glib 中 hash table

Glib 是一个 C 语言编写的库，它本身是 Gnome 的一个部分，后来 Glib 剥离出来，它为 Gnome 提供了一些操作字符串和常用数据结构的工具函数。这些好的工具函数既然可以提供给 gnome，使用，自然也可以提供给我们使用。（靠，这逻辑，怎么这么像 和尚摸的，我自然也摸的，晕死啊）。最近看到我们老大用了 Glib 的 hash 表，在工期紧急的情况下解决了一个功能扩展的问题，所以我也就动了玩玩 Glib 的心思。

Glib 是个好东西，它提供了好多常用的数据结构：双向链表，单链表，hashtable ，平衡二叉树，N 叉树等等。如果你花点时间熟悉开放出来的 API，可以直接用在你自己的程序中，减少自己写这些基础数据结构的 effort。当然了这个拿来主义的理念和我自己的理念不太一致，我不喜欢黑盒子的东西，如果我奉行拿来主义的话中，那我会选择用 python。不喜欢归不喜欢，也不影响我爱心大泛滥学下 glib。

我们看下 Glib 的 hash 表怎么使用，首先给个简单的例子。

```
#include <stdio.h>
#include <string.h>
#include <glib.h>
#include <time.h>
#include <assert.h>


#define TIMES 20
static void free_data(gpointer hash_data)
{
    g_free(hash_data);
    hash_data = NULL;
}

void print_key_value(gpointer key, gpointer value ,gpointer user_data)
{
    printf("%s ------------------->%s\n",(char*)key,(char*)value);
}

int hash_test_1()
{
    GHashTable* name_score = NULL;
    int ret = 0;
    name_score = g_hash_table_new(g_str_hash,g_str_equal);
    if(name_score == NULL)
    {
        fprintf(stderr, "create hash table failed\n");
        return -1;
    }

    g_hash_table_insert(name_score,"Bean","77");
    g_hash_table_insert(name_score,"Ted","79");
    g_hash_table_insert(name_score,"Lucy","87");
```

```
    g_hash_table_insert(name_score,"Jim","90");
    g_hash_table_insert(name_score,"Candy","84");

    g_hash_table_foreach(name_score,print_key_value,NULL);
    char* Bean_score = g_hash_table_lookup(name_score,"Bean");
    if(Bean_score == NULL)
    {
        fprintf(stderr,"can not found Bean Score\n");
        ret = -2;
        goto exit;
    }
    printf("Bean\'s score = %s\n",(char*)Bean_score);

    printf("modify Bean\' Score to 86\n");
    g_hash_table_replace(name_score,"Bean","86");

     Bean_score = g_hash_table_lookup(name_score,"Bean");
    if(Bean_score == NULL)
    {
        fprintf(stderr,"can not found Bean Score after modify\n");
        ret = -2;
        goto exit;

    }

    printf("Bean\'s score = %s\n",Bean_score);
exit:
     g_hash_table_destroy(name_score);

    return ret;

}

int main()
{
     hash_test_1();
    // hash_test_2();
}
```

这个例子是一个最简单的版本了，原因在于 key 和 value 都不是动态分配（malloc）出来的，不需要释放，所以我们不需要传递释放 key 和释放 value 的函数。

这个版本太简单，他不符合实战的需求，我们搞一个稍复杂点的版本：

```
#include <stdio.h>
#include <string.h>
#include <glib.h>
#include <time.h>
#include <assert.h>
```

```c
#define TIMES 20
static void free_data(gpointer hash_data)
{
    g_free(hash_data);
    hash_data = NULL;
}


void print_key_value(gpointer key, gpointer value ,gpointer user_data)
{
    printf("%4s ---------->%s\n",(char*)key,(char*)value);
}

int hash_test_2()
{
     GHashTable* dictionary = NULL;
     dictionary = g_hash_table_new_full(g_str_hash,g_str_equal,free_data,free_data);

     if(dictionary == NULL)
     {
         fprintf(stderr,"create hash table failed\n");
         return -1;
     }

     srandom(time(NULL));

     int i = 0;
     int ret = 0;
     char key[64] ;
     char value[64] ;
     for(i = 0; i< TIMES;i++)
     {
         snprintf(key,sizeof(key),"%d",i);
         snprintf(value,sizeof(value), "%d", random());

         char* key_in_hash = strdup(key);
         char* value_in_hash = strdup(value);

         if( value_in_hash == NULL || key_in_hash == NULL)
         {
             ret = -2;
             fprintf(stderr,"key or value malloc failed\n");
             goto exit;
         }

         if(strcmp(key_in_hash,"10") == 0)
```

```
        {
            printf("before insert key(10) address(%p) : value(%s)
address(%p)\n",key_in_hash,value_in_hash,value_in_hash);
        }
        g_hash_table_insert(dictionary, key_in_hash,value_in_hash);

    }

    g_hash_table_foreach(dictionary,print_key_value,NULL);
    printf("there are %d records in dictory\n",(unsigned int) g_hash_table_size(dictionary));

    char* key_10 = NULL;
    char* value_10 = NULL;
    ret = g_hash_table_lookup_extended(dictionary,"10",(void **)&key_10, (void **)&value_10);
    if(ret==FALSE)
    {
        fprintf(stderr, "can not the key 10\n");
        goto exit;
    }
    else
    {
        fprintf(stderr,"In dictionary, key(%s) address(%p) : value (%s)
address(%p)\n",key_10,key_10,value_10,value_10);
    }

    char* key_10_new = strdup("10");
    char* value_10_new = strdup("new 10 value");

    g_hash_table_replace(dictionary,key_10_new,value_10_new);


    ret = g_hash_table_lookup_extended(dictionary,"10",(void **)&key_10,(void**)&value_10);
    if(ret == FALSE)
    {
        fprintf(stderr, "found failed after modify\n");

    }
    else
        printf("After replace In dictionary, key(%s) address(%p) : value (%s)
address(%p)\n",key_10,key_10,value_10,value_10);

    g_hash_table_remove(dictionary,"10");
    value_10 = g_hash_table_lookup(dictionary,"10");
    assert(value_10 == NULL);
```

```
    ret = 0;
exit:
    g_hash_table_destroy(dictionary);
    return ret;
}

int main()
{
     hash_test_2();
}
```

编译：

```
gcc -o use ghash_use.c `pkg-config --cflags --libs glib-2.0 `
```

输出：

```
root@manu:~/code/c/self/ghash# ./use
before insert key(10) address(0x82b29c8) : value(890994488) address(0x82b29d8)
  9 ---------->518218022
 10 ---------->890994488
 11 ---------->1367601760
 12 ---------->154022116
 13 ---------->1596002810
 14 ---------->297169373
 15 ---------->309362452
 16 ---------->2009687739
 17 ---------->899619098
 18 ---------->1938444585
 19 ---------->858182021
  0 ---------->737059251
  1 ---------->809558677
  2 ---------->1707784285
  3 ---------->2000110468
  4 ---------->155424423
  5 ---------->671731059
  6 ---------->1157942798
  7 ---------->1512213641
  8 ---------->488939051
there are 20 records in dictory
In dictionary, key(10) address(0x82b29c8) : value (890994488) address(0x82b29d8)
After replace In dictionary, key(10) address(0x82b2b08) : value (new 10 value) address(0x82b2b18)
```

我们的例子用到了如下的 API：

1 创建 hash table

```
name_score = g_hash_table_new(g_str_hash,g_str_equal)
```

我们的 hash table 叫做 name_score，使用了 g_hash_table_new 这个 API 去创建，这个 API 本质是：

```
GHashTable *
g_hash_table_new (GHashFunc hash_func,
```

```
                GEqualFunc key_equal_func)
{
  return g_hash_table_new_full (hash_func, key_equal_func, NULL, NULL);
}

GHashTable *
g_hash_table_new_full (GHashFunc hash_func,
                       GEqualFunc key_equal_func,
                       GDestroyNotify key_destroy_func,
                       GDestroyNotify value_destroy_func)
```

我们看到了,g_hash_table_new 是 g_hash_table_new_full 的弱化版，弱化在 key_destroy_func 和 value_destroy_func 都是 NULL。对于我们第二个例子，key value 的值都是我们 malloc 出来的（strdup），为了防止内存泄漏，我们销毁 key-value 对的时候，必须释放这些空间。如何释放？创建 hashtable 的时候指定。 从两个参数的含义当需要的时候，释放 key 需要对 key 对应的地址调用 key_destroy_func，value 亦然。

什么是需要的时候呢？

最直观的就是将这个 hashtable 销毁的时候， 也就是我们调用 g_hash_table_destroy 的时候，hash table 会销毁插入到 hashtable 的每一个 key-value 对

再其次就是删除 key-value 对。我们调用 g_hash_table_remove()的时候。会根据 key 找到对应 key-valude，根据创建时有无对应销毁函数分别销毁之

最难想到的是 replace。 首先看下 replace 的 API

```
void
g_hash_table_replace (GHashTable *hash_table,
                      gpointer key,
                      gpointer value)
{
  g_hash_table_insert_internal (hash_table, key, value, TRUE);
}
```

这个 API 里面要求用户提供的新的 key 和 value，注意此处，replace 和 insert 的本质几乎是一样的，如果你创建 hash table 的时候，指定了 key_destroy_func 和 value_destroy_func，这两个函数必须等够施加在你提供的 key value 上。何意？

```
  char* key_10_new = strdup("10");
  char* value_10_new = strdup("new 10 value");
```

```
g_hash_table_replace(dictionary,key_10_new,value_10_new);
```

这是正确的用法，因为 hashtable 创建的时候制定了 free_data 作为 key 和 value 的销毁函数。如下是错误的用法，我学习 glib hash 之初，没少犯错误：

```
static void free_data(gpointer hash_data)
{
    g_free(hash_data);
    hash_data = NULL;
}

dictionary = g_hash_table_new_full(g_str_hash,g_str_equal,free_data,free_data);
g_hash_table_replace(dictionary,"10","new 10 value");
```

原因就在于，等 destroy 的时候会将 free_data 函数施加在"10" 和 “new 10 value”，两个常量字符串上，从而引发错误。

g_hash_table_replace 拿着 key 查找有没有对应的 key-value 对。如果找不到，就赤裸裸的变成了 insert 操作了，如果找的到 old 的 key-value 对，那么就需要释放了。

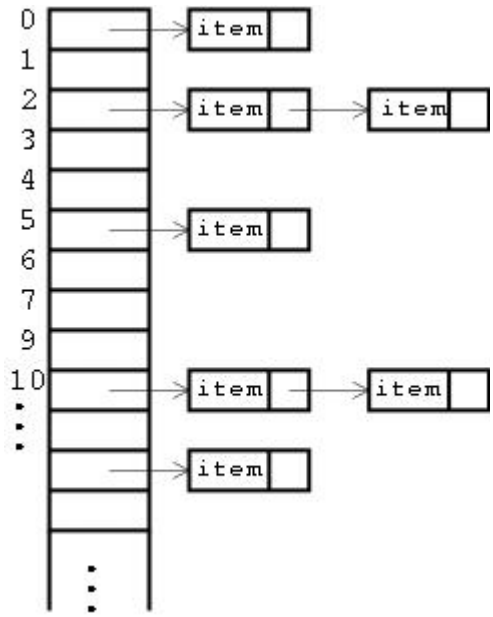
2 insert 和 replace

因为我在学习 glib hashtable replace 的时候，写的代码总是段错误，一怒之下，我就看了 glib 的 source code。发现 glib 的实现挺有意思的。

```
void
g_hash_table_insert (GHashTable *hash_table,
                     gpointer key,
                     gpointer value)
{
  g_hash_table_insert_internal (hash_table, key, value, FALSE);
}

void
g_hash_table_replace (GHashTable *hash_table,
                      gpointer key,
                      gpointer value)
{
  g_hash_table_insert_internal (hash_table, key, value, TRUE);
}
```

我们到了这里，就不得不说下 Glib 对 hash table 的实现 了。我们常见的 hash table 的实现，基本是 bucket list + 链表，何意？



如上图所示，前面是一排桶，对一一个 key-value 对，通过 key 使用 hash function 计算桶号，放入合适的桶中。如果桶中已经有了相同的 hash 值，这叫冲突。冲突的解决办法是链入链表。lookup 的时候，首先根据 key 计算出桶号，依次遍历桶后面挂在每个 key- value，知道找到对应的 key 为止。这个方法通俗易懂，我见过很多 hash table 都是这么写的，你要是让我写 hash table ，我也这么写。这么写好不好，当然很好。但是也有不好的地方。链表是缓存杀手。一次命中也就罢了，如果命不中，链表 next 的内存位置几乎肯定用不上 cache。之前我写 queue，stack 用链表的时候，已经有网友指出这一点。

glib 是如何实现的呢？ glib 用的是数组来实现的。数组的好处不多说了，内存连续从而增大了缓存命中的概率。严格意义上讲，glib 的 hash 是由三个数组：

```
struct _GHashTable{
    ....
    gpointer *keys;
    guint *hashes;
    gpointer *values;
    ....
}
```

看下创建过程 g_hash_table_new_full，如何初始化这三个数组：

```
hash_table->keys = g_new0 (gpointer, hash_table->size);

hash_table->values = hash_table->keys;

hash_table->hashes = g_new0 (guint, hash_table->size)
```

有些人看到这里可能迷惑了，明明是两个数组啊，keys 和 hash 都开辟了空间，可是 values 没有开辟复用了 key 的数组。严格意义上将，对于 hash table 来讲，是三个数组，此处初始化两个数组，key 和 value 复用一个的原因，是为了照顾 set 集合。集合的概念和 hash table 是很像的，只不过 hash table 是 key-value 对，set 的概念只有 key，将一个元素插入集合，在集合中查找某个元素，这么一想，set 和 hash table 本质是一样的，set 不过是弱化版的 hash table。对于 set 来说，只有 key 没有 value，所以，value 一开始是指向 key 的数组的。当然，这种兼顾 hash table 和 set 给我们带来的一定的困惑。不过没关系，记住 hash table 本质是有三个数组就好了。真正的 value 数组的开辟是在 g_hash_table_insert_node 里面实现的：

```
if (G_UNLIKELY (hash_table->keys == hash_table->values && key != value))
    hash_table->values = g_memdup (hash_table->keys, sizeof (gpointer) * hash_table->size);
```

另外我看过 glib-2.24.0 的代码，那时候还是创建 hashtable 的时候，直接分配三个数组。glib-2.34.0 为了照顾 set，已经改成现在这个样子。

接下来我们将描述如何利用数组，做成 hash table 的。这就不得不讲整个 hash table 中最重要的两个 function，说最重要，绝非虚言，绝不是考前老师划重点 ，到处都是重点的行径

1 g_hash_table_lookup_node

```
static inline guint
g_hash_table_lookup_node (GHashTable *hash_table,
                          gconstpointer key,
                          guint *hash_return)
{
  guint node_index;
  guint node_hash;
  guint hash_value;
  guint first_tombstone = 0;
  gboolean have_tombstone = FALSE;
  guint step = 0;

  hash_value = hash_table->hash_func (key);
  if (G_UNLIKELY (!HASH_IS_REAL (hash_value)))
    hash_value = 2;

  *hash_return = hash_value;

  node_index = hash_value % hash_table->mod;
  node_hash = hash_table->hashes[node_index];
```

```
  while (!HASH_IS_UNUSED (node_hash))
    {
      /* We first check if our full hash values
       * are equal so we can avoid calling the full-blown
       * key equality function in most cases.
       */
      if (node_hash == hash_value)
        {
          gpointer node_key = hash_table->keys[node_index];

          if (hash_table->key_equal_func)
            {
              if (hash_table->key_equal_func (node_key, key))
                return node_index;
            }
          else if (node_key == key)
            {
              return node_index;
            }
        }
      else if (HASH_IS_TOMBSTONE (node_hash) && !have_tombstone)
        {
          first_tombstone = node_index;
          have_tombstone = TRUE;
        }

      step++;
      node_index += step;
      node_index &= hash_table->mask;
      node_hash = hash_table->hashes[node_index];
    }

  if (have_tombstone)
    return first_tombstone;

  return node_index;
}
```

这个函数是如此的重要，以至于我不得不无耻的把整个函数都搬出来了。上来就是闷头一棒，what is the fuck HASH_IS_REAL/HASH_IS_UNUSED/HASH_IS_TOMBSTONE ?

原谅我的粗俗，看这个简短函数的时候我的直观感受就是这个。

```
#define UNUSED_HASH_VALUE 0
```

```
#define TOMBSTONE_HASH_VALUE 1
#define HASH_IS_UNUSED(h_) ((h_) == UNUSED_HASH_VALUE)
#define HASH_IS_TOMBSTONE(h_) ((h_) == TOMBSTONE_HASH_VALUE)
#define HASH_IS_REAL(h_) ((h_) >= 2)
```

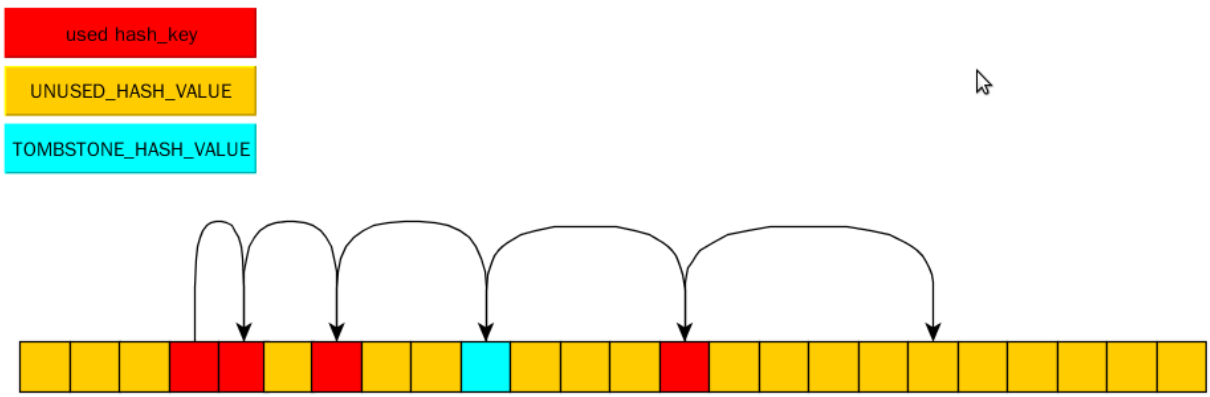
其实和前面 hash table 桶的概念是一样的，只不过 0 号和 1 号被特殊处理了。如果你的 key 通过 hash 函数散列以后，发现你的桶号是 0 或者是 1,那么你的桶号强制改成 2 号。那 0 号和 1 号干啥用呢？ 因为前面提到的数组有个数组叫做 hashes，它记录的是已经存在在 hash table 中所有的 key 经过散列之后的值（hash_key）。有两种情况是特殊的，

1 数组的此位置从来没有被插入过，那么 hashes 这个数组的此位置 存储的是 0，UNUSED_HASH_VALUE

2 曾将存放过某个 hash_key，但是被删除了，OK，记录成 TOMBSTONE_HASH_VALUE。

UNUSED_HASH_VALUE，告诉我们的，此处是天涯海角，是人类活动的极限，确切的说，是如果冲突了，和你有相同 hash_key 的那些 key 的活动的极限，从来没有和你冲突的 key 可到过此处，如果你找到了此处，依然没有找到你要找的 key，就没有必要继续找下去了。

TOMBSTONE_HASH_VALUE，告诉我们，曾经有个和你冲突的 key（你们两个具有相同的 hash_key）到达过此处，但是，后来被移除了。这表示



如上图所示，hashtable 的数组 hashes 里面的记录分三种，如上图三种颜色，

第一种是 UNUSED_HASH_VALUE.也就是里面的值是 0.表示此位置可用，我们可以将 key 存放到

key 数组的此位置，value 数组也是同理。可以想见，刚初始化的的 hash table 全部是这个颜色的，统统可用。插入效率很高，直接插入对应位置即可。

第二中颜色是红色，表示冲突了，已经有一个 key 占用了此位置，对不起，请查找其他位置。查找规则是

```
step++;

node_index += step
```

如上图所示，直到遇到第一个 UNUSED_HASH_VALUE，就不要再浪费时间继续差找了。为何？

注意，key 通过 hash function 之后得到 hash_key，如果冲突，表示 hash_key 相同，那么大家查找的起点都是上图第一个红色位置，步进的规则又是相同的，如果后面仍有相同 hash_key，此处必不会为 UNUSED_HASH_VALUE。此处为 UNUSED_HASH_VALUE，表示具有相同 hash_key 的 key-value 足迹从没有到达此处。hash_key 都不一样，key 也必然不一样。所以没有继续查找的必要了。

另外一种颜色表示，曾有和我具有相同 hash_key 的兄弟到达此处，但是斯人已去，空余一个坑位。

代码的含义就比较好懂了，

1 遇到 UNUSED_HASH_VALUE 之前，和每个红色的比较 key 值，如果 key 值相同，不必多说，找到了相同的 key。返回这个位置。

2 遇到 UNUSED_HASH_VALUE 之前，如果遇到了 TOMBSTONE_HASH_VALUE，把遇到的第一个坑位记住

3 遇到了 UNUSED_HASH_VALUE 表示找不到相同的 key，可以返回了。有 TOMBSTONE_HASH_VALUE 的坑位则返回第一个这种坑位，否则返回遇到的第一个 UNUSED_HASH_VALUE 类型坑位。

第二个重要函数就要迫不及待的闪亮登场了：

2 g_hash_table_insert_node

第二个函数虽然重要，但是远不及第一个函数重要，第一个函数真正反映了 hash 的设计思想，如何处理碰撞，是全 hash table 的精华所在。但是这个函数，则承担了一些脏活累活。这个函数没那么重要，我依然把他全部 copy 了下拉。好吧，我本身就这么无耻。

```
static void
g_hash_table_insert_node (GHashTable *hash_table,
                          guint node_index,
                          guint key_hash,
                          gpointer key,
                          gpointer value,
                          gboolean keep_new_key,
                          gboolean reusing_key)
{
  guint old_hash;
  gpointer old_key;
  gpointer old_value;
```

```
  if (G_UNLIKELY (hash_table->keys == hash_table->values && key != value))
    hash_table->values = g_memdup (hash_table->keys, sizeof (gpointer) * hash_table->size);

  old_hash = hash_table->hashes[node_index];
  old_key = hash_table->keys[node_index];
  old_value = hash_table->values[node_index];

  if (HASH_IS_REAL (old_hash)) //找到的红色坑位，不仅仅是 hash_key 相等，而且 key 也相等
    {
      if (keep_new_key)
        hash_table->keys[node_index] = key;
      hash_table->values[node_index] = value;
    }
  else
    {
      hash_table->keys[node_index] = key;
      hash_table->values[node_index] = value;
      hash_table->hashes[node_index] = key_hash;

      hash_table->nnodes++;

      if (HASH_IS_UNUSED (old_hash))
        {
          /* We replaced an empty node, and not a tombstone */
          hash_table->noccupied++;
          g_hash_table_maybe_resize (hash_table);
        }

#ifndef G_DISABLE_ASSERT
      hash_table->version++;
#endif
    }

  if (HASH_IS_REAL (old_hash))
    {
      if (hash_table->key_destroy_func && !reusing_key)
        hash_table->key_destroy_func (keep_new_key ? old_key : key);
      if (hash_table->value_destroy_func)
        hash_table->value_destroy_func (old_value);
    }
}
```

算了，不多说了，理解了第一个函数，再理解这个函数就是摧枯拉朽了。只要记住，hash_key 相等表示冲突，这个坑位是三种颜色的，如前所述 。

我希望大家注意 keep_new_key 这个标志位，对于 g_hash_table_insert，这个标志位传递是

FALSE，对于 g_hash_table_replace，这个标志传递的是 TRUE。两者仅仅在对待 old_key 的态度上不同，对于 insert，仍然使用 old_key 释放新 key，而 replace 则相反，仅此而已。

最后的最后，对 glib hash table 性能感兴趣的，可以去此处 http://incise.org/hash-table-benchmarks.html， 对 API 感兴趣的可以去此处 https://developer.gnome.org/glib/unstable/glib-Hash-Tables.html#g-hash-table-unref，对源码看兴趣的可以去此处 http://ftp.gnome.org/pub/GNOME/sources/glib/，都不感兴趣的，谢谢你能看到此处，这是一个奇迹。

参考文献：

1 glib 是如何实现 hash table 的

2 glib-2.34.0 源码。