LevelDB 源码分析「八、完整流程」

2019.09.06 SF-Zhou

本系列之前的数篇博文从面向对象的角度分析了 LevelDB 中的核心组件，理解了每个类的作用。本篇将从面向过程的角度，分析 LevelDB 创建、打开和读写的完整流程。

1. 新建数据库并写入

```
#include <cassert>
#include "leveldb/db.h"

int main() {
  leveldb::DB* db;

  {
    leveldb::Options options;
    options.create_if_missing = true;
    options.compression = leveldb::kNoCompression;
    leveldb::Status status = leveldb::DB::Open(options, "testdb", &db);
    assert(status.ok());
  }

  {
    leveldb::WriteOptions options;
    leveldb::Status status = db->Put(options, "[Key]", "[Value]");
    assert(status.ok());
```

```
  }

  delete db;
}
```

确保当前路径下没有 testdb 目录，然后执行上方的代码。可以在写入前加入 getchar() 中断以观察中间的状态，测试得到的中间状态如下：

```
❱ ls -l
total 24
-rw-r--r--  1 sfzhou  staff     0B Sep  7 10:02 000003.log
-rw-r--r--  1 sfzhou  staff    16B Sep  7 10:02 CURRENT
-rw-r--r--  1 sfzhou  staff     0B Sep  7 10:02 LOCK
-rw-r--r--  1 sfzhou  staff    56B Sep  7 10:02 LOG
-rw-r--r--  1 sfzhou  staff    50B Sep  7 10:02 MANIFEST-000002
```

待写入完成后，文件 000003.log 的大小变为 34，其余大小不变。

接下来沿着代码一步一步分析，这里推荐使用 VS Code 查看函数实现。首先来看 leveldb::DB::Open：

```
Status DB::Open(const Options& options, const std::string& dbname, DB** dbptr) {
  *dbptr = nullptr;

  DBImpl* impl = new DBImpl(options, dbname);
  impl->mutex_.Lock();
  VersionEdit edit;
  // Recover handles create_if_missing, error_if_exists
```

```
// 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_);
}
if (s.ok()) {
  impl->DeleteObsoleteFiles();
  impl->MaybeScheduleCompaction();
}
impl->mutex_.Unlock();
if (s.ok()) {
  assert(impl->mem_ != nullptr);
  *dbptr = impl;
```

```
  } else {
    delete impl;
  }
  return s;
}
```

先给输出参数 *dbptr 赋空指针，然后 new 一个 DBImpl 对象。 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_)) {}
```

都是成员变量的初始化，大部分成员置为 0。值得关注的 SanitizeOptions 实现如下：

```
template <class T, class V>
static void ClipToRange(T* ptr, V minvalue, V maxvalue) {
  if (static_cast<V>(*ptr) > maxvalue) *ptr = maxvalue;
  if (static_cast<V>(*ptr) < minvalue) *ptr = minvalue;
}
Options SanitizeOptions(const std::string& dbname,
                        const InternalKeyComparator* icmp,
                        const InternalFilterPolicy* ipolicy,
                        const Options& src) {
  Options result = src;
  result.comparator = icmp;
  result.filter_policy = (src.filter_policy != nullptr) ? ipolicy : nullptr;
  ClipToRange(&result.max_open_files, 64 + kNumNonTableCacheFiles, 50000);
  ClipToRange(&result.write_buffer_size, 64 << 10, 1 << 30);
  ClipToRange(&result.max_file_size, 1 << 20, 1 << 30);
  ClipToRange(&result.block_size, 1 << 10, 4 << 20);
  if (result.info_log == nullptr) {
    // Open a log file in the same directory as the db
    src.env->CreateDir(dbname);  // In case it does not exist
    src.env->RenameFile(InfoLogFileName(dbname), OldInfoLogFileName(dbname));
    Status s = src.env->NewLogger(InfoLogFileName(dbname), &result.info_log);
    if (!s.ok()) {
```

```
      // No place suitable for logging
      result.info_log = nullptr;
    }
  }
  if (result.block_cache == nullptr) {
    result.block_cache = NewLRUCache(8 << 20);
  }
  return result;
}
```

SanitizeOptions 函数将原 src 中的属性进行了安全裁剪，并且创建了 dbname 目录、将原 Info Log 文件重命名并创建了新的 Info Log 文件，也就是 testdb/LOG 文件。最后按需创建了 block cache 并返回作为 DBImpl 的 options_。 DBImpl 的构造函数还为 table_cache_、 tmp_batch_ 和 version_ 创建了新对象。

回到 leveldb::DB::Open，创建 DBImpl 后上锁并执行 impl->Recover：

```
Status DBImpl::Recover(VersionEdit* edit, bool* save_manifest) {
  mutex_.AssertHeld();

  // Ignore error from CreateDir since the creation of the DB is
  // committed only when the descriptor is created, and this directory
  // may already exist from a previous failed creation attempt.
  env_->CreateDir(dbname_);
  assert(db_lock_ == nullptr);
  Status s = env_->LockFile(LockFileName(dbname_), &db_lock_);
  if (!s.ok()) {
    return s;
  }
```

```
  if (!env_->FileExists(CurrentFileName(dbname_))) {
    if (options_.create_if_missing) {
      s = NewDB();
      if (!s.ok()) {

        return s;
      }
    } else {
      return Status::InvalidArgument(
          dbname_, "does not exist (create_if_missing is false)");
    }
  } else {
    if (options_.error_if_exists) {
      return Status::InvalidArgument(dbname_,
                                     "exists (error_if_exists is true)");
    }
  }

  s = versions_->Recover(save_manifest);
  if (!s.ok()) {
    return s;
  }

  ...

  return Status::OK();
}
```

尝试创建 dbname 目录且忽略错误，启用文件锁 testdb/LOCK 以阻止其他进程操作当

前数据库。由于当前没有 CURRENT 文件，并且我们开启了 options.create_if_missing，这里会继续调用 NewDB 函数：

```
Status DBImpl::NewDB() {
  VersionEdit new_db;
  new_db.SetComparatorName(user_comparator()->Name());
  new_db.SetLogNumber(0);
  new_db.SetNextFile(2);
  new_db.SetLastSequence(0);

  const std::string manifest = DescriptorFileName(dbname_, 1);
  WritableFile* file;
  Status s = env_->NewWritableFile(manifest, &file);
  if (!s.ok()) {
    return s;
  }
  {
    log::Writer log(file);
    std::string record;
    new_db.EncodeTo(&record);
    s = log.AddRecord(record);
    if (s.ok()) {
      s = file->Close();
    }
  }
  delete file;
  if (s.ok()) {
    // Make "CURRENT" file that points to the new manifest file.
    s = SetCurrentFile(env_, dbname_, 1);
  } else {
```

```
    env_->DeleteFile(manifest);
  }
  return s;
}
```

NewDB 中创建了一个新的 VersionEdit 对象，将日志编号设为 0，MANIFEST 编号设为 1，NextFile 编号自然是 2 了。而后将 VersionEdit 对象转为日志 Record 写入 MANIFEST 文件中，并将 CURRENT 指向该 MANIFEST。

ちょっと待って（桥豆麻袋），先前观察到的 MANIFEST 文件名为 MANIFEST-000002，和这里分析的编号为 1 并不相符，应该是中间发生了什么。回到 DBImpl::Recover 继续看，接下来将会执行 versions_->Recover，该函数在上一篇文章中有贴出源码，可以翻回去看一下这里不贴了。简述其过程：根据 CURRENT 文件，读取指向的 MANIFEST 文件中的 VersionEdit 记录，并合成 Version。贴一部分与编号相关的代码：

```
Version* v = new Version(this);
builder.SaveTo(v);
// Install recovered version
Finalize(v);
AppendVersion(v);
manifest_file_number_ = next_file;
next_file_number_ = next_file + 1;
last_sequence_ = last_sequence;
log_number_ = log_number;
prev_log_number_ = prev_log_number;
```

```
// See if we can reuse the existing MANIFEST file.
if (ReuseManifest(dscname, current)) {
  // No need to save new manifest
} else {


  *save_manifest = true;
}
```

可以看到这里将 manifest_file_number_ 设为了 next_file，也就是 2。Options::reuse_logs 默认为关闭状态，故这里会将 save_manifest 设为 true，所以后面保存 MANIFEST 时编号就是 2 了。

回到 DBImpl::Recover，当 versions_->Recover 执行完成后，会读取当前存在的日志文件。而新数据库并没有日志，中间过程就跳过、直接返回了，进而回到 leveldb::DB::Open：

```
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_);
}
```

impl->mem_ 依然保持为空，故申请新的日志编号 3，创建日志文件和对应的内存数据库，也就是 testdb/000003.log 。 save_manifest 为真，则调用 version_->LogAndApply 将当前版本 edit 写入文件，也就是最终看到的 testdb/MANIFEST-000002 。这里硬核一点，直接看下 MANIFEST 文件的内容：

```
❱ hexdump MANIFEST-000002
0000000 56 f9 b8 f8 1c 00 01 01 1a 6c 65 76 65 6c 64 62
0000010 2e 42 79 74 65 77 69 73 65 43 6f 6d 70 61 72 61
0000020 74 6f 72 a4 9c 8b be 08 00 01 02 03 09 00 03 04
0000030 04 00
```

version_->LogAndApply 创建 MANIFEST 文件后，会先执行 VersionSet::WriteSnapshot：

```
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
  for (int level = 0; level < config::kNumLevels; level++) {
    if (!compact_pointer_[level].empty()) {
      InternalKey key;
      key.DecodeFrom(compact_pointer_[level]);
      edit.SetCompactPointer(level, key);
    }
  }

  // Save files
  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);
}
```

这里首先会将比较器的名字作为 Record 写入 MANIFEST 文件中。根据本系列第三篇博文中分析的日志记录方式，每一条 Record 会有 7 字节的 Header，其中前 4 字节为 CRC 校验值可以不理会，5、6 字节为记录的长度，这里是 0x1c = 28，最后是 Record 类

型，这里的类型是 kFullType = 1 。而后的 28 字节便是记录了比较器名字的 edit 对象，其内容需要根据编码的方式进行解码：

```
// Tag numbers for serialized VersionEdit.  These numbers are written to
// disk and should not be changed.
enum Tag {
  kComparator = 1,
  kLogNumber = 2,
  kNextFileNumber = 3,
  kLastSequence = 4,
  kCompactPointer = 5,
  kDeletedFile = 6,
  kNewFile = 7,
  // 8 was used for large value refs
  kPrevLogNumber = 9
};

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);
```

```
  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());
  }
}
```

```
01 1a 6c 65 76 65 6c 64 62 2e 42 79 74 65 77 69 73 65 43 6f 6d 70 61 72 61 74 6f 72
```

这里的 kComparator=1，而后是比较器的长度 0x1a=26 和名字对应的 ASCII 码，翻译过来就是 leveldb.BytewiseComparator。之后的第二条记录便是 leveldb::DB:OpenDB 中的 edit 对象的 Record 记录：

```
a4 9c 8b be 08 00 01 02 03 09 00 03 04 04 00
```

一样的 7 字节 Header，Record 长度为 8，具体内容为：

1. kLogNumber=02，对应的日志编号为 3；
2. kPrevLogNumber=09，对应的上一个日志编号为 0；
3. kNextFileNumber=03，对应的下一个文件编号为 4；
4. kLastSequence=04，对应的最新序列号为 0。

最后回到 leveldb::DB::OpenDB：

```
if (s.ok()) {
  impl->DeleteObsoleteFiles();
  impl->MaybeScheduleCompaction();
}
```

创建或恢复成功后，执行 DBImpl::DeleteObsoleteFiles 删除废弃文件：

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

  if (!bg_error_.ok()) {
    // After a background error, we don't know whether a new version may
    // or may not have been committed, so we cannot safely garbage collect.
    return;
  }

  // Make a set of all of the live files
  std::set<uint64_t> live = pending_outputs_;
  versions_->AddLiveFiles(&live);

  std::vector<std::string> filenames;
  env_->GetChildren(dbname_, &filenames);  // Ignoring errors on purpose
  uint64_t number;
  FileType type;
  std::vector<std::string> files_to_delete;
  for (std::string& filename : filenames) {
    if (ParseFileName(filename, &number, &type)) {
      bool keep = true;
      switch (type) {
        case kLogFile:
          keep = ((number >= versions_->LogNumber()) ||
                  (number == versions_->PrevLogNumber()));
          break;
        case kDescriptorFile:
          // Keep my manifest file, and any newer incarnations'
          // (in case there is a race that allows other incarnations)
```

```
          keep = (number >= versions_->ManifestFileNumber());
          break;
        case kTableFile:
          keep = (live.find(number) != live.end());
          break;

        case kTempFile:
          // Any temp files that are currently being written to must
          // be recorded in pending_outputs_, which is inserted into "live"
          keep = (live.find(number) != live.end());
          break;
        case kCurrentFile:
        case kDBLockFile:
        case kInfoLogFile:
          keep = true;
          break;
      }

      if (!keep) {
        files_to_delete.push_back(std::move(filename));
        if (type == kTableFile) {
          table_cache_->Evict(number);
        }
        Log(options_.info_log, "Delete type=%d #%lld\n", static_cast<int>(type),
            static_cast<unsigned long long>(number));
      }
    }
  }

  // While deleting all files unblock other threads. All files being deleted
  // have unique names which will not collide with newly created files and
```

```
  // are therefore safe to delete while allowing other threads to proceed.
  mutex_.Unlock();
  for (const std::string& filename : files_to_delete) {
    env_->DeleteFile(dbname_ + "/" + filename);
  }

  mutex_.Lock();
}
```

这里的废弃文件也包括创建不久的 testdb/MANIFEST-000001 。至此 leveldb::DB::Open 函数分析完毕。

而后执行的写入操作 db->Put，会将数据写入日志文件和内存数据库。此时使用的日志文件编号为 3，也就是 testdb/000003.db，其写入后的内容为：

```
❱ hexdump 000003.log
0000000 aa a0 87 24 1b 00 01 01 00 00 00 00 00 00 00 01
0000010 00 00 00 01 05 5b 4b 65 79 5d 07 5b 56 61 6c 75
0000020 65 5d
```

一样的 7 字节 Header，Record 长度为 0x1b=27，内容为 WriteBatch 编码结果。参考本系列第三篇博文，内容的前 8 字节为序列号，这里是 1；其后的 4 字节为键值对数量，这里也是 1；再后面就是附带长度编码的键值对，分别是 [Key] 和 [Value]。最后代码结束，删除了 db 对象，也就得到了前文叙述的文件状态。

2. 打开数据库并读取

```cpp
#include <cassert>
#include <iostream>
#include "leveldb/db.h"

int main() {
  leveldb::DB* db;

  {
    leveldb::Options options;
    options.compression = leveldb::kNoCompression;
    leveldb::Status status = leveldb::DB::Open(options, "testdb", &db);
    assert(status.ok());
  }

  {
    std::string key = "[Key]";
    std::string value;
    leveldb::ReadOptions read_options;
    leveldb::Status status = db->Get(read_options, key, &value);
    assert(status.ok());
    std::cout << "Key: " << key << ", Value: " << value << std::endl;
  }

  delete db;
}
```

在上一节创建的数据库的基础上，执行上方的代码。首先打开数据库，再读取 Key 对应的 Value。最终的数据库文件状态为：

```
❱ ls -l
```

```
total 40
-rw-r--r--  1 sfzhou  staff   124B Sep  7 09:57 000005.ldb
-rw-r--r--  1 sfzhou  staff     0B Sep  7 09:57 000006.log
-rw-r--r--  1 sfzhou  staff    16B Sep  7 09:57 CURRENT
-rw-r--r--  1 sfzhou  staff     0B Sep  7 09:57 LOCK

-rw-r--r--  1 sfzhou  staff   304B Sep  7 09:57 LOG
-rw-r--r--  1 sfzhou  staff    56B Sep  7 09:57 LOG.old
-rw-r--r--  1 sfzhou  staff    82B Sep  7 09:57 MANIFEST-000004
```

和之前一样，沿着代码分析打开数据库的流程。DB::Open 前期的步骤一致，直接跳到 impl->Recover。由于存在 CURRENT 文件，所以就跳过了 NewDB 的步骤。随后依然是 versions_->Recover，读取 CURRENT 文件、继而读取 MANIFEST。如上一节中所分析的，LogNumber 为 3，NextFileNumber 为 4，故最后新建的 MANIFEST 文件编号为 4。随后回到 impl->Recover 执行恢复日志文件数据：

```
Status DBImpl::Recover(VersionEdit* edit, bool* save_manifest) {
  ...

  SequenceNumber max_sequence(0);

  // Recover from all newer log files than the ones named in the
  // descriptor (new log files may have been added by the previous
  // incarnation without registering them in the descriptor).
  //
  // Note that PrevLogNumber() is no longer used, but we pay
  // attention to it in case we are recovering a database
  // produced by an older version of leveldb.
  const uint64_t min_log = versions_->LogNumber();
```

```
const uint64_t prev_log = versions_->PrevLogNumber();
std::vector<std::string> filenames;
s = env_->GetChildren(dbname_, &filenames);
if (!s.ok()) {
  return s;

}
std::set<uint64_t> expected;
versions_->AddLiveFiles(&expected);
uint64_t number;
FileType type;
std::vector<uint64_t> logs;
for (size_t i = 0; i < filenames.size(); i++) {
  if (ParseFileName(filenames[i], &number, &type)) {
    expected.erase(number);
    if (type == kLogFile && ((number >= min_log) || (number == prev_log)))
      logs.push_back(number);
  }
}
if (!expected.empty()) {
  char buf[50];
  snprintf(buf, sizeof(buf), "%d missing files; e.g.",
           static_cast<int>(expected.size()));
  return Status::Corruption(buf, TableFileName(dbname_, *(expected.begin())));
}

// Recover in the order in which the logs were generated
std::sort(logs.begin(), logs.end());
for (size_t i = 0; i < logs.size(); i++) {
  s = RecoverLogFile(logs[i], (i == logs.size() - 1), save_manifest, edit,
                     &max_sequence);
```

```
    if (!s.ok()) {
      return s;
    }

    // The previous incarnation may not have written any MANIFEST

    // records after allocating this log number.  So we manually
    // update the file number allocation counter in VersionSet.
    versions_->MarkFileNumberUsed(logs[i]);
  }

  if (versions_->LastSequence() < max_sequence) {
    versions_->SetLastSequence(max_sequence);
  }

  return Status::OK();
}
```

搜索数据库目录下的、符合条件的日志文件，然后执行 DBImpl::RecoverLogFile 进行恢复：

```
Status DBImpl::RecoverLogFile(uint64_t log_number, bool last_log,
                              bool* save_manifest, VersionEdit* edit,
                              SequenceNumber* max_sequence) {
  struct LogReporter : public log::Reader::Reporter {
    Env* env;
    Logger* info_log;
    const char* fname;
    Status* status;  // null if options_.paranoid_checks==false
    void Corruption(size_t bytes, const Status& s) override {
```

```
    Log(info_log, "%s%s: dropping %d bytes; %s",
        (this->status == nullptr ? "(ignoring error) " : ""), fname,
        static_cast<int>(bytes), s.ToString().c_str());
    if (this->status != nullptr && this->status->ok()) *this->status = s;
  }
};

mutex_.AssertHeld();

// Open the log file
std::string fname = LogFileName(dbname_, log_number);
SequentialFile* file;
Status status = env_->NewSequentialFile(fname, &file);
if (!status.ok()) {
  MaybeIgnoreError(&status);
  return status;
}

// Create the log reader.
LogReporter reporter;
reporter.env = env_;
reporter.info_log = options_.info_log;
reporter.fname = fname.c_str();
reporter.status = (options_.paranoid_checks ? &status : nullptr);
// We intentionally make log::Reader do checksumming even if
// paranoid_checks==false so that corruptions cause entire commits
// to be skipped instead of propagating bad information (like overly
// large sequence numbers).
log::Reader reader(file, &reporter, true /*checksum*/, 0 /*initial_offset*/);
Log(options_.info_log, "Recovering log #%llu",
```

```
    (unsigned long long)log_number);

// Read all the records and add to a memtable
std::string scratch;
Slice record;

WriteBatch batch;
int compactions = 0;
MemTable* mem = nullptr;
while (reader.ReadRecord(&record, &scratch) && status.ok()) {
  if (record.size() < 12) {
    reporter.Corruption(record.size(),
                        Status::Corruption("log record too small"));
    continue;
  }
  WriteBatchInternal::SetContents(&batch, record);

  if (mem == nullptr) {
    mem = new MemTable(internal_comparator_);
    mem->Ref();
  }
  status = WriteBatchInternal::InsertInto(&batch, mem);
  MaybeIgnoreError(&status);
  if (!status.ok()) {
    break;
  }
  const SequenceNumber last_seq = WriteBatchInternal::Sequence(&batch) +
                                  WriteBatchInternal::Count(&batch) - 1;
  if (last_seq > *max_sequence) {
    *max_sequence = last_seq;
  }
```

```
  if (mem->ApproximateMemoryUsage() > options_.write_buffer_size) {
    compactions++;
    *save_manifest = true;
    status = WriteLevel0Table(mem, edit, nullptr);

    mem->Unref();
    mem = nullptr;
    if (!status.ok()) {
      // Reflect errors immediately so that conditions like full
      // file-systems cause the DB::Open() to fail.
      break;
    }
  }
}

delete file;

// See if we should keep reusing the last log file.
if (status.ok() && options_.reuse_logs && last_log && compactions == 0) {
  assert(logfile_ == nullptr);
  assert(log_ == nullptr);
  assert(mem_ == nullptr);
  uint64_t lfile_size;
  if (env_->GetFileSize(fname, &lfile_size).ok() &&
      env_->NewAppendableFile(fname, &logfile_).ok()) {
    Log(options_.info_log, "Reusing old log %s \n", fname.c_str());
    log_ = new log::Writer(logfile_, lfile_size);
    logfile_number_ = log_number;
    if (mem != nullptr) {
      mem_ = mem;
```

```
          mem = nullptr;
        } else {
          // mem can be nullptr if lognum exists but was empty.
          mem_ = new MemTable(internal_comparator_);
          mem_->Ref();

        }
      }
    }

    if (mem != nullptr) {
      // mem did not get reused; compact it.
      if (status.ok()) {
        *save_manifest = true;
        status = WriteLevel0Table(mem, edit, nullptr);
      }
      mem->Unref();
    }

    return status;
  }
```

DBImpl::RecoverLogFile 将日志文件中的数据读取到内存数据库中，同时更新序列号，最后将内存数据库中的数据写入 Level0 的 Sorted Table 里，也就是最终状态里的 testdb/000005.ldb 。执行完成后回到 DBImpl::Recover，再回到 DB::Open 里。再后面就与上一节中的行为一致了。

最后来看下 MANIFEST 的内容：

```
❱ hexdump MANIFEST-000004
0000000 56 f9 b8 f8 1c 00 01 01 1a 6c 65 76 65 6c 64 62
0000010 2e 42 79 74 65 77 69 73 65 43 6f 6d 70 61 72 61
0000020 74 6f 72 9e bc a9 67 28 00 01 02 06 09 00 03 07
0000030 04 01 07 00 05 7c 0d 5b 4b 65 79 5d 01 01 00 00

0000040 00 00 00 00 0d 5b 4b 65 79 5d 01 01 00 00 00 00
0000050 00 00
```

前面的一段比较器名和上一节中的一致，直接来看第二条记录，也就是第 0x23 个字节开始看。4 字节 CRC，忽略；记录长度为 0x28=40，后面依旧是 VersionEdit 的编码结果：

1. kLogNumber=02，对应的日志编号为 6；
2. kPrevLogNumber=09，对应的上一个日志编号为 0；
3. kNextFileNumber=03，对应的下一个文件编号为 7；
4. kLastSequence=04，对应的最新序列号为 1；
5. kNewFile=07，对应新增文件，其 Level 为 0，编号为 5，文件大小 0x7c=124，最小键和最大键都是长度为 0x0d = 13 的 5b 4b 65 79 5d 01 01 00 00 00 00 00 00，格式为 Internal Key。Internal Key 后面的 8 个字节为综合序列号，对应 kTypeValue=1，序列号为 1。

至此数据库打开的流程分析完毕。接下来看 leveldb::DB::Get：

```
Status DBImpl::Get(const ReadOptions& options, const Slice& key,
                   std::string* value) {
  Status s;
```

```
MutexLock l(&mutex_);
SequenceNumber snapshot;
if (options.snapshot != nullptr) {
  snapshot =
      static_cast<const SnapshotImpl*>(options.snapshot)->sequence_number();
} else {
  snapshot = versions_->LastSequence();
}

MemTable* mem = mem_;
MemTable* imm = imm_;
Version* current = versions_->current();
mem->Ref();
if (imm != nullptr) imm->Ref();
current->Ref();

bool have_stat_update = false;
Version::GetStats stats;

// Unlock while reading from files and memtables
{
  mutex_.Unlock();
  // First look in the memtable, then in the immutable memtable (if any).
  LookupKey lkey(key, snapshot);
  if (mem->Get(lkey, value, &s)) {
    // Done
  } else if (imm != nullptr && imm->Get(lkey, value, &s)) {
    // Done
  } else {
    s = current->Get(options, lkey, value, &stats);
```

```
      have_stat_update = true;
    }
    mutex_.Lock();
  }


  if (have_stat_update && current->UpdateStats(stats)) {
    MaybeScheduleCompaction();
  }
  mem->Unref();
  if (imm != nullptr) imm->Unref();
  current->Unref();
  return s;
}
```

DBImpl::Get 首先会尝试从内存数据中读取数据，如果找不到则会使用 Version::Get 读取，其过程参见上一篇博文。至此读取的流程也分析完毕。

总结

1. LevelDB 每次打开数据库时都会创建新的 MANIFEST 文件（默认情况下 Options::reuse_logs=false ）；
2. 存储在日志中的键值对，会在下一次打开数据库时转为 .ldb 文件。

Sign in to comment

Styling with Markdown is supported

Sign in with GitHub