【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。

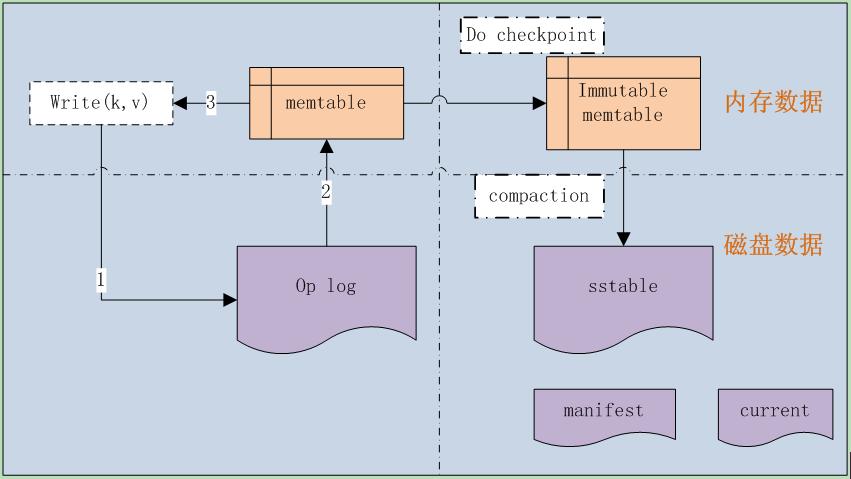


图1-1

**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](http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.8933)>>。

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. 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. 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](#_int_Coding)。

在操作log中使用的是Fixed存储格式。

* 1. 字符比较

是基于unsigned char的，而非char。

1. 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。

* 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》）。

* 1. 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 

* 1. 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。

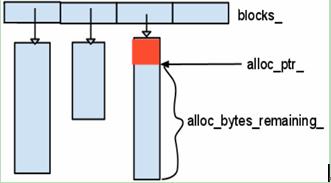


图2.3-1

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。

* 1. 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。

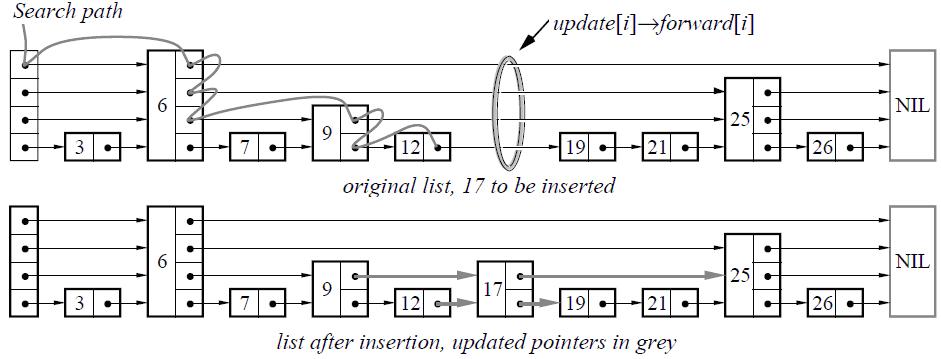


图2.4-1

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。

* 1. 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。

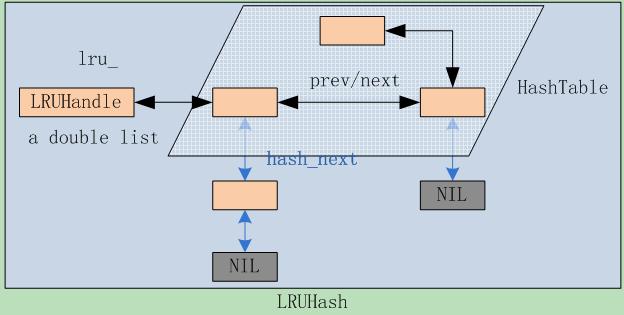


图2.5-1

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。

* 1. 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.

* 1. function: Hash.cc : Hash()
  2. namespace: crc32c

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

* 1. class Histogram

1. int Coding整形编码  
   “Coding.h Coding.cc”

轻松一刻，[前面约定中](#_一些约定)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。

* 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);

}

* 1. 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;

}

1. 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。

* 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) 。

* 1. 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。

* 1. 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。

* 1. 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。

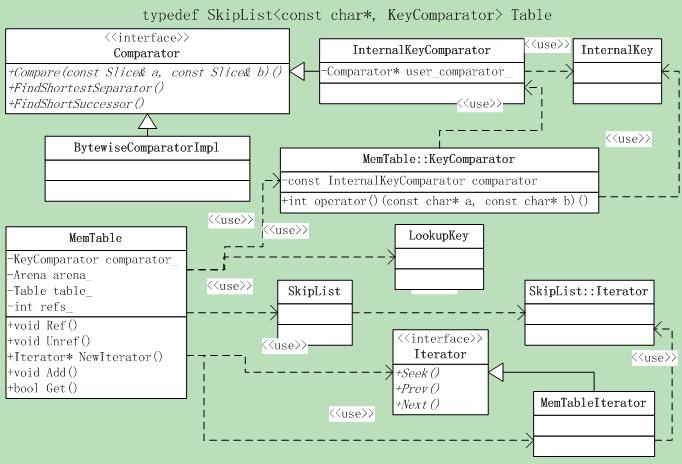


图4.4-1

* 1. 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。

* + 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);

}

* + 1. 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.

* 1. 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，

1. // Three-way comparison. Returns value:
2. // < 0 iff "a" < "b",
3. // == 0 iff "a" == "b",
4. // > 0 iff "a" > "b"
5. virtual int Compare(const Slice& a, const Slice& b) const = 0;
6. // The name of the comparator. Used to check for comparator
7. // mismatches (i.e., a DB created with one comparator is
8. // accessed using a different comparator.
9. //
10. // The client of this package should switch to a new name whenever
11. // the comparator implementation changes in a way that will cause
12. // the relative ordering of any two keys to change.
13. //
14. // Names starting with "leveldb." are reserved and should not be used
15. // by any clients of this package.
16. 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：

1. // Advanced functions: these are used to reduce the space requirements
2. // for internal data structures like index blocks.
3. // 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。
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) **const** = 0;
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。

* + 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。

* + 1. 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：

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

函数实现：

1. {
2. // 尝试更新user key，基于指定的user comparator
3. Slice user\_start = ExtractUserKey(\*start);
4. Slice user\_limit = ExtractUserKey(limit);
5. std::string tmp(user\_start.data(), user\_start.size());
6. user\_comparator\_->FindShortestSeparator(&tmp, user\_limit);
7. **if**(tmp.size()<user\_start.size()&&user\_comparator\_->Compare(user\_start, tmp)<0)
8. {
9. // 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，
10. // 使用最大的sequence number，以保证排在相同user key记录序列的第一个
11. PutFixed64(&tmp, PackSequenceAndType(kMaxSequenceNumber,kValueTypeForSeek));
12. assert(**this**->Compare(\*start, tmp) < 0);
13. assert(**this**->Compare(tmp, limit) < 0);
14. start->swap(tmp);
15. } //endif
16. } //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,kValueTypeForSeek));
10. assert(**this**->Compare(\*key, tmp) < 0);
11. key->swap(tmp);
12. } //end if
13. }
    * 1. 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：

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

代码实现如下：

1. // Format of an entry is concatenation of:
2. //  key\_size     : varint32 of internal\_key.size()
3. //  key bytes    : char[internal\_key.size()]
4. //  value\_size   : varint32 of value.size()
5. //  value bytes  : char[value.size()]
6. **size\_t** key\_size = key.size();
7. **size\_t** val\_size = value.size();
8. **size\_t** internal\_key\_size = key\_size + 8;
9. **const** **size\_t** encoded\_len =
10. VarintLength(internal\_key\_size) + internal\_key\_size +
11. VarintLength(val\_size) + val\_size;
12. **char**\* buf = arena\_.Allocate(encoded\_len);
13. **char**\* p = EncodeVarint32(buf, internal\_key\_size);
14. memcpy(p, key.data(), key\_size);
15. p += key\_size;
16. EncodeFixed64(p, (s << 8) | type);
17. p += 8;
18. p = EncodeVarint32(p, val\_size);
19. memcpy(p, value.data(), val\_size);
20. assert((p + val\_size) - buf == encoded\_len);
21. 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

* + 1. 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：

1. Slice memkey = key.memtable\_key();
2. Table::Iterator iter(&table\_);
3. iter.Seek(memkey.data()); // seek到value>= memkey.data()的第一个记录
4. **if** (iter.Valid()) {
5. // 这里不需要再检查sequence number了，因为Seek()已经跳过了所有
6. // 值更大的sequence number了
7. // entry format is:
8. // klength varint32
9. // userkey char[klength]
10. // tag uint64
11. // vlength varint32
12. // value char[vlength]
13. // Check that it belongs to same user key. We do not check the
14. // sequence number since the Seek() call above should have skipped
15. // all entries with overly large sequence numbers.
16. **const** **char**\* entry = iter.key();
17. uint32\_t key\_length;
18. **const** **char**\* key\_ptr = GetVarint32Ptr(entry, entry+5, &key\_length);
19. // 比较user key是否相同，key\_ptr开始的len(internal key) -8 byte是user key
20. **if** (comparator\_.comparator.user\_comparator()->Compare(
21. Slice(key\_ptr, key\_length - 8),
22. key.user\_key()) == 0) {
23. // len(internal key)的后8byte是 |sequence number | value type|
24. **const** uint64\_t tag = DecodeFixed64(key\_ptr + key\_length - 8);
25. **switch** (**static\_cast**<ValueType>(tag & 0xff)) {
26. **case** kTypeValue: { // 只取出value
27. Slice v = GetLengthPrefixedSlice(key\_ptr + key\_length);
28. value->assign(v.data(), v.size());
29. **return** **true**;
30. }
31. **case** kTypeDeletion:  // 删除
32. \*s = Status::NotFound(Slice());
33. **return** **true**;
34. }
35. }
36. }  //end if
37. **return** **false**;
38. }//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。

* 1. 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）

1. 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。

* 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

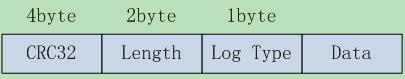
    type: uint8           // One of FULL,FIRST, MIDDLE, LAST

    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。

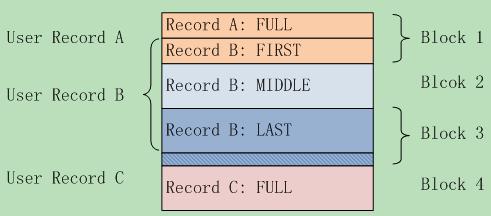


图5.1-1

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.

* 1. 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。

* 1. 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。

* + 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。

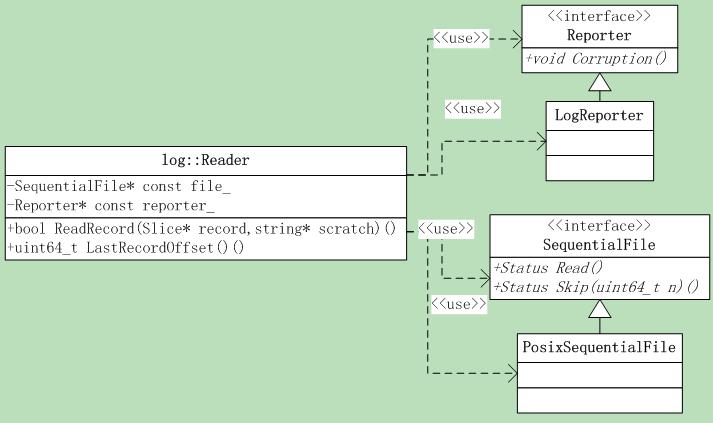


图5.3-1

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

* + 1. 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。

* + 1. 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。

1. 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。

* 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；

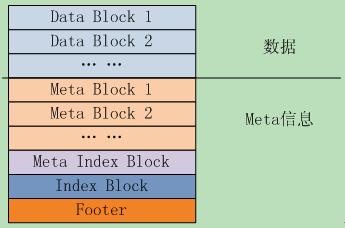


图6.1-1

5）Footer，文件的最后，大小固定，其格式如图6.1-2所示。

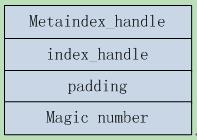


图6.1-2

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）。

* 1. Block存储格式
     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。

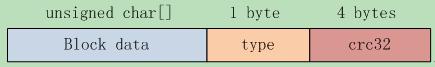


图6.2-1

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。

* + 1. 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。

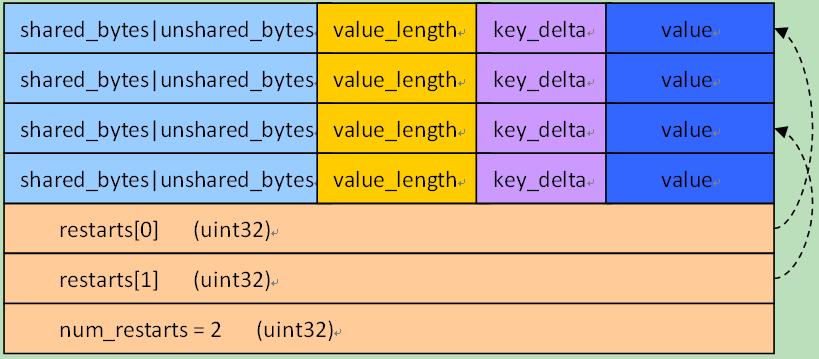


图6.2-2

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。

* 1. Block的构建与读取
     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
   * 1. 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；

1. assert(!finished\_);
2. assert(counter\_ <= options\_->block\_restart\_interval);
3. 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；

1. Slice last\_key\_piece(last\_key\_);
2. **if** (counter\_ < options\_->block\_restart\_interval) { //前缀压缩
3. // 计算key与last\_key\_的公共前缀
4. **const** **size\_t** min\_length= std::min(last\_key\_piece.size(), key.size());
5. **while** ((shared < min\_length)&& (last\_key\_piece[shared] == key[shared]))  {
6. shared++;
7. }**else**{ // 新的重启点
8. restarts\_.push\_back(buffer\_.size());
9. counter\_ = 0;
10. }

* 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。

1. **const** **size\_t** non\_shared = key.size() - shared; // key前缀之后的字符串长度
2. // append"<shared><non\_shared><value\_size>" 到buffer\_
3. PutVarint32(&buffer\_, shared);
4. PutVarint32(&buffer\_, non\_shared);
5. PutVarint32(&buffer\_, value.size());
6. // 其后是前缀之后的字符串 + value
7. buffer\_.append(key.data() + shared, non\_shared);
8. buffer\_.append(value.data(), value.size());
9. // 更新状态 ，last\_key\_ = key及计数器counter\_
10. last\_key\_.resize(shared);   // 连一个string的赋值都要照顾到，使内存copy最小化
11. last\_key\_.append(key.data() + shared, non\_shared);
12. assert(Slice(last\_key\_) == key);
13. counter\_++;
    * 1. 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.

1. **for** (**size\_t** i = 0; i < restarts\_.size(); i++) {  // 重启点
2. PutFixed32(&buffer\_, restarts\_[i]);
3. }
4. PutFixed32(&buffer\_, restarts\_.size());    // 重启点数量
5. finished\_ = **true**;
6. **return** Slice(buffer\_);
   * 1. 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。

1. **void** BlockBuilder::Reset() {
2. buffer\_.clear();  restarts\_.clear();  last\_key\_.clear();
3. restarts\_.push\_back(0);       // 第一个重启点位置总是 0
4. counter\_ = 0;
5. finished\_ = **false**;
6. }
8. **size\_t** BlockBuilder::CurrentSizeEstimate () **const** {
9. // buffer大小 +重启点数组长度 + 重启点长度(uint32)
10. **return** (buffer\_.size() +  restarts\_.size() \* **sizeof**(uint32\_t) + **sizeof**(uint32\_t));
11. }

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

* + 1. 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);
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
   * 1. 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. 新版代码：
9. Block::**Block**(const BlockContents& contents)
10. : data\_(contents.data.data()),
11. size\_(contents.data.size()),
12. owned\_(contents.heap\_allocated) {
13. **if** (size\_ < sizeof(uint32\_t)) {
14. size\_ = 0; // Error marker
15. } **else** {
16. size\_t max\_restarts\_allowed = (size\_-sizeof(uint32\_t)) / sizeof(uint32\_t);
17. **if** (NumRestarts() > max\_restarts\_allowed) {
18. // The size is too small for NumRestarts()
19. size\_ = 0;
20. } **else** {
21. restart\_offset\_ = size\_ - (1 + NumRestarts()) \* sizeof(uint32\_t);
22. }
23. }
24. }

**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))

* + 1. 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。

1. current\_ = NextEntryOffset(); // (value\_.data() + value\_.size()) - data\_
2. **const** **char**\* p = data\_ +current\_;
3. **const** **char**\* limit = data\_ +restarts\_; // Restarts come right after data
4. **if** (p >= limit) { // entry到头了，标记为invalid.
5. current\_ = restarts\_;
6. restart\_index\_ =num\_restarts\_;
7. **return** **false**;
8. }

* 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。

1. uint32\_t shared, non\_shared,value\_length;
2. p = **DecodeEntry**(p, limit, &shared, &non\_shared, &value\_length);
3. **if** (p == NULL || key\_.size()< shared) {
4. **CorruptionError**();
5. **return** **false**;
6. } **else** { // 成功
7. key\_.resize(shared);
8. key\_.append(p, non\_shared);
9. value\_ = Slice(p +non\_shared, value\_length);
10. **while** (restart\_index\_ + 1< num\_restarts\_ && **GetRestartPoint**(restart\_index\_ + 1) < current\_) {
11. ++restart\_index\_;
12. }
13. **return** **true**;
14. }

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。

1. **const** uint32\_t original = current\_;
2. **while** (GetRestartPoint(restart\_index\_)>= original) {
3. **if** (restart\_index\_ == 0) { // 到第一个entry了，标记invalid状态
4. current\_ = restarts\_;
5. restart\_index\_ =num\_restarts\_;
6. **return**;
7. }
8. restart\_index\_--;
9. }
10. **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。

1. **void** **SeekToRestartPoint**(uint32\_tindex) {
2. key\_.clear();
3. restart\_index\_ = index;
4. // ParseNextKey()会设置current\_;
5. //ParseNextKey()从value\_结尾开始, 因此需要相应的设置value\_
6. uint32\_t offset = **GetRestartPoint**(index);
7. value\_ = Slice(data\_ + offset,0); // value长度设置为0，字符串指针是data\_+offset
8. }

> **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。

1. **virtual** **void** SeekToFirst() {
2. SeekToRestartPoint(0);
3. ParseNextKey();
4. }
6. **virtual** **void** SeekToLast() {
7. 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。

* 1. 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。

* + 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；

> 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。

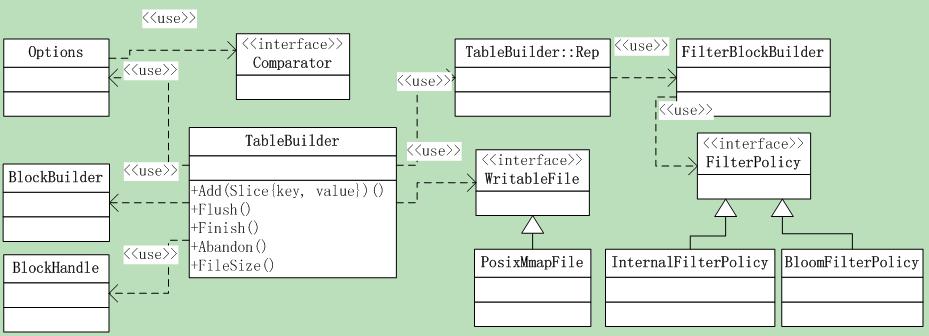


图6.3-1

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
9. **int64**\_t num\_entries; //当前data block的个数，初始0
10. **bool** closed;          //调用了Finish() or Abandon()，初始false
11. **FilterBlockBuilder**\* filter\_block; //根据filter数据快速定位key是否在block中
12. **bool** pending\_index\_entry; //见下面的Add函数，初始false
13. **BlockHandle** pending\_handle; //添加到index block的value（data block的信息）
14. **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。

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。

* + 1. 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。

1. Rep\* r = rep\_;
2. assert(!r->closed);
3. **if** (!ok()) **return**;
4. **if** (r->num\_entries > 0) {
5. assert(r->options.comparator->Compare(key, Slice(r->last\_key))> 0);
6. }

* 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。

1. **if** (r->pending\_index\_entry) {
2. assert(r->data\_block.empty());
3. r->options.comparator->FindShortestSeparator(&r->last\_key,key);
4. std::string handle\_encoding;
5. r->pending\_handle.EncodeTo(&handle\_encoding);
6. r->index\_block.Add(r->last\_key, Slice(handle\_encoding));
7. r->pending\_index\_entry =**false**;
8. }

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();
   * 1. 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. // 保证pending\_index\_entry为false，即data block的Add已经完成
6. assert(!r->pending\_index\_entry);
7. // 写入data block，并设置其index entry信息—BlockHandle对象
8. WriteBlock(&r->data\_block, &r->pending\_handle);
9. //写入成功，则Flush文件，并设置r->pending\_index\_entry为true，
10. //以根据下一个data block的first key调整index entry的key—即r->last\_key
11. **if** (ok()) {
12. r->pending\_index\_entry =**true**;
13. r->status =r->file->Flush();
14. }
15. **if** (r->filter\_block != NULL){ //将data block在sstable中的偏移加入到filter block中
16. r->filter\_block->StartBlock(r->offset); // 并指明开始新的data block
17. }
    * 1. 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;
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](http://blog.csdn.net/sparkliang/article/details/8653370)

1. WriteRawBlock(block\_contents,type, handle);
2. r->compressed\_output.clear();
3. block->Reset();
   * 1. 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。

1. Rep\* r = rep\_;
2. handle->set\_offset(r->offset); // 为index设置data block的handle信息
3. handle->set\_size(block\_contents.size());
5. nbsp;r->status =r->file->Append(block\_contents); // 写入data block内容
6. **if** (r->status.ok()) {// 写入1byte的type和4bytes的crc32
7. chartrailer[kBlockTrailerSize];
8. trailer[0] = type;
9. uint32\_t crc = crc32c::Value(block\_contents.data(),block\_contents.size());
10. crc = crc32c::Extend(crc, trailer, 1);  // Extend crc to cover block type
11. EncodeFixed32(trailer+1, crc32c::Mask(crc));
12. r->status =r->file->Append(Slice(trailer, kBlockTrailerSize));
13. **if** (r->status.ok()) { // 写入成功更新offset-下一个data block的写入偏移
14. r->offset +=block\_contents.size() + kBlockTrailerSize;
15. }
16. }
    * 1. 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。

1. Rep\* r = rep\_;
2. Flush();
3. assert(!r->closed);
4. r->closed = **true**;

BlockHandle filter\_block\_handle,metaindex\_block\_handle, index\_block\_handle;

* S2 写入filter block到文件中

1. **if** (ok() &&r->filter\_block != NULL) {
2. WriteRawBlock(r->filter\_block->Finish(), kNoCompression,&filter\_block\_handle);
3. }

* 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。

1. **if** (ok()) {
2. BlockBuildermeta\_index\_block(&r->options);
3. **if** (r->filter\_block !=NULL) {
4. //加入从"filter.Name"到filter data位置的映射
5. std::string key ="filter.";
6. key.append(r->options.filter\_policy->Name());
7. std::string handle\_encoding;
8. filter\_block\_handle.EncodeTo(&handle\_encoding);
9. meta\_index\_block.Add(key,handle\_encoding);
10. }
11. // TODO(postrelease): Add stats and other metablocks
12. WriteBlock(&meta\_index\_block, &metaindex\_block\_handle);
13. }

* 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。

1. **if** (ok()) {
2. **if** (r->pending\_index\_entry){ // Flush时会被设置为true
3. r->options.comparator->FindShortSuccessor(&r->last\_key);
4. std::string handle\_encoding;
5. r->pending\_handle.EncodeTo(&handle\_encoding);
6. r->index\_block.Add(r->last\_key, Slice(handle\_encoding)); // 加入到index block中
7. r->pending\_index\_entry =**false**;
8. }
9. WriteBlock(&r->index\_block, &index\_block\_handle);
10. }

* S5 写入Footer。

1. **if** (ok()) {
2. Footer footer;
3. footer.set\_metaindex\_handle(metaindex\_block\_handle);
4. footer.set\_index\_handle(index\_block\_handle);
5. std::string footer\_encoding;
6. footer.EncodeTo(&footer\_encoding);
8. 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。

* 1. 读取sstable文件
     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。

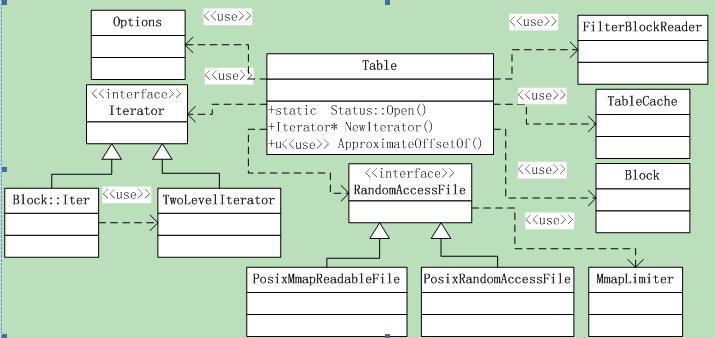


图6.5-1

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。

* + 1. 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. }
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;
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.

* + 1. 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。

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。

* + 1. 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](http://blog.csdn.net/sparkliang/article/details/8681759)

1. Block\* meta = **newBlock**(contents);
2. Iterator\* iter = meta->**NewIterator**(BytewiseComparator());
3. std::string key ="filter.";
4. key.append(rep\_->options.filter\_policy->Name());
5. iter->Seek(key);
6. **if** (iter->Valid() &&iter->key() == Slice(key)) **ReadFilter**(iter->value());
7. **delete** iter;
8. **delete** meta;
   * 1. 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。

1. ReadOptions opt;
2. BlockContents block;
3. ReadBlock(rep\_->file, opt,filter\_handle, &block);
4. **if** (block.heap\_allocated)rep\_->filter\_data = block.data.data(); // 需要自行释放内存
5. 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。

* 1. 遍历Table   
     “Two\_level\_iterator.cc Table.cc”
     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。

* + 1. 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：

1. **virtual** **bool** **Valid**() **const** {
2. **return** data\_iter\_.Valid();
3. }
4. **virtual** Slice **key**() **const** {
5. assert(Valid());
6. **return** data\_iter\_.key();
7. }
8. **virtual** Slice **value**() **const** {
9. assert(Valid());
10. **return** data\_iter\_.value();
11. }

* 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。

1. **void** **InitDataBlock**();
2. **void** **SetDataIterator**(Iterator\* data\_iter); //设置date\_iter\_ = data\_iter
3. **voidSkipEmptyDataBlocksForward**();
4. **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：
5. **if** (!index\_iter\_.Valid()) SetDataIterator(NULL);// index\_iter非法
6. **else** {
7. Slice handle = index\_iter\_.value();
8. **if** (data\_iter\_.iter() != NULL && handle.compare(data\_block\_handle\_) == 0) {
9. //data\_iter已经在该block data上了，无须改变
10. } **else** { // 根据handle数据定位data iter
11. Iterator\* iter = (\*block\_function\_)(arg\_, options\_, handle);
12. data\_block\_handle\_.assign(handle.data(), handle.size());
13. SetDataIterator(iter);
14. }
15. }
    * S2.2 **SkipEmptyDataBlocksForward**，向前跳过空的datablock，函数实现如下：
16. **while** (data\_iter\_.iter() == NULL|| !data\_iter\_.Valid()) { // 跳到下一个block
17. **if** (!index\_iter\_.Valid()) { // 如果index iter非法，设置data iteration为NULL
18. SetDataIterator(NULL);
19. **return**;
20. }
21. index\_iter\_.Next();
22. InitDataBlock();
23. **if** (data\_iter\_.iter() != NULL)data\_iter\_.SeekToFirst(); // 跳转到开始
24. }
    * S2.3 **SkipEmptyDataBlocksBackward**，向后跳过空的datablock，函数实现如下：
25. **while** (data\_iter\_.iter() == NULL|| !data\_iter\_.Valid()) { // 跳到前一个block
26. **if** (!index\_iter\_.Valid()) { // 如果index iter非法，设置data iteration为NULL
27. SetDataIterator(NULL);
28. **return**;
29. }
30. index\_iter\_.Prev();
31. InitDataBlock();
32. **if** (data\_iter\_.iter() != NULL)data\_iter\_.SeekToLast();// 跳转到开始
33. }

* S3 了解了几个跳转的辅助函数，再来看**Seek**系接口。

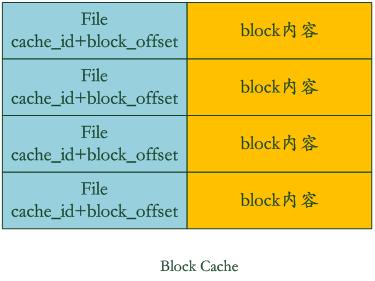
1. **void** TwoLevelIterator::**Seek**(**const** Slice& target) {
2. index\_iter\_.Seek(target);
3. InitDataBlock(); // 根据index iter设置data iter
4. **if** (data\_iter\_.iter() != NULL) data\_iter\_.Seek(target); // 调整data iter跳转到target
5. SkipEmptyDataBlocksForward(); // 调整iter，跳过空的block
6. }
8. **void** TwoLevelIterator::SeekToFirst() {
9. index\_iter\_.SeekToFirst();
10. InitDataBlock();// 根据index iter设置data iter
11. **if** (data\_iter\_.iter() != NULL)data\_iter\_.SeekToFirst();
12. SkipEmptyDataBlocksForward();// 调整iter，跳过空的block
13. }
15. **void** TwoLevelIterator::SeekToLast() {
16. index\_iter\_.SeekToLast();
17. InitDataBlock();// 根据index iter设置data iter
18. **if** (data\_iter\_.iter() != NULL)data\_iter\_.SeekToLast();
19. SkipEmptyDataBlocksBackward();// 调整iter，跳过空的block
20. }
22. **void** TwoLevelIterator::Next() {
23. assert(Valid());
24. data\_iter\_.Next();
25. SkipEmptyDataBlocksForward();// 调整iter，跳过空的block
26. }
28. **void** TwoLevelIterator::Prev() {
29. assert(Valid());
30. data\_iter\_.Prev();
31. SkipEmptyDataBlocksBackward();// 调整iter，跳过空的block
32. }
    * 1. 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。



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。

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. }
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. }
    1. 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。
  1. 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->value());
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。

1. TableCache

这章的内容比较简单，篇幅也不长。

* 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：

1. Env\* **const** env\_; // 用来操作文件
2. **const** std::string dbname\_; // db名
3. 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：

1. **void** **Evict**(uint64\_t file\_number);
2. // 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。
3. EncodeFixed64(buf,file\_number); cache\_->Erase(Slice(buf, **sizeof**(buf)));
4. Iterator\* **NewIterator**(const ReadOptions& options, uint64\_t file\_number,
5. uint64\_t file\_size, Table\*\*tableptr = NULL);
6. //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
7. Status **Get**(constReadOptions& options,
8. uint64\_t file\_number,uint64\_t file\_size,
9. **const** Slice& k,**void**\* arg,
10. **void**(\*handle\_result)(**void**\*, **const** Slice&, **const** Slice&));
11. // 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).
    1. TableCache::Get()

先来看看Get接口，只有几行代码：

1. Cache::Handle\* handle = NULL;
2. Status s =FindTable(file\_number, file\_size, &handle);
3. **if** (s.ok()) {
4. Table\* t =**reinterpret\_cast**<TableAndFile\*>(cache\_->Value(handle))->table;
5. s = t->InternalGet(options,k, arg, saver);
6. 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。

* 1. 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;
   1. 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。

1. **char** buf[**sizeof**(file\_number)];
2. EncodeFixed64(buf, file\_number);
3. Slice key(buf, **sizeof**(buf));
4. \*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。

1. std::string fname =TableFileName(dbname\_, file\_number);
2. RandomAccessFile\* file = NULL;
3. Table\* table = NULL;
4. s =env\_->NewRandomAccessFile(fname, &file);
5. **if** (s.ok()) s =Table::Open(\*options\_, file, file\_size, &table);

* S3 如果Table::Open成功则，插入到Cache中。

1. TableAndFile\* tf = newTableAndFile(table, file);
2. \*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。

* 1. 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。

1. **static** **void** DeleteEntry(**const** Slice& key, **void**\* value) {
2. TableAndFile\* tf =**reinterpret\_cast**<TableAndFile\*>(value);
3. **delete** tf->table;
4. **delete** tf->file;
5. **delete** tf;
6. }

另外一个是为Iterator注册的清除函数UnrefEntry。

1. **static** **void** UnrefEntry(**void**\* arg1, **void**\* arg2) {
2. Cache\* cache =**reinterpret\_cast**<Cache\*>(arg1);
3. Cache::Handle\* h =**reinterpret\_cast**<Cache::Handle\*>(arg2);
4. cache->Release(h);
5. }
6. FilterPolicy&Bloom
   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。

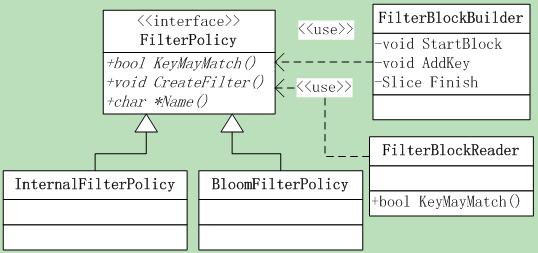


图8.1-1

* 1. 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) **const** {
2. returnuser\_policy\_->**KeyMayMatch**(ExtractUserKey(key), f);
3. }
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. }
   1. BloomFilter
      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。

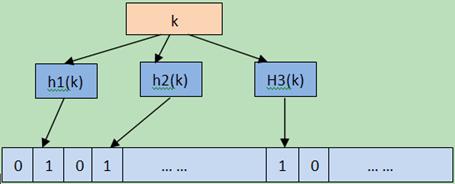


图8.3-1

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](http://www.eecs.harvard.edu/~kirsch/pubs/bbbf/esa06.pdf)】

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。

* + 1. 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

* + 1. 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。

1. **size\_t** bits = n \* bits\_per\_key\_;
2. **if** (bits < 64) bits = 64; // 如果n太小FP会很高，限定filter的最小长度
3. **size\_t** bytes = (bits + 7) / 8;// 圆整到8byte
4. bits = bytes \* 8; // bit计算的空间大小
5. **const** **size\_t** init\_size =dst->size();
6. 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。

1. **char**\* array =&(\*dst)[init\_size];
2. **for** (**size\_t** i = 0; i < n;i++) {
3. // double-hashing，分析参见[Kirsch,Mitzenmacher 2006]
4. uint32\_t h =BloomHash(keys[i]); // h1函数
5. **const** uint32\_t delta = (h>> 17) | (h << 15); // h2函数、由h1 Rotate right 17 bits
6. **for** (**size\_t** j = 0; j <k\_; j++) { // double-hashing生产k\_个的hash值
7. **const** uint32\_t bitpos = h% bits; // 在bits array上设置第bitpos位
8. array[bitpos/8] |= (1<< (bitpos % 8));
9. h += delta;
10. }
11. }

Bloom Filter的创建就完成了。

* + 1. 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。

1. **const** **size\_t** len =bloom\_filter.size();
2. **if** (len < 2) **return** **false**;
3. **const** **char**\* array = bloom\_filter.data();
4. **const** **size\_t** bits = (len - 1)\* 8;
5. **const** **size\_t** k = array[len-1];// 使用filter的k，而不是k\_，这样更灵活
6. **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。

1. uint32\_t h = BloomHash(key);
2. **const** uint32\_t delta = (h>> 17) | (h << 15);  // Rotate right 17 bits
3. **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
   1. 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。

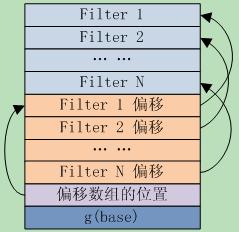


图8.4-1 filter block data

了解了格式，再分析构建和读取filter的代码就很简单了。

* 1. 构建FilterBlock
     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**。

* + 1. 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
5. **return**;
6. }
7. //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。
8. start\_.push\_back(keys\_.size());  // Simplify lengthcomputation
9. tmp\_keys\_.resize(num\_keys);
10. **for** (**size\_t** i = 0; i <num\_keys; i++) {
11. **const** **char**\* base =keys\_.data() + start\_[i]; // 开始指针
12. **size\_t** length = start\_[i+1] -start\_[i]; // 长度
13. tmp\_keys\_[i] = Slice(base,length);
14. }
15. //S3 为当前的key集合生产filter，并append到result\_
16. filter\_offsets\_.push\_back(result\_.size());
17. policy\_->CreateFilter(&tmp\_keys\_[0], num\_keys, &result\_);
18. //S4 清空，重置状态
19. tmp\_keys\_.clear();
20. keys\_.clear();
21. start\_.clear();
    * 1. 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。

1. Slice k = key;
2. start\_.push\_back(keys\_.size());
3. keys\_.append(k.data(),k.size());
   * 1. 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：

1. //S1 如果start\_数字不空，把为的key列表生产filter
2. **if** (!start\_.empty()) GenerateFilter();
3. //S2 从0开始顺序存储各filter的偏移值，见filter block data的数据格式。
4. **const** uint32\_t array\_offset =result\_.size();
5. **for** (**size\_t** i = 0; i < filter\_offsets\_.size();i++) {
6. PutFixed32(&result\_,filter\_offsets\_[i]);
7. }
8. //S3 最后是filter个数，和shift常量（11），并返回结果
9. PutFixed32(&result\_,array\_offset);
10. result\_.push\_back(kFilterBaseLg);  // Save encoding parameter in result
11. **return** Slice(result\_);
    * 1. 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。

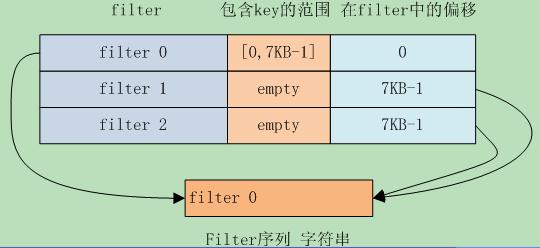


图8.5-1

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。

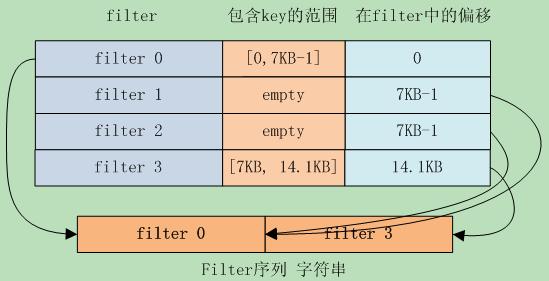


图8.5-2

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。

* 1. 读取FilterBlock
     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);

* + 1. 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
    * 1. 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。

1. 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。

* 1. DB文件管理
     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。

* + 1. 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
  1. 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。

* + 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。
  + 1. 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

* 1. Recovery & GC
     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

* + 1. 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。

* 1. 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。

* + 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。

* + 1. 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。

* + 1. 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。

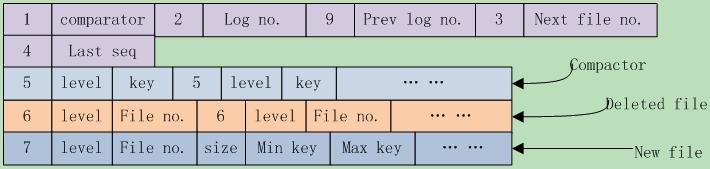


图 9.3-1

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。

* 1. DB接口
     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** Slice& 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::string\* 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;
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;
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);
    * 1. 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。

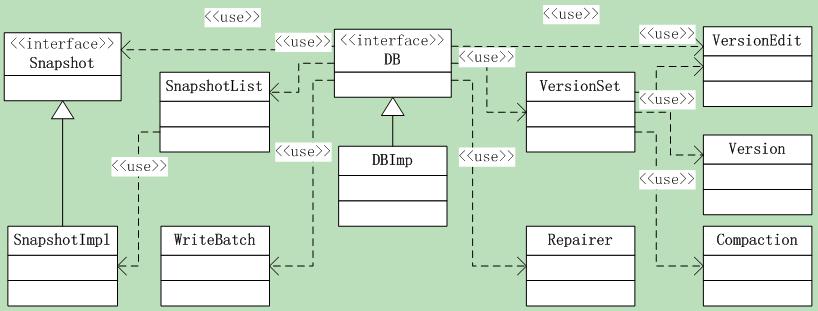


图9.5-1

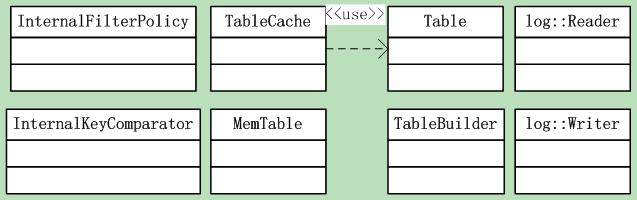


图9.5-2

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

* 1. 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](http://blog.csdn.net/sparkliang/article/details/8776583)

1. //== 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。
2. Env\* **const** env\_; // 环境，封装了系统相关的文件操作、线程等等
3. **const** InternalKeyComparatorinternal\_comparator\_;  // key comparator
4. **const** InternalFilterPolicyinternal\_filter\_policy\_;      // filter policy
5. **const** Options options\_;  //options\_.comparator == &internal\_comparator\_
6. **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
35. 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。

* 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);
5. // 给定@key查找value，如果找到保存在@\*val并返回OK。
6. // 否则返回non-OK，设置@ \*stats.
7. // 要求：没有hold lock
8. **struct** GetStats {
9. FileMetaData\* seek\_file;
10. **int** seek\_file\_level;
11. };
12. Status Get(const ReadOptions&, **const** LookupKey& key, std::string\* val, GetStats\* stats);
14. // 把@stats加入到当前状态中，如果需要触发新的compaction返回true
15. // 要求：hold lock
16. **bool** UpdateStats(const GetStats& stats);
17. **void** GetOverlappingInputs(int level,
18. **const** InternalKey\* begin,         // NULL指在所有key之前
19. **const** InternalKey\* end,           // NULL指在所有key之后
20. std::vector<FileMetaData\*>\* inputs);
22. // 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。
23. // @smallest\_user\_key==NULL表示比DB中所有key都小的key.
24. // @largest\_user\_key==NULL表示比DB中所有key都大的key.
25. **bool** OverlapInLevel(**int** level,**const** Slice\* smallest\_user\_key, **const** Slice\* largest\_user\_key);
27. // 返回我们应该在哪个level上放置新的memtable compaction，
28. // 该compaction覆盖了范围[smallest\_user\_key,largest\_user\_key].
29. **int** PickLevelForMemTableOutput(**const** Slice& smallest\_user\_key,
30. **const** Slice& largest\_user\_key);
32. **int** NumFiles(**int** level) **const** {**return** files\_[level].size(); } // 指定level的sstable个数
    1. 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。

1. **for** (**size\_t** i = 0; i <files\_[0].size(); i++) {
2. iters->push\_back(vset\_->table\_cache\_->NewIterator(// versionset::table\_cache\_
3. options,files\_[0][i]->number, files\_[0][i]->file\_size));
4. }

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

1. **for** (**int** ll = 1; ll <config::kNumLevels; ll++) {
2. **if**(!files\_[ll].empty()) iters->push\_back(NewConcatenatingIterator(options,level));
3. }

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)

1. TableCache\* cache =**reinterpret\_cast**<TableCache\*>(arg);
2. **if** (file\_value.size() != 16) { // 错误
3. return NewErrorIterator(Status::Corruption("xxx"));
4. } **else** {
5. return cache->NewIterator(options,
6. DecodeFixed64(file\_value.data()), // filenumber
7. DecodeFixed64(file\_value.data() + 8)); // filesize
8. }

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。

* 1. 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\_, target);}
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. }
9. **virtual** **void** Prev() {
10. assert(Valid());
11. **if** (index\_ == 0) index\_ =flist\_->size(); // Marks as invalid
12. **else** index\_--;
13. }
15. Slice key() **const** {
16. assert(Valid());
17. **return**(\*flist\_)[index\_]->largest.Encode(); // 返回当前sstable包含的largest key
18. }
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。

* 1. 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
7. }
8. **if** (tmp.empty()) **continue**;
9. std::sort(tmp.begin(),tmp.end(), NewestFirst); // 排序
10. 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。
11. //二分查找，找到第一个largest key >=ikey的file index
12. uint32\_t index =FindFile(vset\_->icmp\_, files\_[level], ikey);
13. **if** (index >= num\_files) { // 未找到，文件不存在
14. files = NULL;  num\_files = 0;
15. } **else** {
16. tmp2 = files[index];
17. **if**(ucmp->Compare(user\_key, tmp2->smallest.user\_key()) < 0) {
18. // 找到的文件其所有key都大于user\_key，等于文件不存在
19. files = NULL;  num\_files = 0;
20. } **else** {
21. files = &tmp2;  num\_files = 1;
22. }
23. }

* 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。

1. **if**(last\_file\_read != NULL && stats->seek\_file == NULL) {
2. // 本次读取不止seek了一个文件，记录第一个
3. stats->seek\_file =last\_file\_read;
4. stats->seek\_file\_level= last\_file\_read\_level;
5. }
6. FileMetaData\* f = files[i];
7. last\_file\_read = f; // 记录本次读取的level和file
8. 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()。
9. Saver saver; // 初始化saver
10. saver.state = kNotFound;
11. saver.ucmp = ucmp;
12. saver.user\_key = user\_key;
13. saver.value = value;
14. s = vset\_->table\_cache\_->Get(options,f->number, f->file\_size,
15. ikey, &saver, SaveValue);
16. **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。
17. **switch** (saver.state) {
18. **case** kNotFound: **break**; // 继续搜索下一个更早的sstable文件
19. **case** kFound:  **return** s; // 找到
20. **case** kDeleted: // 已删除
21. s =Status::NotFound(Slice());  // 为了效率，使用空的错误字符串
22. **return** s;
23. **case** kCorrupt: // 数据损坏
24. s =Status::Corruption("corrupted key for ", user\_key);
25. **return** s;
26. }

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：

1. Saver\* s =**reinterpret\_cast**<Saver\*>(arg);
2. ParsedInternalKey parsed\_key; // 解析ikey到ParsedInternalKey
3. **if** (!ParseInternalKey(ikey,&parsed\_key)) s->state = kCorrupt; // 解析失败
4. **else** {
5. **if**(s->ucmp->Compare(parsed\_key.user\_key, s->user\_key) == 0) { // 比较user key
6. s->state =(parsed\_key.type == kTypeValue) ? kFound : kDeleted;
7. **if** (s->state == kFound)s->value->assign(v.data(), v.size()); // 找到，保存结果
8. }
9. }

下面要分析的几个函数，或多或少都和compaction相关。

* 1. 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：

1. FileMetaData\* f =stats.seek\_file;
2. **if** (f != NULL) {
3. f->allowed\_seeks--;
4. **if** (f->allowed\_seeks <=0 && file\_to\_compact\_ == NULL) {
5. file\_to\_compact\_ = f;
6. file\_to\_compact\_level\_ =stats.seek\_file\_level;
7. **return** **true**;
8. }
9. }
10. **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。

* 1. 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。

1. inputs->clear();
2. 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. }
    1. 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。

1. // S1 乱序、可能相交的文件集合，依次查找
2. **for** (**size\_t** i = 0; i <files.size(); i++) {
3. **const** FileMetaData\* f =files[i];
4. **if**(AfterFile(ucmp,smallest\_user\_key, f) || BeforeFile(ucmp, largest\_user\_key, f)){
5. } **else** **return** **true**;  // 有重合
6. }
7. **return** **false**; // 没找到
8. // S2 有序&互不相交，直接二分查找
9. uint32\_t index = 0;
10. **if** (smallest\_user\_key != NULL) {
11. // Findthe earliest possible internal key smallest\_user\_key
12. InternalKeysmall(\*smallest\_user\_key, kMaxSequenceNumber,kValueTypeForSeek);
13. index = FindFile(icmp, files,small.Encode());
14. }
15. **if** (index >= files.size())**return** **false**; // 不存在比smallest\_user\_key小的key
16. **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。

1. **static** **bool** AfterFile(**const** Comparator\* ucmp,
2. **const** Slice\* user\_key, constFileMetaData\* f) {
3. **return** (user\_key!=NULL&& ucmp->Compare(\*user\_key, f->largest.user\_key())>0);
4. }
5. **static** **bool** BeforeFile(**const** Comparator\* ucmp,
6. constSlice\* user\_key, **const** FileMetaData\* f) {
7. **return** (user\_key!=NULL&& ucmp->Compare(\*user\_key, f->smallest.user\_key())<0);
8. }
   1. 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。

* 1. 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。

1. 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。

* 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。

1. 恢复函数，从磁盘恢复最后保存的元信息

Status **Recover**();

1. 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;

1. 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)

1. 对于@v中的@key，返回db中的大概位置

uint64\_t **ApproximateOffsetOf**(Version\* v, const InternalKey& key);

1. 其它一些简单接口，信息获取或者设置，如下：
2. Version\* current() **const** { **return** current\_; } //返回current version
3. // 当前的MANIFEST文件号
4. uint64\_t ManifestFileNumber() **const** { **return** manifest\_file\_number\_; }
5. uint64\_t NewFileNumber() { **return** next\_file\_number\_++; } // 分配并返回新的文件编号
6. uint64\_t LogNumber() **const** { **return** log\_number\_; } // 返回当前log文件编号
7. // 返回正在compact的log文件编号，如果没有返回0
8. uint64\_t PrevLogNumber() **const** { **return** prev\_log\_number\_; }
9. // 获取、设置last sequence，set时不能后退
10. uint64\_t LastSequence() **const** { **return** last\_sequence\_; }
11. **void** SetLastSequence(uint64\_t s) {
12. assert(s >=last\_sequence\_);
13. last\_sequence\_ = s;
14. }
15. // 返回指定level中所有sstable文件大小的和
16. int64\_t NumLevelBytes(**int** level) **const**;
17. // 返回指定level的文件个数
18. **int** NumLevelFiles(**int** level) **const**;
19. // 重用@file\_number，限制很严格：@file\_number必须是最后分配的那个
20. // 要求: @file\_number是NewFileNumber()返回的.
21. **void** ReuseFileNumber(uint64\_t file\_number) {
22. **if** (next\_file\_number\_ ==file\_number + 1) next\_file\_number\_ = file\_number;
23. }
24. // 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)
25. // 这个就是Version::GetOverlappingInputs()函数的简单应用
26. int64\_t MaxNextLevelOverlappingBytes();
27. // 获取函数，把所有version的所有level的文件加入到@live中
28. **void** AddLiveFiles(std::set<uint64\_t>\* live)
29. // 返回一个可读的单行信息——每个level的文件数，保存在\*scratch中
30. **struct** LevelSummaryStorage {**char** buffer[100]; };
31. **const** **char**\* LevelSummary(LevelSummaryStorage\* scratch) **const**;

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

* 1. 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。

* + 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。

1. std::string current; // MANIFEST文件名
2. ReadFileToString(env\_, CurrentFileName(dbname\_), &current)
3. std::string dscname = dbname\_ + "/" + current;
4. SequentialFile\* file;
5. 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。

1. Builder builder(**this**, current\_);
2. **while** (reader.ReadRecord(&record, &scratch) && s.ok()) {
3. VersionEdit edit;
4. s = edit.DecodeFrom(record);
5. **if** (s.ok())builder.Apply(&edit);
6. **if** (edit.has\_log\_number\_) { // log number, file number, …逐个判断
7. log\_number =edit.log\_number\_;
8. have\_log\_number = **true**;
9. }
10. … …
11. }

* 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。

1. **if** (s.ok()) {
2. Version\* v = newVersion(**this**);
3. builder.SaveTo(v);
4. // 安装恢复的version
5. Finalize(v);
6. AppendVersion(v);
7. manifest\_file\_number\_ =next\_file;
8. next\_file\_number\_ = next\_file+ 1;
9. last\_sequence\_ = last\_sequence;
10. log\_number\_ = log\_number;
11. prev\_log\_number\_ =prev\_log\_number;
12. }

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。

* + 1. 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。

* + 1. 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。

* 1. 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。

* + 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。

1. Version\* v = **new** Version(**this**);
2. {
3. Builder builder(**this**,current\_);
4. builder.Apply(edit);
5. builder.SaveTo(v);
6. }
7. 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。

1. std::string new\_manifest\_file;
2. Status s;
3. **if** (descriptor\_log\_ == NULL) {
4. // 这里不需要unlock \*mu因为我们只会在第一次调用LogAndApply时
5. // 才走到这里(打开数据库时).
6. assert(descriptor\_file\_ ==NULL); // 文件指针和log::Writer都应该是NULL
7. new\_manifest\_file =DescriptorFileName(dbname\_, manifest\_file\_number\_);
8. edit->SetNextFile(next\_file\_number\_);
9. s =env\_->NewWritableFile(new\_manifest\_file, &descriptor\_file\_);
10. **if** (s.ok()) {
11. descriptor\_log\_ = **new** log::Writer(descriptor\_file\_);
12. s =WriteSnapshot(descriptor\_log\_); // 写入快照
13. }
14. }

* 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。

1. mu->Unlock();
2. **if** (s.ok()) {
3. std::string record;
4. edit->EncodeTo(&record);// 序列化current version信息
5. s =descriptor\_log\_->AddRecord(record); // append到MANIFEST log中
6. **if** (s.ok()) s =descriptor\_file\_->Sync();
7. **if** (!s.ok()) {
8. Log(options\_->info\_log,"MANIFEST write: %s\n", s.ToString().c\_str());
9. **if** (**ManifestContains**(record)) { // 返回出错，其实确实写成功了
10. Log(options\_->info\_log, "MANIFEST contains log record despiteerror ");
11. s = Status::OK();
12. }
13. }
14. }
15. //如果刚才创建了一个MANIFEST文件，通过写一个指向它的CURRENT文件
16. //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
17. **if** (s.ok() &&!new\_manifest\_file.empty()) {
18. s = SetCurrentFile(env\_,dbname\_, manifest\_file\_number\_);
19. }
20. mu->Lock();

* S5 安装这个新的version

1. **if** (s.ok()) { // 安装这个version
2. AppendVersion(v);
3. log\_number\_ =edit->log\_number\_;
4. prev\_log\_number\_ =edit->prev\_log\_number\_;
5. } **else** { // 失败了，删除
6. **delete** v;
7. **if**(!new\_manifest\_file.empty()) {
8. **delete** descriptor\_log\_;
9. **delete** descriptor\_file\_;
10. descriptor\_log\_ =descriptor\_file\_ = NULL;
11. env\_->DeleteFile(new\_manifest\_file);
12. }
13. }

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。

* + 1. 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。

* + 1. 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。

* 1. 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。

1. 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。

* 1. **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。

* + 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。

* + 1. **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。

* + 1. **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；

* + 1. **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。

1. 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。

* 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\_。

1. uint64\_t new\_log\_number =impl->versions\_->NewFileNumber();
2. WritableFile\* lfile;
3. s =options.env->NewWritableFile(LogFileName(dbname, new\_log\_number),&lfile);
4. **if** (s.ok()) {
5. edit.SetLogNumber(new\_log\_number);
6. impl->logfile\_ = lfile;
7. impl->logfile\_number\_ =new\_log\_number;
8. impl->log\_ = newlog::Writer(lfile);
9. s =impl->versions\_->LogAndApply(&edit, &impl->mutex\_);
10. }

 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。

1. **if** (s.ok()) {
2. impl->DeleteObsoleteFiles();
3. impl->MaybeScheduleCompaction();
4. }
5. impl->mutex\_.Unlock();
6. **if** (s.ok()) \*dbptr = impl;
7. **return** s;

以上就是DB::Open的主题逻辑。

* 1. 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。

1. mem\_(newMemTable(internal\_comparator\_)),
2. 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\_comparator\_);
   1. 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。

* 1. 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，
21. //所以这里我们手动更新VersionSet中的文件编号计数器
22. versions\_->MarkFileNumberUsed(logs[i]);
23. }
24. //  S4.3 更新VersionSet的sequence
25. **if** (s.ok()) {
26. **if**(versions\_->LastSequence() < max\_sequence)
27. versions\_->SetLastSequence(max\_sequence);
28. }

上面就是Recover的执行流程。

* 1. 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：

1. // 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。
2. std::set<uint64\_t> live =pending\_outputs\_;
3. versions\_->AddLiveFiles(&live); //该函数其后分析
4. // S2 列举db的所有文件
5. std::vector<std::string>filenames;
6. env\_->GetChildren(dbname\_,&filenames);
7. // S3 遍历所有列举的文件，根据文件类型，分别处理；
8. uint64\_t number;
9. FileType type;
10. **for** (**size\_t** i = 0; i <filenames.size(); i++) {
11. **if** (ParseFileName(filenames[i], &number,&type)) {
12. **bool** keep = **true**; //false表明是过期文件
13. // S3.1 kLogFile，log文件，根据log编号判断是否过期
14. keep = ((number >=versions\_->LogNumber()) ||
15. (number ==versions\_->PrevLogNumber()));
16. // S3.2 kDescriptorFile，MANIFEST文件，根据versionset记录的编号判断
17. keep = (number >=versions\_->ManifestFileNumber());
18. // S3.3 kTableFile，sstable文件，只要在live中就不能删除
19. // 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。

* 1. 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 small"));
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(&batch)- 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。

* 1. 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。

1. DB的关闭&销毁
   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

* 1. 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。

1. 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。

* 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：

1. // 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。
2. MutexLock l(&mutex\_);
3. SequenceNumber snapshot;
4. **if** (options.snapshot != NULL)
5. snapshot =**reinterpret\_cast**<**const** SnapshotImpl\*>(options.snapshot)->number\_;
6. **else** snapshot =versions\_->LastSequence(); // 取当前版本的最后Sequence
7. MemTable \*mem = mem\_, \*imm =imm\_;
8. Version\* current =versions\_->current();
9. mem->Ref();
10. **if** (imm != NULL) imm->Ref();
11. current->Ref();
12. // S2 从sstable文件和MemTable中读取时，释放锁mutex；之后再次锁mutex。
13. **bool** have\_stat\_update = **false**;
14. Version::GetStats stats;
15. {
16. mutex\_.Unlock();
17. // 先从memtable中查询，再从immutable memtable中查询
18. LookupKey lkey(key, snapshot);
19. **if** (mem->Get(lkey, value,&s)) {
20. } **else** **if** (imm != NULL&& imm->Get(lkey, value, &s)) {
21. } **else** { // 需要从sstable文件中查询
22. s = current->Get(options,lkey, value, &stats);
23. have\_stat\_update = **true**; // 记录之，用于compaction
24. }
25. mutex\_.Lock();
26. }
27. // 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。
28. **if** (have\_stat\_update &¤t->UpdateStats(stats)) {
29. MaybeScheduleCompaction();
30. }
31. mem->Unref();
32. **if** (imm != NULL)imm->Unref();
33. current->Unref();

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

* 1. 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。

1. Iterator\* DBImpl::NewIterator(**const** ReadOptions& options) {
2. SequenceNumber latest\_snapshot;
3. Iterator\* internal\_iter =NewInternalIterator(options, &latest\_snapshot);
4. returnNewDBIterator(&dbname\_, env\_, user\_comparator(), internal\_iter,
5. (options.snapshot != NULL
6. ? **reinterpret\_cast**<constSnapshotImpl\*>(options.snapshot)->number\_
7. : latest\_snapshot));
8. }

其中，函数NewDBIterator直接返回了一个DBIter指针

1. Iterator\* NewDBIterator(**const** std::string\* dbname, Env\* env,
2. **const** Comparator\*user\_key\_comparator, Iterator\* internal\_iter,
3. **const** SequenceNumber& sequence) {
4. **return** **new** DBIter(dbname, env,user\_key\_comparator, internal\_iter, sequence);
5. }

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。

1. Iterator\* DBImpl::NewInternalIterator(**const** ReadOptions& options,
2. SequenceNumber\*latest\_snapshot) {
3. IterState\* cleanup = newIterState;
4. mutex\_.Lock();
5. // 根据last sequence设置lastest snapshot，并收集所有的子iterator
6. \*latest\_snapshot =versions\_->LastSequence();
7. std::vector<Iterator\*>list;
8. list.push\_back(mem\_->NewIterator()); // >memtable
9. mem\_->Ref();
10. **if** (imm\_ != NULL) {
11. list.push\_back(imm\_->NewIterator()); // >immutablememtable
12. imm\_->Ref();
13. }
14. versions\_->current()->AddIterators(options, &list); // >current的所有sstable
15. Iterator\* internal\_iter = NewMergingIterator(&internal\_comparator\_,&list[0], list.size());
16. versions\_->current()->Ref();
17. // 注册清理机制
18. cleanup->mu = &mutex\_;
19. 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。

* 1. 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。

* + 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. }
    * 1. 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. }
    * 1. 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。

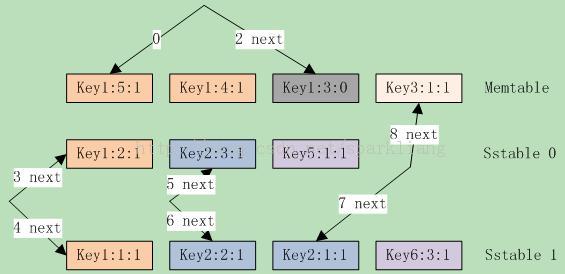


图14.3-1 Next的移动

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：

1. **virtual** **void** Next() {
2. assert(Valid());
3. // 确保所有的子Iterator都定位在key()之后.
4. // 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
5. // 因为current\_是最小的子Iterator，并且key() = current\_->key()。
6. // 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. }
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。