LevelDB 源码分析「十、其他细节」

2019.09.16 SF-Zhou

本篇为 LevelDB 源码分析的最后一篇博文，将会分析 LevelDB 中同步、原子量、单元测试和构建系统的一些细节。

1. 同步 Synchronization

LevelDB 中有大量并发访问的场景，也就需要同步的支持。LevelDB 使用的仍然是 C++ 标准库中的互斥量和条件变量，做了简单的封装 port/port_stdcxx.h：

```
#include <condition_variable>  // NOLINT
#include <mutex>  // NOLINT

// Thinly wraps std::mutex.
class LOCKABLE Mutex {
 public:
  Mutex() = default;
  ~Mutex() = default;

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

  void Lock() EXCLUSIVE_LOCK_FUNCTION() { mu_.lock(); }
  void Unlock() UNLOCK_FUNCTION() { mu_.unlock(); }
  void AssertHeld() ASSERT_EXCLUSIVE_LOCK() {}
```

```
  void AssertHeld() ASSERT_EXCLUSIVE_LOCK() {}

 private:
  friend class CondVar;
  std::mutex mu_;
};

// Thinly wraps std::condition_variable.
class CondVar {
 public:
  explicit CondVar(Mutex* mu) : mu_(mu) { assert(mu != nullptr); }
  ~CondVar() = default;

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

  void Wait() {
    std::unique_lock<std::mutex> lock(mu_->mu_, std::adopt_lock);
    cv_.wait(lock);
    lock.release();
  }
  void Signal() { cv_.notify_one(); }
  void SignalAll() { cv_.notify_all(); }

 private:
  std::condition_variable cv_;
  Mutex* const mu_;
};
```

其中类似 EXCLUSIVE_LOCK_FUNCTION 的宏定义于 port/thread_annotations.h ，

作为线程安全分析的标注，在 Clang 环境下设定 -Wthread-safety 继而在编译期完成线程安全检查，详细资料参见文献 1。

这里分析一下 DBImpl::Write 中控制同步的部分：

```
class DBImpl {
  // Queue of writers.
  std::deque<Writer*> writers_ GUARDED_BY(mutex_);
  WriteBatch* tmp_batch_ GUARDED_BY(mutex_);
  ...
}

// Information kept for every waiting writer
struct DBImpl::Writer {
  explicit Writer(port::Mutex* mu)
      : batch(nullptr), sync(false), done(false), cv(mu) {}

  Status status;
  WriteBatch* batch;
  bool sync;
  bool done;
  port::CondVar cv;
};

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* updates = BuildBatchGroup(&last_writer);
  WriteBatchInternal::SetSequence(updates, last_sequence + 1);
  last_sequence += WriteBatchInternal::Count(updates);

  // 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()) {
        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);
    }
  }
  if (updates == 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: Writer list must be non-empty
// REQUIRES: First writer must have a non-null batch
WriteBatch* DBImpl::BuildBatchGroup(Writer** last_writer) {
  mutex_.AssertHeld();
  assert(!writers_.empty());
  Writer* first = writers_.front();
  WriteBatch* result = first->batch;
  assert(result != nullptr);

  size_t size = WriteBatchInternal::ByteSize(first->batch);

  // Allow the group to grow up to a maximum size, but if the
  // original write is small, limit the growth so we do not slow
  // down the small write too much.
  size_t max_size = 1 << 20;
  if (size <= (128 << 10)) {
    max_size = size + (128 << 10);
  }

  *last_writer = first;
  std::deque<Writer*>::iterator iter = writers_.begin();
  ++iter;  // Advance past "first"
  for (; iter != writers_.end(); ++iter) {
```

```
    Writer* w = *iter;
    if (w->sync && !first->sync) {
      // Do not include a sync write into a batch handled by a non-sync write.
      break;
    }


    if (w->batch != nullptr) {
      size += WriteBatchInternal::ByteSize(w->batch);
      if (size > max_size) {
        // Do not make batch too big
        break;
      }

      // Append to *result
      if (result == first->batch) {
        // Switch to temporary batch instead of disturbing caller's batch
        result = tmp_batch_;
        assert(WriteBatchInternal::Count(result) == 0);
        WriteBatchInternal::Append(result, first->batch);
      }
      WriteBatchInternal::Append(result, w->batch);
    }
    *last_writer = w;
  }
  return result;
}
```

假设现在有编号为 [1, 2, 3, 4] 的四个线程基本同时调用写入操作，发生的事件如下：

1. 每个线程内各自构造 Writer；

1. 每个线程中各自构造 writer；
2. 1 号线程较快地构造了 MutexLock 拿到锁，[2, 3, 4] 则阻塞在此处；
3. 1 号线程将写入请求插入双向队列中，跳过循环，继续向下走；
4. 1 号线程执行 BuildBatchGroup，由于队列中只有自身一个请求，不会发生合并；

5. 1 号线程执行 mutex_.Unlock() 释放锁，随后执行写入操作，完成后执行 mutex_.Lock() 再次获得锁；
6. 1 号线程在循环中从双向队列里将写入请求弹出，最后通知队列顶的 2 号线程唤醒；
7. 1号线程析构局部变量、释放锁。

在第 5 步发生释放锁的同时：

1. 2 号线程获得锁，将写入请求插入双向队列中，由于请求不在队列顶端，进而进入循环、等待、释放锁；
2. 3 号线程获得锁，将写入请求插入双向队列中，由于请求不在队列顶端，进而进入循环、等待、释放锁；
3. 此时 1 号线程写入完成、执行 mutex_.Lock() 获得锁，4 号线程继续等待；
4. 1 号线程执行结束、释放锁，2 号线程唤醒获得锁，执行 BuildBatchGroup 将队列中的 3 号线程中的写入请求合并；
5. 2 号线程执行 mutex_.Unlock() 解锁，随后执行写入操作，完成后执行 mutex_.Lock() 再次获得锁。与此同时 4 号线程获得锁，将写入请求插入双向队列中，等待、释放锁；
6. 2 号线程在循环中从双向队列里将写入请求弹出，将 3 号线程的写入请求标记为完成，尝试唤醒 3 号线程；
7. 2 号线程析构局部变量、释放锁，3 号线程唤醒、获得锁、判断已完成、返回、释放

7. 2 号线程构造部变量、释放锁。3 号线程唤醒、获得锁，判断已完成，返回、释放锁；
8. 4 号线程唤醒、获得锁，正常执行。

上述合并操作依赖写入时的释放锁操作，这使得其他线程有机会加入队列、然后等待，在下一次获得锁时合并队列中的其他写入请求。

2. 原子量 Atomic

由于并发访问，LevelDB 中也大量使用了原子量，且使用了两种不同的内存模型。一种是 Relaxed Ordering，其只保证原子性、不保证并发时的执行顺序，一般在计数功能中使用，例如内存池中的内存使用量：

```
size_t MemoryUsage() const {
  return memory_usage_.load(std::memory_order_relaxed);
}

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;
}
```

另一种是 Release-Acquire Ordering，其不仅可以保证原子性，还可以保证一定程度的

执行顺序。以下摘录自参考文献 7：

> *If an atomic store in thread A is tagged memory_order_release and an atomic load in thread B from the same variable is tagged memory_order_acquire , all memory writes (non-atomic and relaxed atomic) that happened-before the atomic store from the point of view of thread A, become visible side-effects in thread B. That is, once the atomic load is completed, thread B is guaranteed to see everything thread A wrote to memory.*

LevelDB 中的使用举例：

```
template <typename Key, class Comparator>
class SkipList {
  ...
  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);
  }
}

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

```
// TODO(opt): We can use a barrier-free variant of FindGreaterOrEqual()
// here since Insert() is externally synchronized.
Node* prev[kMaxHeight];
Node* x = FindGreaterOrEqual(key, prev);


// Our data structure does not allow duplicate insertion
assert(x == nullptr || !Equal(key, x->key));

int height = RandomHeight();
if (height > GetMaxHeight()) {
  for (int i = GetMaxHeight(); i < height; i++) {
    prev[i] = head_;
  }
  // It is ok to mutate max_height_ without any synchronization
  // with concurrent readers.  A concurrent reader that observes
  // the new value of max_height_ will see either the old value of
  // new level pointers from head_ (nullptr), or a new value set in
  // the loop below.  In the former case the reader will
  // immediately drop to the next level since nullptr sorts after all
  // keys.  In the latter case the reader will use the new node.
  max_height_.store(height, std::memory_order_relaxed);
}

x = NewNode(key, height);
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));
  prev[i]->SetNext(i, x);
}
```

```
}
```

3. 单元测试

LevelDB 中的单元测试并没有使用自家的 Google Test，而是自己实现了一套简单的测试工具，位于 util/testharness.h：

```
// An instance of Tester is allocated to hold temporary state during
// the execution of an assertion.
class Tester {
 private:
  bool ok_;
  const char* fname_;
  int line_;
  std::stringstream ss_;

 public:
  Tester(const char* f, int l) : ok_(true), fname_(f), line_(l) {}

  ~Tester() {
    if (!ok_) {
      fprintf(stderr, "%s:%d:%s\n", fname_, line_, ss_.str().c_str());
      exit(1);
    }
  }

  Tester& Is(bool b, const char* msg) {
    if (!b) {
```

```
      ss_ << " Assertion failure " << msg;
      ok_ = false;
    }
    return *this;
  }


  Tester& IsOk(const Status& s) {
    if (!s.ok()) {
      ss_ << " " << s.ToString();
      ok_ = false;
    }
    return *this;
  }

#define BINARY_OP(name, op)                             \
  template <class X, class Y>                           \
  Tester& name(const X& x, const Y& y) {                \
    if (!(x op y)) {                                    \
      ss_ << " failed: " << x << (" " #op " ") << y;    \
      ok_ = false;                                      \
    }                                                   \
    return *this;                                       \
  }

  BINARY_OP(IsEq, ==)
  BINARY_OP(IsNe, !=)
  BINARY_OP(IsGe, >=)
  BINARY_OP(IsGt, >)
  BINARY_OP(IsLe, <=)
  BINARY_OP(IsLt, <)
```

```
#undef BINARY_OP

  // Attach the specified value to the error message if an error has occurred
  template <class V>
  Tester& operator<<(const V& value) {

    if (!ok_) {
      ss_ << " " << value;
    }
    return *this;
  }
};

#define ASSERT_TRUE(c) ::leveldb::test::Tester(__FILE__, __LINE__).Is((c), #c)
#define ASSERT_OK(s) ::leveldb::test::Tester(__FILE__, __LINE__).IsOk((s))
#define ASSERT_EQ(a, b) \
  ::leveldb::test::Tester(__FILE__, __LINE__).IsEq((a), (b))
#define ASSERT_NE(a, b) \
  ::leveldb::test::Tester(__FILE__, __LINE__).IsNe((a), (b))
#define ASSERT_GE(a, b) \
  ::leveldb::test::Tester(__FILE__, __LINE__).IsGe((a), (b))
#define ASSERT_GT(a, b) \
  ::leveldb::test::Tester(__FILE__, __LINE__).IsGt((a), (b))
#define ASSERT_LE(a, b) \
  ::leveldb::test::Tester(__FILE__, __LINE__).IsLe((a), (b))
#define ASSERT_LT(a, b) \
  ::leveldb::test::Tester(__FILE__, __LINE__).IsLt((a), (b))
```

BINARY_OP 宏简化了比较运算函数的定义，使用完后及时 undef 也避免了污染。继续：

```
#define TCONCAT(a, b) TCONCAT1(a, b)
#define TCONCAT1(a, b) a##b

#define TEST(base, name)                                                \
  class TCONCAT(_Test_, name) : public base {                           \
   public:                                                              \
    void _Run();                                                        \
    static void _RunIt() {                                              \
      TCONCAT(_Test_, name) t;                                          \
      t._Run();                                                         \
    }                                                                   \
  };                                                                    \
  bool TCONCAT(_Test_ignored_, name) = ::leveldb::test::RegisterTest( \
      #base, #name, &TCONCAT(_Test_, name)::_RunIt);                    \
  void TCONCAT(_Test_, name)::_Run()

// Register the specified test.  Typically not used directly, but
// invoked via the macro expansion of TEST.
bool RegisterTest(const char* base, const char* name, void (*func)());
```

定义 TCONCAT1 宏是为了让 TCONCAT 像函数一样支持嵌套调用。全局变量 TCONCAT(_Test_ignored_, name) 则可以实现在 main 函数前对测试类进行注册。注册函数和执行所有测试的实现位于 util/testharness.cc :

```
namespace {
struct Test {
  const char* base;
```

```
  const char* name;
  void (*func)();
};
std::vector<Test>* tests;
}  // namespace

bool RegisterTest(const char* base, const char* name, void (*func)()) {
  if (tests == nullptr) {
    tests = new std::vector<Test>;
  }
  Test t;
  t.base = base;
  t.name = name;
  t.func = func;
  tests->push_back(t);
  return true;
}

int RunAllTests() {
  const char* matcher = getenv("LEVELDB_TESTS");

  int num = 0;
  if (tests != nullptr) {
    for (size_t i = 0; i < tests->size(); i++) {
      const Test& t = (*tests)[i];
      if (matcher != nullptr) {
        std::string name = t.base;
        name.push_back('.');
        name.append(t.name);
        if (strstr(name.c_str(), matcher) == nullptr) {
          continue;
```

```
        }
      }
      fprintf(stderr, "==== Test %s.%s\n", t.base, t.name);
      (*t.func)();
      ++num;

    }
  }
  fprintf(stderr, "==== PASSED %d tests\n", num);
  return 0;
}
```

4. 构建系统

LevelDB 使用 CMake 作为其构建系统，搭建了跨平台、可配置的编译系统。例如使用 CMake 变量 WIN32 实现不同环境的编译：

```
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)

if (WIN32)
  target_sources(leveldb
    PRIVATE
      "${PROJECT_SOURCE_DIR}/util/env_windows.cc"
      "${PROJECT_SOURCE_DIR}/util/windows_logger.h"
```

```
    ${PROJECT_SOURCE_DIR}/util/windows_logger.h
  )
else (WIN32)
  target_sources(leveldb
    PRIVATE
      "${PROJECT_SOURCE_DIR}/util/env_posix.cc"
      "${PROJECT_SOURCE_DIR}/util/posix_logger.h"
  )
endif (WIN32)
```

使用 configure_file 和 cmakedefine01 将 CMake 中的变量转为代码中的宏定义：

```
check_library_exists(crc32c crc32c_value "" HAVE_CRC32C)

configure_file(
  "${PROJECT_SOURCE_DIR}/port/port_config.h.in"
  "${PROJECT_BINARY_DIR}/${LEVELDB_PORT_CONFIG_DIR}/port_config.h"
)
```

对应的 port/port_config.h.in：

```
// Define to 1 if you have Google CRC32C.
#if !defined(HAVE_CRC32C)
#cmakedefine01 HAVE_CRC32C
#endif  // !defined(HAVE_CRC32C)

/* after compile */
// Define to 1 if you have Google CRC32C.
#if !defined(HAVE_CRC32C)
```

```
#define HAVE_CRC32C 0
#endif  // !defined(HAVE_CRC32C
```

使用 check_cxx_source_compiles 检查编译器的特性：

```
# Test whether -Wthread-safety is available. See
# https://clang.llvm.org/docs/ThreadSafetyAnalysis.html
# -Werror is necessary because unknown attributes only generate warnings.
set(OLD_CMAKE_REQUIRED_FLAGS ${CMAKE_REQUIRED_FLAGS})
list(APPEND CMAKE_REQUIRED_FLAGS -Werror -Wthread-safety)
check_cxx_source_compiles("
struct __attribute__((lockable)) Lock {
  void Acquire() __attribute__((exclusive_lock_function()));
  void Release() __attribute__((unlock_function()));
};
struct ThreadSafeType {
  Lock lock_;
  int data_ __attribute__((guarded_by(lock_)));
};
int main() { return 0; }
"  HAVE_CLANG_THREAD_SAFETY)
set(CMAKE_REQUIRED_FLAGS ${OLD_CMAKE_REQUIRED_FLAGS})

if(HAVE_CLANG_THREAD_SAFETY)
  target_compile_options(leveldb
    PUBLIC
      -Werror -Wthread-safety)
endif(HAVE_CLANG_THREAD_SAFETY)
```

References

1. "Thread Safety Analysis", *Clang Documentation*
2. "std::mutex", *C++ Reference*

3. "std::unique_lock", *C++ Reference*
4. "std::condition_variable", *C++ Reference*
5. "std::condition_variable::notify_one", *C++ Reference*
6. "std::atomic", *C++ Reference*
7. "std::memory_order", *C++ Reference*