个人角度的概括：写的可读性极差，排版极度难看，原因：为避免文件链接时 undefind reference， 所以把所有代码都写在了 kvstore.h

设计概括

文件结构：

kvstore.h
- -include 引用头文件
- -define structs
  - -SNode 存放 kvPair
  - -Skiplist 存放 SNode
  - -Index 记录 SSTable 的 key-offset
  - -SSTable 内存中 cache 用于记录 sstable 文件 index 的范围的最小单元

- -SkipList_func() 实现跳表的功能函数
  - -Init
  - -Insert
  - -find
  - -delete
  - -release
  - -createindex 在当前 memtable 满了时导出其对于 Index

- -Class Kvstore 原有的定义好的模板类
  - -kvstore
  - -~kvstore
  - -put
  - -del
  - -get
  - -reset
  - *+createfile 在 level0 生成临时文件
  - *+compaction 合并

核心函数设计：

1. createfile()
   根据 level0 当前状态，生成文件./level0/SSTable(x).hex，每次生产默认调用 compaction（0），意思是从 0 层开始归并，但 compaction 开头检测当前层是否文件溢出，不溢出直接返回，溢出则归并。
   每次创建结束默认调用 skiplist_release 与 init 重载 memtable。

2. compaction(level)

(1)检测当前层的 sstable 是否超标量，不超标直接返回，否则下一步。

(2)建立在超标量的基础上，初始化 memory，遍历当前 level 确定超标量的部分，记其第一个超标的 SSTable 为 overflow_start，最后一个是 overflow_end，总数 o_imm，注意，第零层单独处理，overflow_start 必为 1，overflow_end 必为 3，o_imm 必为 3.

(3)利用 filesystem 操作遍历 level 的 overflow 的文件，统计 key 的覆盖范围。我的遍历办法是循环设置字符串，使得每次打开的文件不一样且递归。你将会在代码多处找到类似的实现方法。

```
overflow_path="./data/level"+to_string(level)+"/SSTable"+to_string(i)+".hex";
```

(4)同理（3），遍历下一层 level+1 的文件，取得所有与（3）范围有交集的文件名。但是这里有优化：针对不同情况减少文件打开次数。

<0>默认情况，认为 level+1 存在且有文件有交集，起始文件和终止文件用 head 与 tail 记录

<1> level+1 不存在或者其路径下没有文件，对应 caseflag=1

<2> level+1 第一个文件的最小值大于 overflow 层范围最大值，对应 caseflag=2

<3> level+1 最后一个文件最大值小于 overflow 层范围最小值，对应 caseflag=3

(5)判断 caseflag 是否为零，为零则依据 head 与 tail 插入 level+1 层数据，从前到后，这个插入必须在（current）level 之前，因为上一层的修改优先级更高，它被用户修改的时间更晚。并且，文件被插入后不能删除，我的做法是改名为“usedss?.hex”,即表明这个文件被合并使用过，但是在整个合并完成前，不应删除，（防止突然退出）。

(6)插入 level 的 overflow 文件，注意顺序与优先级。

(7)生成 tmpss.hex,依据 skiplist 每次修改都对其 bitelength（我这用的 size），除以 2*1024*1024 大致算出生成的临时文件数目 ss_num，（其实你会发现它由于零头问题并不靠谱，之后再具体解释）。随后依据 ss_num，生成临时文件，并创建临时数组储存 sstable range 以放入 key_range 中。

(8)依据 caseflag 进行 rename 处理，大致思路是：

usedss 指代用过的 sstable，它们被改名之后空出来了 tail-head+1 个位置，依据生成 tmpss 的数量 s_num，比较大小考虑要不要把 tail 之后的 sstable 前移或是后移，移动好之后再 rename tmpss(1,s_num)=>sstable(head,head+s_num),最后集中删除 usedss.hex 与 overflow sstable，（这样一旦生成 tmpss 时程序挂掉了，在下次运行时在构造函数读取本地数据时恢复 usedss 成 sstable 就能恢复数据），进入 compaction（level+1）.

ps:关于 2*1024*1024 的说明，正所谓零头害死人，虽然给定 2M 的标准，但在储存的时候插入最后一个数据，此时文件往往比 2M 大一点点，这个数据我这 local 里一般是 2050~2080Kb 不等，在算生成临时文件 tmpss 时，size/2*1024*1024，忽略了零头的溢出，导致有可能在一口气生成 50~60 甚至更多 tmpss 时，每个 tmpss 比预想的都多存了 10~30k，积攒一下大概就有 60*30=1800Kb（最坏），假如原来计算最后一个文件是打算存 1.5M，结果发现存到这的时候数据没了，被前面的 sstable 零头偷偷存完了，显然会引发不必要的错

误。我没想到最优解，于是在算 s_num 的时候用 size/2060*1024（2060Kb），并且最后一个文件尽可能多存，减少上述情况。至少我本地跑 2G 数据的时候没出问题。

性能测试:

1.测试环境

ide：vscode

gcc：9.2.0　　-std=c++17

hardware:

shell: powershell

network: not in need

tools：附带的正确性测试文件 correctness.cc 与持久性测试文件 persistence.cc

2.测试参数与方法

测试项目：

1.时延

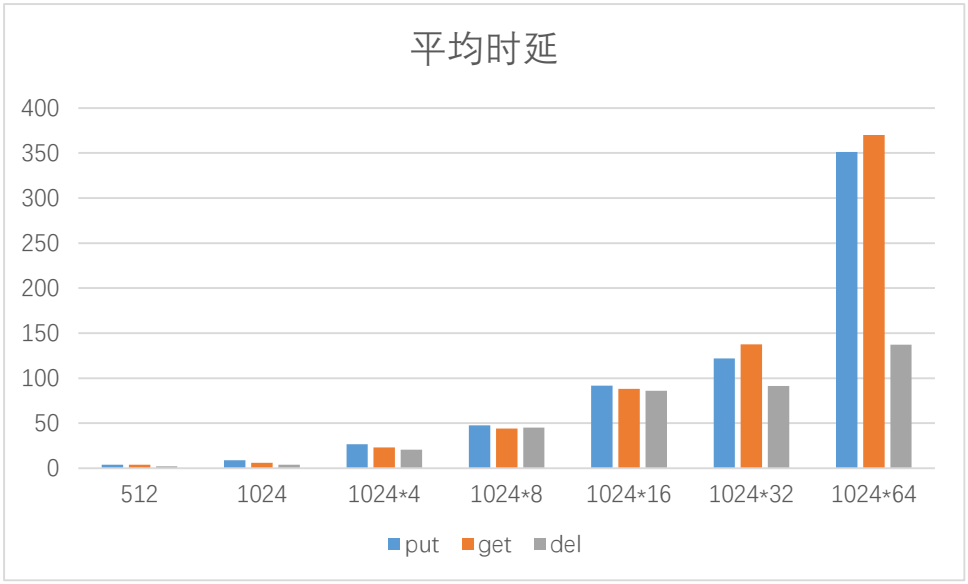
2.吞吐量

参数：正确性测试文件的数据量大小，默认为 1024*64

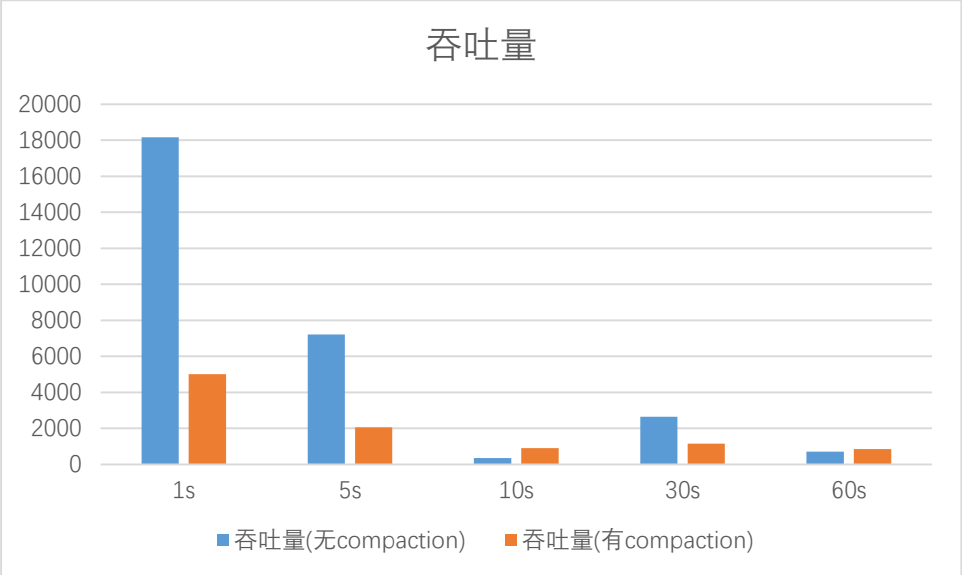
方法 ： 在给定的测试文件基础上进行修改，并引入 ctime 计时，print 计时结果。具体代码就是增加单独的 put、get、del 循环，计算操作总次数与总时间就可以算结果。

3. 测试结果

无compaction									
时延/ms/数据量	512	1024	1024*4	1024*8	1024*16	1024*32	1024*64		
put	2	9	109	391	1503	3998	23009		
get	2	6	94	361	1442	4506	24248		
del	1	4	84	370	1408	2988	8996		
avg/s*10^-6									
put	3.9	8.789	26.661	47.729	91.736	122.001	351.089		
get	3.9	5.859	22.949	44.067	88.017	137.512	369.995		
del	1.9	3.906	20.507	45.166	85.938	91.187	137.268		
时间	1s	5s	6s	10s	11s	30s	31s	60s	61s
处理总数(无compaction)	18158	35461	42670	46304	46653	80696	83347	108300	109005
处理总数(有compaction)	5005	11212	13271	18214	19115	32385	33539	44874	45717
	1s	5s	10s	30s	60s	60+			
吞吐量(无compaction)	18158	7209	349	2651	705	内存满了			
吞吐量(有compaction)	5005	2059	901	1154	843	…			



时间单位：10^-6 s　　(微秒)



注:60s 之后内存满了。

4. 结果分析

表 1：随着插入数据变多，跳表规模变大，对表的相关操作用时呈指数上升，考虑到课上所学，符合时间复杂度 O(logn)。且，删除的本质是插入一个已经存在的 kv 对，不需要遍历到底也不需要生成高度 k，比插入快。同样查找步骤也比插入快。至于为什么最后查找比插入快一丢丢，可能是因为我的测试程序插入在查找之前，插入所有数据之后内存不够，导致系统运行变慢，查找速度变慢。当然这只是猜测。

表二：符合

①起始阶段插入内存比插入文件要快（不需要读写文件）

②由于数据又来越多，无 compaction 自然变慢

③有 compaction 情况，插入到内存还是一样快，但是由于生成的目录变多了，递归层数变多，吞吐量也变小。

④有了 compaction 之后，吞吐量变化趋势变小，后期保持稳定。

⑤10s memtable 发生了什么我也不知道，检查了一下任务管理器感觉是是系统压缩内存相关的问题。