# Glib 库简介

glib 库是 Linux 平台下最常用的 C 语言函数库，它具有很好的可移植性和实用性。glib 是 Gtk+库和 Gnome 的基础。glib 可以在多个平台下使用，比如 Linux、Unix、Windows 等。glib 为许多标准的、常用的 C 语言结构提供了相应的替代物。如果有什么东西本书没有介绍到，请参考 glib 的头文件：glib . h。glib . h 中的头文件很容易理解，很多函数从字面上都能猜出它的用处和用法。如果有兴趣， glib 的源代码也是非常好的学习材料。

glib 的各种实用程序具有一致的接口。它的编码风格是半面向对象，标识符加了一个前缀“g”，这也是一种通行的命名约定。使用 glib 库的程序都应该包含 glib 的头文件 glib . h。如果程序已经包含了 gtk.h 或 gnome.h，则不需要再包含 glib . h。

## 类型定义

glib 的类型定义不是使用 C 的标准类型，它自己有一套类型系统。它们比常用的 C 语言的类型更丰富，也更安全可靠。引进这套系统是为了多种原因。例如， gint32 能保证是 32 位的整数，一些不是标准 C 的类型也能保证。有一些仅仅是为了输入方便，比如 guint 比 unsigned 更容易输入。还有一些仅仅是为了保持一致的命名规则，比如， gchar 和 char 是完全一样的。以下是 glib 基本类型定义：

**整数类型**：gint8、guint8、gint16、guint16、gint32、guint32、gint64、guint64。其中：

**gint8** 是 8 位的整数，guint8 是 8 位的无符号整数，其他依此类推。这些整数类型能够保证大小。

不是所有的平台都提供 64 位整型，如果一个平台有这些，glib 会定义 G_HAVE_GINT64。

整数类型 gshort、glong、gint 和 short、long、int 完全等价。

布尔类型 gboolean：它可使代码更易读，因为普通 C 没有布尔类型。Gboolean 可以取两个值：TRUE 和 FALSE。实际上 FALSE 定义为 0，而 TRUE 定义为非零值。

字符型 gchar 和 char 完全一样，只是为了保持一致的命名。

浮点类型 gfloat、gdouble 和 float、double 完全等价。

指针 gpointer 对应于标准 C 的 void *，但是比 void *更方便。

指针 gconstpointer 对应于标准 C 的 const void *（注意，将 const void *定义为 const gpointer 是行不通的）。

```
sizeof gboolean: 4 Bytes     sizeof gchar: 1 Bytes
```

| | |
|---|---|
| sizeof guchar: 1 Bytes | sizeof gint8: 1 Bytes |
| sizeof guint8: 1 Bytes | sizeof gint16: 2 Bytes |
| sizeof guint16: 2 Bytes | sizeof gint32: 4 Bytes |
| sizeof guint32: 4 Bytes | sizeof gint64: 8 Bytes |
| sizeof guint64: 8 Bytes | sizeof gshort: 2 Bytes |
| sizeof gint: 4 Bytes | sizeof glong: 8 Bytes |
| sizeof gfloat: 4 Bytes | sizeof gdouble: 8 Bytes |
| sizeof gpointer: 8 Bytes | sizeof gconstpointer: 8 Bytes |

## glib 的宏

## 常用宏

glib 定义了一些在 C 程序中常见的宏，详见下面的列表。TRUE / FALSE / NULL 就是 1 / 0 / ( ( void* ) 0 )。MIN( ) /MAX( )返回更小或更大的参数。ABS ( )返回绝对值。CLAMP(x, low,high )若 X 在[ low, high]范围内，则等于 X；如果 X 小于 low，则返回 low；如果 X 大于 high，则返回 high。

一些常用的宏列表

#include <glib.h>

TRUE

FALSE

NULL

MAX(a, b)

MIN(a, b)

ABS ( x )

CLAMP(x, low, high)

有些宏只有 glib 拥有，例如在后面要介绍的 gpointer-to-gint 和 gpointer-to-guint。

大多数 glib 的数据结构都设计成存储一个 gpointer。如果想存储指针来动态分配对象，可以这样做。然而，有时还是想存储一列整数而不想动态地分配它们。虽然 C 标准不能严格保证，但是在多数 glib 支持的平台上，在 gpointer 变量中存储 gint 或 guint 仍是可能的。在某些情况下，需要使用中间类型转换。

下面是示例：

```
gint my_int;
gpointer my_pointer;
my_int = 5;
my_pointer = GINT_TO_POINTER(my_int);
printf("We are storing %d\n", GPOINTER_TO_INT(my_pointer));
```

这些宏允许在一个指针中存储一个整数，但在一个整数中存储一个指针是不行的。如果要实现的话，必须在一个长整型中存储指针。

宏列表：在指针中存储整数的宏

#include <glib.h>

**GINT_TO_POINTER ( p )**

**GPOINTER_TO_INT ( p )**

**GUINT_TO_POINTER( p )**
**GPOINTER_TO_UINT( p )**

## 调试宏

glib 提供了一整套宏，在你的代码中使用它们可以强制执行不变式和前置条件。这些宏很稳定， 也容易使用， 因而 Gtk+大量使用它们。定义了 G_DISABLE_CHECKS 或 G_DISABLE_ASSERT 之后，编译时它们就会消失，所以在软件代码中使用它们不会有性能损失。大量使用它们能够更快速地发现程序的错误。发现错误后，为确保错误不会在以后的 版本中出现，可以添加断言和检查。特别是当编写的代码被其他程序员当作黑盒子使用时，这种检查很有用。用户会立刻知道在调用你的代码时发生了什么错误，而 不是猜测你的代码中有什么缺陷。

当然，应该确保代码不是依赖于一些只用于调试的语句才能正常工作。如果一些语句在生成代码时要取消，这些语句不应该有任何副作用。

宏列表：前提条件检查

#include <glib.h>
**g_return_if_fail(condition)**
**g_return_val_if_fail(condition, retval)**

这个宏列表列出了 glib 的预条件检查宏。对 g_return_if_fail( )，如果条件为假，则打印一个警告信息并且从当前函数立刻返回。g_return_val_if_fail( )与前一个宏类似，但是允许返回一个值。毫无疑问，这些宏很有用—如果大量使用它们，特别是结合 Gtk+的实时类型检查，会节省大量的查找指针和类型错误的时间。

使用这些函数很简单，下面的例子是 glib 中哈希表的实现：

```
void g_hash_table_foreach (GHashTable *hash_table,GHFunc func, gpointer
user_data)
{
  GHashNode *node;
  gint i;
  g_return_if_fail (hash_table != NULL);
  g_return_if_fail (func != NULL);
  for (i = 0; i < hash_table->size; i++)
      for (node = hash_table->nodes[i]; node; node = node->next)
          (* func) (node->key, node->value, user_data);
}
```

如果不检查，这个程序把 NULL 作为参数时将导致一个奇怪的错误。库函数的使用者可能要通过调试器找出错误出现在哪里，甚至要到 glib 的源代码中查找代码的错误是什么。使用这种前提条件检查，他们将得到一个很不错的错误信息，告之不允许使用 NULL 参数。

宏列表：断言

#include <glib.h>
**g_assert(condition )**
**g_assert_not_reached( )**

glib 也有更传统的断言函数。g_assert( )基本上与 assert( )一样，但是对 G_DISABLE_ASSERT 响应（如果定义了 G_DISABLE_ASSERT，则这些语句在编译时不编译进去），以及在所有平台上行为都是一致的。还有一个 g_assert_not_reached( )，如果执行到这个语句，它会调用 abort( )退出程序并且(如果环境支持)转储一个可用于调试的 core 文件。

应该断言用来检查函数或库内部的一致性。g_return_if_fail( )确保传递到程序模块的公用接口的值是合法的。也就是说，如果断言失败，将返回一条信息，通常应该在包含断言的模块中查找错误；如果 g_return_if_fail( )检查失败，通常要在调用这个模块的代码中查找错误。这也是断言与前提条件检查的区别。

下面 glib 日历计算模块的代码说明了这种差别：

```
GDate* g_date_new_dmy (GDateDay day, GDateMonth m, GDateYear y)

{

    GDate *d;
    g_return_val_if_fail (g_date_valid_dmy (day, m, y), NULL);
    d = g_new (GDate, 1);
    d->julian = FALSE;
    d->dmy = TRUE;
    d->month = m;
    d->day = day;
    d->year = y;
    g_assert (g_date_valid (d));
    return d;
}
```

开始的预条件检查确保用户传递合理的年月日值；结尾的断言确保 glib 构造一个健全的对象，输出健全的值。

断言函数 g_assert_not_reached() 用来标识“不可能”的情况，通常用来检测不能处理的所有可能枚举值的 switch 语句：

```
switch (val)
    {
    case FOO_ONE:
        break ;
    case FOO_TWO:
        break ;
    default:
```

```
        /*  无效枚举值* /
        g_assert_not_reached( ) ;
        break ;
    }
```

所有调试宏使用 glib 的 g_log ( )输出警告信息，g_log ( )的警告信息包含发生错误的应用程序或库函数名字，并且还可以使用一个替代的警告打印例程。例如，可以将所有警告信息发送到对话框或 log 文件而不是输出到控制台。

# 内存管理

glib 用自己的 g_变体包装了标准的 malloc( )和 free( )，即 g_malloc() 和 g_free( )。它们有以下几个小优点：

g_malloc()总是返回 gpointer，而不是 char *，所以不必转换返回值。

如果低层的 malloc( )失败，g_malloc( )将退出程序，所以不必检查返回值是否是 NULL。

g_malloc() 对于分配 0 字节返回 NULL。

g_free()忽略任何传递给它的 NULL 指针。

除了这些次要的便利， g_malloc() 和 g_free( )支持各种内存调试和剖析。如果将 enable-mem-check 选项传递给 glib 的 configure 脚本，在释放同一个指针两次时， g_free( )将发出警告。

enable-mem-profie 选项使代码使用统计来维护内存。调用 g_mem_profile ( )时，信息会输出到控制台上。最后，还可以定义 USE_DMALLOC，GLIB 内存封装函数会使用 malloc( )。调试宏

在某些平台上在 d malloc. h 中定义。

函数列表： glib 内存分配

```
#include <glib.h>
gpointer g_malloc(gulong size)
void g_free(gpointer mem)
gpointer g_realloc(gpointer mem,gulong size)
gpointer g_memdup(gconstpointer mem,guint bytesize)
```

用 g_free( )和 g_malloc(), malloc()和 free( )，以及(如果正在使用 C++)new 和 delete 匹配是很重要的，否则，由于这些内存分配函数使用不同内存池(new/delete 调用构造函数和解构函数)，不匹配将会发生很糟糕的事。

另外，g_realloc( )和 realloc( )是等价的。还有一个很方便的函数 g_malloc0 ( )，它将分配的内存每一位都设置为 0；另一个函数 g_memdup( )返回一个从 mem 开始的字节数为 bytesize 的拷贝。为了与 g_malloc( )一致， g_realloc( )和 g_malloc0( )都可以分配 0 字节内存。不过，g_memdup( )不能这样做。g_malloc0 ( )在分配的原始内存中填充未设置的位，而不是设置为数值 0。偶尔会有人期望得到初始化为 0.0 的浮点数组，但这样是做不到的。

最后，还有一些指定类型内存分配的宏，见下面的宏列表。这些宏中的每一个 type 参数都是数据类型名，count 参数是指分配字节数。这些宏能节省大量的输入和操纵数据类型的时间，

还可以减少错误。它们会自动转换为目标指针类型，所以试图将分配的内存赋给错误的指 针类型，应该触发一个编译器警告。

宏列表：内存分配宏

```
#include <glib.h>
g_new(type, count)
g_new0(type, count)
g_renew(type, mem, count)
```

# 字符串处理

glib 提供了很丰富的字符串处理函数，其中有一些是 glib 独有的，一些用于解决移植问题。它们都能与 glib 内存分配例程很好地互操作。如果需要比 gchar *更好的字符串，glib 提供了一个 GString 类型。

函数列表： 字符串操作

```
#include <glib.h>
gint g_snprintf(gchar* buf,gulong n,const gchar* format,...)
gint g_strcasecmp(const gchar* s1,const gchar* s2)
gint g_strncasecmp(const gchar* s1,const gchar* s2,guint n)
```

上面的函数列表显示了一些 ANSI C 函数的 glib 替代品，这些函数在 ANSI C 中是扩展函数，一般都已经实现，但不可移植。对普通的 C 函数库，其中的 sprintf( )函数有安全漏洞，容易造成程序崩溃，而相对安全并得到充分实现的 snprintf( )函数一般都是软件供应商的扩展版本。

在含有 snprintf( )的平台上，g_snprintf( )封装了一个本地的 snprintf( )，并且比原有实现更稳定、安全。以往的 snprintf( )不保证它所填充的缓冲是以 NULL 结束的，但 g_snprintf( )保证了这一点。

g_snprintf 函数在 buf 参数中生成一个最大长度为 n 的字符串。其中 format 是格式字符串，后面的“. . .”是要插入的参数。

g_strcasecmp( )和 g_strncasecmp( )实现两个字符串大小写不敏感的比较，后者可指定需比较的最大长度。strcasecmp( )在多个平台上都是可用的，但是有的平台并没有，所以建议使用 glib 的相应函数。

下面的函数列表中的函数在合适的位置上修改字符串：第一个将字符串转换为小写，第二个将字符串全部转换为大写。g_strreverse( )将字符串颠倒过来。g_strchug( )和 g_strchomp ( )，前者去掉字符串前的空格，后者去掉结尾的空格。宏 **g_strstrip**( )结合这两个函数，删除字符串前后的空格。

函数列表： 修改字符串

```
#include <glib.h>
void g_strdown(gchar* string)
```

**void g_strup(gchar* string)**
**void g_strreverse(gchar* string)**
**gchar* g_strchug(gchar* string)**
**gchar* g_strchomp(gchar* string)**

下面的函数列表显示了几个半标准函数的 glib 封装。g_strtod()类似于 strtod ( )，它把字符串 nptr 转换为 gdouble。* endptr 设置为第一个未转换字符，例如，数字后的任何文本。如果转换失败，*endptr 设置为 nptr 值。*endptr 可以是 NULL，这样函数会忽略这个参数。g_strerror( )和 g_strsignal( )与前面没有“ g_”的函数是等价的，但是它们是可移植的，它们返回错误号或警号数的字符串描述。

函数列表： 字符串转换

#include <glib.h>
**gdouble g_strtod(const gchar* nptr,gchar** endptr)**
**gchar* g_strerror(gint errnum)**
**gchar* g_strsignal(gint signum)**

下面的函数列表显示了 glib 中的字符串分配函数。

g_strdup( )和 g_strndup ( )返回一个已分配内存的字符串或字符串前 n 个字符的拷贝。为与 glib 内存分配函数一致，如果向函数中传递一个 NULL 指针，它们返回 NULL。

printf( )返回带格式的字符串。g_strescape 在它的参数前面通过插入另一个“ \”，将后面的字符转义，返回被转义的字符串。g_strnfill( )根据 length 参数返回填充 fill_char 字符的字符串.

g_strdup_printf( )值得特别注意，它是处理下面代码更简单的方法：

```
gchar* str = g_malloc(256);
g_snprintf(str, 256, "%d printf-style %s", 1, "format");
```

用下面的代码，不需计算缓冲区的大小：

```
gchar* str = g_strdup_printf("%d printf-style %s", 1, "format") ;
```

函数列表： 分配字符串

#include <glib.h>
**gchar * g_strdup(const gchar* str)**
**gchar* g_strndup(const gchar* format, guint n)**
**gchar* g_strdup_printf(const gchar* format, ... )**
**gchar* g_strdup_vprintf(const gchar* format,va_list args)**
**gchar* g_strescape(gchar* string)**
**gchar* g_strnfill(guint length,gchar fill_char)**

g_strconcat() 返回由连接每个参数字符串生成的新字符串，最后一个参数必须是 NULL，让 g_strconcat( )知道何时结束。g_strjoin( )与它类似，但是在每个字符串之间插入由 separator 指定的分隔符。如果 separator 是 NULL，则不会插入分隔符。

下面是 glib 提供的连接字符串的函数。

函数列表：连接字符串的函数

```
#include <glib.h>
gchar* g_strconcat(const gchar* string1, ...)
gchar* g_strjoin(const gchar* separator, ...)
```

最后，下面的函数列表总结了几个处理以 NULL 结束的字符串数组的例程。g_strsplit( )在每个分隔符处分割字符串，返回一个新分配的字符串数组。g_strjoinv( )用可选的分隔符连接字符串数组，返回一个已分配好的字符串。g_s trfreev ( )释放数组中每个字符串，然后释放数组本身。

函数列表： 处理以 NULL 结尾的字符串向量

```
#include <glib.h>
gchar** g_strsplit(const gchar* string, const gchar* delimiter, gint max_tokens)
gchar* g_strjoinv(const gchar* separator, gchar** str_array)
void g_strfreev(gchar** str_array)
```

## 数据结构

glib 实现了许多通用数据结构，如单向链表、双向链表、树和哈希表等。下面的内容介绍 glib 链表、排序二叉树、N-ARY 树以及哈希表的实现。

## 链表

glib 提供了普通的单向链表和双向链表，分别是 GSList 和 GList。这些是由 gpointer 链表实现的，可以使用 GINT_TO_POINTER 和 GPOINTER_TO_INT 宏在链表中保存整数。GSlist 和 GList 有一样的 A P I 接口，除了有 g_list_previous() 函数外没有 g_slist_previous( )函数。本节讨论 GSList 的所有函数，这些也适用于双向链表。

在 glib 实现中，空链表只是一个 NULL 指针。因为它是一个长度为 0 的链表，所以向链表函数传递 NULL 总是安全的。以下是创建链表、添加一个元素的代码：

```
GSList* list = NULL;
gchar* element = g_strdup("a string");
list = g_slist_append(list, element);
```

glib 的链表明显受 Lisp 的影响，因此，空链表是一个特殊的“空”值 g_slist_prepend()操作很像一个恒定时间的操作：把新元素添加到链表前面的操作所花的时间都是一样的。

注意，必须将链表用链表修改函数返回的值替换，以防链表头发生变化。Glib 会处理链表的内存问题，根据需要释放和分配链表链接。

例如，以下的代码删除上面添加的元素并清空链表：

```
list = g_slist_remove(list, element);
```

链表 list 现在是 NULL 。当然，仍需自己释放元素。为了清除整个链表，可使用 g_slist_free ( )，它会快速删除所有的链接。因为 g_slist_free( )函数总是将链表置为 NULL，它不会返回值；并且，如果愿意，可以直接为链表赋值。显然, g_slist_free()只释放链表的单元，它并不知道怎样操作链表内容。

为了访问链表的元素，可以直接访问 GSList 结构：

```
gchar* my_data = list->data;
```

为了遍历整个链表，可以如下操作：

```
GSList* tmp = list;
while (tmp != NULL)
{
   printf("List data: %p\n", tmp->data);
   tmp = g_slist_next(tmp);
}
```

下面的列表显示了用于操作 GSList 元素的基本函数。对所有这些函数，必须将函数返回值赋给链表指针，以防链表头发生变化。注意， glib 不存储指向链表尾的指针，所以前插（prepend）操作是一个恒定时间的操作，而追加（append）、插入和删除所需时间与链表大小成正比。

这意味着用 g_slist_append ( )构造一个链表是一个很糟糕的主意。当需要一个特殊顺序的列表项时，可以先调用 g_slist_prepend( )前插数据，然后调用 g_slist_reverse ( )将链表颠倒过来。

如果预计会频繁向链表中追加列表项，也要为最后的元素保留一个指针。下面的代码可以用来有效地向链表中添加数据：

```
void efficient_append(GSList** list, GSList** list_end, gpointer data)
{
    g_return_if_fail(list != NULL);
    g_return_if_fail(list_end != NULL);
    if (*list == NULL){
        g_assert(*list_end == NULL);
        *list = g_slist_append(*list, data);
```

```
            *list_end = *list;
        }else{
           *list_end = g_slist_append(*list_end, data)->next;
        }
    }
```

要使用这个函数，应该在其他地方存储指向链表和链表尾的指针，并将地址传递给 efficient_append( )：

```
GSList* list = NULL;
GSList* list_end = NULL;
efficient_append(&list, &list_end, g_strdup("Foo"));
efficient_append(&list, &list_end, g_strdup("Bar"));
efficient_append(&list, &list_end, g_strdup("Baz"));
```

当然，应该尽量不使用任何改变链表尾但不更新 list_end 的链表函数。

函数列表：改变链表内容

```
#include <glib.h>
/*  向链表最后追加数据，应将修改过的链表赋给链表指针* /
GSList* g_slist_append(GSList* list, gpointer data)
/*  向链表最前面添加数据，应将修改过的链表赋给链表指针* /
GSList* g_slist_prepend(GSList* list, gpointer data)
/*  在链表的 p o s i t i o n 位置向链表插入数据，应将修改过的链表赋给链表指针* /
GSList* g_slist_insert(GSList* list, gpointer data, gint position)
/ *删除链表中的 d a t a 元素，应将修改过的链表赋给链表指针* /
GSList* g_slist_remove(GSList* list, gpointer data)
```

访问链表元素可以使用下面的函数列表中的函数。这些函数都不改变链表的结构。

g_slist_foreach( )对链表的每一项调用 Gfunc 函数。Gfunc 函数是像下面这样定义的：

```
typedef void (*GFunc)(gpointer data, gpointer user_data);
```

在 g_slist_foreach ( )中，Gfunc 函数会对链表的每个 list->data 调用一次，将 user_data 传递到 g_slist_foreach ( )函数中。

例如, 有一个字符串链表，并且想创建一个类似的链表，让每个字符串做一些变换。下面是相应的代码，使用了前面例子中的 efficient_append ( )函数。

```
typedef struct_AppendContext AppendContext;
struct_AppendContext {
    GSList* list;
    GSList* list_end;
```

```
    const gchar* append;
} ;
static void append_foreach(gpointer data, gpointer user_data)
{
    AppendContext* ac = (AppendContext*) user_data;
    gchar* oldstring = (gchar*) data;
    efficient_append(&ac->list, &ac->list_end,
    g_strconcat(oldstring, ac->append, NULL));
}
GSList *copy_with_append(GSList* list_of_strings, const gchar* append)
{
    AppendContext ac;
    ac.list = NULL;
    ac.list_end = NULL;
    ac.append = append;
    g_slist_foreach(list_of_strings, append_foreach, &ac);
    return ac.list;
}
```

函数列表：访问链表中的数据

```
#include <glib.h>
GSList* g_slist_find(GSList* list, gpointer data)
GSList* g_slist_nth(GSList* list, guint n)
gpointer g_slist_nth_data(GSList* list, guint n)
GSList* g_slist_last(GSList* list)
gint g_slist_index(GSList* list, gpointer data)
void g_slist_foreach(GSList* list, GFunc func, gpointer user_data)
```

还有一些很方便的操纵链表的函数，列在下面的函数列表中。除了 g_slist_copy( )函数，所有这些函数都影响相应的链表。也就是，必须将返回值赋给链表或某个变量，就像向链表中添加和删除元素时所做的那样。而 g_slist_copy ( )返回一个新分配的链表，所以能够继续使用两个链表，最后必须将两个链表都释放。

函数列表： 操纵链表

```
#include <glib.h>
/* 返回链表的长度* /
guint g_slist_length(GSList* list)
/* 将 l i s t 1 和 l i s t 2 两个链表连接成一个新链表* /
GSList* g_slist_concat(GSList* list1,GSList* list2)
/ *将链表的元素颠倒次序* /
GSList* g_slist_reverse(GSList* list)
/ *返回链表 l i s t 的一个拷贝* /
GSList* g_slist_copy(GSList* list)
```

最后，还有一些用于对链表排序的函数，见下面的函数列表。要使用这些函数，必须写一个比较函数 GcompareFunc，就像标准 C 里面的 qsort ( )函数一样。在 glib 里面，比较函数是这个样子：

**typedef gint (*GCompareFunc) (gconstpointer a, gconstpointer b);**

如果 a < b，函数应该返回一个负值；如果 a > b，返回一个正值；如果 a = b，返回 0。

一旦有了比较函数，就可以将一个元素插入到一个已经排序的链表中，或者对整个链表

排序。链表是按升序排序的。使用 g_slist_find_custom( ) 函数，甚至能够循环使用 GCompareFunc 来发现链表元素。注意，在 glib 中，GCompareFunc 的使用是不一致的。有时 glib 需要一个等式判定式，而不是一个 qsort( )风格的函数。不过，在链表 API 中，它的用法是一致的。

不要随意对链表排序，滥用它们很快就会变得效率低下。例如， g_slist_insert_sorted ( )函数将数据插入到链表，同时进行排序，它是一个 O(n)复杂度的操作。但是如果在一个循环中插入多个元素，则每次插入都会进行一次排序，循环的运行时间就是指数级的。**较好的方法是先将元素前插，然后调用 g_slist_sort( )函数对链表排序。**

函数列表： 对链表排序

```
#include <glib.h>
GSList* g_slist_insert_sorted(GSList* list, gpointer data, GCompareFunc func)
GSList* g_slist_sort(GSList* list, GCompareFunc func)
GSList* g_slist_find_custom(GSList* list, gpointer data, GCompareFunc func)
```

# 树

树是一种非常重要的数据结构。在 glib 中有两种不同的树： GTree 是基本的平衡二叉树，它将存储按键值排序成对键值； GNode 存储任意的树结构数据，比如分析树或分类树。

## 1. GTree

使用下面的函数创建和销毁一个 Gtree。其中 GCompareFunc 是类似 GSList 的 qsort ( )风格的比较函数。在这里，用它来比较树的键值。

函数列表：创建和销毁平衡二叉树

```
#include <glib.h>
GTree* g_tree_new(GCompareFunc key_compare_func)
void g_tree_destroy(GTree* tree)
```

操作树中数据的函数列在下面的函数列表中。这些函数可以从字面上理解它们的用处和用法。g_tree_insert( )覆盖任何已有值，所以如果存在的值是指向已分配内存区域的唯一指针，使用时要当心。如果 g_tree_lookup( )没有找到所需的键，返回 NULL，否则返回相应的值。键和值都是 gpointer 类型，但是要使用整型，并用 GPOINTER_TO_INT( ) 和 GPOINTER_TO_UINT( )宏进行转换。

函数列表： 操纵 Gtree 数据

```
#include <glib.h>
void g_tree_insert(GTree* tree,gpointer key,gpointer value)
void g_tree_remove(GTree* tree,gpointer key)
gpointer g_tree_lookup(GTree* tree,gpointer key)
```

下面的函数可以确定树的大小。

函数列表： 获得 Gtree 的大小

```
#include <glib.h>
/ *获得树的节点数* /
gint g_tree_nnodes(GTree* tree)
/ *获得树的高度* /
gint g_tree_height(GTree* tree)
```

使 用 g_tree_traverse( )函数可以遍历整棵树。要使用它，需要一个 GtraverseFunc 遍历函数，它用来给 g_tree_traverse( )函数传递每一对键值对和数据参数。只要 GtraverseFunc 返回 FALSE，遍历继续；返回 TRUE 时，遍历停止。可以用 GtraverseFunc 函数按值搜索整棵树。以下是 GtraverseFunc 的定义：

```
typedef gint (*GTraverseFunc)(gpointer key, gpointer value, gpointer data);
```

GtraverseType 是枚举型，它有四种可能的值。下面是它们在 Gtree 中各自的意思：

**G_IN_ORDER** (中序遍历)首先递归左子树节点(通过 GCompareFunc 比较后,较小的键)，然后对当前节点的键值对调用遍历函数，最后递归右子树。这种遍历方法是根据使用 GCompareFunc 函数从最小到最大遍历。
**G_PRE_ORDER** (先序遍历)对当前节点的键值对调用遍历函数，然后递归左子树，最后递归右子树。
**G_POST_ORDER** (后序遍历)先递归左子树，然后递归右子树，最后对当前节点的键值对调用遍历函数。
**G_LEVEL_ORDER** (水平遍历)在 Gtree 中不允许使用，只能用在 GNode 中。

函数列表： 遍历 Gtree

```
#include <glib.h>
```

void **g_tree_traverse**(GTree* tree, GTraverseFunc traverse_func,
GTraverseType traverse_type, gpointer data)

## 2. GNode

一个 GNode 是一棵 N 维的树，由双链表(父和子链表)实现。这样，大多数链表操作函数在 Gnode API 中都有对等的函数。可以用多种方式遍历。以下是一个 GNode 的声明：

```
typedef struct_GNode GNode;
struct_GNode
{
   gpointer data;
   GNode *next;
   GNode *prev;
   GNode *parent;
   GNode *children;
} ;
```

有一些用来访问 GNode 成员的宏，见下面的宏列表。作为一个 Glist，其中的 data 成员可以直接使用。这些宏分别返回 next、prev 和 children 成员，在将 Glist 解除参照以前，这些宏也检查参数是否为 NULL，如果是，则返回 NULL。

宏列表：访问 GNode 成员

```
#include <glib.h>
/ *返回 GNode 的前一个节点* /
g_node_prev_sibling ( node )
/ *返回 GNode 的下一个节点* /
g_node_next_sibling(node)
/ *返回 GNode 的第一个子节点* /
g_node_first_child(node)
```

用 g_node_new ( )函数创建一个新节点。g_node_new ( )创建一个包含数据，并且无子节点、无父节点的 GNode 节点。通常仅用 g_node_new ( )创建根节点，还有一些宏可以根据需要自动创建新节点。

函数列表: 创建一个 GNode

```
#include <glib.h>
GNode* g_node_new(gpointer data)
```

要创建一棵树，可以用下面函数列表中的函数。为方便循环或递归树，每个操作都返回刚刚添加的节点。

函数列表： 创建一棵 GNode 树

```
#include <glib.h>
/ *在父节点 p a r e n t 的 p o s i t i o n 处插入节点 node * /
GNode* g_node_insert(GNode* parent,gint position,GNode* node)
/ *在父节点 p a r e n t 中的 s i b l i n g 节点之前插入节点 node * /
GNode* g_node_insert_before(GNode* parent,GNode* sibling,GNode* node)
/ *在父节点 p a r e n t 最前面插入节点 node * /
GNode* g_node_prepend(GNode* parent,GNode* node)
```

下面的宏列表列出了一些常用的宏，用于实现对 GNode 的操作。g_node_append ( )和 g_node_prepend ( )类似，其余的宏则带一个 data 参数，自动分配节点，并且调用相关的基本操作函数。

宏列表：向 GNode 添加、插入数据

```
#include <glib.h>
g_node_append(parent, node)
g_node_insert_data(parent, position, data)
g_node_insert_data_before(parent, sibling, data)
g_node_prepend_data(parent, data)
g_node_append_data(parent, data)
```

有两个函数可以从一棵树中删除一个节点。g_node_destroy ( )从树中删除一个节点，销毁它以及它的子节点。g_node_unlink ( )将一个节点删除，并将它转换为一个根节点，也就是，它将一棵子树转换为一棵独立的树。

函数列表： 销毁 GNode

```
#include <glib.h>
void g_node_destroy(GNode* root)
void g_node_unlink(GNode* node)
```

下面宏列表中的两个宏用来检查一个节点是否是最顶部的节点或最底部的节点。根节点没有父节点和兄弟节点，叶节点没有子节点。

宏列表：判断 GNode 的类型

```
#include <glib.h>
G_NODE_IS_ROOT( node )
G_NODE_IS_LEAF( node )
```

下面函数列表中的函数返回 GNode 的一些有用信息，包括它的节点数、根节点、深度以及含有特定数据指针的节点。其中的遍历类型 GtraverseType 在 Gtree 中介绍过。下面是在 GNode 中它的可能取值：

**G_IN_ORDER**：先递归节点最左边的子树，并访问节点本身，然后递归节点子树的其他部分。这不是很有用，因为多数情况用于 Gtree 中。
**G_PRE_ORDER**： 访问当前节点，然后递归每一个子树。
**G_POST_ORDER**： 按序递归每个子树，然后访问当前节点。
**G_LEVEL_ORDER**： 首先访问节点本身，然后每个子树，然后子树的子树，然后子树的子树的子树，以次类推。也就是说，它先访问深度为 0 的节点，然后是深度为 1，然后是深度为 2，等等。

GNode 的树遍历函数有一个 GTraverseFlags 参数。这是一个位域，用来改变遍历的种类。当前仅有三个标志—只访问叶节点，非叶节点，或者所有节点：

**G_TRAVERSE_LEAFS**： 指仅遍历叶节点。
**G_TRAVERSE_NON_LEAFS**： 指仅遍历非叶节点。
**G_TRAVERSE_ALL**： 只是指( G_TRAVERSE_LEAFS | G_TRAVERSE_NON_LEAFS)快捷方式。

函数列表： 取得 GNode 属性

#include <glib.h>
**guint g_node_n_nodes(GNode* root, GTraverseFlags flags)**
**GNode* g_node_get_root(GNode* node)**
**Gboolean g_node_is_ancestor(GNode* node,GNode* descendant)**
**Guint g_node_depth(GNode* node)**
**GNode* g_node_find(GNode* root, GTraverseType order, GTraverseFlags flags, gpointer data)**

其他 GNode 函数都很简单，它们大多数是对树的节点表进行操作，见下面的函数列表。

GNode 有两个独有的函数类型定义：

**typedef gboolean (*GNodeTraverseFunc) (GNode* node, gpointer data);**
**typedef void (*GNodeForeachFunc) (GNode* node, gpointer data);**

这些函数调用以要访问的节点指针以及用户数据作为参数。GNode TraverseFunc 返回 TRUE，停止任何正在进行的遍历，这样就能将 GNode TraverseFunc 与 g_node_traverse()结合起来按值搜索树.

函数列表： 访问 GNode

#include <glib.h>

/ *对 GNode 进行遍历* /
**void g_node_traverse(GNode* root, GTraverseType order, GTraverseFlags flags, gint max_depth, GNodeTraverseFunc func, gpointer data)**
/ *返回 GNode 的最大高度* /

**guint g_node_max_height(GNode* root)**
/ *对 GNode 的每个子节点调用一次 f u n c 函数* /
**void g_node_children_foreach(GNode* node, GTraverseFlags flags, GNodeForeachFunc func, gpointer data)**
/ *颠倒 node 的子节点顺序* /
**void g_node_reverse_children(GNode* node)**
/ *返回节点 node 的子节点个数* /
**guint g_node_n_children(GNode* node)**
/ *返回 node 的第 n 个子节点* /
**GNode* g_node_nth_child(GNode* node, guint n)**
/ *返回 node 的最后一个子节点* /
**GNode* g_node_last_child(GNode* node)**
/ *在 node 中查找值为 data 的节点* /
**GNode* g_node_find_child(GNode* node, GTraverseFlags flags, gpointer data)**
/ *返回子节点 c h i l d 在 node 中的位置* /
**gint g_node_child_position(GNode* node, GNode* child)**
/ *返回数据 data 在 node 中的索引号* /
**gint g_node_child_index(GNode* node, gpointer data)**
/ *以子节点形式返回 node 的第一个兄弟节点* /
**GNode* g_node_first_sibling(GNode* node)**
/ *以子节点形式返回 node 的最后一个兄弟节点* /
**GNode* g_node_last_sibling(GNode* node)**

## 哈希表

GHashTable 是一个简单的哈希表实现，提供一个带有连续时间查寻的关联数组。要使用哈希表，必须提供一个 GHashFunc 函数，当向它传递一个哈希值时，会返回正整数：

**typedef guint (*GHashFunc) (gconstpointer key);**

返回的每个 GUint 数值(表字节数的模数)对应于一个哈希表中的一个“存取窗口”或者“哈希表元”。GHashTable 通过在每个“存取窗口”中存储一个键值对的链表来处理冲突。因而，GHashFunc 函数返回的无符号整数值必须在可能取值中尽可能平均地分配，否则哈希表将退化为一个链表。GHashFunc 也必须快，因为每次查找都要用到它。

除了 GHashFunc，还需要一个 GCompareFunc 比较函数用来测试关键字是否相等。不过，虽然 GCompareFunc 函数原型是一样的，但它在 GHashTable 中的用法和在 GSList、Gtree 中的用法不一样。在 GHashTable 中可以将 GCompareFunc 看作是等式操作符，如果参数是相等的，则返回 TRUE。当哈希冲突导致在相同的“哈希表元”中有多个关键字-值对时，键比较函数用来找到正确的键值对。

使用下面函数列表中的函数创建和销毁一个 GHashTable。注意，glib 并不知道怎样销毁哈希表中保存的数据，它只销毁表本身。

函数列表： GHashTable

```
#include <glib.h>
GHashTable* g_hash_table_new(GHashFunc hash_func, GCompareFunc
key_compare_func)
void g_hash_table_destroy(GHashTable* hash_table)
```

哈希表和比较函数支持最常用的几种键：整数、指针和字符串。这些都列在下面的函数列表中。针对整数的函数接收一个指向 gint 类型的指针，而不是 gint 整数值。如果将 NULL 作为哈希函数的参数传递给 g_hash_table_new ( )，缺省情况下会使用 g_direct_hash( )函数。如果给键比较函数传递 NULL 参数，那么会使用简单的指针比较函数(等同于 g_direct_equal( )，但是没有函数调用)。

函数列表： 哈希表/比较函数

```
#include <glib.h>
guint g_int_hash(gconstpointer v)
gint g_int_equal(gconstpointer v1, gconstpointer v2)
guint g_direct_hash(gconstpointer v)
gint g_direct_equal(gconstpointer v1, gconstpointer v2)
guint g_str_hash(gconstpointer v)
gint g_str_equal(gconstpointer v1, gconstpointer v2)
```

操纵哈希表很简单。插入函数不复制键或值，只是将给定的键值准确插入到哈希表，会覆盖任何已存在的具有相同键的键值对(记住，“相同”是由哈希表和比较函 数决定的)。如果这样做有问题，在插入前必须查找或删除哈希表的键或值。如果动态分配键或值，需要特别注意。

如果 g_hash_table_lookup ( )发现了与键相关联的值，返回这个值，否则，返回 NULL。但有时不能这么做。例如， NULL 可能本身就是一个有效的值。如果使用字符串，特别是动态分配字符串作为键，知道表里的一个键或许并不够，或许想检索出哈希表用来代表“ foo”键的确切的 gchar *值。在这种情况下，可以使用 g_hash_table_lookup_extended ( )。如果检索成功，g_hash_table_lookup_extended ( )返回 TRUE；如果返回 TRUE，则将它发现的键－值对放在给定的位置。

函数列表： 处理 GHashTable

```
#include <glib.h>
void g_hash_table_insert(GHashTable* hash_table, gpointer key,
                         gpointer value)
void g_hash_table_remove(GHashTable * hash_table, gconstpointer key)
gpointer g_hash_table_lookup(GHashTable * hash_table, gconstpointer key)
gboolean g_hash_table_lookup_extended(GHashTable* hash_table,
                         gconstpointer lookup_key,
                         gpointer* orig_key, gpointer* value)
```

GHashTable 保存一个内部数组，它的大小是质数。它保存存储在表中的键-值对数的合计。如果每个有用的“哈希表元”中的键值对平均个数降到 0.3 以下，数组会变小；如果在 3 以上，数组会变大以便减少冲突。不论何时从表中插入或删除键值对，数组都会自动调整大小。这确保了哈希表的内存使用是最优化的。然而，如果正在做大量的插入或删除，会反复重建哈希表，这会急剧降低效率。为了解决这个问题，哈希表可以被“冻结”，即临时禁止调整数组大小。当添加和删除条目已经完成时，简单地“解冻”表，这时会进行一次优化计算。注意，如果添加大量数据，由于哈希冲突，“冻结”的表会“死”掉。在做任何查找以前将表“解冻”就会一切正常。

函数列表： 冻结和解冻 GHashTable

```
#include <glib.h>
/ * *冻结哈希表/
void g_hash_table_freeze(GHashTable* hash_table)
/ *将哈希表解冻* /
void g_hash_table_thaw(GHashTable* hash_table)
```

# GString

除了使用 gchar *进行字符串处理以外， Glib 还定义了一种新的数据类型： GString。它类似于标准 C 的字符串类型，但是 GString 能够自动增长。它的字符串数据是以 NULL 结尾的。这些特性可以防止程序中的缓冲溢出。这是一种非常重要的特性。下面是 GString 的定义：

```
struct GString
{
    gchar *str; /* Points to the st’rsi ncgurrent \0-terminated value. */
    gint len; /* Current length */
} ;
```

用下面的函数创建新的 GString 变量：

**GString *g_string_new( gchar *init );**

这个函数创建一个 GString，将字符串值 i n i t 复制到 GString 中，返回一个指向它的指针。如果 init 参数是 NULL，创建一个空 GString。

**void g_string_free( GString *string, gint free_segment );**

这个函数释放 string 所占据的内存。free_segment 参数是一个布尔类型变量。如果 free_segment 参数是 TRUE，它还释放其中的字符数据。

**GString *g_string_assign( GString *lval, const gchar *rval );**

这个函数将字符从 rval 复制到 lval，销毁 lval 的原有内容。注意，如有必要，lval 会被加长以容纳字符串的内容。这一点和标准的字符串复制函数 strcpy( )相同。下面的函数的意义都是显而易见的。其中以_c 结尾的函数接受一个字符，而不是字符串。

截取 string 字符串，生成一个长度为 len 的子串：

**GString *g_string_truncate( GString *string, gint len );**

将字符串 v a l 追加在 string 后面，返回一个新字符串：

**GString *g_string_append( GString *string, gchar *val );**

将字符 c 追加到 string 后面，返回一个新的字符串：

**GString *g_string_append_c( GString *string, gchar c );**

将字符串 v a l 插入到 string 前面，生成一个新字符串：
**GString *g_string_prepend( GString *string, gchar *val );**
将字符 c 插入到 string 前面，生成一个新字符串：
**GString *g_string_prepend_c( GString *string, gchar c );**
将一个格式化的字符串写到 string 中，类似于标准的 s p r int f 函数：
**void g_string_sprintf( GString *string, gchar *fmt, …) ;**
将一个格式化字符串追加到 string 后面，与上一个函数略有不同：
**void g_string_sprintfa ( GString *string, gchar *fmt, ... );**

## 计时器函数

计时器函数可以用 于为操作计时（例如，记录某项操作用了多长时间）。使用它的第一步是 g_timer_new ( )函数创建一个计时器，然后使用 g_timer_start ( )函数开始对操作计时，使用 g_timer_stop ( )函数停止对操作计时，用 g_timer_elapsed ( )函数判定计时器的运行时间。

创建一个新的计时器：
**GTimer *g_timer_new( void );**
销毁计时器：
**void g_timer_destroy( GTimer *timer );**
开始计时：
**void g_timer_start( GTimer *timer );**
停止计时：
**void g_timer_stop( GTimer *timer );**
计时重新置零：
**void g_timer_reset( GTimer *timer );**
获取计时器流逝的时间：
**gdouble g_timer_elapsed( GTimer *timer,gulong *microseconds );**

## 错误处理函数

**gchar *g_strerror( gint errnum );**
返回一条对应于给定错误代码的错误字符串信息，例如“ no such process”等。输出结果一般采用下面这种形式：
程序名：发生错误的函数名:文件或者描述：**strerror**
下面是一个使用 **g_strerror** 函数的例子：

```
g_print("hello_world:open:%s:%s\n", filename, g_strerror(errno));
```

**void g_error( gchar *format, ... );**
打印一条错误信息。格式与 printf 函数类似，但是它在信息前面添加“ ** ERROR **: ”，然后退出程序。它只用于致命错误。
**void g_warning( gchar *format, ... );**
与上面的函数类似，在信息前面添加“ ** WARNING **:”，不退出应用程序。它可以用于不太严重的错误。

**void g_message( gchar *format, ... );**
在字符串前添加“message: ”，用于显示一条信息。
**gchar *g_strsignal( gint signum );**
打印给定信号号码的 Linux 系统信号的名称。在通用信号处理函数中很有用。

## 实用函数

glib 还提供了一系列实用函数，可以用于获取程序名称、当前目录、临时目录等。这些函数都是在 glib . h 中定义的。

/ *返回应用程序的名称* /
**gchar* g_get_prgname (void);**
/ *设置应用程序的名称* /
**void g_set_prgname (const gchar *prgname);**
/ *返回当前用户的名称* /
**gchar* g_get_user_name (void);**
/ *返回用户的真实名称。该名称来自“ passwd”文件。返回当前用户的主目录* /
**gchar* g_get_real_name (void);**
/ *返回当前使用的临时目录，它按环境变量 TMPDIR、TMP and TEMP 的顺序查找。如果上面的环境变量都没有定义，返回“ / t m p”* /
**gchar* g_get_home_dir (void);**
**gchar* g_get_tmp_dir (void);**
/ *返回当前目录。返回的字符串不再需要时应该用 g_free ( ) 释放* /
**gchar* g_get_current_dir (void);**
/ *获得文件名的不带任何前导目录部分的名称。它返回一个指向给定文件名字符串的指针* /
**gchar* g_basename (const gchar *file_name);**
/ *返回文件名的目录部分。如果文件名不包含目录部分，返回“ .”。返回的字符串不再使用时应该用 g_free( ) 函数释放* /
**gchar* g_dirname (const gchar *file_name);**
/ *如果给定的 file_name 是绝对文件名（包含从根目录开始的完整路径，比如/usr/local），返回 TRUE * /
**gboolean g_path_is_absolute (const gchar *file_name);**
/ *返回一个指向文件名的根部标志（“/”）之后部分的指针。如果文件名 file_name 不是一个绝对路径，返回 NULL * /
**gchar* g_path_skip_root (gchar *file_name);**
/ *指定一个在正常程序终止时要执行的函数* /
**void g_atexit (GVoidFunc func);**

上面介绍的只是 glib 库中的一小部分， glib 的特性远远不止这些。如果想了解其他内容，请参考 glib . h 文件。这里面的绝大多数函数都是简明易懂的。另外， http://www.gtk.org 上的 glib 文档也是极好的资源。
如果你需要一些通用的函数，但 glib 中还没有，考虑写一个 glib 风格的例程，将它贡献到 glib 库中！你自己，以及全世界的 glib 使用者，都将因为你的出色工作而受益。

# Glib 常用数据结构

```
int main(int argc, char** argv)
{
    GSList* list = NULL;
    list = g_slist_append(list, "second");
    list = g_slist_prepend(list, "first");
    g_printf("The list is now %d items long\n", g_slist_length(list));
    list = g_slist_remove(list, "first");
    g_printf("The list is now %d items long\n", g_slist_length(list));
    g_slist_free(list);
    return 0;
}
```

***** Output *****

```
The list is now 2 items long

The list is now 1 items long
```

这段代码中大部分看起来都比较熟悉，不过有一些地方需要考虑；

* 如果调用 g_slist_remove 时传递了一个并不在列表中的条目，那么列表将不会发生变化。
* g_slist_remove 也会返回列表的新起始位置。
* 可以发现，“first”是调用 g_slist_prepend 添加的。这是个调用比 g_slist_append 更快；它是 O(1) 操作而不是 O(n) 操作，原因如前所述，进行附加需要遍历整个列表。所以，如果使用 g_slist_prepend 更为方便，那么就应该使用它。

# 删除重复的条目

这里是当在一个列表中有重复的条目时会发生的问题：

```
//ex-gslist-3.c
#include <glib.h>
int main(int argc, char** argv)
{
    GSList* list = NULL;
    list = g_slist_append(list, "first");
    list = g_slist_append(list, "second");
    list = g_slist_append(list, "second");
```

```
        list = g_slist_append(list, "third");
        list = g_slist_append(list, "third");
        g_printf("The list is now %d items long\n", g_slist_length(list));
        list = g_slist_remove(list, "second");
        list = g_slist_remove_all(list, "third");
        g_printf("The list is now %d items long\n", g_slist_length(list));
        g_slist_free(list);
        return 0;
    }
```

***** Output *****

```
    The list is now 5 items long
    The list is now 2 items long
```

**所以，如果 GSList 包含了两个同样的指针，而调用了 g_slist_remove，那么只会删除第一个指针。不过，可以使用 g_slist_remove_all 删除条目的所有指针。**

## 最后一个、第 n 个和第 n 个数据

在 GSList 中有了一些条目后，可以通过不同的方式提取它们。这里是一些示例，并在 g_printf 语句中给出了解释。

```
//ex-gslist-4.c
    #include <glib.h>
    int main(int argc, char** argv)
    {
        GSList* list = NULL;
        list = g_slist_append(list, "first");
        list = g_slist_append(list, "second");
        list = g_slist_append(list, "third");
        g_printf("The last item is '%s'\n", g_slist_last(list)->data);
        g_printf("The item at index '1' is '%s'\n", g_slist_nth(list, 1)->data);
        g_printf("Now the item at index '1' the easy way: '%s'\n", g_slist_nth_data(list, 1));
        g_printf("And the 'next' item after first item is '%s'\n", g_slist_next(list)->data);
        g_slist_free(list);
        return 0;
    }
```

***** Output *****

```
    The last item is 'third'
    The item at index '1' is 'second'
    Now the item at index '1' the easy way: 'second'
```

And the 'next' item after first item is 'second'

注意，有一些可以作用于 GSList 的快捷函数；可以简单地调用 g_slist_nth_data，而不需调用先 g_slist_nth 然后再反引用（dereference）返回的指针。
最后一个 g_printf 语句稍有不同。g_slist_next 不是一个函数调用，而是一个宏。它展开为一个指向 GSList 中下一元素的指针反引用。在这种情况下，可以看到，我们传递了 GSList 中的第一个元素，于是那个宏展并给出第二个元素。这也是一个快速的操作，因为没有函数调用的开销。

## 使用用户定义的类型 ---结构体

到现在为止我们一直在使用字符串；也就是说，我们只是将指向字符的指针放入到 GSList 中。在下面的代码示例中，将会定义一个 Person 结构体，并将这个结构体的一些实例放入到 GSList 中：

```
//ex-gslist-5.c
#include <glib.h>
#include <malloc.h>
typedef struct {
    char* name;
    int shoe_size;
} Person;
int main(int argc, char** argv)
{
    GSList* list = NULL;
    Person* tom = (Person*)malloc(sizeof(Person));
    tom->name = "Tom";
    tom->shoe_size = 12;
    list = g_slist_append(list, tom);
    Person* fred = g_new(Person, 1); // allocate memory for one Person struct
    fred->name = "Fred";
    fred->shoe_size = 11;
    list = g_slist_append(list, fred);
    g_printf("Tom's shoe size is '%d'\n", ((Person*)list->data)->shoe_size);
    g_printf("The last Person's name is '%s'\n",
    ((Person*)g_slist_last(list)->data)->name);
    g_slist_free(list);
    free(tom);
    g_free(fred);
    return 0;
}
***** Output *****
```

Tom's shoe size is '12'
The last Person's name is 'Fred'

## 关于使用 GLib 和用户定义类型的一些注解：

* 可以像使用字符串一样在 GSList 使用用户定义类型。另外要注意，当从列表中取出条目时，需要进行一些强制类型转换。
* 这个示例使用了另一个 GLib 宏 —— g_new 宏 —— 来创建 Fred Person 实例。这个宏只是展开并使用 malloc 为给定的类型分配适当数量的内存，但是这比手工输入 malloc 函数调用更为简洁。
* 最后，如果要分配内存，那么还需要释放它。可以看到上面的代码示例如何使用 GLib 函数 g_free 来为 Fred Person 实例完成此任务（因为它是使用 g_new 分配的）。在大部分情况下 g_free 只是会包装 free 函数，但是 GLib 也具备内存池功能，g_free 以及其他内存管理函数可以使用它们。

## 组合、反转，等等

GSList 附带了一些便利的工具，可以连接和反转列表。这里是它们的工作方式：

```
//ex-gslist-6.c
#include <glib.h>

void display_list(GSList *list)
{
   GSList *iterator = NULL;
   for (iterator = list; iterator; iterator = iterator->next)
   {
      g_printf("%s ", (int*)iterator->data);
   }
   g_printf("\n");
}
int main(int argc, char** argv)
{
      GSList* list1 = NULL;
      list1 = g_slist_append(list1, "first");
      list1 = g_slist_append(list1, "second");
      GSList* list2 = NULL;
      list2 = g_slist_append(list2, "third");
      list2 = g_slist_append(list2, "fourth");
      GSList* both = g_slist_concat(list1, list2);                    //连接两个表
      display_list(both);
      g_printf("The third item in the concatenated list is '%s'\n", g_slist_nth_data(both, 2));
      GSList* reversed = g_slist_reverse(both);                       //反转一个表
      g_printf("The first item in the reversed list is '%s'\n", reversed->data);
```

```
    display_list(reversed);
    g_slist_free(reversed);
    return 0;
}
```

OUT：：：

```
first second third fourth
The third item in the concatenated list is 'third'
The first item in the reversed list is 'fourth'
fourth third second first
```

正如所预期的，两个列表首尾相连在一起，list2 中的第一个条目成为新的列表中的第三个条目。注意，并没有拷贝条目；它们只是被链接上，这样内存只需要释放一次。

另外，您会发现您只需要使用一个指针反引用（reversed->data）就可以打印出反向列表的第一个条目。由于 GSList 中的每一个条目都是一个指向某个 GSList 结构体的指针，所以要获得第一个条目并不需要调用函数。

## 简单遍历

这里是遍历 GSList 中所有内容的一个直观方法：

```
//ex-gslist-7.c
#include <glib.h>
int main(int argc, char** argv)
{
    GSList* list = NULL, *iterator = NULL;
    list = g_slist_append(list, "first");
    list = g_slist_append(list, "second");
    list = g_slist_append(list, "third");
    for (iterator = list; iterator; iterator = iterator->next) {
        g_printf("Current item is '%s'\n", iterator->data);
    }
    g_slist_free(list);
    return 0;
}
```

```
***** Output *****
  Current item is 'first'
  Current item is 'second'
  Current item is 'third'
```

## 迭代器

迭代器（iterator）对象只是一个声明为指向 GSList 结构体的变量。这看似奇怪，不过却能满足要求。由于单向列表是一系列 GSList 结构体，所以迭代器与列表的类型应该相同。

另外，注意这个代码段使用的是通常的 GLib 用法习惯；在声明 GSList 本身的时候就声明了迭代器变量。

最后，for 循环的退出表达式检查迭代器是否为 NULL。这样是有效的，因为只有当循环传递了列表中的最后一个条目后它才会成为 NULL。

## 使用函数进行高级遍历

遍历列表的另一种方法是使用 g_slist_foreach，并提供一个将为列表中的每一个条目调用的函数。

```
//ex-gslist-8.c
    #include <glib.h>
    void print_iterator(gpointer item, gpointer prefix)
    {
        g_printf("%s %s\n", prefix, item);
    }
    void print_iterator_short(gpointer item)
    {
        g_printf("%s\n", item);
    }
    int main(int argc, char** argv)
    {
        GSList* list = g_slist_append(NULL, g_strdup("1"));
        list = g_slist_append(list, g_strdup("2"));
        list = g_slist_append(list, g_strdup("third"));

        g_printf("Iterating with a function:\n");
        g_slist_foreach(list, print_iterator, "-->");
            //如果第一个条目小于第二个，则 GCompareFunc 返回一个负值，如果相等则返回 0，如果第二个大于第一个则返回一个正值
            //链表中的每个元素都调用上面的子函数  同时在每个输出前加“－－＞”
        g_printf("Iterating with a shorter function:\n");
        g_slist_foreach(list, (GFunc)print_iterator_short,NULL);
        g_printf("Now freeing each item\n");

        g_slist_free(list);
        return 0;
    }
***** Output *****
```

```
Iterating with a function:
--> first
--> second
--> third
Iterating with a shorter function:
first
second
third
Now freeing each item
```

注意:

g_slist_foreach(list, print_iterator, "-->");  中第一个参数是链表
第二个参数是调用的上面的子函数
第三个函数是子函数中第二个参数

**在这个示例中有很多好东西：**

* 一条类似 GSList x = g_slist_append(NULL, [whatever]) 的语句让您能够一举声明、初始化并将第一个条目添加到列表。
*** g_strdup 函数可以方便地复制字符串；如果使用了它，那么要记得释放。**
* g_slist_foreach 允许传递一个指针，这样就可以根据列表中的每个条目有效地为其赋与任意参数。例如，可以传递一个累加器并收集关于列表中每个条目的信息。遍历函数的唯一受 限之处在于它至少要使用一个 gpointer 作为参数；现在可以了解在只接收一个参数时 print_interator_short 如何工作。
* 注意，代码使用一个内置的 GLib 函数作为 g_slist_foreach 的参数来释放所有字符串。在此示例中，所有需要做的只是将 g_free 强制类型转换为 GFunc 以使其生效。注意，仍然可以单独使用 g_slist_free 来释放 GSList 本身。

# 使用 GCompareFunc 排序

可以通过提供一个知道如何比较列表中条目的函数来对 GSLit 进行排序。下面的示例展示了对字符串列表进行排序的一种方法：(((((((((未仔细看)))))))))

//ex-gslist-9.c

```
#include <glib.h>
gint my_comparator(gconstpointer item1, gconstpointer item2)
{
    return g_ascii_strcasecmp(item1, item2);
    //比较两个条目,如果第一个条目小于第二个，则返回一个负值，如果相等则返回 0，如果第二个大于第一个则返回一个正值
}
int main(int argc, char** argv)
{
    GSList* list = g_slist_append(NULL, "Chicago");
    list = g_slist_append(list, "Boston");
```

```
        list = g_slist_append(list, "Albany");
        list = g_slist_sort(list, (GCompareFunc)my_comparator);
        ///按从小到大到数据进行排列
        g_printf("The first item is now '%s'\n", list->data);
        g_printf("The last item is now '%s'\n", g_slist_last(list)->data);
        g_slist_free(list);
        return 0;
    }
***** Output *****
    The first item is now 'Albany'
    The last item is now 'Chicago'
```

注意，如果第一个条目小于第二个，则 GCompareFunc 返回一个负值，如果相等则返回 0，如果第二个大于第一个则返回一个正值。只要您的比较函数符合此规范，在内部它可以做任何所需要的事情。

另外，由于各种其他 GLib 函数也遵循此模式，所以可以简单地委派给它们。实际上，在上面的示例中，可以简单地把 my_comparator 调用替换为 g_slist_sort(list, (GCompareFunc)g_ascii_strcasecmp) 等调用，会获得相同的结果。

## 查找元素

有一些方法可以在 GSList 查找元素。已经介绍了如何简单地遍历列表的全部内容，比较每个条目，直到找到了目标条目。如果已经拥有了要寻找的数据，而只是想要获得它在列表中的位置， 那么可以使用 g_slist_find。最后，可以使用 g_slist_find_custom，它允许您使用一个函数来检查列表中的每一个条目。下面展示了 g_slist_find 和 g_slist_find_custom：

```
//ex-gslist-10.c
    #include <glib.h>
    gint my_finder(gconstpointer item)
    {
        return g_ascii_strcasecmp(item, "second");
    }
    int main(int argc, char** argv)
    {
        GSList* list = g_slist_append(NULL, "first");
        list = g_slist_append(list, "second");
        list = g_slist_append(list, "third");
        GSList* item = g_slist_find(list, "second");
        g_printf("This should be the 'second' item: '%s'\n", item->data);
        item = g_slist_find_custom(list, NULL, (GCompareFunc)my_finder);
        g_printf("Again, this should be the 'second' item: '%s'\n", item->data);
        item = g_slist_find(list, "delta");
        g_printf("'delta' is not in the list, so we get: '%s'\n", item ? item->data : "(null)");
        g_slist_free(list);
        return 0;
    }
```

***** Output *****

```
This should be the 'second' item: 'second'
Again, this should be the 'second' item: 'second'
'delta' is not in the list, so we get: '(null)'
```

注 意，g_slist_find_custom 也使用了一个可以指向任意内容的指针作为第二个参数，所以，如果需要，可以传递进来一些内容以帮助查找函数。另外，GCompare 函数是最后一个参数，而不是第二个参数，因为它在 g_slist_sort 之中。最后，失败的查找会返回 NULL。

## 通过插入进行高级添加

既然已经接触过几次 GCompareFunc，一些更有趣的插入操作会更有意义。使用 g_slist_insert 可以将条目插入到指定的位置，使用 g_slist_insert_before 可以将条目插入到特定位置之前，使用 g_slist_insert_sorted 可以进行有序插入。这里是样例：

//ex-gslist-11.c

```
#include <glib.h>
int main(int argc, char** argv)
{
    GSList* list = g_slist_append(NULL, "Anaheim "), *iterator = NULL;
    list = g_slist_append(list, "Elkton ");
    g_printf("Before inserting 'Boston', second item is: '%s'\n", g_slist_nth(list, 1)->data);
    g_slist_insert(list, "Boston ", 1);
    g_printf("After insertion, second item is: '%s'\n", g_slist_nth(list, 1)->data);
    list = g_slist_insert_before(list, g_slist_nth(list, 2), "Chicago ");
    g_printf("After an insert_before, third item is: '%s'\n", g_slist_nth(list, 2)->data);
    list = g_slist_insert_sorted(list, "Denver ", (GCompareFunc)g_ascii_strcasecmp);
    g_printf("After inserting 'Denver', here's the final list:\n");
    g_slist_foreach(list, (GFunc)g_printf, NULL);
    g_slist_free(list);
    return 0;
}
```

***** Output *****

```
Before inserting 'Boston', second item is: 'Elkton '
After insertion, second item is: 'Boston '
After an insert_before, third item is: 'Chicago '
After inserting 'Denver', here's the final list:
Anaheim Boston Chicago Denver Elkton
```

由于 g_slist_insert_sorted 使用了 GCompareFunc，所以复用内置的 GLib 函数 g_ascii_strcasecmp 也很简单。现在您应该已经能理解为什么在每个条目的末尾有额外的空间；在代码示例的最后加入了另一个 g_slist_foreach 示例，这一次是使用 GFunc 作为参数。

## 实际应用

在先前提到的三个开放源代码的实际应用程序中，可以发现在很多地方使用了 GSList。大部分用法相当普通，有很多插入、附加、删除，等等。不过有一些更有趣的东西。

**Gaim 使用 GSList 来保存当前会话以及大部分插件中的各种内容：**

* gaim-1.2.1/src/away.c 使用有序的 GSList 来保存“离开消息”（客户机离线期间收到的消息）。它使用了一个定制的 GCompareFunc，即 sort_awaymsg_list，用于保存那些以发送者名称排序的消息。

* gaim-1.2.1/src/protocols/gg/gg.c 使用 GSList 来保存允许帐号的一个列表；然后它使用一个定制的查找器来确认某个帐号在列表中。定制的查找器简单地委派 g_ascii_strcasecmp，所以有可能会弃用它，并将 g_ascii_strcasecmp 直接传递给 g_slist_find_custom。

**Evolution 同样使用了很多 GSList：**

* evolution-data-server-1.0.2/calendar/libecal/e-cal-component.c 使用 GSList 来存会议参与者。有时，它会反复调用 g_slist_prepend，并在结束时使用 g_slist_reverse 令条目按期望排序，以此构建那个 GSList。如前所述，这样做比使用 g_slist_append 添加条目更快。

* evolution-2.0.2/addressbook/gui/contact-editor/e-contact-editor.c 在一种卫述句（guard clause）中使用 g_slist_find；它在信号处理器中使用它，以确保它在一个信号回调中接收到的 EContactEditor 在被作为参数传递到某个函数之前能够保存。

**GIMP 也以一些适宜的方式使用了 GSList：**

* gimp-2.2.4/plug-ins/maze/algorithms.c 在迷宫生成（maze-generations）算法中使用 GSList 追踪单元（cells）。

* gimp-2.2.4/app/widgets/gimpclipboard.c 使用 GSList 来保存剪贴板象素缓冲区格式（比如 PNG 和 JPEG）；它向 g_slist_sort 传递一个定制的 GCompareFunc。

* gimp-2.2.4/app/core/gimppreviewcache.c 使用 GSList 作为一种基于大小的队列；它在 GSList 中保存图象预览，使用 g_slist_insert_sorted 优先插入较小的图象。同一文件中的另一个函数会遍历同一个 GSList 并比较每一个条目的表面区域，找出要删除的最小那一个，以此来整理缓存。

## 双向链表

## 概念

双向链表与单向链表非常类似，不过它们包含有另外的指针，以支持更多导航选项；给定双向链表中的一个节点，可以向前移动，也可以向后移动。这使得它们比单向链表更灵活，但也使用了更多内存，所以，除非确实需要这种灵活性，否则不要使用这种双向链表。

GLib 包含有一个名为 GList 的双向链表实现。GList 中的大部分操作与 GSList 中的类似。我们将查看基本用法的一些示例，然后是 GList 所支持的附加操作。

## 基本操作

这里是使用 GList 可以进行的一些常见操作：

```
//ex-glist-1.c
    #include <glib.h>
    int main(int argc, char** argv)
    {
        GList* list = NULL;
        list = g_list_append(list, "Austin ");
        g_printf("The first item is '%s'\n", list->data);
        list = g_list_insert(list, "Baltimore ", 1);
        g_printf("The second item is '%s'\n", g_list_next(list)->data);
        list = g_list_remove(list, "Baltimore ");
        g_printf("After removal of 'Baltimore', the list length is %d\n", g_list_length(list));
        GList* other_list = g_list_append(NULL, "Baltimore ");
        list = g_list_concat(list, other_list);
        g_printf("After concatenation: ");
        g_list_foreach(list, (GFunc)g_printf, NULL);
        list = g_list_reverse(list);
        g_printf("\nAfter reversal: ");
        g_list_foreach(list, (GFunc)g_printf, NULL);
        g_list_free(list);
        return 0;
    }
***** Output *****
    The first item is 'Austin '
    The second item is 'Baltimore '
    After removal of 'Baltimore', the list length is 1
    After concatenation: Austin Baltimore
    After reversal: Baltimore Austin
```

上面的代码看起来非常熟悉!上面所有操作都在 GSList 中出现过；唯一的区别是 GList 的函数名是 g_list 开头，而不是 g_slist。当然，它们全都要使用指向 GList 结构体而不是指向 GSList 结构体。

## 更好的导航

已经了解了一些基本的 GList 操作后，这里是一些可能的操作，唯一的原因就是 GList 中的每个节点都有一个指向前一个节点的链接：

```
//ex-glist-2.c
    #include <glib.h>
    int main(int argc, char** argv)
    {
```

```
    GList* list = g_list_append(NULL, "Austin ");
    list = g_list_append(list, "Bowie ");
    list = g_list_append(list, "Charleston ");
    g_printf("Here's the list: ");
    g_list_foreach(list, (GFunc)g_printf, NULL);
    GList* last = g_list_last(list);
    g_printf("\nThe first item (using g_list_first) is '%s'\n", g_list_first(last)->data);
    g_printf("The next-to-last item is '%s'\n", g_list_previous(last)->data);
    g_printf("The next-to-last item is '%s'\n", g_list_nth_prev(last, 1)->data);
    g_list_free(list);
    return 0;
}

***** Output *****

    Here's the list: Austin Bowie Charleston
    The first item (using g_list_first) is 'Austin '
    The next-to-last item is 'Bowie '
    The next-to-last item is 'Bowie '
```

没有太令人惊讶的事情，不过有一些注解：

* g_list_nth_prev 的第二个参数是一个整数，表明需要向前导航多少个节点。如果传递的值超出了 GList 的范围，那么要准备好处理程序的崩溃。

* g_list_first 是一个 O(n) 操作。它从一个给定的节点开始向后搜索，直到找到 GList 的头。上面的示例是一个最坏的情形，因为遍历是从列表的末尾开始的。同理，g_list_last 也是一个 O(n) 操作。

## 使用链接删除节点

已经了解了在拥有指向其容纳的数据的指针的前提下如何从列表中删除一个节点；g_list_remove 可以很好地完成。如果拥有一个指向节点本身的指针，可以通过一个快速的 O(1) 操作直接删除那个节点。

```
//ex-glist-3.c
#include <glib.h>
int main(int argc, char** argv)
{
    GList* list = g_list_append(NULL, "Austin ");
    list = g_list_append(list, "Bowie ");
    list = g_list_append(list, "Chicago ");
    g_printf("Here's the list: ");
    g_list_foreach(list, (GFunc)g_printf, NULL);
    GList* bowie = g_list_nth(list, 1);

    list = g_list_remove_link(list, bowie);
    g_list_free_1(bowie);
```

```
    g_printf("\nHere's the list after the remove_link call: ");
    g_list_foreach(list, (GFunc)g_printf, NULL);

    list = g_list_delete_link(list, g_list_nth(list, 1));
    g_printf("\nHere's the list after the delete_link call: ");
    g_list_foreach(list, (GFunc)g_printf, NULL);
    g_list_free(list);
    return 0;
}
```

***** Output *****

```
Here's the list: Austin Bowie Chicago
Here's the list after the remove_link call: Austin Chicago
Here's the list after the delete_link call: Austin
```

所以如果拥有一个指向节点而不是数据的指针，那么可以使用 g_list_remove_link 删除那个节点。删除它以后，还需要使用 g_list_free_1 显式地释放它，其所做的事情正如名字所暗示的：它释放一个节点。照例，需要保存好 g_list_remove_link 的返回值，因为那里是列表的新起始位置。

最后，如果想做的只是删除并释放某个节点，那么可以调用 g_list_delete_link 来一步完成。GSList 也有相同的函数；只需要将 g_list 替换为 g_slist，上面的所有信息都适用。

## 索引和位置

如果只是想找出某个条目在 GList 中的位置，那么有两种选择。可以使用 g_list_index，它可以使用条目中的数据找出它，或者可以使用 g_list_position，它使用的是指向那个节点的指针。这个示例展示了这两种方法：

```
//ex-glist-4.c
#include <glib.h>
int main(int argc, char** argv)
{
    GList* list = g_list_append(NULL, "Austin ");
    list = g_list_append(list, "Bowie ");
    list = g_list_append(list, "Bowie ");
    list = g_list_append(list, "Cheyenne ");
    g_printf("Here's the list: ");
    g_list_foreach(list, (GFunc)g_printf, NULL);
    g_printf("\nItem 'Bowie' is located at index %d\n", g_list_index(list, "Bowie "));
    g_printf("Item 'Dallas' is located at index %d\n", g_list_index(list, "Dallas"));
    GList* last = g_list_last(list);
    g_printf("Item 'Cheyenne' is located at index %d\n", g_list_position(list, last));
    g_list_free(list);
    return 0;
}
```

***** Output *****

Here's the list: Austin Bowie Bowie Cheyenne
Item 'Bowie' is located at index 1
Item 'Dallas' is located at index -1
Item 'Cheyenne' is located at index 3

注意，如果 g_list_index 找不到那个数据，则它返回的值为 -1。如果两个节点具有相同的数据值，那么 g_list_index 会返回最先出现的索引。如果找不到指定的节点，g_list_position 也会返回 -1。另外，在 GSList 也有这些方法，只是名字不同。

## 实际应用

让我们来研究在前面提到的开放源代码的应用程序中 GList 的应用。

**Gaim 使用了很多 GList：**

* 在关闭“add a buddy”对话框时，gaim-1.2.1/plugins/gevolution/add_buddy_dialog.c 使用 g_list_foreach 调用来释放对每个联系人的引用。

* gaim-1.2.1/src/account.c 使用一个 GList 来保存所有帐号；它使用 g_list_find 来确保某个帐号在使用 g_list_append 进行添加之前并不存在。

**Evolution 使用 GList:**

* evolution-2.0.2/filter/filter-rule.c 使用 GList 来保存邮件过滤规则的各部分（比如要检查的主题行）； filter_rule_finalise 使用 g_list_foreach 释放对那些部分的引用。

* evolution-2.0.2/calendar/gui/alarm-notify/alarm.c 使用 GList 保存警告； queue_alarm 借助一个定制的 GCompareFunc 使用 g_list_insert_sorted 将新的警告插入到适当的位置。

**在 GIMP 中的应用：**

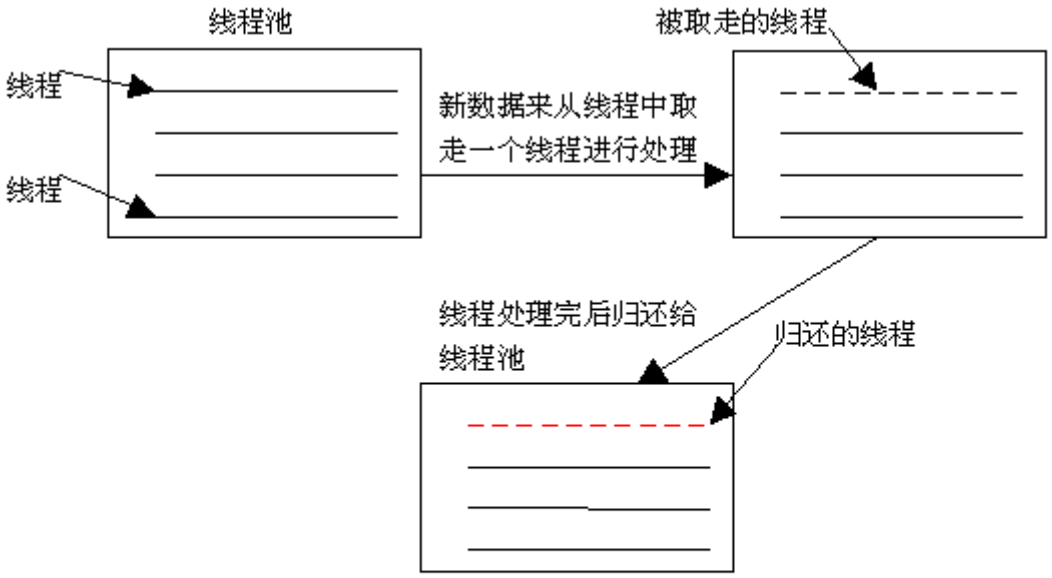
* gimp-2.2.4/app/file/gimprecentlist.c 使用 GList 保存最近访问过的文件；gimp_recent_list_read 从某个 XML 文件描述符中读取文件名，并在返回 GList 之前调用 g_list_reverse。

* gimp-2.2.4/app/vectors/gimpbezierstroke.c 使用 GList 保存笔划连接点（stroke anchors）； gimp_bezier_stroke_connect_stroke 使用 g_list_concat 来帮助将一个笔划连接到另一个笔划。

## glib 库异步队列和线程池代码分析

本文章主要讲了两部分内容：一是分析了异步队列的原理和实现，二是分析线程池的原理和实现。

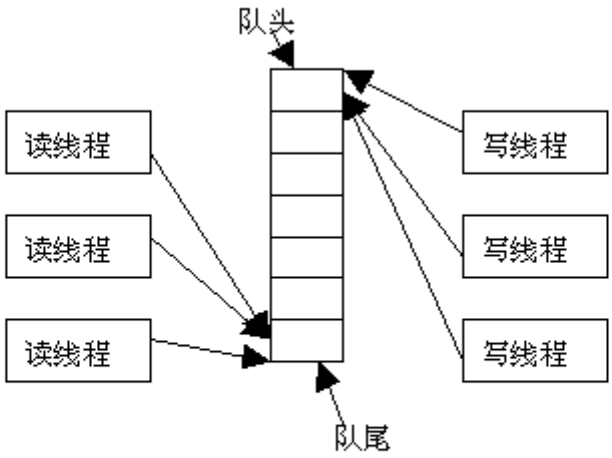
在多线程程序的运行中，如果经常地创建和销毁执行过程相似而所用数据不同的线程，系统的效率，系统资源的利用率将会受到极大的影响。对于这一问题可用类似 glib 库中的线程池的解决办法。




在线程池中的线程被使用的一般过程

我们可以这样想像线程池的处理，当有新的数据要交给线程处理时，主程序/主线程 就从线程池中找到一个未被使用的线程处理这新来的数据，如果线程池中没有找到可用的空闲线程，就新创建一个线程来处理这个数据，并在处理完后不销毁它而是 把这个线程放到线程池中，以备后用。线程池的这个原理和内存管理中 slab 机制有异曲同工之妙!我想无论是线程池的这种处理方式还是 slab 机制，其本质 思想还是一致的。

近来做的项目，在框架中用到了多线程的异步队列，实现形式和 glib 中的异步队列极其相似，而 glib 线程池中的代码也用到了异步队列（名字用同步队列更合适），因此就先分析一下异步队列。异步队列的概念是这样的：**所有的数据组织成队列，供多线程并发访问，而这些并发控制全部在异步队列里面实现，对外面只提供读写接口；当队列中的数据为空时， 如果是读线程访问异步队列，那么这一读线程就等待，直到有数据为止；写线程向队列放数据时，如果有线程在等待数据就唤醒等待线程。**




数据队列就是多线程访问的临界区

异步队列主要代码剖析：

异步队列的数据结构如下：

```
struct_GAsyncQueue
{
    GMutex *mutex; //互斥变量
    GCond *cond; //等待条件
    GQueue *queue; //数据队列
    guint waiting_threads; //等待的读线程个数
    gint32 ref_count;
};
g_async_queue_push_unlocked (GAsyncQueue* queue, gpointer data)
{
    ……//这些点代表一些省略的代码
    //把数据放入队列
    g_queue_push_head (queue->queue, data);
    //现在队列已经有数据了，判断是否有读线程在等待数据，
      //如果有就发送信号唤醒读线程
      if (queue->waiting_threads > 0)
        g_cond_signal (queue->cond);
}
g_async_queue_push (GAsyncQueue* queue, gpointer data)
{
    ……..
    g_mutex_lock (queue->mutex); //在访问临界区前先获得互斥变量
    g_async_queue_push_unlocked (queue, data); //执行写数据操作
    g_mutex_unlock (queue->mutex); //释放互斥变量，以使其它线程可以进入临界区
}
```

从以上的接口可看出，”…._unlocked” 这样的接口就是异步队列这个对象已获得互斥变量的接口，glib 中线程处理相关接口都有类似的命名规则，在接下来的代码分析中，如没有特别的需要就只看”…._unlocked” 这样的接口。

// 读线程从异步队列中获取数据的接口

// try 参数和时间参数在多线程同步/内核多进程的实现中是很常见的东西了，在这里就不再作特殊的解释了。

```
g_async_queue_pop_intern_unlocked (GAsyncQueue *queue,
                                   gboolean     try,
                                   GTimeVal    *end_time)
{
gpointer retval;
//判断是否有数据在队列中，如果没有就要执行 if 语句相应的睡眠等待，直到被写进程唤醒
if (!g_queue_peek_tail_link (queue->queue))
    {
      if (try)//如果 try 为真，则永远不睡眠
       return NULL;

    // 接下来是要让线程进行睡眠等待了，在等待之前先确保等待条件已创建
      if (!queue->cond)
       queue->cond = g_cond_new ();
      if (!end_time) // 等待无时间限制
       {
         queue->waiting_threads++; // 等待线程数加一
      // 这里为什么用循环？因为这是多线程的环境，有可能有多个读线程在等待
      // 当前线程被唤醒时，有可能数据队列中的数据又被别的线程读走了，所以
      // 当前线程就得继续睡眠等待
      // 注意：睡眠等待时会暂时放弃互斥锁，被唤醒时会重新获取互斥锁
       while (!g_queue_peek_tail_link (queue->queue))
            g_cond_wait (queue->cond, queue->mutex);
         queue->waiting_threads--; // 等待线程数减一
       }
      else
       {
         queue->waiting_threads++;
         while (!g_queue_peek_tail_link (queue->queue))
          if (!g_cond_timed_wait (queue->cond, queue->mutex, end_time))
            break;
         queue->waiting_threads--;
         if (!g_queue_peek_tail_link (queue->queue))
           return NULL;
       }
    }
     retval = g_queue_pop_tail (queue->queue);
```

```
    g_assert (retval);
    return retval;
}
/* 返回数据队列的长度，也即数据队列中的数据个数.
* 如果是负值表明是等待数据的线程个数，正数表示数据队列的数据个数
* g_async_queue_length == 0 表示是有 'n' 个数据和' n' 个等待线程在数据队列
* 这种特殊情况可能是在对数据队列加锁或调度时发生
*/
g_async_queue_length_unlocked (GAsyncQueue* queue)
{
    g_return_val_if_fail (queue, 0);
    g_return_val_if_fail (g_atomic_int_get (&queue->ref_count) > 0, 0);
    return queue->queue->length - queue->waiting_threads;
}
```

有了前面的异步队列基础就可以分析线程池是怎么实现的了。在 glib 库中的线程池的实现和使用有两种方式：1. 单个线程池对象不共享方式；2. 多个线程池对象共享线程方式，也即把各个具体的线程池对象创建的把任务做完了的线程统一放在全局线程池中进行统一管理，各个具体的线程池对象要使用线程 时，可以先向全局线程池中取线程，如果全局线程池没有线程了具体的线程池对象就可自行创建线程。

线程池的数据结构有两部分，一部分是在头文件中，另一部分在 C 文件中。这是 C 语言中常用的信息隐藏方法之一，把要暴露给用户的数据放在头文件中，而要隐藏的数据则放在 C 文件中。下面是线程池头文件中的数据结构：

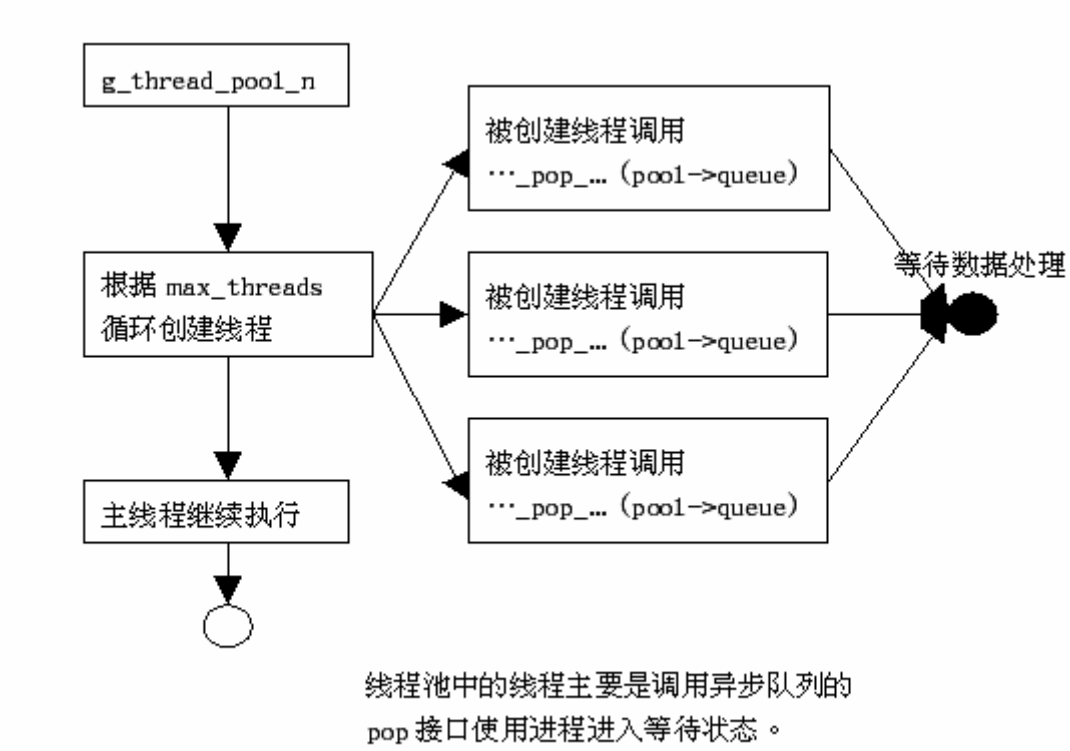
```
typedef struct GThreadPool      GThreadPool;
struct GThreadPool
{
      // 具体处理数据的函数
      // 它的第一个参数为 g_thread_pool_push 进去的数据，也即要执行的任务
    GFunc func;
    gpointer user_data; // func 的第二个参数
    // 通过这个成员控制线程池对象创建的线程是否在全局线程池中共享，
      // TRUE 为不共享，FALSE 为共享
    gboolean exclusive;
};
```

C 文件中线程池的数据结构:

```
typedef struct GRealThreadPool GRealThreadPool;
struct GRealThreadPool
{
    GThreadPool pool; // 头文件已定义
    GAsyncQueue* queue; // 异步数据队列
    GCond* cond;
    gint max_threads; // 线程池对象持有的线程数上限
    gint num_threads;// 池程池对象当前持有的线程数
    gboolean running;
    gboolean immediate;
```

```
    gboolean waiting;
    GCompareDataFunc sort_func;
    gpointer sort_user_data;
};
```

我们可以先来分析单个线程对象不共享的主要实现。在分析它的实现之前，可以先看看一个流程图




从上图可见当主线程有数据交给线程池处理时，只要调用异步队列相关的 push 接口，线程池中的任何一个线程都可以为这服务。根据以上的流程图看看单个线程对象不共享方式的主要实现代码，它的调用从创建线程池对象开始：

g_thread_pool_new--->g_thread_pool_start_thread--->
g_thread_create(g_thread_pool_thread_proxy,pool,FALSE,&local_error)--->>g_thread_pool_wait_for_new_task(pool）----> g_async_queue_pop_unlocked (pool->queue);

```
// max_threads 为 -1 时表示线程池中的线程数无限制并且线程由动态生成
// max_threads 为正整数时，线程池就会预先创建 max_threads 个线程
g_thread_pool_new (GFunc              func,
                   gpointer         user_data,
                   gint             max_threads,
                   gboolean         exclusive,
                   GError         **error)
{
GRealThreadPool *retval;
    ……. //这些点代表一些省略的代码
retval = g_new (GRealThreadPool, 1);
retval->pool.func = func;
retval->pool.user_data = user_data;
```

```
retval->pool.exclusive = exclusive;
retval->queue = g_async_queue_new (); //  创建异步队列
retval->cond = NULL;
retval->max_threads = max_threads;
retval->num_threads = 0;
retval->running = TRUE;
    …….
if (retval->pool.exclusive)
{
      g_async_queue_lock (retval->queue);
      while (retval->num_threads < retval->max_threads)
          {
            GError *local_error = NULL;
            g_thread_pool_start_thread (retval, &local_error);//  起动新的线程
            …….
       }
      g_async_queue_unlock (retval->queue);
}
return (GThreadPool*) retval;
}
g_thread_pool_start_thread (GRealThreadPool *pool,
                        GError          **error)
{
gboolean success = FALSE;
if (pool->num_threads >= pool->max_threads && pool->max_threads != -1)
    /* Enough threads are already running */
    return;
…….
if (!success)
{
      GError *local_error = NULL;
      /* No thread was found, we have to start a new one */
      //  真正创建一个新的线程
      g_thread_create (g_thread_pool_thread_proxy, pool, FALSE, &local_error);
      ……..
}
pool->num_threads++;
}
g_thread_pool_thread_proxy (gpointer data)
{
GRealThreadPool *pool;
pool = data;
……..
g_async_queue_lock (pool->queue);
```

```
while (TRUE)
{
      gpointer task;
      // 线程等待任务，也即等待数据，线程在等待就是处在线程池中的空闲线程
      task = g_thread_pool_wait_for_new_task (pool);
      // 如果线程被唤醒收到并数据就用此线程执行任务，否则继续循环等待
      // 注意：当任务做完时，继续循环又会调用上面的 g_thread_pool_wait_for_new_task
      // 而进入等待状态，
        if (task)
       {
            if (pool->running || !pool->immediate)
             {
               /* A task was received and the thread pool is active, so
              * execute the function.
              */
                g_async_queue_unlock (pool->queue);
                pool->pool.func (task, pool->pool.user_data);
                g_async_queue_lock (pool->queue);
           }
       }
      else
       {
           ……
      }
}
return NULL;
}
g_thread_pool_wait_for_new_task (GRealThreadPool *pool)
{
gpointer task = NULL;
if (pool->running || (!pool->immediate &&
                    g_async_queue_length_unlocked (pool->queue) > 0))
{
      /* This thread pool is still active. */
      if (pool->num_threads > pool->max_threads && pool->max_threads != -1)
       {
           ……..
       }
     else if (pool->pool.exclusive)
       {
           /* Exclusive threads stay attached to the pool. */
        // 调用异步队列的 pop 接口进入等待状态，到此一个线程的创建过程就完成了
           task = g_async_queue_pop_unlocked (pool->queue);
       }
```

```
      else
      {
          ……
      }
}
else
{
    ……
}
return task;
}
```

现在可以结合流程图分析线程池中创建一个线程的一个情景：从函数 g_thread_pool_new 的 while 循环调用了 g_thread_pool_start_thread 函数，在函数中直接调用 g_thread_create 创建线程，被创建的线程调用函数 g_thread_pool_wait_for_new_task 循环等待任务的到来，函数 g_thread_pool_wait_for_new_task 调用 g_async_queue_pop_unlocked (pool->queue) 真正进入等待。如此可知，最终新创建的线程是调用异步队列的 pop 接口进入等待状态的，这样一个线程的创建就大功告成 了。而函数 g_thread_pool_new 的 while 循环结束时就创建了 max_threads 个等待线程，也即这个新建的线程池对象有了 max_threads 个线程以备使用。

创建线程池、线程池中的线程是为了使用它，在线程池中取线程，叫线程干活的过程就很简单多了，这个调用过程：g_thread_pool_push→ g_thread_pool_queue_push_unlocked→ g_async_queue_push_unlocked。可见最终调用的是异步数据队列的 push 接口，把要处理的数据插入队列后它就会唤醒等待异步队列数据的等待线程。

```
g_thread_pool_push (GThreadPool *pool,
                    gpointer       data,
                    GError       **error)
{
……
//
if (g_async_queue_length_unlocked (real->queue) >= 0)
    /* No thread is waiting in the queue */
    g_thread_pool_start_thread (real, error);
g_thread_pool_queue_push_unlocked (real, data);
g_async_queue_unlock (real->queue);
}
g_thread_pool_queue_push_unlocked (GRealThreadPool *pool,
                                   gpointer         data)
{
   ……
    g_async_queue_push_unlocked (pool->queue, data);
}
```

总结：单个线程池对象不共享方式在管理多线程时是以线程池对象中的异步队列为中心，新创建的线程或做完任务的线程并不释放，让它调用异步队列的 pop 接口进入等待状态，而在使用唤醒线程池中的线程就是调用异步队列的 push 接口。

以上对于理解线程池的实现已经足够，多个线程池对象共享线程方式和具体线程池的销毁的技巧，在这里就不讨论了。

# 哈希表

## 概念

到目前为止，本教程只介绍了有序容器，在其中插入的条目会保持特定次序不变。哈希表是另一类容器，也称为“映射”、“联合数组（associative array）”或者“目录（dictionary）”。

正如语文辞典使用一个定义来关联一个词，哈希表使用一个 键（key） 来唯一标识一个值（value）。哈希表可以根据键非常快速地执行插入、查找和删除操作；实际上，如果使用得当，这些可以都是常数时间 —— 也就是 O(1) —— 操作。这比从一个有序列表中查找或删除条目快得多，那是 O(n) 操作。

哈希表之所以能快速执行操作，是因为它们使用 散列函数 来定位键。散列函数获得一个键并为其计算一个唯一的值，称为 散列值（hash）。例如，一个散列函数可以接受一个词并将那个词中的字母数作为散列值返回。那是个不好的散列函数，因为 “fiddle”和“faddle”将会散列为相同的值。

当散列函数为不同的键返回相同的散列值时，取决于哈希表的实现会发生各种不同的事情。哈希表可以使用第二个值覆盖第一个值，也可以将值放入一个列表，或者或以简单地抛出一个错误。

注意，哈希表不是必然比列表更快。如果拥有的条目较少——少于一打左右——那么使用有序的集合会获得更好的性能。那是因为，尽管在哈希表中存储和获取数据需要常数时间，那个常数时间值可能会很大，因为计算条目的散列值相对于反引 用一两个指针会是一个较慢的过程。对于较小的值，简单地遍历有序列表会比进行散列计算更快。

无论何时，重要的是在选择容器时要考虑自己应用程序的具体数据存储需要。如果应用程序很明显需要某种容器，那么没有理由不去使用它。

## 一些简单的哈希表操作

这里是一些示例，可以生动地展示以上的理论：

```
//ex-ghashtable-1.c
    #include <glib.h>
    int main(int argc, char** argv)
    {
        GHashTable* hash = g_hash_table_new(g_str_hash, g_str_equal);
        g_hash_table_insert(hash, "Virginia", "Richmond");
        g_hash_table_insert(hash, "Texas", "Austin");
        g_hash_table_insert(hash, "Ohio", "Columbus");
```

```
    g_printf("There are %d keys in the hash\n", g_hash_table_size(hash));
    g_printf("The capital of Texas is %s\n", g_hash_table_lookup(hash, "Texas"));
    gboolean found = g_hash_table_remove(hash, "Virginia");
    g_printf("The value 'Virginia' was %sfound and removed\n", found ? "" : "not ");
    g_hash_table_destroy(hash);
    return 0;
}
```

***** Output *****

```
There are 3 keys in the hash
The capital of Texas is Austin
The value 'Virginia' was found and removed
```

**有很多新东西，所以给出一些注解：**

* 对 g_hash_table_new 的调用指定了这个哈希表将使用字符串作为键。函数 g_str_hash 和 g_str_equal 是 GLib 的内置函数，因为这很常用。其他内置 散列/等同（equality） 函数包括 g_int_hash /g_int_equal（使用整数作为键）以及 g_direct_hash/g_direct_equal（使用指针作为键）。

* GLists 和 GSLists 拥有一个 g_[container]_free 函数来清除它们；可以使用 g_hash_table_destroy 来清空 GHashTable。

* 当尝试使用 g_hash_table_remove 删除 键/值 对时，会获得一个 gboolean 返回值，表明键是否找到并删除。gboolean 是 真/假 值的一个简单的跨平台 GLib 实现。

* g_hash_table_size 返回哈希表中键的数目。

# 插入和替换值

当使用 g_hash_table_insert 插入键时，GHashTable 首先检查那个键是否已经存在。如果已经存在，那么那个值会被替换，而键不会被替换。如果希望同时替换键和值，那么需要使用 g_hash_table_replace。它稍有不同，因此在下面同时展示了二者：

```
//ex-ghashtable-2.c
#include <glib.h>
static char* texas_1, *texas_2;
void key_destroyed(gpointer data)
{
    g_printf("Got a key destroy call for %s\n", data == texas_1 ? "texas_1" : "texas_2");
}
int main(int argc, char** argv)
{

    GHashTable* hash = g_hash_table_new_full(g_str_hash,
    g_str_equal,   (GDestroyNotify)key_destroyed, NULL);
```

```
    texas_1 = g_strdup("Texas");
    texas_2 = g_strdup("Texas");
    g_hash_table_insert(hash, texas_1, "Austin");
    g_printf("Calling insert with the texas_2 key\n");
    g_hash_table_insert(hash, texas_2, "Houston");
    g_printf("Calling replace with the texas_2 key\n");
    g_hash_table_replace(hash, texas_2, "Houston");
    g_printf("Destroying hash, so goodbye texas_2\n");
    g_hash_table_destroy(hash);
    g_free(texas_1);
    g_free(texas_2);
    return 0;
}
```

***** Output *****

```
Calling insert with the texas_2 key
Got a key destroy call for texas_2
Calling replace with the texas_2 key
Got a key destroy call for texas_1
Destroying hash, so goodbye texas_2
Got a key destroy call for texas_2
```

从输出可以看到，当 g_hash_table_insert 尝试插入与现有键相同的字符串（Texas）时，GHashTable 只是简单的释放传递进来的键（texas_2），并令当前键（texas_1）保持不变。但是当 g_hash_table_replace 做同样的事情时，texas_1 键被销毁，并在使用它的地方使用 texas_2 键。更多注解：

* 当创建新的 GHashTable 时，可以使用 g_hash_table_full 来提供一个 GDestroyNotify 实现，在键被销毁时调用它。这让您能够为那个键进行完全的资源清除，或者（在本例中）去查看在键变化时实际发生的事情。

* 在前面的 GSList 部分已经出现过 g_strdup；在这里使用它来分配字符串 Texas 的两个拷贝。可以发现，GHashTable 函数 g_str_hash 和 g_str_equal 正确地检测到，尽管指针指向不同的内存位置，但实际上字符串是相同的。为了避免内存泄漏，在函数的末尾必须释放 texas_1 和 texas_2 当然，在本例中这并不重要，因为程序会退出，但是无论如何能够清除是最好的。

## 遍历 键/值 对

有时需要遍历所有的 键/值 对。这里是如何使用 g_hash_table_foreach 来完成那项任务：

```
//ex-ghashtable-3.c
  #include <glib.h>
  void iterator(gpointer key, gpointer value ,gpointer user_data)
```

```
{
  g_printf(user_data, *(gint*)key, value);
}

int main(int argc, char** argv)
{
    GHashTable* hash = g_hash_table_new(g_int_hash, g_int_equal);

    gint* k_one = g_new(gint, 1), *k_two = g_new(gint, 1), *k_three = g_new(gint, 1);
    *   k_one = 1, *k_two=2, *k_three = 3;

    g_hash_table_insert(hash, k_one, "one");
    g_hash_table_insert(hash, k_two, "four");
    g_hash_table_insert(hash, k_three, "nine");

    g_hash_table_foreach(hash, (GHFunc)iterator, "The square of %d is %s   \n");
    g_hash_table_destroy(hash);
    return 0;
}
***** Output *****

    The square of 1 is one
    The square of 2 is four
    The square of 3 is nine
```

在这个示例中有一些细微的不同之处：

* 可以发现，使用 GLib 提供的散列函数 g_int_hash 和 g_int_equal 让您能够使用指向整数的指针作为键。本示例使用的是整数的 GLib 跨平台抽象： gint。
* g_hash_table_foreach 与您已经了解的 g_slist_foreach 和 g_list_foreach 函数非常类似。唯一的区别是，传递到 g_hash_table_foreach 的 GHFunc 要接受三个参数，而不是两个。在本例中，传递进入一个用来格式化键和字符串的打印的字符串作为第三个参数。另外，尽管在本示例时键恰巧是以它们插入的顺序 打印出来，但绝对不保证那个键插入的顺序会被保留。查找条目 //(//((((((((((((((((((((未完全理解))))))))))))))))))))

使用 g_hash_table_find 函数来查找某个特定的值。这个函数支持查看每一个 键/值 对，直到定位到期望的值。这里是一个示例：

```
//ex-ghashtable-4.c
    #include <glib.h>
    void value_destroyed(gpointer data)
    {
        g_printf("Got a value destroy call for %d\n", GPOINTER_TO_INT(data));
    }
```

```
gboolean finder(gpointer key, gpointer value, gpointer user_data)
{
    return (GPOINTER_TO_INT(key) + GPOINTER_TO_INT(value)) == 42;
}


int main(int argc, char** argv)
{
    GHashTable* hash = g_hash_table_new_full(g_direct_hash,
    g_direct_equal,   NULL,   (GDestroyNotify)value_destroyed);

    g_hash_table_insert(hash, GINT_TO_POINTER(6), GINT_TO_POINTER(36));
    g_hash_table_insert(hash, GINT_TO_POINTER(10), GINT_TO_POINTER(12));
    g_hash_table_insert(hash, GINT_TO_POINTER(20), GINT_TO_POINTER(22));

    gpointer item_ptr = g_hash_table_find(hash, (GHRFunc)finder, NULL);
    gint item = GPOINTER_TO_INT(item_ptr);
    g_printf("%d + %d == 42\n", item, 42-item);

    g_hash_table_destroy(hash);
    return 0;
}
```

***** Output *****

```
36 + 6 == 42
Got a value destroy call for 36
Got a value destroy call for 22
Got a value destroy call for 12
```

照例，本示例介绍了 g_hash_table_find 以及其他一些内容：

* GHRFunc 返回 TRUE 时，g_hash_table_find 返回第一个值。如果 GHRFunc 作用于任意条目都都不返回 TRUE（这表明没有找到合适的条目），则它返回 NULL。
* 本示例介绍了另一组内置的 GLib 散列函数：g_direct_hash 和 g_direct_equal。这组函数支持使用指针作为键，但却没有尝试去解释指针背后的数据。由于要将指针放入 GHashTable，所以需要使用一些便利的 GLib 宏（GINT_TO_POINTER 和 GPOINTER_TO_INT）来在整数与指针之间进行转换。
* 最后，本示例创建了 GHashTable，并给予它一个 GDestroyNotify 回调函数，以使得您可以查看条目是何时被销毁的。大部分情况下您会希望在一个与此类似的函数中释放某些内存，不过出于示例的目的，这个实现只是打印出一条消息。

## 难处理的情形：从表中删除

偶尔可能需要从一个 GHashTable 中删除某个条目，但却没有获得 GHashTable 所提供的任意 GDestroyNotify 函数的回调。要完成此任务，或者可以根据具体的键使用 g_hash_table_steal，或者根据所有匹配某个条件的键使用 g_hash_table_foreach_steal。

```
//ex-ghashtable-5.c
#include <glib.h>
gboolean wide_open(gpointer key, gpointer value, gpointer user_data)
{
return TRUE;
}

void key_destroyed(gpointer data)
{
g_printf("Got a GDestroyNotify callback\n");
}

int main(int argc, char** argv)
{
    GHashTable* hash = g_hash_table_new_full(g_str_hash, g_str_equal,
    (GDestroyNotify)key_destroyed,(GDestroyNotify)key_destroyed);

    g_hash_table_insert(hash, "Texas", "Austin");
    g_hash_table_insert(hash, "Virginia", "Richmond");
    g_hash_table_insert(hash, "Ohio", "Columbus");
    g_hash_table_insert(hash, "Oregon", "Salem");
    g_hash_table_insert(hash, "New York", "Albany");

    g_printf("Removing New York, you should see two callbacks\n");
    g_hash_table_remove(hash, "New York");
    if (g_hash_table_steal(hash, "Texas"))
    {
        g_printf("Texas has been stolen, %d items remaining\n",
    g_hash_table_size(hash));
    }
    g_printf("Stealing remaining items\n");
    g_hash_table_foreach_steal(hash, (GHRFunc)wide_open, NULL);
    g_printf("Destroying the GHashTable, but it's empty, so no callbacks\n");
    g_hash_table_destroy(hash);
    return 0;
}
```

***** Output *****

```
Removing New York, you should see two callbacks
Got a GDestroyNotify callback
Got a GDestroyNotify callback
Texas has been stolen, 3 items remaining
Stealing remaining items
Destroying the GHashTable, but it's empty, so no callbacks
```

## 高级查找：找到键和值

针对需要从表中同时获得键和值的情况，GHashTable 提供了一个 g_hash_table_lookup_extended 函数。它与 g_hash_table_lookup 非常类似，但要接受更多两个参数。这些都是“out”参数；也就是说，它们是双重间接指针，当数据被定位时将指向它。这里是它的工作方式：

```
//ex-ghashtable-6.c
#include <glib.h>
int main(int argc, char** argv)
{
    GHashTable* hash = g_hash_table_new(g_str_hash, g_str_equal);

    g_hash_table_insert(hash, "Texas", "Austin");
    g_hash_table_insert(hash, "Virginia", "Richmond");
    g_hash_table_insert(hash, "Ohio", "Columbus");

    char* state = NULL;
    char* capital = NULL;
    char** key_ptr = &state;
    char** value_ptr = &capital;

    gboolean result = g_hash_table_lookup_extended(hash, "Ohio", (gpointer*)key_ptr,
    (gpointer*)value_ptr);
    if (result)
    {
        g_printf("Found that the capital of %s is %s\n", capital, state);
    }
    if (!g_hash_table_lookup_extended(hash, "Vermont", (gpointer*)key_ptr,
    (gpointer*)value_ptr))
    {
        g_printf("Couldn't find Vermont in the hash table\n");
    }

    g_hash_table_destroy(hash);
    return 0;
}
```

***** Output *****

```
Found that the capital of Columbus is Ohio
Couldn't find Vermont in the hash table
```

初始化能够接收 键/值 数据的变量有些复杂，但考虑到它是从函数返回多于一个值的途径，这可以理解。注意，如果您为后两个参数之一传递了 NULL，则 g_hash_table_lookup_extended 仍会工作，只是不是填充 NULL 参数。

## 每个键多个值

到目前为止已经介绍了每个键只拥有一个值的散列。不过有时您需要让一个键持有多个值。当出现这种需求时，使用 GSList 作为值并及 GSList 添加新的值通常是一个好的解决方案。不过，这需要稍多一些工作，如本例中所示：

```
//ex-ghashtable-7.c
#include <glib.h>
void print(gpointer key, gpointer value, gpointer data)
{
    g_printf("Here are some cities in %s: ", key);
    g_slist_foreach((GSList*)value, (GFunc)g_printf, NULL);
    g_printf("\n");
}
void destroy(gpointer key, gpointer value, gpointer data)
{
    g_printf("Freeing a GSList, first item is %s\n", ((GSList*)value)->data);
    g_slist_free(value);
}

int main(int argc, char** argv)
{
    GHashTable* hash = g_hash_table_new(g_str_hash, g_str_equal);
    g_hash_table_insert(hash, "Texas",  g_slist_append(g_hash_table_lookup(hash,
    "Texas"), "Austin "));
    g_hash_table_insert(hash, "Texas",  g_slist_append(g_hash_table_lookup(hash,
    "Texas"), "Houston "));
    g_hash_table_insert(hash, "Virginia",  g_slist_append(g_hash_table_lookup(hash,
    "Virginia"), "Richmond "));
    g_hash_table_insert(hash, "Virginia",  g_slist_append(g_hash_table_lookup(hash,
    "Virginia"), "Keysville "));
    g_hash_table_foreach(hash, print, NULL);
    g_hash_table_foreach(hash, destroy, NULL);
    g_hash_table_destroy(hash);
```

```
    return 0;
}
***** Output *****

    Here are some cities in Texas: Austin Houston
    Here are some cities in Virginia: Richmond Keysville
    Freeing a GSList, first item is Austin
    Freeing a GSList, first item is Richmond
```

g_slist_append 接受 NULL 作为 GSList 的合法参数，示例中的“insert a new city”代码利用了这一事实；它不需要检查这是不是添加到给定州的列表的第一个城市。

当销毁 GHashTable 时，必须记住在释放哈希表本身之前先释放那些 GSList。注意，如果没有在那些列表中使用静态字符串，这会更为复杂；在那种情况下需要在释放列表本身之前先释放每个 GSList 之中的每个条目。这个示例所展示的内容之一是各种 foreach 函数多么实用 —— 它们可以节省很多输入。

## 现实应用

这里是如何使用 GHashTables 的样例。

**在 Gaim 中：**

* gaim-1.2.1/src/buddyicon.c 使用 GHashTable 来保持对“好友图标（buddy icons）”的追踪。键是好友的用户名，值是指向 GaimBuddyIcon 结构体的指针。

* gaim-1.2.1/src/protocols/yahoo/yahoo.c 是这三个应用程序中唯一使用 g_hash_table_steal 的地方。它使用 g_hash_table_steal 作为构建帐号名到好友列表的映射的代码片断的组成部分。

**在 Evolution 中：**

* evolution-2.0.2/smime/gui/certificate-manager.c 使用 GHashTable 来追踪 S/MIME 证书的根源；键是组织名，值是指向 GtkTreeIter 的指针。

* evolution-data-server-1.0.2/calendar/libecal/e-cal.c 使用 GHashTable 来追踪时区；键是时区 ID 字符串，值是某个 icaltimezone 结构体的字符串描述。

**在 GIMP 中：**

* gimp-2.2.4/libgimp/gimp.c 使用 GHashTable 追踪临时的过程。在整个代码基（codebase）中唯一使用 g_hash_table_lookup_extended 的地方，它使用 g_hash_table_lookup_extended 调用来找到某个过程，以使得在删除那个过程之前能首先释放散列键的内存。

* gimp-2.2.4/app/core/gimp.c 使用 GHashTable 来保存图像；键是图像的 ID（一个整数），值是指向 GimpImage 结构体的指针。

# 数组

## 概念

到目前为止我们已经介绍了两类有序集合：GSList 和 GList。它们非常相似，因为都依赖于指针来从一个元素链接到下一个条目，或者，在 GList 中，链接到前一个条目。不过，有另外一类不使用链接的有序集合；它的功能与 C 数组多少有些类似。

它叫做 GArray，提供一个具备索引的单一类型的有序集合，能够为了容纳新条目而增加大小。

相对于链表，数组有什么优势？一方面，索引访问。也就是说，如果想获得数组中的第十五个元素，只需要调用一个能够在常数时间内获取它的函数；不需要手工地遍 历到那个位置，那将是一个 O(n) 操作。数组知道自己的大小，所以查询其大小是一个 O(1) 操作而不是 O(n) 操作。

## 基本操作

这里是向数组添加和删除数据的一些主要方法：

```
//ex-garray-1.c
    #include <glib.h>
    int main(int argc, char** argv)
    {
        GArray* a = g_array_new(FALSE, FALSE, sizeof(char*));
        char* first = "hello", *second = "there", *third = "world";
        g_array_append_val(a, first);
        g_array_append_val(a, second);
        g_array_append_val(a, third);
        g_printf("There are now %d items in the array\n", a->len);
        g_printf("The first item is '%s'\n", g_array_index(a, char*, 0));
        g_printf("The third item is '%s'\n", g_array_index(a, char*, 2));
        g_array_remove_index(a, 1);
        g_printf("There are now %d items in the array\n", a->len);
        g_array_free(a, FALSE);
        return 0;
    }
```

***** Output *****

```
    There are now 3 items in the array
    The first item is 'hello'
    The third item is 'world'
    There are now 2 items in the array
```

## 需要考虑的几点：

* 在创建一个 GArray 时需要考虑一些选项。在上面的示例中，g_array_new 的前两个参数表明了数组是否要以零元素作为终止符，是否数组中的新元素自动设置为零。第三个参数告诉数组它将要保存哪种类型的数据。在这个示例中要创建一 个保存 char* 类型数据的数组；在数组中放入任何其他东西都会导致段错误（segfaults）。
* g_array_append_val 是一个设计不能接受字面值（literal value）的宏，所以不能调用 g_array_append_val(a, 42)。而是那个值需要放入一个变量吕，然后将那个变量传递到 g_array_append_val 中。作为对这种不方便之处的一种慰藉，g_array_append_val 的速度非常快。
* GArray 是具有一个成员变量 len 的结构体，所以为了获得数组的大小，只需要直接引用那个变量；不需要函数调用。
* GArray 的大小以二幂次系数增长。也就是说，如果某个 GArray 包含四个条目，而且您又增加了另一个，那么在内部它会创建另一个拥有八个元素的 GArray，将四个现有元素拷贝到它，然后添加新的元素。这个改变大小的过程需要时间，所以，如果知道将要有大量的元素，那么创建 GArray 时预分配期望的大小会更有效。

## 更多 new/free 选项

本示例中包含创建和销毁 GArray 的一些不同方法：

```
//ex-garray-2.c
    #include <glib.h>
    int main(int argc, char** argv)
    {
        GArray* a = g_array_sized_new(TRUE, TRUE, sizeof(int), 16);
        g_printf("Array preallocation is hidden, so array size == %d\n", a->len);
        g_printf("Array was init'd to zeros, so 3rd item is = %d\n", g_array_index(a, int, 2));
        g_array_free(a, FALSE);

        // this creates an empty array, then resizes it to 16 elements
        a = g_array_new(FALSE, FALSE, sizeof(char));
        g_array_set_size(a, 16);
        g_array_free(a, FALSE);

        a = g_array_new(FALSE, FALSE, sizeof(char));
        char* x = g_strdup("hello world");
        g_array_append_val(a, x);
        g_array_free(a, TRUE);
```

```
    return 0;
}
```

***** Output *****

```
    Array preallocation is hidden, so array size == 0
    Array was init'd to zeros, so 3rd item is = 0
```

注意，由于 GArray 以二幂次系数增长，所有，将数组的大小设置为接近二的某次幂（比如十四）的值会令效率较低。不要那样，而是直接使用最接近的二的某次幂。

## 添加数据的更多方法

到目前为止您已经看到如何使用 g_array_append_val 将数据添加到数组。不过，有其他的方式可以将数据置入数组，如下所示：

```
//ex-garray-3.c
#include <glib.h>
void prt(GArray* a)
{
    g_printf("Array holds: ");
    int i;
    for (i = 0; i < a->len; i++)
       g_printf("%d ", g_array_index(a, int, i));
    g_printf("\n");
}
int main(int argc, char** argv)
{
    GArray* a = g_array_new(FALSE, FALSE, sizeof(int));
    g_printf("Array is empty, so appending some values\n");
    int x[2] = {4,5};
    g_array_append_vals(a, &x, 2);
    prt(a);
    g_printf("Now to prepend some values\n");
    int y[2] = {2,3};
    g_array_prepend_vals(a, &y, 2);
    prt(a);
    g_printf("And one more prepend\n");
    int z = 1;
    g_array_prepend_val(a, z);
    prt(a);
    g_array_free(a, FALSE);
    return 0;
}
```

***** Output *****

Array is empty, so appending some values
Array holds: 4 5
Now to prepend some values
Array holds: 2 3 4 5
And one more prepend
Array holds: 1 2 3 4 5

可以将多个值附加到数组，可以在最前添加一个值，也可以在最前添加多个值。不过，在向最前添加值的时候要小心；它是一个 O(n) 操作，因为 GArray 必须要将当前所有值向后移动，为新的数据腾出空间。在附加或者向最前添加多个值时，还需要使用变量，不过这是相当直观的，因为可以附加或者向最前添加整个 数组。

## 插入数据

也可以向数组中各个不同的位置插入数据；不是限于只能附加或者向最前添加条目。这里是其工作方式：

```
//ex-garray-4.c
#include <glib.h>
void prt(GArray* a)
{
    g_printf("Array holds: ");
    int i;
    for (i = 0; i < a->len; i++)
        g_printf("%d ", g_array_index(a, int, i));
    g_printf("\n");
}
int main(int argc, char** argv)
{
    GArray* a = g_array_new(FALSE, FALSE, sizeof(int));
    int x[2] = {1,5};
    g_array_append_vals(a, &x, 2);
    prt(a);
    g_printf("Inserting a '2'\n");
    int b = 2;
    g_array_insert_val(a, 1, b);
    prt(a);
    g_printf("Inserting multiple values\n");
    int y[2] = {3,4};
    g_array_insert_vals(a, 2, y, 2);
    prt(a);
    g_array_free(a, FALSE);
    return 0;
}
```

***** Output *****

```
Array holds: 1 5
Inserting a '2'
Array holds: 1 2 5
Inserting multiple values
Array holds: 1 2 3 4 5
```

注意，这些插入函数涉及到了向后拷贝列表中当前元素，目的是为新条目准备空间，所以使用 g_array_insert_vals 比反复使用 g_array_insert_val 更好。

## 删除数据

有三种方法可以从 GArray 删除数据：

* g_array_remove_index 和 g_array_remove_range，这两个函数会保持现有顺序
* g_array_remove_index_fast，不保持现有顺序

这里是所有三种方法的示例：

```
//ex-garray-5.c
    #include <glib.h>
    void prt(GArray* a)
    {
        int i;
        g_printf("Array holds: ");
        for (i = 0; i < a->len; i++)
            g_printf("%d ", g_array_index(a, int, i));
        g_printf("\n");
    }
    int main(int argc, char** argv)
    {
        GArray* a = g_array_new(FALSE, FALSE, sizeof(int));
        int x[6] = {1,2,3,4,5,6};
        g_array_append_vals(a, &x, 6);
        prt(a);
        g_printf("Removing the first item\n");
        g_array_remove_index(a, 0);
        prt(a);
        g_printf("Removing the first two items\n");
        g_array_remove_range(a, 0, 2);
        prt(a);
        g_printf("Removing the first item very quickly\n");
        g_array_remove_index_fast(a, 0);
        prt(a);
        g_array_free(a, FALSE);
        return 0;
    }
```

```
***** Output *****
    Array holds: 1 2 3 4 5 6
    Removing the first item
    Array holds: 2 3 4 5 6
    Removing the first two items
    Array holds: 4 5 6
    Removing the first item very quickly
    Array holds: 6 5
```

不只是您可能会对 g_array_remove_fast 的使用情形感到奇怪；三个开放源代码的应用程序都没有使用这个函数。

## 排序

对 GArray 排序很直观；它使用的是在 GList 和 GSList 部分已经出现过的 GCompareFunc：

```
//ex-garray-6.c
#include <glib.h>
void prt(GArray* a)
{
    int i;
    g_printf("Array holds: ");
    for (i = 0; i < a->len; i++)
        g_printf("%d ", g_array_index(a, int, i));
    g_printf("\n");
}
int compare_ints(gpointer a, gpointer b)
{
    int* x = (int*)a;
    int* y = (int*)b;
    return *x - *y;
}
int main(int argc, char** argv)
{
    GArray* a = g_array_new(FALSE, FALSE, sizeof(int));
    int x[6] = {2,1,6,5,4,3};
    g_array_append_vals(a, &x, 6);
    prt(a);
    g_printf("Sorting\n");
    g_array_sort(a, (GCompareFunc)compare_ints);
    prt(a);
    g_array_free(a, FALSE);
```

```
    return 0;
}
```

***** Output *****

```
Array holds: 2 1 6 5 4 3
Sorting
Array holds: 1 2 3 4 5 6
```

注意，比较函数会得到指向数据条目的一个指针，所以在这种情况下，需要将它们强制类型转换为指向正确类型的指针，然后反引用那个指针，得到实际的数据条目。 GArray 还包括另一个排序函数，即 g_array_sort_with_data，它会接受指向另外一段数据的一个指针。

顺便提一句，这三个示例应用程序都没有使用 g_array_sort 和 g_array_sort_with_data。不过，无论如何，知道它们是可用的，都会有所帮助。

## 指针数组

GLib 还提供了 GPtrArray，这是一个为保存指针专门设计的数组。使用它比使用基本的 GArray 更简单，因为在创建或者添加、索引元素时不需要指定具体类型。它与 GArray 非常类似，所以我们将只是回顾基本操作的一些示例：

```
//ex-garray-7.c
#include <glib.h>
#include <stdio.h>
int main(int argc, char** argv)
{
    GPtrArray* a = g_ptr_array_new();
    g_ptr_array_add(a, g_strdup("hello "));
    g_ptr_array_add(a, g_strdup("again "));
    g_ptr_array_add(a, g_strdup("there "));
    g_ptr_array_add(a, g_strdup("world "));
    g_ptr_array_add(a, g_strdup("\n"));
    g_printf(">Here are the GPtrArray contents\n");
    g_ptr_array_foreach(a, (GFunc)g_printf, NULL);
    g_printf(">Removing the third item\n");
    g_ptr_array_remove_index(a, 2);
    g_ptr_array_foreach(a, (GFunc)g_printf, NULL);
    g_printf(">Removing the second and third item\n");
    g_ptr_array_remove_range(a, 1, 2);
    g_ptr_array_foreach(a, (GFunc)g_printf, NULL);
    g_printf("The first item is '%s'\n", g_ptr_array_index(a, 0));
    g_ptr_array_free(a, TRUE);
    return 0;
}
```

***** Output *****

```
>Here are the GPtrArray contents
hello again there world
>Removing the third item
hello again world
>Removing the second and third item
hello
The first item is 'hello '
```

可以了解如何使用只支持指针的数组编写更为直观的 API。这可能能够解释为什么在 Evolution 中使用 g_ptr_array_new 的次数为 178 次，而 g_array_new 只使用了 45 次。大部分情况下只支持指针的数组就足够了!

## 字节数组

GLib 提供了另一个特定类型的数组是 GByteArray。它与您已经了解的类型非常类似，不过有一些区别，因为它是为存储二进制数据而设计的。它非常便于在循环中读取二进制数据，因为它隐 藏了“read into a buffer-resize buffer-read some more”的周期。这里是一些示例代码：

```
//ex-garray-8.c
#include <glib.h>
int main(int argc, char** argv)
{
    GByteArray* a = g_byte_array_new();
    guint8 x = 0xFF;
    g_byte_array_append(a, &x, sizeof(x));
    g_printf("The first byte value (in decimal) is %d\n", a->data[0]);
    x = 0x01;
    g_byte_array_prepend(a, &x, sizeof(x));
    g_printf("After prepending, the first value is %d\n", a->data[0]);
    g_byte_array_remove_index(a, 0);
    g_printf("After removal, the first value is again %d\n", a->data[0]);
    g_byte_array_append(g_byte_array_append(a, &x, sizeof(x)), &x, sizeof(x));
    g_printf("After two appends, array length is %d\n", a->len);
    g_byte_array_free(a, TRUE);
    return 0;
}
```

***** Output *****

```
The first byte value (in decimal) is 255
```

After prepending, the first value is 1
After removal, the first value is again 255
After two appends, array length is 3

还可以发现，在这里使用了一个新的 GLib 类型：guint8。这是跨平台的 8-位 无符号整数，有益于在本例中准确描述字节。

另 外，在这里可以了解到 g_byte_array_append 如何返回 GByteArray。所以，这就使得可以像编写方法链（method chaining）那样将一些附加动作嵌套起来。不过，嵌套多于两个或三个可能并不是一个好办法，除非想让代码变更得 LISP 那样。

## 实际应用

在示例应用程序中使用了各种不同的 GLib 数组类型，虽然不如已经接触的其他容器那样广泛。

Gaim 只使用了 GPtrArrays，而且只在一两种情形下使用到了。gaim-1.2.1/src/gtkpounce.c 使用一个 GPtrArray 来保持对一些 GUI 窗口小部件的追踪，在发生各种事件（比如好友登录进来）时它们会被触发。

Evolution 所使用的大部分是 GPtrArrays，不过也使用了很多 GArrays 和 GByteArrays：

* evolution-2.0.2/widgets/misc/e-filter-bar.h 在 GPtrArrays 中保持一些搜索过滤器的类型。

* evolution-2.0.2/camel/providers/imap4/camel-imap4-store.c 使用 GPtrArray 来追踪 IMAP 文件夹中的条目；它使用 g_ptr_array_sort 以及一个委派给 strcmp 的 GCompareFunc。

GIMP 使用了相当多的 GArray，只使用了很少 GPtrArrays 和 GByteArrays：

* gimp-2.2.4/app/tools/gimptransformtool.c 使用 GArray 来追踪 GimpCoord 实例列表。

* gimp-2.2.4/app/base/boundary.c 中，由点（points）填充起来的 GArray 是极好的 simplify_subdivide 函数的一部分；递归传递的指向 GArray 的双重间接指针是边界简化（boundary simplification）例程的一部分。

## 树

## 概念

树是另一个实用的容器。树中有一个可以拥有子节点的根节点，每个子节点可以有更多子节点，依此类推。

树结构体的示例包括文件系统或者电子邮件客户机；它其中有包含文件夹的文件夹，文件夹中可以有更多文件夹。另外，是否还记得哈希表部分的最后出现的多值示 例？（例如，以字符串作为键，GList 作为值。）由于那些 GLish 值可以容纳更多 GHashTables，那就成为

GHashTable 中构造树结构体的示例。相对于付出努力想使用树一样使用其他容器，使用 GTree 简单得多。

GLib 包括两种树结构：**GTree（平衡二叉树 实现）**以及 **GNode（n-叉 树实现）**。

二叉树的特殊属性是，树中每个节点拥有不超过两个子节点；平衡二叉树 表示元素以特定的次序保持，以进行更快速地搜索。保持元素的平衡意味着删除和插入可能会较慢，因为树本身可能需要进行内部重新平衡，不过寻找某个条目是 O(log n) 操作。
相反，n-叉 树 节点可以有很多子节点。本教程主要关注二叉树，不过也会有一些 n-叉 树的示例。
中序遍历：：左子树－>根－>右子树 eg::DBGEACHFI
前序遍历：：根－>左子树－>右子树 eg::ABDEGCFHI
前序遍历：：左子树－>右子树 －>根 eg::GEBHIFCA

# 树的基本操作

这里是在树中可以执行的一些基本操作：

```
//ex-gtree-1.c
    #include <glib.h>
    int main(int argc, char** argv)
    {
        GTree* t = g_tree_new((GCompareFunc)g_ascii_strcasecmp);
        g_tree_insert(t, "c", "Chicago");
        g_printf("The tree height is %d because there's only one node\n", g_tree_height(t));
        g_tree_insert(t, "b", "Boston");
        g_tree_insert(t, "d", "Detroit");
        g_printf("Height is %d since c is root; b and d are children\n", g_tree_height(t));
        g_printf("There are %d nodes in the tree\n", g_tree_nnodes(t));
        g_tree_remove(t, "d");
        g_printf("After remove(), there are %d nodes in the tree\n", g_tree_nnodes(t));
        g_tree_destroy(t);
        return 0;
    }
    ***** Output *****

    The tree height is 1 because there's only one node
    Height is 2 since c is root; b and d are children
    There are 3 nodes in the tree
    After remove(), there are 2 nodes in the tree
```

关于代码的一些注解：

* 可以看到，GTree 中的每一个节点都包含一个 键-值 对。键用来确保树的平衡、节点插入到适当的位置，并确保值的指针指向应该追踪的 “payload”。
* 必须为 g_tree_new 提供一个 GCompareFunc，以使得 GTree 知道如何对键进行比较。这可以是一个内置函数，如上所示，或者可以自己编写。
* 树“高度（height）”只是从顶到底（包括顶和底）节点的数目。要执行这个函数，GTree 必须从它的根开始向下移动直到到达某个叶子节点。 g_tree_nnodes 更为复杂；它要完全遍历整棵树。

## 替换和提取

在前面的 GHashTable 部分已经看到了 replace 和 steal 函数名，关于 GTree 的函数也是如此。g_tree_replace 会同时替换一个 GTree 条目的键和值，不同于 g_tree_insert，如果要插入的键是重复的，则它只是将值替换。不需要调用任何 GDestroyNotify 函数，g_tree_steal 就可以删除一个节点。这里是一个示例：

```
//ex-gtree-2.c
    #include <glib.h>
    void key_d(gpointer data)
    {
        g_printf("Key %s destroyed\n", data);
    }
    void value_d(gpointer data)
    {
        g_printf("Value %s destroyed\n", data);
    }
    int main(int argc, char** argv)
    {
        GTree* t = g_tree_new_full((GCompareDataFunc)g_ascii_strcasecmp,  NULL,
        (GDestroyNotify)key_d, (GDestroyNotify)value_d);

        g_tree_insert(t, "c", "Chicago");
        g_tree_insert(t, "b", "Boston");
        g_tree_insert(t, "d", "Detroit");

        g_printf(">Replacing 'b', should get destroy callbacks\n");
        g_tree_replace(t, "b", "Billings");

        g_printf(">Stealing 'b', no destroy notifications will occur\n");
        g_tree_steal(t, "b");
        g_printf(">Destroying entire tree now\n");
        g_tree_destroy(t);
        return 0;
    }

    ***** Output *****
```

```
>Replacing 'b', should get destroy callbacks
Value Boston destroyed
Key b destroyed
>Stealing 'b', no destroy notifications will occur
>Destroying entire tree now
Key d destroyed
Value Detroit destroyed
Key c destroyed
Value Chicago destroyed
```

在这个示例中，使用 g_tree_new_full 创建了一个 GTree；与 GHashTable 类似，可以注册键或值的销毁的任意组合的通知。g_tree_new_full 的第二个参数可以包含传递给 GCompareFunc 的数据，不过在此并不需要。

## 查找数据

GTree 具备只查找键或者同时查找键和值的方法。这与在 GHashTable 部分中接触到的非常类似；有一个 lookup 以及一个 lookup_extended。这里是一个示例：

```
//ex-gtree-3.c
#include <glib.h>

int main(int argc, char** argv)
{
    GTree* t = g_tree_new((GCompareFunc)g_ascii_strcasecmp);

    g_tree_insert(t, "c", "Chicago");
    g_tree_insert(t, "b", "Boston");
    g_tree_insert(t, "d", "Detroit");

    g_printf("The data at 'b' is %s\n", g_tree_lookup(t, "b"));
    g_printf("%s\n",   g_tree_lookup(t, "a") ? "My goodness!" : "As expected, couldn't
    find 'a'");

    gpointer* key = NULL;
    gpointer* value = NULL;
    g_tree_lookup_extended(t, "c", (gpointer*)&key, (gpointer*)&value);
    g_printf("The data at '%s' is %s\n", key, value);
    gboolean found = g_tree_lookup_extended(t, "a", (gpointer*)&key,
    (gpointer*)&value);
    g_printf("%s\n", found ? "My goodness!" : "As expected, couldn't find 'a'");

    g_tree_destroy(t);
    return 0;
}
```

***** Output *****

```
The data at 'b' is Boston
As expected, couldn't find 'a'
The data at 'c' is Chicago
As expected, couldn't find 'a'
```

这里再次用到了双重间接指针技术。由于 g_tree_lookup_extended 需要提供多个值，所以它接受两个指向指针的指针（一个指向键，另一个指向值）。注意，如果 g_tree_lookup 找不到键，则它返回一个 NULL gpointer，而如果 g_tree_lookup_extended 不能找到目标，则它返回一个 FALSE gboolean 值。

## 使用 foreach 列出树

GTree 提供了一个 g_tree_foreach 函数，用来以有序的顺序遍历整棵对。这里是一个示例：

```
//ex-gtree-4.c
#include <glib.h>
gboolean iter_all(gpointer key, gpointer value, gpointer data)
{
    g_printf("%s, %s\n", key, value);
    return FALSE;   //返回 FALSE，可以一点不差地遍历整棵树。返回 TRUE 时停止遍历。
}
gboolean iter_some(gpointer key, gpointer value, gpointer data)
{
    g_printf("%s, %s\n", key, value);
    return g_ascii_strcasecmp(key, "b") == 0;
}
int main(int argc, char** argv)
{
    GTree* t = g_tree_new((GCompareFunc)g_ascii_strcasecmp);

    g_tree_insert(t, "d", "Detroit");
    g_tree_insert(t, "a", "Atlanta");
    g_tree_insert(t, "c", "Chicago");
    g_tree_insert(t, "b", "Boston");

    g_printf("Iterating all nodes\n");
    g_tree_foreach(t, (GTraverseFunc)iter_all, NULL);

    g_printf("Iterating some of the nodes\n");
    g_tree_foreach(t, (GTraverseFunc)iter_some, NULL);

    g_tree_destroy(t);
```

```
    return 0;
}
```

***** Output *****

```
    Iterating all nodes
    a, Atlanta
    b, Boston
    c, Chicago
    d, Detroit
    Iterating some of the nodes
    a, Atlanta
    b, Boston
```

注意，当 iter_some 返回 TRUE 时，遍历停止。这就使得 g_tree_foreach 可用来搜索到某个点，累加匹配某个条件前 10 个条目，或者此类事情。当然，令 GTraverseFunc 返回 FALSE，可以一点不差地遍历整棵树。

另外，要注意在使用 g_tree_foreach 遍历树时不应该修改它。

有一个废弃的函数，即 g_tree_traverse，它本意是提供遍历树的另一种方法。例如，可以 后序（post order） 访问节点，也就是自底向上访问树。不过，这个函数从 2001 年起就废弃了，从而 GTree 文档建议在所有使用它的地方替换为使用 g_tree_foreach 或者 n-叉树。这里所研究的开放源代码的应用程序都没有使用它，很不错。

## 搜索

可以使用 g_tree_foreach 搜索条目，如果知道键，可以使用 g_tree_lookup。不过，要进行更复杂地搜索，可以使用 g_tree_search 函数。这里是其工作方式：

```
//ex-gtree-5.c
#include <glib.h>
gint finder(gpointer key, gpointer user_data)
{
    int len = strlen((char*)key);
    if (len == 3) {
        return 0;
    }
    return (len < 3) ? 1 : -1;
    }
int main(int argc, char** argv)
{
    GTree* t = g_tree_new((GCompareFunc)g_ascii_strcasecmp);
    g_tree_insert(t, "dddd", "Detroit");
    g_tree_insert(t, "a", "Annandale");
    g_tree_insert(t, "ccc", "Cleveland");
```

```
    g_tree_insert(t, "bb", "Boston");
    gpointer value = g_tree_search(t, (GCompareFunc)finder, NULL);
    g_printf("Located value %s; its key is 3 characters long\n", value);
    g_tree_destroy(t);
    return 0;
}
```

***** Output *****

```
    Located value Cleveland; its key is 3 characters long
```

注意，传递到 g_tree_search 的 GCompareFunc 实际上决定了搜索如何进行，根据搜索的方式返回 0、1 或者 -1。这个函数甚至可以在搜索进行的时候改变条件；Evolution 在使用 g_tree_search 管理其内存使用时就这样做了。

## 不只是二叉：n-叉 树

GLib n-叉 树实现基于 GNode 数据结构；以前所述，它允许每个父节点有多个子节点。好像很少会用到它，不过，完整起见，这里给出一个用法示例：

```
//ex-gtree-6.c
#include <glib.h>
gboolean iter(GNode* n, gpointer data) {
    g_printf("%s ", n->data);
    return FALSE;
}
int main(int argc, char** argv)
{
    GNode* root = g_node_new("Atlanta");
    g_node_append(root, g_node_new("Detroit"));
    GNode* portland = g_node_prepend(root, g_node_new("Portland"));

    g_printf(">Some cities to start with\n");
    g_node_traverse(root, G_PRE_ORDER, G_TRAVERSE_ALL, -1, iter, NULL);
    g_printf("\n>Inserting Coos Bay before Portland\n");
    g_node_insert_data_before(root, portland, "Coos Bay");
    g_node_traverse(root, G_PRE_ORDER, G_TRAVERSE_ALL, -1, iter, NULL);
    g_printf("\n>Reversing the child nodes\n");
    g_node_reverse_children(root);
    g_node_traverse(root, G_PRE_ORDER, G_TRAVERSE_ALL, -1, iter, NULL);
    g_printf("\n>Root node is %s\n", g_node_get_root(portland)->data);
    g_printf(">Portland node index is %d\n", g_node_child_index(root, "Portland"));
    g_node_destroy(root);
    return 0;
```

```
}

***** Output *****

    >Some cities to start with
    Atlanta Portland Detroit
    >Inserting Coos Bay before Portland
    Atlanta Coos Bay Portland Detroit
    >Reversing the child nodes
    Atlanta Detroit Portland Coos Bay
    >Root node is Atlanta
    >Portland node index is 1
```

可见，GNode 允许将节点放置几乎任何地方；可以在合适时访问它们。这非常灵活，不过它可能会过于灵活，以至于确定其实际使用情形有些困难。实际上，在此所研究的三个开放源代码的应用程序都没有使用它!

## 实际应用

GTree 是一个复杂的结构体，其应用不如我们前面研究其他容器那样广泛。Gaim 根本没有使用它。不过，GIMP 和 Evolution 用到了一些。

**GIMP：**

* gimp-2.2.4/app/menus/plug-in-menus.c 使用 GTree 来保持插件菜单条目。它使用 g_tree_foreach 和一个定制的 GTraverseFunc 来遍历 GTree，向 GimpUIManager 添加插件程序。它使用标准的 C 程序库函数 strcmp 作为自己的 GCompareData 函数。

* gimp-2.2.4/plug-ins/script-fu/script-fu-scripts.c 使用 GTree 来保存“script-fu”脚本。GTree 中的每一个值实际上都是由脚本构成的 GList。

**Evolution：**

Evolution 的 evolution-2.0.2/e-util/e-memory.c 使用 GTree 作为计算未使用内存块的算法的一部分。它使用了一个定制的 GCompareFunc，即 tree_compare，来对_cleaninfo 结构体进行排序，这些结构体指向的是空闲的内存块。

## 队列

## 概念

队列是另一个便利的数据结构。一个 队列 会保存一列条目，而且访问形式通常是向最后添加条目，从最前删除条目。当需要按到达顺序进行处理时，这很有实用。标准队列的一个变种是“双端队列 （double-ended queue）”，或者说是 dequeue，它支持在队列的两端进行添加或者删除。

不过，在很多情况下最好避免使用队列。队列搜索不是特别快（是 O(n) 操作），所以，如果需要经常进行搜索，那么哈希表或者树可能更实用。这同样适用于需要访问队列中随机元素的情形；如果是那样，那么将会对队列进行很多次线性扫描。

GLib 提供了一个使用 GQueue 的 dequeue 实现；它支持标准队列操作。它的基础是双向链表（GList），所以它也支持很多其他操作，比如在队列之中进行插入和删除。不过，如果您发现自己经常要使 用这些功能，那么可能需要重新考虑容器的选择；或许另一个容器更为合适。

## 基本操作

这里是以“排队买票（ticket line）”为模型的一些基本的 GQueue 操作：

```
//ex-gqueue-1.c
    #include <glib.h>
    int main(int argc, char** argv)
    {
        GQueue* q = g_queue_new();
        g_printf("Is the queue empty?   %s, adding folks\n",   g_queue_is_empty(q) ? "Yes" :
        "No");
        g_queue_push_tail(q, "Alice");
        g_queue_push_tail(q, "Bob");
        g_queue_push_tail(q, "Fred");
        g_printf("First in line is %s\n", g_queue_peek_head(q));
        g_printf("Last in line is %s\n", g_queue_peek_tail(q));
        g_printf("The queue is %d people long\n",   g_queue_get_length(q));
        g_printf("%s just bought a ticket\n", g_queue_pop_head(q));
        g_printf("Now %s is first in line\n", g_queue_peek_head(q));
        g_printf("Someone's cutting to the front of the line\n");
        g_queue_push_head(q, "Big Jim");
        g_printf("Now %s is first in line\n", g_queue_peek_head(q));
        g_queue_free(q);
        return 0;
    }

    ***** Output *****
        Is the queue empty?   Yes, adding folks
        First in line is Alice
        Last in line is Fred
        The queue is 3 people long
        Alice just bought a ticket
        Now Bob is first in line
        Someone's cutting to the front of the line
        Now Big Jim is first in line
```

大部分方法名称都是完全自我描述的，不过有一些更细致之处：

* 向队列压入和取出条目的各种操作不返回任何内容，所以，为了使用队列，您需要保持 g_queue_new 返回的指针。

* 队列的两端都可以用于添加和删除。如果要模拟排队买票时排在后面的人离开转到另一个队列去购买，也是完全可行的。

* 有非破坏性的 peek 操作可以检查队列头或尾的条目。

* g_queue_free 不接受帮助释放每个条目的函数，所以需要手工去完成；这与 GSList 相同。

## 删除和插入条目

虽然通常只通过在队列的末端 添加/删除 条目来修改它，但 GQueue 允许删除任意条目以及在任意位置插入条目。这里是其示例：

```
//ex-gqueue-2.c
    #include <glib.h>
    int main(int argc, char** argv)
    {
        GQueue* q = g_queue_new();
        g_queue_push_tail(q, "Alice");
        g_queue_push_tail(q, "Bob");
        g_queue_push_tail(q, "Fred");
        g_printf("Queue is Alice, Bob, and Fred; removing Bob\n");
        int fred_pos = g_queue_index(q, "Fred");
        g_queue_remove(q, "Bob");
        g_printf("Fred moved from %d to %d\n", fred_pos, g_queue_index(q, "Fred"));
        g_printf("Bill is cutting in line\n");
        GList* fred_ptr = g_queue_peek_tail_link(q);
        g_queue_insert_before(q, fred_ptr, "Bill");
        g_printf("Middle person is now %s\n", g_queue_peek_nth(q, 1));
        g_printf("%s is still at the end\n", g_queue_peek_tail(q));
        g_queue_free(q);
        return 0;
    }

    ***** Output *****

        Queue is Alice, Bob, and Fred; removing Bob
        Fred moved from 2 to 1
        Bill is cutting in line
        Middle person is now Bill
        Fred is still at the end
```

有很多新函数：

* g_queue_index 在队列中扫描某个条目并返回其索引；如果它不能找到那个条目，则返回 -1。

* 为了向队列的中间插入一个新条目，需要一个指向希望插入位置的指针。如您所见，通过调用一个“peek link”函数，就可以进行此处理；这些函数包括：g_queue_peek_tail_link、g_queue_peek_head_link 以及 g_queue_peek_nth_link，它们会返回一个 GList。然后可以将一个条目插入到 GList 之前或者之后。

* g_queue_remove 允许从队列中的任何位置删除某个条目。继续使用“排队买票”模型，这表示人们可以离开队列；他们组成队列后并不固定在其中。

## 查找条目

在先前的示例中已经看到，在拥有一个指向条目数据的指针或者知道其索引的条件下如何去得到它。不过，类似其他 GLib 容器， GQueue 也包括一些查找函数：g_queue_find 和 g_queue_find_custom：

```
//ex-gqueue-3.c
    #include <glib.h>
    gint finder(gpointer a, gpointer b) {
        return strcmp(a,b);
    }
    int main(int argc, char** argv)
    {
        GQueue* q = g_queue_new();
        g_queue_push_tail(q, "Alice");
        g_queue_push_tail(q, "Bob");
        g_queue_push_tail(q, "Fred");
        g_queue_push_tail(q, "Jim");
        GList* fred_link = g_queue_find(q, "Fred");
        g_printf("The fred node indeed contains %s\n", fred_link->data);
        GList* joe_link = g_queue_find(q, "Joe");
        g_printf("Finding 'Joe' yields a %s link\n", joe_link ? "good" : "null");
        GList* bob = g_queue_find_custom(q, "Bob", (GCompareFunc)finder);
        g_printf("Custom finder found %s\n", bob->data);
        bob = g_queue_find_custom(q, "Bob", (GCompareFunc)g_ascii_strcasecmp);
        g_printf("g_ascii_strcasecmp also found %s\n", bob->data);
        g_queue_free(q);
        return 0;
    }

    ***** Output *****

        The fred node indeed contains Fred
        Finding 'Joe' yields a null link
        Custom finder found Bob
        g_ascii_strcasecmp also found Bob
```

注意，如果 g_queue_find 找不到条目，则它会返回 null。并且可以在上面的示例中传递一个库函数（比如 g_ascii_strcasecmp）或者一个定制的函数（比如 finder）作为 g_queue_find_custom 的 GCompareFunc 参数。

## 使用队列：拷贝、反转和遍历每一个（foreach）

由于 GQueue 的基础是 GList，所以它支持一些列表处理操作。这里是如何使用 g_queue_copy、 g_queue_reverse 和 g_queue_foreach 的示例：

```
//ex-gqueue-4.c
#include <glib.h>
int main(int argc, char** argv)
{
    GQueue* q = g_queue_new();
    g_queue_push_tail(q, "Alice ");
    g_queue_push_tail(q, "Bob ");
    g_queue_push_tail(q, "Fred ");
    g_printf("Starting out, the queue is: ");
    g_queue_foreach(q, (GFunc)g_printf, NULL);
    g_queue_reverse(q);
    g_printf("\nAfter reversal, it's: ");
    g_queue_foreach(q, (GFunc)g_printf, NULL);
    GQueue* new_q = g_queue_copy(q);
    g_queue_reverse(new_q);
    g_printf("\nNewly copied and re-reversed queue is: ");
    g_queue_foreach(new_q, (GFunc)g_printf, NULL);
    g_queue_free(q);
    g_queue_free(new_q);
    return 0;
}

***** Output *****

    Starting out, the queue is: Alice Bob Fred
    After reversal, it's: Fred Bob Alice
    Newly copied and re-reversed queue is: Alice Bob Fred
```

g_queue_reverse 和 g_queue_foreach 很直观；您已经看到它们在各种其他有序集合中得到了应用。不过，使用 g_queue_copy 时需要稍加留心，因为拷贝的是指针而不是数据。所以，当释放数据时，一定不要进行重复释放。

## 使用链接的更多乐趣

已经了解了链接的一些示例；这里是一些便利的链接删除函数。不要忘记 GQueue 中的每个条目实际上是都是一个 GList 结构体，数据存储在“data”成员中：

//ex-gqueue-5.c

```
#include <glib.h>
int main(int argc, char** argv)
{
    GQueue* q = g_queue_new();
    g_queue_push_tail(q, "Alice ");
    g_queue_push_tail(q, "Bob ");
    g_queue_push_tail(q, "Fred ");
    g_queue_push_tail(q, "Jim ");
    g_printf("Starting out, the queue is: ");
    g_queue_foreach(q, (GFunc)g_printf, NULL);
    GList* fred_link = g_queue_peek_nth_link(q, 2);
    g_printf("\nThe link at index 2 contains %s\n", fred_link->data);
    g_queue_unlink(q, fred_link);
    g_list_free(fred_link);
    GList* jim_link = g_queue_peek_nth_link(q, 2);
    g_printf("Now index 2 contains %s\n", jim_link->data);
    g_queue_delete_link(q, jim_link);
    g_printf("Now the queue is: ");
    g_queue_foreach(q, (GFunc)g_printf, NULL);
    g_queue_free(q);
    return 0;
}
***** Output *****
    Starting out, the queue is: Alice Bob Fred Jim
    The link at index 2 contains Fred
    Now index 2 contains Jim
    Now the queue is: Alice Bob
```

注意，g_queue_unlink 并不释放没有被链接的 GList 结构体，所以需要自己去完成。并且，由于它是一个 GList 结构体，所以需要使用 g_list_free 函数来释放它 —— 而不是简单的 g_free 函数。当然，更简单的是调用 g_queue_delete_link 并让它为您释放内存。

## 排序

队列排序好像不太常见，不过由于各种其他链表操作都得到了支持（比如 insert 和 remove），所以此操作也得到了支持。如果恰巧您希望重新对队列进行排序，将高优先级的条目移动到前端，那么这也会很便利。这里是一个示例：

//ex-gqueue-6.c

```
#include <glib.h>
```

```
typedef struct {
    char* name;
    int priority;
} Task;
Task* make_task(char* name, int priority)
{
    Task* t = g_new(Task, 1);
    t->name = name;
    t->priority = priority;
    return t;
}
void prt(gpointer item)
{
    g_printf("%s  ", ((Task*)item)->name);
}
gint sorter(gconstpointer a, gconstpointer b, gpointer data)
{
    return ((Task*)a)->priority - ((Task*)b)->priority;
}
int main(int argc, char** argv)
{
    GQueue* q = g_queue_new();
    g_queue_push_tail(q, make_task("Reboot server", 2));
    g_queue_push_tail(q, make_task("Pull cable", 2));
    g_queue_push_tail(q, make_task("Nethack", 1));
    g_queue_push_tail(q, make_task("New monitor", 3));
    g_printf("Original queue: ");
    g_queue_foreach(q, (GFunc)prt, NULL);
    g_queue_sort(q, (GCompareDataFunc)sorter, NULL);
    g_printf("\nSorted queue: ");
    g_queue_foreach(q, (GFunc)prt, NULL);
    g_queue_free(q);
    return 0;
}
```

***** Output *****

```
Original queue: Reboot server  Pull cable  Nethack  New monitor
Sorted queue: Nethack  Reboot server  Pull cable  New monitor
```

现在您就拥有了一个模拟您的工作的 GQueue，偶尔还可以对它进行排序，可以欣喜地发现，Nethack 被提升到了其正确的位置，到了队列的最前端!

## 实际应用

GQueue 没有在 Evolution 中得到应用，但是 GIMP 和 Gaim 用到了它。

**GIMP：**

* gimp-2.2.4/app/core/gimpimage-contiguous-region.c 在一个查找相邻片段的工具函数中使用 GQueue 存储一系列坐标。只要片段保存邻接，新的点就会被压入到队列末端，然后在下一个循环迭代中取出并被检查。

* gimp-2.2.4/app/vectors/gimpvectors-import.c 使用 GQueue 作为 Scalable Vector Graphics（SVG）解析器的一部分。它被当做栈使用，条目的压入和取出都在队列的头上进行。

**Gaim：**

* gaim-1.2.1/src/protocols/msn/switchboard.c 使用 GQueue 来追踪发出的消息。新的消息压入到队列的尾部，当发送后从头部取出。

* gaim-1.2.1/src/proxy.c 使用 GQueue 追踪 DNS 查找请求。它使用队列作为应用程序代码与 DNS 子进程之间的临时保存区域。

# 关系

# 概念

GRelation 类似一张简单的数据库表；它包含一系列记录，或者 元组（tuples），每一个包含某干个域。每个元组必须拥有相同数目的域，可以为任意的域指定索引，以支持对那个域进行查找。

作为示例，可以使用一系列元组来保存名字，一个域中保存名，第二个域中保存姓。两个域都可以被索引，以使得使用名或者姓都可以进行快速查找。

GRelation 有一个缺点，那就是每个元组最多只能包含两个域。因此，不能将它作为内存中的数据库表缓存，除非表中列非常少。我在 gtk-app-devel-list 邮件列表中搜索关于此问题的注解，发现早在 2000 年 2 月讨论到了一个补丁，它可以将此扩展到四个域，但好像它从来没有加入到发行版本中。

GRelation 好像是一个鲜为人知的结构体；本教程中研究的开放源代码的应用程序当前都没有使用它。在 Web 上浏览时发现了一个开放源代码的电子邮件客户机（Sylpheed-claws），出于各种不同目的使用了它，包括追踪 IMAP 文件夹和消息线程。所有它需要的可能只是一些宣传!

# 基本操作

这里是一个示例，创建一个具有两个索引域的新的 GRelation，然后插入一些记录并执行一些基本的信息查询：

```
//ex-grelation-1.c
    #include <glib.h>
    int main(int argc, char** argv)
    {
        GRelation* r = g_relation_new(2);
        g_relation_index(r, 0, g_str_hash, g_str_equal);
        g_relation_index(r, 1, g_str_hash, g_str_equal);
```

```
    g_relation_insert(r, "Virginia", "Richmond");
    g_relation_insert(r, "New Jersey", "Trenton");
    g_relation_insert(r, "New York", "Albany");
    g_relation_insert(r, "Virginia", "Farmville");
    g_relation_insert(r, "Wisconsin", "Madison");
    g_relation_insert(r, "Virginia", "Keysville");
    gboolean found = g_relation_exists(r, "New York", "Albany");
    g_printf("New York %s found in the relation\n", found ? "was" : "was not");
    gint count = g_relation_count(r, "Virginia", 0);
    g_printf("Virginia appears in the relation %d times\n", count);
    g_relation_destroy(r);
    return 0;
}
***** Output *****
    New York was found in the relation
    Virginia appears in the relation 3 times
```

注意，索引恰好是在调用 g_relation_new 之后而在调用 g_relation_insert 之前添加的。这是因为 g_relation_count 等其他 GRelation 函数要依赖现有的索引，如果索引不存在，则在运行时会出错。

上面的代码中包括一个 g_relation_exists，用来查看“New York”是否在 GRelation 中。这个请求会精确匹配关系中的每一个域；可以在任意一个索引的域上使用 g_relation_count 进行匹配。

在前面的 GHashTable 部分已经接触过 g_str_hash 和 g_str_equal；在这里使用它们来对 GRelation 中的索引域进行快速查找。

## 选择元组

数据存入 GRelation 中后，可以使用 g_relation_select 函数来取出它。结果是一个指向 GTuples 结构体的指针，通过它进一步查询可以获得实际的数据。这里是它的使用方法：

```
//ex-grelation-2.c
#include <glib.h>
int main(int argc, char** argv)
{
    GRelation* r = g_relation_new(2);
    g_relation_index(r, 0, g_str_hash, g_str_equal);
    g_relation_index(r, 1, g_str_hash, g_str_equal);
    g_relation_insert(r, "Virginia", "Richmond");
    g_relation_insert(r, "New Jersey", "Trenton");
    g_relation_insert(r, "New York", "Albany");
    g_relation_insert(r, "Virginia", "Farmville");
    g_relation_insert(r, "Wisconsin", "Madison");
    g_relation_insert(r, "Virginia", "Keysville");
    GTuples* t = g_relation_select(r, "Virginia", 0);
    g_printf("Some cities in Virginia:\n");
```

```
    int i;
    for (i=0; i < t->len; i++) {
        g_printf("%d) %s\n", i, g_tuples_index(t, i, 1));
    }
    g_tuples_destroy(t);
    t = g_relation_select(r, "Vermont", 0);
    g_printf("Number of Vermont cities in the GRelation: %d\n", t->len);
    g_tuples_destroy(t);
    g_relation_destroy(r);
    return 0;
}
***** Output *****
    Some cities in Virginia:
    0) Farmville
    1) Keysville
    2) Richmond
    Number of Vermont cities in the GRelation: 0
```

## 关于选择和遍历元组的一些注解：

* g_relation_select 返回的 GTuples 结构体中的记录没有特定的次序。要找出返回了多少记录，请使用 GTuple 结构体中的 len 成员。

* g_tuples_index 接受三个参数：
    - o GTuple 结构体
    - o 正在查询的记录的索引
    - o 希望获得的域的索引

* 注意，需要调用 g_tuples_destroy 来正确地释放在 g_relation_select 期间所分配的内存。就算是记录实际上并没有被 GTuples 对象引用，这也是有效的。

## 总结

结束语：

在本教程中，研究了如何使用 GLib 程序库中的数据结构。研究了可以如何使用这些容器来有效地管理程序的数据，还研究了在几个流行的开放源代码项目中这些容器如何得到应用。在此过程中介绍了很多 GLib 类型、宏以及字符串处理函数。

GLib 包括很多其他的优秀功能：它有一个线程-抽象（threading-abstraction）层，一个可移植-套接字（portable- sockets）层，消息日志工具，日期和时间函数，文件工具，随机数生成，等等，还有很多。值得去研究这些模块。并且，如果有兴趣且有能力，您甚至可以 改进某些文档 —— 例如，记法扫描器的文档中包含了一个注释，内容是它需要一些示例代码，并需要进一步详述。如果您从开放源代码的代码中受益，那么请不要忘记帮助改进它!