Ethereum Header/Block 结构体解读

Header 结构体

```
// 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        *big.Int       `json:"timestamp"        gencodec:"required"`
    Extra       []byte         `json:"extraData"        gencodec:"required"`
    MixDigest   common.Hash    `json:"mixHash"          gencodec:"required"`
    Nonce       BlockNonce     `json:"nonce"            gencodec:"required"`
}
```

- Header 的字段涉及共识机制，建议和 consensus/ethash/consensus.go 中 verifyHeader 对比起来学习

ParentHash

- 父区块的 Hash 值

UncleHash

- 叔区块的 Hash 值

```
b.header.UncleHash = CalcUncleHash(uncles)

func CalcUncleHash(uncles []*Header) common.Hash {
    return rlpHash(uncles)
}
```

Header/Block Hash 的计算方法

- 相同高度的 Header 和 Block 的 Hash 值是一样的
  - 因此本质上，HeaderChain 和 BlockChain 是等价的
- Hash 算法的输入是整个 Header 的 RLP Encode 结果

```
// Hash returns the keccak256 hash of b's header.
// The hash is computed on the first call and cached thereafter.
func (b *Block) Hash() common.Hash {
    if hash := b.hash.Load(); hash != nil {
        return hash.(common.Hash)
    }
    v := b.header.Hash()
    b.hash.Store(v)
    return v
}

// Hash returns the block hash of the header, which is simply the keccak256
hash of its
// RLP encoding.
func (h *Header) Hash() common.Hash {
    return rlpHash(h)
}

func rlpHash(x interface{}) (h common.Hash) {
    hw := sha3.NewKeccak256()
    rlp.Encode(hw, x)
    hw.Sum(h[:0])
    return h
}
```

Coinbase

- 矿工地址

Root

- StateDB 的 Trie 的 Root Hash

TxHash

- 本 block 内所有 Transaction 构成的 Trie 的 Root Hash

```
b.header.TxHash = DeriveSha(Transactions(txs))
```

- 从 Transaction list 构建 Trie
  i. 将 []*Transaction 转为 Transactions（实现了 DerivableList interface）
  ii. 使用 DeriveSha 将 Transactions 构建成 Trie，并计算 Root Hash
    - key 为在 list 里的索引值
    - value 为 Transaction 的 RLP Encode 结果

```
type DerivableList interface {
    Len() int
    GetRlp(i int) []byte
}

// Transactions is a Transaction slice type for basic sorting.
type Transactions []*Transaction

// Len returns the length of s.
func (s Transactions) Len() int { return len(s) }

// Swap swaps the i'th and the j'th element in s.
func (s Transactions) Swap(i, j int) { s[i], s[j] = s[j], s[i] }

// GetRlp implements Rlpable and returns the i'th element of s in rlp.
func (s Transactions) GetRlp(i int) []byte {
    enc, _ := rlp.EncodeToBytes(s[i])
    return enc
}
```

```
func DeriveSha(list DerivableList) common.Hash {
    keybuf := new(bytes.Buffer)
    trie := new(trie.Trie)
    for i := 0; i < list.Len(); i++ {
        keybuf.Reset()
        rlp.Encode(keybuf, uint(i))
        trie.Update(keybuf.Bytes(), list.GetRlp(i))
    }
    return trie.Hash()
}
```

ReceiptHash

- 本 block 内所有 Receipt 构成的 Trie 的 Root Hash（计算方法同 TxHash）

```
b.header.ReceiptHash = DeriveSha(Receipts(receipts))
```

Bloom

- 由 receipts 构建，将所有的 Log 信息压缩到 Bloom 里

```
b.header.Bloom = CreateBloom(receipts)
```

```
func CreateBloom(receipts Receipts) Bloom {
    bin := new(big.Int)
    for _, receipt := range receipts {
        bin.Or(bin, LogsBloom(receipt.Logs))
    }

    return BytesToBloom(bin.Bytes())
}

func LogsBloom(logs []*Log) *big.Int {
    bin := new(big.Int)
    for _, log := range logs {
        bin.Or(bin, bloom9(log.Address.Bytes()))
        for _, b := range log.Topics {
            bin.Or(bin, bloom9(b[:]))
        }
    }

    return bin
}

func bloom9(b []byte) *big.Int {
    b = crypto.Keccak256(b[:])

    r := new(big.Int)

    for i := 0; i < 6; i += 2 {
        t := big.NewInt(1)
        b := (uint(b[i+1]) + (uint(b[i]) << 8)) & 2047
        r.Or(r, t.Lsh(t, b))
    }

    return r
}
```

Difficulty

- 本 Block 的**目标难度值**，PoW 重要的一个参数
- 和 parent block 的 Difficulty 以及本 block 的 Time 有关
- 和 Number 有关
- 各个阶段的规则不完全一致
- 总体规则
    - 和 parent block 的 Difficulty 的差值不能过大
    - 和 parent block 的时间差越小，目标难度值越高
    - 随着时间差增大，目标难度值降低
    - 未来会随着 Number 的增大，呈指数增长（趋向于切换到 PoS 机制）

```
// Verify the block's difficulty based in it's timestamp and parent's diffic
ulty
    expected := ethash.CalcDifficulty(chain, header.Time.Uint64(), parent)

    if expected.Cmp(header.Difficulty) != 0 {
        return fmt.Errorf("invalid difficulty: have %v, want %v", header.Dif
ficulty, expected)
    }

// CalcDifficulty is the difficulty adjustment algorithm. It returns
// the difficulty that a new block should have when created at time
// given the parent block's time and difficulty.
func CalcDifficulty(config *params.ChainConfig, time uint64, parent *types.H
eader) *big.Int {
    next := new(big.Int).Add(parent.Number, big1)
    switch {
    case config.IsByzantium(next):
        return calcDifficultyByzantium(time, parent)
    case config.IsHomestead(next):
        return calcDifficultyHomestead(time, parent)
    default:
        return calcDifficultyFrontier(time, parent)
    }
}
```

- Byzantium 规则
    - 目前主网上实际使用的规则（从 4370000 区块开始使用）

- H_d 表示 Header 的 Diffculty
- H_s 表示 Header 的 Time
- H_o 表示 Header 的 UncleHash
- H_i 表示 Header 的 Number
- $P(H)$ 表示 Parent Header

$$\zeta = \begin{cases} 2 - \frac{H_s - P(H)_{H_s}}{9}, & P(H)_{H_o} \neq nil \\ 1 - \frac{H_s - P(H)_{H_s}}{9}, & P(H)_{H_o} = nil \end{cases}$$

$$\varepsilon = 2^{\frac{\max(0, H_i - 3000000)}{100000} - 2}$$

$$H_d = P(H)_{H_d} + \frac{P(H)_{H_d}}{2048} * \max(\zeta, -99) + \varepsilon$$

1. ζ 用于控制区块的打包时间在一个相对稳定值（目前在 15s 左右）
   i. H_s 越大，ζ 越小，对应的 Difficulty 越低
   ii. 对包含了 uncle 的 block，Difficulty 有一定提升，原因在这里
   iii. **TODO: 如何推导出 15s 是最后稳定收敛的值?**
2. ε 用于在未来能指数级别的提升 Difficulty，使得网络从 PoW 切换到 PoS
   i. 为了延迟 Difficulty 指数增长时期的到来，Byzantium 阶段使用了 $\text{fakeBlockNumber} = H_i - 3000000$

- Homestead 规则
  - 在 1150000 到 4369999 区块使用
  - 已经废弃

$$\zeta = 1 - \frac{H_s - P(H)_{H_s}}{10}$$

$$\varepsilon = 2^{\frac{H_i}{100000} - 2}$$

$$H_d = P(H)_{H_d} + \frac{P(H)_{H_d}}{2048} * \max(\zeta, -99) + \varepsilon$$

- Frontier 规则

Number

- Block 编号（高度），必须是 parent block 增 1

```
    // Verify that the block number is parent's +1
    if diff := new(big.Int).Sub(header.Number, parent.Number); diff.Cmp(big.
NewInt(1)) != 0 {
        return consensus.ErrInvalidNumber
    }
```

GasLimit

- Block 的 Gas 最大值
- miner 有资格调整新区块的 GasLimit，但是调整的范围被约束（**属于共识机制的一部分**）
  - 和前一个区块的差值，不能超过前一个区块 GasLimit 的 1/GasLimitBoundDivisor (主网中 GasLimitBoundDivisor = 1024)
  - 不能小于 MinGasLimit（主网中 MinGasLimit = 5000）
  - Verify 代码如下：

```
    // Verify that the gas limit remains within allowed bounds
    diff := new(big.Int).Set(parent.GasLimit)
    diff = diff.Sub(diff, header.GasLimit)
    diff.Abs(diff)

    limit := new(big.Int).Set(parent.GasLimit)
    limit = limit.Div(limit, params.GasLimitBoundDivisor)

    if diff.Cmp(limit) >= 0 || header.GasLimit.Cmp(params.MinGasLimit) < 0 {
        return fmt.Errorf("invalid gas limit: have %v, want %v += %v", heade
r.GasLimit, parent.GasLimit, limit)
    }
```

- miner 一种调整 GasLimit 的方式是（注意：本方法**并不属于共识机制的一部分**！）
  - 判断父区块的 GasUsed 是否超过了父区块 GasLimit 的 2/3
    - 如果超过了，则增大 GasLimit
    - 否则，减小 GasLimit
  - 如果 GasLimit < params.MinGasLimit，则直接使用 params.MinGasLimit
  - 如果 GasLimit < params.TargetGasLimit，则尽最大可能去逼近 params.TargetGasLimit
    - 注意：**不是直接赋值为 params.TargetGasLimit，那样可能会和共识机制冲突!**

```
// CalcGasLimit computes the gas limit of the next block after parent.
// This is miner strategy, not consensus protocol.
func CalcGasLimit(parent *types.Block) uint64 {
```

```
    // contrib = (parentGasUsed * 3 / 2) / 1024
    contrib := (parent.GasUsed() + parent.GasUsed()/2) / params.GasLimitBoun
dDivisor

    // decay = parentGasLimit / 1024 -1
    decay := parent.GasLimit()/params.GasLimitBoundDivisor - 1

    /*
        strategy: gasLimit of block-to-mine is set based on parent's
        gasUsed value.  if parentGasUsed > parentGasLimit * (2/3) then we
        increase it, otherwise lower it (or leave it unchanged if it's right
        at that usage) the amount increased/decreased depends on how far awa
y
        from parentGasLimit * (2/3) parentGasUsed is.
    */
    limit := parent.GasLimit() - decay + contrib
    if limit < params.MinGasLimit {
        limit = params.MinGasLimit
    }
    // however, if we're now below the target (TargetGasLimit) we increase t
he
    // limit as much as we can (parentGasLimit / 1024 -1)
    if limit < params.TargetGasLimit {
        limit = parent.GasLimit() + decay
        if limit > params.TargetGasLimit {
            limit = params.TargetGasLimit
        }
    }
    return limit
}
```

GasUsed

- Block 实际消耗的 Gas 值

Time

- 区块生成时间
- 共识
  - Block 的 Time 应该大于所有的 uncle block 和 parent block 的 Time
  - 不能比本地的时间超前超过 allowedFutureBlockTime （主网为 15 s）

```
    // Verify the header's timestamp
    if uncle {
```

```
        if header.Time.Cmp(math.MaxBig256) > 0 {
            return errLargeBlockTime
        }
    } else {
        if header.Time.Cmp(big.NewInt(time.Now().Add(allowedFutureBlockTime)
.Unix())) > 0 {
            return consensus.ErrFutureBlock
        }
    }
    if header.Time.Cmp(parent.Time) <= 0 {
        return errZeroBlockTime
    }
```

Extra

- Block 的额外数据，不得大于 32 字节！

MixDigest

- 256 bit
- PoW 计算的**输出**之一

Nonce

- 64 bit
- PoW 计算的**输入**之一

PoW 相关

$$(m, n) = \mathrm{PoW}(H^o, H_n, \mathbf{d})$$

- 输入
  - H^o 表示除了 Nonce 和 MixDigest 外的 Header
    - 改变 Extra 可以改变 PoW 的输出结果
  - H_n 是 Nonce
    - 改变 Nonce 可以改变 PoW 的输出结果
  - $\mathbf{d}$ 表示当前的 DAG
- 输出
  - m 保存在 MixDigest 中，即 H_m

- n 需要满足 $n \leq \frac{2^{256}}{H_d}$ （即为 PoW 过程）

- Source
  - hashimotoLight 即是 PoW 的实现

```
digest, result := hashimotoLight(size, cache.cache, header.HashNoNonce().Byt
es(), header.Nonce.Uint64())

cache := ethash.cache(number)
size := datasetSize(number)

// HashNoNonce returns the hash which is used as input for the proof-of-work
 search.
func (h *Header) HashNoNonce() common.Hash {
    return rlpHash([]interface{}{
        h.ParentHash,
        h.UncleHash,
        h.Coinbase,
        h.Root,
        h.TxHash,
        h.ReceiptHash,
        h.Bloom,
        h.Difficulty,
        h.Number,
        h.GasLimit,
        h.GasUsed,
        h.Time,
        h.Extra,
    })
}

 // hashimotoLight aggregates data from the full dataset (using only a small
// in-memory cache) in order to produce our final value for a particular hea
der
// hash and nonce.
func hashimotoLight(size uint64, cache []uint32, hash []byte, nonce uint64)
([]byte, []byte) {
    keccak512 := makeHasher(sha3.NewKeccak512())

    lookup := func(index uint32) []uint32 {
        rawData := generateDatasetItem(cache, index, keccak512)

        data := make([]uint32, len(rawData)/4)
        for i := 0; i < len(data); i++ {
            data[i] = binary.LittleEndian.Uint32(rawData[i*4:])
        }
```

```
        return data
    }
    return hashimoto(hash, nonce, size, lookup)
}
```

- Verify

```
    if !bytes.Equal(header.MixDigest[:], digest) {
        return errInvalidMixDigest
    }

    target := new(big.Int).Div(maxUint256, header.Difficulty)
    if new(big.Int).SetBytes(result).Cmp(target) > 0 {
        return errInvalidPoW
    }
```

Block 结构体

```
// 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{}
}
```

- 本质上就是 Header 加上 Transaction 的具体信息