以太坊数据持久化ethdb源码解析

一、levelDB简介

go-ethereum所有的数据存储在levelDB这个Google开源的KeyValue文件数据库中，整个区块链的所有数据都存储在一个levelDB的数据库中，levelDB支持按照文件大小切分文件的功能，所以我们看到的区块链的数据都是一个一个小文件，其实这些小文件都是一个同一个levelDB实例。这里简单的看下levelDB的go封装代码。

levelDB官方网站介绍的特点

特点：

- key和value都是任意长度的字节数组；
- entry（即一条K-V记录）默认是按照key的字典顺序存储的，当然开发者也可以重载这个排序函数；
- 提供的基本操作接口：Put()、Delete()、Get()、Batch()；
- 支持批量操作以原子操作进行；
- 可以创建数据全景的snapshot(快照)，并允许在快照中查找数据；
- 可以通过前向（或后向）迭代器遍历数据（迭代器会隐含的创建一个snapshot）；
- 自动使用Snappy压缩数据；
- 可移植性；

限制：

- 非关系型数据模型（NoSQL），不支持sql语句，也不支持索引；
- 一次只允许一个进程访问一个特定的数据库；
- 没有内置的C/S架构，但开发者可以使用LevelDB库自己封装一个server；

源码所在的目录在ethereum/ethdb目录。代码比较简单， 分为下面三个文件

- database.go levelDB的封装代码
- memory_database.go 供测试用的基于内存的数据库，不会持久化为文件，仅供测试
- interface.go 定义了数据库的接口
- database_test.go 测试案例

二、interface.go

看下面的代码，基本上定义了KeyValue数据库的基本操作， Put， Get， Has， Delete等基本操

作，levelDB是不支持SQL的，基本可以理解为数据结构里面的Map。

```
package ethdb
const IdealBatchSize = 100 * 1024

// Putter wraps the database write operation supported by both batches and regular
databases.
//Putter接口定义了批量操作和普通操作的写入接口
type Putter interface {
    Put(key []byte, value []byte) error
}

// Database wraps all database operations. All methods are safe for concurrent use.
//数据库接口定义了所有的数据库操作， 所有的方法都是多线程安全的。
type Database interface {
    Putter
    Get(key []byte) ([]byte, error)
    Has(key []byte) (bool, error)
    Delete(key []byte) error
    Close()
    NewBatch() Batch
}

// Batch is a write-only database that commits changes to its host database
// when Write is called. Batch cannot be used concurrently.
//批量操作接口，不能多线程同时使用，当Write方法被调用的时候，数据库会提交写入的更改。
type Batch interface {
    Putter
    ValueSize() int // amount of data in the batch
    Write() error
}
```

三、memory_database.go

这个基本上就是封装了一个内存的Map结构。然后使用了一把锁来对多线程进行资源的保护。

```
type MemDatabase struct {
    db map[string][]byte
    lock sync.RWMutex
}

func NewMemDatabase() (*MemDatabase, error) {
    return &MemDatabase{
        db: make(map[string][]byte),
    }, nil
```

```
}

func (db *MemDatabase) Put(key []byte, value []byte) error {
    db.lock.Lock()
    defer db.lock.Unlock()
    db.db[string(key)] = common.CopyBytes(value)
    return nil
}
func (db *MemDatabase) Has(key []byte) (bool, error) {
    db.lock.RLock()
    defer db.lock.RUnlock()

    _, ok := db.db[string(key)]
    return ok, nil
}
```

然后是Batch的操作。也比较简单，一看便明白。

```
type kv struct{ k, v []byte }
type memBatch struct {
    db *MemDatabase
    writes []kv
    size int
}
func (b *memBatch) Put(key, value []byte) error {
    b.writes = append(b.writes, kv{common.CopyBytes(key), common.CopyBytes(value)})
    b.size += len(value)
    return nil
}
func (b *memBatch) Write() error {
    b.db.lock.Lock()
    defer b.db.lock.Unlock()

    for _, kv := range b.writes {
        b.db.db[string(kv.k)] = kv.v
    }
    return nil
}
```

四、database.go

这个就是实际ethereum客户端使用的代码， 封装了levelDB的接口。

```
import (
    "strconv"
```

```
    "strings"
    "sync"
    "time"

    "github.com/ethereum/go-ethereum/log"
    "github.com/ethereum/go-ethereum/metrics"
    "github.com/syndtr/goleveldb/leveldb"
    "github.com/syndtr/goleveldb/leveldb/errors"
    "github.com/syndtr/goleveldb/leveldb/filter"
    "github.com/syndtr/goleveldb/leveldb/iterator"
    "github.com/syndtr/goleveldb/leveldb/opt"
    gometrics "github.com/rcrowley/go-metrics"
)
```

使用了github.com/syndtr/goleveldb/leveldb的leveldb的封装，所以一些使用的文档可以在那里找到。可以看到，数据结构主要增加了很多的Mertrics用来记录数据库的使用情况，增加了quitChan用来处理停止时候的一些情况，这个后面会分析。如果下面代码可能有疑问的地方应该再Filter: filter.NewBloomFilter(10)这个可以暂时不用关注，这个是levelDB里面用来进行性能优化的一个选项，可以不用理会。

```
type LDBDatabase struct {
    fn string // filename for reporting
    db *leveldb.DB // LevelDB instance

    getTimer gometrics.Timer // Timer for measuring the database get request counts
 and latencies
    putTimer gometrics.Timer // Timer for measuring the database put request counts
 and latencies
    ...metrics

    quitLock sync.Mutex // Mutex protecting the quit channel access
    quitChan chan chan error // Quit channel to stop the metrics collection before
closing the database

    log log.Logger // Contextual logger tracking the database path
}

// NewLDBDatabase returns a LevelDB wrapped object.
func NewLDBDatabase(file string, cache int, handles int) (*LDBDatabase, error) {
    logger := log.New("database", file)
    // Ensure we have some minimal caching and file guarantees
    if cache < 16 {
        cache = 16
    }
    if handles < 16 {
        handles = 16
```

```
    }
    logger.Info("Allocated cache and file handles", "cache", cache, "handles", hand
les)
    // Open the db and recover any potential corruptions
    db, err := leveldb.OpenFile(file, &opt.Options{
        OpenFilesCacheCapacity: handles,
        BlockCacheCapacity: cache / 2 * opt.MiB,
        WriteBuffer: cache / 4 * opt.MiB, // Two of these are used internally
        Filter: filter.NewBloomFilter(10),
    })
    if _, corrupted := err.(*errors.ErrCorrupted); corrupted {
        db, err = leveldb.RecoverFile(file, nil)
    }
    // (Re)check for errors and abort if opening of the db failed
    if err != nil {
        return nil, err
    }
    return &LDBDatabase{
        fn: file,
        db: db,
        log: logger,
    }, nil
}
```

再看看下面的Put和Has的代码，因为github.com/syndtr/goleveldb/leveldb封装之后的代码是支持多线程同时访问的，所以下面这些代码是不用使用锁来保护的，这个可以注意一下。这里面大部分的代码都是直接调用leveldb的封装，所以不详细介绍了。 有一个比较有意思的地方是Metrics代码。

```
// Put puts the given key / value to the queue
func (db *LDBDatabase) Put(key []byte, value []byte) error {
    // Measure the database put latency, if requested
    if db.putTimer != nil {
        defer db.putTimer.UpdateSince(time.Now())
    }
    // Generate the data to write to disk, update the meter and write
    //value = rle.Compress(value)

    if db.writeMeter != nil {
        db.writeMeter.Mark(int64(len(value)))
    }
    return db.db.Put(key, value, nil)
}

func (db *LDBDatabase) Has(key []byte) (bool, error) {
    return db.db.Has(key, nil)
```

```
}
```

五、Metrics的处理

之前在创建NewLDBDatabase的时候，并没有初始化内部的很多Mertrics，这个时候Mertrics是为nil的。初始化Mertrics是在Meter方法中。外部传入了一个prefix参数，然后创建了各种Mertrics(具体如何创建Merter，会后续在Meter专题进行分析),然后创建了quitChan。 最后启动了一个线程调用了db.meter方法。

```
// Meter configures the database metrics collectors and
func (db *LDBDatabase) Meter(prefix string) {
    // Short circuit metering if the metrics system is disabled
    if !metrics.Enabled {
        return
    }
    // Initialize all the metrics collector at the requested prefix
    db.getTimer = metrics.NewTimer(prefix + "user/gets")
    db.putTimer = metrics.NewTimer(prefix + "user/puts")
    db.delTimer = metrics.NewTimer(prefix + "user/dels")
    db.missMeter = metrics.NewMeter(prefix + "user/misses")
    db.readMeter = metrics.NewMeter(prefix + "user/reads")
    db.writeMeter = metrics.NewMeter(prefix + "user/writes")
    db.compTimeMeter = metrics.NewMeter(prefix + "compact/time")
    db.compReadMeter = metrics.NewMeter(prefix + "compact/input")
    db.compWriteMeter = metrics.NewMeter(prefix + "compact/output")

    // Create a quit channel for the periodic collector and run it
    db.quitLock.Lock()
    db.quitChan = make(chan chan error)
    db.quitLock.Unlock()

    go db.meter(3 * time.Second)
}
```

这个方法每3秒钟获取一次leveldb内部的计数器，然后把他们公布到metrics子系统。 这是一个无限循环的方法， 直到quitChan收到了一个退出信号。

```
// meter periodically retrieves internal leveldb counters and reports them to
// the metrics subsystem.
// This is how a stats table look like (currently):
//下面的注释就是我们调用 db.db.GetProperty("leveldb.stats")返回的字符串，后续的代码需要解析这个字符串并把信息写入到Meter中。

// Compactions
```

```
// Level | Tables | Size(MB) | Time(sec) | Read(MB) | Write(MB)
// ------+-----------+--------------+--------------+--------------+-----------
----
// 0 | 0 | 0.00000 | 1.27969 | 0.00000 | 12.31098
// 1 | 85 | 109.27913 | 28.09293 | 213.92493 | 214.26294
// 2 | 523 | 1000.37159 | 7.26059 | 66.86342 | 66.77884
// 3 | 570 | 1113.18458 | 0.00000 | 0.00000 | 0.00000

func (db *LDBDatabase) meter(refresh time.Duration) {
    // Create the counters to store current and previous values
    counters := make([][]float64, 2)
    for i := 0; i < 2; i++ {
        counters[i] = make([]float64, 3)
    }
    // Iterate ad infinitum and collect the stats
    for i := 1; ; i++ {
        // Retrieve the database stats
        stats, err := db.db.GetProperty("leveldb.stats")
        if err != nil {
            db.log.Error("Failed to read database stats", "err", err)
            return
        }
        // Find the compaction table, skip the header
        lines := strings.Split(stats, "\n")
        for len(lines) > 0 && strings.TrimSpace(lines[0]) != "Compactions" {
            lines = lines[1:]
        }
        if len(lines) <= 3 {
            db.log.Error("Compaction table not found")
            return
        }
        lines = lines[3:]

        // Iterate over all the table rows, and accumulate the entries
        for j := 0; j < len(counters[i%2]); j++ {
            counters[i%2][j] = 0
        }
        for _, line := range lines {
            parts := strings.Split(line, "|")
            if len(parts) != 6 {
                break
            }
            for idx, counter := range parts[3:] {
                value, err := strconv.ParseFloat(strings.TrimSpace(counter), 64)
                if err != nil {
                    db.log.Error("Compaction entry parsing failed", "err", err)
                    return
                }
```

```
                counters[i%2][idx] += value
            }
        }
        // Update all the requested meters
        if db.compTimeMeter != nil {
            db.compTimeMeter.Mark(int64((counters[i%2][0] - counters[(i-1)%2][0]) *
 1000 * 1000 * 1000))
        }
        if db.compReadMeter != nil {
            db.compReadMeter.Mark(int64((counters[i%2][1] - counters[(i-1)%2][1]) *
 1024 * 1024))
        }
        if db.compWriteMeter != nil {
            db.compWriteMeter.Mark(int64((counters[i%2][2] - counters[(i-1)%2][2])
* 1024 * 1024))
        }
        // Sleep a bit, then repeat the stats collection
        select {
        case errc := <-db.quitChan:
            // Quit requesting, stop hammering the database
            errc <- nil
            return

        case <-time.After(refresh):
            // Timeout, gather a new set of stats
        }
    }
}
```