資料結構 HW_2

110303581 陳俊瑋

➢ Mylib.h

其內容與 HW1 的前半段幾乎一樣，也就是定義 KeyValue 和 DataBase 的結構，還有定義四個操作(CRUD)的函數。由於前份作業已經提過就不在贅述細節。有些不一樣是在我多定義了 `pre_insert` 和 `read_notprint` 這兩個函式，為的是不要在 Set 假資料和 Get 假資料的時候做 `printf` 的動作。

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define INPUT_SIZE 100
#define COMMAND_SIZE 20
#define KEY_SIZE 50
#define VALUE_SIZE 100

typedef struct KeyValue {
    char key[KEY_SIZE];
    char value[VALUE_SIZE];   //age
} KeyValue;

typedef struct Database {
    KeyValue* data;     //存一堆 KeyValue 的指標
    int size;
} Database;

Database* createDatabase() {
    Database* db = (Database*)malloc(sizeof(Database));
    db->size = 0;
    db->data = NULL;
    return db;
}

//創建 inset
void insert(Database* db, const char* key, const char* value) {
    db->data = (KeyValue*)realloc(db->data, sizeof(KeyValue) *
(size_t)(db->size + 1));

    strcpy(db->data[db->size].key, key);
    strcpy(db->data[db->size].value, value);
    db->size++;

    printf("Insert [%s] Successfully.\n", key);
```

```c
}

//不會顯示 Successful 的創建
void pre_insert(Database* db, const char* key, const char* value) {
    db->data = (KeyValue*)realloc(db->data, sizeof(KeyValue) *
(size_t)(db->size + 1));

    strcpy(db->data[db->size].key, key);
    strcpy(db->data[db->size].value, value);
    db->size++;
}

//讀取 read
void read(Database* db, const char* key) {
    for (int i = 0; i < db->size; i++) {
        if (strcmp(db->data[i].key, key) == 0) {
            printf("{\n");
            printf("    Name: %s\n", db->data[i].key);
            printf("    Age: %s\n", db->data[i].value);
            printf("}\n");
            return;
        }
    }
    printf("Not Found [%s] in Database.\n", key);
}

//不會 printf 的讀取
void read_notprint(Database* db, const char* key) {
    for (int i = 0; i < db->size; i++) {
        if (strcmp(db->data[i].key, key) == 0) {
            return;
        }
    }
    printf("Not Found [%s] in Database.\n", key);
}

//更新 update
void update(Database* db, const char* key, const char* value) {
    for (int i = 0; i < db->size; i++) {
        if (strcmp(db->data[i].key, key) == 0) {
            strcpy(db->data[i].value, value);
            printf("Update [%s] Successfully\n", key);
            return;
        }
    }
    printf("[%s] Not Found. Cannot update.\n", key);
}
```

```c
//刪除 delete
void del(Database* db, const char* key) {
    for (int i = 0; i < db->size; i++) {
        if (strcmp(db->data[i].key, key) == 0) {
            // 將後面的資料往前移動
            for (int j = i; j < db->size - 1; j++) {
                strcpy(db->data[j].key, db->data[j + 1].key);
                strcpy(db->data[j].value, db->data[j + 1].value);
            }
            db->size--;

            printf("Deleted [%s] Successfully.\n", key);
            return;
        }
    }
    printf("Key not found. Cannot remove.\n");
}
```

- hw2.c

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <malloc.h>
#include "mylib.h"
#include <hiredis/hiredis.h>

#define FAKE_DATA_NUM 1000000
```

前面先是 include 會用到的函式，包括前面的 mylib.h 和要來拿進行比對的 hiredis.h。我也 define 了一個變數 `FAKE_DATA_NUM` 來給定假資料的數量。

```c
int main() {
    redisContext *redis_conn = redisConnect("127.0.0.1", 6379); // 初始化 Redis 連接
    if (redis_conn == NULL || redis_conn->err) {
        if (redis_conn) {
            printf("Redis connection error: %s\n", redis_conn->errstr);
            return 1;
        } else {
            printf("Can't allocate Redis context\n");
            return 1;
        }
    }
```

這邊進到 `main()`，先對 Redis 的連接進行初始化。我也多寫了一串 `if-else` 來判斷有沒有與 Redis 成功連接，並且輸出 error。

```c
    Database* db = createDatabase();
    char input[INPUT_SIZE];

    // insert(db, "William", "20");
    // insert(db, "Una", "19");

    // 生成假資料
    srand(time(NULL));
    double MYREDIS_CREATE_START, MYREDIS_CREATE_END, MYREDIS_GET_START, MYREDIS_GET_END, HIREDIS_CREATE_START, HIREDIS_CREATE_END, HIREDIS_GET_START, HIREDIS_GET_END;
    int FAKE_Data[FAKE_DATA_NUM];
    for (int i = 0; i < FAKE_DATA_NUM; i++){
        char key[KEY_SIZE];
        char value[VALUE_SIZE];
        int age = (rand() % 99) + 1;
```

```
        FAKE_Data[i] = age;
    }
    printf("Fake Data Done.\n");
```

一開始先初始化資料庫，然後再定義一個陣列(input)用來存取使用者的輸入。

之後我定義了一堆變數，是後面計算時間要用的。

我生成假資料的方式是 key 是從 0~999999，而 value 則是 1~99 隨機選一個數字。

```
    // 放進 myredis
    MYREDIS_CREATE_START = clock();
    for (int i = 0; i < FAKE_DATA_NUM; i++){
        char key[KEY_SIZE];
        char value[VALUE_SIZE];
        sprintf(key, "%d", i);
        sprintf(value, "%d", FAKE_Data[i]);

        pre_insert(db, key, value);
    }
    MYREDIS_CREATE_END = clock();
    printf("set MYREDIS\n");

    //myredis 使用的記憶體容量
    struct mallinfo info = mallinfo();
    int MYREDIS_total_allocated = info.arena;
    printf("MYREDIS memory\n");
```

這邊將假資料存放進我的資料庫(myredis)，並在開始前和結束使用 `clock()`記錄 CPU 的所使用的時間。最後用`<malloc.h>`裡的函數 `mallinfo()`獲取記憶體的分配資訊，其中`.arena` 是記憶體總量。

```
    //hiredis SET 前的記憶體容量
    redisReply *info_reply_for_memory_Before = redisCommand(redis_conn,
"INFO");
    long long redis_memory_used_Before = 0;
    if (info_reply_for_memory_Before != NULL) {
        char *info_str = info_reply_for_memory_Before->str;
        char *mem_usage_pos = strstr(info_str, "used_memory:");
        if (mem_usage_pos) {
            sscanf(mem_usage_pos + strlen("used_memory:"), "%lld",
&redis_memory_used_Before);
        }
    }
    freeReplyObject(info_reply_for_memory_Before);  // 釋放命令
    printf("HIREDIS memory before\n");
```

```
    // 放進 hiredis
    HIREDIS_CREATE_START = clock();
    for (int i = 0; i < FAKE_DATA_NUM; i++){
        char key[KEY_SIZE];
        char value[VALUE_SIZE];
        sprintf(key, "%d", i);
        sprintf(value, "%d", FAKE_Data[i]);

        redisReply *set_reply = redisCommand(redis_conn, "SET %s %s",
key, value);
        freeReplyObject(set_reply);
    }
    HIREDIS_CREATE_END = clock();
    printf("set HIREDIS\n");

    //hiredis SET 後的記憶體容量
    redisReply *info_reply_for_memory_After = redisCommand(redis_conn,
"INFO");
    long long redis_memory_used_After = 0;
    if (info_reply_for_memory_After != NULL) {
        char *info_str = info_reply_for_memory_After->str;
        char *mem_usage_pos = strstr(info_str, "used_memory:");
        if (mem_usage_pos) {
            sscanf(mem_usage_pos + strlen("used_memory:"), "%lld",
&redis_memory_used_After);
        }
    }
    freeReplyObject(info_reply_for_memory_After);

    long long redis_memory_used = redis_memory_used_After -
redis_memory_used_Before;
    printf("HIREDIS memory after\n");
```

此大段程式碼分為三個部分: 紀錄 hiredis 傳入假資料前的容量、將假資料傳入 hiredis、以及紀錄 hiredis 傳入假資料後的容量。

記錄傳入前和傳入後的程式類似，都是在 hiredis 發送 `INFO` 命令， 而後判斷有沒有成功得到資料(不等於 `NULL`)，然後在用`->str` 找到回傳資料的字串。再用 `strstr` 標記 `used_memory:`這個字串的位置，最後用 `sscanf` 提取 `used_memory:`後面的字串，得到使用的記憶體容量，同時也釋放命令的記憶體。

我將前後所得到的記憶體容量做相減，以獲得傳入假資料所使用的容量。

```
    //從 myredis GET 全部資料
    MYREDIS_GET_START = clock();
    for (int i = 0; i < FAKE_DATA_NUM; i++){
        char key[KEY_SIZE];
        sprintf(key, "%d", i);
```

```
        read_notprint(db, key);
    }
    MYREDIS_GET_END = clock();
    printf("myredis GET\n");

    //從 hiredis GET 全部資料
    HIREDIS_GET_START = clock();
    for (int i = 0; i < FAKE_DATA_NUM; i++){
        char key[KEY_SIZE];
        sprintf(key, "%d", i);

        redisReply *get_reply = redisCommand(redis_conn, "GET %s", key);
        freeReplyObject(get_reply);
    }
    HIREDIS_GET_END = clock();
    printf("hiredis GET\n");
```

這邊是計算 myredis 和 hiredis 讀取這一百萬筆假資料需要多少的時間，也一樣使用 `clock()`來記錄時間，然後用 `for` 來 Get 每一筆資料。

```
    //計算時間
    double MYREDIS_create_total_time = (MYREDIS_CREATE_END -
MYREDIS_CREATE_START) / CLOCKS_PER_SEC;
    double MYREDIS_create_per_time = MYREDIS_create_total_time /
FAKE_DATA_NUM;

    double MYREDIS_GET_total_time = (MYREDIS_GET_END -
MYREDIS_GET_START) / CLOCKS_PER_SEC;
    double MYREDIS_GET_per_time = MYREDIS_GET_total_time /
FAKE_DATA_NUM;


    double HIREDIS_create_total_time = (HIREDIS_CREATE_END -
HIREDIS_CREATE_START) / CLOCKS_PER_SEC;
    double HIREDIS_create_per_time = HIREDIS_create_total_time /
FAKE_DATA_NUM;

    double HIREDIS_GET_total_time = (HIREDIS_GET_END -
HIREDIS_GET_START) / CLOCKS_PER_SEC;
    double HIREDIS_GET_per_time = HIREDIS_GET_total_time /
FAKE_DATA_NUM;
```

這邊是計算前面用 `clock()`來提取的時間，用相減的方式來計算每個步驟所花的時間。

```
    // MYREDIS
    printf("MYREDIS:\n");
    printf("MYREDIS Create %d set of Data Total Time: %lf sec\n",
FAKE_DATA_NUM, MYREDIS_create_total_time);
    printf("MYREDIS Create Per Data Time: %lf sec\n",
MYREDIS_create_per_time);

    printf("MYREDIS Total Memory Allocated: %d bytes\n",
MYREDIS_total_allocated);

    printf("MYREDIS GET %d set of Data Total Time: %lf sec\n",
FAKE_DATA_NUM, MYREDIS_GET_total_time);
    printf("MYREDIS GET Per Data Time: %lf sec\n",
MYREDIS_GET_per_time);

    printf("============================================================
================\n");

    // HIREDIS
    printf("HIREDIS:\n");
    printf("HIREDIS Create Total Time: %lf sec\n",
HIREDIS_create_total_time);
    printf("HIREDIS Create Per Data Time: %lf sec\n",
HIREDIS_create_per_time);

    printf("HIREDIS Memory Used: %lld bytes\n", redis_memory_used);

    printf("HIREDIS GET %d set of Data Total Time: %lf sec\n",
FAKE_DATA_NUM, HIREDIS_GET_total_time);
    printf("HIREDIS GET Per Data Time: %lf sec\n",
HIREDIS_GET_per_time);
```

而後 print 程式的結果。

至於後面就是 `do-while` 迴圈，讓使用者可以一直輸入新的指令，跟上次作業 HW1 一樣就不再贅述。

輸出結果:

```
MYREDIS:
MYREDIS Create 1000000 set of Data Total Time: 0.915907 sec
MYREDIS Create Per Data Time: 0.000001 sec
MYREDIS Total Memory Allocated: 135168 bytes
MYREDIS GET 1000000 set of Data Total Time: 6004.177431 sec
MYREDIS GET Per Data Time: 0.006004 sec
==========================================================================
HIREDIS:
HIREDIS Create Total Time: 11.996146 sec
HIREDIS Create Per Data Time: 0.000012 sec
HIREDIS Memory Used: 49376 bytes
HIREDIS GET 1000000 set of Data Total Time: 18.585435 sec
HIREDIS GET Per Data Time: 0.000019 sec
```

此為 myredis 和 hiredis 的比較。

”= = ” 的上面為 myredis，下面為 hiredis。

可以看到在 Get 資料方面 hiredis 是快上超級多，這也是我程式可以改進的地方。