区块和交易等数据最终都是存储在 *leveldb* 数据库中的，本文介绍区块和交易在 *leveldb* 中的存储格式。

在 `core/database_util.go` 中封装了所有与区块存储和读取相关的代码，通过这些代码可以弄清楚区块、交易等数据结构在数据库中是如何存储的。

# 区块存储

*leveldb* 是一个 *key-value* 数据库，所有数据都是以键-值对的形式存储。*key* 一般与 *hash* 相关，*value* 一般是要存储的数据结构的 *RLP* 编码。区块存储时将区块头和区块体分开存储。

**区块头的存储格式为：**

```
headerPrefix + num (uint64 big endian) + hash -> rlpEncode(header)
```

其中 *key* 由区块头前缀、区块号（*uint64* 大端格式）、区块 *hash* 构成，*value* 是区块头的 *RLP* 编码。

**区块体的存储格式为：**

```
bodyPrefix + num (uint64 big endian) + hash -> rlpEncode(block body)
```

其中 *key* 由区块体前缀、区块号（*uint64* 大端格式）、区块 *hash* 构成，*value* 是区块体的 *RLP* 编码。

*key* 中的前缀可以用来区分数据的类型，在 `core/database_util.go` 中定义了各种前缀：

```
headerPrefix       = []byte("h")   // headerPrefix + num (uint64 big endian) + hash -> header

tdSuffix           = []byte("t")   // headerPrefix + num (uint64 big endian) + hash + tdSuffix -> td

numSuffix          = []byte("n")   // headerPrefix + num (uint64 big endian) + numSuffix -> hash

blockHashPrefix    = []byte("H")   // blockHashPrefix + hash -> num (uint64 big endian)

bodyPrefix         = []byte("b")   // bodyPrefix + num (uint64 big endian) + hash -> block body
```

其中 `headerPrefix` 定义了区块头 *key* 的前缀为 `h`，`bodyPrefix` 定义了区块体 *key* 的前缀为 `b`。

下面是存储区块头的函数：

```
// WriteHeader serializes a block header into the database.

func WriteHeader(db ethdb.Database, header *types.Header) error {
```

```
            data, err := rlp.EncodeToBytes(header)
            if err != nil {
                        return err
            }
            hash := header.Hash().Bytes()
            num := header.Number.Uint64()
            encNum := encodeBlockNumber(num)
            key := append(blockHashPrefix, hash...)
            if err := db.Put(key, encNum); err != nil {
                        glog.Fatalf("failed to store hash to number mapping into database: %v", err)
            }
            key = append(append(headerPrefix, encNum...), hash...)
            if err := db.Put(key, data); err != nil {
                        glog.Fatalf("failed to store header into database: %v", err)
            }
            glog.V(logger.Debug).Infof("stored header #%v [%x…]", header.Number, hash[:4])
            return nil}
```

它是先对区块头进行 *RLP* 编码，`encodeBlockNumber`将区块号转换成大端格式，然后组装 *key*。这里先向数据库中存储一条 `区块hash->区块号` 的记录，最后将区块头的 *RLP* 编码写到数据库中。

下面是存储区块体的函数：

```
// WriteBody serializes the body of a block into the database.
func WriteBody(db ethdb.Database, hash common.Hash, number uint64, body *types.Body) error {
            data, err := rlp.EncodeToBytes(body)
            if err != nil {
                        return err
```

```
            }

            return WriteBodyRLP(db, hash, number, data)}

// WriteBodyRLP writes a serialized body of a block into the database.

func WriteBodyRLP(db ethdb.Database, hash common.Hash, number uint64, rlp rlp.RawValue) error {

            key := append(append(bodyPrefix, encodeBlockNumber(number)...), hash.Bytes()...)

            if err := db.Put(key, rlp); err != nil {

                        glog.Fatalf("failed to store block body into database: %v", err)

            }

            glog.V(logger.Debug).Infof("stored block body [%x…]", hash.Bytes()[:4])

            return nil}
```

`WriteBody` 先对区块体进行 *RLP* 编码，然后调用 `WriteBodyRLP` 将区块体的 *RLP* 编码写到数据库中。

`WriteBodyRLP` 根据上面的规则组装 *key*，然后向数据库中写入一条记录。

还有一个 `WriteBlock` 函数分别调用 `WriteBody` 和 `WriteHeader` 将区块写到数据库中。此外还有 `GetHeader` `GetBody` `GetBlock` 函数用于从数据库中读取区块。

# 交易存储

除了区块外，数据库中还存储了所有的交易，每条交易的存储格式如下：

```
txHash -> rlpEncode(tx)

txHash + txMetaSuffix -> rlpEncode(txMeta)
```

每条交易对应存储两条数据，一条是交易本身，一条是交易的元信息（*meta*）。交易以交易的 *hash* 为 *key*、交易的 *RLP* 编码为 *value* 存储；元信息以 *txHash+txMetaSuffix* 为 *key*、元信息的 *RLP* 编码为 *value* 存储。元信息中包含交易所在区块的区块 *hash*、区块号、交易在区块中的索引。具体可以看 `WriteTransactions` 函数：

```
// WriteTransactions stores the transactions associated with a specific block// into the given database. Beside writing the transaction, the function also// stores a metadata entry along with the transaction, detailing the position// of this within the blockchain.
```

```
func WriteTransactions(db ethdb.Database, block *types.Block) error {

            batch := db.NewBatch()



            // Iterate over each transaction and encode it with its metadata

            for i, tx := range block.Transactions() {

                        // Encode and queue up the transaction for storage

                        data, err := rlp.EncodeToBytes(tx)

                        if err != nil {

                                    return err

                        }

                        if err := batch.Put(tx.Hash().Bytes(), data); err != nil {

                                    return err

                        }

                        // Encode and queue up the transaction metadata for storage

                        meta := struct {

                                    BlockHash  common.Hash

                                    BlockIndex uint64

                                    Index      uint64

                        }{

                                    BlockHash:  block.Hash(),

                                    BlockIndex: block.NumberU64(),

                                    Index:      uint64(i),

                        }

                        data, err = rlp.EncodeToBytes(meta)

                        if err != nil {
```

```
                                return err

                        }

                        if err := batch.Put(append(tx.Hash().Bytes(), txMetaSuffix...), data); err != n
il {

                                return err

                        }

                }

                // Write the scheduled data into the database

                if err := batch.Write(); err != nil {

                        glog.Fatalf("failed to store transactions into database: %v", err)

                }

                return nil}
```

此外还有`GetTransaction`函数，根据交易 *hash* 从数据库中读取交易，它返回对应的交易、交易所在区块的区块 *hash*、交易所在区块的区块号、交易在区块中的索引。