Merkle Patricia Tree代码分析

代码参考来源：https://github.com/ethereumjs/merkle-patricia-tree

主要分析MPT中重要代码

一、编码：Raw，Hex，HP编码的转换

1.Raw

Raw为原始的key数组，无需转换。

2. Hex

- ```
  func keybytesToHex(str []byte) []byte {
      l := len(str)*2 + 1
      var nibbles = make([]byte, l)
      for i, b := range str {
          nibbles[i*2] = b / 16
          nibbles[i*2+1] = b % 16
      }
      nibbles[l-1] = 16
      return nibbles
  }
  ```
- **Raw为原本的编码格式，将其转换为Hex编码格式，由于Hex编码是按照4bit的格式，而Raw为8bit。代码将Raw一分为二，赋给两个Hex格式的数据。**

3. HP
  ```

- func hexToCompact(hex []byte) []byte {
    terminator := byte(0)
    if hasTerm(hex) {
        terminator = 1
        hex = hex[:len(hex)-1]
    }
    buf := make([]byte, len(hex)/2+1)
    buf[0] = terminator << 5
    if len(hex)&1 == 1 {
        buf[0] |= 1 << 4
        buf[0] |= hex[0]
        hex = hex[1:]
    }
    decodeNibbles(hex, buf[1:])
    return buf
}

hex 编码解决了 key 是 keybytes 形式的数据插入 MPT 的问题，但这种方式有数据冗余的问题，对于 shortNode，目前 hex 格式下的 key，长度会是原来 keybytes 格式下的两倍，这一点对于节点的哈希计算影响很大，compact 编码用于对 hex 格式进行优化。compact encoding 的主要思路是将 Hex 格式字符串先恢复到 keybytes 格式，同时加入当前编码的标记位，表示奇偶不同长度的 hex 格式。

二、MPT中的重要操作

1. 初始化

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

```
        trie.root = rootnode
    }
    return trie, nil
}
```

判断根节点的哈希值是否为空，如果为空，则将创建一个树，如果不为空，则通过resolveHash函数加载MPT树

2. 增加一个节点

```
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 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
        }
        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
        }
        if matchlen == 0 {
            return true, branch, nil
```

```
        }
        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:
        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))
    }
}
```

这是一个递归调用的方法，从根节点开始，一直往下找，直到找到可以插入的点，进行插入操作。

大体上分为四种情况分析：（根据节点类型）

1. **shortNode：**

已知shortNode为叶子节点或扩展节点，先计算出节点的前缀，并比较前缀的长度与key的长度，若value一样，则两个节点完全相同，插入失败，返回false。

如果公共前缀不完全匹配，那么就需要把公共前缀提取出来形成一个独立的节点(扩展节点),扩展节点后面连接一个branch节点，branch节点后面看情况连接两个short节点。首先构建一个branch

节点(branch := &fullNode{flags: t.newFlag()}),然后在branch节点的Children位置调用t.insert插入剩下的两个short节点。

2. **fullNode**

   只要节点不dirty不出错，那么直接将其children递归增加到树上。

3. **nil**

   说明节点为空，不做操作，直接返回shortnode

4. **hashNode**

   查找得知，hashnode代表节点还未加载，所以需要先将节点加载到内存中再进行insert。

3. 删除一个节点

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

func (t *Trie) TryDelete(key []byte) error {
    k := keybytesToHex(key)
    _, n, err := t.delete(t.root, nil, k)
    if err != nil {
        return err
    }
    t.root = n
    return nil
}

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
        }
        if matchlen == len(key) {
            return true, nil, nil
        }
        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:
            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

        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 {
                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
                }
            }
```

```
            return true, &shortNode{[]byte{byte(pos)}, n.Children[pos], t.newFlag()}, nil
        }
        return true, n, nil

    case valueNode:
        return true, nil, nil

    case nil:
        return false, nil, nil

    case hashNode:
        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))
    }
}
```

与增加一个节点类似，需要考虑四种情况：（如果为空节点，直接返回空节点）

1. **shortNode：**

   先对比节点长度与key的长度，如果节点长度小，则说明所要删除的节点与此不匹配，只返回值不进行操作，而如何长度相等，则匹配成功，返回值并代表成功删除。如果前两种情况都没发生，则说明需要删除child，递归进行删除。

2. **fullNode：**

   如果 fullNode 删除后有两个及以上的子节点，删除后返回根节点即可，如果删除后只有一个子节点，那么需要将该根节点变成 shortNode 返回。

3. **nil：**

   节点为空，直接返回

4. **hashNode：**

   说明节点还不在内存中，先调入内存再进行删除。

4. 改动一个节点

```
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 {
    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
}
```

判断节点的value是否为空，如果为空则返回错误将其删除，否则将调用insert。

5. 查找一个节点

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

func (t *Trie) TryGet(key []byte) ([]byte, error) {
    key = keybytesToHex(key)
```

```
    value, newroot, didResolve, err := t.tryGet(t.root, 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)]) {
            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
            n.flags.gen = t.cachegen
        }
        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.flags.gen = t.cachegen
            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))
        }
    }
```

查找操作就是简单的遍历树。

三、提交

提交阶段会将内存中的所有 trie 写入到数据库中。在这期间会根据 dirty 值做一件非常重要的事情，如果 dirty 置为 true 了，表明其代表的父节点有改动需要提交，hashNode 的成员 hash 设为空，需重新进行计算得到 hashNode 的哈希值。

```
func (t *Trie) Commit(onleaf LeafCallback) (root common.Hash, err error) {
    if t.db == nil {
        panic("commit called on trie with nil database")
    }
    hash, cached, err := t.hashRoot(t.db, onleaf)
    if err != nil {
        return common.Hash{}, err
    }
    t.root = cached
    t.cachegen++
    return common.BytesToHash(hash.(hashNode)), nil
}

func (t *Trie) hashRoot(db *Database, onleaf LeafCallback) (node, node, error) {
    if t.root == nil {
        return hashNode(emptyRoot.Bytes()), nil, nil
    }
    h := newHasher(t.cachegen, t.cachelimit, onleaf)
    defer returnHasherToPool(h)
    return h.hash(t.root, db, true)
}

func (h *hasher) hash(n node, db *Database, force bool) (node, node, error) {
    if hash, dirty := n.cache(); hash != nil {
        if db == nil {
```

```
                return hash, n, nil
            }
            if n.canUnload(h.cachegen, h.cachelimit) {
                cacheUnloadCounter.Inc(1)
                return hash, hash, nil
            }
            if !dirty {
                return hash, n, nil
            }
        }
        collapsed, cached, err := h.hashChildren(n, db)
        if err != nil {
            return hashNode{}, n, err
        }
        hashed, err := h.store(collapsed, db, force)
        if err != nil {
            return hashNode{}, n, err
        }
        cachedHash, _ := hashed.(hashNode)
        switch cn := cached.(type) {
        case *shortNode:
            cn.flags.hash = cachedHash
            if db != nil {
                cn.flags.dirty = false
            }
        case *fullNode:
            cn.flags.hash = cachedHash
            if db != nil {
                cn.flags.dirty = false
            }
        }
        return hashed, cached, nil
}

func (h *hasher) hashChildren(original node, db *Database) (node, node, error) {
        var err error

        switch n := original.(type) {
        case *shortNode:
            collapsed, cached := n.copy(), n.copy()
```

```
            collapsed.Key = hexToCompact(n.Key)
            cached.Key = common.CopyBytes(n.Key)

            if _, ok := n.Val.(valueNode); !ok {
                collapsed.Val, cached.Val, err = h.hash(n.Val, db, false)
                if err != nil {
                    return original, original, err
                }
            }
            return collapsed, cached, nil

        case *fullNode:
            collapsed, cached := n.copy(), n.copy()

            for i := 0; i < 16; i++ {
                if n.Children[i] != nil {
                    collapsed.Children[i], cached.Children[i], err = h.hash(n.Children[i], db, false)
                    if err != nil {
                        return original, original, err
                    }
                }
            }
            cached.Children[16] = n.Children[16]
            return collapsed, cached, nil

        default:
            return n, original, nil
        }
}
```

大体上看，就是利用hash递归整个树，重新对MPT进行hash，把存放在cache中的值保存下来，利用hashchildren把叶子依次替换，直到递归到根节点，

四、序列化与反序列化

```
func (h *hasher) store(n node, db *Database, force bool) (node, error) {
    if _, isHash := n.(hashNode); n == nil || isHash {
        return n, nil
    }
    h.tmp.Reset()
    if err := rlp.Encode(&h.tmp, n); err != nil {
```

```
            panic("encode error: " + err.Error())
        }
        if len(h.tmp) < 32 && !force {
            return n, nil
        }
        hash, _ := n.cache()
        if hash == nil {
            hash = h.makeHashNode(h.tmp)
        }

        if db != nil {
            hash := common.BytesToHash(hash)

            db.lock.Lock()
            db.insert(hash, h.tmp, n)
            db.lock.Unlock()

            if h.onleaf != nil {
                switch n := n.(type) {
                case *shortNode:
                    if child, ok := n.Val.(valueNode); ok {
                        h.onleaf(child, hash)
                    }
                case *fullNode:
                    for i := 0; i < 16; i++ {
                        if child, ok := n.Children[i].(valueNode); ok {
                            h.onleaf(child, hash)
                        }
                    }
                }
            }
        }
        return hash, nil
    }
```

它通过调用rip.Encode 对节点进行 RLP 编码，然后计算哈希值，通过 db.insert 将其插入数据库，需注意的是分别对 shortNode，fullNode 调用的 onleaf方法，它用来存储外部的 MPT。

在遍历的过程中，如果遇到了 hashNode，需要动态加载这个节点，这个方法就是 resolve。resolve 方法会调用 `trie/database.go` 的 `node` 方法，先从内存里拿，否则从硬盘里拿，接着调用 `trie/node.go` 的 `mustDecodeNode`， `mustDecodeNode` 是 `decodeNode` 的简单封装，

在这里可以看到调用 rlp 模块进行反序列化的过程，根据 RLP 的 list 的长度来判断这个编码是什么节点，如果是2那么就是 shortNode，如果是17就是 fullNode，根据节点的不同调用相应的 decode 方法。（来源于knarfeh笔记）