GTK+ 简介

关于本初级教程 I

本 GTK+程序初级教程采用了“引导”式的方法来教你如何使用 GTK+。本教程所用的编程语言为 C 语言。并且整个教程中方法已经在 Linux 中测试通过。本教程适用 GTK+的初级兼代中级水平的程序员。

GTK+

GTK+ 是一种函数库是用来帮助制作图形交互界面的。整个函数库都是由 C 语言来编写的。GTK+函数库通常也叫做 GIMP 工具包。因为，一开始 GTK+是用来编写“GIMP”–这个图形处理软件的。 从那以后， GTK+就逐渐成为了 Linux and BSDUnix 下最受欢迎的用于图形交互界面（GUI）设计的工具包之一。时至今日,大多数的 Linux 下的 GUI 软件（有图形交互界面）都是由 GTK+或者 QT 来编写的。GTK+是一种面向对象式的 API(applicationprogramming interface)。Glib 是 GTK+的基础，而这种“面向对象系统”正是由“Glib”来提供的。 **GObject** 也就是这种面向对象的机制可以为 GTK+绑定很多种开发语言。目前存在的语言有：C++, Python, Perl, Java, C#，PHP,等其他高级语言。

GTK+ 和以下“函数库”存在着依赖关系

- Glib
- Pango
- ATK
- GDK
- GdkPixbuf
- Cairo

Glib 是一种通用的函数库。她提供了各种各样的语言特性，譬如说：各种数据类型，字符串函数，错误通知，消息队列和线程。**Pango** 是一种函数库，用来实现国际化和本地化的功能。**ATK** 一种平易近人的工具函数包，她提供了快捷键服务为肢体有缺陷的人使用电脑提供了便利。 **GDK** 是一种函数库，她为整个 GTK+图形库系统提供了一些底层的“图形实现”和“窗口实

现”的方法。在 Linux 中 GDK 是位于 X 服务器和 GTK+函数库之间的。在最近的 GTK+发行版本中，越来越多的功能性函数，都交给了 Cairo 函数库来处理。 **GdkPixbuf** 函数库是一种函数库工具包用于加载图像和维护图像“缓存”的（pixel buffer）。 **Cairo** 是一种函数库用于制作二维图像。从 GTK+2.8 版本以后，Cairo 就正式成为 GTK+系统中的一员了。

Gnome and XFce 桌面环境系统都是用 GTK+来编程实现的. SWT and wxWidgets 是种很著名的编程框架,也是用 GTK+来编程实现的。最杰出的 GTK+软件的代表是 Firefox （火狐浏览器）和 Inkscape。

编译 GTK+ 应用程序

有一个非常方便的工具--"pkg-config",可以帮助我们编译 GTK+的应用程序。**pgk-config** 可以提供各种安装函数库（譬如 GDK,Pango 等）的位置。简单点说,就是我们如果想使用某种函数库，Pkg-config 就会为我们提供所需要的 lib 与 include 文件的位置。 pkg-config 是从一些通常以“.pc”结尾的文件中，得到所需要的信息的。

```
gcc -o simple simple.c `pkg-config --libs --cflags gtk+-2.0`
```

上面的编译命令，就是在展示我们如何去编译一个简单的源文件——“simple.c”。

```
$ pkg-config --cflags gtk+-2.0

-I/usr/include/gtk-2.0 -I/usr/lib/gtk-2.0/include
-I/usr/include/atk-1.0

-I/usr/include/cairo -I/usr/include/pango-1.0 -I/usr/include/glib-2.0

-I/usr/lib/glib-2.0/include -I/usr/include/freetype2
-I/usr/include/libpng12
```

上面列出了 pkg-config 为你自动提供的编译所需要的 include 文件的信息。

```
$ pkg-config --libs gtk+-2.0

-lgtk-x11-2.0 -lgdk-x11-2.0 -latk-1.0 -lgdk_pixbuf-2.0 -lm
-lpangocairo-1.0
```

```
-lfontconfig -lXext -lXrender -lXinerama -lXi -lXrandr

-lXcursor -lXfixes -lpango-1.0 -lcairo -lX11 -lgobject-2.0

-lgmodule-2.0 -ldl -lglib-2.0
```

上面列出了 pkg-config 为你自动提供的编译所需要的 lib 文件的信息。

Sources

- gtk.org
- gtkforums.com
- GTK+ / Gnome application development

第一个 GTK+程序

在这一章节中，我们将开始编写第一个 GTK+程序。

超级简单的例子

我们要“制造”一个超级简单的 GTK+程序。就是显示一个空白的窗口。

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

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

  GtkWidget *window;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_widget_show(window);

  gtk_main();

  return 0;
```

```
}
```

这个例子就是为我们显示了一个空白的窗口。

```
  gcc -o simple simple.c `pkg-config --libs --cflags gtk+-2.0`
```

这就是我们用来编译这个例子的命令。下面我们将对这个简单的程序进行详细的解读。

```
gtk_init(&argc, &argv);
```

这就是在初始化整个 GTK+程序，是每一个 GTK+程序必不可少的部分。

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

这里我们首先生成了一个构件—— **GtkWindow**. 这个窗口构件的种类是 **GTK_WINDOW_TOPLEVEL**. . Toplevel 窗口拥有一个标题栏和边框。他们同意由窗口管理器进行管理。

```
gtk_widget_show(window);
```

在我们生成了一个窗口构件以后，必不可少的是，我们需要用这句语句来显示构件。

```
gtk_main();
```

这句代码语句将进入“主循环”。 在这一点上，GTK+程序将安静的等待“事件”(event)的发生，以便做出相应的反应。

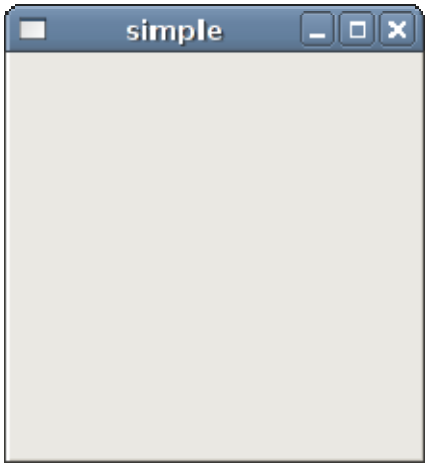

Figure: Simple

生成一个窗口

如果我们不安排窗口的摆放位置的话，那么窗口管理器将为我们给这个窗口“安一个家”。在下个例子中，我们将走进“窗口”。

```
#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), "Center");

  gtk_window_set_default_size(GTK_WINDOW(window), 230, 150);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_widget_show(window);

  g_signal_connect_swapped(G_OBJECT(window), "destroy",
G_CALLBACK(gtk_main_quit), NULL);

  gtk_main();

  return 0;

}
```

在我们上面的这个例子，我将深入了解窗口构件（window widget），为窗口构件设置标题和大小。

```
gtk_window_set_title(GTK_WINDOW(window), "Center");
```

gtk_window_set_title() 这个函数就可以为 window 设置一个标题，如果我们不用这个函数的话，GTK+将用源文件的名字来作为窗口的标题。

```
gtk_window_set_default_size(GTK_WINDOW(window), 230, 150);
```

上面这个代码片段为 window 设置了 230x150 像素的大小。值得注意的是，我们这里提到的大小是指主窗口的大小；而不包括窗口管理器提供的修饰或者装饰的部分。

```
gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
```

这段代码把窗口设定在显示器的中央。

```
g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), NULL);
```

在之前的例子中，我们没有设置窗口的关闭，当我按下右上角的“X”时。我们可以看到，如果是从命令行的方式来运行例子程序的话，默认情况下窗口程序并不会对你刚刚的动作作出反应（当然，最新的窗口管理器，譬如 X11 会强行关闭）。我们必须要明确为这个例子程序连接上一个关闭的信号（the destroy signal），就是 **gtk_main_quit()** 这个函数。

应用程序图标的制作

在下面的例子中，我们会进行应用程序图标的制作。大多数的窗口管理器都会把图标放置在标题栏的左上方和任务栏上。

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

GdkPixbuf *create_pixbuf(const gchar * filename){

   GdkPixbuf *pixbuf;

   GError *error = NULL;

   pixbuf = gdk_pixbuf_new_from_file(filename, &error);

   if(!pixbuf) {

      fprintf(stderr, "%s\n", error->message);

      g_error_free(error);

   }

   return pixbuf;

}
```

```
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), "icon");

  gtk_window_set_default_size(GTK_WINDOW(window), 230, 150);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_window_set_icon(GTK_WINDOW(window), create_pixbuf("web.png"));


  gtk_widget_show(window);


  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), NULL);

  gtk_main();

  return 0;

}
```

上面的就是为一个填加上图标的窗口程序了.

```
gtk_window_set_icon(GTK_WINDOW(window), create_pixbuf("web.png"));
```

函数 **gtk_window_set_icon()** 是为窗口设置图标用的。函数 **create_pixbuf()** 作用是从一个 png 图象文件中生成 **GdkPixbuf** 类型数据。

```
pixbuf = gdk_pixbuf_new_from_file(filename, &error);
```

根据官方公布的文档说明，函数 **gdk_pixbuf_new_from_file()** 一个文件中加载图象数据，从而生成一个新的 pixbuf。 至于文件中包含图象的格式，是由系统自动检测的。如果该函数返回值是 NULL 的话，程序就会出现错误。

Figure: Icon

增加 和 减少

下面我们将用一个简单的示例，来完成了“GTK+程序设计初级教程”的第一阶段，在这里我们用到了三个构件：两个按钮和一个标签。这个标签将保存一个整数，两个按钮会分别增加和减少这个数。

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

gint count = 0;

char buf[5];

void increase(GtkWidget *widget, gpointer label){

  count++;

  sprintf(buf, "%d", count);

  gtk_label_set_text(label, buf);

}

void decrease(GtkWidget *widget, gpointer label){

  count--;

  sprintf(buf, "%d", count);

  gtk_label_set_text(label, buf);
```

```
}

int main(int argc, char** argv) {
  GtkWidget *label;
  GtkWidget *window;
  GtkWidget *frame;
  GtkWidget *plus;
  GtkWidget *minus;
  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
  gtk_window_set_default_size(GTK_WINDOW(window), 250, 180);
  gtk_window_set_title(GTK_WINDOW(window), "+-");
  frame = gtk_fixed_new();
  gtk_container_add(GTK_CONTAINER(window), frame);
  plus = gtk_button_new_with_label("+");
  gtk_widget_set_size_request(plus, 80, 35);
  gtk_fixed_put(GTK_FIXED(frame), plus, 50, 20);
  minus = gtk_button_new_with_label("-");
  gtk_widget_set_size_request(minus, 80, 35);
```

```
  gtk_fixed_put(GTK_FIXED(frame), minus, 50, 80);

  label = gtk_label_new("0");

  gtk_fixed_put(GTK_FIXED(frame), label, 190, 58);

  gtk_widget_show_all(window);

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

  g_signal_connect(plus, "clicked", G_CALLBACK(increase), label);

  g_signal_connect(minus, "clicked", G_CALLBACK(decrease), label);

  gtk_main();

  return 0;
```

```
}
```

这个示例代码完的功能是：增加和减少对象 GtkLabel 的值。

```
g_signal_connect(plus, "clicked", G_CALLBACK(increase), label);
```

我们把回调函数 **increase()**和增加按钮进行了连结。还有值得注意的是我们把 label 作为回调函数调用的参数。 这样的话就可以在回调函数 increase()中对 label 进行处理。

```
count++;

sprintf(buf, "%d", count);

gtk_label_set_text(label, buf);
```

在“增加”的回调函数中，增加数字。 然后在 label 中的数字就会随之增加。

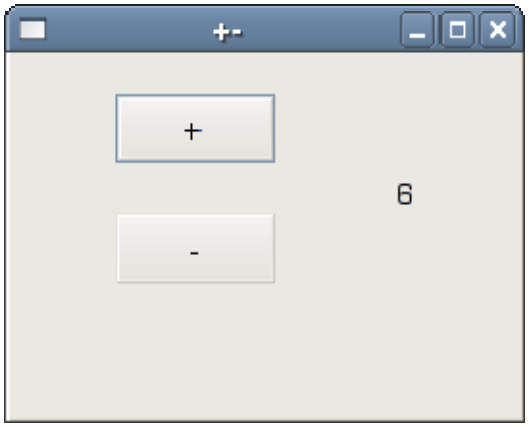

Figure: Increase - Decrease

“菜单”（menubar）和“工具栏”（toolbars）

在这个部分的 GTK+程序设计教程中，我们使用“菜单”和“工具栏”。

“菜单”（ menubar） 是 GUI 程序中最为常见的部分之一。各种各样的命令和功能都可以借以“菜单”来实现。 当我们习惯在终端（console）中启动应用程序的时候，必须要记得很多复杂的命令和参数 ，在本章节中我们将 这一切都转化为可见的操作。菜单和工具栏中标准化的操作，将让你摆脱学习新软件所耗费的大量时间和精力。

简单的菜单示列

在我们的第一个例子中，我们将生成一个含有文件菜单的菜单栏。文件菜单将只有一个菜单条（menu item）。如果点击这个菜单条程序将退出。

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

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

  GtkWidget *window;

  GtkWidget *vbox;

  GtkWidget *menubar;

  GtkWidget *filemenu;

  GtkWidget *file;
```

```
  GtkWidget *quit;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_window_set_default_size(GTK_WINDOW(window), 250, 200);

  gtk_window_set_title(GTK_WINDOW(window), "menu");

  vbox = gtk_vbox_new(FALSE, 0);

  gtk_container_add(GTK_CONTAINER(window), vbox);

  menubar = gtk_menu_bar_new();

  filemenu = gtk_menu_new();

  file = gtk_menu_item_new_with_label("File");

  quit = gtk_menu_item_new_with_label("Quit");

  gtk_menu_item_set_submenu(GTK_MENU_ITEM(file), filemenu);
```

```
  gtk_menu_shell_append(GTK_MENU_SHELL(filemenu), quit);
```

```
  gtk_menu_shell_append(GTK_MENU_SHELL(menubar), file);

  gtk_box_pack_start(GTK_BOX(vbox), menubar, FALSE, FALSE, 3);

  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), NULL);

  g_signal_connect(G_OBJECT(quit),
"activate",G_CALLBACK(gtk_main_quit), NULL);
```

```
    gtk_widget_show_all(window);

    gtk_main();

    return 0;

}
```

生成一个菜单栏的确会让人有点疑惑。我们要牢记的是一个菜单栏和一个菜单都是源属于同一个构件的，也就是菜单外壳（**menu shell**）。菜单选项（**menu items** ）是一个只对菜单有效的子构件。他们通常用来实现子菜单。

```
menubar = gtk_menu_bar_new();

filemenu = gtk_menu_new();
```

在上面的代码中我们生成了一个菜单栏构件（menubar）和一个菜单构件(menu)。

```
gtk_menu_item_set_submenu(GTK_MENU_ITEM(file), filemenu);
```

上面的代码就会生成一个名为“文件”的菜单。这也就是说其实菜单栏就是一个菜单外壳。很显然这里的文件菜单也是一个菜单外壳。这就是为什么我们把文件菜单称为子菜单或者说是一个子外壳。

```
gtk_menu_shell_append(GTK_MENU_SHELL(filemenu), quit);

gtk_menu_shell_append(GTK_MENU_SHELL(menubar), file);
```

菜单选项由函数 **gtk_menu_shell_append()** f 来实现。然后一般情况下菜单选项被填加进菜单外壳里。 在我们的这个例子中，“quit”菜单选项是被填加进“file”菜单栏里，然后类似的是“file”菜单选项被填加进菜单中（menubar）。

```
g_signal_connect(G_OBJECT(quit),"activate",G_CALLBACK(gtk_main_quit),
NULL);
```

当你单击“quit”菜单按钮，程序就会退出。

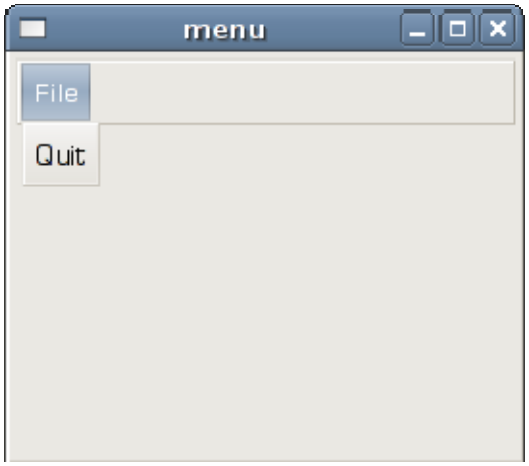

Figure: Simple menu

图象菜单, mnemonics &accelerators

在接下来的这个例子中，我们将更进一步的去探索，在 GTK+ 系统中我们可以应用的功能。**Accelerators** 是快捷键的意思，用来方便的用键盘上的组合键激活一个菜单选项。 **Mnemonics** 也是是快捷键用于用于 GUI 的基础。他们的具体表现都为带有下画线的字符。（译者：似乎说的不清楚啊，呵呵。具体的区别还是请参见下面的具体代码实现以及对应的程序效果图）

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

#include <gdk/gdkkeysyms.h>

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

  GtkWidget *window;

  GtkWidget *vbox;

  GtkWidget *menubar;
```

```
  GtkWidget *filemenu;
```

```
  GtkWidget *file;

  GtkWidget *new;
```

```
    GtkWidget *open;
    GtkWidget *quit;
    GtkWidget *sep;
    GtkAccelGroup *accel_group = NULL;
    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
    gtk_window_set_default_size(GTK_WINDOW(window), 250, 200);
    gtk_window_set_title(GTK_WINDOW(window), "menu");
    vbox = gtk_vbox_new(FALSE, 0);
    gtk_container_add(GTK_CONTAINER(window), vbox);
    menubar = gtk_menu_bar_new();
    filemenu = gtk_menu_new();
    accel_group = gtk_accel_group_new();
    gtk_window_add_accel_group(GTK_WINDOW(window), accel_group);
    file = gtk_menu_item_new_with_mnemonic("_File");
    new = gtk_image_menu_item_new_from_stock(GTK_STOCK_NEW, NULL);
    open = gtk_image_menu_item_new_from_stock(GTK_STOCK_OPEN, NULL);
    sep = gtk_separator_menu_item_new();
```

```
  quit = gtk_image_menu_item_new_from_stock(GTK_STOCK_QUIT,
accel_group);

  gtk_widget_add_accelerator(quit, "activate", accel_group, GDK_q,
GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE);

  gtk_menu_item_set_submenu(GTK_MENU_ITEM(file), filemenu);

  gtk_menu_shell_append(GTK_MENU_SHELL(filemenu), new);

  gtk_menu_shell_append(GTK_MENU_SHELL(filemenu), open);

  gtk_menu_shell_append(GTK_MENU_SHELL(filemenu), sep);

  gtk_menu_shell_append(GTK_MENU_SHELL(filemenu), quit);

  gtk_menu_shell_append(GTK_MENU_SHELL(menubar), file);

  gtk_box_pack_start(GTK_BOX(vbox), menubar, FALSE, FALSE, 3);

  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), NULL);

  g_signal_connect(G_OBJECT(quit),
"activate",G_CALLBACK(gtk_main_quit), NULL);

  gtk_widget_show_all(window);

  gtk_main();

  return 0;

}
```

在上面的整个代码程序中，向大家展示了是如何向一个菜单选项中去填加一个图象的。当然也包括了如何使用 accelerator 以及 mnemonics 。

```
  accel_group = gtk_accel_group_new();

  gtk_window_add_accel_group(GTK_WINDOW(window), accel_group);
```

```
  ...

  quit = gtk_image_menu_item_new_from_stock(GTK_STOCK_QUIT,
accel_group);

  gtk_widget_add_accelerator(quit, "activate", accel_group, GDK_q,
GDK_CONTROL_MASK, GTK_ACCEL_VISIBLE);
```

一个 accelerator 组有好多个 accelerators 快捷键。这里我们生成了一个“Ctrl + q” accelerators 快捷键。

```
 file = gtk_menu_item_new_with_mnemonic("_File");
```

我们需要调用函数 **gtk_menu_item_new_with_mnemonic()** 来生成一个 mnemonic 快捷键。你可以按下键盘上的“ Alt + F”就可以看到. mnemonic 快捷键的效果。

```
 new = gtk_image_menu_item_new_from_stock(GTK_STOCK_NEW, NULL);

 open = gtk_image_menu_item_new_from_stock(GTK_STOCK_OPEN, NULL);
```

在上面的代码中我们生成了两个带有图象的菜单选项。在所调用的函数的第二个参数中，我们设置为 NULL，这样达到的效果是我们自动生成了 accelerators 快捷键。我们也为菜单选项分别提供了图象与文字。

```
 sep = gtk_separator_menu_item_new();
```

菜单选项能够被一个水平的分割线隔开。这样的话我们就可以从逻辑上把一些菜单选项给区分开来。

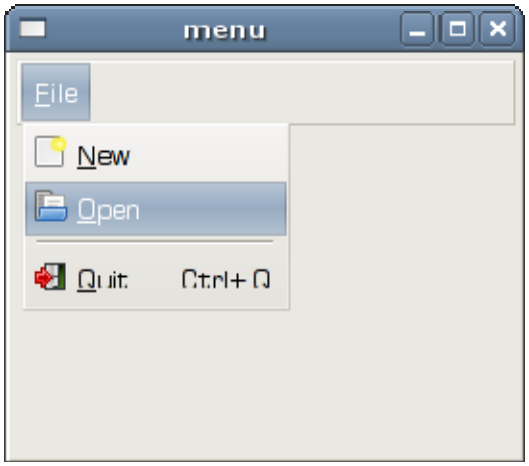

Figure: Menu example

选择（Check）菜单选项 （menu item）

GtkCheckMenuItem 便是一个可以生成带有选择的菜单选项。

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

void toggle_statusbar(GtkWidget *widget, gpointer statusbar) {

  if (gtk_check_menu_item_get_active(GTK_CHECK_MENU_ITEM(widget))) {

    gtk_widget_show(statusbar);

  } else { gtk_widget_hide(statusbar); }

}

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

  GtkWidget *window;

  GtkWidget *vbox;

  GtkWidget *menubar;

  GtkWidget *viewmenu;

  GtkWidget *view;

  GtkWidget *tog_stat;

  GtkWidget *statusbar;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_window_set_default_size(GTK_WINDOW(window), 250, 200);
```

```
  gtk_window_set_title(GTK_WINDOW(window), "view statusbar");
```

```
  vbox = gtk_vbox_new(FALSE, 0);

  gtk_container_add(GTK_CONTAINER(window), vbox);

  menubar = gtk_menu_bar_new();

  viewmenu = gtk_menu_new();

  view = gtk_menu_item_new_with_label("View");

  tog_stat = gtk_check_menu_item_new_with_label("View Statusbar");

  gtk_check_menu_item_set_active(GTK_CHECK_MENU_ITEM(tog_stat),TRUE);

  gtk_menu_item_set_submenu(GTK_MENU_ITEM(view), viewmenu);

  gtk_menu_shell_append(GTK_MENU_SHELL(viewmenu), tog_stat);

  gtk_menu_shell_append(GTK_MENU_SHELL(menubar), view);

  gtk_box_pack_start(GTK_BOX(vbox), menubar, FALSE, FALSE, 3);

  statusbar = gtk_statusbar_new();

  gtk_box_pack_end(GTK_BOX(vbox), statusbar, FALSE, TRUE, 1);

  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), NULL);

  g_signal_connect(G_OBJECT(tog_stat), "activate",
G_CALLBACK(toggle_statusbar), statusbar);

  gtk_widget_show_all(window);

  gtk_main();

  return 0;
```

```
}
```

在我们的代码示例中，我们展示了如何去制造一个带有选择框的菜单选项。具体的功能是：如果选择框被选中则“状态栏”就会显示出来，反之则不会显示。

```
tog_stat = gtk_check_menu_item_new_with_label("View Statusbar");
```

函数 **gtk_check_menu_item_new_with_label()** 可以生成一个新的带有选择框的菜单选项。

```
if (gtk_check_menu_item_get_active(GTK_CHECK_MENU_ITEM(widget))) {

  gtk_widget_show(statusbar);

} else {gtk_widget_hide(statusbar); }
```

如果选择框被选中则“状态栏”就会显示出来，反之则不会显示。

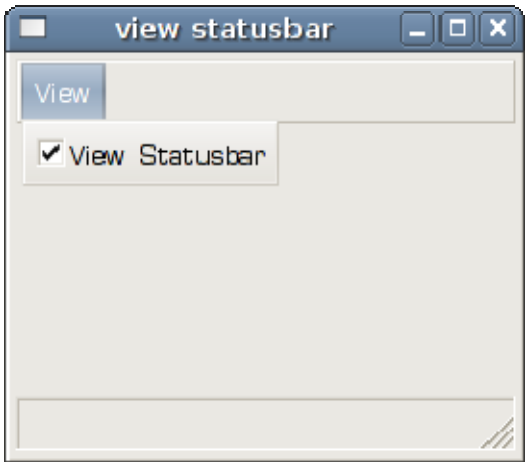

Figure: Check menu item

工具栏（A toolbar）

菜单栏为我们编程时实现某种功能提供了方便与快捷。在接下来的章节中，我们将为你展示一种在特定情况下可以更加便捷的的方法——制造一个“工具栏。”

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

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

  GtkWidget *window;
```

```
GtkWidget *vbox;

GtkWidget *toolbar;

GtkToolItem *new;

GtkToolItem *open;

GtkToolItem *save;

GtkToolItem *sep;

GtkToolItem *exit;

gtk_init(&argc, &argv);

window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

gtk_window_set_default_size(GTK_WINDOW(window), 250, 200);

gtk_window_set_title(GTK_WINDOW(window), "toolbar");

vbox = gtk_vbox_new(FALSE, 0);

gtk_container_add(GTK_CONTAINER(window), vbox);

toolbar = gtk_toolbar_new();

gtk_toolbar_set_style(GTK_TOOLBAR(toolbar), GTK_TOOLBAR_ICONS);

gtk_container_set_border_width(GTK_CONTAINER(toolbar), 2);

new = gtk_tool_button_new_from_stock(GTK_STOCK_NEW);

gtk_toolbar_insert(GTK_TOOLBAR(toolbar), new, -1);

open = gtk_tool_button_new_from_stock(GTK_STOCK_OPEN);
```

```
  gtk_toolbar_insert(GTK_TOOLBAR(toolbar), open, -1);

  save = gtk_tool_button_new_from_stock(GTK_STOCK_SAVE);

  gtk_toolbar_insert(GTK_TOOLBAR(toolbar), save, -1);

  sep = gtk_separator_tool_item_new();

  gtk_toolbar_insert(GTK_TOOLBAR(toolbar), sep, -1);

  exit = gtk_tool_button_new_from_stock(GTK_STOCK_QUIT);

  gtk_toolbar_insert(GTK_TOOLBAR(toolbar), exit, -1);

  gtk_box_pack_start(GTK_BOX(vbox), toolbar, FALSE, FALSE, 5);

  g_signal_connect(G_OBJECT(exit), "clicked",
G_CALLBACK(gtk_main_quit), NULL);

  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), NULL);

  gtk_widget_show_all(window);

  gtk_main();

  return 0;

}
```

以上的代码中，我们制作了一个简单的工具栏实现。

```
  toolbar = gtk_toolbar_new();

  gtk_toolbar_set_style(GTK_TOOLBAR(toolbar), GTK_TOOLBAR_ICONS)
```

从上面的两行代码中你应该可以看出来，我们生成了一个“崭新”的工具栏：）。我们还特地使他们都用图片来显示，没有包含文字。

```
 new = gtk_tool_button_new_from_stock(GTK_STOCK_NEW);
```

```
gtk_toolbar_insert(GTK_TOOLBAR(toolbar), new, -1);
```

从 stock 中我们生成了一个新的工具栏按钮。要想把工具栏按钮插入到工具栏中，很简单！只需要调用函数 **gtk_toolbar_insert()** 就可以搞定。

```
sep = gtk_separator_tool_item_new();

gtk_toolbar_insert(GTK_TOOLBAR(toolbar), sep, -1);
```

上面的代码，我们生成了一个分割线把工具栏按钮们分开（只是逻辑上的分组需要）。

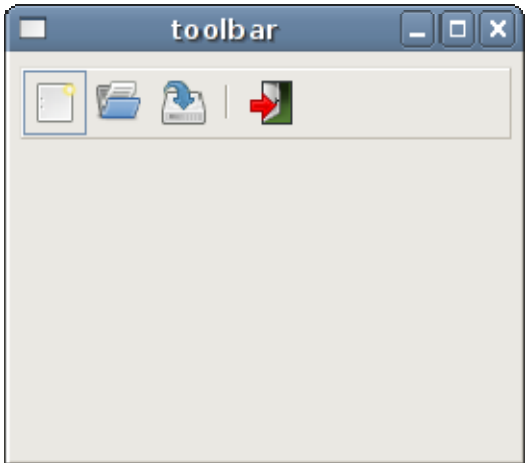

Figure: Toolbar

功能失效（Undo redo）

在接下来的例子中，我们将展示一个神奇的功能：使工具栏中的一个按钮功能失效（让他变成阴影）。这在 GUI 设计中是一个常见的技巧。 举个例子：当我们把一片文章单击保存后，那个保存按钮就会变成阴影状，也就是功能失效了。就是来提示你：保存功能已经执行过了，不需要再执行保存功能了。

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

#include <string.h>

void undo_redo(GtkWidget *widget,  gpointer item) {

  static int count = 2;

  const char *name = gtk_widget_get_name(widget);

  if ( strcmp(name, "undo") ) { count++;} else { count--;}
```

```
  if (count < 0) {
     gtk_widget_set_sensitive(widget, FALSE);
     gtk_widget_set_sensitive(item, TRUE);
  }
  if (count > 5) {
     gtk_widget_set_sensitive(widget, FALSE);
     gtk_widget_set_sensitive(item, TRUE);
  }
}
int main( int argc, char *argv[]){
  GtkWidget *window;
  GtkWidget *vbox;
  GtkWidget *toolbar;
  GtkToolItem *undo;
  GtkToolItem *redo;
  GtkToolItem *sep;
  GtkToolItem *exit;
  gtk_init(&argc, &argv);
  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
```

```
  gtk_window_set_default_size(GTK_WINDOW(window), 250, 200);

  gtk_window_set_title(GTK_WINDOW(window), "undoredo");

  vbox = gtk_vbox_new(FALSE, 0);

  gtk_container_add(GTK_CONTAINER(window), vbox);

  toolbar = gtk_toolbar_new();

  gtk_toolbar_set_style(GTK_TOOLBAR(toolbar), GTK_TOOLBAR_ICONS);

  gtk_container_set_border_width(GTK_CONTAINER(toolbar), 2);

  undo = gtk_tool_button_new_from_stock(GTK_STOCK_UNDO);

  gtk_widget_set_name(GTK_WIDGET(undo), "undo");

  gtk_toolbar_insert(GTK_TOOLBAR(toolbar), undo, -1);

  redo = gtk_tool_button_new_from_stock(GTK_STOCK_REDO);

  gtk_toolbar_insert(GTK_TOOLBAR(toolbar), redo, -1);

  sep = gtk_separator_tool_item_new();

  gtk_toolbar_insert(GTK_TOOLBAR(toolbar), sep, -1);

  exit = gtk_tool_button_new_from_stock(GTK_STOCK_QUIT);

  gtk_toolbar_insert(GTK_TOOLBAR(toolbar), exit, -1);

  gtk_box_pack_start(GTK_BOX(vbox), toolbar, FALSE, FALSE, 5);

  g_signal_connect(G_OBJECT(undo), "clicked", G_CALLBACK(undo_redo),
redo);

  g_signal_connect(G_OBJECT(redo), "clicked", G_CALLBACK(undo_redo),
undo);
```

```
  g_signal_connect(G_OBJECT(exit), "clicked",
G_CALLBACK(gtk_main_quit), NULL);

  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), NULL);

  gtk_widget_show_all(window);

  gtk_main();

  return 0;

}
```

我们的例子中用到了 GTK+ stock 来实现是 “功能有效”还是“功能失效”。当你单击几下 按钮后，那个按钮就会变成阴影状，也就是说他从“功能有效”变成了“功能失效”。

```
if (count < 0) {

   gtk_widget_set_sensitive(widget, FALSE);

   gtk_widget_set_sensitive(item, TRUE);

}

if (count > 5) {

   gtk_widget_set_sensitive(widget, FALSE);

   gtk_widget_set_sensitive(item, TRUE);

}
```

gtk_widget_set_sensitive() 是被用来告诉计算机是否要击活一个工具栏按钮。

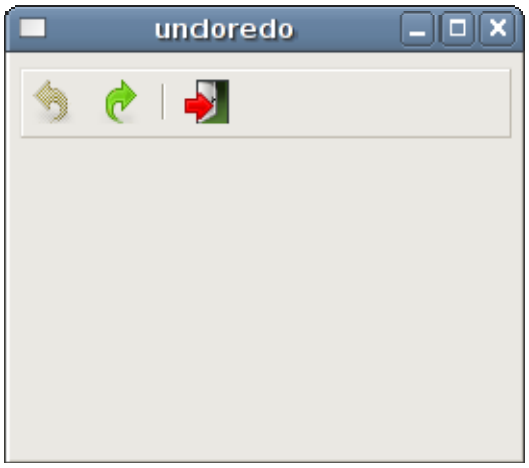

Figure: Undo redo

GTK+ 布局管理

在本章中，我们将讲述如何将构件布置在窗口与对话框中。

当我们在设计应用程序的图形界面时，我们首先要决定的是在程序中用到哪种构件和管理应用程序中的这些构件。为了方便管理我们的构件，在 GTK+通常使用不可见的构件称作 **layout containers**. 。在本章节中，我们将设计其中的
—— **GtkAlignment**, **GtkFixed**, **GtkVBox** 和 **GtkTable**.

GtkFixed

容器构件 **GtkFixed** 用于布置子构件在一个固定的位置和设定固定的大小。这种构件并不是属于自动的布局关系器。实质上，在我们设计的大多数应用程序中，我并不使用 GtkFixed；而在只用于一些比较特殊的场合。例如，游戏，含有绘图功能的专用软件，那些需要移动和调整大小的软件（正如电子表格中的图表）以及那些小型的教育用途软件。

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

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

  GtkWidget *window;

  GtkWidget *fixed;

  GtkWidget *button1;

  GtkWidget *button2;
```

```
    GtkWidget *button3;

    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), 290, 200);
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
    fixed = gtk_fixed_new();
    gtk_container_add(GTK_CONTAINER(window), fixed);
    button1 = gtk_button_new_with_label("Button");
    gtk_fixed_put(GTK_FIXED(fixed), button1, 150, 50);
    gtk_widget_set_size_request(button1, 80, 35);
    button2 = gtk_button_new_with_label("Button");
    gtk_fixed_put(GTK_FIXED(fixed), button2, 15, 15);
    gtk_widget_set_size_request(button2, 80, 35);
    button3 = gtk_button_new_with_label("Button");
    gtk_fixed_put(GTK_FIXED(fixed), button3, 100, 100);
    gtk_widget_set_size_request(button3, 80, 35);
    g_signal_connect_swapped(G_OBJECT(window), "destroy",
G_CALLBACK(gtk_main_quit), NULL);
    gtk_widget_show_all(window);
```

```
    gtk_main();

    return 0;

}
```

在我上面的这个例子中，我用代码生成了三个按钮构件然后把他们布局在固定的坐标上。当我们如果试图去改变窗口的大小的时候，其中按钮将会保持他们的大小和之前的坐标。

```
fixed = gtk_fixed_new();
```

上面的代码就可以生成了一个 **GtkFixed** 的容器构件。

```
gtk_fixed_put(GTK_FIXED(fixed), button1, 150, 50);
```

第一个按钮就 **gtk_fixed_put()**函数来进行布局，坐标为 x=150, y=50.

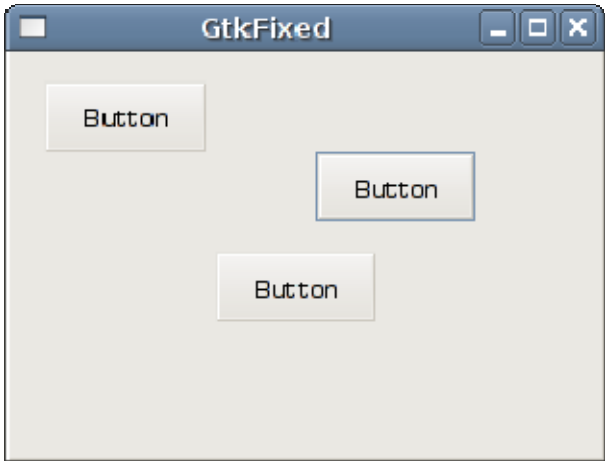

Figure: GtkFixed container

GtkVBox

GtkVBox 是一种用于垂直布局的容器型构件。他把放置在他中的子构件放置在一个单独的列中。类似的是 **GtkHBox** 也有相似的功能，有区别的在于他是用于水平布局，他的子构件是布置在一个单独的行中的。

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

int main( int argc, char *argv[]){
```

```
    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *settings;

    GtkWidget *accounts;

    GtkWidget *loans;
    GtkWidget *cash;
    GtkWidget *debts;
    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
    gtk_window_set_default_size(GTK_WINDOW(window), 230, 250);
    gtk_window_set_title(GTK_WINDOW(window), "GtkVBox");
    gtk_container_set_border_width(GTK_CONTAINER(window), 5);
    vbox = gtk_vbox_new(TRUE, 1);
    gtk_container_add(GTK_CONTAINER(window), vbox);
    settings = gtk_button_new_with_label("Settings");
    accounts = gtk_button_new_with_label("Accounts");
    loans = gtk_button_new_with_label("Loans");
    cash = gtk_button_new_with_label("Cash");
```

```
  debts = gtk_button_new_with_label("Debts");

  gtk_box_pack_start(GTK_BOX(vbox), settings, TRUE, TRUE, 0);

  gtk_box_pack_start(GTK_BOX(vbox), accounts, TRUE, TRUE, 0);

  gtk_box_pack_start(GTK_BOX(vbox), loans, TRUE, TRUE, 0);

  gtk_box_pack_start(GTK_BOX(vbox), cash, TRUE, TRUE, 0);

  gtk_box_pack_start(GTK_BOX(vbox), debts, TRUE, TRUE, 0);

  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), G_OBJECT(window));

  gtk_widget_show_all(window);

  gtk_main();

  return 0;

}
```

上面的这个按钮就显示了 **GtkVBox** 的作用。他把五个按钮都布局在同一列上。如果你改变程序窗口的大小，其中的子构件（如按钮 button）也会改变大小。

```
vbox = gtk_vbox_new(TRUE, 1);
```

上面程序中生成了 **GtkVBox**。 我们把其中的第一个参数设置为 TRUE。这就意味着，程序中我的按钮都为同样的大小。至于按钮之间的距离大小被设置为“1”象素。

```
gtk_box_pack_start(GTK_BOX(vbox), settings, TRUE, TRUE, 0);
```

在上面的程序中我们把“settings “按钮布局在 vbox 容器构件中。 至于函数实参中的前两个参数，分别是容器构件和我们要放置的子构件。接下来的三个参数中分别是 expand, fill 和 padding。 值得注意的是如果 fill 对应的参数是 FALSE,则按钮就不会充满整个 vbox 构件。比较类似的是，如果之前在 gtk_vbox_new(TRUE, 1);已经设置按钮都是等宽高了，所以 expand 对应的参数，是完全没有效果的。（译者注：此处建议 fill expand 都设为 TRUE,至于具体区别和含义可在编程时感受）

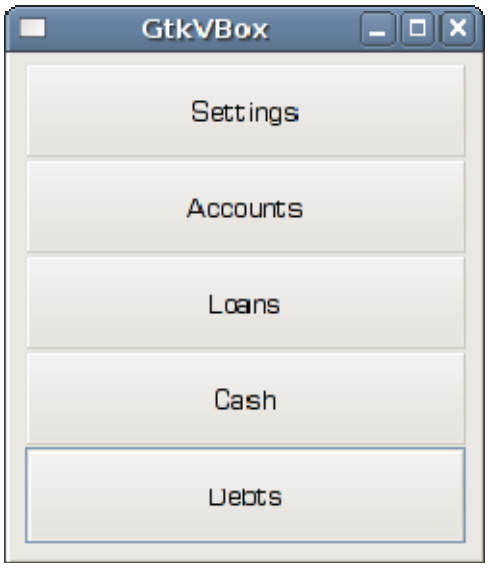

Figure: GtkVBox container

GtkTable

GtkTable 布局构件即可以按照行也可以按照列来布局她的子构件。

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

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

  GtkWidget *window;

  GtkWidget *table;

  GtkWidget *button;

  char *values[16] = { "7", "8", "9", "/", "4", "5", "6", "*", "1", "2",
"3", "-", "0", ".", "=", "+" };

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_window_set_default_size(GTK_WINDOW(window), 250, 180);
```

```
  gtk_window_set_title(GTK_WINDOW(window), "GtkTable");

  gtk_container_set_border_width(GTK_CONTAINER(window), 5);

  table = gtk_table_new(4, 4, TRUE);

  gtk_table_set_row_spacings(GTK_TABLE(table), 2);

  gtk_table_set_col_spacings(GTK_TABLE(table), 2);

  int i = 0;

  int j = 0;

  int pos = 0;

  for( i=0; i < 4; i++) {

    for( j=0; j < 4; j++) {

      button = gtk_button_new_with_label(values[pos]);

      gtk_table_attach_defaults(GTK_TABLE(table), button, j, j+1, i,
i+1 );

      pos++;

    }

  }

  gtk_container_add(GTK_CONTAINER(window), table);

  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), G_OBJECT(window));

  gtk_widget_show_all(window);

  gtk_main();
```

```
  return 0;

}
```

在以上的例子中，我们将仿照计算器编写一系列按钮。

```
table = gtk_table_new(4, 4, TRUE);
```

上面我们生成了一个新的 **GtkTable** 布局构件，并设置为 4 行与 4 列。

```
gtk_table_set_row_spacings(GTK_TABLE(table), 2);

gtk_table_set_col_spacings(GTK_TABLE(table), 2);
```

上面我们就设置了每行与每列的距离。

```
for( i=0; i < 4; i++) {

  for( j=0; j < 4; j++) {

    button = gtk_button_new_with_label(values[pos]);

    gtk_table_attach_defaults(GTK_TABLE(table), button, j, j+1, i,
i+1 );

    pos++;

  }

}
```

以上代码将生成 16 个按钮并把他们布局在 GtkTable 容器构件中。

Figure: GtkTable container

GtkAlignment

GtkAlignment 容器构件控制了她的子构件的对齐方式与大小。

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

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

  GtkWidget *window;

  GtkWidget *ok;

  GtkWidget *close;

  GtkWidget *vbox;

  GtkWidget *hbox;

  GtkWidget *halign;

  GtkWidget *valign;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
```

```
    gtk_window_set_default_size(GTK_WINDOW(window), 350, 200);

    gtk_window_set_title(GTK_WINDOW(window), "GtkAlignment");

    gtk_container_set_border_width(GTK_CONTAINER(window), 10);

    vbox = gtk_vbox_new(FALSE, 5);

    valign = gtk_alignment_new(0, 1, 0, 0);

    gtk_container_add(GTK_CONTAINER(vbox), valign);

    gtk_container_add(GTK_CONTAINER(window), vbox);

    hbox = gtk_hbox_new(TRUE, 3);

    ok = gtk_button_new_with_label("OK");

    gtk_widget_set_size_request(ok, 70, 30);

    gtk_container_add(GTK_CONTAINER(hbox), ok);

    close = gtk_button_new_with_label("Close");

    gtk_container_add(GTK_CONTAINER(hbox), close);

    halign = gtk_alignment_new(1, 0, 0, 0);

    gtk_container_add(GTK_CONTAINER(halign), hbox);

    gtk_box_pack_start(GTK_BOX(vbox), halign, FALSE, FALSE, 0);

    g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), G_OBJECT(window));

    gtk_widget_show_all(window);

    gtk_main();
```

```
    return 0;

}
```

在上面的例子中，我们把两个按钮布局在了一个窗口的右下角。为了实现这个效果，我们用一个水平盒子构件 horizontal box 、一个竖直盒子构件 vertical box 和两个对齐容器构件（alignment containers）。

```
valign = gtk_alignment_new(0, 1, 0, 0);
```

上面的代码中我们生成了一个对齐容器构件。

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

然后我们把对齐容器构件布局在水平盒子中（vbox）。

```
hbox = gtk_hbox_new(TRUE, 3);

ok = gtk_button_new_with_label("OK");

gtk_widget_set_size_request(ok, 70, 30);

gtk_container_add(GTK_CONTAINER(hbox), ok);

close = gtk_button_new_with_label("Close");

gtk_container_add(GTK_CONTAINER(hbox), close);
```

上面代码中，我们生成了一个水平盒子（ horizontal box） 然后把两个按钮布局在其中。

```
halign = gtk_alignment_new(1, 0, 0, 0);

gtk_container_add(GTK_CONTAINER(halign), hbox);

gtk_box_pack_start(GTK_BOX(vbox), halign, FALSE, FALSE, 0);
```

上面的代码中将生成一个对齐容器构件然后把布局在她中的子构件布局在右边。我们把水平盒子（ horizontal box）添加到对齐容器构件中，然后又把对齐容器构件添加到竖直盒子中（vertical box）。 最后，我要振臂高呼一下，：）对齐容器构件（ alignment container ）中只能放置一个子构件，这就是为什么我们要用到那么多盒子来帮助我们布局那两个按钮了。

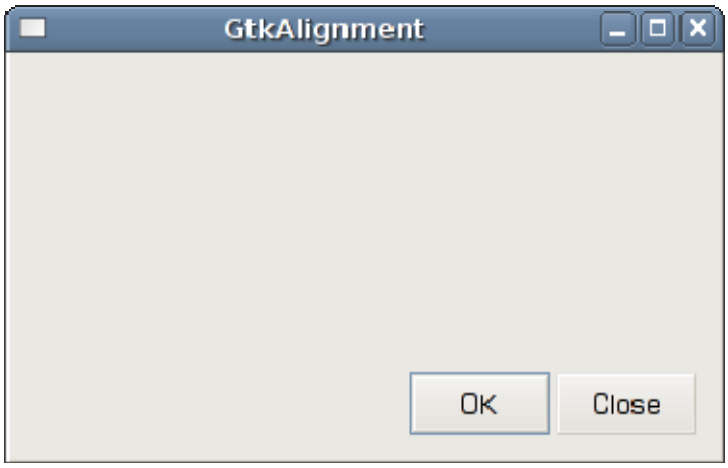

Figure: GtkAlignment container

Windows

接下来我们将展示一个更加高级一点的例子。具体就是展示一个窗口，你可以在 JDeveloper IDE（一种 java 的集成开发软件）中发现这个例子的身影。

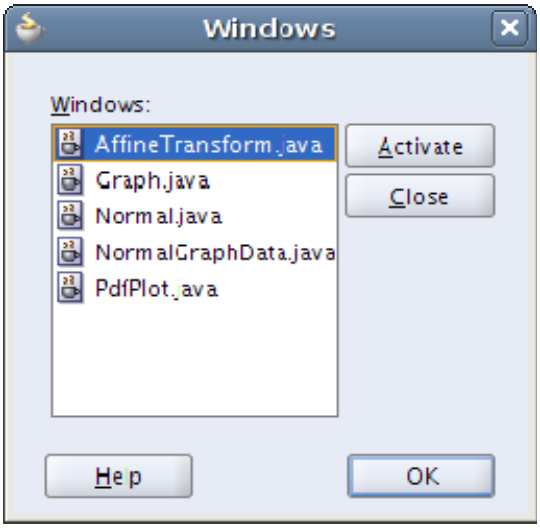

Figure: Windows dialog in JDeveloper

The dialog shows all opened windows, or more precisely tabs in JDeveloper application.

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

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

  GtkWidget *window;
```

```
    GtkWidget *table;

    GtkWidget *title;

    GtkWidget *activate;
    GtkWidget *halign;
    GtkWidget *halign2;
    GtkWidget *valign;
    GtkWidget *close;
    GtkWidget *wins;
    GtkWidget *help;
    GtkWidget *ok;
    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
    gtk_widget_set_size_request (window, 300, 250);
    gtk_window_set_resizable(GTK_WINDOW(window), FALSE);
    gtk_window_set_title(GTK_WINDOW(window), "Windows");
    gtk_container_set_border_width(GTK_CONTAINER(window), 15);
    table = gtk_table_new(8, 4, FALSE);
    gtk_table_set_col_spacings(GTK_TABLE(table), 3);
```

```
  title = gtk_label_new("Windows");

  halign = gtk_alignment_new(0, 0, 0, 0);
```

```
  gtk_container_add(GTK_CONTAINER(halign), title);
```

```
  gtk_table_attach(GTK_TABLE(table), halign, 0, 1, 0, 1, GTK_FILL,
GTK_FILL, 0, 0);

  wins = gtk_text_view_new();

  gtk_text_view_set_editable(GTK_TEXT_VIEW(wins), FALSE);

  gtk_text_view_set_cursor_visible(GTK_TEXT_VIEW(wins), FALSE);

  gtk_table_attach(GTK_TABLE(table), wins, 0, 2, 1, 3, GTK_FILL |
GTK_EXPAND, GTK_FILL | GTK_EXPAND, 1, 1);

  activate = gtk_button_new_with_label("Activate");

  gtk_widget_set_size_request(activate, 50, 30);

  gtk_table_attach(GTK_TABLE(table), activate, 3, 4, 1, 2,
GTK_FILL, GTK_SHRINK, 1, 1);

  valign = gtk_alignment_new(0, 0, 0, 0);

  close = gtk_button_new_with_label("Close");

  gtk_widget_set_size_request(close, 70, 30);

  gtk_container_add(GTK_CONTAINER(valign), close);

  gtk_table_set_row_spacing(GTK_TABLE(table), 1, 3);

  gtk_table_attach(GTK_TABLE(table), valign, 3, 4, 2, 3,
GTK_FILL, GTK_FILL | GTK_EXPAND, 1, 1);

  halign2 = gtk_alignment_new(0, 1, 0, 0);
```

```
  help = gtk_button_new_with_label("Help");

  gtk_container_add(GTK_CONTAINER(halign2), help);
```

```
  gtk_widget_set_size_request(help, 70, 30);
```

```
  gtk_table_set_row_spacing(GTK_TABLE(table), 3, 6);

  gtk_table_attach(GTK_TABLE(table), halign2, 0, 1, 4, 5,
GTK_FILL, GTK_FILL, 0, 0);

  ok = gtk_button_new_with_label("OK");

  gtk_widget_set_size_request(ok, 70, 30);

  gtk_table_attach(GTK_TABLE(table), ok, 3, 4, 4, 5, GTK_FILL, GTK_FILL,
0, 0);

  gtk_container_add(GTK_CONTAINER(window), table);

  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), G_OBJECT(window));

  gtk_widget_show_all(window);

  gtk_main();

  return 0;

}
```

以上代码将生成一个简单的 GTK+ 窗口。

```
 table = gtk_table_new(8, 4, FALSE);
```

我们使用 table 表格容器构件来进行布局。

```
 title = gtk_label_new("Windows");
```

```
 halign = gtk_alignment_new(0, 0, 0, 0);

 gtk_container_add(GTK_CONTAINER(halign), title);

 gtk_table_attach(GTK_TABLE(table), halign, 0, 1, 0, 1, GTK_FILL,
GTK_FILL, 0, 0);
```

上面的代码生成了一个标签，设为居左。这个标签被布局在 **GtkTable** 构件容器的第一列。

```
 wins = gtk_text_view_new();

 gtk_text_view_set_editable(GTK_TEXT_VIEW(wins), FALSE);

 gtk_text_view_set_cursor_visible(GTK_TEXT_VIEW(wins), FALSE);

 gtk_table_attach(GTK_TABLE(table), wins, 0, 2, 1, 3, GTK_FILL |
GTK_EXPAND, GTK_FILL | GTK_EXPAND, 1, 1);
```

文本显示构件占据了两行和两列。我们把该文本编辑构件的属性设置为 editable 和光标隐藏（hide the cursor）。

```
 valign = gtk_alignment_new(0, 0, 0, 0);

 close = gtk_button_new_with_label("Close");

 gtk_widget_set_size_request(close, 70, 30);

 gtk_container_add(GTK_CONTAINER(valign), close);

 gtk_table_set_row_spacing(GTK_TABLE(table), 1, 3);

 gtk_table_attach(GTK_TABLE(table), valign, 3, 4, 2, 3, GTK_FILL,
GTK_FILL | GTK_EXPAND, 1, 1);
```

我们把两个靠在一起的按钮布局在文本编辑构件的左边也就是第四行（我们是从 0 开始记数的），我们把这两个按钮布局在“对齐构件”（alignment widget）中，这样我们就可以把他俩布局在顶部了。

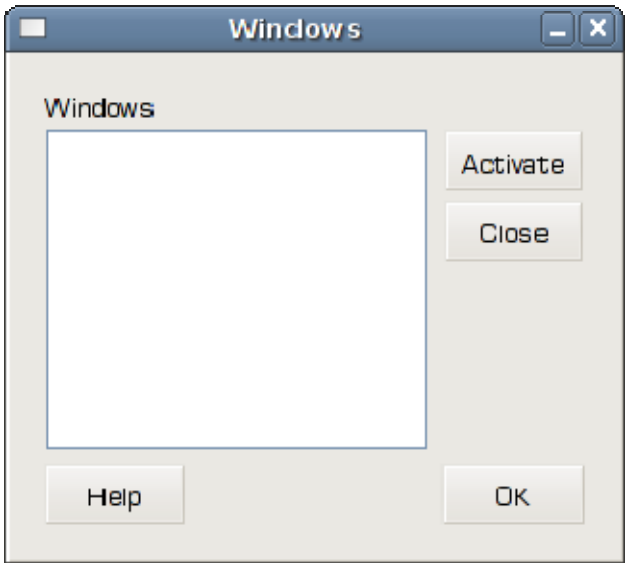

Figure: Windows

GTK+ 中的事件（events）和信号（signals）

在我们教程的这一个章节中，我们会谈一谈 GTK+函数工具库中的，“事件”系统 。

GTK+函数工具库是基于 **“事件”**系统的。所有的 GUI 应用程序无一例外都是基于“事件”驱动的。假如没有“事件”发生，则应用程序就什么都不会做。在 GTK+中一个事件就是从 X 窗口服务器传出来的一个消息。当一个“事件”发生时，他就会通过发送一个“信号”来表示他已经做出了反应。利用 GTK+还可以为“信号”绑定专门的回调函数。也就是说回调函数只对他特定的“信号”才有反应并执行。

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

void button_clicked(GtkWidget *widget, gpointer data){

  g_print("clicked\n");

}

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

  GtkWidget *window;

  GtkWidget *fixed;

  GtkWidget *button;
```

```
    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

    gtk_window_set_title(GTK_WINDOW(window), "GtkButton");

    gtk_window_set_default_size(GTK_WINDOW(window), 230, 150);

    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

    fixed = gtk_fixed_new();

    gtk_container_add(GTK_CONTAINER(window), fixed);

    button = gtk_button_new_with_label("Click");

    gtk_fixed_put(GTK_FIXED(fixed), button, 50, 50);

    gtk_widget_set_size_request(button, 80, 35);

    g_signal_connect(G_OBJECT(button), "clicked",
G_CALLBACK(button_clicked), NULL);

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

    gtk_widget_show_all(window);

    gtk_main();

    return 0;

}
```

在我们上面的这个示例应用程序中，我们有两个“信号”。一个是信号 **clicked**,也就是单击；另外则是信号 **destroy**。

```
 g_signal_connect(G_OBJECT(button),
"clicked",G_CALLBACK(button_clicked), NULL);
```

我们用函数 **g_signal_connect()** 去“连接”信号“clicked”和回调函 **button_clicked()** 。

```
 void button_clicked(GtkWidget *widget, gpointer data)
{   g_print("clicked\n"); }
```

这个回调函数执行的功能是向终端输出“clicked”字符串。这个函数的第一行参就是那个发射信号的对象。在我们的这个例子中实参便是构件“Click button”。第二个行参是可以选择有无的。我们可以利用这个参数向回调函数传递特定的数据。在我们的例子中，并没有传递任何的参数。所以我们就在 g_signal_connect()中调用回调函数的时候在调用的第二个实参中填上了“NULL”。

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

如果我们单击窗口右上角的“X”或者按 Atl + F4，一个 **destroy** 就被发射出去了。然后呢，我们为这个信号所绑定的回调函数 **gtk_main_quit()** 将执行，他的功能是终止整个应用程序。

移动窗口（Moving window）

在下一个例子中，我们将展示是如何对“移动窗口”这个事件做出反应的。

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

void frame_callback(GtkWindow *window, GdkEvent *event, gpointer data){

   int x, y;

   char buf[10];

   x = event->configure.x;

   y = event->configure.y;

   sprintf(buf, "%d, %d", x, y);

   gtk_window_set_title(window, buf);

}
```

```
int main(int argc, char *argv[]){

  GtkWidget *window;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_window_set_default_size(GTK_WINDOW(window), 230, 150);

  gtk_window_set_title(GTK_WINDOW(window), "Simple");

  gtk_widget_add_events(GTK_WIDGET(window), GDK_CONFIGURE);

  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), G_OBJECT(window));

  g_signal_connect(G_OBJECT(window),
"configure-event",G_CALLBACK(frame_callback), NULL);

  gtk_widget_show(window);

  gtk_main();

  return 0;

}
```

在这个例子中，我们随时追踪显示了位于左上角的标题栏的位置。

```
gtk_widget_add_events(GTK_WIDGET(window), GDK_CONFIGURE);
```

上面这行代码说明了，哪一个构件将回接受到事件，并对事件的发生做出反应。一些事件使用时要已与特定的构件组装好了，另外的一些事件不得不要用一个函数 **gtk_widget_add_events()** 去武装他。把事件类型 **GDK_CONFIGURE** 填加到这个函数中。事件类型 **GDK_CONFIGURE** 包含了所有的大小、位置和用于存储事件次序的栈结构。

```
 g_signal_connect(G_OBJECT(window),
"configure-event",G_CALLBACK(frame_callback), NULL);
```

从上面可以看出，信号“ **configure-event**“被发射了 ，则所绑定构件的大小、位置与次序栈都被捕获了。

```
void frame_callback(GtkWindow *window,GdkEvent *event, gpointer data) {

   int x, y;

   char buf[10];

   x = event->configure.x;

   y = event->configure.y;

   sprintf(buf, "%d, %d", x, y);

   gtk_window_set_title(window, buf);

 }
```

这个回调函数有三个行参。分别是：反射信号的构件， **GdkEvent** 和可选择的行参.。我们获取了位置坐标（x,y），并把他放在了标题栏上。

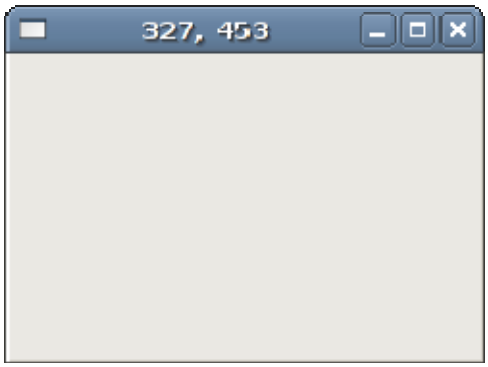

Figure: Move event

鼠标进入信号（The enter signal）

在接下来的章节中，我们将展示如何对“**鼠标的进入**”信号作出反应。当我们的鼠标移动到我们所绑定的那个构件上时就会发出“鼠标进入信号”。

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

```
void enter_button(GtkWidget *widget, gpointer data) {
  GdkColor color;
  color.red = 27000;
  color.green = 30325;

  color.blue = 34181;

  gtk_widget_modify_bg(widget, GTK_STATE_PRELIGHT, &color);
}
int main( int argc, char *argv[]){
  GtkWidget *window;
  GtkWidget *fixed;
  GtkWidget *button;
  gtk_init(&argc, &argv);
  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
  gtk_window_set_default_size(GTK_WINDOW(window), 230, 150);
  gtk_window_set_title(GTK_WINDOW(window), "enter signal");
  fixed = gtk_fixed_new();
  gtk_container_add(GTK_CONTAINER(window), fixed);
  button = gtk_button_new_with_label("Button");
```

```
    gtk_widget_set_size_request(button, 80, 35);

    gtk_fixed_put(GTK_FIXED(fixed), button, 50, 50);

    g_signal_connect(G_OBJECT(button),
"enter",G_CALLBACK(enter_button), NULL);

    g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), NULL);
```

```
    gtk_widget_show_all(window);
```

```
    gtk_main();

    return 0;

}
```

上面的程序想实现这样的一个功能：当我们的鼠标掠过按钮构件的时候,我们的程序代码可以改变那个按钮的背景颜色。

```
 g_signal_connect(G_OBJECT(button), "enter", G_CALLBACK(enter_button),
NULL);
```

当信号“**enter**”发生时，我们会调用函数 **enter_button()**

```
 GdkColor color;

 color.red = 27000;

 color.green = 30325;

 color.blue = 34181;

 gtk_widget_modify_bg(widget, GTK_STATE_PRELIGHT, &color);
```

在对应的回调函数中，我们通过调用函数 **gtk_widget_modify_bg()**来改变按钮的颜色。

回调函数解除绑定（Disconnecting a callback）

既然可以为一个信号绑定一个回调函数，我们当然也可以解除一个绑定。在接下来的代码示范示例中就是这样的一个例子。

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

int handler_id;

void button_clicked(GtkWidget *widget, gpointer data)
{ g_print("clicked\n");}
```

```
void toogle_signal(GtkWidget *widget, gpointer window){
```

```
  if (gtk_toggle_button_get_active(GTK_TOGGLE_BUTTON(widget))) {

     handler_id = g_signal_connect(G_OBJECT(window),
"clicked",G_CALLBACK(button_clicked), NULL);

  } else { g_signal_handler_disconnect(window, handler_id);}

}

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

  GtkWidget *window;

  GtkWidget *fixed;

  GtkWidget *button;

  GtkWidget *check;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_window_set_default_size(GTK_WINDOW(window), 250, 150);
```

```
  gtk_window_set_title(GTK_WINDOW(window), "Disconnect");

  fixed = gtk_fixed_new();

  gtk_container_add(GTK_CONTAINER(window), fixed);

  button = gtk_button_new_with_label("Click");

  gtk_widget_set_size_request(button, 80, 30);

  gtk_fixed_put(GTK_FIXED(fixed), button, 30, 50);
```

```
  check = gtk_check_button_new_with_label("Connect");
```

```
  gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(check), TRUE);

  gtk_fixed_put(GTK_FIXED(fixed), check, 130, 50);

  handler_id = g_signal_connect(G_OBJECT(button),
"clicked",G_CALLBACK(button_clicked), NULL);

  g_signal_connect(G_OBJECT(check),
"clicked",G_CALLBACK(toogle_signal), (gpointer) button);

  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), NULL);

  gtk_widget_show_all(window);

  gtk_main();

  return 0;

}
```

在例子中，我们生成了一个按钮和一个选择框。那个选择框的功能便是绑定或者解除绑定一个回调函数与信号“clicked”之间的关系。

```
 handler_id = g_signal_connect(G_OBJECT(button),
"clicked",G_CALLBACK(button_clicked), NULL);
```

g_signal_connect()函数执行后会返回一个“id”数据。这就实现了对回调函数的唯一标示。

```
 if (gtk_toggle_button_get_active(GTK_TOGGLE_BUTTON(widget))) {

    handler_id = g_signal_connect(G_OBJECT(window),
"clicked",G_CALLBACK(button_clicked), NULL);

 } else { g_signal_handler_disconnect(window, handler_id); }
```

这段代码决定了选择框的状态，如果选中了，就绑定否则就解除绑定。

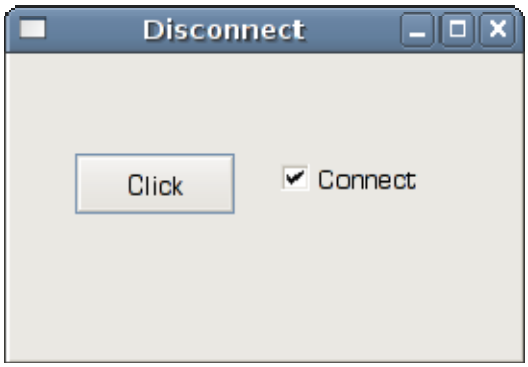

Figure: Disconnect

（拖与放的示例）Drag and Drop example

在接下一个示例中，我们将展示一个有趣的特性。我们将生成一个没有边框的窗口，然后我们将说明如何才能拖动和放置这样的一个窗口。

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

gboolean on_button_press (GtkWidget* widget,GdkEventButton * event,
GdkWindowEdge edge){

  if (event->type == GDK_BUTTON_PRESS) {

    if (event->button == 1)
{ gtk_window_begin_move_drag(GTK_WINDOW(gtk_widget_get_toplevel(widge
t)),event->button,event->x_root,event->y_root,event->time); }

  return FALSE;
```

```
}

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

  GtkWidget *window;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_window_set_default_size(GTK_WINDOW(window), 230, 150);

  gtk_window_set_title(GTK_WINDOW(window), "Drag & drop");

  gtk_window_set_decorated(GTK_WINDOW (window), FALSE);

  gtk_widget_add_events(window, GDK_BUTTON_PRESS_MASK);

  g_signal_connect(G_OBJECT(window),
"button-press-event",G_CALLBACK(on_button_press), NULL);

  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), G_OBJECT(window));

  gtk_widget_show(window);

  gtk_main();

  return 0;

}
```

上面的这个示例程序向我们展示了如何才能生成一个可以拖拉和放置的无边框的窗口程序。

```
gtk_window_set_decorated(GTK_WINDOW (window), FALSE);
```

我们去除了窗口中的一些修饰性的部分。也就是说我们的这个窗口没有边框和标题栏。

```
  g_signal_connect(G_OBJECT(window),
"button-press-event",G_CALLBACK(on_button_press), NULL);
```

我们把信号 **button-press-event** 绑定在窗口中。

```
gboolean on_button_press (GtkWidget* widget,GdkEventButton* event,
GdkWindowEdge edge){

  if (event->type == GDK_BUTTON_PRESS)  {

    if (event->button == 1)
{ gtk_window_begin_move_drag(GTK_WINDOW(gtk_widget_get_toplevel(widge
t)),event->button,event->x_root,event->y_root,event->time); }

  }

  return FALSE;

}
```

在回调函数 **on_button_press()**中，我们放置了拖动和放置的代码。 我们检测是否鼠标被单击（左击）。然后条件判断执行函数 **gtk_window_begin_move_drag()** 进行拖放操作。

（一个定时器） A timer example

在接下来的示例中我们将向你展示如何去生成一个定时器。定时器通常应用于当我们要做一些重复工作的场合。譬如一个时钟，一个倒记时和增加动态的视觉效应。

```
#include <cairo.h>

#include <gtk/gtk.h>

#include <time.h>

static char buffer[256];

static gboolean

on_expose_event(GtkWidget *widget,GdkEventExpose *event,gpointer
data){
```

```
  cairo_t *cr;

  cr = gdk_cairo_create(widget->window);

  cairo_move_to(cr, 30, 30);

  cairo_show_text(cr, buffer);

  cairo_destroy(cr);

  return FALSE;

}
```

```
static gboolean
```

```
time_handler(GtkWidget *widget){

  if (widget->window == NULL) return FALSE;

  time_t curtime;

  struct tm *loctime;

  curtime = time(NULL);

  loctime = localtime(&curtime);

  strftime(buffer, 256, "%T", loctime);

  gtk_widget_queue_draw(widget);

  return TRUE;

}

int main (int argc, char *argv[]){
```

```
  GtkWidget *window;

  GtkWidget *darea;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  darea = gtk_drawing_area_new();

  gtk_container_add(GTK_CONTAINER (window), darea);

  g_signal_connect(darea, "expose-event",G_CALLBACK(on_expose_event),
NULL);

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

```
  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
```

```
  gtk_window_set_default_size(GTK_WINDOW(window), 170, 100);

  gtk_window_set_title(GTK_WINDOW(window), "timer");

  g_timeout_add(1000, (GSourceFunc) time_handler, (gpointer) window);

  gtk_widget_show_all(window);

  time_handler(window);

  gtk_main();

  return 0;

}
```

我们将要在窗口中，“画”一个显示当前时间的动态效果。所用的工具，就是在我们一开始曾经提到过的 Cairo 函数工具库。

```
g_signal_connect(darea,
"expose-event",G_CALLBACK(on_expose_event),NULL);
```

首先我们要把在回调函数 **on_expose_event()**中画出时间的数值。这个回调函数与信号进行了绑定 **expose-event** 。如果这个信号发射出去了，这个窗口就会按照我们程序所安排的一样——“立即刷新”。

```
g_timeout_add(1000, (GSourceFunc) time_handler, (gpointer) window);
```

上面的这个函数注册了一个 GTK+系统中的一个定时器（是抽象的不可见的）。函数 **time_handler()** 就会被按照我们在函数中的设置在设定的时间内不断的运行。在例子中，我们设定了这个时间为 1 秒，一旦这个函数 **g_timeout_add()**返回 FALSE，定时器函数 **time_handler()**就会运行。

```
time_handler(window);
```

这行代码的作用是立即启动定时器，否则的话，你会看到 1 秒的延迟。

```
cairo_t *cr;

cr = gdk_cairo_create(widget->window);

cairo_move_to(cr, 30, 30);

cairo_show_text(cr, buffer);

cairo_destroy(cr);
```

以上便是用来在当前窗口上画出当前时间的代码。如果你想了解更多的关于 Cairo 函数库的一些情况请访问 .

```
if (widget->window == NULL) return FALSE;
```

在我们刷新窗口前，很显然要先注销窗口，这样的话，定时函数便被自动调用了；可当我们关闭窗口时，定时器也会被调用，这是我们不想看到了，所以需要加上以上的代码，阻止在已经关闭的构件上运行定时器。

```
time_t curtime;

struct tm *loctime;

curtime = time(NULL);

loctime = localtime(&curtime);
```

```
strftime(buffer, 256, "%T", loctime);
```

上面的代码获取了系统当前的时间。

```
gtk_widget_queue_draw(widget);
```

这段代码会注销窗口，然后信号 **expose-event** 就会被激活发射出去。从而被绑定的回调函数就会执行。

GTK+系统中的对话框（GTK+ dialogs）

在接下来的章节中我们将着重介绍 GTK+ 系统中的对话框。

对话框窗口是众多 GUI 应用程序中不可或缺的部分。对话框经常是很多人进行信息交流的桥梁。在计算机中，对话框也经常扮演着我们和应用程序进行对话的工具。对话框可以用来输入数据，修改数据，或者改变应用程序的使用设置信息。对话框是一个人机交互的重要手段。

消息对话框（Message dialogs）

消息对话框可以方便的在你的应用程序中，跳出来显示一些有用的信息。当然可以包含文字或者图象。

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

void show_info(GtkWidget *widget, gpointer window){

  GtkWidget *dialog;

  dialog = gtk_message_dialog_new(window,

            GTK_DIALOG_DESTROY_WITH_PARENT,

            GTK_MESSAGE_INFO,

            GTK_BUTTONS_OK,

            "Download Completed", "title");

  gtk_window_set_title(GTK_WINDOW(dialog), "Information");
```

```
  gtk_dialog_run(GTK_DIALOG(dialog));
  gtk_widget_destroy(dialog);
}
void show_error(GtkWidget *widget, gpointer window){
  GtkWidget *dialog;
  dialog = gtk_message_dialog_new(window,
            GTK_DIALOG_DESTROY_WITH_PARENT,
            GTK_MESSAGE_ERROR,
            GTK_BUTTONS_OK,
            "Error loading file");
  gtk_window_set_title(GTK_WINDOW(dialog), "Error");
  gtk_dialog_run(GTK_DIALOG(dialog));
  gtk_widget_destroy(dialog);
}
void show_question(GtkWidget *widget, gpointer window){
  GtkWidget *dialog;
  dialog = gtk_message_dialog_new(window,
            GTK_DIALOG_DESTROY_WITH_PARENT,
            GTK_MESSAGE_QUESTION,
```

```
            GTK_BUTTONS_YES_NO,
            "Are you sure to quit?");
  gtk_window_set_title(GTK_WINDOW(dialog), "Question");
  gtk_dialog_run(GTK_DIALOG(dialog));
  gtk_widget_destroy(dialog);
}
void show_warning(GtkWidget *widget, gpointer window){
  GtkWidget *dialog;
  dialog = gtk_message_dialog_new(window,
            GTK_DIALOG_DESTROY_WITH_PARENT,
            GTK_MESSAGE_WARNING,
            GTK_BUTTONS_OK,
            "Unallowed operation");
  gtk_window_set_title(GTK_WINDOW(dialog), "Warning");
  gtk_dialog_run(GTK_DIALOG(dialog));
  gtk_widget_destroy(dialog);
}
int main( int argc, char *argv[]){
  GtkWidget *window;
```

```
    GtkWidget *table;
    GtkWidget *info;
    GtkWidget *warn;
    GtkWidget *que;
    GtkWidget *err;
    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
    gtk_window_set_default_size(GTK_WINDOW(window), 220, 150);
    gtk_window_set_title(GTK_WINDOW(window), "Message dialogs");
    table = gtk_table_new(2, 2, TRUE);
    gtk_table_set_row_spacings(GTK_TABLE(table), 2);

    gtk_table_set_col_spacings(GTK_TABLE(table), 2);

    info = gtk_button_new_with_label("Info");
    warn = gtk_button_new_with_label("Warning");
    que = gtk_button_new_with_label("Question");
    err = gtk_button_new_with_label("Error");
    gtk_table_attach(GTK_TABLE(table), info, 0, 1, 0, 1,
GTK_FILL,GTK_FILL, 3, 3);
```

```
  gtk_table_attach(GTK_TABLE(table), warn, 1, 2, 0, 1, GTK_FILL,
GTK_FILL, 3, 3);

  gtk_table_attach(GTK_TABLE(table), que, 0, 1, 1, 2, GTK_FILL,
GTK_FILL, 3, 3);

  gtk_table_attach(GTK_TABLE(table), err, 1, 2, 1, 2, GTK_FILL,
GTK_FILL, 3, 3);

  gtk_container_add(GTK_CONTAINER(window), table);

  gtk_container_set_border_width(GTK_CONTAINER(window), 15);

  g_signal_connect(G_OBJECT(info), "clicked", G_CALLBACK(show_info),
(gpointer) window);

  g_signal_connect(G_OBJECT(warn), "clicked",
G_CALLBACK(show_warning), (gpointer) window);

  g_signal_connect(G_OBJECT(que), "clicked",
G_CALLBACK(show_question), (gpointer) window);

  g_signal_connect(G_OBJECT(err), "clicked", G_CALLBACK(show_error),
(gpointer) window);

  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), G_OBJECT(window));
```

```
  gtk_widget_show_all(window);
```

```
  gtk_main();

  return 0;

}
```

在上面的示例中，我们展示了四种消息对话框。Information, Warning, Question 和 Error 消息对话框。

```
GtkWidget *dialog;

dialog = gtk_message_dialog_new(window,

            GTK_DIALOG_DESTROY_WITH_PARENT,

            GTK_MESSAGE_QUESTION,

            GTK_BUTTONS_YES_NO,

            "Are you sure to quit?");
```

在函数 show_question() 中，我们安排跳出了一个对话框。至于消息对话框是用函数 **gtk_message_dialog_new()** 。 至于函数中的参数设置是在说明你想要显示那种样式的对话框。系统常量 **GTK_MESSAGE_QUESTION** 是在说明我们想要生成一个 question 对话框。系统常量 **GTK_BUTTONS_YES_NO** 将生成“yes“和”no “两个按钮。最后一个参数是我们想要在对话框中显示的文字。

```
gtk_window_set_title(GTK_WINDOW(dialog), "Warning");

gtk_dialog_run(GTK_DIALOG(dialog));

gtk_widget_destroy(dialog);
```

这里，我们为我们刚刚身成的消息对话框设置一个标题。最后运行这个对话框，接着设置了这个对话框必须手动关闭。

GTK 应用程序信息对话框（GtkAboutDialog）

GTK 应用程序对话框的是用来显示应用程序中的有关提示信息的。GTK 应用程序对话框可以显示应用程序的 logo ，名称，版本，版权，网站或者认证之类的信息。当然也可以在其中，给程序的作者，文档整理者，翻译者带来名誉上的声望。

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

void show_about(GtkWidget *widget, gpointer data){

  GdkPixbuf *pixbuf = gdk_pixbuf_new_from_file("battery.png", NULL);

  GtkWidget *dialog = gtk_about_dialog_new();

  gtk_about_dialog_set_name(GTK_ABOUT_DIALOG(dialog), "Battery");

  gtk_about_dialog_set_version(GTK_ABOUT_DIALOG(dialog), "0.9");

  gtk_about_dialog_set_copyright(GTK_ABOUT_DIALOG(dialog), "(c) Jan
Bodnar");

  gtk_about_dialog_set_comments(GTK_ABOUT_DIALOG(dialog), "Battery is
a simple tool for battery checking.");
```

```
  gtk_about_dialog_set_website(GTK_ABOUT_DIALOG(dialog),
"http://www.batteryhq.net");
```

```
  gtk_about_dialog_set_logo(GTK_ABOUT_DIALOG(dialog), pixbuf);

  g_object_unref(pixbuf), pixbuf = NULL;

  gtk_dialog_run(GTK_DIALOG (dialog));

  gtk_widget_destroy(dialog);

}

int main( int argc, char *argv[]){
```

```
  GtkWidget *window;
  GtkWidget *about;
  GdkPixbuf *battery;
  gtk_init(&argc, &argv);
  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
  gtk_window_set_default_size(GTK_WINDOW(window), 220, 150);
  gtk_window_set_title(GTK_WINDOW(window), "Battery");
  gtk_container_set_border_width(GTK_CONTAINER(window), 15);
  gtk_widget_add_events(window, GDK_BUTTON_PRESS_MASK);
  battery = gtk_image_get_pixbuf(GTK_IMAGE(
      gtk_image_new_from_file("battery.png")));
```

```
  g_signal_connect(G_OBJECT(window),
"button-press-event",G_CALLBACK(show_about), (gpointer) window);
```

```
  g_signal_connect_swapped(G_OBJECT(window), "destroy",
G_CALLBACK(gtk_main_quit), G_OBJECT(window));
  gtk_widget_show_all(window);
  gtk_main();
  return 0;
}
```

上面的代码中我们用了构件 GtkAboutDialog 以及该构件的一些特性。我们单击应用程序客户端窗口，该 GTK 应用程序信息对话框就会跳出来。：—）

```
GtkWidget *dialog = gtk_about_dialog_new();
```

我们要生成一个新的 GtkAboutDialog 构件。

```
gtk_about_dialog_set_name(GTK_ABOUT_DIALOG(dialog), "Battery");

gtk_about_dialog_set_version(GTK_ABOUT_DIALOG(dialog), "0.9");

gtk_about_dialog_set_copyright(GTK_ABOUT_DIALOG(dialog), "(c) Jan Bodnar");
```

这个函数是用来设置一个名字，版本以及版权的。

```
GdkPixbuf *pixbuf = gdk_pixbuf_new_from_file("battery.png", NULL);

...

gtk_about_dialog_set_logo(GTK_ABOUT_DIALOG(dialog), pixbuf);

g_object_unref(pixbuf), pixbuf = NULL;
```

这段代码为我们的对话框设置了一个 logo 图标。

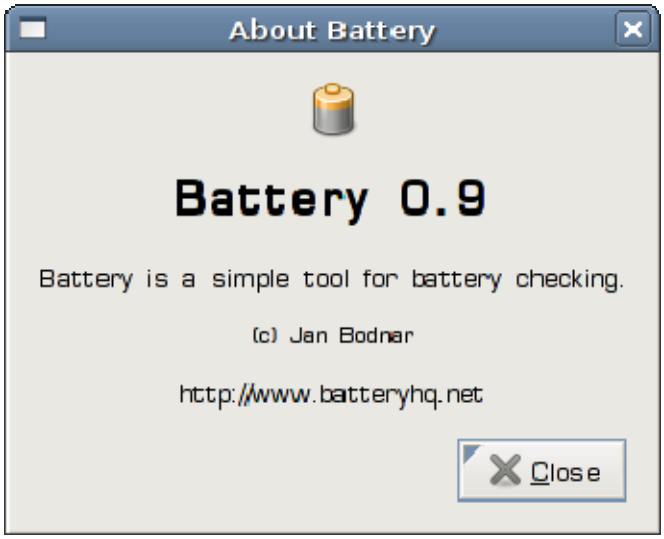

Figure: GtkAboutDialog

GTK 字体选择对话框（GtkFontSelectionDialog）

GTK 字体选择对话框是用来让我们选择字体的。这在一些应用程序中很有代表性。尤其是一些文字处理或者文字排版的软件。

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

void select_font(GtkWidget *widget, gpointer label){

  GtkResponseType result;

  GtkWidget *dialog = gtk_font_selection_dialog_new("Select Font");

  result = gtk_dialog_run(GTK_DIALOG(dialog));

  if (result == GTK_RESPONSE_OK || result == GTK_RESPONSE_APPLY) {

    PangoFontDescription *font_desc;

    gchar *fontname = gtk_font_selection_dialog_get_font_name(

                            GTK_FONT_SELECTION_DIALOG(dialog));

    font_desc = pango_font_description_from_string(fontname);

    gtk_widget_modify_font(GTK_WIDGET(label), font_desc);

    g_free(fontname);

   }

  gtk_widget_destroy(dialog);

}

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

  GtkWidget *window;

  GtkWidget *label;
```

```
GtkWidget *vbox;
GtkWidget *toolbar;
GtkToolItem *font;
gtk_init(&argc, &argv);
window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
gtk_window_set_default_size(GTK_WINDOW(window), 280, 200);
gtk_window_set_title(GTK_WINDOW(window), "Font Selection Dialog");
vbox = gtk_vbox_new(FALSE, 0);
gtk_container_add(GTK_CONTAINER(window), vbox);
toolbar = gtk_toolbar_new();
gtk_toolbar_set_style(GTK_TOOLBAR(toolbar), GTK_TOOLBAR_ICONS);

gtk_container_set_border_width(GTK_CONTAINER(toolbar), 2);

font = gtk_tool_button_new_from_stock(GTK_STOCK_SELECT_FONT);
gtk_toolbar_insert(GTK_TOOLBAR(toolbar), font, -1);
gtk_box_pack_start(GTK_BOX(vbox), toolbar, FALSE, FALSE, 5);
label = gtk_label_new("ZetCode");
gtk_label_set_justify(GTK_LABEL(label), GTK_JUSTIFY_CENTER);
gtk_box_pack_start(GTK_BOX(vbox), label, TRUE, FALSE, 5);
```

```
  g_signal_connect(G_OBJECT(font), "clicked", G_CALLBACK(select_font),
label);

  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), NULL);

  gtk_widget_show_all(window);

  gtk_main();

  return 0;

}
```

在上面的代码示例中，我们在窗口的中央放置了一个简单标签；如果你点击工具栏按钮，那么字体选择对话框就会跳出来 .

```
  GtkWidget *dialog = gtk_font_selection_dialog_new("Select Font");

  result = gtk_dialog_run(GTK_DIALOG(dialog));
```

我们生成了一个字体选择对话框构件即 **GtkFontSelectionDialog。**

```
 if (result == GTK_RESPONSE_OK || result == GTK_RESPONSE_APPLY)

 { PangoFontDescription *font_desc;

   gchar *fontname = gtk_font_selection_dialog_get_font_name(

                            GTK_FONT_SELECTION_DIALOG(dialog));

   font_desc = pango_font_description_from_string(fontname);

   gtk_widget_modify_font(GTK_WIDGET(label), font_desc);

   g_free(fontname);  }
```

如果用户点击“OK“按钮。我们就得到了字体的相关信息，并且把该设置信息作用于前面生成的标签。

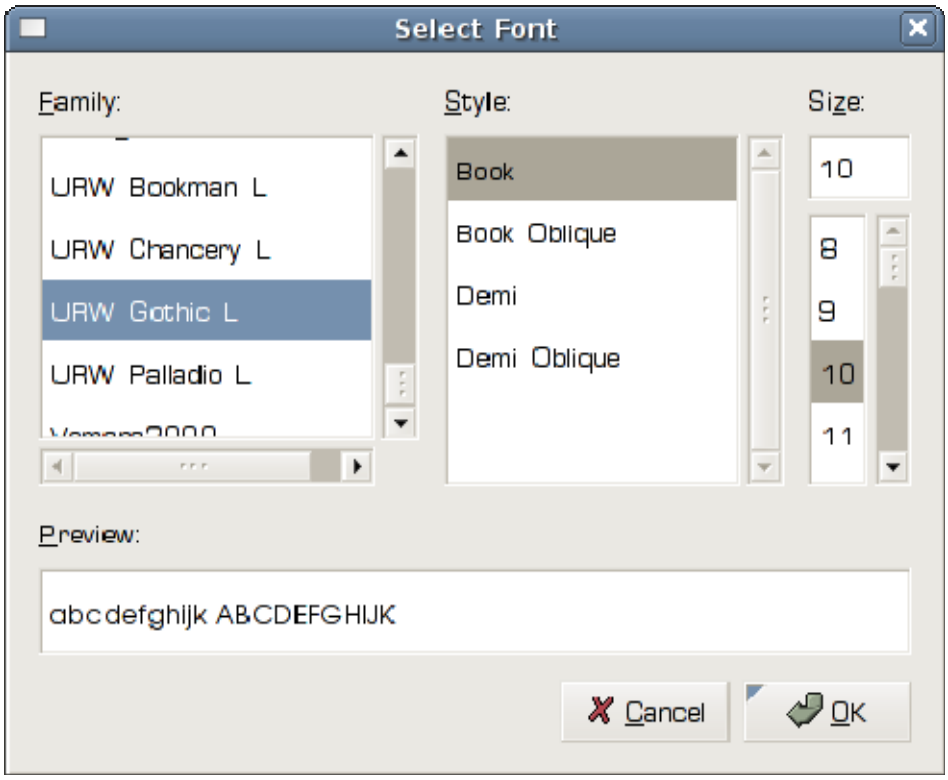

Figure: GtkFontSelectionDialog

GTK 色彩选择对话框（GtkColorSelectionDialog）

顾名思义 GTK 色彩选择对话框就是一个用于颜色选择的对话框。

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

void select_font(GtkWidget *widget, gpointer label){

  GtkResponseType result;

  GtkColorSelection *colorsel;

  GtkWidget *dialog = gtk_color_selection_dialog_new("Font Color");

  result = gtk_dialog_run(GTK_DIALOG(dialog));

  if (result == GTK_RESPONSE_OK)  {
```

```
    GdkColor color;
    colorsel = GTK_COLOR_SELECTION(
                    GTK_COLOR_SELECTION_DIALOG(dialog)->colorsel);
    gtk_color_selection_get_current_color(colorsel, &color);
    gtk_widget_modify_fg(GTK_WIDGET(label),GTK_STATE_NORMAL,&color);}
  gtk_widget_destroy(dialog);
}
int main( int argc, char *argv[]){
  GtkWidget *window;
  GtkWidget *widget;
  GtkWidget *label;
  GtkWidget *vbox;
  GtkWidget *toolbar;
  GtkToolItem *font;
  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
  gtk_window_set_default_size(GTK_WINDOW(window), 280, 200);
  gtk_window_set_title(GTK_WINDOW(window), "Color Selection Dialog");
```

```
  vbox = gtk_vbox_new(FALSE, 0);

  gtk_container_add(GTK_CONTAINER(window), vbox);

  toolbar = gtk_toolbar_new();

  gtk_toolbar_set_style(GTK_TOOLBAR(toolbar), GTK_TOOLBAR_ICONS);

  gtk_container_set_border_width(GTK_CONTAINER(toolbar), 2);

  font = gtk_tool_button_new_from_stock(GTK_STOCK_SELECT_COLOR);

  gtk_toolbar_insert(GTK_TOOLBAR(toolbar), font, -1);

  gtk_box_pack_start(GTK_BOX(vbox), toolbar, FALSE, FALSE, 5);

  label = gtk_label_new("ZetCode");

  gtk_label_set_justify(GTK_LABEL(label), GTK_JUSTIFY_CENTER);

  gtk_box_pack_start(GTK_BOX(vbox), label, TRUE, FALSE, 5);

  g_signal_connect(G_OBJECT(font), "clicked", G_CALLBACK(select_font),
label);

  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), NULL);

  gtk_widget_show_all(window);

  gtk_main();


  return 0;


}
```

这个示例与前面的字体选择对话框非常的类似。不过这里我们要完成的任务是改变标签文字的颜色。

```
 GtkWidget *dialog = gtk_color_selection_dialog_new("Font Color");

 result = gtk_dialog_run(GTK_DIALOG(dialog));
```

当然首先是生成一个 **GtkColorSelectionDialog** 构件。

```
 if (result == GTK_RESPONSE_OK) {

   GdkColor color;

   colorsel =
GTK_COLOR_SELECTION(GTK_COLOR_SELECTION_DIALOG(dialog)->colorsel);

   gtk_color_selection_get_current_color(colorsel,&color);

   gtk_widget_modify_fg(GTK_WIDGET(label),GTK_STATE_NORMAL, &color);

 }
```

当我们点击 OK 后，我们就得到了相关的颜色设置信息，并我们把这个设置用于改变标签文字的颜色。

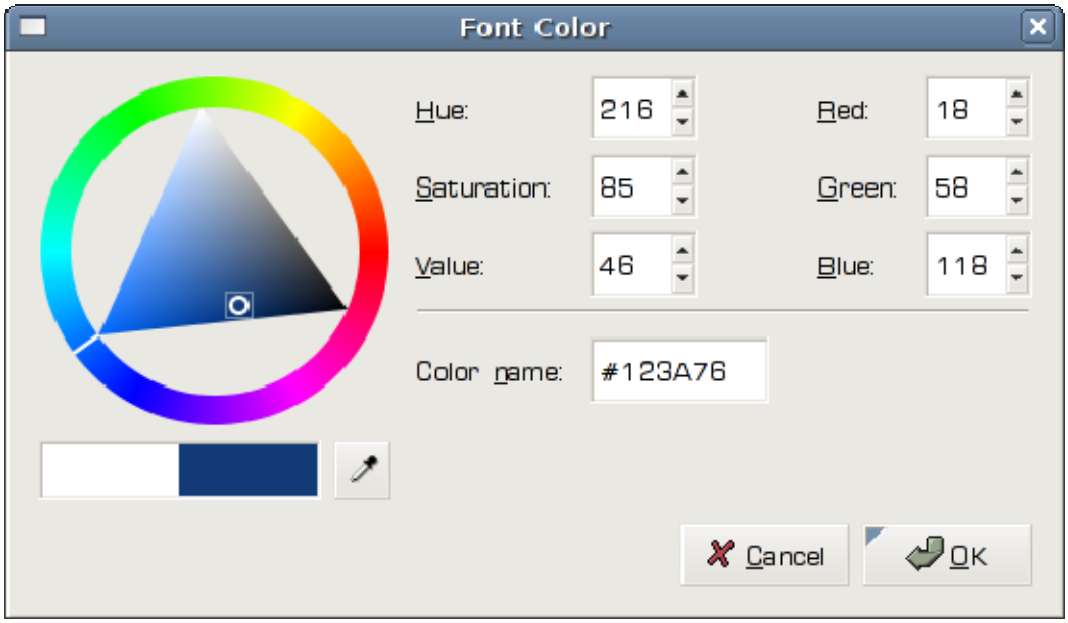

Figure: GtkColorSelectionDialog

GTK+中的构件（GTK+ Widgets）

在本章的 GTK+程序设计教程中，我们将带领大家去更深一步的去领略 GTK+构件的魅力。

毫无疑问，构件是建立一个 GUI 应用程序的基础。在很多年来的发展下，一些特定的构件逐渐就成为了大多数编程工具集（toolkits）和几乎所有操作系统的公用标准了。譬如说，按钮，选择框或者是一个滑动条。至于像 GTK+这样的编程工具集，一开始他的设计人员所抱的哲学态度是：保持构件的数量在一个合理的范围内。正是由于这一点，越来越多的专用构件，被抽象为通用的构件，供大家使用。

GtkButton

GtkButton 是一种很简单易用的构件，通常被用于触发一个动作。

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

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

  GtkWidget *window;

  GtkWidget *fixed;

  GtkWidget *button;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_title(GTK_WINDOW(window), "GtkButton");

  gtk_window_set_default_size(GTK_WINDOW(window), 230, 150);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  fixed = gtk_fixed_new();

  gtk_container_add(GTK_CONTAINER(window), fixed);

  button = gtk_button_new_with_label("Quit");

  gtk_fixed_put(GTK_FIXED(fixed), button, 50, 50);

  gtk_widget_set_size_request(button, 80, 35);
```

```
  g_signal_connect(G_OBJECT(button), "clicked",
G_CALLBACK(gtk_main_quit), G_OBJECT(window));

  g_signal_connect_swapped(G_OBJECT(window), "destroy",
G_CALLBACK(gtk_main_quit), NULL);

  gtk_widget_show_all(window);

  gtk_main();

  return 0;

}
```

上面的这个程序示例就是向大家展示一个按钮构件被放置在一个 fixed 容器构件里。当你按下那个按钮后，程序就会退出。

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

上面这段代码生成了一个带有标签的 **GtkButton** 构件。

```
  g_signal_connect(G_OBJECT(button), "clicked",
G_CALLBACK(gtk_main_quit), G_OBJECT(window));
```

然后我们再把信号 **clicked** 与这个按钮进行绑定。这个信号会引发函数 **gtk_main_quit()** 的调用执行，这个函数的功能正是要使整个程序终止退出。

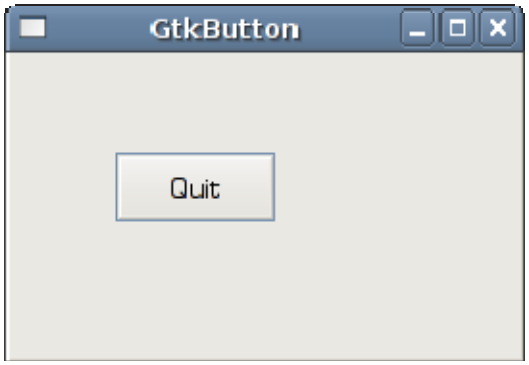

Figure: GtkButton

GtkCheckButton

GtkCheckButton 同样是一个构件，有两种状态。“开”和“关”，开表示一个可见的复选标记。

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

void toggle_title(GtkWidget *widget, gpointer window){

  if (gtk_toggle_button_get_active(GTK_TOGGLE_BUTTON(widget))) {

      gtk_window_set_title(window, "GtkCheckButton");

  } else { gtk_window_set_title(window, "");}

}

int main(int argc, char** argv) {

  GtkWidget *window;

  GtkWidget *frame;

  GtkWidget *check;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_window_set_default_size(GTK_WINDOW(window), 230, 150);

  gtk_window_set_title(GTK_WINDOW(window), "GtkCheckButton");

  frame = gtk_fixed_new();

  gtk_container_add(GTK_CONTAINER(window), frame);

  check = gtk_check_button_new_with_label("Show title");

  gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(check), TRUE);
```

```
  GTK_WIDGET_UNSET_FLAGS(check, GTK_CAN_FOCUS);

  gtk_fixed_put(GTK_FIXED(frame), check, 50, 50);

  g_signal_connect_swapped(window,
"destroy",G_CALLBACK(gtk_main_quit), NULL);

  g_signal_connect(check, "clicked", G_CALLBACK(toggle_title),
(gpointer) window);

  gtk_widget_show_all(window);

  gtk_main();

  return 0;

}
```

我们将要展示的功能是：标题栏的显示状态根据构件 **GtkCheckButton** 的状态变化而变化。

```
check = gtk_check_button_new_with_label("Show title");

gtk_toggle_button_set_active(GTK_TOGGLE_BUTTON(check), TRUE);
```

一个 **GtkCheckButton** 构件被生成了并且默认为已标记（状态为开）， 因为我们想一开始标题栏是默认显示的。

```
GTK_WIDGET_UNSET_FLAGS(check, GTK_CAN_FOCUS);
```

这行代码是取消了对复选框的默认锁定，这样做的原因很简单，是因为我不大喜欢在复选框上真的“罩”上一个“框”，我个人认为那样不大好看。：）

```
if (gtk_toggle_button_get_active(GTK_TOGGLE_BUTTON(widget))) {

   gtk_window_set_title(window, "GtkCheckButton");

} else {gtk_window_set_title(window, "");}
```

这样的话，功能就完成了：标题栏的显示状态根据构件 **GtkCheckButton** 的状态变化而变化。

Figure: GtkCheckButton

GtkFrame

GtkFrame 是一种装饰性的框架，还可以为他设置一个标签（可有可无）。

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

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

  GtkWidget *window;

  GtkWidget *table;

  GtkWidget *frame1;

  GtkWidget *frame2;

  GtkWidget *frame3;

  GtkWidget *frame4;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_window_set_default_size(GTK_WINDOW(window), 250, 250);
```

```
    gtk_window_set_title(GTK_WINDOW(window), "GtkFrame");

    gtk_container_set_border_width(GTK_CONTAINER(window), 10);

    table = gtk_table_new(2, 2, TRUE);

    gtk_table_set_row_spacings(GTK_TABLE(table), 10);

    gtk_table_set_col_spacings(GTK_TABLE(table), 10);

    gtk_container_add(GTK_CONTAINER(window), table);

    frame1 = gtk_frame_new("Shadow In");

    gtk_frame_set_shadow_type(GTK_FRAME(frame1), GTK_SHADOW_IN);

    frame2 = gtk_frame_new("Shadow Out");

    gtk_frame_set_shadow_type(GTK_FRAME(frame2), GTK_SHADOW_OUT);

    frame3 = gtk_frame_new("Shadow Etched In");

    gtk_frame_set_shadow_type(GTK_FRAME(frame3), GTK_SHADOW_ETCHED_IN);

    frame4 = gtk_frame_new("Shadow Etched Out");

    gtk_frame_set_shadow_type(GTK_FRAME(frame4),
  GTK_SHADOW_ETCHED_OUT);
```

```
    gtk_table_attach_defaults(GTK_TABLE(table), frame1, 0, 1, 0, 1);
```

```
    gtk_table_attach_defaults(GTK_TABLE(table), frame2, 0, 1, 1, 2);

    gtk_table_attach_defaults(GTK_TABLE(table), frame3, 1, 2, 0, 1);

    gtk_table_attach_defaults(GTK_TABLE(table), frame4, 1, 2, 1, 2);
```

```
  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), G_OBJECT(window));

  gtk_widget_show_all(window);

  gtk_main();

  return 0;

}
```

这个示例向我们展示了四种不同风格的 frame 框架。这些框架构件是利用表格布局法进行布局的。

```
frame1 = gtk_frame_new("Shadow In");

gtk_frame_set_shadow_type(GTK_FRAME(frame1), GTK_SHADOW_IN);
```

我们生成了一个 **GtkFrame** 构件，并且还为他设置了阴影种类（shadow type）。

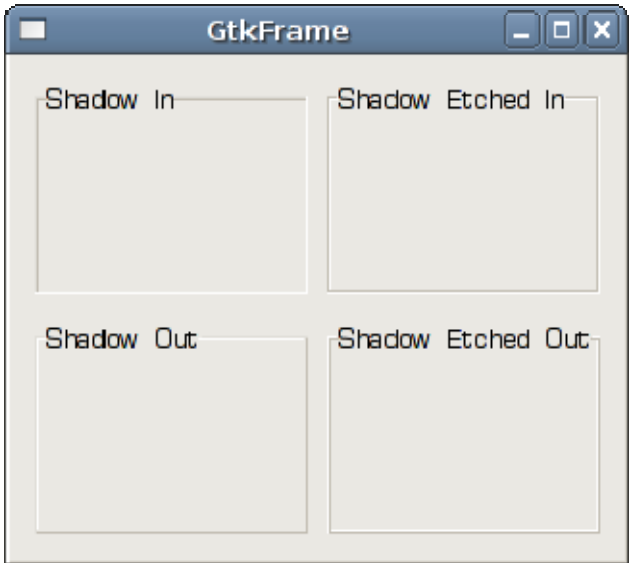

Figure: GtkFrame

GtkLabel

GtkLabel 的功能很明显，是用来显示文字的。

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

```
int main( int argc, char *argv[]){

  GtkWidget *window;

  GtkWidget *label;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_window_set_title(GTK_WINDOW(window), "Nymphetamine");

  gtk_window_set_default_size(GTK_WINDOW(window), 350, 400);

  label = gtk_label_new("Cold was my soul\n\Untold was the pain\n\I faced
when you left me\n\A rose in the rain....\n\So I swore to the razor\n\That
never, enchained\n\Would your dark nails of faith\n\Be pushed through my
veins again\n\\n\Bared on your tomb\n\I'm a prayer for your
loneliness\n\And would you ever soon\n\Come above onto me?\n\For once
upon a time\n\On the binds of your lowliness\n\I could always find the
slot for your sacred key ");

  gtk_label_set_justify(GTK_LABEL(label), GTK_JUSTIFY_CENTER);

  gtk_container_add(GTK_CONTAINER(window), label);

  g_signal_connect_swapped(window, "destroy",G_CALLBACK
(gtk_main_quit), NULL);

  gtk_widget_show_all(window);

  gtk_main();

  return 0;

}
```

这个示例向我们展示了一首歌的歌词。

```
  label = gtk_label_new("Cold was my soul\n\Untold was the pain\n\...
```

我们这里生成了一个 **GtkLabel** 构件。我们利用了换行符来完成多行显示的效果。

```
 gtk_label_set_justify(GTK_LABEL(label), GTK_JUSTIFY_CENTER);
```

哈哈`~我们把标签放在中央，大功告成!

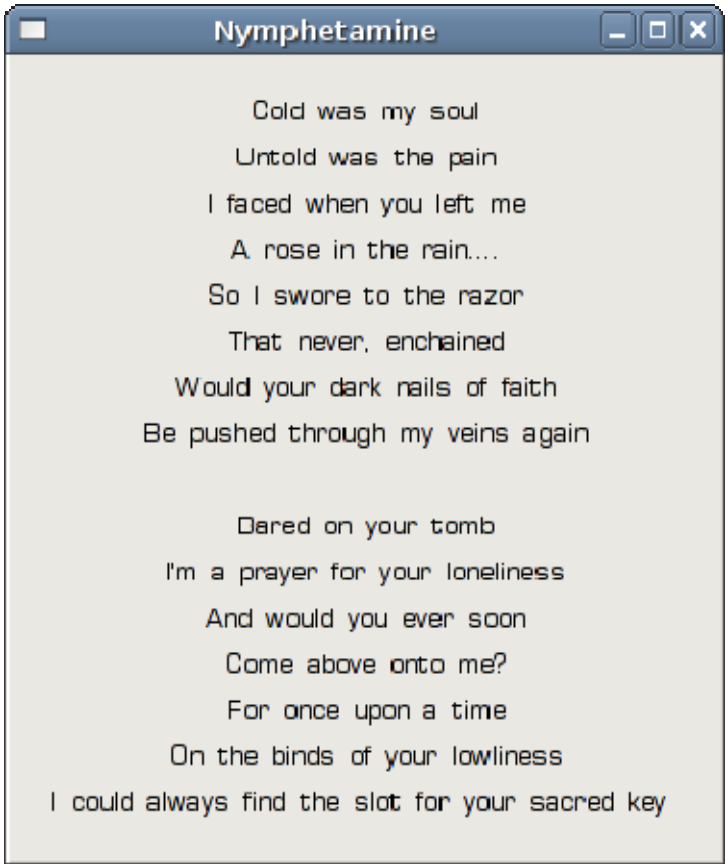

Figure: GtkLabel

在 GtkLabel 构件中我们也能够使用语法 markup 。（就是用来改变文字样式的语法）下面的这个示例就是展示我们是如何完成这个工作的。

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

```
int main( int argc, char *argv[]){
```

```
  GtkWidget *window;

  GtkWidget *label;
```

```
  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_window_set_title(GTK_WINDOW(window), "markup label");

  char *str = "ZetCode, Knowledge only matters";

  label = gtk_label_new(NULL);

  gtk_label_set_markup(GTK_LABEL(label), str);

  gtk_label_set_justify(GTK_LABEL(label), GTK_JUSTIFY_CENTER);

  gtk_container_add(GTK_CONTAINER(window), label);

  gtk_widget_show(label);

  gtk_window_set_default_size(GTK_WINDOW(window), 300, 100);

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

  gtk_widget_show(window);

  gtk_main();

  return 0;

}
```

这个例子是，我们让一部分文字有突出的效果。

```
 char *str = "<b>ZetCode</b>, Knowledge only matters";
```

这就是我们要显示的文字内容。

```
  label = gtk_label_new(NULL);
```

```
  gtk_label_set_markup(GTK_LABEL(label), str);
```

我们生成了一个空的标签构件，然后把样式化的文字（markup text）添加到标签构件里。大功再次告成！ －__－！

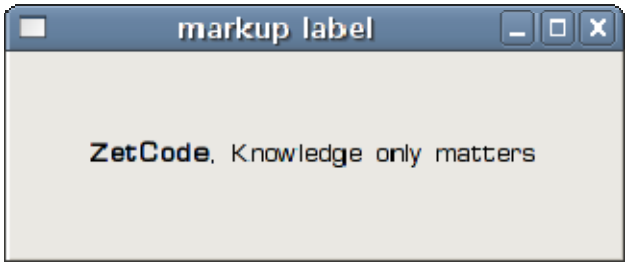

Figure: markup label

GTK+中的构件 II(Widgets)

在本章的 GTK+程序设计中，我们仍然要继续向大家介绍和展示各种各样的构件。

GtkComboBox

GtkComboBox 构件的作用是让程序使用者根据不同的需求从很多选项中进行选择。

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

void combo_selected(GtkWidget *widget, gpointer window){

  gchar *text =  gtk_combo_box_get_active_text(GTK_COMBO_BOX(widget));

  gtk_label_set_text(GTK_LABEL(window), text);

  g_free(text);

}
```

```
int main( int argc, char *argv[]){
```

```
  GtkWidget *window;

  GtkWidget *fixed;
```

```
    GtkWidget *combo;
    GtkWidget *label;

    gtk_init(&argc, &argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

    gtk_window_set_title(GTK_WINDOW(window), "GtkCombo");

    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

    gtk_window_set_default_size(GTK_WINDOW(window), 230, 150);

    fixed = gtk_fixed_new();

    combo = gtk_combo_box_new_text();

    gtk_combo_box_append_text(GTK_COMBO_BOX(combo), "Ubuntu");

    gtk_combo_box_append_text(GTK_COMBO_BOX(combo), "Mandriva");

    gtk_combo_box_append_text(GTK_COMBO_BOX(combo), "Fedora");

    gtk_combo_box_append_text(GTK_COMBO_BOX(combo), "Mint");

    gtk_combo_box_append_text(GTK_COMBO_BOX(combo), "Gentoo");

    gtk_combo_box_append_text(GTK_COMBO_BOX(combo), "Debian");

    gtk_fixed_put(GTK_FIXED(fixed), combo, 50, 50);

    gtk_container_add(GTK_CONTAINER(window), fixed);

    label = gtk_label_new("-");

    gtk_fixed_put(GTK_FIXED(fixed), label, 50, 110);
```

```
  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), G_OBJECT(window));

  g_signal_connect(G_OBJECT(combo),
"changed",G_CALLBACK(combo_selected), (gpointer) label);

  gtk_widget_show_all(window);

  gtk_main();

  return 0;

}
```

上面的这个例子主要是完成一个个下拉选择框（ combo box）和一个标签（label）。在这里下拉选择框有六个选项。他们的名字都是 Linux 操作系统的不同发行版本。标签中的内容就是我们所选择的那个选项的内容。

```
combo = gtk_combo_box_new_text();

gtk_combo_box_append_text(GTK_COMBO_BOX(combo), "Ubuntu");

gtk_combo_box_append_text(GTK_COMBO_BOX(combo), "Mandriva");

gtk_combo_box_append_text(GTK_COMBO_BOX(combo), "Fedora");

gtk_combo_box_append_text(GTK_COMBO_BOX(combo), "Mint");

gtk_combo_box_append_text(GTK_COMBO_BOX(combo), "Gentoo");

gtk_combo_box_append_text(GTK_COMBO_BOX(combo), "Debian");
```

在上面，我们生成了一个 **GtkComboBox** 构件；然后又把 Linux 发行版本的名字填加到其中去。

```
label = gtk_label_new("-");
```

同样我们也生成了一个标签构件。

```
gchar *text =  gtk_combo_box_get_active_text(GTK_COMBO_BOX(widget));
```

```
gtk_label_set_text(GTK_LABEL(window), text);

g_free(text);
```

上面的代码表明，我们从所选的选项中获得了文本内容，并把此内容的传递给了标签构件。在官方公布的 GTK+编程文本中显示：函数 **gtk_combo_box_get_active_text()** 的返回值是当前最新击活选项所对应的内容。这也就是说，我们有必要要释放对应的内存空间。

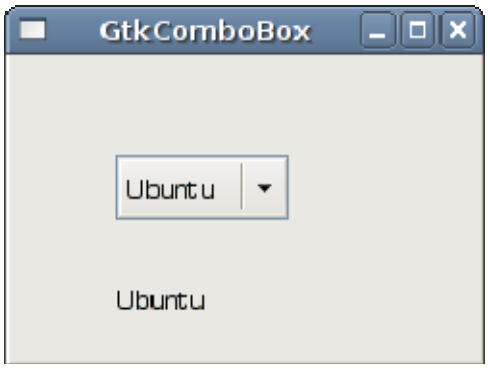

Figure: GtkComboBox

GtkHSeparator

GtkHSeparator 构件是一条水平分割线。这也属于一种布局构件。这一类的构件都是出于界面设计需求的考虑。他的兄弟构件也就是竖直分割线构件 **GtkVSeparator** 。

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

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

  GtkWidget *window;

  GtkWidget *label1;

  GtkWidget *label2;

  GtkWidget *hseparator;

  GtkWidget *vbox;
```

```
  gtk_init(&argc, &argv);
```

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

```
  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_window_set_title(GTK_WINDOW(window), "GtkHSeparator");

  gtk_window_set_resizable(GTK_WINDOW(window), FALSE);

  gtk_container_set_border_width(GTK_CONTAINER(window), 20);

  label1 = gtk_label_new("Zinc is a moderately reactive, blue gray metal
\that tarnishes in moist air and burns in air with a bright bluish-green
flame,\giving off fumes of zinc oxide. It reacts with acids, alkalis and
other non-metals.\If not completely pure, zinc reacts with dilute acids
to release hydrogen.");

  gtk_label_set_line_wrap(GTK_LABEL(label1), TRUE);

  label2 = gtk_label_new("Copper is an essential trace nutrient to all
high \plants and animals. In animals, including humans, it is found
primarily in \the bloodstream, as a co-factor in various enzymes, and in
copper-based pigments. \However, in sufficient amounts, copper can be
poisonous and even fatal to organisms.");

  gtk_label_set_line_wrap(GTK_LABEL(label2), TRUE);

  vbox = gtk_vbox_new(FALSE, 10);

  gtk_container_add(GTK_CONTAINER(window), vbox);

  hseparator = gtk_hseparator_new();

  gtk_box_pack_start(GTK_BOX(vbox), label1, FALSE, TRUE, 0);

  gtk_box_pack_start(GTK_BOX(vbox), hseparator, FALSE, TRUE, 10);

  gtk_box_pack_start(GTK_BOX(vbox), label2, FALSE, TRUE, 0);
```

```
  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), G_OBJECT(window));
```

```
    gtk_widget_show_all(window);

    gtk_main();

    return 0;

}
```

以上的代码正是在向我们展示了两个 label 构件，内容是讲的两种化学元素的定义；接着呢这两个构件又被一个叫做 horizontal 的 GtkHSeparator 构件给分割开来了。这样看起来，整个程序界面就美观多啦！不是么？：）

```
    label1 = gtk_label_new("Zinc is a moderately reactive, blue gray metal
\that tarnishes in moist air and burns in air with a bright bluish-green
flame,\giving off fumes of zinc oxide. It reacts with acids, alkalis and
other non-metals.\If not completely pure, zinc reacts with dilute acids
to release hydrogen.");
```

我们首先生成了第一个 label 构件，内容是锌的定义。

```
 gtk_label_set_line_wrap(GTK_LABEL(label2), TRUE);
```

当然得有换行啦！上面的代码就是给文本进行换行的。

```
hseparator = gtk_hseparator_new();
```

然后我们就又生成了一个水平分割器。（horizontal separator）

```
 gtk_box_pack_start(GTK_BOX(vbox), label1, FALSE, TRUE, 0);

 gtk_box_pack_start(GTK_BOX(vbox), hseparator, FALSE, TRUE, 10);

 gtk_box_pack_start(GTK_BOX(vbox), label2, FALSE, TRUE, 0);
```

最后我们把分割器放置在两个标签中间。

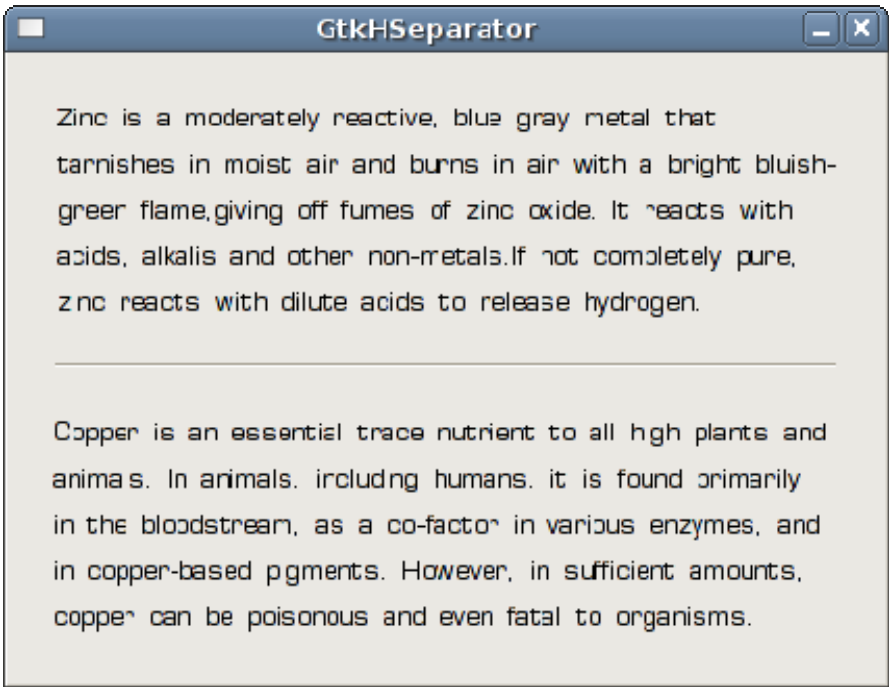

Figure: GtkHSeparator

GtkEntry

GtkEntry 构件是什么？她其实就是一个只拥有单行文本输入框的构件。她主要是用来进行单行文本的输入。

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

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

  GtkWidget *window;

  GtkWidget *table;

  GtkWidget *label1;

  GtkWidget *label2;

  GtkWidget *label3;

  GtkWidget *entry1;

  GtkWidget *entry2;
```

```
GtkWidget *entry3;

gtk_init(&argc, &argv);

window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

gtk_window_set_title(GTK_WINDOW(window), "GtkEntry");

gtk_container_set_border_width(GTK_CONTAINER(window), 10);

table = gtk_table_new(3, 2, FALSE);

gtk_container_add(GTK_CONTAINER(window), table);

label1 = gtk_label_new("Name");

label2 = gtk_label_new("Age");

label3 = gtk_label_new("Occupation");

gtk_table_attach(GTK_TABLE(table), label1, 0, 1, 0, 1,

    GTK_FILL | GTK_SHRINK, GTK_FILL | GTK_SHRINK, 5, 5);

gtk_table_attach(GTK_TABLE(table), label2, 0, 1, 1, 2,

    GTK_FILL | GTK_SHRINK, GTK_FILL | GTK_SHRINK, 5, 5);

gtk_table_attach(GTK_TABLE(table), label3, 0, 1, 2, 3,

    GTK_FILL | GTK_SHRINK, GTK_FILL | GTK_SHRINK, 5, 5);

entry1 = gtk_entry_new();

entry2 = gtk_entry_new();

entry3 = gtk_entry_new();
```

```
    gtk_table_attach(GTK_TABLE(table), entry1, 1, 2, 0, 1,
        GTK_FILL | GTK_SHRINK, GTK_FILL | GTK_SHRINK, 5, 5);
    gtk_table_attach(GTK_TABLE(table), entry2, 1, 2, 1, 2,
        GTK_FILL | GTK_SHRINK, GTK_FILL | GTK_SHRINK, 5, 5);
    gtk_table_attach(GTK_TABLE(table), entry3, 1, 2, 2, 3,
        GTK_FILL | GTK_SHRINK, GTK_FILL | GTK_SHRINK, 5, 5);
    gtk_widget_show(table);
    gtk_widget_show(label1);
    gtk_widget_show(label2);
    gtk_widget_show(label3);
    gtk_widget_show(entry1);
    gtk_widget_show(entry2);
    gtk_widget_show(entry3);
    gtk_widget_show(window);
    g_signal_connect(window, "destroy",G_CALLBACK(gtk_main_quit), NULL);
    gtk_main();
    return 0;
}
```

在本节的例子中我们向大家展示的是三个文本输入框和分别对应的三个标签。

```
 table = gtk_table_new(3, 2, FALSE);
```

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

为了方便我们管理构件，我们使用了 table 容器构件。

```
entry1 = gtk_entry_new();

entry2 = gtk_entry_new();

entry3 = gtk_entry_new();
```

生成三个文本输入框。

```
gtk_table_attach(GTK_TABLE(table), entry1, 1, 2, 0, 1,

    GTK_FILL | GTK_SHRINK, GTK_FILL | GTK_SHRINK, 5, 5);

gtk_table_attach(GTK_TABLE(table), entry2, 1, 2, 1, 2,

    GTK_FILL | GTK_SHRINK, GTK_FILL | GTK_SHRINK, 5, 5);

gtk_table_attach(GTK_TABLE(table), entry3, 1, 2, 2, 3,

    GTK_FILL | GTK_SHRINK, GTK_FILL | GTK_SHRINK, 5, 5);
```

把构件放置到 table 构件中。

Figure: GtkEntry

GtkImage

GtkImage 构件功能是用来显示图象的。

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

```
int main( int argc, char *argv[]){

  GtkWidget *window;

  GtkWidget *image;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_window_set_default_size(GTK_WINDOW(window), 230, 150);

  gtk_window_set_title(GTK_WINDOW(window), "Red Rock");

  gtk_window_set_resizable(GTK_WINDOW(window), FALSE);

  gtk_container_set_border_width(GTK_CONTAINER(window), 2);

  image = gtk_image_new_from_file("redrock.png");

  gtk_container_add(GTK_CONTAINER(window), image);

  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), G_OBJECT(window));

  gtk_widget_show_all(window);

  gtk_main();

  return 0;

}
```

在我们本节的这个例子中，我们向大家展示，我们是如何把一个城堡的图象用 GTK+ 显示出来的。这个城堡叫做 Red Rock 位于斯洛伐克的西部。如果你喜欢的话可以下载这个图片。 here.

```
 gtk_container_set_border_width(GTK_CONTAINER(window), 2);
```

我们给这个图片设置了 2px 的边框大小。

```
image = gtk_image_new_from_file("redrock.png");

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

我们从一个图象文件中加载了图象，并把他放到布局构件中。

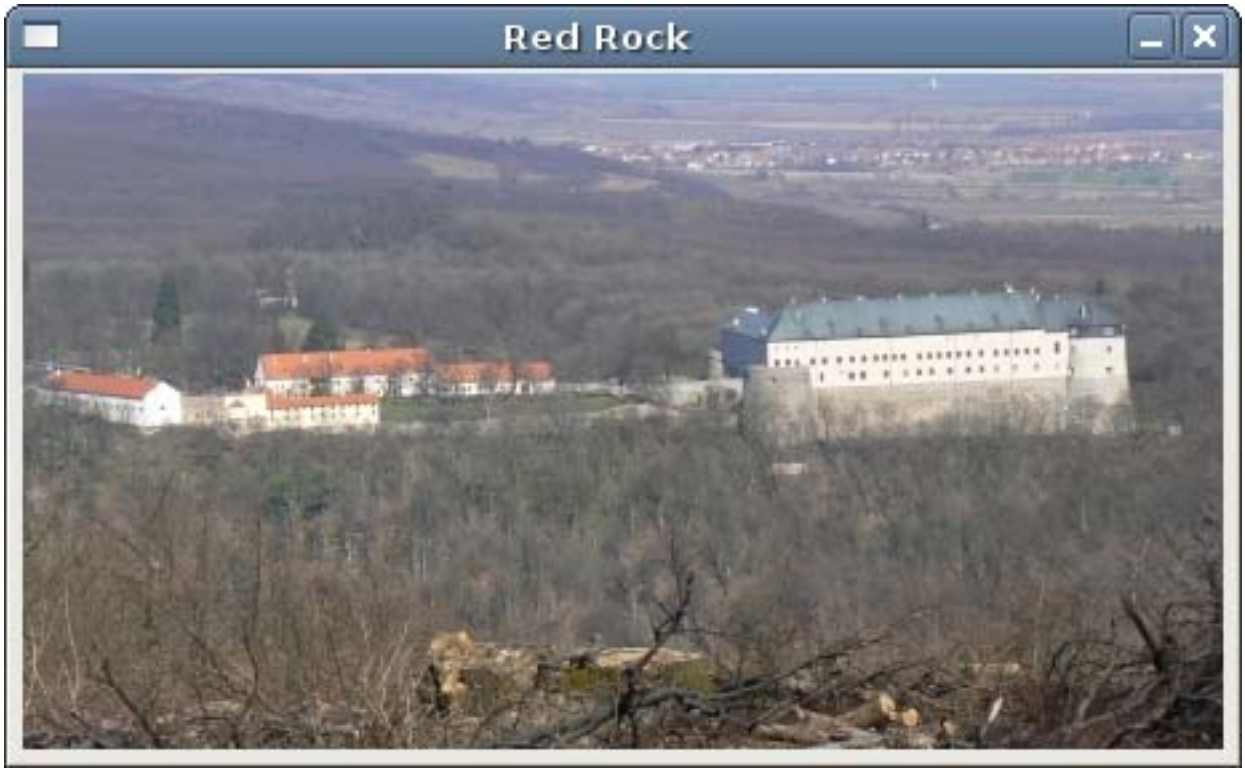

Figure: GtkImage

GtkStatusbar

构件 **GtkStatusbar** 是的功能是用来显示状态信息用的。他通常被自动强制放置于应用程序窗口的底部。

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

void button_pressed(GtkWidget *widget, gpointer window){

   gchar *str;

   str = g_strdup_printf("Button %s
clicked",gtk_button_get_label(GTK_BUTTON(widget)));
```

```
gtk_statusbar_push(GTK_STATUSBAR(window),gtk_statusbar_get_context_id
(GTK_STATUSBAR(window), str), str);

   g_free(str);

}

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

  GtkWidget *window;

  GtkWidget *fixed;

  GtkWidget *button1;

  GtkWidget *button2;

  GtkWidget *statusbar;

  GtkWidget *vbox;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_window_set_default_size(GTK_WINDOW(window), 280, 150);

  gtk_window_set_title(GTK_WINDOW(window), "GtkStatusbar");

  vbox = gtk_vbox_new(FALSE, 2);

  fixed = gtk_fixed_new();

  gtk_container_add(GTK_CONTAINER(window), vbox);

  gtk_box_pack_start(GTK_BOX(vbox), fixed, TRUE, TRUE, 1);
```

```
  button1 = gtk_button_new_with_label("OK");

  gtk_widget_set_size_request(button1, 80, 30 );

  button2 = gtk_button_new_with_label("Apply");

  gtk_widget_set_size_request(button2, 80, 30 );

  gtk_fixed_put(GTK_FIXED(fixed), button1, 50, 50);

  gtk_fixed_put(GTK_FIXED(fixed), button2, 150, 50);

  statusbar = gtk_statusbar_new();

  gtk_box_pack_start(GTK_BOX(vbox), statusbar, FALSE, TRUE, 1);

  g_signal_connect(G_OBJECT(button1),
"clicked",G_CALLBACK(button_pressed), G_OBJECT(statusbar));

  g_signal_connect(G_OBJECT(button2),
"clicked",G_CALLBACK(button_pressed), G_OBJECT(statusbar));

  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), G_OBJECT(window));

  gtk_widget_show_all(window);

  gtk_main();

  return 0;

}
```

在我们上面的这段代码示例中，我们展示了两个按钮和一个状态栏。如果我们单击按钮，一条信息便在状态栏中显示出来。也就说，状态栏反映了哪个按钮被我们按下了。

```
 gchar *str;

 str = g_strdup_printf("Button %s clicked",
gtk_button_get_label(GTK_BUTTON(widget)));
```

这里我们生成了一条消息。

```
gtk_statusbar_push(GTK_STATUSBAR(window),gtk_statusbar_get_context_id
(GTK_STATUSBAR(window), str), str);
```

然后我们把这条消息放置在状态栏中。

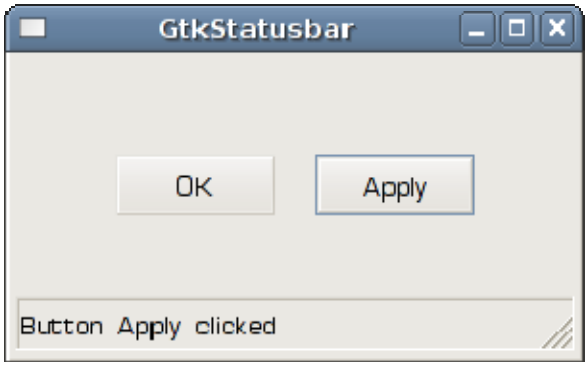

Figure: GtkStatusbar

GtkIconView

构件 GtkIconView 的功能是用来在一个栅格中显示一系列的图标。

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

#include <assert.h>

enum{ COL_DISPLAY_NAME, COL_PIXBUF, NUM_COLS};

GtkTreeModel * init_model(void){

  GtkListStore *list_store;

  GdkPixbuf *p1, *p2, *p3, *p4;

  GtkTreeIter iter;

  GError *err = NULL;

  p1 = gdk_pixbuf_new_from_file("ubuntu.png", &err);
```

```
  p2 = gdk_pixbuf_new_from_file("gnumeric.png", &err);

  p3 = gdk_pixbuf_new_from_file("blender.png", &err);

  p4 = gdk_pixbuf_new_from_file("inkscape.png", &err);

  assert(err==NULL);

  list_store = gtk_list_store_new(NUM_COLS, G_TYPE_STRING,
GDK_TYPE_PIXBUF);

  int i = 0;

  for (i; i < 50; i++) {

    gtk_list_store_append(list_store, &iter);

    gtk_list_store_set(list_store, &iter, COL_DISPLAY_NAME, "ubuntu",
COL_PIXBUF, p1, -1);

    gtk_list_store_append(list_store, &iter);

    gtk_list_store_set(list_store, &iter, COL_DISPLAY_NAME,
"gnumeric", COL_PIXBUF, p2, -1);

    gtk_list_store_append(list_store, &iter);

    gtk_list_store_set(list_store, &iter, COL_DISPLAY_NAME, "blender",
COL_PIXBUF, p3, -1);

    gtk_list_store_append(list_store, &iter);

    gtk_list_store_set(list_store, &iter, COL_DISPLAY_NAME,
"inkscape", COL_PIXBUF, p4, -1);

  }

  return GTK_TREE_MODEL(list_store);
```

```
}

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

  GtkWidget *window;

  GtkWidget *icon_view;
  GtkWidget *sw;
  gtk_init (&argc, &argv);
  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title(GTK_WINDOW (window), "Icon View");
  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
  gtk_container_set_border_width(GTK_CONTAINER(window), 10);
  gtk_widget_set_size_request(window, 350, 330);
  sw = gtk_scrolled_window_new(NULL, NULL);
  gtk_container_add(GTK_CONTAINER (window), sw);
  gtk_scrolled_window_set_policy(GTK_SCROLLED_WINDOW(sw),
GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);
  gtk_scrolled_window_set_shadow_type(GTK_SCROLLED_WINDOW(sw),
      GTK_SHADOW_IN);
  icon_view = gtk_icon_view_new_with_model(init_model());
 gtk_container_add(GTK_CONTAINER(sw), icon_view);
```

```
gtk_icon_view_set_text_column(GTK_ICON_VIEW(icon_view),
COL_DISPLAY_NAME);

gtk_icon_view_set_pixbuf_column(GTK_ICON_VIEW(icon_view),COL_PIXBUF);

gtk_icon_view_set_selection_mode(GTK_ICON_VIEW(icon_view),GTK_SELECTI
ON_MULTIPLE);
```

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

```
  gtk_widget_show_all(window);

  gtk_main();

  return 0;

}
```

在本节的代码示例中，我们显示了 200 个小图标。他们分别是四个著名开源项目的图标。

```
 p1 = gdk_pixbuf_new_from_file("ubuntu.png", &err);

 p2 = gdk_pixbuf_new_from_file("gnumeric.png", &err);

 p3 = gdk_pixbuf_new_from_file("blender.png", &err);

 p4 = gdk_pixbuf_new_from_file("inkscape.png", &err);
```

我们从磁盘文件中载入了四个图标。

```
 list_store =
gtk_list_store_new(NUM_COLS,G_TYPE_STRING,GDK_TYPE_PIXBUF);
```

我们将存储文本和像素缓冲区（pixbuf）的数据。

```
 gtk_list_store_append(list_store, &iter);

 gtk_list_store_set(list_store, &iter, COL_DISPLAY_NAME, "ubuntu",
COL_PIXBUF, p1, -1);
```

上面这段代码，正是把新图标填加到准备显示的缓存区中。

```
icon_view = gtk_icon_view_new_with_model(init_model());
gtk_container_add(GTK_CONTAINER(sw), icon_view);
gtk_icon_view_set_text_column(GTK_ICON_VIEW(icon_view),
COL_DISPLAY_NAME);
gtk_icon_view_set_pixbuf_column(GTK_ICON_VIEW(icon_view),COL_PIXBUF);
```

最后，我们又生成了一个 **GtkIconView** 构件然后把图标和图标文本整合在一起。（译者注：如果本节理解起来有难度的话，正是说明你对 MVC 的概念还不是很清楚，建议查看相关文档：））

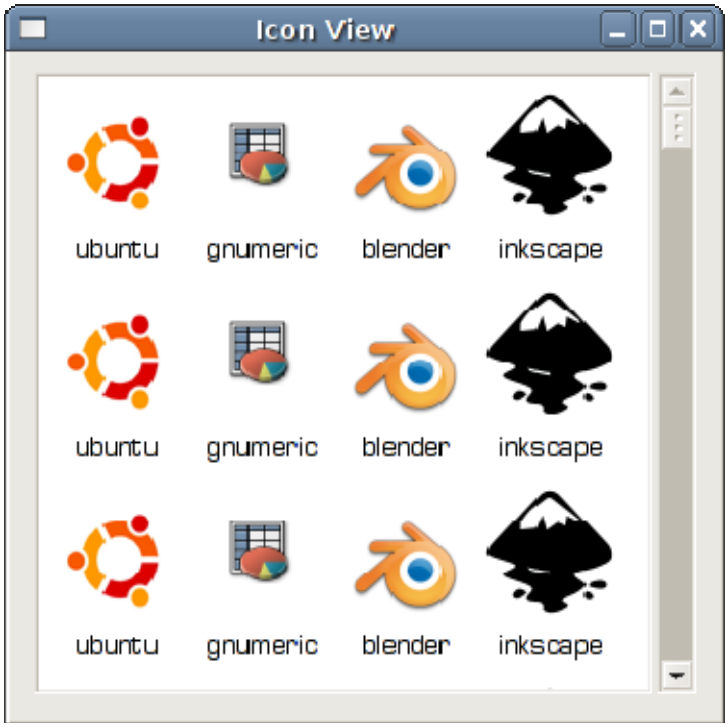

Figure: Icon View

GTK+中的树状列表构件(GtkTreeView)

在本章的 GTK+程序设计教程中，我们将向大家重点介绍非常常用也有点复杂的构件——GtkTreeView 。

GtkTreeView 构件是一个高级的构件，利用他你就可以制作出漂亮的普通列表或者是树状的列表。这个构件里可以包含一或者多行。他的构架呢？正是采用了大名鼎鼎的 MVC (Model View Controller) 设计框架。也就是说数据和显示方式是进行了一种分离的操作。

之前我们有说过复杂这个问题，于是在 GtktreeView 构件中确实还有着其他几个独立的对象结构（objects）。其中 **GtkCellRenderer** 就决定了在 **GtkTreeViewColumn**. 中的数据究竟是如何来进行显示呈现的。**GtkListStore** 和 **GtkTreeStore** 的功能为体现模型（model）的作用。也就是说他们是用来处理和分析将要在 GtkTreeView 显示的数据的。 **GtkTreeIter** 则是一个数据结构被用于在 GtkTreeView 构件中，对行中的数据进行操作。**GtkTreeSelection** 则是用来处理选项的。

一个简单的列表构件示例(Simple List View)

在这个例子中将向大家展示一个简单的列表效果。显示的数据仅仅是文本。

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

enum{  LIST_ITEM = 0,  N_COLUMNS};
```

```
static void init_list(GtkWidget *list){
```

```
  GtkCellRenderer *renderer;

  GtkTreeViewColumn *column;

  GtkListStore *store;

  renderer = gtk_cell_renderer_text_new();

  column = gtk_tree_view_column_new_with_attributes("List Items",
renderer, "text", LIST_ITEM, NULL);

  gtk_tree_view_append_column(GTK_TREE_VIEW(list), column);

  store = gtk_list_store_new(N_COLUMNS, G_TYPE_STRING);

  gtk_tree_view_set_model(GTK_TREE_VIEW(list),GTK_TREE_MODEL(store));

  g_object_unref(store);

}

static void add_to_list(GtkWidget *list, const gchar *str){

  GtkListStore *store;

  GtkTreeIter iter;
```

```
  store =
GTK_LIST_STORE(gtk_tree_view_get_model(GTK_TREE_VIEW(list)));

  gtk_list_store_append(store, &iter);

  gtk_list_store_set(store, &iter, LIST_ITEM, str, -1);

}

void  on_changed(GtkWidget *widget, gpointer label) {

  GtkTreeIter iter;

  GtkTreeModel *model;

  char *value;

  if (gtk_tree_selection_get_selected(GTK_TREE_SELECTION(widget),
&model, &iter)) {

    gtk_tree_model_get(model, &iter, LIST_ITEM, &value,  -1);

    gtk_label_set_text(GTK_LABEL(label), value);

    g_free(value);

  }

}

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

  GtkWidget *window;

  GtkWidget *list;

  GtkWidget *vbox;
```

```
    GtkWidget *label;
    GtkTreeSelection *selection;
    gtk_init(&argc, &argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window), 10);
    gtk_widget_set_size_request(window, 270, 250);

    gtk_window_set_title(GTK_WINDOW(window), "List View");

    list = gtk_tree_view_new();
    gtk_tree_view_set_headers_visible(GTK_TREE_VIEW(list), FALSE);
    vbox = gtk_vbox_new(FALSE, 0);
    gtk_box_pack_start(GTK_BOX(vbox), list, TRUE, TRUE, 5);
    label = gtk_label_new("");
    gtk_label_set_justify(GTK_LABEL(label), GTK_JUSTIFY_CENTER);
    gtk_box_pack_start(GTK_BOX(vbox), label, FALSE, FALSE, 5);
    gtk_container_add(GTK_CONTAINER(window), vbox);
    init_list(list);
    add_to_list(list, "Aliens");
    add_to_list(list, "Leon");
```

```
  add_to_list(list, "Capote");

  add_to_list(list, "Saving private Ryan");

  add_to_list(list, "Der Untergang");

  selection = gtk_tree_view_get_selection(GTK_TREE_VIEW(list));

  g_signal_connect(selection, "changed",G_CALLBACK(on_changed),
label);

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

  gtk_widget_show_all(window);


  gtk_main ();


  return 0;

}
```

在我们上面的这个示例代码中，我们将向大家展示的是 5 个条目并布置于 **GtkTreeView** 构件中。我们首先在 window 中放置一个 **GtkVBox** 构件。 在这个 GtkVBox 构件中含有两个构件：**GtkTreeView** 和 **GtkLabel**。

```
  list = gtk_tree_view_new();

  gtk_tree_view_set_headers_visible(GTK_TREE_VIEW(list), FALSE);
```

上面的代码生成了一个 **GtkTreeView** 构件并且栏数被设置为 FALSE 即只有一栏。

```
  label = gtk_label_new("");

  gtk_label_set_justify(GTK_LABEL(label), GTK_JUSTIFY_CENTER);

  gtk_box_pack_start(GTK_BOX(vbox), label, FALSE, FALSE, 5);
```

生成了一个 **GtkLabel** 构件，并且把它放置在 **GtkTreeView** 构件的下方，设置为居中。

```
init_list(list);
```

调用 list()函数，初始化构件 list。

```
renderer = gtk_cell_renderer_text_new();

column = gtk_tree_view_column_new_with_attributes("List
Items",renderer, "text", LIST_ITEM, NULL);

gtk_tree_view_append_column(GTK_TREE_VIEW(list), column);
```

在初始化函数中，我们生成了只有一栏的 GtkTreeView。

```
store = gtk_list_store_new(N_COLUMNS, G_TYPE_STRING);

gtk_tree_view_set_model(GTK_TREE_VIEW(list), GTK_TREE_MODEL(store));
```

接下来我们又生成了一个 **GtkListStore** 构件(a model) 然后把它与 list 构件绑定。

```
g_object_unref(store);
```

这个 model 被自动的销毁，以释放内存空间。

```
add_to_list(list, "Aliens");
```

上面就是在调用 add_to_list（）函数，实现向 list 中在增加一个选项的功能。

```
store = GTK_LIST_STORE(gtk_tree_view_get_model(GTK_TREE_VIEW(list)));

gtk_list_store_append(store, &iter);

gtk_list_store_set(store, &iter, LIST_ITEM, str, -1);
```

在函数 **add_to_list()** 中，我们利用系统函数 **gtk_tree_view_get_model()**来获得 model。我们生成新的一行并把行中的数据交给 model 处理，这里正是借助 **GtkTreeIter** 来完成这个功能。

```
selection = gtk_tree_view_get_selection(GTK_TREE_VIEW(list));
```

GtkTreeSelection 际上并不需要明确生成。在这里，我们是利用 **GtkTreeView** 构件自动来生成。来帮助完成这项工作的正如你所见到的是系统函数 **gtk_tree_view_get_selection()**。

```
g_signal_connect(selection, "changed",G_CALLBACK(on_changed), label);
```

这个就很好理解了，把 changed 信号与 **GtkTreeSelection** 绑定，我们就可以与回调函数 **on_changed()**建立了联系。

```
gtk_tree_model_get(model, &iter, LIST_ITEM, &value,  -1);

gtk_label_set_text(GTK_LABEL(label), value);
```

在这个回调函数里，我们取得了对应行的数据，当然是通过 iter 来获取的。

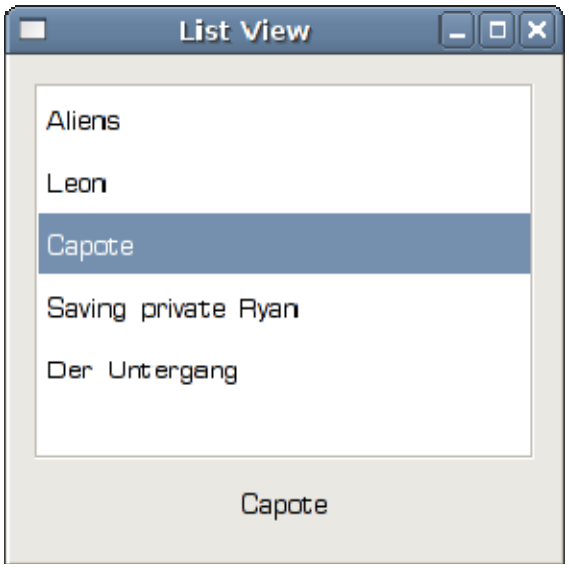

Figure: List View

高级列表（Advanced List View）

在第二个例子中，我们将在前者的基础上填加一些额外的功能。我们将实现能够列表中填加或者去处其中的数据项。

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

enum{  LIST_ITEM = 0,  N_COLUMNS};

GtkWidget *list;

static void append_item(GtkWidget *widget, gpointer entry){

  GtkListStore *store;

  GtkTreeIter  iter;

  const char *str = gtk_entry_get_text(entry);
```

```
  store =
GTK_LIST_STORE(gtk_tree_view_get_model(GTK_TREE_VIEW(list)));

  gtk_list_store_append(store, &iter);

  gtk_list_store_set(store, &iter, LIST_ITEM, str, -1);

}

static void remove_item(GtkWidget *widget, gpointer selection){

  GtkListStore *store;

  GtkTreeModel *model;

  GtkTreeIter  iter;

  store=GTK_LIST_STORE(gtk_tree_view_get_model(GTK_TREE_VIEW(list)));

  model = gtk_tree_view_get_model (GTK_TREE_VIEW (list));

  if (gtk_tree_model_get_iter_first(model, &iter) == FALSE) return;

  if (gtk_tree_selection_get_selected(GTK_TREE_SELECTION(selection),
&model, &iter)){ gtk_list_store_remove(store, &iter);}

}

static void remove_all(GtkWidget *widget, gpointer selection){

  GtkListStore *store;

  GtkTreeModel *model;

  GtkTreeIter  iter;

  store=GTK_LIST_STORE(gtk_tree_view_get_model(GTK_TREE_VIEW(list)));
```

```
  model = gtk_tree_view_get_model (GTK_TREE_VIEW (list));

  if (gtk_tree_model_get_iter_first(model, &iter) == FALSE) return;

  gtk_list_store_clear(store);

}
```

```
static void init_list(GtkWidget *list){
```

```
  GtkCellRenderer    *renderer;

  GtkTreeViewColumn  *column;

  GtkListStore       *store;

  renderer = gtk_cell_renderer_text_new();

  column = gtk_tree_view_column_new_with_attributes("List Item",
renderer, "text", LIST_ITEM, NULL);

  gtk_tree_view_append_column(GTK_TREE_VIEW (list), column);

  store = gtk_list_store_new (N_COLUMNS, G_TYPE_STRING);

  gtk_tree_view_set_model(GTK_TREE_VIEW (list),
GTK_TREE_MODEL(store));

  g_object_unref(store);

}

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

  GtkWidget *window;

  GtkWidget *sw;
```

```
  GtkWidget *remove;

  GtkWidget *add;

  GtkWidget *removeAll;

  GtkWidget *entry;

  GtkWidget *vbox;

  GtkWidget *hbox;

  GtkTreeSelection *selection;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  sw = gtk_scrolled_window_new(NULL, NULL);

  list = gtk_tree_view_new();

  gtk_window_set_title (GTK_WINDOW (window), "List View");

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_container_set_border_width (GTK_CONTAINER (window), 10);

  gtk_widget_set_size_request (window, 370, 270);

  gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW(sw),
GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);

  gtk_scrolled_window_set_shadow_type (GTK_SCROLLED_WINDOW(sw),
GTK_SHADOW_ETCHED_IN);

  gtk_tree_view_set_headers_visible (GTK_TREE_VIEW (list), FALSE);

  vbox = gtk_vbox_new(FALSE, 0);
```

```
  gtk_box_pack_start(GTK_BOX(vbox), sw, TRUE, TRUE, 5);

  hbox = gtk_hbox_new(TRUE, 5);

  add = gtk_button_new_with_label("Add");

  remove = gtk_button_new_with_label("Remove");

  removeAll = gtk_button_new_with_label("Remove All");

  entry = gtk_entry_new();

  gtk_box_pack_start(GTK_BOX(hbox), add, FALSE, TRUE, 3);

  gtk_box_pack_start(GTK_BOX(hbox), entry, FALSE, TRUE, 3);

  gtk_box_pack_start(GTK_BOX(hbox), remove, FALSE, TRUE, 3);

  gtk_box_pack_start(GTK_BOX(hbox), removeAll, FALSE, TRUE, 3);

  gtk_box_pack_start(GTK_BOX(vbox), hbox, FALSE, TRUE, 3);

  gtk_container_add(GTK_CONTAINER (sw), list);

  gtk_container_add(GTK_CONTAINER (window), vbox);

  init_list(list);

  selection  = gtk_tree_view_get_selection(GTK_TREE_VIEW(list));

  g_signal_connect(G_OBJECT(add), "clicked",
G_CALLBACK(append_item), entry);

  g_signal_connect(G_OBJECT(remove), "clicked",
G_CALLBACK(remove_item), selection);

  g_signal_connect(G_OBJECT(removeAll), "clicked",
G_CALLBACK(remove_all), selection);
```

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

  gtk_widget_show_all(window);

  gtk_main ();

  return 0;
```

```
}
```

与前面的例子中的 label 不同的是，我们生成了三个按钮和一个单行文本输入框。我们将实现能够动态的为列表增加一个新的数据项或者去处选中的数据项以及全部数据项。

```
 sw = gtk_scrolled_window_new(NULL, NULL);

 ...

 gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW(sw),
GTK_POLICY_AUTOMATIC, GTK_POLICY_AUTOMATIC);

 gtk_scrolled_window_set_shadow_type (GTK_SCROLLED_WINDOW(sw),
GTK_SHADOW_ETCHED_IN);

 ...

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

 ...

 gtk_container_add(GTK_CONTAINER (sw), list);
```

GtkTreeView 构件被放置在带有滑块的窗口中。

```
 if (gtk_tree_selection_get_selected(GTK_TREE_SELECTION(selection),
&model, &iter)) {   gtk_list_store_remove(store, &iter); }
```

系统函数 **gtk_list_store_remove()**的功能是去处列表中的所选的数据项。

```
gtk_list_store_clear(store);
```

系统函数 **gtk_list_store_clear()**将用于清除列表中的所有数据项。

```
if (gtk_tree_model_get_iter_first(model, &iter) == FALSE) return;
```

上面的代码是用于检查是否在列表中还存有剩下的数据项。很显然，我们能够把列表清除的一干二净。

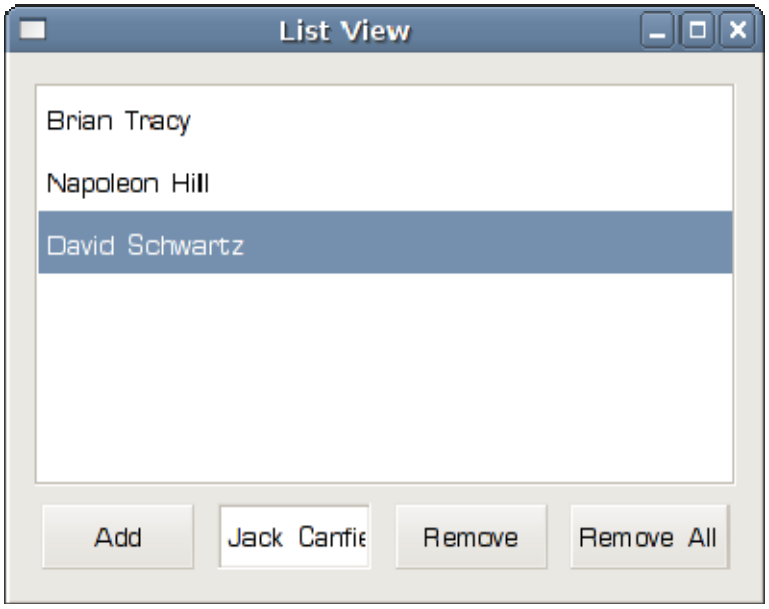

Figure: Advanced List View

树状视图（Tree View）

接着，我们将向大家展示如何运用构件 **GtkTreeView** 来去显示有等级差异的数据项。在先前的两个例子中，我们我们用到了列表试图，现在我们介绍树状视图。

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

enum{  COLUMN = 0,  NUM_COLS} ;

void  on_changed(GtkWidget *widget, gpointer statusbar) {

  GtkTreeIter iter;

  GtkTreeModel *model;

  char *value;
```

```
  if (gtk_tree_selection_get_selected(

      GTK_TREE_SELECTION(widget), &model, &iter)) {

    gtk_tree_model_get(model, &iter, COLUMN, &value,  -1);

    gtk_statusbar_push(GTK_STATUSBAR(statusbar),
gtk_statusbar_get_context_id(GTK_STATUSBAR(statusbar),
value), value);

    g_free(value);

  }
```

```
}
```

```
static GtkTreeModel *

create_and_fill_model (void){

  GtkTreeStore *treestore;

  GtkTreeIter toplevel, child;

  treestore = gtk_tree_store_new(NUM_COLS, G_TYPE_STRING);

  gtk_tree_store_append(treestore, &toplevel, NULL);

  gtk_tree_store_set(treestore, &toplevel,COLUMN, "Scripting
languages", -1);

  gtk_tree_store_append(treestore, &child, &toplevel);

  gtk_tree_store_set(treestore, &child, COLUMN, "Python", -1);

  gtk_tree_store_append(treestore, &child, &toplevel);

  gtk_tree_store_set(treestore, &child, COLUMN, "Perl",-1);
```

```
  gtk_tree_store_append(treestore, &child, &toplevel);
  gtk_tree_store_set(treestore, &child, COLUMN, "PHP",-1);
  gtk_tree_store_append(treestore, &toplevel, NULL);
  gtk_tree_store_set(treestore, &toplevel, COLUMN, "Compiled
languages",-1);
  gtk_tree_store_append(treestore, &child, &toplevel);
  gtk_tree_store_set(treestore, &child, COLUMN, "C",-1);
  gtk_tree_store_append(treestore, &child, &toplevel);
  gtk_tree_store_set(treestore, &child, COLUMN, "C++",-1);
  gtk_tree_store_append(treestore, &child, &toplevel);
  gtk_tree_store_set(treestore, &child, COLUMN, "Java",-1);
  return GTK_TREE_MODEL(treestore);
}
static GtkWidget *
create_view_and_model (void){
  GtkTreeViewColumn *col;
  GtkCellRenderer *renderer;
  GtkWidget *view;
  GtkTreeModel *model;
```

```
  view = gtk_tree_view_new();
  col = gtk_tree_view_column_new();
  gtk_tree_view_column_set_title(col, "Programming languages");
  gtk_tree_view_append_column(GTK_TREE_VIEW(view), col);
  renderer = gtk_cell_renderer_text_new();
  gtk_tree_view_column_pack_start(col, renderer, TRUE);
  gtk_tree_view_column_add_attribute(col, renderer, "text", COLUMN);
  model = create_and_fill_model();
  gtk_tree_view_set_model(GTK_TREE_VIEW(view), model);
  g_object_unref(model);
  return view;
}
Int main (int argc, char **argv){
  GtkWidget *window;
  GtkWidget *view;
  GtkTreeSelection *selection;
  GtkWidget *vbox;
  GtkWidget *statusbar;
  gtk_init(&argc, &argv);
  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
```

```
  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_window_set_title(GTK_WINDOW(window), "Tree View");

  gtk_widget_set_size_request (window, 350, 300);

  vbox = gtk_vbox_new(FALSE, 2);
```

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

```
  view = create_view_and_model();

  selection = gtk_tree_view_get_selection(GTK_TREE_VIEW(view));

  gtk_box_pack_start(GTK_BOX(vbox), view, TRUE, TRUE, 1);

  statusbar = gtk_statusbar_new();

  gtk_box_pack_start(GTK_BOX(vbox), statusbar, FALSE, TRUE, 1);

  g_signal_connect(selection, "changed", G_CALLBACK(on_changed),
statusbar);

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

  gtk_widget_show_all(window);

  gtk_main();

  return 0;

}
```

在我们上面的示例中，我们来完成一项任务：把脚本语言和传统编程语言对应的数据项，进行区分。“语言种类”作为其对应数据项中的顶层节点，也就是说是一行数据列表的“头头”。当前选种的数据项，将在状态栏中显示出来。

从上面的这些步骤中，我们可以清晰的看到，树状视图与列表视图的生成方法很相似。

```
GtkTreeStore *treestore;
```

这里我们当然要使用一个不同的 model—— **GtkTreeStore**。

```
treestore = gtk_tree_store_new(NUM_COLS, G_TYPE_STRING);
```

我们生成的 **GtkTreeStore** 只有一列。

```
gtk_tree_store_append(treestore, &toplevel, NULL);

gtk_tree_store_set(treestore, &toplevel, COLUMN, "Scripting
languages",-1);
```

这其中的代码就是在完成一个顶层节点的操作。

```
gtk_tree_store_append(treestore, &child, &toplevel);

gtk_tree_store_set(treestore, &child, COLUMN, "Python",-1);
```

上面的代码在生成一个子数据项。

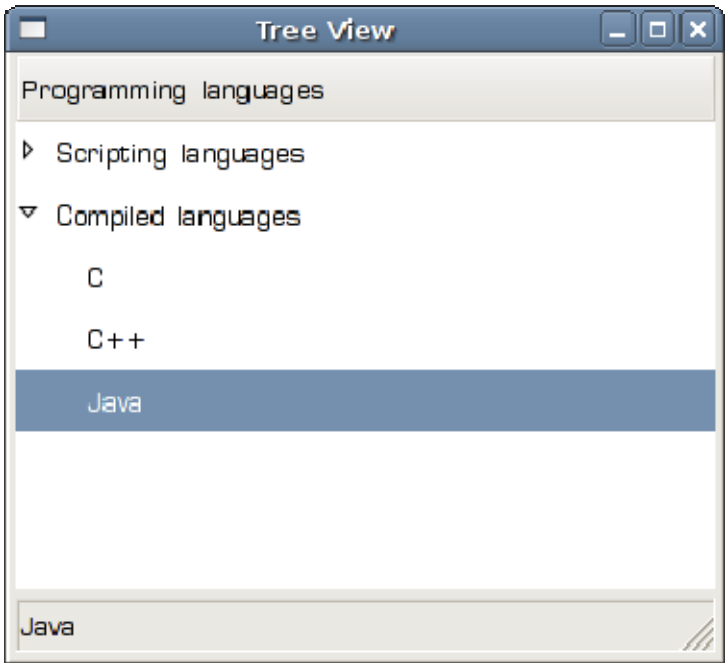

Figure: Tree View

Gtk 中的文本视图（GtkTexView Widget）

在本章的 Gtk+程序设计教程中，我们将重点介绍 GtkTexView 构件。

GtkTexView w 构件被常常用来显示和编辑多行的文本。正如我们一再提到的，**GtkTexBuffer** 构件也是给予 MVC 的设计。GtkTextView 就是显示（view）元素而 GtkTexBuffer 则代表了 model 元素。GtkTexBuffer 常常被用来处理文本数据。**GtkTextTag** 则是一种被用于文本的属性。**GtkTextIter** 则是代表了两个字符之间的空隙。那么很好理解，文本的排版操作多用 iterators。

简单的例子（Simple example）

在我们的第一个例子中，我们将向大家展示 GtkTexView 的一些功能。我们还将教大家怎么样去应用各种各样的文本标记（ tags ）。

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

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

  GtkWidget *window;

  GtkWidget *view;

  GtkWidget *vbox;

  GtkTextBuffer *buffer;

  GtkTextIter start, end;

  GtkTextIter iter;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_window_set_default_size(GTK_WINDOW(window), 250, 200);

  gtk_window_set_title(GTK_WINDOW(window), "TextView");

  gtk_container_set_border_width(GTK_CONTAINER(window), 5);

  GTK_WINDOW(window)->allow_shrink = TRUE;
```

```
  vbox = gtk_vbox_new(FALSE, 0);
```

```
  view = gtk_text_view_new();
```

```
  gtk_box_pack_start(GTK_BOX(vbox), view, TRUE, TRUE, 0);

  buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(view));

  gtk_text_buffer_create_tag(buffer, "gap","pixels_above_lines", 30,
NULL);

  gtk_text_buffer_create_tag(buffer, "lmarg", "left_margin", 5, NULL);

  gtk_text_buffer_create_tag(buffer, "blue_fg", "foreground", "blue",
NULL);

  gtk_text_buffer_create_tag(buffer, "gray_bg", "background", "gray",
NULL);

  gtk_text_buffer_create_tag(buffer, "italic", "style",
PANGO_STYLE_ITALIC, NULL);

  gtk_text_buffer_create_tag(buffer, "bold", "weight",
PANGO_WEIGHT_BOLD, NULL);

  gtk_text_buffer_get_iter_at_offset(buffer, &iter, 0);

  gtk_text_buffer_insert(buffer, &iter, "Plain text\n", -1);

  gtk_text_buffer_insert_with_tags_by_name(buffer, &iter, "Colored
Text\n", -1, "blue_fg", "lmarg",  NULL);

  gtk_text_buffer_insert_with_tags_by_name (buffer, &iter, "Text with
colored background\n", -1, "lmarg", "gray_bg", NULL);

  gtk_text_buffer_insert_with_tags_by_name (buffer, &iter, "Text in
italics\n", -1, "italic", "lmarg",  NULL);
```

```
  gtk_text_buffer_insert_with_tags_by_name (buffer, &iter, "Bold
text\n", -1, "bold", "lmarg",  NULL);

  gtk_container_add(GTK_CONTAINER(window), vbox);

  g_signal_connect_swapped(G_OBJECT(window),
"destroy",G_CALLBACK(gtk_main_quit), G_OBJECT(window));

  gtk_widget_show_all(window);
```

```
  gtk_main();
```

```
  return 0;

}
```

这个例子展示了如何利用各种各样的文本标记（ **GtkTextTags**）来显示文本。

```
 view = gtk_text_view_new();
```

生成一个 **GtkTextView**。

```
 gtk_text_buffer_create_tag(buffer, "blue_fg", "foreground", "blue",
NULL);
```

这就是一个运用 **GtkTextTag** 的例子，这个标记改变了文本的颜色。

```
 gtk_text_buffer_insert_with_tags_by_name(buffer, &iter, "Colored
Text\n", -1, "blue_fg", "lmarg",  NULL);
```

这个代码插入了一些文本，并运用了一个特殊的文本标记 **blue_fg**。

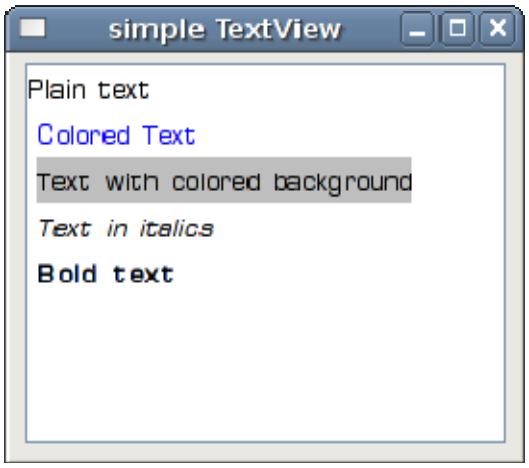

Figure: Simple TextView

行和栏（Lines and Columns）

在接下来的示例中将要显示文本编辑光标目前处于的行数和列数。

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

update_statusbar(GtkTextBuffer *buffer, GtkStatusbar  *statusbar){

  gchar *msg;

  gint row, col;

  GtkTextIter iter;

  gtk_statusbar_pop(statusbar, 0);

  gtk_text_buffer_get_iter_at_mark(buffer, &iter,
gtk_text_buffer_get_insert(buffer));

  row = gtk_text_iter_get_line(&iter);

  col = gtk_text_iter_get_line_offset(&iter);

  msg = g_strdup_printf("Col %d Ln %d", col+1, row+1);
```

```

  gtk_statusbar_push(statusbar, 0, msg);
```

```
  g_free(msg);

}

static void mark_set_callback(GtkTextBuffer *buffer, const GtkTextIter
*new_location, GtkTextMark *mark, gpointer data){

  update_statusbar(buffer, GTK_STATUSBAR(data));

}

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

  GtkWidget *window;

  GtkWidget *vbox;

  GtkWidget *toolbar;

  GtkWidget *view;

  GtkWidget *statusbar;

  GtkToolItem *exit;

  GtkTextBuffer *buffer;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_window_set_default_size(GTK_WINDOW(window), 250, 200);

  gtk_window_set_title(GTK_WINDOW(window), "lines & cols");

  vbox = gtk_vbox_new(FALSE, 0);
```

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

  toolbar = gtk_toolbar_new();

  gtk_toolbar_set_style(GTK_TOOLBAR(toolbar), GTK_TOOLBAR_ICONS);

  exit = gtk_tool_button_new_from_stock(GTK_STOCK_QUIT);


  gtk_toolbar_insert(GTK_TOOLBAR(toolbar), exit, -1);


  gtk_box_pack_start(GTK_BOX(vbox), toolbar, FALSE, FALSE, 5);

  view = gtk_text_view_new();

  gtk_box_pack_start(GTK_BOX(vbox), view, TRUE, TRUE, 0);

  gtk_widget_grab_focus(view);

  buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(view));

  statusbar = gtk_statusbar_new();

  gtk_box_pack_start(GTK_BOX(vbox), statusbar, FALSE, FALSE, 0);

  g_signal_connect(G_OBJECT(exit),
"clicked",G_CALLBACK(gtk_main_quit), NULL);

  g_signal_connect(buffer, "changed",G_CALLBACK(update_statusbar),
statusbar);

  g_signal_connect_object(buffer,
"mark_set",G_CALLBACK(mark_set_callback), statusbar, 0);

  g_signal_connect_swapped(G_OBJECT(window), "destroy",
G_CALLBACK(gtk_main_quit), NULL);

  gtk_widget_show_all(window);

  update_statusbar(buffer, GTK_STATUSBAR (statusbar));
```

```
  gtk_main();

  return 0;

}
```

在上面的代码示例中，我们完成了在状态栏中显示当前文本编辑光标所处于的行和列数。

```
 view = gtk_text_view_new();
```

生成一了 **GtkTextView** 构件。

```
 g_signal_connect(buffer, "changed",G_CALLBACK(update_statusbar),
statusbar);
```

如果我们要更改文本，我们只需要调用回调函数 **update_statusbar()** 就可以了。

```
 g_signal_connect_object(buffer, "mark_set",
G_CALLBACK(mark_set_callback), statusbar, 0);
```

当光标在移动的时候， **mark_set** 信号就被发射出去了。

```
 gtk_statusbar_pop(statusbar, 0);
```

这段代码功能是清除了先前的任何一些状态栏中的信息。

```
 gtk_text_buffer_get_iter_at_mark(buffer, &iter,
gtk_text_buffer_get_insert(buffer));

 row = gtk_text_iter_get_line(&iter);

 col = gtk_text_iter_get_line_offset(&iter);
```

显然上面的代码是在获取当前所处于的行号与列号。

```
 msg = g_strdup_printf("Col %d Ln %d", col+1, row+1);
```

上面的代码准备好，状态栏中显示出来的行号与列号的内容。

```
 gtk_statusbar_push(statusbar, 0, msg);
```

然后，我们就在状态栏上显示文本。

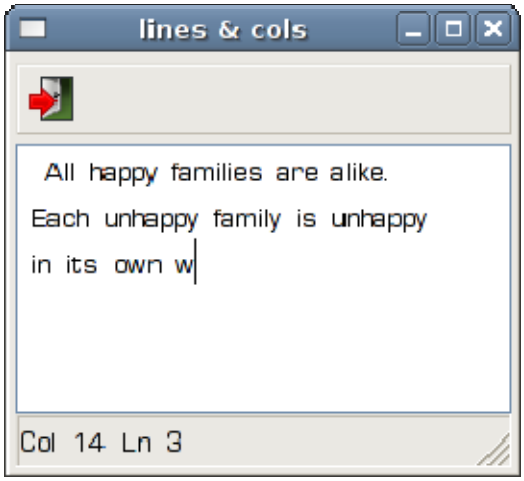

Figure: Lines & Columns

监测& 突显（Search & Highlight）

在接下来的例子中，我们将要在 **GtkTextBuffer** 中做一些监测的工作。我们还将把一些文本的内容进行“突显”处理。

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

#include <gdk/gdkkeysyms.h>

gboolean key_pressed(GtkWidget * window, GdkEventKey* event,
GtkTextBuffer *buffer) {

  GtkTextIter start_sel, end_sel;

  GtkTextIter start_find, end_find;

  GtkTextIter start_match, end_match;

  gboolean selected;

  gchar *text;

  if ((event->type == GDK_KEY_PRESS) && (event->state &
GDK_CONTROL_MASK)) {
```

```
    switch (event->keyval) {

      case GDK_m :

      selected = gtk_text_buffer_get_selection_bounds(buffer,
&start_sel, &end_sel);

      if (selected) {

        gtk_text_buffer_get_start_iter(buffer, &start_find);

        gtk_text_buffer_get_end_iter(buffer, &end_find);

        gtk_text_buffer_remove_tag_by_name(buffer, "gray_bg",
&start_find, &end_find);

        text = (char *) gtk_text_buffer_get_text(buffer, &start_sel,
&end_sel, FALSE);

        while ( gtk_text_iter_forward_search(&start_find, text,
GTK_TEXT_SEARCH_TEXT_ONLY | GTK_TEXT_SEARCH_VISIBLE_ONLY,
&start_match, &end_match, NULL) ) {

          gtk_text_buffer_apply_tag_by_name(buffer, "gray_bg",
&start_match, &end_match);

          int offset = gtk_text_iter_get_offset(&end_match);

          gtk_text_buffer_get_iter_at_offset(buffer, &start_find,
offset);

        }

        g_free(text);

      }

      break;

      case GDK_r:
```

```
        gtk_text_buffer_get_start_iter(buffer, &start_find);

        gtk_text_buffer_get_end_iter(buffer, &end_find);

        gtk_text_buffer_remove_tag_by_name(buffer, "gray_bg",
&start_find, &end_find);

      break;

    }

  }

  return FALSE;

}

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

  GtkWidget *window;

  GtkWidget *view;

  GtkWidget *vbox;

  GtkTextBuffer *buffer;

  GtkTextIter start, end;

  GtkTextIter iter;

  gtk_init(&argc, &argv);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

  gtk_window_set_default_size(GTK_WINDOW(window), 250, 200);
```

```
  gtk_window_set_title(GTK_WINDOW(window), "Search & Highlight");

  gtk_container_set_border_width(GTK_CONTAINER(window), 5);

  GTK_WINDOW(window)->allow_shrink = TRUE;

  vbox = gtk_vbox_new(FALSE, 0);

  view = gtk_text_view_new();

  gtk_widget_add_events(view, GDK_BUTTON_PRESS_MASK);

  gtk_box_pack_start(GTK_BOX(vbox), view, TRUE, TRUE, 0);

  buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(view));

  gtk_text_buffer_create_tag(buffer, "gray_bg", "background", "gray",
NULL);

  gtk_container_add(GTK_CONTAINER(window), vbox);

  g_signal_connect_swapped(G_OBJECT(window), "destroy",
G_CALLBACK(gtk_main_quit), G_OBJECT(window));

  g_signal_connect(G_OBJECT(window), "key-press-event",
G_CALLBACK(key_pressed), buffer);

  gtk_widget_show_all(window);

  gtk_main();

  return 0;

}
```

在我们的示例中，我们用到了键盘的快捷键。Ctrl + M 是用来突显我们当前所选取的文本内容。Ctrl + R 则是用来取消上面的操作。

```
 gtk_text_buffer_create_tag(buffer, "gray_bg", "background", "gray",
NULL);
```

我们在例子中会再次用到 **GtkTextTag** 。这个标记可以使文本的背景反白。

```
 selected = gtk_text_buffer_get_selection_bounds(buffer, &start_sel,
&end_sel);
```

这里我们得到我们选中的文本所具有的起始和终点位置。

```
 gtk_text_buffer_get_start_iter(buffer, &start_find);

 gtk_text_buffer_get_end_iter(buffer, &end_find);
```

我们得到了文本缓冲区（text buffer）的起始和终点位置。

```
 gtk_text_buffer_remove_tag_by_name(buffer, "gray_bg", &start_find,
&end_find);
```

上面就是，把先前的标记去处。

```
 text = (char *) gtk_text_buffer_get_text(buffer, &start_sel, &end_sel,
FALSE);
```

接着我们得到了所选择的文本内容，我们将要进行监测。

```
 while ( gtk_text_iter_forward_search(&start_find, text,

         GTK_TEXT_SEARCH_TEXT_ONLY |

         GTK_TEXT_SEARCH_VISIBLE_ONLY,

         &start_match, &end_match, NULL) ) {

   gtk_text_buffer_apply_tag_by_name(buffer, "gray_bg", &start_match,
&end_match);

   int offset = gtk_text_iter_get_offset(&end_match);

   gtk_text_buffer_get_iter_at_offset(buffer, &start_find, offset);

 }
```

这段代码将检测所有我们所选择的文本后的所发生的事件，一旦发现与我们定义的内容有匹配就应用我们设定好的标记。在匹配工作完成之后，单词的尾端将将被成下次监视操作的首端。

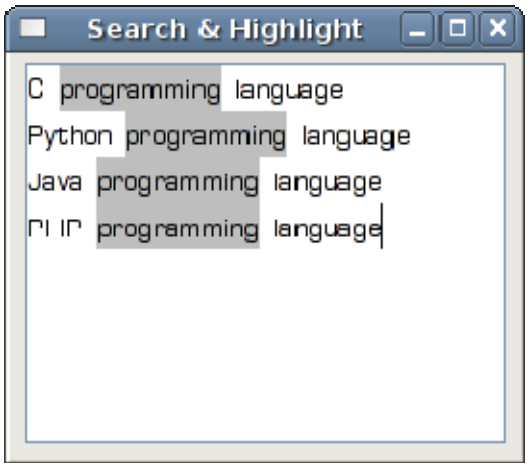

Figure: Search & Highlight

GTK+中的通用构件（Custom GTK+ widget）

在本章节，也就是本教程最后一章啦！我们将向大家战士如何去自己 DIY 一个通用的构件。在内容里我们还将用到 Cairo 图形函数工具库

CPU widget

在接下来的示例中我们将一步一步地制作一个“CPU 构件”。

```
/* cpu.h */

#ifndef __CPU_H

#define __CPU_H

#include <gtk/gtk.h>

#include <cairo.h>

G_BEGIN_DECLS

#define GTK_CPU(obj) GTK_CHECK_CAST(obj, gtk_cpu_get_type (), GtkCpu)

#define GTK_CPU_CLASS(klass) GTK_CHECK_CLASS_CAST(klass,
gtk_cpu_get_type(), GtkCpuClass)

#define GTK_IS_CPU(obj) GTK_CHECK_TYPE(obj, gtk_cpu_get_type())
```

```
typedef struct _GtkCpu GtkCpu;

typedef struct _GtkCpuClass GtkCpuClass;

struct _GtkCpu { GtkWidget widget; gint sel;};

struct _GtkCpuClass { GtkWidgetClass parent_class;};

GtkType gtk_cpu_get_type(void);

void gtk_cpu_set_sel(GtkCpu *cpu, gint sel);

GtkWidget * gtk_cpu_new();

G_END_DECLS

#endif /* __CPU_H */

/* cpu.c */

#include "cpu.h"

static void gtk_cpu_class_init(GtkCpuClass *klass);

static void gtk_cpu_init(GtkCpu *cpu);

static void gtk_cpu_size_request(GtkWidget *widget,GtkRequisition
*requisition);

static void gtk_cpu_size_allocate(GtkWidget *widget,GtkAllocation
*allocation);

static void gtk_cpu_realize(GtkWidget *widget);

static gboolean gtk_cpu_expose(GtkWidget *widget, GdkEventExpose
*event);

static void gtk_cpu_paint(GtkWidget *widget);
```

```
static void gtk_cpu_destroy(GtkObject *object);

GtkType

gtk_cpu_get_type(void){

  static GtkType gtk_cpu_type = 0;

  if (!gtk_cpu_type) {

      static const GtkTypeInfo gtk_cpu_info = {

          "GtkCpu",

          sizeof(GtkCpu),

          sizeof(GtkCpuClass),

          (GtkClassInitFunc) gtk_cpu_class_init,

          (GtkObjectInitFunc) gtk_cpu_init,

          NULL,

          NULL,

          (GtkClassInitFunc) NULL

      };

      gtk_cpu_type = gtk_type_unique(GTK_TYPE_WIDGET, &gtk_cpu_info);

  }

  return gtk_cpu_type;

}
```

```
void gtk_cpu_set_state(GtkCpu *cpu, gint num){
    cpu->sel = num;
    gtk_cpu_paint(GTK_WIDGET(cpu));
}
GtkWidget * gtk_cpu_new(){
    return GTK_WIDGET(gtk_type_new(gtk_cpu_get_type()));
}
static void gtk_cpu_class_init(GtkCpuClass *klass){
  GtkWidgetClass *widget_class;
  GtkObjectClass *object_class;
  widget_class = (GtkWidgetClass *) klass;
  object_class = (GtkObjectClass *) klass;
  widget_class->realize = gtk_cpu_realize;
  widget_class->size_request = gtk_cpu_size_request;
  widget_class->size_allocate = gtk_cpu_size_allocate;
  widget_class->expose_event = gtk_cpu_expose;
  object_class->destroy = gtk_cpu_destroy;
}
```

```
static void gtk_cpu_init(GtkCpu *cpu){ cpu->sel = 0;}
```

```
static void gtk_cpu_size_request(GtkWidget *widget, GtkRequisition
*requisition){

  g_return_if_fail(widget != NULL);

  g_return_if_fail(GTK_IS_CPU(widget));

  g_return_if_fail(requisition != NULL);

  requisition->width = 80;

  requisition->height = 100;

}
```

```
static void gtk_cpu_size_allocate(GtkWidget *widget, GtkAllocation
*allocation){
```

```
  g_return_if_fail(widget != NULL);

  g_return_if_fail(GTK_IS_CPU(widget));

  g_return_if_fail(allocation != NULL);

  widget->allocation = *allocation;

  if (GTK_WIDGET_REALIZED(widget)) {


gdk_window_move_resize(widget->window,allocation->x,allocation->y,
allocation->width, allocation->height);

   }

}
```

```
static void gtk_cpu_realize(GtkWidget *widget){
```

```
  GdkWindowAttr attributes;

  guint attributes_mask;

  g_return_if_fail(widget != NULL);

  g_return_if_fail(GTK_IS_CPU(widget));

  GTK_WIDGET_SET_FLAGS(widget, GTK_REALIZED);

  attributes.window_type = GDK_WINDOW_CHILD;

  attributes.x = widget->allocation.x;

  attributes.y = widget->allocation.y;

  attributes.width = 80;

  attributes.height = 100;

  attributes.wclass = GDK_INPUT_OUTPUT;

  attributes.event_mask = gtk_widget_get_events(widget) |
GDK_EXPOSURE_MASK;
```

```
  attributes_mask = GDK_WA_X | GDK_WA_Y;
```

```
  widget->window = gdk_window_new(gtk_widget_get_parent_window
(widget),&attributes, attributes_mask);

  gdk_window_set_user_data(widget->window, widget);

  widget->style = gtk_style_attach(widget->style, widget->window);

  gtk_style_set_background(widget->style,
widget->window,GTK_STATE_NORMAL);

}
```

```
static gboolean gtk_cpu_expose(GtkWidget *widget, GdkEventExpose
*event){

  g_return_val_if_fail(widget != NULL, FALSE);

  g_return_val_if_fail(GTK_IS_CPU(widget), FALSE);

  g_return_val_if_fail(event != NULL, FALSE);

  gtk_cpu_paint(widget);

  return FALSE;

}

static void gtk_cpu_paint(GtkWidget *widget){

  cairo_t *cr;

  cr = gdk_cairo_create(widget->window);

  cairo_translate(cr, 0, 7);

  cairo_set_source_rgb(cr, 0, 0, 0);

  cairo_paint(cr);

  gint pos = GTK_CPU(widget)->sel;

  gint rect = pos / 5;

  cairo_set_source_rgb(cr, 0.2, 0.4, 0);

  gint i;

  for ( i = 1; i <= 20; i++) {
```

```
        if (i > 20 - rect) {
            cairo_set_source_rgb(cr, 0.6, 1.0, 0);
        } else { cairo_set_source_rgb(cr, 0.2, 0.4, 0); }
        cairo_rectangle(cr, 8, i*4, 30, 3);
        cairo_rectangle(cr, 42, i*4, 30, 3);
        cairo_fill(cr);
    }
    cairo_destroy(cr);
}
static void gtk_cpu_destroy(GtkObject *object){
    GtkCpu *cpu;
    GtkCpuClass *klass;
    g_return_if_fail(object != NULL);
    g_return_if_fail(GTK_IS_CPU(object));
    cpu = GTK_CPU(object);
    klass = gtk_type_class(gtk_widget_get_type());
    if (GTK_OBJECT_CLASS(klass)->destroy) {
        (* GTK_OBJECT_CLASS(klass)->destroy) (object);
    }
}
```

```
/* main.c */

#include "cpu.h"

static void set_value(GtkWidget * widget, gpointer data){
  GdkRegion *region;
  GtkRange *range = (GtkRange *) widget;
  GtkWidget *cpu = (GtkWidget *) data;
  GTK_CPU(cpu)->sel = gtk_range_get_value(range);
  region = gdk_drawable_get_clip_region(cpu->window);
  gdk_window_invalidate_region(cpu->window, region, TRUE);
  gdk_window_process_updates(cpu->window, TRUE);
}
int main (int argc, char ** argv){
  GtkWidget *window;

  GtkWidget *cpu;

  GtkWidget *fixed;
  GtkWidget *scale;
  gtk_init(&argc, &argv);
  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title(GTK_WINDOW(window), "CPU widget");
```

```
    gtk_window_set_position(GTK_WINDOW(window), GTK_WIN_POS_CENTER);

    gtk_window_set_default_size(GTK_WINDOW(window), 200, 180);

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

    fixed = gtk_fixed_new();

    gtk_container_add(GTK_CONTAINER(window), fixed);

    cpu = gtk_cpu_new();


    gtk_fixed_put(GTK_FIXED(fixed), cpu, 30, 40);


    scale = gtk_vscale_new_with_range(0.0, 100.0, 1.0);

    gtk_range_set_inverted(GTK_RANGE(scale), TRUE);

    gtk_scale_set_value_pos(GTK_SCALE(scale), GTK_POS_TOP);

    gtk_widget_set_size_request(scale, 50, 120);

    gtk_fixed_put(GTK_FIXED(fixed), scale, 130, 20);

    g_signal_connect(G_OBJECT(scale), "value_changed",
G_CALLBACK(set_value), (gpointer) cpu);

    gtk_widget_show(cpu);

    gtk_widget_show(fixed);

    gtk_widget_show_all(window);

    gtk_main();

    return 0;
```

```
}
```

你可以看到上面的完整代码，其实在这里我们所制作的 CPU 构件也是属于一种 **GtkWidget**，我们利用了 Cairo API 来绘制我们所需要的效果。我们绘制了一个黑色的背景还有 40 个小的长方形。这个小的长方形被用两种颜色进行绘制：深绿色和翠绿色。 **GtkVScale** 构件用来控制翠绿色的长方形框在构件上绘制的个数。

这个示例第一眼看起来或许还满复杂的。-_-!!但是说实话，并不是想象中的那么复杂。大多数的代码都是样板代码，当我们要尝试去制作了一新的构件的时候，很多代码都是重复的。

绘制图形的工作都会在功能函数 **gtk_cpu_paint()**中进行。

```
  cairo_t *cr;

  cr = gdk_cairo_create(widget->window);

  cairo_translate(cr, 0, 7);

  cairo_set_source_rgb(cr, 0, 0, 0);

  cairo_paint(cr);
```

按照惯例，我们生成了一个 cairo context。我们还设置了大小。并在接下来还为这个构件设置了背景色——黑色。

```
gint pos = GTK_CPU(widget)->sel;

gint rect = pos / 5;
```

这里我们用到了变量 sel 中的数字。 她其实是从滑块构件（scale widget）中的当前位置获得的。那个滑块共有 100 个数字。我们通过滑块当前的位置所对应数字值换算出要有多少个翠绿色的小长方形，然后我们进行绘制操作。

```
gint i;

for ( i = 1; i <= 20; i++) {

    if (i > 20 - rect) {

        cairo_set_source_rgb(cr, 0.6, 1.0, 0);

    } else { cairo_set_source_rgb(cr, 0.2, 0.4, 0); }
```

```
    cairo_rectangle(cr, 8, i*4, 30, 3);

    cairo_rectangle(cr, 42, i*4, 30, 3);

    cairo_fill(cr);

}
```

依据我们所需要的“跨度”数据。我们绘制了一共 40 个小长方形包括深绿色和翠绿色在内。值得醒一下的是，我们画长方形的顺序，在 GTK+ 系统中是从上往下来的，所以换算时要小心。

```
GtkRange *range = (GtkRange *) widget;

GtkWidget *cpu = (GtkWidget *) data;

GTK_CPU(cpu)->sel = gtk_range_get_value(range);
```

在函数 **set_value()** 中，我们把滑块的当前值传递给 CPU 构件。

```
GdkRegion *region;

...

region = gdk_drawable_get_clip_region(cpu->window);

gdk_window_invalidate_region(cpu->window, region, TRUE);

gdk_window_process_updates(cpu->window, TRUE);
```

上面的这段代码致使 CPU 所在的构件窗口失效，并进行自我刷新的操作；这样我们就可以实现动态的效果啦`~~

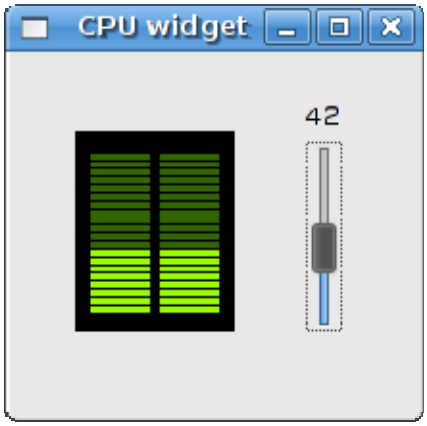

Figure: CPU widget