第一章 入门

1.1 Hello!World!

1.11 第一个 GTK 程序

不可免俗的，从最简单的基本窗口产生开始介绍，窗口标题就叫作「哈啰！GTK+!」好了，请使用任一编辑器来编辑一个 helloGtk.c 的档案，内容如下：

helloGtk.c

```
#include <gtk/gtk.h>

int main(int argc, char *argv[]) {
    GtkWidget *window;

    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "哈啰！ GTK+！ ");
    gtk_widget_show(window);
    gtk_main();

    return 0;
}
```

首先 include 必要的 GTK 标头档案，接着先看到 gtk_init()，这个函式会先初始化函式库，设定预设信号处理，并让 GTK 有机会处理传递给程序的命令列自变量，GTK 会检查是否有以下的自变量并处理：

- gtk-module
- g-fatal-warnings
- gtk-debug
- gtk-no-debug
- gdk-debug
- gdk-no-debug
- display
- sync
- name
- class

这些自变量会从自变量列中移除，剩下的部份留待您自己的程序逻辑来处理。

GTK 虽然使用 C 来撰写，但是透过 GObject 函式库，可以支持对象导向的对象封装、继承观念，透过宏还可以支持多型的观念（至少概念上达到一些部份），一个 GTK 对象阶层如下所示：

```
GObject
 +--GInitiallyUnowned
     +-- GtkObject
           +-- GtkWidget
                 +-- GtkContainer
                       +-- GtkBin
                             +-- GtkWindow
```

gtk_window_new()会建立一个 GtkWindow，这是一个基本的窗口对象，GtkWindow 继承自 GtkBin，GtkBin 继承自 GtkContainer，GtkContainer 可以容纳其它 widget，所以在 GtkWindow 中可以置放其它的 widget，而它们全都是 GtkWidget 的后代。

在函式库的组织上，GTK+的参考手册 中，若要查询与 GtkWindow 设定的相关函式，也就是 gtk_window 开头的函式名称，则直接查询 GtkWindow 的说明文件。

在范例中，您使用 gtk_window_new()在内存中产生一个 GtkWindow（但还不是真正出现在屏幕画面中），参数设定为 GtkWindowType，有两个可用的设定：

GTK_WINDOW_TOPLEVEL：一个有外框的标准 GTK 窗口。
GTK_WINDOW_POPUP：一个蹦现窗口，像是对话框、蹦现选单或提示文字。

若要设定 GtkWindow 标题文字，则使用 gtk_window_set_title()，GTK_WINDOW 宏用以将 window 对象转型为 GtkWindow 型态。

gtk_window_new()只是在内存中产生一个 GtkWindow，若要真正在屏幕画面中显示 GTK 的 widget，则使用 gtk_widget_show()，最后呼叫 gtk_main()，这会将程序的控制权交给 GTK，由 GTK 来等待键盘、按钮等事件或是档案 IO 通知。
您可以使用以下的指令来进行编译与执行：

```
$ gcc helloGtk.c -o helloGtk `pkg-config --cflags --libs gtk+-2.0`
$ ./helloGtk
```

pkg-config 会先取得 GTK 的标头文件位置与函式库信息，然后再供给 gcc 进行编译，一个程序的执行画面如下所示，这个程序的原始码使用 UTF8 编码，在编译执行之后，可直接显示中文：

由于 gtk_main()会将控制权交给了 GTK，直到呼叫 gtk_main_quit()之前都不会返回，这个范例目前还没有实作这个部份（之后还会介绍如何实作），因此这个窗口您按下右上 X 钮也不会真正关闭，而必须在文字模式下先使用 Ctrl+C 强制中断程序。

有关 gtk_main()、gtk_main_quit()等函式的说明，可以参考 GTK+参考文件中的 Main loop and Events。

1.12 GTK 的对象导向架构

GTK 基本上是使用 C 语言来撰写，即使 C 语言本身不支持对象导向，但 GTK 在架构上运用了一些方式，使得使用 GTK 时可以支持许多对象导向的概念。

在对象导向的封装特性上，GTK 以结构（structure）的方式来模拟类别，事实上 GTK 也直接称这些结构为类别，以建构 GtkWindow 的程序代码为例：

```
window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
```

在函式的组织上，与 GtkWindow 相关的函式，都是以 gtk_window 名称作为开头，gtk_window_new()就像是对象导向程序语言中的建构式，如果要设置 GtkWindow 的相关属性，例如标题名称：

```
gtk_window_set_title(GTK_WINDOW(window), "哈啰！GTK+！");
```

gtk_window_set_title()的第一个参数接受 GtkWindow 指标，透过这种方式，让实际上属于全域的函式，看来就像是专属于 GtkWindow 所使用，就如同对象上所带有的公开（public）方法（method）或成员函式（member function），而在私有（private）的模拟上，GTK 使用 static 函式，例如在 gtkwindow.c 原始程序代码中，可以看到：

```
static void gtk_window_dispose          (GObject      *object);
static void gtk_window_destroy          (GtkObject    *object);
static void gtk_window_finalize         (GObject      *object);
static void gtk_window_show             (GtkWidget    *widget);
```

```
static void gtk_window_hide               (GtkWidget        *widget);
static void gtk_window_map                (GtkWidget        *widget);
static void gtk_window_unmap              (GtkWidget        *widget);
static void gtk_window_realize            (GtkWidget        *widget);
static void gtk_window_unrealize          (GtkWidget        *widget);
```

这些 static 函式不会出现在 gtkwindow.h 标头文件中，仅可在 gtkwindow.c 中使用，这看起来就像是 GtkWindow 的专属私用函式。

在继承上，GTK 实际上使用结构链接（link）的方式，在 第一个 GTK 程序 中看过以下的继承关系：

```
GObject
 +--GInitiallyUnowned
     +-- GtkObject
           +-- GtkWidget
                 +-- GtkContainer
                       +-- GtkBin
                             +-- GtkWindow
```

以 GtkContainer 为例，在 gtkcontainer.h 中有如下的定义：

```
typedef struct _GtkContainer       GtkContainer;
...

struct _GtkContainer
{
  GtkWidget widget;

  GtkWidget *focus_child;

  guint border_width : 16;

  /*< private >*/
  guint need_resize : 1;
  guint resize_mode : 2;
  guint reallocate_redraws : 1;
  guint has_focus_chain : 1;
};
```

GtkContainer 的成员中有一个 GtkWidget，而再来看到 gtkwidget.h：

```
typedef struct _GtkWindow          GtkWindow;
...
```

```
struct _GtkWidget
{
  /* The object structure needs to be the first
   *  element in the widget structure in order for
   *  the object mechanism to work correctly. This
   *  allows a GtkWidget pointer to be cast to a
   *  GtkObject pointer.
   */
  GtkObject object;
    ...
};
```

GtkWidget 中有个成员为 GtkObject，以如此的链接关系来维持对象上的继承关系架构，而在 gtk_window_set_title()函式的使用例子中：

```
gtk_window_set_title(GTK_WINDOW(window), "哈啰！GTK+！");
```

GTK_WINDOW 是一个宏，用来进行指针型态转型动作：

```
#define GTK_WINDOW(obj)
          (G_TYPE_CHECK_INSTANCE_CAST((obj), GTK_TYPE_WINDOW, GtkWindow))
```

G_TYPE_CHECK_INSTANCE_CAST 宏定义在 GLib 的 gtype.h（/usr/include/glib-2.0/gobject/gtype.h）中：

```
#define G_TYPE_CHECK_INSTANCE_CAST(instance, g_type, c_type)
            (_G_TYPE_CIC ((instance), (g_type), c_type))
```

G_TYPE_CHECK_INSTANCE_CAST 宏会检查 instance 是否为 g_type 的一个实例，如果不是的话就发出警示讯息，若是的话就将指针转型为 c_type 型态（参考 G_TYPE_CHECK_INSTANCE_CAST 在线文件说明）。

即便在熟悉 C++、Java 等支持对象导向程序语言的人来说，这样的架构在对象导向的概念上并不完整，但确实在易读与维护性上加强了不少。

1.2 Signal 与 Callback

1.21 使用 Signal 关闭窗口

在 第一个 GTK 程序 中，当您按下窗口右上 X 钮时，在 GTK 窗口的预设处理中，只会隐藏窗口，而不会直接关闭程序，GTK 有一套 Signal 与 Callback 函式的处理机制，在某个动作发生时，GTK 会发出特定 Signal，若您想要进行某些处理，则需定义 Callback 函式，并透过 g_signal_connect()等函式，将 Signal 与 Callback 函式加以连结。

以按下窗口右上 X 钮为例，按下 X 钮后，GTK 预设会发出"destroy"的 Signal，您可以使用

g_signal_connect()将之连结至 gtk_main_quit()函式，这个函式会结束 gtk_main()的循环处理因而结束 GTK 程序：

```
#include <gtk/gtk.h>

int main(int argc, char *argv[]) {
    GtkWidget *window;

    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "哈啰！ GTK+！ ");

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show(window);
    gtk_main();

    return 0;
}
```

从 g_、G_开头的函式名称可以知道，它们是 GLib 所提供的函式与宏（在 GTK+ 2.0 中，Signal 处理已由 GTK 移至 GLib），g_signal_connect()第一个参数，必须是 GtkObject 或其衍生的类别实例，代表 Signal 发出的来源对象，第二个参数是感兴趣的信号，第三个参数是 Callback 函式，G_CALLBACK 宏强制会转换函式型态为无参数无传回值的 GCallback 函式型态：

```
#define G_CALLBACK(f) ((GCallback) (f))
```

在这边使用 GTK 的 gtk_main_quit()函式，第四个参数是可以传递给 Callback 函式的相关资料，在这边不需要，设定为 NULL 即可。

g_signal_connect()实际上是宏（定义在/usr/include/glib-2.0/gobject/gsignal.h），方便使用 g_signal_connect_data()函式：

```
#define g_signal_connect(instance, detailed_signal, c_handler, data) \
    g_signal_connect_data ((instance), (detailed_signal), (c_handler), (data),
NULL, (GConnectFlags) 0)
```

g_signal_connect_data()会传回 gulong 型态的 handler id，函式的定义如下：

```
gulong g_signal_connect_data(gpointer instance,
                             const gchar *detailed_signal,
                             GCallback c_handler,
                             gpointer data,
                             GClosureNotify destroy_data,
```

```
                                GConnectFlags connect_flags);
```

如果您打算将 Signal 与 Callback 断开连结，可以根据传回的 handler id 来使用 g_signal_handler_disconnect() 函式，也可以根据 handler id 来使用 g_signal_handler_is_connected()函式，测试 Signal 的连接状态：

```
void g_signal_handler_disconnect(gpointer object, gulong id);
gboolean g_signal_handler_is_connected(gpointer instance, gulong handler_id);
```

例如一个连接 Signal 与断开 Signal 的程序片段如下：

```
....
gulong handler_id = g_signal_connect(GTK_OBJECT(window), "destroy",
                                     G_CALLBACK(gtk_main_quit), NULL);
....
if (g_signal_handler_is_connected(window, id)) {
    g_signal_handler_disconnect(GTK_OBJECTS(window), handler_id);
}
```

若只是想暂停（block）某 Signal 处理，则可以使用 g_signal_handler_block()，想恢复被暂停的 Signal 处理，则可以使用 g_signal_handler_unblock()：

```
void g_signal_handler_block(gpointer object, gulong id);
void g_signal_handler_unblock(gpointer object, gulong id);
```

一个被 g_signal_handler_block() 函式呼叫 n 次的 Signal 处理，也必须被 g_signal_handler_unblock()相对应的次数，才可以恢复原本来未暂停的状态。

若在未知 handler id 的情况下，想要中断、暂停或恢复信号连结，则可以尝试使用 g_signal_handlers_disconnect_by_func() 、 g_signal_handlers_block_by_func() 、 g_signal_handlers_unblock_by_func()，这三者其实都是宏：

```
#define g_signal_handlers_disconnect_by_func(instance, func, data) \
    g_signal_handlers_disconnect_matched ((instance), \
    (GSignalMatchType) (G_SIGNAL_MATCH_FUNC | G_SIGNAL_MATCH_DATA), \
    0, 0, NULL, (func), (data))

#define g_signal_handlers_block_by_func(instance, func, data) \
    g_signal_handlers_block_matched ((instance), \
    (GSignalMatchType) (G_SIGNAL_MATCH_FUNC | G_SIGNAL_MATCH_DATA), \
                                       0, 0, NULL, (func), (data))

#define g_signal_handlers_unblock_by_func(instance, func, data) \
    g_signal_handlers_unblock_matched ((instance), \
```

```
    (GSignalMatchType) (G_SIGNAL_MATCH_FUNC | G_SIGNAL_MATCH_DATA), \
    0, 0, NULL, (func), (data))
```

g_signal_handlers_disconnect_matched() 、 g_signal_handlers_block_matched () 、g_signal_handlers_unblock_matched()会传回 guint 的数值，表示符合的 handler 数目：

```
guint g_signal_handlers_disconnect_matched(gpointer instance,
                                           GSignalMatchType mask,
                                           guint signal_id,
                                           GQuark detail,
                                           GClosure *closure,
                                           gpointer func,
                                           gpointer data);

guint g_signal_handlers_block_matched(gpointer instance,
                                      GSignalMatchType mask,
                                      guint signal_id,
                                      GQuark detail,
                                      GClosure *closure,
                                      gpointer func,
                                      gpointer data);

guint g_signal_handlers_unblock_matched(gpointer instance,
                                        GSignalMatchType mask,
                                        guint signal_id,
                                        GQuark detail,
                                        GClosure *closure,
                                        gpointer func,
                                        gpointer data);
```

更多有关 Signal 函式的说明，可以参考 GObject 参考文件的 Signals。

1.22 自订 callback 函式

您可以自订 callback 函式，使用 g_signa_connect()函式连接，在指定的 Signal 发生时呼叫自订的 callback 函式，自订的 callback 函式原型基本上是以下的形式：

```
void func_name(GtkWidget *widget, ..., gpointer user_data);
```

第一个参数是发出信号的 Widget 指针，第二个参数是呼叫 callback 函式时，所要传递给 callback 函式的相关数据，举个实际的例子，下面这个范例会有一个按钮，当按下按钮时，会在主控台下显示指定的讯息：

callback_demo.c

```
#include <gtk/gtk.h>

// 自订 Callback 函式
void button_clicked(GtkWidget *button, gpointer data) {
    g_print("按钮按下：%s\n", (char *) data);
}

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *button;

    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "哈啰！ GTK+！ ");

    button = gtk_button_new_with_label("按我");
    gtk_container_add(GTK_CONTAINER(window), button);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);
    g_signal_connect(GTK_OBJECT(button), "clicked",
                     G_CALLBACK(button_clicked), "哈啰！ 按钮！ ");

    gtk_widget_show(window);
    gtk_widget_show(button);

    gtk_main();

    return 0;
}
```

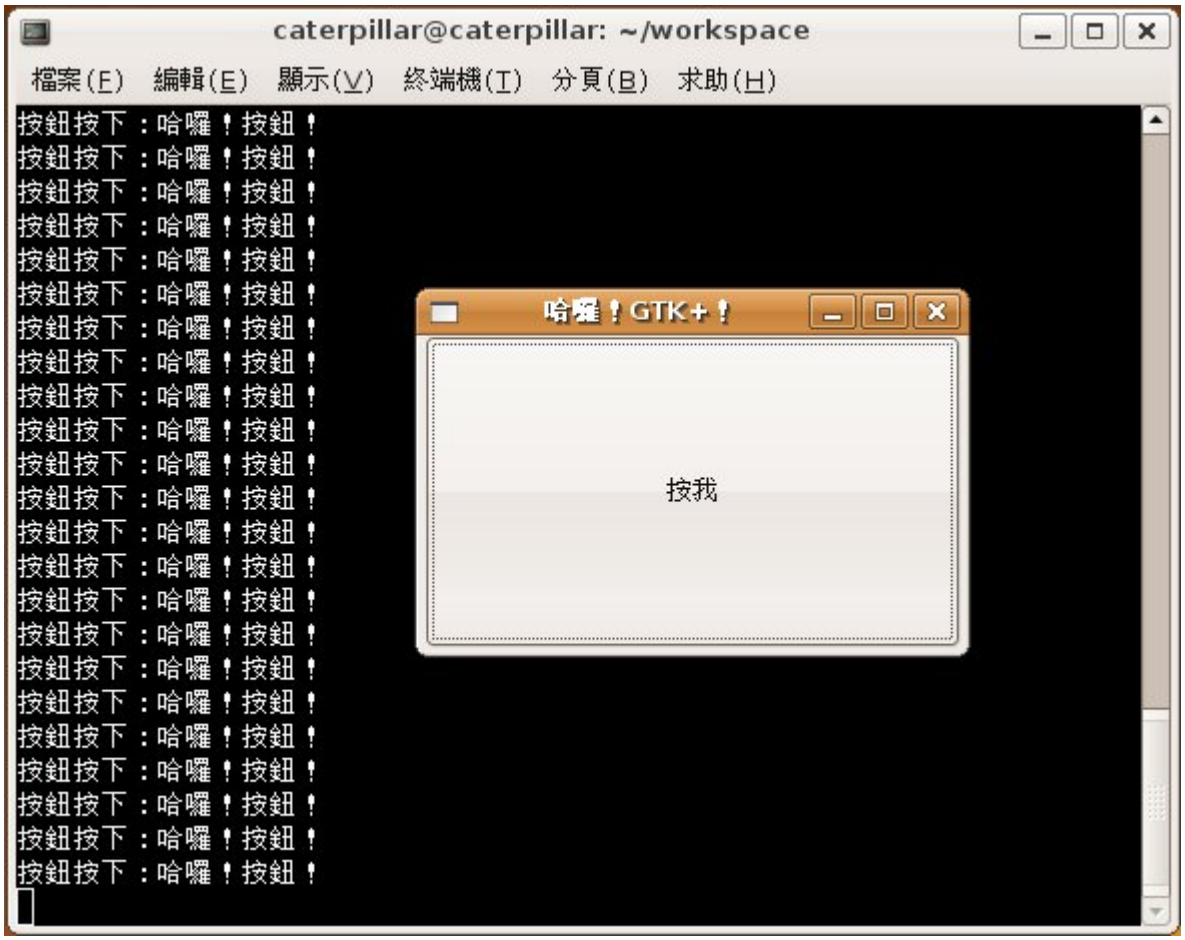
这个范例是以 使用 Signal 关闭窗口 为基础，增加了一个按钮，程序中注意的是 Callback 函式的定义，而要产生一个具有文字的按钮，可以使用 gtk_button_new_with_label()函式，由于 GtkWindow 是 GtkContainer 的子类，所以它可以容纳其它的 Widget 组件，在这边使用 gtk_container_add()函式将 GtkButton 加入 GtkWindow 之中，目前没有设置任何版面配置，所以按钮就填满整个窗口。

要连接自订的 Callback 函式，一样使用 g_signal_connect()，注意到虽然 G_CALLBACK 宏会将函式指针转换为无参数无传回值的型态，但这不代表 Callback 函式不可以传回值或带有参数，实际操作时是依当时所连接的函式型态而定，在这边，g_signal_connect()最后一个参数将传递给 button_clicked()函式的 data 参数。

要显示加入的 GtkButton，在这边再次使用 gtk_widget_show()，您也可以直接使用 gtk_widget_show_all()，将所有在指定的 GtkContainer 中的组件都显示出来：

```
gtk_widget_show_all(window);
```

一个执行时的画面如下所示：



另一个较少使用的 Signal 连结函式是 g_signal_connect_swapped()，它实际上也是个宏，方便使用 g_signal_connect_data()函式，定义如下：

```
#define g_signal_connect_swapped(instance, detailed_signal, c_handler, data) \
    g_signal_connect_data ((instance), (detailed_signal), (c_handler), (data),
NULL, G_CONNECT_SWAPPED)
```

它所对应的 Callback 函式应如下定义：

```
void func_name(gpointer user_data, ..., GtkWidget *widget);
```

简单的说，就是两个参数对调，g_signal_connect_swapped()常用来接结仅接受一个单独 Widget 作为参数的 callback 函式，举个例子来说，像 gtk_widget_destroy() 函式：

```
void gtk_widget_destroy(GtkWidget *widget);
```

这个函式会消除指定的 Widget，若您在程序中，想要指定消除一个 Widget，则可以使用这个函式，举例来说，若要按下按钮后，消除 GtkWindow，则可以如下：

callback_demo.c

```
#include <gtk/gtk.h>

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *button;

    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "哈啰！ GTK+！ ");

    button = gtk_button_new_with_label("按我");
    gtk_container_add(GTK_CONTAINER(window), button);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);
    g_signal_connect_swapped(GTK_OBJECT(button), "clicked",
                             G_CALLBACK(gtk_widget_destroy), window);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

这个程序在按下按钮后，会消去 window 所储存的 GtkWindow，由于这是唯一个 GtkWindow，消除后程序也就跟着结束。您也可以如前一个范例，在自订的 button_clicked()函式中呼叫 gtk_widget_destroy()：

callback_demo.c

```
#include <gtk/gtk.h>

// 自订 Callback 函式
void button_clicked(GtkWidget *button, gpointer data) {
    gtk_widget_destroy((GtkWidget*) data);
}

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *button;

    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
```

```
    gtk_window_set_title(GTK_WINDOW(window), "哈啰！ GTK+！ ");

    button = gtk_button_new_with_label("按我");
    gtk_container_add(GTK_CONTAINER(window), button);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);
    g_signal_connect(GTK_OBJECT(button), "clicked",
                     G_CALLBACK(button_clicked), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

虽然程序的执行也可以透过按下按钮消除 GtkWindow，但直接使用 g_signal_connect_swapped()会是比较直接的方式。

另也还有个 g_signal_connect_after()函式值得注意，它实际也是宏，定义如下：

```
#define g_signal_connect_after(instance, detailed_signal, c_handler, data) \
    g_signal_connect_data ((instance), (detailed_signal), (c_handler), (data),
NULL, G_CONNECT_AFTER)
```

若您使用 g_signal_connect_after()来连接 callback 函式，则该 callback 函式，会在所有使用 g_signal_connect()设定的 callback 函式执行之后再执行。

1.23 内建 Signal 的发射与停止

GTK 的 Signal 不一定得由事件来发出，您可以主动发出 Signal，利用 g_signal_emit_by_name()，您可以指定一个对象已建立的 Signal 名称来发出该 Signal。

下面这个程序利用 POSIX 执行绪（GLib 亦有提供 GThread 来启用多执行绪），改写 自订 callback 函式 中的范例，每秒发出一个 GtkButton 的"clicked" Signal，程序开始后即使您没有按下按钮，也会在终端机下显示"按钮按下：哈啰！按钮！"的讯息：

signal_emit_demo.c

```
#include <gtk/gtk.h>
#include <pthread.h>

void *signal_thread(void *arg) {
    int i;

    for(i = 0; i < 5; i++) {
```

```
            sleep(1);
            g_signal_emit_by_name(arg, "clicked");
        }

        pthread_exit("Thread exit");
}

// 自订 Callback 函式
void button_clicked(GtkWidget *button, gpointer data) {
        g_print("按钮按下：%s\n", (char *) data);
}

int main(int argc, char *argv[]) {
        pthread_t a_thread;

        GtkWidget *window;
        GtkWidget *button;

        gtk_init(&argc, &argv);
        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        gtk_window_set_title(GTK_WINDOW(window), "哈啰！GTK+！");

        button = gtk_button_new_with_label("按我");
        gtk_container_add(GTK_CONTAINER(window), button);

        g_signal_connect(GTK_OBJECT(window), "destroy",
                                G_CALLBACK(gtk_main_quit), NULL);
        g_signal_connect(GTK_OBJECT(button), "clicked",
                                G_CALLBACK(button_clicked), "哈啰！按钮！");

        gtk_widget_show(window);
        gtk_widget_show(button);

        pthread_create(&a_thread, NULL, signal_thread, button);

        gtk_main();

        return 0;
}
```

程序执行后，会使用另一个执行绪，每秒发出一个"clicked" Signal，为了使用 POSIX 执行绪，编译这个程序时需要定义_REENTRANT 及使用 pthread 链接库：

```
$ gcc signal_emit_demo.c -o signal_emit_demo -D_REENTRANT -lpthread `pkg-config --cflags --libs gtk+-2.0`
```

g_signal_emit_by_name()可以发出 Signal，如果您想要中止 Signal 的传播，则可以使用 g_signal_stop_by_name()，例如在某个 Signal 处理函式处理完毕后，若不想让其它的 Signal 处理函式继续处理了，则可以使用 g_signal_stop_by_name()来停止 Signal。

若不想使用pthread来撰写这个程序，则可以考虑使用GLib的 **Timeout** 。

1.3 事件处理

1.31GDK 事件结构

GTK 透过 GDK 来处理事件，GDK 会将每个接受到的 XEvent 转换为 GdkEvent，然后传播给 GtkWidget，引发一个与事件相对应的事件 Signal，再透过 Callback 函式处理事件。

GdkEvent 是个 C union 的定义：

```
union GdkEvent
{
  GdkEventType            type;
  GdkEventAny            any;
  GdkEventExpose        expose;
  GdkEventNoExpose        no_expose;
  GdkEventVisibility        visibility;
  GdkEventMotion        motion;
  GdkEventButton        button;
  GdkEventScroll            scroll;
  GdkEventKey            key;
  GdkEventCrossing        crossing;
  GdkEventFocus            focus_change;
  GdkEventConfigure        configure;
  GdkEventProperty        property;
  GdkEventSelection        selection;
  GdkEventOwnerChange          owner_change;
  GdkEventProximity        proximity;
  GdkEventClient        client;
  GdkEventDND            dnd;
  GdkEventWindowState       window_state;
  GdkEventSetting           setting;
  GdkEventGrabBroken        grab_broken;
};
```

GdkEvent 代表所有的事件型态，其成员 GdkEventType 为 enum 型态，可透过它来了解目前的事件是哪个类型：

```
GdkEventType type = event->type; // event 的宣告为 GdkEvent *event
```

```
switch(type) {
    case GDK_DELETE:
        g_print("GDK_DELETE");
        ...
        break;
    case GDK_DESTROY:
        g_print("GDK_DELETE");
        ...
        break;
        ...
}
```

一个使用的例子是，您可以设计一个接受所有事件的处理函式，根据 GdkEventType 来决定对所有事件作过滤处理，方式就如上面的程序代码片段所示范的，一个 GdkEvent 可能对应多个 GdkEventType,例如 GdkButtonEvent 可以对应的 GdkEventType 有 GDK_BUTTON_PRESS、GDK_2BUTTON_PRESS、GDK_3BUTTON_PRESS 与 GDK_BUTTON_RELEASE。

每个事件结构有其个别成员，例如 GdkEventButton 有 x 与 y 成员，代表鼠标光标相对于窗口的位置：

```
typedef struct {
  GdkEventType type;
  GdkWindow *window;
  gint8 send_event;
  guint32 time;
  gdouble x;
  gdouble y;
  gdouble *axes;
  guint state;
  guint button;
  GdkDevice *device;
  gdouble x_root, y_root;
} GdkEventButton;
```

您可以这么存取 x 或 y 成员：

```
gdouble x = event->button.x;
gdouble y = event->button.y;
```

或是将之转型为 GdkEventButton 再进行存取：

```
gdouble x = ((GdkEventButton*)event)->x;
gdouble y = ((GdkEventButton*)event)->y;
```

GdkEvent 传播给 GtkWidget，引发一个与事件相对应的事件 Signal，您可以使用 g_signal_connect()函式连结该 Signal 与 Callback 函式来处理事件，事件类型与对应的事件 Signal 名称，基本上就是去掉 GDK_名称，转为小写并加上_event 名称，例如：

GDK_DELETE	delete_event
GDK_DESTROY	destroy_event
GDK_EXPOSE	expose_event
GDK_MOTION_NOTIFY	motion_notify_event
GDK_BUTTON_PRESS	button_press_event
GDK_2BUTTON_PRESS	button_press_event
GDK_3BUTTON_PRESS	button_press_event
GDK_BUTTON_RELEASE	button_release_event
GDK_KEY_PRESS	key_press_event
GDK_KEY_RELEASE	key_release_event
GDK_ENTER_NOTIFY	enter_notify_event
GDK_LEAVE_NOTIFY	leave_notify_event
GDK_FOCUS_CHANGE	focus_in_event, focus_out_event
GDK_CONFIGURE	configure_event
GDK_MAP	map_event
GDK_UNMAP	unmap_event
GDK_PROPERTY_NOTIFY	property_notify_event
GDK_SELECTION_CLEAR	selection_clear_event
GDK_SELECTION_REQUEST	selection_request_event
GDK_SELECTION_NOTIFY	selection_notify_event
GDK_PROXIMITY_IN	proximity_in_event
GDK_PROXIMITY_OUT	proximity_out_event
GDK_CLIENT_EVENT	client_event
GDK_VISIBILITY_NOTIFY	visibility_notify_event
GDK_WINDOW_STATE	window_state_event
GDK_NO_EXPOSE	no_expose_event

如果是"event"，则代表所有的事件。多个事件类型可能会对应同一个事件 Signal，GDK_BUTTON、GDK_2BUTTON、GDK_3BUTTON，在事件的 Signal 连结时，都是使用 button_press_event，在 Callback 函式中，再根据 GdkEventType 加以处理，部份的事件则有

特殊的方式处理，例如鼠标拖曳事件。

您也可以参考 **GDK Event Types**，以及 **Events** 中之说明。

1.32GTK 事件处理函式

为了连结一个事件 Signal 与 Callback 函式，一样是使用 g_signal_connect()函式，不过处理事件 Signal 的 Callback 函式与纯綷的 GTK Signal 的 Callback 函式在宣告时有些不同，以下是处理事件 Signal 的 Callback 函式宣告方式：

```
gboolean callback_func(
          GtkWidget *widget, GdkEvent *event, gpointer callback_data);
```

Callback 函式多了一个 GdkEvent*参数，而传回值的部份，可以控制事件是否进行下一步传播，传回 TRUE 表示这个事件到止已获得处理，事件不用继续传播，传回 FALSE 表示事件继续传播。

事件 Signal 的处理函式会在 GTK Signal 的处理函式之前先处理，以按下按钮为例，基本上的顺序为：

```
按钮按下 --> 发出 GDK_BUTTON_PRESS --> GDK 预设处理函式
                  --> 发出 button_press_event Signal --> GTK 预设处理函式
         --> 发出 clicked Signal --> GTK 预设处理函式
```

您可以设置事件 Signal 的 Callback 函式，拦截 button_press_event，当处理完传回 TRUE 时，就不会继续预设的 GTK 处理函式，也就不会发出 clicked 的 Signal，只有在传回 FALSE 时，才会发出 clicked 的 Signal，则设置的 GTK Signal 处理函式才会被执行。

以下这个例子为例：

event_demo.c

```
#include <gtk/gtk.h>

// Signal 处理函式
void button_clicked_callback(GtkWidget *button, gpointer data) {
    g_print("clicked 处理函式\n");
}

// 事件处理函式
gboolean button_press_callback(
              GtkWidget *button, GdkEvent *event, gpointer data) {
    GdkEventType type = event->type;
```

```
    if(type == GDK_BUTTON_PRESS) {
        g_print("button_press_event 处理函式(%d, %d)\n",
                (gint) event->button.x, (gint) event->button.y);
    }

    return FALSE;
}
int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *button;

    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "哈啰！ GTK+！ ");

    button = gtk_button_new_with_label("按我");
    gtk_container_add(GTK_CONTAINER(window), button);

    g_signal_connect(GTK_OBJECT(button), "event",
                     G_CALLBACK(button_press_callback), NULL);
    g_signal_connect(GTK_OBJECT(button), "clicked",
                     G_CALLBACK(button_clicked_callback), NULL);
    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show(window);
    gtk_widget_show(button);

    gtk_main();
    return 0;
}
```

在上面的程序中，以 button_clicked_callback()函式来处理事件，因为 g_signal_connect()中设定为"event"，表示所有事件都会经过 button_press_callback 处理，所以函式中使用 if 判断 GdkEventType，只有当鼠标按下时，显示鼠标的位置，最后传回 FALSE，执行结果如下所示：

```
button_press_event 处理函式(58, 44)
clicked 处理函式
button_press_event 处理函式(134, 108)
clicked 处理函式
button_press_event 处理函式(66, 149)
clicked 处理函式
button_press_event 处理函式(146, 44)
clicked 处理函式
```

如果 button_clicked_callback()函式传回 TRUE，则 button_clicked_callback()函式将不执行，"clicked 处理函式"文字将不会显示。
如果您已知要处理特定的事件类型，则您可以在 Callback 函式上宣告特定的事件类型，并在 g_signal_connect()时指定特定的事件 Signal，例如上面的范例也可以修改为以下，而执行结果相同：

event_demo.c

```
#include <gtk/gtk.h>

// 自订 Callback 函式
void button_clicked_callback(GtkWidget *button, gpointer data) {
    g_print("clicked 处理函式\n");
}

gboolean button_press_callback(
                GtkWidget *button, GdkEventButton *event, gpointer data) {
    g_print("button_press_event 处理函式(%d, %d)\n",
            (gint) event->x, (gint) event->y);

    return FALSE;
}

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *button;

    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "哈啰！ GTK+！ ");

    button = gtk_button_new_with_label("按我");
    gtk_container_add(GTK_CONTAINER(window), button);

    g_signal_connect(GTK_OBJECT(button), "button_press_event",
                     G_CALLBACK(button_press_callback), NULL);
    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);
    g_signal_connect(GTK_OBJECT(button), "clicked",
                     G_CALLBACK(button_clicked_callback), NULL);

    gtk_widget_show(window);
    gtk_widget_show(button);
```

```
    gtk_main();

    return 0;
}
```

1.33 件屏蔽（Event Mask）

事件屏蔽决定了一个 Widget 会接收到哪些事件，不同的 Widget 会有不同预设的事件屏蔽，您可以使用 gtk_widget_set_events()来设定事件屏蔽：

```
void gtk_widget_set_events(GtkWidget *widget, gint events);
```

使用 gtk_widget_set_events()要特别小心，因为您直接改变了 Widget 的事件屏蔽，不小心的话有可能破坏了 Widget 的功能，您可以使用 gtk_widget_events()增加一些特别事件的事件屏蔽：

```
void gtk_widget_add_events(GtkWidget *widget, gint events);
```

gtk_widget_set_events()或 gtk_widget_events()必须在 Widget 实现之前呼叫才有作用，events 是 GdkEventMask 的 enum：

```
typedef enum
{
  GDK_EXPOSURE_MASK        = 1 << 1,
  GDK_POINTER_MOTION_MASK    = 1 << 2,
  GDK_POINTER_MOTION_HINT_MASK    = 1 << 3,
  GDK_BUTTON_MOTION_MASK    = 1 << 4,
  GDK_BUTTON1_MOTION_MASK    = 1 << 5,
  GDK_BUTTON2_MOTION_MASK    = 1 << 6,
  GDK_BUTTON3_MOTION_MASK    = 1 << 7,
  GDK_BUTTON_PRESS_MASK        = 1 << 8,
  GDK_BUTTON_RELEASE_MASK    = 1 << 9,
  GDK_KEY_PRESS_MASK        = 1 << 10,
  GDK_KEY_RELEASE_MASK        = 1 << 11,
  GDK_ENTER_NOTIFY_MASK        = 1 << 12,
  GDK_LEAVE_NOTIFY_MASK        = 1 << 13,
  GDK_FOCUS_CHANGE_MASK        = 1 << 14,
  GDK_STRUCTURE_MASK        = 1 << 15,
  GDK_PROPERTY_CHANGE_MASK    = 1 << 16,
  GDK_VISIBILITY_NOTIFY_MASK    = 1 << 17,
  GDK_PROXIMITY_IN_MASK        = 1 << 18,
  GDK_PROXIMITY_OUT_MASK    = 1 << 19,
```

```
  GDK_SUBSTRUCTURE_MASK         = 1 << 20,
  GDK_SCROLL_MASK                = 1 << 21,
  GDK_ALL_EVENTS_MASK         = 0x3FFFFE
} GdkEventMask;
```

事件屏蔽与事件的对应如下（取自 **Events** 表格三）：

屏蔽	事件类型
GDK_EXPOSURE_MASK	GDK_EXPOSE
GDK_POINTER_MOTION_MASK	GDK_MOTION_NOTIFY
GDK_POINTER_MOTION_HINT_MASK	N/A
GDK_BUTTON_MOTION_MASK	GDK_MOTION_NOTIFY (鼠标按钮按下)
GDK_BUTTON1_MOTION_MASK	GDK_MOTION_NOTIFY (鼠标按钮 1 按下)
GDK_BUTTON2_MOTION_MASK	GDK_MOTION_NOTIFY (鼠标按钮 2 按下)
GDK_BUTTON3_MOTION_MASK	GDK_MOTION_NOTIFY (鼠标按钮 3 按下)
GDK_BUTTON_PRESS_MASK	GDK_BUTTON_PRESS, GDK_2BUTTON_PRESS, GDK_3BUTTON_PRESS
GDK_BUTTON_RELEASE_MASK	GDK_BUTTON_RELEASE
GDK_KEY_PRESS_MASK	GDK_KEY_PRESS
GDK_KEY_RELEASE_MASK	GDK_KEY_RELEASE
GDK_ENTER_NOTIFY_MASK	GDK_ENTER_NOTIFY
GDK_LEAVE_NOTIFY_MASK	GDK_LEAVE_NOTIFY
GDK_FOCUS_CHANGE_MASK	GDK_FOCUS_IN, GDK_FOCUS_OUT
GDK_STRUCTURE_MASK	GDK_CONFIGURE, GDK_DESTROY, GDK_MAP, GDK_UNMAP
GDK_PROPERTY_CHANGE_MASK	GDK_PROPERTY_NOTIFY
GDK_VISIBILITY_NOTIFY_MASK	GDK_VISIBILITY_NOTIFY
GDK_PROXIMITY_IN_MASK	GDK_PROXIMITY_IN
GDK_PROXIMITY_OUT_MASK	GDK_PROXIMITY_OUT
GDK_SUBSTRUCTURE_MASK	对子窗口接收 GDK_STRUCTURE_MASK 事件
GDK_ALL_EVENTS_MASK	所有事件

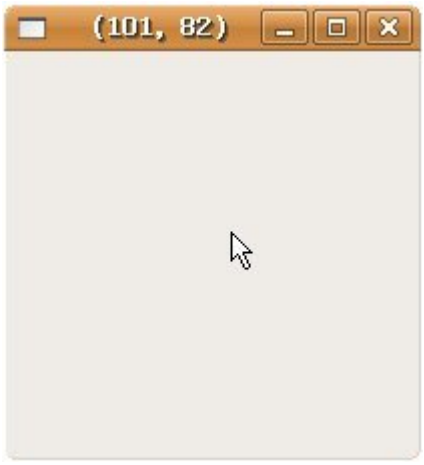
举个实际的例子来说，GtkWindow 预设是不接收鼠标移动事件，您要使用 gtk_widget_events() 增加 GDK_POINTER_MOTION 屏蔽，才可以捕捉鼠标移动事件，例如下面的程序在鼠标于窗口中移动时，将在标题列中显示目前的坐标值：

motion.c

```
#include <gtk/gtk.h>

gboolean motion_event_handler(
                GtkWidget *widget, GdkEventMotion *event, gpointer data) {
    char pos[20];
```

```
	sprintf(pos, "(%d, %d)", (int) event->x, (int) event->y);
	gtk_window_set_title(GTK_WINDOW(widget), pos);

	return FALSE;
}

int main(int argc, char *argv[]) {
	GtkWidget *window;

	gtk_init(&argc, &argv);
	window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

	g_signal_connect(GTK_OBJECT(window), "destroy",
				G_CALLBACK(gtk_main_quit), NULL);

	gtk_widget_add_events(window, GDK_POINTER_MOTION_MASK);

	g_signal_connect(GTK_OBJECT(window), "motion_notify_event",
				G_CALLBACK(motion_event_handler), NULL);

	gtk_widget_show(window);

	gtk_main();

	return 0;
}
```

一个执行的画面如下所示：



1.4 基本版面配置

1.41GtkHBox 与 GtkVBox

设计窗口程序的人都知道，在窗口程序中最麻烦也最难的就是版面配置，每次都为了组件的位置摆放在伤脑筋，目前为止在 自订 callback 函式 看到的范例，，仅曾经单纯的将

GtkButton 置放入 GtkContainer 之中，GtkButton 预设填满整个窗口，若是有多个组件，组件位置是否会适当的自我调整大小、位置（或像是字号自动调整之类的），以配合窗口缩放展现适当的观感等，这些都是版面配置的议题。

窗口程序的解决方案都会提供一些现成的版面配置方式，让您可以不必自行配置组件位置，在 GTK 中，基本版面配置可以透过 GtkBox 这个 Widget 来进行，这是个不可视的(Invisible)组件，可以容纳其它 Widget，其继承架构关系如下：

```
GObject
 +----GInitiallyUnowned
       +----GtkObject
             +----GtkWidget
                   +----GtkContainer
                         +----GtkBox
                               +----GtkVBox
                               +----GtkHBox
                               +----GtkButtonBox
```

GtkBox 继承自 GtkContainer，有两个主要子类别 GtkVBox 与 GtkHBox，而 GtkButtonBox 与前两者类似，主要是作为按钮群组版面配置使用。

以下先说明 GtkVBox 与 GtkHBox 的使用，基本上两者使用是相似的，要建立一个 GtkHBox，可以如下撰写：

```
GtkWidget *hbox = gtk_hbox_new(TRUE, 5);
```

第一个参数决定 GtkHBox 中所有的组件是否平均分配空间，第二个参数则设定两个组件之间的空间，单位是像素(Pixel)。若要将组件加入 GtkHBox 中，则要使用 gtk_box_pack_start()或 gtk_box_pack_end()，前者将组件从 GtkHBox 的左边开始加入（如果是 GtkVBox 就是从上面），后者则加至右边（如果是 GtkVBox 就是从下面），例如：

```
GtkWidget *spinButton = gtk_spin_button_new_with_range(0.0, 100.0, 1.0);

gtk_box_pack_start(GTK_BOX(hbox), spinButton, TRUE, TRUE, 5);
```

第一个布尔参数设定 Widget 是否使用所有的可用空间，设定为 TRUE 时，Widget 的可用空间会随着 GtkHBox 大小改变而改变（但 Widget 组件本身不变）。第二个参数只有在第一个参数为 TRUE 时才有用，可设定 Widget 是否填满可用空间，设定为 TRUE 时，Widget 的大小会随 GtkHBox 大小改变而改变。

以下直接看例子，使用 GtkHBox 进行组件的版面配置，您以水平的方式来摆放组件：

gtkhbox_demo.c

```
#include <gtk/gtk.h>
```

```
void value_changed_callback(GtkSpinButton *spinButton, gpointer data) {
	gint value = gtk_spin_button_get_value_as_int(spinButton);
	GString *text = g_string_new("");
	g_string_sprintf(text, "%d", value);
	gtk_label_set_text(GTK_LABEL(data), text->str);
}

int main(int argc, char *argv[]) {
	GtkWidget *window;
	GtkWidget *spinButton;
	GtkWidget *label;
	GtkWidget *hbox;

	gtk_init(&argc, &argv);

	window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
	gtk_window_set_title(GTK_WINDOW(window), "GtkHBox");
	gtk_window_set_default_size(GTK_WINDOW(window), 200, 50);

	spinButton = gtk_spin_button_new_with_range(0.0, 100.0, 1.0);
	label = gtk_label_new("0");
	hbox = gtk_hbox_new(TRUE, 5);

	gtk_box_pack_start(GTK_BOX(hbox), spinButton, TRUE, TRUE, 5);
	gtk_box_pack_start(GTK_BOX(hbox), label, TRUE, TRUE, 5);
	gtk_container_add(GTK_CONTAINER(window), hbox);

	g_signal_connect(GTK_OBJECT(spinButton), "value_changed",
				G_CALLBACK(value_changed_callback), label);
	g_signal_connect(GTK_OBJECT(window), "destroy",
				G_CALLBACK(gtk_main_quit), NULL);

	gtk_widget_show_all(window);

	gtk_main();

	return 0;
}
```

程序中使用到 GtkSpinButton 与 GtkLabel，并作了适当的 Signal，让您可以设定数字并显示在另一个文字组件上，GtkSpinButton 与 GtkLabel 之后还会介绍，现在请注意粗体字有关版面配置的程序代码即可，一个执行的画面如下所示：

根据 gtk_box_pack_start()的设定，组件会自动填满窗口，如果您拉动窗口，则当中的组件也会适当的变动大小：

如果使用 GtkVBox 来改写上面的范例：

gtkvbox_demo.c

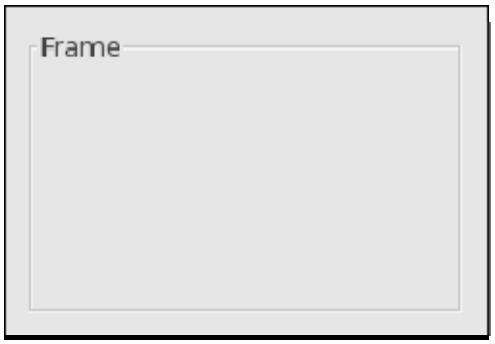
```
#include <gtk/gtk.h>

void value_changed_callback(GtkSpinButton *spinButton, gpointer data) {
	gint value = gtk_spin_button_get_value_as_int(spinButton);
	GString *text = g_string_new("");
	g_string_sprintf(text, "%d", value);
	gtk_label_set_text(GTK_LABEL(data), text->str);
}

int main(int argc, char *argv[]) {
	GtkWidget *window;
	GtkWidget *spinButton;
	GtkWidget *label;
	GtkWidget *vbox;

	gtk_init(&argc, &argv);

	window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
	gtk_window_set_title(GTK_WINDOW(window), "GtkVBox");
	gtk_window_set_default_size(GTK_WINDOW(window), 200, 50);

	spinButton = gtk_spin_button_new_with_range(0.0, 100.0, 1.0);
	label = gtk_label_new("0");
	vbox = gtk_vbox_new(TRUE, 5);

	gtk_box_pack_start(GTK_BOX(vbox), spinButton, TRUE, TRUE, 5);
	gtk_box_pack_start(GTK_BOX(vbox), label, TRUE, TRUE, 5);
	gtk_container_add(GTK_CONTAINER(window), vbox);

	g_signal_connect(GTK_OBJECT(spinButton), "value_changed",
```

```
                    G_CALLBACK(value_changed_callback), label);
    g_signal_connect(GTK_OBJECT(window), "destroy",
                    G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

一个执行画面如下所示：

1.42GtkFrame 与 GtkButtonBox

GtkFrame 也是 GtkContainer 的子类，可以容纳其它 Widget，并呈现出以下的外观（图片取自 GtkFrame 文件）：



要建立 GtkFrame，并将组件置于其中，基本上是如下的程序代码片段：

```
GtkWidget *gtkFrame = gtk_frame_new("GtkHButtonBox");
gtk_container_add(GTK_CONTAINER(gtkFrame), hbuttonBox);
```

GtkButtonBox 则是一个可以容纳按钮群组的不可视版面配置组件，您可以使用 gtk_hbutton_box_new()函式建立一个水平置放按钮的 GtkButtonBox，或是使用 gtk_vbutton_box_new()建立一个垂直建立按钮的 GtkButtonBox，在建立 GtkButtonBox 之后，您可以使用 gtk_button_box_set_layout()来设置按钮的排列方式，依给定的 GtkButtonBoxStyle 进行设定：

```
typedef enum
```

```
{
  GTK_BUTTONBOX_DEFAULT_STYLE,
  GTK_BUTTONBOX_SPREAD,
  GTK_BUTTONBOX_EDGE,
  GTK_BUTTONBOX_START,
  GTK_BUTTONBOX_END,
  GTK_BUTTONBOX_CENTER
} GtkButtonBoxStyle;
```

下面的程序代码使用了 GtkBox、GtkButtonBox 及 GtkFrame 进行了较复杂的版面配置，要注意的是组件加入的顺序，以及设定不同 GtkButtonBoxStyle 后的呈现方式：

layout_demo.c

```
#include <gtk/gtk.h>

// 建立水平按钮群组
GtkWidget* createHButtonBoxWithFrame() {
    int i;
    GtkWidget *hbuttonBox = gtk_hbutton_box_new();
    GtkWidget *gtkFrame = gtk_frame_new("GtkHButtonBox");
    gtk_container_add(GTK_CONTAINER(gtkFrame), hbuttonBox);

    gtk_button_box_set_layout(GTK_BUTTON_BOX(hbuttonBox),
                                    GTK_BUTTONBOX_SPREAD);
    for(i = 0; i < 4; i++) {
        gtk_box_pack_start(GTK_BOX(hbuttonBox),
            gtk_button_new_with_label("HBtn"), TRUE, TRUE, 5);
    }

    return gtkFrame;
}

// 建立垂直按钮群组
GtkWidget* createVButtonBoxWithFrame() {
    int i;
    GtkWidget *vbuttonBox = gtk_vbutton_box_new();
    GtkWidget *gtkFrame = gtk_frame_new("GtkVButtonBox");
    gtk_container_add(GTK_CONTAINER(gtkFrame), vbuttonBox);

    gtk_button_box_set_layout(GTK_BUTTON_BOX(vbuttonBox),
                                    GTK_BUTTONBOX_START);
    for(i = 0; i < 4; i++) {
        gtk_box_pack_start(GTK_BOX(vbuttonBox),
```

```
                gtk_button_new_with_label("VBtn"), TRUE, TRUE, 5);
        }

        return gtkFrame;
}

int main(int argc, char *argv[]) {
        GtkWidget *window;
        GtkWidget *vbox;
        GtkWidget *hbox;
        GtkWidget *gtkFrame1, *gtkFrame2;

        gtk_init(&argc, &argv);

        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        gtk_window_set_title(GTK_WINDOW(window), "GtkFrame & GtkButtonBox");
        gtk_window_set_default_size(GTK_WINDOW(window), 500, 400);

        // 最外层为 GtkFrame，当中包括 GtkVBox
        vbox = gtk_vbox_new(TRUE, 5);
        gtkFrame1 = gtk_frame_new("GtkVBox");
        gtk_container_add(GTK_CONTAINER(gtkFrame1), vbox);

        // 建立水平按钮群组，加入 GtkVBox
        gtk_box_pack_start(GTK_BOX(vbox),
                createHButtonBoxWithFrame(), TRUE, TRUE, 5);

        // 建立 GtkFrame，当中包括 GtkHBox，加入 GtkVBox 中
        hbox = gtk_hbox_new(TRUE, 5);
        gtkFrame2 = gtk_frame_new("GtkHBox");
        gtk_container_add(GTK_CONTAINER(gtkFrame2), hbox);
        gtk_box_pack_start(GTK_BOX(vbox), gtkFrame2, TRUE, TRUE, 5);

        // 分别建立两个垂直按钮群组，加入 GtkHBox 中
        gtk_box_pack_start(GTK_BOX(hbox),
                createVButtonBoxWithFrame(), TRUE, TRUE, 5);
        gtk_box_pack_start(GTK_BOX(hbox),
                createVButtonBoxWithFrame(), TRUE, TRUE, 5);

        gtk_container_add(GTK_CONTAINER(window), gtkFrame1);

        g_signal_connect(GTK_OBJECT(window), "destroy",
                                G_CALLBACK(gtk_main_quit), NULL);
```

```
    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

执行的成果如下所示：

1.43GtkTable

在进行组件排版的时候，将组件以类似表格的方式排放也是很常见的排版方式，在 GTK 中，您可以使用 GtkTable，您可以使用 gtk_table_new()函式来建立：

```
GtkWidget *table = gtk_table_new(3, 3, TRUE);
```

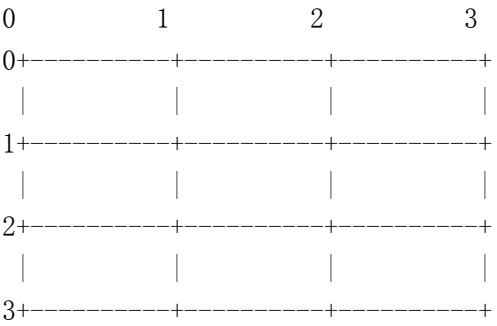
上面的程序代码片段也建立 3 列（row）、3 行（column）的表格，第三个参数则设定表格中的空间是否平均分配，所以若设定为 TRUE，每一格的空间将取决于置于其中最大的组件。

要将组件置于表格之中，可以使用 gtk_table_attach()函式：

```
void gtk_table_attach( GtkTable         *table,
                       GtkWidget        *child,
```

```
                        guint            left_attach,
                        guint            right_attach,
                        guint            top_attach,
                        guint            bottom_attach,
                        GtkAttachOptions xoptions,
                        GtkAttachOptions yoptions,
                        guint            xpadding,
                        guint            ypadding );
```

这个函式中的 left_attach、right_attach、top_attach、bottom_attach，决定了组件将占据的空间，例如若是 3X3 表格：



若要让组件占据左上格空间，则 left_attach 为 0、right_attach 为 1、top_attach 为 0、bottom_attach 为 1，若要让组件占据右下格空间，则 left_attach 为 1、right_attach 为 2、top_attach 为 1、bottom_attach 为 2，若要让组件占据底下两格空间，则 left_attach 为 0、right_attach 为 2、top_attach 为 1、bottom_attach 为 2，依此类推。

xoptions 与 yoptions 为组件占据空间的方式，可以指定以下的值，可以使用 OR 结合值：

GTK_FILL：若组件原本小于可用空间，则组件会填满可用空间。
GTK_SHRINK：若组件原本大于可用空间，则组件会缩小以符合可用空间。
GTK_EXPAND：表格会扩展以符合组件大小。

您可以使用 gtk_table_attach_defaults()函式，预设选项为 GTK_FILL | GTK_EXPAND，padding 都设为 0：

```
void gtk_table_attach_defaults( GtkTable  *table,
                                GtkWidget *widget,
                                guint      left_attach,
                                guint      right_attach,
                                guint      top_attach,
                                guint      bottom_attach );
```

下面的程序先示范简单的 GtkTable 使用方式：

gtk_table_demo.c

```
#include <gtk/gtk.h>

int main(int argc, char *argv[]) {
	GtkWidget *window;
	GtkWidget *table;
	GtkWidget *label;

	const char *text[] = {"One", "Two", "Three",
				"Four", "Five", "Six",
				"Seven", "Eight", "Nine"};
	int i, j, k;

	gtk_init(&argc, &argv);

	window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
	gtk_window_set_title(GTK_WINDOW(window), "GtkTable");
	gtk_window_set_default_size(GTK_WINDOW(window), 250, 150);

	table = gtk_table_new(3, 3, TRUE);
	for(i = 0, k = 0; i < 3; i++, k = k + 3) {
		for(j = 0; j < 3; j++) {
			label = gtk_label_new(text[k + j]);
			gtk_table_attach_defaults(
				GTK_TABLE(table), label, j, j + 1, i, i + 1);
		}
	}

	gtk_container_add(GTK_CONTAINER(window), table);

	g_signal_connect(GTK_OBJECT(window), "destroy",
				G_CALLBACK(gtk_main_quit), NULL);

	gtk_widget_show_all(window);

	gtk_main();

	return 0;
}
```

程序执行时的参考画面如下：

GtkTable 的 left_attach、right_attach、top_attach、bottom_attach 指定方式，可以让您方便的达到组件跨数格的方式，在 Table Packing Example 中有个例子，您也可以再参考 Packing Using Tables 中有关 GtkTable 的说明。

第二章 基本图形组件

2.1 按钮

2.11GtkButton 与 GtkToggleButton

在 自订 callback 函式 中使用过 GtkButton 作为范例，最基本的 GtkButton 建立方式是透过 gtk_button_new_with_label()来建立一个指定文字的按钮，您也可以使用 gtk_button_new_with_mnemonic()，这会建立一个具有提示底线及快捷键功能的按钮，例如将 自订 callback 函式 范例的 gtk_button_new_with_label()该行换成 gtk_button_new_with_mnemonic()：

```
button = gtk_button_new_with_mnemonic("_Press");
```

指定的按钮文字是以底线作为开头，这会在显示的按钮文字上以「Press」的方式呈现，并且按下 Alt+P 时即可触发按钮 Clicked，执行时的画面如下：

若想要制作双态按钮（Toggle Button），则可以使用 GtkToggleButton，您可以

gtk_toggle_button_new_with_label()、gtk_toggle_button_new_with_mnemonic()函式来建立组件，双态按钮有停驻及非停驻两种状态，可以使用 gtk_toggle_button_get_active()取得双态按钮的状态，也可以透过 gtk_toggle_button_set_active()直接设定按钮为停驻或非停驻状态，双态按钮所引发的 Signal 是"toggled"，若是使用 gtk_toggle_button_set_active()改变了停驻状态，则会同时引发"clicked"及"toggled"两个 Signal。

下面这个程序示范了 GtkToggleButton 的外观与 Signal 处理：

gtk_toggle_button_demo.c

```
#include <gtk/gtk.h>

void toggled_callback(GtkToggleButton *toggleBtn, gpointer data) {
    if (gtk_toggle_button_get_active(toggleBtn)) {
        gtk_button_set_label(GTK_BUTTON(toggleBtn), "停驻");
    } else {
        gtk_button_set_label(GTK_BUTTON(toggleBtn), "未停驻");
    }
}

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *toggleBtn;
    GtkWidget *hbox;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GtkToggleButton");
    gtk_window_set_default_size(GTK_WINDOW(window), 250, 50);

    toggleBtn = gtk_toggle_button_new_with_label("停驻");
    gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(toggleBtn), TRUE);

    gtk_container_add(GTK_CONTAINER(window), toggleBtn);

    g_signal_connect(GTK_OBJECT(toggleBtn), "toggled",
                     G_CALLBACK(toggled_callback), NULL);
    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();
```

```
    return 0;
}
```

以下为执行时的画面参考：

2.12 影像及文字按钮

您可以建立具备影像及文字的按钮，最基本的方式可以透过 gtk_button_new_from_stock()，这可以让您从 GTK 既有的影像资源中取用影像给按钮使用，gtk_button_new_from_stock()使用时的参数与出现的影像，可在 Stock Items 查询。

以实际的例子来说，将 自订 callback 函式 中的按钮产生程序片段改为以下：

```
button = gtk_button_new_from_stock(GTK_STOCK_YES);
```

则程序执行时的参考画面如下所示：

另一个为按钮创立影像及文字的方式，是使用 gtk_button_new()函式建立一个按钮组件，利用 GtkHBox

与 GtkVBox 建立 Box 并内含图片及文字，再将这个 Box 透过 gtk_container_add()加入建立的按钮组件之

中，下面这个程序是一个简单的示范：

button_with_image_demo.c

```
#include <gtk/gtk.h>

GtkWidget* buttonWithImage(gchar *filename, gchar *text) {
	GtkWidget *box;
	GtkWidget *label;
	GtkWidget *image;

	box = gtk_hbox_new(FALSE, 5);
	gtk_container_set_border_width(GTK_CONTAINER(box), 2);

	image = gtk_image_new_from_file(filename);
	label = gtk_label_new(text);

	gtk_box_pack_start(GTK_BOX(box), image, FALSE, FALSE, 5);
	gtk_box_pack_start(GTK_BOX(box), label, FALSE, FALSE, 5);

	gtk_widget_show (image);
	gtk_widget_show (label);

	return box;
}

int main(int argc, char *argv[]) {
	GtkWidget *window;
	GtkWidget *button;
	GtkWidget *box;

	gtk_init(&argc, &argv);
	window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
	gtk_window_set_title(GTK_WINDOW(window), "哈啰！ GTK+！ ");
	gtk_window_set_default_size(GTK_WINDOW(window), 200, 50);

	button = gtk_button_new();
	box = buttonWithImage("caterpillar.gif", "caterpillar");

	gtk_container_add(GTK_CONTAINER(button), box);
	gtk_container_add(GTK_CONTAINER(window), button);

	g_signal_connect(GTK_OBJECT(window), "destroy",
				G_CALLBACK(gtk_main_quit), NULL);
```

```
    gtk_widget_show(window);
    gtk_widget_show(button);
    gtk_widget_show(box);

    gtk_main();

    return 0;
}
```

执行的结果画面如下所示：

2.13GtkCheckButton 与 GtkRadioButton

核取钮（Check Button）是可以进行选项复选的组件，单选钮（Radio Button）是只能进行选项单选的组件，在 GTK 中分别使用 GtkCheckButton 与 GtkRadioButton 来负责，它们是双态按钮 GtkToggleButton 的子类：

```
GtkToggleButton
    +----GtkCheckButton
    +----GtkRadioButton
```

建立核取钮或单选钮的方式与建立一般按钮类似，您可以使用 gtk_check_button_new()、gtk_check_button_new_with_label()、gtk_check_button_new_with_mnemonic()，来建立核取钮，GtkRadioButton 必须使用按钮群组（Button Group）来加以群组，来表示哪些单选项为一个群组，彼此互斥，同时间只能选择一个，例如 gtk_radio_button_new_with_label()函式：

```
GtkWidget *gtk_radio_button_new_with_label(GSList *group, const gchar  *label);
```

第一次建立 GtkRadioButton 时，GSList 的参数直接设为 NULL 即可，接着使用 gtk_radio_button_get_group()获得按钮群组;

```
GSList *gtk_radio_button_get_group( GtkRadioButton *radio_button);
```

例如以下的程序代码会先建立一个 GtkRadioButton，接着从已建立的 GtkRadioButton 获得按钮群组，然后再用以建立另一个 GtkRadioButton，如此两个单选钮就属于同一个群组，

同时间只能选取一个：

```
GtkWidget *radio1 = gtk_radio_button_new_with_label(NULL, "Linux");
GtkWidget *radio2 = gtk_radio_button_new_with_label(
            gtk_radio_button_get_group(GTK_RADIO_BUTTON(radio1)) , "Windows");
```

您也可以使用 gtk_radio_button_new_with_label_from_widget()，这可以省略 gtk_radio_button_get_group()这道手续，例如：

```
GtkWidget *radio1 = gtk_radio_button_new_with_label(NULL, "Linux");
GtkWidget *radio2 = gtk_radio_button_new_with_label_from_widget(
                        GTK_RADIO_BUTTON(radio1) , "Windows");
```

由于 GtkCheckButton 与 GtkRadioButton 都属于 GtkToggleButton 的子类，如果要设定按钮为选取状态，则直接使用 gtk_toggle_button_set_active()函式即可。

下面的程序为 GtkCheckButton 与 GtkRadioButton 配置的基本示范：

check_radio_demo.c

```
#include <gtk/gtk.h>

GtkWidget* checkButtonsNew() {
    GtkWidget *check1, *check2;
    GtkWidget *vbox;

    check1 = gtk_check_button_new_with_label("Java is good!");
    check2 = gtk_check_button_new_with_label("C++ is good!");

    vbox = gtk_vbox_new(TRUE, 5);
    gtk_box_pack_start(GTK_BOX(vbox), check1, TRUE, TRUE, 5);
    gtk_box_pack_start(GTK_BOX(vbox), check2, TRUE, TRUE, 5);

    return vbox;
}

GtkWidget* radioButtonsNew() {
    GtkWidget *radio1, *radio2, *radio3;
    GtkWidget *vbox;

    radio1 = gtk_radio_button_new_with_label(NULL, "Linux");
    radio2 = gtk_radio_button_new_with_label_from_widget(
                    GTK_RADIO_BUTTON(radio1) , "Windows");
```

```
        radio3 = gtk_radio_button_new_with_label_from_widget(
                        GTK_RADIO_BUTTON(radio1) , "Mac");

        vbox = gtk_vbox_new(TRUE, 5);
        gtk_box_pack_start(GTK_BOX(vbox), radio1, TRUE, TRUE, 5);
        gtk_box_pack_start(GTK_BOX(vbox), radio2, TRUE, TRUE, 5);
        gtk_box_pack_start(GTK_BOX(vbox), radio3, TRUE, TRUE, 5);

        return vbox;
}

int main(int argc, char *argv[]) {
        GtkWidget *window;
        GtkWidget *vbox;
        GtkWidget *frame;

        gtk_init(&argc, &argv);
        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        gtk_window_set_title(GTK_WINDOW(window), "GtkCheckButton & GtkRadioButton");
        gtk_window_set_default_size(GTK_WINDOW(window), 400, 50);

        vbox = gtk_vbox_new(FALSE, 5);

        gtk_box_pack_start(GTK_BOX(vbox), checkButtonsNew(), TRUE, TRUE, 5);

        frame = gtk_frame_new("Favorite OS");
        gtk_container_add(GTK_CONTAINER(frame), radioButtonsNew());

        gtk_box_pack_start(GTK_BOX(vbox), frame, TRUE, TRUE, 5);

        gtk_container_add(GTK_CONTAINER(window), vbox);

        g_signal_connect(GTK_OBJECT(window), "destroy",
                                G_CALLBACK(gtk_main_quit), NULL);

        gtk_widget_show_all(window);

        gtk_main();

        return 0;
}
```

执行的画面如下所示：

2.2 对话框

2.21GtkMessageDialog

在程序中常出现一些简单的对话或消息框，有 GTK 中，它们都是 GtkDialog 的子类，在这边先介绍一些 GTK 内建的对话框，常见类型之一是 GtkMessageDialog，首先看个简单的范例，改写 自订 callback 函式 中的例子，按下按钮后会出现消息框：

message_dialog_demo.c

```
#include <gtk/gtk.h>

void button_clicked(GtkWidget *button, gpointer data) {
    GtkWidget *dialog = gtk_message_dialog_new(GTK_WINDOW(data),
                                    GTK_DIALOG_MODAL, GTK_MESSAGE_INFO,
                                    GTK_BUTTONS_OK, "良葛格学习笔记");
    gtk_window_set_title(GTK_WINDOW(dialog), "GtkMessageDialog");
    gtk_message_dialog_format_secondary_text(
        GTK_MESSAGE_DIALOG(dialog), "http://caterpillar.onlyfun.net");
    gtk_dialog_run(GTK_DIALOG(dialog));
    gtk_widget_destroy(dialog);
}

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *button;

    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GtkMessageDialog");

    button = gtk_button_new_with_label("按我");
```

```
    gtk_container_add(GTK_CONTAINER(window), button);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);
    g_signal_connect(GTK_OBJECT(button), "clicked",
                     G_CALLBACK(button_clicked), window);

    gtk_widget_show(window);
    gtk_widget_show(button);

    gtk_main();

    return 0;
}
```

gtk_message_dialog_new()的宣告如下：

```
GtkWidget* gtk_message_dialog_new(GtkWindow *parent,
                                  GtkDialogFlags flags,
                                  GtkMessageType type,
                                  GtkButtonsType buttons,
                                  const gchar *message_format,
                                  ...);
```

设定 parent，当消息框是独占式（Modal）时，若不响应消息框，则无法操作其父窗口，GtkDialogFlags 则可以设定：

GTK_DIALOG_MODAL：设定为独占窗口。
GTK_DIALOG_DESTROY_WITH_PARENT：如果 parent 被销毁，则一并销毁对话框。
GTK_DIALOG_NO_SEPARATOR：在对话框中不出现分隔线，在 使用 GtkDialog 自订对话框 中会再介绍。

GtkMessageType 则指定这是哪个类型的讯息：

GTK_MESSAGE_INFO
GTK_MESSAGE_WARNING
GTK_MESSAGE_QUESTION
GTK_MESSAGE_ERROR
GTK_MESSAGE_OTHER

GtkButtonsType 则可以设定几个预设的按钮类型：

GTK_BUTTONS_NONE
GTK_BUTTONS_OK

GTK_BUTTONS_CLOSE
GTK_BUTTONS_CANCEL
GTK_BUTTONS_YES_NO
GTK_BUTTONS_OK_CANCEL

message_format 的设定，其实是类似 printf()的字符串格式，例如可以这么设定：

```
GtkWidget *dialog = gtk_message_dialog_new(GTK_WINDOW(data),
                                GTK_DIALOG_MODAL, GTK_MESSAGE_INFO,
                                GTK_BUTTONS_OK, "%s 已删除", filename);
```

要显示对话框，方法之一是使用 gtk_dialog_run()，这会将对话框以独占模式显示，无论其 GtkDialogFlags 设定为何，因为这会呼叫 gtk_window_set_modal()函式，将对话框设定为独占模式，当响应对话框之后，您要使用 gtk_widget_destroy()销毁对话框。

以下是一个执行时对话框的参考画面：

gtk_dialog_run()结束后会传回一个 response id，为 GtkResponseType 的 enum 值，代表使用者按下的按钮，如果对话框只是被关闭（按下 X），则会传回 GTK_RESPONSE_NONE，您可以依这个传回值来决定响应对话框后的下一步动作，例如 GtkDialog 的说明文件中有这么一段范例程序：

```
gint result = gtk_dialog_run (GTK_DIALOG (dialog));
switch (result)
  {
    case GTK_RESPONSE_ACCEPT:
       do_application_specific_something ();
       break;
    default:
       do_nothing_since_dialog_was_cancelled ();
       break;
  }
gtk_widget_destroy (dialog);
```

2.22GtkAboutDialog

GtkAboutDialog 是 GtkDialog 的子类别，提供您在制作应用程序「关于 XXX」时的内建对话框，直接以程序实例来看看它的一些功能：

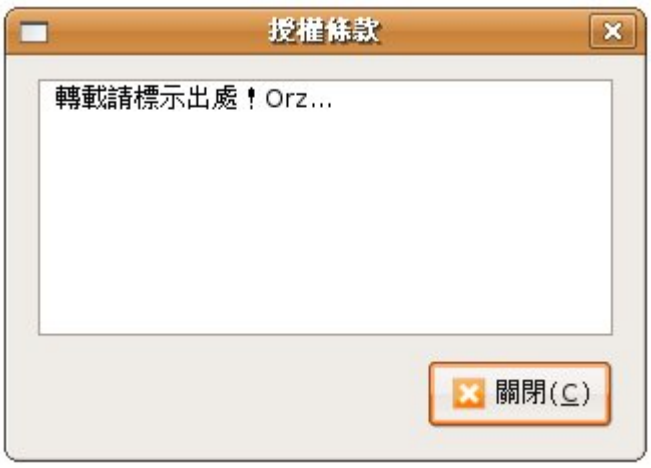
gtk_about_dialog_demo.c

```
#include <gtk/gtk.h>

void button_clicked(GtkWidget *button, gpointer data) {
    GtkWidget *dialog;
    GdkPixbuf *logo;
    GError *error = NULL;

    dialog = gtk_about_dialog_new();

    // 载入 LOGO 图档
    logo = gdk_pixbuf_new_from_file("caterpillar.gif", &error);

    if(error == NULL) {
        gtk_about_dialog_set_logo(GTK_ABOUT_DIALOG(dialog), logo);
    }
    else {
        g_print("Error: %s\n", error->message);
        g_error_free (error);
    }

    gtk_about_dialog_set_name(GTK_ABOUT_DIALOG (dialog), "良葛格学习笔记");
    gtk_about_dialog_set_version(GTK_ABOUT_DIALOG(dialog), "2.0");
    gtk_about_dialog_set_comments(
        GTK_ABOUT_DIALOG(dialog), "C/C++、Java、Open Source");
    gtk_about_dialog_set_copyright(
        GTK_ABOUT_DIALOG(dialog),"(C) 2008 caterpillar");

    gtk_about_dialog_set_license(
        GTK_ABOUT_DIALOG(dialog), "转载请标示出处！Orz...");
    gtk_about_dialog_set_website(
        GTK_ABOUT_DIALOG(dialog), "http://caterpillar.onlyfun.net");

    gtk_dialog_run(GTK_DIALOG(dialog));
    gtk_widget_destroy(dialog);
}

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *button;
```

```
    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GtkAboutDialog");

    button = gtk_button_new_with_label("按我");
    gtk_container_add(GTK_CONTAINER(window), button);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);
    g_signal_connect(GTK_OBJECT(button), "clicked",
                     G_CALLBACK(button_clicked), window);

    gtk_widget_show(window);
    gtk_widget_show(button);

    gtk_main();

    return 0;
}
```

要设定对话框的图文件，可以使用 gtk_about_dialog_set_logo()函式，这会需要 GdkPixbuf 型态的自变量，您可以使用 gdk_pixbuf_new_from_file()来加载图档，若加载时发生错误，会将错误相关信息设定给 GError 自变量，剩下的函式之作用，直接来看执行的画面即可了解：

「授权条款」按钮的功能是 gtk_about_dialog_set_license()函式所设定的，按下该钮后会出现以下的画面：



2.23GtkColorButton 与 GtkColorSelectionDialog

GtkColorButton 是一个外观为目前选择颜色的按钮，按下时会出现选取颜色的对话框，您可以使用 gtk_color_button_new()建立按钮，或是使用 gtk_color_button_new_with_color()指定初始颜色建立按钮，使用 gtk_color_button_set_color ()或 gtk_color_button_get_color()设定或取得目前的颜色，而这与一个 GdkColor 有关：

```
typedef struct {
  guint32 pixel;
  guint16 red;
  guint16 green;
  guint16 blue;
} GdkColor;
```

从 GdkColor 的成员名称上可以了解到，GdkColor 包括的有像素、红、绿、蓝信息，可指定的值为 0 到 65535，例如指定 RGB 信息的话：

```
GdkColor color;
color.red = 65535;
color.green = 65535;
color.blue = 65535;
```

如果您想要使用"#FFFFFF"这样的方式来指定颜色的话，则可以使用 gdk_color_parse()函式，例如：

```
gdk_color_parse("FF0000", &color);
```

下面这个范例，可以使用 GtkColorButton 来选择颜色，并根据所选取的颜色来改变文字的颜色，文字是使用 GtkLabel，要改变 GtkLabel 文字的颜色，则使用 gtk_widget_modify_fg() 函式：

gtk_color_button_demo.c

```
#include <gtk/gtk.h>

void color_changed(GtkColorButton *button, gpointer data) {
	GdkColor color;
	gtk_color_button_get_color(button, &color);
	gtk_widget_modify_fg(GTK_WIDGET(data), GTK_STATE_NORMAL, &color);
}

int main(int argc, char *argv[]) {
	GtkWidget *window;
	GtkWidget *label;
	GtkWidget *button;
	GtkWidget *box;
	GdkColor color;

	gtk_init(&argc, &argv);
	window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
	gtk_window_set_title(GTK_WINDOW(window), "GtkColorButton");

	gdk_color_parse("#FF0000", &color);
	label = gtk_label_new("Color me");
	gtk_widget_modify_fg(label, GTK_STATE_NORMAL, &color);
	button = gtk_color_button_new_with_color(&color);

	box = gtk_hbox_new(TRUE, 5);
	gtk_box_pack_start(GTK_BOX(box), label, TRUE, TRUE, 5);
	gtk_box_pack_start(GTK_BOX(box), button, TRUE, TRUE, 5);

	gtk_container_add(GTK_CONTAINER(window), box);

	g_signal_connect(GTK_OBJECT(window), "destroy",
				G_CALLBACK(gtk_main_quit), NULL);
	g_signal_connect(GTK_OBJECT(button), "color_set",
				G_CALLBACK(color_changed), label);

	gtk_widget_show_all(window);

	gtk_main();
```

```
    return 0;
}
```

gtk_widget_modify_fg()函式的第二个参数 GtkStateType 设定为 GTK_STATE_NORMAL，表示设定组件在一般状态的前景色。一个执行时的画面如下：

按下按钮时所出现的对话框如下：

实际上该对话框即为 GtkColorSelectionDialog，GtkColorSelectionDialog 实际上是由 GtkColorSelection 及 GtkDialog 组成，GtkColorSelectionDialog 的 colorsel 成员就是指向 GtkColorSelection。

GtkColorSelection 是一个 Widget，包括了色彩轮、色相（Hue）、彩度（Saturation）、明度（Value）、红、绿、蓝的输入设定，您可以单独立使用 GtkColorSelection，或是简单的利用 GtkColorSelectionDialog。

下面这个程序直接使用 GtkColorSelectionDialog，制作出类似上面范例的作用：

gtk_color_selection_dialog_demo.c

```
#include <gtk/gtk.h>

void color_changed(GtkButton *button, gpointer label) {
    GtkWidget *dialog =
            gtk_color_selection_dialog_new ("GtkColorSelectionDialog");
    // 取得 GtkColorSelection
    GtkWidget *colorsel = GTK_COLOR_SELECTION_DIALOG (dialog)->colorsel;
    // 取得 GtkStyle，这是为了接下来可以取得背景信息
```

```
        GtkStyle *style = gtk_widget_get_style(GTK_WIDGET(button));
        // GtkStyle 的 bg 成员可以取得背景信息
        GdkColor color = style->bg[GTK_STATE_NORMAL];

        // 设定 GtkColorSelection 出现时的目前颜色
        gtk_color_selection_set_current_color(
                GTK_COLOR_SELECTION(colorsel), &color);

        gtk_dialog_run(GTK_DIALOG(dialog));

        // 设定 GtkColorSelection 的目前颜色
       gtk_color_selection_get_current_color(
                GTK_COLOR_SELECTION(colorsel), &color);

        // 修改按钮背景色
        gtk_widget_modify_bg(GTK_WIDGET(button), GTK_STATE_NORMAL, &color);
        // 修改 GtkLabel 前景色
        gtk_widget_modify_fg(GTK_WIDGET(label), GTK_STATE_NORMAL, &color);

        gtk_widget_destroy(dialog);
}

int main(int argc, char *argv[]) {
        GtkWidget *window;
        GtkWidget *label;
        GtkWidget *button;
        GtkWidget *box;
        GdkColor color;

        gtk_init(&argc, &argv);
        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        gtk_window_set_title(GTK_WINDOW(window), "GtkColorSelectionDialog");

        gdk_color_parse("#FF0000", &color);

        label = gtk_label_new("Color me");
        gtk_widget_modify_fg(label, GTK_STATE_NORMAL, &color);

        button = gtk_button_new();
        // 修改按钮背景色
        gtk_widget_modify_bg(button, GTK_STATE_NORMAL, &color);

        box = gtk_hbox_new(TRUE, 5);
        gtk_box_pack_start(GTK_BOX(box), label, TRUE, TRUE, 5);
```

```
    gtk_box_pack_start(GTK_BOX(box), button, TRUE, TRUE, 5);
    gtk_container_add(GTK_CONTAINER(window), box);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);
    g_signal_connect(GTK_OBJECT(button), "clicked",
                     G_CALLBACK(color_changed), label);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

2.24GtkFontButton 与 GtkFontSelectionDialog

GtkFontButton 是一个外观为目前选择字型名称的按钮，按下时会出现选取字型的对话框，您可以使用 gtk_font_button_new()建立按钮，或是使用 gtk_color_button_new_with_font()指定初始字型建立按钮，指定时使用字符串指定，格式为"Family Style Size"。Family 为字型名称，例如"Arial"、"Serif"这样的名称。Style 为字型样式，可设定的样式依字型而有所不同，通常可以指定的有粗体"Bold"、斜体"Italic"、粗斜体"Bold Italic"。Size 是字型的大小。

使用 gtk_font_button_set_font_name()或 gtk_font_button_get_font_name()设定或取得目前的字型名称，如果您打算使用指定的字型来改变组件的字型外观，则可以使用 gtk_widget_modify_font()，这会需要一个 PangoFontDescription 的字型描述结构，要取得指定字型的 PangoFontDescription，可以使用 pango_font_description_from_string()并指定字型名称，例如：

```
const gchar *font = gtk_font_button_get_font_name (button);
PangoFontDescription *fontDesc = pango_font_description_from_string (font);
gtk_widget_modify_font(label, fontDesc);
```

下面这个范例，可以让您按下按钮进行字型选取，并依选择的字型改变文字的字型外观：

```
gtk_font_button_demo.c
#include <gtk/gtk.h>

void font_changed(GtkFontButton *button, gpointer label) {
    const gchar *fontName = gtk_font_button_get_font_name(button);
    PangoFontDescription *fontDesc =
```

```
                pango_font_description_from_string(fontName);
        gtk_widget_modify_font(GTK_WIDGET(label), fontDesc);
}

int main(int argc, char *argv[]) {
        GtkWidget *window;
        GtkWidget *label;
        GtkWidget *button;
        GtkWidget *box;
        PangoFontDescription *fontDesc;

        gtk_init(&argc, &argv);
        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        gtk_window_set_title(GTK_WINDOW(window), "GtkFontButton");

        label = gtk_label_new("Hello World!");
        fontDesc = pango_font_description_from_string("Courier Pitch Bold 12");
        gtk_widget_modify_font(label, fontDesc);

        button = gtk_font_button_new_with_font("Courier Pitch Bold 12");
        box = gtk_hbox_new(TRUE, 5);
        gtk_box_pack_start(GTK_BOX(box), label, TRUE, TRUE, 5);
        gtk_box_pack_start(GTK_BOX(box), button, TRUE, TRUE, 5);

        gtk_container_add(GTK_CONTAINER(window), box);

        g_signal_connect(GTK_OBJECT(window), "destroy",
                                G_CALLBACK(gtk_main_quit), NULL);
        g_signal_connect(GTK_OBJECT(button), "font_set",
                                G_CALLBACK(font_changed), label);

        gtk_widget_show_all(window);

        gtk_main();

        return 0;
}
```

执行时的外观如下：

按下按钮后的对话框外观如下所示：

改变字型后的画面如下所示：
按下按钮后出现的对话框，实际上是 GtkFontSelectionDialog，您也可以自行使用 GtkFontDialogDialog 来完成上面的范例：

gtk_font_dialog.c

```
#include <gtk/gtk.h>

void font_changed(GtkButton *button, gpointer label) {
	PangoFontDescription *fontDesc;

	GtkWidget *dialog = gtk_font_selection_dialog_new("Choose a Font");
	const gchar *fontName = gtk_button_get_label(button);
	gtk_font_selection_dialog_set_font_name(
		GTK_FONT_SELECTION_DIALOG(dialog), fontName);

	gtk_dialog_run(GTK_DIALOG(dialog));

	fontName = gtk_font_selection_dialog_get_font_name(
			GTK_FONT_SELECTION_DIALOG(dialog));
	gtk_button_set_label(button, fontName);
	fontDesc = pango_font_description_from_string(fontName);
	gtk_widget_modify_font(GTK_WIDGET(label), fontDesc);

	gtk_widget_destroy(dialog);
}
```

```
int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *label;
    GtkWidget *button;
    GtkWidget *box;
    PangoFontDescription *fontDesc;

    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GtkFontSelectionDialog");

    label = gtk_label_new("Hello World!");
    fontDesc = pango_font_description_from_string("Courier Pitch Bold 12");
    gtk_widget_modify_font(label, fontDesc);

    button = gtk_button_new_with_label("Courier Pitch Bold 12");
    box = gtk_hbox_new(TRUE, 5);
    gtk_box_pack_start(GTK_BOX(box), label, TRUE, TRUE, 5);
    gtk_box_pack_start(GTK_BOX(box), button, TRUE, TRUE, 5);

    gtk_container_add(GTK_CONTAINER(window), box);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);
    g_signal_connect(GTK_OBJECT(button), "clicked",
                     G_CALLBACK(font_changed), label);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

2.25GtkFileChooserButton 与 GtkFileChooserDialog

在窗口程序中开启档案或另存盘案的动作，会使用档案对话框来让使用者方便的选取或决定文件名称，GtkFileChooserButton 可以提供按下后出档案对话框的功能，您可以使用 gtk_file_chooser_button_new()来建立按钮，使用时必须指定 GtkFileChooserAction：

GTK_FILE_CHOOSER_ACTION_OPEN：选择档案的对话框
GTK_FILE_CHOOSER_ACTION_SAVE：储存档案的对话框

GTK_FILE_CHOOSER_ACTION_SELECT_FOLDER：选择数据夹的对话框
GTK_FILE_CHOOSER_ACTION_CREATE_FOLDER：建立数据夹的对话框

GtkFileChooserButton 实作了 GtkFileChooser 接口，您可以使用
gtk_file_chooser_set_filename()设定对话框目前的文件名称，使用
gtk_file_chooser_get_filename()取得目前的文件名称，使用
gtk_file_chooser_set_current_folder()设定目前的目录名称等。

出现的对话框可以设定文件名过滤，这需要 GtkFileFilter，可使用 gtk_file_filter_new()建立，例如：

```
GtkFileFilter *filter1 = gtk_file_filter_new();
GtkFileFilter *filter2 = gtk_file_filter_new();
gtk_file_filter_set_name(filter1, "Images Files");
gtk_file_filter_set_name(filter2, "All Files");
gtk_file_filter_add_pattern(filter1, "*.png");
gtk_file_filter_add_pattern(filter1, "*.xpm");
gtk_file_filter_add_pattern(filter1, "*.jpg");
gtk_file_filter_add_pattern(filter2, "*");
```

下面这个程序是个简单的范例，使用 GtkFileChooserDialog 来选择档案并显示名称在 GtkLabel 上：

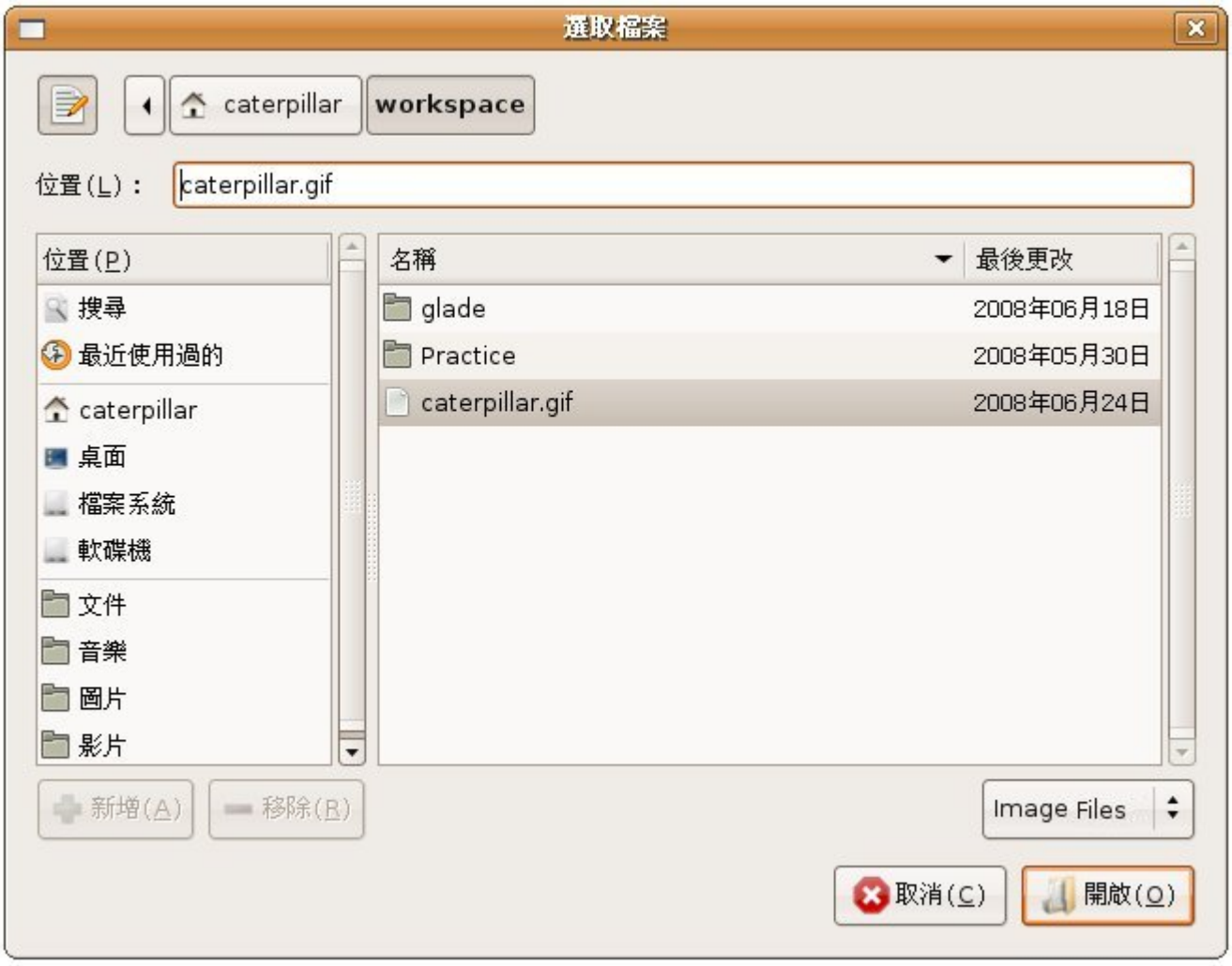
gtk_file_chooser_button_demo.c

```
#include <gtk/gtk.h>

void file_changed(GtkFileChooserButton *button, GtkLabel *label) {
	gchar *file = gtk_file_chooser_get_filename(GTK_FILE_CHOOSER(button));
	gtk_label_set_text(label, file);
}

int main(int argc, char *argv[]) {
	GtkWidget *window;
	GtkWidget *label;
	GtkWidget *button;
	GtkFileFilter *filter1, *filter2;
	GtkWidget *box;
	const gchar *filename = "/home/caterpillar/workspace/caterpillar.gif";

	gtk_init(&argc, &argv);
	window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
	gtk_window_set_title(GTK_WINDOW(window), "GtkFileChooserButton");
```

```
    button = gtk_file_chooser_button_new(
                        "选取档案", GTK_FILE_CHOOSER_ACTION_OPEN);
    gtk_file_chooser_set_filename(GTK_FILE_CHOOSER(button), filename);

    filter1 = gtk_file_filter_new();
    filter2 = gtk_file_filter_new();
    gtk_file_filter_set_name(filter1, "Image Files");
    gtk_file_filter_set_name(filter2, "All Files");
    gtk_file_filter_add_pattern(filter1, "*.png");
    gtk_file_filter_add_pattern(filter1, "*.gif");
    gtk_file_filter_add_pattern(filter1, "*.jpg");
    gtk_file_filter_add_pattern(filter2, "*");
    gtk_file_chooser_add_filter(GTK_FILE_CHOOSER(button), filter1);
    gtk_file_chooser_add_filter(GTK_FILE_CHOOSER(button), filter2);

    label = gtk_label_new(filename);

    box = gtk_vbox_new(TRUE, 5);
    gtk_box_pack_start(GTK_BOX(box), button, TRUE, TRUE, 5);
    gtk_box_pack_start(GTK_BOX(box), label, TRUE, TRUE, 5);

    gtk_container_add(GTK_CONTAINER(window), box);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                        G_CALLBACK(gtk_main_quit), NULL);
    g_signal_connect(GTK_OBJECT(button), "selection_changed",
                        G_CALLBACK(file_changed), label);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

执行时的外观如下所示：

选取档案时的对话框外观如下所示：



按下按钮所出现的对话框实际为 GtkFileChooserDialog， GtkFileChooserDialog 为 GtkFileChooserWidget 与 GtkDialog 的组合， GtkFileChooserWidget 可以嵌入其它组件，无论是 GtkFileChooserButton、 GtkFileChooserDialog、GtkFileChooserWidget 都实作了 GtkFileChooser 接口，都可以使用 GtkFileChooser 中的函式进行设定或相关操作。

以下的程序使用 GtkFileChooserDialog 来实作类似上面的范例：

gtk_file_chooser_dialog_demo.c

```
#include <gtk/gtk.h>

void file_changed(GtkButton *button, GtkWidget **widgets) {
        GtkWidget *dialog = gtk_file_chooser_dialog_new (
                                        "另存新檔 ...", GTK_WINDOW(widgets[0]),
                                        GTK_FILE_CHOOSER_ACTION_SAVE,
                                        GTK_STOCK_CANCEL, GTK_RESPONSE_CANCEL,
                                GTK_STOCK_SAVE, GTK_RESPONSE_ACCEPT,
                                        NULL
                        );
```

```
        GtkFileFilter *filter1 = gtk_file_filter_new();
        GtkFileFilter *filter2 = gtk_file_filter_new();
        gtk_file_filter_set_name(filter1, "Image Files");
        gtk_file_filter_set_name(filter2, "All Files");
        gtk_file_filter_add_pattern(filter1, "*.png");
        gtk_file_filter_add_pattern(filter1, "*.gif");
        gtk_file_filter_add_pattern(filter1, "*.jpg");
        gtk_file_filter_add_pattern(filter2, "*");
        gtk_file_chooser_add_filter(GTK_FILE_CHOOSER(dialog), filter1);
        gtk_file_chooser_add_filter(GTK_FILE_CHOOSER(dialog), filter2);

        if(gtk_dialog_run(GTK_DIALOG(dialog)) == GTK_RESPONSE_ACCEPT) {
                gchar *fileName = gtk_file_chooser_get_filename(
                                                GTK_FILE_CHOOSER (dialog));
                gtk_label_set_text(GTK_LABEL(widgets[1]), fileName);
                g_free(fileName);
        }

        gtk_widget_destroy(dialog);
}

int main(int argc, char *argv[]) {
        GtkWidget *window;
        GtkWidget *label;
        GtkWidget *button;
        GtkWidget *box;
        GtkWidget *widgets[2];
        const gchar *filename = "/home/caterpillar/workspace/caterpillar.gif";

        gtk_init(&argc, &argv);
        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        gtk_window_set_title(GTK_WINDOW(window), "GtkFileChooserDialog");

        button = gtk_button_new_with_label("选择档案");
        label = gtk_label_new(filename);

        box = gtk_vbox_new(TRUE, 5);
        gtk_box_pack_start(GTK_BOX(box), button, TRUE, TRUE, 5);
        gtk_box_pack_start(GTK_BOX(box), label, TRUE, TRUE, 5);

        gtk_container_add(GTK_CONTAINER(window), box);

        widgets[0] = window;
        widgets[1] = label;
```

```
    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);
    g_signal_connect(GTK_OBJECT(button), "clicked",
                     G_CALLBACK(file_changed), widgets);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

2.26 使用 GtkDialog 自订对话框

若想要自订对话框，则可以使用 GtkDialog 来组合组件，GtkDialog 预设是分作两个区域的组件，如下图所示：

基本上，对话框上方是置放各式组件的区域，下方是置放按钮的区域，中间以分隔线作区隔，从类别结构上来看：

```
typedef struct {
  GtkWidget *vbox;
  GtkWidget *action_area;
} GtkDialog;
```

GtkDialog 上方 vbox 为一个 GtkVBox，下方 action_area 为一个 GtkHButtonBox，中间的分隔线则为 GtkHSeparator，分隔线可以使用 gtk_dialog_set_has_separator()函式来设定是否显示。自订对话框，事实上就是类似组合窗口中各个组件及其版面配置。

先前所示范的对话框都是独占（Modal）对话框，也就是若不响应对话框，则无法操作其 parent 窗口，通常对话框都是独占的，因为它们常作为要使用者立即响应以便进行下一步的作用，在也可以简化窗口操作的设计，若是设定为非独占式，则操作设计上会复杂许多，最基本的就是对话框关闭后，何时该销毁对话框的问题，一个方法是可以将对话框的 "response" Signal 连接至 gtk_widget_destroy()函式，当对话框上的有个按钮被按下或窗口被关闭时，销毁对话框。

下面这个范例程序以自订对话框的方式仿真 GtkMessageDialog 中的讯息对话框，并示范非独占式对话框的使用：

gtk_dialog_demo.c

```
#include <gtk/gtk.h>

void clicked_callback(GtkButton *button, GtkWindow *window) {
    GtkWidget *dialog = gtk_dialog_new_with_buttons("GtkDialog", window,
                                    GTK_DIALOG_DESTROY_WITH_PARENT,
                                    GTK_STOCK_OK, GTK_RESPONSE_OK,
                                    NULL);

    GtkWidget *label = gtk_label_new(NULL);
    GtkWidget *image = gtk_image_new_from_stock(GTK_STOCK_DIALOG_INFO,
                                                GTK_ICON_SIZE_DIALOG);
    GtkWidget *hbox = gtk_hbox_new(FALSE, 5);

    gtk_dialog_set_has_separator(GTK_DIALOG(dialog), FALSE);
    gtk_label_set_markup(GTK_LABEL(label),
      "<b>良葛格学习笔记</b>\nhttp://caterpillar.onlyfun.net");

    gtk_container_set_border_width(GTK_CONTAINER (hbox), 10);
    gtk_box_pack_start_defaults(GTK_BOX(hbox), image);
    gtk_box_pack_start_defaults(GTK_BOX(hbox), label);

    gtk_box_pack_start_defaults(GTK_BOX(GTK_DIALOG(dialog)->vbox), hbox);

    g_signal_connect(G_OBJECT(dialog), "response",
                     G_CALLBACK(gtk_widget_destroy), NULL);

    gtk_widget_show_all(dialog);
}

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *button;

    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GtkDialog");

    button = gtk_button_new_with_label("按我");

    gtk_container_add(GTK_CONTAINER(window), button);
```

```
    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);
    g_signal_connect(GTK_OBJECT(button), "clicked",
                     G_CALLBACK(clicked_callback), window);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

一个执行的画面如下所示：

2.3 文字字段

2.31GtkEntry

GtkEntry 提供一个文字输入字段，可以输入文字或是设定为一般显示、密码显示或搭配 GtkEntryCompletion 制作自动完成功能等，自动完成的范例稍微复杂，在这边先不提及，以下的程序是个简单的设定示范，范例中使用了 GtkTable 进行版面配置：

gtk_entry_demo.c

```
#include <gtk/gtk.h>

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *nameEntry, *pwdEntry;
    GtkWidget *table;

    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GtkEntry");

    nameEntry = gtk_entry_new();
    pwdEntry = gtk_entry_new();
    gtk_entry_set_visibility(GTK_ENTRY(pwdEntry), FALSE);
```

```
    table = gtk_table_new(2, 2, TRUE);
    gtk_table_attach_defaults(GTK_TABLE(table),
            gtk_label_new("Name"), 0, 1, 0, 1);
    gtk_table_attach_defaults(GTK_TABLE(table), nameEntry, 1, 2, 0, 1);
    gtk_table_attach_defaults(GTK_TABLE(table),
            gtk_label_new("Password"), 0, 1, 1, 2);
    gtk_table_attach_defaults(GTK_TABLE(table), pwdEntry, 1, 2, 1, 2);

    gtk_container_add(GTK_CONTAINER(window), table);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                        G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

gtk_entry_set_visibility()可以设定输入字符是否可见，设定为 FALSE 的话，密码显示会使用屏蔽字符(像是*)来响应使用者的输入，您可以使用 gtk_entry_set_invisible_char()设定屏蔽字符。

GtkEntry 还可以使用 gtk_entry_set_alignment()设定文字对齐方式，可设定的值为 0 到 1，表示水平方向由左到右的对齐位置，也可以使用 gtk_entry_set_editable()设定 GtkEntry 的字段是否可编辑。

下图为执行时的参考画面：

2.32GtkSpinButton

GtkSpinButton 是个可以让使用者输入数值的组件，具有上下调整的箭头按钮，可以设定可输入数值的上下限、小数字数与按下箭头的递增（减）值等，虽然名为 Button，但 GtkSpinButton 实际是衍生自 GtkEntry：

```
GtkWidget
 +----GtkEntry
       +----GtkSpinButton
```

您可以使用 gtk_spin_button_new()函式来新增一个 GtkSpinButton：

```
GtkWidget* gtk_spin_button_new(
              GtkAdjustment *adjustment, gdouble climb_rate, guint digits);
```

climb_rate 是设定按下按钮时，数值改变的加速度，为一个 0.0 到 1.0 的设定，digits 则是设定显示数值的小数字数，这个函式还需要一个 GtkAdjustment，这个对象用以控制数值的范围、上下限、递增（减）值等：

```
GtkObject* gtk_adjustment_new(gdouble value,          // 初始值
                              gdouble lower,          // 下界值
                              gdouble upper,          // 上界值
                              gdouble step_increment, // 递增（减）值
                              gdouble page_increment, // 对 GtkSpinButton 较无意
义
                              gdouble page_size);     // 对 GtkSpinButton 较无意
义
```

GtkAdjustment 也可以用于其它的 Widget，page_increment 是按下 PageDown、PageUp 时的端增（减）量， page_size 是组件可以显示的大小，但就 GtkSpinButton 而言，仅 step_increment 的设定有意义。

您也可以使用 gtk_spin_button_new_with_range()，直接指定最小值、最大值与递增（减）值即可，这个函式会自动产生 GtkAdjustment：

```
GtkWidget* gtk_spin_button_new_with_range(gdouble min,
                                          gdouble max,
                                          gdouble step);
```

在 GtkHBox 与 GtkVBox 就曾经使用过 GtkSpinButton，当时使用的就是 gtk_spin_button_new_with_range()函式来建立 GtkSpinButton，在这边改写一下那边的范例，改为自行建立 GtkAdjustment 并使用 gtk_spin_button_new()函式来新增一个 GtkSpinButton：

gtk_spin_button_demo.c

```
#include <gtk/gtk.h>

void value_changed_callback(GtkSpinButton *spinButton, gpointer data) {
    gdouble value = gtk_spin_button_get_value(spinButton);
    GString *text = g_string_new("");
```

```
    g_string_sprintf(text, "%.2f", value);
    gtk_label_set_text(GTK_LABEL(data), text->str);
}

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *spinButton;
    GtkObject *adjustment;
    GtkWidget *label;
    GtkWidget *hbox;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GtkSpinButton");
    gtk_window_set_default_size(GTK_WINDOW(window), 250, 50);

    adjustment = gtk_adjustment_new(0.0, 0.0, 100.0, 0.05, 0.0, 0.0);
    spinButton = gtk_spin_button_new(GTK_ADJUSTMENT(adjustment), 0.01, 2);
    label = gtk_label_new("0.00");
    hbox = gtk_hbox_new(TRUE, 5);

    gtk_box_pack_start(GTK_BOX(hbox), spinButton, TRUE, TRUE, 5);
    gtk_box_pack_start(GTK_BOX(hbox), label, TRUE, TRUE, 5);
    gtk_container_add(GTK_CONTAINER(window), hbox);

    g_signal_connect(GTK_OBJECT(spinButton), "value_changed",
                     G_CALLBACK(value_changed_callback), label);
    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

一个执行的结果如下所示：

2.33GtkTextView

GtkTextView 可用于显示文字与编辑文字，其本身并不具备滚动条功能，您可以使用 GtkScrolledWindow 套于其上，为其增加滚动条功能，GtkScrolledWindow 可以简单的使用 gtk_scrolled_window_new()来建立：

```
GtkWidget *scrolledWindow = gtk_scrolled_window_new(NULL, NULL);
```

gtk_scrolled_window_new()的两个参数为 GtkAdjustment，如果您给定为 NULL，则会自动生成预设的两个 GtkAdjustment。要将 GtkTextView 加入 GtkScrolledWindow 可以如下：

```
GtkWidget *textView = gtk_text_view_new();
gtk_container_add(GTK_CONTAINER(scrolledWindow), textView);
```

若要取得或设定 GtkTextView 中的文字，则要取得其内部 GtkTextBuffer 缓冲区对象，一个例子如下：

```
GtkTextBuffer *buffer;
buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(textView));
gtk_text_buffer_set_text(buffer, "Hello! World!", -1);
```

设定文字是使用 gtk_text_buffer_set_text()函式，想取得文字则是使用 gtk_text_buffer_get_text()。

以下这个程序综合了之前所介绍过的几个组件，像是 GtkVBox、GtkFileChooserButton 以及 GtkTextView 组件来进行文本文件的读取与显示，其中关于档案读取的部份，使用了 GLib 的 g_file_get_contents() 等函式，简单的达到开启档案读取的动作：

gtk_text_view_demo.c

```
#include <gtk/gtk.h>

void file_changed(GtkFileChooserButton *button, GtkTextView *textView) {
    gchar *filename;
    gchar *content;
    gsize bytes;
    GError *error = NULL;
    GtkTextBuffer *buffer;

    filename = gtk_file_chooser_get_filename(GTK_FILE_CHOOSER(button));

    g_file_get_contents(filename, &content, &bytes, &error);
```

```
        if (error != NULL) {
                g_printf(error->message);
                g_clear_error(&error);
        }

        buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW (textView));
        gtk_text_buffer_set_text (buffer, content, -1);
}

int main(int argc, char *argv[]) {
        GtkWidget *window;
        GtkWidget *scrolledWindow;
        GtkWidget *textView;
        GtkWidget *button;
        GtkWidget *vbox;
        PangoFontDescription *fontDesc;

        gtk_init(&argc, &argv);

        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        gtk_window_set_title(GTK_WINDOW(window), "GtkTextView");
        gtk_window_set_default_size(GTK_WINDOW(window), 250, 50);

        scrolledWindow = gtk_scrolled_window_new(NULL, NULL);
        textView = gtk_text_view_new();
        fontDesc = pango_font_description_from_string("Courier Pitch Bold 12");
        gtk_widget_modify_font(textView, fontDesc);

        gtk_container_add(GTK_CONTAINER(scrolledWindow), textView);

        button = gtk_file_chooser_button_new(
                        "选取档案", GTK_FILE_CHOOSER_ACTION_OPEN);

        vbox = gtk_vbox_new(FALSE, 5);
        gtk_box_pack_start(GTK_BOX(vbox), scrolledWindow, TRUE, TRUE, 5);
        gtk_box_pack_start(GTK_BOX(vbox), button, FALSE, FALSE, 5);

        gtk_container_add(GTK_CONTAINER(window), vbox);

        g_signal_connect(GTK_OBJECT(button), "selection_changed",
                                G_CALLBACK(file_changed), textView);
        g_signal_connect(GTK_OBJECT(window), "destroy",
                                G_CALLBACK(gtk_main_quit), NULL);
```

```
    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

一个读取档案后的执行画面如下所示：

2.4 选项清单

2.41GtkComboBox

GtkComboBox 可以建立下拉选单，以供使用者选取项目，GtkComboBox 实现了 Model-View 模式，可提供丰富的项目类型与表现方式，但这也提高了程序撰写时的复杂度，为此，GtkComboBox 提供了 gtk_combo_box_new_text()以建立一般常用的文字下拉列表功能之 GtkComboBox，若想要设定下拉列表中的文字项目，则可以使用 gtk_combo_box_append_text()、 gtk_combo_box_insert_text()、gtk_combo_box_prepend_text()、gtk_combo_box_remove_text()、gtk_combo_box_get_active_text()等函式。

下面这个程序是个简单的示范，实作只有文字选项的 GtkComboBox，作为介绍 GtkComboBox 的开始，下拉选定项目后，会在下方的 GtkLabel 显示所选中的文字：

gtk_combo_box_demo.c

```
#include <gtk/gtk.h>

gboolean combo_changed(GtkComboBox *comboBox, GtkLabel *label) {
```

```
    gchar *active = gtk_combo_box_get_active_text(comboBox);
    gtk_label_set_text(label, active);
}

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *comboBox;
    GtkWidget *label;
    GtkWidget *vbox;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GtkComboBox");
    gtk_window_set_default_size(GTK_WINDOW(window), 200, 50);

    comboBox = gtk_combo_box_new_text();
    gtk_combo_box_append_text(GTK_COMBO_BOX(comboBox), "caterpillar");
    gtk_combo_box_append_text(GTK_COMBO_BOX(comboBox), "momor");
    gtk_combo_box_append_text(GTK_COMBO_BOX(comboBox), "hamimi");
    gtk_combo_box_append_text(GTK_COMBO_BOX(comboBox), "bush");
    gtk_combo_box_set_active(GTK_COMBO_BOX(comboBox), 0);

    label = gtk_label_new("caterpillar");
    vbox = gtk_vbox_new(TRUE, 5);

    gtk_box_pack_start(GTK_BOX(vbox), comboBox, TRUE, TRUE, 5);
    gtk_box_pack_start(GTK_BOX(vbox), label, TRUE, TRUE, 5);
    gtk_container_add(GTK_CONTAINER(window), vbox);

    g_signal_connect(GTK_OBJECT(comboBox), "changed",
                     G_CALLBACK(combo_changed), label);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

一个执行时的画面如下所示：

2.42GtkComboBox 与 GtkListStore

在 GtkComboBox 中的范例，是 GtkComboBox 最简单的用法，选项只有纯文字，若想要进一步让 GtkComboBox 呈现更多的功能与样貌，则必须了解 GtkComboBox 的 Model-View 设计。

GtkComboBox 的外观是 View 对象的部份，选项数据的部份则是 Model 对象的部份，GtkComboBox 使用的 Model 对象为实作 GtkTreeModel 接口的对象，例如 GtkListStore 或 GtkTreeStore，其实作了 Model 对象的数据储存与存取方式等，在这边先介绍 GtkComboBox 与 GtkListStore 的使用。

GtkListStore 是没有阶层平坦式的数据，当下拉选单需要的是直接列示选项就可以使用，GtkListStore 中可以设定文字、图片、组件等，要建立 GtkListStore，必须指明要建立几个字段以及字段中的型态，例如：

```
GtkListStore *store = gtk_list_store_new(2, GDK_TYPE_PIXBUF, G_TYPE_STRING);
```

这个程序片段将建立一个具有两个字段的 GtkListStore，一个字段储存图片，使用 GDK_TYPE_PIXBUF 来指定,字段将储存的是 GdkPixbuf,另一个储存文字,使用 G_TYPE_STRING 来指定。

您可以使用 gdk_pixbuf_new_from_file()读取图档并传回其 GdkPixbuf，第二个参数是 GError，若不需要可以设定为 NULL：

```
GdkPixbuf *pixbuf = gdk_pixbuf_new_from_file(files[i], NULL);
```

GtkListStore 使用 GtkTreeIter 作为内部的数据位置指针，当您使用 gtk_list_store_append()时，会将 GtkTreeIter 指向 GtkListStore 下一列的位置，接着您再搭配 gtk_list_store_set()设定 GtkListStore 该位置的字段数据，例如：

```
GtkTreeIter iter;
gtk_list_store_append(store, &iter);
gtk_list_store_set(store, &iter,
                   0, pixbuf,
                   1, filename,
                   -1);
```

gtk_list_store_set()的前两个参数分别为GtkListStore与GtkTreeIter，之后则两两成对指定字段索引与数据，最后以-1作为结束。

有了GtlListStore这个Model对象之后，接着可以用以建立View，也就是GtkComboBox，您可以使用gtk_combo_box_new_with_model()来建立：

```
GtkWidget *comboBox = gtk_combo_box_new_with_model(GTK_TREE_MODEL(store));
```

您的数据要如何显示，需要对应的GtkCellRenderer来进行绘制，哪个字段要使用哪个GtkCellRenderer以及该字段的一些相关属性，则要告知GtkCellLayout，GtkComboBox有实作 GtkCellLayout接口，因此，您可以如下使用gtk_cell_layout_pack_start()设定GtkCellRenderer绘制哪个字段,以及使用gtk_cell_layout_set_attributes()设定相关属性：

```
GtkCellRender *render;
renderer = gtk_cell_renderer_pixbuf_new();
gtk_cell_layout_pack_start(GTK_CELL_LAYOUT(comboBox), renderer, FALSE);
gtk_cell_layout_set_attributes(GTK_CELL_LAYOUT(comboBox), renderer,
                               "pixbuf", 0, // "pixbuf" 设定图像
                               NULL);  // 最后以 NULL 结尾
renderer = gtk_cell_renderer_text_new();
gtk_cell_layout_pack_start(GTK_CELL_LAYOUT(comboBox), renderer, FALSE);
gtk_cell_layout_set_attributes(GTK_CELL_LAYOUT(comboBox), renderer,
                               "text", 1, // "text" 设定文字
                               NULL);
```

以上这些大致上是设定GtkComboBox的Model与View的基本流程。假设现在使用者选择下拉列表中的选项，则要取得选项数据，则需先取得 Model，也就是CtkComboBox中的GtkListStore,并取得选中的选项之GtkTreeIter(记得吗？GtkTreeIter指向GtkListStore中的某列数据)，然后再使用gtk_tree_model_get()取得想要的字段值，例如：

```
gboolean combo_changed(GtkComboBox *comboBox, GtkLabel *label) {
    GtkTreeModel *model = gtk_combo_box_get_model(comboBox);
    GtkTreeIter iter;
    gchar *active;
    gtk_combo_box_get_active_iter(comboBox, &iter);
    gtk_tree_model_get(model, &iter,
                       1, &active,
                       -1);
```

```
    gtk_label_set_text(label, active);
}
```

综合以上说明，来改写一下 GtkComboBox 中的范例，让下拉选项可以拥有一个小图示：

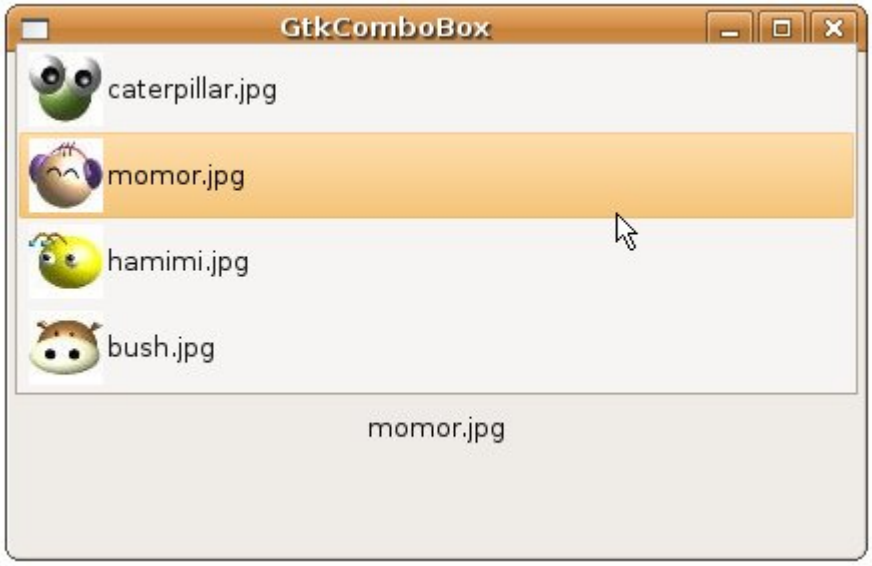
gtk_combo_box_with_icon_demo.c

```
#include <gtk/gtk.h>

enum {
    PIXBUF_COL,
    TEXT_COL
};

GtkTreeModel* createModel() {
    const gchar *files[] = {"caterpillar.jpg", "momor.jpg",
                            "hamimi.jpg", "bush.jpg"};
    GdkPixbuf *pixbuf;
    GtkTreeIter iter;
    GtkListStore *store;
    gint i;

    store = gtk_list_store_new(2, GDK_TYPE_PIXBUF, G_TYPE_STRING);

    for(i = 0; i < 4; i++) {
        pixbuf = gdk_pixbuf_new_from_file(files[i], NULL);
        gtk_list_store_append(store, &iter);
        gtk_list_store_set(store, &iter,
                           PIXBUF_COL, pixbuf,
                           TEXT_COL, files[i],
                   -1);
        gdk_pixbuf_unref(pixbuf);
    }

    return GTK_TREE_MODEL(store);
}

gboolean combo_changed(GtkComboBox *comboBox, GtkLabel *label) {
    GtkTreeModel *model = gtk_combo_box_get_model(comboBox);
    GtkTreeIter iter;
    gchar *active;
    gtk_combo_box_get_active_iter(comboBox, &iter);
    gtk_tree_model_get(model, &iter,
                       1, &active,
```

```
                                -1);

        gtk_label_set_text(label, active);
}

int main(int argc, char *argv[]) {
        GtkWidget *window;
        GtkWidget *comboBox;
        GtkCellRenderer *renderer;
        GtkWidget *label;
        GtkWidget *vbox;

        gtk_init(&argc, &argv);

        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        gtk_window_set_title(GTK_WINDOW(window), "GtkComboBox");
        gtk_window_set_default_size(GTK_WINDOW(window), 200, 50);

        comboBox = gtk_combo_box_new_with_model(createModel());
        gtk_combo_box_set_active(GTK_COMBO_BOX(comboBox), 0);
        renderer = gtk_cell_renderer_pixbuf_new();
        gtk_cell_layout_pack_start(GTK_CELL_LAYOUT(comboBox), renderer, FALSE);
        gtk_cell_layout_set_attributes(GTK_CELL_LAYOUT(comboBox), renderer,
                                "pixbuf", PIXBUF_COL,
                                NULL);
        renderer = gtk_cell_renderer_text_new();
        gtk_cell_layout_pack_start(GTK_CELL_LAYOUT(comboBox), renderer, FALSE);
        gtk_cell_layout_set_attributes(GTK_CELL_LAYOUT(comboBox), renderer,
                                "text", TEXT_COL,
                                NULL);

        label = gtk_label_new("caterpillar.jpg");
        vbox = gtk_vbox_new(TRUE, 5);

        gtk_box_pack_start(GTK_BOX(vbox), comboBox, TRUE, TRUE, 5);
        gtk_box_pack_start(GTK_BOX(vbox), label, TRUE, TRUE, 5);
        gtk_container_add(GTK_CONTAINER(window), vbox);

        g_signal_connect(GTK_OBJECT(comboBox), "changed",
                                G_CALLBACK(combo_changed), label);

        g_signal_connect(GTK_OBJECT(window), "destroy",
                                G_CALLBACK(gtk_main_quit), NULL);
```

```
    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

一个执行的结果如下所示：



2.43GtkComboBox 与 GtkTreeStore

GtkComboBox 与 GtkListStore 中介绍了平坦无阶层的选项如何制作，若想要制作有阶层的树状结构，则要搭配 GtkTreeStore 来使用，主要的差别在于，GtkTreeStroe 具有父子节点关系，也因此在加入子节点时，必须指明父节点为谁，加入节点可以使用 gtk_tree_store_append()函式，设定节点资料可以使用 gtk_tree_store_set()函式：

```
void gtk_tree_store_append(GtkTreeStore *tree_store,
                           GtkTreeIter *iter,
                           GtkTreeIter *parent);
void gtk_tree_store_set(GtkTreeStore *tree_store,
                        GtkTreeIter *iter,
                        ...);
```

使用 gtk_tree_store_append()时若无父节点，则第三个参数设定为 NULL，表示这是最上层节点，也因此，您必须有两个 GtkTreeIter，一个指向目前 GtkTreeStore 中的父节点位置，一个用以指向子节点位置。

下面这个程序改写 GtkComboBox 与 GtkListStore，使其具有子阶层，在 GtkComboBox 中会以子选单方式呈现，子选单的内容是随机数选取决定的，程序的改写主要都是在 Model 的建立部份：

gtk_combo_box_with_tree_demo.c

```
#include <gtk/gtk.h>

enum {
	PIXBUF_COL,
	TEXT_COL
};

GtkTreeModel* createModel() {
	const gchar *files[] = {"caterpillar.jpg", "momor.jpg",
					"hamimi.jpg", "bush.jpg"};
	gchar *stocks[] = {
		GTK_STOCK_DIALOG_WARNING,
		GTK_STOCK_STOP,
		GTK_STOCK_NEW,
		GTK_STOCK_CLEAR,
		GTK_STOCK_OPEN
	};

	gchar *stockNames[] = {
		"WARNING",
		"STOP",
		"NEW",
		"GTK_STOCK_CLEAR",
		"GTK_STOCK_OPEN"
	};

	GtkWidget *cellView;
	GdkPixbuf *pixbuf;
	GtkTreeIter iter1, iter2;
	GtkTreeStore *store;
	gint i, j, s;

	store = gtk_tree_store_new(2, GDK_TYPE_PIXBUF, G_TYPE_STRING);
	cellView = gtk_cell_view_new();
	for(i = 0; i < 4; i++) {
		pixbuf = gdk_pixbuf_new_from_file(files[i], NULL);
		gtk_tree_store_append(store, &iter1, NULL);
		gtk_tree_store_set(store, &iter1,
					PIXBUF_COL, pixbuf,
					TEXT_COL, files[i],
				-1);
		gdk_pixbuf_unref(pixbuf);
```

```
            for(j = 0; j < 3; j++) {
                s = rand() % 5;
                pixbuf = gtk_widget_render_icon(cellView, stocks[s],
                                GTK_ICON_SIZE_BUTTON, NULL);
                gtk_tree_store_append(store, &iter2, &iter1);
                gtk_tree_store_set(store, &iter2,
                                        PIXBUF_COL, pixbuf,
                                        TEXT_COL, stockNames[s],
                                       -1);
                 gdk_pixbuf_unref(pixbuf);
            }
        }

        return GTK_TREE_MODEL(store);
}

gboolean combo_changed(GtkComboBox *comboBox, GtkLabel *label) {
        GtkTreeModel *model = gtk_combo_box_get_model(comboBox);
        GtkTreeIter iter;
        gchar *active;
        gtk_combo_box_get_active_iter(comboBox, &iter);
        gtk_tree_model_get(model, &iter,
                                1, &active,
                                -1);

        gtk_label_set_text(label, active);
}

int main(int argc, char *argv[]) {
        GtkWidget *window;
        GtkWidget *comboBox;
        GtkCellRenderer *renderer;
        GtkWidget *label;
        GtkWidget *vbox;

        gtk_init(&argc, &argv);

        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        gtk_window_set_title(GTK_WINDOW(window), "GtkComboBox");
        gtk_window_set_default_size(GTK_WINDOW(window), 200, 50);

        comboBox = gtk_combo_box_new_with_model(createModel());
        gtk_combo_box_set_active(GTK_COMBO_BOX(comboBox), 0);
```

```
    renderer = gtk_cell_renderer_pixbuf_new();
    gtk_cell_layout_pack_start(GTK_CELL_LAYOUT(comboBox), renderer, FALSE);
    gtk_cell_layout_set_attributes(GTK_CELL_LAYOUT(comboBox), renderer,
                    "pixbuf", PIXBUF_COL,
                    NULL);
    renderer = gtk_cell_renderer_text_new();
    gtk_cell_layout_pack_start(GTK_CELL_LAYOUT(comboBox), renderer, FALSE);
    gtk_cell_layout_set_attributes(GTK_CELL_LAYOUT(comboBox), renderer,
                    "text", TEXT_COL,
                    NULL);

    label = gtk_label_new("caterpillar.jpg");
    vbox = gtk_vbox_new(TRUE, 5);

    gtk_box_pack_start(GTK_BOX(vbox), comboBox, TRUE, TRUE, 5);
    gtk_box_pack_start(GTK_BOX(vbox), label, TRUE, TRUE, 5);
    gtk_container_add(GTK_CONTAINER(window), vbox);

    g_signal_connect(GTK_OBJECT(comboBox), "changed",
                        G_CALLBACK(combo_changed), label);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                        G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

一个执行的结果如下所示：

2.44GtkTreeView 与 GtkListStore

在看这篇之前，请先看过 GtkComboBox 以及 GtkComboBox 与 GtkListStore，由于以 Model-View 的方式设计，您可以在不修改 Model 的情况下，为 Model 换上另一个显示的外观（View），例如为 GtkComboBox 与 GtkListStore 范例中建立的 GtkListStore 换上 GtkTreeView 的外观。

具体来说，也就是该范例中的 createModel()内容无需改变，将 GtkComboBox 的相对应程序代码，换成建立 GtkTreeView 的程序代码，例如：

```
GtkWidget *treeView = gtk_tree_view_new_with_model(createModel());
GtkCellRenderer *renderer = gtk_cell_renderer_pixbuf_new();
GtkTreeViewColumn *column = gtk_tree_view_column_new_with_attributes(
                                "Icon", renderer,
                                "pixbuf", PIXBUF_COL,
                                 NULL);
gtk_tree_view_append_column(GTK_TREE_VIEW (treeView), column);

renderer = gtk_cell_renderer_text_new();
column = gtk_tree_view_column_new_with_attributes(
                "Filename", renderer,
                "text", TEXT_COL,
                 NULL);
gtk_tree_view_append_column(GTK_TREE_VIEW (treeView), column);
```

GtkTreeViewColumn 是 GtkTreeView 中的列代表，可用以设定该列的内容如何绘制以及一些相关属性，基本上以上程序片段中，gtk_tree_view_column_new_with_attributes()与

gtk_tree_view_append_column()函式可以用
gtk_tree_view_insert_column_with_attributes()函式来简化，也就是可以简化为：

```
GtkWidget *treeView = gtk_tree_view_new_with_model(createModel());
GtkCellRenderer *renderer = gtk_cell_renderer_pixbuf_new();
gtk_tree_view_insert_column_with_attributes(
        GTK_TREE_VIEW(treeView), -1, "Icon", renderer,
        "pixbuf", PIXBUF_COL,
        NULL);

renderer = gtk_cell_renderer_text_new();
gtk_tree_view_insert_column_with_attributes(
        GTK_TREE_VIEW(treeView), -1, "Filename", renderer,
        "text", TEXT_COL,
        NULL);
```

在 GtkTreeView 当中的选择，是以为个 GtkTreeSelection 对象作代表，当选择改变时，会发出"changed"的 Signal，所以要连结 Signal 与 Callback 函式，可以如下进行：

```
GtkTreeSelection *selection =
gtk_tree_view_get_selection(GTK_TREE_VIEW(treeView));
g_signal_connect(G_OBJECT(selection), "changed",
                 G_CALLBACK(selection_changed), label);
```

至于 Callback 函式的部份大同小异,您要从传递的 GtkTreeSelection 中取得 GtkTreeView，从 GtkTreeView 中取得 GtkTreeModel，再使用 gtk_tree_selection_get_selected()将 GtkTreeIter 指向选中的列，以取得您想取得的字段数据：

```
gboolean selection_changed(GtkTreeSelection *selection, GtkLabel *label) {
    GtkTreeView *treeView;
    GtkTreeModel *model;
    GtkTreeIter iter;
    gchar *active;

    treeView = gtk_tree_selection_get_tree_view(selection);
    model = gtk_tree_view_get_model(treeView);
    gtk_tree_selection_get_selected(selection, &model, &iter);
    gtk_tree_model_get(model, &iter,
                       1, &active,
                       -1);

    gtk_label_set_text(label, active);
```

}

下面的范例是以上说明的综合示范：

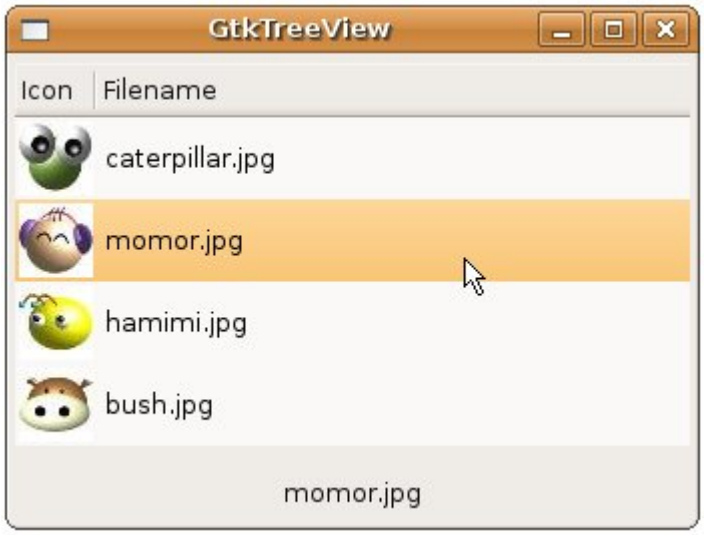
gtk_tree_view_demo.c

```
#include <gtk/gtk.h>

enum {
	PIXBUF_COL,
	TEXT_COL
};

GtkTreeModel* createModel() {
	const gchar *files[] = {"caterpillar.jpg", "momor.jpg",
	                        "hamimi.jpg", "bush.jpg"};
	GdkPixbuf *pixbuf;
	GtkTreeIter iter;
	GtkListStore *store;
	gint i;

	store = gtk_list_store_new(2, GDK_TYPE_PIXBUF, G_TYPE_STRING);

	for(i = 0; i < 4; i++) {
		pixbuf = gdk_pixbuf_new_from_file(files[i], NULL);
		gtk_list_store_append(store, &iter);
		gtk_list_store_set(store, &iter,
		                   PIXBUF_COL, pixbuf,
		                   TEXT_COL, files[i],
		                   -1);
		gdk_pixbuf_unref(pixbuf);
	}

	return GTK_TREE_MODEL(store);
}

gboolean selection_changed(GtkTreeSelection *selection, GtkLabel *label) {
	GtkTreeView *treeView;
	GtkTreeModel *model;
	GtkTreeIter iter;
	gchar *active;

	treeView = gtk_tree_selection_get_tree_view(selection);
	model = gtk_tree_view_get_model(treeView);
```

```
        gtk_tree_selection_get_selected(selection, &model, &iter);
        gtk_tree_model_get(model, &iter,
                                    1, &active,
                                    -1);

        gtk_label_set_text(label, active);
}

int main(int argc, char *argv[]) {
        GtkWidget *window;
        GtkWidget *treeView;
        GtkCellRenderer *renderer;
        GtkTreeViewColumn *column;
        GtkWidget *label;
        GtkWidget *vbox;
        GtkTreeSelection *selection;

        gtk_init(&argc, &argv);

        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        gtk_window_set_title(GTK_WINDOW(window), "GtkTreeView");
        gtk_window_set_default_size(GTK_WINDOW(window), 200, 50);

        treeView = gtk_tree_view_new_with_model(createModel());

        renderer = gtk_cell_renderer_pixbuf_new();
        column = gtk_tree_view_column_new_with_attributes(
                                    "Icon", renderer,
                                    "pixbuf", PIXBUF_COL,
                                     NULL);
        gtk_tree_view_append_column(GTK_TREE_VIEW (treeView), column);

        renderer = gtk_cell_renderer_text_new();
        column = gtk_tree_view_column_new_with_attributes(
                                    "Filename", renderer,
                                    "text", TEXT_COL,
                                     NULL);
        gtk_tree_view_append_column(GTK_TREE_VIEW (treeView), column);

        label = gtk_label_new("caterpillar.jpg");
        vbox = gtk_vbox_new(FALSE, 5);

        gtk_box_pack_start(GTK_BOX(vbox), treeView, TRUE, TRUE, 5);
        gtk_box_pack_start(GTK_BOX(vbox), label, TRUE, TRUE, 5);
```

```
    gtk_container_add(GTK_CONTAINER(window), vbox);

    selection = gtk_tree_view_get_selection(GTK_TREE_VIEW(treeView));
    g_signal_connect(G_OBJECT(selection), "changed",
                     G_CALLBACK(selection_changed), label);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

一个执行的结果画面如下所示：



2.45GtkTreeView 与 GtkTreeStrore

有了 GtkTreeView 与 GtkListStore 的基础，要在 GtkTreeView 搭配 GtkTreeStore 就没什么好解释的了，下面这个范例，只是将 GtkTreeView 与 GtkListStore 范例中的 createModel()函式，换成 GtkComboBox 与 GtkTreeStore 中的 createModel()函式，剩下的都没有改变，也就是为 GtkComboBox 与 GtkTreeStore 中的 Model 换上 GtkTreeView 的外观显示：

gtk_tree_view_with_tree_store.c

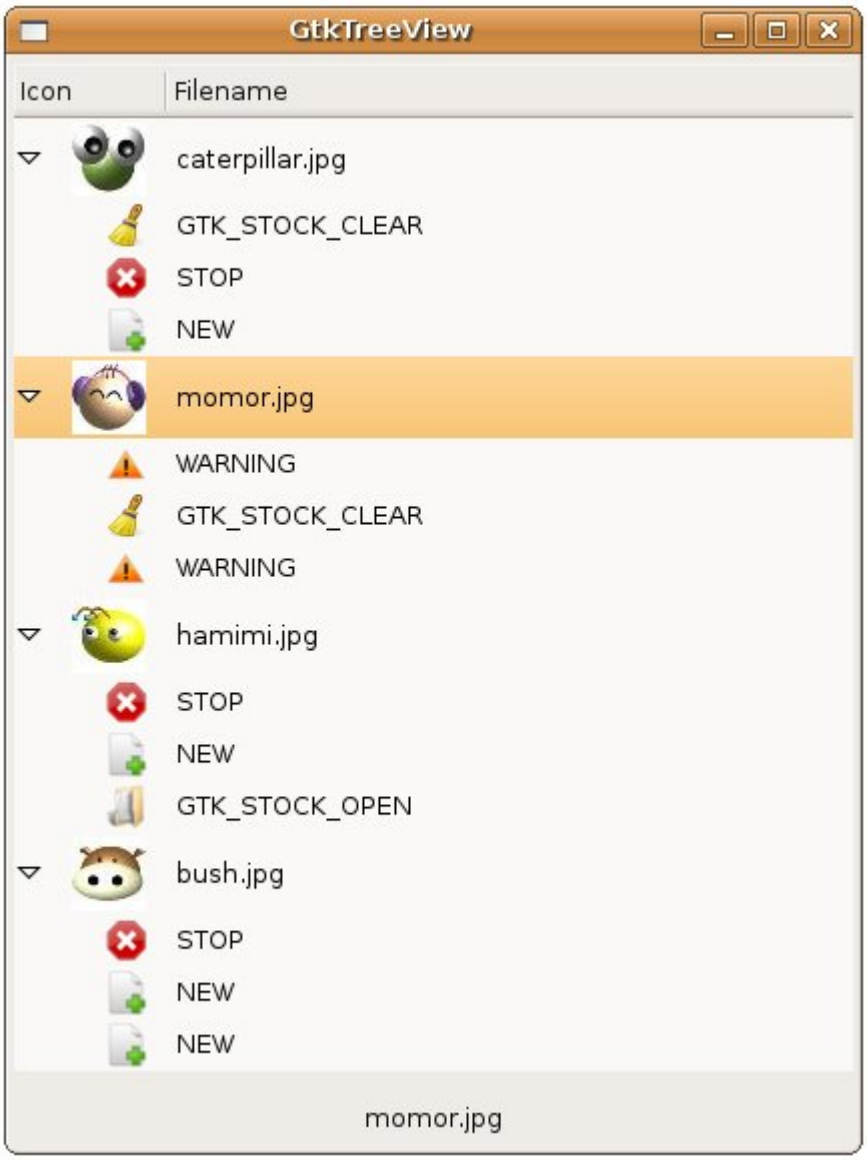
```
#include <gtk/gtk.h>

enum {
    PIXBUF_COL,
    TEXT_COL
```

```
};

GtkTreeModel* createModel() {
	const gchar *files[] = {"caterpillar.jpg", "momor.jpg",
					"hamimi.jpg", "bush.jpg"};
	gchar *stocks[] = {
		GTK_STOCK_DIALOG_WARNING,
		GTK_STOCK_STOP,
		GTK_STOCK_NEW,
		GTK_STOCK_CLEAR,
		GTK_STOCK_OPEN
	};

	gchar *stockNames[] = {
		"WARNING",
		"STOP",
		"NEW",
		"GTK_STOCK_CLEAR",
		"GTK_STOCK_OPEN"
	};

	GtkWidget *cellView;
	GdkPixbuf *pixbuf;
	GtkTreeIter iter1, iter2;
	GtkTreeStore *store;
	gint i, j, s;

	store = gtk_tree_store_new(2, GDK_TYPE_PIXBUF, G_TYPE_STRING);
	cellView = gtk_cell_view_new();
	for(i = 0; i < 4; i++) {
		pixbuf = gdk_pixbuf_new_from_file(files[i], NULL);
		gtk_tree_store_append(store, &iter1, NULL);
		gtk_tree_store_set(store, &iter1,
					PIXBUF_COL, pixbuf,
					TEXT_COL, files[i],
				-1);
		gdk_pixbuf_unref(pixbuf);

		for(j = 0; j < 3; j++) {
			s = rand() % 5;
			pixbuf = gtk_widget_render_icon(cellView, stocks[s],
					GTK_ICON_SIZE_BUTTON, NULL);
			gtk_tree_store_append(store, &iter2, &iter1);
			gtk_tree_store_set(store, &iter2,
```

```
                                        PIXBUF_COL, pixbuf,
                                        TEXT_COL, stockNames[s],
                                       -1);
                gdk_pixbuf_unref(pixbuf);
            }
        }

        return GTK_TREE_MODEL(store);
}

gboolean selection_changed(GtkTreeSelection *selection, GtkLabel *label) {
        GtkTreeView *treeView;
        GtkTreeModel *model;
        GtkTreeIter iter;
        gchar *active;

        treeView = gtk_tree_selection_get_tree_view(selection);
        model = gtk_tree_view_get_model(treeView);
        gtk_tree_selection_get_selected(selection, &model, &iter);
        gtk_tree_model_get(model, &iter,
                                1, &active,
                                -1);

        gtk_label_set_text(label, active);
}

int main(int argc, char *argv[]) {
        GtkWidget *window;
        GtkWidget *treeView;
        GtkCellRenderer *renderer;
        GtkTreeViewColumn *column;
        GtkWidget *label;
        GtkWidget *vbox;
        GtkTreeSelection *selection;

        gtk_init(&argc, &argv);

        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        gtk_window_set_title(GTK_WINDOW(window), "GtkTreeView");
        gtk_window_set_default_size(GTK_WINDOW(window), 200, 50);

        treeView = gtk_tree_view_new_with_model(createModel());

        renderer = gtk_cell_renderer_pixbuf_new();
```

```
    column = gtk_tree_view_column_new_with_attributes(
                                "Icon", renderer,
                                "pixbuf", PIXBUF_COL,
                                 NULL);
    gtk_tree_view_append_column(GTK_TREE_VIEW (treeView), column);

    renderer = gtk_cell_renderer_text_new();
    column = gtk_tree_view_column_new_with_attributes(
                                "Filename", renderer,
                                "text", TEXT_COL,
                                 NULL);
    gtk_tree_view_append_column(GTK_TREE_VIEW (treeView), column);

    label = gtk_label_new("caterpillar.jpg");
    vbox = gtk_vbox_new(FALSE, 5);

    gtk_box_pack_start(GTK_BOX(vbox), treeView, TRUE, TRUE, 5);
    gtk_box_pack_start(GTK_BOX(vbox), label, TRUE, TRUE, 5);
    gtk_container_add(GTK_CONTAINER(window), vbox);

    selection = gtk_tree_view_get_selection(GTK_TREE_VIEW(treeView));
    g_signal_connect(G_OBJECT(selection), "changed",
                                G_CALLBACK(selection_changed), label);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                                G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

一个执行的结果如下所示：



2.5 版面组件

2.51GtkNotebook

在版面配置上，可以会使用 GtkNootbook 来作功能页的分类，它提供多个显示页，可以藉由上方的标签来选择所要的功能页面，下面的程序简单的示范如何将组件加入 GtkNotebook 成为一个标签页，其中 createTab()函式用以建立一个 GtkHBox，内含图片与文字，用以作为功能页的卷标显示，要加入一个功能页，可以使用 gtk_notebook_append_page()函式，指定功能页内容及卷标。

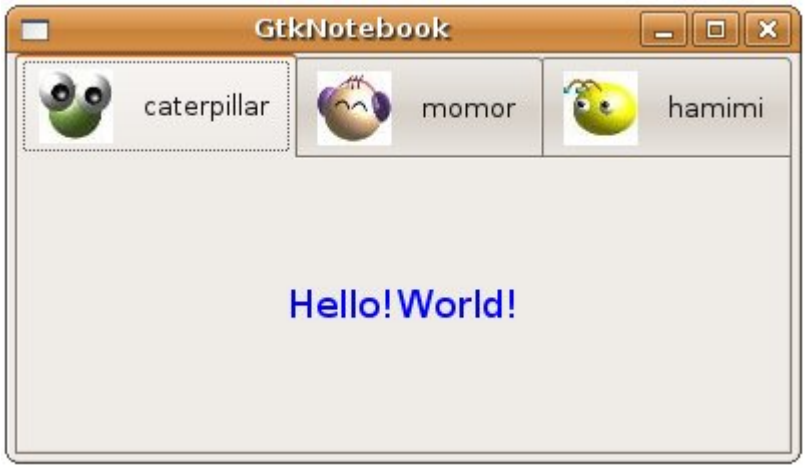
gtk_note_book_demo.c

```
#include <gtk/gtk.h>

GtkWidget* createTab(gchar *filename, gchar *text) {
    GtkWidget *box;
```

```
    GtkWidget *label;
    GtkWidget *image;

    box = gtk_hbox_new(FALSE, 5);
    gtk_container_set_border_width(GTK_CONTAINER(box), 2);

    image = gtk_image_new_from_file(filename);
    label = gtk_label_new(text);

    gtk_box_pack_start(GTK_BOX(box), image, FALSE, FALSE, 5);
    gtk_box_pack_start(GTK_BOX(box), label, FALSE, FALSE, 5);

    gtk_widget_show (image);
    gtk_widget_show (label);

    return box;
}

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *notebook;
    GtkWidget *tab;
    GtkWidget *label;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GtkNotebook");
    gtk_window_set_default_size(GTK_WINDOW(window), 300, 200);

    notebook = gtk_notebook_new();

    tab = createTab("caterpillar.jpg", "caterpillar");
    label = gtk_label_new(NULL);
    gtk_label_set_markup(GTK_LABEL(label),
        "<span foreground='blue' size='x-large'>Hello!World!</span>");
    gtk_notebook_append_page(GTK_NOTEBOOK(notebook), label, tab);

    tab = createTab("momor.jpg", "momor");
    label = gtk_label_new(NULL);
    gtk_label_set_markup(GTK_LABEL(label), "<big>Big text 2</big>");
    gtk_notebook_append_page(GTK_NOTEBOOK(notebook), label, tab);

    tab = createTab("hamimi.jpg", "hamimi");
```

```
    label = gtk_label_new(NULL);
    gtk_label_set_markup(GTK_LABEL(label), "<big>Big text 3</big>");
    gtk_notebook_append_page(GTK_NOTEBOOK(notebook), label, tab);

    gtk_container_add(GTK_CONTAINER(window), notebook);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

一个执行的结果如下所示：



2.52GtkPaned

GtkPaned 是个版面分割组件，可以将窗口版面进行水平切割或垂直切割，您可以使用以下的函式来建立水平或垂直切割的 GtkPaned：

```
GtkWidget *gtk_hpaned_new(void);
GtkWidget *gtk_vpaned_new(void);
```

一个 GtkPaned 有左右或上下两个区域可以加入子组件，您可以使用 gtk_paned_pack1()将组件加入左边或上面，使用 gtk_paned_pack2()将组件加入右边或下面：

```
void gtk_paned_pack1(GtkPaned *paned,
                     GtkWidget *child,
                     gboolean resize,
```

```
                        gboolean shrink);
void gtk_paned_pack2(GtkPaned *paned,
                        GtkWidget *child,
                        gboolean resize,
                        gboolean shrink);
```

一个最简单的范例如下所示：

gtk_paned_demo.c

```
#include <gtk/gtk.h>

int main(int argc, char *argv[]) {
	GtkWidget *window;
	GtkWidget *hPaned;

	gtk_init(&argc, &argv);

	window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
	gtk_window_set_title(GTK_WINDOW(window), "GtkPaned");
	gtk_window_set_default_size(GTK_WINDOW(window), 300, 200);

	hPaned = gtk_hpaned_new();

	gtk_paned_pack1(GTK_PANED(hPaned),
				gtk_frame_new("GtkPaned Add1 Here"), FALSE, FALSE);
	gtk_paned_pack2(GTK_PANED(hPaned),
				gtk_text_view_new(), TRUE, FALSE);

	gtk_container_add(GTK_CONTAINER(window), hPaned);

	g_signal_connect(GTK_OBJECT(window), "destroy",
				G_CALLBACK(gtk_main_quit), NULL);

	gtk_widget_show_all(window);

	gtk_main();

	return 0;
}
```

直接来看执行画面：

gtk_paned_pack1()与 gtk_paned_pack2()可以使用有预设参数的 gtk_paned_add1()及 gtk_paned_add2()来简化。若要作较复杂的版面切割，则可以搭配垂直与水平的切割组合，例如：

gtk_hv_paned_demo.c

```
#include <gtk/gtk.h>

enum {
    PIXBUF_COL,
    TEXT_COL
};

GtkTreeModel* createModel() {
    const gchar *files[] = {"caterpillar.jpg", "momor.jpg",
                            "hamimi.jpg", "bush.jpg"};
    GdkPixbuf *pixbuf;
    GtkTreeIter iter;
    GtkListStore *store;
    gint i;

    store = gtk_list_store_new(2, GDK_TYPE_PIXBUF, G_TYPE_STRING);

    for(i = 0; i < 4; i++) {
        pixbuf = gdk_pixbuf_new_from_file(files[i], NULL);
        gtk_list_store_append(store, &iter);
        gtk_list_store_set(store, &iter,
                           PIXBUF_COL, pixbuf,
                           TEXT_COL, files[i],
                   -1);
        gdk_pixbuf_unref(pixbuf);
    }
```

```
    return GTK_TREE_MODEL(store);
}

GtkWidget* createTreeView() {
    GtkWidget *treeView;
    GtkTreeViewColumn *column;
    GtkCellRenderer *renderer;

    treeView = gtk_tree_view_new_with_model(createModel());

    renderer = gtk_cell_renderer_pixbuf_new();
    column = gtk_tree_view_column_new_with_attributes(
                        "Icon", renderer,
                        "pixbuf", PIXBUF_COL,
                         NULL);
    gtk_tree_view_append_column(GTK_TREE_VIEW (treeView), column);

    renderer = gtk_cell_renderer_text_new();
    column = gtk_tree_view_column_new_with_attributes(
                        "Filename", renderer,
                        "text", TEXT_COL,
                         NULL);
    gtk_tree_view_append_column(GTK_TREE_VIEW (treeView), column);

    return treeView;
}

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *hPaned, *vPaned;
    GtkWidget *treeView;
    GtkWidget *label;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GtkPaned");
    gtk_window_set_default_size(GTK_WINDOW(window), 320, 200);

    hPaned = gtk_hpaned_new();
    vPaned = gtk_vpaned_new();

    treeView = createTreeView();
```

```
    gtk_paned_add1(GTK_PANED(hPaned), treeView);
    gtk_paned_add2(GTK_PANED(hPaned), vPaned);

    label = gtk_label_new(NULL);
    gtk_label_set_markup(GTK_LABEL(label),
        "<span foreground='blue' size='x-large'>Hello!World!</span>");

    gtk_paned_add1(GTK_PANED(vPaned), label);
    gtk_paned_add2(GTK_PANED(vPaned), gtk_text_view_new());

    gtk_container_add(GTK_CONTAINER(window), hPaned);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                        G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

程序主要专注在组件的加入位置即可，所完成的画面如下所示：

2.53GtkScrolledWindow

有些组件预设并没有滚动条，当窗口或父组件无法显示其大小时，只会显示部份区域，但不会出现滚动条，如果您希望这类组件可以出现滚动条，则可以使用 GtkScrolledWindow，在 GtkTextView 中曾经使用过 GtkScrolledWindow，在使用 gtk_scrolled_window_new()建立

GtkScrolledWindow 时可以给定 GtkAdjustment，在 GtkSpinButton 中介绍过它的使用，如果您给定为 NULL，则会自动生成预设的两个 GtkAdjustment：

```
GtkWidget *scrolledWindow = gtk_scrolled_window_new(NULL, NULL);
```

GtkScrolledWindow 包括水平滚动条、垂直滚动条，可使用 gtk_scrolled_window_set_policy()滚动条的出现策略：

```
void gtk_scrolled_window_set_policy( GtkScrolledWindow *scrolled_window,
                                     GtkPolicyType      hscrollbar_policy,
                                     GtkPolicyType      vscrollbar_policy );
```

可设定的策略有 GTK_POLICY_AUTOMATIC 或 GTK_POLICY_ALWAYS。若要将组件加入 GtkScrolledWindow，若组件本身具有原生卷动能力（例如 GtkTextView），则可使用 gtk_container_add()，若组件本身不具备有原生卷动能力（例如 GtkImage），则可以使用 gtk_scrolled_window_add_with_viewport()，View Port 组件的作用，是允许在其中放置一个超过 View Port 大小的组件，您可以看到的是 View Port 设定范围中的部份，gtk_scrolled_window_add_with_viewport()会为组件加上一个 GtkViewPort，然后再将 GtkViewPort 加至 GtkScrolledWindow 的简便函式。

在下面的这个范例程序中，使用 GtkImage 来设定显示图片，在这边使用 GtkScrolledWindow 为其加上滚动条功能：

gtk_scrolled_window_demo.c

```
#include <gtk/gtk.h>

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *image;
    GtkWidget *scrolledWindow;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GtkScrolledWindow");
    gtk_window_set_default_size(GTK_WINDOW(window), 320, 200);

    image = gtk_image_new_from_file("kaka.jpg");
    scrolledWindow = gtk_scrolled_window_new(NULL, NULL);

    gtk_scrolled_window_add_with_viewport(
        GTK_SCROLLED_WINDOW(scrolledWindow), image);
```

```
    gtk_container_add(GTK_CONTAINER(window), scrolledWindow);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

一个执行的结果如下所示：

2.54GtkAlignment、 GtkFixed 与 GtkLayout

GtkAlignment 实际上是 GtkContainer 的子类，可以设定它当中的子组件对齐与大小：

```
GtkContainer
 +----GtkBin
       +----GtkAlignment
```

GtkAlignment 可以设定四个数值，xalign、yalign、xscale 与 yscale，可设定的值为 0.0 到 1.0。xalign 设定组件的靠左（0.0）或靠右对齐（1.0），yalign 设定组件的靠上（0.0）或靠下对齐（1.0），如果两个值都设定为 1.0 则无作用。xscale 与 yscale 设定组件如何扩展以填满所配置的空间，0.0 表示无需填满，1.0 表示完全填满。

下面的范例是个简单的示范：

```
gtk_alignment_demo.c
#include <gtk/gtk.h>
```

```
int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *alignment;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GtkAlignment");
    gtk_window_set_default_size(GTK_WINDOW(window), 320, 200);

    button = gtk_button_new_with_label("Press");
    alignment = gtk_alignment_new(1, 0.3, 0.5, 0.9);

    gtk_container_add(GTK_CONTAINER(alignment), button);
    gtk_container_add(GTK_CONTAINER(window), alignment);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

执行的结果如下所示：

GtkFixed 也是 GtkContainer 的一个子类，它允许组件依所设定的位置来自由摆放：

```
GtkContainer
 +----GtkFixed
```

您可以使用 gtk_fixed_put()来指定位置摆放组件，使用 gtk_fixed_move()来移动组件至指定的位置：

```
void gtk_fixed_put(GtkFixed *fixed,
                   GtkWidget *widget,
                   gint x,
                   gint y);

void gtk_fixed_move(GtkFixed *fixed,
                    GtkWidget *widget,
                    gint x,
                    gint y);
```

一个简单的范例如下所示：

gtk_fixed_demo.c

```
#include <gtk/gtk.h>

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *button1, *button2, *button3;
    GtkWidget *fixed;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GtkFixed");
    gtk_window_set_default_size(GTK_WINDOW(window), 320, 200);

    button1 = gtk_button_new_with_label("Press 1");
    button2 = gtk_button_new_with_label("Press 2");
    button3 = gtk_button_new_with_label("Press 3");

    fixed = gtk_fixed_new();

    gtk_fixed_put(GTK_FIXED(fixed), button1, 10, 10);
    gtk_fixed_put(GTK_FIXED(fixed), button2, 50, 100);
    gtk_fixed_put(GTK_FIXED(fixed), button3, 200, 80);

    gtk_container_add(GTK_CONTAINER(window), fixed);
```

```
    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

一个执行的结果如下所示：

GtkLayout 使用则与 GtkFixed 类似，不过提供的是无限卷动区域，您可以参考 GtkLayout 文件说明 或 Layout Container。

2.55GtkFrame 与 GtkAspectFrame

GtkFrame 是 GtkContainer 的子类，可以容纳子组件，并拥有一个外框及卷标文字：

```
GtkContainer
 +----GtkBin
       +----GtkFrame
```

您可以使用 gtk_frame_new()建立一个 GtkFrame，并在建立时指定卷标文字，或是在之后使用 gtk_frame_set_label ()设定文字，卷标文字的位置预设是左上边，您可以使用 gtk_frame_set_label_align()设定卷标文字的位置：

```
GtkWidget* gtk_frame_new(const gchar *label);

void gtk_frame_set_label(GtkFrame *frame,
                         const gchar *label);

void gtk_frame_set_label_align(GtkFrame *frame,
```

```
                                gfloat xalign,
                                gfloat yalign);
```

gtk_frame_set_label_align()可以设定 xalign 与 yalign，设定值为 0.0 到 1.0，xalign 设定水平对齐，yalign 设定文字的垂直对齐，您也可以使用 gtk_frame_set_shadow_type() 来设定外框样式：

```
void gtk_frame_set_shadow_type(GtkFrame      *frame,
                               GtkShadowType  type);
```

下面的程序是个简单的设定范例：

gtk_frame_demo.c

```
#include <gtk/gtk.h>

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *frame;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GtkFrame");
    gtk_window_set_default_size(GTK_WINDOW(window), 200, 150);

    frame = gtk_frame_new("caterpillar");
    gtk_frame_set_label_align(GTK_FRAME(frame), 0.5, 0.5);
    gtk_frame_set_shadow_type(GTK_FRAME(frame), GTK_SHADOW_ETCHED_OUT);

    gtk_container_add(GTK_CONTAINER(window), frame);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

下图是一个执行的结果画面：

GtkAspectFrame 是 GtkFrame 的子类，使用上与 GtkFrame 类似，但多了 xsize/ysize 的比例设定，可用以限定子组件的长宽比例：

```
GtkWidget* gtk_aspect_frame_new(const gchar *label,
                                gfloat xalign,
                                gfloat yalign,
                                gfloat ratio,
                                gboolean obey_child);
```

xalign 与 yalign 的作用与 GtkFrame 相同，ratio 就是比例设定，obey_child 设定为 TRUE 时，表示依子组件的长宽比例来显示，若设定为 FALSE，则表示依 ratio 的比例设定来显示。

下面的范例是个简单的示范，由于 GtkAspectFrame 的 ratio 设定为 2（xsize/ysize），所以按钮组件的显示将依这个比例：

gtk_aspect_frame_demo.c

```
#include <gtk/gtk.h>

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *frame;
    GtkWidget *button;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GtkAspectFrame");
    gtk_window_set_default_size(GTK_WINDOW(window), 300, 300);

    frame = gtk_aspect_frame_new("caterpillar", 0.5, 0.5, 2, FALSE);
    gtk_frame_set_shadow_type(GTK_FRAME(frame), GTK_SHADOW_ETCHED_OUT);

    button = gtk_button_new_with_label("Press");
    gtk_container_add(GTK_CONTAINER(frame), button);
```

```
    gtk_container_add(GTK_CONTAINER(window), frame);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

一个执行的结果如下所示：

第三章 进阶组件使用

3.1 选单元件

3.11GtkHandleBox

加入 GtkHandleBox 的组件，是一个可以被拿下来（torn off）的组件，或称之为所谓的浮动组件，GtkHandleBox 是 GtkContainer 的子类：

```
GtkContainer
 +----GtkBin
       +----GtkHandleBox
```

这样的拿下（tear off）功能，常见的就是用来制作一个浮动选单，或是一个浮动工具列，下面这个范例是个简单的示范，您可以了解 GtkHandleBox 的设定及功用，范例中包括一个

放进 GtkHandleBox 的 GtkLabel，以及一个放到 GtkVBox 的 GtkTextView：

gtk_handle_box_demo.c

```
#include <gtk/gtk.h>

int main(int argc, char *argv[]) {
	GtkWidget *window;
	GtkWidget *vbox;
	GtkWidget *handleBox;

	gtk_init(&argc, &argv);

	window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
	gtk_window_set_title(GTK_WINDOW(window), "GtkHandleBox");
	gtk_window_set_default_size(GTK_WINDOW(window), 300, 200);

	vbox = gtk_vbox_new(FALSE, 5);

	handleBox = gtk_handle_box_new();
	gtk_handle_box_set_handle_position(
		GTK_HANDLE_BOX(handleBox), GTK_POS_LEFT);
	gtk_container_add(GTK_CONTAINER(handleBox),
				gtk_label_new("\n 选单位置\n"));

	gtk_box_pack_start(GTK_BOX(vbox), handleBox, FALSE, FALSE,  5);
	gtk_box_pack_start(GTK_BOX(vbox), gtk_text_view_new(), TRUE, TRUE, 5);

	gtk_container_add(GTK_CONTAINER(window), vbox);

	g_signal_connect(GTK_OBJECT(window), "destroy",
				G_CALLBACK(gtk_main_quit), NULL);

	gtk_widget_show_all(window);

	gtk_main();

	return 0;
}
```

gtk_handle_box_set_handle_position()是用来设定浮动组件的停驻位置。下图是一个执行的结果画面：

下图是把那个 GtkHJandleBox「拿下来」的一个画面：

3.12GtkMenuBar、 GtkMenu 与 GtkMenuItem

GtkMenuBar、GtkMenu、GtkMenuItem 是用来制作窗口程序的选单功能，以下是三个组件的作用示意图：

图中的 File、Edit、About、Open、Save、分隔线、Close 等项目，都是 GtkMenuItem 的实例，在 File 底下有子选单，这是 GtkMenu 的实例，窗口上方则为 GtkMenuBar 的实例。

上图中，File、Edit、About 附加至 GtkMenuBar 中：

```
GtkMenuBar *menubar = gtk_menu_bar_new();
// 将 File、Edit、About 附加至 GtkMenuBar 中
gtk_menu_bar_append(menubar, createFileMenuItem());
gtk_menu_bar_append(menubar, gtk_menu_item_new_with_mnemonic("_Edit"));
gtk_menu_bar_append(menubar, gtk_menu_item_new_with_mnemonic("_About"));
```

Open、Save、分隔线、Close 加入至 GtkMenu，并设定为 File 这个 GtkMenuItem 的子选单：

```
GtkWidget *rootFileItem;
GtkWidget *fileMenu;
GtkWidget *openMenuItem;
GtkWidget *saveMenuItem;
GtkWidget *closeMenuItem;

rootFileItem = gtk_menu_item_new_with_mnemonic("_File");
fileMenu = gtk_menu_new();
openMenuItem = gtk_menu_item_new_with_label("Open");
saveMenuItem = gtk_menu_item_new_with_label("Save");
closeMenuItem = gtk_menu_item_new_with_label("Close");

// 将 Open、Save、分隔线、Close 这些 GtkMenuItem 加入 GtkMenu 中
gtk_menu_shell_append(GTK_MENU_SHELL(fileMenu), openMenuItem);
gtk_menu_shell_append(GTK_MENU_SHELL(fileMenu), saveMenuItem);
gtk_menu_shell_append(GTK_MENU_SHELL(fileMenu), gtk_separator_menu_item_new());
gtk_menu_shell_append(GTK_MENU_SHELL(fileMenu), closeMenuItem);

// 设定为 File 这个 GtkMenuItem 的子选单
gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootFileItem), fileMenu);
```

当 GtkMenuItem 被选中时，会发出 activate 的 Signal，您可以用以连结进行相对应处理的 callback 函式。

下面的范例是个简单的示范，程序中若按下 Open、Save，会在文字模式下显示对应的文字，按下 Close 则会关闭窗口，执行的结果就是上面的图所表示的：

gtk_menu_demo.c

```
#include <gtk/gtk.h>
```

```
void itemPressed(GtkMenuItem *menuItem, gpointer data) {
	g_print("%s\n", data);
}

GtkWidget* createFileMenuItem() {
	GtkWidget *rootFileItem;
	GtkWidget *fileMenu;
	GtkWidget *openMenuItem;
	GtkWidget *saveMenuItem;
	GtkWidget *closeMenuItem;

	rootFileItem = gtk_menu_item_new_with_mnemonic("_File");
	fileMenu = gtk_menu_new();
	openMenuItem = gtk_menu_item_new_with_label("Open");
	saveMenuItem = gtk_menu_item_new_with_label("Save");
	closeMenuItem = gtk_menu_item_new_with_label("Close");

	gtk_menu_shell_append(GTK_MENU_SHELL(fileMenu), openMenuItem);
	gtk_menu_shell_append(GTK_MENU_SHELL(fileMenu), saveMenuItem);
	gtk_menu_shell_append(GTK_MENU_SHELL(fileMenu),
				gtk_separator_menu_item_new());
	gtk_menu_shell_append(GTK_MENU_SHELL(fileMenu), closeMenuItem);

	gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootFileItem), fileMenu);

	g_signal_connect(GTK_OBJECT(openMenuItem), "activate",
				G_CALLBACK(itemPressed), "Open ....");
	g_signal_connect(GTK_OBJECT(saveMenuItem), "activate",
				G_CALLBACK(itemPressed), "Save ....");
	g_signal_connect(GTK_OBJECT(closeMenuItem), "activate",
				G_CALLBACK(gtk_main_quit), NULL);

	return rootFileItem;
}

int main(int argc, char *argv[]) {
	GtkWidget *window;
	GtkWidget *vbox;
	GtkWidget *menubar;

	gtk_init(&argc, &argv);

	window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
```

```
    gtk_window_set_title(GTK_WINDOW(window), "GtkMenuBar");
    gtk_window_set_default_size(GTK_WINDOW(window), 300, 200);

    menubar = gtk_menu_bar_new();

    gtk_menu_bar_append(menubar, createFileMenuItem());
    gtk_menu_bar_append(menubar, gtk_menu_item_new_with_mnemonic("_Edit"));
    gtk_menu_bar_append(menubar, gtk_menu_item_new_with_mnemonic("_About"));

    vbox = gtk_vbox_new(FALSE, 5);
    gtk_box_pack_start(GTK_BOX(vbox), menubar, FALSE, FALSE, 2);

    gtk_container_add(GTK_CONTAINER(window), vbox);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                        G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

您可以将 GtkMenuBar 置入 GtkHandleBox 中，这会让 GtkMenuBar 成为一个可拿下来（tear off）的浮动选单列，例如：

```
    ....
    handleBox = gtk_handle_box_new();
    gtk_container_add(GTK_CONTAINER(handleBox), menubar);

    vbox = gtk_vbox_new(FALSE, 5);
    gtk_box_pack_start(GTK_BOX(vbox), handleBox, FALSE, FALSE, 2);

    gtk_container_add(GTK_CONTAINER(window), vbox);
    ....
```

一个执行时的结果如下所示：

把选单拿下来成为浮动的画面如下所示：

3.13GtkCheckMenuItem、 GtkRadioMenuItem 与 GtkTearoffMenuItem

GtkCheckMenuItem、GtkRadioMenuItem 与 GtkTearoffMenuItem 都是 GtkMenuItem 的子类，GtkCheckMenuItem 可以产生一个复选框，GtkRadioMenuItem 产生单选钮，它们的使用上与 GtkCheckButton 与 GtkRadioButton 类似。

由于 GtkRadioMenuItem 是单选钮，因此必须设定其群组，第一次产生 GtkRadioMenuItem 时群组可设定为 NULL，第二次则使用 gtk_radio_menu_item_group()取得第一个 GtkRadioMenuItem 的群组（GSList），然后再用以产生第二个 GtkRadioMenuIte：

```
cutMenuItem = gtk_radio_menu_item_new_with_label(NULL, "Cut");
copyMenuItem = gtk_radio_menu_item_new_with_label(
                    gtk_radio_menu_item_group(GTK_RADIO_MENU_ITEM(cutMenuItem)),
                    "Copy");
```

GtkTearoffMenuItem 是个特殊的选项，外观为一个虚线，按下后，会使得整个 GtkMenu 剥离（tear off）而成为浮动选单。

您可以改写一下 GtkMenuBar、GtkMenu 与 GtkMenuItem 中的范例，新增一个函式：

```
...
GtkWidget* createEditMenuItem() {
    GtkWidget *rootEditItem;
    GtkWidget *editMenu;
    GtkWidget *cutMenuItem;
    GtkWidget *copyMenuItem;
    GtkWidget *pasteMenuItem;

    rootEditItem = gtk_menu_item_new_with_mnemonic("_Edit");
    editMenu = gtk_menu_new();
    cutMenuItem = gtk_radio_menu_item_new_with_label(NULL, "Cut");
    copyMenuItem = gtk_radio_menu_item_new_with_label(
                gtk_radio_menu_item_group(GTK_RADIO_MENU_ITEM(cutMenuItem)),
                "Copy");
    pasteMenuItem = gtk_check_menu_item_new_with_label("Paste");

    gtk_menu_shell_append(GTK_MENU_SHELL(editMenu),
                          gtk_tearoff_menu_item_new());
    gtk_menu_shell_append(GTK_MENU_SHELL(editMenu), cutMenuItem);
    gtk_menu_shell_append(GTK_MENU_SHELL(editMenu), copyMenuItem);
    gtk_menu_shell_append(GTK_MENU_SHELL(editMenu), pasteMenuItem);

    gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootEditItem), editMenu);

    return rootEditItem;
}
...
```

然后改一下 main()函式中附加 GtkMenuBar 选项 Edit 的部份：

```
gtk_menu_bar_append(menubar, createEditMenuItem());
```

下图为改写后，Edit 选单的画面：

下图为 Edit 选单成为浮动的一个画面：

3.14GtkUIManager

使用撰写程序的方式来建构选单、工具列等使用者界面，过程有时过于繁琐，您可以使用 GtkUIManager 从一个或多个使用者接口定义文件读取接口定义，并自动建立相对应的 Gtk 组件，使用者接口定义文件是一个 XML 档案。

举个实际的例子来说，可以改写一下 GtkMenuBar、GtkMenu 与 GtkMenuItem 中的范例，使用 GtkUIManager 与 XML 定义档来作出相同的效果，若 XML 定义档如下所示：

gtk_ui_manager.xml

```
<ui>
  <menubar name="MenuBar">
    <menu action="File">
      <menuitem action="Open"/>
      <menuitem action="Save"/>
      <separator/>
      <menuitem action="Close"/>
    </menu>
```

```
    <menu action="Edit">
      <menuitem action="Cut"/>
      <menuitem action="Copy"/>
      <menuitem action="Paste"/>
    </menu>
    <menu action="Help">
      <menuitem action="About"/>
    </menu>
  </menubar>
</ui>
```

"name"属性可以让您在建构程序的时候，依名称来取得相对应的 Gtk 组件，而"action"将对应于 GtkAction，您可以使用 GtkActionEntry 来建构 GtkAction，GtkActionEntry 的定义如下：

```
typedef struct {
  const gchar     *name;
  const gchar     *stock_id;
  const gchar     *label;
  const gchar     *accelerator;
  const gchar     *tooltip;
  GCallback  callback;
} GtkActionEntry;
```

第一个 name 成员即对应定义档中的"name"属性，其它则为图标、文字、快速键、提示与 callback 函式，一个设定范例如下：

```
static GtkActionEntry entries[] = {
  { "File", NULL, "_File" },
  { "Open", GTK_STOCK_OPEN, "Open",
     "<control>O", "Open File", G_CALLBACK(itemPressed)},
  { "Save", GTK_STOCK_SAVE, "Save",
     "<control>S", "Save File", G_CALLBACK(itemPressed)},
  { "Close", GTK_STOCK_QUIT, "Close",
     "<control>Q", "Close File", G_CALLBACK(gtk_main_quit)},
  { "Edit", NULL, "_Edit" },
  { "Cut", NULL, "Copy"},
  { "Copy", NULL, "Copy"},
  { "Paste", NULL, "Paste"},
  { "Help", NULL, "_Help" },
  { "About", NULL, "About" }
};
```

GtkAction 被组织为 GtkActionGrouop，定义了 GtkActionEntry 之后，您可以使用

gtk_action_group_add_actions()函式将之加入 GtkActionGroup 之中：

```
gtk_action_group_add_actions(actionGroup, entries, 10, NULL);
```

接着建构 GtkUIManager，并使用 gtk_ui_manager_insert_action_group() 加入 GtkActionGroup，然后使用 gtk_ui_manager_add_ui_from_file()读取使用者界面定义文件：

```
GtkUIManager *ui = gtk_ui_manager_new();
gtk_ui_manager_insert_action_group(ui, actionGroup, 0);
gtk_ui_manager_add_ui_from_file(ui, "gtk_ui_manager.xml", NULL);
```

GtkUIManager将会自动建构相对应的Gtk组件，并依"action"设定建立相对应的GtkAction。

若要从 GtkUIManager 中取得组件，则可以使用 gtk_ui_manager_get_widget()并依"name"属性之设定来取得，例如取得"MenuBar"并加入 GtkVBox 中：

```
GtkWidget *vbox = gtk_vbox_new(FALSE, 5);
gtk_box_pack_start(GTK_BOX(vbox),
    gtk_ui_manager_get_widget(ui, "/MenuBar"), FALSE, FALSE, 2);
```

下面的程序是个完整的范例：

gtk_ui_manager_demo.c

```
#include <gtk/gtk.h>

void itemPressed(GtkMenuItem *menuItem, gpointer data) {
    g_print("%s\n", gtk_action_get_name(GTK_ACTION(menuItem)));
}

static GtkActionEntry entries[] = {
  { "File", NULL, "_File" },
  { "Open", GTK_STOCK_OPEN, "Open",
            "<control>O", "Open File", G_CALLBACK(itemPressed)},
  { "Save", GTK_STOCK_SAVE, "Save",
            "<control>S", "Save File", G_CALLBACK(itemPressed)},
  { "Close", GTK_STOCK_QUIT, "Close",
            "<control>Q", "Close File", G_CALLBACK(gtk_main_quit)},
  { "Edit", NULL, "_Edit" },
  { "Cut", NULL, "Copy"},
  { "Copy", NULL, "Copy"},
```

```
    { "Paste", NULL, "Paste"},
    { "Help", NULL, "_Help" },
    { "About", NULL, "About" }
};

int main(int argc, char *argv[]) {
        GtkWidget *window;
        GtkActionGroup *actionGroup;
        GtkUIManager *ui;
        GtkWidget *vbox;

        gtk_init(&argc, &argv);

        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        gtk_window_set_title(GTK_WINDOW(window), "GtkUIManager");
        gtk_window_set_default_size(GTK_WINDOW(window), 300, 200);

        actionGroup = gtk_action_group_new("Actions");
        gtk_action_group_add_actions(actionGroup, entries, 10, NULL);

        ui = gtk_ui_manager_new();
        gtk_ui_manager_insert_action_group(ui, actionGroup, 0);
        gtk_ui_manager_add_ui_from_file(ui, "gtk_ui_manager.xml", NULL);

        vbox = gtk_vbox_new(FALSE, 5);
        gtk_box_pack_start(GTK_BOX(vbox),
                gtk_ui_manager_get_widget(ui, "/MenuBar"), FALSE, FALSE, 2);

        gtk_container_add(GTK_CONTAINER(window), vbox);

        g_signal_connect(GTK_OBJECT(window), "destroy",
                                G_CALLBACK(gtk_main_quit), NULL);

        gtk_widget_show_all(window);

        gtk_main();

        return 0;
}
```

一个执行的画面如下所示：

更详细的 GtkUIManager 使用，可以参考文件 GtkUIManager，或是 gtk-demo 中的 UI Manager 范例。

3.2 列组件

3.21GtkProgressBar

GtkProgressBar 常用来显示目前的工作进度，例如程序安装、档案复制、下载等，可以使用 gtk_progress_bar_new()建立一个 GtkProgressBar，若要设定进度比例，可以使用 gtk_progress_bar_set_fraction()函式，可设定的值为 0.0 到 1.0，相对应的要取得目前进度比例，可以使用 gtk_progress_bar_get_fraction()函式。

GtkProgressBar 可以使用 gtk_progress_bar_set_text()设定文字显示，使用 gtk_progress_bar_get_text()取得文字，另外，您还可以使用 gtk_progress_bar_set_orientation()设定进度列的移动方向，可设定的值有：

GTK_PROGRESS_LEFT_TO_RIGHT：从左向右（预设）
GTK_PROGRESS_RIGHT_TO_LEFT：从右向左
GTK_PROGRESS_BOTTOM_TO_TOP： 从下向上
GTK_PROGRESS_TOP_TO_BOTTOM：从上向下

下面的程序使用 gtk_timeout_add() 设定一个定时器，仿真进度递增的状况：

gtk_progress_bar_demo.c

```
#include <gtk/gtk.h>

gboolean timeout_callback(gpointer data) {
    gdouble value;
    GString *text;

    value = gtk_progress_bar_get_fraction(GTK_PROGRESS_BAR(data));

    value += 0.01;
```

```
        if(value > 1.0) {
            value = 0.0;
        }

        gtk_progress_bar_set_fraction(GTK_PROGRESS_BAR(data), value);

        text = g_string_new(
                    gtk_progress_bar_get_text(GTK_PROGRESS_BAR(data)));
        g_string_sprintf(text, "%d %%", (int) (value * 100));

        gtk_progress_bar_set_text(GTK_PROGRESS_BAR(data), text->str);

        return TRUE;
}

int main(int      argc, char *argv[]) {
        GtkWidget *window;
        GtkWidget *progressBar;
        gint timer;

        gtk_init(&argc, &argv);

        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        gtk_window_set_title(GTK_WINDOW(window), "GtkProgressBar");
        gtk_window_set_default_size(GTK_WINDOW(window), 300, 30);

        progressBar = gtk_progress_bar_new();
        timer = gtk_timeout_add(100, timeout_callback, progressBar);

        gtk_container_add(GTK_CONTAINER(window), progressBar);

        g_signal_connect(GTK_OBJECT(window), "destroy",
                                G_CALLBACK(gtk_main_quit), NULL);

        gtk_widget_show_all(window);

        gtk_main();

        return 0;
}
```

执行的结果如下所示：

3.22GtkToolBar

GtkToolBar 可以让您制作工具列，将一些常用指令群组并依使用者需求而显示于使用接口上，要建立 GtkToolBar，只要使用 gtk_toolbar_new()函式，要插入一个项目，则使用 gtk_toolbar_insert()，所插入的项目是 GtkToolItem 的实例，而 GtkToolItem 可以直接使用 gtk_tool_item_set_tooltip_text()设定提示文字，但若您要设定影像与文字，则必须知道，GtkToolItem 是 GtkBin 的子类这个事实：

```
GtkContainer
 +----GtkBin
       +----GtkToolItem
```

所以若您要插入影像或文字,或者是其它的组件,例如按钮等,则要类似 影像及文字按钮 中介绍的方式自行处理。

下面的范例，使用 GtkMenuBar、GtkMenu 与 GtkMenuItem 中的范例为基础，加上了工具列的功能，其中关于 GtkToolBar 设定相关的部份，已使用粗体字加以标示：

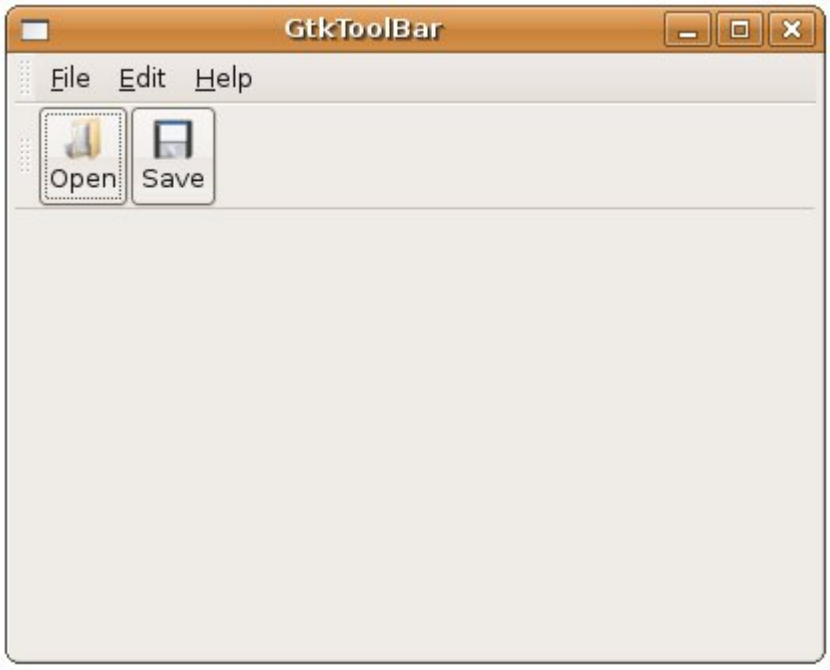
gtk_tool_bar_demo.c

```
#include <gtk/gtk.h>

void itemPressed(GtkMenuItem *menuItem, gpointer data) {
    g_print("%s\n", data);
}

GtkWidget* createFileMenuItem() {
    GtkWidget *rootFileItem;
    GtkWidget *fileMenu;
    GtkWidget *openMenuItem;
    GtkWidget *saveMenuItem;
    GtkWidget *closeMenuItem;

    rootFileItem = gtk_menu_item_new_with_mnemonic("_File");
    fileMenu = gtk_menu_new();
    openMenuItem = gtk_menu_item_new_with_label("Open");
    saveMenuItem = gtk_menu_item_new_with_label("Save");
    closeMenuItem = gtk_menu_item_new_with_label("Close");

    gtk_menu_shell_append(GTK_MENU_SHELL(fileMenu), openMenuItem);
    gtk_menu_shell_append(GTK_MENU_SHELL(fileMenu), saveMenuItem);
    gtk_menu_shell_append(GTK_MENU_SHELL(fileMenu),
                          gtk_separator_menu_item_new());
```

```
    gtk_menu_shell_append(GTK_MENU_SHELL(fileMenu), closeMenuItem);

    gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootFileItem), fileMenu);

    g_signal_connect(GTK_OBJECT(openMenuItem), "activate",
                     G_CALLBACK(itemPressed), "Open ....");
    g_signal_connect(GTK_OBJECT(saveMenuItem), "activate",
                     G_CALLBACK(itemPressed), "Save ....");
    g_signal_connect(GTK_OBJECT(closeMenuItem), "activate",
                     G_CALLBACK(gtk_main_quit), NULL);

    return rootFileItem;
}

// 建立一个内含按钮、文字与图片的 GtkToolItem
GtkToolItem* createToolItem(gchar *stock_id, gchar *text) {
    GtkToolItem *open;
    GtkWidget *box;
    GtkWidget *label;
    GtkWidget *image;
    GtkWidget *button;

    open =  gtk_tool_item_new();
    gtk_tool_item_set_tooltip_text(open, "Open File..");

    box = gtk_vbox_new(FALSE, 0);

    image = gtk_image_new_from_stock(stock_id, GTK_ICON_SIZE_SMALL_TOOLBAR);
    label = gtk_label_new(text);

    gtk_box_pack_start(GTK_BOX(box), image, FALSE, FALSE, 0);
    gtk_box_pack_start(GTK_BOX(box), label, FALSE, FALSE, 0);

    button = gtk_button_new();

    gtk_container_add(GTK_CONTAINER(button), box);
    gtk_container_add(GTK_CONTAINER(open), button);

    return open;
}

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *menubarBox;
```

```
    GtkWidget *toolbarBox;
    GtkWidget *vbox;
    GtkWidget *menubar;
    GtkWidget *toolbar;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GtkToolBar");
    gtk_window_set_default_size(GTK_WINDOW(window), 400, 300);

    menubar = gtk_menu_bar_new();

    gtk_menu_bar_append(menubar, createFileMenuItem());
    gtk_menu_bar_append(menubar, gtk_menu_item_new_with_mnemonic("_Edit"));
    gtk_menu_bar_append(menubar, gtk_menu_item_new_with_mnemonic("_Help"));

    vbox = gtk_vbox_new(FALSE, 0);

    menubarBox = gtk_handle_box_new();
    gtk_container_add(GTK_CONTAINER(menubarBox), menubar);
    gtk_box_pack_start(GTK_BOX(vbox), menubarBox, FALSE, FALSE, 0);

    toolbar = gtk_toolbar_new();
    gtk_toolbar_insert(GTK_TOOLBAR(toolbar),
                       createToolItem(GTK_STOCK_OPEN, "Open"), 0);
    gtk_toolbar_insert(GTK_TOOLBAR(toolbar),
                       createToolItem(GTK_STOCK_SAVE, "Save"), 1);

    toolbarBox = gtk_handle_box_new();
    gtk_container_add(GTK_CONTAINER(toolbarBox), toolbar);
    gtk_box_pack_start(GTK_BOX(vbox), toolbarBox, FALSE, FALSE, 0);

    gtk_container_add(GTK_CONTAINER(window), vbox);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

执行的结果如下所示：



3.23GtkStatusBar

状态列通常位于窗口的底部，用以显示目前窗口操作状况的一些简单讯息，在 GTK 中的状态列组件是 GtkStatusBar，您可以使用 gtk_statusbar_new()来建立。

窗口中各个组件或操作都可以有相对应的状态讯息，为了让状态列区别哪个讯息属于哪个组件或操作，GtkStatusBar 使用 Context ID 来加以识别，您可以使用 gtk_statusbar_get_context_id()并给定一个名称以取得对应的 Context ID，例如：

```
GtkWidget *statusbar;
gint contextId;
statusbar = gtk_statusbar_new();
contextId = gtk_statusbar_get_context_id(
                GTK_STATUSBAR(statusbar), "Editor Messages");
```

取得 Context ID 之后，若要向 GtkStatusBar 加入或移除状态讯息，可以使用以下的几个函式：

```
guint gtk_statusbar_push(GtkStatusbar *statusbar,
                         guint context_id,
                         const gchar *text);

void gtk_statusbar_pop(GtkStatusbar *statusbar,
                       guint context_id);
```

```
void gtk_statusbar_remove(GtkStatusbar *statusbar,
                          guint context_id,
                          guint message_id);
```

GtkStatusBar 的讯息采堆栈（Stack）储存，采后进先出的方式，若要加入讯息，则使用 gtk_statusbar_push()，这会传回一个 message_id，您可以根据 message_id 与 context_id 使用 gtk_statusbar_remove()函式指定删除该讯息，而 gtk_statusbar_pop()则是直接删除 Context ID 所对应的堆栈讯息中最上面的讯息。

下面的程序以 GtkToolBar 为基础，增加一个 GtkTextView 与 GtkStatusBar，让执行后的外观更像一个窗口程序文字编辑器：

gtk_statusbar_demo.c

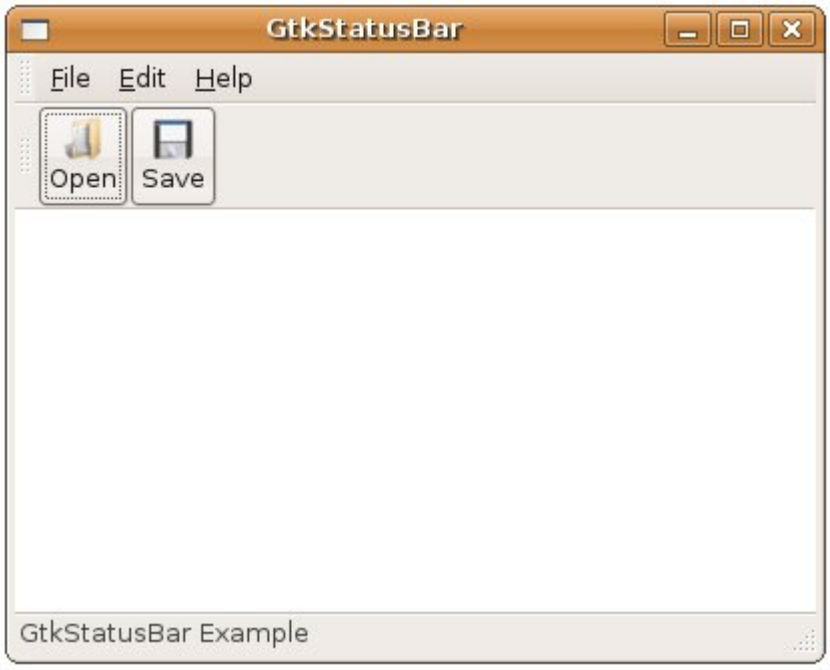
```
#include <gtk/gtk.h>

void itemPressed(GtkMenuItem *menuItem, gpointer data) {
    g_print("%s\n", data);
}

GtkWidget* createFileMenuItem() {
    GtkWidget *rootFileItem;
    GtkWidget *fileMenu;
    GtkWidget *openMenuItem;
    GtkWidget *saveMenuItem;
    GtkWidget *closeMenuItem;

    rootFileItem = gtk_menu_item_new_with_mnemonic("_File");
    fileMenu = gtk_menu_new();
    openMenuItem = gtk_menu_item_new_with_label("Open");
    saveMenuItem = gtk_menu_item_new_with_label("Save");
    closeMenuItem = gtk_menu_item_new_with_label("Close");

    gtk_menu_shell_append(GTK_MENU_SHELL(fileMenu), openMenuItem);
    gtk_menu_shell_append(GTK_MENU_SHELL(fileMenu), saveMenuItem);
    gtk_menu_shell_append(GTK_MENU_SHELL(fileMenu),
                          gtk_separator_menu_item_new());
    gtk_menu_shell_append(GTK_MENU_SHELL(fileMenu), closeMenuItem);

    gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootFileItem), fileMenu);

    g_signal_connect(GTK_OBJECT(openMenuItem), "activate",
                     G_CALLBACK(itemPressed), "Open ....");
```

```
        g_signal_connect(GTK_OBJECT(saveMenuItem), "activate",
                                G_CALLBACK(itemPressed), "Save ....");
        g_signal_connect(GTK_OBJECT(closeMenuItem), "activate",
                                G_CALLBACK(gtk_main_quit), NULL);

        return rootFileItem;
}

// 建立一个内含按钮、文字与图片的 GtkToolItem
GtkToolItem* createToolItem(gchar *stock_id, gchar *text) {
        GtkToolItem *open;
        GtkWidget *box;
        GtkWidget *label;
        GtkWidget *image;
        GtkWidget *button;

        open =    gtk_tool_item_new();
        gtk_tool_item_set_tooltip_text(open, "Open File..");

        box = gtk_vbox_new(FALSE, 0);

        image = gtk_image_new_from_stock(stock_id, GTK_ICON_SIZE_SMALL_TOOLBAR);
        label = gtk_label_new(text);

        gtk_box_pack_start(GTK_BOX(box), image, FALSE, FALSE, 0);
        gtk_box_pack_start(GTK_BOX(box), label, FALSE, FALSE, 0);

        button = gtk_button_new();

        gtk_container_add(GTK_CONTAINER(button), box);
        gtk_container_add(GTK_CONTAINER(open), button);

        return open;
}

int main(int argc, char *argv[]) {
        GtkWidget *window;
        GtkWidget *menubarBox;
        GtkWidget *toolbarBox;
        GtkWidget *vbox;
        GtkWidget *menubar;
        GtkWidget *toolbar;
        GtkWidget *statusbar;
        gint contextId;
```

```
gtk_init(&argc, &argv);

window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
gtk_window_set_title(GTK_WINDOW(window), "GtkStatusBar");
gtk_window_set_default_size(GTK_WINDOW(window), 400, 300);

menubar = gtk_menu_bar_new();

gtk_menu_bar_append(menubar, createFileMenuItem());
gtk_menu_bar_append(menubar, gtk_menu_item_new_with_mnemonic("_Edit"));
gtk_menu_bar_append(menubar, gtk_menu_item_new_with_mnemonic("_Help"));

vbox = gtk_vbox_new(FALSE, 0);

menubarBox = gtk_handle_box_new();
gtk_container_add(GTK_CONTAINER(menubarBox), menubar);
gtk_box_pack_start(GTK_BOX(vbox), menubarBox, FALSE, FALSE, 0);

toolbar = gtk_toolbar_new();
gtk_toolbar_insert(GTK_TOOLBAR(toolbar),
                           createToolItem(GTK_STOCK_OPEN, "Open"), 0);
gtk_toolbar_insert(GTK_TOOLBAR(toolbar),
                           createToolItem(GTK_STOCK_SAVE, "Save"), 1);

toolbarBox = gtk_handle_box_new();
gtk_container_add(GTK_CONTAINER(toolbarBox), toolbar);
gtk_box_pack_start(GTK_BOX(vbox), toolbarBox, FALSE, FALSE, 0);

gtk_box_pack_start(GTK_BOX(vbox), gtk_text_view_new(), TRUE, TRUE, 0);

statusbar = gtk_statusbar_new();
contextId = gtk_statusbar_get_context_id(
                    GTK_STATUSBAR(statusbar), "Editor Messages");
gtk_statusbar_push(GTK_STATUSBAR(statusbar),
                             contextId, "GtkStatusBar Example");
gtk_box_pack_start(GTK_BOX(vbox), statusbar, FALSE, FALSE, 0);

gtk_container_add(GTK_CONTAINER(window), vbox);

g_signal_connect(GTK_OBJECT(window), "destroy",
                        G_CALLBACK(gtk_main_quit), NULL);

gtk_widget_show_all(window);
```

```
    gtk_main();

    return 0;
}
```

执行的结果画面如下所示：



3.3 其它组件

3.31GtkStyle 与资源档

GtkStyle 是 GTK 与 GDK 之间一个重要的抽象层，它允许您自订组件的外观样式，而不用直接亲自处理 GDK 的绘图，GtkStyle 为 GtkWidget 的成员之一：

```
typedef struct {
  /* The style for the widget. The style contains the
   *  colors the widget should be drawn in for each state
   *  along with graphics contexts used to draw with and
   *  the font to use for text.
   */
  GtkStyle *GSEAL (style);
   ....
} GtkWidget;
```

如说明文件中所称的，GtkStyle 包括了组件的颜色信息，以及每个状态下如何绘制的讯息：

```
typedef struct _GtkStyle GtkStyle;
```

```
struct _GtkStyle {
  GtkStyleClass *klass;

  GdkColor fg[5];
  GdkColor bg[5];
  GdkColor light[5];
  GdkColor dark[5];
  GdkColor mid[5];
  GdkColor text[5];
  GdkColor base[5];

  GdkColor black;
  GdkColor white;
  GdkFont *font;

  GdkGC *fg_gc[5];
  GdkGC *bg_gc[5];
  GdkGC *light_gc[5];
  GdkGC *dark_gc[5];
  GdkGC *mid_gc[5];
  GdkGC *text_gc[5];
  GdkGC *base_gc[5];
  GdkGC *black_gc;
  GdkGC *white_gc;

  GdkPixmap *bg_pixmap[5];

  /* private */

  gint ref_count;
  gint attach_count;

  gint depth;
  GdkColormap *colormap;

  GtkThemeEngine *engine;

  gpointer        engine_data;

  GtkRcStyle     *rc_style;

  GSList         *styles;
};
```

您可以使用 GtkWidget 的 gtk_widget_style_get() 来取得 GtkStyle，使用 gtk_widget_set_style()函式设定 GtkStyle 等，一个使用的范例在 GtkColorButton 与 GtkColorSelectionDialog 可以找到，在 GtkStyle 定义中您可以发现的是，它们都是有五个元素的数组，这是因为可以区分为五个不同状态下的显示颜色：

GTK_STATE_NORMAL：一般操作状态
GTK_STATE_ACTIVE：活动状态，例如按下按钮
GTK_STATE_PRELIGHT：若组件可以响应鼠标按下，而鼠标越过组件的状态
GTK_STATE_SELECTED：被选择的状态
GTK_STATE_INSENSITIVE：指出不响应使用者动作的状态

GtkStyle 的样式可以撰写在一个资源文件中，例如写一个.rc 檔如下：

styles.rc

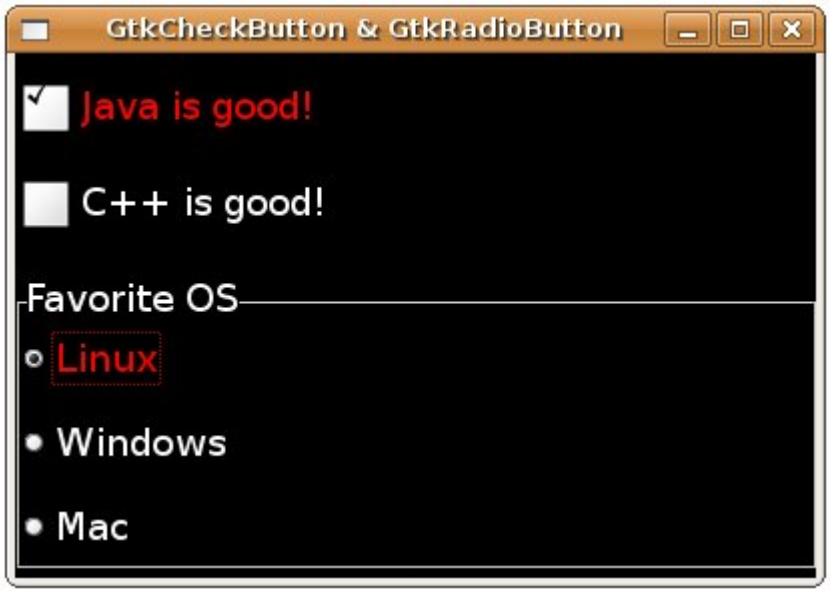
```
style "widgets" {
      fg[ACTIVE] = "#FF0000"
      fg[SELECTED] = "#003366"
      fg[NORMAL] = "#FFFFFF"
      fg[PRELIGHT] = "#FFFFFF"
      fg[INSENSITIVE] = "#999999"

      bg[ACTIVE] = "#003366"
      bg[SELECTED] = "#FFFFFF"
      bg[NORMAL] = "#000000"
      bg[PRELIGHT] = "#003366"
      bg[INSENSITIVE] = "#666666"
}

style "labels" = "widgets" {
      font_name = "Algerian 14"
}
style "buttons" = "widgets" {
      GtkButton::inner-border = { 10, 10, 10, 10 }
}

style "radios" = "buttons" {
      GtkCheckButton::indicator-size = 10
}

style "checks" = "buttons" {
      GtkCheckButton::indicator-size = 25
}
```

class "GtkWindow" style "widgets"
class "GtkLabel" style "labels"
class "GtkCheckButton" style "checks"
class "GtkRadioButton" style "radios"
class "Gtk*Button" style "buttons"
语法上非常简单，每个样式可有一个名称，并指定要修改的属性，样式之间还可以继承，例如上面的设定中，"labels" = "widgets"表示"labels"继承 "widgets"的属性设定，最底下是样式名称的取名，而最后一行，"Gtk*Button"表示设定 Gtk 开头而 Button 结尾的类别名称都符合。

可以将这个.rc 档案套用至 GtkCheckButton 与 GtkRadioButton 的范例中，只要在 gtk_init()后写下 gtk_rc_parse()函式并指定样式档案：

```
....
gtk_init(&argc, &argv);
gtk_rc_parse("styles.rc");
....
```

重新编译并执行程序，一个执行范例如下所示：



3.32GtkLabel

到目前为止一直在用的 GtkLabel，其实可以设定更多的样式，例如简单的标记（markup）、换行、对齐、自动换行等。

要设定标记可以使用 gtk_label_set_markup()函式，要设定换行，可以使用'\n'来断行，或是使用 gtk_label_set_line_wrap()设定依容器宽度自动换行，要设定对齐可以使用 gtk_label_set_justify()，预设是置中对齐，可以设定的值包括了：

GTK_JUSTIFY_CENTER：置中对齐(预设)

GTK_JUSTIFY_LEFT：靠左对齐
GTK_JUSTIFY_RIGHT：靠右对齐
GTK_JUSTIFY_FILL：左右对齐

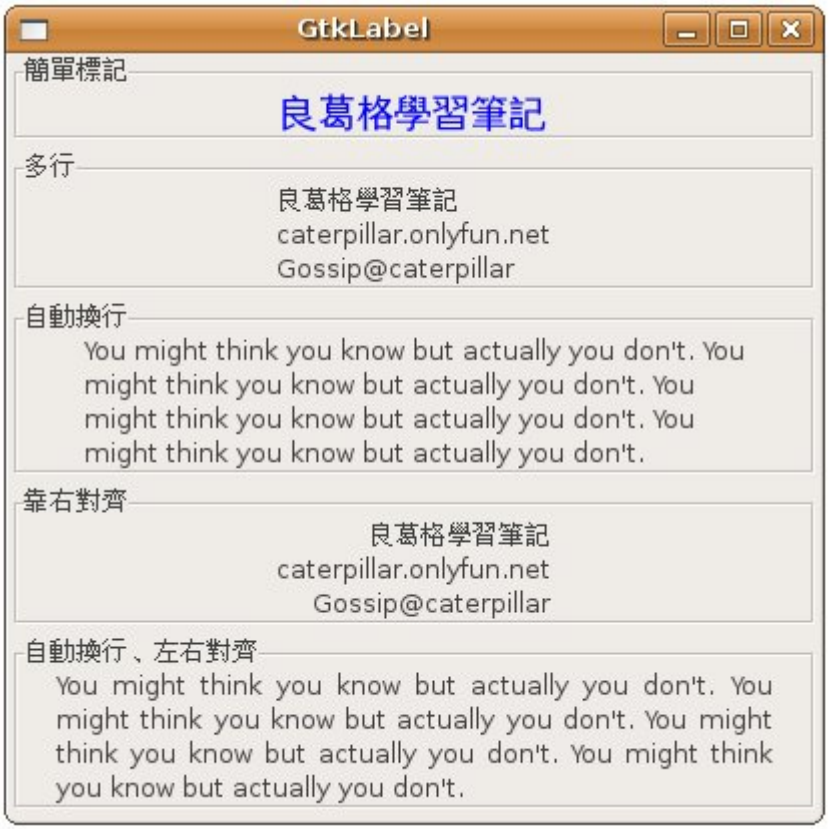
以下使用一个程序实例，来看看设定过程与执行后的效果：

gtk_label_demo.c

```
#include <gtk/gtk.h>

int main(int      argc, char *argv[]) {
      GtkWidget *window;
      GtkWidget *vbox;
      GtkWidget *frame;
      GtkWidget *label;

      gtk_init(&argc, &argv);

      window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
      gtk_window_set_title(GTK_WINDOW(window), "GtkLabel");
      gtk_window_set_default_size(GTK_WINDOW(window), 400, 400);

      vbox = gtk_vbox_new(FALSE, 5);

      frame = gtk_frame_new("简单标记");
      label = gtk_label_new(NULL);
      gtk_label_set_markup(GTK_LABEL(label),
          "<span foreground='blue' size='x-large'>良葛格学习笔记</span>");
      gtk_container_add(GTK_CONTAINER(frame), label);
      gtk_box_pack_start(GTK_BOX(vbox), frame, FALSE, FALSE, 0);

      frame = gtk_frame_new("多行");
      label = gtk_label_new("良葛格学习笔记\ncaterpillar.onlyfun.net\n" \
                  "Gossip@caterpillar");
      gtk_container_add(GTK_CONTAINER(frame), label);
      gtk_box_pack_start(GTK_BOX(vbox), frame, FALSE, FALSE, 0);

      frame = gtk_frame_new("自动换行");
      label = gtk_label_new("You might think you know but actually you don't. " \
                                    "You might think you know but actually you don't. " \
                                    "You might think you know but actually you don't. " \
                                    "You might think you know but actually you don't.");
      gtk_label_set_line_wrap(GTK_LABEL(label), TRUE);
      gtk_container_add(GTK_CONTAINER(frame), label);
      gtk_box_pack_start(GTK_BOX(vbox), frame, FALSE, FALSE, 0);
```

```
    frame = gtk_frame_new("靠右对齐");
    label = gtk_label_new("良葛格学习笔记\ncaterpillar.onlyfun.net\n" \
                          "Gossip@caterpillar");
    gtk_label_set_justify(GTK_LABEL(label), GTK_JUSTIFY_RIGHT);
    gtk_container_add(GTK_CONTAINER(frame), label);
    gtk_box_pack_start(GTK_BOX(vbox), frame, FALSE, FALSE, 0);

    frame = gtk_frame_new("自动换行、左右对齐");
    label = gtk_label_new("You might think you know but actually you don't. " \
                          "You might think you know but actually you don't. " \
                          "You might think you know but actually you don't. " \
                          "You might think you know but actually you don't.");
    gtk_label_set_line_wrap(GTK_LABEL(label), TRUE);
    gtk_label_set_justify(GTK_LABEL(label), GTK_JUSTIFY_FILL);
    gtk_container_add(GTK_CONTAINER(frame), label);
    gtk_box_pack_start(GTK_BOX(vbox), frame, FALSE, FALSE, 0);

    gtk_container_add(GTK_CONTAINER(window), vbox);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

执行结果如下所示：



3.33GtkScale

GtkScale 可以是水平或垂直的拉杆组件，您可以拉动上面的拉杆，拉杆的位置代表某个值并可以加以显示，您可以使用 gtk_hscale_new_with_range()建立一个水平并具有上、下界及递增值拉杆组件，gtk_vscale_new_with_range()则是垂直组件，GtkScale 基本上会显示浮点数，您可以使用 gtk_scale_set_digits()设定显示的位数，数字的显示位置可以有上、下、左
、右，可使用 gtk_scale_set_value_pos()来设定，可设定的值为 GTK_POS_TOP、GTK_POS_BUTTOM、GTK_POS_LEFT 与 GTK_POS_RIGHT。

下面的程序是个简单的示范：

gtk_scale_demo.c

```
#include <gtk/gtk.h>

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *scale;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
```

```
    gtk_window_set_title(GTK_WINDOW(window), "GtkScale");
    gtk_window_set_default_size(GTK_WINDOW(window), 200, 30);

    scale = gtk_hscale_new_with_range(0.0, 100.0, 1.0);
    gtk_scale_set_digits(GTK_SCALE(scale), 0);
    gtk_scale_set_value_pos(GTK_SCALE(scale), GTK_POS_TOP);

    gtk_container_add(GTK_CONTAINER(window), scale);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

一个执行的结果画面如下所示：

3.34GtkEntryCompletion

先前看过 GtkEntry 的范例，主要是作为使用者输入文字之用，您可以搭配 GtkEntryCompletion 来让 GtkEntry 拥有自动完成功能，这需要使用到 GtkListStore 与 GtkTreeIter，这两个类别在 GtkComboBox 与 GtkListStore 曾经介绍过，只要您会使用 GtkEntry、GtkListStore 与 GtkTreeIter，制作自动完成就不是什么困难的事。

以下直接使用实例来示范：

gtk_entrycompletion_demo.c

```
#include<gtk/gtk.h>

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *label;
    GtkWidget *entry;
    GtkEntryCompletion *completion;
    GtkListStore *store;
    GtkTreeIter iter;
```

```
int i;

// 作为自动完成时的项目提示
gchar *topics[] = {
            "C", "C++", "Java", "JSP", "JSF", "JUnit", "JavaScript" };

gtk_init (&argc, &argv);

window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
gtk_window_set_title(GTK_WINDOW(window), "GtkEntryCompletion");
gtk_window_set_default_size(GTK_WINDOW(window), 300, 50);

// 使用 GtkListStore 储存项目提示
store = gtk_list_store_new(1, G_TYPE_STRING);
for(i = 0; i < 7; i++) {
    gtk_list_store_append(store, &iter);
    gtk_list_store_set(store, &iter, 0, topics[i], -1);
}

// 将 GtkListStore 设定给 GtkEntryCompletion
completion = gtk_entry_completion_new();
gtk_entry_completion_set_model(completion, GTK_TREE_MODEL(store));
gtk_entry_completion_set_text_column (completion, 0);

label = gtk_label_new("请输入技术主题");

// 建立 GtkEntry
entry = gtk_entry_new();

// 设定 GtkEntryCompletion
gtk_entry_set_completion(GTK_ENTRY(entry), completion);

vbox = gtk_vbox_new(FALSE, 5);
gtk_box_pack_start(GTK_BOX (vbox), label, FALSE, FALSE, 0);
gtk_box_pack_start(GTK_BOX (vbox), entry, FALSE, FALSE, 0);
gtk_container_add(GTK_CONTAINER (window), vbox);

g_signal_connect(GTK_OBJECT(window), "destroy",
                    G_CALLBACK(gtk_main_quit), NULL);

gtk_widget_show_all (window);
gtk_main();

return 0;
```

}
一个执行的结果如下所示：

3.35GtkArrow

GtkArrow 是个很无聊（误）很方便的组件，可以让您画出上、下、左、右方向的箭头，您可以把它放到按钮之类的组件上显示箭头，例如：

gtkarrow_demo.c

```
#include <gtk/gtk.h>

GtkWidget *create_gtk_arrow(GtkArrowType arrow_type,
                            GtkShadowType shadow_type) {
    GtkWidget *arrow;
    GtkWidget *button;

    arrow = gtk_arrow_new(arrow_type, shadow_type);
    button = gtk_button_new();
    gtk_container_add(GTK_CONTAINER(button), arrow);

    return button;
}

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *table;
    GtkWidget *button;

    gtk_init (&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW (window), "GtkArrow");
    gtk_window_set_default_size(GTK_WINDOW(window), 250, 150);
```

```
    table = gtk_table_new(3, 3, TRUE);

    gtk_table_attach_defaults(GTK_TABLE(table),
        create_gtk_arrow(GTK_ARROW_UP, GTK_SHADOW_IN), 1, 2, 0, 1);
    gtk_table_attach_defaults(GTK_TABLE(table),
        create_gtk_arrow(GTK_ARROW_LEFT, GTK_SHADOW_OUT), 0, 1, 1, 2);
    gtk_table_attach_defaults(GTK_TABLE(table),
        create_gtk_arrow(GTK_ARROW_DOWN, GTK_SHADOW_ETCHED_IN), 1, 2, 2,
3);
    gtk_table_attach_defaults(GTK_TABLE(table),
        create_gtk_arrow(GTK_ARROW_RIGHT, GTK_SHADOW_ETCHED_OUT), 2, 3, 1,
2);

    gtk_container_add(GTK_CONTAINER (window), table);

    g_signal_connect(G_OBJECT (window), "destroy",
                     G_CALLBACK (gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

上面这个程序建立四个有箭头显示的按钮，并放置到 GtkTable 之中，一个执行的结果如下所示：

3.36GtkRuler

GtkRuler 可以是一个垂直或水平外观的标尺组件，您可以设定它的上下界，也可以让它上面的光标跟随鼠标的移动，您可以使用 gtk_hruler_new()或 gtk_vruler_new()来建立水平或垂直标尺组件，使用 gtk_ruler_set_metric()可以设定度量单位，可以设定的值有 GTK_PIXELS（像素）、GTK_INCHES（英吋）、GTK_CENTIMETERS（公厘）。

您可以设定 GtkRuler 的范围：

```
void gtk_ruler_set_range(GtkRuler *ruler,
                         gdouble lower,
                         gdouble upper,
                         gdouble position,
                         gdouble max_size);
```

lower 与 upper 用来设定标尺的上下界，position 设定目前标尺上小光标的显示位置，max_size 则是用来计算标尺上可以显示刻度及数字的详细程度时使用，设定越小的数字，标尺刻度或数字会越细，设定越大的数字，标尺刻度或数字范围会越大。

若要让标尺上的光标跟随鼠标的位置而移动，首先鼠标移动范围的组件必须能接受鼠标移动事件，例如设定 GtkWindow 接受鼠标移动事件：

```
gtk_widget_set_events(window, GDK_POINTER_MOTION_MASK |
                              GDK_POINTER_MOTION_HINT_MASK);
```

而鼠标移动的 motion_notify_event 信号，必须连接到 GtkRuler 的 motion_notify_event 函式，例如：

```
#define EVENT_METHOD(i, x) GTK_WIDGET_GET_CLASS(i)->x



g_signal_connect_swapped(G_OBJECT(window), "motion_notify_event",
                         G_CALLBACK(EVENT_METHOD(hrule, motion_notify_event)),
                         hrule);
```

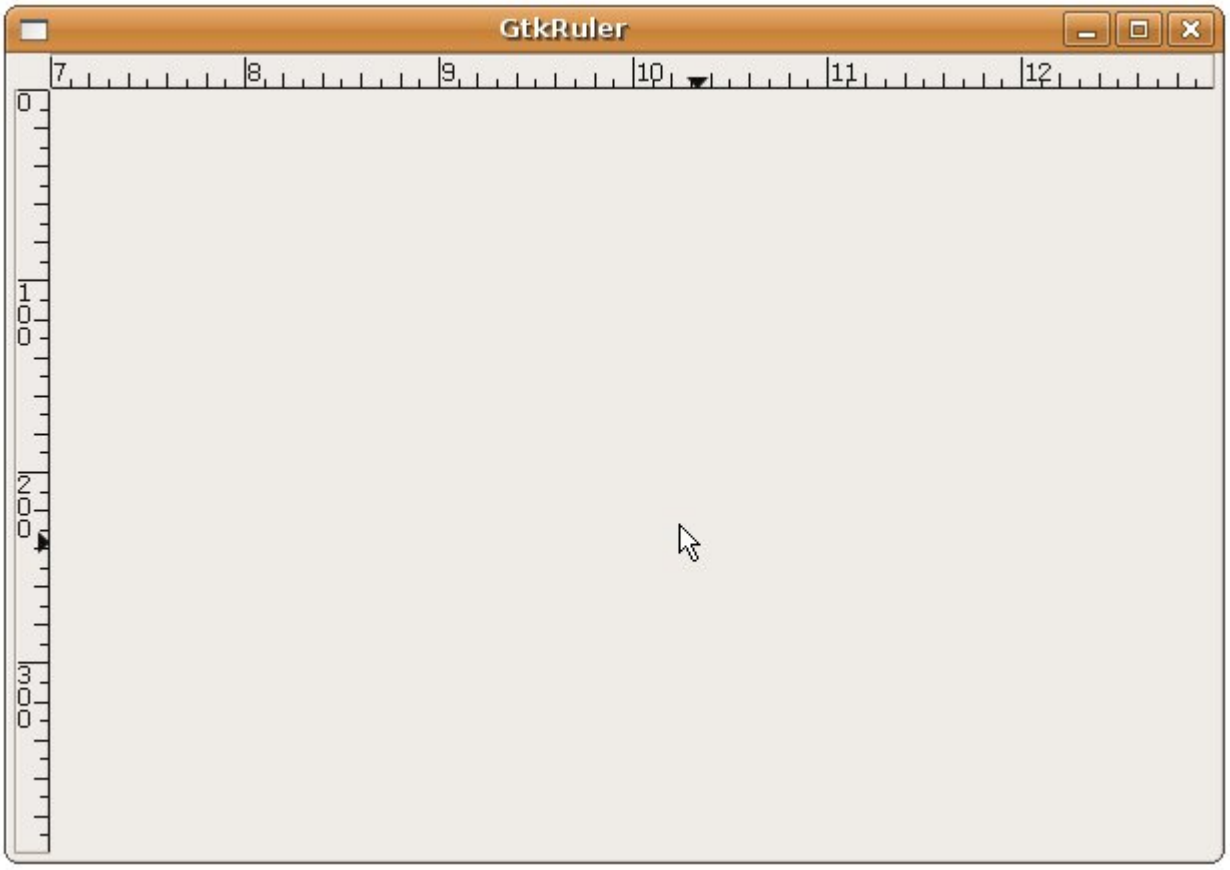
下面的范例示范如何让 GtkRuler 跟随鼠标在 GtkWindow 上的移动，GtkRuler 的位置则是利用 GtkTable 的放置：

gtkruler_demo.c

```
#include <gtk/gtk.h>

#define EVENT_METHOD(i, x) GTK_WIDGET_GET_CLASS(i)->x

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *table;
```

```
    GtkWidget *hrule;
    GtkWidget *vrule;

    gtk_init (&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW (window), "GtkRuler");
    gtk_window_set_default_size(GTK_WINDOW(window), 600, 400);

    gtk_widget_set_events(window, GDK_POINTER_MOTION_MASK |
                                  GDK_POINTER_MOTION_HINT_MASK);

    table = gtk_table_new (2, 2, FALSE);
    gtk_container_add(GTK_CONTAINER(window), table);

    hrule = gtk_hruler_new();
    gtk_ruler_set_metric(GTK_RULER(hrule), GTK_PIXELS);
    gtk_ruler_set_range(GTK_RULER(hrule), 7, 13, 0, 20);
    g_signal_connect_swapped(G_OBJECT(window), "motion_notify_event",
                             G_CALLBACK(EVENT_METHOD(hrule,
motion_notify_event)),
                             hrule);
    gtk_table_attach(GTK_TABLE(table), hrule, 1, 2, 0, 1,
                     GTK_EXPAND|GTK_SHRINK|GTK_FILL, GTK_FILL, 0, 0);

    vrule = gtk_vruler_new();
    gtk_ruler_set_metric(GTK_RULER(vrule), GTK_PIXELS);
    gtk_ruler_set_range(GTK_RULER(vrule), 0, 400, 10, 400);
    g_signal_connect_swapped(G_OBJECT(window), "motion_notify_event",
                             G_CALLBACK(EVENT_METHOD(vrule,
motion_notify_event)),
                             vrule);
    gtk_table_attach(GTK_TABLE(table), vrule, 0, 1, 1, 2,
                     GTK_FILL, GTK_EXPAND|GTK_SHRINK|GTK_FILL, 0, 0);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);
    gtk_widget_show_all(window);
    gtk_main();

    return 0;
}
```

一个执行的结果画面如下所示：



3.37GtkAssistant

在应用程序安装或是使用者注册、设定时，可以提供使用者「精灵」(Wizard) 进行一些选项设定与信息填写，在 Step by Step 的过程中，提示使用者完成所有必要的选项设定或信息填写，精灵可以使用 GtkAssistant 类别来提供这个功能。

GtkAssistant 中每一步的画面，要参考至一个 GtkWidget，所以您可以事先设计好一个 GtkWidget，当中进行组件置放、设定版面管理、设定图片、标题等，接着使用 gtk_assistant_append_page () 附加为 GtkAssistant 的一个页面，接着要使用 gtk_assistant_set_page_type()设定好页面类型：

GTK_ASSISTANT_PAGE_CONTENT：一般内容页面
GTK_ASSISTANT_PAGE_INTRO：简介页面，通常是精灵的开始
GTK_ASSISTANT_PAGE_CONFIRM：确认页面，通常是精灵的结束
GTK_ASSISTANT_PAGE_SUMMARY：显示使用者的变更信息
GTK_ASSISTANT_PAGE_PROGRESS：进度页面，通常是精灵中的某个步骤

每个页面预设的「下一步」(例如 GTK_ASSISTANT_PAGE_INTRO 或 GTK_ASSISTANT_PAGE_PROGRESS 等) 或「套用」(例如 GTK_ASSISTANT_PAGE_CONFIRM) 预设是无法作用的，您要使用 gtk_assistant_set_page_complete()并设定 complete 参数为 TRUE，「下一步」或「套用」按钮才会有作用。

下面这个程序是个简单的示范，程序中将用 GtkAssistant 建立精灵：

gtkassistant_demo.c

```
#include <gtk/gtk.h>

GtkWidget* gtk_assistant_page_new(GtkWidget *assistant,
                                  GtkWidget *widget,
                                  gchar *title,
                                  gchar *image,
                                  GtkAssistantPageType type) {
    gtk_assistant_append_page(GTK_ASSISTANT(assistant), widget);
    gtk_assistant_set_page_title(GTK_ASSISTANT(assistant), widget, title);
    gtk_assistant_set_page_side_image(GTK_ASSISTANT(assistant),
            widget, gdk_pixbuf_new_from_file(image, NULL));
    gtk_assistant_set_page_type(GTK_ASSISTANT(assistant), widget, type);
    gtk_assistant_set_page_complete(GTK_ASSISTANT(assistant), widget, TRUE);

    return assistant;
}

int main(int argc, char *argv[]) {
    GtkWidget *assistant;
    GtkWidget *label;

    gtk_init (&argc, &argv);

    assistant = gtk_assistant_new();
    gtk_window_set_title(GTK_WINDOW(assistant), "GtkAssistant");
    gtk_window_set_default_size(GTK_WINDOW(assistant), 300, 180);

    gtk_assistant_page_new(assistant, gtk_label_new("简介"),
        "精灵开始", "caterpillar.jpg", GTK_ASSISTANT_PAGE_INTRO);
    gtk_assistant_page_new(assistant, gtk_label_new("说明内容或组件"),
        "精灵第二页", "caterpillar.jpg", GTK_ASSISTANT_PAGE_PROGRESS);
    gtk_assistant_page_new(assistant, gtk_label_new("说明内容或组件"),
        "精灵第三页", "caterpillar.jpg", GTK_ASSISTANT_PAGE_PROGRESS);
    gtk_assistant_page_new(assistant, gtk_label_new("说明内容或组件"),
        "精灵结束", "caterpillar.jpg", GTK_ASSISTANT_PAGE_CONFIRM);

    g_signal_connect(GTK_OBJECT(assistant), "cancel",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(assistant);
```

```
    gtk_main();

    return 0;
}
```

一个执行的结果如下所示：

3.38GtkCalendar

GtkCalendar 是个可以显示月历的组件，只要使用 gtk_window_new()建立组件，就可以拥有一个完整的日历组件显示，若要设定日期，则可以使用 gtk_calendar_select_month()设定年及月份，使用 gtk_calendar_select_day()设定日。

使用 gtk_calendar_select_month()设定时要注意的是，月份可设定的数值是从 0 到 11，0 表示 1 月，11 表示 12 月，而使用 gtk_calendar_select_day()设定值则为 1 到 31，或是设定为 0 表示不选取日。

您可以使用 gtk_calendar_mark_day()、gtk_calendar_unmark_day()或 gtk_calendar_clear_marks()设定日期标记，若要得知被标记的日期信息，可以透过 GtkCalendar 结构的成员 num_marked_dates 得知有几天被标记了，marked_date 为一个数组，可用以得知哪一天被标记了，例如：

```
if (calendar->marked_date[26-1]) {
    // 日期 26 被标记了
}
```

要注意的是，数组索引值是从 0 开始，所以存取 marked_date 时，日期实际上要减去 1，才会是对应的索引。

另外，Calendar 成员中的 month、year 与 selected_day 分别表示目前看到的月、年及所选中的日，若要取得选中的年、月、日，则可以使用 gtk_calendar_get_date()函式，您必须提供三个变量的地址给它，执行过后，三个变量中就会储存对应的年、月、日：

```
void gtk_calendar_get_date(GtkCalendar *calendar,
                           guint *year,
                           guint *month,
                           guint *day);
```

下面这个程序是个简单的示范：

gtkcalendar_demo.c

```
#include <gtk/gtk.h>

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *calendar;
    guint year = 1975;
    guint month = 5 - 1; // 5 月
    guint day = 26;
    guint mark_day = 19;

    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GtkCalendar");

    calendar = gtk_calendar_new();
    gtk_calendar_select_month(GTK_CALENDAR(calendar), month, year);
    gtk_calendar_select_day(GTK_CALENDAR(calendar), day);
    gtk_calendar_mark_day(GTK_CALENDAR(calendar), mark_day);

    gtk_container_add(GTK_CONTAINER(window), calendar);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

一个执行结果如下所示：

3.39GtkDrawingArea

GtkDrawingArea 是一个用来进行绘图的组件，绘图的时候，是将之绘制在 window 成员上，在绘图时要处理的事件有：

realize：组件被初始时
configure_event：组件尺寸改变时
expose_event：组件需要重绘时

要在 GtkDrawingArea 上绘图，完整的函式内容可以参考 Drawing Primitives，基本上每个绘图函式都会有一个 GdkGC 自变量，它主要包括了前景色、背景色、线宽等信息，您从 GtkDrawingArea 的 GtkStyle 中可以取得对应的 GdkGC 信息。

下面这个程序是个简单的示范：

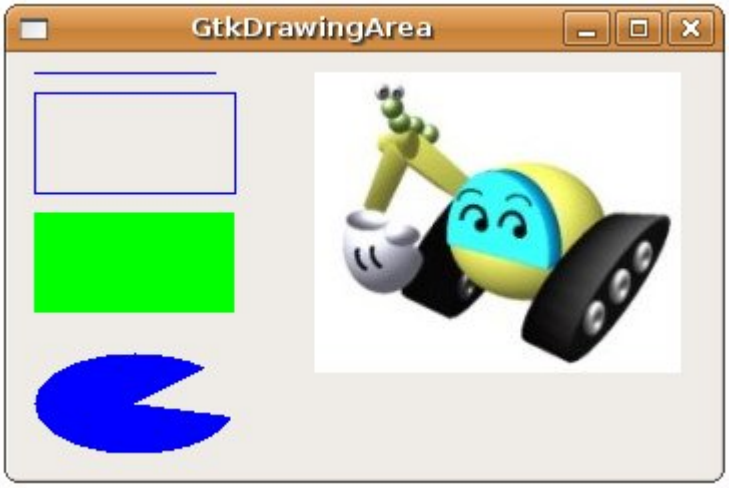
gtkdrawingarea_demo.c

```
#include <gtk/gtk.h>

gboolean expose_event_callback(GtkWidget *widget,
                               GdkEventExpose *event,
                               gpointer data) {

    GdkGC *gc = widget->style->fg_gc[GTK_WIDGET_STATE(widget)];
    GdkDrawable *drawable = widget->window;
    GdkColor color;

    // 画一条线
    gdk_draw_line(drawable, gc, 10, 10, 100, 10);
    // 画一个空心矩形
    gdk_draw_rectangle(drawable, gc, FALSE, 10, 20, 100, 50);

    color.green = 65535;
```

```
        color.blue = 0;
        gdk_gc_set_rgb_fg_color(gc, &color);

        // 画一个实心矩形
        gdk_draw_rectangle(drawable, gc, TRUE, 10, 80, 100, 50);

        color.green = 0;
        color.blue = 65535;
        gdk_gc_set_rgb_fg_color(gc, &color);

        // 画一个扇形
        gdk_draw_arc(drawable, gc, TRUE,
                        10, 150, 100, 50, 45 * 64, 300 * 64);

        // 画一张图
        gdk_draw_pixbuf(drawable, gc, gdk_pixbuf_new_from_file("caterpillar.jpg", NULL)
                , 0, 0, 150, 10, -1, -1,
                GDK_RGB_DITHER_NORMAL, 0, 0);

        return TRUE;
}

int main(int argc, char *argv[]) {
        GtkWidget *window;
        GtkWidget *drawing_area;

        gtk_init(&argc, &argv);
        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        gtk_window_set_title(GTK_WINDOW(window), "GtkDrawingArea");

        drawing_area = gtk_drawing_area_new();
        gtk_widget_set_size_request(drawing_area, 350, 200);

        g_signal_connect(GTK_OBJECT(drawing_area), "expose_event",
                                G_CALLBACK(expose_event_callback), NULL);

        gtk_container_add(GTK_CONTAINER(window), drawing_area);

        g_signal_connect(GTK_OBJECT(window), "destroy",
                                G_CALLBACK(gtk_main_quit), NULL);

        gtk_widget_show_all(window);

        gtk_main();
```

```
    return 0;
}
```

执行结果如下：



第四章 GLib

4.1 基本型态、宏、公用（Utility）函式

4.11GLib 基本型态与宏

为了易用与可移植性，GLib 定义了一些基本数据型态，例如在 C 语言中，并没有定义布尔数型态，而在 GLib 中则定义了 gboolean 这个型态，可以设定 TRUE 或 FALSE 值，这些型态基本上是使用 typedef 来定义：

```
typedef gint    gboolean;
```

GLib 定义的基本数据型态可概略分为四大类：

对应 C 的整数型态：gchar、gint、gshort、glong、gfloat、 gdouble。
对应 C 但更易于使用的型态：gpointer、gconstpointer、 guchar、guint、gushort、gulong。
不是标准 C 的新型态：gboolean、gsize、gssize。
保证在各平台长度相同的型态：gint8、guint8、gint16、 guint16、gint32、guint32、gint64、guint64.。

您可以参考 Basic Types 了解每个型态的定义方式。

在 GTK 中，处处可见宏，这些宏可以让您在写程序时更为方便，像是最基本的 TRUE、FALSE，一些方便的宏函式如 ABS（取绝对值）、MAX （取最大值）、MIN（取最小值）等，您可以参考 Standard Macros 了解一些常用宏函式的定义。

4.12GTimer

GTimer 是个定时器，当您需要量测两个执行时间点的间隔时就可以使用，例如程序执行的开始与结束时间，您可以使用 g_timer_new()建立一个新的 GTimer，若不再需要时则使用 g_timer_destroy()加以毁弃。

在 g_timer_new()之后，会自动标示启动时间，您也可以使用 g_timer_start()再度标示启动时间，并于 g_timer_elapsed()被呼叫时，传回自启动后的时间。

若使用 g_timer_start()标示启动时间，并使用 g_timer_end()标示结束时间，则于 g_timer_elapsed()被呼叫时，将传回启动后时间与结束时间的间隔，在使用 g_timer_end()标示结束时间之后，您可以使用 g_timer_continue()重新继续 GTimer 的计时。

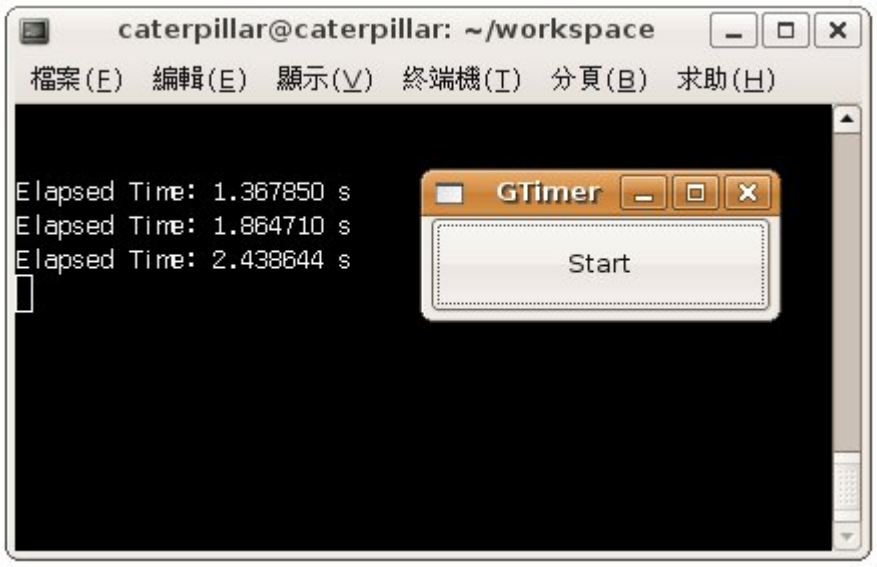
下面这个程序是个简单的示范，可以计算两次按钮之间的时间间隔：

g_timer_demo.c

```
#include <gtk/gtk.h>

void button_pressed(GtkButton *button, GTimer *timer) {
    static gdouble elapsedTime;
    static gboolean isRunning = FALSE;

    if(isRunning) {
        elapsedTime = g_timer_elapsed(timer, NULL);
        gtk_button_set_label(button, "Start");
        g_print("Elapsed Time: %f s\n", elapsedTime);
    }
    else {
        g_timer_start(timer);
        gtk_button_set_label(button, "Stop");
    }

    isRunning = !isRunning;
}

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *button;
    GTimer *timer;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "GTimer");
```

```
    gtk_window_set_default_size(GTK_WINDOW(window), 150, 50);

    button = gtk_button_new_with_label("Start");

    gtk_container_add(GTK_CONTAINER(window), button);

    timer = g_timer_new();

    g_signal_connect(GTK_OBJECT(button), "clicked",
                     G_CALLBACK(button_pressed), timer);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
}
```

一个执行的结果画面如下所示：



4.13Timeout 与 Idle

如果您要定时让程序去作某件事，则可以使用 g_timeout_add()或 g_timeout_add_full()，g_timeout_add()的定义如下：

```
guint g_timeout_add(guint interval,
                    GSourceFunc function,
                    gpointer data);
```

第一个参数是时间间隔，以毫秒为单位，第二个参数是时间到的回呼函式，第三个参数是传给回呼函式的资料，以 内建 Signal 的发射与停止 中的范例来说，可以使用 g_timeout_add()改写如下而执行结果相同：

g_timeout_demo.c

```
#include <gtk/gtk.h>

gboolean timeout_callback(GtkButton *button) {
    static gint count = 0;
    if(count < 5) {
        g_signal_emit_by_name(button, "clicked");
        count++;
        return TRUE;
    }
    else {
        return FALSE;
    }
}

// 自订 Callback 函式
void button_clicked(GtkWidget *button, gpointer data) {
    g_print("按钮按下：%s\n", (char *) data);
}

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *button;

    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "哈啰！GTK+！");

    button = gtk_button_new_with_label("按我");
    gtk_container_add(GTK_CONTAINER(window), button);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);
    g_signal_connect(GTK_OBJECT(button), "clicked",
                     G_CALLBACK(button_clicked), "哈啰！按钮！");

    g_timeout_add(1000, (GSourceFunc) timeout_callback, button);
```

```
    gtk_widget_show(window);
    gtk_widget_show(button);


    gtk_main();

    return 0;
}
```

在回呼函式中，若传回 TRUE 则继续下一次计时，定时器的下一次计时，会是在回呼函式执行完毕后开始，传回 FALSE 则定时器结束并自动销毁，若您使用 g_timeout_add_full()：

```
guint g_timeout_add_full(gint priority,
                         guint interval,
                         GSourceFunc function,
                         gpointer data,
                         GDestroyNotify notify);
```

第一个参数为时间到时的执行优先权，可以设定的优先权如下：

```
G_PRIORITY_HIGH
G_PRIORITY_DEFAULT（预设）
G_PRIORITY_HIGH_IDLE
G_PRIORITY_DEFAULT_IDLE
G_PRIORITY_LOW
```

最后一个参数则是定时器被移除时要执行的函式。

相对于计时重复执行某个动作，您可以使用 g_idle_add()或 g_idle_add_full()函式，让程序在没有什么事情作的时候（例如没有任何使用者操作，没有任何需要运算的程序代码时），也可以作一些事情，若使用 g_idle_add()：

```
guint g_idle_add(GSourceFunc function,
                 gpointer data);
```

第一个参数是回呼函式，第二个参数是传递给回呼函式的数据，例如下面这个范例，在使用者不作任何事时，就会执行指定的 idle 函式，而按下按钮时就执行按钮的回呼函式：

g_idle_demo.c

```
#include <gtk/gtk.h>

gboolean idle_callback(gpointer data) {
    g_print("%s。。XD\n", data);
```

```
	return TRUE;
}

void button_pressed(GtkButton *button, gpointer data) {
	int i;
	for(i = 0; i < 10; i++) {
		g_print("%s...\n", data);
		sleep(1);
	}
}

int main(int argc, char *argv[]) {
	GtkWidget *window;
	GtkWidget *button;

	gtk_init(&argc, &argv);

	window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
	gtk_window_set_title(GTK_WINDOW(window), "GTimer");
	gtk_window_set_default_size(GTK_WINDOW(window), 150, 50);

	button = gtk_button_new_with_label("按我");

	gtk_container_add(GTK_CONTAINER(window), button);

	g_signal_connect(GTK_OBJECT(button), "clicked",
				G_CALLBACK(button_pressed), "do something");

	g_signal_connect(GTK_OBJECT(window), "destroy",
				G_CALLBACK(gtk_main_quit), NULL);

	g_idle_add((GSourceFunc) idle_callback, "无事可作");

	gtk_widget_show_all(window);

	gtk_main();

	return 0;
}
```

同样的，指定的 idle 函式若传回 FALSE 则会移除 idle 功能，若是使用 g_idle_add_full()：

```
guint g_idle_add_full(gint priority,
                      GSourceFunc function,
                      gpointer data,
                      GDestroyNotify notify);
```

其上的参数与 g_timeout_add_full()类似。

4.14 环境信息

要撰写一个可以跨平台的程序（无论是要重新编译或基于虚拟机器实时执行），与环境相关的信息如何设定与取得是必须解决的，例如使用者家（home）路径、暂存路径、主机信息等，这些相关信息，可以使用 GLib 的 Miscellaneous Utility Functions 中所提供的函式来取得。

以下的范例程序示范了几个环境信息的取得，以及环境变量的取得与设定：

environment_info_demo.c

```
#include <glib.h>

int main(int argc, char *argv[]) {
    printf("Host name\t: %s\n", g_get_host_name());
        // 取得使用者真实姓名
    printf("Real name\t: %s\n", g_get_real_name());
        // 取得使用者账号名称
    printf("User name\t: %s\n", g_get_user_name());
        // 取得目前所在路径
    printf("Current dir\t: %s\n", g_get_current_dir());
        // 取得使用者家目录
    printf("Home dir\t: %s\n", g_get_home_dir());
        // 取得暂存目录
    printf("Temp dir\t: %s\n", g_get_tmp_dir());
    // 取得 PATH 环境变量
    printf("PATH\t\t: %s\n", g_getenv("PATH"));
        // 设定 CLASSPATH 环境变量，FALSE 表示若已设定则不覆写
    g_setenv("CLASSPATH", "D:\\Temp", FALSE);
        // 取得 CLASSPATH 环境变量
    printf("CLASSPATH\t: %s\n", g_getenv("CLASSPATH"));
        // 取消 CLASSPATH 环境变量
    g_unsetenv("CLASSPATH");
    printf("CLASSPATH\t: %s\n", g_getenv("CLASSPATH"));

    return 0;
}
```

一个执行结果如下所示：

```
Host name       : CATERPILLAR-PC
Real name       : caterpillar
User name       : caterpillar
Current dir     : D:\Temp
Home dir        : C:\Users\caterpillar
```

```
Temp dir        : C:\Users\CATERP~1\AppData\Local\Temp
PATH            : C:\Windows\system32;C:\Windows;C:\Windows\System32\Wbem;C:\Pro
gram Files\ASUS Security Center\ASUS Security Protect Manager\bin;C:\Program Fil
es\Java\jdk1.6.0_07\bin;C:\Winware\MinGW\bin;C:\Winware\Qt\4.3.3\bin;C:\Winware\
Gtk\bin
CLASSPATH       : D:\Temp
CLASSPATH       : (null)
```

4.15 日志（Logging）

程序中不免会出现错误，当错误发生时，您可以使用 printf()或是 g_print()在主控台（Console）显示讯息给使用者，如果是在窗口程序中，可能是使用消息框，您也可能想针对某个层级的讯息作个别处理，例如储存在 log 档案之中，在 GLib 中，您可以使用 Message Logging 中所介绍的函式来进行日志功能。

要进行日志，首先最基本的就是使用 g_log()函式：

```
void g_log(const gchar *log_domain,
           GLogLevelFlags log_level,
           const gchar *format,
           ...);
```

第一个参数是 log_domain，用来区别日志讯息的发出者，若有设定日志的处理函式，则 log_domain 亦会传递给处理函式，如果您没有指定，则预设会使用 G_LOG_DOMAIN，函式库会定义 G_LOG_DOMAIN，以区别于其它的函式库，例如 GTK 在它的 Makefine 中定义为"Gtk"：

```
INCLUDES = -DG_LOG_DOMAIN=\"Gtk\"
```

第二个参数是日志层级，可以设定为以下的值：

G_LOG_LEVEL_ERROR（致命的，FATAL）
G_LOG_LEVEL_CRITICAL
G_LOG_LEVEL_WARNING
G_LOG_LEVEL_MESSAGE
G_LOG_LEVEL_INFO
G_LOG_LEVEL_DEBUG

另外还有两个 G_LOG_FLAG_FATAL 与 G_LOG_FLAG_RECURSION，作为内部的旗标使用，其中与 G_LOG_FLAG_FATAL 相关联的，例如 G_LOG_LEVEL_ERROR，是属于严重的致命讯息，当日志时以这个层级输出时，应用程序会被中止并呼叫核心倾印（dump）。

第三个参数是要输出的讯息，其它则是额外的讯息。

GLib 还提供了五个宏函式，方便使用日志与相对应的讯息层级：

```
#define g_message(...)
#define g_warning(...)
#define g_critical(...)
#define g_error(...)
#define g_debug(...)
```

先前说过，G_LOG_FLAG_FATAL 是内部旗标，预设是 G_LOG_LEVEL_ERROR 与之关联，如果您想让其它层级的讯息也成为 FATAL 的，则可以使用 g_log_set_always_fatal()函式，例如将 DEBUG 与 CRITICAL 设定为 FATAL：

```
g_log_set_always_fatal(G_LOG_LEVEL_DEBUG | G_LOG_LEVEL_CRITICAL);
```

对于日志讯息，您可以设定相对应的处理函式，这是使用 g_log_set_handler()函式来达成：

```
guint g_log_set_handler(const gchar *log_domain,
                        GLogLevelFlags log_levels,
                        GLogFunc log_func,
                        gpointer user_data);
```

传回的整数值为 Handler Id，其中 GLogFunc 为回呼函式，它的宣告定义如下：

```
void (*GLogFunc) (const gchar *log_domain,
                  GLogLevelFlags log_level,
                  const gchar *message,
                  gpointer user_data);
```

设定讯息处理函式之后，若想移除，则可以使用 g_log_remove_handler()函式，根据 Handler ID 及 log domain 来移除：

```
void g_log_remove_handler(const gchar *log_domain,
                          guint handler_id);
```

4.2 输入输出

4.21 基本档案读写

若要进行档案的基本读写，您可以使用 g_file_get_contents()、g_file_set_contents()

函式，这两个函式会对档案作整个读取与整个写入的动作。

以下直接以程序作为示范，您可以从命令列自变量设定档案来源与目的地，将档案读入，显示在屏幕上并写入另一个档案：

file_demo.c

```
#include <glib.h>

handle_error(GError *error) {
	if(error != NULL) {
		g_printf(error->message);
		g_clear_error(&error);
	}
}

int main(int argc, char *argv[]) {
	gchar *src, *dest;
	gchar *content;
	gsize length;
	GError *error = NULL;

	if(argc >= 3) {
		src = argv[1];
		dest = argv[2];
	}
	else {
		return 1;
	}

	if (!g_file_test(src, G_FILE_TEST_EXISTS)) {
		g_error("Error: File does not exist!");
	}

	g_file_get_contents(src, &content, &length, &error);
	handle_error(error);
	g_print("%s\n", content);

	g_file_set_contents(dest, content, -1, &error);
	handle_error(error);

	g_free (content);

	return 0;
}
```

g_file_test()函式可以测试档案的几个状态：

G_FILE_TEST_IS_REGULAR：不是符号连结文件或目录
G_FILE_TEST_IS_SYMLINK：符号连结文件
G_FILE_TEST_IS_DIR：目录
G_FILE_TEST_IS_EXECUTABLE：可执行档
G_FILE_TEST_EXISTS：档案是否存在

g_file_get_contents()可以指定档案，将档案读入 content 中，并将长度读入 length，如果读取成功会传回 TRUE，失败会传回 FALSE，错误相关信息会设定给 GError，而 g_file_set_contents()的使用类似，长度设定为-1 表示写入整个字符串。

在 GtkTextView 中曾写过一个与 GTK 图形组件结合的档案读取程序。

4.22 目录信息

如果您要取得目录的信息，可以使用 GDir 及其相关的函式，您可以使用 g_dir_open()函式指定一个目录路径，这会传回 GDir 对象，接着您可以使用 g_dir_read_name()读取目录下的文件名称。

直接来看个范例，下面这个程序示范如何从命令列自变量输入要查询的目录，并显示该目录下的内容：

dir_demo.c

```
#include <glib.h>

void listDir(const gchar *parent, int hier) {
    const gchar *file, *fullPath;
    GDir *dir;
    int i;

    dir = g_dir_open(parent, 0, NULL);

    while((file = g_dir_read_name(dir))) {
        for(i = 0; i < hier; i++) {
            g_print("    ");
        }

        fullPath = g_build_filename(parent, file, NULL);
        if(g_file_test(fullPath, G_FILE_TEST_IS_DIR)) {
            g_print("[%s]\n", file);
            listDir(fullPath, hier + 1);
        }
        else {
```

```
                g_print("%s\n", file);
            }
        }

        g_dir_close(dir);
}

int main(int argc, char *argv[]) {
        listDir(argv[1], 0);
        return 0;
}
```

在程序中，g_build_filename()可以协助建立档案路径名称，您可以指定目录或文件名称，最后以一个 NULL 作为结尾，g_build_filename()会自行依操作系统，使用适当的目录分隔符（Linux 下的/或 Windows 下的\），建立完整的档案或目录路径。

一个执行的范例如下所示：

```
$ dir_demo ~/library
[hibernate-3.2]
    build.bat
    build.sh
    build.xml
    changelog.txt
    [doc]
        [api]
            allclasses-frame.html
            allclasses-noframe.html
            constant-values.html
            deprecated-list.html
            help-doc.html
            index-all.html
            index.html
            jdstyle.css
            [org]
                [hibernate]
                    [action]
                ..略
```

另外还有几个简便的档案操作函式，像是g_rename()可更改文件名称，g_remove()可移除档案，g_rmdir()可删除目录，g_mkdir()可建立目录等，这些都可以在 **File Utilities** 中查询到使用方式。

4.23GIOChannel 与 档案处理

在 基本档案读写 中使用 g_file_get_contents()、g_file_set_contents()函式，会对档

案作整个读取与整个写入的动作，若您想要对档案作一些逐字符、逐行读取、附加等操作，则可以使用 GIOChannel。

下面这个程序改写 基本档案读写 中的范例，使用 GIOChannel 来进行档案读写的动作：

g_io_channel_demo.c

```
#include <glib.h>

handle_error(GError *error) {
	if(error != NULL) {
		g_printf(error->message);
		g_clear_error(&error);
	}
}

int main(int argc, char *argv[]) {
	gchar *src, *dest;
	GIOChannel *ioChannel;
	gchar *content;
	gsize length;
	GError *error = NULL;

	if(argc >= 3) {
		src = argv[1];
		dest = argv[2];
	}
	else {
		return 1;
	}

	if (!g_file_test(src, G_FILE_TEST_EXISTS)) {
		g_error("Error: File does not exist!");
	}

	ioChannel = g_io_channel_new_file(src, "r", &error);
	handle_error(error);
	g_io_channel_read_to_end(ioChannel, &content, &length, NULL);
	g_io_channel_close(ioChannel);

	ioChannel = g_io_channel_new_file(dest, "w", &error);
	handle_error(error);
	g_io_channel_write_chars(ioChannel, content, -1, &length, NULL);
	g_io_channel_close(ioChannel);
```

```
    g_free(content);

    return 0;
}
```

您使用的是 g_io_channel_new_file()函式来建立 GIOChannel，建立时可以使用"r"、"w"、"a"、"r+"、"w+"、"a+"等档案模式，其作用与使用 fopen() 时的模式相同。

程序中使用的是 g_io_channel_read_to_end()函式，一次读取所有的档案内容，您也可以使用 g_io_channel_read_chars() 、 g_io_channel_read_line() 、g_io_channel_read_line_string()等函式，来对档案作不同的读取动作。

4.24GIOChannel 与 Pipe

在 Linux 系统中，想要在两个处理程序之间传送数据，必须使用 pipe，您可以使用 pipe()函式来开启 pipe，您要传入两个 File Descriptor：

```
gint parent_to_child[2];
if(pipe(parent_to_child) == -1) {
    g_error("错误: %s\n", g_strerror(errno));
    return 1;
}
```

开启 pipe 之后，任何写入 parent_to_child[1]的数据，可以从 parent_to_child[1]读得。

在 GIOChannel 与 档案处理 中，看过如何使用 GIOChannel 来处理档案，在 Linux 中很多对象或数据都被视作档案，所以您也可以利用 GIOChannel 来处理 pipe 的数据，您可以加入 watch，监看 GIOChannel 中的数据，当有数据进行读写时会发出事件，您可利用 callback 函式予以处理。

您可以使用 g_io_channel_unix_new()函式从 pipe 的 File Descriptor 中建立 GIOChannel，例如，假设 input[0]是 pipe 中写出数据的 File Descriptior，可以如下建立 GIOChannel：

```
GIOChannel *channel_read = g_io_channel_unix_new(input[0]);
if(channel_read == NULL) {
    g_error("错误: 无法建立 GIOChannels! \n");
}
```

若要对 GIOChannel 进行监看，可以使用 g_io_add_watch()函式，例如：

```
if(!g_io_add_watch(channel_read, G_IO_IN | G_IO_HUP,
                   (GIOFunc) iochannel_read, (gpointer) data)) {
    g_error("错误: 无法对 GIOChannel 进行监看\n");
```

```
}
```

第二个参数是监看的条件：

G_IO_IN：有待读取数据
G_IO_OUT：可写入资料
G_IO_PRI：有待读取的紧急数据
G_IO_ERR：发生错误
G_IO_HUP：连接挂断
G_IO_NVAL：无效请求，File Descriptor 没有开启

第三个参数是监看条件发生时的 callback 函式，第四个参数是传递给 callback 的数据。

您可以使用 g_io_channel_write_chars()函式写入字符至 GIOChannel 中，例如：

```
GIOStatus ret_value = g_io_channel_write_chars(
                        channel, text->str, -1, &length, NULL);
if(ret_value == G_IO_STATUS_ERROR) {
    g_error("错误: 无法写入 pipe! \n");
}
else {
    g_io_channel_flush(channel, NULL);
}
```

可使用 g_io_channel_read_line()从 GIOChannel 中读入数据：

```
ret_value = g_io_channel_read_line(channel, &message, &length, NULL, NULL);

if(ret_value == G_IO_STATUS_ERROR) {
    g_error("错误: 无法读取! \n");
}
```

以下这个程序是个结合 GIOChannel 与 Pipe 的范例，程序会有 fork 一个子程序，父程序透过 Pipe 将数据传给子程序，并利用 GIOChannel 加入 watch，当父程序写入数据时，子程序执行 callback 函式以作对应的数据处理，执行结果中，程序会有两个窗口，分别属于父子程序，父程序窗口的 GtkSpinButton 拉动时，子程序窗口会显示对应数字：

giochannel_pipe_demo.c

```
#include<gtk/gtk.h>
#include<errno.h>
#include<unistd.h>
```

```
void value_changed_callback(GtkSpinButton *spinButton, GIOChannel *channel);
gboolean iochannel_read(GIOChannel *channel, GIOCondition condition, GtkLabel *label);
void setup_parent(gint output[]);
void setup_child(gint input[]);

int main(int argc, char *argv[]) {
	gint parent_to_child[2];

	if(pipe(parent_to_child) == -1) { //  开启 Pipe
		g_error("Error: %s\n", g_strerror(errno));
		return 1;
	}

	// fork  子程序
	switch(fork()) {
		case -1:
			g_error("错误: %s\n", g_strerror(errno));
			break;
		case 0: //  这是子程序
			gtk_init(&argc, &argv);
			setup_child(parent_to_child);
			break;
		default: //  这是父程序
			gtk_init(&argc, &argv);
			setup_parent(parent_to_child);
	}

	gtk_main();

	return 0;
}

// GtkSpinButton 的 callback
void value_changed_callback(GtkSpinButton *spinButton, GIOChannel *channel) {
	GIOStatus ret_value;
	gint value;
	GString *text;
	gsize length;

	value = gtk_spin_button_get_value_as_int(spinButton);
	text = g_string_new("");
	g_string_sprintf(text, "%d\n", value);
```

```
    // 写入资料至 GIOChannel
    ret_value = g_io_channel_write_chars(channel, text->str, -1, &length, NULL);
    if(ret_value == G_IO_STATUS_ERROR) {
        g_error("错误: 无法写入 GIOChannel ！\n");
    }
    else {
        g_io_channel_flush(channel, NULL);
    }
}

gboolean iochannel_read(GIOChannel *channel,
                        GIOCondition condition, GtkLabel *label) {
    GIOStatus ret_value;
    gchar *message;
    gsize length;

    if(condition & G_IO_HUP) {
        g_error("错误: Pipe 已中断！\n");
    }

    ret_value = g_io_channel_read_line(channel, &message, &length, NULL, NULL);

    if(ret_value == G_IO_STATUS_ERROR) {
        g_error("错误: 无法读取！\n");
    }

    message[length-1] = 0;
    gtk_label_set_text(label, message);

    return TRUE;
}

void setup_parent(gint output[]) {
    GtkWidget *window;
    GtkWidget *spinButton;
    GIOChannel *channel_write;

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "Parent");
    gtk_window_set_default_size(GTK_WINDOW(window), 250, 50);

    spinButton = gtk_spin_button_new_with_range(0.0, 100.0, 1.0);
    gtk_container_add(GTK_CONTAINER(window), spinButton);
```

```
    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    // 关闭不用的 Pipe 输出
    close(output[0]);

    // 建立 GIOChannel
    channel_write = g_io_channel_unix_new(output[1]);
    if(channel_write == NULL) {
        g_error("错误: 无法建立 GIOChannels！\n");
    }

    g_signal_connect(G_OBJECT(spinButton), "value_changed",
                     G_CALLBACK(value_changed_callback), (gpointer) channel_write);

    gtk_widget_show_all(window);
}

void setup_child(gint input[]) {
    GtkWidget *window;
    GtkWidget *label;
    GIOChannel *channel_read;

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "Child");
    gtk_window_set_default_size(GTK_WINDOW(window), 250, 50);

    label = gtk_label_new("0");
    gtk_container_add(GTK_CONTAINER(window), label);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);

    // 关闭不必要的 Pipe 输入
    close(input[1]);

    channel_read = g_io_channel_unix_new(input[0]);
    if(channel_read == NULL) {
        g_error("错误: 无法建立 GIOChannels！\n");
    }

    if(!g_io_add_watch(channel_read, G_IO_IN | G_IO_HUP,
        (GIOFunc) iochannel_read, (gpointer) label)) {
        g_error("错误: 无法对 GIOChannel 加入 watch！\n");
```

```
    }

    gtk_widget_show_all(window);
}
```

执行的一个结果画面如下所示：

4.3 数据结构、内存配置

4.31GString

GString 是 GLib 所提供的对字符串处理的型态，GString 保有字符串的长度信息，当您对 GString 进行插入、附加时，GString 会自动调整长度，您也可以搭配一些 GLib 的函式来方便的处理字符串。

GString 的定义如下：

```
typedef struct {
  gchar *str;
  gsize len;
  gsize allocated_len;
} GString;
```

str 为 null 结尾的 C 字符串之参考，len 为目前字符串不包括 null 结尾的长度，allocated_len 为 GString 所配置的缓冲区长度，如果字符串长度超出这个长度会自动重新配置。

您有三种方式可以建立 GString：

```
GString* g_string_new(const gchar *init);
GString* g_string_new_len(const gchar *init, gssize len);
GString* g_string_sized_new(gsize dfl_size);
```

第一个函式依所给的 init 字符串来建立适当 len 的 GString，并保留适当的 allocated_len，建立的时候是将 init 字符复制至 GString 中。第二个函式则是指定 len 来建立 GString，因为是自行指定，所以 len 必须超过 init 的长度。第三个函式则是指定 allocated_len 来

建立 GString。

您可以从下面的范例程序看出三个函式的作用：

gstring_demo.c

```
#include <glib.h>

int main(int argc, char *argv[]) {
	GString *string = g_string_new("Justin");
	g_print("len = %d, allocated_len = %d\n",
			string->len, string->allocated_len);
	g_string_free(string, FALSE);

	string = g_string_new_len("Justin", 32);
	g_print("len = %d, allocated_len = %d\n",
			string->len, string->allocated_len);
	g_string_free(string, FALSE);

	string = g_string_sized_new(32);
	g_print("len = %d, allocated_len = %d\n",
			string->len, string->allocated_len);
	g_string_free(string, FALSE);

	return 0;
}
```

若不使用 GString 时，可以使用 g_string_free()释放，其第二个参数若为 TRUE，则会连同 C 的字符串一同释放。

执行的结果如下所示：

```
len = 6, allocated_len = 16
len = 32, allocated_len = 64
len = 0, allocated_len = 64
```

字符串的串接可以使用 g_string_append()等函式，例如：

```
GString *string = g_string_new("哈啰！");
g_string_append(string, "GTK 程序设计！");
g_print("%s\n", string->str);
```

这一段程序代码会在主控台上显示 "哈啰！GTK 程序设计！"（以 UTF8 撰写程序的话可以显示中文），若想要在前端附加则使用 g_string_prepend()等函式，若想要中间插入字符则使用 g_string_insert()等函式。

除了单纯的附加、插入字符等函式之外，以下还有几个常用的操作字符串的函式：

g_string_equal()	判断两个 GString 的字符内容是否相同
g_string_ascii_up()或 g_utf8_strup()	转换 GString 中的字符为小写
g_string_ascii_down()或 g_utf8_strdown()	转换 GString 中的字符为大写
g_string_printf()	如 printf()一样的方式，在 GString 中格式化字符串

一个简单的范例如下所示:
gstring_demo.c

```
#include <glib.h>

int main(int argc, char *argv[]) {
	GString *string1, *string2;
	gboolean is_eq;

	string1 = g_string_sized_new(16);
	g_string_printf(string1, "This is %s speaking!", "caterpillar");
	g_print("%s\n", string1->str);

	string2 = g_string_new("This is caterpillar speaking!");
	is_eq = g_string_equal(string1, string2);
	g_printf("equal: %s\n", is_eq ? "TRUE" : "FALSE");

	g_string_ascii_up(string1);
	g_printf("Upper: %s\n", string1->str);
	g_string_ascii_down(string1);
	g_printf("Down: %s\n", string1->str);

	g_string_free(string1, FALSE);
	g_string_free(string2, FALSE);

	return 0;
}
```

执行结果如下所示：

```
This is caterpillar speaking!
equal: TRUE
Upper: THIS IS CATERPILLAR SPEAKING!
Down: this is caterpillar speaking!
```

除了以上所介绍的，您还可以参考 **GString 说明文件**，另外 GLib 对于字符串还提供了 **String Utility Functions**，包括更多的字符串处理函式。

4.32GArray、GPtrArray、 GByteArray

在处理 C 的数组时，您必须处理数组长度的问题，您可以使用 GLib 的 GArray，并搭配各个

所提供的函式，在使用数组上会更为方便，GArray 的定义如下：

```
typedef struct {
  gchar *data;
  guint len;
} GArray;
```

您有两种建立 GArray 的方式：

```
GArray* g_array_sized_new(gboolean zero_terminated,
                          gboolean clear_,
                          guint element_size,
                          guint reserved_size);

GArray* g_array_new(gboolean zero_terminated,
                    gboolean clear_,
                    guint element_size);
```

g_array_sized_new()的第一个参数 zero_terminated 设定为 TRUE 的话，会加入最后一个额外元素，全部的位都设定为 0，clear_设定为 TRUE 的话，数组的全部元素会设定为 0，element_size 则是用来设定每个元素的长度，reserved_size 则是用以设定数组的长度，g_array_new()则是宏定义的简化版本，预设长度为 0，若加入新的元素，则自动增加数组长度。

要加入新的元素，可以使用 g_array_append_val()函式，要依索引取出元素，则可以使用 g_array_index()函式，下面这个程序是个简单的示范：

garray_demo.c

```
#include <glib.h>

int main(int argc, char *argv[]) {
    GArray *garray;
    gint i;

    garray = g_array_new(FALSE, FALSE, sizeof(gint));

    for (i = 0; i < 5; i++) {
        g_array_append_val(garray, i);
    }

    for (i = 0; i < 5; i++) {
        g_printf("%d\n", g_array_index(garray, gint, i));
```

```
    }

    g_array_free (garray, TRUE);

    return 0;
}
```

执行结果如下：

```
0
1
2
3
4
```

搭配GArray的函式还有g_array_remove_index()、g_array_sort()等，您可以参考 **Arrays** 的说明。

与GArray类似的是GPtrArray，只不过GArray储存的是数值（若是structs，则会复制至GArray中），而 GPtrArray 储存的是指标，GPtrArray 的定义如下：

```
typedef struct {
  gpointer *pdata;
  guint     len;
} GPtrArray;
```

举个例子来说，若要储存 GString 的指标，则可以如下所示：

gptrarray_demo.c

```
#include <glib.h>

int main(int argc, char *argv[]) {
    GPtrArray *array;
    gint i;
    GString *text;

    array = g_ptr_array_new();
    for (i = 0; i < 5; i++) {
        text = g_string_sized_new(5);
        g_string_printf(text, "TEST %d", i);
        g_ptr_array_add(array, text);
    }

    for (i = 0; i < 5; i++) {
        text = g_ptr_array_index(array, i);
        g_printf("%s\n", text->str);
    }
```

```
    g_ptr_array_free(array, TRUE);

    return 0;
}
```

执行结果如下所示：

```
TEST 0
TEST 1
TEST 2
TEST 3
TEST 4
```

GByteArray 则允许您储存 guint8 的数据，用于储存字节数据，为 GArray 的一个简化形式，其定义如下：

```
typedef struct {
   guint8 *data;
   guint len;
} GByteArray;
```

4.33GSList、 GList

GSList 是一个单向链接（Link）的节点，其定义如下：

```
typedef struct {
  gpointer data;
  GSList *next;
} GSList;
```

data 是节点数据（对象）的地址信息，next 是下一个节点数据的地址信息，搭配 GSList 的相关函式，您可以简单的进行链接节点的附加、插入、删除等动作，例如使用 g_slist_append()、g_slist_prepend()附加节点，使用 g_slist_sort()进行排序等。

下面这个程序是个简单的示范，使用 GSList 实作堆栈：

gslist_demo.c

```
#include <glib.h>

void for_callback(GString *string, gpointer user_data) {
    if(string) {
        printf("%s\n", string->str);
    }
}
```

```
int main(int argc, char *argv[]) {
    GString *string;
    GSList *list;
    int select;
    char input[10];

    list = NULL; //  一开始是没有节点的

    while(TRUE) {
        printf(
            "\n 请输入选项(-1 结束)：  (1)新增至堆栈  (2)删除顶端值  (3)显示所有内容");
        printf("\n$c>");
        scanf("%d", &select);

        if(select == -1) {
            break;
        }

        switch(select) {
            case 1:
                printf("\n 输入值：");
                scanf("%s", &input);
                string = g_string_new(input);
                list = g_slist_prepend(list, string);
                break;
            case 2:
                string = list->data;
                list = g_slist_remove(list, string);
                printf("\n 顶端值移除：%s", string->str);
                break;
            case 3:
                g_slist_foreach(list, (GFunc) for_callback, NULL);
                break;
            default:
                printf("\n 选项错误！");
        }
    }

    printf("\n");

    g_slist_free(list);

    return 0;
}
```

一个执行的结果如下所示：

```
请输入选项(-1 结束)：  (1)新增至堆栈  (2)删除顶端值  (3)显示所有内容
$c>1

输入值：caterpillar

请输入选项(-1 结束)：  (1)新增至堆栈  (2)删除顶端值  (3)显示所有内容
$c>1

输入值：momor

请输入选项(-1 结束)：  (1)新增至堆栈  (2)删除顶端值  (3)显示所有内容
$c>1

输入值：bush

请输入选项(-1 结束)：  (1)新增至堆栈  (2)删除顶端值  (3)显示所有内容
$c>3
bush
momor
caterpillar

请输入选项(-1 结束)：  (1)新增至堆栈  (2)删除顶端值  (3)显示所有内容
$c>2

顶端值移除：bush
请输入选项(-1 结束)：  (1)新增至堆栈  (2)删除顶端值  (3)显示所有内容
$c>-1
```

GList 则是双向链接，其定义如下：

```
typedef struct {
  gpointer data;
  GList *next;
  GList *prev;
} GList;
```

prev 是指向前一个节点，关于其搭配使用的函式，可参考 GList 说明文件。

4.34GHashTable

GHashTable 可以让您以杂凑表的方式来储存数据，储存时指定 Key 演算出 Hash 值以决定数据储存位置，要取回数据，也是指定 Key 演算出数据储存位置，以快速取得数据。

简单的说，您将 GHashTable 当作一个有很多间房间的房子，每个房间的门有一把钥匙，您将数据储存至房间中时，要顺便拥有一把钥匙，下次要取回资料时，就是根据这把钥匙取得。

您可以使用 g_hash_table_new()来建立 GHashTable：

```
GHashTable* g_hash_table_new(GHashFunc hash_func,
                             GEqualFunc key_equal_func);
```

g_hash_table_new()要指定一个演算 Hash 值的函式，GLib 提供了如 g_int_hash()、g_str_hash()函式可以直接使用，您也可以自订自己的演算 Hash 值的函式，例如：

```
guint hash_func(gconstpointer key) {
    ...
    return ...;
}
```

演算出 Hash 是决定储存的位置，接下来要确认 Key 的相等性，GLib 提供了如 g_int_equal()及 g_str_equal()函式可直接使用，同样的，您也可以自订函式：

```
gboolean key_equal_func(gconstpointer a, gconstpointer b) {
    ...
    return ...;
}
```

下面这个程序是个简单的示范：

ghashtable_demo.c

```
#include <glib.h>

int main(int argc, char *argv[]) {
    GHashTable *hashTable = g_hash_table_new(
                                    g_str_hash, g_str_equal);

    g_hash_table_insert(hashTable, "caterpillar", "caterpillar's message!!");
    g_hash_table_insert(hashTable, "justin", "justin's message!!");

    g_print("%s\n", g_hash_table_lookup(hashTable, "caterpillar"));
    g_print("%s\n", g_hash_table_lookup(hashTable, "justin"));

    g_hash_table_destroy(hashTable);
```

```
    return 0;
}
```

程序的执行结果如下：

```
justin's message!!
caterpillar's message!!
```

以下则示范如何使用 g_hash_table_iter_next()函式来进行 GHashTable 的迭代：

ghashtable_demo.c

```
#include <glib.h>

int main(int argc, char *argv[]) {
    GHashTableIter iter;
    gpointer key, value;
    GHashTable *hashTable;

    hashTable = g_hash_table_new(g_str_hash, g_str_equal);

    g_hash_table_insert(hashTable, "justin", "justin's message!!");
    g_hash_table_insert(hashTable, "momor", "momor's message!!");
    g_hash_table_insert(hashTable, "caterpillar", "caterpillar's message!!");

    g_hash_table_iter_init (&iter, hashTable);
    while(g_hash_table_iter_next(&iter, &key, &value)) {
        g_print("key\t: %s\nvalue\t: %s\n\n", key , value);
    }

    g_hash_table_destroy(hashTable);

    return 0;
}
```

程序的执行结果如下：

```
key     : justin
value   : justin's message!!

key     : caterpillar
value   : caterpillar's message!!

key     : momor
value   : momor's message!!
```

4.35GTree 与 GNode

GTree 实现了平衡二元树结构，在新增数据时会自动进行排序，并尝试维持树的高度与平衡，

您可以利用 Key 来储存资料至树中，并利用 Key 来快速取得数据。

直接来看个简单的例子：

gtree_demo.c

```
#include <glib.h>

gint key_compare_func(gconstpointer a, gconstpointer b) {
	return g_strcmp0(a, b);
}

gboolean traverse_func(gpointer key, gpointer value, gpointer data) {
	g_print("key\t: %s\nvalue\t: %s\n\n", key, value);
	return FALSE;
}

int main(int argc, char *argv[]) {
	GTree *tree;

	tree = g_tree_new(key_compare_func);
	g_tree_insert(tree, "justin", "justin's message!!");
	g_tree_insert(tree, "momor", "momor's message!!");
	g_tree_insert(tree, "caterpillar", "caterpillar's message!!");

	g_tree_foreach(tree, traverse_func, NULL);

	g_tree_destroy(tree);

	return 0;
}
```

这个程序会建立一个平衡二元树，利用指定的 key_compare_func 比较 Key 的大小，在这边利用 g_strcmp0()来比较字符串顺序，程序中插入三笔数据，插入的数据会自动依 Key 排序，所以取回时会是排序后的结果：

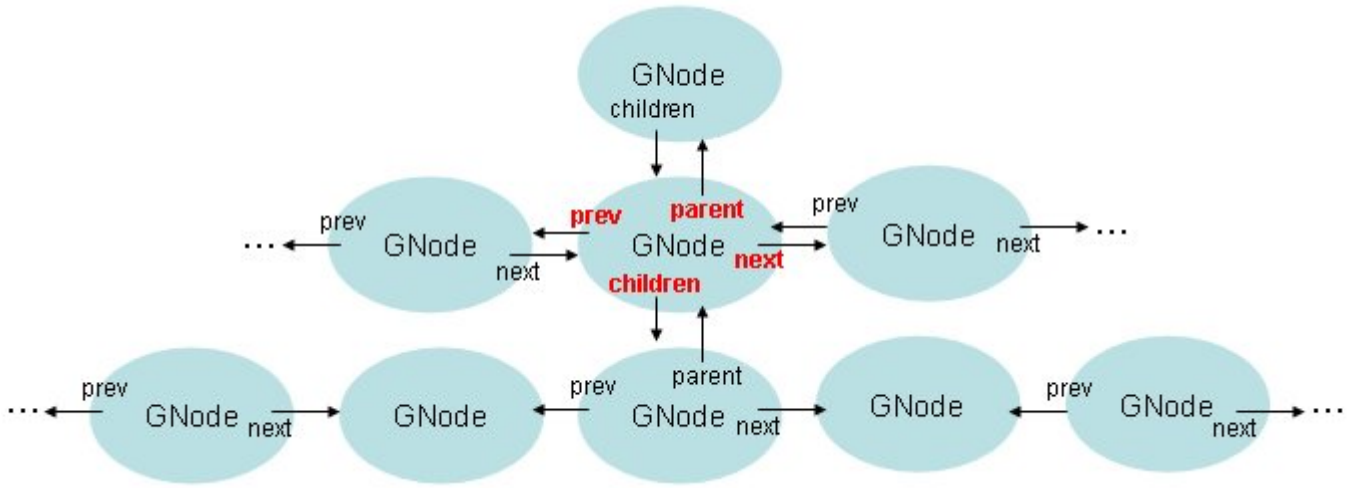
```
key	: caterpillar
value	: caterpillar's message!!

key	: justin
value	: justin's message!!

key	: momor
value	: momor's message!!
```

GNode 则是另一种允许您建立任意分枝节点的树结构，其定义如下：

```
typedef struct {
```

```
  gpointer data;
  GNode *next;
  GNode *prev;
  GNode *parent;
  GNode *children;
} GNode;
```

其中 parent、children 为父子节点，prev、next 是兄弟节点，其关系如下图所示：



4.36 内存配置

GLib 提供了对 C 标准 malloc ()、calloc、realloc()、free() 等函式的可携版本内存相关处理函式，例如 g_malloc()、 g_malloc0()、g_realloc()、g_free()等函式，例如 g_malloc()定义如下：

```
gpointer g_malloc (gsize n_bytes);
```

g_malloc0()则是会在配置内存之后，将所有的值设定为 0，以下举个简单的动态数组配置的例子：

g_malloc_demo.c

```
#include <glib.h>

int main(int argc, char *argv[]) {
  int size = 0;

  printf("请输入数组长度：");
  scanf("%d", &size);
  int *arr = g_malloc0(size * sizeof(int));
```

```
 int i;

 printf("显示元素值：\n");
 for(i = 0; i < size; i++) {
 printf("arr[%d] = %d\n", i, *(arr+i));
 }

 printf("指定元素值：\n");
 for(i = 0; i < size; i++) {
 printf("arr[%d] = ", i);
 scanf("%d" , arr + i);
 }

 printf("显示元素值：\n");
 for(i = 0; i < size; i++) {
 printf("arr[%d] = %d\n", i, *(arr+i));
 }

 g_free(arr);

 return 0;
}
```

一个执行结果如下所示：

```
请输入数组长度：3
显示元素值：
arr[0] = 0
arr[1] = 0
arr[2] = 0
指定元素值：
arr[0] = 1
arr[1] = 2
arr[2] = 3
显示元素值：
arr[0] = 1
arr[1] = 2
arr[2] = 3
```

g_malloc()配置失败，则应用程序会中止，您可以使用 g_try_malloc()，在配置失败后会传回 NULL 值。

如果要建立 struct 配置，则可以使用 g_new()，使用 g_new0()则会将所有 strut 初始为 0，例如：

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

```
struct _Ball {
    char color[10];
    double radius;
};
typedef struct _Ball Ball;

int main(int argc, char *argv[]) {
    Ball *ball = g_new(Ball, 3);

    ....

    return 0;
}
```

传回的指针已经是相对应的型态，无需再进行 CAST。

您可以看看 Memory Allocation 了解更多有关 GLib 的内存配置函式。

4.4 执行绪

4.41GThread

一个进程（Process）是一个包括有自身执行地址的程序，在一个多任务的操作系统中，可以分配 CPU 时间给每一个进程，CPU 在片段时间中执行某个进程，然后下一个时间片段跳至另一个进程去执行，由于转换速度很快，这使得每个程序像是在同时进行处理一般。

一个执行绪是进程中的一个执行流程，一个进程中可以同时包括多个执行绪，也就是说一个程序中同时可能进行多个不同的子流程，这使得一个程序可以像是同时间处理多个事务，例如一方面接受网络上的数据，另一方面同时计算数据并显示结果，一个多执行绪程序可以同时间处理多个子流程。

在 GLib 中，提供 GThread 来实现可携式的执行绪解决方案，以 内建 Signal 的发射与停止中的范例来说，当中使用到 pthread，因而只能在 Linux 之类的系统中执行，您可以改写为使用 GThread 的方式，例如：

gthread_demo.c

```
#include <gtk/gtk.h>

gpointer signal_thread(gpointer arg) {
    int i;
    for(i = 0; i < 5; i++) {
        g_usleep(1000000); // 暂停一秒
        g_signal_emit_by_name(arg, "clicked");
```

```
    }
}

// 自订 Callback 函式
void button_clicked(GtkWidget *button, gpointer data) {
    g_print("按钮按下：%s\n", (char *) data);
}

int main(int argc, char *argv[]) {
    GtkWidget *window;
    GtkWidget *button;

    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window), "哈啰！ GTK+！ ");

    button = gtk_button_new_with_label("按我");
    gtk_container_add(GTK_CONTAINER(window), button);

    g_signal_connect(GTK_OBJECT(window), "destroy",
                     G_CALLBACK(gtk_main_quit), NULL);
    g_signal_connect(GTK_OBJECT(button), "clicked",
                     G_CALLBACK(button_clicked), "哈啰！ 按钮！ ");

    gtk_widget_show(window);
    gtk_widget_show(button);

    if(!g_thread_supported()) {
        g_thread_init(NULL);
    }

    g_thread_create(signal_thread, button, FALSE, NULL);

    gtk_main();

    return 0;
}
```

在使用 g_thread_create()函式之前，先使用 g_thread_supported()函式检查一下执行绪系统是否已初始化， signal_thread 是自订的 callback 函式，所建立的 GThread 会执行该函式，g_usleep()是用来暂停执行绪之用，单位是微秒。

为了编译这个程序，您必须设定 gthread-2.0 链接库路径信息，可以使用 pkg-config 取得这个信息，例如：

```
pkg-config --libs gthread-2.0
```

事实上，GThread 要在建立 GMainLoop 下才能使用，Main Loop 的目的是等待事件的发生，并呼叫适当的 callback 函式，在 GTK 的程序，不用自行建立 Main Loop，因为在 gtk_init() 中已帮您建立，而 Main Loop 的执行则是在 gtk_main()中替您进行。

下面这个范例程序，示范了如何自行建立 Main Loop，并建立三个执行绪，其中一个执行绪会执行 checking_thread()函式，以检查另两个执行绪是否已完成，若已完成则结束 Main Loop：

gthread_demo.c

```
#include <gtk/gtk.h>

gboolean thread1_end = FALSE;
gboolean thread2_end = FALSE;

gpointer thread1(gpointer data) {
	int i;
	for(i = 0; i < 10; i++) {
		g_print("Thread1: %s\n", data);
		g_usleep(1000000);
	}
	thread1_end = TRUE;
}

gpointer thread2(gpointer data) {
	int i;
	for(i = 0; i < 10; i++) {
		g_print("Thread2: %s\n", data);
		g_usleep(1000000);
	}
	thread2_end = TRUE;
}

gpointer checking_thread(gpointer mloop) {
	while(TRUE) {
		if(thread1_end && thread2_end) {
			g_main_loop_quit(mloop);
			break;
		}
		g_usleep(1000);
	}
}
```

```
int main(int argc, char *argv[]) {
	GMainLoop *mloop;

	if(!g_thread_supported()) {
		g_thread_init(NULL);
	}

	mloop = g_main_loop_new(NULL, FALSE);

	g_thread_create(thread1, "Running", FALSE, NULL);
	g_thread_create(thread2, "Going", FALSE, NULL);
	g_thread_create(checking_thread, mloop, FALSE, NULL);

	g_main_loop_run(mloop);

	return 0;
}
```

一个执行的结果如下所示：

```
Thread1: Running
Thread2: Going
Thread2: Going
Thread1: Running
Thread1: Running
Thread2: Going
Thread2: Going
Thread1: Running
Thread1: Running
Thread2: Going
Thread1: Running
Thread2: Going
Thread2: Going
Thread1: Running
Thread2: Going
Thread1: Running
Thread1: Running
Thread2: Going
Thread1: Running
Thread2: Going
```

4.42GMutex

如果您的程序只是一个单执行绪，单一流程的程序，那么通常您只要注意到程序逻辑的正确，

您的程序通常就可以正确的执行您想要的功能，但当您的程序是多执行绪程序，多流程同时执行时，那么您就要注意到更多的细节，例如在多执行绪共享同一对象的数据时。

如果一个对象所持有的数据可以被多执行绪同时共享存取时，您必须考虑到「数据同步」的问题，所谓数据同步指的是两份数据的整体性一致，例如对象 A 有 name 与 id 两个属性，而有一份 A1 数据有 name 与 id 的数据要更新对象 A 的属性，如果 A1 的 name 与 id 设定给 A 对象完成，则称 A1 与 A 同步，如果 A1 数据在更新了对象的 name 属性时，突然插入了一份 A2 数据更新了 A 对象的 id 属性，则显然的 A1 数据与 A 就不同步，A2 数据与 A 也不同步。

数据在多执行绪下共享时，就容易因为同时多个执行绪可能更新同一个对象的信息，而造成对象数据的不同步，因为数据的不同步而可能引发的错误通常不易察觉，而且可能是在您程序执行了几千几万次之后，才会发生错误，而这通常会发生在您的产品已经上线之后，甚至是程序已经执行了几年之后。

这边举个简单的例子：

gmutex_demo.c

```
#include <glib.h>

struct _User {
	GString *name;
	GString *id;
	glong count;
};
typedef struct _User User;

void user_set_name_id(User *user, GString *name, GString *id) {
	user->name = name;
	user->id = id;
	if(!user_check_name_id(user)) {
		g_print("%d: illegal name or id....\n", user->count);
	}
	user->count++;
}

gboolean user_check_name_id(User *user) {
	return user->name->str[0] == user->id->str[0];
}

gpointer thread1(gpointer user) {
	GString *name = g_string_new("Justin Lin");
	GString *id = g_string_new("J.L.");
	while(TRUE) {
		user_set_name_id(user, name, id);
```

```
	}
}

gpointer thread2(gpointer user) {
	GString *name = g_string_new("Shang Hwang");
	GString *id = g_string_new("S.H.");
	while(TRUE) {
		user_set_name_id(user, name, id);
	}
}

int main(int argc, char *argv[]) {
	GMainLoop *mloop;

	if(!g_thread_supported()) {
		g_thread_init(NULL);
	}

	User user;

	mloop = g_main_loop_new(NULL, FALSE);

	g_thread_create(thread1, &user, FALSE, NULL);
	g_thread_create(thread2, &user, FALSE, NULL);

	g_main_loop_run(mloop);

	return 0;
}
```

在这个程序中，您可以设定使用者的名称与缩写 id，并简单检查一下名称与 id 的第一个字是否相同，单就这个程序本身而言，user_set_name_id()并没有任何的错误，但如果它被用于多执行绪的程序中，而且同一个对象被多个执行存取时，就会"有可能"发生错误，一个执行的可能结果如下（为简化范例，并无设置停止条件，请直接 Ctrl+C 结束程序）：

```
51307: illegal name or id....
94812: illegal name or id....
140423: illegal name or id....
174257: illegal name or id....
214260: illegal name or id....
214260: illegal name or id....
259266: illegal name or id....
314738: illegal name or id....
350144: illegal name or id....
402701: illegal name or id....
444026: illegal name or id....
```

```
481165: illegal name or id....
....
```

看到了吗？如果以单执行绪的观点来看，上面的讯息在测试中根本不可能出现，然而在这个程序中却出现了错误，而且重点是，第一次错误是发生在第 51307 次的设定（您的计算机上可能是不同的数字），如果您在程序完成并开始应用之后，这个时间点可能是几个月甚至几年之后。

问题出现哪？在于这边：

```
void user_set_name_id(User *user, GString *name, GString *id) {
    user->name = name;
    user->id = id;
    if(!user_check_name_id(user)) {
        g_print("%d: illegal name or id....\n", user->count);
    }
    user->count++;
}
```

虽然您设定给它的参数并没有问题，在某个时间点时，thread1 设定了"Justin Lin"，"J.L."给 name 与 id，在进行测试的前一刻，thread2 可能此时刚好呼叫 user_set_name_id()，在 name 被设定为"Shang Hwang"时，user_check_name_id()开始执行，此时 name 等于"Shang Hwang"，而 id 还是"J.L."，所以 user_check_name_id()就会传回 FALSE，结果就显示了错误讯息。

您必须同步数据对对象的更新，也就是在有一个执行绪正在设定 user 对象的数据时，不可以又被另一个执行绪同时进行设定，您可以使用 GMutex 来进行这个动作，例如：

gmutex_demo.c

```
#include <glib.h>

GMutex *mutex = NULL;

struct _User {
    GString *name;
    GString *id;
    glong count;
};
typedef struct _User User;

void user_set_name_id(User *user, GString *name, GString *id) {
    if(g_mutex_trylock(mutex)) {
        user->name = name;
```

```
            user->id = id;
            if(!user_check_name_id(user)) {
                g_print("%d: illegal name or id....\n", user->count);
            }
            user->count++;
            g_mutex_unlock(mutex);
        }
        else {
            g_usleep(1000);
        }
}

gboolean user_check_name_id(User *user) {
        return user->name->str[0] == user->id->str[0];
}

gpointer thread1(gpointer user) {
        GString *name = g_string_new("Justin Lin");
        GString *id = g_string_new("J.L.");
        while(TRUE) {
            user_set_name_id(user, name, id);
        }
}

gpointer thread2(gpointer user) {
        GString *name = g_string_new("Shang Hwang");
        GString *id = g_string_new("S.H.");
        while(TRUE) {
            user_set_name_id(user, name, id);
        }
}

int main(int argc, char *argv[]) {
        GMainLoop *mloop;

        if(!g_thread_supported()) {
            g_thread_init(NULL);
        }

        User user;

        mloop = g_main_loop_new(NULL, FALSE);

        mutex = g_mutex_new();
```

```
    g_thread_create(thread1, &user, FALSE, NULL);
    g_thread_create(thread2, &user, FALSE, NULL);

    g_main_loop_run(mloop);

    return 0;
}
```

g_mutex_trylock()会尝试锁定 GMutex，如果成功就传回 TRUE，并继续执行程序代码，若此时有其它的执行绪也尝试鑽定 GMutex，则会传回 FALSE，并无法执行 GMutex 现已锁定的程序代码范围，在这个程序中，则是使用 if-else，在无法锁定时，先睡眠 1 毫秒后再尝试。

4.43GCond

在执行绪的同步化时，有些条件下执行绪必须等待，有些条件下则不用，这可以使用 GCond 来达到。

例如在生产者（Producer）与消费者（Consumer）的例子中，如果生产者会将产品交到仓库，而消费者从仓库取走产品，仓库一次只能持有固定数量产品，如果生产者生产了过多的产品，仓库叫生产者等一下（wait），如仓库中有空位放产品了再发信号（signal）通知生产者继续生产，如果仓库中没有产品了，仓库会告诉消费者等一下（wait），如果仓库中有产品了再发信号（signal）通知消费者来取走产品。

以下举一个最简单的：生产者每次生产一个 int 整数交至仓库，而消费者从仓库取走整数，仓库一次只能持有一个整数，以程序实例来看：

gcond_demo.c

```
#include <glib.h>

GMutex *mutex = NULL;
GCond* cond = NULL;
int storage = -1;

gboolean producer_thread_end = FALSE;
gboolean consumer_thread_end = FALSE;

// 生产者透过此函式设定产品
void produce_product(int product) {
    g_mutex_trylock(mutex);

    if(storage != -1) {
        // 目前没有空间收产品，请稍候!
        g_cond_wait(cond, mutex);
    }
```

```
        storage = product;
        g_print("生产者设定  %d\n", storage);

        // 唤醒消费者可以继续工作了
        g_cond_signal(cond);

        g_mutex_unlock(mutex);
}

// 消费者透过此函式取走产品
int consume_product() {
        g_mutex_trylock(mutex);

        if(storage == -1) {
                // 缺货了，请稍候!
                g_cond_wait(cond, mutex);
        }

        int p = storage;
        g_print("消费者取走  %d\n", storage);

        storage = -1;

        // 唤醒生产者可以继续工作了
        g_cond_signal(cond);

        g_mutex_unlock(mutex);

        return p;
}

// 生产者执行绪会执行此函式
gpointer producer_thread(gpointer data) {
        int i;
        for(i = 1; i <= 10; i++) {
                g_usleep(rand());
                produce_product(i);
        }
        producer_thread_end = TRUE;
}

// 消费者执行绪会执行此函式
gpointer consumer_thread(gpointer data) {
```

```
    int i;
    for(i = 1; i <= 10; i++) {
        g_usleep(rand());
        consume_product();
    }
    consumer_thread_end = TRUE;
}

gpointer checking_thread(gpointer mloop) {
    while(TRUE) {
        if(producer_thread_end && consumer_thread_end) {
            g_main_loop_quit(mloop);
            break;
        }
        g_usleep(1000);
    }
}

int main(int argc, char *argv[]) {
    GMainLoop *mloop;

    if(!g_thread_supported()) {
        g_thread_init(NULL);
    }

    mloop = g_main_loop_new(NULL, FALSE);

    mutex = g_mutex_new();
    cond = g_cond_new();

    g_thread_create(producer_thread, NULL, FALSE, NULL);
    g_thread_create(consumer_thread, NULL, FALSE, NULL);
    g_thread_create(checking_thread, mloop, FALSE, NULL);

    g_main_loop_run(mloop);

    return 0;
}
```

执行结果：

```
生产者设定 1
消费者取走 1
生产者设定 2
消费者取走 2
生产者设定 3
```

```
消费者取走 3
生产者设定 4
消费者取走 4
生产者设定 5
消费者取走 5
生产者设定 6
消费者取走 6
生产者设定 7
消费者取走 7
生产者设定 8
消费者取走 8
生产者设定 9
消费者取走 9
生产者设定 10
消费者取走 10
```

生产者会生产 10 个整数，而消费者会消耗 10 个整数，由于仓库只能放置一个整数，所以每生产一个就消耗一个，其结果如上所示是无误的。