
```

同样也是启动了一个goroutine来接收TxPreEvent事件，看一下txBroadcastLoop()函数：

```
func (pm *ProtocolManager) txBroadcastLoop() {
    for {
        select {
        case event := <-pm.txCh:
            pm.BroadcastTx(event.Tx.Hash(), event.Tx)

        // Err() channel will be closed when unsubscribing.
        case <-pm.txSub.Err():
```

```
            return
        }
    }
}
```

继续跟踪BroadcastTx()函数：

```
func (pm *ProtocolManager) BroadcastTxs(txs types.Transactions) {
    var txset = make(map[*peer]types.Transactions)

    // Broadcast transactions to a batch of peers not knowing about it
    for _, tx := range txs {
        peers := pm.peers.PeersWithoutTx(tx.Hash())
        for _, peer := range peers {
            txset[peer] = append(txset[peer], tx)
        }
        log.Trace("Broadcast transaction", "hash", tx.Hash(), "recipients", len(pee
rs))
    }
    // FIXME include this again: peers = peers[:int(math.Sqrt(float64(len(peers))))
]
    for peer, txs := range txset {
        peer.AsyncSendTransactions(txs)
    }
}
```

可以看到，这里会通过P2P向所有没有该交易的结点发送该交易。