

##Leveldb源码笔记之读操作

####key逻辑分类 根据我们之前文章的描述，leveldb的数据存储可能存在在内存的memtable中，或者磁盘的sstable中，但是key的实际存储格式会略微有差异，代码里按照存储的位置，划分为以下几种类型：

- memtable: 逻辑上称为memtable_key
- sstable: 逻辑上称为internal_key
- key: 用户提供的key，我们称之为user_key

当用户去查询某个key时，leveldb会先利用key构建起LookupKey类。LookupKey类内部的完整数据即memtable_key，可以方便的利用成员函数截取memtable_key,internal_key,user_key以方便去memtable和sstable中查询。事实上LookupKey是由key, sequence number组成的，如之前文章提到：

- 如果普通Get()操作，sequence number为last sequence number
- 如果是使用的snapshot，sequence number为snapshot sequence number

```
1 // dbformat.h
2 // lookup key format:
3 // start_      kstart_          end_
4 // |           |
5 // |           |<--user_key-->|
6 // |           |<-----internal_key----->|
7 // |<-----memtable_key----->|
8 //
9 // | 1--5 byte | klength byte |          8 byte   |
10 //
11 // | klength + 8 | raw key    | pack(sequence number, type) |
12 //
13 // A helper class useful for DBImpl::Get()
14 class LookupKey {
15 public:
16     // Initialize *this for looking up user_key at a snapshot with
17     // the specified sequence number.
18     LookupKey(const Slice& user_key, SequenceNumber sequence);
19
20     ~LookupKey();
21
22     // Return a key suitable for lookup in a MemTable.
23     Slice memtable_key() const { return Slice(start_, end_ - start_); }
24
25     // Return an internal key (suitable for passing to an internal iterator)
```

```

26 slice internal_key() const { return slice(kstart_, end_ - kstart_); }
27
28 // Return the user key
29 slice user_key() const { return slice(kstart_, end_ - kstart_ - 8); }
30
31 private:
32 const char* start_;
33 const char* kstart_;
34 const char* end_;
35 char space_[200];      // Avoid allocation for short keys
36
37 // No copying allowed
38 LookupKey(const LookupKey&);
39 void operator=(const LookupKey&);
40 };

```

如图：

Key 分类

Memtable key

1--5 Bytes	Key size Bytes	8 Bytes
Key size + 8	Key	(Sequence number << 8) KTypeValue

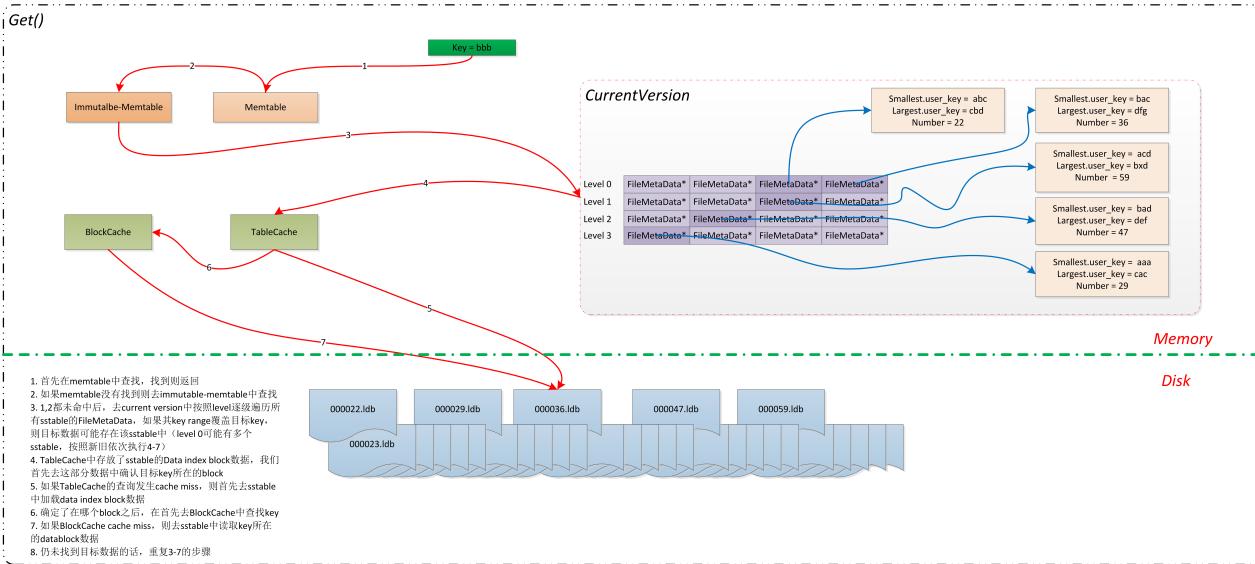
Internal key

Key size Bytes	8 Bytes
key	(Sequence number << 8) KTypeValue

User key

Key size Bytes
key

###读操作 图示Get()操作的基本逻辑如下：



以上我们是假设sstable没有filter的情况下操作逻辑

####cache 无论是table cache, 还是block cache, 都是使用了相同的数据结构LRUCache来实现的, 区别只在于内部存储的数据不同。 LRUCache是通过k/v方式存储的, 对于: ####TableCache:

- key: 其实就是file number

```

1 // table_cache.cc
2 char buf[sizeof(file_number)];
3 EncodeFixed64(buf, file_number);
4 slice key(buf, sizeof(buf));

```

- value: TableAndFile, 其实主要是sstable index block里的数据

```

1 // table_cache.cc
2 struct TableAndFile {
3     RandomAccessFile* file;
4     Table* table;
5 };
6
7 // table.cc
8 // Table里的主要数据即下述
9 struct Table::Rep {
10     ~Rep() {
11         delete filter;
12         delete [] filter_data;
13         delete index_block;
14     }
15
16     Options options;
17     Status status;
18     RandomAccessFile* file;
19     uint64_t cache_id;
20     FilterBlockReader* filter;
21     const char* filter_data;

```

```
22     BlockHandle metaindex_handle; // Handle to metaindex_block: saved
23     from footer
24     Block* index_block;
25 }
```

####BlockCache:

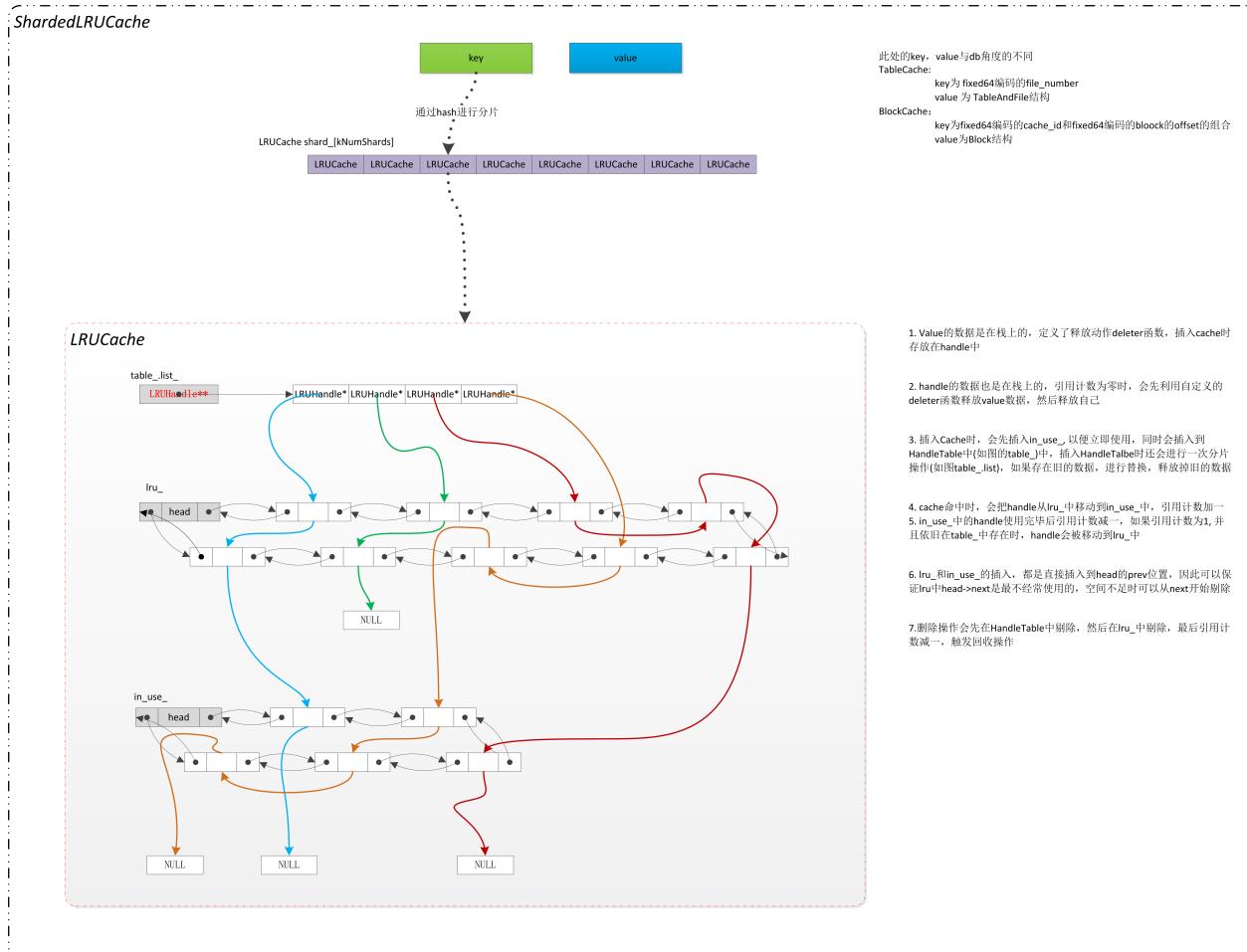
- key: 其实是 cache_id 和 block 在sstable中的offset的组合

```
1 // table.cc
2 char cache_key_buffer[16];
3 // 构造block_cache 的key
4 EncodeFixed64(cache_key_buffer, table->rep_->cache_id);
5 EncodeFixed64(cache_key_buffer+8, handle.offset());
6 Slice key(cache_key_buffer, sizeof(cache_key_buffer));
```

- value: data block 内容

```
1 // block.h
2 class Block {
3 public:
4     // Initialize the block with the specified contents.
5     explicit Block(const BlockContents& contents);
6
7     ~Block();
8
9     size_t size() const { return size_; }
10    Iterator* NewIterator(const Comparator* comparator);
11
12 private:
13     uint32_t NumRestarts() const;
14
15     const char* data_;
16     size_t size_;
17     uint32_t restart_offset_; // offset in data_ of restart array
18     bool owned_; // Block owns data_[]
19
20     // No copying allowed
21     Block(const Block&);
22     void operator=(const Block&);
23
24     class Iter;
25 };
```

####cache 逻辑结构图示



##leveldb源码笔记之Compact

简介 leveldb中只有minor compaction 和 major compaction两种

- 代码中通过调用DBImpl::MaybeScheduleCompaction()来触发两种compaction

```

1 // db_impl.cc
2 void DBImpl::MaybeScheduleCompaction() {
3     mutex_.AssertHeld();
4     // 确保只有一个后台线程在做compact
5     if (bg_compaction_scheduled_) {
6         // Already scheduled
7     } else if (shutting_down_.Acquire_Load()) {
8         // DB is being deleted; no more background compactions
9     } else if (!bg_error_.ok()) {
10        // Already got an error; no more changes
11    } else if (imm_ == NULL &&
12               manual_compaction_ == NULL &&
13               !versions_->NeedsCompaction()) {
14        // No work to be done
15    } else {
16        bg_compaction_scheduled_ = true;
17        // 启动compact线程，主要逻辑是通过DBImpl::BackgroundCompaction()实现
18        env_->Schedule(&DBImpl::BGwork, this);

```

```
19 }
20 }
```

调用时机:

1. 每次写入前，需要确保空间充足，如果空间不足，尝试将memtable转换为immutable-memtable，之后调用DBImpl::MaybeScheduleCompaction()
2. 每次重启db，binlog recover结束后，会触发调用DBImpl::MaybeScheduleCompaction()
3. 每次读取一条记录结束时会触发调用DBImpl::MaybeScheduleCompaction()

###minor compaction 方式:

- 将immutable-memtable dump到磁盘，形成sstable
- sstable一般位于level-0,如果sstable的key范围和当前level没有重叠会尝试下移，最多不会超过 config::kMaxMemCompactLevel(默认为2) 触发时机:
- 每次调用BackGroudCompaction如果存在immutable-memtable都会触发将其dump到磁盘

###major compaction 方式:

- 将level-n的sstable 与 level-(n+1)中与之存在key范围重叠的sstable多路归并，生成level-(n+1)的 sstable
- 如果是level-0,则由于level-0中sstable之间key有重叠，所以level-0参与compact的sstable可能不止一个 触发时机:
- 第一种是size触发类型(优先):

```
1 // version_set.cc
2 void VersionSet::Finalize(version* v) {
3     // Precomputed best level for next compaction
4     int best_level = -1;
5     double best_score = -1;
6
7     for (int level = 0; level < config::kNumLevels-1; level++) {
8         double score;
9         if (level == 0) {
10             // we treat level-0 specially by bounding the number of files
11             // instead of number of bytes for two reasons:
12             //
13             // 对于较大的write buffer，不过多的进行level-0的compactions是好的
14             // (1) With larger write-buffer sizes, it is nice not to do too
15             // many level-0 compactions.
16             //
17             // 因为每次读操作都会触发level-0的归并，因此当个别的文件size很小的时候
18             // 我们期望避免level-0有太多文件存在
19             // (2) The files in level-0 are merged on every read and
20             // therefore we wish to avoid too many files when the individual
21             // file size is small (perhaps because of a small write-buffer
22             // setting, or very high compression ratios, or lots of
23             // overwrites/deletions).
24             score = v->files_[level].size() /
25                 static_cast<double>(config::kL0_CompactionTrigger);
```

```

26 } else {
27     // Compute the ratio of current size to size limit.
28     const uint64_t level_bytes = TotalFileSize(v->files_[level]);
29     score = static_cast<double>(level_bytes) / MaxBytesForLevel(level);
30 }
31
32 if (score > best_score) {
33     best_level = level;
34     best_score = score;
35 }
36
37 v->compaction_level_ = best_level;
38 v->compaction_score_ = best_score;
39
40 }

```

- 对于level-0:
- score = level-0文件数/config::kL0_CompactionTrigger(默认为4)
- 对于level-n(n>0):
- score = 当前level的字节数 / (10n * 220) 220 即1MB
- score >= 1,当前level就会被标识起来, 等待触发 compaction

第二种是seek触发:

```

1 // version_edit.h
2
3 // 记录了文件编号, 文件大小, 最小key, 最大key
4 // sstable文件的命名就是按照file number + 特定后缀完成的
5 struct FileMetaData {
6     int refs;
7     int allowed_seeks;           // Seek allowed until compaction
8     uint64_t number;
9     uint64_t file_size;          // File size in bytes
10    InternalKey smallest;       // Smallest internal key served by table
11    InternalKey largest;         // Largest internal key served by table
12
13    FileMetaData() : refs(0), allowed_seeks(1 << 30), file_size(0) { }
14 };
15
16 // version_set.cc
17
18 // Apply all of the edits in *edit to the current state.
19 void Apply(VersionEdit* edit) {
20     ...
21     for (size_t i = 0; i < edit->new_files_.size(); i++) {
22         const int level = edit->new_files_[i].first;
23         FileMetaData* f = new FileMetaData(edit->new_files_[i].second);
24         f->refs = 1;
25         // We arrange to automatically compact this file after

```

```

26 // a certain number of seeks. Let's assume:
27 // (1) One seek costs 10ms
28 // (2) Writing or reading 1MB costs 10ms (100MB/s)
29 // (3) A compaction of 1MB does 25MB of IO:
30 //      1MB read from this level
31 //      10-12MB read from next level(boundaries may be misaligned)
32 //      10-12MB written to next level
33 // This implies that 25 seeks cost the same as the compaction
34 // of 1MB of data. I.e., one seek costs approximately the
35 // same as the compaction of 40KB of data. we are a little
36 // conservative and allow approximately one seek for every 16KB
37 // of data before triggering a compaction.
38 // 1次seek相当与compact 40kb的数据,
39 // 那么n次seek大概和compact一个sstable相当( $n = \text{sstable\_size} / 40\text{kb}$ )
40 // 保守点, 这里搞了个16kb
41 f->allowed_seeks = (f->file_size / 16384); //  $2^{14} == 16384 == 16\text{kb}$ 
42 if (f->allowed_seeks < 100) f->allowed_seeks = 100;
43 ...
44 }
45 ...
46 }

```

- 当一个新的sstable建立时，会有一个allowed_seeks的初值：
- 作者认为1次sstable的seek（此处的seek就是指去sstable里查找指定key），相当于compact 40kb的数据，那么 sstable size / 40kb 次的seek操作，大概和compact一个 sstable 相当
- 保守的做法，allowed_seeks的初值为file_size/16kb
- 如果allowed_seeks小于100，令其为100 每当Get操作触发磁盘读，即sstable被读取，该数值就会减一；如果有多个sstable被读取，则仅首个被读取的sstable的allowed_seeks减一 allowed_seeks == 0 时，该sstable以及其所处level会被标识起来，等待触发 compaction

###sstable选择：

- 针对size触发类型，默认从当前level的首个sstable开始执行
- seek触发相对简单，sstable已经选择好了
- 对于level-0,需要将与选中的sstable存在key重叠的sstable也包含进此次compact
- 对于level-(n+1)，需要将与level-n中选中的sstable存在key重叠的sstable包含进此次compact

1 由于level-(n+1)多个sstable的参与扩展了整个compact的key的范围，我们可以使用该key范围将level-n中更多的sstable包含进此次compact 前提是保证level-n更多sstable的参与不会导致level-(n+1)的sstable数量再次增长。同时，参与整个compaction的字节数不超过 kExpandedCompactionByteSizeLimit = 25 * kTargetFileSize = 25 * 2MB;

- 为了保持公平，保证某个level中每个sstable都有机会参与compact:
- 存储当前level首次compact的sstable(s)的largest key，存入compact_point_[level]
- 当前level如果再次被size触发进行compact时，选择首个largest key大于compact_point_[level] sstable进行compact

##leveldb源码笔记之MVCC && Manifest

###MVCC

- 问题 针对同一条记录，如果读和写在同一时间发生时，reader可能会读取到不一致或者写了一半的数据
- 常见解决方案
- 悲观锁：最简单的方式,即通过锁来控制并发，但是效率非常的低,增加的产生死锁的机会
- 乐观锁：它假设多用户并发的事物在处理时不会彼此互相影响，各事物能够在不产生锁的情况下处理各自影响的那部分数据。在提交数据更新之前，每个事务会先检查在该事务读取数据后，有没有其他事务又修改了该数据。如果其他事务有更新的话，正在提交的事务会进行回滚;这样做不会有锁竞争更不会产生思索，但如果数据竞争的概率较高，效率也会受影响
- MVCC – Multiversion concurrency control: 每一个执行操作的用户，看到的都是数据库特定时刻的快照(snapshot), writer的任何未完成的修改都不会被其他的用户所看到;当对数据进行更新的时候并不是直接覆盖，而是先进行标记, 然后在其他地方添加新的数据，从而形成一个新版本, 此时再来读取的reader看到的就是最新的版本了。所以这种处理策略是维护了多个版本的数据的,但只有一个是最新的。

Key/Value 如前文所述，leveldb中写入一条记录，仅仅是先写入binlog，然后写入memtable

- binlog: binlog的写入只需要append，无需并发控制
- memtable: memtable是使用Memory Barriers技术实现的无锁的skiplist
- 更新: 真正写入memtable中参与skiplist排序的key其实是包含sequence number的，所以更新操作其实只是写入了一条新的k/v记录, 真正的更新由compact完成
- 删除: 如前文提到，删除一条Key时，仅仅是将type标记为kTypeDeletion，写入(同上述写入逻辑)了一条新的记录，并没有真正删除,真正的删除也是由compact完成的

Sequence Number

- sequence number 是一个由VersionSet直接持有的全局的编号，每次写入（注意批量写入时sequence number是相同的），就会递增
- 根据我们之前对写入操作的分析，我们可以看到，当插入一条key的时候，实际参与排序，存储的是key和sequence number以及type组成的 InternalKey
- 当我们进行Get操作时，我们只需要找到目标key，同时其sequence number <= specific sequence number
- 普通的读取，sepcific sequence number == last sequence number
- snapshot读取，sepcific sequenc number == snapshot sequence number

Snapshot snapshot 其实就是一个sequence number，获取snapshot，即获取当前的last sequence number 例如：

```

1  string key = 'a';
2  string value = 'b';
3  leveldb::Status s = db->Put(leveldb::WriteOptions(), key, value);
4  assert(s.ok())
5  leveldb::ReadOptions options;
6  options.snapshot = db->GetSnapshot();
7  string value = 'c';
8  leveldb::Status s = db->Put(leveldb::WriteOptions(), key, value);
9  assert(s.ok())
10 // ...
11 // ...
12 value.clear();
13 s = db->Get(leveldb::ReadOptions(), key, &value);    // value == 'c'

```

```

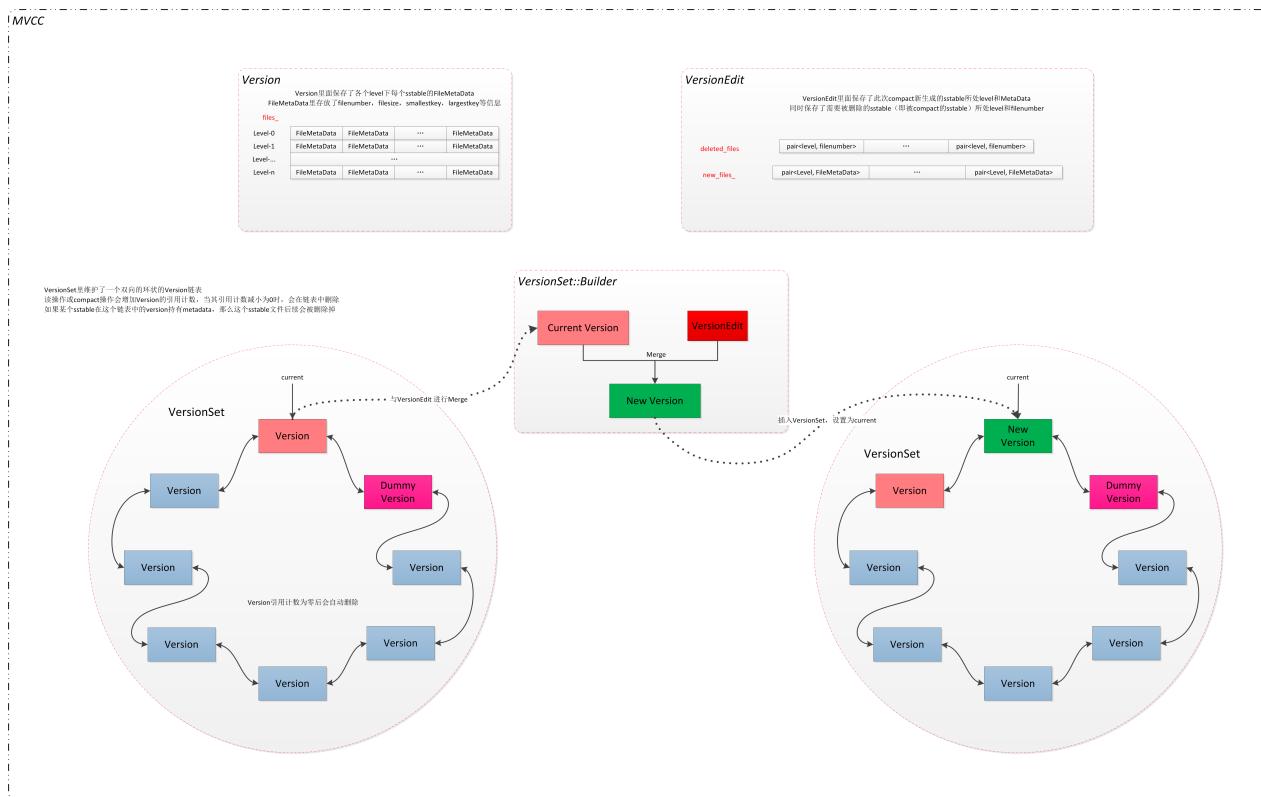
14 assert(s.ok())
15 s = db->Get(options, key, &value); // value == 'b'
16 assert(s.ok())

```

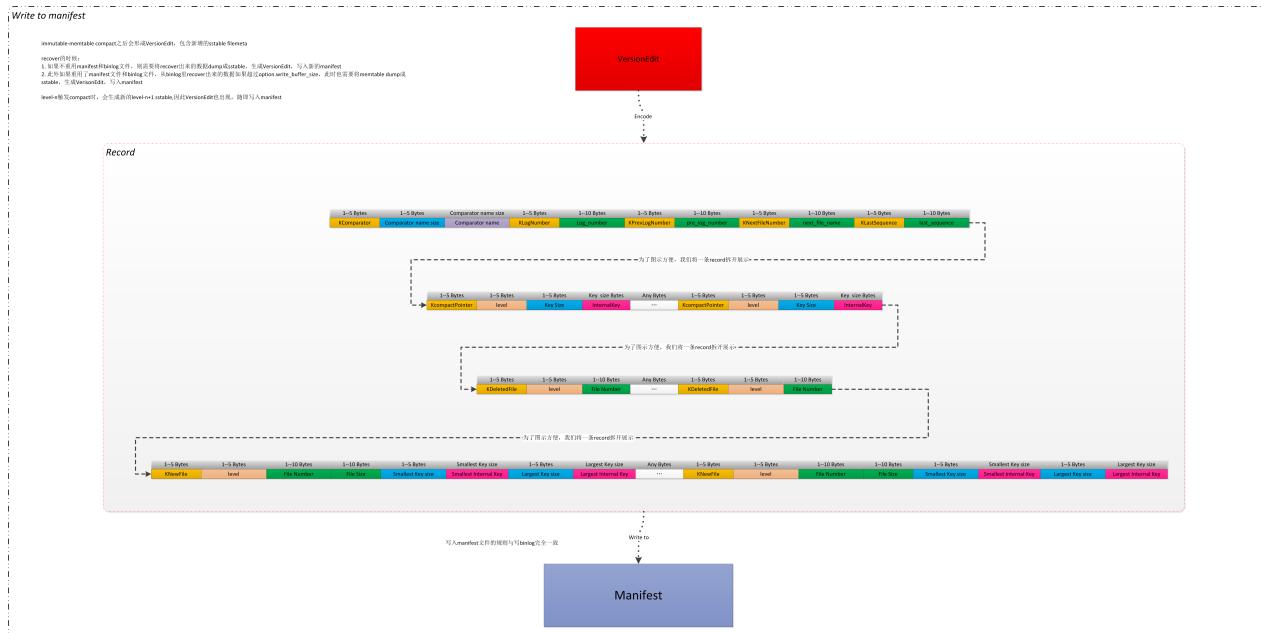
- 我们知道在sstable compact的时候，才会执行真正的删除或覆盖，而覆盖则是如果发现两条相同的记录会丢弃旧的(sequence number较小)一条，但是这同时会破坏掉snapshot
- 那么key = 'a', value = 'b'是如何避免compact时被丢弃掉的呢？
- db在内存中记录了当前用户持有的所有snapshot
- smallest snapshot = has snapshot ? oldest snapshot : last sequence number
- 当进行compact时，如果发现两条相同的记录，只有当两条记录的sequence number都小于smallest snapshot 时才丢弃掉其中sequence number较小的一条

####Sstable级别的MVCC是利用Version和VersionEdit实现的：

- 只有一个current version，持有了最新的sstable集合
- VersionEdit代表了一次current version的更新，新增了那些sstable，哪些sstable已经没用了等

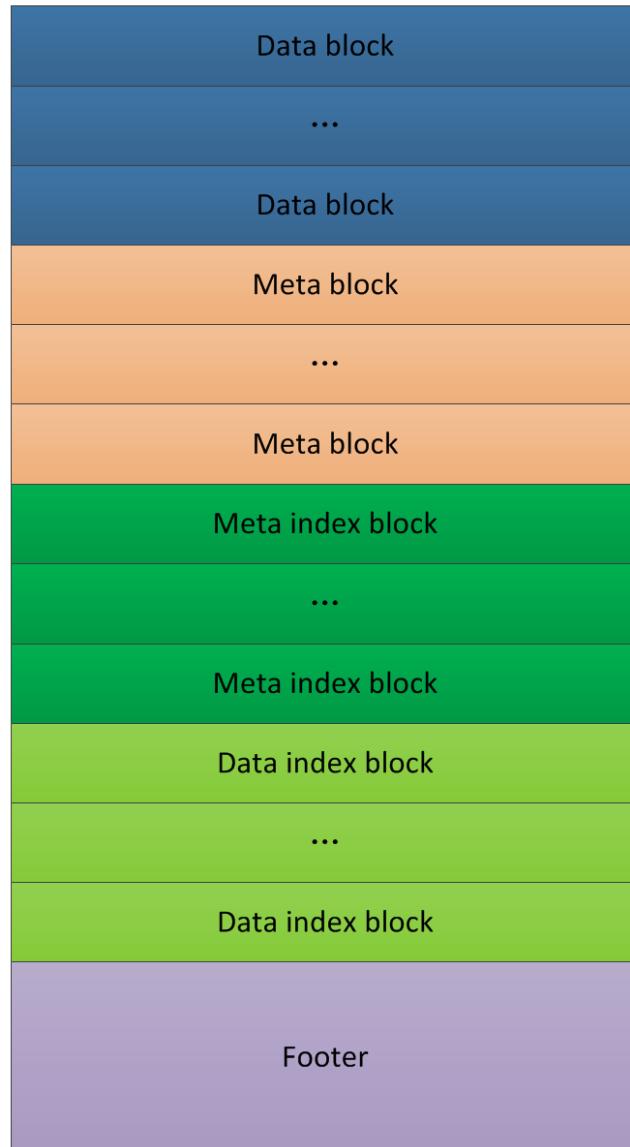


####Mainifest 每次current version 更新的数据(即新产生的VersionEdit)都写入mainifest文件，以便重启时recover



##Leveldb源码笔记之sstable 整体看下sstable的组要组成，如下：

Sstable

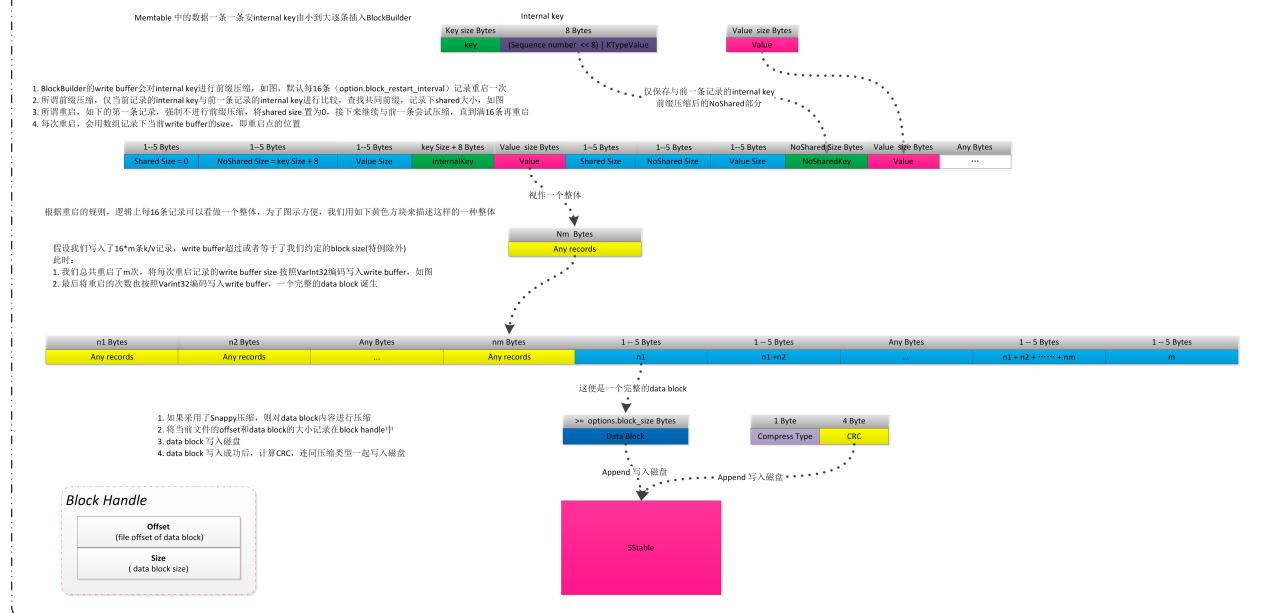


####sstable 生成细节 sstable 生成时机:

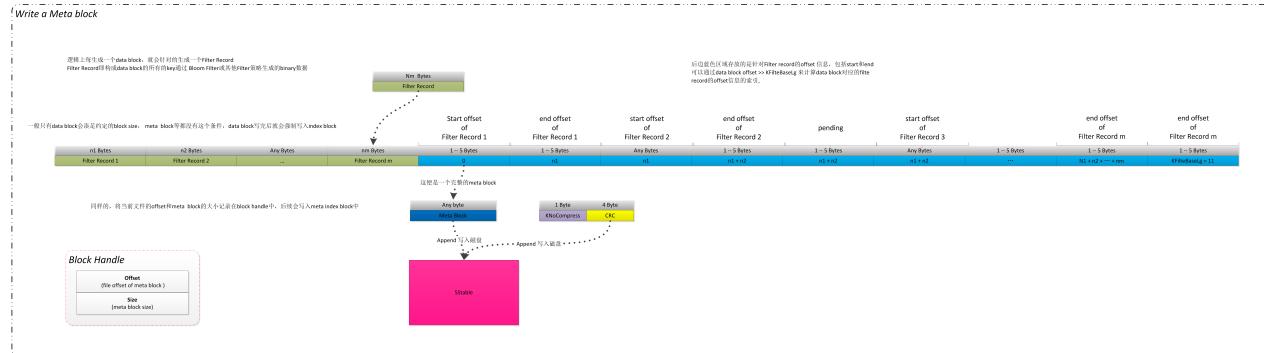
- minor compaction immutable-memtable 中的key/value dump到磁盘，生成sstable
- major compaction sstable compact (level-n sstable(s)与level-n+1 stables多路归并) 生成 level-n+1的sstable

首先是写入data block:

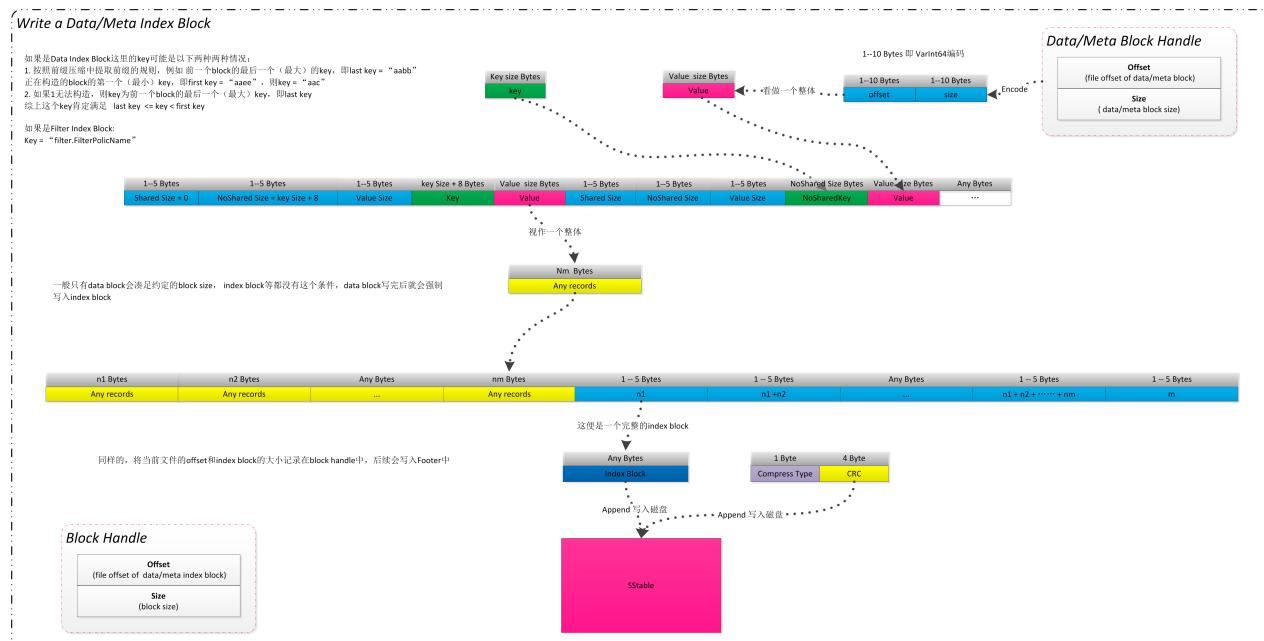
Write a Data Block



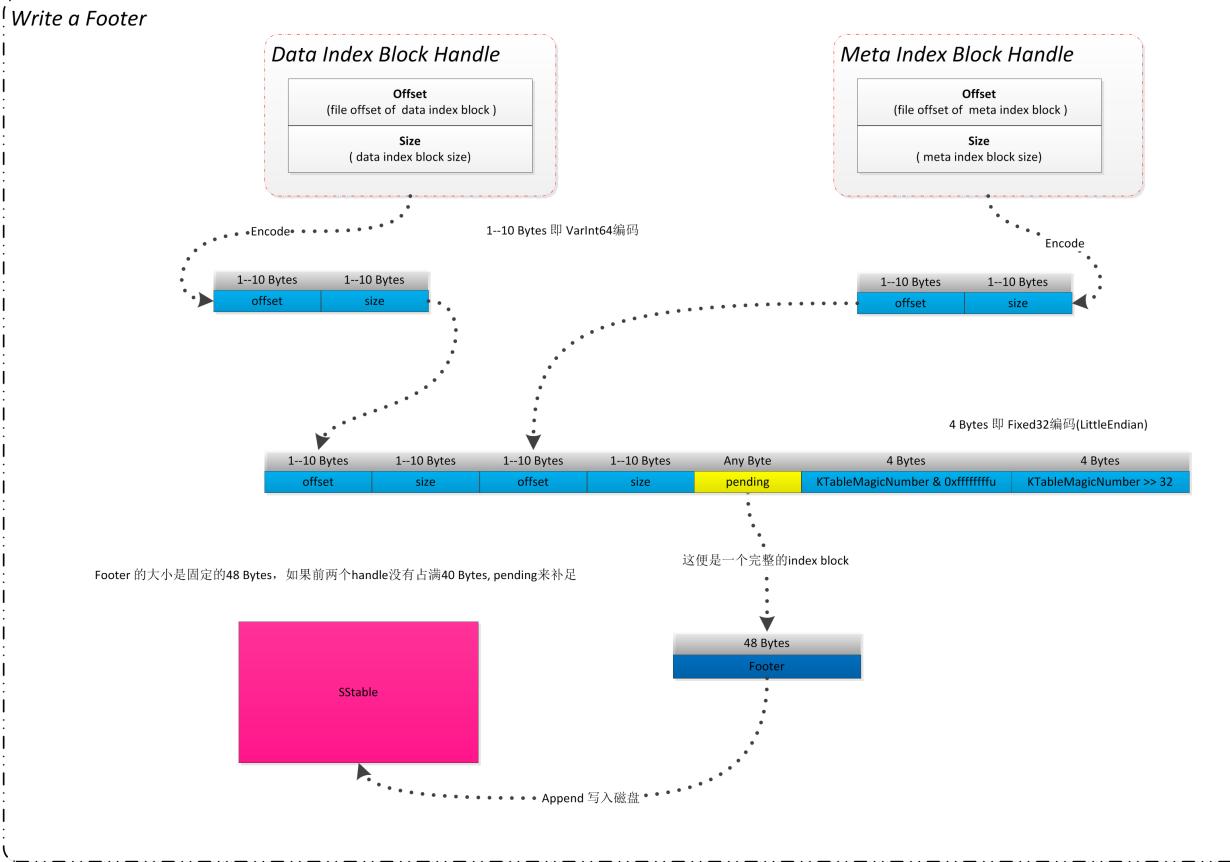
data block都写入完成后，接下来是meta block:



然后是data/meta block索引信息data/meta index block写入:



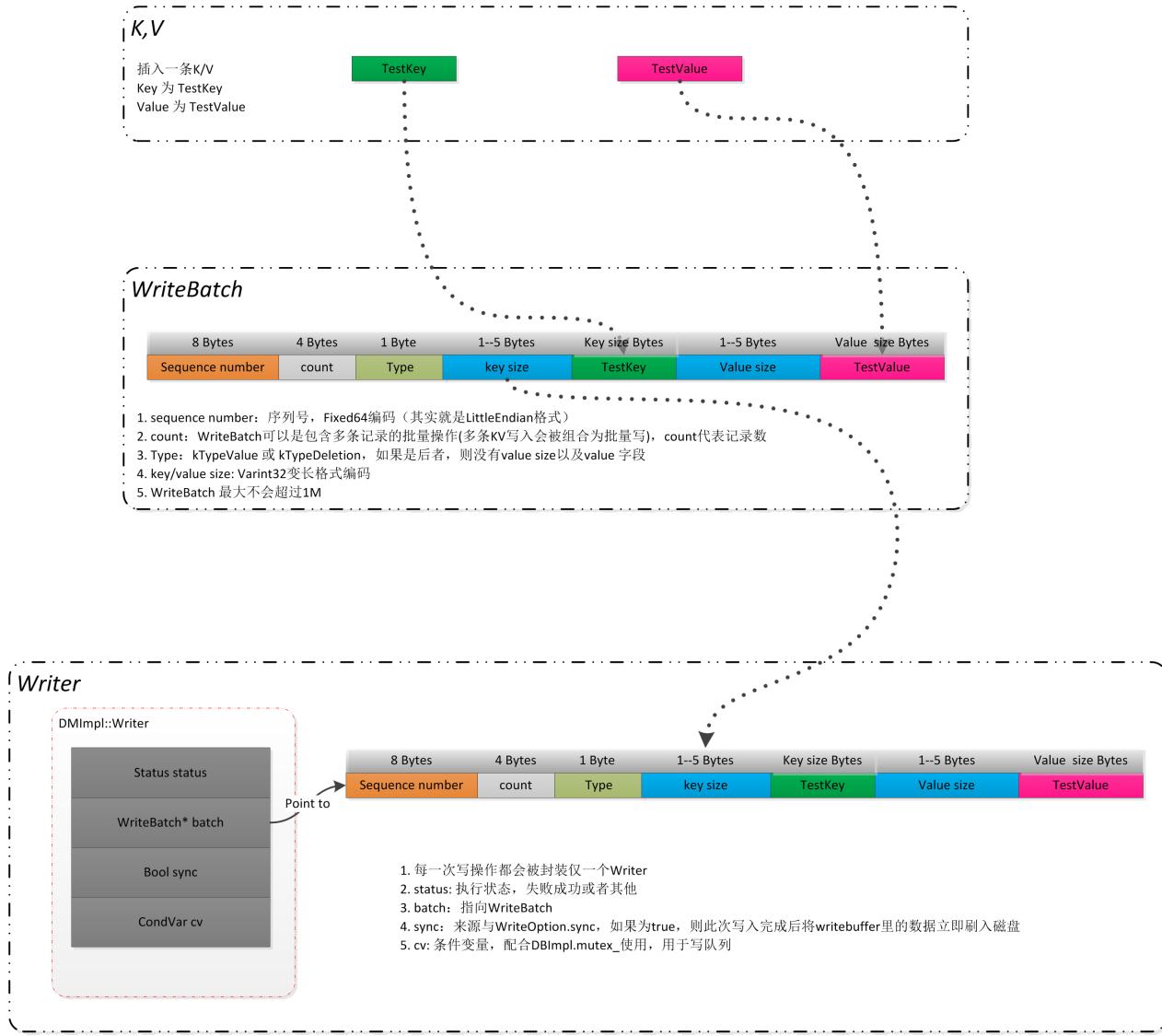
最后将index block的索引信息写入Footer



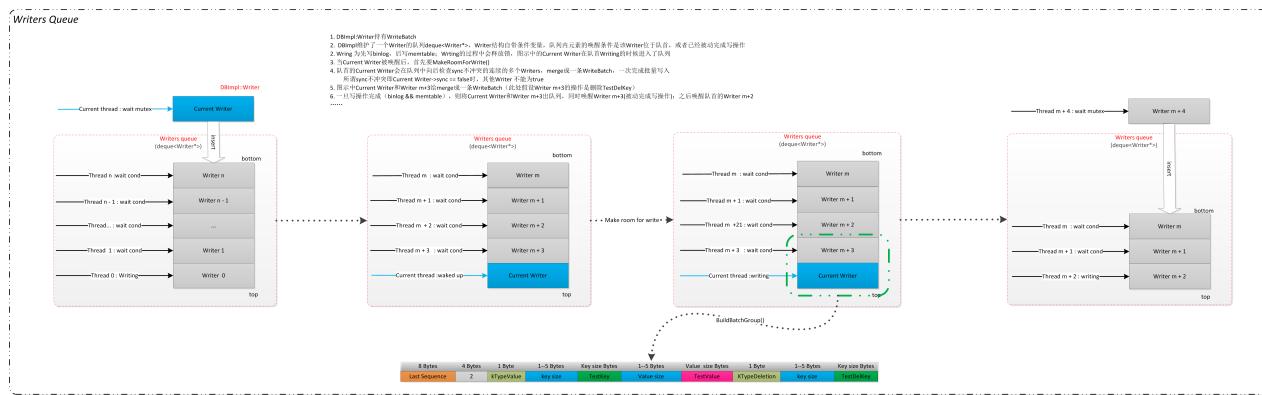
一个完整的sstable形成!

##Leveldb源码笔记之写入操作

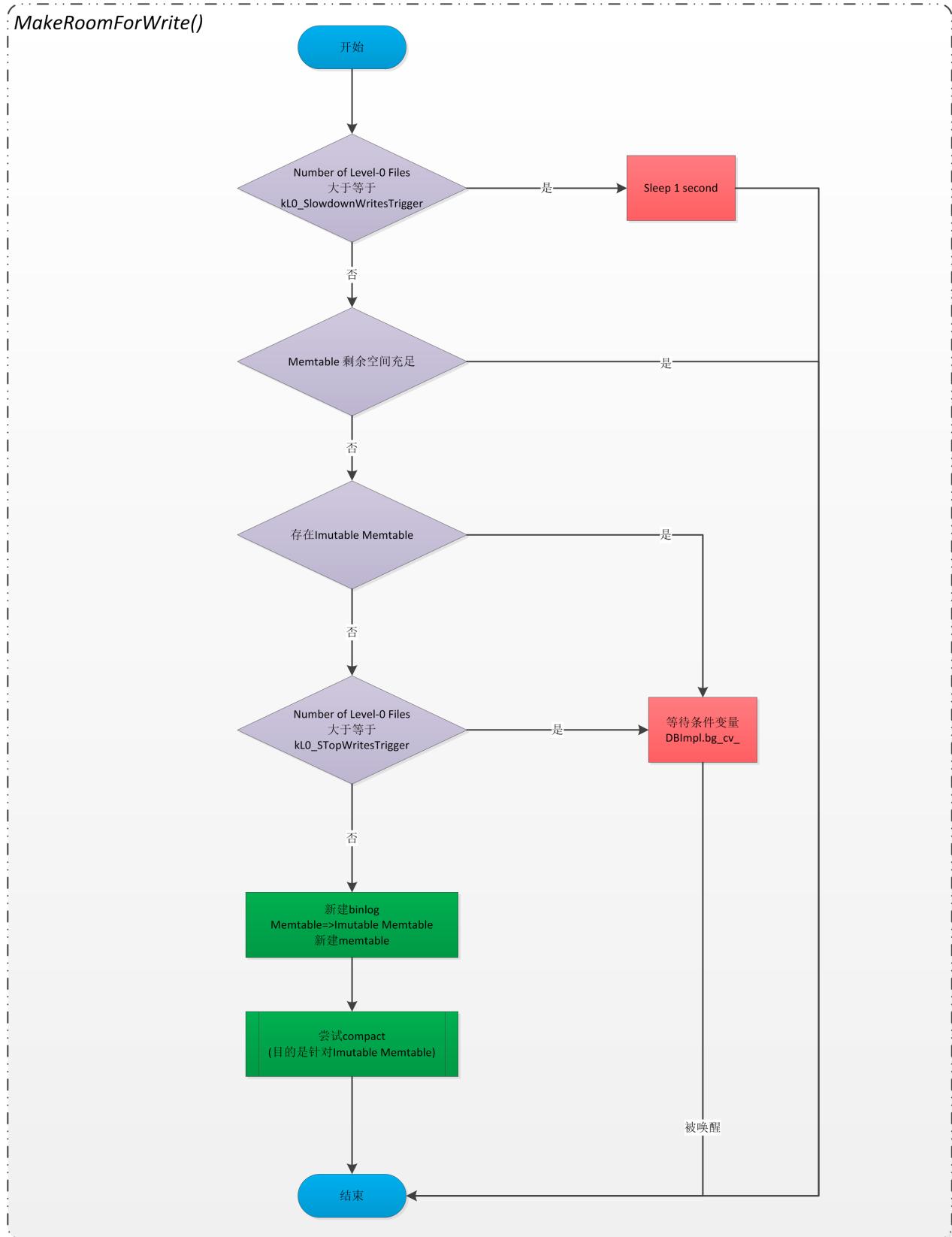
- 插入一条K/V记录



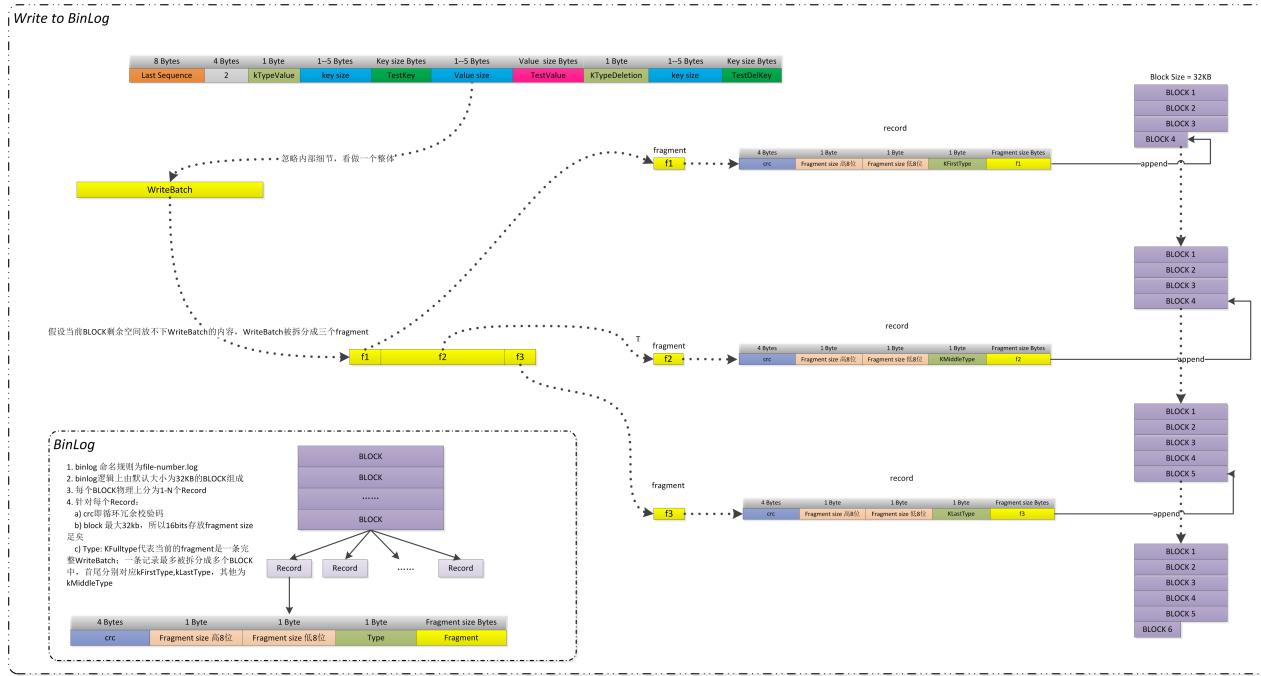
持有Writer的线程进入Writers队列,细节如下:



MakeRoomForWrite的流程图：



记录会首先写入磁盘上的binlog，避免程序crash时内存数据丢失：



1 此处我们做了一个极度夸张的假设来做演示：两条记录的大小超出一个block的大小，以至于被一切为三

KV记录插入内存中的Memtable:

