以太坊学习-周汇报

刘亮鑫

2023-9-15

目录

1 以太坊基础知识

1.1 数据结构

1.1.1 stateRoot

1. 账户状态：
    - 区块链中的每个账户都有一个状态，包括账户地址、余额和合约状态。
    - 账户地址是唯一标识账户的以太坊地址，以 0x 开头的 40 位十六进制字符串。
    - 余额表示账户中的以太币（ETH）数量。
    - 合约状态是合约账户的特有状态，包括合约的存储数据和代码。
2. 状态树的结构：
    - 状态树是一个树状数据结构，类似于默克尔树（Merkle Tree）。
    - 每个叶子节点表示一个账户的状态，其中包括账户地址、余额和合约状态。
    - 账户地址经过哈希函数处理，形成节点的键（key）。
    - 状态树的根节点包含了整个状态树的哈希值，通常称为“状态树根”。
3. 状态的变化：
    - 当用户执行一笔交易（例如，转账或调用智能合约）时，状态树会根据交易的影响进行更新。
    - 如果是转账交易，状态树会减少发送者账户的余额，增加接收者账户的余额。
    - 如果是智能合约交易，合约代码可能会修改状态树中的合约状态。
4. 存储和验证：
    - 区块链网络的每个节点都存储了完整的状态树，以便验证交易和区块。
    - 当用户查询某个账户的余额时，节点可以通过状态树快速检索。
    - 状态树的哈希值在区块头中，用于验证区块的完整性。
5. 不可变性：
    - 一旦状态树的某个状态被写入，它就变得不可变，不能直接修改。
    - 任何修改都需要通过执行交易来更新状态，交易会产生新的状态树。
6. 状态树的优势：
    - 状态树使区块链具有不可篡改性，因为历史状态不能轻易更改。
    - 它提供了透明的账户余额和状态查询功能，允许用户验证账户的真实性。
    - 状态树的根哈希值被包括在区块头中，可以确保区块链的完整性和一致性。

1.1.2 stateRoot 原理

状态树构造的原理：数据压缩存储

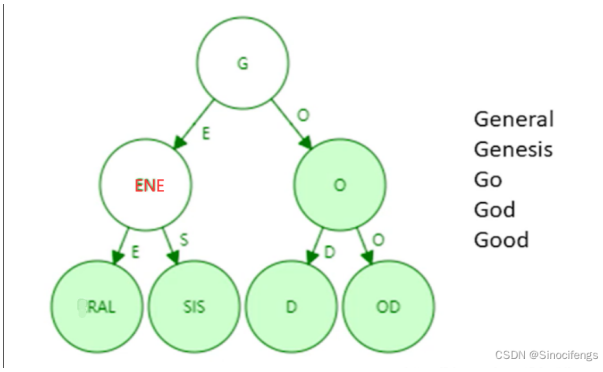

图 1: Patricia trie

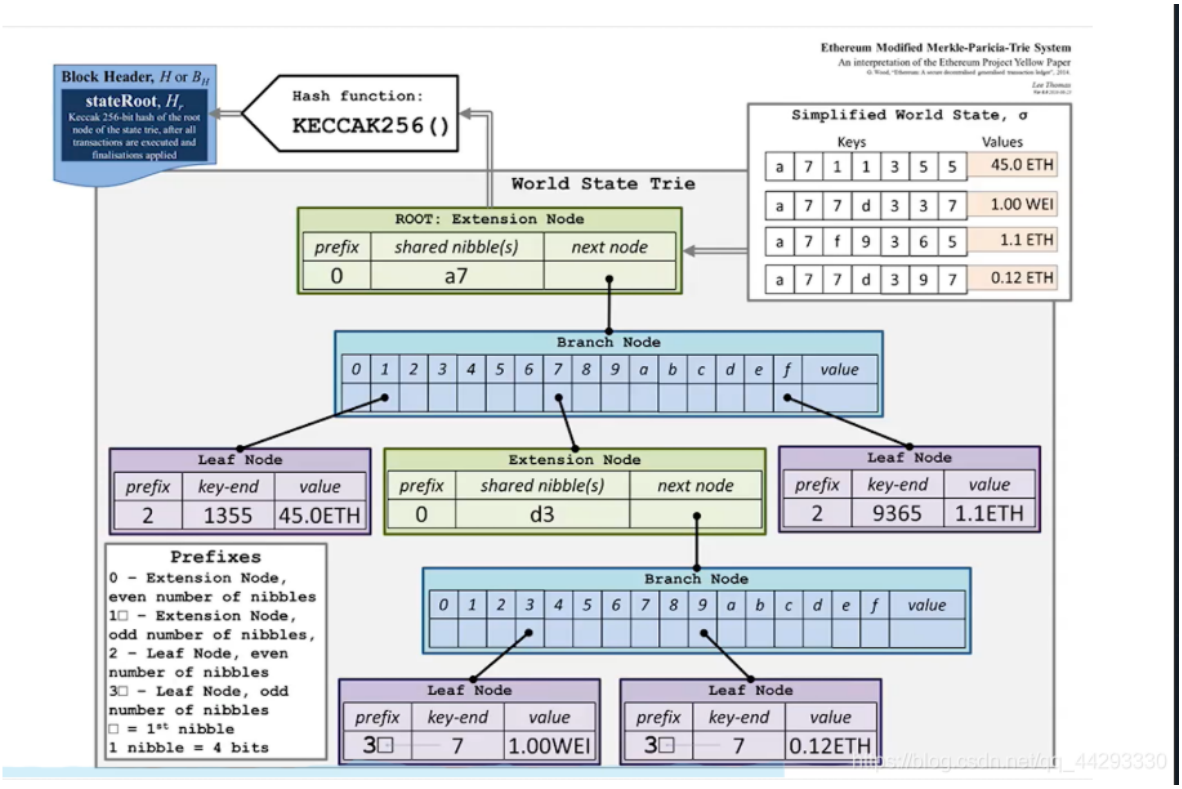

图 2: 以太坊状态树

1.1.3 交易树和收据树

每次发布一个区块时，区块中的交易会形成一颗 Merkle Tree，即交易树。此外，以太坊还添加了一个收据树，每个交易执行完之后形成一个收据，记录交易相关信息。也就是说，交易树和收据树上的节点是一一对应的。由于以太坊智能合约执行较为复杂，

通过增加收据树，便于快速查询执行结果。交易树和收据树都是 M(Merkle)PT，而 BTC 中都采用普通的 MT(Merkle Tree)。（可能就仅仅是为了三棵树代码复用好所以这样设计的）MPT 的好处是支持查找操作，通过键值沿着树进行查找即可。对于状态树，查找键值为账户地址；对于交易树和收据树，查找键值为交易在发布的区块中的序号。

交易树和收据树只将当前区块中的交易组织起来，而状态树将所有账户的状态都包含进去，无论这些账户是否与当前区块中交易有关系。多个区块状态树共享节点，而交易树和收据树依照区块独立。

交易树和收据树的用途：

1. 向轻节点提供 Merkle Proof。
2. 更加复杂的查找操作

1.1.4 Bloom filter(布隆过滤器)

数据查询

Bloom Filter（布隆过滤器）是一种数据结构，用于高效地检查一个元素是否属于一个集合。它通过使用位向量和一组哈希函数来实现这一目标。布隆过滤器的主要优势在于它的高效性和低内存占用。

- 全面讲解链接:https://blog.csdn.net/qq_41125219/article/details/119982158

1.2 共识算法

1.2.1 GHOST 协议

根据自己查阅的共识协议文献，已经详细研究过其属性

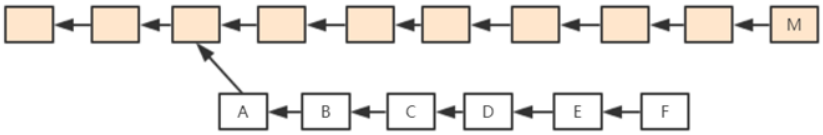

图 3: GHSOT 协议

- 最长链规则
- GHOST 协议
- Tree 结构
- DAG 结构

表 1: Structure: consensus protocols comparison

Protocol Name	Consensus Algorithm	Consensus Structure	Unit	Consensus methods	Model
Bitcoin	Longest Chain	Tree	Block	Pow	Open
GHOST	Weight	Tree	Block	Pow	Open
Ouro.	Longest Chain	Tree	Block	Elect	Open
C-Nak-Pos	Longest Chain	Tree	Block	Pos	Open
Scalable-Pos	Longest Chain	Tree	Block	Pos	Open
IOTA	Divergence	DAG	Bundle	Blockless	Openless
Nano	Parallel	DAG	Tx	Trading pair	Openless
Hashgraph	Parallel	DAG	Event	Gsp by Gsp	Open
Spectre	Divergence	DAG	Block	Pairwise vote	Openless
Phantom	Divergence	DAG	Block	k-cluster	Openless
Meshcash	Divergence	DAG	Block	Layered ptcl	Openless
CDAG	Convergence	DAG	Block	C-Block	Openless
Conflux	Convergence	DAG	Block	Adaptive ptcl	Openless
Inclusive	Convergence	DAG	Block	Invole Uncles	Openless

1.2.2 POW-POS 协议

改进 Pow 协议，已经详细研究过其算法流程

见表2和图4

1.3 智能合约应用

进一步的工作。。。

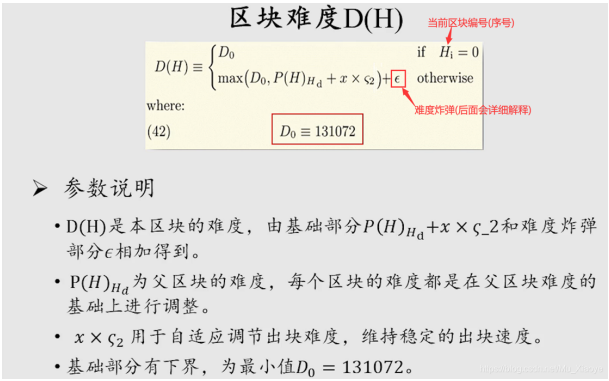

图 4: 难度调整

表 2: Type: consensus protocols omparison

Protocol Name	Consensus Algorithm	Consensus Type	Network Model	Fault Tolerance	Communication Complexity	Latency
Algorand	VRF	PoS	syn	$\alpha < 1/3$	$O(cn)$	$O(\Delta) \sim \infty$
Ouroboros	PVSS	PoS	syn	$\alpha < 1/2$	$O(n^3)$	$O(\Delta)$
Ouro.Praos	VRF	PoS	part-syn	$\alpha < 1/2$	$O(n)$	$O(\Delta) \sim \infty$
Ouro.Genesis	VRF	PoS	part-syn	$\alpha < 1/2$	$O(n)$	$O(\Delta) \sim \infty$
RandShare	PVSS	-	asyn	$\alpha < 1/3$	$O(n^3)$	$O(\delta)$
RandHound	PVSS	-	syn	$\alpha < 1/3$	$O(c^2 n)$	$O(\Delta) \sim \infty$
RandHerd	PVSS	-	syn	$\infty < 1/3$	$O(c^2 \log n)$	$O(\delta)$
Dfinity	VRF+Thr.Sig	PoS	asyn	$\infty < 1/3$	$O(cn)$	$O(\Delta) \sim \infty$
RandChain	SeqPoW	PoW	syn	$\alpha < 1/2$	$O(n)$	$t + \delta$
StrongChain	Weak-solutions	PoW	syn	$\alpha < 1/2$	$O(n)$	-
Crystal	VRF+QC	PoW	syn	$\alpha < 1/2$	$O(n)$	$t + 2\Delta$
Bitcoin	Hash(-) $\leq$ T	PoW	syn	$\alpha < 1/2$	-	-
Bitcoin-NG	Key block	PoW	syn	$\propto < 1/2$	-	$O(\Delta)$
ByzCoin	CoSi	PoW	syn	$\propto < 1/2$	-	$O(\Delta)$
Tendermint	Round-Robin	PoS	syn	$\alpha < 1/3$	$O(cn)$	$O(\delta)$
Cont-VDF	VDF	PoW	asyn	$\alpha < 1/2$	$O(n)$	$O(\Delta)$

2 以太坊编译

2.1 geth 客户端

- geth 客户端部署链接：`https://geth.ethereum.org/docs/getting-started`

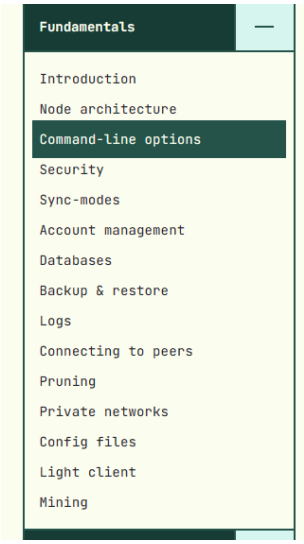

图 5: introduce

初始化文件配置：

```
{  "config": {
        "chainId": 15,
        "homesteadBlock": 0,
        "eip155Block": 0,
        "eip158Block": 0},
    "coinbase" : "0x0000000000000000000000000000000000000000"
      ,
    "difficulty" : "0x40000",
    "extraData" : "",
    "gasLimit" : "0xffffffff",
    "nonce" : "0x0000000000000042",
    "mixhash" : "0x0000000000000000000000000000000000000000000000000"
      ,
    "parentHash" : "0x0000000000000000000000000000000000000000"
      ,
    "timestamp" : "0x00",
    "alloc": {}}
```

2.2 Geth 客户端使用

图 6: 交互

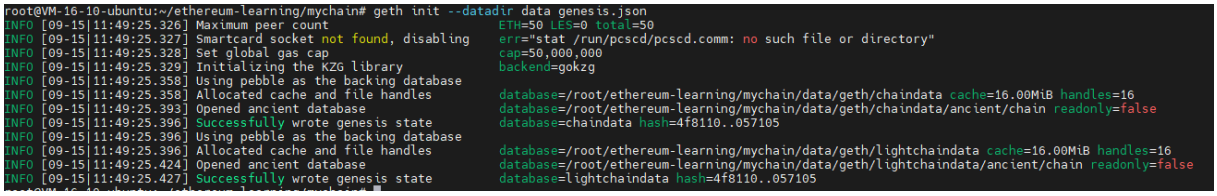

图 7: 初始化

1. 常用命令:
   //查看区块链数 eth.blockNumber

2. 创建账户
   eth.accounts eth.getBalance(user1) personal.newAccount(”123456”) // 列出当前所有账户 personal.listAccounts
   // coinbase 账户 eth.coinbase
   // 钱包信息 personal.listWallets

3. 挖矿
   // 查询余额参数为账户地址, 这里查询的是矿工地址 eth.getBalance(eth.coinbase)
   // 挖矿开始 miner.start() miner.stop() eth.getTransaction() // 挖矿结束 miner.stop()

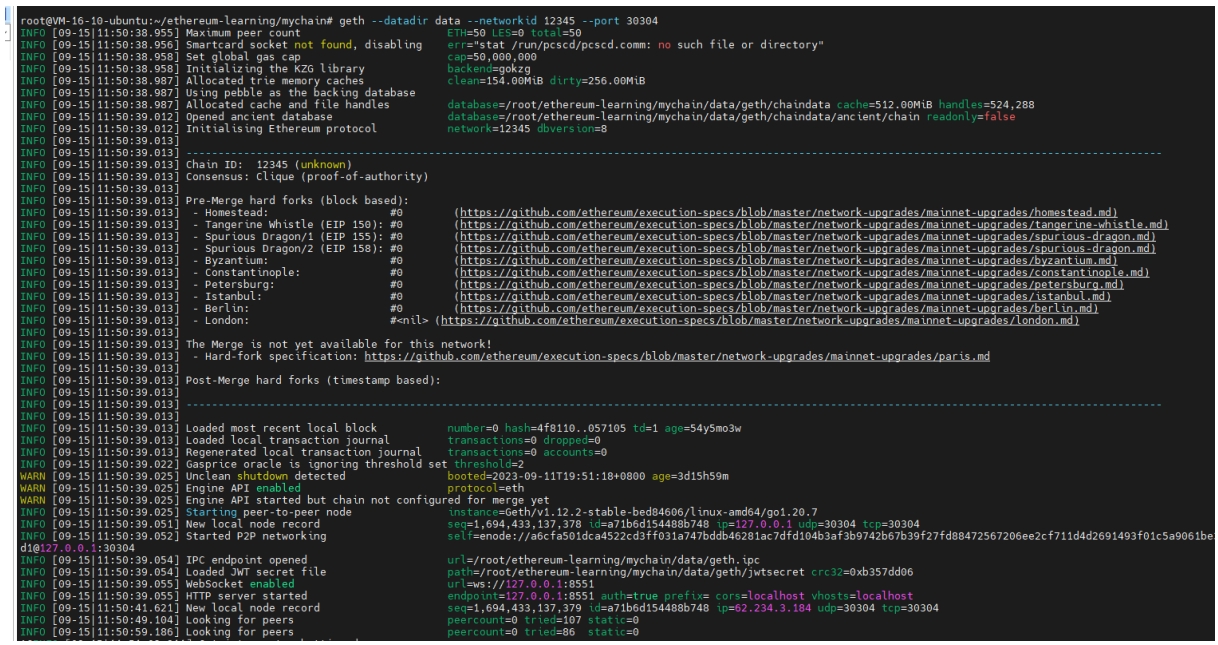

图 8: 私有链启动

4. 转账
   eth.sendtransaction()

5. 链互通

2.3 智能合约的应用

进一步的工作:。。。

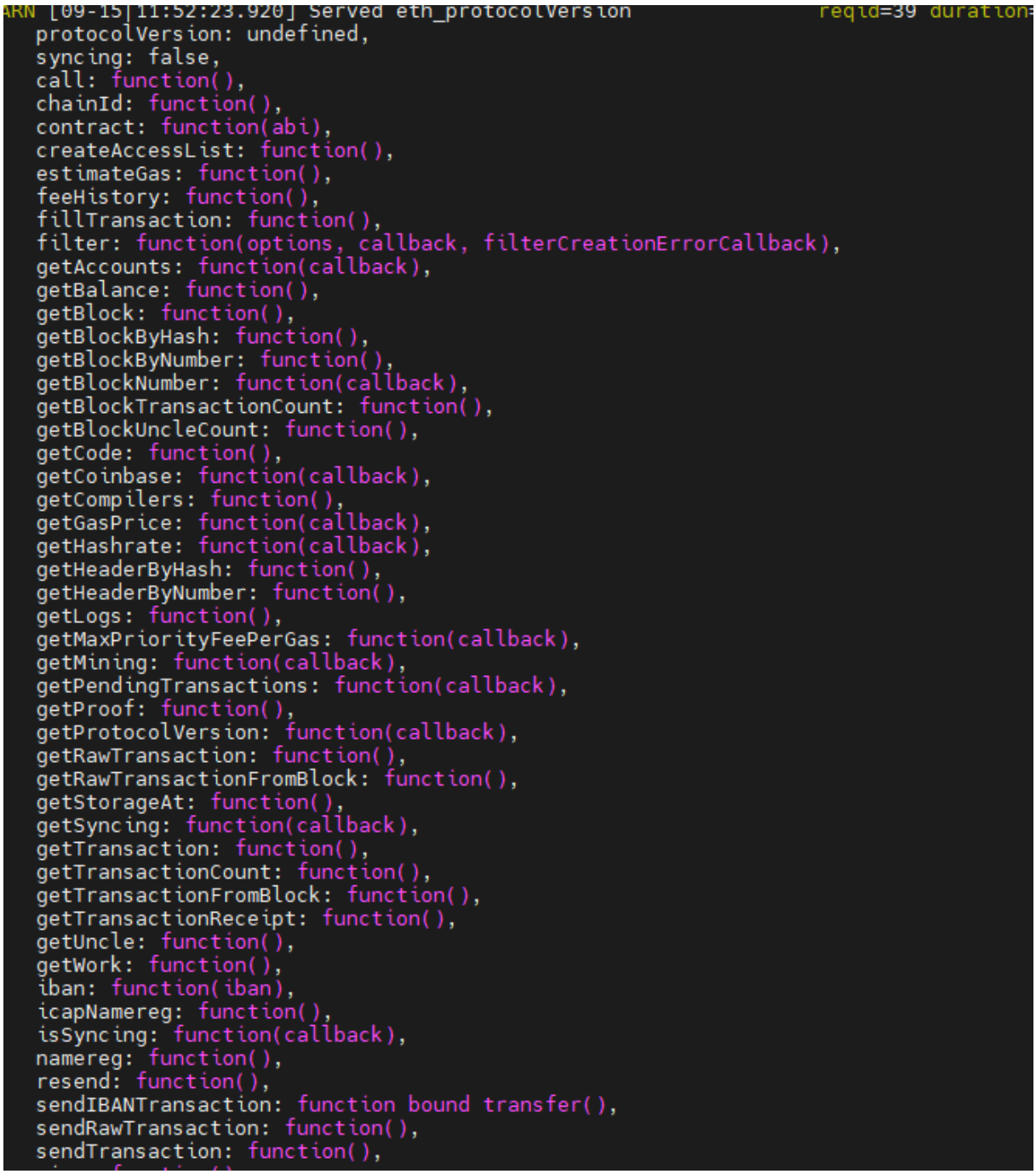
ARN [09-15|11:52:23.920] Served eth_protocolVersion reqid=39 duration=
protocolVersion: undefined,
syncing: false,
call: function(),
chainId: function(),
contract: function(abi),
createAccessList: function(),
estimateGas: function(),
feeHistory: function(),
fillTransaction: function(),
filter: function(options, callback, filterCreationErrorCallback),
getAccounts: function(callback),
getBalance: function(),
getBlock: function(),
getBlockByHash: function(),
getBlockByNumber: function(),
getBlockNumber: function(callback),
getBlockTransactionCount: function(),
getBlockUncleCount: function(),
getCode: function(),
getCoinbase: function(callback),
getCompilers: function(),
getGasPrice: function(callback),
getHashrate: function(callback),
getHeaderByHash: function(),
getHeaderByNumber: function(),
getLogs: function(),
getMaxPriorityFeePerGas: function(callback),
getMining: function(callback),
getPendingTransactions: function(callback),
getProof: function(),
getProtocolVersion: function(callback),
getRawTransaction: function(),
getRawTransactionFromBlock: function(),
getStorageAt: function(),
getSyncing: function(callback),
getTransaction: function(),
getTransactionCount: function(),
getTransactionFromBlock: function(),
getTransactionReceipt: function(),
getUncle: function(),
getWork: function(),
iban: function(iban),
icapNamereg: function(),
isSyncing: function(callback),
namereg: function(),
resend: function(),
sendIBANTransaction: function bound transfer(),
sendRawTransaction: function(),
sendTransaction: function(),

图 9: 可调用函数

3 以太坊源码分析

链接：https://github.com/chaors/Ethereum_read/blob/master/Docs/0x06%20MPT%E6%BA%90%E7%A0%81%E8%A7%A3%E6%9E%90.md

3.1 区块数据结构

```
type Genesis struct {
    // 配置文件，用于指定链的chainId(network id)
    Config     *params.ChainConfig `json:"config"`
    // 随机数，与Mixhash组合用于满足POW算法要求
    Nonce      uint64              `json:"nonce"`
    // 时间戳
    Timestamp  uint64              `json:"timestamp"`
    // 区块额外信息
    ExtraData  []byte              `json:"extraData"`
    // Gas消耗量限制
    GasLimit   uint64              `json:"gasLimit"   gencodec:"required"`
    // 区块难度值
    Difficulty *big.Int            `json:"difficulty" gencodec:"required"`
    // 由上个区块的一部分生成的Hash，和Nonce组合用于找到满足POW算法的条件
    Mixhash    common.Hash         `json:"mixHash"`
    // 矿工地址
    Coinbase   common.Address      `json:"coinbase"`
    // 创世区块初始状态
    Alloc      GenesisAlloc        `json:"alloc"      gencodec:"required"`

    // These fields are used for consensus tests. Please don't use them
    // in actual genesis blocks.
    /** 下面字段用于共识测试
    */
    Number     uint64      `json:"number"`
    GasUsed    uint64      `json:"gasUsed"`
```

```
    // 父区块哈希
    ParentHash common.Hash `json:"parentHash"`}
```

3.2 区块头数据结构

```
type Header struct {
    // ParentHash 是前一个区块的哈希值。
    ParentHash common.Hash `json:"parentHash" gencodec:"required"`

    // UncleHash 是叔块的哈希值。
    UncleHash common.Hash `json:"sha3Uncles" gencodec:"required"`

    // Coinbase 是挖矿奖励的接收地址。
    Coinbase common.Address `json:"miner"`

    // Root 是状态树的根哈希值。
    Root common.Hash `json:"stateRoot" gencodec:"required"`

    // TxHash 是交易树的根哈希值。
    TxHash common.Hash `json:"transactionsRoot" gencodec:"required"`

    // ReceiptHash 是收据树的根哈希值。
    ReceiptHash common.Hash `json:"receiptsRoot" gencodec:"required"`

    // Bloom 是 Bloom 过滤器，用于快速查询日志。
    Bloom Bloom `json:"logsBloom" gencodec:"required"`

    // Difficulty 是挖矿的难度。
    Difficulty *big.Int `json:"difficulty" gencodec:"required"`

    // Number 是区块的高度。
    Number *big.Int `json:"number" gencodec:"required"`

```

```
    // GasLimit 是区块中的 gas 限制。
    GasLimit uint64 `json:"gasLimit" gencodec:"required"`

    // GasUsed 是已使用的 gas 数量。
    GasUsed uint64 `json:"gasUsed" gencodec:"required"`

    // Time 是区块的时间戳。
    Time uint64 `json:"timestamp" gencodec:"required"`

    // Extra 是区块的附加数据。
    Extra []byte `json:"extraData" gencodec:"required"`

    // MixDigest 是用于 PoW 挖矿的混合哈希。
    MixDigest common.Hash `json:"mixHash"`

    // Nonce 是挖矿的随机数。
    Nonce BlockNonce `json:"nonce"`

    // BaseFee 是 EIP-1559 中引入的基本费用，对于传统头部被忽略。
    BaseFee *big.Int `json:"baseFeePerGas" rlp:"optional"`

    // WithdrawalsHash 是 EIP-4895 中引入的提款哈希，对于传统头部被忽略。
    WithdrawalsHash *common.Hash `json:"withdrawalsRoot" rlp:"optional"`

    // BlobGasUsed 是 EIP-4844 中引入的 BlobGasUsed，对于传统头部被忽略。
    BlobGasUsed *uint64 `json:"blobGasUsed" rlp:"optional"`

    // ExcessBlobGas 是 EIP-4844 中引入的 ExcessBlobGas，对于传统头部被忽略。
    ExcessBlobGas *uint64 `json:"excessBlobGas" rlp:"optional"`

    // ParentBeaconRoot 是 EIP-4788 中引入的 ParentBeaconRoot，对于传统头部被忽略。
```

```
    ParentBeaconRoot *common.Hash `json:"parentBeaconBlockRoot" rlp:"optional"`}
```

3.3 交易结构

```
//交易结构体
type Transaction struct {
    //交易数据
    data txdata
    // caches
    hash atomic.Value
    size atomic.Value
    from atomic.Value
}

type txdata struct {

    //发送者发起的交易总数
    AccountNonce uint64          `json:"nonce"    gencodec:"required"`
    //交易的Gas价格
    Price        *big.Int        `json:"gasPrice" gencodec:"required"`
    //交易允许消耗的最大Gas
    GasLimit     uint64          `json:"gas"      gencodec:"required"`
    //交易接收者地址
    Recipient    *common.Address `json:"to"       rlp:"nil"` // nil means contract creation
    //交易额
    Amount       *big.Int        `json:"value"    gencodec:"required"`
    //其他数据
    Payload      []byte          `json:"input"    gencodec:"required"`

    // Signature values
```

```
    // 交易相关签名数据
    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.
    //交易HAsh
    Hash *common.Hash `json:"hash" rlp:"-"`
}
```

3.4 mpt 树

```
// MPT几种节点结构
type (
    // 分支节点，它的结构体现了原生trie的设计特点
    fullNode struct {
        // 17个子节点，其中16个为0x0-0xf;第17个子节点存放数据
        Children [17]node // Actual trie node data to encode/decode (needs custom encoder)
        // 缓存节点的Hash值，同时标记dirty值来决定节点是否必须写入数据库
        flags    nodeFlag
    }
    // 扩展节点和叶子节点，它的结构体现了PatriciaTrie的设计特点
    // 区别在于扩展节点的value指向下一个节点的hash值(hashNode)；叶子节点的value是数据的RLP编码(valueNode)
    shortNode struct {
        Key   []byte
        Val   node
        flags nodeFlag
    }
    //节点哈希，用于实现节点的折叠(参考MerkleTree设计特点)
    hashNode  []byte
    //存储数据
    valueNode []byte
)
```