2018.5.16

借鉴了<https://blog.csdn.net/sparkliang/article/details/8567602> 分析levelDB的系列博客

首先从一些杂项开始入手，即先分析util文件夹下的文件

1. arena.h，arena.cc为一个简单的内存池

分配策略如下：

用一个vector保存每次new出来的block块的首指针

1. 若需要分配的内存大小小于当前块剩余的块大小，那么直接返回该处首指针，并将记录位置的指针后移，否则，跳转至b;
2. 判断需要分配内存大小与定义的默认块的四分之一的关系，若大于，则直接调用函数分配一个要求大小的内存块出来挂在vector上，返回该块首指针；否则，浪费掉当前块剩余的部分，重新分配一个默认块大小的块，为4KB，挂在vector上，并将记录剩余块大小的指针移到合适的位置，返回该块首指针。
3. skipList.h 跳跃表，用来代替平衡树一种数据结构

跳跃表讲解：<http://blog.jobbole.com/111731/>

1. cache.cc 实现LRUCache

在struct LRUHandle中的 void\* value,使用void\*来保存数据，利于可分配不同大小的数据，在堆上new出来。

HandleTable:: Insert函数：里面使用了二重指针，是为了方便插入，不需要记录插入位置的前面的指针，同时，如果要插入的数据在链表中的话，新的节点会替代旧节点的位置，同时将旧节点返回。

在LRUHandle中三个指针分别为next,prev,next\_hash;处于cache中的节点分别处于两个链表中，一个有next\_hash指示的hash表中，这个表示用来通过key和hash值快速找到对应节点的；另外一个是next,prev指示的一个双向链表，而这个双向链表分为两个一个in\_use\_,一个lru\_，主要为了区别不同节点的引用个数，in\_use\_是表示正在被使用的节点，而lru\_是表示当前没被使用，有可能会被使用而加入到in\_use\_中，或者是由于容量不足，而将lru\_中的节点换出（同时会删除该节点在hash表中的指针关系）。Hash表中所有节点=in\_use\_的节点+lru\_的节点

1. memtable.h和memtable.cc 主要使用到前面提及的skiplist来实现

数据类型Varin32由五个字节构成，而skipList中的key值的格式为一个Varint32（数据长度）+数据

Memtable.cc中的关键函数：

Add:往skiplist里加入键值对

格式为：

Format of an entry is concatenation of:

key\_size : varint32 of internal\_key.size()

key bytes : char[internal\_key.size()]

value\_size : varint32 of value.size()

value bytes : char[value.size()]

在Memtable中并没有删除函数，删除某个key的value是在通过插入一条记录实现的（即add）,会打上一个Key的删除标记，即在ValueType字段上标识（kTypeDeletion），真正的删除操作在后面的Compation过程中，lazy delete

SequenceNumber（7bytes）代表不同操作的时间顺序

ValueType（1byte）操作类型

1. dbformat.cc中的InternalKeyComparator类继承自Comparator类

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：

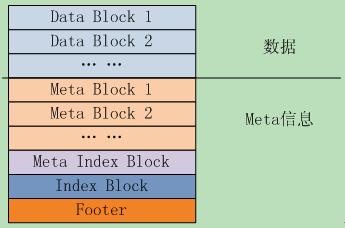
>1 首先根据user key按升序排列

>2 然后根据sequence number按降序排列（可以使得越近期产生的数据保持在链表的前面，有效的节省查找时间）

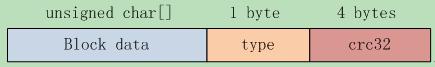
>3 最后根据value type按降序排列

1. 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。
2. log\_writer.h
3. log\_reader.h
4. SSTable，磁盘的物理存储存储

SSTable的文件组织：



虽然该格式中有不同的类型的block，如Data block,Meta block;但是他们都是以Block的格式方式组织的，Block的结构如下：

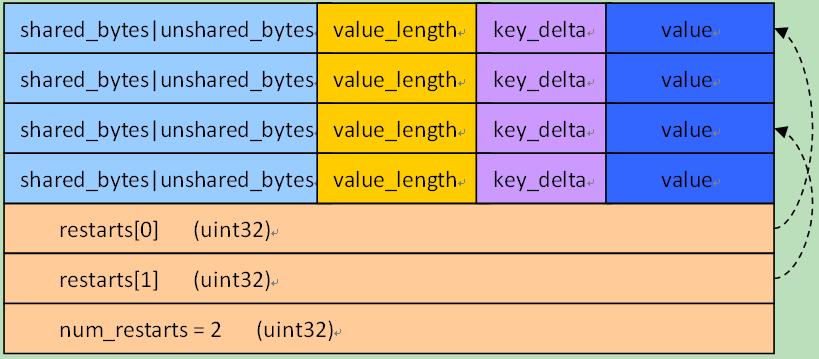


虽然block有好几种，但是Block Data都是有序的k/v对，因此写入、读取BlockData的接口都是统一的，对于Block Data的管理也都是相同的。

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。

block的存储示意图：



1. Block的构建，BlockBulider.h
2. Block的读取，Block.h
3. 创建sstable文件，TableBuilder类