**\*Project22: research report on MPT**

**1.概述**

Merkle Patricia Tree（又称为Merkle Patricia Trie）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：

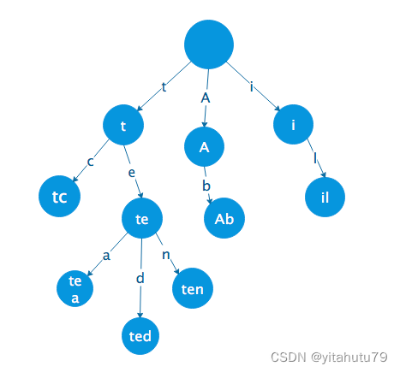
1. 存储任意长度的key-value键值对数据；
2. 提供了一种快速计算所维护数据集哈希标识的机制；
3. 提供了快速状态回滚的机制；
4. 提供了一种称为默克尔证明的证明方法，进行轻节点的扩展，实现简单支付验证；

**2.相关概念**

**前缀树**

前缀树（又称字典树），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。

如下图所示，图中共有6个叶子节点，其key的值分别为（1）tc（2）tea（3）ted（4）ten（5）Ab（6）il。



**默克尔树**

Merkle树是由计算机科学家 Ralph Merkle 在很多年前提出的，并以他本人的名字来命名,由于在比特币网络中用到了这种数据结构来进行数据正确性的验证，在这里简要地介绍一下merkle树的特点及原理。

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。

**特点**

1. 默克尔树大多数是二叉树，也可以多叉树，无论是几叉树，它都具有树结构的所有特点；
2. 默克尔树叶子节点的value是数据项的内容，或者是数据项的哈希值；
3. 非叶子节点的value根据其孩子节点的信息，然后按照Hash算法计算而得出的；

**3.** **结构设计**

**节点分类**

MPT树中，树节点可以分为以下四类：

空节点、分支节点、叶子节点、扩展节点。

空节点用来表示空串。

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。

叶子节点与扩展节点的定义相似，如下所示：

type shortNode struct {

Key []byte

Val node

flags nodeFlag

}

其中关键的字段为：

Key：用来存储属于该节点范围的key；

Val：用来存储该节点的内容；

**key值编码**

在以太坊中，MPT树的key值共有三种不同的编码方式，以满足不同场景的不同需求。

三种编码方式分别为：

1. Raw编码（原生的字符）；
2. Hex编码（扩展的16进制编码）；
3. Hex-Prefix编码（16进制前缀编码）；

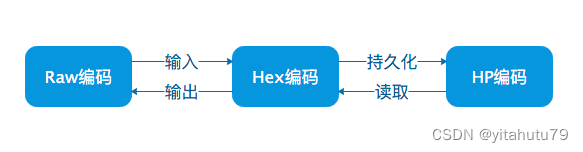
Raw编码就是原生的key值，不做任何改变。这种编码方式的key，是MPT对外提供接口的默认编码方式。

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。

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）

**三种编码的转换关系**



以上三种编码方式的转换关系为：

Raw编码：原生的key编码，是MPT对外提供接口中使用的编码方式，当数据项被插入到树中时，Raw编码被转换成Hex编码；

Hex编码：16进制扩展编码，用于对内存中树节点key进行编码，当树节点被持久化到数据库时，Hex编码被转换成HP编码；

HP编码：16进制前缀编码，用于对数据库中树节点key进行编码，当树节点被加载到内存时，HP编码被转换成Hex编码；

**4. 基本操作**

**Get**

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

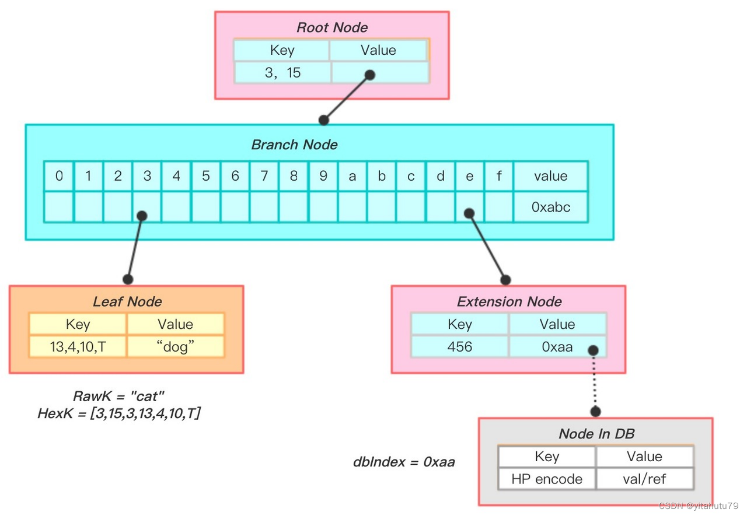
内容一致的路径；

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。

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。

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。

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。



**Insert**

插入操作也是基于查找过程完成的，一个插入过程为：

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；

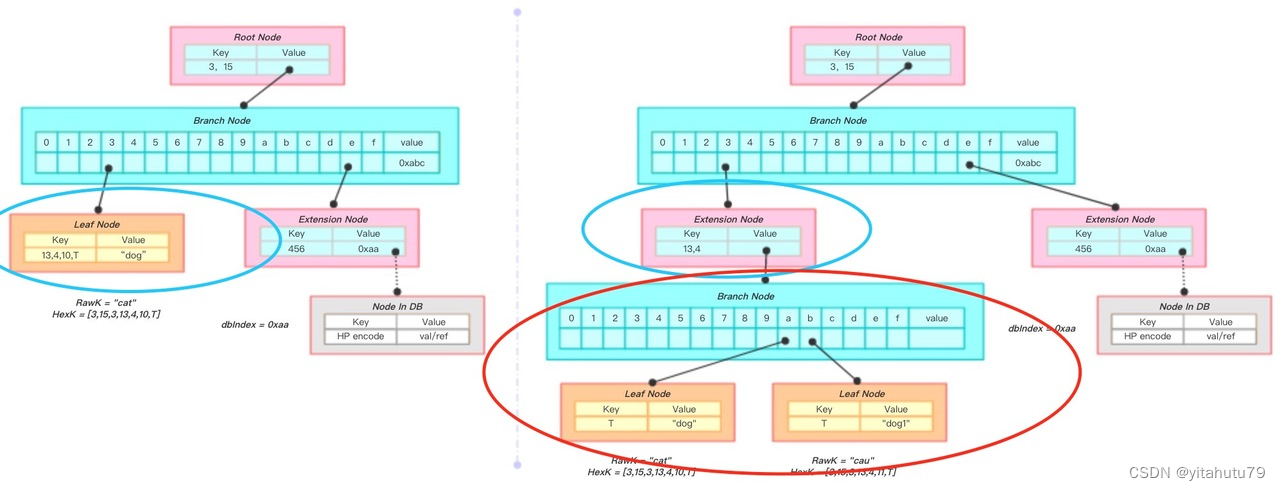
（2）剩余的搜索路径为空（完全匹配），则将新节点的内容存储在分支节点的第17个孩子节点项中（Value）；

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）；

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；



**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：

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；

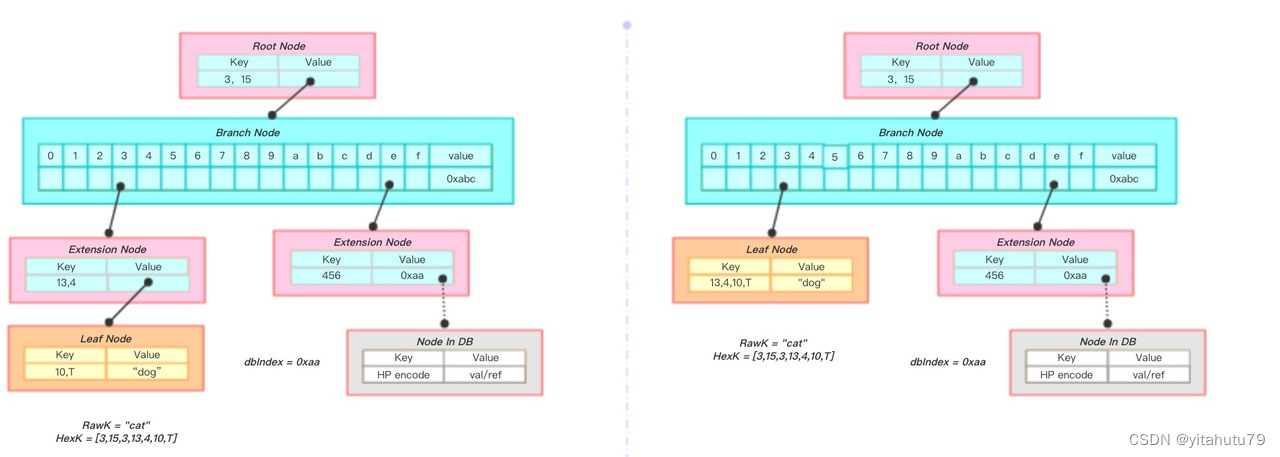
（3）若node的key是剩余搜索路径的前缀，则对该节点的Val做递归的删除调用；

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；

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；



**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。

**Commit**

Commit函数提供将内存中的MPT数据持久化到数据库的功能。在第一章中我们提到的MPT具有快速计算所维护数据集哈希标识以快速状态回滚的能力，也都是在该函数中实现的。

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。