Research report on MPT

=======

Go语法说明

-------

没有学过go语言，临时学习了一下，整体结构和c++这种语言差异不大，但是顺序语义还是有区别的

Go语言声明的变量需要用空格隔开

数据类型与c++相似，但是（本次读的代码里）多了byte（一个字节也就是uint8）

Go语言声明变量使用var

Var identifier typr这个结构

常量使用const（）

运算符类似于c++

条件表达式：if后直接跟布尔表达式

函数结构：

func function\_name( [parameter list] ) [return\_types] {

函数体

}

其他代码中出现的语句：

Nil是go语言中预先的标识符，可以直接使用而不用声明（在本代码中多次出现）

Panic：Go中可以抛出一个panic的异常，然后在defer中通过recover捕获这个异常，然后正常处理。（同样在本代码中出现）

赋值用：/：=

代码解读

----

这个代码真的十分难读，可能主要是因为go语言不熟悉，大量的nil使用和符号带来了理解困难

节点类型

1. type (
2. fullNode struct {
3. Children [17]node // Children前16个代表0-9a-f的子节点
4. flags    nodeFlag
5. }
6. shortNode struct {
7. Key   []byte //key是一个数组
8. Val   node //指向一个节点
9. flags nodeFlag //标识树是否变化
10. }
11. hashNode  []byte
12. valueNode []byte
13. )

Trie部分：

结构：

1. type Trie struct {
2. db    \*Database
3. root  node
4. owner common.Hash
5. unhashed int
6. tracer \*tracer
7. }

Unhashed 用于跟踪自从上次被插入的叶节点

Tracer用于跟踪新的添加或删除，会在每次操作后reset

1. func (t \*Trie) newFlag() nodeFlag {
2. return nodeFlag{dirty: true}
3. }

Newflag返回新创建节点的缓存标志值

1. func (t \*Trie) Copy() \*Trie {
2. return &Trie{
3. db:       t.db,
4. root:     t.root,
5. owner:    t.owner,
6. unhashed: t.unhashed,
7. tracer:   t.tracer.copy(),
8. }
9. }

返回Trie的一个copy

1. func New(owner common.Hash, root common.Hash, db \*Database) (\*Trie, error) {
2. return newTrie(owner, root, db)
3. }

New函数从db中创建一个含有已存在根节点的trie，调用如下函数：

1. func newTrie(owner common.Hash, root common.Hash, db \*Database) (\*Trie, error) {
2. if db == nil {
3. panic("trie.New called without a database")
4. }
5. trie := &Trie{
6. db:    db,
7. owner: owner,
8. }
9. if root != (common.Hash{}) && root != emptyRoot {
10. rootnode, err := trie.resolveHash(root[:], nil)
11. if err != nil {
12. return nil, err
13. }
14. trie.root = rootnode
15. }
16. return trie, nil
17. }

这个函数用于构造一个trie

1. func newWithRootNode(root node) \*Trie {
2. return &Trie{
3. root: root,
4. db: NewDatabase(rawdb.NewMemoryDatabase()),
5. }
6. }

这个相似的函数用于初始化一个树（给出一个root）

1. func (t \*Trie) NodeIterator(start []byte) NodeIterator {
2. return newNodeIterator(t, start)
3. }

这个函数没看懂在做什么，返回了一个树节点的迭代

1. func (t \*Trie) Get(key []byte) []byte {
2. res, err := t.TryGet(key)
3. if err != nil {
4. log.Error(fmt.Sprintf("Unhandled trie error: %v", err))
5. }
6. return res
7. }

Get返回在trie中储存的key值，这个值不能为调用所改变（否则报错）

1. func (t \*Trie) TryGet(key []byte) ([]byte, error) {
2. value, newroot, didResolve, err := t.tryGet(t.root, keybytesToHex(key), 0)
3. if err == nil && didResolve {
4. t.root = newroot
5. }
6. return value, err
7. }

这个和上面的函数相似，检查了是否存在节点

1. func (t \*Trie) TryGetNode(path []byte) ([]byte, int, error) {
2. item, newroot, resolved, err := t.tryGetNode(t.root, compactToHex(path), 0)
3. if err != nil {
4. return nil, resolved, err
5. }
6. if resolved > 0 {
7. t.root = newroot
8. }
9. if item == nil {
10. return nil, resolved, nil
11. }
12. return item, resolved, err
13. }

这个函数通过编码的路径来检索一个节点，如果有不存在的路径请求，就把他丢弃

如果到达了请求的路径，就返回现在的节点

以上的函数在源代码中很多有重载函数（不再复述）

1. func (t \*Trie) Update(key, value []byte) {
2. if err := t.TryUpdate(key, value); err != nil {
3. log.Error(fmt.Sprintf("Unhandled trie error: %v", err))
4. }
5. }

Update将key与trie中的value关联，随后调用get返回value，如果value长度为零，存在的值都将从trie中删除然后调用get返回nil

1. func (t \*Trie) insert(n node, prefix, key []byte, value node) (bool, node, error) {
2. if len(key) == 0 {
3. if v, ok := n.(valueNode); ok {
4. return !bytes.Equal(v, value.(valueNode)), value, nil
5. }
6. return true, value, nil
7. }
8. switch n := n.(type) {
9. case \*shortNode:
10. matchlen := prefixLen(key, n.Key)
11. //*如果整个键匹配，就保持shortnode不变并且值更新value*
12. if matchlen == len(n.Key) {
13. dirty, nn, err := t.insert(n.Val, append(prefix, key[:matchlen]...), key[matchlen:], value)
14. if !dirty || err != nil {
15. return false, n, err
16. }
17. return true, &shortNode{n.Key, nn, t.newFlag()}, nil
18. }
19. branch := &fullNode{flags: t.newFlag()}
20. var err error
21. \_, branch.Children[n.Key[matchlen]], err = t.insert(nil, append(prefix, n.Key[:matchlen+1]...), n.Key[matchlen+1:], n.Val)
22. if err != nil {
23. return false, nil, err
24. }
25. \_, branch.Children[key[matchlen]], err = t.insert(nil, append(prefix, key[:matchlen+1]...), key[matchlen+1:], value)
26. if err != nil {
27. return false, nil, err
28. }
29. //新的分支节点被创建为原始短节点的子节点，跟踪程序中新插入的节点
30. if matchlen == 0 {
31. return true, branch, nil
32. }
33. t.tracer.onInsert(append(prefix, key[:matchlen]...))
34. //用一个指向分支的短节点替换
35. return true, &shortNode{key[:matchlen], branch, t.newFlag()}, nil
36. case \*fullNode:
37. dirty, nn, err := t.insert(n.Children[key[0]], append(prefix, key[0]), key[1:], value)
38. if !dirty || err != nil {
39. return false, n, err
40. }
41. n = n.copy()
42. n.flags = t.newFlag()
43. n.Children[key[0]] = nn
44. return true, n, nil
45. case nil:
46. //创建新的短节点并trace它，节点标识符是短节点的路径
47. t.tracer.onInsert(prefix)
48. return true, &shortNode{key, value, t.newFlag()}, nil
49. case hashNode:
50. 当碰到trie还没被加载的部分，加载节点并插入
51. rn, err := t.resolveHash(n, prefix)
52. if err != nil {
53. return false, nil, err
54. }
55. dirty, nn, err := t.insert(rn, prefix, key, value)
56. if !dirty || err != nil {
57. return false, rn, err
58. }
59. return true, nn, nil
60. default:
61. panic(fmt.Sprintf("%T: invalid node: %v", n, n))
62. }
63. }

这个函数比较长，说明改为注释

1. func (t \*Trie) Delete(key []byte) {
2. if err := t.TryDelete(key); err != nil {
3. log.Error(fmt.Sprintf("Unhandled trie error: %v", err))
4. }
5. }

从trie删除这个key对应的所有存在的value

1. func concat(s1 []byte, s2 ...byte) []byte {
2. r := make([]byte, len(s1)+len(s2))
3. copy(r, s1)
4. copy(r[len(s1):], s2)
5. return r
6. }

拼接字符串

1. func (t \*Trie) Hash() common.Hash {
2. hash, cached, \_ := t.hashRoot()
3. t.root = cached
4. return common.BytesToHash(hash.(hashNode))
5. }

Hash返回trie的根哈希值

1. func (t \*Trie) Commit(onleaf LeafCallback) (common.Hash, int, error) {
2. if t.db == nil {
3. panic("commit called on trie with nil database")
4. }
5. defer t.tracer.reset()
6. if t.root == nil {
7. return emptyRoot, 0, nil
8. }
9. //导出所有dirty节点的哈希
10. rootHash := t.Hash()
11. h := newCommitter()
12. defer returnCommitterToPool(h)
13. //在启动之前，首先检查是否需要commit
14. if hashedNode, dirty := t.root.cache(); !dirty {
15. //将根节点替换为原始哈希值以确保所有被resolved的节点在commit后被删除
16. t.root = hashedNode
17. return rootHash, 0, nil
18. }
19. var wg sync.WaitGroup
20. if onleaf != nil {
21. h.onleaf = onleaf
22. h.leafCh = make(chan \*leaf, leafChanSize)
23. wg.Add(1)
24. go func() {
25. defer wg.Done()
26. h.commitLoop(t.db)
27. }()
28. }
29. newRoot, committed, err := h.Commit(t.root, t.db)
30. if onleaf != nil {
31. //在newcommitter中创建leafch，commitloop只从中读取，commit是写入操作
32. close(h.leafCh)
33. wg.Wait()
34. }
35. if err != nil {
36. return common.Hash{}, 0, err
37. }
38. t.root = newRoot
39. return rootHash, committed, nil
40. }

Commit函数，将所有节点写进trie的数据库，并跟踪内外部引用，同样由于函数较长，说明改为注释。

1. func (t \*Trie) hashRoot() (node, node, error) {
2. if t.root == nil {
3. return hashNode(emptyRoot.Bytes()), nil, nil
4. }
5. h := newHasher(t.unhashed >= 100)
6. defer returnHasherToPool(h)
7. hashed, cached := h.hash(t.root, true)
8. t.unhashed = 0
9. return hashed, cached, nil
10. }

计算根节点哈希值（最关键的函数没有之一）

1. func (t \*Trie) Reset() {
2. t.root = nil
3. t.owner = common.Hash{}
4. t.unhashed = 0
5. t.tracer.reset()
6. }

充值根节点并清除状态