分析涉及 Merkle Patricia Trie 的以太坊源代码

概述：

Merkle Patricia Trie 简称 MPT，MPT 分别是由三种树组合而成的数据结构，有 Trie 树、Patricia Trie 以及 Merkle 树。

Trie 树，又名为字典树、单词查找或者前缀树，是一种用于快速查找的多分叉树结构，如数字的 Trie 树就是一个十叉树。Trie 树的特点有三个，分别为若根节点不包含字符，那除根节点之外每个节点只包含一个字符、从根节点到某个节点，路径经过的节点连上就为该节点的字符串、每个节点的所有子节点包含的字符串都不同。

Patricia Trie，又名前缀树，是从 Trie 树优化而来的。如果存在一个父节点只有一个子节点，那么父节点将会与子节点合并，这样就可以减少存储空间而且也提高了查找效率。

Merkle 树，又名 Hash 树，是存储哈希值(hash)的树。Merkle 树的叶子都是用于存储数据（hash 值），而父节点则是存储相对应的子节点串联字符串的 hash 值，一级接一级地往上直到根节点。因此，若是修改了子节点的数据，则对应的父节点和 hash 值就会改变，根节点也会随之改变，所以这 Merkle 树就是为了保证数据不被篡改而设计的技术。

以太坊的 MPT（Merkle Patricia Trie）是结合了 Merkle 树和 Patricia 树的数据结构，所以同时拥有着查询和安全性的特点。以太坊的区块头中有三个字段是包含了 MPT 树相关的，分别是交易树（txTrieRoot）、收据树（receiptTrieRoot）、状态树（stateRoot），这些树分别存储了这三颗树的树根 hash 值。

以太坊对这两特点进行优化与改进：

1. **保证树的安全性**

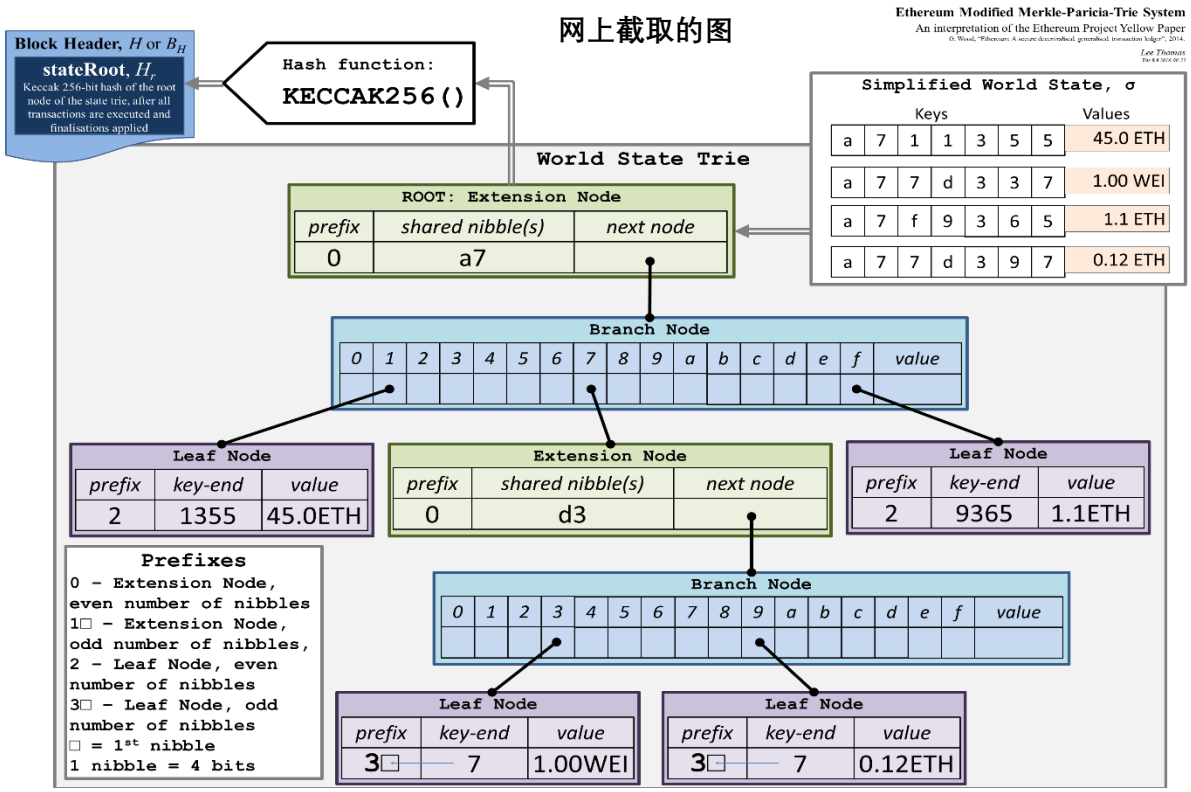
   节点与节点之间的联系不再采用内存指针的方式，而是采用 hash 值得方式。比如由一个父节点存储一个值然后再执行下一个节点的 hash 值，然后将这个 hash 值与实际对应的节点存储在数据库中。当有人篡改此节点的值时，父节点的 hash 值也会被篡改直到根节点也跟着修改，所以我们只需要验证根节点的 hash 值就可以知道以下节点的数据有没有被篡改过。

2. **引入更多节点来提高安全性**

   MPT 节点分为 4 种节点，各个节点存在的功能不同：

   1. 空节点（NULL），一个空串
   2. 叶节点（Leaf Node），包含两项数据（key，value），key 为 16 进制编码，value 是 RLP 编码。
   3. 扩展节点，value 是其他节点散列值，通过这个散列值去链接到其他的节点。
   4. 分支节点，长度为 17 的列表，key 还是 16 进制编码，加上 value，前 16 个元素对应 key 的 16 个十六进制字符，如果一个键值在这个分支终止了，那么最后的一个元素表示为一个值。分支节点既是搜索的终止也可是路径的中间点。

MPT 简单的结构图：



数据结构

MPT 的数据结构可分为 Node 结构和 Trie 结构。Node 结构可分为四种类型，分别是 fullNode、shortNode、hashNode、valueNode，fullNode 对应的是以上提到的分支节点，shortNode 对应的则是叶子节点和扩展节点，这俩类型都为结构性节点。那么除了结构性节点外，还有数据节点也就是 valueNode 和 hashNode，valueNode 只用于存储 value，hashNode 则保存的是 key，key 的对应数据都保存在数据库中。

```
type node interface {
	fstring(string) string
	cache() (hashNode, bool)
}

type(
	fullNode struct {
		Children [17]node // Actual trie node data to encode/decode (needs custom
encoder)
		flags    nodeFlag
	}
	shortNode struct {
		Key   []byte
		Val   node
		flags nodeFlag
	}
	hashNode[]byte
	valueNode[]byte
)
```

Trie 结构，root 为当前根节点，db 是后端数据库的 KV 存储，trie 的结构最终都是需要通过 KV 的形式存储到数据库里去，然后需要时才从数据库里加载。rootHash 是用于存储 hash 值的，通过这个 hash 值可以在数据库里实现出整颗 trie 树。Cache 表示是当前 trie 树的时代，每次调用 Commit 操作的时候，会增加 Trie 树的 Cache 时代。Cache 时代会被附加在 Node 节点上，如果当前的 Cache 时代 - Cachelimit 参数大于 Node 的 Cache 时代，则 Node 会从 Cache 里面卸载，以节约内存。

```
type Trie struct {
        db *Database
        root node
        // Cache generation values.
        // new nodes are tagged with the current generation and unloaded
        // when their generation is older than Cachelimit.
        []byte  roothash
        uint16  Cache , Cachelimit
}
```

创建一个新的 trie 树，初始化 trie 树调用 New 函数，函数接收一个 hash 值和 Database 参数，如果 hash 值不是空（!NULL），说明是从数据库加载的一个已经存在的 trie 树，就调用 trie.resolveHash 方法来加载整颗 trie 树。如果 hash 值为空，则新建一颗 trie 树返回。

```
func New(root common.Hash, db* Database) (*Trie, error) {
        if db == nil{
                panic("trie.New called without a database")
        }
        trie: = &Trie{
                db: db,
        }
        if root != (common.Hash{}) && root != emptyRoot{
                rootnode, err : = trie.resolveHash(root[:], nil)
                if err != nil {
                        return nil, err
                }
                trie.root = rootnode
        }
        return trie, nil
}
```

操作流程

组成 MPT 树结构的基本操作流程分别有三种，是查找、插入或更新、删除等。

一、查找（Get 方法）

```
func(t* Trie) TryGet(key[]byte) ([]byte, error) {
		value, newroot, didResolve, err = t.tryGet(t.root, keybytesToHex(key), 0)
		if err == nil && didResolve{
			t.root = newroot
		}
		return value, err
}

func(t* Trie) tryGet(origNode node, key[]byte, pos int) (value[]byte, newnode node, didResolve
bool, err error) {
	switch n = (origNode).(type) {
	case nil:
		return nil, nil, false, nil
	case valueNode:
		return n, n, false, nil
	case *shortNode:
			if len(key) - pos < len(n.Key) || !bytes.Equal(n.Key, key[pos:pos +
len(n.Key)]) {
				// key not found in trie
				return nil, n, false, nil
			}
			value, newnode, didResolve, err = t.tryGet(n.Val, key, pos + len(n.Key))
			if err == nil && didResolve{
				n = n.copy()
				n.Val = newnode
			}
			return value, n, didResolve, err
	case *fullNode:
			value, newnode, didResolve, err = t.tryGet(n.Children[key[pos]], key,
pos + 1)
			if err == nil && didResolve{
				n = n.copy()
				n.Children[key[pos]] = newnode
			}
			return value, n, didResolve, err
	case hashNode:
			child, err = t.resolveHash(n, key[:pos])
			if err != nil{
				return nil, n, true, err
			}
			value, newnode, _, err = t.tryGet(child, key, pos)
			return value, newnode, true, err
	default:
			panic(fmt.Sprintf("%T: invalid node: %v", origNode, origNode))
	}
}
```

Get 操作流程是为了查找 Key 的值（Raw 编码转换成 Hex 编码），得到的值为搜索路径，从根节点开始搜索与搜索路径内容相同的路径。

Case 节点：

若当前节点为叶子节点（valueNode），存储的内容为数据项的内容，且搜索路径的内容与叶子节点的 key 相同，则表示找到该节点，反之没找到则该节点在树中不存在。

若当前节点为 hash 结点（hashNode）且存储的内容是 hash 值，则利用 hash 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利用递归调用查找(Get)函数。

若当前节点为扩展节点（shortNode），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（fullnode），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子节点，将剩余的搜索路径作为参数递归调用查找函数。

二、更新/插入

Update 是 insert 和 delete 的结合。当用户调用 Update 函数的时候，若 value 不为空，则转换为 insert 函数调用；若 value 为空，则转换为 delete 函数调用。

Insert 插入过程是根据 Get 函数调用后，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，又可分为几个 Case 节点：

若 Node 为叶子节点/扩展结点（shortNode），如果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。如果剩余的搜索路径与当前节点的 key 不完全一致，则将该节点的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子节点作为两个子节点插入到分支节点的子列表中，同时当前节点转换成一个扩展节点；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，修改节点的 dirty 为 true，hash 置空，将节点的标记更新为现在的。

若 Node 为分支节点（fullNode）,如果剩余的搜索路径不为空，则将新的节点作为一个叶子节点插入到对应的子列表中；如果剩余的搜索路径为空，则将新节点的内容存储在分支节点的第 17 个子节点项中的 Value 值。

若当前是一个空树（nil），直接构造一个 shortNode。

若当前为 hash 节点（hashNode），表示当前节点还没有加载到内存里面来，还是存放在数据库里面，那么就调用 t.resolveHash(n,prefix)方法来加载到内存，然后对加载出来的节点调用 insert 方法来进行插入。

```
func(t *Trie) insert(n node, prefix, key[]byte, value node) (bool, node, error) {
		if len(key) == 0 {
//如果key=0，则表示value为一个valueNode，则直接将n转为valueNode冰河value进行比较。看看是否是合法的valueNode
			if v, ok := n.(valueNode); ok{
				return !bytes.Equal(v, value.(valueNode)), value, nil
			}
			return true, value, nil
		}
		switch n := n.(type) { // n : Node
		case *shortNode:
			matchlen := prefixLen(key, n.Key)
			// If the whole key matches, keep this short node as is
			// and only update the value.
			if matchlen == len(n.Key) {
				dirty, nn, err := t.insert(n.Val, append(prefix,
key[:matchlen]...), key[matchlen:], value)
				if !dirty || err != nil{
					return false, n, err
				}
				return true, &shortNode{ n.Key, nn, t.newFlag() }, nil
			}
			// Otherwise branch out at the index where they differ.
			branch := &fullNode{ flags: t.newFlag() }
			var err error
			_, branch.Children[n.Key[matchlen]], err = t.insert(nil, append(prefix,
n.Key[:matchlen + 1]...), n.Key[matchlen + 1:], n.Val)
			if err != nil{
				return false, nil, err
			}
			_, branch.Children[key[matchlen]], err = t.insert(nil, append(prefix,
key[:matchlen + 1]...), key[matchlen + 1:], value)
			if err != nil{
				return false, nil, err
			}
			// Replace this shortNode with the branch if it occurs at index 0.
			if matchlen == 0 {
				return true, branch, nil
			}
			// Otherwise, replace it with a short node leading up to the branch.
			return true, &shortNode{ key[:matchlen], branch, t.newFlag() }, nil

		case *fullNode:
			dirty, nn, err := t.insert(n.Children[key[0]], append(prefix, key[0]),
key[1:], value)
			if !dirty || err != nil{
				return false, n, err
			}
			n = n.copy()
			n.flags = t.newFlag()
			n.Children[key[0]] = nn
			return true, n, nil

		case nil:
			return true, &shortNode{ key, value, t.newFlag() }, nil
```

```
        case hashNode:
                // We've hit a part of the trie that isn't loaded yet. Load
                // the node and insert into it. This leaves all child nodes on
                // the path to the value in the trie.
                rn, err := t.resolveHash(n, prefix)
                if err != nil {
                        return false, nil, err
                }
                dirty, nn, err := t.insert(rn, prefix, key, value)
                if !dirty || err != nil {
                        return false, rn, err
                }
                return true, nn, nil

        default:
                panic(fmt.Sprintf("%T: invalid node: %v", n, n))
        }
}
```

三、删除

Delete 删除过程是根据 Get 函数调用后，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，又可分为几个 Case 节点：

若 Node 为叶子节点/扩展结点(shortNode)，如果剩余的搜索路径与 Node 节点的 key 完全一致，则将整个 Node 删除。如果 Node 的 key 是剩余搜索路径的前缀，则对该节点的 Val 做递归删除调用，否则说明要删除的节点不存在于树中，删除失败

若 Node 节点为分支节点（fullNode），删除孩子列表中相应下标的节点。删除结束后，若 Node 的孩子个数只剩下一个，那么将分支节点替换成叶子节点或扩展节点。若删除成功，则将被修改节点的 dirt 改为 true，hash 置放为空，且将节点的标记更新为现在的。

若 Node 节点为叶子节点（valueNode），直接返回。

若 Node 节点为空（nil），返回空。

若当前为 hash 节点（hashNode），表示当前节点还没有加载到内存里面来，还是存放在数据库里面，那么就调用 t.resolveHash(n,prefix)方法来加载到内存，然后对加载出来的节点调用 delete 方法来进行删除。

```
// TryDelete removes any existing value for key from the trie.
// If a node was not found in the database, a MissingNodeError is returned.
func(t* Trie) TryDelete(key[]byte) error {
                t.unhashed++
                k := keybytesToHex(key)
                _, n, err := t.delete(t.root, nil, k)
```

```
                if err != nil{
                        return err
                }
                t.root = n
                return nil
        }
// delete returns the new root of the trie with key deleted.
// It reduces the trie to minimal form by simplifying
// nodes on the way up after deleting recursively.

func(t* Trie) delete(n node, prefix, key[]byte) (bool, node, error) {
        switch n : = n.(type) {
        case *shortNode:
                matchlen: = prefixLen(key, n.Key)
                if matchlen < len(n.Key) {
                        return false, n, nil // don't replace n on mismatch
                }
                if matchlen == len(key) {
                        return true, nil, nil // remove n entirely for whole matches
                }
                // The key is longer than n.Key. Remove the remaining suffix
                // from the subtrie. Child can never be nil here since the
                // subtrie must contain at least two other values with keys
                // longer than n.Key.
                dirty, child, err : = t.delete(n.Val, append(prefix, key[:len(n.Key)]...),
key[len(n.Key):])
                if !dirty || err != nil{
                        return false, n, err
                }
                switch child : = child.(type) {
                        case *shortNode:
                                // Deleting from the subtrie reduced it to another
                                // short node. Merge the nodes to avoid creating a
                                // shortNode{..., shortNode{...}}. Use concat (which
                                // always creates a new slice) instead of append to
                                // avoid modifying n.Key since it might be shared with
                                // other nodes.
                                return true, & shortNode{ concat(n.Key, child.Key...),
child.Val, t.newFlag() }, nil
                        default:
                                return true, & shortNode{ n.Key, child, t.newFlag() }, nil
                }

        case* fullNode:
                dirty, nn, err : = t.delete(n.Children[key[0]], append(prefix, key[0]),
key[1:])
                if !dirty || err != nil{
                        return false, n, err
                }
                n = n.copy()
                n.flags = t.newFlag()
                n.Children[key[0]] = nn

                // Check how many non-nil entries are left after deleting and
                // reduce the full node to a short node if only one entry is
                // left. Since n must've contained at least two children
```

```
                // before deletion (otherwise it would not be a full node) n
                // can never be reduced to nil.
                // When the loop is done, pos contains the index of the single
                // value that is left in n or -2 if n contains at least two
                // values.
                pos: = -1
                for i, cld : = range & n.Children{
                        if cld != nil {
                                if pos == -1 {
                                        pos = i
                                }
                        else {
                                pos = -2
                                break
                                }
                        }
                }
                if pos >= 0 {
                        if pos != 16 {
                                // If the remaining entry is a short node, it replaces
                                // n and its key gets the missing nibble tacked to the
                                // front. This avoids creating an invalid
                                // shortNode{..., shortNode{...}}.  Since the entry
                                // might not be loaded yet, resolve it just for this
                                // check.
                                cnode, err : = t.resolve(n.Children[pos], prefix)
                                if err != nil{
                                        return false, nil, err
                                }
                                if cnode, ok : = cnode.(*shortNode); ok{
                                        k: = append([]byte{byte(pos)}, cnode.Key...)
                                        return true,& shortNode{k, cnode.Val, t.newFlag()}, nil
                                }
                        }
                        // Otherwise, n is replaced by a one-nibble short node
                        // containing the child.
                        return true, & shortNode{ [] byte{byte(pos)}, n.Children[pos],
t.newFlag() }, nil
                }
                // n still contains at least two values and cannot be reduced.
                 return true, n, nil

        case valueNode:
                return true, nil, nil

        case nil:
                return false, nil, nil

        case hashNode:
                // We've hit a part of the trie that isn't loaded yet. Load
                // the node and delete from it. This leaves all child nodes on
                // the path to the value in the trie.
                rn, err : = t.resolveHash(n, prefix)
                if err != nil{
                        return false, nil, err
                }
```

```
                dirty, nn, err : = t.delete(rn, prefix, key)
                if !dirty || err != nil{
                        return false, rn, err
                }
                return true, nn, nil

        default:
                panic(fmt.Sprintf("%T: invalid node: %v (%v)", n, n, key))
        }
}
```

MFT 的作用

1. MPT 可以存储任意长度的 key 和 value 值数据
2. MPT 提供了一种快速计算所维护数据集哈希标识的机制，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进行了修改，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。
3. MPT 提供了快速状态回滚的机制，即区块链内容发生了重组织，链头发生切换或者是区块链的世界状态（账户信息）需要进行回滚，即对之前的操作进行撤销。
4. 以太坊提供了默克尔证明的证明方法，进行轻节点的扩展，实现简单的支付验证，即在无需维护具体交易信息的前提下，证明某一笔交易是否存在于区块链中。

学号：3190300677

姓名：姚熙源