状态(state)

LevelDB特点：

1. 键和值都是任意的字节数组
2. 数据根据键排序，排序规则可重载
3. 有三种基本的操作：Put(key,value)， Get(key)， Delete(key)
4. 支持多个操作组合的原子操作
5. 用户可以创建临时快照，得到一个一致的数据视图
6. 支持向前和向后的数据迭代
7. 数据用Snappy压缩库自动压缩
8. 外部操作（如文件系统操作等）通过一个虚拟接口使用，用户可以对操作系统进行定制相应操作

在Ethereum的世界里，数据的最终存储形式是[k,v]键值对，目前使用的[k,v]型底层数据库是LevelDB；所有与交易，操作相关的数据，其呈现的集合形式是Block(Header)；如果以Block为单位链接起来，则构成更大粒度的BlockChain(HeaderChain)；若以Block作切割，那么Transaction和Contract就是更小的粒度；所有交易或操作的结果，将以各个个体账户的状态(state)存在，账户的呈现形式是stateObject，所有账户的集合受StateDB管理。

成员分散存储在底层数据库

Header和Block的主要成员变量，最终还是要存储在底层数据库中。Ethereum 选用的是LevelDB,属于非关系型数据库，存储单元是[k,v]键值对。我们来看看具体的存储方式(core/database_util.go)

key	value
'h' + num + hash	header's RLP raw data
'h' + num + hash + 't'	td
'h' + num + 'n'	hash
'H' + hash	num
'b' + num + hash	body's RLP raw data
'r' + num + hash	receipts RLP
'l' + hash	tx/receipt lookup metadata

这里的hash就是该Block(或Header)对象的RLP哈希值，在代码中也被称为canonical hash；num是Number的uint64类型,大端(big endian)整型数。可以发现，num和hash是key中出现最多的成分；同时num和hash还分别作为value被单独存储，而每当此时则另一方必组成key。这些信息都在强烈的暗示，num(Number)和hash是Block最为重要的两个属性：num用来确定Block在整个区块链中所处的位置，hash用来辨识惟一的Block/Header对象。

通过以上的设计，Block结构体的所有重要成员，都被存储进了底层数据库。当所有Block对象的信息都已经写进数据库后，我们就可以使用BlockChain结构体来处理整个块链。

区块链的操作

从逻辑上讲，既然BlockChain和HeaderChain都管理着一个类似单向链表的结构，那么它们提供的操作方法肯定包括插入，删除，和查找。

查找比较简单，以BlockChain为例，它有一个成员currentBlock，指向当前最新的Block，而HeaderChain也有一个类似的成员currentHeader。除此之外，底层数据库里还分别存有当前最新Block和Header的canonical hash：

key	value
"LastHeader"	hash
"LastBlock"	hash
"LastFast"	hash

这里“LastFast”所存储的是在一种特别的同步方式FastSync下，最新Block的canonical hash。FastSync相比于FullSync，可以仅仅同步Header而不考虑Body，故此得名Fast。

以BlockChain为例，通过"LastBlock"为key从数据库中获取最新的Block之后，用num逐一遍历，得到目标Block的num后，用'h'+num+'n'作key，就可以从数据库中获取目标canonical hash。

插入和删除。区块链跟普通单向链表有一点非常明显的不同，在于Header的前向指针ParentHash是不能修改的，即当前区块的父区块是不能修改的。所以在插入的实现中，当决定写入一个新的Header进底层数据库时，从这个Header开始回溯，要保证它的parent，以及parent的parent等等，都已经写入数据库了。只有这样，才能确保从创世块(num为0)起始，直到当前新写入的区块，整个链式结构是完整的，没有中断或分叉。删除的情形也类似，要从num最大的区块开始，逐步回溯。在BlockChain的操作里，删除一般是伴随着插入出现的，即当需要插入新区块时，才可能有旧的区块需要被删除，这种情形在代码里被称为reorg。

stateObject

以太坊的全局"共享状态"是有很多小对象(账户)来组成的，每个账户都有一个与之关联的状态(state)和一个20字节的地址(address)。

账户状态有四个组成部分：

nonce：如果账户是一个外部账户，nonce代表从此账户地址发送的交易序号。如果账户是一个合约账户，nonce表示此账户创建的合约序号；

balance：此账户拥有的Wei的数量。1Ether=10^18Wei；

storageRoot：Merkle Patricia树的根节点Hash值。Merkle树会将此账户存储内容的hash值进行编码，默认是空值；

codeHash：此账户EVM代码的hash值。对于合约账户，就是被Hash的代码作为codeHash保存。对于外部账户，codeHash域是一个空字符串的Hash值。

stateObject定义了一种类型名为storage的map结构，用来存放[]Hash,Hash]类型的数据对，也就是State数据。当SetState()调用发生时，storage内部State数据被更新，相应标示为"dirty"。之后，待有需要时(比如updateRoot()调用)，那些标为"dirty"的State数据被一起写入storage trie，而storage trie中的所有内容在CommitTo()调用时再一起提交到底层数据库。

对合约来说，可以认为storage是一个很大的数组，全部初始化为空值，数组中的每个值宽度为32-byte，一共有2^256个，智能合约能任意读也能任意写数组中任一坐标

以如下合约为例

```
contract StorageTest {
    uint256 a;
    uint256[2] b;

    struct Entry {
        uint256 id;
        uint256 value;
    }
    Entry c;
}
```

In the above code:

- a存储在slot0.
- b存储在 slots 1, 和 slot 2
- c slots 3, slot 4

动态数组是这样保存的

```
contract StorageTest {
    uint256 a;      // slot 0
    uint256[2] b;  // slots 1-2

    struct Entry {
        uint256 id;
        uint256 value;
    }
    Entry c;       // slots 3-4
    Entry[] d;
}
```

slot5保存的是d的长度，而d的2个元素是保存在slot数组的hash(5)开始的2个slot中

map的保存

```
contract StorageTest {
    uint256 a;      // slot 0
    uint256[2] b;  // slots 1-2

    struct Entry {
        uint256 id;
        uint256 value;
    }
    Entry c;       // slots 3-4
    Entry[] d;     // slot 5 for length, keccak256(5)+ for data

    mapping(uint256 => uint256) e;
}
```

上例中代表e的slot为空，因为map类型并没有长度需要存储，而map中的数据则是存储在hash(map的key值，6）这个slot里

以下方法可以直接从智能合约里取map的值

```
function mapLocation(uint256 slot, uint256 key) public pure returns (uint256) {
    return uint256(keccak256(key, slot));
}
```

以太坊节点的状态更新策略

以太坊新装节点有3种同步策略，full、fast（默认）、light

full是应用已同步到最新区块的节点又收到新区块的时候的全校验策略来处理从1开始的每个区块

区块确认算法如下：

1. 检查区块引用的上一个区块是否存在和有效。
2. 检查区块的时间戳是否比引用的上一个区块大，而且小于15分钟。
3. 检查区块序号、难度值、 交易根，叔根和燃料限额（许多以太坊特有的底层概念）是否有效。
4. 检查区块的工作量证明是否有效。
5. 将S[0]赋值为上一个区块的STATE_ROOT。
6. 将TX赋值为区块的交易列表，一共有n笔交易。对于属于0……n-1的i，进行状态转换S[i+1] = APPLY(S[i],TX[i])。如果任何一个转换发生错误，或者程序执行到此处所花费的燃料（gas）超过了GASLIMIT，返回错误。
7. 用S[n]给S_FINAL赋值，向矿工支付区块奖励。
8. 检查S-FINAL是否与STATE_ROOT相同。如果相同，区块是有效的。否则，区块是无效的。

这一确认方法乍看起来似乎效率很低，因为它需要存储每个区块的所有状态，但是事实上以太坊的确认效率可以与比特币相提并论。原因是状态存储在树结构中（tree structure），每增加一个区块只需要改变树结构的一小部分。因此，一般而言，两个相邻的区块的树结构的大部分应该是相同的，因此存储一次数据，可以利用指针（即子树哈希）引用两次。一种被称为“帕特里夏树”（“Patricia Tree”）的树结构可以实现这一点，其中包括了对默克尔树概念的修改，不仅允许改变节点，而且还可以插入和删除节点。另外，因为所有的状态

信息是最后一个区块的一部分，所以没有必要存储全部的区块历史-这一方法如果能够可以应用到比特币系统中，经计算可以对存储空间有10-20倍的节省。

fast会分别从已连接节点下载block headers、block receipts、state entries到较近区块而不进行校验

```
INFO [03-27|16:42:41] Block synchronisation started
INFO [03-27|16:42:44] Imported new state entries              count=237 elapsed=8.644µs processed=237 pending=3793 retry=2 duplicate=0 unexpected=0
INFO [03-27|16:42:44] Imported new state entries              count=384 elapsed=6.016µs processed=621 pending=9937 retry=2 duplicate=0 unexpected=0
INFO [03-27|16:42:45] Imported new block headers              count=384 elapsed=2.077s  number=384 hash=d3d5d5…c79cf3 ignored=0
INFO [03-27|16:42:45] Imported new block receipts             count=2   elapsed=297.974µs number=2   hash=b495a1…4698c9 size=8.00B  ignored=0
INFO [03-27|16:42:45] Imported new block headers              count=384 elapsed=68.475ms  number=768 hash=5cfe57…c19c3a ignored=0
INFO [03-27|16:42:45] Imported new block headers              count=1152 elapsed=258.730ms number=1920 hash=80013e…3549c2 ignored=0
INFO [03-27|16:42:45] Imported new block receipts             count=4    elapsed=207.334µs number=6    hash=1f1aed…6b326e size=1.10kB ignored=0
INFO [03-27|16:42:45] Imported new block receipts             count=26   elapsed=1.418ms   number=32   hash=88be69…60ae13 size=1.18kB ignored=0
INFO [03-27|16:42:45] Imported new state entries              count=384  elapsed=7.484µs   processed=1005 pending=15141 retry=2 duplicate=0 unexpected=0
```

在控制台输入eth.syncing可以查看区块和states的同步状态

```
> eth.syncing
{
  currentBlock: 5334707,
  highestBlock: 5334772,
  knownStates: 94805206,
  pulledStates: 94792947,
  startingBlock: 5334580
}
```

light只下载区块头，当从本节点提交交易时，实际上是本节点自动到已连接的全节点上进行校验后再广播的