一步步实现k-v数据存储

代码写得很垃圾，不敢叫他k-v数据库，暂时就叫k-v数据存储吧。

一、接口设计

接口设计借鉴了LevelDB，实际上这个项目很多东西都是从LevelDB学来的，后面也会逐一介绍到。

```
DB *db;
DB::Open(string dbname,&db);
db->Put( key,  value);
db->Get( key, *value);
delete db;
```

- Put

  ```
  bool Put(string key,string value);
  ```

- Get

  ```
  bool Get(string key,string *value);
  ```

- Delete

  ```
  delete db
  ```

实际上，个人认为应该实现Close()接口对应Open()，但不太明白LevelDB的作者为什么没有这样做。这里暂且先依循LevelDB的做法，可以之后再修改。

二、存储模型

采用了Bitcask存储模型。Bitcask 是一个日志型、基于hash表结构的key-value存储模型，以Bitcask为存储模型的K-V系统有 Riak 和 beansdb 新版本等。

Bitcask简洁、优雅的key/value存储引擎给出了非常清晰的Bitcask存储模型的介绍，并给出了非常简洁的java实现。

> # Bitcask简洁、优雅的key/value存储引擎
>
> 在关系数据库存储上，Btree一直是主角，但在某些情况下，log(n)的读写操作并不是总是让人满意。 Bitcask是一种连续写入很快速的Key/Value数据存储结构，读写操作的时间复杂度近似常量。
>
> ####Bitcask连续写入操作 Bitcask具有高效的连续写入操作，连续写操作类似向log文件追加记录，因此Bitcask也被称作是日志结构存储。
>
> Bitcask将存储对象的key、value分别存储：
>
> - 在内存中对key创建索引
> - 磁盘文件存储value数据

> 当有数据需要写入时，磁盘无需遍历文件，直接写入到数据块或者文件的末尾，避免了磁盘机械查找的时间，写入磁盘之后，只需要在内存的HashMap中更新相应的索引,内存中用HashMap来保存一条记录的索引部分，一条索引包含的信息如下：
>
> - [Key: Jason, Filename: employee.db, Offset:0, Size:146, ModifiedDate:2343432312]
> - [Key: Bill, Filename: employee.db, Offset:146, Size:146, ModifiedDate:5489354345]
>
> Key表示一条记录的主键，查找通过它在HashMap中找到完整索引信息 Filename是磁盘文件名字，通过它和Offset找到Value在磁盘的开始位置 Offset是Value在文件中偏移量，通过它和Size可以读取一条记录 Size是Value所占的磁盘大小，单位是Byte 假设目前数据库中已有上述两条的记录，当我要写入key为 "Jobs"， value为: object的一条新记录时， 只需要在文件employee.db的末尾写入value=object，在HashMap中添加索引：[Key: Jobs, Filename: employee.db, Offset:292, Size:146, ModifiedDate:9489354343] 即可。
>
> 最后数据库就包含了三条索引信息：
>
> - [Key: Jason, Filename: employee.db, Offset:0, Size:146, ModifiedDate:2343432312]
> - [Key: Bill, Filename: employee.db, Offset:146, Size:150, ModifiedDate:5489354345]
> - [Key: Jobs, Filename: employee.db, Offset:294, Size:136, ModifiedDate:948965443]
>
> ####Bitcask随机读取操作
>
> 由于数据在内存当中使用HashMap作为索引，查找索引的时间为Hash查找的时间，近似常量。比如查找Bill，直接通过Key就可以找到它的索引信息，再根据索引信息，找到value在文件位置和大小，精确读取出bytes，反序列化成value对象。 当然在value存入文件时需要序列化内存对象成bytes。磁盘读取的过程的时间复杂度也是常量， 并不会随时数据的增大而增大。
>
> ####Bitcask 数据删除和更新 一条记录包含了索引和数据两个部分,删除索引容易，但要彻底的删除数据不是件容易的事情（参考磁盘空间整理）。对于更新数据，Bitcask通常采用的策略是append一条新数据，并更新已有的索引，至于旧数据则在清理数据的时候把它删除掉。
>
> ####Bitcask适合的场景
>
> - 适合连续写入，随机的读取，连续读取性能不如Btree；
> - 记录的key可以完全的载入内存；
> - value的大小比key大很多,否则意义不大；

在实际实现时，在index里多存一个固定长度“头”，表示当前index的长度。读取index的时候，先把头读取出来，然后根据读出的数据再把完整的index读出来，因此，index的编码函数如下

```
void EncodeIndex(char *buf, const Index &idx) {
    int pos = 0;
    int idx_size = idx.GetIndexSize();
    //index长度
    memcpy(buf+pos,&idx_size,sizeof(int));
    pos += sizeof(int);
    // file_id(替代filename), offset,  size , key
    memcpy(buf+pos,&idx.file_id_,sizeof(int));
    pos += sizeof(int);
    memcpy(buf+pos,&idx.offset_,sizeof(int));
    pos += sizeof(int);
    memcpy(buf+pos,&idx.key_size_,sizeof(int));
    pos += sizeof(int);
    memcpy(buf+pos,idx.key_.c_str(),idx.key_size_);
    pos += idx.key_size_;
    memcpy(buf+pos,&idx.value_size_,sizeof(int));
}
```

三、内存索引

我们需要在内存中对key创建索引，用链表来存当然可以，但O(n)的复杂度在数据很多时候还是很低效。因此这里选择了SkipList作为存储索引的数据结构，搜索复杂度是logn。LevelDB也是采用的SkipList作为内存中的索引。

跳表由 William Pugh 在 1990 年在论文Skip Lists: A Probabilistic Alternative to Balanced Trees

> Skip lists are a data structure that can be used in place of balanced trees.
> Skip lists use probabilistic balancing rather than strictly enforced balancing and as a result the algorithms for insertion and deletion in skip lists are much simpler and significantly faster than equivalent algorithms for balanced trees.
> 跳表是一种可以取代平衡树的数据结构。
> 跳表使用概率均衡而非严格均衡策略，从而相对于平衡树，大大简化和加速了元素的插入和删除。

本来我想尝试着自己实现一个SkipList，实际上我也确实这么做了。但在实现了一个简单的SkipList之后，我发现还需要处理线程安全的问题。最后索性直接copy了LevelDB里的SkipList，下面对其中的代码写一些解释和自己的理解。如果对SkipList不太了解，可以先看一下这篇文章了解相关概念。

首先，一个SkipList主要需要有以下成员和方法

```
struct SkipList::Node {
private:
    // 长度等于节点高度的数组。next_[0]是最低级链表
    port::AtomicPointer next_[1];   //
    Key key;
};
class SkipList {
private:
    Node *const head_;
    port::AtomicPointer max_height_;   // Height of the entire list
public:
    void Insert(const Key &key);
    bool Get(const Key &key, Key *result) const;
}
```

初始化

```

SkipList<Key, Comparator>::SkipList(Comparator cmp)
        : compare_(cmp),
          head_(NewNode(Key() /* any key will do */, kMaxHeight)),
          max_height_(reinterpret_cast<void *>(1)),
          rnd_(0xdeadbeef) {
    for (int i = 0; i < kMaxHeight; i++) {
        head_->SetNext(i, nullptr);
    }
}
```

1. 创建一个kMaxHeight层的头节点，并让每一层都指向nullptr，nullptr的key被认为是无穷大
2. rnd*是一个Random对象，用来在Insert的时候生成随机数*
3. 下面详细输送说max_height和AtomicPointer类

   max_height是skiplist最高节点的层数，也称为skiplist的层数。

在插入Node的时候，可能会修改skiplist的层数。为了支持线程安全，max_height_ 是一个AtomicPointer对象，成员只有一个void * rep指针，下面的初始化将1转化为为 void*之后传入AtomicPointer构造函数来初始化max_height。AtomicPointer通过设置内存屏障(MemoryBarrier)来保证线程安全。这也可以通过c++11的原子类来写，实际上LevelDB里也写了这样的实现，它根据宏LEVELDB_HAVE_MEMORY_BARRIER来切换。这个项目是在windows系统下写的，windows已经定义过MemoryBarrier（void）的宏，可以直接使用内存屏障来实现AtomicPointer。

关于LevelDB中的SkipList可以去leveldb看官方的源码，在写这个文章的时候，发现了一个兄弟写的一些注释和理解，看上去蛮不错的也挂出来。源码注解

四、To Do List

- 缓冲区

目前只是bitcask通过index能快速找到key-value对在文件中的位置，但每次读取都是访问磁盘的，非常消耗时间。可以写一个Buffer对象保存文件的内容到内存中作为读取缓冲，读取k-v对时先从内存中读取。没找到再到磁盘读取。

同样的，对于写入，Buffer还应该有一个输出缓冲区保存待写文件，在缓冲区满了之后，再一起dump。buffer池。

但这个项目的实现用了fread和fwrite，第一次fread和fwrite时会分别提供默认4096字节的读和写缓冲区，也可以自己调用setvbuf函数设置缓冲区。fread从文件读取_bufsiz大小的数据存放在读缓冲区，后续如无必要，不会读文件，而是直接读缓冲区。同理，fwrite将数据写入写缓冲区，除非缓冲区已满，否则不写入文件。

- 修改

  简单实现Delete很简单，bitcask的Put和修改几乎一样的，只需要给每个index设置一个时间戳，取最新的index，删除只需要把原来的index设置删除标记。

  但需要这样做可能会造成资源浪费，需要定期进行垃圾清理，目前还不知道怎么实现

- 内存管理

  由于这只是个demo，内存管理方面并不严格，统统用的int，后面还有很多东西需要修改、重构。