壹、程式解釋

設計方向：

此作業繼承作業五的內容增加了操作hash的相關操作，將原本的key-value 中 value union 結構內新增一個 HASH_TABLE 的結構，裡面包含了 HASH_NODE 結構的陣列，陣列大小預設是 10、load factor 為 0.7，如果 node 數量超過 table 的 70% 就會重新建一個長度為原本兩倍的 HASH_NODE 陣列。作業六中將會利用此 HASH_TABLE 結構來實現 redis 的 HSET、HGET、HDEL功能，另外此作業會額外引入 ev.h 函式庫，來實現redis的自動 expire 功能和將原本的程式改成事件觸發導向的程式。同於作業五，會將上述相關的函數都打包在一個nosql.c 檔中，並作為函式庫提供main.c 使用。在main.c 中模仿實際操作redis-cli 的個個指令方式設計。

nosql.c 新增程式碼解釋：

第 52~72 行：

此函數在給定資料庫下中搜尋 key 如果不存在就會輸出錯誤；如果存在會先將 timeout 事件停止在重新初始化 timeout 事件，最後重啟 timeout 事件，並回傳 0 代表程式執行成功。

第 21~50 行：

此函數為 timeout 事件觸發會呼叫的 callback 函數，會去一一比對每個 node 的 timer 物件是否和觸發的 timer 物件地址一樣，如果一樣就代表該 node 的 timer 觸發，也就是 timeout 了，之後先釋放記憶體空間和將該 node 的前一個 node 的 next 指位器指向該 node 的下一個 node，最後就釋放該 node 的空間。

main.c 新增程式碼解釋：

第 17 行：

宣告一個 io 事件監聽。

第 23 行：

初始化 io 事件監聽。

第 24 行：

啟動 io 事件監聽。

第 26 行：

以默認模式啟動 io 事件監聽循環。

第 29~579 行：

將原本 while 迴圈重複執行的判斷移到此 callback 函數中，此函數在 io 事件觸發後會被呼叫。

第 581~583 行：

回傳 my_redis 資料庫的地址。

貳、完整程式碼(新增的部分)

nosql.c:

```
#define TYPE_NONE 5
#define LOAD_FACTOR 0.7
```

```c
#define INITIAL_EXPIRE_TIME 600
// #define INIT_HASH_TABLE_SIZE 10
extern db** get_db();

void delete_expired_cb(EV_P_ ev_timer *w, int revents){
    db** nosqldb = get_db();
    db* tmp_node = *nosqldb;
    db* pre_node = NULL;
    while(tmp_node->next != NULL) { //search every node to find
whether have the same key existed
        // printf("what?\n");
        if(&(tmp_node->timeout) != w) { //search next node
!!!!!len不一様比較
            // printf("where are you? %s\n", tmp_node->key);
            pre_node = tmp_node;
            tmp_node = tmp_node->next;
        }
        else{
            // printf("I found you!\n");
            if(pre_node){
                // printf("you aren't the first one!");
                pre_node->next = tmp_node->next;
            }
            else{
                // printf("you are the first one!");
                *nosqldb = (*nosqldb)->next;
                *nosqldb = init(*nosqldb);
            }
            free(tmp_node->key);
            tmp_node->key = NULL;
            free(tmp_node);
            tmp_node = NULL;
            break;
        }
    }
}

int set_timeout(db* nosqldb, char* key, int time_s){
    db * tmp_node = nosqldb;
    int find_key = 0;
    while(tmp_node->next != NULL) { //search every node to find
whether have the same key existed
```

```
        if(strcmp(tmp_node->key, key)) { //search next node
!!!!!len不一様比較
            tmp_node = tmp_node->next;
        }
        else{
            find_key = 1;
            ev_timer_stop(loop, &tmp_node->timeout);
            ev_timer_init(&tmp_node->timeout, delete_expired_cb,
time_s, 0.);
            ev_timer_start (loop, &tmp_node->timeout);
            break;
        }
    }
    if(!find_key){
        printf("key not found in timeout\n");
        return 1;
    }
    return 0;
}

int hash(char* key, int hash_table_size){
    int c, sum=0;
    while(c=*key++){
        sum+=c;
    }
    return sum % hash_table_size;
}

void free_hashed_table(HASH_NODE ** table){
    HASH_NODE* hash_node;
    while(hash_node = *table++){
        HASH_NODE* pre_node;
        while(hash_node){
            pre_node = hash_node;
            hash_node = hash_node->next;
            free(pre_node->field);
            free(pre_node->value);
            free(pre_node);
        }
    }
    printf("table NULL? %d\n", table==NULL);
    // free(table);
}
```

```
void init_hash_node(HASH_NODE* hash_node, char* field, char*
value){
    // hash_node = (HASH_NODE*) malloc(sizeof(HASH_NODE));
    hash_node->field = (char*)
malloc(sizeof(char)*(strlen(field)+1));
    hash_node->value = (char*)
malloc(sizeof(char)*(strlen(value)+1));
    strcpy(hash_node->field, field);
    strcpy(hash_node->value, value);
    hash_node->next = NULL;
}

void resize_hash_table(HASH_TABLE* hash_table){
    int new_size = hash_table->table_size * 2;
    HASH_NODE** new_table = (HASH_NODE**) calloc(new_size,
sizeof(HASH_NODE*));
    for(int i=0;i<hash_table->table_size;i++){ //把舊table的資料搬運
到新table
        HASH_NODE* tmp_hash_node = hash_table->table[i];
        while(tmp_hash_node){
            HASH_NODE* appended_hash_node;
            while(appended_hash_node){
                appended_hash_node = appended_hash_node->next;
            }
            appended_hash_node = (HASH_NODE*)
malloc(sizeof(HASH_NODE));
            init_hash_node(appended_hash_node,
tmp_hash_node->field, tmp_hash_node->value);
            new_table[hash(tmp_hash_node->field,new_size)] =
appended_hash_node;
            tmp_hash_node = tmp_hash_node->next;
        }
    }
    free_hashed_table(hash_table->table);
    free(hash_table->table);
    hash_table->table = new_table;
    hash_table->table_size = new_size;
}

int hash_del(db* nosqldb, char* hash_table_name, char* field){
    db *tmp_node = nosqldb;
    HASH_TABLE* tmp_hash_table;
```

```
    int found_table = 0;
    while(tmp_node->next != NULL) {
        //判斷hash_table有沒有存在
        if(!strcmp(tmp_node->key, hash_table_name)){
            if(tmp_node->value_type != TYPE_HASH_TABLE){
                return -1; //type error
            }
            found_table = 1;
            tmp_hash_table = tmp_node->value.hash_table;
            break;
        }
        tmp_node = tmp_node->next;
    }
    if(!found_table){
        return 0;
    }
    HASH_NODE* hash_node = tmp_hash_table->table[hash(field,
tmp_hash_table->table_size)];
    HASH_NODE* pre_hash_node = NULL;
    while(hash_node){
        if(!strcmp(hash_node->field, field)){
            if(pre_hash_node){
                pre_hash_node->next = hash_node->next;
            }
            else{
                tmp_hash_table->table[hash(field,
tmp_hash_table->table_size)] = hash_node->next;
            }
            free(hash_node->field);
            free(hash_node->value);
            free(hash_node);
            return 1;
        }
        pre_hash_node = hash_node;
        hash_node = hash_node->next;
    }
    return 0;
}

char* hash_get(db* nosqldb, char* hash_table_name /*node的key*/,
char* field){
    db *tmp_node = nosqldb;
    HASH_TABLE* tmp_hash_table;
```

```
    int found_table = 0;
    while(tmp_node->next != NULL) {
        //判斷hash_table有沒有存在
        if(!strcmp(tmp_node->key, hash_table_name)){
            if(tmp_node->value_type != TYPE_HASH_TABLE){
                printf("type error\n");
                return NULL; //type error
            }
            found_table = 1;
            tmp_hash_table = tmp_node->value.hash_table;
            break;
        }
        tmp_node = tmp_node->next;
    }
    if(!found_table){
        printf("NULL\n");
        return NULL;
    }
    HASH_NODE* hash_node = tmp_hash_table->table[hash(field,
tmp_hash_table->table_size)];
    while(hash_node){
        if(!strcmp(hash_node->field, field)){
            return hash_node->value;
        }
        hash_node = hash_node->next;
    }
    printf("NULL\n");
    return NULL;
}

int hash_set(db* nosqldb, char* hash_table_name /*node的key*/,
char* field, char* value){
    db *tmp_node = nosqldb;
    HASH_TABLE* tmp_hash_table;
    int found_table = 0;
    while(tmp_node->next != NULL) {
        //判斷hash_table有沒有存在
        if(!strcmp(tmp_node->key, hash_table_name)){
            if(tmp_node->value_type != TYPE_HASH_TABLE){
                return -1; //type error
            }
            found_table = 1;
            tmp_hash_table = tmp_node->value.hash_table;
```

```
            break;
        }
        tmp_node = tmp_node->next;
    }
    if(found_table){ //找到table, 要找有沒有該feild
        int key_idx = hash(field, tmp_hash_table->table_size);
        HASH_NODE* tmp_hash_node =
tmp_hash_table->table[key_idx];
        HASH_NODE* pre_hash_node = NULL;
        while(tmp_hash_node){
            if(!strcmp(tmp_hash_node->field, field)){
                tmp_hash_node->value = (char*)
realloc(tmp_hash_node->value, (strlen(value)+1)*sizeof(char));
                strcpy(tmp_hash_node->value, value);
                return 0; //update field
            }
            pre_hash_node = tmp_hash_node;
            tmp_hash_node->next;
        }
        if(!pre_hash_node){ //hash table該index為空
            tmp_hash_table->table[key_idx] = (HASH_NODE*)
malloc(sizeof(HASH_NODE));
            init_hash_node(tmp_hash_table->table[key_idx], field,
value);
            tmp_hash_table->num_node++;
            // 判斷load factor

if((float)tmp_hash_table->num_node/tmp_hash_table->table_size >
LOAD_FACTOR){
                printf("%f\n",
(float)tmp_hash_table->num_node/tmp_hash_table->table_size);
                resize_hash_table(tmp_hash_table);
            }
            return 1; //創新field
        }
        else{
            pre_hash_node->next = (HASH_NODE*)
malloc(sizeof(HASH_NODE));
            init_hash_node(pre_hash_node->next, field, value);
            return 1; //創新field
        }
    }
    // 沒找到hash table
```

```
    tmp_node->key = (char*)
malloc(sizeof(char)*(strlen(hash_table_name)+1));
    strcpy(tmp_node->key, hash_table_name);
    tmp_node->value_type = TYPE_HASH_TABLE;
    tmp_node->value.hash_table = (HASH_TABLE*)
malloc(sizeof(HASH_TABLE));
    tmp_node->value.hash_table->table_size =
INIT_HASH_TABLE_SIZE;
    tmp_node->value.hash_table->num_node = 1;
    tmp_node->value.hash_table->table = (HASH_NODE**)
calloc(INIT_HASH_TABLE_SIZE, sizeof(HASH_NODE*));
    // memset(tmp_node->value.hash_table->table, NULL,
INIT_HASH_TABLE_SIZE*sizeof(HASH_NODE*)); //把hash table的值預設為
NULL
    HASH_NODE* hash_node = (HASH_NODE*)
malloc(sizeof(HASH_NODE));
    init_hash_node(hash_node, field, value);
    tmp_node->value.hash_table->table[hash(field,
INIT_HASH_TABLE_SIZE)] = hash_node;

    tmp_node->next = init(tmp_node->next);
    set_timeout(nosqldb, hash_table_name, INITIAL_EXPIRE_TIME);
    return 1;
}
```

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ev.h>
#include "nosql.h"

#define MAX_KEY_LEN 15
#define MAX_VALUE_LEN 30
#define LINE_LEN 150
#define TRUE 1
#define COMMAND_LEN 16

void run_redis_cb();
db** get_db();
db * my_redis;
// struct ev_loop *loop;
ev_io stdin_watcher;
```

```
int main(){
    my_redis = init(my_redis);

    loop = EV_DEFAULT;
    ev_io_init(&stdin_watcher, run_redis_cb, 0, EV_READ);
    ev_io_start(loop, &stdin_watcher);

    ev_run(loop, 0);
}

void run_redis_cb(){
    char cmd[COMMAND_LEN+1];
    char key[MAX_KEY_LEN+1];
    char hash_table_key[MAX_KEY_LEN+1];
    char value[MAX_VALUE_LEN+1];
    char line[LINE_LEN+2]; //多一個'\n'字元
    scanf("%16s", cmd);
        if(!strcmp(cmd, "SET") || !strcmp(cmd, "set")){
            scanf("%15s %30s", key, value);
            add_or_update_data(my_redis, key, value);
            // printf("OK\n");
        }

            .
            .
            .
            .

        else if(!strcmp(cmd, "HSET") || !strcmp(cmd, "hset")){
            scanf("%15s %15s %30s", hash_table_key, key, value);
            int result = hash_set(my_redis, hash_table_key, key,
value);
            if(result == -1){
                printf("type error\n");
            }
            else{
                printf("%d\n", result);
            }
        }
        else if(!strcmp(cmd, "HGET") || !strcmp(cmd, "hget")){
            scanf("%15s %15s", hash_table_key, key);
```

```
            char* result = hash_get(my_redis, hash_table_key,
key);
            if(result){
                printf("%s\n", result);
            }
        }
        else if(!strcmp(cmd, "HDEL") || !strcmp(cmd, "hdel")){
            scanf("%15s %15s", hash_table_key, key);
            int result = hash_del(my_redis, hash_table_key, key);
            if(result==-1){
                printf("type error\n");
            }
            else{
                printf("%d\n", result);
            }
        }
        else if(!strcmp(cmd, "EXPIRE") || !strcmp(cmd,
"expire")){
            int time_s;
            scanf("%15s %d", key, &time_s);

            int result = set_timeout(my_redis, key, time_s);
            if(!result){
                printf("0\n");
            }
        }
        else if(!strcmp(cmd, "EXIT") || !strcmp(cmd, "exit")){
            // break;
            ev_break(EV_A_ EVBREAK_ALL);
        }
        else{
            printf("command not found\n");
        }
}

db** get_db(){
    return &my_redis;
}
```