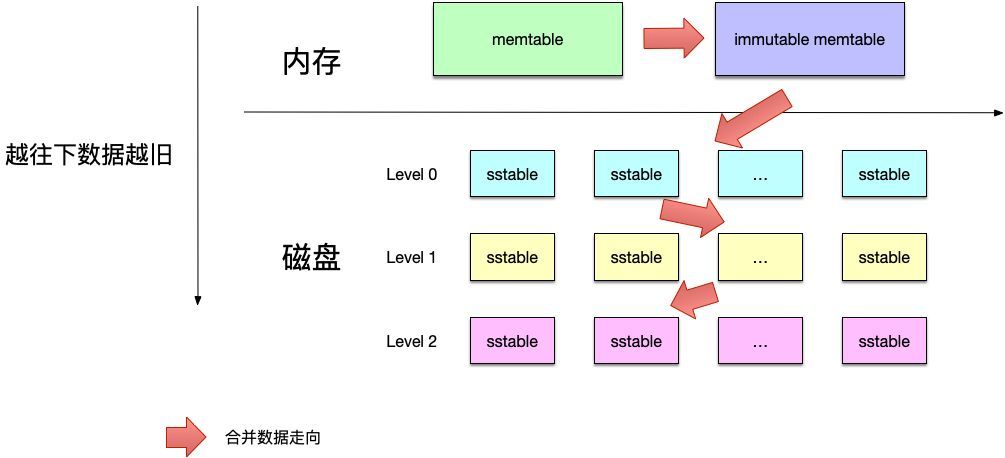
# **经典开源代码分析——Leveldb高效存储实现**

导读：LevelDB是Google开源的持久化KV数据库，在其高性能的背后，将数据拆分成多层进行存储。本文作者深入分析了LevelDB存储模块的设计和源码实现，快速了解LevelDB高性能背后的原理。

作者 codedump codedump.info 博主，多年从事互联网服务器后台开发工作。可访问作者博客阅读 codedump 更多文章。

本文基于leveldb 1.9.0代码。

****整体架构****

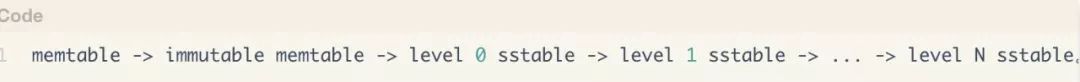


如上图，leveldb的数据存储在内存以及磁盘上，其中：

* memtable：存储在内存中的数据，使用skiplist实现。
* immutable memtable：与memtable一样，只不过这个memtable不能再进行修改，会将其中的数据落盘到level 0的sstable中。
* 多层sstable：leveldb使用多个层次来存储sstable文件，这些文件分布在磁盘上，这些文件都是根据键值有序排列的，其中0级的sstable的键值可能会重叠，而level 1及以上的sstable文件不会重叠。

在上面这个存储层次中，越靠上的数据越新，即同一个键值如果同时存在于memtable和immutable memtable中，则以memtable中的为准。

另外，图中还使用箭头来表示了合并数据的走向，即：



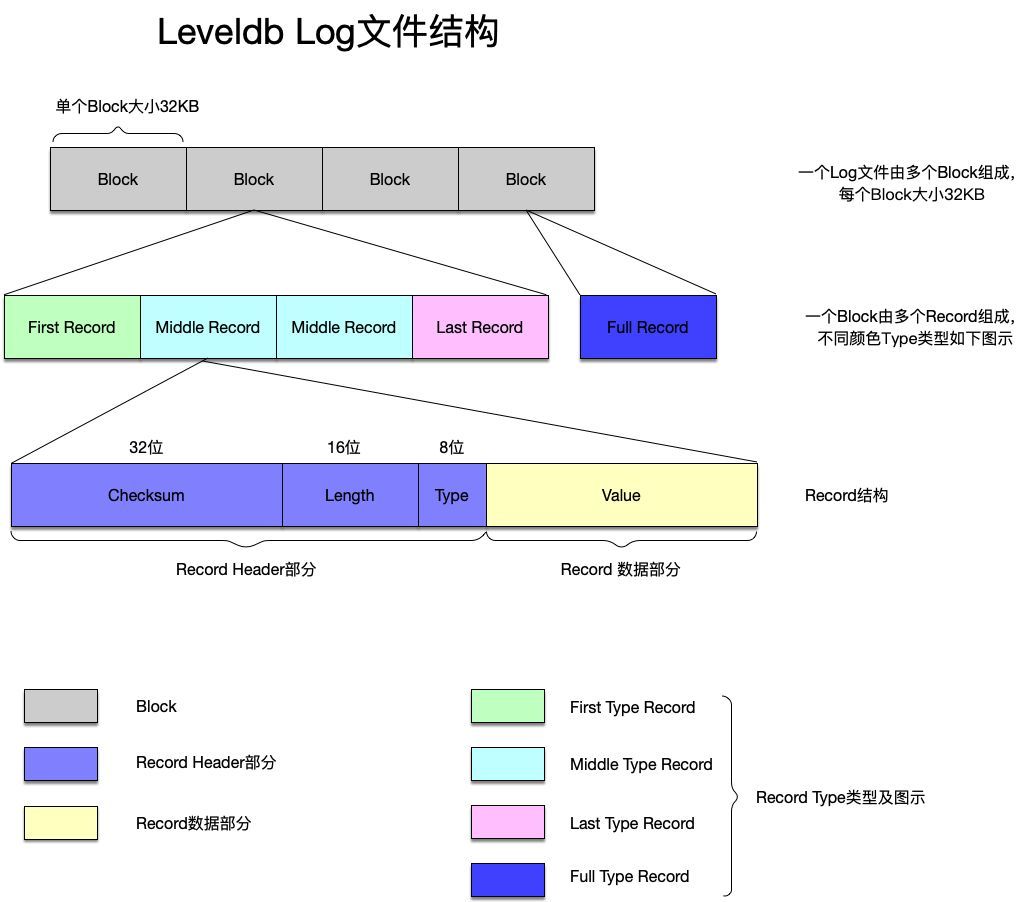
以下将针对这几部分展开讨论。

****Log文件****

写入数据的时候，最开始会写入到log文件中，由于是顺序写入文件，所以写入速度很快，可以马上返回。

来看Log文件的结构：

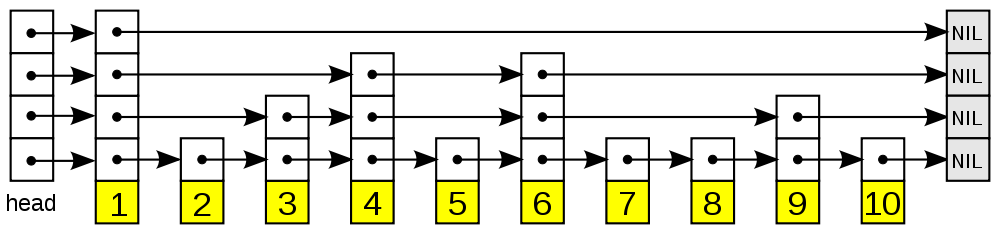
* 一个Log文件由多个Block组成，每个Block大小为32KB。
* 一个Block内部又有多个Record组成，Record分为四种类型：
  + Full：一个Record占满了整个Block存储空间。
  + First：一个Block的第一个Record。
  + Last：一个Block的最后一个Record。
  + Middle：其余的都是Middle类型的Record。
* Record的结构如下：
  + 32位长度的CRC Checksum：存储这个Record的数据校验值，用于检测Record合法性。
  + 16位长度的Length：存储数据部分长度。
  + 8位长度的Type：存储Record类型，就是上面说的四种类型。
  + Header部分
  + 数据部分



****memtable****

memtable用于存储在内存中还未落盘到sstable中的数据，这部分使用跳表（skiplist）做为底层的数据结构，这里先简单描述一下跳表的工作原理。

如果数据存放在一个普通的有序链表中，那么查找数据的时间复杂度就是O(n)。跳表的设计思想在于：链表中的每个元素，都有多个层次，查找某一个元素时，遍历该链表的时候，根据层次来跳过（skip）中间某些明显不满足需求的元素，以达到加快查找速度的目的，如下图所示：



在以上这个跳表中，查找元素6的流程，大体如下：

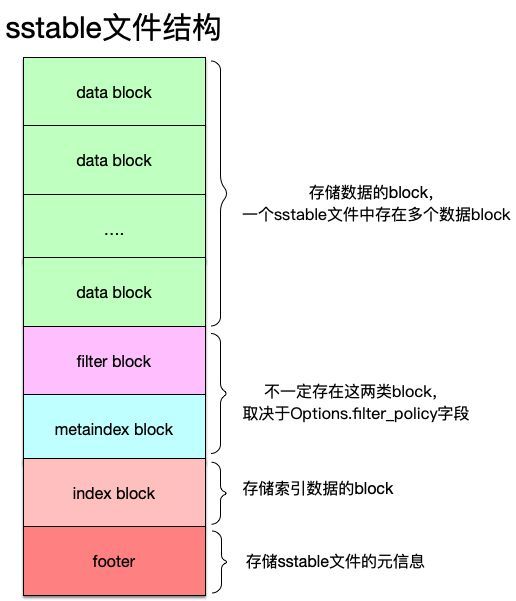
* 构建一个每个链表元素最多有5个元素的跳表。
* 由于6大于链表的第一个元素1，因此如果存在必然在1之后的元素中，因此进入元素1的指针数组中，从上往下查找元素4：
  + 第一层：指向的指针为Nil空指针，不满足需求，继续往下查找；
  + 第二层：指向的指针保存的数据为4，小于待查找的元素4，因此如果元素6存在也必然在4之后，因此指针跳转到元素4所在的位置，继续从上往下开始查找。
* 到了元素4所在的指针数组，开始从上往下继续查找：
  + 第一层：指向的指针保存的数据为6，查找完毕。

从上面的分析过程中可以看到：

* 跳表是一种以牺牲更多的存储空间换取查找速度，即“空间换时间”的数据结构。
* 跳表的每一层也都是一个有序链表。
* 如果一个元素出现在第i层的链表中，那么也必然会在第i层以下的链表中出现。
* 链表的每个节点中，垂直方向的数组存储的数据都是一样的，水平方向的指针指向链表的下一个元素。
* 最底层的链表包含所有元素，也就是说，在最底层数据结构退化为一个普通的有序链表。

****sstable文件**** ****大体结构****

首先来看sstable文件的整体结构，如下图：



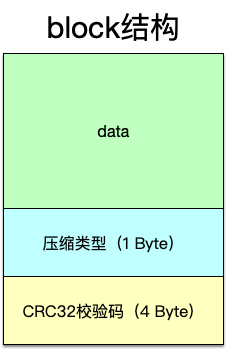
sstable文件中分为以下几个组成部分：

* data block：存储数据的block，由于一个block大小固定，因此每个sstable文件中有多个data block。
* filter block以及metaindex block：这两个block不一定存在于sstable，取决于Options中的filter\_policy参数值，后面就不对这两部分进行讲解。
* index block：存储的是索引数据，即可以根据index block中的数据快速定位到数据处于哪个data block的哪个位置。
* footer：脚注数据，每个footer数据信息大小固定，存储一个sstable文件的元信息（meta data）。

可以看到，上面这几部分数据，从文件的组织来看自上而下，先有了数据再有索引数据，最后才是文件自身的元信息。原因在于：索引数据是针对数据的索引信息，在数据没有写完毕之前，索引信息还会发生改变，所以索引数据要等数据写完；而元信息就是针对索引数据的索引，同样要依赖于索引信息写完毕才可以。

****block****

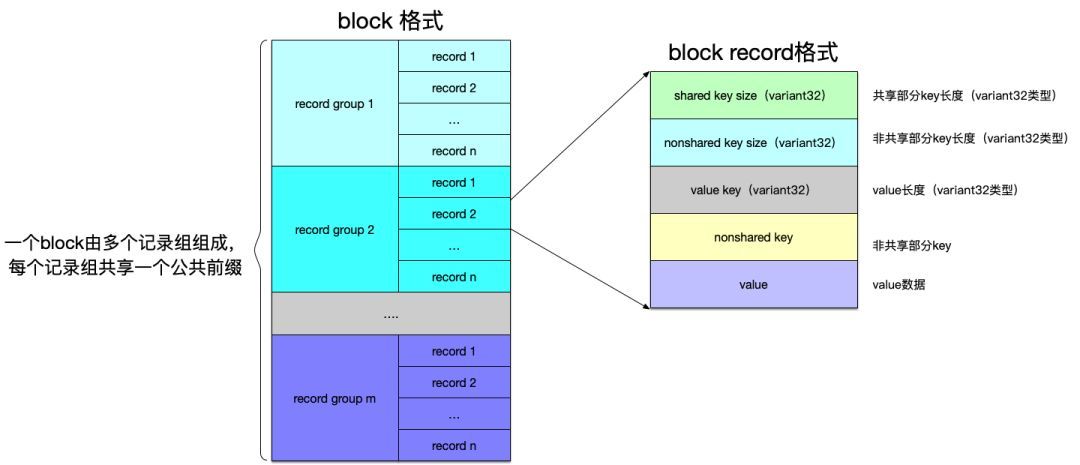
上面几部分数据中，除去footer之外，内部都是以block的形式来组织数据，接着看block的结构，如下图：



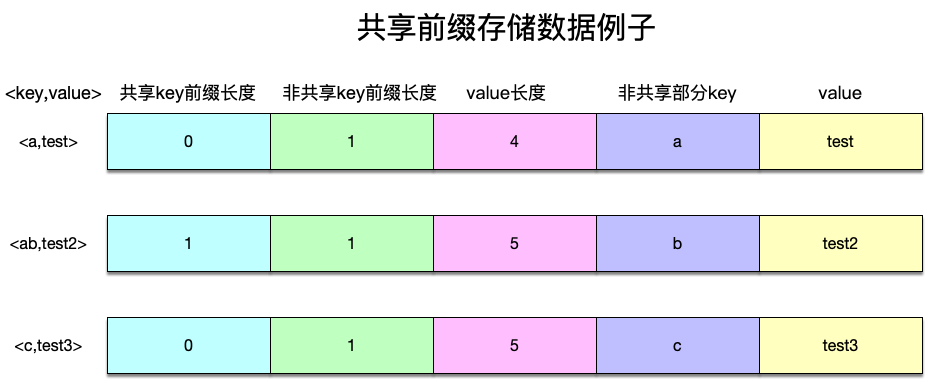
从上面看出，实际上存储数据的block大同小异：最开始的一部分存储数据，然后存储类型，最后一部分存储这个block的校验码以做合法性校验。

以上只是对block大体结构的分析，在数据部分，每一条数据记录leveldb使用前缀压缩（prefix-compressed）方式来存储。这种算法的原理是：针对一组数据，取出一个公共的前缀，而在该组中的其它字符串只保存非公共的字符串做为key即可，由于sstable保存KV数据是严格按照key的顺序来排序的，所以这样能节省出保存key数据的空间来。

如下图所示：一个block内部划分了多个记录组（record group），每个记录组内部又由多条记录（record）组成。在同一个记录组内部，以本组的第一条数据的键值做为公共前缀，后续的记录数据键值部分只存放与公共前缀非共享部分的数据即可。



以记录的三个数据<a,test>、<ab,test2>、<c,test3>为例，假设这三个数据在同一个record group内，那么对应的record记录如下所示：

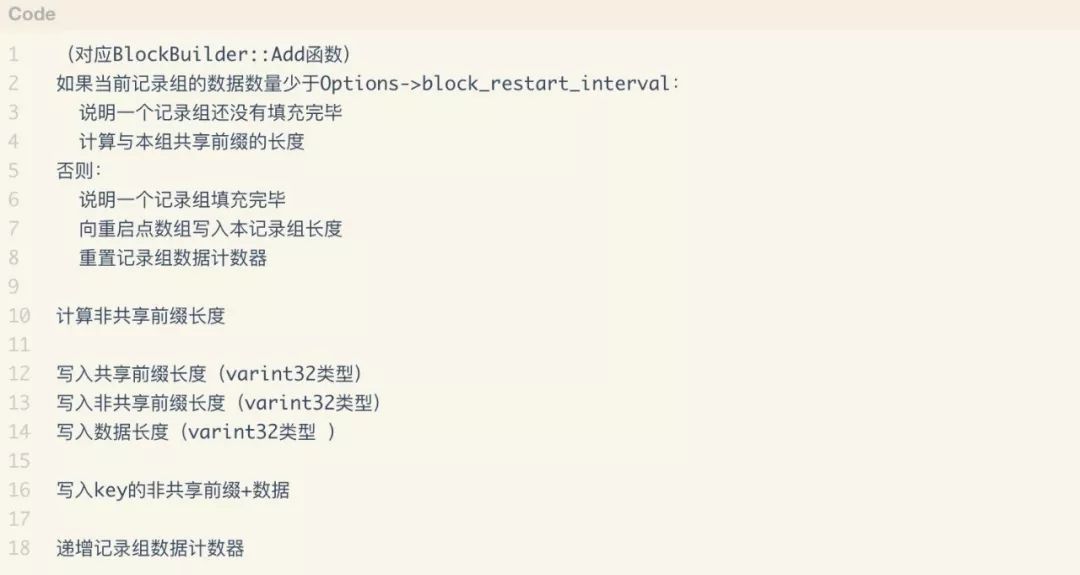


说明如下：

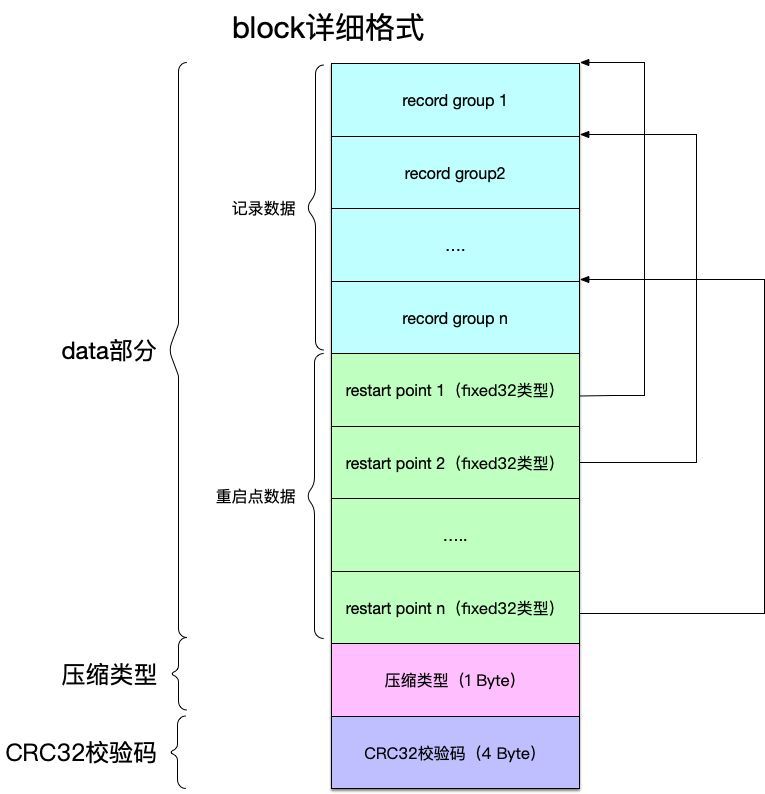
* <a,test>：由于这对数据是这一组记录组的第一组数据，因此共享的前缀部分长度为0，因为每一组都以第一条数据的key为共享前缀，因此针对第一条数据本身，其共享前缀长度就是0了。
* <ab,test2>：键值“ab”与本组公共前缀（即本组的第一条数据键值“a”）有公共前缀“a”，因此共享前缀长度为1，非共享部分键值为剩下的“b”。
* <c,test3>：键值“c”与本组公共前缀“a”没有重合部分，因此共享前缀长度为0。

因为一个block内部有多个记录组，因此还需要另外的数据来记录不同记录组的位置，这部分数据被称为“重启点（restart point）”，重启点首先会以数组的形式保存下来，直到该block数据需要落盘的情况下才会写到block结尾处。

有了以上的准备，就可以来具体看添加数据的代码了，向一个block中添加数据的伪代码如下。



有了前面的这些准备，再在前面block格式的基础上展开，得到更加详细的格式如下：



block详细格式还是划分为三大部分，其中：

* 数据部分
  + 多个记录组组成的记录数据。
  + 多个重启点数据组成的重启点数组数据，每个元素记录对应的记录组在block中的偏移量，类型为fixed32类型。
* 压缩类型，大小为1 Byte。
* CRC32校验数据，大小为4 Byte。

****footer****

footer做为存储sstable文件原信息部分的数据，格式相对简单，如下图：

****Iterator的设计****

迭代器的设计是leveldb中的一大亮点，leveldb设计了一个虚拟基类Iterator，其中定义了诸如遍历、查询之类的接口，而该基类有多种实现。原因在于：leveldb中存在多种数据结构和多种使用场景，如：

* 保存内存中数据的memtable。
* 保存落盘数据的sstable，而就前面分析而言，一个sstable中还有不同的block，需要根据index block来定位数据处于哪个data block。
* 进行合并的时候，每次最多合并两个层次的文件，在这个过程中需要对待合并的文件集合进行遍历。前面分析的DBImpl::DoCompactionWork函数，就是通过iterator来遍历待合并文件进行合并操作的。

逐个来看不同的iterator实现以及其使用场景。

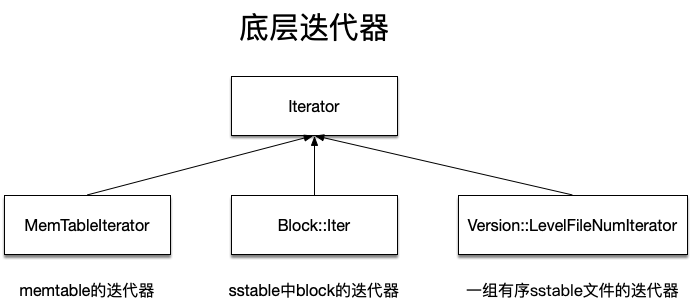
迭代器大体分为两类：

* 底层迭代器：处于最底层，直接访问底层数据结构，而不依赖于其他迭代器的迭代器。
* 组合迭代器：组合各种迭代器（包括底层和组合迭代器）完成工作的迭代器。

****底层迭代器****

底层迭代器有以下三种：

* MemTableIterator：用于实现对memtable的迭代遍历，由于memtable由skiplist实现，因此内部封装了对skiplist的迭代访问。
* Block::Iter：前面分析sstable的时候，讲到一个sstable内部其实有多个block组成，这个迭代器就是按照block的结构进行迭代访问的迭代器。
* Version::LevelFileNumIterator：每个level都由多个sstable文件组成，说白了就是一个sstable类型的数组。除了level 0之外，其余level的sstable的键值之间没有重叠关系，而LevelFileNumIterator就是用于迭代一组有序sstable文件的迭代器。



****组合迭代器****

组合迭代器内部都包含至少一个迭代器，组合起来完成迭代工作，有如下几类。

****TwoLevelIterator****

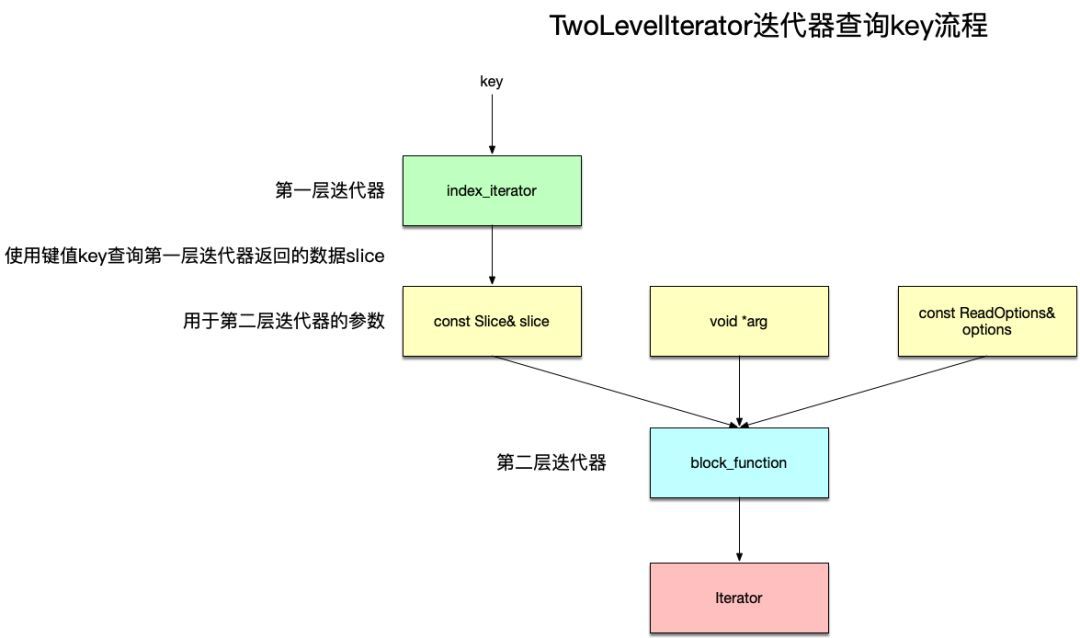
顾名思义，TwoLevelIterator表示两层迭代器，创建TwoLevelIterator的函数原型为：



参数说明如下：

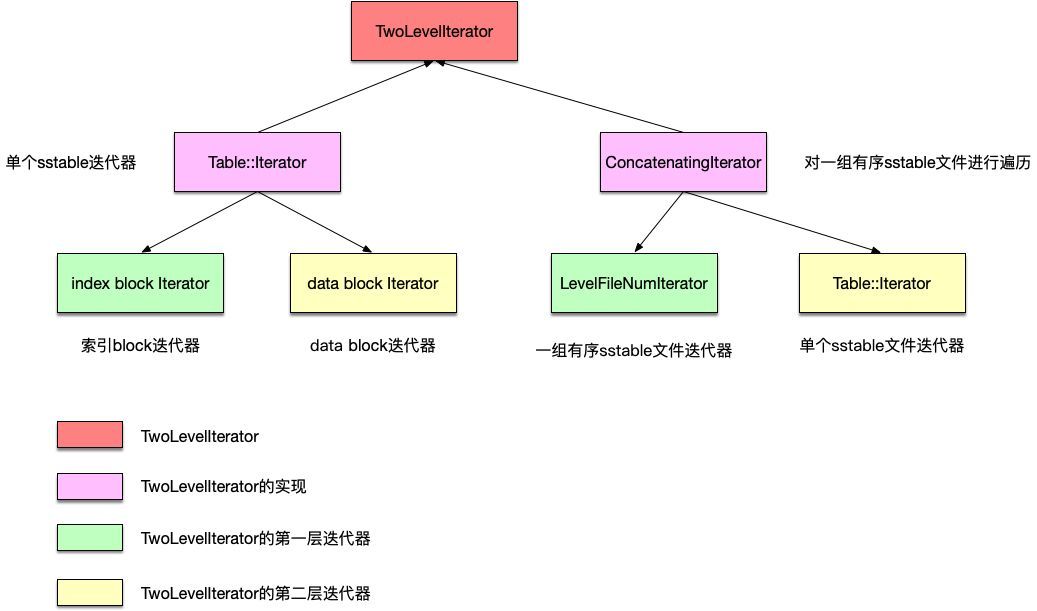
* Iterator\* index\_iter：索引迭代器，可以理解为第一层的迭代器。
* BlockFunction \*block\_function：这是一个函数指针，根据index\_iter迭代器的返回结果来再创建一个迭代器，即针对查询索引返回数据的迭代器。其函数参数有三个，其中前面两个由下面的arg以及options参数指定，而第三个参数slice就是index\_iterator返回的索引数据。
* void\* arg：传入BlockFunction函数的第一个参数。
* const ReadOptions& options：传入BlockFunction函数的第二个参数。

综合以上，TwoLevelIterator的工作流程如下：



TwoLevelIterator有如下两类：

* Table::Iterator：实现对于单个sstable文件的迭代。由于一个sstable文件中有多个block，而又划分为index block和data block，查询数据时，先根据键值到index block中查询到对应的data block，再进入data block中进行查询，这个查询过程实际就是一个两层的查找过程：先查索引数据，再查数据。因此Table::Iterator类型的TwoLevelIterator，组合了index block的Block::Iter，以及data block的Block::Iter。
* ConcatenatingIterator：组合了LevelFileNumIterator以及Table::Iterator，用于在某一层内的sstable文件中查询数据。因此它的第一层迭代器就是前面的LevelFileNumIterator，用于根据一个键值在一组有序的sstable文件中定位到所在的文件，而第二层的迭代器是Table::Iterator，用于在第一层迭代器LevelFileNumIterator中查询到的sstable文件中查询键值。另外，既然ConcatenatingIterator处理的是有序sstable文件，那么level 0的sstable文件就不会使用这种迭代器进行访问，因为level 0文件之间有重叠键值。



****MergingIterator****

用于合并流程的迭代器。在合并过程中，需要操作memtable、immutable memtable、level 0 sstable以及非level 0的sstable，该迭代器将这些存储数据结构体的迭代器，统一进行归并排序：

* memtable以及immutable memtable：使用前面提过的MemtableIterator迭代器。
* level 0 sstable：由于level 0的sstable文件之间键值有重叠，所以使用的是level 0的sstable文件各自的Table::Iterator。
* level 1层级及以上的sstable：使用前面介绍过的ConcatenatingIterator，即可以针对一组有序sstable文件进行遍历的iterator。

由于以上每种类型的iterator，内部遍历数据都是有序的，所以MergingIterator内部做的事情，就是将对这些iterator的遍历结果进行“归并”。MergingIterator内部有如下变量：

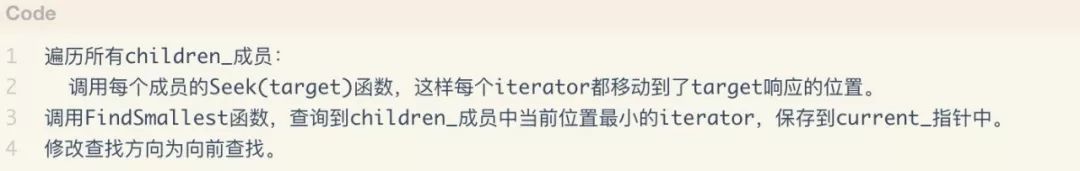
* const Comparator\* comparator\_：用于键值比较的函数operator。
* IteratorWrapper\* children\_：存储传入的iterator的数组。
* int n\_：children\_数组的大小。
* IteratorWrapper\* current\_：保存当前查询到的位置所在的iterator。
* Direction direction\_：保存查找的方向，有向前和向后两种查询防线。

可以看到，current\_以及direction\_两个成员是用于保存当前查找状态的成员。

构建MergingIterator迭代器传入的Comparator是InternalKeyComparator，其比较逻辑是：

* 首先比较键值是否相等，不相等则直接返回比较结果。
* 如果键值相等，那么从键值中decode出来sequence值，对比sequence值，对sequence值进行降序比较。由于这个值是单调递增的，因此越新的数据sequence值越大。换言之，在存储层次中（依次为memtable->immutable memtable->level 0 sstable->level n sstable）越靠上面的数据，在键值相同的情况下越小。

Seek(target)函数的伪代码：

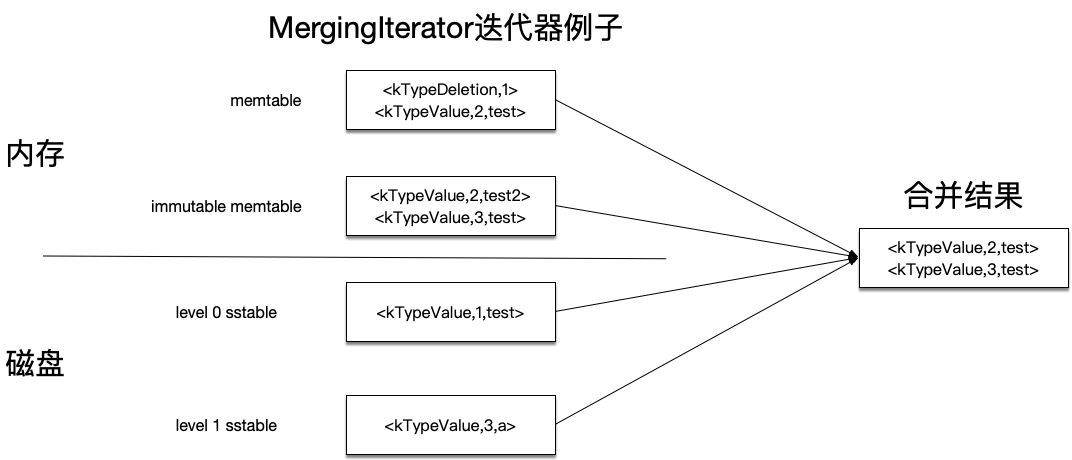


而FindSmallest函数的实现，是遍历children\_找到最小的child保存到current\_指针中。前面分析InternalKeyComparator提到过，相同键值的数据，根据sequence值进行降序排列，即数据越新的数据其sequence值越大，在这个排序中查找的结果就越在上面。因此，FindSmallest函数如果在memtable、level 0中都找到了相同键值，将优先选择memtable中的数据。

MergingIterator迭代器的其它实现不再做解释，简单理解：针对一组iterator的查询结果进行归并排序。对于同样一个键值，只取位置在存储位置上最靠上面的数据。

这么做的原因在于：一个键值的数据可能被写入多次，而需要以最后一次写入的数据为准，合并时将丢弃掉不在存储最上面的数据。

以下面的例子来说明MergingIterator迭代器的合并结果。



在上图的左半边，是合并前的数据分布情况，依次为：

* memtable：键值1的删除记录，以及键值<2,test>。
* immutable memtable：键值<2,tesat2>以及<3,test>。
* level 0 sstable：键值<1,test>。
* level 1 sstable：键值<3,a>。

合并的结果如上图的右边所示：

* 键1：因为第一条键1的记录是在memtable中的删除记录，所以键1将被删除，即不会出现在合并结果中。
* 键2：最靠上面的关于键2的存储记录是<2,test>，这条记录保存在合并结果中，而immutable memtable的记录<2,test2>将被丢弃，因为这条记录不是最新的。
* 键3：使用了immutable memtable中的记录<3,test>，丢弃了level 1 sstable中的<3,a>这条记录。

****核心流程****

有了前面几种核心数据结构的了解，下面谈leveldb中的几个核心流程。

****修改流程****

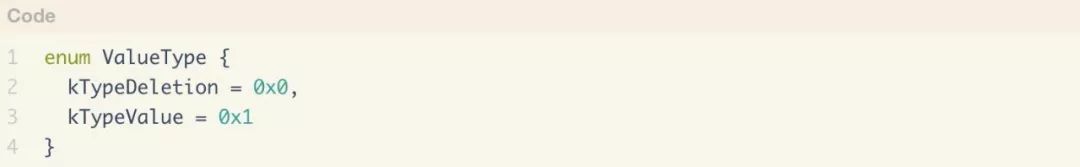
修改数据，分为两类：正常的写入数据操作以及删除数据操作。

先看正常的写入数据操作：

* append一条记录到log文件中，虽然这是一次写磁盘操作，但是由于是在文件末尾做的顺序写操作，所以效率并不低。
* 向当前的memtable中写入一条数据。这个动作看似简单，但是如果在来不及合并的时候，可能会出现阻塞，在后面合并操作中再展开解释。

完成以上两步之后，就可以认为完成了更新数据的操作。实际上只有一次文件末尾的顺序写操作，以及一次写内存操作，如果不考虑会被合并操作阻塞的情况，实际上还是很快的。

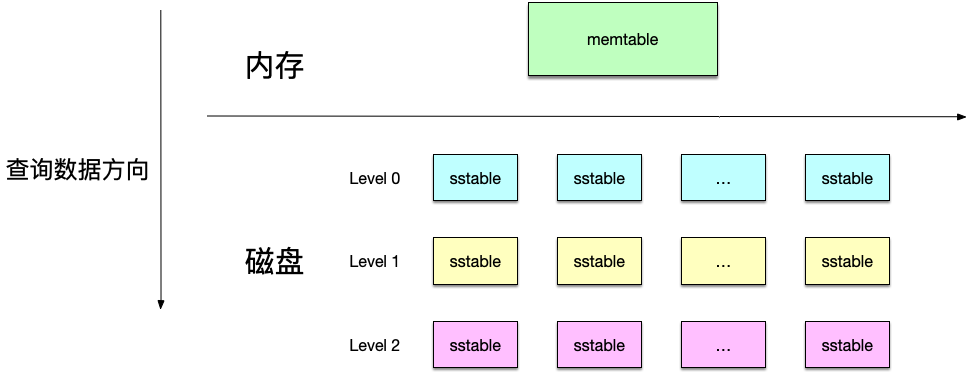
再来看删除数据操作。leveldb中，删除一个数据，其实也是添加一条新的记录，只不过记录类型是删除类型，代码中通过枚举变量定义了这两种操作类型：



这样看起来，leveldb删除数据时，并不会真的去删除一条数据，而是打上了一个标记，那么问题就来了：如果写入数据操作与删除数据操作，只是类型不同，在查询数据的时候又如何知道数据是否存在？看下面的读数据流程。

****读流程****

向leveldb中查询一个数据，也是从上而下，先查内存然后到磁盘上的sstable文件查询，如下图所示：



* 先在内存中的memtable中查询数据，查到则返回；
* 否则在磁盘中的sstable文件中查询数据，从0级开始往下查询，查到则返回；

这样自上而下的原因就在于leveldb的设计：越是在上层的数据越新，距离当前时间越短。

举例而言，对于键值key而言，首先写入kv对<key,data>，然后再删除键值key数据。第一次写入的数据，可能因为合并的原因以及到了sstable文件上，而再次删除键值key的数据时，根据上面的解释，其实也是写入数据，只不过标记为删除。于是，越后写入的数据，越在上面这个层次的上面，这样从上往下查询时就能先查找到后写入的数据，此时看到了数据已经被标记为删除，就可以认为数据不存在了。

那么，前面写入的数据实际上已经没有用了，但是又占用了空间，这部分数据就等待着后面的合并流程来合并数据最后删除掉。

****合并流程****

****核心数据结构****

首先来看与合并相关的核心数据结构。

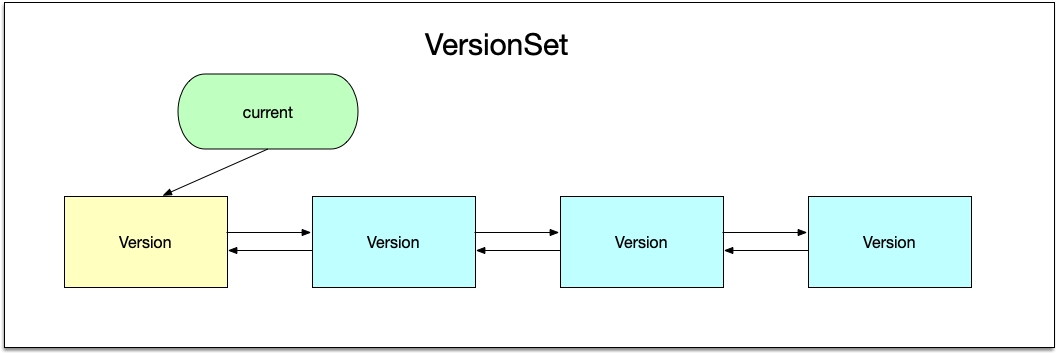
每一次合并过程以及将memtable中的数据落盘到sstable的过程，都涉及到sstable文件的增删，而这每一次操作，都对应到一个版本。

在leveldb中，使用Version类来存储一个版本的元信息，主要包括：

* std::vector files\_[config::kNumLevels]：用于存储所有级别sstable文件的FileMetaData数组，可以看到这个成员是一个数组，而数组的每个元素又是一个vector，这其中数组部分使用level级别来进行索引，同级别的sstable信息存储在vector 中。
* FileMetaData\* file\_to\_compact\_和int file\_to\_compact\_level\_：下一次进行合并时的文件和级别。
* double compaction\_score\_和int compaction\_level\_：当前最大compact分数和对应的级别，在Finalize函数中进行计算，具体计算的规则会在下面介绍。

可以看到，Version保存的主要是两类数据：当前sstable文件的信息，以及下一次合并时的信息。

所有的级别，也就是Version类，使用双向链表串联起来，保存到VersionSet中，而VersionSet中有一个current指针，用于指向当前的Version。

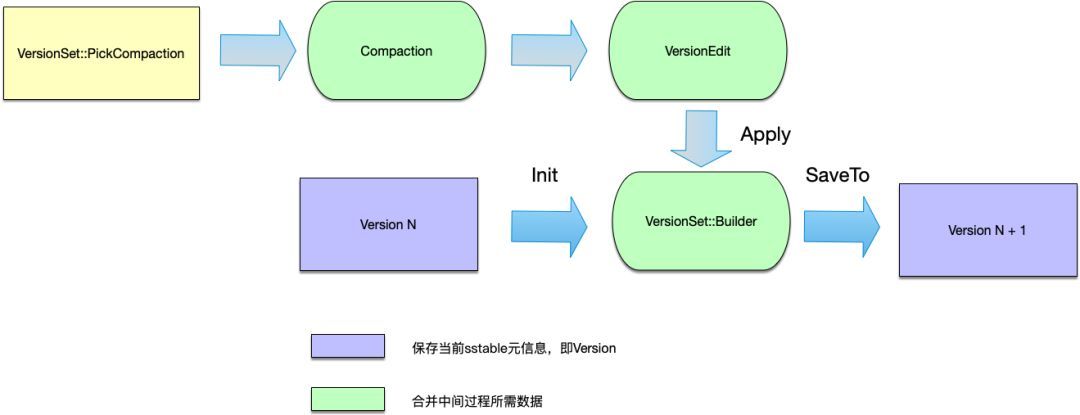


当进行合并时，首先需要挑选出需要进行合并的文件，这个操作的入口函数是VersionSet::PickCompaction，该函数的返回值是一个Compaction结构体，该结构体内部保存合并相关的信息，Compaction结构体中最重要的成员是VersionEdit类成员，这个成员用于保存合并过程中有删减的sstable文件：

* DeletedFileSet deleted\_files\_：合并后待删除的sstable文件。
* std::vector< std::pair > new\_files\_：合并后新增的sstable文件。

可以认为：version N + version N edit = version N + 1，即：第N版本的sstable信息，在经过该版本合并的VersionEdit之后，形成了Version N+1版本。

另外还有一个VersionSet::Builder，用于保存合并中间过程的数据，其本质是将所有VersoinEdit内容存储到VersionSet::Builder中，最后一次产生新版本的Version。



****合并条件及原理****

leveldb会不断产生新的sstable文件，这时候需要对这些文件进行合并，否则磁盘会一直增大，查询速度也会下降。

这部分讲解合并触发的条件以及进行合并的原理。

leveldb大致在以下两种条件下会触发合并操作：

* 需要新生成memtable的情况下，此时必然会把原来的memtable切换为immutable memtable，后者又需要及时落盘成为新的sstable文件，将immutable memtable数据落盘为sstable文件的流程称为”minor compaction“，因为有新的sstable文件产生，所以需要合并文件减少sstable文件的数量。
* 查询数据时，某些sstable总是查找不到数据，此时可能是因为数据太过分散了，也需要将文件合并。

以上两种情况，对应到leveldb代码中就是以下几个地方：

* 调用DB::Open文件打开数据库文件时，由于之前可能已经存在了一些文件，这时会做检查，满足条件的情况下会进行合并操作。
* 调用DB::Write函数写入数据时，调用MakeRoomForWrite函数分配空间，此时如果需要新分配一个memtable，也会触发合并操作。
* 调用DB::Get函数查询数据时，某些文件查询的次数超过了阈值，此时也会进行合并操作。

另外还需要提一下合并的两种类型：

* minor compaction：将内存的数据落地到磁盘上的迁移过程，对应于leveldb就是将immutable memtable数据落盘为sstable文件的流程。
* major compaction：sstable之间的文件进行合并的流程。

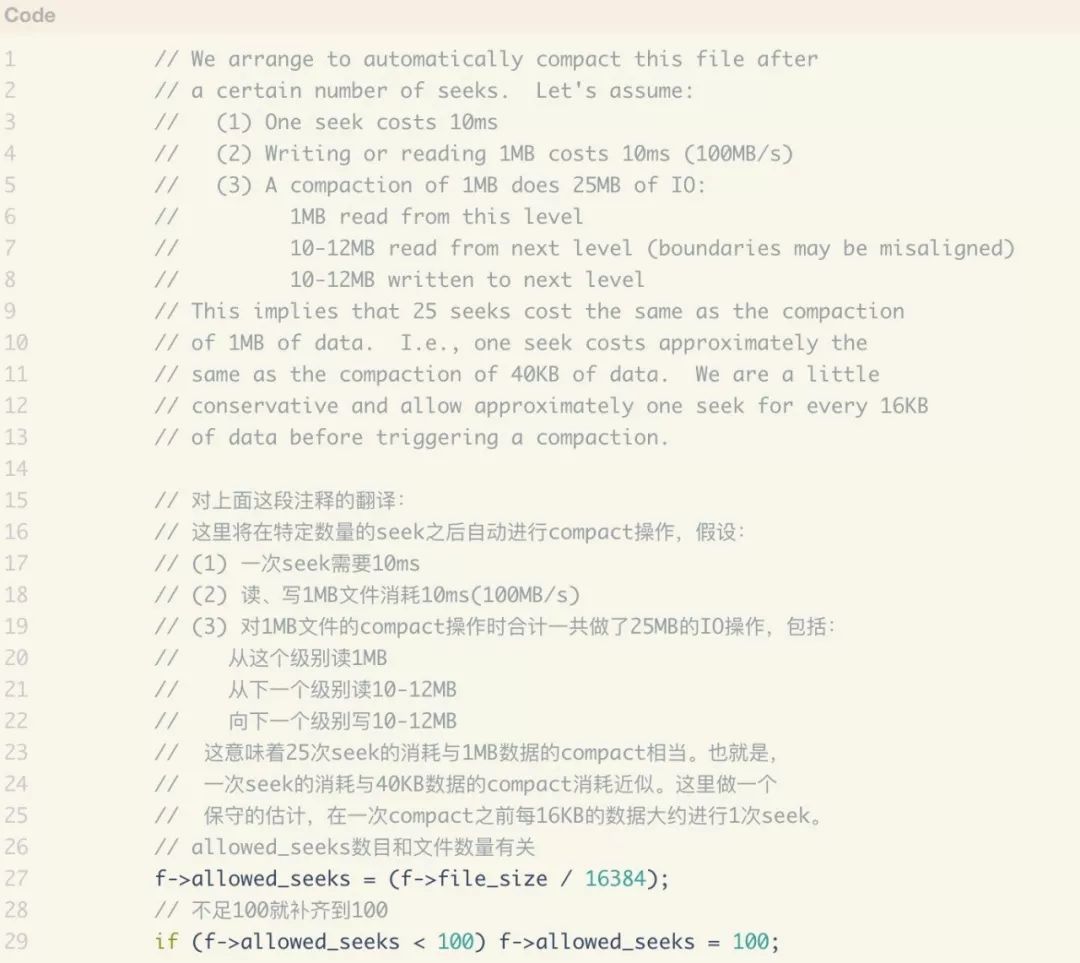
****选择进行合并的文件****

函数VersionSet::PickCompaction用于构建出一次合并对应的Compaction结构体。来看这个函数主要的流程。

在挑选哪些文件需要合并时，依赖于两个原则：

* 首先考虑每一层文件的数量：这个数量的计数，对应到Version的compaction\_score\_中，在每次VersionSet::Finalize函数中，都会首先进行预计算这个值，那个级别的分数高，下一次就优先选择该层次来做合并。对于不同的层次，计算的规则也不同：
  + level 0：0级文件的数量除以kL0\_CompactionTrigger来计算分数。
  + 非0级：该层次的所有文件大小/MaxBytesForLevel(level)来计算分数。
* 如果上面的计算中，compaction\_score\_为0，那么就需要具体针对一个文件来进行合并。leveldb中，在FileMetaData结构体里有一个成员allowed\_seeks，表示在该文件中查询某个键值时最多允许的定位次数，当这个值为0时，意味这个文件多次查询都没有查询到数据，因此这个文件就需要进行合并了。

文件的allowed\_seeks在VersionSet::Builder::Apply函数中进行计算：



如果是第一种情况，即compaction\_score\_ >= 1的情况来选择合并文件，还涉及到一个合并点的问题（compact point），即leveldb会保存上一次进行合并的键值，这一次会从这个键值以后开始寻找需要进行合并的文件。

而如果合并层次是0级，因为0级文件中的键值有重叠的情况，因此还需要遍历0级文件中键值范围与这次合并文件由重叠的一起加入进来。

在这之后，调用VersionSet::SetupOtherInputs函数，用于获取同级别以及更上一层也就是level + 1级别中满足合并范围条件的文件，这就构成了待合并的文件集合。



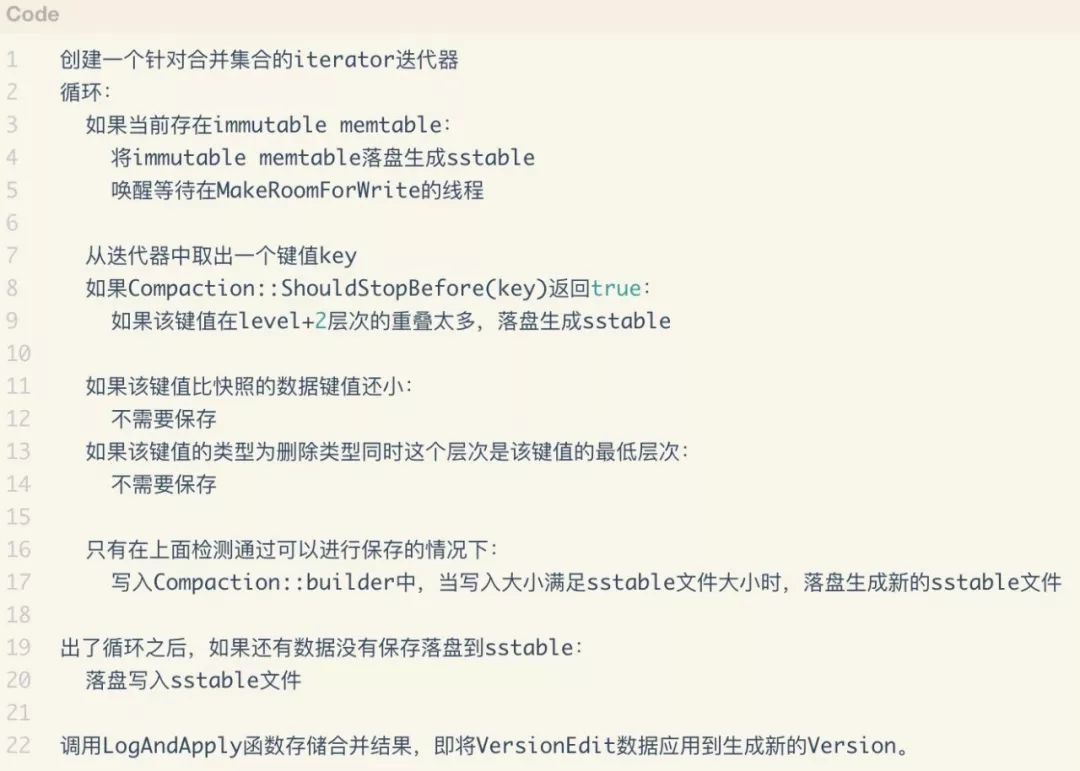
如上图所示：

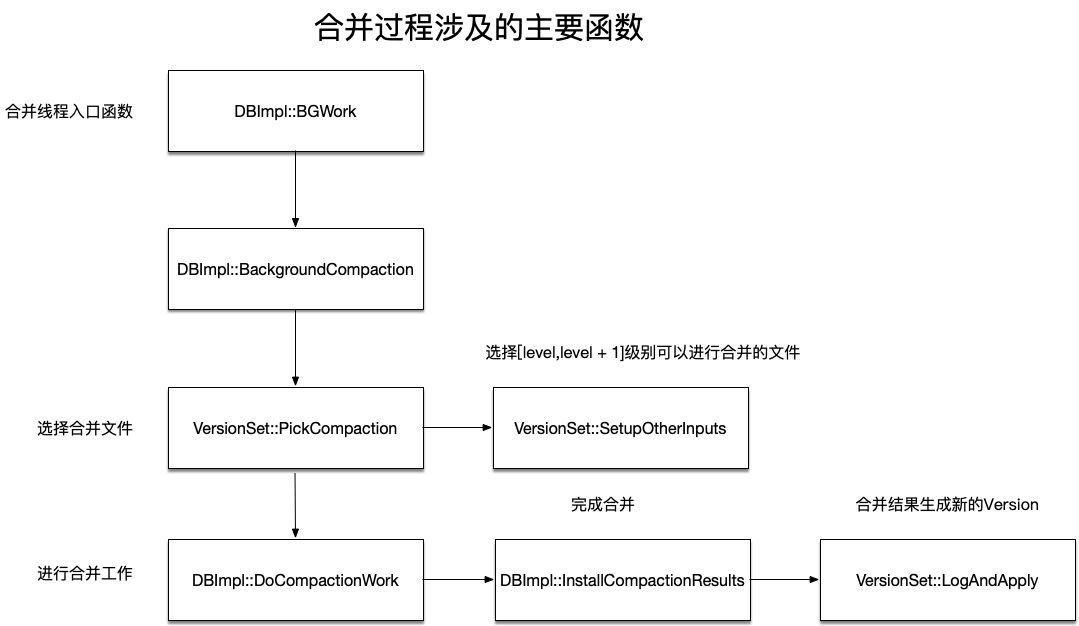
* 此时选择进行合并的文件，其键值是[1,2,4,5]。
* 由于该文件在level 0级别，sstable文件的键值有重叠，同时还在在其上面一层选择同样键值范围有重叠的sstable文件，选择的结果就是绿色的sstable文件，这些将做为这次合并进行归并排序的文件。

****合并流程****

以上调用VersionSet::PickCompaction函数选择完毕了待合并的文件及层次之后，就来到DBImpl::DoCompactionWork函数中进行实际的合并工作。

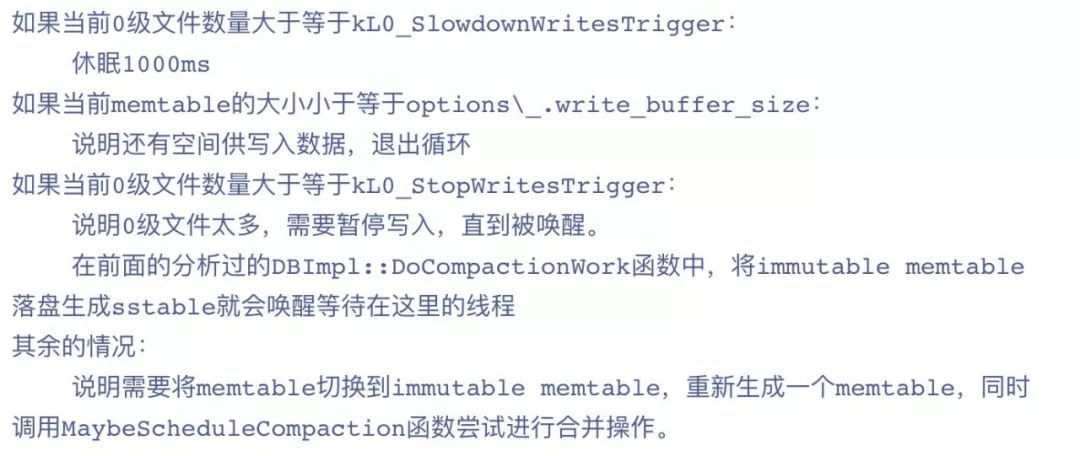
该函数的原理不算复杂，就是遍历文件集合进行合并：





****合并操作对读写流程的影响****

leveldb将用户的读写操作，与合并工作放在不同的线程中处理。当用户需要写入数据进行分配空间时，会首先调用DBImpl::MakeRoomForWrite函数进行空间的分配，该函数有可能因为合并流程被阻塞，有以下几种可能性：



****参考资料****

* 数据分析与处理之二（Leveldb 实现原理）：https://www.cnblogs.com/haippy/archive/2011/12/04/2276064.html
* Leveldb之Iterator总结：http://kernelmaker.github.io/Leveldb\_Iterator