16以太坊状态树

我们需要完成的任务是一个用户地址到用户状态的映射。以太坊中用的用户地址是160位，20个字节，40个16进制的数。外部账户和合约账户的状态。

那么我们设计什么样的数据结构来实现这个映射呢？像个key-value pair，一个用户地址是key，状态是value。所以可以设计成一个哈希表。但这样的设计有什么问题呢？如果需要提供merkle proof，怎么才能提供呢？比如说你要和一个人签合同，需要另外一个人证明一下他有多少钱，那么提供这个证明呢？一种办法就是把哈希表中存储的状态组织成一个merkle tree然后将根哈希发布出去，这样别人就可以验证你自己有没有修改。但这样的问题是，因为以太坊是账户制，一旦发布新的交易之后，就肯定会修改哈希表中的内容（因为需要修改余额），那就需要再重新把哈希表中所有的状态再组织成一个新的merkle tree。这个代价太大了。实际上真正发生变化的账户是一小部分，只有发生交易的那些账户的余额会变化。比特币为什么没有这个问题呢？因为比特币是transaction based，所以merkle tree存放的是这个区块里的交易信息，而这些交易信息是不能被修改的。所以没有代价，因为每个区块里包含的交易都是不同的，所以会组织成不同的merkle tree。而且比特币一个区块最多只包含4000个左右的交易，也就是最多把4000个leaf node组织成一个merkle tree。然而以太坊如果要构建merkle tree，需要把所有的账户遍历一遍构建出一棵新的merkle tree。代价太大了。merkle tree除了可以提供merkle proof，还可以用来保证所有全节点之间状态的一致性。

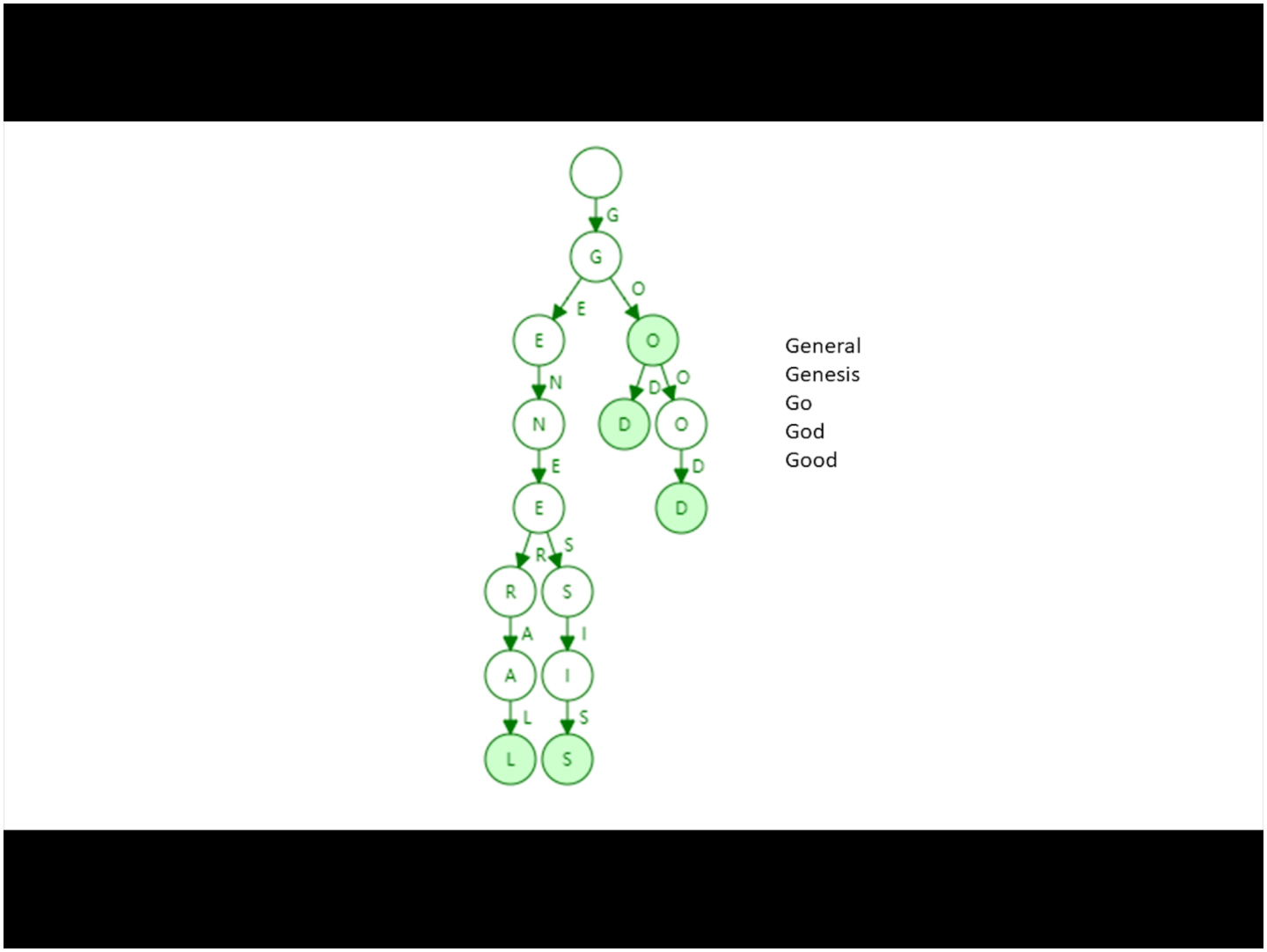
有没有可能不用哈希表了，直接用merkle tree把所有用户存起来，每次交易改动的只是一小部分账户的信息。但这样的问题是merkle tree其实没有提供一个高效的查找和修改的办法。还有一个问题是如果这样构建merkle tree，那么这个merkle tree需不需要将账户按照一定顺序来排序。如果不排序查找的速度会很慢，而且没有办法证明一个交易不在这个区块里non-membership。除此之外，leaf node是账户的信息，如果不规定顺序的话，构建merkle tree的方式就不唯一，导致算出来的根哈希值是不一样的。但比特币也是不排序的，为什么比特币不存在这个问题？其实比特币中也会构建出来不同的merkle tree，并没有规定必须要包含某些交易。每个节点在本地构建一个候选区块，节点自己决定包含哪些交易和顺序。但最后只有有记账权的那个节点说了才算，你在本地是怎么构造的，只要最后没有抢得记账权，都是没用的。为什么以太坊不能这样呢？如果以太坊也这样干的话，需要把账户的信息发布到区块链上。比如说每个节点自己决定账户信息排列的顺序然后组织成一个自己的merkle tree，抢到记账权之后发布到区块链上。但是因为账户信息的数量和交易的数量的数量级相差甚远，导致如果将所有的账户信息都上链的话是不太可行的。

那么用Sorted Merkle Tree？那么每产生一个新的账户的时候，就需要让所有的节点都知道。否则如果新的账户插入在这个merkle tree里的时候，可能会导致不一致。所以即使用Sorted Merkle Tree，插入的代价是非常大的，可能会导致半棵树重新构造。

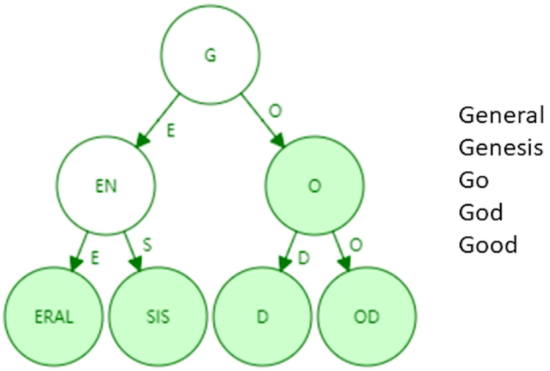
以太坊采用的是MPT。Merkle Patricia Trie。

先讲一下Trie。英文中有个单词叫做retrieval。就是从这来的，中文叫做字典树。

这就是一个Trie。每个节点的分叉数量，取决于key的取值范围。比如说这里是字母，那么每个分支最多只有26种可能。以太坊中，每个账户的地址可以写成40个16进制数字，那么分支最多有17种可能性，就是0-f加上终点。第二就是Trie的查询速度取决于分支的长度。在以太坊里因为账户都是40个十六进制数字，所以长度都是40。第三就是Trie是不会发生碰撞的，就是输入的两串数字不一样，最后一定会在不同的节点结束。第四就是给定一组输入，只要这组输入的内容不变，那么构造出来的Trie就是一模一样的。第五就是每当更新一个账户的时候，只需要在对应的分支上修改，而不需要影响其他的地方。但是Trie的问题就是比较浪费存储空间。

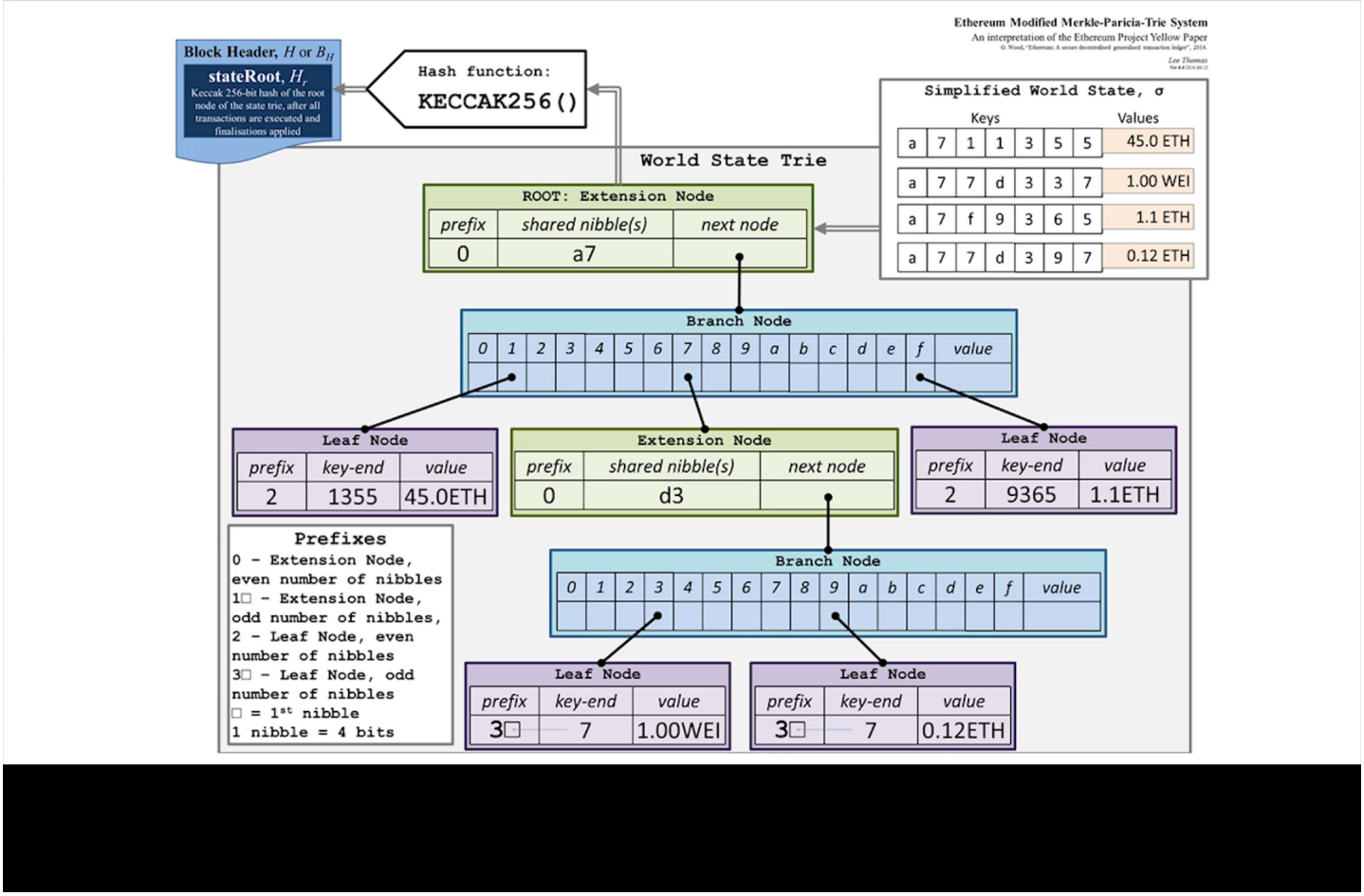


为了解决这个问题，我们引入了Patricia tree，或者叫做Patricia trie。就是加入了路径压缩。这么做的好处就是明显树变矮了，需要的内存空间就少了。但是这样的问题就是想插入一个新的单词的时候，会需要重新进入压缩。



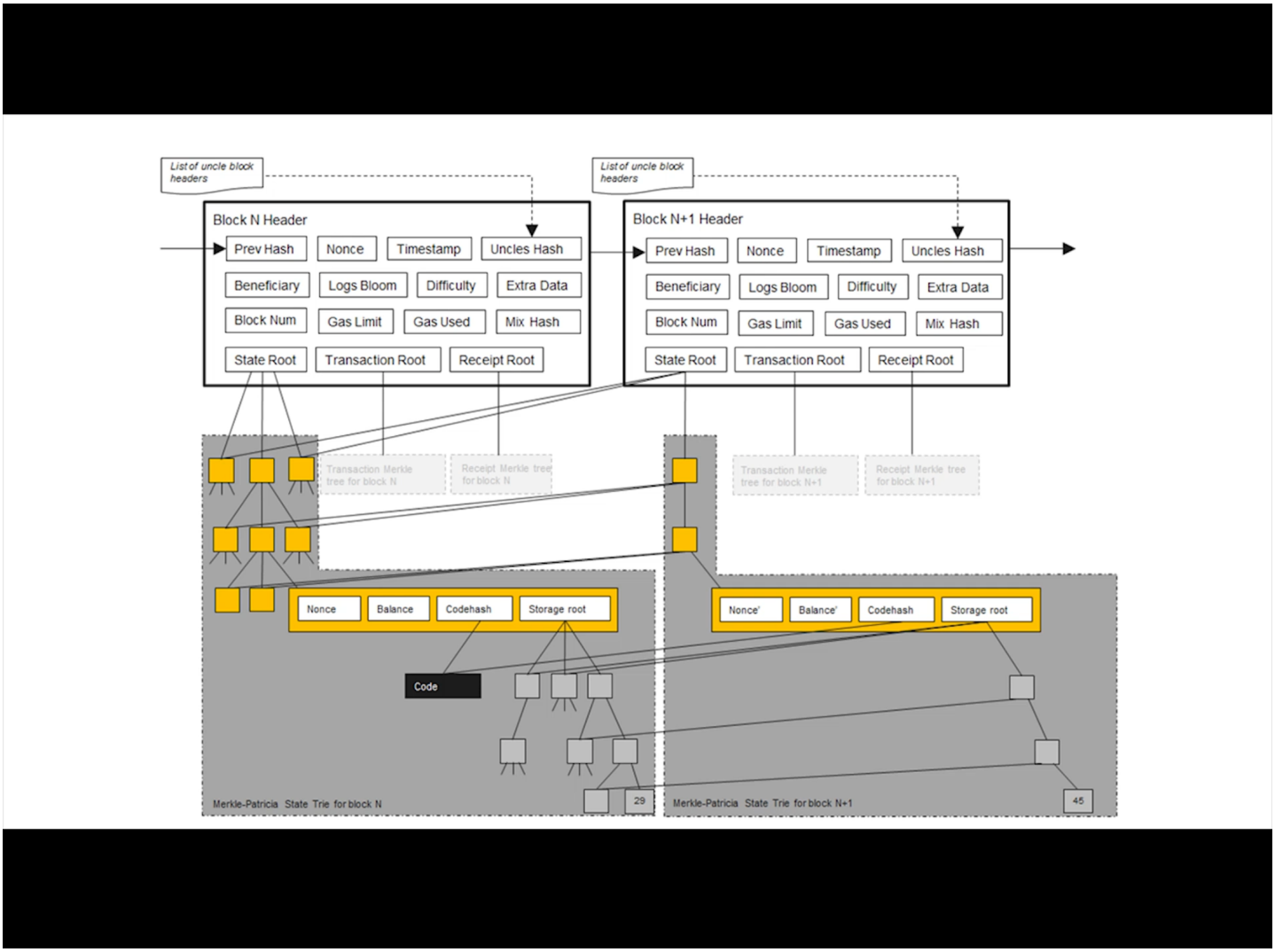
那么时候应该进行压缩呢？如果输入的内容分布的比较稀疏的时候，做压缩就比较有效果。如果比较密集，就没有效果了。以太坊的账户是160 bits，那么账户的空间就是2的160次方，所以这是一个非常大的数，所以账户的地址分布就会是非常非常稀疏的。这么稀疏的意义，就是防止我一直在本地产生地址去和别人的地址发生碰撞。

MPT是Merkle Patricia Tree。MPT和PT有啥区别呢？把普通指针换成了哈希指针。所以就可以计算出来一个根哈希值，然后这个根哈希值也会写在以太坊的区块的块头中的。这样可以保证每个账户的状态不会被私自篡改，还可以用来做merkle proof。直接把你账户所在的这个分支从leaf node到root node的路径发送给轻节点作为merkle proof。除此之外，还可以用来证明non-membership。直接看如果一个账户的地址存在，应该出现在这个MPT的那个位置，然后看看有没有，如果没有的话就说明不存在。



extension node是压缩路径的，leaf node就是最后的，branch node就是分开的。

以太坊中的合约账户的storage也是一颗小的MPT。用来维护变量和变量取值。



当storage中的一个变量值发生改变的时候，只需要改这个变量值，然后重新算哈希的时候，没变的变量值可以直接用上一个区块中的。

所以系统中每个节点需要维护的不是一个MPT，而是每次在一个新的区块中新建一个新的MPT，只是这个新的MPT中只有少量的值发生了修改，大部分用的还是原来的MPT中的信息。

但是为什么不直接在原有的MPT上修改呢？而是新建一个新的MPT。这是因为可能有两个节点同时获得了记账权，更新了两个区块到链上，这个时候就出现了分支。只有一个分支最终会成为最长合法链，而另外一个分支的节点没有办法直接丢弃那些区块然后接着最长合法链往下挖，因为余额已经发生了改变，所以必须回滚到之前的状态才可以。这个时候就体现了保存以前状态的重要性，否则无法进行回滚。

```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        *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"`
}
```

```
144  // Block represents an entire block in the Ethereum blockchain.
145  type Block struct {
146          header       *Header
147          uncles       []*Header
148          transactions Transactions
149
150          // caches
151          hash atomic.Value
152          size atomic.Value
153
154          // Td is used by package core to store the total difficulty
155          // of the chain up to and including the block.
156          td *big.Int
157
158          // These fields are used by package eth to track
159          // inter-peer block relay.
160          ReceivedAt   time.Time
161          ReceivedFrom interface{}
162  }
```

```
177  // "external" block encoding. used for eth protocol, etc.
178  type extblock struct {
179          Header *Header
180          Txs    []*Transaction
181          Uncles []*Header
182  }
```

每个账户的状态，其实是经过RLP(Recursive Length Prefix)序列化之后放到MTP里的。RLP的特点是极简主义。RLP只支持一种类型，就是nested array of bytes，就是字节嵌套的数组。