Research on MPT

一、概述

默克尔帕特里夏树（Merkle Patricia Tree）简称MPT树，MPT树结合了字典树和默克尔树的优点，在压缩字典树中根节点是空的，而MPT树可以在根节点保存整棵树的哈希校验和，而校验和的生成则是采用了和默克尔树生成一致的方式。Merkle Tree(默克尔树) 用于保证数据安全，Patricia Tree(基数树,也叫基数特里树或压缩前缀树) 用于提升树的读写效率。
以太坊采用MPT树来保存，交易，收据以及世界状态，为了压缩整体的树高，降低操作的复杂度，以太坊又对MPT树进行了一些工程上的优化。

字典树（Trie)

Trie又称前缀树或字典树，是一种有序多叉树.在计算机科学中， Trie是一种有序树，用于保存关联数组，其中的键通常是字符串。 与二叉查找树不同，键不是直接保存在节点中，而是由节点在树中的位置决定。 一个节点的所有子孙都有相同的前缀，也就是这个节点对应的字符串，而根节点对应空字符串。 一般情况下，不是所有的节点都有对应的值，只有叶子节点和部分内部节点所对应的键才有相关的值。

Trie 中的键通常是字符串，但也可以是其它的结构。 Trie 的算法可以很容易地修改为处理其它结构的有序序列，比如一串数字或者形状的排列。比如，bitwise Trie 中的键是一串比特，可以用于表示整数或者内存地址

优点

- 插入和查询的效率都很高，都是O(m)，m是插入或查询字符串的长度。
- 可以对数据按照字典序排序。

缺点

- 空间消耗的比较大。

典型场景

- 单词频次统计。
- 字符串匹配。
- 字符串字典序排序。
- 前缀匹配，比如一些搜索框的自动提示。

默克尔树（Merkle tree)

默克尔树首先计算叶子节点的hash值，然后将相邻两个节点的哈希进行合并，合并完成后计算这个字符串的哈希值，直到根节点为止，如果是单个节点，可以复制单节点的哈希，然后合并哈希再重复上面的过程。

优点

可以高效安全的验证数据结构的内容。

典型场景

p2p网络分块下载文件的时候，快速校验下载到的数据是否完整，是否遭到破坏。

默克尔帕特里夏树（Merkle Patricia Tree）

MPT树结合了字典树和默克尔树的优点，在压缩字典树中根节点是空的，而MPT树可以在根节点保存整棵树的哈希校验和，而校验和的生成则是采用了和默克尔树生成一致的方式。 以太坊采用MPT树来保存，交易，交易的收据以及世界状态，为了`压缩整体的树高，降低操作的复杂度`，以太坊又对MPT树进行了一些优化。将树节点分成了四种；

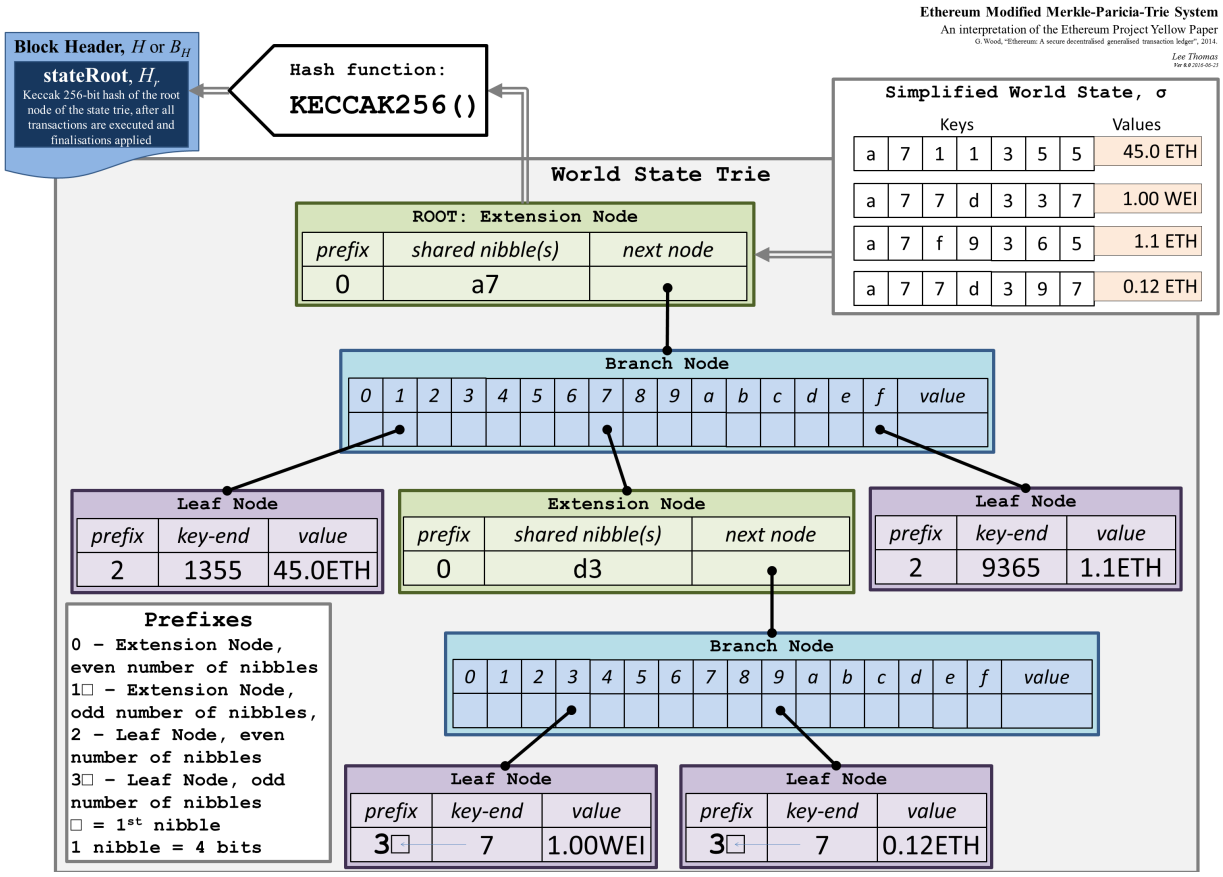
- fullNode: 分支节点，fullNode[16]的类型是 valueNode。前 16 个元素对应键中可能存在的一个十六进制字符。如果键[key,value]在对应的分支处结束，则在列表末尾存储 value 。
- shortNode: 叶子节点或者扩展节点，当 shortNode.Key的末尾字节是终止符`16`时表示为叶子节点。当 shortNode 是叶子节点是，Val 是 valueNode。
- hashNode: 应该取名为 collapsedNode 折叠节点更合适些，但因为其值是一个哈希值当做指针使用，所以取名 hashNode。使用这个哈希值可以从数据库读取节点数据展开节点。
- valueNode: 数据节点，实际的业务数据值，严格来说他不属于树中的节点，它只存在于 fullNode.Children 或者 shortNode.Val 中。

各类key

在改进过程中，为适应不同场景应用，以太坊定义了几种不同类型的 key 。

1. keybytes： 数据的原始 key
2. Secure Key: 是 Keccak256(keybytes) 结果，用于规避 key 深度攻击，长度固定为 32 字节。
3. Hex Key: 将 Key 进行半字节拆解后的 key ，用于 MPT 的树路径中和降低子节点水平宽度。
4. HP Key: Hex 前缀编码(hex prefix encoding)，在节点存持久化时，将对节点 key 进行压缩编码，并加入节点类型标签，以便从存储读取节点数据后可分辨节点类型。

通过以太坊黄皮书中很经典的一张图，来了解不同节点的具体结构和作用



可以看到有四个状态要存储在世界状态的MPT树中，需要存入的值是键值对的形式。自顶向下，我们首先看到的`keccak256`生成的根哈希，参考默克尔树的`Top Hash`，其次看到的是绿色的`扩展节点Extension Node`，其中`共同前缀shared nibble`是a7，采用了压缩前缀树的方式进行了合并，接着看到蓝色的`分支节点Branch Node`，其中有表示十六进制的字符和一个value，最后的value是fullnode的数据部分，最后看到紫色的`叶子节点leadfNode`用来存储具体的数据，它也是对路径进行了压缩。

MPT树持久化

在Go实现的以太坊中，MPT树最终是存储在LevelDB数据库中的，LevelDB是Google开源的持久化KV单机数据库，具有很高的随机写，顺序读/写性能。以太坊通过对叶子节点按照一定的编码规则编码后存入LevelDB。

以太坊的MPT树提供了三种不同的编码方式来满足不同场景的不同需求，三种编码方式为；

- Raw编码（原生字符）
- Hex编码（扩展16进制编码）
- Hex-Prefix编码（16进制前缀编码）

三者的关系如下图所示，分别解决的是MPT对外提供接口的编码，在内存中的编码，和持久化到数据库中的编码。

Raw编码

MPT对外提供的API采用的就是Raw编码方式，这种编码方式不会对key进行修改，如果key是“foo”, value是"bar"，编码后的key就是["f", "o", "o"]。

假设我们要把`a`作为key放入MPT树，key可以直接用`a`的ASCII表示97就可以了。

Hex编码

可以发现采用Raw编码以后，从a-z一共26个字母，如果采用`分支节点（BranchNode）`存储的话需要26个空间，如果再加上0-9一共10个数字和一个value，总共需要37个空间，以太坊的开发者权衡了一下觉得太多了，于是就改良了编码方式，有了Hex编码。

以太坊先定义了一个新单位`nibble`，一个`nibble`表示4个bit，0.5个byte。然后按照如下规则编码；

- 将Raw输入的每个字符（1byte）拆分成2个nibble，前4位和后4位各一个nibble；
- 将每个nibble扩展为1个byte（8个bit）；
- 然后分别将Raw编码后的十六进制结果的每个b进行如下操作
  - b / 16；
- b % 16；

例如

a的ASCII编码为99（十进制），转换十六进制为63
采用Hex编码
[0] = 63 / 16 = 3
[1] = 63 % 16 = 15
编码后的结果 [3, 15]

HP编码（Hex-Prefix Encoding 16进制前缀编码）

前面介绍的Hex编码后的数据是在内存中的，如果要对Hex编码后的数据进行持久化，就会发现一个问题，我们对原数据进行了扩展，本来1个byte的数据被我们变成了2个byte，显然这对于存储来说是不可接受的，于是就又有了HP编码。

HP编码的过程如下；

- 输入key（Hex编码的结果）如果有标识符，则去掉这个标识符。
- key的头部填充一个nibble，填充的规则如下
  - 如果key的nibble长度是偶数则最后一位0

- 如果key的nibble长度是奇数则最后一位1
- 如果key是`扩展节点`则倒数第二位是0
- 如果key是`叶子节点`则倒数第二位是1

例子：nibble长度是奇数的扩展节点填充为0001。

举个例子：

```
"cat"经过HP编码后的结果 [3，15，3，13，4，10，16]
再用HP编码
1. 去掉16，同时表明这个是叶子节点。
2. 叶子节点，nibble数量是奇数个，这两个条件得出需要填充的值为 0010 0000
3. 将HP编码后的结果用二进制表示 [0010，0000，0011，1111，0011，1101，0100，1010]
4. 将HP编码后的结果合并成byte，为[00100000，00111111，00111101，01001010]转换为十进制是
[32，63，61，74]
```

相较与cat的Raw编码，经过上面的Hex编码和HP编码，既可以能在内存中构建出MPT树，又可以尽可能减小存储所占用的空间，不得不说以太坊设计的巧妙。

安全的MPT

在上面介绍三种编码并没有解决一个问题，如果我们的key非常的长，会导致树非常的深，读写性能急剧的下降，如果有人不怀好意设计了一个独特的key甚至是可以起到DDOS攻击的作用，为了避免上面的问题，以太坊对key进行了一个特别的操作。

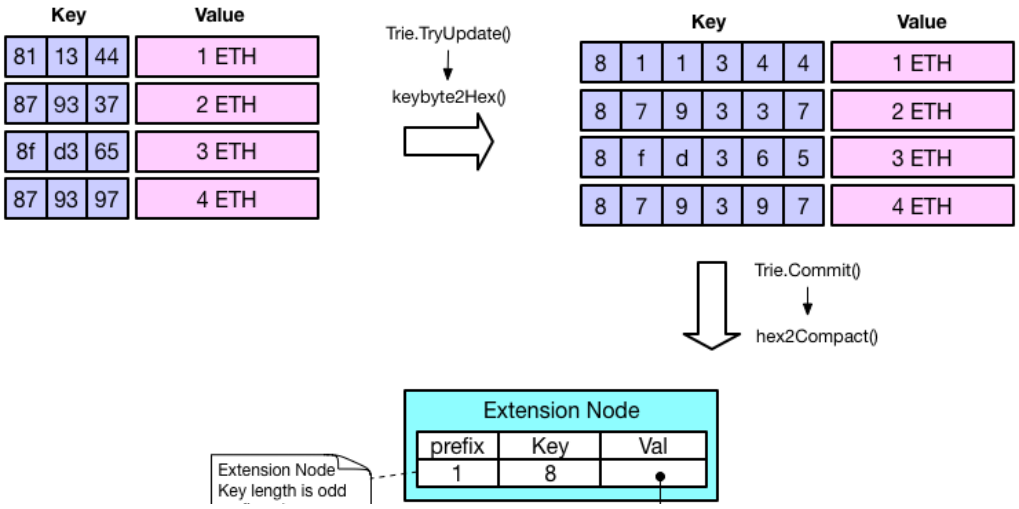
将所有的key都进行了一个`keccak256(key)`的操作，这样就保证了所有key的长度都为一致定长。

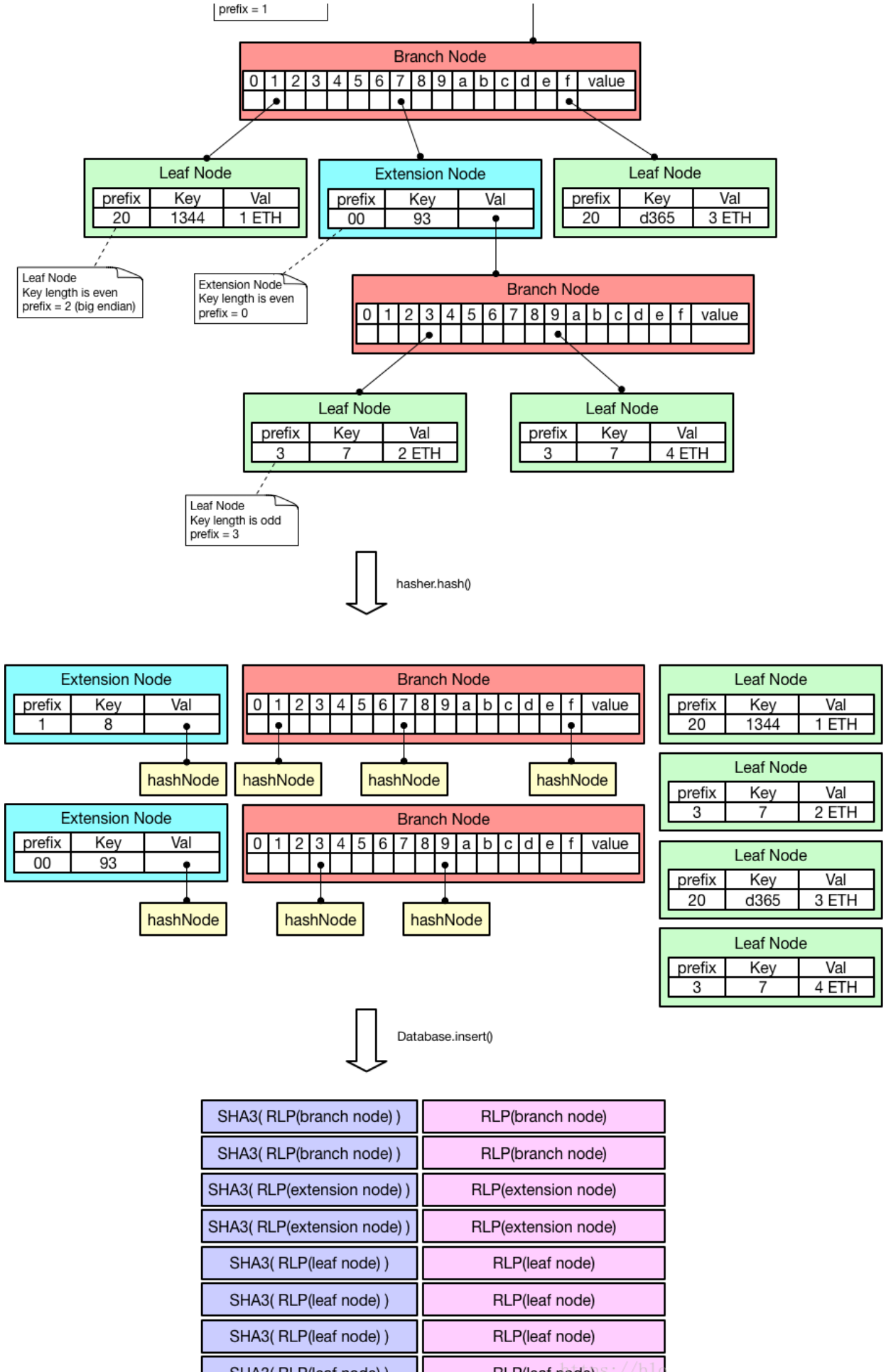
持久化MPT

MPT树节点Key的三种编码形式，但是这三种编码都是对key进行的操作，最终持久化到LevelDB中是k-v的形式，还需要对value进行处理。

在以太坊存储键值对之前会采用RLP编码对键值对进行转码，将键值对编码后作为value，计算编码后数据的哈希（keccak256）作为key，存储在levelDB中。

在具体的实现中，为了避免出现相同的key，以太坊会给key增加一些前缀用作区分，比如合约中的MPT树，会增加合约地址，区块数据会增加表示区块的字符和区块号。 MPT树是以太坊非常非常核心的数据结构，在存储区块，交易，交易回执中都有用到，下图展示了MPT树的全貌，可以再感受一下MPT树的精巧。




prefix = 1
Branch Node
0 1 2 3 4 5 6 7 8 9 a b c d e f value
Leaf Node
prefix Key Val
20 1344 1 ETH
Extension Node
prefix Key Val
00 93
Leaf Node
prefix Key Val
20 d365 3 ETH
Leaf Node
Key length is even
prefix = 2 (big endian)
Extension Node
Key length is even
prefix = 0
Branch Node
0 1 2 3 4 5 6 7 8 9 a b c d e f value
Leaf Node
prefix Key Val
3 7 2 ETH
Leaf Node
prefix Key Val
3 7 4 ETH
Leaf Node
Key length is odd
prefix = 3
hasher.hash()
Extension Node
prefix Key Val
1 8
hashNode
Branch Node
0 1 2 3 4 5 6 7 8 9 a b c d e f value
hashNode
hashNode
hashNode
Leaf Node
prefix Key Val
20 1344 1 ETH
Leaf Node
prefix Key Val
3 7 2 ETH
Leaf Node
prefix Key Val
20 d365 3 ETH
Leaf Node
prefix Key Val
3 7 4 ETH
Extension Node
prefix Key Val
00 93
hashNode
Branch Node
0 1 2 3 4 5 6 7 8 9 a b c d e f value
hashNode
hashNode
Database.insert()
SHA3(RLP(branch node))
RLP(branch node)
SHA3(RLP(branch node))
RLP(branch node)
SHA3(RLP(extension node))
RLP(extension node)
SHA3(RLP(extension node))
RLP(extension node)
SHA3(RLP(leaf node))
RLP(leaf node)
SHA3(RLP(leaf node))
RLP(leaf node)
SHA3(RLP(leaf node))
RLP(leaf node)
SHA3(RLP(leaf node))
RLP(leaf node)

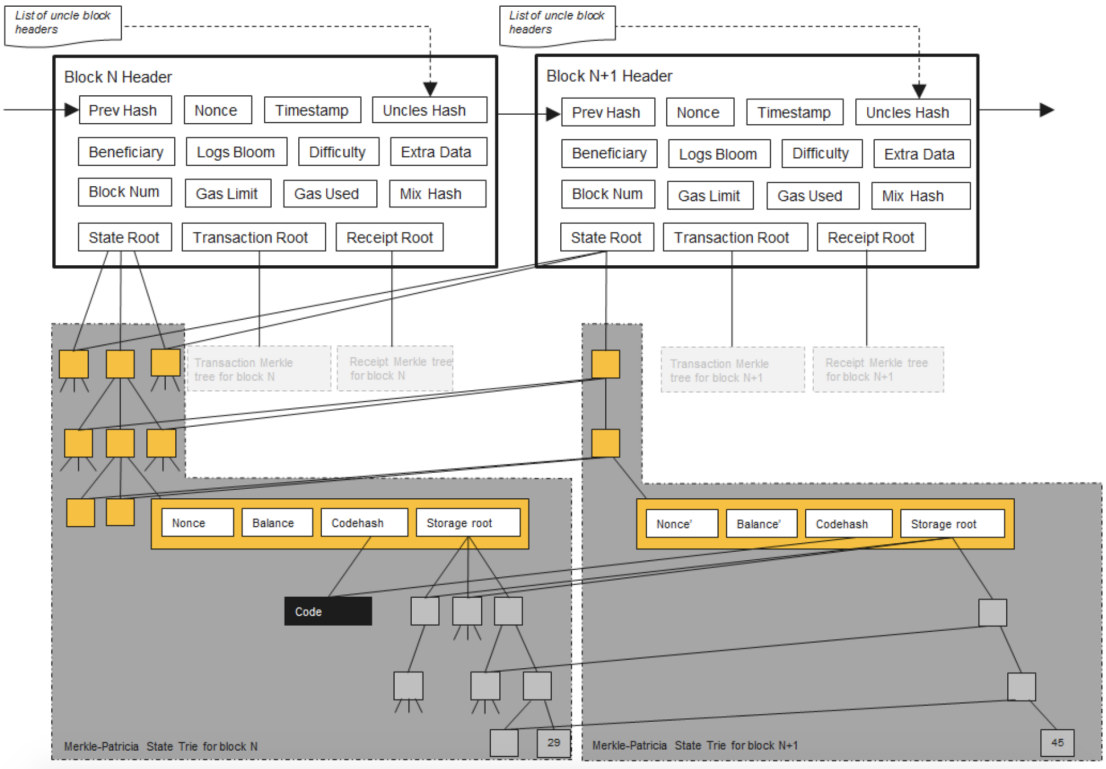
其实随着数据的膨胀，LevelDB本身的读写速度都会变慢，这个是LevelDB实现导致的，这个也是制约MPT树性能的重要因素。

MPT树应用

以太坊区块中有三颗MPT树，分别是状态树，交易树，收据树 ，分别存储了以太坊中的世界状态，本区块的交易，本区块的交易回执，其中交易和交易回执是一一对应的。

状态树

当以太坊每次执行一笔交易的时候，以太坊对应的世界状态都会发生改变。以太坊作为一个公链平台，上面有大量合约所对应的海量状态，如果每次发生状态改变都重建状态树无疑会消耗巨大的计算资源。为了解决这个问题，以太坊提出了增量修改状态树的方法。每次以太坊状态发生改变后并不会去修改原来的MPT树，而是会新建一些分支，如下图所示；



以太坊每次状态发生改变后，只会影响到世界状态MPT树的少量节点，新生成的世界状态树只需要重新计算受到影响的少量节点和与之相连节点的哈希即可。

以太坊的状态非常适用于用MPT树来存储，MPT树的特点与以太坊状态的特点非常契合，当采用MPT树存储以太坊状态以后可以带来如下的优势；

- 当一个账户的余额发生改变后，对应路径的哈希也发生了变化，然后自底而上的更新对应路径上的哈希值，直至Satet Root，这样可以计算最少的哈希次数。
- 以太坊中的全节点维护的是增量的MPT状态树，因为每次一个区块对世界状态的修改都只是很小的一部分，增量修改既有利于区块回滚，又可以节约开销。
- 在以太坊中区块临时分叉很普遍，但是由于以太坊智能合约的复杂性，如果不记录原始状态，很难根据合约代码回滚状态。

收据树

以太坊在智能合约执行时会产生一个交易回执（Receipt）记录了此笔交易的执行结果，交易信息和区块信息。

交易回执 Recipe 信息

Status	执行结果
CumulativeGasUsed	区块累计已用Gas
Logs	交易事件日志
TxHash	交易哈希
ContractAddress	新合约地址
GasUsed	交易消耗的Gas
Bloom	交易事件日志布隆信息
BlockHash	交易所在区块哈希
BlockNumber	交易所在区块高度
TransactionIndex	交易在区块交易集中的索引

当查询轻节点查询通过布隆过滤器找到交易后，为了避免误识，还会再次查询回执来避免误识。

交易树

交易树的作用是提供了交易的默克尔证明，证明某个交易被打包到某个区块里，轻节点不用存储区块体仅根据提供的默克尔证明就可以快速判断交易是否已经被打包。

三颗树的差异

交易树和收据树只依赖当前的区块，而状态树是把链上所有状态都包含进去，交易树和收据树是独立的，状态树会共享树的节点。

为什么状态树要包含所有链上所有的状态呢?

举个例子，当一笔转账操作的发起时，需要判断发起账户是否有足够的ETH来完成这笔转账，这个时候要通过查找状态树查看对应账户的状态，但是如果为了节约空间，只保存了当前区块账户的状态，就需要逐块查找，非常影响性能，甚至这个转账交易的发起者都不存