Merkle Patricia Tree

以太坊区块的头部包括一个区块头，一个交易的列表和一个uncle区块的列表。在区块头部包括了交易的hash树根，用来校验交易的列表。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。

基本数据结构：

1，空节点：列表是空的

2，标准叶子节点

[key,value]的形式 key是找到这个value的key-end

3，扩展节点

[key，value]的列表，但是这里的value是其他节点的hash

4，分支节点

长度为17的list，前16位是用于查询路径的元素。

判断依据：

```
        if node == BLANK_NODE:
            return NODE_TYPE_BLANK

        if len(node) == 2:
            nibbles = unpack_to_nibbles(node[0])
            has_terminator = (nibbles and nibbles[-1]
==NIBBLE_TERMINATOR)
            return NODE_TYPE_LEAF if has_terminator\
                else NODE_TYPE_EXTENSION
        if len(node) == 17:
            return NODE_TYPE_BRANCH
```

因为字母表是16进制的，所以每个节点可能有16个孩子。因为有两种[key,value]节点(叶节点和扩展节点)，引进一种特殊的终止符标识来标识key所对应的是值是真实的值，还是其他节点的hash。如果终止符标记被打开，那么key对应的是叶节点，对应的值是真实的value。如果终止符标记被关闭，那么值就是用于在数据块中查询对应的节点的hash。无论key奇数长度还是偶数长度，HP多可以对其进行编码。最后我们注意到一个单独的hex字符或者4bit二进制数字，即一个nibble。

HP编码：一个nibble被加到key前，对终止符的状态和奇偶性进行编码。最低位表示奇偶性，第二低位编码终止符状态。如果key是偶数长度，那么加上另外一个nubble，值为0来保持整体的偶特性。

```
    if nibbles[-1:] == [NIBBLE_TERMINATOR]:
        flags = 2
        nibbles = nibbles[:-1]
    else:
        flags = 0

    oddlen = len(nibbles) % 2
    flags |= oddlen   # set lowest bit if odd number of nibbles
    if oddlen:
        nibbles = [flags] + nibbles
    else:
        nibbles = [flags, 0] + nibbles
    o = ''
    for i in range(0, len(nibbles), 2):
        o += chr(16 * nibbles[i] + nibbles[i + 1])
    return o
```

操作

_update_and_delete_storage函数

```
old_node = node[:]
new_node = self._update(node, key, value)
```

更新节点调用update函数

若该节点为空，对key加上终止符，然后进行HP编码

```
if node_type == NODE_TYPE_BLANK:
    if PRINT: print ('blank')
    return [pack_nibbles(with_terminator(key)), value]
```

若为分支节点 key为空就把node的list的最后一位设置为为更新的值 若key不为空就递归更新key[0]为根的子树，

```
lif node_type == NODE_TYPE_BRANCH:
            if PRINT: print ('branch')
            if not key:
                if PRINT: print ('\tdone', node)
                node[-1] = value
                if PRINT: print ('\t', node)

            else:
                if PRINT: print ('recursive branch')
                if PRINT: print ('\t', node, key, value)
                new_node = self._update_and_delete_storage(
                    self._decode_to_node(node[key[0]]),
                    key[1:], value)
                if PRINT: print ('\t', new_node)
                node[key[0]] = (self._encode_node(new_node))
                if PRINT: print ('\t', node)
            return node
```

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)

```
elif is_key_value_type(node_type):
    if PRINT: print 'kv'
    return self._update_kv_node(node, key, value)
```

找到最长前缀：

```
for i in range(min(len(curr_key), len(key))):
    if key[i] != curr_key[i]:
        break
    prefix_length = i + 1

remain_key = key[prefix_length:]
remain_curr_key = curr_key[prefix_length:]
```

如果 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，即调用update_and_delete_storage(self._decode_to_node(node[1]), remain_key, value)

```
        if remain_key == [] == remain_curr_key:
            if PRINT: print 'keys were same', node[0], key
            if not is_inner:
                if PRINT: print 'not an extension node'
                return [node[0], value]
            if PRINT: print 'yes an extension node!'
            new_node = self._update_and_delete_storage(
                self._decode_to_node(node[1]), remain_key, value)
```

若 remain_curr_key == [] 说明curr_key是key的一部分*

```
elif remain_curr_key == []:
    if is_inner:
        new_node =
self._update_and_delete_storage(self._decode_to_node(node[1]),
remain_key, value)
```

如果node是扩展节点，递归更新node所链接的子节点

```
   else:
        new_node = [BLANK_NODE] * 17
        new_node[-1] = node[1]

 new_node[remain_key[0]]=self._encode_node([pack_nibbles(with_terminato
r(remain_key[1:])),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]

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]

```
if len(remain_curr_key) == 1 and is_inner:

    new_node[remain_curr_key[0]] = node[1]
else:

    new_node[remain_curr_key[0]] =
self._encode_node([pack_nibbles(adapt_terminator(remain_curr_key[1:],
not is_inner)),node[1]])

if remain_key == []:
    new_node[-1] = value
else:
    new_node[remain_key[0]] =
self._encode_node([pack_nibbles(with_terminator(remain_key[1:])),
value])
```

如果前缀长度不为0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

```
if prefix_length:
    # create node for key prefix
    if PRINT: print 'prefix length', prefix_length
    new_node= [pack_nibbles(curr_key[:prefix_length]),
            self._encode_node(new_node)]
    if PRINT: print 'new node type', self._get_node_type(new_node)
    return new_node
 else:
            return new_node
```

删除delete_node_storage

1. _delete_branch_node

node为分支节点。如果key为空，表示删除分支节点的值，直接另node[-1]=BLANK_NODE, 返回node的__normalize__branch__node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，即调用self.delete__and__delete__storage(self.__decode__to__node(node[key[0]]), key[1:])。返回新节点

```
if not key:
    node[-1] = BLANK_NODE
    return self._normalize_branch_node(node)
encoded_new_sub_node=self._encode_node(self._delete_and_delete_storage
(self._decode_to_node(node[key[0]]), key[1:]))
if encoded_new_sub_node == node[key[0]]:
    return node
node[key[0]] = encoded_new_sub_node
if encoded_new_sub_node == BLANK_NODE:
    return self._normalize_branch_node(node)
return node
```

2._delete_kv_node

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。

```
if not starts_with(key, curr_key):
    # key not found
    return node
```

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

```
if node_type == NODE_TYPE_LEAF:
    return 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，即调用 _delete_and_delete_storage(self._decode_to_node(node[1]), key[len(curr_key):])。

```
new_sub_node = self._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。

```
if self._encode_node(new_sub_node) == node[1]:
    return 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。

```
 if is_key_value_type(new_sub_node_type):

            new_key = curr_key + unpack_to_nibbles(new_sub_node[0])
            return [pack_nibbles(new_key), new_sub_node[1]]
```

如果新子节点是branch节点，node的value指向这个新子节点，返回。

```
if new_sub_node_type == NODE_TYPE_BRANCH:
    return [pack_nibbles(curr_key), self._encode_node(new_sub_node)]
```

3.查找_get():

如果node是空节点，返回空节点

```
if node_type == NODE_TYPE_BLANK:
    return BLANK_NODE
```

如果node是branch节点:

如果key不为空 直接返回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:])

```
if node_type == NODE_TYPE_BRANCH:
    # already reach the expected node
    if not key:
        return node[-1]
    sub_node = self._decode_to_node(node[key[0]])
    return self._get(sub_node, key[1:])
```

如果node是叶子节点， 如果node[1]不为空就返回 curr_key 否则返回 空节点

```
if node_type == NODE_TYPE_LEAF:
    return node[1] if key == curr_key else BLANK_NODE
```

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，返回空节点

```
if node_type == NODE_TYPE_EXTENSION:
    # traverse child nodes
    if starts_with(key, curr_key):
        sub_node = self._decode_to_node(node[1])
        return self._get(sub_node, key[len(curr_key):])
    else:
        return BLANK_NODE
```

最终的数据还是以键值对的形式存储在LevelDB中的，MPT树相当于提供了一个缓存。

参考:

https://www.cnblogs.com/fengzhiwu/p/5584809.html

https://github.com/ebuchman/understanding_ethereum_trie

https://zhuanlan.zhihu.com/p/85657095