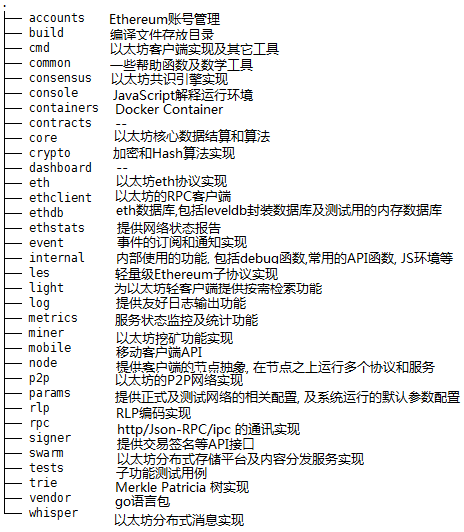
**以太坊源码分析**

**一、介绍**

以太坊是2015年由Vitalik Buterin 通过众筹融资推出的首个内置图灵完备编程语言的通用区块链协议和全球共享分布式应用平台。理论上，用户可以按照自己的意愿在以太坊平台上创建并执行任意复杂的操作，从而高效、快速地开发出包括加密货币在内的多种去中心化区块链应用。以太坊的出现为智能合约注入了新的活力，并改变了区块链技术地应用格局，使其不再局限于比特币等加密货币涉及的支付领域，而开始有机会应用到各方面。

**二、源码分析**

本文以以太坊的官方go语言版本实现go-ethereum为目标，分析其源码实现。



**三、内容概念**

以太坊在整体上可看做是一个基于交易的状态机：起始于一个创世状态，然后随着交易的执行，状态逐步改变一直到最终状态，这个最终状态就是以太坊世界的权威版本。

1.账户交易

以太坊中引入了账户的概念以取代比特币UTXO模型，账户分为外部账户和合约账户两类，两类账户都具有与之关联的账户状态和账户地址，都可以存储以太坊专用加密货币（以太币），区别在于外部账户由用户私钥控制，没有代码与之关联；合约账户则由合约代码控制，有代码与之关联。

当一个账户向另一个账户发送一笔被签名的消息数据包时，就产生了一笔交易。

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:"-"`

}

1. 区块

一个区块包含了一系列的交易，矿工节点收集本地发起的及网络中其它节点广播的新交易，验证交易的有效性，然后将它们打包到一个原始区块中，最后通过挖矿得到一个数学机制的“工作量证明”写到该区块，从而得到一个新的合法区块，广播到网络中，在其它矿工验证区块有效后添加到主链上。

type Block struct {

    header       \*Header

    uncles       []\*Header

    transactions Transactions

// caches

    hash atomic.Value

    size atomic.Value

// Td is used by package core to store the total difficulty

    // of the chain up to and including the block.

    td \*big.Int

// These fields are used by package eth to track

    // inter-peer block relay.

    ReceivedAt   time.Time

    ReceivedFrom interface{}}

1. 区块链

区块链代表给定了一个带有起源块的数据库的规范链。区块链管理链导入、恢复和链重组。 将块导入到区块链是根据两阶段验证器定义的规则集进行的。块的处理是使用处理所包含事务的处理器来完成的。状态的验证在验证器的第二部分完成。失败会导致导入中止。 区块链还帮助从数据库中包含的任何\*\*链以及代表规范链的块中返回块。重要的是要注意，获取块可以返回任何块，并且不需要包含在规范块中，因为获取块字节数总是代表非规范链。

type BlockChain struct {

    chainConfig \*params.ChainConfig // Chain & network configuration

    cacheConfig \*CacheConfig        // Cache configuration for pruning

    db     ethdb.Database // Low level persistent database to store final content in

    triegc \*prque.Prque   // Priority queue mapping block numbers to tries to gc

    gcproc time.Duration  // Accumulates canonical block processing for trie dumping

 hc            \*HeaderChain

    rmLogsFeed    event.Feed

    chainFeed     event.Feed

    chainSideFeed event.Feed

    chainHeadFeed event.Feed

    logsFeed      event.Feed

    scope         event.SubscriptionScope

    genesisBlock  \*types.Block

mu      sync.RWMutex // global mutex for locking chain operations

    chainmu sync.RWMutex // blockchain insertion lock

    procmu  sync.RWMutex // block processor lock

checkpoint       int          // checkpoint counts towards the new checkpoint

    currentBlock     atomic.Value // Current head of the block chain

    currentFastBlock atomic.Value // Current head of the fast-sync chain (may be above the bloc stateCache   state.Database // State database to reuse between imports (contains state cache)

    bodyCache    \*lru.Cache     // Cache for the most recent block bodies

    bodyRLPCache \*lru.Cache     // Cache for the most recent block bodies in RLP encoded format

    blockCache   \*lru.Cache     // Cache for the most recent entire blocks

    futureBlocks \*lru.Cache     // future blocks are blocks added for later processing

quit    chan struct{} // blockchain quit channel

    running int32         // running must be called atomically

    // procInterrupt must be atomically called

    procInterrupt int32          // interrupt signaler for block processing

    wg            sync.WaitGroup // chain processing wait group for shutting down

engine    consensus.Engine

    processor Processor // block processor interface

    validator Validator // block and state validator interface

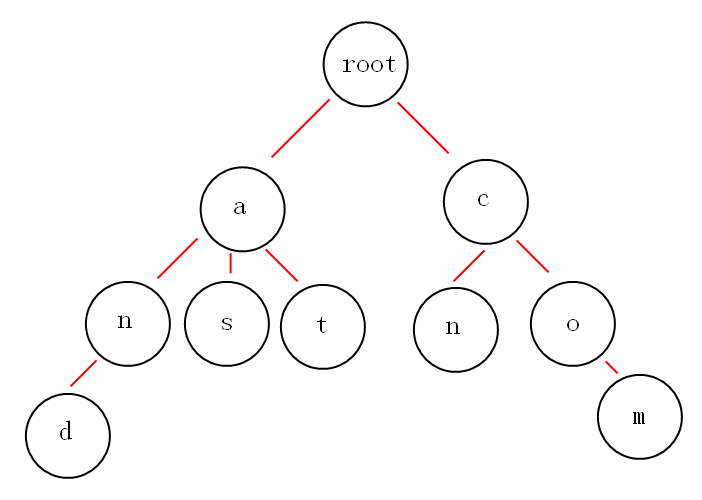
    vmConfig  vm.Config

badBlocks \*lru.Cache // Bad block cache

}

四、

MPT

MPT，即Merkel Patricia Tree，是混合了 Merkle Tree和Patricia Tree的产物。它提供了可以持久化的任意长度的二进制Key-Value之间的映射关系。以太坊区块头部中有3个字段是和MPT树相关的：账户状态(Header.Root，即stateRoot)、交易集根(Header.TxHash，即transactionsRoot)、收据根(Header.ReceiptHash，即receiptsRoot)。

Trie树也叫作Radix树，为了提高效率，以太坊在实现上对其做了一些改进。在一般的radix树中，key是从树根到对应value得真实的路径。即从根节点开始，key中的每个字符会标识走那个子节点从而到达相应value。Value被存储在叶子节点，是每条路径的终止。假如key来自一个包含N个字符的字母表，那么树中的每个节点都可能会有多达N个孩子，树的最大深度是key的最大长度。

// Trie is a Merkle Patricia Trie.

// The zero value is an empty trie with no database.

// Use New to create a trie that sits on top of a database.

//

// Trie is not safe for concurrent use.

type Trie struct {

    db           \*Database

    root         node

    originalRoot common.Hash

    // Cache generation values.

    // cachegen increases by one with each commit operation.

    // new nodes are tagged with the current generation and unloaded

    // when their generation is older than than cachegen-cachelimit.

    cachegen, cachelimit uint16

}

以太坊对Radix树的实现做了很多改进。

首先，为了保证树的加密安全，每个节点通过他的hash被引用，而非32bit或64bit的内存地址，即树的Merkle部分是一个节点的确定性加密的hash。一个非叶节点存储在leveldb关系型数据库中，数据库中的key是节点的RLP编码的sha3哈希，value是节点的RLP编码。

想要获得一个非叶节点的子节点，只需要根据子节点的hash访问数据库获得节点的RLP编码，然后解码就行了。

通过这种模式，根节点就成为了整个树的加密签名，如果一颗给定trie的跟hash是公开的，那么所有人都可以提供一种证明，通过提供每步向上的路径证明特定的key是否含有给定的值。

func concat(s1 []byte, s2 ...byte) []byte {

    r := make([]byte, len(s1)+len(s2))

    copy(r, s1)

    copy(r[len(s1):], s2)

    return r

}

func (t \*Trie) resolve(n node, prefix []byte) (node, error) {

    if n, ok := n.(hashNode); ok {

        return t.resolveHash(n, prefix)

    }

    return n, nil

}

func (t \*Trie) resolveHash(n hashNode, prefix []byte) (node, error) {

    cacheMissCounter.Inc(1)

  hash := common.BytesToHash(n)

    if node := t.db.node(hash, t.cachegen); node != nil {

        return node, nil

    }

    return nil, &MissingNodeError{NodeHash: hash, Path: prefix}

}

// Root returns the root hash of the trie.

// Deprecated: use Hash instead.

func (t \*Trie) Root() []byte { return t.Hash().Bytes() }

// Hash returns the root hash of the trie. It does not write to the

// database and can be used even if the trie doesn't have one.

func (t \*Trie) Hash() common.Hash {

    hash, cached, \_ := t.hashRoot(nil, nil)

    t.root = cached

    return common.BytesToHash(hash.(hashNode))

}

// Commit writes all nodes to the trie's memory database, tracking the internal

// and external (for account tries) references.

func (t \*Trie) Commit(onleaf LeafCallback) (root common.Hash, err error) {

    if t.db == nil {

        panic("commit called on trie with nil database")

    }

    hash, cached, err := t.hashRoot(t.db, onleaf)

    if err != nil {

        return common.Hash{}, err

    }

    t.root = cached

    t.cachegen++

    return common.BytesToHash(hash.(hashNode)), nil

}

func (t \*Trie) hashRoot(db \*Database, onleaf LeafCallback) (node, node, error) {

    if t.root == nil {

        return hashNode(emptyRoot.Bytes()), nil, nil

    }

    h := newHasher(t.cachegen, t.cachelimit, onleaf)

    defer returnHasherToPool(h)

    return h.hash(t.root, db, true)

}

**四、操作流程**

用户只能通过外部账户在以太坊中发起交易，交易可以包含二进制交易负载数据和以太币，交易过程中可能产生一系列消息调用。

1. 启动

go-ethereum编译出来的官方客户端程序 geth，提供了庞大的子命令和命令行参数，分别控制节点的运行模式、挖矿参数、网络参数、交易及调试参数等，这些选项由geth处理后修改相应的默认配置项，控制geth节点的行为。我们先从geth的启动流程开始分析，了解节点运行所需的核心组件及相互关系。

geth使用 urfave/cli 库封装了命令行参数解析过程，抽象出Flags/Commands这些模块，用户只需要提供一些模块的配置即可。导入包以后调用 cli.NewApp() 创建一个实例，然后调用 Run() 方法执行 app.Action 入口函数。

geth的启动入口在main包 cmd/geth/main.go中，默认首先执行包体的 init()函数，为app指定Action(geth)、Commands、Flags、Copyright等信息, 然后在main()中调用app.Run()正式启动，进入geth(…)函数：

func init() {

    // Initialize the CLI app and start Geth

    app.Action = geth

    app.HideVersion = true // we have a command to print the version

    app.Copyright = "Copyright 2013-2018 The go-ethereum Authors"

    app.Commands = []cli.Command{

        // See chaincmd.go:

        initCommand,

        importCommand,

        exportCommand,

        importPreimagesCommand,

        exportPreimagesCommand,

        copydbCommand,

        removedbCommand,

        dumpCommand,

        // See monitorcmd.go:

        monitorCommand,

        // See accountcmd.go:

        accountCommand,

        walletCommand,

        // See consolecmd.go:

        consoleCommand,

        attachCommand,

        javascriptCommand,

        // See misccmd.go:

        makecacheCommand,

        makedagCommand,

        versionCommand,

        bugCommand,

        licenseCommand,

        // See config.go

        dumpConfigCommand,

    }

geth(…)函数所做的事看起来很简单，就是先创建一个Node，然后启动节点运行，直到Node退出，程序结束。

现在我们有了配置好的Node节点，和Eth、Dashboard、Whisper服务相关的配置，现在调用startNode(…)在节点上启动相应的服务。

startNode的主要调用在utils.StartNode(stack)中，stack是上面创建的Node节点，其它的几个部分代码是账户密码解锁、设置钱包打开关闭事件监听及启动挖矿（如果指定了—mine）。

func (n \*Node) Start() error {

    n.lock.Lock()

    defer n.lock.Unlock()

    // Short circuit if the node's already running

    if n.server != nil {

        return ErrNodeRunning

    }

    if err := n.openDataDir(); err != nil {

        return err

    }

    // Initialize the p2p server. This creates the node key and

    // discovery databases.

    n.serverConfig = n.config.P2P

    n.serverConfig.PrivateKey = n.config.NodeKey()

    n.serverConfig.Name = n.config.NodeName()

    n.serverConfig.Logger = n.log

    if n.serverConfig.StaticNodes == nil {

        n.serverConfig.StaticNodes = n.config.StaticNodes()

    }

    if n.serverConfig.TrustedNodes == nil {

        n.serverConfig.TrustedNodes = n.config.TrustedNodes()

    }

    if n.serverConfig.NodeDatabase == "" {

        n.serverConfig.NodeDatabase = n.config.NodeDB()

    }

    running := &p2p.Server{Config: n.serverConfig}

    n.log.Info("Starting peer-to-peer node", "instance", n.serverConfig.Name)

    // Otherwise copy and specialize the P2P configuration

    services := make(map[reflect.Type]Service)

    for \_, constructor := range n.serviceFuncs {

        // Create a new context for the particular service

        ctx := &ServiceContext{

            config:         n.config,

            services:       make(map[reflect.Type]Service),

            EventMux:       n.eventmux,

            AccountManager: n.accman,

        }

        for kind, s := range services { // copy needed for threaded access

            ctx.services[kind] = s

        }

        // Construct and save the service

        service, err := constructor(ctx)

        if err != nil {

            return err

        }

        kind := reflect.TypeOf(service)

        if \_, exists := services[kind]; exists {

            return &DuplicateServiceError{Kind: kind}

        }

        services[kind] = service

    }

    // Gather the protocols and start the freshly assembled P2P server

    for \_, service := range services {

        running.Protocols = append(running.Protocols, service.Protocols()...)

    }

    if err := running.Start(); err != nil {

        return convertFileLockError(err)

    }

    // Start each of the services

    started := []reflect.Type{}

    for kind, service := range services {

        // Start the next service, stopping all previous upon failure

        if err := service.Start(running); err != nil {

            for \_, kind := range started {

                services[kind].Stop()

            }

            running.Stop()

 return err

        }

        // Mark the service started for potential cleanup

        started = append(started, kind)

    }

    // Lastly start the configured RPC interfaces

    if err := n.startRPC(services); err != nil {

        for \_, service := range services {

            service.Stop()

        }

        running.Stop()

        return err

    }

    // Finish initializing the startup

    n.services = services

    n.server = running

    n.stop = make(chan struct{})

    return nil

}

作为核心的*eth.Ethereum*，它的Start()正式宣告了节点的启动完成。

type Ethereum struct {

    config      \*Config

    chainConfig \*params.ChainConfig

    // Channel for shutting down the service

    shutdownChan chan bool // Channel for shutting down the Ethereum

    // Handlers

    txPool          \*core.TxPool

    blockchain      \*core.BlockChain

    protocolManager \*ProtocolManager

    lesServer       LesServer

    // DB interfaces

    chainDb ethdb.Database // Block chain database

    eventMux       \*event.TypeMux

    engine         consensus.Engine

    accountManager \*accounts.Manager

    bloomRequests chan chan \*bloombits.Retrieval // Channel receiving bloom data retrieval requests

    bloomIndexer  \*core.ChainIndexer             // Bloom indexer operating during block imports

    APIBackend \*EthAPIBackend

    miner     \*miner.Miner

    gasPrice  \*big.Int

    etherbase common.Address

    networkID     uint64

    netRPCService \*ethapi.PublicNetAPI

    lock sync.RWMutex // Protects the variadic fields (e.g. gas price and etherbase)

}

*eth.Ethereum*类其实现以太坊全节点功能模块的容器，它内含管理动态变化交易的交易池TxPool，完成ETH协议交互的ProtocolManager，有执行挖矿的Mine, 有维护区块链数据的BlockChain, 而它需要为这些功能模块提供正确运行所需的ETH相关配置、链配置、数据库接口、共识引擎、账户管理、gasPrice等。它的创建过程就是创建上述核心模块并初始化的过程：

(1). 打开数据目录下的 $DATADIR/chaindata 数据库

(2). 从数据库里读出chain配置和创世块的哈希值

(3). 创建一个新的Ethereum结构，除了从入参config和ctx取出需要的变量config、accountManager、gasPrice等进行传递赋值，还创建了共识引擎、BloomIndexer

(4). core.NewBlockChain(…)从数据库“chaindata”中加载创世块、已知的最新块和对应的TD（Total Difficulty，用来确定最重的权威主链）

(5). 启动bloomIndexer服务

(6). 对于本地交易如果指定了交易池的Journal选项，则会将本地交易持久化到数据库

(7). 创建交易池TxPool，交易池中存放本地交易和从网络接收到的交易

(8). 创建ProtocolManager

(9). 创建矿工Miner

(10). 创建API处理服务

(11). 返回创建的Ethereum指针

Now, eth.Ethereum.Start(..):

(1). 启动bloom位数据获取的协程提供服务

(2). 提供RPC接口中的network相关命令的处理函数

(3). 启动protocolManager

(4). 如果提供LES请求支持的话启动lesServer

如果命令行指定了开启挖矿, 或者是开发模式, 就启动挖矿. 如果没有指定的话, 还有一种方式启动挖矿, 就是在终端console里输入命令: web3.miner.start().

它一共开启了4个go协程:

(1). 向TxPool订阅新交易事件, 然后开启BroadcastLoop(), 一旦有新交易, 它就会把该交易通知给不包含该交易的节点;

(2). 订阅挖出新区块的事件, 然后开启minedBroadcastLoop(), 一旦有新区块, 它就会把该区块发送给网络中的其它节点, 告诉它们本节点有该区块

(3). 开启 syncer()同步区块

(4). 开启txsyncLoop(), 向新连接的节点发送本交易池中Pending的交易

1. 同步区块

在代码中定义了三种同步模式：

const (

    FullSync  SyncMode = iota // Synchronise the entire blockchain history from full blocks

    FastSync                  // Quickly download the headers, full sync only at the chain head

    LightSync                 // Download only the headers and terminate afterwards

)

以新加入网络的节点A和挖矿节点B为例：

1 两个节点先hadeshake同步下各自的genesis、td、head等信息

2 连接成功后，节点B发送TxMsg消息把自己的txpool中的tx同步给节点A

    然后各自循环监听对方的消息

3 节点A此时使用fast sync同步数据，依次发送GetBlockHeadersMsg、GetBlockBodiesMsg、GetReceiptsMsg、GetNodeDataMsg获取block header、block body、receipt和state数据

4 节点B对应的返回BlockHeadersMsg、BlockBodiesMsg、ReceiptsMsg、NodeDataMsg

5 节点A收到数据把header和body组成block存入自己的leveldb数据库，一并存入receipts和state数据

6 节点B挖出block后会向A同步区块，发送NewBlockMsg或者NewBlockHashesMsg(取决于节点A位于节点B的节点列表位置），如果是NewBlockMsg，那么节点A直接验证完存入本地；如果是NewBlockHashesMsg节点A会交给fetcher去获取header和body，然后再组织成block存入本地

1. 广播交易

func (pm \*ProtocolManager) Start(maxPeers int) {

    pm.maxPeers = maxPeers

    // broadcast transactions

    pm.txsCh = make(chan core.NewTxsEvent, txChanSize)

    pm.txsSub = pm.txpool.SubscribeNewTxsEvent(pm.txsCh)

    go pm.txBroadcastLoop()

    // broadcast mined blocks

    pm.minedBlockSub = pm.eventMux.Subscribe(core.NewMinedBlockEvent{})

    go pm.minedBroadcastLoop()

    // start sync handlers

    go pm.syncer()

    go pm.txsyncLoop()

}

当自身节点收到了客户端的交易或者收到其他节点发来的交易加到txpool中时，Send订阅消息：core/tx\_pool.go

func (pool \*TxPool) add(tx \*types.Transaction, local bool) (bool, error) {

    // We've directly injected a replacement transaction, notify subsystems

    go pool.txFeed.Send(TxPreEvent{tx})

}

这样txBroadcastLoop中就读到了tx：

func (self \*ProtocolManager) txBroadcastLoop() {

    for {

        select {

        case event := <-self.txCh:

            self.BroadcastTx(event.Tx.Hash(), event.Tx)

        // Err() channel will be closed when unsubscribing.

        case <-self.txSub.Err():

            return

        }

    }

}

1. 广播区块

当打包新区块成功时，会Post这个event：miner/worker.go

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.WriteBlockWithState(block, work.receipts, work.state)

            if err != nil {

                log.Error("Failed writing block to chain", "err", err)

                continue

            }

            // check if canon block and write transactions

            if stat == core.CanonStatTy {

                // implicit by posting ChainHeadEvent

                mustCommitNewWork = false

            }

            // Broadcast the block and announce chain insertion event

            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)

            // Insert the block into the set of pending ones to wait for confirmations

            self.unconfirmed.Insert(block.NumberU64(), block.Hash())

            if mustCommitNewWork {

                self.commitNewWork()

            }

        }

    }

}

这里收到新的区块后，调用BroadcastBlock广播区块，看代码是调用了两次，一次第二个参数为true，一次为fasle

func (pm \*ProtocolManager) BroadcastBlock(block \*types.Block, propagate bool) {

    hash := block.Hash()

    peers := pm.peers.PeersWithoutBlock(hash)

    // If propagation is requested, send to a subset of the peer

    if propagate {

        // Calculate the TD of the block (it's not imported yet, so block.Td is not valid)

        var td \*big.Int

        if parent := pm.blockchain.GetBlock(block.ParentHash(), block.NumberU64()-1); parent != nil {

            td = new(big.Int).Add(block.Difficulty(), pm.blockchain.GetTd(block.ParentHash(), block.NumberU64()-1))

        } else {

            log.Error("Propagating dangling block", "number", block.Number(), "hash", hash)

            return

        }

        // Send the block to a subset of our peers

        transfer := peers[:int(math.Sqrt(float64(len(peers))))]

        for \_, peer := range transfer {

            peer.SendNewBlock(block, td)

        }

        log.Trace("Propagated block", "hash", hash, "recipients", len(transfer), "duration", common.PrettyDuration(time.Since(block.ReceivedAt)))

        return

    }

    // Otherwise if the block is indeed in out own chain, announce it

    if pm.blockchain.HasBlock(hash, block.NumberU64()) {

        for \_, peer := range peers {

            peer.SendNewBlockHashes([]common.Hash{hash}, []uint64{block.NumberU64()})

        }

        log.Trace("Announced block", "hash", hash, "recipients", len(peers), "duration", common.PrettyDuration(time.Since(block.ReceivedAt)))

    }

}

**五、作用**

目前，以太坊已成为全球最大和最完善的共享分布式平台之一，这其中既有其优于比特币的特性带来的竞争红利，也有包括摩根大通，微软在内的大型企业组成的以太坊企业联盟带来的正面效果。

作为引领区块链2.0模式的区块链项目，以太坊致力于构建去中心化、五中心化所有权、高容错、零停机、防篡改、可执行点对点智能合约的全球共享基础设施--世界计算机。以太坊区块链数据库由众多连接到网络节点共同维护和更新，每个网络节点都运行着以太坊虚拟机并执行相同的指令，任何人可以上传、执行共享以太坊上的程序及数据，全球成千上万计算机组成的鲁棒性极强，去中心化的共识网络和其他区块链密码学和经济刺激手段保证计算安全性和程序有效执行，最终实现去中心化互联网，将区块链技术所具有的去中心化，开放，安全等特点引入到所有能被计算的领域。