LevelDB 源码分析「三、高性能写操作」

2019.08.05 SF-Zhou

本系列的前两篇介绍了 LevelDB 中使用的数据结构，并没有牵涉到 LevelDB 的核心实现。接下来的几篇将着重介绍 LevelDB 核心组件，包括日志、内存数据库、SortedTable、Compaction 和版本管理。本篇着重阐述高性能写操作的秘密：日志和内存数据库。

怎样最快地把键值对存起来？不考虑查找的速度的话，追加地写入文件是最快的，查找时反向查找。举个例子  ：

```
dict[1] = "LY"
dict[2] = "SF"
dict[3] = "MX"
del dict[1]
dict[2] = "ST"
```

上面代码中的 5 个操作，顺序地写入文件，每次添加一行，可以得到类似如下的记录：

```
Add 1: "LY"
Add 2: "SF"
Add 3: "MX"
Del 1
Add 2: "ST"
```

查找时反向查找，例如查找 key=2，返回最后一行最新的结果 "ST"；查找 key=1，返回倒数第二行的删除操作。LevelDB 中写操作使用了相似的技术，其写入分为两步：

1. 将数据追加到日志中；
2. 将数据插入内存数据库。

追加到日志一来保证了写入速度，二来保证了数据不会丢失，只要日志写入了磁盘，即使机器断电了，重启后也可以根据日志恢复出数据来；插入内存数据库同样维持着高性能，当内存数据库的小大到达一定规模时，会将当前的内存数据库持久化并建立新的内存数据库。

1. 批量写操作 WriteBatch

LevelDB 的键值对写入接口为 DB::Put(options, key, value)，删除某个键值对的接口为 DB::Delete(options, key)，其对应的实现为：

```
// source: db/db_impl.cc

Status DB::Put(const WriteOptions& opt, const Slice& key, const Slice& value) {
  WriteBatch batch;
  batch.Put(key, value);
  return Write(opt, &batch);
}

Status DB::Delete(const WriteOptions& opt, const Slice& key) {
```

```
  WriteBatch batch;
  batch.Delete(key);
  return Write(opt, &batch);
}
```

插入和删除操作首先被打包成一个 WriteBatch 。其定义于 include/leveldb/write_batch.h：

```
// WriteBatch holds a collection of updates to apply atomically to a DB.
//
// The updates are applied in the order in which they are added
// to the WriteBatch.  For example, the value of "key" will be "v3"
// after the following batch is written:
//
//    batch.Put("key", "v1");
//    batch.Delete("key");
//    batch.Put("key", "v2");
//    batch.Put("key", "v3");
//
// Multiple threads can invoke const methods on a WriteBatch without
// external synchronization, but if any of the threads may call a
// non-const method, all threads accessing the same WriteBatch must use
// external synchronization.

#include <string>

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

```
namespace leveldb {

class Slice;

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_
};

}  // namespace leveldb
```

WriteBatch 接口中除了提到的 Put 和 Delete，还提供了一个 Append 方法可以将其他 WriteBatch 合并过来。另外提供了一个 Iterate 迭代函数和对应的 Handler 类接口，后面会使用到。值得注意的还有 friend class WriteBatchInternal;，这种预先定义一个友元类、后期则可以在该友元类中直接访问私有变量和方法，适合一些不方便暴露出来的内部操

作。接看看 WriteBatchInternal 的定义 db/write_batch_internal.h：

```
#include "db/dbformat.h"
#include "leveldb/write_batch.h"


namespace leveldb {

class MemTable;

// 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);

};

}  // namespace leveldb
```

类中全部是静态函数，并且附带至少一个 WriteBatch* batch 参数。因为友元类的原因这些函数里均可以访问 WriteBatch 里唯一的私有成员 rep_ 。 WriteBatch 和 WriteBatchInternal 函数实现均位于 db/write_batch.cc ，为了方便阅读我会把内部的函数重新排序：

```
// 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);
}
```

```
size_t WriteBatch::ApproximateSize() const { return rep_.size(); }

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


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

SequenceNumber WriteBatchInternal::Sequence(const WriteBatch* b) {
  return SequenceNumber(DecodeFixed64(b->rep_.data()));
}

void WriteBatchInternal::SetSequence(WriteBatch* b, SequenceNumber seq) {
  EncodeFixed64(&b->rep_[0], seq);
}
```

WriteBatch::rep_ 的前 12 个字节定义为 Header，存储了 sequence number 和 count。EncodeFixed 和 DecodeFixed 系列函数实现了数值到字符串的编解码，有兴趣可以前往 util/coding.cc 查看实现，这里不详细介绍了。接下来看 Put 和 Delete 的实现：

```
void WriteBatch::Put(const Slice& key, const Slice& value) {
  WriteBatchInternal::SetCount(this, WriteBatchInternal::Count(this) + 1);
  rep_.push_back(static_cast<char>(kTypeValue));
  PutLengthPrefixedSlice(&rep_, key);
  PutLengthPrefixedSlice(&rep_, value);
}
```

```
void WriteBatch::Delete(const Slice& key) {
  WriteBatchInternal::SetCount(this, WriteBatchInternal::Count(this) + 1);
  rep_.push_back(static_cast<char>(kTypeDeletion));
  PutLengthPrefixedSlice(&rep_, key);
}

void WriteBatch::Append(const WriteBatch& source) {
  WriteBatchInternal::Append(this, &source);
}

void WriteBatchInternal::SetContents(WriteBatch* b, const Slice& contents) {
  assert(contents.size() >= kHeader);
  b->rep_.assign(contents.data(), contents.size());
}

void WriteBatchInternal::Append(WriteBatch* dst, const WriteBatch* src) {
  SetCount(dst, Count(dst) + Count(src));
  assert(src->rep_.size() >= kHeader);
  dst->rep_.append(src->rep_.data() + kHeader, src->rep_.size() - kHeader);
}
```

Put 和 Delete 首先将计数加一，在 rep_ 中写入操作类型，再写入键值对。PutLengthPrefixedSlice 函数会先写入字符串的长度，再写入字符串的内容。WriteBatchInternal 的赋值和追加均是对 rep_ 的进行操作。继续看迭代函数和 Handle 的部分：

```
Status WriteBatch::Iterate(Handler* handler) const {
  Slice input(rep_);
```

```
if (input.size() < kHeader) {
  return Status::Corruption("malformed WriteBatch (too small)");
}

input.remove_prefix(kHeader);

Slice key, value;
int found = 0;
while (!input.empty()) {
  found++;
  char tag = input[0];
  input.remove_prefix(1);
  switch (tag) {
    case kTypeValue:
      if (GetLengthPrefixedSlice(&input, &key) &&
          GetLengthPrefixedSlice(&input, &value)) {
        handler->Put(key, value);
      } else {
        return Status::Corruption("bad WriteBatch Put");
      }
      break;
    case kTypeDeletion:
      if (GetLengthPrefixedSlice(&input, &key)) {
        handler->Delete(key);
      } else {
        return Status::Corruption("bad WriteBatch Delete");
      }
      break;
    default:
      return Status::Corruption("unknown WriteBatch tag");
  }
```

```
  }
  if (found != WriteBatchInternal::Count(this)) {
    return Status::Corruption("WriteBatch has wrong count");
  } else {
    return Status::OK();

  }
}

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

Status WriteBatchInternal::InsertInto(const WriteBatch* b, MemTable* memtable) {
  MemTableInserter inserter;
  inserter.sequence_ = WriteBatchInternal::Sequence(b);
  inserter.mem_ = memtable;
  return b->Iterate(&inserter);
```

```
}
```

迭代函数 WriteBatch::Iterate 会按照顺序将 rep_ 中存储的键值对操作放到 hander 上执行。下面的匿名空间里定义了一个继承 Handler 的子类 MemTableInserter，将 Put 和 Delete 转到 MemTable 上执行。MemTable 即为内存数据库，本文稍后介绍。WriteBatchInternal::InsertInto 就直接根据 MemTable 构造 MemTableInserter。这样做的好处，可能就是 WriteBatch::Iterate 与 MemTable 解耦，Handler 可以自行替换。

综合来看，WriteBatch 将所有的修改和删除操作均存储到一个字符串中，并且提供了内存数据库的迭代接口。而单个字符串也可以非常方便地进行持久化，这一点也会在日志部分有所体现。

2. 日志 Log

在 DB::Write 函数中，核心的写入步骤代码如下：

```
// 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(updates));
  bool sync_error = false;
  if (status.ok() && options.sync) {
    status = logfile_->Sync();
    if (!status.ok()) {
```

```
      if (!status.ok()) {
        sync_error = true;
      }
    }
    if (status.ok()) {
      status = WriteBatchInternal::InsertInto(updates, 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);
    }
  }
```

先追加到日志，再写入内存数据库。这里的 log_ 为成员变量，类型为 log::Writer，其定义位于 db/log_writer.h：

```
#include <stdint.h>

#include "db/log_format.h"
#include "leveldb/slice.h"
#include "leveldb/status.h"

namespace leveldb {

class WritableFile;

namespace log {
```

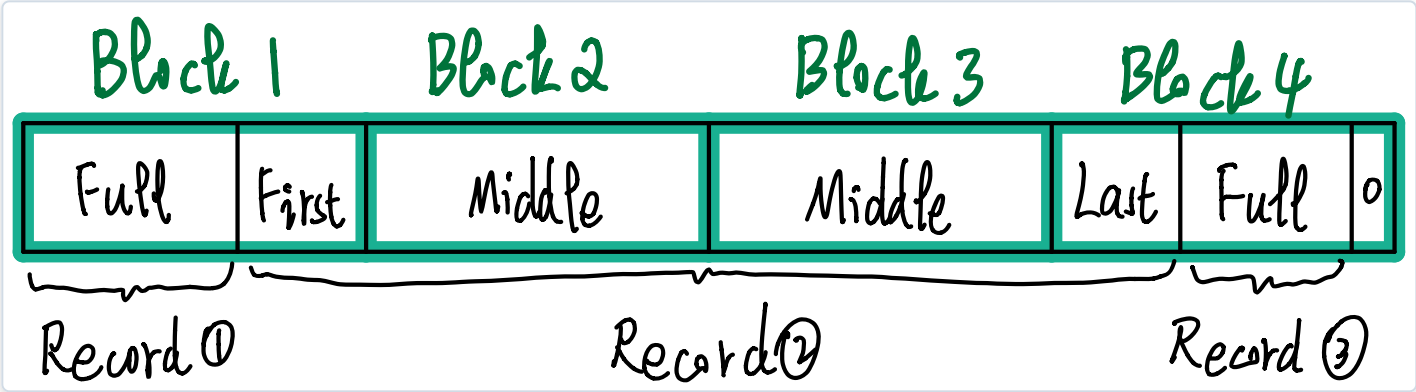
```
class Writer {
 public:
  // Create a writer that will append data to "*dest".
  // "*dest" must be initially empty.

  // "*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];
```

```
};

}  // namespace log
}  // namespace leveldb
```

公开的接口只有一个 Log::Writer::AddRecord，也就是 DB::Write 中调用的函数。另外类中还有一个私有数组 type_crc_，内部存储了预先计算的几种类型的 CRC32 校验值。常量 kMaxRecordType 定义于 db/log_format.h，该文件定义了日志格式相关的几个常量：

```
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;

// Header is checksum (4 bytes), length (2 bytes), type (1 byte).
static const int kHeaderSize = 4 + 2 + 1;
```

日志类中的 AddRecord 函数每次调用会增加一条记录 Record，同时我们需要保证以后可以按顺序读取出每一条 Record。为了提升日志的读取速度速度，LevelDB 引入了 Block 的概念。在读写文件时会按照一个一个 Block 来读写，默认的 Block 大小为 kBlockSize = 32KB 。而一条记录可能比 Block 还要长，所以还需要对过长的 Record 做合适的切分，切成片段 Fragment 后再放入 Block 中。Fragment 分为三种类型，分别是 kFirstType 、 kMiddleType 和 kLastType ，参看下图：



Log Format

继续看 db/log_writer.cc 的具体实现：

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

#include <stdint.h>

#include "leveldb/env.h"
#include "util/coding.h"
#include "util/crc32c.h"

namespace leveldb {
```

```
namespace log {

static void InitTypeCrc(uint32_t* type_crc) {
  for (int i = 0; i <= kMaxRecordType; i++) {
    char t = static_cast<char>(i);

    type_crc[i] = crc32c::Value(&t, 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::EmitPhysicalRecord(RecordType t, const char* ptr,
                                  size_t length) {
  assert(length <= 0xffff);  // Must fit in two bytes
  assert(block_offset_ + kHeaderSize + length <= kBlockSize);

  // 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);


  // 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;
}

}  // namespace log
}  // namespace leveldb
```

构造时 InitTypeCrc 函数初始化了 type_crc_ 数组，另外如果构造时有 dest_length 参数，则将 block_offset_ 设为 dest_length % kBlockSize 。至于函数 EmitPhysicalRecord ，从函数名来看其作用是触发物理记录。该函数先构造了一个 Record Header buf ，前 4 字节存储 CRC32 校验值，后面依次存储长度和 Record 类型。最终会将 Header、字节流写入文件并刷新，并且更新 block_offset_ 。最后来看下 Log::Writer::AddRecord 函数的实现：

```
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;
  do {
    const int leftover = kBlockSize - block_offset_;
    assert(leftover >= 0);
    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);

    const size_t avail = kBlockSize - block_offset_ - kHeaderSize;
    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;
    }

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

函数首先会计算当前 Block 剩余空间大小 leftover，如果连 Header 都没法写进去，就直接填充 0 进去，后期读取时会直接过滤掉。而后计算可用的空间大小 avail 和当前写入的长度 fragment_length 以及对应的记录类型 type，最后调用 EmitPhysicalRecord 刷入文件。这种方式可以保证写 Record 时按照 BlockSize 对齐。

综上来看，写入时首先会把 WriteBatch::rep_ 对齐地追加到日志中，写入时做了合适的切分，并且加入了 CRC 校验。有写肯定有读，当进行恢复操作时就会读取上述日志，Log::Reader 代码实现位于 db/log_reader.h 和 db/log_reader.cc，读取的过程即为写入的逆过程，有兴趣可以自行阅读。

5 comments – *powered by giscus*

Oldest | Newest

suntzu93 Apr 9, 2020 edited

Thank you for the great article. I'm adding trying to use encrypt/decrypt for leveldb and my idea is encrypt data before write to file in env_posix.cc , method WriteUnbuffered(const char *data, size_t size) . But I don't know if encrypt " char *data " has any effect to Record Header , CRC32... . I greatly appreciate your comments on this matter.

↑ 1 ☺

0 replies

SF-Zhou Apr 9, 2020 Owner

> **@suntzu93**
> Thank you for the great article. I'm adding trying to use encrypt/decrypt for leveldb and my idea is encrypt data before write to file in env_posix.cc , method WriteUnbuffered(const char *data, size_t size) . But I don't know if encrypt " char *data " has any effect to Record Header , CRC32... . I greatly appreciate your comments on this matter.

I think it's ok if you do symmetrical decryption after `PosixRandomAccessFile::Read` . The position of reading and writing may need to be mapped according to the encryption algorithm.

↑ 1 ☺

0 replies

 **suntzu93** Apr 9, 2020

> **@suntzu93**
> Thank you for the great article. I'm adding trying to use encrypt/decrypt for leveldb and my idea is encrypt data before write to file in env_posix.cc , method WriteUnbuffered(const char *data, size_t size) . But I don't know if encrypt " char *data " has any effect to Record Header , CRC32... . I greatly appreciate your comments on this matter.
>
> I think it's ok if you do symmetrical decryption after `PosixRandomAccessFile::Read` . The position of reading and writing may need to be mapped according to the encryption algorithm.

Thanks for your answer, I'm doing with AES. I created a dex array from *data but only the first 5 bytes can be encrypted, the key and value cannot be encrypted. I'm using this library https://github.com/kokke/tiny-AES-c and it's working perfect with normal string. Do you have any idea about the type of "char *data"?

↑ 1 ☺ 0 replies

 **SF-Zhou** Apr 9, 2020 Owner

> **@suntzu93**
>
> > **@suntzu93**
> > Thank you for the great article. I'm adding trying to use encrypt/decrypt for leveldb and my idea is encrypt data before write to file in env_posix.cc , method WriteUnbuffered(const char *data, size_t size) . But I don't know if encrypt " char *data " has any effect to Record Header , CRC32... . I greatly appreciate your

> "data" has any effect to Record Header , CRC32... . I greatly appreciate your comments on this matter.
>
> I think it's ok if you do symmetrical decryption after `PosixRandomAccessFile::Read` . The position of reading and writing may need to be mapped according to the encryption algorithm.
>
> Thanks for your answer, I'm doing with AES. I created a dex array from *data but only the first 5 bytes can be encrypted, the key and value cannot be encrypted. I'm using this library https://github.com/kokke/tiny-AES-c and it's working perfect with normal string. Do you have any idea about the type of "char *data"?

The `data` in `WriteUnbuffered(const char *data, size_t size)` is not a c string, but a byte stream. It can contain `\0` . For the AES algorithm, I have not used it before and may not be able to help.

↑ 1 ☺ 0 replies

suntzu93 Apr 9, 2020

Really thank you for your help, now I have a keywork to continue my work. Have a nice day bro.

↑ 1 ☺ 0 replies

Write Preview Aa