第八周周报

汪子龙

本周主要完成软件杯的项目优化，同时学习了 go ethereum 的源码，了解项目的整体结构，对几个重要的组成部分进行深入理解。

一、项目整体

项目是按照功能模块划分的目录，其中每个文件夹的功能如下：

文件夹	功能
accounts	实现了以太坊客户端的钱包和账户管理
build	编译和构建的一些脚本和配置
cmd	提供命令行工具
common	公共的工具类
consensus	实现共识算法
console	console 对象
contracts	实现部分合约
core	核心数据结构和算法
crypto	加密和哈希算法
eth	以太坊网络协议
ethclient	以太坊 rpc 客户端
ethdb	以太坊数据库相关功能
ethstats	以太坊网络状态报告
event	进程内部的事件发布和订阅
graphql	节点数据的 graphql 接口
internal	编译、接口、调试等内部实现
les	轻量级协议子集
light	为轻量级客户端提供检索功能
log	日志
metrics	磁盘计数器
miner	区块创建和挖矿相关
mobile	移动端
node	以太坊的节点（全节点/轻量级节点）
p2p	p2p 网络协议
params	参数
rlp	以太坊序列化处理方法
rpc	远程调用
signer	交易签名方法
swarm	swarm 网络处理
tests	用于测试
trie	实现了以太坊中一种非常重要的数据结构 Merkle Patricia Tries
whisper	whisper 节点的协议

二、重点数据结构

1.Hash

common/types.go

由 Keccak-256 算法计算的 32 字节哈希值。

```go
// Hash represents the 32 byte Keccak256 hash of arbitrary data.
type Hash [HashLength]byte
```

2.Address

common/types.go

以太坊账户的 20 字节地址。

```go
// Address represents the 20 byte address of an Ethereum account.
type Address [AddressLength]byte
```

3.Account

accounts/accounts.go

以太坊账户，包含账户地址和可选的后端资源定位 url。

```go
// Account represents an Ethereum account located at a specific location defined
// by the optional URL field.
type Account struct {
    Address common.Address `json:"address"` // Ethereum account address derived from the key
    URL     URL            `json:"url"`     // Optional resource locator within a backend
}
```

4.Header

core/types/block.go

定义了以太坊的区块头结构，其中各字段含义如下：

字段	含义
ParentHash	父节点的哈希
UncleHash	叔节点的哈希
Coinbase	区块矿工的地址
TxHash	交易的哈希
ReceiptHash	交易回执的哈希
Difficulty	区块难度
Number	区块高度
GasLimit	gas 限度
GasUsed	消耗总 gas
Time	上链时间

```go
// Header represents a block header in the Ethereum blockchain.
type Header struct {
  ParentHash  common.Hash    `json:"parentHash"       gencodec:"required"`
  UncleHash   common.Hash    `json:"sha3Uncles"       gencodec:"required"`
  Coinbase    common.Address `json:"miner"            gencodec:"required"`
  Root        common.Hash    `json:"stateRoot"        gencodec:"required"`
  TxHash      common.Hash    `json:"transactionsRoot" gencodec:"required"`
  ReceiptHash common.Hash    `json:"receiptsRoot"     gencodec:"required"`
  Bloom       Bloom           `json:"logsBloom"        gencodec:"required"`
  Difficulty  *big.Int       `json:"difficulty"       gencodec:"required"`
  Number      *big.Int       `json:"number"           gencodec:"required"`
  GasLimit    uint64          `json:"gasLimit"         gencodec:"required"`
  GasUsed     uint64          `json:"gasUsed"          gencodec:"required"`
  Time        uint64          `json:"timestamp"        gencodec:"required"`
  Extra       []byte          `json:"extraData"        gencodec:"required"`
  MixDigest   common.Hash    `json:"mixHash"`
  Nonce       BlockNonce      `json:"nonce"`
}
```

5.Block

core/types/block.go

定义以太坊的一个区块，包含区块的区块头、叔节点、所有交易以及其他信息。

```go
// Block represents an entire block in the Ethereum blockchain.
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{}
}
```

6.Transaction

core/types/transaction.go

定义了链上的交易结构，包含交易详细信息、时间、哈希、发起人等信息。

其中交易详细信息（Nonce、gasPice、gas、接收账户地址、交易总额等）存储在一个 txdata 对象中，结构定义如下：

```go
type Transaction struct {
  data txdata    // Consensus contents of a transaction
  time time.Time // Time first seen locally (spam avoidance)

  // caches
  hash atomic.Value
  size atomic.Value
  from atomic.Value
}
```

```go
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:"-"`
}
```

三、共识算法分析

consensus/consensus.go 中定义了链读取器（ChainReader）和共识引擎（Engine）。

ChainReader 便于查询链上的区块和区块头等信息，结构定义如下：

```go
// ChainHeaderReader defines a small collection of methods needed to access the local
// blockchain during header verification.
type ChainHeaderReader interface {
  // Config retrieves the blockchain's chain configuration.
  Config() *params.ChainConfig

  // CurrentHeader retrieves the current header from the local chain.
  CurrentHeader() *types.Header

  // GetHeader retrieves a block header from the database by hash and number.
  GetHeader(hash common.Hash, number uint64) *types.Header

  // GetHeaderByNumber retrieves a block header from the database by number.
  GetHeaderByNumber(number uint64) *types.Header

  // GetHeaderByHash retrieves a block header from the database by its hash.
  GetHeaderByHash(hash common.Hash) *types.Header
}

// ChainReader defines a small collection of methods needed to access the local
// blockchain during header and/or uncle verification.
type ChainReader interface {
  ChainHeaderReader

  // GetBlock retrieves a block from the database by hash and number.
  GetBlock(hash common.Hash, number uint64) *types.Block
}
```

Engine 是各种共识算法的接口，定义了共识的各种函数。

```go
// Engine is an algorithm agnostic consensus engine.
type Engine interface {
  // Author retrieves the Ethereum address of the account that minted the given
  // block, which may be different from the header's coinbase if a consensus
  // engine is based on signatures.
  Author(header *types.Header) (common.Address, error)

  // VerifyHeader checks whether a header conforms to the consensus rules of a
  // given engine. Verifying the seal may be done optionally here, or explicitly
  // via the VerifySeal method.
  VerifyHeader(chain ChainHeaderReader, header *types.Header, seal bool) error

  // VerifyHeaders is similar to VerifyHeader, but verifies a batch of headers
  // concurrently. The method returns a quit channel to abort the operations and
  // a results channel to retrieve the async verifications (the order is that of
  // the input slice).
  VerifyHeaders(chain ChainHeaderReader, headers []*types.Header, seals []bool) (chan<- struct{}, <-chan error)

  // VerifyUncles verifies that the given block's uncles conform to the consensus
  // rules of a given engine.
  VerifyUncles(chain ChainReader, block *types.Block) error

  // VerifySeal checks whether the crypto seal on a header is valid according to
  // the consensus rules of the given engine.
  VerifySeal(chain ChainHeaderReader, header *types.Header) error

  // Prepare initializes the consensus fields of a block header according to the
  // rules of a particular engine. The changes are executed inline.
  Prepare(chain ChainHeaderReader, header *types.Header) error

  // Finalize runs any post-transaction state modifications (e.g. block rewards)
  // but does not assemble the block.
  //
  // Note: The block header and state database might be updated to reflect any
  // consensus rules that happen at finalization (e.g. block rewards).
  Finalize(chain ChainHeaderReader, header *types.Header, state *state.StateDB, txs []*types.Transaction,
    uncles []*types.Header)

  // FinalizeAndAssemble runs any post-transaction state modifications (e.g. block
  // rewards) and assembles the final block.
  //
  // Note: The block header and state database might be updated to reflect any
  // consensus rules that happen at finalization (e.g. block rewards).
  FinalizeAndAssemble(chain ChainHeaderReader, header *types.Header, state *state.StateDB, txs []*types.Transaction,
    uncles []*types.Header, receipts []*types.Receipt) (*types.Block, error)

  // Seal generates a new sealing request for the given input block and pushes
  // the result into the given channel.
  //
  // Note, the method returns immediately and will send the result async. More
  // than one result may also be returned depending on the consensus algorithm.
  Seal(chain ChainHeaderReader, block *types.Block, results chan<- *types.Block, stop <-chan struct{}) error

  // SealHash returns the hash of a block prior to it being sealed.
  SealHash(header *types.Header) common.Hash

  // CalcDifficulty is the difficulty adjustment algorithm. It returns the difficulty
  // that a new block should have.
  CalcDifficulty(chain ChainHeaderReader, time uint64, parent *types.Header) *big.Int

  // APIs returns the RPC APIs this consensus engine provides.
  APIs(chain ChainHeaderReader) []rpc.API

  // Close terminates any background threads maintained by the consensus engine.
  Close() error
}
```

各方法的实现及作用如下：

方法	作用
`Author()`	获取挖出指定区块的以太坊账户地址
`VerifyHeader()`	检查区块头是否符合给定引擎的共识规则
`VerifyHeaders()`	与方法 `VerifyHeader()` 类似，但能够同时验证一批区块头
`VerifyUncles()`	验证给定区块的叔区块是否符合给定引擎的共识规则

VerifySeal()	根据给定引擎的共识规则检查区块头中的签名是否有效
Prepare()	根据特定引擎的共识规则初始化区块头的共识字段
Finalize()	运行任何后事务状态修改（例如区块奖励）并组装最终的区块
Seal()	为给定的输入区块生成新的签名请求，并将结果推送到给定的通道
SealHash()	返回区块在被签名之前的哈希值
CalcDifficulty()	难度调整算法，返回新区块应该具有的难度
API()	返回此共识引擎提供的 RPC API
Close()	终止共识引擎维护的所有后台线程

以太坊有两个共识算法：clique 和 ethash，分别位于 consensus/clique 和 consensus/ethash，前者实现 PoA(权威证明)共识，后者实现 PoW(工作量证明)共识。

1.clique 算法

clique 是一种 PoA 算法，PoA 共识中出块权掌握在部分“专家”手里，而普通人是无法参与的，这样牺牲了一部去中心化的特性，换来了一种可控性。PoA 算法两大核心问题是：如何实现签名者的引进和踢出、如何控制出块时机。

以下列出源码中一些重要的数据定义，便于后续解释 clique 算法：

checkpoint	一个特殊的block,高度是EPOCH_LENGTH的整数倍,block中不包含投票信息但包含当时所有的签名者列表
SIGNER_COUNT	某一时刻签名者的数量
SIGNER_LIMIT	连续的块的数量，在这些连续的块中，某一签名者最多只能签一个块；同时也是投票生效的票数的最小值
BLOCK_PERIOD	两个相邻的块的 Time 字段的最小差值，也是出块周期
EPOCH_LENGTH	两个 checkpoint 之间的 block 的数量。达到这个数量后会生成 checkpoint 以及清除当前所有未生效的投票信息
DIFF_INTURN	出块状态之一，此状态代表“按道理已经轮到我出块”
DIFF_NOTURN	出块状态之一，此状态代表“按道理还没轮到我出块”

clique 中签名者的引进和踢出是通过现有签名者投票实现的,投票规则如下：

- 投票信息保存在 block 中。一个 block 只有一个投票信息，且只能在自己生成的 block 上保存。针对某个被投人的票数超过 SIGNER_LIMIT 时，投票结果立即生效。
- 投票生效后，立即清除所有被投人是当前生效人的投票信息。如果投的是踢出票，则被投人之前投出的、但还未生效的投票全部失效
- 踢出一个签名者以后，可能会导致原来不通过的投票理论上可以通过。clique 不特意处理这种情况，等待下次统计时再判断
- 发起一个投票后，客户端不会被立即清除投票信息，而是在之后每次出块时都会选一个继续投票。因为区块链中的有效区块有重新调整的可能性，所以不能认为投票生效了之后就会一直生效。

- 无效的投票：被投票人不在当前签名者列表中但投的是踢出票，或被投票人在当前签名列表中但投的是引进票
- 为了使编码简单，无效的投票不会受到惩罚
- 在每个 EPOCH_LENGTH 内，一个签名者给同一个账号地址重复投票时，会先将上次的投票信息清除，然后再统计本次的投票信息
- 每个 checkpoint 不进行投票，而只是包含当前签名者列表信息。对于其它区块，可以用来携带投票信息

clique 算法的出块时间是固定的，由 BLOCK_PERIOD 决定，出块权的确定遵循以下原则：

- 签名者在签名者列表中且在 SIGNER_LIMIT 内没出过块
- 如果签名者是 DIFF_INTURN 状态,则拥有较高出块权（等待出块时间到来，签名区块并立即广播出去）
- 如果签名者是 DIFF_NOTURN 状态,则拥有较低出块权（等待出块时间到来，再延迟一下（延迟时间为 rand(SIGNER_COUNT * 500ms)

clique 算法目前在官方应用上只用于测试网络，但我们自己创建私人网络时也可以使用。

2.ethash 算法

ethash 是一种 PoW 算法，是可能有出块权的共识机制：在每次出块时，只要计算工作做得够快，抢在别人之前计算出满足条件，就有权出块。

在以太坊区块的 Header 结构体里，有一个 Difficulty 字段，它定义了当前块的“出块难度”，在验证时，要求计算得到哈希值作为一个整数必须小于 Difficulty 值。

除此之外，Header 中还有一个 MixDigest 字段，ethash 在计算哈希时会得到两个值，除了用来与 Difficulty 字段进行比较的哈希值，还有一个就是用来与 MixDigest 进行比较的哈希值。要想验证通过,这个啥希值必须与 MixDigest 字段中的一致。