LevelDB笔记前述：编译运行

2024年7月9日 19:44

首先在README.md中介绍了大致的信息。

快速开始：

```
mkdir -p build && cd build
cmake -DCMAKE_BUILD_TYPE=Release .. && cmake --build .
```

cmake生成适用于特定平台和编译器的构建系统配置文件，例如生成 Makefile 或其他构建系统所需的文件。它处理跨平台的配置和依赖管理等高级任务。

make本身通常不是跨平台的，它依赖于特定的Makefile格式和规则。根据已经存在的 Makefile 来执行具体的编译和链接操作，以构建项目。

c++环境中cmake的工作过程：

1. 编写CMakeLists.txt文件
   这是 cmake 的配置脚本，开发者在其中定义项目的各种信息，如项目名称、源文件列表、目标类型（可执行文件、库等）、依赖库、编译选项等。
2. 运行cmake命令
   当执行 cmake 命令时，它会读取当前目录及其子目录中的 CMakeLists.txt 文件。
   解析其中的指令和设置，并进行一系列的检查和计算。
3. 处理依赖关系
   检查项目所依赖的外部库和头文件。
   确定如何找到和链接这些依赖项，包括查找系统库和第三方库的路径。
4. 生成构建系统文件
   根据项目的配置和目标平台，cmake 生成相应的构建系统文件，如 Makefile（在 Unix/Linux 系统中）、Visual Studio 项目文件（在 Windows 中）等。
5. 构建项目
   利用生成的构建系统文件，使用对应的构建工具（如 make 、ninja 或 Visual Studio 等）进行编译和链接，生成最终的可执行文件或库。

下面是leveldb中的CMakeLists.txt文件（部分）：

cmake版本与项目名称以及版本与其使用的语言：

```
cmake_minimum_required(VERSION 3.9)
project(leveldb VERSION 1.23.0 LANGUAGES C CXX)
```

C和C++标准如果没有被覆盖的话，执行set语句

```
# C standard can be overridden when this is used as a sub-project.
if(NOT CMAKE_C_STANDARD)
  # This project can use C11, but will gracefully decay down to C89.
  set(CMAKE_C_STANDARD 11)
  set(CMAKE_C_STANDARD_REQUIRED OFF)
  set(CMAKE_C_EXTENSIONS OFF)
endif(NOT CMAKE_C_STANDARD)
# C++ standard can be overridden when this is used as a sub-project.
```

```
if(NOT CMAKE_CXX_STANDARD)
  # This project requires C++11.
  set(CMAKE_CXX_STANDARD 11)
  set(CMAKE_CXX_STANDARD_REQUIRED ON)
  set(CMAKE_CXX_EXTENSIONS OFF)
endif(NOT CMAKE_CXX_STANDARD)
```

检查平台是Windows还是Posix，定义一个NAME来表示啥平台

```
if (WIN32)
  set(LEVELDB_PLATFORM_NAME LEVELDB_PLATFORM_WINDOWS)
  # TODO(cmumford): Make UNICODE configurable for Windows.
  add_definitions(-D_UNICODE -DUNICODE)
else (WIN32)
  set(LEVELDB_PLATFORM_NAME LEVELDB_PLATFORM_POSIX)
endif (WIN32)
```

。。。之后再补

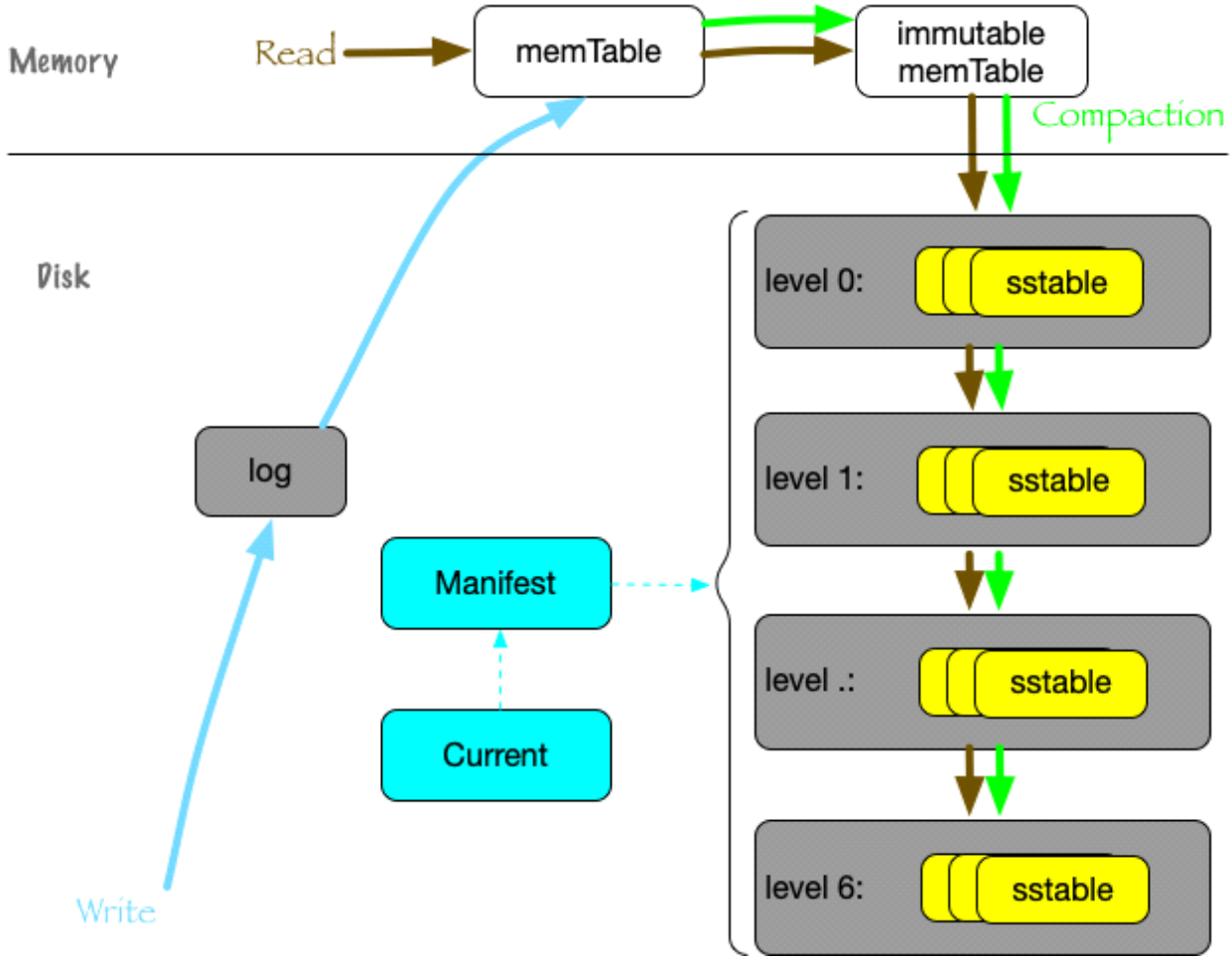
LeveDB笔记0：架构与概述

2024年9月23日 10:03

【深度知识】LevelDB从入门到原理详解-腾讯云开发者社区-腾讯云 (tencent.com)

Archive - Ying (izualzhy.cn)

leveldb-handbook — leveldb-handbook 文档



leveldb 主要由以下组件组成：

1. log: write-ahead logging是数据库的一种常见手段，数据按照 ->log->mem 的顺序更新，由于数据已经持久化到磁盘，因此即使进程异常也能够保证数据的完整性，同时这里是追加写，因此写性能极高。
2. memtable: 最近写入的 key-value 数据，内存存储，读取数据首先从这里查找。
3. immutable memtable: 为了限制内存大小，当 memtable 达到一定大小后，会转换为immutable memtable。后台线程会把immutable memtable 持久化到硬盘，持久化的文件称为 level-0 sstable，这个过程称为 minor compact.
4. sstable: 由上层(or上上层)的 sstable 合并成新的sstable，并写入到下一层，这个过程称为 major compact，因此层数越小，数据越新，层数越大，数据越久远。
5. manifest: read/compaction 过程可能是同时进行的，因此需要能记录对应的文件集合，manifest就是起到记录各阶段文件集合信息的，为了更快速的查找，可能还会记录一些附加信息，例如文件大小、最大最小 key 等。

LevelDB笔记1：接口

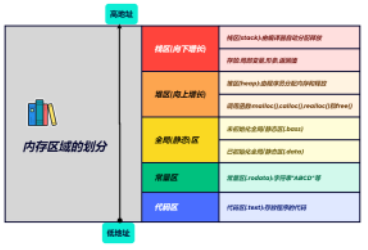
2024年7月26日　11:16

接口：主要是DB这个类

```
namespace leveldb {
static const int kMajorVersion = 1;
static const int kMinorVersion = 23;
struct Options;
struct ReadOptions;
struct WriteOptions;
class WriteBatch;

class LEVELDB_EXPORT Snapshot {
 protected:
  virtual ~Snapshot();
};
// A range of keys
struct LEVELDB_EXPORT Range {
  Range() = default;
  Range(const Slice& s, const Slice& l) : start(s), limit(l) {}
  Slice start;  // Included in the range
  Slice limit;  // Not included in the range
};


class LEVELDB_EXPORT DB {
 public:
  static Status Open(const Options& options, const std::string& name,
                     DB** dbptr);
  //构造函数与析构函数
  DB() = default;
  DB(const DB&) = delete;
  DB& operator=(const DB&) = delete;
  virtual ~DB();

  virtual Status Put(const WriteOptions& options, const Slice& key,
                     const Slice& value) = 0;

  virtual Status Delete(const WriteOptions& options, const Slice& key) = 0;

  virtual Status Write(const WriteOptions& options, WriteBatch* updates) = 0;

  virtual Status Get(const ReadOptions& options, const Slice& key,
                     std::string* value) = 0;

  virtual Iterator* NewIterator(const ReadOptions& options) = 0;

  virtual const Snapshot* GetSnapshot() = 0;
  virtual void ReleaseSnapshot(const Snapshot* snapshot) = 0;
  virtual bool GetProperty(const Slice& property, std::string* value) = 0;
  virtual void GetApproximateSizes(const Range* range, int n,
                                   uint64_t* sizes) = 0;
  virtual void CompactRange(const Slice* begin, const Slice* end) = 0;
};
LEVELDB_EXPORT Status DestroyDB(const std::string& name,
                                const Options& options);
LEVELDB_EXPORT Status RepairDB(const std::string& dbname,
                               const Options& options);
}  // namespace leveldb
```

这个地方是DB这个类中的静态成员函数
在类中的静态成员变量属于类，该类的所有实例化后的对象共享静态成员变量。注意类的静态变量必须在类内声明，类外初始化。

提到静态成员不得不知道内存分布了：



全局静态区：分为bass段和data段
data段存放初始化的全局变量和静态变量；
bass段存放未初始化的全局变量和静态变量；
程序执行结束的时候才会自动释放；
存储在静态数据区的变量会在程序刚开始运行时就完成初始化，也是唯一的一次初始化。

这里将默认的构造函数显式声明出来，

下面来看Open函数：

```
Status DB::Open(const Options& options, const std::string& dbname, DB** dbptr)
 {
  *dbptr = nullptr;
  DBImpl* impl = new DBImpl(options, dbname);
  impl->mutex_.Lock();          //加锁，以确保线程安全
  VersionEdit edit;
  // Recover handles create_if_missing, error_if_exists
  bool save_manifest = false;
  Status s = impl->Recover(&edit, &save_manifest);
  if (s.ok() && impl->mem_ == nullptr) {
    // Create new log and a corresponding memtable.
    uint64_t new_log_number = impl->versions_->NewFileNumber();
    WritableFile* lfile;
    s = options.env->NewWritableFile(LogFileName(dbname, new_log_number),
                                     &lfile);
    if (s.ok()) {
      edit.SetLogNumber(new_log_number);
      impl->logfile_ = lfile;
      impl->logfile_number_ = new_log_number;
      impl->log_ = new log::Writer(lfile);
      impl->mem_ = new MemTable(impl->internal_comparator_);
      impl->mem_->Ref();
    }
  }
  if (s.ok() && save_manifest) {
    edit.SetPrevLogNumber(0);  // No older logs needed after recovery.
    edit.SetLogNumber(impl->logfile_number_);
    s = impl->versions_->LogAndApply(&edit, &impl->mutex_);
  }
```

return的是Status，这个函数打开一个数据库，没有的话新建一个数据库；
dbptr：指向DB类指针的指针，用以返回打开的数据库实例

DBImpl是是DB的子类，这个类在后面介绍；

调用impl->Recover()来进行恢复数据库

回复成功后且内存表为空，则创建一个新的日志文件和内存表。

保存Manifest文件

移除过时文件并安排压缩任务

```
  if (s.ok() && save_manifest) {
    edit.SetPrevLogNumber(0);  // No older logs needed after recovery.
    edit.SetLogNumber(impl->logfile_number_);
    s = impl->versions_->LogAndApply(&edit, &impl->mutex_);
  }
  if (s.ok()) {
    impl->RemoveObsoleteFiles();
    impl->MaybeScheduleCompaction();
  }
  impl->mutex_.Unlock();        解锁
  if (s.ok()) {
    assert(impl->mem_ != nullptr);
    *dbptr = impl;
  } else {
    delete impl;
  }
  return s;
}
```

移除过时文件并安排压缩任务

DB接口的实现在DBImpl类中，该类继承于父类DB：其定义在db_impl.h中（很长在这里没有再去进一步写出来）。

我们直接来看Put（）函数即可：
这是他的声明：

```
  Status Put(const WriteOptions&, const Slice& key,
             const Slice& value) override;
```

具体的实现过程如下：

```
Status DBImpl::Put(const WriteOptions& o, const Slice& key, const Slice& val) {
  return DB::Put(o, key, val);
}
```

put（）实现如下：

```
// Default implementations of convenience methods that subclasses of DB
// can call if they wish
Status DB::Put(const WriteOptions& opt, const Slice& key, const Slice& value) {
  WriteBatch batch;
  batch.Put(key, value);
  return Write(opt, &batch);
}
```

在这个函数内部调用了batch.Put（key，value）函数，返回了Write(opt, &batch)
那么首先的第一个问题就是：WriteBatch batch这句，WriteBatch是什么？？
参见LevelDB笔记7：WriteBatch

问题：为什么在子类DBImpl类中的成员方法Put（）中反而调用了父类DB::Put()，而且DB类中的Put方法不是虚方法吗？为什么直接使用：：这两个冒号标志直接就可以调用该方法？为什么要这样一环套一环的实现Put？

然后调用了方法：Write执行写入

```
Status DBImpl::Write(const WriteOptions& options, WriteBatch* updates) {
  Writer w(&mutex_);
  w.batch = updates;
  w.sync = options.sync;
  w.done = false;
  MutexLock l(&mutex_);
  writers_.push_back(&w);
  while (!w.done && &w != writers_.front()) {
    w.cv.Wait();
  }
  if (w.done) {
    return w.status;
  }
  // May temporarily unlock and wait.
  Status status = MakeRoomForWrite(updates == nullptr);
  uint64_t last_sequence = versions_->LastSequence();
  Writer* last_writer = &w;
  if (status.ok() && updates != nullptr) {  // nullptr batch is for compactions
    WriteBatch* write_batch = BuildBatchGroup(&last_writer);
    WriteBatchInternal::SetSequence(write_batch, last_sequence + 1);
    last_sequence += WriteBatchInternal::Count(write_batch);
    // Add to log and apply to memtable.  We can release the lock
    // during this phase since &w is currently responsible for logging
    // and protects against concurrent loggers and concurrent writes
    // into mem_.
    {
      mutex_.Unlock();
      status = log_->AddRecord(WriteBatchInternal::Contents(write_batch));
      bool sync_error = false;
      if (status.ok() && options.sync) {
        status = logfile_->Sync();
        if (!status.ok()) {
          sync_error = true;
        }
      }
      if (status.ok()) {
        status = WriteBatchInternal::InsertInto(write_batch, mem_);
      }
      mutex_.Lock();
```

```
      if (sync_error) {
        // The state of the log file is indeterminate: the log record we
        // just added may or may not show up when the DB is re-opened.
        // So we force the DB into a mode where all future writes fail.
        RecordBackgroundError(status);
      }
    }
    if (write_batch == tmp_batch_) tmp_batch_->Clear();
    versions_->SetLastSequence(last_sequence);
  }
  while (true) {
    Writer* ready = writers_.front();
    writers_.pop_front();
    if (ready != &w) {
      ready->status = status;
      ready->done = true;
      ready->cv.Signal();
    }
    if (ready == last_writer) break;
  }
  // Notify new head of write queue
  if (!writers_.empty()) {
    writers_.front()->cv.Signal();
  }
  return status;
}
```

Write过程主要调用上面黄色标注的两个过程。
一是追加写到log中，二是写入到内存中。

其实AddRecord（Slice）过程的参数是
```
  static Slice Contents(const WriteBatch* batch) { return Slice(batch->rep_); }
```
写到log中的详细过程见LevelDB笔记5

下面是DBImpl的构造函数：
```
DBImpl::DBImpl(const Options& raw_options, const std::string& dbname)
    : env_(raw_options.env),
      internal_comparator_(raw_options.comparator),
      internal_filter_policy_(raw_options.filter_policy),
      options_(SanitizeOptions(dbname, &internal_comparator_,
                               &internal_filter_policy_, raw_options)),
      owns_info_log_(options_.info_log != raw_options.info_log),
      owns_cache_(options_.block_cache != raw_options.block_cache),
      dbname_(dbname),
      table_cache_(new TableCache(dbname_, options_, TableCacheSize(options_))),
      db_lock_(nullptr),
      shutting_down_(false),
      background_work_finished_signal_(&mutex_),
      mem_(nullptr),
      imm_(nullptr),
      has_imm_(false),
      logfile_(nullptr),
      logfile_number_(0),
      log_(nullptr),
      seed_(0),
      tmp_batch_(new WriteBatch),
      background_compaction_scheduled_(false),
      manual_compaction_(nullptr),
      versions_(new VersionSet(dbname_, &options_, table_cache_,
                               &internal_comparator_)) {}
```

LevelDB笔记2：SkipList详解

2024年7月29日　　10:17

这是leveldb中定义的SkipList的类：

```
class SkipList {
 private:
  struct Node;
 public:
  // Create a new SkipList object that will use "cmp" for comparing keys,
  // and will allocate memory using "*arena".  Objects allocated in the arena
  // must remain allocated for the lifetime of the skiplist object.
  explicit SkipList(Comparator cmp, Arena* arena);
  SkipList(const SkipList&) = delete;
  SkipList& operator=(const SkipList&) = delete;
  // Insert key into the list.
  // REQUIRES: nothing that compares equal to key is currently in the list.
  void Insert(const Key& key);
  // Returns true iff an entry that compares equal to key is in the list.
  bool Contains(const Key& key) const;
  // Iteration over the contents of a skip list
  class Iterator {
   public:
    // Initialize an iterator over the specified list.
    // The returned iterator is not valid.
    explicit Iterator(const SkipList* list);
    // Returns true iff the iterator is positioned at a valid node.
    bool Valid() const;
    // Returns the key at the current position.
    // REQUIRES: Valid()
    const Key& key() const;
    // Advances to the next position.
    // REQUIRES: Valid()
    void Next();
    // Advances to the previous position.
    // REQUIRES: Valid()
    void Prev();
    // Advance to the first entry with a key >= target
    void Seek(const Key& target);
    // Position at the first entry in list.
    // Final state of iterator is Valid() iff list is not empty.
    void SeekToFirst();
    // Position at the last entry in list.
    // Final state of iterator is Valid() iff list is not empty.
    void SeekToLast();
   private:
    const SkipList* list_;
    Node* node_;
    // Intentionally copyable
  };
 private:
  enum { kMaxHeight = 12 };
  inline int GetMaxHeight() const {
    return max_height_.load(std::memory_order_relaxed);
  }
  Node* NewNode(const Key& key, int height);
  int RandomHeight();
  bool Equal(const Key& a, const Key& b) const { return (compare_(a, b) == 0); }
```

```
  // Return true if key is greater than the data stored in "n"
  bool KeyIsAfterNode(const Key& key, Node* n) const;
  // Return the earliest node that comes at or after key.
  // Return nullptr if there is no such node.
  //
  // If prev is non-null, fills prev[level] with pointer to previous
  // node at "level" for every level in [0..max_height_-1].
  Node* FindGreaterOrEqual(const Key& key, Node** prev) const;
  // Return the latest node with a key < key.
  // Return head_ if there is no such node.
  Node* FindLessThan(const Key& key) const;
  // Return the last node in the list.
  // Return head_ if list is empty.
  Node* FindLast() const;
  // Immutable after construction
  Comparator const compare_;
  Arena* const arena_;  // Arena used for allocations of nodes
  Node* const head_;
  // Modified only by Insert().  Read racily by readers, but stale
  // values are ok.
  std::atomic<int> max_height_;  // Height of the entire list
  // Read/written only by Insert().
  Random rnd_;
};
```

先看看SkipList的构造函数：

```
template<typename Key, class Comparator>
SkipList<Key, Comparator>::SkipList(Comparator cmp, Arena* arena)
    : compare_(cmp),
      arena_(arena),
      head_(NewNode(0 /* any key will do */, kMaxHeight)),
      max_height_(1),
      rnd_(0xdeadbeef) {
  for (int i = 0; i < kMaxHeight; i++) {
    head_->SetNext(i, nullptr);
  }
}
```

Head_为跳表的头节点，构造函数中用NewNode（）函数的返回值进行初始化Head_，

NewNode（）是SkipList这个类的private类型的成员函数，用来创建并初始化跳表中的新节点：

```
  char* const node_memory = arena_->AllocateAligned(
      sizeof(Node) + sizeof(std::atomic<Node*>) * (height - 1));
  return new (node_memory) Node(key);
```

主要调用 arena_->AllocateAligned () 这个方法

从内存池arena中分配对齐的内存块。arena这个内存池的详解见LevelDB笔记3中内存池详解。

分配的内存空间主要是Node的大小，和next_指针数组的大小。

所有Node对象都通过NewNode()构造出来：先通过arena_分配内存（给这个新的Node对象），然后通过**placement new**的方式调用Node的构造函数（Placement new的详解参见C++中的new的笔记）

因此，可以看到在上面的SkipList构造函数里面初始化了head_，高度为kMaxHeight，并且设置每一层的后继节点为nullptr

来看levelDB中跳表的插入过程是如何实现的：

```
template <typename Key, class Comparator>
void SkipList<Key, Comparator>::Insert(const Key& key) {

  Node* prev[kMaxHeight];       //指针数组，跳表的每一层如第2层，prev[2]就指向一个Node结点
```

```
  Node* x = FindGreaterOrEqual(key, prev);  //在跳表的每层中查找第一个≥key值的结点，然后将其前驱记录在prev的指针数组中。

  assert(x == nullptr || !Equal(key, x->key));

  int height = RandomHeight();      //确定要插入结点的高度，它利用随机函数生成的
  // 如果比当前的跳表最大高度都要大，那么更新最大高度，并将新增高度层的前驱节点指向跳表的头节点（head_）。
  if (height > GetMaxHeight()) {
    for (int i = GetMaxHeight(); i < height; i++) {
      prev[i] = head_;
    }
    max_height_.store(height, std::memory_order_relaxed);
  }

  //创建新节点，指定其高度为Height
  x = NewNode(key, height);

  //依次将这个新节点插入到height以下的索引层中（包括最底层数据层）
  //插入结点x到prev及prev->next中间
  for (int i = 0; i < height; i++) {
    // NoBarrier_SetNext() suffices since we will add a barrier when
    // we publish a pointer to "x" in prev[i].
    x->NoBarrier_SetNext(i, prev[i]->NoBarrier_Next(i));
    //先修改x结点，再修改prev结点
    prev[i]->SetNext(i, x);
  }
}
```

我们已经知道在跳表插入过程中会以某一概率插入到相应的索引层中，而在levelDB中跳表插入结点高度确定取决于下面这个函数：

```
template <typename Key, class Comparator>
int SkipList<Key, Comparator>::RandomHeight() {
  // Increase height with probability 1 in kBranching
  static const unsigned int kBranching = 4;
  int height = 1;
  while (height < kMaxHeight && rnd_.OneIn(kBranching)) {
    height++;
  }
  assert(height > 0);
  assert(height <= kMaxHeight);
  return height;
}
```

```
  static const unsigned int kBranching = 4;这个其实就是我们可以自定义的概率值，当插入一个新结点的时候，以１／４的概率构建一级索引（执行一次while循环），以１／１６的概率建立二级索引（执行两次while循环），依次类推。

 bool OneIn(int n) { return (Next() % n) == 0; }
```

Next（）用来高效生成一个伪随机数（LCG线性同余生成器），然后当返回值能被ｎ整除时，ＯｎｅＩＮ（）函数返回True，而传递进来的ｎ即为上面的ｋＢｒａｎｇｃｈｉｎｇ＝４，这说明Ｎｅｘｔ（）％４，有着１／４的概率返回True，也就意味着ｗｈｉｌｅ循环内，每一次判断都有１／４的几率对Ｈｅｉｇｈｔ＋＋

```
bool SkipList<Key, Comparator>::Contains(const Key& key) const {
  Node* x = FindGreaterOrEqual(key, nullptr);
  if (x != nullptr && Equal(key, x->key)) {
    return true;
  } else {
    return false;
  }
}
```

```
typename SkipList<Key, Comparator>::Node*
SkipList<Key, Comparator>::FindGreaterOrEqual(const Key& key,
                                              Node** prev) const {
  Node* x = head_;          //指向跳表的头节点
  int level = GetMaxHeight() - 1;   //
  while (true) {
    Node* next = x->Next(level);    //返回在第level层的x结点的下一个结点的指针
    if (KeyIsAfterNode(key, next)) {  //判断key是否大于当前层next结点的键，如果是，则
      // Keep searching in this list
      x = next;
    } else {
```

//Note:如果单纯为了判断是否相等，这里可以加一个判断直接返回了，没必要再level--到0再返回，不过复杂度没有变化

//第一个if语句用来对prev[level]进行复制，迭代到最后它指向当前level层Node结点的key刚好大于等于要find的key的前一个结点，即它用来记录查找key值过程中的刚好要执行down的那一个结点指

针，当然最底层0层，也就是prev【0】这个指针指向底层链表中要查找结点的前驱结点（虽然有可能查找不到，但总之他会指向 ≥ 找到的Node结点前驱 （找到的这个Node结点的key刚好 要满足 ≥ 要查找的key

```
      if (prev != nullptr) prev[level] = x;
      if (level == 0) {
        return next;
      } else {
        // Switch to next list
        level--;
      }
    }
  }
}
```

```
template <typename Key, class Comparator>
bool SkipList<Key, Comparator>::KeyIsAfterNode(const Key& key, Node* n) const {
  // null n is considered infinite
  return (n != nullptr) &&                          ;
}
```

Node

```
// Implementation details follow
template <typename Key, class Comparator>
struct SkipList<Key, Comparator>::Node {
  explicit Node(const Key& k) : key(k) {}
  Key const key;

  //获取或者设置该节点在第n层的后继节点
  // Accessors/mutators for links.  Wrapped in methods so we can
  // add the appropriate barriers as necessary.
  Node* Next(int n) {
    assert(n >= 0);
    // Use an 'acquire load' so that we observe a fully initialized
    // version of the returned Node.
    return next_[n].load(std::memory_order_acquire);
  }
  void SetNext(int n, Node* x) {
    assert(n >= 0);
    // Use a 'release store' so that anybody who reads through this
    // pointer observes a fully initialized version of the inserted node.
    next_[n].store(x, std::memory_order_release);
  }
  // No-barrier variants that can be safely used in a few locations.
  Node* NoBarrier_Next(int n) {
    assert(n >= 0);
    return next_[n].load(std::memory_order_relaxed);
  }
  void NoBarrier_SetNext(int n, Node* x) {
    assert(n >= 0);
    next_[n].store(x, std::memory_order_relaxed);
  }
 private:
  // Array of length equal to the node height.  next_[0] is lowest level link.
  std::atomic<Node*> next_[1];
};
```

```
  Node* FindLessThan(const Key& key) const;
  Node* FindLast() const;

typename SkipList<Key, Comparator>::Node*
SkipList<Key, Comparator>::FindLessThan(const Key& key) const {
  Node* x = head_;
  int level = GetMaxHeight() - 1;
  while (true) {
    assert(x == head_ || compare_(x->key, key) < 0);
    Node* next = x->Next(level);
    if (next == nullptr || compare_(next->key, key) >= 0) {
      if (level == 0) {
        return x;
      } else {
        // Switch to next list
        level--;
      }
    } else {
      x = next;
    }
  }
}

typename SkipList<Key, Comparator>::Node* SkipList<Key, Comparator>::FindLast()
    const {
  Node* x = head_;
  int level = GetMaxHeight() - 1;
  while (true) {
    Node* next = x->Next(level);
    if (next == nullptr) {
      if (level == 0) {
        return x;
      } else {
        // Switch to next list
        level--;
      }
    } else {
      x = next;
    }
  }
}



class Iterator {
   public:
    // Initialize an iterator over the specified list.
    // The returned iterator is not valid.
    explicit Iterator(const SkipList* list);
    // Returns true iff the iterator is positioned at a valid node.
    bool Valid() const;
    // Returns the key at the current position.
```

```
    // REQUIRES: Valid()
    const Key& key() const;
    // Advances to the next position.
    // REQUIRES: Valid()
    void Next();
    // Advances to the previous position.
    // REQUIRES: Valid()
    void Prev();
    // Advance to the first entry with a key >= target
    void Seek(const Key& target);
    // Position at the first entry in list.
    // Final state of iterator is Valid() iff list is not empty.
    void SeekToFirst();
    // Position at the last entry in list.
    // Final state of iterator is Valid() iff list is not empty.
    void SeekToLast();
  private:
    const SkipList* list_;
    Node* node_;
    // Intentionally copyable
  };
```

```
inline SkipList<Key, Comparator>::Iterator::Iterator(const SkipList* list) {
  list_ = list;        //将迭代器与特定的跳表实例关联起来，使得后续的迭代操作都基于这个跳表进行。
  node_ = nullptr;    //刚创建的迭代器不会指向跳表中的任何结点。
}
```

2024年8月13日 18:50

```
class Arena {
 public:
  Arena();
  Arena(const Arena&) = delete;
  Arena& operator=(const Arena&) = delete;
  ~Arena();
  // Return a pointer to a newly allocated memory block of "bytes" bytes.
  char* Allocate(size_t bytes);
  // Allocate memory with the normal alignment guarantees provided by malloc.
  char* AllocateAligned(size_t bytes);
  // Returns an estimate of the total memory usage of data allocated
  // by the arena.
  size_t MemoryUsage() const {
    return memory_usage_.load(std::memory_order_relaxed);
  }
 private:
  char* AllocateFallback(size_t bytes);
  char* AllocateNewBlock(size_t block_bytes);
  // Allocation state
  char* alloc_ptr_;
  size_t alloc_bytes_remaining_;
  // Array of new[] allocated memory blocks
  std::vector<char*> blocks_;
  // Total memory usage of the arena.
  //
  // TODO(costan): This member is accessed via atomics, but the others are
  //               accessed without any locking. Is this OK?
  std::atomic<size_t> memory_usage_;
};
```

```
Arena::Arena()
    : alloc_ptr_(nullptr), alloc_bytes_remaining_(0), memory_usage_(0) {}
```

```
// Allocation state
  char* alloc_ptr_;       //指向当前可用内存块的起始位置。
  size_t alloc_bytes_remaining_;            //表示当前内存块中剩余的可用字节数。
//上面这俩变量一起用于跟踪当前内存块的使用情况。

  // Array of new[] allocated memory blocks
  std::vector<char*> blocks_;
  // Total memory usage of the arena.
  //
  // TODO(costan): This member is accessed via atomics, but the others are
  //               accessed without any locking. Is this OK?
  std::atomic<size_t> memory_usage_;
```

```
  std::vector<char*> blocks_;
```

```
Arena::~Arena() {
  for (size_t i = 0; i < blocks_.size(); i++) {
    delete[] blocks_[i];
  }
}
```

```
inline char* Arena::Allocate(size_t bytes) {
  // The semantics of what to return are a bit messy if we allow
  // 0-byte allocations, so we disallow them here (we don't need
  // them for our internal use).
  assert(bytes > 0);
  if (bytes <= alloc_bytes_remaining_) {
    char* result = alloc_ptr_;
    alloc_ptr_ += bytes;
    alloc_bytes_remaining_ -= bytes;
    return result;
  }
  return AllocateFallback(bytes);
}
```

这个函数用来分配内存空间，如果剩余空间足够分配，则返回分配空间那块的首地址，如果不足够分配，那么调用Arena类内部私有的成员方法AllocateFallback()来处理

```
char* Arena::AllocateAligned(size_t bytes) {
  const int align = (sizeof(void*) > 8) ? sizeof(void*) : 8;
  static_assert((align & (align - 1)) == 0,
                "Pointer size should be a power of 2");
  size_t current_mod = reinterpret_cast<uintptr_t>(alloc_ptr_) & (align - 1);
  size_t slop = (current_mod == 0 ? 0 : align - current_mod);
  size_t needed = bytes + slop;
  char* result;
  if (needed <= alloc_bytes_remaining_) {
    result = alloc_ptr_ + slop;
    alloc_ptr_ += needed;
    alloc_bytes_remaining_ -= needed;
  } else {
    // AllocateFallback always returned aligned memory
    result = AllocateFallback(bytes);
  }
  assert((reinterpret_cast<uintptr_t>(result) & (align - 1)) == 0);
  return result;
}
```

```
char* Arena::AllocateFallback(size_t bytes) {
  if (bytes > kBlockSize / 4) {
    // Object is more than a quarter of our block size.  Allocate it
separately
    // to avoid wasting too much space in leftover bytes.
    char* result = AllocateNewBlock(bytes);
    return result;
```

```
  }
  // We waste the remaining space in the current block.
  alloc_ptr_ = AllocateNewBlock(kBlockSize);
  alloc_bytes_remaining_ = kBlockSize;
  char* result = alloc_ptr_;
  alloc_ptr_ += bytes;
  alloc_bytes_remaining_ -= bytes;
  return result;
}
```

```
char* Arena::AllocateNewBlock(size_t block_bytes) {
  char* result = new char[block_bytes];
  blocks_.push_back(result);
  memory_usage_.fetch_add(block_bytes + sizeof(char*),
                          std::memory_order_relaxed);
  return result;
}
```

```
char* result = new char[block_bytes];
```

```
blocks_.push_back(result);
```

```
  memory_usage_.fetch_add(block_bytes + sizeof(char*),
                          std::memory_order_relaxed);
```

```
class MemTable {
 public:
  // MemTables are reference counted.  The initial reference count
  // is zero and the caller must call Ref() at least once.
  explicit MemTable(const InternalKeyComparator& comparator);
  MemTable(const MemTable&) = delete;
  MemTable& operator=(const MemTable&) = delete;
  // Increase reference count.
  void Ref() { ++refs_; }
  // Drop reference count.  Delete if no more references exist.
  void Unref() {
    --refs_;
    assert(refs_ >= 0);
    if (refs_ <= 0) {
      delete this;
    }
  }
  // Returns an estimate of the number of bytes of data in use by this
  // data structure. It is safe to call when MemTable is being modified.
  size_t ApproximateMemoryUsage();
  // Return an iterator that yields the contents of the memtable.
  //
  // The caller must ensure that the underlying MemTable remains live
  // while the returned iterator is live.  The keys returned by this
  // iterator are internal keys encoded by AppendInternalKey in the
  // db/format.{h,cc} module.
  Iterator* NewIterator();
  // Add an entry into memtable that maps key to value at the
  // specified sequence number and with the specified type.
  // Typically value will be empty if type==kTypeDeletion.
  void Add(SequenceNumber seq, ValueType type, const Slice& key,
           const Slice& value);
  // If memtable contains a value for key, store it in *value and return true.
  // If memtable contains a deletion for key, store a NotFound() error
  // in *status and return true.
  // Else, return false.
  bool Get(const LookupKey& key, std::string* value, Status* s);
 private:
  friend class MemTableIterator;
  friend class MemTableBackwardIterator;
  struct KeyComparator {
    const InternalKeyComparator comparator;
    explicit KeyComparator(const InternalKeyComparator& c) : comparator(c) {}
    int operator()(const char* a, const char* b) const;
  };
  typedef SkipList<const char*, KeyComparator> Table;
  ~MemTable();  // Private since only Unref() should be used to delete it
  KeyComparator comparator_;
  int refs_;
  Arena arena_;
  Table table_;
};
```

```
    void Ref() { ++refs_; }
    // Drop reference count.  Delete if no more references exist.
    void Unref() {
      --refs_;
      assert(refs_ >= 0);
      if (refs_ <= 0) {
        delete this;
      }
    }
```

2.主要接口:

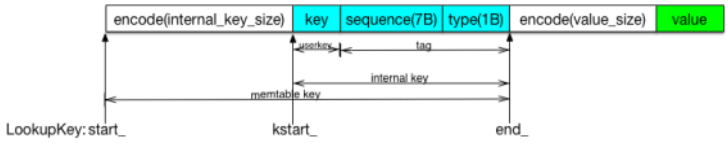
```
    size_t ApproximateMemoryUsage();
    Iterator* NewIterator();


    void Add(SequenceNumber seq, ValueType type, const Slice& key,
             const Slice& value);
    bool Get(const LookupKey& key, std::string* value, Status* s);
```

```
MemTable::MemTable(const InternalKeyComparator& comparator)
    : comparator_(comparator), refs_(0), table_(comparator_, &arena_) {}

MemTable::~MemTable() { assert(refs_ == 0); }
```

```
assert(refs_ == 0);
```

```
  KeyComparator comparator_;
  int refs_;
  Arena arena_;
  Table table_;
```

```
void MemTable::Add(SequenceNumber s, ValueType type, const Slice& key,
                   const Slice& value) {
  // Format of an entry is concatenation of:
  //  key_size     : varint32 of internal_key.size()
  //  key bytes    : char[internal_key.size()]
  //  tag          : uint64((sequence << 8) | type)
  //  value_size   : varint32 of value.size()
  //  value bytes  : char[value.size()]
  size_t key_size = key.size();        //键的字节长度
  size_t val_size = value.size();     //值得字节长度
  size_t internal_key_size = key_size + 8;    //键的长度加上8字节得tag，作为内部键的大
小，tag在前面的注释中已经说明是序列号左移8位后与type相或（即它两拼接起来即可）。
  const size_t encoded_len = VarintLength(internal_key_size) +
                             internal_key_size + VarintLength(val_size) +
                             val_size;
//encoded_len计算了编码后的总长度
//用encoded_len的大小在Arena中分配内存，用于存储编码后的数据。
  char* buf = arena_.Allocate(encoded_len);

  char* p = EncodeVarint32(buf, internal_key_size);
  std::memcpy(p, key.data(), key_size);
  p += key_size;
  EncodeFixed64(p, (s << 8) | type);
  p += 8;
  p = EncodeVarint32(p, val_size);
  std::memcpy(p, value.data(), val_size);
  assert(p + val_size == buf + encoded_len);
```

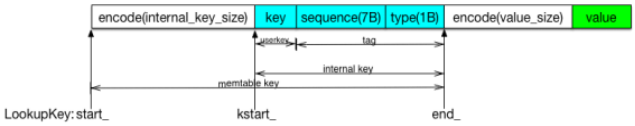



```
void SkipList<Key, Comparator>::Insert(const Key& key)
```

```
template <typename Key, class Comparator>
```

```
bool MemTable::Get(const LookupKey& key, std::string* value, Status* s) {

  Slice memkey = key.memtable_key();
  Table::Iterator iter(&table_);
  iter.Seek(memkey.data());
```

//如果seek之后的这个迭代器有效




```
  if (iter.Valid()) {
    // entry format is:
    //    klength  varint32
    //    userkey  char[klength]
    //    tag      uint64
    //    vlength  varint32
    //    value    char[vlength]
    // Check that it belongs to same user key.  We do not check the
    // sequence number since the Seek() call above should have skipped
    // all entries with overly large sequence numbers.
    const char* entry = iter.key();
    uint32_t key_length;
    const char* key_ptr = GetVarint32Ptr(entry, entry + 5, &key_length);

    if (comparator_.comparator.user_comparator()->Compare(
            Slice(key_ptr, key_length - 8), key.user_key()) == 0) {
      // Correct user key
      const uint64_t tag = DecodeFixed64(key_ptr + key_length - 8);
      switch (static_cast<ValueType>(tag & 0xff)) {
        case kTypeValue: {
          Slice v = GetLengthPrefixedSlice(key_ptr + key_length);
          value->assign(v.data(), v.size());
          return true;
        }
        case kTypeDeletion:
          *s = Status::NotFound(Slice());
          return true;
      }
    }
  }
  return false;
}
```

```
inline void SkipList<Key, Comparator>::Iterator::Seek(const Key& target) {
  node_ = list_->FindGreaterOrEqual(target, nullptr);
}
```

```
    const char* entry = iter.key();

inline const Key& SkipList<Key, Comparator>::Iterator::key() const {
  assert(Valid());
  return node_->key;
}
```

```
                                    const LookupKey& key
```

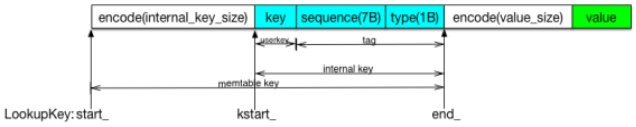
```
// A helper class useful for DBImpl::Get()
class LookupKey {
 public:
  // Initialize *this for looking up user_key at a snapshot with
  // the specified sequence number.
  LookupKey(const Slice& user_key, SequenceNumber sequence);
  LookupKey(const LookupKey&) = delete;
  LookupKey& operator=(const LookupKey&) = delete;
  ~LookupKey();

  // Return a key suitable for lookup in a MemTable.
  Slice memtable_key() const { return Slice(start_, end_ - start_); }
  // Return an internal key (suitable for passing to an internal iterator)
  Slice internal_key() const { return Slice(kstart_, end_ - kstart_); }
  // Return the user key
  Slice user_key() const { return Slice(kstart_, end_ - kstart_ - 8); }

private:
  // We construct a char array of the form:
  //    klength  varint32               <-- start_
  //    userkey  char[klength]          <-- kstart_
  //    tag      uint64
  //                                    <-- end_
  // The array is a suitable MemTable key.
  // The suffix starting with "userkey" can be used as an InternalKey.
```

```
  const char* start_;
  const char* kstart_;
  const char* end_;
  char space_[200];  // Avoid allocation for short keys
};
```

```
LookupKey::LookupKey(const Slice& user_key, SequenceNumber s) {
  size_t usize = user_key.size();
  size_t needed = usize + 13;  // A conservative estimate
```

//sequence Number+ValueType占8个字节，encode（internal key size）至多占5个字节，预估最多占13个字节

```
  char* dst;
  if (needed <= sizeof(space_)) {
    dst = space_;
  } else {
    dst = new char[needed];
  }
  start_ = dst;
  dst = EncodeVarint32(dst, usize + 8);
  kstart_ = dst;
  std::memcpy(dst, user_key.data(), usize);
  dst += usize;
  EncodeFixed64(dst, PackSequenceAndType(s, kValueTypeForSeek));
  dst += 8;
  end_ = dst;
}

inline LookupKey::~LookupKey() {
  if (start_ != space_) delete[] start_;
}
```




```
 const char* entry = iter.key();
    uint32_t key_length;
    const char* key_ptr = GetVarint32Ptr(entry, entry + 5, &key_length);
```

```
inline const char* GetVarint32Ptr(const char* p, const char* limit,
                                  uint32_t* value) {
  if (p < limit) {
```

//首先将p指向的那个字节的内存解析出来赋给result

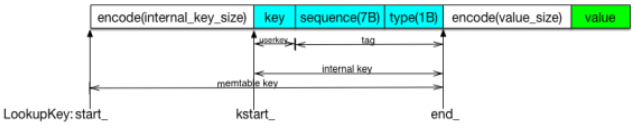
```
    uint32_t result = *(reinterpret_cast<const uint8_t*>(p));
```

//如果解析出来的result值最高位不是1，则将该值保存在*value中用以返回，然后返回p+1的地址

```
    if ((result & 128) == 0) {
      *value = result;
      return p + 1;
    }
  }
```

//执行下面这个函数的时候，有两种情况：一是p>limit了，二是p指向的数据，即解析出来的result的值的最高位是1（说明key不完整）。0

```
  return GetVarint32PtrFallback(p, limit, value);
}
```



注意：key_ptr指向的是kstart_的位置

```
if (comparator_.comparator.user_comparator()->Compare(
            Slice(key_ptr, key_length - 8), key.user_key()) == 0) {
      // Correct user key
```

//解析出internalKey的尾部标记（tag），其中包含了值的类型和序列号。

//tag是64位的解析出的sequence+type的那8个字节内容（64位）

```
      const uint64_t tag = DecodeFixed64(key_ptr + key_length - 8);
```

```
      //将tag&0xff的结构静态转化为ValueType类型，tag是8B是64位的数据
      //tag & 0xff 这个结果自然就是保留tag最低为的那8位，即最后一个字节，即type值

       switch (static_cast<ValueType>(tag & 0xff)) {
         case kTypeValue: {
            //解析出存储的值，并将其赋给value
           Slice v = GetLengthPrefixedSlice(key_ptr + key_length);
           value->assign(v.data(), v.size());
           return true;
         }
         case kTypeDeletion:
            //类型是kTypeDeletion，则表示改键已经被删除，函数返回true，并设置状态s为
NotFound
           *s = Status::NotFound(Slice());
           return true;
       }
     }
```

```
KeyComparator comparator_;

struct KeyComparator {
    const InternalKeyComparator comparator;
    explicit KeyComparator(const InternalKeyComparator& c) : comparator(c) {}
    int operator()(const char* a, const char* b) const;
  };


enum ValueType { kTypeDeletion = 0x0, kTypeValue = 0x1 };


int MemTable::KeyComparator::operator()(const char* aptr,
                                        const char* bptr) const {
  // Internal keys are encoded as length-prefixed strings.
  Slice a = GetLengthPrefixedSlice(aptr);
  Slice b = GetLengthPrefixedSlice(bptr);
  return comparator.Compare(a, b);
}



int InternalKeyComparator::Compare(const Slice& akey, const Slice& bkey) const {
  // Order by:
  //    increasing user key (according to user-supplied comparator)
  //    decreasing sequence number
  //    decreasing type (though sequence# should be enough to disambiguate)

  // 先比较user_key
  int r = user_comparator_->Compare(ExtractUserKey(akey), ExtractUserKey(bkey));
  //如果是同一个user_key，那么再比较tag，解析出来sequence number进行比较
  //sequence number越大，排序结果越小
  if (r == 0) {
    const uint64_t anum = DecodeFixed64(akey.data() + akey.size() - 8);
    const uint64_t bnum = DecodeFixed64(bkey.data() + bkey.size() - 8);
    if (anum > bnum) {
      r = -1;
    } else if (anum < bnum) {
      r = +1;
    }
  }
  return r;
}



// A comparator for internal keys that uses a specified comparator for
// the user key portion and breaks ties by decreasing sequence number.
class InternalKeyComparator : public Comparator {
 private:
  const Comparator* user_comparator_;
 public:
  explicit InternalKeyComparator(const Comparator* c) : user_comparator_(c) {}
  const char* Name() const override;
  int Compare(const Slice& a, const Slice& b) const override;
  void FindShortestSeparator(std::string* start,
                             const Slice& limit) const override;
  void FindShortSuccessor(std::string* key) const override;
  const Comparator* user_comparator() const { return user_comparator_; }
  int Compare(const InternalKey& a, const InternalKey& b) const;
};


class LEVELDB_EXPORT Comparator {
 public:
  virtual ~Comparator();
  // Three-way comparison.  Returns value:
  //   < 0 iff "a" < "b",
```

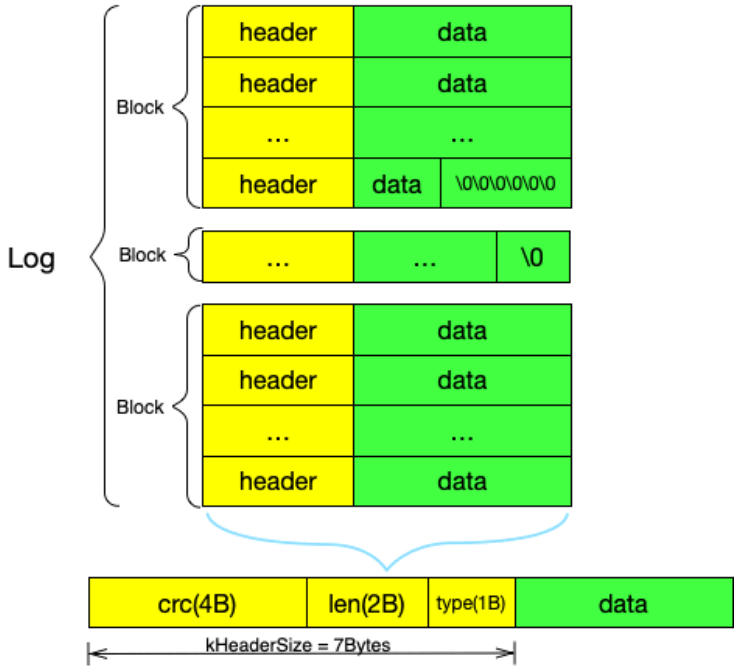
```
        //   == 0 iff "a" == "b",
        //   > 0 iff "a" > "b"
        virtual int Compare(const Slice& a, const Slice& b) const = 0;
        // The name of the comparator.  Used to check for comparator
        // mismatches (i.e., a DB created with one comparator is
        // accessed using a different comparator.
        //
        // The client of this package should switch to a new name whenever
        // the comparator implementation changes in a way that will cause
        // the relative ordering of any two keys to change.
        //
        // Names starting with "leveldb." are reserved and should not be used
        // by any clients of this package.
        virtual const char* Name() const = 0;
        // Advanced functions: these are used to reduce the space requirements
        // for internal data structures like index blocks.
        // If *start < limit, changes *start to a short string in [start,limit).
        // Simple comparator implementations may return with *start unchanged,
        // i.e., an implementation of this method that does nothing is correct.
        virtual void FindShortestSeparator(std::string* start,
                                           const Slice& limit) const = 0;
        // Changes *key to a short string >= *key.
        // Simple comparator implementations may return with *key unchanged,
        // i.e., an implementation of this method that does nothing is correct.
        virtual void FindShortSuccessor(std::string* key) const = 0;
      };

  Iterator* NewIterator();

Iterator* MemTable::NewIterator() { return new MemTableIterator(&table_); }


class MemTableIterator : public Iterator {
 public:
  explicit MemTableIterator(MemTable::Table* table) : iter_(table) {}
  MemTableIterator(const MemTableIterator&) = delete;
  MemTableIterator& operator=(const MemTableIterator&) = delete;
  ~MemTableIterator() override = default;
  bool Valid() const override { return iter_.Valid(); }
  void Seek(const Slice& k) override { iter_.Seek(EncodeKey(&tmp_, k)); }
  void SeekToFirst() override { iter_.SeekToFirst(); }
  void SeekToLast() override { iter_.SeekToLast(); }
  void Next() override { iter_.Next(); }
  void Prev() override { iter_.Prev(); }
  Slice key() const override { return GetLengthPrefixedSlice(iter_.key()); }
  Slice value() const override {
    Slice key_slice = GetLengthPrefixedSlice(iter_.key());
    return GetLengthPrefixedSlice(key_slice.data() + key_slice.size());
  }
  Status status() const override { return Status::OK(); }
 private:
  MemTable::Table::Iterator iter_;
  std::string tmp_;  // For passing to EncodeKey
};
```

2024年8月17日 21:01

```
namespace log {
class Writer {
 public:
  // Create a writer that will append data to "*dest".
  // "*dest" must be initially empty.
  // "*dest" must remain live while this Writer is in use.
  explicit Writer(WritableFile* dest);
  // Create a writer that will append data to "*dest".
  // "*dest" must have initial length "dest_length".
  // "*dest" must remain live while this Writer is in use.
  Writer(WritableFile* dest, uint64_t dest_length);

  Writer(const Writer&) = delete;
  Writer& operator=(const Writer&) = delete;
  ~Writer();
  Status AddRecord(const Slice& slice);
 private:
  Status EmitPhysicalRecord(RecordType type, const char* ptr, size_t length);
  WritableFile* dest_;
  int block_offset_;  // Current offset in block
  // crc32c values for all supported record types.  These are
  // pre-computed to reduce the overhead of computing the crc of the
  // record type stored in the header.
  uint32_t type_crc_[kMaxRecordType + 1];
};
}
```

```
Writer::Writer(WritableFile* dest) : dest_(dest), block_offset_(0) {
  InitTypeCrc(type_crc_);
```

```
}

Writer::Writer(WritableFile* dest, uint64_t dest_length)
    : dest_(dest), block_offset_(dest_length % kBlockSize) {
  InitTypeCrc(type_crc_);
}

Writer::~Writer() = default;
```




```
Status Writer::AddRecord(const Slice& slice) {
  const char* ptr = slice.data();
  size_t left = slice.size();
  // Fragment the record if necessary and emit it.  Note that if slice
  // is empty, we still want to iterate once to emit a single
  // zero-length record
  Status s;
  bool begin = true;

 //下面是一个do-while循环，
  do {
    const int leftover = kBlockSize - block_offset_; //当前块剩余的可用容量
    assert(leftover >= 0);
    //如果该块中剩余的空间连header都放不下，那么将这些空间全部置为0x00，调用的是dest_->
Append()函数
    if (leftover < kHeaderSize) {
      // Switch to a new block
      if (leftover > 0) {
        // Fill the trailer (literal below relies on kHeaderSize being 7)
```

```
        static_assert(kHeaderSize == 7, "");
        dest_->Append(Slice("\x00\x00\x00\x00\x00\x00", leftover));
      }
      block_offset_ = 0;
    }
    // Invariant: we never leave < kHeaderSize bytes in a block.
    assert(kBlockSize - block_offset_ - kHeaderSize >= 0);
    //保证当前块可用容量在减去header空间后仍有余量（大于等于0）
    const size_t avail = kBlockSize - block_offset_ - kHeaderSize;//当前块的可储存数据的可用余量
    //判断要追加写入的整体数据是否需要分片（块大小是有限的，要写入的一个Slice的数据可能跨越两个或三个块），
    const size_t fragment_length = (left < avail) ? left : avail;
    RecordType type;
    const bool end = (left == fragment_length);
    if (begin && end) {
      type = kFullType;
    } else if (begin) {
      type = kFirstType;
    } else if (end) {
      type = kLastType;
    } else {
      type = kMiddleType;
    }

    //log:: Writer类中的唯一一个私有方法
    s = EmitPhysicalRecord(type, ptr, fragment_length);

    ptr += fragment_length;
    left -= fragment_length;
    begin = false;
  } while (s.ok() && left > 0);
  return s;
}


enum RecordType {
  // Zero is reserved for preallocated files
  kZeroType = 0,
  kFullType = 1,
  // For fragments
  kFirstType = 2,
  kMiddleType = 3,
  kLastType = 4
};
static const int kMaxRecordType = kLastType;
static const int kBlockSize = 32768;  //32KB
// Header is checksum (4 bytes), length (2 bytes), type (1 byte).
static const int kHeaderSize = 4 + 2 + 1;


        dest_->Append(Slice("\x00\x00\x00\x00\x00\x00", leftover));

        s = EmitPhysicalRecord(type, ptr, fragment_length);
```

```
Status Writer::EmitPhysicalRecord(RecordType t, const char* ptr,
                                  size_t length) {
  assert(length <= 0xffff);  // Must fit in two bytes
  assert(block_offset_ + kHeaderSize + length <= kBlockSize);

//首先构建header中的格式和内容，4+2+1（4B crc+2B length+1B的record type），存储在buf中
  // Format the header
  char buf[kHeaderSize];
  buf[4] = static_cast<char>(length & 0xff);
  buf[5] = static_cast<char>(length >> 8);
  buf[6] = static_cast<char>(t);
  // Compute the crc of the record type and the payload.
  uint32_t crc = crc32c::Extend(type_crc_[t], ptr, length);
  crc = crc32c::Mask(crc);  // Adjust for storage
  EncodeFixed32(buf, crc);

//将buf中的内容header和ptr指向的Slice数据追加到dest_中
  // Write the header and the payload
  Status s = dest_->Append(Slice(buf, kHeaderSize));
  if (s.ok()) {
    s = dest_->Append(Slice(ptr, length));
    if (s.ok()) {
      s = dest_->Flush();
    }
  }
  block_offset_ += kHeaderSize + length;
  return s;
}
```

dest_->Append (Slice)

```
  WritableFile* dest_;
```

```
class LEVELDB_EXPORT WritableFile {
 public:
  WritableFile() = default;
  WritableFile(const WritableFile&) = delete;
  WritableFile& operator=(const WritableFile&) = delete;
  virtual ~WritableFile();
  virtual Status Append(const Slice& data) = 0;
  virtual Status Close() = 0;
  virtual Status Flush() = 0;
  virtual Status Sync() = 0;
};
```

```
  // buf_[0, pos_ - 1] contains data to be written to fd_.
  char buf_[kWritableFileBufferSize];
  size_t pos_;
```

```
  int fd_;
  const bool is_manifest_;  // True if the file's name starts with MANIFEST.
  const std::string filename_;
  const std::string dirname_;  // The directory of filename_.

   constexpr const size_t kWritableFileBufferSize = 65536;


class PosixWritableFile final : public WritableFile {
 public:
   //首先是构造函数与析构函数
  PosixWritableFile(std::string filename, int fd)
      : pos_(0),
        fd_(fd),
        is_manifest_(IsManifest(filename)),
        filename_(std::move(filename)),
        dirname_(Dirname(filename_)) {}
  ~PosixWritableFile() override {
    if (fd_ >= 0) {
      // Ignoring any potential errors
      Close();
    }
  }

  Status Append(const Slice& data) override {
    size_t write_size = data.size();
    const char* write_data = data.data();

   //主要是调用std::memcpy函数将要追加写入的数据存到该类的私有成员变量
buf_[kWritableFileBufferSize]当中，pos_代表当前缓冲区buf_中数据的位置索引
  //然后更新相应索引或指针位置，返回OK状态。
    // Fit as much as possible into buffer.
    size_t copy_size = std::min(write_size, kWritableFileBufferSize - pos_);
    std::memcpy(buf_ + pos_, write_data, copy_size);
    write_data += copy_size;
    write_size -= copy_size;
    pos_ += copy_size;
    if (write_size == 0) {
      return Status::OK();
    }

  //如果数据无法完全放入缓冲区buf_中，则先刷新缓冲区
  //刷新缓冲区的过程主要调用该类中的成员函数FlushBuffer（）
  // FlushBuffer （）将缓冲区的数据写入文件（写入磁盘中），然后清空缓冲区
  //事实上，FlushBuffer（）这个函数内部也是调用WriteUnbuffered（）函数
    // Can't fit in buffer, so need to do at least one write.
    Status status = FlushBuffer();
    if (!status.ok()) {
      return status;
    }

  //如果剩余数据量小于缓冲区的大小，那么先放入到缓冲区，然后返回OK即可
  //否则，数据量大于或等于缓冲区大小，则调用WriteUnbuffered（）,直接将这些数据直接写入文
件，而不通过缓冲区。
```

```
    // Small writes go to buffer, large writes are written directly.
    if (write_size < kWritableFileBufferSize) {
      std::memcpy(buf_, write_data, write_size);
      pos_ = write_size;
      return Status::OK();
    }
    return WriteUnbuffered(write_data, write_size);
  }
```

我们再分析一下Append这个追加的过程，总结就是以下三点：

1. **缓冲区优化：**
   该方法首先尝试将数据放入内部缓冲区，以减少频繁的磁盘I/O操作；
2. **数据处理：**
   如果数据量较大或缓冲区已满，就会将缓冲区内容先写入磁盘，然后根据剩余数据量选择继续使用缓冲区或直接写入文件；
3. **高效写入：**
   小数据块优先放入缓冲区，大数据块则直接写入文件，以实现更高效的写入操作。

```
//close()函数内部实际上调用了POSIX中的系统调用close（fd_）用以关闭该文件
  Status Close() override {
    Status status = FlushBuffer();
    const int close_result = ::close(fd_);
    if (close_result < 0 && status.ok()) {
      status = PosixError(filename_, errno);
    }
    fd_ = -1;
    return status;
  }


  Status Flush() override { return FlushBuffer(); }
  Status Sync() override {
    // Ensure new files referred to by the manifest are in the filesystem.
    //
    // This needs to happen before the manifest file is flushed to disk, to
    // avoid crashing in a state where the manifest refers to files that are not
    // yet on disk.
    Status status = SyncDirIfManifest();
    if (!status.ok()) {
      return status;
    }
    status = FlushBuffer();
    if (!status.ok()) {
      return status;
    }
    return SyncFd(fd_, filename_);
  }
 private:
  Status FlushBuffer() {
    Status status = WriteUnbuffered(buf_, pos_);
    pos_ = 0;
    return status;
  }
```

```
//WriteUnbuffered（）这个函数实现了将数据写入文件的过程，而不使用缓冲区（事实上缓冲区buf_
满了之后也会调用这个函数WriteUnbuffered(buf_, pos_)，这个过程分装在FlushBuffer（）函数中；
//在该函数内部直接调用了POSIX的”write“系统调用用来执行写入。

  Status WriteUnbuffered(const char* data, size_t size) {
    while (size > 0) {
      //调用系统调用write来执行写入
      ssize_t write_result = ::write(fd_, data, size);
      //write_result为负数，说明写入失败；
      //不为负数时，write_result代表返回成功写入的字节数
      if (write_result < 0) {
        //检查errno是否为EINTR（表示系统调用被中断）。如果是，调用continue，重新尝试写入
        if (errno == EINTR) {
          continue;  // Retry
        }
        //如果errno不是EINTR，调用PosixError（filename_，errno）生成一个错误状态并返回。
        //PosixError会返回一个表示系统调用错误的Status对象。
        return PosixError(filename_, errno);
      }
      data += write_result;
      size -= write_result;
    }
    return Status::OK();
  }
//注意：在上面这个函数中给的系统调用write过程可能不能一次性写入所有数据，因此该函数采用
while循环的方式，处理剩余的数据写入。


  Status SyncDirIfManifest() {
    Status status;
    if (!is_manifest_) {
      return status;
    }
    int fd = ::open(dirname_.c_str(), O_RDONLY | kOpenBaseFlags);
    if (fd < 0) {
      status = PosixError(dirname_, errno);
    } else {
      status = SyncFd(fd, dirname_);
      ::close(fd);
    }
    return status;
  }
  // Ensures that all the caches associated with the given file descriptor's
  // data are flushed all the way to durable media, and can withstand power
  // failures.
  //
  // The path argument is only used to populate the description string in the
  // returned Status if an error occurs.
  static Status SyncFd(int fd, const std::string& fd_path) {
#if HAVE_FULLFSYNC
    // On macOS and iOS, fsync() doesn't guarantee durability past power
    // failures. fcntl(F_FULLFSYNC) is required for that purpose. Some
    // filesystems don't support fcntl(F_FULLFSYNC), and require a fallback to
    // fsync().
```

```
    if (::fcntl(fd, F_FULLFSYNC) == 0) {
      return Status::OK();
    }
#endif  // HAVE_FULLFSYNC
#if HAVE_FDATASYNC
    bool sync_success = ::fdatasync(fd) == 0;
#else
    bool sync_success = ::fsync(fd) == 0;
#endif  // HAVE_FDATASYNC
    if (sync_success) {
      return Status::OK();
    }
    return PosixError(fd_path, errno);
  }
  // Returns the directory name in a path pointing to a file.
  //
  // Returns "." if the path does not contain any directory separator.
  static std::string Dirname(const std::string& filename) {
    std::string::size_type separator_pos = filename.rfind('/');
    if (separator_pos == std::string::npos) {
      return std::string(".");
    }
    // The filename component should not contain a path separator. If it does,
    // the splitting was done incorrectly.
    assert(filename.find('/', separator_pos + 1) == std::string::npos);
    return filename.substr(0, separator_pos);
  }
  // Extracts the file name from a path pointing to a file.
  //
  // The returned Slice points to |filename|'s data buffer, so it is only valid
  // while |filename| is alive and unchanged.
  static Slice Basename(const std::string& filename) {
    std::string::size_type separator_pos = filename.rfind('/');
    if (separator_pos == std::string::npos) {
      return Slice(filename);
    }
    // The filename component should not contain a path separator. If it does,
    // the splitting was done incorrectly.
    assert(filename.find('/', separator_pos + 1) == std::string::npos);
    return Slice(filename.data() + separator_pos + 1,
                 filename.length() - separator_pos - 1);
  }
  // True if the given file is a manifest file.
  static bool IsManifest(const std::string& filename) {
    return Basename(filename).starts_with("MANIFEST");
  }
  // buf_[0, pos_ - 1] contains data to be written to fd_.
  char buf_[kWritableFileBufferSize];
  size_t pos_;
  int fd_;
  const bool is_manifest_;  // True if the file's name starts with MANIFEST.
  const std::string filename_;
  const std::string dirname_;  // The directory of filename_.
};
```

LevelDB笔记6：Comparator

2024年8月17日 11:30

LevelDB笔记7：WriteBatch

2024年8月19日 20:22

这是WriteBatch类：

```
class LEVELDB_EXPORT WriteBatch {
 public:
  class LEVELDB_EXPORT Handler {
   public:
    virtual ~Handler();
    virtual void Put(const Slice& key, const Slice& value) = 0;
    virtual void Delete(const Slice& key) = 0;
  };
  WriteBatch();
  // Intentionally copyable.
  WriteBatch(const WriteBatch&) = default;
  WriteBatch& operator=(const WriteBatch&) = default;
  ~WriteBatch();
  // Store the mapping "key->value" in the database.
  void Put(const Slice& key, const Slice& value);
  // If the database contains a mapping for "key", erase it.  Else do nothing.
  void Delete(const Slice& key);
  // Clear all updates buffered in this batch.
  void Clear();
  // The size of the database changes caused by this batch.
  //
  // This number is tied to implementation details, and may change across
  // releases. It is intended for LevelDB usage metrics.
  size_t ApproximateSize() const;
  // Copies the operations in "source" to this batch.
  //
  // This runs in O(source size) time. However, the constant factor is better
  // than calling Iterate() over the source batch with a Handler that replicates
  // the operations into this batch.
  void Append(const WriteBatch& source);
  // Support for iterating over the contents of a batch.
  Status Iterate(Handler* handler) const;
 private:
  friend class WriteBatchInternal;
  std::string rep_;  // See comment in write_batch.cc for the format of rep_
};
```

WriteBatch这个类包含一个私有的成员变量：

```
   std::string rep_;
```

包含一个私有的友元类：

```
  friend class WriteBatchInternal;
```

这是WriteBatchInternal类的定义：

```
// WriteBatchInternal provides static methods for manipulating a
// WriteBatch that we don't want in the public WriteBatch interface.
class WriteBatchInternal {
 public:
  // Return the number of entries in the batch.
  static int Count(const WriteBatch* batch);
  // Set the count for the number of entries in the batch.
  static void SetCount(WriteBatch* batch, int n);
  // Return the sequence number for the start of this batch.
  static SequenceNumber Sequence(const WriteBatch* batch);
  // Store the specified number as the sequence number for the start of
  // this batch.
  static void SetSequence(WriteBatch* batch, SequenceNumber seq);
  static Slice Contents(const WriteBatch* batch) { return Slice(batch->rep_); }
  static size_t ByteSize(const WriteBatch* batch) { return batch->rep_.size(); }
  static void SetContents(WriteBatch* batch, const Slice& contents);
  static Status InsertInto(const WriteBatch* batch, MemTable* memtable);
  static void Append(WriteBatch* dst, const WriteBatch* src);
};
```

```
// WriteBatch header has an 8-byte sequence number followed by a 4-byte count.
static const size_t kHeader = 12;
WriteBatch::WriteBatch() { Clear(); }
WriteBatch::~WriteBatch() = default;
WriteBatch::Handler::~Handler() = default;
void WriteBatch::Clear() {
  rep_.clear();
  rep_.resize(kHeader);
}
```

我们先单纯来看WriteBatch类中外部接口Put函数：

```
void WriteBatch::Put(const Slice& key, const Slice& value) {
  WriteBatchInternal::SetCount(this, WriteBatchInternal::Count(this) + 1);
  //将kTypeValue（0x01）静态转换为char类型（一个字节），以字符串的形式追加到rep_的末尾
  rep_.push_back(static_cast<char>(kTypeValue));
  //在rep_中追加键值对
  PutLengthPrefixedSlice(&rep_, key);
  PutLengthPrefixedSlice(&rep_, value);
}
```

它调用了相应的接口：

先看

```
  WriteBatchInternal::SetCount(this, WriteBatchInternal::Count(this) + 1);
```

定义如下：

```
void WriteBatchInternal::SetCount(WriteBatch* b, int n) {
  EncodeFixed32(&b->rep_[8], n);
}
```

SetCount() 函数第二个参数调用了WriteBatchInternal：：count(this) 这个函数，定义如下：

```
int WriteBatchInternal::Count(const WriteBatch* b) {
  return DecodeFixed32(b->rep_.data() + 8);
}
```

事实上Count（）这个函数返回的是b这个对象的count数值，rep_结构中的count数值。

来看DecodeFixed32（）这个函数，这个函数接受一个const char*类型的指针，事实上再调用过程中，它指向一个32位的字符类型数组首地址，也即是4B。这个函数主要实现强制类型转换的过程，将char*指针指向的那块4B的数据解析出来，1B接着1B的拼接起来形成返回值uint323_t.
除此之外数据按照小端方式存放。

```
inline uint32_t DecodeFixed32(const char* ptr) {
  //首先，ptr被重新解释为一个指向uint8_t的指针，并存储在buffer中。
  const uint8_t* const buffer = reinterpret_cast<const uint8_t*>(ptr);
  // Recent clang and gcc optimize this to a single mov / ldr instruction.
  //这里的注释信息提到的意思是，现代的clang和gcc编译器都会将这段代码优化为单个汇编指令，例如mov或者ldr。这意味着编译器会自动生成高效的机器码来读取32位数据，而不需要逐字节处理。
  return (static_cast<uint32_t>(buffer[0])) |
         (static_cast<uint32_t>(buffer[1]) << 8) |
         (static_cast<uint32_t>(buffer[2]) << 16) |
         (static_cast<uint32_t>(buffer[3]) << 24);
}
```

小端：数据的最低有效字节存储在内存的最低地址（即首位）；
大端：数据的最高有效字节存储在内存的最低地址。

那么下面这个函数的作用呢?

```
 EncodeFixed32(&b->rep_[8], n);
```

我们来看看这个函数定义：

```
inline void EncodeFixed32(char* dst, uint32_t value) {
  uint8_t* const buffer = reinterpret_cast<uint8_t*>(dst);
  // Recent clang and gcc optimize this to a single mov / str instruction.
  buffer[0] = static_cast<uint8_t>(value);
  buffer[1] = static_cast<uint8_t>(value >> 8);
  buffer[2] = static_cast<uint8_t>(value >> 16);
  buffer[3] = static_cast<uint8_t>(value >> 24);
}
```

总的来说就是将value的值存到dst指向的那块存储空间中，在

```
 EncodeFixed32(&b->rep_[8], n);
```

这个调用的目的就是将n的值更新原来rep_结构中的count值。

来看看是如何将键值对追加到rep_中的，主要是调用了下面这个函数：

```
  PutLengthPrefixedSlice(&rep_, key);
```

定义如下：

```
void PutLengthPrefixedSlice(std::string* dst, const Slice& value) {
  PutVarint32(dst, value.size());
  dst->append(value.data(), value.size());
}
```

很简单就连个过程一个是追加value的长度（可变），一个就是调用append（）追加value本身的值

PutVarint32（）函数的定义如下：

```
void PutVarint32(std::string* dst, uint32_t v) {
  char buf[5];
  char* ptr = EncodeVarint32(buf, v);
  dst->append(buf, ptr - buf);
}
```

EncodeVarint32() 该函数，用于将一个32位无符号整数v编码为变长整数格式，并将其写入到一个字符缓冲区dst中

变长整数编码是一种压缩数字的方法，它根据数字的大小选择不同的字节数进行存储，从而节省空间。

```
char* EncodeVarint32(char* dst, uint32_t v) {
  // Operate on characters as unsigneds
  uint8_t* ptr = reinterpret_cast<uint8_t*>(dst);
  static const int B = 128;
  if (v < (1 << 7)) {
    *(ptr++) = v;
  } else if (v < (1 << 14)) {
    *(ptr++) = v | B;
    *(ptr++) = v >> 7;
  } else if (v < (1 << 21)) {
    *(ptr++) = v | B;
    *(ptr++) = (v >> 7) | B;
    *(ptr++) = v >> 14;
  } else if (v < (1 << 28)) {
    *(ptr++) = v | B;
    *(ptr++) = (v >> 7) | B;
    *(ptr++) = (v >> 14) | B;
    *(ptr++) = v >> 21;
  } else {
    *(ptr++) = v | B;
    *(ptr++) = (v >> 7) | B;
    *(ptr++) = (v >> 14) | B;
    *(ptr++) = (v >> 21) | B;
    *(ptr++) = v >> 28;
  }
  return reinterpret_cast<char*>(ptr);
}
```

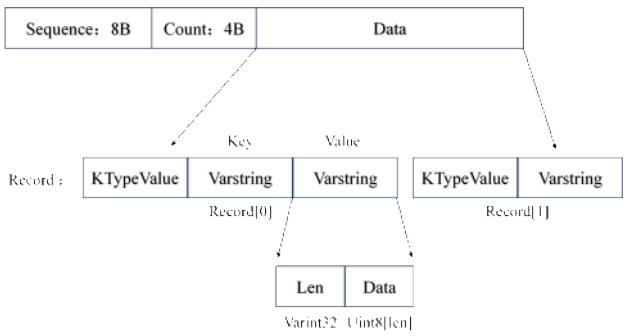
此外，在WriteBatch类中，有一个私有的成员变量：

```
  std::string rep_;
```

这个变量的含义是什么呢，以及它为什么是string类型的呢?

```
// WriteBatch::rep_ :=
//    sequence: fixed64
//    count: fixed32
//    data: record[count]
// record :=
//    kTypeValue varstring varstring         |
//    kTypeDeletion varstring
// varstring :=
//    len: varint32
//    data: uint8[len]
```

这些注释是write_batch.cc中的注释。如下图所示，即为std::string rep_的结构。



```
enum ValueType { kTypeDeletion = 0x0, kTypeValue = 0x1 };
```
kTypeValue的类型枚举出来了。

此外，WriteBatchInternal类中有如下的成员方法：
```
 static Status InsertInto(const WriteBatch* batch, MemTable* memtable);
```
在DBImpl：Write（）这个过程中被调用：
```
if (status.ok()) {
        status = WriteBatchInternal::InsertInto(write_batch, mem_);
      }
```
看起来作用像是把要写的这一批次，写入到内存mem_中

具体来看该函数的定义：
```
Status WriteBatchInternal::InsertInto(const WriteBatch* b, MemTable* memtable) {
  MemTableInserter inserter;
  inserter.sequence_ = WriteBatchInternal::Sequence(b);
  inserter.mem_ = memtable;
  return b->Iterate(&inserter);
}
```

先看看MemTableInserter（）这个玩意是个啥?
```
namespace {
class MemTableInserter : public WriteBatch::Handler {
 public:
  SequenceNumber sequence_;
  MemTable* mem_;
  void Put(const Slice& key, const Slice& value) override {
    mem_->Add(sequence_, kTypeValue, key, value);
    sequence_++;
  }
  void Delete(const Slice& key) override {
    mem_->Add(sequence_, kTypeDeletion, key, Slice());
    sequence_++;
  }
};
}  // namespace
```
Handler是WriteBatch类中的成员类
```
  class LEVELDB_EXPORT Handler {
   public:
    virtual ~Handler();
    virtual void Put(const Slice& key, const Slice& value) = 0;
    virtual void Delete(const Slice& key) = 0;
  };
```
MemTableInserter类继承于上面这个类，实现了两个成员方法：Put和Delete，还包含两个成员变量：
```
  SequenceNumber sequence_;
  MemTable* mem_;
```

在InsertInfo（）函数中的这两步骤：
```
  inserter.sequence_ = WriteBatchInternal::Sequence(b);
  inserter.mem_ = memtable;
```

Sequence（b）的定义如下：
```
SequenceNumber WriteBatchInternal::Sequence(const WriteBatch* b) {
  return SequenceNumber(DecodeFixed64(b->rep_.data()));
}
```
？这里有个问题SequenceNumber是啥？为什么需要这个东西

其实他的定义如下：他是64位的无符号整数。
```
typedef uint64_t SequenceNumber;
```

最后看这一句：
```
  return b->Iterate(&inserter);
```
来看看WriteBatch类中的这个Iterate（Handler*）这个成员方法是啥样的具体是

```
Status WriteBatch::Iterate(Handler* handler) const {
  Slice input(rep_);
  if (input.size() < kHeader) {
    return Status::Corruption("malformed WriteBatch (too small)");
  }
  input.remove_prefix(kHeader);    //rep_结构中跳过sequenceNumber和Count
  Slice key, value;
  int found = 0;
  while (!input.empty()) {
    found++;
    char tag = input[0];    //input[0]此时指向的就是Ktype
```

```
    input.remove_prefix(1);  //移动input位置，指向具体的键
    switch (tag) {
      case kTypeValue:
        if (GetLengthPrefixedSlice(&input, &key) &&
            GetLengthPrefixedSlice(&input, &value)) {
          handler->Put(key, value);          //调用Put () 函数
        } else {
          return Status::Corruption("bad WriteBatch Put");
        }
        break;
      case kTypeDeletion:
        if (GetLengthPrefixedSlice(&input, &key)) {
          handler->Delete(key);          //调用Delete () 函数
        } else {
          return Status::Corruption("bad WriteBatch Delete");
        }
        break;
      default:
        return Status::Corruption("unknown WriteBatch tag");
    }
  }
   //检查遍历的操作数是否与批处理的记录数量一致
  if (found != WriteBatchInternal::Count(this)) {
    return Status::Corruption("WriteBatch has wrong count");
  } else {
    return Status::OK();
  }
}
```

可以看到上述代码重点插入删除调用的是Put和Delete函数，这两在

`class MemTableInserter : public WriteBatch::Handler{}`中

下面是定义：

```
  void Put(const Slice& key, const Slice& value) override {
    mem_->Add(sequence_, kTypeValue, key, value);
    sequence_++;
  }
  void Delete(const Slice& key) override {
    mem_->Add(sequence_, kTypeDeletion, key, Slice());
    sequence_++;
  }
```

可以清楚的看到主要是调用mem中的Add方法，这个内容参见LevelDB笔记4：MemTable。

LevelDB笔记8：再谈Write

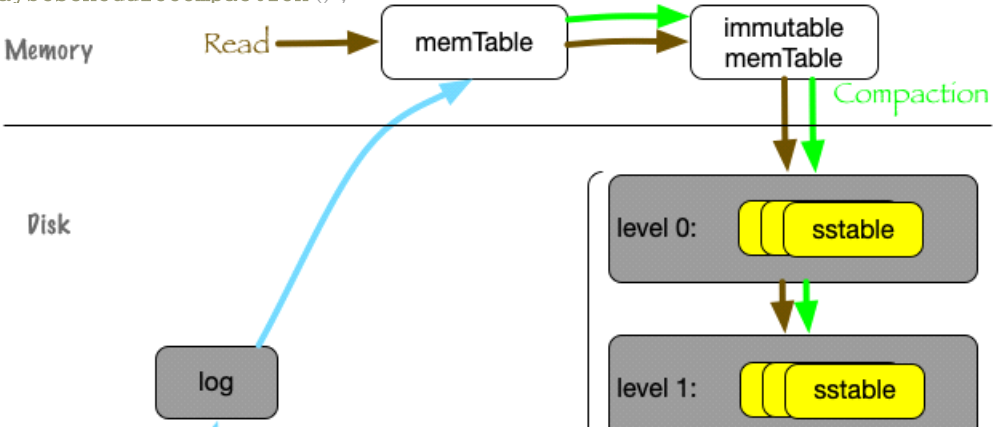
2024年8月20日　　21:07

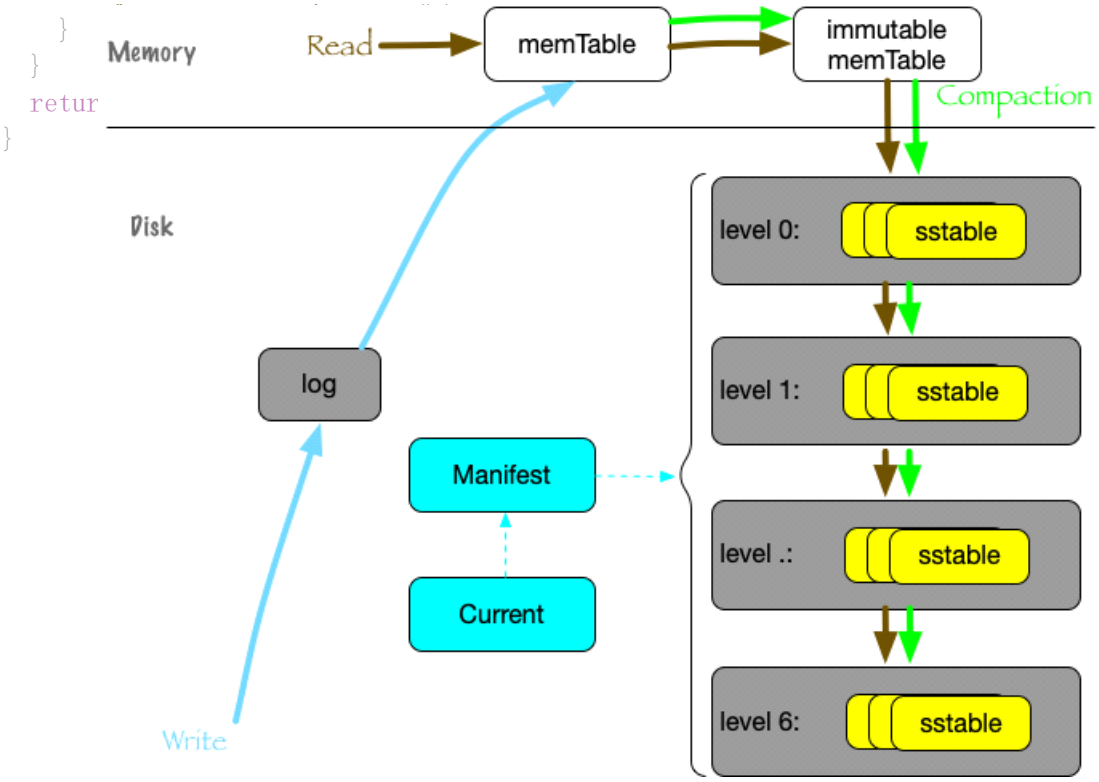
我们再来从这个Write过程入手分析一下整个写入过程。
在之前的笔记中已经讲了数据是如何写入到log然后写入到mem中的，也就是Write函数下面所示的黄色代码，所示的过程所实现的。
那么，写入mem后，需要持久化到磁盘中吧，那么它是如何持久化的呢？
蓝色部分的过程内部恰好包含了该步骤，即MakeRoomForWrite（）函数。
本章再来先从整体出发，重点分析该函数以及它里面调用了哪些方法哪些结构，实现怎么实现的等，先做一个全局的流程分析。再在之后的笔记中介绍相应的结构和方法等。

```
Status DBImpl::Write(const WriteOptions& options, WriteBatch* updates) {
  Writer w(&mutex_);
  w.batch = updates;
  w.sync = options.sync;
  w.done = false;
  MutexLock l(&mutex_);
  writers_.push_back(&w);
  while (!w.done && &w != writers_.front()) {
    w.cv.Wait();
  }
  if (w.done) {
    return w.status;
  }
  // May temporarily unlock and wait.
  Status status = MakeRoomForWrite(updates == nullptr);
  uint64_t last_sequence = versions_->LastSequence();
  Writer* last_writer = &w;
  if (status.ok() && updates != nullptr) {  // nullptr batch is for compactions
    WriteBatch* write_batch = BuildBatchGroup(&last_writer);
    WriteBatchInternal::SetSequence(write_batch, last_sequence + 1);
    last_sequence += WriteBatchInternal::Count(write_batch);
    // Add to log and apply to memtable.  We can release the lock
    // during this phase since &w is currently responsible for logging
    // and protects against concurrent loggers and concurrent writes
    // into mem_.
    {
      mutex_.Unlock();
      status = log_->AddRecord(WriteBatchInternal::Contents(write_batch));
      bool sync_error = false;
      if (status.ok() && options.sync) {
        status = logfile_->Sync();
        if (!status.ok()) {
          sync_error = true;
        }
      }
      if (status.ok()) {
        status = WriteBatchInternal::InsertInto(write_batch, mem_);
      }
```

```
      mutex_.Lock();
      if (sync_error) {
        // The state of the log file is indeterminate: the log record we
        // just added may or may not show up when the DB is re-opened.
        // So we force the DB into a mode where all future writes fail.
        RecordBackgroundError(status);
      }
    }
    if (write_batch == tmp_batch_) tmp_batch_->Clear();
    versions_->SetLastSequence(last_sequence);
  }
  while (true) {
    Writer* ready = writers_.front();
    writers_.pop_front();
    if (ready != &w) {
      ready->status = status;
      ready->done = true;
      ready->cv.Signal();
    }
    if (ready == last_writer) break;
  }
  // Notify new head of write queue
  if (!writers_.empty()) {
    writers_.front()->cv.Signal();
  }
  return status;
}



// REQUIRES: mutex_ is held
// REQUIRES: this thread is currently at the front of the writer queue
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) >=
                                  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_;
      s = logfile_->Close();
      if (!s.ok()) {
        // We may have lost some data written to the previous log file.
        // Switch to the new log file anyway, but record as a background
        // error so we do not attempt any more writes.
        //
        // We could perhaps attempt to save the memtable corresponding
        // to log file and suppress the error if that works, but that
        // would add more complexity in a critical code path.
        RecordBackgroundError(s);
      }
      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();
    }
  }
  retur
}
```





上面MakeRoomForWrite(bool)函数看起来很嘈杂，实际上主要构成是一个while循环，如下所示：

```
Status DBImpl::MakeRoomForWrite(bool force) {
  mutex_.AssertHeld();
  assert(!writers_.empty());
  bool allow_delay = !force;
  Status s;
  while (true) {
    if (!bg_error_.ok()) {…
    } else if (allow_delay && versions_->NumLevelFiles(0) >=
                              config::kL0_SlowdownWritesTrigger) {…
    } else if (!force &&
               (mem_->ApproximateMemoryUsage() <= options_.write_buffer_size)) {…
    } else if (imm_ != nullptr) {…
    } else if (versions_->NumLevelFiles(0) >= config::kL0_StopWritesTrigger) {…
    } else {…
  }
  return s;
}
```

Write（）调用MakeRoomForWrite（）传进来的参数是updates==null
里面含有多个if else语句，下面我们逐个的进行分析：

1. 第一个if

```
if (!bg_error_.ok()) {
      // Yield previous error
      s = bg_error_;
      break;
```

其中bg_error是DBImpl类中的一个成员变量：

```
  Status bg_error_ GUARDED_BY(mutex_);
  // Have we encountered a background error in paranoid mode?
```

这是他的注释，总之感官上就是判断是否出错了，有错了则返回即可；

2. 第二个if

```
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();
  }
```

其中kL0_SlowdownWritesTrigger的定义如下：

```
// Soft limit on number of level-0 files.  We slow down writes at this point.
static const int kL0_SlowdownWritesTrigger = 8;
```

整体来看就是判断第L0层文件是否超过了一个设定好的阈值kL0_SlowdownWritesTrigger，如果是里面调用了env_的SleepForMicroseconds(1000)，来实现延迟写入速度的目的，以避免过多的L0层文件的堆积；

为什么需要控制？？？这里先做一个简单介绍，要理解这里先理解LevelDB中的Compaction操作。

LevelDB的L0层使用的是“无排序合并”的方式，L0层文件越多，查询时需要检查的文件数量就越多，这一方面增加了查询延迟，还会增加写入操作所需的合并（compaction）工作量。

短暂延迟设计：

- 如果系统检测到 L0 文件数量接近触发点，则在写入操作中加入短暂的延迟（1毫秒）。这是为了平衡系统负载，减少高峰时的写入延迟差异，并为后台压缩线程争取时间。
- 这种设计有助于将写入负载均匀分布，避免在硬限制达到后突然长时间的写入阻塞，从而提升系统的稳定性。

释放CPU资源：

在延迟写入的同时，“mutex_.Unlock（）"将会释放互斥锁，使得后台压缩线程有机会获得CPU资源。如果压缩线程和写入线程共享一个核心，这样做可以有效利用CPU资源，减少写入操作和后台压缩操作之间的竞争。

最后设置：

```
    allow_delay = false;  // Do not delay a single write more than once
```

意思是确保单个写入操作中不会产生多次延迟。因为这个while循环，当当前这个延迟时间结束后再次执行while循环，不会再次调用这个if里面的情况了。这样的设置，可以避免在较高负载下，单次写入操作被多次延迟，从而造成不必要的写入延迟增加。

总之设计目的主要就以下三点：

- **防止 L0 文件过载**：
  - 如果 L0 层文件数量过多，可能导致查询性能急剧下降，并且后续写入操作将变得非常昂贵（因为需要大量的合并操作）。通过延迟写入，可以控制 L0 文件的增长速度，避免达到硬限制。
- **平滑写入延迟**：

- 在负载高峰时，系统通过短暂的延迟来均衡写入负载，避免在达到硬限制后突然遇到非常长的写入延迟，从而实现更平滑的性能表现。

- **优化 CPU 资源分配**：
  - 通过短暂延迟并释放互斥锁，系统允许后台压缩线程在高负载时获得更多的 CPU 资源，优化了整体性能。

3. 第三个if

```
else if (!force &&
             (mem_->ApproximateMemoryUsage() <= options_.write_buffer_size)) {
    // There is room in current memtable
    break;
  }
```

这个看起来就一个判断语句，如果当前memtable块中仍余量充足，那么直接返回即可。

4. 第四个if

```
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();
  }
```

Imm_也是MemTable类型的指针，它指向Memtable being compacted（即immutable Memtable），如果它非空也就是说compact操作仍然进行着，这时需要暂且等待。
并且将Current memtable full；waiting……这个显示信息记录到log中；
最后调用background_work_finished_signal_.Wait()进行wait。等待压缩操作完成后，再继续。

5. 第五个if

```
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();
  }
```

这个也是一样，如果level0文件的数量超过了kL0_StopWritesTrigger，系统将暂停写入操作并等待后台工作完成。

6. 以上几种if都不满足

```
else {
    // Attempt to switch to a new memtable and trigger compaction of old
    assert(versions_->PrevLogNumber() == 0);
```

//断言确保在启动新的memtable之前，之前的日志文件号已经被清零。日志文件号用于追踪当前日志文件的memtable

```
    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;
    }
```

//生成新的日志文件，如果文件创建失败，则会重用刚才分配的文件号，并退出该逻辑。这样做的目的是为了避免在紧凑循环中不断消耗文件号空间。

//然后关闭旧的日志文件

```
      delete log_;
      s = logfile_->Close();

      if (!s.ok()) {
        // We may have lost some data written to the previous log file.
        // Switch to the new log file anyway, but record as a background
        // error so we do not attempt any more writes.
        //
        // We could perhaps attempt to save the memtable corresponding
        // to log file and suppress the error if that works, but that
        // would add more complexity in a critical code path.
        RecordBackgroundError(s);
      }
      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();
    }
```

先理解总体含义：

主要是处理当前memtable（内存表）已满的情况，在这种情况下，系统需要为新的写入操作腾出空间。这段代码主要涉及切换到一个新的memtable，并将当前的memtable标记为不可变的immutable memtable，以便在后台触发压缩（compaction）操作。

最后调用了MaybeScheduleCompaction()函数

定义如下

再来细节的看代码。注释写在代码上。

```
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);
  }
}
```

1. **MemTable 和 Log 文件的关系：**
   - 当 LevelDB 接收到写操作时，数据会被写入当前的 MemTable 和对应的日志文件（Log File）。

- 日志文件用于持久化数据，防止数据在系统崩溃或重启时丢失。每次写入操作首先会被追加到日志文件中，确保即使系统崩溃，日志文件中的数据也可以在重启时重放以恢复 MemTable 的内容。
- 当一个 MemTable 被填满时，LevelDB 会切换到一个新的 MemTable，并同时创建一个新的日志文件来记录后续的写操作。

2. **多个 MemTable 与日志文件：**
   - 如果当前 MemTable 已满并且被标记为不可变（Immutable），而新的 MemTable 开始接收写操作，那么在 Immutable MemTable 还没有被压缩（Compaction）并写入 SST 文件之前，它仍然保留着旧的日志文件。
   - 此时，新的 MemTable 会有一个新的日志文件。因此，在某些场景下，可能会有多个日志文件存在，但一般一个 MemTable 对应一个日志文件。
3. **日志文件的重用：**
   - LevelDB 允许日志文件号的重用。如果日志文件成功关闭，并且所有写入操作都被确认持久化，日志文件号可能会被重用以避免浪费文件号空间。

总结

虽然通常来说，一个 MemTable 会对应一个日志文件，但在系统运行过程中，有时可能会有多个日志文件存在，尤其是在进行后台压缩操作时。因此，不能严格地说每个 MemTable 只对应一个日志文件，但大多数情况下，这种对应关系是成立的。

注：这个内容是GPT生成的，不过感觉生成的没毛病。

em，先分析到此，笔记9就准备介绍最后黄色标记的这一句：

```
    env_->Schedule(&DBImpl::BGWork, this);
```

怎么去执行Schedule的，以及env_这是什么东西?

LevelDB笔记9：Env

2024年8月22日　　14:36

```
    env_->Schedule(&DBImpl::BGWork, this);
```

接着笔记8

```
class LEVELDB_EXPORT Env {
 public:
  Env();
  Env(const Env&) = delete;
  Env& operator=(const Env&) = delete;
  virtual ~Env();
  // Return a default environment suitable for the current operating
  // system.  Sophisticated users may wish to provide their own Env
  // implementation instead of relying on this default environment.
  //
  // The result of Default() belongs to leveldb and must never be deleted.
  static Env* Default();
  // Create an object that sequentially reads the file with the specified name.
  // On success, stores a pointer to the new file in *result and returns OK.
  // On failure stores nullptr in *result and returns non-OK.  If the file does
  // not exist, returns a non-OK status.  Implementations should return a
  // NotFound status when the file does not exist.
  //
  // The returned file will only be accessed by one thread at a time.
  virtual Status NewSequentialFile(const std::string& fname,
                                   SequentialFile** result) = 0;
  // Create an object supporting random-access reads from the file with the
  // specified name.  On success, stores a pointer to the new file in
  // *result and returns OK.  On failure stores nullptr in *result and
  // returns non-OK.  If the file does not exist, returns a non-OK
  // status.  Implementations should return a NotFound status when the file does
  // not exist.
  //
  // The returned file may be concurrently accessed by multiple threads.
  virtual Status NewRandomAccessFile(const std::string& fname,
                                     RandomAccessFile** result) = 0;
  // Create an object that writes to a new file with the specified
  // name.  Deletes any existing file with the same name and creates a
  // new file.  On success, stores a pointer to the new file in
  // *result and returns OK.  On failure stores nullptr in *result and
  // returns non-OK.
  //
  // The returned file will only be accessed by one thread at a time.
  virtual Status NewWritableFile(const std::string& fname,
                                 WritableFile** result) = 0;
  // Create an object that either appends to an existing file, or
  // writes to a new file (if the file does not exist to begin with).
  // On success, stores a pointer to the new file in *result and
  // returns OK.  On failure stores nullptr in *result and returns
  // non-OK.
  //
```

```
  // The returned file will only be accessed by one thread at a time.
  //
  // May return an IsNotSupportedError error if this Env does
  // not allow appending to an existing file.  Users of Env (including
  // the leveldb implementation) must be prepared to deal with
  // an Env that does not support appending.
  virtual Status NewAppendableFile(const std::string& fname,
                                   WritableFile** result);
  // Returns true iff the named file exists.
  virtual bool FileExists(const std::string& fname) = 0;
  // Store in *result the names of the children of the specified directory.
  // The names are relative to "dir".
  // Original contents of *results are dropped.
  virtual Status GetChildren(const std::string& dir,
                             std::vector<std::string>* result) = 0;
  // Delete the named file.
  //
  // The default implementation calls DeleteFile, to support legacy Env
  // implementations. Updated Env implementations must override RemoveFile and
  // ignore the existence of DeleteFile. Updated code calling into the Env API
  // must call RemoveFile instead of DeleteFile.
  //
  // A future release will remove DeleteDir and the default implementation of
  // RemoveDir.
  virtual Status RemoveFile(const std::string& fname);
  // DEPRECATED: Modern Env implementations should override RemoveFile instead.
  //
  // The default implementation calls RemoveFile, to support legacy Env user
  // code that calls this method on modern Env implementations. Modern Env user
  // code should call RemoveFile.
  //
  // A future release will remove this method.
  virtual Status DeleteFile(const std::string& fname);
  // Create the specified directory.
  virtual Status CreateDir(const std::string& dirname) = 0;
  // Delete the specified directory.
  //
  // The default implementation calls DeleteDir, to support legacy Env
  // implementations. Updated Env implementations must override RemoveDir and
  // ignore the existence of DeleteDir. Modern code calling into the Env API
  // must call RemoveDir instead of DeleteDir.
  //
  // A future release will remove DeleteDir and the default implementation of
  // RemoveDir.
  virtual Status RemoveDir(const std::string& dirname);
  // DEPRECATED: Modern Env implementations should override RemoveDir instead.
  //
  // The default implementation calls RemoveDir, to support legacy Env user
  // code that calls this method on modern Env implementations. Modern Env user
  // code should call RemoveDir.
  //
  // A future release will remove this method.
  virtual Status DeleteDir(const std::string& dirname);
  // Store the size of fname in *file_size.
  virtual Status GetFileSize(const std::string& fname, uint64_t* file_size) = 0;
  // Rename file src to target.
```

```
  virtual Status RenameFile(const std::string& src,
                            const std::string& target) = 0;
  // Lock the specified file.  Used to prevent concurrent access to
  // the same db by multiple processes.  On failure, stores nullptr in
  // *lock and returns non-OK.
  //
  // On success, stores a pointer to the object that represents the
  // acquired lock in *lock and returns OK.  The caller should call
  // UnlockFile(*lock) to release the lock.  If the process exits,
  // the lock will be automatically released.
  //
  // If somebody else already holds the lock, finishes immediately
  // with a failure.  I.e., this call does not wait for existing locks
  // to go away.
  //
  // May create the named file if it does not already exist.
  virtual Status LockFile(const std::string& fname, FileLock** lock) = 0;
  // Release the lock acquired by a previous successful call to LockFile.
  // REQUIRES: lock was returned by a successful LockFile() call
  // REQUIRES: lock has not already been unlocked.
  virtual Status UnlockFile(FileLock* lock) = 0;
  // Arrange to run "(*function)(arg)" once in a background thread.
  //
  // "function" may run in an unspecified thread.  Multiple functions
  // added to the same Env may run concurrently in different threads.
  // I.e., the caller may not assume that background work items are
  // serialized.
  virtual void Schedule(void (*function)(void* arg), void* arg) = 0;
  // Start a new thread, invoking "function(arg)" within the new thread.
  // When "function(arg)" returns, the thread will be destroyed.
  virtual void StartThread(void (*function)(void* arg), void* arg) = 0;
  // *path is set to a temporary directory that can be used for testing. It may
  // or may not have just been created. The directory may or may not differ
  // between runs of the same process, but subsequent calls will return the
  // same directory.
  virtual Status GetTestDirectory(std::string* path) = 0;
  // Create and return a log file for storing informational messages.
  virtual Status NewLogger(const std::string& fname, Logger** result) = 0;
  // Returns the number of micro-seconds since some fixed point in time. Only
  // useful for computing deltas of time.
  virtual uint64_t NowMicros() = 0;
  // Sleep/delay the thread for the prescribed number of micro-seconds.
  virtual void SleepForMicroseconds(int micros) = 0;
};
// A file abstraction for reading sequentially through a file
class LEVELDB_EXPORT SequentialFile {
 public:
  SequentialFile() = default;
  SequentialFile(const SequentialFile&) = delete;
  SequentialFile& operator=(const SequentialFile&) = delete;
  virtual ~SequentialFile();
  // Read up to "n" bytes from the file.  "scratch[0..n-1]" may be
  // written by this routine.  Sets "*result" to the data that was
  // read (including if fewer than "n" bytes were successfully read).
  // May set "*result" to point at data in "scratch[0..n-1]", so
  // "scratch[0..n-1]" must be live when "*result" is used.
```

```
  // If an error was encountered, returns a non-OK status.
  //
  // REQUIRES: External synchronization
  virtual Status Read(size_t n, Slice* result, char* scratch) = 0;
  // Skip "n" bytes from the file. This is guaranteed to be no
  // slower that reading the same data, but may be faster.
  //
  // If end of file is reached, skipping will stop at the end of the
  // file, and Skip will return OK.
  //
  // REQUIRES: External synchronization
  virtual Status Skip(uint64_t n) = 0;
};
```

在env_posix.cc和env_windows.cc中实现了在linux和windows系统下的Env这个抽象类的具体的实现。

```
class PosixEnv : public Env {
 public:
  PosixEnv();
  ~PosixEnv() override {
    static const char msg[] =
        "PosixEnv singleton destroyed. Unsupported behavior!\n";
    std::fwrite(msg, 1, sizeof(msg), stderr);
    std::abort();
  }
  Status NewSequentialFile(const std::string& filename,
                           SequentialFile** result) override {
    int fd = ::open(filename.c_str(), O_RDONLY | kOpenBaseFlags);
    if (fd < 0) {
      *result = nullptr;
      return PosixError(filename, errno);
    }
    *result = new PosixSequentialFile(filename, fd);
    return Status::OK();
  }
  Status NewRandomAccessFile(const std::string& filename,
                             RandomAccessFile** result) override {
    *result = nullptr;
    int fd = ::open(filename.c_str(), O_RDONLY | kOpenBaseFlags);
    if (fd < 0) {
      return PosixError(filename, errno);
    }
    if (!mmap_limiter_.Acquire()) {
      *result = new PosixRandomAccessFile(filename, fd, &fd_limiter_);
```

```
    return Status::OK();
  }
  uint64_t file_size;
  Status status = GetFileSize(filename, &file_size);
  if (status.ok()) {
    void* mmap_base =
        ::mmap(/*addr=*/nullptr, file_size, PROT_READ, MAP_SHARED, fd, 0);
    if (mmap_base != MAP_FAILED) {
      *result = new PosixMmapReadableFile(filename,
                                          reinterpret_cast<char*>(mmap_base),
                                          file_size, &mmap_limiter_);
    } else {
      status = PosixError(filename, errno);
    }
  }
  ::close(fd);
  if (!status.ok()) {
    mmap_limiter_.Release();
  }
  return status;
}
Status NewWritableFile(const std::string& filename,
                       WritableFile** result) override {
  int fd = ::open(filename.c_str(),
                  O_TRUNC | O_WRONLY | O_CREAT | kOpenBaseFlags, 0644);
  if (fd < 0) {
    *result = nullptr;
    return PosixError(filename, errno);
  }
  *result = new PosixWritableFile(filename, fd);
  return Status::OK();
}
Status NewAppendableFile(const std::string& filename,
                         WritableFile** result) override {
  int fd = ::open(filename.c_str(),
                  O_APPEND | O_WRONLY | O_CREAT | kOpenBaseFlags, 0644);
  if (fd < 0) {
    *result = nullptr;
    return PosixError(filename, errno);
  }
  *result = new PosixWritableFile(filename, fd);
  return Status::OK();
}
bool FileExists(const std::string& filename) override {
  return ::access(filename.c_str(), F_OK) == 0;
}
Status GetChildren(const std::string& directory_path,
                   std::vector<std::string>* result) override {
  result->clear();
  ::DIR* dir = ::opendir(directory_path.c_str());
  if (dir == nullptr) {
    return PosixError(directory_path, errno);
  }
  struct ::dirent* entry;
  while ((entry = ::readdir(dir)) != nullptr) {
    result->emplace_back(entry->d_name);
```

```
    }
    ::closedir(dir);
    return Status::OK();
  }
  Status RemoveFile(const std::string& filename) override {
    if (::unlink(filename.c_str()) != 0) {
      return PosixError(filename, errno);
    }
    return Status::OK();
  }
  Status CreateDir(const std::string& dirname) override {
    if (::mkdir(dirname.c_str(), 0755) != 0) {
      return PosixError(dirname, errno);
    }
    return Status::OK();
  }
  Status RemoveDir(const std::string& dirname) override {
    if (::rmdir(dirname.c_str()) != 0) {
      return PosixError(dirname, errno);
    }
    return Status::OK();
  }
  Status GetFileSize(const std::string& filename, uint64_t* size) override {
    struct ::stat file_stat;
    if (::stat(filename.c_str(), &file_stat) != 0) {
      *size = 0;
      return PosixError(filename, errno);
    }
    *size = file_stat.st_size;
    return Status::OK();
  }
  Status RenameFile(const std::string& from, const std::string& to) override {
    if (std::rename(from.c_str(), to.c_str()) != 0) {
      return PosixError(from, errno);
    }
    return Status::OK();
  }
  Status LockFile(const std::string& filename, FileLock** lock) override {
    *lock = nullptr;
    int fd = ::open(filename.c_str(), O_RDWR | O_CREAT | kOpenBaseFlags, 0644);
    if (fd < 0) {
      return PosixError(filename, errno);
    }
    if (!locks_.Insert(filename)) {
      ::close(fd);
      return Status::IOError("lock " + filename, "already held by process");
    }
    if (LockOrUnlock(fd, true) == -1) {
      int lock_errno = errno;
      ::close(fd);
      locks_.Remove(filename);
      return PosixError("lock " + filename, lock_errno);
    }
    *lock = new PosixFileLock(fd, filename);
    return Status::OK();
  }
```

```
  Status UnlockFile(FileLock* lock) override {
    PosixFileLock* posix_file_lock = static_cast<PosixFileLock*>(lock);
    if (LockOrUnlock(posix_file_lock->fd(), false) == -1) {
      return PosixError("unlock " + posix_file_lock->filename(), errno);
    }
    locks_.Remove(posix_file_lock->filename());
    ::close(posix_file_lock->fd());
    delete posix_file_lock;
    return Status::OK();
  }
  void Schedule(void (*background_work_function)(void* background_work_arg),
                void* background_work_arg) override;
  void StartThread(void (*thread_main)(void* thread_main_arg),
                   void* thread_main_arg) override {
    std::thread new_thread(thread_main, thread_main_arg);
    new_thread.detach();
  }
  Status GetTestDirectory(std::string* result) override {
    const char* env = std::getenv("TEST_TMPDIR");
    if (env && env[0] != '\0') {
      *result = env;
    } else {
      char buf[100];
      std::snprintf(buf, sizeof(buf), "/tmp/leveldbtest-%d",
                    static_cast<int>(::geteuid()));
      *result = buf;
    }
    // The CreateDir status is ignored because the directory may already exist.
    CreateDir(*result);
    return Status::OK();
  }
  Status NewLogger(const std::string& filename, Logger** result) override {
    int fd = ::open(filename.c_str(),
                    O_APPEND | O_WRONLY | O_CREAT | kOpenBaseFlags, 0644);
    if (fd < 0) {
      *result = nullptr;
      return PosixError(filename, errno);
    }
    std::FILE* fp = ::fdopen(fd, "w");
    if (fp == nullptr) {
      ::close(fd);
      *result = nullptr;
      return PosixError(filename, errno);
    } else {
      *result = new PosixLogger(fp);
      return Status::OK();
    }
  }
  uint64_t NowMicros() override {
    static constexpr uint64_t kUsecondsPerSecond = 1000000;
    struct ::timeval tv;
    ::gettimeofday(&tv, nullptr);
    return static_cast<uint64_t>(tv.tv_sec) * kUsecondsPerSecond + tv.tv_usec;
  }
  void SleepForMicroseconds(int micros) override {
    std::this_thread::sleep_for(std::chrono::microseconds(micros));
```

```
  }
 private:
  void BackgroundThreadMain();
  static void BackgroundThreadEntryPoint(PosixEnv* env) {
    env->BackgroundThreadMain();
  }

  // Stores the work item data in a Schedule() call.
  //
  // Instances are constructed on the thread calling Schedule() and used on the
  // background thread.
  //
  // This structure is thread-safe beca
  struct BackgroundWorkItem {
    explicit BackgroundWorkItem(void (*function)(void* arg), void* arg)
        : function(function), arg(arg) {}

    void (*const function)(void*);
    void* const arg;
  };

  port::Mutex background_work_mutex_;
  port::CondVar background_work_cv_ GUARDED_BY(background_work_mutex_);
  bool started_background_thread_ GUARDED_BY(background_work_mutex_);

  std::queue<BackgroundWorkItem> background_work_queue_
      GUARDED_BY(background_work_mutex_);

  PosixLockTable locks_;  // Thread-safe.
  Limiter mmap_limiter_;  // Thread-safe.
  Limiter fd_limiter_;    // Thread-safe.
};
```

PosixEnv类中Schedule函数的实现如下所示：

```
void PosixEnv::Schedule(
    void (*background_work_function)(void* background_work_arg),
    void* background_work_arg) {
  background_work_mutex_.Lock();

  // Start the background thread, if we haven't done so already.
  if (!started_background_thread_) {
    started_background_thread_ = true;
    std::thread background_thread(PosixEnv::BackgroundThreadEntryPoint, this);
    background_thread.detach();
  }

  // If the queue is empty, the background thread may be waiting for work.
  if (background_work_queue_.empty()) {
    background_work_cv_.Signal();
  }

  background_work_queue_.emplace(background_work_function, background_work_arg);
  background_work_mutex_.Unlock();
}
```

在Schedule函数中，第一个if语句：如果未启动后台线程，则启动他，启动过程实际上在BackgroundThreadMain中，将background_work_queue_队首的任务提取出来然后执行。

第二个if语句是如果任务队列是空的，则说明使用条件变量唤醒后台工作，因为此时就要将任务压入

```
  void BackgroundThreadMain();
  static void BackgroundThreadEntryPoint(PosixEnv* env) {
    env->BackgroundThreadMain();
  }
```

再来看看调用的BackgroundThreadMain()函数的定义：

```
void PosixEnv::BackgroundThreadMain() {
  while (true) {
    background_work_mutex_.Lock();

    // Wait until there is work to be done.
    while (background_work_queue_.empty()) {
      background_work_cv_.Wait();
    }

    assert(!background_work_queue_.empty());
    auto background_work_function = background_work_queue_.front().function;
    void* background_work_arg = background_work_queue_.front().arg;
```

```
    background_work_queue_.pop();
    background_work_mutex_.Unlock();
    background_work_function(background_work_arg);
  }
}
```

发现启动的这个后台线程，由这么一个函数控制，如果任务队列是空的则Wait，否则循环（while）的过程不断的从background_work_queue_队列中取出元素，然后执行。

另外注意这个BackgroundThreadMain()是个死循环，只要队列中有任务（有元素）则不断的取出，然后执行。在这里**实际上最后一句执行的就是**BGWork(void* db)**这个过程。**进一步调用了**BackgroundCall（）**这个过程。

```
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();
}
```

那么在上面这个过程中它会调用**BackgroundCompaction（）**函数，关键点也就在这个地方了：

DBImpl:: BackgroundCompaction 函数是执行后台压缩任务的核心函数。

定义如下所示：

事实上这个函数会处理两种压缩情景：

- **MemTable 压缩**：如果当前存在不可变的 MemTable（imm_），它会优先将其压缩并写入到磁盘中。
- **文件压缩**：在没有 MemTable 需要压缩的情况下，它会执行对 SST 文件的压缩，可能是自动的也可能是手动触发的。

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

  //(1)Memtable压缩
  //如果是不可变的Memtable，调用CompactMemtable（）进行压缩
  if (imm_ != nullptr) {
    CompactMemTable();
```

```
  return;
}
```

//(2)文件压缩

```
Compaction* c;
bool is_manual = (manual_compaction_ != nullptr);
InternalKey manual_end;
//手动压缩
if (is_manual) {
  ManualCompaction* m = manual_compaction_;
  c = versions_->CompactRange(m->level, m->begin, m->end);
  m->done = (c == nullptr);
  if (c != nullptr) {
    manual_end = c->input(0, c->num_input_files(0) - 1)->largest;
  }
  Log(options_.info_log,
      "Manual compaction at level-%d from %s .. %s; will stop at %s\n",
      m->level, (m->begin ? m->begin->DebugString().c_str() : "(begin)"),
      (m->end ? m->end->DebugString().c_str() : "(end)"),
      (m->done ? "(end)" : manual_end.DebugString().c_str()));
} else {
//自动压缩
  c = versions_->PickCompaction();
}


Status status;
if (c == nullptr) {
  // Nothing to do
} else if (!is_manual && c->IsTrivialMove()) {
  // Move file to next level
  assert(c->num_input_files(0) == 1);
  FileMetaData* f = c->input(0, 0);
  c->edit()->RemoveFile(c->level(), f->number);
  c->edit()->AddFile(c->level() + 1, f->number, f->file_size, f->smallest,
                     f->largest);
  status = versions_->LogAndApply(c->edit(), &mutex_);
  if (!status.ok()) {
    RecordBackgroundError(status);
  }
  VersionSet::LevelSummaryStorage tmp;
  Log(options_.info_log, "Moved #%lld to level-%d %lld bytes %s: %s\n",
      static_cast<unsigned long long>(f->number), c->level() + 1,
      static_cast<unsigned long long>(f->file_size),
      status.ToString().c_str(), versions_->LevelSummary(&tmp));
} else {
  CompactionState* compact = new CompactionState(c);
  status = DoCompactionWork(compact);
  if (!status.ok()) {
    RecordBackgroundError(status);
  }
  CleanupCompaction(compact);
  c->ReleaseInputs();
  RemoveObsoleteFiles();
}
```

```
  delete c;

  //压缩结果处理，一般是错误处理
  if (status.ok()) {
    // Done
  } else if (shutting_down_.load(std::memory_order_acquire)) {
    // Ignore compaction errors found during shutting down
  } else {
    Log(options_.info_log, "Compaction error: %s", status.ToString().c_str());
  }

  //手动压缩结束处理
  if (is_manual) {
    ManualCompaction* m = manual_compaction_;
    if (!status.ok()) {
      m->done = true;
    }
    if (!m->done) {
      // We only compacted part of the requested range.  Update *m
      // to the range that is left to be compacted.
      m->tmp_storage = manual_end;
      m->begin = &m->tmp_storage;
    }
    manual_compaction_ = nullptr;
  }
}
```

LevelDB笔记10：Block

2024年8月22日 17:00

leveldb里面sstable文件里，有多个数据block，其中data block、index block、Meta index block都是由类BlockBuilder负责生成的。

来看看这个类的实现：

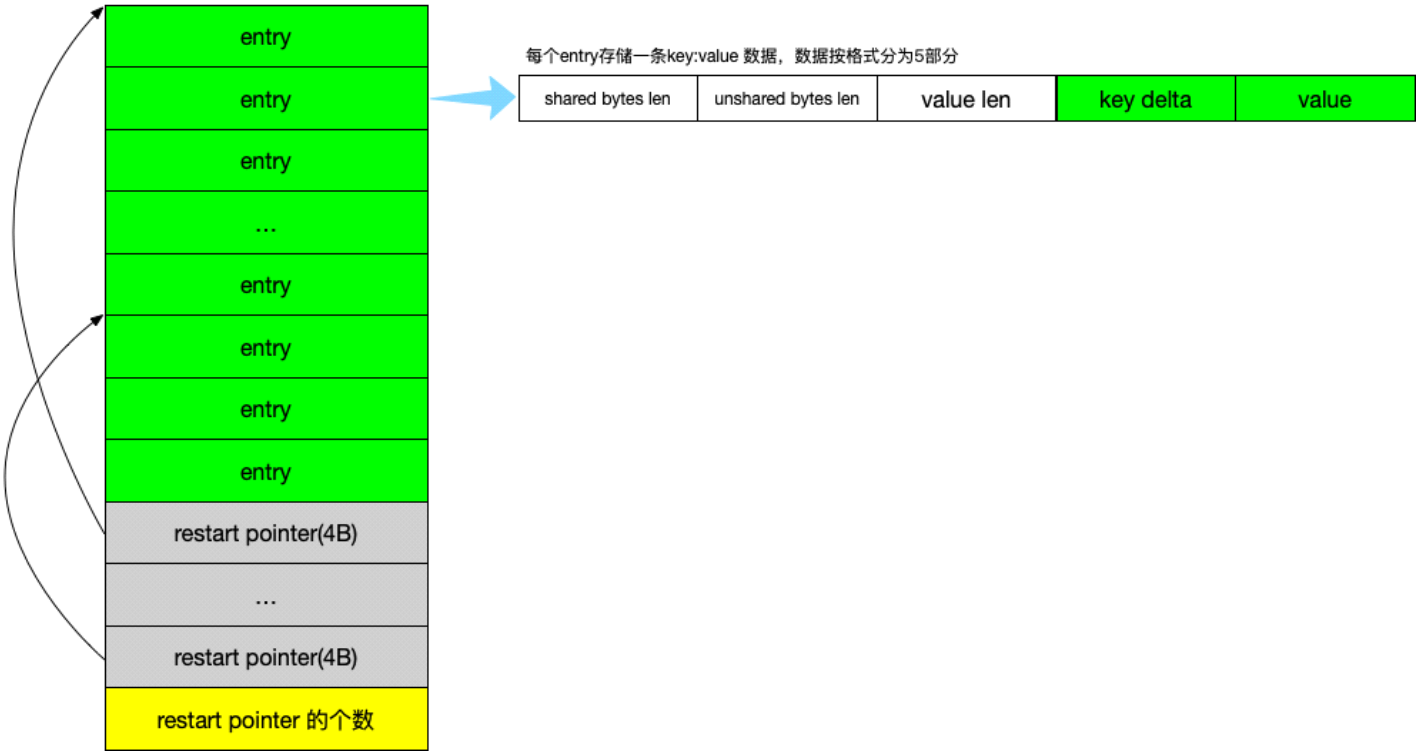
```
class BlockBuilder {
 public:
  explicit BlockBuilder(const Options* options);
  BlockBuilder(const BlockBuilder&) = delete;
  BlockBuilder& operator=(const BlockBuilder&) = delete;
 // Reset the contents as if the BlockBuilder was just constructed.
  void Reset();
 // REQUIRES: Finish() has not 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_;
};
```

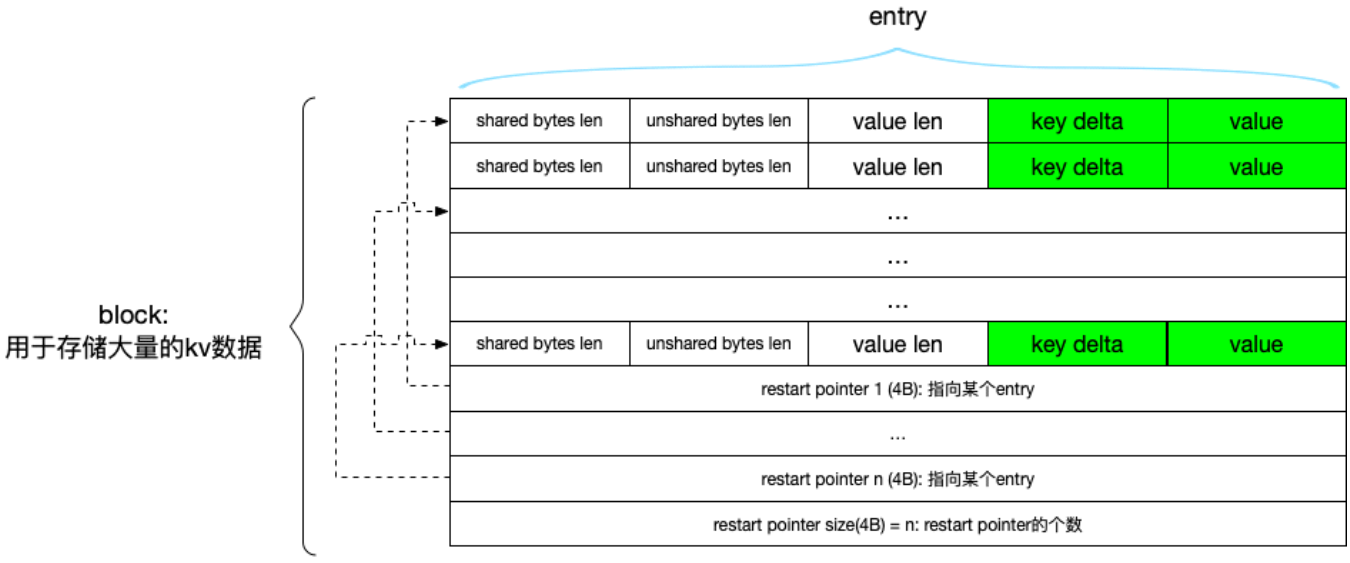
构造函数：

```
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
}
```

一般而言存储键值要存储其长度和键值对本身，但有一个很重要的特性就是key有序，levelDB的

作者发现实际使用时，排序后的大量key之间经常有相同的前缀，因此一个entry的数据格式如下所示：





先来了解一下基本概念：

- **前缀压缩（Prefix Compression）**：在数据存储中，如果连续的键之间有相同的前缀，那么可以**只存储不同的部分以节省空间。**这称为前缀压缩。
- **重启点（Restart Points）**：为了防止过度压缩造成解压困难，每隔一定数量的键（block_restart_interval），会记录一个完整的键，从这个键开始，重新进行前缀压缩。重启点的索引存储在 restarts_ 向量中。

听BlockBuilder这个类的名字，就是新建一个块的操作，但可以看到整个类的成员变量如下：

```
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_;
```

从图中可以看到restarts是个vector对象，存储着uint32_t类型的数据，负责指向某些entry个体。

数据怎么存呢？
这个只有一个buffer_，听名字就是缓冲区的意思。

那么来看Add（）函数：

```
void BlockBuilder::Add(const Slice& key, const Slice& value) {
  Slice last_key_piece(last_key_);
```

//首先是断言：检查finished_是否为false，确保当前块尚未完成写入；检查当前写入的键的数量不超过block_restart_interval；最后断言当前块要么是空的，要么当前键比上一个键更大（保证键按顺序存储）。

```
  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;
```

//如果当前键数目小于block_restart_interval，则尝试计算当前键和前一个键的共享前缀长度（shared），这一步是前缀压缩的核心部分。

//否则达到restart间隔，则将buffer_的大小存储在restarts_中，并重置计数器counter_,从而重启压缩。这意味着从下一个键开始，将重新存储完整的键而不仅存储差异部分。

```
  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;
  }
```

//将共享部分长度、非共享部分长度和值的大小作为变长整数（Varint32）存储在buffer_中

```
  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());
```

//进一步将key和value追加到buffer_中

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

  //更新last_key_使其等于当前key，以便在下次Add调用时进行比较和压缩
  // Update state
  last_key_.resize(shared);     //保留相同的前缀部分，shared部分
  last_key_.append(key.data() + shared, non_shared); //然后将key与last_key的不同部分追加到last_key_中
  assert(Slice(last_key_) == key);
  counter_++;
}
```

此外在levelDB中block_restart_interval=16
我们可以很清楚的看到，在Add函数中，只是将每一个entry加入到buffer_中了，并没有做到存储到磁盘的功能，也没有像在上面那张图那样在block中包含restarts_相关数据。
事实上，这个过程由Finish函数完成：

```
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_);
}
```

restarts_数组元素大小是4B，并没有使用varint，这样在读取的时候可以更方便一些。

```
size_t BlockBuilder::CurrentSizeEstimate() const {
  return (buffer_.size() +                      // Raw data buffer
          restarts_.size() * sizeof(uint32_t) +  // Restart array
          sizeof(uint32_t));                     // Restart array length
}
```

LevelDB笔记11：Block读取

2024年8月23日 10:16

BlockBuilder写入后的数据，是通过class Block解析完成的。

先来看Block这个类：

```
class Block {
 public:
  // Initialize the block with the specified contents.
  explicit Block(const BlockContents& contents);
  Block(const Block&) = delete;
  Block& operator=(const Block&) = delete;
  ~Block();
  size_t size() const { return size_; }
  Iterator* NewIterator(const Comparator* comparator);
 private:
  class Iter;
  uint32_t NumRestarts() const;
  const char* data_;
  size_t size_;
  uint32_t restart_offset_;  // Offset in data_ of restart array
  bool owned_;               // Block owns data_[]
};
```

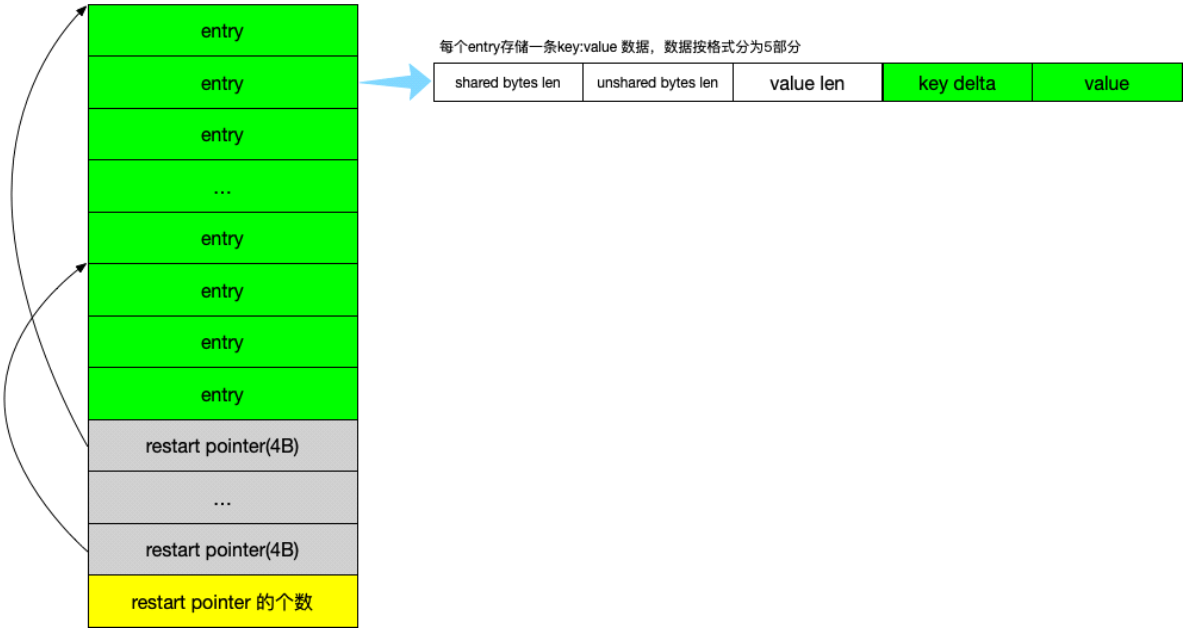
来看它的构造函数：

```
Block::Block(const BlockContents& contents)
    : data_(contents.data.data()),
      size_(contents.data.size()),
      owned_(contents.heap_allocated) {
  if (size_ < sizeof(uint32_t)) {
    size_ = 0;  // Error marker
  } else {
    size_t max_restarts_allowed = (size_ - sizeof(uint32_t)) / sizeof(uint32_t);
    if (NumRestarts() > max_restarts_allowed) {
      // The size is too small for NumRestarts()
      size_ = 0;
    } else {
      restart_offset_ = size_ - (1 + NumRestarts()) * sizeof(uint32_t);
    }
  }
}
```

参数是const BlockContents&类型的，这是一个结构体：

```
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()
```

```
};
```



先不管别的，已知BlockBuilder的中需要由restart point，每隔Option.block_restart_interval个entry（对应一对键值对），保存地址至restarts数组中。而且在构建好的Block中最后4B记录了restarts数组的元素个数N。再往前4*N bytes记录了restart的首地址，由此很容易对齐进行读取。

当查找key的时候，最朴素的做法就是逐个读取，直到遇到符合条件的key。

但借助block中key有序的这个特性，通过比对restart point所指向的key，可以利用二分法的方法快速定位并查找相应key。

block的读取实际上是交给leveldb:: Block:: Iter完成的，通过NewIterator构造，使用后需要手动delete。来看看Iter和NewIterator的方法：

```
class Block::Iter : public Iterator {
 private:
  const Comparator* const comparator_;
  //指向底层块数据的指针，这些数据是一个连续的字节数组，包含entry和restarts信息。
  const char* const data_;       // underlying block contents
  //快速定位到restarts数组的首地址的位置，即表示restarts数组在块中的偏移地址。
  uint32_t const restarts_;      // Offset of restart array (list of fixed32)
```

```
  uint32_t const num_restarts_;  // Number of uint32_t entries in restart array
  // current_ is offset in data_ of current entry.  >= restarts_ if !Valid
  uint32_t current_;     //当前条目entry在data_中的偏移量，如果迭代器无效该值大于等于restarts_
  uint32_t restart_index_;  // Index of restart block in which current_ falls当前条目entry所属的restarts点索引，用于在遍历过程中快速定位
  std::string key_;     //当前条目的键
  Slice value_;         //当前条目的值
  Status status_;       //迭代器的状态
  inline int Compare(const Slice& a, const Slice& b) const {
    return comparator_->Compare(a, b);
  }
  // Return the offset in data_ just past the end of the current entry.
  inline uint32_t NextEntryOffset() const {
    return (value_.data() + value_.size()) - data_;
  }
  uint32_t GetRestartPoint(uint32_t index) {
    assert(index < num_restarts_);
    return DecodeFixed32(data_ + restarts_ + index * sizeof(uint32_t));
  }
  void SeekToRestartPoint(uint32_t index) {
    key_.clear();
    restart_index_ = index;
    // current_ will be fixed by ParseNextKey();
    // ParseNextKey() starts at the end of value_, so set value_ accordingly
    uint32_t offset = GetRestartPoint(index);
    value_ = Slice(data_ + offset, 0);
  }
//上面都是私有的成员变量和成员函数
//下面是公有的
 public:
  //(1)构造函数
  Iter(const Comparator* comparator, const char* data, uint32_t restarts,
       uint32_t num_restarts)
      : comparator_(comparator),
        data_(data),
        restarts_(restarts),
        num_restarts_(num_restarts),
        current_(restarts_),
        restart_index_(num_restarts_) {
    assert(num_restarts_ > 0);
  }
  //迭代器是否有效，current_小于restarts_时是有效的
  bool Valid() const override { return current_ < restarts_; }
  Status status() const override { return status_; }
  Slice key() const override {
    assert(Valid());
    return key_;
  }
  Slice value() const override {
    assert(Valid());
    return value_;
  }
```

```
  //迭代器跳到下一条entry
  void Next() override {
    assert(Valid());
    ParseNextKey();
  }
  //迭代器跳到上一条entry
  void Prev() override {
    assert(Valid());    //验证迭代器是否有效
    // Scan backwards to a restart point before current_
    const uint32_t original = current_;

    //while循环用以判断restar_index_是不是代表的是original之前的一个重启点。
    while (GetRestartPoint(restart_index_) >= original) {
      if (restart_index_ == 0) {
        // No more entries
        current_ = restarts_;
        restart_index_ = num_restarts_;
        return;
      }
      restart_index_--;
    }
    //将迭代器定位到该重启点的起始位置，确保从该点开始，顺序的读取条目知道找到目标条目
    SeekToRestartPoint(restart_index_);
    do {
      // Loop until end of current entry hits the start of original entry
    } while (ParseNextKey() && NextEntryOffset() < original);
  }

  // seek函数是迭代器的核心方法
  void Seek(const Slice& target) override {
    // Binary search in restart array to find the last restart point
    // with a key < target
    uint32_t left = 0;
    uint32_t right = num_restarts_ - 1;
    int current_key_compare = 0;        //用以存储当前键和目标键之间的比较结果
    if (Valid()) {
      //如果当前迭代器状态有效，即当前指向的键有效，首先比较当前键和目标键
      // If we're already scanning, use the current position as a starting
      // point. This is beneficial if the key we're seeking to is ahead of the
      // current position.
      current_key_compare = Compare(key_, target);
      //如果当前键小于target key，更新left为restarts_index_
      if (current_key_compare < 0) {
        // key_ is smaller than target
        left = restart_index_;
      } else if (current_key_compare > 0) {
        right = restart_index_;
      } else {
        // We're seeking to the key we're already at.当前迭代器指向的key就是要seek的
target，直接返回即可
        return;
      }
```

```
    }

    //在restarts_中进行二分查找，找到最后一个小于target key的重启点
    while (left < right) {
      uint32_t mid = (left + right + 1) / 2;
      uint32_t region_offset = GetRestartPoint(mid);
      uint32_t shared, non_shared, value_length;
      const char* key_ptr =
          DecodeEntry(data_ + region_offset, data_ + restarts_, &shared,
                      &non_shared, &value_length);
      if (key_ptr == nullptr || (shared != 0)) {
        CorruptionError();
        return;
      }
      Slice mid_key(key_ptr, non_shared);
      if (Compare(mid_key, target) < 0) {
        // Key at "mid" is smaller than "target".  Therefore all
        // blocks before "mid" are uninteresting.
        left = mid;
      } else {
        // Key at "mid" is >= "target".  Therefore all blocks at or
        // after "mid" are uninteresting.
        right = mid - 1;
      }
    }
    // We might be able to use our current position within the restart block.
    // This is true if we determined the key we desire is in the current block
    // and is after than the current key.
    //如果二分查找到的left与当前迭代器中的restarts_index_相等，并且当前key比target key小说明此时无需重新定位restarts_index_,否则需要调用SeekToRestartsPoint(left)函数，定位到left位置的迭代器的指向位置
    assert(current_key_compare == 0 || Valid());
    bool skip_seek = left == restart_index_ && current_key_compare < 0;
    if (!skip_seek) {
      SeekToRestartPoint(left);
    }
    //从找到的重启点开始，进行线性扫描，每次调用ParseNextKey（）获取下一个键，一旦找到一个键大于或等于目标键的entry，函数即返回
    // Linear search (within restart block) for first key >= target
    while (true) {
      if (!ParseNextKey()) {
        return;
      }
      if (Compare(key_, target) >= 0) {
        return;
      }
    }
  }

  void SeekToFirst() override {
    SeekToRestartPoint(0);
    ParseNextKey();
  }
  void SeekToLast() override {
```

```
      SeekToRestartPoint(num_restarts_ - 1);
      while (ParseNextKey() && NextEntryOffset() < restarts_) {
        // Keep skipping
      }
    }


  private:
    void CorruptionError() {
      current_ = restarts_;
      restart_index_ = num_restarts_;
      status_ = Status::Corruption("bad entry in block");
      key_.clear();
      value_.clear();
    }
    bool ParseNextKey() {
      current_ = NextEntryOffset();
      const char* p = data_ + current_;
      const char* limit = data_ + restarts_;  // Restarts come right after data
      if (p >= limit) {
        // No more entries to return.  Mark as invalid.
        current_ = restarts_;
        restart_index_ = num_restarts_;
        return false;
      }
      // Decode next entry
      uint32_t shared, non_shared, value_length;
      p = DecodeEntry(p, limit, &shared, &non_shared, &value_length);
      if (p == nullptr || key_.size() < shared) {
        CorruptionError();
        return false;
      } else {
        //首先更新key_和value_
        key_.resize(shared);
        key_.append(p, non_shared);
        value_ = Slice(p + non_shared, value_length);
        //然后更新restart_index，指向当前条目所属的那个共享前缀的第一个条目entry
        while (restart_index_ + 1 < num_restarts_ &&
               GetRestartPoint(restart_index_ + 1) < current_) {
          ++restart_index_;
        }
        return true;
      }
    }
```

可以发现在Iter类中的很多成员方法中，都会调用类中私有的成员方法：ParseNextKey()

```
bool ParseNextKey() {
    current_ = NextEntryOffset();
    const char* p = data_ + current_;       //p就指向下一条条目entry
    const char* limit = data_ + restarts_;  // Restarts come right after data一个限制，防止超出entry条目
    if (p >= limit) {
      //没有下一个entry了，返回false
      // No more entries to return.  Mark as invalid.
      current_ = restarts_;
      restart_index_ = num_restarts_;
      return false;
    }
    // Decode next entry
```

```
Iterator* Block::NewIterator(const Comparator* comparator) {
  if (size_ < sizeof(uint32_t)) {
    return NewErrorIterator(Status::Corruption("bad block contents"));
  }
  const uint32_t num_restarts = NumRestarts();
  if (num_restarts == 0) {
    return NewEmptyIterator();
  } else {
    return new Iter(comparator, data_, restart_offset_, num_restarts);
  }
}
```

```
    return new Iter(comparator, data_, restart_offset_, num_restarts);
  }
}
// Decode next entry
```

//解析p指向的那个entry条目，调用DecodeEntry（）过程

```
uint32_t shared, non_shared, value_length;
p = DecodeEntry(p, limit, &shared, &non_shared, &value_length); //最后返回的这个p指向的是p指向的那个entry条目的key dealt的起始位置
if (p == nullptr || key_.size() < shared) {
```

//前一个entry和后一个entry共享前缀，前一个entry的key的size也就必须满足要大于等于后一个shared的值，不满足则说明出错了

```
  CorruptionError();
  return false;
} else {
```

//更新成员变量key_和value

```
  key_.resize(shared);
  key_.append(p, non_shared);
  value_ = Slice(p + non_shared, value_length);
```

//重新更新restart_index_的指针

```
  while (restart_index_ + 1 < num_restarts_ &&
         GetRestartPoint(restart_index_ + 1) < current_) {
    ++restart_index_;
  }
  return true;
}
}
```

我们从这个函数先入手看看它的作用是什么：

```
// Return the offset in data_ just past the end of the current entry.
```

这个函数就是返回当前条目entry的下一条entry相对于该block起始位置的偏移量。

```
inline uint32_t NextEntryOffset() const {
  return (value_.data() + value_.size()) - data_;
}
```

如何解析entry的?

```
static inline const char* DecodeEntry(const char* p, const char* limit,
                                      uint32_t* shared, uint32_t* non_shared,
                                      uint32_t* value_length) {
```

//p和limit之间至少有三个bytes，用来存储shared、non_shared和value_length三个字段。

```
if (limit - p < 3) return nullptr;
```

//解析出p指向的那个entry的前三个bytes，分别赋值给shared、non_shared和value_length。

```
*shared = reinterpret_cast<const uint8_t*>(p)[0];
*non_shared = reinterpret_cast<const uint8_t*>(p)[1];
*value_length = reinterpret_cast<const uint8_t*>(p)[2];
//这里？？？
//
if ((*shared | *non_shared | *value_length) < 128) {
```

//块速路径

//变长编码，如果最高位为1说明编码不完整，也就是说当该字节小于128的时候，最高位肯定不为1，这就说明，shared、non_shared和value_length均可以用一个字节表示，直接对p+3即可定位到key delta的位置

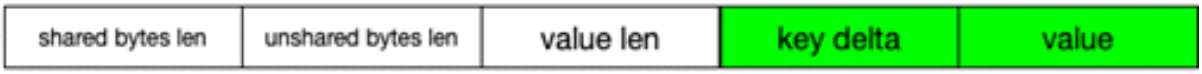
```
  // Fast path: all three values are encoded in one byte each
  p += 3;
```

```
  } else {
    //慢速路径
    if ((p = GetVarint32Ptr(p, limit, shared)) == nullptr) return nullptr;
    if ((p = GetVarint32Ptr(p, limit, non_shared)) == nullptr) return nullptr;
    if ((p = GetVarint32Ptr(p, limit, value_length)) == nullptr) return nullptr;
  }
```

//检查是否越界，该段代码检查从当前指针p到limit的剩余字节数，确保它足够大以容纳非共享键的字节数和值的字节数。如果不足，说明数据有误或被损坏，返回nullptr

```
  if (static_cast<uint32_t>(limit - p) < (*non_shared + *value_length)) {
    return nullptr;
  }
  return p; //最后返回的时指向key delta的指针
}
```

每个entry存储一条key:value 数据，数据按格式分为5部分



2024年8月24日 11:27

```
class FilterBlockBuilder

class FilterBlockReader
```

```
// A FilterBlockBuilder is used to construct all of the filters for a
// particular Table.  It generates a single string which is stored as
// a special block in the Table.
//
// The sequence of calls to FilterBlockBuilder must match the regexp:
//      (StartBlock AddKey*)* Finish
class FilterBlockBuilder {
 public:
  explicit FilterBlockBuilder(const FilterPolicy*);
  FilterBlockBuilder(const FilterBlockBuilder&) = delete;
  FilterBlockBuilder& operator=(const FilterBlockBuilder&) = delete;
  void StartBlock(uint64_t block_offset);
  void AddKey(const Slice& key);
  Slice Finish();
 private:
  void GenerateFilter();
  const FilterPolicy* policy_;
  std::string keys_;             // Flattened key contents
  std::vector<size_t> start_;    // Starting index in keys_ of each key
  std::string result_;           // Filter data computed so far
  std::vector<Slice> tmp_keys_;  // policy_->CreateFilter() argument
  std::vector<uint32_t> filter_offsets_;   //存储每个过滤器起始位置的偏移量，用
于定位不同的过滤器
};
```

```
FilterBlockBuilder::FilterBlockBuilder(const FilterPolicy* policy)
    : policy_(policy) {}

  const FilterPolicy* policy_;
```

FilterPolicy

```
class LEVELDB_EXPORT FilterPolicy {
 public:
```

```
  virtual ~FilterPolicy();
  // Return the name of this policy.  Note that if the filter encoding
  // changes in an incompatible way, the name returned by this method
  // must be changed.  Otherwise, old incompatible filters may be
  // passed to methods of this type.
  virtual const char* Name() const = 0;
  // keys[0,n-1] contains a list of keys (potentially with duplicates)
  // that are ordered according to the user supplied comparator.
  // Append a filter that summarizes keys[0,n-1] to *dst.
  //
  // Warning: do not change the initial contents of *dst.  Instead,
  // append the newly constructed filter to *dst.
  virtual void CreateFilter(const Slice* keys, int n,
                            std::string* dst) const = 0;
  // "filter" contains the data appended by a preceding call to
  // CreateFilter() on this class.  This method must return true if
  // the key was in the list of keys passed to CreateFilter().
  // This method may return true or false if the key was not on the
  // list, but it should aim to return false with a high probability.
  virtual bool KeyMayMatch(const Slice& key, const Slice& filter) const = 0;
};



struct leveldb_filterpolicy_t : public FilterPolicy

class InternalFilterPolicy : public FilterPolicy

class BloomFilterPolicy : public FilterPolicy



class BloomFilterPolicy : public FilterPolicy {
 public:
  explicit BloomFilterPolicy(int bits_per_key) : bits_per_key_(bits_per_key) {
    // We intentionally round down to reduce probing cost a little bit
    k_ = static_cast<size_t>(bits_per_key * 0.69);  // 0.69 =~ ln(2)
    if (k_ < 1) k_ = 1;
    if (k_ > 30) k_ = 30;
  }
  const char* Name() const override { return "leveldb.BuiltinBloomFilter2"; }
  void CreateFilter(const Slice* keys, int n, std::string* dst) const override
{
    // Compute bloom filter size (in both bits and bytes)
    //计算布隆过滤器所需的总位数，这里bits_per_key是每个键分配的比特数，通过构造函数传递和设置。通常来说更多的位数会降低假阳性率，但会使用更多的内存。
    size_t bits = n * bits_per_key_;
    // For small n, we can see a very high false positive rate.  Fix it
    // by enforcing a minimum bloom filter length.
    //为了防止在键数很小时出现较高的假阳性率，布隆过滤器的最小长度被设置为64位，这样做是为了在小规模数据集情况下，也能维持较低的假阳性率。
    if (bits < 64) bits = 64;
    //计算布隆过滤器所需的字节数，并将bits调整为字节对齐（bits是8的倍数）。这种计算方式确保了即使是部分字节位数也被计算在内。
    size_t bytes = (bits + 7) / 8;
    bits = bytes * 8;
    const size_t init_size = dst->size();   //存储当前目标字符串的大小
```

```
    dst->resize(init_size + bytes, 0);        //调整目标字符串dst的大小
    dst->push_back(static_cast<char>(k_));  // Remember # of probes in filter
```
将哈希函数的数量k_作为一个字符添加到字符串末尾。
```
    char* array = &(*dst)[init_size];    //获取指向新创建的布隆过滤器在目标字符
```
串中位置的指针。这个指针用于直接操作字符串的字节数组。

```
    //有n个键，对每一个键而言他自然对应k_个无偏函数（哈希），所以这里是双重
```
for循环进行处理。
```
    for (int i = 0; i < n; i++) {
      // Use double-hashing to generate a sequence of hash values.
      // See analysis in [Kirsch,Mitzenmacher 2006].
      uint32_t h = BloomHash(keys[i]);     //返回第i个键的哈希值h
      const uint32_t delta = (h >> 17) | (h << 15);  // Rotate right 17 bits
      for (size_t j = 0; j < k_; j++) {
        const uint32_t bitpos = h % bits;
        array[bitpos / 8] |= (1 << (bitpos % 8));
        h += delta;
      }
    }
  }

  bool KeyMayMatch(const Slice& key, const Slice& bloom_filter) const override
{
    const size_t len = bloom_filter.size();
    if (len < 2) return false;
    const char* array = bloom_filter.data();
    const size_t bits = (len - 1) * 8;
    // Use the encoded k so that we can read filters generated by
    // bloom filters created using different parameters.
    const size_t k = array[len - 1];
    if (k > 30) {
      // Reserved for potentially new encodings for short bloom filters.
      // Consider it a match.
      return true;
    }
    uint32_t h = BloomHash(key);
    const uint32_t delta = (h >> 17) | (h << 15);  // Rotate right 17 bits
    for (size_t j = 0; j < k; j++) {
      const uint32_t bitpos = h % bits;
      if ((array[bitpos / 8] & (1 << (bitpos % 8))) == 0) return false;
      h += delta;
    }
    return true;
  }
 private:
  size_t bits_per_key_;      //每个键的比特数，用于决定布隆过滤器的密度
  size_t k_;                 //哈希函数的数量，用于布隆过滤器中的探测。
};



static uint32_t BloomHash(const Slice& key) {
  return Hash(key.data(), key.size(), 0xbc9f1d34);
}

uint32_t Hash(const char* data, size_t n, uint32_t seed) {
  // Similar to murmur hash
  const uint32_t m = 0xc6a4a793;
```

```
  const uint32_t r = 24;
  const char* limit = data + n;
  uint32_t h = seed ^ (n * m);    //哈希值，初始化为seed和n*m的异或结果。
  // Pick up four bytes at a time
  //处理每4字节数据
  while (data + 4 <= limit) {
    uint32_t w = DecodeFixed32(data);  //读取四个字节并将它们组合成为一个32位的无符号整数。
     //之后对哈希值h进行更新
    data += 4;
    h += w;    //加上读取的4字节整数
    h *= m;    //乘以常数m
    h ^= (h >> 16);    //对结果进行右移16位，然后与自身进行异或操作（进一步增加混合效果）。
  }
  // Pick up remaining bytes
  //处理剩余字节（不满4B的部分）
  //使用位移操作将剩余的字节累加到哈希值h中
  //再次使用乘法和位移操作来增加混合效果
  switch (limit - data) {
    case 3:
      h += static_cast<uint8_t>(data[2]) << 16;
      FALLTHROUGH_INTENDED;
    case 2:
      h += static_cast<uint8_t>(data[1]) << 8;
      FALLTHROUGH_INTENDED;
    case 1:
      h += static_cast<uint8_t>(data[0]);
      h *= m;
      h ^= (h >> r);
      break;
  }
  return h;  //返回最终计算的32位的哈希值
} void StartBlock(uint64_t block_offset);
  void AddKey(const Slice& key);
  Slice Finish();

  const FilterPolicy* policy_;
  std::string keys_;              // Flattened key contents
  std::vector<size_t> start_;     // Starting index in keys_ of each key
  std::string result_;            // Filter data computed so far
  std::vector<Slice> tmp_keys_;  // policy_->CreateFilter() argument
  std::vector<uint32_t> filter_offsets_;
```


```
void FilterBlockBuilder::AddKey(const Slice& key) {
  Slice k = key;
  start_.push_back(keys_.size());
  keys_.append(k.data(), k.size());
}
```

```
void FilterBlockBuilder::StartBlock(uint64_t block_offset) {
  uint64_t filter_index = (block_offset / kFilterBase);
  assert(filter_index >= filter_offsets_.size());
  while (filter_index > filter_offsets_.size()) {
    GenerateFilter();
  }
}

// Generate new filter every 2KB of data
static const size_t kFilterBaseLg = 11;
static const size_t kFilterBase = 1 << kFilterBaseLg;   // 1左移11位，2048=2KB，也就是说一个filter data大小是2KB的
```

```
void FilterBlockBuilder::GenerateFilter() {
  const size_t num_keys = start_.size();   //start_size()也就是key的数量
  if (num_keys == 0) {
    // Fast path if there are no keys for this filter
    filter_offsets_.push_back(result_.size());
    return;
  }

  //构建键的列表，keys_中的所有键以slice对象的形式存储在tmp_keys_中。
  // Make list of keys from flattened key structure
  start_.push_back(keys_.size());  // Simplify length computation
  tmp_keys_.resize(num_keys);
  for (size_t i = 0; i < num_keys; i++) {
    const char* base = keys_.data() + start_[i];
```

```
    size_t length = start_[i + 1] - start_[i];
    tmp_keys_[i] = Slice(base, length);
  }

  // Generate filter for current set of keys and append to result_.
  //生成布隆过滤器
  filter_offsets_.push_back(result_.size());   //将当前的result_的大小（偏移量）添加到filter_offsets_中
  policy_->CreateFilter(&tmp_keys_[0], static_cast<int>(num_keys), &result_);//调用CreateFilter方法，以生成新的布隆过滤器并将其追加到result_中。

  //然后清理临时数据
  tmp_keys_.clear();
  keys_.clear();
  start_.clear();
}
```

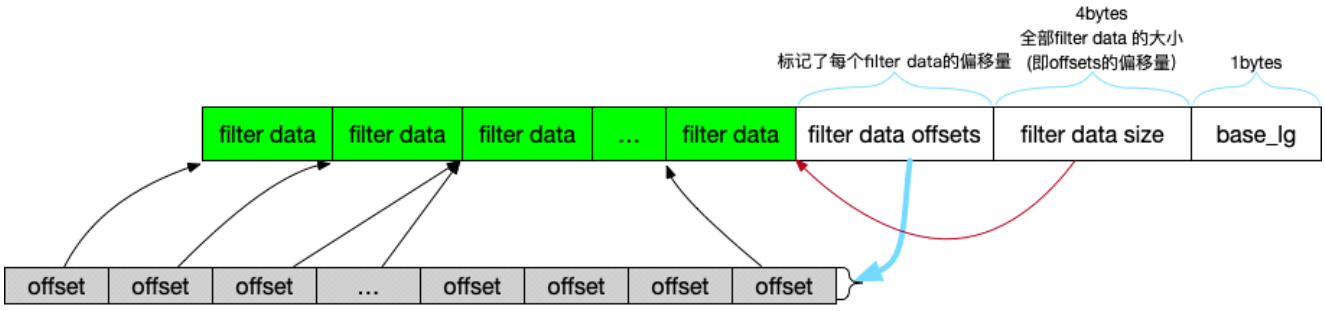
整体来看，FilterBlockBuilder::GenerateFilter()的作用通常是为当前的一组键生成过滤器，以便在读取时快速检测某个键是否存在。这是通过调用之前定义的CreateFilter方法实现的。
通过result_进行返回，它用来存储所有生成的过滤器结果的string。
总结：

1. GenerateFilter方法通过将当前已添加的键集合传递给CreateFilter来生成布隆过滤器；
2. 过滤器生成的结果被追加到result_中，同时将每个过滤器在result_中的起始位置记录在filter_offsets_中；
3. 该方法的设计目标是优化读取性能，通过使用布隆过滤器来快速检查键是否可能存在，从而减少不必要的磁盘读取操作。

最后来看最后一个成员方法**Finish（）**：

```
Slice FilterBlockBuilder::Finish() {
  //检查并生成未处理的过滤器，如果有调用GenerateFilter（）生成过滤器。确保所有添加的键都被处理过并生成的相应的过滤器
  if (!start_.empty()) {
    GenerateFilter();
  }

  //记录每个过滤器的偏移量追加到result_结构中，result_是string类型
  // Append array of per-filter offsets
  const uint32_t array_offset = result_.size();
  for (size_t i = 0; i < filter_offsets_.size(); i++) {
    PutFixed32(&result_, filter_offsets_[i]);
  }
  PutFixed32(&result_, array_offset);
  result_.push_back(kFilterBaseLg);  // Save encoding parameter in result
  return Slice(result_);
}

static const size_t kFilterBaseLg = 11;
```

Finish（）函数用于完成过滤器块构建的方法，主要职责是将已生成的过滤器结果和相关元数据以特定格式存储到结果字符串中。该方法用于实现数据库系统中的布隆过滤器块。



那么现在整体的数据格式就成型了，如上图所示，FilterBlockBuilder单纯的组织数据格式，并不会直接操作文件。

那么至此为止，FilterBlockBuilder这个类整体就有了一个整体的认识了。但似乎还是很迷糊，比如说在StartBlock()中为什么用while循环循环调用了GenerateFilter呢？

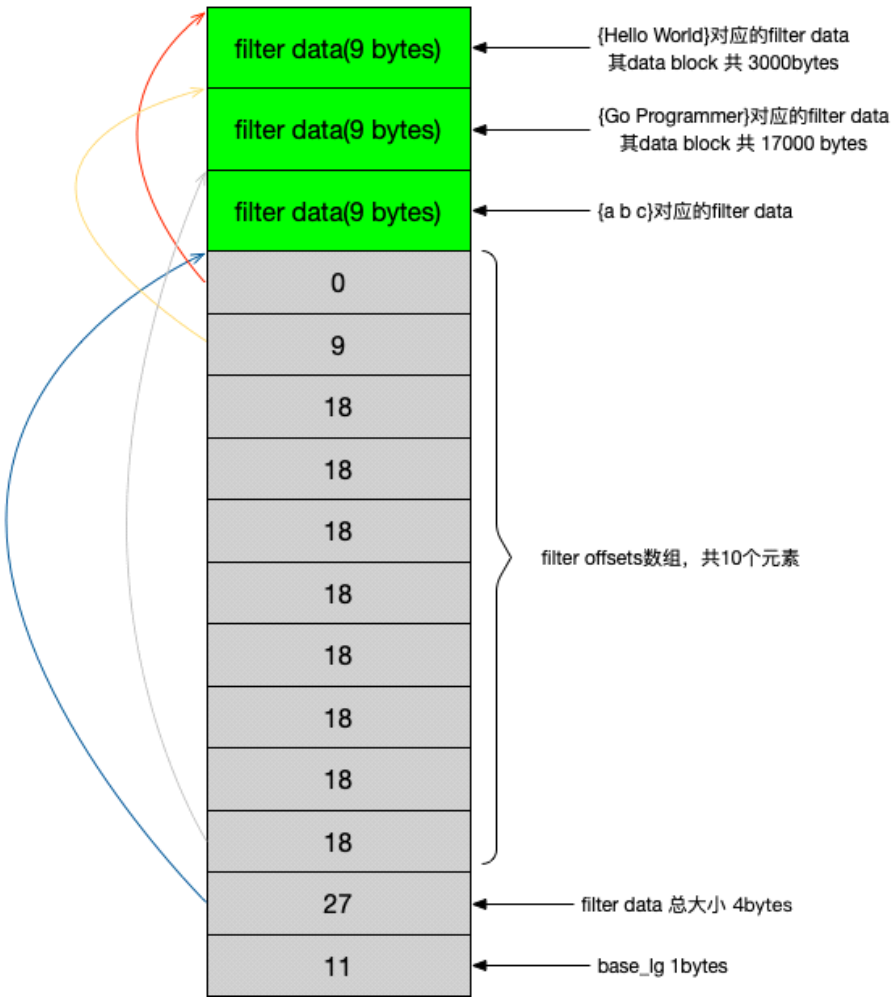
```
class FilterBlockReader {
 public:
  // REQUIRES: "contents" and *policy must stay live while *this is live.
  FilterBlockReader(const FilterPolicy* policy, const Slice& contents);
  bool KeyMayMatch(uint64_t block_offset, const Slice& key);
 private:
  const FilterPolicy* policy_;
  const char* data_;    // Pointer to filter data (at block-start)指向过滤器数据的指针
  const char* offset_;  // Pointer to beginning of offset array (at block-end)指向偏移量数组的指针，
  size_t num_;          // Number of entries in offset array偏移量数组中的条目数
  size_t base_lg_;      // Encoding parameter (see kFilterBaseLg in .cc file)编码参数，通常表示块大小的对数
};
```

```
FilterBlockReader::FilterBlockReader(const FilterPolicy* policy,
                                     const Slice& contents)
    : policy_(policy), data_(nullptr), offset_(nullptr), num_(0), base_lg_(0) {

  //在上图的数据构成的示意图中，最右端至少含有1B的base_lg和4B的filter data size
  size_t n = contents.size();
  if (n < 5) return;  // 1 byte for base_lg_ and 4 for start of offset array
  base_lg_ = contents[n - 1];
  //last_word用来记录解析出来的filter data size的值，这个值就是filters data offsets数组的开始地址
  uint32_t last_word = DecodeFixed32(contents.data() + n - 5);
  if (last_word > n - 5) return;
  data_ = contents.data();    //指向块的起始地址
  offset_ = data_ + last_word;  //指向偏移数组的起始位置
  num_ = (n - 5 - last_word) / 4;  //计算偏移数组中的条目数
}

bool FilterBlockReader::KeyMayMatch(uint64_t block_offset, const Slice& key) {
  uint64_t index = block_offset >> base_lg_;
  if (index < num_) {
    // [start, limit) 标记了一个block_offset对应的filter data
    uint32_t start = DecodeFixed32(offset_ + index * 4); //从偏移数组中获取开始位置
```

```
    uint32_t limit = DecodeFixed32(offset_ + index * 4 + 4); //从偏移数组中获取结束位置
    if (start <= limit && limit <= static_cast<size_t>(offset_ - data_)) {
      //检查范围的有效性后；取出filter data，判断key是否存在
      //创建一个Slice filter用来表示filter data的内容
      //然后过滤策略检查键是否可能存在于该filter data中
      Slice filter = Slice(data_ + start, limit - start);
      return policy_->KeyMayMatch(key, filter);
    } else if (start == limit) {
      // Empty filters do not match any keys
      return false;
    }
  }
  return true;  // Errors are treated as potential matches
}
```

```
bool FilterBlockReader::KeyMayMatch(uint64_t block_offset, const Slice& key) {
```

```
void FilterBlockBuilder::StartBlock(uint64_t block_offset) {
```

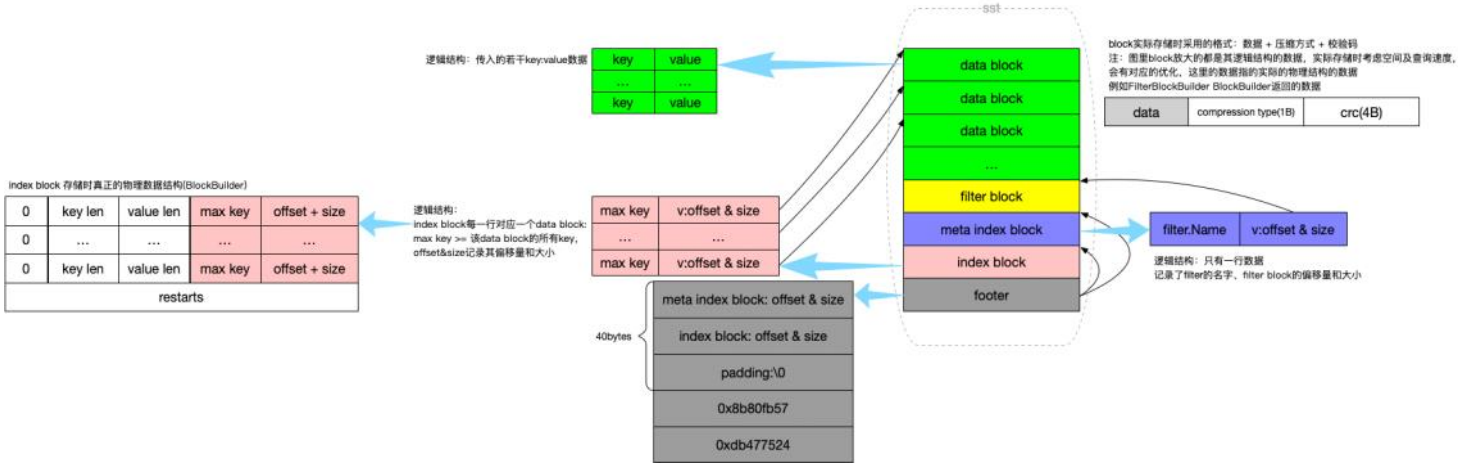

filter data(9 bytes)
filter data(9 bytes)
filter data(9 bytes)
0
9
18
18
18
18
18
18
18
18
27
11
{Hello World}对应的filter data
其data block 共 3000bytes
{Go Programmer}对应的filter data
其data block 共 17000 bytes
{a b c}对应的filter data
filter offsets数组，共10个元素
filter data 总大小 4bytes
base_lg 1bytes

LevelDB笔记13：sstable

2024年8月26日 16:02

levelDB无论哪一层文件，都是sstable的格式，即Sorted String Table。

前面笔记中的block和filter block都是sstable的一个组件，负责构造部分数据格式。
本文介绍 sstable 的设计意图以及完整的数据格式实现，然后通过源码介绍数据格式的构造过程。



从leveldb:: TableBuilder这个类说起。

```
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_;
};
```

构造函数：

```
TableBuilder::TableBuilder(const Options& options, WritableFile* file)
```

```
    : rep_(new Rep(options, file)) {
  if (rep_->filter_block != nullptr) {
    rep_->filter_block->StartBlock(0);
  }
}
```

可以看到构造函数中对内部的成员函数rep_进行了初始化，那么这个rep_是什么呢？？？

在TableBuilder中也没有其他的成员变量了，只有一个rep_，因此我们先来看这玩意是什么吧。

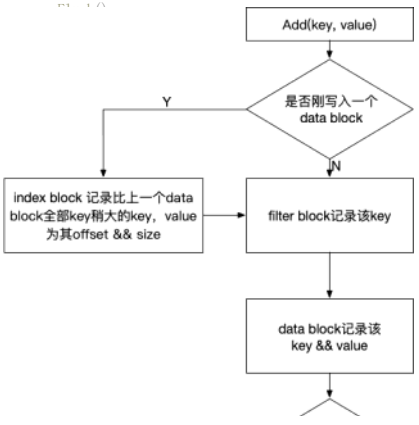
```
  struct Rep;
  Rep* rep_;
```

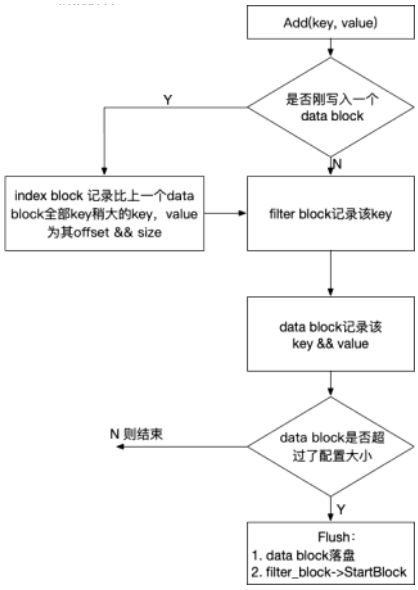
结构体类型，定义如下：

```
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;
};
```

这个Rep封装的数据格式好像看起来也很复杂，先理解Add再说

```
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);
  }
  //1.处理索引条目
  // pending_index_entry如果为true，说明当前data_block已满，更新index_block的信息
  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;
  }
  //2.更新过滤器块
  if (r->filter_block != nullptr) {
    r->filter_block->AddKey(key);
  }
  //3.更新状态和数据块
  r->last_key.assign(key.data(), key.size());
  r->num_entries++;
  r->data_block.Add(key, value);
  //4.检查数据块大小并可能触发写入
  const size_t estimated_block_size = r->data_block.CurrentSizeEstimate();
  if (estimated_block_size >= r->options.block_size) {
```





上图我们从整体的角度展示了整个Add的过程。

但在Add源码中用到了很多比如说新的数据结构或类型看起来还是很模糊。下面对这些进行分析。

先来看这俩：

```
// 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
```

Pending_index_entry这个值用来表示data_block是否已满，已满的话需要将相应索引加入index_block（数据块的索引）中。

那么先来看BlockHandle这个结构：

```
// 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_;
};
```

在Add过程的第一步中处理索引条目时候：

```
    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;
```

这几步是核心步骤，那么先来看EncodeTo（）这个函数吧。

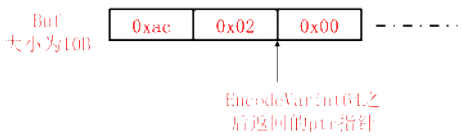
```
void BlockHandle::EncodeTo(std::string* dst) const {
  // Sanity check that all fields have been set
  assert(offset_ != ~static_cast<uint64_t>(0));
  assert(size_ != ~static_cast<uint64_t>(0));
  PutVarint64(dst, offset_);
  PutVarint64(dst, size_);
}
```

Offset_和size_都是uint64类型的。

EncodeTo这个函数主要调用了PutVarint64这个函数，来实现将BlockHandle类中私有的成员变量offset_和size_以变长编码的形式append到dst这个指针指向的string对象中。

```
char* EncodeVarint64(char* dst, uint64_t v) {
  //将一个uint64_t类型的无符号整数编码为可变长度的字节表示（Varint编码）
  //返回一个指向编码后字节序列结尾的指针
  static const int B = 128;
  uint8_t* ptr = reinterpret_cast<uint8_t*>(dst);
  while (v >= B) {
    *(ptr++) = v | B;
    v >>= 7;
  }
  *(ptr++) = static_cast<uint8_t>(v);
  return reinterpret_cast<char*>(ptr);
}
void PutVarint64(std::string* dst, uint64_t v) {
  //使用EncodeVarint64函数将一个64位无符号整数v编码后，追加到给定的字符串对象dst中。
  char buf[10];
  char* ptr = EncodeVarint64(buf, v);
  dst->append(buf, ptr - buf);
```

}



举个直观的例子：

```
std::string encoded;
uint64_t value = 300;
PutVarint64(&encoded, value);
```

运行过程：

1. 300的二进制表示：0000 0001 0010 1100
2. buf是一个大小为10的字符数组，用于存储编码后的字节
3. 调用EncodeVarint64（buf，300）

   第一轮循环：

   v=300>128,进入循环，取300&127：0010 1100 | 1000 0000 =1010 1100（用首位1来表示还未结束变长编码）；

   ptr指针前移一个位置；

   v>>=7，v右移七位，300>>7 =2

   第二轮循环，v=2小于128，不进入循环体

   直接将2写入ptr指向的数据位置，然后ptr前移一个位置；

   最后返回ptr的指针。

4. 返回到PutVarint64函数中，buf此时只用了前两个字节也就是原先300编码后得到的数据0xac和0x02；
5. 最后将这两字节追加到dst中。

```
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;
```

这个过程理解了那么Add(kHandle: key, EncodeTo(&handle_encoding))这个过程也就自然很清晰了。

接着就来看index_block.Add函数了。

Index_block也是一个BlockBuilder的实例对象，即这个索引块也是通过BlockBuilder来构建的。

实际上通过offset_和size我们便可以定位data block地址和大小，EncodeTo函数就实现了将offset和size以变长编码的形式追加到hanle_encoding这个string对象中。不像数据块的Add，索引块的Add是将整个数据块的offset和size整合成一个value，将last_key作为键，调用BlockBuilder中的成员方法Add，实现index_block的初步构建。

```
// 2.更新过滤器块
  if (r->filter_block != nullptr) {
    r->filter_block->AddKey(key);
  }
```

过滤块的详细信息见笔记12.

```
// 3. 更新状态和数据块
  r->last_key.assign(key.data(), key.size());    //更新last_key
  r->num_entries++;
  r->data_block.Add(key, value);
```

我们已经说过data_block和index_block都是BlockBuilder类型的对象。他们使用同一套机制。

```
  // 4.检查数据块大小并可能触发写入
  const size_t estimated_block_size = r->data_block.CurrentSizeEstimate();
  if (estimated_block_size >= r->options.block_size) {
    Flush();
  }
```

其中size_t block_size = 4 * 1024;

可以看到最后调用Flush()函数触发写入。

那么可以这样理解，在Add函数中，当数据块大小达到预定的块大小限制时，会调用Flush（）将数据块写入文件。

```
void TableBuilder::Flush() {
  // 1 状态检查
  Rep* r = rep_;
  assert(!r->closed);
  if (!ok()) return;
  if (r->data_block.empty()) return;
  assert(!r->pending_index_entry);
 // 2 写入数据块
  WriteBlock(&r->data_block, &r->pending_handle);
 // 3 更新状态
  if (ok()) {
    r->pending_index_entry = true;
    r->status = r->file->Flush();  //调用底层文件的flush（）方法，确保数据被刷新到磁盘，避免缓冲区数据丢失
  }
 // 4 更新过滤器块
  if (r->filter_block != nullptr) {
    r->filter_block->StartBlock(r->offset);
  }
}
```

在进一步理解Flush（）过程之前，我们先来分析**Finish（）**函数：

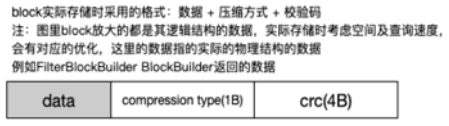
```
Status TableBuilder::Finish() {
```

```
  //1.初始化：
  Rep* r = rep_;
  Flush();          //调用Flush()方法，将当前的data_block写入文件
  assert(!r->closed);  //确保Table尚未关闭
  r->closed = true;    //
  //事先创建BlockHandle对象，来存储相应块的偏移量和大小信息，一定要注意这个大小size不包含CompressionType位和crc位
  BlockHandle filter_block_handle, metaindex_block_handle, index_block_handle;

  //2.写入过滤块
  // Write filter block
  if (ok() && r->filter_block != nullptr) {
    WriteRawBlock(r->filter_block->Finish(), kNoCompression,
                  &filter_block_handle);
  }
  //3.写入meta index block
  // 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);
  }
  //4.写入索引块
  // 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;
    }
    //可以看到写入索引块的上面这几个步骤实质上的只是调用Add将键值插入到block中，但我们知道block的结构除了键值对的多个entry，还需追加restarts_数组和其数量。这个过程既然没有在上面实现，那就只能在下面WriteBlock（）中实现了。
    WriteBlock(&r->index_block, &index_block_handle);
  }
  //5.写入Footer
  // 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;
}
```

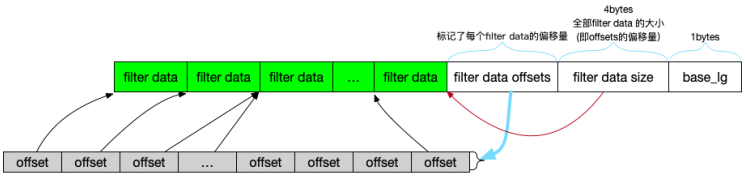
来看这个过程：

```
    WriteRawBlock(r->filter_block->Finish(), kNoCompression,
                  &filter_block_handle);
```

这个过程将r->filter_block->Finish()的返回值作为第一个参数传入，这个过程就是实现了对filter_block的构造（见笔记12），事实上这个构造好的并不是真正存入盘后的filter_block，如下图所示。r->filter_block->Finish()返回的Slice数据就是下图中展示的data部分。
依次写入block_contents、1B的compression_type、4B的crc，其中后五个字节称BlockTrailer
Data block、filter block、meta index block和index block都按照下面这个格式组织，区别是block_contents格式不同。
特别要注意：handel用于返回，记录写入前的文件的offset和block_contents大小。



data的部分结构如下图所示：




```
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; //更新偏移量
    }
  }
}
```

其中kBlockTrailerSize是:

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

其中CompressionType是个枚举类型数据，如下所示:

```
// DB contents are stored in a set of blocks, each of which holds a
// sequence of key,value pairs.  Each block may be compressed before
// being stored in a file.  The following enum describes which
// compression method (if any) is used to compress a block.
enum CompressionType {
  // NOTE: do not change the values of existing entries, as these are
  // part of the persistent format on disk.
  kNoCompression = 0x0,
  kSnappyCompression = 0x1,
  kZstdCompression = 0x2,
};
```

在Finish（）过程的3、4步骤，都调用了WriteBlock函数:

```
    WriteBlock(&meta_index_block, &metaindex_block_handle);
    WriteBlock(&r->index_block, &index_block_handle);
```

定义如下:

```
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();    //在这个Status TableBuilder::Finish()中只是加入了键值对
```

entry多个，这里调用block->Finish（）过程实现了完整的BlockBuilder过程，将restarts追加到block中。

```
  Slice block_contents;
  CompressionType type = r->options.compression;    //压缩类型
  // TODO(postrelease): Support more compression options: zlib?
  switch (type) {
    case kNoCompression:
      block_contents = raw;     //没有压缩，原始BlockBuilder构建的block就作为
```

block_contents的内容，在这个基础上加上1byte的compressionType和4bytes的crc校验位

```
      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;
    }
    case kZstdCompression: {
      std::string* compressed = &r->compressed_output;
      if (port::Zstd_Compress(r->options.zstd_compression_level, raw.data(),
                              raw.size(), compressed) &&
          compressed->size() < raw.size() - (raw.size() / 8u)) {
        block_contents = *compressed;
      } else {
        // Zstd 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();
}
```

截至目前，Status TableBuilder::Finish()这个过程中只剩下写入Footer的过程不清楚了，其中的主要代码如下:

```
    //5. 写入Footer
    // 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();
      }
    }
```

//处理完压缩的情况后调用WriteRawBlock函数，实现在盘中存储结构的构建，就是在block_contents（BLockBuilder构建的block）基础上，添加1B的CompressionType和4bytes的crc

先来看什么是Footer?

```
// 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_;
};
```

在第5步写入Footer的过程中：

```
    std::string footer_encoding;
    footer.EncodeTo(&footer_encoding);
    r->status = r->file->Append(footer_encoding);
```

主要就这三步，构造了一个string类型的footer_encoding，然后调用EncodeTo的成员方法；最后调用Append（）实现写入。

EncodeTo实现了将完整的Footer信息包含40B的metaindex_handle_和index_handle信息，还有两个4B的魔数追加到dst中，并用dst进行返回。之后再调用`r->file->Append(footer_encoding)`实现了将组件好的整个footer累加到file中。

```
void Footer::EncodeTo(std::string* dst) const {
  const size_t original_size = dst->size();
  metaindex_handle_.EncodeTo(dst);
  index_handle_.EncodeTo(dst);
  dst->resize(2 * BlockHandle::kMaxEncodedLength);  // Padding
  PutFixed32(dst, static_cast<uint32_t>(kTableMagicNumber & 0xffffffffu));
  PutFixed32(dst, static_cast<uint32_t>(kTableMagicNumber >> 32));
  assert(dst->size() == original_size + kEncodedLength);
  (void)original_size;  // Disable unused variable warning.
}
```

其中：

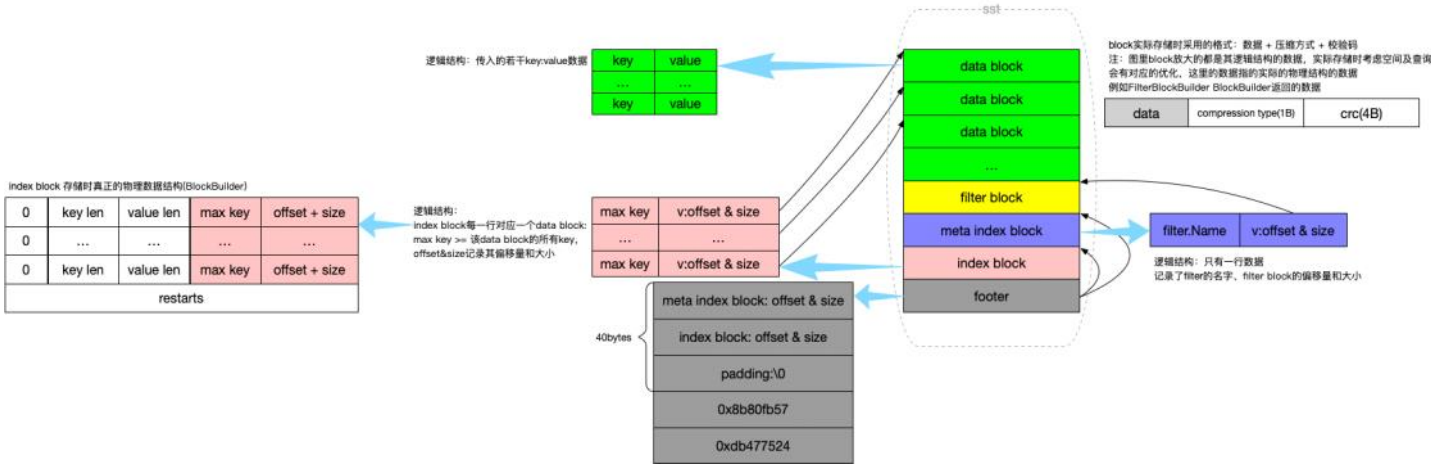
```
  enum { kMaxEncodedLength = 10 + 10 };
  static const uint64_t kTableMagicNumber = 0xdb4775248b80fb57ull;
```

读取时直接 seek 到 f.size() - 48，读取接下来的 48 个 bytes，就可以读到 footer EncodeTo 后的数据了。

LevelDB笔记14：sstable读取

2024年8月27日 23:24

前面主要介绍了sstable，包括sstable的结构组成及写入的源码分析，本文主要介绍下
对应的读取过程。再次将整个结构图搬过来：



基本过程：

读取的基本过程，简单总结就是四个字：**按图索骥**

各个索引在这个过程中发挥了很大的作用。

首先是seek到文件末尾读取固定48个字节大小的footer，这也是为什么footer是定长的原因。

然后解析出meta_index_block以及index_block.

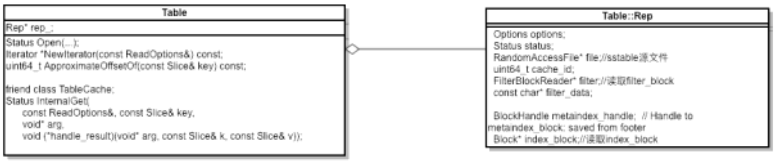
然后通过meta_index_block解析出filter block，通过index_block解析出data_block。

查找时，先通过filter block查找是否存在，然后通过data_block解析出对应的value。

源码位于table.cc

相关类图：

空心菱形箭头表示聚合关系（可以看C++设计模式的笔记）



可以看到Table的对外接口比较简单。具体源码如下：

```
// A Table is a sorted map from strings to strings.  Tables are
// immutable and persistent.  A Table may be safely accessed from
// multiple threads without external synchronization.
class LEVELDB_EXPORT Table {
 public:
  // Attempt to open the table that is stored in bytes [0..file_size)
  // of "file", and read the metadata entries necessary to allow
  // retrieving data from the table.
  //
  // If successful, returns ok and sets "*table" to the newly opened
  // table.  The client should delete "*table" when no longer needed.
  // If there was an error while initializing the table, sets "*table"
  // to nullptr and returns a non-ok status.  Does not take ownership of
  // "*source", but the client must ensure that "source" remains live
  // for the duration of the returned table's lifetime.
  //
  // *file must remain live while this Table is in use.
  static Status Open(const Options& options, RandomAccessFile* file,
                     uint64_t file_size, Table** table);
  Table(const Table&) = delete;
  Table& operator=(const Table&) = delete;
  ~Table();
  // Returns a new iterator over the table contents.
  // The result of NewIterator() is initially invalid (caller must
  // call one of the Seek methods on the iterator before using it).
  Iterator* NewIterator(const ReadOptions&) const;
  // Given a key, return an approximate byte offset in the file where
  // the data for that key begins (or would begin if the key were
  // present in the file).  The returned value is in terms of file
  // bytes, and so includes effects like compression of the underlying data.
```

```
  // E.g., the approximate offset of the last key in the table will
  // be close to the file length.
  uint64_t ApproximateOffsetOf(const Slice& key) const;
 private:
  friend class TableCache;
  struct Rep;
  static Iterator* BlockReader(void*, const ReadOptions&, const Slice&);
  explicit Table(Rep* rep) : rep_(rep) {}
  // Calls (*handle_result)(arg, ...) with the entry found after a call
  // to Seek(key).  May not make such a call if filter policy says
  // that key is not present.
  Status InternalGet(const ReadOptions&, const Slice& key, void* arg,
                     void (*handle_result)(void* arg, const Slice& k,
                                           const Slice& v));
  void ReadMeta(const Footer& footer);
  void ReadFilter(const Slice& filter_handle_value);
  Rep* const rep_;
};
```

其中Rep的结构如下：

```
struct Table::Rep {
  ~Rep() {
    delete filter;
    delete[] filter_data;
    delete index_block;
  }
  Options options;
  Status status;
  RandomAccessFile* file;
  uint64_t cache_id;
  FilterBlockReader* filter;
  const char* filter_data;
  BlockHandle metaindex_handle;  // Handle to metaindex_block: saved from footer
  Block* index_block;
};
```

从Open函数入手，它用于打开sstable文件，并初始化一个Table*的对象。

```
Status Table::Open(const Options& options, RandomAccessFile* file,
                   uint64_t size, Table** table) {
  //1 初始化table指针
  *table = nullptr;
  //2 文件大小检查
  if (size < Footer::kEncodedLength) {
    //kEncodeLength=48，代表着Footer结构的定长大小48B
    return Status::Corruption("file is too short to be an sstable");
  }
 //3 读取Footer
  char footer_space[Footer::kEncodedLength];
  Slice footer_input;
  Status s = file->Read(size - Footer::kEncodedLength, Footer::kEncodedLength,
                        &footer_input, footer_space);  //从文件末尾读取48字节到footer_space中，并将其包装到Slice对象
footer_input中。
  if (!s.ok()) return s;
  Footer footer;
  s = footer.DecodeFrom(&footer_input); //解码Footer
  if (!s.ok()) return s;

  //4 读取索引块
  // Read the index block
  BlockContents index_block_contents;
  ReadOptions opt;
  if (options.paranoid_checks) {
    opt.verify_checksums = true;
  } //如果启用了严格检查，则验证校验和
  s = ReadBlock(file, opt, footer.index_handle(), &index_block_contents); //调用ReadBlock函数，读取由
footer.index_handle()指定的索引块。用index_block_contents加以记录并返回。

  //5 构建Table对象
  if (s.ok()) {
    // We've successfully read the footer and the index block: we're
    // ready to serve requests.
    Block* index_block = new Block(index_block_contents); //创建一个新的Block对象来封装索引块内容。
    Rep* rep = new Table::Rep;
    rep->options = options;
    rep->file = file;
    rep->metaindex_handle = footer.metaindex_handle();
    rep->index_block = index_block;
    rep->cache_id = (options.block_cache ? options.block_cache->NewId() : 0);
    rep->filter_data = nullptr;
    rep->filter = nullptr;
    *table = new Table(rep); 创建一个新的Table对象，并将其指针赋值给table
    (*table)->ReadMeta(footer); //调用ReadMeta方法，读取元数据块的信息。
  }
  return s;
}
```

Table::Open这是一个静态方法，用于打开一个已经存在的SSTable文件，并将其内容加载到内存中以供查询，总的来说这部分代码的主要功能是：

1. 检查文件大小是否足够；
2. 读取文件的Footer（尾部），从中提取元数据索引和索引块的句柄；
3. 根据从Footer中提取的信息读取索引块；

再来看迭代器：

```
Iterator* NewIterator(const ReadOptions&) const;
```

定义如下：初始化Table对象，准备好用于数据读取。

```
Iterator* Table::NewIterator(const ReadOptions& options) const {
  return NewTwoLevelIterator(
      rep_->index_block->NewIterator(rep_->options.comparator),
      &Table::BlockReader, const_cast<Table*>(this), options);
}
```

第一个参数传入index_block的iterator，用于第一层查找。查找到的value会传递给第二个参数

（函数指针），该函数支持解析value的data block，第三、四个参数都在函数调用时使用。

```
Iterator* NewTwoLevelIterator(Iterator* index_iter,
                              BlockFunction block_function, void* arg,
                              const ReadOptions& options) {
  return new TwoLevelIterator(index_iter, block_function, arg, options);
}
```

NewTwoLevelIterator实际上是返回TwoLevelIterator

TwoLevelIterator这个两级的迭代器类继承自Iterator，实现了Seek、Prev、Next、key和value等一系列接口。

为什么要这么起名字？TwoLevelIterator？？？

我们试着考虑一下sstable的数据格式，不考虑filter的话，查找是一个二层递进的过程：先查找index block，查看可能处于哪个data block，然后查找data block，找到相对应的value，因此需要两层Iterator，分别用于index block和data block。

事实上，TwoLevelIterator是LevelDB中实现的一种迭代器，主要用于遍历由两级索引结构构成的数据库。具体说就是这种迭代器用于处理有index block和data block两层结构的SSTable文件。

那么我们先来看TwoLevelIterator这个类：

```
class TwoLevelIterator : public Iterator {
 public:
  TwoLevelIterator(Iterator* index_iter, BlockFunction block_function,
                   void* arg, const ReadOptions& options);
  ~TwoLevelIterator() override;
  void Seek(const Slice& target) override;
  void SeekToFirst() override;
  void SeekToLast() override;
  void Next() override;
  void Prev() override;
  bool Valid() const override { return data_iter_.Valid(); }
  Slice key() const override {
    assert(Valid());
    return data_iter_.key();
  }
  Slice value() const override {
    assert(Valid());
    return data_iter_.value();
  }
  Status status() const override {
    // It's d be nice if status() returned a const Status& instead of a Status
    if (!index_iter_.status().ok()) {
      return index_iter_.status();
    } else if (data_iter_.iter() != nullptr && !data_iter_.status().ok()) {
      return data_iter_.status();
    } else {
      return status_;
    }
  }
 private:
  void SaveError(const Status& s) {
    if (status_.ok() && !s.ok()) status_ = s;
  }
  void SkipEmptyDataBlocksForward();
  void SkipEmptyDataBlocksBackward();
  void SetDataIterator(Iterator* data_iter);
  void InitDataBlock();
  BlockFunction block_function_;
  void* arg_;
  const ReadOptions options_;
  Status status_;
  IteratorWrapper index_iter_;
  IteratorWrapper data_iter_;  // May be nullptr
  // If data_iter_ is non-null, then "data_block_handle_" holds the
  // "index_value" passed to block_function_ to create the data_iter_.
  std::string data_block_handle_;
};
```

其中的BlockFunction为：

```
typedef Iterator* (*BlockFunction)(void*, const ReadOptions&, const Slice&);
```

构造函数为：

```
TwoLevelIterator::TwoLevelIterator(Iterator* index_iter,
                                   BlockFunction block_function, void* arg,
                                   const ReadOptions& options)
    : block_function_(block_function),
      arg_(arg),
      options_(options),
      index_iter_(index_iter),
      data_iter_(nullptr) {}
```

在这个TwoLevelIterator中，正如其名，可以发现存在两个迭代器，分别指向index block 和data block。

```
  IteratorWrapper index_iter_;
  IteratorWrapper data_iter_;  // May be nullptr
```

试着从Seek过程入手进行理解：

```
  Seek(const Slice& target) override;
```

定义如下：

```
void TwoLevelIterator::Seek(const Slice& target) {
  //先在index block找到第一个>=target的k:v，v是某个data_block的size&offset
  index_iter_.Seek(target);
  //根据v读取data_block，data_iter_指向该data_block内的k:v
  InitDataBlock();    //TwoLevelIterator中的私有的成员方法，调用这个过程实现对TwoLevelIterator类中的data_iter_的初始化，因为在TwoLevelIterator构造函数中初始化了data_iter_为nullter
  if (data_iter_.iter() != nullptr) data_iter_.Seek(target);
  SkipEmptyDataBlocksForward();
}
```

先通过index_iter找到第一个>=target的key，对应的value是某个data block的offset&&size，接下来继续在data block内查找。

先通过index_iter_找到第一个 >= target 的 key, 对应的 value 是某个 data block 的 size&offset，

接下来继续在该 data block 内查找。

为什么可以直接在这个 data block 内查找? 我们看下原因。

首先，(index_iter_ - 1).key() < target，而index_iter_ - 1对应的 data block 内所有的 key 都满足 <= (index_iter_ - 1)->key()，因此该 data block1 内所有的 key 都满足< target.

其次，index_iter_.key() >= target，而index_iter_对应的 data block2 内所有的 key 都满足 <= index_iter_->key()。
而 data block123是连续的。
因此，如果 target 存在于该 sstable，那么一定存在于index_iter_当前指向的 data block.
注：
如果index_iter_指向第一条记录，那么index_iter - 1无效，但不影响该条件成立。
关于index_iter_的 key 的取值，参考[leveldb-sstable r->last_key]的介绍

之后就是调用InitDataBlock初始化，使得data_iter_指向该data block，其中就用到了传入的block_function
先来看InitDataBlock这个函数:

```
void TwoLevelIterator::InitDataBlock() {
  if (!index_iter_.Valid()) {
    SetDataIterator(nullptr);
  } else {
    Slice handle = index_iter_.value();
    if (data_iter_.iter() != nullptr &&
        handle.compare(data_block_handle_) == 0) {
      // data_iter_ is already constructed with this iterator, so
      // no need to change anything
    } else {
      Iterator* iter = (*block_function_)(arg_, options_, handle);
      data_block_handle_.assign(handle.data(), handle.size());
      SetDataIterator(iter);
    }
  }
}
```

一定要注意上面的index_iter这个是rep_->index_block->NewIterator(rep_->options.comparator这个函数返回的迭代器，这部分可以参见笔记11Block读取的部分，调用的实际上是Block类中的NewIterator()方法来实现建立一个迭代器（在此处是index_iter）

```
typedef Iterator* (*BlockFunction)(void*, const ReadOptions&, const Slice&);
```

Block_function是一个函数指针类型，它接受三个参数，返回一个指向Iterator对象的指针。
其中的成员变量： BlockFunction block_function_;
Block_function_用于存储一个指向特定实现的函数指针，
Table::BlockReader
在上面调用过程中，这个block_function会指向Table：：BlockReader这个方法。
在实际调用过程中上面标黄的部分会被替换为：

```
Iterator* iter = (Table::BlockReader)(arg_, options_, handle);
```

InitDataBlock函数中传递给BlockReader函数的handel就是index_block相应的某个表项的value的值（即对应data_block的offset和size）
那么我们需要先来看Table:: BlockReader

```
// Convert an index iterator value (i.e., an encoded BlockHandle)
// into an iterator over the contents of the corresponding block.
Iterator* Table::BlockReader(void* arg, const ReadOptions& options,
                             const Slice& index_value) {
```

//index_value表示的是当前索引块的迭代器指向的那个条目（那个索引）的value（实际上就是对应数据块的offset和size编码后的value），在后面调用了handle.DecodeFrom()的过程实现了解析。

```
  Table* table = reinterpret_cast<Table*>(arg);
  Cache* block_cache = table->rep_->options.block_cache;
  Block* block = nullptr;
  Cache::Handle* cache_handle = nullptr;
  BlockHandle handle;
  Slice input = index_value;
  Status s = handle.DecodeFrom(&input);
```

//将data_block对应的offset和size从index_value中解析出来，意味着此时handle可以定位相应的data_block了

```
  // We intentionally allow extra stuff in index_value so that we
  // can add more features in the future.
  if (s.ok()) {
    BlockContents contents;
    if (block_cache != nullptr) {
      char cache_key_buffer[16];
      EncodeFixed64(cache_key_buffer, table->rep_->cache_id);
      EncodeFixed64(cache_key_buffer + 8, handle.offset());
      Slice key(cache_key_buffer, sizeof(cache_key_buffer));
      cache_handle = block_cache->Lookup(key);
      if (cache_handle != nullptr) {
        block = reinterpret_cast<Block*>(block_cache->Value(cache_handle));
      } else {
        s = ReadBlock(table->rep_->file, options, handle, &contents);
        if (s.ok()) {
          block = new Block(contents);
          if (contents.cachable && options.fill_cache) {
            cache_handle = block_cache->Insert(key, block, block->size(),
                                               &DeleteCachedBlock);
          }
        }
      }
    } else {
      s = ReadBlock(table->rep_->file, options, handle, &contents);
      if (s.ok()) {
        block = new Block(contents);
      }
    }
  }
```

//直观上上面的作用便是从handle对应的位置（offset和size）中得到相应data_block的内容contents，用contents这个变量返回，该数据块的内容。之后调用Block的构造函数新建一个保存该内容的数据块。
在上面的过程中调用了LRUCache的相关内容，这个可以结合参见笔记15和16进行学习。
//新建完这个data_block后，调用Block->NewIterator()方法以返回一个data_block中的迭代器

iter，可以结合笔记11Block的读取过程进行分析。

```
  Iterator* iter;
  if (block != nullptr) {
    iter = block->NewIterator(table->rep_->options.comparator);
    if (cache_handle == nullptr) {
      iter->RegisterCleanup(&DeleteBlock, block, nullptr);
    } else {
      iter->RegisterCleanup(&ReleaseBlock, block_cache, cache_handle);
    }
  } else {
    iter = NewErrorIterator(s);
  }
  return iter;
}
```

得到数据块的iter后调用了：

```
  if (block != nullptr) {
    iter = block->NewIterator(table->rep_->options.comparator);
    if (cache_handle == nullptr) {
      iter->RegisterCleanup(&DeleteBlock, block, nullptr);
    } else {
      iter->RegisterCleanup(&ReleaseBlock, block_cache, cache_handle);
    }
  } else {
    iter = NewErrorIterator(s);
  }
  return iter;
}
```

这个方法也是TwoLevelIterator类中的一个私有的成员方法：更新设置了data_iter_.

该函数定义如下：

```
void TwoLevelIterator::SetDataIterator(Iterator* data_iter) {
  if (data_iter_.iter() != nullptr) SaveError(data_iter_.status());
  data_iter_.Set(data_iter);
}
```

在这个TwoLevelIterator类中，有两层索引，一层是index_block的迭代器，一层是data_block的迭代器，我们在之前理解的时候将IteratorWrapper这个类等价于Iterator了，我们在这里再看看这个IteratorWrappeder这个类是啥样的。

```
  IteratorWrapper index_iter_;
  IteratorWrapper data_iter_;  // May be nullptr
```

这个类的实现在源码的iterator_wrapper.h中，具体内容如下所示：

```
// A internal wrapper class with an interface similar to Iterator that
// caches the valid() and key() results for an underlying iterator.
// This can help avoid virtual function calls and also gives better
// cache locality.
class IteratorWrapper {
 public:
  IteratorWrapper() : iter_(nullptr), valid_(false) {}
  explicit IteratorWrapper(Iterator* iter) : iter_(nullptr) { Set(iter); }
  ~IteratorWrapper() { delete iter_; }
  Iterator* iter() const { return iter_; }
  // Takes ownership of "iter" and will delete it when destroyed, or
  // when Set() is invoked again.
  void Set(Iterator* iter) {
    delete iter_;
    iter_ = iter;
    if (iter_ == nullptr) {
      valid_ = false;
    } else {
      Update();
    }
  }
  // Iterator interface methods
  bool Valid() const { return valid_; }
  Slice key() const {
    assert(Valid());
    return key_;
  }
  Slice value() const {
    assert(Valid());
    return iter_->value();
  }
  // Methods below require iter() != nullptr
  Status status() const {
    assert(iter_);
    return iter_->status();
  }
  void Next() {
    assert(iter_);
    iter_->Next();
    Update();
  }
  void Prev() {
    assert(iter_);
    iter_->Prev();
    Update();
  }
  void Seek(const Slice& k) {
    assert(iter_);
    iter_->Seek(k);
    Update();
  }
  void SeekToFirst() {
    assert(iter_);
    iter_->SeekToFirst();
    Update();
  }
  void SeekToLast() {
    assert(iter_);
    iter_->SeekToLast();
    Update();
  }
 private:
  void Update() {
    valid_ = iter_->Valid();
    if (valid_) {
      key_ = iter_->key();
    }
  }
  Iterator* iter_;
  bool valid_;
  Slice key_;
};
```

再来理一遍Seek的过程：

1. 先根据target在index_block中查找相应的索引条目（一级迭代器）
2. 根据索引条目定位到相应的data_block中；
3. 再在data_block中查找相应的与target匹配的条目entry（即key：value）（二级迭代器）

从Table出发再来理一遍如何定位到最终的键值对上：

1. 首先需要打开table；
2. 创建一个迭代器Iterator* Table::NewIterator
3. Tabel:: NewIterator(.)的过程实际上返回一个TwoLevelIterator的实例对象；
4. 在TwoLevelIterator对象中封装了index_iter_和data_iter_用来表示两层迭代器（index_block和data_block）

```
  IteratorWrapper index_iter_;
  IteratorWrapper data_iter_;  // May be nullptr
```

上面所述的过程忽略了过滤块的过程和作用，这点在哪呢？？？

```
  friend class TableCache;
```

Table中声明了友元类TableCache，主要用于直接调用InteralGet这个函数（Table类中的私有成员函数），查找过程跟NewIterator很像，只是多了很多filter的过程，如果filter->KeyMayMatch显示不存在，那么直接返回。

这个内容后续再补充。。。。

LevelDB笔记15：LRUCache的实现

2024年8月29日 11:54

先试着从源码出发，找到了cache.h的源文件，发现这个文件下定义了一个抽象的基类Cache（不能直接实例化），定义了一系列的关于缓存操作的接口，具体的缓存行为（如插入、查找、删除等）需要在派生类中实现。LevelDB中的有关的实现在源码中的cache.cc中。

1. 这是cache.h中的内容：

```
LEVELDB_EXPORT Cache* NewLRUCache(size_t capacity);


// Create a new cache with a fixed size capacity.  This implementation
// of Cache uses a least-recently-used eviction policy.
LEVELDB_EXPORT Cache* NewLRUCache(size_t capacity);
class LEVELDB_EXPORT Cache {
 public:
  Cache() = default;
  Cache(const Cache&) = delete;
  Cache& operator=(const Cache&) = delete;
  // Destroys all existing entries by calling the "deleter"
  // function that was passed to the constructor.
  virtual ~Cache();
  // Opaque handle to an entry stored in the cache.
  struct Handle {};
  // Insert a mapping from key->value into the cache and assign it
  // the specified charge against the total cache capacity.
  //
  // Returns a handle that corresponds to the mapping.  The caller
  // must call this->Release(handle) when the returned mapping is no
  // longer needed.
  //
  // When the inserted entry is no longer needed, the key and
  // value will be passed to "deleter".
  virtual Handle* Insert(const Slice& key, void* value, size_t charge,
                         void (*deleter)(const Slice& key, void* value)) = 0;
  // If the cache has no mapping for "key", returns nullptr.
  //
  // Else return a handle that corresponds to the mapping.  The caller
  // must call this->Release(handle) when the returned mapping is no
  // longer needed.
  virtual Handle* Lookup(const Slice& key) = 0;
  // Release a mapping returned by a previous Lookup().
  // REQUIRES: handle must not have been released yet.
  // REQUIRES: handle must have been returned by a method on *this.
  virtual void Release(Handle* handle) = 0;
  // Return the value encapsulated in a handle returned by a
  // successful Lookup().
  // REQUIRES: handle must not have been released yet.
```

```
  // REQUIRES: handle must have been returned by a method on *this.
  virtual void* Value(Handle* handle) = 0;
  // If the cache contains entry for key, erase it.  Note that the
  // underlying entry will be kept around until all existing handles
  // to it have been released.
  virtual void Erase(const Slice& key) = 0;
  // Return a new numeric id.  May be used by multiple clients who are
  // sharing the same cache to partition the key space.  Typically the
  // client will allocate a new id at startup and prepend the id to
  // its cache keys.
  virtual uint64_t NewId() = 0;
  // Remove all cache entries that are not actively in use.  Memory-constrained
  // applications may wish to call this method to reduce memory usage.
  // Default implementation of Prune() does nothing.  Subclasses are strongly
  // encouraged to override the default implementation.  A future release of
  // leveldb may change Prune() to a pure abstract method.
  virtual void Prune() {}
  // Return an estimate of the combined charges of all elements stored in the
  // cache.
  virtual size_t TotalCharge() const = 0;
};
```

2. 这是cache.cc中定义的相关类和结构：

```
> struct LRUHandle {…

  // We provide our own simple hash table since it removes a whole bunch
  // of porting hacks and is also faster than some of the built-in hash
  // table implementations in some of the compiler/runtime combinations
  // we have tested.  E.g., readrandom speeds up by ~5% over the g++
  // 4.4.3's builtin hashtable.
> class HandleTable {…

  // A single shard of sharded cache.
> class LRUCache {…
```

```
  static const int kNumShardBits = 4;
  static const int kNumShards = 1 << kNumShardBits;

> class ShardedLRUCache : public Cache {…

  }  // end anonymous namespace

  Cache* NewLRUCache(size_t capacity) { return new ShardedLRUCache(capacity); }
```

可以看到在cache.cc中只有一个类SharedLRUCache继承自Cache类。
除了这个类还定义了LRUHandle、HandleTable和LRUCache的类。

那么应该从什么地方入手学习LevelDB中的缓存机制呢?

先不管别的，可以看到在cache.h中定义了全局的方法：

```
LEVELDB_EXPORT Cache* NewLRUCache(size_t capacity);
```

源码中的注释是：

```
// Create a new cache with a fixed size capacity.  This implementation
// of Cache uses a least-recently-used eviction policy.
```

也就是说这个方法创建了一个有着固定大小的新的cache，缓存淘汰机制采用LRU算法。

这样感觉说的很笼统，那么首先来看这个函数的具体定义，他就在cache.cc的文件中：

定义如下：

```
Cache* NewLRUCache(size_t capacity) { return new ShardedLRUCache(capacity); }
```

好像就是实例化了一个SharedLRUCache对象以返回就完了。那么这个SharedLRUCache是啥样的呢?

它的详细结构如下：

```
class ShardedLRUCache : public Cache {
 private:
  LRUCache shard_[kNumShards];
  port::Mutex id_mutex_;
  uint64_t last_id_;
  static inline uint32_t HashSlice(const Slice& s) {
    return Hash(s.data(), s.size(), 0);
  }
  static uint32_t Shard(uint32_t hash) { return hash >> (32 - kNumShardBits); }
 public:
  //构造函数，首先计算了什么per_share然后根据此调用了LRUCache的成员方法SetCapacity（）
  explicit ShardedLRUCache(size_t capacity) : last_id_(0) {
    const size_t per_shard = (capacity + (kNumShards - 1)) / kNumShards;
    for (int s = 0; s < kNumShards; s++) {
      shard_[s].SetCapacity(per_shard);
    }
  }
  ~ShardedLRUCache() override {}
  Handle* Insert(const Slice& key, void* value, size_t charge,
                 void (*deleter)(const Slice& key, void* value)) override {
    const uint32_t hash = HashSlice(key);
    return shard_[Shard(hash)].Insert(key, hash, value, charge, deleter);
  }
  Handle* Lookup(const Slice& key) override {
    const uint32_t hash = HashSlice(key);
    return shard_[Shard(hash)].Lookup(key, hash);
  }
  void Release(Handle* handle) override {
    LRUHandle* h = reinterpret_cast<LRUHandle*>(handle);
    shard_[Shard(h->hash)].Release(handle);
  }
  void Erase(const Slice& key) override {
    const uint32_t hash = HashSlice(key);
    shard_[Shard(hash)].Erase(key, hash);
  }
  void* Value(Handle* handle) override {
    return reinterpret_cast<LRUHandle*>(handle)->value;
  }
  uint64_t NewId() override {
    MutexLock l(&id_mutex_);
```

```
    return ++(last_id_);
  }
  void Prune() override {
    for (int s = 0; s < kNumShards; s++) {
      shard_[s].Prune();
    }
  }
  size_t TotalCharge() const override {
    size_t total = 0;
    for (int s = 0; s < kNumShards; s++) {
      total += shard_[s].TotalCharge();
    }
    return total;
  }
};
```

既然这个类SharedCache中主要的成员变量就是

```
  LRUCache shard_[kNumShards];
```

那么先来看**LRUCache**是什么吧。

```
// A single shard of sharded cache.
class LRUCache {
 public:
  LRUCache();
  ~LRUCache();
  // Separate from constructor so caller can easily make an array of LRUCache
  void SetCapacity(size_t capacity) { capacity_ = capacity; }
  // Like Cache methods, but with an extra "hash" parameter.
  Cache::Handle* Insert(const Slice& key, uint32_t hash, void* value,
                        size_t charge,
                        void (*deleter)(const Slice& key, void* value));
  Cache::Handle* Lookup(const Slice& key, uint32_t hash);
  void Release(Cache::Handle* handle);
  void Erase(const Slice& key, uint32_t hash);
  void Prune();
  size_t TotalCharge() const {
    MutexLock l(&mutex_);
    return usage_;
  }
 private:
  void LRU_Remove(LRUHandle* e);
  void LRU_Append(LRUHandle* list, LRUHandle* e);
  void Ref(LRUHandle* e);
  void Unref(LRUHandle* e);
  bool FinishErase(LRUHandle* e) EXCLUSIVE_LOCKS_REQUIRED(mutex_);
  // Initialized before use.
  size_t capacity_;
  // mutex_ protects the following state.
  mutable port::Mutex mutex_;
  size_t usage_ GUARDED_BY(mutex_);
  // Dummy head of LRU list.
  // lru.prev is newest entry, lru.next is oldest entry.
  // Entries have refs==1 and in_cache==true.
  LRUHandle lru_ GUARDED_BY(mutex_);
  // Dummy head of in-use list.
  // Entries are in use by clients, and have refs >= 2 and in_cache==true.
```

```
  LRUHandle in_use_ GUARDED_BY(mutex_);
  HandleTable table_ GUARDED_BY(mutex_);
};
```

统筹一看这个LRUCache这个类，发现里面聚合了LRUHandle和HandleTable这两个类，来看这两个结构的具体实现：

首先是**LRUHandle**，这是个结构体：

```
struct LRUHandle {
  void* value;
  void (*deleter)(const Slice&, void* value);
  LRUHandle* next_hash; // HandleTable hash冲突时候指向下一个LRUHandle*
  LRUHandle* next; //LRU链表双向指针
  LRUHandle* prev; //LRU链表双向指针
   //用于计算LRUCache容量
  size_t charge;  // TODO(opt): Only allow uint32_t?
  size_t key_length; //key长度
   //是否在LRUCache in_use_链表
  bool in_cache;     // Whether entry is in the cache.
  //引用计数，用于删除数据
  uint32_t refs;     // References, including cache reference, if present.
  //key对应的hash值
  uint32_t hash;     // Hash of key(); used for fast sharding and comparisons
 // 占位符，结构体末尾，通过key_length获取真正的key
  char key_data[1];  // Beginning of key
  Slice key() const {
    // next is only equal to this if the LRU handle is the list head of an
    // empty list. List heads never have meaningful keys.
    assert(next != this);
    return Slice(key_data, key_length);
  }
};
```

这个LRUHandle这个类承载了很多功能，这大概是没有命名为LRUNode的原因：

1. 存储Key：Value数据对
2. LRU链表，按照顺序标记Least Recently Used
3. HashTable Bucket 的链表
4. 引用计数及清理

这是**HandleTable**：

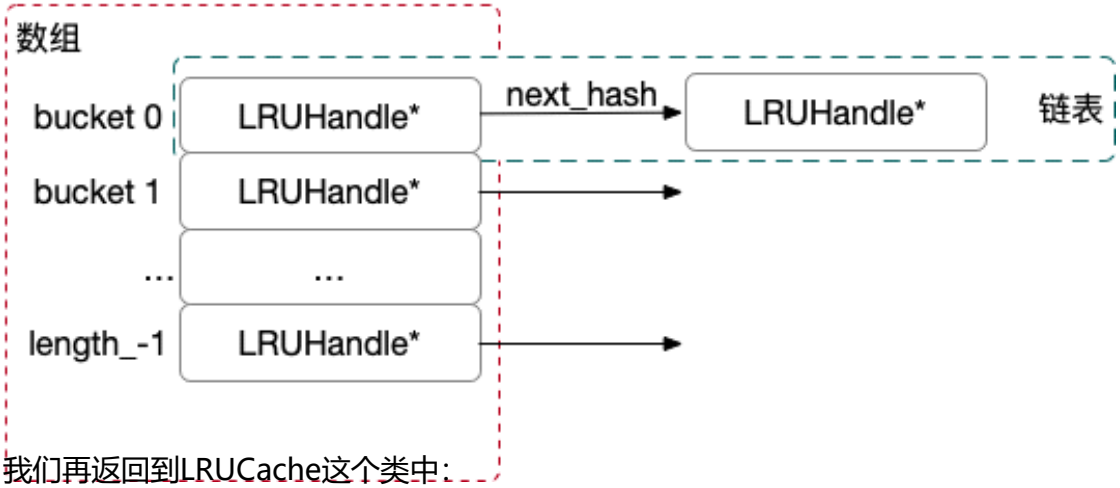
```
class HandleTable {
 public:
  //构造函数与析构函数
   构造函数中调用了该类中私有的Resize（）方法；析构函数中使用delete释放list_所占的内存
  HandleTable() : length_(0), elems_(0), list_(nullptr) { Resize(); }
  ~HandleTable() { delete[] list_; }
  LRUHandle* Lookup(const Slice& key, uint32_t hash) {
    return *FindPointer(key, hash);
  }
  LRUHandle* Insert(LRUHandle* h) {
```

```
    LRUHandle** ptr = FindPointer(h->key(), h->hash);
    LRUHandle* old = *ptr;
    h->next_hash = (old == nullptr ? nullptr : old->next_hash);
    *ptr = h;
    if (old == nullptr) {
      ++elems_;
      if (elems_ > length_) {
        // Since each cache entry is fairly large, we aim for a small
        // average linked list length (<= 1).
        Resize();
      }
    }
    return old;
  }
  LRUHandle* Remove(const Slice& key, uint32_t hash) {
    LRUHandle** ptr = FindPointer(key, hash);
    LRUHandle* result = *ptr;
    if (result != nullptr) {
      *ptr = result->next_hash;
      --elems_;
    }
    return result;
  }
 private:
  // The table consists of an array of buckets where each bucket is
  // a linked list of cache entries that hash into the bucket.
  uint32_t length_;       //哈希表的当前长度（即桶的数量），这个长度始终是2的幂，这样可以通过位运算实现快速求模操作
  uint32_t elems_;        //当前哈希表中存储的元素数量
  LRUHandle** list_;      //指向LRUHandle*类型数组的指针。每个元素是一个指向链表头的指针，链表中存储的是哈希值相同的缓存项
  // Return a pointer to slot that points to a cache entry that
  // matches key/hash.  If there is no such cache entry, return a
  // pointer to the trailing slot in the corresponding linked list.
  LRUHandle** FindPointer(const Slice& key, uint32_t hash) {
    LRUHandle** ptr = &list_[hash & (length_ - 1)];
    while (*ptr != nullptr && ((*ptr)->hash != hash || key != (*ptr)->key())) {
      ptr = &(*ptr)->next_hash;
    }
    return ptr;  //注意这个返回值，是指向LRUHandle*的指针，也就是指针的指针（哈希表中的结点并不是真的LRUHandle结点，哈希表中的结点存放的仅仅是指向真正结点（LRUHandle）的指针。
  }
```

//Resize（）方法是HandleTable类中用于动态调整哈希表大小的关键函数。在HandleTable中，哈希表是用来管理缓存项的，使用连地址法来处理哈希冲突。Resize（）方法的主要目的是在哈希表中的元素数量超过表的当前容量时，增大哈希表的大小，以保存查找、插入和删除操作的效率。

```
  void Resize() {
    uint32_t new_length = 4;      //计算新的哈希表长度，初始设置位4
    while (new_length < elems_) {
      new_length *= 2;            //新长度至少为现有长度的2倍
    }
```

```
    LRUHandle** new_list = new LRUHandle*[new_length];//分配一个新的数组用于新哈希表
    memset(new_list, 0, sizeof(new_list[0]) * new_length);//将新数组初始化为nullptr
    uint32_t count = 0;
    for (uint32_t i = 0; i < length_; i++) {
      LRUHandle* h = list_[i];
      while (h != nullptr) {
        LRUHandle* next = h->next_hash;
        uint32_t hash = h->hash;
        LRUHandle** ptr = &new_list[hash & (new_length - 1)];
        h->next_hash = *ptr;
        *ptr = h;
        h = next;
        count++;
      }
    }
    assert(elems_ == count);
    delete[] list_;
    list_ = new_list;
    length_ = new_length;
  }
};
```



我们再返回到LRUCache这个类中：

先来看他的构造函数：

```
LRUCache::LRUCache() : capacity_(0), usage_(0) {
  // Make empty circular linked lists.
  lru_.next = &lru_;
  lru_.prev = &lru_;
  in_use_.next = &in_use_;
  in_use_.prev = &in_use_;
}
```

这是这个类中的成员变量：

```
// Initialized before use.
size_t capacity_;
  // mutex_ protects the following state.
  mutable port::Mutex mutex_;
  size_t usage_ GUARDED_BY(mutex_);
  // Dummy head of LRU list.
  // lru.prev is newest entry, lru.next is oldest entry.
  // Entries have refs==1 and in_cache==true.
  LRUHandle lru_ GUARDED_BY(mutex_);
  // Dummy head of in-use list.
```

HandleTable即哈希表，levelDB里手动实现了一版，根据注释介绍相比g++原生的性能要高一些。

bucket桶的个数初始化大小为4，随着元素增加动态修改，使用数组实现，同一个bucket里面，**使用链表存储全部的LRUHandle***，最新的插入数据排在链表尾部。

核心函数是FindPointer。

注意在哈希表的每个bucket中存放的是一个链表，链表的每个结点不是LRUHandle，而是LRUHandle*，是指向LRUHandle的指针。

个人觉得这个地方理解还很模糊，后面再认真分析。

//Insert和Lookup等操作时都先加锁

//用于跟capacity_比较，判断是否超过容量

```
  // Entries are in use by clients, and have refs >= 2 and in_cache==true.
  LRUHandle in_use_ GUARDED_BY(mutex_);
  HandleTable table_ GUARDED_BY(mutex_);
```

Insert和**Lookup**这两个方法是主要的成员方法：

```
  Cache::Handle* Insert(const Slice& key, uint32_t hash, void* value,
                        size_t charge,
                        void (*deleter)(const Slice& key, void* value));
  Cache::Handle* Lookup(const Slice& key, uint32_t hash);
```

先来看**Lookup**过程：

这个过程比较简单，直接调用哈希表中的Lookup方法，返回一个指向对应LRUHandle结点的指针。除此之外还调用了引用计数Ref（e）

```
Cache::Handle* LRUCache::Lookup(const Slice& key, uint32_t hash) {
  MutexLock l(&mutex_);
  LRUHandle* e = table_.Lookup(key, hash);
  if (e != nullptr) {
    Ref(e);
  }
  return reinterpret_cast<Cache::Handle*>(e); // 已知e是一个指向LRUHandle的指针，那么这一句的使用强制类型转换转换成Cache: :Handle这是什么意思?
}
```

其中Ref()的过程如下：就像下面注释中写的那样，如果在lru_list中，就将其移动到in_use_表中。

```
void LRUCache::Ref(LRUHandle* e) {
  if (e->refs == 1 && e->in_cache) {  // If on lru_ list, move to in_use_ list.
    LRU_Remove(e);
    LRU_Append(&in_use_, e);
  }
  e->refs++;
}
```

总的来说，在Lookup过程中，成员变量table_则用于实现O(1)的查找，例如Lookup先查找table_，然后在更新lru_、in_use链表。

再来看**Insert**过程：

```
Cache::Handle* LRUCache::Insert(const Slice& key, uint32_t hash, void* value,
                                size_t charge,
                                void (*deleter)(const Slice& key,
                                                void* value)) {
  MutexLock l(&mutex_);
  //(1)首先申请动态大小的LRUHandle内存，初始化该结构体
  LRUHandle* e =
      reinterpret_cast<LRUHandle*>(malloc(sizeof(LRUHandle) - 1 + key.size()));
  e->value = value;
  e->deleter = deleter;
  e->charge = charge;
  e->key_length = key.size();
  e->hash = hash;
  e->in_cache = false;
  e->refs = 1;  // for the returned handle.
  std::memcpy(e->key_data, key.data(), key.size());
  //(2)如果存在可用容量缓存的，那么refs++成为2，e->cache的标志改为true象征着e指示的那个
```

结点就在cache中，然后调用LRU_Append(&in_use_,e)将结点加入到in_use_队列中。

```
  if (capacity_ > 0) {
    e->refs++;  // for the cache's reference.
    e->in_cache = true;
    LRU_Append(&in_use_, e);
    usage_ += charge;
```

// 接着插入到table_，如果key&&hash之前就存在，那么HandleTable：：Insert会返回原来的LRUHandle*对象指针，调用FinishErase（）清理进入状态1.

```
    FinishErase(table_.Insert(e));
  } else {  // don't cache. (capacity_==0 is supported and turns off caching.)
    // next is read by key() in an assert, so it must be initialized
    e->next = nullptr;
  }
```

（3）如果容量超限，执行淘汰策略

```
  while (usage_ > capacity_ && lru_.next != &lru_) {
    LRUHandle* old = lru_.next;  //lru_.next是最老的结点，首先被淘汰。
    assert(old->refs == 1);
    bool erased = FinishErase(table_.Remove(old->key(), old->hash()));
    if (!erased) {  // to avoid unused variable when compiled NDEBUG
      assert(erased);
    }
  }
  return reinterpret_cast<Cache::Handle*>(e);
}
```

在Insert过程中，总是会调用FinishErase函数进行处理：

具体定义如下：

```
// If e != nullptr, finish removing *e from the cache; it has already been
// removed from the hash table.  Return whether e != nullptr.
bool LRUCache::FinishErase(LRUHandle* e) {
  if (e != nullptr) {
    assert(e->in_cache);
    LRU_Remove(e);
    e->in_cache = false;
    usage_ -= e->charge;
    Unref(e);
  }
  return e != nullptr;
}
```

其中LRU_Remove和LRU_Append的过程如下所示：类似于双链表的插入和删除操作：

```
void LRUCache::LRU_Remove(LRUHandle* e) {
  e->next->prev = e->prev;
  e->prev->next = e->next;
}
void LRUCache::LRU_Append(LRUHandle* list, LRUHandle* e) {
  // Make "e" newest entry by inserting just before *list
  e->next = list;
  e->prev = list->prev;
  e->prev->next = e;
  e->next->prev = e;
}
```

在 Insert 函数中，先 LRU_Append 再 FinishErase 的操作顺序看似矛盾，实则合理：

- 先将新条目添加到 in_use_ 链表表示它将成为活跃条目。
- FinishErase 的调用是为了清理可能存在的旧条目，或者在超出容量限制时移除最老的条目。

- 先将新条目添加到 in_use_ 链表表示它将成为活跃条目。
- FinishErase 的调用是为了清理可能存在的旧条目，或者在超出容量限制时移除最老的条目。

为了保证refs值准确，无论Insert还是Lookup，都需要及时调用Release释放，使得节点能够进入lru_或者释放内存。
这种设计模式保证了缓存的一致性和有效性，确保不会出现重复键冲突，也不会超出缓存容量限制，是典型的LRU缓存管理方式。

```
void LRUCache::Release(Cache::Handle* handle) { MutexLock l(&mutex_);
Unref(reinterpret_cast<LRUHandle*>(handle)); }
```

最后我们可以返回到类SharedLRUCache中了，
LRUCache的接口都会加锁，为了更少的锁持有时间以及跟高的缓存命中率，可以定义多个LRUCache分别处理不同hash取模后的缓存处理。
SharedLRUCache就是这个作用，管理16个LRUCache，外部调用接口都委托调用具体某个LRUCache。

LevelDB笔记16：LRUCache的使用

2024年9月2日　　15:21

常规理解Cache缓存的是用户写入/读取的原始的key:value数据，但在LevelDB中并**不是**这样的，实际上，levelDB缓存了两种类型的数据：**Block**和**Table**。

Block

通过内存缓存高频访问data block，避免从磁盘读取，从而提高数据的访问效率。

从sstable的读取角度出发，参见**笔记14**。读取过程使用两级的迭代器最终定位到相应的data_block中。
在这个过程中调用了下面这个函数**BlockReader**：

```
// Convert an index iterator value (i.e., an encoded BlockHandle)
// into an iterator over the contents of the corresponding block.
Iterator* Table::BlockReader(void* arg, const ReadOptions& options,
                             const Slice& index_value) {
```

//index_value表示的是当前索引块的迭代器指向的那个条目（那个索引）的value（实际上就是对应数据块的offset和size编码后的value），在后面调用了handle.DecodeFrom()的过程实现了解析。

```
  Table* table = reinterpret_cast<Table*>(arg);
  Cache* block_cache = table->rep_->options.block_cache;
  Block* block = nullptr;
  Cache::Handle* cache_handle = nullptr;
  BlockHandle handle;
  Slice input = index_value;
  Status s = handle.DecodeFrom(&input);
```

//将data_block对应的offset和size从index_value中解析出来，意味着此时handle可以定位相应的data_block了

```
  // We intentionally allow extra stuff in index_value so that we
  // can add more features in the future.
  if (s.ok()) {
    BlockContents contents;
    if (block_cache != nullptr) {
      char cache_key_buffer[16];
      EncodeFixed64(cache_key_buffer, table->rep_->cache_id);
      EncodeFixed64(cache_key_buffer + 8, handle.offset());
      Slice key(cache_key_buffer, sizeof(cache_key_buffer));
      cache_handle = block_cache->Lookup(key);
      if (cache_handle != nullptr) {
        block = reinterpret_cast<Block*>(block_cache->Value(cache_handle));
```

```
      } else {
        s = ReadBlock(table->rep_->file, options, handle, &contents);
        if (s.ok()) {
          block = new Block(contents);
          if (contents.cachable && options.fill_cache) {
            cache_handle = block_cache->Insert(key, block, block->size(),
                                               &DeleteCachedBlock);
          }
        }
      }
    } else {  //block_cache == nullptr
      s = ReadBlock(table->rep_->file, options, handle, &contents);
      if (s.ok()) {
        block = new Block(contents);
      }
    }
  }
```

//直观上上面的作用便是从handle对应的位置（offset和size）中得到相应data_block的内容contents，用contents这个变量返回，该数据块的内容。之后调用Block的构造函数新建一个保存该内容的数据块。

在上面的过程中调用了LRUCache的相关内容，这个可以结合参见笔记15和16进行学习。

//新建完这个data_block后，调用Block->NewIterator()方法以返回一个data_block中的迭代器iter，可以结合笔记11Block的读取过程进行分析。

```
  Iterator* iter;
  if (block != nullptr) {
    iter = block->NewIterator(table->rep_->options.comparator);
    if (cache_handle == nullptr) {
      iter->RegisterCleanup(&DeleteBlock, block, nullptr);
    } else {
      iter->RegisterCleanup(&ReleaseBlock, block_cache, cache_handle);
    }
  } else {
    iter = NewErrorIterator(s);
  }
  return iter;
}
```

重点来看上面加粗的部分：

整体逻辑就是

1. 如果有缓存那么尝试着从缓存中读取
   a. 如果缓存中读不到，从文件中读取，并将该块数据插入到缓存中
   b. 如果缓存中读到，只需如下所示用block来记录相应块的指针。

```
block = reinterpret_cast<Block*>(block_cache->Value(cache_handle));
```

2. 如果没缓存直接从文件中读取

是否存在缓存是根据**block_cache**这个变量是否为空来判断的；

```
Cache* block_cache = table->rep_->options.block_cache;
```

如果有缓存就是这个block_cache!=nullptr的时候：

```
// cache key = (cache_id + offset)
// cache_id不同Table间保证唯一
// 同一Table中不同的data block有唯一的offset
// 因此可以作为cache key
 char cache_key_buffer[16];
 EncodeFixed64(cache_key_buffer, table->rep_->cache_id);
 EncodeFixed64(cache_key_buffer + 8, handle.offset());
 Slice key(cache_key_buffer, sizeof(cache_key_buffer));
 cache_handle = block_cache->Lookup(key);
 if (cache_handle != nullptr) {
   block = reinterpret_cast<Block*>(block_cache->Value(cache_handle));
```

缓存key使用cache_id+offset来唯一表示一个Block，

上面首先调用SharedLRUCache：：Lookup（key）来查找缓存中是否存在key对应的那个LRUHandle，在其内部实际上调用了LRUCache：：Lookup（key，hash）函数。如下所示：

```
Handle* Lookup(const Slice& key) override {
  const uint32_t hash = HashSlice(key);
  return shard_[Shard(hash)].Lookup(key, hash);
}
```

那么这样就自然返回到**笔记15**中讲述的内容了。

```
Cache::Handle* LRUCache::Lookup(const Slice& key, uint32_t hash) {
  MutexLock l(&mutex_);
  LRUHandle* e = table_.Lookup(key, hash);
  if (e != nullptr) {
    Ref(e);
  }
  return reinterpret_cast<Cache::Handle*>(e);
}
```

如果在缓存中存在该key，那么返回给cache_handle的实际上是一个在哈希表LRUCache:: table_中相应LRUHandle的指针。当他非空的时候，调用block_cache->Value(cache_handle)

定义如下：

```
void* Value(Handle* handle) override {
  return reinterpret_cast<LRUHandle*>(handle)->value;
}
```

完整的是：

```
block = reinterpret_cast<Block*>(block_cache->Value(cache_handle));
```

从这里可以看到并验证了这里的**数据缓存是以block为单位的**，并不是以单个的key：value对，一定要注意这里的value并不是真正数据，而是一个指针，用强制类型转换成Block*的类型，最终指向的是一个data_block。

如果cache_handel为nullptr，意思就是缓存中没有找到这个相应的data_block，那么：

```
else {
      s = ReadBlock(table->rep_->file, options, handle, &contents);
      if (s.ok()) {
        block = new Block(contents);
        if (contents.cachable && options.fill_cache) {
          cache_handle = block_cache->Insert(key, block, block->size(),
                                             &DeleteCachedBlock);
        }
      }
    }
```

发现Insert的charge参数是恰好是整个block的大小，因此能够限制内存使用量。

BlockReader返回的是一个迭代器，随着迭代器的销毁，cache:: Handle 会Release：

```
iter->RegisterCleanup(&ReleaseBlock, block_cache, cache_handle);
```

而随着数据的淘汰，会调用传入的deleter函数销毁Block*

```
static void DeleteCachedBlock(const Slice& key, void* value) {
  Block* block = reinterpret_cast<Block*>(value);
  delete block;
}
```

读文件的时候调用的是下面这个**ReadBlock**（）的过程：

```
s = ReadBlock(table->rep_->file, options, handle, &contents);
```

首先第一个参数是file如下所示

```
  RandomAccessFile* file;
```

handle是BlockHandle的类型只是对应block的size和offset

来看ReadBlock这个过程

```
Status ReadBlock(RandomAccessFile* file, const ReadOptions& options,
                 const BlockHandle& handle, BlockContents* result) {
```

（1）初始化result

```
  result->data = Slice();
  result->cachable = false;
  result->heap_allocated = false;
  // Read the block contents as well as the type/crc footer.
  // See table_builder.cc for the code that built this structure.
```

（2）读取数据块和校验

```
  size_t n = static_cast<size_t>(handle.size());
  char* buf = new char[n + kBlockTrailerSize]; //存储到sstable上的data_block尾部添加了4B的crc和1B的压缩类型
  Slice contents;
  Status s = file->Read(handle.offset(), n + kBlockTrailerSize, &contents, buf);
  if (!s.ok()) {
    delete[] buf;
    return s;
  }

  if (contents.size() != n + kBlockTrailerSize) {//校验读取的大小
    delete[] buf;
    return Status::Corruption("truncated block read");
  }
  // Check the crc of the type and the block contents
```

（3）crc校验数据块的正确性

```
  const char* data = contents.data();  // Pointer to where Read put the data
  if (options.verify_checksums) {
    const uint32_t crc = crc32c::Unmask(DecodeFixed32(data + n + 1));
    const uint32_t actual = crc32c::Value(data, n + 1);
    if (actual != crc) {
      delete[] buf;
      s = Status::Corruption("block checksum mismatch");
      return s;
```

```
  }
}
```

（4）解压缩或者直接使用数据：最终数据存储在results中以返回

```
switch (data[n]) {
  case kNoCompression:
    if (data != buf) {
      // File implementation gave us pointer to some other data.
      // Use it directly under the assumption that it will be live
      // while the file is open.
      delete[] buf;
      result->data = Slice(data, n);
      result->heap_allocated = false;
      result->cachable = false;  // Do not double-cache
    } else {
      result->data = Slice(buf, n);
      result->heap_allocated = true;
      result->cachable = true;
    }
    // Ok
    break;
  case kSnappyCompression: {
    size_t ulength = 0;
    if (!port::Snappy_GetUncompressedLength(data, n, &ulength)) {
      delete[] buf;
      return Status::Corruption("corrupted snappy compressed block length");
    }
    char* ubuf = new char[ulength];
    if (!port::Snappy_Uncompress(data, n, ubuf)) {
      delete[] buf;
      delete[] ubuf;
      return Status::Corruption("corrupted snappy compressed block contents");
    }
    delete[] buf;
    result->data = Slice(ubuf, ulength);
    result->heap_allocated = true;
    result->cachable = true;
    break;
  }
  case kZstdCompression: {
    size_t ulength = 0;
    if (!port::Zstd_GetUncompressedLength(data, n, &ulength)) {
      delete[] buf;
      return Status::Corruption("corrupted zstd compressed block length");
    }
    char* ubuf = new char[ulength];
    if (!port::Zstd_Uncompress(data, n, ubuf)) {
      delete[] buf;
      delete[] ubuf;
      return Status::Corruption("corrupted zstd compressed block contents");
    }
    delete[] buf;
    result->data = Slice(ubuf, ulength);
    result->heap_allocated = true;
    result->cachable = true;
    break;
  }
```

```
    default:
      delete[] buf;
      return Status::Corruption("bad block type");
  }
  return Status::OK();
```

Table

通过内存缓存高频打开的文件句柄及部分内容，避免从磁盘读取，从而提高sstable文件的访问速率，同时控制打开的文件句柄数目。

这个函数中的核心功能就是从文件中读取指定数据块，并根据块的压缩类型进行处理。

初始化：

从文件中读取相应块主要是调用了：

```
  table_cache_ (new TableCache(dbname_, options_, TableCacheSize(options_))),
  Status s = file->Read(handle.offset(), n + kBlockTrailerSize, &contents, buf);
```

在DBImpl类中的构造函数列表里进行初始化table_cache

那么这个函数是啥样的呢？以及file这个类型是啥样的呢？-这部分后续再分析

Table_cache变量的定义如下，他的类型为TableCache*

```
  // table_cache_ provides its own synchronization
  TableCache* const table_cache_;
```

那么**TableCache**是什么呢？其定义如下：

```
class TableCache {
 public:
  TableCache(const std::string& dbname, const Options& options, int entries);
  TableCache(const TableCache&) = delete;
  TableCache& operator=(const TableCache&) = delete;
  ~TableCache();
  // Return an iterator for the specified file number (the corresponding
  // file length must be exactly "file_size" bytes).  If "tableptr" is
  // non-null, also sets "*tableptr" to point to the Table object
  // underlying the returned iterator, or to nullptr if no Table object
  // underlies the returned iterator.  The returned "*tableptr" object is owned
  // by the cache and should not be deleted, and is valid for as long as the
  // returned iterator is live.
  Iterator* NewIterator(const ReadOptions& options, uint64_t file_number,
                        uint64_t file_size, Table** tableptr = nullptr);
  // If a seek to internal key "k" in specified file finds an entry,
  // call (*handle_result)(arg, found_key, found_value).
  Status Get(const ReadOptions& options, uint64_t file_number,
             uint64_t file_size, const Slice& k, void* arg,
             void (*handle_result)(void*, const Slice&, const Slice&));
  // Evict any entry for the specified file number
  void Evict(uint64_t file_number);
 private:
  Status FindTable(uint64_t file_number, uint64_t file_size, Cache::Handle**);
  Env* const env_;
  const std::string dbname_;
  const Options& options_;
  Cache* cache_;
};
```

返回到sstable读取的笔记15中。

从Open函数入手，它用于打开sstable文件，并初始化一个Table*的对象。

```
Status Table::Open(const Options& options, RandomAccessFile* file,
                   uint64_t size, Table** table) {
  //1 初始化table指针
```

```
  *table = nullptr;
  //2 文件大小检查
  if (size < Footer::kEncodedLength) {
    //kEncodeLength=48，代表着Footer结构的定长大小48B
    return Status::Corruption("file is too short to be an sstable");
  }
  //3 读取Footer
//这个地方可以与笔记13中的Footer相关内容进行结合学习，
  char footer_space[Footer::kEncodedLength];
  Slice footer_input;
  Status s = file->Read(size - Footer::kEncodedLength, Footer::kEncodedLength,
                        &footer_input, footer_space);  //从文件末尾读取48字节到
footer_space中，并将其包装到Slice对象footer_input中。
  if (!s.ok()) return s;
  Footer footer;
  s = footer.DecodeFrom(&footer_input); //解码读出的内容至footer对象中，Footer的两个成员
变量metaindex_handle_和index_handle_;他们都是BlockHandle类型的，用于指示过滤块和索引块的
索引信息。
  if (!s.ok()) return s;

  //4 读取索引块
  // Read the index block
  BlockContents index_block_contents;
  ReadOptions opt;
  if (options.paranoid_checks) {
    opt.verify_checksums = true;
  }  //如果启用了严格检查，则验证校验和
  s = ReadBlock(file, opt, footer.index_handle(), &index_block_contents); //调用
ReadBlock函数，读取由footer.index_handle()指定的索引块。用index_block_contents加以记录并
返回。

  //5 构建Table对象
  if (s.ok()) {
    // We've successfully read the footer and the index block: we're
    // ready to serve requests.
    Block* index_block = new Block(index_block_contents); //创建一个新的Block对象来封装
索引块内容。
    Rep* rep = new Table::Rep;
    rep->options = options;
    rep->file = file;
    rep->metaindex_handle = footer.metaindex_handle();
    rep->index_block = index_block;
    rep->cache_id = (options.block_cache ? options.block_cache->NewId() : 0);
    rep->filter_data = nullptr;
    rep->filter = nullptr;
    *table = new Table(rep);  创建一个新的Table对象，并将其指针赋值给table
    (*table)->ReadMeta(footer); //调用ReadMeta方法，读取元数据块的信息。
  }
  return s;
}
```

在Table中的构造函数是私有类型，它在Open的时候被调用，实例化一个新的Table对象

```
  explicit Table(Rep* rep) : rep_(rep) {}
```

在初始化Table类中的私有成员变量rep_后，接着调用了(*table)->ReadMeta(footer); ReadMeta方法是Table类中的一个私有的成员方法，其定义如下：

```
void Table::ReadMeta(const Footer& footer) {
  //1 检查过滤策略是否为空（暗示这个块是个空块，直接返回即可。）
  if (rep_->options.filter_policy == nullptr) {
    return;  // Do not need any metadata
  }
  // TODO(sanjay): Skip this if footer.metaindex_handle() size indicates
  // it is an empty block.
  // 2 设置读取选项，如果数据库选项中开启了paranoid_checks，则启用校验和验证(verify_checksums = true)，用于更严格的检查数据读取过程中的完整性。
  ReadOptions opt;
  if (rep_->options.paranoid_checks) {
    opt.verify_checksums = true;
  }
  // /3读取meta_index_block中的内容，
  BlockContents contents;
  if (!ReadBlock(rep_->file, opt, footer.metaindex_handle(), &contents).ok()) {
    // Do not propagate errors since meta info is not needed for operation
    return;
  }
  //4 遍历meta_index_block
  Block* meta = new Block(contents);
  //通过NewIterator函数，使用字节比较器（‘BytewiseComparator’）创建一个用于遍历该块内容的迭代器iter。
  Iterator* iter = meta->NewIterator(BytewiseComparator());
  //5 构建一个key（如filter. BloomFilter）这样的key之后，调用iter->Seek(key)在这个meta_index_block中查找与这个key对应的过滤器数据位置。
  std::string key = "filter.";
  key.append(rep_->options.filter_policy->Name());
  iter->Seek(key);
  //6 读取过滤器
  if (iter->Valid() && iter->key() == Slice(key)) {
    ReadFilter(iter->value());
  }
  delete iter;
  delete meta;
}
```

那么来看看ReadFilter这个过程：

```
void Table::ReadFilter(const Slice& filter_handle_value) {
  Slice v = filter_handle_value;
  BlockHandle filter_handle;
  if (!filter_handle.DecodeFrom(&v).ok()) {
    return;
  }
  // We might want to unify with ReadBlock() if we start
```

```
  // requiring checksum verification in Table::Open.
  ReadOptions opt;
  if (rep_->options.paranoid_checks) {
    opt.verify_checksums = true;
  }
  BlockContents block;
  if (!ReadBlock(rep_->file, opt, filter_handle, &block).ok()) {
    return;
  }
  if (block.heap_allocated) {
    rep_->filter_data = block.data.data();  // Will need to delete later
  }
  rep_->filter = new FilterBlockReader(rep_->options.filter_policy, block.data);
}
```

参数filter_handle_value这个句柄包含了filter_block在文件中的offset和size信息，然后首先调用DecodeFrom来对这个句柄解码，之后调用ReadBlock对这个filter_handle指示的那个filter_block进行读取，用BlockContents 类型的对象block进行返回读取到的内容。

最后实现了：将相应过滤器块的信息存储在Table的成员rep中。

rep_->filter_data: 存储了指向过滤器数据的指针，这些数据可能是从堆上分配的，将来需要手动释放。

rep_->filter: 创建了一个新的 FilterBlockReader 对象，处理过滤器块的逻辑，用于执行后续的查询操作时，快速判断某些键是否存在。

Table怎么缓存的好像还不是很清晰leveldb笔记之12:LRUCache的使用 - Ying (izualzhy.cn)这是人家写的笔记，**后续还需要再好好理一理。**

LevelDB笔记17：minor compaction

2024年9月4日　　14:58

可以先看笔记9：Env

事实上到最后调用的是**BackgroundCompaction（）**这个过程

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

  //(1)Memtable压缩
  //如果是不可变的Memtable，调用CompactMemtable（）进行压缩，即minor Compaction的过程
  if (imm_ != nullptr) {
    CompactMemTable();
    return;
  }

  //(2)文件压缩
  Compaction* c;
  bool is_manual = (manual_compaction_ != nullptr);
  InternalKey manual_end;
  //手动压缩
  if (is_manual) {
    ManualCompaction* m = manual_compaction_;
    c = versions_->CompactRange(m->level, m->begin, m->end);
    m->done = (c == nullptr);
    if (c != nullptr) {
      manual_end = c->input(0, c->num_input_files(0) - 1)->largest;
    }
    Log(options_.info_log,
        "Manual compaction at level-%d from %s .. %s; will stop at %s\n",
        m->level, (m->begin ? m->begin->DebugString().c_str() : "(begin)"),
        (m->end ? m->end->DebugString().c_str() : "(end)"),
        (m->done ? "(end)" : manual_end.DebugString().c_str()));
  } else {
  //自动压缩
    c = versions_->PickCompaction();
  }


  Status status;
  if (c == nullptr) {
    // Nothing to do
  } else if (!is_manual && c->IsTrivialMove()) {
```

```
    // Move file to next level
    assert(c->num_input_files(0) == 1);
    FileMetaData* f = c->input(0, 0);
    c->edit()->RemoveFile(c->level(), f->number);
    c->edit()->AddFile(c->level() + 1, f->number, f->file_size, f->smallest,
                       f->largest);
    status = versions_->LogAndApply(c->edit(), &mutex_);
    if (!status.ok()) {
      RecordBackgroundError(status);
    }
    VersionSet::LevelSummaryStorage tmp;
    Log(options_.info_log, "Moved #%lld to level-%d %lld bytes %s: %s\n",
        static_cast<unsigned long long>(f->number), c->level() + 1,
        static_cast<unsigned long long>(f->file_size),
        status.ToString().c_str(), versions_->LevelSummary(&tmp));
  } else {
    CompactionState* compact = new CompactionState(c);
    status = DoCompactionWork(compact);
    if (!status.ok()) {
      RecordBackgroundError(status);
    }
    CleanupCompaction(compact);
    c->ReleaseInputs();
    RemoveObsoleteFiles();
  }
  delete c;
```

//压缩结果处理，一般是错误处理

```
  if (status.ok()) {
    // Done
  } else if (shutting_down_.load(std::memory_order_acquire)) {
    // Ignore compaction errors found during shutting down
  } else {
    Log(options_.info_log, "Compaction error: %s", status.ToString().c_str());
  }
```

//手动压缩结束处理

```
  if (is_manual) {
    ManualCompaction* m = manual_compaction_;
    if (!status.ok()) {
      m->done = true;
    }
    if (!m->done) {
      // We only compacted part of the requested range.  Update *m
      // to the range that is left to be compacted.
      m->tmp_storage = manual_end;
      m->begin = &m->tmp_storage;
    }
    manual_compaction_ = nullptr;
  }
}
```

compaction的过程分为minor 和major compaction两个过程。首先来看前者：

Minor Compaction这个过程实际上就对应Memtable持久化为sstable的过程。

源码入口就在上面这个函数中，如果immutable memtable存在，首先进行Minor Compaction，调用了**CompactMemTable（）**这个过程。也在DBImpl这个大类中：

```
void DBImpl::CompactMemTable() {
  //(1)锁的断言与前置条件
  mutex_.AssertHeld();  //确保在调用CompactMemTable（）时当前线程已经持有互斥锁mutex_，防止多线程竞争访问共享数据
  assert(imm_ != nullptr); //确保存在需要压缩和持久化的内存表即immutable
  // Save the contents of the memtable as a new Table
  //(2) 创建VersionEdit和获取当前版本，那么LevelDB中的Version是什么？？？
  VersionEdit edit;
  Version* base = versions_->current();
  base->Ref();
  //(3) 写入sstable
  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");
  }
  //(4) 将本次文件（edit）更新信息version_
  // 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_);
  }
  //(5) 处理压缩完成后的状态，删除无用文件
  if (s.ok()) {
    // Commit to the new state
    imm_->Unref();
    imm_ = nullptr;
    has_imm_.store(false, std::memory_order_release);//通过原子操作将has_imm_设置为false，通知其他线程当前没有不可变的memtable。
    RemoveObsoleteFiles();
  } else {
    RecordBackgroundError(s);
  }
}
```

直观上，**CompactMemTable**主要流程分为三个部分：

1. WriteLevel0Table(imm_, &edit, base)：imm_落盘成为新的 sst 文件，文件信息记录到 edit
2. versions_->LogAndApply(&edit, &mutex_)：将本次文件更新信息versions_，当前的文件（包含新的 sst 文件）作为数据库的一个最新状态，后续读写都会基于该状态
3. ~~DeleteObsoleeteFiles：删除一些无用文件~~

先来看第一个：**WriteLevel0Table**过程：

先来看第一个：**WriteLevel0Table**过程：

```
Status DBImpl::WriteLevel0Table(MemTable* mem, VersionEdit* edit,
                                Version* base) {
  mutex_.AssertHeld(); //锁定断言，确保当前线程持有互斥锁（mutex_）
  const uint64_t start_micros = env_->NowMicros();//获取当前时间（微妙级别），
用于测量写操作的持续时间，后续用于性能统计。
  FileMetaData meta;
  meta.number = versions_->NewFileNumber(); //顺序生成sstable的编号，用于文件
名
  pending_outputs_.insert(meta.number);
  Iterator* iter = mem->NewIterator();
  Log(options_.info_log, "Level-0 table #%llu: started",
      (unsigned long long)meta.number);     //日志记录，表明SSTable的创建过程开
始。
  Status s;
  {
    mutex_.Unlock(); //释放mutex_锁，允许其他线程进行并发操作
    //更新memtable中全部数据到xxx.ldb文件中
    //meta记录key range，file_size等sst信息
    s = BuildTable(dbname_, env_, options_, table_cache_, iter, &meta);
   //iter用过遍历MemTable，通过BuildTable将数据写入到sstable，该函数实际上
就是调用了TableBuilder
    mutex_.Lock(); //完成写入后再次获得锁，以便后续操作可以安全的修改共享数据结
构
  }
  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; //释放分配给迭代器的内存，防止内存泄漏
  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.
  // 如果file_size为0，表示文件已被删除，此时不应该将其添加到清单中
  //整个if语句实现了版本管理：确定sstable的级别，然后更新VersionEdit
  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);
  }
 // 收集Compaction统计信息
  CompactionStats stats; //用于保存SSTable写入操作的统计信息。
  stats.micros = env_->NowMicros() - start_micros;//计算写入sstable花费的时间
  stats.bytes_written = meta.file_size; // 记录写入的sstable大小（字节）
```

```
  stats.micros = env_->NowMicros() - start_micros;//计算写入sstable花费的时间
  stats.bytes_written = meta.file_size; //记录写入的sstable大小（字节）
  stats_[level].Add(stats); //将统计信息汇总到当前对应级别的compaction统计中，
这些数据可用于监控、调整和优化数据库性能。
  return s;
}
```

上面这个过程中：

```
  Iterator* iter = mem->NewIterator();
```

首先新建了一个MemTableIterator的的迭代器，实际上这个MemTableIterator封装了跳表中的迭代器。参见**笔记4.**

```
Iterator* MemTable::NewIterator() { return new MemTableIterator(&table_); }
```

iter就是一个指向MemTableIterator类型对象的指针。

其中**BuildTable ()** 是核心的一个过程，其源码在builder.cc中，如下所示：

```
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());
    Slice key;
    for (; iter->Valid(); iter->Next()) {
      key = iter->key();
      builder->Add(key, iter->value());  //(1)
    }
    if (!key.empty()) {
      meta->largest.DecodeFrom(key);
    }
    // Finish and check for builder errors
    s = builder->Finish();        //(2)
    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_->RemoveFile(fname);
  }
  return s;
}
```

这就是一个完整的 minor compaction过程，通过 minor compaction，内存里的 memtable 源源不断的转化为磁盘上的 sst 文件，正如前面强调的这一过程最重要的原则，就是高性能的写转化。因此并没有考虑不同文件间数据的重复和顺序，当读取数据时，总是需要读取 level 0 所有的文件，因此对读并不友好。major compaction就是为了解决这一问题，不断的把上层的 sst 文件通过二路归并转化为下层的 sst 文件。

从 compact 里，就开始引入 leveldb 里版本的概念了，minor compaction 过程相对简单一些，因此，接下来一篇笔记，开始从上面提到的版本的各个接口，介绍 leveldb 里的版本的设计思想。

LevelDB笔记18：Version

2024年9月9日 11:03

接笔记17，minor compaction作用是将immtable中的内容持久化到level 0的sstable中，这个过程主要调用的是DBImpl : :CompactMemtable()这个过程，而在其中主要调用了DBImpl : :WriteLevel0Table()这个过程。

```
//(2) 创建VersionEdit和获取当前版本，那么LevelDB中的Version是什么？？？
VersionEdit edit;
Version* base = versions_->current();
base->Ref();
//(3) 写入sstable
Status s = WriteLevel0Table(imm_, &edit, base);
```

1. 为什么要有版本管理

compaction简言之，是新增与删除文件的过程，例如minor compaction是新增一个文件，而对于major compaction，则是归并N个文件到M个新文件，这N+M个历史文件与新文件，共同存储在磁盘上，因此需要一个文件管理系统，能够识别出哪些是当前的sstable files，哪些属于历史文件。

所以版本管理的作用之一，就是**记录compaction之后，数据库由哪些文件组成。**

compaction是leveldb单独的线程，当我们读取某个ssable文件时，可能该文件正在compact，也就是作为N个历史文件之一。那这个文件虽然不在新的version里，但是也不能删除，该文件属于之前的某个version。

所以版本管理的作用之二，就是**记录文件属于哪个version。**

一句话，就是**Version（版本）管理负责管理磁盘上的文件，以保证levelDB数据的准确性。**

2. 入手分析：Version的关键结构

我们先感官上分析一下，接着minor compaction的地方：

```
VersionEdit edit;
Version* base = versions_->current();
base->Ref();
//写入sstable
Status s = WriteLevel0Table(imm_, &edit, base);
```

base是Version*类型的指针，也就是说versions_->current()的过程返回一个指向Version对象的指针。Versions_是类DBImpl中的一个私有成员变量：

```
VersionSet* const versions_ GUARDED_BY(mutex_);
```

Note: C++中的const相关内容见笔记C++: Const

但是这个私有成员是VersionSet*类型的！！！

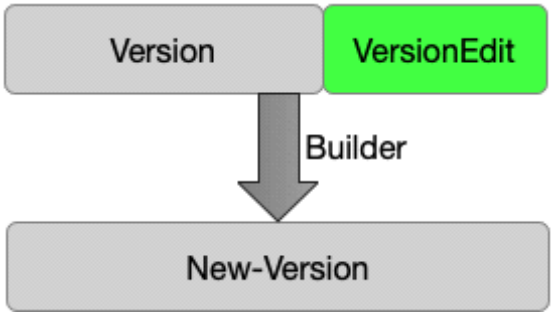
那么，从这几句代码基本上可以得到levelDB中**设计Version的主要三个结构（三个大**

类）：VersionEdit、Version和VersionSet。

2.1 delta

每次compaction都是新增与删除文件，在原来文件版本的基础上，生成一个新的版本，也就是：**Version + Delta = New-Version**

在levelDB具体实现中，负责管理Delta的类是VersionEdit，某个版本使用Version记录。



Operator + =则由类Builder实现。

2.2 链表

New-Version生成后，仅在Version下存在的文件并不一定会被立刻删除，例如有的文件还在被读取，或者程序退出了没有来得及删除。

因此多个version会同时存在，之间是链表的关系，当某个version彻底没有使用后，其独有的文件才能被删除，同时从链表里删除该version。实现这个功能的类是VersionSet。

2.3 从minor compaction入手简单分析

```
  VersionEdit edit;
  Version* base = versions_->current();
```

调用了Versionset类中的成员函数current()方法。

```
  Version* current() const { return current_; }
```

其中current_是Versionset类中的成员变量：

```
  Version* current_;        // == dummy_versions_.prev_
```

也就是说Versionset实现了对当前version的管理，current返回当前的version版本。

写入

```
  Status s = WriteLevel0Table(imm_, &edit, base);
```

再次返回到笔记17中来看WriteLevel0Table的过程：下面只挑选部分关键语句进行叙述：

```
Status DBImpl::WriteLevel0Table(MemTable* mem, VersionEdit* edit,
                                Version* base) {
  FileMetaData meta;
  meta.number = versions_->NewFileNumber(); //顺序生成sstable的编号，用于文件名
    // (1) 将内存中的数据转换为磁盘上的sstable
    s = BuildTable(dbname_, env_, options_, table_cache_, iter, &meta);
    //iter用过遍历MemTable，通过BuildTable将数据写入到sstable，该函数实际上就是调用了TableBuilder，&meta是指向FileMetaData的指针，用于存储新SSTable的元数据。
    // (2) 确定SSTable的适当级别，然后更新VersionEdit
    if (s.ok() && meta.file_size > 0) {
    const Slice min_user_key = meta.smallest.user_key();
```

```
    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);
  }
......
```

此外，在笔记17中已经介绍了DBImpl::CompactMemTable()这个中调用了：

```
WriteLevel0Table(MemTable* mem, VersionEdit* edit,
                                Version* base)
```

可以接着看CompactMemTable后续的步骤含有比如说

```
    s = versions_->LogAndApply(&edit, &mutex_);
```

那么这样，基本上对VersionSet、VersionEdit和Version有一个大致的认识了。
下面我们根据这几个结构的源码实现做深入理解。

3. VersionEdit

这个VersionEdit的结构声明在version_edit.h的头文件中，此外该头文件中还包含了对FileMetaData结构的定义。上面所讲的meta就是该类型的对象实例。

3.1 FileMetaData

```
struct FileMetaData {
  FileMetaData() : refs(0), allowed_seeks(1 << 30), file_size(0) {}
  int refs;
  int allowed_seeks;  // Seeks allowed until compaction
  uint64_t number;
  uint64_t file_size;    // File size in bytes
  InternalKey smallest;  // Smallest internal key served by table
  InternalKey largest;   // Largest internal key served by table
};
```

包含了文件号，文件大小，和最大和最小的InternalKey，注意在之前确定级别的时候是通过比如说 meta.smallest.user_key() 解析出了user_key，之后根据user_key进行调用level=base->PickLevelForMemTableOutput(min_user_key, max_user_key)过程的。

3.2 VersionEdit

```
class VersionEdit {
 public:
  VersionEdit() { Clear(); }
  ~VersionEdit() = default;
  void Clear();
  void SetComparatorName(const Slice& name) {
    has_comparator_ = true;
    comparator_ = name.ToString();
  }
  void SetLogNumber(uint64_t num) {
    has_log_number_ = true;
```

```
    log_number_ = num;
  }
  void SetPrevLogNumber(uint64_t num) {
    has_prev_log_number_ = true;
    prev_log_number_ = num;
  }
  void SetNextFile(uint64_t num) {
    has_next_file_number_ = true;
    next_file_number_ = num;
  }
  void SetLastSequence(SequenceNumber seq) {
    has_last_sequence_ = true;
    last_sequence_ = seq;
  }
  void SetCompactPointer(int level, const InternalKey& key) {
    compact_pointers_.push_back(std::make_pair(level, key));
  }
  // Add the specified file at the specified number.
  // REQUIRES: This version has not been saved (see VersionSet::SaveTo)
  // REQUIRES: "smallest" and "largest" are smallest and largest keys in file
  void AddFile(int level, uint64_t file, uint64_t file_size,
               const InternalKey& smallest, const InternalKey& largest) {
    FileMetaData f;
    f.number = file;
    f.file_size = file_size;
    f.smallest = smallest;
    f.largest = largest;
    new_files_.push_back(std::make_pair(level, f));
  }
  // Delete the specified "file" from the specified "level".
  void RemoveFile(int level, uint64_t file) {
    deleted_files_.insert(std::make_pair(level, file));
  }
  void EncodeTo(std::string* dst) const;
  Status DecodeFrom(const Slice& src);
  std::string DebugString() const;

 private:
  friend class VersionSet;
  typedef std::set<std::pair<int, uint64_t>> DeletedFileSet;
  std::string comparator_;
  uint64_t log_number_;
  uint64_t prev_log_number_;
  uint64_t next_file_number_;
  SequenceNumber last_sequence_;
  bool has_comparator_;
  bool has_log_number_;
  bool has_prev_log_number_;
  bool has_next_file_number_;
  bool has_last_sequence_;
  std::vector<std::pair<int, InternalKey>> compact_pointers_;

  DeletedFileSet deleted_files_;
  std::vector<std::pair<int, FileMetaData>> new_files_;
};
```

VersionEdit即delta，最重要的两个成员变量就是新增与删除文件，即上面的**deleted_files**和

new_files_。对外接口上，**AddFile()**就是更新了new_files_，记录**{level，FileMetaData}**对到new_files_中。见上面的AddFile过程。

4. VersionSet

该结构声明在version_set.h的头文件中。

具体如下：

```
class VersionSet {
 public:
  VersionSet(const std::string& dbname, const Options* options,
             TableCache* table_cache, const InternalKeyComparator*);
  VersionSet(const VersionSet&) = delete;
  VersionSet& operator=(const VersionSet&) = delete;
  ~VersionSet();
  // Apply *edit to the current version to form a new descriptor that
  // is both saved to persistent state and installed as the new
  // current version.  Will release *mu while actually writing to the file.
  // REQUIRES: *mu is held on entry.
  // REQUIRES: no other thread concurrently calls LogAndApply()
  Status LogAndApply(VersionEdit* edit, port::Mutex* mu)
      EXCLUSIVE_LOCKS_REQUIRED(mu);
  // Recover the last saved descriptor from persistent storage.
  Status Recover(bool* save_manifest);
  // Return the current version.
  Version* current() const { return current_; }
  // Return the current manifest file number
  uint64_t ManifestFileNumber() const { return manifest_file_number_; }
  // Allocate and return a new file number
  uint64_t NewFileNumber() { return next_file_number_++; }
  // Arrange to reuse "file_number" unless a newer file number has
  // already been allocated.
  // REQUIRES: "file_number" was returned by a call to NewFileNumber().
  void ReuseFileNumber(uint64_t file_number) {
    if (next_file_number_ == file_number + 1) {
      next_file_number_ = file_number;
    }
  }
  // Return the number of Table files at the specified level.
  int NumLevelFiles(int level) const;
  // Return the combined file size of all files at the specified level.
  int64_t NumLevelBytes(int level) const;
  // Return the last sequence number.
  uint64_t LastSequence() const { return last_sequence_; }
  // Set the last sequence number to s.
  void SetLastSequence(uint64_t s) {
    assert(s >= last_sequence_);
    last_sequence_ = s;
  }
  // Mark the specified file number as used.
  void MarkFileNumberUsed(uint64_t number);
  // Return the current log file number.
  uint64_t LogNumber() const { return log_number_; }
  // Return the log file number for the log file that is currently
  // being compacted, or zero if there is no such log file.
  uint64_t PrevLogNumber() const { return prev_log_number_; }
```

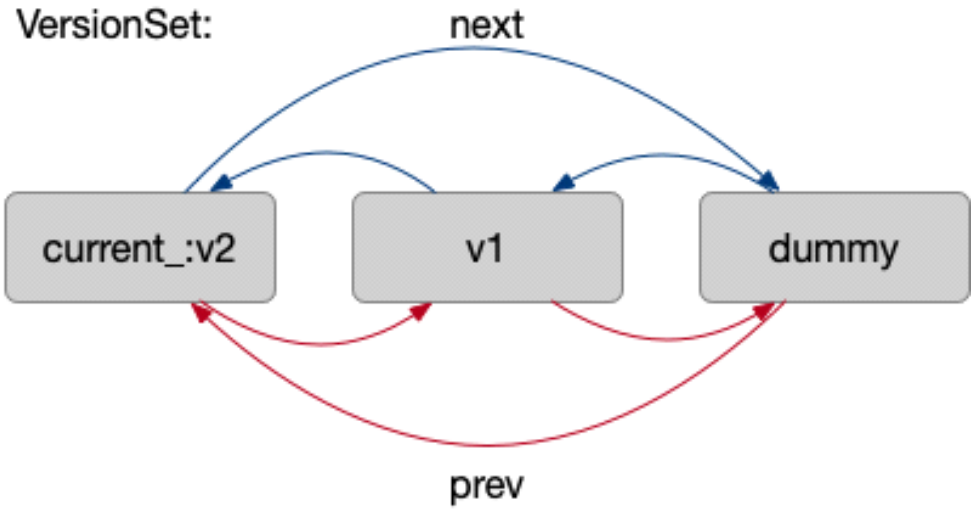
```
  // Pick level and inputs for a new compaction.
  // Returns nullptr if there is no compaction to be done.
  // Otherwise returns a pointer to a heap-allocated object that
  // describes the compaction.  Caller should delete the result.
  Compaction* PickCompaction();
  // Return a compaction object for compacting the range [begin,end] in
  // the specified level.  Returns nullptr if there is nothing in that
  // level that overlaps the specified range.  Caller should delete
  // the result.
  Compaction* CompactRange(int level, const InternalKey* begin,
                           const InternalKey* end);
  // Return the maximum overlapping data (in bytes) at next level for any
  // file at a level >= 1.
  int64_t MaxNextLevelOverlappingBytes();
  // Create an iterator that reads over the compaction inputs for "*c".
  // The caller should delete the iterator when no longer needed.
  Iterator* MakeInputIterator(Compaction* c);
  // Returns true iff some level needs a compaction.
  bool NeedsCompaction() const {
    Version* v = current_;
    return (v->compaction_score_ >= 1) || (v->file_to_compact_ != nullptr);
  }
  // Add all files listed in any live version to *live.
  // May also mutate some internal state.
  void AddLiveFiles(std::set<uint64_t>* live);
  // Return the approximate offset in the database of the data for
  // "key" as of version "v".
  uint64_t ApproximateOffsetOf(Version* v, const InternalKey& key);
  // Return a human-readable short (single-line) summary of the number
  // of files per level.  Uses *scratch as backing store.
  struct LevelSummaryStorage {
    char buffer[100];
  };
  const char* LevelSummary(LevelSummaryStorage* scratch) const;
 private:
  class Builder;
  friend class Compaction;
  friend class Version;
  bool ReuseManifest(const std::string& dscname, const std::string& dscbase);
  void Finalize(Version* v);
  void GetRange(const std::vector<FileMetaData*>& inputs, InternalKey* smallest,
                InternalKey* largest);
  void GetRange2(const std::vector<FileMetaData*>& inputs1,
                 const std::vector<FileMetaData*>& inputs2,
                 InternalKey* smallest, InternalKey* largest);
  void SetupOtherInputs(Compaction* c);
  // Save current contents to *log
  Status WriteSnapshot(log::Writer* log);
  void AppendVersion(Version* v);
  Env* const env_;
  const std::string dbname_;
  const Options* const options_;
  TableCache* const table_cache_;
  const InternalKeyComparator icmp_;
  uint64_t next_file_number_;
  uint64_t manifest_file_number_;
```

```
  uint64_t last_sequence_;
  uint64_t log_number_;
  uint64_t prev_log_number_;  // 0 or backing store for memtable being compacted
  // Opened lazily
  WritableFile* descriptor_file_;
  log::Writer* descriptor_log_;
  Version dummy_versions_;  // Head of circular doubly-linked list of versions.
  Version* current_;        // == dummy_versions_.prev_
  // Per-level key at which the next compaction at that level should start.
  // Either an empty string, or a valid InternalKey.
  std::string compact_pointer_[config::kNumLevels];
};
```

随着Builder的不断执行，新的version被构造出来。VersionSet就负责管理多个版本，对应的变量全局唯一，在DBImpl的构造函数中初始化：

versions_(new VersionSet(dbname_, &options_, table_cache_,
&internal_comparator_)) {

其管理一个双向链表：



4.1 LogAndApply

前面已经说过了VersionSet实质上是完成了对Version的管理。

在一开始调用了version_->current()返回了当前的Version，也就是2.3节中的base。

然后调用edit->AddFile()和versions_->LogAndApply(&edit, &mutex_)，实现了将本次的edit更新到了当前版本中。

Addfile（）的过程在类VersionEdit中，相对简单。

下面我们来看versions_->LogAndApply(&edit, &mutex_)实现了什么。

```
Status VersionSet::LogAndApply(VersionEdit* edit, port::Mutex* mu) {
//(1) 检查和设置日志编号
  if (edit->has_log_number_) {
    assert(edit->log_number_ >= log_number_);
    assert(edit->log_number_ < next_file_number_);
  } else {
    edit->SetLogNumber(log_number_);
  }
  if (!edit->has_prev_log_number_) {
```

```
    edit->SetPrevLogNumber(prev_log_number_);
  }
  edit->SetNextFile(next_file_number_);
  edit->SetLastSequence(last_sequence_);
```

//(2) 创建新的Version对象

```
  Version* v = new Version(this);
  {
    Builder builder(this, current_);
    builder.Apply(edit);
    builder.SaveTo(v);
  }
  Finalize(v);   //调整Version
  // Initialize new descriptor log file if necessary by creating
  // a temporary file that contains a snapshot of the current version.
```

//(3) 初始化新的MANIFEST文件

```
  std::string new_manifest_file;
  Status s;
  if (descriptor_log_ == nullptr) {
    // No reason to unlock *mu here since we only hit this path in the
    // first call to LogAndApply (when opening the database).
    assert(descriptor_file_ == nullptr);
    new_manifest_file = DescriptorFileName(dbname_, manifest_file_number_);
    s = env_->NewWritableFile(new_manifest_file, &descriptor_file_);
    if (s.ok()) {
      descriptor_log_ = new log::Writer(descriptor_file_);
      s = WriteSnapshot(descriptor_log_);
    }
  }
  // Unlock during expensive MANIFEST log write
```

//(4) 解锁并写入MANIFEST文件

```
  {
    mu->Unlock();
    // Write new record to MANIFEST log
    if (s.ok()) {
      std::string record;
      edit->EncodeTo(&record);
      s = descriptor_log_->AddRecord(record);
      if (s.ok()) {
        s = descriptor_file_->Sync();
      }
      if (!s.ok()) {
        Log(options_->info_log, "MANIFEST write: %s\n", s.ToString().c_str());
      }
    }
    // If we just created a new descriptor file, install it by writing a
    // new CURRENT file that points to it.
    if (s.ok() && !new_manifest_file.empty()) {
      s = SetCurrentFile(env_, dbname_, manifest_file_number_);
    }
    mu->Lock();
  }
  // Install the new version
```

//(5) 安装新版本

```
  if (s.ok()) {
```

```
    AppendVersion(v);
    log_number_ = edit->log_number_;
    prev_log_number_ = edit->prev_log_number_;
  } else {
    delete v;
    if (!new_manifest_file.empty()) {
      delete descriptor_log_;
      delete descriptor_file_;
      descriptor_log_ = nullptr;
      descriptor_file_ = nullptr;
      env_->RemoveFile(new_manifest_file);
    }
  }
  return s;
}
```

LogAndApply总结:

- **版本更新与元数据管理:** LogAndApply 负责将 VersionEdit 中的变更应用到当前版本，生成一个新的数据库状态版本。
- **持久化元数据:** VersionEdit 的变更会被写入 MANIFEST 文件，这是数据库元数据的持久化文件，记录了每个版本的更改。通过 CURRENT 文件，数据库在重启时可以找到最新的 MANIFEST。
- **线程安全:** 操作过程中通过锁机制 (mu->Unlock() 和 mu->Lock()) 确保并发安全，尤其是在耗时的文件写入过程中释放锁，允许其他操作并发执行。
- **错误处理:** 如果在写入过程中发生错误，会进行适当的资源清理和回滚，避免影响数据库的完整性。

接下来我们分别从这5个过程进行分析:

Builder是一个辅助类，用于将当前版本（current_）和新的VersionEdit进行合并，**见第5节Builder辅助类。**

4.2 Finalize(v)

```
//(2) 创建新的Version对象
 Version* v = new Version(this);
 {
   Builder builder(this, current_);
   builder.Apply(edit);
   builder.SaveTo(v);
 }
 Finalize(v);   //调整Version
```

在创建了一个新的version对象之后，调用了Finalize（），这是VersionSet类中的一个私有成员方法，我们来看其源码实现了什么。

```
void VersionSet::Finalize(Version* v) {
  // Precomputed best level for next compaction
  int best_level = -1;
  double best_score = -1;
  for (int level = 0; level < config::kNumLevels - 1; level++) {
    double score;
    if (level == 0) {
```

```
        // We treat level-0 specially by bounding the number of files
        // instead of number of bytes for two reasons:
        //
        // (1) With larger write-buffer sizes, it is nice not to do too
        // many level-0 compactions.
        //
        // (2) The files in level-0 are merged on every read and
        // therefore we wish to avoid too many files when the individual
        // file size is small (perhaps because of a small write-buffer
        // setting, or very high compression ratios, or lots of
        // overwrites/deletions).
        score = v->files_[level].size() /
                static_cast<double>(config::kL0_CompactionTrigger);
      } else {
        // Compute the ratio of current size to size limit.
        const uint64_t level_bytes = TotalFileSize(v->files_[level]);
        score =
            static_cast<double>(level_bytes) / MaxBytesForLevel(options_, level);
      }
      if (score > best_score) {
        best_level = level;
        best_score = score;
      }
    }
    v->compaction_level_ = best_level;
    v->compaction_score_ = best_score;
  }
```

好了，停在这块，我们已经初步的有了一个认识，但还是很迷惑，不知道MANIFEST之类的是什么东西，这将在笔记19中接着介绍。

有关这个Finalize（v）是干什么的，参见LevelDB**笔记20**.

5. 辅助类Builder

源码仍然位于version_set.cc中。

Builder是VersionSet中私有的一个成员类。

这个代码量也不少，我们先从其构造和析构函数入手：

//构造函数

```
 Builder(VersionSet* vset, Version* base) : vset_(vset), base_(base) {
    base_->Ref();
    BySmallestKey cmp;
    cmp.internal_comparator = &vset_->icmp_;
    for (int level = 0; level < config::kNumLevels; level++) {
      levels_[level].added_files = new FileSet(cmp);
    }
  }
```

//析构函数

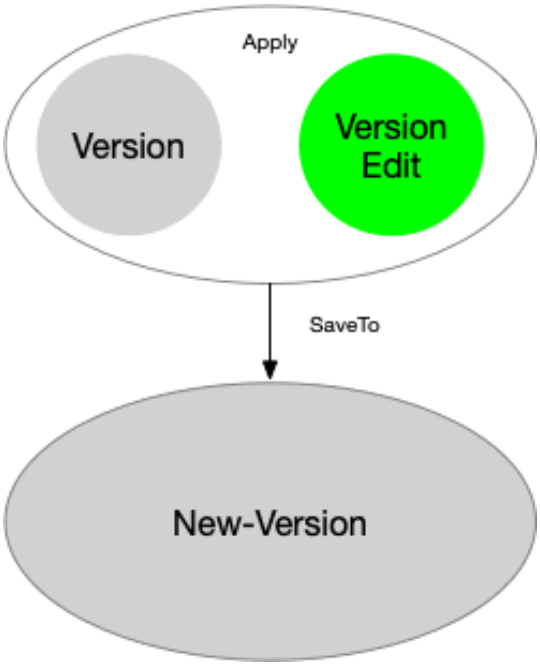
```
  ~Builder() {
    for (int level = 0; level < config::kNumLevels; level++) {
      const FileSet* added = levels_[level].added_files;
      std::vector<FileMetaData*> to_unref;
      to_unref.reserve(added->size());
      for (FileSet::const_iterator it = added->begin(); it != added->end();
           ++it) {
        to_unref.push_back(*it);
```

```
      }
      delete added;
      for (uint32_t i = 0; i < to_unref.size(); i++) {
        FileMetaData* f = to_unref[i];
        f->refs--;
        if (f->refs <= 0) {
          delete f;
        }
      }
    }
    base_->Unref();
  }
```

其中私有的成员变量包括：

```
  // Helper to sort by v->files_[file_number].smallest
  struct BySmallestKey {
    const InternalKeyComparator* internal_comparator;
    bool operator()(FileMetaData* f1, FileMetaData* f2) const {
      int r = internal_comparator->Compare(f1->smallest, f2->smallest);
      if (r != 0) {
        return (r < 0);
      } else {
        // Break ties by file number
        return (f1->number < f2->number);
      }
    }
  };
  typedef std::set<FileMetaData*, BySmallestKey> FileSet;
  struct LevelState {
    std::set<uint64_t> deleted_files;
    FileSet* added_files;
  };
  VersionSet* vset_;
  Version* base_;
  LevelState levels_[config::kNumLevels]; //levels_存储了每一层的added_files及deleted_files
```

提供的两个主要的外部接口是：**Apply(edit)**和**SaveTo(v)**：
通过这两个过程实现了Version+VersionEdit=Version的功能，其中+=分别对应Apply和SaveTo两个接口。



定义具体如下：

```
  // Apply all of the edits in *edit to the current state.
  void Apply(const VersionEdit* edit) {
    // Update compaction pointers
    //下次compact的起始key
    for (size_t i = 0; i < edit->compact_pointers_.size(); i++) {
      const int level = edit->compact_pointers_[i].first;
      vset_->compact_pointer_[level] =
          edit->compact_pointers_[i].second.Encode().ToString();
    }
    // Delete files
    //记录可删除文件到各level对应的delete_files
    for (const auto& deleted_file_set_kvp : edit->deleted_files_) {
      const int level = deleted_file_set_kvp.first;
      const uint64_t number = deleted_file_set_kvp.second;
      levels_[level].deleted_files.insert(number);
    }
    // Add new files
    //记录新增文件到added_files，并计算该文件的allowed_seeks（用于触发compact）
    for (size_t i = 0; i < edit->new_files_.size(); i++) {
      const int level = edit->new_files_[i].first;
      FileMetaData* f = new FileMetaData(edit->new_files_[i].second);
      f->refs = 1;
      // We arrange to automatically compact this file after
      // a certain number of seeks.  Let's assume:
      //   (1) One seek costs 10ms
      //   (2) Writing or reading 1MB costs 10ms (100MB/s)
      //   (3) A compaction of 1MB does 25MB of IO:
      //         1MB read from this level
      //         10-12MB read from next level (boundaries may be misaligned)
      //         10-12MB written to next level
      // This implies that 25 seeks cost the same as the compaction
      // of 1MB of data.  I.e., one seek costs approximately the
      // same as the compaction of 40KB of data.  We are a little
      // conservative and allow approximately one seek for every 16KB
      // of data before triggering a compaction.
      f->allowed_seeks = static_cast<int>((f->file_size / 16384U));
```

```
      if (f->allowed_seeks < 100) f->allowed_seeks = 100;
      levels_[level].deleted_files.erase(f->number);
      levels_[level].added_files->insert(f);
    }
  }
```

//经过Apply之后，levels_更新完成。接下来就是SaveTo了，塔就是将levels_记录的deleted/added files作用于原来的，生成新的version v，就是遍历每一层，将delta的文件更新到合适位置。

```
  // Save the current state in *v.
  void SaveTo(Version* v) {
    BySmallestKey cmp;
    cmp.internal_comparator = &vset_->icmp_;
    for (int level = 0; level < config::kNumLevels; level++) {
      // Merge the set of added files with the set of pre-existing files.
      // Drop any deleted files.  Store the result in *v.
      //当前level的原有文件
      const std::vector<FileMetaData*>& base_files = base_->files_[level];
      std::vector<FileMetaData*>::const_iterator base_iter = base_files.begin();
      std::vector<FileMetaData*>::const_iterator base_end = base_files.end();
      //edit里面的新增文件
      const FileSet* added_files = levels_[level].added_files;
      //新的version vector预留空间，防止频繁copy恶化性能
      v->files_[level].reserve(base_files.size() + added_files->size());
      //遍历所有的新增文件，按照顺序把base_files 和added_files有序加到v->files_
      for (const auto& added_file : *added_files) {
        // Add all smaller files listed in base_
        //按照BysmallestKey排序，找到原有文件中比added_iter小的文件，加入到v
        for (std::vector<FileMetaData*>::const_iterator bpos =
                 std::upper_bound(base_iter, base_end, added_file, cmp);
             base_iter != bpos; ++base_iter) {
          MaybeAddFile(v, level, *base_iter);
        }
        //接着把added_iter加入到v，这样就保证了v里面文件的顺序
        MaybeAddFile(v, level, added_file);
      }
      // Add remaining base files
      for (; base_iter != base_end; ++base_iter) {
        MaybeAddFile(v, level, *base_iter);
      }
#ifndef NDEBUG
      // Make sure there is no overlap in levels > 0
      if (level > 0) {
        for (uint32_t i = 1; i < v->files_[level].size(); i++) {
          const InternalKey& prev_end = v->files_[level][i - 1]->largest;
          const InternalKey& this_begin = v->files_[level][i]->smallest;
          if (vset_->icmp_.Compare(prev_end, this_begin) >= 0) {
            std::fprintf(stderr, "overlapping ranges in same level %s vs. %s\n",
                         prev_end.DebugString().c_str(),
                         this_begin.DebugString().c_str());
            std::abort();
          }
        }
      }
```

```
#endif
    }
  }
```

6 Version

在第2节中说了，在更新VersionEdit之前，也就是调用edit->AddFile之前，先是调用了Version->**PickLevelForMemTableOutput**()这个过程，来为刚从memtable持久化的sstable，选择一个合适的level。

Version用于表示某次compaction后的数据库状态，管理当前的文件集合，因此最重要的一个成员变量files_表示每一层的全部sstable文件：

```
  std::vector<FileMetaData*> files_[config::kNumLevels];
```

找一个合适的level放置新从memtable dump出的sstable

注意：不一定总是放到level 0，尽量放到更大的level

（1）如果[small，large]于0层有重叠，则直接返回0

（2）如果与level+1文件有重叠，或者与level+2层文件重叠过大，则都不应该放入level+1，直接返回level

返回的level最大为2.

```
int Version::PickLevelForMemTableOutput(const Slice& smallest_user_key,
                                        const Slice& largest_user_key) {
  //默认放到level 0
  int level = 0;
  if (!OverlapInLevel(0, &smallest_user_key, &largest_user_key)) {
    //如果level 0的文件没有交集
    // Push to next level if there is no overlap in next level,
    // and the #bytes overlapping in the level after that are limited.
    InternalKey start(smallest_user_key, kMaxSequenceNumber, kValueTypeForSeek);
    InternalKey limit(largest_user_key, 0, static_cast<ValueType>(0));
    std::vector<FileMetaData*> overlaps;
    while (level < config::kMaxMemCompactLevel) {
      //kMaxMemCompactLevel = 2，因此level= 0 or 1
      //与level+1（下一层）文件有交集，只能返回该level层，目的是为了保证下一层文件是有序的
      if (OverlapInLevel(level + 1, &smallest_user_key, &largest_user_key)) {
        break;
      }
      if (level + 2 < config::kNumLevels) {
        //如果level+2（下两层）的文件与key range有重叠的文件大小超过20M
        //目的是避免放入level+1层后，与level+2 compact时文件过大
        // Check that file does not overlap too many grandparent bytes.
        GetOverlappingInputs(level + 2, &start, &limit, &overlaps);
        const int64_t sum = TotalFileSize(overlaps);
        if (sum > MaxGrandParentOverlapBytes(vset_->options_)) {
          break;
        }
      }
      level++;
```

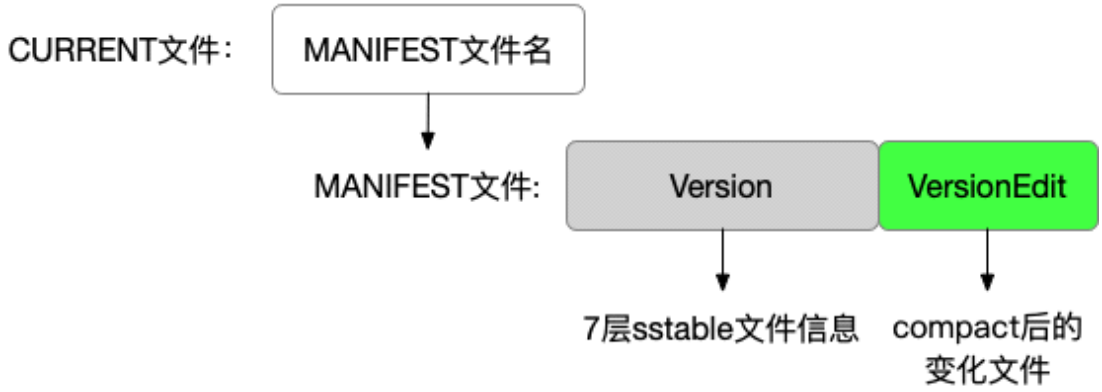
```
    }
  }
  return level;
}
```

选择的几个原则：

1. Level 0的sstable数量有着严格的限制，因此尽可能尝试放到一个更大的level；
2. 大于level 0的各层文件间时是有序的，如果放到对应的层数会导致文件间不严格有序，会影响读取，则不再尝试；
3. 如果放到level +1层，于level+2层的文件重叠很大，就会导致compact到该文件时，压力过大，则不再尝试。这算是一个预测，放到level层能够缓冲这一点。
4. 最大返回level 2，这大概是个经验值。

LevelDB笔记19：MANIFEST与Version

2024年9月25日 16:13



在 LevelDB 中，**MANIFEST 文件** 是用于记录数据库元数据和版本信息的重要文件。它扮演着数据库版本历史的持久化日志的角色，确保数据库在持久化和恢复时能够保持一致性。以下是 MANIFEST 文件的详细解释，以及它与 **Version** 的关系及在 LevelDB 中的作用。

1. MANIFEST 文件是什么？

MANIFEST 文件是一个**顺序日志**文件，用于存储数据库版本变更的详细记录。它包含了每次数据库元数据更新时的操作记录，比如哪些 SSTable 文件被添加、删除，哪个文件编号正在被使用，日志文件的编号等。

每当数据库状态发生变化时，这些变更信息会被记录到 MANIFEST 文件中。它是 LevelDB 数据库的核心文件之一，因为它保存了数据库状态的历史版本。

2. MANIFEST 文件与 Version 的关系

- **Version** 代表了数据库在某一时刻的具体状态或“视图”。它包含了 SSTable 文件的位置、哪些文件是当前活动的，哪些文件是不可见的等信息。
- **MANIFEST 文件** 是用于持久化存储这些 Version 对象的变化。每次 VersionEdit（表示一次数据库状态的修改）被应用后，数据库生成一个新的 Version，并且将这个 VersionEdit 的内容记录到 MANIFEST 文件中。
- 当 LevelDB 启动时，它会读取 CURRENT 文件，找到指向的最新的 MANIFEST 文件。然后根据这个 MANIFEST 文件，重建数据库的最新 Version 对象。也就是说，**MANIFEST 文件记录了所有的 VersionEdit 操作，保证数据库可以从任何时刻的状态正确恢复**。

3. MANIFEST 文件在 LevelDB 中的作用

MANIFEST 文件在 LevelDB 中扮演着几个关键角色：

（1）记录数据库元数据变更

每当数据库发生元数据的变化（例如添加、删除 SSTable 文件，或者变更某个文件的编号）时，VersionEdit 对象会记录这些变化，而这些 VersionEdit 会被编码成二进制记录，并写入 MANIFEST 文件。

（2）持久化数据库的状态

MANIFEST 文件保证了数据库的元数据是持久化的。当数据库崩溃或关闭时，LevelDB 可以在下次启动时通过读取 MANIFEST 文件恢复到最新状态。

- 每次数据库写操作后，新的 VersionEdit 会被写入 MANIFEST 文件并同步到磁盘，确保即使系统宕机，数据库元数据的更新也不会丢失。

（3）恢复数据库状态

当数据库启动时，LevelDB 会从 CURRENT 文件中读取到最新的 MANIFEST 文件路径，然后解析 MANIFEST 文件中的所有 VersionEdit 记录，并基于这些记录重建数据库的状态。**MANIFEST 文件是 LevelDB 实现数据库恢复的基础**。

- 如果 MANIFEST 文件中没有这些历史记录，数据库将无法知道哪些 SSTable 文件有效，导致数据丢失。

（4）维护数据库的历史

MANIFEST 文件记录了数据库元数据的每一次变更，维护了数据库的历史版本信息。这对于调试、审计以及回溯非常有用。

4. MANIFEST 文件工作流程

在 LevelDB 中，MANIFEST 文件与 VersionSet、VersionEdit 和 Version 紧密相关。它的工作流程大致如下：

1. **数据库状态变更**：当数据库发生变化时（例如新的数据被插入、合并操作），LevelDB 会创建一个 VersionEdit 对象，记录这些变更。
2. **记录到 MANIFEST 文件**：VersionEdit 被应用后，通过 VersionSet::LogAndApply 将它的内容编码并追加到 MANIFEST 文件中。通过这种方式，所有的版本更新操作都会被写入到 MANIFEST。
3. **同步到磁盘**：为了保证数据一致性，每次写入操作后，MANIFEST 文件会被同步到磁盘，确保所有变更被持久化。
4. **创建新的 Version**：这些 VersionEdit 最终生成一个新的 Version，并成为数据库的当前版本。
5. **恢复过程**：当数据库重启时，LevelDB 读取 MANIFEST 文件中的所有 VersionEdit 记录，逐步应用这些记录并重建最新的 Version，恢复到最后一次成功写入的状态。

5. CURRENT 文件与 MANIFEST 的关系

- **CURRENT 文件**：CURRENT 文件是一个小文件，内容只有一行，指向当前正在使用的 MANIFEST 文件。例如，MANIFEST-000001。它确保 LevelDB 知道启动时应该使用哪个 MANIFEST 文件来恢复状态。
- **MANIFEST 文件与 CURRENT 文件的协作**：每次 LevelDB 创建新的 MANIFEST 文件后，会更新 CURRENT 文件，让它指向新的 MANIFEST 文件路径。

总结

- **MANIFEST 文件** 是 LevelDB 中用于记录数据库元数据变化的日志文件。
- 它记录了数据库的每次状态变化（通过 VersionEdit），持久化了每次的 Version，确保数据库在宕机后能够正确恢复。
- MANIFEST 文件与 Version 的关系紧密，它保证了每次 VersionEdit 的变更被保存，并通过 CURRENT 文件跟踪数据库的最新版本。

MANIFEST 是 LevelDB 中维持数据一致性、数据库状态恢复的核心机制之一。

那么，我们再次回到笔记18中后面的函数**LogAndApply()**：

```
Status VersionSet::LogAndApply(VersionEdit* edit, port::Mutex* mu) {
//(1) 检查和设置日志编号
  ......
//(2) 创建新的Version对象
  Version* v = new Version(this);
  {
```

```
    Builder builder(this, current_);
    builder.Apply(edit);
    builder.SaveTo(v);
  }
  Finalize(v);   //调整Version
  // Initialize new descriptor log file if necessary by creating
  // a temporary file that contains a snapshot of the current version.
```

//(3) 初始化新的MANIFEST文件

```
  std::string new_manifest_file;
  Status s;
  if (descriptor_log_ == nullptr) {
    // No reason to unlock *mu here since we only hit this path in the
    // first call to LogAndApply (when opening the database).
    assert(descriptor_file_ == nullptr);
    new_manifest_file = DescriptorFileName(dbname_, manifest_file_number_);
    s = env_->NewWritableFile(new_manifest_file, &descriptor_file_);
    if (s.ok()) {
      descriptor_log_ = new log::Writer(descriptor_file_);
      s = WriteSnapshot(descriptor_log_);
    }
  }
  // Unlock during expensive MANIFEST log write
```

//(4) 解锁并写入MANIFEST文件

```
  {
    mu->Unlock();
    // Write new record to MANIFEST log
    if (s.ok()) {
      std::string record;
      edit->EncodeTo(&record);
      s = descriptor_log_->AddRecord(record);
      if (s.ok()) {
        s = descriptor_file_->Sync();
      }
      if (!s.ok()) {
        Log(options_->info_log, "MANIFEST write: %s\n", s.ToString().c_str());
      }
    }
    // If we just created a new descriptor file, install it by writing a
    // new CURRENT file that points to it.
    if (s.ok() && !new_manifest_file.empty()) {
      s = SetCurrentFile(env_, dbname_, manifest_file_number_);
    }
    mu->Lock();
  }
  // Install the new version
```

//(5) 安装新版本

```
  if (s.ok()) {
    AppendVersion(v);
    log_number_ = edit->log_number_;
    prev_log_number_ = edit->prev_log_number_;
  } else {
    delete v;
    if (!new_manifest_file.empty()) {
      delete descriptor_log_;
      delete descriptor_file_;
```

```
      descriptor_log_ = nullptr;
      descriptor_file_ = nullptr;
      env_->RemoveFile(new_manifest_file);
    }
  }
  return s;
}
```

我们主要来看（3）和（4）部分：

（3）初始化新的MANIFEST文件（如果需要）

```
 std::string new_manifest_file;
  Status s;
```

首先判断了descriptor_log_是否为空，

```
  if (descriptor_log_ == nullptr) {
    // No reason to unlock *mu here since we only hit this path in the
    // first call to LogAndApply (when opening the database).
    assert(descriptor_file_ == nullptr);
    new_manifest_file = DescriptorFileName(dbname_, manifest_file_number_);
    s = env_->NewWritableFile(new_manifest_file, &descriptor_file_);
    if (s.ok()) {
      descriptor_log_ = new log::Writer(descriptor_file_);
      s = WriteSnapshot(descriptor_log_);
    }
  }
```

上述descriptor_log_与descriptor_file是VersionSet中的私有成员变量，定义如下：

```
  WritableFile* descriptor_file_;
  log::Writer* descriptor_log_;
```

首先：

```
    new_manifest_file = DescriptorFileName(dbname_, manifest_file_number_);
```

DecriptorFileName函数会生成新的MANIFEST文件名，定义如下：

```
std::string DescriptorFileName(const std::string& dbname, uint64_t number) {
  assert(number > 0);
  char buf[100];
  std::snprintf(buf, sizeof(buf), "/MANIFEST-%06llu",
                static_cast<unsigned long long>(number));
  return dbname + buf;
}
```

比如说调用std::string filename = DescriptorFileName("/path/to/db", 1);会返回/path/to/db/MANIFEST-000001。std::snprintf：这个函数会将格式化后的字符串写入 buf。

接下来调用了

```
    s = env_->NewWritableFile(new_manifest_file, &descriptor_file_);   (1)
```

这里可以看一下笔记9：Env。下面是POSIXEnv类中的**NewWritableFile**函数的实现：

```
Status NewWritableFile(const std::string& filename,
                       WritableFile** result) override {
    int fd = ::open(filename.c_str(),
                    O_TRUNC | O_WRONLY | O_CREAT | kOpenBaseFlags, 0644);
    // 调用POSIX系统的open函数，用于打开或创建文件。返回文件标识符fd
    if (fd < 0) {
      *result = nullptr;
      return PosixError(filename, errno);
```

```
    }
    *result = new PosixWritableFile(filename, fd);
    return Status::OK();
  }
```

从上面这个函数中可以看到该函数实现了：打开或创建了名为传进来的new_manifest_file的文件；之后使用返回的文件标识符fd和文件名创建了一个新的PosixWritableFile对象。

WritableFile的内容在哪？？？

WriteableFile的内容可以参见笔记5：Log中有。

那么PosixWritableFile的内容可以结合笔记5：log

最后再回到（1）上，实则是descriptor_file_指向了刚才新建的那个PosixWritableFile对象。

```
    if (s.ok()) {
      descriptor_log_ = new log::Writer(descriptor_file_);
      s = WriteSnapshot(descriptor_log_);
    }
```

然后，以descriptor_file_为参数实例化了一个新的log:: Writer类型的对象，然后用descriptor_log_指向了它。

最后来看**WriteSnapshot（descriptor_log_）**:他在VersionSet中：

```
Status VersionSet::WriteSnapshot(log::Writer* log) {
  // TODO: Break up into multiple records to reduce memory usage on recovery?
  // Save metadata
  VersionEdit edit;
  edit.SetComparatorName(icmp_.user_comparator()->Name());
   // Save compaction pointers
      //(1) 保存压缩指针
  for (int level = 0; level < config::kNumLevels; level++) {
    if (!compact_pointer_[level].empty()) {
      InternalKey key;
      key.DecodeFrom(compact_pointer_[level]); //compact_pointer_是string类型大小为kNumlevels的数组，存储着internalkey
      edit.SetCompactPointer(level, key);  //将Pair<level,key>这对数据存入edit->std::vector<std::pair<int, InternalKey>> compact_pointers_中
    }
  }
  // Save files
      //(2) 保存文件信息
  for (int level = 0; level < config::kNumLevels; level++) {
    const std::vector<FileMetaData*>& files = current_->files_[level];
    for (size_t i = 0; i < files.size(); i++) {
      const FileMetaData* f = files[i];
      edit.AddFile(level, f->number, f->file_size, f->smallest, f->largest);
    }
  }
  std::string record;
  edit.EncodeTo(&record);
  return log->AddRecord(record);
}
```

VersionSet::WriteSnapshot是LevelDB中用于保存当前版本快照（snapshot）的函数。通过将数据库的元数据写入日志（MANIFEST文件）来保存数据库的当前状态，以便在恢复时可以重建这个版本。它主要就是**三个环节：一是新构建了一个VersionEdit；二是调用edit.EncodeTo(&record)将**

VersionEdit对象编码成一个字符串记录，包含比较器名称、压缩指针、文件信息等所有元数据；三是调用log->AddRecord(record)将这个编码后的字符串记录添加到log中，即将数据写入日志文件。

接下来看看是如何存入MANIFEST文件中的：

```
void VersionEdit::EncodeTo(std::string* dst) const {
  if (has_comparator_) {
    PutVarint32(dst, kComparator);
    PutLengthPrefixedSlice(dst, comparator_);
  }
  if (has_log_number_) {
    PutVarint32(dst, kLogNumber);
    PutVarint64(dst, log_number_);
  }
  if (has_prev_log_number_) {
    PutVarint32(dst, kPrevLogNumber);
    PutVarint64(dst, prev_log_number_);
  }
  if (has_next_file_number_) {
    PutVarint32(dst, kNextFileNumber);
    PutVarint64(dst, next_file_number_);
  }
  if (has_last_sequence_) {
    PutVarint32(dst, kLastSequence);
    PutVarint64(dst, last_sequence_);
  }
  for (size_t i = 0; i < compact_pointers_.size(); i++) {
    PutVarint32(dst, kCompactPointer);
    PutVarint32(dst, compact_pointers_[i].first);  // level
    PutLengthPrefixedSlice(dst, compact_pointers_[i].second.Encode());
  }
  for (const auto& deleted_file_kvp : deleted_files_) {
    PutVarint32(dst, kDeletedFile);
    PutVarint32(dst, deleted_file_kvp.first);   // level
    PutVarint64(dst, deleted_file_kvp.second);  // file number
  }
  for (size_t i = 0; i < new_files_.size(); i++) {
    const FileMetaData& f = new_files_[i].second;
    PutVarint32(dst, kNewFile);
    PutVarint32(dst, new_files_[i].first);  // level
    PutVarint64(dst, f.number);
    PutVarint64(dst, f.file_size);
    PutLengthPrefixedSlice(dst, f.smallest.Encode());
    PutLengthPrefixedSlice(dst, f.largest.Encode());
  }
}
```

那么上面这些过程清楚之后，解锁并写入MANIFEST的过程也就可以理解了。

```
// If we just created a new descriptor file, install it by writing a
    // new CURRENT file that points to it.
    if (s.ok() && !new_manifest_file.empty()) {
      s = SetCurrentFile(env_, dbname_, manifest_file_number_);
    }

Status SetCurrentFile(Env* env, const std::string& dbname,
```

```
                         uint64_t descriptor_number) {
  // Remove leading "dbname/" and add newline to manifest file name
  std::string manifest = DescriptorFileName(dbname, descriptor_number);
  Slice contents = manifest;
  assert(contents.starts_with(dbname + "/"));
  contents.remove_prefix(dbname.size() + 1);
  std::string tmp = TempFileName(dbname, descriptor_number);
  Status s = WriteStringToFileSync(env, contents.ToString() + "\n", tmp);
  if (s.ok()) {
    s = env->RenameFile(tmp, CurrentFileName(dbname));
  }
  if (!s.ok()) {
    env->RemoveFile(tmp);
  }
  return s;
}
```

在LogAndApply（）函数的最后部分，调用了

```
void VersionSet::AppendVersion(Version* v) {
  // Make "v" current
  assert(v->refs_ == 0);
  assert(v != current_);
  if (current_ != nullptr) {
    current_->Unref();
  }
  current_ = v;
  v->Ref();

  // Append to linked list
  v->prev_ = dummy_versions_.prev_;
  v->next_ = &dummy_versions_;
  v->prev_->next_ = v;
  v->next_->prev_ = v;
}
```

截至到此，从笔记17的minor compaction过程出发，讲了其过程主要做了什么，写磁盘sstable0然后更新最新version，在这个过程中需要维护日志文件MANIFEST，用来记录在当前Version上所作的更新部分VersionEdit。

Minor Compaction完成的是从immutable到持久化到stable0过程的转换，可以再次回到笔记17中查看**WriteLevel0Table**() 的过程进行梳理。

到此基本可以理解LevelDB中版本相关的内容：Version、VersionEdit、VersionSet，以及其余的诸如MANIFEST文件之类的东西了。

这里可以去看

【深度知识】LevelDB从入门到原理详解-腾讯云开发者社区-腾讯云 (tencent.com)这篇笔记。

LevelDB笔记20：seek_compaction&size_compaction

2024年9月26日 19:57

1. Seek_compaction

一个直观的想法就是如果文件多次seek但是没有查找到数据，那么就应该被compact了，否则就会浪费更多的seek。用一次compact来解决长久空seek的问题，本质上还是如何平衡读写的思想。

我的理解就是一个文件很长时间内没有被seek，或者说seek了但并没有找到要seek的那个数据，所以这个文件很大程度上就是相对比较cold的，所以就应该被compact。

具体的，当一个新文件更新进入版本管理，计算该文件允许seek但是没有查找数据的最大次数，当超过该次数后，就该compact该文件了。

对应到代码实现，是在VersionSet:: Builder:: Apply阶段

```
// Apply all of the edits in *edit to the current state.
 void Apply(const VersionEdit* edit) {
   // Update compaction pointers
       / / 下次compact的起始key
   for (size_t i = 0; i < edit->compact_pointers_.size(); i++) {
     const int level = edit->compact_pointers_[i].first;
     vset_->compact_pointer_[level] =
         edit->compact_pointers_[i].second.Encode().ToString();
   }
   // Delete files
       / / 记录可删除文件到各level对应的delete_files
   for (const auto& deleted_file_set_kvp : edit->deleted_files_) {
     const int level = deleted_file_set_kvp.first;
     const uint64_t number = deleted_file_set_kvp.second;
     levels_[level].deleted_files.insert(number);
   }
   // Add new files
       / / 记录新增文件到added_files，并计算该文件的allowed_seeks（用于触发compact）
   for (size_t i = 0; i < edit->new_files_.size(); i++) {
     const int level = edit->new_files_[i].first;
     FileMetaData* f = new FileMetaData(edit->new_files_[i].second);
     f->refs = 1;
     // We arrange to automatically compact this file after
     // a certain number of seeks.  Let's assume:
     //   (1) One seek costs 10ms
```

```
    //   (2) Writing or reading 1MB costs 10ms (100MB/s)
    //   (3) A compaction of 1MB does 25MB of IO:
    //         1MB read from this level
    //         10-12MB read from next level (boundaries may be misaligned)
    //         10-12MB written to next level
    // This implies that 25 seeks cost the same as the compaction
    // of 1MB of data.  I.e., one seek costs approximately the
    // same as the compaction of 40KB of data.  We are a little
    // conservative and allow approximately one seek for every 16KB
    // of data before triggering a compaction.
```

```
// 1. 1次seek花费10ms // 2. 1M读写花费10ms // 3. 1M文件的compact需要25M IO(读写10-12MB的下一层文件)，为什么10-12M?经验值? // 因此1M的compact时间 = 25次seek时间 = 250ms // 也就是40K的compact时间 = 1次seek时间，保守点取16KB，即t = 16K的compact时间 = 1次seek时间 // compact这个文件的时间: file_size / 16K // 如果文件seek很多次但是没有找到key，时间和已经比compact时间要大，就应该compact了 // 这个次数记录到f->allowed_seeks
    f->allowed_seeks = static_cast<int>((f->file_size / 16384U));
    if (f->allowed_seeks < 100) f->allowed_seeks = 100;
    levels_[level].deleted_files.erase(f->number);
    levels_[level].added_files->insert(f);
  }
 }
```

主要用到的变量就是**allowed_seeks**，即允许seek的最大次数，当allowed_seeks这个值降为0的时候，就需要对这个文件执行compact操作了。

当查找文件而没有找到时，allowed_seeks--，降为0时该文件标记到file_to_compact_:

```
bool Version::UpdateStats(const GetStats& stats) {
 FileMetaData* f = stats.seek_file;
 if (f != nullptr) {
  f->allowed_seeks--;
      / / 如果allowed_seeks将到0，则记录该文件需要compact了
  if (f->allowed_seeks <= 0 && file_to_compact_ == nullptr) {
   file_to_compact_ = f;
   file_to_compact_level_ = stats.seek_file_level;
   return true;
  }
 }
 return false;
}
```

2. Size_compaction

```
//(2) 创建新的Version对象
  Version* v = new Version(this);
  {
    Builder builder(this, current_);
    builder.Apply(edit);
    builder.SaveTo(v);
  }
  Finalize(v);    //调整Version
```

在创建了一个新的version对象之后，调用了Finalize（），这是VersionSet类中的一个私有成员方法，我们来看其源码实现了什么。

```
void VersionSet::Finalize(Version* v) {
  // Precomputed best level for next compaction
  int best_level = -1;
  double best_score = -1;
  for (int level = 0; level < config::kNumLevels - 1; level++) {
    double score;
    if (level == 0) {
      // We treat level-0 specially by bounding the number of files
      // instead of number of bytes for two reasons:
      //
      // (1) With larger write-buffer sizes, it is nice not to do too
      // many level-0 compactions.
      //
      // (2) The files in level-0 are merged on every read and
      // therefore we wish to avoid too many files when the individual
      // file size is small (perhaps because of a small write-buffer
      // setting, or very high compression ratios, or lots of
      // overwrites/deletions).
            // 特殊处理level-0，通过文件数量而不是字节数来限制。
      score = v->files_[level].size() /
          static_cast<double>(config::kL0_CompactionTrigger);
    } else {
      // Compute the ratio of current size to size limit.
            // 计算当前级别的字节数与该级别最大允许字节数的比例。
      const uint64_t level_bytes = TotalFileSize(v->files_[level]);
      score =
          static_cast<double>(level_bytes) / MaxBytesForLevel(options_, level);
    }
    if (score > best_score) {
      best_level = level;
      best_score = score;
    }
```

```
  }
  v->compaction_level_ = best_level;
  v->compaction_score_ = best_score;
}
```

Finalize(version v)这个函数封装在Versionset类中实现，接受一个version v的对象作为参数。

再来总结一下VersionSet和Version：在LevelDB中，VersionSet类负责管理数据库的多个版本（Version对象），并处理版本之间的更新和维护。Version对象表示数据库在某一时刻的状态，包括各个级别level中的文件（SSTable文件）的分布。

Finalize函数是VersionSet类中的一个私有成员方法，负责在创建和更新Version对象后，确定下一个需要压缩（compaction）的最佳级别（level）。

两个作用：

（1）**确定下一个需要进行compaction的最佳级别**：通过计算每个level的compaction_score_，选择得分最高的级别作为下一次压缩操作的目标级别。

（2）**更新Version对象的压缩级别和得分**：将计算出的最佳级别和对应的得分赋值给Version对象，以便后续的压缩操作参考。

总的来看函数内部是一个for循环，对每一level的的文件进行了考虑计算得到了相应层的compaction_score_，然后统计得到所有层中最大score的那一个compaction_score和相应的level，最后赋值给best_score和best_level。

第0层level由于该层文件是重叠的，每次读取要遍历该0层的所有的文件，而其余层文件都是不重叠的。因此level-0需要特殊处理：

```
score = v->files_[level].size() /
        static_cast<double>(config::kL0_CompactionTrigger);
```

用当前level 0的文件数量除以一个触发阈值kL0_CompcationTrigger，得分越高表示文件数量越多，压缩得必要性越大。

为什么要这样单独处理呢？？原因是：

- **控制文件数量**：Level 0 的压缩触发是基于文件数量而不是字节数。这是因为 Level 0 的文件经常被读取和合并，过多的文件会影响读取性能。
- **优化读性能**：Level 0 的文件在读取时需要合并，因此希望通过限制文件数量来减少合并的开销。

其余几层level的处理：

```
const uint64_t level_bytes = TotalFileSize(v->files_[level]);
    score =
      static_cast<double>(level_bytes) / MaxBytesForLevel(options_, level);
```

如上所示，通过计算当前级别的总字节数level_bytes，然后除以该级别的最大允许字节数MaxBytesForLevel(options_, level)，得分越高，表示当前字节数越接近或超过限制，compaction的必要性越大。

总结

Finalize 函数在 LevelDB 中的作用是通过评估各个级别的文件数量和字节数，确定下一次压缩操作的最佳级别。这有助于维护数据库的存储结构，确保系统性能和稳定性。具体来

说，Finalize 函数：

1. **评估每个级别的压缩得分**：
    - Level 0 根据文件数量评估。
    - 其他级别根据总字节数与最大允许字节数的比例评估。
2. **选择得分最高的级别作为下一次压缩的目标级别。**
3. **更新 Version 对象的压缩级别和得分**，以便后续的压缩操作参考。

通过这种方式，LevelDB 能够动态地管理数据在不同级别之间的分布，优化存储和读取性能，避免数据堆积导致的性能下降。

在major compact选择文件时就会用到上述两个条件，这个地方见下一章的**LevelDB笔记21：major compaction中的PickCompaction( ) 。**

因此，版本管理的作用之三，就是在新增或者遍历文件的过程中，为major compact筛选文件。

LevelDB笔记21：major compaction之筛选文件

2024年10月8日　　10:35

1 前序

可以先看笔记9：Env

事实上到最后调用的是**DBImpl ::BackgroundCompaction（）**这个过程

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

//(1)Memtable压缩，**Minor Compaction**

//如果是不可变的Memtable，调用CompactMemtable（）进行压缩，即minor Compaction的过程

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

//(2)文件压缩

```
  Compaction* c;
  bool is_manual = (manual_compaction_ != nullptr);
  InternalKey manual_end;
  //手动压缩
  if (is_manual) {
    ManualCompaction* m = manual_compaction_;
    c = versions_->CompactRange(m->level, m->begin, m->end);
    m->done = (c == nullptr);
    if (c != nullptr) {
      manual_end = c->input(0, c->num_input_files(0) - 1)->largest;
    }
    Log(options_.info_log,
        "Manual compaction at level-%d from %s .. %s; will stop at %s\n",
        m->level, (m->begin ? m->begin->DebugString().c_str() : "(begin)"),
        (m->end ? m->end->DebugString().c_str() : "(end)"),
        (m->done ? "(end)" : manual_end.DebugString().c_str()));
  } else {
  //自动压缩
    c = versions_->PickCompaction();
  }


  Status status;
  if (c == nullptr) {
    // Nothing to do
  } else if (!is_manual && c->IsTrivialMove()) {
    // Move file to next level
    assert(c->num_input_files(0) == 1);
    FileMetaData* f = c->input(0, 0);
    c->edit()->RemoveFile(c->level(), f->number);
```

```
    c->edit()->AddFile(c->level() + 1, f->number, f->file_size, f->smallest,
                       f->largest);
    status = versions_->LogAndApply(c->edit(), &mutex_);
    if (!status.ok()) {
      RecordBackgroundError(status);
    }
    VersionSet::LevelSummaryStorage tmp;
    Log(options_.info_log, "Moved #%lld to level-%d %lld bytes %s: %s\n",
        static_cast<unsigned long long>(f->number), c->level() + 1,
        static_cast<unsigned long long>(f->file_size),
        status.ToString().c_str(), versions_->LevelSummary(&tmp));
  } else {
    CompactionState* compact = new CompactionState(c);
    status = DoCompactionWork(compact);
    if (!status.ok()) {
      RecordBackgroundError(status);
    }
    CleanupCompaction(compact);
    c->ReleaseInputs();
    RemoveObsoleteFiles();
  }
  delete c;
```

//压缩结果处理，一般是错误处理

```
  if (status.ok()) {
    // Done
  } else if (shutting_down_.load(std::memory_order_acquire)) {
    // Ignore compaction errors found during shutting down
  } else {
    Log(options_.info_log, "Compaction error: %s", status.ToString().c_str());
  }
```

//手动压缩结束处理

```
  if (is_manual) {
    ManualCompaction* m = manual_compaction_;
    if (!status.ok()) {
      m->done = true;
    }
    if (!m->done) {
      // We only compacted part of the requested range.  Update *m
      // to the range that is left to be compacted.
      m->tmp_storage = manual_end;
      m->begin = &m->tmp_storage;
    }
    manual_compaction_ = nullptr;
  }
}
```

很明显**major compact过程的第一步**首先是调用**versions_->PickCompaction()**
先来看compaction这个结构是什么，再来具体的分析PickCompaction（）这个过程。

2 Compaction

compaction这个类封装了有关compact相关的信息。

```
// A Compaction encapsulates information about a compaction.
class Compaction {
 public:
  ~Compaction();
  // Return the level that is being compacted.  Inputs from "level"
  // and "level+1" will be merged to produce a set of "level+1" files.
  int level() const { return level_; }
  // Return the object that holds the edits to the descriptor done
  // by this compaction.
  VersionEdit* edit() { return &edit_; }
  // "which" must be either 0 or 1
  int num_input_files(int which) const { return inputs_[which].size(); }
  // Return the ith input file at "level()+which" ("which" must be 0 or 1).
  FileMetaData* input(int which, int i) const { return inputs_[which][i]; }
  // Maximum size of files to build during this compaction.
  uint64_t MaxOutputFileSize() const { return max_output_file_size_; }
  // Is this a trivial compaction that can be implemented by just
  // moving a single input file to the next level (no merging or splitting)
  bool IsTrivialMove() const;
  // Add all inputs to this compaction as delete operations to *edit.
  void AddInputDeletions(VersionEdit* edit);
  // Returns true if the information we have available guarantees that
  // the compaction is producing data in "level+1" for which no data exists
  // in levels greater than "level+1".
  bool IsBaseLevelForKey(const Slice& user_key);
  // Returns true iff we should stop building the current output
  // before processing "internal_key".
  bool ShouldStopBefore(const Slice& internal_key);
  // Release the input version for the compaction, once the compaction
  // is successful.
  void ReleaseInputs();
 private:
  friend class Version;
  friend class VersionSet;
  Compaction(const Options* options, int level);
  int level_;
  uint64_t max_output_file_size_;
  Version* input_version_;
  VersionEdit edit_;
  // Each compaction reads inputs from "level_" and "level_+1"
  std::vector<FileMetaData*> inputs_[2];  // The two sets of inputs
  // State used to check for number of overlapping grandparent files
  // (parent == level_ + 1, grandparent == level_ + 2)
  std::vector<FileMetaData*> grandparents_;
  size_t grandparent_index_;  // Index in grandparent_starts_
  bool seen_key_;             // Some output key has been seen
  int64_t overlapped_bytes_;  // Bytes of overlap between current output
                              // and grandparent files
  // State for implementing IsBaseLevelForKey
  // level_ptrs_ holds indices into input_version_->levels_: our state
  // is that we are positioned at one of the file ranges for each
  // higher level than the ones involved in this compaction (i.e. for
  // all L >= level_ + 2).
  size_t level_ptrs_[config::kNumLevels];
};
```

3 PickCompaction( )

```
Compaction* VersionSet::PickCompaction() {
  Compaction* c;
  int level;
  // We prefer compactions triggered by too much data in a level over
  // the compactions triggered by seeks.
  const bool size_compaction = (current_->compaction_score_ >= 1);
  const bool seek_compaction = (current_->file_to_compact_ != nullptr);
  // (1) 基于大小的compaction
  if (size_compaction) {
    level = current_->compaction_level_; // 获取当前版本标记的压缩级别，这个值的计算在笔记20中的size_compaction中有介绍
    assert(level >= 0);
    assert(level + 1 < config::kNumLevels);
    c = new Compaction(options_, level);  //创建一个新的Compaction对象，指定压缩级别为level
    // Pick the first file that comes after compact_pointer_[level]
    // 遍历当前level的所有文件，选择第一个符合条件的文件：
    for (size_t i = 0; i < current_->files_[level].size(); i++) {
      FileMetaData* f = current_->files_[level][i];
      if (compact_pointer_[level].empty() ||
          icmp_.Compare(f->largest.Encode(), compact_pointer_[level]) > 0) {
        // 条件1：当前级别的压缩指针为空则选择该level的第一个文件
        // 条件2：如果该文件f的最大键大于压缩指针，则选择该文件
        //这里的遗留问题：compact_pointer_是什么？其含义和作用是什么
        c->inputs_[0].push_back(f);  //找到一个文件后将该文件f加入压缩任务
        break;
      }
    }
    if (c->inputs_[0].empty()) {
      // Wrap-around to the beginning of the key space
      // 如果遍历后没有找到符合条件的文件，则将当前级别的第一个文件加入压缩任务，进入环绕处理，什么叫做环绕处理？
      c->inputs_[0].push_back(current_->files_[level][0]);
    }
  }
  // (2) 基于seek次数（读取操作）
  else if (seek_compaction) {
    level = current_->file_to_compact_level_;
    c = new Compaction(options_, level);
    c->inputs_[0].push_back(current_->file_to_compact_);
  } else {
    return nullptr;  // 无需压缩，则返回空即可
  }
  c->input_version_ = current_;  // 将当前版本设置为压缩任务的输入版本
  c->input_version_->Ref();      // 增加当前版本的引用计数，防止在压缩任务执行过程中被释放
  // Files in level 0 may overlap each other, so pick up all overlapping ones
```

```
  // (3) 特殊处理Level 0
  if (level == 0) {
    // level 0的文件可能存在键值范围重叠，因此需要合并所有重叠的文件进行压缩
    InternalKey smallest, largest;
    GetRange(c->inputs_[0], &smallest, &largest); //获取当前选择文件的最小键和最大键，确定压缩范围
    // Note that the next call will discard the file we placed in
    // c->inputs_[0] earlier and replace it with an overlapping set
    // which will include the picked file.
    current_->GetOverlappingInputs(0, &smallest, &largest, &c->inputs_[0]); //获取重叠的文件，根据压缩范围，获取所有与之重叠的文件，加入压缩任务。
    assert(!c->inputs_[0].empty());
  }
  SetupOtherInputs(c); //设置其他输入文件
  return c;
}
```

VersionSet::PickCompaction 函数在 LevelDB 的压缩机制中起到了关键作用。它通过评估不同级别的文件分布和压缩触发条件，选择最适合进行压缩的级别和文件，从而优化数据库的存储结构和性能。

4 PickCompaction中的辅助函数：

```
// Stores the minimal range that covers all entries in inputs in
// *smallest, *largest.
// REQUIRES: inputs is not empty
```

4.1 GetRange

```
void VersionSet::GetRange(const std::vector<FileMetaData*>& inputs,
                          InternalKey* smallest, InternalKey* largest) {
  assert(!inputs.empty());
  smallest->Clear();
  largest->Clear();
  for (size_t i = 0; i < inputs.size(); i++) {
    FileMetaData* f = inputs[i];
    if (i == 0) {
      *smallest = f->smallest;
      *largest = f->largest;
    } else {
      if (icmp_.Compare(f->smallest, *smallest) < 0) {
        *smallest = f->smallest;
      }
      if (icmp_.Compare(f->largest, *largest) > 0) {
        *largest = f->largest;
      }
    }
  }
}
```

GetRange负责得到inputs中的所有文件FileMetaData的最小键与最大键；在PickCompaction函数中当level=0的时候被调用，用来确定此时刚加入input[0]的文件的键值范围，进而进一步确定在本层中（0层）是否仍存在重叠的文件。如果有，则把这些有重叠

键值的文件都加进来。这个过程的完成正是调用了current_->GetOverlappingInputs(0, &smallest, &largest, &c->inputs_[0]);这个GetOverlappingInputs这个函数。

也就是说先通过GetRange获取文件的key range，然后根据key range得到一个最全的文件列表。

4.2 GetOverlappingInputs

```
// Store in "*inputs" all files in "level" that overlap [begin,end]
void Version::GetOverlappingInputs(int level, const InternalKey* begin,
                                   const InternalKey* end,
                                   std::vector<FileMetaData*>* inputs) {
  assert(level >= 0);
  assert(level < config::kNumLevels);
  inputs->clear();
  Slice user_begin, user_end;
  if (begin != nullptr) {
    user_begin = begin->user_key();
  }
  if (end != nullptr) {
    user_end = end->user_key();
  }
  const Comparator* user_cmp = vset_->icmp_.user_comparator();
  for (size_t i = 0; i < files_[level].size();) {
    FileMetaData* f = files_[level][i++];
    const Slice file_start = f->smallest.user_key();
    const Slice file_limit = f->largest.user_key();
    if (begin != nullptr && user_cmp->Compare(file_limit, user_begin) < 0) {
      // "f" is completely before specified range; skip it
    } else if (end != nullptr && user_cmp->Compare(file_start, user_end) > 0) {
      // "f" is completely after specified range; skip it
    } else {
      inputs->push_back(f);
      if (level == 0) {
        // Level-0 files may overlap each other.  So check if the newly
        // added file has expanded the range.  If so, restart search.
        if (begin != nullptr && user_cmp->Compare(file_start, user_begin) < 0) {
          user_begin = file_start;
          inputs->clear();
          i = 0;
        } else if (end != nullptr &&
                   user_cmp->Compare(file_limit, user_end) > 0) {
          user_end = file_limit;
          inputs->clear();
          i = 0;
        }
      }
    }
  }
}
```

4.3 SetupOtherInputs

Inputs_[0]填充了第一层要参与compact的文件，接下来就是要计算下一层参与compact的文件了，记录到inputs_[1]。

```
void VersionSet::SetupOtherInputs(Compaction* c) {
```

```
  const int level = c->level();      //获取压缩的level
  InternalKey smallest, largest;
  //这里AddBoundaryInputs函数的作用是什么？？？
  AddBoundaryInputs(icmp_, current_->files_[level], &c->inputs_[0]);
  GetRange(c->inputs_[0], &smallest, &largest);  //获取当前压缩任务的输入文件的键值范
围
  //再次调用GetOverlappingInputs函数获取下一级别中，与当前压缩任务键值范围重叠的文件
  current_->GetOverlappingInputs(level + 1, &smallest, &largest,
                                 &c->inputs_[1]);
  AddBoundaryInputs(icmp_, current_->files_[level + 1], &c->inputs_[1]);
  // Get entire range covered by compaction
  InternalKey all_start, all_limit;
  GetRange2(c->inputs_[0], c->inputs_[1], &all_start, &all_limit);

  // See if we can grow the number of inputs in "level" without
  // changing the number of "level+1" files we pick up.
  if (!c->inputs_[1].empty()) {
    std::vector<FileMetaData*> expanded0;
    current_->GetOverlappingInputs(level, &all_start, &all_limit, &expanded0);
    AddBoundaryInputs(icmp_, current_->files_[level], &expanded0);
    const int64_t inputs0_size = TotalFileSize(c->inputs_[0]);
    const int64_t inputs1_size = TotalFileSize(c->inputs_[1]);
    const int64_t expanded0_size = TotalFileSize(expanded0);
    if (expanded0.size() > c->inputs_[0].size() &&
        inputs1_size + expanded0_size <
            ExpandedCompactionByteSizeLimit(options_)) {
      InternalKey new_start, new_limit;
      GetRange(expanded0, &new_start, &new_limit);
      std::vector<FileMetaData*> expanded1;
      current_->GetOverlappingInputs(level + 1, &new_start, &new_limit,
                                     &expanded1);
      AddBoundaryInputs(icmp_, current_->files_[level + 1], &expanded1);
      if (expanded1.size() == c->inputs_[1].size()) {
        Log(options_->info_log,
            "Expanding@%d %d+%d (%ld+%ld bytes) to %d+%d (%ld+%ld bytes)\n",
            level, int(c->inputs_[0].size()), int(c->inputs_[1].size()),
            long(inputs0_size), long(inputs1_size), int(expanded0.size()),
            int(expanded1.size()), long(expanded0_size), long(inputs1_size));
        smallest = new_start;
        largest = new_limit;
        c->inputs_[0] = expanded0;
        c->inputs_[1] = expanded1;
        GetRange2(c->inputs_[0], c->inputs_[1], &all_start, &all_limit);
      }
    }
  }
  // Compute the set of grandparent files that overlap this compaction
  // (parent == level+1; grandparent == level+2)
  if (level + 2 < config::kNumLevels) {
    current_->GetOverlappingInputs(level + 2, &all_start, &all_limit,
                                   &c->grandparents_);
  }
  // Update the place where we will do the next compaction for this level.
  // We update this immediately instead of waiting for the VersionEdit
```

```
  // to be applied so that if the compaction fails, we will try a different
  // key range next time.
  compact_pointer_[level] = largest.Encode().ToString();
  c->edit_.SetCompactPointer(level, largest);
}
```

总结一下这个SetupOtherInput函数的作用：

1.我们已经在PickCompaction函数中对input_[0]做了初始化，即找了一个合适的（基于seek_compaction或size_compaction两种策略）文件需要压缩的，将其加入了inputs_[0]中，这一层是level。

2. 如果level是第0层，那么在PickCompaction也对这种情况做了特殊处理，即对第0层做了遍历，将所有与之前找到的需要压缩的那一个文件有重叠的所有文件，都加入到了inputs_[0]中；

3. compaction是将level和level+1层中重叠的文件进行合并，但经过1和2过程后input_中已经包含了level层的目前需要压缩的文件总和，但这并不完整，还需要Level+1层的相应的文件，所以SetUpOtherInput这个函数需要将Level+1层需要压缩的文件挑选出来，然后加入input_[1]中；

4. 经过3步骤后，此时input_[1]中的文件键值的范围可能超过了input_[0]中文件的键值范围，比如说前者是[5,10]，后者是[3,7]，但事实上level层还有可能含有例如像[7,10]这个键值范围的文件，那么自然就需要将这个[7,10]范围内的文件再次加入到input_[0]中。即这个过程就是用input_[1]反推input_[0]是否需要扩展文件。

5. 如果需要扩展即：

(1) `expanded0.size() > c->inputs_[0].size()`

但扩展后的压缩任务不会超出预定义的字节大小限制。

(2) `inputs1_size + expanded0_size < ExpandedCompactionByteSizeLimit(options_)`

也就是说合并的文件的总量在ExpandedCompactionByteSizeLimit之内（防止过度Compact）

以上两者均成立的时候才会进一步增加参与compact的文件，更新到inputs_

6. 总结就是个来回判断的过程，再次梳理一遍：

a. 根据inputs_[0]确定inputs_[1]
b. 根据inputs_[1]反过来看下能否扩大inputs_[0]
c. inptus_[0]扩大的话，记录到expanded0
d. 根据expanded[0]看下是否会导致inputs_[1]增大
e. 如果inputs[1]没有增大，那就扩大 compact 的 level 层的文件范围

也就是：

在不增加Level+1层文件，同时不会导致compact的文件过大的前提下，尽量增加level层的文件数目。

到此，参与 compact 的文件集合就已经确定了，为了避免这些文件合并到 level + 1 层后，跟 level + 2 层有重叠的文件太多，届时合并 level + 1 和 level + 2 层压力太大，因此我们还需要记录下 level + 2 层的文件，后续 compact 时用于提前结束的判断：

```
  // Compute the set of grandparent files that overlap this compaction
  // (parent == level+1; grandparent == level+2)
  // level + 2层有overlap的文件，记录到c->grandparents_
  if (level + 2 < config::kNumLevels) {
    //level + 2层overlap的文件记录到c->grandparents_
    current_->GetOverlappingInputs(level + 2, &all_start, &all_limit,
                                   &c->grandparents_);
  }
```

接着记录compact_pointer_到c->edit_，在后续PickCompaction入口时使用。

```
  // Update the place where we will do the next compaction for this level.
  // We update this immediately instead of waiting for the VersionEdit
  // to be applied so that if the compaction fails, we will try a different
  // key range next time.
  // 记录该层本次compact的最大key
  compact_pointer_[level] = largest.Encode().ToString();
  c->edit_.SetCompactPointer(level, largest);
```

最后一步，就是返回筛选的结果c:

```
//选取一层需要compact的文件列表，及相关的下层文件列表，记录在Compaction*
Compaction* VersionSet::PickCompaction() {
  Compaction* c;
  ...
  return c;
}
```

再来理一理Compaction的作用:

在LevelDB中，随着数据的不断写入，Level 0的sstable文件数量会迅速增加，导致读取性能下降。压缩的主要任务是将多个SStable文件合并为一个更大的文件，并将数据从较低级别移动到较高级别，从而优化存储结构，提升读取效率，并回收磁盘空间。

随着数据不断写入，Level 0可能积累大量文件，导致读取性能下降。

LevelDB笔记22：再谈Compaction

2024年10月11日　16:13

compaction — leveldb-handbook 文档

本质：

（1）内部数据重合整合的机制;
（2）平衡读写速率的有效手段。

作用：

（1）数据持久化：minor compaction（一次内存数据持久化的过程产出一个0层SSTable文件）；

（2）提高读写效率：

写效率相当高（？？？想想为什么），读操作0层各个文件中的数据可以重叠，所以读取要遍历0层所有的文件。如果不进行compaction，那么持续不断的写入，会导致0层文件越来越多，而读操作也愈发耗时。因此有了major compaction的过程。

（3）平衡读写差异：

一次major compaction的过程本质上是一个多路归并的过程，既有大量的磁盘读开销，也有大量的磁盘写开销，显然这是一个严重的性能瓶颈。

但当用户写入的速度始终大于major compaction的速度时，就会导致0层的文件数量还是不断上升，用户的读取效率持续下降。所以LevelDB中规定：

- 当0层文件数量超过`SlowdownTrigger`时，写入的速度主要减慢；
- 当0层文件数量超过`PauseTrigger`时，写入暂停，直至Major Compaction完成；

故compaction也可以起到平衡读写差异的作用。

到这里，也就可以理解在笔记8中的`Status DBImpl::MakeRoomForWrite(bool force)`这个过程的那几个if语句了，为什么要超过一定阈值后，暂停写入的操作等待compaction的过程了。

（3）整理数据：

LevelDB的每一条数据项都有一个版本信息，标识着这条数据的新旧程度。这也就意味着同样的一个key，在LevelDB中可能存在着多条数据项，且每个数据项包含了不同版本的内容。

为了尽量减少数据集所占用的磁盘空间大小，LevelDB在major compaction的过程中，对不同版本的数据项进行合并。

接下来详细的Minor Compaction和Major Compaction可以查看compaction — leveldb-handbook 文档该链接和前几篇LevelDB有关Compaction内容的笔记。

触发LevelDB进行major compaction的条件：

（1）当0层文件超过预定的上限（个数，默认为4个）
（2）当level i层文件的总大小超过（10^i）MB
（3）当某个文件无效读取的次数过多

由于第0层存在过多的文件数，且存在重叠数据，所以读取时需要遍历所有的0层文件，过多的0层文件数，会导致读取效率的下降。因此compaction的一个目的是为了**提高读取的效率。**

对于i>=1层的sstable文件，相应层的文件是不重叠有序的，因此每层读取时最多会定位到一个sstable文件进行读取该文件即可，因此本身对于读取效率的影响不会太大。针对于这部分数据发生compaction的条件，**从提升读取效率转变成了降低compaction的IO开销。**为什么这样说呢?

假设levelDB的合并策略只有一条，即当0层文件个数超过阈值时进行与1层文件进行合并，那么这样下去会导致1层文件的个数越来越多或者总的数据量越来越大，而**通常一次合并时，0层文件的key的取值范围是很大的，导致每一次0层文件与1层文件进行合并时，1层文件输入文件的总数据量非常庞大。**所以不仅需要控制0层文件的个数，同样，每一层文件的总大小也需要进行控制，是的每次进行compaction时，IO开销尽量保持常量。

故LevelDB规定，1层文件总大小上限为10MB，２层为100MB，以此类推，最高层（7层）没有限制。

以上两个机制能够保证随着合并的进行，数据时严格下沉的，但仍然存在一个问题：**如果0层文件完成合并之后，1层文件同时达到了数据上限，同时需要合并，甚至于情况更糟糕更高层2-n层同时达到数据上限都需要进行合并。**

其中的一种优化机制是：

- source层的文件个数只有一个；
- source层文件与source+1层文件没有重叠；
- source层文件与source+2层的文件重叠部分不超过10个文件；

当满足这三个条件的时候，可以将source层的该文件直接移至source+1层。但是很明显这个条件很苛刻，还是无法解决上述问题。为了避免可能存在这种巨大的合并开销，LevelDB引入了第三个机制：“**错峰合并**”。

那么**（1）如何找寻这种适合错峰合并的文件（2）以及如何判断哪个时机是适合进行错峰合并的呢?**

对于问题（1）Leveldb的作者认为，一个文件一次查询的开销为10ms, 若某个文件的查询次数过多，且查询在该文件中不命中, 那么这种行为就可以视为无效的查询开销，这种文件就可以进行错峰合并。

对于问题（2），对于一个1MB的文件，对其合并的开销为25ms。因此当一个文件1MB的文件无效查询超过25次时，便可以对其进行合并。

对于一个1MB的文件，其合并开销为（1）source层1MB的文件读取，（2）source+1层 10-12MB的文件读取（3）source+1层10-12MB的文件写入。

故1MB的文件compact需要花费（1+12+12）*10ms=250ms，10ms是1MB的读/写所花费的时间。而1次seek花费10ms，因此1MB的compact时间=25次seek时间=250ms

也就是说40KB文件的compact时间= 1次seek时间，保守点取16KB，即t=16KB的compaction时间=1次seek的时间。

总结25MB的文件IO开销，除以100MB / s的文件IO速度，估计开销为25ms。

也就是说，我们肯定不希望一个文件被多次的seek且又在其中找不到相应的数据，换个角度想，虽然一次Major Compaction会花费一定的时间（比一次seek的时间长），但是如果多次seek都查找不到，甚至于多次seek的耗费时间大于了一次compaction的时间，那么还不如把这个文件compact了，这样还能使得减少seek时对该文件的无效seek次数。

采样探测

事实上这个地方在seek_compaction的笔记20中已经叙述过了。

16KB数据1次compaction的时间=1次seek的时间，故allow_seek= file_size/16K；用来记录这个文件可以允许的最大的seek次数。

若seek过程中，该在文件中访问命中，则不做任何处理；若在该文件中访问不命中，则对该文件的allow_seek标志做减一操作。

如果allow_seek减少到0，触发对该文件的错峰合并。

Compaction过程

整个compaction可以简单的分为以下几步：

1. 寻找合适的输入文件
2. 根据key重叠情况扩大输入文件集合；
3. 多路合并；
4. 积分计算。

寻找输入文件：

不同情况下发起的合并动作，其初始的输入文件不同。

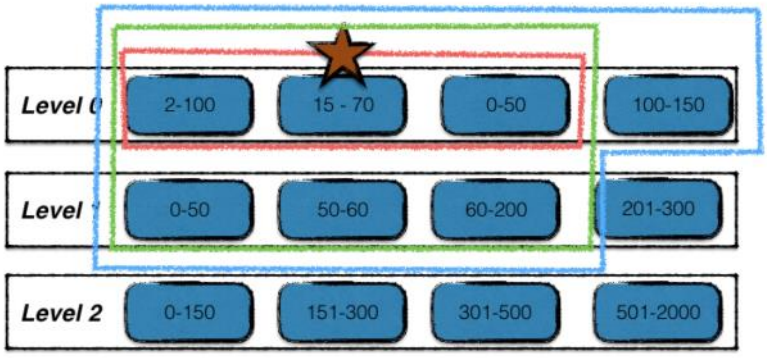
（1）对于***Level 0层文件数过多引发的合并场景或由于level i层文件总量过大的合并场景***，采用轮转的方法选择起始输入文件，记录了上一次该层合并的文件的最大key，下一次则选择在此key之后的首个文件。

（2）对于***错峰合并***，起始输入文件则为该查询次数过多的文件。

扩大输入文件集合：

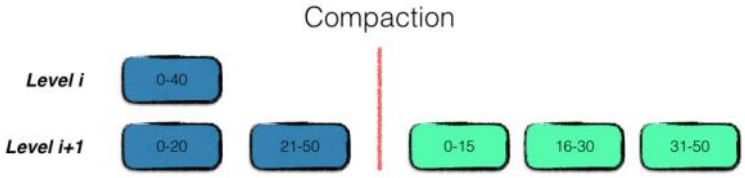
主要有如下四个过程：

1. 红星标注的为起始输入文件；
2. 在level i层中，查找与起始输入文件有key重叠的文件，如图中红线所标注，最终构成level i层的输入文件；
3. 利用level i层的输入文件，在level i+1层找寻有key重叠的文件，结果为绿线标注的文件，构成level i，i+1层的输入文件；
4. 最后利用两层的输入文件，在不扩大level i+1输入文件的前提下，查找level i层的有key重叠的文件，结果为蓝线标准的文件，构成最终的输入文件；



多路合并：

将Level i层的文件，与level i+1 层的文件中的数据项，按序整理之后，输出到level i+1层的若干个新文件中，即合并完成



注意在整理的过程中，需要将冗余的数据进行清理，即**同一条数据的多个版本信息，只保留最新的那一份。**

但是要注意，**某些仍然在使用的旧版本的数据，在此时不能立刻删除**，而得等到用户使用结束，释放句柄后，根据引用计数来进行清除。

积分计算：

每一次compaction都会消除若干source层的旧文件，新增source+1层的新文件，因此触发进行合并的条件状态可能也发生了变化。

故在LevelDB中，使用了计分牌来维护每一层文件的文件个数及数据总量信息，来**挑选出下一个需要进行合并的层数。**

计分的规则很简单：

（1）对于0层文件，该层的分数为文件总数 / 4；

（2）对于非0层文件，该层的分数为文件数据总量 / 数据总量上限；

将得分最高的层数记录，若该得分超过1，则为下一次进行合并的层数；

用户行为

由于leveldb内部进行compaction时有trivial move优化，且根据内部的文件格式组织，用户在使用leveldb时，可以尽量将大批量需要写入的数据进行预排序，利用空间局部性，尽量减少多路合并的IO开销。

LevelDB笔记23：major compaction

2024年10月17日 16:52

Minor compaction主要解决内存数据持久化磁盘。Major compaction则负责将磁盘上的数据合并，每合并一次，数据就落到更低一层。

1 major compaction的作用
随着数据合并到更大的level，一个明显的好处就是清理冗余数据。
如果不同level的sst文件里，存在相同的key，那么更底层的数据就可以删除不再保留（不考虑snapshot的情况下）。
为了充分利用磁盘高性能的顺序写，删除数据也是顺序写入删除标记，而真正删除数据，也是在major compact的过程中的。
所以，一个作用是能够节省磁盘空间。
Level 0的数据之间是无序的，每次查找都需要遍历所有可能重叠的文件，而归并到level 1之后，数据变得有序，带查找的文件数目变少。
所以另外一个作用是能够提高读效率。

事实上到最后调用的是DBImpl ::BackgroundCompaction（）这个过程

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

//(1)Memtable压缩，**Minor Compaction**
//如果是不可变的Memtable，调用CompactMemtable（）进行压缩，即minor Compaction的过程

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

//(2)文件压缩

```
  Compaction* c;
  bool is_manual = (manual_compaction_ != nullptr);
  InternalKey manual_end;
  //手动压缩
  if (is_manual) {
    ManualCompaction* m = manual_compaction_;
    c = versions_->CompactRange(m->level, m->begin, m->end);
    m->done = (c == nullptr);
    if (c != nullptr) {
      manual_end = c->input(0, c->num_input_files(0) - 1)->largest;
    }
    Log(options_.info_log,
        "Manual compaction at level-%d from %s .. %s; will stop at %s\n",
        m->level, (m->begin ? m->begin->DebugString().c_str() : "(begin)"),
        (m->end ? m->end->DebugString().c_str() : "(end)"),
        (m->done ? "(end)" : manual_end.DebugString().c_str()));
```

```
} else {
//自动压缩
  c = versions_->PickCompaction();
}


Status status;
if (c == nullptr) {
  // Nothing to do
} else if (!is_manual && c->IsTrivialMove()) {
  // Move file to next level
  assert(c->num_input_files(0) == 1);
  FileMetaData* f = c->input(0, 0);
  c->edit()->RemoveFile(c->level(), f->number);
  c->edit()->AddFile(c->level() + 1, f->number, f->file_size, f->smallest,
                     f->largest);
  status = versions_->LogAndApply(c->edit(), &mutex_);
  if (!status.ok()) {
    RecordBackgroundError(status);
  }
  VersionSet::LevelSummaryStorage tmp;
  Log(options_.info_log, "Moved #%lld to level-%d %lld bytes %s: %s\n",
      static_cast<unsigned long long>(f->number), c->level() + 1,
      static_cast<unsigned long long>(f->file_size),
      status.ToString().c_str(), versions_->LevelSummary(&tmp));
} else {
  CompactionState* compact = new CompactionState(c);
  status = DoCompactionWork(compact);
  if (!status.ok()) {
    RecordBackgroundError(status);
  }
  CleanupCompaction(compact);
  c->ReleaseInputs();
  RemoveObsoleteFiles();
}
delete c;
```

//压缩结果处理，一般是错误处理

```
if (status.ok()) {
  // Done
} else if (shutting_down_.load(std::memory_order_acquire)) {
  // Ignore compaction errors found during shutting down
} else {
  Log(options_.info_log, "Compaction error: %s", status.ToString().c_str());
}
```

//手动压缩结束处理

```
if (is_manual) {
  ManualCompaction* m = manual_compaction_;
  if (!status.ok()) {
    m->done = true;
  }
  if (!m->done) {
    // We only compacted part of the requested range.  Update *m
    // to the range that is left to be compacted.
```

```
      m->tmp_storage = manual_end;
      m->begin = &m->tmp_storage;
    }
    manual_compaction_ = nullptr;
  }
}
```

在之前的笔记中，已经讲了有关CompactMemTable（minor compact）和PickCompaction这两个过程，后者返回一个Compaction类型的结构体指针。记录了挑选出来的合适的需要进行major compact操作的所有文件。在c->inputs_[2]这个数组中存储。这个内容参见笔记22筛选文件。

```
  //自动压缩
    c = versions_->PickCompaction();
```

接下来就是major compact的核心过程了。如下所示：

```
Status status;
if (c == nullptr) {
  // Nothing to do
} else if (!is_manual && c->IsTrivialMove()) {…
} else {…
delete c;
```

LevelDB笔记24：Iterator总结

2024年10月17日 16:44