資料結構 HW6 report 徐松廷 110501521

1. 作業目標:

   延續先前作業的程式，支援類 Redis Hash 的操作，指令包含 HSET，HGET，HDEL，EXPIRE。

   a. 利用 chaining 解決 hash collision 問題

   b. 每次操作時計算 load factor，設計閥值，放大與縮小 hash table size。

   c. 將所有開發程式執行流程，改為事件觸發導向，即利用 callback 函式處理輸入指令與自動 expire 功能。

2. code review

   a. data structure 定義:

   因為 radis hash 的每個 key 中，都會包含很多個 field，而一個 field 會對應到一個 value，所以我們在這邊 hash table 的本體是個 array，array 的每個格子放的是一個 hash_node 的 Pointer，會對應到 key 以及網下串接下去的 hash_node，而每個 hash_node 又會有自己的 field_node linked list 來記錄這個 key(即 hash node)包含了哪些 field:value。

```c
struct field_node{
    char* key;
    char * field;
    char * value;
    struct field_node* next_field_node;
};
struct hash_node{
    char* key;
    struct field_node* field_node_head;
    struct hash_node* ptr_to_next_hash_node;
};
struct hash_TB_array {
    struct hash_node* TB_array[MAXHASHTABLESIZE];//這邊存放的會是第一個hash node
};
```

   b. 計算 load factor 並在需要時調整整個 hash table 的大小

根據要求我們要在 hash table 的 load factor 太大時調整 hash table 的大小，所以我這邊是設定當 load factor 大於 0.1 時要重新建立一個 table size 為當前 10 倍的 hash table 並且將原本的 table 覆蓋，實際的實作方式如以下這個 function，最後會 return 新的 hash table。

```c
struct hash_TB_array* change_table_size(struct hash_TB_array* ptr_to_TB_array,int *cur_size){
    double load_factor=ptr_to_TB_array->key_num /(*cur_size);
    struct hash_TB_array *newHashTable = (struct hash_TB_array *)malloc(sizeof(struct hash_TB_array));
    if(load_factor>0.1){
        (*cur_size)=(*cur_size)*10;
        for (int i = 0; i < MAXHASHTABLESIZE; i++) {
            struct hash_node* cur_node = ptr_to_TB_array->TB_array[i];
            while (cur_node != NULL){
                struct field_node* cur_field_node=cur_node->field_node_head;
                while (cur_field_node!=NULL){
                    int index=hash(key,cur_size);
                    insert(newHashTable,cur_field_node->key,index,cur_field_node->field,cur_field_node->value);
                    cur_field_node=cur_field_node->next_field_node;
                }
                printf("\n");
                cur_node = cur_node->ptr_to_next_hash_node;
            }
        }
    }
    return newHashTable;
}
```

c. **HSET function**:這邊的 insert 基本上就是一般的插入，如果當前這個 hash node 是空的就向系統要空間，如果不為空，那這個 field node 就直接串在後面。

```
void insert(struct hash_TB_array* ptr_to_TB_array, char* key,char* field_name,char* value) {//包括插入和更新
    int index = hash(key);
    struct hash_node* cur_node=ptr_to_TB_array->TB_array[index];

    while(cur_node!=NULL){
        if(strcmp(cur_node->key,key)==0){//找到key的位置了
            //printf("1\n");
            struct field_node* cur_field_node=cur_node->field_node_head;
            //printf("2\n");
            while (cur_field_node!=NULL){
                if(strcmp(cur_field_node->field,field_name)==0){//把value覆蓋到已經建立好的filed就好
                    strcpy(cur_field_node->value, value);
                    return;
                }
                cur_field_node=cur_field_node->next_field_node;
            }
            struct field_node* ptr_to_new_field_node=(struct field_node*)malloc(sizeof(struct field_node));
            ptr_to_new_field_node->field =(char*)malloc(strlen(field_name)+1);
            strcpy(ptr_to_new_field_node->field, field_name);
            ptr_to_new_field_node->value =(char*)malloc(strlen(value)+1);
            strcpy(ptr_to_new_field_node->value, value);
            ptr_to_new_field_node->next_field_node=cur_node->field_node_head;
            cur_node->field_node_head=ptr_to_new_field_node;//把新的filed node接在這個hash jey node的head
            return;
        }
```

```
        cur_node=cur_node->ptr_to_next_hash_node;
    }
    struct hash_node* ptr_to_new_hash_node = (struct hash_node*)malloc(sizeof(struct hash_node));
    ptr_to_new_hash_node->key = (char*)malloc(strlen(key) + 1);
    strcpy(ptr_to_new_hash_node->key, key);
    struct field_node* ptr_to_new_field_node=(struct field_node*)malloc(sizeof(struct field_node));
    ptr_to_new_field_node->field =(char*)malloc(strlen(field_name)+1);
    strcpy(ptr_to_new_field_node->field, field_name);
    ptr_to_new_field_node->value =(char*)malloc(strlen(value)+1);
    strcpy(ptr_to_new_field_node->value, value);
    ptr_to_new_hash_node->field_node_head=ptr_to_new_field_node;//為了處理碰撞問題，Array存放的Pointer會指向Linked list的開
    ptr_to_new_hash_node->ptr_to_next_hash_node = ptr_to_TB_array->TB_array[index];
    ptr_to_TB_array->TB_array[index] = ptr_to_new_hash_node;
    ptr_to_TB_array->key_num++;
}
```

d. **HDEL function**:這邊也是檢查到對應的 key 和 filed 後，就把該 field_node 刪除，如果該 hash_node 的所有成員都被刪除了，這邊的刪除沒啥問題

```
void delete_field(struct hash_TB_array* ptr_to_TB_array, char* key,char* field_name) {//包括插入和更新
    int index = hash(key);
    struct hash_node* prev_node=NULL;
    struct hash_node* cur_node=ptr_to_TB_array->TB_array[index];
    int key_node_empty_bool=0;
    while(cur_node!=NULL){
        if(strcmp(cur_node->key,key)==0){//找到key的位置了
            struct field_node* cur_field_node=cur_node->field_node_head;//取出當前field_node的head
            struct field_node* prev_field_node=NULL;

            while (cur_field_node!=NULL){
                if(strcmp(cur_field_node->field,field_name)==0){//把這個value-filed node刪除就好
                    if(prev_field_node==NULL){//表示要刪除的node再第一個(head)
                        if(cur_field_node->next_field_node==NULL)key_node_empty_bool=1;
                        cur_node->field_node_head=cur_field_node->next_field_node;
                    }
                    else prev_field_node->next_field_node=cur_field_node->next_field_node;//要刪除的node在第二個或之後
                    if(key_node_empty_bool==0){
                        printf("%s : %s deleted successfully\n",key,field_name);
                        return ;
                    }//表示不用刪除整個key node
                }
                prev_field_node=cur_field_node;
                cur_field_node=cur_field_node->next_field_node;
            }
```

```
            if(key_node_empty_bool==1){
                if(prev_node!=NULL){
                    prev_node->ptr_to_next_hash_node=cur_node->ptr_to_next_hash_node;
                }
                else ptr_to_TB_array->TB_array[index]=cur_node->ptr_to_next_hash_node;
                ptr_to_TB_array->key_num--;
                printf("%s : %s deleted successfully\n",key,field_name);
                return;
            }
        }
        prev_node=cur_node;
        cur_node=cur_node->ptr_to_next_hash_node;
    }
    printf("%s : %s not found\n",key,field_name);
    return;//刪除失敗
}
```

e. HGET:這邊的查詢也是利用 hash 找到對應的 index 後，再往 collision chain 搜尋對應的 node。

```
char* get(struct hash_TB_array* ptr_to_TB_array, char* key,char* field_name){
    int index = hash(key);
    struct hash_node* cur_node=ptr_to_TB_array->TB_array[index];
    int find_key_bool=0;
    while(cur_node!=NULL){
        if(strcmp(cur_node->key,key)==0){//找到key的位置了
            find_key_bool=1;
            struct field_node* cur_field_node=cur_node->field_node_head;//取出當前field_node的head

            while (cur_field_node!=NULL){
                if(strcmp(cur_field_node->field,field_name)==0){//把這個value-filed node刪除就好
                    return cur_field_node->value;
                }

                cur_field_node=cur_field_node->next_field_node;
            }

        }
        if(find_key_bool==1) return NULL;
        cur_node=cur_node->ptr_to_next_hash_node;
    }
    //表示沒找到
    return NULL;
}
```

f. EXPIRE():這邊作業的要求是要用 Libev 計時，再指定秒數之後刪除該 node，所以這邊是用 libev 監聽後繼時，再依照時間刪除

```
void delayed_delete(EV_P_ ev_timer *w, int revents) {
    struct delayed_data *data = (struct delayed_data *)w->data;
    delete_key(data->myHashTable, data->key); // 在延遲後執行 delete() 函數
    free(data->key);
    free(data);
}
```

```
else if(strcmp(command,"EXPIRE")==0){
    //printf("1\n");
    struct delayed_data *expire_data = (struct delayed_data *)malloc(sizeof(struct delayed_data));
    //printf("2\n");
    expire_data->myHashTable=(struct hash_TB_array *)malloc(sizeof(struct hash_TB_array));
    expire_data->myHashTable=myHashTable;
    expire_data->key=(char *)malloc(sizeof(char));
    strcpy(expire_data->key ,key);
    //printf("3\n");
    int delay_second = atoi(fields[0]); // 將字串轉換為整數
    //printf("4\n");
    ev_timer *delay_timer = (ev_timer *)malloc(sizeof(ev_timer));
    ev_timer_init(delay_timer, delayed_delete, delay_second, 0.0); // 5 秒延遲
    //printf("5\n");
    delay_timer->data =expire_data;
    ev_timer_start(loop, delay_timer);
}
```

g. Main():main()也是用 libev 進行事件監聽，並且不斷循環，具體實作方式如下:

```c
int main() {
    // 初始化 libev
    struct ev_loop *loop = EV_DEFAULT;
    ev_io stdin_watcher;
    struct hash_TB_array *myHashTable = (struct hash_TB_array *)malloc(sizeof(struct hash_TB_array));

    int i;
    myHashTable->key_num=0;
    for (i = 0; i < MAXHASHTABLESIZE; i++) {
        myHashTable->TB_array[i] = NULL;
    }

    // 開始監聽標準輸入，並傳遞哈希表給回調函數
    ev_io_init(&stdin_watcher, stdin_cb, 0, EV_READ);
    stdin_watcher.data = myHashTable; // 傳遞哈希表
    ev_io_start(loop, &stdin_watcher);

    // 開始 libev 主循環
    ev_run(loop, 0);

    return 0;
}
```

3. 編譯方式:

gcc -c redis_hash.c -o redis_hash.o

gcc -c redis_dll.c -o redis_dll.o

gcc -c redis_str.c -o redis_str.o

gcc -o main main.c redis_dll.o redis_str.o redis_hash.o -lev

4. 在 Ubuntu 上的執行結果

```
eason@LAPTOP-Q69P3FAE:/mnt/c/Users/user/DS_HW6$ gcc -o test test.c
eason@LAPTOP-Q69P3FAE:/mnt/c/Users/user/DS_HW6$ ./test
HSET key1 field1 value1 field2 value2
please input the command(HSET, HGET, HDEL,EXPIRE, SHOW): HSET key2 field1 value1 field2 value2
please input the command(HSET, HGET, HDEL,EXPIRE, SHOW): HSET key3 field1 value1 field2 value2 field3 value3
please input the command(HSET, HGET, HDEL,EXPIRE, SHOW): HSET key4 field1 value1 field2 value2 field3 value3
 field4 value4
please input the command(HSET, HGET, HDEL,EXPIRE, SHOW): SHOW
All the key members:
key: key1
value: value2 field: field2 , value: value1 field: field1
key: key2
value: value2 field: field2 , value: value1 field: field1
key: key3
value: value3 field: field3 , value: value2 field: field2 , value: value1 field: field1
key: key4
value: value4 field: field4 , value: value3 field: field3 , value: value2 field: field2 , valu
d: field1
please input the command(HSET, HGET, HDEL,EXPIRE, SHOW): HDEL key3 field2
key3 : field2 deleted successfully
```

HSET 可以一次在同一個 key 中插入很多組 filed :value

使用 HSET 建立資料

使用 SHOW 印出當前在 table 中的所有成員

使用 HDEL 印刪除特定 field node，成功刪除會顯示成功

Key3:filed2 已經被刪除了，所以無法被 HGET

```
please input the command(HSET, HGET, HDEL,EXPIRE, SHOW): HGET key3 field2
the value to  key3 : field2 not found
please input the command(HSET, HGET, HDEL,EXPIRE, SHOW): HGET key3 field1
the value to  key3 : field1 is value1
please input the command(HSET, HGET, HDEL,EXPIRE, SHOW): EXPIRE key1 10
please input the command(HSET, HGET, HDEL,EXPIRE, SHOW): SHOW
All the key members:
key: key1
value: value2 field: field2 , value: value1 field: field1
key: key2
value: value2 field: field2 , value: value1 field: field1
key: key3
value: value3 field: field3 , value: value1 field: field1
key: key4
value: value4 field: field4 , value: value3 field: field3 , value: value2 field: field2 , value: value1 fiel
d: field1
```

使用 HGET 找出 Key3:filed1 的 value

EXPIRE 測試，在 10 秒數完以前，先 SHOW 一次，確認 key1 的資料都還在

5. 說明:這次作業的困難點主要是用事件監聽，要確保特定的指令執行完成之後，才能執行其他指令，這個部分跟寫網頁後端有點像，感覺如果將來要當後端工程師，網頁的知識還是要具備一下才會比較順利。

   感謝助教的批改 祝助教新年快樂!!!