浙江大学

本科实验报告

课程名称：	区块链与数字货币
实验名称：	以太坊 MPT 源码分析
姓　　名：	唐敏跃
学　　院：	计算机学院
系：	计算机系
专　　业：	软件工程
学　　号：	3180104412
指导教师：	杨小虎

2020 年　12 月 18 日

1. MPT 简介

MPT 是以太坊存储数据的核心结构，是由 Merkle Tree 和 Patricia Tree 结合而成的一种树形结构。它存储了以太坊区块链中的账户以及交易信息。

MPT 的作用：

1) 存储任意长度的 key-value 键值对数据；
2) 提供了一种快速计算所维护数据集哈希标识的机制；
3) 提供了快速状态回滚的机制；
4) 提供了一种称为默克尔证明的证明方法，进行轻节点的扩展，实现简单支付验证；

MPT 有三种节点类型：

1) 分支结点（branch node）：包含 16 个分支，以及 1 个 value
2) 扩展结点（extension node）：只有 1 个子结点
3) 叶子结点（leaf node）：没有子结点，包含一个 value

2. 源码分析

由于能力有限，这里只分析 MPT 树的基本操作：创建、插入、查找、删除。其余操作例如序列化和反序列化，默克尔证明等在此不涉及。

1. MPT 中 Node 的定义。node 一共分为四种类型：fullNode，shortNode，hashNode，valueNode。fullNode 对应了分支节点，包含 17 个 node，其中有 16 个分支和 1 个 value。shortNode 对应了扩展节点和叶子节点，通过 Val 的类型判断具体是扩展节点还是叶子节点。shortNode.Val 如果为数据节点（valueNode），则说明此节点为数据节点；如果为另一个节点的引用，则说明此节点为扩展节点。hashNode 存储节点在数据库中的哈希索引，目的是为了方便树的动态加载。在没用到节点具体内容时，可以只用 hashNode 代表此节点；用到该节点时，再从数据库加载节点的具体内容。

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

type (
    fullNode struct {
        Children [17]node //
        flags    nodeFlag
    }
    shortNode struct {
        Key   []byte
        Val   node
        flags nodeFlag
    }
    hashNode  []byte
    valueNode []byte
)
```

2. Trie 的结构。root 存储了当前树的根节点。db 是数据库指针。Trie 的结构最终都要以 KV 的形式存入数据库，然后启动的时候从数据库中加载相应数据。unhashed 记录了上次 hash 操作以后新插入的叶结点数量。

```
type Trie struct {
    db   *Database
    root node
    // Keep track of the number leafs which have been inserted since the last
    // hashing operation. This number will not directly map to the number of
    // actually unhashed nodes
    unhashed int
}
```

3. Trie 树的初始化。New 函数接收一个 hash 值和一个 database 指针。如果 hash 值不为空，则将其作为根节点的 hash 值，调用 trie.resolveHash 从数据库加载 Trie 树。如果 hash 值为空，则新建一个树返回。

```
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
}
```

4. Trie 树的查找。首先将需要查找 Key 的 Raw 编码转换成 Hex 编码，得到搜索路径；然后从根节点递归搜索与搜索路径内容一致的路径。
Get 为封装好的查找接口。其中调用了 TryGet

```
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
}
```

TryGet 将 Key 转为 Hex 编码，并调用了 tryGet 真正执行查找操作。

```
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
}
```

tryGet 中进行了 trie 的递归遍历。

若当前节点为 shortNode。首先检验当前节点前缀是否与搜索路径匹配，若不匹配则直接返回；若匹配则递归调用 tryGet 函数，同时将搜索路径减去当前节点前缀。**pos+len(n.Key)** 完成了搜索路径“减去”前缀的操作。

```
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
```

若当前节点为 fullNode。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。key[pos]代表还未匹配的搜索路径第一个字节，由它选择需要查找的孩子节点，因此会有 n.Children[key[pos]]。

```
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
```

若当前节点为 hashNode，说明该节点还未加载，需要先调用 resolveHash 函数从数据库加载该节点。

```
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
```

若当前节点为 valueNode，说明已查找到对应 value，直接返回即可。

```
case valueNode:
    return n, n, false, nil
```

5. Trie 树的插入。

Insert 是一个递归操作，从根节点开始往下寻找可以插入的节点，执行插入操作。参数 node 是当前操作的节点，prefix 是已经处理完的 key， key 是还没有处理完的 key，value 是需要插入的节点。 返回值 bool 代表插入是否成功，node 返回插入成功后的 value 节点，error 返回错误信息。

```
func (t *Trie) insert(n node, prefix, key []byte, value node) (bool, node, error)
```

如果 key 已经全部处理完，说明节点插入完成。

```
if len(key) == 0 {
    if v, ok := n.(valueNode); ok {
        return !bytes.Equal(v, value.(valueNode)), value, nil
    }
    return true, value, nil
}
```

如果当前节点类型是 nil，则说明当前子树为空，新建一个 shortNode 节点返回作为树根节点。

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

如果当前的节点是 fullNode，则递归调用子结点的 insert 操作。子结点同样由 key 中的第一个字节决定。

```
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
```

如果当前节点为 shortNode，则有多种可能的操作。如果 shortNode 的 key 和 value 的 key 匹配，那么直接更新 shortNode 的值，把该 shortNode 作为插入后的节点返回即可。如果不匹配，则需要将 shortNode 的子节点改为一个分支节点（branch 节点）。分支节点再连接原来 shortNode 的子结点和新插入的节点。

（源代码截图较长，在此不放出）。

如果当前节点为 hashNode，说明该节点还未从数据库加载，需要先通过 hash 值从数据库加载对应节点，再在该节点上调用 insert 函数进行处理。

```
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
```

6. Trie 树的删除操作。

删除的主要操作在 delete 函数中。基本原理和插入类似，递归寻找需要删除的节点。

```
func (t *Trie) delete(n node, prefix, key []byte) (bool, node, error)
```

若当前节点为 shortNode。如果节点的 key 与搜索路径完全匹配，则直接删除当前节点。若 key 为搜索路径的前缀，则递归调用 delete 函数。若不匹配，则删除失败。

```
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
    }
    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
    }
```

若当前节点为 fullNode，则删除相应下标的子结点。删除结束后，如果节点只剩下一个，则将该节点转变为一个 shortNode。
（源代码较长，在此不放出截图）

hashNode，valueNode，nil 的处理都和 insert 操作类似，在此不再赘述。