以太坊源码解析：state

2019-06-19 fatcat22 ethereum 原创 ethereum state 源码解析

> 本篇文章分析的源码地址为：https://github.com/ethereum/go-ethereum
> 分支：master
> commit id: 257bfff316e4efb8952fbeb67c91f86af579cb0a

引言

对于任何一个区块链项目来说，账户信息的保存都是关键。在以太坊中，保存账户信息的代码是由 state 模块实现的。

与比特币区块链不同的是，以太坊没有使用 UTXO（Unspent Transaction Output） 模型，而是使用的账户余额模型。这篇文章，我们就来看看 state 模块是如何实现账户余额模型的。

什么是 state

要介绍 state，就不得不提区块链的账户模型了。在各个区块链项目中，都会有一个账户地址，类似于我们的银行账户；每个账户都会对应着一些信息，比如有多少币等，类似于我们在银行某个账户下的余额。而保存这些账户对应信息的方式，就是账户模型。

目前在区块链的世界里有两种账户模型：UTXO(Unspent Transaction Output) 模型和账户余额模型。UTXO 的中文翻译为 「未花费的交易输出」。这种方式不记录账户余额，只记录每笔交易（一次转账就是一笔交易），账户的余额是通过计算账户的所有历史交易得出的（想像一下如果你知道你老婆/老公的银行账户的每笔交易，那么你就可以算出她/他现在卡还有多少钱了）。（这篇文章里我们不详细讨论 UTXO 模型，感兴趣的读者可以自己搜索相关文章，网上的讲解还是挺多的）。

账户余额模型与我们常用到的银行账户相似，都是保存了我们账户的余额。当有人给我们转账时，就将余额的数字加上转账的值；当我们转账给别人时，就将余额的数字减去转账的值。

这么看来，账户余额模型是比较容易理解的。以太坊使用的就是账户余额模型，而实现这一模型的，正是 state 模块。之所以模块名叫 state，我猜也是因为它就像一个状态机：它记录了每个账户的状态，每当有交易发生时，就更改一下相应账户的状态。

state 模块中主要的对象是 `StateDB` 对象，正是它记录了每个账户的信息，其中包含 balance（以太币的数量）、nonce（交易标号，见「重放攻击」小节），等信息。这从它的方法中就能看出来，比如：

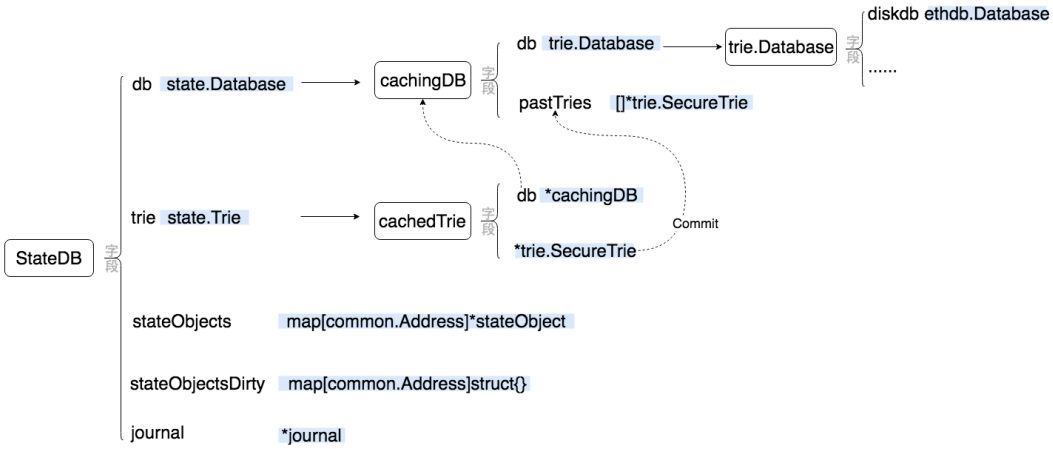
```
func (self *StateDB) GetBalance(addr common.Address) *big.Int
func (self *StateDB) AddBalance(addr common.Address, amount *big.Int)
func (self *StateDB) SubBalance(addr common.Address, amount *big.Int)
func (self *StateDB) SetBalance(addr common.Address, amount *big.Int)
func (self *StateDB) GetNonce(addr common.Address) uint64
func (self *StateDB) SetNonce(addr common.Address, nonce uint64)
func (self *StateDB) GetCode(addr common.Address) []byte
func (self *StateDB) SetCode(addr common.Address, code []byte)
......
```

所以，总得来说，以太坊的 state 实现了账户余额模型，保存了以太坊账户的所有信息。每当这些信息发生改变，state 对象就会发生改变，就像一个状态机一样。

实现架构

state 的实现其实比较简单，但由于加了一些缓存功能，乍看上去会觉得比较乱。我画了一张图来示意 state 的主要实现：

Content



从图上可以看出，state 模块对外的主要对象是 `StateDB`，这个对象提供了各种管理账户信息的方法（可以很容易地在 state/statedb.go 中查看到，这里就不一一列举了）。在对象的内部主要有四个字段，下面我们分别简单的解释一下这四个字段。

`stateObjects` 是一个 map，用来缓存所有从数据库（也就是 `trie` 字段）中读取出来的账户信息，无论这些信息是否被修改过都会缓存在这里。因此在需要将所有数据提交到数据库或 `trie` 字段代表的 trie 对象时，只需将 `stateObjects` 中保存的信息提交即可（当然需要借助 `stateObjectsDirty` 字段踢除没被修改过的信息）。

`stateObjectsDirty` 很显然是用来记录哪些账户信息被修改过了。需要注意的是，这个字段并不时刻与 `stateObjects` 对应，并且也不会在账户信息被修改时立即修改这个字段。事实上，这个字段是与 `journal` 字段相关的：在进行某些操作时（`StateDB.Finalise` 和 `StateDB.Commit`）时，才会将 `journal` 字段中记录的被修改的账户整理到 `stateObjectsDirty` 中。

`journal` 字段记录了 `StateDB` 对象的所有操作，以便用来进行回滚操作。需要注意的是，当 `stateObjects` 中的所有信息被写入到 `trie` 字段代表的 trie 树中后，`journal` 字段会被清空，无法再进行回滚了。

`db` 字段代表的是一个从数据库中访问 trie 对象的对象，比如 `OpenTrie`。`db` 是一个接口类型，但在实际的调用代码中，它只有一个实例就是 `cachingDB` 这个对象。这个对象是对 `trie.Database` 的一个包装，通过 `pastTries` 字段缓存了部分曾经访问过的 trie 对象。这样当下次再次访问这个 trie 时，就不需要从数据库中读取了。这个缓存的功能需要 `cachedTrie` 对象的配合，在将 trie 提交到数据库的同时通知 `cachingDB` 进行缓存。

`trie` 字段也是一个接口类型的字段，它代表了保存账户信息的 trie 对象。在实际调用代码中，它也只有一个实例，就是 `cachedTrie` 这个对象。这个对象是对 `trie.SecureTrie` 的一个封装，主要修改是改写了 `trie.SecureTrie` 的 Commit 方法：在新的 Commit 方法中，除了调用 `trie.SecureTrie` 的 Commit 方法外，还会与 `cachingDB` 配合，调用 `cachingDB.pushTrie` 将当前的 `trie.SecureTtrie` 缓存到 `cachingDB` 中。

可以看出代表账户余额模型的对象就是 `StateDB` 对象，它就像一个 KV 数据库，以账户地址作为 Key、以账户信息作为 Value 进行数据的存储和查询。其底层使用 trie 对象来实现这种类似 KV 结构的数据的存储。而在数据库层面，`StateDB` 又增加了一些缓存机制，使得运行时效率更高（但也使得代码更复杂一点）。

但不得不说，由于底层使用 trie 对象保存所有数据，`StateDB` 与 KV 数据库不同的是，在任意时刻或提交数据到数据库（Commit）后，可以得到 trie 对象的哈希值。在生成区块时，这个哈希值是保存在区块头的 `Root` 字段中的。如此一来，就能随时从数据库中读取任意区块的 state 信息了。

功能解析

了解了 state 模块的整体设计结构以后，其实代码是很容易读懂的。因此我不打算面面俱到的介绍 state 模块的所有功能。在这一节里，我选了几个比较重要的功能，进行简单的介绍。

保存账户信息

前面我们说过，`StateDB` 就像一个 KV 数据库，底层使用 trie 保存数据。那么到底 K 是什么、 V 是什么呢？

其实要了解 KV 分别是什么，从下面的代码就可以看出来：

```
func (self *StateDB) getStateObject(addr common.Address) (stateObject *stateObject) {
    ......
    enc, err := self.trie.TryGet(addr[:])
    if len(enc) == 0 {
        self.setError(err)
        return nil
    }
    var data Account
    if err := rlp.DecodeBytes(enc, &data); err != nil {
        return nil
    }
    ......
}
```

很明显，所谓的 Key 就是账户地址，Value 就是一个 `Account` 对象。它的定义如下：

```
type Account struct {
    Nonce    uint64
    Balance  *big.Int
    Root     common.Hash // merkle root of the storage trie
    CodeHash []byte
}
```

因此每个账户中，都包含了四个信息：代表账户操作编号的 Nonce，代表账户余额的 Balance，代表数据存储 trie 哈希的 Root，和代表合约代码哈希的 CodeHash。我们会在介绍以太坊合约时再介绍 Root 和 CodeHash 字段代表的含义；关于 Nonce 的意义，参看本篇文章中的「重放攻击」小节。

可以看到，所谓的余额模型真的非常简单，就是用 Balance 字段记录当前余额就可以了。在矿工生成一个新的区块时，会处理所有的交易和合约。如果涉及到A账户给B账户转账的操作，就从A的账户中减去交易数值，然后给B账户加上同一数值。当所有交易处理完成后，这些交易引起的 `StateDB` 的变化使其内容的 trie 对象生成了新的 root 哈希；矿工将这个啥希记录到区块的 header 中。这样当别人收到这个区块时，可以重复这一过程，如果得到的 `StateDB` 的哈希与区块头中记录的哈希一致，则交易验证成功，说明这个区块的交易是正确的。

重放攻击

我们知道，以太坊中的转账是通过一个个交易完成的。现在设想这样一个场景：A 给 B 发起了一笔转账交易，这次交易被所有矿工确认为合法的，因此转账成功；这时候 B 机灵一动，突然想到这笔交易既然之前被认定为合法的，那再次把这笔交易发给矿工，应该还是合法的吧？毕竟交易本身的数据没有变过。所以 B 把之前 A 发起的这笔交易找出来，又重新发到了网络上。这就是重放攻击。如果没有预防措施，那么 B 就可以用这个交易不断地把 A 的钱转给自己。

在以太坊中，防止重放攻击的任务正是由前面提到的 `Account` 结构中的 `Nonce` 字段完成的。具体来说，以太坊中每笔交易中都需要记录一个发起账户的 nonce 值：

```
type Transaction struct {
    data txdata
    ......
}

type txdata struct {
    AccountNonce uint64          `json:"nonce"    gencodec:"required"`
    ......
}
```

一笔交易就是由 `Transaction` 结构代表的，而 `txdata` 结构中的 `AccountNonce` 就是 `Account.Nonce` 的值。在构造一笔交易时，发起者需要将 `txdata.AccountNonce` 字段设置为发起账户的 `Account.Nonce` 值加 1。

在矿工出块进行验证时，会对 `Transaction` 中的 `AccountNonce` 值进行验证，如果这个值确实比发起者账户的 `Account.Nonce` 值大 1，则为有效的；否则这个交易目前是无效的（如果 `txdata.AccountNonce` 与 `Account.Nonce` 的差 > 1，说明这笔交易可能以后会生效，就暂时保留；如果这个差 < 1，则直接丢弃这个交易）。

除了验证 `Transaction` 与 `Account` 的 nonce 值，还需要在 `Transaction` 结构整体验证成功、转账完成后，将发起账户的 `Account.Nonce` 值加 1。这样才能在使用这笔交易发起重放攻击后，让这种攻击失效。

可以看到，`Account.Nonce` 主要功能就是用来避免重放攻击。但这需要代表交易的 `Transaction` 结构和矿工的配合，即 `Transaction` 中有 `AccountNonce` 字段记录着此次转账完成后账户的 `Account.Nonce` 值应该是多少；而矿工需要验证这个值，且在转账完成后修改账户的 `Account.Nonce` 值。

快照与回滚

在 `StateDB` 的实现中，还有快照与回滚的功能，这两个功能主要是由下面两个方法提供的：

```
func (self *StateDB) Snapshot() int {
    id := self.nextRevisionId
    self.nextRevisionId++
    self.validRevisions = append(self.validRevisions, revision{id, self.journal.length()})
    return id
}

func (self *StateDB) RevertToSnapshot(revid int) {
    // 根据快照 id，从 validRevisions 中查找快照信息
    idx := sort.Search(len(self.validRevisions), func(i int) bool {
        return self.validRevisions[i].id >= revid
    })
    if idx == len(self.validRevisions) || self.validRevisions[idx].id != revid {
        panic(fmt.Errorf("revision id %v cannot be reverted", revid))
    }
    snapshot := self.validRevisions[idx].journalIndex

    // 恢复快照
    self.journal.revert(self, snapshot)
    self.validRevisions = self.validRevisions[:idx]
}
```

`StateDB.Snapshot` 方法创建一个快照，返回一个 `int` 值作为快照的 ID。`StateDB.RevertToSnapshot` 用这个 ID 将 `StateDB` 的状态恢复到某一个快照状态。

这两个方法的实现都很简单，从中可以看出，`StateDB.nextRevisionId` 字段用来生成快照的有效 ID，而 `StateDB.validRevisions` 记录所有有效快照的信息。关键实现其实在 `StateDB.journal` 字段中，这个字段的类型是 `journal` 结构。我们详细看一下这个结构的实现。

`journal` 结构在 state/journal.go 中，它的定义如下：

```
type journal struct {
    entries []journalEntry         // Current changes tracked by the journal
    dirties map[common.Address]int // Dirty accounts and the number of changes
}
```

其中 `entries` 字段的类型是 `journalEntry` 类型的数组，`journalEntry` 是一个接口类型，主要方法就是用来恢复数据的 `revert` 方法，它代表了对某一操作进行回滚的操作，因此实现了 `journalEntry` 接口的对象有很多个，我把它们罗列在这里：

- `type createObjectChange struct`
- `type resetObjectChange struct`
- `type suicideChange struct`
- `type balanceChange struct`
- `type nonceChange struct`
- `type storageChange struct`
- `type codeChange struct`
- `type refundChange struct`
- `type addLogChange struct`
- `type addPreimageChange struct`
- `type touchChange struct`

可以看到这些代表具体回滚操作的对象，对应了所有对 `StateDB` 的操作。每当有对 `StateDB` 的操作时，就会构造一个对应的回滚操作并调用 `journal.append` 方法将其加入到 `journal.entries` 中。比如对于增加余额的操作：

```
func (self *StateDB) AddBalance(addr common.Address, amount *big.Int) {
    stateObject := self.GetOrNewStateObject(addr)
    if stateObject != nil {
        stateObject.AddBalance(amount)
    }
}

func (c *stateObject) AddBalance(amount *big.Int) {
    ......
    c.SetBalance(new(big.Int).Add(c.Balance(), amount))
}

func (self *stateObject) SetBalance(amount *big.Int) {
    // 构造 SetBalance 的回滚操作 balanceChange 并加其记录到 `journal.entries` 中
    self.db.journal.append(balanceChange{
        account: &self.address,
        prev:    new(big.Int).Set(self.data.Balance),
    })
    self.setBalance(amount)
}
```

`journal.append` 的实现很简单直接：

```
func (j *journal) append(entry journalEntry) {
    j.entries = append(j.entries, entry)
    if addr := entry.dirtied(); addr != nil {
        j.dirties[*addr]++
    }
}
```

这样，`journal.entries` 中积累了所有操作的回滚操作。当调用 `StateDB.RevertToSnapshot` 进行回滚操作时，就会调用 `journal.revert` 方法：

```
func (j *journal) revert(statedb *StateDB, snapshot int) {
    for i := len(j.entries) - 1; i >= snapshot; i-- {
        // Undo the changes made by the operation
        j.entries[i].revert(statedb)

        // Drop any dirty tracking induced by the change
        if addr := j.entries[i].dirtied(); addr != nil {
            if j.dirties[*addr]--; j.dirties[*addr] == 0 {
                delete(j.dirties, *addr)
            }
        }
```

```
    }
    j.entries = j.entries[:snapshot]
}
```

在 `journal.revert` 中，会从 `journal.entries` 中最后一项开始、向前至参数中指定的项，调用它们的 revert 方法。我们以 `balanceChange` 为例看看这些回滚对象是如何操作的。刚才提到过在修改 balance 的 `stateObject.SetBalance` 中会构造一个 `balanceChange` 对象：

```
func (self *stateObject) SetBalance(amount *big.Int) {
    // 构造 SetBalance 的回滚操作 balanceChange 并加其记录到 `journal.entries` 中
    self.db.journal.append(balanceChange{
        account: &self.address,
        prev:    new(big.Int).Set(self.data.Balance),
    })
    self.setBalance(amount)
}
```

其中 `balanceChange.prev` 字段保存了修改之前的 balance 值。那么在 `balanceChange.revert` 中就将这个值重新恢复到账户信息中就行了：

```
func (ch balanceChange) revert(s *StateDB) {
    s.getStateObject(*ch.account).setBalance(ch.prev)
}
```

注意这里调用的是 `stateObject.setBalance` 而不是 `stateObject.SetBalance`，后者会再次将修改加入到 `journal` 中，这并不是我们想要的操作。

现在我们可以总结一下 state 模块是如何实现快照和回滚功能的：

1. 将所有可能的修改作一个统计。
2. 实现所有可能操作对应的回滚操作对象。
3. 在每次进行操作前，将对应的回滚对象加入到回滚操作的数组中，例如 `journal.entries`。
4. 要在当前状态下创建一个快照，就记录下当前 `journal.entries` 的长度（因为 `journal.entries` 中一个数组）。
5. 要恢复某个快照（即实现回滚操作），就从 `journal.entries` 中最后一项开始，向前至指定的快照索引，逐一调用这些对象的 revert 操作。

其实还是挺简单的，我们日常开发中要实现类似的功能，也可以参考这个实现方式。

要注意一点的是快照与回滚只能针对还未提交到数据库中的账户信息，即存在于 `stateObject` 中的信息。如果已经被提交到数据库中，就无法回滚了。（其实要想实现也是可以的，只是以太坊的代码没有这么实现而已）

总结

在这篇文章里，我们介绍了以太坊的 state 模块。 state 模块实现了以太坊的账户余额模型，它以一种类似 KV 数据库的形式存储账户信息，其中以账户地址作为 Key，以账户信息（`Account` 结构）作为 Value。它的底层使用 trie 对象对数据进行存储，这样不但可以快速获取某一账户的信息，还可以得到 trie 的 root 哈希保存到区块头中，这样矿工生成区块后，别的节点就可以重现交易对账户的修改，并将最终的保存账户信息的 trie 的 root 哈希与区块头中保存的哈希进行比较，方便对区块进行验证。

以上就是对 state 模块的所有分析。水平有限，如果有错误还请留言或邮件指出，非常感谢。

Similar Posts

- 初识联盟链1: Fabric 是什么
- PBFT代码篇：fabric 中的 PBFT 实现
- 实用拜占庭容错算法（PBFT）
- 拜占庭将军问题

- 数据结构与算法：B树
- 以太坊源码解析：evm

上一篇 以太坊源码解析：区块同步-fetcher

下一篇 以太坊源码解析：rpc

Comments

Comments (5)

Login

Sort by: **Date** Rating Last Activity

fuchengshun · *134 weeks ago* +1

写的真好，每篇都看了

Reply **1 reply** · *active 132 weeks ago*

yangzhe · *132 weeks ago* +1

哈哈 谢谢鼓励！你是第一个留言的 ^_^

Reply

test · *132 weeks ago* +1

test

Reply **1 reply** · *active 131 weeks ago*

fatcat 15p · *131 weeks ago* +2

reply test

Reply

binbinbin · *123 weeks ago* +1

写的太好了

Reply

Post a new comment

Enter text right here!

Comment as a Guest, or login:

Name

Displayed next to your comments.

Email

Not displayed publicly.

Website (optional)

If you have a website, link to it here.

Subscribe to None

Submit Comment

Contact me at:

Site powered by Jekyll & Github Pages. Theme designed by HyG.