GTK+ 2.0 教程

版本号： V_0.1.0 2002 年 6 月 25 日

本文是有关通过 C 语言接口使用 GTK (the GIMP Toolkit) 的教程。

Table of Contents

简介

GTK (GIMP Toolkit) 是一套用于创建图形用户界面的工具包。它遵循 LGPL 许可证，所以你可以用它来开发开源软件、自由软件，甚至是封闭源代码的商业软件，而不用花费任何钱来购买许可证和使用权。

GTK 被称为 GIMP 工具包是因为最初写它是用来开发 GIMP (GNU 图像处理程序) 的，但是它现在已经被用于很多软件项目了，包括 GNOME (GNU 网络对象模型环境)。GTK 是在 GDK (GIMP Drawing Kit) 和 gdk-pixbuf 的基础上建立起来的，GDK 基本上是对访问窗口的底层函数 (在 X 窗口系统中是 Xlib) 的一层封装，gdk-pixbuf 是一个用于客户端图像处理的库。

GTK 的创建者是：

- Peter Mattis petm@xcf.berkeley.edu
- Spencer Kimball spencer@xcf.berkeley.edu
- Josh MacDonald jmacd@xcf.berkeley.edu

GTK 的当前维护者是：

- Owen Taylor otaylor@redhat.com
- Tim Janik timj@gtk.org

GTK 实质上是一个面向对象的应用程序接口 (API)。尽管完全用 C 写成的，但它是基于类和回调函数 (指向函数的指针) 的思想实现的。

还 有一个名为 GLib 的第三个组件，包含一些标准函数的替代函数，以及一些处理链表等数据结构的函数等。这些替代函数被用来增强 GTK 的可移植性，因为它们所实现的一些函数在其它 Unix 系统上未实现或不符合标准，比如 g_strerror()。一些是对 libc 的对应函数的增强，比如 g_malloc() 具有增强的调试功能。

在 2.0 版中，GLib 又加入这样一些新内容：构成 GTK 类层次基础的类型系统 (type system)，在 GTK 中广泛使用的信号系统，对各种不同平台的线程 API 进行抽象而得的一个线程 API，以及一个加载模块的工具。

作为最后一个组件，GTK 使用了 Pango 库来处理国际化文字输出。

本 教程讲述 GTK 的 C 接口。还有许多其它语言的 GTK 绑定如 C++、Perl、Python、TOM、Ada95、Objective C、Free Pascal、Eiffel、Java 和 C#。如果你想使用 GTK 其它语言的绑定，请先查看该绑定的文档。有时这些文档会讲一些重要的概念，然后你再来参考本教程。还有一些跨平台的 API (如 wxWindows 和 V)，它们把 GTK 作为一个支持的平台。同样，先参考它们的文档。

如果你用 C++ 来开发 GTK 应用程序，有以下几点需要注意。已有一个 GTK 的 C++ 绑定叫做 GTK-- (译者注：现在叫做 gtkmm)，提供一个更符合 C++ 规范的接口，你可以先看看这个接口。如果你由于种种原因不喜欢这种方法，还有另外两种使用 GTK 的方法。首先，你可以只使用 C++ 中的 C 子集来调用 GTK，这样就可以使用本教程描述的 C 接口。其次，你可以用下述方法同时使用 GTK 和 C++：把所用的回调函数定义为 C++ 类中的静态成员函数，然后仍然使用 C 接口来调用 GTK。如果你选择后一种方法，你可以把指向要操作的对象的指针 (即所谓的 "this")作为回调函数的 data 参数。选择哪一种方法仅仅是个人的喜好问题，因为不管用哪一种方法，你都会得到 C++ 和 GTK。它们都不需要特殊的预处理程序，因此你可以同时使用标准 C++ 和 GTK。

本教程试图尽可能详细地描述 GTK，但是肯定不能面面俱到。本教程假设你能够较好的理解 C 语言，并且了解怎样编写一个 C 程序。有 X 编程经验会很有帮助，但不是必要条件。如果 GTK 是你学习的第一个构件工具包，请告诉我们你怎样找到这个教程，以及学习时有什么困难。还有其它一些语言的绑定，如 C++、Objective C、ADA、Guile 等，但我不了解这些。

本教程仍在不断完善中。请到 http://www.gtk.org/ 查看更新情况。

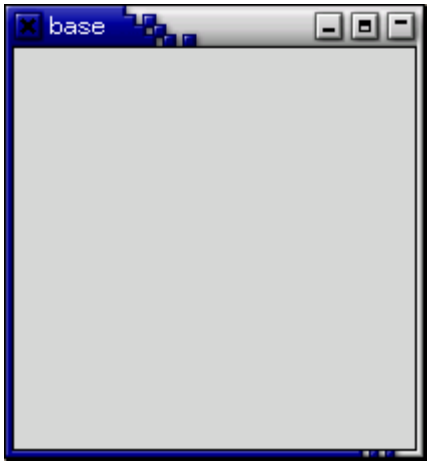
我非常乐意听到你在使用本教程学习 GTK 时遇到的各种困难，并欢迎对怎样改进此文档提出建议。更多信息请参阅投稿这一章。

从这里开始

你首先做的第一件事，当然是下载 GTK 源程序，并安装它。你总是能从 ftp.gtk.org 得到最新版本。你也可以在 http://www.gtk.org/ 上查看其它 GTK 源程序的信息。GTK 使用 GNU autoconf 配置。解压缩后，输入 ./configure --help 查看选项列表

GTK 源码发布包中包含教程中所有示例的代码，每个示例中包含有 Makefiles 文件，用以方便编译。

一开始介绍 GTK，我们会尽可能从简单的程序开始。这个程序创建 200x200 大小的窗口，没有办法退出，除非你从 shell 中将它杀掉。




```
#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;
}
```

你可以用 gcc 编译上面的程序：

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

不常用的编译参数在下面 编译 Hello World 程序 中解释。

所有程序应该包含 gtk/gtk.h，其中声明了变量、函数以及数据结构等，这些东西会在你的程序中使用。

下一行：

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

这 个函数 gtk_init(gint *argc, gchar ***argv) 会在每个 GTK 应用程序中调用。该函数设定了默认的视频(visual)和颜色映射模式(color map)，接着会调用函数 gdk_init(gint *argc, gchar ***argv)。该函数初始化要使用的库，设定默认的信号处理，并检查传递给你的程序的命令行参数，寻找下列之一：

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

这些参数将会从参数表中删除，留下它不能识别的给你的程序解析或忽略。这就创建了可以被所有 GTK 程序接受的一组标准参数。

下面两行程序会创建并显示一个窗口

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

GTK_WINDOW_TOPLEVEL 参数指我们要使用窗口管理器来修饰和放置窗口。这里不会创建一个 0x0 大小的窗口，一个没有子构件的窗口默认大小设置为 200x200，这样你仍然能操作它。

gtk_widget_show() 函数让 GTK 知道，我们已经设置完构件的属性，可以显示它了。

最后一行进入 GTK 主处理循环。

```
gtk_main ();
```

gtk_main() 是另一个可以在每个 GTK 程序中见到的函数调用。当程序运行到这里，GTK 会“睡着”等待 X 事件 (如按钮或键盘按下)、超时(timeouts)或文件 IO 通知发生。在我们的示例中，事件被忽略。

用 GTK 来写 Hello World

好，现在来写一个只有一个按钮构件的程序，这是一个标准的 GTK Hello World。


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

/* 这是一个回调函数。data 参数在本示例中被忽略。
 * 后面有更多的回调函数示例。*/
void hello( GtkWidget *widget,
            gpointer   data )
{
    g_print ("Hello World\n");
}

gint delete_event( GtkWidget *widget,
                   GdkEvent  *event,
                   gpointer   data )
{
    /* 如果你的 "delete_event" 信号处理函数返回 FALSE，GTK 会发出 "destroy" 信号。
     * 返回 TRUE，你不希望关闭窗口。
     * 当你想弹出“你确定要退出吗?”对话框时它很有用。*/

    g_print ("delete event occurred\n");
```

```
    /* 改 TRUE 为 FALSE 程序会关闭。*/

    return TRUE;
}

/* 另一个回调函数 */
void destroy( GtkWidget *widget,
              gpointer   data )
{
    gtk_main_quit ();
}

int main( int   argc,
          char *argv[] )
{
    /* GtkWidget 是构件的存储类型 */
    GtkWidget *window;
    GtkWidget *button;

    /* 这个函数在所有的 GTK 程序都要调用。参数由命令行中解析出来并且送到该程序中*/
    gtk_init (&argc, &argv);

    /* 创建一个新窗口 */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    /* 当窗口收到 "delete_event" 信号 (这个信号由窗口管理器发出，通常是“关闭”
     * 选项或是标题栏上的关闭按钮发出的)，我们让它调用在前面定义的 delete_event() 函数。
     * 传给回调函数的 data 参数值是 NULL，它会被回调函数忽略。*/
    g_signal_connect (G_OBJECT (window), "delete_event",
                      G_CALLBACK (delete_event), NULL);

    /* 在这里我们连接 "destroy" 事件到一个信号处理函数。
     * 对这个窗口调用 gtk_widget_destroy() 函数或在 "delete_event" 回调函数中返回 FALSE 值
     * 都会触发这个事件。*/
    g_signal_connect (G_OBJECT (window), "destroy",
                      G_CALLBACK (destroy), NULL);

    /* 设置窗口边框的宽度。*/
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* 创建一个标签为 "Hello World" 的新按钮。*/
    button = gtk_button_new_with_label ("Hello World");

    /* 当按钮收到 "clicked" 信号时会调用 hello() 函数，并将 NULL 传给
     * 它作为参数。hello() 函数在前面定义了。*/
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (hello), NULL);

    /* 当点击按钮时，会通过调用 gtk_widget_destroy(window) 来关闭窗口。
     * "destroy" 信号会从这里或从窗口管理器发出。*/
    g_signal_connect_swapped (G_OBJECT (button), "clicked",
                              G_CALLBACK (gtk_widget_destroy),
                              window);

    /* 把按钮放入窗口 (一个 gtk 容器) 中。*/
    gtk_container_add (GTK_CONTAINER (window), button);

    /* 最后一步是显示新创建的按钮和窗口 */
    gtk_widget_show (button);
    gtk_widget_show (window);

    /* 所有的 GTK 程序必须有一个 gtk_main() 函数。程序运行停在这里
     * 等待事件 (如键盘事件或鼠标事件) 的发生。*/
    gtk_main ();
```

```
    return 0;
}
```

编译 Hello World 程序

编译命令是：

```
gcc -Wall -g helloworld.c -o helloworld `pkg-config --cflags gtk+-2.0` \
`pkg-config --libs gtk+-2.0`
```

这里使用了程序 pkg-config，可以从 www.freedesktop.org 得到。这个程序读取 GTK 附带的 .pc 文件来决定编译 GTK 程序需要的编译选项。pkg-config --cflags gtk+-2.0 列出 include 目录，pkg-config --libs gtk+-2.0 列出编译连接库，也可以合在一起，像这样：pkg-config --cflags --libs gtk+-2.0。

注意上面编译命令中使用的单引号类型是很重要的。(译者注：这里使用了“命令替换”。命令替换(command substitution)使得可以捕获一个命令的输出而在另一个命令中替换它。这个单引号不是回车键左边的那个，而是 ESC 键下面的那个。)

连接时常用的库：

- GTK 库(-lgtk)，构件库，基于 GDK。
- GDK 库(-lgdk)，Xlib 库的封装(wrapper)。
- gdk-pixbuf 库(-lgdk_pixbuf)，图像处理库。
- Pango 库(-lpango)，处理国际化文本。
- gobject 库(-lgobject)，包含作为 GTK 基础的类型系统。
- gmodule 库(-lgmodule)，动态运行库。
- GLib 库(-lglib)，包含各种函数；这个示例里只用了 g_print()。GTK 是基于 GLib 的，因此你总需要这个库。详见 GLib 这一章。
- Xlib 库(-lX11)，GDK 要使用。
- Xext 库(-lXext)，包含共享内存位图和其它 X 扩展。
- math 库(-lm)，数学库，这个被 GTK 因各种目的而使用。

信号和回调函数的原理

> 在 2.0 版，信号系统已从 GTK 移到 GLib，因此在函数和类型的说明中有前缀 "g_" 而不是 "gtk_"。我们不打算介绍 GLib 2.0 信号系统相对 GTK 1.2 信号系统扩展的细节。

在我们详细分析 *helloworld* 程序之前，我们会讨论信号和回调函数。GTK 是一个事件驱动的工具包，意味着它会等在 gtk_main() 那里，直到下一个事件发生，才把控制权传给适当的函数。

控 制权的传递是使用“信号”的办法来完成的。(注意这里的信号并不等同于 Unix 系统里的信号，并且也不是用它们实现的，虽然使用的术语是一样的。) 当一个事件发生时，如按一下鼠标键，所按的构件会“发出”适当的信号。这就是 GTK 的工作机制。有所有构件都继承的信号，如 "destroy"，有构件专有的信号，如开关 (toggle) 按钮发出的 "toggled" 信号。

要使一个按钮执行一个动作，我们需设置信号和信号处理函数之间的连接。可以这样使用函数来设置连接：

```
gulong g_signal_connect( gpointer      *object,
                         const gchar   *name,
                         GCallback     func,
                         gpointer      func_data );
```

第一个参数是要发出信号的构件，第二个参数是你想要连接的信号的名称，第三个参数是信号被捕获时所要调用的函数，第四个参数是你想传递给这个函数的数据。

第三个参数指定的函数叫做回调函数，一般为下面的形式：

```
void callback_func( GtkWidget *widget,
                    gpointer   callback_data );
```

第一个参数是一个指向发出信号的构件的指针，第二个参数是一个指向数据的指针，就是上面 g_signal_connect() 函数的最后一个参数传进来的数据。

注意上面回调函数的声明只是一般的形式，有些构件的特殊信号会用不同的调用参数。

另一个在 *helloworld* 示例中使用的调用，是：

```
gulong g_signal_connect_swapped( gpointer     *object,
                                 const gchar  *name,
                                 GCallback    func,
                                 gpointer     *slot_object );
```

g_signal_connect_swapped() 和 g_signal_connect() 相同，只是回调函数只用一个参数，一个指向 GTK 对象的指针。所以当使用这个函数连接信号时，回调函数应该是这样的形式

```
void callback_func( GtkObject *object );
```

这个对象通常是一个构件。然而我们一般不用函数 g_signal_connect_swapped() 设置回调。它们常用来调用一个只接受一个单独的构件或者对象作为参数的 GTK 函数，如同我们的 *helloworld* 示例中那样。

拥 有两个函数来设置信号连接的目的只是为了允许回调函数有不同数目的参数。GTK 库中许多函数仅接受一个单独的构件指针作为其参数，所以对于这些函数你要用 g_signal_connect_swapped()，然而对你自己定义的函数，你可能需要附加的数据提供给你的回调函数。

事件

除有前面描述的信号机制外，还有一套 *events* 反映 X 事件机制。回调函数可以与这些事件连接。这些事件是：

- event
- button_press_event
- button_release_event
- scroll_event
- motion_notify_event
- delete_event
- destroy_event
- expose_event
- key_press_event
- key_release_event
- enter_notify_event
- leave_notify_event
- configure_event
- focus_in_event
- focus_out_event
- map_event
- unmap_event
- property_notify_event
- selection_clear_event
- selection_request_event
- selection_notify_event
- proximity_in_event
- proximity_out_event
- visibility_notify_event
- client_event
- no_expose_event
- window_state_event

为了连接一个回调函数到这些事件之一，你使用函数 g_signal_connect()，像前面介绍的一样，用上面事件名之一作为 name 参数。事件的回调函数与信号的回调函数有一点点不同：

```
gint callback_func( GtkWidget *widget,
                    GdkEvent  *event,
```

```
                    gpointer   callback_data );
```

GdkEvent 是一个 C 联合结构，它的类型依赖于上述事件中的哪个事件发生了。为了让我们得知发生了哪个事件，每个可能的类型都有一个 type 成员来反映发生的事件。事件结构的其它部分将依赖于这个事件的类型。类型的可能的值有：

```
GDK_NOTHING
GDK_DELETE
GDK_DESTROY
GDK_EXPOSE
GDK_MOTION_NOTIFY
GDK_BUTTON_PRESS
GDK_2BUTTON_PRESS
GDK_3BUTTON_PRESS
GDK_BUTTON_RELEASE
GDK_KEY_PRESS
GDK_KEY_RELEASE
GDK_ENTER_NOTIFY
GDK_LEAVE_NOTIFY
GDK_FOCUS_CHANGE
GDK_CONFIGURE
GDK_MAP
GDK_UNMAP
GDK_PROPERTY_NOTIFY
GDK_SELECTION_CLEAR
GDK_SELECTION_REQUEST
GDK_SELECTION_NOTIFY
GDK_PROXIMITY_IN
GDK_PROXIMITY_OUT
GDK_DRAG_ENTER
GDK_DRAG_LEAVE
GDK_DRAG_MOTION
GDK_DRAG_STATUS
GDK_DROP_START
GDK_DROP_FINISHED
GDK_CLIENT_EVENT
GDK_VISIBILITY_NOTIFY
GDK_NO_EXPOSE
GDK_SCROLL
GDK_WINDOW_STATE
GDK_SETTING
```

所以，连接一个回调函数到这些事件之一，我们会这样用：

```
g_signal_connect (G_OBJECT (button), "button_press_event",
                  G_CALLBACK (button_press_callback), NULL);
```

这里假定 button 是一个按钮构件。现在，当鼠标位于按钮上并按一下鼠标时，函数 button_press_callback() 会被调用。这个函数应该声明为：

```
static gint button_press_callback( GtkWidget      *widget,
                                   GdkEventButton *event,
                                   gpointer        data );
```

注意，我们可以把第二个参数类型声明为 GdkEventButton，因为我们知道哪个类型的事件会发生。

这个函数的返回值指示这个事件是否应该由 GTK 事件处理机制做进一步的传播。返回 TRUE 指示这个事件已经处理了，且不应该做进一步传播。返回 FALSE 继续正常的事件处理。详见高级事件和信号处理这一章。

GdkEvent 数据类型详情请参见附录 GDK 事件类型。

GDK 选中区和拖放的接口函数也发出许多事件，在 GTK 中用信号来反映。下列信号的内容详见源构件上的信号和目的构件上的信号这两章：

- selection_received
- selection_get
- drag_begin_event
- drag_end_event

- drag_data_delete
- drag_motion
- drag_drop
- drag_data_get
- drag_data_received

Hello World 详解

现在我们知道基本理论了，让我们来详细分析 *helloworld* 示例程序。

这是按钮被点击时要调用的回调函数。在这个示例中我们忽略了参数 widget 和 data，但是使用这些参数也不难。下一个示例会使用 data 参数来告诉我们哪个按钮被按下了。

```
void hello( GtkWidget *widget,
            gpointer   data )
{
    g_print ("Hello World\n");
}
```

接下来的一个回调函数有点特殊。"delete_event" 在窗口管理器发送这个事件给应用程序时发生。我们在这里可以选择对这些事件做什么。可以忽略它们，可以做一点响应，或是简单地退出程序。

这个回调函数返回的值让 GTK 知道该如何去做。返回 TRUE，让它知道我们不想让 "destroy" 信号被发出，保持程序继续运行。返回 FALSE，我们让 "destroy" 信号发出，这接着会调用 "destroy" 信号处理函数。

```
gint delete_event( GtkWidget *widget,
                   GdkEvent  *event,
                   gpointer   data )
{
    g_print ("delete event occurred\n");

    return TRUE;
}
```

这里是另一个回调函数，它通过调用 gtk_main_quit() 来退出程序。这个函数告诉 GTK 当控制权返回给它时就从 gtk_main 退出。

```
void destroy( GtkWidget *widget,
              gpointer   data )
{
    gtk_main_quit ();
}
```

我假设你知道 main() 函数...是的，像其它程序一样，所有的 GTK 程序有一个 main() 函数。

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

接下来声明两个指向 GtkWidget 类型的结构的指针。它们被用于创建一个窗口和一个按钮。

```
    GtkWidget *window;
    GtkWidget *button;
```

这里又是 gtk_init()。跟前面一样，这个初始化工具包，分析命令行里的参数。它从参数列表中删除任何可以识别的参数，并且修改 argc 和 argv，使这些被删除的参数好像从来就不存在一样，而允许你的程序分析剩余的参数。

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

创建一个新窗口。这个很直观。它为 GtkWidget *window 结构分配了内存，这样 window 现在指向了一个有效的结构。它建立了一个新窗口，但是这个窗口直到在程序后面部分我们调用 gtk_widget_show(window) 后才会显示出来。

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

这 有两个连接一个信号处理函数到一个对象 (本例中，就是 window) 的示例。这里，"delete_event" 和 "destroy" 信号被捕获。当我们用窗口管理器去关闭窗口或调用函数 gtk_widget_destroy() 并将 window 构件作为对象传给它来销毁时，"delete_event" 信号发出。当我们在 "delete_event" 信号处理函数中返回 FALSE 值时，"destroy" 信号发出。G_OBJECT 和 G_CALLBACK 是宏，为我们执行类型转换和检测，同时也增加了代码的可读性。

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

接下来这个函数用于设置容器对象的属性。设置窗口边框宽度为 10 个象素。在设置构件属性这一章还有其它类似函数。

再次，GTK_CONTAINER 也是一个执行类型转换的宏。

```
gtk_container_set_border_width (GTK_CONTAINER (window), 10);
```

这个函数调用创建一个新按钮。在内存中分配空间给一个新的 GtkWidget 结构，初始化它，并使 button 指针指向它。它显示后上面会有个 "Hello World" 标签。

```
button = gtk_button_new_with_label ("Hello World");
```

在 这，我们让这个按钮做一些有用的事。我们给按钮设置信号处理函数，因此当按钮发出 "clicked" 信号时，hello() 函数被调用。我们忽略了 data 参数，简单地传送 NULL 给 hello() 回调函数。显而易见，当我们用鼠标点击按钮时，信号 "clicked" 被发出。

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

我 们也要使用这个按钮退出程序。这将演示 "destroy" 信号怎样由窗口管理器引发，或由我们的程序引发。当我们按下按钮时，和上面一样，它首先调用 hello() 回调函数，然后是这个函数，这依赖于它们被设置连接的顺序。你可以拥有许多回调函数，所有的回调按你设置连接的顺序依次执行。因为 gtk_widget_destroy() 函数只接受 GtkWidget *widget 作为唯一的参数，我们这里用 g_signal_connect_swapped() 函数代替正统的 g_signal_connect()。

```
g_signal_connect_swapped (G_OBJECT (button), "clicked",
                          G_CALLBACK (gtk_widget_destroy),
                          G_OBJECT (window));
```

这是一个组装调用，在组装构件这一章将作深入讲解。不过它相当容易理解。它简单地告诉 GTK 要把按钮放在窗口里，也就是它显示的地方。注意一个 GTK 容器只能包含一个构件。还有其它的构件，在后面介绍，设计为用来以各种方法布局多个构件。

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

一切准备就绪。所有信号处理函数连接好了，按钮也放进了窗口，我们让 GTK 在屏幕上“显示”这些构件。窗口构件最后显示，这样整个窗口会一下弹出，而不是先见到窗口弹出后再见到按钮。虽然这个简单的示例中，你不会注意到。

```
gtk_widget_show (button);

gtk_widget_show (window);
```

接着，当然，我们调用 gtk_main() 函数来等待来自 X 服务器的事件，当这些事件到来时，调用构件发出信号。

```
gtk_main ();
```

最后返回，调用函数 gtk_quit() 后控制权返回到这里。

```
return 0;
```

现 在，当我们用鼠标点击一个 GTK 按钮，构件发出一个 "clicked" 信号。为了让我们利用这个信息，程序设置了一个信号处理器来捕获那个信号，它按我们的选择依次调用函数。在我们的示例中，当按下按钮时，以 NULL 作为参数调用函数 hello()，然后调用该信号的下一个处理函数，该函数调用 gtk_widget_destroy() 函数，把窗口构件作为参数传递给它，销毁窗口构件。这导致窗口发出 "destroy" 信号，它被捕获，并且调用我们的 destroy() 回调函数，简单地退出 GTK。

如果用窗口管理器去关闭窗口，它会引发 "delete_event"。这会调用我们的 "delete_event" 处理函数。如果我们在函数中返回 TRUE，窗口还是留在那里，什么事也不发生。返回 FALSE，会使 GTK 发出 "destroy" 信号，它当然会调用 "destroy" 回调，退出 GTK。

继续

数据类型

你或许发现前述示例中有几个地方需要解释。gint、gchar 等等。你看到的这些被分别定义类型别名(typedefs)到 int 和 char，它们是 GLib 系统的一部分。这用来避免在计算时对简单数据类型的大小(size)的依赖。

一个好的示例是，"gint32" 被定义为任何平台的 32 位整数，无论是 64 位的 alpha 还是 32 位的 i386。该类型定义非常直观。它们都在 glib/glib.h 里定义 (这个文件被 gtk.h 包含了)。

你也将注意到 GTK 有在函数要一个 GtkObject 作为参数时传入 GtkWidget 的能力。GTK 的设计是面向对象的，一个构件是一个对象。

深入探索信号处理函数

让我们来看一下函数 gtk_signal_connect() 的声明。

```
gulong g_signal_connect( gpointer object,
                         const gchar *name,
                         GCallback func,
                         gpointer func_data );
```

注意到返回值的类型 gulong 了吗？这是一个识别你的回调函数的标识。前面讲过，每个信号和每个对象可以有多个回调函数，并且它们会按设置的顺序依次运行。

利用这个标识，你可以用下面的函数从列表中删除这个回调:

```
void g_signal_handler_disconnect( gpointer object,
                                  gulong   id );
```

所以，通过传递你想在上面删除处理函数的构件，以及某个 signal_connect 函数返回的标识，你就可以中断一个信号处理函数的连接。

你也可以用 g_signal_handler_block() 和 g_signal_handler_unblock() 这类函数来暂时断开信号处理函数的连接。

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

void g_signal_handlers_block_by_func( gpointer  object,
                                      GCallback func,
                                      gpointer  data );

void g_signal_handler_unblock( gpointer object,
                               gulong   id );

void g_signal_handlers_unblock_by_func( gpointer  object,
                                        GCallback func,
                                        gpointer  data );
```

改进了的 Hello World

让我们来看一下稍微改进了的 *helloworld*，它对回调作了更好的示范。这也会将我们带入下一个主题，组装构件。

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

/* 我们新改进的回调函数。传递到该函数的数据将打印到标准输出(stdout)。*/
void callback( GtkWidget *widget,
               gpointer   data )
{
    g_print ("Hello again - %s was pressed\n", (gchar *) data);
}

/* 另一个回调函数 */
gint delete_event( GtkWidget *widget,
                   GdkEvent  *event,
                   gpointer   data )
{
    gtk_main_quit ();
    return FALSE;
}

int main( int   argc,
          char *argv[] )
{
    /* GtkWidget 是构件的存储类型 */
    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *box1;

    /* 这个函数在所有的 GTK 程序都要调用。参数由命令行中解析出来并且送到该程序中。*/
    gtk_init (&argc, &argv);

    /* 创建一个新窗口 */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    /* 这是一个新的调用，设置窗口标题为"Hello Buttons!" */
    gtk_window_set_title (GTK_WINDOW (window), "Hello Buttons!");

    /* 在这里我们为 delete_event 设置了一个处理函数来立即退出 GTK。*/
    g_signal_connect (G_OBJECT (window), "delete_event",
                      G_CALLBACK (delete_event), NULL);

    /* 设置窗口边框的宽度。 */
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* 我们创建了一个组装盒。详情参见“组装”章节。
     * 我们看不见组装盒，它仅被作为排列构件的工具。*/
    box1 = gtk_hbox_new (FALSE, 0);

    /* 把组装盒放入主窗口中。*/
    gtk_container_add (GTK_CONTAINER (window), box1);

    /* 创建一个标签为 "Button 1" 的新按钮。*/
    button = gtk_button_new_with_label ("Button 1");

    /* 当按钮被按下时，我们调用 "callback" 函数，并将一个指向 "button 1" 的
     * 指针作为它的参数。*/
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (callback), "button 1");

    /* 代替 gtk_container_add，我们把按钮放入不可见的组装盒，该组合盒已经组
     * 装进窗口中了。*/
    gtk_box_pack_start (GTK_BOX(box1), button, TRUE, TRUE, 0);

    /* 总是记住这一步，它告诉 GTK 我们为这个按钮做的准备工作已经完成了，现
     * 在可以显示它了。*/
    gtk_widget_show (button);
```

```
    /* 同样创建第二个按钮。*/
    button = gtk_button_new_with_label ("Button 2");

    /* 以不同的参数调用相同的回调函数，用指向 "button 2" 的指针代替。*/
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (callback), "button 2");

    gtk_box_pack_start(GTK_BOX (box1), button, TRUE, TRUE, 0);

    /* 显示的顺序并不重要，但我建议最后显示窗口。这样它们会同时显示出来。*/
    gtk_widget_show (button);

    gtk_widget_show (box1);

    gtk_widget_show (window);

    /* 停在这里等待事件发生。*/
    gtk_main ();

    return 0;
}
```

用 和我们第一个示例相同的连接参数来编译这个程序，你会发现这次程序不能退出，你不得不使用窗口管理器或命令行去杀死它。插入第三个"Quit"按钮来退出 程序对读者来说是一个好的练习。你也可能想在读下一章时用这个程序测试 gtk_box_pack_start() 的各种选项。试试改变窗口的大小，并观察其行为。

组装构件

创建一个应用软件的时候，你可能希望在窗口里放置超过一个以上的构件。我们的第一个 *helloworld* 示例仅用了一个构件，因此我们能够简单地使用 gtk_container_add()来“组装”这个构件到窗口中。但当你想要放置更多的构件到一个窗口中时，如何控制各个构件的定位呢？这时就要用到组装(Packing)了。

组装盒的原理

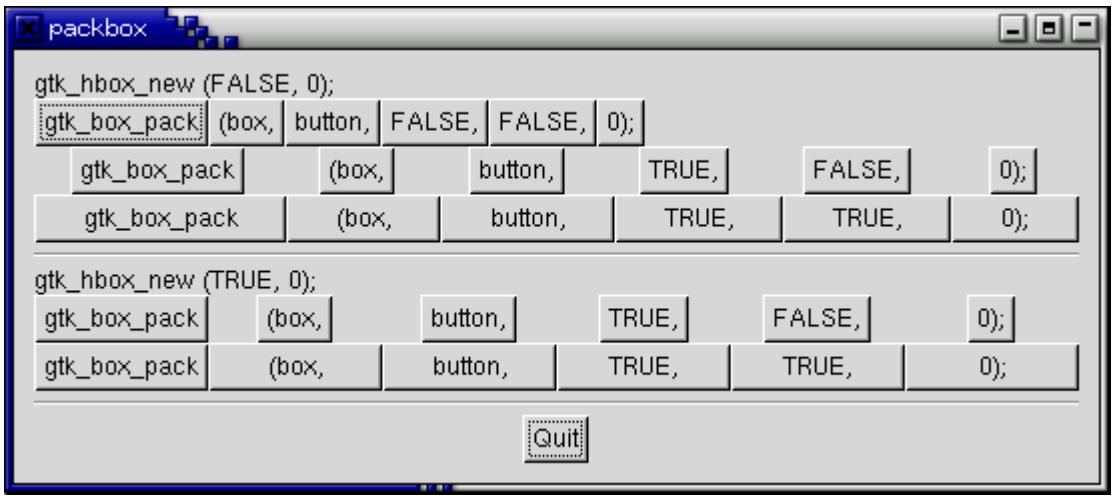
多 数组装是通过创建一些“盒(boxes)”来达成的，这是些不可见的构件容器，它们有两种形式：一种是横向盒(horizontal box)，一种是纵向盒(vertical box)。当我们组装构件到横向盒里时，这些构件就依着我们调用的顺序由左至右或从右到左水平地插入进去。在纵向盒里，则从顶部到底部或相反地组装构件， 你可以使用任意的盒组合，比如盒套盒或者盒挨着盒，用以产生你想要的效果。

要 创建一个新的横向盒我们调用 gtk_hbox_new()，对于纵向盒，用 gtk_vbox_new()。gtk_box_pack_start() 和 gtk_box_pack_end() 函数用来将对象组装到这些容器中。gtk_box_pack_start() 将对象从上到下组装到纵向盒中，或者从左到右组装到横向盒中。gtk_box_pack_end() 则相反，从下到上组装到纵向盒中，或者从右到左组装到横向盒中。使用这些函数允许我们调整自己的构件向左或向右对齐，同时也可以混入一些其它的方法来达到 我们想要的设计效果。在我们的示例中多数使用 gtk_box_pack_start()。被组装的对象可以是另一个容器或构件。事实上，许多构件本身就是容器，包括按钮，只不过我们通常在按钮中只放 入一个标签。

通过使用这些调用，GTK 就会知道要把构件放到哪里去，并且会自动做调整大小及其它美化的事情。至于如何组装你的构件这里还有一些选项。正如你能想到的，在放置和创建构件时，这些方法给了我们很多的弹性。

盒的细节

由于存在这样的弹性，所以在一开始使用 GTK 中的组装盒(packing box)的时候会有点让人迷惑。这里有许多选项，并且它们不容易一眼看出是如何组合在一起的。然而到最后，这里基本上只有五种不同的风格。



每一行包含一个带有若干按钮的横向盒。gtk_box_pack 是组装每个按钮到横向盒(hbox)的简写。每个按钮都是以同样的方式组装到横向盒里的（例如，以同样参数调用 gtk_box_pack_start() 函数）。

这是 gtk_box_pack_start() 函数的声明。

```
void gtk_box_pack_start( GtkBox    *box,
                         GtkWidget *child,
                         gboolean   expand,
                         gboolean   fill,
                         guint      padding );
```

第一个参数是你要把对象组装进去的盒，第二个就是该对象。目前这些对象将都是按钮，即我们要将这些按钮组装到盒中。

gtk_box_pack_start() 和 gtk_box_pack_end() 中的 expand 参数是用来控制构件在盒中是充满所有多余空间这样盒会扩展到充满所有分配给它的空间（TURE）；还是盒收缩到仅仅符合构件的大小（FALSE）。设置 expand 为 FALSE 将允许你向左或向右对齐你的构件。否则，它们会在盒中展开，同样的效果只要用 gkt_box_pack_start() 或 gtk_box_pack_end() 之一就能实现。

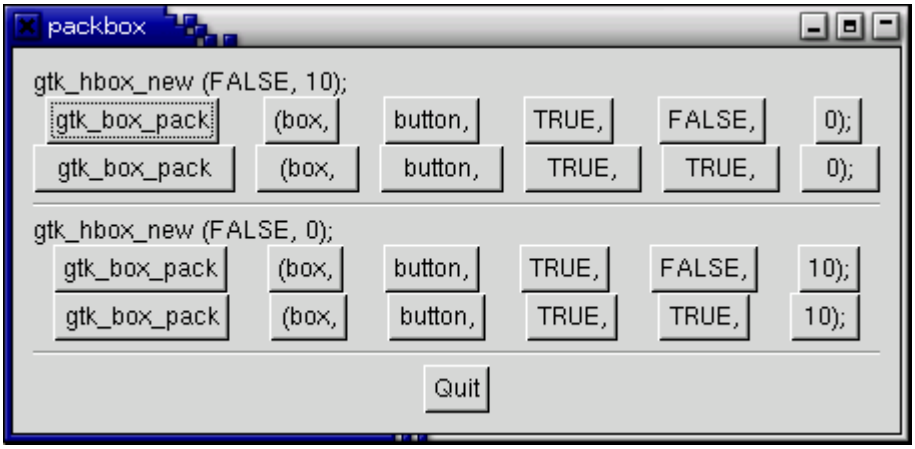
fill 参数在 gtk_box_pack 中控制多余空间是分配给对象本身（TRUE），还是让多余空间围绕在这些对象周围分布（FALSE）。它只有在 expand 参数也为 TRUE 时才会生效。

当创建一个新盒时，函数看起来像下面这样：

```
GtkWidget *gtk_hbox_new ( gboolean homogeneous,
                          gint     spacing );
```

gtk_hbox_new() 的 homogeneous 参数（对于 gtk_vbox_new()也是一样）控制是否盒里的每个对象具有相同的大小（例如，在横向盒中等宽，或在纵向盒中等高）。若它被设置，gtk_box_pack()常规函数的 expand 参数就被忽略了，它本质上总被开启。

spacing（当盒被创建时设置）和 padding（当元素被组装时设置）有什么区别呢？Spacing 是加在对象之间，而 papadding 加在对象的每一边。看下面这张图应该会明白一点：



这面是产生上面这些图片的代码，其中做了不少注释，所以我希望你看下去不会有任何问题。自己编译它并玩玩它吧。

组装示范程序

```
#include <stdio.h>
#include <stdlib.h>
#include "gtk/gtk.h"

gint delete_event( GtkWidget *widget,
                   GdkEvent  *event,
                   gpointer   data )
{
    gtk_main_quit ();
    return FALSE;
}

/* 生成一个填满按钮-标签的横向盒。我们将感兴趣的参数传递进了这个函数。
 * 我们不显示这个盒，但显示它内部的所有东西。 */
GtkWidget *make_box( gboolean homogeneous,
                     gint     spacing,
                     gboolean expand,
                     gboolean fill,
                     guint    padding )
{
    GtkWidget *box;
    GtkWidget *button;
    char padstr[80];

    /* 以合适的 homogeneous 和 spacing 设置创建一个新的横向盒 */
    box = gtk_hbox_new (homogeneous, spacing);

    /* 以合适的设置创建一系列的按钮 */
    button = gtk_button_new_with_label ("gtk_box_pack");
    gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
    gtk_widget_show (button);

    button = gtk_button_new_with_label ("(box,");
    gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
    gtk_widget_show (button);

    button = gtk_button_new_with_label ("button,");
    gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
    gtk_widget_show (button);

    /* 根据 expand 的值创建一个带标签的按钮 */
    if (expand == TRUE)
            button = gtk_button_new_with_label ("TRUE,");
    else
            button = gtk_button_new_with_label ("FALSE,");

    gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
    gtk_widget_show (button);

    /* 这个和上面根据 "expand" 创建按钮一样，不过用
     * 了简化的形式。 */
    button = gtk_button_new_with_label (fill ? "TRUE," : "FALSE,");
    gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
    gtk_widget_show (button);

    sprintf (padstr, "%d);", padding);

    button = gtk_button_new_with_label (padstr);
    gtk_box_pack_start (GTK_BOX (box), button, expand, fill, padding);
    gtk_widget_show (button);

    return box;
```

```
}

int main( int   argc,
          char *argv[])
{
    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *box1;
    GtkWidget *box2;
    GtkWidget *separator;
    GtkWidget *label;
    GtkWidget *quitbox;
    int which;

    /* 初始化   */
    gtk_init (&argc, &argv);

    if (argc != 2) {
        fprintf (stderr, "usage: packbox num, where num is 1, 2, or 3.\n");
        /* 这个在对 GTK 进行收尾处理后以退出状态为 1 退出。 */
        exit (1);
    }

    which = atoi (argv[1]);

    /* 创建窗口 */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    /* 你应该总是记住连接 delete_event 信号到主窗口。这对
     * 适当的直觉行为很重要 */
    g_signal_connect (G_OBJECT (window), "delete_event",
                      G_CALLBACK (delete_event), NULL);
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* 我们创建一个纵向盒（vbox）把横向盒组装进来。
     * 这使我们可以将填满按钮的横向盒一个个堆叠到
     * 这个纵向盒里。 */
    box1 = gtk_vbox_new (FALSE, 0);

    /* 显示哪个示例。这些对应于上面的图片。 */
    switch (which) {
    case 1:
        /* 创建一个新标签。 */
        label = gtk_label_new ("gtk_hbox_new (FALSE, 0);");

        /* 使标签靠左排列。我们将在构件属性部分讨
         * 论这个函数和其它的函数。 */
        gtk_misc_set_alignment (GTK_MISC (label), 0, 0);

        /* 将标签组装到纵向盒（vbox box1）里。记住加到纵向盒里的
         * 构件将依次一个放在另一个上面地组装。 */
        gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 0);

        /* 显示标签 */
        gtk_widget_show (label);

        /* 调用我们生成盒的函数 - homogeneous = FALSE, spacing = 0,
         * expand = FALSE, fill = FALSE, padding = 0 */
        box2 = make_box (FALSE, 0, FALSE, FALSE, 0);
        gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
        gtk_widget_show (box2);

        /* 调用我们生成盒的函数 - homogeneous = FALSE, spacing = 0,
         * expand = TRUE, fill = FALSE, padding = 0 */
        box2 = make_box (FALSE, 0, TRUE, FALSE, 0);
```

```
    gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
    gtk_widget_show (box2);

    /* 参数是： homogeneous, spacing, expand, fill, padding */
    box2 = make_box (FALSE, 0, TRUE, TRUE, 0);
    gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
    gtk_widget_show (box2);

    /* 创建一个分隔线，以后我们会更详细地学习这些，
     * 但它们确实很简单。 */
    separator = gtk_hseparator_new ();

    /* 组装分隔线到纵向盒。记住这些构件每个都被组装
    进了一个纵向盒，所以它们被垂直地堆叠。 */
    gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
    gtk_widget_show (separator);

    /* 创建另一个新标签，并显示它。 */
    label = gtk_label_new ("gtk_hbox_new (TRUE, 0);");
    gtk_misc_set_alignment (GTK_MISC (label), 0, 0);
    gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 0);
    gtk_widget_show (label);

    /* 参数是： homogeneous, spacing, expand, fill, padding */
    box2 = make_box (TRUE, 0, TRUE, FALSE, 0);
    gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
    gtk_widget_show (box2);

    /* 参数是： homogeneous, spacing, expand, fill, padding */
    box2 = make_box (TRUE, 0, TRUE, TRUE, 0);
    gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
    gtk_widget_show (box2);

    /* 另一个新分隔线。 */
    separator = gtk_hseparator_new ();
    /* gtk_box_pack_start 的最后三个参数是：
     * expand, fill, padding. */
    gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
    gtk_widget_show (separator);

    break;

case 2:

    /* 创建一个新标签，记住 box1 是一个纵向
     * 盒，它在 main() 前面部分创建 */
    label = gtk_label_new ("gtk_hbox_new (FALSE, 10);");
    gtk_misc_set_alignment (GTK_MISC (label), 0, 0);
    gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 0);
    gtk_widget_show (label);

    /* 参数是： homogeneous, spacing, expand, fill, padding */
    box2 = make_box (FALSE, 10, TRUE, FALSE, 0);
    gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
    gtk_widget_show (box2);

    /* 参数是： homogeneous, spacing, expand, fill, padding */
    box2 = make_box (FALSE, 10, TRUE, TRUE, 0);
    gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
    gtk_widget_show (box2);

    separator = gtk_hseparator_new ();
    /* gtk_box_pack_start 的最后三个参数是：
     * expand, fill, padding. */
    gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
```

```
      gtk_widget_show (separator);

      label = gtk_label_new ("gtk_hbox_new (FALSE, 0);");
      gtk_misc_set_alignment (GTK_MISC (label), 0, 0);
      gtk_box_pack_start (GTK_BOX (box1), label, FALSE, FALSE, 0);
      gtk_widget_show (label);

      /* 参数是： homogeneous, spacing, expand, fill, padding */
      box2 = make_box (FALSE, 0, TRUE, FALSE, 10);
      gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
      gtk_widget_show (box2);

      /* 参数是： homogeneous, spacing, expand, fill, padding */
      box2 = make_box (FALSE, 0, TRUE, TRUE, 10);
      gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
      gtk_widget_show (box2);

      separator = gtk_hseparator_new ();
      /* gtk_box_pack_start 的最后三个参数是： expand, fill, padding。 */
      gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
      gtk_widget_show (separator);
      break;

case 3:

      /* 这个示范了用 gtk_box_pack_end() 来右对齐构
       * 件的能力。首先，我们像前面一样创建一个新盒。 */
      box2 = make_box (FALSE, 0, FALSE, FALSE, 0);

      /* 创建将放在末端的标签。 */
      label = gtk_label_new ("end");
      /* 用 gtk_box_pack_end()组装它，这样它被放到
       * 在 make_box()调用里创建的横向盒的右端。 */
      gtk_box_pack_end (GTK_BOX (box2), label, FALSE, FALSE, 0);
      /* 显示标签。 */
      gtk_widget_show (label);

      /* 将 box2 组装进 box1 */
      gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, FALSE, 0);
      gtk_widget_show (box2);

      /* 放在底部的分隔线。 */
      separator = gtk_hseparator_new ();
      /* 这个明确地设置分隔线的宽度为 400 象素点和 5 象素点高。这样我们创建
       * 的横向盒也将为 400 象素点宽，并且"end"标签将和横向盒里其它的标签
       * 分开。否则，横向盒里的所有构件将尽量紧密地组装在一起。 */
      gtk_widget_set_size_request (separator, 400, 5);
      /* 将分隔线组装到在 main()前面部分创建的纵向盒（box1）里。 */
      gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 5);
      gtk_widget_show (separator);
}

/* 创建另一个新的横向盒.. 记住我们要用多少就能用多少！ */
quitbox = gtk_hbox_new (FALSE, 0);

/* 退出按钮。 */
button = gtk_button_new_with_label ("Quit");

/* 设置这个信号以在按钮被点击时终止程序 */
g_signal_connect_swapped (G_OBJECT (button), "clicked",
                          G_CALLBACK (gtk_main_quit),
                          window);
/* 将按钮组装进 quitbox。
 * gtk_box_pack_start 的最后三个参数是：
```

```
     * expand, fill, padding. */
    gtk_box_pack_start (GTK_BOX (quitbox), button, TRUE, FALSE, 0);
    /* pack the quitbox into the vbox (box1) */
    gtk_box_pack_start (GTK_BOX (box1), quitbox, FALSE, FALSE, 0);

    /* 将现在包含了我们所有构件的纵向盒（box1）组装进主窗口。 */
    gtk_container_add (GTK_CONTAINER (window), box1);

    /* 并显示剩下的所有东西 */
    gtk_widget_show (button);
    gtk_widget_show (quitbox);

    gtk_widget_show (box1);
    /* 最后显示窗口，这样所有东西一次性出现。 */
    gtk_widget_show (window);

    /* 当然，还有我们的主函数。 */
    gtk_main ();

    /* 当 gtk_main_quit() 被调用时控制权(Control)返回到
     * 这里，但当 exit()被使用时并不会。 */

    return 0;
}
```

用表组装

让我们看看另一种组装的方法 - 表(Tables)。 在某些情况下这是极其有用的。

使用表的时候，我们建立格来放入构件。构件可以占满我们所指定的所有空间.

第一个要看的，当然是 gtk_table_new()这个函数：

```
GtkWidget *gtk_table_new( guint    rows,
                          guint    columns,
                          gboolean homogeneous );
```

第一个参数是表中要安排的行的数量，而第二个，显然，就是列的数量。

homogeneous 参数跟表格框(table's boxes)的大小处理有关。如果 homogeneous 是 TRUE，所有表格框的大小都将调整为表中最大构件的大小。如果 homogeneous 为 FALSE，每个表格框将会按照同行中最高的构件，与同列中最宽的构件来决定自身的大小。

行与列为从 0 到 n 编号，而 n 是我们在调用 gtk_table_new 时所指定的值。所以，如果你指定 rows = 2 及 columns = 2, 布局图会看起来像这样：

```
 0        1        2
0+----------+----------+
 |        |        |
1+----------+----------+
 |        |        |
2+----------+----------+
```

注意坐标系统开始于左上角。要向框中放置一个构件，使用下面的函数：

```
void gtk_table_attach( GtkTable        *table,
                       GtkWidget       *child,
                       guint           left_attach,
                       guint           right_attach,
                       guint           top_attach,
                       guint           bottom_attach,
                       GtkAttachOptions xoptions,
                       GtkAttachOptions yoptions,
                       guint           xpadding,
```

```
                        guint           ypadding );
```

第一个参数（"table"）是你已经创建的表，第二个参数（"child"）是你想放进表里的构件。

left_attach 和 right_attach 参数指定构件放置的位置，并使用多少框来放。如果你想在 2x2 的表中的右下表项(table entry)处放入一个按钮，并且想让它*只*充满这个项，则 left_attach 应该为 = 1, right_attach = 2, top_attach = 1, bottom_attach = 2。

现在，如果你想让一个构件占据我们这个 2x2 表的整个顶行，你就用 left_attach = 0, right_attach = 2, top_attach = 0, bottom_attach = 1。

xoptions 及 yoptions 是用来指定组装时的选项，可以通过使用“位或”运算以允许多重选项。

这些选项是：

GTK_FILL

如果表框大于构件，同时 GTK_FILL 被指定，该构件会扩展开以使用所有可用的空间。

GTK_SHRINK

如果表构件分配到的空间比需求的小（通常是用户在改变窗口大小的时候），那么构件将会推到窗口的底部以外的区域，无法看见。如果 GTK_SHRINK 被指定了，构件将和表一起缩小。

GTK_EXPAND

这会导致表扩展以用完窗口中所有的保留空间。

Padding 和在盒(boxes)中的一样，在构件的周围产生一个指定象素的空白区域。

gtk_table_attach()有*很多*选项，所以，这里有一个简写：

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

X 及 Y 选项默认为 GTK_FILL | GTK_EXPAND，X 和 Y 的 padding 则设为 0。其余的参数与前面的函数一样。

我们还有 gtk_table_set_row_spacing() 和 gtk_table_set_col_spacing()。这些在指定的行或列之间插入空白。

```
void gtk_table_set_row_spacing( GtkTable *table,
                                guint     row,
                                guint     spacing );
```

和

```
void gtk_table_set_col_spacing ( GtkTable *table,
                                 guint     column,
                                 guint     spacing );
```

注意，对列来说，空白插到列的右边，对行来说，空白插入行的下边。

也可以为所有的行或/和列设置相同的间隔：

```
void gtk_table_set_row_spacings( GtkTable *table,
                                 guint    spacing );
```

和，

```
void gtk_table_set_col_spacings( GtkTable *table,
```

```
                                guint     spacing );
```

注意，用这些调用，最后一行和最后一列并不会有任何空白存在。

表组装示例

这里我们创建一个包含一个 2x2 表的窗口，表中放入三个按钮。前两个按钮将放在上面那行里。而第三个，quit 按钮，放在下面那行，并占据了两列。这就是说它看起来应该像这样：

这里是源代码：

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

/* 我们的回调。
 * 传到这个函数的数据被打印到标准输出 */
void callback( GtkWidget *widget,
               gpointer   data )
{
    g_print ("Hello again - %s was pressed\n", (char *) data);
}

/* 这个回调退出程序 */
gint delete_event( GtkWidget *widget,
                   GdkEvent  *event,
                   gpointer   data )
{
    gtk_main_quit ();
    return FALSE;
}

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

    gtk_init (&argc, &argv);

    /* 创建一个新窗口 */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    /* 设置窗口标题 */
    gtk_window_set_title (GTK_WINDOW (window), "Table");

    /* 为 delete_event 设置一个立即退出 GTK 的处理函数。 */
    g_signal_connect (G_OBJECT (window), "delete_event",
                      G_CALLBACK (delete_event), NULL);

    /* 设置窗口的边框宽度。 */
    gtk_container_set_border_width (GTK_CONTAINER (window), 20);

    /* 创建一个 2x2 的表 */
    table = gtk_table_new (2, 2, TRUE);

    /* 将表放进主窗口 */
```

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

    /* 创建第一个按钮 */
    button = gtk_button_new_with_label ("button 1");

    /* 当这个按钮被点击时，我们调用 "callback" 函数，并将一个
     * 指向"button 1"的指针作为它的参数 */
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (callback), (gpointer) "button 1");


    /* 将 button 1 插入表的左上象限(quadrant) */
    gtk_table_attach_defaults (GTK_TABLE (table), button, 0, 1, 0, 1);

    gtk_widget_show (button);

    /* 创建第二个按钮 */

    button = gtk_button_new_with_label ("button 2");

    /* 当这个按钮被点击时，我们调用 "callback" 函数，并将一个
     * 指向"button 2"的指针作为它的参数 */
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (callback), (gpointer) "button 2");
    /* 将 button 2 插入表的右上象限 */
    gtk_table_attach_defaults (GTK_TABLE (table), button, 1, 2, 0, 1);

    gtk_widget_show (button);

    /* 创建"Quit"按钮 */
    button = gtk_button_new_with_label ("Quit");

    /* 当这个按钮被点击时，我们调用 "delete_event" 函数接着
     * 程序就退出了 */
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (delete_event), NULL);

    /* 将退出按钮插入表的下面两个象限 */
    gtk_table_attach_defaults (GTK_TABLE (table), button, 0, 2, 1, 2);

    gtk_widget_show (button);

    gtk_widget_show (table);
    gtk_widget_show (window);

    gtk_main ();

    return 0;
}
```

构件概述

在 GTK 中创建一个构件的一般步骤是：

1. gtk_*_new() - 创建各种构件的函数。这些函数都将在本文档中作详细的介绍。
2. 把所有想连接的信号都连接到对应的信号处理函数。
3. 设定构件的属性。
4. 用 gtk_container_add() or gtk_box_pack_start()等适当的函数把构件放置到一个容器构件中。

5. gtk_widget_show() 显示构件。

gtk_widget_show() 让 GTK 知道我们已经完成设定构件属性的工作，并且能够让它显示出来了。你也可以用 gtk_widget_hide 使构件再次隐藏起来。各个构件显示的顺序并不重要，不过我建议在最后显示窗口，这样整个窗口就可以一次弹出来，而不是让用户看着窗口里的构件一个个生成并 显示出来。在窗口(也是一个构件)用 gtk_widget_show() 函数显示出来之前，它的子构件并不会被显示。

类型转换

再继续下去你会发现，GTK 使用了一套类型转换系统。它利用的是一套宏，这些宏在转换前还会测试能否进行转换。一些常见的宏是:

```
G_OBJECT (object)
GTK_WIDGET (widget)
GTK_OBJECT (object)
GTK_SIGNAL_FUNC (function)
GTK_CONTAINER (container)
GTK_WINDOW (window)
GTK_BOX (box)
```

这些都用来在函数中转换参数。你将在范例中看到它们的使用，只要看看函数的声明就可以知道什么时候需要使用它们。

在下节介绍的类的组织里，你将知道所有的 GtkWidget 都是从 GObject 这个基本类派生来的。也就是说，你能把 widget 传给任何需要以 object 类型作为参数的函数 - 只要用一下 G_OBJECT() 宏。

例如:

```
g_signal_connect( G_OBJECT (button), "clicked",
                  G_CALLBACK (callback_function), callback_data);
```

这样把 button 转换为一个 object，并提供一个函数指针作为回调函数。

很多构件也是容器，看一下节介绍的类的组织，你就会注意到很多构件是从容器类派生来的。这些构件都可以用 GTK_CONTAINER 宏转换后传递给需要以 container 作为参数的函数。

可惜的是，这些宏在本文档中并没有广泛地提及，不过我推荐去看一下 GTK 的头文件或者 GTK API 参考手册。这将很有用。实际上，通过看函数的声明来学习一个构件的使用也不难。

构件的组织

作为参考，这里是实现构件的类组织树。 (不推荐使用的构件和一些辅助的类被省略了。)

```
GObject
 |
 GtkObject
   +GtkWidget
   | +GtkMisc
   | | +GtkLabel
   | | | `GtkAccelLabel
   | | +GtkArrow
   | | `GtkImage
   | +GtkContainer
   | | +GtkBin
   | | | +GtkAlignment
   | | | +GtkFrame
   | | | | `GtkAspectFrame
   | | | +GtkButton
   | | | | +GtkToggleButton
   | | | | | `GtkCheckButton
   | | | | |   `GtkRadioButton
   | | | | `GtkOptionMenu
   | | | +GtkItem
   | | | | +GtkMenuItem
   | | | |   +GtkCheckMenuItem
   | | | |   | `GtkRadioMenuItem
   | | | |   +GtkImageMenuItem
```

```
| | | |   +GtkSeparatorMenuItem
| | | |   `GtkTearoffMenuItem
| | | +GtkWindow
| | | | +GtkDialog
| | | | | +GtkColorSelectionDialog
| | | | | +GtkFileSelection
| | | | | +GtkFontSelectionDialog
| | | | | +GtkInputDialog
| | | | | `GtkMessageDialog
| | | | `GtkPlug
| | | +GtkEventBox
| | | +GtkHandleBox
| | | +GtkScrolledWindow
| | | `GtkViewport
| | +GtkBox
| | | +GtkButtonBox
| | | | +GtkHButtonBox
| | | | `GtkVButtonBox
| | | +GtkVBox
| | | | +GtkColorSelection
| | | | +GtkFontSelection
| | | | `GtkGammaCurve
| | | `GtkHBox
| | |   +GtkCombo
| | |   `GtkStatusbar
| | +GtkFixed
| | +GtkPaned
| | | +GtkHPaned
| | | `GtkVPaned
| | +GtkLayout
| | +GtkMenuShell
| | | +GtkMenuBar
| | | `GtkMenu
| | +GtkNotebook
| | +GtkSocket
| | +GtkTable
| | +GtkTextView
| | +GtkToolbar
| | `GtkTreeView
| +GtkCalendar
| +GtkDrawingArea
| | `GtkCurve
| +GtkEditable
| | +GtkEntry
| |   `GtkSpinButton
| +GtkRuler
| | +GtkHRuler
| | `GtkVRuler
| +GtkRange
| | +GtkScale
| | | +GtkHScale
| | | `GtkVScale
| | `GtkScrollbar
| |   +GtkHScrollbar
| |   `GtkVScrollbar
| +GtkSeparator
| | +GtkHSeparator
| | `GtkVSeparator
| +GtkInvisible
| +GtkPreview
| `GtkProgressBar
+GtkAdjustment
+GtkCellRenderer
| +GtkCellRendererPixbuf
| +GtkCellRendererText
| +GtkCellRendererToggle
+GtkItemFactory
+GtkTooltips
```

`GtkTreeViewColumn

无窗口构件

下面列出的构件没有关联的窗口。如果你想截取它们的信号，你需要使用事件盒。请看事件盒构件的介绍。

GtkAlignment
GtkArrow
GtkBin
GtkBox
GtkButton
GtkCheckButton
GtkFixed
GtkImage
GtkLabel
GtkMenuItem
GtkNotebook
GtkPaned
GtkRadioButton
GtkRange
GtkScrolledWindow
GtkSeparator
GtkTable
GtkToolbar
GtkAspectFrame
GtkFrame
GtkVBox
GtkHBox
GtkVSeparator
GtkHSeparator

接着我们将挨个介绍每个构件，编写一些简单的函数来显示它们。另一个好的源代码是 GTK 附带的 testgtk 程序，你可以在 tests/testgtk.c 里面看到。

按钮构件

一般按钮 Normal Buttons

在 前面我们见到得最多的就是按钮构件了。它们十分简单。创建按钮有好几种方法。你可以用 gtk_button_new_with_label() 或 gtk_button_new_with_mnemonic() 来创建一个带标签的按钮，用 gtk_button_new_from_stock() 来从一个原料(stock)项创建一个包含图像和文本的按钮，或者用 gtk_button_new() 创建一个空白按钮。接着你可以决定把一个标签或位图(pixmap)组装到这个新创建的按钮里。要这样做，创建一个新的盒，然后用常见的 gtk_box_pack_start() 把你的对象组装到盒里，再用 gtk_container_add() 把盒组装到按钮里。

这里是一个用函数 gtk_button_new() 去创建一个带图像和标签的按钮的示例。我把创建盒的代码分离了出来，这样你可以在你自己的程序中使用它。后面还有更多使用图像的示例。


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

/* 创建一个新的横向盒，它包含一个图像和一个标签，并返回这个盒。*/

GtkWidget *xpm_label_box( gchar     *xpm_filename,
                          gchar     *label_text )
{
    GtkWidget *box;
```

```
    GtkWidget *label;
    GtkWidget *image;

    /* 为图像和标签创建盒 */
    box = gtk_hbox_new (FALSE, 0);
    gtk_container_set_border_width (GTK_CONTAINER (box), 2);

    /* 创建一个图像 */
    image = gtk_image_new_from_file (xpm_filename);

    /* 为按钮创建一个标签 */
    label = gtk_label_new (label_text);

    /* 把图像和标签组装到盒子里 */
    gtk_box_pack_start (GTK_BOX (box), image, FALSE, FALSE, 3);
    gtk_box_pack_start (GTK_BOX (box), label, FALSE, FALSE, 3);

    gtk_widget_show (image);
    gtk_widget_show (label);

    return box;
}

/* 我们常见的回调函数 */
void callback( GtkWidget *widget,
               gpointer   data )
{
    g_print ("Hello again - %s was pressed\n", (char *) data);
}

int main( int   argc,
          char *argv[] )
{
    /* GtkWidget 是构件的存储类型 */
    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *box;

    gtk_init (&argc, &argv);

    /* 创建一个新窗口 */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    gtk_window_set_title (GTK_WINDOW (window), "Pixmap'd Buttons!");

    /* 对所有的窗口做这一步是一个好主意。*/
    g_signal_connect (G_OBJECT (window), "destroy",
                      G_CALLBACK (gtk_main_quit), NULL);

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

    /* 设置窗口边框的宽度。*/
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* 创建一个新的按钮 */
    button = gtk_button_new ();

    /* 连接按钮的 "clicked" 信号到我们的回调 */
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (callback), (gpointer) "cool button");

    /* 调用我们的创建盒的函数 */
    box = xpm_label_box ("info.xpm", "cool button");

    /* 组装和显示所有的构件 */
```

```
    gtk_widget_show (box);

    gtk_container_add (GTK_CONTAINER (button), box);

    gtk_widget_show (button);

    gtk_container_add (GTK_CONTAINER (window), button);

    gtk_widget_show (window);

    /* 停在这里，等待事件发生。 */
    gtk_main ();

    return 0;
}
```

函数 xpm_label_box() 可用于组装图像和标签到任何可以作为容器的构件里。

按钮构件有如下信号：

- pressed --- 当鼠标键在按钮构件里按下时发出
- released --- 当鼠标键在按钮构件里释放时发出
- clicked --- 当鼠标键在按钮构件里按下并接着在按钮构件里释放时发出
- enter --- 当鼠标光标进入按钮构件时发出
- leave --- 当鼠标光标离开按钮构件时发出

开关按钮 Toggle Buttons

开关按钮由一般按钮派生而来，并且非常相似，只是开关按钮有两个状态，通过点击可以切换。它们可以是被按下的 (depressed)，当你再点击一下，他们会弹起来。再点击一下，它们又会再弹下去。

开关按钮是复选按钮和单选按钮的基础，所以单选按钮和复选按钮继承了许多开关按钮的函数调用。我会在讲到它们时指出这些来。

创建一个新的开关按钮：

```
GtkWidget *gtk_toggle_button_new( void );

GtkWidget *gtk_toggle_button_new_with_label( const gchar *label );

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

你可以猜到，创建开关按钮应该和一般按钮构件相同。第一个函数是创建一个空白的开关按钮，后面两个函数创建带标签的开关按钮。其中 _mnemonic() 函数处理标签中的以 '_' 为前缀的助记语法符。

我们是通过读取开关构件 (包括单选和复选按钮) 结构的 active 域，来检测开关按钮的状态。之前要用 GTK_TOGGLE_BUTTON 宏把构件指针转换为开关构件指针。我们关心的各种开关按钮 (开关按钮，复选按钮，和单选按钮构件) 的信号是 "toggled" 信号。为了检测这些按钮的状态，设置一个处理函数以捕获 "toggled" 信号，并且通过读取结构测定它的状态。该回调函数如下：

```
void toggle_button_callback (GtkWidget *widget, gpointer data)
{
    if (gtk_toggle_button_get_active (GTK_TOGGLE_BUTTON (widget)))
    {
        /* 如果运行到这里，开关按钮是按下的 */

    } else {

        /* 如果运行到这里，开关按钮是弹起的 */
    }
}
```

设置开关按钮和它的孩子 (即单选和复选按钮) 的状态，用如下函数：

```
void gtk_toggle_button_set_active( GtkToggleButton *toggle_button,
                                   gboolean         is_active );
```

上面的调用可以用来设置开关按钮，以及它的孩子单选和复选按钮的状态。将你所创建的按钮作为第一个参数传入，以及一个 TRUE 或 FALSE 值作为第二个状态参数来指定它应该是下 (按下) 还是上 (弹起)。默认是上，即 FALSE。

注意，当你使用 gtk_toggle_button_set_active() 函数，并且状态也实际改变了时，它会导致按钮发出 "clicked" 和 "toggled" 信号。

```
gboolean gtk_toggle_button_get_active   (GtkToggleButton *toggle_button);
```

返回值是开关按钮的当前状态。

复选按钮 Check Buttons

复选按钮继承了开关按钮的许多属性和功能，但看起来有一点点不同。不像开关按钮那样文字在按钮内部,复合按钮左边是一个小的方框,而文字在其右边。这些常用在应用程序中以切换各选项的开和关。

创建函数和普通按钮的类似。

```
GtkWidget *gtk_check_button_new( void );

GtkWidget *gtk_check_button_new_with_label ( const gchar *label );

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

函数 gtk_check_button_new_with_label() 创建一个带标签的复选按钮。

检测复选按钮的状态的方法和开关按钮是完全相同的。

单选按钮 Radio Buttons

单选按钮与复选按钮相似，只是单选按钮是分组的，在一组中只有一个处于选中/按下状态。这在你的应用程序中要从几个选项中选一个的地方可以用到。

用这些调用之一来创建一个新的单选按钮:

```
GtkWidget *gtk_radio_button_new( GSList *group );

GtkWidget *gtk_radio_button_new_from_widget( GtkRadioButton *group );

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

GtkWidget* gtk_radio_button_new_with_label_from_widget( GtkRadioButton *group,
                                                        const gchar    *label );

GtkWidget *gtk_radio_button_new_with_mnemonic( GSList *group,
                                               const gchar  *label );

GtkWidget *gtk_radio_button_new_with_mnemonic_from_widget( GtkRadioButton *group,
                                                           const gchar  *label );
```

你可能注意到了，这些 调用有个额外的参数。它们需要一个组以正常运作。第一次调用 gtk_radio_button_new() 或 gtk_radio_button_new_with_label() 应该传递 NULL 值作为第一个参数。接着用如下函数创建一个组:

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

有 一点很重要，必须为每个添加到组的新按钮调用 gtk_radio_button_get_group()，并把前一个按钮作为参数。返回的结果再传给下一个调用 gtk_radio_button_new() 或 gtk_radio_button_new_with_label()。这样才能建立连锁的按钮。看一下下面的示例会更清楚一些。

你可以使用下面的语法来稍微缩短上面的步骤，它不需要一个变量来存储按钮列表:

```
    button2 = gtk_radio_button_new_with_label(
```

```
                gtk_radio_button_get_group (GTK_RADIO_BUTTON (button1)),
                "button2");
```

而 _from_widget() 创建函数可以让你做得更简洁些，它完全省略了 gtk_radio_button_get_group() 调用。在下面示例的第三个按钮就是用这种方法创建的。

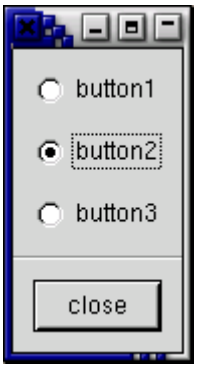
```
    button2 = gtk_radio_button_new_with_label_from_widget(
                GTK_RADIO_BUTTON (button1),
                "button2");
```

明确地指定哪个按钮应该被默认按下也是个好主意，用：

```
void gtk_toggle_button_set_active( GtkToggleButton *toggle_button,
                                   gboolean        state );
```

这 在开关按钮部分描述过，在这里它也确切地以同样的方式工作。多个单选按钮组合到一起后，组中一次只能有一个被激活。如果用户点击一个单选按钮，接着点另一 个，第一个单选按钮会首先发出 "toggled" 信号 (以报告变得不激活了)，然后第二个也会发出 "toggled" 信号 (以报告变得激活了)。

下面的示例创建一个含三个按钮的单选按钮组。




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

gint close_application( GtkWidget *widget,
                        GdkEvent  *event,
                        gpointer   data )
{
  gtk_main_quit ();
  return FALSE;
}

int main( int   argc,
          char *argv[] )
{
    GtkWidget *window = NULL;
    GtkWidget *box1;
    GtkWidget *box2;
    GtkWidget *button;
    GtkWidget *separator;
    GSList *group;

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    g_signal_connect (G_OBJECT (window), "delete_event",
                      G_CALLBACK (close_application),
                      NULL);

    gtk_window_set_title (GTK_WINDOW (window), "radio buttons");
    gtk_container_set_border_width (GTK_CONTAINER (window), 0);
```

```
    box1 = gtk_vbox_new (FALSE, 0);
    gtk_container_add (GTK_CONTAINER (window), box1);
    gtk_widget_show (box1);

    box2 = gtk_vbox_new (FALSE, 10);
    gtk_container_set_border_width (GTK_CONTAINER (box2), 10);
    gtk_box_pack_start (GTK_BOX (box1), box2, TRUE, TRUE, 0);
    gtk_widget_show (box2);

    button = gtk_radio_button_new_with_label (NULL, "button1");
    gtk_box_pack_start (GTK_BOX (box2), button, TRUE, TRUE, 0);
    gtk_widget_show (button);

    group = gtk_radio_button_get_group (GTK_RADIO_BUTTON (button));
    button = gtk_radio_button_new_with_label (group, "button2");
    gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (button), TRUE);
    gtk_box_pack_start (GTK_BOX (box2), button, TRUE, TRUE, 0);
    gtk_widget_show (button);

    button = gtk_radio_button_new_with_label_from_widget (GTK_RADIO_BUTTON (button),
                                                          "button3");
    gtk_box_pack_start (GTK_BOX (box2), button, TRUE, TRUE, 0);
    gtk_widget_show (button);

    separator = gtk_hseparator_new ();
    gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 0);
    gtk_widget_show (separator);

    box2 = gtk_vbox_new (FALSE, 10);
    gtk_container_set_border_width (GTK_CONTAINER (box2), 10);
    gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, TRUE, 0);
    gtk_widget_show (box2);

    button = gtk_button_new_with_label ("close");
    g_signal_connect_swapped (G_OBJECT (button), "clicked",
                              G_CALLBACK (close_application),
                              window);
    gtk_box_pack_start (GTK_BOX (box2), button, TRUE, TRUE, 0);
    GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);
    gtk_widget_grab_default (button);
    gtk_widget_show (button);
    gtk_widget_show (window);

    gtk_main ();

    return 0;
}
```

调整对象 Adjustments

GTK 有多种构件能够由用户通过鼠标或键盘进行调整，比如范围构件。还有一些构件，比如说 GtkText 和 GtkViewport，内部都有一些可调整的属性。

很 明显，当用户调整范围构件的值时，应用程序需要对值的变化进行响应。一种办法就是当构件的调整值发生变化时，让每个构件引发自己的信号,将新值传递到信号 处理函数中，或者让它在构件的内部数据结构中查找构件的值。但是，也许需要将这个调整值同时连接到几个构件上，使得调整一个值时，其它的构件都随之响应。 最明显的示例就是将一个滚动条连接到一个视角构件(viewport)或者滚动的文本区(text area)上。如果每个构件都要有自己的设置或获取调整值的方法，程序员或许需要自己编写很复杂的信号处理函数，以便将这些不同构件之间的变化同步或相关 联。

GTK 用一个调整对象(Adjustment object)解决了这个问题。调整对象不是构件，但是为构件提供了一种以抽象、灵活的方法来传递调整值信息。调整对象最明显的用处就是为范围构件(比如 滚动条和比例构件)储存配置参数和值。然而，因为调整对象是从 Object 派生的，在其正常的数据结构之外，它还具有一些特殊的功能。最重要的是，它们能 够引发信号，就像构件一样，这些信号不仅能够让程序对用户在可调整构件上的输入进行响应，还能在可调整构件之间透明地传播调整值。

在许多其它的构件中都能够看到调整对象的用处。比如进度条、视角、滚动窗口等。

创建一个调整对象

许多使用调整对象的构件都能够自动创建它，但是有些情况下，必须自己手工创建。用下面的函数创建调整对象：

```
GtkObject *gtk_adjustment_new( gdouble value,
                               gdouble lower,
                               gdouble upper,
                               gdouble step_increment,
                               gdouble page_increment,
                               gdouble page_size );
```

其中的 value 参数是要赋给调整对象的初始值，通常对应于一个可调整构件的最高或最低位置。lower 参数指定调整对象能取的最低值，step_increment 参数指定用户能小步增加的值，page_increment 是用户能大步调整的值。page_size 参数通常用于设置分栏构件(panning widget)的可视区域。upper 参数用于表示分栏构件的子构件的最底部或最右边的坐标。因而，它*不*一定总是 value 能取的最大值，因为这些构件的 page_size 通常是非零值(value 能取的最大值一般是 upper-page_size)。

轻松使用调整对象

可调整构件大致可以分为两组：一组对这些值使用特定的单位，另一组将这些值当作任意数值。后一组包括范围构件：滚动条、比例构件(scales)、进度条以及微调按钮(spin button)。这些构件的值都可以使用鼠标和键盘直接进行调整。它们将调整对象的 lower 和 upper 值当作用户能够操纵的调整值的范围。缺省时，它们只会修改调整对象的 value 参数,也就是说，它们的范围一般都是不变的。

另 一组包含文本构件、视角构件、复合列表框(compound list)以及滚动窗口构件。所有这些构件都是间接通过滚动条进行调整的。所有使用调整对象的构件都可以使用自己的调整对象，或者使用你创建的调整对象， 但是最好让这一类构件都使用它们自己的调整对象。一般它们都对 value 以外的参数作了新的解释,对这些值的解释各个构件都有所不同,你需要阅读它们的源代码。

现在，你也许在想，文本构件和视角构件里的调整对象除了 value 参数以外，其他的参数都是由它们自己控制的，而滚动条就*只*修改调整对象的 value 参数，如果在滚动条和文本构件之间*共享*调整对象，操纵滚动条会自动调整文本构件吗？当然会，就像下面的代码所做的：

```
/* 视角构件会自动为自己创建一个调整对象 */
   viewport = gtk_viewport_new (NULL, NULL);
   /* 让垂直滚动条使用视角构件已经创建的调整对象 */
   vscrollbar = gtk_vscrollbar_new (gtk_viewport_get_vadjustment (viewport));
```

“调整对象”的内部机制

你可能要问，如果我想创建一个信号处理函数，当用户调整范围构件或微调按钮时让这个处理函数进行响应，应该从调整对象中取什么值，怎样从中取值呢？要解决这个问题，先看一下 _GtkAdjustment 结构的定义：

```
struct _GtkAdjustment
{
   GtkObject parent_instance;

   gdouble lower;
   gdouble upper;
   gdouble value;
   gdouble step_increment;
   gdouble page_increment;
   gdouble page_size;
};
```

如果你不喜欢像一个 *real* C 程序员那样直接从结构中取值,你可以使用下面的函数来获取调整对象的 value 参数值：

```
gdouble gtk_adjustment_get_value( GtkAdjustment *adjustment);
```

因为设置调整对象的值时，通常想让每个使用这个调整对象的构件对值的改变做出响应，GTK 提供了下面的函数：

```
void gtk_adjustment_set_value( GtkAdjustment *adjustment,
```

```
                                gdouble       value );
```

前面说过，和其它构件一样，调整对象是 Object 的子类，因而，它也能够引发信号。这也是为什么当滚动条和其它可调整构件共享调整对象时它们能够自动更新的原因。所有的可调整构件都为它们的调整对象的 value_changed 信号设置了一个信号处理函数。下面是这个信号在_GtkAdjustmentClass 结构中的定义：

```
void (* value_changed) (GtkAdjustment *adjustment);
```

各 种使用调整对象的构件都会在它们的值发生变化时引发它们的调整对象的信号。这种情况发生在当用户用鼠标使范围构件的滑块移动和当程序使用 gtk_adjustment_set_value() 函数显式地改变调整对象的值时。所以，如果有一个比例构件，想在它的值改变时改变一幅画的旋转角度，应该创建像下面这样的回调函数：

```
void cb_rotate_picture (GtkAdjustment *adj, GtkWidget *picture)
{
  set_picture_rotation (picture, gtk_adjustment_get_value (adj));
...
```

再将这个回调函数连接到构件的调整对象上：

```
g_signal_connect (G_OBJECT (adj), "value_changed",
                  G_CALLBACK (cb_rotate_picture), picture);
```

当构件重新配置了它的调整对象的 upper 或 lower 参数时(比如，用户向文本构件添加了更多的文本时)，发生了什么？在这种情况下，它会引发一个 changed 信号：

```
void (* changed) (GtkAdjustment *adjustment);
```

范围构件一般为这个信号设置回调函数，构件会改变它们的外观以反映变化。例如，滚动条上的滑块会根据它的调整对象的 lower 和 upper 参数之间的差值的变化而伸长或缩短。

一般不需要处理这个信号，除非你想要写一个新的范围构件。不过，如果直接改变了调整对象的任何参数，应该引发这个信号，以便相关构件重新配置自己。用下面的函数引发这个信号：

```
g_signal_emit_by_name (G_OBJECT (adjustment), "changed");
```

现在尽情使用调整对象吧！

范围构件 Range Widgets

范 围构件(Range Widgets)是一大类构件，包含常见的滚动条构件(Scrollbar Widgets)和较少见的“比例”构件(Scale Widgets)。尽管这两种构件是用于不同的目的，它们在功能和实现上都是非常相似的。所有范围构件共用一套公用的图形元素，每一个都有自己的 X 窗口，并能接收事件。它们都包含一个“滑槽(trough)”和一个“滑块(slider)”(在一些其它 GUI 环境下又称"thumbwheel")。 用鼠标指针拖动滑块可以在滑槽中前后移动，在滑块前后的滑槽中点击，根据不同的鼠标按键，滑块就会向接近点击处的方向移动一点，或完全到位，或移动特定的 距离。

在前面的调整对象里提到过，所有范围构件都是与一个调整对象相关联的。该对象会计算滑块的长度和在滑槽中的位置。当用户操纵滑块时，范围构件会改变调整对象的值。

滚动条构件 Scrollbar Widgets

这些都是标准的，到处被使用的滚动条(Scrollbar)。一般只用于滚动其它的构件，比如列表、文本构件，或视角构件(在很多情况下使用滚动窗口构件更方便)。对其它目的,应该使用比例构件，因为它更友好，而且有更多的特性。

分别有水平和垂直滚动条两种类型。实在不必对它们作说明。你用下面的函数创建滚动条：

```
GtkWidget *gtk_hscrollbar_new( GtkAdjustment *adjustment );

GtkWidget *gtk_vscrollbar_new( GtkAdjustment *adjustment );
```

这就是它们所有的相关函数(如果你不相信，去看一下它的头文件！)。adjustment 参数可以是一个指向已有调整对象的指针或 NULL，当为 NULL 时会自动创建一个。如果希望将新创建的调整对象传递给其它构件的构建函数，例如文本构件的构建函数，在这种情况下指定 NULL 是很有用的。

比例构件 Scale Widgets

比例构件(Scale widgets)一般用于允许用户在一个指定的取值范围你可视地选择和操纵一个值。例如，在图片的缩放预览中调整放大倍数，或控制一种颜色的亮度，或在指定屏幕保护启动之前不活动的时间间隔时，可能需要用到比例构件。

创建一个比例构件

像滚动条一样，有水平和垂直两种不同类型的比例构件。(大多数程序员似乎喜欢水平的比例构件。)既然在本质上它们的工作方式是相同的，那么不需要对它们分别对待。下面的函数分别创建垂直和水平的比例构件：

```
GtkWidget *gtk_vscale_new( GtkAdjustment *adjustment );

GtkWidget *gtk_vscale_new_with_range( gdouble min,
                                      gdouble max,
                                      gdouble step );

GtkWidget *gtk_hscale_new( GtkAdjustment *adjustment );

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

adjustment 参数可以是一个已经用 gtk_adjustment_new() 创建了的调整对象，或 NULL，此时,会创建一个匿名的调整对象，所有的值都设为 0.0(在此处用处不大)。为了避免把自己搞糊涂，你可能想要创建一个 page_size 值设为 0.0 的调整对象，让它的 upper 值与用户能选择的最高值相对应。而 _new_with_range() 函数会照顾到创建一个适当的调整对象。(如果你*已经*彻底困惑了，看看调整对象这一章，它解释了调整对象的作用和怎样创建和操作它们。)

函数和信号(至少讲了函数)

比例构件可以在滑槽的旁边以数字形式显示其当前值。默认行为是显示值,但是可以用下面这个函数改变其行为：

```
void gtk_scale_set_draw_value( GtkScale *scale,
                               gboolean draw_value );
```

可以猜到，draw_value 取值为 TRUE 或 FALSE，结果是显示或不显示。

缺省情况下，比例构件显示的值，也就是在它的调整对象定义中的 value 域，圆整到一位小数。可以用以下函数改变显示的小数位：

```
void gtk_scale_set_digits( GtkScale *scale,
                           gint     digits );
```

digits 是要显示的小数位数。可以将 digits 设置为任意位数，但是实际上屏幕上最多只能显示 13 位小数。

最后，显示的值可以放在滑槽附近的不同位置：

```
void gtk_scale_set_value_pos( GtkScale        *scale,
                              GtkPositionType  pos );
```

参数 pos 是 GtkPositionType 类型，可以取以下值之一：

```
GTK_POS_LEFT
GTK_POS_RIGHT
GTK_POS_TOP
GTK_POS_BOTTOM
```

如果将值显示在滑槽的“侧面”(例如，在水平比例构件的滑槽的顶部和底部)，显示的值将跟随滑块上下移动。

所有前面讲的函数都在<gtk/gtkscale.h>里定义。当你包含了<gtk/gtk.h>文件时，所有 GTK 构件的头文件都自动包含了。但你应该去察看一下所有你感兴趣的构件的头文件，这样才能学到它们的更多的功能和特性。

常用的范围函数

范围构件本质上来说都是相当复杂的，不过，像所有“基本类”构件一样，绝大部分复杂性只有当你想彻底了解它时才吸引人。同样,几乎所有它定义的函数和信号都只在用它们写派生构件时才真正用到。然而，在<gtk/gtkrange.h>中还是有一些很有用的函数，它们对所有范围构件都起作用。

设置更新方式

范围构件的“更新方式”定义了用户与构件交互时的它的调整对象的 value 值如何变化，以及如何引发 "value_changed" 信号给调整对象。更新方式在<gtk/gtkenums.h>中定义为 enum GtkUpdateType 类型，有以下取值：

GTK_UPDATE_CONTINUOUS

这是默认值。"value_changed" 信号是连续引发，例如，每当滑块移动，甚至移动最小数量时都会引发。

GTK_UPDATE_DISCONTINUOUS

只有滑块停止移动，用户释放鼠标键时才引发 "value_changed" 信号。

GTK_UPDATE_DELAYED

当用户释放鼠标键，或者滑块短期停止移动时才引发 "value_changed" 信号。

范围构件的更新方式可以用以下方法设置：用 GTK_RANGE(widget)宏将构件转换，并将它传递给这个函数：

```
void gtk_range_set_update_policy( GtkRange      *range,
                                  GtkUpdateType  policy);
```

获得和设置调整对象

猜得出，用以下函数"快速"取得和设置调整对象：

```
GtkAdjustment* gtk_range_get_adjustment( GtkRange *range );

void gtk_range_set_adjustment( GtkRange      *range,
                               GtkAdjustment *adjustment );
```

gtk_range_get_adjustment()返回一个指向 range 所连接的调整对象的指针。

如果将 range 正在使用的调整对象传递给 gtk_range_set_adjustment()函数，什么也不会发生，不管是否改变了其内部的值。如果是将一个新的调整对象传递给它，它会将旧的调整对象(如果存在)解除引用(unreference)(可能会销毁它)，将适当的信号连接到新的调整对象，并且调用私有函数 gtk_range_adjustment_changed()，该函数将(或至少假装会...)重新计算滑块的尺寸和/或位置，并在需要时重新绘出该构件。正如在调整对象部分所提到的，如果想重新使用同一个调整对象，当直接修改它的值时，应该引发一个 "changed" 信号给它，像这样：

```
g_signal_emit_by_name (G_OBJECT (adjustment), "changed");
```

键盘和鼠标绑定

所有的 GTK 范围构件在鼠标点击交互时的方式差不多是相同的。在滑槽上点击鼠标左键(button-1)使调整对象的 value 值加上或减去一个 page_increment，滑块也移动相应的距离。在滑槽上点击鼠标中键(button-2)将使滑块跳到鼠标点击处。在滑槽上点击鼠标右键(button-3)或在滚动条的箭头上点任意鼠标键会使它的调整对象的 value 值一次改变一个 step_increment 值。

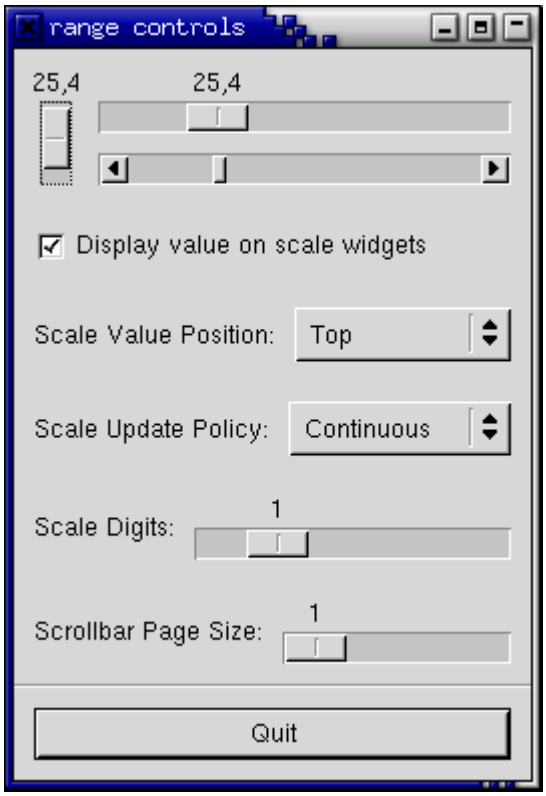
滚动条是不能获得焦点的，因此没有按键绑定。对其它的范围构件(当然，只在该构件获得焦点时有效)来说，水平和垂直范围构件两者的按键绑定*没有*一点区别。

所有范围构件都可以用左、右、上和下方向键操作，Page Up 和 Page Down 键也一样。方向键以 step_increment 为单位向上或向下移动滑块，而 Page Up 和 Page Down 以 page_increment 为单位移动它。

用户可以使用键盘让滑块在滑槽的两端之间自由移动。用 Home 和 End 键就行了。

示例

这个示例可以说是 testgtk.c 里的"范围控制"测试部分的修改版。它主要是在一个窗口上放置了三个范围构件，都连接到同一个调整对象，并使用上面以及调整对象部分提到的一些调整参数的控制方法，这样你可以看到它们怎样影响这些构件的使用效果。




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

GtkWidget *hscale, *vscale;

void cb_pos_menu_select( GtkWidget       *item,
                         GtkPositionType  pos )
{
    /* 设置两个比例构件的比例值的显示位置 */
    gtk_scale_set_value_pos (GTK_SCALE (hscale), pos);
    gtk_scale_set_value_pos (GTK_SCALE (vscale), pos);
}

void cb_update_menu_select( GtkWidget     *item,
                            GtkUpdateType  policy )
{
    /* 设置两个比例构件的更新方式 */
    gtk_range_set_update_policy (GTK_RANGE (hscale), policy);
    gtk_range_set_update_policy (GTK_RANGE (vscale), policy);
}

void cb_digits_scale( GtkAdjustment *adj )
{
    /* 设置 adj->value 圆整的小数位数 */
    gtk_scale_set_digits (GTK_SCALE (hscale), (gint) adj->value);
    gtk_scale_set_digits (GTK_SCALE (vscale), (gint) adj->value);
}
```

```
void cb_page_size( GtkAdjustment *get,
                   GtkAdjustment *set )
{
    /* 将示例调整对象的 page size 和 page increment size 设置
     * 为"Page Size"比例构件指定的值 */
    set->page_size = get->value;
    set->page_increment = get->value;

    /* 设置调整对象的值并使它引发一个 "changed" 信号，以重新配置所有
     * 已经连接到这个调整对象的构件。 */
    gtk_adjustment_set_value (set, CLAMP (set->value,
                                          set->lower,
                                          (set->upper - set->page_size)));
}

void cb_draw_value( GtkToggleButton *button )
{
    /* 根据复选按钮的状态设置在比例构件上是否显示比例值 */
    gtk_scale_set_draw_value (GTK_SCALE (hscale), button->active);
    gtk_scale_set_draw_value (GTK_SCALE (vscale), button->active);
}

/* 方便的函数 */

GtkWidget *make_menu_item (gchar     *name,
                           GCallback  callback,
                           gpointer   data)
{
    GtkWidget *item;

    item = gtk_menu_item_new_with_label (name);
    g_signal_connect (G_OBJECT (item), "activate",
                      callback, data);
    gtk_widget_show (item);

    return item;
}

void scale_set_default_values( GtkScale *scale )
{
    gtk_range_set_update_policy (GTK_RANGE (scale),
                                 GTK_UPDATE_CONTINUOUS);
    gtk_scale_set_digits (scale, 1);
    gtk_scale_set_value_pos (scale, GTK_POS_TOP);
    gtk_scale_set_draw_value (scale, TRUE);
}

/* 创建示例窗口 */

void create_range_controls( void )
{
    GtkWidget *window;
    GtkWidget *box1, *box2, *box3;
    GtkWidget *button;
    GtkWidget *scrollbar;
    GtkWidget *separator;
    GtkWidget *opt, *menu, *item;
    GtkWidget *label;
    GtkWidget *scale;
    GtkObject *adj1, *adj2;

    /* 标准的创建窗口代码 */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    g_signal_connect (G_OBJECT (window), "destroy",
                      G_CALLBACK (gtk_main_quit),
                      NULL);
```

```
gtk_window_set_title (GTK_WINDOW (window), "range controls");

box1 = gtk_vbox_new (FALSE, 0);
gtk_container_add (GTK_CONTAINER (window), box1);
gtk_widget_show (box1);

box2 = gtk_hbox_new (FALSE, 10);
gtk_container_set_border_width (GTK_CONTAINER (box2), 10);
gtk_box_pack_start (GTK_BOX (box1), box2, TRUE, TRUE, 0);
gtk_widget_show (box2);

/* value, lower, upper, step_increment, page_increment, page_size */
/* 注意，page_size 值只对滚动条构件有区别，并且，你实际上能取得的最高值
 * 是(upper - page_size)。 */
adj1 = gtk_adjustment_new (0.0, 0.0, 101.0, 0.1, 1.0, 1.0);

vscale = gtk_vscale_new (GTK_ADJUSTMENT (adj1));
scale_set_default_values (GTK_SCALE (vscale));
gtk_box_pack_start (GTK_BOX (box2), vscale, TRUE, TRUE, 0);
gtk_widget_show (vscale);

box3 = gtk_vbox_new (FALSE, 10);
gtk_box_pack_start (GTK_BOX (box2), box3, TRUE, TRUE, 0);
gtk_widget_show (box3);

/* 重新使用同一个调整对象 */
hscale = gtk_hscale_new (GTK_ADJUSTMENT (adj1));
gtk_widget_set_size_request (GTK_WIDGET (hscale), 200, -1);
scale_set_default_values (GTK_SCALE (hscale));
gtk_box_pack_start (GTK_BOX (box3), hscale, TRUE, TRUE, 0);
gtk_widget_show (hscale);

/* 再次重用同一个调整对象 */
scrollbar = gtk_hscrollbar_new (GTK_ADJUSTMENT (adj1));
/* 注意，这导致当滚动条移动时，比例构件总是连续更新 */
gtk_range_set_update_policy (GTK_RANGE (scrollbar),
                             GTK_UPDATE_CONTINUOUS);
gtk_box_pack_start (GTK_BOX (box3), scrollbar, TRUE, TRUE, 0);
gtk_widget_show (scrollbar);

box2 = gtk_hbox_new (FALSE, 10);
gtk_container_set_border_width (GTK_CONTAINER (box2), 10);
gtk_box_pack_start (GTK_BOX (box1), box2, TRUE, TRUE, 0);
gtk_widget_show (box2);

/* 用一个复选按钮控制是否显示比例构件的值 */
button = gtk_check_button_new_with_label("Display value on scale widgets");
gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (button), TRUE);
g_signal_connect (G_OBJECT (button), "toggled",
                  G_CALLBACK (cb_draw_value), NULL);
gtk_box_pack_start (GTK_BOX (box2), button, TRUE, TRUE, 0);
gtk_widget_show (button);

box2 = gtk_hbox_new (FALSE, 10);
gtk_container_set_border_width (GTK_CONTAINER (box2), 10);

/* 用一个选项菜单以改变显示值的位置 */
label = gtk_label_new ("Scale Value Position:");
gtk_box_pack_start (GTK_BOX (box2), label, FALSE, FALSE, 0);
gtk_widget_show (label);

opt = gtk_option_menu_new ();
menu = gtk_menu_new ();

item = make_menu_item ("Top",
                       G_CALLBACK (cb_pos_menu_select),
                       GINT_TO_POINTER (GTK_POS_TOP));
```

```
gtk_menu_shell_append (GTK_MENU_SHELL (menu), item);

item = make_menu_item ("Bottom", G_CALLBACK (cb_pos_menu_select),
                       GINT_TO_POINTER (GTK_POS_BOTTOM));
gtk_menu_shell_append (GTK_MENU_SHELL (menu), item);

item = make_menu_item ("Left", G_CALLBACK (cb_pos_menu_select),
                       GINT_TO_POINTER (GTK_POS_LEFT));
gtk_menu_shell_append (GTK_MENU_SHELL (menu), item);

item = make_menu_item ("Right", G_CALLBACK (cb_pos_menu_select),
                       GINT_TO_POINTER (GTK_POS_RIGHT));
gtk_menu_shell_append (GTK_MENU_SHELL (menu), item);

gtk_option_menu_set_menu (GTK_OPTION_MENU (opt), menu);
gtk_box_pack_start (GTK_BOX (box2), opt, TRUE, TRUE, 0);
gtk_widget_show (opt);

gtk_box_pack_start (GTK_BOX (box1), box2, TRUE, TRUE, 0);
gtk_widget_show (box2);

box2 = gtk_hbox_new (FALSE, 10);
gtk_container_set_border_width (GTK_CONTAINER (box2), 10);

/* 另一个选项菜单，这里是用于设置比例构件的更新方式 */
label = gtk_label_new ("Scale Update Policy:");
gtk_box_pack_start (GTK_BOX (box2), label, FALSE, FALSE, 0);
gtk_widget_show (label);

opt = gtk_option_menu_new ();
menu = gtk_menu_new ();

item = make_menu_item ("Continuous",
                       G_CALLBACK (cb_update_menu_select),
                       GINT_TO_POINTER (GTK_UPDATE_CONTINUOUS));
gtk_menu_shell_append (GTK_MENU_SHELL (menu), item);

item = make_menu_item ("Discontinuous",
                       G_CALLBACK (cb_update_menu_select),
                       GINT_TO_POINTER (GTK_UPDATE_DISCONTINUOUS));
gtk_menu_shell_append (GTK_MENU_SHELL (menu), item);

item = make_menu_item ("Delayed",
                       G_CALLBACK (cb_update_menu_select),
                       GINT_TO_POINTER (GTK_UPDATE_DELAYED));
gtk_menu_shell_append (GTK_MENU_SHELL (menu), item);

gtk_option_menu_set_menu (GTK_OPTION_MENU (opt), menu);
gtk_box_pack_start (GTK_BOX (box2), opt, TRUE, TRUE, 0);
gtk_widget_show (opt);

gtk_box_pack_start (GTK_BOX (box1), box2, TRUE, TRUE, 0);
gtk_widget_show (box2);

box2 = gtk_hbox_new (FALSE, 10);
gtk_container_set_border_width (GTK_CONTAINER (box2), 10);

/* 一个水平比例构件，用于调整示例比例构件的显示小数位数。 */
label = gtk_label_new ("Scale Digits:");
gtk_box_pack_start (GTK_BOX (box2), label, FALSE, FALSE, 0);
gtk_widget_show (label);

adj2 = gtk_adjustment_new (1.0, 0.0, 5.0, 1.0, 1.0, 0.0);
g_signal_connect (G_OBJECT (adj2), "value_changed",
                  G_CALLBACK (cb_digits_scale), NULL);
scale = gtk_hscale_new (GTK_ADJUSTMENT (adj2));
gtk_scale_set_digits (GTK_SCALE (scale), 0);
gtk_box_pack_start (GTK_BOX (box2), scale, TRUE, TRUE, 0);
```

```
    gtk_widget_show (scale);

    gtk_box_pack_start (GTK_BOX (box1), box2, TRUE, TRUE, 0);
    gtk_widget_show (box2);

    box2 = gtk_hbox_new (FALSE, 10);
    gtk_container_set_border_width (GTK_CONTAINER (box2), 10);

    /* 以及，最后一个水平比例构件用于调整滚动条的 page size。 */
    label = gtk_label_new ("Scrollbar Page Size:");
    gtk_box_pack_start (GTK_BOX (box2), label, FALSE, FALSE, 0);
    gtk_widget_show (label);

    adj2 = gtk_adjustment_new (1.0, 1.0, 101.0, 1.0, 1.0, 0.0);
    g_signal_connect (G_OBJECT (adj2), "value_changed",
                      G_CALLBACK (cb_page_size), adj1);
    scale = gtk_hscale_new (GTK_ADJUSTMENT (adj2));
    gtk_scale_set_digits (GTK_SCALE (scale), 0);
    gtk_box_pack_start (GTK_BOX (box2), scale, TRUE, TRUE, 0);
    gtk_widget_show (scale);

    gtk_box_pack_start (GTK_BOX (box1), box2, TRUE, TRUE, 0);
    gtk_widget_show (box2);

    separator = gtk_hseparator_new ();
    gtk_box_pack_start (GTK_BOX (box1), separator, FALSE, TRUE, 0);
    gtk_widget_show (separator);

    box2 = gtk_vbox_new (FALSE, 10);
    gtk_container_set_border_width (GTK_CONTAINER (box2), 10);
    gtk_box_pack_start (GTK_BOX (box1), box2, FALSE, TRUE, 0);
    gtk_widget_show (box2);

    button = gtk_button_new_with_label ("Quit");
    g_signal_connect_swapped (G_OBJECT (button), "clicked",
                              G_CALLBACK (gtk_main_quit),
                              NULL);
    gtk_box_pack_start (GTK_BOX (box2), button, TRUE, TRUE, 0);
    GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);
    gtk_widget_grab_default (button);
    gtk_widget_show (button);

    gtk_widget_show (window);
}

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

    create_range_controls ();

    gtk_main ();

    return 0;
}
```

可以注 意到程序没有对 "delete_event" 事件调用 g_signal_connect()，仅仅对 "destroy" 信号调用了该函数。但是 "destroy" 函数一样会执行，因为未经处理的 "delete_event" 事件会引发一个 "destroy" 信号给窗口。

杂项构件

标签 Labels

标签(Labels)是 GTK 中最常用的构件，实际上它很简单。因为没有相关联的 X 窗口，标签不能引发信号。如果需要获取或引发信号，可以将它放在一个事件盒中，或放在按钮构件里面。

用以下函数创建一个新标签:

```
GtkWidget *gtk_label_new( const char *str );

GtkWidget *gtk_label_new_with_mnemonic( const char *str );
```

唯一的参数是要由标签显示的字符串。

创建标签后，要改变标签你的文本，用以下函数:

```
void gtk_label_set_text( GtkLabel   *label,
                         const char *str );
```

第一参数是前面创建的标签(用 GTK_LABEL() 宏转换)，第二个参数是新的字符串。

如果需要，新字符串需要的空间会做自动调整。在字符串中放置换行符，可以创建多行标签。

用以下函数取得标签的当前文本:

```
const gchar* gtk_label_get_text( GtkLabel  *label );
```

不要释放返回的字符串，因为 GTK 内部要使用它。

标签的文本可以用以下函数设置对齐方式:

```
void gtk_label_set_justify( GtkLabel         *label,
                            GtkJustification  jtype );
```

jtype 的值可以是:

GTK_JUSTIFY_LEFT 左对齐
GTK_JUSTIFY_RIGHT 右对齐
GTK_JUSTIFY_CENTER 居中对齐(默认)
GTK_JUSTIFY_FILL 充满

标签构件的文本会自动换行。用以下函数激活“自动换行”:

```
void gtk_label_set_line_wrap (GtkLabel *label,
                              gboolean  wrap);
```

wrap 参数可取 TRUE 或 FALSE。

如果想要使标签加下划线，可以在标签中设置显示模式:

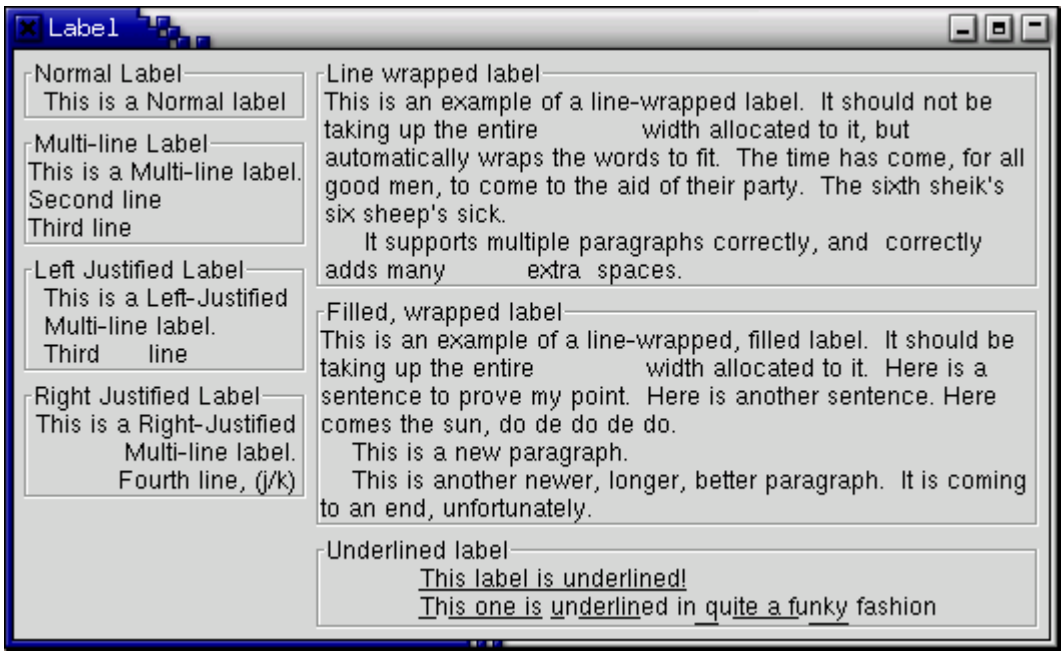
```
void       gtk_label_set_pattern   (GtkLabel         *label,
                                    const gchar       *pattern);
```

pattern 参数指定下划线的外观。它由一串下划线和空格组成。下划线指示标签的相应字符应该加一个下划线。例如,"__ __" 将在标签的第 1、第 2 个字符和第 8、第 9 个字符加下划线。

如 果你只是想创建一个用下划线代表快捷键("mnemonic")的标签，你应该用 gtk_label_new_with_mnemonic() 或 gtk_label_set_text_with_mnemonic()，而不是用 gtk_label_set_pattern()。

下面是一个说明这些函数的短示例。这个示例用框架构件(Frame widget)能更好地示范标签的风格。现在你不用理会这点,框架构件以后再作介绍。

在 GTK+ 2.0 里，标签文本里能包含改变字体等文本属性的标记,并且标签能设置为可以被选择(用来复制-粘贴)。这些高级特性在这里并不介绍。




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

int main( int   argc,
          char *argv[] )
{
  static GtkWidget *window = NULL;
  GtkWidget *hbox;
  GtkWidget *vbox;
  GtkWidget *frame;
  GtkWidget *label;

  /* 初始化 */
  gtk_init (&argc, &argv);

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

  gtk_window_set_title (GTK_WINDOW (window), "Label");
  vbox = gtk_vbox_new (FALSE, 5);
  hbox = gtk_hbox_new (FALSE, 5);
  gtk_container_add (GTK_CONTAINER (window), hbox);
  gtk_box_pack_start (GTK_BOX (hbox), vbox, FALSE, FALSE, 0);
  gtk_container_set_border_width (GTK_CONTAINER (window), 5);

  frame = gtk_frame_new ("Normal Label");
  label = gtk_label_new ("This is a Normal label");
  gtk_container_add (GTK_CONTAINER (frame), label);
  gtk_box_pack_start (GTK_BOX (vbox), frame, FALSE, FALSE, 0);

  frame = gtk_frame_new ("Multi-line Label");
  label = gtk_label_new ("This is a Multi-line label.\nSecond line\n" \
                         "Third line");
  gtk_container_add (GTK_CONTAINER (frame), label);
  gtk_box_pack_start (GTK_BOX (vbox), frame, FALSE, FALSE, 0);

  frame = gtk_frame_new ("Left Justified Label");
  label = gtk_label_new ("This is a Left-Justified\n" \
                         "Multi-line label.\nThird      line");
  gtk_label_set_justify (GTK_LABEL (label), GTK_JUSTIFY_LEFT);
  gtk_container_add (GTK_CONTAINER (frame), label);
  gtk_box_pack_start (GTK_BOX (vbox), frame, FALSE, FALSE, 0);
```

```
  frame = gtk_frame_new ("Right Justified Label");
  label = gtk_label_new ("This is a Right-Justified\nMulti-line label.\n" \
                         "Fourth line, (j/k)");
  gtk_label_set_justify (GTK_LABEL (label), GTK_JUSTIFY_RIGHT);
  gtk_container_add (GTK_CONTAINER (frame), label);
  gtk_box_pack_start (GTK_BOX (vbox), frame, FALSE, FALSE, 0);

  vbox = gtk_vbox_new (FALSE, 5);
  gtk_box_pack_start (GTK_BOX (hbox), vbox, FALSE, FALSE, 0);
  frame = gtk_frame_new ("Line wrapped label");
  label = gtk_label_new ("This is an example of a line-wrapped label.  It " \
                         "should not be taking up the entire            " /* 用一段较长的空白字符来测试空白的自动排列 */\
                         "width allocated to it, but automatically " \
                         "wraps the words to fit.  " \
                         "The time has come, for all good men, to come to " \
                         "the aid of their party.  " \
                         "The sixth sheik's six sheep's sick.\n" \
                         "     It supports multiple paragraphs correctly, " \
                         "and  correctly   adds "\
                         "many         extra  spaces. ");
  gtk_label_set_line_wrap (GTK_LABEL (label), TRUE);
  gtk_container_add (GTK_CONTAINER (frame), label);
  gtk_box_pack_start (GTK_BOX (vbox), frame, FALSE, FALSE, 0);

  frame = gtk_frame_new ("Filled, wrapped label");
  label = gtk_label_new ("This is an example of a line-wrapped, filled label.  " \
                         "It should be taking "\
                         "up the entire             width allocated to it.  " \
                         "Here is a sentence to prove "\
                         "my point.  Here is another sentence. "\
                         "Here comes the sun, do de do de do.\n"\
                         "    This is a new paragraph.\n"\
                         "    This is another newer, longer, better " \
                         "paragraph.  It is coming to an end, "\
                         "unfortunately.");
  gtk_label_set_justify (GTK_LABEL (label), GTK_JUSTIFY_FILL);
  gtk_label_set_line_wrap (GTK_LABEL (label), TRUE);
  gtk_container_add (GTK_CONTAINER (frame), label);
  gtk_box_pack_start (GTK_BOX (vbox), frame, FALSE, FALSE, 0);

  frame = gtk_frame_new ("Underlined label");
  label = gtk_label_new ("This label is underlined!\n"
                         "This one is underlined in quite a funky fashion");
  gtk_label_set_justify (GTK_LABEL (label), GTK_JUSTIFY_LEFT);
  gtk_label_set_pattern (GTK_LABEL (label),
                         "_________________________ _ _________ _ ______     __ _______ ___");
  gtk_container_add (GTK_CONTAINER (frame), label);
  gtk_box_pack_start (GTK_BOX (vbox), frame, FALSE, FALSE, 0);

  gtk_widget_show_all (window);

  gtk_main ();

  return 0;
}
```

箭头 **Arrows**

箭头构件(Arrow widget)画一个箭头，面向几种不同的方向，并有几种不同的风格。在许多应用程序中，常用于创建带箭头的按钮。和标签构件一样，它不能引发信号。

只有两个函数用来操纵箭头构件：

```
GtkWidget *gtk_arrow_new( GtkArrowType   arrow_type,
                          GtkShadowType  shadow_type );
```

```
void gtk_arrow_set( GtkArrow      *arrow,
                    GtkArrowType   arrow_type,
                    GtkShadowType  shadow_type );
```

第一个函数创建新的箭头构件，指明构件的类型和外观；第二个函数用来改变箭头构件类型和外观。arrow_type 参数可以取下列值：

GTK_ARROW_UP 向上
GTK_ARROW_DOWN 向下
GTK_ARROW_LEFT 向左
GTK_ARROW_RIGHT 向右

显然，这些值指示箭头指向哪个方向，shadow_type 参数可以取下列值：

GTK_SHADOW_IN
GTK_SHADOW_OUT (缺省值)
GTK_SHADOW_ETCHED_IN
GTK_SHADOW_ETCHED_OUT

下面是说明这些类型和外观的示例。


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

/* 用指定的参数创建一个箭头构件并将它组装到按钮中 */
GtkWidget *create_arrow_button( GtkArrowType  arrow_type,
                                GtkShadowType shadow_type )
{
  GtkWidget *button;
  GtkWidget *arrow;

  button = gtk_button_new ();
  arrow = gtk_arrow_new (arrow_type, shadow_type);

  gtk_container_add (GTK_CONTAINER (button), arrow);

  gtk_widget_show (button);
  gtk_widget_show (arrow);

  return button;
}

int main( int   argc,
          char *argv[] )
{
  /* GtkWidget 是构件的存储类型 */
  GtkWidget *window;
  GtkWidget *button;
  GtkWidget *box;

  /* 初始化 */
  gtk_init (&argc, &argv);

  /* 创建一个新窗口 */
  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

  gtk_window_set_title (GTK_WINDOW (window), "Arrow Buttons");

  /* 对所有的窗口都这样做是一个好主意 */
```

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

    /* 设置窗口的边框的宽度 */
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* 建一个组装盒以容纳箭头/按钮 */
    box = gtk_hbox_new (FALSE, 0);
    gtk_container_set_border_width (GTK_CONTAINER (box), 2);
    gtk_container_add (GTK_CONTAINER (window), box);

    /* 组装、显示所有的构件 */
    gtk_widget_show (box);

    button = create_arrow_button (GTK_ARROW_UP, GTK_SHADOW_IN);
    gtk_box_pack_start (GTK_BOX (box), button, FALSE, FALSE, 3);

    button = create_arrow_button (GTK_ARROW_DOWN, GTK_SHADOW_OUT);
    gtk_box_pack_start (GTK_BOX (box), button, FALSE, FALSE, 3);

    button = create_arrow_button (GTK_ARROW_LEFT, GTK_SHADOW_ETCHED_IN);
    gtk_box_pack_start (GTK_BOX (box), button, FALSE, FALSE, 3);

    button = create_arrow_button (GTK_ARROW_RIGHT, GTK_SHADOW_ETCHED_OUT);
    gtk_box_pack_start (GTK_BOX (box), button, FALSE, FALSE, 3);

    gtk_widget_show (window);

    /* 进入主循环，等待用户的动作 */
    gtk_main ();

    return 0;
}
```

工具提示对象 The Tooltips Object

工具提示对象(Tooltips)就是当鼠标指针移到按钮或其它构件上并停留几秒时，弹出的文本串。工具提示对象很容易使用，所以在此仅仅对它们进行解释，不再举例。如果你想要看些代码，可以看看 GTK 附带的 testgtk.c 程序.

不接收事件的构件(没有自己的 X 窗口的构件)不能和工具提示对象一起工作。

可以使用 gtk_tooltips_new() 函数创建工具提示对象。因为 GtkTooltips 对象可以重复使用，一般在应用程序中仅需要调用这个函数一次。

```
GtkTooltips *gtk_tooltips_new( void );
```

一旦已创建新的工具提示，并且希望在某个构件上应用它，可调用以下函数设置它：

```
void gtk_tooltips_set_tip( GtkTooltips *tooltips,
                           GtkWidget   *widget,
                           const gchar *tip_text,
                           const gchar *tip_private );
```

第一个参数是已经创建的工具提示对象，其后第二个参数是希望弹出工具提示的构件，第三个参数是要弹出的文本。最后一个参数是作为标识符的文本串，当用 GtkTipsQuery 实现上下文敏感的帮助时要引用该标识符。目前,你可以把它设置为 NULL.

下面有个短示例：

```
GtkTooltips *tooltips;
GtkWidget *button;
.
.
.
tooltips = gtk_tooltips_new ();
```

```
button = gtk_button_new_with_label ("button 1");
.
.
.
gtk_tooltips_set_tip (tooltips, button, "This is button 1", NULL);
```

还有其它与工具提示有关的函数，下面仅仅列出一些函数的简要描述。

```
void gtk_tooltips_enable( GtkTooltips *tooltips );
```

激活已经禁用的工具提示对象。

```
void gtk_tooltips_disable( GtkTooltips *tooltips );
```

禁用已经激活的工具提示对象。

上面是所有与工具提示有关的函数，实际上比你想要知道的还多 :-)

进度条 Progress Bars

进度条用于显示正在进行的操作的状态。在下面的代码中可以看出，它相当容易使用。下面的内容从创建一个新进度条开始。

```
GtkWidget *gtk_progress_bar_new( void );
```

创建进度条后就可以使用它了。

```
void gtk_progress_bar_set_fraction ( GtkProgressBar *pbar,
                                     gdouble        fraction );
```

第一个参数是希望操作的进度条，第二个参数是“已完成”的百分比，意思是进度条从 0-100%已经填充的数量。它以 0～1 范围的实数传递给函数。

GTK 1.2 版已经给进度条添加了一个新的功能，那就是允许它以不同的方法显示其值，并通知用户它的当前值和范围。

进度条可以用以下函数设置它的移动方向：

```
void gtk_progress_bar_set_orientation( GtkProgressBar *pbar,
                                       GtkProgressBarOrientation orientation );
```

orientation 参数可以取下列值之一，以指示进度条的移动方向：

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

除了指示进度已经发生的数量以外，进度条还可以设置为仅仅指示有活动在继续,即活动状态。这在进度无法按数值度量的情况下很有用。用下面的函数来表明进度有了些进展。

```
void gtk_progress_bar_pulse ( GtkProgressBar *progress );
```

活动指示的步数由以下函数设置：

```
void gtk_progress_bar_set_pulse_step( GtkProgressBar *pbar,
                                      gdouble        fraction );
```

在非活动状态下，进度条可以用下列函数在滑槽里显示一个可配置的文本串：

```
void gtk_progress_bar_set_text( GtkProgressBar *progress,
                                const gchar    *text );
```

注意,gtk_progress_set_text()不再支持 GTK+ 1.2 版进度条里那种类似 printf()的格式参数

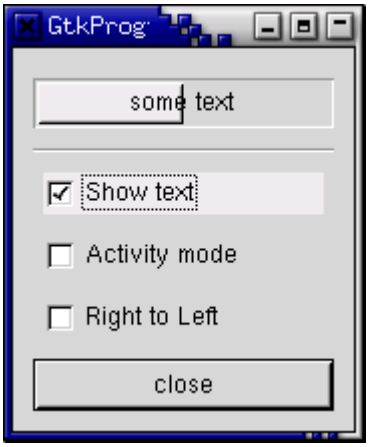
你可以通过调用 gtk_progess_bar_set_text() 并把 NULL 作为第二个参数来关闭文本串的显示.

进度条的当前文本设置能由下面的函数取得。不要释放返回的字符串.

```
const gchar *gtk_progress_bar_get_text( GtkProgressBar *pbar );
```

进度条通常和 timeouts 或其它类似函数同时使用(详见超时、I/O 和 Idle 函数这一章)，使应用程序就像是多任务一样。一般都以同样的方式调用 gtk_progress_bar_set_fraction() 或 gtk_progress_bar_pulse() 函数。

下面是一个进度条的示例，用 timeout 函数更新进度条的值。代码也演示了怎样复位进度条。




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

typedef struct _ProgressData {
  GtkWidget *window;
  GtkWidget *pbar;
  int timer;
  gboolean activity_mode;
} ProgressData;

/* 更新进度条，这样就能够看到进度条的移动 */
gint progress_timeout( gpointer data )
{
  ProgressData *pdata = (ProgressData *)data;
  gdouble new_val;

  if (pdata->activity_mode)
    gtk_progress_bar_pulse (GTK_PROGRESS_BAR (pdata->pbar));
  else
    {
      /* 使用在调整对象中设置的取值范围计算进度条的值 */

      new_val = gtk_progress_bar_get_fraction (GTK_PROGRESS_BAR (pdata->pbar)) + 0.01;

      if (new_val > 1.0)
        new_val = 0.0;

      /* 设置进度条的新值 */
      gtk_progress_bar_set_fraction (GTK_PROGRESS_BAR (pdata->pbar), new_val);
    }

  /* 这是一个 timeout 函数，返回 TRUE，这样它就能够继续被调用 */
  return TRUE;
}

/* 回调函数，切换在进度条你的滑槽上的文本显示 */
void toggle_show_text( GtkWidget    *widget,
```

```
                              ProgressData *pdata )
{
  const gchar *text;

  text = gtk_progress_bar_get_text (GTK_PROGRESS_BAR (pdata->pbar));
  if (text && *text)
    gtk_progress_bar_set_text (GTK_PROGRESS_BAR (pdata->pbar), "");
  else
    gtk_progress_bar_set_text (GTK_PROGRESS_BAR (pdata->pbar), "some text");
}

/* 回调函数，切换进度条的活动模式 */
void toggle_activity_mode( GtkWidget    *widget,
                           ProgressData *pdata )
{
  pdata->activity_mode = !pdata->activity_mode;
  if (pdata->activity_mode)
       gtk_progress_bar_pulse (GTK_PROGRESS_BAR (pdata->pbar));
  else
       gtk_progress_bar_set_fraction (GTK_PROGRESS_BAR (pdata->pbar), 0.0);
}


/* 回调函数，切换进度条的移动方向 */
void toggle_orientation( GtkWidget    *widget,
                         ProgressData *pdata )
{
  switch (gtk_progress_bar_get_orientation (GTK_PROGRESS_BAR (pdata->pbar))) {
  case GTK_PROGRESS_LEFT_TO_RIGHT:
    gtk_progress_bar_set_orientation (GTK_PROGRESS_BAR (pdata->pbar),
                                      GTK_PROGRESS_RIGHT_TO_LEFT);
    break;
  case GTK_PROGRESS_RIGHT_TO_LEFT:
    gtk_progress_bar_set_orientation (GTK_PROGRESS_BAR (pdata->pbar),
                                      GTK_PROGRESS_LEFT_TO_RIGHT);
    break;
  default:
    // 什么也不做
  }
}


/* 清除分配的内存，删除定时器(timer) */
void destroy_progress( GtkWidget     *widget,
                       ProgressData *pdata)
{
    gtk_timeout_remove (pdata->timer);
    pdata->timer = 0;
    pdata->window = NULL;
    g_free (pdata);
    gtk_main_quit ();
}

int main( int   argc,
          char *argv[])
{
    ProgressData *pdata;
    GtkWidget *align;
    GtkWidget *separator;
    GtkWidget *table;
    GtkWidget *button;
    GtkWidget *check;
    GtkWidget *vbox;

    gtk_init (&argc, &argv);

    /* 为传递到回调函数中的数据分配内存 */
```

```
pdata = g_malloc (sizeof (ProgressData));

pdata->window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_window_set_resizable (GTK_WINDOW (pdata->window), TRUE);

g_signal_connect (G_OBJECT (pdata->window), "destroy",
                  G_CALLBACK (destroy_progress),
                  pdata);
gtk_window_set_title (GTK_WINDOW (pdata->window), "GtkProgressBar");
gtk_container_set_border_width (GTK_CONTAINER (pdata->window), 0);

vbox = gtk_vbox_new (FALSE, 5);
gtk_container_set_border_width (GTK_CONTAINER (vbox), 10);
gtk_container_add (GTK_CONTAINER (pdata->window), vbox);
gtk_widget_show (vbox);

/* 创建一个居中对齐的对象 */
align = gtk_alignment_new (0.5, 0.5, 0, 0);
gtk_box_pack_start (GTK_BOX (vbox), align, FALSE, FALSE, 5);
gtk_widget_show (align);

/* 创建进度条 */
pdata->pbar = gtk_progress_bar_new ();

gtk_container_add (GTK_CONTAINER (align), pdata->pbar);
gtk_widget_show (pdata->pbar);

/* 加一个定时器(timer)，以更新进度条的值 */
pdata->timer = gtk_timeout_add (100, progress_timeout, pdata);

separator = gtk_hseparator_new ();
gtk_box_pack_start (GTK_BOX (vbox), separator, FALSE, FALSE, 0);
gtk_widget_show (separator);

/* 行数、列数、同质性(homogeneous) */
table = gtk_table_new (2, 2, FALSE);
gtk_box_pack_start (GTK_BOX (vbox), table, FALSE, TRUE, 0);
gtk_widget_show (table);

/* 添加一个复选按钮，以选择是否显示在滑槽里的文本 */
check = gtk_check_button_new_with_label ("Show text");
gtk_table_attach (GTK_TABLE (table), check, 0, 1, 0, 1,
                  GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
                  5, 5);
g_signal_connect (G_OBJECT (check), "clicked",
                  G_CALLBACK (toggle_show_text),
                  pdata);
gtk_widget_show (check);

/* 添加一个复选按钮，切换活动状态 */
check = gtk_check_button_new_with_label ("Activity mode");
gtk_table_attach (GTK_TABLE (table), check, 0, 1, 1, 2,
                  GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
                  5, 5);
g_signal_connect (G_OBJECT (check), "clicked",
                  G_CALLBACK (toggle_activity_mode),
                  pdata);
gtk_widget_show (check);

/* 添加一个复选按钮，切换移动方向 */
check = gtk_check_button_new_with_label ("Right to Left");
gtk_table_attach (GTK_TABLE (table), check, 0, 1, 2, 3,
                  GTK_EXPAND | GTK_FILL, GTK_EXPAND | GTK_FILL,
                  5, 5);
g_signal_connect (G_OBJECT (check), "clicked",
                  G_CALLBACK (toggle_orientation),
                  pdata);
```

```
    gtk_widget_show (check);

    /* 添加一个按钮，用来退出应用程序 */
    button = gtk_button_new_with_label ("close");
    g_signal_connect_swapped (G_OBJECT (button), "clicked",
                              G_CALLBACK (gtk_widget_destroy),
                              pdata->window);
    gtk_box_pack_start (GTK_BOX (vbox), button, FALSE, FALSE, 0);

    /* 将按钮设置为能缺省的构件 */
    GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);

    /* 将缺省焦点设置到这个按钮上，使之成为缺省按钮，只要按回车键
     * 就相当于点击了这个按钮 */
       //译者注： 能缺省的构件在获取焦点后成为缺省构件,用户按方向键等可以切换焦点。
    gtk_widget_grab_default (button);
    gtk_widget_show (button);

    gtk_widget_show (pdata->window);

    gtk_main ();

    return 0;
}
```

对话框 Dialogs

对话构件非常简单，事实上它仅仅是一个预先组装了几个构件到里面的窗口。对话框的数据结构是：

```
struct GtkDialog
{
       GtkWindow window;

       GtkWidget *vbox;
       GtkWidget *action_area;
};
```

从上面可以看到，对话框只是简单地创建一个窗口，并在顶部组装一个纵向盒(vbox)，然后在这个纵向盒中组装一个分隔线(separator)，再加一个称为“活动区(action_area)”的横向盒(hbox).

对话框构件可以用于弹出消息，或者其它类似的任务。这里有两个函数来创建一个新的对话框：

```
GtkWidget *gtk_dialog_new( void );

GtkWidget *gtk_dialog_new_with_buttons( const gchar     *title,
                                        GtkWindow      *parent,
                                        GtkDialogFlags  flags,
                                        const gchar    *first_button_text,
                                        ... );
```

第一个函数将创建一个空地对话框，你现在就可以使用它了。你可以组装一个按钮到它的活动区(action_area)，就像下面这样：

```
    button = ...
    gtk_box_pack_start (GTK_BOX (GTK_DIALOG (window)->action_area),
                        button, TRUE, TRUE, 0);
    gtk_widget_show (button);
```

你可以通过组装来扩充活动区，比如，增加一个标签，可以像下面这样做：

```
    label = gtk_label_new ("Dialogs are groovy");
    gtk_box_pack_start (GTK_BOX (GTK_DIALOG (window)->vbox),
                        label, TRUE, TRUE, 0);
```

```
gtk_widget_show (label);
```

作为一个示例，可以在活动区里面组装两个按钮：一个“取消”按钮和一个“确定”按钮，再在纵向盒(vbox)里组装一个标签，以便向用户提出疑问，或显示一个错误信息等。然后可以把不同信号连接到每个按钮，对用户的选择进行响应。

如果由对话框提供的纵向和横向盒的简单功能不能满足你的需要，可以简单地在组装盒中添加其它布局构件。例如，可以在纵向盒中添加一个组装表(table)。

更复杂的 gtk_dialog_new_with_buttons() 函数允许你设置下面的一个或多个参数。

GTK_DIALOG_MODAL

使对话框使用独占模式。

GTK_DIALOG_DESTROY_WITH_PARENTS

保证对话框在指定父窗口被关闭时也一起关闭。

GTK_DIALOG_NO_SEPARATOR

省略纵向盒与活动区之间的分隔线

标尺 **Rulers**

标尺构件(Ruler widgets)一般用于在给定窗口中指示鼠标指针的位置。一个窗口可以有一个横跨整个窗口宽度的水平标尺和一个占据整个窗口高度的垂直标尺。标尺上有一个小三角形的指示器标出鼠标指针相对于标尺的精确位置。

首先，必须创建标尺。水平和垂直标尺用下面的函数创建：

```
GtkWidget *gtk_hruler_new( void );    /* 水平标尺 */

GtkWidget *gtk_vruler_new( void );    /* 垂直标尺 */
```

一旦创建了标尺，我们就能指定它的度量单位。标尺的度量单位可以是 GTK_PIXELS，GTK_INCHES 或 GTK_CENTIMETERS。可以用下面的函数设置：

```
void gtk_ruler_set_metric( GtkRuler      *ruler,
                           GtkMetricType  metric );
```

默认的度量单位是 GTK_PIXELS。

```
gtk_ruler_set_metric( GTK_RULER(ruler), GTK_PIXELS );
```

标尺构件的另一个重要属性是怎样标志刻度单位以及位置指示器一开始应该放在哪里。可以用下面的函数设置：

```
void gtk_ruler_set_range( GtkRuler *ruler,
                          gdouble   lower,
                          gdouble   upper,
                          gdouble   position,
                          gdouble   max_size );
```

其中 lower 和 upper 参数定义标尺的范围，max_size 是要显示的最大可能数值。Position 定义了标尺的指针指示器的初始位置。

下面这句使垂直标尺能跨越 800 像素宽的窗口。

```
gtk_ruler_set_range( GTK_RULER(vruler), 0, 800, 0, 800);
```

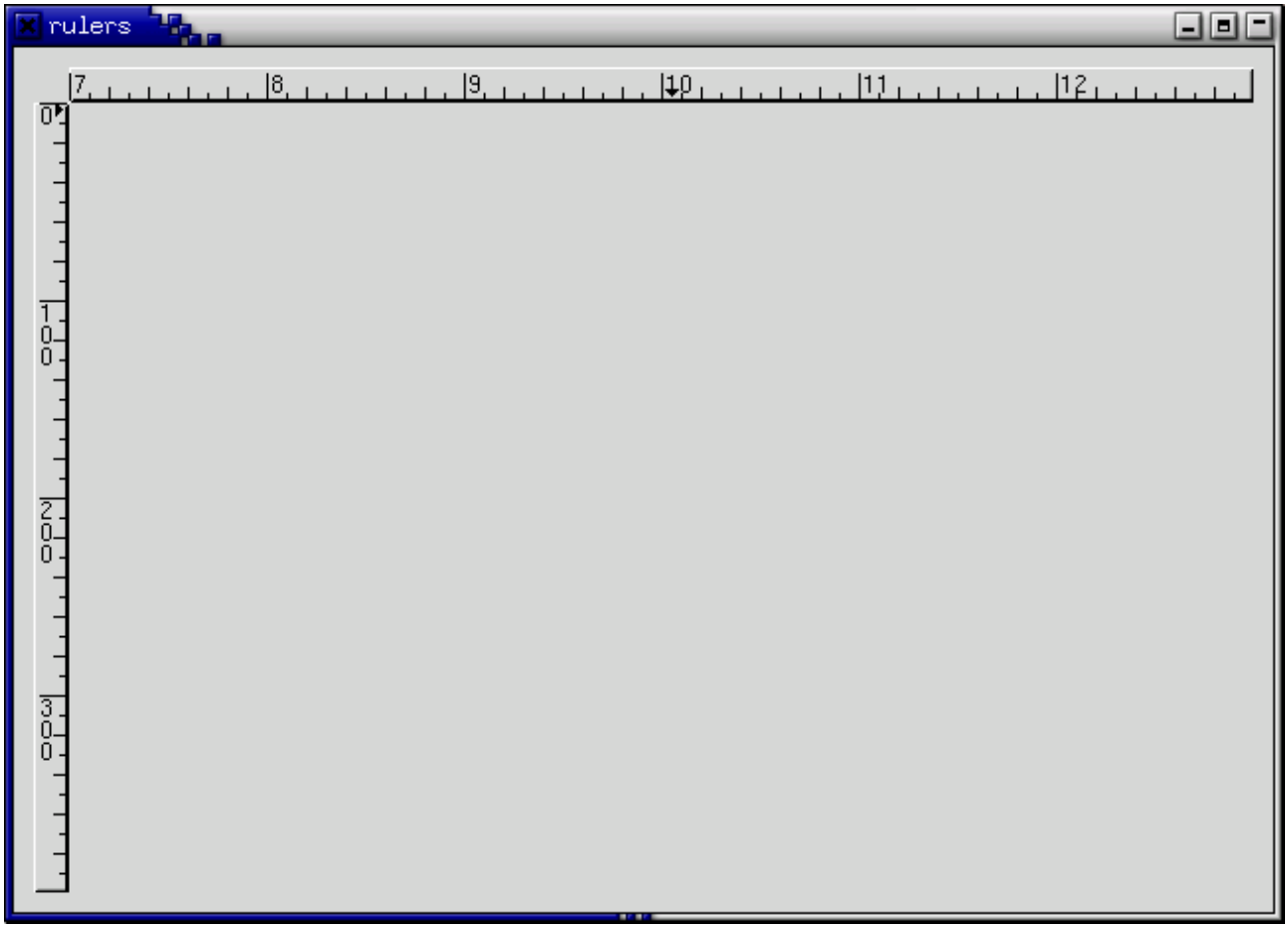
标尺上显示标志会从 0 到 800，每 100 个像素一个数字。如果我们想让标尺的范围为从 7 到 16，可以使用下面的代码：

```
gtk_ruler_set_range( GTK_RULER(vruler), 7, 16, 0, 20);
```

标 尺上的指示器是一个小三角形的标记，指示鼠标指针相对于标尺的位置。如果标尺是用于跟踪鼠标器指针的，应该将 motion_notify_event 信号 连接到标尺的 motion_notify_event 方法(method)。要跟踪鼠标在整个窗口区域你的移动，应该这样做:

```
#define EVENT_METHOD(i, x) GTK_WIDGET_GET_CLASS(i)->x

    g_signal_connect_swapped (G_OBJECT (area), "motion_notify_event",
            G_CALLBACK (EVENT_METHOD (ruler, motion_notify_event)),
            G_OBJECT (ruler));
```

下 列示例创建一个绘图区(drawing area)，上面加一个水平标尺，左边加一个垂直标尺。绘图区的大小是 600 像素宽 ×400 像素高。水平标尺范围是从 7 到 13，每 100 像素加一个刻度； 垂直标尺范围从 0 到 400，每 100 像素加一个刻度。绘图区和标尺的定位是用一个组装表(table)实现的。




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

#define EVENT_METHOD(i, x) GTK_WIDGET_GET_CLASS(i)->x

#define XSIZE  600
#define YSIZE  400

/* 当点击"close"按钮时，退出应用程序 */
gint close_application( GtkWidget *widget,
                        GdkEvent  *event,
                        gpointer   data )
{
    gtk_main_quit ();
    return FALSE;
}

/* 主函数 */
int main( int   argc,
```

```
        char *argv[] ) {
    GtkWidget *window, *table, *area, *hrule, *vrule;

    /* 初始化，创建主窗口 */
    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    g_signal_connect (G_OBJECT (window), "delete_event",
                      G_CALLBACK (close_application), NULL);
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* 创建一个组装表，绘图区和标尺放在里面 */
    table = gtk_table_new (3, 2, FALSE);
    gtk_container_add (GTK_CONTAINER (window), table);

    area = gtk_drawing_area_new ();
    gtk_widget_set_size_request (GTK_WIDGET (area), XSIZE, YSIZE);
    gtk_table_attach (GTK_TABLE (table), area, 1, 2, 1, 2,
                      GTK_EXPAND|GTK_FILL, GTK_FILL, 0, 0);
    gtk_widget_set_events (area, GDK_POINTER_MOTION_MASK |
                                 GDK_POINTER_MOTION_HINT_MASK);

    /* 水平标尺放在顶部。鼠标移动穿过绘图区时，一个
     * motion_notify_event 被传递给标尺的相应的事件处理函数 */
    hrule = gtk_hruler_new ();
    gtk_ruler_set_metric (GTK_RULER (hrule), GTK_PIXELS);
    gtk_ruler_set_range (GTK_RULER (hrule), 7, 13, 0, 20);
    g_signal_connect_swapped (G_OBJECT (area), "motion_notify_event",
                              G_CALLBACK (EVENT_METHOD (hrule, motion_notify_event)),
                              hrule);
    gtk_table_attach (GTK_TABLE (table), hrule, 1, 2, 0, 1,
                      GTK_EXPAND|GTK_SHRINK|GTK_FILL, GTK_FILL, 0, 0);

    /* 垂直标尺放在左边。当鼠标移动穿过绘图区时，一个
     * motion_notify_event 被传递到标尺相应的事件处理函数中 */
    vrule = gtk_vruler_new ();
    gtk_ruler_set_metric (GTK_RULER (vrule), GTK_PIXELS);
    gtk_ruler_set_range (GTK_RULER (vrule), 0, YSIZE, 10, YSIZE );
    g_signal_connect_swapped (G_OBJECT (area), "motion_notify_event",
                              G_CALLBACK (EVENT_METHOD (vrule, motion_notify_event)),
                              vrule);
    gtk_table_attach (GTK_TABLE (table), vrule, 0, 1, 1, 2,
                      GTK_FILL, GTK_EXPAND|GTK_SHRINK|GTK_FILL, 0, 0);

    /* 现在显示所有的构件 */
    gtk_widget_show (area);
    gtk_widget_show (hrule);
    gtk_widget_show (vrule);
    gtk_widget_show (table);
    gtk_widget_show (window);
    gtk_main ();

    return 0;
}
```

状态栏 Statusbars

状态栏(Statusbars)是一些简单的构件，一般用来显示文本消息。它将文本消息压入到一个栈里面，当弹出当前消息时，将重新显示前一条文本消息。

为 了让应用程序的不同部分使用同一个状态栏显示消息，状态栏构件使用上下文标识符(Context Identifiers)来识别不同“用户”。在栈顶部的消息就是要显示的消息，不管它的上下文是什么。消息在栈里面是以后进先出(last-in- first-out)的方式保存的，而不是按上下文标识符顺序。

状态栏构件用下面的函数创建：

```
GtkWidget *gtk_statusbar_new( void );
```

用一个上下文的简短文本描述调用下面的函数，可以获得新的上下文标识符：

```
guint gtk_statusbar_get_context_id( GtkStatusbar *statusbar,
                                    const gchar  *context_description );
```

有三个函数用来操作状态栏：

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

void gtk_statusbar_pop( GtkStatusbar *statusbar)
                        guint         context_id );

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

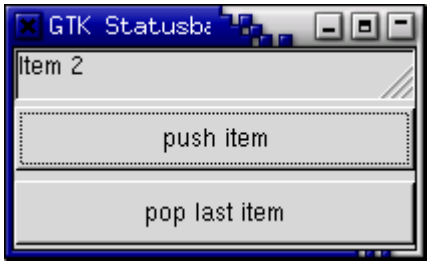
第一个函数 gtk_statusbar_push() 用于将新消息加到状态栏中，它返回一个消息标识符(Message Identifier)。这个标识符可以和上下文标识符一起传给 gtk_statusbar_remove 函数以将该消息从状态栏的栈中删除。

函数 gtk_statusbar_pop() 删除在栈中给定上下文标识符的最上面的一条消息。

除了显示消息，状态栏还可以显示一个大小改变把柄(resize grip)，用户可以用鼠标拖动它来改变窗口的大小，就像拖动窗口边框一样．下面的函数控制大小改变把柄的显示．

```
void     gtk_statusbar_set_has_resize_grip( GtkStatusbar *statusbar,
                                            gboolean      setting );

gboolean gtk_statusbar_get_has_resize_grip( GtkStatusbar *statusbar );
```

下面的示例创建了一个状态栏和两个按钮，一个将消息压入到状态栏栈中，另一个将最上面一条消息弹出。




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

GtkWidget *status_bar;

void push_item( GtkWidget *widget,
                gpointer   data )
{
  static int count = 1;
  char buff[20];

  g_snprintf (buff, 20, "Item %d", count++);
  gtk_statusbar_push (GTK_STATUSBAR (status_bar), GPOINTER_TO_INT (data), buff);

  return;
}

void pop_item( GtkWidget *widget,
               gpointer   data )
{
```

```
  gtk_statusbar_pop (GTK_STATUSBAR (status_bar), GPOINTER_TO_INT (data));
  return;
}

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

    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *button;

    gint context_id;

    gtk_init (&argc, &argv);

    /* 创建新窗口 */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_widget_set_size_request (GTK_WIDGET (window), 200, 100);
    gtk_window_set_title (GTK_WINDOW (window), "GTK Statusbar Example");
    g_signal_connect (G_OBJECT (window), "delete_event",
                      G_CALLBACK (exit), NULL);

    vbox = gtk_vbox_new (FALSE, 1);
    gtk_container_add (GTK_CONTAINER (window), vbox);
    gtk_widget_show (vbox);

    status_bar = gtk_statusbar_new ();
    gtk_box_pack_start (GTK_BOX (vbox), status_bar, TRUE, TRUE, 0);
    gtk_widget_show (status_bar);

    context_id = gtk_statusbar_get_context_id(
                          GTK_STATUSBAR (status_bar), "Statusbar example");

    button = gtk_button_new_with_label ("push item");
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (push_item), GINT_TO_POINTER (context_id));
    gtk_box_pack_start (GTK_BOX (vbox), button, TRUE, TRUE, 2);
    gtk_widget_show (button);

    button = gtk_button_new_with_label ("pop last item");
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (pop_item), GINT_TO_POINTER (context_id));
    gtk_box_pack_start (GTK_BOX (vbox), button, TRUE, TRUE, 2);
    gtk_widget_show (button);

    /* 将窗口最后显示，让整个窗口一次性出现在屏幕上． */
    gtk_widget_show (window);

    gtk_main ();

    return 0;
}
```

文本输入构件 **Text Entries**

文本输入构件(Entry widget)允许在一个单行文本框里输入和显示一行文本。文本可以用函数进行操作，如将新的文本替换、前插、追加到文本输入构件的当前内容中。

用下面的函数创建一个文本输入构件:

```
GtkWidget *gtk_entry_new( void );
```

下面的函数改变文本输入构件当前的文本内容。

```
void gtk_entry_set_text( GtkEntry    *entry,
```

```
                                const gchar *text );
```

gtk_entry_set_text() 函数用新的内容(contents)取代文本输入构件当前的内容。你可以注意到文本输入构件的类(class Entry)体现了可编辑的接口(Editable interface)(是的,gobject 提供了类似 Java 的接口),它包含更多的函数来操作内容。

文本输入构件的内容可以用下面的函数获取。这在下面介绍的回调函数中是很有用的。

```
const gchar *gtk_entry_get_text( GtkEntry *entry );
```

这个函数返回的值在其内部被使用，不要用 free()或 g_free()释放它。

如果我们不想用户通过输入文字改变文本输入构件的内容，我们可以改变它的可编辑状态。

```
void gtk_editable_set_editable( GtkEditable *entry,
                                gboolean     editable );
```

上面的函数可以让我们通过传递一个 TRUE 或 FALSE 值作为 editable 参数来改变文本输入构件的可编辑状态。

如果想让文本输入构件输入的文本不回显(比如用于接收口令)，可以使用下面的函数，它也是取一个布尔值作为参数

```
void gtk_entry_set_visibility( GtkEntry *entry,
                               gboolean  visible );
```

文本你的某一部分可以用下面的函数设置为被选中。这个常在为文本输入构件设置了一个缺省值时使用，以方便用户删除它。

```
void gtk_editable_select_region( GtkEditable *entry,
                                 gint        start,
                                 gint        end );
```

如果我们想在用户输入文本时进行响应，可以为 activate 或 changed 信号设置回调函数。当用户在文本输入构件内部按回车键时引发 Activate 信号；在每次文本输入构件的文本发生变化时引发 Changed 信号，比如，每输入或删除一个字符。

下面的代码是一个使用文本输入构件的示例。


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

void enter_callback( GtkWidget *widget,
                     GtkWidget *entry )
{
  const gchar *entry_text;
  entry_text = gtk_entry_get_text (GTK_ENTRY (entry));
  printf("Entry contents: %s\n", entry_text);
}

void entry_toggle_editable( GtkWidget *checkbutton,
                            GtkWidget *entry )
{
  gtk_editable_set_editable (GTK_EDITABLE (entry),
                             GTK_TOGGLE_BUTTON (checkbutton)->active);
}

void entry_toggle_visibility( GtkWidget *checkbutton,
                              GtkWidget *entry )
```

```
{
  gtk_entry_set_visibility (GTK_ENTRY (entry),
                            GTK_TOGGLE_BUTTON (checkbutton)->active);
}

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

    GtkWidget *window;
    GtkWidget *vbox, *hbox;
    GtkWidget *entry;
    GtkWidget *button;
    GtkWidget *check;
    gint tmp_pos;

    gtk_init (&argc, &argv);

    /* 创建一个新窗口 */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_widget_set_size_request (GTK_WIDGET (window), 200, 100);
    gtk_window_set_title (GTK_WINDOW (window), "GTK Entry");
    g_signal_connect (G_OBJECT (window), "destroy",
                      G_CALLBACK (gtk_main_quit), NULL);
    g_signal_connect_swapped (G_OBJECT (window), "delete_event",
                              G_CALLBACK (gtk_widget_destroy),
                              window);

    vbox = gtk_vbox_new (FALSE, 0);
    gtk_container_add (GTK_CONTAINER (window), vbox);
    gtk_widget_show (vbox);

    entry = gtk_entry_new ();
    gtk_entry_set_max_length (GTK_ENTRY (entry), 50);
    g_signal_connect (G_OBJECT (entry), "activate",
                      G_CALLBACK (enter_callback),
                      entry);
    gtk_entry_set_text (GTK_ENTRY (entry), "hello");
    tmp_pos = GTK_ENTRY (entry)->text_length;
    gtk_editable_insert_text (GTK_EDITABLE (entry), " world", -1, &tmp_pos);
    gtk_editable_select_region (GTK_EDITABLE (entry),
                                0, GTK_ENTRY (entry)->text_length);
    gtk_box_pack_start (GTK_BOX (vbox), entry, TRUE, TRUE, 0);
    gtk_widget_show (entry);

    hbox = gtk_hbox_new (FALSE, 0);
    gtk_container_add (GTK_CONTAINER (vbox), hbox);
    gtk_widget_show (hbox);

    check = gtk_check_button_new_with_label ("Editable");
    gtk_box_pack_start (GTK_BOX (hbox), check, TRUE, TRUE, 0);
    g_signal_connect (G_OBJECT (check), "toggled",
                      G_CALLBACK (entry_toggle_editable), entry);
    gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (check), TRUE);
    gtk_widget_show (check);

    check = gtk_check_button_new_with_label ("Visible");
    gtk_box_pack_start (GTK_BOX (hbox), check, TRUE, TRUE, 0);
    g_signal_connect (G_OBJECT (check), "toggled",
                      G_CALLBACK (entry_toggle_visibility), entry);
    gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (check), TRUE);
    gtk_widget_show (check);

    button = gtk_button_new_from_stock (GTK_STOCK_CLOSE);
    g_signal_connect_swapped (G_OBJECT (button), "clicked",
                              G_CALLBACK (gtk_widget_destroy),
                              window);
    gtk_box_pack_start (GTK_BOX (vbox), button, TRUE, TRUE, 0);
```

```
    GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);
    gtk_widget_grab_default (button);
    gtk_widget_show (button);

    gtk_widget_show (window);

    gtk_main();

    return 0;
}
```

微调按钮 Spin Buttons

微调按钮(Spin Button)构件通常用于让用户从一个取值范围你选择一个值。它由一个文本输入框和旁边的向上和向下两个按钮组成。点击某一个按钮会让文本输入框你的数值大小在一定范围你改变。文本输入框里也可以直接输入一个特定值。

微调按钮构件允许其中的数值没有小数位或具有指定的小数位，并且数值可以按一种可配置的方式增加或减小。在按钮较长时间呈按下状态时，构件的数值会根据工具按下时间的长短加速变化。

微调按钮用一个调整对象来维护该按钮能够取值的范围。微调按钮构件因此而具有了很强大的功能。

下面是创建调整对象的函数。这里的用意是展示其中所包含的数值的意义：

```
GtkObject *gtk_adjustment_new( gdouble value,
                               gdouble lower,
                               gdouble upper,
                               gdouble step_increment,
                               gdouble page_increment,
                               gdouble page_size );
```

调整对象的这些属性在微调按钮构件中有如下用处：

- value：微调按钮构件的初始值
- lower：构件允许的最小值
- upper：构件允许的最大值
- step_increment：当鼠标左键按下时构件一次增加/减小的值
- page_increment：当鼠标右键按下时构件一次增加/减小的值
- page_size：没有用到

另外，当用鼠标中间键点击按钮时，可以直接跳到构件的 upper 或 lower 值。下面看看怎样创建一个微调按钮构件：

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

其中的 climb_rate 参数是介于 0.0 和 1.0 间的值，指明构件数值变化的加速度(长时间按住按钮，数值会加速变化)。digits 参数指定要显示的值的小数位数。

创建微调按钮构件之后，还可以用下面的函数对其重新配置：

```
void gtk_spin_button_configure( GtkSpinButton *spin_button,
                                GtkAdjustment *adjustment,
                                gdouble        climb_rate,
                                guint          digits );
```

其中 spin_button 参数就是要重新配置的构件。其它的参数与创建时的意思相同。

使用下面的两个函数可以设置或获取构件内部使用的调整对象：

```
void gtk_spin_button_set_adjustment( GtkSpinButton  *spin_button,
                                     GtkAdjustment  *adjustment );
```

```
GtkAdjustment *gtk_spin_button_get_adjustment( GtkSpinButton *spin_button );
```

显示数值的小数位数可以用下面的函数改变：

```
void gtk_spin_button_set_digits( GtkSpinButton *spin_button,
                                 guint          digits) ;
```

微调按钮上当前显示的数值可以用下面的函数改变：

```
void gtk_spin_button_set_value( GtkSpinButton *spin_button,
                                gdouble        value );
```

微调按钮构件的当前值可以以浮点数或整数的形式获得。使用下面的函数：

```
gdouble gtk_spin_button_get_value ( GtkSpinButton *spin_button );

gint gtk_spin_button_get_value_as_int( GtkSpinButton *spin_button );
```

如果想以当前值为基数改变微调按钮的值，可以使用下面的函数：

```
void gtk_spin_button_spin( GtkSpinButton *spin_button,
                           GtkSpinType    direction,
                           gdouble        increment );
```

其中，direction 参数可以取下面的值：

```
GTK_SPIN_STEP_FORWARD
GTK_SPIN_STEP_BACKWARD
GTK_SPIN_PAGE_FORWARD
GTK_SPIN_PAGE_BACKWARD
GTK_SPIN_HOME
GTK_SPIN_END
GTK_SPIN_USER_DEFINED
```

这个函数中包含的一些功能将在下面详细介绍。其中的许多设置都使用了与微调按钮构件相关联的调整对象的值。

GTK_SPIN_STEP_FORWARD 和 GTK_SPIN_STEP_BACKWARD 将构件的值按 increment 参数指定的数值增大或减小，除非 increment 参数是 0。这种情况下，构件的值将按与其相关联的调整对象的 step_increment 值改变。

GTK_SPIN_PAGE_FORWARD 和 GTK_SPIN_PAGE_BACKWARD 只是简单地按 increment 参数改变微调按钮构件的值。

GTK_SPIN_HOME 将构件的值设置为相关联调整对象的范围的最小值。

GTK_SPIN_END 将构件的值设置为相关联调整对象的范围的最大值。

GTK_SPIN_USER_DEFINED 简单地按指定的数值改变构件的数值。

介绍了设置和获取微调按钮的范围属性的函数，下面再介绍影响微调按钮构件的外观和行为的函数。

要介绍的第一个函数就是限制微调按钮构件的文本框只能输入数值。这样就阻止了用户输入任何非法的字符：

```
void gtk_spin_button_set_numeric( GtkSpinButton *spin_button,
                                  gboolean       numeric );
```

可以设置让微调按钮构件在 upper 和 lower 之间循环。也就是当达到最大值后再向上调整回到最小值，当达到最小值后再向下调整变为最大值。可以用下面的函数实现：

```
void gtk_spin_button_set_wrap( GtkSpinButton *spin_button,
                               gboolean       wrap );
```

可以设置让微调按钮构件将其值圆整到最接近 step_increment 的值(在该微调按钮构件使用的调整对象中设置的)。用下面的函数实现：

```
void gtk_spin_button_set_snap_to_ticks( GtkSpinButton  *spin_button,
                                        gboolean        snap_to_ticks );
```

微调按钮构件的更新方式可以用下面的函数改变：

```
void gtk_spin_button_set_update_policy( GtkSpinButton  *spin_button,
                                        GtkSpinButtonUpdatePolicy policy );
```

其中 policy 参数可以取 GTK_UPDATE_ALWAYS 或 GTK_UPDATE_IF_VALID。

这些更新方式影响微调按钮构件在解析插入文本并将其值与调整对象的值同步时的行为。

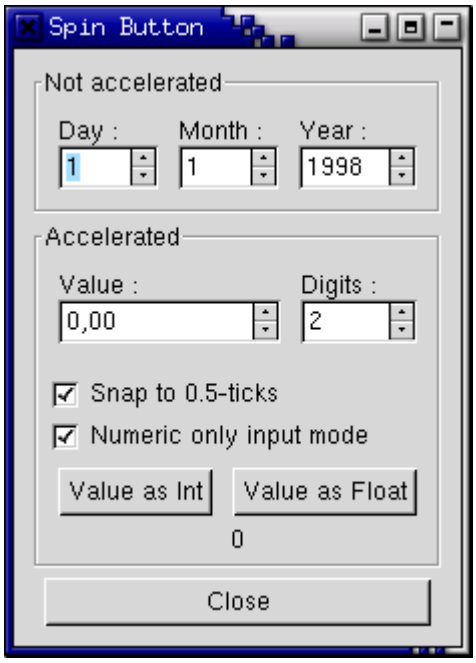
在 GTK_UPDATE_IF_VALID 方式下，微调按钮构件只有在输入文本是其调整对象指定范围你合法的值时才进行更新，否则文本会被重置为当前的值。

在 GTK_UPDATE_ALWAYS 方式下，我们将忽略在文本转换为数值时的错误。

最后，可以强行要求微调按钮构件更新自己：

```
void gtk_spin_button_update( GtkSpinButton  *spin_button );
```

下面是一个使用微调按钮构件的示例。




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

static GtkWidget *spinner1;

void toggle_snap( GtkWidget     *widget,
                  GtkSpinButton *spin )
{
  gtk_spin_button_set_snap_to_ticks (spin, GTK_TOGGLE_BUTTON (widget)->active);
}

void toggle_numeric( GtkWidget *widget,
                     GtkSpinButton *spin )
{
  gtk_spin_button_set_numeric (spin, GTK_TOGGLE_BUTTON (widget)->active);
}

void change_digits( GtkWidget *widget,
                    GtkSpinButton *spin )
{
  gtk_spin_button_set_digits (GTK_SPIN_BUTTON (spinner1),
                              gtk_spin_button_get_value_as_int (spin));
}
```

```
void get_value( GtkWidget *widget,
                gpointer data )
{
  gchar buf[32];
  GtkLabel *label;
  GtkSpinButton *spin;

  spin = GTK_SPIN_BUTTON (spinner1);
  label = GTK_LABEL (g_object_get_data (G_OBJECT (widget), "user_data"));
  if (GPOINTER_TO_INT (data) == 1)
    sprintf (buf, "%d", gtk_spin_button_get_value_as_int (spin));
  else
    sprintf (buf, "%0.*f", spin->digits,
             gtk_spin_button_get_value (spin));
  gtk_label_set_text (label, buf);
}


int main( int   argc,
          char *argv[] )
{
  GtkWidget *window;
  GtkWidget *frame;
  GtkWidget *hbox;
  GtkWidget *main_vbox;
  GtkWidget *vbox;
  GtkWidget *vbox2;
  GtkWidget *spinner2;
  GtkWidget *spinner;
  GtkWidget *button;
  GtkWidget *label;
  GtkWidget *val_label;
  GtkAdjustment *adj;

  /* 初始化 */
  gtk_init (&argc, &argv);

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

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

  gtk_window_set_title (GTK_WINDOW (window), "Spin Button");

  main_vbox = gtk_vbox_new (FALSE, 5);
  gtk_container_set_border_width (GTK_CONTAINER (main_vbox), 10);
  gtk_container_add (GTK_CONTAINER (window), main_vbox);

  frame = gtk_frame_new ("Not accelerated");
  gtk_box_pack_start (GTK_BOX (main_vbox), frame, TRUE, TRUE, 0);

  vbox = gtk_vbox_new (FALSE, 0);
  gtk_container_set_border_width (GTK_CONTAINER (vbox), 5);
  gtk_container_add (GTK_CONTAINER (frame), vbox);

  /* 日、月、年微调器 */

  hbox = gtk_hbox_new (FALSE, 0);
  gtk_box_pack_start (GTK_BOX (vbox), hbox, TRUE, TRUE, 5);

  vbox2 = gtk_vbox_new (FALSE, 0);
  gtk_box_pack_start (GTK_BOX (hbox), vbox2, TRUE, TRUE, 5);

  label = gtk_label_new ("Day :");
  gtk_misc_set_alignment (GTK_MISC (label), 0, 0.5);
  gtk_box_pack_start (GTK_BOX (vbox2), label, FALSE, TRUE, 0);
```

```
adj = (GtkAdjustment *) gtk_adjustment_new (1.0, 1.0, 31.0, 1.0,
                                            5.0, 0.0);
spinner = gtk_spin_button_new (adj, 0, 0);
gtk_spin_button_set_wrap (GTK_SPIN_BUTTON (spinner), TRUE);
gtk_box_pack_start (GTK_BOX (vbox2), spinner, FALSE, TRUE, 0);

vbox2 = gtk_vbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (hbox), vbox2, TRUE, TRUE, 5);

label = gtk_label_new ("Month :");
gtk_misc_set_alignment (GTK_MISC (label), 0, 0.5);
gtk_box_pack_start (GTK_BOX (vbox2), label, FALSE, TRUE, 0);

adj = (GtkAdjustment *) gtk_adjustment_new (1.0, 1.0, 12.0, 1.0,
                                            5.0, 0.0);
spinner = gtk_spin_button_new (adj, 0, 0);
gtk_spin_button_set_wrap (GTK_SPIN_BUTTON (spinner), TRUE);
gtk_box_pack_start (GTK_BOX (vbox2), spinner, FALSE, TRUE, 0);

vbox2 = gtk_vbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (hbox), vbox2, TRUE, TRUE, 5);

label = gtk_label_new ("Year :");
gtk_misc_set_alignment (GTK_MISC (label), 0, 0.5);
gtk_box_pack_start (GTK_BOX (vbox2), label, FALSE, TRUE, 0);

adj = (GtkAdjustment *) gtk_adjustment_new (1998.0, 0.0, 2100.0,
                                            1.0, 100.0, 0.0);
spinner = gtk_spin_button_new (adj, 0, 0);
gtk_spin_button_set_wrap (GTK_SPIN_BUTTON (spinner), FALSE);
gtk_widget_set_size_request (spinner, 55, -1);
gtk_box_pack_start (GTK_BOX (vbox2), spinner, FALSE, TRUE, 0);

frame = gtk_frame_new ("Accelerated");
gtk_box_pack_start (GTK_BOX (main_vbox), frame, TRUE, TRUE, 0);

vbox = gtk_vbox_new (FALSE, 0);
gtk_container_set_border_width (GTK_CONTAINER (vbox), 5);
gtk_container_add (GTK_CONTAINER (frame), vbox);

hbox = gtk_hbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (vbox), hbox, FALSE, TRUE, 5);

vbox2 = gtk_vbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (hbox), vbox2, TRUE, TRUE, 5);

label = gtk_label_new ("Value :");
gtk_misc_set_alignment (GTK_MISC (label), 0, 0.5);
gtk_box_pack_start (GTK_BOX (vbox2), label, FALSE, TRUE, 0);

adj = (GtkAdjustment *) gtk_adjustment_new (0.0, -10000.0, 10000.0,
                                            0.5, 100.0, 0.0);
spinner1 = gtk_spin_button_new (adj, 1.0, 2);
gtk_spin_button_set_wrap (GTK_SPIN_BUTTON (spinner1), TRUE);
gtk_widget_set_size_request (spinner1, 100, -1);
gtk_box_pack_start (GTK_BOX (vbox2), spinner1, FALSE, TRUE, 0);

vbox2 = gtk_vbox_new (FALSE, 0);
gtk_box_pack_start (GTK_BOX (hbox), vbox2, TRUE, TRUE, 5);

label = gtk_label_new ("Digits :");
gtk_misc_set_alignment (GTK_MISC (label), 0, 0.5);
gtk_box_pack_start (GTK_BOX (vbox2), label, FALSE, TRUE, 0);

adj = (GtkAdjustment *) gtk_adjustment_new (2, 1, 5, 1, 1, 0);
spinner2 = gtk_spin_button_new (adj, 0.0, 0);
gtk_spin_button_set_wrap (GTK_SPIN_BUTTON (spinner2), TRUE);
g_signal_connect (G_OBJECT (adj), "value_changed",
```

```
                        G_CALLBACK (change_digits),
                        spinner2);
  gtk_box_pack_start (GTK_BOX (vbox2), spinner2, FALSE, TRUE, 0);

  hbox = gtk_hbox_new (FALSE, 0);
  gtk_box_pack_start (GTK_BOX (vbox), hbox, FALSE, TRUE, 5);

  button = gtk_check_button_new_with_label ("Snap to 0.5-ticks");
  g_signal_connect (G_OBJECT (button), "clicked",
                        G_CALLBACK (toggle_snap),
                        spinner1);
  gtk_box_pack_start (GTK_BOX (vbox), button, TRUE, TRUE, 0);
  gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (button), TRUE);

  button = gtk_check_button_new_with_label ("Numeric only input mode");
  g_signal_connect (G_OBJECT (button), "clicked",
                        G_CALLBACK (toggle_numeric),
                        spinner1);
  gtk_box_pack_start (GTK_BOX (vbox), button, TRUE, TRUE, 0);
  gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (button), TRUE);

  val_label = gtk_label_new ("");

  hbox = gtk_hbox_new (FALSE, 0);
  gtk_box_pack_start (GTK_BOX (vbox), hbox, FALSE, TRUE, 5);
  button = gtk_button_new_with_label ("Value as Int");
  g_object_set_data (G_OBJECT (button), "user_data", val_label);
  g_signal_connect (G_OBJECT (button), "clicked",
                        G_CALLBACK (get_value),
                        GINT_TO_POINTER (1));
  gtk_box_pack_start (GTK_BOX (hbox), button, TRUE, TRUE, 5);

  button = gtk_button_new_with_label ("Value as Float");
  g_object_set_data (G_OBJECT (button), "user_data", val_label);
  g_signal_connect (G_OBJECT (button), "clicked",
                        G_CALLBACK (get_value),
                        GINT_TO_POINTER (2));
  gtk_box_pack_start (GTK_BOX (hbox), button, TRUE, TRUE, 5);

  gtk_box_pack_start (GTK_BOX (vbox), val_label, TRUE, TRUE, 0);
  gtk_label_set_text (GTK_LABEL (val_label), "0");

  hbox = gtk_hbox_new (FALSE, 0);
  gtk_box_pack_start (GTK_BOX (main_vbox), hbox, FALSE, TRUE, 0);

  button = gtk_button_new_with_label ("Close");
  g_signal_connect_swapped (G_OBJECT (button), "clicked",
                            G_CALLBACK (gtk_widget_destroy),
                            window);
  gtk_box_pack_start (GTK_BOX (hbox), button, TRUE, TRUE, 5);

  gtk_widget_show_all (window);

  /* 进入事件循环 */
  gtk_main ();

  return 0;
}
```

组合框 Combo Box

组合框(combo box)是另一个很简单的构件，实际上它仅仅是其它构件的集合。从用户的观点来说，这个构件是由一个文本输入构件和一个下拉菜单组成的，用户可以从一个预先定义的列表里面选择一个选项，同时，用户也可以直接在文本框里面输入文本。

下面是从定义组合框构件的结构里面摘取出来的，从中可以看到组合框构件是由什么构件组合形成的：

```
struct _GtkCombo {
        GtkHBox hbox;
        GtkWidget *entry;
        GtkWidget *button;
        GtkWidget *popup;
        GtkWidget *popwin;
        GtkWidget *list;
        ...  };
```

可以看到，组合框构件有两个主要部分：一个输入框和一个列表。

用下面的函数创建组合框构件：

```
GtkWidget *gtk_combo_new( void );
```

现在，如果想设置显示在输入框部分中的字符串，可以直接操纵组合框构件内部的文本输入构件：

```
gtk_entry_set_text (GTK_ENTRY (GTK_COMBO (combo)->entry), "My String.");
```

要设置下拉列表中的值，可以使用下面的函数：

```
void gtk_combo_set_popdown_strings( GtkCombo *combo,
                                    GList    *strings );
```

在使用这个函数之前，先得将要添加的字符串组合成一个 GList 链表。GList 是一个双向链表，是 GLib 的一部分，而 GLib 是 GTK 的基础。暂时你可以先设置一个 GList 指针，其值设为 NULL，然后用下面的函数将字符串追加到链表当中：

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

要注意的是：一定要将 GList 链表的初值设为 NULL，必须将 g_list_append() 函数返回的值赋给要操作的链表本身。

下面是一段典型的代码，用于创建一个选项列表：

```
GList *glist = NULL;

    glist = g_list_append (glist, "String 1");
    glist = g_list_append (glist, "String 2");
    glist = g_list_append (glist, "String 3");
    glist = g_list_append (glist, "String 4");

    gtk_combo_set_popdown_strings (GTK_COMBO (combo), glist);

    /* 现在可以释放 glist 了，组合框已经复制了一份 */
```

组合框将传给它的 glist 结构里的字符串复制了一份。因此，在恰当的情况下，你应该确认释放了链表所使用的内存。

到这里为止，你已经有了一个可以使用的组合框构件了。有几个行为是可以改变的。下面是相关的函数：

```
void gtk_combo_set_use_arrows( GtkCombo *combo,
                               gboolean  val );

void gtk_combo_set_use_arrows_always( GtkCombo *combo,
                                      gboolean  val );

void gtk_combo_set_case_sensitive( GtkCombo *combo,
                                   gboolean  val );
```

gtk_combo_set_use_arrows() 让用户用上/下方向键改变文本输入构件你的值。这并不会弹出列表框，只是用列表中的下一个列表项替换了文本输入框中的文本(向上则取上一个值，向下则取下 一个值)。这是通过搜索当前项在列表中的位置并选择前一项/下一项来实现的。通常，在一个输入框中方向键是用来改变焦点的(也可以用 TAB 键)。注意，如 果当前项是列表的最后一项，按向下的方向键会改变焦点的位置(这对当前项为列表的第一项时按向上方向键也适用)。

如果当前值并不在列表中，则 gtk_combo_set_use_arrows() 函数的功能会失效。

同 样地，gtk_combo_set_use_arrows_always()允许使用上/下方向键在下拉列表中选取列表项，但是它在列表项中循

环，也就是 当列表项位于第一个表项时按向上方向键，会跳到最后一个，当列表项位于最后一个表项时按向下方向键，会跳到第一个。这样可以完全禁止使用方向键改变焦点。

gtk_combo_set_case_sensitive() 函数 切换 GTK 是否以大小写敏感的方式搜索其中的列表项。这在组合框根据内部文本输入构件中的文本查找列表值时使用。可以将其设置为大小写敏感或不敏感。如果用户同时按 下 MOD-1 和“Tab”键，组合框构件还可以简单地补全当前输入。MOD-1 一般被 xmodmap 工具映射为“Alt”键。注意，一些窗口管理器也要使用这种组合键方式，这将覆盖 GTK 中这个组合键的使用。

注意，我们使用的是组合框构件，它能够为我们从一个下拉列表中选择一个选项。这一点是很直截了当的。大多数时候，你可能很关心怎样从其中的文本输入构件中获取数据。组合框构件内部的文本输入构件可以用 GTK_ENTRY (GTK_COMBO (combo)->entry)访问。一般你想要做的两件主要工作一个是连接到 activate 信号，当用户按回车键时就能够进行响应，另一个就是读出其中的文本。第一件工作可以用下面的方法实现：

```
g_signal_connect (G_OBJECT (GTK_COMBO (combo)->entry), "activate",
                  G_CALLBACK (my_callback_function), my_data);
```

可以使用下面的函数在任意时候取得文本输入构件中的文本：

```
gchar *gtk_entry_get_text( GtkEntry *entry );
```

具体做法如下：

```
    gchar *string;

    string = gtk_entry_get_text (GTK_ENTRY (GTK_COMBO (combo)->entry));
```

这就是取得文本输入框中字符串的方法。下面有一个函数

```
void gtk_combo_disable_activate( GtkCombo *combo );
```

它将屏蔽组合框构件内部的文本输入构件的 activate 信号。我不明白你为什么想要用它，不过还是提供了。

日历 **Calendar**

日历(Calendar)构件是显示和获取每月日期等信息的高效方法。它是一个很容易创建和使用的构件。

创建日历构件的方法和其它构件的类似：

```
GtkWidget *gtk_calendar_new( void );
```

有时候，需要同时对构件的外观和内容做很多的修改。这时候可能会引起构件的多次更新，导致屏幕闪烁。可以在修改之前使用一个函数将构件“冻结”，然后在修改完成之后再用一个函数将构件“解冻”。这样，构件在整个过程中只做一次更新。

```
void gtk_calendar_freeze( GtkCalendar *Calendar );

void gtk_calendar_thaw( GtkCalendar *Calendar );
```

这两个函数和其它构件的冻结/解冻(freeze/thaw)函数作用完全一样。

日历构件有几个选项，可以用来改变构件的外观和操作方式。使用下面的函数可以改变这些选项：

```
void gtk_calendar_display_options( GtkCalendar              *calendar,
                                   GtkCalendarDisplayOptions  flags );
```

函数中的 flags 参数可以将下面的五种选项中的一个或者多个用逻辑位或(|)操作符组合起来：

GTK_CALENDAR_SHOW_HEADING

这个选项指定在绘制日历构件时，应该显示月份和年份。

GTK_CALENDAR_SHOW_DAY_NAMES

这个选项指定用三个字母的缩写显示每一天是星期几(比如 Mon、Tue 等)。

GTK_CALENDAR_NO_MONTH_CHANGE

这个选项指定用户不应该也不能够改变显示的月份。如果只想显示某个特定的月份，则可以使用这个选项。比如，在窗口上同时为一年的 12 个月分别设置一个日历构件时。

GTK_CALENDAR_SHOW_WEEK_NUMBERS

这个选项指定应该在日历的左边显示每一周在全年的周序号(例如；1 月 1 日是第 1 周，12 月 31 日是第 52 周)。

GTK_CALENDAR_WEEK_START_MONDAY

这个选项指定在日历构件中每一周是从星期一开始而不是从星期天开始。缺省设置是从星期天开始。此选项只影响日期在构件中从左到右的排列顺序。

下面的函数用于设置当前要显示的日期：

```
gint gtk_calendar_select_month( GtkCalendar *calendar,
                                guint        month,
                                guint        year );

void gtk_calendar_select_day( GtkCalendar *calendar,
                              guint        day );
```

gtk_calendar_select_month()的返回值是一个布尔值，指示设置是否成功。

使用 gtk_calendar_select_day()函数，如果指定的日期是合法的，会在日历构件中选中该日期。如果 day 参数的值是 0，将清除当前的选择。

除了可以选中一个日期以外，在一个月中可以有任意个日期被“标记”。被“标记”的日期会在日历构件中高亮显示。下面的函数用于标记日期和取消标记：

```
gint gtk_calendar_mark_day( GtkCalendar *calendar,
                            guint        day);

gint gtk_calendar_unmark_day( GtkCalendar *calendar,
                              guint        day);

void gtk_calendar_clear_marks( GtkCalendar *calendar);
```

当前标记的日期存储在一个 GtkCalendar 结构的数组中。数组的长度是 31，这样，要想知道某个特定的日期是否被标记，可以访问数值中相应的元素(注意，在 C 语言中，数值是从 0 开始编号的)。例如：

```
GtkCalendar *calendar;
calendar = gtk_calendar_new ();

...

/* 当月 7 日被标记了吗？ */
if (calendar->marked_date[7-1])
    /* 若执行此处的代码，表明 7 日已经被标记 */
```

注意，在月份和年份变化时，被标记的日期是不会变化的。

日历构件的最后一个函数用于取得当前选中的日/月/年值：

```
void gtk_calendar_get_date( GtkCalendar *calendar,
                            guint       *year,
                            guint       *month,
                            guint       *day );
```

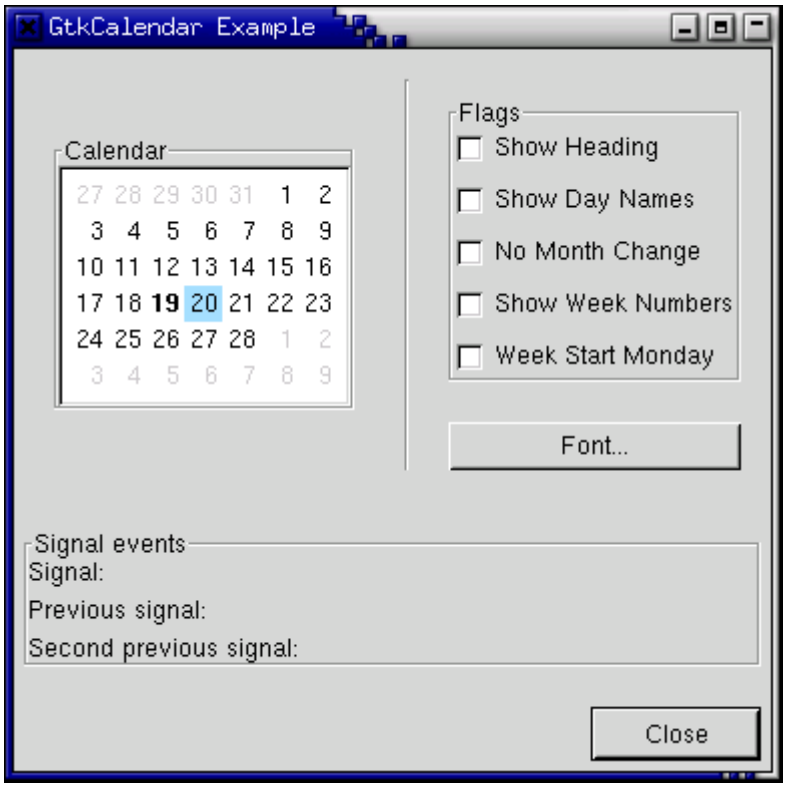
使用这个函数时，需要先声明几个 guint 类型的变量，再把变量地址传递给函数。所需要的返回值就存放在这几个变量中。

如果将某一个参数设置为 NULL，则不返回该值。

日历构件能够引发许多信号，用于指示日期被选中以及选择发生的变化。信号的意义很容易理解。信号名称如下：

- month_changed
- day_selected
- day_selected_double_click
- prev_month
- next_month
- prev_year
- next_year

下面是一个日历构件的示例，运用了上面介绍的各项特性。




```
/* GTK - GIMP 工具包
 * 版权 (C) 1995-1997 Peter Mattis, Spencer Kimball 和 Josh MacDonald 所有
 *
 * 本程序是自由软件。你可以在自由软件基金发布的 GNU GPL 的条款下重新分发
 * 或修改它。GPL 可以使用版本 2 或(由你选择)任何随后的版本。
 *
 * 本程序分发的目的是它可能对其他人有用，但不提供任何的担保，包括隐含的
 * 和适合特定用途的保证。请查阅 GNU 通用公共许可证获得详细的信息。
 *
 * 你应该已经随该软件一起收到一份 GNU 通用公共许可。如果还没有，请写信给
 * Free Software Foundation, Inc., 59 Temple Place - Suite 330,
 * Boston, MA 02111-1307, USA.
 */

#include <gtk/gtk.h>
#include <stdio.h>
#include <string.h>
#include <time.h>

#define DEF_PAD 10
```

```
#define DEF_PAD_SMALL 5

#define TM_YEAR_BASE 1900

typedef struct _CalendarData {
   GtkWidget *flag_checkboxes[5];
   gboolean  settings[5];
   gchar     *font;
   GtkWidget *font_dialog;
   GtkWidget *window;
   GtkWidget *prev2_sig;
   GtkWidget *prev_sig;
   GtkWidget *last_sig;
   GtkWidget *month;
} CalendarData;

enum {
   calendar_show_header,
   calendar_show_days,
   calendar_month_change,
   calendar_show_week,
   calendar_monday_first
};

/*
 * GtkCalendar 日历构件
 */

void calendar_date_to_string( CalendarData *data,
                              char         *buffer,
                              gint         buff_len )
{
   struct tm tm;
   time_t time;

   memset (&tm, 0, sizeof (tm));
   gtk_calendar_get_date (GTK_CALENDAR (data->window),
                          &tm.tm_year, &tm.tm_mon, &tm.tm_mday);
   tm.tm_year -= TM_YEAR_BASE;
   time = mktime (&tm);
   strftime (buffer, buff_len-1, "%x", gmtime (&time));
}

void calendar_set_signal_strings( char         *sig_str,
                                  CalendarData *data)
{
   const gchar *prev_sig;

   prev_sig = gtk_label_get_text (GTK_LABEL (data->prev_sig));
   gtk_label_set_text (GTK_LABEL (data->prev2_sig), prev_sig);

   prev_sig = gtk_label_get_text (GTK_LABEL (data->last_sig));
   gtk_label_set_text (GTK_LABEL (data->prev_sig), prev_sig);
   gtk_label_set_text (GTK_LABEL (data->last_sig), sig_str);
}

void calendar_month_changed( GtkWidget    *widget,
                             CalendarData *data )
{
   char buffer[256] = "month_changed: ";

   calendar_date_to_string (data, buffer+15, 256-15);
   calendar_set_signal_strings (buffer, data);
}

void calendar_day_selected( GtkWidget    *widget,
                            CalendarData *data )
{
```

```
  char buffer[256] = "day_selected: ";

  calendar_date_to_string (data, buffer+14, 256-14);
  calendar_set_signal_strings (buffer, data);
}

void calendar_day_selected_double_click( GtkWidget    *widget,
                                         CalendarData *data )
{
  struct tm tm;
  char buffer[256] = "day_selected_double_click: ";

  calendar_date_to_string (data, buffer+27, 256-27);
  calendar_set_signal_strings (buffer, data);

  memset (&tm, 0, sizeof (tm));
  gtk_calendar_get_date (GTK_CALENDAR (data->window),
                         &tm.tm_year, &tm.tm_mon, &tm.tm_mday);
  tm.tm_year -= TM_YEAR_BASE;

  if (GTK_CALENDAR (data->window)->marked_date[tm.tm_mday-1] == 0)
    {
      gtk_calendar_mark_day (GTK_CALENDAR (data->window), tm.tm_mday);
    }
  else
    {
      gtk_calendar_unmark_day (GTK_CALENDAR (data->window), tm.tm_mday);
    }
}

void calendar_prev_month( GtkWidget    *widget,
                          CalendarData *data )
{
  char buffer[256] = "prev_month: ";

  calendar_date_to_string (data, buffer+12, 256-12);
  calendar_set_signal_strings (buffer, data);
}

void calendar_next_month( GtkWidget    *widget,
                          CalendarData *data )
{
  char buffer[256] = "next_month: ";

  calendar_date_to_string (data, buffer+12, 256-12);
  calendar_set_signal_strings (buffer, data);
}

void calendar_prev_year( GtkWidget    *widget,
                         CalendarData *data )
{
  char buffer[256] = "prev_year: ";

  calendar_date_to_string (data, buffer+11, 256-11);
  calendar_set_signal_strings (buffer, data);
}

void calendar_next_year( GtkWidget    *widget,
                         CalendarData *data )
{
  char buffer[256] = "next_year: ";

  calendar_date_to_string (data, buffer+11, 256-11);
  calendar_set_signal_strings (buffer, data);
}


void calendar_set_flags( CalendarData *calendar )
{
```

```
  gint i;
  gint options = 0;
  for (i = 0; i < 5; i++)
    if (calendar->settings[i])
      {
        options=options + (1<<i);
      }
  if (calendar->window)
    gtk_calendar_display_options (GTK_CALENDAR (calendar->window), options);
}

void calendar_toggle_flag( GtkWidget    *toggle,
                           CalendarData *calendar )
{
  gint i;
  gint j;
  j = 0;
  for (i = 0; i < 5; i++)
    if (calendar->flag_checkboxes[i] == toggle)
      j = i;

  calendar->settings[j] = !calendar->settings[j];
  calendar_set_flags (calendar);

}

void calendar_font_selection_ok( GtkWidget    *button,
                                 CalendarData *calendar )
{
  GtkStyle *style;
  PangoFontDescription *font_desc;

  calendar->font = gtk_font_selection_dialog_get_font_name (
                       GTK_FONT_SELECTION_DIALOG (calendar->font_dialog));
  if (calendar->window)
    {
      font_desc = pango_font_description_from_string (calendar->font);
      if (font_desc)
        {
          style = gtk_style_copy (gtk_widget_get_style (calendar->window));
          style->font_desc = font_desc;
          gtk_widget_set_style (calendar->window, style);
        }
    }
}

void calendar_select_font( GtkWidget    *button,
                           CalendarData *calendar )
{
  GtkWidget *window;

  if (!calendar->font_dialog) {
    window = gtk_font_selection_dialog_new ("Font Selection Dialog");
    g_return_if_fail (GTK_IS_FONT_SELECTION_DIALOG (window));
    calendar->font_dialog = window;

    gtk_window_set_position (GTK_WINDOW (window), GTK_WIN_POS_MOUSE);

    g_signal_connect (G_OBJECT (window), "destroy",
                      G_CALLBACK (gtk_widget_destroyed),
                      &calendar->font_dialog);

    g_signal_connect (G_OBJECT (GTK_FONT_SELECTION_DIALOG (window)->ok_button),
                      "clicked", G_CALLBACK (calendar_font_selection_ok),
                      calendar);
    g_signal_connect_swapped (G_OBJECT (GTK_FONT_SELECTION_DIALOG (window)->cancel_button),
                              "clicked",
                              G_CALLBACK (gtk_widget_destroy),
                              calendar->font_dialog);
```

```
  }
  window=calendar->font_dialog;
  if (!GTK_WIDGET_VISIBLE (window))
    gtk_widget_show (window);
  else
    gtk_widget_destroy (window);

}

void create_calendar()
{
  GtkWidget *window;
  GtkWidget *vbox, *vbox2, *vbox3;
  GtkWidget *hbox;
  GtkWidget *hbbox;
  GtkWidget *calendar;
  GtkWidget *toggle;
  GtkWidget *button;
  GtkWidget *frame;
  GtkWidget *separator;
  GtkWidget *label;
  GtkWidget *bbox;
  static CalendarData calendar_data;
  gint i;

  struct {
    char *label;
  } flags[] =
    {
      { "Show Heading" },
      { "Show Day Names" },
      { "No Month Change" },
      { "Show Week Numbers" },
      { "Week Start Monday" }
    };


  calendar_data.window = NULL;
  calendar_data.font = NULL;
  calendar_data.font_dialog = NULL;

  for (i = 0; i < 5; i++) {
    calendar_data.settings[i] = 0;
  }

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "GtkCalendar Example");
  gtk_container_set_border_width (GTK_CONTAINER (window), 5);
  g_signal_connect (G_OBJECT (window), "destroy",
                    G_CALLBACK (gtk_main_quit),
                    NULL);
  g_signal_connect (G_OBJECT (window), "delete-event",
                    G_CALLBACK (gtk_false),
                    NULL);

  gtk_window_set_resizable (GTK_WINDOW (window), FALSE);

  vbox = gtk_vbox_new (FALSE, DEF_PAD);
  gtk_container_add (GTK_CONTAINER (window), vbox);

  /*
   * 顶级窗口，其中包含日历构件，设置日历各参数的复选按钮和设置字体的按钮
   */

  hbox = gtk_hbox_new (FALSE, DEF_PAD);
  gtk_box_pack_start (GTK_BOX (vbox), hbox, TRUE, TRUE, DEF_PAD);
  hbbox = gtk_hbutton_box_new ();
  gtk_box_pack_start (GTK_BOX (hbox), hbbox, FALSE, FALSE, DEF_PAD);
```

```
gtk_button_box_set_layout (GTK_BUTTON_BOX(hbbox), GTK_BUTTONBOX_SPREAD);
gtk_box_set_spacing (GTK_BOX (hbbox), 5);

/* 日历构件 */
frame = gtk_frame_new ("Calendar");
gtk_box_pack_start (GTK_BOX (hbbox), frame, FALSE, TRUE, DEF_PAD);
calendar=gtk_calendar_new ();
calendar_data.window = calendar;
calendar_set_flags (&calendar_data);
gtk_calendar_mark_day (GTK_CALENDAR (calendar), 19);
gtk_container_add( GTK_CONTAINER (frame), calendar);
g_signal_connect (G_OBJECT (calendar), "month_changed",
                  G_CALLBACK (calendar_month_changed),
                  &calendar_data);
g_signal_connect (G_OBJECT (calendar), "day_selected",
                  G_CALLBACK (calendar_day_selected),
                  &calendar_data);
g_signal_connect (G_OBJECT (calendar), "day_selected_double_click",
                  G_CALLBACK (calendar_day_selected_double_click),
                  &calendar_data);
g_signal_connect (G_OBJECT (calendar), "prev_month",
                  G_CALLBACK (calendar_prev_month),
                  &calendar_data);
g_signal_connect (G_OBJECT (calendar), "next_month",
                  G_CALLBACK (calendar_next_month),
                  &calendar_data);
g_signal_connect (G_OBJECT (calendar), "prev_year",
                  G_CALLBACK (calendar_prev_year),
                  &calendar_data);
g_signal_connect (G_OBJECT (calendar), "next_year",
                  G_CALLBACK (calendar_next_year),
                  &calendar_data);


separator = gtk_vseparator_new ();
gtk_box_pack_start (GTK_BOX (hbox), separator, FALSE, TRUE, 0);

vbox2 = gtk_vbox_new (FALSE, DEF_PAD);
gtk_box_pack_start (GTK_BOX (hbox), vbox2, FALSE, FALSE, DEF_PAD);

/* 创建一个框架，放入设置各种参数的复选按钮 */

frame = gtk_frame_new ("Flags");
gtk_box_pack_start (GTK_BOX (vbox2), frame, TRUE, TRUE, DEF_PAD);
vbox3 = gtk_vbox_new (TRUE, DEF_PAD_SMALL);
gtk_container_add (GTK_CONTAINER (frame), vbox3);

for (i = 0; i < 5; i++)
  {
    toggle = gtk_check_button_new_with_label (flags[i].label);
    g_signal_connect (G_OBJECT (toggle),
                      "toggled",
                      G_CALLBACK (calendar_toggle_flag),
                      &calendar_data);
    gtk_box_pack_start (GTK_BOX (vbox3), toggle, TRUE, TRUE, 0);
    calendar_data.flag_checkboxes[i] = toggle;
  }
/* 创建一个按钮，用于设置字体 */
button = gtk_button_new_with_label ("Font...");
g_signal_connect (G_OBJECT (button),
                  "clicked",
                  G_CALLBACK (calendar_select_font),
                  &calendar_data);
gtk_box_pack_start (GTK_BOX (vbox2), button, FALSE, FALSE, 0);

/*
 * 创建“信号-事件”部分
 */
```

```
  frame = gtk_frame_new ("Signal events");
  gtk_box_pack_start (GTK_BOX (vbox), frame, TRUE, TRUE, DEF_PAD);

  vbox2 = gtk_vbox_new (TRUE, DEF_PAD_SMALL);
  gtk_container_add (GTK_CONTAINER (frame), vbox2);

  hbox = gtk_hbox_new (FALSE, 3);
  gtk_box_pack_start (GTK_BOX (vbox2), hbox, FALSE, TRUE, 0);
  label = gtk_label_new ("Signal:");
  gtk_box_pack_start (GTK_BOX (hbox), label, FALSE, TRUE, 0);
  calendar_data.last_sig = gtk_label_new ("");
  gtk_box_pack_start (GTK_BOX (hbox), calendar_data.last_sig, FALSE, TRUE, 0);

  hbox = gtk_hbox_new (FALSE, 3);
  gtk_box_pack_start (GTK_BOX (vbox2), hbox, FALSE, TRUE, 0);
  label = gtk_label_new ("Previous signal:");
  gtk_box_pack_start (GTK_BOX (hbox), label, FALSE, TRUE, 0);
  calendar_data.prev_sig = gtk_label_new ("");
  gtk_box_pack_start (GTK_BOX (hbox), calendar_data.prev_sig, FALSE, TRUE, 0);

  hbox = gtk_hbox_new (FALSE, 3);
  gtk_box_pack_start (GTK_BOX (vbox2), hbox, FALSE, TRUE, 0);
  label = gtk_label_new ("Second previous signal:");
  gtk_box_pack_start (GTK_BOX (hbox), label, FALSE, TRUE, 0);
  calendar_data.prev2_sig = gtk_label_new ("");
  gtk_box_pack_start (GTK_BOX (hbox), calendar_data.prev2_sig, FALSE, TRUE, 0);

  bbox = gtk_hbutton_box_new ();
  gtk_box_pack_start (GTK_BOX (vbox), bbox, FALSE, FALSE, 0);
  gtk_button_box_set_layout (GTK_BUTTON_BOX (bbox), GTK_BUTTONBOX_END);

  button = gtk_button_new_with_label ("Close");
  g_signal_connect (G_OBJECT (button), "clicked",
                    G_CALLBACK (gtk_main_quit),
                    NULL);
  gtk_container_add (GTK_CONTAINER (bbox), button);
  GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);
  gtk_widget_grab_default (button);

  gtk_widget_show_all (window);
}


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

  create_calendar ();

  gtk_main ();

  return 0;
}
```

颜色选择 Color Selection

颜 色选择(color selection)构件是一个用来交互式地选择颜色的构件。这个组合构件让用户通过操纵 RGB 值(红绿蓝)和 HSV 值(色度、饱和度、纯度)来选择颜 色。这是通过调整滑动条(sliders)的值或者文本输入构件的值，或者从一个色度/饱和度/纯度条上选择相应的颜色来实现的。你还可以通过它来设置颜 色的透明性。

目前，颜色选择构件只能引发一种信号： color_changed。它是在构件你的颜色值发生变化时，或者通过 gtk_color_selection_set_color() 函数显式设置构件的颜色值时引发。

现在可以看一下颜色选择构件能够为我们提供一些什么。这个构件有两种风格：GtkColorSelection 和 GtkColorSelectionDialog。

```
GtkWidget *gtk_color_selection_new( void );
```

你将很少直接使用这个函数。它创建一个孤立的颜色选择构件，并需要将其放在某个窗口上。颜色选择构件是从 VBox 构件派生的。

```
GtkWidget *gtk_color_selection_dialog_new( const gchar *title );
```

这 是最常用的颜色选择构件的构建函数，它创建一个颜色选择对话框。它内部有一个框架构件，框架构件中包含了一个颜色选择构件、一个垂直分隔线构件、一个包含 了 Ok、Cancel、Help 三个按钮的横向盒。你可以通过访问颜色选择对话框构件结构中的"ok_button", "cancel_button"和"help_button"构件来访问它们(例如：GTK_COLOR_SELECTION_DIALOG (colorseldialog)->ok_button))。

```
void gtk_color_selection_set_has_opacity_control( GtkColorSelection *colorsel,
                                                  gboolean           has_opacity );
```

颜色选择构件支持调整颜色的不透明性(一般也称为 alpha 通道)。缺省值是禁用这个特性。调用下面的函数，将 has_opacity 设置为 TRUE 启用该特性。同样，has_opacity 设置为 FALSE 时将禁用此特性。

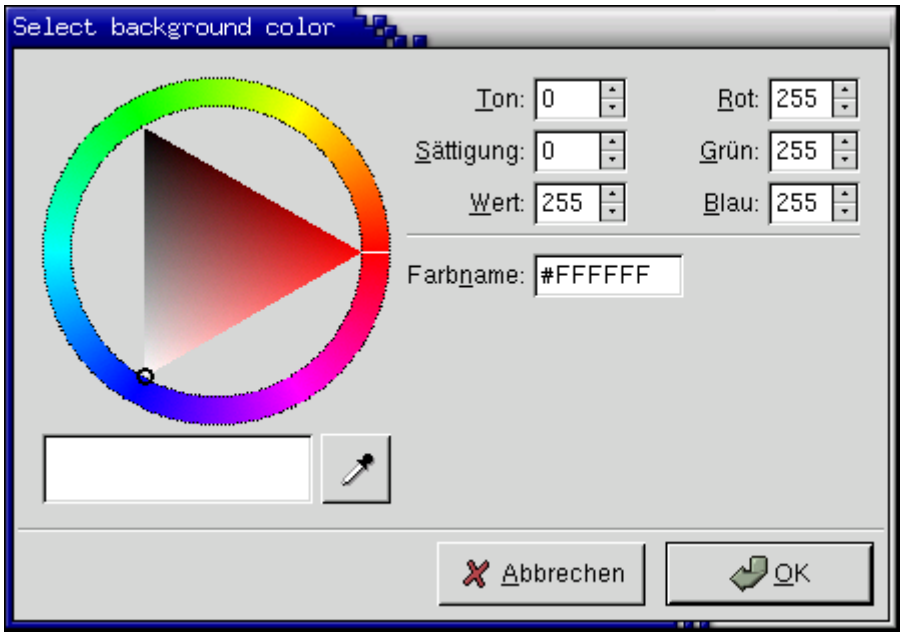
```
void gtk_color_selection_set_current_color( GtkColorSelection *colorsel,
                                            GdkColor          *color );

void gtk_color_selection_set_current_alpha( GtkColorSelection *colorsel,
                                            guint16           alpha );
```

可 以调用 gtk_color_selection_set_current_color() 函数显式地设置颜色选择构件的当前颜色，其中的 color 参数是一个指向 GdkColor 的指针。 gtk_color_selection_set_current_alpha()用来设置不透明度(alpha 通道)。其中的 alpha 值应该在 0(完 全透明)和 65636(完全不透明)之间。

```
void gtk_color_selection_get_current_color( GtkColorSelection *colorsel,
                                            GdkColor *color );

void gtk_color_selection_get_current_alpha( GtkColorSelection *colorsel,
                                            guint16           *alpha );
```

当需要查询当前颜色值时，典型情况是接收到一个 "color_changed" 信号时，使用这些函数。

下面是一个简单的示例，它演示了如何使用颜色选择对话框构件。这个程序显示了一个包含绘图区的窗口。点击它会打开一个颜色选择对话框，改变颜色选择对话框中的颜色，会改变绘图区的背景色。



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

GtkWidget *colorseldlg = NULL;
GtkWidget *drawingarea = NULL;
GdkColor color;

/* 颜色改变信号的处理函数 */

void color_changed_cb( GtkWidget        *widget,
                       GtkColorSelection *colorsel )
{
  GdkColor ncolor;

  gtk_color_selection_get_current_color (colorsel, &ncolor);
  gtk_widget_modify_bg (drawingarea, GTK_STATE_NORMAL, &ncolor);
}

/* 绘图区事件处理函数 */

gint area_event( GtkWidget *widget,
                 GdkEvent  *event,
                 gpointer   client_data )
{
  gint handled = FALSE;
  gint response;
  GtkColorSelection *colorsel;

  /* 检查是否接收到一个鼠标按键按下事件 */

  if (event->type == GDK_BUTTON_PRESS)
    {
      handled = TRUE;

       /* 创建颜色选择对话框 */
      if (colorseldlg == NULL)
        colorseldlg = gtk_color_selection_dialog_new ("Select background color");

      /* 获取颜色选择构件 */
      colorsel = GTK_COLOR_SELECTION (GTK_COLOR_SELECTION_DIALOG (colorseldlg)->colorsel);

      gtk_color_selection_set_previous_color (colorsel, &color);
      gtk_color_selection_set_current_color (colorsel, &color);
      gtk_color_selection_set_has_palette (colorsel, TRUE);

      /* 为 "color_changed" 信号设置回调函数，将用户数据设置为
       * 颜色选择构件 */
      g_signal_connect (G_OBJECT (colorsel), "color_changed",
                        G_CALLBACK (color_changed_cb), (gpointer)colorsel);

      /* 显示对话框 */
      response = gtk_dialog_run (GTK_DIALOG (colorseldlg));

      if (response == GTK_RESPONSE_OK)
        gtk_color_selection_get_current_color (colorsel, &color);
      else
        gtk_widget_modify_bg (drawingarea, GTK_STATE_NORMAL, &color);

      gtk_widget_hide (colorseldlg);
    }

  return handled;
}

/* 关闭、退出的事件处理函数 */
```

```
gint destroy_window( GtkWidget *widget,
                     GdkEvent  *event,
                     gpointer   client_data )
{
  gtk_main_quit ();
  return TRUE;
}

/* 主函数 */

gint main( gint   argc,
           gchar *argv[] )
{
  GtkWidget *window;

  /* 初始化，处理并删去跟 gtk 有关的命令参数 */

  gtk_init (&argc, &argv);

  /* 创建顶级窗口，设置标题，以及窗口是否可缩放 */

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Color selection test");
  gtk_window_set_policy (GTK_WINDOW (window), TRUE, TRUE, TRUE);

  /* 为 "delete" 和 "destroy" 事件设置回调函数以便退出 */

  g_signal_connect (GTK_OBJECT (window), "delete_event",
                    GTK_SIGNAL_FUNC (destroy_window), (gpointer)window);

  /* 创建绘图区，设置尺寸，捕获鼠标按键事件 */

  drawingarea = gtk_drawing_area_new ();

  color.red = 0;
  color.blue = 65535;
  color.green = 0;
  gtk_widget_modify_bg (drawingarea, GTK_STATE_NORMAL, &color);

  gtk_widget_set_size_request (GTK_WIDGET (drawingarea), 200, 200);

  gtk_widget_set_events (drawingarea, GDK_BUTTON_PRESS_MASK);

  g_signal_connect (GTK_OBJECT (drawingarea), "event",
                    GTK_SIGNAL_FUNC (area_event), (gpointer)drawingarea);

  /* 将绘图区添加到窗口中，然后显示它们 */

  gtk_container_add (GTK_CONTAINER (window), drawingarea);

  gtk_widget_show (drawingarea);
  gtk_widget_show (window);

  /* 进入 gtk 主循环(这个函数从不会返回) */

  gtk_main ();

  /* 满足性情暴躁的编译器 */

  return 0;
}
```

文件选择 File Selections

文件选择(file selection)构件是一种快速、简单的显示文件对话框的方法。它带有“Ok”、“Cancel”、“Help”按钮，可以极大地减少编程时间。

可以用下面的方法创建文件选择构件：

```
GtkWidget *gtk_file_selection_new( const gchar *title );
```

要设置文件名，例如，要在打开时指向指定目录，或者给定一个缺省文件名，可以使用下面的函数：

```
void gtk_file_selection_set_filename( GtkFileSelection *filesel,
                                      const gchar      *filename );
```

要获取用户输入或点击选中的文本，可以使用下面的函数：

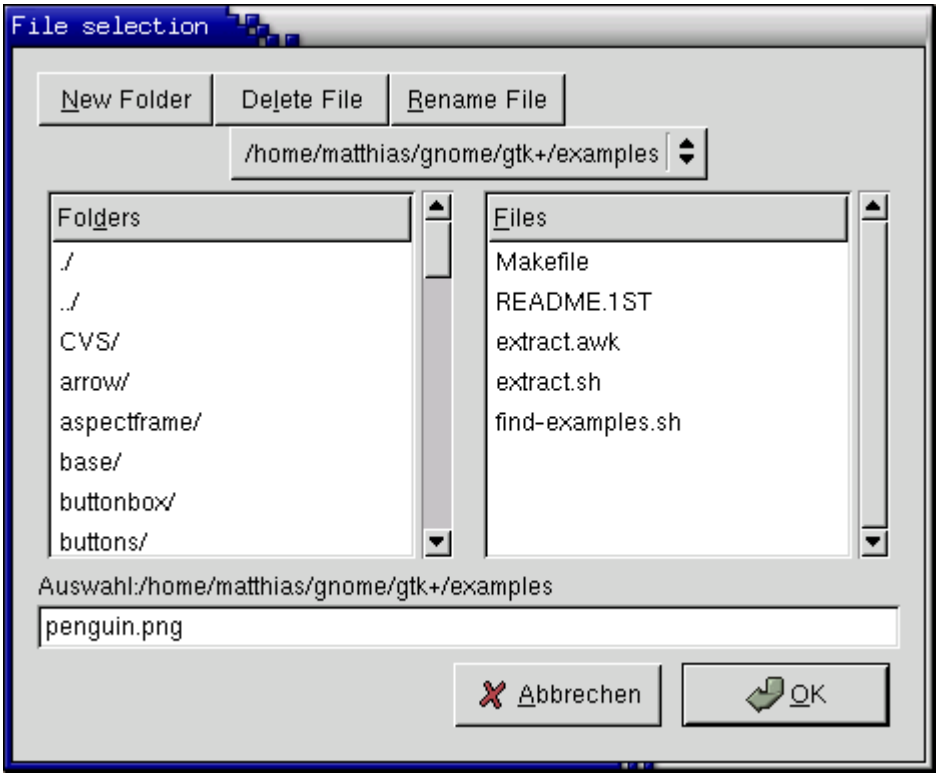
```
gchar *gtk_file_selection_get_filename( GtkFileSelection *filesel );
```

还有几个指向文件选择构件内部的构件的指针，它们是：

```
dir_list
file_list
selection_entry
selection_text
main_vbox
ok_button
cancel_button
help_button
```

在为文件选择构件的信号设置回调函数时，极有可能用到 ok_button、cancel_button 和 help_button 指针。

下面的示例是来自 testgtk.c 中的一段代码。修改了一下使它可以独立运行。你可以看到，创建一个文件选择构件并不费多少功夫。在这个示例中，Help 按钮出现在屏幕上，但是它什么也不做，因为没有为它的信号设置回调函数。



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

/* 获得文件名，并将它打印到控制台(console)上 */
void file_ok_sel( GtkWidget        *w,
                  GtkFileSelection *fs )
{
    g_print ("%s\n", gtk_file_selection_get_filename (GTK_FILE_SELECTION (fs)));
}

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

    gtk_init (&argc, &argv);

    /* 创建一个新的文件选择构件 */
    filew = gtk_file_selection_new ("File selection");

    g_signal_connect (G_OBJECT (filew), "destroy",
                      G_CALLBACK (gtk_main_quit), NULL);
    /* 为 ok_button 按钮设置回调函数，连接到 file_ok_sel function 函数 */
    g_signal_connect (G_OBJECT (GTK_FILE_SELECTION (filew)->ok_button),
                      "clicked",
                      G_CALLBACK (file_ok_sel), filew);

    /* 为 cancel_button 设置回调函数，销毁构件 */
    g_signal_connect_swapped (G_OBJECT (GTK_FILE_SELECTION (filew)->cancel_button),
                              "clicked",
                              G_CALLBACK (gtk_widget_destroy), filew);

    /* 设置文件名，比如这个一个文件保存对话框，我们给了一个缺省文件名 */
    gtk_file_selection_set_filename (GTK_FILE_SELECTION(filew),
                                     "penguin.png");

    gtk_widget_show (filew);
    gtk_main ();
    return 0;
}
```

容器构件 Container Widgets

事件盒 The EventBox

一 些 GTK 构件没有与之相关联的 X 窗口，所以它们只在其父构件上显示其外观。由于这个原因，它们不能接收任何事件，并且，如果它们尺寸设置不正确，它们也不会自动剪裁(译者注：裁剪就是使 构件只显示一部分)，这样可能会把界面弄得乱糟糟的。如果要想构件接收事件，可以使用事件盒(EventBox)。

初 一看，事件盒构件好像完全没有什么用。它在屏幕上什么也不画，并且对事件也不做响应。但是，它有一个功能：为它的子构件提供一个 X 窗口。因为许多 G T K 构件并没有相关联的 X 窗口，所以这一点很重要。虽然没有 X 窗口会节省内存，提高系统性能，但它也有一些弱点。没有 X 窗口的构件不能接收事件，并且对它的任何内容不能实施剪裁。虽然事件盒构件的名称*事件盒*强调了它的事件处理功能，它也能用于剪裁构件(更多的信息请看下面的示例)。

用以下函数创建一个新的事件盒构件：

```
GtkWidget *gtk_event_box_new( void );
```

然后子构件就可以添加到这个事件盒里面：

```
gtk_container_add (GTK_CONTAINER (event_box), child_widget);
```

下面的示例演示了事件盒的用途：创建一个标签，将它剪裁，放到一个小盒子里面，然后设置让鼠标点击时程序退出。改变窗口的尺寸会使标签构件的尺寸发生变化。

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

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

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    gtk_window_set_title (GTK_WINDOW (window), "Event Box");

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

    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* 创建一个事件盒，然后将它加到顶级窗口上 */

    event_box = gtk_event_box_new ();
    gtk_container_add (GTK_CONTAINER (window), event_box);
    gtk_widget_show (event_box);

    /* 创建一个长标签 */

    label = gtk_label_new ("Click here to quit, quit, quit, quit, quit");
    gtk_container_add (GTK_CONTAINER (event_box), label);
    gtk_widget_show (label);

    /* 将标签剪裁短。 */
    gtk_widget_set_size_request (label, 110, 20);

    /* 为它绑定一个动作 */
    gtk_widget_set_events (event_box, GDK_BUTTON_PRESS_MASK);
    g_signal_connect (G_OBJECT (event_box), "button_press_event",
                      G_CALLBACK (exit), NULL);

    /* 又是一个要有 X 窗口的东西... */
       //译者注:  要设置鼠标式样也要有 X 窗口。gtk_widget_realize 使构件在没有 show 之前就形成 X 窗口。

    gtk_widget_realize (event_box);
    gdk_window_set_cursor (event_box->window, gdk_cursor_new (GDK_HAND1));

    gtk_widget_show (window);

    gtk_main ();

    return 0;
}
```

对齐构件 The Alignment widget

对齐(alignment)构件允许将一个构件放在相对于对齐构件窗口的某个位置和尺寸上。例如，将一个构件放在窗口的正中间时，就要使用对齐构件。

只有如下两个函数与对齐构件相关：

```
GtkWidget* gtk_alignment_new( gfloat xalign,
                              gfloat yalign,
                              gfloat xscale,
                              gfloat yscale );

void gtk_alignment_set( GtkAlignment *alignment,
                        gfloat       xalign,
                        gfloat       yalign,
                        gfloat       xscale,
                        gfloat       yscale );
```

第一个函数用指定的参数创建新的对齐构件。第二个函数用于改变对齐构件的参数。

上面函数的所有四个参数都是介于 0.0 与 1.0 间的浮点数。xalign 和 yalign 参数影响放在对齐构件里的构件的位置。xscale 和 yscale 参数影响分配给构件的空间总数。

可以用下面的函数将子构件添加到对齐构件中：

```
gtk_container_add (GTK_CONTAINER (alignment), child_widget);
```

要看关于对齐构件的示例，可以参考进度条构件的示例。

固定容器 Fixed Container

固定容器(The Fixed container)允许将构件放在窗口的固定位置，这个位置是相对于固定容器的左上角的。构件的位置可以动态改变。

只有少数几个与固定容器构件相关的函数：

```
GtkWidget* gtk_fixed_new( void );

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

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

gtk_fixed_new() 函数用于创建新的固定容器。

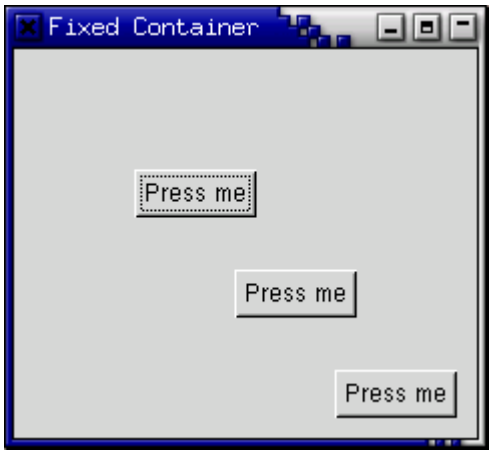
gtk_fixed_put() 函数将 widget 放在 fixed 的由 x 和 y 指定的位置。

gtk_fixed_move() 函数将指定构件移动到新位置。

```
void gtk_fixed_set_has_window( GtkFixed  *fixed,
                               gboolean   has_window );

gboolean gtk_fixed_get_has_window( GtkFixed *fixed );
```

通常，固定容器没有它们自己的 X 窗口。由于这点在早期版本的 GTK 中是不同的，gtk_fixed_set_has_window() 函数可以使创建的固定容器*有*它们自己的窗口。这个必须在构件实例化(realizing)之前调用。

下面的示例演示了怎样使用固定容器。




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

/* 我准备偷点懒，用一些全局变量储存固定容器里构件的位置。 */
gint x = 50;
gint y = 50;

/* 这个回调函数将按钮移动到固定容器的新位置。 */
void move_button( GtkWidget *widget,
                  GtkWidget *fixed )
{
  x = (x + 30) % 300;
  y = (y + 50) % 300;
  gtk_fixed_move (GTK_FIXED (fixed), widget, x, y);
}

int main( int   argc,
          char *argv[] )
{
  /* GtkWidget 是构件的存储类型 */
  GtkWidget *window;
  GtkWidget *fixed;
  GtkWidget *button;
  gint i;

  /* 初始化 */
  gtk_init (&argc, &argv);

  /* 创建一个新窗口 */
  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Fixed Container");

  /* 为窗口的 "destroy" 事件设置一个信号处理函数 */
  g_signal_connect (G_OBJECT (window), "destroy",
                    G_CALLBACK (gtk_main_quit), NULL);

  /* 设置窗口的边框宽度 */
  gtk_container_set_border_width (GTK_CONTAINER (window), 10);

  /* 创建一个固定容器 */
  fixed = gtk_fixed_new ();
  gtk_container_add (GTK_CONTAINER (window), fixed);
  gtk_widget_show (fixed);

  for (i = 1 ; i <= 3 ; i++) {
    /* 创建一个标签为"Press me"的新按钮 */
    button = gtk_button_new_with_label ("Press me");

    /* 当按钮接收到 "clicked" 信号时，调用 move_button() 函数，并把这个固定
     * 容器作为参数传给它 */
```

```
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (move_button), fixed);

    /* 将按钮组装到一个固定容器的窗口中 */
    gtk_fixed_put (GTK_FIXED (fixed), button, i*50, i*50);

    /* 最后一步是显示新建的构件 */
    gtk_widget_show (button);
  }

  /* 显示窗口 */
  gtk_widget_show (window);

  /* 进入事件循环 */
  gtk_main ();

  return 0;
}
```

布局容器 Layout Container

布 局容器(The Layout container)与固定容器(the Fixed container)类似，不过它可以在一个无限的滚动区域定位构件(其实也不能大于 2^32 像素)。在 X 系统中，窗口的宽度和高度只能限于在 32767 像素以你。布局容器构件使用一些特殊的技巧(doing some exotic stuff using window and bit gravities)越过这种限制。所以，即使在滚动区域你有很多子构件，也可以平滑地滚动。

用以下函数创建布局容器:

```
GtkWidget *gtk_layout_new( GtkAdjustment *hadjustment,
                           GtkAdjustment *vadjustment );
```

可以看到，你可以有选择地指定布局容器滚动时要使用的调整对象。

你可以用下面的两个函数在布局容器构件你添加和移动构件。

```
void gtk_layout_put( GtkLayout *layout,
                     GtkWidget *widget,
                     gint       x,
                     gint       y );

void gtk_layout_move( GtkLayout *layout,
                      GtkWidget *widget,
                      gint       x,
                      gint       y );
```

布局容器构件的尺寸可以用接下来的这个函数指定:

```
void gtk_layout_set_size( GtkLayout *layout,
                          guint      width,
                          guint      height );
```

最后 4 个函数用于操纵垂直和水平的调整对象。

```
GtkAdjustment* gtk_layout_get_hadjustment( GtkLayout *layout );

GtkAdjustment* gtk_layout_get_vadjustment( GtkLayout *layout );

void gtk_layout_set_hadjustment( GtkLayout     *layout,
                                 GtkAdjustment *adjustment );

void gtk_layout_set_vadjustment( GtkLayout     *layout,
                                 GtkAdjustment *adjustment);
```

框架 Frames

框架(Frames)可以用于在盒子中封装一个或一组构件，框架本身还可以有一个标签。标签的位置和盒子的风格可以灵活改变。

框架可以用下面的函数创建:

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

标签缺省放在框架的左上角。传递 NULL 值作为 label 参数时，框架不显示标签。标签文本可以用下面的函数改变。

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

标签的位置可以用下面的函数改变:

```
void gtk_frame_set_label_align( GtkFrame *frame,
                                gfloat    xalign,
                                gfloat    yalign );
```

xalign 和 yalign 参数取值范围介于 0.0 和 1.0 之间。xalign 指定标签在框架构件上部水平线上的位置。yalign 目前还没有被使用。xalign 的缺省值是 0.0，它将标签放在框架构件的最左端。

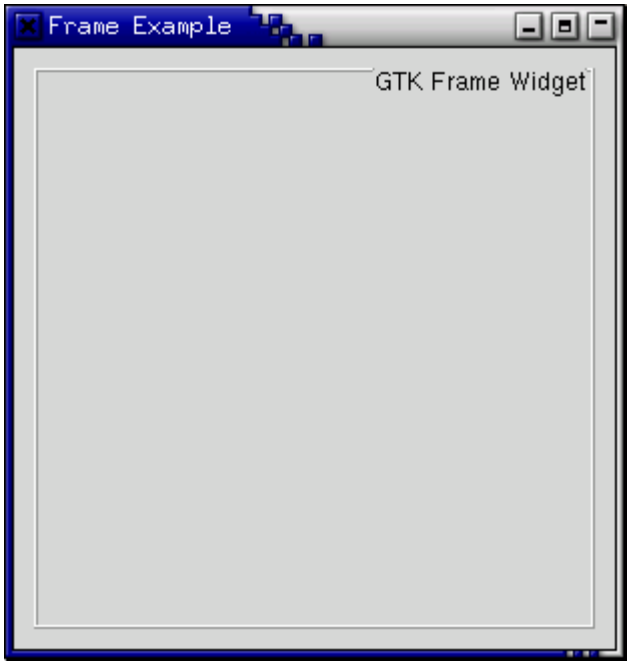
下面的函数改变盒子的风格，用于显示框架的轮廓。

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

type 参数可以取以下值之一:

GTK_SHADOW_NONE
GTK_SHADOW_IN
GTK_SHADOW_OUT
GTK_SHADOW_ETCHED_IN (缺省值)
GTK_SHADOW_ETCHED_OUT

下面的示例演示了怎样使用框架构件。



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

int main( int   argc,
          char *argv[] )
{
  /* GtkWidget 是构件的存储类型 */
  GtkWidget *window;
  GtkWidget *frame;

  /* 初始化 */
  gtk_init (&argc, &argv);

  /* 创建一个新窗口 */
  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Frame Example");

  /* 在这里我们将 "destroy" 事件连接到一个回调函数 */
  g_signal_connect (G_OBJECT (window), "destroy",
                    G_CALLBACK (gtk_main_quit), NULL);

  gtk_widget_set_size_request (window, 300, 300);
  /* 设置窗口的边框宽度 */
  gtk_container_set_border_width (GTK_CONTAINER (window), 10);

  /* 创建一个框架 */
  frame = gtk_frame_new (NULL);
  gtk_container_add (GTK_CONTAINER (window), frame);

  /* 设置框架的标签 */
  gtk_frame_set_label (GTK_FRAME (frame), "GTK Frame Widget");

  /* 将标签定位在框架的右边 */
  gtk_frame_set_label_align (GTK_FRAME (frame), 1.0, 0.0);

  /* 设置框架的风格 */
  gtk_frame_set_shadow_type (GTK_FRAME (frame), GTK_SHADOW_ETCHED_OUT);

  gtk_widget_show (frame);

  /* 显示窗口 */
  gtk_widget_show (window);

  /* 进入事件循环 */
  gtk_main ();

  return 0;
}
```

比例框架 Aspect Frames

比 例框架构件(The aspect frame widget)和框架构件(frame widget)差不多，除了它还会使子构件的外观比例（也就是宽和长的比例）保持一定值，如果需要，还会在构件中增加额外的可用空间。这很有用，例如，想 预览一个大的图片。当用户改变窗口的尺寸时，预览器的尺寸应该随之改变，但是外观比例要与原来图片的尺寸保持一致。

用下面的函数创建一个新的比例框架:

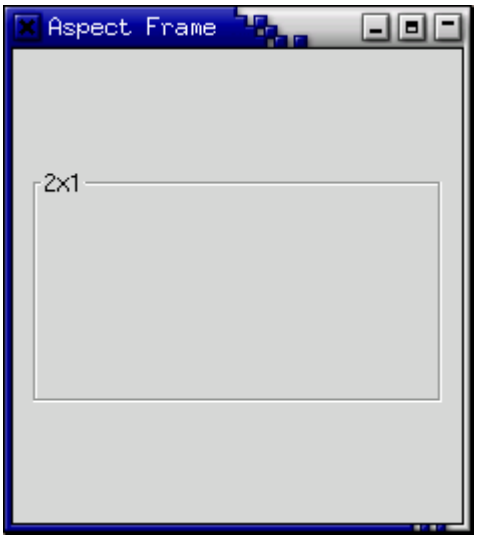
```
GtkWidget *gtk_aspect_frame_new( const gchar *label,
                                 gfloat       xalign,
                                 gfloat       yalign,
                                 gfloat       ratio,
                                 gboolean     obey_child);
```

xalign 和 yalign 参数的作用和创建对齐构件(Alignment widgets)时的一样。如果 obey_child 参数设置为 TRUE，子构件的长宽比例会和它所请求的理想长宽比例相匹配。否则，比例值由 ratio 参数指定。

用以下函数可以改变已有比例框架构件的选项：

```
void gtk_aspect_frame_set( GtkAspectFrame *aspect_frame,
                           gfloat         xalign,
                           gfloat         yalign,
                           gfloat         ratio,
                           gboolean       obey_child);
```

在下面的示例中，程序用一个比例框架构件显示一个绘图区，纵横比例总是 2:1，而不管用户如何改变顶级窗口的尺寸。




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

int main( int argc,
          char *argv[] )
{
    GtkWidget *window;
    GtkWidget *aspect_frame;
    GtkWidget *drawing_area;
    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Aspect Frame");
    g_signal_connect (G_OBJECT (window), "destroy",
                      G_CALLBACK (gtk_main_quit), NULL);
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* 创建一个比例框架，将它添加到顶级窗口中 */

    aspect_frame = gtk_aspect_frame_new ("2x1", /* label */
                                         0.5, /* center x */
                                         0.5, /* center y */
                                         2, /* xsize/ysize = 2 */
                                         FALSE /* ignore child's aspect */);

    gtk_container_add (GTK_CONTAINER (window), aspect_frame);
    gtk_widget_show (aspect_frame);

    /* 添加一个子构件到比例框架中 */

    drawing_area = gtk_drawing_area_new ();

    /* 要求一个 200×200 的窗口，但是比例框架会给出一个 200×100
     * 的窗口，因为我们已经指定了 2×1 的比例值 */
    gtk_widget_set_size_request (drawing_area, 200, 200);
    gtk_container_add (GTK_CONTAINER (aspect_frame), drawing_area);
    gtk_widget_show (drawing_area);
```

```
    gtk_widget_show (window);
    gtk_main ();
    return 0;
}
```

分栏窗口构件 Paned Window Widgets

如 果想要将一个窗口分成两个部分，可以使用分栏窗口构件(The paned window widgets)。窗口两部分的尺寸由用户控制，它们之间有一个凹槽，上面有一个手柄，用户可以拖动此手柄改变两部分的比例。窗口划分可以是水平 (HPaned)或垂直的(VPaned)。

用以下函数之一创建一个新的分栏窗口：

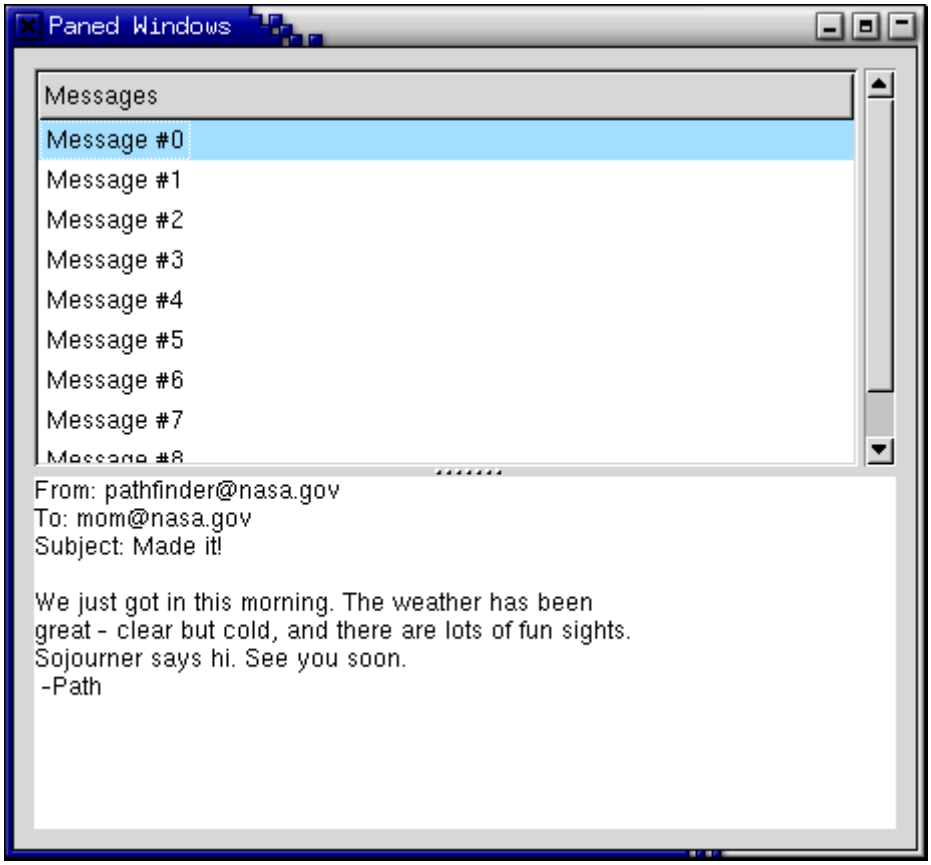
```
GtkWidget *gtk_hpaned_new (void);

GtkWidget *gtk_vpaned_new (void);
```

创建了分栏窗口构件后，可以在它的两边添加子构件。用下面的函数完成：

```
void gtk_paned_add1 (GtkPaned *paned, GtkWidget *child);

void gtk_paned_add2 (GtkPaned *paned, GtkWidget *child);
```

gtk_paned_add1()将子构件添加到分栏窗口的左边或顶部。gtk_paned_add2()将子构件添加到分栏窗口的右边或下部。

在 下面的示例中，创建了一个假想的 email 程序的用户界面。窗口被垂直划分为两个部分，上面部分显示一个 email 信息列表，下部显示 email 文本信 息。程序大部分都是漂亮直接的。有两点要注意：在文本构件实例化(realized)前文本不能加到文本构件中。但你可以调用 gtk_widget_realize() 函数完成，不过，作为一个可变通技巧的展示，我们为构件的 "realize" 信号设置一个信号处理函数，并在这个函数里面添加文本。还有，我们需要为表格(table)中包含文本窗口和它的滚动条的格子设置 GTK_SHRINK 选项，以便当窗口的下面部分变小时，下部的构件能够自动地缩小，而不是被压到窗口的底部去，只部分显示。



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

/* 创建一个"信息"列表 */
GtkWidget *create_list( void )
{

     GtkWidget *scrolled_window;
     GtkWidget *tree_view;
     GtkListStore *model;
     GtkTreeIter iter;
     GtkCellRenderer *cell;
     GtkTreeViewColumn *column;

     int i;

     /* 创建一个新的滚动窗口(scrolled window)，只有需要时，滚动条才出现 */
     scrolled_window = gtk_scrolled_window_new (NULL, NULL);
     gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                     GTK_POLICY_AUTOMATIC,
                                     GTK_POLICY_AUTOMATIC);

     model = gtk_list_store_new (1, G_TYPE_STRING);
     tree_view = gtk_tree_view_new ();
     gtk_scrolled_window_add_with_viewport (GTK_SCROLLED_WINDOW (scrolled_window),
                                            tree_view);
     gtk_tree_view_set_model (GTK_TREE_VIEW (tree_view), GTK_TREE_MODEL (model));
     gtk_widget_show (tree_view);

     /* 在窗口中添加一些消息 */
     for (i = 0; i < 10; i++) {
          gchar *msg = g_strdup_printf ("Message #%d", i);
          gtk_list_store_append (GTK_LIST_STORE (model), &iter);
          gtk_list_store_set (GTK_LIST_STORE (model),
                              &iter,
                              0, msg,
                              -1);
          g_free (msg);
     }

     cell = gtk_cell_renderer_text_new ();

     column = gtk_tree_view_column_new_with_attributes ("Messages",
                                                        cell,
                                                        "text", 0,
                                                        NULL);

     gtk_tree_view_append_column (GTK_TREE_VIEW (tree_view),
                                  GTK_TREE_VIEW_COLUMN (column));

     return scrolled_window;
}

/* 向文本构件中添加一些文本 - 这是当窗口被实例化(realized)时调用的回调函数。
 * 我们也可以用 gtk_widget_realize 强行将窗口实例化，但这必须在它的层次关系
 *  确定后(be part of a hierarchy)才行。 */
 // 译者注： 构件的层次关系就是其 parent 被确定。将一个子构件加到一个容器中
 // 时，其 parent 就是这个容器。层次关系被确定要求，其 parent 的 parent...也
 // 确定了。顶级窗口可以不要 parent。只是我的经验理解。

void insert_text (GtkTextBuffer *buffer)
{
    GtkTextIter iter;

    gtk_text_buffer_get_iter_at_offset (buffer, &iter, 0);
```

```
    gtk_text_buffer_insert (buffer, &iter,
     "From: pathfinder@nasa.gov\n"
     "To: mom@nasa.gov\n"
     "Subject: Made it!\n"
     "\n"
     "We just got in this morning. The weather has been\n"
     "great - clear but cold, and there are lots of fun sights.\n"
     "Sojourner says hi. See you soon.\n"
     " -Path\n", -1);
}

/* 创建一个滚动文本区域，用于显示一个"信息" */
GtkWidget *create_text( void )
{
    GtkWidget *scrolled_window;
    GtkWidget *view;
    GtkTextBuffer *buffer;

    view = gtk_text_view_new ();
    buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (view));

    scrolled_window = gtk_scrolled_window_new (NULL, NULL);
    gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                    GTK_POLICY_AUTOMATIC,
                                    GTK_POLICY_AUTOMATIC);

    gtk_container_add (GTK_CONTAINER (scrolled_window), view);
    insert_text (buffer);

    gtk_widget_show_all (scrolled_window);

    return scrolled_window;
}

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

     gtk_init (&argc, &argv);

     window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
     gtk_window_set_title (GTK_WINDOW (window), "Paned Windows");
     g_signal_connect (G_OBJECT (window), "destroy",
                       G_CALLBACK (gtk_main_quit), NULL);
     gtk_container_set_border_width (GTK_CONTAINER (window), 10);
     gtk_widget_set_size_request (GTK_WIDGET (window), 450, 400);

     /* 在顶级窗口上添加一个垂直分栏窗口构件 */

     vpaned = gtk_vpaned_new ();
     gtk_container_add (GTK_CONTAINER (window), vpaned);
     gtk_widget_show (vpaned);

     /* 在分栏窗口的两部分各添加一些构件 */

     list = create_list ();
     gtk_paned_add1 (GTK_PANED (vpaned), list);
     gtk_widget_show (list);

     text = create_text ();
     gtk_paned_add2 (GTK_PANED (vpaned), text);
     gtk_widget_show (text);
     gtk_widget_show (window);
```

```
    gtk_main ();

    return 0;
}
```

视角 Viewports

一般很少直接使用视角(Viewport)构件。多数情况下是使用滚动窗口构件，它内部使用了视角。

视角构件允许在其中放置一个超过自身大小的构件，这样你可以一次看构件的一部分。它用调整对象定义当前显示的区域。

用下面的函数创建一个视角。

```
GtkWidget *gtk_viewport_new( GtkAdjustment *hadjustment,
                             GtkAdjustment *vadjustment );
```

可以看到，创建构件时能够指定构件使用的水平和垂直调整对象。如果给函数传递 NULL 参数，构件会自己创建调整对象。

创建构件后，可以用下面四个函数取得和设置它的调整对象：

```
GtkAdjustment *gtk_viewport_get_hadjustment (GtkViewport *viewport );

GtkAdjustment *gtk_viewport_get_vadjustment (GtkViewport *viewport );

void gtk_viewport_set_hadjustment( GtkViewport   *viewport,
                                   GtkAdjustment *adjustment );

void gtk_viewport_set_vadjustment( GtkViewport   *viewport,
                                   GtkAdjustment *adjustment );
```

剩下的这个函数用于改变视角的外观：

```
void gtk_viewport_set_shadow_type( GtkViewport   *viewport,
                                   GtkShadowType  type );
```

type 参数可以取以下值：

```
  GTK_SHADOW_NONE,
  GTK_SHADOW_IN,
  GTK_SHADOW_OUT,
  GTK_SHADOW_ETCHED_IN,
  GTK_SHADOW_ETCHED_OUT
```

滚动窗口 Scrolled Windows

滚动窗口(Scrolled windows)用于创建一个可滚动区域，并将其它构件放入其中。可以在滚动窗口中插入任何其它构件，在其内部的构件不论尺寸大小都可以通过滚动条访问到。

用下面的函数创建新的滚动窗口。

```
GtkWidget *gtk_scrolled_window_new( GtkAdjustment *hadjustment,
                                    GtkAdjustment *vadjustment );
```

第一个参数是水平方向的调整对象，第二个参数是垂直方向的调整对象。它们一般都设置为 NULL。

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

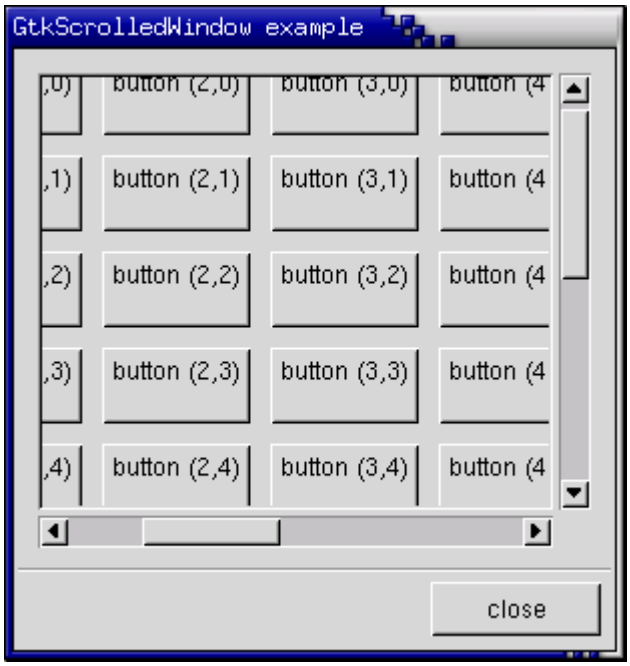
这个函数可以设置滚动条出现的方式。第一个参数是要设置的滚动窗口，第二个设置水平滚动条出现的方式，第三个参数设置垂直滚动条出现的方式。

滚动条的方式取值可以为 GTK_POLICY_AUTOMATIC 或 GTK_POLICY_ALWAYS。当要求滚动条根据需要自动出现时，可设为 GTK_POLICY_AUTOMATIC ；若设为 GTK_POLICY_ALWAYS，滚动条会一直出现在滚动窗口上。

可以用下面的函数将构件放到滚动窗口里：

```
void gtk_scrolled_window_add_with_viewport( GtkScrolledWindow *scrolled_window,
                                            GtkWidget         *child);
```

下面是一个简单示例：在滚动窗口构件中放置一个表格构件，并在表格中放 100 个开关按钮。我将只对那些你可能比较陌生的代码作些注释。




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

void destroy( GtkWidget *widget,
              gpointer   data )
{
    gtk_main_quit ();
}

int main( int   argc,
          char *argv[] )
{
    static GtkWidget *window;
    GtkWidget *scrolled_window;
    GtkWidget *table;
    GtkWidget *button;
    char buffer[32];
    int i, j;

    gtk_init (&argc, &argv);

    /* 创建一个新的对话框窗口，滚动窗口就放在这个窗口上 */
    window = gtk_dialog_new ();
    g_signal_connect (G_OBJECT (window), "destroy",
                      G_CALLBACK (destroy), NULL);
    gtk_window_set_title (GTK_WINDOW (window), "GtkScrolledWindow example");
    gtk_container_set_border_width (GTK_CONTAINER (window), 0);
    gtk_widget_set_size_request (window, 300, 300);

    /* 创建一个新的滚动窗口。 */
    scrolled_window = gtk_scrolled_window_new (NULL, NULL);

    gtk_container_set_border_width (GTK_CONTAINER (scrolled_window), 10);
```

```
    /* 滚动条的出现方式可以是 GTK_POLICY_AUTOMATIC 或 GTK_POLICY_ALWAYS。
     * 设为 GTK_POLICY_AUTOMATIC 将自动决定是否需要出现滚动条
     * 而设为 GTK_POLICY_ALWAYS，将一直显示一个滚动条
     * 第一个是设置水平滚动条，第二个是垂直滚动条 */
    gtk_scrolled_window_set_policy (GTK_SCROLLED_WINDOW (scrolled_window),
                                    GTK_POLICY_AUTOMATIC, GTK_POLICY_ALWAYS);
    /* 对话框窗口内部包含一个 vbox 构件 */
    gtk_box_pack_start (GTK_BOX (GTK_DIALOG(window)->vbox), scrolled_window,
                        TRUE, TRUE, 0);
    gtk_widget_show (scrolled_window);

    /* 创建一个包含 10×10 个格子的表格 */
    table = gtk_table_new (10, 10, FALSE);

    /* 设置 x 和 y 方向的行间间距为 10 像素 */
    gtk_table_set_row_spacings (GTK_TABLE (table), 10);
    gtk_table_set_col_spacings (GTK_TABLE (table), 10);

    /* 将表格组装到滚动窗口中 */
    gtk_scrolled_window_add_with_viewport (
                    GTK_SCROLLED_WINDOW (scrolled_window), table);
    gtk_widget_show (table);

    /* 简单地在表格中添加许多开关按钮以展示滚动窗口 */
    for (i = 0; i < 10; i++)
        for (j = 0; j < 10; j++) {
            sprintf (buffer, "button (%d,%d)\n", i, j);
            button = gtk_toggle_button_new_with_label (buffer);
            gtk_table_attach_defaults (GTK_TABLE (table), button,
                                       i, i+1, j, j+1);
            gtk_widget_show (button);
        }

    /* 在对话框的底部添加一个"close"按钮 */
    button = gtk_button_new_with_label ("close");
    g_signal_connect_swapped (G_OBJECT (button), "clicked",
                              G_CALLBACK (gtk_widget_destroy),
                              window);

    /* 让按钮能被缺省 */

    GTK_WIDGET_SET_FLAGS (button, GTK_CAN_DEFAULT);
    gtk_box_pack_start (GTK_BOX (GTK_DIALOG (window)->action_area), button, TRUE, TRUE, 0);

    /* 将按钮固定为缺省按钮，只要按回车键就相当于点击了这个按钮 */
    gtk_widget_grab_default (button);
    gtk_widget_show (button);

    gtk_widget_show (window);

    gtk_main();

    return 0;
}
```

尝试改变窗口的大小，可以看到滚动条是如何起作用的。还可以用 gtk_widget_set_size_request() 函数设置窗口或其它构件的缺省尺寸。

按钮盒 Button Boxes

按钮盒(Button Boxes)可以很方便地快速布置一组按钮。它有水平和垂直两种样式。你可以用以下函数创建水平或垂直按钮盒：

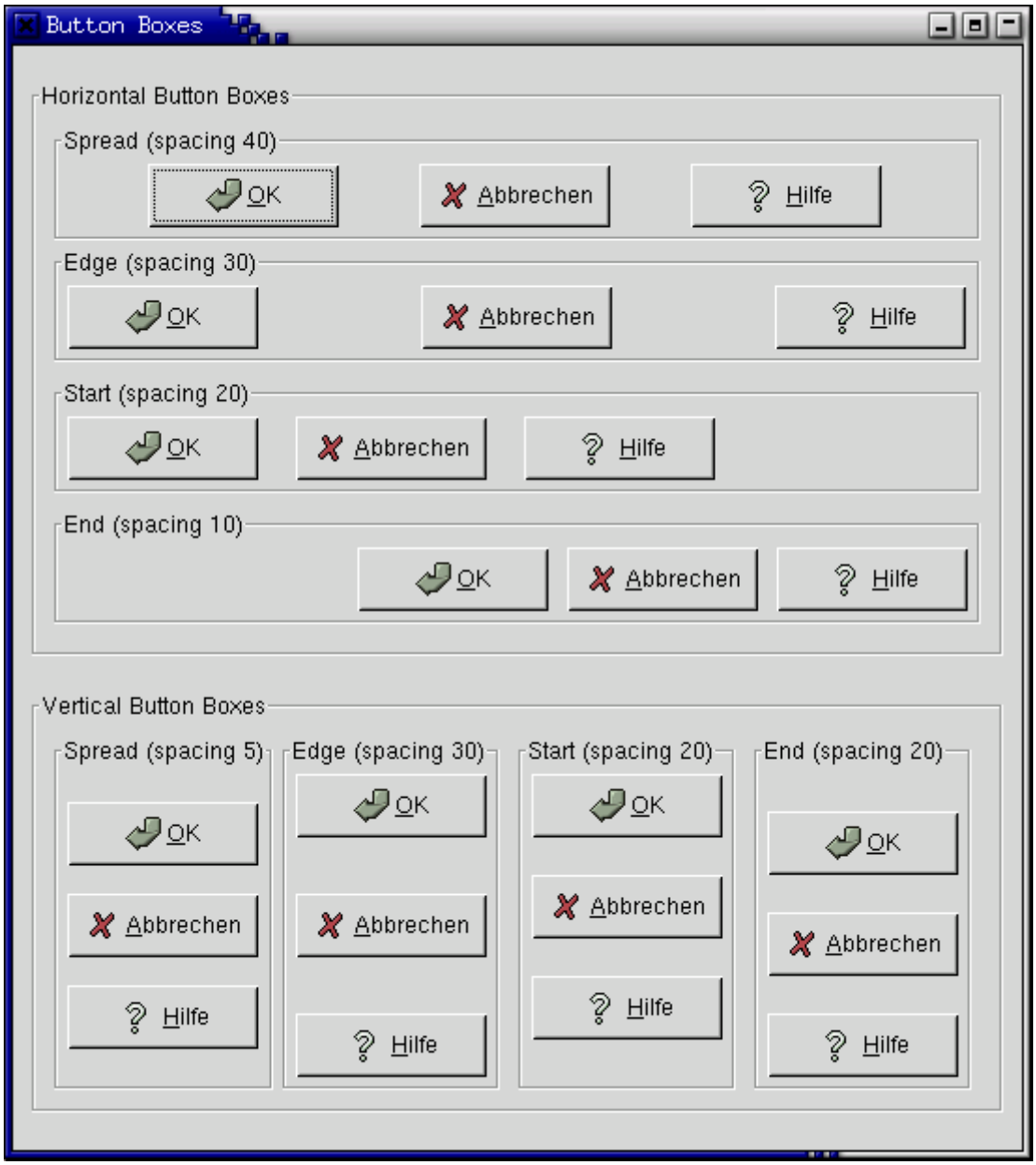
```
GtkWidget *gtk_hbutton_box_new( void );
```

GtkWidget *gtk_vbutton_box_new(void);

将按钮添加到按钮盒中可以用下面这个通常的函数:

```
gtk_container_add (GTK_CONTAINER (button_box), child_widget);
```

下面的示例演示了按钮盒的不同布局设置。




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

/* 用指定的参数创建一个按钮盒 */
GtkWidget *create_bbox( gint  horizontal,
                        char *title,
                        gint  spacing,
                        gint  child_w,
                        gint  child_h,
                        gint  layout )
{
  GtkWidget *frame;
  GtkWidget *bbox;
```

```
  GtkWidget *button;

  frame = gtk_frame_new (title);

  if (horizontal)
    bbox = gtk_hbutton_box_new ();
  else
    bbox = gtk_vbutton_box_new ();

  gtk_container_set_border_width (GTK_CONTAINER (bbox), 5);
  gtk_container_add (GTK_CONTAINER (frame), bbox);

  /* 设置按钮盒的外观 */
  gtk_button_box_set_layout (GTK_BUTTON_BOX (bbox), layout);
  gtk_box_set_spacing (GTK_BOX (bbox), spacing);
  /*gtk_button_box_set_child_size (GTK_BUTTON_BOX (bbox), child_w, child_h);*/

  button = gtk_button_new_from_stock (GTK_STOCK_OK);
  gtk_container_add (GTK_CONTAINER (bbox), button);

  button = gtk_button_new_from_stock (GTK_STOCK_CANCEL);
  gtk_container_add (GTK_CONTAINER (bbox), button);

  button = gtk_button_new_from_stock (GTK_STOCK_HELP);
  gtk_container_add (GTK_CONTAINER (bbox), button);

  return frame;
}

int main( int   argc,
          char *argv[] )
{
  static GtkWidget* window = NULL;
  GtkWidget *main_vbox;
  GtkWidget *vbox;
  GtkWidget *hbox;
  GtkWidget *frame_horz;
  GtkWidget *frame_vert;

  /* 初始化 */
  gtk_init (&argc, &argv);

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Button Boxes");

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

  gtk_container_set_border_width (GTK_CONTAINER (window), 10);

  main_vbox = gtk_vbox_new (FALSE, 0);
  gtk_container_add (GTK_CONTAINER (window), main_vbox);

  frame_horz = gtk_frame_new ("Horizontal Button Boxes");
  gtk_box_pack_start (GTK_BOX (main_vbox), frame_horz, TRUE, TRUE, 10);

  vbox = gtk_vbox_new (FALSE, 0);
  gtk_container_set_border_width (GTK_CONTAINER (vbox), 10);
  gtk_container_add (GTK_CONTAINER (frame_horz), vbox);

  gtk_box_pack_start (GTK_BOX (vbox),
          create_bbox (TRUE, "Spread (spacing 40)", 40, 85, 20, GTK_BUTTONBOX_SPREAD),
                  TRUE, TRUE, 0);

  gtk_box_pack_start (GTK_BOX (vbox),
          create_bbox (TRUE, "Edge (spacing 30)", 30, 85, 20, GTK_BUTTONBOX_EDGE),
                  TRUE, TRUE, 5);
```

```
  gtk_box_pack_start (GTK_BOX (vbox),
            create_bbox (TRUE, "Start (spacing 20)", 20, 85, 20, GTK_BUTTONBOX_START),
                        TRUE, TRUE, 5);

  gtk_box_pack_start (GTK_BOX (vbox),
            create_bbox (TRUE, "End (spacing 10)", 10, 85, 20, GTK_BUTTONBOX_END),
                        TRUE, TRUE, 5);

  frame_vert = gtk_frame_new ("Vertical Button Boxes");
  gtk_box_pack_start (GTK_BOX (main_vbox), frame_vert, TRUE, TRUE, 10);

  hbox = gtk_hbox_new (FALSE, 0);
  gtk_container_set_border_width (GTK_CONTAINER (hbox), 10);
  gtk_container_add (GTK_CONTAINER (frame_vert), hbox);

  gtk_box_pack_start (GTK_BOX (hbox),
            create_bbox (FALSE, "Spread (spacing 5)", 5, 85, 20, GTK_BUTTONBOX_SPREAD),
                        TRUE, TRUE, 0);

  gtk_box_pack_start (GTK_BOX (hbox),
            create_bbox (FALSE, "Edge (spacing 30)", 30, 85, 20, GTK_BUTTONBOX_EDGE),
                        TRUE, TRUE, 5);

  gtk_box_pack_start (GTK_BOX (hbox),
            create_bbox (FALSE, "Start (spacing 20)", 20, 85, 20, GTK_BUTTONBOX_START),
                        TRUE, TRUE, 5);

  gtk_box_pack_start (GTK_BOX (hbox),
            create_bbox (FALSE, "End (spacing 20)", 20, 85, 20, GTK_BUTTONBOX_END),
                        TRUE, TRUE, 5);

  gtk_widget_show_all (window);

  /* 进入事件循环 */
  gtk_main ();

  return 0;
}
```

工具栏 Toolbar

工具栏(Toolbars)常用来将一些构件分组，这样能够简化定制它们的外观和布局。典型情况下工具栏由带图标和标签以及工具提示的按钮组成，不过，其它构件也可以放在工具栏里面。最后，各工具栏组件可以水平或垂直排列，还可以显示图标或标签，或者两者都显示。

用下面的函数创建一个工具栏（可能有些人已经猜到了）：

```
GtkWidget *gtk_toolbar_new( void );
```

创 建工具栏以后，可以向其中追加、前插和插入工具栏项（这里意指简单文本字符串）或元素（这里意指任何构件类型）。要想描述一个工具栏上的对象，需要一个标 签文本、一个工具提示文本、一个私有工具提示文本、一个图标和一个回调函数。例如，要前插或追加一个按钮，应该使用下面的函数：

```
GtkWidget *gtk_toolbar_append_item( GtkToolbar    *toolbar,
                                    const char    *text,
                                    const char    *tooltip_text,
                                    const char    *tooltip_private_text,
                                    GtkWidget     *icon,
                                    GtkSignalFunc  callback,
                                    gpointer       user_data );

GtkWidget *gtk_toolbar_prepend_item( GtkToolbar    *toolbar,
                                     const char    *text,
                                     const char    *tooltip_text,
                                     const char    *tooltip_private_text,
```

```
                                        GtkWidget     *icon,
                                        GtkSignalFunc  callback,
                                        gpointer       user_data );
```

如果要使用 gtk_toolbar_insert_item()，除上面函数中要指定的参数以外，还要指定插入对象的位置，形式如下：

```
GtkWidget *gtk_toolbar_insert_item( GtkToolbar    *toolbar,
                                    const char    *text,
                                    const char    *tooltip_text,
                                    const char    *tooltip_private_text,
                                    GtkWidget     *icon,
                                    GtkSignalFunc  callback,
                                    gpointer       user_data,
                                    gint           position );
```

要简单地在工具栏项之间添加空白区域，可以使用下面的函数：

```
void gtk_toolbar_append_space( GtkToolbar *toolbar );

void gtk_toolbar_prepend_space( GtkToolbar *toolbar );

void gtk_toolbar_insert_space( GtkToolbar *toolbar,
                               gint        position );
```

如果需要，工具栏的放置方向和它的式样可以在运行时用下面的函数设置：

```
void gtk_toolbar_set_orientation( GtkToolbar     *toolbar,
                                  GtkOrientation  orientation );

void gtk_toolbar_set_style( GtkToolbar      *toolbar,
                            GtkToolbarStyle  style );

void gtk_toolbar_set_tooltips( GtkToolbar *toolbar,
                               gint        enable );
```

上面的 orientation 参数取 GTK_ORIENTATION_HORIZONTAL 或 GTK_ORIENTATION_VERTICAL。style 参数用于设置工具栏项的外观，可以取 GTK_TOOLBAR_ICONS，GTK_TOOLBAR_TEXT 或 GTK_TOOLBAR_BOTH。

要想了解工具栏还能做什么，看一看下面的程序(在代码之间我们插入了一些解释)：

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

/* 这个函数连接到 Close 按钮或者从窗口管理器关闭窗口的事件上 */
gint delete_event (GtkWidget *widget, GdkEvent *event, gpointer data)
{
  gtk_main_quit ();
  return FALSE;
}
```

上面的代码和其它的 GTK 应用程序差别不大，有一点不同的是：我们包含了一个漂亮的 XPM 图片，用作所有按钮的图标。

```
GtkWidget* close_button; /* 这个按钮将引发一个信号以
                          * 关闭应用程序 */
GtkWidget* tooltips_button; /* 启用/禁用工具提示 */
GtkWidget* text_button,
         * icon_button,
         * both_button; /* 切换工具栏风格的单选按钮 */
GtkWidget* entry; /* 一个文本输入构件，用于演示任何构件都可以组装到
                   * 工具栏里 */
```

事实上，不是上面所有的构件都是必须的，我把它们放在一起，是为了让事情更清晰。

```
/* 很简单...当按钮进行状态切换时，我们检查哪一个按钮是活动的，依此设置工具栏的式样
```

```
 * 注意，工具栏是作为用户数据传递到回调函数的！ */
void radio_event (GtkWidget *widget, gpointer data)
{
  if (GTK_TOGGLE_BUTTON (text_button)->active)
    gtk_toolbar_set_style (GTK_TOOLBAR (data), GTK_TOOLBAR_TEXT);
  else if (GTK_TOGGLE_BUTTON (icon_button)->active)
    gtk_toolbar_set_style (GTK_TOOLBAR (data), GTK_TOOLBAR_ICONS);
  else if (GTK_TOGGLE_BUTTON (both_button)->active)
    gtk_toolbar_set_style (GTK_TOOLBAR (data), GTK_TOOLBAR_BOTH);
}

/* 更简单，检查给定开关按钮的状态，依此启用或禁用工具提示 */
void toggle_event (GtkWidget *widget, gpointer data)
{
  gtk_toolbar_set_tooltips (GTK_TOOLBAR (data),
                            GTK_TOGGLE_BUTTON (widget)->active );
}
```

上面只是当工具栏上的一个按钮被按下时要调用的两个回调函数。你应该已经熟悉了这些东西，如果你已经使用过开关按钮（以及单选按钮）。

```
int main (int argc, char *argv[])
{
  /* 下面是主窗口（一个对话框）和一个把柄盒(handlebox) */
  GtkWidget* dialog;
  GtkWidget* handlebox;

  /* 好了,我们需要一个工具栏,一个带掩码(mask)的图标（所有的按钮共用一个掩码）以及
   * 一个放图标的图标构件（但我们会为每个按钮创建一个分割构件） */
  GtkWidget * toolbar;
  GtkWidget * iconw;

  /* 这个在所有的 GTK 程序中都被调用。 */
  gtk_init (&argc, &argv);

  /* 用给定的标题和尺寸创建一个新窗口 */
  dialog = gtk_dialog_new ();
  gtk_window_set_title (GTK_WINDOW (dialog), "GTKToolbar Tutorial");
  gtk_widget_set_size_request (GTK_WIDGET (dialog), 600, 300);
  GTK_WINDOW (dialog)->allow_shrink = TRUE;

  /* 在关闭窗口时退出 */
  g_signal_connect (G_OBJECT (dialog), "delete_event",
                    G_CALLBACK (delete_event), NULL);

  /* 需要实例化窗口,因为我们要在它的内容中为工具栏设置图片 */
  gtk_widget_realize (dialog);

  /* 我们将工具栏放在一个手柄构件(handle box)上，
   * 这样它可以从主窗口上移开 */
  handlebox = gtk_handle_box_new ();
  gtk_box_pack_start (GTK_BOX (GTK_DIALOG (dialog)->vbox),
                      handlebox, FALSE, FALSE, 5);
```

上 面的代码和任何其它 Gtk 应用程序都差不多。它们进行 GTK 初始化，创建主窗口等。唯一需要解释的是：一个手柄盒(a handle box)。手柄盒只是一个可以在其中组装构件的盒子。它和普通盒子的区别在于它能从一个父窗口移开(事实上，手柄盒保留在父窗口上，但是它缩小为一个非常 小的矩形，同时它的所有内容重新放在一个新的可自由移动的浮动窗口上)。拥有一个可浮动工具栏给人感觉非常好，所以这两种构件经常同时使用。

```
  /* 工具栏设置为水平的，同时带有图标和文本
   * 在每个项之间有 5 像素的间距，
   * 并且，我们也将它放在手柄盒上 */
  toolbar = gtk_toolbar_new ();
  gtk_toolbar_set_orientation (GTK_TOOLBAR (toolbar), GTK_ORIENTATION_HORIZONTAL);
```

```
gtk_toolbar_set_style (GTK_TOOLBAR (toolbar), GTK_TOOLBAR_BOTH);
gtk_container_set_border_width (GTK_CONTAINER (toolbar), 5);
gtk_toolbar_set_space_size (GTK_TOOLBAR (toolbar), 5);
gtk_container_add (GTK_CONTAINER (handlebox), toolbar);
```

上面的代码初始化工具栏构件。

```
/* 工具栏上第一项是<close>按钮 */
iconw = gtk_image_new_from_file ("gtk.xpm"); /* 图标构件 */
close_button =
  gtk_toolbar_append_item (GTK_TOOLBAR (toolbar), /* 工具栏 */
                           "Close",            /* 按钮标签 */
                           "Closes this app",    /* 按钮的工具提示 */
                           "Private",          /* 工具提示的私有信息 */
                           iconw,              /* 图标构件 */
                           GTK_SIGNAL_FUNC (delete_event), /* 一个信号 */
                           NULL);
gtk_toolbar_append_space (GTK_TOOLBAR (toolbar)); /* 工具栏项后的空白 */
```

在 上面的代码中，可以看到最简单的情况：在工具栏上增加一个按钮。在追加一个新的工具栏项前，必须构造一个图片(image)构件用作该项的图标，这个步骤 我们要对每一个工具栏项重复一次。在工具栏项之间还要增加间隔空间，这样后面的工具栏项就不会一个接一个紧挨着。可以看 到，gtk_toolbar_append_item()返回一个指向新创建的按钮构件的指针，所以我们可以用正常的方式使用它。

```
/* 现在，我们创建单选按钮组... */
iconw = gtk_image_new_from_file ("gtk.xpm");
icon_button = gtk_toolbar_append_element (
                      GTK_TOOLBAR (toolbar),
                      GTK_TOOLBAR_CHILD_RADIOBUTTON, /* 元素类型 */
                      NULL,                  /* 指向构件的指针 */
                      "Icon",                /* 标签 */
                      "Only icons in toolbar",    /* 工具提示 */
                      "Private",              /* 工具提示的私有字符串 */
                      iconw,                 /* 图标 */
                      GTK_SIGNAL_FUNC (radio_event), /* 信号 */
                      toolbar);              /* 信号传递的数据 */
gtk_toolbar_append_space (GTK_TOOLBAR (toolbar));
```

这里我们开始创建一个单选按钮组。用 gtk_toolbar_append_element 就行了。事实上，使用这个函数，我们能够添加简单的工具栏项或空白间隔（类型为 GTK_TOOLBAR_CHILD_SPACE 或 GTK_TOOLBAR_CHILD_BUTTON）。在上面的示例中，我们先创建了一个单选按钮组。要为这个组创建其它单选按钮，需要一个指向前一个按钮的指针，这样按钮的清单可以很容易组织起来（看在本文档前面部分的单选按钮节）。

```
/* 后面的单选按钮引用前面创建的 */
iconw = gtk_image_new_from_file ("gtk.xpm");
text_button =
  gtk_toolbar_append_element (GTK_TOOLBAR (toolbar),
                              GTK_TOOLBAR_CHILD_RADIOBUTTON,
                              icon_button,
                              "Text",
                              "Only texts in toolbar",
                              "Private",
                              iconw,
                              GTK_SIGNAL_FUNC (radio_event),
                              toolbar);
gtk_toolbar_append_space (GTK_TOOLBAR (toolbar));

iconw = gtk_image_new_from_file ("gtk.xpm");
both_button =
  gtk_toolbar_append_element (GTK_TOOLBAR (toolbar),
                              GTK_TOOLBAR_CHILD_RADIOBUTTON,
                              text_button,
```

```
                                    "Both",
                                    "Icons and text in toolbar",
                                    "Private",
                                    iconw,
                                    GTK_SIGNAL_FUNC (radio_event),
                                    toolbar);
  gtk_toolbar_append_space (GTK_TOOLBAR (toolbar));
  gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (both_button), TRUE);
```

最后，我们必须手工设置其中一个按钮的状态(否则它们全部处于活动状态，并阻止我们在它们之间做出选择)。

```
  /* 下面只是一个简单的开关按钮 */
  iconw = gtk_image_new_from_file ("gtk.xpm");
  tooltips_button =
    gtk_toolbar_append_element (GTK_TOOLBAR (toolbar),
                                    GTK_TOOLBAR_CHILD_TOGGLEBUTTON,
                                    NULL,
                                    "Tooltips",
                                    "Toolbar with or without tips",
                                    "Private",
                                    iconw,
                                    GTK_SIGNAL_FUNC (toggle_event),
                                    toolbar);
  gtk_toolbar_append_space (GTK_TOOLBAR (toolbar));
  gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (tooltips_button), TRUE);
```

开关按钮的创建方法就很明显了（如果你已经知道怎么创建单选按钮了）。

```
  /* 要将一个构件组装到工具栏上，只需创建它，然后将它追
   * 加到工具栏上，同时设置合适的工具提示 */
  entry = gtk_entry_new ();
  gtk_toolbar_append_widget (GTK_TOOLBAR (toolbar),
                                  entry,
                                  "This is just an entry",
                                  "Private");

  /* 因为它不是工具栏自己创建的，所以我们还需要显示它 */
  gtk_widget_show (entry);
```

可以看到，将任何构件添加到工具栏上都是非常简单的。唯一要记住的是，这个构件必须手工显示(与此相反，工具栏自己创建的工具栏项随工具栏一起显示)。

```
  /* 好了，现在可以显示所有的东西了 */
  gtk_widget_show (toolbar);
  gtk_widget_show (handlebox);
  gtk_widget_show (dialog);

  /* 进入主循环，等待用户的操作 */
  gtk_main ();

  return 0;
}
```

这样，我们就到了工具栏教程的末尾。当然，还需要一个漂亮的 XPM 图标。下面就是：

```
/* XPM */
static char * gtk_xpm[] = {
"32 39 5 1",
".      c none",
"+      c black",
"@      c #3070E0",
"#      c #F05050",
"$      c #35E035",
"................+...............",
".............+++++.............",
"...........+++++@@++...........",
".........+++++@@@@@@++.........",
".......++++@@@@@@@@@@++.......",
".....++++@@++++++++@@@@++.....",
"....+++@@@@++++++++++++@@@@++....",
"...+++@@@@+++@@@@@@@@++++@@@@@+....",
"..+++@@@@+++@@@@@@@@@@+++@@@@@@++..",
".++@@@@@@@+++@@@@@@@@@@@@@@@@@@@@++",
".+#+@@@@@@@++@@@@@+++@@@@@@@@@@@@@@+",
".+###++@@@@+++@@@@+++++@@@@@@@@@$@.",
".+#####++@@@@+++@@@+++@@@@@@++$$$@.",
".+#######+++@@@+++++++@@@@@@+@$$$$@.",
".+########+++@@@@+++@@@@@++@$$$$$$+.",
".+#########++++@@@@@@@@++@$$$$$$$$+.",
".+##########+###+@@@@+++$$$$$$@@$$+.",
".+########+##+###+@@++@$$$$$$++$$$+.",
".+#########+##+###+@@$$$$$$$@+@$$@+.",
".+##########+#+++@$$@+@$$@++$$$@+.",
".+###########+#++$$@+@$$++$$$$+..",
".++###########+#++$$@+@$++@$$$$+..",
".+###########+##++$$++@+++$$$$$+..",
".++##########+##+++$$++++@$$$$$+..",
".++###########+#++$$+++++@$$$@+..",
".+###########+##++$$+++@++++@$@+..",
".+###########+##++$$++@$$@+++$@@..",
".++###########+##++$$++$$$$$+@$@++.",
".++###########+##++$$++$$$$$$$$+++.",
".+++#########+##++$$++$$$$$$$@+++.",
"..+++##########+@$$+@$$$$$$+++...",
"...+++#########+@$$$$$$$$@+++....",
".....+++#######+@$$$$$$$+++......",
"......+++#####+@$$$$@++........",
".......+++####+@$$$+++.........",
".........++###+$$$@++..........",
"..........++##+$@+++...........",
"...........+++++++.............",
".............++++..............."};
```

笔记本 **Notebooks**

笔记本构件(The NoteBook Widget)是互相重叠的页面集合，每一页都包含不同的信息，且一次只有一个页面是可见的。该构件在 GUI(图形用户接口)编程中很常用。要显示大量的相似信息，同时把它们分别显示时，使用这种构件是一个很好的方法。

第一个你要知道的函数调用，你可能已经猜到了，是用来创建一个新的笔记本构件。

```
GtkWidget *gtk_notebook_new( void );
```

一旦创建了笔记本构件，就可以使用一系列的函数操作该构件。下面将对它们进行分别讨论。

先看一下怎样定位页面指示器—或称页标签，可以有四种位置：上、下、左或右。

```
void gtk_notebook_set_tab_pos( GtkNotebook     *notebook,
                               GtkPositionType  pos );
```

GtkPositionType 参数可以取以下几个值，从字面上很容易理解它们的含义：

GTK_POS_LEFT
GTK_POS_RIGHT
GTK_POS_TOP
GTK_POS_BOTTOM

GTK_POS_TOP 是缺省值。

下面看一下怎样向笔记本中添加页面。有三种方法向笔记本中添加页面。前两种方法是非常相似的。

```
void gtk_notebook_append_page( GtkNotebook *notebook,
                               GtkWidget   *child,
                               GtkWidget   *tab_label );

void gtk_notebook_prepend_page( GtkNotebook *notebook,
                                GtkWidget   *child,
                                GtkWidget   *tab_label );
```

这些函数通过向插入页面到笔记本的后端（append）或前端（prepend）来添加页面。child 是放在笔记本页面里的子构件，tab_label 是要添加的页面的标签。child 构件必须另外创建，一般是一个包含一套选项设置的容器构件，比如一个表格。

最后一个添加页面的函数与前两个函数类似，不过允许指定页面插入的位置。

```
void gtk_notebook_insert_page( GtkNotebook *notebook,
                               GtkWidget   *child,
                               GtkWidget   *tab_label,
                               gint         position );
```

其中的参数与_append_和_prepend_函数一样，还包含一个额外参数，position。该参数指定页面应该插入到哪一页。注意，第一页位置为 0。

前面介绍了怎样添加一个页面，下面介绍怎样从笔记本中删除一个页面。

```
void gtk_notebook_remove_page( GtkNotebook *notebook,
                               gint         page_num );
```

这个函数从 notebook 指定的笔记本中删除由 page_num 参数指定的页面。

用这个函数找出笔记本的当前页面：

```
gint gtk_notebook_get_current_page( GtkNotebook *notebook );
```

下 面两个函数将笔记本的页面向前或向后移动。对要操作的笔记本构件使用以下函数就可以了。注意：当笔记本正在最后一页时，调用 gtk_notebook_next_page() 函数，笔记本会跳到第一页。同样，如果笔记本在第一页，调用了函数 gtk_notebook_prev_page()，笔记本构件会跳到最后一页。

```
void gtk_notebook_next_page( GtkNoteBook *notebook );

void gtk_notebook_prev_page( GtkNoteBook *notebook );
```

下面这个函数设置“活动”页面。比如你想笔记本的第 5 页被打开，你将使用这个函数。不使用这个函数时笔记本默认显示第一页。

```
void gtk_notebook_set_current_page( GtkNotebook *notebook,
                                    gint         page_num );
```

下面两个函数分别显示或隐藏笔记本的页标签以及它的边框。

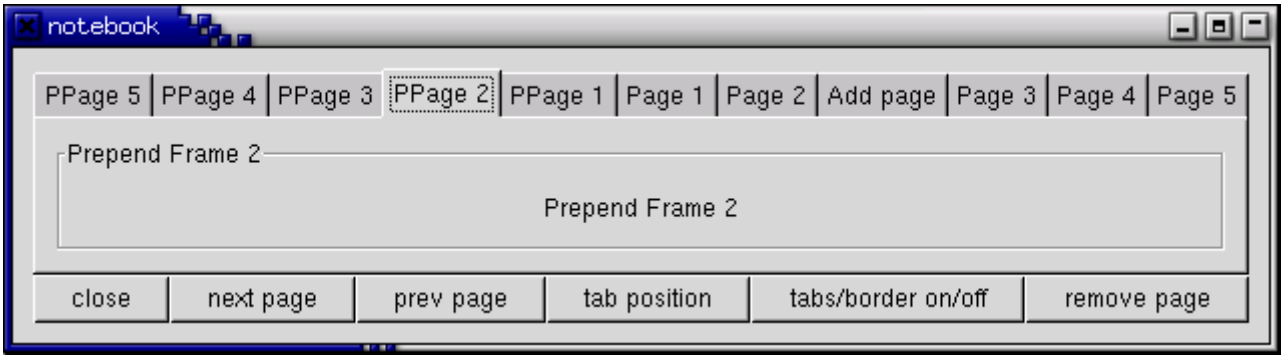
```
void gtk_notebook_set_show_tabs( GtkNotebook *notebook,
                                 gboolean     show_tabs );

void gtk_notebook_set_show_border( GtkNotebook *notebook,
                                   gboolean     show_border );
```

如果页面较多，标签页在页面上排列不下时，可以用下面这个函数。它允许用两个箭头按钮来滚动标签页。

```
void gtk_notebook_set_scrollable( GtkNotebook *notebook,
                                  gboolean     scrollable );
```

show_tabs, show_border 和 scrollable 参数可以为 TRUE 或 FALSE。

下面看一个示例，它由 GTK 发布版附带的 testgtk.c 扩展而来。这个小程序创建了一个含一个笔记本构件和 6 个按钮的窗口。笔记本包含 11 页，由三种方式添加进来：追加、插入、前插。点击按钮可以改变页标签的位置，显示/隐藏页标签和边框，删除一页，向前或向后移动标签页，以及退出程序。




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

/* 这个函数旋转页标签的位置 */
void rotate_book( GtkButton   *button,
                  GtkNotebook *notebook )
{
    gtk_notebook_set_tab_pos (notebook, (notebook->tab_pos + 1) % 4);
}

/* 显示/隐藏页标签和边框 */
void tabsborder_book( GtkButton   *button,
                      GtkNotebook *notebook )
{
    gint tval = FALSE;
    gint bval = FALSE;
    if (notebook->show_tabs == 0)
        tval = TRUE;
    if (notebook->show_border == 0)
        bval = TRUE;

    gtk_notebook_set_show_tabs (notebook, tval);
    gtk_notebook_set_show_border (notebook, bval);
}

/* 从笔记本上删除一个页面 */
void remove_book( GtkButton   *button,
                  GtkNotebook *notebook )
{
    gint page;

    page = gtk_notebook_get_current_page (notebook);
    gtk_notebook_remove_page (notebook, page);
    /* 需要刷新构件 --
     这会迫使构件重绘自身。 */
    gtk_widget_queue_draw (GTK_WIDGET (notebook));
}

gint delete( GtkWidget *widget,
             GtkWidget *event,
             gpointer   data )
{
    gtk_main_quit ();
```

```
    return FALSE;
}

int main( int argc,
          char *argv[] )
{
    GtkWidget *window;
    GtkWidget *button;
    GtkWidget *table;
    GtkWidget *notebook;
    GtkWidget *frame;
    GtkWidget *label;
    GtkWidget *checkbutton;
    int i;
    char bufferf[32];
    char bufferl[32];

    gtk_init (&argc, &argv);

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

    g_signal_connect (G_OBJECT (window), "delete_event",
                      G_CALLBACK (delete), NULL);

    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    table = gtk_table_new (3, 6, FALSE);
    gtk_container_add (GTK_CONTAINER (window), table);

    /* 创建一个新的笔记本，将标签页放在顶部 */
    notebook = gtk_notebook_new ();
    gtk_notebook_set_tab_pos (GTK_NOTEBOOK (notebook), GTK_POS_TOP);
    gtk_table_attach_defaults (GTK_TABLE (table), notebook, 0, 6, 0, 1);
    gtk_widget_show (notebook);

    /* 在笔记本后面追加几个页面 */
    for (i = 0; i < 5; i++) {
        sprintf(bufferf, "Append Frame %d", i + 1);
        sprintf(bufferl, "Page %d", i + 1);

        frame = gtk_frame_new (bufferf);
        gtk_container_set_border_width (GTK_CONTAINER (frame), 10);
        gtk_widget_set_size_request (frame, 100, 75);
        gtk_widget_show (frame);

        label = gtk_label_new (bufferf);
        gtk_container_add (GTK_CONTAINER (frame), label);
        gtk_widget_show (label);

        label = gtk_label_new (bufferl);
        gtk_notebook_append_page (GTK_NOTEBOOK (notebook), frame, label);
    }

    /* 在指定位置添加页面 */
    checkbutton = gtk_check_button_new_with_label ("Check me please!");
    gtk_widget_set_size_request (checkbutton, 100, 75);
    gtk_widget_show (checkbutton);

    label = gtk_label_new ("Add page");
    gtk_notebook_insert_page (GTK_NOTEBOOK (notebook), checkbutton, label, 2);

    /* 最后向笔记本前插页面 */
    for (i = 0; i < 5; i++) {
        sprintf (bufferf, "Prepend Frame %d", i + 1);
        sprintf (bufferl, "PPage %d", i + 1);

        frame = gtk_frame_new (bufferf);
        gtk_container_set_border_width (GTK_CONTAINER (frame), 10);
```

```
        gtk_widget_set_size_request (frame, 100, 75);
        gtk_widget_show (frame);

        label = gtk_label_new (bufferf);
        gtk_container_add (GTK_CONTAINER (frame), label);
        gtk_widget_show (label);

        label = gtk_label_new (bufferl);
        gtk_notebook_prepend_page (GTK_NOTEBOOK (notebook), frame, label);
    }

    /* 设置起始页(第 4 页) */
    gtk_notebook_set_current_page (GTK_NOTEBOOK (notebook), 3);

    /* 创建一排按钮 */
    button = gtk_button_new_with_label ("close");
    g_signal_connect_swapped (G_OBJECT (button), "clicked",
                              G_CALLBACK (delete), NULL);
    gtk_table_attach_defaults (GTK_TABLE (table), button, 0, 1, 1, 2);
    gtk_widget_show (button);

    button = gtk_button_new_with_label ("next page");
    g_signal_connect_swapped (G_OBJECT (button), "clicked",
                              G_CALLBACK (gtk_notebook_next_page),
                              notebook);
    gtk_table_attach_defaults (GTK_TABLE (table), button, 1, 2, 1, 2);
    gtk_widget_show (button);

    button = gtk_button_new_with_label ("prev page");
    g_signal_connect_swapped (G_OBJECT (button), "clicked",
                              G_CALLBACK (gtk_notebook_prev_page),
                              notebook);
    gtk_table_attach_defaults (GTK_TABLE (table), button, 2, 3, 1, 2);
    gtk_widget_show (button);

    button = gtk_button_new_with_label ("tab position");
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (rotate_book),
                      notebook);
    gtk_table_attach_defaults (GTK_TABLE (table), button, 3, 4, 1, 2);
    gtk_widget_show (button);

    button = gtk_button_new_with_label ("tabs/border on/off");
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (tabsborder_book),
                      notebook);
    gtk_table_attach_defaults (GTK_TABLE (table), button, 4, 5, 1, 2);
    gtk_widget_show (button);

    button = gtk_button_new_with_label ("remove page");
    g_signal_connect (G_OBJECT (button), "clicked",
                      G_CALLBACK (remove_book),
                      notebook);
    gtk_table_attach_defaults (GTK_TABLE (table), button, 5, 6, 1, 2);
    gtk_widget_show (button);

    gtk_widget_show (table);
    gtk_widget_show (window);

    gtk_main ();

    return 0;
}
```

我希望这些能帮你在你自己的应用程序中创建笔记本。

菜单构件

有 两种创建菜单的方法：一个容易的，一种难的。它们各有各的用处，不过一般你可以使用套件(Itemfactory)（容易的方法）。“难”的方法是直接使 用各调用来创建所有的菜单。容易的方法是使用 gtk_item_factory 调用。这要简单得多，但每种方法各有优点和缺点。

套件要容易使用得多，加新的菜单也方便些，虽然用手工方法写一些封装(wrapper)函数来创建菜单能对可用性大有帮助。使用套件，不能在菜单上增加图片或 '/' 字符。

手工创建菜单

按照现实教学中的惯例，我们将先介绍难的方法。

创建菜单栏和子菜单时要用到三种构件：

- 一个菜单项(menu item)，就是用户要选择的东西，比如，"Save"
- 一个菜单(menu)，作为菜单项的容器，以及
- 一个菜单栏(menubar)，是各个单独菜单的容器。

菜单项构件有两个不同的用处，这情况有一点复杂。既有组装到菜单里的构件，也有组装到菜单栏中，当被选中时激活菜单的构件。

让我们看一下用来创建菜单和菜单栏的函数。第一个函数用来创建一个新的菜单栏。

```
GtkWidget *gtk_menu_bar_new( void );
```

这个相当自我解释的函数创建一个新的菜单栏。你用 gtk_container_add() 组装它到一个窗口，或盒组装(box_pack)函数来将它组装到一个盒子中 — 就像按钮一样。

```
GtkWidget *gtk_menu_new( void );
```

这个函数返回指向一个新菜单的指针。它从不会真正显示（用 gtk_widget_show()），它只是一个菜单项的容器。我希望你看了后面的示例后会弄清楚一些。

接下来的三个调用用来创建被组装到菜单（和菜单栏）中的菜单项。

```
GtkWidget *gtk_menu_item_new( void );

GtkWidget *gtk_menu_item_new_with_label( const char *label );

GtkWidget *gtk_menu_item_new_with_mnemnonic( const char *label );
```

这些调用用来创建将显示的菜单项。记住要区别用 gtk_menu_new() 创建的“菜单”和用 gtk_menu_item_new() 函数创建的“菜单项”。有了相关联动作的菜单项将是一个真实的按钮，而菜单将是一个包含菜单项的容器。

gtk_menu_item_new_with_label() 和 gtk_menu_item_new() 函数正如你读了按钮部分后料想的一样。其一创建一个已经有一个标签组装进来了的新的菜单项，另一个仅仅创建一个空白的菜单项。

在创建一个菜单项后你要将它放到一个菜单里。用函数 gtk_menu_append 就行了。为了截取何时这个项被用户选中，我们要用平常的方法连接到 activate 信号。所以，如果我们要创建一个标准的 File 菜单，包括 Open，Save 和 Quit 选项，代码将像这样：

```
file_menu = gtk_menu_new ();    /* 不必显示菜单 */

/* 创建菜单项 */
open_item = gtk_menu_item_new_with_label ("Open");
save_item = gtk_menu_item_new_with_label ("Save");
quit_item = gtk_menu_item_new_with_label ("Quit");

/* 将它们加到菜单中 */
gtk_menu_append (GTK_MENU (file_menu), open_item);
gtk_menu_append (GTK_MENU (file_menu), save_item);
gtk_menu_append (GTK_MENU (file_menu), quit_item);

/* 将回调函数绑定到 activate 信号 */
```

```
g_signal_connect_swapped (G_OBJECT (open_item), "activate",
                          G_CALLBACK (menuitem_response),
                          (gpointer) "file.open");
g_signal_connect_swapped (G_OBJECT (save_item), "activate",
                          G_CALLBACK (menuitem_response),
                          (gpointer) "file.save");

/* 我们可以绑定 Quit 菜单项到我们的退出函数 */
g_signal_connect_swapped (G_OBJECT (quit_item), "activate",
                          G_CALLBACK (destroy),
                          (gpointer) "file.quit");

/* 一定要显示菜单项 */
gtk_widget_show (open_item);
gtk_widget_show (save_item);
gtk_widget_show (quit_item);
```

这时我们有了我们的菜单。现在我们要创建一个菜单栏，并为 File 条目(entry)创建一个菜单项，我们的菜单就加在这个上。代码看起来像这样:

```
menu_bar = gtk_menu_bar_new ();
gtk_container_add (GTK_CONTAINER (window), menu_bar);
gtk_widget_show (menu_bar);

file_item = gtk_menu_item_new_with_label ("File");
gtk_widget_show (file_item);
```

现在我们要把菜单和 file_item 关联起来。用这个函数可以做到:

```
void gtk_menu_item_set_submenu( GtkMenuItem *menu_item,
                                GtkWidget   *submenu );
```

那么，我们的示例接下来就是

```
gtk_menu_item_set_submenu (GTK_MENU_ITEM (file_item), file_menu);
```

所有剩下要做的就是将菜单加到菜单栏，用这个函数完成:

```
void gtk_menu_bar_append( GtkMenuBar *menu_bar,
                          GtkWidget  *menu_item );
```

在我们的情况下就像这样了:

```
gtk_menu_bar_append (GTK_MENU_BAR (menu_bar), file_item);
```

如果我们想让菜单在菜单栏上右对齐，例如帮助菜单就经常是这样，我们可以在绑定它到菜单栏之前使用下面的函数（本例中又是对 file_item 使用）。

```
void gtk_menu_item_right_justify( GtkMenuItem *menu_item );
```

这里是一个对创建一个附带了菜单的菜单栏所需步骤的概要:

- 用 gtk_menu_new() 创建一个新的菜单
- 多次调用 gtk_menu_item_new() 创建每个你想在你的菜单上出现的菜单项。并使用 gtk_menu_append() 将每个新的菜单项放到菜单上。
- 用 gtk_menu_item_new() 创建一个菜单项。这将是菜单的根(root)，上面显示的文本将自己出现在菜单栏上。
- 用 gtk_menu_item_set_submenu() 将菜单绑定到根菜单项（就是上一步创建的那个）。
- 用 gtk_menu_bar_new 创建一个新的菜单栏。在一个菜单栏上创建一系列菜单时这步只要做一次就行了。
- 用 gtk_menu_bar_append() 将根菜单项放到菜单栏上。

创建一个弹出菜单几乎也一样。不同的是菜单不会被菜单栏“自动”弹出，而是在 button-press 事件（例如）里调用函

数 gtk_menu_popup() 时明确地弹出。有这些步骤:

- 创建一个事件处理函数。它要有如下原型:

```
static gint handler (GtkWidget *widget,
                     GdkEvent  *event);
```

  并且它会根据 event 得到菜单弹出的地方。

- 在事件处理函数里，如果这是一个鼠标按钮按下事件，把 event 当作鼠标按键事件（本来就是）并像示例代码那样利用它传递信息给 gtk_menu_popup()。

- 绑定那个事件处理函数到一个构件用

```
g_signal_connect_swapped (G_OBJECT (widget), "event",
                          G_CALLBACK (handler),
                          G_OBJECT (menu));
```

  其中 widget 是你要绑定到的构件，handler 是处理函数，而 menu 是一个用 gtk_menu_new() 创建的菜单。它可以是一个也被菜单栏弹出的菜单，示例代码里就做了示范。

手工菜单示例

这些应该差不多了。现在看一个示例来帮你弄明白些。


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

static gint button_press (GtkWidget *, GdkEvent *);
static void menuitem_response (gchar *);

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

    GtkWidget *window;
    GtkWidget *menu;
    GtkWidget *menu_bar;
    GtkWidget *root_menu;
    GtkWidget *menu_items;
    GtkWidget *vbox;
    GtkWidget *button;
    char buf[128];
    int i;

    gtk_init (&argc, &argv);

    /* 创建一个新窗口 */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_widget_set_size_request (GTK_WIDGET (window), 200, 100);
    gtk_window_set_title (GTK_WINDOW (window), "GTK Menu Test");
    g_signal_connect (G_OBJECT (window), "delete_event",
                      G_CALLBACK (gtk_main_quit), NULL);

    /* 初始化菜单构件，记住，永远也不要用
     * gtk_show_widget() 来显示菜单构件。
```

```
 * 这个是包含菜单项的菜单，当你在程序的"Root Menu"上点击时
 * 它会弹出来 */
menu = gtk_menu_new ();

/* 接着我们用一个小循环为"test-menu"产生三个菜单条目。
 * 注意对 gtk_menu_append 的调用。这里我们将一序列的菜单项
 * 加到我们的菜单上。通常，我们也捕获每个菜单项的"clicked"
 * 信号并为它设置一个回调，不过在这里这个被省略了以节省空间。 */

for (i = 0; i < 3; i++)
    {
        /* 将名称复制到 buf. */
        sprintf (buf, "Test-undermenu - %d", i);

        /* 创建一个新的菜单项，名称为... */
        menu_items = gtk_menu_item_new_with_label (buf);

        /* ...并将它加到菜单。 */
        gtk_menu_shell_append (GTK_MENU_SHELL (menu), menu_items);

        /* 当菜单项被选中时做点有趣的事 */
        g_signal_connect_swapped (G_OBJECT (menu_items), "activate",
                                  G_CALLBACK (menuitem_response),
                                  g_strdup (buf));

        /* 显示构件 */
        gtk_widget_show (menu_items);
    }

/* 这个是根菜单，将成为显示在菜单栏上的标签。
 * 这里不会附上一个信号处理函数，因为它只是在
 * 被按下时弹出其余的菜单。 */
root_menu = gtk_menu_item_new_with_label ("Root Menu");

gtk_widget_show (root_menu);

/* 现在我们指定我们想要让新创建的"menu"成
 * 为"root menu"的菜单 */
gtk_menu_item_set_submenu (GTK_MENU_ITEM (root_menu), menu);

/* 将一个菜单和一个按钮放到这个纵向盒子里： */
vbox = gtk_vbox_new (FALSE, 0);
gtk_container_add (GTK_CONTAINER (window), vbox);
gtk_widget_show (vbox);

/* 创建一个菜单栏以包含菜单，并将它加到主窗口 */
menu_bar = gtk_menu_bar_new ();
gtk_box_pack_start (GTK_BOX (vbox), menu_bar, FALSE, FALSE, 2);
gtk_widget_show (menu_bar);

/* 创建一个按钮，它带了一个弹出菜单 */
button = gtk_button_new_with_label ("press me");
g_signal_connect_swapped (G_OBJECT (button), "event",
                          G_CALLBACK (button_press),
                          menu);
gtk_box_pack_end (GTK_BOX (vbox), button, TRUE, TRUE, 2);
gtk_widget_show (button);

/* 最后把菜单项添加到菜单栏上 -- 这就是我
 * 咆哮了多次的“根”菜单项 =) */
gtk_menu_shell_append (GTK_MENU_SHELL (menu_bar), root_menu);

/* 总是在最后一步显示窗口，这样它一次性整个地出现在屏幕上。 */
```

```
    gtk_widget_show (window);

    gtk_main ();

    return 0;
}

/* 对鼠标按下作出回应，弹出 widget 变量传递进来的菜单。
 *
 * 注意"widget"参数是被传递进来的菜单，不是
 * 被按下的按钮。
 */

static gint button_press( GtkWidget *widget,
                          GdkEvent *event )
{

    if (event->type == GDK_BUTTON_PRESS) {
        GdkEventButton *bevent = (GdkEventButton *) event;
        gtk_menu_popup (GTK_MENU (widget), NULL, NULL, NULL, NULL,
                        bevent->button, bevent->time);
        /* 告诉调用代码我们已经处理了这个事件；事件传播(buck)在
         * 这里停止。 */
        return TRUE;
    }

    /* 告诉调用代码我们没有处理这个事件；继续传播它。 */
    return FALSE;
}


/* 当菜单项被选中时打印一个字符串 */

static void menuitem_response( gchar *string )
{
    printf ("%s\n", string);
}
```

你也可以设置一个菜单项为不敏感(insensitive)，也可以使用一个加速表(accelerator table)，绑定按键到菜单函数。

使用套件

现在我们已经介绍了难的办法，这里介绍怎样用 gtk_item_factory 调用来做。

套件示例

这里是一个使用 GTK 套件的示例。

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

/* 必须的基本回调 */
static void print_hello( GtkWidget *w,
                         gpointer   data )
{
  g_message ("Hello, World!\n");
}

/* 这是用来生成新菜单的 GtkItemFactoryEntry 结构。
    第一项：菜单路径。下划线后的字母指出菜单打开时
```

```
            的快捷键。
    第二项：这个条目的快捷键
    第三项：回调函数。
    第四项：回调动作。这个改变被调用的函数的参数。默认是 0.
    第五项：项类型，用来定义它是哪种项，
            这里是可能的值：

            NULL            -> "<Item>"
            ""              -> "<Item>"
            "<Title>"       -> 创建一个标题(title)项
            "<Item>"        -> 创建一个简单(simple)项
            "<CheckItem>"   -> 创建一个检查(check)项
            "<ToggleItem>"  -> 创建一个开关(toggle)项
            "<RadioItem>"   -> 创建一个选择(radio)项
            <path>          -> 选择项连接到的路径
            "<Separator>"   -> 创建一个分隔线(separator)
            "<Branch>"      -> 创建一个包含子项的项（可选）
            "<LastBranch>"  -> 创建一个右对齐的分枝(branch)
*/

static GtkItemFactoryEntry menu_items[] = {
  { "/_File",         NULL,         NULL, 0, "<Branch>" },
  { "/File/_New",     "<control>N", print_hello, 0, NULL },
  { "/File/_Open",    "<control>O", print_hello, 0, NULL },
  { "/File/_Save",    "<control>S", print_hello, 0, NULL },
  { "/File/Save _As", NULL,         NULL, 0, NULL },
  { "/File/sep1",     NULL,         NULL, 0, "<Separator>" },
  { "/File/Quit",     "<control>Q", gtk_main_quit, 0, NULL },
  { "/_Options",      NULL,         NULL, 0, "<Branch>" },
  { "/Options/Test",  NULL,         NULL, 0, NULL },
  { "/_Help",         NULL,         NULL, 0, "<LastBranch>" },
  { "/_Help/About",   NULL,         NULL, 0, NULL },
};


void get_main_menu( GtkWidget  *window,
                    GtkWidget **menubar )
{
  GtkItemFactory *item_factory;
  GtkAccelGroup *accel_group;
  gint nmenu_items = sizeof (menu_items) / sizeof (menu_items[0]);

  accel_group = gtk_accel_group_new ();

  /* 这个函数初始化套件。
     参数 1：菜单类型 - 可以是 GTK_TYPE_MENU_BAR, GTK_TYPE_MENU,
             或 GTK_TYPE_OPTION_MENU。
     参数 2：菜单路径。
     参数 3：指向一个 gtk_accel_group 的指针。套件在生成菜单时设置
             好加速表(accelerator table)。
  */

  item_factory = gtk_item_factory_new (GTK_TYPE_MENU_BAR, "<main>",
                                       accel_group);

  /* 这个函数生成菜单项。把数组里项的数量，数组自身，和菜单项的任
     意回调数据依次传递给套件。 */
  gtk_item_factory_create_items (item_factory, nmenu_items, menu_items, NULL);

  /* 把新的加速组绑定到窗口。 */
  gtk_window_add_accel_group (GTK_WINDOW (window), accel_group);

  if (menubar)
```

```
    /* 最后，返回套件已经创建的菜单栏。 */
    *menubar = gtk_item_factory_get_widget (item_factory, "<main>");
}

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

  gtk_init (&argc, &argv);

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  g_signal_connect (G_OBJECT (window), "destroy",
                    G_CALLBACK (gtk_main_quit),
                    NULL);
  gtk_window_set_title (GTK_WINDOW (window), "Item Factory");
  gtk_widget_set_size_request (GTK_WIDGET (window), 300, 200);

  main_vbox = gtk_vbox_new (FALSE, 1);
  gtk_container_set_border_width (GTK_CONTAINER (main_vbox), 1);
  gtk_container_add (GTK_CONTAINER (window), main_vbox);
  gtk_widget_show (main_vbox);

  get_main_menu (window, &menubar);
  gtk_box_pack_start (GTK_BOX (main_vbox), menubar, FALSE, TRUE, 0);
  gtk_widget_show (menubar);

  gtk_widget_show (window);

  gtk_main ();

  return 0;
}
```

目前，这里只有这个示例。以后会增加一个说明和许多'o'评论。

无文档构件

这一章需要作者来写！欢迎投稿。

如果你一定要使用这些无文档构件，我强烈建议你看一下这些构件在 GTK 发布中的头文件。GTK 的函数名非常形象的。如果你想知道它如何工作，简单的看一下函数的描述就不难学会一个构件如何使用。

当你学会了这些无文档构件的所有函数，请考虑写成一个手册，以便其他人从你的经验中获益。

快捷标签 Accel Label

选项菜单 Option Menu

菜单项 Menu Items

复选菜单项 Check Menu Item

单选菜单项 Radio Menu Item

分隔菜单项 Separator Menu Item

分离菜单项 Tearoff Menu Item

曲线图 Curves

绘图区 Drawing Area

字体选择对话框 Font Selection Dialog

消息对话框 Message Dialog

Gamma 曲线图

图像 Image

插头和插座 Plugs and Sockets

树视区 Tree View

文本视区 Text View

设置构件的属性

这里描述对构件进行操作的函数。它们可以用于设置构件的风格、间距和大小，等等。

```
void gtk_widget_activate( GtkWidget *widget );

void gtk_widget_set_name( GtkWidget *widget,
                          gchar     *name );

gchar *gtk_widget_get_name( GtkWidget *widget );

void gtk_widget_set_sensitive( GtkWidget *widget,
                               gboolean   sensitive );

void gtk_widget_set_style( GtkWidget *widget,
                           GtkStyle  *style );

GtkStyle *gtk_widget_get_style( GtkWidget *widget );

GtkStyle *gtk_widget_get_default_style( void );

void gtk_widget_set_size_request ( GtkWidget *widget,
```

```
                                    gint       width,
                                    gint       height );

void gtk_widget_grab_focus( GtkWidget *widget );

void gtk_widget_show( GtkWidget *widget );

void gtk_widget_hide( GtkWidget *widget );
```

超时、IO 和 Idle 函数

超时 Timeouts

你也许想知道如何让 GTK 在 gtk_main 中做有用的工作。你有几种选择。用下面的函数，你可以创建一个超时函数，每隔一段时间该函数被调用。

```
gint gtk_timeout_add( guint32     interval,
                      GtkFunction function,
                      gpointer    data );
```

第一个参数是以毫秒为单位的你的函数被调用的时间间隔。第二个参数是你想调用的函数。第三个参数是传递给回调函数的数据。返回值是一个整型的标记，该标记可以用于停止超时，用函数：

```
void gtk_timeout_remove( gint tag );
```

你也可以通过从回调函数返回零或 FALSE 来停止超时函数。很显然，如果你想让你的函数继续被调用，应该返回一个非零值，如 TRUE。

回调函数应该声明为：

```
gint timeout_callback( gpointer data );
```

监控 IO

GDK (GTK 的底层库) 的一个极好的特性是能让它检查文件描述符 (如 open(2) 或 socket(2) 返回的) 上的数据。这对网络应用程序尤其有用。这个函数：

```
gint gdk_input_add( gint              source,
                    GdkInputCondition condition,
                    GdkInputFunction  function,
                    gpointer          data );
```

第一个参数是你想监控的文件描述符。第二个参数指定你想让 GDK 寻找什么。它可以是：

- GDK_INPUT_READ --- 当文件描述符有数据可供读取时调用你的函数。
- GDK_INPUT_WRITE --- 当文件描述符做好写的准备时调用你的函数。

我确信你已经看出来了，第三个参数是当满足上述条件时你想要调用的函数，第四个参数是传递给该函数的参数。

返回值是一个标记，该标记可用于让 GDK 停止对文件描述符的监控，用函数：

```
void gdk_input_remove( gint tag );
```

回调函数应该声明为：

```
void input_callback( gpointer          data,
                     gint              source,
                     GdkInputCondition condition );
```

source 和 condition 就是前述的参数。

Idle 函数

如果你想在程序空闲时调用一个函数怎么办?

```
gint gtk_idle_add( GtkFunction function,
                   gpointer    data );
```

该函数让 GTK 在程序空闲时调用指定的函数。

```
void gtk_idle_remove( gint tag );
```

我不会解释各参数的意义，它们跟前面的差不多。第一个参数指向的函数会在任何机会出现时被调用。和其它函数一样，返回 FALSE 停止调用 idle 函数。

高级事件和信号处理

信号函数

连接和断开信号处理函数

```
gulong g_signal_connect( GObject     *object,
                         const gchar *name,
                         GCallback    func,
                         gpointer     func_data );

gulong g_signal_connect_after( GObject       *object,
                               const gchar   *name,
                               GCallback      func,
                               gpointer       func_data );

gulong g_signal_connect_swapped( GObject       *object,
                                 const gchar   *name,
                                 GCallback      func,
                                 GObject       *slot_object );

void g_signal_handler_disconnect( GObject *object,
                                  gulong   handler_id );

void g_signal_handlers_disconnect_by_func( GObject   *object,
                                           GCallback  func,
                                           gpointer   data );
```

阻塞和反阻塞信号处理函数

```
void g_signal_handler_block( GObject *object,
                             gulong   handler_id);

void g_signal_handlers_block_by_func( GObject   *object,
                                      GCallback  func,
                                      gpointer   data );

void g_signal_handler_unblock( GObject *object,
                               gulong   handler_id );

void g_signal_handler_unblock_by_func( GObject   *object,
                                       GCallback  func,
                                       gpointer   data );
```

发出和停止信号

```
void g_signal_emit( GObject *object,
                    guint     signal_id,
                    ... );

void g_signal_emit_by_name( GObject     *object,
                            const gchar *name,
                            ... );

void g_signal_emitv( const GValue *instance_and_params,
                     guint         signal_id,
                     GQuark        detail,
                     GValue       *return_value );

void g_signal_stop_emission( GObject *object,
                             guint    signal_id,
                             GQuark   detail );

void g_signal_stop_emission_by_name( GObject   *object,
                                     const gchar *detailed_signal );
```

信号的发射和传播

信号发射是 GTK 为指定的对象和信号运行所有处理函数的过程。

首先，注意从信号发射返回的值是*最后一个*处理函数执行后返回的值。因为所有事件信号都是 GTK_RUN_LAST 类型，GTK 提供的处理函数将成为默认处理函数，除非你用 gtk_signal_connect_after() 设置连接。

一个事件 (如"button_press_event") 的处理过程是：

- 从事件发生的构件开始。
- 发出通常的 "event" 信号。如果信号处理函数返回 TRUE 值，停止所有的处理。
- 否则，发出一个指定的，"button_press_event" 信号。如果它返回 TRUE，停止所有的处理。
- 否则，转到父构件，重复前两步。
- 继续直到某些信号处理函数返回 TRUE，或者直到达到了最顶层的构件。

上述问题的一些重点是：

- 如果信号有一个默认的处理函数，那么你的处理函数返回的值不起作用，除非你用 gtk_signal_connect_after() 设置连接。(译者注：这是因为默认处理函数最后运行，而信号发射的返回值取最后一个处理函数的返回值。)
- 为 了阻止默认的信号处理函数运行，你需要用 gtk_signal_connect() 设置连接，并使用 gtk_signal_emit_stop_by_name() - 返回值只影响信号是否传播，不影响当前信号的发射。(译者注：也就是说，gtk_signal_emit_stop_by_name() 才影响信号的发射。停止发射，就使连接到这个信号的其它函数不继续被调用。)

操作选中区

概述

选中区(Selections)是 X 和 GTK 提供的图形程序之间传递信息的方法之一。一个选中区标识了一块数据，例如，用户用某种方式(比如拖动鼠标)选择的一部分文本。一个显示区(即*用户*)同一时间里只能有一个应用程序能得到选中区，所以当一个程序声称一个选中区时，前一个选中区所有者必须告诉用户旧选中区已经被放弃了。其它程序能得到选中区内容的不同的形式，叫做*目标*(targets)。可以有任意多个选中区，但大多数 X 程序只会处理一个，叫做*主选中区*(primary selection)。

大 多数情况下，一个 GTK 应用程序并不必自己处理选中区。标准的构件，比如文本输入构件，已经有了在适当的情况下(例如，当用户在文本上拖动时)声称选中区的能力，也能得到其它构 件或程序的选中区内容(比如，当用户按鼠标中键时)。然而，有些情况下你可能想使其它构件具有提供选中区的能力，或者你想得到默认未支持的目标 (targets)。

处理选中区时要理解的一个基本概念是*原子*(atom)。原子是一个唯一地标识一个字符串(在一个确定的显示区)的整数。某些原子被 X 服务器预定义了，其中一些原子在 gtk.h 中有对应的常量。例如 GDK_PRIMARY_SELECTION 常量对应于字符串"PRIMARY"。其它情况下，你要使用 gdk_atom_intern()函数，根据字符串以获得对应的原子，使用 gdk_atom_name()函数,以获得原子的名称。选中区和目标都是通过原子来标识的。

获取选中区信息

获取选中区信息是一个异步的过程。开始这个过程，调用：

```
gboolean gtk_selection_convert( GtkWidget *widget,
                                GdkAtom    selection,
                                GdkAtom    target,
                                guint32    time );
```

这个将选中区内容*转换*为 target 指定的目标形式。如果可能，time 参数应该为选中区被触发事件产生的时间。这对确认事件以用户要求它们的顺序产生有帮助。然而，如果不行(例如，转换由 "clicked" 信号触发)，你也可以使用 GDK_CURRENT_TIME 常量。

当选中区所有者响应要求时，会发送一个 "selection_received" 信号到你的程序。对应的信号处理函数将得到一个指向 GtkSelectionData 结构的指针，它的定义如下：

```
struct _GtkSelectionData
{
  GdkAtom selection;
  GdkAtom target;
  GdkAtom type;
  gint    format;
  guchar *data;
  gint    length;
};
```

你可以将 selection 和 target 值传给 gtk_selection_convert()函数。type 是一个标识选中区所有者返回数据的类型的原子。一些可能值是："STRING"，由拉丁-1(latin-1)字符组成的字符串，"ATOM"，一些原子，"INTEGER"，一个整数，等等。大多数目标只能返回一种类型。format 给定其单位(比如字符型)的比特数。通常，接收数据时你不必关心这个。data 是一个指向返回数据的指针，length 指定返回数据的长度，以字节为单位。如果 length 是负数，则表示发生错误且选中区的信息无法获得。这可能在没有应用程序拥有选中区时发生，或者你要求了一个应用程序不支持的目标形式。缓冲区实际上总会比 length 指示的长一个字节；增加的这一字节总为 0 值,这样就不必为了使字符串以 NULL 结尾而另外复制一份。

在下面的示例里，我们获取"TARGETS"形式的目标，它是一个选中区内容能转换为的目标形式的列表。

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

void selection_received( GtkWidget        *widget,
                         GtkSelectionData *selection_data,
                         gpointer          data );

/* 当用户点击"Get Targets"按钮时调用的信息处理函数 */
void get_targets( GtkWidget *widget,
                  gpointer data )
{
  static GdkAtom targets_atom = GDK_NONE;
  GtkWidget *window = (GtkWidget *)data;

  /* 得到"TARGETS"字符串对应的原子 */
  if (targets_atom == GDK_NONE)
    targets_atom = gdk_atom_intern ("TARGETS", FALSE);

  /* 要求获取主选中区的"TARGETS"形式的目标 */
  gtk_selection_convert (window, GDK_SELECTION_PRIMARY, targets_atom,
                         GDK_CURRENT_TIME);
```

```
}

/* 当选中区所有者返回数据时调用的信号处理函数 */
void selection_received( GtkWidget        *widget,
                         GtkSelectionData *selection_data,
                         gpointer          data )
{
  GdkAtom *atoms;
  GList *item_list;
  int i;

  /* **** 重要 **** 检测获取信息是否成功   */
  if (selection_data->length < 0)
    {
      g_print ("Selection retrieval failed\n");
      return;
    }
  /* 确认得到的数据为原来要求的形式 */
  if (selection_data->type != GDK_SELECTION_TYPE_ATOM)
    {
      g_print ("Selection \"TARGETS\" was not returned as atoms!\n");
      return;
    }

  /* 打印接收到的原子 */
  atoms = (GdkAtom *)selection_data->data;

  item_list = NULL;
  for (i = 0; i < selection_data->length / sizeof(GdkAtom); i++)
    {
      char *name;
      name = gdk_atom_name (atoms[i]);
      if (name != NULL)
        g_print ("%s\n",name);
      else
        g_print ("(bad atom)\n");
    }

  return;
}

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

  gtk_init (&argc, &argv);

  /* 创建顶级窗口 */

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Event Box");
  gtk_container_set_border_width (GTK_CONTAINER (window), 10);

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

  /* 创建一个按钮，用户按它以获取目标 */

  button = gtk_button_new_with_label ("Get Targets");
  gtk_container_add (GTK_CONTAINER (window), button);

  g_signal_connect (G_OBJECT (button), "clicked",
                    G_CALLBACK (get_targets), window);
  g_signal_connect (G_OBJECT (window), "selection_received",
                    G_CALLBACK (selection_received), NULL);
```

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

  gtk_main ();

  return 0;
}
```

提供选中区

提供选中区要复杂一点。你必须注册当选中区被要求时将调用的处理函数。每处理一对选中区/目标，你就要调用一次下面的函数：

```
void gtk_selection_add_target (GtkWidget          *widget,
                               GdkAtom            selection,
                               GdkAtom            target,
                               guint              info);
```

widget，selection 和 target 标识了这个处理函数要操作的要求。当对选中区的一个要求被接收，"selection_get" 信号将被调用。info 能用来标识回调函数里的指定目标。

回调函数形式如下：

```
void  "selection_get" (GtkWidget         *widget,
                       GtkSelectionData   *selection_data,
                       guint              info,
                       guint              time);
```

GtkSelectionData 和上面介绍的一样，但这次，我们要负责提供 type，format，data 和 length 值(format 值在这里很重要－X 服务器根据它来确定数据是否要以字节为单位处理。通常它的值为 8 -*如*字符型 -或 32-*如*整型)。调用下面的函数来设置这些值：

```
void gtk_selection_data_set( GtkSelectionData *selection_data,
                             GdkAtom           type,
                             gint              format,
                             guchar            *data,
                             gint              length );
```

这个函数会适当地将数据复制一份这样你就不必为保留这些数据操心了。(不要手工填充 GtkSelectionData 结构里的值。)

用户做了某个操作后，你可以通过下面的函数声称选中区的所有权。

```
gboolean gtk_selection_owner_set( GtkWidget *widget,
                                  GdkAtom    selection,
                                  guint32    time );
```

如果其它应用程序要求得到选中区,你将接收到"selection_clear_event"信号。

下面是一个提供选中区的示例，它给开关按钮加了选中区功能。当开关按钮被按下，程序要求得到主选中区。它只支持"STRING"目标(除了 GTK 自身已支持的"TARGETS"等目标)。当这个目标被要求时，一个描叙时间的字符串被返回。

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

GtkWidget *selection_button;
GtkWidget *selection_widget;

/* 当用户触发选中区时的回调 */
void selection_toggled( GtkWidget *widget,
                        gint      *have_selection )
{
  if (GTK_TOGGLE_BUTTON (widget)->active)
    {
```

```
      *have_selection = gtk_selection_owner_set (selection_widget,
                                                 GDK_SELECTION_PRIMARY,
                                                 GDK_CURRENT_TIME);
      /* 如果声称选中区失败，则使按钮返回未选中状态 */
      if (!*have_selection)
        gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (widget), FALSE);
    }
  else
    {
      if (*have_selection)
        {
          /* 在设置所有者为 NULL 来清空选中区前，
           * 先检测自己是不是真正的所有者 */
          if (gdk_selection_owner_get (GDK_SELECTION_PRIMARY) == widget->window)
            gtk_selection_owner_set (NULL, GDK_SELECTION_PRIMARY,
                                     GDK_CURRENT_TIME);
          *have_selection = FALSE;
        }
    }
}

/* 当其它应用程序声称选中区时调用 */
gint selection_clear( GtkWidget         *widget,
                      GdkEventSelection *event,
                      gint              *have_selection )
{
  *have_selection = FALSE;
  gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (selection_button), FALSE);

  return TRUE;
}

/* 将当前时间作为选中区内容提供。 */
void selection_handle( GtkWidget       *widget,
                       GtkSelectionData *selection_data,
                       guint            info,
                       guint            time_stamp,
                       gpointer         data )
{
  gchar *timestr;
  time_t current_time;

  current_time = time (NULL);
  timestr = asctime (localtime (&current_time));
  /* 当我们返回单独一个字符串时，它不必以 NULL 结尾。
     它将被自动完成 */

  gtk_selection_data_set (selection_data, GDK_SELECTION_TYPE_STRING,
                          8, timestr, strlen (timestr));
}

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

  static int have_selection = FALSE;

  gtk_init (&argc, &argv);

  /* 创建顶级窗口 */

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  gtk_window_set_title (GTK_WINDOW (window), "Event Box");
  gtk_container_set_border_width (GTK_CONTAINER (window), 10);

  g_signal_connect (G_OBJECT (window), "destroy",
```

```
                        G_CALLBACK (exit), NULL);

  /* 创建一个开关按钮作为选中区 */

  selection_widget = gtk_invisible_new ();
  selection_button = gtk_toggle_button_new_with_label ("Claim Selection");
  gtk_container_add (GTK_CONTAINER (window), selection_button);
  gtk_widget_show (selection_button);

  g_signal_connect (G_OBJECT (selection_button), "toggled",
                    G_CALLBACK (selection_toggled), &have_selection);
  g_signal_connect (G_OBJECT (selection_widget), "selection_clear_event",
                    G_CALLBACK (selection_clear), &have_selection);

  gtk_selection_add_target (selection_widget,
                            GDK_SELECTION_PRIMARY,
                            GDK_SELECTION_TYPE_STRING,
                            1);
  g_signal_connect (G_OBJECT (selection_widget), "selection_get",
                    G_CALLBACK (selection_handle), &have_selection);

  gtk_widget_show (selection_button);
  gtk_widget_show (window);

  gtk_main ();

  return 0;
}
```

拖放

GTK+ 有一套高级的用来支持图形程序之间通过拖放系统传递信息的函数。GTK+ 能在低级的 Xdnd 和 Motif 拖放协议基础上实施拖放。

概述

一个支持 GTK+ 拖放的应用程序先要定义和设置能拖放的构件。每个构件都可以是拖放的来源端和/或目标端。注意这些构件必须有一个关联的 X 窗口，用 GTK_WIDGET_NO_WINDOW(widget) 检测。

源构件可以发送拖动数据，因此允许用户把东西从它们上面拖出去，同时目标构件能接收拖动数据。拖放的目标端能限制它们接受谁的拖动数据，比如，同一应用程序或任意应用程序(包括它自己)。

发 送和接收拖动数据是利用 GTK+ 信号来进行的。投放一个项到一个目的构件需要一个数据请求(源构件使用)和数据接收信号处理函数(目的构件使用)。还可以连接一些附加的信号处理函数，如 果你想要知道何时拖动开始(在拖动开始的最初阶段)，何时投放成功，以及何时整个拖放过程完成(是否成功)。

当接收到请求时，你的应用程序将需要为源构件提供数据，还包括一个拖动数据请求信号处理函数。而对目的构件将需要一个拖动数据接收信号处理函数。

一个典型的拖放周期将像下面这样:

1. 拖动开始。
2. 拖动数据请求(当投放发生)。
3. 拖动数据被接收(可能在同一个应用程序内部或不同应用程序之间)。
4. 拖动数据被删除(如果拖动是移动)。
5. 拖放过程完成。

在它们之间也有一些次要的步骤，但这些我们等下再作详细介绍。

属性

拖动数据有下面的这些属性:

- 拖动行为类型(比如 GDK_ACTION_COPY, GDK_ACTION_MOVE)。
- 客户端指定的任意拖放类型(一个名称和数字组成的对)。
- 发送和接收的数据格式类型。

拖 动行为的意义很明显，它们指定构件是否能以指定的行为来拖动，比如 GDK_ACTION_COPY 和/或 GDK_ACTION_MOVE。GDK_ACTION_COPY 将是一个典型的拖放，它的源数据并不会被删除，而 GDK_ACTION_MOVE 将和 GDK_ACTION_COPY 一样，只是“建议”在接收信号处理函数调用之后将源数据删除掉。还有一些附加的拖动行为包括 GDK_ACTION_LINK 等，当你对拖放的使用达到更高水平时，你可能会使用到。

客户端指定的任意拖放类型要更加灵活些，因为你的程序将明确地定义和检测它。你需要指定一个名称和/或数字以设置目的构件能接收的特定拖放类型。用一个名称要可靠些，因为其它应用程序可能也正好用了同样的数字来区别它的完全不同的类型。

发送和接收的数据格式类型(*selection target*)只在请求和接收数据处理函数中使用。术语 *selection target* 有些容易令人误解。它应该是一个适合在 GTK+ 选中区(剪切/复制和粘贴)中使用的术语。*selection target* 实际是指被发送和接收的数据的格式类型(比如 GdkAtom, integer, 或 string)。你的请求数据处理函数必须指定它发送出去的数据的类型(*selection target*)，同时接收数据处理函数要处理接收到的数据的类型(*selection target*)。

函数

设置源构件

gtk_drag_source_set()函数指定一套在构件上拖动操作时的目标类型。

```
void gtk_drag_source_set( GtkWidget            *widget,
                          GdkModifierType      start_button_mask,
                          const GtkTargetEntry *targets,
                          gint                 n_targets,
                          GdkDragAction        actions );
```

这些参数的作用如下：

- widget 指定拖动源构件
- start_button_mask 指定能引发拖动操作的鼠标按键的位掩码(bitmask)，比如 GDK_BUTTON1_MASK
- targets 指定一个此拖动支持的目标数据类型的表格
- n_targets 指定上面的目标的数量
- actions 指定从这个窗口进行拖动时可能的行为的位掩码

targets 参数是一个由下面的结构组成的数组：

```
struct GtkTargetEntry {
    gchar *target;
    guint  flags;
    guint  info;
 };
```

这一结构指定一个代表拖动类型的字符串，可选的标记和应用程序指派的整数标识符。

如果一个构件不必再担当拖放操作的源，用 gtk_drag_source_unset()函数可以删除一套拖放目标类型。

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

源构件上的信号

在一个拖放操作中源构件将被发送下面的这些信号。

Table 1. 源构件的信号

drag_begin	void (*drag_begin)(GtkWidget *widget, GdkDragContext *dc, gpointer data)

drag_motion	gboolean (*drag_motion)(GtkWidget *widget, GdkDragContext *dc, gint x, gint y, guint t, gpointer data)
drag_data_get	void (*drag_data_get)(GtkWidget *widget, GdkDragContext *dc, GtkSelectionData *selection_data, guint info, guint t, gpointer data)
drag_data_delete	void (*drag_data_delete)(GtkWidget *widget, GdkDragContext *dc, gpointer data)
drag_drop	gboolean (*drag_drop)(GtkWidget *widget, GdkDragContext *dc, gint x, gint y, guint t, gpointer data)
drag_end	void (*drag_end)(GtkWidget *widget, GdkDragContext *dc, gpointer data)

设置目的构件

gtk_drag_dest_set()指定这个构件可以接收拖动并指定它能接收的拖动的类型。

gtk_drag_dest_unset()指定构件不再能接收拖动了。

```
void gtk_drag_dest_set( GtkWidget           *widget,
                        GtkDestDefaults      flags,
                        const GtkTargetEntry *targets,
                        gint                 n_targets,
                        GdkDragAction        actions );

void gtk_drag_dest_unset( GtkWidget *widget );
```

目的构件上的信号

在一个拖放操作中目的构件将被发送下面的这些信号。

Table 2. 目的构件的信号

drag_data_received	void (*drag_data_received)(GtkWidget *widget, GdkDragContext *dc, gint x, gint y, GtkSelectionData *selection_data, guint info, guint t, gpointer data)

GLib

Glib 是一种底层库，创建 GDK 和 GTK 应用程序时该库可提供许多有用的定义和函数。它们包括基本类型及限制的定义、标准宏、类型转换、字节序、存储分配、警告和断言、消息记录、计时器、字符串 工具 (string utilities)、挂钩 (hook) 函数、一个句法扫描器、动态加载模块和自动字符串补全，它也定义了许多数据结构(及其相应的操作)，包括存储块、双向链表、单向链表、哈希表、串(动态增 长)、串块(串的组)、数组(大小可随元素的加入而增长)、平衡二叉树、N 叉树、夸克 (quark，一种字符串和其唯一的整数标识之间的双向关联)、键数据列表(可由字符串或整数标识访问的数据元素列表)、关系和元组(可通过任一位置号索 引的数据表格)以及缓存 (cache)。

下面将总结 Glib 库函数的一些功能，没有包括所有函数，数据结构或操作。有关 Glib 库的更完整的信息请看 Glib 文档。你可以从 http://www.gtk.org/ 得到。

如果使用其它语言，应查看该语言的绑定的文档，有时该语言可能已经内建了相同的功能，有时可能没有。

定义

许多标准类型的极值定义是:

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

G_MAXLONG

下面的 typedefs 也是定义。余下未定义的则根据硬件平台的不同而动态设置。请记住如果要想使程序具有可移植性就不要计算指针的大小。一个指针在 Alpha 上是 8 个字节，而在 Intel 80x86 系列 cpu 上是 4 个字节。

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

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

float   gfloat;
double  gdouble;
long double gldouble;

void* gpointer;

gint8
guint8
gint16
guint16
gint32
guint32
```

双向链表

以 下的函数用于创建、管理和销毁标准双向链表。链表中每个元素都包含一块数据和指向前后元素的指针。这使得通过链表的双向移动变的容易。数据项的类型是 "gpointer"，意指数据可为一指向实际数据的指针或 (通过类型转换) 为一数值 (但不要设想 int 和 gpointer 有相同的大小!)。这些函数在内部按块为链表元素分配空间，这比单为每个元素分配空间更有效率。

不存在专用于创建链表的函数。而是简单地创建一个类型为 Glist* 的变量，并把它的值设置为 NULL；NULL 被当作空表。

向 链表中加入一个新元素，使用函数 g_list_append()、g_list_prepend()、g_list_insert() 或 g_list_insert_sorted() 。无论如何，函数都接收一个指向表头的指针作为参数，并返回一个指向表头的指针(可能和接收的指针不同)。因此，对所有添加或撤除链表元素的操作，一定要 保存返回值!

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

此函数把一个新元素(具有值 data)加到链表尾。

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

此函数把一个新元素(具有值 data)加到链表头。

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

此函数插入一个新元素(具有值 data)到链表中指定位置，如果位置是 0，它和 g_list_prepend() 函数作用相同，如果位置，它和 g_list_append() 函数作用相同。

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

此函数从表中移除一个具有值 data 的元素，如果该元素不存在，链表不变。

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

此函数释放由 GIist 使用的所有存储区，如果表元素空间是通过动态分配的，则应首先被释放。

还有许多其它支持双向链表的 Glib 函数;查看文档获得更多的信息。这儿列出几个更有用的函数的声明:

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

GList *g_list_reverse( GList *list );

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

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

GList *g_list_last( GList *list );

GList *g_list_first( GList *list );

gint g_list_length( GList *list );

void g_list_foreach( GList    *list,
                     GFunc     func,
                     gpointer  user_data );
```

单向链表

以上的许多函数用于单向链表是一样的。下面是链表操作函数的部分列表:

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

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

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

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

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

GSList *g_slist_reverse( GSList *list );

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

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

GSList *g_slist_last( GSList *list );

gint g_slist_length( GSList *list );

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

存储管理

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

这是 malloc() 函数的替代函数，不需要检查返回值，因为此函数已替你做这件事了。如果存储分配因任何原因失败，应用程序将被终止。

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

和上一函数相同，但在返回指向所分配存储块的指针之前，将该存储块清 0。

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

重新分配由 mem 开始，大小为 size 字节的存储块。明显地，该存储块先前已被分配。

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

释放分配的存储块。这很简单。如果 mem 为 NULL，则直接返回。

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

把用过的存储块的内容转储到一个文件中。但要这样做，需要将#define MEM_PROFILE 加到文件 glib/gmem.c 的开始处，然后重新运行命令 make 和 make install。

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

检查存储位置的有效性。需要将#define MEM_CHECK 加到文件 glib/gmem.c 的开始处，然后重新运行命令 make 和 make install。

计时器

计 时器函数可用于对操作计时(即查看操作所耗费的时间)。首先用 g_timer_new() 函数建立一个新计时器，然后用 g_timer_start() 函数启动计时，用 gh_timer_stop() 停止计时，用 g_timer_elapsed() 函数决定所耗的时间。

```
GTimer *g_timer_new( void );

void g_timer_destroy( GTimer *timer );

void g_timer_start( GTimer  *timer );

void g_timer_stop( GTimer  *timer );

void g_timer_reset( GTimer  *timer );

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

字符串处理

GLib 定义了一个叫做 GString 的新类型，该类型相似于标准 C 中的 string，但可以自动增长。字符串数据以 null 结尾。该类型可以防止程序中的缓冲区溢出错误。这是一个非常重要的特性，因此我推荐使用 GString。GString 有一简单定义：

```
struct GString
{
  gchar *str; /* 指向当前以‘\0’结尾的字符串。 */
  gint len; /* 当前长度 */
};
```

如你所预想的，有许多对一个 GString 型字符串的操作。

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

这个函数构造一个 GString 型字符串，它把 init 指向的字符串值复制到 GString 型字符串中，返回一个指向它的指针。建一个初始为空的 GString 型字符串则传递一个 NULL 作为参数。

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

该函数释放给定的 GString 型字符串的存储空间。如果参数 free_segment 为 TRUE，也释放其字符数据。

```
GString *g_string_assign( GString     *lval,
                          const gchar *rval );
```

该函数将 rval 中的字符复制到 lval 中，覆盖 lval 中以前的内容。注意为装下复制的字符串，lval 会变长，这是和标准的 strcpy() 数不一样的。

以下函数的功能是相当明显的(带有_c 的版本接受一个字符而不是一个字符串):

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

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

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

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

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

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

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

实用程序和错误处理函数

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

替代 strdup 函数。把原字符串内容复制到新分配的存储块中，返回指向它的指针。

```
gchar *g_strerror( gint errnum );
```

我推荐使用此函数处理所有错误信息，它比 perror() 和其它类似函数更好，更具可移植性。此函数的输出通常为如下格式:

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

这里有一个在我们的 hello_world 程序中调用此函数的示例:

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

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

打印错误信息。格式同于 printf，但在错误信息前加上了"** ERROR **"，并且退出程序。仅用在致命错误上。

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

和前一函数功能相同，只是错误信息前是 " ** WARNING ** "，且不退出程序。

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

在传递的字符串前打印 "message:"

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

替代 printf() 函数。

本章最后一个函数:

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

对所给信号的号码打印出相应的信号名称。在通用信号处理函数中有用。

以上所有的函数体或多或少都是从 glib.h 中获得的，任何人如关注某一函数的文档说明，只要给我发一封电子邮件。

GTK 的 rc 文件

GTK 有自己缺省处理应用程序的方法，这就是使用 rc 配置文件。这些文件可用于给几乎任何构件设置颜色，也能为一些构件的背景贴上一幅像素图。

rc 文件的功能

在你的应用程序开始处，应包含一个如下的函数调用:

```
void gtk_rc_parse( char *filename );
```

把 rc 文件名传递给被调用的函数，随后 GTK 会解析这个文件，并且使用文件中所定义构件类型的风格设置。

如果希望定义一套和其它构件集或同一构件集中其它逻辑部分具有不同风格的特定构件集，使用以下函数调用:

```
void gtk_widget_set_name( GtkWidget *widget,
                          gchar     *name );
```

把新创建的构件作为第一个参数，把你给该构件定的名程作为第二个参数。这使你能够在 rc 文件中按名称更改这个构件的属性。

如果我们使用了一个如下的函数调用:

```
button = gtk_button_new_with_label ("Special Button");
gtk_widget_set_name (button,"special button");
```

那么这个按钮取名为"special button"，并且在 rc 文件中可能通过像"special button.GtkButton"这样的名称找到它。
[<--- Verify ME!]

后面作为示例的 rc 文件，设置主窗口的属性，告诉所有子窗口继承在"main button"风格项中规定的风格。在应用程序中的代码为:

```
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_widget_set_name (window, "main window");
```

然后在 rc 文件中用如下一行定义它的风格:

```
widget "main window.*GtkButton*" style "main_button"
```

该定义把"主窗口"中的所有按钮构件设置为 rc 文件中定义的"main_buttons"风格。

如你所见，这是一个非常强大和灵活的系统。如何最大限度的利用它的优势需要使用你的想象力。

GTK rc 文件的格式

以下示例说明了 rc 文件的格式。它取自 GTK 发行包中名为 testgtkrc 的文件中，但我增加了一些注解和内容。你也许希望在自己的应用程序中包含这些注解以有助于用户在使用过程中进行小调整。

有几个指令用于改变一个构件的属性。

- fg - 设置构件的背景色。
- bg - 设置构件的前景色。
- bg_pixmap - 设置构件的背景为一幅像素图。
- font - 设置构件使用的字体。

另外，一个构件可以有几种状态，你可以为每种状态设置不同的颜色，图像和字体。这些状态是:

- NORMAL - 构件的一般状态，如鼠标不位于构件上方，键未按下时构件所处的状态。
- PRELIGHT - 当鼠标移到构件上方，在此状态下定义的颜色生效。
- ACTIVE - 当处于构件上的鼠标键被按下或点击时，该构件被激活，相应的属性值将生效。
- INSENSITIVE - 当构件被设置为不敏感(insensitive)时，它不能被激活，相应属性值起作用。
- SELECTED - 选中一个对象时，取相应的属性值。

当用"fg"和"bg"关键字设置构件的颜色时，其格式是:

```
fg[<STATE>] = { Red, Green, Blue }
```

这 里 STATE 是前述状态(PRELIGHT,ACTIVE,...)之一,Red,Green,Blue 是范围 0-1.0 间的数值, {1.0,1.0,1.0}表示白色。这些数值必须是浮点型的，否则将被存为 0，因此写为"1"的数值是无效的，应写为"1.0".0 值写为"0"却不 错，因为即使系统认不出也没关系。所有系统不认识的数值都会被设为 0。

bg_pixmap 除了用一个文件名取代颜色名外和以上格式很相似。

pixmap_path 是一个由‘:’分隔开的路径表。当搜索你定义的图像文件时选择这些路径。

字体命令简单的就是:

```
font = "<font name>"
```

唯一困难的是描述字体的字符串部分。用 xfontsel 或类似工具会有帮助。

"widget_class" 设置一类构件的风格。这些类在构件概述中的类组织里列出。

"widget" 指令把一个特别命名的构件集设置为一个要求的风格，覆盖所有已有的风格。这些构件使用 _widget_set_name() 函数调用在应用程序中注册。这样你可以设置任一组构件的属性，而不是设置整个构件类的属性。我请求你为这些特定名称的构件写好文档，用户可能要定制它们。

当使用关键字 parent 作为属性值时，这个构件将继承它的父构件的属性。

定义一个风格时，可能会将以前定义风格的属性值分配给新定义的风格。

```
style "main_button" = "button"
{
  font = "-adobe-helvetica-medium-r-normal--*-100-*-*-*-*-*-*"
  bg[PRELIGHT] = { 0.75, 0, 0 }
}
```

这个示例用"button 构件的风格创建了一个新的 "main_button" 构件的风格，只改变 "button" 构件风格中的 font 属性和 prelight 背景色属性。

当然，许多属性不能应用在所有构件上，按常识这是不难分辨的。而任何能在一个构件上起作用的东西，都应能作为它的属性。

rc 文件示例

```
# pixmap_path "<dir 1>:<dir 2>:<dir 3>:..."
#
pixmap_path "/usr/include/X11R6/pixmaps:/home/imain/pixmaps"
#
# style <name> [= <name>]
```

```
# {
#   <option>
# }
#
# widget <widget_set> style <style_name>
# widget_class <widget_class_set> style <style_name>

# 这里列出所有状态。注意有些状态不能用在一定的构件上。
#
# NORMAL -构件的一般状态，如鼠标不位于构件上方，键未按下时构件所处的状态。
#
# PRELIGHT - 当鼠标移到构件上方，在此状态下定义的颜色生效。
#
#
# ACTIVE -当处于构件上的鼠标键被按下或点击时，该构件被激活，相应的属性值将生效。
#
#
# INSENSITIVE -当构件被设置为不敏感 (insensitive) 时，它不能被激活，相应属性值起作用。
#
#
# SELECTED -选中一个对象时，取相应的属性值。
#
# 给定了这些状态，我们就能使用以下指令设置在这些状态下构件的属性。
#
# fg - 设置构件的前景色。
# fg - 设置构件的背景色。
# bg_pixmap - 设置构件的背景为一幅象素图。
# font - 设置给定构件所用的字体。
#

# 本例设置一种名为"button"的风格。这个名称实际不重要，因为在本文件的后面这个风格都分
# 配给了实际的构件。

style "window"
{
  #此处设置窗口背景为指定的像素图。
  #bg_pixmap[<STATE>] = "<pixmap filename>"
  bg_pixmap[NORMAL] = "warning.xpm"
}

style "scale"
{
  #设置"NORMAL"状态下前景色(字体颜色)为红色。

  fg[NORMAL] = { 1.0, 0, 0 }

  #设置此构件的背景像素图为其父构件的背景像素图。
  bg_pixmap[NORMAL] = "<parent>"
}

style "button"
{
  # 显示一个按钮的所有可能状态，唯一未用的状态是 SELECTED。

  fg[PRELIGHT] = { 0, 1.0, 1.0 }
  bg[PRELIGHT] = { 0, 0, 1.0 }
  bg[ACTIVE] = { 1.0, 0, 0 }
  fg[ACTIVE] = { 0, 1.0, 0 }
  bg[NORMAL] = { 1.0, 1.0, 0 }
  fg[NORMAL] = { .99, 0, .99 }
  bg[INSENSITIVE] = { 1.0, 1.0, 1.0 }
  fg[INSENSITIVE] = { 1.0, 0, 1.0 }
}
```

```
# 在本例中，我们继承"button"风格的属性，然后重设 PRELIGHT 状态下的字体和背景色以创建一个新的"main_button"风格。

style "main_button" = "button"
{
  font = "-adobe-helvetica-medium-r-normal--*-100-*-*-*-*-*-*"
  bg[PRELIGHT] = { 0.75, 0, 0 }
}

style "toggle_button" = "button"
{
  fg[NORMAL] = { 1.0, 0, 0 }
  fg[ACTIVE] = { 1.0, 0, 0 }

  # 这里设置 toggle_button 的背景像素图为其父构件的像素图(在应用程序中已定义)。
  bg_pixmap[NORMAL] = "<parent>"
}

style "text"
{
  bg_pixmap[NORMAL] = "marble.xpm"
  fg[NORMAL] = { 1.0, 1.0, 1.0 }
}

style "ruler"
{
  font = "-adobe-helvetica-medium-r-normal--*-80-*-*-*-*-*-*"
}

# pixmap_path "~/.pixmaps"

# 下面设置使用以上所定义风格的构件类型。
# 构件类型是在类的组织中列出的，但是恰有可能在本文档中列出供用户参考。

widget_class "GtkWindow" style "window"
widget_class "GtkDialog" style "window"
widget_class "GtkFileSelection" style "window"
widget_class "*Gtk*Scale" style "scale"
widget_class "*GtkCheckButton*" style "toggle_button"
widget_class "*GtkRadioButton*" style "toggle_button"
widget_class "*GtkButton*" style "button"
widget_class "*Ruler" style "ruler"
widget_class "*GtkText" style "text"

# 设置作为“main windows”的子构件的所有按钮构件为 main_button 风格。
# 这些(专门命名的构件）都必须附有文档说明让用户更好地使用它们。
widget "main window.*GtkButton*" style "main_button"
```

编写你自己的构件

概述

虽 然随 GTK 提供了许多类型的构件，这些构件也基本能满足需要，但有时你仍然需要创建自己的构件类型。因为 GTK 完全应用了构件继承，并且有接近你的需求的构件，通常只要几行代码就可以编写一个新的构件类型。但是在开始之前，确信没有人编写了一个同样的构件。这样可 以避免重复劳动，使 GTK 构件保持最小的数目，且有助于保持代码和不同应用程序接口的一致性。另一方面，一旦你做完了自己的构件，告知大家，其他人会因此而受益。最好的发布地点可 能是 gtk 邮件列表。

你可以从教程中获得示例构件的完整源代码，或从:

http://www.gtk.org/~otaylor/gtk/tutorial/

一个构件的剖析

要想创建一个新的构件，最重要的是要对 GTK 对象的工作原理有所了解。这一节只是一个简述，详见参考手册。

GTK 构件具有面向对象的特性。然而，它是用标准的 C 实现的。这极大的改善了在当前 C++ 编译器上使用的可移植性和稳定性；但是，这也意味着写构件的人必须注意一些实现的细节。一个构件类的所有实例(比如所有的按钮构件)的共有信息存储在*类结构*里，类的信号信息只以该结构存储了一份(充当 C 中的虚函数)。为了支持继承，类结构的第一个域必须是它的父类结构的一个拷贝。GtkButton 的类结构声明如下：

```
struct _GtkButtonClass
{
  GtkContainerClass parent_class;

  void (* pressed)  (GtkButton *button);
  void (* released) (GtkButton *button);
  void (* clicked)  (GtkButton *button);
  void (* enter)    (GtkButton *button);
  void (* leave)    (GtkButton *button);
};
```

当将一个按钮视为容器时(例如，当调整它的大小时)，它的类结构被转换为 GtkContainerClass，并且相应的域用于处理信号。

每一个构件也有一个结构，它是创建每个实例的基础。该结构有为每个构件的实例存储不同信息的域。我们把该结构称为*对象结构*。如下是按钮类：

```
struct _GtkButton
{
  GtkContainer container;

  GtkWidget *child;

  guint in_button : 1;
  guint button_down : 1;
};
```

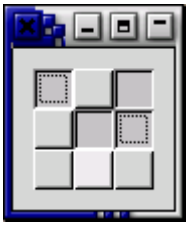
请注意，它同类结构相似，第一个域是父类的对象结构，因此该结构在需要时可以转换为父类的对象结构。

创建一个复合构件

介绍

你可能乐于创建这样一种类型的构件，它仅仅是其它 GTK 构件的一个组合。为组装用户界面元素以重复使用提供了便利的方法。在标准发布中的文件选择和颜色选择构件就是这种类型构件的示例。

我们将要在这一节创建一个井字游戏构件，一个 3X3 的开关按钮矩阵，当同一列、同一行或是对角线的所有三个按钮都被按下时触发一个信号。



选择一个父类

通常，一个复合构件的父类是一个容纳复合构件的所有元素的容器类。例如，文件选择构件的父类就是对话框类。因为我们的按钮排列在一个表中，看起来应该让表类作为我们的父类。但不幸的是，这样无法工作。构件的创建分为两个函数 - 一个用户调用的 WIDGETNAME_new() 函数，另一个是 WIDGETNAME_init() 函数作基本的初始化构件的工作，它不使用传递给 _new() 函数的参数。子构件只调用父构件的 _init 函数。但是这个分工不能在表中正常工作，因为创建表时需要知道表的行数和列数。除非我们想重新实现 gtk_table_new() 的大多数功能,我们最好避免从表派生构件。由于这个原因，代替表，我们从纵向盒派生构件，然后把表放入纵向盒。

头文件

每个构件类有一个头文件，该头文件用于声明构件的对象、类结构和公共函数。有两个特性是值得指出的。为避免重复定

义，我们把整个头文件放入如下语句里：

```
#ifndef __TICTACTOE_H__
#define __TICTACTOE_H__
.
.
.
#endif /* __TICTACTOE_H__ */
```

并且让 C++ 程序也能包含该头文件：

```
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
.
.
.
#ifdef __cplusplus
}
#endif /* __cplusplus */
```

在我们的头文件里，和函数、结构一起声明的还有三个标准宏。 TICTACTOE(obj), TICTACTOE_CLASS(class) 和 IS_TICTACTOE(obj)，这三个宏分别是将指针转换为指向对象、类的指针和检测一个对象是否是一个井字游戏构件。

下面是全部的头文件：

```
/* GTK - GIMP 工具包
 * 版权 (C) 1995-1997 Peter Mattis, Spencer Kimball 和 Josh MacDonald 所有
 *
 * 本程序是自由软件。你可以在自由软件基金发布的 GNU GPL 的条款下重新分发
 * 或修改它。GPL 可以使用版本 2 或(由你选择)任何随后的版本。
 *
 * 本程序分发的目的是它可能对其他人有用，但不提供任何的担保，包括隐含的
 * 和适合特定用途的保证。请查阅 GNU 通用公共许可证获得详细的信息。
 *
 * 你应该已经随该软件一起收到一份 GNU 通用公共许可。如果还没有，请写信给
 * Free Software Foundation, Inc., 59 Temple Place - Suite 330,
 * Boston, MA 02111-1307, USA.
 */

#ifndef __TICTACTOE_H__
#define __TICTACTOE_H__


#include <gdk/gdk.h>
#include <gtk/gtkvbox.h>


#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */

#define TICTACTOE(obj)          GTK_CHECK_CAST (obj, tictactoe_get_type (), Tictactoe)
#define TICTACTOE_CLASS(klass)  GTK_CHECK_CLASS_CAST (klass, tictactoe_get_type (), TictactoeClass)
#define IS_TICTACTOE(obj)       GTK_CHECK_TYPE (obj, tictactoe_get_type ())


typedef struct _Tictactoe       Tictactoe;
typedef struct _TictactoeClass  TictactoeClass;

struct _Tictactoe
{
  GtkVBox vbox;

  GtkWidget *buttons[3][3];
};
```

```
struct _TictactoeClass
{
  GtkVBoxClass parent_class;

  void (* tictactoe) (Tictactoe *ttt);
};

GtkType        tictactoe_get_type        (void);
GtkWidget*     tictactoe_new             (void);
void           tictactoe_clear           (Tictactoe *ttt);

#ifdef __cplusplus
}
#endif /* __cplusplus */

#endif /* __TICTACTOE_H__ */
```

_get_type() 函数

现在，我们继续实现我们的构件。WIDGETNAME_get_type() 函数是每个构件的核心函数。当第一次调用时，该函数告之 GTK 这个构件类并得到一个能唯一识别该构件类的 ID。之后的调用，只返回这个 ID。

```
GtkType
tictactoe_get_type ()
{
  static guint ttt_type = 0;

  if (!ttt_type)
    {
      GtkTypeInfo ttt_info =
      {
        "Tictactoe",
        sizeof (Tictactoe),
        sizeof (TictactoeClass),
        (GtkClassInitFunc) tictactoe_class_init,
        (GtkObjectInitFunc) tictactoe_init,
        (GtkArgSetFunc) NULL,
        (GtkArgGetFunc) NULL
      };

      ttt_type = gtk_type_unique (gtk_vbox_get_type (), &ttt_info);
    }

  return ttt_type;
}
```

GtkTypeInfo 结构定义如下：

```
struct _GtkTypeInfo
{
  gchar *type_name;
  guint object_size;
  guint class_size;
  GtkClassInitFunc class_init_func;
  GtkObjectInitFunc object_init_func;
  GtkArgSetFunc arg_set_func;
  GtkArgGetFunc arg_get_func;
};
```

这个结构的域很形象。我们在这里忽略 arg_set_func 和 arg_get_func 这两个域：它们很重要，但大部分还未实现，它们允许解释性语言方便地设置构件的属性。一旦 GTK 正确的填充了该结构，它就知道了如何去创建一个特殊构件类型的对象。

_class_init() 函数

WIDGETNAME_class_init()函数初始化构件类结构的域，并为类设置任何信号。我们的井字游戏构件是这样的：

```
enum {
  TICTACTOE_SIGNAL,
  LAST_SIGNAL
};


static gint tictactoe_signals[LAST_SIGNAL] = { 0 };

static void
tictactoe_class_init (TictactoeClass *class)
{
  GtkObjectClass *object_class;

  object_class = (GtkObjectClass*) class;

  tictactoe_signals[TICTACTOE_SIGNAL] = gtk_signal_new ("tictactoe",
                                                        GTK_RUN_FIRST,
                                                        object_class->type,
                                                        GTK_SIGNAL_OFFSET (TictactoeClass, tictactoe),
                                                        gtk_signal_default_marshaller, GTK_TYPE_NONE, 0);


  gtk_object_class_add_signals (object_class, tictactoe_signals, LAST_SIGNAL);

  class->tictactoe = NULL;
}
```

我们的构件只有一个 tictactoe 信号，当连成一行、一列或一个对角线时，该信号被触发。并不是每个复合构件都需要信号，因此当你第一次阅读这节时，可以跳过看下一节，因为这一节对初学者有点难。

函数：

```
gint gtk_signal_new( const gchar        *name,
                     GtkSignalRunType    run_type,
                     GtkType             object_type,
                     gint                function_offset,
                     GtkSignalMarshaller  marshaller,
                     GtkType             return_val,
                     guint               nparams,
                     ...);
```

创建一个新信号。参数是：

- name：信号的名称。
- run_type：设定是在用户处理函数之前还是之后运行默认处理函数。该值通常会是 GTK_RUN_FIRST 或 GTK_RUN_LAST，虽然也有其它值。
- object_type：发出该信号的对象的 ID。(也可能是子对象的 ID。)
- function_offset：类结构指针相对于默认处理函数的偏移量。
- marshaller：一个用于调用信号处理函数的函数。对于除了发出信号的对象和用户数据外没有其它的参数的信号处理函数，我们可以使用事先提供的 marshaller 函数 gtk_signal_default_marshaller。
- return_val：返回值的类型。
- nparams：信号处理函数的参数个数 (除了前面提到的"发出信号的对象"和"用户数据"这两个默认参数)
- ...：参数的数据类型。

当指定类型时，要用到 GtkType 枚举类型：

```
typedef enum
{
  GTK_TYPE_INVALID,
```

```
  GTK_TYPE_NONE,
  GTK_TYPE_CHAR,
  GTK_TYPE_BOOL,
  GTK_TYPE_INT,
  GTK_TYPE_UINT,
  GTK_TYPE_LONG,
  GTK_TYPE_ULONG,
  GTK_TYPE_FLOAT,
  GTK_TYPE_DOUBLE,
  GTK_TYPE_STRING,
  GTK_TYPE_ENUM,
  GTK_TYPE_FLAGS,
  GTK_TYPE_BOXED,
  GTK_TYPE_FOREIGN,
  GTK_TYPE_CALLBACK,
  GTK_TYPE_ARGS,

  GTK_TYPE_POINTER,

  /* 如果下面两个最终能被删除就好了。*/
  GTK_TYPE_SIGNAL,
  GTK_TYPE_C_CALLBACK,

  GTK_TYPE_OBJECT

} GtkFundamentalType;
```

gtk_signal_new() 返回一个能识别信号的唯一整数，我们把它存储在 tictactoe_signals 数组里，并用枚举值来做索引。(依照惯例，枚举成员是信号的名称，且是大写的，但在这里会与 TICTACTOE() 宏冲突，因此我们用 TICTACTOE_SIGNAL 代替。)

创建了信号之后，我们需要让 GTK 将我们的信号和 Tictactoe 类联系起来。调用 gtk_object_class_add_signals() 函数可以做这件事。然后我们把指向 "tictactoe" 信号默认处理函数的指针设为 NULL，表明没有默认的动作。

_init() 函数

每个构件类也需要一个初始化对象结构的函数。通常，该函数有个相当有限的任务，就是设置结构成员为缺省值。对于复合构件，这个函数还创建成分(component)构件。

```
static void
tictactoe_init (Tictactoe *ttt)
{
  GtkWidget *table;
  gint i,j;

  table = gtk_table_new (3, 3, TRUE);
  gtk_container_add (GTK_CONTAINER(ttt), table);
  gtk_widget_show (table);

  for (i=0;i<3; i++)
    for (j=0;j<3; j++)
      {
        ttt->buttons[i][j] = gtk_toggle_button_new ();
        gtk_table_attach_defaults (GTK_TABLE(table), ttt->buttons[i][j],
                                   i, i+1, j, j+1);
        gtk_signal_connect (GTK_OBJECT (ttt->buttons[i][j]), "toggled",
                            GTK_SIGNAL_FUNC (tictactoe_toggle), ttt);
        gtk_widget_set_size_request (ttt->buttons[i][j], 20, 20);
        gtk_widget_show (ttt->buttons[i][j]);
      }
}
```

其余的...

另外还有一个函数每个构件(除了基本的构件类型，如 Bin 不能实例化)都需要 - 被用户调用以创建一个该类型的对象的函

数。这个调用通常是 WIDGETNAME_new()。在有些构件里，该函数获得几个参数，依据这几个参数做一些设置，我们的井字游戏构件没有这样做。另外两个函数是针对井字游戏构件的。

tictactoe_clear() 是一个公共函数，它复位构件中的按钮。注意 gtk_signal_handler_block_by_data() 函数用来防止按钮切换信号处理函数被不必要地触发。

tictactoe_toggle() 是当用户点击按钮时调用的信号处理函数。它判断开关按钮中是否出现了导致赢的组合，如果是，则发出 "tictactoe" 信号。

```
GtkWidget*
tictactoe_new ()
{
  return GTK_WIDGET ( gtk_type_new (tictactoe_get_type ()));
}

void
tictactoe_clear (Tictactoe *ttt)
{
  int i,j;

  for (i=0;i<3;i++)
    for (j=0;j<3;j++)
      {
        gtk_signal_handler_block_by_data (GTK_OBJECT(ttt->buttons[i][j]), ttt);
        gtk_toggle_button_set_active (GTK_TOGGLE_BUTTON (ttt->buttons[i][j]),
                                      FALSE);
        gtk_signal_handler_unblock_by_data (GTK_OBJECT(ttt->buttons[i][j]), ttt);
      }
}

static void
tictactoe_toggle (GtkWidget *widget, Tictactoe *ttt)
{
  int i,k;

  static int rwins[8][3] = { { 0, 0, 0 }, { 1, 1, 1 }, { 2, 2, 2 },
                             { 0, 1, 2 }, { 0, 1, 2 }, { 0, 1, 2 },
                             { 0, 1, 2 }, { 0, 1, 2 } };
  static int cwins[8][3] = { { 0, 1, 2 }, { 0, 1, 2 }, { 0, 1, 2 },
                             { 0, 0, 0 }, { 1, 1, 1 }, { 2, 2, 2 },
                             { 0, 1, 2 }, { 2, 1, 0 } };

  int success, found;

  for (k=0; k<8; k++)
    {
      success = TRUE;
      found = FALSE;

      for (i=0;i<3;i++)
        {
          success = success &&
            GTK_TOGGLE_BUTTON(ttt->buttons[rwins[k][i]][cwins[k][i]])->active;
          found = found ||
            ttt->buttons[rwins[k][i]][cwins[k][i]] == widget;
        }

      if (success && found)
        {
          gtk_signal_emit (GTK_OBJECT (ttt),
                           tictactoe_signals[TICTACTOE_SIGNAL]);
          break;
        }
    }
}
```

最后，一个使用我们的井字游戏构件的程序示例：

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

/* 当赢了时调用 */
void
win (GtkWidget *widget, gpointer data)
{
  g_print ("Yay!\n");
  tictactoe_clear (TICTACTOE (widget));
}

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

  gtk_init (&argc, &argv);

  window = gtk_window_new (GTK_WINDOW_TOPLEVEL);

  gtk_window_set_title (GTK_WINDOW (window), "Aspect Frame");

  gtk_signal_connect (GTK_OBJECT (window), "destroy",
                      GTK_SIGNAL_FUNC (gtk_exit), NULL);

  gtk_container_set_border_width (GTK_CONTAINER (window), 10);

  /* 创建一个新的井字游戏构件 */
  ttt = tictactoe_new ();
  gtk_container_add (GTK_CONTAINER (window), ttt);
  gtk_widget_show (ttt);

  /* 连接 "tictactoe" 信号的处理函数 */
  gtk_signal_connect (GTK_OBJECT (ttt), "tictactoe",
                      GTK_SIGNAL_FUNC (win), NULL);

  gtk_widget_show (window);

  gtk_main ();

  return 0;
}
```

从头创建构件

介绍

在这一节，我们将学习如何让构件把自己显示在屏幕上，以及如何让构件与事件交互。期间，我们将做一个表盘构件，用户可以拖动表盘上的指针来设定值。

在屏幕上显示构件

在屏幕上显示需要几个相关步骤。在调用 WIDGETNAME_new() 创建构件之后，如下几个函数需要用到：

- WIDGETNAME_realize() 如果构件有 X 窗口，该函数负责为构件创建 X 窗口。
- WIDGETNAME_map() 在用户调用 gtk_widget_show() 之后会调用该函数。它负责确保构件绘制在屏幕上。对于容器类，该函数必须调用每个子构件的 map() 函数。
- WIDGETNAME_draw() 当为构件或它的一个祖先调用 gtk_widget_draw() 时该函数被调用。它实际上是调用绘制函数在屏幕上绘制构件。对于容器构件，该函数必须为它的子构件调用 gtk_widget_draw()。
- WIDGETNAME_expose() 是构件的暴露事件处理函数。它调用绘制函数把暴露的部分绘制在屏幕上。对于容器构件，该函数必须为无窗口子构件产生暴露事件。(如果它们有自己的窗口，X 会产生必需的暴露事件。)

你可能注意到后面的两个函数十分相似，都是负责在屏幕上绘制构件。实际上许多构件并不真正关心它们之间的不同。构件类里的默认 draw() 函数只是简单的为重绘区域产生一个暴露事件。然而，一些构件通过区分这两个函数可以减少操作。例如，如果一个构件有多个 X 窗口，因为暴露事件标识了暴露的窗口，它可以只重绘受影响的窗口，调用 draw() 是不可能这样的。

容器构件，即使它们自身并不关心这个差别，也不能简单的使用默认 draw() 函数，因为它的子构件可能需要注意这个差别。然而，在两个函数里重复绘制代码是一种浪费的。按惯例，构件有一个名为 WIDGETNAME_paint() 的函数做实际的绘制构件的工作，draw() 和 expose() 函数再调用它。

在我们的示例里，因为表盘构件不是一个容器构件，并且只有一个窗口，我们采用最简便的方法，用默认的 draw() 函数，并且仅仅实现一个 expose() 函数。

表盘构件的原形

正 像陆上动物是从泥里爬出的两栖动物的变体，GTK 构件是其它的、以前写的构件的变体。因此，虽然这个章节命名为“从头创建构件”，但表盘构件实际上是从范围构件的源码上开始的。以它为起点是因为如果我们 的表盘构件能与比例构件有相同的接口会好一些，比例构件是范围构件的继承。所以，虽然源代码在下面以完整的形式出现，它不能说是*从头写*出来的。如果你还不熟悉比例构件如何以应用程序作者的观点来运作，最好先看一下前面的章节。

主体

我们的构件中的相当多的一部分看起来与井字游戏构件十分相似。首先，我们有一个头文件：

```
/* GTK - GIMP 工具包
 * 版权 (C) 1995-1997 Peter Mattis, Spencer Kimball 和 Josh MacDonald 所有
 *
 * 本程序是自由软件。你可以在自由软件基金发布的 GNU GPL 的条款下重新分发
 * 或修改它。GPL 可以使用版本 2 或(由你选择)任何随后的版本。
 *
 * 本程序分发的目的是它可能对其他人有用，但不提供任何的担保，包括隐含的
 * 和适合特定用途的保证。请查阅 GNU 通用公共许可证获得详细的信息。
 *
 * 你应该已经随该软件一起收到一份 GNU 通用公共许可。如果还没有，请写信给
 * Free Software Foundation, Inc., 59 Temple Place - Suite 330,
 * Boston, MA 02111-1307, USA.
 */

#ifndef __GTK_DIAL_H__
#define __GTK_DIAL_H__

#include <gdk/gdk.h>
#include <gtk/gtkadjustment.h>
#include <gtk/gtkwidget.h>


#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
```

```
#define GTK_DIAL(obj)          GTK_CHECK_CAST (obj, gtk_dial_get_type (), GtkDial)
#define GTK_DIAL_CLASS(klass)  GTK_CHECK_CLASS_CAST (klass, gtk_dial_get_type (), GtkDialClass)
#define GTK_IS_DIAL(obj)       GTK_CHECK_TYPE (obj, gtk_dial_get_type ())


typedef struct _GtkDial        GtkDial;
typedef struct _GtkDialClass   GtkDialClass;

struct _GtkDial
{
  GtkWidget widget;

  /* 更新方式 (GTK_UPDATE_[CONTINUOUS/DELAYED/DISCONTINUOUS]) */
  guint policy : 2;

  /* 当前按下的按钮，如果没有该值是 0 */
  guint8 button;

  /* 表盘指针的大小 */
  gint radius;
  gint pointer_width;

  /* 更新计时器的 ID，如果没有该值是 0 */
  guint32 timer;

  /* 当前角度 */
  gfloat angle;

  /* 将从调整对象中得到的旧值保存起来，这样在改变时我们就会知道 */
  gfloat old_value;
  gfloat old_lower;
  gfloat old_upper;

  /* 为这个表盘构件存储数据的调整对象 */
  GtkAdjustment *adjustment;
};

struct _GtkDialClass
{
  GtkWidgetClass parent_class;
};


GtkWidget*     gtk_dial_new                 (GtkAdjustment *adjustment);
GtkType        gtk_dial_get_type            (void);
GtkAdjustment* gtk_dial_get_adjustment      (GtkDial      *dial);
void           gtk_dial_set_update_policy   (GtkDial      *dial,
                                             GtkUpdateType  policy);

void           gtk_dial_set_adjustment      (GtkDial      *dial,
                                             GtkAdjustment *adjustment);
#ifdef __cplusplus
}
#endif /* __cplusplus */


#endif /* __GTK_DIAL_H__ */
```

因为相对于上一个构件，这个构件我们要做的工作更多，所以在数据结构里有更多的域，但是其它地方一样。

接下来，在包含了头文件和声明了几个常量之后，我们有几个提供构件信息的函数和初始化构件的函数：

```
#include <math.h>
#include <stdio.h>
#include <gtk/gtkmain.h>
#include <gtk/gtksignal.h>
```

```
#include "gtkdial.h"

#define SCROLL_DELAY_LENGTH  300
#define DIAL_DEFAULT_SIZE 100

/* 声明 */

[ 省略以节省空间 ]

/* 局部数据 */

static GtkWidgetClass *parent_class = NULL;

GtkType
gtk_dial_get_type ()
{
  static GtkType dial_type = 0;

  if (!dial_type)
    {
      static const GtkTypeInfo dial_info =
      {
        "GtkDial",
        sizeof (GtkDial),
        sizeof (GtkDialClass),
        (GtkClassInitFunc) gtk_dial_class_init,
        (GtkObjectInitFunc) gtk_dial_init,
        /* reserved_1 */ NULL,
        /* reserved_1 */ NULL,
        (GtkClassInitFunc) NULL
      };

      dial_type = gtk_type_unique (GTK_TYPE_WIDGET, &dial_info);
    }

  return dial_type;
}

static void
gtk_dial_class_init (GtkDialClass *class)
{
  GtkObjectClass *object_class;
  GtkWidgetClass *widget_class;

  object_class = (GtkObjectClass*) class;
  widget_class = (GtkWidgetClass*) class;

  parent_class = gtk_type_class (gtk_widget_get_type ());

  object_class->destroy = gtk_dial_destroy;

  widget_class->realize = gtk_dial_realize;
  widget_class->expose_event = gtk_dial_expose;
  widget_class->size_request = gtk_dial_size_request;
  widget_class->size_allocate = gtk_dial_size_allocate;
  widget_class->button_press_event = gtk_dial_button_press;
  widget_class->button_release_event = gtk_dial_button_release;
  widget_class->motion_notify_event = gtk_dial_motion_notify;
}

static void
gtk_dial_init (GtkDial *dial)
{
  dial->button = 0;
  dial->policy = GTK_UPDATE_CONTINUOUS;
  dial->timer = 0;
  dial->radius = 0;
  dial->pointer_width = 0;
```

```
  dial->angle = 0.0;
  dial->old_value = 0.0;
  dial->old_lower = 0.0;
  dial->old_upper = 0.0;
  dial->adjustment = NULL;
}

GtkWidget*
gtk_dial_new (GtkAdjustment *adjustment)
{
  GtkDial *dial;

  dial = gtk_type_new (gtk_dial_get_type ());

  if (!adjustment)
    adjustment = (GtkAdjustment*) gtk_adjustment_new (0.0, 0.0, 0.0, 0.0, 0.0, 0.0);

  gtk_dial_set_adjustment (dial, adjustment);

  return GTK_WIDGET (dial);
}

static void
gtk_dial_destroy (GtkObject *object)
{
  GtkDial *dial;

  g_return_if_fail (object != NULL);
  g_return_if_fail (GTK_IS_DIAL (object));

  dial = GTK_DIAL (object);

  if (dial->adjustment)
    gtk_object_unref (GTK_OBJECT (dial->adjustment));

  if (GTK_OBJECT_CLASS (parent_class)->destroy)
    (* GTK_OBJECT_CLASS (parent_class)->destroy) (object);
}
```

注意 init() 函数所做的工作比井字游戏构件少，因为这个构件不是复合构件，new() 函数所做的工作多一些，因为现在它具有一个参数。还要注意，当我们存储一个到调整对象的指针的时候，我们增加它的引用计数，(并在不再使用它的时候相应的减少它)，这样 GTK 就能明了在何时可以安全的销毁这个对象。

还有几个操作构件选项的函数：

```
GtkAdjustment*
gtk_dial_get_adjustment (GtkDial *dial)
{
  g_return_val_if_fail (dial != NULL, NULL);
  g_return_val_if_fail (GTK_IS_DIAL (dial), NULL);

  return dial->adjustment;
}

void
gtk_dial_set_update_policy (GtkDial      *dial,
                            GtkUpdateType  policy)
{
  g_return_if_fail (dial != NULL);
  g_return_if_fail (GTK_IS_DIAL (dial));

  dial->policy = policy;
}

void
gtk_dial_set_adjustment (GtkDial      *dial,
                         GtkAdjustment *adjustment)
{
```

```
  g_return_if_fail (dial != NULL);
  g_return_if_fail (GTK_IS_DIAL (dial));

  if (dial->adjustment)
    {
      gtk_signal_disconnect_by_data (GTK_OBJECT (dial->adjustment), (gpointer) dial);
      gtk_object_unref (GTK_OBJECT (dial->adjustment));
    }

  dial->adjustment = adjustment;
  gtk_object_ref (GTK_OBJECT (dial->adjustment));

  gtk_signal_connect (GTK_OBJECT (adjustment), "changed",
                      (GtkSignalFunc) gtk_dial_adjustment_changed,
                      (gpointer) dial);
  gtk_signal_connect (GTK_OBJECT (adjustment), "value_changed",
                      (GtkSignalFunc) gtk_dial_adjustment_value_changed,
                      (gpointer) dial);

  dial->old_value = adjustment->value;
  dial->old_lower = adjustment->lower;
  dial->old_upper = adjustment->upper;

  gtk_dial_update (dial);
}
```

gtk_dial_realize()

现在我们来看几个新的函数。第一个是创建 X 窗口的函数。注意有一个掩码传递给函数 gdk_window_new()，它指出 GdkWindowAttr 结构的哪些域实际上有数据在里面(其余的值会设为默认值)。同时也应该注意创建构件的事件掩码的方法。我们调用 gtk_widget_get_events() 去获取用户为这个构件设置的事件掩码 (用 gtk_widget_set_events())，并把我们需要的事件加入其中。

创建窗口之后，我们设置它的风格和背景，并把指向构件的指针放入 GdkWindow 的用户数据域。最后一步允许 GTK 分派这个窗口的事件到正确的构件。

```
static void
gtk_dial_realize (GtkWidget *widget)
{
  GtkDial *dial;
  GdkWindowAttr attributes;
  gint attributes_mask;

  g_return_if_fail (widget != NULL);
  g_return_if_fail (GTK_IS_DIAL (widget));

  GTK_WIDGET_SET_FLAGS (widget, GTK_REALIZED);
  dial = GTK_DIAL (widget);

  attributes.x = widget->allocation.x;
  attributes.y = widget->allocation.y;
  attributes.width = widget->allocation.width;
  attributes.height = widget->allocation.height;
  attributes.wclass = GDK_INPUT_OUTPUT;
  attributes.window_type = GDK_WINDOW_CHILD;
  attributes.event_mask = gtk_widget_get_events (widget) |
    GDK_EXPOSURE_MASK | GDK_BUTTON_PRESS_MASK |
    GDK_BUTTON_RELEASE_MASK | GDK_POINTER_MOTION_MASK |
    GDK_POINTER_MOTION_HINT_MASK;
  attributes.visual = gtk_widget_get_visual (widget);
  attributes.colormap = gtk_widget_get_colormap (widget);

  attributes_mask = GDK_WA_X | GDK_WA_Y | GDK_WA_VISUAL | GDK_WA_COLORMAP;
  widget->window = gdk_window_new (widget->parent->window, &attributes, attributes_mask);

  widget->style = gtk_style_attach (widget->style, widget->window);
```

```
  gdk_window_set_user_data (widget->window, widget);

  gtk_style_set_background (widget->style, widget->window, GTK_STATE_ACTIVE);
}
```

大小磋商

在包含构件的窗口第一次被显示前和每当窗口布局改变时，GTK 会询问每个子构件所期望的大小。函数 gtk_dial_size_request() 处理这个请求。因为我们的构件不是一个容器构件，且在其上也没有容器构件，我们仅仅返回一个合理的缺省值。

```
static void
gtk_dial_size_request (GtkWidget      *widget,
                       GtkRequisition *requisition)
{
  requisition->width = DIAL_DEFAULT_SIZE;
  requisition->height = DIAL_DEFAULT_SIZE;
}
```

在所有的构件已经请求了一个想要的大小之后，就开始计算窗口的布局，且每个子构件被告知它们实际的大小。通常，它至少是请求的大小，但是，如果，比如用户调整了窗口的大小，它偶尔可能小于请求的大小。函数 gtk_dial_size_allocate() 处理大小通知。注意在计算部件将要使用的大小的同时，这个例程也把构件的 X 窗口移到新位置和和设置新的大小。

```
static void
gtk_dial_size_allocate (GtkWidget     *widget,
                        GtkAllocation *allocation)
{
  GtkDial *dial;

  g_return_if_fail (widget != NULL);
  g_return_if_fail (GTK_IS_DIAL (widget));
  g_return_if_fail (allocation != NULL);

  widget->allocation = *allocation;
  if (GTK_WIDGET_REALIZED (widget))
    {
      dial = GTK_DIAL (widget);

      gdk_window_move_resize (widget->window,
                              allocation->x, allocation->y,
                              allocation->width, allocation->height);

      dial->radius = MAX(allocation->width,allocation->height) * 0.45;
      dial->pointer_width = dial->radius / 5;
    }
}
```

gtk_dial_expose()

像前面讲的一样，构件的所有的绘制在暴露事件处理函数里做。这里不需要多讲，除了它用三维阴影法，按照存储在构件的风格里的颜色，用函数 gtk_draw_polygon 绘制表的指针。

```
static gint
gtk_dial_expose (GtkWidget      *widget,
                 GdkEventExpose *event)
{
  GtkDial *dial;
  GdkPoint points[3];
  gdouble s,c;
  gdouble theta;
  gint xc, yc;
  gint tick_length;
  gint i;
```

```
  g_return_val_if_fail (widget != NULL, FALSE);
  g_return_val_if_fail (GTK_IS_DIAL (widget), FALSE);
  g_return_val_if_fail (event != NULL, FALSE);

  if (event->count > 0)
    return FALSE;

  dial = GTK_DIAL (widget);

  gdk_window_clear_area (widget->window,
                         0, 0,
                         widget->allocation.width,
                         widget->allocation.height);

  xc = widget->allocation.width/2;
  yc = widget->allocation.height/2;

  /* 绘制刻度 */

  for (i=0; i<25; i++)
    {
      theta = (i*M_PI/18. - M_PI/6.);
      s = sin(theta);
      c = cos(theta);

      tick_length = (i%6 == 0) ? dial->pointer_width : dial->pointer_width/2;

      gdk_draw_line (widget->window,
                     widget->style->fg_gc[widget->state],
                     xc + c*(dial->radius - tick_length),
                     yc - s*(dial->radius - tick_length),
                     xc + c*dial->radius,
                     yc - s*dial->radius);
    }

  /* 绘制指针 */

  s = sin(dial->angle);
  c = cos(dial->angle);


  points[0].x = xc + s*dial->pointer_width/2;
  points[0].y = yc + c*dial->pointer_width/2;
  points[1].x = xc + c*dial->radius;
  points[1].y = yc - s*dial->radius;
  points[2].x = xc - s*dial->pointer_width/2;
  points[2].y = yc - c*dial->pointer_width/2;

  gtk_draw_polygon (widget->style,
                    widget->window,
                    GTK_STATE_NORMAL,
                    GTK_SHADOW_OUT,
                    points, 3,
                    TRUE);

  return FALSE;
}
```

事件处理

我们的构件还剩下处理各种类型的事件的代码，但我们会发现和许多其它 GTK 程序里的没多大区别。可以产生两种类型的事件，一个是用户可以点击构件并拖动指针，另一个是通过外部的情况来改变调整对象的值。

当用户点击构件时，我们检查看这个点击是否是在表盘的指针里，如果是这样，把用户所点击的按钮存入构件结构的 button 域，并且调用 gtk_grab_add() 强占所有鼠标事件。随后的鼠标移动引发控制值被重新计算(通过函数 gtk_dial_update_mouse)。按照已经设定的方式(policy)，"value_changed" 事件被立即产生

(GTK_UPDATE_CONTINUOUS)，在用 gtk_timeout_add()添加的定时器里定义的一段延迟后(GTK_UPDATE_DELAYED)，或只在按钮被释放时 (GTK_UPDATE_DISCONTINUOUS)产生。

```
static gint
gtk_dial_button_press (GtkWidget      *widget,
                       GdkEventButton *event)
{
  GtkDial *dial;
  gint dx, dy;
  double s, c;
  double d_parallel;
  double d_perpendicular;

  g_return_val_if_fail (widget != NULL, FALSE);
  g_return_val_if_fail (GTK_IS_DIAL (widget), FALSE);
  g_return_val_if_fail (event != NULL, FALSE);

  dial = GTK_DIAL (widget);

  /* 判断按钮是否是在表盘指针你按下 - 我们通过计算鼠标按下
     点到表盘指针中线的水平和垂直距离来判断。 */

  dx = event->x - widget->allocation.width / 2;
  dy = widget->allocation.height / 2 - event->y;

  s = sin(dial->angle);
  c = cos(dial->angle);

  d_parallel = s*dy + c*dx;
  d_perpendicular = fabs(s*dx - c*dy);

  if (!dial->button &&
(d_perpendicular < dial->pointer_width/2) &&
(d_parallel > - dial->pointer_width))
    {
      gtk_grab_add (widget);

      dial->button = event->button;

      gtk_dial_update_mouse (dial, event->x, event->y);
    }

  return FALSE;
}

static gint
gtk_dial_button_release (GtkWidget      *widget,
                         GdkEventButton *event)
{
  GtkDial *dial;

  g_return_val_if_fail (widget != NULL, FALSE);
  g_return_val_if_fail (GTK_IS_DIAL (widget), FALSE);
  g_return_val_if_fail (event != NULL, FALSE);

  dial = GTK_DIAL (widget);

  if (dial->button == event->button)
    {
      gtk_grab_remove (widget);

      dial->button = 0;

      if (dial->policy == GTK_UPDATE_DELAYED)
        gtk_timeout_remove (dial->timer);

      if ((dial->policy != GTK_UPDATE_CONTINUOUS) &&
(dial->old_value != dial->adjustment->value))
```

```
            gtk_signal_emit_by_name (GTK_OBJECT (dial->adjustment), "value_changed");
        }

      return FALSE;
}

static gint
gtk_dial_motion_notify (GtkWidget      *widget,
                        GdkEventMotion *event)
{
  GtkDial *dial;
  GdkModifierType mods;
  gint x, y, mask;

  g_return_val_if_fail (widget != NULL, FALSE);
  g_return_val_if_fail (GTK_IS_DIAL (widget), FALSE);
  g_return_val_if_fail (event != NULL, FALSE);

  dial = GTK_DIAL (widget);

  if (dial->button != 0)
    {
      x = event->x;
      y = event->y;

      if (event->is_hint || (event->window != widget->window))
        gdk_window_get_pointer (widget->window, &x, &y, &mods);

      switch (dial->button)
        {
        case 1:
          mask = GDK_BUTTON1_MASK;
          break;
        case 2:
          mask = GDK_BUTTON2_MASK;
          break;
        case 3:
          mask = GDK_BUTTON3_MASK;
          break;
        default:
          mask = 0;
          break;
        }

      if (mods & mask)
        gtk_dial_update_mouse (dial, x,y);
    }

  return FALSE;
}

static gint
gtk_dial_timer (GtkDial *dial)
{
  g_return_val_if_fail (dial != NULL, FALSE);
  g_return_val_if_fail (GTK_IS_DIAL (dial), FALSE);

  if (dial->policy == GTK_UPDATE_DELAYED)
    gtk_signal_emit_by_name (GTK_OBJECT (dial->adjustment), "value_changed");

  return FALSE;
}

static void
gtk_dial_update_mouse (GtkDial *dial, gint x, gint y)
{
  gint xc, yc;
  gfloat old_value;
```

```
  g_return_if_fail (dial != NULL);
  g_return_if_fail (GTK_IS_DIAL (dial));

  xc = GTK_WIDGET(dial)->allocation.width / 2;
  yc = GTK_WIDGET(dial)->allocation.height / 2;

  old_value = dial->adjustment->value;
  dial->angle = atan2(yc-y, x-xc);

  if (dial->angle < -M_PI/2.)
    dial->angle += 2*M_PI;

  if (dial->angle < -M_PI/6)
    dial->angle = -M_PI/6;

  if (dial->angle > 7.*M_PI/6.)
    dial->angle = 7.*M_PI/6.;

  dial->adjustment->value = dial->adjustment->lower + (7.*M_PI/6 - dial->angle) *
    (dial->adjustment->upper - dial->adjustment->lower) / (4.*M_PI/3.);

  if (dial->adjustment->value != old_value)
    {
      if (dial->policy == GTK_UPDATE_CONTINUOUS)
        {
          gtk_signal_emit_by_name (GTK_OBJECT (dial->adjustment), "value_changed");
        }
      else
        {
          gtk_widget_draw (GTK_WIDGET(dial), NULL);

          if (dial->policy == GTK_UPDATE_DELAYED)
            {
              if (dial->timer)
                gtk_timeout_remove (dial->timer);

              dial->timer = gtk_timeout_add (SCROLL_DELAY_LENGTH,
                                             (GtkFunction) gtk_dial_timer,
                                             (gpointer) dial);
            }
        }
    }
}
```

通过外部方式产生的对 Adjustment 的改变通过 "changed" 和 "value_changed" 信号传到我们的构件。处理这些事情的处理函数将调用 gtk_dial_update()来验证参数，计算新的表盘指针角度，并重新绘制构件 (通过调用 gtk_widget_draw() 函数)。

```
static void
gtk_dial_update (GtkDial *dial)
{
  gfloat new_value;

  g_return_if_fail (dial != NULL);
  g_return_if_fail (GTK_IS_DIAL (dial));

  new_value = dial->adjustment->value;

  if (new_value < dial->adjustment->lower)
    new_value = dial->adjustment->lower;

  if (new_value > dial->adjustment->upper)
    new_value = dial->adjustment->upper;

  if (new_value != dial->adjustment->value)
    {
      dial->adjustment->value = new_value;
      gtk_signal_emit_by_name (GTK_OBJECT (dial->adjustment), "value_changed");
```

```
    }

  dial->angle = 7.*M_PI/6. - (new_value - dial->adjustment->lower) * 4.*M_PI/3. /
    (dial->adjustment->upper - dial->adjustment->lower);

  gtk_widget_draw (GTK_WIDGET(dial), NULL);
}

static void
gtk_dial_adjustment_changed (GtkAdjustment *adjustment,
                             gpointer       data)
{
  GtkDial *dial;

  g_return_if_fail (adjustment != NULL);
  g_return_if_fail (data != NULL);

  dial = GTK_DIAL (data);

  if ((dial->old_value != adjustment->value) ||
      (dial->old_lower != adjustment->lower) ||
      (dial->old_upper != adjustment->upper))
    {
      gtk_dial_update (dial);

      dial->old_value = adjustment->value;
      dial->old_lower = adjustment->lower;
      dial->old_upper = adjustment->upper;
    }
}

static void
gtk_dial_adjustment_value_changed (GtkAdjustment *adjustment,
                                   gpointer       data)
{
  GtkDial *dial;

  g_return_if_fail (adjustment != NULL);
  g_return_if_fail (data != NULL);

  dial = GTK_DIAL (data);

  if (dial->old_value != adjustment->value)
    {
      gtk_dial_update (dial);

      dial->old_value = adjustment->value;
    }
}
```

可能的增强

迄今为止我们描绘的 Dial 构件有大约 670 行代码。不过我们真正完成的只有一点点，头文件和模板占了其中的很大一部分。然而，对这个构件还有很多地方可以进行增强。

- 如 果你试一下这个构件，你会发现当拖动 pointer 转圈的时候有闪烁。这是因为每次表盘指针移动，整个构件在重绘前都要被擦除。最好的处理这个问题的方法 就是把这些变化绘制到一个不显示在屏幕上的 pixmap 上，然后一步将最后结果直接复制到屏幕上。(进度条构件就是以这种方式绘制它自身。)
- 用户应该可以通过上下光标键来增加或减少这个值。
- 最好让构件有一些按钮来小步或大步增加或减少这个值。虽然有可能用你含的(embedded)按钮来实现这个，但我们还是想让这个按钮在持续被按下的时候认为用户按下了很多次，就像滚动条上的箭头一样。在范围构件的代码中可以找到实现这种动作的大部分代码。
- 表盘构件可以做成一个容器构件，在以上所述的按钮中间表盘构件的底部放置一个简单的子构件。用户可以自己选择加入一个标签或文本输入构件来显示表盘的当前值。

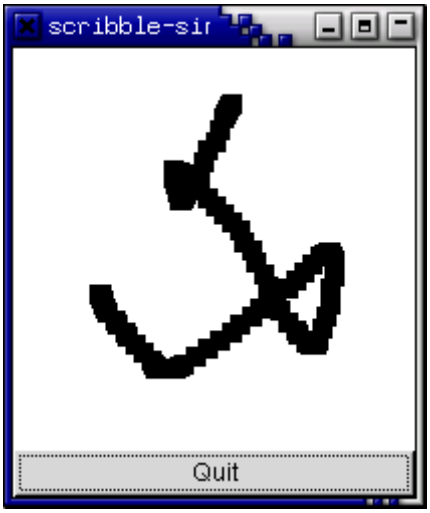
深入的学习

创 建一个构件涉及很多细节问题，我们以上只是说了一小部分。如果你想写出自己的构件，GTK 源代码是最好的例子。问你自己几个关于你想做的构件的问题：是不是一个容器构件？它是否有自己的窗口？它是否是一个对已经存在的构件的修改？然后找到一个 类似的构件，开始修改。祝你好运！

涂鸦板，一个简单的绘图程序

概述

在 这一章，我们会创建一个绘图程序。期间，我们讲解如何处理鼠标事件、如何在窗口你绘图和如何使用后端位图绘制更好的效果。在创建了简单的绘图程序之后，我 们会扩展该程序，通过添加对 XInput 设备的支持，如手写板。GTK 提供用来从这种设备轻易地获得如压力和倾角这样的扩展信息的例程。



事件处理

我们已经讨论了 GTK 信号中的高级的事件，如单选菜单项。然而，有时学习一些低级的事件也是有好用的，如鼠标移动或按一个键。在 GTK 中有信号与这些低级*事件*联系。这些信号的处理函数有额外的参数，该函数是一个结构指针，包含事件的信息。例如，传递给移动事件处理函数的参数是一个 GdkEventMotion 类型的结构指针，如下：

```
struct _GdkEventMotion
{
  GdkEventType type;
  GdkWindow *window;
  guint32 time;
  gdouble x;
  gdouble y;
  ...
  guint state;
  ...
};
```

type 会设置为事件的类型，如移动事件是 GDK_MOTION_NOTIFY，window 是发生事件的窗口。x 和 y 给出事件的座标。state 指出事件发生时的状态(按下了那个修正键或鼠标键)。它是如下值的位或：

```
GDK_SHIFT_MASK
GDK_LOCK_MASK
GDK_CONTROL_MASK
GDK_MOD1_MASK
GDK_MOD2_MASK
GDK_MOD3_MASK
GDK_MOD4_MASK
GDK_MOD5_MASK
GDK_BUTTON1_MASK
GDK_BUTTON2_MASK
GDK_BUTTON3_MASK
GDK_BUTTON4_MASK
```

GDK_BUTTON5_MASK

至于其它信号，我们调用函数 gtk_signal_connect()来决定事件发生时调用的处理函数。但是我们也需要让 GTK 知道我们想接收的事件。可以用如下函数:

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

第二个参数为我们感兴趣的事件。它为不同类型事件的位或。事件类型的列表如下:

GDK_EXPOSURE_MASK
GDK_POINTER_MOTION_MASK
GDK_POINTER_MOTION_HINT_MASK
GDK_BUTTON_MOTION_MASK
GDK_BUTTON1_MOTION_MASK
GDK_BUTTON2_MOTION_MASK
GDK_BUTTON3_MOTION_MASK
GDK_BUTTON_PRESS_MASK
GDK_BUTTON_RELEASE_MASK
GDK_KEY_PRESS_MASK
GDK_KEY_RELEASE_MASK
GDK_ENTER_NOTIFY_MASK
GDK_LEAVE_NOTIFY_MASK
GDK_FOCUS_CHANGE_MASK
GDK_STRUCTURE_MASK
GDK_PROPERTY_CHANGE_MASK
GDK_PROXIMITY_IN_MASK
GDK_PROXIMITY_OUT_MASK

当调用函数 gtk_widget_set_events()时，有几点需注意。首先，该函数必须在一个 GTK 构件的 X 窗口创建之前调用。实际上，意味者你应该在创建一个构件之后立即调用该函数。其次，构件必须有一个相关联的 X 窗口。为了提高效益，许多构件类型没有属于自己的窗口，它们绘制在父窗口上。这些构件是:

GtkAlignment
GtkArrow
GtkBin
GtkBox
GtkImage
GtkItem
GtkLabel
GtkPixmap
GtkScrolledWindow
GtkSeparator
GtkTable
GtkAspectFrame
GtkFrame
GtkVBox
GtkHBox
GtkVSeparator
GtkHSeparator

为了捕获这些构件的事件，你需要使用事件盒构件。详见 事件盒。

对于我们的绘图程序，我们想知道什么时候鼠标键按下和什么时候鼠标移动，因此我们要用 GDK_POINTER_MOTION_MASK 和 GDK_BUTTON_PRESS_MASK。我们也想知道什么时候窗口需要重新绘制，因此我们也要用 GDK_EXPOSURE_MASK。虽然我们也想在窗口尺寸改变时得到消息，不过我们不必用 GDK_STRUCTURE_MASK 标志，因为所有的窗口都自动设了该标志。

只用 GDK_POINTER_MOTION_MASK 是 有问题的。这会使服务器在每次用户移动鼠标时向事件队列添加一个移动事件。假设处理一个移动事件需要 0.1 秒，但是 X 服务器每 0.05 秒添加一个新的移动 事件。如果用户绘制要花 5 秒，那么在释放鼠标键后我们的程序会中断 5 秒！我们所需要的只是为我们处理的每个事件的获取一个移动事件。解决这个问题的方法是 要用 GDK_POINTER_MOTION_HINT_MASK。

当我们用 GDK_POINTER_MOTION_HINT_MASK 时，在指针进入我们的窗口之后、或在一个按钮按下或释放事件之后，服务器在指针首次移动时向我们发送一个移动事件。后发的移动事件都会被压制，直到我们用如下函数去获取鼠标指针的

位置：

```
GdkWindow*    gdk_window_get_pointer    (GdkWindow       *window,
                                         gint            *x,
                                         gint            *y,
                                         GdkModifierType *mask);
```

(还有另外一个函数 gtk_widget_get_pointer()，它有相似的接口，不过它不是很有用，因为它仅仅获取鼠标指针的位置，而不管按下了那个键。)

设置我们的窗口事件的代码如下：

```
  gtk_signal_connect (GTK_OBJECT (drawing_area), "expose_event",
                      (GtkSignalFunc) expose_event, NULL);
  gtk_signal_connect (GTK_OBJECT(drawing_area),"configure_event",
                      (GtkSignalFunc) configure_event, NULL);
  gtk_signal_connect (GTK_OBJECT (drawing_area), "motion_notify_event",
                      (GtkSignalFunc) motion_notify_event, NULL);
  gtk_signal_connect (GTK_OBJECT (drawing_area), "button_press_event",
                      (GtkSignalFunc) button_press_event, NULL);

  gtk_widget_set_events (drawing_area, GDK_EXPOSURE_MASK
                         | GDK_LEAVE_NOTIFY_MASK
                         | GDK_BUTTON_PRESS_MASK
                         | GDK_POINTER_MOTION_MASK
                         | GDK_POINTER_MOTION_HINT_MASK);
```

我们对在下一节讲解"expose_event"和"configure_event"的处理函数。"motion_notify_event"和"button_press_event"的处理函数很简单：

```
static gint
button_press_event (GtkWidget *widget, GdkEventButton *event)
{
  if (event->button == 1 && pixmap != NULL)
      draw_brush (widget, event->x, event->y);

  return TRUE;
}

static gint
motion_notify_event (GtkWidget *widget, GdkEventMotion *event)
{
  int x, y;
  GdkModifierType state;

  if (event->is_hint)
    gdk_window_get_pointer (event->window, &x, &y, &state);
  else
    {
      x = event->x;
      y = event->y;
      state = event->state;
    }

  if (state & GDK_BUTTON1_MASK && pixmap != NULL)
    draw_brush (widget, x, y);

  return TRUE;
}
```

绘图区构件和绘图

现在，我们开始向屏幕绘图。我们使用的构件是绘图区构件。一个绘图区构件本质上是一个 X 窗口，没有其它的东西。它是一个空白的画布，我们可以在其上绘制需要的东西。一个绘图区构件用如下函数创建：

```
GtkWidget* gtk_drawing_area_new       (void);
```

用如下函数设置构件的默认大小：

```
void      gtk_drawing_area_size      (GtkDrawingArea     *darea,
                                      gint               width,
                                      gint               height);
```

当调用函数 gtk_widget_set_size_request()或用户手动调整包含绘图区的窗口的大小时，默认大小可以无效，这对所有的构件都是一样的。

当我们创建绘图区构件时应该注意，我们*完全*负责绘制其上的内容。如果我们的窗口被遮住后暴露出来，我们得到一个暴露事件，我们必须重绘先前被遮住的部分。

为了能正确的重绘，我们必须记住绘制在屏幕上的内容。另外，这显然很麻烦，如果窗口的一部分被清除了，我们需一步步的重绘。解决的办法是使用一个*后端位图*。我们用向图像中绘制来代替直接向屏幕绘制，当图像改变或图像的一部分需要显示，我们复制相应的部分到屏幕上。

用如下函数创建后端位图：

```
GdkPixmap* gdk_pixmap_new           (GdkWindow  *window,
                                     gint       width,
                                     gint       height,
                                     gint       depth);
```

window 参数设置一个 GDK 窗口，位图继承该窗口的所有属性。width 和 height 设置位图的大小。depth 设置*颜色深度*，那是每个象素的二进制位数，如果 depth 设为-1，它会自动匹配窗口的颜色深度。

我们在事件"configure_event"的处理函数中创建位图。这个事件会在我们改变窗口大小时产生，包括窗口创建时。

```
/* 绘制区的后端位图 */
static GdkPixmap *pixmap = NULL;

/* 创建一个适当大小的后端位图 */
static gint
configure_event (GtkWidget *widget, GdkEventConfigure *event)
{
  if (pixmap)
    gdk_pixmap_unref(pixmap);

  pixmap = gdk_pixmap_new(widget->window,
                          widget->allocation.width,
                          widget->allocation.height,
                          -1);
  gdk_draw_rectangle (pixmap,
                      widget->style->white_gc,
                      TRUE,
                      0, 0,
                      widget->allocation.width,
                      widget->allocation.height);

  return TRUE;
}
```

调用函数 gdk_draw_rectangle()清除位图，并初始化为白色。后面我们会详细讲解。

我们的暴露事件处理函数只是简单复制相应部分的位图到屏幕上(用暴露事件的 event->area 来确定重绘区域)：

```
/* 从后端位图重新绘制屏幕 */
static gint
expose_event (GtkWidget *widget, GdkEventExpose *event)
{
  gdk_draw_pixmap(widget->window,
                  widget->style->fg_gc[GTK_WIDGET_STATE (widget)],
                  pixmap,
                  event->area.x, event->area.y,
                  event->area.x, event->area.y,
```

```
                    event->area.width, event->area.height);

  return FALSE;
}
```

现在我们来看如何保持屏幕跟随位图的更新，如何在位图上绘制我们需要的东西？ GTK 的 GDK 库中有许多函数用于在*可绘区域*绘图。可绘区域可以是窗口、位图或黑白图。在上面我们已经见到了两个，gdk_draw_rectangle()和 gdk_draw_pixmap()。这些函数的完全列表如下：

```
gdk_draw_line ()
gdk_draw_rectangle ()
gdk_draw_arc ()
gdk_draw_polygon ()
gdk_draw_string ()
gdk_draw_text ()
gdk_draw_pixmap ()
gdk_draw_bitmap ()
gdk_draw_image ()
gdk_draw_points ()
gdk_draw_segments ()
```

详见参考文档或头文件<gdk/gdk.h>。这些函数的头两个参数都相同。第一个参数是可绘区域。第二个参数是*图像关联* (GC)。

一 个图像关联封装一些信息，如前景色、背景色和线宽。GDK 有一组函数用于创建和修改图像关联，但为了方便，我们仅使用预定义的图像关联。每个构件有一个相 关联的风格。(可以在 gtkrc 文件中修改，详见 GTK 的 rc 文件)其中，存储了许多图像关联。一些访问这些图像 关联的示例如下：

```
widget->style->white_gc
widget->style->black_gc
widget->style->fg_gc[GTK_STATE_NORMAL]
widget->style->bg_gc[GTK_WIDGET_STATE(widget)]
```

域值 fg_gc、bg_gc、dark_gc 和 light_gc 索引取值靠一个 GtkStateType 类型的参数，该类型可以取如下值：

```
GTK_STATE_NORMAL,
GTK_STATE_ACTIVE,
GTK_STATE_PRELIGHT,
GTK_STATE_SELECTED,
GTK_STATE_INSENSITIVE
```

例如，GTK_STATE_SELECTED 默认的前景色是白色，默认的背景色是暗蓝色。

我们的函数 draw_brush()做实际的屏幕绘制工作。函数如下：

```
/* 在屏幕上绘制一个矩形 */
static void
draw_brush (GtkWidget *widget, gdouble x, gdouble y)
{
  GdkRectangle update_rect;

  update_rect.x = x - 5;
  update_rect.y = y - 5;
  update_rect.width = 10;
  update_rect.height = 10;
  gdk_draw_rectangle (pixmap,
                      widget->style->black_gc,
                      TRUE,
                      update_rect.x, update_rect.y,
                      update_rect.width, update_rect.height);
  gtk_widget_draw (widget, &update_rect);
}
```

在位图上绘制了矩形之后，我们调用如下函数：

```
void      gtk_widget_draw             (GtkWidget        *widget,
```

```
                                GdkRectangle       *area);
```

它会通知 X 参数 area 给定的区域需要更新。 X 会最终会产生一个暴露事件(混合区域需要多次调用函数 gtk_widget_draw())，然后会调用暴露事件处理函数，复制相应的部分到屏幕上。

现在我们已经有了一个较完整的绘图程序，只差主窗口部分了。

添加 XInput 支持

现在可以买到很便宜的输入设备，如手写板，用它绘图很方便。它可以用于代替鼠标，但这样失去了这个设备的许多优点：

- 压感
- 倾角报告
- 子象素定位
- 多输入(如铅笔和擦子)

关于 XInput 扩展的更多信息请参见 XInput HOWTO。

我们看 GdkEventMotion 结构的完全定义，我们会发现它包含支持扩展设备信息的域。

```
struct _GdkEventMotion
{
  GdkEventType type;
  GdkWindow *window;
  guint32 time;
  gdouble x;
  gdouble y;
  gdouble pressure;
  gdouble xtilt;
  gdouble ytilt;
  guint state;
  gint16 is_hint;
  GdkInputSource source;
  guint32 deviceid;
};
```

pressure 是压力，0 到 1 之间的浮点值。xtilt 和 ytilt 可以取-1 到 1 之间的值，表征在每个方向的倾斜度数。source 和 deviceid 用不同的方法指出发生事件的设备。source 给出设备的简短信息。它可以取如下枚举值：

```
GDK_SOURCE_MOUSE
GDK_SOURCE_PEN
GDK_SOURCE_ERASER
GDK_SOURCE_CURSOR
```

deviceid 是设备的统一数字 ID。它可用于得到设备的进一步信息，通过调用函数 gdk_input_list_devices()。特殊值 GDK_CORE_POINTER 用于主要指点设备。(通常是鼠标)

允许扩展设备信息

为了让 GTK 知道我们对扩展设备信息感兴趣，我们只需添加如下一行：

```
gtk_widget_set_extension_events (drawing_area, GDK_EXTENSION_EVENTS_CURSOR);
```

值 GDK_EXTENSION_EVENTS_CURSOR 说明我们对扩展事件感兴趣，且不想绘制自己的光标。关于绘制光标内容详见进一步讲解。我们也可以给出值 GDK_EXTENSION_EVENTS_ALL，如果我们想绘制自己的光标。 或给出值 GDK_EXTENSION_EVENTS_NONE 反转默认条件。

然而这还没有完，默认，扩展设备是不允许的。我们需要一个机制让用户去允许和配置扩展设备。下面的程序处理一个 InputDialog 构件。

```
void
input_dialog_destroy (GtkWidget *w, gpointer data)
{
```

```
  *((GtkWidget **)data) = NULL;
}

void
create_input_dialog ()
{
  static GtkWidget *inputd = NULL;

  if (!inputd)
    {
      inputd = gtk_input_dialog_new();

      gtk_signal_connect (GTK_OBJECT(inputd), "destroy",
                          (GtkSignalFunc)input_dialog_destroy, &inputd);
      gtk_signal_connect_object (GTK_OBJECT(GTK_INPUT_DIALOG(inputd)->close_button),
                                 "clicked",
                                 (GtkSignalFunc)gtk_widget_hide,
                                 GTK_OBJECT(inputd));
      gtk_widget_hide ( GTK_INPUT_DIALOG(inputd)->save_button);

      gtk_widget_show (inputd);
    }
  else
    {
      if (!GTK_WIDGET_MAPPED(inputd))
        gtk_widget_show(inputd);
      else
        gdk_window_raise(inputd->window);
    }
}
```

InputDialog 有两个按钮"关闭"和"保存"，默认它们没有被指定动作。在上面的函数，我们用"关闭"隐藏对话框，隐藏"保存"按钮，因为我们在这个程序里不用它。

使用扩展设备信息

一旦我们允许了这个设备，我们就能在事件结构中的额外域使用扩展设备信息。事实上，总是可以安全的使用这个信息，因为这些域值是合法的，甚至在扩展事件不允许时。

一旦改变，我们必须调用函数 gdk_input_window_get_pointer()代替 gdk_window_get_pointer。这是必要的，因为函数 gdk_window_get_pointer 不返回扩展设备信息。

```
void gdk_input_window_get_pointer( GdkWindow       *window,
                                   guint32         deviceid,
                                   gdouble         *x,
                                   gdouble         *y,
                                   gdouble         *pressure,
                                   gdouble         *xtilt,
                                   gdouble         *ytilt,
                                   GdkModifierType *mask);
```

当我们调用这个函数时，我们需要指定设备 ID 和窗口。通常，我们会从一个事件结构的 deviceid 域得到设备 ID。当扩展事件不允许时，这个函数也会返回合法的值。(这样 event->deviceid 是 GDK_CORE_POINTER)。

因此我们的按钮按下和鼠标移动事件处理函数的基本结构不需要改变，我们只需要添加处理扩展信息的代码。

```
static gint
button_press_event (GtkWidget *widget, GdkEventButton *event)
{
  print_button_press (event->deviceid);

  if (event->button == 1 && pixmap != NULL)
    draw_brush (widget, event->source, event->x, event->y, event->pressure);

  return TRUE;
}
```

```
static gint
motion_notify_event (GtkWidget *widget, GdkEventMotion *event)
{
  gdouble x, y;
  gdouble pressure;
  GdkModifierType state;

  if (event->is_hint)
    gdk_input_window_get_pointer (event->window, event->deviceid,
                                  &x, &y, &pressure, NULL, NULL, &state);
  else
    {
      x = event->x;
      y = event->y;
      pressure = event->pressure;
      state = event->state;
    }

  if (state & GDK_BUTTON1_MASK && pixmap != NULL)
    draw_brush (widget, event->source, x, y, pressure);

  return TRUE;
}
```

我们也需要对新的信息做些事。我们的新的 draw_brush()函数根据每一个 event->source 绘制不同的颜色，依据压力改变画刷的大小。

```
/* 在屏幕上画一个矩形，大小依据压力，颜色依据设备的类型 */
static void
draw_brush (GtkWidget *widget, GdkInputSource source,
            gdouble x, gdouble y, gdouble pressure)
{
  GdkGC *gc;
  GdkRectangle update_rect;

  switch (source)
    {
    case GDK_SOURCE_MOUSE:
      gc = widget->style->dark_gc[GTK_WIDGET_STATE (widget)];
      break;
    case GDK_SOURCE_PEN:
      gc = widget->style->black_gc;
      break;
    case GDK_SOURCE_ERASER:
      gc = widget->style->white_gc;
      break;
    default:
      gc = widget->style->light_gc[GTK_WIDGET_STATE (widget)];
    }

  update_rect.x = x - 10 * pressure;
  update_rect.y = y - 10 * pressure;
  update_rect.width = 20 * pressure;
  update_rect.height = 20 * pressure;
  gdk_draw_rectangle (pixmap, gc, TRUE,
                      update_rect.x, update_rect.y,
                      update_rect.width, update_rect.height);
  gtk_widget_draw (widget, &update_rect);
}
```

得到更多关于设备的信息

作为一个如何得到更多关于设备的信息的示例，我们的程序在每次按钮按下时打印设备名。用如下函数可以得到设备名：

```
GList *gdk_input_list_devices           (void);
```

返回值是一个 GdkDeviceInfo 结构的 GList(GLib 库的一个链表类型)GdkDeviceInfo 结构定义如下：

```
struct _GdkDeviceInfo
{
  guint32 deviceid;
  gchar *name;
  GdkInputSource source;
  GdkInputMode mode;
  gint has_cursor;
  gint num_axes;
  GdkAxisUse *axes;
  gint num_keys;
  GdkDeviceKey *keys;
};
```

这些域的大部分都是可以忽略的配置信息，除非你要实现 XInput 配置保存。我们感兴趣的域是 name，它是 X 分配给设备的名子。其它的不是配置信息的域是 has_cursor。如果 has_cursor 是 FALSE，我们需要自绘制光标。但因为我们已经指定了 GDK_EXTENSION_EVENTS_CURSOR，所以我们不必关心这个。

函数 print_button_press()简单的重复，直到找到匹配，然后打印出设备名。

```
static void
print_button_press (guint32 deviceid)
{
  GList *tmp_list;

  /* gdk_input_list_devices 返回一个内部列表，因此我们后面不必释放它。*/
  tmp_list = gdk_input_list_devices();

  while (tmp_list)
    {
      GdkDeviceInfo *info = (GdkDeviceInfo *)tmp_list->data;

      if (info->deviceid == deviceid)
        {
          printf("Button press on device '%s'\n", info->name);
          return;
        }

      tmp_list = tmp_list->next;
    }
}
```

我们的程序已经完全添加了对 XInput 设备的支持。

进一步的讲解

虽然我们的程序已经很好的支持了 XInput，但是它缺乏一些特性，我们想让它成为一个全功能的程序。首先，用户不想每次在程序运行时配置设备，因此我们应该允许用户保存设备配置。这是通过获取 gdk_input_list_devices()的返回值，并把配置写入一个文件。

为了程序下次运行时恢复状态，GDK 提供修改设备配置的函数：

```
gdk_input_set_extension_events()
gdk_input_set_source()
gdk_input_set_mode()
gdk_input_set_axes()
gdk_input_set_key()
```

(gdk_input_list_devices()返回的列表不能直接修改。)在绘图程序 gsumi 中可以发现它的用法。(http://www.msc.cornell.edu/~otaylor/gsumi/)其实做这个，最好使用所有程序标准的方法。这也许属于比 GTK 稍高级的库，也许在 GNOME 库中。

另一个缺点是我们上面提到的，缺乏光标绘制。当前平台 XFree86 不允许同时用一个设备和主指点设备在一个应用程序中。详见 XInput-HOWTO。更好的应用程序应该绘制自己的光标。

一 个程序要绘制自己的光标，需要两方面：确定当前设备是否需要绘制光标，确定当前设备是否"in proximity"。(如果当前设备是手写板，最好在笔尖离开平板时不显示光标。当设备是触摸板时，那叫做"in proximity"。)首先要搜索设备列表，寻找设备名。其次是选择 "proximity_out" 事件。它的用法见 GTK 发布中的示例程序"testinput".

编写 GTK 应用程序的技巧

使 用 GNU autoconf 和 automake。它们是你的好帮手，Automake 检查 C 文件，判断它们如何相互依赖，并且生成一个 Makefile 以使文件能按照正确的顺序被编译。Autoconf 允许软件安装过程自动配置，处理了大量的系统相关问题并且增加了可移植性。我准备在这里写一个有关它们快速入门。

当写 C 代码时，只使用 C 的注释(以“/*”开头，以“*/”结尾)，不要用 C++ 风格的注释(“//”)。虽然许多 C 编译器能明白 C++ 的注释，但其它的一些不能，并且 ANSI C 标准并不要求 C++ 风格的注释被处理为注释。