LevelDB 源码分析「五、Sorted Table」

2019.08.15 SF-Zhou

本系列的上一篇介绍了内存数据库，并且提到了内存数据库的大小限制问题。当内存数据块占用的内存达到阈值时（LevelDB 默认 4MB），会将当前的内存数据库 mem_ 转为不可修改的 imm_，并且为 mem_ 赋值一个新的内存数据库。这使得内存数据库的大小始终保持在阈值以下，同时保持着超高的读写性能。而不可修改的 imm_ 会经历 Compaction 过程，转为 Sorted Table 存储到磁盘中。本篇将详细阐述该过程。

1. Compact 内存数据库

在 DBImpl::Write 写入内存数据库前，会先调用 DBImpl::MakeRoomForWrite 申请空间：

```
Status DBImpl::MakeRoomForWrite(bool force) {
  mutex_.AssertHeld();
  assert(!writers_.empty());
  bool allow_delay = !force;
  Status s;
  while (true) {
    if (!bg_error_.ok()) {
      // Yield previous error
      s = bg_error_;
      break;
    } else if (allow_delay && versions_->NumLevelFiles(0) >=
```

```
} else if (allow_delay && versions_->NumLevelFiles(0) >=
                              config::kL0_SlowdownWritesTrigger) {
  // We are getting close to hitting a hard limit on the number of
  // L0 files.  Rather than delaying a single write by several
  // seconds when we hit the hard limit, start delaying each
  // individual write by 1ms to reduce latency variance.  Also,
  // this delay hands over some CPU to the compaction thread in
  // case it is sharing the same core as the writer.
  mutex_.Unlock();
  env_->SleepForMicroseconds(1000);
  allow_delay = false;  // Do not delay a single write more than once
  mutex_.Lock();
} else if (!force &&
           (mem_->ApproximateMemoryUsage() <= options_.write_buffer_size)) {
  // There is room in current memtable
  break;
} else if (imm_ != nullptr) {
  // We have filled up the current memtable, but the previous
  // one is still being compacted, so we wait.
  Log(options_.info_log, "Current memtable full; waiting...\n");
  background_work_finished_signal_.Wait();
} else if (versions_->NumLevelFiles(0) >= config::kL0_StopWritesTrigger) {
  // There are too many level-0 files.
  Log(options_.info_log, "Too many L0 files; waiting...\n");
  background_work_finished_signal_.Wait();
} else {
  // Attempt to switch to a new memtable and trigger compaction of old
  assert(versions_->PrevLogNumber() == 0);
  uint64_t new_log_number = versions_->NewFileNumber();
  WritableFile* lfile = nullptr;
  s = env_->NewWritableFile(LogFileName(dbname_, new_log_number), &lfile);
```

```
      if (!s.ok()) {
        // Avoid chewing through file number space in a tight loop.
        versions_->ReuseFileNumber(new_log_number);
        break;
      }

      delete log_;
      delete logfile_;
      logfile_ = lfile;
      logfile_number_ = new_log_number;
      log_ = new log::Writer(lfile);
      imm_ = mem_;
      has_imm_.store(true, std::memory_order_release);
      mem_ = new MemTable(internal_comparator_);
      mem_->Ref();
      force = false;  // Do not force another compaction if have room
      MaybeScheduleCompaction();
    }
  }
  return s;
}
```

在 While 循环中，会依次检查：

1. 是否有后台错误，如果是就退出；
2. 是否允许延迟、并且现有的 L0 文件稍多，如果是则释放锁、等待 1 毫秒、再等待锁，并且不允许再次等待；
3. 是否非强制模式、并且内存数据库大小没有超出阈值，如果是就还有空间、可以退出了；

4. 是否 imm_ 还在 Compact 中，如果是就等待 Compact 完成（条件变量）；
5. 是否现有的 L0 文件太多，如果是就等待 Compact 完成（条件变量）；
6. 以上都不是，那就说明当前的 mem_ 满了，把它丢给 imm_，再新建一个 mem_，触发 Compaction。

除了第 1 步和第 3 步，其他步骤完成后均重新进入循环，最终从 1 或 3 中退出。第 2 步中当 L0 文件数目达到 kL0_SlowdownWritesTrigger 限制时，对每个写入请求等待 1 毫秒，也就减缓了写入的速度；第 5 步中当 L0 文件数目达到 kL0_StopWritesTrigger 限制时，就强制等待当前的 Compact 完成，直到 L0 文件数目小于限制时，下次循环才能到达第 6 步。

继续看 MaybeScheduleCompaction 及其后续过程：

```
void DBImpl::MaybeScheduleCompaction() {
  mutex_.AssertHeld();
  if (background_compaction_scheduled_) {
    // Already scheduled
  } else if (shutting_down_.load(std::memory_order_acquire)) {
    // DB is being deleted; no more background compactions
  } else if (!bg_error_.ok()) {
    // Already got an error; no more changes
  } else if (imm_ == nullptr && manual_compaction_ == nullptr &&
             !versions_->NeedsCompaction()) {
    // No work to be done
  } else {
    background_compaction_scheduled_ = true;
    env_->Schedule(&DBImpl::BGWork, this);
  }
```

```
}

void DBImpl::BGWork(void* db) {
  reinterpret_cast<DBImpl*>(db)->BackgroundCall();
}


void DBImpl::BackgroundCall() {
  MutexLock l(&mutex_);
  assert(background_compaction_scheduled_);
  if (shutting_down_.load(std::memory_order_acquire)) {
    // No more background work when shutting down.
  } else if (!bg_error_.ok()) {
    // No more background work after a background error.
  } else {
    BackgroundCompaction();
  }

  background_compaction_scheduled_ = false;

  // Previous compaction may have produced too many files in a level,
  // so reschedule another compaction if needed.
  MaybeScheduleCompaction();
  background_work_finished_signal_.SignalAll();
}

void DBImpl::BackgroundCompaction() {
  mutex_.AssertHeld();

  if (imm_ != nullptr) {
    CompactMemTable();
```

```
    return;
  }
  // ...
}


void DBImpl::CompactMemTable() {
  mutex_.AssertHeld();
  assert(imm_ != nullptr);

  // Save the contents of the memtable as a new Table
  VersionEdit edit;
  Version* base = versions_->current();
  base->Ref();
  Status s = WriteLevel0Table(imm_, &edit, base);
  base->Unref();

  if (s.ok() && shutting_down_.load(std::memory_order_acquire)) {
    s = Status::IOError("Deleting DB during memtable compaction");
  }

  // Replace immutable memtable with the generated Table
  if (s.ok()) {
    edit.SetPrevLogNumber(0);
    edit.SetLogNumber(logfile_number_);  // Earlier logs no longer needed
    s = versions_->LogAndApply(&edit, &mutex_);
  }

  if (s.ok()) {
    // Commit to the new state
    imm_->Unref();
```

```
    imm_ = nullptr;
    has_imm_.store(false, std::memory_order_release);
    DeleteObsoleteFiles();
  } else {
    RecordBackgroundError(s);

  }
}

Status DBImpl::WriteLevel0Table(MemTable* mem, VersionEdit* edit,
                                Version* base) {
  mutex_.AssertHeld();
  const uint64_t start_micros = env_->NowMicros();
  FileMetaData meta;
  meta.number = versions_->NewFileNumber();
  pending_outputs_.insert(meta.number);
  Iterator* iter = mem->NewIterator();
  Log(options_.info_log, "Level-0 table #%llu: started",
      (unsigned long long)meta.number);

  Status s;
  {
    mutex_.Unlock();
    s = BuildTable(dbname_, env_, options_, table_cache_, iter, &meta);
    mutex_.Lock();
  }

  Log(options_.info_log, "Level-0 table #%llu: %lld bytes %s",
      (unsigned long long)meta.number, (unsigned long long)meta.file_size,
      s.ToString().c_str());
  delete iter;
```

```
  pending_outputs_.erase(meta.number);

  // Note that if file_size is zero, the file has been deleted and
  // should not be added to the manifest.
  int level = 0;

  if (s.ok() && meta.file_size > 0) {
    const Slice min_user_key = meta.smallest.user_key();
    const Slice max_user_key = meta.largest.user_key();
    if (base != nullptr) {
      level = base->PickLevelForMemTableOutput(min_user_key, max_user_key);
    }
    edit->AddFile(level, meta.number, meta.file_size, meta.smallest,
                  meta.largest);
  }

  CompactionStats stats;
  stats.micros = env_->NowMicros() - start_micros;
  stats.bytes_written = meta.file_size;
  stats_[level].Add(stats);
  return s;
}
```

MaybeScheduleCompaction 经过一些判断，最后调用 env_->Schedule 创建一个 Detached 的线程启动 DBImpl::BGWork，继而调用 DBImpl::BackgroundCall，继而调用 DBImpl::BackgroundCompaction。该函数在 imm_ 非空的情况下，直接执行 CompactMemTable 返回。CompactMemTable 的核心操作为 WriteLevel0Table(imm_, &edit, base)，将会把 imm_ 中的数据写到 Level0 Table 里。这是我们第一次碰触到 LevelDB 名字中 Level 的边缘。

如果先不看 Version 相关的部分， WriteLevel0Table 会调用 BuildTable 建 Sorted Table。该函数的实现位于 db/builder.cc：

```
#include "db/builder.h"

#include "db/dbformat.h"
#include "db/filename.h"
#include "db/table_cache.h"
#include "db/version_edit.h"
#include "leveldb/db.h"
#include "leveldb/env.h"
#include "leveldb/iterator.h"

namespace leveldb {

Status BuildTable(const std::string& dbname, Env* env, const Options& options,
                  TableCache* table_cache, Iterator* iter, FileMetaData* meta) {
  Status s;
  meta->file_size = 0;
  iter->SeekToFirst();

  std::string fname = TableFileName(dbname, meta->number);
  if (iter->Valid()) {
    WritableFile* file;
    s = env->NewWritableFile(fname, &file);
    if (!s.ok()) {
      return s;
    }
```

```
}

TableBuilder* builder = new TableBuilder(options, file);
meta->smallest.DecodeFrom(iter->key());
for (; iter->Valid(); iter->Next()) {
  Slice key = iter->key();
  meta->largest.DecodeFrom(key);
  builder->Add(key, iter->value());
}

// Finish and check for builder errors
s = builder->Finish();
if (s.ok()) {
  meta->file_size = builder->FileSize();
  assert(meta->file_size > 0);
}
delete builder;

// Finish and check for file errors
if (s.ok()) {
  s = file->Sync();
}
if (s.ok()) {
  s = file->Close();
}
delete file;
file = nullptr;

if (s.ok()) {
  // Verify that the table is usable
  Iterator* it = table_cache->NewIterator(ReadOptions(), meta->number,
```

```
                                              meta->file_size);
      s = it->status();
      delete it;
    }
  }


  // Check for input iterator errors
  if (!iter->status().ok()) {
    s = iter->status();
  }

  if (s.ok() && meta->file_size > 0) {
    // Keep it
  } else {
    env->DeleteFile(fname);
  }
  return s;
}

}  // namespace leveldb
```

核心操作是遍历迭代器，将键值对加入到一个 TableBuilder 对象里，同时维护 meta 信息，包括 smallest、 largest 和 file_size。

2. Block Builder

在介绍 TableBuilder 之前，需要先介绍它的依赖 BlockBuilder。先看 table/block_builder.h：

```
class BlockBuilder {
 public:
  explicit BlockBuilder(const Options* options);


  BlockBuilder(const BlockBuilder&) = delete;
  BlockBuilder& operator=(const BlockBuilder&) = delete;

  // Reset the contents as if the BlockBuilder was just constructed.
  void Reset();

  // REQUIRES: Finish() has not been called since the last call to Reset().
  // REQUIRES: key is larger than any previously added key
  void Add(const Slice& key, const Slice& value);

  // Finish building the block and return a slice that refers to the
  // block contents.  The returned slice will remain valid for the
  // lifetime of this builder or until Reset() is called.
  Slice Finish();

  // Returns an estimate of the current (uncompressed) size of the block
  // we are building.
  size_t CurrentSizeEstimate() const;

  // Return true iff no entries have been added since the last Reset()
  bool empty() const { return buffer_.empty(); }

 private:
  const Options* options_;
  std::string buffer_;              // Destination buffer
```

```
  std::vector<uint32_t> restarts_;  // Restart points
  int counter_;                     // Number of entries emitted since restart
  bool finished_;                   // Has Finish() been called?
  std::string last_key_;
};
```

接口看不出什么，继续看实现 table/block_builder.cc：

```
// BlockBuilder generates blocks where keys are prefix-compressed:
//
// When we store a key, we drop the prefix shared with the previous
// string.  This helps reduce the space requirement significantly.
// Furthermore, once every K keys, we do not apply the prefix
// compression and store the entire key.  We call this a "restart
// point".  The tail end of the block stores the offsets of all of the
// restart points, and can be used to do a binary search when looking
// for a particular key.  Values are stored as-is (without compression)
// immediately following the corresponding key.
//
// An entry for a particular key-value pair has the form:
//     shared_bytes: varint32
//     unshared_bytes: varint32
//     value_length: varint32
//     key_delta: char[unshared_bytes]
//     value: char[value_length]
// shared_bytes == 0 for restart points.
//
// The trailer of the block has the form:
//     restarts: uint32[num_restarts]
//     num_restarts: uint32
```

```
// restarts[i] contains the offset within the block of the ith restart point.

#include "table/block_builder.h"

#include <assert.h>

#include <algorithm>

#include "leveldb/comparator.h"
#include "leveldb/options.h"
#include "util/coding.h"

namespace leveldb {

BlockBuilder::BlockBuilder(const Options* options)
    : options_(options), restarts_(), counter_(0), finished_(false) {
  assert(options->block_restart_interval >= 1);
  restarts_.push_back(0);  // First restart point is at offset 0
}

void BlockBuilder::Reset() {
  buffer_.clear();
  restarts_.clear();
  restarts_.push_back(0);  // First restart point is at offset 0
  counter_ = 0;
  finished_ = false;
  last_key_.clear();
}

size_t BlockBuilder::CurrentSizeEstimate() const {
  return (buffer_.size() +                        // Raw data buffer
```

```
          restarts_.size() * sizeof(uint32_t) +  // Restart array
          sizeof(uint32_t));                      // Restart array length
}

Slice BlockBuilder::Finish() {

  // Append restart array
  for (size_t i = 0; i < restarts_.size(); i++) {
    PutFixed32(&buffer_, restarts_[i]);
  }
  PutFixed32(&buffer_, restarts_.size());
  finished_ = true;
  return Slice(buffer_);
}

void BlockBuilder::Add(const Slice& key, const Slice& value) {
  Slice last_key_piece(last_key_);
  assert(!finished_);
  assert(counter_ <= options_->block_restart_interval);
  assert(buffer_.empty()  // No values yet?
         || options_->comparator->Compare(key, last_key_piece) > 0);
  size_t shared = 0;
  if (counter_ < options_->block_restart_interval) {
    // See how much sharing to do with previous string
    const size_t min_length = std::min(last_key_piece.size(), key.size());
    while ((shared < min_length) && (last_key_piece[shared] == key[shared])) {
      shared++;
    }
  } else {
    // Restart compression
    restarts_.push_back(buffer_.size());
```

```
    counter_ = 0;
  }
  const size_t non_shared = key.size() - shared;

  // Add "<shared><non_shared><value_size>" to buffer_

  PutVarint32(&buffer_, shared);
  PutVarint32(&buffer_, non_shared);
  PutVarint32(&buffer_, value.size());

  // Add string delta to buffer_ followed by value
  buffer_.append(key.data() + shared, non_shared);
  buffer_.append(value.data(), value.size());

  // Update state
  last_key_.resize(shared);
  last_key_.append(key.data() + shared, non_shared);
  assert(Slice(last_key_) == key);
  counter_++;
}

}  // namespace leveldb
```

文件头部的英文注释写得十分详细。为了节约空间， LevelDB 存储键值对时，会利用局部性原理，将省略掉键与上一个键的共同前缀部分。存储一键值对时，按照下面的方式存储：

```
shared_bytes: varint32          # Key 与上一个键共享的长度
unshared_bytes: varint32        # Key 非共享长度
value_length: varint32          # Value 的长度
```

```
value_length: varint32          # value 的长度
key_delta: char[unshared_bytes] # Key 非共享部分
value: char[value_length]       # Value
```

省略前缀节约了空间，但就无法方便地执行二分查找了。LevelDB 的解决方案是每隔 K=16 个键值队设定一个复活点（没错想想超级玛丽的复活点），复活点上将完整存储键、不进行前缀共享。 Block 的结尾存储所有复活点的位置，查找时先在复活点上做二分查找，确定大概位置后再遍历恢复 Key 后对比。为了更好地表达这件事，这里画个表：

Original Key	Shared	Unshared	Key Delta
Abel	**0**	4	Abel
Abner	2	3	ner
Abram	2	3	ram
Adam	1	3	dam
Adelbert	**0**	8	Adelbert
Adrian	2	4	rian
Alan	1	3	lan
Albert	2	4	bert

这里使用的复活点间隔 K=4 。使用该方案每个条目需要额外至少一个字节存储共享的长度 shared ，但同时会节约 shared 个字节。上表中整体会节约 2+2+1+2+1+2-8=2 个字节。当需要读第八个键时，需要先读取前面最近的复活点 Adelbert ，根据共享前缀恢复第六个键 Ad + rian ，再恢复第七个键 A + lan ，最后恢复 Al + bert ，所以这里的 K 并不能

六个键 Ad：nan，再恢复第七个键 A：lan，最后恢复 Al：bert。所以这里的 K 并不能取得过大。

BlockBuilder::Finish 时，会将所有的复活点位置及复活点数量写到 buffer_ 里，以实现解析。对应的解析代码位于 table/block.cc，后续文章会再分析。

3. Table Builder

TableBuilder 的接口位于 include/leveldb/table_builder.h：

```
// TableBuilder provides the interface used to build a Table
// (an immutable and sorted map from keys to values).
//
// Multiple threads can invoke const methods on a TableBuilder without
// external synchronization, but if any of the threads may call a
// non-const method, all threads accessing the same TableBuilder must use
// external synchronization.

#ifndef STORAGE_LEVELDB_INCLUDE_TABLE_BUILDER_H_
#define STORAGE_LEVELDB_INCLUDE_TABLE_BUILDER_H_

#include <stdint.h>

#include "leveldb/export.h"
#include "leveldb/options.h"
#include "leveldb/status.h"

namespace leveldb {

class BlockBuilder;
```

```
class BlockHandle;
class WritableFile;

class LEVELDB_EXPORT TableBuilder {
 public:
  // Create a builder that will store the contents of the table it is
  // building in *file.  Does not close the file.  It is up to the
  // caller to close the file after calling Finish().
  TableBuilder(const Options& options, WritableFile* file);

  TableBuilder(const TableBuilder&) = delete;
  TableBuilder& operator=(const TableBuilder&) = delete;

  // REQUIRES: Either Finish() or Abandon() has been called.
  ~TableBuilder();

  // Change the options used by this builder.  Note: only some of the
  // option fields can be changed after construction.  If a field is
  // not allowed to change dynamically and its value in the structure
  // passed to the constructor is different from its value in the
  // structure passed to this method, this method will return an error
  // without changing any fields.
  Status ChangeOptions(const Options& options);

  // Add key,value to the table being constructed.
  // REQUIRES: key is after any previously added key according to comparator.
  // REQUIRES: Finish(), Abandon() have not been called
  void Add(const Slice& key, const Slice& value);

  // Advanced operation: flush any buffered key/value pairs to file.
  // Can be used to ensure that two adjacent entries never live in
```

```
  // the same data block.  Most clients should not need to use this method.
  // REQUIRES: Finish(), Abandon() have not been called
  void Flush();

  // Return non-ok iff some error has been detected.

  Status status() const;

  // Finish building the table.  Stops using the file passed to the
  // constructor after this function returns.
  // REQUIRES: Finish(), Abandon() have not been called
  Status Finish();

  // Indicate that the contents of this builder should be abandoned.  Stops
  // using the file passed to the constructor after this function returns.
  // If the caller is not going to call Finish(), it must call Abandon()
  // before destroying this builder.
  // REQUIRES: Finish(), Abandon() have not been called
  void Abandon();

  // Number of calls to Add() so far.
  uint64_t NumEntries() const;

  // Size of the file generated so far.  If invoked after a successful
  // Finish() call, returns the size of the final generated file.
  uint64_t FileSize() const;

 private:
  bool ok() const { return status().ok(); }
  void WriteBlock(BlockBuilder* block, BlockHandle* handle);
  void WriteRawBlock(const Slice& data, CompressionType, BlockHandle* handle);
```

```
  struct Rep;
  Rep* rep_;
};


}  // namespace leveldb

#endif  // STORAGE_LEVELDB_INCLUDE_TABLE_BUILDER_H_
```

接口代码里，LevelDB 都提供了详细的英文注释，不再赘述。这里刚好遇到 pImpl 范式，多说一点。对 C++ 来说，ABI 始终是一个痛点。Zero-Overhead 的 OOP，会把这部分 Overhead 加到编译期上。当一个类的数据成员发生变化时，会影响该类对象的大小和内存布局，进而导致所有依赖该类的文件需要重新编译。如果使用动态链接库，当版本升级、类发生变化时，由于 ABI 不兼容会导致无法直接替换动态链接库文件完成升级。而使用 pImpl 范式，将类的成员和类的声明隔离开，使用一个指针指向存储成员的对象 rep_ 上。当成员发生变化时，也不需要重新编译，并且保证了 ABI 的稳定。该方案的缺点是访问成员时增加了一次指针寻址，不过瑕不掩瑜，和智能指针一样耗费微小的代价、带来极大的提升。LevelDB 的源代码中大量使用了该范式。接着看 TableBuilder 的实现 table/table_builder.cc：

```
struct TableBuilder::Rep {
  Rep(const Options& opt, WritableFile* f)
      : options(opt),
        index_block_options(opt),
        file(f),
        offset(0),
```

```
      data_block(&options),
      index_block(&index_block_options),
      num_entries(0),
      closed(false),
      filter_block(opt.filter_policy == nullptr

                       ? nullptr
                       : new FilterBlockBuilder(opt.filter_policy)),
      pending_index_entry(false) {
  index_block_options.block_restart_interval = 1;
}

Options options;
Options index_block_options;
WritableFile* file;
uint64_t offset;
Status status;
BlockBuilder data_block;
BlockBuilder index_block;
std::string last_key;
int64_t num_entries;
bool closed;  // Either Finish() or Abandon() has been called.
FilterBlockBuilder* filter_block;

// We do not emit the index entry for a block until we have seen the
// first key for the next data block.  This allows us to use shorter
// keys in the index block.  For example, consider a block boundary
// between the keys "the quick brown fox" and "the who".  We can use
// "the r" as the key for the index block entry since it is >= all
// entries in the first block and < all entries in subsequent
// blocks.
```

```
  //
  // Invariant: r->pending_index_entry is true only if data_block is empty.
  bool pending_index_entry;
  BlockHandle pending_handle;  // Handle to add to index block


  std::string compressed_output;
};

TableBuilder::TableBuilder(const Options& options, WritableFile* file)
    : rep_(new Rep(options, file)) {
  if (rep_->filter_block != nullptr) {
    rep_->filter_block->StartBlock(0);
  }
}

TableBuilder::~TableBuilder() {
  assert(rep_->closed);  // Catch errors where caller forgot to call Finish()
  delete rep_->filter_block;
  delete rep_;
}

Status TableBuilder::status() const { return rep_->status; }

uint64_t TableBuilder::NumEntries() const { return rep_->num_entries; }

uint64_t TableBuilder::FileSize() const { return rep_->offset; }
```

首先是 TableBuilder::Rep 的定义。该定义位于 .cc 文件内，即使发生修改也仅仅会重新编译文件本身。 TableBuilder 的构造函数中会初始化 rep_ 对象，并且在析构函数中

将 rep_ 删除。而需要访问成员变量时，都需要使用 rep_-> 进行多一次的寻址。

TableBuilder::rep_ 包含两个 BlockBuilder 对象，分别用来存储键值对数据和元数据。接下来是 TableBuilder::Add 等函数的实现。由于依赖的原因，这里需要先看

table/format.h：

```
class Block;
class RandomAccessFile;
struct ReadOptions;

// BlockHandle is a pointer to the extent of a file that stores a data
// block or a meta block.
class BlockHandle {
 public:
  // Maximum encoding length of a BlockHandle
  enum { kMaxEncodedLength = 10 + 10 };

  BlockHandle();

  // The offset of the block in the file.
  uint64_t offset() const { return offset_; }
  void set_offset(uint64_t offset) { offset_ = offset; }

  // The size of the stored block
  uint64_t size() const { return size_; }
  void set_size(uint64_t size) { size_ = size; }

  void EncodeTo(std::string* dst) const;
```

```
  Status DecodeFrom(Slice* input);

 private:
  uint64_t offset_;
  uint64_t size_;
};

// Footer encapsulates the fixed information stored at the tail
// end of every table file.
class Footer {
 public:
  // Encoded length of a Footer.  Note that the serialization of a
  // Footer will always occupy exactly this many bytes.  It consists
  // of two block handles and a magic number.
  enum { kEncodedLength = 2 * BlockHandle::kMaxEncodedLength + 8 };

  Footer() = default;

  // The block handle for the metaindex block of the table
  const BlockHandle& metaindex_handle() const { return metaindex_handle_; }
  void set_metaindex_handle(const BlockHandle& h) { metaindex_handle_ = h; }

  // The block handle for the index block of the table
  const BlockHandle& index_handle() const { return index_handle_; }
  void set_index_handle(const BlockHandle& h) { index_handle_ = h; }

  void EncodeTo(std::string* dst) const;
  Status DecodeFrom(Slice* input);

 private:
  BlockHandle metaindex_handle_;
```

```
  BlockHandle index_handle_;
};

// kTableMagicNumber was picked by running
//    echo http://code.google.com/p/leveldb/ | sha1sum

// and taking the leading 64 bits.
static const uint64_t kTableMagicNumber = 0xdb4775248b80fb57ull;

// 1-byte type + 32-bit crc
static const size_t kBlockTrailerSize = 5;

struct BlockContents {
  Slice data;           // Actual contents of data
  bool cachable;        // True iff data can be cached
  bool heap_allocated;  // True iff caller should delete[] data.data()
};

// Read the block identified by "handle" from "file".  On failure
// return non-OK.  On success fill *result and return OK.
Status ReadBlock(RandomAccessFile* file, const ReadOptions& options,
                 const BlockHandle& handle, BlockContents* result);

// Implementation details follow.  Clients should ignore,

inline BlockHandle::BlockHandle()
    : offset_(~static_cast<uint64_t>(0)), size_(~static_cast<uint64_t>(0)) {}
```

BlockBuilder 将 Block 的数据写到字节流中，当需要进行解析时，必须知道其起始位置和长度，以便于读取复活点信息。故这里引入了 BlockHandle，存储 Block 数据所在的

位置 offset_ 和长度 size_ 。其 EncodeTo 和 DecodeFrom 函数的实现都非常简单，不再赘述。有了 BlockHandle ，就可以读取对应的 Block 数据， Footer 就用来存储两个 BlockHandle 。最后来看下 TableBuilder 的核心部分：

```
void TableBuilder::Add(const Slice& key, const Slice& value) {
  Rep* r = rep_;
  assert(!r->closed);
  if (!ok()) return;
  if (r->num_entries > 0) {
    assert(r->options.comparator->Compare(key, Slice(r->last_key)) > 0);
  }

  if (r->pending_index_entry) {
    assert(r->data_block.empty());
    r->options.comparator->FindShortestSeparator(&r->last_key, key);
    std::string handle_encoding;
    r->pending_handle.EncodeTo(&handle_encoding);
    r->index_block.Add(r->last_key, Slice(handle_encoding));
    r->pending_index_entry = false;
  }

  if (r->filter_block != nullptr) {
    r->filter_block->AddKey(key);
  }

  r->last_key.assign(key.data(), key.size());
  r->num_entries++;
  r->data_block.Add(key, value);
```

```
  const size_t estimated_block_size = r->data_block.CurrentSizeEstimate();
  if (estimated_block_size >= r->options.block_size) {
    Flush();
  }
}

void TableBuilder::Flush() {
  Rep* r = rep_;
  assert(!r->closed);
  if (!ok()) return;
  if (r->data_block.empty()) return;
  assert(!r->pending_index_entry);
  WriteBlock(&r->data_block, &r->pending_handle);
  if (ok()) {
    r->pending_index_entry = true;
    r->status = r->file->Flush();
  }
  if (r->filter_block != nullptr) {
    r->filter_block->StartBlock(r->offset);
  }
}

void TableBuilder::WriteBlock(BlockBuilder* block, BlockHandle* handle) {
  // File format contains a sequence of blocks where each block has:
  //    block_data: uint8[n]
  //    type: uint8
  //    crc: uint32
  assert(ok());
  Rep* r = rep_;
  Slice raw = block->Finish();
```

```
  Slice block_contents;
  CompressionType type = r->options.compression;
  // TODO(postrelease): Support more compression options: zlib?
  switch (type) {

    case kNoCompression:
      block_contents = raw;
      break;

    case kSnappyCompression: {
      std::string* compressed = &r->compressed_output;
      if (port::Snappy_Compress(raw.data(), raw.size(), compressed) &&
          compressed->size() < raw.size() - (raw.size() / 8u)) {
        block_contents = *compressed;
      } else {
        // Snappy not supported, or compressed less than 12.5%, so just
        // store uncompressed form
        block_contents = raw;
        type = kNoCompression;
      }
      break;
    }
  }
  WriteRawBlock(block_contents, type, handle);
  r->compressed_output.clear();
  block->Reset();
}

void TableBuilder::WriteRawBlock(const Slice& block_contents,
                                 CompressionType type, BlockHandle* handle) {
```

```
  Rep* r = rep_;
  handle->set_offset(r->offset);
  handle->set_size(block_contents.size());
  r->status = r->file->Append(block_contents);
  if (r->status.ok()) {

    char trailer[kBlockTrailerSize];
    trailer[0] = type;
    uint32_t crc = crc32c::Value(block_contents.data(), block_contents.size());
    crc = crc32c::Extend(crc, trailer, 1);  // Extend crc to cover block type
    EncodeFixed32(trailer + 1, crc32c::Mask(crc));
    r->status = r->file->Append(Slice(trailer, kBlockTrailerSize));
    if (r->status.ok()) {
      r->offset += block_contents.size() + kBlockTrailerSize;
    }
  }
}

Status TableBuilder::Finish() {
  Rep* r = rep_;
  Flush();
  assert(!r->closed);
  r->closed = true;

  BlockHandle filter_block_handle, metaindex_block_handle, index_block_handle;

  // Write filter block
  if (ok() && r->filter_block != nullptr) {
    WriteRawBlock(r->filter_block->Finish(), kNoCompression,
                  &filter_block_handle);
  }
```

```
// Write metaindex block
if (ok()) {
  BlockBuilder meta_index_block(&r->options);
  if (r->filter_block != nullptr) {

    // Add mapping from "filter.Name" to location of filter data
    std::string key = "filter.";
    key.append(r->options.filter_policy->Name());
    std::string handle_encoding;
    filter_block_handle.EncodeTo(&handle_encoding);
    meta_index_block.Add(key, handle_encoding);
  }

  // TODO(postrelease): Add stats and other meta blocks
  WriteBlock(&meta_index_block, &metaindex_block_handle);
}

// Write index block
if (ok()) {
  if (r->pending_index_entry) {
    r->options.comparator->FindShortSuccessor(&r->last_key);
    std::string handle_encoding;
    r->pending_handle.EncodeTo(&handle_encoding);
    r->index_block.Add(r->last_key, Slice(handle_encoding));
    r->pending_index_entry = false;
  }
  WriteBlock(&r->index_block, &index_block_handle);
}

// Write footer
```

```
  if (ok()) {
    Footer footer;
    footer.set_metaindex_handle(metaindex_block_handle);
    footer.set_index_handle(index_block_handle);
    std::string footer_encoding;

    footer.EncodeTo(&footer_encoding);
    r->status = r->file->Append(footer_encoding);
    if (r->status.ok()) {
      r->offset += footer_encoding.size();
    }
  }
  return r->status;
}
```

每次执行 Add 操作时，会将键值对写入 data_block_ 中；当 data_block_ 的大小超过阈值时（默认 4KB），将会把该 Block 写入文件，并将 Block 的 last_key 作为 Key，将 offset 和 size 信息作为 Value 加入到 index_block_ 中。当执行 TableBuilder::Finish 操作时，会将 index_block_ 也写入文件，其对应的 index_block_handle_ 写到最后的 Footer 里。这样就完成了 Sorted Table 文件的构建。

反推回来，当需要读取一个 Sorted Table 文件时，首先读取文件末尾的 Footer，根据存储的 Index Block Handle 可以读取到每个 Data Block 对应的 Data Block Handler 信息，进而读取到对应的 Data Block。读取的详细过程将在下一篇博文中分析。

总结

TableBuilder 建好 Sorted Table 后，存储为后缀 .ldb 的物理文件，并且将其加入版本

管理中。Sorted Table 建好后是不可修改的，进而可以很好地支持并发访问，并且对缓存友好。

题外话：Sorted Table 本来存储的文件文件格式是 .sst，但后来由于 Windows 系统的原因，LevelDB 将后缀改为 .ldb，不过仍然保持着对 .sst 的兼容。

4 comments *– powered by giscus*

Oldest Newest

IceHe Aug 17, 2019

很棒