【前言：看了一点 oceanbase，没有意志力继续坚持下去了，暂时就此中断，基本上算把 master 看完了，比较重要的 update server 和 merge server 代码却没有细看。中间又陆续研究了 hadoop 的源码，主要是 name node 和写入 pipeline。主要的目的是想看看 name node 对 namespace 的管理，以及 hadoop 在写入操作时，client、data node 和 name node 之间是如何交互的，特别是涉及到 namenode 的，以及写入出现错误时的处理逻辑。没办法，和分布式存储扯不开了。

其后看到了 Leveldb，除去测试部分，代码不超过 1.5w 行。这是一个单机 k/v 存储系统，决定看完它，并把源码分析完整的写下来，还是会很有帮助的。我比较厌烦太复杂的东西，而 Leveldb 的逻辑很清晰，代码不多、风格很好，功能就不用讲了，正合我的胃口。 BTW，分析 Leveldb 也参考了网上一些朋友写的分析 blog，如巴山独钓。】

Leveldb 源码分析

2012 年 1 月 21 号开始研究下 leveldb 的代码，Google 两位大牛开发的单机 KV 存储系统，涉及到了 skip list、内存 KV table、LRU cache 管理、table 文件存储、operation log 系统等。先从边边角角的小角色开始扫。

不得不说，Google 大牛的代码风格太好了，读起来很舒服，不像有些开源项目，很快就看不下去了。

开始之前先来看看 Leveldb 的基本框架，几大关键组件，如图 1-1 所示。

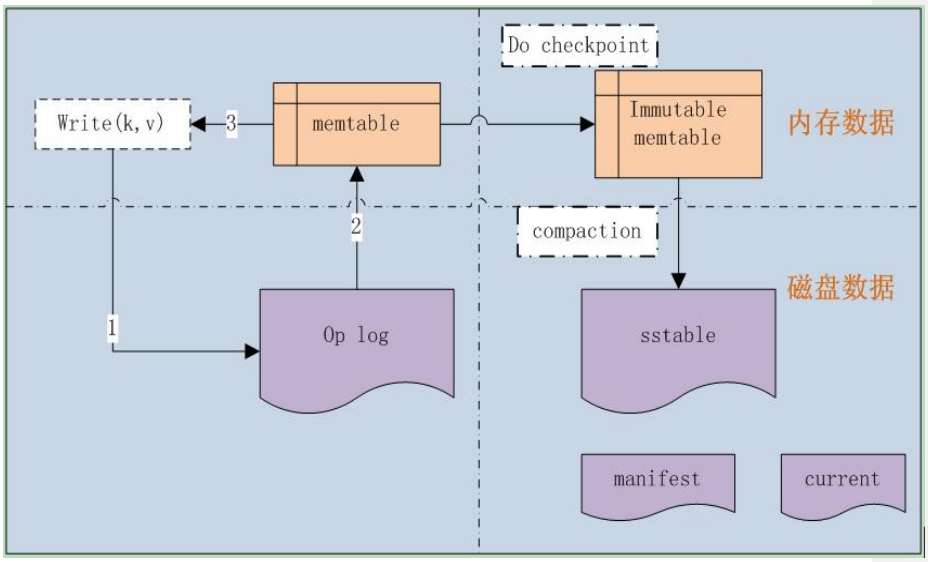


图 1-1

Leveldb 是一种基于 operation log 的文件系统，是 Log-Structured-Merge Tree 的典型实现。LSM 源自 Ousterhout 和 Rosenblum 在 1991 年发表的经典论文<<The Design and Implementation of a Log-Structured File System >>。

由于采用了 op log，它就可以把随机的磁盘写操作，变成了对 op log 的 append 操作，因此提高了 IO 效率，最新的数据则存储在内存 memtable 中。

当 op log 文件大小超过限定值时，就定时做 check point。Leveldb 会生成新的 Log 文件

批注 [g1]: When

和 Memtable，后台调度会将 Immutable Memtable 的数据导出到磁盘，形成一个新的 SSTable 文件。SSTable 就是由内存中的数据不断导出并进行 Compaction 操作后形成的，而且 SSTable 的所有文件是一种层级结构，第一层为 Level 0，第二层为 Level 1，依次类推，层级逐渐增高，这也是为何称之为 LevelDb 的原因。

1 一些约定

先说下代码中的一些约定：

1.1 字节序

Leveldb 对于数字的存储是 **little-endian** 的，在把 int32 或者 int64 转换为 char*的函数中，是按照先低位再高位的顺序存放的，也就是 little-endian 的。

1.2 VarInt

把一个 int32 或者 int64 格式化到字符串中，除了上面说的 little-endian 字节序外，大部分还是**变长存储的**，也就是 VarInt。对于 VarInt，每 byte 的有效存储是 7bit 的，用最高的第八 bit 位来表示是否结束，如果是 1 就表示后面还有一个 byte 的数字，否则表示结束。直接见 Encode 和 Decode 函数。

在操作 log 中使用的是 Fixed 存储格式。

1.3 字符比较

是基于 unsigned char 的，而非 char。

2 基本数据结构

别看是基本数据结构，有些也不是那么简单的，像 LRU Cache 管理和 Skip list 那都算是 leveldb 的核心数据结构。

2.1 class Slice (slice.h)

Leveldb 中的基本数据结构，它包括 size_t size_和一个指向外部字节数组的指针(char *data)。和 string 一样，允许字符串中包含'\0'。

提供一些基本接口，可以把 const char*和 string 转换为 Slice；把 Slice 转换为 string，取得数据指针 const char*。

这样做的好处是：直接操控指针避免不必要的数据拷贝（《实现解析》）。

2.2 class Status (status.h)

Leveldb 中的返回状态，将错误号和错误信息封装成 Status 类，统一进行处理。并定义了几种具体的返回状态，如成功或者文件不存在等。

为了节省空间 Status 并没有用 std::string 来存储错误信息，而是将返回码(code), 错误信息 message 及长度打包存储于一个字符串数组中。

成功状态 OK 是 NULL state_，否则 state_ 是一个包含如下信息的数组:

state_[0..3] == 消息 length of message
state_[4] == 消息 code
state_[5..] ==消息 message

2.3 Arena

Leveldb 的简单的内存池，它所作的工作十分简单，申请内存时，将申请到的内存块放入 std::vector<char*> blocks_;中，在 Arena 的生命周期结束后，统一释放掉所有申请到的内存，内部结构如图 2.3-1 所示。

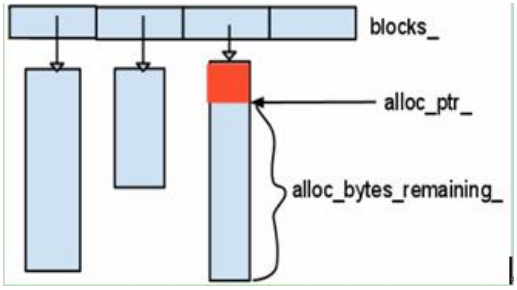


图 2.3-1

Arena 主要提供了两个申请函数：其中一个直接分配内存(Allocate)，另一个可以申请对齐的内存空间(AllocateAligned)。Arena 没有直接调用 delete/free 函数，而是由 Arena 的析构函数统一释放所有的内存。

应该说这是和 leveldb 特定的应用场景相关的，比如一个 memtable 使用一个 Arena，当 memtable 被释放时，由 Arena 统一释放其内存。

2.4 class SkipList

Skip list(跳跃表）是一种可以代替平衡树的数据结构。Skip lists 应用概率来保证平衡，平衡树采用严格的旋转（比如平衡二叉树有左旋右旋）来保证平衡，因此 Skip list 比较容易实现，而且相比平衡树有着**较高的运行效率**。

从概率上保持数据结构的平衡比显式的保持数据结构平衡要简单的多。对于大多数应用，用 skip list 要比用树更自然，算法也会相对简单。由于 skip list 比较简单，实现起来会比较容易，虽然和平衡树有着相同的时间复杂度(O(logn))，但是 skip list 的常数项相对小很多。skip list 在空间上也比较节省。一个节点平均只需要 1.333 个指针（甚至更少），并且不需要存储保持平衡的变量。

如图 2.4-1 所示。

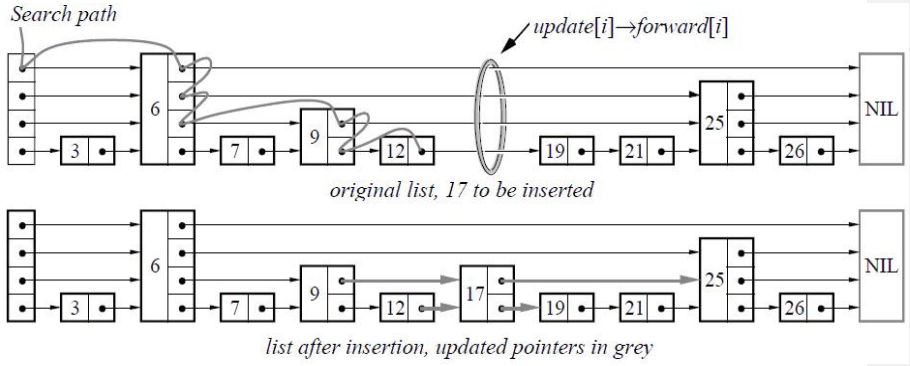


图 2.4-1

在 Leveldb 中，skip list 是实现 memtable 的核心数据结构，memtable 的 KV 数据都存储在 skip

list 中。

2.5 class Cache

Leveldb 内部通过双向链表实现了一个标准版的 LRUCache，先上个示意图，看看几个数据之间的关系，如图 2.5-1。

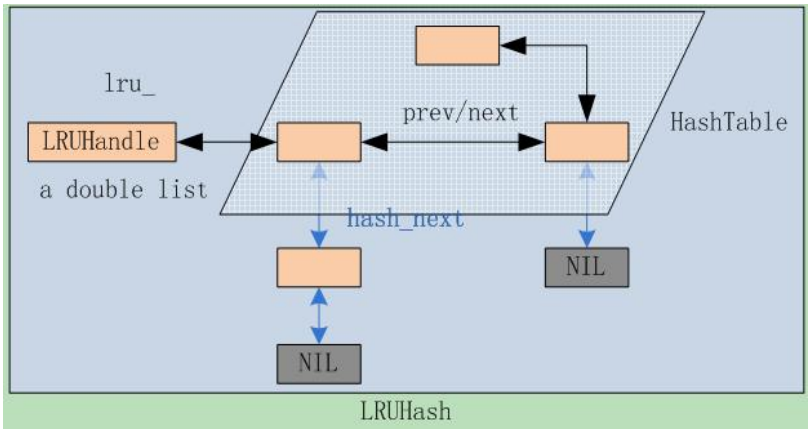


图 2.5-1

接下来说说 Leveldb 实现 LRUCache 的几个步骤，很直观明了。

- S1 定义一个 struct LRUHandle 结构体，代表 cache 中的元素。它包含了几个主要的成员：

➢ void* value; //这个存储的是 cache 的数据；

➢ void (*deleter)(const Slice&, void* value);//这个是数据从 Cache 中清除时执行的清理函数；

后面的三个成员事关 LRUCache 的数据的组织结构：

➢ LRUHandle *next_hash;

//指向节点在 hash table 链表中的下一个 hash(key)相同的元素，在有碰撞时 Leveldb 采用的是链表法。最后一个节点的 next_hash 为 NULL。

➢ LRUHandle *next, *prev;

//节点在双向链表中的前驱后继节点指针，**所有的 cache 数据都是存储在一个双向 list 中**，最前面的是最新加入的，每次新加入的位置都是 head->next。所以每次剔除的规则就是剔除 list tail。

➢ char key_data[1]; // Beginning of key

//一个巧妙的 key_data，保存了 key 的第一个字符，key_data 这个指针值放哪了？？

- S2 Leveldb 自己实现了一个 hash table：class HandleTable，而不是使用系统提供的 hash table。这个类就是基本的 hash 操作：Lookup、Insert 和 Delete。Hash table 的作用是根据 key 快速查找元素是否在 cache 中，并返回 LRUHandle 节点指针，由此就能快速定位节点在 HandleTable 和 LRUHandle 中的位置。

  它是通过 LRUHandle 的成员 next_hash 组织起来的。

  HandleTable 使用 LRUHandle **list_存储所有的 hash 节点，其实就是一个二维数组，一维是不同的 hash(key)，另一维则是相同 hash(key)的碰撞 list。

每次当 hash 节点数超过当前一维数组的长度后，都会做 Resize 操作：

LRUHandle** new_list = new LRUHandle*[new_length];

然后复制 list_到 new_list 中，并删除旧的 list_。

- S3 基于 HandleTable 和 LRUHandle，实现了一个标准的 class LRUcache，并内置了 port::Mutex mutex_;保护锁，是线程安全的。
  其中存储所有数据的双向链表是 LRUHandle lru_，这是一个 list head；
  Hash 表则是 HandleTable table_；

```
LRUHandle** FindPointer(const Slice& key, uint32_t hash) {      //通过 HashTable (list_)进行查找
    LRUHandle** ptr = &list_[hash & (length_ - 1)];
    while (*ptr != NULL &&
           ((*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++) {     //重组 HashTable（list_ 复制到 new_list）
      LRUHandle* h = list_[i];
      while (h != NULL) {
        LRUHandle* next = h->next_hash;
        Slice key = h->key();
        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;
  }
```

- S4 class ShardedLRUCache 类，实际上到 S3，一个标准的 LRU Cache 已经实现了，为何还要更近一步呢？答案就是速度！
  为了多线程访问，尽可能快速，减少锁开销，ShardedLRUCache 内部有 16 个 LRUCache，

查找 Key 时首先计算 key 属于哪一个分片，分片的计算方法是取 32 位 hash 值的高 4 位，然后在相应的 LRUCache 中进行查找，这样就大大减少了多线程的访问锁的开销。

LRUCache shard_[kNumShards]

它就是一个包装类，实现都在 LRUCache 类中。

2.6 class Random

The comments: A very simple random number generator. Not especially good at generating **truly random bits**, but good enough for our needs in this package.

2.7 function: Hash.cc : Hash()

2.8 namespace: crc32c

// A portable implementation of crc32c, optimized to handle four bytes at a time.

2.9 class Histogram

3 int Coding 整形编码

“Coding.h Coding.cc”

轻松一刻，前面约定中讲过 Leveldb 使用了很多 VarInt 型编码，典型的如后面将涉及到的各种 key。其中的编码、解码函数分为 VarInt 和 FixedInt 两种。int32 和 int64 操作都是类似的。

3.1 Encode(编码)

首先是 **FixedInt** 编码，直接上代码，很简单明了。

```
void EncodeFixed32(char* buf, uint32_t value) {
#if __BYTE_ORDER == __LITTLE_ENDIAN
  memcpy(buf, &value, sizeof(value));
#else
  buf[0] = value & 0xff;
  buf[1] = (value >> 8) & 0xff;
  buf[2] = (value >> 16) & 0xff;
  buf[3] = (value >> 24) & 0xff;
#endif
}
```

下面是 **VarInt** 编码，int32 和 int64 格式，代码如下，有效位是 7bit 的，因此把 uint32 按

7bit 分割，对 unsigned char 赋值时，超出 0xFF 会自动截断，因此直接*(ptr++) = v|B 即可，不需要再把(v|B)与 0xFF 作&操作。

```
char* EncodeVarint32(char* dst, uint32_t v) {
  // Operate on characters as unsigneds
  unsigned char* ptr = reinterpret_cast<unsigned char*>(dst);
  static const int B = 128;
  if (v < (1<<7)) {
    *(ptr++) = v;
  } else if (v < (1<<14)) {
    *(ptr++) = v | B;
    *(ptr++) = v>>7;
  } else if (v < (1<<21)) {
    *(ptr++) = v | B;
    *(ptr++) = (v>>7) | B;
    *(ptr++) = v>>14;
  } else if (v < (1<<28)) {
    *(ptr++) = v | B;
    *(ptr++) = (v>>7) | B;
    *(ptr++) = (v>>14) | B;
    *(ptr++) = v>>21;
  } else {
    *(ptr++) = v | B;
    *(ptr++) = (v>>7) | B;
    *(ptr++) = (v>>14) | B;
    *(ptr++) = (v>>21) | B;
    *(ptr++) = v>>28;
  }
  return reinterpret_cast<char*>(ptr);
}

char* EncodeVarint64(char* dst, uint64_t v) {// 对于 uint64，直接循环

  static const int B = 128;
  unsigned char* ptr = reinterpret_cast<unsigned char*>(dst);
  while (v >= B) {
    *(ptr++) = (v & (B-1)) | B;
    v >>= 7;
  }
  *(ptr++) = static_cast<unsigned char>(v);
  return reinterpret_cast<char*>(ptr);
}
```

3.2 Decode

Fixed Int 的 Decode 操作，代码如下：

```
inline uint32_t DecodeFixed32(const char* ptr)
{
  if (port::kLittleEndian) {
    uint32_t result;
    memcpy(&result, ptr,sizeof(result));  // gcc optimizes this to a plain load
    return result;
  } else {
    return((static_cast<uint32_t>(static_cast<unsigned char>(ptr[0])))
       |(static_cast<uint32_t>(static_cast<unsigned char>(ptr[1])) <<8)
       | (static_cast<uint32_t>(static_cast<unsignedchar>(ptr[2])) << 16)
       |(static_cast<uint32_t>(static_cast<unsigned char>(ptr[3])) <<24));
  }
}
```

再来看看 VarInt 的解码，很简单，依次读取 1byte，直到最高位为 0 的 byte 结束，取低 7bit，作(<<7)移位操作组合成 Int。看代码：

```
const char* GetVarint32Ptr(const char* p, const char* limit,   uint32_t* value)
{
  if (p < limit) {
    uint32_t result =*(reinterpret_cast<const unsigned char*>(p));
    if ((result & 128) == 0) {
      *value = result;
      return p + 1;
    }
  }
  return GetVarint32PtrFallback(p,limit, value);
}
const char* GetVarint32PtrFallback(const char* p, const char* limit,uint32_t* value)
{
  uint32_t result = 0;
  for (uint32_t shift = 0; shift<= 28 && p < limit; shift += 7) {
    uint32_t byte =*(reinterpret_cast<const unsigned char*>(p));
    p++;
    if (byte & 128) { // More bytes are present
      result |= ((byte & 127)<< shift);
    } else {
      result |= (byte <<shift);
      *value = result;
      returnreinterpret_cast<const char*>(p);
    }
  }
  return NULL;
}
```

4 class Memtable

“MemTable.cc MemTable.h SkipList.h Dbformat.cc Dbformat.h”

Memtable 是 leveldb 很重要的一块，leveldb 的核心之一。我们肯定关注 KV 数据在 Memtable 中是如何组织的，秘密在 Skip list 中。

4.1 用途

在 Leveldb 中，所有内存中的 KV 数据都存储在 Memtable 中，物理 disk 则存储在 SSTable 中。在系统运行过程中，如果 Memtable 中的数据占用内存到达指定值(Options.write_buffer_size) size_t write_buffer_size;，则 Leveldb 就自动将 Memtable 转换为 immutable Memtable，并自动生成新的 Memtable，也就是 Copy-On-Write 机制了。

Immutable Memtable 则被新的线程 Dump 到磁盘中，Dump 结束则该 Immutable Memtable 就可以释放了。因名知意，Immutable Memtable 是只读的。

所以可见，最新的数据都是存储在 Memtable 中的，Immutable Memtable 和物理 SSTable 则是某个时点的数据。

为了防止系统 down 机导致内存数据 Memtable 或者 Immutable Memtable 丢失，leveldb 自然也依赖于 log 机制来保证可靠性了。

Memtable 提供了写入 KV 记录，删除以及读取 KV 记录的接口，但是事实上 Memtable 并不执行真正的删除操作,删除某个 Key 的 Value 在 Memtable 内是作为插入一条记录实施的，但是会打上一个 Key 的删除标记，真正的删除操作在后面的 Compaction 过程中，**lazy delete**(--GL--总结的不错) 。

4.2 核心是 Skip list

另外，Memtable 中的 KV 对是根据 Key 排序的，leveldb 在插入等操作时保证 key 的有序性。想想，前面看到的 Skip list 不正是合适的人选吗，因此 Memtable 的核心数据结构是一个 Skip list，Memtable 只是一个接口类。当然随之而来的一个问题就是 Skip list 是如何组织 KV 数据对的，在后面分析 Memtable 的插入、查询接口时我们将会看到答案。

4.3 接口说明

先来看看 Memtable 的接口：

```
void Ref() { ++refs_; }
void Unref();
Iterator* NewIterator();
void Add(SequenceNumber seq, ValueType type, const Slice& key, const Slice& value);
bool Get(const LookupKey& key, std::string* value, Status* s);
```

首先 Memtable 是**基于引用计数的机制**，如果引用计数为 0，则在 Unref 中删除自己，Ref 和 Unref 就是干这个的。

NewIterator 是返回一个迭代器，可以遍历访问 memtable 的内部数据，很好的设计思想，这种方式隐藏了 table 的内部实现。外部调用者必须保证使用 Iterator 访问 Memtable 的时候该 Memtable 是 live 的。

Add 和 Get 是添加和获取记录的接口，没有 Delete，还记得前面说过，memtable 的 delete 实际上是插入一条 type 为 kTypeDeletion 的记录。

4.4 类图

先来看看 Memtable 相关的整体类层次吧，并不复杂，还是相当清晰的。见图 4.4-1。

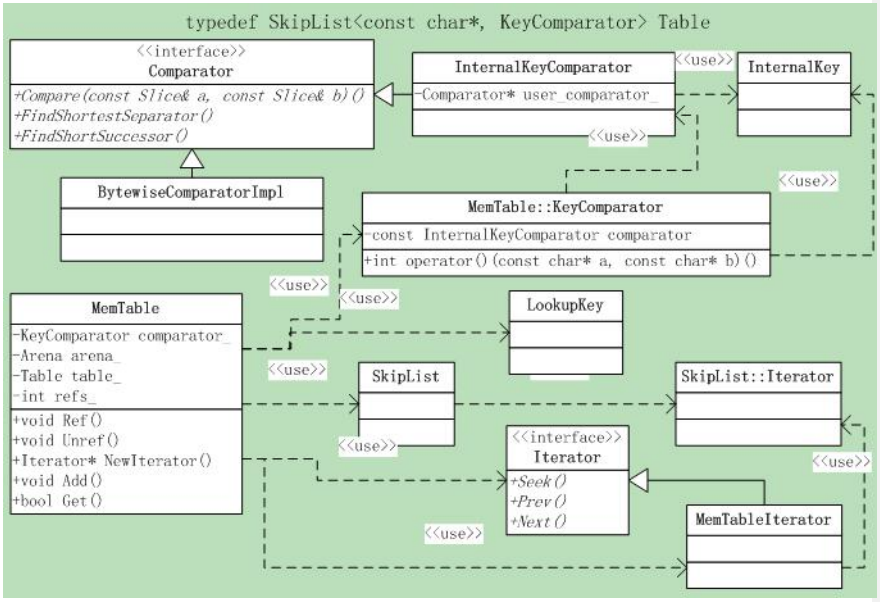


图 4.4-1

4.5 Key 结构 ——“Dbformat.cc && Dbformat.h”

Memtable 是一个 KV 存储结构，那么这个 key 肯定是个重点了，在分析接口实现之前，有必要仔细分析一下 Memtable 对 key 的使用。

这里面有 5 个 key 的概念，可能会让人混淆，下面就来一个一个的分析。

4.5.1 InternalKey & ParsedInternalKey & User Key

InternalKey（class InternalKey）是一个复合概念，是有几个部分组合成的一个 key，ParsedInternalKey（struct ParsedInternalKey）就是对 InternalKey 分拆后的结果，先来看看 ParsedInternalKey 的成员，这是一个 struct：

>Slice user_key;

>SequenceNumber sequence;

>ValueType type;

class InternalKey 是一个只存储了一个 string，它使用一个 DecodeFrom() 函数将 Slice 类型的 InternalKey 解码出 string 类型的 InternalKey.

void **DecodeFrom**(const Slice& s) { rep_.assign(s.data(), s.size()); }

也就是说 InternalKey 是由 User_key.data + SequenceNumber + ValueType 组合而成的，顺便先分析下几个 Key 相关的函数，它们是了解 Internal Key 和 User Key 的关键。

首先是 InternalKey 和 ParsedInternalKey 相互转换的两个函数，如下。

Inline bool **ParseInternalKey** (const Slice& internal_key, ParsedInternalKey* result);

```
//将 internal_key（Slice）解析出来为 result

void AppendInternalKey (std::string* result, const ParsedInternalKey& key);
// Append the serialization of "key" to *result.
//将 key（ParsedInternalKey）序列化为 result（Internel key）
```

函数实现很简单，就是字符串的拼接与把字符串按字节拆分，代码略过。

根据实现，容易得到 **InternalKey 的格式**为：

| User_key.data (string)| sequence number (7 B) | value type (1 B) |

由此还可知道 sequence number 大小是 7 bytes，sequence number 是所有基于 op log 系统的关键数据，它唯一指定了不同操作的时间顺序。

把 user key 放到前面的原因是，这样对同一个 user key 的操作就可以按照 sequence number 顺序连续存放了，不同的 user key 是互不相干的，因此把它们的操作放在一起也没有什么意义。

另外用户可以为 user key 定制比较函数，系统默认是字母序的。

下面的两个函数是分别从 InternalKey 中拆分出 User Key 和 Value Type 的，非常直观，代码也附上吧。

```
inline Slice ExtractUserKey(const Slice& internal_key)
{
  assert(internal_key.size() >= 8);
  return Slice(internal_key.data(), internal_key.size() - 8);
}

inline ValueType ExtractValueType(const Slice& internal_key)
{
  assert(internal_key.size() >= 8);
  const size_t n = internal_key.size();
  uint64_t num = DecodeFixed64(internal_key.data() + n - 8);
  unsigned char c = num & 0xff;
  return static_cast<ValueType>(c);
}
```

4. 5. 2 LookupKey & Memtable Key

Memtable 的查询接口传入的是 LookupKey（class LookupKey），它也是由 User Key 和 Sequence Number 组合而成的，从其构造函数：LookupKey(const Slice& user_key, SequenceNumber s)中分析出 LookupKey 的格式为：

| Size (Varint32)| User key (string) | sequence number (7 bytes) | value type (1 byte) |

两点：

>这里的 Size 是 user key 长度+8，相当于 InteralKey 的长度；

>value type 是 kValueTypeForSeek，它等于 kTypeValue。

>由于 LookupKey 的 size 是变长存储的，因此它使用 kstart_记录了 user key string 的起始地址，否则将不能正确的获取 size 和 user key；

LookupKey 导出了三个函数，可以分别从 LookupKey 得到 Internal Key，Memtable Key 和 User Key，如下：

```
 // 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); }
```

其中 start_是 LookupKey 字符串的开始，end_是结束，kstart_是 start_+ sizeof(varint32)，也就是 user key 字符串的起始地址。

```
// We construct a char array of the form:
//      klenKgth    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.
```

4.6 class Comparator
“Comparator.cc Comparator.h”

弄清楚了 key，接下来就要看看 key 的使用了，先从 Comparator 开始分析。首先 Comparator 是一个抽象类，导出了几个接口。

其中 Name()和 Compare()接口都很明了，

```
// Three-way comparison.  Returns value:
//   < 0 iff "a" < "b",
//   == 0 iff "a" == "b",
//   > 0 iff "a" > "b"
virtual int Compare(const Slice& a, const Slice& b) const = 0;
```

```
// The name of the comparator.  Used to check for comparator
// mismatches (i.e., a DB created with one comparator is
// accessed using a different comparator.
//
// The client of this package should switch to a new name whenever
// the comparator implementation changes in a way that will cause
// the relative ordering of any two keys to change.
//
// Names starting with "leveldb." are reserved and should not be used
// by any clients of this package.
virtual const char* Name() const = 0;
```

另外的两个 Find xxx 接口都有什么功能呢，直接看程序注释：

```
// Advanced functions: these are used to reduce the space requirements
// for internal data structures like index blocks.
// 这两个函数：用于减少像 index blocks 这样的内部数据结构占用的空间
```

```
4.  // 其中的*start 和*key 参数都是 IN OUT 的。
5.
6.  // If *start < limit, changes *start to a short string in [start,limit).
7.  // Simple comparator implementations may return with *start unchanged,
8.  // i.e., an implementation of this method that does nothing is correct.
9.  // 这个函数的作用就是：如果*start < limit，就在[start,limit)中找到一个
10. // 短字符串，并赋给*start 返回
11. // 简单的 comparator 实现可能不改变*start，这也是正确的
12. // e.g. *start = “the car”  limit = “the color”
13. // *start 最后变为：”the cb”  [diff_size = 6, diff_index = 5]
14. virtual void FindShortestSeparator(std::string* start, const Slice& limit) c
    onst = 0;
15.
16. // Changes *key to a short string >= *key.
17. // Simple comparator implementations may return with *key unchanged,
18. // i.e., an implementation of this method that does nothing is correct.
19. //这个函数的作用就是：找一个>= *key 的短字符串
20. //简单的 comparator 实现可能不改变*key，这也是正确的
21. virtual void FindShortSuccessor(std::string* key) const = 0;
```

其中的实现类有两个，一个是内置的 BytewiseComparatorImpl，另一个是InternalKeyComparator。下面分别来分析。

4.6.1 class BytewiseComparatorImpl

首先是重载的 Name 和比较函数 Compare()，比较函数如其名，就是字符串比较，如下：

virtual const char* **Name**() const {return"leveldb.BytewiseComparator";}

virtual int **Compare**(const Slice& a, const Slice& b) const {return a.compare(b);}

再来看看 Byte wise 的 comparator 是如何实现 FindShortestSeparator()的，没什么特别的，代码 + 注释如下：

```
virtual void FindShortestSeparator(std::string* start, const Slice& limit) const
{
  // 首先计算共同前缀字符串的长度
   size_t min_length = std::min(start->size(), limit.size());
  size_t diff_index = 0;
  while ((diff_index < min_length) && ((*start)[diff_index] == limit[diff_index])) {
   diff_index++;
  }
  if (diff_index >= min_length) {
   // 说明*start 是 limit 的前缀，或者反之，此时不作修改，直接返回
   } else {
    // 尝试执行字符 start[diff_index]++，设置 start 长度为 diff_index+1，并返回
    // ++条件：字符< oxff 并且字符+1 < limit 上该 index 的字符
    uint8_t diff_byte = static_cast<uint8_t>((*start)[diff_index]);
    if (diff_byte < static_cast<uint8_t>(0xff) &&
     diff_byte + 1 < static_cast<uint8_t>(limit[diff_index])) {
```

```
      (*start)[diff_index]++;
      start->resize(diff_index + 1);
      assert(Compare(*start, limit) < 0);
    }
  }
}
```

最后是 FindShortSuccessor()，这个更简单了，代码+注释如下：

```
virtual void FindShortSuccessor(std::string* key) const
{
  // 找到第一个可以++的字符，执行++后，截断字符串；
  // e.g. "\xff\xff\x30\x34" → "\xff\xff\x31"
  // 如果找不到说明*key 的字符都是 0xff 啊，那就不作修改，直接返回
  size_t n = key->size();
  for (size_t i = 0; i < n; i++) {
   const uint8_t byte = (*key)[i];
   if (byte != static_cast<uint8_t>(0xff)) {
    (*key)[i] = byte + 1;
    key->resize(i+1);
    return;
   }
  }
}
```

Leveldb 内建的基于 Bytewise 的 comparator 类就这么多内容了，下面再来看看 InternalKeyComparator。

4.6.2 class InternalKeyComparator

从上面对 Internal Key 的讨论可知，由于它是由 user key 和 sequence number 和 value type 组合而成的，因此它还需要 user key 的比较，所以 InternalKeyComparator 有一个 Comparator* user_comparator_ 成员，用于 user key 的比较。在 leveldb 中的名字为："leveldb.InternalKeyComparator"，下面来看看比较函数：

Compare(const Slice& akey, const Slice& bkey)，代码很简单，其比较逻辑是：

- S1 首先比较 user key，基于用户设置的 user_comparator_，如果 user key 不相等就直接返回比较结果；否则执行进入 S2；
- S2 取出 8 字节的 sequence number | value type，如果 akey 的 > bkey 的则返回-1，如果 akey 的<bkey 的返回 1，相等返回 0；

由此可见其排序比较依据依次是：

```
1.  // Order by:
2.  //    increasing user key (according to user-supplied comparator)
3.  //    decreasing sequence number
4.  //    decreasing type (though sequence# should be enough to disambiguate)
```

虽然比较时 value type 并不重要，因为 sequence number 是唯一的，但是直接取出 8byte 的 sequence number | value type，然后做比较更方便，不需要再次移位提取出 7byte 的 sequence number，又何乐而不为呢。这也是把 value type 安排在低 1byte 的好处吧，排序的

两个依据就是 user key 和 sequence number。

接下来就该看看其 FindShortestSeparator()函数实现了，该函数取出 Internal Key 中的 user key 字段，根据 user 指定的 comparator 找到并替换 start，如果 start 被替换了，就用新的 start 更新 Internal Key，并使用最大的 sequence number。否则保持不变。

函数声明：

```
5.  void InternalKeyComparator::FindShortestSeparator(std::string* start, const
    Slice& limit) const;
```

函数实现：

```
6.  {
7.  // 尝试更新 user key，基于指定的 user comparator
8.  Slice user_start = ExtractUserKey(*start);
9.  Slice user_limit = ExtractUserKey(limit);
10. std::string tmp(user_start.data(), user_start.size());
11. user_comparator_->FindShortestSeparator(&tmp, user_limit);
12. if(tmp.size()<user_start.size()&&user_comparator_->Compare(user_start, tmp)<
    0)
13. {
14.   // if user key 在物理上长度变短了，但其逻辑值变大了.生产新的*start 时，
15.   // 使用最大的 sequence number，以保证排在相同 user key 记录序列的第一个
16.    PutFixed64(&tmp, PackSequenceAndType(kMaxSequenceNumber,kValueTypeForSeek
    ));
17.   assert(this->Compare(*start, tmp) < 0);
18.   assert(this->Compare(tmp, limit) < 0);
19.   start->swap(tmp);
20. } //endif
21. } //end func
```

接下来是 FindShortSuccessor(std::string* key)函数，该函数取出 Internal Key 中的 user key 字段，根据 user 指定的 comparator 找到并替换 key，如果 key 被替换了，就用新的 key 更新 Internal Key，并使用最大的 sequence number。否则保持不变。实现逻辑如下：

```
1.  void InternalKeyComparator::FindShortSuccessor(std::string* key) const {
2.  Slice user_key = ExtractUserKey(*key);
3.  // 尝试更新 user key，基于指定的 user comparator
4.  std::string tmp(user_key.data(), user_key.size());
5.  user_comparator_->FindShortSuccessor(&tmp);
6.  if(tmp.size()<user_key.size() && user_comparator_->Compare(user_key, tmp)<0)
     {
7.     // user key 在物理上长度变短了，但其逻辑值变大了.生产新的*start 时，
8.     // 使用最大的 sequence number，以保证排在相同 user key 记录序列的第一个
9.      PutFixed64(&tmp, PackSequenceAndType(kMaxSequenceNumber,kValueTypeForSee
    k));
```

```
    assert(this->Compare(*key, tmp) < 0);
    key->swap(tmp);
  }  //end if
}
```

4.6.3 Memtable::Insert()

把相关的 Key 和 Key Comparator 都弄清楚后，是时候分析 memtable 本身了。首先是向 memtable 插入记录的接口，函数原型如下：

```
void Add(SequenceNumber seq, ValueType type, const Slice& key, const Slice&
value);
```

代码实现如下：

```
// 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);
```

根据代码，我们可以分析出 KV 记录在 skip list 的存储格式等信息，首先总长度为：VarInt(Internal Key size) len + internal key + VarInt(value) len + value。它们的相互衔接也就是 KV 的存储格式：

| VarInt(Internal Key size) len | internal key |VarInt(value) len |value|

其中前面说过：

internal key = |user key |sequence number |type |

Internal key size = user key size + 8

4.6.4 Memtable::Get()

Memtable 的查找接口，根据一个 LookupKey 找到响应的记录，函数声明：

```
1.  bool MemTable::Get(const LookupKey& key, std::string* value, Status* s)
2.  {
```

函数实现如下：

```
3.  Slice memkey = key.memtable_key();
4.  Table::Iterator iter(&table_);
5.  iter.Seek(memkey.data()); // seek 到 value>= memkey.data()的第一个记录
6.   if (iter.Valid()) {
7.     // 这里不需要再检查 sequence number 了，因为 Seek()已经跳过了所有
8.     // 值更大的 sequence number 了
9.     // entry format is:
10.    //    klength  varint32
11.    //    userkey  char[klength]
12.    //    tag      uint64
13.    //    vlength  varint32
14.    //    value    char[vlength]
15.    // Check that it belongs to same user key.  We do not check the
16.    // sequence number since the Seek() call above should have skipped
17.    // all entries with overly large sequence numbers.
18.    const char* entry = iter.key();
19.    uint32_t key_length;
20.    const char* key_ptr = GetVarint32Ptr(entry, entry+5, &key_length);
21.    // 比较 user key 是否相同，key_ptr 开始的 len(internal key) -8 byte 是
    user key
22.    if (comparator_.comparator.user_comparator()->Compare(
23.        Slice(key_ptr, key_length - 8),
24.        key.user_key()) == 0) {
25.       // len(internal key)的后 8byte 是 |sequence number | value type|
26.       const uint64_t tag = DecodeFixed64(key_ptr + key_length - 8);
27.       switch (static_cast<ValueType>(tag & 0xff)) {
28.       case kTypeValue: {   // 只取出 value
29.         Slice v = GetLengthPrefixedSlice(key_ptr + key_length);
30.         value->assign(v.data(), v.size());
31.         return true;
32.       }
33.       case kTypeDeletion:  // 删除
34.         *s = Status::NotFound(Slice());
35.         return true;
```

```
36.     }
37.   }
38. }  //end if
39. return false;
40. }//end Get
```

这段代码，主要就是一个 Seek 函数，根据传入的 LookupKey 得到在 memtable 中存储的 key(memtable_key())，然后调用 Skip list::Iterator 的 Seek 函数查找。Seek 直接调用 Skip list 的 FindGreaterOrEqual(key)接口，返回大于等于 key 的 Iterator。然后取出 user key 判断是否和传入的 user key 相同，①如果相同则取出 value，判断 InteralKey 的 Value Type 为 kTypeDeletion，则 *s=Status::NotFound(Slice())，返回 true；②如果不相同，则返回 false。

4.7 小结

Memtable 到此就分析完毕了，本质上就是一个有序的 Skip list，排序基于 user key 的 sequence number，其排序比较依据依次是：

>1 首先根据 user key 按升序排列

>2 然后根据 sequence number 按降序排列

>3 最后根据 value type 按降序排列（这个其实无关紧要）

5 操作 Log

分析完 KV 在内存中的存储，接下来就是操作日志。所有的写操作都必须先成功的 append 到操作日志中，然后再更新内存 memtable。这样做有两个有点：1 可以将随机的写 IO 变成 append，极大的提高写磁盘速度；2 防止在节点 down 机导致内存数据丢失，造成数据丢失，这对系统来说是个灾难。 在各种高效的存储系统中，这已经是口水技术了。

5.1 格式

在源码下的文档 doc/log_format.txt 中，作者详细描述了 log 格式：

The log file contents are a sequence of 32KB blocks. The only exception is that the tail of the file may contain a partial block.

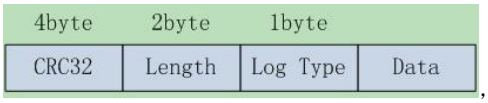
Each block consists of a sequence of records:

```
block:= record* trailer?
record :=
checksum: uint32     // crc32c of type and data[] ; little-endian
length: uint16       // little-endian
type: uint8          // One of FULL,FIRST, MIDDLE, LAST
data: uint8[length]
```

A record never starts within the last six bytes of a block (since it won't fit). Any leftover bytes here form thetrailer, which must consist entirely of zero bytes and must be skipped byreaders.

翻译过来就是：

Leveldb 把日志文件切分成了大小为 32KB 的连续 block 块，block 由连续的 log record 组成，log record 的格式为：



，注意：CRC32, Length 都是 little-endian 的。

Log Type 有 4 种：FULL = 1、FIRST = 2、MIDDLE = 3、LAST = 4。FULL 类型表明该 log record 包含了完整的 user record；而 user record 可能内容很多，超过了 block 的可用大小，就需要分成几条 log record，第一条类型为 FIRST，中间的为 MIDDLE，最后一条为 LAST。也就是：

> FULL，说明该 log record 包含一个完整的 user record；

> FIRST，说明是 user record 的第一条 log record

> MIDDLE，说明是 user record 中间的 log record

> LAST，说明是 user record 最后的一条 log record

文档中的例子，考虑到如下序列的 user records：

A: length 1000

B: length 97270

C: length 8000

A 作为 FULL 类型的 record 存储在第一个 block 中；B 将被拆分成 3 条 log record，分别存储在第 1、2、3 个 block 中，这时 block3 还剩 6byte，将被填充为 0；C 将作为 FULL 类型的 record 存储在 block 4 中。如图 5.1-1 所示。

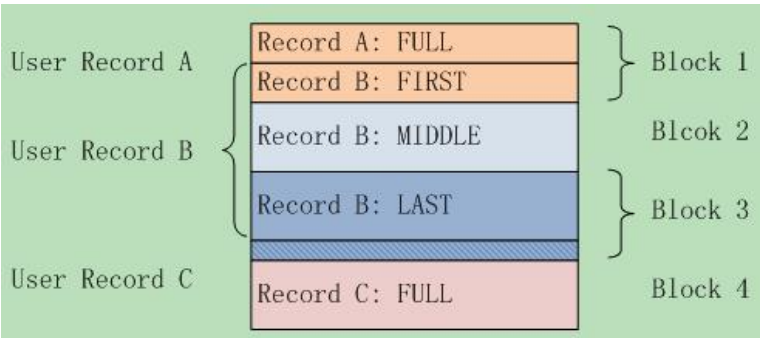


图 5.1-1

由于一条 logrecord 长度最短为 7，如果一个 block 的剩余空间<=6byte，那么将被填充为空字符串，另外长度为 7 的 log record 是不包括任何用户数据的。

Aside: if exactly seven bytes are left in the current block, and a new non-zero length record is added, the writer must emit a FIRST record (which contains zero bytes of user data) to fill up the trailing seven bytes of the block and then emit all of the user data in subsequent blocks.

“另外，如果当前块正好只剩下 7B 了，那么新添加一个长度不为零的 user record 时，剩下的 7B 将被填充为 head，类型为 FIRST，但是 user record 的数据仍然要被传输到下一个块中去。”

Some benefits over the recordio format:

(1) We do not need any heuristics for resyncing - just go to next block boundary and scan. If there is a corruption, skip to the next block. As a side-benefit, we do not get confused when part of the contents of one log file are embedded as a record inside another log file.

(2) Splitting at approximate boundaries (e.g., for mapreduce) is simple: find the next block boundary and skip records until we hit a FULL or FIRST record.

(3) We do not need extra buffering for large records.

Some downsides compared to recordio format:

(1) No packing of tiny records. This could be fixed by adding a new record type, so it is a shortcoming of the current implementation, not necessarily the format.

(2) No compression. Again, this could be fixed by adding new record types.

5.2 写日志

“Log_writer.cc Log_ writer.h”

写比读简单，而且写入决定了读，所以从写开始分析。

有意思的是在写文件时，Leveldb 使用了内存映射文件，内存映射文件的读写效率比普通文件要高，关于内存映射文件为何更高效，这篇文章写的不错：

http://blog.csdn.net/mg0832058/article/details/5890688

其中涉及到的类层次比较简单，如图 5.2-1.

图 5.2-1

注意 Writer 类的成员 type_crc_数组，这里存放的为 Record Type 预先计算的 CRC32 值，因为 Record Type 是固定的几种，为了效率。

Writer 类只有一个接口，就是 AddRecord()，传入 Slice 参数，下面来看函数实现。

首先取出 slice 的字符串指针和长度，初始化 begin=true，表明是第一条 log record。

```
const char* ptr = slice.data();
size_t left = slice.size();
bool begin = true;
```

然后进入一个 do{}while 循环，直到写入出错，或者成功写入全部数据，如下：

- S1 首先查看当前 block 是否<7，如果<7 则补位，并重置 block 偏移。

```
dest_->Append(Slice("\x00\x00\x00\x00\x00\x00",leftover));
block_offset_ = 0;
```

- S2 计算 block 剩余大小，以及本次 log record 可写入数据长度

```
const size_t avail =kBlockSize - block_offset_ - kHeaderSize;
const size_t fragment_length = (left <avail) ? left : avail;
```

- S3 根据两个值，判断 log type

```
RecordType type;
const bool end = (left ==fragment_length); // 两者相等，表明写完
if (begin && end)   type = kFullType;
else if (begin)      type = kFirstType;
```

```
else if (end)        type = kLastType;
else                 type = kMiddleType;
```

- S4 调用 EmitPhysicalRecord 函数，append 日志；并更新指针、剩余长度和 begin 标记。

```
s = EmitPhysicalRecord(type, ptr,fragment_length);
ptr += fragment_length;
left -= fragment_length;
begin = false;
```

接下来看看 EmitPhysicalRecord 函数，这是实际写入的地方，涉及到 log 的存储格式。函数声明为：StatusWriter::EmitPhysicalRecord(RecordType t, const char* ptr, size_t n)

参数 ptr 为用户 record 数据，参数 n 为 record 长度，不包含 log header。

- S1 计算 header，并 Append 到 log 文件，共 7byte 格式为：

| CRC32 (4 byte) | payload length lower + high (2 byte) | type (1byte)|

```
char buf[kHeaderSize];
  buf[4] = static_cast<char>(n& 0xff);
  buf[5] =static_cast<char>(n >> 8);
  buf[6] =static_cast<char>(t);
 // 计算 record type 和 payload 的 CRC 校验值
  uint32_t crc = crc32c::Extend(type_crc_[t], ptr, n);
  crc = crc32c::Mask(crc);      // 空间调整
  EncodeFixed32(buf, crc);
  dest_->Append(Slice(buf,kHeaderSize));
```

- S2 写入 payload，并 Flush，更新 block 的当前偏移

```
s =dest_->Append(Slice(ptr, n));
s = dest_->Flush();
block_offset_ += kHeaderSize + n;
```

以上就是写日志的逻辑，很直观。

5.3 读日志

“Log_reader.cc Log_reader.h Log_format.h Log_format.txt”

日志读取显然比写入要复杂，要检查 checksum，检查是否有损坏等等，处理各种错误。

5.3.1 类层次

先来看看读取涉及到的类图，如图 5.3-1。

class Reader 主要用到了两个接口：

一个是汇报错误的 class Reporter，

另一个是 log 文件读取类 class SequentialFile。

> Reporter 的接口只有一个：void **Corruption**(size_t bytes,const Status& status);

> SequentialFile 有两个接口：

```
Status Read(size_t n, Slice* result, char* scratch);
Status Skip(uint64_t n);
```

说明下，Read 接口有一个*result 参数传递结果就行了，为何还有一个*scratch 呢，这个就和 Slice 相关了。它的字符串指针是传入的外部 char*指针，自己并不负责内存的管理与分配。因此 Read 接口需要调用者提供一个字符串指针，实际存放字符串的地方。

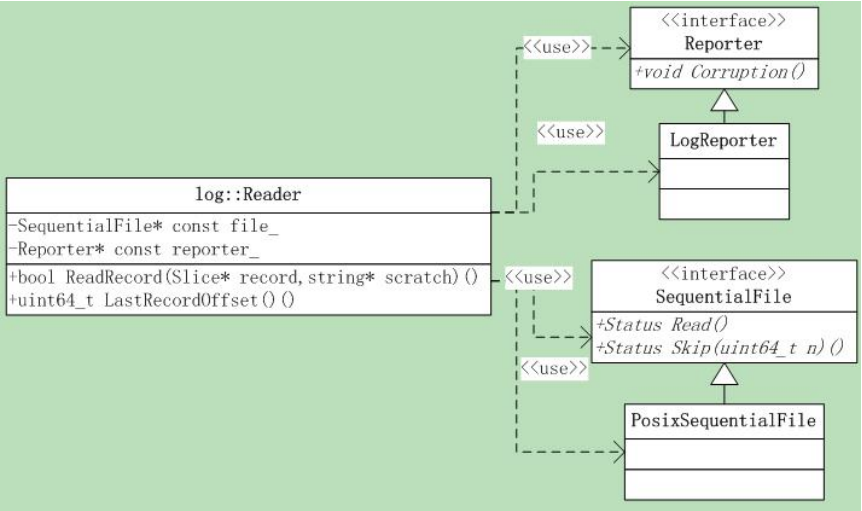


图 5.3-1

Reader 类有几个成员变量，需要注意：

bool eof_； // 上次 Read()返回长度< kBlockSize，暗示到了文件结尾 EOF // Last Read() indicated EOF by returning < kBlockSize;因为 Read()每次读一个块（kBlockSize）大小。

uint64_t last_record_offset_; // 函数 ReadRecord()返回的上一个 record 的偏移: Offset of the last record returned by ReadRecord.

uint64_t end_of_buffer_offset_; // Offset of the first location past the end of buffer_ -- 当前的读取偏移

uint64_t const initial_offset_; // 偏移，从哪里开始读取第一条 record

Slice buffer_; // 读取的内容

5.3.2 日志读取流程

Reader 只有一个接口，那就是 **ReadRecord**，下面来分析下这个函数。

```
// Read the next record into *record.  Returns true if read
// successfully, false if we hit end of the input.  May use
// "*scratch" as temporary storage.  The contents filled in *record
// will only be valid until the next mutating operation on this
// reader or the next mutation to *scratch.
bool ReadRecord(Slice* record, std::string* scratch);
```

- S1 根据 initial_offset_ 跳转到调用者指定的位置，开始读取日志文件。跳转就是直接调用 SequentialFile 的 Seek 接口。

另外，需要先调整调用者传入的 initial_offset_参数，调整和跳转逻辑在 SkipToInitialBlock 函数中。

```
if (last_record_offset_ <initial_offset_) { // 当前偏移 < 指定的偏移，需要 Seek
  if (!SkipToInitialBlock()) return false;
}
```

下面的代码是 **SkipToInitialBlock()**函数调整 read offset 的逻辑：

```
// 计算在 block 内的偏移位置，并调整到开始读取 block 的起始位置
size_t offset_in_block =initial_offset_ % kBlockSize;
uint64_t block_start_location =initial_offset_ - offset_in_block;
```

```
// 如果偏移在最后的 6byte 里，肯定不是一条完整的记录，跳到下一个 block
if (offset_in_block >kBlockSize - 6) {
 offset_in_block = 0;
 block_start_location +=kBlockSize;
}
end_of_buffer_offset_ =block_start_location; // 设置读取偏移
if (block_start_location > 0)  file_->Skip(block_start_location); // 跳转
```

首先计算出 initial_offset_在 block 内的偏移位置(offset_in_block)，然后调整(Skip)到要读取 block 的起始位置(block_start_location)。开始读取日志的时候都要保证读取的是完整的 block，这就是调整的目的。

同时成员变量 end_of_buffer_offset_记录了这个值(要读取 block 的起始位置)，在后续读取中会用到。

- S2 在开始 while 循环前首先初始化几个标记：

```
// 当前是否在 fragment 内，也就是遇到了 FIRST 类型的 record
bool in_fragmented_record = false;
uint64_t prospective_record_offset = 0; // 我们正在读取的逻辑 record 的偏移
```

- S3 进入到 while(true)循环，直到读取到 KLastType 或者 KFullType 的 record，或者到了文件结尾。从日志文件读取完整的 record 是 ReadPhysicalRecord 函数完成的。

读取出现错误时，并不会退出循环，而是汇报错误，继续执行，直到成功读取一条 user record，或者遇到文件结尾。

  - S3.1 从文件读取 record

```
uint64_t physical_record_offset = end_of_buffer_offset_ -buffer_.size();
const unsigned int record_type = ReadPhysicalRecord(&fragment);
```

physical_record_offset 存储的是当前正在读取的 record 的偏移值。接下来根据不同的 record_type 类型，分别处理，一共有 7 种情况：

  - S3.2 FULL type(kFullType)，表明是一条完整的 log record，成功返回读取的 user record 数据。另外需要对早期版本做些 work around，早期的 Leveldb 会在 block 的结尾生产一条空的 kFirstType log record。

```
if (in_fragmented_record) {
 if (scratch->empty())in_fragmented_record = false;
 else ReportCorruption(scratch->size(),"partial record without end(1)");
}
prospective_record_offset= physical_record_offset;
scratch->clear(); // 清空 scratch，读取成功不需要返回 scratch 数据
*record = fragment;
last_record_offset_ =prospective_record_offset; // 更新 last record offset
return true;
```

  - S3.3 FIRST type(kFirstType)，表明是一系列 logrecord(fragment)的第一个 record。同样需要对早期版本做 work around。

把数据读取到 scratch 中，直到成功读取了 LAST 类型的 log record，才把数据返回到 result 中，继续下次的读取循环。

如果再次遇到 FIRST or FULL 类型的 log record，如果 scratch 不为空，就说明日志文件有错误。

```
if (in_fragmented_record) {
```

```
 if (scratch->empty())in_fragmented_record = false;
 else ReportCorruption(scratch->size(),"partial record without end(2)");
}
 prospective_record_offset =physical_record_offset;
 scratch->assign(fragment.data(), fragment.size()); //赋值给 scratch
in_fragmented_record =true; // 设置 fragment 标记为 true
```

- S3.4 MIDDLE type(kMiddleType)，这个处理很简单，如果不是在 fragment 中，报告错误，否则直接 append 到 scratch 中就可以了。

```
 if (!in_fragmented_record){
   ReportCorruption(fragment.size(),"missing start of fragmentedrecord(1)");
}else {scratch->append(fragment.data(),fragment.size());}
```

- S3.5 LAST type(kLastType)，说明是一系列 log record(fragment)中的最后一条。如果不在 fragment 中，报告错误。

```
if (!in_fragmented_record) {
        ReportCorruption(fragment.size(),"missing start of fragmentedrecord(2)");
} else {
        scratch->append(fragment.data(), fragment.size());
        *record = Slice(*scratch);
        last_record_offset_ =prospective_record_offset;
         return true;
}
```

至此，4 种正常的 log record type 已经处理完成，下面 3 种情况是其它的错误处理，类型声明在 Logger 类中：

```
 enum {
   kEof = kMaxRecordType + 1, // 遇到文件结尾
   // 非法的 record，当前有 3 中情况会返回 bad record：
   // * CRC 校验失败 (ReadPhysicalRecord reports adrop)
   // * 长度为 0 (No drop is reported)
   // * 在指定的 initial_offset 之外 (No drop is reported)
  kBadRecord = kMaxRecordType +2
 };
```

- S3.6 遇到文件结尾 kEof，返回 false。不返回任何结果。

```
  if (in_fragmented_record) {
       ReportCorruption(scratch->size(), "partial record withoutend(3)");
       scratch->clear();
   }
   return false;
```

- S3.7 非法的 record(kBadRecord)，如果在 fragment 中，则报告错误。

```
   if (in_fragmented_record){
      ReportCorruption(scratch->size(), "error in middle ofrecord");
      in_fragmented_record = false;
       scratch->clear();
   }
```

- S3.8 缺省分支，遇到非法的 record 类型，报告错误，清空 scratch。

```
ReportCorruption(..., "unknownrecord type %u", record_type);
in_fragmented_record = false; // 重置 fragment 标记
scratch->clear();// 清空 scratch
```

上面就是 ReadRecord 的全部逻辑，解释起来还有些费力。

5.3.3 从 log 文件读取 record

就是前面讲过的 **ReadPhysicalRecord** 函数，它调用 SequentialFile 的 **Read** 接口，从文件读取数据。

该函数开始就进入了一个 while(true)循环，其目的是为了读取到一个完整的 record。读取的内容存放在成员变量 buffer_中。这样的逻辑有些奇怪，实际上，完全不需要一个 while(true)循环的。

函数基本逻辑如下：

- S1 如果 buffer_小于 record header 的大小 kHeaderSize(==7)，进入如下的几个分支：
  - S1.1 如果 eof_为 false，表明还没有到文件结尾，清空 buffer，并读取数据。

```
buffer_.clear(); // 因为上次肯定读取了一个完整的 record
Status status =file_->Read(kBlockSize, &buffer_, backing_store_);
end_of_buffer_offset_ +=buffer_.size(); // 更新 buffer 读取偏移值
 if (!status.ok()) { // 读取失败，设置 eof_为 true，报告错误并返回 kEof
    buffer_.clear();
    ReportDrop(kBlockSize,status);
    eof_ = true;
    return kEof;
} else if (buffer_.size()< kBlockSize){
    eof_ = true; // 实际读取字节<指定(Block Size)，表明到了文件结尾
}
continue; // 继续下次循环
```

  - S1.2 如果 eof_为 true 并且 buffer 为空，表明已经到了文件结尾，正常结束，返回 kEof。
  - S1.3 否则，也就是 eof_为 true，buffer 不为空，说明文件结尾包含了一个不完整的 record，报告错误，返回 kEof。

```
size_t drop_size =buffer_.size();
buffer_.clear();
ReportCorruption(drop_size,"truncated record at end of file");
return kEof;
```

- S2 进入到这里表明上次循环中的 Read 读取到了一个完整的 log record，continue 后的第二次循环判断 buffer_.size() >= kHeaderSize 将执行到此处。

  解析出 log record 的 header 部分，判断长度是否一致。

  根据 log 的格式，前 4byte 是 crc32。后面就是 length 和 type，解析如下：

```
const char* header = buffer_.data();
const uint32_t length = ((header[4])& 0xff) | ((header[5]&0xff)<<8)
const uint32_t type = header[6];
if (kHeaderSize + length >buffer_.size()) { // 长度超出了，汇报错误
 size_t drop_size =buffer_.size();
 buffer_.clear();
 ReportCorruption(drop_size,"bad record length");
 return kBadRecord; // 返回 kBadRecord
```

```
}
if (type == kZeroType&& length == 0) { // 对于 Zero Type 类型，不汇报错误
 buffer_.clear();
 return kBadRecord; // 依然返回 kBadRecord
}
```

- S3 校验 CRC32，如果校验出错，则汇报错误，并返回 kBadRecord。
- S4 如果 record 的开始位置在 initial offset 之前，则跳过，并返回 kBadRecord，否则返回 record 数据和 type。

```
buffer_.remove_prefix(kHeaderSize+ length);
if (end_of_buffer_offset_ -buffer_.size() - kHeaderSize - length < initial_offset_) {
 result->clear();
 return kBadRecord;
}
*result = Slice(header +kHeaderSize, length);
return type;
```

从 log 文件读取 record 的逻辑就是这样的。至此，读日志的逻辑也完成了。接下来将进入磁盘存储的 sstable 部分。

6 SSTable

SSTable 是 Leveldb 的核心之一，是表数据最终在磁盘上的物理存储。也是体量比较大的模块。

6.1 SSTable 的文件组织

作者在文档 doc/table_format.txt 中描述了表的逻辑结构，如图 6.1-1 所示。逻辑上可分为两大块，数据存储区 Data Block，以及各种 Meta 信息。

1）文件中的 k/v 对是有序存储的，并且被划分到连续排列的 Data Block 里面，这些 Data Block 从文件头开始顺序存储，Data Block 的存储格式代码在 block_builder.cc 中；

2）紧跟在 Data Block 之后的是 Meta Block，其格式代码也在 block_builder.cc 中；Meta Block 存储的是 Filter 信息，比如 Bloom 过滤器(bloom filter)，用于快速定位 key 是否在（不在）data block 中。

3）MetaIndex Block 是对 Meta Block 的索引，它只有一条记录，key 是 meta index 的名字（也就是 Filter 的名字），value 为指向 meta block 的 BlockHandle；

BlockHandle 是一个 class，成员 uint64_t offset_是 Block 在文件中的偏移，成员 uint64_t size_是 block 的大小；

4）Index block 是对 Data Block 的索引，对于其中的每个记录，其 key >=Data Block 最后一条记录的 key，同时<其后 Data Block 的第一条记录的 key；value 是指向 data block 的 BlockHandle；

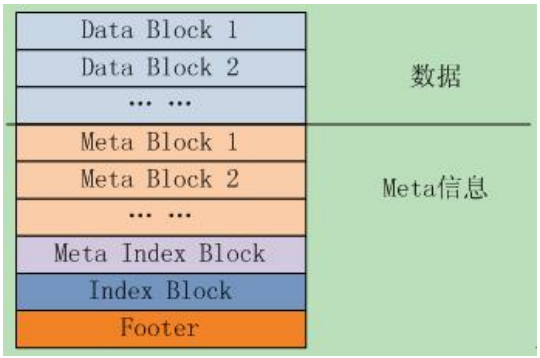


图 6.1-1

5）Footer，文件的最后，大小固定，其格式如图 6.1-2 所示。

图 6.1-2

成员 metaindex_handle 指出了 meta index block 的起始位置和大小；成员 index_handle 指出了 index block 的起始地址和大小；这两个字段都是 BlockHandle 对象，可以理解为索引的索引，通过 Footer 可以直接定位到 metaindex 和 index block。再后面是一个填充区和魔数（0xdb4775248b80fb57）。

6.2 Block 存储格式

6.2.1 Block 的逻辑存储

Data Block 是具体的 k/v 数据对存储区域，此外还有存储 meta 的 metaIndex Block，存储 data block 索引信息的 Index Block 等等，他们都是以 Block 的方式存储的。来看看 Block 是如何组织的。每个 Block 有三部分构成：block data, type, crc32，如图 6.2-1 所示。

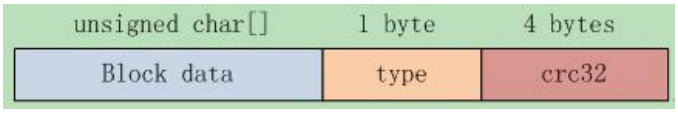


图 6.2-1

类型 type 指明使用的是哪种压缩方式，当前支持 none 和 snappy 压缩。

虽然 block 有好几种，但是 Block Data 都是有序的 k/v 对，因此写入、读取 BlockData 的接口都是统一的，对于 Block Data 的管理也都是相同的。

对 Block 的写入、读取将在创建、读取 sstable 时分析，知道了格式之后，其读取写入代码都是很直观的。

由于 sstable 对数据的存储格式都是 Block，因此在分析 sstable 的读取和写入逻辑之前，我们先来分析下 Leveldb 对 Block Data 的管理。

Leveldb 对 Block Data 的管理是读写分离的，读取后的遍历查询操作由 Block 类实现，BlockData 的构建则由 BlockBuilder 类实现。

6.2.2 重启点-restartpoint

BlockBuilder 对 key 的存储是前缀压缩的，对于有序的字符串来讲，这能极大的减少存储空间。但是却增加了查找的时间复杂度，为了兼顾查找效率，每隔 K 个 key，leveldb 就不使用前缀压缩，而是存储整个 key，这就是重启点（restartpoint）。

在构建 Block 时，有参数 Options::block_restart_interval 定每隔几个 key(block_restart_interval)就直接存储一个重启点 key。

Block 在结尾记录所有重启点的偏移，可以二分查找指定的 key。Value 直接存储在 key 的后面，无压缩。

对于一个 k/v 对，其在 block 中的存储格式为：

> 共享前缀长度　　shared_bytes:　varint32
> 前缀之后的字符串长度　unshared_bytes:　varint32
> 值的长度　　value_length:　varint32
> 前缀之后的字符串　key_delta:　char[unshared_bytes]
> 值　　value:　char[value_length]

对于重启点，shared_bytes= 0

Block 的结尾段格式是：

> restarts:　　uint32[num_restarts]
> num_restarts:　uint32 // 重启点个数

元素 restarts[i]存储的是 block 的第 i 个重启点的偏移。很明显第一个 k/v 对，总是第一个重启点，也就是 restarts[0] = 0;

图 6.2-2 给出了 block 的存储示意图。

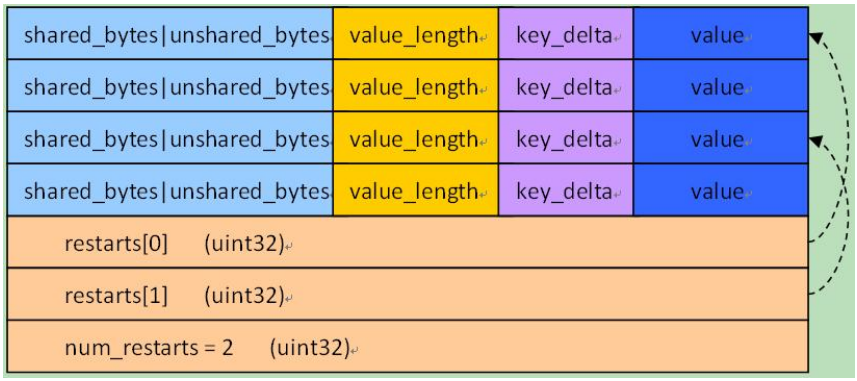


图 6.2-2

总体来看 Block 可分为 k/v 存储区和后面的重启点存储区两部分，其中 k/v 的存储格式如前面所讲，可看做 4 部分：

前缀压缩的 key 长度信息 + value 长度 + key 前缀之后的字符串+ value

最后一个 4byte 为重启点的个数。

对 Block 的存储格式了解之后，对 Block 的构建和读取的代码分析就是很直观的事情了。见下面的分析。

6.3 Block 的构建与读取

6.3.1 class BlockBuilder

首先从 Block 的构建开始，这就是 BlockBuilder 类，来看下 BlockBuilder 的函数接口，一共有 5 个：

```
1.  void Reset(); // 重设内容，通常在 Finish 之后调用已构建新的 block
2.  //添加 k/v，要求：Reset()之后没有调用过 Finish()；Key > 任何已加入的 key
3.  void Add(const Slice& key,const Slice& value);
4.  // 结束构建 block，并返回指向 block 内容的指针
5.  Slice Finish();// 返回 Slice 的生存周期：Builder 的生存周期，or 直到 Reset()被调
    用
6.  size_t CurrentSizeEstimate()const; // 返回正在构建 block 的未压缩大小–估计值
7.  bool empty() const { return buffer_.empty();} // 没有 entry 则返回 true
```

主要成员变量如下：

```
1.  std::string            buffer_; // block 的内容
2.  std::vector<uint32_t>  restarts_;  // 重启点-后面会分析到
3.  int                  counter_;  // 重启后生成的 entry 数
4.  std::string            last_key_; // 记录最后添加的 key
```

6.3.2 BlockBuilder::Add()

调用 Add 函数向当前 Block 中新加入一个 k/v 对{key, value}。函数处理逻辑如下：

- S1 保证新加入的 key > 已加入的任何一个 key；

```
assert(!finished_);
assert(counter_ <= options_->block_restart_interval);
assert(buffer_.empty() || options_->comparator->Compare(key,last_key_piece)
> 0);
```

- S2 如果计数器 counter < opions->block_restart_interval，则使用前缀算法压缩 key，否则就把 key 作为一个重启点，无压缩存储；

```
Slice last_key_piece(last_key_);
if (counter_ < options_->block_restart_interval) { //前缀压缩
    // 计算 key 与 last_key_的公共前缀
    const size_t min_length= std::min(last_key_piece.size(), key.size());
    while ((shared < min_length)&& (last_key_piece[shared] == key[shared]))
    {
        shared++;
    }else{ // 新的重启点
        restarts_.push_back(buffer_.size());
        counter_ = 0;
}
```

- S3 根据上面的数据格式存储 k/v 对，追加到 buffer 中，并更新 block 状态。

```
const size_t non_shared = key.size() - shared; // key 前缀之后的字符串长度
// append"<shared><non_shared><value_size>" 到 buffer_
PutVarint32(&buffer_, shared);
PutVarint32(&buffer_, non_shared);
PutVarint32(&buffer_, value.size());
// 其后是前缀之后的字符串 + value
buffer_.append(key.data() + shared, non_shared);
buffer_.append(value.data(), value.size());
// 更新状态 ，last_key_ = key 及计数器 counter_
last_key_.resize(shared);   // 连一个 string 的赋值都要照顾到，使内存 copy 最小化
last_key_.append(key.data() + shared, non_shared);
assert(Slice(last_key_) == key);
counter_++;
```

6.3.3 BlockBuilder::Finish()

调用该函数完成 Block 的构建，很简单，压入重启点信息，并返回 buffer_，设置结束标记 finished_:

```
// Finish building the block and return a slice that refers to the

// block contents.  The returned slice will remain valid for the

// lifetime of this builder or until Reset() is called.
```

```
for (size_t i = 0; i < restarts_.size(); i++) {  // 重启点
      PutFixed32(&buffer_, restarts_[i]);
}
PutFixed32(&buffer_, restarts_.size());    // 重启点数量
finished_ = true;
return Slice(buffer_);
```

6.3.4 BlockBuilder::Reset() & 大小

还有 Reset 和 CurrentSizeEstimate 两个函数，Reset 复位函数，清空各个信息；函数 CurrentSizeEstimate 返回 block 的预计大小，从函数实现来看，应该在调用 Finish 之前调用该函数。

```
void BlockBuilder::Reset() {
   buffer_.clear();  restarts_.clear();  last_key_.clear();
   restarts_.push_back(0);       // 第一个重启点位置总是 0
   counter_ = 0;
   finished_ = false;
}

size_t BlockBuilder::CurrentSizeEstimate () const {
   // buffer 大小 +重启点数组长度 + 重启点长度(uint32)
  return (buffer_.size() +  restarts_.size() * sizeof(uint32_t) + sizeof(uint32_t));
}
```

Block 的构建就这些内容了，下面开始分析 Block 的读取，就是类 Block。

6.3.5 class Block

对 Block 的读取是由类 Block 完成的，先来看看其函数接口和关键成员变量。

Block 只有两个函数接口，通过 Iterator 对象，调用者就可以遍历访问 Block 的存储的 k/v 对了；以及几个成员变量，如下：

```
1.    size_t size() const { return size_; }
2.    Iterator* NewIterator(constComparator* comparator);
3.
4.    const char* data_; // block 数据指针
5.    size_t size_;      // block 数据大小
6.    uint32_t restart_offset_;     // 重启点数组在 data_中的偏移
7.    bool owned_;              //data_[]是否是 Block 拥有的
```

6.3.6 Block 初始化

Block 的构造函数接受一个 BlockContents 对象 contents 初始化，BlockContents 是一个有 3 个成员的结构体。

```
1.    >data = Slice();
2.    >cachable = false; // 无 cache
3.    >heap_allocated = false; // 非 heap 分配
4.  根据 contents 为成员赋值
5.  data_ = contents.data.data();
6.  size_ =contents.data.size();
7.  owned_ = contents.heap_allocated;
```

然后从 data 中解析出重启点数组，如果数据太小，或者重启点计算出错，就设置 size_=0，表明该 block data 解析失败.

```
1.  if (size_ < sizeof(uint32_t)){
2.    size_ = 0;  // 出错了
3.  } else {
4.    restart_offset_ = size_ - (1 +NumRestarts()) * sizeof(uint32_t);
5.    if (restart_offset_ > size_- sizeof(uint32_t))
6.       size_ = 0;
7.  }
8.
```

```
1.  新版代码：
2.  Block::Block(const BlockContents& contents)
3.      : data_(contents.data.data()),
4.        size_(contents.data.size()),
5.        owned_(contents.heap_allocated) {
6.    if (size_ < sizeof(uint32_t)) {
7.      size_ = 0;  // Error marker
```

```
9.    } else {

8.      size_t max_restarts_allowed = (size_-sizeof(uint32_t)) /
    sizeof(uint32_t);
9.      if (NumRestarts() > max_restarts_allowed) {
10.       // The size is too small for NumRestarts()
11.       size_ = 0;
12.     } else {
13.       restart_offset_ = size_ - (1 + NumRestarts()) * sizeof(uint32_t);
14.     }
15.   }
16. }
```

NumRestarts()函数就是从最后的 uint32 解析出重启点的个数，并返回:

return DecodeFixed32(data_ +size_ - sizeof(uint32_t))

6. 3. 7 Block::Iter

这是一个用以遍历 Block 内部数据的内部类，它继承了 Iterator 接口。函数 NewIterator 返回 Block::Iter 对象：return new Iter(cmp, data_,restart_offset_, num_restarts);

下面我们就分析 **Iter 的实现**。

主要成员变量有：

```
1.  const Comparator* constcomparator_; // key 比较器
2.  const char* const data_;      // block 内容
3.  uint32_t const restarts_;     // 重启点(uint32 数组)在 data 中的偏移
4.  uint32_t const num_restarts_; // 重启点个数
5.  uint32_t current_; // 当前 entry 在 data 中的偏移.  >= restarts_表明非法
6.  uint32_t restart_index_;  // current_所在的重启点的 index
```

下面来看看对 Iterator 接口的实现，简单函数略过。

>首先是 **Next**()函数，直接调用 private 函数 **ParseNextKey**()跳到下一个 k/v 对
> **ParseNextKey** 函数实现如下：

- S1 跳到下一个 entry，其位置紧邻在当前 value_之后。如果已经是最后一个 entry 了，返回 false，标记 current_为 invalid。

```
current_ = NextEntryOffset(); // (value_.data() + value_.size()) - data_
const char* p = data_ +current_;
const char* limit = data_ +restarts_; // Restarts come right after data
if (p >= limit) { // entry 到头了，标记为 invalid.
  current_ = restarts_;
  restart_index_ =num_restarts_;
  return false;
}
```

- S2 解析出 entry，解析出错则设置错误状态，记录错误并返回 false。解析成功则根据信息组成 key_和 value_，并更新重启点 index。

```
uint32_t shared, non_shared,value_length;
p = DecodeEntry(p, limit, &shared, &non_shared, &value_length);
if (p == NULL || key_.size()< shared) {
  CorruptionError();
  return false;
} else { // 成功
  key_.resize(shared);
  key_.append(p, non_shared);
  value_ = Slice(p +non_shared, value_length);
  while (restart_index_ + 1< num_restarts_ && GetRestartPoint(restart_index_ + 1) < current_) {
      ++restart_index_;
  }
  return true;
}
```

函数 **DecodeEntry** 从字符串[p, limit)解析出 key 的前缀长度、key 前缀之后的字符串长度和 value 的长度这三个 vint32 值，代码很简单。

函数 **CorruptionError** 将 current_和 restart_index_都设置为 invalid 状态，并在 status 中设置错误状态。

函数 **GetRestartPoint** 从 data 中读取指定 restart index 的偏移值 restart[index]，并返回：DecodeFixed32(data_ + restarts_ +index * sizeof(uint32_t);

- 接下来看看 **Prev** 函数，Previous 操作分为两步：首先回到 current_之前的重启点，然后再向后直到 current_，实现如下：

- S1 首先向前回跳到在 current_前面的那个重启点，并定位到重启点的 k/v 对开始位置。

```
const uint32_t original = current_;
while (GetRestartPoint(restart_index_)>= original) {
        if (restart_index_ == 0) { // 到第一个 entry 了，标记 invalid 状态
            current_ = restarts_;
            restart_index_ =num_restarts_;
            return;
       }
       restart_index_--;
}
SeekToRestartPoint(restart_index_);//根据 restart index 定位到重启点的 k/v 对
```

- S2 第二步，从重启点位置开始向后遍历，直到遇到 original 前面的那个 k/v 对。

do {} while (ParseNextKey() &&NextEntryOffset() < original);

说说上面遇到的 **SeekToRestartPoint()**函数，它只是设置了几个有限的状态，其它值将在函数 ParseNextKey()中设置。感觉这有点 tricky，这里的 value_并不是 k/v 对的 value，而只是一个指向 k/v 对起始位置的 0 长度指针，这样后面的 ParseNextKey 函数将会取出重启点的 k/v 值。

```
void SeekToRestartPoint(uint32_tindex) {
  key_.clear();
  restart_index_ = index;
  // ParseNextKey()会设置 current_;
  //ParseNextKey()从 value_结尾开始，因此需要相应的设置 value_
  uint32_t offset = GetRestartPoint(index);
  value_ = Slice(data_ + offset,0); // value 长度设置为 0，字符串指针是 data_+offset
}
```

> **SeekToFirst/SeekToLast**，这两个函数都很简单，借助于前面的 SeekToResartPoint 函数就可以完成。

```
virtual void SeekToFirst() {
  SeekToRestartPoint(0);
  ParseNextKey();
}

virtual void SeekToLast() {
  SeekToRestartPoint(num_restarts_ - 1);
```

```
8.    while (ParseNextKey()&& NextEntryOffset() < restarts_) {} //Keep skipping

9.  }
```

> 最后一个：**Seek()**函数，跳到指定的 target(Slice)，函数逻辑如下：

- S1 二分查找，找到 key < target 的最后一个重启点，典型的二分查找算法，代码就不再贴了。

- S2 找到后，跳转到重启点，其索引由 left 指定，这是前面二分查找到的结果。如前面所分析的，value_指向重启点的地址，而 size_指定为 0，这样 ParseNextKey 函数将会取出重启点的 k/v 值。

  SeekToRestartPoint(left);

- S3 自重启点线性向下，直到遇到 key>= target 的 k/v 对。

```
1.  while (true) {
2.    if (!ParseNextKey()) return;
3.    if (Compare(key_, target)>= 0) return;
4.  }
```

上面就是 Block::Iter 的全部实现逻辑，这样 Block 的创建和读取遍历都已经分析完毕。

6.4 创建 sstable 文件

了解了 sstable 文件的存储格式，以及 Data Block 的组织，下面就可以分析如何创建 sstable 文件了。相关代码在 table_builder.h/.cc 以及 block_builder.h/.cc（构建 Block）中。

6.4.1 class TableBuilder

构建 sstable 文件的类是 TableBuilder，该类提供了几个有限的方法可以用来添加 k/v 对，Flush 到文件中等等，它依赖于 BlockBuilder 来构建 Block。

TableBuilder 的几个接口说明下：

> void **Add**(const Slice& key, const Slice& value)，向当前正在构建的 sstable 添加新的{key, value}对，要求根据 Option 指定的 Comparator，key 必须位于所有前面添加的 key 之后；

> void **Flush**()，将当前缓存的 k/v 全部 flush 到文件中，一个高级方法，大部分的 client 不需要直接调用该方法；

批注 [g2]: ?

> void **Finish**()，结束表的构建，该方法被调用后，将不再会使用传入的 WritableFile；

> void **Abandon**()，结束表的构建，并丢弃当前缓存的内容，该方法被调用后，将不再会使用传入的 WritableFile；【只是设置 closed 为 true，无其他操作】

一旦 Finish()/Abandon()方法被调用，将不能再次执行 Flush 或者 Add 操作。

下面来看看涉及到的类，如图 6.3-1 所示。

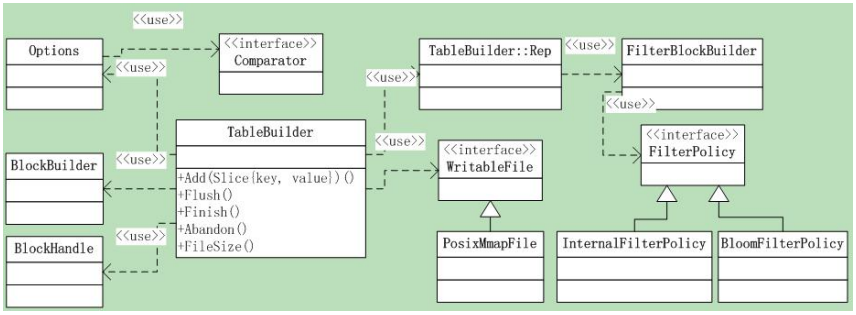


图 6.3-1

其中 WritableFile 和 op log 一样，使用的都是内存映射文件。Options 是一些调用者可设置的选项。

TableBuilder 只有一个成员变量 Rep* rep_，实际上 Rep 结构体的成员就是 TableBuilder 所有的成员变量；这样做的目的，可能是为了隐藏其内部细节。Rep 的定义也是在.cc 文件中，对外是透明的。

简单解释下成员的含义：

```
1.  Options options;   // data block 的选项
2.  Options index_block_options; // index block 的选项
3.  WritableFile* file;  // sstable 文件
4.  uint64_t offset; // 要写入 data block 在 sstable 文件中的偏移，初始 0
5.  Status status; //当前状态-初始 ok
6.  BlockBuilder data_block; //当前操作的 data block
7.  BlockBuilder index_block; // sstable 的 index block
8.  std::string last_key; //当前 data block 最后的 k/v 对的 key
```

```
int64_t num_entries; //当前 data block 的个数，初始 0
bool closed;          //调用了 Finish() or Abandon()，初始 false
FilterBlockBuilder* filter_block; //根据 filter 数据快速定位 key 是否在 block 中
bool pending_index_entry; //见下面的 Add 函数，初始 false
BlockHandle pending_handle; //添加到 index block 的 value（data block 的信息）
std::string compressed_output;//压缩后的 data block，临时存储，写入后即被清空
```

filter_block 是存储的过滤器信息，它会存储{key，对应 data block 在 sstable 的偏移值}，不一定是完全精确的，以快速定位给定 key 是否在 data block 中。

批注 [g3]: ?

下面分析如何向 sstable 中添加 k/v 对，创建并持久化 sstable。其它函数都比较简单，略过。另外对于 Abandon，简单设置 closed=true 即返回。

6.4.2 添加 k/v 对

这是通过方法 **Add**(constSlice& key, const Slice& value)完成的，没有返回值。下面分析该函数的逻辑：

- S1 首先保证文件没有 close，也就是没有调用过 Finish/Abandon，以及保证当前 status 是 ok 的；如果当前有缓存的 kv 对，保证新加入的 key 是最大的。

```
Rep* r = rep_;
assert(!r->closed);
if (!ok()) return;
if (r->num_entries > 0) {
   assert(r->options.comparator->Compare(key, Slice(r->last_key))> 0);
}
```

- S2 如果标记 r->pending_index_entry 为 true，表明遇到下一个 data block 的第一个 k/v，根据 key 调整 r->last_key，这是通过 Comparator 的 FindShortestSeparator 完成的。

```
if (r->pending_index_entry) {
   assert(r->data_block.empty());
   r->options.comparator->FindShortestSeparator(&r->last_key,key);
   std::string handle_encoding;
   r->pending_handle.EncodeTo(&handle_encoding);
   r->index_block.Add(r->last_key, Slice(handle_encoding));
   r->pending_index_entry =false;
}
```

接下来将 pending_handle 加入到 index block 中{r->last_key, r->pending_handle’sstring}。最后将 r->pending_index_entry 设置为 false。

值得讲讲 pending_index_entry 这个标记的意义，见代码注释：

```
// We do not emit the index entry for a block until we have seen the
// first key for the next data block.  This allows us to use shorter
// keys in the index block.  For example, consider a block boundary
// between the keys "the quick brown fox" and "the who".  We can use
// "the r" as the key for the index block entry since it is >= all
// entries in the first block and < all entries in subsequent blocks.
// Invariant: r->pending_index_entry is true only if data_block is empty.
```

直到遇到下一个 databock 的第一个 key 时，我们才为上一个 datablock 生成 index entry，这样的好处是：可以为 index 使用较短的 key；比如上一个 data block 最后一个 k/v 的 key 是"the quick brown fox"，其后继 data block 的第一个 key 是"the who"，我们就可以用一个较短的字符串"the r"作为上一个 data block 的 index block entry 的 key。

简而言之，就是在开始下一个 datablock 时，Leveldb 才将上一个 data block 加入到 index block 中。标记 pending_index_entry 就是干这个用的，对应 data block 的 index entry 信息就保存在（BlockHandle）pending_handle。

- S3 如果 filter_block 不为空，就把 key 加入到 filter_block 中。

```
1.  if (r->filter_block != NULL) {
2.     r->filter_block->AddKey(key);
3.  }
```

- S4 设置 r->last_key = key，将(key, value)添加到 r->data_block 中，并更新 entry 数。

```
1.  r->last_key.assign(key.data(), key.size());
2.  r->num_entries++;
3.  r->data_block.Add(key,value);
```

- S5 如果 data block 的个数超过限制，就立刻 Flush 到文件中。

```
1.  const size_testimated_block_size = r->data_block.CurrentSizeEstimate();
2.  if (estimated_block_size >=r->options.block_size) Flush();
```

6.4.3 Flush 文件

该函数逻辑比较简单，直接见代码如下：

```
1.  Rep* r = rep_;
2.  assert(!r->closed);
3.  if (!ok()) return;  // 首先保证未关闭，且状态 ok
4.  if (r->data_block.empty()) return; // data block 是空的
5.
6.  // 保证 pending_index_entry 为 false，即 data block 的 Add 已经完成
7.  assert(!r->pending_index_entry);
8.
9.  // 写入 data block，并设置其 index entry 信息—BlockHandle 对象
10. WriteBlock(&r->data_block, &r->pending_handle);
11.
12. //写入成功，则 Flush 文件，并设置 r->pending_index_entry 为 true，
13. //以根据下一个 data block 的 first key 调整 index entry 的 key—即 r->last_key
14. if (ok()) {
15.   r->pending_index_entry =true;
16.   r->status =r->file->Flush();
17. }
18. if (r->filter_block != NULL){ //将 data block 在 sstable 中的偏移加入到
    filter block 中
19.   r->filter_block->StartBlock(r->offset); // 并指明开始新的 data block
20. }
```

6.4.4 WriteBlock 函数

在 Flush 文件时，会调用 WriteBlock 函数将 data block 写入到文件中，该函数同时还设置 data block 的 index entry 信息。原型为：

void WriteBlock(BlockBuilder* block, BlockHandle* handle)

该函数做些预处理工作，序列化要写入的 data block，根据需要压缩数据，真正的写入逻辑是在 WriteRawBlock 函数中。下面分析该函数的处理逻辑。

- S1 获得 block 的序列化数据 Slice，根据配置参数决定是否压缩，以及根据压缩格式压缩数据内容。对于 Snappy 压缩，如果压缩率太低<12.5%，还是作为未压缩内容存储。

  BlockBuilder 的 Finish()函数将 data block 的数据序列化成一个 Slice。

```
1.  Rep* r = rep_;
2.  Slice raw = block->Finish(); // 获得 data block 的序列化字符串
3.  Slice block_contents;
4.  CompressionType type =r->options.compression;
5.
6.  switch (type) {
7.    case kNoCompression: block_contents= raw; break; // 不压缩
8.    case kSnappyCompression: { // snappy 压缩格式
9.      std::string* compressed =&r->compressed_output;
10.     if(port::Snappy_Compress(raw.data(), raw.size(), compressed) &&
11.         compressed->size()< raw.size() - (raw.size() / 8u)) {
12.         block_contents =*compressed;
13.     } else { // 如果不支持 Snappy，或者压缩率低于 12.5%，依然当作不压缩存储
14.       block_contents = raw;
15.       type = kNoCompression;
16.     }
17.     break;
18.   }
19. }
```

- S2 将 data 内容写入到文件，并重置 block 成初始化状态，清空 compressedoutput。

[cpp] view plaincopy

```
1.  WriteRawBlock(block_contents,type, handle);
2.  r->compressed_output.clear();
3.  block->Reset();
```

6.4.5 WriteRawBlock 函数

在 WriteBlock 把准备工作都做好后，就可以写入到 sstable 文件中了。来看函数原型：

void WriteRawBlock(const Slice& data, CompressionType, BlockHandle*handle);

函数逻辑很简单，见代码。

```
Rep* r = rep_;
handle->set_offset(r->offset); // 为 index 设置 data block 的 handle 信息
handle->set_size(block_contents.size());

nbsp;r->status =r->file->Append(block_contents); // 写入 data block 内容
if (r->status.ok()) {// 写入 1byte 的 type 和 4bytes 的 crc32
  chartrailer[kBlockTrailerSize];
  trailer[0] = type;
  uint32_t crc = crc32c::Value(block_contents.data(),block_contents.size());

  crc = crc32c::Extend(crc, trailer, 1);  // Extend crc to
cover block type
  EncodeFixed32(trailer+1, crc32c::Mask(crc));
  r->status =r->file->Append(Slice(trailer, kBlockTrailerSize));
  if (r->status.ok()) { // 写入成功更新 offset-下一个 data block 的写入偏移
    r->offset +=block_contents.size() + kBlockTrailerSize;
  }
}
```

6.4.6 Finish 函数

调用 Finish 函数，表明调用者将所有已经添加的 k/v 对持久化到 sstable，并关闭 sstable 文件。

该函数逻辑很清晰，可分为 5 部分。

- S1 首先调用 Flush，写入最后的一块 data block，然后设置关闭标志 closed=true。表明该 sstable 已经关闭，不能再添加 k/v 对。

```
Rep* r = rep_;
Flush();
assert(!r->closed);
r->closed = true;
```

BlockHandle filter_block_handle,metaindex_block_handle, index_block_handle;

- S2 写入 filter block 到文件中

```
if (ok() &&r->filter_block != NULL) {
  WriteRawBlock(r->filter_block->Finish(), kNoCompression,&filter_block_hand
le);
}
```

- S3 写入 meta index block 到文件中

如果 filterblock 不为 NULL，则加入从"filter.Name"到 filter data 位置的映射。通过 meta index block，可以根据 filter 名字快速定位到 filter 的数据区。

```
if (ok()) {
  BlockBuildermeta_index_block(&r->options);
  if (r->filter_block !=NULL) {
    //加入从"filter.Name"到 filter data 位置的映射
    std::string key ="filter.";
    key.append(r->options.filter_policy->Name());
    std::string handle_encoding;
    filter_block_handle.EncodeTo(&handle_encoding);
    meta_index_block.Add(key,handle_encoding);
  }
  // TODO(postrelease): Add stats and other metablocks
  WriteBlock(&meta_index_block, &metaindex_block_handle);
}
```

- S4 写入 index block，如果成功 Flush 过 data block，那么需要为最后一块 data block 设置 index block，并加入到 index block 中。

```
if (ok()) {
  if (r->pending_index_entry){ // Flush 时会被设置为 true
    r->options.comparator->FindShortSuccessor(&r->last_key);
    std::string handle_encoding;
    r->pending_handle.EncodeTo(&handle_encoding);
    r->index_block.Add(r->last_key, Slice(handle_encoding)); // 加入到
  index block 中
    r->pending_index_entry =false;
  }
  WriteBlock(&r->index_block, &index_block_handle);
}
```

- S5 写入 Footer。

```
if (ok()) {
  Footer footer;
  footer.set_metaindex_handle(metaindex_block_handle);
  footer.set_index_handle(index_block_handle);
  std::string footer_encoding;
  footer.EncodeTo(&footer_encoding);

  r->status =r->file->Append(footer_encoding);
```

```
9.    if (r->status.ok()) {
10.     r->offset +=footer_encoding.size();
11.   }
12. }
```

整个写入流程就分析完了，对于 Datablock 和 Filter Block 的操作将在 Data block 和 Filter Block 中单独分析，下面的读取相同。

6.5 读取 sstable 文件

6.5.1 Class Table

Sstable 文件的读取逻辑在类 Table 中，其中涉及到的类还是比较多的，如图 6.5-1 所示。

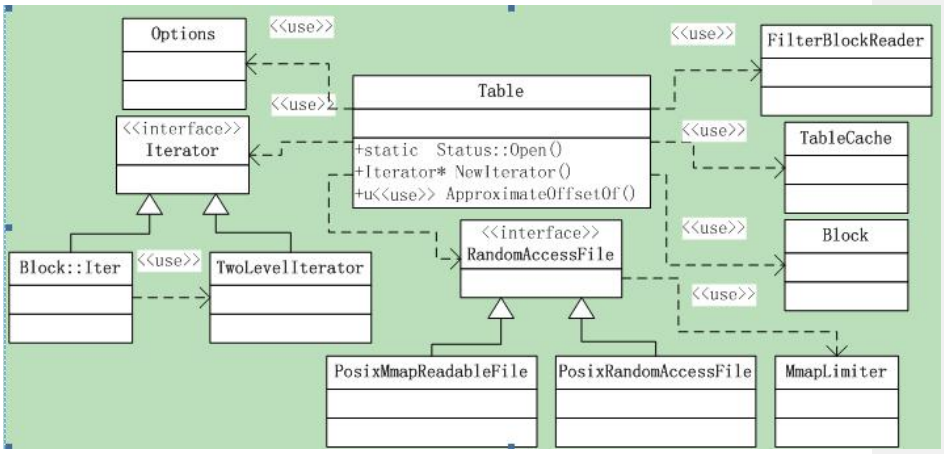


图 6.5-1

Table 类导出的函数只有 3 个，先从这三个导出函数开始分析。其中涉及到的类（包括上图中为画出的）都会一一遇到，然后再一一拆解。

本节分析 sstable 的打开逻辑，后面再分析 key 的查找与数据遍历。

6.5.2 Table::Open()

打开一个 sstable 文件，函数声明为：

static Status **Open**(const Options& options, RandomAccessFile* file, uint64_t file_size, Table** table);

这是 Table 类的一个静态函数，如果操作成功，指针*table 指向新打开的表，否则返回错误。

要打开的文件和大小分别由参数 file 和 file_size 指定；option 是一些选项；

下面就分析下函数逻辑：

- S1 首先从文件的结尾读取 Footer，并 Decode 到 Footer 对象中，如果文件长度小于 Footer 的长度，则报错。Footer 的 decode 很简单，就是根据前面的 Footer 结构，解析并判断 magic number 是否正确，解析出 meta index 和 index block 的偏移和长度。

```
1.  *table = NULL;
2.  if (size <Footer::kEncodedLength) { // 文件太短
3.    returnStatus::InvalidArgument("file is too short to be an sstable");
4.  }
5.
6.  charfooter_space[Footer::kEncodedLength]; // Footer 大小是固定的
7.  Slice footer_input;
8.  Status s = file->Read(size -
    Footer::kEncodedLength, Footer::kEncodedLength,
9.                        &footer_input, footer_space);
10. if (!s.ok()) return s;
11.
12. Footer footer;
13. s =footer.DecodeFrom(&footer_input);
14. if (!s.ok()) return s;
```

- S2 解析出了 Footer，我们就可以读取 index block 和 meta index 了，首先读取 index block。

```
1.  BlockContents contents;
2.  Block* index_block = NULL;
3.  if (s.ok()) {
4.    s = ReadBlock(file, ReadOptions(),footer.index_handle(), &contents);
5.    if (s.ok()) {
6.      index_block = newBlock(contents);
7.    }
8.  }
```

这是通过调用 **ReadBlock()**完成的，下面会分析这个函数。

- S3 已经成功读取了 footer 和 index block，此时 table 已经可以响应请求了。构建 table 对象，并读取 metaindex 数据构建 filter policy。如果 option 打开了 cache，还要为 table 创建 cache。

```
1.  if (s.ok()) {
2.    // 已成功读取 footer 和 index block: 可以响应请求了
3.    Rep* rep = new Table::Rep;
4.    rep->options = options;
5.    rep->file = file;
6.    rep->metaindex_handle =footer.metaindex_handle();
7.    rep->index_block =index_block;
8.    rep->cache_id =(options.block_cache ? options.block_cache->NewId() : 0);
9.    rep->filter_data = rep->filter= NULL;
10.   *table = new Table(rep);
11.   (*table)->ReadMeta(footer);// 调用 ReadMeta 读取 metaindex
12. } else {
13.   if (index_block) deleteindex_block;
14. }
```

到这里，Table 的打开操作就已经为完成了。下面来分析上面用到的 ReadBlock()和 ReadMeta()函数.

6.5.3 ReadBlock()

前面讲过 block 的格式，以及 Block 的写入（TableBuilder::WriteRawBlock），现在我们可以轻松的分析 Block 的读取操作了。

这是一个全局函数(format.cc)，声明为：

Status **ReadBlock**(RandomAccessFile* file, const ReadOptions& options, const BlockHandle&handle, BlockContents* result);

下面来分析实现逻辑：

- S1 初始化结果 result，BlockContents 是一个有 3 个成员的结构体。

```
1.  result->data = Slice();
2.  result->cachable = false; // 无 cache
3.  result->heap_allocated =false; // 非 heap 分配
```

- S2 根据 handle 指定的偏移和大小，读取 block 内容，type 和 crc32 值，其中常量 kBlockTrailerSize=5= 1byte 的 type 和 4bytes 的 crc32。

Status s = file->Read(handle.offset(),handle.size() + kBlockTrailerSize, &contents, buf);

- S3 如果 option 要校验 CRC32，则计算 content + type 的 CRC32 并校验。
- S4 最后根据 type 指定的存储类型，如果是非压缩的，则直接取数据赋给 result，否则先解压，把解压结果赋给 result，目前支持的是 snappy 压缩。

另外，文件的 Read 接口返回的 Slice 结果，其 data 指针可能没有使用我们传入的 buf，如果没有，那么释放 Slice 的 data 指针就是我们的事情，否则就是文件来管理的。

批注 [g4]: ?

```
1.  if (data != buf) { // 文件自己管理，cacheable 等标记设置为 false
2.     delete[] buf;
3.     result->data =Slice(data, n);
4.     result->heap_allocated= result->cachable =false;
5.   } else { // 读取者自己管理，标记设置为 true
6.     result->data =Slice(buf, n);
7.     result->heap_allocated= result->cachable = true;
8.   }
```

对于压缩存储，解压后的字符串存储需要读取者自行分配的，所以标记都是 true。

6.5.4 Table::ReadMeta()

根据 footer 的 metaindex_handle 读取 Meta Block，构造一个 meta 的 Block 对象，新建一个该 Block 对象的 Iterator，查找 key 为“filter.***”的 entry，然后根据该 entry 的 value 调用 ReadFilter；

解决完了 Block 的读取，接下来就是 meta 的读取了。函数声明为：

void Table::**ReadMeta**(const Footer& footer);

函数逻辑并不复杂，

- S1 首先调用 ReadBlock 读取 meta 的内容。

```
1.  if(rep_->options.filter_policy == NULL) return; // 不需要 metadata
2.  ReadOptions opt;
3.  BlockContents contents;
4.  if (!ReadBlock(rep_->file,opt, footer.metaindex_handle(), &contents).ok()) {
        return; // 失败了也没报错，因为没有 meta 信息也没关系
5.  }
```

- S2 根据读取的 content 构建 Block，找到指定的 filter；如果找到了就调用 ReadFilter 构建 filter 对象。Block 的分析留在后面。

[cpp] view plaincopy

```cpp
Block* meta = newBlock(contents);
Iterator* iter = meta->NewIterator(BytewiseComparator());
std::string key ="filter.";
key.append(rep_->options.filter_policy->Name());
iter->Seek(key);
if (iter->Valid() &&iter->key() == Slice(key)) ReadFilter(iter->value());
delete iter;
delete meta;
```

6.5.5 Table::ReadFilter()

根据 Block Handle 指定的偏移和大小，读取 filter，函数声明：

void **ReadFilter**(const Slice& filter_handle_value);

简单分析下函数逻辑

- S1 从传入的 filter_handle_value Decode 出 BlockHandle，这是 filter 的偏移和大小；

BlockHandle filter_handle;

filter_handle.DecodeFrom(&filter_handle_value);

- S2 根据解析出的位置读取 filter 内容，ReadBlock。如果 block 的 heap_allocated 为 true，表明需要自行释放内存，因此要把指针保存在 filter_data 中。最后根据读取的 data 创建 FilterBlockReader 对象。

```cpp
ReadOptions opt;
BlockContents block;
ReadBlock(rep_->file, opt,filter_handle, &block);
if (block.heap_allocated)rep_->filter_data = block.data.data(); // 需要自行释放内存
rep_->filter = newFilterBlockReader(rep_->options.filter_policy, block.data);
```

以上就是 sstable 文件的读取操作，不算复杂。

6.6 遍历 Table

“Two_level_iterator.cc Table.cc”

6.6.1 遍历接口

Table 导出了一个返回 Iterator 的接口，通过 Iterator 对象，调用者就可以遍历 Table 的内容，它简单的返回了一个 TwoLevelIterator 对象。见函数实现：

```
1.  Iterator* Table::NewIterator(const ReadOptions&options) const
2.  {
3.      return NewTwoLevelIterator(
4.            rep_->index_block->NewIterator(rep_->options.comparator),
5.            &Table::BlockReader,const_cast<Table*>(this), options);
6.  }
7.  // 函数 NewTwoLevelIterator 创建了一个 TwoLevelIterator 对象：
8.  Iterator* NewTwoLevelIterator(
9.      Iterator* index_iter, BlockFunction block_function,
10.     void* arg, constReadOptions& options)
11. {
12.   return newTwoLevelIterator(index_iter, block_function, arg, options);
13. }
```

这里有一个函数指针 BlockFunction，类型为：
typedef Iterator* (***BlockFunction**)(void*, const ReadOptions&, constSlice&);
为什么叫 TwoLevelIterator 呢，下面就来看看。

6.6.2 TwoLevelIterator

它也是 Iterator 的子类，之所以叫 two level 应该是不仅可以迭代其中存储的对象，它还接受了一个函数 **BlockFunction**，可以遍历存储的对象，可见它是专门为 Table 定制的。我们已经知道各种 Block 的存储格式都是相同的，但是各自 block data 存储的 k/v 又互不相同，于是我们就需要一个途径，能够在使用同一个方式遍历不同的 block 时，又能解析这些 k/v。这就是 BlockFunction，它又返回了一个针对 block data 的 Iterator。Block 和 block data 存储的 k/v 对的 key 是统一的。
先来看 TwoLevelIterator 类的主要成员变量：

```
1.   BlockFunction block_function_; // block 操作函数
2.  void* arg_;  // BlockFunction 的自定义参数
3.  const ReadOptions options_; // BlockFunction 的 read option 参数
4.  Status status_;  // 当前状态
5.  IteratorWrapper index_iter_; // 遍历 block 的迭代器
6.  IteratorWrapper data_iter_; // May be NULL-遍历 block data 的迭代器
7.  // 如果 data_iter_ != NULL，data_block_handle_保存的是传递给
8.  // block_function_的 index value，以用来创建 data_iter_
9.  std::string data_block_handle_;
```

下面分析一下对于 Iterator 几个接口的实现。

- S1 对于其 Key 和 Value 接口都是返回的 data_iter_对应的 key 和 value：

```
virtual bool Valid() const {
  return data_iter_.Valid();
}
virtual Slice key() const {
  assert(Valid());
  return data_iter_.key();
}
virtual Slice value() const {
  assert(Valid());
  return data_iter_.value();
}
```

- S2 在分析 Seek 系函数之前，有必要先了解下面这几个函数的用途。

```
void InitDataBlock();
void SetDataIterator(Iterator* data_iter); //设置 date_iter_ = data_iter
voidSkipEmptyDataBlocksForward();
voidSkipEmptyDataBlocksBackward();
```

  - S2.1 首先是 **InitDataBlock**()，它是根据 index_iter 来初始化 data_iter；当定位到新的 block 时，需要更新 data Iterator，指向该 block 中 k/v 对的合适位置，函数如下：

```
if (!index_iter_.Valid()) SetDataIterator(NULL);// index_iter 非法
else {
  Slice handle = index_iter_.value();
  if (data_iter_.iter() != NULL
&& handle.compare(data_block_handle_) == 0) {
    //data_iter 已经在该 block data 上了，无须改变
  } else { // 根据 handle 数据定位 data iter
    Iterator* iter = (*block_function_)(arg_, options_, handle);
    data_block_handle_.assign(handle.data(), handle.size());
    SetDataIterator(iter);
  }
}
```

  - S2.2 **SkipEmptyDataBlocksForward**，向前跳过空的 datablock，函数实现如下：

```
while (data_iter_.iter() == NULL|| !data_iter_.Valid()) { // 跳到下一个
block
  if (!index_iter_.Valid()) { // 如果 index iter 非法，设置 data iteration 为
NULL
    SetDataIterator(NULL);
```

```
    return;
  }
  index_iter_.Next();
  InitDataBlock();
  if (data_iter_.iter() != NULL)data_iter_.SeekToFirst(); // 跳转到开始
}
```

- S2.3 **SkipEmptyDataBlocksBackward**，向后跳过空的 datablock，函数实现如下：

```
while (data_iter_.iter() == NULL|| !data_iter_.Valid()) { // 跳到前一个 block
  if (!index_iter_.Valid()) { // 如果 index iter 非法，设置 data iteration 为 NULL
    SetDataIterator(NULL);
    return;
  }
  index_iter_.Prev();
  InitDataBlock();
  if (data_iter_.iter() != NULL)data_iter_.SeekToLast();// 跳转到开始
}
```

- S3 了解了几个跳转的辅助函数，再来看 **Seek** 系接口。

```
void TwoLevelIterator::Seek(const Slice& target) {
  index_iter_.Seek(target);
  InitDataBlock(); // 根据 index iter 设置 data iter
  if (data_iter_.iter() != NULL) data_iter_.Seek(target); // 调整 data iter 跳转到 target
  SkipEmptyDataBlocksForward(); // 调整 iter，跳过空的 block
}

void TwoLevelIterator::SeekToFirst() {
  index_iter_.SeekToFirst();
  InitDataBlock();// 根据 index iter 设置 data iter
  if (data_iter_.iter() != NULL)data_iter_.SeekToFirst();
  SkipEmptyDataBlocksForward();// 调整 iter，跳过空的 block
}

void TwoLevelIterator::SeekToLast() {
  index_iter_.SeekToLast();
  InitDataBlock();// 根据 index iter 设置 data iter
  if (data_iter_.iter() != NULL)data_iter_.SeekToLast();
  SkipEmptyDataBlocksBackward();// 调整 iter，跳过空的 block
}
```

```
21.
22. void TwoLevelIterator::Next() {
23.   assert(Valid());
24.   data_iter_.Next();
25.   SkipEmptyDataBlocksForward();// 调整 iter，跳过空的 block
26. }
27.
28. void TwoLevelIterator::Prev() {
29.   assert(Valid());
30.   data_iter_.Prev();
31.   SkipEmptyDataBlocksBackward();// 调整 iter，跳过空的 block
32. }
```

6.6.3 **BlockReader()** *"Table.cc Line154"*

"Table.cc Line213"：上面传递给 TwoLevelIterator 的函数是 Table::BlockReader 函数，声明如下：

static Iterator* Table::**BlockReader**(void* arg, const ReadOptions&options,
const Slice& index_value);

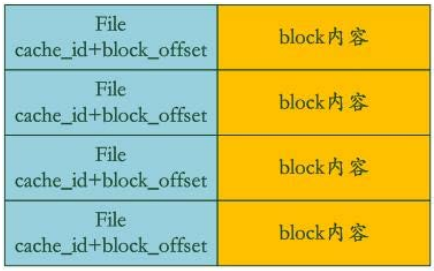
它根据参数指明的 blockdata，返回一个 iterator 对象，调用者就可以通过这个 iterator 对象遍历 blockdata 存储的 k/v 对，这其中用到了 LRUCache。

函数实现逻辑如下：

- S1 从参数中解析出 BlockHandle 对象，其中 arg 就是 Table 对象，index_value 存储的是 BlockHandle 对象，读取 Block 的索引。

```
1.  Table* table =reinterpret_cast<Table*>(arg);
2.  Block* block = NULL;
3.  Cache::Handle* cache_handle =NULL;
4.  BlockHandle handle;
5.  Slice input = index_value;
6.  Status s =handle.DecodeFrom(&input);
```

- S2 根据 block handle，首先尝试从 cache 中直接取出 block（根据 rep_对象的 cache_id 和 block handle 的 offset 组成的 16 位 key 在 block cache 中查找），不在 cache 中则调用 ReadBlock 从文件读取，读取成功后，根据 option 尝试将 block 加入到 LRU cache 中。并在 Insert 的时候注册了释放函数 DeleteCachedBlock。



Block Cache

```
1.  Cache* block_cache =table->rep_->options.block_cache;
2.  BlockContents contents;
3.  if (block_cache != NULL) {
4.    char cache_key_buffer[16]; // cache key的格式为table.cache_id + offset
5.    EncodeFixed64(cache_key_buffer, table->rep_->cache_id);
6.    EncodeFixed64(cache_key_buffer+8, handle.offset());
7.    Slice key(cache_key_buffer,sizeof(cache_key_buffer));
8.    cache_handle =block_cache->Lookup(key); // 尝试从LRU cache中查找
9.    if (cache_handle != NULL) { // 找到则直接取值
10.     block =reinterpret_cast<Block*>(block_cache->Value(cache_handle));
11.   } else { // 否则直接从文件读取
12.     s =ReadBlock(table->rep_->file, options, handle, &contents);
13.     if (s.ok()) {
14.       block = new Block(contents);
15.       if (contents.cachable&& options.fill_cache) // 尝试加到cache中
16.         cache_handle = block_cache->Insert(
17.             key, block,block->size(), &DeleteCachedBlock);
18.     }
19.   }
20. } else {
21.   s = ReadBlock(table->rep_->file, options, handle, &contents);
22.   if (s.ok()) block = newBlock(contents);
23. }
```

- S3 如果读取到了 block，调用 Block::NewIterator 接口创建 Iterator，如果 cache handle 为 NULL，则注册 DeleteBlock，否则注册 ReleaseBlock，事后清理。

批注 [g5]: ?

```
1.  Iterator* iter;
2.  if (block != NULL) {
3.    iter =block->NewIterator(table->rep_->options.comparator);
4.    if (cache_handle == NULL)  iter->RegisterCleanup(&DeleteBlock,block, NULL)
    ;
5.    else iter->RegisterCleanup(&ReleaseBlock,block_cache, cache_handle);
```

```
6.  } else iter = NewErrorIterator(s);
```

处理结束，最后返回 iter。这里简单列下这几个静态函数，都很简单：

```
1.  static void DeleteBlock(void* arg, void* ignored) {
2.   delete reinterpret_cast<Block*>(arg);
3.  }
4.  static void DeleteCachedBlock(const Slice& key, void* value) {
5.    Block* block = reinterpret_cast<Block*>(value);
6.    delete block;
7.  }
8.
9.  static void ReleaseBlock(void* arg, void* h) {
10.   Cache* cache =reinterpret_cast<Cache*>(arg);
11.   Cache::Handle* handle =reinterpret_cast<Cache::Handle*>(h);
12.   cache->Release(handle);
13. }
```

6.7 定位 key

“Table.cc”

这里并不是精确的定位，而是在 Table 中找到第一个>=指定 key 的 k/v 对，然后返回其 value 在 sstable 文件中的偏移。也是 Table 类的一个接口：
uint64_t Table::**ApproximateOffsetOf**(const Slice& key) const;
函数实现比较简单：

- S1 调用 Block::Iter 的 Seek 函数定位

```
1.  Iterator* index_iter=rep_->index_block->NewIterator(rep_->options.comparator
    );
2.  index_iter->Seek(key);
3.  uint64_t result;
```

- S2 如果 index_iter 是合法的值，并且 Decode 成功，返回结果 offset。

```
1.  BlockHandle handle;
2.  handle.DecodeFrom(&index_iter->value());
3.  result = handle.offset();
```

- S3 其它情况，设置 result 为 rep_->metaindex_handle.offset()，metaindex 的偏移在文件结尾附近。

6.8 获取 Key

“Table.cc”—InternalGet()

InternalGet，这是为 TableCache 开的一个口子。这是一个 private 函数，声明为：
Status Table::**InternalGet**(const ReadOptions& options, constSlice& k,

void*arg, void (***saver**)(void*, const Slice&, const Slice&))

其中又有函数指针，在找到数据后，就调用传入的函数指针 **saver** 执行调用者的自定义处理逻辑，并且 TableCache 可能会做缓存。

函数逻辑如下：

- S1 首先根据传入的 key 定位数据，这需要 indexblock 的 Iterator。

```
1.  Iterator* iiter =rep_->index_block->NewIterator(rep_->options.comparator);
2.  iiter->Seek(k);
```

- S2 如果 key 是合法的，取出其 filter 指针，如果使用了 filter，则检查 key 是否存在，这可以快速判断，提升效率。

```
1.  Status s;
2.  Slice handle_value =iiter->value();
3.  FilterBlockReader* filter = rep_->filter;
4.  BlockHandle handle;
5.  if (filter != NULL && handle.DecodeFrom(&handle_value).ok()
6.    && !filter->KeyMayMatch(handle.offset(),k)) { // key 不存在
7.  } else{// 否则就要读取 block，并查找其 k/v 对
8.      Slice handle = iiter->value();
9.      Iterator* block_iter =BlockReader(this, options, iiter->value());
10.     block_iter->Seek(k);
11.     if (block_iter->Valid())(*saver)(arg, block_iter->key(), block_iter->val
    ue());
12.     s = block_iter->status();
13.     delete block_iter;
14. }
```

- S3 最后返回结果，删除临时变量。

```
1.  if (s.ok()) s =iiter->status();
2.  delete iiter;
3.  return s;
```

随着有关 sstable 文件读取的结束，sstable 的源码也就分析完了，其中我们还遗漏了一些功课要做，那就是 Filter 和 TableCache 部分。

7 TableCache

这章的内容比较简单，篇幅也不长。

7.1 TableCache 简介

TableCache 缓存的是 Table 对象，每个 DB 一个，它内部使用一个 LRUCache 缓存所有的 table 对象，实际上其内容是文件编号{file number, TableAndFile*}。TableAndFile 是一个拥有 2 个变量的结构体：RandomAccessFile*和 Table*；

TableCache 类的主要成员变量有：

```
Env* const env_; // 用来操作文件
const std::string dbname_; // db 名
Cache* cache_;  // LRUCache
```

三个函数接口，其中的参数@file_number 是文件编号，@file_size 是文件大小：

```
void Evict(uint64_t file_number);
// 该函数用以清除指定文件所有 cache 的 entry，函数实现很简单，就是根据 file number 清除 cache 对象。

EncodeFixed64(buf,file_number); cache_->Erase(Slice(buf, sizeof(buf)));
Iterator* NewIterator(const ReadOptions& options, uint64_t file_number,
                      uint64_t file_size, Table**tableptr = NULL);
//该函数为指定的 file 返回一个 iterator(对应的文件长度必须是"file_size"字节). 如果 tableptr 不是 NULL，那么*tableptr 保存的是底层的 Table 指针。返回的*tableptr 是 cache 拥有的，不能被删除，生命周期同返回的 iterator

Status Get(constReadOptions& options,
           uint64_t file_number,uint64_t file_size,
           const Slice& k,void* arg,
           void(*handle_result)(void*, const Slice&, const Slice&));
// 这是一个查找函数，如果在指定文件中 seek 到 internal key "k" 找到一个 entry，就调用 (*handle_result)(arg,found_key, found_value).
```

7.2 TableCache::Get()

先来看看 Get 接口，只有几行代码：

```
Cache::Handle* handle = NULL;
Status s =FindTable(file_number, file_size, &handle);
if (s.ok()) {
  Table* t =reinterpret_cast<TableAndFile*>(cache_->Value(handle))->table;
  s = t->InternalGet(options,k, arg, saver);
  cache_->Release(handle);
```

```
7.  }
8.  return s;
```

首先根据 file_number 找到 Table 的 cache 对象，如果找到了就调用 Table::InternalGet，对查找结果的处理在调用者传入的 saver 回调函数中。

Cache 在 Lookup 找到 cache 对象后，如果不再使用需要调用 Release 减引用计数。这个见 Cache 的接口说明。

7.3 TableCache::NewIterator() 遍历

函数 **NewIterator**()，返回一个可以遍历 Table 对象的 Iterator 指针，函数逻辑：

- S1 初始化 tableptr，调用 FindTable，返回 cache 对象

```
1.  if (tableptr != NULL) *tableptr =NULL;
2.  Cache::Handle* handle = NULL;
3.  Status s =FindTable(file_number, file_size, &handle);
4.  if (!s.ok()) return NewErrorIterator(s);
```

- S2 从 cache 对象中取出 Table 对象指针，调用其 NewIterator 返回 Iterator 对象，并为 Iterator 注册一个 cleanup 函数。

```
1.  Table* table =reinterpret_cast<TableAndFile*>(cache_->Value(handle))->table;

2.  Iterator* result =table->NewIterator(options);
3.  result->RegisterCleanup(&UnrefEntry, cache_, handle);
4.  if (tableptr != NULL) *tableptr= table;
5.  return result;
```

7.4 TableCache::FindTable()

前面的遍历和 Get 函数都依赖于 FindTable 这个私有函数完成对 cache 的查找，下面就来看看该函数的逻辑。函数声明为：

Status **FindTable**(uint64_t file_number, uint64_t file_size,Cache::Handle** handle)

函数流程为：

- S1 首先根据 file number 从 cache 中查找 table，找到就直接返回成功。

```
char buf[sizeof(file_number)];
EncodeFixed64(buf, file_number);
Slice key(buf, sizeof(buf));
*handle = cache_->Lookup(key);
```

- S2 如果没有找到，说明 table 不在 cache 中，则根据 file number 和 db name 打开一个 RadomAccessFile。Table 文件格式为：<db name>/<filenumber(%6u)>.sst。如果文件打开成功，则调用 Table::Open 读取 sstable 文件。

```
std::string fname =TableFileName(dbname_, file_number);
RandomAccessFile* file = NULL;
Table* table = NULL;
s =env_->NewRandomAccessFile(fname, &file);
if (s.ok()) s =Table::Open(*options_, file, file_size, &table);
```

- S3 如果 Table::Open 成功则，插入到 Cache 中。

```
TableAndFile* tf = newTableAndFile(table, file);
*handle = cache_->Insert(key,tf, 1, &DeleteEntry);
```

如果失败，则删除 file，直接返回失败，失败的结果是不会 cache 的。

7.5 辅助函数

有点啰嗦，不过还是写一下吧。其中一个是为 LRUCache 注册的删除函数 DeleteEntry。

```
static void DeleteEntry(const Slice& key, void* value) {
  TableAndFile* tf =reinterpret_cast<TableAndFile*>(value);
  delete tf->table;
  delete tf->file;
  delete tf;
}
```

另外一个是为 Iterator 注册的清除函数 UnrefEntry。

```
static void UnrefEntry(void* arg1, void* arg2) {
  Cache* cache =reinterpret_cast<Cache*>(arg1);
  Cache::Handle* h =reinterpret_cast<Cache::Handle*>(arg2);
  cache->Release(h);
}
```

8 FilterPolicy&Bloom

8.1 FilterPolicy

因名知意，FilterPolicy 是用于 key 过滤的，可以快速的排除不存在的 key。前面介绍 Table 的时候，在 Table::InternalGet 函数中有过一面之缘。

FilterPolicy 有 3 个接口：

virtual const char* **Name**() const = 0; // 返回 filter 的名字

virtual void **CreateFilter**(const Slice* keys, int n, std::string* dst)const = 0;

virtual bool **KeyMayMatch**(const Slice& key, const Slice& filter)const = 0;

> CreateFilter 接口，它根据指定的参数创建过滤器，并将结果 append 到 dst 中，注意：不能修改 dst 的原始内容，只做 append。

参数@keys[0,n-1]包含依据用户提供的 comparator 排序的 key 列表--可重复，并把根据这些 key 创建的 filter 追加到@*dst 中。

> KeyMayMatch，参数@filter 包含了调用 CreateFilter 函数 append 的数据，如果 key 在传递函数 CreateFilter 的 key 列表中，则必须返回 true。

注意，它不需要精确，也就是即使 key 不在前面传递的 key 列表中，也可以返回 true，但是如果 key 在列表中，就必须返回 true。

涉及到的类如图 8.1-1 所示。

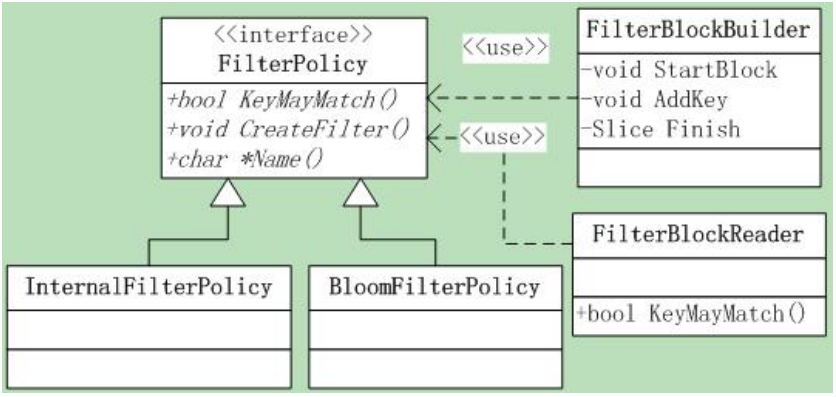


图 8.1-1

8.2 InternalFilterPolicy

这是一个简单的 FilterPolicy 的 wrapper，以方便的把 FilterPolicy 应用在 InternalKey 上，InternalKey 是 Leveldb 内部使用的 key，这些前面都讲过。它所做的就是从 InternalKey 拆分得到 user key，然后在 user key 上做 FilterPolicy 的操作。

它有一个成员：constFilterPolicy* const user_policy_;
其 Name()返回的是 user_policy_->Name()；

```
1.  bool InternalFilterPolicy::KeyMayMatch(const Slice& key, constSlice& f) cons
    t {
2.    returnuser_policy_->KeyMayMatch(ExtractUserKey(key), f);
3.  }
4.
5.  void InternalFilterPolicy::CreateFilter(const Slice* keys, int n,std::string
    * dst) const {
6.    Slice* mkey =const_cast<Slice*>(keys);
7.    for (int i = 0; i < n; i++)mkey[i] = ExtractUserKey(keys[i]);
8.    user_policy_->CreateFilter(keys, n, dst);
9.  }
```

8.3 BloomFilter

8.3.1 基本理论

Bloom Filter 实际上是一种 hash 算法，数学之美系列有专门介绍。它是由巴顿.布隆于一九七零年提出的，它实际上是一个很长的二进制向量和一系列随机映射函数。
Bloom Filter 将元素映射到一个长度为 m 的 bit 向量上的一个 bit，当这个 bit 是 1 时，就表示这个元素在集合内。使用 hash 的缺点就是元素很多时可能有冲突，为了减少误判，就使用 k 个 hash 函数计算出 k 个 bit，只要有一个 bit 为 0，就说明元素肯定不在集合内。下面的图 8.3-1 是一个示意图。

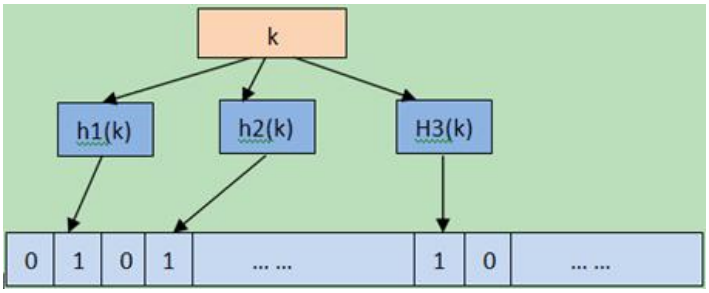


图 8.3-1

在 leveldb 的实现中，Name()返回"leveldb.BuiltinBloomFilter"，因此 metaindex block 中的 key 就是"filter.leveldb.BuiltinBloomFilter"。Leveldb 使用了 double hashing 来模拟多个 hash 函数，当然这里不是用来解决冲突的。

和线性再探测（linearprobing）一样，Double hashing 从一个 hash 值开始，重复向

前迭代，直到解决冲突或者搜索完 hash 表。不同的是，double hashing 使用的是另外一个 hash 函数，而不是固定的步长。

给定两个独立的 hash 函数 h1 和 h2，对于 hash 表 T 和值 k，第 i 次迭代计算出的位置就是：h(i, k) = (h1(k) + i*h2(k)) mod |T|。

对此，Leveldb 选择的 hash 函数是：
Gi(x)=H1(x)+iH2(x)
H2(x)=(H1(x)>>17) | (H1(x)<<15)

H1 是一个基本的 hash 函数，H2 是由 H1 循环右移得到的，Gi(x)就是第 i 次循环得到的 hash 值。【理论分析可参考论文 Kirsch,Mitzenmacher2006】

在 bloom_filter 的数据的最后一个字节保存 k_的值，k_实际上就是 G(x)的个数，也就是计算时采用的 hash 函数个数。

8.3.2 BloomFilter 参数

这里先来说下其两个成员变量：bits_per_key_和 key_；其实这就是 Bloom Hashing 的两个关键参数。
变量 k_实际上就是模拟的 hash 函数的个数；
关于变量 bits_per_key_，对于 n 个 key，其 hash table 的大小就是 bits_per_key_。它的值越大，发生冲突的概率就越低，那么 bloom hashing 误判的概率就越低。因此这是一个时间空间的 trade-off。
对于 hash(key)，在平均意义上，发生冲突的概率就是 1/ bits_per_key_。
它们在构造函数中根据传入的参数 bits_per_key 初始化。

```
1.  bits_per_key_ = bits_per_key;
2.  k_ =static_cast<size_t>(bits_per_key * 0.69); // 0.69 =~ ln(2)
3.  if (k_ < 1) k_ = 1;
4.  if (k_ > 30) k_ = 30;
```

模拟 hash 函数的个数 k_取值为 bits_per_key_*ln(2)，为何不是 0.5 或者 0.4 了，可能是什么理论推导的结果吧，不了解了。//TODO

8.3.3 建立 BloomFilter

了解了上面的理论，再来看 leveldb 对 Bloom Filter 的实现就轻松多了，先来看 Bloom Filter 的构建。这就是 FilterPolicy::CreateFilter 接口的实现：
void CreateFilter(const Slice* keys, int n, std::string* dst) const
下面分析其实现代码，大概有如下几个步骤：

- S1 首先根据 key 个数分配 filter 空间，并圆整到 8byte。

```
size_t bits = n * bits_per_key_;
if (bits < 64) bits = 64; // 如果 n 太小 FP 会很高，限定 filter 的最小长度
size_t bytes = (bits + 7) / 8;// 圆整到 8byte
bits = bytes * 8; // bit 计算的空间大小
const size_t init_size =dst->size();
dst->resize(init_size +bytes, 0); // 分配空间
```

- S2 在 filter 最后的字节位压入 hash 函数个数
  dst->push_back(static_cast<char>(k_));// Remember # of probes in filter
- S3 对于每个 key，使用 double-hashing 生产一系列的 hash 值 h(K_个)，设置 bits array 的第 h 位=1。

```
char* array =&(*dst)[init_size];
for (size_t i = 0; i < n;i++) {
  // double-hashing，分析参见[Kirsch,Mitzenmacher 2006]
  uint32_t h =BloomHash(keys[i]); // h1 函数
  const uint32_t delta = (h>> 17) | (h << 15); // h2 函数、由
h1 Rotate right 17 bits
  for (size_t j = 0; j <k_; j++) { // double-hashing 生产 k_个的 hash 值
    const uint32_t bitpos = h% bits; // 在 bits array 上设置第 bitpos 位
    array[bitpos/8] |= (1<< (bitpos % 8));
    h += delta;
  }
}
```

Bloom Filter 的创建就完成了。

8.3.4 查找 BloomFilter

在指定的 filer 中查找 key 是否存在，这就是 bloom filter 的查找函数：
bool KeyMayMatch(const Slice& key, const Slice& bloom_filter)，函数逻辑如下：

- S1 准备工作，并做些基本判断。

```
const size_t len =bloom_filter.size();
if (len < 2) return false;
const char* array = bloom_filter.data();
const size_t bits = (len - 1)* 8;
const size_t k = array[len-1];// 使用 filter 的 k，而不是 k_，这样更灵活
if (k > 30) return true; // 为短 bloom filter 保留，当前认为直接 match
```

- S2 计算 key 的 hash 值，重复计算阶段的步骤，循环计算 k 个 hash 值，只要有一个结果对应的 bit 位为 0，就认为不匹配，否则认为匹配。

```
uint32_t h = BloomHash(key);
const uint32_t delta = (h>> 17) | (h << 15);  // Rotate right 17 bits
for (size_t j = 0; j < k;j++) {
```

```
4.    const uint32_t bitpos = h %bits;
5.    if ((array[bitpos/8] &(1 << (bitpos % 8))) == 0) return false; // notmatch

6.    h += delta;
7.  }
8.  return true; // match
```

8.4 Filter Block 格式

Filter Block 也就是前面 sstable 中的 meta block，位于 data block 之后。
如果打开 db 时指定了 FilterPolicy，那么每个创建的 table 都会保存一个 filter block，table 中的 metaindex 就包含一条从"filter.<N>到 filter block 的 BlockHandle 的映射，其中"<N>"是 filter policy 的 Name()函数返回的 string。

Filter block 存储了一连串的 filter 值，其中第 i 个 filter 保存的是 block b 中所有的 key 通过 FilterPolicy::CreateFilter()计算得到的结果，block b 在 sstable 文件中的偏移满足[i*base ... (i+1)*base-1]。

当前 base 是 2KB，举个例子，如果 block X 和 Y 在 sstable 的起始位置都在[0KB, 2KB-1]中，X 和 Y 中的所有 key 调用 FilterPolicy::CreateFilter()的计算结果都将生产到同一个 filter 中，而且该 filter 是 filter block 的第一个 filter。

Filter block 也是一个 block，其格式遵从 block 的基本格式：|block data| type | crc32|。其中 block dat 的格式如图 8.4-1 所示。

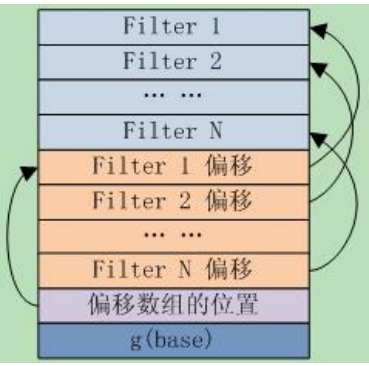


图 8.4-1 filter block data

了解了格式，再分析构建和读取 filter 的代码就很简单了。

8.5 构建 FilterBlock

8.5.1 FilterBlockBuilder

了解了 filter 机制，现在来看看 filter block 的构建，这就是类 FilterBlockBuilder。它为指定的 table 构建所有的 filter，结果是一个 string 字符串，并作为一个 block 存放在 table 中。它有三个函数接口：

```
1.  // 开始构建新的 filter block，TableBuilder 在构造函数和 Flush 中调用
2.  void StartBlock(uint64_tblock_offset);
3.  // 添加 key，TableBuilder 每次向 data block 中加入 key 时调用
4.  void AddKey(const Slice&key);
5.  // 结束构建，TableBuilder 在结束对 table 的构建时调用
6.  Slice Finish();
```

FilterBlockBuilder 的构建顺序必须满足如下范式：(StartBlock AddKey*)* Finish，显然这和前面讲过的 BlockBuilder 有所不同。
其成员变量有：

```
1.  const FilterPolicy* policy_; // filter 类型，构造函数参数指定
2.  std::string keys_;          //Flattened key contents
3.  std::vector<size_t> start_;   // 各 key 在 keys_中的位置
4.  std::string result_;          // 当前计算出的 filter data
5.  std::vector<uint32_t>filter_offsets_; // 各个 filter 在 result_中的位置
6.  std::vector<Slice> tmp_keys_;// policy_->CreateFilter()参数
```

前面说过 base 是 2KB，这对应两个常量 kFilterBase =11, kFilterBase =(1<<kFilterBaseLg)；其实从后面的实现来看 tmp_keys_完全不必作为成员变量，直接作为函数 GenerateFilter() 的栈变量就可以。下面就分别分析三个函数**接口**。

8.5.2 FilterBlockBuilder::StartBlock()

它根据参数 block_offset 计算出 filter index，然后循环调用 GenerateFilter 生产新的 Filter。

```
1.  uint64_t filter_index =(block_offset / kFilterBase);
2.  assert(filter_index >=filter_offsets_.size());
3.  while (filter_index >filter_offsets_.size()) GenerateFilter();
```

我们来到 GenerateFilter 这个函数，看看它的逻辑。

```
1.  //S1 如果 filter 中 key 个数为 0，则直接压入 result_.size()并返回
2.    const size_t num_keys =start_.size();
3.    if (num_keys == 0) { // there are no keys for this filter
4.      filter_offsets_.push_back(result_.size()); //result_.size()应该是 0
```

```
    return;
  }
//S2 从 key 创建临时 key list，根据 key 的序列字符串 kyes_和各 key 在 keys_中的开始位置 start_依次提取出 key。
  start_.push_back(keys_.size());  // Simplify lengthcomputation
  tmp_keys_.resize(num_keys);
  for (size_t i = 0; i <num_keys; i++) {
    const char* base =keys_.data() + start_[i]; // 开始指针
    size_t length = start_[i+1] -start_[i]; // 长度
    tmp_keys_[i] = Slice(base,length);
  }
//S3 为当前的 key 集合生产 filter，并 append 到 result_
  filter_offsets_.push_back(result_.size());
  policy_->CreateFilter(&tmp_keys_[0], num_keys, &result_);
//S4 清空，重置状态
  tmp_keys_.clear();
  keys_.clear();
  start_.clear();
```

8.5.3 FilterBlockBuilder::AddKey()

这个接口很简单，就是把 key 添加到 key_中，并在 start_中记录位置。

```
Slice k = key;
start_.push_back(keys_.size());
keys_.append(k.data(),k.size());
```

8.5.4 FilterBlockBuilder::Finish()

调用这个函数说明整个 table 的 data block 已经构建完了，可以生产最终的 filter block 了，在 TableBuilder::Finish 函数中被调用，向 sstable 写入 meta block。
函数逻辑为：

```
//S1 如果 start_数字不空，把为的 key 列表生产 filter
  if (!start_.empty()) GenerateFilter();
//S2 从 0 开始顺序存储各 filter 的偏移值，见 filter block data 的数据格式。
  const uint32_t array_offset =result_.size();
  for (size_t i = 0; i < filter_offsets_.size();i++) {
    PutFixed32(&result_,filter_offsets_[i]);
  }
//S3 最后是 filter 个数，和 shift 常量（11），并返回结果
  PutFixed32(&result_,array_offset);
  result_.push_back(kFilterBaseLg);  // Save encoding parameter in result
  return Slice(result_);
```

8.5.5 简单示例

让我们根据 TableBuilder 对 FilterBlockBuilder 接口的调用范式：
(StartBlock AddKey*)* Finish 以及上面的函数实现，结合一个简单例子看看 leveldb 是如何为 data block 创建 filter block（也就是 meta block）的。
考虑两个 datablock，在 sstable 的范围分别是：Block 1 [0, 7KB-1], Block 2 [7KB, 14.1KB]

- S1 首先 TableBuilder 为 Block 1 调用 FilterBlockBuilder::StartBlock(0)，该函数直接返回；
- S2 然后依次向 Block 1 加入 k/v，其中会调用 FilterBlockBuilder::AddKey，FilterBlockBuilder 记录这些 key。
- S3 下一次 TableBuilder 添加 k/v 时，例行检查发现 Block 1 的大小超过设置，则执行 Flush 操作，Flush 操作在写入 Block 1 后，开始准备 Block 2 并更新 block offset=7KB，最后调用 FilterBlockBuilder::StartBlock(7KB)，开始为 Block 2 构建 Filter。
- S4 在 FilterBlockBuilder::StartBlock(7KB)中，计算出 filter index = 3，触发 3 次 GenerateFilter 函数，为 Block 1 添加的那些 key 列表创建 filter，其中第 2、3 次循环创建的是空 filter。

此时 filter 的结构如图 8.5-1 所示。

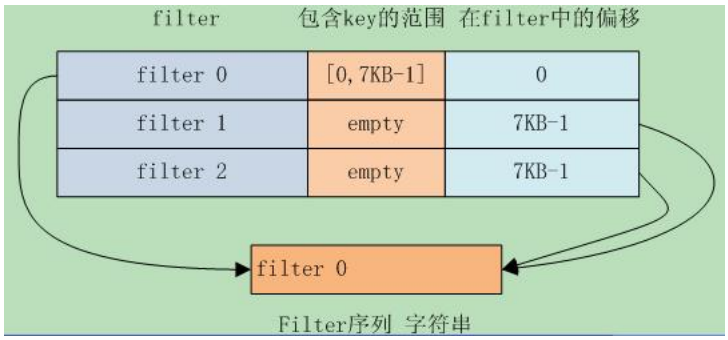


图 8.5-1

在 StartBlock(7KB)时会向 filter 的偏移数组 filter_offsets_压入两个包含空 key set 的元素，filter_offsets_[1]和 filter_offsets_[2]，它们的值都等于 7KB-1。

- S5 Block 2 构建结束，TableBuilder 调用 Finish 结束 table 的构建，这会再次触发 Flush 操作，在写入 Block 2 后，为 Block 2 的 key 创建 filter。最终的 filter 如图 8.5-2 所示。

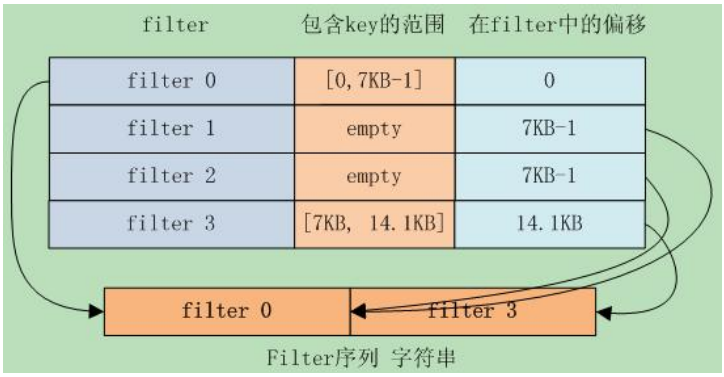


图 8.5-2

这里如果 Block 1 的范围是[0, 1.8KB-1]，Block 2 从 1.8KB 开始，那么 Block 2 将会和 Block 1 共用一个 filter，它们的 filter 都被生成到 filter 0 中。
当然在 TableBuilder 构建表时，Block 的大小是根据参数配置的，也是基本均匀的。

8.6 读取 FilterBlock

8.6.1 FilterBlockReader

FilterBlock 的读取操作在 FilterBlockReader 类中，它的主要功能是根据传入的 FilterPolicy 和 filter，进行 key 的匹配查找。
它有如下的几个成员变量：

```
1.  const FilterPolicy* policy_; // filter 策略
2.  const char* data_;    // filter data 指针 (at block-start)
3.  const char* offset_;  // offset array 的开始地址 (at block-end)
4.  size_t num_;       // offsetarray 元素个数
5.  size_t base_lg_;    // 还记得 kFilterBaseLg 吗
```

Filter 策略和 filter block 内容都由构造函数传入。一个接口函数，就是 key 的批判查找：
bool KeyMayMatch(uint64_t block_offset, const Slice& key);

8.6.2 构造

在构造函数中，根据存储格式解析出偏移数组开始指针、个数等信息。

```
1.  FilterBlockReader::FilterBlockReader(const FilterPolicy* policy, constSlice&
    contents)
2.      : policy_(policy),data_(NULL), offset_(NULL), num_(0), base_lg_(0) {
3.    size_t n = contents.size();
4.    if (n < 5) return;  // 1 byte forbase_lg_ and 4 for start of offset array
```

```
5.    base_lg_ = contents[n-1]; // 最后 1byte 存的是 base
6.    uint32_t last_word =DecodeFixed32(contents.data() + n - 5); //偏移数组的位
    置
7.    if (last_word > n - 5)return;
8.    data_ = contents.data();
9.    offset_ = data_ + last_word; // 偏移数组开始指针
10.   num_ = (n - 5 - last_word) / 4; // 计算出 filter 个数
```

8.6.3 查找

查找函数传入两个参数，@block_offset 是查找 data block 在 sstable 中的偏移，Filter 根据此偏移计算 filter 的编号；@key 是查找的 key。
声明如下：
bool FilterBlockReader::KeyMayMatch(uint64_t block_offset, constSlice& key)
它首先计算出 filterindex，根据 index 解析出 filter 的 range，如果是合法的 range，就从 data_中取出 filter，调用 policy_做 key 的匹配查询。函数实现：

```
1.  uint64_t index = block_offset>> base_lg_; // 计算出 filter index
2.  if (index < num_) {
3.    uint32_t start =DecodeFixed32(offset_ + index*4); // 解析出 filter 的 range
4.    uint32_t limit =DecodeFixed32(offset_ + index*4 + 4);
5.    if (start <= limit&& limit <= (offset_ - data_)) {
6.      Slice filter = Slice(data_ +start, limit - start); // 根据 range 得到
    filter
7.      returnpolicy_->KeyMayMatch(key, filter);
8.    } else if (start == limit) {
9.      return false;  // 空 filter 不匹配任何 key
10.   }
11. }
12. return true; // 当匹配处理
```

至此，FilterPolicy 和 Bloom 就分析完了。

9 LevelDB 框架

到此为止，基本上 Leveldb 的主要功能组件都已经分析完了，下面就是把它们组合在一起，形成一个高性能的 k/v 存储系统。这就是 leveldb::DB 类。
这里先看一下 LevelDB 的导出接口和涉及的类，后面将依次以接口分析的方式展开。
而实际上 leveldb::DB 只是一个接口类，真正的实现和框架类是 DBImpl 这个类，正是它集合了上面的各种组件。
此外，还有 Leveldb 对版本的控制，执行版本控制的是 Version 和 VersionSet 类。
在 leveldb 的源码中，DBImpl 和 VersionSet 是两个庞然大物，体量基本算是最大的。对于这两个类的分析，也会分散在打开、销毁和快照等等这些功能中，很难在一个地方集中分析。

作者在文档 impl.html 中描述了 leveldb 的实现，其中包括文件组织、compaction 和 recovery 等等。下面的 9.1 和 9.2 基本都是翻译子 impl.html 文档。
在进入框架代码之前，先来了解下 leveldb 的文件组织和管理。

9.1 DB 文件管理

9.1.1 文件类型

对于一个数据库 Level 包含如下的 6 种文件:

1 <dbname>/[0-9]+.log：db 操作日志
这就是前面分析过的操作日志，log 文件包含了最新的 db 更新，每个更新都以 append 的方式追加到文件结尾。当 log 文件达到预定大小时（缺省大约 4MB），leveldb 就把它转换为一个有序表（如下-2），并创建一个新的 log 文件。
当前的 log 文件在内存中的存在形式就是 memtable，每次 read 操作都会访问 memtable，以保证 read 读取到的是最新的数据。

2 <dbname>/[0-9]+.sst：db 的 sstable 文件
这两个就是前面分析过的静态 sstable 文件，sstable 存储了以 key 排序的元素。每个元素或者是 key 对应的 value，或者是 key 的删除标记（删除标记可以掩盖更老 sstable 文件中过期的 value）。
Leveldb 把 sstable 文件通过 level 的方式组织起来，从 log 文件中生成的 sstable 被放在 level 0。当 level 0 的 sstable 文件个数超过设置（当前为 4 个）时，leveldb 就把所有的 level 0 文件，以及有重合的 level 1 文件 merge 起来，组织成一个新的 level 1 文件（每个 level 1 文件大小为 2MB）。
Level 0 的 SSTable 文件（后缀为.sst）和 Level>1 的文件相比有特殊性：这个层级内的.sst 文件，两个文件可能存在 key 重叠。对于 Level>0，同层 sstable 文件的 key 不会重叠。考虑 level>0，level 中的文件的总大小超过 10^level MB 时（如 level=1 是 10MB，level=2 是 100MB），那么 level 中的一个文件，以及所有 level+1 中和它有重叠的文件，会被 merge 到 level+1 层的一系列新文件。Merge 操作的作用是将更新从低一级 level 迁移到最高级，只使用批量读写（最小化 seek 操作，提高效率）。

3 <dbname>/MANIFEST-[0-9]+：DB 元信息文件
它记录的是 leveldb 的元信息，比如 DB 使用的 Comparator 名，以及各 SSTable 文件的管理信息：如 Level 层数、文件名、最小 key 和最大 key 等等。

4 <dbname>/CURRENT：记录当前正在使用的 Manifest 文件
它的内容就是当前的 manifest 文件名；因为在 LevleDb 的运行过程中，随着 Compaction 的进行，新的 SSTable 文件被产生，老的文件被废弃。并生成新的 Manifest 文件来记载 sstable 的变动，而 CURRENT 则用来记录我们关心的 Manifest 文件。
当 db 被重新打开时，leveldb 总是生产一个新的 manifest 文件。Manifest 文件使用 log 的格式，对服务状态的改变（新加或删除的文件）都会追加到该 log 中。
上面的 log 文件、sst 文件、清单文件，末尾都带着序列号，其序号都是单调递增的（随着 next_file_number 从 1 开始递增），以保证不和之前的文件名重复。

5 <dbname>/log：系统的运行日志，记录系统的运行信息或者错误日志。

6 <dbname>/dbtmp：临时数据库文件，repair 时临时生成的。

这里就涉及到几个关键的 number 计数器，log 文件编号，下一个文件（sstable、log 和 manifest）编号，sequence。

所有正在使用的文件编号，包括 log、sstable 和 manifest 都应该小于下一个文件编号计数器。

9.1.2 Level 0

当操作 log 超过一定大小时（缺省是 1MB），执行如下操作：

- S1 创建新的 memtable 和 log 文件，并重导向新的更新到新 memtable 和 log 中；
- S2 在后台：
  - S2.1 将前一个 memtable 的内容 dump 到 sstable 文件；
  - S2.2 丢弃前一个 memtable；
  - S2.3 删除旧的 log 文件和 memtable
  - S2.4 把创建的 sstable 文件放到 level 0

9.2 Compaction

- 会主动触发 compact 的情况（when to call **MaybeScheduleCompaction**）
  - db 启动时，恢复完毕，会主动触发 compact。
  - 直接调用 compact 相关的函数，会把 compact 的 key-range 指定在 manual_compaction 中。
  - 每次进行写操作（put/delete）检查（DBImpl::MakeRoomForWrite()）时，如果发现 memtable 已经写满并且没有 immutable memtable，会将 memtable 置为 immutable memtable，生成新的 memtable，同时触发 compact。
  - Get 操作时，如果有超过一个 sstable 文件进行了 IO，会检查做 IO 的最后一个文件是否达到了 compact 的条件（allowed_seeks 用光,参见 Version），达到条件，则主动触发 compact。

当 level L 的总文件大小查过限制时，我们就在后台执行 compaction 操作。Compaction 操作从 level L 中选择一个文件 f，以及选择中所有和 f 有重叠的文件。如果某个 level (L+1) 的文件 ff 只是和 f 部分重合，compaction 依然选择 ff 的完整内容作为输入，在 compaction 后 f 和 ff 都会被丢弃。

另外：因为 level 0 有些特殊（同层文件可能有重合），从 level 0 到 level 1 的 compaction 就需要特殊对待：level 0 的 compaction 可能会选择多个 level 0 文件，如果它们之间有重叠。

Compaction 将选择的文件内容 merge 起来，并生成到一系列的 level (L+1)文件中，如果输出文件超过设置（2MB），就切换到新的。当输出文件的 key 范围太大以至于和超过 10 个 level (L+2)文件有重合时，也会切换。后一个规则确保了 level (L+1)的文件不会和过多的 level (L+2)文件有重合，其后的 level (L+1) compaction 不会选择过多的 level (L+2)文件。

老的文件会被丢弃，新创建的文件将加入到 server 状态中。

Compaction 操作在 key 空间中循环执行，详细讲一点就是，对于每个 level，我们记录上次 compaction 的 ending key。Level 的下一次 compaction 将选择 ending key 之后的第一个文件（如果这样的文件不存在，将会跳到 key 空间的开始）。

Compaction 会忽略被写覆盖的值，如果更高一层的 level 没有文件的范围包含了这个 key，key 的删除标记也会被忽略。

批注 [g6]: DB::Open

批注 [g7]: 目前只发现 TEST_CompactRange 函数会指定 manual_compaction 成员

批注 [g8]: DBImpl::Write -- Db_impl.cc *Line1144*

批注 [g9]: DBImpl::Get -- Db_impl.cc *Line1068*

批注 [g10]: 只是更高一层吗？
还是说全部更高层

批注 [g11]: 更高层没有该 key，该 KV 难道不是被删除吗，怎么又是被忽略？

9.2.1 时间

Level 0 的 compaction 最多从 level 0 读取 4 个 1MB 的文件，以及所有的 level 1 文件（10MB），也就是我们将读取 14MB，并写入 14MB。

Level > 0 的 compaction，从 level L 选择一个 2MB 的文件，最坏情况下，将会和 levelL+1 的 12 个文件有重合（10：level L+1 的总文件大小是 level L 的 10 倍；边界的 2：level L 的文件范围通常不会和 level L+1 的文件对齐）。因此 Compaction 将会读 26MB，写 26MB。对于 100MB/s 的磁盘 IO 来讲，compaction 将最坏需要 0.5 秒。

如果磁盘 IO 更低，比如 10MB/s，那么 compaction 就需要更长的时间 5 秒。如果 user 以 10MB/s 的速度写入，我们可能生成很多 level 0 文件（50 个来装载 5*10MB 的数据）。这将会严重影响读取效率，因为需要 merge 更多的文件。

- 解决方法 1：为了降低该问题，我们可能想增加 log 切换的阈值，缺点就是，log 文件越大，对应的 memtable 文件就越大，这需要更多的内存。
- 解决方法 2：当 level 0 文件太多时，人工降低写入速度。
- 解决方法 3：降低 merge 的开销，如把 level 0 文件都无压缩的存放在 cache 中。

9.2.2 文件数

对于更高的 level 我们可以创建更大的文件，而不是 2MB，代价就是更多突发性的 compaction。或者，我们可以考虑分区，把文件放存放多目录中。

在 2011 年 2 月 4 号，作者做了一个实验，在 ext3 文件系统中打开 100KB 的文件，结果表明可以不需要分区。

文件数	文件打开 ms
1000	9
10000	10
100000	16

9.3 Recovery & GC

9.3.1 Recovery

Db 恢复的步骤：

S1 首先从 CURRENT 读取最后提交的 MANIFEST

S2 读取 MANIFEST 内容

S3 清除过期文件

S4 这里可以打开所有的 sstable 文件，但是更好的方案是 lazy open

S5 把 log 转换为新的 level 0 sstable

S6 将新写操作导向到新的 log 文件，从恢复的序号开始

9.3.2 GC

垃圾回收，每次 compaction 和 recovery 之后都会有文件被废弃，成为垃圾文件。GC 就是删除这些文件的，它在每次 compaction 和 recovery 完成之后被调用。

9.4 版本控制

当执行一次 compaction 后，Leveldb 将在当前版本基础上创建一个新版本，当前版本就变成了历史版本。还有，如果你创建了一个 Iterator，那么该 Iterator 所依附的版本将不会

被 leveldb 删除。

在 leveldb 中，Version 就代表了一个版本，它包括当前磁盘及内存中的所有文件信息。在所有的 version 中，只有一个是 CURRENT。

VersionSet 是所有 Version 的集合，这是个 version 的管理机构。

前面讲过的 VersionEdit 记录了 Version 之间的变化，相当于 delta 增量，表示又增加了多少文件，删除了文件。也就是说：Version0 + VersionEdit --> Version1。

每次文件有变动时，leveldb 就把变动记录到一个 VersionEdit 变量中，然后通过 VersionEdit 把变动应用到 current version 上，并把 current version 的快照，也就是 db 元信息保存到 MANIFEST 文件中。

另外，MANIFEST 文件组织是以 VersionEdit 的形式写入的，它本身是一个 log 文件格式，采用 log::Writer/Reader 的方式读写，一个 VersionEdit 就是一条 log record。

9.4.1 VersionSet

和 DBImpl 一样，下面就初识一下 Version 和 VersionSet。
先来看看 Version 的成员：

```
1.  std::vector<FileMetaData*>files_[config::kNumLevels]; // sstable 文件列表
2.  // Next fileto compact based on seek stats. 下一个要 compact 的文件
3.  FileMetaData* file_to_compact_;
4.  int file_to_compact_level_;
5.  // 下一个应该 compact 的 level 和 compaction 分数.
6.  // 分数 < 1 说明 compaction 并不紧迫. 这些字段在 Finalize()中初始化
7.  double compaction_score_;
8.  int compaction_level_;
```

可见一个 Version 就是一个 sstable 文件集合，以及它管理的 compact 状态。Version 通过 Version* prev 和*next 指针构成了一个 Version 双向循环链表，表头指针则在 VersionSet 中（初始都指向自己）。

下面是 VersionSet 的成员。可见它除了通过 Version 管理所有的 sstable 文件外，还关心 manifest 文件信息，以及控制 log 文件等编号。

```
1.  //=== 第一组，直接来自于 DBImple，构造函数传入
2.  Env* const env_; // 操作系统封装
3.  const std::string dbname_;
4.  const Options* const options_;
5.  TableCache* const table_cache_; // table cache
6.  const InternalKeyComparatoricmp_;
7.  //=== 第二组，db 元信息相关
8.  uint64_t next_file_number_; // log 文件编号
9.  uint64_t manifest_file_number_; // manifest 文件编号
10. uint64_t last_sequence_;
11. uint64_t log_number_; // log 编号
12. uint64_t prev_log_number_; // 0 or backingstore for memtable being compacted

13. //=== 第三组，menifest 文件相关
```

```
14. WritableFile* descriptor_file_;
15. log::Writer* descriptor_log_;
16. //=== 第四组，版本管理
17. Version dummy_versions_; // versions 双向链表 head.
18. Version* current_; // ==dummy_versions_.prev_
19. // level 下一次 compaction 的开始 key，空字符串或者合法的 InternalKey
20. std::stringcompact_pointer_[config::kNumLevels];
```

关于版本控制大概了解其 Version 和 VersionEdit 的功能和管理范围，详细的函数操作在后面再慢慢揭开。

9.4.2 VersionEdit

LevelDB 中对 Manifest 的 Decode/Encode 是通过类 VersionEdit 完成的，Menifest 文件保存了 LevelDB 的管理元信息。VersionEdit 这个名字起的蛮有意思，每一次 compaction，都好比是生成了一个新的 DB 版本，对应的 Menifest 则保存着这个版本的 DB 元信息。

VersionEdit 并不操作文件，只是为 Manifest 文件读写准备好数据、从读取的数据中解析出 DB 元信息。

VersionEdit 有两个作用：

1 当版本间有增量变动时，VersionEdit 记录了这种变动；

2 写入到 MANIFEST 时，先将 current version 的 db 元信息保存到一个 VersionEdit 中，然后在组织成一个 log record 写入文件；

了解了 VersionEdit 的作用，来看看这个类导出的函数接口：

```
1.  void Clear(); // 清空信息
2.  void Setxxx(); // 一系列的 Set 函数，设置信息
3.  // 添加 sstable 文件信息，要求：DB 元信息还没有写入磁盘 Manifest 文件
4.  // @level：.sst 文件层次；@file 文件编号-用作文件名 @size 文件大小
5.  // @smallest, @largest：sst 文件包含 k/v 对的最大最小 key
6.  void AddFile(int level, uint64_t file, uint64_t file_size,
7.                 constInternalKey& smallest, const InternalKey& largest)
8.  void DeleteFile(int level, uint64_t file) // 从指定的 level 删除文件
9.  void EncodeTo(std::string* dst) const // 将信息 Encode 到一个 string 中
10. Status DecodeFrom(const Slice& src) // 从 Slice 中 Decode 出 DB 元信息
11. //==================下面是成员变量，由此可大概窥得 DB 元信息的内容。
12. typedef std::set< std::pair<int, uint64_t> > DeletedFileSet;
13. std::string comparator_; // key comparator 名字
14. uint64_t log_number_; // 日志编号
15. uint64_t prev_log_number_; // 前一个日志编号
16. uint64_t next_file_number_; // 下一个文件编号
17. SequenceNumber last_sequence_; // 上一个 seq
18. bool has_comparator_; // 是否有 comparator
19. bool has_log_number_;// 是否有 log_number_
20. bool has_prev_log_number_;// 是否有 prev_log_number_
21. bool has_next_file_number_;// 是否有 next_file_number_
22. bool has_last_sequence_;// 是否有 last_sequence_
```

```
23. std::vector< std::pair<int, InternalKey> >compact_pointers_; // compact 点
24. DeletedFileSet deleted_files_; // 删除文件集合
25. std::vector< std::pair<int, FileMetaData> > new_files_; // 新文件集合
```

Set 系列的函数都很简单，就是根据参数设置相应的信息。

AddFile 函数就是根据参数生产一个 FileMetaData 对象，把 sstable 文件信息添加到 new_files_数组中。

DeleteFile 函数则是把参数指定的文件添加到 deleted_files 中；

SetCompactPointer 函数把{level, key}指定的 compact 点加入到 compact_pointers_中。

执行序列化和发序列化的是 Decode 和 Encode 函数，根据这些代码，我们可以了解 Manifest 文件的存储格式。序列化函数逻辑都很直观，不详细说了。

9.4.3 Manifest 文件格式

前面说过 Manifest 文件记录了 leveldb 的管理元信息，这些元信息到底都包含哪些内容呢？下面就来一一列示。

首先是使用的 comparator 名、log 编号、前一个 log 编号、下一个文件编号、上一个序列号。这些都是日志、sstable 文件使用到的重要信息，这些字段不一定必然存在。

Leveldb 在写入每个字段之前，都会先写入一个 varint 型数字来标记后面的字段类型。在读取时，先读取此字段，根据类型解析后面的信息。一共有 9 种类型：

kComparator = 1, kLogNumber = 2, kNextFileNumber = 3, kLastSequence = 4,

kCompactPointer = 5, kDeletedFile = 6, kNewFile = 7, kPrevLogNumber = 9

// 8 was used for large value refs

其中 8 另有它用。

其次是 compact_pointers_，可能有多个，写入格式为{kCompactPointer, level, internal key}。

其后是 deleted_files_，可能有多个，格式为{kDeletedFile, level, file number}。

最后是 new_files_，可能有多个，格式为

{kNewFile, level, file number, file size, min key, max key}。

对于版本间变动它是新加的文件集合，对于 MANIFEST 快照是该版本包含的所有 sstable 文件集合。

一张图表示一下，如图 9.3-1 所示。

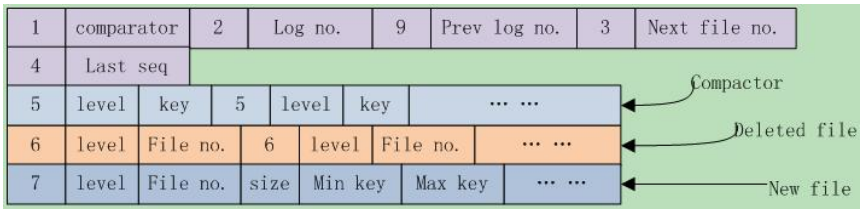


图 9.3-1

其中的数字都是 varint 存储格式，string 都是以 varint 指明其长度，后面跟实际的字符串内容。

9.5 DB 接口

9.5.1 接口函数

除了 DB 类， leveldb 还导出了 C 语言风格的接口：接口和实现在 c.h&c.cc，它其实是对 leveldb::DB 的一层封装。

DB 是一个持久化的有序 map{key, value}，它是线程安全的。DB 只是一个虚基类，下面来看看其接口：

首先是一个静态函数，打开一个 db，成功返回 OK，打开的 db 指针保存在*dbptr 中，用完后，调用者需要调用 delete *dbptr 删除之。

static Status Open(const Options& options, const std::string&name, DB** dbptr);

下面几个是纯虚函数，最后还有两个全局函数，为何不像 Open 一样作为静态函数呢。

注：在几个更新接口中，可考虑设置 options.sync = true。另外，虽然是纯虚函数，但是 leveldb 还是提供了缺省的实现。

```
1.   // 设置 db 项{key, value}
2.   virtual Status Put(const WriteOptions& options, const Slice&key, const Sli
   ce& value) = 0;
3.   // 在 db 中删除"key"，key 不存在依然返回成功
4.   virtual Status Delete(const WriteOptions& options, const Slice&key) = 0;
5.   // 更新操作
6.   virtual Status Write(const WriteOptions& options, WriteBatch*updates) = 0;

7.   // 获取操作，如果 db 中有”key”项则返回结果，没有就返回 Status::IsNotFound()
8.   virtual Status Get(const ReadOptions& options, const Slice& key,std::strin
   g* value) = 0;
9.   // 返回 heap 分配的 iterator，访问 db 的内容，返回的 iterator 的位置是 invalid 的
10.  // 在使用之前，调用者必须先调用 Seek。
11.  virtual Iterator* NewIterator(const ReadOptions& options) = 0;
12.  // 返回当前 db 状态的 handle，和 handle 一起创建的 Iterator 看到的都是
13.  // 当前 db 状态的稳定快照。不再使用时，应该调用 ReleaseSnapshot(result)
14.  virtual const Snapshot* GetSnapshot() = 0;
15.
16.  // 释放获取的 db 快照
17.  virtual voidReleaseSnapshot(const Snapshot* snapshot) = 0;
18.  // 借此方法 DB 实现可以展现它们的属性状态. 如果"property" 是合法的，
19.  // 设置"*value"为属性的当前状态值并返回 true，否则返回 false.
20.  // 合法属性名包括：
21.  //
22.  //  >"leveldb.num-files-at-level<N>"- 返回 level <N>的文件个数，
23.  //     <N> 是 level 数的 ASCII 值 (e.g. "0").
```

```
24.   //  >"leveldb.stats" - 返回描述 db 内部操作统计的多行 string
25.   //  >"leveldb.sstables" - 返回一个多行 string，描述构成 db 内容的所有 sstable
26.   virtual bool GetProperty(constSlice& property, std::string* value) = 0;
27.   //"sizes[i]"保存的是"[range[i].start.. range[i].limit)"中的 key 使用的文件空
    间.
28.   // 注：返回的是文件系统的使用空间大概值，
29.   //      如果用户数据以 10 倍压缩，那么返回值就是对应用户数据的 1/10
30.   //      结果可能不包含最近写入的数据大小.
31.   virtual voidGetApproximateSizes(const Range* range, int n, uint64_t* sizes
    ) = 0;
32.   // Compactkey 范围[*begin,*end]的底层存储，删除和被覆盖的版本将会被抛弃
33.   // 数据会被重新组织，以减少访问开销
34.   // 注：那些不了解底层实现的用户不应该调用该方法。
35.   //begin==NULL 被当作 db 中所有 key 之前的 key.
36.   //end==NULL 被当作 db 中所有 key 之后的 key.
37.   // 所以下面的调用将会 compact 整个 db:
38.   //    db->CompactRange(NULL, NULL);
39.   virtual void CompactRange(constSlice* begin, const Slice* end) = 0;
40.
41. // 最后是两个全局函数--删除和修复 DB
42. // 要小心，该方法将删除指定 db 的所有内容
43. Status DestroyDB(const std::string& name, const Options&options);
44. // 如果 db 不能打开了，你可能调用该方法尝试纠正尽可能多的数据
45. // 可能会丢失数据，所以调用时要小心
46. Status RepairDB(const std::string& dbname, const Options&options);
```

9.5.2 类图

这里又会设计到几个功能类，如图 9.5-1 所示。此外还有前面我们讲过的几大组件：操作日志的读写类、内存 MemTable 类、InternalFilterPolicy 类、Internal Key 比较类、以及 sstable 的读取构建类。如图 9.5-2 所示。

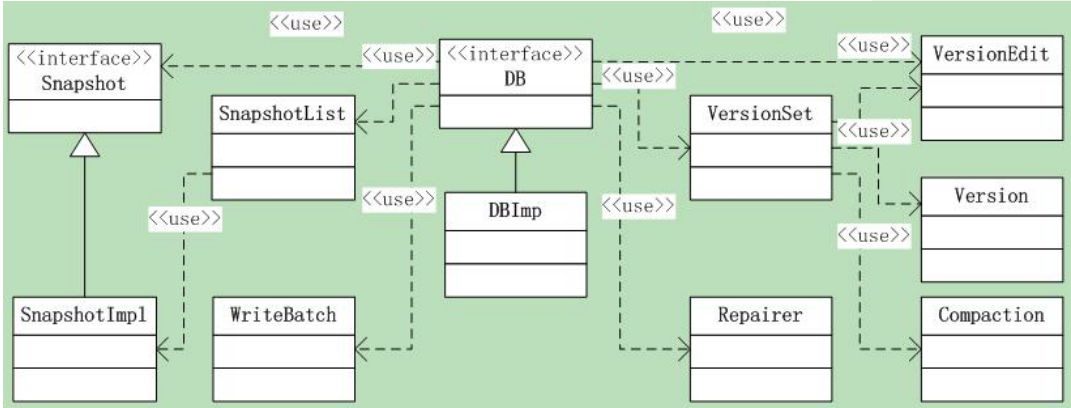


图 9.5-1

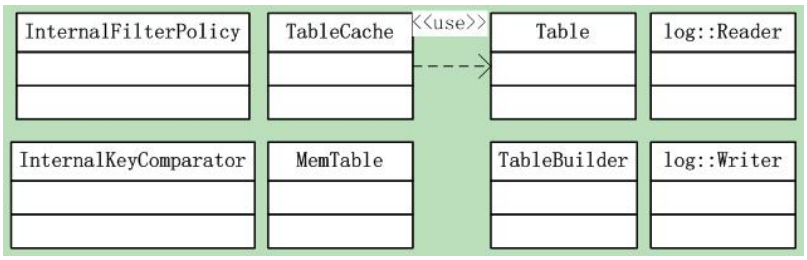


图 9.5-2

这里涉及的类很多，snapshot 是内存快照，Version 和 VersionSet 类。

9.6 DBImpl 类

在向下继续之前，有必要先了解下 DBImpl 这个具体的实现类。主要是它的成员变量，这说明了它都利用了哪些组件。

整篇代码里面，这算是一个庞然大物了。现在只是先打第一个照面吧，后面的路还很长，先来看看类成员。

[cpp] view plaincopy

```cpp
//== 第一组，他们在构造函数中初始化后将不再改变。其中，InternalKeyComparator 和 InternalFilterPolicy 已经分别在 Memtable 和 FilterPolicy 中分析过。
  Env* const env_; // 环境，封装了系统相关的文件操作、线程等等
  const InternalKeyComparatorinternal_comparator_;  // key comparator
  const InternalFilterPolicyinternal_filter_policy_;      // filter policy
  const Options options_;  //options_.comparator == &internal_comparator_
  bool owns_info_log_;
```

```
7.    bool owns_cache_;
8.    const std::string dbname_;
9.  //== 第二组，只有两个。
10.   TableCache* table_cache_; // Table cache，线程安全的
11.   FileLock* db_lock_;// 锁 db 文件，persistent state，直到 leveldb 进程结束
12. //== 第三组，被 mutex_包含的状态和成员
13.   port::Mutex mutex_; // 互斥锁
14.   port::AtomicPointershutting_down_;
15.   port::CondVar bg_cv_;    // 在 background work 结束时激发
16.   MemTable* mem_;
17.   MemTable* imm_;       // Memtablebeing compacted
18.   port::AtomicPointerhas_imm_;  // BGthread 用来检查是否是非 NULL 的 imm_
19.   // 这三个是 log 相关的
20.   WritableFile* logfile_;     // log 文件
21.   uint64_t logfile_number_; // log 文件编号
22.   log::Writer* log_;        // log writer
23. //== 第四组，没有规律
24.   std::deque<Writer*>writers_; // writers 队列.
25.   WriteBatch* tmp_batch_;
26.   SnapshotList snapshots_; //snapshot 列表
27.   // Setof table files to protect from deletion because they are
28.   // part ofongoing compactions.
29.   std::set<uint64_t>pending_outputs_; // 待 copact 的文件列表，保护以防误删
30.   bool bg_compaction_scheduled_; // 是否有后台 compaction 在调度或者运行?
31.   Status bg_error_; // paranoid mode 下是否有后台错误?
32.   ManualCompaction*manual_compaction_; // 手动 compaction 信息
33.   CompactionStatsstats_[config::kNumLevels]; // compaction 状态
34.   VersionSet* versions_; // 多版本 DB 文件，又一个庞然大物
```

10 Version 分析

先来分析 leveldb 对单版本的 sstable 文件管理，主要集中在 Version 类中。前面的 10.4 节已经说明了 Version 类的功能和成员，这里分析其函数接口和代码实现。
Version 不会修改其管理的 sstable 文件，只有读取操作。

10.1 Version 接口

先来看看 Version 类的接口函数，接下来再一一分析。

```
1.  // 追加一系列 iterator 到 @*iters 中，将在 merge 到一起时生成该 Version 的内容
2.  // 要求: Version 已经保存了(见 VersionSet::SaveTo)
3.  void AddIterators(constReadOptions&, std::vector<Iterator*>* iters);
4.
```

```
// 给定@key 查找 value，如果找到保存在@*val 并返回 OK。
// 否则返回 non-OK，设置@ *stats.
// 要求：没有 hold lock
struct GetStats {
  FileMetaData* seek_file;
  int seek_file_level;
};
Status Get(const ReadOptions&, const LookupKey& key, std::string* val,
GetStats* stats);

// 把@stats 加入到当前状态中，如果需要触发新的 compaction 返回 true
// 要求：hold lock
bool UpdateStats(const GetStats& stats);
void GetOverlappingInputs(int level,
    const InternalKey* begin,         // NULL 指在所有 key 之前
    const InternalKey* end,           // NULL 指在所有 key 之后
    std::vector<FileMetaData*>* inputs);

// 如果指定 level 中的某些文件和[*smallest_user_key,*largest_user_key]有重合就返
回 true。
// @smallest_user_key==NULL 表示比 DB 中所有 key 都小的 key.
// @largest_user_key==NULL 表示比 DB 中所有 key 都大的 key.
bool OverlapInLevel(int level,const Slice*
smallest_user_key, const Slice* largest_user_key);

// 返回我们应该在哪个 level 上放置新的 memtable compaction，
// 该 compaction 覆盖了范围[smallest_user_key,largest_user_key].
int PickLevelForMemTableOutput(const Slice& smallest_user_key,
                               const Slice& largest_user_key);

int NumFiles(int level) const {return files_[level].size(); } // 指定 level 的
sstable 个数
```

10.2 Version::AddIterators()

该函数最终在 DB::NewIterators()接口中被调用，调用层次为：

DBImpl::NewIterator()->DBImpl::NewInternalIterator()->Version::AddIterators()。

函数功能是为该 Version 中的所有 sstable 都创建一个 Two Level Iterator，以遍历 sstable 的内容。

对于 level=0 级别的 sstable 文件，直接通过 TableCache::NewIterator()接口创建，这会直接载入 sstable 文件到内存 cache 中。

对于 level>0 级别的 sstable 文件，通过函数 NewTwoLevelIterator()创建一个 TwoLevelIterator，这就使用了 lazy open 的机制。

下面来分析函数代码：

- S1 对于 level=0 级别的 sstable 文件，直接装入 cache，level0 的 sstable 文件可能有重合，需要 merge。

```
for (size_t i = 0; i <files_[0].size(); i++) {
 iters->push_back(vset_->table_cache_->NewIterator(// versionset::table_cache_
          options,files_[0][i]->number, files_[0][i]->file_size));
}
```

- S2 对于 level>0 级别的 sstable 文件，lazy open 机制，它们不会有重叠。

```
for (int ll = 1; ll <config::kNumLevels; ll++) {
  if(!files_[ll].empty()) iters->push_back(NewConcatenatingIterator(options, level));
}
```

函数 NewConcatenatingIterator()直接返回一个 TwoLevelIterator 对象：

return NewTwoLevelIterator(new LevelFileNumIterator(vset_->icmp_,&files_[level]), &GetFileIterator,vset_->table_cache_, options);

其第一级 iterator 是一个 LevelFileNumIterator，第二级的迭代函数是 GetFileIterator，下面就来分别分析之。

GetFileIterator 是一个静态函数，很简单，直接返回 TableCache::NewIterator()。函数声明为：

static Iterator* GetFileIterator(void* arg,const ReadOptions& options, constSlice& file_value)

```
TableCache* cache =reinterpret_cast<TableCache*>(arg);
if (file_value.size() != 16) { // 错误
  return NewErrorIterator(Status::Corruption("xxx"));
} else {
  return cache->NewIterator(options,
                DecodeFixed64(file_value.data()), // filenumber
                DecodeFixed64(file_value.data() + 8)); // filesize
}
```

这里的 file_value 是取自于 LevelFileNumIterator 的 value，它的 value()函数把 file number 和 size 以 Fixed 8byte 的方式压缩成一个 Slice 对象并返回。

10.3 Version::LevelFileNumIterator 类

这也是一个继承了 Iterator 的子类，一个内部 Iterator。给定一个 version/level 对，生成该 level 内的文件信息。对于给定的 entry，key()返回的是文件中所包含的最大的 key，value()返回的是|file number(8 bytes)|file size(8 bytes)|串。

它的构造函数接受两个参数：InternalKeyComparator&，用于 key 的比较；vector<FileMetaData*>*，指向 version 的所有 sstable 文件列表。

```
LevelFileNumIterator(const InternalKeyComparator& icmp,
                     const std::vector<FileMetaData*>* flist)
                     : icmp_(icmp), flist_(flist),index_(flist->size()) {} // Marks as invalid
```

来看看其接口实现，不罗嗦，全部都列出来。

Valid 函数、SeekToxx 和 Next/Prev 函数都很简单，毕竟容器是一个 vector。Seek 函数调用了 FindFile，这个函数后面会分析。

```
1.  virtual void Seek(constSlice& target) { index_ = FindFile(icmp_, *flist_, ta
    rget);}
2.  virtual void SeekToFirst() {index_ = 0; }
3.  virtual void SeekToLast() {index_ = flist_->empty() ? 0 : flist_->size() - 1
    ;}
4.  virtual void Next() {
5.    assert(Valid());
6.    index_++;
7.  }
8.
9.  virtual void Prev() {
10.   assert(Valid());
11.   if (index_ == 0) index_ =flist_->size(); // Marks as invalid
12.   else index_--;
13. }
14.
15. Slice key() const {
16.   assert(Valid());
17.   return(*flist_)[index_]->largest.Encode(); // 返回当前 sstable 包含的
    largest key
18. }
19.
20. Slice value() const { // 根据|number|size|的格式 Fixed int 压缩
21.   assert(Valid());
22.   EncodeFixed64(value_buf_,(*flist_)[index_]->number);
23.   EncodeFixed64(value_buf_+8,(*flist_)[index_]->file_size);
24.   return Slice(value_buf_,sizeof(value_buf_));
25. }
```

来看 FindFile，这其实是一个二分查找函数，因为传入的 sstable 文件列表是有序的，因此可以使用二分查找算法。就不再列出代码了。

10.4 Version::Get()

查找函数，直接在 DBImpl::Get()中被调用，函数原型为：
Status Version::Get(const ReadOptions& options, constLookupKey& k, std::string* value, GetStats* stats)

如果本次 Get 不止 seek 了一个文件（仅会发生在 level 0 的情况），就将搜索的第一个文件保存在 stats 中。如果 stat 有数据返回，表明本次读取在搜索到包含 key 的 sstable 文件之前，还做了其它无谓的搜索。这个结果将用在 UpdateStats()中。

这个函数逻辑还是有些复杂的，来看看代码。

- S1 首先，取得必要的信息，初始化几个临时变量

```
1.  Slice ikey = k.internal_key();
2.  Slice user_key = k.user_key();
3.  const Comparator* ucmp =vset_->icmp_.user_comparator();
4.  Status s;
5.  stats->seek_file = NULL;
6.  stats->seek_file_level = -1;
7.  FileMetaData* last_file_read =NULL; // 在找到>1 个文件时，读取时记录上一个
8.  int last_file_read_level = -1;        // 这仅发生在 level 0 的情况下
9.  std::vector<FileMetaData*> tmp;
10. FileMetaData* tmp2;
```

- S2 从 0 开始遍历所有的 level，依次查找。因为 entry 不会跨越 level，因此如果在某个 level 中找到了 entry，那么就无需在后面的 level 中查找了。

```
1.  for (int level = 0; level <config::kNumLevels; level++) {
2.    size_t num_files = files_[level].size();
3.    if (num_files == 0) continue; // 本层没有文件，则直接跳过
4.     // 取得 level 下的所有 sstable 文件列表，搜索本层
5.    FileMetaData* const* files = &files_[level][0];
```

后面的所有逻辑都在 for 循环体中。

- S3 遍历 level 下的 sstable 文件列表，搜索，注意对于 level=0 和>0 的 sstable 文件的处理，由于 level 0 文件之间的 key 可能有重叠，因此处理逻辑有别于>0 的 level。
  - S3.1 对于 level 0，文件可能有重叠，找到所有和 user_key 有重叠的文件，然后根据时间顺序从最新的文件依次处理。

```
1.  tmp.reserve(num_files);
2.  for (uint32_t i = 0; i <num_files; i++) { // 遍历 level 0 下的所有 sstable 文
    件
3.    FileMetaData* f =files[i];
4.    if(ucmp->Compare(user_key, f->smallest.user_key()) >= 0 &&
5.    ucmp->Compare(user_key, f->largest.user_key()) <= 0)
6.    tmp.push_back(f); // sstable 文件有 user_key 有重叠
```

```
}
if (tmp.empty()) continue;
std::sort(tmp.begin(),tmp.end(), NewestFirst); // 排序
files = &tmp[0]; num_files= tmp.size();// 指向 tmp 指针和大小
```

- S3.2 对于 level>0，leveldb 保证 sstable 文件之间不会有重叠，所以处理逻辑有别于 level 0，直接根据 ikey 定位到 sstable 文件即可。

```
//二分查找，找到第一个 largest key >=ikey 的 file index
uint32_t index =FindFile(vset_->icmp_, files_[level], ikey);
if (index >= num_files) { // 未找到，文件不存在
  files = NULL;  num_files = 0;
} else {
  tmp2 = files[index];
  if(ucmp->Compare(user_key, tmp2->smallest.user_key()) < 0) {
    // 找到的文件其所有 key 都大于 user_key，等于文件不存在
    files = NULL;  num_files = 0;
  } else {
    files = &tmp2;  num_files = 1;
  }
}
```

- S4 遍历找到的文件，存在 files 中，其个数为 num_files。
  for (uint32_t i = 0; i <num_files; ++i) {

后面的逻辑都在这一层循环中，只要在某个文件中找到了 k/v 对，就跳出 for 循环。

- S4.1 如果本次读取不止搜索了一个文件，记录之，这仅会发生在 level 0 的情况下。

```
if(last_file_read != NULL && stats->seek_file == NULL) {
   // 本次读取不止 seek 了一个文件，记录第一个
   stats->seek_file =last_file_read;
   stats->seek_file_level= last_file_read_level;
 }
 FileMetaData* f = files[i];
 last_file_read = f; // 记录本次读取的 level 和 file
 last_file_read_level =level;
```

- S4.2 调用 TableCache::Get()尝试获取{ikey, value}，如果返回 OK 则进入 S4.3，否则直接返回，传递的回调函数是 SaveValue()。

```
Saver saver; // 初始化 saver
saver.state = kNotFound;
saver.ucmp = ucmp;
saver.user_key = user_key;
```

```
saver.value = value;
s = vset_->table_cache_->Get(options,f->number, f->file_size,
                             ikey, &saver, SaveValue);
if (!s.ok()) return s;
```

- S4.3 根据 saver 的状态判断，如果是 Not Found 则向下搜索下一个更早的 sstable 文件，其它值则返回。

```
switch (saver.state) {
  case kNotFound: break; // 继续搜索下一个更早的 sstable 文件
  case kFound:  return s; // 找到
  case kDeleted: // 已删除
    s =Status::NotFound(Slice());  // 为了效率，使用空的错误字符串
    return s;
  case kCorrupt: // 数据损坏
    s =Status::Corruption("corrupted key for ", user_key);
    return s;
}
```

以上就是 Version::Get()的代码逻辑，如果 level 0 的 sstable 文件太多的话，会影响读取速度，这也是为什么进行 compaction 的原因。

另外，还有一个传递给 TableCache::Get()的 saver 函数，下面就来简单分析下。这是一个静态函数：static void SaveValue(void* arg,const Slice& ikey, const Slice& v)。它内部使用了结构体 Saver：

```
struct Saver {
    SaverState state;
    const Comparator* ucmp; // user key 比较器
    Slice user_key;
    std::string* value;
};
```

函数 **SaveValue** 的逻辑很简单，首先解析 Table 传入的 InternalKey，然后根据指定的 Comparator 判断 user key 是否是要查找的 user key。如果是并且 type 是 kTypeValue，则设置到 Saver::*value 中，并返回 kFound，否则返回 kDeleted。代码如下：

```
Saver* s =reinterpret_cast<Saver*>(arg);
ParsedInternalKey parsed_key; // 解析 ikey 到 ParsedInternalKey
if (!ParseInternalKey(ikey,&parsed_key)) s->state = kCorrupt; // 解析失败
else {
  if(s->ucmp->Compare(parsed_key.user_key, s->user_key) == 0) { // 比较 user key
    s->state =(parsed_key.type == kTypeValue) ? kFound : kDeleted;
    if (s->state == kFound)s->value->assign(v.data(), v.size()); // 找到，保存结果
  }
```

```
}
```

下面要分析的几个函数，或多或少都和 compaction 相关。

10.5 Version::UpdateStats()

当 Get 操作直接搜寻 memtable 没有命中时，就需要调用 Version::Get()函数从磁盘 load 数据文件并查找。如果此次 Get 不止 seek 了一个文件，就记录第一个文件到 stat 并返回。其后 leveldb 就会调用 UpdateStats(stat)。

Stat 表明在指定 key range 查找 key 时，都要先 seek 此文件，才能在后续的 sstable 文件中找到 key。

该函数是将 stat 记录的 sstable 文件的 allowed_seeks 减 1，减到 0 就执行 compaction。也就是说如果文件被 seek 的次数超过了限制，表明读取效率已经很低，需要执行 compaction 了。所以说 allowed_seeks 是对 compaction 流程的有一个优化。

函数声明：bool Version::UpdateStats(const GetStats& stats)

函数逻辑很简单：

```
FileMetaData* f =stats.seek_file;
if (f != NULL) {
  f->allowed_seeks--;
  if (f->allowed_seeks <=0 && file_to_compact_ == NULL) {
    file_to_compact_ = f;
    file_to_compact_level_ =stats.seek_file_level;
    return true;
  }
}
return false;
```

变量 allowed_seeks 的值在 sstable 文件加入到 version 时确定，也就是后面将遇到的 VersionSet::Builder::Apply()函数。

10.6 Version::GetOverlappingInputs()

它在指定 level 中找出和[begin, end]有重合的 sstable 文件，函数声明为：

void Version::GetOverlappingInputs(int level,

const InternalKey* begin, constInternalKey* end, std::vector<FileMetaData*>* inputs)

要注意的是，对于 level0，由于文件可能有重合，其处理具有特殊性。当在 level 0 中找到有 sstable 文件和[begin, end]重合时，会相应的将 begin/end 扩展到文件的 min key/max key，然后重新开始搜索。

了解了功能，下面分析函数实现代码，逻辑还是很直观的。

- S1 首先根据参数初始化查找变量。

```
inputs->clear();
Slice user_begin, user_end;
```

```
3.  if (begin != NULL) user_begin =begin->user_key();
4.  if (end != NULL)  user_end = end->user_key();
5.  const Comparator* user_cmp =vset_->icmp_.user_comparator();
```

- S2 遍历该层的 sstable 文件，比较 sstable 的{minkey,max key}和传入的[begin, end]，如果有重合就记录文件到@inputs 中，需要对 level 0 做特殊处理。

```
1.  for (size_t i = 0; i <files_[level].size(); ) {
2.    FileMetaData* f =files_[level][i++];
3.    const Slice file_start =f->smallest.user_key();
4.    const Slice file_limit =f->largest.user_key();
5.    if (begin != NULL &&user_cmp->Compare(file_limit, user_begin) < 0) {
6.      //"f" 中的 k/v 全部在指定范围之前; 跳过
7.    } else if (end != NULL&& user_cmp->Compare(file_start, user_end) > 0) {
8.      //"f" 中的 k/v 全部在指定范围之后; 跳过
9.    } else {
10.     inputs->push_back(f); // 有重合，记录
11.     if (level == 0) {
12.       // 对于 level 0，sstable 文件可能相互有重叠，所以要检查新加的文件
13.       // 是否范围更大，如果是则扩展范围重新开始搜索
14.       if (begin != NULL&& user_cmp->Compare(file_start, user_begin) < 0) {
15.         user_begin = file_start;
16.         inputs->clear();
17.         i = 0;
18.       } else if (end != NULL&& user_cmp->Compare(file_limit, user_end) > 0)
    {
19.         user_end = file_limit;
20.         inputs->clear();
21.         i = 0;
22.       }
23.     }
24.   }
25. }
```

10.7 Version::OverlapInLevel()

检查是否和指定 level 的文件有重合，该函数直接调用了 SomeFileOverlapsRange()，这两个函数的声明为：

```
bool Version::OverlapInLevel(int level,
                             const Slice*smallest_user_key, const Slice* largest_user_key){
    return SomeFileOverlapsRange(vset_->icmp_,(level > 0), files_[level],
                             smallest_user_key, largest_user_key);
}
bool SomeFileOverlapsRange(const InternalKeyComparator& icmp,
```

bool disjoint_sorted_files,const std::vector<FileMetaData*>& files,
const Slice*smallest_user_key, const Slice* largest_user_key);
所以下面直接分析 SomeFileOverlapsRange()函数的逻辑，代码很直观。
disjoint_sorted_files=true，表明文件集合是互不相交、有序的，对于乱序的、可能有交集的文件集合，需要逐个查找，找到有重合的就返回 true；对于有序、互不相交的文件集合，直接执行二分查找。

```
// S1 乱序、可能相交的文件集合，依次查找
    for (size_t i = 0; i <files.size(); i++) {
      const FileMetaData* f =files[i];
      if(AfterFile(ucmp,smallest_user_key, f) || BeforeFile(ucmp, largest_user_key, f)){
      } else return true;  // 有重合
    }
    return false; // 没找到
// S2 有序&互不相交，直接二分查找
  uint32_t index = 0;
  if (smallest_user_key != NULL) {
    // Findthe earliest possible internal key smallest_user_key
    InternalKeysmall(*smallest_user_key, kMaxSequenceNumber,kValueTypeForSeek);
    index = FindFile(icmp, files,small.Encode());
  }
  if (index >= files.size())return false; // 不存在比 smallest_user_key 小的 key
  return !BeforeFile(ucmp,largest_user_key, files[index]); //保证在 largest_user_key 之后
```

上面的逻辑使用到了 AfterFile()和 BeforeFile()两个辅助函数，都很简单。

```
static bool AfterFile(const Comparator* ucmp,
                      const Slice* user_key, constFileMetaData* f) {
  return (user_key!=NULL&& ucmp->Compare(*user_key, f->largest.user_key())>0);
}
static bool BeforeFile(const Comparator* ucmp,
                      constSlice* user_key, const FileMetaData* f) {
  return (user_key!=NULL&& ucmp->Compare(*user_key, f->smallest.user_key())<0);
}
```

10.8 Version::PickLevelForMemTableOutput()

函数返回我们应该在哪个 level 上放置新的 memtable compaction，这个 compaction

覆盖了范围[smallest_user_key,largest_user_key]。
该函数的调用链为：
DBImpl::RecoverLogFile/DBImpl::CompactMemTable -> DBImpl::WriteLevel0Table->Version::PickLevelForMemTableOutput
函数声明如下：
int Version::PickLevelForMemTableOutput(const Slice& smallest_user_key, constSlice& largest_user_key)
如果 level 0 没有找到重合就向下一层找，最大查找层次为 kMaxMemCompactLevel = 2。
如果在 level 0or1 找到了重合，就返回 level 0。否则查找 level 2，如果 level 2 有重合就返回 level 1，否则返回 level 2。

函数实现：

```
1.  int level = 0;
2.  if (!OverlapInLevel(0,&smallest_user_key, &largest_user_key)) { //level 0无重合
3.    // 如果下一层没有重叠，就压到下一层，
4.    // andthe #bytes overlapping in the level after that are limited.
5.    InternalKeystart(smallest_user_key, kMaxSequenceNumber, kValueTypeForSeek);
6.    InternalKeylimit(largest_user_key, 0, static_cast<ValueType>(0));
7.    std::vector<FileMetaData*> overlaps;
8.    while (level <config::kMaxMemCompactLevel) {
9.      if (OverlapInLevel(level +1, &smallest_user_key, &largest_user_key))
10.        break; // 检查 level + 1 层，有重叠就跳出循环
11.     GetOverlappingInputs(level +2, &start, &limit, &overlaps); // 没理解这个调用
12.     const int64_t sum =TotalFileSize(overlaps);
13.     if (sum >kMaxGrandParentOverlapBytes) break;
14.     level++;
15.   }
16. }
17. return level;
```

这个函数在整个 compaction 逻辑中的作用在分析 DBImpl 时再来结合整个流程分析，现在只需要了解它找到一个 level 存放新的 compaction 就行了。

如果返回 level = 0，表明在 level 0 或者 1 和指定的 range 有重叠；如果返回 1，表明在 level2 和指定的 range 有重叠；否则就返回 2（kMaxMemCompactLevel）。

也就是说在 compactmemtable 的时候，写入的 sstable 文件不一定总是在 level 0，如果比较顺利，没有重合的，它可能会写到 level1 或者 level2 中。

10.9 小结

Version 是管理某个版本的所有 sstable 的类，就其导出接口而言，无非是遍历 sstable，查找 k/v。以及为 compaction 做些事情，给定 range，检查重叠情况。

而它不会修改它管理的 sstable 这些文件，对这些文件而言它是只读操作接口。

11 VersionSet 分析

Version 之后就是 VersionSet，它并不是 Version 的简单集合，还肩负了不少的处理逻辑。这里的分析不涉及到 compaction 相关的部分，这部分会单独分析。包括 log 等各种编号计数器，compaction 点的管理等等。

11.1 VersionSet 接口

1. 首先是构造函数，VersionSet 会使用到 TableCache，这个是调用者传入的。TableCache 用于 Get k/v 操作。

VersionSet(const std::string& dbname, const Options* options, TableCache*table_cache, const InternalKeyComparator*)

VersionSet 的构造函数很简单，除了根据参数初始化，还有两个地方值得注意：

- N1 next_file_number_从 2 开始；
- N2 创建新的 Version 并加入到 Version 链表中，并设置 CURRENT=新创建 version；

其它的数字初始化为 0，指针初始化为 NULL。

2. 恢复函数，从磁盘恢复最后保存的元信息

Status **Recover**();

3. 标记指定的文件编号已经被使用了

void **MarkFileNumberUsed**(uint64_t number)

逻辑很简单，就是根据编号更新文件编号计数器：

if (next_file_number_ <= number) next_file_number_ = number + 1;

4. 在 current version 上应用指定的 VersionEdit，生成新的 MANIFEST 信息，保存到磁盘上，并用作 current version。

要求：没有其它线程并发调用；要用于 mu；Status **LogAndApply**(VersionEdit* edit, port::Mutex* mu) EXCLUSIVE_LOCKS_REQUIRED(mu)

5. 对于@v 中的@key，返回 db 中的大概位置

uint64_t **ApproximateOffsetOf**(Version* v, const InternalKey& key);

6. 其它一些简单接口，信息获取或者设置，如下：

```
1.  Version* current() const { return current_; } //返回 current version
2.  // 当前的 MANIFEST 文件号
3.  uint64_t ManifestFileNumber() const { return manifest_file_number_; }
4.  uint64_t NewFileNumber() { return next_file_number_++; } // 分配并返回新的文件
    编号
5.  uint64_t LogNumber() const { return log_number_; } // 返回当前 log 文件编号
6.  // 返回正在 compact 的 log 文件编号，如果没有返回 0
7.  uint64_t PrevLogNumber() const { return prev_log_number_; }
8.  // 获取、设置 last sequence，set 时不能后退
9.  uint64_t LastSequence() const { return last_sequence_; }
```

```
void SetLastSequence(uint64_t s) {
    assert(s >=last_sequence_);
    last_sequence_ = s;
}
// 返回指定 level 中所有 sstable 文件大小的和
int64_t NumLevelBytes(int level) const;
// 返回指定 level 的文件个数
int NumLevelFiles(int level) const;
// 重用@file_number，限制很严格：@file_number 必须是最后分配的那个
// 要求：@file_number 是 NewFileNumber()返回的.
void ReuseFileNumber(uint64_t file_number) {
  if (next_file_number_ ==file_number + 1) next_file_number_ = file_number;

}
// 对于所有 level>0，遍历文件，找到和下一层文件的重叠数据的最大值(in bytes)
// 这个就是 Version::GetOverlappingInputs()函数的简单应用
int64_t MaxNextLevelOverlappingBytes();
// 获取函数，把所有 version 的所有 level 的文件加入到@live 中
void AddLiveFiles(std::set<uint64_t>* live)
// 返回一个可读的单行信息——每个 level 的文件数，保存在*scratch 中
struct LevelSummaryStorage {char buffer[100]; };
const char* LevelSummary(LevelSummaryStorage* scratch) const;
```

下面就来分析这两个接口 Recover、LogAndApply 以及 ApproximateOffsetOf。

11.2 Recover()

对于 VersionSet 而言，Recover 就是根据 CURRENT 指定的 MANIFEST，读取 db 元信息到 this VersionSet 对象的第二组成员中（db 元信息）。这是 9.3 介绍的 Recovery 流程的开始部分。

11.2.1 函数流程

下面就来分析其具体逻辑。

- S1 读取 CURRENT 文件，获得最新的 MANIFEST 文件名，根据文件名打开 MANIFEST 文件。CURRENT 文件以\n 结尾，读取后需要 trim 下。

```
std::string current; // MANIFEST 文件名
ReadFileToString(env_, CurrentFileName(dbname_), &current)
std::string dscname = dbname_ + "/" + current;
SequentialFile* file;
env_->NewSequentialFile(dscname, &file);
```

- S2 读取 MANIFEST 内容，MANIFEST 是以 log 的方式写入的，因此这里调用的是 log::Reader 来读取。然后调用 VersionEdit::DecodeFrom，从内容解析出 VersionEdit 对象，并将 VersionEdit 记录的改动应用到 versionset 中。读取 MANIFEST 中的 log number, prev log number, nextfile number, last sequence。

```
Builder builder(this, current_);
while (reader.ReadRecord(&record, &scratch) && s.ok()) {
      VersionEdit edit;
      s = edit.DecodeFrom(record);
      if (s.ok())builder.Apply(&edit);
      if (edit.has_log_number_) { // log number, file number, …逐个判断
        log_number =edit.log_number_;
        have_log_number = true;
      }
      … …
}
```

- S3 将读取到的 log number, prev log number 标记为已使用。

MarkFileNumberUsed(prev_log_number);
MarkFileNumberUsed(log_number);

- S4 最后，如果一切顺利就创建新的 Version，并应用读取的几个 number。

```
if (s.ok()) {
  Version* v = newVersion(this);
  builder.SaveTo(v);
  // 安装恢复的 version
  Finalize(v);
  AppendVersion(v);
  manifest_file_number_ =next_file;
  next_file_number_ = next_file+ 1;
  last_sequence_ = last_sequence;
  log_number_ = log_number;
  prev_log_number_ =prev_log_number;
}
```

Finalize(v)和 AppendVersion(v)用来安装并使用 version v，在 AppendVersion 函数中会将 current version 设置为 v。下面就来分别分析这两个函数。

11.2.2 Finalize()

函数声明：void **Finalize**(Version*v)；
该函数依照规则为下次的 compaction 计算出最适用的 level，对于 level 0 和>0 需要分别对待，逻辑如下。

- S1 对于 level 0 以文件个数计算，kL0_CompactionTrigger 默认配置为 4。

score =v->files_[level].size()/static_cast<double>(config::kL0_CompactionTrigger);

- S2 对于 level>0，根据 level 内的文件总大小计算

const uint64_t level_bytes = TotalFileSize(v->files_[level]);
score = static_cast<double>(level_bytes) /MaxBytesForLevel(level);

- S3 最后把计算结果保存到 v 的两个成员 compaction_level_和 compaction_score_中。

其中函数 MaxBytesForLevel 根据 level 返回其本层文件总大小的预定最大值。
计算规则为：1048576.0* level^10。
这里就有一个问题，为何 level0 和其它 level 计算方法不同，原因如下，这也是 leveldb 为 compaction 所做的另一个优化。
>1 对于较大的写缓存（write-buffer），做太多的 level 0 compaction 并不好
>2 每次 read 操作都要 merge level 0 的所有文件，因此我们不希望 level 0 有太多的小文件存在（比如写缓存太小，或者压缩比较高，或者覆盖/删除较多导致小文件太多）。

看起来这里的写缓存应该就是配置的操作 log 大小。

11.2.3 AppendVersion()

函数声明：void **AppendVersion**(Version*v)

把 v 加入到 versionset 中，并设置为 current version。并对老的 current version 执行 Uref()。

在双向循环链表中的位置在 dummy_versions_之前。

11.3 LogAndApply()

函数声明：Status **LogAndApply**(VersionEdit*edit, port::Mutex* mu)
前面接口小节中讲过其功能：在 current version 上 应用 指定的 VersionEdit，生成新的 MANIFEST 信息，保存到磁盘上，并用作 current version，故为 Log And Apply。
参数 edit 也会被函数修改。

11.3.1 函数流程

下面就来具体分析函数代码。

- S1 为 edit 设置 log number 等 4 个计数器。

```
1.  if (edit->has_log_number_) {
2.    assert(edit->log_number_>= log_number_);
3.    assert(edit->log_number_< next_file_number_);
4.  } else edit->SetLogNumber(log_number_);
5.  if(!edit->has_prev_log_number_) edit->SetPrevLogNumber(prev_log_number_);
6.  edit->SetNextFile(next_file_number_);
7.  edit->SetLastSequence(last_sequence_);
```

```
1.  要保证 edit 自己的 log number 是比较大的那个，否则就是致命错误。保证 edit 的
    log number 小于 next file number，否则就是致命错误-见 9.1 小节。
```

- S2 创建一个新的 Version v，并把新的 edit 变动保存到 v 中。

```
Version* v = new Version(this);
{
  Builder builder(this,current_);
  builder.Apply(edit);
  builder.SaveTo(v);
}
Finalize(v); //如前分析，只是为 v 计算执行 compaction 的最佳 level
```

- S3 如果 MANIFEST 文件指针不存在，就创建并初始化一个新的 MANIFEST 文件。这只会发生在第一次打开数据库时。这个 MANIFEST 文件保存了 current version 的快照。

```
std::string new_manifest_file;
Status s;
if (descriptor_log_ == NULL) {
  // 这里不需要 unlock *mu 因为我们只会在第一次调用 LogAndApply 时
  // 才走到这里(打开数据库时).
  assert(descriptor_file_ ==NULL); // 文件指针和 log::Writer 都应该是 NULL
  new_manifest_file =DescriptorFileName(dbname_, manifest_file_number_);
  edit->SetNextFile(next_file_number_);
  s =env_->NewWritableFile(new_manifest_file, &descriptor_file_);
  if (s.ok()) {
    descriptor_log_ = new log::Writer(descriptor_file_);
    s =WriteSnapshot(descriptor_log_); // 写入快照
  }
}
```

- S4 向 MANIFEST 写入一条新的 log，记录 current version 的信息。在文件写操作时 unlock 锁，写入完成后，再重新 lock，以防止浪费在长时间的 IO 操作上。

```
mu->Unlock();
if (s.ok()) {
  std::string record;
  edit->EncodeTo(&record);// 序列化 current version 信息
  s =descriptor_log_->AddRecord(record); // append 到 MANIFEST log 中
  if (s.ok()) s =descriptor_file_->Sync();
  if (!s.ok()) {
    Log(options_->info_log,"MANIFEST write: %s\n", s.ToString().c_str());
    if (ManifestContains(record)) { // 返回出错，其实确实写成功了
```

```
      Log(options_->info_log, "MANIFEST contains log record despiteerror ");

      s = Status::OK();
    }
  }
}
//如果刚才创建了一个 MANIFEST 文件，通过写一个指向它的 CURRENT 文件
//安装它；不需要再次检查 MANIFEST 是否出错，因为如果出错后面会删除它
if (s.ok() &&!new_manifest_file.empty()) {
  s = SetCurrentFile(env_,dbname_, manifest_file_number_);
}
mu->Lock();
```

- S5 安装这个新的 version

```
if (s.ok()) { // 安装这个 version
  AppendVersion(v);
  log_number_ =edit->log_number_;
  prev_log_number_ =edit->prev_log_number_;
} else { // 失败了，删除
  delete v;
  if(!new_manifest_file.empty()) {
    delete descriptor_log_;
    delete descriptor_file_;
    descriptor_log_ =descriptor_file_ = NULL;
    env_->DeleteFile(new_manifest_file);
  }
}
```

流程的 S4 中，函数会检查 MANIFEST 文件是否已经有了这条 record，那么什么时候会有呢？
主函数使用到了几个新的辅助函数 WriteSnapshot，ManifestContains 和 SetCurrentFile，下面就来分析。

11.3.2 WriteSnapshot()

函数声明：Status **WriteSnapshot**(log::Writer*log)
把 current version 保存到*log 中，信息包括 comparator 名字、compaction 点和各级 sstable 文件，函数逻辑很直观。

- S1 首先声明一个新的 VersionEdit edit；
- S2 设置 comparator：edit.SetComparatorName(icmp_.user_comparator()->Name());
- S3 遍历所有 level，根据 compact_pointer_[level]，设置 compaction 点：edit.SetCompactPointer(level, key);
- S4 遍历所有 level，根据 current_->files_，设置 sstable 文件集合：edit.AddFile(level, xxx)
- S5 根据序列化并 append 到 log（MANIFEST 文件）中；

std::string record;
edit.EncodeTo(&record);
returnlog->AddRecord(record);
以上就是 WriteSnapshot 的代码逻辑。

11.3.3 ManifestContains()

函数声明：bool **ManifestContains**(conststd::string& record)
如果当前 MANIFEST 包含指定的 record 就返回 true，来看看函数逻辑。

- S1 根据当前的 manifest_file_number_文件编号打开文件，创建 SequentialFile 对象
- S2 根据创建的 SequentialFile 对象创建 log::Reader，以读取文件
- S3 调用 log::Reader 的 ReadRecord 依次读取 record，如果和指定的 record 相同，就返回 true，没有相同的 record 就返回 false

SetCurrentFile 很简单，就是根据指定 manifest 文件编号，构造出 MANIFEST 文件名，并写入到 CURRENT 即可。

11.4 ApproximateOffsetOf()

函数声明：uint64_t **ApproximateOffsetOf**(Version* v, const InternalKey& ikey)
在指定的 version 中查找指定 key 的大概位置。

假设 version 中有 n 个 sstable 文件，并且落在了第 i 个 sstable 的 key 空间内，那么返回的位置= sstable1 文件大小+sstable2 文件大小 + … + sstable (i-1)文件大小
+ key 在 sstable i 中的大概偏移。

可分为两段逻辑。

1. 首先直接和 sstable 的 max key 作比较，如果 key > max key，直接跳过该文件，还记得 sstable 文件是有序排列的。

对于 level >0 的文件集合而言，如果如果 key < sstable 文件的 min key，则直接跳出循环，因为后续的 sstable 的 min key 肯定大于 key。

2. key 在 sstable i 中的大概偏移使用的是 Table:: ApproximateOffsetOf(target)接口，前面分析过，它返回的是 Table 中>= target 的 key 的位置。

VersionSet 的相关函数暂时分析到这里，compaction 部分后需单独分析。

11.5 VersionSet::Builder 类

Builder 是一个内部辅助类，其主要作用是：
1 把一个 MANIFEST 记录的元信息应用到版本管理器 VersionSet 中；
2 把当前的版本状态设置到一个 Version 对象中。

11.5.1 成员与构造

Builder 的 vset_与 base_都是调用者传入的，此外它还为 FileMetaData 定义了一个比较类 BySmallestKey，首先依照文件的 min key，小的在前；如果 min key 相等则 file number 小的在前。

```
1.   typedefstd::set<FileMetaData*, BySmallestKey> FileSet;
2.   struct LevelState { // 这个是记录添加和删除的文件
3.     std::set<uint64_t>deleted_files;
4.     FileSet* added_files; // 保证添加文件的顺序是有效定义的
5.   };
6.   VersionSet* vset_;
7.   Version* base_;
8.   LevelStatelevels_[config::kNumLevels];
9. // 其接口有 3 个：
10.   void Apply(VersionEdit* edit)
11.   void SaveTo(Version* v)
12.   void MaybeAddFile(Version* v, int level, FileMetaData* f)
```

构造函数执行简单的初始化操作，在析构时，遍历检查 LevelState::added_files，如果文件引用计数为 0，则删除文件。

11.5.2 Apply()

函数声明：void **Apply**(VersionEdit* edit)，该函数将 edit 中的修改 应用到 当前 vset_中。注意除了 compaction 点直接修改了 vset_，其它删除和新加文件的变动只是先存储在 Builder 自己的成员变量 levels_[7]中，在调用 SaveTo(v)函数时才施加到 v 上。

- S1 把 edit 记录的 compaction 点应用到当前状态
edit->compact_pointers_ => vset_->compact_pointer_
- S2 把 edit 记录的已删除文件应用到当前状态
edit->deleted_files_ => levels_[level].deleted_files
- S3 把 edit 记录的新加文件应用到当前状态，这里会初始化文件的 allowed_seeks 值，以在文件被无谓 seek 指定次数后自动执行 compaction，这里作者阐述了其设置规则。

```
1.    for (size_t i = 0; i <edit->new_files_.size(); i++) {
2.       const int level =edit->new_files_[i].first;
3.       FileMetaData* f = newFileMetaData(edit->new_files_[i].second);
4.       f->refs = 1;
5.       f->allowed_seeks = (f->file_size /16384); // 16KB-见下面
6.       if (f->allowed_seeks <100) f->allowed_seeks = 100;
7.       levels_[level].deleted_files.erase(f->number); // 以防万一
8.       levels_[level].added_files->insert(f);
9. }
```

值 allowed_seeks 事关 compaction 的优化，其计算依据如下，首先假设：

>1 一次 seek 时间为 10ms
>2 写入 10MB 数据的时间为 10ms（100MB/s）
>3 compact 1MB 的数据需要执行 25MB 的 IO
->从本层读取 1MB
->从下一层读取 10-12MB（文件的 key range 边界可能是非对齐的）
->向下一层写入 10-12MB
这意味这 25 次 seek 的代价等同于 compact 1MB 的数据，也就是一次 seek 花费的时间大约相当于 compact 40KB 的数据。基于保守的角度考虑，对于每 16KB 的数据，我们允许它在触发 compaction 之前能做一次 seek。

11.5.3 MaybeAddFile()

函数声明：void **MaybeAddFile**(Version* v, int level, FileMetaData* f)
该函数尝试将 f 加入到 levels_[level]文件 set 中。
要满足两个条件：
>1 文件不能被删除，也就是不能在 levels_[level].deleted_files 集合中；
>2 保证文件之间的 key 是连续的，即基于比较器 vset_->icmp_，f 的 min key 要大于 levels_[level]集合中最后一个文件的 max key；

11.5.4 SaveTo()

把当前的状态存储到 v 中返回，函数声明：void **SaveTo**(Version* v)

函数逻辑：For 循环遍历所有的 level[0, config::kNumLevels-1]，把新加的文件和已存在的文件 merge 在一起，丢弃已删除的文件，结果保存在 v 中。对于 level> 0，还要确保集合中的文件没有重合。

- S1 merge 流程

```
1.  conststd::vector<FileMetaData*>& base_files = base_->files_[level]; // 原文件集合
2.  std::vector<FileMetaData*>::const_iterator base_iter =base_files.begin();
3.  std::vector<FileMetaData*>::const_iterator base_end =base_files.end();
4.   const FileSet* added =levels_[level].added_files;
5.   v->files_[level].reserve(base_files.size()+ added->size());
6.   for (FileSet::const_iteratoradded_iter = added->begin(); added_iter !=added->end(); ++added_iter) {
7.      //加入 base_中小于 added_iter 的那些文件
8.      for(std::vector<FileMetaData*>::const_iterator bpos = std::upper_bound(base_iter,base_end, *added_iter, cmp);
9.          base_iter != bpos;++base_iter) { // base_iter 逐次向后移到
10.       MaybeAddFile(v, level,*base_iter);
11.     }
12.     MaybeAddFile(v, level,*added_iter); // 加入 added_iter
13.  }
14.  // 添加 base_剩余的那些文件
15.  for (; base_iter != base_end;++base_iter) MaybeAddFile(v, level, *base_iter);
```

对象 cmp 就是前面定义的比较仿函数 BySmallestKey 对象。

- S2 检查流程，保证 level>0 的文件集合无重叠，基于 vset_->icmp_，确保文件 i-1 的 max key < 文件 i 的 min key。

12 DB 的打开

先分析 LevelDB 是如何打开 db 的，万物始于创建。在打开流程中有几个辅助函数：DBImpl()，DBImpl::Recover, DBImpl::DeleteObsoleteFiles, DBImpl::RecoverLogFile, DBImpl::MaybeScheduleCompaction。

12.1 DB::Open()

打开一个 db，进行 PUT、GET 操作，就是前面的静态函数 DB::Open 的工作。如果操作成功，它就返回一个 db 指针。前面说过 DB 就是一个接口类，其具体实现在 DBImp 类中，这是一个 DB 的子类。

函数声明为：

Status DB::**Open**(const Options& options, const std::string&dbname, DB** dbptr);

分解来看，Open()函数主要有以下 5 个执行步骤。

- S1 创建 DBImpl 对象，其后进入 DBImpl::Recover()函数执行 S2 和 S3。
- S2 从已存在的 db 文件恢复 db 数据，根据 CURRENT 记录的 MANIFEST 文件读取 db 元信息；这通过调用 VersionSet::Recover()完成。
- S3 然后过滤出那些最近的更新 log，前一个版本可能新加了这些 log，但并没有记录在 MANIFEST 中。然后依次根据时间顺序，调用 DBImpl::RecoverLogFile()从旧到新回放这些操作 log。回放 log 时可能会修改 db 元信息，比如 dump 了新的 level 0 文件，因此它将返回一个 VersionEdit 对象，记录 db 元信息的变动。
- S4 如果 DBImpl::Recover()返回成功，就执行 VersionSet::LogAndApply()应用 VersionEdit，并保存当前的 DB 信息到新的 MANIFEST 文件中。
- S5 最后删除一些过期文件，并检查是否需要执行 compaction，如果需要，就启动后台线程执行。

下面就来具体分析 Open 函数的代码，在 Open 函数中涉及到上面的 3 个流程。

S1 首先创建 DBImpl 对象，锁定并试图做 Recover 操作。Recover 操作用来处理创建 flag，比如存在就返回失败等等，尝试从已存在的 sstable 文件恢复 db。并返回 db 元信息的变动信息，一个 VersionEdit 对象。

```
1.  DBImpl* impl = newDBImpl(options, dbname);
2.  impl->mutex_.Lock(); // 锁 db
3.  VersionEdit edit;
4.  Status s =impl->Recover(&edit); // 处理 flag&恢复：
    create_if_missing,error_if_exists
```

S2 如果 Recover 返回成功，则调用 VersionSet 取得新的 log 文件编号——实际上是在当前基础上+1，准备新的 log 文件。如果 log 文件创建成功，则根据 log 文件创建 log::Writer。然后执行 VersionSet::LogAndApply，根据 edit 记录的增量变动生成新的 current version，并写入 MANIFEST 文件。
函数 NewFileNumber(){returnnext_file_number_++;}，直接返回 next_file_number_。

```
uint64_t new_log_number =impl->versions_->NewFileNumber();
WritableFile* lfile;
s =options.env->NewWritableFile(LogFileName(dbname, new_log_number),&lfile);

if (s.ok()) {
  edit.SetLogNumber(new_log_number);
  impl->logfile_ = lfile;
  impl->logfile_number_ =new_log_number;
  impl->log_ = newlog::Writer(lfile);
  s =impl->versions_->LogAndApply(&edit, &impl->mutex_);
}
```

S3 如果 VersionSet::LogAndApply 返回成功，则删除过期文件，检查是否需要执行 compaction，最终返回创建的 DBImpl 对象。

```
  if (s.ok()) {
   impl->DeleteObsoleteFiles();
   impl->MaybeScheduleCompaction();
  }
impl->mutex_.Unlock();
if (s.ok()) *dbptr = impl;
return s;
```

以上就是 DB::Open 的主题逻辑。

12.2 DBImpl::DBImpl()

构造函数做的都是初始化操作，DBImpl::DBImpl(const Options& options, const std::string&dbname)
首先是初始化列表中，直接根据参数赋值，或者直接初始化。Comparator 和 filter policy 都是参数传入的。在传递 option 时会首先将 option 中的参数合法化，logfile_number_初始化为 0，指针初始化为 NULL。
创建 MemTable，并增加引用计数，创建 WriteBatch。

```
    mem_(newMemTable(internal_comparator_)),
    tmp_batch_(new WriteBatch),
```

```
3.      mem_->Ref();
4.  // 然后在函数体中，创建 TableCache 和 VersionSet。
5.  // 为其他预留 10 个文件，其余的都给 TableCache.
6.  const int table_cache_size =options.max_open_files - 10;
7.  table_cache_ = newTableCache(dbname_, &options_, table_cache_size);
8.  versions_ = newVersionSet(dbname_, &options_, table_cache_, &internal_compar
    ator_);
```

12.3 DBImp::NewDB()

当外部在调用 DB::Open()时设置了 option 指定如果 db 不存在就创建，如果 db 不存在 leveldb 就会调用函数创建新的 db。判断 db 是否存在的依据是<db name>/CURRENT 文件是否存在。其逻辑很简单。

```
1.  // S1 首先生产 DB 元信息，设置 comparator 名，以及 log 文件编号、文件编号，以及
    seq no。
2.    VersionEdit new_db;
3.    new_db.SetComparatorName(user_comparator()->Name());
4.    new_db.SetLogNumber(0);
5.    new_db.SetNextFile(2);
6.    new_db.SetLastSequence(0);
7.  // S2 生产 MANIFEST 文件，将 db 元信息写入 MANIFEST 文件。
8.    const std::string manifest =DescriptorFileName(dbname_, 1);
9.    WritableFile* file;
10.   Status s =env_->NewWritableFile(manifest, &file);
11.   if (!s.ok()) return s;
12.   {
13.     log::Writer log(file);
14.     std::string record;
15.     new_db.EncodeTo(&record);
16.     s = log.AddRecord(record);
17.     if (s.ok()) s =file->Close();
18.   }
19.   delete file;
20. // S3 如果成功，就把 MANIFEST 文件名写入到 CURRENT 文件中
21.   if (s.ok()) s =SetCurrentFile(env_, dbname_, 1);
22.   elseenv_->DeleteFile(manifest);
23.   return s;
```

这就是创建新 DB 的逻辑，很简单。

12.4 DBImpl::Recover()

函数声明为：StatusDBImpl::Recover(VersionEdit* edit)，如果调用成功则设置 VersionEdit。Recover 的基本功能是：首先是处理创建 flag，比如存在就返回失败等等；然后是尝试从已存在的 sstable 文件恢复 db；最后如果发现有大于原信息记录的 log 编号的 log 文件，则需要回放 log，更新 db 数据。回放期间 db 可能会 dump 新的 level 0 文件，因此需要把 db 元信息的变动记录到 edit 中返回。函数逻辑如下：

S1 创建目录，目录以 db name 命名，忽略任何创建错误，然后尝试获取 db name/LOCK 文件锁，失败则返回。

```
env_->CreateDir(dbname_);
Status s =env_->LockFile(LockFileName(dbname_), &db_lock_);
if (!s.ok()) return s;
```

S2 根据 CURRENT 文件是否存在，以及 option 参数执行检查。

如果文件不存在&create_is_missing=true，则调用函数 NewDB()创建；否则报错。

如果文件存在& error_if_exists=true，则报错。

S3 调用 VersionSet 的 Recover()函数，就是从文件中恢复数据。如果出错则打开失败，成功则向下执行 S4。

```
s = versions_->Recover();
```

S4 尝试从所有比 manifest 文件中记录的 log 要新的 log 文件中恢复（前一个版本可能会添加新的 log 文件，却没有记录在 manifest 中）。另外，函数 PrevLogNumber()已经不再用了，仅为了兼容老版本。

```
1.  //  S4.1 这里先找出所有满足条件的 log 文件：比 manifest 文件记录的 log 编号更新。
2.    SequenceNumber max_sequence(0);
3.    const uint64_t min_log =versions_->LogNumber();
4.    const uint64_t prev_log =versions_->PrevLogNumber();
5.    std::vector<std::string>filenames;
6.    s =env_->GetChildren(dbname_, &filenames); // 列出目录内的所有文件
7.    uint64_t number;
8.    FileType type;
9.    std::vector<uint64_t>logs;
10.   for (size_t i = 0; i <filenames.size(); i++) { // 检查 log 文件是否比 min log
    更新
11.     if(ParseFileName(filenames[i], &number, &type) && type ==kLogFile
12.         && ((number >=min_log) || (number == prev_log))) {
13.       logs.push_back(number);
14.     }
15.   }
16. //  S4.2 找到 log 文件后，首先排序，保证按照生成顺序，依次回放 log。并把 DB 元信息的
    变动（sstable 文件的变动）追加到 edit 中返回。
17.     std::sort(logs.begin(),logs.end());
18.     for (size_t i = 0; i <logs.size(); i++) {
19.       s = RecoverLogFile(logs[i],edit, &max_sequence);
20.       // 前一版可能在生成该 log 编号后没有记录在 MANIFEST 中，
```

```
    //所以这里我们手动更新 VersionSet 中的文件编号计数器
    versions_->MarkFileNumberUsed(logs[i]);
}
//  S4.3 更新 VersionSet 的 sequence
    if (s.ok()) {
      if(versions_->LastSequence() < max_sequence)
         versions_->SetLastSequence(max_sequence);
}
```

上面就是 Recover 的执行流程。

12.5 DBImpl::DeleteObsoleteFiles()

这个是垃圾回收函数，如前所述，每次 compaction 和 recovery 之后都会有文件被废弃。DeleteObsoleteFiles 就是删除这些垃圾文件的，它在每次 compaction 和 recovery 完成之后被调用。

其调用点包括：DBImpl::CompactMemTable,DBImpl::BackgroundCompaction, 以及 DB::Open 的 recovery 步骤之后。

它会删除所有过期的 log 文件，没有被任何 level 引用到、或不是正在执行的 compaction 的 output 的 sstable 文件。

该函数没有参数，其代码逻辑也很直观，就是列出 db 的所有文件，对不同类型的文件分别判断，如果是过期文件，就删除之，如下：

```
// S1 首先，确保不会删除 pending 文件，将 versionset 正在使用的所有文件加入到 live 中。
  std::set<uint64_t> live =pending_outputs_;
  versions_->AddLiveFiles(&live); //该函数其后分析
// S2 列举 db 的所有文件
  std::vector<std::string>filenames;
  env_->GetChildren(dbname_,&filenames);
// S3 遍历所有列举的文件，根据文件类型，分别处理：
  uint64_t number;
  FileType type;
  for (size_t i = 0; i <filenames.size(); i++) {
     if (ParseFileName(filenames[i], &number,&type)) {
         bool keep = true; //false 表明是过期文件
         // S3.1 kLogFile，log 文件，根据 log 编号判断是否过期
              keep = ((number >=versions_->LogNumber()) ||
                   (number ==versions_->PrevLogNumber()));
         // S3.2 kDescriptorFile，MANIFEST 文件，根据 versionset 记录的编号判断
              keep = (number >=versions_->ManifestFileNumber());
         // S3.3 kTableFile，sstable 文件，只要在 live 中就不能删除
         // S3.4 kTempFile，如果是正在写的文件，只要在 live 中就不能删除
```

```
20.              keep = (live.find(number) != live.end());
21.          // S3.5 kCurrentFile,kDBLockFile, kInfoLogFile，不能删除
22.              keep = true;
23.      // S3.6 如果 keep 为 false，表明需要删除文件，如果是 table 还要从 cache 中删
    除
24.           if (!keep) {
25.              if(type == kTableFile) table_cache_->Evict(number);
26.              Log(options_.info_log, "Delete type=%d #%lld\n",type, number);

27.              env_->DeleteFile(dbname_ + "/" +filenames[i]);
28.           }
29.      }
30.   }
```

这就是删除过期文件的逻辑，其中调用到了 VersionSet::AddLiveFiles 函数，保证不会删除 active 的文件。

函数 DbImpl::MaybeScheduleCompaction()放在 Compaction 一节分析，基本逻辑就是如果需要 compaction，就启动后台线程执行 compaction 操作。

12.6 DBImpl::RecoverLogFile()

函数声明：StatusRecoverLogFile(uint64_t log_number, VersionEdit* edit,SequenceNumber* max_sequence)
参数说明：
@log_number 是指定的 log 文件编号
@edit 记录 db 元信息的变化——sstable 文件变动
@max_sequence 返回 max{log 记录的最大序号, *max_sequence}
该函数打开指定的 log 文件，回放日志。期间可能会执行 compaction，生产新的 level 0sstable 文件，记录文件变动到 edit 中。
它声明了一个局部类 LogReporter 以打印错误日志，没什么好说的，下面来看代码逻辑。

```
1.  // S1 打开 log 文件返回 SequentialFile*file，出错就返回，否则向下执行 S2。
2.  // S2 根据 log 文件句柄 file 创建 log::Reader，准备读取 log。
3.    log::Reader reader(file,&reporter, true/*checksum*/,0/*initial_offset*/);

4.  // S3 依次读取所有的 log 记录，并插入到新生成的 memtable 中。这里使用到了批量更新接
    口 WriteBatch，具体后面再分析。
5.    std::string scratch;
6.    Slice record;
7.    WriteBatch batch;
8.    MemTable* mem = NULL;
9.    while(reader.ReadRecord(&record, &scratch) && status.ok()) { // 读取全部
    log
```

```
10.     if (record.size() < 12) { // log 数据错误，不满足最小长度 12
11.      reporter.Corruption(record.size(), Status::Corruption("log recordtoo sm
    all"));
12.       continue;
13.     }
14.    WriteBatchInternal::SetContents(&batch, record); // log 内容设置到
    WriteBatch 中
15.    if (mem == NULL) { // 创建 memtable
16.       mem = new MemTable(internal_comparator_);
17.       mem->Ref();
18.     }
19.     status =WriteBatchInternal::InsertInto(&batch, mem); // 插入到 memtable
    中
20.     MaybeIgnoreError(&status);
21.     if (!status.ok()) break;
22.     const SequenceNumber last_seq=
23.        WriteBatchInternal::Sequence(&batch) + WriteBatchInternal::Count(&bat
    ch)- 1;
24.     if (last_seq >*max_sequence) *max_sequence = last_seq; // 更新
    max sequence
25.     // 如果 mem 的内存超过设置值，则执行 compaction，如果 compaction 出错，
26.     // 立刻返回错误，DB::Open 失败
27.     if(mem->ApproximateMemoryUsage() > options_.write_buffer_size) {
28.       status =WriteLevel0Table(mem, edit, NULL);
29.       if (!status.ok()) break;
30.       mem->Unref(); // 释放当前 memtable
31.       mem = NULL;
32.     }
33.   }
34. // S4 扫尾工作，如果 mem != NULL，说明还需要 dump 到新的 sstable 文件中。
35.   if (status.ok() && mem!= NULL) {// 如果 compaction 出错，立刻返回错误
36.     status = WriteLevel0Table(mem,edit, NULL);
37.   }
38.   if (mem != NULL)mem->Unref();
39.   delete file;
40.   return status;
```

把 MemTabledump 到 sstable 是函数 WriteLevel0Table 的工作，其实这是 compaction 的一部分，准备放在 compaction 一节来分析。

12.7 小结

如上 DB 打开的逻辑就已经分析完了，打开逻辑参见 DB::Open()中描述的 5 个步骤。此外还有两个东东：把 Memtable dump 到 sstable 的 WriteLevel0Table()函数，以及批量

修改 WriteBatch。第一个放在后面的 compaction 一节，第二个放在 DB 更新操作中。接下来就是 db 的关闭。

13 DB 的关闭&销毁

13.1 DB 关闭

外部调用者通过 DB::Open()获取一个 DB*对象，如果要关闭打开的 DB*db 对象，则直接 delete db 即可，这会调用到 DBImpl 的析构函数。
析构依次执行如下的 5 个逻辑：
S1 等待后台 compaction 任务结束
S2 释放 db 文件锁，<dbname>/lock 文件
S3 删除 VersionSet 对象，并释放 MemTable 对象
S4 删除 log 相关以及 TableCache 对象
S5 删除 options 的 block_cache 以及 info_log 对象

13.2 DB 销毁

函数声明：StatusDestroyDB(const std::string& dbname, const Options& options)
该函数会删除掉 db 的数据内容，要谨慎使用。函数逻辑为：
S1 获取 dbname 目录的文件列表到 filenames 中，如果为空则直接返回，否则进入 S2。
S2 锁文件<dbname>/lock，如果锁成功就执行 S3
S3 遍历 filenames 文件列表，过滤掉 lock 文件，依次调用 DeleteFile 删除。
S4 释放 lock 文件，并删除之，然后删除文件夹。
Destory 就执行完了，如果删除文件出现错误，记录之，依然继续删除下一个。最后返回错误代码。
看来这一章很短小。DB 的打开关闭分析完毕。

14 DB 的查询与遍历

分析完如何打开和关闭 db，本章就继续分析如何从 db 中根据 key 查询 value，以及遍历整个 db。

14.1 DbImpl::Get()

函数声明：Status DBImpl::**Get**(const ReadOptions& options, const Slice& key, std::string* value)

从 DB 中查询 key 对应的 value，参数@options 指定读取操作的选项，典型的如 snapshot 号，从指定的快照中读取。快照本质上就是一个 sequence 号，后面将单独在快照一章中分析。
下面就来分析下函数逻辑：

```
// S1 锁 mutex，防止并发，如果指定 option 则尝试获取 snapshot；然后增加 MemTable 的引用值。
  MutexLock l(&mutex_);
  SequenceNumber snapshot;
  if (options.snapshot != NULL)
    snapshot =reinterpret_cast<const SnapshotImpl*>(options.snapshot)->number_;
  else snapshot =versions_->LastSequence(); // 取当前版本的最后 Sequence
  MemTable *mem = mem_, *imm =imm_;
  Version* current =versions_->current();
  mem->Ref();
  if (imm != NULL) imm->Ref();
  current->Ref();
// S2 从 sstable 文件和 MemTable 中读取时，释放锁 mutex；之后再次锁 mutex。
  bool have_stat_update = false;
  Version::GetStats stats;
  {
    mutex_.Unlock();
    // 先从 memtable 中查询，再从 immutable memtable 中查询
    LookupKey lkey(key, snapshot);
    if (mem->Get(lkey, value,&s)) {
    } else if (imm != NULL&& imm->Get(lkey, value, &s)) {
    } else { // 需要从 sstable 文件中查询
      s = current->Get(options,lkey, value, &stats);
      have_stat_update = true; // 记录之，用于 compaction
    }
    mutex_.Lock();
  }
// S3 如果是从 sstable 文件查询出来的，检查是否需要做 compaction。最后把 MemTable 的引用计数减 1。
  if (have_stat_update &¤t->UpdateStats(stats)) {
    MaybeScheduleCompaction();
  }
  mem->Unref();
  if (imm != NULL)imm->Unref();
  current->Unref();
```

查询是比较简单的操作，UpdateStats 在前面 Version 一节已经分析过。

14.2 DBImpl::NewIterator()

函数声明：Iterator* DBImpl::**NewIterator**(const ReadOptions& options)
通过该函数生产了一个 Iterator*对象，调用这就可以基于该对象遍历 db 内容了。
函数很简单，调用两个函数创建了一个二级 Iterator。

```
Iterator* DBImpl::NewIterator(const ReadOptions& options) {
  SequenceNumber latest_snapshot;
  Iterator* internal_iter =NewInternalIterator(options, &latest_snapshot);
  returnNewDBIterator(&dbname_, env_, user_comparator(), internal_iter,
      (options.snapshot != NULL
       ? reinterpret_cast<constSnapshotImpl*>(options.snapshot)->number_
       : latest_snapshot));
}
```

其中，函数 NewDBIterator 直接返回了一个 DBIter 指针

```
Iterator* NewDBIterator(const std::string* dbname, Env* env,
    const Comparator*user_key_comparator, Iterator* internal_iter,
    const SequenceNumber& sequence) {
  return new DBIter(dbname, env,user_key_comparator, internal_iter, sequence
);
}
```

函数 NewInternalIterator 有一些处理逻辑，就是收集所有能用到的 iterator，生产一个 Merging Iterator。这包括 MemTable，Immutable MemTable，以及各 sstable。

```
Iterator* DBImpl::NewInternalIterator(const ReadOptions& options,
                                      SequenceNumber*latest_snapshot) {
  IterState* cleanup = newIterState;
  mutex_.Lock();
  // 根据 last sequence 设置 lastest snapshot，并收集所有的子 iterator
  *latest_snapshot =versions_->LastSequence();
  std::vector<Iterator*>list;
  list.push_back(mem_->NewIterator()); // >memtable
  mem_->Ref();
  if (imm_ != NULL) {
   list.push_back(imm_->NewIterator()); // >immutablememtable
    imm_->Ref();
  }
  versions_->current()->AddIterators(options, &list); // >current 的所有
sstable
  Iterator* internal_iter = NewMergingIterator(&internal_comparator_,&list[0
], list.size());
  versions_->current()->Ref();
  // 注册清理机制
  cleanup->mu = &mutex_;
  cleanup->mem = mem_;
```

```
20.   cleanup->imm = imm_;
21.   cleanup->version =versions_->current();
22.   internal_iter->RegisterCleanup(CleanupIteratorState, cleanup, NULL);
23.   mutex_.Unlock();
24.   return internal_iter;
25. }
```

这个清理函数 CleanupIteratorState 是很简单的，对注册的对象做一下 Unref 操作即可。

```
1.  static void CleanupIteratorState(void* arg1, void* arg2) {
2.    IterState* state =reinterpret_cast<IterState*>(arg1);
3.    state->mu->Lock();
4.    state->mem->Unref();
5.    if (state->imm != NULL)state->imm->Unref();
6.    state->version->Unref();
7.    state->mu->Unlock();
8.    delete state;
9.  }
```

可见对于 db 的遍历依赖于 DBIter 和 Merging Iterator 这两个迭代器，它们都是 Iterator 接口的实现子类。

14.3 MergingIterator

MergingIterator 是一个合并迭代器，它内部使用了一组自 Iterator，保存在其成员数组 children_中。如上面的函数 NewInternalIterator，包括 memtable，immutable memtable，以及各 sstable 文件；它所做的就是根据调用者指定的 key 和 sequence，从这些 Iterator 中找到合适的记录。
在分析其 Iterator 接口之前，先来看看两个辅助函数 FindSmallest 和 FindLargest。FindSmallest 从 0 开始向后遍历内部 Iterator 数组，找到 key 最小的 Iterator，并设置到 current_；FindLargest 从最后一个向前遍历内部 Iterator 数组，找到 key 最大的 Iterator，并设置到 current_；
MergingIterator 还定义了两个移动方向：kForward，向前移动；kReverse，向后移动。

14.3.1 Get 系接口

下面就把其接口拖出来一个一个分析，首先是简单接口，key 和 value 都是返回 current_的值，current_是当前 seek 到的 Iterator 位置。

```
1.  virtual Slice key() const {
2.     assert(Valid());
3.     return current_->key();
4.    }
```

```
5.    virtual Slice value() const {
6.      assert(Valid());
7.      return current_->value();
8.    }
9.    virtual Status status() const {
10.     Status status;
11.     for (int i = 0; i < n_;i++) { // 只有所有内部 Iterator 都 ok 时，才返回 ok
12.       status =children_[i].status();
13.       if (!status.ok()) break;
14.     }
15.     return status;
16.   }
```

14.3.2 Seek 系接口

然后是几个 seek 系的函数，也比较简单，都是依次调用内部 Iterator 的 seek 系函数。然后做 merge，对于 Seek 和 SeekToFirst 都调用 FindSmallest；对于 SeekToLast 调用 FindLargest。

```
1.  virtual void SeekToFirst() {
2.    for (int i = 0; i < n_;i++) children_[i].SeekToFirst();
3.    FindSmallest();
4.    direction_ = kForward;
5.  }
6.  virtual void SeekToLast() {
7.    for (int i = 0; i < n_;i++) children_[i].SeekToLast();
8.    FindLargest();
9.    direction_ = kReverse;
10. }
11. virtual void Seek(constSlice& target) {
12.   for (int i = 0; i < n_;i++) children_[i].Seek(target);
13.   FindSmallest();
14.   direction_ = kForward;
15. }
```

14.3.3 逐步移动

最后就是 Next 和 Prev 函数，完成迭代遍历。这可能会有点绕。下面分别来说明。
首先，在 Next 移动时，如果当前 direction 不是 kForward 的，也就是上一次调用了 Prev 或者 SeekToLast 函数，就需要先调整除 current 之外的所有 iterator，为什么要做这种调整呢？啰嗦一点，考虑如下的场景，如图 14.3-1 所示。

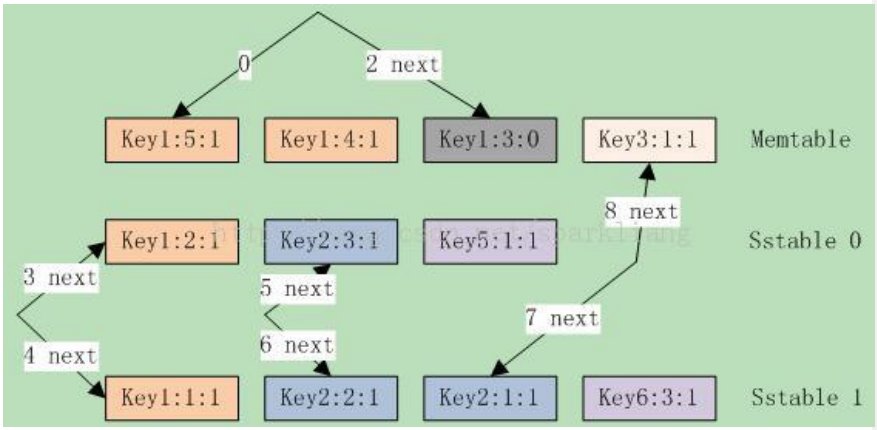


图 14.3-1 Next 的移动

当前 direction 为 kReverse，并且有：Current = memtable Iterator。各 Iterator 位置为：{memtable, stable 0, sstable1} ={ key3:1:1, key2:3:1, key2:1:1}，这符合 prev 操作的 largest key 要求。

注：需要说明下，对于每个 update 操作，leveldb 都会赋予一个全局唯一的 sequence 号，且是递增的。例子中的 sequence 号可理解为每个 key 的相对值，后面也是如此。

接下来我们来分析 Prev 移动的操作。

第一次 Prev，current(memtable iterator)移动到 key1:3:0 上，3 者中最大者变成 sstable0；因此 current 修改为 sstable0；

第二次 Prev，current(sstable0 Iterator)移动到 key1:2:1 上，3 者中最大者变成 sstable1；因此 current 修改为 sstable1:

此时各 Iterator 的位置为{memtable, sstable 0, sstable1} = { key1:3:0, key1:2:1, key2:2:1}，并且 current=sstable1。

接下来再调用 Next，显然当前 Key()为 key2:2:1，综合考虑 3 个 iterator，两次 Next()的调用结果应该是 key2:1:1 和 key3:1:1。而 memtable 和 sstable0 指向的 key 却是 key1:3:0 和 key1:2:1，这时就需要调整 memtable 和 sstable0 了，使他们都定位到 Key()之后，也就是 key3:1:1 和 key2:3:1 上。

然后 current(current1)Next 移动到 key2:1:1 上。这就是 Next 时的调整逻辑，同理，对于 Prev 也有相同的调整逻辑。代码如下：

```
virtual void Next() {
    assert(Valid());
    // 确保所有的子 Iterator 都定位在 key()之后.
    // 如果我们在正向移动，对于除 current_外的所有子 Iterator 这点已然成立
    // 因为 current_是最小的子 Iterator，并且 key() = current_->key()。
    // 否则，我们需要明确设置其它的子 Iterator
```

```
7.      if (direction_ != kForward) {
8.        for (int i = 0; i < n_;i++) { // 把所有 current 之外的 Iterator 定位到
    key()之后
9.          IteratorWrapper* child =&children_[i];
10.         if (child != current_) {
11.           child->Seek(key());
12.           if (child->Valid()&& comparator_->Compare(key(), child->key()) ==
    0)
13.             child->Next(); // key 等于 current_->key()的，再向后移动一位
14.         }
15.       }
16.       direction_ = kForward;
17.     }
18.     // current 也向后移一位，然后再查找 key 最小的 Iterator
19.     current_->Next();
20.     FindSmallest();
21.   }
22.
23.   virtual void Prev() {
24.     assert(Valid());
25.     // 确保所有的子 Iterator 都定位在 key()之前.
26.     // 如果我们在逆向移动，对于除 current_外的所有子 Iterator 这点已然成立
27.     // 因为 current_是最大的，并且 key() = current_->key()
28.     // 否则，我们需要明确设置其它的子 Iterator
29.     if (direction_ != kReverse) {
30.       for (int i = 0; i < n_;i++) {
31.         IteratorWrapper* child =&children_[i];
32.         if (child != current_) {
33.           child->Seek(key());
34.           if (child->Valid()) {
35.             // child 位于>=key()的第一个 entry 上，prev 移动一位到<key()
36.             child->Prev();
37.           } else { // child 所有的 entry 都 < key()，直接 seek 到 last 即可
38.            child->SeekToLast();
39.           }
40.         }
41.       }
42.       direction_ = kReverse;
43.     }
44.     //current 也向前移一位，然后再查找 key 最大的 Iterator
45.     current_->Prev();
46.     FindLargest();
47.   }
```

这就是 MergingIterator 的全部代码逻辑了，每次 Next 或者 Prev 移动时，都要重新遍历所有的子 Iterator 以找到 key 最小或最大的 Iterator 作为 current_。这就是 merge 的语义所在了。
但是它没有考虑到删除标记等问题，因此直接使用 MergingIterator 是不能正确的遍历 DB 的，这些问题留待给 DBIter 来解决。