Merkle Patricia Tree 是一种经过改良的、融合了 Merkle tree 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 树中的节点：

1. 空节点(NULL)

   简单的表示空，在代码中是一个空串。

2. 叶子节点(leaf)

   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)

   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。

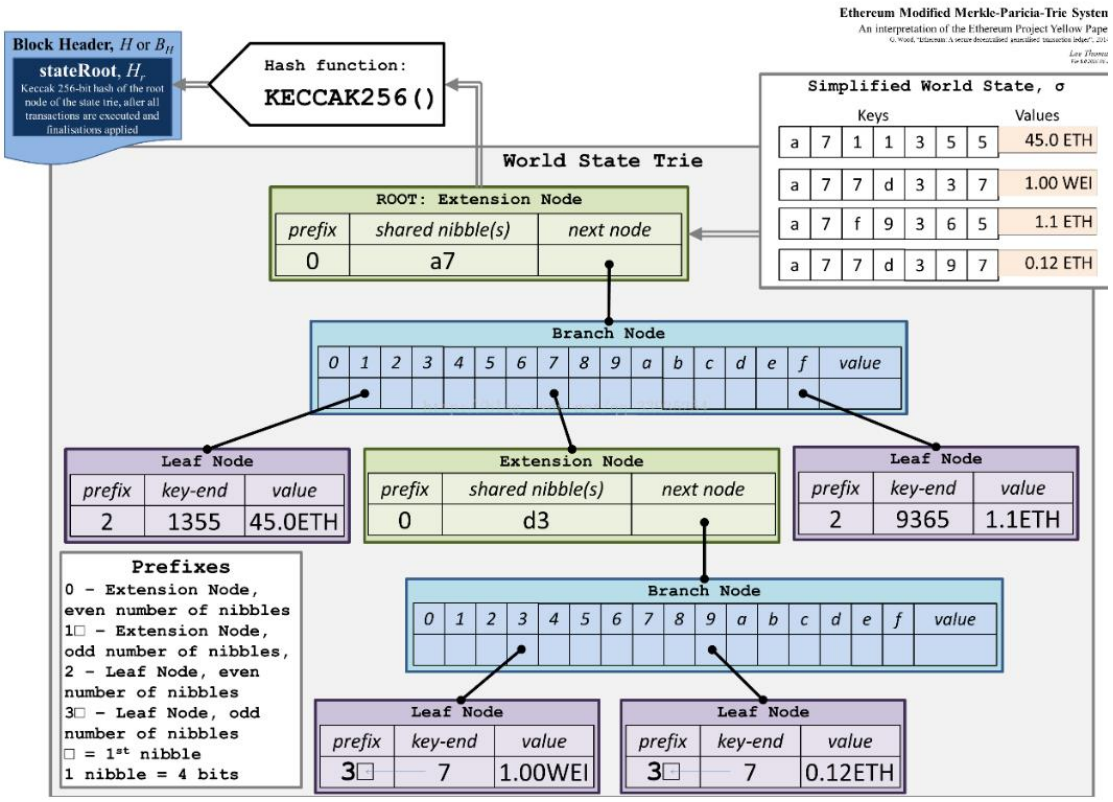
4. 扩展节点(extension)

   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。

在以太坊中 MPT 树的 key 值共有三中不同的编码方式，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）；

MPT 树的特点如下：

1. 叶子节点和分支节点可以保存 value，扩展节点保存 key。
2. 没有公共的 key 就成为 2 个叶子节点；key1=[1,2,3] key2=[2,2,3]。
3. 有公共的 key 需要提取为一个扩展节点；key1=[1,2,3] key2=[1,3,3] => ex-node=[1],下一级分支 node 的 key。
4. 如果公共的 key 也是一个完整的 key，数据保存到下一级的分支节点中；key1=[1,2] key2=[1,2,3] =>ex-node=[1,2],下一级分支 node 的 key; 下一级分支=[3],上一级 key 对应的 value。



同时 MPT 节点有个 flag 字，nodeFlag，记录了一些辅助数据：

1. 节点哈希：若该字段不为空，则当需要进行哈希计算时，可以跳过计算过程而直接使用上次计算的结果（当节点变脏时，该字段被置空）。

2. 诞生标志：当该节点第一次被载入内存中（或被修改时），会被赋予一个计数值作为诞生标志，该标志会被作为节点驱除的依据，清除内存中“太老”的未被修改的节点，防止占用的内存空间过多。

3. 脏标志：当一个节点被修改时，该标志位被置为 1。

flag.hash 会保存该节点采用 merkle tree 类似算法生成的 hash。同时会将 hash 和源数据以<hash, node.rlp.rawdata>方式保存在 leveldb 数据库中。这样后面通过 hash 就可以反推出节点数据。

MPT 主要有以下几种核心的基本操作：

一.Get

一次 Get 操作的过程为：

1. 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。

2. 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。

二. 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 4.1 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：

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；

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：

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；

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；

三. 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 4.1 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：

若剩余的搜索路径与 node 的 Key 完全一致，则将整个 node 删除；

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；

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：

删除孩子列表中相应下标标志的节点；

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；

四. Update

更新操作就是 4.2Insert 与 4.3Delete 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。

五. 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。

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；
3. 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)。

MPT 树主要存在以下优劣：

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：每次树操作需要增加一次哈希计算；需要在数据库中存储额外的 sha3(key)与 key 之间的对应关系；