

简约设计の艺术


一个自由程序员的琐碎

目录视图

摘要视图

RSS 订阅


个人资料



88250

+ 加关注

发私信



访问: 4197312次

积分: 41304分

排名: 第21名

原创: 1195篇

转载: 326篇

译文: 42篇

评论: 2834条

文章搜索

文章分类

Adoration (7)

Agile Develeopment (25)

Alipay (2)

Android (2)

Apache (2)

Architecture Design (18)

B3log (31)

Beyond (2)

BeyondTrack (8)

Book (1)

BPEL (7)

Buzz (2)

C# &.Net (36)

C/C++ (60)

Cache (4)

CAS & SAML & SSO (7)

Chinasb (1)

Chrome (1)

Code Name:i0y0l (4)

Compile Principles (8)

Data-Structrue/Algorithms (41)

Database (31)

Design Patterns (22)

DHTML (27)

Eclipse (36)

EJB 3.x (24)

CSDN博客第三方发布工具有奖调查

专访成晓旭: 云计算催生行业信息化新格局

感知计算: 开发人员的十大资源

CSDN博客第二期云计算最佳博主评选

2013年10月微软MVP申请开始

原

glib常用库函数和一些定义

分类: C/C++

2008-01-31 16:31

1880人阅读

评论(2)

收藏

举报

list

string

timer

tree

null

table

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。

使用glib库的程序都应该包含glib的头文件glib.h。

glib基本类型定义:

整数类型:

gint8、guint8、gint16、guint16、gint32、guint32、gint64、guint64。

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。

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。

指针类型:

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)

glib的宏

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)

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。

English	(4)
EverBox	(1)
Fiddlededee	(87)
FireFox	(2)
FreeMarker	(1)
GAE	(32)
Game	(4)
GFW	(1)
Git	(5)
GlassFish	(5)
Go	(4)
Google	(19)
Guice	(1)
Hibernate Framework	(30)
HTML5	(2)
HttpClient	(1)
IoC/DI	(2)
J2EE/JavaEE	(123)
J2SE/JavaSE	(174)
Java	(63)
Java Persistence API	(25)
Java Server Faces	(40)
JavaEE	(14)
JavaEE Security	(3)
JavaFX	(25)
JavaScript	(7)
JBoss Seam	(26)
jBPM	(8)
JDK 7	(3)
Joke	(3)
JPA	(2)
jQuery	(1)
jsdoc	(1)
jsoup	(3)
JSR-299	(21)
JSR-330	(2)
Life in Programming	(153)
LivaPlayer	(14)
Mathematics	(9)
Maven	(8)
Maven 2	(18)
Memcached	(1)
MultiMedia	(17)
Music	(16)
My Linux	(165)
NetBeans	(225)
Network Engineering	(63)
node.js	(1)
Open Source	(261)
Optimization	(1)
Oracle	(6)
ORM	(4)
OSGi	(18)
PHP	(9)
Portal & Portlet	(8)
Python	(1)
Quartz	(1)
Regular Expression	(16)
Ruby & Rails	(19)
SCA	(1)
Shell Programming	(15)
Software Engineering	(42)

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。

在某些情况下，需要使用中间类型转换。

```
////////////////////////////////////
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:

定义了G_DISABLE_CHECKS或G_DISABLE_ASSERT之后，编译时它们就会消失。

宏列表:

前提条件检查

```
#include <glib.h>

g_return_if_fail ( condition )
g_return_val_if_fail(condition, retval)
```

```
////////////////////////////////////
```

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->nodelist[i]; node; node = node->next)

            (* func) (node->key, node->value, user_data);
}
```

```
////////////////////////////////////
```

宏列表:

断言

```
#include <glib.h>

g_assert( condition )
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。

断言与前提条件检查的区别:

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。

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。

Software Quality Assurance (2)

Software Testing (15)

Spring Framework (25)

StoneAgeDict (28)

Struts Framework (3)

Subversion (10)

SWT/JFace/RCP (25)

SyntaxHighlighter (1)

System Analyst exam (13)

Tencent (1)

TestNG (2)

TeX/LaTeX (9)

Text Categorization (9)

Ubuntu (3)

UML Modeling (12)

VI/VIM (1)

Web (4)

Web Browser (2)

Web Service (5)

Web UI Design (23)

Wicket (5)

Windows (54)

Wine (1)

Workflow & BPM (17)

にほんごのべんきょう (4)

li (0)

B3log Announcement (3)

B3log Solo (1)

文章存档

2013年05月 (1)

2012年11月 (1)

2012年08月 (1)

2012年02月 (1)

2011年10月 (1)

展开

阅读排行

当代 IT 大牛排行榜 (438455)

NetBeans 时事通讯 (刊) (294752)

Seam 2.1.1.GA 发布! (240256)

JBoss Seam 框架下的单 (220039)

20个漂亮xp桌面主题 (72099)

了解 NoSQL 的必读资料 (65769)

初学UML之-----用例图 (64610)

数学符号大全 (38918)

Web Beans (JSR-299): (30463)

《星际争霸》成为大学课 (26971)

评论排行

十二个理由让你不得不欺 (106)

2009 年个人回忆与总结 (91)

Linux下的千千静听——L (69)

20个漂亮xp桌面主题 (49)

初学UML之-----用例图 (48)

书评：简洁代码——敏捷转 (44)

程序员的幽默 (43)

准备入职支付宝 (39)

HTML 5 WebSocket 示例 (38)

NetBeans IDE 6.9 Beta (38)

```
////////////////////////////////////
```

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.

```
##### 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。

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)
```

g_realloc()和realloc()是等价的。

g_malloc0(), 它将分配的内存每一位都设置为0;

推荐文章

最新评论

GAE Java 应用性能优化
wilder2000: mark

基于 JSF + Spring + JPA 构建敏捷I
az690236414: 把源码发给我还
吗? 690236414@qq.com谢谢了,
最好是我能加你好友, 好相互交流

数学符号大全
低调小一: 多谢了, 转载了, 楼主辛
苦

java.util.logging日志功能使用快速,
ccssddnnbbookkee: 很详细, 谢谢

使用 CAS 在 Tomcat 中实现单点登
hooqee: 不错!

哪本书是对程序员最有影响、每个科
起飞--为梦想而飞: 我一本都没看
过, 需要好好看看

Single SignOn - Integrating Liferay
logoc: 咱能, 一天到晚不转来转去
吗。还他妈的专家

暂时告别 CSDN 博客, 移居 GAE
bubble: 顶b3blog 对搜索引擎优化
方面做的如何

加入中国 HTML5 研究小组
簡約_Billy: 想您致敬,

Java 开源博客——B3log Solo 0.5.
簡約_Billy:

Code snips

Java Code examples

HTML代码示例

C 代码示例

E-books

中国 IT 实验室

中文电子书网

网络中国 - E 书

CSDN 下载频道

偶要雷锋 - 分享社区

Linux/Ubuntu

Gnome-Look

Ubuntu 中文官方论坛

deviantART Search

GetDeb

KDE-Look

LinuxToy

Compiz Themes

ChinaUnix

Compiz-Fusion

My friends

光光的Blog~

ZY Tough

师傅 dorainm

Eleven 的专栏

Meteor 的专栏

Vanessa 的小窝

秋歌的专栏

金秋风采

eleven-china

野地的枯草

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。

g_memdup()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。

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

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)
```

在含有sprintf()的平台上, g_snprintf()封装了一个本地的sprintf(), 并且比原有实现更稳定、安全。

以往的sprintf()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,“...”是要插入的参数。

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)
宏g_strstrip()结合以上两个函数, 删除字符串前后的空格。
```

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)
```

函数列表: 分配字符串

```
#include <glib.h>

gchar * g_strdup(const gchar* str)
gchar* g_strdup(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)
////////////////////////////////////
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: 连接字符串的函数

```
#include <glib.h>

gchar* g_strconcat(const gchar* string1,... )
```

blog.csdn.net/DL88250/article/details/2075352

4/18

狼猫窝

云南科软

88250 @ Solo (RSS)

My projects

BeyondTrack @ Java.net

LivaPlayer

Drop

B3log Solo

B3log Latke

Super stars :-)

Don Knuth's Home Page

Martin Fowler

Alan Turing

Uncle Bob (Robert C. Martin)

Bjarne Stroustrup's Homepage

Richard Stallman's Home Page

Technologies

IBM 软件技术

CSDN

UML 官方

Eclipse.org

Apache Software

LEX & YACC Page

Java 开源大全

JBoss.org

PHP 官方

Springframework.org

NetBeans 中文社区

SourceForge.net

JavaWorld@TW

hibernate.org

Extreme Programming

Ruby on Rails

Ruby 中文社区论坛

JavaFX Script Reference

JavaFX Home

Open Source Initiative

Facelets DevDoc

Testng.org

java-source

JavaEye

NetBeans Dream Team

InfoQ

OpenEBS

JBoss Seam

J 道

HTML 4.01 Spec

歇歇脚

HTTP 1.1 Status Code

开源中国社区

HTML5 研究小组

```
gchar* g_strjoin(const gchar* separator,...)

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 #####

链表~~~~~

glib提供了普通的单向链表和双向链表, 分别是GSList 和GList。

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);

删除上面添加的元素并清空链表:

list = g_slist_remove(list, element);

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;

g_slist_free()只释放链表的单元, 它并不知道怎样操作链表内容。

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"));
```

```
////////////////////////////////////
```

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)
```

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)
```

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。

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~~~~~

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。

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)
```

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)
```

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 )
```

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;
};
```

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)
```

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)
```


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)
```

函数列表: 销毁GNode

```
#include <glib.h>

void g_node_destroy(GNode* root)
void g_node_unlink(GNode* node)
```

宏列表: 判断G n o d e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G N o d e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)
```

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data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)

```

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);
```

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。

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)
```

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)
```

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 )

```

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 #####
```

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。

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_sprintf( GString *string,gchar *fmt,... );
```

```
##### 计时器函数 #####
```

```

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。
使用g_strerror函数:
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。

##### 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_prpname (void);
/* 设置应用程序的名称 */
void g_set_prpname (const gchar *prpname);
/* 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、TMPandTEMP 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。

如果了解其他内容，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提供许多有用的函数及定义。我它们列在这里并做简短的解释。很多都是与libc重复，对这些我不再详述。这些大致上是用来参考，您知道有什么东西可以用就好。

17.1 定义

为保持资料型态的一致，这里有一些定义：

```
G_MINFLOAT
G_MAXFLOAT
G_MINDOUBLE
G_MAXDOUBLE
G_MINSHORT
G_MAXSHORT
G_MININT
G_MAXINT
G_MINLONG
G_MAXLONG
```

此外，以下的typedefs。没有列出来的是会变的，要看是在那一种平台上。如果您想要具有可移植性，记得避免去使用sizeof(pointer)。例如，一个指标在Alpha上是8 bytes，但在Inter上为4 bytes。

```
char      gchar;
short     gshort;
long      glong;
int       gint;
char      gboolean;

unsigned  char      gchar;
unsigned  short     gushort;
unsigned  long      gulong;
unsigned  int       guint;

float     gfloat;
double    gdouble;
long double gldouble;
```

```
void* gpointer;
```

```
gint8
```

```
guint8
```

```
gint16
```

```
guint16
```

```
gint32
```

```
guint32
```

17.2 双向链结串列

以下函数用来产生，管理及销毁双向链结串列.

```
GList* g_list_alloc (void);
```

```
void g_list_free (GList *list);
```

```
void g_list_free_1 (GList *list);
```

```
GList* g_list_append (GList *list,
                      gpointer data);
```

```
GList* g_list_prepend (GList *list,
                      gpointer data);
```

```
GList* g_list_insert (GList *list,
                      gpointer data,
                      gint position);
```

```
GList* g_list_remove (GList *list,
                      gpointer data);
```

```
GList* g_list_remove_link (GList *list,
                           GList *link);
```

```
GList* g_list_reverse (GList *list);
```

```
GList* g_list_nth (GList *list,
                  gint n);
```

```
GList* g_list_find (GList *list,
                   gpointer data);
```

```
GList* g_list_last (GList *list);
```

```
GList* g_list_first (GList *list);
```

```
gint g_list_length (GList *list);
```

```
void g_list_foreach (GList *list,
```

GFunc func,
gpointer user_data);

17.3 单向链结串列
以下函数是用来管理单向链结串列:

```
GSList* g_slist_alloc      (void);

void      g_slist_free      (GSList      *list);

void      g_slist_free_1    (GSList      *list);

GSList* g_slist_append      (GSList      *list,
                             gpointer      data);

GSList* g_slist_prepend     (GSList      *list,
                             gpointer      data);

GSList* g_slist_insert      (GSList      *list,
                             gpointer      data,
                             gint          position);

GSList* g_slist_remove      (GSList      *list,
                             gpointer      data);

GSList* g_slist_remove_link (GSList      *list,
                             GSList      *link);

GSList* g_slist_reverse     (GSList      *list);

GSList* g_slist_nth         (GSList      *list,
                             gint          n);

GSList* g_slist_find        (GSList      *list,
                             gpointer      data);

GSList* g_slist_last        (GSList      *list);

gint      g_slist_length    (GSList      *list);

void      g_slist_foreach   (GSList      *list,
                             GFunc        func,
                             gpointer      user_data);
```

17.4 记忆体管理

```
gpointer g_malloc      (gulong      size);
```

这是替代malloc()用的。你不需要去检查返回值，因为它已经帮你做好了，保证。

```
gpointer g_malloc0 (gulong size);
```

一样，不过会在返回之前将记忆体归零。

```
gpointer g_realloc (gpointer mem,
                  gulong size);
```

重定记忆体大小。

```
void g_free (gpointer mem);
```

```
void g_mem_profile (void);
```

将记忆体的使用状况写到一个档案，不过您必须要在glib/gmem.c里面，加#define MEM_PROFILE，然後重新编译。

```
void g_mem_check (gpointer mem);
```

检查记忆体位置是否有效。您必须要在glib/gmem.c上加#define MEM_CHECK，然後重新编译。

17.5 Timers

Timer函数..

```
GTimer* g_timer_new (void);
```

```
void g_timer_destroy (GTimer *timer);
```

```
void g_timer_start (GTimer *timer);
```

```
void g_timer_stop (GTimer *timer);
```

```
void g_timer_reset (GTimer *timer);
```

```
gdouble g_timer_elapsed (GTimer *timer,
                        gulong *microseconds);
```

17.6 字符串处理

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

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

```
GString* g_string_assign (GString *lval,
```



```
gchar *rval);

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

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

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

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

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

void g_string_sprintf (GString *string,
                       gchar *fmt,
                       ...);

void g_string_sprintfa (GString *string,
                        gchar *fmt,
                        ...);
```

17.7 工具及除错函数

```
gchar* g_strdup (const gchar *str);

gchar* g_strerror (gint errnum);
```

我建议您使用这个来做所有错误讯息。 这玩意好多了。 它比perror()来的具有可移植性。 输出为以下形式:

```
program name:function that failed:file or further description:strerror
```

这里是"hello world"用到的一些函数:

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

```
void g_error (gchar *format, ...);
```

显示错误讯息, 其格式与printf一样, 但会加个"*** ERROR **: ", 然後离开程式. 只在严重错误时使用.

```
void g_warning (gchar *format, ...);
```

跟上面一样, 但加个"*** WARNING **: ", 不离开程式.

```
void g_message (gchar *format, ...);
```

加个"message: ".

```
void g_print      (gchar *format, ...);
```

printf()的替代品.

最後一个:

```
gchar* g_strsignal (gint signum);
```

列印Unix系统的信号名称, 在信号处理时很有用.


这些大都从glib.h中而来.

分享到:  

上一篇: [Linux平台下的JNI开发\[88250原创\]](#)


下一篇: [基于 JSF + Spring + JPA 构建敏捷的Web应用\[88250原创\]](#)

查看评论

1楼 [yunsn0303](#) 2011-04-05 23:59发表 



哎 无数资料都是从学长你这里看到的 惭愧啊[e03]

Re: [88250](#) 2011-04-06 01:15发表 



回复 yunsn0303: :-)

您还没有登录,请[\[登录\]](#)或[\[注册\]](#)

* 以上用户言论只代表其个人观点, 不代表CSDN网站的观点或立场

[公司简介](#) | [招贤纳士](#) | [广告服务](#) | [银行汇款帐号](#) | [联系方式](#) | [版权声明](#) | [法律顾问](#) | [问题报告](#)



QQ客服



微博客服



论坛反馈



联系邮箱: webmaster@csdn.net



服务热线: 400-600-2320

京 ICP 证 070598 号

北京创新乐知信息技术有限公司 版权所有

世纪乐知(北京)网络技术有限公司 提供技术支持

江苏乐知网络技术有限公司 提供商务支持

Copyright © 1999-2012, CSDN.NET, All Rights Reserved

