以太坊黄皮书详解（一）

2018-05-23 Ethereum

写在篇头

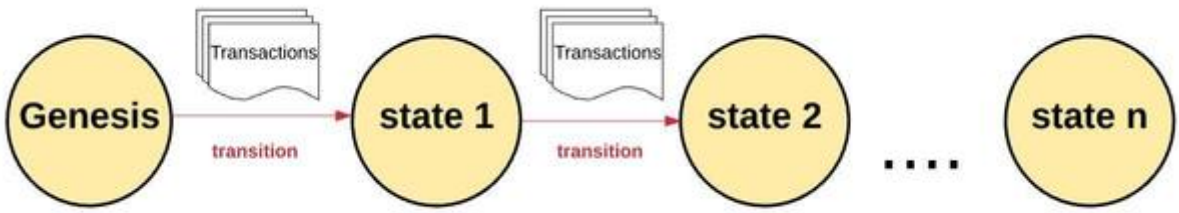
本文是对以太坊的黄皮书的解析，并参照go-ethereum中的实现，将相应的代码也列了出来。黄皮书中使用了大量的公式将以太坊的一些流程和状态都公式化了。看不看得懂公式对理解的影响不大。本文中对公式进行了解析。嫌麻烦的可以跳过每部分公式解析的部分。

一、区块链范型

以太坊本质是一个基于交易的状态机（transaction-based state machine）。其以初始状态（genesis state）为起点，通过执行交易来到达新的状态。

$$\sigma_{t+1} \equiv \Upsilon(\boldsymbol{\sigma}_t, T) \tag{1}$$

公式1表示t+1时的状态，是由t时的状态经过交易T转变而来。转变函数为Υ。 如下图所示



$$\boldsymbol{\sigma}_{t+1} \equiv \Pi(\boldsymbol{\sigma}_t, B) \tag{2}$$

$$B \equiv (\ldots, (T_0, T_1, \ldots)) \tag{3}$$

$$\Pi(\boldsymbol{\sigma}, B) \equiv \Omega(B, \Upsilon(\Upsilon(\boldsymbol{\sigma}, T_0), T_1)\ldots) \tag{4}$$

公式2–4是从区块的角度来描述状态的转化过程。 公式2: t+1时的状态，是由t时的状态经过区块B转变而来。转变函数为Π。

公式3: 区块B是包含了一系列交易T的集合。

公式4: 区块的状态转变函数Π，相当于逐条的执行交易的状态转变Υ，然后完成所有交易转变后再经过Ω进行一次状态转换。（这个地方Ω实际上是给矿工挖坑奖励。）

在以太坊中的实际情况就是区块验证和执行的过程。

- 逐一的执行交易（也就是使用交易转变函数操作Υ状态集）。实际就是交易比方是A向B转10ether，则A账户值–10，B账户值+10。（当然执行过程中还有gas消耗，这个后面详述）
- 等整个block交易执行完毕后，需要对矿工进行奖励。也就是需要使用Ω进行一次状态转换。

1.1 货币

以太坊中有以下四种单位的货币。以太坊中的各种计算都是以Wei为单位的。 *（看有的地方好像有更多种单位，我这边是直接按照黄皮书走的）*

Multiplier	Name
10^0	Wei
10^{12}	Szabo
10^{15}	Finney
10^{18}	Ether

1.2 分叉

以太坊的正确运行建立在其链上只有一个链是有效的，所有人都必须要接受它。拥有多个状态（或多个链）会摧毁这个系统，因为它在哪个是正确状态的问题上不可能得到统一结果。如果链分叉了，你有可能在一条链上拥有10个币，一条链上拥有20个币，另一条链上拥有40个币。在这种场景下，是没有办法确定哪个链才是最”有效的“。不论什么时候只要多个路径产生了，一个”分叉“就会出现。 为了确定哪个路径才是最有效的以及防止多条链的产生，以太坊使用了一个叫做“GHOST协议(GHOST protocol.)”的数学机制。

简单来说，GHOST协议就是让我们必须选择一个在其上完成计算最多的路径。一个方法确定路径就是使用最近一个区块（叶子区块）的区块号，区块号代表着当前路径上总的区块数（不包含创世纪区块）。区块号越大，路径就会越长，就说明越多的挖矿算力被消耗在此路径上以达到叶子区块。

二、 区块、状态与交易

2.1 世界状态

以太坊中的世界状态指地址(Address)与账户状态(Account State)的集合。世界状态并不是存储在链上，而是通过Merkle Patricia tree来维护。 账户状态（Account State）包含四个属性。

- nonce: 如果账户是一个外部拥有账户，nonce代表从此账户地址发送的交易序号。如果账户是一个合约账户，nonce代表此账户创建的合约序号。用$\boldsymbol{\sigma}[a]_{\mathbf{n}}$来表示。
- balance：此地址拥有Wei的数量。1Ether=10^18Wei。用$\boldsymbol{\sigma}[a]_{\mathbf{b}}$来表示。
- storageRoot： 理论上是指Merkle Patricia树的根节点256位的Hash值。用$\boldsymbol{\sigma}[a]_{\mathbf{s}}$来表示。公式6中有介绍。
- codeHash：此账户EVM代码的hash值。对于外部拥有账户，codeHash域是一个空字符串 的Hash值。对于合约账户，就是代码的Hash作为codeHash保存。用$\boldsymbol{\sigma}[a]_{\mathbf{c}}$来表示。

```
// github.com/ethereum/go-ethereum/core/state/state_object.go
type Account struct {
        Nonce    uint64
        Balance  *big.Int
        Root     common.Hash // merkle root of the storage trie
        CodeHash []byte
}
```

关于storageRoot的补充

$$TRIE\left(L_I^*(\boldsymbol{\sigma}[a]_{\mathrm{s}})\right) \equiv \boldsymbol{\sigma}[a]_{\mathrm{s}} \tag{6}$$

$$L_I\left((k,v)\right) \equiv \left(KEC(k), RLP(v)\right) \tag{7}$$

$$k \in \mathbb{B}_{32} \quad \wedge \quad v \in \mathbb{N} \tag{8}$$

公式6, 由于有些时候我们不仅需要state的hash值的trie，而是需要其对应的kv数据也包含其中。所以以太坊中的存储State的树，不仅包含State的hash，同时也包含了存储这

个账户的address的hash和它对应的data也就是其Account的值的数据对的集合。这里storageRoot实际上是这样的树的根节点hash值。

公式7，指state对应的kv数据的RLP的形式化表示，是k的hash值作为key，value是v的RLP表示。也就是以太坊中实际存储的state是账户address的hash ($KEC(k)$) ,与其数据Account内容的RLP ($RLP(v)$)。

公式8，指公式7中的k是32的字符数组。这个是由KECCAK256算法保证的。

注：原本公式8中要求的是v是个正整数，但是我看来下代码和下文公式10，感觉这里的v都应该是Account的内容

一些符号化定义

以太坊中的账户有两类，一类是外部账户，一类是合约账户。其中外部账户被私钥控制且没有任何代码与之关联。合约账户，被它们的合约代码控制且有代码与之关联。以下几个公式定义了账户的各种状态。

$$L_S(\boldsymbol{\sigma}) \equiv \{p(a) : \boldsymbol{\sigma}[a] \neq \varnothing\} \tag{9}$$

其中

$$p(a) \equiv \Big(KEC(a), RLP\Big((\boldsymbol{\sigma}[a]_{\mathrm{n}}, \boldsymbol{\sigma}[a]_{\mathrm{b}}, \boldsymbol{\sigma}[a]_{\mathrm{s}}, \boldsymbol{\sigma}[a]_{\mathrm{c}})\Big)\Big) \tag{10}$$

$$\forall a : \boldsymbol{\sigma}[a] = \varnothing \vee (a \in \mathbb{B}_{20} \wedge v(\boldsymbol{\sigma}[a])) \tag{11}$$

$$v(x) \equiv x_{\mathrm{n}} \in \mathbb{N}_{256} \wedge x_{\mathrm{b}} \in \mathbb{N}_{256} \wedge x_{\mathrm{s}} \in \mathbb{B}_{32} \wedge x_{\mathrm{c}} \in \mathbb{B}_{32} \tag{12}$$

$$\textsc{empty}(\boldsymbol{\sigma}, a) \quad \equiv \quad \boldsymbol{\sigma}[a]_{\mathrm{c}} = KEC\big(()\big) \wedge \boldsymbol{\sigma}[a]_{\mathrm{n}} = 0 \wedge \boldsymbol{\sigma}[a]_{\mathrm{b}} = 0 \tag{13}$$

$$\textsc{dead}(\boldsymbol{\sigma}, a) \quad \equiv \quad \boldsymbol{\sigma}[a] = \varnothing \vee \textsc{empty}(\boldsymbol{\sigma}, a) \tag{14}$$

公式9，定义了函数L_S，意思是若账户a不为空，则返回账户$p(a)$。

公式10，定义$p(a)$,$p(a)$其实就是我们上面公式7，解释k，v对的时候的kv对。包括address的hash值，以及Account内容的RLP结果。

公式11与公式12，对账户a做了定义，表示账户要么为空，要么就是一个a为20个长度的字符，其nonce值为小于2^{256}的正整数，balance值为小于2^{256}的正整数，storageRoot为32位的字符，codeHash为32的字符。

公式13，定义了空账户。若一个账户，其地址为空字符，并且该账户nonce值为0，balance值也为0.

公式14，定义了死账户，死账户要么为空∅，要么是一个EMPTY账户。

2.2 交易

- nonce: 与发送该交易的账户的nonce值一致。用T_{n}表示。
- gasPrice: 表示每gas的单价为多少wei。用T_{p}表示。
- gasLimit:执行该条交易最大被允许使用的gas数目。用T_{g}表示。
- to:160位的接受者地址。当交易位创建合约时，该值位空。用T_{t}表示。
- value:表示发送者发送的wei的数目。该值为向接受者转移的wei的数目，或者是创建合约时作为合约账户的初始wei数目。用T_{v}表示。
- v,r,s: 交易的签名信息，用以决定交易的发送者。分别用T_{w},T_{r},T_{s}表示。
- init:如果是创建合约的交易,则init表示一段不限长度的EVM-Code用以合约账户初始化的过程。用T_{i}表示。
- data: 调用合约的交易，会包含一段不限长度的输入信息，用T_{d}表示。

```go
// github.com/ethereum/go-ethereum/core/types/transaction.go
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 cont
        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:"-"`
}
```

一些符号化定义

$$L_T(T) \equiv \begin{cases} (T_{\mathrm{n}}, T_{\mathrm{p}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, T_{\mathbf{i}}, T_{\mathrm{w}}, T_{\mathrm{r}}, T_{\mathrm{s}}) & \text{if } T_{\mathrm{t}} = \varnothing \\ (T_{\mathrm{n}}, T_{\mathrm{p}}, T_{\mathrm{g}}, T_{\mathrm{t}}, T_{\mathrm{v}}, T_{\mathbf{d}}, T_{\mathrm{w}}, T_{\mathrm{r}}, T_{\mathrm{s}}) & \text{otherwise} \end{cases} \tag{15}$$

$$\begin{array}{lllllll} T_{\mathrm{n}} \in \mathbb{N}_{256} & \wedge & T_{\mathrm{v}} \in \mathbb{N}_{256} & \wedge & T_{\mathrm{p}} \in \mathbb{N}_{256} & \wedge \\ T_{\mathrm{g}} \in \mathbb{N}_{256} & \wedge & T_{\mathrm{w}} \in \mathbb{N}_{5} & \wedge & T_{\mathrm{r}} \in \mathbb{N}_{256} & \wedge \\ T_{\mathrm{s}} \in \mathbb{N}_{256} & \wedge & T_{\mathbf{d}} \in \mathbb{B} & \wedge & T_{\mathbf{i}} \in \mathbb{B} & \end{array} \tag{16}$$

$$\mathbb{N}_{\mathrm{n}} = \{P : P \in \mathbb{N} \wedge P < 2^{n}\} \tag{17}$$

$$T_{\mathbf{t}} \in \begin{cases} \mathbb{B}_{20} & \text{if} \quad T_{\mathrm{t}} \neq \varnothing \\ \mathbb{B}_{0} & \text{otherwise} \end{cases} \tag{18}$$

以太坊中根据交易中的to值是否为空，可以判断交易是创建合约还是执行合约。

公式15表示，如果to的值T_{t}为空，则交易是创建合约的交易，需要有init数据。交易的RLP形式化可以表示为nonce T_{n}，gasPrice T_{p}, gasLimit T_{g}, to T_{t}，value T_{v}，**init** T_{i}，“v ，r， s” $T_{\mathrm{w}}, T_{\mathrm{r}}, T_{\mathrm{s}}$。如果$T_{\mathrm{t}}$不为空，则交易是执行合约的交易，需要有data数据。交易的RLP形式化可以表示为nonce T_{n}，gasPrice T_{p}, gasLimit T_{g}, to T_{t}，value T_{v}，**data** T_{d}，“v ，r， s” $T_{\mathrm{w}}, T_{\mathrm{r}}, T_{\mathrm{s}}$。

公式16，是对交易的各个字段限制的符号化定义。其意思是nonce、value、gasPrice、gasLimit以及特殊的用来验证签名的r和s 都是小于2^{256}的正整数，用来验证签名的v (T_{w})是小于2^{5}的正整数。而init和data都是未知长度的字符数组。

公式17，是对$\mathbb{N}_{\mathrm{n}}$的定义，即小于2^{n}的正整数。

公式18，是对交易中的to字段的符号化定义，当其不为空的时候，是20位的字符，为空的时候是0位字符。

2.3 区块

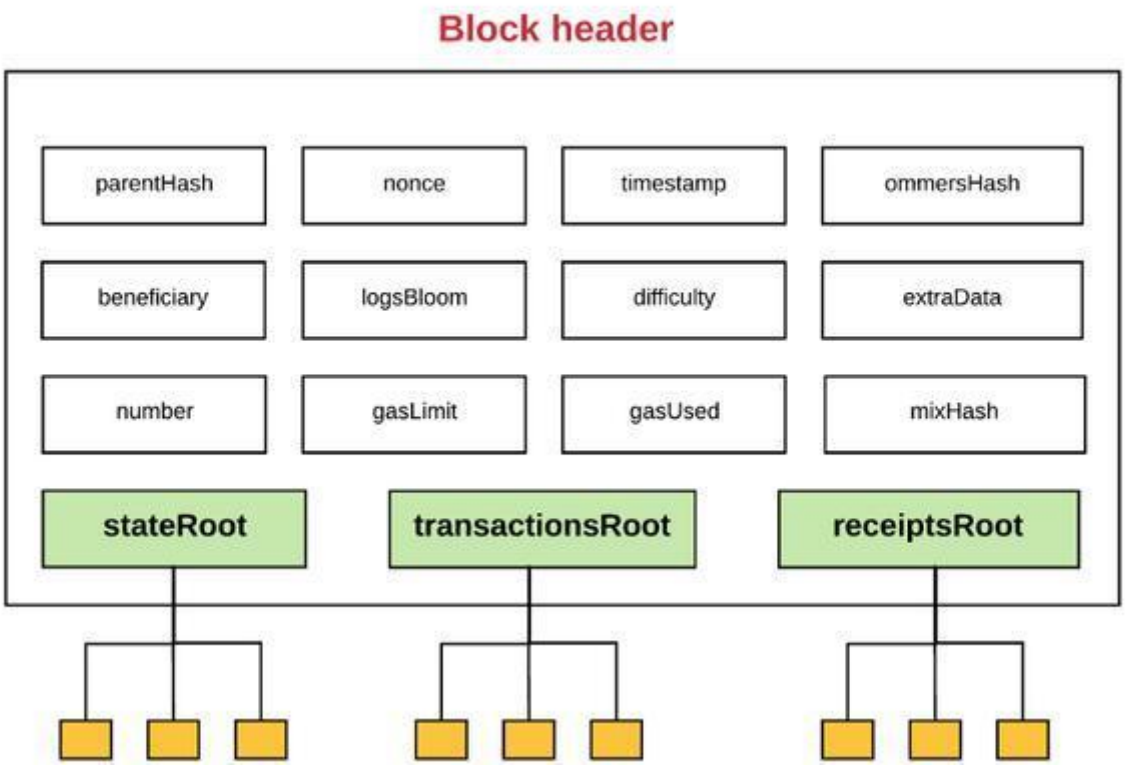
以太坊中的一个区块由区块头Header，以及交易列表$B_{\mathbf{T}}$,以及ommerblock的header集合$B_{\mathbf{U}}$三部分组成。

Header包括以下字段。

- parentHash: 父节点的hash值。用H_{p}表示。
- ommersHash: uncle节点的hash值，这块是跟GHOST相关的，用H_{o}表示。
- beneficiary: 矿工address，用H_{c}表示。
- stateRoot: 当所有交易都执行完毕后的世界状态树的根节点，用H_{r}表示。
- transactionsRoot:交易列表的根节点，用H_{t}表示。
- receiptsRoot:收据的根节点，用H_{e}表示。

  logsBloom:日志过滤器，用H_{b}表示。*这个暂时没细看，不太确定。*

- difficulty:区块难度，根据上一个区块的难度以及时间戳算出来的值，用H_{d}表示。
- number:区块号，用H_{i}表示。
- gasLimit: 区块的gas数量限制，即区块中交易使用掉的gas值不应该超过该值。用H_{l}表示。
- gasUsed: 区块使用掉的gas数量，用H_{g}表示。
- timestamp:时间戳，用H_{s}表示。
- extraData:额外的数据，合法的交易对长度有限制，用H_{x}表示。
- mixHash: 与nonce一起用作工作量证明，用H_{m}表示。
- nonce:与mixHash一起用作工作量证明，用H_{n}表示。




```go
// github.com/ethereum/go-ethereum/core/types/block.go
// "external" block encoding. used for eth protocol, etc.
type extblock struct {
        Header *Header
        Txs    []*Transaction
        Uncles []*Header
}

// 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"`
}
// Receipt represents the results of a transaction.
type Receipt struct {
        // Consensus fields
        PostState         []byte `json:"root"`
        Status            uint   `json:"status"`
        CumulativeGasUsed uint64 `json:"cumulativeGasUsed" gencodec:"required"`
        Bloom             Bloom  `json:"logsBloom"         gencodec:"required"`
        Logs              []*Log `json:"logs"              gencodec:"required"`

        // Implementation fields (don't reorder!)
        TxHash          common.Hash    `json:"transactionHash" gencodec:"required"
        ContractAddress common.Address `json:"contractAddress"`
        GasUsed         uint64         `json:"gasUsed" gencodec:"required"`
}
```

区块的符号化定义

$$B \equiv (B_H, B_{\mathbf{T}}, B_{\mathbf{U}}) \tag{19}$$

公式19，表示区块由三部分组成，区块头B_H，交易列表$B_{\mathbf{T}}$,以及ommerblock的header集合$B_{\mathbf{U}}$。

2.3.1 交易收据

以太坊中为了将交易的信息进行编码，以方便索引以及查找或者零知识证明等相关的东西，为每条交易定义了一定收据。对于第i个交易，其收据用B_R[i]表示。 每条收据都是一个四元组，包括区块当前累计使用的gas值R_{u}，交易执行产生的log R_{l}，日志过滤器R_{b}，以及状态码R_{z}。

$$R \equiv (R_{\mathrm{u}}, R_{\mathrm{b}}, R_{\mathbf{l}}, R_{\mathrm{z}}) \tag{20}$$

$$L_R(R) \equiv (0 \in \mathbb{B}_{256}, R_{\mathrm{u}}, R_{\mathrm{b}}, R_{\mathbf{l}}) \tag{21}$$

$$R_{\mathrm{z}} \in \mathbb{N} \tag{22}$$

$$R_{\mathrm{u}} \in \mathbb{N} \quad \wedge \quad R_{\mathrm{b}} \in \mathbb{B}_{256} \tag{23}$$

$$O \equiv (O_{\mathrm{a}}, (O_{\mathbf{t}0}, O_{\mathbf{t}1}, \ldots), O_{\mathbf{d}}) \tag{24}$$

$$O_{\mathrm{a}} \in \mathbb{B}_{20} \wedge \forall_{t \in O_{\mathbf{t}}} : t \in \mathbb{B}_{32} \wedge O_{\mathbf{d}} \in \mathbb{B} \tag{25}$$

公式20对区块头中的收据作了定义，收据是个四元组，四元组定义如前文。

公式21，收据的RLP形式化表示为$L_R(R)$。其中$0 \in \mathbb{B}_{256}$在之前版本的协议中是交易执行之前的stateRoot。现在被替换为0.

公式22，表示R_{z}状态码是正整数。

公式23对收据中当前累计使用的gas值R_{u}，和日志过滤器R_{b}进行了描述。显然累计gas值R_{u}是一个正整数。而日志过滤器R_{b}是256位字符。

公式24对交易执行的日志R_{l}进行了解释。

公式25对其限制进行了描述。R_{l}是日志条目的序列。日志条目需要包括纪录日志者的地址，以及日志话题分类，以及实际数据。日志条目用O来表示，用O_{a}表示日志纪录者的address，用O_{t}来表示一些列32位字符的日志主题(log topics),用O_{d}来表示字符数据。其中日志纪录者的address O_{a}是20位字符，每一个日志分类话题O_{t}是一个32位字符，而日志数据O_{d}是未知长度的字符。

公式26-30.对日志过滤函数做了定义，这块涉及到东西是数据操作层面的，不影响对流程的理解。暂不作解释。

2.3.2 整体的合法性

上面介绍过了区块包含区块头，区块交易列表，以及区块的ommer区块的头三部分。以太坊中判断一个区块是否合法，首先需要对区块整体上做合法性判断。见公式31

- 其区块头的状态树的根stateRoot也就是 H_{r}，是否确实是状态树的根。
- 其区块头的ommerHash也就是 H_{o},是否与区块的ommer区块的头部分的hash值一致。
- 其区块头中的transactionRoot也就是H_{t}, f即区块中交易的树的根，是否和区块存储的交易列表中的交易一一对应。一一对应关系见公式32，是将交易所在列表中的索引的RLP作为键，交易内容v的RLP作为值的键值对。

- 其区块头中的recieptRoot也就是H_{e}，即区块中收据的树的根，是否和区块存储的交易列表一一对应，是不是每条交易都有一条相应的收据。一一对应关系见公式32，是将收据所在列表中的索引的RLP作为键，收据内容v的RLP作为值的键值对。
- 区块头中logsBloom也就是H_{b}，是否包含了区块的交易的所有日志。
- 执行该区块之前的状态树的根节点，是否与其父区块的中的根节点一致。见公式33.

$$\begin{array}{lcll} H_{\mathrm{r}} & \equiv & \mathtt{TRIE}(L_S(\Pi(\boldsymbol{\sigma}, B))) & \wedge \\ H_{\mathrm{o}} & \equiv & \mathtt{KEC}(\mathtt{RLP}(L_H^*(B_{\mathbf{U}}))) & \wedge \\ H_{\mathrm{t}} & \equiv & \mathtt{TRIE}(\{\forall i < \|B_{\mathbf{T}}\|, i \in \mathbb{P}: \quad p(i, L_T(B_{\mathbf{T}}[i]))\}) & \wedge \\ H_{\mathrm{e}} & \equiv & \mathtt{TRIE}(\{\forall i < \|B_{\mathbf{R}}\|, i \in \mathbb{P}: \quad p(i, L_R(B_{\mathbf{R}}[i]))\}) & \wedge \\ H_{\mathrm{b}} & \equiv & \bigvee_{\mathbf{r} \in B_{\mathbf{R}}} \left(\mathbf{r}_{\mathrm{b}}\right) & \end{array} \tag{31}$$

$$p(k, v) \equiv \left(\mathtt{RLP}(k), \mathtt{RLP}(v)\right) \tag{32}$$

$$\mathtt{TRIE}(L_S(\boldsymbol{\sigma})) = P(B_H)_{H_{\mathrm{r}}} \tag{33}$$

2.3.3 序列化

对区块，以及区块头的序列化表示如下。

$$L_H(H) \equiv (H_{\mathrm{p}}, H_{\mathrm{o}}, H_{\mathrm{c}}, H_{\mathrm{r}}, H_{\mathrm{t}}, H_{\mathrm{e}}, H_{\mathrm{b}}, H_{\mathrm{d}}, H_{\mathrm{i}}, H_{\mathrm{l}}, H_{\mathrm{g}}, H_{\mathrm{s}}, H_{\mathrm{x}}, H_{\mathrm{m}}, H_{\mathrm{n}}) \tag{34}$$

$$L_B(B) \equiv \left(L_H(B_H), L_T^*(B_{\mathbf{T}}), L_H^*(B_{\mathbf{U}})\right) \tag{35}$$

$$f^*\left((x_0, x_1, \ldots)\right) \equiv \left(f(x_0), f(x_1), \ldots\right) \tag{36}$$

$$\begin{array}{lllllll} H_{\mathrm{p}} \in \mathbb{B}_{32} & \wedge & H_{\mathrm{o}} \in \mathbb{B}_{32} & \wedge & H_{\mathrm{c}} \in \mathbb{B}_{20} & \wedge \\ H_{\mathrm{r}} \in \mathbb{B}_{32} & \wedge & H_{\mathrm{t}} \in \mathbb{B}_{32} & \wedge & H_{\mathrm{e}} \in \mathbb{B}_{32} & \wedge \\ H_{\mathrm{b}} \in \mathbb{B}_{256} & \wedge & H_{\mathrm{d}} \in \mathbb{N} & \wedge & H_{\mathrm{i}} \in \mathbb{N} & \wedge \\ H_{\mathrm{l}} \in \mathbb{N} & \wedge & H_{\mathrm{g}} \in \mathbb{N} & \wedge & H_{\mathrm{s}} \in \mathbb{N}_{256} & \wedge \\ H_{\mathrm{x}} \in \mathbb{B} & \wedge & H_{\mathrm{m}} \in \mathbb{B}_{32} & \wedge & H_{\mathrm{n}} \in \mathbb{B}_8 & \end{array} \tag{37}$$

公式34是区块头的序列化表示。

公式35是区块的序列化表示，即分别将区块头序列化，区块的交易列表，ommer区块头序列化。其中交易序列化函数L_T见公式15.

公式36表示列表序列化和当个序列化的关系，列表的序列化，就是把列表中的元素分别序列化，然后将结果组成列表。

公式37是对区块头中属性的限制规定：

- 各种hash值都是32位字符。包括parentHash H_{p},ommerHash H_{o}, stateRoot H_{r}, transactionRoot H_{t}, RecieptRoot H_{e}, mixHash H_{m}。
- 受益人也就是挖矿的人的地址beneficiary H_{c},是20位字符。
- 日志过滤器logBoom H_{b},是256位字符。
- 难度，区块号，gas限制，用掉的gas等都是正整数。difficulty H_{d}, Number H_{i},gasLimit H_{l} gasUsed H_{g}
- 时间戳timestamp H_{s}是个大的正整数，其值小于2^{256}。
- 额外的数据extraData H_{x}是未知长度字符。
- nonce H_{n}是8位的字符。

2.3.4 区块头的合法性

确定区块是否合法除了整体性的合法校验，还需要对区块头进行更进一步的校验。主要校验规则有以下几点：

- parentHash正确。即parentHash与其父区块的头的hash一致。
- number为父区块number值加一。
- difficuilty难度正确。区块合理的难度跟父区块难度，以及当前区块时间戳和父区块时间戳间隔以及区块编号有关。难度可以起到一定的调节出块时间的作用。，可以看出当出块变快（也就是出块间隔变小之后）难度会增加，相反难度会减小。
- gasLimit和上一个区块的差值在规定范围内。
- gasUsed小于等于gasLimit
- timestamp时间戳必须大于上一区块的时间戳。
- mixHash和nonce必须满足PoW。
- extraData最多为32个字节。

```
// verifyHeader checks whether a header conforms to the consensus rules of the
// stock Ethereum ethash engine.
// See YP section 4.3.4. "Block Header Validity"
func (ethash *Ethash) verifyHeader(chain consensus.ChainReader, header, parent *ty
        // Ensure that the header's extra-data section is of a reasonable size
      // 验证extraData的长度
        if uint64(len(header.Extra)) > params.MaximumExtraDataSize {
                return fmt.Errorf("extra-data too long: %d > %d", len(header.Extra
        }
```

```
        // Verify the header's timestamp
    // 验证时间戳是否超过大小限制，是否过大，是否大于上一区块的时间戳等
        if uncle {
                if header.Time.Cmp(math.MaxBig256) > 0 {
                        return errLargeBlockTime
                }
        } else {
                if header.Time.Cmp(big.NewInt(time.Now().Add(allowedFutureBlockTim
                        return consensus.ErrFutureBlock
                }
        }
        if header.Time.Cmp(parent.Time) <= 0 {
                return errZeroBlockTime
        }

      // 验证难度是否正确
        // Verify the block's difficulty based in it's timestamp and parent's diff
        expected := ethash.CalcDifficulty(chain, header.Time.Uint64(), parent)

        if expected.Cmp(header.Difficulty) != 0 {
                return fmt.Errorf("invalid difficulty: have %v, want %v", header.D
        }
        // Verify that the gas limit is <= 2^63-1
        cap := uint64(0x7fffffffffffffff)
     //验证gasLimit是否超了上限
        if header.GasLimit > cap {
                return fmt.Errorf("invalid gasLimit: have %v, max %v", header.GasL
        }
     //验证已用的gas值是否小于等于gasLimit
        // Verify that the gasUsed is <= gasLimit
        if header.GasUsed > header.GasLimit {
                return fmt.Errorf("invalid gasUsed: have %d, gasLimit %d", header.
        }

        // Verify that the gas limit remains within allowed bounds
    //判断gasLimit与父区块的gasLimit差值是否在规定范围内
        diff := int64(parent.GasLimit) - int64(header.GasLimit)
        if diff < 0 {
                diff *= -1
        }
        limit := parent.GasLimit / params.GasLimitBoundDivisor

        if uint64(diff) >= limit || header.GasLimit < params.MinGasLimit {
                return fmt.Errorf("invalid gas limit: have %d, want %d += %d", hea
        }
        // Verify that the block number is parent's +1
```

```
        //验证区块号，是否是父区块号+1
        if diff := new(big.Int).Sub(header.Number, parent.Number); diff.Cmp(big.Ne
                return consensus.ErrInvalidNumber
        }
        // Verify the engine specific seal securing the block
    //验证PoW
        if seal {
                if err := ethash.VerifySeal(chain, header); err != nil {
                        return err
                }
        }
        // If all checks passed, validate any special fields for hard forks
        if err := misc.VerifyDAOHeaderExtraData(chain.Config(), header); err != ni
                return err
        }
        if err := misc.VerifyForkHashes(chain.Config(), header, uncle); err != nil
                return err
        }
        return nil
}
```

校验区块号和区块hash的符号化表示

$$P(H) \equiv B' : \mathrm{KEC}(\mathrm{RLP}(B'_H)) = H_\mathrm{p} \tag{39}$$

$$H_\mathrm{i} \equiv P(H)_{H_\mathrm{i}} + 1 \tag{40}$$

公式39表示，parentHash值应该为父节点Header的hash值。

公式40表示，number为父节点number + 1.

难度计算的符号化表示

$$D(H) \equiv \begin{cases} D_0 & \text{if} \quad H_\mathrm{i} = 0 \\ \max\left(D_0, P(H)_{H_\mathrm{d}} + x \times \varsigma_2 + \epsilon\right) & \text{otherwise} \end{cases} \tag{41}$$

$$D_0 \equiv 131072 \tag{42}$$

$$x \equiv \left\lfloor \frac{P(H)_{H_\mathrm{d}}}{2048} \right\rfloor \tag{43}$$

$$\varsigma_2 \equiv \max\left(y - \left\lfloor \frac{H_\mathrm{s} - P(H)_{H_\mathrm{s}}}{9} \right\rfloor, -99\right) \tag{44}$$

$$y \equiv \begin{cases} 1 & \text{if } \|P(H)_\mathbf{U}\| = 0 \\ 2 & \text{otherwise} \end{cases}$$

$$\epsilon \equiv \left\lfloor 2^{\lfloor H_{\mathrm{i}}' \div 100000 \rfloor - 2} \right\rfloor \tag{45}$$

$$H_{\mathrm{i}}' \equiv \max(H_{\mathrm{i}} - 3000000, 0) \tag{46}$$

公式41–46为difficulty的计算方法。

公式41，42表示，当区块编号为0的时候其难度值是固定好的，在这里用D_0表示，其值为131072.对于其他区块，其难度值需要根据其父区块难度值以及一些其他因素，出块的间隔时间，区块编号等有关进行调节的，若小于D_0，则难度值调整为D_0。

公式43，调节系数（the adjustment factor ）x的定义。

公式44，难度系数（diculty parameter）ς_2的定义。该系数主要与出块间隔时间有关，当间隔大的时候，系数变大，难度也会相应变大，当间隔小的时候，系数变小，难度也会变小。使得区块链在整体上出块时间是趋于稳定的。其中y值根据父节点的uncle节点是否为空而有所区别，可以看出当父节点的uncle不为空的时候，y值为2，说明当前的分叉程度较大，适当调大难度，一定程度上会减少分叉。

公式45，46，“difficulty bomb”, or “ice age” ϵ的定义。（看说明好像是为了将来切PoS共识的时候，调节难度用）

```
// 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 (ethash *Ethash) CalcDifficulty(chain consensus.ChainReader, time uint64, par
        return CalcDifficulty(chain.Config(), time, parent)
}

// 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.Header)
        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)
        }
}
```

```go
// 黄皮书中的介绍的是Byzantium难度协议，所以这里只给出相应的代码。其他几种难度调节协议只是参数值
// calcDifficultyByzantium 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. The calculation uses the Byzantium rules.
func calcDifficultyByzantium(time uint64, parent *types.Header) *big.Int {
        // https://github.com/ethereum/EIPs/issues/100.
        // algorithm:
        // diff = (parent_diff +
        //         (parent_diff / 2048 * max((2 if len(parent.uncles) else 1) - ((
        //        ) + 2^(periodCount - 2)

        bigTime := new(big.Int).SetUint64(time)
        bigParentTime := new(big.Int).Set(parent.Time)

        // holds intermediate values to make the algo easier to read & audit
        x := new(big.Int)
        y := new(big.Int)

        // (2 if len(parent_uncles) else 1) - (block_timestamp - parent_timestamp)
        x.Sub(bigTime, bigParentTime)
        x.Div(x, big9)
        if parent.UncleHash == types.EmptyUncleHash {
                x.Sub(big1, x)
        } else {
                x.Sub(big2, x)
        }
        // max((2 if len(parent_uncles) else 1) - (block_timestamp - parent_timest
        if x.Cmp(bigMinus99) < 0 {
                x.Set(bigMinus99)
        }
        // parent_diff + (parent_diff / 2048 * max((2 if len(parent.uncles) else 1
        y.Div(parent.Difficulty, params.DifficultyBoundDivisor)
        x.Mul(y, x)
        x.Add(parent.Difficulty, x)

        // minimum difficulty can ever be (before exponential factor)
        if x.Cmp(params.MinimumDifficulty) < 0 {
                x.Set(params.MinimumDifficulty)
        }
        // calculate a fake block number for the ice-age delay:
        //   https://github.com/ethereum/EIPs/pull/669
        //   fake_block_number = min(0, block.number - 3_000_000
        fakeBlockNumber := new(big.Int)
        if parent.Number.Cmp(big2999999) >= 0 {
                fakeBlockNumber = fakeBlockNumber.Sub(parent.Number, big2999999) /
        }
```

```
        // for the exponential factor
        periodCount := fakeBlockNumber
        periodCount.Div(periodCount, expDiffPeriod)

        // the exponential factor, commonly referred to as "the bomb"
        // diff = diff + 2^(periodCount - 2)
        if periodCount.Cmp(big1) > 0 {
                y.Sub(periodCount, big2)
                y.Exp(big2, y, nil)
                x.Add(x, y)
        }
        return x
}
```

gasLimit限制的符号化表示

$$
\begin{aligned}
& H_{\mathrm{l}} < P(H)_{H_{\mathrm{l}}} + \left\lfloor \frac{P(H)_{H_{\mathrm{l}}}}{1024} \right\rfloor \quad \wedge \\
& H_{\mathrm{l}} > P(H)_{H_{\mathrm{l}}} - \left\lfloor \frac{P(H)_{H_{\mathrm{l}}}}{1024} \right\rfloor \quad \wedge \\
& H_{\mathrm{l}} \geqslant 5000
\end{aligned} \tag{47}
$$

公式47表示区块的gasLimit必须大于等于5000，且其和上一个区块的gasLimit差值不超过$\left\lfloor \frac{P(H)_{H_l}}{1024} \right\rfloor$

时间戳的符号化表示

$$
H_{\mathrm{s}} > P(H)_{H_{\mathrm{s}}} \tag{48}
$$

公式48表示当前区块的时间戳必须大于父区块的时间戳。（代码中要求区块的时间戳不能比当前时间大15秒以上）

mixHash和nonce相关符号化表示

$$
n \leqslant \frac{2^{256}}{H_{\mathrm{d}}} \quad \wedge \quad m = H_{\mathrm{m}} \tag{49}
$$

公式49表示，nonce值和mixHash需要满足PoW。

区块头验证的符号化表示

$$
\begin{aligned}
V(H) \equiv n \leqslant \frac{2^{256}}{H_{\mathrm{d}}} \wedge m = H_{\mathrm{m}} \quad \wedge \\
H_{\mathrm{d}} = D(H) \quad \wedge \\
H_{\mathrm{g}} \leq H_{\mathrm{l}} \quad \wedge \\
H_{\mathrm{l}} < P(H)_{H1} + \left\lfloor \frac{P(H)_{H_{\mathrm{l}}}}{1024} \right\rfloor \quad \wedge \\
H_{\mathrm{l}} > P(H)_{H1} - \left\lfloor \frac{P(H)_{H_{\mathrm{l}}}}{1024} \right\rfloor \quad \wedge \\
H_{\mathrm{l}} \geqslant 5000 \quad \wedge \\
H_{\mathrm{s}} > P(H)_{H\mathrm{s}} \quad \wedge \\
H_{\mathrm{i}} = P(H)_{H\mathrm{i}} + 1 \quad \wedge \\
\|H_{\mathrm{x}}\| \leq 32
\end{aligned}
\tag{50}
$$

相关的含义见本小节开始部分对区块头验证的那几点。

图片来源于网络侵删

< Raft在etcd中的实现（五）snapshot相关 以太坊黄皮书详解（二） >