PoW挖矿源码解析

一、共识算法PoW

Proof of Work，工作证明。工作量证明，顾名思义，就是证明你做了多少工作，比特币、以太坊等主流区块链数字货币都是基POW。比特币或者以太坊在Block的生成过程中使用了PoW机制，一个符合要求的Block Hash由N个前导零构成，零的个数取决于网络的难度值。要得到合理的Block Hash需要经过大量尝试计算，计算时间取决于机器的哈希运算速度。当某个节点提供出一个合理的Block Hash值，说明该节点确实经过了大量的尝试计算，当然，并不能得出计算次数的绝对值，因为寻找合理hash是一个概率事件。当节点拥有占全网n%的算力时，该节点即有n/100的概率找到Block Hash。

二、旷工

miner即矿工的意思，矿工要做的工作就是“挖矿”，挖矿就是将一系列最新未封装到块中的交易封装到一个新的区块的过程。

三、挖矿

下面采用代码调试的顺序，来分析以太坊miner.go文件源码内容。整个以太坊挖矿相关的操作都是通过Miner结构体暴露出来的方法：

```
type Miner struct {
    mux *event.TypeMux // 时间锁
    worker *worker // 旷工
    coinbase common.Address // 初始地址
    mining   int32 // 代表挖矿进行中的状态
    eth      Backend // Backend对象，Backend是一个自定义接口封装了所有挖矿所需方法。
    engine   consensus.Engine // 共识引擎
    canStart    int32 // 是否可以挖矿
    shouldStart int32 // 是否应该挖矿
}

事件锁，事件发生时，会有一个TypeMux将时间分派给注册的接收者。接收者可以注册以处理特定类型的事件。在mux结束后调用的任何操作将返回ErrMuxClosed。
共识引擎，获得共识算法的工具对象，以提供后续共识相关操作使用。
```

worker对象 ，因为外部有一个单独的worker.go文件，Miner包含了这个worker对象。上面注释给出的是“旷工”，每个miner都会有一个worker成员对象，可以理解为旷工，他负责全部具体的挖

矿工作流程。

```
type worker struct {
         config *params.ChainConfig
         engine consensus.Engine

mu sync.Mutex

// update loop
mux          *event.TypeMux
txCh         chan core.TxPreEvent
txSub        event.Subscription
chainHeadCh  chan core.ChainHeadEvent
chainHeadSub event.Subscription
chainSideCh  chan core.ChainSideEvent
chainSideSub event.Subscription
wg           sync.WaitGroup

agents map[Agent]struct{} // worker的Agent的map集合属性
recv   chan *Result

eth     Backend
chain   *core.BlockChain
proc    core.Validator
chainDb ethdb.Database

coinbase common.Address
extra    []byte

currentMu sync.Mutex
current   *Work

uncleMu        sync.Mutex
possibleUncles map[common.Hash]*types.Block

unconfirmed *unconfirmedBlocks // 本地挖出的等待确认的块

mining int32
atWork int32
```

}
worker的属性非常多而具体了，下面我们重点分析ChainConfig属性

```
type ChainConfig struct {
    ChainId *big.Int `json:"chainId"` // 主键id，用来防止replay attack重发攻击
```

```
    HomesteadBlock *big.Int `json:"homesteadBlock,omitempty"` // 默认Homestead置为0

    DAOForkBlock   *big.Int `json:"daoForkBlock,omitempty"`   // TheDAO分叉处理
    DAOForkSupport bool     `json:"daoForkSupport,omitempty"` // TheDAO分叉处理

    // EIP150 implements the Gas price changes (https://github.com/ethereum/EIPs/issues
    /150)
    EIP150Block *big.Int    `json:"eip150Block,omitempty"` // EIP150 HF block (nil = no
     fork)
    EIP150Hash  common.Hash `json:"eip150Hash,omitempty"`  // EIP150 HF hash (needed fo
    r header only clients as only gas pricing changed)

    EIP155Block *big.Int `json:"eip155Block,omitempty"` // EIP155 HF block，没有硬分叉置
    为0
    EIP158Block *big.Int `json:"eip158Block,omitempty"` // EIP158 HF block，没有硬分叉置
    为0

    ByzantiumBlock *big.Int `json:"byzantiumBlock,omitempty"` // Byzantium switch block
     (nil = no fork, 0 = already on byzantium)

    // Various consensus engines
    Ethash *EthashConfig `json:"ethash,omitempty"`
    Clique *CliqueConfig `json:"clique,omitempty"`
```

}
ChainConfig就是链的配置文件，ChainConfig中包含了ChainID等属性，其中有很多都是针对以太坊以前发生的问题进行的专门配置。
• ChainId可以预防replay攻击。
• Homestead是以太坊发展蓝图中的一个阶段。这个字段目前我感觉没有太大作用。
• EIPs(Ethereum Improvement Proposals)，是以太坊更新改善的方案，对应后面的数字就是以太坊github源码issue的编号。

一个miner拥有一个worker，一个worker拥有多个agent。Agent接口是定义在Worker.go文件中，Agent 可以注册到worker

```
type Agent interface {
    Work() chan<- *Work
    SetReturnCh(chan<- *Result)
    Stop()
    Start()
    GetHashRate() int64
}
```

这个接口有两个实现CpuAgent和RemoteAgent。这里使用的是CpuAgent，该Agent会完成一个

块的出块工作，同级的多个Agent是竞争关系，最终通过共识算法完成出一个块的工作。

```
type CpuAgent struct {
    mu sync.Mutex // 锁

workCh        chan *Work // Work通道对象
stop          chan struct{} // 结构体通道对象
quitCurrentOp chan struct{} // 结构体通道对象
returnCh      chan<- *Result // Result指针通道

chain  consensus.ChainReader
engine consensus.Engine

isMining int32 // agent是否正在挖矿的标志位
```

}
开始挖矿，首先通过New函数，初始化一个miner实例，

```
func New(eth Backend, config *params.ChainConfig, mux *event.TypeMux, engine consen
sus.Engine) *Miner {
    miner := &Miner{
        eth:      eth,
        mux:      mux,
        engine:   engine,
        worker:   newWorker(config, engine, common.Address{}, eth, mux),
        canStart: 1,
    }
    miner.Register(NewCpuAgent(eth.BlockChain(), engine))
    go miner.update()

    return miner
}
```

在创建miner实例时，会根据Miner结构体成员属性依次赋值，其worker对象需要调用newWorker构造函数，

```
func newWorker(config *params.ChainConfig, engine consensus.Engine, coinbase common
.Address, eth Backend, mux *event.TypeMux) *worker {
    worker := &worker{
        config:         config,
        engine:         engine,
        eth:            eth,
        mux:            mux,
        txCh:           make(chan core.TxPreEvent, txChanSize),// TxPreEvent事件是Tx
```

```
Pool发出的事件，代表一个新交易tx加入到了交易池中，这时候如果work空闲会将该笔交易收进work.t
xs，准备下一次打包进块。
        chainHeadCh:    make(chan core.ChainHeadEvent, chainHeadChanSize),// ChainH
eadEvent事件，代表已经有一个块作为链头，此时work.update函数会监听到这个事件，则会继续挖新
的区块。
        chainSideCh:    make(chan core.ChainSideEvent, chainSideChanSize),// ChainS
ideEvent事件，代表有一个新块作为链的旁支，会被放到possibleUncles数组中，可能称为叔块。
        chainDb:        eth.ChainDb(),// 区块链数据库
        recv:           make(chan *Result, resultQueueSize),
        chain:          eth.BlockChain(), // 链
        proc:           eth.BlockChain().Validator(),
        possibleUncles: make(map[common.Hash]*types.Block),// 存放可能称为下一个块的
叔块数组
        coinbase:       coinbase,
        agents:         make(map[Agent]struct{}),
        unconfirmed:    newUnconfirmedBlocks(eth.BlockChain(), miningLogAtDepth),//
 返回一个数据结构，包括追踪当前未被确认的区块。
    }
    // 注册TxPreEvent事件到tx pool交易池
    worker.txSub = eth.TxPool().SubscribeTxPreEvent(worker.txCh)
    // 注册事件到blockchain
    worker.chainHeadSub = eth.BlockChain().SubscribeChainHeadEvent(worker.chainHead
Ch)
    worker.chainSideSub = eth.BlockChain().SubscribeChainSideEvent(worker.chainSide
Ch)
    go worker.update()

    go worker.wait()
    worker.commitNewWork()

    return worker
}
```

在创建work实例的时候，会有几个比较重要的事件，包括TxPreEvent、ChainHeadEvent、ChainSideEvent，我在上面代码注释中都有了标识。下面看一下开启新线程执行的worker.update(),

```
    case <-self.chainHeadCh:
        self.commitNewWork()

// Handle ChainSideEvent
case ev := <-self.chainSideCh:
    self.uncleMu.Lock()
    self.possibleUncles[ev.Block.Hash()] = ev.Block
    self.uncleMu.Unlock()
```

```
// Handle TxPreEvent
case ev := <-self.txCh:
```

由于源码较长，我只展示了一部分，我们知道update方法是用来监听处理上面谈到的三个事件即可。下面再看一下worker.wait()方法，

```
func (self *worker) wait() {
    for {
        mustCommitNewWork := true
        for result := range self.recv {
            atomic.AddInt32(&self.atWork, -1)

    if result == nil {
        continue
    }
    block := result.Block
    work := result.Work

    // Update the block hash in all logs since it is now available and not when the
    // receipt/log of individual transactions were created.
    for _, r := range work.receipts {
        for _, l := range r.Logs {
            l.BlockHash = block.Hash()
        }
    }
    for _, log := range work.state.Logs() {
        log.BlockHash = block.Hash()
    }
    stat, err := self.chain.WriteBlockAndState(block, work.receipts, work.state)
    if err != nil {
        log.Error("Failed writing block to chain", "err", err)
        continue
    }
    // 检查是否是标准块，写入交易数据。
    if stat == core.CanonStatTy {
        // 受ChainHeadEvent事件的影响。
        mustCommitNewWork = false
    }
    // 广播一个块声明插入链事件NewMinedBlockEvent
    self.mux.Post(core.NewMinedBlockEvent{Block: block})
    var (
        events []interface{}
        logs   = work.state.Logs()
    )
    events = append(events, core.ChainEvent{Block: block, Hash: block.Hash(), Logs:
 logs})
    if stat == core.CanonStatTy {
```

```
            events = append(events, core.ChainHeadEvent{Block: block})
        }
        self.chain.PostChainEvents(events, logs)

        // 将处理中的数据插入到区块中，等待确认
        self.unconfirmed.Insert(block.NumberU64(), block.Hash())

        if mustCommitNewWork {
            self.commitNewWork() // 多次见到，顾名思义，就是提交新的work
        }
    }
```

wait方法比较长，但必须展示出来的原因是它包含了重要的具体写入块的操作，具体内容请看上面代码中的注释。
通过New方法来初始化创建一个miner实例，入参包括Backend对象，ChainConfig对象属性集合，事件锁，以及指定共识算法引擎，返回一个Miner指针。方法体中对miner对象进行了组装赋值，并且调用了方法NewCpuAgent创建agent的一个实例然后注册到该miner上来，启动一个单独线程执行miner.update()，我们先来看NewCpuAgent方法：

```
func NewCpuAgent(chain consensus.ChainReader, engine consensus.Engine) *CpuAgent {
    miner := &CpuAgent{
        chain:  chain,
        engine: engine,
        stop:   make(chan struct{}, 1),
        workCh: make(chan *Work, 1),
    }
    return miner
}
```

通过NewCpuAgent方法，先组装一个CpuAgent，赋值ChainReader，共识引擎，stop结构体，work通道，然后将这个CpuAgent实例赋值给miner并返回miner。然后让我们回到miner.update()方法：
// update方法可以保持对下载事件的监听，请了解这是一段短型的update循环。

```
func (self *Miner) update() {
    // 注册下载开始事件，下载结束事件，下载失败事件。
    events := self.mux.Subscribe(downloader.StartEvent{}, downloader.DoneEvent{}, d
ownloader.FailedEvent{})
out:
    for ev := range events.Chan() {
        switch ev.Data.(type) {
        case downloader.StartEvent:
            atomic.StoreInt32(&self.canStart, 0)
```

```
            if self.Mining() {// 开始下载对应Miner操作Mining。
                self.Stop()
                atomic.StoreInt32(&self.shouldStart, 1)
                log.Info("Mining aborted due to sync")
            }
        case downloader.DoneEvent, downloader.FailedEvent: // 下载完成和失败都走相同
的分支。
            shouldStart := atomic.LoadInt32(&self.shouldStart) == 1

    atomic.StoreInt32(&self.canStart, 1)
    atomic.StoreInt32(&self.shouldStart, 0)
    if shouldStart {
        self.Start(self.coinbase) // 执行Miner的start方法。
    }
    // 处理完以后要取消订阅
    events.Unsubscribe()
    // 跳出循环，不再监听
    break out
}
```

然后我们来看miner的Mining方法，
// 如果miner的mining属性大于1即返回ture，说明正在挖矿中。

```
func (self *Miner) Mining() bool {
    return atomic.LoadInt32(&self.mining) > 0
}
```

再来看Miner的start方法，它是属于Miner指针实例的方法，首字母大写代表可以被外部所访问，传入一个地址。

```
func (self *Miner) Start(coinbase common.Address) {
    atomic.StoreInt32(&self.shouldStart, 1)
    self.worker.setEtherbase(coinbase)
    self.coinbase = coinbase

if atomic.LoadInt32(&self.canStart) == 0 {
    log.Info("Network syncing, will start miner afterwards")
    return
}
atomic.StoreInt32(&self.mining, 1)

log.Info("Starting mining operation")
self.worker.start()
self.worker.commitNewWork()
```

}
关键代码在self.worker.start()和self.worker.commitNewWork()。先来说worker.start()方法。

```
func (self *worker) start() {
    self.mu.Lock()
    defer self.mu.Unlock()

atomic.StoreInt32(&self.mining, 1)

// spin up agents
for agent := range self.agents {
    agent.Start()
}
```

}
worker.start()实际上遍历启动了它所有的agent。上面提到了，这里走的是CpuAgent的实现。

```
func (self *CpuAgent) Start() {
    if !atomic.CompareAndSwapInt32(&self.isMining, 0, 1) {
        return // agent already started
    }
    go self.update()
}
```

启用一个单独线程去执行CpuAgent的update()方法。update方法与上面miner.update十分相似。

```
func (self *CpuAgent) update() {
out:
    for {
        select {
        case work := <-self.workCh:
            self.mu.Lock()
            if self.quitCurrentOp != nil {
                close(self.quitCurrentOp)
            }
            self.quitCurrentOp = make(chan struct{})
            go self.mine(work, self.quitCurrentOp)
            self.mu.Unlock()
        case <-self.stop:
            self.mu.Lock()
            if self.quitCurrentOp != nil {
                close(self.quitCurrentOp)
                self.quitCurrentOp = nil
            }
```

```
            self.mu.Unlock()
            break out
        }
    }
}
```

out: break out ， 跳出for循环，for循环不断监听self信号，当监测到self停止时，则调用关闭操作代码，并直接挑出循环监听，函数退出。
通过监测CpuAgent的workCh通道，是否有work工作信号进入，如果有agent则开始挖矿，期间要上锁，启用一个单独线程去执行CpuAgent的mine方法。

```
func (self *CpuAgent) mine(work *Work, stop <-chan struct{}) {
    if result, err := self.engine.Seal(self.chain, work.Block, stop); result != nil
 {
        log.Info("Successfully sealed new block", "number", result.Number(), "hash"
, result.Hash())
        self.returnCh <- &Result{work, result}
    } else {
        if err != nil {
            log.Warn("Block sealing failed", "err", err)
        }
        self.returnCh <- nil
    }
}
```

执行到这里，可以看到是调用的CpuAgent的共识引擎的块封装函数Seal来执行具体的挖矿操作。，回到上面继续分析另一个比较重要的方法self.worker.commitNewWork()。
commitNewWork方法源码比较长，这里就不粘贴出来了，这个方法主要的工作是为新块准备基本数据，包括header，txs，uncles等。

四、总结

关于以太坊挖矿的源码粗略的分析就到这，粗略的意思是对于我自己的标准来讲，我并未逐一介绍每个流程控制，还有每行代码的具体意思，只是大致提供了一种看源码的路线，一条一条的进入，再收紧退回，最终完成了一个闭环，让我们对以太坊挖矿的一些具体操作有了了解。这部分源码主要工作是对交易数据池的维护，块数据组织，各种事件的监听处理，miner-worker-agent的分工关系。