LevelDB 源码分析「四、高性能写操作续」

2019.08.10 SF-Zhou

本系列的上一篇介绍了 LevelDB 写操作中构造的 WriteBatch，以及写操作的第一步：追加日志。本篇将继续介绍写操作的第二步：插入内存数据库。

3. 内存数据库 MemTable

MemTable 即为在内存中建立的 KV 数据库，基于跳表，定义于 db/memtable.h：

```
#include <string>

#include "db/dbformat.h"
#include "db/skiplist.h"
#include "leveldb/db.h"
#include "util/arena.h"

namespace leveldb {

class InternalKeyComparator;
class MemTableIterator;

class MemTable {
 public:
  // MemTables are reference counted.  The initial reference count
  // is zero and the caller must call Ref() at least once.
```

```
explicit MemTable(const InternalKeyComparator& comparator);

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


// Increase reference count.
void Ref() { ++refs_; }

// Drop reference count.  Delete if no more references exist.
void Unref() {
  --refs_;
  assert(refs_ >= 0);
  if (refs_ <= 0) {
    delete this;
  }
}

// Returns an estimate of the number of bytes of data in use by this
// data structure. It is safe to call when MemTable is being modified.
size_t ApproximateMemoryUsage();

// Return an iterator that yields the contents of the memtable.
//
// The caller must ensure that the underlying MemTable remains live
// while the returned iterator is live.  The keys returned by this
// iterator are internal keys encoded by AppendInternalKey in the
// db/format.{h,cc} module.
Iterator* NewIterator();

// Add an entry into memtable that maps key to value at the
```

```
  // specified sequence number and with the specified type.
  // Typically value will be empty if type==kTypeDeletion.
  void Add(SequenceNumber seq, ValueType type, const Slice& key,
           const Slice& value);


  // If memtable contains a value for key, store it in *value and return true.
  // If memtable contains a deletion for key, store a NotFound() error
  // in *status and return true.
  // Else, return false.
  bool Get(const LookupKey& key, std::string* value, Status* s);

 private:
  friend class MemTableIterator;
  friend class MemTableBackwardIterator;

  struct KeyComparator {
    const InternalKeyComparator comparator;
    explicit KeyComparator(const InternalKeyComparator& c) : comparator(c) {}
    int operator()(const char* a, const char* b) const;
  };

  typedef SkipList<const char*, KeyComparator> Table;

  ~MemTable();  // Private since only Unref() should be used to delete it

  KeyComparator comparator_;
  int refs_;
  Arena arena_;
  Table table_;
};
```

```
}  // namespace leveldb
```

MemTable 包含比较器 comparator_ 、引用计数 refs_ 、内存池 arena_ 和跳表 table_ 四个成员变量。代码中首先前置声明了 InternalKeyComparator 类，该类的对象是 MemTable 的构造函数参数。该类定义为 db/dbformat.h 中，将在下文中介绍。随后 MemTable 禁止了拷贝构造和赋值操作符，不允许拷贝操作，这应该是内存池的副作用。MemTable 使用引用计数管理自己的生命周期，甚至其析构函数都是私有的。而接口 ApproximateMemoryUsage 实际上返回的是内存池中的内存使用量。接口方面提供了读写接口和迭代器接口。内部定义了结构体 KeyComparator ，对 InternalKeyComparator 进行了封装。为了弄清楚 InternalKeyComparator 到底是什么，我们先转到 db/dbformat.h 查看其定义：

```
enum ValueType { kTypeDeletion = 0x0, kTypeValue = 0x1 };

static const ValueType kValueTypeForSeek = kTypeValue;

typedef uint64_t SequenceNumber;
static const SequenceNumber kMaxSequenceNumber = ((0x1ull << 56) - 1);

struct ParsedInternalKey {
  Slice user_key;
  SequenceNumber sequence;
  ValueType type;
```

```
  ParsedInternalKey() {}  // Intentionally left uninitialized (for speed)
  ParsedInternalKey(const Slice& u, const SequenceNumber& seq, ValueType t)
      : user_key(u), sequence(seq), type(t) {}
  std::string DebugString() const;
};


// Return the length of the encoding of "key".
inline size_t InternalKeyEncodingLength(const ParsedInternalKey& key) {
  return key.user_key.size() + 8;
}

static uint64_t PackSequenceAndType(uint64_t seq, ValueType t) {
  assert(seq <= kMaxSequenceNumber);
  assert(t <= kValueTypeForSeek);
  return (seq << 8) | t;
}

void AppendInternalKey(std::string* result, const ParsedInternalKey& key) {
  result->append(key.user_key.data(), key.user_key.size());
  PutFixed64(result, PackSequenceAndType(key.sequence, key.type));
}

inline bool ParseInternalKey(const Slice& internal_key,
                             ParsedInternalKey* result) {
  const size_t n = internal_key.size();
  if (n < 8) return false;
  uint64_t num = DecodeFixed64(internal_key.data() + n - 8);
  uint8_t c = num & 0xff;
  result->sequence = num >> 8;
  result->type = static_cast<ValueType>(c);
```

```
  result->user_key = Slice(internal_key.data(), n - 8);
  return (c <= static_cast<uint8_t>(kTypeValue));
}

inline Slice ExtractUserKey(const Slice& internal_key) {

  assert(internal_key.size() >= 8);
  return Slice(internal_key.data(), internal_key.size() - 8);
}
```

为了方便阅读这里对函数进行了重排。首先定义了两种值类型 kTypeDeletion 和 kTypeValue，注意 kTypeDeletion 值较小。然后定义了 SequenceNumber 为 64 位无符号数，并且定义了其最大值为 $2^{56}-1$。然后是 ParsedInternalKey 结构体的定义，其内部包括 user_key、sequence 和 type。根据类的名字就可以猜到一点了，Internal Key 应该是对 User Key 的封装，在其基础上加入了序列号 sequence 和值类型 type。函数 PackSequenceAndType 中可以发现， sequence 占用 56bit，剩下 8bit 给 type，一共组成 64 位无符号数，下文称之为合成序列号。所以 InternalKeyEncodingLength 计算的长度是 User Key 的长度加 8。函数 AppendInternalKey 可以将 ParsedInternalKey 转为字符串，先将 User Key 编码进去，再将 64 位合成序列号加入，对应的解析函数 ParseInternalKey 为逆过程。继续：

```
class InternalKeyComparator : public Comparator {
 private:
  const Comparator* user_comparator_;

 public:
  explicit InternalKeyComparator(const Comparator* c) : user_comparator_(c) {}
```

```
  const char* Name() const override;
  int Compare(const Slice& a, const Slice& b) const override;
  void FindShortestSeparator(std::string* start,
                             const Slice& limit) const override;
  void FindShortSuccessor(std::string* key) const override;


  const Comparator* user_comparator() const { return user_comparator_; }

  int Compare(const InternalKey& a, const InternalKey& b) const;
};

const char* InternalKeyComparator::Name() const {
  return "leveldb.InternalKeyComparator";
}

int InternalKeyComparator::Compare(const Slice& akey, const Slice& bkey) const {
  // Order by:
  //    increasing user key (according to user-supplied comparator)
  //    decreasing sequence number
  //    decreasing type (though sequence# should be enough to disambiguate)
  int r = user_comparator_->Compare(ExtractUserKey(akey), ExtractUserKey(bkey));
  if (r == 0) {
    const uint64_t anum = DecodeFixed64(akey.data() + akey.size() - 8);
    const uint64_t bnum = DecodeFixed64(bkey.data() + bkey.size() - 8);
    if (anum > bnum) {
      r = -1;
    } else if (anum < bnum) {
      r = +1;
    }
  }
```

```
  return r;
}
```

InternalKeyComparator 继承于 LevelDB 中定义的比较器 Comparator，其源码位于 include/leveldb/comparator.h ，有兴趣自己阅读。其核心函数即为 Compare，比较两个 Key 的大小。 InternalKeyComparator 对 user_comparator_ 进行了封装，当比较 InternalKey 时，首先使用 user_comparator_ 对 User Key 进行比较，如果相等，则抽取合成序列号进行比较，序列号大的反而在顺序上更小，即降序排列。继续看 InternalKey 的定义：

```
class InternalKey {
 private:
  std::string rep_;

 public:
  InternalKey() {}  // Leave rep_ as empty to indicate it is invalid
  InternalKey(const Slice& user_key, SequenceNumber s, ValueType t) {
    AppendInternalKey(&rep_, ParsedInternalKey(user_key, s, t));
  }

  bool DecodeFrom(const Slice& s) {
    rep_.assign(s.data(), s.size());
    return !rep_.empty();
  }

  Slice Encode() const {
```

```
    assert(!rep_.empty());
    return rep_;
  }

  Slice user_key() const { return ExtractUserKey(rep_); }


  void SetFrom(const ParsedInternalKey& p) {
    rep_.clear();
    AppendInternalKey(&rep_, p);
  }

  void Clear() { rep_.clear(); }

  std::string DebugString() const;
};

inline int InternalKeyComparator::Compare(const InternalKey& a,
                                          const InternalKey& b) const {
  return Compare(a.Encode(), b.Encode());
}
```

InternalKey 存储的信息和 ParseInternalKey 一致，只是存储形式不同，前者直接使用字符串 rep_ 存储 User Key 和合成序列号，对应的比较函数则直接使用 InternalKeyComparator 对 rep_ 进行解析后的比较。该文件中还有 InternalFilterPolicy 的定义，同样是从 Internal Key 解析出 User Key 进行操作。继续看 LookupKey：

```
// A helper class useful for DBImpl::Get()
class LookupKey {
```

```
public:
 // Initialize *this for looking up user_key at a snapshot with
 // the specified sequence number.
 LookupKey(const Slice& user_key, SequenceNumber sequence);


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

 ~LookupKey();

 // Return a key suitable for lookup in a MemTable.
 Slice memtable_key() const { return Slice(start_, end_ - start_); }

 // Return an internal key (suitable for passing to an internal iterator)
 Slice internal_key() const { return Slice(kstart_, end_ - kstart_); }

 // Return the user key
 Slice user_key() const { return Slice(kstart_, end_ - kstart_ - 8); }

private:
 // We construct a char array of the form:
 //    klength  varint32               <-- start_
 //    userkey  char[klength]          <-- kstart_
 //    tag      uint64
 //                                    <-- end_
 // The array is a suitable MemTable key.
 // The suffix starting with "userkey" can be used as an InternalKey.
 const char* start_;
 const char* kstart_;
 const char* end_;
```

```
  char space_[200];  // Avoid allocation for short keys
};

LookupKey::LookupKey(const Slice& user_key, SequenceNumber s) {
  size_t usize = user_key.size();

  size_t needed = usize + 13;  // A conservative estimate
  char* dst;
  if (needed <= sizeof(space_)) {
    dst = space_;
  } else {
    dst = new char[needed];
  }
  start_ = dst;
  dst = EncodeVarint32(dst, usize + 8);
  kstart_ = dst;
  memcpy(dst, user_key.data(), usize);
  dst += usize;
  EncodeFixed64(dst, PackSequenceAndType(s, kValueTypeForSeek));
  dst += 8;
  end_ = dst;
}
```

在 LookupKey 中可以返回三种 Key，画图理理关系：

```
+-----------------------------------------------------+
|        +------------------------------------------+ |
|        |                 Internal Key             | |
| Length +-------------------------+----------+-----+ |
|        |        User Key         | Sequence | Type | |
```

```
|       +---------------------+---------+------+ |
+----------------------------------------------------+
```

回到内存数据库的定义中， KeyComparator 对 InternalKeyComparator 又进行了一次封装，来看下其具体函数定义：

```
static Slice GetLengthPrefixedSlice(const char* data) {
  uint32_t len;
  const char* p = data;
  p = GetVarint32Ptr(p, p + 5, &len);  // +5: we assume "p" is not corrupted
  return Slice(p, len);
}

int MemTable::KeyComparator::operator()(const char* aptr,
                                        const char* bptr) const {
  // Internal keys are encoded as length-prefixed strings.
  Slice a = GetLengthPrefixedSlice(aptr);
  Slice b = GetLengthPrefixedSlice(bptr);
  return comparator.Compare(a, b);
}
```

这里的 comparator 自然是 InternalKeyComparator，从 GetLengthPrefixedSlice 推测该函数的参数为 MemTable Key。继续看 MemTable::Add 和 MemTable::Get 的实现：

```
void MemTable::Add(SequenceNumber s, ValueType type, const Slice& key,
                   const Slice& value) {
```

```
  // Format of an entry is concatenation of:
  //  key_size     : varint32 of internal_key.size()
  //  key bytes    : char[internal_key.size()]
  //  value_size   : varint32 of value.size()
  //  value bytes  : char[value.size()]

  size_t key_size = key.size();
  size_t val_size = value.size();
  size_t internal_key_size = key_size + 8;
  const size_t encoded_len = VarintLength(internal_key_size) +
                             internal_key_size + VarintLength(val_size) +
                             val_size;
  char* buf = arena_.Allocate(encoded_len);
  char* p = EncodeVarint32(buf, internal_key_size);
  memcpy(p, key.data(), key_size);
  p += key_size;
  EncodeFixed64(p, (s << 8) | type);
  p += 8;
  p = EncodeVarint32(p, val_size);
  memcpy(p, value.data(), val_size);
  assert(p + val_size == buf + encoded_len);
  table_.Insert(buf);
}

bool MemTable::Get(const LookupKey& key, std::string* value, Status* s) {
  Slice memkey = key.memtable_key();
  Table::Iterator iter(&table_);
  iter.Seek(memkey.data());
  if (iter.Valid()) {
    // entry format is:
    //    klength  varint32
```

```
    //    userkey  char[klength]
    //    tag      uint64
    //    vlength  varint32
    //    value    char[vlength]
    // Check that it belongs to same user key.  We do not check the

    // sequence number since the Seek() call above should have skipped
    // all entries with overly large sequence numbers.
    const char* entry = iter.key();
    uint32_t key_length;
    const char* key_ptr = GetVarint32Ptr(entry, entry + 5, &key_length);
    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)) {
        case kTypeValue: {
          Slice v = GetLengthPrefixedSlice(key_ptr + key_length);
          value->assign(v.data(), v.size());
          return true;
        }
        case kTypeDeletion:
          *s = Status::NotFound(Slice());
          return true;
      }
    }
  }
  return false;
}
```

MemTable::Add 的注释中写明了条目的格式，除了图中 Key 部分的存储外，条目还

MemTable::Add 的注释中写明了条目的格式，除了图中 Key 部分的存储外，条目还包括 Value 的存储，打包好后插入跳表中。 MemTable::Get 中则根据 MemTable Key 查找对应的条目，查到后使用 user_comparator 进一步比较两者的 User Key 是否一致，如

果完全一致并且值类型不是删除，就把条目中的 Value 读出来放到结果中。最后看下迭代器的实现：

```
static const char* EncodeKey(std::string* scratch, const Slice& target) {
  scratch->clear();
  PutVarint32(scratch, target.size());
  scratch->append(target.data(), target.size());
  return scratch->data();
}

class MemTableIterator : public Iterator {
 public:
  explicit MemTableIterator(MemTable::Table* table) : iter_(table) {}

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

  ~MemTableIterator() override = default;

  bool Valid() const override { return iter_.Valid(); }
  void Seek(const Slice& k) override { iter_.Seek(EncodeKey(&tmp_, k)); }
  void SeekToFirst() override { iter_.SeekToFirst(); }
  void SeekToLast() override { iter_.SeekToLast(); }
  void Next() override { iter_.Next(); }
```

```
  void Prev() override { iter_.Prev(); }
  Slice key() const override { return GetLengthPrefixedSlice(iter_.key()); }
  Slice value() const override {
    Slice key_slice = GetLengthPrefixedSlice(iter_.key());
    return GetLengthPrefixedSlice(key_slice.data() + key_slice.size());

  }

  Status status() const override { return Status::OK(); }

 private:
  MemTable::Table::Iterator iter_;
  std::string tmp_;  // For passing to EncodeKey
};

Iterator* MemTable::NewIterator() { return new MemTableIterator(&table_); }
```

MemTableIterator 对跳表的迭代器进行了一层封装，跳表的条目中编码了 Key 和 Value， MemTableIterator 中对其进行了对应的解析。至此，内存数据库的代码就分析完了。

总结

上一篇和本篇分析了 LevelDB 写操作的具体过程，分析了 WriteBatch 、 Log 和 MemTable 的代码，理解了高性能写操作的原理。不过内存数据库的大小毕竟是有限的，总会需要把内存中的数据持久化到硬盘中，并且还需要提供高速的硬盘数据查询操作，这些又是怎样实现的呢？且听下回分解！

0 comments

Write Preview Aa

Sign in to comment

Styling with Markdown is supported

Sign in with GitHub