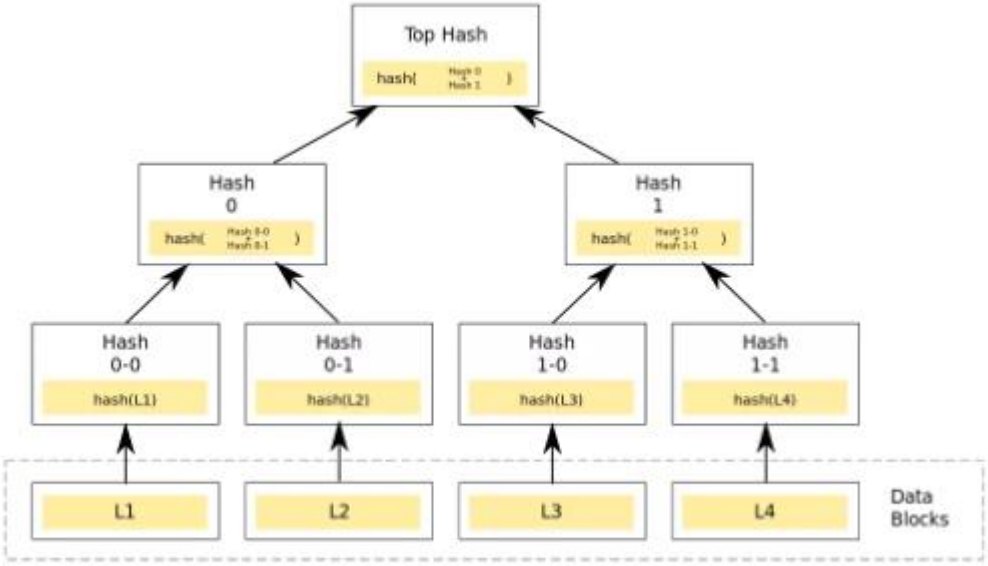
Merkle Patricia Tree 报告

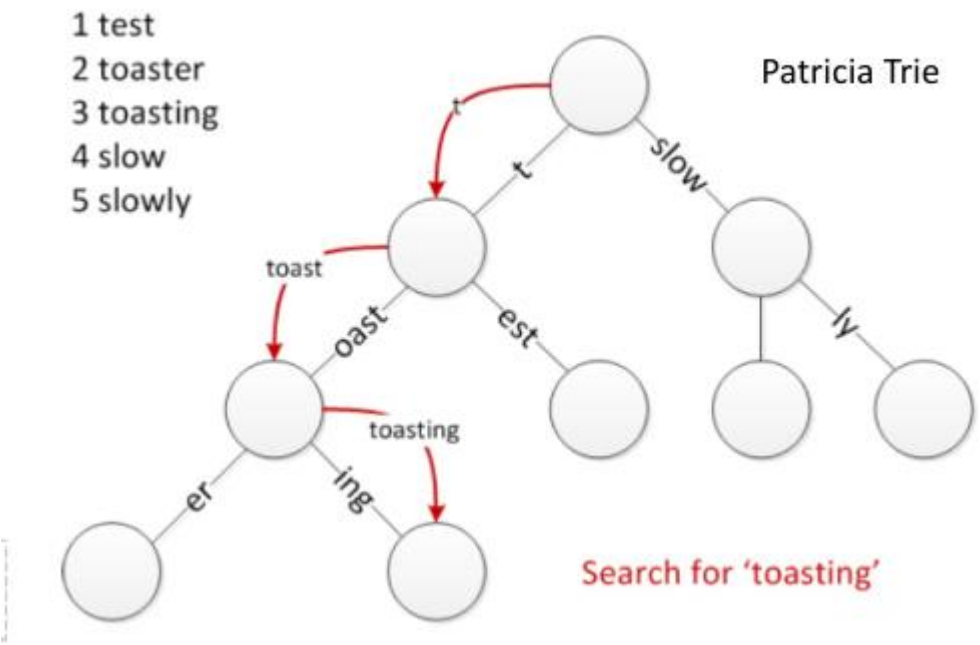
Merkle Patricia Tree 简称 MPT 是一种树形的数据结构，主要被被用于数字货币以太坊中，是从 Merkle tree 和前缀树中发展过来，可以用来做存在性和不存在性证明。

Merkle tree



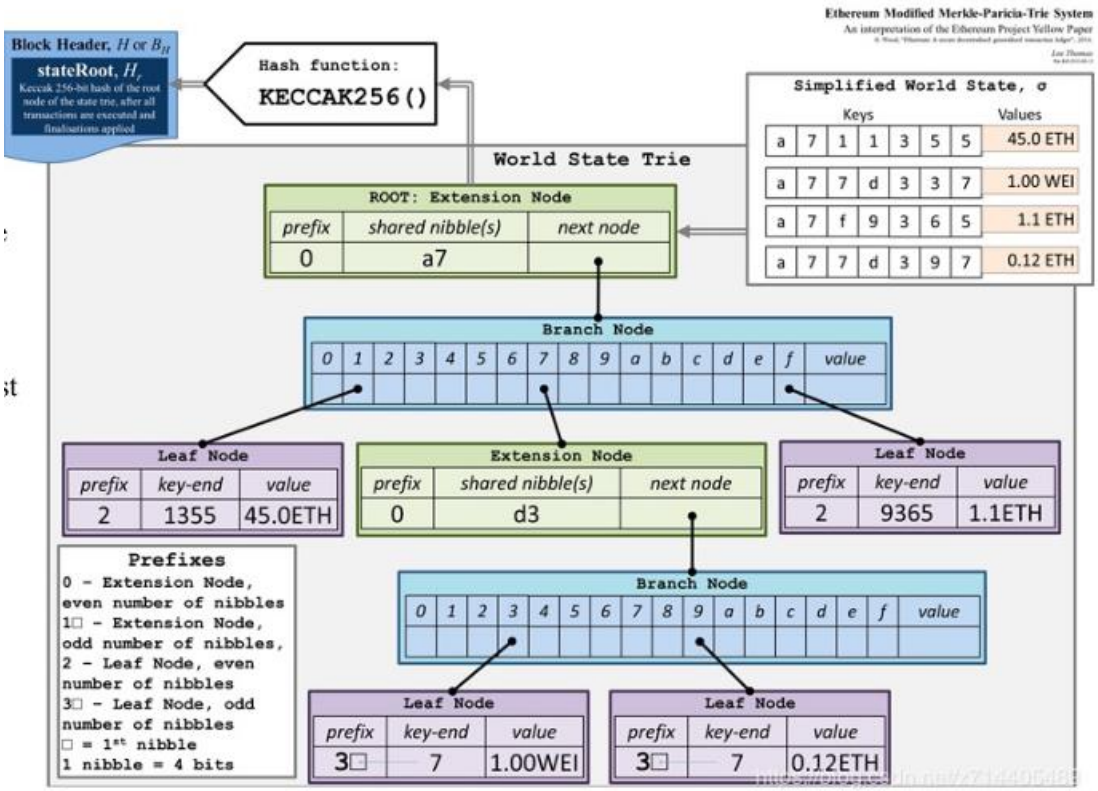
Merkle tree 有三种节点，叶子节点，叶子节点的父结点以及其他节点。叶子节点储存数据，它的父结点储存的是数据的 hash 值，其余的节点储存的是它的左孩子和右孩子储存的 hash 值级联之后的 hash 值。如图，图中最下层是第一种，倒数第二层是第二种，其余是第三种。

Patricia Trie （Radix tree）



即前缀树，比较像英文字典的储存方式，比如现在要找 apple 这个单词，就先找 a，再在所有以 a 开头的单词中找到 p，以此类推。但这样就会产生一个问题，就是如果存储的数据比较长，而且比较少，就会占用许多空间。针对这个问题，可以使用压缩的方法。就是如果一个节点只有一个子节点，就将子节点的内容压缩到父结点之中。

Merkle Patricia Tree



首先是 node 节点，包括 hash 值，编码方式等。

```
type node interface {
        cache() (hashNode, bool)
        encode(w rlp.EncoderBuffer)
        fstring(string) string
}
```

接着是 Fullnode

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

有 17 个子节点，以及一个 nodeflag，其中有 hashnode，即哈希值，以及脏块，显示其有没有被修改。

```
type nodeFlag struct {
        hash  hashNode // cached hash of the node (may be nil)
        dirty bool     // whether the node has changes that must be written to the database
}
```

Shortnode：

```
shortNode struct {
        Key   []byte
        Val   node
        flags nodeFlag
 }
```

Key 是一个键值数组，属于合并节点。Val 指向一个节点，nodeflag 与上面相同。

接着是 tire 的结构

```
type Trie struct {
        db    *Database
        root  node
        owner common.Hash

        // Keep track of the number leaves which have been inserted since the last
        // hashing operation. This number will not directly map to the number of
        // actually unhashed nodes
        unhashed int

        // tracer is the state diff tracer can be used to track newly added/deleted
        // trie node. It will be reset after each commit operation.
        tracer *tracer
}
```

root 为根 node, unhashed 将会跟踪自上次插入以来插入并进行哈希操作的叶子数量。

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

Newtrie 会创造一个新的 trie 节点。先创建一个结构体，在判断 root，当 root 为 nil 的时候，说明创建一个空的 Trie。当 root 不为 nil 的时候，说明为加载一个已经存在的 Trie。

```
func (t *Trie) Update(key, value []byte) {
        if err := t.TryUpdate(key, value); err != nil {
                log.Error(fmt.Sprintf("Unhandled trie error: %v", err))
        }
}
```

```
func (t *Trie) TryUpdate(key, value []byte) error {
        t.unhashed++
        k := keybytesToHex(key)
        if len(value) != 0 {
                _, n, err := t.insert(t.root, nil, k, valueNode(value))
                if err != nil {
                        return err
                }
                t.root = n
        } else {
                _, n, err := t.delete(t.root, nil, k)
                if err != nil {
                        return err
                }
                t.root = n
        }
        return nil
}
```

Update 操作，没有错误的时候执行 tryupdate 操作。Unhashed 加一，将 key 转化成十六进制，当 value 不为空的时候，说明是要刷新节点或者插入一个新的节点。创建一个 valuenode

当 value 为空的时候，说明是要将节点删除，调用了 delete。

```
func (t *Trie) Delete(key []byte) {
        if err := t.TryDelete(key); err != nil {
                log.Error(fmt.Sprintf("Unhandled trie error: %v", err))
        }
}

// 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
}
```

删除操作。

Insert 操作，插入。

```
func (t *Trie) insert(n node, prefix, key []byte, value node) (bool, node, error) {
	if len(key) == 0 {
		if v, ok := n.(valueNode); ok {
			return !bytes.Equal(v, value.(valueNode)), value, nil
		}
		return true, value, nil
	}
	switch n := n.(type) {
	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
		}
```

当 trie 为空的时候，root 就是为 nil。那么第一次插入的时候，走的就是这个流程。创建了 shortnode。并将其返回了。在 Update 函数中，trie 的 root 会被赋值为这个 shortnode。那么第一次的插入就完成了。再继续插入的话，执行流程就走到这里了。第一次插入的 root 肯定是一个 shortnode。对 key 进行判断，如果 key 与要插入的 node 的 key 是一致的话，只需要构建一个新的 shortnode，进行替换即可。

```
        t.tracer.onInsert(append(prefix, key[:matchlen]...))

        // 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:
        // New short node is created and track it in the tracer. The node identifier
        // passed is the path from the root node. Note the valueNode won't be tracked
        // since it's always embedded in its parent.
        t.tracer.onInsert(prefix)

        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))
    }
```

变量 k 为插入节点 node 的 node_key(为了区分变量)，变量 key 为要被插入节点的 node_key。计算其 k 和 key 的重合度。如过重合度相同则直接插入。说明 k 和 key 的长度不一样，那么需要将 shortnode 拆分裂变为两个。pnode，nnode（这里面还有递归插入后续的动作），而当前的 node 则替换成 fullnode，并将两个子节点 set 到这个 fullnode 下。最后使用递归的方法。