MPT REPORT

一．MPT是什么

MPT树（Merkle Patricia Tree）是一种基于前缀树和Merkle树的数据结构，是以太坊（Ethereum）区块链中的关键数据结构之一。它被用来存储和验证区块链上的状态（包括账户、合约和存储的数据）。

MPT树的设计灵感来源于Patricia Trie（也称为Radix Trie），它是一种高效的前缀树数据结构，用于快速存储和检索键值对。与传统的哈希表不同，MPT树可以支持按照键的前缀进行范围查询和证明。

MPT提供了一个基于密码学验证的底层数据结构，是Ethereum用来存储区块数据的核心数据结构，用来存储键值对(key-value)关系。MPT是完全确定性的，这是指在一颗MPT上的一组键值对是唯一确定的，相同内容的键可以保证找到同样的值，并且有同样的根哈希(root hash)。MPT的插入、查找、删除操作的事件复杂度都是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。erkle Tree 协议对比特币的长期持续性可以说是至关重要的。

MPT树的核心思想是使用键的哈希值作为叶子节点，将键值对保存在叶子节点上。每个内部节点都包含一个分支和一个扩展，分支存储了一系列键的共同前缀，扩展则指向下一个节点。通过递归地将键拆分为前缀和后缀，MPT树可以有效地组织和管理大量的键值对。

MPT树的另一个重要特性是其使用了Merkle树的概念。Merkle树是一种哈希树，可以对整个数据集进行完整性和一致性验证。MPT树使用Merkle树的哈希值作为每个节点的标识，并通过验证哈希值来确保树的完整性。

在以太坊中，MPT树被广泛应用于存储区块链状态（包括账户、合约和存储的数据），并且被用来生成区块的状态根哈希。通过MPT树，以太坊可以高效地存储和验证区块链上的状态，并支持高度可扩展的智能合约和去中心化应用。

在2014年4月，比特币网络中的一个全节点-存储和处理所有区块的全部数据的节点-需要占用15GB的存储空间，而且还以每个月超过1GB的速度增长。目前，这一存储空间对台式计算机来说尚可接受，但是手机已经负载不了如此巨大的数据了。未来只有商业机构和爱好者才会充当完整节点。简化支付确认（SPV)协议允许另一种节点存在，这样的节点被成为“轻节点”，它下载区块头，使用区块头确认工作量证明，然后只下载与其交易相关的默克尔树“分支”。这使得轻节点只要下载整个区块链的一小部分就可以安全地确定任何一笔比特币交易的状态和账户的当前余额。

二 安全性分析：

完整性保证：MPT使用节点的哈希值来标识和验证树的完整性。每个节点的哈希值都取决于它的内容，包括键、值以及子节点的哈希值。通过根节点的哈希值，可以验证整个树的完整性。即使树中的一个节点被篡改，它的哈希值将与实际内容不匹配，从而破坏树的完整性。

非可逆性：MPT使用哈希函数进行节点的哈希计算，哈希函数具有非可逆性的特点。这意味着无法通过节点的哈希值逆推出原始的键或值。这种非可逆性提供了保护数据隐私和安全性的重要保障。

防篡改性：MPT的设计通过共享前缀和使用哈希值来压缩存储空间，并减少数据冗余。这使得树中的每个节点都依赖于其父节点和相邻节点的哈希值。如果有人试图篡改或删除树中的某个节点，将会破坏整个树的完整性。因此，MPT提供了一种防篡改的机制。

高效性：MPT通过共享前缀和使用哈希值来优化存储和查询性能。相同前缀的键被合并为一个扩展节点，减少了存储空间的需求。同时，使用哈希值作为节点的标识，可以快速定位和验证节点的内容。这使得MPT具有高效的存储、验证和查询能力。

三．基本结构

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。四，操作

Get操作

将需要查找Key转成Hex编码，得到搜索路径，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。

2. 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。

3. 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操作

1. 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。

### Delete操作

1. 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。

### inclusion proof & exclusion proof & consistency proof

校验过程与传统的Merkle Tree区别不大，其中RFC6962描述了一种针对奇数个数据块build tree以及一致性校验的过程。而非存在性证明需要数据块之间是有序存储的：

非存在性证明的基本逻辑为，找到树中存在的值小于目标元素值的最大元素，并找到树中存在的值大于目标元素值的最小元素，构造这两个元素的存在性证明并去除重复的中间节点。如果根据非存在性证明计算出来的简单Merkle树的散列值与正确的值相等，并且证明中包含的两个叶子节点为相邻的叶子节点，就证明树中确实不存在目标元素。

六，参考文献 https://easythereentropy.wordpress.com/2014/06/04/understan ding-the-ethereum-trie/

https://github.com/ethereum/wiki/wiki/Patricia-Tree <https://github.com/ebuchman/understanding_ethereum_trie>

知乎清源