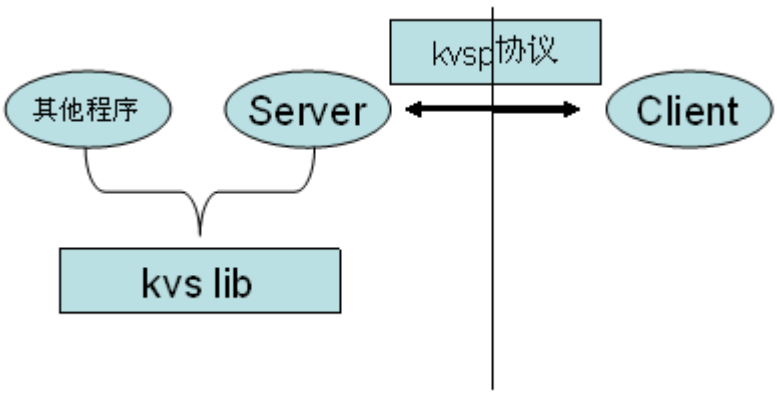
KVSystem 设计文档

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100K, 单机存储1T 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)
- 代码行估计: 5000行

二、总体设计



KVS 系统是基于单个大文件的，将单个文件看做整个硬盘。大文件的布局如图：

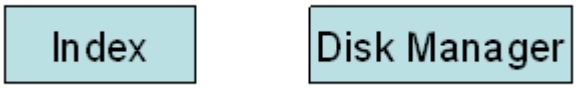
其中 DISK_IMAGE 大小为 550M，是程序运行时需要加载到内存中。DISK_IMAGE 的布局如下：

其中，IMAGE_K_SUPER 用于记录 kvs 系统基本信息，并为周期性备份预留空间；IMAGE_K_INDEX 为索引所需要空间；IMAGE_K_DISK 为空间管理所需要的空间。总大小（以 K 为单位）：

```
#define    IMAGE_K_SIZE          548684
```

三、KVS Lib 的设计

由两个模块构成：



KVS Lib 提供的接口有(kvs.h)：

```
/* 基本接口 */
int    kv_init(KVS_SET*, char** ret_p);
int    kv_exit();
int    kv_get(char* key, int key_size, char* buf, int buf_size);
int    kv_put(char* key, int key_size, char* value, int value_size);
int    kv_delete(char* key, int key_size);

/* 其他接口 */
int    kv_put_index(char* key, int key_size, int value_size);
int    kv_put_seg(int disk_location, char* value, int value_size_in_16k);
int    kv_get_2(char* key, int key_size, char** buf, int* buf_size);
```

1.Index 的设计：

Index 提供的主要接口有(index.h)：

```
STATE    idx_search(char* key, int key_len, PTR_DISK* disk_location, int* slot_cnt, int* last_slot_info);
STATE    idx_insert(char* key, int key_len, PTR_DISK disk_location, int slot_cnt, int last_slot_info);
STATE    idx_delete(char* key, int key_len, PTR_DISK* disk_location);
```

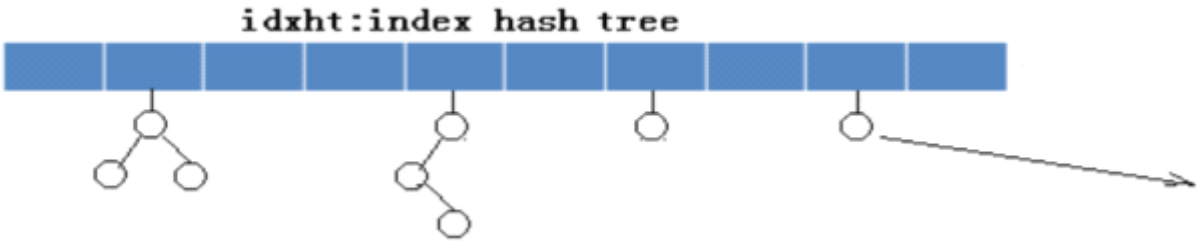
在内存中的布局(index.c)：

总大小（以 K 为单位）：

```
#define    IMAGE_K_INDEX_SIZE    274341
```

理论上，该索引布局支持 1000W 条数据任意插入、删除。

数据结构：



```
typedef struct
{
    HASH        hash_2, hash_3;     //维护树中的偏序关系
    PTR_DISK    disk_loc;           //ke y 对应的磁盘位置
    int         last_slot_info;     //磁盘最后一块的信息，此处是最后一块实际有效的长度
    int         slot_cnt;           //key 对应的 value 占磁盘多少块
    PTR_KW      left_ptr, right_ptr; //维护树的左右指针。
} IDX_NODE;
```

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： id=hash_func_1(key);
（2）树中偏序关系的维护：<hash_func_2(key), hash_func_3(key)>。
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。
由于历史原因，该部分(index.c)代码很糟糕。

2.空间管理的设计：

空间管理器讲大文件分为 16K 为单位的连续块，提供的主要接口有：

```
PTR_DISK    disk_alloc(int chunk_size);
void        disk_free(PTR_DISK disk_location);
```

在内存中的布局：

数据结构：

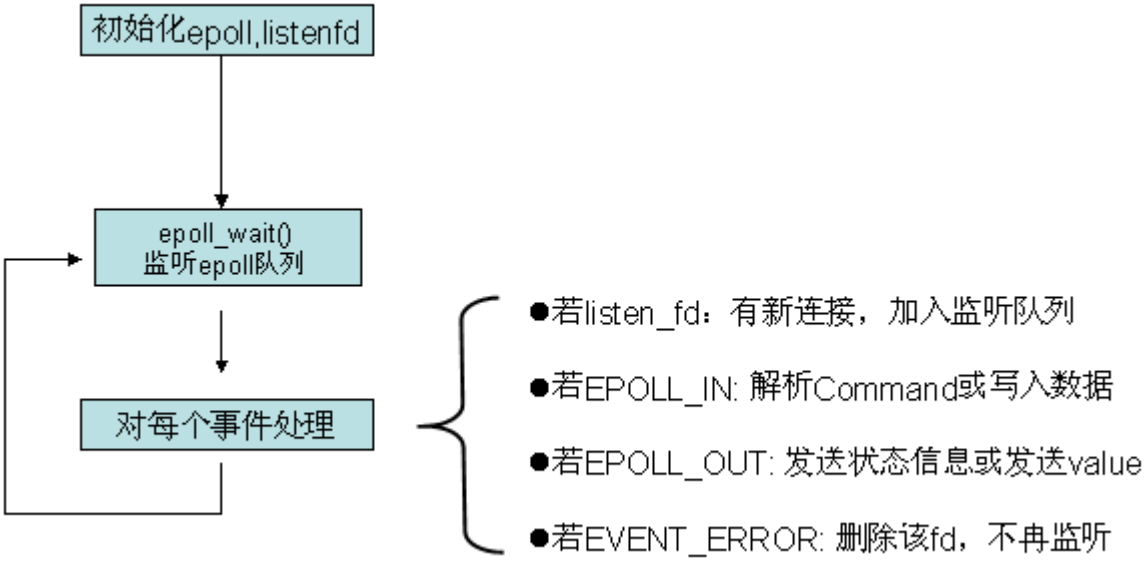
空间管理主要目的实现两个功能：
（1）快速找到指定长度的连续空块
（2）当释放空块，能够快速与左右相邻的空闲块合并为新的大的连续空闲块。
支持这些操作的数据结构：
（1）Free_lists[5M/16K]：用于快速定位指定长度的空闲块。
（2）物理地址链表：用于连接物理地址相邻的块（不一定是空闲的）。
（3）HASH 树：用于给定 空间地址，快速定位到指定 node_id 中。
满足这样的属性：当该块使用时，它在 Hash 树中；当该块被释放(或空闲)时，它在物理地址链表中。

```
typedef struct NODE
{
    int ptr_disk;               //该块地址
    int flist_pre, flist_suc;   // free_lists[]
    int node_pre, node_suc;     // (2)、(3)的复用
    int size;                   //该块长度
    int used;                   //该块空闲或已经被使用
}NODE;
```

四、Server 的设计

Server 端在 IO 模型上有两个选择：多线程模型、IO 复用模型。
我们选择了 IO 复用模型是因为 KVS Lib 本身不是线程安全的，若使用多线程的模型则还需要实现一个队列或者 KVS Lib 线程安全的支持。所以暂时选择 IO 复用模型。

在 IO 复用模型的基础上，选择 Epoll 的 LT 阻塞模型。除此之外，对网络 IO 的操作均为非阻塞，以防止“一个连接出现问题，Server 阻塞”。
Server 流程如图：



五、后续改进

1.Server 启动的配置参数少，之后提供 config_file 的方式启动。Server 启动时要对系统资源、权限等处理。
2.整体性能改进（见《KVSystem 性能测试文档》中的分析)