key-value-system 设计文档

一、 要求

- 项目：单机 key-value 存储系统
- 数据规模：单机存储 1000W 条记录，平均大小 100KB，单机存储 1TB 数据，单条最大长度 5MB
- 响应时间：平均单次请求在 50ms 内完成（SATA 硬盘）
- 接口：put(key, value), get(key), delete(key)
- 代码估计：5000 行

二、 总体设计

系统分成 4 个模块：

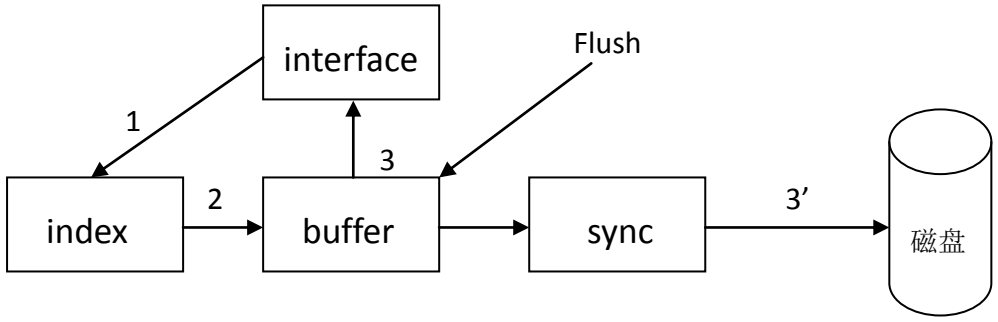
1. interface 模块：对外提供的接口模块，支撑 KV 系统，连接各个模块，安排程序的执行流程。
2. index 模块：根据 key 值，来提供插入、删除、查找结构体 IDX_VALUE_INFO 的信息。由 IDX_VALUE_INFO 可以知道 value 存放的位置和 value 的大小。
3. buffer 模块：把 value *缓存写* 在 buffer 里，使得数据写到磁盘上能够“局部顺序写”，减少写 I/O 时间。另一方面 buffer 可以当*缓存读*，减少读 I/O 时间。Buffer 里同时有一个线程监视 buffer 的情况，适时把 value 写到磁盘上。
4. sync 模块：提供“内存与磁盘间”读写操作的接口，index 模块持久化接口。

系统有 3 个文件： disk_file——存放 value 的大文件。
IMAGE_file——index 模块在内存中的镜像，下次运行可以载入内存继续运行。
log_file——系统输出的 log 信息，可以提示错误。

三、 各模块详细设计

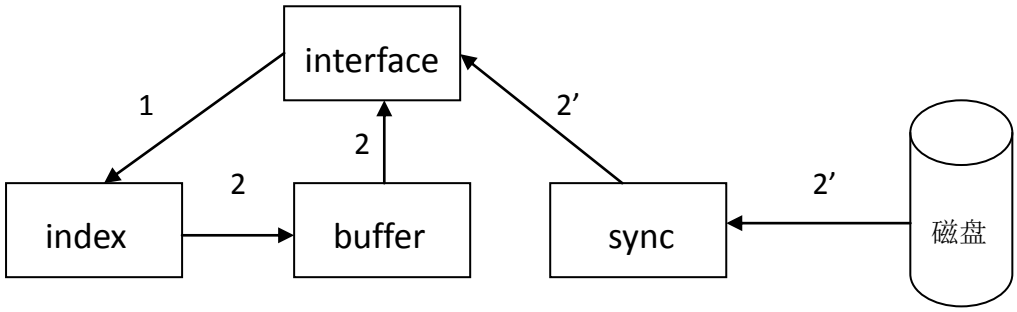
- interface 模块详细设计

put：



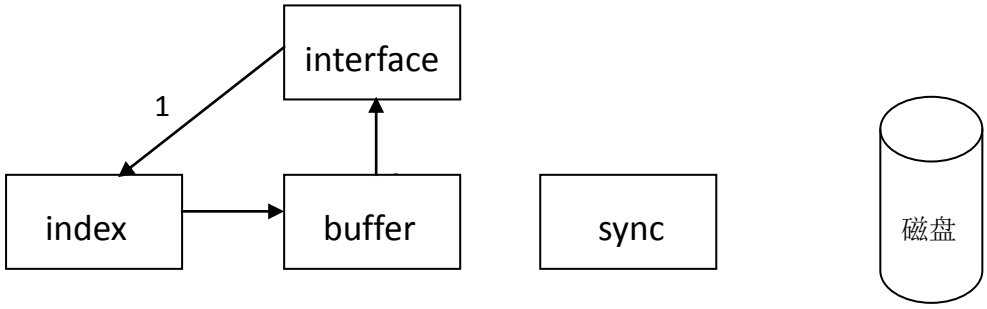
流程： 1. 把 key 插入到 index 中，并且返回和 key 相关联的 value_info 地址给 buffer。
2. 调用 buffer_put，传入 value 内容和记录 value 信息的 value_info 地址并在 buffer 中填写。
3. 返回调用。同时 buffer_put 中 lookout 线程满足 flush 条件则把 value 以包的形式（多条 value）写到磁盘。

GET:



流程： 1. 调用 idx_search，在 index 模块里根据 key，返回 value_info。
2. 根据 value_info，如果 value 在 buffer 里直接从 buffer 读出 value，返回。
2' 否则 value 会在磁盘里，调用 sync_read，从磁盘读出 value，返回。

DELETE:



流程：1.调用 idx_delete(),删除 idx_node 并返回 value_info。
2.根据 value_info，如果 buffer 里有 value，删除 value。（这样可以在 buffer 写到磁盘前就删除，不用占用磁盘 I/O 资源。）

- index 模块详细设计

idx_nodes 是 index 存放 struct IDX_NODE 的池，对 idx_nodes 池的操作有：申请一个 struct IDX_NODE 和释放一个 struct IDX_NODE。

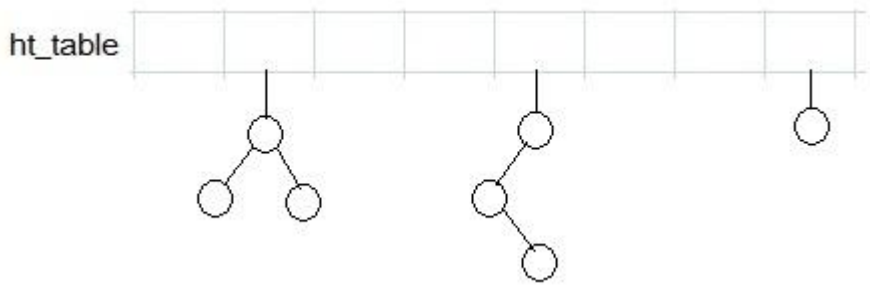
idx_nodes:

1	2	3	4	5	6	7
8	9	10	11	……		

free_idx_nodes 是一个栈，初始情况把所有可用的 node_id 都压入栈，_get_free_idx_node() 弹栈，_put_free_idx_node()压栈。

free_idx_nodes_horizon

hash_tree:



ht_table 是一个 hash 表，冲突的元素以树的形式连接在表后（拉链法的变形），可以在检索数据的时候提升速度。

Index layout:

ht_table	idx_nodes	free_idx_nodes	free_idx_nodes_horizon

说明：索引的设计中，并没有存入 key 值，而是通过以下方式防止哈希冲突的：

(1) hash 槽的选择：ht_table_id = hash_func_1(key);

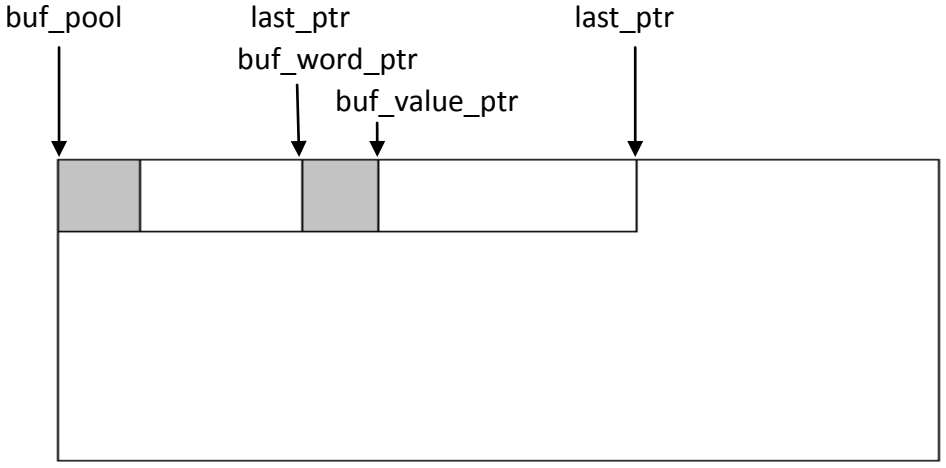
(2) 树中偏序关系的维护：<hash_func_2, hash_func_3>

实验中， 在一个 hash 槽内插入 2000W 条随机生成的 32 位 key 值，没有出现冲突。

- buffer 模块详细设计

value 在 buffer 里的存放形式是：struct buf_word + value，如下图灰色和白色。

对程序中各个变量的说明:

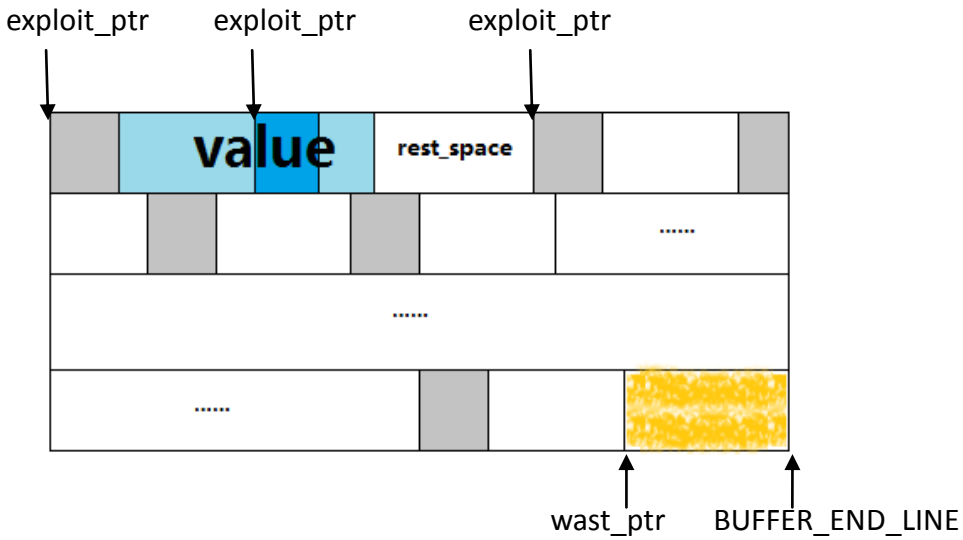


buf_pool：buffer 池的首地址，每次把 buffer 写满了，相关变量重置为 buf_pool 位置。进行下一趟操作。

last_ptr： 空闲空间的地址。每次存放完 value 之后，指针移到 value 后面，value 是一个接着一个存放的。

buf_word_ptr：指向 struct buf_word 的指针。

buf_value_ptr：指向 value 的指针。



蓝色（深蓝和浅蓝）部分是第二趟写的 value。深蓝色表示的是这个 value 覆盖了上次一个 buf_word，后面的 rest_space 是这次开拓的空间剩下的部分，可以给下次用。最后面的黄色是荒废区。如果某一次的 value_size 太大，以至于后面的空间不够必须重头再重新开拓空间，那么这个黄色区域就舍弃掉了，直接从头重新开拓空间。避免出现 value 分开的情况。

exploit_ptr：除了第一趟（first_flag = 1，从 buf_pool 到 BUFFER_END_LINE）直接把 value 一个接着一个地放到空闲空间外，以后每次放一个 value 都需要先“开拓”空间，并修改 struct buf_word。exploit_ptr 就是这个开拓指针。这样做的目的如下：1.防止覆盖旧的有效的 value。2.把 value flush 到磁盘以后，还可以有缓存读（cache）的功能。具体详见 struct buf_word->priority 说明。
rest_space：每次开拓的空间肯定会“大于等于”value_size，这次剩下的空间叫 rest_space，可以放到下次开拓的空间里供下次使用。
wast_ptr：荒废区的首地址。
last_waste_ptr：上一趟的荒废区的首地址。
last_flush_ptr：上次把 value　flush 到磁盘的地址。
not_flush_size：没有 flush 到磁盘的 value 的总大小（不包括 struct buf_word）。给监视线程用的
first_flag：第一趟写 buffer 的标志，这个时候不需要开拓空间。所有空间都是可用的。
disk_ptr：磁盘的空闲地址。这个版本暂时没有 disk 这个模块，磁盘地址放到 buffer 中。

```
enum priority_t { p_unavail, p_flushed, p_not_flush };
typedef struct
{
    struct IDX_VALUE_INFO* value_info_ptr;
    enum priority_t priority;
}buf_word;
```

priority 有 3 个值：p_unavail, p_flushed, p_not_flush;
p_unavail 代表结构体后面的 value 已经无效（被删除）了。
p_flushed 代表结构体后面的 value 已经 flush 到磁盘上，但是在 buffer 中还是有效的。可以通过 buf_ptr 找到 value。
p_not_flush 代表结构体后面的 value 还没有被 flush 到磁盘上。

这里之所以要 value_info_ptr，是因为以下情况需要修改 value_info 的变量：第二趟以后，每次开辟空间，要把 value_info 里的 buffer_ptr 置为 NULL（表示 value 已经不在 buffer 里，priority = 1），代表不能通过 buffer_ptr 在 buffer 里读 value 了，此时只能在磁盘上获取 value。

buffer_put 过程比较复杂，说明一下 buffer_put 的流程：
先保证，这趟下来，后面的空间肯定能放下 value 的值，不然，把剩下的空间设为荒废区，从头重新开始。
如果是第一趟在 buffer 里存 value，那么直接一个接着一个地存放数据。
否则， 先开辟足够大的空间。开辟的过程包括修改 buf_word_ptr->valu_info 的值和 priority 的值。
如果这次放完 value，exploit_ptr 进入了荒废区，那么后面的空间不要了，因为荒废区不符合开辟的条件。

监视线程的流程：
触发写磁盘的动作是：每过 SLEEP_TIME 后检查没有 flush 的 value 总数（称作包）是否超过一个标准值：BUFFER_HORIZON_SIZE，如果 SLEEP_TIME 过后，还不到这个值，就不 flush，理由是 flush 的数量太小，局部顺序写的效果不明显。
当主线程执行 buffer_exit()时，给监视线程发取消请求，监视线程把剩下的所有没 flush 的 value 都 flush 到磁盘上，线程结束，buffer_exit()退出。
触发条件：SLEEP_TIME && BUFFER_HORIZON_SIZE

buffer 各种例子的演示：
假设需要 put 的 value 有如下顺序：20、70、20、10、10、30、85、30
第一趟结果各 value 的 state：

value_1	value_2	value_3	value_4	value_5	waste_space
20	70	20	10	10	20
p_not_flush	p_not_flush	p_not_flush	p_not_flush	p_not_flush	waste_space

现在删除 70 和 20：

value_1	value_2	value_3	value_4	value_5	waste_space
20	70	20	10	10	20
p_not_flush	p_unavail	p_unavail	p_not_flush	p_not_flush	waste_space

监视线程 flush 了 20、10、10，注意 70 和 20 已经删掉，不需要 flush：

value_1	value_2	value_3	value_4	value_5	waste_space
20	70	20	10	10	20
p_flushed	p_unavail	p_unavail	p_flushed	p_flushed	waste_space

第二趟 put 20、85 之后的情况，rest_space 剩余 5 供下次 put 30 使用，其中 value_6 把 value_1 覆盖的时候已经通知 value_2 对应的 value_info，buf_ptr 已经无效，依此类推：

value_6	value_7	rest_space	value_5	waste_space
20	85	5	10	20
p_not_flush	p_not_flush		p_flushed	waste_space

put 30 后的 state：

value_6	value_7	value_5	waste_space
30	85	30	5
p_not_flush	p_not_flush	p_not_flush	waste_space

- sync 模块详细设计

I/O 比较：从 mmap, read, fread 三个文件 I/O 来看，本质上都是在操作系统的 cache 和 VM 层之上工作，但与 cache 的距离越来越远。mmap 让应用直接读写 cache，read 必须做一次向用户空间的搬运，fread 还要经过 libc 这一层。所以选择 mmap 较好。因为数据较大，和系统限制等客观原因，这个版本暂时使用 read, write 作为选择。后续版本会做 mmap 版本，并和 read/write 做性能上的比较测试。

这个模块除了 sync_read()和 sync_write()外，还提供了 index 模块持久化的接口。

四、 后续改进：

1. KVS_ENV 启动配置信息参数较少，之后提供 Direct_IO 参数，可以把 buffer 模块变成可配置的。
2. sync 模块磁盘的读写操作的选择 read/write, mmap 等等选择对比。
3. 提供 Server 端，使数据能在远程调用系统接口。
4. disk_file 没有做磁盘碎片回收机制，后续增加。
5. index 后续要有实时的同步到磁盘上，防止断电等意外出现后数据丢失找不到。
6. 给多种语言提供接口。
7. ……