在 LevelDB 中实现 TTL 功能

陈予曈 朱陈媛

一、实验要求

- 在 LevelDB 中实现键值对的 TTL 功能，使得过期的数据在读取时自动失效，并在适当的时候被合并清理。
- 修改 LevelDB 的源码，实现对 TTL 的支持，包括数据的写入、读取和过期数据的清理。
- 编写测试用例，验证 TTL 功能的正确性和稳定性。

二、设计方案

1.数据编码方式修改

1.1 设计思路

levelDB 中，数据是以键值对的方式存储的。要实现键值对的 TTL 功能，就要把过期时间戳加入存储结构中。在本实验中，我们采用将过期时间戳与值一起存储的方式。

Key	Value	TTL 时间戳（后 19 位）

将过期时间戳与值一起存储的方式具有以下优点：

实现简单：无需为 TTL 功能设计额外的数据结构，仅需在值的末尾添加时间戳信息，编码方式简洁。

访问高效：每次访问数据时可以通过解析直接判断过期状态，不需要额外查询或计算，提升了数据读取的响应速度。

1.2 实现过程

首先，修改 DB::Put 方法，使得 ttl 作为参数传入。

db_impl.c：

```
Status DB::Put(const WriteOptions& opt, const Slice& key, const Slice& value, uint64_t ttl)
  WriteBatch batch;
  batch.Put(key, value, ttl);
  return Write(opt, &batch);
}
```

```
// Convenience methods
Status DBImpl::Put(const WriteOptions& o, const Slice& key, const Slice& val, uint64_t ttl)
  return DB::Put(o, key, val, ttl);
}
```

write_batch.h：

```
  // ----------------------------For TTL-------------------------------
  // 为当前key设置ttl，过期后自动失效
  virtual Status Put(const WriteOptions& options, const Slice& key,
                     const Slice& value, uint64_t ttl) = 0;
```

接着，修改 WriteBatch::Put 方法，通过获取当前时间并添加指定的 TTL（以秒为单位）来计算过期时间戳。然后将该时间戳转换为字符串格式，方便附加到数据值中。

我们使用 std::chrono::system_clock::now() 获取当前时间，并通过 std::chrono::seconds(ttl) 添加 TTL 值。

write_batch.cc：

```
// 添加ttl，新的put方法-橙
void WriteBatch::Put(const Slice& key, const Slice& value, uint64_t ttl) {
  WriteBatchInternal::SetCount(this, WriteBatchInternal::Count(this) + 1);
  rep_.push_back(static_cast<char>(kTypeValue));
  PutLengthPrefixedSlice(&rep_, key);
  // PutLengthPrefixedSlice(&rep_, value);
  // 获取当前时间，加上 TTL 得到过期时间戳
  // 这里要改成seconds-橙
  auto expiration_time = std::chrono::system_clock::now()+ std::chrono::seconds(ttl);
  // 将过期时间戳转换为字符串
  std::time_t expiration_time_t = std::chrono::system_clock::to_time_t(expiration_time);
  std::stringstream ss;
  ss << std::put_time(std::localtime(&expiration_time_t), "%Y-%m-%d %H:%M:%S");
  std::string expiration_time_str = ss.str();
  // 将过期时间戳添加到值中
  std::string value_with_ttl(value.data(), value.size());
  value_with_ttl.append(expiration_time_str);  // 拼接
  PutLengthPrefixedSlice(&rep_, Slice(value_with_ttl.data(), value_with_ttl.size()));
}
```

1.3 遇到的问题和解决方案

1.数据库内含有脏数据

在 Put 方法被修改前，如果跑了测试脚本，会导致 testdb 中存有不含过期时间戳的数据，影响之后测试的准确性。因此，每次跑测试前，最好删除 testdb。

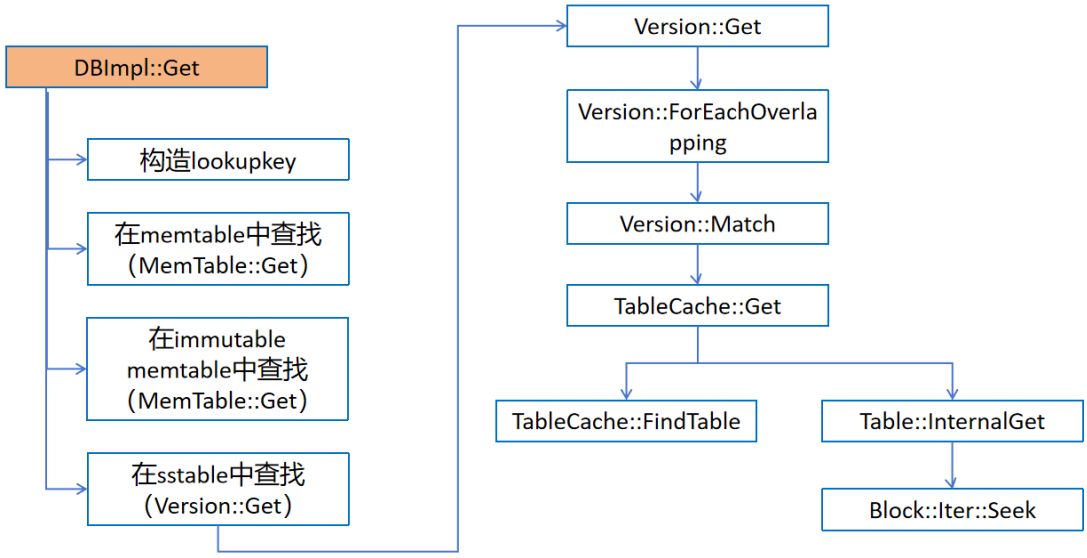
2. 前后计时方式不一致

当储存过期时间戳，与解析过期时间戳时，采用的获取当前系统时间的方法不一致时，会导致有些本应过期的数据没过期，造成系统的稳定性下降，因此，在多人开发时，需要前后统一计时方式。

2. 读取时判断是否过期

2.1 设计思路

levelDB 中读取键值对的流程如下图：



本次实验中主要涉及到的函数有 memtable::Get 和 block::Iter::Seek。

memtable::Get 函数的查询流程：

1） 创建迭代器，遍历 skiplist 查找目标 key（internalkey）

2） 如果找到，检验找到的 user key 是否和目标 user key 一致

3） 取 valuetype，如果 value 确实存在，将 value 返回

block::iter::seek 函数的查询流程：

1） 借助上一次查找的 key 缩减查找范围

2） 在 record 外部二分查找，确定 key 所在的 record

3） 对于 record 内部的 16 个 entry，直接线性查找

对于这两类读取操作，判断数据过期与否，都是在解析键值对编码结构的基础上进行。由于在第一步设计的新的键值对编码结构中，时间戳和 value 存储在

同一个字段中，因此要得到时间戳，需要先定位到 value（此处的 value 是包括时间戳的，即代码中的 value_with_ttl），更具体来说，就是定位到和 user_key 一致的 key、取出键值对中的 value 返回的时候。

对于 memtable 而言，memtable::Get 函数中的下面这段代码，就是定位到正确的 key，并且将其中的 value 解析出来并赋值给返回值 value 的代码。

```
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)) {       //取低8位，即valuetype
    case kTypeValue: {
      Slice v = GetLengthPrefixedSlice(key_ptr + key_length);
      value->assign(v.data(), v.size());
      return true;
    }
    case kTypeDeletion:   //value已经删除了
      *s = Status::NotFound(Slice());
      return true;
```

sstable 相对复杂，在查找时，先查找每一层，在每一层内部遍历（或是二分查找）；然后定位到一个 sstable 后，再对其中的 index block 进行二分查找，定位到一个 index block 索引到的 data block；在这个 data block 里，对重启点执行二分查找，定位到一个重启点后执行线性查找，最后定位到目标 record，取出其中的 value 并返回。

在以上步骤中，包含对 value 的操作的实际上就是最后一步，即在重启点中执行线性查找、并定位到最终的目标 record。因此，需要修改的目标代码段（在 Block::Seek 函数内部）：

```
// Linear search (within restart block) for first key >= target
while (true) {
  if (!ParseNextKey()) {     //一般情况下，parseNextKey函数会返回true，不会return
    return;
  }
  if (Compare(key_, target) >= 0) {   //当key_超过target，说明找到了
    return;
```

2.2 实现过程

1. memtable 的读取

原本的代码中，代码 Slice v = GetLengthPrefixedSlice(key_ptr + key_length) 已经将 value 的 slice 提取出来了，为了便于提取时间戳等操作，需要将 slice 转换成 string 类型，并且将其命名为 value_with_ttl。

首先，标准格式的时间"%Y-%m-%d %H:%M:%S"是 19 个字符，因此需要先判断字符串 value_with_ttl 的长度是否大于等于 19：

- 如果长度小于 19，那么说明时间戳信息不存在，此时就像原本的代码一样，直接将 value 赋值给返回值即可。
- 如果 value_with_ttl 的长度大于等于 19，那么尝试提取过期的时间戳：

先从字符串的末尾提取最后 19 个字符的子串，然后定义一个 std::tm 结构体用于存储解析后的时间信息，并使用 strptime 函数尝试将提取的字符串解析为 std::tm 结构体。对于解析结果：

- 如果 res=nullptr，说明解析失败，这种情况视为数据没有过期，将 value 字段的长度去掉后 19 位，赋值给返回值。
- 如果时间戳解析成功，那么先将存储时间戳的 std::tm 结构体转换为 time_t 类型，同样地，获取当前时间并且也转换成 time_t 类型。比较时间戳和当前时间：
  - 如果时间戳小于等于当前时间，说明数据已经过期，因此将 status 更改为 NotFound，返回的 value 长度为 0。
  - 如果数据没有过期，那么将 value 字段的长度去掉后 19 位，赋值给返回值。

```
case kTypeValue: {
  Slice v = GetLengthPrefixedSlice(key_ptr + key_length);
  // 数据过期则读取不到
  std::string value_with_ttl(v.data(), v.size());
  if (value_with_ttl.size() >= 19) {
    std::string expiration_time_str = value_with_ttl.substr(value_with_ttl.size() - 19); // 提取过期时间戳
    std::tm tm = {};
    char* res = strptime(expiration_time_str.c_str(), "%Y-%m-%d %H:%M:%S", &tm);
    if (res == nullptr) {  // 解析时间戳失败
      value->assign(v.data(), v.size()-19);
      return true;
    } else {
      std::time_t expiration_time = std::mktime(&tm);
      std::time_t current_time = std::chrono::system_clock::to_time_t(std::chrono::system_clock::now());
      if (expiration_time <= current_time) {   // 数据过期
        //std::cerr << "notfound_mem" << std::endl;
        *s = Status::NotFound(Slice());
        value->assign(v.data(), 0);
      }
      else  // 数据未过期
      {
        value->assign(v.data(), v.size()-19);
      }
      return true;
    }
  } else {   // 时间戳信息不存在
    value->assign(v.data(), v.size());
    return true;
  }
}
```

2. sstable 的读取

Block::Seek 函数通过循环调用 ParseNextKey 函数实现重启点内部的线性查找，每次将指针 p 的位置移动到下一个 entry 的开头，对于每一个 entry（即 record），获取当前 entry 中的 key 和 value。

```
key_.resize(shared);            //更改key_为当前entry的shared+non_shared部分
key_.append(p, non_shared);
value_ = Slice(p + non_shared, value_length);   //得到value内容
```

Block::Seek 函数比较得到的 key 和 user_key，如果相等则返回（实际上代码中是大于等于，因为 key 有序排列、查找按照 key 从小到大的顺序，相当于如果 key 超过了 user_key，说明找到了）。

```
// Linear search (within restart block) for first key >= target
while (true) {
  if (!ParseNextKey()) {     //一般情况下, parseNextKey函数会返回true, 不会return
    return;
  }
  if (Compare(key_, target) >= 0) {   //当key_超过target, 说明找到了
    return;
```

现在，我们需要在返回之前对得到的 value 进行检验（检查其中的时间戳）：

主要的代码逻辑和 memtable 的检验相同，主要的不同在于，memtable::Get 函数的 value 和 status 是直接作为返回值返回到上一层函数的，而 Block::Seek 函数没有返回值，value 和 status 是依靠迭代器传递回上一层函数的。

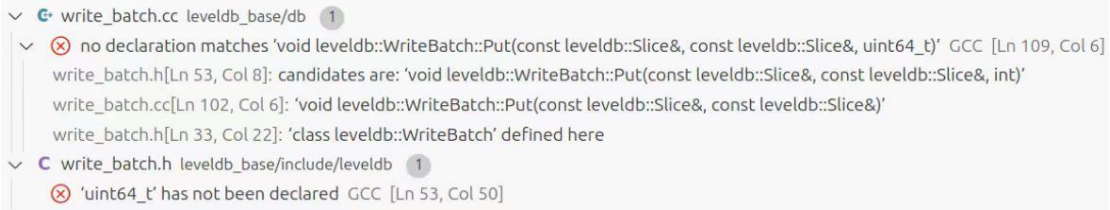
```
if (Compare(key_, target) >= 0) {
  // 解析value
  std::string value_with_ttl(value_.data(), value_.size());
  if (value_with_ttl.size() >= 19) {
    std::string expiration_time_str = value_with_ttl.substr(value_with_ttl.size() - 19); // 提取过期时间戳
    std::tm tm = {};
    char* res = strptime(expiration_time_str.c_str(), "%Y-%m-%d %H:%M:%S", &tm);
    if (res == nullptr) {   // 解析时间戳失败
      value_ = Slice(value_.data(), value_.size()-19);
    } else {
      std::time_t expiration_time = std::mktime(&tm);
      std::time_t current_time = std::chrono::system_clock::to_time_t(std::chrono::system_clock::now());
      if (expiration_time <= current_time) {   // 数据过期
        //std::cerr << "notfound_sst" << std::endl;
        status_ = Status::NotFound(Slice());
        value_ = Slice(value_.data(), 0);
      }
      else   // 数据未过期
      {
        value_ = Slice(value_.data(), value_.size()-19);
      }
    }
  } else {   // 时间戳信息不存在
    value_ = Slice(value_.data(), value_.size());
  }
  return;
}
```

2.3 遇到的问题和解决方案

1. 编译时报错：“uint64_t”has not been declared

解决：查找资料发现，uint64_t 类型是 C99 标准中定义的固定宽度整数类型，GCC 支持 C99 标准，但需要手动启用；另一种方案就是直接添加头文件 stdint.h。

添加头文件后，编译不再报错。



2. 随机种子的问题

启动 ttl_test 后，ReadTTL 用例显示 failed，经过检查，发现可能是随机种子的问题，因为 srand 函数的随机种子是 static_cast<unsigned int>(time(0))，也就是根据当前时间的时间戳生成随机数，这样 InsertData 时插入数据的 key 和查找时所使用的 key 不一致，因此可能会导致读取失败的问题。

解决：将随机种子改成 0，即 srand(0)，固定每次的种子，这样插入和读取时生成的 key 就是同一批了。

3. Open DB failed 错误

ReadTTL 用例通过之后，CompactionTTL 还未开始，直接显示 open DB failed，检查后发现可能是 ReadTTL 测试中打开的 DB 没有清除，导致第二个测试无法打开。

```
[       OK ] TestTTL.ReadTTL (30037 ms)
[ RUN      ] TestTTL.CompactionTTL
open db failed
已放弃 (核心已转储)
```

解决：在 ReadTTL 用例结束处加上“delete db”，再次启动测试，测试通过。

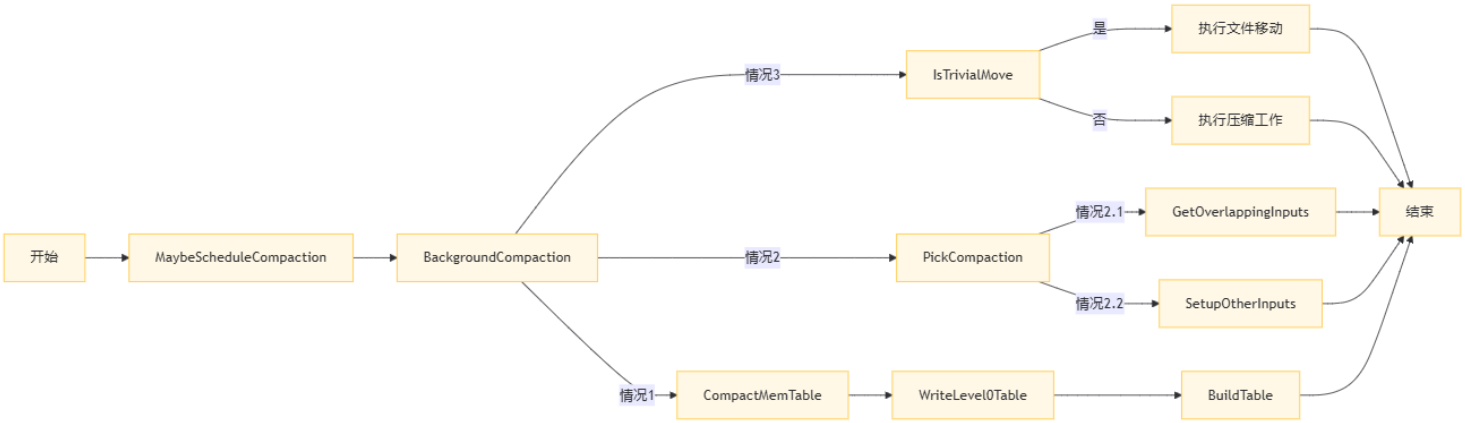
```
    for (int i = 0; i < 100; i++) {
        int key_ = rand() % key_num+1;
        std::string key = std::to_string(key_);
        std::string value;
        status = db->Get(readOptions, key, &value);
        ASSERT_FALSE(status.ok());
    }
    delete db;
}

TEST(TestTTL, CompactionTTL) {
```

3. Compaction 清理

3.1 设计思路

手动触发合并，失效数据会被合并丢弃。在数据合并过程中，删除过期的数据。

LevelDB 中，合并流程如下：

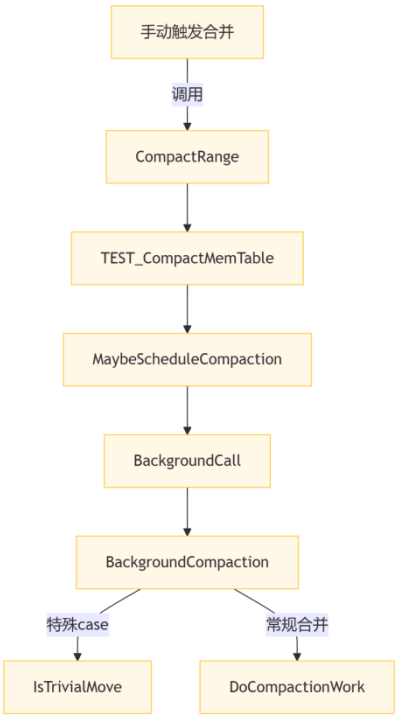


根据测试脚本，手动触发合并的函数是 CompactRange。手动合并可以指定合并范围，脚本里是 nullptr，意味着要合并所有数据。这个函数会先调用 TEST_CompactMemTable，优先进行小合并。然后再遍历需要合并的 level，并逐层调用 TEST_CompactRange，进行合并。

TEST_CompactRange 确定了合并的 begin 和 end，并调用 MaybeScheduleCompaction，也就是 levelDB 中所有合并操作的唯一入口。这个函数对合并前提进行了检查，并在需要合并时通过 Schedule 接口，发起 BackgroundCall，并执行真正的合并过程：BackgroundCompaction。

在 BackgroundCompaction 中，除了优先进行小合并，还有 IsTrivialMove 的特殊 case，其余大合并操作都会调用 DoCompactionWork 函数进行合并操作。通过阅读注释可知，这个函数本身就具有检查数据是否有效的功能。我们可以把检查数据是否过期的逻辑添加进去。因此，我们明确了需要修改的目标函数——DoCompactionWork。

在本次实验中，我们需要涉及到的合并流程如下：



3.2 实现过程

我们利用 DoCompactionWork 本身提供的 drop 变量来标记数据是否过期。首先从输入数据的 value 字段中提取带有 ttl 的完整字符串，将其转换为可处理的 std::string 类型。在处理该字符串时，提取其中的过期时间戳部分（后 19 位），并将其转换为时间格式。

如果当前时间超出了过期时间，系统则将 drop 标记为 true，以指示丢弃该数据；若当前时间尚未超出过期时间或时间戳解析失败，系统则将数据视为有效。实验中还考虑了长度校验，通过判断 value_with_ttl 是否满足最小长度要求，避免了时间戳提取过程中产生的错误。

```
Status DBImpl::DoCompactionWork(CompactionState* compact) {
  while (input->Valid() && !shutting_down_.load(std::memory_order_acquire)) {
    bool drop = false;
    // 检查数据是否过期-橙
    Slice value = input->value();
    std::string value_with_ttl(value.data(), value.size());
    if (value_with_ttl.size() >= 19) {
      std::string expiration_time_str = value_with_ttl.substr(value_with_ttl.size() - 19); //
      std::tm tm = {};
      char* res = strptime(expiration_time_str.c_str(), "%Y-%m-%d %H:%M:%S", &tm);
      if (res == nullptr) {
        std::cerr << "Failed to parse expiration time: " << expiration_time_str << std::endl;
        drop = false; // 解析失败则视为有效
      } else {
        std::time_t expiration_time = std::mktime(&tm);
        std::time_t current_time = std::chrono::system_clock::to_time_t(std::chrono::system_c
        if (expiration_time <= current_time) {
          drop = true; // 数据过期，标记为丢弃
        }
        else
        {
          // 这里注释掉-橙2        You, 前天 • Uncommitted changes
          //std::cerr << "NO TTL." << std::endl;
        }
      }
    } else {
      std::cerr << "Invalid value length for expiration time extraction." << std::endl;
      drop = false; // 无法提取到过期时间，则视为有效
    }
```

3.3 遇到的问题与解决方案

1.手动合并可能无法保证合并所有数据，导致无法完全丢弃过期数据。

解决：修改 CompactRange 的逻辑，确保合并所有数据。

通过 gdb 调试，我们发现如果这里是 level < max_level_with_files，到最后总会有第二层的一个文件不会被合并。有因为合并操作是将 leveli 层的数据合并到 leveli+1 层，所以这里有可能是 level2 的数据没有被合并到。因此，改为小于等于后，level2 层的数据也能被合并了。

```cpp
void DBImpl::CompactRange(const Slice* begin, const Slice* end) {
  int max_level_with_files = 1;
  {
    MutexLock l(&mutex_);
    Version* base = versions_->current();
    for (int level = 1; level < config::kNumLevels; level++) {
      if (base->OverlapInLevel(level, begin, end)) {
        max_level_with_files = level;
      }
    }
  }
  TEST_CompactMemTable();  // TODO(sanjay): Skip if memtable does not overlap
  // 这个改成小于等于，test能过，但这不是改变原本的手动合并逻辑了吗？-橙
  for (int level = 0; level <= max_level_with_files; level++) {
    TEST_CompactRange(level, begin, end);
  }
}      Gabor Cselle, 13年前 • A number of bugfixes: …
```

三、测试用例和测试结果

3.1 测试用例（ttl_test.cc）

```cpp
#include "gtest/gtest.h"

#include "leveldb/env.h"
#include "leveldb/db.h"


using namespace leveldb;

constexpr int value_size = 2048;
constexpr int data_size = 128 << 20;

Status OpenDB(std::string dbName, DB **db) {
  Options options;
  options.create_if_missing = true;
  return DB::Open(options, dbName, db);
}

void InsertData(DB *db, uint64_t ttl/* second */) {
  WriteOptions writeOptions;
  int key_num = data_size / value_size;
  //srand(static_cast<unsigned int>(time(0)));
  srand(0);
  for (int i = 0; i < key_num; i++) {
    int key_ = rand() % key_num+1;
    //int key_ = i + 1;
    std::string key = std::to_string(key_);
    std::string value(value_size, 'a');
    db->Put(writeOptions, key, value, ttl);
  }
}

void GetData(DB *db, int size = (1 << 30)) {
  ReadOptions readOptions;
  int key_num = data_size / value_size;

  // 点查
  //srand(static_cast<unsigned int>(time(0)));
  srand(0);
  for (int i = 0; i < 100; i++) {
    int key_ = rand() % key_num+1;
    std::string key = std::to_string(key_);
    std::string value;
    db->Get(readOptions, key, &value);
  }
}
```

```cpp
TEST(TestTTL, ReadTTL) {
    DB *db;
    if(OpenDB("testdb", &db).ok() == false) {
        std::cerr << "open db failed" << std::endl;
        abort();
    }

    uint64_t ttl = 20;

    InsertData(db, ttl);

    ReadOptions readOptions;
    Status status;
    int key_num = data_size / value_size;
    //srand(static_cast<unsigned int>(time(0)));
    srand(0);
    for (int i = 0; i < 100; i++) {
        int key_ = rand() % key_num+1;
        std::string key = std::to_string(key_);
        std::string value;
        status = db->Get(readOptions, key, &value);
        ASSERT_TRUE(status.ok());
    }

    Env::Default()->SleepForMicroseconds(ttl * 1000000);

    for (int i = 0; i < 100; i++) {
        int key_ = rand() % key_num+1;
        std::string key = std::to_string(key_);
        std::string value;
        status = db->Get(readOptions, key, &value);
        ASSERT_FALSE(status.ok());
    }
  delete db;
}
```

```cpp
TEST(TestTTL, CompactionTTL) {
    DB *db;

    if(OpenDB("testdb", &db).ok() == false) {
        std::cerr << "open db failed" << std::endl;
        abort();
    }

    uint64_t ttl = 20;
    InsertData(db, ttl);
    //这里为什么要定义两个ranges1? -朱陈媛
    leveldb::Range ranges[1];
    ranges[0] = leveldb::Range("-", "A");
    uint64_t sizes[1];
    db->GetApproximateSizes(ranges, 1, sizes);
    // printf("part1\n");
    ASSERT_GT(sizes[0], 0);

    Env::Default()->SleepForMicroseconds(ttl * 1000000);

    db->CompactRange(nullptr, nullptr);
    // 先注释掉重复定义的-朱陈媛
    // leveldb::Range ranges[1];
    ranges[0] = leveldb::Range("-", "A");
    // uint64_t sizes[1];
    db->GetApproximateSizes(ranges, 1, sizes);
    ASSERT_EQ(sizes[0], 0);
}


int main(int argc, char** argv) {
  // All tests currently run with the same read-only file limits.
  testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

3.2 测试结果

```
chen@chen-None:~/leveldb_base-通过read_ttl/build$ "/home/chen/leveldb_base-通过read_ttl/build/ttl_test"
[==========] Running 2 tests from 1 test suite.
[----------] Global test environment set-up.
[----------] 2 tests from TestTTL
[ RUN      ] TestTTL.ReadTTL
[       OK ] TestTTL.ReadTTL (30900 ms)
[ RUN      ] TestTTL.CompactionTTL
[       OK ] TestTTL.CompactionTTL (31922 ms)
[----------] 2 tests from TestTTL (62824 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 1 test suite ran. (62828 ms total)
[  PASSED  ] 2 tests.
```

四、总结

通过本次实验，我们对 leveldb 的代码实现有了更深入的理解、更加关注到代码中的细节部分是如何实现的；并且对于调试方法更加熟悉；同时，在协作编程的过程中，对代码不同模块之间的关系（例如写入和读取过程也会涉及到合并操作）也产生了一些思考，从而学着从整体性的角度理解代码、分析代码的实现逻辑。