Nosql資料庫開發 - HW6

學生:潘兆新
系級:資電二
學號:111504506

此次作業實現的功能如下:

1. HashTable指令
2. expire指令
3. 事件驅動機制
4. 自動縮放/擴增的HashTable

HashTable指令

先前於HW3的時經過架構整改，將原本的hashtable.h / hashtable.c重構為dict.h / dict.c，撰寫更多更完善的API，並且使用DBobj型態統一管理變數的創建與釋放，使的開發其他類型的資料結構更加容易，使hset, hget, hdel透過呼叫dict的API輕鬆達成HashTable裡面內嵌一個HashtTable。

```c
// only availible on `dict`
int hset(DB *db, const char * const key, const char * const field, const char * const val); // key
char* hget(DB *db, const char * const key, const char * const field);
int hdel(DB *db, const char * const key, const char * const field);
int hinfo(DB *db, const char * const key);
```

hset key field value

```c
int hset(DB *db, const char * const key, const char * const field, const char * const val){
    if(db == NULL || key == NULL || field == NULL || val == NULL)
        return FAIL;

    // create a dict
    DictEntry *de = dict_set_key(db->dict, key); // it's novalue-type
    // 原生 string
    if(de->type == TYPE_NOVALUE){ // create new dict
        DBobj *obj_dict = DBobj_create(TYPE_DICT);
        // set key - `field`
        DictEntry *de_ = dict_set_key((Dict *)DBobj_get_val(obj_dict), field);
        // set value
        dictEntry_set_string(de_, val); // TODO: state
        // attach new dict to db
        return dictEntry_set_val(de, obj_dict);
    }else if(de->type == TYPE_DICT){// dict has existed. `de->v.val` is dict
        // set key
        DictEntry *de_ = dict_set_key((Dict *)DBobj_get_val(de->v.val), field);
        // set value
        return dictEntry_set_string(de_, val);
    }else
        return FAIL;
}
```

首先使用dict_set_key嘗試創建一個新的key-value pair，如果創建成功物件型態為TYPE_NOVALUE，此時需要使用DBobj_create來創建新的dict物件，將其新增到資料庫的dict當中。

如果使用dict_set_key的時候key已經存在，則會回傳已經存在的dictListEntry，de->v.val儲存DBobj的物件，使用Dbobj_get_val取出原始的資料結構dict進行後續的操作。

成功創建新的dict或是取得已經存在的dict之後，使用dict_set_key以及dict_set_string向HashTable內新增一個key-value pair。

hget key field

```
char* hget(DB *db, const char * const key, const char * const field){
    if(db == NULL || key == NULL || field == NULL)
        return NULL;

    DictEntry *de = dict_get(db->dict, key);
    if(de == NULL)
        return NULL;
    if(de->type == TYPE_DICT){
        DictEntry *de_ = dict_get((Dict *)DBobj_get_val(de->v.val), field);
        assert(de_->type == TYPE_STRING);
        return de_->v.val; // TODO: 統一string儲存
    }else
        return NULL;
}
```

如果只需要取出HashTable某一個field的值，則只需要連續使用兩次dict_get來獲得value。

第一次使用dict_get會獲得HashTable，第二次使用dict_get會得到這個field對應的value。

hdel key field

```
int hdel(DB *db, const char * const key, const char * const field){
    if(db == NULL || key == NULL || field == NULL)
        return FAIL;

    DictEntry *de = dict_get(db->dict, key); // it's novalue-type
    if(de->type == TYPE_DICT)
        return dict_del((Dict *)DBobj_get_val(de->v.val), field);
    else
        return FAIL;
}
```

如果需要刪除HashTable當中某一個field-value pair，先使用dict_get取得HashTable，再使用dict_del刪除HashTable當中的元素。

expire指令

expire key time (sec)

為了使用事件驅動的方法，引入libev撰寫程式。

expire是資料庫的API，expire_cb則是libev event watcher的callback。

```
int expire(DB *db, const char * const key, uint64_t time);
static void expire_cb(struct ev_loop *loop, ev_timer *w, int revent);
```

```
int expire(DB *db, const char * const key, uint64_t time){
    // expire time (sec)
    if(db == NULL || key == NULL || time <= 0)
        return FAIL;

    if(db->key_time == NULL)
        db->key_time = dict_create(BUCKET_SIZE_LIST[0]);
    // set key
    DictEntry *de = dict_set_key(db->key_time, key); // set unsigned int
    // set value to novalue-type
    if(de->type == TYPE_NOVALUE || de->type == TYPE_UNSIGNED_INT64){
        dictEntry_set_unsignedint(de, time);
        // set timer
        // create watcher
        ev_expire *timer = (ev_expire*)malloc(sizeof(ev_expire));
        timer->db = db;
        // timer->key = key; // Is it OK ?
        timer->key = (char*)malloc(sizeof(char)*64); // 64
        strncpy(timer->key, key, 64); // Is it OK ?
        ev_timer_init(&timer->w, expire_cb, time, 0.); // 0: no repeat
        ev_timer_start(loop, &timer->w);
        return SUCCESS;
    }else
        return FAIL;
}
```

```c
static void expire_cb(struct ev_loop *loop, ev_timer *w, int revent){
    // delete key, value pair from db->dict
    // delete that key from db->key->time
    ev_expire *timer = (ev_expire*)w;
    dict_del(timer->db->dict, timer->key); // expired key, value pair
    // null key ?
    dict_del(timer->db->key_time, timer->key);
    free(timer->key); // BAD ?
    free(timer);
}
```

expire流程如下：

main主程式啟動eventloop, 使用io_watcher監聽stdin,

透過stdin read input, 解析輸入的參數, 如果符合`expire key time`, 則呼叫db.c當中的expire函式,

創建並儲存libev的ev_timer, 設定時間觸發callback,
callback函式必須執行以下邏輯：

需要額外取得參數, 刪除資料庫當中指定的key-value pair,

釋放ev_timer。

expire函式邏輯如下：

存取資料庫的dict, db->key_time, 如果不存在則創建一個dict,

使用dict_set_key嘗試新增key, value為TYPE_NOVALUE,

如果已經存在或是型態錯誤則返回失敗, 反之將value設定為過期秒數,

創建一個自定義的struct ev_expire, 結構當中我們定義一個ev_timer的成員, 剩餘的欄位放置參數。(來自官方解法: c - libev pass argument to callback - Stack Overflow)

```c
typedef struct ev_expire{
    struct ev_timer w;
    DB *db;
    char *key; // TODO:
} ev_expire;
```

初始化並啟動timer, 等待觸發timer的callback expire_cb,

expire_cb 程式邏輯：

使用以下方法將ev_timer轉換成我們自定義的struct，以從中獲取參數，
ev_expire *timer = (ev_expire*)w;

刪除資料庫當中指定的key-value pair，

釋放timer。

事件驅動機制

```
int main(){

    db = DB_create(BUCKET_SIZE_LIST[0]);

    hello();
    printf("db > ");  // db > 提示符
    fflush(stdout);

    initialize_loop();
    ev_io stdin_watcher;

    ev_io_init(&stdin_watcher, stdin_cb, STDIN_FILENO, EV_READ);
    ev_io_start(loop, &stdin_watcher);

    // ev_idle_start(loop, &(db->dict->watcher->w)); // In the begin

    ev_run(loop, 0);
}
```

```c
static void stdin_cb(struct ev_loop *loop, ev_io *w, int revents) {
    char buffer[BUFFER_SIZE];
    int argv = 0;
    char args[N_MAX_TOKEN][BUFFER_SIZE];
    ssize_t nread = read(STDIN_FILENO, buffer, sizeof(buffer) - 1);
    if (nread > 0) {
        buffer[nread] = '\0'; // the last character
        // printf("Received input: %s", buffer);

        if (buffer[nread - 1] == '\n') {
            buffer[nread - 1] = '\0';
        }

        if (strncmp(buffer, "exit", 4) == 0) {
            ev_break(EV_A_ EVBREAK_ALL);  // 終止事件循環
            printf("Bye.\n");
            fflush(stdout);
            DB_free(db);
            return;
        }

        char *token = strtok(buffer, " ");
        while (token != NULL) {
            // printf("Token: %s\n", token);
```

使用libev的io_watcher來監聽標準輸入，流程如下：

初始化資料庫，

初始化eventloop，

註冊並且啟動io watcher，

執行eventloop。

當io watcher監聽有標準輸入的時候，觸發stdin_cb，這時候可以從緩衝區讀入input string，解析命令以及參數，再執行相對應的函式。

自動縮放/擴增的**HashTable**

最有挑戰性的功能之一。

為了讓所有的HashTable實現自動縮放，參考redis的dict實現，我將架構調整如下：

```c
typedef struct Dict{
    DictEntryList **bucket[2];
    size_t n_bucket[2];
    size_t n_entry[2];
    // 用default size去初始化
    int used; // default 0 ; otherwise 1
    int migrating; // default 0
    int migrate_idx; // default 0
    struct ev_migrate *watcher; // store callback
} Dict;
```

```c
typedef struct ev_migrate{
    ev_idle w;
    Dict *dict;
} ev_migrate;
```

每一個dict都具有兩個bucket,

used紀錄當前使用的bucket,

migrating紀錄當前是否正在遷移,

migrate_idx紀錄當前遷移到的bucket index,

ev_migrate watcher是一個自定義的struct, 第一個欄位是ev_idle, 其他欄位則存放callback function需要的參數。

因為操作dict的時候需要判斷當前正在使用的bucket以及是否正在進行migration, 所以許多dict的API實現都需要修改:

Dict* dict_create(size_t bucket_size)
    創建dict的時候分配兩個不同大小的bucket,
    初始化watcher, idle的時候觸發callback。
int dict_free(Dict *dict)
    需要釋放兩個bucket
DictEntry* dict_set_key(Dict *dict, const char * const key): 向dict新增key
    如果正在進行migration, 將key新增到migration之後的bucket,
    如果目前沒有進行migration, 將key新增到現在使用的bucket (dict->used)。
DictEntry *dict_get(Dict *dict, const char * const key)
    如果正在進行migration, 需要尋找兩個bucket
    如果目前沒有進行migration, 只需要尋找目前正在使用的bucket
int dict_del(Dict *dict, const char * const key)
    如果正在進行migration, 需要尋找兩個bucket
    如果目前沒有進行migration, 只需要尋找目前正在使用的bucket

migration callback function邏輯:

因為程式過長不便貼上, 請至dict.c當中查看。

void migrate_cb(struct ev_loop *loop, ev_idle *w, int revent)

使用watcher取得參數dict,

如果當前沒有進行migration (dict->migrating == 0), 計算當前正在使用的bucket load factor大小。

如果大於LF_MAX就進行migration（set dict->migrating = 1), 檢查另外一個bucket的大小是否大於原本bucket, 如果沒有則重新分配一個較大的bucket, 結束callback。

如果小於LF_MIN就進行migration（set dict->migrating = 1), 檢查另外一個bucket的大小是否小於原本bucket, 如果沒有則重新分配一個較小的bucket, 結束callback。

提早結束callback, 是為了避免當其他優先級更高的事件需要處理的時候把pocess卡住。

如果當前正在進行migration (dict->migrating == 1), 則從當前的migrate_idx開始往下搜尋直到有非空的dictEntryList (當collision發生的時候, chain在同一個bucket index上的鏈結串列), 只取出一個dictEntry, 將他移動到另外一個bucket。

一次只移動一個dictEntry, 是為了避免當其他優先級更高的事件需要處理的時候把pocess卡住。

現在將LF_MAX設為0.2, LF_MIN設為0.1以進行演示:

```
Bye.
chrispan@LAPTOP-CITVRO3A:/mnt/c/Users/User/Desktop/data structure/hw3$ ./main
**************************************
        Author: chris-pan
        Version: 1.0
        A simple Nosql db
**************************************
db > set k1 v1
Set (k1, v1) successfully.
db > set k2 v2
Set (k2, v2) successfully.
db > set k3 v3
Set (k3, v3) successfully.
db > set k4 v4
Set (k4, v4) successfully.
db > set k5 v5
Set (k5, v5) successfully.
db > set k6 v6
Set (k6, v6) successfully.
db > set k7 v7
Set (k7, v7) successfully.
db > set k8 v8
Set (k8, v8) successfully.
db > set k9 v9
Set (k9, v9) successfully.
db > set k10 v10
Set (k10, v10) successfully.
db > info
Database state:
Used table: 0
In migration: 0
table 0 n_bucket: 53
table 0 n_entry: 10
table 1 n_bucket: 97
table 1 n_entry: 0
db >
```

此時已經加入10個元素，還沒有超過LF_MAX，嘗試再加入一個元素。

```
db > info
Database state:
Used table: 0
In migration: 0
table 0 n_bucket: 53
table 0 n_entry: 10
table 1 n_bucket: 97
table 1 n_entry: 0
db > set k11 v11
Set (k11, v11) successfully.
db > info
Database state:
Used table: 1
In migration: 0
table 0 n_bucket: 53
table 0 n_entry: 0
table 1 n_bucket: 97
table 1 n_entry: 11
db >
```

可以看到有11個元素的時候，超過了LF_MAX，此時觸發migration，將element遷移到另外一個bucket。

```
db > info
Database state:
Used table: 1
In migration: 0
table 0 n_bucket: 53
table 0 n_entry: 0
table 1 n_bucket: 97
table 1 n_entry: 11
db > del k11
Delete Key `k11` successfully.
db > info
Database state:
Used table: 1
In migration: 0
table 0 n_bucket: 53
table 0 n_entry: 0
table 1 n_bucket: 97
table 1 n_entry: 10
db > del k10
Delete Key `k10` successfully.
db > info
Database state:
Used table: 0
In migration: 0
table 0 n_bucket: 53
table 0 n_entry: 9
table 1 n_bucket: 97
table 1 n_entry: 0
db >
```

此時再將兩個元素刪除，小於LF_MIN，migration啟動，將元素遷移到較小的bucket，實現了自動縮放HashTable的操作。