Report On Merkle Patricia Tree

Merkle Patricia Tree 能够存储任意长度的 key-value 键值对数据，符合以太坊的 state 模型，提供了一种快速计算所维护数据集哈希标识的机制，提供了快速状态回滚的机制。

目录

一，MPT 树的节点;

1,空节点(NULL) - represented as the empty string

简单的表示空，在代码中是一个空串。

2,叶子节点(leaf) - a 2-item node [encodedPath, value]

表示为 [key,value]的一个键值对，其中 key 是 key 的一种特殊十六进制编码(MP 编码)， value 是 value 的 RLP 编码。

3,分支节点(branch) - a 17-item node [v0 … v15, vt]

因为 MPT 树中的 key 被编码成一种特殊的 16 进制的表示，再加上最后的 value，所以分支节点是一个 长度为 17 的 list ** ** ， 前 16 个元素对应着 key 中的 16 个可能的十六进制字符 ， 如果有一个[key,value]对在这个分支节点终止，最后一个元素代表一个值 ，即分支节点既可以搜索路径的终止也可以是路径的中间节点。

分支节点的定义如下：

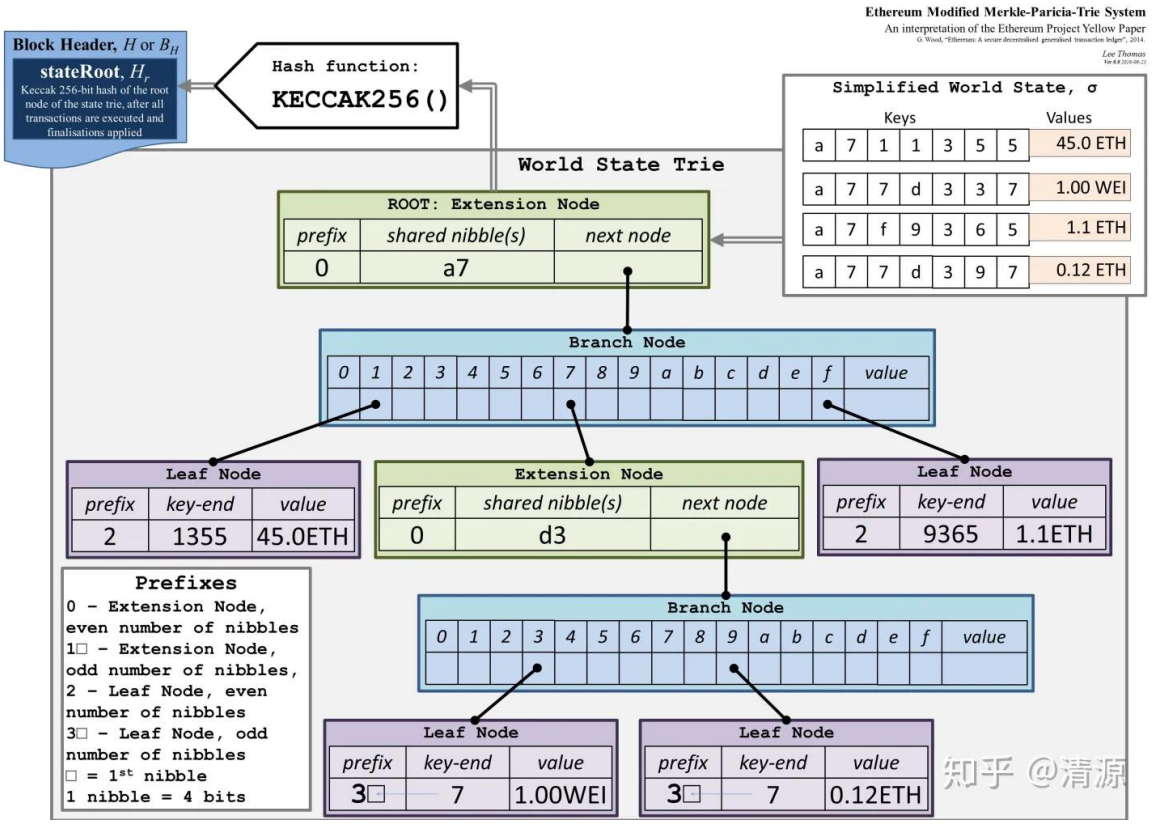
```
type fullNode struct {
        Children [17]node // Actual trie node data to encode/decode (needs custom encoder)
        flags    nodeFlag
}
// nodeFlag contains caching-related metadata about a node.
type nodeFlag struct {
    hash  hashNode // cached hash of the node (may be nil)
    gen   uint16   // cache generation counter
    dirty bool     // whether the node has changes that must be written to the database
}
```

4,扩展节点(extension) - a 2-item node [encodedPath, key]

也是[key，value]的一个键值对 ，但是这里的 value 是其他节点的 hash 值 ，这个 hash 可以被用来查询数据库中的节点。也就是说通过 hash 链接到其他节点。

因此，有两种[key,value]节点(叶节点和扩展节点).

二，MPT 树的简单结构



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。

三，对 key 值的编码

1，Raw 编码（原生字符）

原生的 key 值，不做任何改变。这种编码方式的 key，是 MPT 对外提供接口的默认编码方式。

例如一条 key 为“cat”，value 为“dog”的数据项，其 key 的 Raw 编码就是[‘c’，‘a’，‘t’]，换成 ASCII 表示方式就是[63, 61, 74]（Hex）

2，Hex 编码（扩展的十六进制编码）

Hex 编码就是把一个 8 位的字节数据用两个十六进制数展示出来，编码时，将 8 位二进制码重新分组成两个 4 位的字节，其中一个字节的低 4 位是原字节的高四位，另一个字节的低 4 位是原数据的低 4 位，高 4 位都补 0，然后输出这两个字节对应十六进制数字作为编码。Hex 编码后的长度是源数据的 2 倍。

3，HP 编码

目的是区分 leaf 和 extension；把奇数路径变成偶数路径，方法是

如果有 terminator（16）那么就去掉 terminator；根据表格给 key

加上 prefix。

```
node type    path length    |    prefix    hexchar
-----------------------------------------------
extension    even           |    0000      0x0
extension    odd            |    0001      0x1
leaf         even           |    0010      0x2
leaf         odd            |    0011      0x3
```

编码转换：

四，安全的 MPT：

以上介绍的 MPT 树，可以用来存储内容为任何长度的 key-value 数据项。倘若数据项的 key 长度没有限制时，当树中维护的数据量较大时，仍然会造成整棵树的深度变得越来越深，会造成以下影响：

1，查询一个节点可能会需要许多次 IO 读取，效率低下；

2，系统易遭受 Dos 攻击，攻击者可以通过在合约中存储特定的数据，“构造”一棵拥有一条很长路径的树，然后不断地调用 SLOAD 指令读取该树节点的内容，造成系统执行效率极度下降；

所有的 key 其实是一种明文的形式进行存储；

为了解决以上问题，在以太坊中对 MPT 再进行了一次封装，对数据项的 key 进行了一次哈希计算，因此最终作为参数传入到 MPT 接口的数据项其实是(sha3(key), value)

优势：

传入 MPT 接口的 key 是固定长度的（32 字节），可以避免出现树中出现长度很长的路径；

劣势：

1，每次树操作需要增加一次哈希计算；

2，需要在数据库中存储额外的 sha3(key)与 key 之间的对应关系。

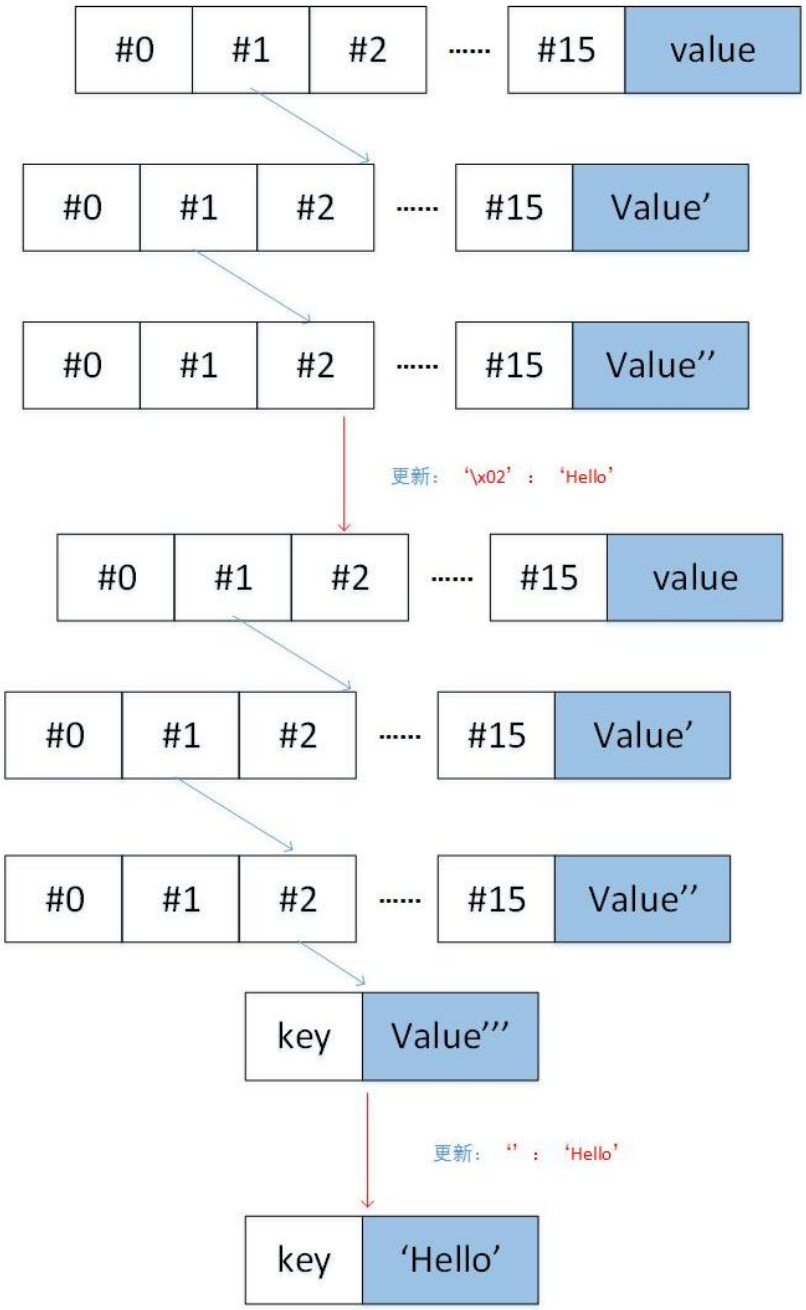
五，MPT 树的操作

1，更新

函数_update_and_delete_storage(self, node, key, value)

.i)如果 node 是空节点，直接返[pack_nibbles(with_terminator(key)), value]，即对 key 加上终止符，然后进行 HP 编码。

.ii)如果 node 是分支节点，如果 key 为空，则说明更新的是分支节点的 value，直接将 node[-1]设置成 value 就行了。如果 key 不为空，则递归更新以 key[0]位置为根的子树，即沿着 key 往下找，即调用 _update_and_delete_storage(self._decode_to_node(node[key[0]]), key[1:], value)。

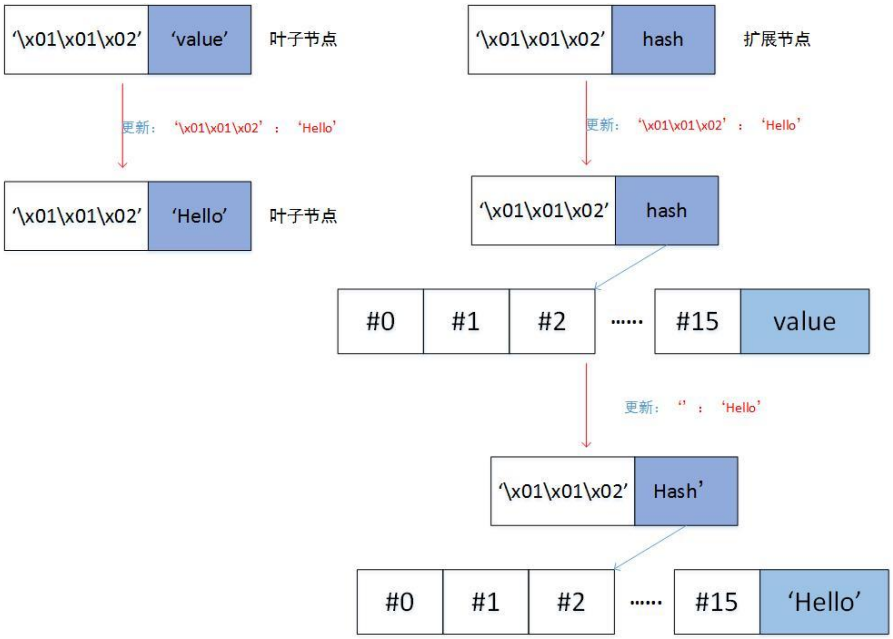


.iii) 如果 node 是 kv 节点（叶子节点或者扩展节点），调用 _update_kv_node(self, node, key, value)，见步骤 iv

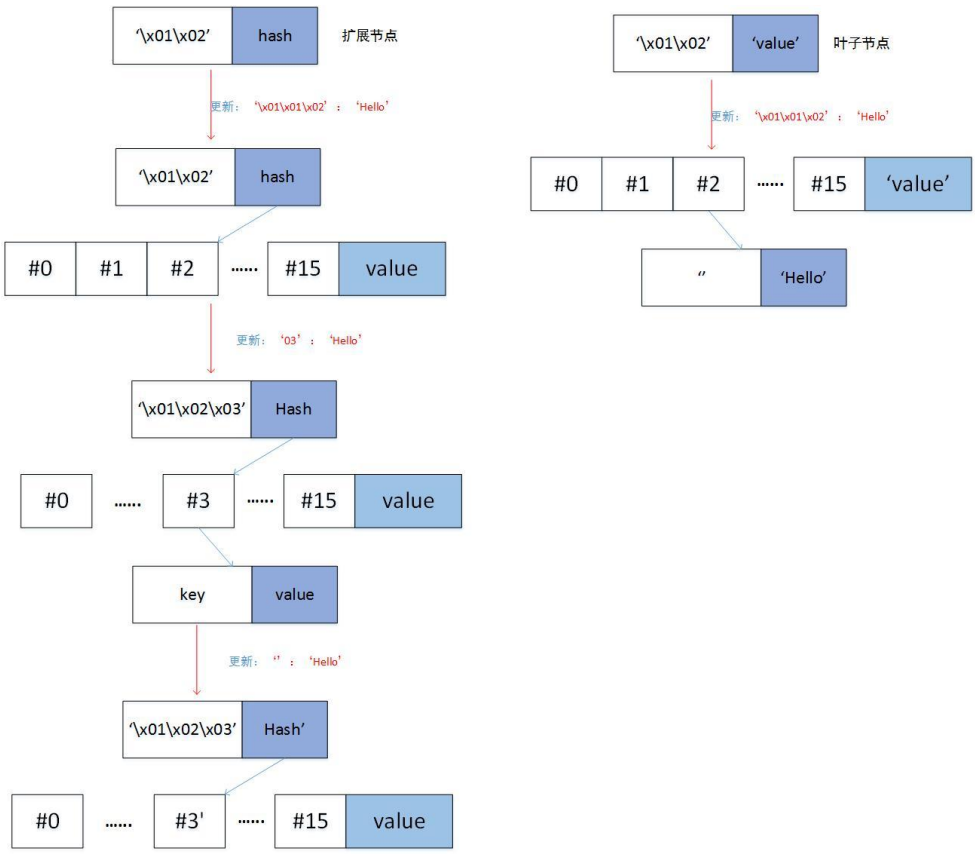
.iv) curr_key 是 node 的 key，找到 curr_key 和 key 的最长公共前缀，长度为 prefix_length。Key 剩余的部分为 remain_key，curr_key

剩余的部分为 remain_curr_key。

a) 如果 remain_key==[]== remain_curr_key,即 key 和 curr_key 相等，那么如果 node 是叶子节点，直接返回[node[0], value]。如果 node 是扩展节点，那么递归更新 node 所链接的子节点，_update_and_delete_storage(self._decode_to_node(node[1]), remain_key,value)

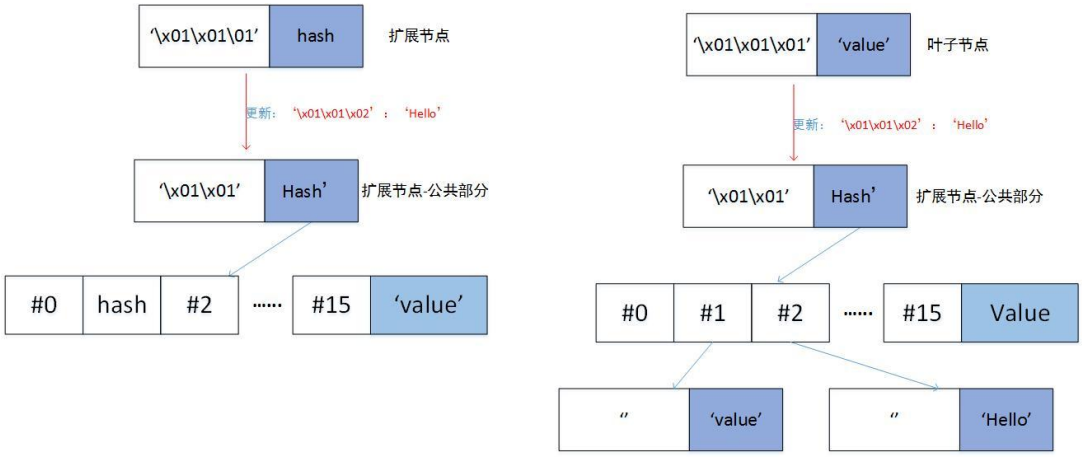


b) 如果 remain_curr_key == []，即 curr_key 是 key 的一部分。如果 node 是扩展节点，递归更新 node 所链接的子节点，即调用 _update_and_delete_storage(self._decode_to_node(node[1]), remain_key, value)；如果 node 是叶子节点，那么创建一个分支节点，分支节点的 value 是当前 node 的 value，分支节点的 remain_key[0]位置指向一个叶子节点，这个叶子节点是 [pack_nibbles(with_terminator(remain_key[1:])), value]



c) 否则，创建一个分支节点。如果 curr_key 只剩下了一个字符，并且 node 是扩展节点，那么这个分支节点的 remain_curr_key[0]的分支是 node[1]，即存储 node 的 value。否则，这个分支节点的 remain_curr_key[0]的分支指向一个新的节点，这个新的节点的 key 是 remain_curr_key[1:]的 HP 编码，value 是 node[1]。如果 remain_key 为空，那么新的分支节点的 value 是要参数中的 value，否则，新的分支节点的 remain_key[0]的分支指向一个新的节点，这个新的节点是[pack_nibbles(with_terminator(remain_key[1:])), value]

d) 如果 key 和 curr_key 有公共部分，为公共部分创建一个扩展节点，此扩展节点的 value 链接到上面步骤创建的新节点，返回这个扩展节点；否则直接返回上面步骤创建的新节点



.v 删除老的 node 返回新的 node

2，删除

删除的过程和更新的过程类似，函数名：

_delete_and_delete_storage(self, key)

.i) 如果 node 为空节点，直接返回空节点

.ii) 如果 node 为分支节点。如果 key 为空，表示删除分支节点的值，直接令 node[-1]=‘’，返回 node 的正规化的结果。如果 key 不为空，递归查找 node 的子节点，然后删除对应的 value，self._delete_and_delete_storage(self._decode_to_node(node[key[0]]), key[1:])。返回新节点

.iii) 如果 node 为 kv 节点，curr_key 是当前 node 的 key。

a) 如果 key 不是以 curr_key 开头，说明 key 不在 node 为根的

子树内，直接返回 node。

b) 否则，如果 node 是叶节点，返回 BLANK_NODE if key == curr_key else node。

c)如果 node 是扩展节点，递归删除 node 的子节点，即 _delete_and_delete_storage(self._decode_to_node(node[1]), key[len(curr_key):])。如果新的子节点和 node[-1]相等直接返回 node。否则，如果新的子节点是 kv 节点，将 curr_key 与新子节点的可以串联当做 key，新子节点的 value 当做 vlaue，返回。如果新子节点是 branch 节点，node 的 value 指向这个新子节点，返回。

3，查找

查找操作更简单，是一个递归查找的过程函数名为：_get(self, node, key)

i. 如果 node 是空节点，返回空节点

ii. 如果node是分支节点,如果key为空,返回分支节点的value;否则递归查找 node 的子节点，即调用 _get(self._decode_to_node(node[key[0]]), key[1:])

iii. 如果 node 是叶子节点，返回 node[1] if key == curr_key else ‘’

iv. 如果 node 是扩展节点，如果 key 以 curr_key 开头，递归查找 node 的子节点，即调用_get(self._decode_to_node(node[1]), key[len(curr_key):])；否则，说明 key 不在以 node 为根的子树里，返回空

六，参考文献

https://easythereentropy.wordpress.com/2014/06/04/understanding-the-ethereum-trie/

https://github.com/ethereum/wiki/wiki/Patricia-Tree

https://github.com/ebuchman/understanding_ethereum_trie

https://github.com/ethereum/pyethereum

https://blog.csdn.net/shangsongwww/article/details/90300598