**Merkle** **Patricia Trie**

**1．概述**：

**1.1** **MTP：**

在以太坊中，一种经过改良的默克尔树非常关键，是以太坊数据安全与效率的保障，此树在以太坊中称之为 MPT（默克尔压缩前缀树）。 MPT 全称是 Merkle Patricia Trie 也叫 Merkle Patricia Tree，是一种经过改良的、融合了默克尔树和前缀树两种树结构优点的数据结构，是以太坊中用来组织管理账户数据、生成交易集合哈希的重要数据结构。

MPT树有以下几个作用：

1. 存储任意长度的key-value键值对数据；
2. 提供了一种快速计算所维护数据集哈希标识的机制；
3. 提供了快速状态回滚的机制；
4. 提供了一种称为默克尔证明的证明方法，进行轻节点的扩展，实现简单支付验证；

**1.2 Patricia Trie：**

用于保存关联数组，其键（key）的内容通常为字符串。前缀树节点在树中的位置是由其键的内容所决定的，即前缀树的key值被编码在根节点到该节点的路径中。

相比于哈希表，使用前缀树来进行查询拥有共同前缀key的数据时十分高效，例如在字典中查找前缀为pre的单词，对于哈希表来说，需要遍历整个表，时间效率为O(n)；然而对于前缀树来说，只需要在树中找到前缀为pre的节点，且遍历以这个节点为根节点的子树即可。且相比于哈希表，在前缀树不会存在哈希冲突的问题。

但其直接查找效率低下，且可能会造成空间浪费。

**1.3 Merkle Trie**

在比特币网络中用这种数据结构来进行数据正确性的验证， 在比特币网络中，merkle树被用来归纳一个区块中的所有交易，同时生成整个交易集合的数字指纹。此外，由于merkle树的存在，使得在比特币这种公链的场景下，扩展一种“轻节点”实现简单支付验证变成可能。

merkle树的特点之一就是当树节点内容发生变化时，能够在前一次哈希计算的基础上，仅仅将被修改的树节点进行哈希重计算，便能得到一个新的根哈希用来代表整棵树的状态；

采用默克尔树，可以在公链环境下扩展一种“轻节点”。通过轻节点，可以实现在非信任的公链环境中验证某一笔交易是否被收录在区块链账本的功能。这使得像比特币，以太坊这样的区块链能够运行在个人PC，智能手机等拥有小存储容量的终端上。

merkle树的缺点为其存储空间开销大。

**2**. **MTP结构**

**2.1节点分类**

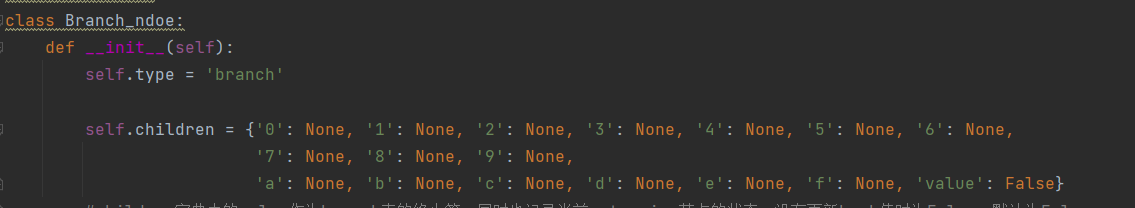
空节点：用来表示空串

分支节点：

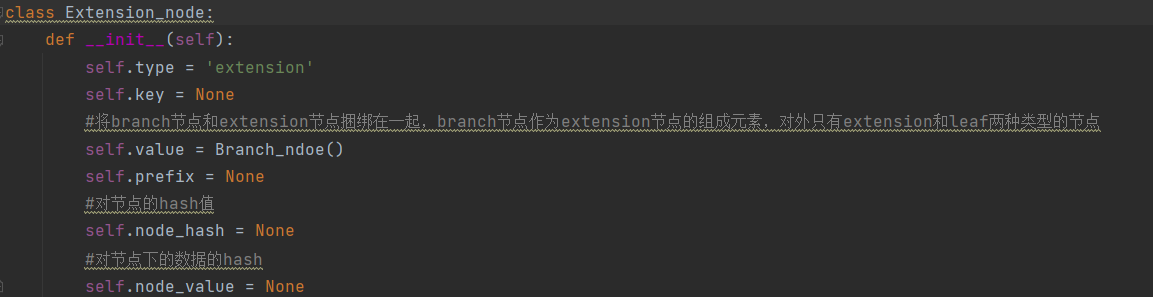
分支节点用来表示MPT树中所有拥有超过1个孩子节点以上的非叶子节点。

与前缀树相同，MPT同样是把key-value数据项的key编码在树的路径中，但是key的每一个字节值的范围太大（[0-127]），因此在以太坊中，在进行树操作之前，首先会进行一个key编码的转换（下节会详述），将一个字节的高低四位内容分拆成两个字节存储。通过编码转换，key’的每一位的值范围都在[0, 15]内。因此，一个分支节点的孩子至多只有16个。以太坊通过这种方式，减小了每个分支节点的容量，但是在一定程度上增加了树高。

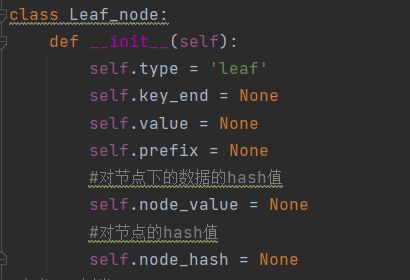
分支节点的孩子列表中，最后一个元素是用来存储自身的内容。



扩展节点：拥有两个元素，编码路径encodedPath和键key。encodedPath，包含了下面不分叉的那部分路径，也就是把路径压缩的那部分；key，是指向下一个节点的hash（hash，也即在底层db中的存储位置）。



叶子节点：拥有两个元素，编码路径encodedPath和值value。对应的叶子节点也有类似的数据结构，他的数据结构是一个编码路径encodedPath，因为是叶子节点，后面就没有分叉路径，它的第二个元素就是自己的value。



**2.2 编码方式**

Raw编码

Raw编码就是原生的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。

Hex编码

为了减少分支节点孩子的个数，需要将key的编码进行转换，将原key的高低四位分拆成两个字节进行存储。这种转换后的key的编码方式，就是Hex编码。

从Raw编码向Hex编码的转换规则：

1)将Raw编码的每个字符，根据高4位低4位拆成两个字节；

2)若该Key对应的节点存储的是真实的数据项内容（即该节点是叶子节点），则在末位添加一个ASCII值为16的字符作为终止标志符；

3)若该key对应的节点存储的是另外一个节点的哈希索引（即该节点是扩展节点），则不加任何字符；

HP (Hex-Prefix)编码

以太坊就提出了一种HP编码对存储在数据库中的叶子／扩展节点的key进行编码区分。在将这两类节点持久化到数据库之前，首先会对该节点的key做编码方式的转换，即从Hex编码转换成HP编码。

HP编码的规则如下：

1)若原key的末尾字节的值为16（即该节点是叶子节点），去掉该字节；

2)在key之前增加一个半字节，其中最低位用来编码原本key长度的奇偶信息，key长度为奇数，则该位为1；低2位中编码一个特殊的终止标记符，若该节点为叶子节点，则该位为1；

3)若原本key的长度为奇数，则在key之前再增加一个值为0x0的半字节；

4)将原本key的内容作压缩，即将两个字符以高4位低4位进行划分，存储在一个字节中（Hex扩展的逆过程）；

**3.相关操作**

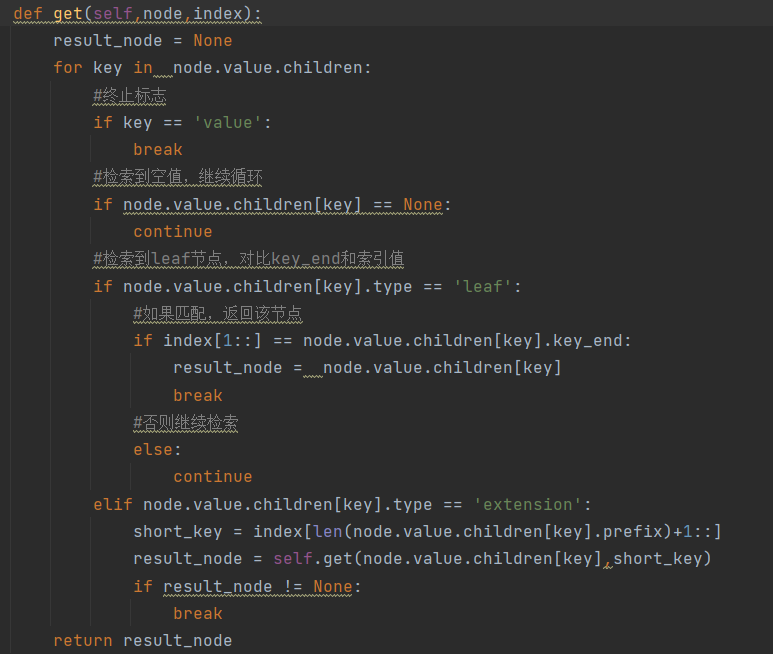
**3.1 Get**

一次Get操作的过程为：

* 将需要查找Key的Raw编码转换成Hex编码，得到的内容称之为搜索路径；
* 从根节点开始搜寻与搜索路径

内容一致的路径；

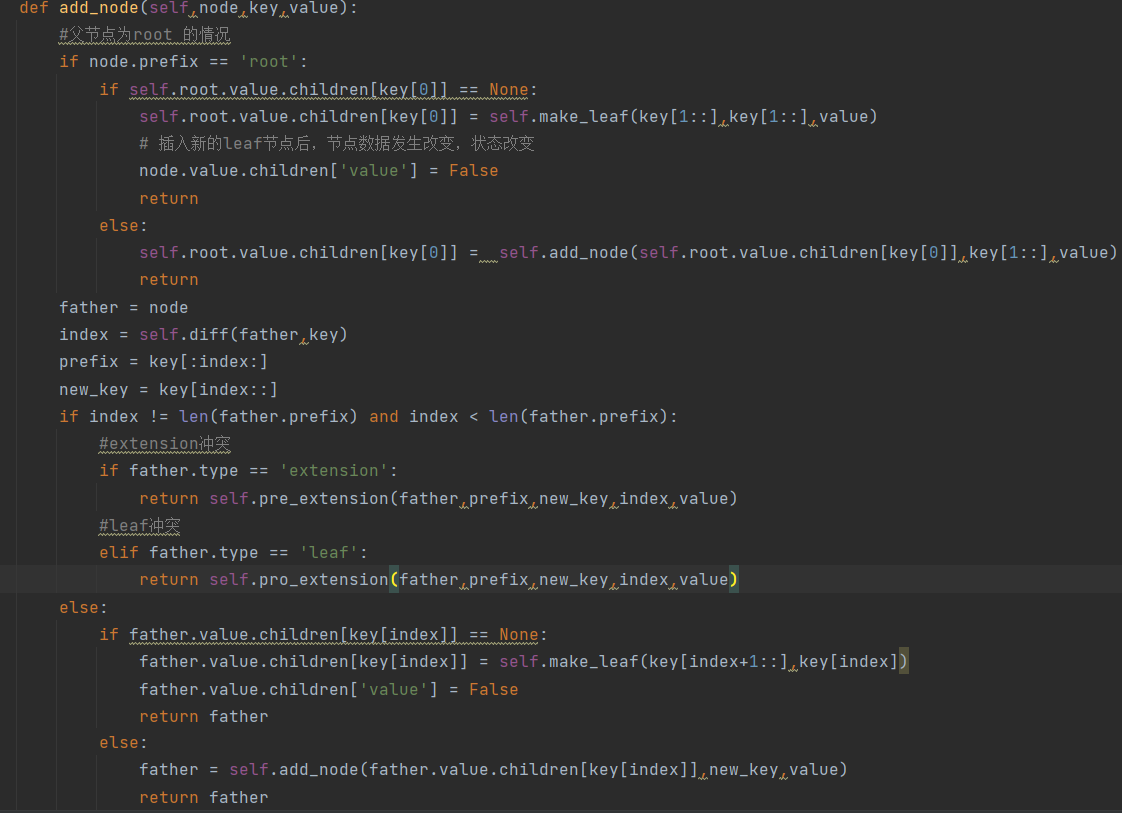
1. 若当前节点为叶子节点，存储的内容是数据项的内容，且搜索路径的内容与叶子节点的key一致，则表示找到该节点；反之则表示该节点在树中不存在。
2. 若当前节点为扩展节点，且存储的内容是哈希索引，则利用哈希索引从数据库中加载该节点，再将搜索路径作为参数，对新解析出来的节点递归地调用查找函数。
3. 若当前节点为扩展节点，存储的内容是另外一个节点的引用，且当前节点的key是搜索路径的前缀，则将搜索路径减去当前节点的key，将剩余的搜索路径作为参数，对其子节点递归地调用查找函数；若当前节点的key不是搜索路径的前缀，表示该节点在树中不存在。
4. 若当前节点为分支节点，若搜索路径为空，则返回分支节点的存储内容；反之利用搜索路径的第一个字节选择分支节点的孩子节点，将剩余的搜索路径作为参数递归地调用查找函数。



**3.2 Insert**

插入操作也是基于查找过程完成的，一个插入过程为：

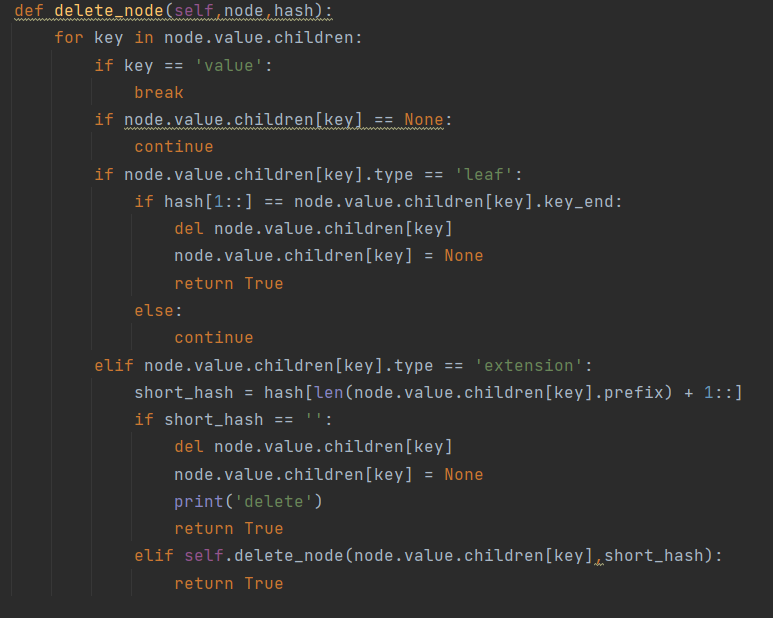
1. 根据3.1中描述的查找步骤，首先找到与新插入节点拥有最长相同路径前缀的节点，记为Node；
2. 若该Node为分支节点：  
   （1）剩余的搜索路径不为空，则将新节点作为一个叶子节点插入到对应的孩子列表中；  
   （2）剩余的搜索路径为空（完全匹配），则将新节点的内容存储在分支节点的第17个孩子节点项中（Value）；
3. 若该节点为叶子／扩展节点：  
   （1）剩余的搜索路径与当前节点的key一致，则把当前节点Val更新即可；  
   （2）剩余的搜索路径与当前节点的key不完全一致，则将叶子／扩展节点的孩子节点替换成分支节点，将新节点与当前节点key的共同前缀作为当前节点的key，将新节点与当前节点的孩子节点作为两个孩子插入到分支节点的孩子列表中，同时当前节点转换成了一个扩展节点（若新节点与当前节点没有共同前缀，则直接用生成的分支节点替换当前节点）；
4. 若插入成功，则将被修改节点的dirty标志置为true，hash标志置空（之前的结果已经不可能用），且将节点的诞生标记更新为现在；



**3.3 Delete**

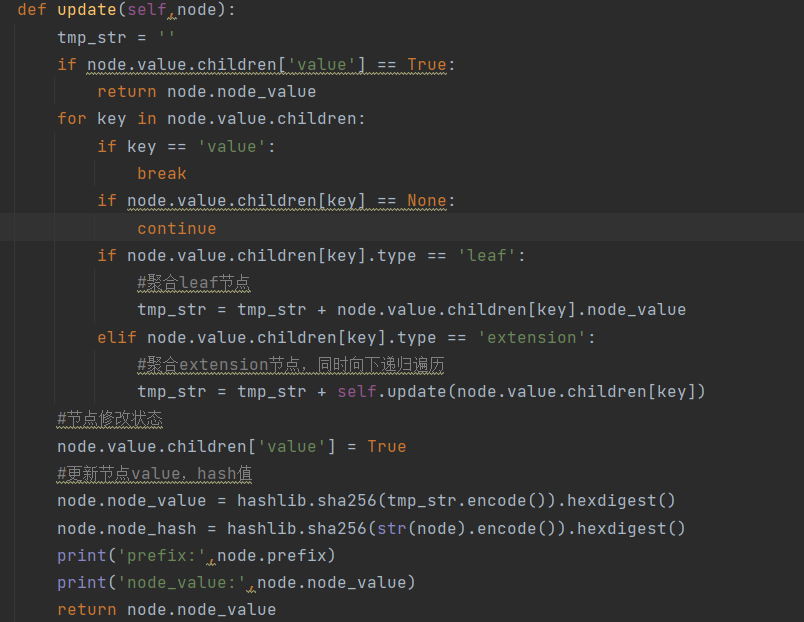
删除操作与插入操作类似，都需要借助查找过程完成，一次删除过程为：

1. 根据3.1中描述的查找步骤，找到与需要插入的节点拥有最长相同路径前缀的节点，记为Node；
2. 若Node为叶子／扩展节点：  
   （1）若剩余的搜索路径与node的Key完全一致，则将整个node删除；  
   （2）若剩余的搜索路径与node的key不匹配，则表示需要删除的节点不存于树中，删除失败；  
   （3）若node的key是剩余搜索路径的前缀，则对该节点的Val做递归的删除调用；
3. 若Node为分支节点：  
   （1） 删除孩子列表中相应下标标志的节点；  
   （2） 删除结束，若Node的孩子个数只剩下一个，那么将分支节点替换成一个叶子／扩展节点；
4. 若删除成功，则将被修改节点的dirty标志置为true，hash标志置空（之前的结果已经不可能用），且将节点的诞生标记更新为现在；



**3.4 Update**

更新操作就是3.2Insert与3.3Delete的结合。当用户调用Update函数时，若value不为空，则隐式地转为调用Insert；若value为空，则隐式地转为调用Delete，故在此不再赘述。



**3.5 Commit**

Commit函数提供将内存中的MPT数据持久化到数据库的功能。

在commit完成后，所有变脏的树节点会重新进行哈希计算，并且将新内容写入数据库；最终新的根节点哈希将被作为MPT的最新状态被返回。

一次MPT树提交是一个递归调用的过程，在介绍MPT提交过程之前，我们首先介绍单个节点是如何进行哈希计算和存储的。

**单节点**

1. 首先是对该节点进行脏位的判断，若当前节点未被修改，则直接返回该节点的哈希值，调用结束（此外，若当前节点既未被修改，同时存在于内存的时间又”过长“，则将以该节点为根节点的子树从内存中驱除）；
2. 该节点为脏节点，对该节点进行哈希重计算。首先是对当前节点的孩子节点进行哈希计算，对孩子节点的哈希计算是利用递归地对节点进行处理完成。这一步骤的目的是将孩子节点的信息各自转换成一个哈希值进行表示；。
3. 对当前节点进行哈希计算。哈希计算利用sha256哈希算法对当前节点的RLP编码进行哈希计算；

对于分支节点来说，该节点的RLP编码就是对其孩子列表的内容进行编码，且在第二步中，所有的孩子节点所有已经被转换成了一个哈希值；  
对于叶子／扩展节点来说，该节点的RLP编码就是对其Key，Value字段进行编码。同样在第二步中，若Value指代的是另外一个节点的引用，则已经被转换成了一个哈希值（在第二步中，Key已经被转换成了HP编码）；

1. 将当前节点的数据存入数据库，存储的格式为[节点哈希值，节点的RLP编码]。
2. 将自身的dirty标志置为false，并将计算所得的哈希值进行缓存；

**MPT树的提交过程**

在理解单节点的提交过程后，MPT树的提交过程就是以根节点为入口，对根节点进行提交调用即可。

**节点过老的判断依据**

判断一个节点在内存中存在时间是否过长的依据是：

* 该节点未被修改；
* 当前MPT的计数器减去节点的诞生标志超过了固定的上限；
* 每当MPT调用一次Commit函数，MPT的计数器发生自增；

**实现功能**

1. 快速计算所维护数据集哈希标识

这个特点体现在单节点计算的第一步，即在节点哈希计算之前会对该节点的状态进行判断，只有当该节点的内容变脏，才会进行哈希重计算、数据库持久化等操作。如此一来，在某一次事务操作中，对整棵MPT树的部分节点的内容产生了修改，那么一次哈希重计算，仅需对这些被修改的节点、以及从这些节点到根节点路径上的节点进行重计算，便能重新获得整棵树的新哈希。

1. 快速状态[回滚](https://so.csdn.net/so/search?q=%E5%9B%9E%E6%BB%9A&spm=1001.2101.3001.7020)

在公链的环境下，采用POW算法是可能会造成分叉而导致区块链状态进行回滚的。在以太坊中，由于出块时间短，这种分叉的几率很大，区块链状态回滚的现象很频繁。

所谓的状态回滚指的是：  
（1）区块链内容发生了重组织，链头发生切换  
（2）区块链的世界状态（账户信息）需要进行回滚，即对之前的操作进行撤销。

MPT树就提供了一种机制，可以当区块碰撞发生了，零延迟地完成世界状态的回滚。这种优势的代价就是需要浪费存储空间去冗余地存储每个节点的历史状态。

每个节点在数据库中的存储都是值驱动的。当一个节点的内容发生了变化，其哈希相应改变，而MPT将哈希作为数据库中的索引，也就实现了对于每一个值，在数据库中都有一条确定的记录。而MPT是根据节点哈希来关联父子节点的，因此每当一个节点的内容发生变化，最终对于父节点来说，改变的只是一个哈希索引值；父节点的内容也由此改变，产生了一个新的父节点，递归地将这种影响传递到根节点。最终，一次改变对应创建了一条从被改节点到根节点的新路径，而旧节点依然可以根据旧根节点通过旧路径访问得到。

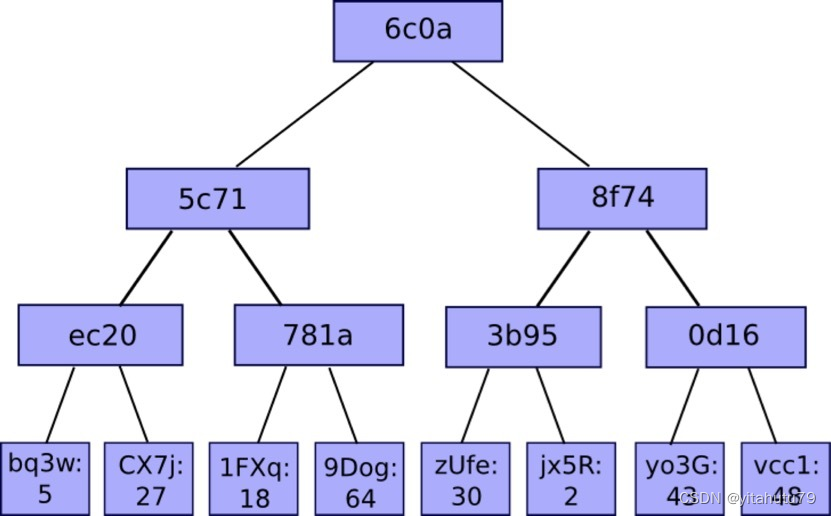
在以太坊中，发生分叉而进行世界状态回滚时，仅需要用旧的MPT根节点作为入口，即可完成“状态回滚”。

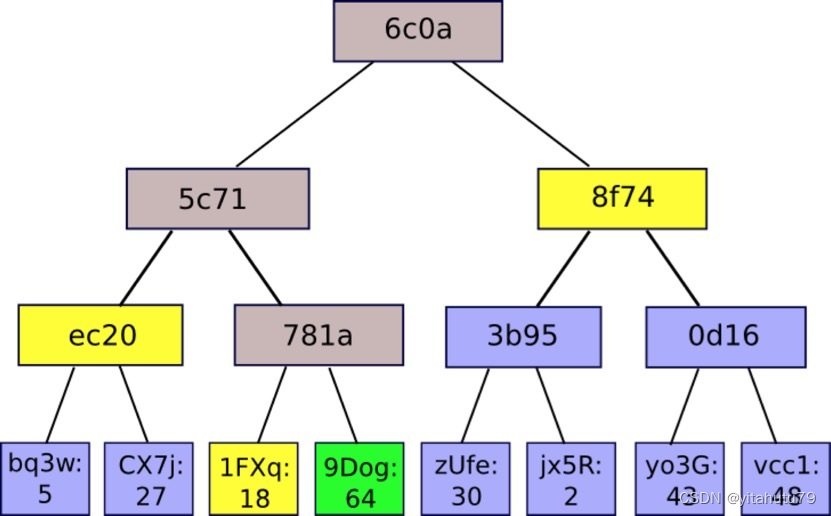
**4.** **默克尔证明**

默克尔证明指一个轻节点向一个全节点发起一次证明请求，询问全节点完整的默克尔树中，是否存在一个指定的节点；全节点向轻节点返回一个默克尔证明路径，由轻节点进行计算，验证存在性。

**4.1 默克尔证明过程**

如有棵如下图所示的merkle树，如果某个轻节点想要验证9Dog:64这个树节点是否存在与默克尔树中，只需要向全节点发送该请求，全节点会返回一个1FXq:18, ec20,  
8f74的一个路径（默克尔路径，如图2黄色框所表示的）。得到路径之后，轻节点利用9Dog:64与1FXq:18求哈希，在与ec20求哈希，最后与8f74求哈希，得到的结果与本地维护的根哈希相比，是否相等。





**4.2默克尔证明安全性**

（1）若全节点返回的是一条恶意的路径？

试图为一个不存在于区块链中的节点伪造一条合法的merkle路径，使得最终的计算结果与区块头中的默克尔根哈希相同。

由于哈希的计算具有不可预测性，使得一个恶意的“全”节点想要为一条不存在的节点伪造一条“伪路径”使得最终计算的根哈希与轻节点所维护的根哈希相同是不可能的。

（2）为什么不直接向全节点请求该节点是否存在于区块链中？

由于在公链的环境中，无法判断请求的全节点是否为恶意节点，因此直接向某一个或者多个全节点请求得到的结果是无法得到保证的。但是轻节点本地维护的区块头信息，是经过工作量证明验证的，也就是经过共识一定正确的，若利用全节点提供的默克尔路径，与待验 证的节点进行哈希计算，若最终结果与本地维护的区块头中根哈希一致，则能够证明该节点一定存在于默克尔树中。

**3.总结**

·MPT提供了一个基于密码学验证的底层数据结构，用来存储键值对（key-value）关系

可以进行数据校验，防止篡改，MPT的目的是用来存储键值对关系，与默克尔树有些不同，因为默克尔树存的是hash，所有的hash两两算一个父hash，一直算到根节点，而MPT不仅要存储hash，还要存储键值对关系。

·MPT是确定性的

 这是指在一颗MPT上，一组键值对是唯一确定的，在没有修改的前提下，每次访问，根据同一个键可以保证找到同样的值，并且有同样的根哈希。

·MPT的插入、查找、删除操作的时间复杂度都是O(log(n))

相对于其它基于复杂比较的树结构（比如红黑树），MPT更容易理解，也更易于编码实现。