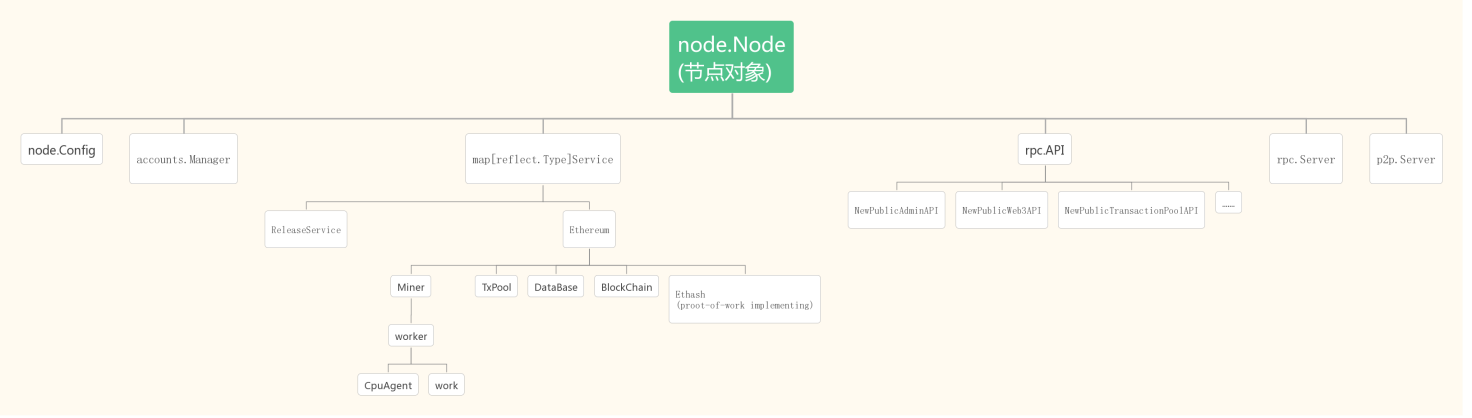
1. ethereum

**一：主要封装结构之间的依赖关系图：**

**二：在命令行中通过rpc控制节点（以提交交易为例）**

**1：注册rpc接口**

**package** ethapi

**func** GetAPIs(apiBackend Backend) []rpc.API {

**return** []rpc.API{

{

Namespace: **"eth"**,

Version: **"1.0"**,

Service: NewPublicTransactionPoolAPI

(apiBackend),

Public: ***true***,

}, {

**2：api接口调用**

命令行命令会被解析成rpc调用，对应服务对象的某个函数，例如提交交易命令：

> eth.sendTransaction({from: acc0, to: acc1, value: amount})

最终调用的api函数是：

**package** ethapi

**func** (s \*PublicTransactionPoolAPI) SendTransaction(ctx context.Context, args SendTxArgs) (common.Hash, error) {

PublicTransactionPoolAPI的实例，在第1步中注册到rpc接口服务。

**三：保存交易、合约到交易池中：**

1：通过步骤二，一个新的交易或者合约提交到节点服务器后，首先使用账户私钥进行签名：

**func** (w \*keystoreWallet) SignTx(account accounts.Account, tx \*types.Transaction, chainID \*big.Int) (\*types.Transaction, error) {

其中会考虑EIP155方案，避免以太坊分叉重发攻击。

2：然后存放在TxPool.queue中，然后转移TxPool.pending中：

**func** submitTransaction(ctx context.Context, b Backend, tx \*types.Transaction) (common.Hash, error) {

调用

**func** (b \*EthApiBackend) SendTx(ctx context.Context, signedTx \*ty

pes.Transaction) error {

**func** (pool \*TxPool) Add(tx \*types.Transaction) error {

**func** (pool \*TxPool) promoteExecutables(state \*state.StateDB) {

转移到TxPool.pending的实现，较复杂。

3：如果是合约（to为空），则根据from和noce生成合约地址：

**if** tx.To() == nil {

signer := types.MakeSigner(b.ChainConfig(), b.CurrentBlock().Number())

from, \_ := types.Sender(signer, tx)

addr := crypto.CreateAddress(from, tx.Nonce())

log.Info(**"Submitted contract creation"**, **"fullhash"**, tx.Hash().Hex(), **"contract"**, addr.Hex())

} **else** {

**四：交易的执行：**

交易、合约的执行是在挖矿工作量正忙之前完成，这一串行操作是由channel消息驱动：

**func** (self \*worker) update() {

**for** event := **range** self.events.Chan() {

*// A real event arrived, process interesting content*

**switch** ev := event.Data.(**type**) {

**case** core.ChainHeadEvent:

self.commitNewWork()

**func** (self \*worker) commitNewWork() {

1：准备工作

生成新block的时间戳，必要时sleep，以控制挖矿的时机

生成block header

计算block difficulty

**func** (ethash \*Ethash) Prepare(chain consensus.ChainReader, header \*types.Header) error {

处理DAO分叉

创建新的挖矿work对象

2：执行交易

逐个执行TxPool.pending中缓存的交易：

**func** (env \*Work) commitTransactions(mux \*event.TypeMux, txs \*types.TransactionsByPriceAndNonce, gasPrice \*big.Int, bc \*core.BlockChain, coinbase common.Address) {

这个过程还会执行合约代码：

**func** (evm \*EVM) Create(caller ContractRef, code []byte, gas uint64, value \*big.Int) (ret []byte, contractAddr common.Address, leftOverGas uint64, err error) {

**func** (evm \*Interpreter) Run(contract \*Contract, input []byte) (ret []byte, err error) {

执行结果分三种情况：gas不足，其他原因失败，成功。

输出结果分别存放在：Work.lowGasTxs,Work.failedTxs,Work.txs、Work.receipts

后续挖矿只处理成功的交易。

3：如果有叔分支，处理叔分支

4：根据前面的输出，生成新的block，供后续的挖矿使用

**五：挖矿**

挖矿通知由channel消息控制：

**func** (self \*worker) push(work \*Work) {

**if** atomic.LoadInt32(&self.mining) != 1 {

**return**

}

**for** agent := **range** self.agents {

atomic.AddInt32(&self.atWork, 1)

**if** ch := agent.Work(); ch != nil {

ch <- work

}

}

}

挖矿工作量证明实现：

**func** (ethash \*Ethash) Seal(chain consensus.ChainReader, block \*types.Block, stop <-**chan struct**{}) (\*types.Block, error) {

根据设置的miner并发数，启动相应数量的协程，同时进行挖矿。如果并发数设置为0,则获取当期机器的CPU逻辑内核数，作为挖矿并发数。挖矿并发函数：

**func** (ethash \*Ethash) mine(block \*types.Block, id int, seed uint64, abort **chan struct**{}, found **chan** \*types.Block) {

使用无限循环，不断尝试随机数计算block hash，直到满足PoW或者收到终止消息。

**六：后续处理挖到的矿**

新挖到的矿会通过**channel**消息逐层传给**worker**：

**Ethash-→CpuAgent-->worker**

***func*** *(ethash \*Ethash) mine(block \*types.Block, id int, seed uint64, abort chan struct{}, found chan \*types.Block) {*

***// Seal and return a block (if still needed)***

**select** {

**case** found <- block.WithSeal(header):

**func** (ethash \*Ethash) Seal(chain consensus.ChainReader, block \*types.Block, stop <-chan struct{}) (\*types.Block, error) {

**…...**

**select** {

**case** <-stop:

*// Outside abort, stop all miner threads*

close(abort)

**case** result = <-found:

**func** (self \*CpuAgent) mine(work \*Work, stop <-chan struct{}) {

self.returnCh <- &Result{work, result}

**func** (self \*worker) wait() {

**for** result := **range** self.recv {

保存新block数据到数据库

**func** (self \*LDBDatabase) Put(key []byte, value []byte) {

数据都是以key-->val的形式保存在数据库。他们的对应关系是：

headerPrefix = []byte(**"h"**) *// headerPrefix + num (uint64 big endian) + hash -> header*

tdSuffix = []byte(**"t"**) *// headerPrefix + num (uint64 big endian) + hash + tdSuffix -> td*

numSuffix = []byte(**"n"**) *// headerPrefix + num (uint64 big endian) + numSuffix -> hash*

blockHashPrefix = []byte(**"H"**) *// blockHashPrefix + hash -> num (uint64 big endian)*

bodyPrefix = []byte(**"b"**) *// bodyPrefix + num (uint64 big endian) + hash -> block body*

blockReceiptsPrefix = []byte(**"r"**) *// blockReceiptsPrefix + num (uint64 big endian) + hash -> block receipts*

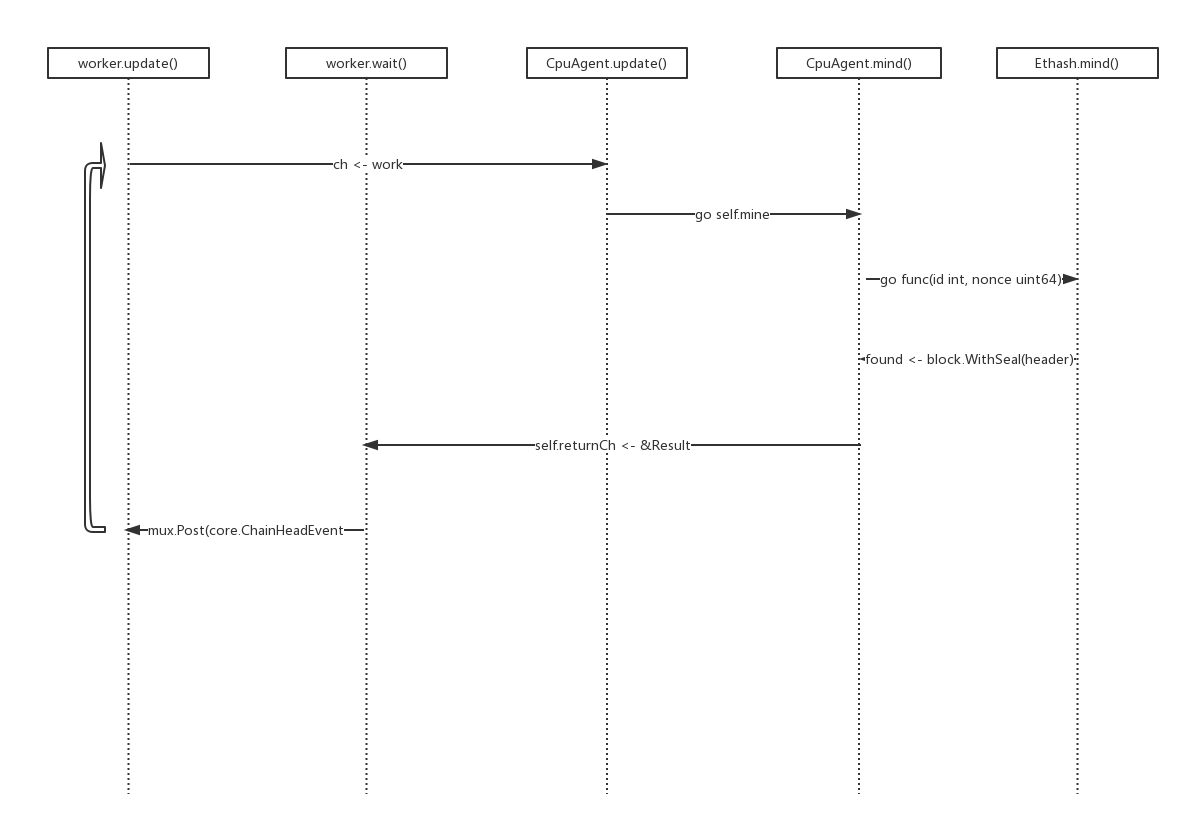
preimagePrefix = **"secure-key-"** *// preimagePrefix + hash -> preimage*

广播消息NewMinedBlockEvent、ChainEvent、ChainHeadEvent

、types.Log，其中消息ChainHeadEvent会触发worker.update()，进行下一次交易处理、挖矿操作。

将新block添加到待确认队列中

**七：挖矿各个协程之间的驱动关系图：**



**八：通过P2P与外界同步数据**

1：广播本地交易

在通过rpc提交交易时，转移到TxPool.pending之前，先判断交易是否有效，无效的将被直接移除，不予处理：

**func** (pool \*TxPool) promoteExecutables(state \*state.StateDB) {

drops, \_ := list.Filter(state.GetBalance(addr))

**for** \_, tx := **range** drops {

**if** tx.Txtype() != 0 {

hash := tx.Hash()

log.Debug(**"Removed unpayable queued transaction"**, **"hash"**, hash)

delete(pool.all, hash)

queuedNofundsCounter.Inc(1)

}

}

交易有效判断，先判断账户余额是否充足，如果不充足将做必要的调整（删除部分交易）

**func** (l \*txList) Filter(threshold \*big.Int) (types.Transactions, types.Transactions) {

可以被执行的交易将被保存在TxPool.pending,同时通过订阅广播通知：

**go** pool.eventMux.Post(TxPreEvent{tx})

订阅者收到消息后，通过p2p进行全网广播：

**func** (self \*ProtocolManager) txBroadcastLoop() {

**for** obj := **range** self.txSub.Chan() {

**func** (pm \*ProtocolManager) BroadcastTx(hash common.Hash, tx \*types.Transaction) {

**func** (p \*peer) SendTransactions(txs types.Transactions) error {

**for** \_, tx := **range** txs {

p.knownTxs.Add(tx.Hash())

}

**return** p2p.Send(p.rw, ***TxMsg***, txs)

}

2：广播本地新block

本地挖到block之后，通知到worker.wait()做后续处理，在wait()中将block存储到数据库之后，发出订阅广播通知：

self.mux.Post(core.NewMinedBlockEvent{Block: block})

挖矿广播循环函数收到通知后，通过p2p进行全网广播：

*// Mined broadcast loop*

**func** (self \*ProtocolManager) minedBroadcastLoop() {

*// automatically stops if unsubscribe*

**for** obj := **range** self.minedBlockSub.Chan() {

**switch** ev := obj.Data.(**type**) {

**case** core.NewMinedBlockEvent:

self.BroadcastBlock(ev.Block, ***true***) *// First propagate block to peers*

self.BroadcastBlock(ev.Block, ***false***) *// Only then announce to the rest*

}

}

}

3：P2P接收外部交易

通过for无限循环，不停得监听每个远程端点(peer)的网络消息：

**func** (pm \*ProtocolManager) handle(p \*peer) error {

*// main loop. handle incoming messages.*

**for** {

**if** err := pm.handleMsg(p); err != nil {

p.Log().Debug(**"Ethereum message handling failed"**, **"err"**, err)

**return** err

}

}

**func** (pm \*ProtocolManager) handleMsg(p \*peer) error {

*// Read the next message from the remote peer, and ensure it's fully consumed*

msg, err := p.rw.ReadMsg()

**case** msg.Code == ***TxMsg***:

*// Transactions arrived, make sure we have a valid and fresh chain to handle them*

**if** atomic.LoadUint32(&pm.acceptTxs) == 0 {

**break**

}

*// Transactions can be processed, parse all of them and deliver to the pool*

**var** txs []\*types.Transaction

**if** err := msg.Decode(&txs); err != nil {

**return** errResp(***ErrDecode***, **"msg %v: %v"**, msg, err)

}

**for** i, tx := **range** txs {

*// Validate and mark the remote transaction*

**if** tx == nil {

**return** errResp(***ErrDecode***, **"transaction %d is nil"**, i)

}

p.MarkTransaction(tx.Hash())

}

pm.txpool.AddBatch(txs)

p2p同步得到的交易信息，最终也会缓存在TxPool.pending，等待挖矿流程启动，对交易进行执行。后续的处理与本地提交交易相同。

4：P2P接收新block

p2p同步的block消息源头与交易消息相同：

**func** (pm \*ProtocolManager) handleMsg(p \*peer) error {

*// Read the next message from the remote peer, and ensure it's fully consumed*

msg, err := p.rw.ReadMsg()

**case** msg.Code == ***NewBlockMsg***:

消息传递给fetcher：

**func** (f \*Fetcher) Enqueue(peer string, block \*types.Block) error {

在fetcher的loop()函数做异步处理：

**func** (f \*Fetcher) loop() {

*// Iterate the block fetching until a quit is requested*

fetchTimer := time.NewTimer(0)

completeTimer := time.NewTimer(0)

**for** {

然后尝试将block上链：

**func** (self \*BlockChain) InsertChain(chain types.Blocks) (int, error) {

在上链的过程中，会判断是否由分叉，如果有，则根据TD判断谁是主链，非主链的block会被广播订阅通知：

events = append(events, ChainSideEvent{block})

**go** self.postChainEvents(events, coalescedLogs)

ChainSideEvent消息会被worker收到，并将相应block缓存到possibleUncles，以便挖矿流程启动后，对叔节点进行处理：

**func** (self \*worker) update() {

**for** event := **range** self.events.Chan() {

**case** core.ChainSideEvent:

self.uncleMu.Lock()

self.possibleUncles[ev.Block.Hash()] = ev.Block

self.uncleMu.Unlock()

**九：订阅消息通知体系**

从上面的分析可以看出，订阅消息在内部异步协调中起到很大作用。经过分析，主要的订阅同步功能来自一个订阅同步对象。该同步对象在各个封装实例之间的传递关系，以及分别的发布、订阅情况，整理如下：

