LevelDB 源码分析「一、基本数据结构」

2019.07.26 SF-Zhou

断断续续大半年，LevelDB 的源代码快看完了。期间经常会发出由衷的感叹：Google 的代码写得真好。为了督促自己尽快看完，同时也为了真正地从 LevelDB 源码里汲取养分，所以开出一个新系列「LevelDB 源码分析」，希望能整理输出一些干货。作为系列的第一篇，本文会介绍 LevelDB 中的基本数据结构，包括 Slice 、 Hash 、 LRUCache 。

1. 字符串封装 Slice

Slice 定义于 include/leveldb/slice.h ，源码不过百行：

```
#include <assert.h>
#include <stddef.h>
#include <string.h>

#include <string>

#include "leveldb/export.h"

namespace leveldb {

class LEVELDB_EXPORT Slice {
 public:
  // Create an empty slice.
```

```
Slice() : data_(""), size_(0) {}

// Create a slice that refers to d[0,n-1].
Slice(const char* d, size_t n) : data_(d), size_(n) {}

// Create a slice that refers to the contents of "s"
Slice(const std::string& s) : data_(s.data()), size_(s.size()) {}

// Create a slice that refers to s[0,strlen(s)-1]
Slice(const char* s) : data_(s), size_(strlen(s)) {}

// Intentionally copyable.
Slice(const Slice&) = default;
Slice& operator=(const Slice&) = default;

// Return a pointer to the beginning of the referenced data
const char* data() const { return data_; }

// Return the length (in bytes) of the referenced data
size_t size() const { return size_; }

// Return true iff the length of the referenced data is zero
bool empty() const { return size_ == 0; }

// Return the ith byte in the referenced data.
// REQUIRES: n < size()
char operator[](size_t n) const {
  assert(n < size());
  return data_[n];
}
```

```
  // Change this slice to refer to an empty array
  void clear() {
    data_ = "";
    size_ = 0;
  }


  // Drop the first "n" bytes from this slice.
  void remove_prefix(size_t n) {
    assert(n <= size());
    data_ += n;
    size_ -= n;
  }

  // Return a string that contains the copy of the referenced data.
  std::string ToString() const { return std::string(data_, size_); }

  // Three-way comparison.  Returns value:
  //   <  0 iff "*this" <  "b",
  //   == 0 iff "*this" == "b",
  //   >  0 iff "*this" >  "b"
  int compare(const Slice& b) const;

  // Return true iff "x" is a prefix of "*this"
  bool starts_with(const Slice& x) const {
    return ((size_ >= x.size_) && (memcmp(data_, x.data_, x.size_) == 0));
  }

 private:
  const char* data_;
  size_t size_;
```

```
};

inline bool operator==(const Slice& x, const Slice& y) {
  return ((x.size() == y.size()) &&
          (memcmp(x.data(), y.data(), x.size()) == 0));

}

inline bool operator!=(const Slice& x, const Slice& y) { return !(x == y); }

inline int Slice::compare(const Slice& b) const {
  const size_t min_len = (size_ < b.size_) ? size_ : b.size_;
  int r = memcmp(data_, b.data_, min_len);
  if (r == 0) {
    if (size_ < b.size_)
      r = -1;
    else if (size_ > b.size_)
      r = +1;
  }
  return r;
}

}  // namespace leveldb
```

没有外部依赖，代码也非常清晰易懂。整个定义可以分为四个部分：构造、获取、修改和比较。

构造：默认构造为空字符串，字符串构造提供了带长度和不带长度，并且支持默认的复制构造函数和赋值操作符（毕竟只有 data_ 和 size_ 两个属性）。

获取：获取 Slice 的基本信息，支持导出为 std::string 。

修改：支持 clear 操作字符串清空，也支持 remove_prefix 将指定长度的前缀去除。注意 data_ 的类型为 const char * ，对应的字符串内容是不可修改的， Slice 只能修改字符串的起始位置。

比较： Slice::compare 实现了非常严谨的字符串比较，返回 0/1/-1。注意为了提高性能， operator== 并没有直接调用 Slice::compare 。

值得注意的是， Slice 本身并没有任何内存管理，仅仅是 C 风格字符串及其长度的封装。

2. 哈希函数 Hash

哈希函数定义于 util/hash.[h/cc] ，源代码如下：

```
#include "util/hash.h"

#include <string.h>

#include "util/coding.h"

// The FALLTHROUGH_INTENDED macro can be used to annotate implicit fall-through
// between switch labels. The real definition should be provided externally.
// This one is a fallback version for unsupported compilers.
#ifndef FALLTHROUGH_INTENDED
#define FALLTHROUGH_INTENDED \
```

```
  do {                        \
  } while (0)
#endif

namespace leveldb {


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);

  // Pick up four bytes at a time
  while (data + 4 <= limit) {
    uint32_t w = DecodeFixed32(data);
    data += 4;
    h += w;
    h *= m;
    h ^= (h >> 16);
  }

  // Pick up remaining bytes
  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;
}

}  // namespace leveldb
```

每次按照四字节长度读取字节流中的数据 w，并使用普通的哈希函数计算哈希值。计算过程中使用 uint32_t 的自然溢出特性。四字节读取则为了加速，最终可能剩下 3/2/1 个多余的字节，使用 switch 语句补充计算，以实现最好的性能。

这里 FALLTHROUGH_INTENDED 宏并无实际作用，仅仅作为一种“我确定我这里想跳过”的标志。 do {} while(0) 对代码无影响，这种写法也会出现在一些多行的宏定义里（见链接）。

LevelDB 中哈希表和布隆过滤器会使用到该哈希函数。

3. 缓存 LRUCache

LevelDB 中使用的是 Least Recently Used Cache，即最近最少使用缓存。缓存接口定义于 include/leveldb/cache.h，去除掉注释后接口如下（其实建议自己看看源代码的注释）：

```
#include <stdint.h>

#include "leveldb/export.h"
#include "leveldb/slice.h"

namespace leveldb {

class LEVELDB_EXPORT Cache;

LEVELDB_EXPORT Cache* NewLRUCache(size_t capacity);

class LEVELDB_EXPORT Cache {
 public:
  Cache() = default;
  Cache(const Cache&) = delete;
  Cache& operator=(const Cache&) = delete;
  virtual ~Cache();

  struct Handle {};
  virtual Handle* Insert(const Slice& key, void* value, size_t charge,
                         void (*deleter)(const Slice& key, void* value)) = 0;
  virtual Handle* Lookup(const Slice& key) = 0;
  virtual void Release(Handle* handle) = 0;
  virtual void* Value(Handle* handle) = 0;
  virtual void Erase(const Slice& key) = 0;
  virtual uint64_t NewId() = 0;
  virtual void Prune() {}
  virtual size_t TotalCharge() const = 0;

 private:
```

```
  void LRU_Remove(Handle* e);
  void LRU_Append(Handle* e);
  void Unref(Handle* e);

  struct Rep;

  Rep* rep_;
};
```

接口仅依赖 Slice，接口也很容易看懂。 Cache 中定义的 Handle 仅作为指针类型使用，实际上使用 void * 也并无区别， Handle 增加语意而已。而 Rep *rep_ 则是经典的 pImpl 范式，但 cache.cc 中并没有使用到该机制，注释掉这两行不影响编译，所以留到后续文章中再介绍吧。

NewLRUCache 作为工厂函数，可以生产一个 LRUCache，其定义于 util/cache.cc：

```
class ShardedLRUCache : public Cache {
  ...
}

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

LRUCahce 的实现依靠双向环形链表和哈希表。其中双向环形链表维护 Recently 属性，哈希表维护 Used 属性。双向环形链表和哈希表的节点信息都存储于 LRUHandle 结构中：

```
struct LRUHandle {
  void* value;
  void (*deleter)(const Slice&, void* value);
  LRUHandle* next_hash;
  LRUHandle* next;

  LRUHandle* prev;
  size_t charge;  // TODO(opt): Only allow uint32_t?
  size_t key_length;
  bool in_cache;     // Whether entry is in the cache.
  uint32_t refs;     // References, including cache reference, if present.
  uint32_t hash;     // Hash of key(); used for fast sharding and comparisons
  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);
  }
};
```

依次解释每一项属性：

1. value 为缓存存储的数据，类型无关；
2. deleter 为键值对的析构函数指针；
3. next_hash 为开放式哈希表中同一个桶下存储链表时使用的指针；
4. next 和 prev 自然是双向环形链接的前后指针；

5. charge 为当前节点的缓存费用，比如一个字符串的费用可能就是它的长度；
6. key_length 为 key 的长度；
7. in_cache 为节点是否在缓存里的标志；
8. refs 为引用计数，当计数为 0 时则可以用 deleter 清理掉；
9. hash 为 key 的哈希值；
10. key_data 为变长的 key 数据，最小长度为 1， malloc 时动态指定长度。

接着看哈希表的实现：

```
class HandleTable {
 public:
  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_;
 uint32_t elems_;
 LRUHandle** list_;

 // 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;
  }

  void Resize() {

    uint32_t new_length = 4;
    while (new_length < elems_) {
      new_length *= 2;
    }
    LRUHandle** new_list = new LRUHandle*[new_length];
    memset(new_list, 0, sizeof(new_list[0]) * new_length);
    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;
  }
};
```

一个标准的开放式哈希实现。属性中 length_ 存储桶的数量， elems_ 存储哈希表中节点数， list_ 则为桶数组。每一个桶里存储 hash 值相同的一系列节点，这些节点构成一个链表，通过 next_hash 属性连接。

FindPointer 函数返回一个二级指针。无论是 list_[i] 还是 entry->next_hash ，均为 LRUHandle * ，那么一个节点总会有一个正确的 LRUHandle * 变量指向它，该函数就返回这个变量的指针。说起来有点绕，仔细看懂就好。

看懂后，理解 Insert 和 Remove 都不难。 Resize 则根据存储的节点数，对哈希表进行缩放。如果不缩放，这样的结构会退化到链表的复杂度。使用 2 的幂可以规避掉哈希值的模除，同样可以加速。 Resize 时会遍历每一个节点，将其从原位置取出，重新计算哈希值放到新位置，每次会加到桶中链表的头部。 Resize 过程中链表需要拒绝其他请求。

最后看 LRUCache 的实现就很简单了:

```
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_);
};


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_;
}

LRUCache::~LRUCache() {
  assert(in_use_.next == &in_use_);  // Error if caller has an unreleased handle
  for (LRUHandle* e = lru_.next; e != &lru_;) {
    LRUHandle* next = e->next;
    assert(e->in_cache);
    e->in_cache = false;
    assert(e->refs == 1);  // Invariant of lru_ list.
    Unref(e);
    e = next;
  }
}

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++;
}

void LRUCache::Unref(LRUHandle* e) {
  assert(e->refs > 0);

  e->refs--;
  if (e->refs == 0) {  // Deallocate.
    assert(!e->in_cache);
    (*e->deleter)(e->key(), e->value);
    free(e);
  } else if (e->in_cache && e->refs == 1) {
    // No longer in use; move to lru_ list.
    LRU_Remove(e);
    LRU_Append(&lru_, e);
  }
}

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;
}
```

```
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);
}

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

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_);

  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.
  memcpy(e->key_data, key.data(), key.size());
```

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

    usage_ += charge;
    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;
  }
  while (usage_ > capacity_ && lru_.next != &lru_) {
    LRUHandle* old = lru_.next;
    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);
}

// 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;
}


void LRUCache::Erase(const Slice& key, uint32_t hash) {
  MutexLock l(&mutex_);
  FinishErase(table_.Remove(key, hash));
}

void LRUCache::Prune() {
  MutexLock l(&mutex_);
  while (lru_.next != &lru_) {
    LRUHandle* e = lru_.next;
    assert(e->refs == 1);
    bool erased = FinishErase(table_.Remove(e->key(), e->hash));
    if (!erased) {  // to avoid unused variable when compiled NDEBUG
      assert(erased);
    }
  }
}
```

LRUCache 中存储了两条链表， lru_ 和 in_use_，分别记录普通节点和外部正在使用中的节点。外部正在使用中的节点是不可删除的，将二者区分开也方便做对应的清理。 Ref 和 Unref 分别增删引用计数，并完成节点在 lru_ 和 in_use_ 的交换，以及计数为 0 时做最后的删除。

双向链表，会将最新使用的节点放到链表的末端。这样在容量超标时，删除链表头部的、长时间未用的节点即可。该逻辑实现于 Insert 函数的结尾。

LRUCache 中在添删查操作中均使用互斥锁完成额外同步。 LevelDB 中的锁将在后续文章中详细介绍。

最后看分片 ShardedLRUCache 的实现：

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

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:
  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;

  }
};
```

LevelDB 默认将 LRUCache 分为 2^4 块，取哈希值的高 4 位作为分片的位置（取低 4 位的话分片基本就白做了）。分片可以提高查询和插入的速度，减少锁的压力，是提高缓存性能的常用方法。

总结

本文简单介绍了 LevelDB 中的 Slice、 Hash 和 LRUCache 的实现。慢慢会觉得代码中的每个细节都是有意义的，不可忽略。 LevelDB 源代码不超过 3 万行，非常推荐学习 C++ 的同学阅读。建议可以先读 util 部分，这里是通用的数据结构，没有太多依赖。

下一篇会继续介绍 LevelDB 中的其他数据结构，包括布隆过滤器、内存池和跳表。