StateDB和Trie （上）

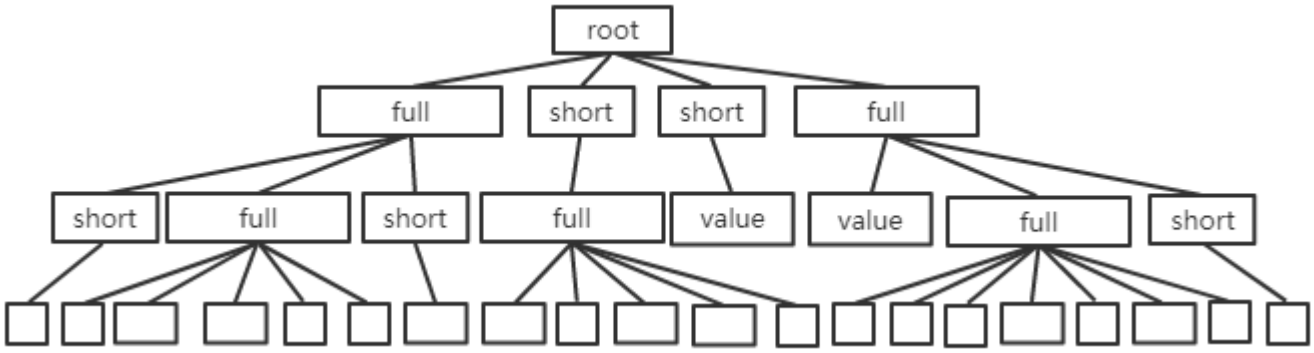
在以太坊中，所有和账户相关的状态信息都是通过 StateDB 来存储和获取的。StateDB 作为表层和其他逻辑模块交互，在 StateDB 之后使用 Merkle Patricia Trie (MPT) 结构来构建编码后的 state 关系，用于快速索引以及回滚等操作。MPT 中的所有节点最后都会以 `key - value` 的形式存入磁盘数据库。

这篇文章将重点介绍 StateDB 和 Trie 在以太坊中的实现以及两者在账户状态存储流程中所扮演的角色。

Merkle Patricia Trie

MPT 是结合了 Merkle Tree 和 Patricia Tree 的特点后创建的树形数据结构。其包含了如下的一些特点：

- 能存储任意长度的键值对数据。
- 支持 Merkle Proof，用于节点的快速校验。
- 能快速的查询 key 所对应的 value 数据。



在以太坊中，MPT被定义为四种不同类型的节点：`fullNode, shortNode, valueNode, hashNode`：

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

`valueNode` 存储具体的 value 数据，它的 key 是从 root 到此节点的路径上所有 key 的总和。

`hashNode` 存储一个数据库中其他节点的哈希用作索引。

`shortNode` 是 MPT 的枝干节点之一。"Key" 字段存储当前 shortNode 之后所有 node 共同的一段前缀 key。"Val" 字段存储一个后续的节点。如果从根节点到当前节点所组成的 key 前缀已经键值对结构中的"键"完全吻合，且没有其它符合此前缀的键值对存在，则后续节点为一个 valueNode。如果满足此前缀 key 的键值对组合多于一个，则后续存储一个 fullNode。

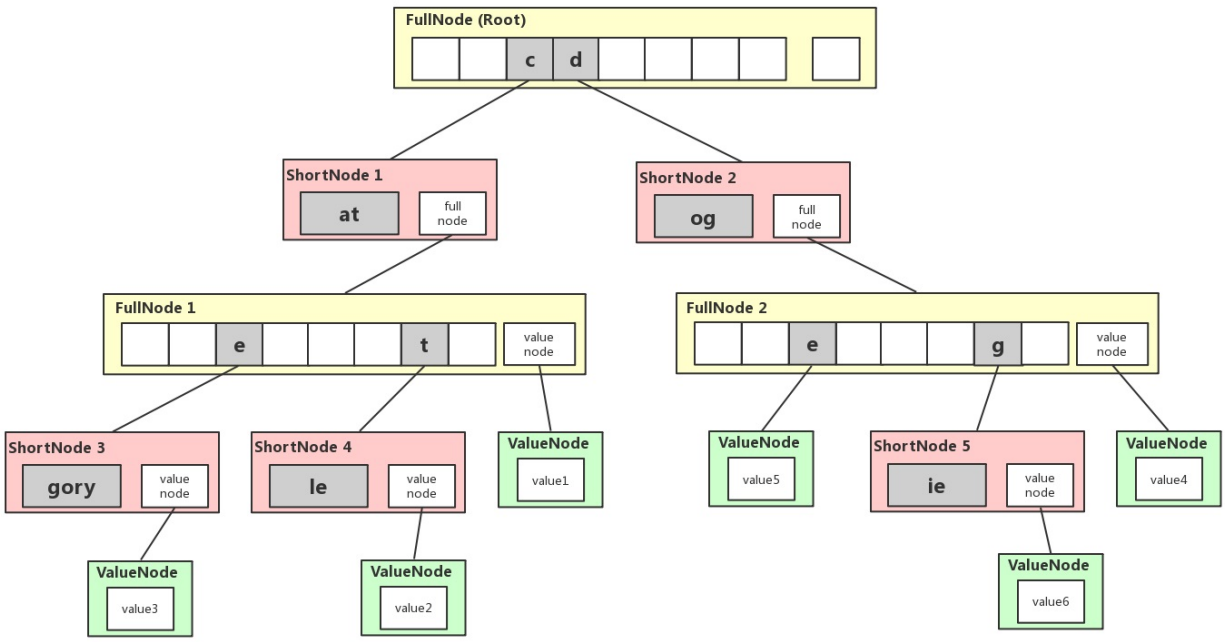
`fullNode` 是 MPT 的枝干节点之一。和 shortNode 不同的是，fullNode 没有 Key 字段，只有一个 node 数组 "Children"。fullNode 主要作用是存储有共同前缀 key 但是在后续key值产生分歧的所有键值对。Children 数组中的每一个元素都表示一个前缀符号（具体可以参考下面的例子），用不同的前缀符号来分隔不同的键值对。以太坊中 Children 数组的长度为 17，因为涉及具体的编码方式，所以不在这里展开讲为什么这么设置。可以简单将 17 理解为 16 + 1，16 进制加上本身的 value 值。

具体例子

光看字面比较抽象，我们看看具体的例子。假设现在我们有 6 个键值对：

key	value
cat	value1
cattle	value2
category	value3
dog	value4
doge	value5
doggie	value6

他们在 MPT 中就会以如下的方式存储：



可以看到，因为存在前缀分歧，所以 Root 节点是 fullNode。后续节点分成两派，key 以 `c` 为前缀和以 `d` 为前缀。我们关注以 `d` 为前缀的三个键值对，它们的 key 分别是 "dog"、"doge" 和 "doggie"。除了开头的 `d`

以外，它们还有个共同的前缀 `og`。因此 Root 节点往下引申一个 ShortNode 2。ShortNode 2 的 key 就是 "og", 又因为除了 og 其它部分都存在分歧，所以 ShortNode 2 的 Val 字段存储的是一个 FullNode（参考上面对 shortNode 的解释）。

上面提到过，fullNode 存在17个元素，可以简单理解为 16 进制加上本身 value 值构成 17 个元素。在我们的例子中，此处的 FullNode 2 已经构成了 `dog` 这个前缀 key（根节点的 d 加上后续 shortNode 的 og），已经完全吻合 `dog - value4` 这个键值对中的 key，所以 FullNode 2 的末尾便是一个 ValueNode，ValueNode 的值是 value4。

FullNode 2 的前缀 key 再加上其内部的 `e` 能够构成前缀 key "doge"，所以 `e` 位置下面直接引申一个 ValueNode，其值便是 `doge - value5` 这键值对内的 value5。

FullNode 2 的前缀 key 加上其内部的 `g` 组成 key 前缀 `dogg`。剩下的符合 `dogg` 前缀的只有 `doggie - value6`。所以 `g` 下面引申一个 ShortNode 5，其节点的 Key 字段为 `ie`，Val 字段为一个值为 value6 的 ValueNode。

另外三个前缀为 `c` 的键值对则同理可以得到图中的结构。

MPT 的编码

上面的例子中所有的数据都是没有经过编码的，这会造成什么问题呢？之前提过 FullNode 的每个 Children 元素都代表一个前缀符号，如果不对key进行编码则很难将这个前缀符号规范化，使得前缀的种类过多无法确定 Children 数组的长度。为了解决这个问题，以太坊使用 Hex 编码对所有键值对的 key 进行编码。编码后所有的 key 就都由 `[0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f]` 这 16 个十六进制符号组成。编码后，FullNode 的 Children 就可以分割成 16 份，每份对应各自的前缀符号，这也是为什么以太坊 FullNode 的 Children 数组长度是 17（16 个子节点 + 本身节点）。

举个例子：假设键值对 `doggie - value6` 经过编码后变为 `0x646f67676965 - value6`，那么在 Root 节点中这个键值对就会被分到 `6` 这个子节点下面，也就是 Children 数组的第七个元素中。

StateDB

StateDB 作为账户状态的更新以及查询入口，基于具体逻辑的方法调用。比如账户余额的更新，nonce 的查询等。同时，它还肩负着所有合约数据的存储查询。为了支持数据的快速查询以及区块的回滚操作，StateDB 使用 MPT 结构作为其下层的存储方式。

先来看看 StateDB 的数据结构：

```
type StateDB struct {
    db   Database
    trie Trie

    // This map holds 'live' objects, which will get modified while processing a
state transition.
    stateObjects      map[common.Address]*stateObject
    stateObjectsDirty map[common.Address]struct{}

    // 其余字段省去
    ...
}
```

- db - 用于连接下层 trie 数据库的字段。本身不存储数据，为了调取 TrieDB 存在。
- trie - 当前所有账户信息构建的 MPT 结构。
- stateObjects - 存储缓存的账户 state 信息。
- stateObjectsDirty - 标记被更改的 state 对象，用于后续的 commit 操作。

StateDB 通过操作和查询 `stateObjects` 中缓存的 state 对象来完成业务逻辑的执行。如果 stateObjects 中找不到需要操作的对象，则通过 `createObject(addr common.Address)` 方法从 `trie` 字段的 MPT 中读取对应的 state 对象并放入缓存中。

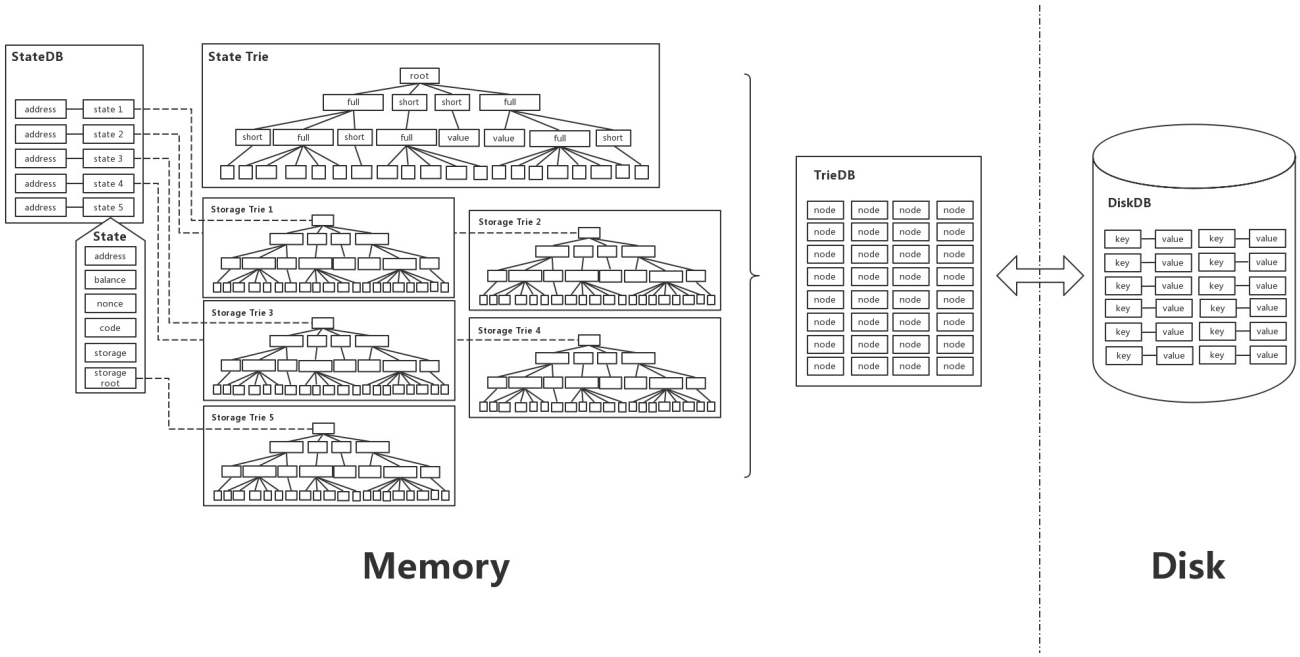
stateObject

stateObject 是以太坊中用于存储每个账户信息的数据结构：

```
type stateObject struct {
    address  common.Address
    addrHash common.Hash // hash of ethereum address of the account
    data     Account
    db       *StateDB

    // Write caches.
    trie Trie // storage trie, which becomes non-nil on first access
    code Code // contract bytecode, which gets set when code is loaded

    originStorage Storage // Storage cache of original entries to dedup rewrites
    dirtyStorage  Storage // Storage entries that need to be flushed to disk

    // Cache flags.
    // When an object is marked suicided it will be delete from the trie
    // during the "update" phase of the state transition.
    dirtyCode bool // true if the code was updated
    suicided  bool
    deleted   bool
}

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

data 字段保存账户的余额，Nonce等信息。同时，在后续落盘 MPT 的过程中，主要存储的内容就是经过编码序列化后的 data 字段。

这里值得特别提一点的是 stateObject 的 trie 字段和 data 中的 Root 字段。和 StateDB 中的 trie 不同，此处的 trie 是用来存储此 state 地址下的合约数据的。每个地址账户都会有属于自己的一棵 trie 用来做合约存储。data 中的 Root 则是这个 trie 的根节点的哈希。在将 state 数据存入 StateDB 的 trie 的过程中会带上这个 Root，这么做的好处主要是能将合约数据和世界状态数据绑定在一起，增加关联性和安全性，同时，在后续的回滚操作中能通过世界状态 trie 的 Root 还原出包括合约数据的所有状态。



由于篇幅有限，上篇就先只介绍 MPT 和 StateDB 本身。在下篇，我们将重点结合上图讲解 StateDB 和 MPT 两者的工作结构，以及不同类型 Transaction 执行过程中 StateDB 和 MPT 的逻辑流程。

（完）