GLib 学习笔记

目录

1 基本类型

出于易用与可移植性等方面的考虑，GLib 定义了一些常用的数据类型。

1.1 标准 C 所缺乏的数据类型

事实上，GLib 所定义的这部分数据类型，多数在 C99 中已有定义，但是它们均非标准 C 的内建类型，通常是在各种 C 标准库的头文件定义（并且不同的 C 标准库所提供的头文件名可能会不相同）。GLib 之所以提供相应的替代类型，主要是将 C 标准库隐匿，向用户提供一组风格较为一致的数据类型。

C 语言没有提供布尔类型，GLib 基于 int 类型定义了 gboolean 类型，并定义了 FALSE 与 TRUE 宏：

```
1  #ifndef FALSE
2  #define FALSE (0)
3  #endif
4
5  #ifndef TRUE
6  #define TRUE (!FALSE)
7  #endif
```

为了增强 C 程序的可移植性，C 语言标准提供了 size_t 类型，一种无符号整型变量，适于计量存储空间中可容纳的数据项目个数，在数组下标和内存管理函数之类的地方广泛使用，例如 sizeof 操作符的返回结果即为 size_t 类型。size_t 类型的存储宽度是平台依赖的，例如在 32 位环境中，其宽度为 32 位，而在 64 位环境中，其宽度则为 64 位。size_t 的有符号变体是 ssize_t。GLib 定义了 gsize 与 gssize 类型，分别等同于 size_t 与 ssize_t。GLib 的这种克隆行为显得有点多余，但事实上，size_t 与 ssize_t 一般是在 C 标准库的头文件中定义，程序中使用时通常需要：

```
1  #include <stdlib.h>
```

为此 GLib 不辞劳苦，让我们有了一点小解脱。gsize 与 gssize 的取值范围分别为 [0, G_MAXSIZE] 与 [G_MINSSIZE, G_MAXSSIZE]。

C99 中提供了 off64_t 类型，是 64 位宽的整型类型，用于表示文件偏移量，主要用于大文件的数据存取。像对待 size_t 那样，GLib 也提供了 off64_t 的克隆，即 goffset 类型，并定义了相应取值范围 [G_MINOFFSET, G_MAXOFFSET]。

很多情况下，C 程序员会将指针类型的变量强制转换为整型变量进行一些运算（多用于需要精确控制数据在内存中的精确布局）。在 32 位平台上，由于指针类型的宽度和 int 相同，所以指针类型与 int 类型的转换问题通常不受重视，但是在 64 位平台上，由于目前几乎所有的 64 位操作系统均采用 LP64 模型，即 int 的宽度是 32 位，而指针类型的宽度为 64 位，此时指针类型与 int 类型的转换便会存在隐患。为此，C99 中提供了 intptr_t 数据类型，以保证 C 程序跨平台移植的安全性。intptr_t 是有符号的整型类型，其无符号变体为 uintptr_t，GLib 提供的替代类型分别为 gintptr 与 guintptr，并且提供了相应的输入/输出格式 G_GINTPTR_MODIFIER 与 G_GINTPTR_FORMAT，例如：

```
1  #include <glib.h>
2  ... ...
3  gchar *s = "hello world!";
4  gintptr p = (gintptr)s;
5  g_print ("%#" G_GINTPTR_MODIFIER "x\n", p);
6  g_print ("%" G_GINTPTR_FORMAT "\n", p);
7  ... ...
```

记住，对于使用 GLib 库的程序，gcc 的编译命令为：

```
$ gcc `pkg-config --cflags --libs glit-2.0` foo.c -o foo
```

1.2 跨平台等宽整型类型

这部分数据类型是一组固定宽度的整型类型，有 g[u]int[8, 16, 32, 64][1]，GLib 所做的工作就是在它支持的所有平台上保证它们的宽度一致。

设 $x \in \{8, 16, 32, 64\}$，gint$x$ 的取值范围为 [G_MININTx, G_MAXINTx]，guint$x$ 的取值范围为 [0, G_MAXUINTx]；g[u]int$x$ 的输入/输出格式为 G_GINTx_[MODIFIER, FORMAT]。

1.3 标准 C 数据类型封装

这部分数据类型均为标准 C 的内建类型，GLib 采用封装的形式进行了些许简化。

首先，C 的无类型指针 void * 与指向常量数据的无类型指针 const void * 分别被 GLib 定义为 gpointer 与 gconstpointer。

[1] 我喜欢的简写方式，可解读为：gint8, guint8, gint16, guint16, gint32, guint32, gint64, guint64。

然后，C 的无符号字符、短整型、整型与长整型，即 unsigned [char, short, int, long] 被 GLib 定义为 gu[char, short, int, long]。

1.4 还有一些画蛇添足的类型

为了保持与前三部分数据类型编码风格的一致，GLib 行为艺术般的将标准 C 内建的数据类型 $\{x|x \in$ char, int, short, long, float, double 重新取名为 gx，并且界定它们的取值范围为 [-G_MAXX, G_MAXX]，$X \in$ CHAR, INT, SHORT, LONG, FLOAT, DOUBLE。

1.5 数值宏

GLib 对数学计算中经常用到的几个数值提供了宏定义。有些 C 标准库提供了相关宏定义，但也有未提供的。GLib 能做到的就是它可以保证它会一直提供这些宏。

G_E 表示自然基底 e。

G_PI, G_PI_2, G_PI_4 分别表示 $\pi, 2\pi, 4\pi$。

G_LN2 与 G_LN10 分别表示 $\ln 2$ 与 $\ln 10$。

G_LOG_2_BASE_10 表示 $\log_{10} 2$ 。

G_SQRT2 表示 $\sqrt{2}$。

此外，GLib 还提供了 IEEE754 单/双精度浮点数移码的宏定义，并提供便于访问 IEEE754 格式单/双精度浮点数的符号、阶码与尾数的 union 类型，例如：

```
1  GFloatIEEE754 a = {3.1415};
2  g_print ("%u\n", a.mpn.sign);
3  g_print ("%u\n", a.mpn.biased_exponent);
4  g_print ("%u\n", a.mpn.mantissa);
```

1.6 想知道更多

GLib 基本类型的详细说明见 GLib 参考手册的“GLib Fundamentals”部分以及 glib/gtypes.h 文件。

GLib 的基本类型的外延部分还提供了整型类型与指针类型的原子操作，由于我的程序环境不涉及多线程编程问题，所以此部分内容被我故意忽略了，若需要了解这部分内容，可查阅 GLib 参考手册的“Atomic Operations”部分。

2 主事件循环

GUI 应用程序大都是事件驱动的。事件可以来自于用户输入，比如键盘、鼠标事件，也可以来自于系统内部，比如定时器事件、套接字事件和文件事件等。在没有任何事件的情况下，应用程序处于睡眠状态。因此，GUI 应用程序需要主事件循环 (Main eventl oop) 的支持，主事件循环可以控制 GUI 应用程序什么时候进入睡眠状态，什么时候被唤醒去工作。GLib 便提供了一套这样的主事件循环机制。

2.1 GMainLoop、GMainContext 与 GSource

GLib 提供了 GMainLoop 类型，用于表示主事件循环。GMainLoop 的创建与使用可以参考下面的代码，它可以生成一个长眠不死的程序。

```
1  #include <glib.h>
2
3  int
4  main (void)
5  {
6          GMainLoop *main_loop = g_main_loop_new (NULL, TRUE);
7
8          g_main_loop_run (main_loop);
9          g_main_loop_unref(main_loop);
10
11         return 0;
12 }
```

显然，g_main_loop_new 函数用于构建一个 GMainLoop 类型的变量，为其分配内存并进行初始化，而 g_main_loop_run 函数可以根据 GMainLoop 变量中所记录的一些状态信息来控制程序睡眠抑或工作。

g_main_loop_new 函数的第一个参数是主事件循环环境 (GMainContext)，第二个参数用于设置主事件循环的运行状态。这两个参数均与 GMainLoop 类型的数据结构有关。GMainLoop 的数据结构很简单，其主体部分是主事件循环环境，此外是运行状态与引用计数[2]，具体如下：

```
typedef struct _GMainLoop GMainLoop;
struct _GMainLoop {
        GMainContext *context;
        gboolean is_running;
        gint ref_count;
};
```

如果 g_main_loop_new 函数第一个参数为 NULL，那么在 g_main_loop_new 函数内部会为 GMainLoop 分配默认的主事件循环环境。主事件循环建立之后，可使用 g_main_loop_get_context 函数获取主事件循环环境。

主事件循环环境内部包含事件源 (GSource) 列表。GSource 是事件的抽象表示。对于任何事件，只要实现 GSource 定义的接口，便可将其挂至 GMainContext 的事件源列表中，从而可在 g_main_loop_run 函数中响应该事件。

2.2 DIY 一个简单的事件源

自定义的事件源是一个继承 GSource 的结构体，即自定义事件源的结构体的第一个成员是 GSource 结构体，其后便可放置程序所需数据，例如：

```
typedef struct _MySource MySource;
struct _MySource {
        GSource _source;
        gchar text[256];
}
```

实现了事件源数据结构的定义之后，还需要实现事件源所规定的接口，主要表现为 prepare, check, dispatch, finalize 等事件处理函数（回调函数），它们包含于 GSourceFuncs 结构体中。

将 GSourceFuncs 结构以及事件源结构的存储空间宽度作为参数传给 g_source_new 便可构造一个新的事件源，继而可使用 g_source_attach 函数将新的事件源添加到主循环环境中。

下面这个示例可创建一个只会讲“Hello world!”的事件源，并将其添加到主事件循环默认的 GMainContext 中。

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

typedef struct _MySource MySource;
struct _MySource {
        GSource source;
```

[2] g_main_loop_unref 函数便是根据 GMainLoop 的引用计数是否为 0 来决定是否释放 GMainLoop 的存储空间。

```
7           gchar text[256];
8   };
9
10  static gboolean
11  prepare (GSource *source, gint *timeout)
12  {
13          *timeout = 0;
14          return TRUE;
15  }
16
17  static gboolean
18  check (GSource *source)
19  {
20          return TRUE;
21  }
22
23  static gboolean
24  dispatch (GSource *source, GSourceFunc callback, gpointer user_data)
25  {
26          MySource *mysource = (MySource *)source;
27          g_print ("%s\n", mysource->text);
28
29          return TRUE;
30  }
31
32  int
33  main (void)
34  {
35          GMainLoop *loop = g_main_loop_new (NULL, TRUE);
36          GMainContext *context = g_main_loop_get_context (loop);
37
38          GSourceFuncs source_funcs = {prepare, check,
39                                       dispatch, finalize};
40          GSource *source = g_source_new (&source_funcs,
41                                          sizeof(MySource));
42
43          g_sprintf (((MySource *)source)->text, "Hello world!");
44          g_source_attach (source, context);
45          g_source_unref (source);
46
47          g_main_loop_run (loop);
48
49          g_main_context_unref (context);
50          g_main_loop_unref (loop);
51
52          return 0;
53  }
```

上述程序的 g_main_loop_run 函数运行时，会迭代访问 GMainContext 的事件源列表，步骤大致如下：

a. g_main_loop_run 通过调用事件源的 prepare 接口[3] 并判断其返回值以确定各事件源是否作好准备。如果各事件源的 prepare 接口的返回值为 TRUE，即表示该事件源已经作好准备，否则表示尚未做好准备。显然，上述程序所定义的事件源是已经作好了准备。

b. 若某事件源尚未作好准备，那么 g_main_loop 会在处理完那些已经准备好的事件后再次询问该事件源是否作好准备，这一过程是通过调用事件源的 check 接口而实现的，如果事件源依然未作好准备，即 check 接口的返回 FALSE，那么 g_main_loop_run 会让主事件循环进入睡眠状态。主事件循环的睡眠时间是步骤 a 中遍历时间源时所统计的最小时间间隔，例如在 prepare 接口中可以像下面这样设置时间间隔。到达一定时间后，g_main_loop_run 会唤醒主事件循环，再次询问。如此周而复始，直至事件源的 prepare 接口返回值为 TRUE。

```
1  static gboolean
2  prepare (GSource *source, gint *timeout)
3  {
4          *timeout = 1000; /* set time interval one second */
5          return TRUE;
6  }
```

c. 若事件源 prepare 与 check 函数返回值均为 TRUE，则 g_main_loop_run 会调用事件源的 dispatch 接口，由该接口调用事件源的响应函数。事件源的响应函数是回调函数，可使用 g_source_set_callback 函数进行设定。在上例中，我们没有为自定义的事件源提供响应函数。

事实上，GLib 的主循环机制对事件源的处理比上述过程还要复杂，下文将继续进行探索。

2.3 事件源的类型

GLib 的事件源有三种类型：空闲（idle）类型、定时（timeout）类型与文件类型。

上一节我们自定义的事件源 MySource 实际上就是空闲类型。所谓空闲类型的事件源，是指那些只有在主事件循环无其他事件源处理时才会被处理的事件源。GLib 提供了预定义的空闲事件源类型，其用法见下面的示例。

```
1  #include <glib.h>
2
```

[3] 实际上是间接调用，即 g_main_loop_run 调用 g_main_context_iteration，后者会逐个调用事件源的 prepare 接口。

```
3   static gboolean
4   idle_func (gpointer data)
5   {
6           g_print ("%s\n", (gchar *)data);
7
8           return TRUE;
9   }
10
11  int
12  main (void)
13  {
14          GMainLoop *loop = g_main_loop_new (NULL, TRUE);
15          GMainContext *context = g_main_loop_get_context (loop);
16
17          g_idle_add (idle_func, "Hello world!");
18          g_main_loop_run (loop);
19
20          g_main_context_unref (context);
21          g_main_loop_unref (loop);
22
23          return 0;
24  }
```

这个示例与上一节那个示例的运行结果是相同的。上述示例中，idle_func 是 idle 事件源的响应函数，如果该函数返回值为 TRUE，那么它会在主事件循环空闲时重复被执行；如果 idle_func 的返回值为 FALSE，那么该函数在执行一次后，便被主事件循环从事件源中移除。

如果在上述示例的 idle_func 函数中调用 g_main_loop_quit 函数，那么可以使主事件循环在处理了一次 idle 事件源之后便退出。例如：

```
1   #include <glib.h>
2
3   static gboolean
4   idle_func (gpointer data)
5   {
6           g_print ("Just once!\n");
7           g_main_loop_quit (data);
8
9           return TRUE;
10  }
11
12  int
13  main (void)
14  {
15          GMainLoop *loop = g_main_loop_new (NULL, TRUE);
16          GMainContext *context = g_main_loop_get_context (loop);
17
18          g_idle_add (idle_func, loop);
```

```
19          g_main_loop_run (loop);
20
21          g_main_context_unref (context);
22          g_main_loop_unref (loop);
23
24          return 0;
25  }
```

这个程序运行时，它会说一句“Just once!”然后就退出了。

g_idle_add 函数内部定义了一个空闲事件源，并将用户定义的回调函数设为空闲事件源的响应函数，然后将该事件源挂到主循环环境。

第二类事件源是定时器。GLib 也提供了预定义的定时器事件源，其用法与 GLib 预定义的空闲事件源类似。例如：

```
1   #include <glib.h>
2
3   static gboolean
4   timeout_func (gpointer data)
5   {
6           static guint i = 0;
7           i += 2;
8           g_print ("%d\n", i);
9
10          return TRUE;
11  }
12
13  int
14  main (void)
15  {
16          GMainLoop *loop = g_main_loop_new (NULL, TRUE);
17          GMainContext *context = g_main_loop_get_context (loop);
18
19          g_timeout_add (2000, timeout_func, loop);
20          g_main_loop_run (loop);
21
22          g_main_context_unref (context);
23          g_main_loop_unref (loop);
24
25          return 0;
26  }
```

如果要自定义定时器类型的事件源，只需让事件源的 prepare 与 check 接口在时间超过所设定的时间间隔时返回 TRUE，否则返回 FALSE。

文件类型的事件源要稍微难理解一些，因为涉及到了操作系统层面的 poll 机制。所谓 poll 机制，就是操作系统提供的对文件描述符所关联的文件的状态监视功能，例如向文件中写入数据，那么文件的状态可以表示为 POLLOUT，而从文件中读取数据，那么文件的状态就变为 POLLIN。GLib 为 Unix 系统与

Windows 系统的 poll 机制进行了封装，并且可以将文件与主事件循环的事件源建立关联，在主循环的过程中，g_main_loop_run 会轮询各个关联到文件的事件源，并处理相应的事件响应。

文件类型的事件源，其 prepare，check，dispatch 等接口的执行次序如下：

- 主事件循环会首先调用 check 接口，询问事件源是否准备好。因为此时，g_main_loop_run 尚未轮询那些与文件相关联的事件源，所以文件类型的事件源，其 check 接口的返回值应该是 FALSE。
- 主事件循环调用 g_main_context_iteration 轮询各事件源，探寻是否有文件类型事件源的状态发生变化，并记录变化结果。
- 主循环调用 check 接口，询问事件是否准备好。此时，如果文件类型事件源的状态变化符合要求，那么就返回 TRUEE，否则返回 FALSE。
- 如果 prepare 与 chec k 接口的返回值均为 TRUE，那么此时主事件循环会调用 dispatch 接口分发消息。

下面的示例展示了一个自定义的文件类型事件源的基本用法。该示例所产生的程序接受用户在终端中输入的字符串，并统计输入的字符数量。

```
1  #include <glib.h>
2
3  typedef struct _MySource MySource;
4  struct _MySource {
5          GSource _source;
6          GIOChannel *channel;
7          GPollFD fd;
8  };
9
10 static gboolean
11 watch (GIOChannel *channel)
12 {
13         gsize len = 0;
14         gchar *buffer = NULL;
15
16         g_io_channel_read_line (channel, &buffer, &len, NULL, NULL);
17         if (len > 0)
18                 g_print ("%d\n", len);
19         g_free (buffer);
20
21         return TRUE;
22 }
23
24 static gboolean
25 prepare (GSource *source, gint *timeout)
26 {
27         *timeout = -1;
28         return FALSE;
29 }
```

```
30
31  static gboolean
32  check (GSource *source)
33  {
34          MySource *mysource = (MySource *)source;
35          if (mysource->fd.revents != mysource->fd.events)
36                  return FALSE;
37          return TRUE;
38  }
39
40  static gboolean
41  dispatch (GSource *source, GSourceFunc callback, gpointer user_data)
42  {
43          MySource *mysource = (MySource *)source;
44          if (callback)
45                  callback (mysource->channel);
46
47          return TRUE;
48  }
49
50  static void
51  finalize (GSource *source)
52  {
53          MySource *mysource = (MySource *)source;
54
55          if (mysource->channel)
56                  g_io_channel_unref (mysource->channel);
57  }
58
59  int
60  main(int argc, char* argv[])
61  {
62          GMainLoop *loop = g_main_loop_new (NULL, FALSE);
63
64          GSourceFuncs funcs = {prepare, check, dispatch, finalize};
65          GSource *source = g_source_new (&funcs, sizeof(MySource));
66          MySource *mysource = (MySource *) source;
67
68          mysource->channel = g_io_channel_new_file ("test", "r", NULL);
69          mysource->fd.fd = g_io_channel_unix_get_fd (mysource->channel);
70          mysource->fd.events = G_IO_IN;
71          g_source_add_poll (source, &mysource->fd);
72          g_source_set_callback (source, (GSourceFunc)watch, NULL, NULL);
73          g_source_set_priority (source, G_PRIORITY_DEFAULT_IDLE);
74          g_source_attach (source, NULL);
75          g_source_unref (source);
76
77          g_main_loop_run(loop);
78
```

```
79          g_main_context_unref (g_main_loop_get_context (loop));
80          g_main_loop_unref(loop);
81
82          return 0;
83  }
```

像空闲类型与计时器类型事件源那样，GLib 也提供了预定义的文件类型事件源，使用它可以将上例简化为：

```
1   #include <glib.h>
2
3   gboolean
4   io_watch (GIOChannel *channel, GIOCondition condition, gpointer data)
5   {
6           gsize len = 0;
7           gchar *buffer = NULL;
8
9           g_io_channel_read_line (channel, &buffer, &len, NULL, NULL);
10
11          if (len > 0)
12                  g_print ("%d\n", len);
13
14          g_free (buffer);
15
16          return TRUE;
17  }
18
19  int
20  main(int argc, char* argv[])
21  {
22          GMainLoop *loop = g_main_loop_new (NULL, FALSE);
23
24          GIOChannel* channel = g_io_channel_unix_new (1);
25          if (channel){
26                  g_io_add_watch(channel, G_IO_IN, io_watch, NULL);
27                  g_io_channel_unref(channel);
28          }
29
30          g_main_loop_run(loop);
31
32          g_main_context_unref (g_main_loop_get_context (loop));
33          g_main_loop_unref(loop);
34
35          return 0;
36  }
```

2.4 想知道更多

建议阅读 GLib Reference 的“The Main Event Loop”以及“IO Channels”部分。

2.5 补充

退出主事件循环之后，调用 g_main_loop_unref 只能够释放主事件循环的数据结构所占用的存储空间，主事件循环环境的数据结构所占用的存储空间需要手动释放。我是通过一些小示例的验证才有了这一认识，GLib 文档对此没有什么特别的说明。

3 线程

线程在行为上类似进程，但与进程不同的是，一个进程的所有线程处于同一内存空间之中。这没什么不好，通过共享的内存空间，线程之间的通信会很简单，但是这也会很糟，程序如果设计的不够好，会出现海森堡 bug（嗯，海森堡测不准原理），这样的问题都是拜线程并发性质之所赐。不过，程序员可以通过一些同步原语迫使线程按照一定的次序执行。

GLib 对各个平台的线程进行了封装，努力消除多线程程序的跨平台障碍。GLib 提供了一些互斥锁(Mutex)原语用于保护内存区域的访问，提供了一些条件变量原语用于控制线程同步，提供了线程私有数据原语，可以让线程可以沦落为小资，此外就是提供了线程创建与管理原语，以满足跨平台移植的需求。

3.1 简单的示例

在说更多的废话之前，还是先看代码：

```
1  #include <glib.h>
2
3  static gpointer
4  thread_1 (gpointer data)
5  {
6          for (int i = 0; i < 3; i++)
                   g_print ("I'm thread_1\n");
7
8          return NULL;
9  }
10
11 static gpointer
12 thread_2 (gpointer data)
13 {
14         for (int i = 0; i < 3; i++)
                   g_print ("I'm thread_2\n");
15
16         return NULL;
17 }
18
```

```
19  int
20  main (void)
21  {
22  /* Only run the test, if threads are enabled
23     and a default thread implementation is available */
24  #if defined(G_THREADS_ENABLED) && !defined(G_THREADS_IMPL_NONE)
25          g_thread_init (NULL);
26
27          g_thread_create (thread_1, NULL, TRUE, NULL);
28          g_thread_create (thread_2, NULL, TRUE, NULL);
29
30          for (int i = 0; i < 3; i++)
                    g_print ("I'm main thread\n");
31  #endif
32          return 0;
33  }
```

编译上述代码的命令如下：

```
gcc `pkg-config --cflags --libs glib-2.0 gthread-2.0` \
    -std=c99 test.c -o test
```

程序的运行结果有可能是：

```
I'm thread_2
I'm main thread
I'm thread_2
I'm thread_2
I'm main thread
I'm thread_1
I'm main thread
I'm thread_1
```

之所以说“可能是”，是因为这个程序包含着三个线程，即主线程和两个子线程。这三个线程究竟谁先被执行，这是未知数。这就是线程的并发性质所导致的结果，即便主线程也不例外。

现在，观看上面的线程示例代码。

首先，第 24 与 25 行代码可以完成 GLib 线程环境的初始化工作。虽然在 GLib 程序中，不一定非要在 gthread 线程函数调用之前调用线程初始化函数，但是许多位于 GLib 库之上的库不支持这种随意性，为了让大家都开心，尽量早调用。只有在执行了 g_thread_init 函数之后，GLib 才是线程安全的。

第 27 与 28 行代码创建了两个线程，也就是说主线程创建了两个子线程，此时这个进程便有了三个线程。这三个线程等待操作系统的选择，通常是谁先被选中那么就先被执行。如果将主线程中的 for 循环代码删除，那么程序也许会什么信息也不输出便终止了。这是因为主线程可能会比它的两个子线程结束的早，导致进程终止，因此那两个子线程英年早逝。

下面，我们将上例的主线程代码稍微改动一下，让主线程等待它的两个子线程结束后再继续执行，如下：

```
1   int
2   main (void)
3   {
4   #if defined(G_THREADS_ENABLED) && !defined(G_THREADS_IMPL_NONE)
5           g_thread_init (NULL);
6
7           GThread *t1 = g_thread_create (thread_1, NULL, TRUE, NULL);
8           GThread *t2 = g_thread_create (thread_2, NULL, TRUE, NULL);
9
10          g_thread_join (t1);
11          g_thread_join (t2);
12  #endif
13          return 0;
14  }
```

如果一个线程创建了另一个线程，那么前者可以通过 g_thread_join 函数将后者设置为被等待者，而它自身则扮演着等待者的脚色。当然，前提是被等待者要接受这份等待，只需将 g_thread_create 函数的第三个参数设为 TRUE 即可。g_thread_join 函数的返回值即为被等待者的返回值。

3.2 互斥

多个线程可能会同时访问进程空间中的同一块数据区域，这样会导致有的线程所读去的数据并不是它所期望的，因为它在读取的时候，数据可能会被其它线程篡改。例如：

```
1   #include <glib.h>
2
3   static gpointer
4   thread_1 (gpointer data)
5   {
6           gint * p = (gint *)data;
7
8           for (int i = 0; i < 4; i++){
                    (*p)++;
9                   g_print ("thread_1 said %d\n", *p);
10          }
11
12          return NULL;
13  }
14
15  static gpointer
16  thread_2 (gpointer data)
17  {
```

```
18          gint *p = (gint *)data;
19
20          for (int i = 0; i < 4; i++){
                    (*p)++;
21                  g_print ("thread_2 said %d\n", *p);
22          }
23
24          return NULL;
25  }
26
27  int
28  main (void)
29  {
30  #if defined(G_THREADS_ENABLED) && !defined(G_THREADS_IMPL_NONE)
31          g_thread_init (NULL);
32
33          gint i = 0;
34
35          GThread *t1 = g_thread_create (thread_1, &i, TRUE, NULL);
36          GThread *t2 = g_thread_create (thread_2, &i, TRUE, NULL);
37
38          g_thread_join (t1);
39          g_thread_join (t2);
40  #endif
41          return 0;
42  }
```

上例按理说，其输出结果不管是线程 thread_1 与 thread_2 谁先被执行，它们输出的数值应该是依次递增的。事实并非如此，可能的输出如下：

```
thread_1 said 2
thread_2 said 1
thread_1 said 3
thread_2 said 4
thread_1 said 5
thread_2 said 6
thread_1 said 7
thread_2 said 8
```

之所以会出现上述情况，是因为 thread_2 将 i 修改为 1 后，正准备输出其数值，结果 thread_1 冲上来把 i 的值篡改为 2 并输出，然后 thread_2 继续输出值为 1 的 i。

那么，有没有办法可以保证当某个线程在访问数据时而不被其他线程所干扰呢？操作系统为线程提供了“数据锁”，学名叫“互斥”，意思就是可以对进程中的数据空间进行锁定，例如：

```
1   #include <glib.h>
2
```

```
3   struct mutex_int_t {
4           GMutex *mutex;
5           gint i;
6   };
7
8   static gpointer
9   thread_1 (gpointer data)
10  {
11          struct mutex_int_t * p = (struct mutex_int_t *)data;
12
13          g_mutex_lock (p->mutex);
14          for (int i = 0; i < 4; i++){
                    p->i++;
15                  g_print ("thread_1 said %d\n", p->i);
16          }
17          g_mutex_unlock (p->mutex);
18
19          return NULL;
20  }
21
22  static gpointer
23  thread_2 (gpointer data)
24  {
25          struct mutex_int_t * p = (struct mutex_int_t *)data;
26
27          g_mutex_lock (p->mutex);
28          for (int i = 0; i < 4; i++){
                    p->i++;
29                  g_print ("thread_2 said %d\n", p->i);
30          }
31          g_mutex_unlock (p->mutex);
32
33          return NULL;
34  }
35
36  int
37  main (void)
38  {
39  #if defined(G_THREADS_ENABLED) && !defined(G_THREADS_IMPL_NONE)
40          g_thread_init (NULL);
41
42          struct mutex_int_t i = {g_mutex_new (), 0};
43
44          GThread *t1 = g_thread_create (thread_1, &i, TRUE, NULL);
45          GThread *t2 = g_thread_create (thread_2, &i, TRUE, NULL);
46
47          g_thread_join (t1);
48          g_thread_join (t2);
49
```

```
50          g_mutex_free (i.mutex);
51  #endif
52          return 0;
53  }
```

对数据区域加锁之后，那么无论线程 thread_1 与 thread_2 谁先被执行，在其修改并输出整型变量 i 的值时不会被另外的线程所干扰，除非它解除锁定。

对于上例，加锁后，多线程的意义就荡然无存了，因其输出结果要么是：

```
thread_2 said 1
thread_2 said 2
thread_2 said 3
thread_2 said 4
thread_1 said 5
thread_1 said 6
thread_1 said 7
thread_1 said 8
```

要么是：

```
thread_1 said 1
thread_1 said 2
thread_1 said 3
thread_1 said 4
thread_2 said 5
thread_2 said 6
thread_2 said 7
thread_2 said 8
```

其效率尚不如单线程程序。

假如上例先被执行的线程是 thread_2，那么如果去掉 thread_2 中的 g_mutex_unlock 函数，那么程序会挂在 thread_1 处。这是因为 thread_2 没有对整型变量 i 数据区域解锁，那么 thread_1 只能是望眼欲穿的等待。那么，有没有办法可以让 thread_1 在得知自己没资格访问某数据区域之时能识相的做它该做的事情呢？办法是有的，只需：

```
1   static gpointer
2   thread_1 (gpointer data)
3   {
4           struct mutex_int_t * p = (struct mutex_int_t *)data;
5
6           if (!g_mutex_trylock (p->mutex))
7                   g_thread_exit (NULL);
8
9           for (int i = 0; i < 4; i++){
                    p->i++;
10                  g_print ("thread_1 said %d\n", p->i);
11          }
```

```
12          g_mutex_unlock (p->mutex);
13
14          return NULL;
15  }
```

这样，一旦 thread_1 发现等不到访问整型变量 i 的机会便识相的退出。当然，这个示例中的 thread_1 太过于消极，虽然我们不能在一棵树上吊死，但也不能因为一时没有找到树，就自个把自己掐死，阿弥陀佛！

3.3 条件变量

互斥，是一种较为简单的线程同步方法，如果配合条件变量，那么可以营造生产者/消费者这样一种模式。例如，可以开启一个线程，用于获取标准输入（stdin）文件的数据，然后开启多个线程，对所获数据进行不同目标的处理，详见：

```
1   #include <glib.h>
2
3   GCond* data_cond = NULL;
4   GMutex* data_mutex = NULL;
5   gpointer current_data = NULL;
6   gchar *buffer = NULL;
7
8   static void
9   consumer_1 (void)
10  {
11          g_mutex_lock (data_mutex);
12          while (!buffer)
13                  g_cond_wait (data_cond, data_mutex);
14          g_print ("%ld\n", g_utf8_strlen (buffer, -1) - 1);
15          g_mutex_unlock (data_mutex);
16  }
17
18  static void
19  consumer_2 (void)
20  {
21          g_mutex_lock (data_mutex);
22          while (!buffer)
23                  g_cond_wait (data_cond, data_mutex);
24          for (int i = g_utf8_strlen (buffer, -1) - 2; i >= 0; i--)
25                  g_print ("%c", buffer[i]);
26          g_print ("\n");
27          g_mutex_unlock (data_mutex);
28  }
29
30  gboolean
31  io_watch (GIOChannel *channel, GIOCondition condition, gpointer data)
```

```
32  {
33          g_mutex_lock (data_mutex);
34          g_io_channel_read_line (channel, &buffer, NULL, NULL, NULL);
35          g_cond_signal (data_cond);
36          g_mutex_unlock (data_mutex);
37
38          GThread *consumer_thread_1 =
39                  g_thread_create ((GThreadFunc)consumer_1,
40                                   NULL, TRUE, NULL);
41
42          GThread *consumer_thread_2 =
43                  g_thread_create ((GThreadFunc)consumer_2,
44                                   NULL, TRUE, NULL);
45
46          g_thread_join (consumer_thread_1);
47          g_thread_join (consumer_thread_2);
48
49          g_free (buffer);
50          buffer = NULL;
51
52          return TRUE;
53  }
54
55  int
56  main (void)
57  {
58  #if defined(G_THREADS_ENABLED) && !defined(G_THREADS_IMPL_NONE)
59          g_thread_init (NULL);
60
61          data_cond = g_cond_new ();
62          data_mutex = g_mutex_new ();
63
64          GMainLoop *loop = g_main_loop_new (NULL, FALSE);
65          GIOChannel* channel = g_io_channel_unix_new (1);
66          if (channel){
67                  g_io_add_watch(channel, G_IO_IN, io_watch, NULL);
68                  g_io_channel_unref(channel);
69          }
70          g_main_loop_run(loop);
71          g_main_context_unref (g_main_loop_get_context (loop));
72          g_main_loop_unref(loop);
73
74          g_mutex_free (data_mutex);
75          g_cond_free (data_cond);
76  #endif
77          return 0;
78  }
```

上例中，主线程是生产者，它利用 GLib 主事件循环与文件事件源机制实现 stdin 中的数据获取，在文件事件源的处理函数 io_watch 中创建了两个子线程，它们是消费者，即它们使用了主线程所获取的数据。

消费者线程创建之后，它们会被挂住，除非生产者对所“生产”的数据解锁。生产者在完成数据获取任务后，会使用 g_cond_signal 函数向消费者线程发送信号，通知它们数据已经准备好，开始享用吧！这两个消费者，一个会统计 stdin 输入的字符串长度，另一个会将字符串倒序输出至 stdout。

上例代码中，消费者线程均包含以下代码段：

```
1          g_mutex_lock (data_mutex);
2          while (!buffer)
3                  g_cond_wait (data_cond, data_mutex);
4          ... ...
5          g_mutex_unlock (data_mutex);
```

是处理条件变量的标准形式。该段代码首先锁定互斥，然后进入 g_cond_wait 开始等待，g_cond_wait 会将互斥设定为非锁定的，此时那个生产者可能正在准备数据，当生产者调用 g_cond_signal 发送信号时，g_cond_wait 便返回，并将互斥设为锁定状态，然后消费者便开始大肆享用生产者所提供的数据，享毕便解除互斥的锁定。

3.4 原子操作

使用互斥，可以保护数据不会被暴走的多个线程破坏，但是它的效率有时会很低。比方说某个线程一旦长时间霸占了某个变量，也许会导致其他线程要么苦苦等待，要么就无所事事的飘过。我这里不是在背后说互斥的坏话，但是如果仅仅是为了整型变量或指针变量的线程安全问题，那么我们应该选择 GLib 提供的“原子”操作，这样可能会提高一些效率。之所以说可能，是因为 GLib 的原子操作是利用内联汇编来实现的，而对于那些不支持内联汇编的平台（C 编译器），GLib 则提供基于互斥的相应实现。例如，在 GLib 内部实现中被应用较多的一个原子操作是 g_atomic_int_inc，其实现代码如下：

```
1   void
2   g_atomic_int_add (volatile gint G_GNUC_MAY_ALIAS *atomic, gint val)
3   {
4      __asm__ __volatile__ ("lock; addl %1,%0"
5                            : "=m" (*atomic)
6                            : "ir" (val), "m" (*atomic));
7   }
8   #define g_atomic_int_inc(atomic) (g_atomic_int_add ((atomic), 1))
```

我承认，很久以前，我曾经看过一段时间的 AT&T 汇编语言，但是上述代码我依然看不大懂。不过，这不妨碍我在程序中使用 g_atomic_int_add 实现线程安全的整数增量为 1 的计算。

3.5 线程池

有的时候，为了提高效率，一个工作线程可能会需要将一部分任务分给一些线程去完成，并且它不需要与那些线程同步（就是所谓的异步线程）。比如，网络服务器的主线程可以接受许多用户的访问，并为每个访问创建一个线程完成相应的任务，例如向用户发送 HTML 页面。但是，如果用户访问非常频繁，那么线程的创建和销毁会降低服务器程序的运行效率。

线程池技术可以预先创建一定数量的线程，从而避免线程频繁创建和销毁的系统开销。也许在计算机程序设计中，最不容易过时的技术之一是牺牲空间换取时间的技术。下面利用 GLib 的主事件循环配合线程池模拟一个 Web 服务器程序，你可以向它发送 url，然后它会装模作样的给你分配一个线程并沉默 5 秒钟，然后输出一句废话。

```
1  #include <glib.h>
2
3  #define NUMBER_OF_THREADS 4
4
5  typedef struct _MyData MyData;
6  struct _MyData {
7          GThreadPool *thread_pool;
8          gchar *buffer;
9  };
10
11 static void
12 send_message (gpointer data, gpointer user_data)
13 {
14         g_usleep (5 * G_USEC_PER_SEC);
15         g_print (">>>> OK! I got %s", (gchar *)data);
16         g_free (data);
17 }
18
19 gboolean
20 io_watch (GIOChannel *channel, GIOCondition condition, gpointer data)
21 {
22         MyData *mydata = (MyData *)data;
23
24         g_io_channel_read_line (channel, &(mydata->buffer), NULL, NULL,
   NULL);
25         g_thread_pool_push (mydata->thread_pool, mydata->buffer, NULL);
26
27         return TRUE;
28 }
29
30 int
31 main (void)
32 {
33 #if defined(G_THREADS_ENABLED) && !defined(G_THREADS_IMPL_NONE)
```

```
34          g_thread_init (NULL);
35
36          MyData mydata = {NULL, NULL};
37          mydata.thread_pool = g_thread_pool_new (send_message,
38                                                  NULL,
39                                                  NUMBER_OF_THREADS,
40                                                  FALSE,
41                                                  NULL);
42
43          GMainLoop *loop = g_main_loop_new (NULL, FALSE);
44          GIOChannel* channel = g_io_channel_unix_new (1);
45          if (channel){
46                  g_io_add_watch(channel, G_IO_IN, io_watch, &mydata);
47                  g_io_channel_unref(channel);
48          }
49
50          g_main_loop_run(loop);
51          g_main_context_unref (g_main_loop_get_context (loop));
52          g_main_loop_unref(loop);
53
54          g_thread_pool_free (mydata.thread_pool, TRUE, TRUE);
55  #endif
56          return 0;
57  }
```

这个程序的输入与输出类似：

```
$ > ./a.out
http://www.linuxsir.org/bbs/forum59.html
http://garfileo.is-programmer.com/
>>>> OK! I got http://www.linuxsir.org/bbs/forum59.html
>>>> OK! I got http://garfileo.is-programmer.com/
http://gentoo-portage.com/newest
>>>> OK! I got http://gentoo-portage.com/newest
```

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将这个线程放到线程池中，以备后用。

3.6 异步队列

在 3.5 节提供的示例中，主线程与子线程之间的“通信”是利用共享内存实现的。为了增强线程通讯的安全性，可以使用异步队列。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。例如：

```
#include <glib.h>

#define NUMBER_OF_THREADS 4

typedef struct _MyData MyData;
struct _MyData {
        GThreadPool *thread_pool;
        GAsyncQueue *queue;
};

static void
send_message (gpointer data, gpointer user_data)
{

        MyData *p = (MyData *)data;
        gchar *buffer = (gchar *)g_async_queue_pop (p->queue);
        g_usleep (5 * G_USEC_PER_SEC);
        g_print (">>>> OK! I got %s", buffer);
        g_free (buffer);
}

gboolean
io_watch (GIOChannel *channel, GIOCondition condition, gpointer data)
{
        MyData *p = (MyData *)data;
        gchar *buffer;

        g_thread_pool_push (p->thread_pool, data, NULL);

        g_io_channel_read_line (channel, &buffer, NULL, NULL, NULL);
        g_async_queue_push (p->queue, buffer);

        return TRUE;
}

int
main (void)
{
#if defined(G_THREADS_ENABLED) && !defined(G_THREADS_IMPL_NONE)
        g_thread_init (NULL);

        MyData data;
        data.thread_pool = g_thread_pool_new (send_message,
                                              NULL,
                                              NUMBER_OF_THREADS,
                                              FALSE,
```

```
47                                                      NULL);
48          data.queue = g_async_queue_new ();
49          GMainLoop *loop = g_main_loop_new (NULL, FALSE);
50          GIOChannel* channel = g_io_channel_unix_new (1);
51          if (channel){
52                  g_io_add_watch(channel, G_IO_IN, io_watch, &data);
53                  g_io_channel_unref(channel);
54          }
55
56          g_main_loop_run(loop);
57          g_main_context_unref (g_main_loop_get_context (loop));
58          g_main_loop_unref(loop);
59
60          g_thread_pool_free (data.thread_pool, TRUE, TRUE);
61          g_async_queue_unref (data.queue);
62  #endif
63          return 0;
64  }
```

3.7 想知道更多

阅读 GLib 手册的“Threads”和“Thread Pools”部分是必须的。

参考文献

索引