前文有述，对于LevelDb来说，写入记录操作很简单，**删除记录仅仅写入一个删除标记就算完事了**，但是读取记录比较复杂，需要在内存以及各个层级文件中依照新鲜程度依次查找，代价很高。为了加快读取速度，levelDb采取了compaction的方式来对已有的记录进行整理压缩，通过这种方式，来删除掉一些不再有效的KV数据，减小数据规模，减少文件数量等。

     levelDb的compaction机制和过程与Bigtable所讲述的是基本一致的，Bigtable中讲到三种类型的compaction: minor ，major和full。所谓minor Compaction，就是把memtable中的数据导出到SSTable文件中；major compaction就是合并不同层级的SSTable文件，而full compaction就是将所有SSTable进行合并。

     LevelDb包含其中两种，minor和major。

    先来看看minor Compaction的过程。Minor compaction 的目的是当内存中的memtable大小到了一定值时，将内容保存到磁盘文件中，图8.1是其机理示意图。

图8.1 minor compaction

     从8.1可以看出，当memtable数量到了一定程度会转换为immutable memtable，此时不能往其中写入记录，只能从中读取KV内容。之前介绍过，immutable memtable其实是一个多层级队列SkipList，其中的记录是根据key有序排列的。所以这个minor compaction实现起来也很简单，就是按照immutable memtable中记录由小到大遍历，并依次写入一个level 0 的新建SSTable文件中，写完后建立文件的index 数据，这样就完成了一次minor compaction。从图中也可以看出，对于被删除的记录，在minor compaction过程中并不真正删除这个记录，原因也很简单，这里只知道要删掉key记录，但是这个KV数据在哪里?那需要复杂的查找，**所以在minor compaction的时候并不做删除，只是将这个key作为一个记录写入文件中，至于真正的删除操作，在以后更高层级的compaction中会去做。**

     当某个level下的SSTable文件数目超过一定设置值后，levelDb会从这个level的SSTable中选择一个文件（level>0），将其和高一层级的level+1的SSTable文件合并，这就是major compaction。

    我们知道**在大于0的层级中，每个SSTable文件内的Key都是由小到大有序存储的，而且不同文件之间的key范围（文件内最小key和最大key之间）不会有任何重叠。**L**evel 0的SSTable文件有些特殊，尽管每个文件也是根据Key由小到大排列，但是因为level 0的文件是通过minor compaction直接生成的，所以任意两个level 0下的两个sstable文件可能再key范围上有重叠。**所以在做major compaction的时候，对于大于level 0的层级，选择其中一个文件就行，但是对于level 0来说，指定某个文件后，本level中很可能有其他SSTable文件的key范围和这个文件有重叠，这种情况下，要找出所有有重叠的文件和level 1的文件进行合并，即**level 0在进行文件选择的时候，可能会有多个文件参与major compaction。**

    levelDb在选定某个level进行compaction后，还要选择是具体哪个文件要进行compaction，levelDb在这里有个小技巧，就是说轮流来，比如这次是文件A进行compaction，那么下次就是在key range上紧挨着文件A的文件B进行compaction，这样每个文件都会有机会轮流和高层的level 文件进行合并。

    如果选好了level L的文件A和level L+1层的文件进行合并，那么问题又来了，应该选择level L+1哪些文件进行合并？levelDb选择L+1层中和文件A在key range上有重叠的所有文件来和文件A进行合并。

     也就是说，选定了level L的文件A,之后在level L+1中找到了所有需要合并的文件B,C,D…..等等。剩下的问题就是具体是如何进行major 合并的？就是说给定了一系列文件，每个文件内部是key有序的，如何对这些文件进行合并，使得新生成的文件仍然Key有序，同时抛掉哪些不再有价值的KV数据。

    图8.2说明了这一过程。

                          图8.2 SSTable Compaction

     Major compaction的过程如下：**对多个文件采用多路归并排序的方式**，依次找出其中最小的Key记录，也就是对多个文件中的所有记录重新进行排序。之后采取一定的标准判断这个Key是否还需要保存，如果判断没有保存价值，那么直接抛掉，如果觉得还需要继续保存，那么就将其写入level L+1层中新生成的一个SSTable文件中。就这样对KV数据一一处理，形成了一系列新的L+1层数据文件，之前的L层文件和L+1层参与compaction 的文件数据此时已经没有意义了，所以全部删除。这样就完成了L层和L+1层文件记录的合并过程。

    那么在major compaction过程中，判断一个KV记录是否抛弃的标准是什么呢？其中一个标准是:对于某个key来说，如果在小于L层中存在这个Key，那么这个KV在major compaction过程中可以抛掉。因为我们前面分析过，对于层级低于L的文件中如果存在同一Key的记录，那么说明对于Key来说，有更新鲜的Value存在，那么过去的Value就等于没有意义了，所以可以删除。

    是的，compacition的过程就是如此简单

* [博客](http://blog.csdn.net/?ref=toolbar)
* [学院](http://edu.csdn.net/?ref=toolbar)
* [下载](http://download.csdn.net/?ref=toolbar)
* [更多](javascript:;)
* 
* [写博客](http://write.blog.csdn.net/postedit?ref=toolbar)
* [登录](https://passport.csdn.net/account/login?ref=toolbar)[注册](http://passport.csdn.net/account/mobileregister?ref=toolbar&action=mobileRegister)

**LevelDB详解**

转载 2016年10月09日 16:42:20

* 11003

[LevelDB](http://www.cnblogs.com/zhangeamon/p/5810400.html)

一、LevelDB入门

LevelDB是Google开源的持久化KV单机数据库，具有很高的随机写，顺序读/写性能，但是随机读的性能很一般，也就是说，LevelDB很适合应用在查询较少，而写很多的场景。LevelDB应用了[LSM](http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.44.2782&rep=rep1&type=pdf" \t "_blank) (Log Structured Merge) 策略，lsm\_tree对索引变更进行延迟及批量处理，并通过一种类似于归并排序的方式高效地将更新迁移到磁盘，降低索引插入开销，关于LSM，本文在后面也会简单提及。

根据[Leveldb官方网站](https://github.com/google/leveldb" \t "_blank)的描述，LevelDB的特点和限制如下：

**特点：**  
1、key和value都是任意长度的字节数组；  
2、entry（即一条K-V记录）默认是按照key的字典顺序存储的，当然开发者也可以重载这个排序函数；  
3、提供的基本操作接口：Put()、Delete()、Get()、Batch()；  
4、支持批量操作以原子操作进行；  
5、可以创建数据全景的snapshot(快照)，并允许在快照中查找数据；  
6、可以通过前向（或后向）迭代器遍历数据（迭代器会隐含的创建一个snapshot）；  
7、自动使用Snappy压缩数据；  
8、可移植性；

**限制：**  
1、非关系型数据模型（NoSQL），不支持sql语句，也不支持索引；  
2、一次只允许一个进程访问一个特定的数据库；  
3、没有内置的C/S架构，但开发者可以使用LevelDB库自己封装一个server；

 python 版示例

安装依赖

pip install  leveldb

[?](http://www.cnblogs.com/zhangeamon/p/5810400.html)

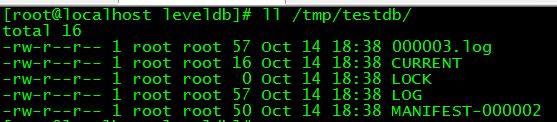
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | python 2.7 示例<br>>>> import leveldb  >>> db = leveldb.LevelDB('./db')  >>> db.Put('hello', 'world')  >>> print db.Get('hello')  world  >>> db.Delete('hello')  >>> db.Get('hello')  Traceback (most recent call last):    File "<stdin>", line 1, in <module>  KeyError  >>> for i in xrange(10):  ...   db.Put(str(i), 'string\_%s' % i)  ...  >>> print list(db.RangeIter(key\_from = '2', key\_to = '5'))  [('2', 'string\_2'), ('3', 'string\_3'), ('4', 'string\_4'), ('5', 'string\_5')]  >>> batch = leveldb.WriteBatch()  >>> for i in xrange(1000):  ...   db.Put(str(i), 'string\_%s' % i)  ...  >>> db.Write(batch, sync = True)<br><br>python 3.5 将 字符串进行编码 如 'key'.encode() ,''.decode() <br><br>用于python3 sock通信<br> |

插入数据示例

[?](http://www.cnblogs.com/zhangeamon/p/5810400.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | import leveldb  db = leveldb.LevelDB('./db')  batch = leveldb.WriteBatch()  for i in xrange(1000000):  ...     db.Put(str(i),'string\_%s' % i)  ...  db.Write(batch,sync=True) |

下面生成一个目录db，里面包含若干文件：



然后简要说下各个文件的含义：

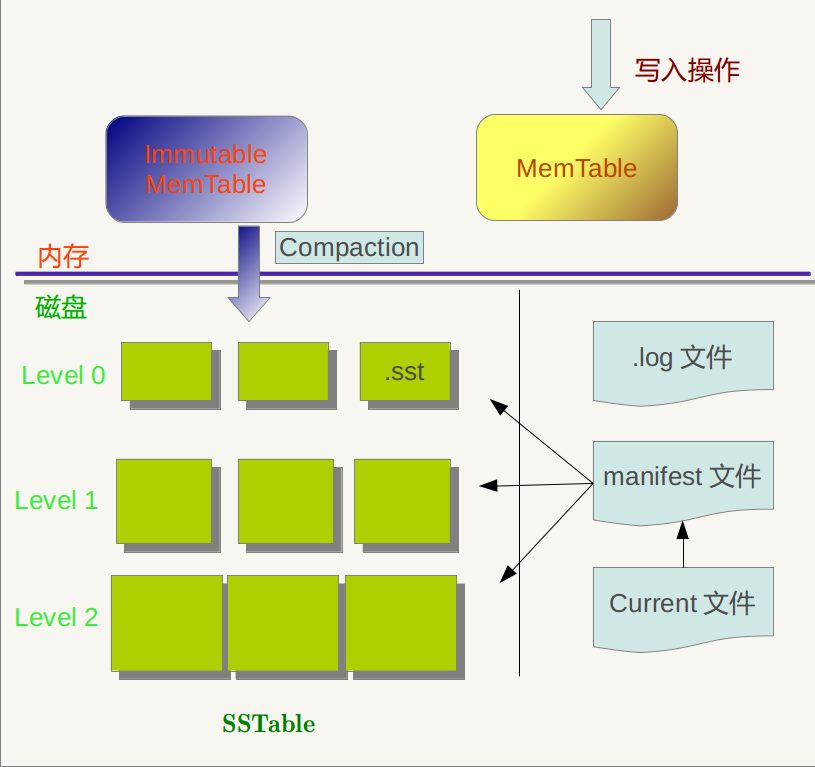
1、CURRENT

2、LOG

3、LOCK

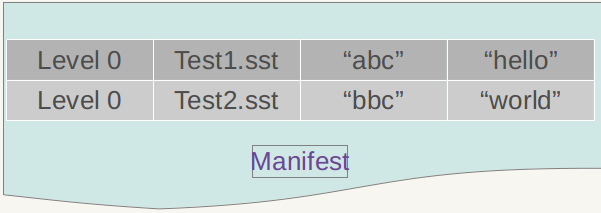
4、MANIFEST

下图是LevelDB运行一段时间后的存储模型快照：内存中的MemTable和Immutable MemTable以及磁盘上的几种主要文件：Current文件，Manifest文件，log文件以及SSTable文件。当然，LevelDb除了这六个主要部分还有一些辅助的文件，但是以上六个文件和数据结构是LevelDb的主体构成元素。



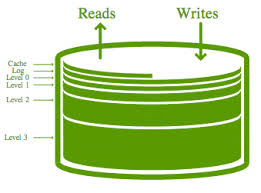
log文件、MemTable、SSTable文件都是用来存储k-v记录的，下面再说说manifest和Current文件的作用。

SSTable中的某个文件属于特定层级，而且其存储的记录是key有序的，那么必然有文件中的最小key和最大key，这是非常重要的信息，Manifest 就记载了SSTable各个文件的管理信息，比如属于哪个Level，文件名称叫啥，最小key和最大key各自是多少。下图是Manifest所存储内容的示意：



另外，在LevleDb的运行过程中，随着Compaction的进行，SSTable文件会发生变化，会有新的文件产生，老的文件被废弃，Manifest也会跟着反映这种变化，此时往往会新生成Manifest文件来记载这种变化，而Current则用来指出哪个Manifest文件才是我们关心的那个Manifest文件。

二、读写数据

****

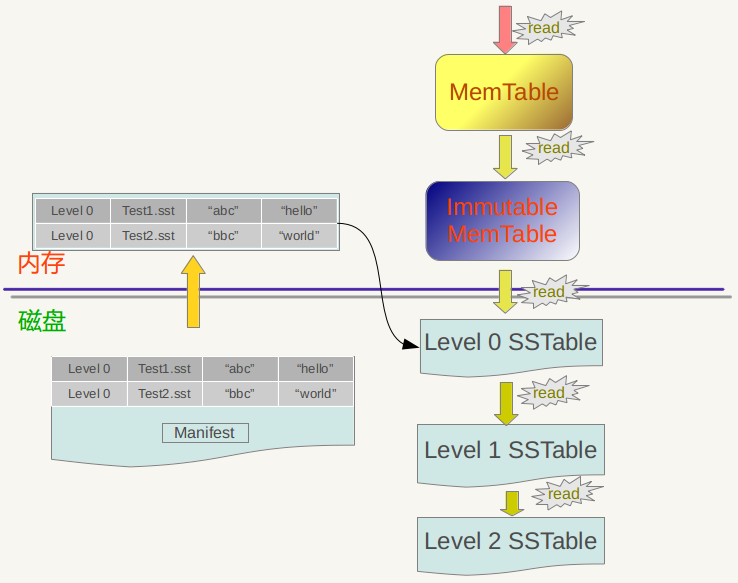
**写操作流程：**

1、顺序写入磁盘log文件；  
2、写入内存memtable（采用skiplist结构实现）；  
3、写入磁盘SST文件(sorted string table files)，这步是数据归档的过程（永久化存储）；

注意：

* log文件的作用是是用于系统崩溃恢复而不丢失数据，假如没有Log文件，因为写入的记录刚开始是保存在内存中的，此时如果系统崩溃，内存中的数据还没有来得及Dump到磁盘，所以会丢失数据；
* 在写memtable时，如果其达到check point（满员）的话，会将其改成immutable memtable（只读），然后等待dump到磁盘SST文件中，此时也会生成新的memtable供写入新数据；
* memtable和sst文件中的key都是有序的，log文件的key是无序的；
* LevelDB删除操作也是插入，只是标记Key为删除状态，真正的删除要到Compaction的时候才去做真正的操作；
* LevelDB没有更新接口，如果需要更新某个Key的值，只需要插入一条新纪录即可；或者先删除旧记录，再插入也可；

**读操作流程：**  
1、在内存中依次查找memtable、immutable memtable；  
2、如果配置了cache，查找cache；  
3、根据mainfest索引文件，在磁盘中查找SST文件；



举个例子：我们先往levelDb里面插入一条数据 {key="www.samecity.com"  value="我们"}，过了几天，samecity网站改名为：69同城，此时我们插入数据{key="www.samecity.com"  value="69同城"}，同样的key,不同的value；逻辑上理解好像levelDb中只有一个存储记录，即第二个记录，但是在levelDb中很可能存在两条记录，即上面的两个记录都在levelDb中存储了，此时如果用户查询key="www.samecity.com"，我们当然希望找到最新的更新记录，也就是第二个记录返回，因此，查找的顺序应该依照数据更新的新鲜度来，对于SSTable文件来说，如果同时在level L和Level L+1找到同一个key，level L的信息一定比level L+1的要新。

三、SSTable文件

SST文件并不是平坦的结构，而是分层组织的，这也是LevelDB名称的来源。

**SST文件的一些实现细节：**

1、每个SST文件大小上限为2MB，所以，LevelDB通常存储了大量的SST文件；  
2、SST文件由若干个4K大小的blocks组成，block也是读/写操作的最小单元；  
3、SST文件的最后一个block是一个index，指向每个data block的起始位置，以及每个block第一个entry的key值（block内的key有序存储）；  
4、使用[Bloom filter](http://www.cnblogs.com/chenny7/p/4074250.html" \t "_blank)加速查找，只要扫描index，就可以快速找出所有可能包含指定entry的block。  
5、同一个block内的key可以共享前缀（只存储一次），这样每个key只要存储自己唯一的后缀就行了。如果block中只有部分key需要共享前缀，在这部分key与其它key之间插入"reset"标识。

由log直接读取的entry会写到Level 0的SST中（最多4个文件）；

当Level 0的4个文件都存储满了，会选择其中一个文件Compact到Level 1的SST中；

注意：Level 0的SSTable文件和其它Level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，比如有两个level 0的sst文件，文件A和文件B，文件A的key范围是：{bar, car}，文件B的Key范围是{blue,samecity}，那么很可能两个文件都存在key=”blood”的记录。对于其它Level的SSTable文件来说，则不会出现同一层级内.sst文件的key重叠现象，就是说Level L中任意两个.sst文件，那么可以保证它们的key值是不会重叠的。

Log：最大4MB (可配置), 会写入Level 0；  
Level 0：最多4个SST文件,；  
Level 1：总大小不超过10MB；  
Level 2：总大小不超过100MB；  
Level 3+：总大小不超过上一个Level ×10的大小。

比如：0 ↠ 4 SST, 1 ↠ 10M, 2 ↠ 100M, 3 ↠ 1G, 4 ↠ 10G, 5 ↠ 100G, 6 ↠ 1T, 7 ↠ 10T

在读操作中，要查找一条entry，先查找log，如果没有找到，然后在Level 0中查找，如果还是没有找到，再依次往更底层的Level顺序查找；如果查找了一条不存在的entry，则要遍历一遍所有的Level才能返回"Not Found"的结果。

在写操作中，新数据总是先插入开头的几个Level中，开头的这几个Level存储量也比较小，因此，对某条entry的修改或删除操作带来的性能影响就比较可控。

可见，SST采取分层结构是为了最大限度减小插入新entry时的开销；

**Compaction操作**

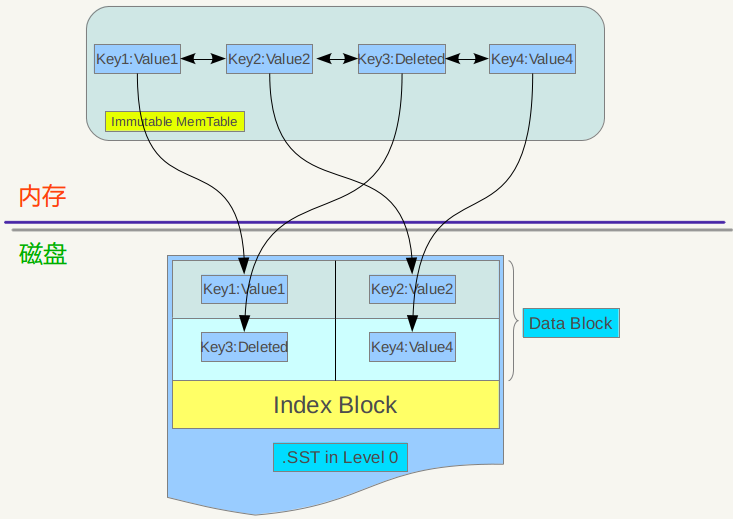
对于LevelDb来说，写入记录操作很简单，删除记录仅仅写入一个删除标记就算完事，但是读取记录比较复杂，需要在内存以及各个层级文件中依照新鲜程度依次查找，代价很高。为了加快读取速度，levelDb采取了compaction的方式来对已有的记录进行整理压缩，通过这种方式，来删除掉一些不再有效的KV数据，减小数据规模，减少文件数量等。

LevelDb的compaction机制和过程与Bigtable所讲述的是基本一致的，Bigtable中讲到三种类型的compaction: minor ，major和full：

* minor Compaction，就是把memtable中的数据导出到SSTable文件中；
* major compaction就是合并不同层级的SSTable文件；
* full compaction就是将所有SSTable进行合并；

LevelDb包含其中两种，minor和major。

Minor compaction 的目的是当内存中的memtable大小到了一定值时，将内容保存到磁盘文件中，如下图：



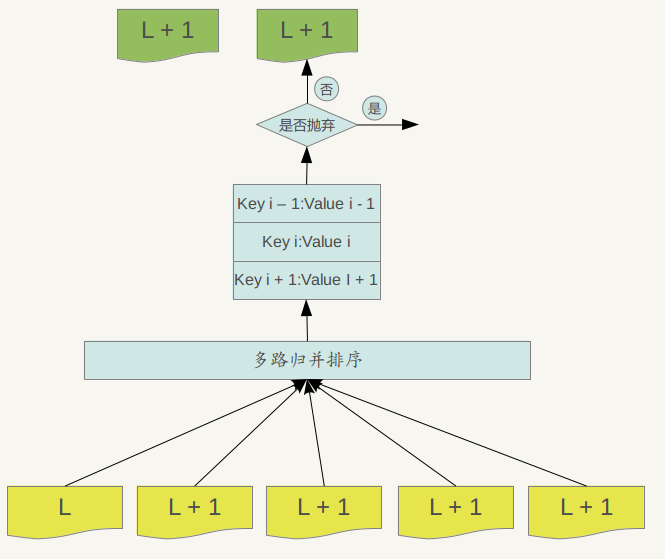
immutable memtable其实是一个SkipList，其中的记录是根据key有序排列的，遍历key并依次写入一个level 0 的新建SSTable文件中，写完后建立文件的index 数据，这样就完成了一次minor compaction。从图中也可以看出，对于被删除的记录，在minor compaction过程中并不真正删除这个记录，原因也很简单，这里只知道要删掉key记录，但是这个KV数据在哪里？那需要复杂的查找，所以在minor compaction的时候并不做删除，只是将这个key作为一个记录写入文件中，至于真正的删除操作，在以后更高层级的compaction中会去做。

当某个level下的SSTable文件数目超过一定设置值后，levelDb会从这个level的SSTable中选择一个文件（level>0），将其和高一层级的level+1的SSTable文件合并，这就是major compaction。

我们知道在大于0的层级中，每个SSTable文件内的Key都是由小到大有序存储的，而且不同文件之间的key范围（文件内最小key和最大key之间）不会有任何重叠。Level 0的SSTable文件有些特殊，尽管每个文件也是根据Key由小到大排列，但是因为level 0的文件是通过minor compaction直接生成的，所以任意两个level 0下的两个sstable文件可能再key范围上有重叠。所以在做major compaction的时候，对于大于level 0的层级，选择其中一个文件就行，但是对于level 0来说，指定某个文件后，本level中很可能有其他SSTable文件的key范围和这个文件有重叠，这种情况下，要找出所有有重叠的文件和level 1的文件进行合并，即level 0在进行文件选择的时候，可能会有多个文件参与major compaction。

LevelDb在选定某个level进行compaction后，还要选择是具体哪个文件要进行compaction，比如这次是文件A进行compaction，那么下次就是在key range上紧挨着文件A的文件B进行compaction，这样每个文件都会有机会轮流和高层的level 文件进行合并。

如果选好了level L的文件A和level L+1层的文件进行合并，那么问题又来了，应该选择level L+1哪些文件进行合并？levelDb选择L+1层中和文件A在key range上有重叠的所有文件来和文件A进行合并。也就是说，选定了level L的文件A，之后在level L+1中找到了所有需要合并的文件B,C,D…..等等。剩下的问题就是具体是如何进行major 合并的？就是说给定了一系列文件，每个文件内部是key有序的，如何对这些文件进行合并，使得新生成的文件仍然Key有序，同时抛掉哪些不再有价值的KV 数据。



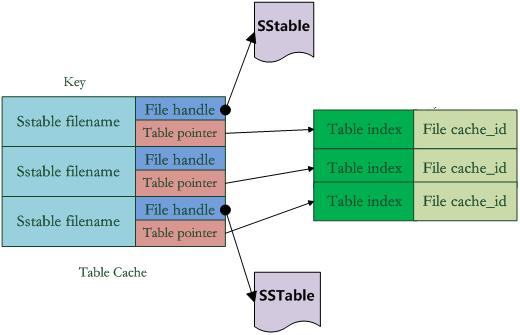
Major compaction的过程如下：对多个文件采用多路归并排序的方式，依次找出其中最小的Key记录，也就是对多个文件中的所有记录重新进行排序。之后采取一定的标准判断这个Key是否还需要保存，如果判断没有保存价值，那么直接抛掉，如果觉得还需要继续保存，那么就将其写入level L+1层中新生成的一个SSTable文件中。就这样对KV数据一一处理，形成了一系列新的L+1层数据文件，之前的L层文件和L+1层参与compaction 的文件数据此时已经没有意义了，所以全部删除。这样就完成了L层和L+1层文件记录的合并过程。

那么在major compaction过程中，判断一个KV记录是否抛弃的标准是什么呢？其中一个标准是：对于某个key来说，如果在小于L层中存在这个Key，那么这个KV在major compaction过程中可以抛掉。因为我们前面分析过，对于层级低于L的文件中如果存在同一Key的记录，那么说明对于Key来说，有更新鲜的Value存在，那么过去的Value就等于没有意义了，所以可以删除。

四、Cache

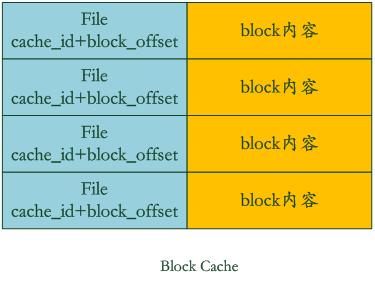
前面讲过对于levelDb来说，读取操作如果没有在内存的memtable中找到记录，要多次进行磁盘访问操作。假设最优情况，即第一次就在level 0中最新的文件中找到了这个key，那么也需要读取2次磁盘，一次是将SSTable的文件中的index部分读入内存，这样根据这个index可以确定key是在哪个block中存储；第二次是读入这个block的内容，然后在内存中查找key对应的value。

LevelDb中引入了两个不同的Cache:Table Cache和Block Cache。其中Block Cache是配置可选的，即在配置文件中指定是否打开这个功能。



如上图，在Table Cache中，key值是SSTable的文件名称，Value部分包含两部分，一个是指向磁盘打开的SSTable文件的文件指针，这是为了方便读取内容；另外一个是指向内存中这个SSTable文件对应的Table结构指针，table结构在内存中，保存了SSTable的index内容以及用来指示block cache用的cache\_id ,当然除此外还有其它一些内容。

比如在get(key)读取操作中，如果levelDb确定了key在某个level下某个文件A的key range范围内，那么需要判断是不是文件A真的包含这个KV。此时，levelDb会首先查找Table Cache，看这个文件是否在缓存里，如果找到了，那么根据index部分就可以查找是哪个block包含这个key。如果没有在缓存中找到文件，那么打开SSTable文件，将其index部分读入内存，然后插入Cache里面，去index里面定位哪个block包含这个Key 。如果确定了文件哪个block包含这个key，那么需要读入block内容，这是第二次读取。



Block Cache是为了加快这个过程的，其中的key是文件的cache\_id加上这个block在文件中的起始位置block\_offset。而value则是这个Block的内容。

如果levelDb发现这个block在block cache中，那么可以避免读取数据，直接在cache里的block内容里面查找key的value就行，如果没找到呢？那么读入block内容并把它插入block cache中。levelDb就是这样通过两个cache来加快读取速度的。从这里可以看出，如果读取的数据局部性比较好，也就是说要读的数据大部分在cache里面都能读到，那么读取效率应该还是很高的，而如果是对key进行顺序读取效率也应该不错，因为一次读入后可以多次被复用。但是如果是随机读取，您可以推断下其效率如何。

五、版本控制

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（当前版本），其它都是历史版本。

当执行一次compaction 或者 创建一个Iterator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。

**VersionSet** 是所有Version的集合，管理着所有存活的Version。

**VersionEdit** 表示Version之间的变化，相当于delta 增量，表示有增加了多少文件，删除了文件：

Version0 + VersionEdit --> Version1

Version0->Version1->Version2->Version3

VersionEdit会保存到MANIFEST文件中，当做数据恢复时就会从MANIFEST文件中读出来重建数据。

Leveldb的这种版本的控制，让我想到了双buffer切换，双buffer切换来自于图形学中，用于解决屏幕绘制时的闪屏问题，在服务器编程中也有用处。

比如我们的服务器上有一个字典库，每天我们需要更新这个字典库，我们可以新开一个buffer，将新的字典库加载到这个新buffer中，等到加载完毕，将字典的指针指向新的字典库。

Leveldb的version管理和双buffer切换类似，但是如果原version被某个iterator引用，那么这个version会一直保持，直到没有被任何一个iterator引用，此时就可以删除这个version。

学习参看

http://blog.csdn.net/qq112928/article/details/21172841

http://www.cnblogs.com/chenny7/p/4026447.html

http://www.cnblogs.com/haippy/archive/2011/12/04/2276064.html