第二部分　Linux编程常用C语言函数库及构件库

第3章　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。

3.1　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）。

3.2　glib的宏

3.2.1　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)
```

3.2.2 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。

3.3 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-profile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dmalloc.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)
```

3.4 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 %", 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_strfreev()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)
```

3.5 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。

3.5.1 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API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position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data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list1和list2两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list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)
```

3.5.2 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，但是要使用整数，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parent的position处插入节点node*/
GNode* g_node_insert(GNode* parent,
            gint position,
            GNode* node)
/*在父节点parent中的sibling节点之前插入节点node*/
GNode* g_node_insert_before(GNode* parent,
                  GNode* sibling,
                  GNode* node)
/*在父节点parent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。 GNodeTraverseFunc返回TRUE，停止任何正在进行的遍历，这样就能将 GnodeTraverseFunc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func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date的节点*/
GNode* g_node_find_child(GNode* node,
                GTraverseFlags flags,
                gpointer data)
/*返回子节点child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的第一个兄弟节点*/
GNode* g_node_last_sibling(GNode* node)
```

3.5.3 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)
```

3.6 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 string's current \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 init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val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val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sprintf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,
                         ... );
```

3.7 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）。使用它的第一步是用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 );
```

3.8 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。

3.9 其他实用函数

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“ /tmp ”*/
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。