Account 상태 조회

이더리움에서 Account 정보는 다음과 같은 구조체를 갖습니다.

```
// package : core/state
type Account struct {
    Nonce    uint64      // 트랜잭션 수, 0으로 시작
    Balance  *big.Int    // 이더 잔고(wei)
    Root     common.Hash // account가 저장될 머클 패트리시아 트리의 루트 노드
    CodeHash []byte      // contract bytecode hash
}
```

이러한 Account 의 정보들을 이더리움에서는 상태(state) 라고 하고
이를 stateObject 구조체로 표현합니다.

```
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
  ...
}

// type
const (
    HashLength    = 32
    AddressLength = 20
)
// package : common
type Address [AddressLength]byte
type Hash [HashLength]byte
```

이더리움에서 데이터들을 트리 자료구조를 사용해 관리를 하고 있습니다.
stateObject 역시 trie를 통해 접근할 수 있고, 상태를 변경한 Account는
CommitTrie() 함수를 호출해서 Trie를 업데이트하고 levelDB에 저장합니다.

getBalance (in go-ethereum)

`getBalance` 호출을 통해서 어떤 식으로 trie에 접근하는지 조금 더 살펴보겠습니다.

이더리움에서 balance 를 조회는 다음과 같은 순서로 진행합니다.

- blockNumber 를 통해 blockHeader 정보 조회
- blockHeader 정보에서 state trie의 root 정보 조회
- root와 leveldb 정보를 가지고 trie 구성
- 조회하려는 Account의 Address 를 key 로 trie에 저장된 value 조회
- value(rlp인코딩) 를 decoding 해서 Account 구조체로 변경
- stateObject 로 변환
- balance 조회

코드를 조금씩 살펴보면 다음과 같습니다.

```
// blockNumber 가 없으면 'latest'
// state 에서 balance 를 조회
// pakcage : ethapi
func (s *PublicBlockChainAPI) GetBalance(ctx context.Context, address common.Addres
    state, _, err := s.b.StateAndHeaderByNumber(ctx, blockNr)
    if state == nil || err != nil {
        return nil, err
    }
    b := state.GetBalance(address)
    return b, state.Error()
}

// StateAndHeaderByNumber 상태 trie 호출
// package : eth
func (b *EthApiBackend) StateAndHeaderByNumber(ctx context.Context, blockNr rpc.Blo
    // Pending state is only known by the miner
    if blockNr == rpc.PendingBlockNumber {
        block, state := b.eth.miner.Pending()
        return state, block.Header(), nil
    }
    // Otherwise resolve the block number and return its state
    header, err := b.HeaderByNumber(ctx, blockNr)
    if header == nil || err != nil {
        return nil, nil, err
    }
    stateDb, err := b.eth.BlockChain().StateAt(header.Root)
    return stateDb, header, err
}

// b.eth.BlockChain().StateAt 이라는 부분은 이 부분을 호출
// package : state
func New(root common.Hash, db Database) (*StateDB, error) {
    tr, err := db.OpenTrie(root)
    if err != nil {
        return nil, err
    }
    return &StateDB{
        db:                db,
        trie:              tr,
```

```
        stateObjects:      make(map[common.Address]*stateObject),
        stateObjectsDirty: make(map[common.Address]struct{}),
        logs:              make(map[common.Hash][]*types.Log),
        preimages:         make(map[common.Hash][]byte),
        journal:           newJournal(),
    }, nil
}

// state.getBalance
// package : state
func (self *StateDB) GetBalance(addr common.Address) *big.Int {
    stateObject := self.getStateObject(addr)
    if stateObject != nil {
        return stateObject.Balance()
    }
    return common.Big0
}

// package : state
func (self *StateDB) getStateObject(addr common.Address) (stateObject *stateObject)
    // Prefer 'live' objects.
    if obj := self.stateObjects[addr]; obj != nil {
        if obj.deleted {
            return nil
        }
        return obj
    }

    // Load the object from the database.
    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 {
        log.Error("Failed to decode state object", "addr", addr, "err", err)
        return nil
    }
    // Insert into the live set.
    obj := newObject(self, addr, data)
    self.setStateObject(obj)
    return obj
}

// newObject
func newObject(db *StateDB, address common.Address, data Account) *stateObject {
    if data.Balance == nil {
        data.Balance = new(big.Int)
    }
    if data.CodeHash == nil {
        data.CodeHash = emptyCodeHash
    }
    return &stateObject{
        db:            db,
        address:       address,
```

```
        addrHash:      crypto.Keccak256Hash(address[:]),
        data:          data,
        cachedStorage: make(Storage),
        dirtyStorage:  make(Storage),
    }
}
```

직접 db에 접근해 account 조회해보기

Javascript(Node)에서 이더리움의 db 파일에 접근해 직접 Account를 조회해보겠습니다.

준비

geth 를 실행해 database file을 얻습니다.
javascript 에서 account 정보를 얻기 위해서 필요한 라이브러리들을 설치합니다.

- merkle-patricia-tree
- rlp
- levelup
- leveldown
- ethereumjs-block
- ethereumjs-account

조회

다음의 순서로 Account 를 조회 합니다.

- blockNumber 변환 (prefix, suffix)
- blockHeader 조회
- stateRoot 조회
- trie 구성
- Account 조회

구현하면 대략 다음과 같이 됩니다.

```
var path = require('path');
var levelup = require('levelup');
var leveldown = require('leveldown');
var ethBlock = require('ethereumjs-block');
```

```
var Account = require('ethereumjs-account');
var utils = require('./libs/utils');
var rlp = require('rlp');

var levelDBPath = path.relative('./', '/Users/amuyu/geth/chaindata');
var db = levelup(leveldown(levelDBPath));

const prefix = utils.stringToHex('h');
const suffix = utils.stringToHex('n');

// blockNumber Key String
var blockNumber = 5156;
var hexBlockNumber = utils.padLeft(utils.decimalToHex(blockNumber), 16);
var keyString = prefix + hexBlockNumber + suffix;
var key = new Buffer(keyString, 'hex');

console.log('Block Number:', key.toString('hex'));

// DB 조회(BlockHeader)
db.get(key, function (er, value) {
  if (er) throw new Error(er);

  console.log('Block Hash:', value.toString('hex'));
  value = value.toString('hex');
  var keyString = prefix + hexBlockNumber + value;
  var key = new Buffer(keyString, 'hex');

  db.get(key, function (er, value) {
    if (er) throw new Error(er);
    // console.log('Raw Block Data:', value.toString('hex'));

        // BlockHeader
    var block = new ethBlock.Header(value);
    var stateRoot = block.stateRoot;
    console.log('State Root:', stateRoot.toString('hex'));

    var root = new Buffer(stateRoot, 'hex');
    var address = new Buffer(address, 'hex');
    var trie = new merklePatriciaTree(db, root);

        // address sha3
    var address = '0x4CE9792D5259194D40F21243F85A19c7e67Fc667';
    var hash = utils.sha3(address).toString('hex');
    // console.log('Hash key:', hash, hash.length);
    var keyAddress = utils.getNaked(hash);    // '0x' 제거
    console.log('keyAddress:', keyAddress);

    trie.get(keyAddress, function (er, re) {
        if (er) throw new Error(er);

        var decoded = rlp.decode(re);

        // type Account struct {
        //     Nonce    uint64      // 트랜잭션 수, 0으로 시작
        //     Balance  *big.Int    // 이더 잔고(wei)
        //     Root     common.Hash // account가 저장될 머클 패트리시아 트리의 루트 노드
```

```
        //     CodeHash []byte      // contract bytecode hash
        // }
        console.log('decoded[1]', decoded[1])
        var balance = new BigNumber(decoded[1].toString('hex'), 16);
        console.log('Address balance:', balance.toString(), 'eather');
        console.log('Root:', decoded[2].toString('hex'));

        var account = new Account(re);
        console.log('account stateRoot:', account.stateRoot.toString('hex'));
        console.log('account code:', account.codeHash.toString('hex'));

        return console.log('Address Data:', decoded);
    });
  });
});
```

실행하면 다음과 같은 결과를 확인할 수 있습니다.

```
Block Number: 6800000000000014246e
Block Hash: 5afe3230b5f14147fe51195f6b9b01342d66c8d04f65ff008dbc6e1df8cf21ed
State Root: c452aaf84477518effd67ec1af60d214225cbff9dda74ff649cb13cb698f3a77
keyAddress: a4dfab5f4fe8d4f5fe6d6175e272391641d8054a188f30a314930411e0251209
// balance
decoded[1] <Buffer 04 e0 fe e3 5e dd 6e 60 00>
Address balance: 89999622000000000000 eather
// root
Root: 56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421
account stateRoot: 56e81f171bcc55a6ff8345e692c0f86e5b48e01b996cadc001622fb5e363b421
// code
account code: c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfad8045d85a470
```

참고로 trie 에서 address 를 key 로 조회하고 값을 디코딩 후, 출력 결과는 다음과 같습니다.

```
//   <Buffer 01>,
//   <Buffer 04 e0 fe e3 5e dd 6e 60 00>,
//   <Buffer 56 e8 1f 17 1b cc 55 a6 ff 83 45 e6 92 c0 f8 6e 5b 48 e0 1b 99 6c ad c
//   <Buffer c5 d2 46 01 86 f7 23 3c 92 7e 7d b2 dc c7 03 c0 e5 00 b6 53 ca 82 27 3

// exports.KECCAK256_NULL_S = 'c5d2460186f7233c927e7db2dcc703c0e500b653ca82273b7bfa
```

참고

blockChain

Data structure in Ethereum. Episode 4: Diving into examples.