目　录

第 1 章　窗口与简单控件

本章重点：

开发图形用户界面(GUI)程序的第一步是创建窗口，然后完善窗口的功能，进一步向窗口中添加一些常用的控件，再就是编程使控件响应相关的事件。用 GTK+2.0 来开发 GUI 程序也是如此。

GTK+2.0 提供一种用最短的代码来编写窗口和控件的方法，还有灵活易用的信号/回调函数机制。通过本章的学习，读者能够创建简单的用户界面，编写一般的回调函数，理解信号/回调函数机制。

本章主要内容：

- 如何使用窗口控件
- 如何使用容器控件
- 如何使用按钮控件
- 如何使用标签、图像控件

1.1　最简单的窗口

本节将介绍用 GTK+2.0 创建一个最简单的窗口，并把它显示出来的方法。同时通过这一过程理解 GTK+2.0 程序结构。

实例说明

窗口是 GUI 编程中直接面对用户的操作对象，创建窗口也就成了初学者学习 GUI 编程的第一步。GTK+2.0 提供了非常简便的创建窗口的方法，用它写出的代码在所有 GUI 开发工具中几乎是最短的，也是最容易理解的。

实现步骤

(1) 启动 Linux，进入 GNOME 桌面环境，打开终端输入如下命令：

```
cd ~
mkdir ourgtk
cd ourgtk
mkdir 1
cd 1
mkdir base
cd base
```

创建工作目录，并进入此目录开始编程。

说明：首先进入用户工作目录，创建一个学习 GTK+2.0 总的工作目录 ourgtk，接着创

建本章的工作目录 1，然后再创建本节的工作目录 base，进入此目录开始工作。

(2) 打开编辑器(GEDIT2.0 见前言)。输入如下代码，以 base.c 为文件名保存到当前目录(base)下：

```
/* 最简单的窗口base.c */
#include <gtk/gtk.h>
int main ( int argc , char* argv[])
{
GtkWidget *window;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_widget_show(window);
    gtk_main();
return FALSE;
}
```

(3) 编辑 Makefile 输入如下代码：

```
CC = gcc
all:
    $(CC) -o base base.c 'pkg-config --cflags --libs gtk+-2.0'
```

注意： $(cc)前面不是空格，而是一个<tab>键，如果用空格代替编译时会出问题。

以 Makefile 为文件名保存到 base 目录下。

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./base 即可运行此程序，运行结果如图 1.1 所示：

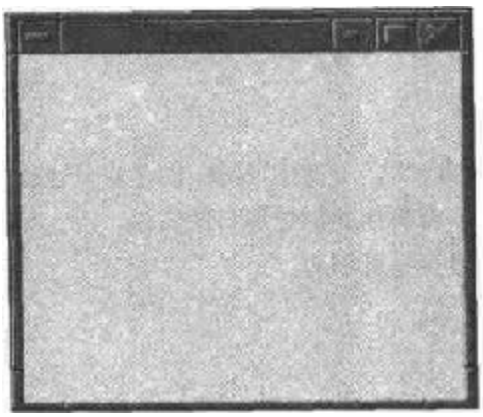

图 1.1 最简单的窗口

实例分析

(1) 设置 include 文件与声明主函数

首先是#include <gtk/gtk.h> 语句，这是每个 GTK+2.0 程序都要包含的头文件。然后是函数声明 int main (int argc , char* argv[])，这是标准的 C 语言主函数的声明。

(2) 声明变量

GtkWidget *window; 声明了窗口控件的指针。其中 GtkWidget 是 GTK+2.0 控件类型，

几乎所有的 GTK+2.0 控件都用这一类型声明；window 是变量名，它与变量类型无关。完全可以给它起一个诸如 asd21i9 这样的名字，不过这只会令程序代码更难读，所以最好起一个易于理解的变量名，比如 window。

(3) 初始化 GTK+2.0 的命令行参数

函数 gtk_init(&argc,&argv); 初始化命令行参数。这在 GTK+2.0 程序中是必需的，不管您的设计中是否使用到命令行参数，都需要用这一函数来初始化。

(4) 创建窗口

代码行 window = gtk_window_new(GTK_WINDOW_TOPLEVEL); 用来创建窗口。函数 gtk_window_new 创建一个窗口并返回这个窗口的控件指针，在这里这个指针的值赋给了变量 window；参数 GTK_WINDOW_TOPLEVEL 指明窗口的类型为最上层的主窗口，它最常用。还可以取另一个值 GTK_WINDOW_POPUP 指明窗口的类型为弹出式的无边框的窗口。

(5) 显示窗口

代码行 gtk_widget_show(window); 用来显示上一步创建的窗口。函数 gtk_widget_show 是用来显示控件的，它没有返回类型，参数是要显示的控件的指针，在这里是窗口 window，所以窗口就显示出来了。

(6) 主事件循环

最后这个函数 gtk_main(); 是最关键的，它是 GTK+2.0 的主事件循环，每一个 GTK+2.0 程序都要有一个，否则程序就无法运行。所谓事件循环是指 GUI 程序运行时等待来自外部用户发出的事件，如键入或鼠标移动等，GTK+2.0 将这些事件包装成信号，用户再根据信号的功能编写相应的回调函数来处理这些事件。这段代码中并未写回调函数，所以这只是一个死循环，不做任何反应。

代码最后返回逻辑值 FALSE，它相当于整型的 0。不写这一行是完全可以的，但写这行代码更能体现程序的完整性和可读性。

(7) 编译运行

以上代码完全可以用下面命令行直接编译：

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

但在终端上输入这么一长串的命令非常繁琐且容易出错，更糟糕的是如果长时间不用的话还可能忘记这串命令。幸好 Linux 提供了 make 工具，这样按上面步骤编辑好 make 的配置文件 Makefile，在命令行中直接输入命令 make 就可以编译了，编译结束后输入命令./base 就可运行这个程序了。(注意：base 前面一定要加上点和斜杠，表示在当前目录下运行，否则的话终端找不到要运行的程序。)

读者还会注意到命令行中的'pkg-config --cflags --libs gtk+-2.0'，它向编译器指出了包含文件的路径，动态链接库路径和要链接哪些动态链接库。pkg-config 是 GTK+2.0 和 GNOME2 系统必备的软件包配置和管理工具，可以在命令中直接运行。

至此，我们完成了最简单的变量的声明、窗口的创建、显示，了解了 GTK+2.0 程序的一般结构、GTK+2.0 程序中的两个关键步骤：gtk_init 和 gtk_main。这些都是 GTK+2.0 程序中必需的，为下面进一步学习 GTK+2.0 编程打下基础。

1.2 完善窗口的功能

本节将介绍如何完善窗口的功能，为窗口或控件添加回调函数，对窗口进行相应的功能设置，使它能真正的退出，更加实用。

实例说明

上节中的示例运行时，单击关闭按钮，窗口会自动关闭，但终端中的提示符却不见了。其实完全可以关闭终端窗口，再开启一个。不过最好的办法是按下 Ctrl+C 键，这样提示符就出现了。

为什么会出现这种情况呢？上面示例中只调用了主事件循环 gtk_main，而未对来自用户的信号做任何处理，所以即使窗口关闭了，循环仍在执行，程序并未真正退出。按下 Ctrl+C 键后，系统向此程序发出了中断信号，程序才真正地终止执行了。

本节示例以解决这一问题为目的，更进一步完善窗口的功能。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/1
mkdir window
cc window
```

创建工作目录，进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 window.c 为文件名保存到当前目录下：

```
/* 完善窗口的功能 window.c */
#include <gtk/gtk.h>

int main ( int argc , char* argv[])
{
    GtkWidget *window;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window)," 一个功能完善的窗口");
    gtk_window_set_default_size(GTK_WINDOW(window),500,100);
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_widget_show(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
```

```
$(CC) -o window window.c 'pkg-config --cflags --libs gtk+2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./window 即可运行此程序，运行结果如图 1.2 所示。

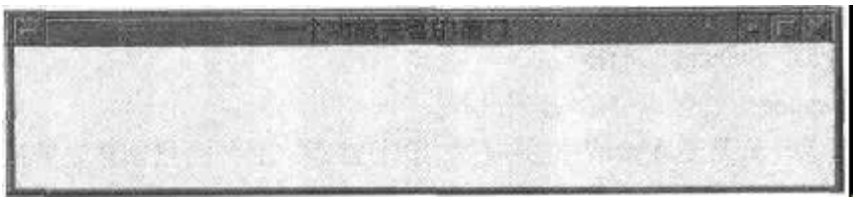

图 1.2 功能完善的窗口

实例分析

(1) 信号与回调函数

GTK+2.0 采用了一种信号/回调函数机制来处理窗口外部传来的事件、消息或信号。即先为窗口或控件定义一系列信号，在编程中引用信号名称为窗口或控件添加回调函数，当信号发生时，程序自动调用为信号连接的回调函数。

窗口的信号有很多，如“key_press_event”在按键时发生，“foucs”在获得焦点时发生等等。这里主要介绍“delete_event”，这一信号在窗口关闭时发生。

(2) 退出程序

退出 GTK+2.0 程序要调用 gtk_main_quit();函数，它的功能就是退出主循环，也就是结束程序的运行。一般情况下当窗口关闭时程序就退出了，但 GTK+2.0 并不主动处理退出程序。这就需要为上面介绍的“delete_event”信号连接回调函数。

(3) 为信号连接回调函数

为窗口或控件加回调函数有两种方式：

一种方式是直接调用已有函数(如退出函数 gtk_main_quit)，在窗口或控件创建完成后直接引用 g_signal_connect 宏，代码如下所示：

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

另一种方式是先定义好回调函数，在窗口或控件创建完成后再引用 g_signal_connect 宏，代码如下所示：

```
/* 声明回调函数 on_delete_event */
void    on_delete_event (GtkWidget *widget,GdkEvent* event,gpointer data)
{
    gtk_main_quit();
    return FALSE;
}
......
/* 在主函数中为窗口的"delete_event"信号加回调函数 */
......
g_signal_connect(G_OBJECT(window),"delete_event",
        G_CALLBACK(on_delete_event),NULL);
......
```

在 GTK+1.2 中经常用第二种方式来退出程序运行，在 GTK+2.0 中就完全可以不用这种方式，采用第一种方式更简单明了。这种方式的好处是：可以在 on_delete_event 函数中加入其他代码来处理程序开始时遗留的问题(如释放已分配的内存、保存程序配置等)，还可以询问用户是否真正退出程序运行。

(4) g_signal_connect 宏的格式

g_signal_connect 宏有 4 个参数，分别是：

连接的对象，就是要连接信号的控件的指针(注意：必须是已创建完的控件的指针)，需要用 G_OBJECT 宏来转换，如本例中的 G_OBJECT(window)；

信号名称，就是要连接的信号名称，为字符串形式，用双引号引起来。不同的控件拥有的信号名称是不一样的，如本例中的窗口控件的信号："delete_event"；

回调函数，指信号发生时调用的函数，这里只用到函数名称，需要用 G_CALLBACK 宏来转换一下，如本例中的 G_CALLBACK(gtk_main_quit)；

传递给回调函数的参数，它的值类型应该为 gpointer 如果不是这一类型可以强制转换，如果没有参数则为 NULL。这里只能传递一个参数，如果有多个参数，可以先将多个参数定义为一个结构，再将此结构作为参数传递过来。

注意： gpointer、gchar、gint、gboolean 等类型是在 GTK+2.0 的基础库 GLIB2.0 中定义的，是对 C 语言基础数据类型的包装。它渗透到 GTK+2.0 编程的每一个角落，读者一定要细致地研究一下，这对进一步学好 GTK+2.0 编程很有好处。

(5) 回调函数的格式

不同的控件的信号不同，不同的信号的回调函数的格式也不同。有一个规律，即多数回调函数是没有返回类型的，名称可以自定义，最好能表达一定的意思，参数有多个，第一个参数是调用此回调函数的控件对象指针，最后一个参数是用户传递给此回调函数的参数，而且固定为 gpointer 类型。如上面介绍的 on_delete_event 函数表示当“delete_event”信号发生时执行，它的第 1 个参数 widget 是指窗口控件的指针，第 2 个参数 event 是指事件类型，第 3 个参数 userdata 是指用户传递的参数。

(6) 改变窗口外观的几个函数

设定窗口的标题：

```
gtk_window_set_title(window,const gchar* title);
```

设定窗口的默认宽高：

```
gtk_window_set_default_size(window,int width,int height);
```

设定窗口的位置：

```
gtk_window_set_position(window,GtkWindowPosition position);
```

其中 position 可以取如下值：

```
GTK_WIN_POS_NONE     不固定
GTK_WIN_POS_CENTER   居中
GTK_WIN_POS_MOUSE    出现在鼠标位置
GTK_WIN_POS_CENTER_ALWAYS        窗口改变尺寸仍居中
```

GTK_WIN_POS_CENTER_ON_PARENT　　居于父窗口的中部

更多的函数见 GTK+2.0 的 API 参考手册。

运行这个程序时会发现，窗口的标题是中文的，窗口的位置处于屏幕中央，且尺寸也变了。更可喜的是关闭窗口后，终端上的提示符号上出现了。

本节的示例主要研究了信号和回调函数的一般格式和用法，这是编写 GTK+2.0 程序的关键所在，读者还应加强这方面的理解。

1.3　Hello 程序

本节将介绍如何向窗口中添加按钮控件，为按钮控件加回调函数——实现一个简单的计数器，并在窗口中显示文字。

实例说明

一般的程序设计课程都以一个 Hello 程序作为第一个示例，在 GTK+2.0 中也很容易做到。首先创建一个窗口，在窗口中添加一个按钮，单击按钮，使之做出反应，在终端上显示一行信息。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/1
mkdir hello
cd hello
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 hello.c 为文件名保存到当前目录下：

```
/* hello程序  hello.c */
#include <gtk/gtk.h>
gint count=1;
void
on_button_clicked   (GtkWidget *button,gpointer userdata)
{
    g_print("你好，这是Hello功能的测试.   ");
    //g_print("Hello . This is a test . ");
    g_print("%d\n",(gint)userdata);
    //g_print("%d\n",count);
    count = count + 1 ;
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *button;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
```

```
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"Hello 功能实现");
    gtk_window_set_default_size(GTK_WINDOW(window),500,100);
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),40);
    button = gtk_button_new_with_label("按下此按钮会在终端上显示一行信息
");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_button_clicked),(gpointer)count);
    gtk_container_add(GTK_CONTAINER(window),button);
    gtk_widget_show(button);
    gtk_widget_show(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o hello hello.c 'pkg-config --cflags --libs gtk+2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./hello 即可运行此程序，运行结果如图 1.3 所示。

图 1.3 HELLO 程序

实例分析

(1) 按钮控件

这里介绍 GTK+2.0 中常用的普通按钮控件。GTK+2.0 控件是统一用 GtkWidget 类型来管理的，按钮控件也不例外。我们可以用 gtk_button_new 来创建一个不显示内容的空按钮，还可以用 gtk_button_new_with_label 来创建一个显示文字的按钮，本示例中用的是后者，相信读者一看就知道。

注意代码书写顺序，先定义，再创建，再加回调函数，最后显示。

看到这里读者一定会明白 GTK+2.0 的函数名称特色：以 gtk 开头，用下划线连接；第二个单词表示控件的类型，如：button、window 等；再后面的相关的单词表示要做的动作，如：new_with_label(新建并带标签)，set_default_size(设定默认的尺寸)等。使读者基本能见其名而知其含义，也非常容易掌握。

(2) 容器

GTK+2.0 中的控件的摆布采用了容器(container)这一概念，即所有 GTK+2.0 控件分成两种，一种是能容纳其他控件的容器，另一种是不能容纳其他控件的非容器控件。容器控件又分成能容纳多个控件的容器和只能容纳一个控件的容器。

窗口控件也是一种容器，它可以把按钮容纳进来。如何操作呢？GTK+2.0 提供了与容器相关的操作函数。以 gtk_container 开头，函数 gtk_container_add 的功能是将另一控件加入到容器中来。它的第一参数是 GtkContainer 型的指针，这就需要将窗口控件指针用宏 GTK_CONTAINER 转换一下，即 GTK_CONTAINER(window)。它的第二参数是要容纳的控件的指针，即 button。另一个常用到的与容器相关的函数是 gtk_container_set_border_width。它用来设定容器边框的宽度，格式如例中所示，宽度的单位是像素，可以根据需要自行设定。如不设定窗口边框的宽度，其默认值是 0，这样的话按钮的显示不明显。

(3) 按钮的回调函数

创建完按钮后，我们给按钮的单击信号“clicked”加了一个自定义的回调函数。这个命名为 on_button_clicked 的函数，其格式一般如下：

```
void 函数名 (GtkButton* button,gpointer userdata)
```

在这个回调函数中，使用了 g_print 函数。它的格式和 C 语言中最常用的 printf 函数格式一样，而且功能也一样，都是向输出设备输出格式化字符串。

代码中我们还定义了一个静态计数器——整型变量 i，即每执行一次这个函数(每单击一次按钮)其值自动加一，这样每次输出的数字也就都不一样了。

运行此程序，单击按钮，终端上就会输出“你好，这是 Hello 功能的测试. 1”，再按一下 1 就会变成 2，如此继续直到关闭窗口。如果您的终端窗口不支持中文，将代码中的中文输出行加上注释，去掉另两行(英文输出)的注释则会显示英文输出结果。

编完这个示例可能有些读者会很失望，因为只在终端中输出信息，如果能直接弹出消息框就好了。不过别着急，第 4 章将详细介绍各种消息框的使用。

1.4 用盒状容器排列按钮

本节将介绍 GTK+2.0 中能容纳多个控件的盒状容器，以及如何灵活使用盒状容器控件，如何在盒状容器中排放多个控件和设置它们的排放属性。

实例说明

当继续向窗口中添加按钮时会发现程序又出错了，这是因为窗口只能容纳一个控件。如何容纳多个控件呢？上节我们知道容器又分成两种，窗口属于只能容纳一个控件的容器，如果向窗口中加一个能容纳多个控件的容器，再向此容器中添加其他别的控件，就可以实现这一目标。

GTK+2.0 中能容纳多个控件的容器主要有盒状容器(GtkBox)、格状容器(GtkTable)、按钮盒(GtkButtonBox)、分隔面板(GtkPanel)、固定布局(GtkFixed)、工具栏(GtkToolbar)等，

其中最常用的是盒状容器和格状容器。

盒状容器是一种按一定顺序和方向紧密排列多个控件的容器。在盒状容器(GtkBox)基础上又分成纵向盒状容器(GtkVBox)和横向盒状容器(GtkHBox)，两者除了创建和排列控件的方式不一样外，其使用方式是一模一样的。

我们还应注意的是容器中还能容纳其他容器，这形成了用 GTK+2.0 创建复杂用户界面的基础。一定要把容器间容纳与被容纳关系理顺。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/1
mkdir pack
cd pack
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 pack.c 为文件名保存到当前目录下：

```
/* 用盒状容器排列按钮 pack.c */
#include <gtk/gtk.h>
int main ( int argc , char* argv[])

    GtkWidget *window;
    GtkWidget *box;
    GtkWidget *button;
    gchar *title = "排列按钮" ;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),title);
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),20);
    box = gtk_hbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),box);
    button = gtk_button_new_with_label("按钮 ");
    gtk_box_pack_start(GTK_BOX(box),button,TRUE,TRUE,3);
    button = gtk_button_new_with_label("按");
    gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,3);
        button = gtk_button_new_with_label("钮");
    gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,3);
    button = gtk_button_new_with_label(" ");
    gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,3);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o pack pack.c 'pkg-config --cflags --libs gtk+2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./pack 即可运行此程序，运行结果如图 1.4 所示。

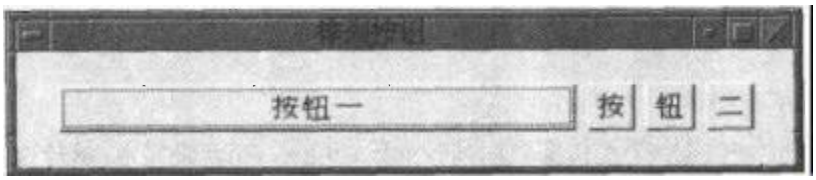

图 1.4　用盒状容器排列按钮

实例分析

(1) 盒状容器

这段代码中我们先是定义盒状容器的控件指针 box，在处理好窗口内容设置后开始创建盒状容器，用 gtk_hbox_new 创建横向盒状容器，用 gtk_vbox_new 创建纵向盒状容器，我们选择了前者。这两个函数都有两个参数，第一个参数是逻辑型值，表示容器内的控件是否均匀排放，我们取 FALSE，即不均匀排放(有大有小)；第二个参数是一个整型值，表示容器中控件的间隔，单位是像素，我们取值 0，即无间隔。应该注意，基础的 GtkBox 类是不能直接创建的。

(2) 向盒状容器中排放控件

向盒状容器中添加控件也可以使用 gtk_container_add 函数，但这样只能加一个控件。最好的办法是用 gtk_box_pack_*系列函数向盒状容器添加并排列控件，这样的函数一共有 4 个，分别是：gtk_box_pack_start、gtk_box_pack_end、gtk_box_pack_ start_defaults 和 gtk_box_pack_end_defaults。

gtk_box_pack_start、gtk_box_pack_end 分别表示按顺序从前到后依次排列控件和从后到前依次排列控件。这两个函数都有 5 个参数，依次是 GTK_BOX(box)，要容纳控件的容器对象；button，被容纳控件的指针；是否扩展，是否添充和与前一控件的间隔。如本示例中的：

```
gtk_box_pack_start(GTK_BOX(box),button,TRUE,TRUE,3);
```

如果不考虑控件间的间隔、是否扩展和是否添充的话还可以使用函数 gtk_box_pack_start_defaults 和 gtk_box_pack_end_defaults。这两函数只有两个参数，即容器对象指针和被容纳对象指针。注意，由于我们创建的是 HBox 或 VBox，而这些函数操作对象是 Box，所以一定不要忘了用 GTK_BOX 宏转换一下，否则编译时会出警告信息，程序也会变得不稳定。如下面的代码：

```
gtk_box_pack_start_defaults(GTK_BOX(box),button);
```

可以用函数 gtk_box_set_homogeneous 和 gtk_box_get_homogeneous 来设定/取得盒状容

器是否允许子控件均匀排放，用函数 gtk_box_set_spacing 和 gtk_box_get_spacing 来设定/取得盒状容器的子控件的间隔。

(3) 创建按钮

这段代码中只定义了一个控件指针 button，我们这里却创建了 4 个按钮，因为每个按钮创建之后就不再对它操作了(即不再明确使用这个指针)，所以下一个按钮创建时还会为这个指针赋值，这在 GTK+2.0 中是允许的。这样做有利有弊，利在于只声明了一个指针变量，代码行数减少了；弊则是如果分别对这几个按钮操作必需再声明不同名称的变量。

(4) 显示所有控件

前面几个示例中每一个控件都要用函数 gtk_widget_show 来显示，这样此段代码还得增加 6 行，1 个窗口、1 个盒状容器和 4 个按钮。这使得代码冗长且不便于维护。这里采用了 gtk_widget_show_all 函数，它的参数是一个容器控件的指针，它取代了那 6 行代码(甚至更多行代码)，显示容器中所有控件。这样一来使代码更简洁明了，一看就懂。

如果认真阅读代码，这个运行结果应该在您的意料之中。因为创建“按钮一”时，设定是否扩展参数和是否填充参数为 TRUE，即允许扩展和填充。而“按”、“钮”、“二”这 3 个按钮都有设为 FALSE，不允许扩展和填充。这样拖长窗口后，“按钮一”会随之变长，而另外 3 个按钮则不会。

另外，还要注意向容器添加控件的过程，即先创建窗口，再创建盒状容器，将盒状容器加入到窗口中，最后创建按钮，将按钮排列到盒状容器中。一般创建后为控件的信号连接回调函数，几乎每个 GTK+2.0 程序都是按照这样的顺序完成界面布局的。

这段代码中只用到了横向盒状容器(GtkVBox)和 gtk_box_pack_start 函数，相信有兴趣的读者一定可以用纵向盒状容器(GtkHBox)和其他函数来编一段属于自己的代码。

1.5 用格状容器排列按钮

本节将介绍如何使用框架控件和能容纳多行多列控件的格状容器以及如何灵活的在格状容器中排放控件。

实例说明

盒状容器只能容纳一行或一列控件，如何容纳多行多列控件呢？向盒状容器中再添加盒状容器。对，这是创建多行多列控件布局的很好的方法。还有一种方法就是用格状容器(GtkTable)控件。

格状容器是一种能容纳多行多列控件且简单易用的容器，它提供了坐标方式表示控件所要摆放的空间的方法，使按行列方式排列多个控件变得更加简单。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/1
mkdir table
cd table
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 table.c 为文件名保存到当前目录下：

```
/* 格状容器 table.c */
#include <gtk/gtk.h>
int main    (int argc, char* argv[])
{
    GtkWidget* window;
    GtkWidget* table;
    GtkWidget* button;
    GtkWidget* frame;
        gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"格状容器");
    gtk_window_set_default_size(GTK_WINDOW(window),200,300);
    g_signal_connect(G_OBJECT(window),"destroy",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_container_set_border_width(GTK_CONTAINER(window),20);
        frame = gtk_frame_new("请注意下列按钮的排列");
    gtk_container_add(GTK_CONTAINER(window),frame);
    table = gtk_table_new(4,4,FALSE);
    gtk_container_set_border_width(GTK_CONTAINER(table),10);
    gtk_table_set_row_spacings(GTK_TABLE(table),5);
    gtk_table_set_col_spacings(GTK_TABLE(table),5);
    gtk_container_add(GTK_CONTAINER(frame),table);
        button = gtk_button_new();
    gtk_table_attach(GTK_TABLE(table),button,0,1,0,1,GTK_FILL,GTK_FI
LL,0,0);
    //0,0-1,1
        button = gtk_button_new();//_with_label("1,3,1,3");
    gtk_table_attach(GTK_TABLE(table),button,1,3,1,3,GTK_FILL,GTK_FI
LL,0,0);
    //1,1-3,3
    button = gtk_button_new();//_with_label("0,1,1,3");
    gtk_table_attach_defaults(GTK_TABLE(table),button,0,1,1,3);
    //0,1-1,3
    button = gtk_button_new(); //_with_label("1,3,0,1");
    gtk_table_attach_defaults(GTK_TABLE(table),button,1,3,0,1);
    //1,0-3,1
    button = gtk_button_new();//_with_label("0,4,3,4");
    gtk_table_attach_defaults(GTK_TABLE(table),button,0,4,3,4);
    //0,3-4,4
        button = gtk_button_new();//_with_label("3,4,0,3");
    gtk_table_attach_defaults(GTK_TABLE(table),button,3,4,0,3);
    //3,0-4,3
        gtk_widget_show_all(window);
    gtk_main();
```

```
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) table.c -o table 'pkg-config gtk+-2.0 --cflags --libs'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./table 即可运行此程序，运行结果如图 1.5 所示。

图 1.5 用格状容器排列按钮

实例分析

(1) 创建框架控件

框架控件(GtkFrame)和窗口一样是一种只能容纳一个控件的对象，但它不能象窗口那样直接显示出来，这一点又和盒状容器一样，必须把它放到窗口中才能显示出来。框架控件在 GTK+2.0 编程中主要起到美化、修饰窗口、规范控件的范围和提示的作用。可以用 gtk_frame _new 函数创建框架，参数是显示在框架上的标题文字。

(2) 创建格状容器

最关键的就是格状容器了，它用 gtk_table_new 创建，带 3 个参数，分别是行数、列数和是否均匀分布。如本例中的：

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

设定格状容器其为 4 行 4 列，且不允许均匀排放子控件。

(3) 排列控件

在格状容器中排列控件使用函数 gtk_table_attach 和 gtk_table_attach_defaults，它们的功能都是向格状容器的指定区域中添加控件。

函数 gtk_table_attach 有 10 个参数，前两个参数分别是格状容器和要放入到格状容器的控件的指针；接下来的 4 个参数是控件在格状容器中的位置坐标；最后 4 个参数是排放控

件的属性和间隔距离。其中第 7 和第 8 个参数的类型为 GtkAttachOptions，可以取 3 个值：GTK EXPAND，可扩展的；GTK_SHRINK，可缩小的；GTK_FILL，可添充的。本例中取值为 GTK_FILL。

这里最难理解的是格状容器的坐标，在格状容器中，从左上角开始按照坐标系的原则分别为(0，0)，横向依次为(1，0)、(2，0)，纵向依次为(0，1)、(0，2)，斜向依次为(1，1)、(2，2)。如本例中的：

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

表示向格状容器中的坐标(1，1)到(3，3)的区域内排放按钮(即中间的大按钮)，属性为添充。

还可以用函数 gtk_table_set_row_spacing 和 gtk_table_set_col_spacing 来设定格状容器中子控件的行间隔和列间隔。

(4) 快捷方法

事实上这只是一个演示格状容器的特例，如果向格状容器中排放的按钮(或控件)没有严格的大小区分(等大)，完全可以用下面代码快捷地创建并显示出来：

```
gint i,j ;
for(i=0; i<4; i++)
    for(j=0; j<4; j++)
    {
        button = gtk_button_new();
        gtk_table_attach_defaults (GTK_TABLE (table), button,
 i, i+1, j, j+1);
            gtk_widget_show(button);
    }
```

本示例演示了格状容器的一般用法，当设计的程序界面控件比较多时，就要考虑用格状容器来排放控件了。读者要结合运行结果认真分析代码，就会用好格状容器，设计出更漂亮的应用界面。

1.6　带图像和快捷键的按钮

本节将介绍如何创建图像控件以及如何自定义函数创建带图像和快捷键的按钮，为按钮的回调函数传递参数。

实例说明

只有文字的按钮有些单调，如何让按钮更漂亮呢？GTK+2.0 还允许创建带快捷键的按钮和从系统自带资源项目中创建带有图像、文字和快捷键按钮，这使编程方便多了。更好的是由于按钮也是一种容器(只能容纳一个控件)，我们完全可以利用按钮的这一特点创建出带有自己图像的按钮来。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/1
mkdir button
cd button
```

创建工作目录，并进入此目录开始编程。

到 GTK+2.0 的演示程序目录下(/usr/share/gtk+-2.0)把两个图像文件 apple-red.png 和 gnome-gmush.png 复制到此目录下。

(2) 打开编辑器，输入以下代码，以 button.c 为文件名保存到当前目录下：

```
/* 带图像的按钮 button.c */
#include <gtk/gtk.h>
void
on_button_clicked   (GtkWidget* button,gpointer data)
{
    //g_print("按钮 %s ",(gchar*)data);
    g_print("Button %s is pressed.\n",(gchar*)data);
    //g_print("被按了  下.  \n");
}
//创建自己按钮的函数
GtkWidget* create_button1 (void)
{
    GtkWidget* box;
    GtkWidget* image;
    GtkWidget* label;
    GtkWidget* button;
    char* title = "红苹果";
    image = gtk_image_new_from_file("apple-red.png");
    label = gtk_label_new(title);
    box  = gtk_vbox_new(FALSE,2);
    gtk_container_set_border_width(GTK_CONTAINER(box),5);
    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);
    button = gtk_button_new();
    gtk_container_add(GTK_CONTAINER(button),box);
    gtk_widget_show(box);
    return button ;
}
GtkWidget* create_button2 (void)
{
    GtkWidget* box;
    GtkWidget* image;
    GtkWidget* label;
    GtkWidget* button;
    char* title = "小蘑菇";
```

```
        image = gtk_image_new_from_file("gnome-gmush.png");
    label = gtk_label_new(title);
    box  = gtk_hbox_new(FALSE,2);
    gtk_container_set_border_width(GTK_CONTAINER(box),5);
    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);
    button = gtk_button_new();
    gtk_container_add(GTK_CONTAINER(button),box);
    gtk_widget_show(box);
    return button ;
}
//主函数
int
main    (int argc, char *argv[])
{
    GtkWidget* window;
    GtkWidget* box;
    GtkWidget* button1;
    GtkWidget* button2;
    GtkWidget* button3;
    GtkWidget* button4;
        gchar* title = "带图像和快捷键的按钮";
    gchar* b1 = "Red apple";//"红苹果"
    gchar* b2 = "Small mushroom";//"小蘑菇"
    gchar* b3 = "Short key";//"快捷键"
    gchar* b4 = "OK";//"确  认"
        gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),title);
    gtk_container_set_border_width(GTK_CONTAINER(window),20);
    g_signal_connect(G_OBJECT(window),"destroy",G_CALLBACK(gtk_main_
quit),NULL);
    box = gtk_hbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),box);
            button1 = create_button1();
    g_signal_connect(G_OBJECT(button1),"clicked",
            G_CALLBACK(on_button_clicked),(gpointer)b1);
    gtk_box_pack_start(GTK_BOX(box),button1,FALSE,FALSE,5);
    button2 = create_button2();
    g_signal_connect(G_OBJECT(button2),"clicked",
            G_CALLBACK(on_button_clicked),(gpointer)b2);
    gtk_box_pack_start(GTK_BOX(box),button2,FALSE,FALSE,5);
            button3 = gtk_button_new_with_mnemonic("快捷键(_H)");
    g_signal_connect(G_OBJECT(button3),"clicked",
            G_CALLBACK(on_button_clicked),(gpointer)b3);
    gtk_box_pack_start(GTK_BOX(box),button3,FALSE,FALSE,5);
    button4 = gtk_button_new_from_stock(GTK_STOCK_OK);
    g_signal_connect(G_OBJECT(button4),"clicked",
            G_CALLBACK(on_button_clicked),(gpointer)b4);
```

```
    gtk_box_pack_start(GTK_BOX(box),button4,FALSE,FALSE,5);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE ;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o button button.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./button 即可运行此程序，运行结果如图 1.6 和图 1.7 所示。

图 1.6 带图像和快捷键的按钮

图 1.7 程序在终端中的输出结果

实例分析

(1) 图像控件

图像控件在 GTK+2.0 编程中经常用到，图像控件最简单的创建方法之一就是直接从图像文件创建，使用函数 gtk_image_new_from_file，它只有一个参数就是要调用的图像文件名，返回结果是图像控件的指针。如本例中的：

```
image = gtk_image_new_from_file("apple-red.png");
```

GTK+2.0 自身带有一个叫 GDKPIXBUF 的程序库，GTK+2.0 使用它来支持多种图像文件格式，如 PNG、JPEG、BMP、PIX、GIF 等。

(2) 标签控件

多数情况下，标签文本在 GTK+2.0 编程中起到提示的作用。用 gtk_label_new 函数可以直接创建一个标签文本，它的参数就是标签要显示的文本字符串。事实上创建带有标签的按钮时，按钮中就包含了一个标签控件。

(3) 创建按钮

这里定义了两个函数来分别创建不同样式的按钮，其中 create_button1 创建的按钮中包含了一个纵向盒状容器来容纳一个图像和一个标签；create_button2 则用到了横向盒状容器。把图像和标签创建完后，用 gtk_box_pack_start 函数排放到盒状容器内，再将盒状容器添加到创建的空按钮内，返回按钮的指针，这样按钮一和按钮二就创建完了。由于返回的值是控件的指针，所以两个函数的返回类型均为 GtkWidget*。

第 3 个按钮的创建用到了函数 gtk_button_new_with_mnenonic。这个函数的参数是按钮要显示的文本，在字母前加一个下划线如“_Help”，则表示此按钮显示的文字为“Help”，快捷键为 Alt+H，本例中就使用此快捷键。

第 4 个按钮的创建用到了函数 gtk_button_new_from_stock，它的意思是从系统自带的资源项目中创建按钮，它只有一个参数，表示以系统自带资源(包括文字、图像和快捷键)的一个字符串命名。GTK+2.0 中用一系列以 GTK_STOCK_ 开头的宏表示，如 GTK_STOCK_CANCEL 表示取消，GTK_STOCK_ADD 表示增加。更多的信息，请参见 GTK+2.0 的 API 参考手册。

(4) 多函数实现

为编写方便，我们将上述两函数放到了主函数的前面。如果在主函数后面定义或在外部文件中定义，则一定要在主函数前面加上函数声明，格式如下：

```
GtkWidget* create_button1  (void);
GtkWidget* create_button2  (void);
```

(5) 回调函数的参数

这段代码中定义了一个回调函数 on_button_clicked，它的主要功能是向终端输出一行文字信息。而且 4 个按钮的单击信号都连接了这个回调函数。为了让输出内容不同，每个按钮连接信号时传递给回调函数的参数都是不同的，如第 1 个按钮传递的参数是字符串 b1，“Red apple”(即“红苹果”)，第 2 个按钮的参数则是 b2，“Small mushroom”(即“小蘑菇”)，第 3 个按钮的参数是 b3，“Short key”(即“快捷键”)，第 4 个按钮的参数是 b4，“OK”(即确认)。应该注意的是一定要将字符串强制转换成 gpointer 类型，如(gpointer)b4。因为回调函数接收的参数也是 gpointer 类型的，它为了把字符串显示出来，还需将这个指针转换为字符串型，如(gchar*)data。另外如果您的终端支持中文的话，参数可以直接设为中文。

通过本例的学习，读者会发现容器在 GTK+2.0 编程的界面设计中起着非常重要的作用。灵活运用容器可以创建出多种多样的界面和不同外观的控件来。

1.7 方向按钮

本节将介绍如何使用箭头控件和利用箭头控件创建方向按钮。

实例说明

GTK+2.0 中提供了一种最简单易用的箭头控件(GtkArrow)，箭头控件经常和按钮一起

使用，构成方向按钮。箭头控件是一个表面显示表示 4 个基本方向箭头的控件。它的特点是能扩展到分配给它的所有容器空间，正好对准或填充到编程者想要的容器空间内。

实现步骤

(1) 打开终端输入如下命令：

```
cc ~/ourgtk/1
mkdir arrow
cd arrow
```

创建工作目录，开始编程工作。

(2) 打开编辑器，输入以下代码，以 arrow.c 为文件名保存到当前目录下：

```
/* 方向按钮  arrow.c */
#include <gtk/gtk.h>
GtkWidget*
create_arrow_button (GtkArrowType arrowtype,GtkShadowType shadowtype)
{
    GtkWidget* button;
    GtkWidget* arrow;
        button = gtk_button_new();
    arrow = gtk_arrow_new(arrowtype,shadowtype);
    gtk_container_add(GTK_CONTAINER(button),arrow);
    gtk_widget_show(arrow);
    return button;
}
//主函数
int
main    (int argc, char *argv[])
{
    GtkWidget* window;
    GtkWidget* box;
    GtkWidget* arrow1;
    GtkWidget* arrow2;
    GtkWidget* arrow3;
    GtkWidget* arrow4;
    char* title = "方向按钮";
            gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),title);
    gtk_container_set_border_width(GTK_CONTAINER(window),20);
    g_signal_connect(G_OBJECT(window),"destroy",G_CALLBACK(gtk_main_
quit),NULL);
    box = gtk_hbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),box);
    gtk_widget_show(box);
        arrow1 = create_arrow_button(GTK_ARROW_LEFT,GTK_SHADOW_IN);
    gtk_box_pack_start(GTK_BOX(box),arrow1,FALSE,FALSE,13);
    gtk_widget_show(arrow1);
        arrow2 = create_arrow_button(GTK_ARROW_UP,GTK_SHADOW_IN);
```

```
    gtk_box_pack_start(GTK_BOX(box),arrow2,FALSE,FALSE,13);
    gtk_widget_show(arrow2);
        arrow3 = create_arrow_button(GTK_ARROW_DOWN,GTK_SHADOW_IN);
    gtk_box_pack_start(GTK_BOX(box),arrow3,FALSE,FALSE,13);
    gtk_widget_show(arrow3);
    arrow4 = create_arrow_button(GTK_ARROW_RIGHT,GTK_SHADOW_IN);
    gtk_box_pack_start(GTK_BOX(box),arrow4,FALSE,FALSE,13);
    gtk_widget_show(arrow4);
        gtk_widget_show(window);
    gtk_main();
    return FALSE ;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o arrow arrow.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./arrow 即可运行此程序，运行结果如图 1.8 所示。

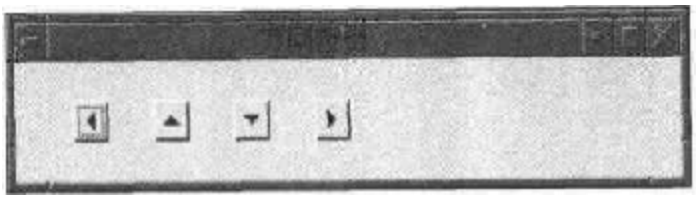

图 1.8　方向按钮

实例分析

(1) 创建箭头控件

用函数 gtk_arrow_new 来创建箭头控件，它有两个参数，分别为设定方向按钮 arrow 的按钮类型和阴影类型。其中设定箭头类型 GtkArrowType 取值为：GTK_ARROW_UP，向上；GTK_ARROW_DOWN，向下；GTK_ARROW_LEFT，向左；GTK_ARROW_RIGHT，向右。设定箭头的阴影类型 GtkShadowTye 取值为：GTK_SHADOW_NONE，无阴影；GTK_SHADOW_IN，阴影向内；GTK_SHADOW_OUT，阴影向外；GTK_SHADOW_ETCHED_IN，阴影向内凹进；GTK_SHADOW_ETCHED_OUT，阴影向外凸起。

(2) 创建方向按钮

这段代码中自定义了一个函数 create_arrow_button 来创建方向按钮，它的两个参数就是创建箭头控件的那两个参数。在这个函数中定义了两个控件指针 button 和 arrow，分别表示按钮和箭头，先创建一个空按钮，再创建箭头，将箭头控件添加到按钮中，显示箭头控件，最后返回按钮的指针。这样在主函数中直接调用创建方向按钮函数，加上不同的参数就可以创建出不同样式的方向按钮了。

箭头控件是 GTK+2.0 中最简单易用的控件，尤其是在一些组合型控件中经常用到。读者还可以更改创建箭头控件的阴影类型参数，编译后看看不同阴影类型的效果。

1.8 创建不同样式的标签

本节将介绍 GTK+2.0 中标签控件(GtkLable)的使用技巧。

实例说明

做几个漂亮的文字标签，会使程序更加吸引人。本示例向读者展示了如何创建多行的、不同对齐方式的、不同颜色和不同字体的文字标签。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/1
mkdir label
cd label
```

创建工作目录，开始编程。

(2) 打开编辑器，输入以下代码，以 label.c 为文件名保存到当前目录下：

```
/* 多种样式的标签 label.c*/
#include <gtk/gtk.h>
int
main    (int argc, char* argv[])
{
    GtkWidget* window;
    GtkWidget* box;
    GtkWidget* label1;
    GtkWidget* label2;
    GtkWidget* label3;
    GtkWidget* label4;
    GtkWidget* frame1;
    GtkWidget* frame2;
    GtkWidget* frame3;
    GtkWidget* frame4;
    gchar* title;
        gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"多种样式的标签");
    g_signal_connect(G_OBJECT(window),"destroy",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_container_set_border_width(GTK_CONTAINER(window),20);
        box = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),box);
        frame1 = gtk_frame_new("标签 一");
    label1 = gtk_label_new("这是第一个标签，居左的. This is the first
label.");
    gtk_container_add(GTK_CONTAINER(frame1),label1);
    gtk_label_set_justify(GTK_LABEL(label1),GTK_JUSTIFY_LEFT);
```

```
    gtk_box_pack_start(GTK_BOX(box),frame1,FALSE,FALSE,5);
    frame2 = gtk_frame_new("标签二");
    label2 =
gtk_label_new("这是第二个标签，它是多行的，\n这还是第二个标签的内容，它是居右
的。");
    gtk_container_add(GTK_CONTAINER(frame2),label2);
    gtk_label_set_justify(GTK_LABEL(label2),GTK_JUSTIFY_RIGHT);
    gtk_box_pack_start(GTK_BOX(box),frame2,FALSE,FALSE,5);
    frame3 = gtk_frame_new("标签三");
    label3 = gtk_label_new(NULL);
    title = "<span foreground=\"red\"
><big><i>这是第三个标签，\n它被格式化成红色的了，并且字体也大
了。</i></big></span>";
    gtk_label_set_markup(GTK_LABEL(label3),title);
    gtk_container_add(GTK_CONTAINER(frame3),label3);
    gtk_box_pack_start(GTK_BOX(box),frame3,FALSE,FALSE,5);
    frame4 = gtk_frame_new("标签四");
    label4 =
gtk_label_new("这也是一个多行标签，它的换行方式和上一个有所不同，主要是编程手段
不一样了，请详细查看一下源码就会明白是怎么回事了。");
    gtk_container_add(GTK_CONTAINER(frame4),label4);
    gtk_label_set_line_wrap(GTK_LABEL(label4),TRUE);
    gtk_box_pack_start(GTK_BOX(box),frame4,FALSE,FALSE,0);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) label.c -o label 'pkg-config gtk+-2.0 --cflags --libs'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./label 即可运行此程序，运行结果如图 1.9 所示。

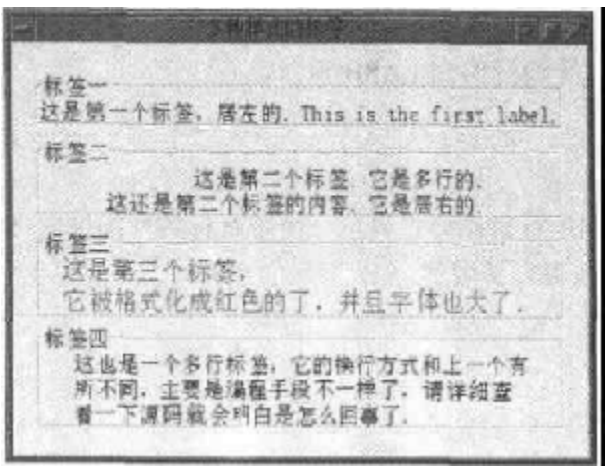

图 1.9　各种样式的标签

实例分析

(1) 标签的对齐方式

设定标签文字的对齐方式用到下面函数：

```
gtk_label_set_justify(GTK_LABEL(label1),GTK_JUSTIFY_LEFT);
```

其后一个参数类型为 GtkJustifaction，表示标签文字的对齐方式。可取值如下：GTK_JUSTIFY_LEFT 居左；GTK_JUSTIFY_RIGHT 居右；GTK_JUSTIFY_CENTER 居中；GTK_JUSTIFY_FILL 均匀分布。

(2) 格式化标签文本

GTK+2.0 中用到了格式化标签文本的功能：

```
gtk_label_set_markup(GTK_LABEL(label3),title);
```

其中 title 为标签文字的内容，这一内容要写成标记语言形式，有点类似 HTML。如：每段文本都要用和2 个标记括起来，表示斜体字，表示粗体字，而且在标记中还可以加参数，foreground 表示前景颜色，background 表示背景颜色，注意赋给这些参数的值应该用双引号引起来，但不能直接用(会引起 C 语言字符串歧意)，要用反斜杠来转意，这样就成了<span foreground = \"red\">这种形式，详细用法请参见 pango 的参考手册的 pangomarkupformat.html 一节。

(3) 自动换行

我们可以在字符串中加入符号"\n"来表示换行，也可以设定标签文字自动换行，用到下面的函数：

```
gtk_label_set_line_wrap(GTK_LABEL(label4),TRUE);
```

只要将后一参数值设为 TRUE 就可以了。

以上只是对标签文字控件的简单设置，还可以将标签格式化为其他国家的语言，另外还可以设定标签文字为可选择方式，这样选择后可以单击右键菜单进行复制等等。

1.9 Splash 窗口

本节介绍如何创建无边框的 SPLASH 窗口。

实例说明

很多程序在一开始运行时都会先显示一个带有标致性图像的 SPLASH 窗口来展示自己，这在 GTK+2.0 中也很容易做到。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/1
mkdir splash
```

```
cd splash
```

创建工作目录，并进入此目录开始编程。

到图像资源目录(/usr/share/pixmaps/splash)下找到图像文件 gnome-splash.png 复制到当前目录下，用 GIMP 自己绘制一幅图像就更好了，不过得用 gnome-splash.png 为文件名保存到此目录。

(2) 打开编辑器，输入以下代码，以 splash.c 为文件名保存到当前目录下：

```
/* 创建SPLASH窗口 splash.c */
#include <gtk/gtk.h>
//主函数
int
main (int argc,char* argv[])
{
    GtkWidget* window ;
    GtkWidget* vbox ;
    GtkWidget* image ;
    GtkWidget* button ;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_POPUP);
    gtk_window_set_title(GTK_WINDOW(window),"Splash窗口");
    g_signal_connect(G_OBJECT(window),"destroy",
          G_CALLBACK(gtk_main_quit),NULL);
    //gtk_container_set_border_width(GTK_CONTAINER(window),20);
    //gtk_window_set_default_size(GTK_WINDOW(window),500,400);
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    image = gtk_image_new_from_file("gnome-splash.png");
    gtk_box_pack_start(GTK_BOX(vbox),image,FALSE,FALSE,0);
    button = gtk_button_new_with_label("Splash窗口");
    g_signal_connect(G_OBJECT(button),"clicked",
                  G_CALLBACK(gtk_main_quit),NULL);
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,0);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o splash splash.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./splash 即可运行此程序，运行结果如图 1.10 所示。

图 1.10 SPLASH 窗口

实例分析

(1) 创建无边框窗口

在调用函数 gtk_window_new 时加参数为 GTK_WINDOW_POPUP，此时创建的窗口显示时没外框。

(2) 注释过的语句

代码中有两行注释过的语句，它们的功能是设定窗口边框的宽度和默认的尺寸。如果加上，运行效果就会不同。读者可以去掉注释编译一下，认真体会理解这两个函数的作用。

SPLASH 窗口是 GUI 编程中的一个小技巧，在 GTK+2.0 中写它的代码行数几乎是最短的，这体现 GTK+2.0 设计的最初思想，即用最简洁的代码创造最完美实用的功能。

本章介绍了 GTK+2.0 中界面设计的最基础内容。包括 GTK+2.0 程序的一般结构、窗口、容器、常用控件按钮、图像、框架、标签等的用法和回调函数的定义和添加等。下一章中我们将介绍应用程序中经常用到的菜单、工具条、状态栏、提示信息等控件的用法。

第 2 章　菜单与工具栏

本章重点：

完整的图形界面应用程序应该包括主窗口、菜单条、工具条，状态栏、窗口的主体区域等元素。GTK+2.0 能创建出完整的应用程序窗口，而且每个窗口还带有特定的浮动窗口控件、条件菜单控件等。本章将详细介绍应用程序界面的各种控件的创建与使用方法。

本章主要内容：

- 各种样式菜单的创建与使用
- 工具条、状态栏的创建与使用
- 条件菜单、浮动窗口的创建与使用
- 弹出式菜单和动态菜单操作
- 用多个文件来维护一个完整的应用程序

2.1　添 加 菜 单

本节将介绍如何为窗口添加菜单条，为菜单条添加多个菜单项，以及为菜单项的信号添加回调函数。

实例说明

菜单在图形界面编程中经常用到，稍复杂的应用程序一般都有多个菜单项来执行各种任务。GTK+2.0 提供了一套非常易用的设计菜单的方法，使用几行代码就可以为程序添加多个菜单项。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk
mkdir 2
cd 2
mkdir menu
cd menu
```

在用户目录下的总目录下创建本章的工作目录 2，进入工作目录 2，创建本节的工作目录 menu，并进入此目录开始编程。

(2) 打开编辑器，输入以下的代码，以 menu.c 为文件名保存到当前目录下：

```
/* 添加菜单 menu.c */
#include <gtk/gtk.h>
void on_menu_activate  (GtkMenuItem* item,gpointer data)
```

```
{
    //g_print("菜单项 %s 被激活\n",(gchar*)data);
    g_print("Menuitem %s is pressed.\n",(gchar*)data);
}
int main    (int argc, char* argv[])
{
    GtkWidget* window;
    GtkWidget* box;
    GtkWidget* menubar;
    GtkWidget* menu;
    GtkWidget* editmenu;
    GtkWidget* helpmenu;
    GtkWidget* rootmenu;
    GtkWidget* menuitem;
    GtkAccelGroup* accel_group ;
        gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"菜单测试程序");
    g_signal_connect(G_OBJECT(window),"destroy",
            G_CALLBACK(gtk_main_quit),NULL);
    accel_group = gtk_accel_group_new();
    gtk_window_add_accel_group(GTK_WINDOW(window),accel_group);
        box = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),box);
            menu = gtk_menu_new();//文件菜单
        menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_NEW,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_menu_activate),//(gpointer)("新建"));
            (gpointer)("New"));
    menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_OPEN,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_menu_activate),//(gpointer)("打开"));
            (gpointer)("Open"));
        menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_SAVE,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_menu_activate),//(gpointer)("保存"));
            (gpointer)("Save"));
    menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_SAVE_AS,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_menu_activate),//(gpointer)("另存为"));
            (gpointer)("Save As"));
        menuitem = gtk_separator_menu_item_new();
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
```

```
        menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_QUIT,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_menu_activate),//(gpointer)("退出"));
            (gpointer)("Exit"));
        rootmenu = gtk_menu_item_new_with_label(" 文件 ");
            gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootmenu),menu);
        menubar = gtk_menu_bar_new();
    gtk_menu_shell_append(GTK_MENU_SHELL(menubar),rootmenu);
        rootmenu = gtk_menu_item_new_with_label(" 编辑 ");
    editmenu = gtk_menu_new();//编辑菜单
    menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_CUT,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(editmenu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_menu_activate),//(gpointer)("剪切"));
            (gpointer)("Cut"));
        menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_COPY,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(editmenu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_menu_activate),//(gpointer)("复制"));
            (gpointer)("Copy"));
    menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_PASTE,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(editmenu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_menu_activate),//(gpointer)("粘贴"));
            (gpointer)("Paste"));
        menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_FIND,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(editmenu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_menu_activate),//(gpointer)("查找"));
            (gpointer)("Search"));
    gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootmenu),editmenu);
    gtk_menu_shell_append(GTK_MENU_SHELL(menubar),rootmenu);
        rootmenu = gtk_menu_item_new_with_label(" 帮助 ");
    helpmenu = gtk_menu_new();
        menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_HELP,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(helpmenu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_menu_activate),//(gpointer)("帮助"));
            (gpointer)("Help"));
    menuitem = gtk_menu_item_new_with_label(" 关于... ");
    gtk_menu_shell_append(GTK_MENU_SHELL(helpmenu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_menu_activate),//(gpointer)("关于"));
            (gpointer)("About"));
```

```
    gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootmenu),helpmenu);
    gtk_menu_shell_append(GTK_MENU_SHELL(menubar),rootmenu);
    gtk_box_pack_start(GTK_BOX(box),menubar,FALSE,FALSE,0);
        gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) menu.c -o menu 'pkg-config gtk+-2.0 --cflags --libs'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./menu 即可运行此程序，运行结果如图 2.1 所示。

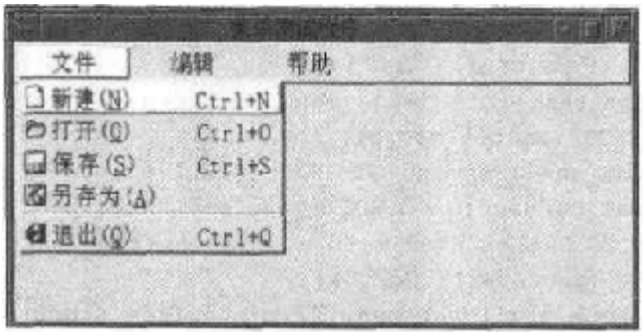

图 2.1 程序运行结果

实例分析

(1) 向窗口中添加菜单的步骤

在 GTK+2.0 中向窗口中添加菜单的步骤一般是：a.创建菜单条(GtkMenuBar)，将菜单条加入到窗口中去；b.创建菜单(GtkMenu)，加入到菜单条中；c.创建菜单项(GtkMenuItem)，加入到菜单中。其前提是先创建一个快捷键集(GtkAccelGroup)加入到窗口中(注意：它是一个非可视对象)。

(2) 用到的函数

创建快捷键集：gtk_accel_group_new()

创建菜单条：gtk_menu_bar_new()

创建菜单：gtk_menu_new()

向菜单条加菜单：gtk_menu_shell_append()

设定菜单项的子菜单：gtk_menu_item_set_submenu()

注意： 所有与菜单有关控件的根类都为 GtkMenuShell，它不能直接创建。

(3) 创建不同样式的菜单项

创建只带文字的菜单项：gtk_menu_item_new_with_label()

从系统资源中创建带图像的菜单项：gtk_image_menu_item_new_from_stock()

创建菜单中的横线：gtk_separator_menu_item_new()

创建菜单中的虚线：gtk_tearoff_menu_item_new()

后两个函数本示例中并未出现，读者可试着加进去。

(4) 菜单项的信号

菜单项的信号不多，最常用的是“activate”信号，表示菜单被激活(被单击)，也用g_signal_connect 宏来连接，格式同按钮控件的“clicked”信号。回调函数的格式如例中所示。

菜单项还有另外两种形式，单选菜单项和多选菜单项。有兴趣的读者可以自己试着做一下。

2.2　创建菜单的快捷方法

本节将介绍如何使用 GTK+2.0 中提供的 GtkItemFactory 对象快捷的创建菜单。

实例说明

上例中代码仍然比较复杂，GTK+2.0 中提供了更为简洁的创建菜单的方法，那就是GtkItemFactory。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/2
mkdir itemfact
cc itemfact
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入如下代码，以 itemfact.c 为文件名保存到当前目录下：

```
/* 创建菜单的快捷方法 itemfact.c */
#include <gtk/gtk.h>
void on_menu_activate  (GtkMenuItem* item,gpointer data);//回调函数声
明
static GtkItemFactoryEntry menu_items[] = {//定义菜单集
{"/ 文件 (_F) ",NULL,NULL,0,"<Branch>"},
{"/ 文件 (_F) /新建",NULL,on_menu_activate,"新建",
"<StockItem>",GTK_STOCK_NEW},
{"/ 文件 (_F) /打开",NULL,on_menu_activate,"打开",
"<StockItem>",GTK_STOCK_OPEN},
{"/ 文件 (_F) /保存",NULL,on_menu_activate,"保存",
"<StockItem>",GTK_STOCK_SAVE},
{"/ 文件 (_F) /另存为",NULL,on_menu_activate,"另存为",
"<StockItem>",GTK_STOCK_SAVE_AS},
{"/ 文件 (_F) /-",NULL,NULL,0,"<Separator>"},
{"/ 文件 (_F) /退出",NULL,on_menu_activate,"退出",
"<StockItem>",GTK_STOCK_QUIT},
```

```
{"/ 编辑 (_E) ",NULL,NULL,0,"<Branch>"},
{"/ 编辑 (_E) /剪切",NULL,on_menu_activate,"剪切",
"<StockItem>",GTK_STOCK_CUT},
{"/ 编辑 (_E) /复制",NULL,on_menu_activate,"复制",
"<StockItem>",GTK_STOCK_COPY},
{"/ 编辑 (_E) /粘贴",NULL,on_menu_activate,"粘贴",
"<StockItem>",GTK_STOCK_PASTE},
{"/ 编辑 (_E) /查找",NULL,on_menu_activate,"查找",
"<StockItem>",GTK_STOCK_FIND},
{"/ 帮助 (_H) ",NULL,NULL,0,"<LastBranch>"},
{"/ 帮助 (_H) /帮助",NULL,on_menu_activate,"帮助",
"<StockItem>",GTK_STOCK_HELP},
{"/ 帮助 (_H) /关于...",NULL,on_menu_activate,"关于",NULL}
};
void on_menu_activate  (GtkMenuItem* item,gpointer data)
{
    //g_print("菜单项");
    //g_print(" %s ",(gchar*)data);
    //g_print("被激活\n");
    g_print("Menu item %s is pressed.\n",(gchar*)data);
}
int main    (int argc,char *argv[])
{
    GtkWidget* window;
    GtkWidget* box;
    GtkWidget* menubar ;
    GtkAccelGroup* accel_group ;
    GtkItemFactory* item_factory;
        gint n = 15;
        gtk_init(&argc,&argv);/* 关键,如果不写此行,有时会出问题 */
        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"添加菜单的快捷方法");
    g_signal_connect(G_OBJECT(window),"destroy",
            G_CALLBACK(gtk_main_quit),NULL);
        accel_group = gtk_accel_group_new();
        box = gtk_vbox_new(FALSE,0);
    gtk_container_add (GTK_CONTAINER (window), box);
    gtk_widget_show(box);
        item_factory = gtk_item_factory_new
(GTK_TYPE_MENU_BAR,"<main>",accel_group);
    gtk_item_factory_create_items(item_factory,n,menu_items,NULL);
    gtk_window_add_accel_group(GTK_WINDOW(window),accel_group);
    menubar = gtk_item_factory_get_widget (item_factory, "<main>");
        gtk_box_pack_start(GTK_BOX(box),menubar,FALSE,FALSE,0);
    gtk_widget_show(menubar);
        gtk_widget_show(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入如下代码：

```
CC = gcc
all:
    $(CC) itemfact.c -o itemfact 'pkg-config gtk+-2.0 --cflags --libs'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./itemfact 即可运行此程序，运行结果如图 2.2 所示。

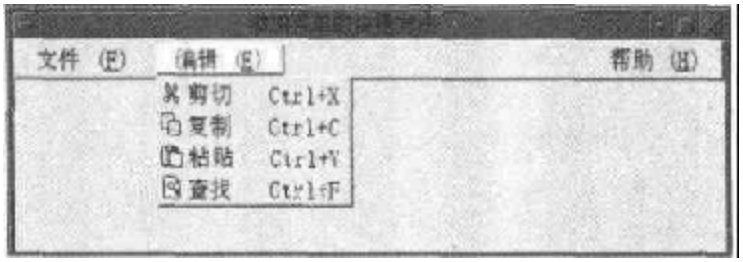

图 2.2　用快捷方式创建的菜单

实例分析

(1) 定义 GtkItemFactoryEntry

欲使用 GtkItemFactory，必需先定义 GtkItemFactoryEntry。它是一个多行的字符串数组，格式说明如表 2.1 所示。

表 2.1　GtkItemFactoryEntry 的格式

格　式	说　明
"/ 文件 (_F)/新建"	显示在菜单上文字以及文字所在的位置
NULL	快捷键
On_menu_activate	“activate”信号的回调函数
"新建"	传递给回调函数的参数
"<StockItem>"	此项的属性，<StockItem>表示系统固有资源
GTK_STOCK_NEW	属性的值，用 GTK_STOCK_*系列宏代替，如无属性可不设此值。

更详细的用法见 GTK+2.0 的 API 参考手册中的 GtkItemFactory 部分。

(2) 通过 GtkItemFactory 创建菜单

首先用 gtk_item_factory_new 函数来创建 GtkItemFactory，参数为已定义好的 GtkItem FactoryEntry。第 2 步用 gtk_item_factory_create_items 函数来创建各菜单项，第 3 步向窗口添加快捷键集 GtkAccelGroup，最后用 gtk_item_factory_get_widget 函数来取得菜单条的控件指针。整个创建菜单过程结束。

菜单项集为 GTK+2.0 编程提供了更为快捷的创建菜单的方法，而在功能上和 2.1 节的创建菜单的方法是一样的。

2.3　创建工具条

本节介绍如何创建和使用工具条控件，灵活的为工具条添加按钮、其他控件和为这些控件的信号添加回调函数。

实例说明

在图形界面编程中，工具条控件经常与菜单一起使用来快速执行某些经常用的菜单项的功能。本例中单独地创建了一个工具条，目的是让读者认真体验一下如何用 GTK+2.0 来创建工具控件，体验与工具条相关的函数的用法。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/2
mkdir toolbar
cd toolbar
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入如下代码，以 toolbar.c 为文件名保存到当前目录下：

```
/* 添加工具条 toolbar.c*/
#include <gtk/gtk.h>
void    on_button_clicked   (GtkButton* button,gpointer data)
{
    g_print("你点击的按钮是 %s \n",(gchar*)data);
}
int main    (int argc, char* argv[])
{
    GtkWidget* window;
    GtkWidget* box ;
    GtkWidget* toolbar ;
    GtkWidget* entry ;
    GtkWidget* label ;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"添加工具条");
    g_signal_connect(G_OBJECT(window),"destroy",
            G_CALLBACK(gtk_main_quit),NULL);
    box = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),box);
        toolbar = gtk_toolbar_new();
    gtk_toolbar_insert_stock(GTK_TOOLBAR(toolbar),GTK_STOCK_NEW,
                "新建一个文件","新建",
                GTK_SIGNAL_FUNC(on_button_clicked),
                (gpointer)("新建"),-1);
    gtk_toolbar_insert_stock(GTK_TOOLBAR(toolbar),GTK_STOCK_OPEN,
                "打开一个文件","打开",
                GTK_SIGNAL_FUNC(on_button_clicked),
                (gpointer)("打开"),-1);
    gtk_toolbar_insert_stock(GTK_TOOLBAR(toolbar),GTK_STOCK_SAVE,
                "保存当前文件","保存",
                GTK_SIGNAL_FUNC(on_button_clicked),
```

```
                (gpointer)("保存"),-1);
    gtk_toolbar_append_space(GTK_TOOLBAR(toolbar));
    label = gtk_label_new(" 文件名: ");
    gtk_toolbar_append_widget(GTK_TOOLBAR(toolbar),label,
                "这是一个标签","标签");
        entry = gtk_entry_new();
    gtk_toolbar_append_widget(GTK_TOOLBAR(toolbar),entry,
                "这是一个录入条","录入");
        gtk_toolbar_append_space(GTK_TOOLBAR(toolbar));
    gtk_toolbar_insert_stock(GTK_TOOLBAR(toolbar),GTK_STOCK_CUT,
                "剪切","剪切",
                GTK_SIGNAL_FUNC(on_button_clicked),
                (gpointer)("剪切"),-1);
    gtk_toolbar_insert_stock(GTK_TOOLBAR(toolbar),GTK_STOCK_COPY,
                "复制","复制",
                GTK_SIGNAL_FUNC(on_button_clicked),
                (gpointer)("复制"),-1);
    gtk_toolbar_insert_stock(GTK_TOOLBAR(toolbar),GTK_STOCK_PASTE,
                "粘贴","粘贴",
                GTK_SIGNAL_FUNC(on_button_clicked),
                (gpointer)("粘贴"),-1);
        gtk_toolbar_append_space(GTK_TOOLBAR(toolbar));
    gtk_toolbar_insert_stock(GTK_TOOLBAR(toolbar),GTK_STOCK_QUIT,
                "退出","退出",
                GTK_SIGNAL_FUNC(on_button_clicked),
                (gpointer)("退出"),-1);
    gtk_toolbar_set_style(GTK_TOOLBAR(toolbar),GTK_TOOLBAR_ICONS);
    gtk_toolbar_set_icon_size(GTK_TOOLBAR(toolbar),
                GTK_ICON_SIZE_SMALL_TOOLBAR);
        gtk_box_pack_start(GTK_BOX(box),toolbar,FALSE,FALSE,0);
        gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入如下代码：

```
CC = gcc
all:
    $(CC) toolbar.c -o toolbar 'pkg-config gtk+-2.0 --cflags --libs'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./toolbar 即可运行此程序，运行结果如图 2.3 所示。

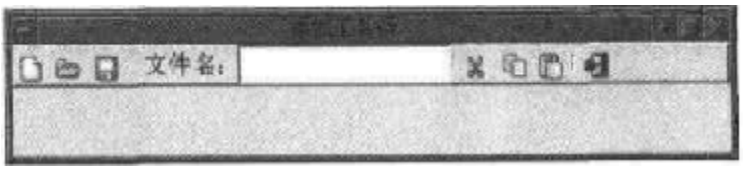

图 2.3　工具条

实例分析

(1) 创建工具条控件

用 gtk_toolbar_new()函数来创建工具条，创建后可添加到窗口或其他容器中。

(2) 向工具条中加按钮

可以用函数 gtk_toolbar_append_item 向工具条中添加带文字或图像的按钮，也可以是 preappend(向前添加)或 insert(插入)。

可以用函数 gtk_toolbar_insert_stock 向工具条中插入系统固有资源项目。如此例中所示，其第 1 个参数指出要添加控件的工具条，第 2 个参数指出系统固有资源项目，第 3、4 个参数指出按钮可用和不可用时的提示信息，第 5 个参数指出回调函数(用 GTK_SIGNAL_FUNC 宏转换一下)，第 6 个参数指出传给回调函数的参数，最后一个参数指出插入按钮的位置(-1 表示最后，NULL 表示依次)。

可以用函数 gtk_toolbar_append_space 向工具条中添加竖线间隔，向前添加和插入的方法同上。

(3) 向工具条中加其他控件

可以用函数 gtk_toolbar_append_element 向工具条中添加其他元素(如单选按钮、多选按钮等控件)，函数 gtk_toolbar_append_widget 向工具条中添加自己创建的控件，我们这里就添加了一个文字标签和一个单行文本录入控件，单行文本录入控件(GtkEntry)可以用函数 gtk_entry_new()来创建。

(4) 设定工具条的外观

用函数 gtk_toolbar_set_style 来设定工具条的外观，参数 GTK_TOOLBAR_ICONS 表示只显示图标。用函数 gtk_toolbar_set_icon_size 来设定工具条中图标的大小，参数 GTK_ICON_SIZE_SMALL_TOOLBAR 表示工具条中使用小图标。

工具条控件的使用让我们有机会编写更复杂的程序，而界面也不会变得零乱，一定要掌握其使用技巧。读者还可以试试其他函数，向工具条中加入更多的其他控件。

2.4 浮动的工具条和菜单

本节将介绍如何使用浮动窗口控件和提示信息对象以及如何创建自定义图像的按钮，为按钮加提示信息。

实例说明

在一些常用软件中都提供了一种浮动的工具条和菜单的功能，可以用鼠标拖动工具条和菜单在屏幕上移动，GTK+2.0 提供了浮动窗口控件(GtkHandleBox)来实现这种功能，本示例就向读者展示如何使用这种控件。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/2
mkdir handle
cd hand
```

创建工作目录，并进入此目录开始编程。找 4 个 GIF 格式的小图像分别表示新建、打开、保存、退出，复制到此目录下，依次命名为 new.gif、open.gif、save.gif、quit.gif。

(2) 打开编辑器，输入如下代码，以 handle.c 为文件名保存到当前目录下。

```
/* 浮动的工具条和菜单 handle.c */
#include <gtk/gtk.h>
//创建自己按钮的函数
GtkWidget* create_button (char *filename)
{
    GtkWidget* image;
    GtkWidget* button;
    image = gtk_image_new_from_file(filename) ;
    gtk_widget_show(image);
    button = gtk_button_new();
    gtk_container_add(GTK_CONTAINER(button),image) ;
    return button ;
}
//主函数
int main    (int argc, char *argv[])
{
    GtkWidget* window;
    GtkWidget* vbox;
    GtkWidget* hbox;
    GtkWidget* box;
    GtkWidget* box1;
    GtkWidget* handle;
    GtkWidget* handle1;
    GtkWidget* menubar;
    GtkWidget* button1;
    GtkWidget* button2;
    GtkWidget* button3;
    GtkWidget* button4;
    GtkWidget* menu;
    GtkWidget* menuitem;
    GtkWidget* rootmenu;
    GtkTooltips* button_tips;
            char* title = "浮动的工具条和菜单";
        gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),title);
    gtk_container_set_border_width(GTK_CONTAINER(window),5);
    g_signal_connect(G_OBJECT(window),"destroy",
                G_CALLBACK(gtk_main_quit),NULL);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
        hbox = gtk_hbox_new(FALSE,0);
```

```
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
        handle = gtk_handle_box_new();
    gtk_box_pack_start(GTK_BOX(hbox),handle,FALSE,FALSE,5);
        box = gtk_hbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(handle),box);
            button1 = create_button("new.gif") ;
    gtk_box_pack_start(GTK_BOX(box),button1,FALSE,FALSE,3);
    button2 = create_button("open.gif");
    gtk_box_pack_start(GTK_BOX(box),button2,FALSE,FALSE,3);
        button3 = create_button("save.gif");
    gtk_box_pack_start(GTK_BOX(box),button3,FALSE,FALSE,3);
    button4 = create_button("quit.gif");
    gtk_box_pack_start(GTK_BOX(box),button4,FALSE,FALSE,3);
    button_tips = gtk_tooltips_new();
    gtk_tooltips_set_tip(GTK_TOOLTIPS(button_tips),button1,
                "创建一个新文件","New");
        gtk_tooltips_set_tip(GTK_TOOLTIPS(button_tips),button2,
                "打开文件","Open");
    gtk_tooltips_set_tip(GTK_TOOLTIPS(button_tips),button3,
                "保存当前编辑的文件","Save");
        gtk_tooltips_set_tip(GTK_TOOLTIPS(button_tips),button4,
                "退出此程序","Quit");
    handle1 = gtk_handle_box_new();
    gtk_box_pack_start(GTK_BOX(hbox),handle1,FALSE,FALSE,5);
        box1 = gtk_hbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(handle1),box1);
        menubar = gtk_menu_bar_new();
    gtk_box_pack_start(GTK_BOX(box1),menubar,FALSE,FALSE,5);
        menu = gtk_menu_new();
    menuitem = gtk_menu_item_new_with_label("菜单项一");
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
        menuitem = gtk_menu_item_new_with_label("菜单项二");
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
        rootmenu = gtk_menu_item_new_with_label("  文件  ");
    gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootmenu),menu);
        gtk_menu_shell_append(GTK_MENU_SHELL(menubar),rootmenu);
        gtk_widget_show_all(window);
    gtk_main();
    return FALSE ;
}
```

(3) 编辑 Makefile 输入以下的代码：

```
CC = gcc
all:
    $(CC) -o handle handle.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./handle 即可运行此程序，运行结果如图 2.4 所示。

图 2.4　浮动的工具条和菜单

实例分析

(1) 创建浮动窗口

这里并未使用上节的工具条，而是用了一个横向盒状容器来容纳自己创建的按钮。用函数 gtk_handle_box_new()创建浮动窗口，然后直接将创建的盒状容器加入到浮动窗口中，菜单也一样。

(2) 创建使用提示信息

提示信息对象(GtkToolTips)不能用 GtkWidget 来声明，需要直接用 GtkToolTips 声明。用函数 gtk_tooltips_new()来创建，用函数 gtk_tooltips_set_tip 来为控件设置提示信息。此函数的第 1 个参数是提示信息对象本身，第 2 个参数是要加提示信息的控件指针，第 3、4 个参数分别是控件可用时和不可用时的提示信息字符串。

浮动窗口控件和提示信息控件在编程中经常用到，此示例只是简单的应用，读者可以把它加到自己的程序中去，或做出更强的功能扩展。

2.5　状　态　栏

本节将介绍如何创建状态栏控件和使用状态栏控件的技巧。

实例说明

一般应用程序的底部都有一个状态栏，用来显示程序当前执行的命令或当前的状态。GTK+2.0 同样为开发者提供了状态栏控件，本示例向读者演示了如何创建状态栏控件，如何向状态栏中加入信息、从状态栏中弹出信息。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/2
mkdir status
cd status
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入如下的代码，以 status.c 为文件名保存到当前目录下。

```
/* 状态栏  status.c */
#include <gtk/gtk.h>
static GtkWidget *statusbar;
```

```
gint i = 0 ;
void    on_push_clicked     (GtkButton* button,gpointer data)
{
    char message[1024];
        i++ ;
    sprintf(message,"这是输入的第 %d 条消息. ",i);
    gtk_statusbar_push(GTK_STATUSBAR(statusbar),i,message);
}
void    on_pop_clicked  (GtkButton* button,gpointer data)
{
    if (i<0) return ;
    gtk_statusbar_pop(GTK_STATUSBAR(statusbar),i);
    i--;
}
void    on_popped       (GtkStatusbar* statusbar,guint id,const gchar*
text)
{
    if(!statusbar->messages)
        i = 0;
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox,*hbox;
    GtkWidget *button;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"状态栏");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),20);
        vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
        statusbar = gtk_statusbar_new();
    g_signal_connect(G_OBJECT(statusbar),"text-popped",
                G_CALLBACK(on_popped),NULL);
    gtk_box_pack_start(GTK_BOX(vbox),statusbar,FALSE,FALSE,5);
        hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
        button = gtk_button_new_with_label("推入");
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(on_push_clicked),NULL);
    gtk_box_pack_start(GTK_BOX(hbox),button,TRUE,TRUE,5);
        button = gtk_button_new_with_label("弹出");
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(on_pop_clicked),NULL);
    gtk_box_pack_start(GTK_BOX(hbox),button,TRUE,TRUE,5);
    gtk_widget_show_all(window);
    gtk_main();
```

```
    return FALSE;
}
```

(3) 编辑 Makefile 输入如下代码：

```
CC = gcc
all:
    $(CC) -o status status.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./status 即可运行此程序，运行结果如图 2.5 所示。

图 2.5　状态栏

实例分析

(1) 创建状态栏的工作方式

首先，状态栏控件的指针在主函数外部定义，并且定义为静态的，其他函数可以非常容易地调用和控制此控件指针，这种方式在以后的单文件编程中经常用到，同时我们还定义了一个计数变量，来控制表示状态栏中的信息数量。

(2) 状态栏的工作方式

状态栏采用了一种栈示工作方式，当向状态栏加显示信息时用 gtk_statusbar_push 函数，删除当前信息时则用弹出函数 gtk_statusbar_pop，状态栏本身有一个“text-popped”信号，在消息弹出时发生。

GTK+2.0 的状态栏控件使用方式很有特色。掌握它的使用方法对理解 GTK+2.0 控件的工作原理很有帮助。

2.6　完整的应用程序窗口

本节示例综合以前学过的内容，介绍如何用多文件来创建完整的应用程序窗口。

实例说明

研究完上几节的控件功能后，做一个完整的应用程序界面已经很容易了。现在采用多文件的方式来做一个完整的应用程序界面，这对维护多文件的大型项目来说很有用。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/2
mkdir app
cd app
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，编辑以下 5 个文件保存到当前目录下。

① 主函数文件，文件名为 main.c，包含 C 语言的主函数

```
/* 主函数文件 main.c*/
#include <gtk/gtk.h>
#include "callbacks.h"
#include "interface.h"
int main (int argc , gchar* argv[])
{
    GtkWidget * window ;
    gtk_init(&argc,&argv);
    window = create_window();
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(on_window_delete_event),NULL);
        gtk_widget_show(window);
    gtk_main();
    return FALSE;
}
```

② 界面代码的头文件 interface.h

```
/* 界面代码的头文件 interface.h */
#ifndef __INTERFACE_H__
#define __INTERFACE_H__
GtkWidget*  create_window   ( void );
#endif
```

③ 界面代码文件 interface.c

```
/* 界面代码文件 interface.c */
#include <gtk/gtk.h>
#include "callbacks.h"
GtkWidget*  create_menu (GtkAccelGroup* accel_group,GtkWidget*
window);
GtkWidget*  create_toolbar  (GtkWidget* window);
GtkWidget*  create_window   (void)
{
    GtkWidget* window;
    GtkWidget* text;
    GtkWidget* scrolledwin;
        GtkWidget* box;
    GtkWidget* statusbar;
    GtkWidget* menubar ;
    GtkWidget* toolbar ;
    GtkAccelGroup* accel_group ;
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"完整的应用程序窗口");
```

```
    gtk_window_set_default_size(GTK_WINDOW(window),400,300);
        accel_group = gtk_accel_group_new();
    gtk_window_add_accel_group(GTK_WINDOW(window),accel_group);
        box = gtk_vbox_new(FALSE,0);
    gtk_container_add (GTK_CONTAINER (window), box);
        menubar = create_menu(accel_group,window);
    gtk_box_pack_start(GTK_BOX(box),menubar,0,0,0);
            toolbar = create_toolbar(window);
    gtk_box_pack_start(GTK_BOX(box),toolbar,0,1,0);
            scrolledwin = gtk_scrolled_window_new(NULL,NULL);
    text = gtk_text_view_new();
    gtk_box_pack_start(GTK_BOX(box),scrolledwin,TRUE,TRUE,0);
        gtk_container_add(GTK_CONTAINER(scrolledwin),text);
    gtk_text_view_set_editable(GTK_TEXT_VIEW(text),TRUE);
    statusbar = gtk_statusbar_new();
    gtk_box_pack_start(GTK_BOX(box),statusbar,FALSE,FALSE,0);
        gtk_widget_show_all(window);
    return window;
}
GtkWidget*  create_menu (GtkAccelGroup* accel_group,GtkWidget*
window)
{
    GtkWidget* menubar;
    GtkWidget* menu;
    GtkWidget* editmenu;
    GtkWidget* helpmenu;
    GtkWidget* rootmenu;
    GtkWidget* menuitem;
        menu = gtk_menu_new();//文件菜单
        menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_NEW,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_file_new_activate),NULL);
        menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_OPEN,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_file_new_activate),NULL);
            menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_SAVE,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
        menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_SAVE_AS,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
            menuitem = gtk_separator_menu_item_new();
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
            menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_QUIT,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
```

```
                G_CALLBACK(on_window_delete_event),NULL);
                rootmenu = gtk_menu_item_new_with_label("  文件  ");
        gtk_widget_show(rootmenu);

        gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootmenu),menu);
            menubar = gtk_menu_bar_new();
        gtk_menu_shell_append(GTK_MENU_SHELL(menubar),rootmenu);
            rootmenu = gtk_menu_item_new_with_label("  编辑  ");
        editmenu = gtk_menu_new();//编辑菜单
        menuitem = gtk_image_menu_item_new_from_stock
    (GTK_STOCK_CUT,accel_group);
        gtk_menu_shell_append(GTK_MENU_SHELL(editmenu),menuitem);
        g_signal_connect(G_OBJECT(menuitem),"activate",
                G_CALLBACK(on_edit_cut_activate),NULL);
                menuitem = gtk_image_menu_item_new_from_stock
    (GTK_STOCK_COPY,accel_group);
        gtk_menu_shell_append(GTK_MENU_SHELL(editmenu),menuitem);
        g_signal_connect(G_OBJECT(menuitem),"activate",
                G_CALLBACK(on_edit_copy_activate),NULL);
                menuitem = gtk_image_menu_item_new_from_stock
    (GTK_STOCK_PASTE,accel_group);
        gtk_menu_shell_append(GTK_MENU_SHELL(editmenu),menuitem);
        g_signal_connect(G_OBJECT(menuitem),"activate",
                G_CALLBACK(on_edit_paste_activate),NULL);
                menuitem = gtk_image_menu_item_new_from_stock
    (GTK_STOCK_FIND,accel_group);
        gtk_menu_shell_append(GTK_MENU_SHELL(editmenu),menuitem);
        g_signal_connect(G_OBJECT(menuitem),"activate",
                G_CALLBACK(on_edit_find_activate),NULL);
                        gtk_widget_show(rootmenu);
        gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootmenu),editmenu);
        gtk_menu_shell_append(GTK_MENU_SHELL(menubar),rootmenu);
            rootmenu = gtk_menu_item_new_with_label("  帮助  ");
        helpmenu = gtk_menu_new();
            menuitem = gtk_image_menu_item_new_from_stock
    (GTK_STOCK_HELP,accel_group);
        gtk_menu_shell_append(GTK_MENU_SHELL(helpmenu),menuitem);
        g_signal_connect(G_OBJECT(menuitem),"activate",
                G_CALLBACK(on_help_help_activate),NULL);
                    menuitem = gtk_menu_item_new_with_label("  关于
    ");
        gtk_menu_shell_append(GTK_MENU_SHELL(helpmenu),menuitem);
        g_signal_connect(G_OBJECT(menuitem),"activate",
                G_CALLBACK(on_help_about_activate),NULL);
        gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootmenu),helpmenu);
        gtk_menu_shell_append(GTK_MENU_SHELL(menubar),rootmenu);
            return menubar ;
    }
    GtkWidget*  create_toolbar  (GtkWidget* window)
    {
        GtkWidget* toolbar ;
```

```
    toolbar = gtk_toolbar_new();
    gtk_toolbar_insert_stock(GTK_TOOLBAR(toolbar),
                GTK_STOCK_NEW,
                "创建一个新文件","新建",
                GTK_SIGNAL_FUNC(on_file_new_activate),
                window,-1);
    gtk_toolbar_insert_stock(GTK_TOOLBAR(toolbar),
                GTK_STOCK_OPEN,
                "打开一个文件","打开",
                GTK_SIGNAL_FUNC(on_file_open_activate),
                toolbar,-1);
        gtk_toolbar_insert_stock(GTK_TOOLBAR(toolbar),
                GTK_STOCK_SAVE,
                "保存当前文件","保存",
                GTK_SIGNAL_FUNC(on_file_save_activate),
                toolbar,-1);
        return toolbar;
}
```

④ 回调函数头文件 callbacks.h

```
/* 回调函数头文件 callbacks.h */
#ifndef __CALLBACKS_H__
#define __CALLBACKS_H__
gboolean        on_window_delete_event  (GtkWidget* widget,
                    GdkEvent *event,
                    gpointer data);
void        on_file_new_activate    (GtkMenuItem* menuitem,
                    gpointer data);
void        on_file_open_activate   (GtkMenuItem* menuitem,
                    gpointer data);
void        on_file_save_activate   (GtkMenuItem* menuitem,
                    gpointer data);
void        on_file_saveas_activate (GtkMenuItem* menuitem,
                    gpointer data);
void        on_file_exit_activate   (GtkMenuItem* menuitem,
                    gpointer data);
void        on_edit_cut_activate    (GtkMenuItem* menuitem,
                    gpointer data);
void        on_edit_copy_activate   (GtkMenuItem* menuitem,
                    gpointer data);
void        on_edit_paste_activate  (GtkMenuItem* menuitem,
                    gpointer data);
void        on_edit_selectall_activate (GtkMenuItem* menuitem,
                    gpointer data);
void        on_edit_find_activate   (GtkMenuItem* menuitem,
                    gpointer data);
void        on_help_help_activate   (GtkMenuItem* menuitem,
                    gpointer data);
void        on_help_about_activate  (GtkMenuItem* menuitem,
                    gpointer data);
```

```
#endif
```

⑤ 回调函数文件 callbacks.c

```
/* 回调函数文件 callbacks.c */
#include <gtk/gtk.h>
gboolean
on_window_delete_event (GtkWidget* widget,
                        GdkEvent *event,
                        gpointer data)
{
    gtk_main_quit();
    return FALSE;
}
void
on_file_new_activate    (GtkMenuItem* menuitem,
                         gpointer data)
{
    g_print("the file_new is clicked .");
}
void        on_file_open_activate   (GtkMenuItem* menuitem,
                         gpointer data)
{
}
void        on_file_save_activate   (GtkMenuItem* menuitem,
                         gpointer data)
{
}
void        on_file_saveas_activate (GtkMenuItem* menuitem,
                         gpointer data)
{
}
void        on_file_exit_activate   (GtkMenuItem* menuitem,
                         gpointer data)
{
}
void        on_edit_cut_activate    (GtkMenuItem* menuitem,
                         gpointer data)
{
}
void        on_edit_copy_activate   (GtkMenuItem* menuitem,
                         gpointer data)
{
}
void        on_edit_paste_activate  (GtkMenuItem* menuitem,
                         gpointer data)
{
}
void        on_edit_selectall_activate (GtkMenuItem* menuitem,
                              gpointer data)
{
```

```
}
void        on_edit_find_activate   (GtkMenuItem* menuitem,
                                     gpointer data)
{
}
void        on_help_help_activate   (GtkMenuItem* menuitem,
                                     gpointer data)
{
}
void        on_help_about_activate  (GtkMenuItem* menuitem,
                                     gpointer data)
{
}
```

(3) 编辑 Makefile 输入如下代码：

```
CC = gcc
all:main.o callbacks.o interface.o
    $(CC) -o app main.o callbacks.o interface.o 'pkg-config --libs
gtk+2.0'
main.o:main.c interface.h callbacks.h
    $(CC) -c main.c 'pkg-config --cflags gtk+2.0'
interface.o:interface.c interface.h callbacks.h
    $(CC) -c interface.c 'pkg-config --cflags gtk+2.0'
callbacks.o:callbacks.c callbacks.h
    $(CC) -c callbacks.c 'pkg-config --cflags gtk+2.0'
clean:
    rm -f *.o
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./app 即可运行此程序，运行结果如图 2.6 所示。

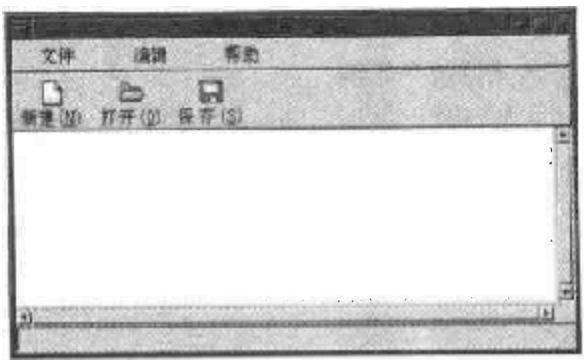

图 2.6　完整的应用程序窗口

实例分析

(1) 组织多个文件

我们把程序分成了 3 个主要部分。一是主程序，包括 C 语言主函数，是程序的运行的启动点；二是界面部分，由 interface.c 和 interface.h 组成，主要用来创建程序的界面；三是功能部分，由 callbacks.c 和 callbacks.h 组成，主要用来实现程序的各种功能。出于演示考

虑，此示例中只定义了各种回调函数的框架，并未加入实用代码。

在 interface.c 中定义了 3 个函数 create_toolbar、create_menu 和 create_window，其中前两个函数被后一个函数引用，外部并未用到，所以在 interface.h 中只声明了后一个创建窗口的函数。

(2) 维护 Makefile 文件

源程序的增多使得 Makefile 文件的行数也增多了，这里提供的 Makefile 并不是最简单的，但最容易理解，相信读者一看就懂。

用多文件来实现应用程序，一定要注意各源程序文件的头文件的定义和包含关系，以及各文件中变量的定义、外部变量的引用等，这对程序的功能的实现和程序代码的可维护性来说都是至关重要的。

2.7 动态菜单操作

本节示例介绍如何在程序运行时动态增加或删除菜单项。

实例说明

在应用程序中经常碰到在程序运行时需要增加或删除某些菜单项的情况，本示例就如何在 GTK+2.0 中实现此功能进行了尝试。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/2
mkdir addmenu
cd addmenu
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入如下代码，以 addmenu.c 为文件名保存到当前目录下。

```
/* 动态菜单操作 addmenu.c */
#include <gtk/gtk.h>
gint i = 2 ;
static GtkWidget *menu;
static GtkWidget *item ;
    void    add_menu    (GtkButton* button,gpointer data)
{
    gchar newitem[1024];
    gint len;
        len = g_list_length(GTK_MENU_SHELL(menu)->children);
    sprintf(newitem,"菜单项 %d",len+1);
    item = gtk_menu_item_new_with_label(newitem);
    gtk_widget_show(item);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),item);
    i++;
}
```

```
void    remove_menu (GtkButton* button,gpointer data)
{
    GList *list=NULL;
        if ( i == 0 ) return ;
    list = g_list_last(GTK_MENU_SHELL(menu)->children);
    GTK_MENU_SHELL(menu)->children = g_list_remove
(GTK_MENU_SHELL(menu)->children,list->data);
    i--;
}
//主函数
int main    (int argc, char *argv[])
{
    GtkWidget* window;
    GtkWidget* vbox;
    GtkWidget* hbox;
    GtkWidget* button;
    GtkWidget *rootmenu,*menubar; //,*item;
        gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"动态添加菜单");
    gtk_container_set_border_width(GTK_CONTAINER(window),5);
    g_signal_connect(G_OBJECT(window),"destroy",
                G_CALLBACK(gtk_main_quit),NULL);
        vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
        menubar = gtk_menu_bar_new();
    gtk_box_pack_start(GTK_BOX(vbox),menubar,FALSE,FALSE,5);
        menu = gtk_menu_new();
    rootmenu = gtk_menu_item_new_with_label("根菜单");
        item = gtk_menu_item_new_with_label("菜单项 1");
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),item);
        item = gtk_menu_item_new_with_label("菜单项 2");
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),item);
        gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootmenu),menu);
        gtk_menu_shell_append(GTK_MENU_SHELL(menubar),rootmenu);
        hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
        button = gtk_button_new_with_label("添加");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(add_menu),NULL);
    gtk_box_pack_start(GTK_BOX(hbox),button,TRUE,TRUE,5);
        button = gtk_button_new_with_label("删除");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(remove_menu),NULL);
    gtk_box_pack_start(GTK_BOX(hbox),button,TRUE,TRUE,5);
        gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入如下代码：

```
CC = gcc
all:
    $(CC) addmenu.c -o addmenu 'pkg-config gtk+-2.0 --cflags --libs'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./addmenu 即可运行此程序，运行结果如图 2.7 所示。

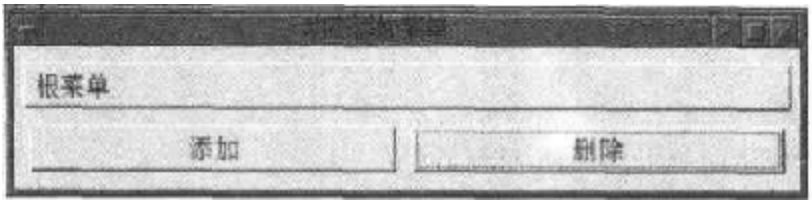

图 2.7 动态菜单操作

实例分析

(1) 添加

菜单控件的基础对象 GtkMenuShell 中有一个双向链表类型(GList)的指针 children，它是用来容纳菜单的子控件(菜单项)指针的，用相关的函数来对它进行操作，就可以得出某一菜单的菜单项数或具体某一菜单项。

在添加菜单时，用 g_list_length 取得菜单项的数量，再根据数量创建新的菜单项，显示出来后用函数 gtk_menu_shell_append 来向菜单添加就可以。

(2) 删除

删除时先定义一个双向链表，用函数 g_list_last 取得此链表最后一个元素的指针，再用函数 g_list_remove 将此指针移除，就实现了删除功能。

双向链表是 GLIB 程序库中提供的一系列常用数据结构之一，它贯穿了 GTK+2.0 的大多数复杂控件。读者一定要仔细研究一下 GLIB 的 API 参考手册，掌握更多数据结构的用法，这会大大提高您的编程水平。

2.8 条 件 菜 单

本节将介绍如何创建使用条件菜单控件。

实例说明

条件菜单是一种多项选一的控件，是 GTK+2.0 特有的，使用它的前提是先创建好一个菜单，然后再根据此菜单创建条件菜单控件。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/2
mkdir option
cd option
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入如下代码，以 optionmenu.c 为文件名保存到当前目录下。

```
/* 条件菜单 optionmenu.c */
#include <gtk/gtk.h>
static GtkWidget* entry;
void    on_menu_changed     (GtkOptionMenu* option,gpointer data)
{
    gchar buf[20];
    gint i;
        i = gtk_option_menu_get_history(option);
    sprintf(buf,"菜单项 %d",i+1);
    gtk_entry_set_text(GTK_ENTRY(entry),buf);
}
int main    (int argc, char* argv[])
{
    GtkWidget* window;
    GtkWidget* option;
    GtkWidget* hbox;
    GtkWidget* menu;
    GtkWidget* item;
    gint i ;
    gchar buf[20];
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"条件菜单");
    g_signal_connect(G_OBJECT(window),"destroy",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_container_set_border_width(GTK_CONTAINER(window),20);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),hbox);
        entry = gtk_entry_new();
    gtk_entry_set_text(GTK_ENTRY(entry),"菜单项 1");
    gtk_box_pack_start(GTK_BOX(hbox),entry,FALSE,FALSE,5);
    menu = gtk_menu_new();
    for(i=1; i<8; i++)
    {
        sprintf(buf,"菜单项 %d",i);
        item = gtk_menu_item_new_with_label(buf);
        gtk_menu_shell_append(GTK_MENU_SHELL(menu),item);
    }
        option = gtk_option_menu_new();
    gtk_option_menu_set_menu(GTK_OPTION_MENU(option),menu);
    g_signal_connect(G_OBJECT(option),"changed",
            G_CALLBACK(on_menu_changed),NULL);
    gtk_box_pack_start(GTK_BOX(hbox),option,FALSE,FALSE,10);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入如下代码：

```
CC = gcc
all:
    $(CC) optionmenu.c -o omenu 'pkg-config gtk+-2.0 --cflags --libs'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./omenu 即可运行此程序，运行结果如图 2.8 所示(右侧控件为条件菜单)。

图 2.8 条件菜单

实例分析

(1) 使用条件菜单

首先创建一个菜单，用 gtk_option_menu_new()函数创建一个空的条件菜单控件，再用函数 gtk_option_menu_set_menu 函数来设定条件菜单控件要显示的菜单，在程序运行中可用此函数来动态更换菜单。条件菜单控件本身有一个“changed”信号，在条件菜单的菜单项变化时发生，我们为此信号加了一个回调函数，使菜单项变化时，单行文本录入控件中的文字也随之变化。

(2) 单行文本录入控件

创建单行文本录入控件见第 2 章第 3 节工具条，函数 gtk_entry_set_text 用来设定单行文本录入控件的文本内容，函数 gtk_option_menu_get_history 用来返回条件菜单当前被选择的菜单项的索引值，通过这两个函数就可以实现此示例的核心功能了。

条件菜单的用法比较简单，在程序运行中有多种情况需要用户选择其中一种时较常用此控件，读者可根据软件功能灵活使用。

2.9 弹出式菜单

本节将介绍如何编程实现弹出式菜单，以及理解运用 GTK+2.0 的信号和 GDK 事件机制。

实例说明

单击鼠标右键弹出关联菜单在应用中经常用到，GTK+2.0 中实现这种功能需要一段代码，本示例就实现了在按钮上单击右键弹出菜单的功能。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/2
mkdir popup
cd popup
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下的代码，以 popup.c 为文件名保存到当前目录下。

```
/* 弹出式菜单 popup.c */
#include <gtk/gtk.h>
static gint //弹出菜单回调函数
my_popup_handler (GtkWidget *widget, GdkEvent *event)
{
    GtkMenu *menu;//菜单
    GdkEventButton *event_button;//要弹出菜单的对象
    g_return_val_if_fail (widget != NULL, FALSE);
    g_return_val_if_fail (GTK_IS_MENU (widget), FALSE);
    g_return_val_if_fail (event != NULL, FALSE);
    menu = GTK_MENU (widget);//转换为菜单
    if (event->type == GDK_BUTTON_PRESS)    //是此事件
      {
        event_button = (GdkEventButton *) event;
        if (event_button->button == 3)  //判断是否为右/三键
          {  //执行操作
            gtk_menu_popup (menu, NULL, NULL, NULL, NULL,
                            event_button->button, event_button->time);
            return TRUE;
          }
      }
    return FALSE;
}
int main    (int argc, char* argv[])
{
    GtkWidget* window;
    GtkWidget* box;
    GtkWidget* button;
    GtkWidget* menubar;
    GtkWidget* menu;
    GtkWidget* editmenu;
    GtkWidget* helpmenu;
    GtkWidget* rootmenu;
    GtkWidget* menuitem;
    GtkAccelGroup* accel_group;
        gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"弹出式菜单");
    g_signal_connect(G_OBJECT(window),"destroy",
            G_CALLBACK(gtk_main_quit),NULL);
    accel_group = gtk_accel_group_new();
    gtk_window_add_accel_group(GTK_WINDOW(window),accel_group);
        box = gtk_vbox_new(FALSE,0);
```

```
    gtk_container_add(GTK_CONTAINER(window),box);
            menu = gtk_menu_new();//文件菜单
        menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_NEW,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
        menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_OPEN,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
            menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_SAVE,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
        menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_SAVE_AS,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
    gtk_widget_show(menuitem);
        menuitem = gtk_separator_menu_item_new();
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
            menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_QUIT,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
            rootmenu = gtk_menu_item_new_with_label(" 文件 ");
        gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootmenu),menu);
        menubar = gtk_menu_bar_new();
    gtk_menu_shell_append(GTK_MENU_SHELL(menubar),rootmenu);
        rootmenu = gtk_menu_item_new_with_label(" 编辑 ");
    editmenu = gtk_menu_new();//编辑菜单
    menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_CUT,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(editmenu),menuitem);
            menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_COPY,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(editmenu),menuitem);
            menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_PASTE,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(editmenu),menuitem);
        menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_FIND,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(editmenu),menuitem);
                        gtk_widget_show(rootmenu);
    gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootmenu),editmenu);
    gtk_menu_shell_append(GTK_MENU_SHELL(menubar),rootmenu);
        rootmenu = gtk_menu_item_new_with_label(" 帮助 ");
    helpmenu = gtk_menu_new();
        menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_HELP,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(helpmenu),menuitem);
                menuitem = gtk_menu_item_new_with_label(" 关于... ");
    gtk_menu_shell_append(GTK_MENU_SHELL(helpmenu),menuitem);
    gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootmenu),helpmenu);
    gtk_menu_shell_append(GTK_MENU_SHELL(menubar),rootmenu);
```

```
    gtk_box_pack_start(GTK_BOX(box),menubar,FALSE,FALSE,5);
        //关键---弹出式菜单设计
    button = gtk_button_new_with_label("单击右键弹出菜单");
    g_signal_connect_swapped(GTK_OBJECT(button),
"button_press_event",
            G_CALLBACK (my_popup_handler), GTK_OBJECT (menu));
    gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,5);
        gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下的代码：

```
CC = gcc
all:
    $(CC) popup.c -o popup 'pkg-config gtk+-2.0 --cflags --libs'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./popup 即可运行此程序，运行结果如图 2.9 所示。

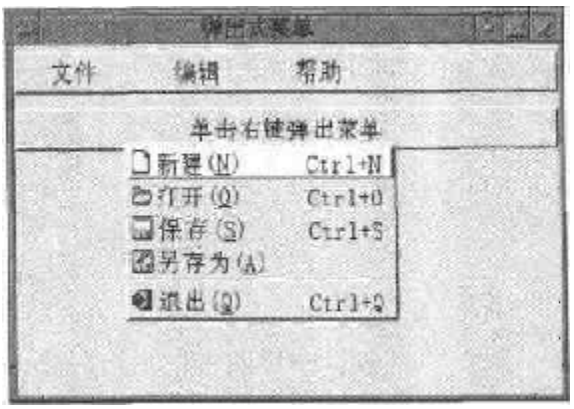

图 2.9　弹出式菜单

实例分析

(1) 控件的“button-press-event”信号

本示例的关键之一就是控件的“button-press-event”信号，几乎每一个继承自 GtkWidget 的控件都拥有这一信号，它在鼠标按下时发生，此信号的回调函数的第二个参数是 GdkEvent 类型，我们可以根据它来判断是否为鼠标右键按下。

宏 g_signal_connect_swapped 与 g_signal_connect 的格式完全一样，不同的是它一般放在 g_signal_connect 之后，表示执行完前一回调函数后继续执行此回调函数。

(2) 弹出菜单

本示例的另一关键所在就是函数 gtk_menu_popup 用来执行弹出菜单操作，其第 2～5 个参数为 NULL 时表示在鼠标当前位置弹出菜单，第 6 个参数表示被按下的按钮，最后一个参数是鼠标按下的时间。

弹出式菜单在应用编程中用途广泛，读者一定要深刻理解这一功能的实现过程，这一过程涉及到GTK+2.0的3个方面，即信号的连接、GDK事件和菜单的弹出操作，在GTK+2.0编程中都非常重要。

在本章中我们学习了菜单、工具条、状态栏、条件菜单、浮动窗口、提示信息等控件的详细用法以及菜单的动态和弹出操作、多文件实现的完整应用程序界面，这些是构筑应用程序的基础。

第3章 常用控件

本章重点：

学习了基础界面和应用界面后，对初学者来说想开发一些简单的软件还有些困难，主要是对还有些控件的使用和编程技巧不太熟练。本章将介绍一些较复杂控件的使用方法和常用的编程技巧。

本章主要内容：

- 介绍常用的容器控件按钮盒、自由布局、分隔面板的简单使用
- 如何调用/执行其他应用程序
- 列表框和下拉列表框的使用
- 框架控件的简单使用
- 图像控件的使用

3.1 按钮盒

本节将介绍如何创建按钮盒控件，设定按钮盒的属性以及如何向按钮盒中排列按钮控件等内容。

实例说明

按钮盒控件(GtkButtonBox)是按特定方式来容纳按钮的容器，它是盒状容器的扩展，也分为横向按钮盒(GtkHButtonBox)和纵向按钮盒(GtkVButtonBox)，在使用方法上与盒状容器类似，本节示例就向读者演示按钮盒的一般使用方法。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/
mkdir 3
cd 3
mkdir buttonbox
cd buttonbox
```

创建本章工作目录3，再创建本节工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 buttonbox.c 为文件名保存到当前目录下。

```
/* 按钮盒 buttonbox.c */
#include <gtk/gtk.h>
GtkWidget* create_button (gint i)
{
    GtkWidget *image;
```

```
    GtkWidget *button;
    GtkWidget *vbox;
    GtkWidget *label;
    switch(i)
    {
    case 1 :
        image = gtk_image_new_from_stock
(GTK_STOCK_YES,GTK_ICON_SIZE_MENU);
        label = gtk_label_new("是");
        break;
    case 2 :
        image =
gtk_image_new_from_stock(GTK_STOCK_NO,GTK_ICON_SIZE_MENU);
        label = gtk_label_new("否");
        break;
    case 3 :
        image = gtk_image_new_from_stock
(GTK_STOCK_HELP,GTK_ICON_SIZE_MENU);
        label = gtk_label_new("帮助");
        break;
    }
        button = gtk_button_new();
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(button),vbox);
    gtk_box_pack_start(GTK_BOX(vbox),image,FALSE,FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,0);
    return button;
}
GtkWidget*  create_button_box   (gboolean ish,
                GtkButtonBoxStyle style,
                const gchar* title)
{
    GtkWidget *frame;
    GtkWidget *button;
    GtkWidget *bbox;
        frame = gtk_frame_new(title);//"居中"));
    if(ish == TRUE)
        bbox = gtk_hbutton_box_new();
    else
        bbox = gtk_vbutton_box_new();
    gtk_box_set_spacing(GTK_BOX(bbox),5);
    gtk_button_box_set_layout(GTK_BUTTON_BOX(bbox),style);
    gtk_container_set_border_width(GTK_CONTAINER(bbox),5);
    gtk_container_add(GTK_CONTAINER(frame),bbox);

    button = create_button(1);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,2);
    button = create_button(2);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,2);
    button = create_button(3);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,2);
```

```
    return frame;
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *hbox;
    GtkWidget *frame;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"按钮盒");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);

    hbox = gtk_hbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),hbox);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(hbox),vbox,FALSE,FALSE,2);
    frame = create_button_box(TRUE,GTK_BUTTONBOX_EDGE,"居中");
    gtk_box_pack_start(GTK_BOX(vbox),frame,FALSE,FALSE,2);
    frame = create_button_box(TRUE,GTK_BUTTONBOX_SPREAD,"紧密");
    gtk_box_pack_start(GTK_BOX(vbox),frame,FALSE,FALSE,2);
    frame = create_button_box(TRUE,GTK_BUTTONBOX_END,"尾对齐");
    gtk_box_pack_start(GTK_BOX(vbox),frame,FALSE,FALSE,2);

    frame = create_button_box(TRUE,GTK_BUTTONBOX_START,"首对齐");
    gtk_box_pack_start(GTK_BOX(vbox),frame,FALSE,FALSE,2);
    frame = create_button_box(FALSE,GTK_BUTTONBOX_EDGE,"居中");
    gtk_box_pack_start(GTK_BOX(hbox),frame,FALSE,FALSE,2);
    frame = create_button_box(FALSE,GTK_BUTTONBOX_SPREAD,"紧密");
    gtk_box_pack_start(GTK_BOX(hbox),frame,FALSE,FALSE,2);

    frame = create_button_box(FALSE,GTK_BUTTONBOX_END,"尾对齐");
    gtk_box_pack_start(GTK_BOX(hbox),frame,FALSE,FALSE,2);
    frame = create_button_box(FALSE,GTK_BUTTONBOX_START,"首对齐");
    gtk_box_pack_start(GTK_BOX(hbox),frame,FALSE,FALSE,2);

    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o box buttonbox.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./box 即可运行此程序，运行结果如图 3.1 所示。

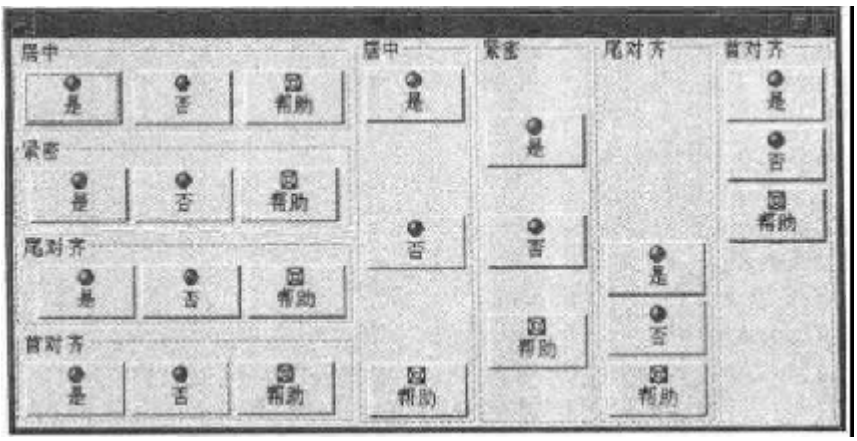

图 3.1 按钮盒

实例分析

(1) 按钮盒的属性

按钮盒的属性有以下几个值：GTK_BUTTONBOX_DEFAULT_STYLE，默认排列；GTK_BUTTONBOX_SPREAD，紧密排列；GTK_BUTTONBOX_EDGE，居中排列；GTK_BUTTONBOX_START，首对齐排列；GTK_BUTTONBOX_END，尾对齐排列。

(2) 控件的间隔

由于按钮盒是继承自盒状容器的，所以可以直接使用与盒状容器的相关函数来操作它，如用函数 gtk_box_set_spacing 来设定按钮盒中按钮之间的间隔空隙。

本例中自定义了两个函数来创建按钮和按钮盒，这样的函数在编程中很好用，读者应仔细看一下，以便在自己的应用程序中直接使用，就不用再重写了。

3.2 规范的框架

本节示例介绍如何使用规范框架控件，设定框架控件的外观、标题等属性。

实例说明

上节示例中大量使用了框架控件(GtkFrame)，但只有一种形式，本节将设定多种框架控件的形式，并使用纵横比框架控件(GtkAspectFrame)，让读者仔细体会一下框架的使用方法。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/3
mkdir asframe
cd asframe
```

创建工作目录，并进入此目录开始编程。绘制一幅规格为 8×8 像素的图像，以文件名

title.bmp 保存到此目录下。

(2) 打开编辑器，输入以下代码，以 asframe.c 为文件名保存到当前目录下。

```
/* 框架  asframe.c */
#include <gtk/gtk.h>

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

    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"不同样式的框架");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),20);

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

    frame = gtk_frame_new("框架一");
    gtk_box_pack_start(GTK_BOX(vbox),frame,FALSE,FALSE,5);
    gtk_frame_set_shadow_type(GTK_FRAME(frame),GTK_SHADOW_OUT);
    label = gtk_label_new("阴影类型：GTK_SHADOW_OUT  标题位置：居左");
    gtk_container_add(GTK_CONTAINER(frame),label);

    frame = gtk_frame_new("框架二");
    gtk_box_pack_start(GTK_BOX(vbox),frame,FALSE,FALSE,5);
    gtk_frame_set_shadow_type(GTK_FRAME(frame),GTK_SHADOW_ETCHED_OUT
);
    gtk_frame_set_label_align(GTK_FRAME(frame),1.0,0);
    label = gtk_label_new("阴影类型：GTK_SHADOW_ETCHED_OUT  标题位置：居
右");
    gtk_container_add(GTK_CONTAINER(frame),label);

    frame = gtk_frame_new(NULL);
    gtk_box_pack_start(GTK_BOX(vbox),frame,FALSE,FALSE,5);
    label = gtk_label_new("阴影类型：默认  标题位置：自定义的图像");
    gtk_container_add(GTK_CONTAINER(frame),label);
    image = gtk_image_new_from_file("title.bmp");
    gtk_frame_set_label_widget(GTK_FRAME(frame),image);

    asframe = gtk_aspect_frame_new("框架四",0,0.8,0.8,TRUE);
    gtk_box_pack_start(GTK_BOX(vbox),asframe,FALSE,FALSE,5);
    label=gtk_label_new("阴影类型：默认  标题位置：居左\n使用了aspectframe
```

```
控件，不能改变大小");
    gtk_container_add(GTK_CONTAINER(asframe),label);

    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o asframe asframe.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./asframe 即可运行此程序，运行结果如图 3.2 所示。

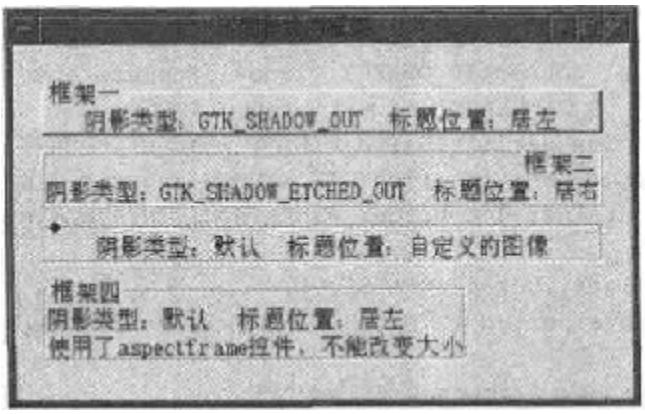

图 3.2 不同样式的框架

实例分析

(1) 框架的外观

可以用函数 gtk_frame_set_shadow_type 来设定框架控件的外观轮廓，其第 2 个参数为 GtkShadowType 类型，可以取以下几个值：GTK_SHADOW_NONE，无外框线；GTK_SHADOW_IN，内向型的外框线；GTK_SHADOW_OUT，外向型的外框线(像个按钮)；GTK_SHADOW_ETCHED_IN 和GTK_SHADOW_ETCHED_OUT一个向内凹一个向外凸。

(2) 纵横比框架控件

纵横比框架控件性能和框架控件在总体上是一样的，只不过它在长宽比例上是固定的，所以此程序在运行时即使拖长了窗口，它也不会占满整个容器的空间。可以使用下面函数来设定规范框架的长宽比：

```
gtk_aspect_frame_set(GTK_ASPECT_FRAME(frame),0,0.8,0.8,TRUE);
```

其第 2 和第 3 个参数分别表示横向和纵向的位置，取值 0 到 1 之间的浮点数，其中 0 表示左边、1 表示右边；第 4 个参数表示横纵比的系数；最后一个参数表示其子控件是否受其约束。

框架控件在 GTK+2.0 编程中经常用到，主要起到美化外观和规范其他控件的作用，读者不妨试一试其他参数的运行效果。

3.3　URL 链接

本节将介绍如何使用事件盒控件、处理事件盒控件的信号和在程序中启动浏览器。

实例说明

我们在编程时为了便于和使用者联系或让使用者查到某些方面的资料，一般在界面中加入一个按钮、图像或标签控件，使用者用鼠标单击此控件后会启动浏览器，自动打开 URL 链接。用 GTK+2.0 实现此功能要用到事件盒(GtkEventBox)控件。因为标签和图像一般不能直接接收外部的鼠标事件，将它们放入到事件盒中后，就可以利用事件盒的可接收外部鼠标事件的特性来处理鼠标事件了。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/3
mkdir url
cd url
```

创建工作目录，并进入此目录开始编程。找一幅 GIF 格式的图像以 linkicon.gif 为文件名保存到此目录下，再编辑一个简单的 HTML 文件，以 our.html 为文件名保存。

(2) 打开编辑器，输入以下代码，以 url.c 为文件名保存到当前目录下。

```
/* URL链接 url.c */
#include <gtk/gtk.h>
static gboolean
button_press_callback (GtkWidget      *event_box,
                       GdkEventButton *event,
                       gpointer        data)
{
    if(event->button == 1)//判断是否为鼠标左键
     system("galeon our.html");//打开our.html文件
    return TRUE;
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *hbox;
    GtkWidget *label;
    GtkWidget *image;
    GtkWidget *eventbox;

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

```
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"URL链接");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    label = gtk_label_new("以下为两种形式的超链接: ");
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,5);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);

    eventbox = gtk_event_box_new();
    g_signal_connect (G_OBJECT (eventbox),"button_press_event",
                  G_CALLBACK (button_press_callback),image);
    gtk_box_pack_start(GTK_BOX(hbox),eventbox,FALSE,FALSE,15);
    label = gtk_label_new("欢迎参观我的网页...");//这是其中的一个链接
    gtk_container_add(GTK_CONTAINER(eventbox),label);

    eventbox = gtk_event_box_new();
    g_signal_connect (G_OBJECT (eventbox),"button_press_event",
                   G_CALLBACK (button_press_callback),image);
    gtk_box_pack_start(GTK_BOX(hbox),eventbox,FALSE,FALSE,15);
    image = gtk_image_new_from_file("linkicon.gif");//图像链接
    gtk_container_add(GTK_CONTAINER(eventbox),image);

    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码:

```
CC = gcc
all:
    $(CC) -o url url.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译;

(5) 编译结束后，执行命令./url 即可运行此程序，运行结果如图 3.3 所示。

图 3.3 URL 链接

实例分析

(1) 事件盒的信号

事件盒的主要用途是为不具有自己窗口的控件来捕获事件的，它具有 GtkWidget 的所有信号类型。这里连接的是“key_press_event”，即单击鼠标信号，此信号的回调函数的第 2 个参数为 GdkEventButton 结构类型，用于存贮鼠标按下时的事件信息，其中的成员 button 为整型，表示鼠标哪一个键按下了，其中 1 为左键、2 为中键、3 为右键。

另外，此回调函数是有返回类型的，返回 TRUE 表示已经接收到事件，所以信号就不会发射了；返回 FALSE 则继续调用回调函数。

此示例中用的是 GNOME 桌面上常用的 GALEON 浏览器，读者还可以试试 NETSCAPE、MOZILA 等其他浏览器。

3.4 列 表 框

本节将介绍如何创建列表控件和使用列表控件的技巧。

实例说明

列表控件(GtkList)在编程中经常用到，一般表示在多个选项中选择一项或多项。本示例的功能是依次向列表框中加列表项，还可以清除这些列表项。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/3
mkdir list
cd list
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 list.c 为文件名保存到当前目录下。

```
/* 列表框 list.c */
#include <gtk/gtk.h>
static int i = 1 ;
GtkWidget *list;
void    list_add(GtkWidget *widget, gpointer data)
{
    gchar buf[100];
    GtkContainer *container;
    GtkWidget *item;

    sprintf(buf,"列表项 %d",i);
    container = GTK_CONTAINER(list);
    item = gtk_list_item_new_with_label(buf);
    gtk_container_add(container,item);
```

```
    gtk_widget_show(item);
    i++;
}
void    list_clear  (GtkButton *button,gpointer data)
{
    gtk_list_clear_items(GTK_LIST(list),0,-1);
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *hbox;
    GtkWidget *button;
    GtkWidget *view;

    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"列表控件");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    view = gtk_viewport_new(NULL,NULL);
    gtk_box_pack_start(GTK_BOX(vbox),view,TRUE,TRUE,5);
    list = gtk_list_new();
    gtk_container_add(GTK_CONTAINER(view),list);
    gtk_list_set_selection_mode(GTK_LIST(list),GTK_SELECTION_SINGLE)

    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    button = gtk_button_new_with_label("插入");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(list_add),NULL);
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);
    button = gtk_button_new_with_label("清除");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(list_clear),NULL);
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);

    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
```

```
$(CC) -o list list.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./list 即可运行此程序，运行结果如图 3.4 所示。

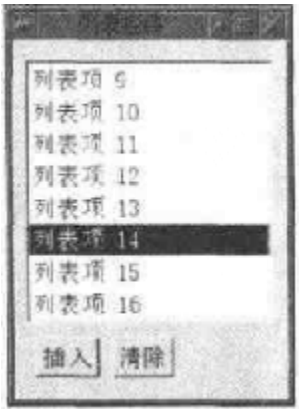

图 3.4　列表框

实例分析

(1) 列表框与列表项控件

列表框一般与列表项控件(GtkListItem)一起使用，本例中用自定义函数来创建一个显示标签文字的列表项，然后插入到列表框中去。

(2) 数据的存贮

列表框用双向链表(GList)数据结构来存贮数据，这使我们必须对它有一定了解，否则的话很难运用此控件。所有的有关双向链表的函数在 GLIB 的 API 参考手册中都能找到。

(3) 视区控件

此例中首次用到了视区控件(GtkViewPort)，它也是一个容器(只能容纳一个控件)，在功能上和框架控件大致相同，主要是美化控件外观的作用，这就是为什么此程序运行时列表框更具有立体感的原因。创建观察框用函数 gtk_viewport_new，这个函数有 2 个参数，都是 GtkAdjustment 类型的，它用来描述可调整的边界的值，默认情况下可用 NULL 代替。

由于某些原因列表框控件在 GTK+2.0 中已经不再被继续支持了，但它保存了原 GTK+1.2 中的所有功能，对初学者来说了解和掌握这一控件的使用在编程中仍然很有益处。

3.5　下拉列表框

本节将介绍如何创建和使用下拉列表框控件以及创建下拉列表框控件的技巧。

实例说明

与列表框不同的是下拉列表框控件(GtkCombo)占用容器的屏幕空间比较小，它多数用在多选一的情况下，这一点与条件菜单相同。本示例创建了两个下拉列表框，向下拉列表框中加入了标签、图像等控件供用户选择。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/3
mkdir combo
cd combo
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 combo.c 为文件名保存到当前目录下。

```
/* 下拉列表框  combo.c */
#include <gtk/gtk.h>
GtkWidget   *create_item    (gint i)
{
    GtkWidget* item;
    GtkWidget* hbox;
    GtkWidget* image;
    GtkWidget* label;

    hbox = gtk_hbox_new(FALSE,0);
    switch(i)
    {
    case 1 :
        image = gtk_image_new_from_stock
(GTK_STOCK_YES,GTK_ICON_SIZE_MENU);
        label = gtk_label_new("列表项一");
        break;
    case 2 :
        image =
gtk_image_new_from_stock(GTK_STOCK_NO,GTK_ICON_SIZE_MENU);
        label = gtk_label_new("列表项二");
        break;
    case 3 :
        image = gtk_image_new_from_stock
(GTK_STOCK_HELP,GTK_ICON_SIZE_MENU);
        label = gtk_label_new("列表项三");
        break;
    case 4 :
        image =
gtk_image_new_from_stock(GTK_STOCK_OK,GTK_ICON_SIZE_MENU);
        label = gtk_label_new("列表项四");
        break;
    case 5 :
        image = gtk_image_new_from_stock
(GTK_STOCK_CANCEL,GTK_ICON_SIZE_MENU);
        label = gtk_label_new("列表项五");
        break;
    }
    gtk_box_pack_start(GTK_BOX(hbox),image,FALSE,FALSE,2);
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,2);
```

```
    item = gtk_list_item_new();
    gtk_container_add(GTK_CONTAINER(item),hbox);
    gtk_widget_show_all(item);
    return item;
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *combo;
    GtkWidget *vbox;
    GtkWidget *label;
    GList* items = NULL;
    GtkWidget *item;

    items = g_list_append(items,"列表项 A");
    items = g_list_append(items,"列表项 B");
    items = g_list_append(items,"列表项 C");
    items = g_list_append(items,"列表项 D");
    items = g_list_append(items,"列表项 E");
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"下拉列表框");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),20);

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    label = gtk_label_new("请选择下列中的一项:");
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,5);
    combo = gtk_combo_new();
    gtk_box_pack_start(GTK_BOX(vbox),combo,FALSE,FALSE,5);
    gtk_combo_set_popdown_strings(GTK_COMBO(combo),items);

    label = gtk_label_new("另一种选择方式: ");
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,5);
    combo = gtk_combo_new();
    gtk_box_pack_start(GTK_BOX(vbox),combo,FALSE,FALSE,5);
    item = create_item(1);
    gtk_combo_set_item_string(GTK_COMBO(combo),GTK_ITEM(item),"项目一
");
    gtk_container_add(GTK_CONTAINER(GTK_COMBO(combo)->list),item);
    item = create_item(2);
    gtk_combo_set_item_string(GTK_COMBO(combo),GTK_ITEM(item),"项目二
");
    gtk_container_add(GTK_CONTAINER(GTK_COMBO(combo)->list),item);
    item = create_item(3);
    gtk_combo_set_item_string(GTK_COMBO(combo),GTK_ITEM(item),"项目三
");
```

```
    gtk_container_add(GTK_CONTAINER(GTK_COMBO(combo)->list),item);
    item = create_item(4);
    gtk_combo_set_item_string(GTK_COMBO(combo),GTK_ITEM(item),"项目四
");
    gtk_container_add(GTK_CONTAINER(GTK_COMBO(combo)->list),item);
    item = create_item(5);
    gtk_combo_set_item_string(GTK_COMBO(combo),GTK_ITEM(item),"项目五
");
    gtk_container_add(GTK_CONTAINER(GTK_COMBO(combo)->list),item);

    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入如下代码：

```
CC = gcc
all:
    $(CC) -o combo combo.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./combo 即可运行此程序，运行结果如图 3.5 所示。

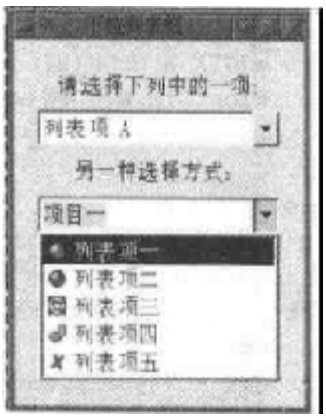

图 3.5 下拉列表框

实例分析

(1) 创建下拉列表框

可以用函数 gtk_combo_new 来直接创建下拉列表框控件，事实上下拉列表框是由一个文字录入控件、一个箭头控件和一个列表框组合而成的，这几个控件之间的关系处理得非常协调。正常情况下箭头是可以激活的，即按下后弹出列表框，用户可用函数来设定这一属性，理解这一特点对使用下拉列表框很有好处。

(2) 向下拉列表框中加数据项

可以用函数 gtk_combo_set_popdown_string 来向下拉列框中添加数据项，要注意的是它的第 2 个参数是 GList 类型的；还可以用容器的方式来向下拉列框中添加复杂的控件作为数据项，如本例中的第 2 个下拉列表框控件，就是先自定义函数来创建一个复合列表项控

件(一个图像和一个标签)，然后用函数 gtk_container_add 向下拉列表框的列表框中添加子控件，型如：gtk_container_add(GTK_CONTAINER(GTK_COMBO(combo)->list),item);，再用函数 gtk_combo_set_item_string 来设定此项在文字录入控件中要显示的字符串。

同样用 gtk_entry_get_text(GTK_ENTRY(GTK_COMBO(combo)->entry))来取得当前选择的项目数字。

下拉列表框控件的使用使我们进一步了解了容器和控件间的包容关系、控件间的继承关系，也让我们看到使用控件转换宏的常用方法，理解这些后您使用 GTK+2.0 控件的水平会大大提高的。

3.6　自 由 布 局

本节将介绍如何创建使用自由布局控件，如何接收和控制键盘的输入，如何动态更改图像控件的显示内容。

实例说明

GTK+2.0 提供了一种固定布局控件(GtkFixed)，它允许使用者按几何坐标方式向其中添加排列控件，并可以移动控件的位置。本例就以此控件为核心，以键盘控制移动坦克图像来完成一个简单的小游戏。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/3
mkdir fixed
cd fixed
```

创建工作目录，并进入此目录开始编程。收集或绘制 4 个坦克图像。4 个图像分别让坦克朝 4 个不同的方向，分别命名为 0.png，2.png，4.png，6.png。

(2) 打开编辑器，输入以下代码，以 tank.c 为文件名保存到当前目录下。

```
/* 坦克游戏 tank.c */
#include <gtk/gtk.h>
#include <gdk/gdkkeysyms.h>
enum _FORWARD { LEFT, UP, RIGHT, DOWN } ;
typedef enum _FORWARD Forward; //定义方向类型
void    move        (Forward fw);
void    key_press   (GtkWidget* widget,GdkEventKey *event,gpointer
data);
static gchar* tank_file[4] = {"0.png","2.png","4.png","6.png"};
static GtkWidget *fixed;
static GtkWidget *tank_image;
gint i = 75 ;
gint j = 75 ;
Forward forward = UP ;//定义方向
```

```
void    move    (Forward fw)
{
    switch(fw)
    {
    case UP :
        j = j - 5 ;
        if ( j < 0 ) j = 175 ;
        gtk_image_set_from_file(GTK_IMAGE(tank_image),tank_file[0]);
        gtk_fixed_move(GTK_FIXED(fixed),tank_image,i,j);
        break;
    case DOWN :
        j = j + 5;
        if ( j > 200 ) j = 0 ;
        gtk_image_set_from_file(GTK_IMAGE(tank_image),tank_file[2]);
        gtk_fixed_move(GTK_FIXED(fixed),tank_image,i,j);
        break;
    case LEFT :
        i = i - 5;
        if ( i < 0 ) i = 175 ;
        gtk_image_set_from_file(GTK_IMAGE(tank_image),tank_file[3]);
        gtk_fixed_move(GTK_FIXED(fixed),tank_image,i,j);
        break;
    case RIGHT :
        i = i + 5;
        if ( i > 200 ) i = 0 ;
        gtk_image_set_from_file(GTK_IMAGE(tank_image),tank_file[1]);
        gtk_fixed_move(GTK_FIXED(fixed),tank_image,i,j);
        break;
    }
}
void    key_press   (GtkWidget* widget,GdkEventKey *event,gpointer
data)
{
    switch(event->keyval)
    {
    case GDK_Up :
        forward = UP;
        move(forward);
        break;
    case GDK_Down :
        forward = DOWN;
        move(forward);
        break;
    case GDK_Left :
        forward = LEFT;
        move(forward);
        break;
    case GDK_Right :
        forward = RIGHT;
        move(forward);
        break;
```

```
    }
}
//主函数
int main    (int argc, char *argv[])
{
    GtkWidget* window;
    GtkWidget* vbox;
    GtkWidget* bbox;
    GtkWidget* sep;
    GtkWidget* frame;
    GtkWidget* button;

    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"坦克游戏");
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
    g_signal_connect(G_OBJECT(window),"destroy",
            G_CALLBACK(gtk_main_quit),NULL);

    g_signal_connect(G_OBJECT(window),"key_press_event",
                G_CALLBACK(key_press),NULL);

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    frame = gtk_frame_new(NULL);
    gtk_frame_set_shadow_type(GTK_FRAME(frame),GTK_SHADOW_ETCHED_OUT
);
    fixed = gtk_fixed_new();
    gtk_fixed_set_has_window(GTK_FIXED(fixed),TRUE);
    gtk_widget_set_size_request(fixed,200,200);
    gtk_container_add(GTK_CONTAINER(frame),fixed);
    gtk_box_pack_start(GTK_BOX(vbox),frame,TRUE,TRUE,5);
    sep = gtk_hseparator_new();
    gtk_box_pack_start(GTK_BOX(vbox),sep,FALSE,FALSE,5);
    bbox = gtk_hbutton_box_new();
    gtk_button_box_set_layout(GTK_BUTTON_BOX(bbox),GTK_BUTTONBOX_END
);
    gtk_box_pack_start(GTK_BOX(vbox),bbox,FALSE,FALSE,5);
    button = gtk_button_new_from_stock(GTK_STOCK_QUIT);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_box_pack_end(GTK_BOX(bbox),button,FALSE,FALSE,5);
    tank_image = gtk_image_new_from_file(tank_file[0]);
    gtk_fixed_put(GTK_FIXED(fixed),tank_image,i,j);

gtk_widget_show_all(window);
    gtk_main();
    return FALSE ;
}
```

(3) 编辑 Makefile 输入如下代码：

```
CC = gcc
all:
    $(CC) -o tank tank.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./tank 即可运行此程序，运行结果如图 3.6 所示。

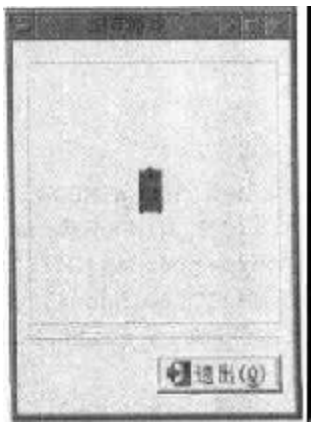

图 3.6 坦克游戏

实例分析

(1) 自由布局控件

自由布局控件很像 Windows 下 delphi 编程中的 Form，您可以用函数 gtk_fixed_put 来向自由布局控件的固定位置摆放任何控件，同时也可以给控件设定固定的尺寸；移动布局控件中的子控件可用函数 gtk_fixed_move。

(2) 接收键盘输入

在 GTK+2.0 编程中接收键盘输入需要向窗口的 key_press_event 信号加回调函数，此回调函数的第 2 个参数是 GdkEventKey 结构类型，它有一个成员 keyval 来表示键值，GTK+2.0 用一系列宏来表示键值，这些键值在头文件/usr/include/gtk-2.0/gdk/gdkkeysyms.h 中定义，所以要把此文件包含进来(直接写 gdk/gdkkeysyms.h)。键值以 GDK_开始的一系列宏来表示(详情见 GDK 的 API 参考手册)，这样可以通过判断这些键值来做出相应的处理。

自由布局控件不同于其他容器控件的使用方法，在 GTK+2.0 中是非常特殊的，还有一个和它相类似的布局控件(GtkLayout)，它们为使用者提供了一种另类的应用控件方式，读者要认真加以体会并掌握。

3.7 图像控件的直接引用

本节示例将介绍如何创建使用 GdkPixbuf 资源，将图像文件转换成数据保存到程序代码中使用和使用 GdkPixbuf 资源的技巧。

实例说明

图像资源在编程中经常用到，本例中用到的是GdkPixbuf资源，这是GTK+2.0特有的支持多种图像格式的资源，在GTK+2.0中经常用到。它可以通过调用外部图像直接创建，也可以将图像转换成数据保存到程序代码中来创建，其中后一种方式可以免除程序运行时找不到图像的烦恼。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/3
mkdir pixmap
cd pixmap
```

创建工作目录，并进入此目录开始编程。找两幅PNG格式的图像，一幅名为pieces.png，一幅名为simple-40.png，复制到此目录下。执行如下命令：

```
gdk-pixbuf-csource --raw --name pieces_inline pieces.png > pieces.h
```

会在当前目录下生成一个名为pieces.h的头文件。

(2) 打开编辑器，输入以下代码，以pixmap.c为文件名保存到当前目录下。

```
/* 图像的直接引用 pixmap.c */
#include <gtk/gtk.h>
#include "pieces.h"

int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *frame;
    GtkWidget *image;
    GdkPixbuf *pixbuf;
    GdkPixbuf *pix1,*pix2,*pix3;

    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"图像的直接引用");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    pixbuf = gdk_pixbuf_new_from_file("simple-40.png",NULL);
    frame = gtk_frame_new("这是一幅完整的图像");
    image = gtk_image_new_from_pixbuf(pixbuf);
```

```
    gtk_box_pack_start(GTK_BOX(vbox),frame,FALSE,FALSE,5);
    gtk_container_add(GTK_CONTAINER(frame),image);

    pix1 = gdk_pixbuf_new_subpixbuf(pixbuf,280,40,40,40);
    frame = gtk_frame_new("从上图中分割出来的一幅图像");
    image = gtk_image_new_from_pixbuf(pix1);
    gtk_container_add(GTK_CONTAINER(frame),image);
    gtk_box_pack_start(GTK_BOX(vbox),frame,FALSE,FALSE,5);

    pix2 =
gdk_pixbuf_new_from_inline(22400+24,pieces_inline,TRUE,NULL);
    frame = gtk_frame_new("内建的一幅图像");
    image = gtk_image_new_from_pixbuf(pix2);
    gtk_container_add(GTK_CONTAINER(frame),image);
    gtk_box_pack_start(GTK_BOX(vbox),frame,FALSE,FALSE,5);

    pix3 = gdk_pixbuf_new_subpixbuf(pix2,120,20,20,20);
    frame = gtk_frame_new("从内建图像中分割出来的一幅图像");
    image = gtk_image_new_from_pixbuf(pix3);
    gtk_container_add(GTK_CONTAINER(frame),image);
    gtk_box_pack_start(GTK_BOX(vbox),frame,FALSE,FALSE,5);

    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o pixmap pixmap.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./pixmap 即可运行此程序，运行结果如图 3.7 所示。

图 3.7 图像的直接引用

实例分析

(1) 将图像转换成 C 语言源代码格式

本例中用 GTK+2.0 自带的一个命令行工具 gdk-pixbuf-csource 来完成图像转换成 C 语言源代码格式的数据，并保存到 C 语言的头文件中，关键是要理解下面的命令行：

```
gdk-pixbuf-csource --raw --name pieces_inline pieces.png > pieces.h
```

此命令将 pieces.png 图像转换成 C 语言源代码格式并将数据结构命名为 pieces_inline，结果输出到 pieces.h 文件中去。打开 pieces.h 文件就可以看到这一数据结构了。还可以执行此命令带--help 选项来查看此命令的详细帮助。

(2) gdk-pixbuf 的作用

GTK+2.0 将 gdk-pixbuf 程序库完整地包含进来，它的主要功能是读取、处理、保存各种格式的图像资源，也包括动画。这里要区分的是 GdkPixbuf 并不是 GtkImage，后者是图像控件，前者是一种资源对象。

本例中用到了 GdkPixbuf 的分割功能，从一幅大的图像中分割出一小部分，然后用这一小部分再创建一幅图像显示出来，事实上完全可以用函数 gdk_pixbuf_save 来将这部分资源保存为图像，读者不妨试一试，还有其他一些功能可参考 gdk-pixbuf 的 API 参考手册。图像在美化用户界面时起到至关重要的作用，而一个整洁美观的用户界面是软件给用户的第一印象，要想创造出更漂亮的界面，一定要在 GdkPixbuf 上下功夫。

3.8　控件属性的综合设置

本节将介绍如何利用已存在的源程序中的功能，动态设定控件的属性，理解控件属性的意义。

实例说明

GTK+2.0 的控件有很多属性，如控件的名称、父控件的名称、是否有子控件、控件的尺寸、是否可以接收外部输入等等，要设定这些属性多数情况下用到 gtk_widget_*系列函数，可以在 GTK+2.0 的 API 参考手册的 gtkwidget.html 中找到。本节示例另辟蹊径，利用 GTK+2.0 的源代码包中的 tests 目录下的 prop_editor.c 和 prop_editor.h 两个现有的文件，在程序运行中动态设置控件的属性。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/3
mkdir prop
cd prop
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 prop.c 为文件名保存到当前目录下。

```
/* 控件的属性 prop.c*/
#include <gtk/gtk.h>
#include "prop-editor.h"
void    on_button_clicked   (GtkButton* button,GtkWidget* data)
{
    GtkWidget* dialog;
    dialog = create_prop_editor(data,NULL);
    gtk_widget_show(dialog);
}
int main    (int argc, char* argv[])
{
    GtkWidget* window, *vbox;
    GtkWidget* label, *button;
        gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"设置控件的属性");
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    label = gtk_label_new("此功能用来设置控件的属性，\n这用到了GTK+2.0源代
码中tests目录下的\nprop-editor.h和prop-editor.c两个文件，\n它们提供了
create_prop_editor函数，\n这使我们设定控件的属性变得非常容易。");
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,5);
    button = gtk_button_new_with_label("点击此按钮来设置此按钮控件的属性
");
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_button_clicked),button);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:    prop.o prop-editor.o
    $(CC) -o prop prop.o prop-editor.o 'pkg-config --libs gtk+-2.0'
prop.o: prop.c  prop-editor.h
    $(CC) -c prop.c 'pkg-config --cflags gtk+-2.0'
prop-editor.o:  prop-editor.c prop-editor.h
    $(CC) -c prop-editor.c 'pkg-config --cflags gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./prop 即可运行此程序，运行结果如图 3.8 和 3.9 所示。

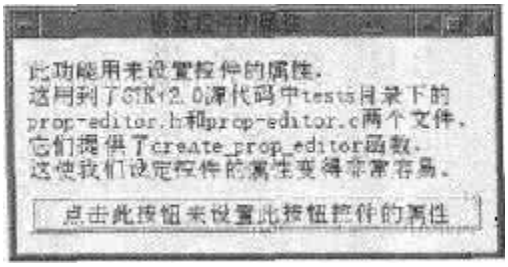

图 3.8　主窗口

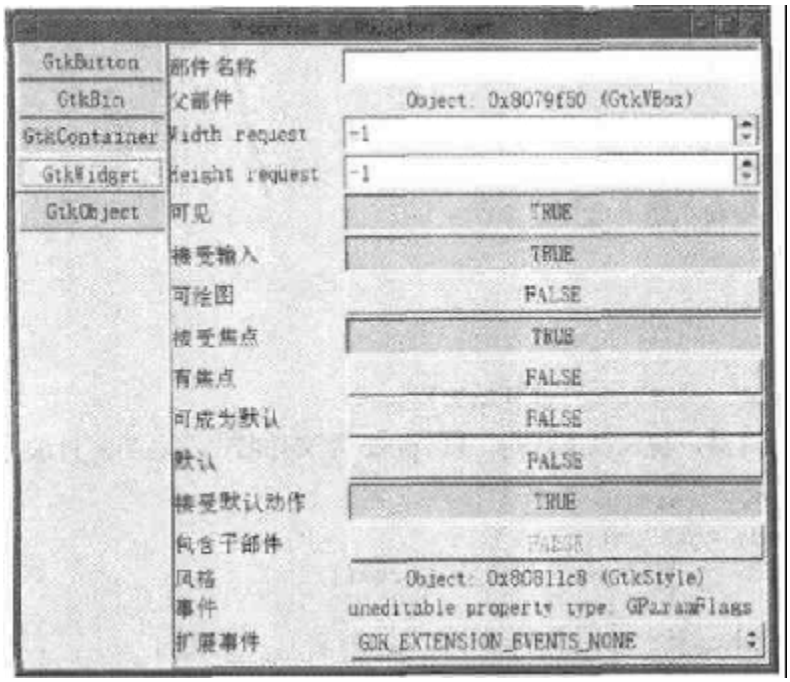

图 3.9　控件属性窗口

实例分析

(1)　两个外部文件

通过观察 prop_editor.c 文件的源程序我们会发现这个程序非常复杂，复杂得超出了我们的想象，但 prop_editor.h 文件中只声明了一个函数：create_prop_editor，它有两个参数，第一个是要显示属性的控件，要转换成 GObject；第二个是 GType 类型的，可以设为 NULL。此函数的返回值就是属性窗口的指针，可以用 gtk_widget_show 函数来将它显示出来。

(2)　运行

运行此程序会发现属性窗口分成多个页面，由下向上表示了控件的继承关系，每个父控件的属性都显示出来，有的可以设置，有的则不可以，如将 GtkWidget 的“可见”属性设为 FALSE，则按钮将会在窗口中消失。

此例主要是通过使用已有的外部的源程序中提供的函数，来查看和设置控件的常见属性，要想了解这些功能是如何实现的，还要认真阅读分析一下源代码才行。另一个关键是更改 Makefile 文件，这是分析源代码的基本功。

3.9 数 字 选 择

本节将介绍如何创建和使用滚动按钮控件，设定滚动按钮控件的属性来进行数字选择。

实例说明

数值调节按钮控件(GtkSpinButton)是最常用的数字选择控件，它可以根据设置来显示并供用户选择整数或浮点数，还可以组合成其他控件。本示例向读者展示了数值调节按钮控件的一般使用方法。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/3
mkdir spin
cd spin
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 spin.c 为文件名保存到当前目录下。

```
/* 滚动按钮 spin.c */
#include <gtk/gtk.h>
int main    (int argc, char* argv[])
{
    GtkWidget* window;
    GtkWidget* vbox;
    GtkWidget* frame;
    GtkWidget* spin;
    GtkWidget* label;
    GtkWidget* vvbox;
    GtkWidget *hbox,*vbox1,*vbox2,*vbox3;
    GtkAdjustment* adj1;

    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"滚动按钮");
    g_signal_connect(G_OBJECT(window),"destroy",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    frame = gtk_frame_new("类似一个日历");
    gtk_box_pack_start(GTK_BOX(vbox),frame,FALSE,FALSE,5);

    hbox = gtk_hbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(frame),hbox);
    gtk_container_set_border_width(GTK_CONTAINER(hbox),10);
```

```
    vbox1 = gtk_vbox_new(TRUE,0);
    gtk_box_pack_start(GTK_BOX(hbox),vbox1,TRUE,TRUE,5);
    label = gtk_label_new("年: ");
    gtk_box_pack_start(GTK_BOX(vbox1),label,FALSE,FALSE,2);
    spin = gtk_spin_button_new_with_range(1900,2100,1);
    gtk_box_pack_start(GTK_BOX(vbox1),spin,FALSE,FALSE,2);
    vbox2 = gtk_vbox_new(TRUE,0);
    gtk_box_pack_start(GTK_BOX(hbox),vbox2,TRUE,TRUE,5);
    label = gtk_label_new("月: ");
    gtk_box_pack_start(GTK_BOX(vbox2),label,FALSE,FALSE,2);
    spin = gtk_spin_button_new_with_range(1,12,1);
    gtk_box_pack_start(GTK_BOX(vbox2),spin,FALSE,FALSE,2);
    vbox3 = gtk_vbox_new(TRUE,0);
    gtk_box_pack_start(GTK_BOX(hbox),vbox3,TRUE,TRUE,5);
    label = gtk_label_new("日: ");
    gtk_box_pack_start(GTK_BOX(vbox3),label,FALSE,FALSE,2);
    spin = gtk_spin_button_new_with_range(1,31,1);
    gtk_box_pack_start(GTK_BOX(vbox3),spin,FALSE,FALSE,2);

    frame = gtk_frame_new(NULL);
    gtk_box_pack_start(GTK_BOX(vbox),frame,FALSE,FALSE,5);
    vvbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(frame),vvbox);
    gtk_container_set_border_width(GTK_CONTAINER(vvbox),10);
    label = gtk_label_new("第一个滚动按钮,\n显示整数,范围: 0 - 100");
    adj1 = (GtkAdjustment *) gtk_adjustment_new (50.0, 0.0, 100.0, 1.0,
5.0, 5.0);
    spin = gtk_spin_button_new(adj1,5,1);
    gtk_box_pack_start(GTK_BOX(vvbox),label,FALSE,FALSE,3);
    gtk_box_pack_start(GTK_BOX(vvbox),spin,FALSE,FALSE,3);

    label = gtk_label_new("第二个滚动按钮,\n显示浮点数,范围: 0.1 - 1.50");
    spin = gtk_spin_button_new_with_range(0,9.9,0.1);
    gtk_box_pack_start(GTK_BOX(vvbox),label,FALSE,FALSE,3);
    gtk_box_pack_start(GTK_BOX(vvbox),spin,FALSE,FALSE,3);

    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码:

```
CC = gcc
all:
    $(CC) spin.c -o spin 'pkg-config gtk+-2.0 --cflags --libs'
```

(4) 在终端中执行 make 命令开始编译;

(5) 编译结束后，执行命令./spin 即可运行此程序，运行结果如图 3.10 所示。

图 3.10 数字选择

实例分析

(1) 滚动按钮

有两个函数可以创建滚动按钮，一个是 gtk_spin_button_new，它有 3 个参数，第 1 个参数是 GtkAdjustment 类型，第 2 个参数是滚动按钮每次按下后数字变化的幅度，第 3 个参数是按钮中文本显示区域中显示的数字；另一个创建滚动按钮的函数是 gtk_spin_button_new_with_range，它也有 3 个参数，第 1 个参数是最小值，第 2 个参数是最大值，第 3 个参数是每次增长的值。

(2) 取得滚动按钮的当前值

本例中未取得滚动按钮的当前值，可以用函数 gtk_spin_button_get_value 来取得，这个值是 gdouble 型的，还可以用 gtk_spin_button_get_value_as_int 函数来取得滚动按钮的整型值。

(3) 滚动按钮的信号

滚动按钮有 4 个属于自己的信号，其中“change_value”和“value_changed”信号分别在变量改变前和改变后发出，对于用滚动按钮来控制其他控件很有用；另两个信号是“input”和“output”，分别在输入和输出时发生。

滚动按钮控件提供非常轻松的选择数字的方式，在用户要输入大量规范数字时经常用到，事实上它是一个单行输入控件和两个方向按钮控件用容器的包装组合而成，然后再用一系列函数将它们联系起来。

3.10 执行命令工具

本节将介绍如何灵活运用单行输入控件(GtkEntry)和按钮控件，为它们的信号添加回调函数实现一个执行命令的小工具。

实例说明

在多数的 Linux 桌面上都提供一个执行用户命令的小工具，它有一个命令输入栏、一

个执行按钮以及其他控件，如果要执行命令，只需要在输入栏中输入命令，单击执行按钮就可以了。本示例就是按此功能设计的。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/3
mkdir command
cd command
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 command.c 为文件名保存到当前目录下。

```
/* 命令工具  command.c */
#include <gtk/gtk.h>
static GtkWidget *entry;
void on_button_clicked (GtkButton* button,gpointer data)
{
    const gchar* command;
    command = gtk_entry_get_text(GTK_ENTRY(entry));
    system(command);
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *hbox;
    GtkWidget *label;
    GtkWidget *button;
        gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"命令窗口");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);

    hbox = gtk_hbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),hbox);
    label = gtk_label_new("输入命令: ");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);

    entry = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(hbox),entry,TRUE,TRUE,5);
        button = gtk_button_new_with_label("执行");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_button_clicked),NULL);
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);

    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
```

```
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o command command.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./command 即可运行此程序，运行结果如图 3.11 所示。

图 3.11 执行命令工具

实例分析

(1) 启动命令

和 URL 一样，此示例也用 Linux 系统提供的 system 函数调用来执行命令，这对初学者很有用。如果您对 Linux 的底层系统编程很有研究的话，还可以用 fork、exec 等系统调用来启动命令。

(2) 甩开终端控制

这个小工具有一个缺点，例如打开一个浏览器浏览网页，当关闭这个窗口时浏览器也会随之关闭，这主要是因为浏览器在打开时进程没有甩开终端控制。我们在编程中也可以让自己的程序进程甩开终端控制，在第 8 章编辑软件一节有此功能的使用，有兴趣的读者可以自行参考。

(3) 命令的输出

正常情况下启动运行命令的结果输出到启动此命令的终端中，我们还可以用编程的方式让输出显示到窗口中来，不过这需要用到 Linux 的一些系统调用和多行文本输出控件，有兴趣的读者可自行研究。

虽然是一个小工具却涉及到 Linux 的多个方面，看来要想学好编程，首先对操作系统一定要深入了解，否则的话难有起色。

3.11 分 隔 面 板

本节将介绍如何创建和使用 GTK+2.0 中提供的分隔面板控件，创建可以灵活改变控件区域大小的用户界面。

实例说明

在常用软件中可以看到一种分隔条控件，鼠标放上去后会改变形状，按下鼠标就可拖动此分隔条从而改变某些区域的大小。GTK+2.0 未提供此分隔条控件，但提供了一种比分隔条控件更好用的分隔面板控件(GtkPanel)，它是一个可以容纳两个控件的容器，容纳的两

个控件之间有一个分隔条，功能和前面所述一样。分隔面板也分横向的(GtkHPanel)和纵向的(GtkVPanel)两种，本示例就向读者演示了这两种控件的简单用法。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/3
mkdir panel
cd panel
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 panel.c 为文件名保存到当前目录下。

```
/* 分隔面板 panel.c */
#include <gtk/gtk.h>
int main    (int argc, char* argv[])
{
    GtkWidget* window;
    GtkWidget* paned;
    GtkWidget* panel1;
    GtkWidget* button1;
    GtkWidget* button2;
    GtkWidget* button3;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"分隔面板测试程序");
    g_signal_connect(G_OBJECT(window),"destroy",
            G_CALLBACK(gtk_main_quit),NULL);
    panel = gtk_hpanel_new();
    gtk_container_add(GTK_CONTAINER(window),panel);
    gtk_widget_show(panel);
    button1 = gtk_button_new_with_label("按钮 一");
    gtk_panel_add1(GTK_PANEL(panel),button1);
    gtk_widget_show(button1);

    panel1 = gtk_vpanel_new();
    gtk_panel_add2(GTK_PANEL(panel),panel1);
    gtk_widget_show(panel1);
    button2 = gtk_button_new_with_label("按钮 二");
    gtk_panel_add1(GTK_PANEL(panel1),button2);
    gtk_widget_show(button2);
    button3 = gtk_button_new_with_label("按钮三");
    gtk_panel_add2(GTK_PANEL(panel1),button3);
    gtk_widget_show(button3);
        gtk_widget_show(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) panel.c -o panel 'pkg-config gtk+-2.0 --cflags --libs'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./panel 即可运行此程序，运行结果如图 3.12 所示。

图 3.12 分隔面板

实例分析

(1) 向分隔面板中加控件

创建分隔面板控件用函数 gtk_vpanel_new(纵向)或 gtk_hpanel_new(横向)来直接创建，向分隔面板中添加控件也很简单，用函数 gtk_panel_add1 和 gtk_panel_add2 就可以实现，其中 add1 表示向分隔面板的第一个容器空间添加，即横向面板的左侧或纵向面板的上部，add2 则依次。

另外也可用 gtk_panel_pack1 和 gtk_panel_pack2 来向面板中添加控件，它的参数多一些，主要是表示子控件的属性，如是否可扩展等；还可以确定分隔条的位置，详细情况可见 GTK+2.0 的 API 参考手册。

更好地运用此控件的关键在于容器之间的嵌套，理解容器和容器间的嵌套关系对使用分隔面板和其他容器控件来说非常重要，灵活使用可以设计出更漂亮的界面来。

本章介绍了用 GTK+2.0 编程的一些技巧，其中多数都非常容易理解，相信读者在学习完后会很快掌握并运用自如。更希望读者能总结出更多的技巧，创建出更多的实用程序。

第4章 对 话 框

本章重点：

对话框在编程中起到了提示、与用户交流信息、选择、判断用户输入的作用，本章中的所有示例都是围绕这些功能来编写的。以使读者能充分领略到GTK+2.0中各种对话框的创建和使用技巧。

本章主要内容：

- 取得用户名和密码的登录对话框
- 多选项和单选项的对话框
- 选择字体、颜色、文件和目录的对话框
- 使用日历控件的日期选择对话框
- 常用的信息输出对话框
- 常用的确认/取消，是/否/取消等编程中需要用户选择的对话框
- 一个类似GNOME2界面的关于对话框

4.1 登 录 窗 口

本节将介绍如何灵活运用布局控件、单行文本输入控件(GtkEntry)、按钮控件等创建一个登录对话框。

实例说明

多数较系统的、有安全性要求的应用程序在开始运行时都显示一个登录对话框，要求输入用户名和密码，为保密起见，密码栏不能显示出字符，只能以*号代替。当用户输入正确后方可登录系统，使用此软件。本示例就初步完成了这一功能。

实现步骤

(1) 打开终端输入以下命令：

```
cd ~/ourgtk/
mkdir 4
cd 4
mkdir pass
cd pass
```

创建本章的总目录4，再在本章目录下创建本节目录pass，进入此目录，开始编程。

(2) 打开编辑器，输入以下代码，以pass.c为文件名保存到当前目录下。

```
/* 登录窗口  pass.c */
#include <gtk/gtk.h>
```

```
static GtkWidget* entry1;
static GtkWidget* entry2;
void
on_button_clicked   (GtkWidget* button,gpointer data)
{
    const gchar *username = gtk_entry_get_text(GTK_ENTRY(entry1));
    const gchar *password = gtk_entry_get_text(GTK_ENTRY(entry2));
    g_print("用户名是: %s  ",username);
    g_print("\n");
    g_print("密码是: %s  ",password);
    g_print("\n");
}
int main ( int argc , char* argv[])
{
    GtkWidget* window;
    GtkWidget* box;
    GtkWidget* box1;
    GtkWidget* box2;
    GtkWidget* label1;
    GtkWidget* label2;
    GtkWidget* button;
    GtkWidget* sep;

    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),"登录窗口");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),20);

    box = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),box);
    box1 = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(box),box1,FALSE,FALSE,5);
    box2 = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(box),box2,FALSE,FALSE,5);
    sep = gtk_hseparator_new();
    gtk_box_pack_start(GTK_BOX(box),sep,FALSE,FALSE,5);

    label1 = gtk_label_new("用户名: ");
    entry1 = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(box1),label1,FALSE,FALSE,5);
    gtk_box_pack_start(GTK_BOX(box1),entry1,FALSE,FALSE,5);

    label2 = gtk_label_new(" 密码: ");
    entry2 = gtk_entry_new();
    gtk_entry_set_visibility(GTK_ENTRY(entry2),FALSE);
    gtk_box_pack_start(GTK_BOX(box2),label2,FALSE,FALSE,5);
    gtk_box_pack_start(GTK_BOX(box2),entry2,FALSE,FALSE,5);
```

```
    button = gtk_button_new_with_label("确认");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_button_clicked),NULL);
    g_signal_connect_swapped(G_OBJECT(button),"clicked",
            G_CALLBACK(gtk_widget_destroy),window);
    gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,5);

    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o pass pass.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./pass 即可运行此程序，运行结果如图 4.1 所示。

图 4.1　登录窗口

实例分析

(1) 创建分隔线

分隔线控件(GtkSeparator)分为横向分隔线(GtkHSeparator)和纵向分隔线(GtkVSeparator)，是最简单的修饰性控件，可以直接创建并显示出来，没有其他属性和信号。

(2) 让密码栏显示*号

用函数 gtk_entry_set_visibility 来设定单行输入控件的显示属性，其第二个参数是 FALSE 则显示的内容全都是*号。

登录对话框是最简单的对话框，在熟悉了前几章的内容后，它的界面非常容易做出来，关键在于如何让密码栏显示*号，并且在用户确认后取得用户名和密码。读者可以把这一对话框加入到自己的程序中，加入处理用户名和密码的代码，使之更加完善。

4.2　创建有多个选项的窗口

本节将介绍如何使用多选按钮控件创建有多个选项的窗口。

实例说明

图形界面应用程序中经常用多选按钮(GtkCheckButton)来表示多种条件同时成立，GTK+2.0 中也有同样的控件并且功能完全相同。本示例就是用框架和纵向盒状容器来排列四个多选按钮，最后输出用户的选择。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/4
mkdir check
cd check
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 checkbutton.c 为文件名保存到当前目录下。

```
/* 多项选择窗口 checkbutton.c */
#include <gtk/gtk.h>
gboolean isbold = FALSE ;
gboolean isitli = FALSE ;
gboolean isuline = FALSE ;
gboolean iscolor = FALSE ;
void on_check_clicked  (GtkWidget* check,gpointer data)
{
    switch((int)data)
    {
    case 1:
        isbold = !isbold;
        break;
    case 2:
        isitli = !isitli;
        break;
    case 3:
        isuline = !isuline;
        break;
    case 4:
        iscolor = !iscolor;
        break;
    }
}
void on_button_clicked (GtkWidget *button,gpointer data)
{
    g_print("字体配置为: ");
    if(isbold)
        g_print("粗体 ");
    if(isitli)
        g_print("斜体 ");
    if(isuline)
        g_print("下划线 ");
```

```
    if(iscolor)
        g_print("彩色 ");
    if( !isbold && !iscolor && !isuline && !isitli )
        g_print("正常，无任何选项");
    g_print("\n");
}
//主函数
int main    (int argc, char *argv[])
{
    GtkWidget* window;
    GtkWidget* frame;
    GtkWidget* box;
    GtkWidget* button;
    GtkWidget* box1;
    GtkWidget* check1;
    GtkWidget* check2;
    GtkWidget* check3;
    GtkWidget* check4;
    char* title = "多项选择窗口";

    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),title);
    gtk_container_set_border_width(GTK_CONTAINER(window),20);
    g_signal_connect(G_OBJECT(window),"destroy",
G_CALLBACK(gtk_main_quit),NULL);

    box = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),box);
    frame = gtk_frame_new("字体选项：");
    gtk_box_pack_start(GTK_BOX(box),frame,FALSE,FALSE,5);

    box1 = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(frame),box1);
    gtk_container_set_border_width(GTK_CONTAINER(box1),10);
    gtk_widget_show(box);

    check1 = gtk_check_button_new_with_label(" 粗体 ");
    g_signal_connect(G_OBJECT(check1),"released",
            G_CALLBACK(on_check_clicked),(gpointer)1);
    gtk_box_pack_start(GTK_BOX(box1),check1,FALSE,FALSE,3);

    check2 = gtk_check_button_new_with_label(" 斜体 ");
    g_signal_connect(G_OBJECT(check2),"released",
            G_CALLBACK(on_check_clicked),(gpointer)2);
    gtk_box_pack_start(GTK_BOX(box1),check2,FALSE,FALSE,3);

    check3 = gtk_check_button_new_with_label(" 下划线 ");
    g_signal_connect(G_OBJECT(check3),"released",
            G_CALLBACK(on_check_clicked),(gpointer)3);
    gtk_box_pack_start(GTK_BOX(box1),check3,FALSE,FALSE,3);
```

```
    check4 = gtk_check_button_new_with_label(" 彩色 ");
    g_signal_connect(G_OBJECT(check4),"released",
            G_CALLBACK(on_check_clicked),(gpointer)4);
    gtk_box_pack_start(GTK_BOX(box1),check4,FALSE,FALSE,3);

    button = gtk_button_new_from_stock(GTK_STOCK_OK);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_button_clicked),NULL);
    g_signal_connect_swapped(G_OBJECT(button),"clicked",
            G_CALLBACK(gtk_widget_destroy),window);
    gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,5);

    gtk_widget_show_all(window);
    gtk_main();
    return FALSE ;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o checkbutton checkbutton.c 'pkg-config --cflags --libs
gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./checkbutton 即可运行此程序，运行结果如图 4.2 所示。

图 4.2 多选项窗口

实例分析

(1) 创建多选按钮

有 3 个函数可以创建多选按钮，分别是 gtk_check_button_new，创建空的(不显示文字)多选按钮；gtk_check_button_new_with_label，创建带文字的多选按钮；gtk_check_button_new_with_mnemonic，创建带快捷键的多选按钮。它们的参数和创建按钮函数的参数是一样的，可参考第一章中的有关创建按钮的内容。

(2) released 信号

本示例中用到了多选按钮的“released”信号(当鼠标按键放开时发生)，并加回调函数 on_check_clicked 来设定多选按钮的状态。在多选按钮上只要把按下的鼠标键放开，就意味着按钮状态的改变，所以在回调函数中将表示多选按钮状态的逻辑值取反就可以实现对多选按钮状态的跟踪，如例中的 bold = !bold;。

多选按钮控件是继承自按钮控件的，所以它具有按钮控件的所有属性和信号，我们可以像设置按钮控件的属性和连接信号那样，来为多选按钮控件设置属性和连接信号。

4.3　创建一个多项选一的窗口

本节示例将介绍如何创建和使用单选按钮控件，并利用它创建一个多项选一的窗口以及如何判断用户的选择。

实例说明

单选按钮(GtkRadioButton)控件经常用来表示程序中有多种情况只有一种情况的用户选择，此示例就创建了 4 个单选按钮代表颜色，用户只能选择其中一种颜色。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/4
mkdir radio
cd radio
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 radiobutton.c 为文件名保存到当前目录下。

```
/* 单项选择窗口  radiobutton.c */
#include <gtk/gtk.h>
static gchar* red = "  红色  " ;
static gchar* green = "  绿色  " ;
static gchar* yellow = "  黄色  " ;
static gchar* blue = "  蓝色  " ;
void on_radio_clicked  (GtkWidget* radio,gint data)
{
    g_print("你选择的颜色是: ");
    switch((int)data)
    {
    case 1:
        g_print("%s",red); break;
    case 2:
        g_print("%s",green); break;
    case 3:
        g_print("%s",yellow); break;
    case 4:
```

```
            g_print("%s",blue); break;
        }
        g_print("\n");
    }
    //主函数
    int main    (int argc, char *argv[])
    {
        GtkWidget* window;
        GtkWidget* frame;
        GtkWidget* box;
        GtkWidget* box1;
        GtkWidget* button1;
        GtkWidget* radio1;
        GtkWidget* radio2;
        GtkWidget* radio3;
        GtkWidget* radio4;
        GSList* group;
        gchar* title = "单项选择窗口" ;

        gtk_init(&argc,&argv);
        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        gtk_window_set_title(GTK_WINDOW(window),title);
        gtk_container_set_border_width(GTK_CONTAINER(window),10);
        g_signal_connect(G_OBJECT(window),"destroy",
    G_CALLBACK(gtk_main_quit),NULL);
        box = gtk_vbox_new(FALSE,0);
        gtk_container_add(GTK_CONTAINER(window),box);
        frame = gtk_frame_new("请选择一种颜色: ");
        gtk_frame_set_shadow_type(GTK_FRAME(frame),GTK_SHADOW_ETCHED_OUT
    );
        gtk_box_pack_start(GTK_BOX(box),frame,FALSE,FALSE,5);
        box1 = gtk_hbox_new(FALSE,10);
        gtk_container_set_border_width(GTK_CONTAINER(box1),10);
        gtk_container_add(GTK_CONTAINER(frame),box1);
        radio1 = gtk_radio_button_new_with_label(NULL,red);//红色
        g_signal_connect(G_OBJECT(radio1),"released",
                G_CALLBACK(on_radio_clicked),(gpointer)1);
        gtk_box_pack_start(GTK_BOX(box1),radio1,FALSE,FALSE,5);
        //创建多选按钮组
        group = gtk_radio_button_get_group(GTK_RADIO_BUTTON(radio1));
        radio2 = gtk_radio_button_new_with_label(group,green);//绿色
        g_signal_connect(G_OBJECT(radio2),"released",
                G_CALLBACK(on_radio_clicked),(gpointer)2);
        gtk_box_pack_start(GTK_BOX(box1),radio2,FALSE,FALSE,5);

        group = gtk_radio_button_get_group(GTK_RADIO_BUTTON(radio2));
        radio3 = gtk_radio_button_new_with_label(group,yellow);//黄色
        g_signal_connect(G_OBJECT(radio3),"released",
                G_CALLBACK(on_radio_clicked),(gpointer)3);
        gtk_box_pack_start(GTK_BOX(box1),radio3,FALSE,FALSE,5);
```

```
    group = gtk_radio_button_get_group(GTK_RADIO_BUTTON(radio3));
    radio4 = gtk_radio_button_new_with_label(group,blue);//蓝色
    g_signal_connect(G_OBJECT(radio4),"released",
            G_CALLBACK(on_radio_clicked),(gpointer)4);
    gtk_box_pack_start(GTK_BOX(box1),radio4,FALSE,FALSE,5);

    button1 = gtk_button_new_from_stock(GTK_STOCK_OK);
    g_signal_connect(G_OBJECT(button1),"clicked",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_box_pack_start(GTK_BOX(box),button1,FALSE,FALSE,5);

    gtk_widget_show_all(window);
    gtk_main();
    return FALSE ;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o button button.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./button 即可运行此程序，运行结果如图 4.3 所示。

图 4.3　单选项窗口

实例分析

(1) 创建单选按钮

创建单选按钮有多个函数，每个函数的第一个参数都是单向链表(GSList)类型的，表示单选按钮所属的组，在创建一组中的第一个单选按钮时，可以将此参数设为 NULL，而余下的单选按钮在创建时首先要用 gtk_radio_button_get_group 函数来取得第一个按钮所属的组，再用此组做参数来创建单选按钮。还可以用 gtk_radio_button_set_group 函数来设定单选按钮所属的组。

除了上述方式外，还可以用函数 gtk_radio_button_new_from_widget 来依次创建一组单选按钮，它的参数是前一单选按钮的控件指针，有兴趣的读者可以自己试一试。

(2) 单选按钮的信号

本例中为单选按钮的“clicked”信号加回调函数来显示用户的选择，这是最容易理解的一种方式，读者还可以用上节中的“released”信号来处理用户的选择。

单选按钮是继承自多选按钮的，单个的单选按钮和多选按钮在功能上是一样的，理解这一点对运用按钮类控件来编程非常重要。

4.4 创建消息框

本节示例将介绍如何使用 GTK+2.0 中的消息对话框控件和使用技巧。

实例说明

能弹出一个消息框而不是在终端中输出是初学者早就梦想的事情了，GTK+2.0 在这方面特别定制了消息对话框(GtkMessageDialog)控件，它有四种形式，分别是信息、错误、警告和问题。本示例就向读者介绍如何创建使用这四种消息对话框。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/4
mkdir mess
cd mess
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 message.c 为文件名保存到当前目录下。

```
/* 消息对话框测试程序 message.c*/
#include <gtk/gtk.h>

static void
on_button_clicked   (GtkWidget* button,gpointer data)
{
    GtkWidget* dialog ;
    GtkMessageType type ;
    gchar *message;
    switch((int)data)
    {
        case 1 :
            message = "这是一个信息提示对话框. ";
            type = GTK_MESSAGE_INFO ; break;
        case 2 :
            message = "这是一个错误提示对话框. ";
            type = GTK_MESSAGE_ERROR ; break;
        case 3 :
            message = "这是一个问题提示对话框. ";
            type = GTK_MESSAGE_QUESTION ; break;
        case 4 :
            message = "这是一个警告提示对话框. ";
            type = GTK_MESSAGE_WARNING ; break;
    }
    //
    dialog = gtk_message_dialog_new(NULL,
            GTK_DIALOG_MODAL | GTK_DIALOG_DESTROY_WITH_PARENT,
```

```
            type ,
            GTK_BUTTONS_OK,
            message);
    gtk_dialog_run(GTK_DIALOG(dialog));
    gtk_widget_destroy(dialog);
}

int main    (int argc, char* argv[])
{
    GtkWidget* window;
    GtkWidget* frame;
    GtkWidget* box;
    GtkWidget* button1;
    GtkWidget* button2;
    GtkWidget* button3;
    GtkWidget* button4;

    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"创建消息框");
    g_signal_connect(G_OBJECT(window),"destroy",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_container_set_border_width(GTK_CONTAINER(window),20);

    frame = gtk_frame_new("四种消息对话框：");
    gtk_container_add(GTK_CONTAINER(window),frame);

    box = gtk_hbox_new(TRUE,0);
    gtk_container_add(GTK_CONTAINER(frame),box);
    gtk_container_set_border_width(GTK_CONTAINER(box),20);

    button1 = gtk_button_new_from_stock(GTK_STOCK_DIALOG_INFO);
    gtk_box_pack_start(GTK_BOX(box),button1,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button1),"clicked",
            G_CALLBACK(on_button_clicked),(gpointer)1);

    button2 = gtk_button_new_from_stock(GTK_STOCK_DIALOG_ERROR);
    gtk_box_pack_start(GTK_BOX(box),button2,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button2),"clicked",
            G_CALLBACK(on_button_clicked),(gpointer)2);

    button3 = gtk_button_new_from_stock(GTK_STOCK_DIALOG_QUESTION);
    gtk_box_pack_start(GTK_BOX(box),button3,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button3),"clicked",
            G_CALLBACK(on_button_clicked),(gpointer)3);

    button4 = gtk_button_new_from_stock(GTK_STOCK_DIALOG_WARNING);
    gtk_box_pack_start(GTK_BOX(box),button4,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button4),"clicked",
            G_CALLBACK(on_button_clicked),(gpointer)4);
    gtk_widget_show_all(window);
```

```
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) message.c -o message 'pkg-config gtk+-2.0 --cflags --libs'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./message 即可运行此程序，运行结果如图 4.4 所示。

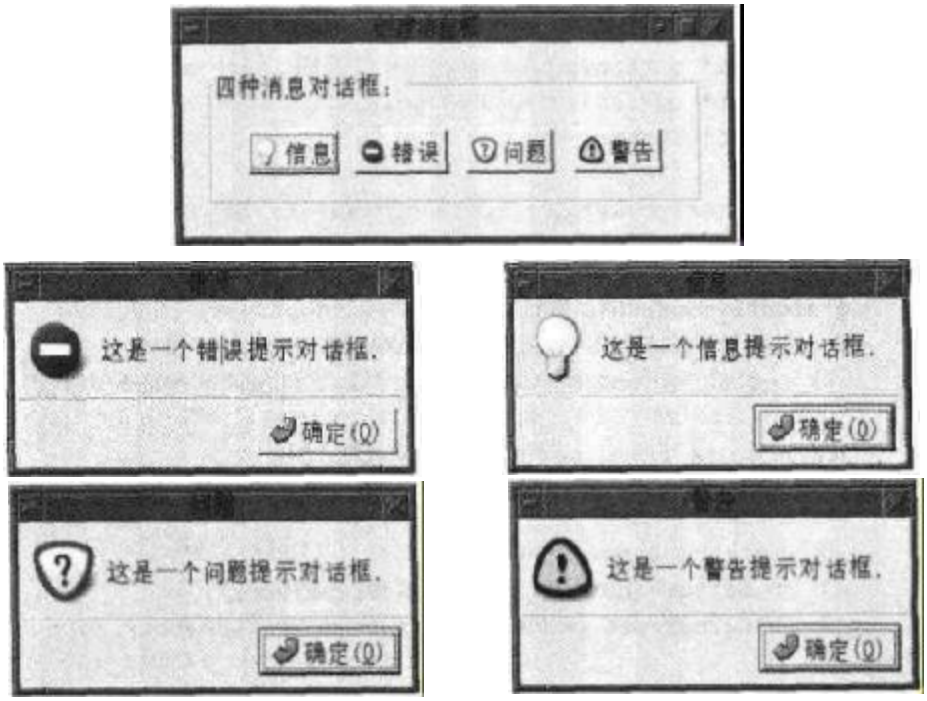

图 4.4 不同样式的消息框

实例分析

(1) 创建消息框控件

消息框控件是 GTK+2.0 中特有的。用 gtk_message_dialog_new 函数来创建，它有如下参数：

① 暂用的父窗口控件，GtkWindow 类型，可以设为 NULL，表示无父窗口。

② 消息框标记，可以取值为 GTK_DIALOG_MODAL，模态对话框；GTK_DIALOG_DES TORY_WITH_PARENT，与父控件一起被销毁；GTK_DIALOG_NO_SEPARATOR，在消息框的按钮上面没有分隔横线。

③ 信息类型，可以取四个宏定义值之一，分别是：GTK_MESSAGE_INFO，信息；GTK_ MESSAGE_ERROR，错误；GTK_MESSAGE_WARNING，警告；GTK_MESSAGE_ QUESTION，问题。

④ 消息框中的按钮类型，有以下宏定义：GTK_BUTTONS_NONE，无按钮；GTK_BUTT_ONS_OK，确认；GTK_BUTTONS_CLOSE，关闭；GTK_BUTTONS_CANCEL，取消；GTK_ YES_NO，是/否按钮；GTK_BUTTONS_OK_CANCEL，确认/取消按钮。

⑤ 格式化的字符串信息，与 g_print 或 printf 格式相同的格式化字符串，用于在消息框中显示输出。

(2) 运行消息框

以上函数返回消息框的控件指针，可以对此指针进行显示、运行等操作。我们这里用的是运行消息框函数 gtk_dialog_run，此函数的参数是消息框的控件指针，返回一个整型值，代表消息框运行时用户点击的按钮标记，编程时根据此标记判断用户的选择。本示例中消息框只用于显示消息目的，所以未加判断，运行结束后直接用 gtk_widget_destroy 函数将其销毁。

本节示例中自定义了一个创建按钮的函数和一个创建消息框的函数，创建按钮的函数在前面章节中出现过，创建消息框函数则是第一次出现，读者可以灵活改动，在自己的程序中创建消息框。

4.5　选择文件和目录

本节示例将介绍如何创建使用文件选择对话框，取得用户的选择以及判断用户选择的是文件还是目录。

实例说明

在图形界面程序中经常用到文件选择对话框(GtkFileSelection)来打开或保存文件，本节示例就向读者介绍这一对话框控件的一般用法。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/4
mkdir filesel
cd filesel
```

创建本节的工作目录，进入目录开始编程。

(2) 打开编辑器，输入以下代码，以 filesel.c 为文件名保存到当前目录下。

```
/* 选择文件对话框  filesel.c */
#include <gtk/gtk.h>
void
on_file_select_ok   (GtkWidget *button,GtkFileSelection *fs)
{
    GtkWidget *dialog;
    gchar message[1024];
    const gchar *filename;
```

```
    filename = gtk_file_selection_get_filename(fs);
    if(g_file_test(filename,G_FILE_TEST_IS_DIR))
        sprintf(message,"你选择的目录是：%s",filename);
    else
        sprintf(message,"你选择的文件是：%s",filename);
    dialog = gtk_message_dialog_new(NULL,
            GTK_DIALOG_DESTROY_WITH_PARENT,
            GTK_MESSAGE_INFO,
            GTK_BUTTONS_OK,
            message,
            NULL);
    gtk_widget_destroy(GTK_WIDGET(fs));
    gtk_dialog_run(GTK_DIALOG(dialog));
    gtk_widget_destroy(dialog);
}
void
on_button_clicked   (GtkWidget *button,gpointer userdata)
{
    GtkWidget* dialog ;
    dialog = gtk_file_selection_new("请选择一个文件或目录：");
    gtk_window_set_position(GTK_WINDOW(dialog),GTK_WIN_POS_CENTER);
    gtk_signal_connect(GTK_OBJECT(dialog),"destroy",
            GTK_SIGNAL_FUNC(gtk_widget_destroy),&dialog);

    gtk_signal_connect(GTK_OBJECT(GTK_FILE_SELECTION(dialog)->ok_but
ton),
            "clicked",
            GTK_SIGNAL_FUNC(on_file_select_ok),
            GTK_FILE_SELECTION(dialog));
    gtk_signal_connect_object(
            GTK_OBJECT(GTK_FILE_SELECTION(dialog)->cancel_button),
            "clicked",
            GTK_SIGNAL_FUNC(gtk_widget_destroy),
            GTK_OBJECT(dialog));
gtk_widget_show(dialog);
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *button;

    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"选择文件和目录对话框");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),20);

    button = gtk_button_new_with_label("按下此按钮来选择文件");
```

```
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_button_clicked),NULL);
    gtk_container_add(GTK_CONTAINER(window),button);
    gtk_widget_show(button);
    gtk_widget_show(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o filesel filesel.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./filesel 即可运行此程序，运行结果如图 4.5 所示。

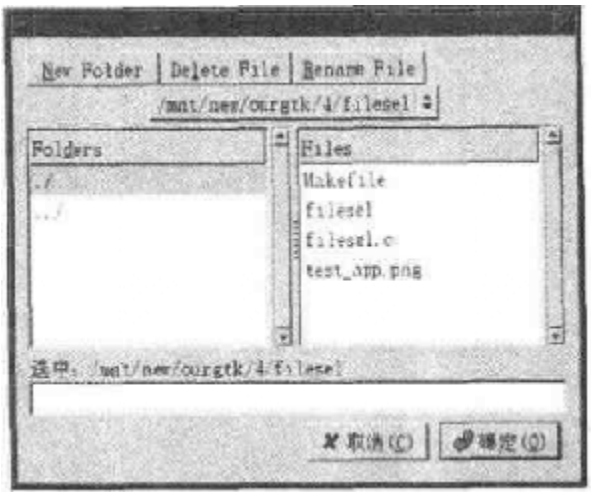

图 4.5　文件选择对话框

实例分析

(1) 文件选择对话框控件

用 gtk_file_selection_new 函数创建文件选择对话框，参数是显示在对话框标题栏的文字。文件选择对话框控件中有两个按钮，确定和取消，它们的控件名称分别是 ok_button 和 cancel_button。如果创建完后直接显示对话框，这两个按钮不起作用，需要为这两个按钮的“clicked”信号加回调函数，方式如代码中所示(需要宏转换)。

(2) 取得用户选择的文件名

函数 gtk_file_selection_get_filename 用来取得用户选择的文件名，这里要注意的是它的返回值类型为 const char*，所以我们先声明的变量也必需是这一类型的，此函数的参数是文件选择对话框型的，要用 GTK_FILE_SELECTION 宏来转换一下。

(3) 文件名和目录名

还可以用 GLIB 中提供文件实用功能中的 g_file_test 函数来测试一下选择的文件名是否为目录，只要将此函数的第二个参数设为 G_FILE_TEST_IS_DIR 就可以了，如果是目录则

返回值为 TRUE，否则的话为 FALSE。

文件选择对话框在编程中经常用到，尤其在为其中的确认和取消按钮添加回调函数时更应注意，本例中用的是 gtk_signal_connect 宏和 GTK_SIGNAL_FUNC 回调类型，读者可以试一试将其改为 g_signal_connect 宏和 G_CALLBACK 回调类型。

4.6 选 择 字 体

本节介绍如何创建使用字体选择对话框，取得用户选择的字体名称。

实例说明

字体选择控件(GtkFontSelection)和字体选择对话框(GtkFontSelectionDialog)的源代码在 GTK+2.0 中被重写了，其目的就是为提高编程执行效率、增强功能和让界面更美观。本示例演示了字体选择对话框的一般用法。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/4
mkdir fontsel
cd fontsel
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 fontsel.c 为文件名保存到当前目录下。

```
/* 选择字体 fontsel.c */
#include <gtk/gtk.h>
void
on_font_select_ok   (GtkWidget *button,GtkFontSelectionDialog *fs)
{
    GtkWidget *dialog;
    gchar message[1024];
    gchar *s = gtk_font_selection_dialog_get_font_name(fs);
    sprintf(message,"你选择的字体是: %s \n",s);
    dialog = gtk_message_dialog_new(NULL,
            GTK_DIALOG_DESTROY_WITH_PARENT,
            GTK_MESSAGE_INFO,
            GTK_BUTTONS_OK,
            message,
            NULL);
    g_free(s);
    gtk_widget_destroy(GTK_WIDGET(fs));
    gtk_dialog_run(GTK_DIALOG(dialog));
    gtk_widget_destroy(dialog);
}
void
on_button_clicked   (GtkWidget *button,gpointer userdata)
```

```
{
    GtkWidget* dialog ;
    dialog = gtk_font_selection_dialog_new("请选择一种字体: ");
    gtk_window_set_position(GTK_WINDOW(dialog),GTK_WIN_POS_CENTER);
    gtk_signal_connect(GTK_OBJECT(dialog),"destroy",
            GTK_SIGNAL_FUNC(gtk_widget_destroy),&dialog);
    gtk_signal_connect(
        GTK_OBJECT(GTK_FONT_SELECTION_DIALOG(dialog)->ok_button),
"clicked",GTK_SIGNAL_FUNC(on_font_select_ok),
        GTK_FONT_SELECTION_DIALOG(dialog));
        gtk_signal_connect_object(

    GTK_CBJECT(GTK_FONT_SELECTION_DIALOG(dialog)->cancel_button),
        "clicked",GTK_SIGNAL_FUNC(gtk_widget_destroy),
        GTK_OBJECT(dialog));
    gtk_widget_show(dialog);
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *button;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"字体选择功能实现");
    gtk_window_set_default_size(GTK_WINDOW(window),500,100);
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),40);

    button = gtk_button_new_with_label("按下此按钮来选择字体");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_button_clicked),NULL);
    gtk_container_add(GTK_CONTAINER(window),button);
    gtk_widget_show(button);

    gtk_widget_show(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o fontsel fontsel.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./fontsel 即可运行此程序，运行结果如图 4.6 所示。

图 4.6 选择字体对话框

实例分析

(1) 字体选择对话框

与文件选择对话框一样，字体选择对话框也包括一个确定按钮(ok_button)和一个取消按钮(cancel_button)，编程时也必需为这两个按钮加回调函数，否则不起作用。可以用函数 gtk_font_selection_dialog_get_font_name 来取得用户选择的字体名，还可以用此字体名来创建 Pango 字体对象在编程中使用。

(2) GdkFont

字体和颜色、光标等都是 GDK 的资源，由 X 服务器统一管理和分配。GTK 通过 GDK 程序库来创建这些资源，GdkFont 是其中之一，用函数 gtk_font_selection_dialog_get_font 来取得字体选择对话框中用户选中的字体，还可以用 gdk_font_*系列函数来对 GdkFont 结构进行操作，更详细的信息参考 GDK 的 API 参考手册。

GTK+2.0 中的国际化字体和文本格式化输出主要都是由 Pango 程序库来完成的，所以要想使用好字体功能，一定要认真看一下有关 Pango 的文档或 Pango 的 API 参考手册。与文件选择对话框不同，字体选择对话框包括字体选择控件，而且字体选择控件和其他控件一样是可以创建并加到其他窗口中去的，有兴趣的读者可以试着做一下。

4.7 选 择 颜 色

本节示例将介绍如何创建使用颜色选择对话框，取得用户选择的颜色，并为绘图区设定为用户选择的颜色。

实例说明

颜色选择控件(GtkColorSelection)和颜色选择对话框(GtkColorSelectionDialog)的源代码在 GTK+2.0 中也被重写了，而且界面更加漂亮了。本节示例就向读者介绍如何应用颜色选择对话框为绘图区(GtkDrawingArea)改变颜色。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/4
mkdir colorsel
cd colorsel
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 color.c 为文件名保存到当前目录下。

```
/* 颜色选择 color.c */
#include <gtk/gtk.h>
//以下几个控件定义为全局静态的，以便其他函数来使用或控制它们
static GtkWidget *window;
static GdkColor color;
static GtkWidget *drawarea;
void    cn_ok_clicked   (GtkButton* button,gpointer data)
{
    GtkWidget *dialog ;
    GtkColorSelection *colorsel;
    gint response;
    dialog = gtk_color_selection_dialog_new("选择颜色");
    gtk_window_set_transient_for (GTK_WINDOW (dialog),
 GTK_WINDOW (window));
    //注意：转换颜色选择对话框的颜色选择子控件
colorsel = GTK_COLOR_SELECTION
(GTK_COLOR_SELECTION_DIALOG(dialog)->colorsel);

    gtk_color_selection_set_has_opacity_control(colorsel,TRUE);
    gtk_color_selection_set_has_palette(colorsel,TRUE);
    gtk_signal_connect(GTK_OBJECT(dialog),"destroy",
             GTK_SIGNAL_FUNC(gtk_widget_destroy),&dialog);

    gtk_color_selection_set_previous_color (colorsel, &color);
    gtk_color_selection_set_current_color (colorsel, &color);
    response = gtk_dialog_run(GTK_DIALOG(dialog));
    //注意：此处需要用户判断对话框的运行结果，如果是确定按钮则改变颜色
    if(response == GTK_RESPONSE_OK )
    {
        gtk_color_selection_get_current_color (colorsel, &color);
        gtk_widget_modify_bg (drawarea, GTK_STATE_NORMAL, &color);
    }
    gtk_widget_destroy(dialog);
}
int main ( int argc , char* argv[])
{
    GtkWidget *vbox;
    GtkWidget *frame;
    GtkWidget *button;
```

```
    color.red = 0;
    color.blue = 65535;
    color.green = 0;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"选择颜色");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),20);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    frame = gtk_frame_new(NULL);
    gtk_frame_set_shadow_type(GTK_FRAME(frame),GTK_SHADOW_IN);
    gtk_box_pack_start(GTK_BOX(vbox),frame,TRUE,TRUE,5);
    drawarea = gtk_drawing_area_new();
    gtk_widget_set_size_request(drawarea,200,200);//设定控件的大小
    gtk_widget_modify_bg (drawarea, GTK_STATE_NORMAL, &color);
//改变控件正常情况下的背景
    gtk_container_add(GTK_CONTAINER(frame),drawarea);

    button = gtk_button_new_from_stock(GTK_STOCK_SELECT_COLOR);
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(on_ok_clicked),NULL);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o selcolor color.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./selcolor 即可运行此程序，运行结果如图 4.7 所示。

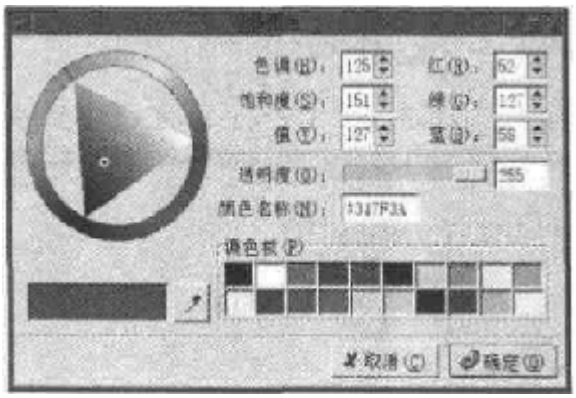

图 4.7 选择颜色对话框

实例分析

(1) 颜色选择对话框

颜色选择对话框在创建方法上和字体选择对话框是一样的，不同的是它有两个属性可以设定一下，一是透明度，用函数 gtk_color_selection_set_has_opacity_control 来设定；一是调色板，用函数 gtk_color_selection_set_has_palette 来设定，默认情况下这两个子控件并不显示出来。多用 gtk_color_selection 系列函数来对对话框内的 GtkColorSelection 控件进行操作。

(2) GdkColor

GdkColor 结构包括三个主要的值，red、green、blue，分别表示红、绿、蓝三种颜色的值，可以取值 0 到 65535，这一结构主要体现的就是所谓的 RGB 形式。本例中的这三个值中 green 取值为 65535，其他为 0，则表示这个值为绿色。

(3) 绘图区控件

绘图区控件主要是定制用户界面中的元素的，以使其显示不同的颜色、形状、大小等。这里主要用 gtk_widget_modify_bg 函数来改变它的背景颜色，此控件的进一步使用见第 6 章高级控件。

颜色选择对话框和字体选择对话框一样，也有一个颜色选择控件可单独使用，或创建后加入到其他的控件中去，而且在编程中经常用到。

4.8 选 择 日 期

本节示例将介绍如何创建并灵活使用日历控件，取得/设定日历控件的时间。

实例说明

日历控件(GtkCalenda)在编程中的主要作用是选择和设定日期，本例中将日历控件加入到一个窗口中，并加了一个按钮，形成日期选择对话框。当点击按钮后会在终端中输出用户选择的日期。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/4
mkdir calenda
cd calenda
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 cale.c 为文件名保存到当前目录下。

```
/* 选择日期 cale.c */
#include <gtk/gtk.h>
void on_ok_clicked (GtkButton* button,gpointer data)
{
    guint year;
```

```
    guint month;
    guint date;
    //注意月份是从0开始的,即0-11
    gtk_calendar_get_date(GTK_CALENDAR(data),&year,&month,&date);
    g_print("你选择的日期是 %d 年 %d 月 %d 日\n",year,month+1,date);
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *box;
    GtkWidget *calendar;
    GtkWidget *button;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"选择日期");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),20);

    box = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),box);
    calendar = gtk_calendar_new();
    gtk_box_pack_start(GTK_BOX(box),calendar,FALSE,FALSE,5);
    button = gtk_button_new_from_stock(GTK_STOCK_OK);
    gtk_box_pack_start(GTK_BOX(box),button,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_ok_clicked),calendar);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o cale cale.c 'pkg-config --cflags
--libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./cale 即可运行此程序，运行结果如图 4.8 所示。

图 4.8 选择日期

实例分析

(1) 取得日历控件的日期

用函数 gtk_calendar_new 创建日历控件，用函数 gtk_calendar_get_date 来取得当前用户选择的日期，它的后 3 个参数分别为代表年、月、日的整型指针，由于是用来存贮这 3 个值的，所以在声明时声明

为整型，引用其地址。

(2) 设定日历控件的时间

还可以用函数 gtk_calendar_select_month 和 gtk_calendar_select_day 来设定日历控件的月份和日期，用函数 gtk_calendar_display_options 来设定日历控件的显示状态，如不显示年份表头、不显示月份表头、星期从星期一开始等。函数 gtk_calendar_mark_day 用来标记特定的日期，函数 gtk_calendar_unmark_day 来作反标记，也可以用 gtk_calendar_clear_marks 来清除标记。

日历控件还有一些特定的信号，如“day_selected”(用户选择日期时发出)、“month_changed”(月份改变时发出)等，编程中为这些信号加回调函数可以达到相应的功能或效果，读者可以自己试一试。

4.9　确认/取消对话框

本节示例将介绍如何灵活使用常用的控件来创建自定义的确认/取消对话框，并在编程中运用此对话框取得用户的选择。

实例说明

确认/取消对话框最常用在程序运行中询问用户是否确认某一操作，如按确认按钮则继续操作，如按取消按钮则恢复原状态不变。本示例就用窗口和一个标签、一个图像、两个按钮来模拟很多软件中都出现过的这类对话框，并达到了此功能。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/4
mkdir okcancel
cd okcancel
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 okcancel.c 为文件名保存到当前目录下。

```
/* 确认/取消对话框 okcancel.c */
#include <gtk/gtk.h>
void    on_ok_clicked   (GtkButton* button,gpointer data)
{
    g_print("你选择的是确认。\n");
}
void    on_cancel_clicked   (GtkButton* button,gpointer data)
{
    g_print("你选择的是取消。\n");
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
```

```
    GtkWidget *vbox;
    GtkWidget *hbox;
    GtkWidget *bbox;
    GtkWidget *button;
    GtkWidget *label;
    GtkWidget *image;
    GtkWidget *sep;
    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),"确定/取消对话框");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    image = gtk_image_new_from_stock
(GTK_STOCK_DIALOG_WARNING,GTK_ICON_SIZE_DIALOG);
    gtk_box_pack_start(GTK_BOX(hbox),image,FALSE,FALSE,5);
    label = gtk_label_new("这是一个确定/取消对话框。\n根据选择做出相应的操作。
");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    sep = gtk_hseparator_new();
    gtk_box_pack_start(GTK_BOX(vbox),sep,FALSE,FALSE,5);
    bbox = gtk_hbutton_box_new();
    gtk_button_box_set_layout(GTK_BUTTON_BOX(bbox),GTK_BUTTONBOX_SPR
EAD);
    gtk_box_pack_start(GTK_BOX(vbox),bbox,FALSE,FALSE,5);
    button = gtk_button_new_from_stock(GTK_STOCK_OK);
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(on_ok_clicked),NULL);
    g_signal_connect_swapped(G_OBJECT(button),"clicked",
                G_CALLBACK(gtk_widget_destroy),window);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,0);

    button = gtk_button_new_from_stock(GTK_STOCK_CANCEL);
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(on_cancel_clicked),NULL);
    g_signal_connect_swapped(G_OBJECT(button),"clicked",
                G_CALLBACK(gtk_widget_destroy),window);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,0);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
```

```
all:
    $(CC) -o okcancel okcancel.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./okcancel 即可运行此程序，运行结果如图 4.9 所示。

图 4.9　确定/取消对话框

实例分析

(1) 信号连接

本示例中的两个按钮都有两个信号连接，每个按钮的其第二个信号连接都是 g_signal_connect_swapped，表示上一个连接的信号回调函数执行完后，继续执行这个连接的函数。而且连接的函数都是 gtk_widget_destroy，参数都是 window，即无论按下哪个按钮，在执行完输出信息后就马上销毁窗口，由于窗口的“delete_event”信号连接了 gtk_main_quit 函数，所以程序结束运行。

读者自己完全可以剪裁一下代码，做成自定义对话框，把它加入到自己编写的软件中去。

4.10　是/否/取消对话框

本节示例将介绍如何灵活运用常用控件设计是/否/取消对话框，和取得此自定义对话框的用户选择。

实例说明

上节示例中的对话框只有两种选择，在软件运行中还经常有三种选择的对话框出现，如编辑软件在运行时如果未保存编辑结果就关闭的话，就会弹出是/否/取消这个有三种选择的对话框，本节示例模拟了这一对话框的基本功能。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/4
mkdir yesno
cd yesno
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 yesno.c 为文件名保存到当前目录下。

```
/* 是/否/取消对话框  yesno.c */
#include <gtk/gtk.h>
void    on_yes_clicked  (GtkButton* button,gpointer data)
{
    g_print("你选择的是 \"是\" 。\n");
}
void    on_no_clicked   (GtkButton* button,gpointer data)
{
    g_print("你选择的是 \"否\" 。\n");
}
void    on_cancel_clicked   (GtkButton* button,gpointer data)
{
    g_print("你选择的是 \"取消\" 。\n");
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *hbox;
    GtkWidget *bbox;
    GtkWidget *button;
    GtkWidget *label;
    GtkWidget *image;
    GtkWidget *sep;
    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),"是/否/取消对话框");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);

    image = gtk_image_new_from_stock
(GTK_STOCK_DIALOG_WARNING,GTK_ICON_SIZE_DIALOG);
    gtk_box_pack_start(GTK_BOX(hbox),image,FALSE,FALSE,5);
    label = gtk_label_new("这是一个 [ 是/否/取消 ] 对话框。\n根据选择做出相
应的操作。");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    sep = gtk_hseparator_new();
    gtk_box_pack_start(GTK_BOX(vbox),sep,FALSE,FALSE,5);
    bbox = gtk_hbutton_box_new();
    gtk_button_box_set_layout(GTK_BUTTON_BOX(bbox),GTK_BUTTONBOX_SPR
EAD);
    gtk_box_pack_start(GTK_BOX(vbox),bbox,FALSE,FALSE,5);
```

```
    buttor = gtk_button_new_from_stock(GTK_STOCK_YES);
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(on_yes_clicked),NULL);
    g_signal_connect_swapped(G_OBJECT(button),"clicked",
                G_CALLBACK(gtk_widget_destroy),window);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,0);
    button = gtk_button_new_from_stock(GTK_STOCK_NO);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_no_clicked),NULL);
    g_signal_connect_swapped(G_OBJECT(button),"clicked",
                G_CALLBACK(gtk_widget_destroy),window);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,0);
    button = gtk_button_new_from_stock(GTK_STOCK_CANCEL);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_cancel_clicked),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,0);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o yesno yesno.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./yesno 即可运行此程序，运行结果如图 4.10 所示。

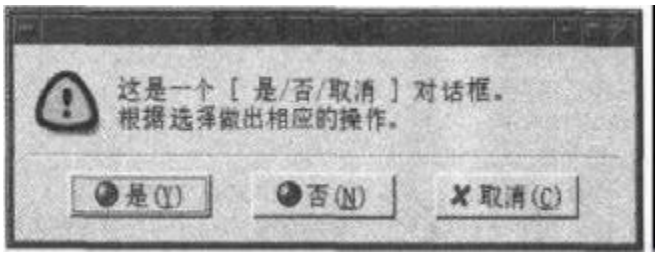

图 4.10　是/否/取消对话框

实例分析

(1) 控件的布局

无论是做一个对话框还是做一个窗口，都要根据头脑中的控件的形状、大小、位置设计出一个良好的布局。此对话框先在窗口中加一个纵向盒状容器，然后依次加入一个横向盒状容器、一个分隔横线和一个按钮盒，再分别向横向盒状容器中加入图像控件和标签控件，向按钮盒中加入 3 个按钮。这并不是惟一的布局方式，不过基本上满足了要求，事实上还可以改成其他方式。

本节示例是上节示例的延续，有兴趣的读者可以将 3 个按钮的回调函数设为一个，通过所加参数来改变输出结果。

4.11 关于对话框

本节示例将介绍如何使用多页显示控件来创建一个类似 GNOME2 桌面环境的关于对话框。

实例说明

关于对话框在软件中经常见到，我们这里仿照 GNOME2 桌面环境中多数应用程序的关于对话框的形式，应用多页显示控件(GtkNotebook)实现这一功能。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/4
mkdir about
cd about
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 about.c 为文件名保存到当前目录下。

```
/* 关于对话框 about.c */
#include <gtk/gtk.h>
static  GtkWidget* credits_window;
GtkWidget*  create_credits  ()
{
    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *notebook;
    GtkWidget *page;
    GtkWidget *label;
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"开发人员");
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    notebook = gtk_notebook_new();
    gtk_box_pack_start(GTK_BOX(vbox),notebook,TRUE,TRUE,5);
    page = gtk_vbox_new(FALSE,0);
    label = gtk_label_new("杨文齐，ywq51305@fm365.com");
    gtk_box_pack_start(GTK_BOX(page),label,FALSE,FALSE,5);
    label = gtk_label_new("宋国伟，gwsong52@sohu.com");
    gtk_box_pack_start(GTK_BOX(page),label,FALSE,FALSE,5);
    label = gtk_label_new("编辑");
    gtk_notebook_append_page(GTK_NOTEBOOK(notebook),page,label);
    page = gtk_vbox_new(FALSE,0);
    label = gtk_label_new("郝春艳，gwsong_jl@sohu.com");
    gtk_box_pack_start(GTK_BOX(page),label,FALSE,FALSE,5);
    label = gtk_label_new("测试");
```

```
    gtk_notebook_append_page(GTK_NOTEBOOK(notebock),page,label);
    gtk_widget_show_all(window);
    return window;
}
void    show_credits    ()
{
    credits_window = create_credits();
    gtk_widget_show(credits_window);
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *bbox;
    GtkWidget *vbox;
    GtkWidget *label;
    GtkWidget *image;
    GtkWidget *sep;
    GtkWidget *button;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"关于对话框");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    image = gtk_image_new_from_file("gnome-gmush.png");
    gtk_box_pack_start(GTK_BOX(vbox),image,FALSE,FALSE,5);
    label = gtk_label_new(NULL);
    gtk_label_set_markup(GTK_LABEL(label),"<span><big>GTK+2.0 实例编
程</big></span>");
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,5);
    label = gtk_label_new("版权所有：清华大学出版社\n作者：宋国伟");
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,5);
    sep = gtk_hseparator_new();
    gtk_box_pack_start(GTK_BOX(vbox),sep,FALSE,FALSE,5);
    bbox = gtk_hbutton_box_new();
    gtk_button_box_set_layout(GTK_BUTTON_BOX(bbox),GTK_BUTTONBOX_EDG
E);
    gtk_box_pack_start(GTK_BOX(vbox),bbox,FALSE,FALSE,5);
    button = gtk_button_new_with_label("开发人员");
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,25);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(show_credits),NULL);
    button = gtk_button_new_from_stock(GTK_STOCK_OK);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,35);
    gtk_widget_show_all(window);
```

```
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o about about.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./about 即可运行此程序，运行结果如图 4.11 所示。

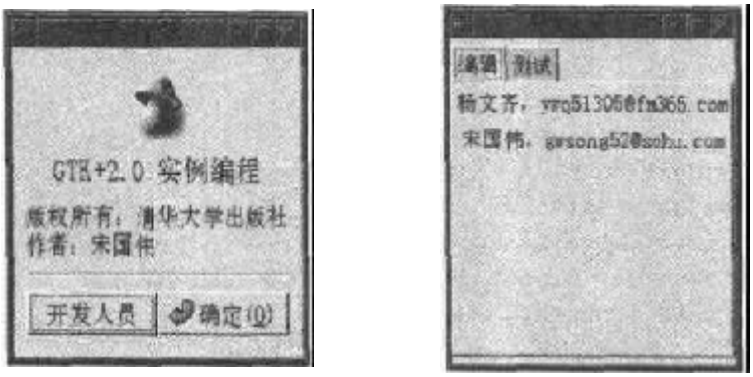

图 4.11 关于对话框

实例分析

(1) 多页显示控件

可以用函数 gtk_notebook_new 来创建空的多页显示控件，然后用 gtk_notebook_append_page 函数来向多页显示控件添加显示页，此函数有 3 个参数，分别为多页显示控件本身、页控件、显示页的标题控件。一般而言每个页都是一个容器，再向其中加一些其他控件做为页的内容。本例中容器用的是纵向盒状容器，添加的内容是标签控件。

多页显示控件是较复杂的控件，还可以设定页标题的弹出式菜单、页标题的位置，向其中动态增加/删除显示页等，详细情况可见 GTK+2.0 的 API 参考手册中的 GtkNotebook 一节。

模拟现有的一些功能简单的程序或程序的一部分，是初学者学习编程、提高编程水平的一个很好的途径，在模拟之前最好先简单规划一下，写出大致的轮廓。

本章向读者介绍了编程中经常用到的对话框。其中包括系统提供的文件选择、字体选择、颜色选择、消息框、自定义对话框和我们自己编写代码的对话框，其中多数程序中都有一个主窗口，让读者能达到一看即懂的目的。在复杂的或大型的软件中，在程序中灵活运用对话框对改善程序的运行效果和增强程序的功能都能起到非常关键的作用。

第5章 综 合 应 用

本章重点：

前4章的内容已经介绍了大部分关于GTK+2.0中常用控件的用法和技巧。本章在已学内容的基础上，综合应用这些功能和技巧，编写了两个常见的工具计算器和每日提示；本章中还介绍了如何在应用程序中实现定时的功能和GTK+1.2中较常用到的复杂控件CList和CTree的用法以及如何实现多窗口同时显示和控制等。

本章主要内容：

- 计算器程序的实现
- 定时功能
- 每日提示的实现
- CList和CTree控件的使用
- 多窗口功能的实现

5.1 计 算 器

本节将介绍如何应用按钮控件、单行输入控件构建计算器的外观，设计函数实现计算器的功能，从而实现一个简单的计算器软件。

实例说明

很多计算机软件模拟了一些现实生活中常见工具的外观和功能，其中计算器软件应该是最典型的一个了，同时它也是初学者入门学习稍复杂一些编程的最好的一个范例。本节示例就实现了一个最简单的计算器软件。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk
mkdir 5
cd 5
mkdir calc
cd calc
```

创建本章总目录5，创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以calc.c为文件名保存到当前目录下。

```
/* 计算器 calc.c */
#include <gtk/gtk.h>
#include <stdlib.h>
```

```
static GtkWidget *entry;    //定义单行输入控件来显示输入输出的数字
gint count = 0 ;    //计位
gdouble nn = 0 ;    //计数
gdouble mm = 0 ;    //计数二
gint  s = 0 ;       //算法
gboolean first = TRUE; //首次输入
gboolean have_dot = FALSE; //是否有小数点
gboolean have_result = FALSE;  //是否有结果输出
gchar number[100]; //保存用户输入的数字
void    clear_all   (void)
{   //清除所有相关标记
    gint i;
    gtk_entry_set_text(GTK_ENTRY(entry),"");
    nn = 0;
    mm = 0 ;
    s = 0;
    count = 0 ;
    first = TRUE;
    have_dot = FALSE;
    have_result = FALSE;
    for(i = 0 ; i < 100 ; i++)
        number[i] = '\0';
}
void    on_num_clicked  (GtkButton* button,gpointer data)
{   //当数字键按下时执行
    const gchar *num;
    gint i;
    if(have_result)
        clear_all();//有结果则全部清除
    if(count == 6) return;  //够6位数则不能再输入数字
    count++;
    num = gtk_button_get_label(GTK_BUTTON(button));//取数
    i = g_strlcat(number,num,100);//保存
    if(first)
        nn = strtod(number,NULL);//数
    else
        mm = strtod(number,NULL);//数
    gtk_entry_set_text(GTK_ENTRY(entry),number);//显示
}
void    on_dot_clicked      (GtkButton* button,gpointer data)
{   //当小数点按下时
    gint i;
    if(have_result)
        clear_all();//全部清除
    if(have_dot == FALSE)   //如果无小数点则可以
    {
        have_dot = TRUE;
        i = g_strlcat(number,".",100);
        gtk_entry_set_text(GTK_ENTRY(entry),number);
    }
    //如果有小数点则不输出
```

```
}
void    on_clear_clicked    (GtkButton* button,gpointer data)
{
    clear_all();//全部清除
}
void    on_suan_clicked     (GtkButton* button,gpointer data)
{   //当计算按钮 +, -, *, / 按下时
    gint i;
    switch(GPOINTER_TO_INT(data))
    {
    case 1: //"+"
        s = 1;
        gtk_entry_set_text(GTK_ENTRY(entry),"");
        first = FALSE ; count = 0; break;
    case 2: //"-"
        s = 2;
        gtk_entry_set_text(GTK_ENTRY(entry),"");
        first = FALSE ; count = 0; break;
    case 3: //"*"
        s = 3;
        gtk_entry_set_text(GTK_ENTRY(entry),"");
        first = FALSE ; count = 0; break;
    case 4: //"/"
        s = 4;
        gtk_entry_set_text(GTK_ENTRY(entry),"");
        first = FALSE ; count = 0; break;
    }
    have_dot = FALSE;
    for(i = 0 ; i < 100 ; i++) //清除数字
        number[i] = '\0';
}
void on_eq_clicked  (GtkButton* button,gpointer data)
{   //当等号键按钮按下时
    double numb;
    gchar num[100];
    gchar *result;
    switch(s)
    {
    case 1: //"+"
        numb = nn+mm;
        break;
    case 2: //"-"
        numb = nn-mm;
        break;
    case 3: //"*"
        numb = nn*mm;
        break;
    case 4: //"/"
        numb = nn/mm;
        break;
    }
```

```
    result = g_ascii_dtostr(num,100,numb);
    gtk_entry_set_text(GTK_ENTRY(entry),result);
    have_result = TRUE;
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *hbox,*hbox1,*hbox2,*hbox3,*hbox4;
    GtkWidget *button;
    GtkWidget *label;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
           G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"计算器");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);

    label = gtk_label_new("Calculator");
    gtk_box_pack_start(GTK_BOX(hbox),label,TRUE,TRUE,5);
    button = gtk_button_new_with_label("C");
    gtk_box_pack_start(GTK_BOX(hbox),button,TRUE,TRUE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(on_clear_clicked),NULL);

    entry = gtk_entry_new();
    gtk_widget_set_direction(entry,GTK_TEXT_DIR_RTL);
    gtk_box_pack_start(GTK_BOX(vbox),entry,FALSE,FALSE,5);
    hbox1 = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox1,FALSE,FALSE,5);
    button = gtk_button_new_with_label("3");
    gtk_box_pack_start(GTK_BOX(hbox1),button,TRUE,TRUE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(on_num_clicked),NULL);
    button = gtk_button_new_with_label("2");
    gtk_box_pack_start(GTK_BOX(hbox1),button,TRUE,TRUE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
           G_CALLBACK(on_num_clicked),NULL);
    button = gtk_button_new_with_label("1");
    gtk_box_pack_start(GTK_BOX(hbox1),button,TRUE,TRUE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(on_num_clicked),NULL);
    button = gtk_button_new_with_label("+");
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(on_suan_clicked),(gpointer)1);
    gtk_box_pack_start(GTK_BOX(hbox1),button,TRUE,TRUE,5);
```

```
hbox2 = gtk_hbox_new(FALSE,0);
gtk_box_pack_start(GTK_BOX(vbox),hbox2,FALSE,FALSE,5);
button = gtk_button_new_with_label("6");
g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_num_clicked),NULL);
gtk_box_pack_start(GTK_BOX(hbox2),button,TRUE,TRUE,5);
button = gtk_button_new_with_label("5");
g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_num_clicked),NULL);
gtk_box_pack_start(GTK_BOX(hbox2),button,TRUE,TRUE,5);
button = gtk_button_new_with_label("4");
g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_num_clicked),NULL);
gtk_box_pack_start(GTK_BOX(hbox2),button,TRUE,TRUE,5);
button = gtk_button_new_with_label("-");
g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_suan_clicked),(gpointer)2);
gtk_box_pack_start(GTK_BOX(hbox2),button,TRUE,TRUE,5);
hbox3 = gtk_hbox_new(FALSE,0);
gtk_box_pack_start(GTK_BOX(vbox),hbox3,FALSE,FALSE,5);
button = gtk_button_new_with_label("9");
g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_num_clicked),NULL);
gtk_box_pack_start(GTK_BOX(hbox3),button,TRUE,TRUE,5);
button = gtk_button_new_with_label("8");
g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_num_clicked),NULL);
gtk_box_pack_start(GTK_BOX(hbox3),button,TRUE,TRUE,5);
button = gtk_button_new_with_label("7");
g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_num_clicked),NULL);
gtk_box_pack_start(GTK_BOX(hbox3),button,TRUE,TRUE,5);
button = gtk_button_new_with_label("*");
g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_suan_clicked),(gpointer)3);
gtk_box_pack_start(GTK_BOX(hbox3),button,TRUE,TRUE,5);

hbox4 = gtk_hbox_new(FALSE,0);
gtk_box_pack_start(GTK_BOX(vbox),hbox4,FALSE,FALSE,5);
button = gtk_button_new_with_label("0");
g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_num_clicked),NULL);
gtk_box_pack_start(GTK_BOX(hbox4),button,TRUE,TRUE,5);
button = gtk_button_new_with_label(".");
g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_dot_clicked),NULL);
gtk_box_pack_start(GTK_BOX(hbox4),button,TRUE,TRUE,5);
button = gtk_button_new_with_label("=");
g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_eq_clicked),NULL);
gtk_box_pack_start(GTK_BOX(hbox4),button,TRUE,TRUE,5);
```

```
    button = gtk_button_new_with_label("/");
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(on_suan_clicked),(gpointer)4);
    gtk_box_pack_start(GTK_BOX(hbox4),button,TRUE,TRUE,5);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o calc calc.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./calc 即可运行此程序，运行结果如图 5.1 所示。

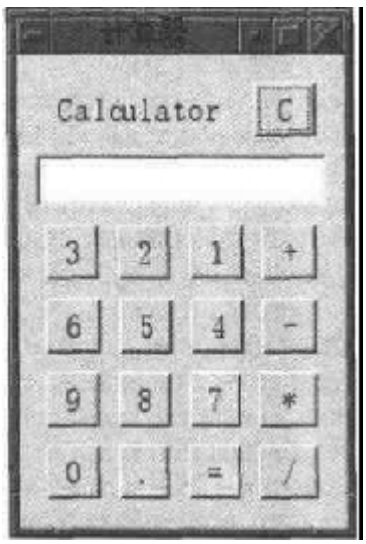

图 5.1 计算器

实例分析

(1) 构建界面

此示例采用一个纵向盒状容器和多个横向盒状容器来设计界面。事实上采用格状容器是一个比较好的选择，读者可以自己试着改一下。另外在单行输入控件的阅读方向属性上用函数 gtk_widget_set_direction 设定为 GTK_TEXT_DIR_RTL，即从右向左，使此控件的数字显示更像一个计算器的显示屏。

(2) 数字按钮的回调函数

数字按钮 0～9 的功能是向计算器输入数字，通过计算和定位，最后显示出来，所以它们的回调函数都是同一个，只不过传递的参数根据不同的按钮传递不同的数字。

(3) 小数点按钮的回调函数

小数点按钮的回调函数是较特殊的一个。首先小数点在一个数中只能出现一次，再按的话则不起作用，小数点按钮在点击后要对所输入的数字做一下处理，使之成为浮点数，

并且让后续数字也转换为相应的浮点数字。

(4) 运算按钮的回调函数

此计算器软件只有加减乘除 4 种运算功能，这 4 个运算按钮用了同一个回调函数，根据传递的参数设定当前的计算方法。

(5) 结果按钮的回调函数

等号按钮的回调函数是根据当前的计算方法进行相应的运算，并将运算结果输出到单行输出控件中。

(6)清除按钮的回调函数

清除按钮的功能是清除当前显示，同时清除所有标记，将标记设为默认等。

本例中还多次用到了 stdlib.h 和 glib 中的字符串转换为浮点数和浮点数转换为字符串等转换函数，它们的详细用法可参见相关的参考手册。

此计算器软件的实现并未做相应的编码前设计。步骤是先设计界面，再为按钮加回调函数，再进一步协调各回调函数间的功能，做出各种标记性变量，如当前运算方法、数一和数二的存贮、是否有小数点、运行结果的显示和转换等。读者在分析源程序时可以按以上层次进行分析，逐步理解。

从此程序中可以看出 GTK+2.0 编程界面代码占了很大一部分，而且很复杂，仔细分析就会发现。其实这是一些排列控件函数和信号连接宏的简单组合，只要掌握了容器和控件的用法，很容易设计出界面来。

5.2 计　时　器

本节示例将介绍如何使用 GTK+2.0 中的定时功能和为定时功能加定时执行的函数，来实现一个简单的数字时钟和计时器。

实例说明

定时功能在程序中用途非常广泛，本节示例设计了一个数字时钟和计时器，单击【开始】按钮开始计时，单击【停止】按钮则停止计时观看计时结果。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/5
mkdir timer
cd timer
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 timer.c 为文件名保存到当前目录下。

```
/* 时钟与计时  timer.c */
#include <gtk/gtk.h>
#include <time.h>
static GtkWidget *clocker;
```

```
static GtkWidget *our_timer;
static GtkWidget *button_begin;//开始按钮
static GtkWidget *button_end;  //结束按钮
gint    timer_id;   //时钟标记
gint    hour = 0;   //时
gint    min = 0;    //分
gint    sec = 0;    //秒
void    clock_begin     ()
{   //时钟开始
    time_t now ;
    struct tm *l_time;
    gchar buf[100];
    now = time((time_t *)NULL);
    l_time = localtime(&now);   //取本地时间
    sprintf(buf,"%d:%d:%d",l_time->tm_hour,l_time->tm_min,l_time->tm
_sec);
    gtk_label_set_text(GTK_LABEL(clocker),buf);
}
void    timer_add       ()
{   //以秒为单位计时
    gchar buf[100];
    sec++;
    if(sec == 60)   //60秒
    {
        min++;
        if(min == 60)   //60分
        {
            hour++;
            min = 0;
        }
        sec = 0;
    }
    sprintf(buf,"%d : %d : %d",hour,min,sec);
    gtk_label_set_text(GTK_LABEL(our_timer),buf);
}
void    timer_begin     ()
{   //开始计时
    gtk_widget_set_sensitive(button_begin,FALSE);
    gtk_widget_set_sensitive(button_end,TRUE);
    timer_id = gtk_timeout_add(1000,(GtkFunction)timer_add,NULL);
}
void    timer_end       ()
{   //结束计时
    gtk_widget_set_sensitive(button_begin,TRUE);
    gtk_widget_set_sensitive(button_end,FALSE);
    gtk_timeout_remove(timer_id);
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox;
```

```
    GtkWidget *hbox;
    GtkWidget *sep;
    GtkWidget *label;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"时钟与计时");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    label = gtk_label_new("简单的数字时钟：");
    gtk_box_pack_start(GTK_BOX(hbox),label,TRUE,TRUE,5);
    clocker = gtk_label_new(NULL);
    gtk_box_pack_start(GTK_BOX(hbox),clocker,TRUE,TRUE,5);
    gtk_timeout_add(1000,(GtkFunction)clock_begin,NULL);
    sep = gtk_hseparator_new();
    gtk_box_pack_start(GTK_BOX(vbox),sep,FALSE,FALSE,5);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    label = gtk_label_new("简单的计时器：");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);

    our_timer = gtk_label_new(NULL);
    gtk_box_pack_start(GTK_BOX(hbox),our_timer,FALSE,FALSE,5);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,TRUE,TRUE,5);
    button_begin = gtk_button_new_with_label("开始");
    g_signal_connect(G_OBJECT(button_begin),"clicked",
                G_CALLBACK(timer_begin),NULL);
    gtk_box_pack_start(GTK_BOX(hbox),button_begin,TRUE,TRUE,5);
    button_end = gtk_button_new_with_label("停止");
    gtk_widget_set_sensitive(button_end,FALSE);
    g_signal_connect(G_OBJECT(button_end),"clicked",
                G_CALLBACK(timer_end),NULL);
    gtk_box_pack_start(GTK_BOX(hbox),button_end,TRUE,TRUE,5);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o timer timer.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./timer 即可运行此程序，运行结果如图 5.2 所示。

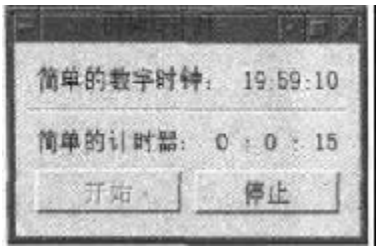

图 5.2 计时器

实例分析

(1) GTK+2.0 的超时功能

GTK+2.0 中超时功能是主循环功能的一部分(GtkTimeout)，可以在界面设计代码中直接为主循环加入超时控制，也可以在回调函数中为主循环加入超时控制，或删除超时控制。

(2) 添加计时器

使用函数 gtk_timeout_add 来为程序添加超时功能，它有 3 个参数，第 1 个参数是计时的间隔时间，以千分之一秒为单位，如果是 1000 的话则表示每隔一秒钟执行一次；第 2 个参数是要执行的函数名，用 GtkFunction 函数来将其转换为函数指针；第 3 个参数是传递给执行函数的参数，为 gpointer 类型，它返回一个整型值表示此时钟的标记。如果想销毁或暂停此时钟，可以用函数 gtk_timeout_remove 来删除此时钟，它的参数就是这个标记。

用函数 gtk_widget_set_sensitive 来设定按钮控件是否可用，第二个参数为 TRUE 则为可用，为 FALSE 则不可用，其他控件也一样。

数字时钟主要是在定时函数中更改文字标签来实现的，此示例的取本地时间函数用到了标准 C 语言中的 time.h 头文件，其中的数据结构和相关函数的说明可见 C 语言的参考手册。

5.3 简单动画实现

本节将介绍如何使用时钟定时动态控制图像的显示内容，实现一个简单的动画。

实例说明

GTK+2.0 本身包含的程序库 gdk-pixbuf，主要是用来处理图像的，其中就包括动画。网页中包含的 GIF 格式动画比较常用，本节示例通过时钟的控制来实现类似的简单的动画。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/5
mkdir gif
cd gif
```

创建本节的工作目录，并进入此目录开始编程。绘制 4 幅图像，使之连续起来有一定的动感，规格为 32×32 像素，命名为 p1.bmp、p2.bmp、p3.bmp、p4.bmp，保存到此目录下。

(2) 打开编辑器，输入以下代码，以 gif.c 为文件名保存到当前目录下。

```
/* 简单的动画  gif.c */
#include <gtk/gtk.h>
static GtkWidget *ourgif ;
static gchar *bmpfile[4] = {"p1.bmp","p2.bmp","p3.bmp","p4.bmp" };
gint i = 0 ;
void    change_bmp  ()
{   //依次更改图像实现动态效果
    gtk_image_set_from_file(GTK_IMAGE(ourgif),bmpfile[i]);
    i++ ;
    if ( i == 4 ) i = 0 ;
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *hbox;
    GtkWidget *label;
    GtkWidget *image;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"简单的动画");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);

    label = gtk_label_new("直接引用GIF动画");
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,5);
    image = gtk_image_new_from_file("hahaboy.gif");
    gtk_box_pack_start(GTK_BOX(vbox),image,FALSE,FALSE,5);
    label = gtk_label_new("四幅静态的图像");
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,5);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    image = gtk_image_new_from_file("p1.bmp");
    gtk_box_pack_start(GTK_BOX(hbox),image,FALSE,FALSE,5);
    image = gtk_image_new_from_file("p2.bmp");
    gtk_box_pack_start(GTK_BOX(hbox),image,FALSE,FALSE,5);
    image = gtk_image_new_from_file("p3.bmp");
    gtk_box_pack_start(GTK_BOX(hbox),image,FALSE,FALSE,5);
    image = gtk_image_new_from_file("p4.bmp");
    gtk_box_pack_start(GTK_BOX(hbox),image,FALSE,FALSE,5);
    label = gtk_label_new("通过时钟控制的动画");
```

```
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,5);
    ourgif = gtk_image_new_from_file("p1.bmp");
    gtk_box_pack_start(GTK_BOX(vbox),ourgif,FALSE,FALSE,5);
    gtk_timeout_add(150,(GtkFunction)change_bmp,NULL);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o gif gif.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./gif 即可运行此程序，运行结果如图 5.3 所示。

图 5.3 简单的动画

实例分析

与第 3 章中的坦克游戏一样，用函数 gtk_image_set_form_file 来改变图像控件所显示的图像，先定义好文件名字符串数组，然后依次显示。

通过时钟控制图像的显示来实现动画功能，是图形界面编程的旧办法了，不过这是向初学者介绍时钟功能的一个好例子。

5.4 每 日 提 示

本节示例将介绍如何灵活运用各种控件，实现一些常见的软件中的每日提示功能。

实例说明

很多大型的软件出于向用户解释其使用功能的目的，在一开始执行时都显示一个每日提示对话框，每次运行显示不同的内容，介绍其使用方法，并且可以上下翻页，本节示例

就在实现此功能上进行了尝试，核心功能主要是更换标签的内容。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/5
mkdir tips
cd tips
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 tips.c 为文件名保存到当前目录下。

```
/* 每日提示 tips.c */
#include <gtk/gtk.h>
//XPM格式的图像数据，以C语言源代码形式存于文件中
static char * book_open_xpm[] = {
    "16 16 4 1",
    "       c None s None",
    ".      c black",
    "X      c #808080",
    "o      c white",
    "                ",
    "  ..            ",
    " .Xo.    ...     ",
    " .Xoo. ..oo.     ",
    " .Xooo.Xooo...   ",
    " .Xooo.oooo.X.   ",
    " .Xooo.Xooo.X.   ",
    " .Xooo.oooo.X.   ",
    " .Xooo.Xoco.X.   ",
    " .Xooo.ooco.X.   ",
    "  .Xoo.Xoo..X.   ",
    "   .Xo.o..ooX.   ",
    "    .X..XXXXX.   ",
    "    ..X.......   ",
    "     ..          ",
    "                "};
//自定义提示信息
static gchar *info[5] = {
"此软件用于测试每日提示功能的实现，如果你发现问题请及时回复。",
"我们的目的是把GTK+2.0的大多数功能奉献给每一位自由软件爱好者和开发者。",
"每一位Linux的支持者都会让我们增加一分信心，Linux最终仍是台式计算机操作系统。",
"计算机软件技术是一种科学技术，它和人类历史上其他的科学技术一样，是允许每一人自由使用的。",
"当前你测试完此程序后，请设法把它附加到你创作的软件当中去，这是你成功的第一步。"
};

static GtkWidget *window;   //主窗口
static GtkWidget *frame;    //框架
static GtkWidget *pre_button;   //上一提示按钮
```

```
static GtkWidget *next_button; //下一提示按钮
static GtkWidget *label;    //提示信息内容标签
static GtkWidget *title;    //框架的标题
gint current_info = 0 ;//当前提示信息计数
GtkWidget*  create_title    (GtkWidget *data)
{   //创建框架控件的标题
    GtkWindow *title;
    GtkWidget *hbox;
    GtkWidget *image;
    GtkWidget *label;
    GdkPixmap *pixmap;
    GdkBitmap *mask;
    pixmap = gdk_pixmap_create_from_xpm_d(
                data->window,&mask,
                &GTK_WIDGET(data)->style->white,
                book_open_xpm);
    image = gtk_image_new_from_pixmap(pixmap,NULL);
    label = gtk_label_new("新的标题");
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(hbox),image,FALSE,FALSE,2);
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,2);
    return hbox ;
}
GtkWidget*  create_button   (gchar* stockid,gchar* title)
{   //创建带图像的按钮
    GtkWidget *button;
    GtkWidget *image;
    GtkWidget *label;
    GtkWidget *hbox;
    image = gtk_image_new_from_stock(stockid,GTK_ICON_SIZE_MENU);
    label = gtk_label_new(title);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(hbox),image,FALSE,FALSE,3);
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,3);
    button = gtk_button_new();
    gtk_container_add(GTK_CONTAINER(button),hbox);
    return button;
}
void    pre_info    (GtkButton *button,gpointer data)
{   //上一提示
    gint i ;
    i = current_info - 1 ;
    if(i == -1) return ;
    if(i == 0 ) gtk_widget_set_sensitive(pre_button,FALSE);
    current_info = i ;
    gtk_widget_set_sensitive(next_button,TRUE);
    gtk_label_set_text(GTK_LABEL(label),info[current_info]);
}
void    next_info   (GtkButton *button,gpointer data)
{   //下一提示
    gint i ;
```

```
    i = current_info + 1 ;
    if(i == 5) return ;
    if(i == 4) gtk_widget_set_sensitive(next_button,FALSE);
    current_info = i ;
    gtk_widget_set_sensitive(pre_button,TRUE);
    gtk_label_set_text(GTK_LABEL(label),info[current_info]);
}
int main ( int argc , char* argv[])
{
    GtkWidget *hbox ;
    GtkWidget *vbox;
    GtkWidget *bbox;
    GtkWidget *button;
    GtkWidget *image;
    GtkWidget *title;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"每日提示");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
    gtk_widget_realize(window);

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,TRUE,TRUE,5);
    image = gtk_image_new_from_file("gnome-gmush.png");
    gtk_box_pack_start(GTK_BOX(hbox),image,FALSE,FALSE,5);
    frame = gtk_frame_new(NULL);
    title = create_title(window);
    gtk_frame_set_label_widget(GTK_FRAME(frame),title);
    gtk_box_pack_start(GTK_BOX(hbox),frame,TRUE,TRUE,5);
    label =  gtk_label_new(NULL);
    gtk_label_set_text(GTK_LABEL(label),info[0]);
    gtk_label_set_line_wrap(GTK_LABEL(label),TRUE);
    gtk_container_add(GTK_CONTAINER(frame),label);
    bbox = gtk_hbutton_box_new();
    gtk_button_box_set_layout(GTK_BUTTON_BOX(bbox),GTK_BUTTONBOX_END
);
    gtk_box_set_spacing(GTK_BOX(bbox),5);
    gtk_box_pack_start(GTK_BOX(vbox),bbox,FALSE,FALSE,5);
    button = gtk_check_button_new_with_label("每次启动时显示");
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,5);
    pre_button = create_button(GTK_STOCK_GO_BACK,"上一提示");
    gtk_widget_set_sensitive(pre_button,FALSE);
    g_signal_connect(G_OBJECT(pre_button),"clicked",
                G_CALLBACK(pre_info),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),pre_button,FALSE,FALSE,5);
    next_button = create_button(GTK_STOCK_GO_FORWARD,"下一提示");
```

```
    g_signal_connect(G_OBJECT(next_button),"clicked",
                G_CALLBACK(next_info),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),next_button,FALSE,FALSE,5);
    button = gtk_button_new_from_stock(GTK_STOCK_OK);
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(gtk_main_quit),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,5);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o tips tips.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./tips 即可运行此程序，运行结果如图 5.4 所示。

图 5.4 每日提示

实例分析

(1) 将图像数据添加到程序中去

此程序的特殊之处在于将一段图像数据直接保存到代码中来，使我们不用考虑外部资源，这在第 3 章的 pixmap 一节中曾出现过(它采用包含文件形式)，也可以将另一幅图像 gnome-gmush.png 也转换成此种资源保存进来，有兴趣的读者不妨试一试。

(2) 更改提示内容

显示内容以静态数组的形式保存在程序中，此处综合运用了 gtk_lable_set_text 函数和 gtk_widget_set_sensitive 函数，来更改显示的内容和设定控钮控件是否可用。

在应用软件中，每日提示是一个很好的修饰和辅助功能，通过对它的学习您可以很快掌握 GTK+2.0 编程中常见的技巧，并能综合使用这些技巧，使您的编程水平步上一个新的台阶。

5.5 表 格 软 件

本节示例将介绍如何创建 GTK+2.0 中的列表控件以及一些常用的列表控件的使用技巧，来丰富程序的功能。

实例说明

表格软件在应用中非常广泛，GTK+1.2 中的列表控件(GtkCList)在 GTK+2.0 中功能并未改变，由于很多读者并不熟悉，所以本例中加以介绍。本节示例采用列表控件为主体，实现了向表格中添加数据，点击相应的按钮会跳转到相应行等功能。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/5
mkdir grid
cd grid
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 grid.c 为文件名保存到当前目录下。

```
/* 表格 grid.c */
#include <gtk/gtk.h>
//定义列表的列标题
static gchar* titles[5]  = {"编号","姓名","性别","出生年月","电子邮件"};
const gchar *new_row[5] ;//定义字符串数组指针，指向要向表格中保存的数据
static  GtkWidget *clist;   //列表
static  GtkWidget *add_win; //添加数据窗口
static  GtkWidget *entry_id;    //编号
static  GtkWidget *entry_name;  //姓名
static  GtkWidget *entry_sex;   //性别
static  GtkWidget *entry_birthday;  //出生年月
static  GtkWidget *entry_email;     //电子邮件
gint current_row = 0 ;  //当前行
gint row_count = 0 ;    //总行数
void    on_ok_clicked   (GtkButton *button,gpointer data)
{
    new_row[0] = gtk_entry_get_text(GTK_ENTRY(entry_id));
    new_row[1] = gtk_entry_get_text(GTK_ENTRY(entry_name));
    new_row[2] = gtk_entry_get_text(GTK_ENTRY(entry_sex));
    new_row[3] = gtk_entry_get_text(GTK_ENTRY(entry_birthday));
    new_row[4] = gtk_entry_get_text(GTK_ENTRY(entry_email));
    //以上代码取得用户输入
    row_count++;
    gtk_clist_append(GTK_CLIST(clist),new_row);//向表格中添加
    gtk_widget_destroy(add_win);
}
void    on_cancel_clicked   (GtkButton *button,gpointer data)
{
    gtk_widget_destroy(add_win);//销毁添加数据的窗口
}
GtkWidget*  create_addwin   (void)
{   //创建添加数据窗口
    GtkWidget* win;
```

```
    GtkWidget* vbox;
    GtkWidget* table;
    GtkWidget* bbox;
    GtkWidget* label;
    GtkWidget* button;
    win = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(win),"添加记录");
    gtk_window_set_position(GTK_WINDOW(win),GTK_WIN_POS_CENTER);
    g_signal_connect(G_OBJECT(win),"delete_event",
            G_CALLBACK(gtk_widget_destroy),win);
    gtk_container_set_border_width(GTK_CONTAINER(win),10);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(win),vbox);
    table = gtk_table_new(5,2,FALSE);
    gtk_box_pack_start(GTK_BOX(vbox),table,FALSE,FALSE,5);
    label = gtk_label_new("编号");
    gtk_table_attach_defaults(GTK_TABLE(table),label,0,1,0,1);
    entry_id = gtk_entry_new();
    gtk_table_attach_defaults(GTK_TABLE(table),entry_id,1,2,0,1);

    label = gtk_label_new("姓名");
    gtk_table_attach_defaults(GTK_TABLE(table),label,0,1,1,2);
    entry_name = gtk_entry_new();
    gtk_table_attach_defaults(GTK_TABLE(table),entry_name,1,2,1,2);
    label = gtk_label_new("性别");
    gtk_table_attach_defaults(GTK_TABLE(table),label,0,1,2,3);
    entry_sex = gtk_entry_new();
    gtk_table_attach_defaults(GTK_TABLE(table),entry_sex,1,2,2,3);
    label = gtk_label_new("出生年月");
    gtk_table_attach_defaults(GTK_TABLE(table),label,0,1,3,4);
    entry_birthday = gtk_entry_new();
    gtk_table_attach_defaults(GTK_TABLE(table),entry_birthday,1,2,3,
4);
    label = gtk_label_new("电子邮件");
    gtk_table_attach_defaults(GTK_TABLE(table),label,0,1,4,5);
    entry_email = gtk_entry_new();
    gtk_table_attach_defaults(GTK_TABLE(table),entry_email,1,2,4,5);
    bbox = gtk_hbutton_box_new();
    gtk_box_pack_start(GTK_BOX(vbox),bbox,FALSE,FALSE,5);
    gtk_box_set_spacing(GTK_BOX(bbox),5);
    gtk_button_box_set_layout(GTK_BUTTON_BOX(bbox),GTK_BUTTONBOX_END
);
    button = gtk_button_new_from_stock(GTK_STOCK_OK);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_ok_clicked),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,5);
    button = gtk_button_new_from_stock(GTK_STOCK_CANCEL);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_cancel_clicked),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,5);
    gtk_widget_show_all(win);
```

```
    return win;
}
GtkWidget*  create_button   (gchar* stockid)
{   //创建带图像的按钮
    GtkWidget *button;
    GtkWidget *image;
    image = gtk_image_new_from_stock(stockid,GTK_ICON_SIZE_MENU);
    button = gtk_button_new();
    gtk_container_add(GTK_CONTAINER(button),image);
    return button;
}
void    goto_first  (GtkButton *button,gpointer data)
{   //转首行
    current_row = 0 ;
    gtk_clist_select_row(GTK_CLIST(clist),current_row,0);
}
void    goto_last   (GtkButton *button,gpointer data)
{   //转尾行
    current_row = row_count-1 ;
    gtk_clist_select_row(GTK_CLIST(clist),current_row,0);
}
void    go_back     (GtkButton *button,gpointer data)
{   //前一行
    current_row -- ;
    if(current_row == -1) return ;
    gtk_clist_select_row(GTK_CLIST(clist),current_row,0);
}
void    go_forward  (GtkButton *button,gpointer data)
{   //下一行
    current_row ++;
    if(current_row > row_count ) return;
    gtk_clist_select_row(GTK_CLIST(clist),current_row,0);
}
void    append_row  (GtkButton *button,gpointer data)
{   //添加数据
    add_win = create_addwin();
    gtk_widget_show(add_win);
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *bbox;
    GtkWidget *button;
    GtkTooltips* button_tips;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"列表软件");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
```

```
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    clist = gtk_clist_new_with_titles(5,titles);
    gtk_box_pack_start(GTK_BOX(vbox),clist,TRUE,TRUE,5);

    bbox = gtk_hbutton_box_new();
    button_tips = gtk_tooltips_new();
    gtk_box_pack_start(GTK_BOX(vbox),bbox,FALSE,FALSE,5);
    gtk_box_set_spacing(GTK_BOX(bbox),5);
    gtk_button_box_set_layout(GTK_BUTTON_BOX(bbox),GTK_BUTTONBOX_END
);
    gtk_button_box_set_child_size(GTK_BUTTON_BOX(bbox),20,20);
    button = create_button(GTK_STOCK_GOTO_FIRST);
    gtk_tooltips_set_tip(GTK_TOOLTIPS(button_tips),button,
                "转到首行","首行");
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(goto_first),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,2);
    button = create_button(GTK_STOCK_GO_BACK);
    gtk_tooltips_set_tip(GTK_TOOLTIPS(button_tips),button,
                "转到前一行","前一行");
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(go_back),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,2);
    button = create_button(GTK_STOCK_GO_FORWARD);
    gtk_tooltips_set_tip(GTK_TOOLTIPS(button_tips),button,
                "转到下一行","下一行");
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(go_forward),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,2);
    button = create_button(GTK_STOCK_GOTO_LAST);
    gtk_tooltips_set_tip(GTK_TOOLTIPS(button_tips),button,
                "转到尾行","尾行");
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(goto_last),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,2);
    button = create_button(GTK_STOCK_ADD);
    gtk_tooltips_set_tip(GTK_TOOLTIPS(button_tips),button,
                "新增一行","新增");
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(append_row),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,2);
    button = create_button(GTK_STOCK_QUIT);
    gtk_tooltips_set_tip(GTK_TOOLTIPS(button_tips),button,
                "退出","退出");
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(gtk_main_quit),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,5);
    gtk_widget_show_all(window);
    gtk_main();
```

```
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o grid grid.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./grid 即可运行此程序，运行结果如图 5.5 所示。

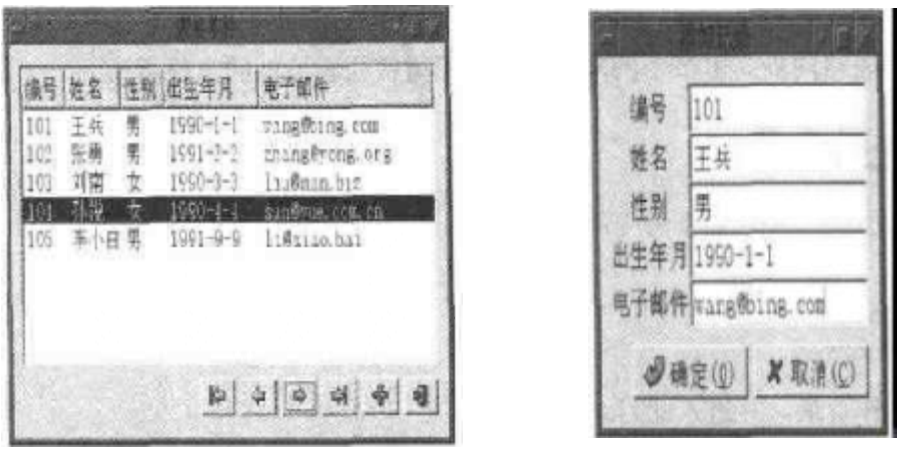

图 5.5　表格软件

实例分析

(1) 创建列表控件

有两个函数可以用来创建列表控件，其中 gtk_clist_new 函数创建一个空的列表控件，参数是一个整型值表示表格的列数；还可以用 gtk_clist_new_with_titles 函数来创建一个显示标题的空的列表控件，第一个参数是表格的列数，第二个参数是表格标题的字符串数组。用第一种方法创建列表控件后，可以用 gtk_clist_set_column_title 函数来设定设置列的标题。

(2) 向表格控件中添加数据

用函数 gtk_clist_append 向列表控件追加数据，数据的格式必须是一个字符串数组的指针，而且要保证它们的一致性。用函数 gtk_clist_select_row 来选择数据行。

本示例用到了两个窗口，主窗口和输入窗口。为了向主窗口中传输子窗口中的数据，我们把子窗口中的单行录入控件指针定义为全局变量，这样在执行插入操作时就可以直接引用这些控件指针来取得输入数据，再把这些数据保存到表格控件中。这并不是实现此功能的惟一方法，但可能是最简单的方法之一。读者还可以用传递结构型数据的方法来实现。

为使外部函数能销毁输入窗口，一定要把输入窗口的指针定义为全局的，在为子窗口的“delete_event”事件添加回调函数时用函数 gtk_widget_destroy，将其参数指定为此窗口的指针，就可以达到此目的。

表格控件在 GTK 编程中非常有用。您还可以向表格中添加图像等数据，对数据进行排序，设定表格控件的背景，设定各单元格的属性等等。它的缺点是不能编辑单元格。GTK+2.0

中新增的树视图控件(GtkTreeView)则具有这一功能，将在下章中介绍。

5.6 树 状 表 格

本节示例将介绍如何创建使用与列表控件相类似的另一种控件——树状列表控件，和此控件的一些使用技巧。

实例说明

与列表控件功能相似的控件还有树状列表控件(GtkCTree)，它的特色是能显示树状控件，并且和表格同步。本节示例向读者演示了树状列表控件的一般用法。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/5
mkdir ctree
cd ctree
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 ctree.c 为文件名保存到当前目录下。

```
/* 树状列表 ctree.c */
#include <gtk/gtk.h>
static gchar *title[2] = {"班级","人数"};
//表示打开状态的图像数据
static char * book_open_xpm[] = {
    "16 16 4 1",
    "       c None s None",
    ".      c black",
    "X      c #808080",
    "o      c white",
    "                ",
    "  ..            ",
    " .Xo.    ...     ",
    " .Xoo. ..oo.     ",
    " .Xooo.Xooo...   ",
    " .Xooo.oooo.X.   ",
    " .Xooo.Xooo.X.   ",
    " .Xooo.oooo.X.   ",
    " .Xooo.Xooo.X.   ",
    " .Xooo.oooo.X.   ",
    "  .Xoo.Xoo..X.   ",
    "   .Xo.o..coX.   ",
    "    .X..XXXXX.   ",
    "    ..X.......   ",
    "     ..          ",
    "                "};
```

```
//表示关闭状态的图像数据
static char * book_closed_xpm[] = {
    "16 16 6 1",
    "       c None s None",
    ".      c black",
    "X      c red",
    "o      c yellow",
    "O      c #808080",
    "#      c white",
    "                ",
    "       ... .    ",
    "     ..XX.      ",
    "   ..XXXXX.     ",
    " ..XXXXXXXX.    ",
    ".ooXXXXXXXXX.   ",
    "..ooXXXXXXXXX.  ",
    ".X.ooXXXXXXXXX. ",
    ".XX.ooXXXXXX..  ",
    " .XX.ooXXX..#O  ",
    "  .XX.oo..##OO. ",
    "   .XX..##OO..  ",
    "    .X.#OO..    ",
    "     ..O..      ",
    "      ..        ",
    "                "};
//表示最后一级的图像数据
static char * mini page xpm[] = {
    "16 16 4 1",
    "       c None s None",
    ".      c black",
    "X      c white",
    "o      c #808080",
    "                ",
    "   .......      ",
    "   .XXXXX..     ",
    "   .XoooX.X.    ",
    "   .XXXXX....   ",
    "   .XooooXoo.o  ",
    "   .XXXXXXXX.o  ",
    "   .XooooooX.o  ",
    "   .XXXXXXXX.o  ",
    "   .XooooooX.o  ",
    "   .XXXXXXXX.o  ",
    "   .XooooooX.o  ",
    "   .XXXXXXXX.o  ",
    "   ..........o  ",
    "    oooooooooo  ",
    "                "};

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

```
    GtkWidget *window, *vbox;
    GdkPixmap *pix1,*pix2,*pix3;
    GtkWidget *ctree, *button;
    GtkCTreeNode *parent,*child;
    gchar *root[2], *text1[2], *text2[2], *sub1[2], *sub2[2], *sub3[2];
    root[0] = "一年级";
    root[1] = "学生总数：120人";
    text1[0] = "女生";
    text1[1] = "20人";
    text2[0] = "男生";
    text2[1] = "20人";
    sub1[0] = "一年一班";
    sub1[1] = "40人";
    sub2[0] = "一年二班";
    sub2[1] = "40人";
    sub3[0] = "一年三班";
    sub3[1] = "40人";
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"树状列表");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
    gtk_widget_realize(window);//如此之后即可创建图像
    pix1 = gdk_pixmap_create_from_xpm_d(window->window,
            NULL,&GTK_WIDGET(window)->style->white,book_open_xpm);
    pix2 = gdk_pixmap_create_from_xpm_d(window->window,
            NULL,&GTK_WIDGET(window)->style->white,book_closed_xpm);
    pix3 = gdk_pixmap_create_from_xpm_d(window->window,
            NULL,&GTK_WIDGET(window)->style->white,mini_page_xpm);
    //以上创建三幅图像
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    ctree = gtk_ctree_new_with_titles(2,0,title);
    gtk_clist_set_column_width(GTK_CLIST(ctree),0,150);
    gtk_clist_set_column_width(GTK_CLIST(ctree),1,200);
    gtk_box_pack_start(GTK_BOX(vbox),ctree,TRUE,TRUE,5);
    parent = gtk_ctree_insert_node(GTK_CTREE(ctree),NULL,NULL,root,5,
                    pix2,NULL,pix1,NULL,FALSE,TRUE);//根节点
    child =
gtk_ctree_insert_node(GTK_CTREE(ctree),parent,NULL,sub1,5,
                    pix2,NULL,pix1,NULL,FALSE,TRUE);//子节点一
    gtk_ctree_insert_node(GTK_CTREE(ctree),child,NULL,text1,5,
                pix3,NULL,NULL,NULL,TRUE,TRUE);//叶一
    gtk_ctree_insert_node(GTK_CTREE(ctree),child,NULL,text2,5,
                pix3,NULL,NULL,NULL,TRUE,TRUE);//叶二
    child =
gtk_ctree_insert_node(GTK_CTREE(ctree),parent,NULL,sub2,5,
                    pix2,NULL,pix1,NULL,FALSE,TRUE);//子节点二
    gtk_ctree_insert_node(GTK_CTREE(ctree),child,NULL,text1,5,
```

```
                pix3,NULL,NULL,NULL,TRUE,TRUE);
    gtk_ctree_insert_node(GTK_CTREE(ctree),child,NULL,text2,5,
                pix3,NULL,NULL,NULL,TRUE,TRUE);
    child =
gtk_ctree_insert_node(GTK_CTREE(ctree),parent,NULL,sub3,5,
                    pix2,NULL,pix1,NULL,FALSE,TRUE);//子节点三
    gtk_ctree_insert_node(GTK_CTREE(ctree),child,NULL,text1,5,
                pix3,NULL,NULL,NULL,TRUE,TRUE);
    gtk_ctree_insert_node(GTK_CTREE(ctree),child,NULL,text2,5,
                pix3,NULL,NULL,NULL,TRUE,TRUE);
            button = gtk_button_new_from_stock(GTK_STOCK_QUIT);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o ctree ctree.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./ctree 即可运行此程序，运行结果如图 5.6 所示。

图 5.6　树状列表

实例分析

(1) 创建树状列表控件

用函数 gtk_ctree_new_with_titles 来创建树状列表，它有 3 个参数，第 1 个表示树状列表的列数；第 2 个参数表示那一列具有树状结构，0 表示左侧；第 3 个参数是字符串数组，表示列的标题。

(2) 向树状列表中添加数据和结点

与树状列表控件相关的 2 个数据结构是 GtkCTreeNode 和 GtkCTreeRow，一个表示树

的结点，一个表示树状列表的行。用函数 gtk_ctree_insert_node 来向树状列表插入节点，返回该节点的指针，参数包括父节点指针、子节点指针(可为 NULL)、显示的数据、展开和关闭时的图像、是否为叶节点、是否可以展开等。

树状列表是一种非常有效的表示数据间树状关系的控件，它的功能也十分强大，而且速度较快，本示例只是演示它的初级功能而已，需要注意的是由于它的 API 函数有些过于复杂，GTK 的开发人员正在对其进行修改，所以在编程时一定要看一下它的 API 函数是否有变化。GTK+2.0 中新增的树视图控件也具有树状显示功能，将在下一章中详细介绍。

5.7 多窗口功能的实现

本节示例将介绍如何在程序中控制多个窗口的显示。

实例说明

有些软件在功能上要求有多个窗口同时显示在屏幕上，主窗口要能控制子窗口的显示、隐藏、接收和传输数据等。本节示例在此功能的实现上做了初步尝试。

实现步骤

(1) 打开终端输入以下命令：

```
cd ~/ourgtk/5
mkdir twowin
cd twowin
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 twowin.c 为文件名保存到当前目录下。

```
/* 多窗口应用程序  twowin.c */
#include <gtk/gtk.h>
static  GtkWidget *main_window; //主窗口
static  GtkWidget *sub_window;  //子窗口
gboolean ishide = FALSE;    //是否隐藏
void    on_show     (GtkButton* button,gpointer data)
{   //如果未隐藏则恢复显示，由图标恢复为窗口
    if(ishide ==FALSE)
        gtk_window_deiconify (GTK_WINDOW(sub_window));
    else
    {//如果隐藏则显示
        gtk_widget_show(sub_window);
        ishide = FALSE;
    }
}
void    on_hide     (GtkButton* button,gpointer data)
{   //图标最小化
    gtk_window_iconify (GTK_WINDOW(sub_window));
}
void    on_sub_delete   (GtkWidget *window,GdkEvent *event,gpointer
```

```
data)
{   //隐藏
    gtk_widget_hide(window);
    ishice = TRUE;
}
GtkWidget*  create_main_window  ()
{   //创建主窗口
    GtkWidget *window;
    GtkWidget *hbox;
    GtkWidget *button;
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"主窗口");
gtk_container_set_border_width(GTK_CONTAINER(window),10);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),hbox);
    button = gtk_button_new_with_label("显示子窗口");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_show),NULL);
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);
    button = gtk_button_new_with_label("隐藏子窗口");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_hide),NULL);
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);
    gtk_widget_show_all(window);
    return window ;
}
GtkWidget*  create_sub_window   ()
{   //创建子窗口
    GtkWidget *window1;
    GtkWidget *viewport;
    GtkWidget *label;
    window1 = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window1),"delete_event",
            G_CALLBACK(on_sub_delete),window1);
    gtk_window_set_title(GTK_WINDOW(window1),"子窗口");
    gtk_window_set_default_size(GTK_WINDOW(window1),500,100);
    gtk_window_set_position(GTK_WINDOW(window1),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window1),10);
    viewport = gtk_viewport_new(NULL,NULL);
    gtk_container_add(GTK_CONTAINER(window1),viewport);
    label = gtk_label_new("此窗口是子窗口\n它的显示和隐藏可由主窗口控制。");
    gtk_container_add(GTK_CONTAINER(viewport),label);
    gtk_widget_show_all(window1);
    return window1;
}
int main ( int argc , char* argv[])
{
    gtk_init(&argc,&argv);
    main_window = create_main_window();
```

```
    gtk_widget_show(main_window);
    gtk_window_move(GTK_WINDOW(main_window),0,0);//移动窗口
    sub_window = create_sub_window();
    gtk_widget_show(sub_window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -W -g twowin.c -o twowin  'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./twowin 即可运行此程序，运行结果如图 5.7 所示。

图 5.7 多窗口功能的实现

实例分析

- 与窗口相关的操作

将窗口图标化用函数 gtk_window_iconfy，相反则用函数 gtk_window_deiconfy。用函数 gtk_widget_hide 可以隐藏控件或窗口。

本示例中还为子窗口的“delete_event”事件加了一个回调函数执行隐藏功能，这样在关闭子窗口时事实上只是将它隐藏了起来。

在以 GTK+2.0 为基础的 GNOME 桌面环境中用窗口管理器(WM，Window Manager)来控制窗口的外观(多数 X 窗口系统中都是这样)。不同的窗口管理器之间程序的运行效果是不一样的。

本章中向读者介绍了 GTK+2.0 编程的一些综合性技巧，对读者进一步开发出功能实用的软件来说有一定推动作用。

第6章 复杂控件

本章重点：

GTK+2.0 新增了两个控件：文本视图控件(TextView)和树视图控件(TreeView)，并重写了一些控件的代码和编程接口，使之性能更加优化，并修复了以前的一些错误。本章就这些控件的用法和功能进行介绍。

本章主要内容：

- 文本视图控件的使用方法
- 树视图控件的使用方法
- 绘图功能的实现
- 用多页显示控件制作一个安装向导界面
- 创建和显示不同形状的光标
- 进度演示控件的使用方法

6.1 文本视图控件

本节示例将介绍如何创建GTK+2.0中的特色控件——文本视图控件以及此控件的一般使用方法和技巧。

实例说明

GTK+2.0 中新增的文本视图控件(GtkTextView)是基于 TK(TCL 和 Python 用到的图形界面开发工具)的集文本显示编辑和格式化输出功能于一身的综合性控件。它支持同一缓冲区的多个视图输出，支持以像素为单位的滚动，通过 Pango 库来支持多国语言文字的输出，具有文本标记功能和类似 HTML 页面中的显示控件功能。

本示例创建了两个文本视图控件和一个文本显示缓冲区，向其中加入多种标记文本、图像和控件，并为控件添加了回调函数，使之单击后能显示对话框。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk
mkdir 6
cd 6
mkdir text
cd text
```

创建本章总目录 6，创建工作目录 text，并进入此目录开始编程。将上一章中 GIF 图像一节中的 4 幅图像复制到当前目录下，再用 GIMP 将此 4 幅图像合并创建一幅 GIF 格式的

动画图像，命名为 pp.gif，保存当前目录下。

(2) 打开编辑器，输入以下代码，以 text.c 为文件名保存到当前目录下。

```
/* 使用textview控件 text.c */
#include <gtk/gtk.h>
//创建信息对话框函数
void    create_message_dialog  (GtkMessageType type, gchar* message)
{
    GtkWidget* dialogx;
    dialogx = gtk_message_dialog_new(NULL,
            GTK_DIALOG_MODAL | GTK_DIALOG_DESTROY_WITH_PARENT,
            type ,GTK_BUTTONS_OK,message);
    gtk_dialog_run(GTK_DIALOG(dialogx));
    gtk_widget_destroy(dialogx);
}
void    on_button_clicked (GtkButton* button, gpointer data)
{
    create_message_dialog(GTK_MESSAGE_INFO,
            "这是由TEXTVIEW中的按钮触发的对话框。");
}
int main ( int argc , char* argv[])
{
    GtkWidget *window, *scwin;
    GtkWidget *vpaned, *view1, *view2;
    GtkWidget *image, *button;
    GdkPixbuf *pixbuf1, *pixbuf2, *pixbuf3, *pixbuf4;
    GtkTextBuffer *buffer;
    GtkTextIter start, end;
    GtkTextChildAnchor *anchor;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"使用textview控件");
    gtk_window_set_default_size(GTK_WINDOW(window),300,200);
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
    vpaned = gtk_vpaned_new();
    gtk_container_add(GTK_CONTAINER(window),vpaned);
 //创建滚动窗口，并设定滚动条为自动显示
    scwin = gtk_scrolled_window_new(NULL,NULL);
    gtk_scrolled_window_set_policy(GTK_SCROLLED_WINDOW(scwin),
                GTK_POLICY_AUTOMATIC,GTK_POLICY_AUTOMATIC);
    view1 = gtk_text_view_new();
    gtk_paned_add1(GTK_PANED(vpaned),scwin);
    gtk_container_add(GTK_CONTAINER(scwin),view1);
    //gtk_text_view_set_left_margin (GTK_TEXT_VIEW (view1), 30);

    pixbuf1 = gdk_pixbuf_new_from_file("p1.bmp",NULL);//创建图像资源
    pixbuf2 = gdk_pixbuf_new_from_file("p2.bmp",NULL);
    pixbuf3 = gdk_pixbuf_new_from_file("p3.bmp",NULL);
```

```
    pixbuf4 = gdk_pixbuf_new_from_file("p4.bmp",NULL);
    image = gtk_image_new_from_file("pp.gif"); //创建图像控件显示动画
    buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(view1));
    //取得文本显示的缓冲区
    gtk_text_buffer_create_tag(buffer, "blue_foreground",
"foreground", "blue", NULL);//创建前景标记
    gtk_text_buffer_create_tag(buffer, "yellow_background",
"background","yellow",NULL);//创建背景标记
    gtk_text_buffer_create_tag(buffer, "simhei", "family", "Simhei",
NULL);
    gtk_text_buffer_create_tag(buffer, "sans", "family", "Sans",NULL);
//以上两行创建字体标记
    gtk_text_buffer_create_tag (buffer, "wide_margins",
                "left_margin", 50, "right_margin", 50,NULL);//边距标
记
    gtk_text_buffer_create_tag (buffer, "heading",
                "justification",GTK_JUSTIFY_LEFT,NULL);//居左
    gtk_text_buffer_create_tag (buffer, "no_wrap",
                "wrap_mode", GTK_WRAP_NONE, NULL);//不换行
    gtk_text_buffer_create_tag (buffer, "word_wrap",
                "wrap_mode", GTK_WRAP_WORD, NULL);//以词为单位换行
    gtk_text_buffer_create_tag(buffer, "center",
                "justification", GTK_JUSTIFY_CENTER, NULL);//居中
    gtk_text_buffer_create_tag(buffer, "right_justify",
                "justification", GTK_JUSTIFY_RIGHT, NULL);//居右
    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_insert(buffer,&end,"这是一段正常的文字。",-1);
//从此处开始带标记插入文字
    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_insert_with_tags_by_name(buffer,&end,"这是一段设了
标记的文字。\n",-1,"blue_foreground",NULL);
    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_insert_with_tags_by_name(buffer,&end,"这是一段背景
是黄色的文字。", -1, "yellow_background",NULL);
    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_insert_with_tags_by_name(buffer,&end,"而这段是黑体
字。\n",-1,"simhei",NULL);
    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_insert_with_tags_by_name(buffer,&end,"这一段文字是
不换行的，所以它会使滚动条变长，不过这不影响程序的运行。",-1,"no_wrap",NULL);
    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_insert_with_tags_by_name(buffer,&end,"\n我们还可以
向其中插入一大段文字，如下面这首诗：\n蓬头稚子学垂纶，侧坐莓苔草映身。路人借问遥招
手，怕得鱼惊不应人。",-1,"wide_margins","word_wrap","center",NULL);

    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_insert_with_tags_by_name(buffer,&end,"\n这段文字是
居左的：\n",-1,"wide_margins","heading","word_wrap",NULL);
    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_insert_with_tags_by_name(buffer,&end,"这段文字是居
中的：\n",-1,"wide_margins","center","word_wrap",NULL);
```

```
    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_insert_with_tags_by_name(buffer,&end,"这段文字则是
居右的。\n",-1,"wide_margins","right_justify","word_wrap",NULL);
    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_insert(buffer,&end,"这是四幅图像，由GDKPIXBUF资源创建
的", -1);
    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_insert_pixbuf(buffer,&end,pixbuf1);//此处插入图像
资源
    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_insert_pixbuf(buffer,&end,pixbuf2);
    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_insert_pixbuf(buffer,&end,pixbuf3);
    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_insert_pixbuf(buffer,&end,pixbuf4);
    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_insert_with_tags_by_name(buffer,&end,"\n这段文字则
用了三种标记。
\n",-1,"blue_foreground","yellow_background","simhei",NULL);
    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_insert(buffer,&end,"这是一幅动画，它是以子控件形式出现
的",-1);
    anchor = gtk_text_buffer_create_child_anchor(buffer,&end);
//创建子控件的位置标记
    gtk_text_view_add_child_at_anchor(GTK_TEXT_VIEW(view1),image,anc
hor);
//向文本视图控件中添加控件
    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_insert(buffer,&end,"\n其它控件也一样加入，如这个按钮
",-1);
    anchor = gtk_text_buffer_create_child_anchor(buffer,&end);
    button = gtk_button_new_with_label("点击显示对话框");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_button_clicked),NULL);
    gtk_text_view_add_child_at_anchor(GTK_TEXT_VIEW(view1),button,an
chor);
    scwin = gtk_scrolled_window_new(NULL,NULL);
    gtk_scrolled_window_set_policy(GTK_SCROLLED_WINDOW(scwin),
                GTK_POLICY_AUTOMATIC,GTK_POLICY_AUTOMATIC);
//创建第二个滚动窗口
    view2 = gtk_text_view_new_with_buffer(buffer);//创建第二个文本视图控
件
    gtk_paned_add2(GTK_PANED(vpaned),scwin);
    gtk_container_add(GTK_CONTAINER(scwin),view2);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o text text.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./text 即可运行此程序，运行结果如图 6.1 所示。

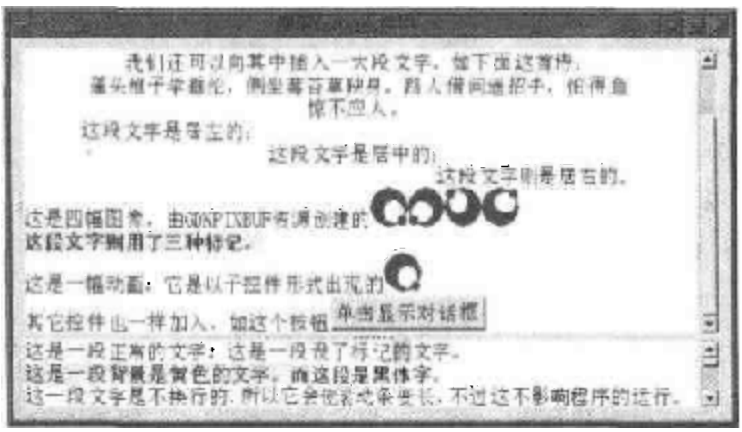

图 6.1　文本视图控件

实例分析

(1) 多行文本编辑控件及相关对象

文本视图控件(GtkTextView)是用以取代原有的文本编辑(GtkText)控件的，它包括一个文本缓冲区(GtkTextBuffer)，用来保存文本控件显示的带有标记性的文字，且一个缓冲区支持多个显示；缓冲区文本的迭代(GtkTextIter)，用来替代缓冲区的某一位置(首部、尾部、某一行等)，以便向其中插入数据或控件；缓冲区的书签(GtkTextMark)，表示缓冲区中的某一具体位置，以便进行滚动操作；缓冲区的文本标记(GtkTextTag)，可以应用到缓冲区中文本的属性标记(如本例中的蓝色前景，居中等)；缓冲区的标记表(GtkTextTagTable)联合多种文本属性标记一起使用。

(2) 定义文本的标记

用函数 gtk_text_view_get_buffer 来取得文本视图控件的缓冲区，然后再用另一个函数 gtk_text_buffer_create_tag 来为此缓冲区来创建文本显示的标记，本例中创建了文字背景、前景、字体、对齐方式、左右边距、换行方式等常用标记，此函数的参数可以有多个，第 1 个为缓冲区指针，第 2 个标记名，第 3 个为标记所属类别，第 4 个为标记的属性名称(可以有多个)，最后一个参数必须是 NULL。

(3) 取得缓冲区的迭代

用函数 gtk_text_buffer_get_end_iter 来取得缓冲区的尾部，它的第二个参数是一个迭代的指针(GtkTextIter*)，一般先声明一个迭代的结构，然后引用它的地址(如例中所示)。相关的函数还有许多，可见 GTK+2.0 的 API 参考手册。

(4) 向缓冲区中添加数据

可以用 gtk_text_buffer_insert 函数直接向缓冲区中插入数据，这样的数据显示为普通的

文字。函数 gtk_text_buffer_insert_with_tag_by_name 向缓冲区中插入带标记的数据，并且用标记的名称来指示标记。这是本示例中用到的两个主要的函数，读者可以细细体会。用函数 gtk_text_buffer_insert_pixbuf 来向缓冲区中插入已创建好的 GdkPixbuf 图像资源。

(5) 缓冲区中的控件

在文本视图控件中用 GtkTextChildAnchor 对象来表示缓冲区中控件的位置(锚点)，用函数 gtk_text_buffer_create_child_anchor 来创建控件位置，返回控件位置的指针；再用函数 gtk_text_view_add_child_at_anchor 来向此位置中添加已创建好的控件指针，控件就会出现在文本视图控件中。同样可以用信号连接宏为添加的控件加回调函数，如本例中的按钮就加了一个显示对话框的回调函数。

本示例主要演示了文本视图控件的格式文本输出功能和插入子控件，这只是此组控件功能的一小部分，要想用好这组功能强大的文本编辑控件，还需要多看 GTK+2.0 的演示程序的源代码和 API 参考手册。

6.2 树视图控件

本节示例将介绍如何创建 GTK+2.0 中的另一特色控件——树视图控件以及此控件的一般使用方法和技巧。

实例说明

GTK+2.0 新增了树视图控件，它是用来显示树型和列表型数据的功能强大的控件，特点是具有将数据显示和数据存贮分开的抽象的显示模型，具有建立定制模型来容纳显示大的数据集的能力，具有通用的显示/表达特色，可以编辑列表中的单元格。

本示例创建了两个数据模型，一个列表显示模型，一个树型显示模型，并分别加入数据显示出来，其中列表显示单元格可以编辑，树型显示可以展开。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/6
mkdir tree
cd tree
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 tree.c 为文件名保存到当前目录下。

```
/* 使用treeview控件  tree.c */
#include <gtk/gtk.h>
enum    {
    ID_COLUMN,  //标号 -- 数字类型
    TOGGLE_COLUMN,  //按钮 -- 按钮类型
    TEXT_COLUMN,    //文字 -- 文本类型
    N_COLUMN        //最后一项
};//定义枚举表示列项
```

```
typedef struct _listitem ListItem;
struct _listitem{
    gint id;
    gboolean toggle;
    const gchar* text;
};//定义每一行的数据结构
ListItem t[5] = {{1,TRUE,"小明"},{2,FALSE,"大宝"},{3,TRUE,"测试用名称
"},{4,FALSE,"显示的文本"},{5,TRUE,"是可编辑的"}
};//定义保存到表格中的数据
GtkListStore*  create_list_model  ( void )        //创建表格模型
{
    GtkListStore *list_store;
    GtkTreeIter iter;
    gint i;
    //创建列表显示模型
    list_store = gtk_list_store_new(N_COLUMN,  G_TYPE_INT,
G_TYPE_BOOLEAN,G_TYPE_STRING);
    for(i=0;i<5;i++)
    {   //向列表显示模型中添加数据
        gtk_list_store_append(list_store,&iter);
        gtk_list_store_set(list_store,&iter,
                ID_COLUMN,t[i].id,
                TOGGLE_COLUMN,t[i].toggle,
                TEXT_COLUMN,t[i].text,-1);
    }
    return list_store;
}
GtkWidget*  create_list (GtkListStore *list_store)
{
    GtkWidget *view;
    GtkTreeModel* model;
    GtkCellRenderer *renderer;
    GtkTreeViewColumn *column;
    //创建列表显示
    model = GTK_TREE_MODEL(list_store);
    view = gtk_tree_view_new_with_model(model);
    renderer = gtk_cell_renderer_text_new();//创建文本型单元格
    column = gtk_tree_view_column_new_with_attributes(
                "数字",renderer,"text",ID_COLUMN,NULL);//创建列
    gtk_tree_view_append_column(GTK_TREE_VIEW(view),column);//添加列
        renderer = gtk_cell_renderer_toggle_new();//创建按钮型单元格
    g_object_set(G_OBJECT(renderer),"activatable",TRUE);//可按
    gtk_cell_renderer_toggle_set_radio(GTK_CELL_RENDERER_TOGGLE(rend
erer),TRUE);//为单选按钮
    column = gtk_tree_view_column_new_with_attributes(
                "按钮",renderer,"active",TOGGLE_COLUMN,NULL);//创建列
    gtk_tree_view_append_column(GTK_TREE_VIEW(view),column);//添加
    renderer = gtk_cell_renderer_text_new();
    g_object_set(G_OBJECT(renderer),"editable",TRUE);//可编辑
    column = gtk_tree_view_column_new_with_attributes(
                "文本",renderer,"text",TEXT_COLUMN,NULL);
```

```
    gtk_tree_view_append_column(GTK_TREE_VIEW(view),column);
    return view;
}
void    show_list   (void)
{   //显示列表
    GtkWidget *window;
    GtkWidget *frame;
    GtkWidget *view;
    GtkListStore *model;
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"destroy",
            G_CALLBACK(gtk_widget_destroy),window);
    gtk_window_set_title(GTK_WINDOW(window),"TREEVIEW -- 使用
LISTSTORE");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
    frame = gtk_frame_new("可编辑的列表");
    gtk_frame_set_label_align(GTK_FRAME(frame),1.0,0);
    gtk_container_add(GTK_CONTAINER(window),frame);
    model = create_list_model();
    view = create_list(model);
    gtk_container_add(GTK_CONTAINER(frame),view);
    gtk_widget_show_all(window);
}
enum {
    XCLASS_COLUMN,  //班级
    LEADER_COLUMN,  //班主任
    STUDENTS_COLUMN,    //学生数
    GIRLS_COLUMN    //女学生
};//定义树型列表项
typedef struct _treeitem TreeItem;
struct _treeitem {
    const char* xclass;
    const char* leader;
    gint students;
    gint girls;
};//定义树型列表显示的数据结构
TreeItem ti[3] = {{"一年级一班","李晓",40,20},{"一年级二班","张宏",44,18},
{"一年级三班","刘丽",42,20}};//定义树型列表显示的数据
TreeItem tj[3] = {{"二年级一班","王可",34,16},{"二年级二班","赵前
",34,18},{"二年级三班","山苛",38,20}};//定义树型列表显示的数据
GtkTreeStore*   create_tree_model   (void)
{
    GtkTreeStore* treestore;
    GtkTreeIter iter,parent;
    gint i;
//创建树型显示模型
    treestore = gtk_tree_store_new(4,G_TYPE_STRING,G_TYPE_STRING,
G_TYPE_INT,G_TYPE_INT);
    gtk_tree_store_append(treestore,&iter,NULL);
    gtk_tree_store_set(treestore,&iter,0,"一年级",-1);
```

```
    gtk_tree_store_append(treestore,&iter,NULL);
    gtk_tree_store_set(treestore,&iter,0," 年级",-1);
    if(gtk_tree_model_get_iter_from_string(GTK_TREE_MODEL(treestore)
,
&parent,"0"))
        for(i=0; i<3; i++)
        {
            gtk_tree_store_append(treestore,&iter,&parent);
            gtk_tree_store_set(treestore,&iter,
                XCLASS_COLUMN,ti[i].xclass,
                LEADER_COLUMN,ti[i].leader,
                STUDENTS_COLUMN,ti[i].students,
                GIRLS_COLUMN,ti[i].girls,-1);
        }
    if(gtk_tree_model_get_iter_from_string(GTK_TREE_MODEL(treestore)
,
&parent,"1"))
        for(i=0; i<3; i++)
        {
            gtk_tree_store_append(treestore,&iter,&parent);
            gtk_tree_store_set(treestore,&iter,
                XCLASS_COLUMN,tj[i].xclass,
                LEADER_COLUMN,tj[i].leader,
                STUDENTS_COLUMN,tj[i].students,
                GIRLS_COLUMN,tj[i].girls,-1);
        }
    return treestore;
}
GtkWidget*  create_tree (GtkTreeStore* treestore)
{
    GtkWidget* view;
    GtkCellRenderer *renderer;
    GtkTreeViewColumn *column;
    view = gtk_tree_view_new_with_model(GTK_TREE_MODEL(treestore));
    renderer = gtk_cell_renderer_text_new();
    column = gtk_tree_view_column_new_with_attributes(
                "班级名称",renderer,"text",XCLASS_COLUMN,NULL);
    gtk_tree_view_append_column(GTK_TREE_VIEW(view),column);
    renderer = gtk_cell_renderer_text_new();
    column = gtk_tree_view_column_new_with_attributes(
                "班主任",renderer,"text",LEADER_COLUMN,NULL);
    gtk_tree_view_append_column(GTK_TREE_VIEW(view),column);
    renderer = gtk_cell_renderer_text_new();
    column = gtk_tree_view_column_new_with_attributes(
                "学生总数",renderer,"text",STUDENTS_COLUMN,NULL);
    gtk_tree_view_append_column(GTK_TREE_VIEW(view),column);
    renderer = gtk_cell_renderer_text_new();
    column = gtk_tree_view_column_new_with_attributes(
                "女学生数",renderer,"text",GIRLS_COLUMN,NULL);
    gtk_tree_view_append_column(GTK_TREE_VIEW(view),column);
    return view;
```

```
}
void    show_tree   (void)
{
    GtkWidget *window;
    GtkWidget *frame;
    GtkWidget *view;
    GtkTreeStore *model;
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"destroy",
            G_CALLBACK(gtk_widget_destroy),window);
    gtk_window_set_title(GTK_WINDOW(window),"TREEVIEW -- 使用
TreeStore");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
    frame = gtk_frame_new("树型列表");
    gtk_container_add(GTK_CONTAINER(window),frame);
    model = create_tree_model();
    view = create_tree(model);
    gtk_container_add(GTK_CONTAINER(frame),view);
    gtk_widget_show_all(window);
}
int main    ( int argc , char* argv[])
{
    GtkWidget *window,*vbox,*button;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"TREEVIEW控件");
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    button = gtk_button_new_with_label("创建列表显示");
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(show_list),NULL);
    button = gtk_button_new_with_label("创建树型显示");
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(show_tree),NULL);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o tree tree.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./tree 即可运行此程序，运行结果如图 6.2 所示。

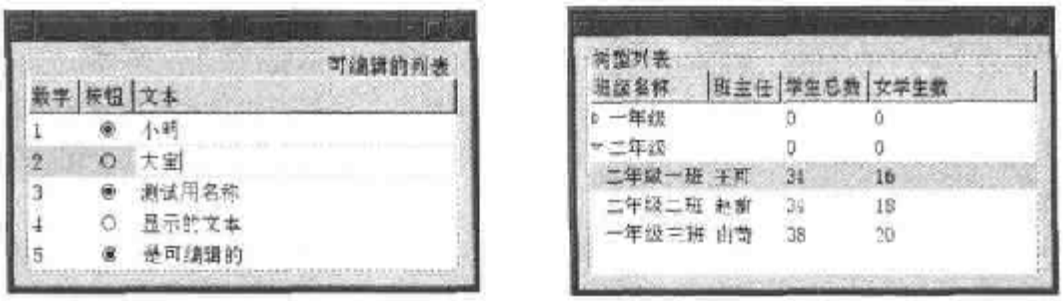

图 6.2　树型控件

实例分析

(1) 树型与列表型控件及相关对象

GTK+2.0 中的树型与列表型控件包括以下几种相关对象：

- 树视图控件(GtkTreeView)，以特定的外观显示指定的数据；
- 树模型(GtkTreeModel)，它的直接形式有两种 GtkListStore 和 GtkTreeStore；
- 树视图的列(GtkTreeViewColumn)，为树视图中的可见的列；
- 树视图的单元格(GtkCellRenderer)，表示树视图中列的单元格，有三种类型：文本型 (GtkCellRendererText)、图像型 (GtkCellRendererPixbuf) 和按钮型 (GtkCellRendererToggle)。还有几个对象本示例中未涉及到，读者可参考 API 手册和演示程序。

(2) 程序的结构

此示例包括两个功能，创建列表型显示和创建树状型显示。结构如图 6.3 所示：

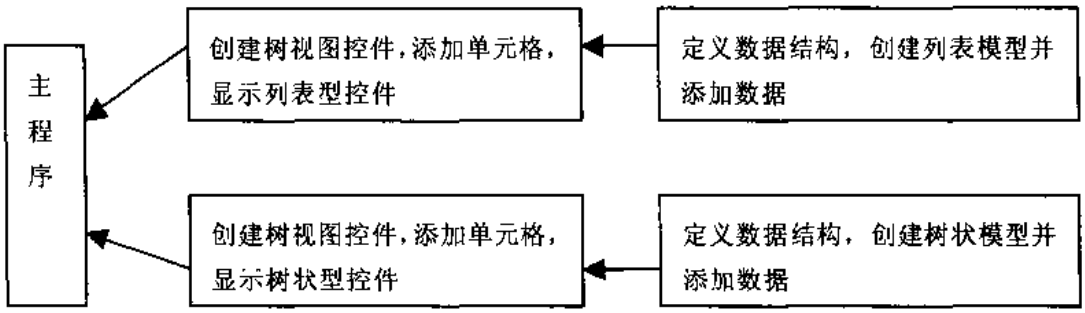

图 6.3　程序结构

(3) 创建显示模型

显示模型分为两种，列表显示模型和树状显示模型。其中用函数 gtk_list_store_new 来创建列表显示模型，用函数 gtk_list_store_append 向模型中添加迭代，用 gtk_list_store_set 设定迭代的各项属性值(即列表中各列的单元格值)，从而达到向列表中添加数据的效果。

树型显示模型用函数 gtk_tree_store_new 来创建，用 gtk_tree_store_append 向模型中添加结点迭代，再用 gtk_tree_model_get_iter_from_string 来取得模型中的结点(从 0 开始)，再用函数 gtk_tree_store_append 添加叶结点，用函数 gtk_tree_store_set 来添加数据。

创建完显示模型后就可以用 gtk_tree_view_new_with_model 来创建列表显示控件了，

但此时的树视图控件并不知道单元格显示的内容，所以还必须为树视图控件创建显示数据的列和单元格。用 gtk_cell_renderer_*_new 系列函数来创建树视图控件的单元格，其中*为 text 表示为文本型单元格，为 toggle 表示按钮型单元格，为 pixbuf 表示图像型单元格；然后再用 gtk_tree_view_column_new_with_attributes 函数来创建列表的显示此类单元格的列，最后将此列用函数 gtk_tree_view_append_column 添加到列表显示中来。至此一个完整的列表显示控件被创建出来了。

GTK+2.0 中新增的树视图控件的功能非常强大，使用过程也相当繁琐，此示例只是它常用功能的一小部分，也是最稳定的一部分，由于它是一个正在开发过程中的控件，所以使用者一定要注意它功能的变化。

6.3 绘图软件的实现

本节将介绍如何在 GTK+2.0 中实现绘图功能。

实例说明

绘图软件是图形界面编程中较重要的角色，它直接体现了图形界面编程的底层绘图部分的功能，它的另一个关键之处是事件的处理。本节示例实现了最基本的绘图功能，可以用鼠标在绘图区控件(GtkDrawingArea)上画出较粗的线条。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/6
mkdir draw
cd draw
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 brush.c 为文件名保存到当前目录下。

```
/* 绘图软件 brush.c */
#include <gtk/gtk.h>
static GtkWidget *window = NULL;
static GdkPixmap *pixmap = NULL;
static gboolean
scribble_configure_event (GtkWidget         *widget,
              GdkEventConfigure *event,
              gpointer       data)
{
    if (pixmap)
        g_object_unref (G_OBJECT (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;
}
static gboolean
scribble_expose_event (GtkWidget      *widget,
              GdkEventExpose *event,
              gpointer        data)
{
    gdk_draw_drawable (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;
}
static void
draw_brush (GtkWidget *widget, gdouble x, gdouble y)
{
    GdkRectangle update_rect;
    update_rect.x = x - 3;
    update_rect.y = y - 3;
    update_rect.width = 6;
    update_rect.height = 6;
    gdk_draw_rectangle (pixmap,
             widget->style->black_gc,
             TRUE,
             update_rect.x, update_rect.y,
             update_rect.width, update_rect.height);
    gdk_window_invalidate_rect (widget->window,
                &update_rect,
                FALSE);
}
static gboolean
scribble_button_press_event (GtkWidget     *widget,
                GdkEventButton *event,
                gpointer        data)
{
    if (pixmap == NULL)
        return FALSE;
    if (event->button == 1)
        draw_brush (widget, event->x, event->y);
    return TRUE;
}
static gboolean
scribble_motion_notify_event (GtkWidget        *widget,
                GdkEventMotion *event,
                gpointer        data)
{
    int x, y;
    GdkModifierType state;
```

```
    if (pixmap == NULL)
        return FALSE;
    gdk_window_get_pointer (event->window, &x, &y, &state);
    if (state & GDK_BUTTON1_MASK)
        draw_brush (widget, x, y);
    return TRUE;
}
static gboolean
checkerboard_expose (GtkWidget      *da,
             GdkEventExpose *event,
             gpointer        data)
{
    gint i, j, xcount, ycount;
    GdkGC *gc1, *gc2;
    GdkColor color;
  #define CHECK_SIZE 10
#define SPACING 2
    gc1 = gdk_gc_new (da->window);
    color.red = 30000;
    color.green = 0;
    color.blue = 30000;
    gdk_gc_set_rgb_fg_color (gc1, &color);
    gc2 = gdk_gc_new (da->window);
    color.red = 65535;
    color.green = 65535;
    color.blue = 65535;
    gdk_gc_set_rgb_fg_color (gc2, &color);
    xcount = 0;
    i = SPACING;
    while (i < da->allocation.width)
    {
        j = SPACING;
        ycount = xcount % 2;
        while (j < da->allocation.height)
        {
            GdkGC *gc;
            if (ycount % 2)
                gc = gc1;
            else
                gc = gc2;
            gdk_draw_rectangle (da->window, gc, TRUE,
                 i, j, CHECK_SIZE, CHECK_SIZE);
            j += CHECK_SIZE + SPACING;
            ++ycount;
        }
        i += CHECK_SIZE + SPACING;
            ++xcount;
    }
    g_object_unref (G_OBJECT (gc1));
    g_object_unref (G_OBJECT (gc2));
    return TRUE;
```

```
}
int main (int argc,char* argv[])
{
    GtkWidget *frame;
    GtkWidget *vbox;
    GtkWidget *da;
    GtkWidget *label;
    gtk_init(&argc,&argv);
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
        gtk_window_set_title (GTK_WINDOW (window), "绘图软件");
    g_signal_connect (G_OBJECT(window), "delete_event",
            G_CALLBACK (gtk_main_quit), NULL);
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);
    vbox = gtk_vbox_new (FALSE, 8);
    gtk_container_set_border_width (GTK_CONTAINER (vbox), 8);
    gtk_container_add (GTK_CONTAINER (window), vbox);
    label = gtk_label_new (NULL);
    gtk_label_set_markup (GTK_LABEL (label),"<u>绘图区域</u>");
    gtk_box_pack_start (GTK_BOX (vbox), label, FALSE, FALSE, 0);

    frame = gtk_frame_new (NULL);
    gtk_frame_set_shadow_type (GTK_FRAME (frame), GTK_SHADOW_IN);
    gtk_box_pack_start (GTK_BOX (vbox), frame, TRUE, TRUE, 0);
        da = gtk_drawing_area_new ();
    gtk_widget_set_size_request (da, 100, 100);
    gtk_container_add (GTK_CONTAINER (frame), da);
    g_signal_connect (da, "expose_event",
            G_CALLBACK (scribble_expose_event), NULL);
    g_signal_connect (da,"configure_event",
            G_CALLBACK (scribble_configure_event), NULL);
    g_signal_connect (da, "motion_notify_event",
            G_CALLBACK (scribble_motion_notify_event), NULL);
    g_signal_connect (da, "button_press_event",
            G_CALLBACK (scribble_button_press_event), NULL);
    gtk_widget_set_events (da, gtk_widget_get_events (da)
                | GDK_LEAVE_NOTIFY_MASK
                | GDK_BUTTON_PRESS_MASK
                | GDK_POINTER_MOTION_MASK
                | GDK_POINTER_MOTION_HINT_MASK);
    gtk_widget_show_all (window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o brush brush.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译：

(5) 编译结束后，执行命令./brush 即可运行此程序，运行结果如图 6.4 所示。

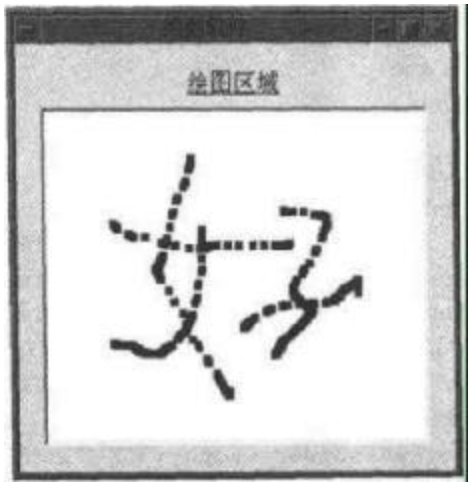

图 6.4 绘图软件

实例分析

绘图区的信号

当前绘图区控件创建完之后，我们应当连接的信号包括：鼠标按键信号(button_press_event)，当前鼠标按下时我们应在当前位置上绘图(本例中是一个小的矩形)；尺寸改变信号(configure_event，expose_event，motion_notify_event)。我们还要用函数 gtk_widget_set_events 来设定绘图区控件的事件的掩码，使这些信号生效。

与绘图相关的函数多数都包含在 GDK(GTK Drawing Kit)程序库中，它是 GTK+2.0 的一部分，主要是与绘图相关的工具，本示例中的所有相关函数和对象都可在 GDK 的 API 参考手册中找到。

此示例是 GTK+2.0 演示程序中绘图部分的一个缩影，也是学习图像编程的一个最简单的示例，想进一步学习 GTK+2.0 中图像编程的朋友一定要研究透。我们不应该忘记的是 GTK+2.0 的最初目标是为创建 GIMP 这一图形处理软件而设计的工具库，所它的绘图和处理图形图像的功能是十分强大的。

6.4 安 装 向 导

本节示例将介绍如何灵活使用笔记本控件创建一个安装向导软件的界面。

实例说明

笔记本控件(GtkNotebook)在第 3 章的关于对话框中用到过，本节示例更进一步应用此控件的其他功能，创建了一个安装向导界面。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/6
mkdir wizard
cd wizard
```

创建工作目录，并进入此目录开始编程。用 GIMP 创建一幅安装标致图像，保存到当前目录下，命名为 wizard.png。

(2) 打开编辑器，输入以下代码，以 wizard.c 为文件名保存到当前目录下。

```
/* 安装向导 wizard.c */
#include <gtk/gtk.h>
static GtkWidget *notebook; //笔记本控件
static GtkWidget *button_prev; //上一步
static GtkWidget *button_next; //下一步
static GtkWidget *button_finish;   //结束
static GtkWidget *button_exit; //退出
GtkWidget*  create_button   (gchar* title,gchar* stockid)
{   //创建带图像的按钮
    GtkWidget *button;
    GtkWidget *image;
    GtkWidget *label;
    GtkWidget *hbox;
    image = gtk_image_new_from_stock(stockid,GTK_ICON_SIZE_MENU);
    label = gtk_label_new(title);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(hbox),image,FALSE,FALSE,3);
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,3);

    button = gtk_button_new();
    gtk_container_add(GTK_CONTAINER(button),hbox);
    return button;
}
void    create_message_dialog   (GtkMessageType type, gchar* message)
{   //创建信息对话框
    GtkWidget* dialogx;
    dialogx = gtk_message_dialog_new(NULL,
            GTK_DIALOG_MODAL | GTK_DIALOG_DESTROY_WITH_PARENT,
            type, GTK_BUTTONS_OK, message);
    gtk_dialog_run(GTK_DIALOG(dialogx));
    gtk_widget_destroy(dialogx);
}
//创建向导的第一个显示页
GtkWidget* create_frame1    (void)
{
    GtkWidget *frame, *vbox, *label;
    frame = gtk_frame_new(NULL);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_frame_set_shadow_type(GTK_FRAME(frame),GTK_SHADOW_ETCHED_OUT
);
    gtk_container_add(GTK_CONTAINER(frame),vbox);
    label = gtk_label_new("欢迎使用本软件！\n本软件是一个.....用途的软件,
\n主要功能是.....\n本软件使用C/C++语言开发，用到.....工具库。\n本软件应用
```

```
在.....平台。\n版权所有：宋国伟 2002年\n联系方式：gwsong52@sohu.com");
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,5);
    return frame;
}
//创建向导的第二个显示页
GtkWidget* create_frame2    (void)
{
    GtkWidget *frame, *vbox, *label;
    frame = gtk_frame_new(NULL);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_frame_set_shadow_type(GTK_FRAME(frame),GTK_SHADOW_ETCHED_OUT
);
    gtk_container_add(GTK_CONTAINER(frame),vbox);
    label = gtk_label_new("本软件遵循GPL2.0协议发行。\n你可以随意分发此软件
而不收取任何费用。\n分发此软件时请保证此软件源代码的完整性。\n此软件可以免费用于商
业用途。");
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,5);
    return frame;
}
//创建向导的第三个显示页
GtkWidget* create_frame3    (void)
{
    GtkWidget *frame, *vbox, *label, *button;
    frame = gtk_frame_new(NULL);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_set_border_width(GTK_CONTAINER(vbox),10);
    gtk_frame_set_shadow_type(GTK_FRAME(frame),GTK_SHADOW_ETCHED_OUT
);
    gtk_container_add(GTK_CONTAINER(frame),vbox);
    label = gtk_label_new("请选择你要安装的软件包：");
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,5);
    button = gtk_check_button_new_with_label("软件源代码包");
    GTK_TOGGLE_BUTTON(button)->active = TRUE;
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    button = gtk_check_button_new_with_label("软件可执行文件包");
    GTK_TOGGLE_BUTTON(button)->active = TRUE;
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    button = gtk_check_button_new_with_label("软件开发工具包");
    GTK_TOGGLE_BUTTON(button)->active = TRUE;
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    button = gtk_check_button_new_with_label("软件说明文档包");
    GTK_TOGGLE_BUTTON(button)->active = TRUE;
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    return frame;
}
//创建向导的第四个显示页
GtkWidget* create_frame4    (void)
{
    GtkWidget *frame, *vbox, *label, *entry;
    frame = gtk_frame_new("软件安装的路径");
    vbox = gtk_vbox_new(FALSE,0);
```

```
    gtk_container_set_border_width(GTK_CONTAINER(vbox),10);
    gtk_frame_set_shadow_type(GTK_FRAME(frame),GTK_SHADOW_ETCHED_OUT
);
    gtk_container_add(GTK_CONTAINER(frame),vbox);
    label = gtk_label_new("可执行文件路径: ");
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,3);
    entry = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(vbox),entry,FALSE,FALSE,3);
    gtk_entry_set_text(GTK_ENTRY(entry),"~/bin");
    label = gtk_label_new("程序库文件路径: ");
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,3);
    entry = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(vbox),entry,FALSE,FALSE,3);
    gtk_entry_set_text(GTK_ENTRY(entry),"~/lib");
    label = gtk_label_new("说明文档路径: ");
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,3);
    entry = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(vbox),entry,FALSE,FALSE,3);
    gtk_entry_set_text(GTK_ENTRY(entry),"~/doc");
    return frame;
}
gboolean run_install    (void)
{
    return TRUE;//此处可以根据用户设置运行相关的安装程序
}
//上一步
void    on_button_prev  (GtkButton* button,gpointer data)
{
    gint id;
    id = gtk_notebook_get_current_page(GTK_NOTEBOOK(notebook));
    if(id == 1)
    {
        gtk_notebook_prev_page(GTK_NOTEBOOK(notebook));
        gtk_widget_set_sensitive(button_prev,FALSE);
    }
    else
    {
        gtk_widget_set_sensitive(button_next,TRUE);
        gtk_notebook_prev_page(GTK_NOTEBOOK(notebook));
    }
}
//下一步
void    on_button_next  (GtkButton* button,gpointer data)
{
    gint id;
    id = gtk_notebook_get_current_page(GTK_NOTEBOOK(notebook));
    if(id == 2)
    {
        gtk_notebook_next_page(GTK_NOTEBOOK(notebook));
        gtk_widget_set_sensitive(button_next,FALSE);
        gtk_widget_set_sensitive(button_finish,TRUE);
```

```
    }
    else
    {
        gtk_widget_set_sensitive(button_prev,TRUE);
        gtk_notebook_next_page(GTK_NOTEBOOK(notebook));
    }
}
//完成
void    on_button_finish    (GtkButton *button,gpointer data)
{
    if(run_install())
        create_message_dialog(GTK_MESSAGE_INFO,"安装成功结束。");
    else
        create_message_dialog(GTK_MESSAGE_ERROR,"安装过程中出错，退出安
装！");
    gtk_main_quit();
}
void    on_button_exit  (GtkButton *button,gpointer data)
{
    create_message_dialog(GTK_MESSAGE_WARNING,"安装并未结束，退出此安装程
序。");
    gtk_main_quit();
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *hbox;
    GtkWidget *vbox;
    GtkWidget *bbox;
    GtkWidget *image;
    GtkWidget *frame;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(on_button_exit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"安装向导");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),hbox);
    image = gtk_image_new_from_file("wizard.png");
    gtk_box_pack_start(GTK_BOX(hbox),image,FALSE,FALSE,5);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(hbox),vbox,FALSE,FALSE,5);
    notebook = gtk_notebook_new();
    gtk_notebook_set_show_tabs(GTK_NOTEBOOK(notebook),FALSE);//不显示
 页头
    gtk_box_pack_start(GTK_BOX(vbox),notebook,TRUE,TRUE,5);
    frame = create_frame1();
    gtk_notebook_append_page(GTK_NOTEBOOK(notebook),frame,NULL);
    frame = create_frame2();
```

```
    gtk_notebook_append_page(GTK_NOTEBOOK(notebook),frame,NULL);
    frame = create_frame3();
    gtk_notebook_append_page(GTK_NOTEBOOK(notebook),frame,NULL);
    frame = create_frame4();
    gtk_notebook_append_page(GTK_NOTEBOOK(notebook),frame,NULL);
    bbox = gtk_hbutton_box_new();
    gtk_button_box_set_layout(GTK_BUTTON_BOX(bbox),GTK_BUTTONBOX_END);
    gtk_box_set_spacing(GTK_BOX(bbox),5);
    gtk_box_pack_start(GTK_BOX(vbox),bbox,FALSE,FALSE,5);
    button_prev = create_button("上一步",GTK_STOCK_GO_BACK);
    gtk_widget_set_sensitive(button_prev,FALSE);
    g_signal_connect(G_OBJECT(button_prev),"clicked",
            G_CALLBACK(on_button_prev),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button_prev,FALSE,FALSE,5);
    button_next = create_button("下一步",GTK_STOCK_GO_FORWARD);
    g_signal_connect(G_OBJECT(button_next),"clicked",
            G_CALLBACK(on_button_next),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button_next,FALSE,FALSE,5);
    button_finish = gtk_button_new_with_label("完成");
    gtk_widget_set_sensitive(button_finish,FALSE);
    g_signal_connect(G_OBJECT(button_finish),"clicked",
            G_CALLBACK(on_button_finish),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button_finish,FALSE,FALSE,5);

    button_exit = gtk_button_new_with_label("取消");
    g_signal_connect(G_OBJECT(button_exit),"clicked",
            G_CALLBACK(on_button_exit),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button_exit,FALSE,FALSE,5);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o wizard wizard.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./wizard 即可运行此程序，运行结果如图 6.5 所示。

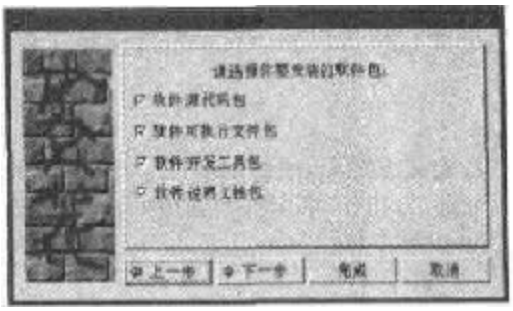

图 6.5　安装向导

实例分析

(1) 多页显示控件的属性

用函数 gtk_notebook_set_show_tabs 来使其不显示页头，这样的话普通用户就看不出这是多页显示控件了。多页显示控件的属性还有显示页头的位置、是否显示弹出菜单、页面容纳不下时是否自动显示左右移动的按钮等。

(2) 创建多个显示页

本示例共创建了 4 个显示页，都以框架的形式出现，用函数 gtk_notebook_append_page 将它们追加进来，如果软件安装步骤超出 4 个，还可以创建更多的显示页添加到程序中来。用函数 gtk_notebook_next_page 和函数 gtk_notebook_prev_page 来设定多页显示控件是显示下一页还是显示上页，用 gtk_notebook_get_current_page 函数取得当前显示页的页号，页码从 0 开始计算。

由于软件安装过程需要用到诸如系统检测、解压缩、复制文件、设置文件权限等很多与 Linux 系统的其他功能，此示例只是创建了安装向导的显示界面，并未实现安装功能。有兴趣的读者可以自行修改编程接口，添加上述功能，真正实现安装。

6.5 不同形状的光标

本节示例将介绍如何灵活使用光标资源和如何创建光标。

实例说明

光标是 X 系统中的一种常用资源，在编程中，有时需要在不同的控件区域显示不同的光标来说明控件的状态，或在不同的运行时刻显示不同的光标表示是否繁忙等。本示例中向读者演示了 GTK+2.0 中定义的所有光标和如何自己做一个光标并将其显示出来。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/6
mkdir cursor
cd cursor
```

创建工作目录，并进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 cursor.c 为文件名保存到当前目录下。

```
/* 光标  cursor.c */
#include <gtk/gtk.h>
GtkWidget *da;  //定义绘图区以改变光标
/* 这些数据是X bitmap格式的，可以用bitmap实用程序创建 */
#define cursor1_width 16
#define cursor1_height 16
static unsigned char cursor1_bits[] = {
  0x80, 0x01, 0x40, 0x02, 0x20, 0x04, 0x10, 0x08, 0x08, 0x10, 0x04, 0x20,
```

```
  0x82, 0x41, 0x41, 0x82, 0x41, 0x82, 0x82, 0x41, 0x04, 0x20, 0x08, 0x10,
  0x10, 0x08, 0x20, 0x04, 0x40, 0x02, 0x80, 0x01};
static unsigned char cursor1mask_bits[] = {
  0x80, 0x01, 0xc0, 0x03, 0x60, 0x06, 0x30, 0x0c, 0x18, 0x18, 0x8c, 0x31,
  0xc6, 0x63, 0x63, 0xc6, 0x63, 0xc6, 0xc6, 0x63, 0x8c, 0x31, 0x18, 0x18,
  0x30, 0x0c, 0x60, 0x06, 0xc0, 0x03, 0x80, 0x01};

void    on_change   (GtkButton* button,gpointer data)
{
    GdkCursor *cursor;  /* 定义光标 */
    GdkPixmap *source, *mask;
    GdkColor fg = { 0, 65535, 0, 0 }; /* 前景：红色 */
    GdkColor bg = { 0, 0, 0, 65535 }; /* 背景：蓝色 */
    //创建GDKPIXMAP图像
    source = gdk_bitmap_create_from_data (NULL, cursor1_bits,
                  cursor1_width, cursor1_height);
    mask = gdk_bitmap_create_from_data (NULL, cursor1mask_bits,
                   cursor1_width, cursor1_height);
//由GDKPIXMAP创建光标
    cursor = gdk_cursor_new_from_pixmap (source, mask, &fg, &bg, 8, 8);
    gdk_pixmap_unref (source);
    gdk_pixmap_unref (mask);
    //设定按钮按钮控件上鼠标的形状
    gdk_window_set_cursor(GTK_WIDGET(button)->window,cursor);
}
//当滚动按钮的值变化时执行
void    on_spin_value_changed   (GtkSpinButton *spin, gpointer data)
{
    GdkCursor *cursor;
    gint id = gtk_spin_button_get_value_as_int(GTK_SPIN_BUTTON(spin));
    cursor = gdk_cursor_new(id);//创建光标
//设定绘图区的鼠标光标的形状
    gdk_window_set_cursor(GTK_WIDGET(da)->window,cursor);
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *button;
    GtkWidget *spin;
    GtkWidget *viewport;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
           G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"多种样式的光标");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    spin = gtk_spin_button_new_with_range(0,152,2);
```

```
    g_signal_connect(G_OBJECT(spin),"value_changed",
            G_CALLBACK(on_spin_value_changed),NULL);
    gtk_box_pack_start(GTK_BOX(vbox),spin,FALSE,FALSE,5);

    da = gtk_drawing_area_new();
    gtk_widget_set_size_request(da,100,100);
    viewport = gtk_viewport_new(NULL,NULL);
    gtk_container_add(GTK_CONTAINER(viewport),da);
    gtk_box_pack_start(GTK_BOX(vbox),viewport,FALSE,FALSE,5);
    button = gtk_button_new_with_label("单击这里改变光标\n这个光标是由自己
的资源创建的");
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(on_change),NULL);
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    button = gtk_button_new_with_label("退出");
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o cursor cursor.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./cursor 即可运行此程序，运行结果如图 6.6 所示。

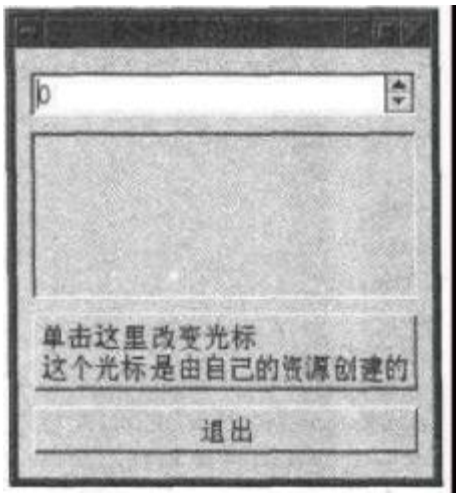

图 6.6 不同形状的光标

实例分析

GTK+2.0 定义了 76 种光标，宏定义值为 0 到 152 间的所有偶数，所以创建滚动按钮时

设定的范围为 0～152，步长值为 2。为使每按一次滚动按钮光标变化一次，我们为滚动按钮的“value_changed”信号加了回调函数，函数中用 gdk_window_set_cursor 函数来为控件设定鼠标移动到其上面时显示的光标。

我们还可以用 bitmap 工具创建一个属于自己的光标，将其保存为 C 语言源程序格式，加入到程序中来，用函数 gdk_cursor_new_from_pixmap 来创建，返回 GdkCursor*型的指针，就可以用它设定控件的光标了。

由于不能将鼠标的形状用抓图软件捕捉下来，所以读者看到的只是程序运行时窗口的外观，只有亲自运行此程序才能看到各种形状的鼠标光标。

6.6　进 度 演 示

本节示例将介绍如何创建进度条控件和滑块控件以及它们的一些使用技巧。

实例说明

进度条控件(GtkProgressBar)和滑块控件(GtkScale)都能表达或指示程序运行中的某些事件的进度。从表面上看进度条可能要主动一些，而滑块则被动一些。本节示例将两种控件的用法集中到一个程序中来，左侧是用两个滑块控制一个按钮的高度和宽度，右侧是通过钟来演示进度条的一般用法。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/6
mkdir scale
cd scale
```

(2) 打开编辑器，输入以下代码，以 scale.c 为文件名保存到当前目录下。

```
/* 滑块与进度条 scale.c */
#include <gtk/gtk.h>
static GtkWidget *scale1;   //滑块一
static GtkWidget *scale2;   //滑块二
static GtkWidget *button;   //按钮
static GtkWidget *label;    //显示进度的标签
static GtkWidget *pbar1;    //进度条
static GtkWidget *check, *check1, *check2; //3个多选按钮
static gint i  ;
static gint j  ;
static gdouble p = 0  ;
gboolean ismode = TRUE;
//当前滑块一的值改变时执行
void    on_scale1_value_changed(GtkScale *scale, gpointer data)
{
    i = (gint)gtk_range_get_value(GTK_RANGE(scale1));
    gtk_widget_set_usize(button,i,j);
```

```
}
//当前滑块二的值改变时执行
void    on_scale2_value_changed(GtkScale *scale, gpointer data)
{
    j = (gint)gtk_range_get_value(GTK_RANGE(scale2));
    gtk_widget_set_usize(button,i,j);
}
//进度开始
void    progress_begin  (void)
{
    gchar buf[256];
    p = p + 0.01;
    if(p > 1)
        p = 0;
    gtk_progress_bar_set_fraction(GTK_PROGRESS_BAR(pbar1),p);
    //gtk_progress_set_value(GTK_PROGRESS(pbar1),p);
    sprintf(buf,"此处显示进度状况：%.2f%%",p);
    gtk_label_set_text(GTK_LABEL(label),buf);
}
void    on_check     (GtkButton *button,gpointer data)
{
    if(GTK_TOGGLE_BUTTON(check)->active)
    gtk_progress_bar_set_bar_style(GTK_PROGRESS_BAR(pbar1),
GTK_PROGRESS_DISCRETE);
    else
    gtk_progress_bar_set_bar_style(GTK_PROGRESS_BAR(pbar1),
GTK_PROGRESS_CONTINUOUS);
}
void    on_check_text   (GtkButton *button,gpointer data)
{
    if(GTK_TOGGLE_BUTTON(check1)->active)
        gtk_progress_set_format_string(GTK_PROGRESS(pbar1),"%v");
    else
        gtk_progress_bar_set_text(GTK_PROGRESS_BAR(pbar1),"进度条");
}
void    on_check_active (GtkButton *button, gpointer data)
{
    if(GTK_TOGGLE_BUTTON(check2)->active)
    {

    gtk_progress_bar_set_activity_step(GTK_PROGRESS_BAR(pbar1),2);

    gtk_progress_bar_set_activity_blocks(GTK_PROGRESS_BAR(pbar1),10)
;
        gtk_progress_set_activity_mode(GTK_PROGRESS(pbar1),TRUE);
    }
    else
        gtk_progress_set_activity_mode(GTK_PROGRESS(pbar1),FALSE);
}
int main ( int argc , char* argv[])
{
```

```
GtkWidget *window;
GtkWidget *vbox, *hbox, *hbox1;
GtkWidget *fixed, *sep;
gtk_init(&argc,&argv);
window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
g_signal_connect(G_OBJECT(window),"delete_event",
        G_CALLBACK(gtk_main_quit),NULL);
gtk_window_set_title(GTK_WINDOW(window),"滑块与进度条");
gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
gtk_container_set_border_width(GTK_CONTAINER(window),10);
hbox = gtk_hbox_new(FALSE,0);
gtk_container_add(GTK_CONTAINER(window),hbox);
vbox = gtk_vbox_new(FALSE,0);
gtk_box_pack_start(GTK_BOX(hbox),vbox,FALSE,FALSE,5);

scale1 = gtk_hscale_new_with_range(1,100,1);
g_signal_connect(G_OBJECT(scale1),"value_changed",
        G_CALLBACK(on_scale1_value_changed),NULL);
gtk_box_pack_start(GTK_BOX(vbox),scale1,FALSE,FALSE,5);

hbox1 = gtk_hbox_new(FALSE,0);
gtk_box_pack_start(GTK_BOX(vbox),hbox1,TRUE,TRUE,5);
scale2 = gtk_vscale_new_with_range(1,100,1);
g_signal_connect(G_OBJECT(scale2),"value_changed",
        G_CALLBACK(on_scale2_value_changed),NULL);
gtk_scale_set_value_pos(GTK_SCALE(scale2),GTK_POS_LEFT);
gtk_box_pack_start(GTK_BOX(hbox1),scale2,FALSE,FALSE,5);
fixed = gtk_fixed_new();
button = gtk_button_new();
gtk_widget_set_usize(button,100,100);
gtk_widget_set_usize(fixed,150,150);
gtk_box_pack_start(GTK_BOX(hbox1),fixed,FALSE,FALSE,5);
gtk_fixed_put(GTK_FIXED(fixed),button,30,30);
sep = gtk_vseparator_new();
gtk_box_pack_start(GTK_BOX(hbox),sep,FALSE,FALSE,5);
vbox = gtk_vbox_new(FALSE,0);
gtk_box_pack_start(GTK_BOX(hbox),vbox,FALSE,FALSE,5);
sep = gtk_label_new("需要用户定义文字显示");
gtk_box_pack_start(GTK_BOX(vbox),sep,FALSE,FALSE,5);
label = gtk_label_new("此处显示进度状况：");
gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,5);
pbar1 = gtk_progress_bar_new();
gtk_progress_bar_set_text(GTK_PROGRESS_BAR(pbar1),"进度条");
gtk_timeout_add(200,(GtkFunction)progress_begin,NULL);
gtk_box_pack_start(GTK_BOX(vbox),pbar1,FALSE,FALSE,5);

check = gtk_check_button_new_with_label("另一种风格");
g_signal_connect(G_OBJECT(check),"clicked",
        G_CALLBACK(on_check),NULL);
gtk_box_pack_start(GTK_BOX(vbox),check,FALSE,FALSE,5);
check1 = gtk_check_button_new_with_label("显示动态文字");
```

```
    g_signal_connect(G_OBJECT(check1),"clicked",
            G_CALLBACK(on_check_text),NULL);
    gtk_box_pack_start(GTK_BOX(vbox),check1,FALSE,FALSE,5);
    check2 = gtk_check_button_new_with_label("活动方式");
    g_signal_connect(G_OBJECT(check2),"clicked",
            G_CALLBACK(on_check_active),NULL);
    gtk_box_pack_start(GTK_BOX(vbox),check2,FALSE,FALSE,5);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o scale scale.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./scale 即可运行此程序，运行结果如图 6.7 所示。

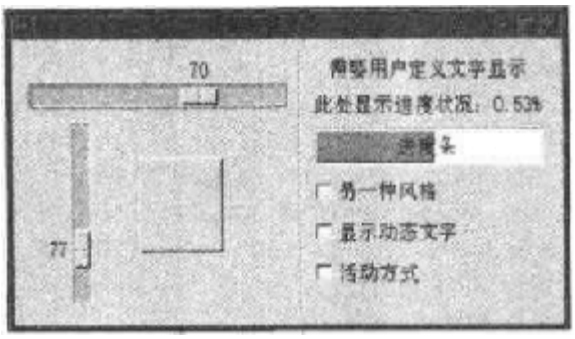

图 6.7 进度演示

实例分析

(1) 控制按钮的大小

此示例的第一部分是演示如何控制按钮的大小，由两个滑块控件，一个控制按钮的宽度，一个控制按钮的高度，这里要注意的是按钮不能直接放到盒状容器中，如果直接放到盒状容器中我们只能看到按钮宽度的改变，而不能看到按钮高度的改变，这是因为盒状容器是按一定的宽度或高度来排放控件的，如何改变这种情况呢？先向盒状容器中放一个自由布局控件，再将按钮放到自由布局控件中去，这样就完成了此功能。

(2) 滑块控件的信号

本示例为滑块控件的“value_changed”信号加了回调函数，使滑块的值改变时执行函数 gtk_widget_set_usize，从而来改变按钮的尺寸。

(3) 进度条控件

在 GTK+2.0 中进度条控件的代码被重写了，优化了许多编程接口。我们这里加了一个时钟，在时钟执行函数中改变进度条的进度，同时也改标签的文字，使之同步变化。

本示例中还改变了进度条控件的属性，使之显示不同的风格，读者仔细看一下示例就会清楚其中的奥妙了。

进度条控件在编程中经常和线程一起使用，来表现处理某些事件的进度，由于线程的使用在 GTK+2.0 中不太容易理解，所以放到本书的最后一章高级应用中讲解。

本章中介绍了 GTK+2.0 中的一些复杂控件的用法和两个经常用到的程序的界面的设计。这些在编程中属于难点问题，希望本章示例能帮助读者朋友攻克这些难点，进一步提高编程水平。

第 7 章　自定义控件与游戏

本章重点：

前面我们研究了GTK+2.0中的大多数控件，很多读者一定想要做一个属于自己的控件，本章将介绍如何创建自己的控件和综合应用各种控件编写简单的游戏程序。

本章主要内容：

- 简单的自定义控件
- 稍复杂的自定义控件
- 小蛇吃豆游戏的实现
- 一个赌筹码的游戏

7.1　组合成的简单文件选择控件

本节示例将介绍如何实现一个由两个常用控件组合成的简单的文件选择控件，并通过此控件的实现来研究如何编写自定义控件。

实例说明

本示例创建了一个简单的文件选择控件，它由一个单行录入控件和一个按钮组成，单击按钮后会弹出文件选择对话框，如果我们选择了一个文件，则单行录入控件中会显示出我们选择的文件名和它的完整的路径。这个控件在使用上和其他控件的使用方法一样。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/
mkdir 7
cd 7
mkdir simple
cd simple
```

创建本章的工作总目录7，创建本节的工作子目录simple，进入此目录开始编程。

(2) 打开编辑器，在当前目录下编辑以下3个文件：

① 控件的头文件simple.h，代码如下：

```
#ifndef __SIMPLE_H__
#define __SIMPLE_H__
//需要包含的头文件
#include <gdk/gdk.h>
#include <gtk/gtkvbox.h>
```

```
//为C++扩展做准备
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
//三个转换与判断宏
#define SIMPLE(obj)          GTK_CHECK_CAST (obj, simple_get_type (),
Simple)
#define SIMPLE_CLASS(klass)  GTK_CHECK_CLASS_CAST (klass,
simple_get_type (), SimpleClass)
#define IS_SIMPLE(obj)       GTK_CHECK_TYPE (obj, simple_get_type
())
//直接与控件相关的结构
typedef struct _Simple       Simple;
typedef struct _SimpleClass  SimpleClass;
//结构一，包含各种成员变量
struct _Simple
{
    GtkHBox hbox;
    GtkWidget *button;
    GtkWidget *entry;
};
//结构二，包含各种成员函数指针
struct _SimpleClass
{
  GtkHBoxClass parent_class;
  void (* simple) (Simple *sss);
};
//与此控件相关的操作函数
GtkType       simple_get_type       (void);
GtkWidget*    simple_new            (void);
//结束定义
#ifdef __cplusplus
}
#endif /* __cplusplus */
#endif /* __SIMPLE_H__ */
```

② 控件的实现 simple.c，代码如下：

```
/* 这是一个简单的文件名选择控件 */
#include <gtk/gtk.h>
#include "simple.h"
enum {
  LAST_SIGNAL   //最后一个信号标志，此示例未用
};
//以下两个函数是自定义的
static GtkWidget *dialog = NULL ;//对话框指针
static void savename (GtkButton *button,Simple *sss);
//以下两个函数是必须的
static void simple_class_init         (SimpleClass *klass);
static void simple_init               (Simple *sss);
static void simple_clicked            (GtkWidget *widget,Simple
```

```
*sss) ;
//取得simple的类型
GType simple_get_type (void)
{
    static GType sss_type = 0;
    if (!sss_type)
    {   //定义类型信息结构
        static const GTypeInfo sss_info =
                    {
            sizeof (SimpleClass),
                        NULL, NULL,
                        (GClassInitFunc) simple_class_init,
                        NULL, NULL, sizeof (Simple),
                        0, (GInstanceInitFunc) simple_init,
                    };
        //注册静态类型
        sss_type = g_type_register_static (GTK_TYPE_HBOX,
 "Simple", &sss_info, 0);
}
    return sss_type;
}
//初始化控件的函数成员指针
static void simple_class_init(SimpleClass *class)
{
    GtkObjectClass *object_class;
    object_class = (GtkObjectClass*) class;
    class->simple = NULL;
}
//初始化控件的成员变量
static void simple_init(Simple *sss)
{
    sss->entry = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(sss),sss->entry,TRUE,TRUE,1);
    sss->button = gtk_button_new_with_label("...");
    gtk_box_pack_start(GTK_BOX(sss),sss->button,FALSE,FALSE,0);
    g_signal_connect(GTK_OBJECT(sss->button),"clicked",
            G_CALLBACK(simple_clicked),sss);
    gtk_widget_show_all(sss);
}
//创建控件的函数
GtkWidget* simple_new(void)
{
    return GTK_WIDGET (g_object_new (simple_get_type (), NULL));
}
//当单击按钮时执行
static void simple_clicked(GtkWidget *widget, Simple *sss)
{
    dialog = gtk_file_selection_new("请选择一个文件: ");
    g_signal_connect (G_OBJECT (GTK_FILE_SELECTION
(dialog)->ok_button),
                    "clicked", G_CALLBACK (savename), sss);
```

```
g_signal_connect_swapped (G_OBJECT (
GTK_FILE_SELECTION (dialog)->ok_button), "clicked",
        G_CALLBACK (gtk_widget_destroy), (gpointer) dialog);
    g_signal_connect(
G_OBJECT (GTK_FILE_SELECTION (dialog)->cancel_button),
            "clicked",G_CALLBACK(gtk_widget_destroy),(gpointer)
dialog);
    gtk_widget_show (dialog);
}
//在单行录入中显示文件名
static void savename(GtkButton *button, Simple *sss)
{
    gtk_entry_set_text(GTK_ENTRY(sss->entry),

    gtk_file_selection_get_filename(GTK_FILE_SELECTION(dialog)))
;
}
```

③ 测试简单控件的程序

```
/* 测试简单控件的程序 simple_test.c */
#include <gtk/gtk.h>
#include "simple.h"
int main    (int argc,char *argv[] )
{
    GtkWidget *window;
    GtkWidget *sss;
    gtk_init (&argc, &argv);
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "简单控件测试程序");
    g_signal_connect (G_OBJECT (window), "destroy",
            G_CALLBACK (gtk_main_quit), NULL);
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);
    sss = simple_new ();
    gtk_container_add (GTK_CONTAINER (window), sss);
    gtk_widget_show (sss);
    gtk_widget_show (window);
    gtk_main ();
    return 0;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
CFLAGS = -Wall
simple_test: simple.o simple_test.o
    $(CC) simple_test.o simple.o -o simple_test 'pkg-config --libs
gtk+-2.0'
simple_test.o: simple_test.c simple.h
    $(CC) -c simple_test.c -o simple_test.o $(CFLAGS) 'pkg-config
gtk+-2.0 --cflags'
simple.o: simple.c simple.h
```

```
    $(CC) -c simple.c -o simple.o $(CFLAGS) 'pkg-config gtk+-2.0
--cflags'
clean:
    rm -f *.o simple_test
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./simple_test 即可运行此程序，运行结果如图 7.1 所示。

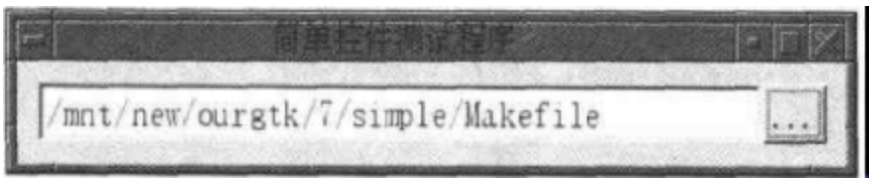

图 7.1 组合成的简单控件

实例分析

(1) 控件的继承

常用的组合控件都继承自盒状容器的某一种(横向或纵向)，定义其他子控件，协调子控件的功能和子控件之间的关系，实现组合控件。继承关系体现在控件结构和控件类结构中定义有父类的相应的实例。如本例中的 Simple 结构中有一个 GtkHBox 结构、SimpleClass 结构中有一个 GtkHBoxClass 结构，这两处最能体现此控件是继承自横向盒状容器的。

(2) 控件的头文件的定义

一般情况下，自定义控件由头文件和代码实现两部分组成，其中头文件中定义了转换宏、与控件相关的结构、与控件相关的函数等内容。

首先是头文件中的 3 个宏，是用来转换和判断控件指针的，它们由 GObject 系统的 GType 类型的检测宏构成。其中 SIMPLE(obj)将对象 obj 转换为 Simple 结构；SIMPLE_CLASS(klass)将 klass 转换为 SimpleClass 结构；IS_SIMPLE(obj)判断 obj 是否为 Simple 结构。

然后是与控件相关的两个结构，其中_Simple 包含其父对象的结构(hbox)和此控件的其他成员变量，这里分别是按钮(button)和单行录入控件(entry)；_SimpleClass 中包含父对象的类结构(GtkHBoxClass)和一个函数指针。

最后是与控件相关的两个函数，都是必须有的。一个是创建此控件的函数 simple_new，它返回此控件的指针，使用上与其他控件一样；另一个是 simple_get_type 函数，用于取得控件的注册类型，这个函数主要是在控件中使用，编程中很少用到。

(3) 控件的代码实现

在控件的代码实现 simple.c 中有两个函数是必须的，就是控件初始化函数 simple_init 和控件类初始化函数 simple_class_init，这两个函数在注册控件类型中用到(关键)。

其中 simple_get_type 中用到了 GTypeInfo 结构和 g_type_ register_static 函数，分别为保存控件的注册信息和注册此控件的静态类型。控件类初始化函数 simple_class_init 我们做得非常简单，只是将控件类强制转换成 GTK+2.0 的对象类结构，并将类结构中的 simple 指针置为 NULL，因为编程中并未用到。函数 simple_init 为控件的初始化函数，我们这里就是创建子控件，为子控件加回调函数，最后显示出来，这与普通的 GTK+2.0 编程几乎一样，

读者完全可以理解。我们这里定义了两个回调函数 simple_clicked(当按钮按下时执行)和 savename(将文件名保存到单行录入控件中)。最后就是创建控件的函数 simple_new，它只有一行，返回用 GTK_WIDGET 宏转换的我们定义的控件的类型。

我们这里需要注意的是虽然看上去它是一个单行录入控件和一个按钮两个控件，但事实上它现在确实是一个控件了。

7.2　八皇后游戏

本节示例将通过一个八皇后游戏控件的实现来介绍如何编程实现稍复杂的控件，和为自定义控件定义信号。

实例说明

八皇后问题在数据结构和算法等教材中都提到过，我们这个示例目的并不是完整解决八皇后问题，而是只要用户解出多种摆法中的一种，即在国际象棋棋盘中放八个皇后，使它们之间互相不能吃到对方。本示例就实现了这一控件，控件中显示 8 行 8 列状态按钮来表示棋盘，用鼠标单击后会显示出皇后棋子，再次单击则清除，放完 8 个皇后棋子，如果放对了显示成功信息，放错了显示出错信息。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/7
mkdir queen
cd queen
```

创建本节的工作目录，进入此目录开始编程。找一幅国际象棋中皇后棋子的图像，规格为 40×40 像素，转换为 PNG 格式，以 q.png 为文件名保存到当前目录下。

(2) 打开编辑器，编辑以下文件保存到当前目录下：

① 八皇后游戏控件的头文件 queen.h，代码如下：

```
#ifndef __QUEEN_H__
#define __QUEEN_H__
#include <gdk/gdk.h>
#include <gtk/gtkvbox.h>
#ifdef __cplusplus
extern "C" {
#endif /* __cplusplus */
//三个转换宏
#define QUEEN(obj)          GTK_CHECK_CAST (obj, queen_get_type (),
Queen)
#define QUEEN_CLASS(klass)  GTK_CHECK_CLASS_CAST (klass,
queen_get_type (), QueenClass)
#define IS_QUEEN(obj)       GTK_CHECK_TYPE (obj, queen_get_type ())
//两个与控件相关的结构
```

```
typecef struct _Queen      Queen;
typedef struct _QueenClass  QueenClass;
struct _Queen
{
  GtkVBox vbox;
  GtkWidget *buttons[8][8];//定义8x8按钮数组来表示棋盘
};
struct _QueenClass
{
  GtkVBoxClass parent_class;
  void (* queen) (Queen *qqq);
};
//三个与控件相关的函数
GtkType       queen_get_type        (void);  //取得此控件的类型
GtkWidget*     queen_new            (void);  //创建八皇后控件
void          queen_clear           (Queen *qqq);     //清除棋盘上皇后
#ifdef __cplusplus
}
#endif /* __cplusplus */
#endif /* __QUEEN_H__ */
```

② 八皇后游戏控件的代码实现 queen.c

```
/* 八皇后问题控件 queen.c */
#include <gtk/gtk.h>
#include "queen.h"
//定义控件的信号的枚举编号
enum {
  QUEEN_SIGNAL, //清除信号
  LAST_SIGNAL   //最后一个信号
};
//定义数组保存摆放在棋盘中的皇后的位置，0表示无皇后，1表示有皇后
static int area[8][8]= {{0,0,0,0,0,0,0,0}, {0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0},  {0,0,0,0,0,0,0,0},  {0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0},  {0,0,0,0,0,0,0,0},  {0,0,0,0,0,0,0,0}};
static int all = 0 ;//皇后总数
static void dialog_show( Queen *qqq );//显示出错对话框并清除棋盘中的皇
后
//以下三个函数为初始化控件成员函数指针，初始化控件成员变量和当状态按钮状态改
变时的回调
static void queen_class_init         (QueenClass *klass);
static void queen_init               (Queen      *qqq);
static void queen_toggle             (GtkWidget *widget, Queen
*qqq);
static gint queen_signals[LAST_SIGNAL] = { 0 };//保存信号值
//取得控件类型
GType queen_get_type(void)
{
    static GType qqq_type = 0 ;
    if (!qqq_type)
    {   //控件的类型信息结构
```

```
        static const GTypeInfo qqq_info =
                {
                    sizeof (QueenClass),
                    NULL, NULL,
                    (GClassInitFunc) queen_class_init,
                    NULL, NULL,
                    sizeof (Queen), 0,
                (GInstanceInitFunc) queen_init,
                };
        qqq_type = g_type_register_static (GTK_TYPE_VBOX, "Queen",
&qqq_info, 0);//注册控件类型
    ;
    return qqq_type;
}
//初始化成员函数指针
static void queen_class_init (QueenClass *class)
{
    GtkObjectClass *object_class;
    object_class = (GtkObjectClass*) class;
    //定义信号,命名为queen
    queen_signals[QUEEN_SIGNAL] = g_signal_new ("queen",
                    G_TYPE_FROM_CLASS (object_class),
                    G_SIGNAL_RUN_FIRST,
                    0, NULL, NULL, g_cclosure_marshal_VOID__VOID,
                    G_TYPE_NONE, 0, NULL);
    class->queen = NULL;
}
//初始化控件变量成员
static void queen_init (Queen *qqq)
{
    GtkWidget *table;
    GdkColor col1;
    GdkColor col2;
    gint i,j;
  //定义颜色
    col1.red = 0; col1.blue = 0; col1.green = 0 ;
    col2.red = 56000;  col2.blue = 56000;  col2.green = 56000;
  //创建格状容器
    table = gtk_table_new (8, 8, TRUE);
    gtk_container_add (GTK_CONTAINER (qqq), table);
    gtk_widget_show (table);
//创建状态按钮并添加到格状容器中
    for (i = 0; i < 8; i++)
        for (j = 0; j < 8; j++)
        {
            qqq->buttons[i][j] = gtk_toggle_button_new ();
            if( (i%2)==0 )//双行
            {
                if( (j%2)!=0 )//单格
                {
                    gtk_widget_modify_bg(
```

```
                        GTK_WIDGET(qqq->buttons[i][j]),
                        GTK_STATE_NORMAL,&col1);    //白色
                }
                else    //双格
                {
                    gtk_widget_modify_bg(
                        GTK_WIDGET(qqq->buttons[i][j]),
                        GTK_STATE_NORMAL,&col2);    //黑色
                }
            }
            else    //单行
            {
                if( (j%2)==0 )  //双格
                {
                    gtk_widget_modify_bg(
                        GTK_WIDGET(qqq->buttons[i][j]),
                        GTK_STATE_NORMAL,&col1);    //白色
                }
                else    //单格
                {
                    gtk_widget_modify_bg(
                        GTK_WIDGET(qqq->buttons[i][j]),
                        GTK_STATE_NORMAL,&col2);    //黑色
                }
            }
            gtk_table_attach_defaults (GTK_TABLE (table),
                        qqq->buttons[i][j], i, i+1, j, j+1);//放按
钮
            g_signal_connect (G_OBJECT (qqq->buttons[i][j]),
"toggled",
                        G_CALLBACK (queen_toggle), qqq);//加回调函
数
            gtk_widget_set_size_request (qqq->buttons[i][j], 44,
44);
            }//设置尺寸
    gtk_widget_show_all(qqq);
}
//创建八皇后控件的函数
GtkWidget* queen_new (void)
{
    return GTK_WIDGET (g_object_new (queen_get_type (), NULL));
}
//清除棋盘上棋子的函数
void    queen_clear (Queen *qqq)
{
    int i,j;
    for (i = 0; i < 8; i++)
        for (j = 0; j < 8; j++)
            {
                g_signal_handlers_block_by_func(
G_OBJECT(qqq->buttons[i][j]),NULL,qqq);
```

```
            gtk_toggle_button_set_active(
GTK_TOGGLE_BUTTON(qqq->buttons[i][j]),FALSE);
                g_signal_handlers_unblock_by_func(
G_OBJECT(qqq->buttons[i][j]),NULL,qqq);
    area[i][j] = 0 ;
    all = 0 ;
            }
}
//当表示棋盘格的状态按钮的状态改变时执行
static void queen_toggle (GtkWidget *widget, Queen *qqq)
{
    int i,j,m,n ;
    gboolean istrue=TRUE;
    GtkWidget*  image ;
    GtkWidget*  child ;
    gpointer iref ;
    GtkWidget* dialog ;
    image = gtk_image_new_from_file("q.png");//创建图象控件
    iref = g_object_ref(image);//引用图像控件指针
    //查找按钮
    for( i = 0 ; i < 8 ; i++ )
        for( j = 0 ; j < 8 ; j++ )
        {
            if(GTK_TOGGLE_BUTTON(qqq->buttons[i][j])->active ==
TRUE)
            {
                 if ( area[i][j] == 0 )
                 {
                    child = gtk_bin_get_child(
GTK_BIN(qqq->buttons[i][j]));
                    if(child != NULL)
                    {   //如子控件不为空，则移除状态按钮的子控件
                        gtk_container_remove(
                        GTK_CONTAINER(qqq->buttons[i][j]),child);
                    }
                    gtk_container_add( //添加图像
GTK_CONTAINER(qqq->buttons[i][j]),iref);
                    gtk_widget_show(iref);//显示图像
                    area[i][j] = 1 ;
                    all++ ;
                 }
            }
            else
            {
                if ( area[i][j]==1 )
                {   //设定按钮的标签文本为空格，即清除图像
                    gtk_button_set_label(
GTK_BUTTON(qqq->buttons[i][j])," ");
                    area[i][j] = 0 ;
                    all-- ;
                }
```

```
            }
        }
    //如果已经摆好8个皇后则执行下面的判断
    if(all == 8)
    {
        for(i = 0 ; i < 8 ; i++ )
            for(j = 0 ; j < 8 ; j++ )
            {
                if(area[i][j]==1)
                {
            //-----------------------------------
            m = i ; n = j ;
            for(;;) //向左查找
            {
                n++;
                if( n > 7 ) break;
                if(area[m][n]==1)
                {
                    istrue = FALSE;break;
                }
            }

            if( istrue == FALSE )
            {
                dialog_show(qqq);
                return ;
            }
            //
            m = i ; n = j ;
            for(;;) //向右查找
            {
                n--;
                if( n < 0 ) break;
                if(area[m][n]==1)
                {
                    istrue = FALSE;break;
                }
            }
            if( istrue == FALSE )
            {
                dialog_show(qqq);
                return ;
            }
            //
            m = i ; n = j ;
            for(;;) //向上查找
            {
                m++;
                if( m > 7 ) break;
                if(area[m][n]==1)
                {
```

```
            istrue = FALSE;break;
        }
    }
    if( istrue == FALSE )
    {
        dialog_show(qqq);
        return ;
    }
    //
    m = i ; n = j ;
    for(;;) //向下查找
    {
        m--;
        if( m < 0 ) break;
        if(area[m][n]==1)
        {
            istrue = FALSE;break;
        }
    }
    if( istrue == FALSE )
    {
        dialog_show(qqq);
        return ;
    }
    //
    m = i ; n = j ;
    for(;;) //向右上查找
    {
        m++; n++;
        if( m > 7 || n > 7 ) break;
        if(area[m][n]==1)
        {
            istrue = FALSE;break;
        }
    }
    if( istrue == FALSE )
    {
        dialog_show(qqq);
        return ;
    }
    //
    m = i ; n = j ;
    for(;;) //向左下查找
    {
        m--; n--;
        if( m < 0 || n < 0 ) break;
        if(area[m][n]==1)
        {
            istrue = FALSE;break;
        }
    }
```

```
if( istrue == FALSE )
{
    dialog_show(qqq);
    return ;
}
//
m = i ; n = j ;
for(;;) //向左上查找
{
    m++; n--;
    if( m > 7 || n < 0 ) break;
    if(area[m][n]==1)
    {
        istrue = FALSE;break;
    }
}
if( istrue == FALSE )
{
    dialog_show(qqq);
    return ;
}
//
m = i ; n = j ;
for(;;) //向右下查找
{
    m--; n++;
    if( m < 0 || n > 7 ) break;
    if(area[m][n]==1)
    {
        istrue = FALSE;break;
    }
}
if( istrue == FALSE )
{
    dialog_show(qqq);
    return ;
}
//-----------------------------------
        }
    }
if( istrue == TRUE )
{   //返回值正确则说明摆法正确
    dialog = gtk_message_dialog_new(NULL,
                GTK_DIALOG_MODAL|
                GTK_DIALOG_DESTROY_WITH_PARENT,
                GTK_MESSAGE_INFO,
                GTK_BUTTONS_OK,
                "恭喜！\n您已经出色的完成了八皇后问题. ");
    gtk_dialog_run(GTK_DIALOG(dialog));
    gtk_widget_destroy(dialog);
    all = 0 ;
```

```
                //如超过八次则自动清除,发射queen信号,则执行queen_clear函数

    g_signal_emit(G_OBJECT(qqq),queen_signals[QUEEN_SIGNAL],0);
        }
    }
}
//显示出错对话框
static void dialog_show(Queen *qqq)
{
    GtkWidget* dialog ;
    dialog = gtk_message_dialog_new(NULL,
                    GTK_DIALOG_MODAL|
                    GTK_DIALOG_DESTROY_WITH_PARENT,
                    GTK_MESSAGE_INFO,
                    GTK_BUTTONS_OK,
                    "非常遗憾！\n您没有完成八皇后问题．继续努力吧！");
    //
    gtk_dialog_run(GTK_DIALOG(dialog));
    gtk_widget_destroy(dialog);
    all = 0 ;
    //如超过八次则自动清除,发射queen信号,则执行queen_clear函数
    g_signal_emit(G_OBJECT(qqq),queen_signals[QUEEN_SIGNAL],0);
}
```

③ 八皇后问题测试程序

```
/* 八皇后问题测试程序  queen_test.c*/
#include <stdlib.h>
#include <gtk/gtk.h>
#include "queen.h"
//八皇后控件的“queen”信号的回调函数
static void win(GtkWidget *qqq, gpointer data)
{
    queen_clear (QUEEN(qqq));//清除棋盘中的皇后
}
int main(int argc, char *argv[])
{
    GtkWidget *window, *vbox, *hbox;
    GtkWidget *label, *image, *statusbar;
    GtkWidget *qqq;
    gtk_init (&argc, &argv);
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "八皇后问题");
    g_signal_connect (G_OBJECT (window), "delete_event",
            G_CALLBACK (gtk_main_quit), NULL);
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,3);
    image =
```

```
    gtk_image_new_from_stock(GTK_STOCK_HELP,GTK_ICON_SIZE_BUTTON);
        gtk_box_pack_start(GTK_BOX(hbox),image,FALSE,FALSE,3);
        label = gtk_label_new("这是用来测试八皇后问题控件的程序");
        gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,3);
        qqq = queen_new ();
        gtk_box_pack_start(GTK_BOX(vbox),qqq,FALSE,FALSE,3);
        //向自定义控件连接信号"queen"
        g_signal_connect (G_OBJECT (qqq), "queen", G_CALLBACK(win),
    NULL);

        statusbar = gtk_statusbar_new();
        gtk_box_pack_start(GTK_BOX(vbox),statusbar,FALSE,FALSE,3);
        gtk_statusbar_push(statusbar,1,"作者：宋国伟  Email:
    gwsong52@sohu.com ");
        gtk_widget_show_all(window);
        gtk_main ();
        return 0;
    }
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
CFLAGS = -Wall
queen_test: queen.o queen_test.o
    $(CC) queen_test.o queen.o -o qqq_test 'pkg-config --libs gtk+-2.0'
queen_test.o: queen_test.c queen.h
    $(CC) -c queen_test.c -o queen_test.o $(CFLAGS) 'pkg-config gtk+-2.0
--cflags'
queen.o: queen.c queen.h
    $(CC) -c queen.c -o queen.o $(CFLAGS) 'pkg-config gtk+-2.0 --cflags'
clean:
    rm -f *.o queen_test
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./qqq_test 即可运行此程序，运行结果如图 7.2 所示。

图 7.2 八皇后游戏

实例分析

(1) 控件的头文件

此控件的头文件与上节基本上是一样的，不同之处在于它是基于纵向盒状容器的，子控件为 8×8 的控件数组(状态按钮类型)，还多了一个相关的 queen_clear 函数。

(2) 控件的代码实现

在此控件的代码实现部分，控件初始化函数和控件类初始化函数同样是必须的，较上节的控件代码多出了信号枚举编号和信号值数组(上节中未用到)，函数 queen_get_type 与上节的 simple_get_type 除了一些名称改为 queen 以外，框架基本上是一样的，在控件类初始化函数 queen_class_init 中多出了信号定义部分，采用 g_signal_new 函数来定义新的信号，我们这里命名为“queen”，详细使用可见 GObject 的 API 参考手册。

创建控件函数 queen_new 与其他控件的创建函数基本相同，函数 queen_clear 用来清除控件棋盘上的所有皇后棋子。

在控件的初始化函数 queen_init 中我们定义了一个格状容器。用以容纳这 8×8 个状态按钮，定义了两种颜色来画棋盘的黑白格，循环创建状态按钮，并为按钮的“toggled”事件加回调函数 queen_toggle，这一回调函数的功能是判断按钮上是否皇后棋子，如果没有则画皇后棋子，如果有则清除此皇后棋子；如果棋子数够 8 个则执行判断，否则不判断。判断摆法是否正确的功能也在此函数中，我采用的方法是先找出第一个棋子，分别向此棋子的 8 个不同方向查找是否有皇后棋子存在，如果存在则摆法失败，显示失败消息，如果不存在则继续查找，依次直到 8 个棋子都查找完毕，如果还未找到则说明摆法成功，显示成功消息。

在显示完消息后发出“queen”信号，即清除信号，我们在测试程序中为创建的控件的“queen”信号加回调函数，回调函数中就用到了 queen_clear 函数。

GTK+2.0 和 GDK 中的所有对象都是基于 GObject 的，这种用 C 语言实现的面向对象的方法非常具有独创性，是初学者要掌握的难点，一定要认真分析才能掌握，另外阅读 GObject 的 API 参考手册也是必要的。

7.3　小 蛇 吃 豆

本节示例将介绍如何灵活运用各种控件和时钟控制、键盘输入等常用编程技巧和方法来实现一个小蛇吃豆的游戏。

实例说明

小蛇吃豆游戏是 PC 上最早出现的最简单的字符界面游戏之一。我们这里虽然用的是图型界面，采用的还是字符模式下的方法，只不过是用图像来模拟字符。此示例在初始化时显示窗口之后，窗口上会显示一个由 5 个小方块图像构成的小蛇和一个不同颜色的豆子，单击开始按钮，小蛇会自动向下移动，到最下面后会从上方出现，如此循环往复。当按下键盘上的 4 个方向键时，小蛇会转向移动，直到蛇的头部碰上豆子，蛇的长度自动增加 5 个图像，程序继续运行，一直到蛇的长度为 30 个小方块图像时则不再出现豆子。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/7
mkdir snake
cd snake
```

创建本节的工作目录，进入此目录开始编程。创建三幅规格为 8×8 像素的图像分别命名为 back.png(背景)、snake.png(蛇)和 bean.png(豆子)保存在当前目录下：

(2) 打开编辑器，输入以下代码，以 snake.c 为文件名保存到当前目录下。

```
/* 小蛇吃豆 snake.c */
#include <gtk/gtk.h>
#include <gdk/gdkkeysyms.h>
#define LEFT        1
#define UP          2
#define RIGHT       3
#define DOWN        4   //定义四个方向
#define MAXLENGTH   30  //蛇的最大长度
//定义点结构，保存蛇和豆子的点的相对位置
struct _point{
    int x;  int y;
};
typedef struct _point point;
static GdkPixbuf *pixbuf = NULL ;  //背景图像
static GdkPixbuf *pix1 = NULL ;    //蛇图像
static GdkPixbuf *pix2 = NULL ;    //豆子图像
static GtkWidget *image[60][40];//高40宽60共2400个图像控件模拟蛇移动的背景
static GtkWidget *label;           //显示小蛇长度的文字
gchar locate[30][2];    //保存蛇的们置
gint slength;       //保存蛇的长度
gint now_forward = DOWN;    //蛇的当前运行方向
gint forward = DOWN;    //蛇的受控运行方向
gint level = 1;         //蛇的长度级别
static gint timer = 0;//时钟标记
gint headx; //蛇头
gint heady;
gint tailx; //蛇尾
gint taily;
gint beanx; //豆的位置
gint beany;

void    init    (gint length);  //初始蛇的位置
void    erase   ();     //抹去尾部
void    draw    ();     //画上头部
void    move    ();     //移动蛇
void    clean   ();     //清除蛇
void    game_run    ();     //运行此游戏
point   rand_point  ();     //产生随机数
void    bean    ();     //画豆
```

```
void    on_begin_clicked        (GtkButton* button,gpointer data);//开
始
void    on_end_clicked      (GtkButton* button,gpointer data);//结束
//随机产生位置
point   rand_point  ()
{
    point p;
    p.x = g_random_int_range(0,60);
    p.y = g_random_int_range(0,40);
    return p;
}
//画豆
void    bean    ()
{
    point p;
    p = rand_point();
    if( p.x == 5 )  p.x++;
    gtk_image_set_from_pixbuf(GTK_IMAGE(image[p.x][p.y]),pix2);
    beanx = p.x;
    beany = p.y;
}
void    init    (gint length)
{
    gint i;
    gint x,y;
    for(i=0; i<length; i++)
    {
        locate[i][0] = 5;
        locate[i][1] = i;
        gtk_image_set_from_pixbuf(GTK_IMAGE(image[5][i]),pix1);
    }
    slength = length;
}
void    clean   ()
{
    gint i,x,y;
    for(i=0; i<slength;i++)
    {
        x = locate[i][0];
        y = locate[i][1];
        gtk_image_set_from_pixbuf(GTK_IMAGE(image[x][y]),pixbuf);
    }
}
void    erase   ()
{
    gtk_image_set_from_pixbuf(GTK_IMAGE(image[tailx][taily]),pixbuf)
;
}
void    draw    ()
{
    gtk_image_set_from_pixbuf(GTK_IMAGE(image[headx][heady]),pix1);
```

```
}
void    move    ()
{
    gint i,len;
    gchar buf[1024];
    len = slength - 1;
    tailx = locate[0][0];
    taily = locate[0][1];
    headx = locate[len][0];
    heady = locate[len][1];
//根据不同的方向改变蛇的位置
    switch(forward)
    {
    case LEFT:  //左
        erase();
        headx--;
        if(headx == -1) headx = 59 ;
        for(i=0; i<len; i++)
        {
            locate[i][0] = locate[i+1][0];
            locate[i][1] = locate[i+1][1];
        }
        locate[len][0] = headx;
        locate[len][1] = heady;
        draw();
        break;
    case UP:    //上
        erase();
        heady--;
        if(heady == -1) heady = 39 ;
        for(i=0; i<len; i++)
        {
            locate[i][0] = locate[i+1][0];
            locate[i][1] = locate[i+1][1];
        }
        locate[len][0] = headx;
        locate[len][1] = heady;
        draw();
        break;
    case RIGHT: //右
        erase();
        headx++;
        if(headx == 60) headx = 0 ;
        for(i=0; i<len; i++)
        {
            locate[i][0] = locate[i+1][0];
            locate[i][1] = locate[i+1][1];
        }
        locate[len][0] = headx;
        locate[len][1] = heady;
        draw();
```

```
            break;
        case DOWN:  //下
            erase();
            heady++;
            if(heady == 40) heady = 0 ;
            for(i=0; i<len; i++)
            {
                locate[i][0] = locate[i+1][0];
                locate[i][1] = locate[i+1][1];
            }
            locate[len][0] = headx;
            locate[len][1] = heady;
            draw();
            break;
        }
        //判断是否吃到豆了
        if( (beanx == headx) && (beany == heady) )
        {
            level++;
            if(level == 7)  return;
            clean();
            sprintf(buf,"小蛇长度: %d  ",level);
            gtk_label_set_text(GTK_LABEL(label),buf);
            now_forward = forward = DOWN;
            init(level*5);
            bean();
        }
}
void    game_run    ()
{
    move();
}
//接收键盘输入
void    key_press   (GtkWidget* widget,GdkEventKey *event,gpointer
data)
{
    switch(event->keyval)
    {
    case GDK_Up :
        if(now_forward != DOWN)
            forward = now_forward = UP;
        break;
    case GDK_Down :
        if(now_forward != UP)
            forward = now_forward = DOWN;
        break;
    case GDK_Left :
        if(now_forward != RIGHT)
            forward = now_forward = LEFT;
        break;
    case GDK_Right :
```

```
            if(now_forward != LEFT)
                forward = now_forward = RIGHT;
            break;
        }
    }

    void    on_begin_clicked    (GtkButton *button, gpointer data)
    {
        timer = gtk_timeout_add(50,(GtkFunction)game_run,NULL);
    }

    void    on_end_clicked      (GtkButton *button, gpointer data)
    {
        gtk_timeout_remove(timer);
    }
    int main ( int argc , char* argv[])
    {
        GtkWidget *window;
        GtkWidget *table;
        GtkWidget *vbox;
        GtkWidget *bbox;
        GtkWidget *button;
        GtkWidget *sep;
        gint i,j;
        gtk_init(&argc,&argv);
        window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
        g_signal_connect(G_OBJECT(window),"delete_event",
                G_CALLBACK(gtk_main_quit),NULL);
        g_signal_connect(G_OBJECT(window),"key_press_event",
                G_CALLBACK(key_press),NULL);
        gtk_window_set_title(GTK_WINDOW(window),"小蛇吃豆");
        gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
        gtk_container_set_border_width(GTK_CONTAINER(window),5);

        vbox = gtk_vbox_new(FALSE,0);
        gtk_container_add(GTK_CONTAINER(window),vbox);
        table = gtk_table_new(40,40,FALSE);
        gtk_box_pack_start(GTK_BOX(vbox),table,FALSE,FALSE,5);
        pixbuf = gdk_pixbuf_new_from_file("back.png",NULL);
        pix1   = gdk_pixbuf_new_from_file("snake.png",NULL);
        pix2   = gdk_pixbuf_new_from_file("bean.png",NULL);
        for(i=0;i<60;i++)   //循环创建60x40个图像
            for(j=0;j<40;j++)
            {
                image[i][j] = gtk_image_new_from_pixbuf(pixbuf);
                gtk_table_attach(GTK_TABLE(table),image[i][j],
                        i,i+1,
                        j,j+1,
                    GTK_FILL|GTK_EXPAND,
                        GTK_FILL|GTK_EXPAND,
                        0,0);
```

```
        }
    sep = gtk_hseparator_new();
    gtk_box_pack_start(GTK_BOX(vbox),sep,FALSE,FALSE,5);
    bbox = gtk_hbutton_box_new();
    gtk_box_pack_start(GTK_BOX(vbox),bbox,FALSE,FALSE,5);
    gtk_button_box_set_layout(GTK_BUTTON_BOX(bbox),GTK_BUTTONBOX_END
);
    label = gtk_label_new("小蛇长度：1  ");
    gtk_box_pack_start(GTK_BOX(bbox),label,TRUE,TRUE,5);
    button = gtk_button_new_with_label("开始");
    gtk_box_pack_start(GTK_BOX(bbox),button,TRUE,TRUE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_begin_clicked),NULL);
    button = gtk_button_new_with_label("停止");
    gtk_box_pack_start(GTK_BOX(bbox),button,TRUE,TRUE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_end_clicked),NULL);

    init(5);//画蛇
    bean(); //画豆
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o snake snake.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./snake 即可运行此程序，运行结果如图 7.3 所示。

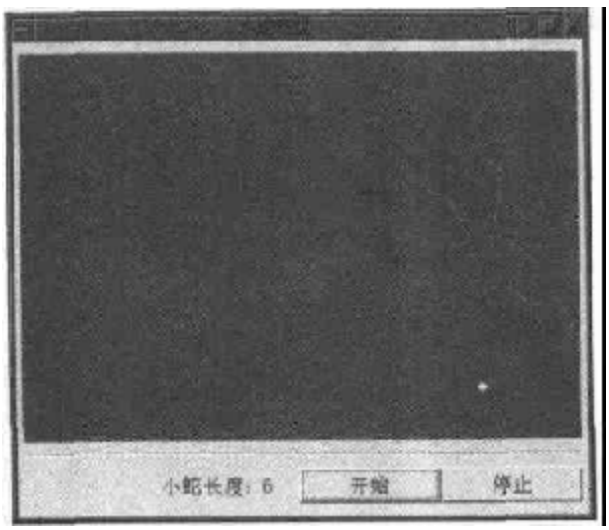

图 7.3　小蛇吃豆

实例分析

(1) 实现过程

首先以数组的形式定义了 60×40 个图像控件，每个图像都是 8×8 像素的深绿色小图像，再用格状控件将它们均匀排列，形成游戏的背景。蛇是由 8×8 像素的图像中显示红色的菱形构成，豆子则是黄色的。我们只要将背景中不同位置的图像改为蛇的图像，一条小蛇就显示出来了；再加上一个时钟，隔一断时间改变一下蛇的位置，就会出现移动效果；在为窗口的“key_press_event”信号加回调函数，即可接收键盘输入(与第 3 章中的坦克游戏相同)。

(2) 位置与移动

我们定义了 4 个整型值的宏来表示蛇移动的方向，将蛇的位置保存到一个二维的数组中，画蛇时循环调用此数组，移动时根据方向改变数组中的值，移动的方法是擦去尾部，画上头部，即将蛇尾部的图像用背景图像覆盖，再将蛇头部的背景图像用蛇的图像画出来。蛇头部移出边界后从另一侧出来，是用判断来做的，即其值等于 60 则重置为 0，其值等于 -1 则重置为 59，这是指横向说的，纵向可依此而定。

此游戏是笔者以前用 Turbo C 在 DOS 下开发的字符界面的小蛇吃豆游戏的一个图像版，将已经开发过的软件重新用 GTK+2.0 来实现，对学习运用 GTK+2.0 来说非常有好处。

7.4 老 虎 机

本节示例将通过一个赌筹码的游戏来介绍如何综合运用各种控件、时钟控制等编程择法进行复杂功能的编程。

实例说明

本示例简单的实现了一个赌筹码的老虎机游戏，采用 9 个图像，纵向 3 个为一组。游戏开始时玩家拥有 500 个筹码，压入筹码后单击确认，这 3 组图像会向不同的方向滚动，单击停止后图像停止滚动，当 3 组都停止后判断是否有 3 个相同的图像出现在同一横行或斜行，如果有则玩家赢得不同倍数的筹码(根据图像而定)，如没有则玩家输掉筹码。压入的筹码可以单击数字按钮来实现，还可以单击 3 个单选按钮来设定图像的滚动速度为慢、中、快三个速度。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/7
mkdir tiger
cd tiger
```

创建本节的工作目录，进入此目录开始编程，到 GTK+2.0 的演示代码中找到以下 6 幅图像：gnome-gmush.png、gnome-gsame.png、apple-red.png、gnome-foot.png、gnome-gimp.png 、

gnome-applets.png 保存到当前目录下。

(2) 打开编辑器，输入以下代码，以 tiger.c 为文件名保存到当前目录下。

```
/* 老虎机  tiger.c */
#include <gtk/gtk.h>
#include <stdlib.h>
#define MAX_MONEY 500
static char* filename[6] = {"gnome-gmush.png","gnome-gsame.png",
"apple-red.png","gnome-foot.png","gnome-gimp.png","gnome-applets.png
"};
gint moneys = MAX_MONEY;        /* 定义筹码总数为500 */
gint outmoney = 0;      /* 定义压入的筹码数为0 */
gint level = 1;     /* 速度级别 */
gint tiger[3][3] = {{0,0,0},{0,0,0},{0,0,0}};/* 初始值均为0 */
static GtkWidget *image[3][3] ;     /* 显示的九幅图像 */
static GtkWidget *entry1 = NULL ;   /* 显示筹码的文字条 */
static GtkWidget *entry2 = NULL ;   /* 显示压入的筹码数的文字条*/
static GtkWidget *b1,*b2,*b3;       /* 3个停止按钮 */
static GtkWidget *ok,*cancel;       /* 确认和取消按钮 */
static GtkWidget *radio1,*radio2,*radio3;  /* 3个定义速度的单选按钮 */
gint id1 = 1;   //定义三个图像组标记
gint id2 = 2;
gint id3 = 3;
static gint timer11 = 0 ;//定义慢速的时钟标记
static gint timer21 = 0 ;
static gint timer31 = 0 ;
static gint timer12 = 0 ;//定义中速的时钟标记
static gint timer22 = 0 ;
static gint timer32 = 0 ;

static gint timer13 = 0 ;//定义快速的时钟标记
static gint timer23 = 0 ;
static gint timer33 = 0 ;
gint count = 0 ;        /* 计数 */
void game_run1  (); //运行第一列
void game_run2  (); //运行第二列
void game_run3  (); //运行第三列
void    re_again    (void);
void    is_best (gint i,gboolean xxx);
void    is_ok   (void);
static void on_okbutton_clicked(GtkButton* button,gpointer data);
static void on_radio_clicked   (GtkWidget* widget,gpointer data);
static void on_stop_clicked(GtkButton* button,gpointer data);
static void on_number_clicked  (GtkButton* button,gpointer data);
static void on_cancel_clicked  (GtkButton* button,gpointer data);
GtkWidget*  create_bbox (); /* 创建带三个停止按钮的盒子 */
GtkWidget*  create_vbox1    (); /* 创建选择速度的按钮盒 */
GtkWidget*  create_vbox2    (); /* 创建显示筹码的按钮盒 */
/* 设定筹码数 */
void    re_again    (void)
{
```

```
    const char m[100];
    g_ascii_dtostr(m,100,moneys);
    gtk_entry_set_text(GTK_ENTRY(entry1),m);
}
/* 判断赢得的筹码数 */
void    is_best (gint i,gboolean xxx)
{
    gint b;/* 倍数 */
    gint mo;/* 筹码数 */
    switch(i)
    {
        case 1 : b = 1 ; break;
    case 2 : b = 2 ; break;
        case 3 : b = 5 ; break;
    case 4 : b = 10 ; break;
        case 5 : b = 20 ; break;
    case 6 : b = 50 ; break;
    }
    if(xxx = TRUE)
    {
        b = 100; //b*10;
    }
    mo = outmoney*b;/* 赢得的筹码数 */
    moneys = moneys+mo;
    re_again();/* 设定筹码数 */
}
/* 判断是否赢筹码 */
void    is_ok   (void)
{
    /* 此处判断是否有横向一行或斜向一行是同一幅图像的情况出现 */
    gint result;
    gboolean xxx = FALSE;
    /*横向第一行*/
    if((tiger[0][0] == tiger[1][0])&&(tiger[0][0] == tiger[2][0]))
    {
        result = tiger[0][0];
        is_best(result,xxx);
    }
    /*横向第二行*/
    if((tiger[0][1] == tiger[1][1])&&(tiger[0][1] == tiger[2][1]))
    {
        result = tiger[0][1];
        is_best(result,xxx);
    }
    /*横向第三行*/
    if((tiger[0][2] == tiger[1][2])&&(tiger[0][2] == tiger[2][2]))
    {
        result = tiger[0][2];
        is_best(result,xxx);
    }
    /*斜向第一行*/
```

```
    if((tiger[0][0] == tiger[1][1])&&(tiger[0][0] == tiger[2][2]))
    {
        result = tiger[0][0];
        xxx = TRUE;
        is_best(result,xxx);
    }
    /*斜向第二行*/
    if((tiger[0][2] == tiger[1][1])&&(tiger[0][2] == tiger[2][0]))
    {
        result = tiger[0][2];
        xxx = FALSE;
        is_best(result,xxx);
    }

    moneys = moneys - outmoney;/* 输掉筹码 */
    re_again();
}
void    game_run1   ()  /* 运行第一列 */
{
    int i;
    i = id1;
    i++;
    if (i == 6) i = 0 ;
    gtk_image_set_from_file(GTK_IMAGE(image[0][0]),filename[i]);
    tiger[0][0] = i+1;
    i++;
    if (i == 6) i = 0;
    gtk_image_set_from_file(GTK_IMAGE(image[0][1]),filename[i]);
    tiger[0][1] = i+1;
    i++;
    if (i == 6) i = 0;
    gtk_image_set_from_file(GTK_IMAGE(image[0][2]),filename[i]);
    tiger[0][2] = i+1;
    id1++;
    if (id1 == 6) id1 = 0;
}
  void  game_run2   ()  /* 运行第二列 */
{
    int i;
    i = id2;
    i--;
    if (i == -1) i = 5 ;
    gtk_image_set_from_file(GTK_IMAGE(image[1][0]),filename[i]);
    tiger[1][0] = i+1;
    i--;
    if (i == -1) i = 5;
    gtk_image_set_from_file(GTK_IMAGE(image[1][1]),filename[i]);
    tiger[1][1] = i+1;
    i--;
    if (i == -1) i = 5;
    gtk_image_set_from_file(GTK_IMAGE(image[1][2]),filename[i]);
```

```
    tiger[1][2] = i+1;
    id2--;
    if (id2 == -1) id2 = 5;
}
void    game_run3   ()  /* 运行第三列 */
{
    int i;
    i = id3;
    i++;
    if (i == 6) i = 0 ;
    gtk_image_set_from_file(GTK_IMAGE(image[2][0]),filename[i]);
    tiger[2][0] = i+1;
    i++;
    if (i == 6) i = 0;
    gtk_image_set_from_file(GTK_IMAGE(image[2][1]),filename[i]);
    tiger[2][1] = i+1;
    i++;
    if (i == 6) i = 0;
    gtk_image_set_from_file(GTK_IMAGE(image[2][2]),filename[i]);
    tiger[2][2] = i+1;
    id3++;
    if (id3 == 6) id3 = 0;
}
/* 当确认按钮按下时执行 */
static void on_okbutton_clicked     (GtkButton* button,gpointer data)
{
    const gchar *m;
    m = gtk_entry_get_text(GTK_ENTRY(entry2));
    outmoney = strtod(m,NULL) ;/* 计算压入的筹码数 */
    if(outmoney == 0)
        return;
    switch(level)   /* 开始运行 */
    {
    case 1 :
        timer11 = gtk_timeout_add(600,(GtkFunction)game_run1,NULL);
        timer12 = gtk_timeout_add(600,(GtkFunction)game_run2,NULL);
        timer13 = gtk_timeout_add(600,(GtkFunction)game_run3,NULL);
        break;
    case 2 :
        timer21 = gtk_timeout_add(300,(GtkFunction)game_run1,NULL);
        timer22 = gtk_timeout_add(300,(GtkFunction)game_run2,NULL);
        timer23 = gtk_timeout_add(300,(GtkFunction)game_run3,NULL);
        break;
    case 3 :
        timer31 = gtk_timeout_add(100,(GtkFunction)game_run1,NULL);
        timer32 = gtk_timeout_add(100,(GtkFunction)game_run2,NULL);
        timer33 = gtk_timeout_add(100,(GtkFunction)game_run3,NULL);
        break;
    }
    gtk_widget_set_sensitive(GTK_WIDGET(button),FALSE);
    gtk_widget_set_sensitive(GTK_WIDGET(b1),TRUE);
```

```
    gtk_widget_set_sensitive(GTK_WIDGET(b2),TRUE);
    gtk_widget_set_sensitive(GTK_WIDGET(b3),TRUE);
    count = 0;
}
/* 当单选按钮按下时执行 */
static void on_radio_clicked    (GtkWidget* widget,gpointer data)
{
    level = GPOINTER_TO_INT(data); /* 改变速度 */
}
/* 当前停止按钮按下时执行 */
static void on_stop_clicked     (GtkButton* button,gpointer data)
{
    switch(GPOINTER_TO_INT(data))   /* 停止 */
    {
    case 1 :
        if(level == 1)  {gtk_timeout_remove(timer11);timer11 = 0;}
        if(level == 2)  {gtk_timeout_remove(timer21);timer21 = 0;}
        if(level == 3)  {gtk_timeout_remove(timer31);timer31 = 0;}
        break;
    case 2 :
        if(level == 1)  {gtk_timeout_remove(timer12);timer12 = 0;}
        if(level == 2)  {gtk_timeout_remove(timer22);timer22 = 0;}
        if(level == 3)  {gtk_timeout_remove(timer32);timer32 = 0;}
        break;
    case 3 :
        if(level == 1)  {gtk_timeout_remove(timer13);timer13 = 0;}
        if(level == 2)  {gtk_timeout_remove(timer23);timer23 = 0;}
        if(level == 3)  {gtk_timeout_remove(timer33);timer33 = 0;}
        break;
    }
    gtk_widget_set_sensitive(GTK_WIDGET(button),FALSE);
    count++;
    if (count == 3)
    {
        is_ok();
        gtk_widget_set_sensitive(GTK_WIDGET(ok),TRUE);
    }
}
/* 当数字按钮按下执行 */
static void on_number_clicked  (GtkButton* button,gpointer data)
{
    const char *num ;
    num = gtk_button_get_label(GTK_BUTTON(button));
    gtk_entry_set_text(GTK_ENTRY(entry2),num);
}
/* 当取消按钮按下时执行 */
static void on_cancel_clicked  (GtkButton* button,gpointer data)
{
    gtk_entry_set_text(GTK_ENTRY(entry2),"");/* 清除输入的内容 */
}
/* 当帮助按钮按下时执行 */
```

```
static  void on_help_clicked    (GtkButton* button,gpointer data)
{   /* 此处显示帮助内容 */
    GtkWidget* window;
    GtkWidget* viewport;
    GtkWidget* label;
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"关于老虎机游戏");
    gtk_container_set_border_width(GTK_CONTAINER(window),5);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_widget_destroy),window);

    viewport = gtk_viewport_new(NULL,NULL);
    gtk_container_add(GTK_CONTAINER(window),viewport);
    label = gtk_label_new("游戏规则：\n先压入一定数量的筹码，注意筹码数一定要
小于现有的筹码总数\n按确定开始转动，按停止按钮则停止转动\n全部停止后，如果有三个图
像在同一横行或在同一斜行则为赢，否则为输\n作者：宋国伟");
    gtk_container_add(GTK_CONTAINER(viewport),label);
    gtk_widget_show_all(window);
}
GtkWidget*  create_bbox     ()  /* 创建带一个停止按钮的盒子 */
{
    GtkWidget *bbox;
    bbox = gtk_hbox_new(FALSE,0);
    b1 = gtk_button_new_with_label("停止");
    gtk_box_pack_start(GTK_BOX(bbox),b1,FALSE,FALSE,40);
    g_signal_connect(G_OBJECT(b1),"clicked",
            G_CALLBACK(on_stop_clicked),(gpointer)1);
    gtk_widget_set_sensitive(GTK_WIDGET(b1),FALSE);

    b2 = gtk_button_new_with_label("停止");
    gtk_box_pack_start(GTK_BOX(bbox),b2,FALSE,FALSE,40);
    g_signal_connect(G_OBJECT(b2),"clicked",
            G_CALLBACK(on_stop_clicked),(gpointer)2);
    gtk_widget_set_sensitive(GTK_WIDGET(b2),FALSE);

    b3 = gtk_button_new_with_label("停止");
    gtk_box_pack_start(GTK_BOX(bbox),b3,FALSE,FALSE,40);
    g_signal_connect(G_OBJECT(b3),"clicked",
            G_CALLBACK(on_stop_clicked),(gpointer)3);
    gtk_widget_set_sensitive(GTK_WIDGET(b3),FALSE);

    gtk_widget_show_all(bbox);
    return bbox;
}
GtkWidget* create_vbox1     ()      /* 创建选择速度的按钮盒 */
{
    GtkWidget *vbox1, *help ;
    GSList *group;
    vbox1 = gtk_vbox_new(FALSE,0);
    radio1 = gtk_radio_button_new_with_label(NULL," 慢速 ");
```

```
    g_signal_connect(G_OBJECT(radio1),"released",
            G_CALLBACK(on_radio_clicked),(gpointer)1);
    gtk_box_pack_start(GTK_BOX(vbox1),radio1,FALSE,FALSE,5);

    group = gtk_radio_button_get_group(GTK_RADIO_BUTTON(radio1));
    radio2 = gtk_radio_button_new_with_label(group," 中速 ");
    g_signal_connect(G_OBJECT(radio2),"released",
            G_CALLBACK(on_radio_clicked),(gpointer)2);
    gtk_box_pack_start(GTK_BOX(vbox1),radio2,FALSE,FALSE,5);
    group = gtk_radio_button_get_group(GTK_RADIO_BUTTON(radio2));
    radio3 = gtk_radio_button_new_with_label(group," 高速 ");
    g_signal_connect(G_OBJECT(radio3),"released",
            G_CALLBACK(on_radio_clicked),(gpointer)3);
    gtk_box_pack_start(GTK_BOX(vbox1),radio3,FALSE,FALSE,5);

    help = gtk_button_new_from_stock(GTK_STOCK_HELP);
    g_signal_connect(G_OBJECT(help),"clicked",
            G_CALLBACK(on_help_clicked),NULL);
    gtk_box_pack_start(GTK_BOX(vbox1),help,FALSE,FALSE,5);

    gtk_widget_show_all(vbox1);
    return vbox1;
}
GtkWidget*  create_vbox2    ()  /* 创建显示筹码的按钮盒 */
{
    GtkWidget *vbox2;
    GtkWidget *hbox1;
    GtkWidget *hbox2;
    GtkWidget *hbox3;
    GtkWidget *hbox4;
    GtkWidget *label;
    GtkWidget *button;
    GtkWidget *cancel;
    gchar num[32];
    gint x ;

    vbox2 = gtk_vbox_new(FALSE,0);

    hbox1 = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox2),hbox1,FALSE,FALSE,5);

    label = gtk_label_new("筹码总数: ");
    gtk_box_pack_start(GTK_BOX(hbox1),label,FALSE,FALSE,5);

    entry1 = gtk_entry_new();
    gtk_widget_set_direction(entry1,GTK_TEXT_DIR_RTL);
    gtk_entry_set_text(GTK_ENTRY(entry1),"500");
    gtk_box_pack_start(GTK_BOX(hbox1),entry1,FALSE,FALSE,5);
    hbox2 = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox2),hbox2,FALSE,FALSE,5);
```

```
    label = gtk_label_new("压入算码: ");
    gtk_box_pack_start(GTK_BOX(hbox2),label,FALSE,FALSE,5);
    entry2 = gtk_entry_new();
    gtk_widget_set_direction(entry2,GTK_TEXT_DIR_RTL);
    gtk_box_pack_start(GTK_BOX(hbox2),entry2,FALSE,FALSE,5);
    hbox3 = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox2),hbox3,FALSE,FALSE,5);
    for(x = 0; x < 9 ; x+=2)
    {
        sprintf(num,"%d",x);
        button = gtk_button_new_with_label(num);
        g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(on_number_clicked),NULL);
        gtk_box_pack_start(GTK_BOX(hbox3),button,FALSE,FALSE,6);
    }
    ok = gtk_button_new_from_stock(GTK_STOCK_OK);
    g_signal_connect(GTK_OBJECT(ok),"clicked",
                G_CALLBACK(on_okbutton_clicked),NULL);
    gtk_box_pack_start(GTK_BOX(hbox3),ok,FALSE,FALSE,5);

    hbox4 = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox2),hbox4,FALSE,FALSE,5);
    for(x = 1; x < 10 ; x+=2)
    {
        sprintf(num,"%d",x);
        button = gtk_button_new_with_label(num);
        g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(on_number_clicked),NULL);
        gtk_box_pack_start(GTK_BOX(hbox4),button,FALSE,FALSE,6);
    }

    cancel = gtk_button_new_from_stock(GTK_STOCK_CANCEL);
    g_signal_connect(G_OBJECT(cancel),"clicked",
            G_CALLBACK(on_cancel_clicked),NULL);
    gtk_box_pack_start(GTK_BOX(hbox4),cancel,FALSE,FALSE,5);

    gtk_widget_show_all(vbox2);
    return vbox2;
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *vbox1;
    GtkWidget *vbox2;
    GtkWidget *hbox;
    GtkWidget *table;
    GtkWidget *frame;
    GtkWidget *bbox;
    GtkWidget *sep;
```

```
    gint i,j;

    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"快乐老虎机");
    gtk_window_set_default_size(GTK_WINDOW(window),350,400);
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    table = gtk_table_new(3,4,FALSE);
    gtk_box_pack_start(GTK_BOX(vbox),table,TRUE,TRUE,5);
    for(i=0;i<3;i++)
        for(j=0;j<3;j++)        /* 依次显示图像 */
        {
            image[i][j] = gtk_image_new_from_file(filename[j]);
            gtk_table_attach(GTK_TABLE(table),
                    image[i][j],
                    i,i+1,
                    j,j+1,
                    GTK_FILL|GTK_EXPAND,
                    GTK_FILL|GTK_EXPAND,
                    5,5);
        }
        bbox = create_bbox();
    gtk_box_pack_start(GTK_BOX(vbox),bbox,FALSE,FALSE,5);
    sep = gtk_hseparator_new();
    gtk_box_pack_start(GTK_BOX(vbox),sep,FALSE,FALSE,5);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    frame = gtk_frame_new("速度选择");
    gtk_box_pack_start(GTK_BOX(hbox),frame,FALSE,FALSE,5);

    vbox1 = create_vbox1();
    gtk_container_add(GTK_CONTAINER(frame),vbox1);
    vbox2 = create_vbox2();
    gtk_box_pack_start(GTK_BOX(hbox),vbox2,FALSE,FALSE,5);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -W -g tiger.c -o tiger  'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./tiger 即可运行此程序，运行结果如图 7.4 所示。

图 7.4 老虎机

实例分析

(1) 布局

此程序的布局看上去较复杂，实际上可以分为几个大块，定义几个函数后分别创建，在主程序中将创建结果摆放到格状容器中，最后全部显示出来。首先是创建 9 个图像控件和一个格状容器，再将这 9 个图像排列到格状容器中去，然后是创建有 3 个停止按钮的按钮盒，第 3 步是创建显示 3 个速度单选按钮的框架，第 4 步是创建显示筹码总数和压入筹码的标签、单行录入控件和数字等按钮。

(2) 图像的滚动

图像的滚动是用计时器来实现的，这一点和第 5 章中的 GIF 图像有些类似，但与 GIF 图像不同的是为每列的 3 个不同的速度都定义了时钟标记，根据不同的速度设定不同的时钟的秒数，同样在停止时根据不同的速度移除不同的时钟标记。

(3) 输与赢

当单击停止按钮后，显示的图像的序号保存到数组中，3 列图像都停止滚动后，进行判断，主要是判断是否有 3 个图像在同一横行或斜行。如果存在这种情况，根据图像的编号来为玩家添加不同倍数的筹码。如果不存在这种情况，则从玩家的筹码中减去压入的筹码数。

此游戏的创意来自网上的一个同名小游戏，不过它是在 WINDOWS 下用 VB 做的，并未提供源程序，笔者在玩了一会儿后，突发此想，将其用 GTK+2.0 重新模拟了出来，在 Linux 的 GNOME 桌面上运行，效果还可以。这也说明当你对 GTK+2.0 的基础控件和编程技巧有所掌握后，试着模拟一些你用过的小软件，会大大提高你运用 GTK+2.0 编程的水平。

本章介绍了 GTK+2.0 自定义控件的一般设计方法和两个小游戏的设计。事实上这两个游戏与 Linux 桌面上其他游戏不能相提并论，我们主要是通过这两个小游戏的设计实现，更深入的理解 GTK+2.0 编程中的一些功能和控件的使用方法。

第8章 文件操作

本章重点：

与文件相关的操作在编程中经常用到，本章的4个示例就围绕文件操作编写的，其中都涉及到了文件的读写操作、取得文件相关的信息等。在WINDOWS上编程的读者一定要注意Linux上文件和目录的用法和操作以及与文件相关的标准C函数。

本章主要内容：

- 文字编辑软件
- INI配置文件
- 卡片管理软件
- 图像查看软件

8.1 文字编辑软件的实现

本节示例将介绍如何运用文本视图控件来创建一个简单的文字编辑软件，通过此软件的实现来学习组织多文件编程。

实例说明

本示例以第2章中完整的应用程序界面为基础，使用文本视图控件来实现一个文字编辑软件、包括文件的打开、保存、另存为；文字的剪切、复制、粘贴；关于对话框、文件改动是否保存对话框、以UTF8格式保存文件等功能。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/
mkdir 8
cd 8
mkdir edit
cd edit
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，在当前目录下编辑下列文件：

① 主程序main.c，代码如下：

```
/* edit -- main.c*/
#include <gtk/gtk.h>
#include "callbacks.h"
#include "interface.h"
int main (int argc , gchar* argv[])
```

```
{
    GtkWidget * window ;
    if(fork())
        exit(0);
    setsid(); //甩开终端控制
    gtk_init(&argc,&argv);
    window = create_window();
    gtk_widget_show(window);
    gtk_main();
    return FALSE;
}
```

② 界面代码头文件 interface.h，代码如下：

```
/* ecit -- interface.h*/
#ifndef __INTERFACE_H__
#define __INTERFACE_H__
GtkWidget*  create_window   ( void );//创建主窗口
#endif
```

③ 界面代码文件 interface.c，代码如下：

```
/* edit -- interface.c */
#include <gtk/gtk.h>
#include "callbacks.h"
GtkWidget* text;        //文本编辑区
GtkTextBuffer* buffer;  //文本缓冲区
GtkClipboard* clipboard; //剪切板
GtkWidget*  create_window   (void)       //创建窗口函数
{
    GtkWidget* window;  //主窗口
    GtkWidget* scrolledwin; //滚动窗口
    GtkWidget* box;     //盒状容器
    GtkWidget* statusbar;   //状态栏
    GtkWidget* menubar ;    //菜单条
    GtkWidget* menu;
    GtkWidget* editmenu;
    GtkWidget* helpmenu;
    GtkWidget* rootmenu;
    GtkWidget* menuitem;
    GtkWidget* toolbar ;        //工具条
    GtkAccelGroup* accel_group ;    //快捷键集合

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"未命名 -- 简单的编辑软
件");
    gtk_window_set_default_size(GTK_WINDOW(window),700,400);
    accel_group = gtk_accel_group_new();
    gtk_window_add_accel_group(GTK_WINDOW(window),accel_group);
    box = gtk_vbox_new(FALSE,0);//容器
    gtk_container_add (GTK_CONTAINER (window), box);
    text = gtk_text_view_new();
```

```
    buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(text));
    g_signal_connect(G_OBJECT(buffer),"changed",
            G_CALLBACK(on_text_buffer_changed),NULL);
    clipboard = gtk_clipboard_get(GDK_NONE);//默认的剪切板
    menu = gtk_menu_new();//文件菜单
    menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_NEW,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_file_new_activate),(gpointer)text);//新
建
    menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_OPEN,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_file_open_activate),(gpointer)text);//
打开
    menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_SAVE,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_file_save_activate),(gpointer)text);//
保存
    menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_SAVE_AS,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",

    G_CALLBACK(on_file_saveas_activate),(gpointer)text);//另存为
    menuitem = gtk_separator_menu_item_new();
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);//横线
    menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_QUIT,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(menu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",

    G_CALLBACK(on_window_delete_event),(gpointer)text);//退出
    rootmenu = gtk_menu_item_new_with_mnemonic("文件(_F)");
    gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootmenu),menu);
    menubar = gtk_menu_bar_new();
    gtk_menu_shell_append(GTK_MENU_SHELL(menubar),rootmenu);
    rootmenu = gtk_menu_item_new_with_mnemonic("编辑(_E)");
    editmenu = gtk_menu_new();//编辑菜单
    menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_CUT,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(editmenu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_edit_cut_activate),NULL); //剪切
    menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_COPY,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(editmenu),menuitem);
```

```
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_edit_copy_activate),NULL);//复制
    menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_PASTE,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(editmenu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_edit_paste_activate),NULL);//粘贴
    menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_FIND,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(editmenu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_edit_find_activate),NULL);//查找
    gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootmenu),editmenu);
    gtk_menu_shell_append(GTK_MENU_SHELL(menubar),rootmenu);
    rootmenu = gtk_menu_item_new_with_mnemonic("帮助(_H)");
    helpmenu = gtk_menu_new();
    menuitem = gtk_image_menu_item_new_from_stock
(GTK_STOCK_HELP,accel_group);
    gtk_menu_shell_append(GTK_MENU_SHELL(helpmenu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_help_help_activate),NULL);//帮助
    menuitem = gtk_menu_item_new_with_label("关于... ");
    gtk_menu_shell_append(GTK_MENU_SHELL(helpmenu),menuitem);
    g_signal_connect(G_OBJECT(menuitem),"activate",
            G_CALLBACK(on_help_about_activate),NULL);//关于
    gtk_menu_item_set_submenu(GTK_MENU_ITEM(rootmenu),helpmenu);
    gtk_menu_shell_append(GTK_MENU_SHELL(menubar),rootmenu);
    gtk_box_pack_start(GTK_BOX(box),menubar,0,0,0);

    toolbar = gtk_toolbar_new();//创建工具栏
    gtk_toolbar_insert_stock(GTK_TOOLBAR(toolbar),
                GTK_STOCK_NEW, "建立一个新文件","新建",
                GTK_SIGNAL_FUNC(on_file_new_activate),text,-1);
    gtk_toolbar_insert_stock(GTK_TOOLBAR(toolbar),
                GTK_STOCK_OPEN,"打开文件","打开",
                GTK_SIGNAL_FUNC(on_file_open_activate),text,-1);
    gtk_toolbar_insert_stock(GTK_TOOLBAR(toolbar),
                GTK_STOCK_SAVE,"保存当前文件","保存",
                GTK_SIGNAL_FUNC(on_file_save_activate),text,-1);
    gtk_toolbar_append_space(GTK_TOOLBAR(toolbar));
    gtk_toolbar_insert_stock(GTK_TOOLBAR(toolbar),
               GTK_STOCK_CUT,"剪切","剪切",
               G_CALLBACK(on_edit_cut_activate),text,-1);
    gtk_toolbar_insert_stock(GTK_TOOLBAR(toolbar),
               GTK_STOCK_COPY,"复制","复制",
               G_CALLBACK(on_edit_copy_activate),text,-1);
    gtk_toolbar_insert_stock(GTK_TOOLBAR(toolbar),
               GTK_STOCK_PASTE,"粘贴","粘贴",
               G_CALLBACK(on_edit_paste_activate),text,-1);
    gtk_toolbar_append_space(GTK_TOOLBAR(toolbar));
    gtk_toolbar_insert_stock(GTK_TOOLBAR(toolbar),
```

```
                GTK_STOCK_QUIT,"退出","退出",
                G_CALLBACK(on_window_delete_event),text,-1);
    gtk_toolbar_set_style(GTK_TOOLBAR(toolbar),GTK_TOOLBAR_ICONS
);//图标
    gtk_toolbar_set_icon_size(GTK_TOOLBAR(toolbar),
                GTK_ICON_SIZE_SMALL_TOOLBAR);//工具栏为小图标
    gtk_box_pack_start(GTK_BOX(box),toolbar,0,1,0);
    scrolledwin = gtk_scrolled_window_new(NULL,NULL);//滚动窗口
    gtk_box_pack_start(GTK_BOX(box),scrolledwin,TRUE,TRUE,2);
    gtk_container_add(GTK_CONTAINER(scrolledwin),text);
    gtk_text_view_set_editable(GTK_TEXT_VIEW(text),TRUE);
    statusbar = gtk_statusbar_new();//状态栏
    gtk_box_pack_start(GTK_BOX(box),statusbar,FALSE,FALSE,0);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(on_window_delete_event),(gpointer)text);
    gtk_widget_show_all(window);
    return window;
}
```

④ 回调函数头文件 callbacks.h，代码如下：

```
/* edit -- callbacks.h */
#ifndef __CALLBACKS_H__
#define __CALLBACKS_H__
gboolean    on_window_delete_event  (GtkWidget* widget,
                    GdkEvent *event,gpointer data);
void        on_text_buffer_changed  (gpointer data);
void        on_file_new_activate    (GtkMenuItem* menuitem,
    gpointer data);
void        on_file_open_activate   (GtkMenuItem* menuitem,
    gpointer data);
void        on_file_save_activate   (GtkMenuItem* menuitem,
    gpointer data);
void        on_file_saveas_activate (GtkMenuItem* menuitem,
    gpointer data);
void        on_file_exit_activate   (GtkMenuItem* menuitem,
    gpointer data);
void        on_edit_cut_activate    (GtkMenuItem* menuitem,
    gpointer data);
void        on_edit_copy_activate   (GtkMenuItem* menuitem,
    gpointer data);
void        on_edit_paste_activate  (GtkMenuItem* menuitem,
    gpointer data);
void    on_edit_selectall_activate  (GtkMenuItem* menuitem,
    gpointer data);
void        on_edit_find_activate (GtkMenuItem* menuitem,gpointer
data);
void        on_help_help_activate   (GtkMenuItem* menuitem,
    gpointer data);
void        on_help_about_activate  (GtkMenuItem* menuitem,
    gpointer data);
```

```
#endif
```

⑤ 回调函数文件 callbacks.c，代码如下：

```
/* edit--callbacks.c */
#define UTF8(str)
    g_locale_to_utf8(str,sizeof(str),NULL,NULL,NULL)
#include <stdio.h>
#include <sys/stat.h>   //加入有关文件信息操作的包含头文件
#include <unistd.h>
#include <gtk/gtk.h>
extern GtkWidget* text;     //文本编辑区,外部定义在interface.c中
extern GtkTextBuffer* buffer;  //文本缓冲区
extern GtkClipboard* clipboard; //剪切板
static GtkWidget *yesno_dialog ;        //是/否/取消对话框
static GtkWidget *open_dialog = NULL;  //打开文件对话框
static GtkWidget *save_dialog = NULL;  //保存文件对话框
static gchar *filename = NULL ;     //文件名
static gboolean file_modifyed = FALSE; //文件是否被改动
static gboolean is_new = FALSE;
static void new_file    (void) ;
static void open_file   (gpointer data) ;
static void real_open   (gpointer data) ;
static void store_filename (GtkFileSelection *selector,gpointer
user_data);
static void save_file   (gpointer data);
static void real_save   (gpointer data);
static void set_window_title   (GtkWidget* window);
GtkWidget*  create_dialog   ( void ) ;

void on_yes_clicked (GtkButton *button,gpointer data)
{   //当按钮是按下时执行
    gtk_widget_destroy(yesno_dialog);
    if( filename == NULL )
    {   //文件名为空则显示保存对话框
        save_file((gpointer)text) ;
    }
    else
    {   //有文件名则直接保存
        real_save((gpointer)text) ;
    }
}
void on_no_clicked  (GtkButton *button,gpointer data)
{
    gtk_widget_destroy(yesno_dialog);
    gtk_main_quit();
}
void on_cancel_clicked  (GtkButton *button,gpointer data)
{
    gtk_widget_destroy(yesno_dialog);
}
```

```
GtkWidget* create_dialog  ( void ) //创建对话框
{
    GtkWidget *dialog;
    GtkWidget *vbox;
    GtkWidget *label;
    GtkWidget *image;
    GtkWidget *hbox;
    GtkWidget *bbox;
    GtkWidget *button;
    GtkWidget *sep;
    gchar title[1024];

    dialog = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(dialog),"delete_event",
            G_CALLBACK(gtk_widget_destroy),dialog);
    gtk_window_set_modal(GTK_WINDOW(dialog),TRUE);
    gtk_window_set_title(GTK_WINDOW(dialog),"警告");
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(dialog),vbox);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    if(filename == NULL)
        sprintf(title,"文件 \"%s\"\n已经被改动,是否保存?","未命名");
    else
        sprintf(title,"文件 \"%s\"\n已经被改动,是否保存?",filename);

    label = gtk_label_new(title);
    image = gtk_image_new_from_stock
(GTK_STOCK_DIALOG_WARNING,GTK_ICON_SIZE_DIALOG);
    gtk_box_pack_start(GTK_BOX(hbox),image,FALSE,FALSE,5);
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    sep = gtk_hseparator_new();
    gtk_box_pack_start(GTK_BOX(vbox),sep,FALSE,FALSE,5);
    bbox = gtk_hbutton_box_new();
    gtk_box_pack_start(GTK_BOX(vbox),bbox,FALSE,FALSE,5);
    button = gtk_button_new_from_stock(GTK_STOCK_YES);//是
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(on_yes_clicked),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,5);
    button = gtk_button_new_from_stock(GTK_STOCK_NO);//否
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(on_no_clicked),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,5);
    button = gtk_button_new_from_stock(GTK_STOCK_CANCEL);//取消
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(on_cancel_clicked),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,5);
    gtk_widget_show_all(dialog);
    return dialog;
}
gboolean on_window_delete_event(GtkWidget* widget,
```

```
GdkEvent *event,gpointer data)
{
    if(file_modifyed == TRUE)
    {
        yesno_dialog = create_dialog();
        gtk_widget_show(yesno_dialog);
    }
    else
    {
        gtk_main_quit();
        return FALSE;
    }
}
void on_file_new_activate       (GtkMenuItem* menuitem,gpointer
data)
{
    GtkWidget* window ;
    window = gtk_widget_get_toplevel(GTK_WIDGET(menuitem));
    if(file_modifyed == TRUE)
    {
        yesno_dialog = create_dialog();
        gtk_widget_show(yesno_dialog);
    }
    filename = NULL ;
    set_window_title(window);
    new_file();//新建
}
void on_file_open_activate      (GtkMenuItem* menuitem,gpointer
data)
{
    open_file(data);//打开
}
void on_file_save_activate      (GtkMenuItem* menuitem,gpointer
data)
{
    if ( filename == NULL )
        save_file(data);//另存为
    else
        real_save(data);
}
void on_file_saveas_activate        (GtkMenuItem*
menuitem,gpointer data)
{
    save_file(data);//另存为
}
void on_edit_cut_activate       (GtkMenuItem* menuitem,gpointer
data)
{
    gtk_text_buffer_cut_clipboard(buffer,clipboard,TRUE); //剪切
}
void on_edit_copy_activate      (GtkMenuItem* menuitem,gpointer
```

```
data)
{
    gtk_text_buffer_copy_clipboard(buffer,clipboard); //复制
}
void on_edit_paste_activate    (GtkMenuItem* menuitem,gpointer
data)
{
    gtk_text_buffer_paste_clipboard(buffer,clipboard,NULL,TRUE);
//粘贴
}
void on_edit_selectall_activate    (GtkMenuItem*
menuitem,gpointer data)
{   //全选
}
void on_edit_find_activate     (GtkMenuItem* menuitem,gpointer
data)
{   //查找
}
void on_help_help_activate     (GtkMenuItem* menuitem,gpointer
data)
{   //帮助
}
void on_help_about_activate    (GtkMenuItem* menuitem,gpointer
data)
{
    GtkWidget* window ;
    GtkWidget* dialog ; //关于
    GtkWidget* hbox;
    GtkWidget* image;
    GtkWidget* label ;
    gint result;
    window = gtk_widget_get_toplevel(GTK_WIDGET(menuitem));
    dialog = gtk_dialog_new_with_buttons("关于...",
            GTK_WINDOW(window),
            GTK_DIALOG_DESTROY_WITH_PARENT|GTK_DIALOG_MODAL,
            GTK_STOCK_OK,GTK_RESPONSE_OK,NULL);

    hbox = gtk_hbox_new(FALSE,0);
    image = gtk_image_new_from_file("gnome-gmush.png");
    label = gtk_label_new("\n此软件用于测试. \n作者：宋国伟\n2002年5月
11日\n");
    gtk_container_add(GTK_CONTAINER(GTK_DIALOG(dialog)->vbox),hb
ox);
    gtk_box_pack_start(GTK_BOX(hbox),image,FALSE,FALSE,3);
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,3);
    gtk_widget_show(label);//_all(dialog);
    gtk_widget_show(image);
    gtk_widget_show(hbox);
    result = gtk_dialog_run(GTK_DIALOG(dialog));
    if ( result == GTK_RESPONSE_OK )
        gtk_widget_destroy(dialog);
```

```
}
void on_text_buffer_changed(gpointer data)
{
    file_modifyed = TRUE ;
}
void store_filename (GtkFileSelection *selector,gpointer
user_data)
{
    filename = gtk_file_selection_get_filename
(GTK_FILE_SELECTION (user_data));
}
static void open_file   (gpointer data) /*打开文件*/
{
    open_dialog = gtk_file_selection_new("请选择一个文件: ");
    g_signal_connect(
GTK_OBJECT(GTK_FILE_SELECTION(open_dialog)->ok_button),
"clicked",G_CALLBACK(store_filename),open_dialog);
    g_signal_connect_swapped (
        GTK_OBJECT(GTK_FILE_SELECTION (open_dialog)->ok_button),
        "clicked",G_CALLBACK(real_open),data);
    g_signal_connect_swapped (
        GTK_OBJECT (GTK_FILE_SELECTION (open_dialog)->ok_button),
        "clicked",G_CALLBACK (gtk_widget_destroy),open_dialog); //
退出
    g_signal_connect_swapped (
        GTK_OBJECT (GTK_FILE_SELECTION
(open_dialog)->cancel_button),
        "clicked",G_CALLBACK (gtk_widget_destroy),(gpointer)
open_dialog);
    gtk_widget_show(open_dialog);
}
static void real_open   (gpointer data) //真正打开操作
{
    GtkWidget* window;
    FILE *fp ;
    GtkTextIter iter;
    GtkTextIter start,end;
    glong bytes_read ;
    gchar *buf ;        //源缓冲区
    gchar *dbuf ;       //目标缓冲区
    glong dlong ;       //转换后的长度
    glong length ;      //文件长度
    struct stat pstat ; //文件状态结构,用于取得文件长度

    if ( stat(filename,&pstat)==-1)
    {
        g_print("FILE ERROR HERE .....\n");//取文件信息时出错
        return ;
    }
    length = pstat.st_size ;    //取得文件的长度
    buf = g_malloc(length);     //为源缓冲区分配内存
```

```
    gtk_text_buffer_get_start_iter(buffer,&start);
    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_delete(buffer,&start,&end);//清除所有
    gtk_text_buffer_get_iter_at_line_index(buffer,&iter,0,0); //
取标记
    fp = fopen(filename,"r");  //打开文件
    if ( fp == NULL)
    {
        g_print("ERROR : file cann't open .\n");//打开文件时出错
        g_free(buf);
        return;
    }

    bytes_read = fread(buf,sizeof(gchar),length,fp);  //读文件
    if ( g_utf8_validate(buf,bytes_read,NULL) == FALSE )
    {
        dbuf = g_locale_to_utf8(buf,length,NULL,&dlong,NULL);
        //转换为UTF8格式,转换后的长度存在dlong中,指针为dbuf
        if(bytes_read == length)
        {
            gtk_text_buffer_insert(buffer,&iter,dbuf,dlong);
        }
    }
    else
    {
        if(bytes_read == length)
            gtk_text_buffer_insert(buffer,&iter,buf,length);
    }
    if ( ferror(fp) )
    {
        fclose(fp);
        g_free(buf);
        return;
    }
    fclose(fp);
    g_free(buf);
    window = gtk_widget_get_toplevel(GTK_WIDGET(data));
    set_window_title(window);//设定窗口标题
}
static void save_file   (gpointer data)//保存文件
{
    save_dialog = gtk_file_selection_new("请选择一个文件: ");
    g_signal_connect(
GTK_OBJECT(GTK_FILE_SELECTION(save_dialog)->ok_button),

    "clicked",G_CALLBACK(store_filename),save_dialog);//NULL);
    g_signal_connect_swapped (
        GTK_OBJECT(GTK_FILE_SELECTION (save_dialog)->ok_button),
        "clicked",G_CALLBACK(real_save),data);
    g_signal_connect_swapped (
        GTK_OBJECT (GTK_FILE_SELECTION (save_dialog)->ok_button),
```

```
        "clicked",G_CALLBACK (gtk_widget_destroy),
        (gpointer) save_dialog); //退出
    g_signal_connect_swapped (
        GTK_OBJECT (GTK_FILE_SELECTION
(save_dialog)->cancel_button),
        "clicked",G_CALLBACK (gtk_widget_destroy),
        (gpointer) save_dialog); //退出
    gtk_widget_show(save_dialog);
}
static voic real_save  (gpointer data) //真正保存文件
{
    GtkWidget* window;
    FILE *fp;
    GtkTextBuffer *textbuf;
    gchar *buf;
    gchar *dbuf;
    GtkTextIter start, end ;
    textbuf = gtk_text_view_get_buffer(GTK_TEXT_VIEW(data));
    gtk_text_buffer_get_start_iter(textbuf,&start);
    gtk_text_buffer_get_end_iter(textbuf,&end);
    fp = fopen(filename,"w");
    if ( fp == NULL)
    {
        g_print("ERROR : file cann't open .\n");//打开文件时出错
        g_free(buf);
        return;
    }
    buf =
gtk_text_buffer_get_text(textbuf,&start,&end,TRUE);//FALSE);
    dbuf = g_locale_from_utf8(buf,-1,NULL,NULL,NULL);//转换为本地代
码保存
    fputs(dbuf,fp);//写文件
    fclose(fp);
    window = gtk_widget_get_toplevel(GTK_WIDGET(data));
    set_window_title(window);//设定窗口标题
    file_modifyed = FALSE ; //改动标记
}

static void new_file    (void)  //新建文件
{
    GtkTextIter start, end ;
    gtk_text_buffer_get_start_iter(buffer,&start);
    gtk_text_buffer_get_end_iter(buffer,&end);
    gtk_text_buffer_delete(buffer,&start,&end);//清除所有
}
static void set_window_title        (GtkWidget* window) //设定窗口
标题
{
    gchar title[1024] ;
    gchar *noname = "未命名";
    if( filename != NULL )
```

```
            sprintf(title,"%s -- 简单的编辑软件",filename);
        else
            sprintf(title,"%s -- 简单的编辑软件",noname);
        gtk_window_set_title(GTK_WINDOW(window),title);
    }
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:main.o callbacks.c interface.o
    $(CC) -o gb main.o callbacks.o interface.o 'pkg-config --libs
gtk+-2.0'
main.o:main.c interface.h callbacks.h
    $(CC) -c main.c 'pkg-config --cflags gtk+-2.0'
interface.o:interface.c interface.h callbacks.h
    $(CC) -c interface.c 'pkg-config --cflags gtk+-2.0'
callbacks.o:callbacks.c callbacks.h
    $(CC) -c callbacks.c 'pkg-config --cflags gtk+-2.0'
clean:
    rm -f *.o
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./gb 即可运行此程序，运行结果如图 8.1 所示。

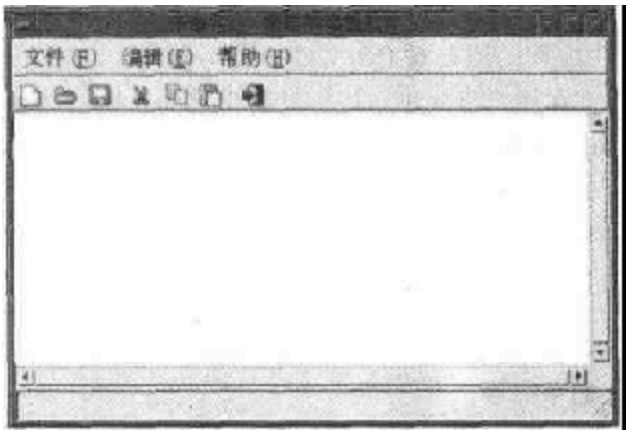

图 8.1　编辑软件的实现

实例分析

(1) 文件的打开与保存

多文件的组织在第 2 章中完整的应用程序界面中出现过，此程序就是在那个示例的基础上开发出来的，主要完善了各种菜单和工具条的功能。其中函数 open_file 和 save_file 主要是显示文件对话框，real_open 和 real_save 则是真正的将读取文件到文本缓冲区中和文本缓冲区中的文字保存到文件中去。而 new_file 函数是创建新文件，即清除文本缓冲区、文件名和设定窗口的标题(用 set_window_title 函数)。文件的打开和保存操作用到了标准 C 程序库中的 fopen 系列函数，其用法相信读者一定早已熟知。

(2) 编辑操作

此示例中的编辑操作主要是用文本缓冲区的复制、粘贴、剪切函数和剪切板对象，还涉及到将本地格式的代码转换为 UTF8 格式(用函数 g_local_to_utf8)和相反的过程(用 g_local_from_utf8 函数)。读者可以在 GTK+2.0 和 GLIB2.0 的 API 参考手册中找到相关的说明。

(3) 文件的改动

此示例中设定了一个文件改动标记，并为文本显示控件的“changed”信号加了回调函数，使文本一改动就设定此标记为 TRUE；当要关闭窗口时判断此标记如为 TRUE 则显示是/否/取消对话框，让用户处理对文件的改动，以确保文件的改动不会丢失。

由于在主程序中使用了甩开终端控制功能，所以当前我们执行完./gb 命令后，终端的提示符马上就显示出来了。相信读者朋友一定会把这一功能运用到自己的编程中去。

8.2 INI 配置文件

本节示例将介绍如何用 C 语言和 GTK+2.0 控件的形式编写 INI 文件对象，并对其进行简单的测试。

实例说明

常在 WINDOWS 中编程的朋友对 INI 文件一定不会陌生，它作为一种配置文件非常有用。而在 Linux 系统中常在用户目录下出现一些类似名为“*.rc”的启动配置文件，这些文件的结构有些地方和 WINDOWS 下的 INI 文件非常相似。本节示例就是用 C 语言和 GTK 控件的形式创建了 INI 对象，实现了 INI 文件的节与项的整型、字符串型和逻辑型数据的写入、读取和保存文件等。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/8
mkdir ini
cd ini
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，编辑以下 3 个文件保存到当前目录下。

① INI 配置文件的头文件 inifile.h，代码如下：

```
/* inifile.h */
#ifndef __INI_FILE__
#define __INI_FILE__
#include <glib.h>
typedef struct  {
    gchar *key;     //键
    gchar *value;   //值
}IniLine;   //定义INI文件的行
```

```
typedef struct {
    gchar *name;    //节
    GList *lines;   //链表
}IniSection;    //定义INI文件的节
typedef struct {
    gchar *filename;    //文件名
    GList *sections;    //节链表
    gboolean changed;   //是否改动
}IniFile;   //定义INI文件对象
IniFile* ini_file_new();    //创建INI文件对象
IniFile* ini_file_open_file(gchar *filename);//创建INI文件对象并打开
文件
gboolean ini_file_write_file(IniFile* ini, gchar* filename);//将
INI 对象保存为文件
void    ini_file_free(IniFile* ini);//释放INI文件对象
//以下为读INI文件的键值的函数
gboolean ini_file_read_string(IniFile *ini,gchar *section,
                gchar *key,gchar ** value);
gboolean ini_file_read_int(IniFile *ini, gchar *section,
                gchar *key, gint *value);
gboolean ini_file_read_boolean(IniFile *ini, gchar *section,
                gchar *key,gboolean *value);
//以下为写INI文件的键值的函数
void ini_file_write_string(IniFile *ini, gchar *section,
                gchar *key,gchar *value);
void ini_file_write_int(IniFile *ini, gchar *section,
                gchar *key, gint value);
void ini_file_write_boolean(IniFile *ini, gchar *section,
                gchar *key,gboolean value);
//以下为更改节名、删除键和节的函数
gboolean ini_file_rename_section(IniFile *ini, gchar *section,
                gchar *section_name);
gboolean ini_file_remove_key(IniFile *ini, gchar *section, gchar
*key);
gboolean ini_file_remove_section(IniFile *ini, gchar *section);
#endif
```

② INI 配置文件的代码实现 inifile.c，代码如下：

```
/* inifile.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/stat.h>
#include <unistd.h>
#include "inifile.h"
//以下为与节相关的创建与查找的函数
static IniSection *ini_file_create_section (IniFile *ini, gchar *
name);
static IniLine   *ini_file_create_string (IniSection *section,
gchar * key,
```

```
                        gchar * value);
static IniSection *ini_file_find_section (IniFile *ini, gchar *
name);
static IniLine    *ini_file_find_string (IniSection *section, gchar
* key);
static IniSection* ini_file_create_section (IniFile *ini, gchar *
name)
{   //创建节
    IniSection *section;
    section = g_malloc0 (sizeof (IniSection));
    section->name = g_strdup (name);
    ini->sections = g_list_append (ini->sections, section);
    return section;
}
static IniLine*
ini_file_create_string (IniSection *section, gchar * key,gchar *
value)
{   //创建字符串行
    IniLine *line;
    line = g_malloc0 (sizeof (IniLine));
    line->key = g_strchug (g_strchomp (g_strdup (key)));
    line->value = g_strchug (g_strchomp (g_strdup (value)));
    section->lines = g_list_append (section->lines, line);
    return line;
}
static IniSection* ini_file_find_section (IniFile *ini, gchar *
name)
{   //查找节
    IniSection *section;
    GList *list;
    list = ini->sections;
    while (list)
    {
        section = (IniSection *) list->data;
        if (!strcasecmp (section->name, name)) return section;
        list = g_list_next (list);
    }
    return NULL;
}
static IniLine*
ini_file_find_string (IniSection *section, gchar * key)
{   //查找字符串
    IniLine *line;
    GList *list;
    list = section->lines;
    while (list)
    {
        line = (IniLine *) list->data;
        if (!strcasecmp (line->key, key))  return line;
        list = g_list_next (list);
    }
```

```
    return NULL;
}
IniFile* ini_file_new()
{
    IniFile *ini;
    ini = g_malloc0(sizeof(IniFile));
    return ini;
}
IniFile* ini_file_open_file(gchar* filename)
{
    IniFile *ini;
    FILE *file;
    gchar *buffer, **lines, *tmp;
    gint i;
    struct stat stats;
    IniSection *section = NULL;
    if (lstat (filename, &stats) == -1)  return NULL;
    if (!(file = fopen (filename, "r"))) return NULL;
    buffer = g_malloc (stats.st_size + 1);
    if (fread (buffer, 1, stats.st_size, file) != stats.st_size)
    {
        g_free (buffer);
        fclose (file);
        return NULL;
    }
    fclose (file);
    buffer[stats.st_size] = '\0';
    ini = g_malloc0 (sizeof (IniFile));
    ini->filename = g_strdup (filename);
    ini->changed = 0;
    lines = g_strsplit (buffer, "\n", 0);
    g_free (buffer);
    i = 0;
    while (lines[i])
    {
        if (lines[i][0] == '[')
        {
            if ((tmp = strchr (lines[i], ']')))
            {
                *tmp = '\0';
                section = ini_file_create_section
(ini,&lines[i][1]);
            }
        }
        else if (lines[i][0] != '#' && section)
        {
            if ((tmp = strchr (lines[i], '=')))
            {
                *tmp = '\0';
                tmp++;
                ini_file_create_string (section, lines[i], tmp);
```

```
            }
        }
        i++;
    }
    g_strfreev (lines);
    return ini;
}
gboolean ini_file_write_file(IniFile* ini, gchar* filename)
{
    FILE *file;
    GList *section_list, *line_list;
    IniSection *section;
    IniLine *line;
    if (!(file = fopen (filename, "w"))) return FALSE;
    section_list = ini->sections;
    while (section_list)
    {
        section = (IniSection *) section_list->data;
        if (section->lines)
        {
            fprintf (file, "[%s]\n", section->name);
            line_list = section->lines;
            while (line_list)
            {
                line = (IniLine *) line_list->data;
                fprintf (file, "%s=%s\n", line->key,line->value);
                line_list = g_list_next (line_list);
            }
            fprintf (file, "\n");
        }
        section_list = g_list_next (section_list);
    }
    fclose (file);
    return TRUE;
}
void ini_file_free(IniFile* ini)
{
    IniSection *section;
    IniLine *line;
    GList *section_list, *line_list;
    g_free (ini->filename);
    section_list = ini->sections;
    while (section_list)
    {
        section = (IniSection *) section_list->data;
        g_free (section->name);
        line_list = section->lines;
        while (line_list)
        {
            line = (IniLine *) line_list->data;
            g_free (line->key);
```

```
            g_free (line->value);
            g_free (line);
            line_list = g_list_next (line_list);
        }
        g_list_free (section->lines);
        g_free (section);
        section_list = g_list_next (section_list);
    }
    g_list_free (ini->sections);
    ini->sections=NULL;
    ini->filename=NULL;
}
gboolean    ini_file_read_string(IniFile *ini, gchar * section,
 gchar * key,gchar ** value)
{   //读字符串型键值
    IniSection *sect;
    IniLine *line;
    *value=NULL;
    if (!(sect = ini_file_find_section (ini, section))) return
FALSE;
    if (!(line = ini_file_find_string (sect, key))) return FALSE;
    *value = g_strdup (line->value);
    return TRUE;
}
gboolean
ini_file_read_int(IniFile *ini, gchar *section, gchar *key, gint
*value)
{   //读整型键值
    gchar *str;
    if (!ini_file_read_string (ini, section, key, &str))
    {
        *value=0;
        return FALSE;
    }
    *value = atoi (str);
    g_free (str);
    return TRUE;
}
gboolean  ini_file_read_boolean(IniFile *ini, gchar *section,
 gchar *key,gboolean *value)
{   //读逻辑型键值
    gchar *str;
    if (!ini_file_read_string (ini, section, key, &str))
    {
        *value= FALSE;
        return FALSE;
    }
    if (!strcmp (str, "0"))
        *value = FALSE;
    else
        *value = TRUE;
```

```
    g_free (str);
    return TRUE;
}
void
ini_file_write_string(IniFile *ini, gchar *section, gchar
*key,gchar *value)
{   写字符型键值
    IniSection *sect;
    IniLine *line;
    ini->changed = 1;
    sect = ini_file_find_section (ini, section);
    if (!sect)
        sect = ini_file_create_section (ini, section);
    if ((line = ini_file_find_string (sect, key)))
    {
        g_free (line->value);
        line->value = g_strchug (g_strchomp (g_strdup (value)));
    }
    else
        ini_file_create_string (sect, key, value);
}
void
ini_file_write_int(IniFile *ini, gchar *section, gchar *key, gint
value)
{   //写整型键值
    gchar *strvalue;
    strvalue = g_strdup_printf ("%d", value);
    ini_file_write_string (ini, section, key, strvalue);
    g_free (strvalue);
}
void
ini_file_write_boolean(IniFile *ini, gchar *section,
gchar *key,gboolean value)
{   //写逻辑型键值
    if (value)
        ini_file_write_string (ini, section, key, "1");
    else
        ini_file_write_string (ini, section, key, "0");
}
gboolean
ini_file_rename_section(IniFile *ini, gchar *section,gchar
*section_name)
{   //改节名
    IniSection *sect;
    sect = ini_file_find_section (ini, section);
    if (sect)
    {
        ini->changed = 1;
        g_free (sect->name);
        sect->name = g_strdup (section_name);
    }
```

```
}
gboolean
ini_file_remove_key(IniFile *ini, gchar *section, gchar *key)
{   //删除键
    IniSection *sect;
    IniLine *line;
    sect = ini_file_find_section (ini, section);
    if (sect)
    {
        line = ini_file_find_string (sect, key);
        if (line)
        {
            ini->changed = 1;
            g_free (line->key);
            g_free (line->value);
            g_free (line);
            sect->lines = g_list_remove (sect->lines, line);
        }
    }
}
gboolean
ini_file_remove_section(IniFile *ini, gchar *section)
{   //删除节
    IniSection *sect;
    IniLine *line;
    GList *line_list;
    sect = ini_file_find_section (ini, section);
    if (sect)
    {
        ini->changed = 1;
        g_free (sect->name);
        line_list = sect->lines;
        while (line_list)
        {
            line = (IniLine *) line_list->data;
            g_free (line->key);
            g_free (line->value);
            g_free (line);
            line_list = g_list_next (line_list);
        }
        g_list_free (sect->lines);
        g_free (sect);
        ini->sections = g_list_remove (ini->sections, sect);
    }
}
```

③ 使用 INI 文件，代码如下：

```
/* 使用INI文件 ini.c */
#include <gtk/gtk.h>
#include "inifile.h"
```

```
void    on_button_clicked   (GtkButton* button,gpointer data)
{
    IniFile *ini ;
    ini = ini_file_new();
    ini_file_write_string(ini,"SYSTEM","gtk+","2.0.2");
    ini_file_write_int(ini,"SYSTEM","NO.",1234);
    ini_file_write_boolean(ini,"SYSTEM","Regsition",TRUE);
    ini_file_write_string(ini,"ABOUT","software","for you");
    ini_file_write_file(ini,"TEST.INI");
    ini_file_free(ini);
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *button;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"使用INI文件");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTE
R);
    gtk_container_set_border_width(GTK_CONTAINER(window),20);
    button = gtk_button_new_with_label
("单击此按钮会创建一个名为\nTEST.INI的文件");
    g_signal_connect(G_OBJECT(button),"clicked",
                G_CALLBACK(on_button_clicked),NULL);
    gtk_container_add(GTK_CONTAINER(window),button);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
ini:ini.o inifile.o inifile.h
    $(CC) -o ini ini.o inifile.o 'pkg-config --libs gtk+-2.0'
ini.o:ini.c
    $(CC) -c ini.c -o ini.o 'pkg-config --cflags gtk+-2.0'
inifile.o:inifile.c inifile.h
    $(CC) -c inifile.c -o inifile.o 'pkg-config --cflags --libs glib-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./ini 即可运行此程序，运行结果如图 8.2 所示。

图 8.2 使用 INI 文件

实例分析

(1) 运行结果

程序运行后，单击按钮，会创建一名为 test.ini 的文件，文件的内容的如下：

```
[SYSTEM]
gtk+-2.0.2
NO.=1234
Regsit.cn=1
[ABOUT]
software=for you
```

(2) 各函数的实现

此示例的代码中运用到了 GLIB 中的链表(GList)等数据结构，运用了 GLIB 中的分配内存函数 g_malloc 和释放内存函数 g_free，最为关键的是有关字符串的转换和文件的读写等操作，读者一定要细细体会其中的奥妙。

灵活使用此示例中的各个函数您一定会做出一个非常好用的配置文件来。此示例是在参考网上的一些开源项目的源代码来实现的，开源软件是学习编程的一个良好的工具，多看一些开源软件的代码对提高编程水平来说是必要的。

8.3 名 片 管 理

本节示例将通过灵活使用 INI 配置文件和综合运用 GTK+2.0 控件来实现一个简单的名片管理软件。

实例说明

上节示例中 INI 文件的测试非常简单，本节我们用 INI 文件来做一个名片管理程序，将名片的各项数据保存到 INI 文件中，然后再依次读取出来。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/8
mkdir card
cd card
```

创建本节的工作目录，进入此目录开始编程，将上节中的两个文件 inifile.h 和 inifile.c 复制到当前目录下。

(2) 打开编辑器，输入以下代码，以 card.c 为文件名保存到当前目录下。

```
/* 卡片管理 card.c */
#include <gtk/gtk.h>
#include "inifile.h"
/* 卡片共定义五项，姓名，性别，地址，职业，EMAIL */
/* 此处定义与系统相关的变量，保存卡片的各个项目 */
```

```
static GtkWidget *frame, *name, *sex ;
static GtkWidget *job, *address, *email;
gint current = 1;   //当前卡片号
gint count = 0;     //卡片总数
gboolean    isrebuilc = FALSE;
static GtkWidget *win; /* 此处定义与添加窗口相关的控件 */
static GtkWidget* name_entry;  /* 姓名 */
static GtkWidget* sex_entry;   /* 性别 */
static GtkWidget *radio1,*radio2;
static GtkWidget* addr_entry;  /* 地址 */
static GtkWidget* job_entry;   /* 职业 */
static GtkWidget* email_entry; /* 电子邮件 */
void    rebuild     (void);     //重建
void    read_item   (gint i);   //读卡片
void    save_item   (gint i);   //写卡片
void    read_card   (void)
{
    IniFile *ini;
    if(g_file_test("./our.card",G_FILE_TEST_EXISTS) == TRUE)
    {
        ini = ini_file_open_file("our.card");
        ini_file_read_int(ini,"SYSTEM","count",&count);
        ini_file_free(ini);
    }
    else
    {
        isrebuild = TRUE;
        rebuild();
    }
}
void    rebuild (void)
{
    IniFile *ini;/* 重建卡片库 */
    ini = ini_file_new();
    ini_file_write_string(ini,"SYSTEM","about","卡片管理系统");
    ini_file_write_int(ini,"SYSTEM","count",0);
    ini_file_write_file(ini,"our.card");
    ini_file_free(ini);
}
void    read_item   (gint i)
{
    IniFile *ini;   /* 读卡片项 */
    gchar id[10];
    gchar *as_item;
    gchar buf[256];
    if(count == 0)  return;
    sprintf(id,"%d",i);
    ini = ini_file_open_file("our.card");
    ini_file_read_string(ini,id,"name",&as_item);
    sprintf(buf,"姓名: %s",as_item);
    gtk_label_set_text(GTK_LABEL(name),buf);
```

```
    ini_file_read_string(ini,id,"sex",&as_item);
    sprintf(buf,"性别: %s",as_item);
    gtk_label_set_text(GTK_LABEL(sex),buf);
    ini_file_read_string(ini,id,"address",&as_item);
    sprintf(buf,"地址: %s",as_item);
    gtk_label_set_text(GTK_LABEL(address),buf);
    ini_file_read_string(ini,id,"job",&as_item);
    sprintf(buf,"职业: %s",as_item);
    gtk_label_set_text(GTK_LABEL(job),buf);

    ini_file_read_string(ini,id,"email",&as_item);
    sprintf(buf,"电子邮件: %s",as_item);
    gtk_label_set_text(GTK_LABEL(email),buf);
    ini_file_free(ini);
}
void    save_item   (gint i)
{
    const gchar *address;
    const gchar *job;
    const gchar *email;
    gchar *sex;
    const gchar *name;
    gchar idbuf[10];
    IniFile *ini;   /* 写卡片项 */
    ini = ini_file_open_file("our.card");
    sprintf(idbuf,"%d",i);
    name = gtk_entry_get_text(GTK_ENTRY(name_entry));
    ini_file_write_string(ini,idbuf,"name",name);//姓名
    if(GTK_TOGGLE_BUTTON(radio1)->active)  //判断单选按钮是否被选中
        sex = "男";
    else
        sex = "女";
    ini_file_write_string(ini,idbuf,"sex",sex);//性别
    address = gtk_entry_get_text(GTK_ENTRY(addr_entry));
    ini_file_write_string(ini,idbuf,"address",address);//住址
    job = gtk_entry_get_text(GTK_ENTRY(job_entry));
    ini_file_write_string(ini,idbuf,"job",job);

    email = gtk_entry_get_text(GTK_ENTRY(email_entry));
    ini_file_write_string(ini,idbuf,"email",email);//电子邮件
    ini_file_write_int(ini,"SYSTEM","count",i);
    ini_file_write_file(ini,"our.card");
    ini_file_free(ini);
}
void    on_ok_clicked   (GtkButton* buttor,gpointer data)
{
    count++;
    save_item(count);
    gtk_widget_destroy(win);
}
GtkWidget*  create_button   (gchar* stockid)
```

```
{
    GtkWidget* button;
    GtkWidget* image;
    image = gtk_image_new_from_stock(stockid,GTK_ICON_SIZE_MENU);
    button = gtk_button_new();
    gtk_container_add(GTK_CONTAINER(button),image);
    return button;
}
GtkWidget* create_addwin   (void)  //创建添加记录的窗口
{
    GtkWidget* window;
    GtkWidget* vbox;
    GtkWidget* bbox;
    GtkWidget* label;
    GtkWidget* table;
    GtkWidget* button;
    GtkWidget* sep;

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_widget_destroy),window);
    gtk_window_set_title(GTK_WINDOW(window),"添加记录");
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    table = gtk_table_new(5,2,FALSE);
    gtk_box_pack_start(GTK_BOX(vbox),table,FALSE,FALSE,5);
    label = gtk_label_new("姓名: ");
    gtk_table_attach_defaults(GTK_TABLE(table),label,0,1,0,1);
    name_entry = gtk_entry_new();
    gtk_table_attach_defaults(GTK_TABLE(table),name_entry,1,2,0,1);
    label = gtk_label_new("性别: ");
    gtk_table_attach_defaults(GTK_TABLE(table),label,0,1,1,2);
    sex_entry = gtk_hbox_new(FALSE,0);//gtk_entry_new();
    gtk_table_attach_defaults(GTK_TABLE(table),sex_entry,1,2,1,2);
    radio1 = gtk_radio_button_new_with_label(NULL,"男");
    radio2 = gtk_radio_button_new_with_label_from_widget(radio1,"女
");
    gtk_box_pack_start(GTK_BOX(sex_entry),radio1,FALSE,FALSE,2);
    gtk_box_pack_start(GTK_BOX(sex_entry),radio2,FALSE,FALSE,2);
    label = gtk_label_new("住址: ");
    gtk_table_attach_defaults(GTK_TABLE(table),label,0,1,2,3);
    addr_entry = gtk_entry_new();
    gtk_table_attach_defaults(GTK_TABLE(table),addr_entry,1,2,2,3);
    label = gtk_label_new("职业: ");
    gtk_table_attach_defaults(GTK_TABLE(table),label,0,1,3,4);
    job_entry = gtk_entry_new();
    gtk_table_attach_defaults(GTK_TABLE(table),job_entry,1,2,3,4);

    label = gtk_label_new("Email: ");
    gtk_table_attach_defaults(GTK_TABLE(table),label,0,1,4,5);
```

```
    email_entry = gtk_entry_new();
    gtk_table_attach_defaults(GTK_TABLE(table),email_entry,1,2,4,5);
    sep = gtk_hseparator_new();
    gtk_box_pack_start(GTK_BOX(vbox),sep,FALSE,FALSE,5);
    bbox = gtk_hbutton_box_new();
    gtk_box_pack_start(GTK_BOX(vbox),bbox,FALSE,FALSE,5);

    button = gtk_button_new_from_stock(GTK_STOCK_OK);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_ok_clicked),NULL);
    gtk_box_pack_start(GTK_BOX(bbox),button,FALSE,FALSE,5);
    gtk_widget_show_all(window);
    return window;
}
static  void    goto_top        (GtkButton* button,gpointer data)
{   //首记录
    current = 1;
    read_item(current);
}
static  void    go_up       (GtkButton* button,gpointer data)
{   //上一记录
    current--;
    if(current == 0)
    {
        current = 1 ;
        return ;
    }
    read_item(current);
}
static  void    go_down     (GtkButton* button,gpointer data)
{   //下一记录
    current++;
    if(current > count)
    {
        current = count ;
        return ;
    }
    read_item(current);
}
static  void    goto_bottom (GtkButton* button,gpointer data)
{   //尾记录
    current = count;
    read_item(current);
}
static  void    add_item        (GtkButton* button,gpointer data)
{   //添加记录
    win = create_addwin();
    gtk_widget_show(win);
}
void    show_info   (void)
{
```

```
    GtkWidget* dialog;/* 显示重建信息 */
    dialog = gtk_message_dialog_new(NULL,
        GTK_DIALOG_MODAL | GTK_DIALOG_DESTROY_WITH_PARENT,
        GTK_MESSAGE_INFO,
        GTK_BUTTONS_OK,
        "未发现卡片数据库，在此目录下重建成功。\n名称为：our.card");
    gtk_dialog_run(GTK_DIALOG(dialog));
    gtk_widget_destroy(dialog);
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget* paned;
    GtkWidget* cbox;        /* 上面的框架 */
    GtkWidget* hbox;        /* 下面的按钮 */
    GtkWidget* button;
    GtkWidget* sep;
    GtkTooltips* button_tips;

    gtk_init(&argc,&argv);
    read_card();    /* 读卡片库,测试其是否存在 */
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"卡片管理");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);

    paned = gtk_vpaned_new();
    gtk_container_add(GTK_CONTAINER(window),paned);
    frame = gtk_frame_new(NULL);
    gtk_paned_add1(GTK_PANED(paned),frame);
    cbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(frame),cbox);
    gtk_container_set_border_width(GTK_CONTAINER(cbox),5);

    name = gtk_label_new("姓名：");
    gtk_box_pack_start(GTK_BOX(cbox),name,FALSE,FALSE,2);
    sex = gtk_label_new("性别：");
    gtk_box_pack_start(GTK_BOX(cbox),sex,FALSE,FALSE,2);
    address = gtk_label_new("住址：");
    gtk_box_pack_start(GTK_BOX(cbox),address,FALSE,FALSE,2);
    job = gtk_label_new("职业：");
    gtk_box_pack_start(GTK_BOX(cbox),job,FALSE,FALSE,2);
    email = gtk_label_new("电子邮件：");
    gtk_box_pack_start(GTK_BOX(cbox),email,FALSE,FALSE,2);

    hbox = gtk_hbox_new(FALSE,0);/* 创建下面的按钮 */
    gtk_container_set_border_width(GTK_CONTAINER(hbox),5);
    gtk_paned_add2(GTK_PANED(paned),hbox);
    button_tips = gtk_tooltips_new();
```

```
    button = create_button(GTK_STOCK_GOTO_FIRST);
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,2);
    gtk_tooltips_set_tip(GTK_TOOLTIPS(button_tips),button,
"第一张","首张");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(goto_top),NULL);
    button = create_button(GTK_STOCK_GO_BACK);
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,2);
    gtk_tooltips_set_tip(GTK_TOOLTIPS(button_tips),button,
"上一张","上张");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(go_up),NULL);

    button = create_button(GTK_STOCK_GO_FORWARD);
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,2);
    gtk_tooltips_set_tip(GTK_TOOLTIPS(button_tips),button,
"下一张","下张");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(go_down),NULL);
    button = create_button(GTK_STOCK_GOTO_LAST);
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,2);
    gtk_tooltips_set_tip(GTK_TOOLTIPS(button_tips),button,
"最后一张","尾张");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(goto_bottom),NULL);
    sep = gtk_vseparator_new();
    gtk_box_pack_start(GTK_BOX(hbox),sep,FALSE,FALSE,2);

    button = create_button(GTK_STOCK_ADD);
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,2);
    gtk_tooltips_set_tip(GTK_TOOLTIPS(button_tips),button,
"增加一张","增加");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(add_item),NULL);
    sep = gtk_vseparator_new();
    gtk_box_pack_start(GTK_BOX(hbox),sep,FALSE,FALSE,2);
    button = create_button(GTK_STOCK_QUIT);
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,2);
    gtk_tooltips_set_tip(GTK_TOOLTIPS(button_tips),button,
"退出卡片管理","退出");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_widget_show_all(window);
    if(isrebuild == TRUE)
        show_info();
    else
        read_item(1);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc3
card:   card.o inifile.o
    $(CC) -o card card.o inifile.o 'pkg-config --libs gtk+-2.0'
card.o: card.c
    $(CC) -c card.c 'pkg-config --cflags gtk+-2.0'
inifile.o:inifile.c inifile.h
    $(CC) -c inifile.c 'pkg-config --cflags glib-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./card 即可运行此程序，运行结果如图 8.3 所示。

图 8.3 名片管理

实例分析

(1) 功能与外观

此示例是用一个名为 our.card 的文件来保存名片信息，而且名片的内容只有 5 项，在主窗口的外观上与第 5 章的表格软件有些类似，添加窗口的“性别”一项采用了单选按钮的形式，这更符合用户的输入习惯。

(2) 综合运用各种控件的能力

此示例中用到了前面学过的多种控件，综合运用各种控件和编程技巧，在文件的设计和读写上还是比较有特点的。程序一开始运行搜索当前目录下是否有名片文件“our.card”，如果没有则重新创建，如果有则用 INI 形式打开，并读取其内容显示到框架中去。名片的数量保存在[SYSTEM]节的 count 键值中，可以根据此值来上下翻动名片。添加名片的方法和第 5 章中的向表格中添加数据的方法相似，只不过是用 INI 的形式来读写文件。

此示例进一步发挥了 INI 文件的用法和 GTK+2.0 控件的技巧，是一个很好的综合应用，有兴趣的读者可以为它加上删除名片的功能。

8.4 图片查看器

本节示例将通过灵活运用图像控件和创建窗口的功能来实现一个简单的图片查看软件。

实例说明

图片查看软件对初学者来说是一个较有诱惑力的课题，尤其是对图像有特殊兴趣的朋友。本示例非常简单，主窗口中显示了一幅图像和一个按钮，单击此按钮，弹出文件选择对话框，选择图像文件，单击确定，程序会打开此图像文件并新建一窗口将图像显示出来。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/8
mkdir imageview
cd imageview
```

创建本节的工作目录，进入此目录开始编程。用 GIMP 创建一幅 PNG 格式的图像，以文件名 viewer.png 保存到当前目录下。

(2) 打开编辑器，输入以下代码，以 view.c 为文件名保存到当前目录下。

```
/* 图像查看软件 view.c */
#include <gtk/gtk.h>
static GtkWidget* dialog = NULL;
GtkWidget*  create_view (gchar* filename)
{   //创建新的图像窗口
    GtkWidget* window;
    GtkWidget* image;
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),filename);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
    image = gtk_image_new_from_file(filename);
    gtk_container_add(GTK_CONTAINER(window),image);
    gtk_widget_show_all(window);
    return window;
}
void    on_ok   (GtkButton* button, gpointer data)
{   //文件选择确定时
    const char* filename;
    GtkWidget* window;
    filename =
gtk_file_selection_get_filename(GTK_FILE_SELECTION(data));
    window = create_view(filename);//创建新图像窗口
    gtk_widget_show(window);
}
void    on_cancel   (GtkButton *button,gpointer data)
{   //取消
    gtk_widget_destroy(dialog);
}
void    on_clicked (GtkButton *button, gpointer data)
```

```
{   //打开文件选择对话框
    dialog = gtk_file_selection_new("请选择一个图像文件");
    g_signal_connect(G_OBJECT(dialog),"destroy",
            G_CALLBACK(gtk_widget_destroy),dialog);
    g_signal_connect(G_OBJECT(GTK_FILE_SELECTION(dialog)->ok_button)
,
"clicked", G_CALLBACK(on_ok),dialog);
    g_signal_connect_swapped(G_OBJECT(
GTK_FILE_SELECTION(dialog)->ok_button),"clicked",
            G_CALLBACK(on_cancel),NULL);
    g_signal_connect(G_OBJECT(GTK_FILE_SELECTION(dialog)->cancel_but
ton),
"clicked", G_CALLBACK(on_cancel),NULL);
    gtk_widget_show(dialog);
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *image, *button;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"图像查看软件");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    image = gtk_image_new_from_file("viewer.png");
    gtk_box_pack_start(GTK_BOX(vbox),image,FALSE,FALSE,5);
    button = gtk_button_new_with_label("单击此按钮来选择图像文件");
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_clicked),NULL);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o view view.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./view 即可运行此程序，运行结果如图 8.4 所示。

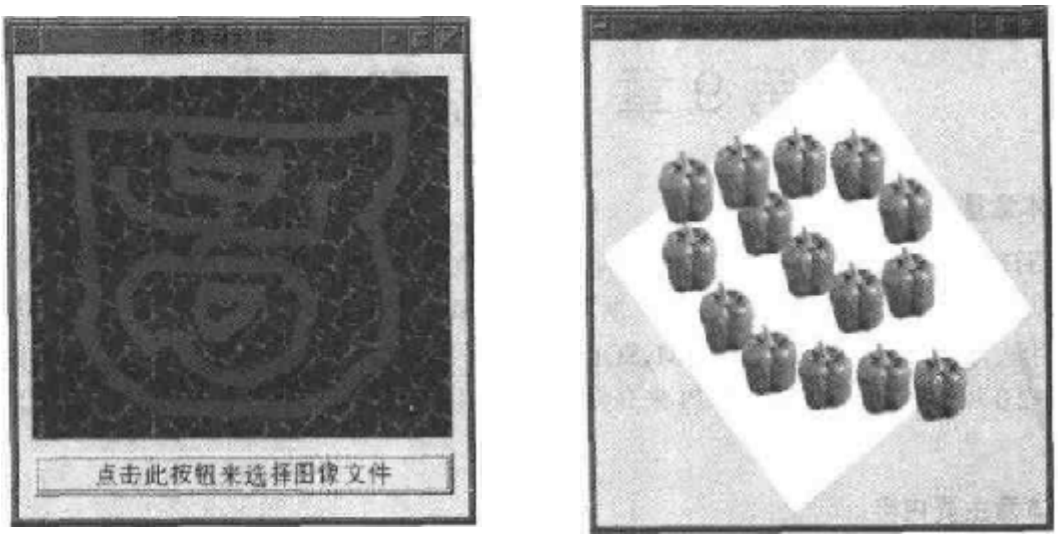

图 8.4　图片查看软件

实例分析

此示例的关键是用函数gtk_file_selection_get_filename取得文件选择对话框选择的文件名，然后根据此文件名用函数 gtk_image_new_from_file 来创建图像控件，创建窗口，将图像控件加到窗口中去，再将窗口显示出来。

这大概是最简短的图像查看软件了，还需要解决几个问题，当选择的文件不是图像时会显示一个默认的图像，我们的希望是提示“非图像文件”，文件规格较小时我们可以完整的看到，要是超过屏幕大小该怎么办呢？非常希望有兴趣的读者解决此类问题。

本章中的 4 个示例可以说是以文件操作为核心，灵活运用 GTK+2.0 的各种控件来实现的，尤其在多文件的组织上。

第 9 章　数据库编程

本章重点：

GTK+2.0 是图形界面设计工具，本身不具有数据库功能，由于它采用 C 语言设计，具有强大的创建应用程序界面的功能，所以很容易和一些数据库系统在底层结合，开发出数据应用程序。本章以网上常用的 MySQL 数据库系统为 GTK+2.0 的结合对象，介绍如何用 GTK+2.0 和 MySQL 的 C 语言 API 来开发数据库应用程序。读者学习本章的前提是熟悉 SQL 语言。

本章主要内容：

- 连接、断开数据库服务器
- 创建、删除数据库和数据表
- 向数据表中插入数据
- 从数据表中读取数据
- 大型数据的插入和读取

9.1　连接 MySQL 服务器与创建数据库、数据表

本节示例将介绍如何编程实现与 MySQL 数据库服务器的连接与断开，如何在程序中执行 SQL 语句实现数据库、数据表的创建与删除以及利用对话框简化用户操作。

实例说明

通过图形界面程序来操作数据库，这样的程序在 Windows 环境下有很多。此示例运用 GTK+2.0 编写界面，结合 MySQL 的 C 语言 API，实现了数据服务器的连接、断开，数据的创建、删除、选用和数据表的创建、删除等功能，用到了 GTK+2.0 中的对话框以及一些常用控件。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/
mkdir 9
cd 9
mkdir 1c
cd 1c
```

创建本章总目录和本节工作目录，进入目录开始编程。

(2) 打开编辑器，输入以下代码，以 create.c 为文件名保存到当前目录下。

```
/* 创建删除数据库和数据表 create.c */
```

```
#include <gtk/gtk.h>
#include <mysql.h>
MySQL *myconnect = NULL ;
gboolean isclosed = TRUE;  //是否关闭
gboolean iscreate = TRUE;  //是否创建
static GtkWidget *dialog = NULL ;  //数据库对话框
static GtkWidget *entry = NULL ;
static GtkWidget *table_dialog = NULL; //创建数据表对话框
static GtkWidget *db_entry = NULL;
static GtkWidget *text = NULL; //存贮字段值
static GtkTextBuffer *text_buffer;
gboolean isok = FALSE; //是否准备好
static GtkWidget *drop_table_dialog = NULL ;//删除数据表对话框
static GtkWidget *drop_db_entry = NULL ;
static GtkWidget *drop_table_entry = NULL ;
static GtkTextBuffer *message_buffer;
static GtkWidget *mlabel;

void    create_message_dialog   (GtkMessageType type, gchar* message)
{
    GtkWidget* dialogx;
    dialogx = gtk_message_dialog_new(NULL,
            GTK_DIALOG_MODAL | GTK_DIALOG_DESTROY_WITH_PARENT,
            type , GTK_BUTTONS_OK, message);
    gtk_dialog_run(GTK_DIALOG(dialogx));
    gtk_widget_destroy(dialogx);
}
void    on_dialog_yes   (GtkButton *button, gpointer data)
{
    gchar query_buf[4096];
    const gchar* dbname;
    dbname = gtk_entry_get_text(GTK_ENTRY(entry));
    if(iscreate == TRUE)
    {
        sprintf(query_buf,"CREATE DATABASE %s",dbname);
        if(mysql_query(myconnect,query_buf)==0)
        {
            create_message_dialog(GTK_MESSAGE_INFO,"成功创建数据库！");
        }
        else
        {
            create_message_dialog(GTK_MESSAGE_ERROR,"创建数据库时出错！
");
        }
    }
    else
    {
        sprintf(query_buf,"DROP DATABASE %s",dbname);
        if(mysql_query(myconnect, query_buf) == 0)
        {
            create_message_dialog(GTK_MESSAGE_WARNING,"数据库已经被删
```

```
除！");
        }
        else
        {
            create_message_dialog(GTK_MESSAGE_ERROR,"删除数据库时出错！
");
        }
    }
    gtk_widget_destroy(dialog);
}
void    on_dialog_no    (GtkButton *button, gpointer data)
{
    gtk_widget_destroy(dialog);
}
//创建删除数据库对话框
void    create_run_dialog   (gchar *title)
{
    GtkWidget *label, *vbox, *button, *sep, *hbox;
    dialog = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(dialog),title);
    g_signal_connect(G_OBJECT(dialog),"delete_event",
            G_CALLBACK(gtk_widget_destroy),dialog);
    gtk_container_set_border_width(GTK_CONTAINER(dialog),10);

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(dialog),vbox);
    label = gtk_label_new("输入数据库名: ");
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,5);
    entry = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(vbox),entry,FALSE,FALSE,5);
    sep = gtk_hseparator_new();
    gtk_box_pack_start(GTK_BOX(vbox),sep,FALSE,FALSE,5);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    button = gtk_button_new_from_stock(GTK_STOCK_YES);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_dialog_yes),NULL);
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);
    button = gtk_button_new_from_stock(GTK_STOCK_NO);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_dialog_no),NULL);
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);
    gtk_widget_show_all(dialog);
}
void    on_db_use   (GtkButton *button,gpointer data)
{
    const char* query;
    char query_buf[1024];
    query = gtk_entry_get_text(GTK_ENTRY(db_entry));
    sprintf(query_buf,"USE %s",query);
    if(mysql_query(myconnect,query_buf) == 0 )
```

```
    {
        create_message_dialog(GTK_MESSAGE_INFO,"数据库选用成功！");
        gtk_widget_set_sensitive(db_entry,FALSE);
    }
    else
    {
        create_message_dialog(GTK_MESSAGE_ERROR,"选用数据库时出错！");
    }
}
void    on_create_table_yes (GtkButton* button, gpointer data)
{
    gchar* sql_query;
    GtkTextIter iter1,iter2;
    gtk_text_buffer_get_start_iter(text_buffer,&iter1);
    gtk_text_buffer_get_end_iter(text_buffer,&iter2);
    sql_query =
gtk_text_buffer_get_text(text_buffer,&iter1,&iter2,FALSE);
    //
    if(mysql_query(myconnect,sql_query) == 0)
    {
        create_message_dialog(GTK_MESSAGE_INFO,"成功创建数据表！");
    }
    else
    {
        create_message_dialog(GTK_MESSAGE_ERROR,"运行SQL语句时出错！");
    }
}

void    create_table_dialog (gchar* title)//创建数据表对话框
{
    GtkWidget *vbox, *hbox, *label, *button, *viewport;
    table_dialog = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(table_dialog),title);
    g_signal_connect(G_OBJECT(table_dialog),"delete_event",
            G_CALLBACK(gtk_widget_destroy),table_dialog);
    gtk_container_set_border_width(GTK_CONTAINER(table_dialog),10);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(table_dialog),vbox);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    label = gtk_label_new("数据库名称：");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    db_entry = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(hbox),db_entry,FALSE,FALSE,5);
    button = gtk_button_new_with_label("选用");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_db_use),NULL);
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);

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

```
    viewport = gtk_viewport_new(NULL,NULL);
    gtk_box_pack_start(GTK_BOX(hbox),viewport,TRUE,TRUE,5);
    text = gtk_text_view_new();
    //gtk_widget_set_usize(text,100,100);
    gtk_container_add(GTK_CONTAINER(viewport),text);
    text_buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(text));

    vbox = gtk_vbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(hbox),vbox,FALSE,FALSE,5);
    label = gtk_label_new("请在左侧的\n文本输入区\n域内输入创\n建数据表的
\nSQL语我句，\n单击下面的\n按钮即可执\n行创建数据\n表操作");
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,5);
    button = gtk_button_new_with_label("创建数据表");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_create_table_yes),NULL);
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    gtk_widget_show_all(table_dialog);
}
void    on_drop_table_yes   (GtkButton* button, gpointer data)
{
    gchar query_buf[4096];
    const gchar *dbname, *tablename;
    dbname = gtk_entry_get_text(GTK_ENTRY(drop_db_entry));
    tablename = gtk_entry_get_text(GTK_ENTRY(drop_table_entry));
    sprintf(query_buf,"USE %s",dbname);
    if(mysql_query(myconnect,query_buf) == 0)
    {
        sprintf(query_buf,"DROP TABLE %s",tablename);
        if(mysql_query(myconnect,query_buf) == 0)
        {
            create_message_dialog(GTK_MESSAGE_WARNING,
                "数据库已经打开，数据表成功删除！");
        }
        else
        {
            create_message_dialog(GTK_MESSAGE_ERROR,
                "数据库已经打开，但数据表并未删除 ！");
        }
    }
    else
    {
        create_message_dialog(GTK_MESSAGE_ERROR,"打开数据库时出错！");
    }
    gtk_widget_destroy(drop_table_dialog);
}
void    on_drop_table_no    (GtkButton* button, gpointer data)
{
    gtk_widget_destroy(drop_table_dialog);
}
void    create_drop_table_dialog    (gchar* title)//删除数据表对话框
{
```

```
    GtkWidget *vbox, *hbox, *label,*button,*sep;
    drop_table_dialog = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(drop_table_dialog),title);
    g_signal_connect(G_OBJECT(drop_table_dialog),"delete_event",
            G_CALLBACK(gtk_widget_destroy),drop_table_dialog);
    gtk_container_set_border_width(GTK_CONTAINER(drop_table_dialog),
10);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(drop_table_dialog),vbox);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    label = gtk_label_new("打开的数据库: ");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    drop_db_entry = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(hbox),drop_db_entry,FALSE,FALSE,5);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    label = gtk_label_new("要删除的数据表: ");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    drop_table_entry = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(hbox),drop_table_entry,FALSE,FALSE,5)

    sep = gtk_hseparator_new();
    gtk_box_pack_start(GTK_BOX(vbox),sep,FALSE,FALSE,5);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    button = gtk_button_new_from_stock(GTK_STOCK_YES);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_drop_table_yes),NULL);
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);

    button = gtk_button_new_from_stock(GTK_STOCK_NO);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_drop_table_no),NULL);
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);
    gtk_widget_show_all(drop_table_dialog);
}
gboolean my_connect () //连接
{
    myconnect = mysql_init(myconnect);
    if(mysql_real_connect(myconnect,"localhost",
NULL,NULL,NULL,MySQL_PORT,NULL,0))
    {
        return TRUE;
    }
    else
    {
        myconnect = NULL;
        return FALSE;
    }
}
```

```
void my_disconnect()    //断开
{
    mysql_close(myconnect);
    myconnect = NULL;
}
//-------------------------------------------------------------
void    on_button_connect (GtkButton *button, gpointer data)
{
    if ( my_connect() == FALSE )
    {
        gtk_label_set_text(GTK_LABEL(mlabel),"错误：不能与数据库服务器连
接。");
    }
    else
    {
        gtk_label_set_text(GTK_LABEL(mlabel),"信息：成功与数据库服务器连
接。");
        isclosed = FALSE;
    }
}

void    on_button_disconnect (GtkButton *button, gpointer data)
{
    my_disconnect();
    isclosed = TRUE;
    gtk_label_set_text(GTK_LABEL(mlabel),"注意:成功与数据库服务器断开。");
}
void    on_create   (GtkButton* button, gpointer data)
{
    create_run_dialog("创建mysql数据库");
    iscreate = TRUE;
}
void    on_drop     (GtkButton* button, gpointer data)
{
    create_run_dialog("删除mysql数据库");
    iscreate = FALSE;
}
void    on_create_table (GtkButton* button, gpointer data)
{
    create_table_dialog("创建数据表");
}
void    on_drop_table   (GtkButton* button, gpointer data)
{
    create_drop_table_dialog("删除数据表");
}
void    on_delete_event (GtkWidget* widget, GdkEvent* event, gpointer
data)
{
    if(isclosed == FALSE)
    {
```

```
        my_disconnect();
        create_message_dialog(GTK_MESSAGE_INFO,"成功与服务器断开！");
    }
    gtk_main_quit();
}
int main (int argc, char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox1, *hbox, *vbox, *viewport;
    GtkWidget *button, *message;
    GtkTextIter iter;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(on_delete_event),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"创建/删除数据库和数据表");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);

    vbox1 = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox1);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox1),hbox,FALSE,FALSE,5);
    message = gtk_text_view_new();
    gtk_box_pack_start(GTK_BOX(hbox),message,TRUE,TRUE,5);
    message_buffer =
gtk_text_view_get_buffer(GTK_TEXT_VIEW(message));
    gtk_text_buffer_get_end_iter(message_buffer,&iter);
    gtk_text_buffer_insert(message_buffer,&iter,"此处用来显示一些与软件
\n使用有关的数据信息。\n可以到程序中找到这段代码，\n并应用到自己的回调函数中。\n
与MySQL数据库服务器连接\n需要用到服务器名、用户名\n和密码等参数，这些参数的\n设
置可以到MySQL的帮助\n文档中找到，MySQL的C语\n言API的详细说明也可以在\n此文档中
找到。",-1);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(hbox),vbox,FALSE,FALSE,5);
    button = gtk_button_new_with_label("连接服务器");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_button_connect),NULL);
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    button = gtk_button_new_with_label("断开");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_button_disconnect),NULL);
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    button = gtk_button_new_with_label("创建数据库");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_create),NULL);
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    button = gtk_button_new_with_label("删除数据库");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_drop),NULL);
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
```

```
    button = gtk_button_new_with_label("创建数据表");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_create_table),NULL);
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    button = gtk_button_new_with_label("删除数据表");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_drop_table),NULL);
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    viewport = gtk_viewport_new(NULL,NULL);
    gtk_box_pack_start(GTK_BOX(vbox1),viewport,FALSE,FALSE,5);
    mlabel = gtk_label_new("此处也可以显示提示信息");
    gtk_container_add(GTK_CONTAINER(viewport),mlabel);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
LIB = -I/usr/include/mysql -L/usr/lib/mysql -lmysqlclient
all:
    $(CC) -o create create.c $(LIB) 'pkg-config gtk+-2.0 --cflags --libs'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./create 即可运行此程序，运行结果如图 9.1 所示。

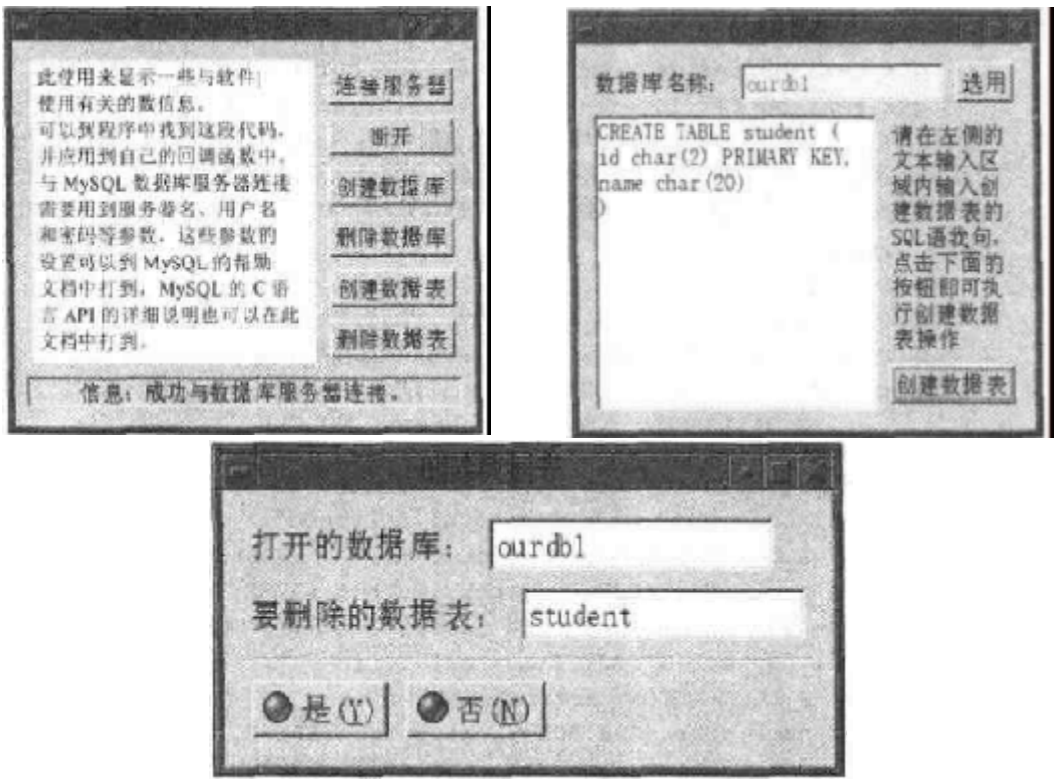

图 9.1 数据库的连接、断开、创建与删除

实例分析

(1) 系统需求

编译此示例的前提是您的Linux系统应安装有MySQL数据库系统软件包和MySQL数据库的C语言开发包(在Linux分发版的光盘中可以找到相关的软件包)，并且在系统运行的服务中选择运行mysqld数据库服务器，如果未运行mysqld数据库服务器，可以在终端命令行中输入：safe-mysqld& 来开启MySQL数据库服务器。

(2) MySQL的C语言API和SQL语言

MySQL数据库系统本身就是由C语言编写的，它的开发包中包含了相当丰富的C语言API函数，这些函数都是以mysql为前缀，还有MySQL中特有的数据类型，下面就本节中用到的加以介绍：

MySQL类型，代表MySQL数据库的句柄(操作对象)，一般用它的指针形式，几乎所有与MySQL相关的操作都用到它。用函数mysql_init来对它进行初始化。用mysql_close函数来关闭释放它。

函数mysql_real_connect用来连接数据库服务器，它有8个参数，第1个参数是上面初始化的MySQL结构；第2个参数是一个IP地址或主机名，这里用的本地主机的名称为“localhost”；第3个参数是用户名称标记，这里用NULL表示当前用户；第4个参数是用户口令，与MySQL系统管理的设置有关，可以为NULL；第5个参数是要选用的数据库名，如指定的话，此数据库被打开；第6个参数指定数据库服务器的端口号，由第2个参数的主机类型决定，这里用宏MySQL_PORT来表示；第7个参数是指定的套接字，一般为NULL；第8个参数通常设为0，也可以用一些宏来表示，详细情况见MySQL的参考手册。此函数如执行成功返回一个MySQL*的数据库连接句柄，如失败返回NULL，据此可以判断是否与数据库服务器连接成功。

函数mysql_query和函数mysql_real_query用来执行SQL语句，其中mysql_query有2个参数，第1个是数据库的连接句柄，第2个参数是执行的SQL语句的字符串指针。函数mysql_real_query功能和前一函数相同，它比前一个函数多一参数，此参数为SQL语句的字符串长度。这两个函数如执行成功则返回0，否则返回非0值，表示出错的原因，本例中并未对返回非0值进行判断，然而多数情况下这是必须的，读者可以在MySQL的C语言API中找到说明。

MySQL数据库支持SQL92标准，但仍有些限制。这里的创建、使用和删除数据库用的都是SQL语句，分别对应CREATE DATABASE、USE和DROP DATABASE语句，删除数据表用的是DROP TABLE语句，这些语句都是动态生成的，不需要用户输入。由于创建数据表的SQL语句CREATE TABLE非常复杂，所以采用直接输入SQL语句来执行的方法，因此读者一定要熟悉SQL语句的使用和MySQL中的数据类型。

(3) 编译和连接

所有的开发MySQL客户端的开发包头文件都安装在/usr/include/mysql目录下，而链接库则安装在/usr/lib/mysql目录下，一般的MySQL客户端程序都要链接其中的mysqlclient动态链接库。

(4) 灵活设置提示信息

本例中用一个观察口容器中加入一文字标签放到窗口的底部作为提示信息，可以用函数 gtk_label_set_text 来改变提示信息，是设置提示信息的一种简单的方法。

此程序中用到了与数据库服务器连接的一系列函数，这在以下的示例中经常用到，读者一定要注意把握，另外有兴趣的读者可以试一下动态生成创建数据表的 SQL 语句 CREATE TABLE。

9.2 向数据表中插入数据

本节示例将介绍如何运用 GTK+2.0 编程动态生成 SQL 中的 INSERT 语句，将数据插入到 MySQL 数据库的数据表中去。

实例说明

在上节示例中我们熟悉了数据库和数据表的创建。在此基础上，本节示例以一个学生成绩表为例，演示如何动态生成 SQL 中的 INSERT 语句，向数据表中插入数据。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/9/
mkdir insert
cd insert
```

创建本节的工作目录，进入此目录开始编程。运行上一节的程序，创建数据库 ourdb1，在数据库 ourdb1 中创建数据表 student，输入以下 SQL 语句：

```
CREATE TABLE student (
id int(2) PRIMARY KEY,
name char(20),
xclass char(20),
yuwen double(5,2),
shuxue double(5,2),
yingyu double(5,2),
zhonghe double(5,2)
)
```

包括编号(id)、姓名(name)、班级(xclass)、语文(yuwen)、数学(shuxue)、英语(yingyu)、综合(zhonghe)等字段。我们的目的就是编程向此数据表中插入数据。

(2) 打开编辑器，输入以下代码，以 insert.c 为文件名保存到当前目录下。

```
/* 向数据表中插入数据 insert.c */
#include <gtk/gtk.h>
#include <mysql.h>
MySQL *myconnect = NULL;
gboolean isclosed = TRUE;
```

```
MySQL_RES * res ;
MySQL_FIELD * fd ;
MySQL_ROW row ;

static GtkWidget *mlabel = NULL;//定义提示信息条
static GtkWidget *c_spin = NULL; //语文
static GtkWidget *m_spin = NULL; //数学
static GtkWidget *e_spin = NULL; //英语
static GtkWidget *a_spin = NULL; //综合
static GtkWidget *name_entry = NULL; //姓名
static GtkWidget *combo = NULL; //班级
static GtkWidget *id_spin = NULL; //编号
void    on_insert (GtkButton *button, gpointer data)
{
    gchar query_buf[4096];
    gdouble chinese,math,english,all;
    const gchar* name;
    const gchar* xclass;
    gint id;

    chinese = gtk_spin_button_get_value(GTK_SPIN_BUTTON(c_spin));
    math = gtk_spin_button_get_value(GTK_SPIN_BUTTON(m_spin));
    english = gtk_spin_button_get_value(GTK_SPIN_BUTTON(e_spin));
    all = gtk_spin_button_get_value(GTK_SPIN_BUTTON(a_spin));
    name = gtk_entry_get_text(GTK_ENTRY(name_entry));
    xclass = gtk_entry_get_text(GTK_COMBO(combo)->entry);
    id = (gint)gtk_spin_button_get_value(GTK_SPIN_BUTTON(id_spin));
    sprintf(query_buf,"INSERT INTO student values
 (\'%d\',\'%s\',\'%s\',\'%.2f\',\'%.2f\',\'%.2f\',\'%.2f\')",
id,name,xclass,chinese,math,english,all);
    if(mysql_query(myconnect,query_buf) == 0)  //执行此SQL语句
    {
        gtk_label_set_text(GTK_LABEL(mlabel),"信息：数据插入成功！");
    }
    else
    {
        gtk_label_set_text(GTK_LABEL(mlabel),"信息：数据插入失败！");
    }
}
GtkWidget* create_combo (void) //创建显示班级名称的下拉列表框
{
    GtkWidget *combo;
    GList *items = NULL;
    items = g_list_append(items,"一年级一班");
    items = g_list_append(items,"一年级二班");
    items = g_list_append(items,"二年级一班");
    items = g_list_append(items,"二年级二班");
    items = g_list_append(items,"三年级一班");
    items = g_list_append(items,"三年级二班");
    combo = gtk_combo_new();
```

```
    gtk_combo_set_popdown_strings(GTK_COMBO(combo),items);
    return combo;
}
gboolean my_connect () //连接
{
    gchar *query_buf = "USE ourdb1";    //打开的同时选用数据库ourdb1
    myconnect = mysql_init(myconnect);
    if(mysql_real_connect(myconnect,"localhost",NULL,NULL,
NULL,MySQL_PORT,NULL,0))
    {
        if(mysql_query(myconnect,query_buf) == 0)
        {
            return TRUE;
        }
        else
        {
            return FALSE;
        }
    }
    else
    {
        myconnect = NULL;
        return FALSE;
    }
}
void my_disconnect()    //断开
{
    mysql_close(myconnect);
}
void    on_delete_event (GtkWidget* widget, GdkEvent* event, gpointer
data)
{
    if(isclosed == FALSE)
    {
        my_disconnect();
    }
    gtk_main_quit();
}
int main (int argc, char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox, *hbox, *viewport;
    GtkWidget *button, *label;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(on_delete_event),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"向数据表中插入数据");
    gtk_window_set_default_size(GTK_WINDOW(window),500,100);
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
```

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

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    label = gtk_label_new("语文：");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    c_spin = gtk_spin_button_new_with_range(0,100,0.5);
    gtk_box_pack_start(GTK_BOX(hbox),c_spin,FALSE,FALSE,5);
    label = gtk_label_new("数学：");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    m_spin = gtk_spin_button_new_with_range(0,100,0.5);
    gtk_box_pack_start(GTK_BOX(hbox),m_spin,FALSE,FALSE,5);
    label = gtk_label_new("英语：");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    e_spin = gtk_spin_button_new_with_range(0,100,0.5);
    gtk_box_pack_start(GTK_BOX(hbox),e_spin,FALSE,FALSE,5);
    label = gtk_label_new("综合：");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    a_spin = gtk_spin_button_new_with_range(0,100,0.5);
    gtk_box_pack_start(GTK_BOX(hbox),a_spin,FALSE,FALSE,5);

    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    label = gtk_label_new("编号:");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    id_spin = gtk_spin_button_new_with_range(1,999,1);
    gtk_box_pack_start(GTK_BOX(hbox),id_spin,FALSE,FALSE,5);
    label = gtk_label_new("班级：");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    combo = create_combo();
    gtk_box_pack_start(GTK_BOX(hbox),combo,FALSE,FALSE,5);
    label = gtk_label_new("姓名：");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    name_entry = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(hbox),name_entry,FALSE,FALSE,5);
    button = gtk_button_new_with_label("插入");
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_insert),NULL);

    viewport = gtk_viewport_new(NULL,NULL);
    gtk_box_pack_start(GTK_BOX(vbox),viewport,FALSE,FALSE,5);
    mlabel = gtk_label_new(NULL);
    gtk_container_add(GTK_CONTAINER(viewport),mlabel);
    if ( my_connect() == FALSE )
    {
        gtk_label_set_text(GTK_LABEL(mlabel),"错误：不能与数据库服务器连
接。");
    }
```

```
    else
    {
        gtk_label_set_text(GTK_LABEL(mlabel),"信息：成功与数据库服务器连
接。");
        isclosed = FALSE;
    }
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
LIB = -I/usr/include/mysql -L/usr/lib/mysql -lmysqlclient
all:
    $(CC) -o insert insert.c $(LIB) 'pkg-config gtk+-2.0 --cflags --libs'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./insert 即可运行此程序，运行结果如图 9.2 所示。

图 9.2 向数据表中插入数据

实例分析

在 MySQL 数据库系统中向数据表插入数据的 SQL 语句 INSERT 有 3 种形式。这里用的是较常用到的最简单的一种即 INSERT INTO 表名 values ('字段值 1', '字段值 2', ...)。这种形式要求在生成的 SQL 语句中带有单引号或双引号，这两种符号不能直接出现在 C 语言的字符串中，所以要用转义字符“\'”或“\"”。

还应该注意的是所有的记录项都应添加好，尤其是编号不能重复，因为它是主键，重复的话会引起数据插入失败。

数据库中不同的数据类型正好对应 GTK+2.0 中的不同的控件来作为表现形式，这样能更好的动态生成 SQL 语句，以便于对数据表进行记录操作。这种插入数据的方法并不是惟一的数据插入方法，也并不是最快的数据插入方法，但这对普通用户来说是一种比较实用和快速插入数据的方法。

9.3 从数据表中选择数据

本节示例将介绍如何运用 MySQL 的 C 语言 API 实现从数据表中的数据选择，并将选择的数据显示到 GTK+2.0 创建的窗口界面中来。

实例说明

上节示例可以向数据表中插入数据，如何将这些数据从数据表中读取出来并显示到窗口中来呢？本节示例以上节创建的数据表和插入的数据为基础，演示如何编程实现将数据表中的数据选择并显示到窗口中的文本显示控件中来。

实现步骤

(1) 打开终端输入下面命令：

```
cd ~ourgtk/11
mkdir Select
cd Select
```

(2) 打开编辑器，输入以下代码，以 select.c 为文件名保存到当前目录下。

```
/* 从数据库中选择数据 select.c */
#include <gtk/gtk.h>
#include <mysql.h>
MySQL *myconnect = NULL;
gboolean isclosed = TRUE;
MySQL_RES * res ;
MySQL_FIELD * fd ;
MySQL_ROW row ;

static GtkWidget *mlabel;
static GtkWidget *text;
GtkTextBuffer *buffer;
static GtkWidget *entry;
GtkTextIter iter;
void    on_select (GtkButton *button, gpointer data)
{
    gchar query_buf[1024];
    gchar info_buf[256];
    const gchar* table;
    gint rows,i,j;
    table = gtk_entry_get_text(GTK_ENTRY(entry));
    sprintf(query_buf,"SELECT * FROM %s",table);
    if(mysql_query(myconnect,query_buf) == 0 )
    {
        gtk_label_set_text(GTK_LABEL(mlabel),"信息：成功选择数据。");
        res = mysql_store_result(myconnect);//保存选择结果
        j = mysql_num_fields(res);//取得字段数
        for(i=0; i<j; i++)
        {
            fd = mysql_fetch_field(res);//取字段相关信息
            gtk_text_buffer_get_end_iter(buffer,&iter);
            gtk_text_buffer_insert(buffer,&iter,fd->name,-1);
            gtk_text_buffer_get_end_iter(buffer,&iter);
            gtk_text_buffer_insert(buffer,&iter,"\t",-1);
```

```
            }
            //以下插入一个换行符
            gtk_text_buffer_get_end_iter(buffer,&iter);
            gtk_text_buffer_insert(buffer,&iter,"\n",-1);
            //以下循环取得表中各行的数据
            while(row = mysql_fetch_row(res))
            {
                for(i=0; i<j; i++)
                {   //根据字段数显示选取结果
                    gtk_text_buffer_get_end_iter(buffer,&iter);
                    gtk_text_buffer_insert(buffer,&iter,row[i],-1);
                    gtk_text_buffer_get_end_iter(buffer,&iter);
                    gtk_text_buffer_insert(buffer,&iter,"\t",-1);
                }
                gtk_text_buffer_get_end_iter(buffer,&iter);//换行
                gtk_text_buffer_insert(buffer,&iter,"\n",-1);
            }
            rows = (gint)mysql_num_rows(res);//取得选取的数据行数
            sprintf(info_buf,"共有 %d 条数据被选择! \n",rows);
            gtk_text_buffer_get_end_iter(buffer,&iter);
            gtk_text_buffer_insert(buffer,&iter,info_buf,-1);
            mysql_free_result(res);//注意，一定要用此函数释放取得的结果
        }
        else
        {
            gtk_label_set_text(GTK_LABEL(mlabel),"信息：执行SQL语句时出错。
");
        }
    }
    gboolean my_connect ()  //连接
    {
        gchar *query_buf = "USE ourdb1";//选用ourdb1数据库
        myconnect = mysql_init(myconnect);
        if(mysql_real_connect(myconnect,"localhost",NULL,NULL,
    NULL,MySQL_PORT,NULL,0))
        {
            if(mysql_query(myconnect,query_buf) == 0 )
                return TRUE;//此种情况说明连接服务器成功，但我们要用的数据库不存在
            else
                return FALSE;
        }
        else
        {
            myconnect = NULL;
            return FALSE;
        }
    }
    void my_disconnect()    //断开
    {
        mysql_close(myconnect);
    }
```

```
void    on_delete_event (GtkWidget* widget, GdkEvent* event, gpointer
data)
{
    if(isclosed == FALSE)
    {
        my_disconnect();
    }
    gtk_main_quit();
}
int main (int argc, char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox, *hbox, *viewport;
    GtkWidget *button, *label;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(on_delete_event),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"从数据表中读取数据");
    gtk_window_set_default_size(GTK_WINDOW(window),500,100);
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    label = gtk_label_new("数据表名: ");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    entry = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(hbox),entry,FALSE,FALSE,5);
    button = gtk_button_new_with_label("选择数据");
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_select),NULL);

    text = gtk_text_view_new();
    gtk_box_pack_start(GTK_BOX(vbox),text,TRUE,TRUE,5);
    buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(text));
    viewport = gtk_viewport_new(NULL,NULL);
    gtk_box_pack_start(GTK_BOX(vbox),viewport,FALSE,FALSE,5);
    mlabel = gtk_label_new(NULL);
    gtk_container_add(GTK_CONTAINER(viewport),mlabel);

    if ( my_connect() == FALSE )
    {
        gtk_label_set_text(GTK_LABEL(mlabel),"错误: 不能与数据库服务器连
接。");
    }
    else
    {
```

```
        gtk_label_set_text(GTK_LABEL(mlabel),"信息：成功与数据库服务器连
接。");
        isclosed = FALSE;
    }
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
LIB = -I/usr/include/mysql -L/usr/lib/mysql -lmysqlclient
all:
    $(CC) -o select select.c $(LIB) 'pkg-config gtk+-2.0 --cflags --libs'
```

(4) 在终端中执行 make 命令开始编译：

(5) 编译结束后，执行命令./select 即可运行此程序，运行结果如图 9.3 所示。

图 9.3 从数据表中选取数据

实例分析

(1) 执行 SELECT 语句

MySQL 中的 SELECT 语句有多种形式。这里只用到了最简单的“SELECT * FROM 数据表名”形式，即将表中的所有数据选取出来。同样用 mysql_query 函数来执行此语句，由于它是有返回结果的，所以还得用函数 mysql_store_result 来取得返回结果，此函数返回一个 MySQL_RES 型指针，表示返回的结果集，在使用完此选择结果后，一定要用函数 mysql_free_result 来释放此结果集。

(2) 取得字段名

取得选取的数据结果后，可以用函数 mysql_num_fields 取得此结果集的字段数，用函数 mysql_fetch_field 取得字段的相关信息。此函数的返回类型为 MySQL_FIELD 型指针，这个与字段信息相关的结构中包含许多成员，只用了 name 项，即字段的名称，更详细的信息可见 MySQL 的 C 语言 API 参考手册；可以根据字段数来依次取得字段的相关信息。

(3) 取得数据行

取得选取的数据结果后，可以用函数 mysql_fetch_row 来取得结果集中的各数据行，它返回 MySQL_ROW 结构，这是一个表示结果集中一个数据行的类型安全的数组(可以保存各种类型的数据)，可以根据字段数，依次保存或显示各字段的值，直到无行可取。用函数 mysql_num_rows 可以取得结果集中的行数，这里将其结果强制转换为整型。

从数据表中选取数据是一个比较繁琐的过程，涉及到了数据表的字段数、字段类型、数据集的保存与释放等，使用者一定要认真分析。此示例将选取的所有数据都显示到文本显示控件中了，这样做比较易于实现，读者还可以用其他方法来表示选择的数据。

9.4　文 档 管 理

本节示例将介绍如何用 GTK+2.0 控件结合 MySQL 数据库的 C 语言 API 来操作 MySQL 数据库中的 BLOB 类型数据，将其保存到数据库中，或从数据库中读取并显示出来。

实例说明

本示例用 MySQL 数据库的 BLOB 型数据功能和 GTK+2.0 的文本显示控件相结合，创建了这样一个管理工具，对文档管理进行了一次尝试。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/9
mkdir document
cd document
```

创建本节工作目录，进入目录开始编程。首先运行本章第一节中的创建数据库示例，创建 ourdb3 数据库，创建数据表，选用 ourdb3 数据库，输入以下 SQL 语句：

```
CREATE TABLE document(
id int(2) PRIMARY KEY,
title varchar(100),
textfile blob
)
```

创建一个简单的数据表，名为 document，包括 3 个字段，分别是标号(id，整型，主键)，文档标题(title，变长字符串)，文档内容(textfile，BLOB 型)。

(2) 打开编辑器，输入以下代码，以 blob.c 为文件名保存到当前目录下。

```
/* 文档管理 blob.c */
#include <gtk/gtk.h>
#include <mysql.h>
#include <stdio.h>
#include <sys/stat.h>   //加入有关文件信息操作的包含头文件
#include <unistd.h>
//定义文字格式转换宏，将本地代码格式转换为UTF8格式
#define UTF8(str) g_locale_to_utf8(str,-1,NULL,NULL,NULL)
MySQL *myconnect = NULL;
gboolean isclosed = TRUE;
static GtkWidget *mlabel = NULL;//定义提示信息条
static GtkWidget *id_spin ;//文档号
static GtkWidget *title_entry; //标题
void    create_view_win (gchar *id, gchar*astitle,gchar *buf)
```

```
{
    GtkWidget *window, *swin, *view;
    GtkTextBuffer *buffer;
    GtkTextIter iter;
    gchar title[256];
    sprintf(title,"标记: %s -- %s",id,astitle);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),title);
    gtk_window_set_default_size(GTK_WINDOW(window),500,100);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_widget_destroy),window);

    swin = gtk_scrolled_window_new(NULL,NULL);
    gtk_container_add(GTK_CONTAINER(window),swin);
    gtk_scrolled_window_set_policy(GTK_SCROLLED_WINDOW(swin),
                GTK_POLICY_AUTOMATIC,GTK_POLICY_AUTOMATIC);
    view = gtk_text_view_new();
    buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(view));
    gtk_text_buffer_get_end_iter(buffer,&iter);
    if(g_utf8_validate(buf,-1,NULL))
        gtk_text_buffer_insert(buffer,&iter,buf,-1);
    else
        gtk_text_buffer_insert(buffer,&iter,UTF8(buf),-1);
    gtk_container_add(GTK_CONTAINER(swin),view);
    gtk_widget_show_all(window);
}
void    on_view (GtkButton *button, gpointer data)
{
    gchar query_buf[1024];
    gint id;
    MySQL_RES * res ;
    MySQL_ROW row ;
    id = (gint)gtk_spin_button_get_value(GTK_SPIN_BUTTON(id_spin));
    sprintf(query_buf, "SELECT * FROM document WHERE id = %d ",id);

    if(mysql_query(myconnect,query_buf) == 0)  //执行此SQL语句
    {
        gtk_label_set_text(GTK_LABEL(mlabel),"信息: 数据选择成功! ");
    }
    else
    {
        gtk_label_set_text(GTK_LABEL(mlabel),"信息: 数据选择失败! ");
    }
    res = mysql_store_result(myconnect);
    if(mysql_num_rows(res) <= 0)
    {
        gtk_label_set_text(GTK_LABEL(mlabel),"信息: 未发现附合条件的数据。
");
        return;
    }
    row = mysql_fetch_row(res);
```

```
    gtk_entry_set_text(GTK_ENTRY(title_entry),row[1]);
    create_view_win(row[0],row[1],row[2]);
    mysql_free_result(res);
}

gboolean insert_file ( gchar *filename )
{
    gchar *query_buf,*buf,*tobuf;
    const gchar* title;
    gint id;
    FILE *fp;
    struct stat pstat;//文件状态结构
    gulong length,tolen;
    if(stat(filename,&pstat) == -1)
    {
        g_print("file stat error \n");
        return FALSE;
    }

    buf = g_malloc(pstat.st_size);
    fp = fopen(filename,"r");
    if(fp == NULL)
    {
        g_print("file open error\n");
        return FALSE;
    }

    length = fread(buf,sizeof(gchar),pstat.st_size,fp);
    tobuf = g_malloc(length*2+1);//分配内存保存转义字符串
    tolen = mysql_escape_string(tobuf,buf,length);
    query_buf = g_malloc(tolen+1024);
    title = gtk_entry_get_text(GTK_ENTRY(title_entry));
    id = (gint)gtk_spin_button_get_value(GTK_SPIN_BUTTON(id_spin));
    sprintf(query_buf,"INSERT INTO document
 VALUES ( \'%d\',\'%s\',\'%s\')",id,title,tobuf);
    if(mysql_query(myconnect,query_buf) == 0)  //执行此SQL语句
    {
        gtk_label_set_text(GTK_LABEL(mlabel),"信息: 数据插入成功! ");
    }
    else
    {
        gtk_label_set_text(GTK_LABEL(mlabel),"信息: 数据插入失败! ");
    }

    g_free(buf);
    g_free(query_buf);
    g_free(tobuf);
    fclose(fp);
}
void    on_select_ok (GtkButton* button, gpointer data)
{
```

```
    gchar *name;
    name = gtk_file_selection_get_filename(GTK_FILE_SELECTION(data));
    insert_file(name);
}
void    create_select_file (gchar* title)
{
    GtkWidget *dialog;
    dialog = gtk_file_selection_new(title);
    gtk_window_set_position(GTK_WINDOW(dialog),GTK_WIN_POS_CENTER);
    g_signal_connect(G_OBJECT(dialog),"delete_event",
            G_CALLBACK(gtk_widget_destroy),dialog);
    g_signal_connect(G_OBJECT(GTK_FILE_SELECTION(dialog)->ok_button)
,
"clicked", G_CALLBACK(on_select_ok), dialog);
    g_signal_connect_swapped(G_OBJECT(GTK_FILE_SELECTION(
dialog)->ok_button),"clicked",
G_CALLBACK(gtk_widget_destroy),dialog);
    g_signal_connect(G_OBJECT(GTK_FILE_SELECTION(dialog)->cancel_but
ton),
            "clicked",G_CALLBACK(gtk_widget_destroy),dialog);
    gtk_widget_show(dialog);
}
void    on_insert (GtkButton *button, gpointer data)
{
    create_select_file("请选择一个文本文件：");
}
gboolean my_connect ()  //连接
{
    gchar *query_buf = "USE ourdb3";    //打开的同时选用数据库ourdb2
    myconnect = mysql_init(myconnect);
    if(mysql_real_connect(myconnect,"localhost",
NULL,NULL,NULL,MySQL_PORT,NULL,0))
    {
        if(mysql_query(myconnect,query_buf) == 0)
        {
            return TRUE;
        }
        else
        {
            return FALSE;
        }
    }
    else
    {
        myconnect = NULL;
        return FALSE;
    }
}
void my_disconnect()    //断开
{
    mysql_close(myconnect);
```

```
}
void    on_delete_event (GtkWidget* widget, GdkEvent* event, gpointer
data)
{
    if(isclosed == FALSE)
    {
        my_disconnect();
    }
    gtk_main_quit();
}
int main (int argc, char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox, *hbox, *viewport;
    GtkWidget *button, *label;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(on_delete_event),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"文档管理");
    //gtk_window_set_default_size(GTK_WINDOW(window),500,100);
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);

    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    label = gtk_label_new("编号:");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    id_spin = gtk_spin_button_new_with_range(1,999,1);
    gtk_box_pack_start(GTK_BOX(hbox),id_spin,FALSE,FALSE,5);
    label = gtk_label_new("标题:");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    title_entry = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(hbox),title_entry,FALSE,FALSE,5);
    button = gtk_button_new_with_label("查看...");
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_view),NULL);
    button = gtk_button_new_with_label("插入...");
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_insert),NULL);

    viewport = gtk_viewport_new(NULL,NULL);
    gtk_box_pack_start(GTK_BOX(vbox),viewport,FALSE,FALSE,5);
    mlabel = gtk_label_new(NULL);
    gtk_container_add(GTK_CONTAINER(viewport),mlabel);
    if ( my_connect() == FALSE )
    {
```

```
        gtk_label_set_text(GTK_LABEL(mlabel),"错误：不能与数据库服务器连
接。");
    }
    else
    {
        gtk_label_set_text(GTK_LABEL(mlabel),"信息：成功与数据库服务器连
接。");
        isclosed = FALSE;
    }
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
LIB = -I/usr/include/mysql -L/usr/lib/mysql -lmysqlclient
all:
    $(CC) -o blob blob.c $(LIB) 'pkg-config gtk+-2.0 --cflags --libs'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./blob 即可运行此程序，运行结果如图 9.4 所示。

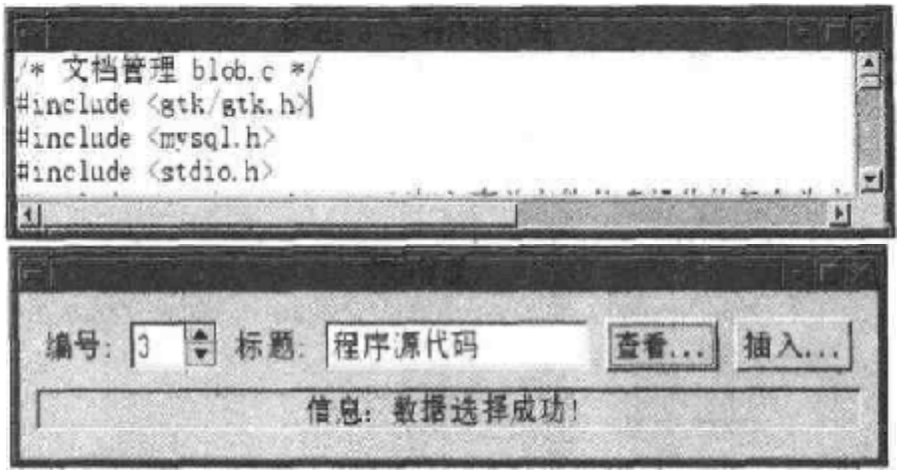

图 9.4 文档管理

实例分析

(1) 插入与查看的过程

此示例运行时先设定好编号，再输入文档标题。单击插入按钮，会显示一个文件选择对话框，选择一个文本文件，单击确认，程序会打开此文件并读出文件内容，生成 INSERT 语句，并运行此语句，将此文档的内容插入到数据库中。如果数据库中已插入了相应的文档，可以选择编号，单击查看按钮，程序会生成“SELECT * FROM document WHERE id=编号”语句，并运行此语句，保存返回结果，生成查看窗口，将结果显示到查看窗口的文本显示控件中去，这样就会看到保存在数据库中的文档了。

(2) 生成超长的 SQL 语句

在生成插入文档的 INSERT 语句时，由于语句中加入了文档的内容，使这个 INSERT 语句比文档还要长，我们用到了 GTK+2.0 中的内存分配函数 g_malloc 来解决这一问题，当然用完后别忘了用函数 g_free 来释放这部分内存。

(3) 转义字符

由于有些文档中有非显示的控制字符，所以必须用 mysql_escape_string 函数或 mysql_real_escape_string 函数将文档中的 NUL 等字符转换为‘\0’等转义字符保存到 SQL 语句中，以使程序能理解，这样必须分配内存，而且分配的内存的长度应该是原字符串长度的两倍再加 1，以保证能保存下这些字符。还有一种情况是文档用的是本地代码，而非 UTF8 格式，这里用 g_local_to_utf8 函数定义的宏 UTF8 将文档统一转换为 UTF8 格式，以使它们能在文本显示控件中显示出来。

要注意到虽然是 BLOB 数据类型，但这里只对文本文档进行了处理，使它能保存文本文档到 MySQL 数据库，并从 MySQL 数据中选取显示出来。而图像数据与文本文档是截然不同的，加之图像格式和存贮方式的不同，使之实现起来并不轻松，有兴趣的读者不妨一试。

本章示例可以说是用 GTK+2.0 为 MySQL 数据库做一个可视化前端的尝试，所有示例都用到了 MySQL 的 C 语言 API 函数和 SQL 语言的常用语句，读者对这些函数一定要有深刻的理解，另外灵活运用 GTK+2.0 控件做应用程序的界面，也使您在协调控件与应用程序功能的关系上得到了良好的训练。

第 10 章　网 络 编 程

本章重点：

网络编程是 Linux 功能的强项，但 GTK+2.0 是一个图形界面开发工具，并未提供网络编程功能。所以最好的办法是用 GTK+2.0 创建界面结合 Linux 网络编程，由于 Linux 的网络应用和 GTK+2.0 都是采用 C 语言编程的，这为两者的结合提供了良好的前提条件。本章将向读者介绍如何为 Linux 的命令行网络工具做一个可视化前端，如何编写简单的 Linux 服务程序和用 GTK+2.0 编写这些服务的客户端。

本章主要内容：

- 为命令行工具 mail 做一个可视化前端
- 编写简单的 ECHO 服务器
- 编写 ECHO 服务器的客户端
- 编写简单的多人聊天服务器
- 编写聊天服务器的客户端

10.1　简单的发 E-mail 的软件

本节示例将介绍如何在 GTK+2.0 编程中启动邮件程序发送邮件，从而实现一些 Linux 常用命令的可视化前端。

实例说明

在 Linux 上有一个最古老的电子邮件收发工具—— mail 命令行工具，本节示例就是运用 GTK+2.0 开发应用程序界面，调用此工具来为用户发送邮件，目的就是为 mail 工具的一个简单应用编写可视化前端。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk
mkdir 10
cd 10
mkdir amail
cd amail
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 about.c 为文件名保存到当前目录下。

```
/* amail.c 简单的邮件发送程序 */
```

```
#include <gtk/gtk.h>
#include <stdio.h>
static GtkWidget *from_entry, *to_entry;
static GtkWidget *sub_entry, *mlabel;
static GtkWidget *text;
GtkTextBuffer *buffer;
void    create_message_dialog  (GtkMessageType type, gchar* message)
{
    GtkWidget* dialogx;
    dialogx = gtk_message_dialog_new(NULL,
            GTK_DIALOG_MODAL | GTK_DIALOG_DESTROY_WITH_PARENT,
            type ,
            GTK_BUTTONS_OK,
            message);
    gtk_dialog_run(GTK_DIALOG(dialogx));
    gtk_widget_destroy(dialogx);
}
gboolean send_mail (gchar* to, gchar *sub, gchar* text)
{
    FILE *fp;
    char command[1024];
    fp = fopen("./letter","w");
    if(fp == NULL)
    {
        gtk_label_set_text(GTK_LABEL(mlabel),"提示信息：文件操作时出错!
");
        return FALSE;
    }
    fputs(text,fp);
    fclose(fp);
    sprintf(command,"mail %s -s %s < letter",to,sub);
    system(command);
    return TRUE;
}
void    on_send (GtkButton* button, gpointer data)
{
    GtkTextIter start,end;
    const char *to, *sub;
    char* buf;
    to = gtk_entry_get_text(GTK_ENTRY(to_entry));
    sub = gtk_entry_get_text(GTK_ENTRY(sub_entry));
    gtk_text_buffer_get_start_iter(buffer,&start);
    gtk_text_buffer_get_end_iter(buffer,&end);
    buf = gtk_text_buffer_get_text(buffer,&start,&end,FALSE);

    if(send_mail(to,sub,buf))
        create_message_dialog(GTK_MESSAGE_INFO,"邮件发送成功! ");
    else
        create_message_dialog(GTK_MESSAGE_ERROR,"邮件发送时出错! ");
}
```

```
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox, *hbox1, *hbox2;
    GtkWidget *label, *button, *view;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"发邮件");
    gtk_window_set_default_size(GTK_WINDOW(window),500,100);
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    hbox1 = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox1,FALSE,FALSE,5);
    label = gtk_label_new("发到: ");
    gtk_box_pack_start(GTK_BOX(hbox1),label,FALSE,FALSE,5);
    to_entry = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(hbox1),to_entry,TRUE,TRUE,5);
    button = gtk_button_new_with_label("发送");
    gtk_box_pack_start(GTK_BOX(hbox1),button,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_send),NULL);
    hbox2 = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox2,FALSE,FALSE,5);
    label = gtk_label_new("主题: ");
    gtk_box_pack_start(GTK_BOX(hbox2),label,FALSE,FALSE,5);
    sub_entry = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(hbox2),sub_entry,TRUE,TRUE,5);
    view = gtk_scrolled_window_new(NULL,NULL);
    gtk_scrolled_window_set_policy(GTK_SCROLLED_WINDOW(view),
        GTK_POLICY_AUTOMATIC,GTK_POLICY_AUTOMATIC);
    text = gtk_text_view_new();
    buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(text));
    gtk_container_add(GTK_CONTAINER(view),text);
    gtk_box_pack_start(GTK_BOX(vbox),view,TRUE,TRUE,5);

    view = gtk_viewport_new(NULL,NULL);
    gtk_box_pack_start(GTK_BOX(vbox),view,FALSE,FALSE,5);
    mlabel = gtk_label_new("提示信息: 填写好以上内容后按发送按钮");
    gtk_container_add(GTK_CONTAINER(view),mlabel);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o amail amail.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./amail 即可运行此程序，运行结果如图 10.1 所示。

图 10.1　简单的发邮件程序

实例分析

(1) 生成临时文件

此示例运行时在当前目录下生成一个名为 letter 的文本文件，将邮件的内容保存在此文件中。再一次发送时则会更新此文件的内容。

(2) 生成命令行

单击发送按钮后，实际上是生成了一个类似如下的命令行：mail 地址 -s 主题 < letter。此命令行主要是向指定地址发送指定主题的邮件，以 letter 文件为邮件内容。

(3) 执行命令

以上命令的执行采用的是 system 系统调用，这个方法比较简单。对 Linux 系统调用较熟悉的读者可以用 exec 系统调用，这样可以检查一下执行的结果。

此程序只是一个简单的演示，而且只能给同一主机上的其他用户发信，所以实用性并不强，但读者完全可以依此来为其他的命令行工具开发可视化前端。

10.2　简单的 ECHO 服务器

本节示例将介绍如何利用 GLIB 和 Linux 的网络功能编程实现一个简单的 ECHO 服务器。

实例说明

服务应用的开发是 Linux 系统最具吸引力的强项之一，套接字(socket)编程则是服务应用开发的关键和核心，但这些并不在 GTK+2.0 编程的范畴，好在 GLIB 在底层提供了对它们的支持，本示例就运用套接字编程，创建了一个 ECHO 服务器，它的功能就是把用户发给它的数据原原本本地传递给用户。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/10
mkdir echo
cd echo
mkdir server
cd server
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 echo_server.c 为文件名保存到当前目录下。

```
/* 服务器端 echo_server */
#include <glib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>
#include <netdb.h>
#include <netinet/in.h>
//定义端口号
#define OURPORT 8088
//当有用户连接时的服务子进程
void    do_service(gint sd)
{
    gchar buf[1024];
    while(read(sd,buf,1024) != -1) //从用户读取数据
    {
        write(sd,buf,1024);//向用户发送数据
    }
}

int main(int argc, char* argv[])
{
    gint sd, newsd; //定义套接字句柄
    struct sockaddr_in *sin;    //套接字地址结构
    gint slen;
    gchar buf[1024];
    sd = socket(AF_INET,SOCK_STREAM,0);//创建套接字
    if(sd == -1)
    {
        g_print("create socket error!\n");//创建时出错
        return -1;
    }
    sin = g_new(struct sockaddr_in,1);//为套接字地址分配内存
    sin->sin_family = AF_INET; //套接字类型
    sin->sin_port = OURPORT;    //端口号
```

```
    slen = sizeof(struct sockaddr_in);
    if(bind(sd,sin,slen)<0) //向指定端口绑定
    {
        g_print("bind error!\n");//绑定时出错
        return  1;
    }
    if(listen(sd,8)<0)  //监听
    {
        g_print("listen error!\n");//监听出错
        return -1;
    }
    for(;;) //死循环，等待用户连接
    {
        newsd = accept(sd,sin,&slen);//取得用户连接的套接字
        if(newsd == -1)
        {
            g_print("accept error!\n");//连接出错
            break;
        }
        switch(fork())  //产生子进程
        {
            case 0:
                do_service(newsd);//执行服务
                break;
            case -1:
                g_print("fork error! \n");//产生了进程时出错
                break;
        }
    }
    close(sd);  //关闭套接字
    g_free(sin);    //释放内存
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) echo_server.c -o echo_server 'pkg-config --cflags --libs
glib-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./echo_server 即可运行此程序。

注意： 此程序并不显示任何结果。

实例分析

(1) 服务器的运行过程

此 ECHO 服务器的运行过程如图 10.2 所示：

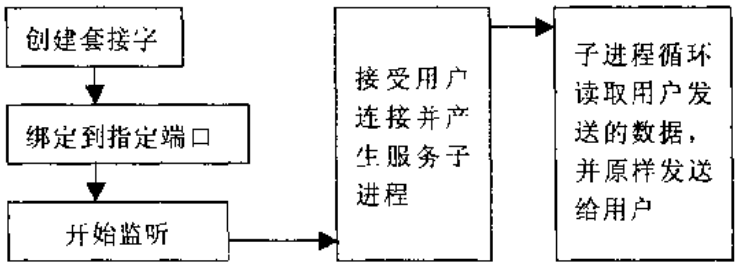

图 10.2 ECHO 服务器的运行过程

(2) 相关的函数

函数 socket 用来生成套接字句柄，函数 bind 用来向指定的端口绑定套接字服务，函数 listen 用来监听客户的最大数量，函数 accept 用来检查用户连接，如果有用户连接，则返回新的套接字句柄，产生的子进程就利用此套接字句柄与客户交流。

(3) 进程

在 Linux 系统中用 fork 系统调用来产生子进程，通过以上的运行过程我们可以看出，每有一个用户连接到服务器，服务器就会产生一个子进程与用户交流。这样当系统上的用户较多时，便会消耗大量的系统资源，所以上述方法只是演示服务应用，并不适于真正的系统服务。

由于程序中用到大量与 GLIB 相关的函数和定义，所以在编译时用到配置文件加-glib 选项，如：'pkg-config --cflags --libs glib-2.0'。

由于 ECHO 服务器陷入了死循环，所以最好的办法是在后台运行或做成守护进程。使程序能长时间在内存中运行。关于守护进程程序的编写方法和以上函数的详细说明可以参考相关的 Linux 网络编程教程。

10.3 简单的 ECHO 客户端

本节示例将介绍如何实现网络应用客户端和 GTK+2.0 界面编程的简单结合。

实例说明

如何用 GTK+2.0 为上节编写的 ECHO 服务器编写客户端，是本节的主要研究课题。首先是要知道 ECHO 服务的端口号，还要知道服务的类型，然后创建套接字，利用套接字向服务器连接，连接后向服务器发送和接收来自服务器的数据，并将来自服务器的数据显示到窗口的文本显示控件当中来。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/10/echo
mkdir client
cd client
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 echo_client.c 为文件名保存到当前目录下。

```
/* 客户端测试 client.c */
#include <gtk/gtk.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
//定义端口号
#define OURPORT 8088

gint sd;
struct sockaddr_in s_in;
gchar username[64];
gchar buf[1024];    //读缓冲区
gchar get_buf[1048];    //写缓冲区
gboolean isconnected = FALSE;

static GtkWidget *text;
static GtkTextBuffer *buffer;
static GtkWidget *message_entry;
gboolean do_connect(void)
{
    GtkTextIter iter;
    gint slen;
    sd = socket(AF_INET,SOCK_STREAM,0);//创建
    if(sd < 0)
    {
        gtk_text_buffer_get_end_iter(buffer,&iter);
        gtk_text_buffer_insert(buffer,&iter,"打开套接字时出错！\n",-1);
        return FALSE;
    }
    s_in.sin_family = AF_INET;
    s_in.sin_port = OURPORT;
    slen = sizeof(s_in);
    if(connect(sd,&s_in,slen) < 0) //连接
    {
        gtk_text_buffer_get_end_iter(buffer,&iter);
        gtk_text_buffer_insert(buffer,&iter,"连接服务器时出错！\n",-1);
        return FALSE;
    }
    else
    {
        gtk_text_buffer_get_end_iter(buffer,&iter);
        gtk_text_buffer_insert(buffer,&iter,username,-1);
        gtk_text_buffer_get_end_iter(buffer,&iter);
        gtk_text_buffer_insert(buffer,&iter,"\n成功与服务器连
接....\n",-1);
        isconnected = TRUE;
        return TRUE;
    }
```

```
}

void    on_send (GtkButton* button, gpointer data)
{
    const char* message;
    GtkTextIter iter;
    if(isconnected==FALSE) return;
    message = gtk_entry_get_text(GTK_ENTRY(message_entry));
    sprintf(buf,"%s\n",message);
    write(sd,buf,1024);
    gtk_entry_set_text(GTK_ENTRY(message_entry),"");
    //
    read(sd,buf,1024);
    sprintf(get_buf,"%s",buf);
    gtk_text_buffer_get_end_iter(buffer,&iter);
    gtk_text_buffer_insert(buffer,&iter,get_buf,-1);
}
void    on_delete_event (GtkWidget *widget, GdkEvent* event, gpointer
data)
{
    close(sd);//关闭
    gtk_main_quit();
}
int main    (int argc, char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox, *hbox, *button, *label, *view;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"ECHO客户端");
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(on_delete_event),NULL);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    label = gtk_label_new("下面显示的内容均来自服务器");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    view = gtk_scrolled_window_new(NULL,NULL);
    gtk_scrolled_window_set_policy(GTK_SCROLLED_WINDOW(view),
        GTK_POLICY_AUTOMATIC,GTK_POLICY_AUTOMATIC);
    text = gtk_text_view_new();
    gtk_box_pack_start(GTK_BOX(vbox),view,TRUE,TRUE,5);
    gtk_container_add(GTK_CONTAINER(view),text);
    buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(text));

    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    label = gtk_label_new("输入消息: ");
```

```
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    message_entry = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(hbox),message_entry,FALSE,FALSE,5);
    button = gtk_button_new_with_label("发送");
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_send),NULL);
    do_connect();
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) echo_client.c -o echo_client 'pkg-config --cflags --libs
gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./echo_client 即可运行此程序，运行结果如图 10.3 所示。

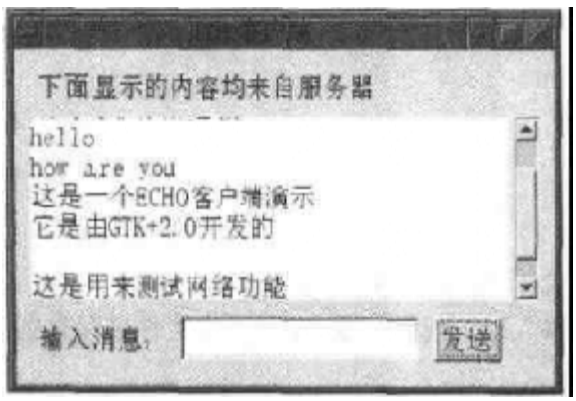

图 10.3　简单的 ECHO 客户端

实例分析

(1) 客户端的工作过程

首先是创建套接字，创建套接字后设置服务类型和端口号，再利用 connect 函数向服务器请求连接，此时如服务器已运行，则显示连接成功，否则显示连接失败。

(2) 发送和接收消息

当连接成功后，我们可以用 read 系统调用来读取来自服务器的数据，同时也可以用 write 系统调用向服务器发送数据。本示例用一个单行录入控件和一个按钮来做发送工作，当单击发送按钮后，程序向服务器发送单行录入控件中输入的数据，清除单行录入控件中的内容，然后接收来服务器的数据，并将其显示到文本显示控件中来。这样我们可以看出文本显示控件中的数据是我们发向服务器的数据通过服务器转发过来的，而不是直接输入到文本显示控件中去的。

(3) 调试

客户端和服务器的编程是应该同时进行编译和调试的，根据两者的运行情况分析问题的所在，好进一步调试和修改。

至此我们完成了简单的 ECHO 服务器和客户端的程序编写，这只是我们利用 GTK+2.0 来进行网络编程的一个尝试，有兴趣的读者可以进一步改进这个程序。

10.4 多人聊天服务器

本节示例将介绍如何利用 GLIB 线程功能和 Linux 的网络编程功能实现服务器向多用户同时发送数据的能力。

实例说明

多人聊天服务器与 ECHO 服务器的不同之处是服务器要向与服务器连接的所有客户发送每一用户发送给服务器的数据，本节示例对此功能进行了尝试。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/10
mkdir meeting
cd meeting
mkdir server
cd server
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 server.c 为文件名保存到当前目录下。

```
/* 服务器端 server.c */
#include <glib.h>
#include <stdio.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/time.h>
#include <unistd.h>
#include <netdb.h>
#include <netinet/in.h>

#define OURPORT 8088
#define MAX_USERS 8
//定义用户数据结构
struct  client {
    gint sd;
    gboolean in_use;
    gchar name[64];
```

```
    gchar buf[1024];
};
typedef struct _client client;
//定义用户数据区
client user[MAX_USERS];
//定义服务线程
void    do_service (gpointer id)
{
    gint j;
    char tobuf[1024];

    while(read(user[GPOINTER_TO_INT(id)].sd,
            user[GPOINTER_TO_INT(id)].buf,1024)!=-1)
    {
        sprintf(tobuf,"%s: %s\n",user[GPOINTER_TO_INT(id)].name,
                    user[GPOINTER_TO_INT(id)].buf);
        for(j=0; j<MAX_USERS; j++)
        {
            if(user[j].in_use)
            {
                write(user[j].sd,tobuf,1024);
                g_print("%s",tobuf);
            }
        }
    }
    //
    user[GPOINTER_TO_INT(id)].in_use = FALSE;
    close(user[GPOINTER_TO_INT(id)].sd);
    //exit(0);
}

int main(int argc, char* argv[])
{
    gint sd, newsd;
    struct sockaddr_in *sin;
    gint slen;
    gint count = 0;
    gint flags;
    gchar buf[1024];
    gchar tobuf[1024];
    gint length,i,j;

    if(!g_thread_supported())
        g_thread_init(NULL);
    else
        g_print("thread not supported\n");

    sd = socket(AF_INET,SOCK_STREAM,0);
    if(sd == -1)
    {
        g_print("create socket error!\n");
```

```
    return -1;
}

sin = g_new(struct sockaddr_in,1);
sin->sin_family = AF_INET;
sin->sin_port = CURPORT;
slen = sizeof(struct sockaddr_in);

if(bind(sd,sin,slen)<0)
{
    g_print("bind error!\n");
    return -1;
}

if(listen(sd,8)<0)
{
    g_print("listen error!\n");
    return -1;
}

for(i=0; i<MAX_USERS; i++)
    user[i].in_use = FALSE;

flags = fcntl(sd,F_GETFL);
fcntl(sd,F_SETFL,flags&~O_NDELAY);

for(;;)
{
    newsd = accept(sd,sin,&slen);
    if(newsd == -1)
    {
        g_print("accept error!\n");
        break;
    }
    else
    {
        if(count >= MAX_USERS)
        {
            sprintf(buf,"用户数量过多服务器不能连接。\n");
            write(newsd,buf,1024);
            close(newsd);
        }
        else
        {
            flags = fcntl(user[i].sd,F_GETFL);
            fcntl(user[i].sd,F_SETFL,O_NONBLOCK);
            user[count].sd = newsd;
            user[count].in_use = TRUE;
            read(newsd,user[count].name,64);
            //创建为用户服务的线程
            g_thread_create((GThreadFunc)do_service,
```

```
                    (gpointer)count,TRUE,NULL);
                count++;
            }
        }
    }//for(;;)

    close(sd);
    g_free(sin);
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o server server.c 'pkg-config --cflags --libs glib-2.0
gthread-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./server 即可运行此程序。

注意： 此程序运行后不产生任何结果。

实例分析

(1) 进程与线程

此示例与 ECHO 服务器的最大不同之处是 ECHO 服务器采用产生新的子进程来与用户交流，而此多人聊天服务器采用创建新的线程来与客户交流。子进程不能直接与父进程共享数据，而且占用系统资源较多，优点是不同客户的需求可以单独处理；线程不能做为单独进程来使用，优点是占用系统资源较少，可以与父进程共享数据；多人聊天服务器需要多用户间的数据共享，所以我们采用线程。有关线程的功能的使用可以参考第 11 章中的“使用线程”一节。

(2) 定义用户数据结构

由于有多个用户同时交互数据，所以要为用户定义数据结构类型，此结构中应包括用户的套接字句柄、用户名、用户是否连接和用户发来的数据。

(3) 同时向多用户发送信息

虽然是多用户聊天，但我们还是应该定义最大用户数，以便于管理和维护。用户连接后在创建的线程中循环检查其他用户是否连接，如果连接则向其发送数据，如此即实现了一个人发送的信息其他多人能同时接收到。此程序的运行过程如图 10.4 所示。

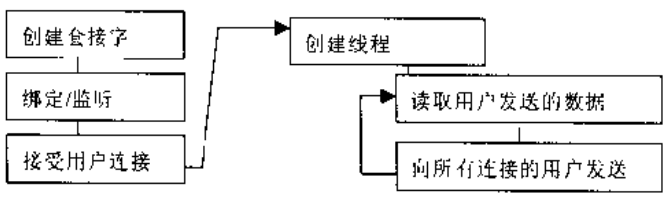

图 10.4　多人聊天室服务器的运行过程

此多人聊天服务器的功能并不完善，如未添加用户注册功能，用户的密码设置，单独

与某人聊天的功能等，但做为一个聊天室的雏型对读者进一步深入 Linux 网络编程可以起到引导的做用。

10.5 多人聊天服务器的客户端

本节示例将介绍如何编程实现登录到上节编写的多人聊天服务器，能随时接收来自服务器的数据。

实例说明

多人聊天室的客户端较 ECHO 的客户端应多出以下功能，客户端运行时不自动登录，需要用户输入用户名后再登录，当用户不输入名称时以“guest”用户名登录，能随时接收来自服务器的数据并显示出来。此示例对上述功能进行了尝试。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/10/meeting
mkdir client
cd client
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 client.c 为文件名保存到当前目录下。

```
/* 客户端测试 client.c */
#include <gtk/gtk.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

#define OURPORT 8088    //定义端口号

gint sd;    //套接字句柄
struct sockaddr_in s_in;    //套接字数据结构
gchar username[64];    //用户名
gchar buf[1024];    //写缓冲区
gchar get_buf[1048];    //读缓冲区
gboolean isconnected = FALSE;  //定义逻辑值表示是否连接

static GtkWidget *text;
static GtkTextBuffer *buffer;  //显示对话内容的文本显示缓冲区
static GtkWidget *message_entry;    //显示输入消息的单行录入控件
static GtkWidget *name_entry;    //输入用户名的单行录入控件
static GtkWidget *login_button;    //登录按钮
//线程要执行的函数，读取来自服务器的数据
void    get_message(void)
```

```
{
    GtkTextIter iter;
    gchar get_buf[1024];
    gchar buf[1024];
    while(read(sd,buf,1024) != -1) //只要读取数据成功就循环执行
    {
        sprintf(get_buf,"%s",buf);
        gdk_threads_enter();    //进入
        gtk_text_buffer_get_end_iter(buffer,&iter);
        gtk_text_buffer_insert(buffer,&iter,get_buf,-1);//显示读取的
数据
        gdk_threads_leave();    //离开
    }
}

void    on_destroy(GtkWidget *widget, GdkEvent *event, gpointer data)
{   //当前关闭登录窗口时执行
    sprintf(username,"guest");
    if(do_connect()==TRUE)
    {
        gtk_widget_set_sensitive(login_button,FALSE);
        g_thread_create((GThreadFunc)get_message,NULL,FALSE,NULL);
    }
    gtk_widget_destroy(widget);
}

void    on_button_clicked(GtkButton *button, gpointer data)
{   //当点击登录窗口的登录按钮时执行
    const gchar*name;

    name = gtk_entry_get_text(GTK_ENTRY(name_entry));
    sprintf(username,"%s",name);
    if(do_connect())
    {
        gtk_widget_set_sensitive(login_button,FALSE);
        g_thread_create((GThreadFunc)get_message,NULL,FALSE,NULL);
    }
    gtk_widget_destroy(data);
}

void    create_win(void)
{   //创建登录窗口
    GtkWidget *win, *vbox;
    GtkWidget *button;

    win = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(win),"输入用户名");
    gtk_container_set_border_width(GTK_CONTAINER(win),10);
    g_signal_connect(G_OBJECT(win),"delete_event",
            G_CALLBACK(on_destroy),NULL);
    gtk_window_set_modal(GTK_WINDOW(win),TRUE);
```

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

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(win),vbox);

    name_entry = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(vbox),name_entry,TRUE,TRUE,5);
    button = gtk_button_new_from_stock(GTK_STOCK_OK);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_button_clicked),win);
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);

    gtk_widget_show_all(win);
}

gboolean do_connect(void)   //连接多人聊天服务器
{
    GtkTextIter iter;
    gint slen;
    sd = socket(AF_INET,SOCK_STREAM,0);//创建
    if(sd < 0)
    {
        gtk_text_buffer_get_end_iter(buffer,&iter);
        gtk_text_buffer_insert(buffer,&iter,"打开套接字时出错! \n",-1);
        return FALSE;
    }
    s_in.sin_family = AF_INET;
    s_in.sin_port = OURPORT;
    slen = sizeof(s_in);
    if(connect(sd,&s_in,slen) < 0) //连接
    {
        gtk_text_buffer_get_end_iter(buffer,&iter);
        gtk_text_buffer_insert(buffer,&iter,"连接服务器时出错! \n",-1);
        return FALSE;
    }
    else
    {
        gtk_text_buffer_get_end_iter(buffer,&iter);
        gtk_text_buffer_insert(buffer,&iter,username,-1);
        gtk_text_buffer_get_end_iter(buffer,&iter);
        gtk_text_buffer_insert(buffer,&iter,"\n成功与服务器连
接....\n",-1);
        //
        write(sd,username,64);//向服务器发送用户名
        //
        isconnected = TRUE;
        return TRUE;
    }
}

void    on_send (GtkButton* button, gpointer data)
```

```
{   //向服务器发送数据
    const char* message;

    if(isconnected==FALSE) return;
    message = gtk_entry_get_text(GTK_ENTRY(message_entry));
    sprintf(buf,"%s\n",message);
    write(sd,buf,1024);//发送
    gtk_entry_set_text(GTK_ENTRY(message_entry),"");
    //清除单行录中的文字
}

void    on_login(GtkWidget *button, gpointer data)
{   //点击登录按钮时执行
    create_win();
}

voidon_delete_event (GtkWidget *widget, GdkEvent* event, gpointer data)
{   //关闭主窗口时执行
    close(sd);//关闭
    gtk_main_quit();
}

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

    if(!g_thread_supported())
        g_thread_init(NULL);    //初始线程
    gtk_init(&argc,&argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"客户端");
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(on_delete_event),NULL);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);

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

    hbox = gtk_hbox_new(FALSE,0);
    gtk_box_pack_start(GTK_BOX(vbox),hbox,FALSE,FALSE,5);
    label = gtk_label_new("点击登录按钮连接服务器");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    login_button = gtk_button_new_with_label("登录");
    gtk_box_pack_start(GTK_BOX(hbox),login_button,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(login_button),"clicked",
            G_CALLBACK(on_login),NULL);

    view = gtk_scrolled_window_new(NULL,NULL);
    gtk_scrolled_window_set_policy(GTK_SCROLLED_WINDOW(view),
```

```
        GTK_POLICY_AUTOMATIC,GTK_POLICY_AUTOMATIC);
    text = gtk_text_view_new();
    gtk_box_pack_start(GTK_BOX(vbox),view,TRUE,TRUE,5);
    gtk_container_add(GTK_CONTAINER(view),text);
    buffer = gtk_text_view_get_buffer(GTK_TEXT_VIEW(text));

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

    label = gtk_label_new("输入消息：");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);

    message_entry = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(hbox),message_entry,FALSE,FALSE,5);

    button = gtk_button_new_with_label("发送");
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_send),NULL);

    gtk_widget_show_all(window);

    gdk_threads_enter();//
    gtk_main();
    gdk_threads_leave();//

    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o client client.c 'pkg-config --cflags --libs gtk+-2.0
gthread-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./client 即可运行此程序，运行结果如图 10.5 所示。

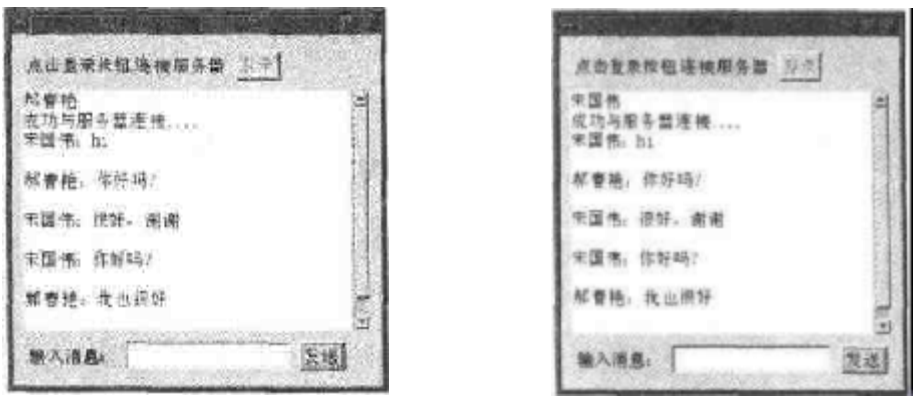

图 10.5 多人聊天的客户端

实例分析

(1) 登录功能

此示例的登录功能除了创建套接字和向服务器发出连接请求外，在连接成功后会自动向服务器发送用户名，服务器则会将此用户名保存到相应的用户数据结构中去，以后每次向其他用户发送数据时，都会将此用户名附加到数据的前面，使其他用户知道发送此信息的人是谁。

(2) 随时接收来自服务器的数据

GLIB 的核心支持部分提供了对线程的支持在 GTK+2.0 中依然可用，然而在 GTK+2.0 的主事件循环中还要用到 gdk_threads_*系列函数，使线程能够改变其他控件的内容。此示例在用户登录后会创建一个线程循环读取来自服务器的数据，如读取成功则显示到文本显示控件中来。

这两节的有关多人聊天室的服务器和客户端的实现是 Linux 套接字编程和创建进程和线程、利用系统调用的一个深化，读者一定要在这方面打好基础，以便进一步研究。

本章中向读者介绍了常见的 Linux 网络服务编程和用 GTK+2.0 做服务的客户端，这只是网络编程的一小部分，但这是初学者学习网络编程的起步基点，学习网络编程的朋友可以由此走向深入。

第11章 高级应用

本章重点：

本章主要介绍GTK+2.0中的一些较复杂的内容，其中包括控件的外观、外观启动文件、线程、用C++封装控件和一些杂项实用功能。这些功能在初学者眼里应当算是高级应用了，它们在使用和理解上可能都需要读者费一番功夫，不过利用好这些功能会使您的程序更加出色。

本章主要内容：

- 控件外观的单独与统一管理
- 如何在GTK+2.0中运用线程
- 如何运用C++控件和对象
- 杂项实用功能

11.1 更改控件的外观

本节示例将介绍在GTK+2.0中如何利用代码更改控件的外观。

实例说明

如果在程序的界面中只使用单一的白底黑字、单一的字体，会使程序变得比较乏味。前面所学的章节中介绍更改控件外观的内容不多，本节示例将详细介绍，实现丰富多彩的控件外观。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk
mkdir 11
cd 11
mkdir style
cd style
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以style.c为文件名保存到当前目录下。

```
/* 设定控件的样式  style.c */
#include <gtk/gtk.h>
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox;
```

```
    GtkWidget *button;
    GtkWidget *label;
    GtkWidget *entry;
    PangoFontDescription *desc;

    static GdkColor red =    { 0, 0xffff, 0,      0      };
    static GdkColor green =  { 0, 0,      0xffff, 0      };
    static GdkColor blue =   { 0, 0,      0,      0xffff };
    static GdkColor yellow = { 0, 0xffff, 0xffff, 0      };
    static GdkColor cyan =   { 0, 0     , 0xffff, 0xffff };
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"设定控件的样式");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTER);
    gtk_container_set_border_width(GTK_CONTAINER(window),15);

    desc = pango_font_description_from_string("Simhei 24");
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);

    button = gtk_button_new_with_label("绿色的按钮");
    gtk_widget_modify_bg(button,GTK_STATE_NORMAL,&green);
    gtk_widget_modify_bg(button,GTK_STATE_ACTIVE,&cyan);
    gtk_widget_modify_bg(button,GTK_STATE_PRELIGHT,&red);
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,8);
    button = gtk_button_new_with_label("大字体的按钮");
    gtk_widget_modify_font(GTK_BIN(button)->child,desc);
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,8);

    button = gtk_button_new_with_label("红色字的按钮");
    gtk_widget_modify_fg(GTK_BIN(button)->child,GTK_STATE_NORMAL,&re
d);
    gtk_widget_modify_fg(GTK_BIN(button)->child,
GTK_STATE_PRELIGHT,&green);
    gtk_widget_modify_fg(GTK_BIN(button)->child,GTK_STATE_ACTIVE,&bl
ue);
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,8);
    label = gtk_label_new("蓝色的标签文字\n主要是改变了前景颜色");
    gtk_widget_modify_fg(label,GTK_STATE_NORMAL,&blue);
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSF,8);
    desc = pango_font_description_from_string("Simsun 24");
    entry = gtk_entry_new();
    gtk_entry_set_text(GTK_ENTRY(entry),"红色的输入文字");
    gtk_widget_modify_font(entry,desc);
    gtk_widget_modify_text(entry,GTK_STATE_NORMAL,&red);
    gtk_box_pack_start(GTK_BOX(vbox),entry,FALSE,FALSE,8);

    entry = gtk_entry_new();
    gtk_widget_modify_font(entry,desc);
```

```
    gtk_entry_set_text(GTK_ENTRY(entry),"绿色的背景");
    gtk_widget_modify_base(entry,GTK_STATE_NORMAL,&green);
    //gtk_widget_modify_text(entry,GTK_STATE_NORMAL,&red);
    gtk_box_pack_start(GTK_BOX(vbox),entry,FALSE,FALSE,8);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o style style.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./style 即可运行此程序，运行结果如图 11.1 所示。

图 11.1 更改控件的外观

实例分析

控件的外观主要包括控件的背景颜色、控件的前景颜色、控件的字体等因素，控件的状态包括正常状态(GTK_STAT_NORMAL)、控件获得焦点时的状态(GTK_STAT_ACTIVE)、鼠标移动到控件上方时的状态(GTK_STAT_PRELIGHT)、控件失去功能时的状态(GTK_STAT_INSENSITIVE)、控件被选中时的状态(GTK_STAT_SELECTED)等等，不同的状态可以设定不同的外观。

用 gtk_widget_modify_*系列函数来设定控件不同状态时的外观，其中函数 gtk_widget_modify_bg 用来设定控件的背景颜色；函数 gtk_widget_modify_fg 用来设定控件的前景颜色；函数 gtk_widget_modify_font 用来设定控件的字体，它的第二个参数是 PANGO 的字体类型，可以用函数 pango_font_description_from_string 来创建，此函数的参数是字体名称和字体的大小组成的字符串；用函数 gtk_widget_modify_text 来改变文字录入控件录入的文字颜色，用函数 gtk_widget_modify_base 来改变文字录控件的背景颜色。

还有一些更改控件外观的函数，可以在 GTK+2.0 的 API 参考手册的 GtkWidget 一节中找到详细说明。

一个漂亮的界面是软件给人的第一外观印象，当软件功能不相上下时，外观的好坏会成为影响使用者选择的重要因素，所以对于应用 GTK+2.0 开发软件的读者一定要认真研究一下这方面的内容。

11.2　做一个桌面主题

本节将通过编制一个简单的外观启动文件来介绍如何编写外观启动文件和在程序中如何使用外观启动文件，从而实现一个桌面主题。

实例说明

上节示例中用函数来改变控件的外观，这在控件较少的小程序中看不出问题，但在大型软件项目中代码会迅速膨胀，以至于令人难以接受。好在 GTK+2.0 提供了外观启动文件(rcfile)功能，它可以让单个或多个程序配置为同一风格的显示外观，而几乎不用多余的编码。本节示例就演示了如何编写和使用外观启动文件做一个桌面主题。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/11
mkdir rcfile
cd rcfile
```

创建本节的工作目录，进入此目录开始编程。用 GIMP 做一幅浅色的背景图像，以 back.png 为文件名保存当前目录下。

(2) 打开编辑器，在当前目录下编辑以下文件：

① 编辑自定义的外观启动文件 our_gtkrc，保存到当前目录下。

```
#启动文件测试
style "default"
{
    fg[INSENSITIVE] = {0,0,0 }
    bg[NORMAL]      = {0.26,0.73,0.30}
    bg[PRELIGHT]    = {0.05,0.39,0.01}
    bg[ACTIVE]      = {0.24,0.67,0.28}
    bg[INSENSITIVE] =  {0.26,0.73,0.30}
    base[NORMAL]      = { 0.83,0.95,0.83 }
    bg_pixmap[NORMAL] = "back.png"
    bg_pixmap[ACTIVE] = "back.png"
    bg_pixmap[PRELIGHT] = "back.png"
    font_name = "simsun 16"
}
class "GtkWidget" style "default"
```

② 编辑测试程序 rc.c，保存到当前目录下。

```
/* GTK+2.0程序的启动文件 rc.c */
#include <gtk/gtk.h>
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *swin, *hbox;
    GtkWidget *button, *label;
    gtk_init(&argc,&argv);
    gtk_rc_parse("our_gtkrc");//关键，此处解析gtkrc文件
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"外观启动文件");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTE
R);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);

    swin = gtk_scrolled_window_new(NULL,NULL);
    gtk_container_add(GTK_CONTAINER(window),swin);
    gtk_scrolled_window_set_policy(GTK_SCROLLED_WINDOW(swin),
                GTK_POLICY_AUTOMATIC,GTK_POLICY_AUTOMATIC);
    hbox = gtk_hbox_new(FALSE,0);
    gtk_container_set_border_width(GTK_CONTAINER(hbox),10);
    gtk_scrolled_window_add_with_viewport(
GTK_SCROLLED_WINDOW(swin),hbox);
    button = gtk_button_new_with_label("普通按钮");
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);
    button = gtk_toggle_button_new_with_label("状态按钮");
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);
    button = gtk_radio_button_new_with_label(NULL,"单选按钮");
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);
    button = gtk_check_button_new_with_label("多选按钮");
    gtk_box_pack_start(GTK_BOX(hbox),button,FALSE,FALSE,5);
    label = gtk_label_new("标签文字");
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    label = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(hbox),label,FALSE,FALSE,5);
    gtk_entry_set_text(GTK_ENTRY(label),"单行录入");
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o ourrc rc.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4)　在终端中执行 make 命令开始编译；

(5)　编译结束后，执行命令./ourrc 即可运行此程序，运行结果如图 11.2 所示。

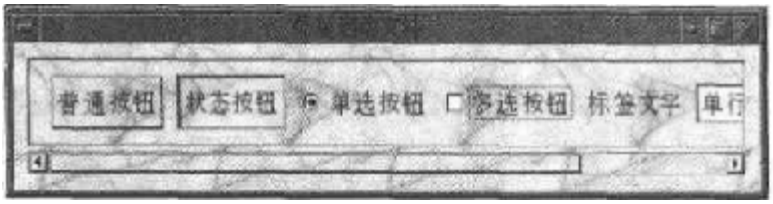

图 11.2　新的外观

实例分析

(1)　关于外观启动文件

在用户目录下的“.gtkrc-2.0”文件，相关的还有“/usr/etc/gtk+-2.0/”目录下的 gtkrc 和“gtkrc.zh_CN”(对应用为中文简体版的 Linux)两个文件。如果把上面编辑的 our_gtkrc 保存为用户目录下的.gtkrc-2.0 的话，您就会发现以前编译的示例的界面外观都会变成这个样子。外观启动文件说明了 GTK+2.0 程序运行时控件的前景颜色、背景颜色、背景图像、字体名称大小等，以下将分别介绍。

(2)　外观启动文件的格式

在外观启动文件中，以#号开始的行为注释行，只起到说明其他行的功能的作用。关键字 include 指明外观启动文件中要包含的文件的路径，如：include “/usr/share/theme”；关键字 pixmap_path 指明图像文件的路径，如：pixmap_path “/usr/share/pixmap”；用关键字 style 来定义一种风格，内容用大括号括上，如本例中的：style “default” { … }，其中包括控件的背景、前景的不同状态的颜色，用 fg[INSENSITIVE]表示窗口失去焦点时的前景，方括号中为控件状态(见上节说明)，等号后面是由大括号括起来的 3 个用逗号分隔开的浮点数，用来表示颜色值，bg 表示背景，bg_pixmap 表示背景图像，等号后面为用引号引起来的图像文件名，此文件可以在当前目录下或前面指定的图像目录下；font_name 表示字体名称。

定义好风格后可以用 class 关键字来为控件指定风格，如本例中的 class “GtkWidget” style “default”，指定所有 GtkWidget 型的控件都是 default 风格的。用户可以在外观启动文件中定义多个风格，为不同的控件指定不同的风格，这样一个桌面主题就完成了。

(3)　单独使用外观启动文件

在程序代码的开头加入函数 gtk_rc_parse(“our_gtkrc”)就可以使程序单独使用自己的外观启动文件，参数为外观文件名，如本例中的 our_gtkrc，相关的函数还有很多。在 GTK+2.0 中，外观启动文件是比较复杂的课题，本示例只是一个简单的介绍，读者可以在 GTK+2.0 的 API 参考手册的 Rcfile 一节中找到相关的详细说明。

需要注意的是外观启动文件只改变 GTK+2.0 程序运行时 GTK+2.0 控件的外观，并不能改变窗口的边框、按钮、标题栏等的外观，因为这些是由 Linux 桌面上的窗口管理器来实现和管理的，这部分又称做窗口管理器的桌面主题。

外观启动文件是所有 GTK+2.0 应用程序一开始运行时就读取和解释的配置文件，它还提供了引擎机制，读者可以到/usr/lib/gtk-2.0/engine 目录中找到这些动态链接库和说明。花哨一些的外观对一些职业编程者和编程爱好者有时并不很重要，但不能排除这是诱使一些

初学者使用的一个重要因素。

11.3 使 用 线 程

本节将介绍如何运用 GLIB 中的线程功能和在 GTK-2.0 中运用线程编程实现动态效果。

实例说明

线程的使用会大大增加程序的功能和提高程序运行的效率，Linux 系统对线程提供了强大的支持。本例演示了如何在 GTK+2.0 中使用线程功能，在窗口中放一个自由布局控件，在自由布局控件中放 2 个按钮“1”和“2”，点击开始按钮，按钮“1”和“2”就会按顺时针方向沿方形路线运动。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/11
mkdir thread
cd thread
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，输入以下代码，以 thread.c 为文件名保存到当前目录下。

```
/* 线程的运用 thread.c */
/* 注意在编译时用到 'pkg-config --cflags --libs gtk+-2.0 gthread-2.0'*/
#include <gtk/gtk.h>
static GtkWidget *fixed;    //自由布局控件
static GtkWidget *button1; //按钮一
static GtkWidget *button2; //按钮二
//第二个线程要运行的函数
void    our_thread2(GtkWidget *button)
{
    gint x,y,towards;
    x = 40;
    y = 40;
    towards = 1;
    for(;;)
    {
        g_usleep(4);
        gdk_threads_enter();
        gtk_fixed_move(GTK_FIXED(fixed),button,x,y);
        switch(towards)
        {
        case 1 : //向右
            x = x + 10 ;
            if(x == 250) towards = 2;
```

```
            break;
        case 2 : //向下
            y = y + 10 ;
            if(y == 250 ) towards = 3 ;
            break;
        case 3 : //向左
            x = x - 10 ;
            if(x == 40) towards = 4;
            break;
        case 4 :    //向上
            y = y - 10 ;
            if(y == 40) towards = 1;
            break;
        }
        gdk_threads_leave();
    }
}
//第二个线程要运行的函数
void    our_thread1(GtkWidget *button)
{
    gint i,j,forward;
    i = 10;
    j = 10;
    forward = 1;
    for(;;)
    {
        g_usleep(1);
        gdk_threads_enter();
        gtk_fixed_move(GTK_FIXED(fixed),button,i,j);
        switch(forward)
        {
        case 1 : //向右
            i = i + 10 ;
            if(i == 290) forward = 2;
            break;
        case 2 : //向下
            j = j + 10 ;
            if(j == 290 ) forward = 3 ;
            break;
        case 3 : //向左
            i = i - 10 ;
            if(i == 10) forward = 4;
            break;
        case 4 :    //向上
            j = j - 10 ;
            if(j == 10) forward = 1;
            break;
        }
        gdk_threads_leave();
    }
```

```
}
void    on_begin(GtkWidget* button, gpointer data)
{
    gtk_widget_set_sensitive(button,FALSE);
    g_thread_create(our_thread1, button1, FALSE, NULL);//创建线程一
    g_thread_create(our_thread2, button2, FALSE, NULL);//创建线程二
}
int main(int argc, char* argv[])
{
    GtkWidget *window, *view;
    GtkWidget *vbox, *button, *label;
    if(!g_thread_supported())
        g_thread_init(NULL);//初始线程
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"线程的运用");
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);

    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    label = gtk_label_new("注意下面的按钮的运动: ");
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,5);
    view = gtk_viewport_new(NULL,NULL);
    gtk_box_pack_start(GTK_BOX(vbox),view,FALSE,FALSE,5);
    fixed = gtk_fixed_new();
    gtk_widget_set_usize(fixed,330,330);
    gtk_container_add(GTK_CONTAINER(view),fixed);
    button1 = gtk_button_new_with_label("1");
    button2 = gtk_button_new_with_label("2");
    gtk_fixed_put(GTK_FIXED(fixed),button1,10,10);
    gtk_fixed_put(GTK_FIXED(fixed),button2,40,40);

    button = gtk_button_new_with_label("开始");
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_begin),NULL);
    gtk_widget_show_all(window);
    gdk_threads_enter();
    gtk_main();
    gdk_threads_leave();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码:

```
CC = gcc
all:
    $(CC) -o thread thread.c 'pkg-config --cflags --libs gtk+-2.0
gthread-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./thread 即可运行此程序，运行结果如图 11.3 所示。

图 11.3　使用线程

实例分析

(1) 线程的初始化

使用函数 g_thread_supported 来判断线程是否已经初始化，如果未初始化返回 FALSE 值，使用函数 g_thread_init 来初始化线程应用，但这并不意味着如此就可以创建线程了，因为GTK+2.0的主循环运行的同时并不允许线程运行。这里我们就用到了gdk_threads_enter 函数，它负责让线程进入 GTK+2.0 的主循环并同时运行线程，也就是说如此之后，线程就可以对 GTK+2.0 主循环中的元素(包括控件、变量等)进行操作了，操作之后，要想使主循环继续接收来自外部的事件，还必须用函数 gdk_threads_leave 来使线程离开，这两个函数一般是成对出现的，没有特殊情况不要拆开使用。

(2) 创建线程

在初始化线程以后，使用函数 g_thread_create 来创建线程，它有 4 个参数，第 1 个参数是线程运行的函数名；第 2 个参数是传递给线程运行的函数的参数；第 3 个参数表示是否允许等待其他线程的加入，TRUE 表示允许；第 4 个参数是一个出错信息的结构，如果创建线程出错，将出错信息保存到此结构中，可以为 NULL。本例中的两个函数分别创建死循环，让按钮不停地沿方形路线不停移动，线程创建后就马上运行。

(3) 程序的编译

GTK+2.0 中所有与线程相关的函数都包含在 GLIB 中的线程部分库中，编译时一定要为包配置工具加 gthread 参数，如本例中的'pkg-config --cflags --libs gthread-2.0'。所有与线程有关的 API 函数都可在 GLIB 的 API 参考手册中的核心支持部分找到。

使用线程的前提是对线程概念和功能的理解，在字符界面中使用线程的演示程序有些初学者并不能深刻理解。此示例借鉴了 Windows 下的开发工具 Dephi 中的线程功能演示程

序，由于采用 GLIB 中的线程功能，使得代码简短易懂。

11.4 动态链接库

本节将介绍如何创建和使用动态链接库和利用 GLIB 中的动态链接库功能。

实例说明

在 Windows 下编程的人对动态链接库一定不陌生，因为很多可执行文件都需要动态链接库的支持，否则不能运行，在 Linux 下也一样。Linux 下的动态链接库多数都存放在/lib、/usr/lib、/usr/local/lib 等目录下，打开这些目录您会发现这些文件(*.a、*.o、*.so)和 Windows 下的*.dll 文件的相似之处——数量惊人。本例演示了如何运用 GTK+2.0 中提供的创建和使用动态链接库的功能。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/11
mkdir dll
cd dll
```

创建本节的工作目录，进入此目录开始编程。

(2) 打开编辑器，在当前目录下编辑如下文件：

① 动态链接库文件 hello.c，代码如下：

```
/* 动态链接库 hello */
#include <gtk/gtk.h>
void    hello   (void)
{
    GtkWidget *dialog;
    dialog = gtk_message_dialog_new(NULL,
            GTK_DIALOG_MODAL | GTK_DIALOG_DESTROY_WITH_PARENT,
            GTK_MESSAGE_INFO, GTK_BUTTONS_OK,
            "您好! \n这是动态链库测试。");
    gtk_dialog_run(GTK_DIALOG(dialog));
    gtk_widget_destroy(dialog);
}
void    create_info_dialog  (gchar*  message)
{
    GtkWidget *dialog;
    dialog = gtk_message_dialog_new(NULL,
            GTK_DIALOG_MODAL|GTK_DIALOG_DESTROY_WITH_PARENT,
            GTK_MESSAGE_INFO, GTK_BUTTONS_OK, message);
    gtk_dialog_run(GTK_DIALOG(dialog));
    gtk_widget_destroy(dialog);
}
```

② 测试程序文件 main.c，代码如下：

```
/* 动态链接库测试  main.c */
#include <gtk/gtk.h>
#include <gmodule.h>
//定义函数指针类型对应hello函数
typedef void (*simplehello) (void);
//定义函数指针类型对应create_info_dialog函数
typedef void (*simplefunc) (gchar *string);

GModule *module;//插件指针
void    on_button_test(GtkButton *button, gpointer data)
{
    simplehello f1;
    simplefunc f2;
    if(g_module_supported())
    {
        gtk_button_set_label(button,"支持动态链接库");
        module =
g_module_open("./libhello.so",G_MODULE_BIND_LAZY);
        g_module_symbol(module,"hello",(gpointer*)&f1);
        f1();//显示您好对话框

    g_module_symbol(module,"create_info_dialog",(gpointer*)&f2);
        f2("此对话框是从动态链接库中调用出来的。");//显示自定义对话框
        g_module_close(module);
    }
    else
    {
        gtk_button_set_label(button,"不支持动态链接库");
    }
}
int main(int argc, char* argv[])
{
    GtkWidget  *window, *button;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"动态链接库测试");
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_window_set_default_size(GTK_WINDOW(window),200,40);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
    button = gtk_button_new_with_label("测试");
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_button_test),NULL);
    gtk_container_add(GTK_CONTAINER(window),button);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) 'pkg-config --cflags gtk+-2.0' -fPIC -DPIC -c hello.c -o
hello.lo
    $(CC) 'pkg-config --libs gtk+-2.0' -shared hello.lo -o libhello.so
    $(CC) main.c -o dll 'pkg-config --cflags --libs gmodule-2.0 gtk+-2.0'
clean:
    rm *.o *.lo *.a *.la .libs -fr
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./dll 即可运行此程序，运行结果如图 11.4 所示。

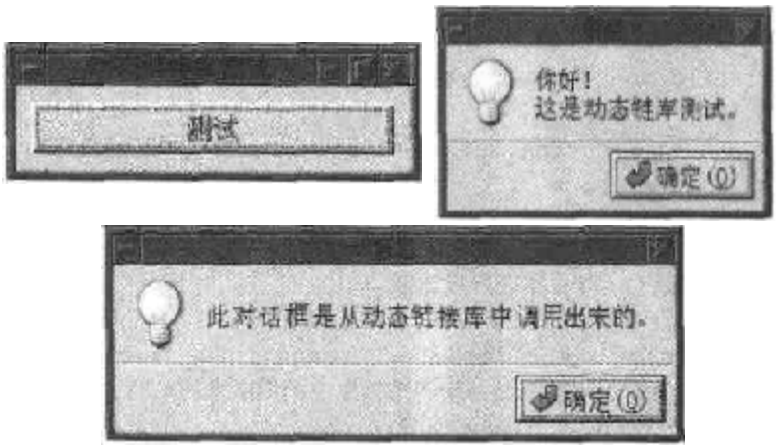

图 11.4 动态链接库

实例分析

(1) 创建动态链接库

在 Linux 中用 C 语言创建动态链接库并不繁琐。首先编辑一个 C 语言源程序，其中包含一个或多个函数，我们这里有两个函数，但不能包括主函数，然后用编译器 gcc 用-c 参数编译，用-o 参数输出*.lo 文件，如本例中的：gcc –fPIC –DPIC –c hello.c –o hello.lo，如此后再进一步用-shared 参数生成*.so 文件，即动态链接库，如：gcc –shared hello.lo –o libhello.so。

(2) 使用动态链接库

GTK+2.0 使用动态链接库的功能在 GLIB 中，称做 gmodule。函数 g_module_supported 用来判断系统是否支持动态链接库；函数 g_module_open 用来打开动态链接库，它返回 GMoudle 类型的指针，它有两个参数，第一个参数是要打开的动态链接库名，第二个是动态链接的标记，它可以取两个值，其中本例中的 G_MODULE_BIND_LAZY 是所有平台都支持的；函数 g_module_symbol 可以取得动态链接库中的函数指针，它有三个参数，第一个参数是上面打开的动态链接库的 GMoudle 指针，第二个参数是动态链接库中的函数名，第三个参数是要调用的函数的类型指针，如本例中前面的声明。其中函数的指针类型要和链接库中定义的函数类型一致，函数名称也一样，定义方法也要认真理解一下。在取得函

数的指针后就可以直接运行此函数了，如例中的 f1()和 f2("字符串")。

(3) 编译调用动态链接库的程序

编译调用动态链接库的 GTK 程序，要为配置工具 pkg-confg 加 gmoudle 参数，一般用法如下：gcc –o hello 'pkg-config --cflags --libs gmodule-2.0'。

本示例只是简单创建和使用动态链接库的演示，更多的内容可见 GLIB 中 API 参考的关于动态链接库的部分。

11.5　用 C++封装控件

本节示例将介绍如何用 C++语言对 GTK+2.0 的控件和对象进行简单的封装。

实例说明

GTK+2.0 是用 C 语言开发的工具库，虽然具有面向对象特色，但仍缺少 C++的很多优点，致使众多的 C++爱好者对它不屑一顾。幸好 GTK+2.0 有着良好的可扩展性，并且 C++是 C 的一个超集，使很多 GTK+2.0 的程序可以直接用 C++编译器编译。事实上只要您掌握了 GTK+2.0 的 C 语言 API，就完全可以用 C++开发 GTK+2.0 程序。本示例就用 C++封装了一个多选按钮控件和一个对话框。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/11
mkdir cpp
cd cpp
```

创建本节的工作目录，进入此目录开始编程。创建两幅图像 active.png 和 normal.png 来表示多选按钮的被选中和未被选中这两种状态。

(2) 打开编辑器，在当前目录下创建 5 个源代码文件：

① 封装对话框的 C++头文件 use.h，代码如下：

```
#ifndef __USE_H
#define __USE_H
#include <gtk/gtk.h>
//定义关于对话框类
class about
public:
    GtkWidget *window;
public:
    about();
    ~about();
    GtkWidget* init();
    void show();
    //定义回调函数
    static void on_button_ok    (GtkButton *button,gpointer
```

```
data);
};
#endif  //__USE_H
```

② 封装对话框的代码实现 use.cpp，代码如下：

```
/* use.cpp */
#include "use.h"
//about类型声明为外部类型
extern about *a;
//空的构造函数
about :: about()
{
}
//空的析构函数
about :: ~about()
{
}
//初始化对话框
GtkWidget*  about :: init   ()
{
    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *frame;
    GtkWidget *label;
    GtkWidget *button;
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),"关于用C++封装GTK+2.0
程序");
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_widget_destroy),window);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);
    vbox = gtk_vbox_new(FALSE,0);
    gtk_container_add(GTK_CONTAINER(window),vbox);
    frame = gtk_frame_new("此处写软件名称：");
    gtk_box_pack_start(GTK_BOX(vbox),frame,FALSE,FALSE,5);
    label = gtk_label_new("此软件用于测试。\n作者：宋国伟\n2002年6月\n
用C++封装GTK+2.0程序有很多方法和途径，\n这对编程爱好者来说是一种非常好的选
择。");
    gtk_container_add(GTK_CONTAINER(frame),label);

    button = gtk_button_new_from_stock(GTK_STOCK_OK);
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_button_ok),NULL);
    return window;
}
//显示对话框
void    about :: show()
{
    window = init();
```

```
    gtk_widget_show_all(window);
}
//确认按钮的回调函数
void    about :: on_button_ok   (GtkButton *button,gpointer data)
{
    gtk_widget_destroy(a->window);//销毁窗口的指针
}
```

③ 自定义多选按钮头文件 ourcheck.h，代码如下：

```
//ourcheck.h
#ifndef __OURCHECK_H__
#define __OURCHECK_H__
#include <gtk/gtk.h>
//自定义多选按钮类
class   ourcheck{
    public:
    GtkWidget *widget, *button, *image, *label;
    public:
    ourcheck(gchar *title);
    ~ourcheck();
    void init(gchar *title);//创建
    //为状态按钮的toggled信号加回调函数
    static void on_toggle(GtkToggleButton* button, gpointer data);
};
#endif  //__OURCHECK_H__
```

④ 自定义多选按钮的代码实现 ourcheck.cpp，代码如下：

```
//ourcheck.cpp
#include "ourcheck.h"
//状态按钮的toggled信号的回调函数实现
void ourcheck :: on_toggle (GtkToggleButton* button, gpointer data)
{
    if(GTK_TOGGLE_BUTTON(button)->active)//根据状态按钮的状态改变图
像
        gtk_image_set_from_file(GTK_IMAGE(data),"active.png");
    else
        gtk_image_set_from_file(GTK_IMAGE(data),"normal.png");
}
//构造函数
ourcheck :: ourcheck(gchar *title)
{
    init(title);
}
//空的析构函数
ourcheck :: ~ourcheck()
{
}
//初始多选按钮
void    ourcheck :: init (gchar* title)
{
```

```
    widget = gtk_hbox_new(FALSE,0);
    button = gtk_toggle_button_new();
    image = gtk_image_new_from_file("normal.png");
    g_signal_connect(G_OBJECT(button),"toggled",
            G_CALLBACK(on_toggle),(gpointer)image);
    gtk_container_add(GTK_CONTAINER(button),image);
    label = gtk_label_new(title);
    gtk_box_pack_start(GTK_BOX(widget),button,FALSE,FALSE,2);
    gtk_box_pack_start(GTK_BOX(widget),label,FALSE,FALSE,2);
    gtk_widget_show_all(widget);
}
```

⑤ 主程序 main.cpp，代码如下：

```
/* main.cpp */
#include <gtk/gtk.h>
#include "use.h"
#include "curcheck.h"
//定义对话框类指针和多选按钮指针
about *a;
ourcheck *ocheck;
void    on_button_clicked   (GtkButton *button,gpointer data)
{
    a = new about();//创建
    a->show();
}
void    on_delete_event (GtkWidget* widget, GdkEvent* event,
gpointer data)
{
    delete a;   //释放
    delete ocheck;
    gtk_main_quit();
}
int main ( int argc , char* argv[])
{
    GtkWidget *window;
    GtkWidget *vbox;
    GtkWidget *button;
    GtkWidget *frame;
    gtk_init(&argc,&argv);
    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(on_delete_event),NULL);
    gtk_window_set_title(GTK_WINDOW(window),"用C++封装GTK+2.0程序
");
    gtk_window_set_position(GTK_WINDOW(window),GTK_WIN_POS_CENTE
R);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);

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

```
    ocheck = new ourcheck("这是自定义的多选按钮");
    gtk_box_pack_start(GTK_BOX(vbox),ocheck->widget,TRUE,TRUE,5)
;
    button = gtk_button_new_with_label("          关于
");
    gtk_box_pack_start(GTK_BOX(vbox),button,TRUE,TRUE,5);
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_button_clicked),NULL);
    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CPP = g++
usecpp:main.o use.o ourcheck.o
    $(CPP) -o usecpp main.o use.o ourcheck.o 'pkg-config --cflags --libs
gtk+-2.0'
main.o:main.cpp use.h
    $(CPP) -c main.cpp 'pkg-config --cflags gtk+-2.0'
use.o :use.cpp use.h
    $(CPP) -c use.cpp 'pkg-config --cflags gtk+-2.0'
ourcheck.o : ourcheck.cpp ourcheck.h
    $(CPP) -c ourcheck.cpp 'pkg-config --cflags gtk+-2.0'
```

(4) 在终端中执行 make 命令开始编译；

(5) 编译结束后，执行命令./usecpp 即可运行此程序，运行结果如图 11.5 所示。

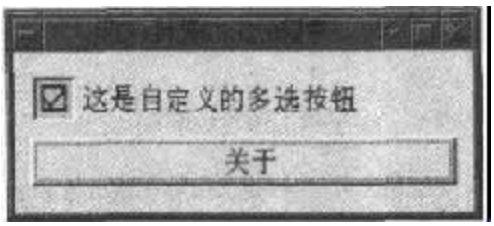

图 11.5　用 C++封装

实例分析

对话框和多选按钮控件都被封装成 C++中的类的形式，这其中的技巧 C++爱好者们能如数家珍地说出很多，我们这里要注意的是一定要把对话框类的对象指针声明为外部的，这使确认的回调函数能正确找到并销毁它。

多选按钮控件事实上是一个状态按钮和一个标签控件的组合，另外我们在封装时并未对此控件加信号，有兴趣的读者可以参照第 7 章的自定义控件部分试一试。

用 C++封装 GTK+2.0 控件和对象，不仅能使代码简化和更具面向对象特色，还可以利用 C++的其他特性如泛型编程和标准模板库(STL)等功能。而且使用多文件来编程更能提高程序的可读性。

最著名的 GTK+2.0 的 C++程序包 gtkmm 可以在 sourceforge.net 网站上找到，另外还有 2 个分别是 VDK(一套可视化的 C++的集成开发环境)和 GCODE(以 C++封装的 GTK+2.0 控件库)。它们都对 GTK+2.0 的控件进行了较深层次的封装，是 C++爱好者的首选研究对象。

11.6 国际化编程

本节将介绍如何用 GTK+2.0 实现国际化编程，使程序对多种语言进行支持。

实例说明

本示例是 GTK+2.0 国际化编程的一个简单应用，它可以支持中文和英文两种语言，即我们在开机时选择英文为登录语种时，运行此示例显示内容为英文；而我们在开机时选择中文为登录语种时，运行此示例显示内容则为中文。

实现步骤

(1) 打开终端输入如下命令：

```
cd ~/ourgtk/11/
mkdir inter
cd inter
```

创建本节工作目录，开始编程。

(2) 打开编辑器，输入以下代码，以 inter.c 为文件名保存到当前目录下。

```
//inter.c   国际化编程
#include <gtk/gtk.h>
#include <libintl.h>
#include <locale.h>     //两个相关包含文件

#define PACKAGE "inter"     //定义国际化信息包名称
#define LOCALEDIR "/home/peter/ourgtk/11/inter/locale" //定义信息包所在
位置

#define _(string)   gettext(string) //定义国际化信息输出的相关宏
#define N_(string)  string

static GtkWidget* entry = NULL;

void    on_button_clicked(GtkWidget *button, gpointer data)
{
    gtk_entry_set_text(GTK_ENTRY(entry),_("this text will put in the
entry"));
}

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

```
    bindtextdomain(PACKAGE,LOCALEDIR);
    //以上函数用来设定国际化翻译包所在位置
    textdomain(PACKAGE);
    //以上函数用来设定国际化翻译包名称，省略了.mo

    gtk_init(&argc,&argv);

    window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title(GTK_WINDOW(window),_("test window"));
    g_signal_connect(G_OBJECT(window),"delete_event",
            G_CALLBACK(gtk_main_quit),NULL);
    gtk_container_set_border_width(GTK_CONTAINER(window),10);

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

    label = gtk_label_new(_("This is a international application."));
    gtk_box_pack_start(GTK_BOX(vbox),label,FALSE,FALSE,5);

    button = gtk_button_new_with_label(_("new button"));
    g_signal_connect(G_OBJECT(button),"clicked",
            G_CALLBACK(on_button_clicked),NULL);
    gtk_box_pack_start(GTK_BOX(vbox),button,FALSE,FALSE,5);

    entry = gtk_entry_new();
    gtk_box_pack_start(GTK_BOX(vbox),entry,FALSE,FALSE,5);

    gtk_widget_show_all(window);
    gtk_main();
    return FALSE;
}
```

(3) 编辑 Makefile 输入以下代码：

```
CC = gcc
all:
    $(CC) -o inter inter.c 'pkg-config --cflags --libs gtk+-2.0'
```

(4) 在终端中输入如下命令：

```
xgettext -k_ -o inter.po inter.c
```

会生成一个名为 inter.po 的文件，我们将其中的相关英文输出内容翻译为中文，主要是将 msgstr“”的引号内加入翻译内容(注意：一定要用 UTF8 格式编码)，文件如下：

```
# SOME DESCRIPTIVE TITLE.
# Copyright (C) YEAR THE PACKAGE'S COPYRIGHT HOLDER
# This file is distributed under the same license as the PACKAGE package.
# FIRST AUTHOR <EMAIL@ADDRESS>, YEAR.
#
#, fuzzy
msgid ""
msgstr ""
```

```
"Project-Ic-Version: PACKAGE VERSION\n"
"POT-Creation-Date: 2002-09-09 18:10+0800\n"
"PO-Revision-Date: YEAR-MO-DA HO:MI+ZONE\n"
"Last-Translator: FULL NAME <EMAIL@ADDRESS>\n"
"Language-Team: LANGUAGE <LL@li.org>\n"
"MIME-Version: 1.0\n"
"Content-Type: text/plain; charset=CHARSET\n"
"Content-Transfer-Encoding: 8bit\n"

#: inter.c:16
msgid "this text will put in the entry"
msgstr "此文本将被放入到单行录入控件之中去"

#: inter.c:31
msgid "test window"
msgstr "测试窗口"

#: inter.c:39
msgid "This is a international application."
msgstr "这是一个国际化的应用程序"

#: inter.c:42
msgid "new button"
msgstr "新建按钮"
```

(5) 在终端中执行如下命令：

```
msgfmt inter.po -o inter.mo
```

生成 inter.mo 文件。

(6) 在终端中执行如下命令：

```
mkdir locale
cd locale
mkdir zh_CN
cd zh_CN
mkdir LC_MESSAGES
cd LC_MESSAGES
cp ../../../inter.mo .
```

在当前目录下创建 locale/zh_CN/LC_MESSAGES 目录，并将生成的 inter.mo 文件复制到上面的目录中，以便程序运行时查找。

(7) 在终端中执行 make 命令开始编译；

(8) 编译结束后，执行命令./inter 即可运行此程序，运行结果如图 11.6 所示。

图 11.6.1 在英文环境下运行的 inter 程序

图 11.6.2 在中文件环境下运行的 inter 程序

实例分析

(1) 国际化编程

在进行国际化编程时需要 gettext 软件包的支持，这也是 GTK+2.0 软件包安装的前提，所以一般不必担心是否安装有此软件包。在程序的源代码中多数包含下面程序段：

```
#include <libintl.h>          //gettext的包含文件
#include <locale.h>      //设定国际化信息翻译包的包含文件
#cefine _(string)   gettext(string) //定义选择国际化信息的函数gettext为以_
开始的宏
#define N_(string)  string  //反向定义
```

如此之后在程序中需要进行国际化输出的地方就可以用这两个宏来替代，这样做有两个好处，一是可以简化编程，二是可以非常方便地输出*.po 文件，同时这也是 GTK+2.0 代码风格的一个标准。

函数 gettext 的功能就是从国际化翻译包中提取相关的输出信息，函数 bindtextdomain 的功能是将相应的目录设定为国际化翻译包所在的目录，函数 textdomain 设定此软件的国际化翻译信息包的名称，省略末尾的 mo。

一般情况下 linux 系统的国际化翻译包均放在/usr/share/locale 目录的相应子目录下，本示例将其设定为此程序的源代码所在目录的 locale 子目录下，需要说明的是，我们将其中的英文翻译为中文，中文的 locale 目录为 zh_CN，而翻译包要放在此目录的 LC_MESSAGES 目录下，所以我们一定要创建好这个目录，并将做好的 inter.mo 文件复制到此目录下。

(2) gettext 工具

在编程时按上面的做好后，可以用 gettext 软件包提供的两个重要工具之一 xgettext 来生成国际化信息包(*.po 文件)，翻译好后再用 msgfmt 工具来生成最终国际化信息翻译包(*.mo 文件)。

工具 xgettext 用到了两个命令行参数，其中-o 用于指定输出文件的名称，-k 用于指定要搜索的字符串的前缀(我们这里是下划线)，后面加入程序源文件的名称即可。

在翻译信息包时，只须将 msgstr“”的引号内加入翻译内容即可。

工具 msgfmt 的功能是将翻译好的国际化信息包进行进一步的编译，使程序中的 gettext 函数能正确地找到相应的翻译信息，它的-o 参数用来指定输出的文件名称，它的输入文件为翻译好的*.po 文件。

本节示例是用 GTK+2.0 进行国际化编程的一个简单尝试，只实现了英文和中文简体两种语言，有能力的读者完全可以将输出的*.po 文件的内容翻译为其他语种，从而进行更广泛的国际化支持。

实例分析

(1) 国际化编程

在进行国际化编程时需要 gettext 软件包的支持，这也是 GTK+2.0 软件包安装的前提，所以一般不必担心是否安装有此软件包。在程序的源代码中多数包含下面程序段：

```
#include <libintl.h>          //gettext的包含文件
#include <locale.h>      //设定国际化信息翻译包的包含文件
#cefine _(string)   gettext(string) //定义选择国际化信息的函数gettext为以_
开始的宏
#define N_(string)  string  //反向定义
```

如此之后在程序中需要进行国际化输出的地方就可以用这两个宏来替代，这样做有两个好处，一是可以简化编程，二是可以非常方便地输出*.po 文件，同时这也是 GTK+2.0 代码风格的一个标准。

函数 gettext 的功能就是从国际化翻译包中提取相关的输出信息，函数 bindtextdomain 的功能是将相应的目录设定为国际化翻译包所在的目录，函数 textdomain 设定此软件的国际化翻译信息包的名称，省略末尾的 mo。

一般情况下 linux 系统的国际化翻译包均放在/usr/share/locale 目录的相应子目录下，本示例将其设定为此程序的源代码所在目录的 locale 子目录下，需要说明的是，我们将其中的英文翻译为中文，中文的 locale 目录为 zh_CN，而翻译包要放在此目录的 LC_MESSAGES 目录下，所以我们一定要创建好这个目录，并将做好的 inter.mo 文件复制到此目录下。

(2) gettext 工具

在编程时按上面的做好后，可以用 gettext 软件包提供的两个重要工具之一 xgettext 来生成国际化信息包(*.po 文件)，翻译好后再用 msgfmt 工具来生成最终国际化信息翻译包(*.mo 文件)。

工具 xgettext 用到了两个命令行参数，其中-o 用于指定输出文件的名称，-k 用于指定要搜索的字符串的前缀(我们这里是下划线)，后面加入程序源文件的名称即可。

在翻译信息包时，只须将 msgstr“”的引号内加入翻译内容即可。

工具 msgfmt 的功能是将翻译好的国际化信息包进行进一步的编译，使程序中的 gettext 函数能正确地找到相应的翻译信息，它的-o 参数用来指定输出的文件名称，它的输入文件为翻译好的*.po 文件。

本节示例是用 GTK+2.0 进行国际化编程的一个简单尝试，只实现了英文和中文简体两种语言，有能力的读者完全可以将输出的*.po 文件的内容翻译为其他语种，从而进行更广泛的国际化支持。

实例分析

(1) 国际化编程

在进行国际化编程时需要 gettext 软件包的支持，这也是 GTK+2.0 软件包安装的前提，所以一般不必担心是否安装有此软件包。在程序的源代码中多数包含下面程序段：

```
#include <libintl.h>          //gettext的包含文件
#include <locale.h>      //设定国际化信息翻译包的包含文件
#cefine _(string)   gettext(string) //定义选择国际化信息的函数gettext为以_
开始的宏
#define N_(string)  string  //反向定义
```

如此之后在程序中需要进行国际化输出的地方就可以用这两个宏来替代，这样做有两个好处，一是可以简化编程，二是可以非常方便地输出*.po 文件，同时这也是 GTK+2.0 代码风格的一个标准。

函数 gettext 的功能就是从国际化翻译包中提取相关的输出信息，函数 bindtextdomain 的功能是将相应的目录设定为国际化翻译包所在的目录，函数 textdomain 设定此软件的国际化翻译信息包的名称，省略末尾的 mo。

一般情况下 linux 系统的国际化翻译包均放在/usr/share/locale 目录的相应子目录下，本示例将其设定为此程序的源代码所在目录的 locale 子目录下，需要说明的是，我们将其中的英文翻译为中文，中文的 locale 目录为 zh_CN，而翻译包要放在此目录的 LC_MESSAGES 目录下，所以我们一定要创建好这个目录，并将做好的 inter.mo 文件复制到此目录下。

(2) gettext 工具

在编程时按上面的做好后，可以用 gettext 软件包提供的两个重要工具之一 xgettext 来生成国际化信息包(*.po 文件)，翻译好后再用 msgfmt 工具来生成最终国际化信息翻译包(*.mo 文件)。

工具 xgettext 用到了两个命令行参数，其中-o 用于指定输出文件的名称，-k 用于指定要搜索的字符串的前缀(我们这里是下划线)，后面加入程序源文件的名称即可。

在翻译信息包时，只须将 msgstr“”的引号内加入翻译内容即可。

工具 msgfmt 的功能是将翻译好的国际化信息包进行进一步的编译，使程序中的 gettext 函数能正确地找到相应的翻译信息，它的-o 参数用来指定输出的文件名称，它的输入文件为翻译好的*.po 文件。

本节示例是用 GTK+2.0 进行国际化编程的一个简单尝试，只实现了英文和中文简体两种语言，有能力的读者完全可以将输出的*.po 文件的内容翻译为其他语种，从而进行更广泛的国际化支持。

实例分析

(1) 国际化编程

在进行国际化编程时需要 gettext 软件包的支持，这也是 GTK+2.0 软件包安装的前提，所以一般不必担心是否安装有此软件包。在程序的源代码中多数包含下面程序段：

```
#include <libintl.h>          //gettext的包含文件
#include <locale.h>      //设定国际化信息翻译包的包含文件
#cefine _(string)   gettext(string) //定义选择国际化信息的函数gettext为以_
开始的宏
#define N_(string)  string  //反向定义
```

如此之后在程序中需要进行国际化输出的地方就可以用这两个宏来替代，这样做有两个好处，一是可以简化编程，二是可以非常方便地输出*.po 文件，同时这也是 GTK+2.0 代码风格的一个标准。

函数 gettext 的功能就是从国际化翻译包中提取相关的输出信息，函数 bindtextdomain 的功能是将相应的目录设定为国际化翻译包所在的目录，函数 textdomain 设定此软件的国际化翻译信息包的名称，省略末尾的 mo。

一般情况下 linux 系统的国际化翻译包均放在/usr/share/locale 目录的相应子目录下，本示例将其设定为此程序的源代码所在目录的 locale 子目录下，需要说明的是，我们将其中的英文翻译为中文，中文的 locale 目录为 zh_CN，而翻译包要放在此目录的 LC_MESSAGES 目录下，所以我们一定要创建好这个目录，并将做好的 inter.mo 文件复制到此目录下。

(2) gettext 工具

在编程时按上面的做好后，可以用 gettext 软件包提供的两个重要工具之一 xgettext 来生成国际化信息包(*.po 文件)，翻译好后再用 msgfmt 工具来生成最终国际化信息翻译包(*.mo 文件)。

工具 xgettext 用到了两个命令行参数，其中-o 用于指定输出文件的名称，-k 用于指定要搜索的字符串的前缀(我们这里是下划线)，后面加入程序源文件的名称即可。

在翻译信息包时，只须将 msgstr“”的引号内加入翻译内容即可。

工具 msgfmt 的功能是将翻译好的国际化信息包进行进一步的编译，使程序中的 gettext 函数能正确地找到相应的翻译信息，它的-o 参数用来指定输出的文件名称，它的输入文件为翻译好的*.po 文件。

本节示例是用 GTK+2.0 进行国际化编程的一个简单尝试，只实现了英文和中文简体两种语言，有能力的读者完全可以将输出的*.po 文件的内容翻译为其他语种，从而进行更广泛的国际化支持。

实例分析

(1) 国际化编程

在进行国际化编程时需要 gettext 软件包的支持，这也是 GTK+2.0 软件包安装的前提，所以一般不必担心是否安装有此软件包。在程序的源代码中多数包含下面程序段：

```
#include <libintl.h>          //gettext的包含文件
#include <locale.h>      //设定国际化信息翻译包的包含文件
#cefine _(string)   gettext(string) //定义选择国际化信息的函数gettext为以_
开始的宏
#define N_(string)  string  //反向定义
```

如此之后在程序中需要进行国际化输出的地方就可以用这两个宏来替代，这样做有两个好处，一是可以简化编程，二是可以非常方便地输出*.po 文件，同时这也是 GTK+2.0 代码风格的一个标准。

函数 gettext 的功能就是从国际化翻译包中提取相关的输出信息，函数 bindtextdomain 的功能是将相应的目录设定为国际化翻译包所在的目录，函数 textdomain 设定此软件的国际化翻译信息包的名称，省略末尾的 mo。

一般情况下 linux 系统的国际化翻译包均放在/usr/share/locale 目录的相应子目录下，本示例将其设定为此程序的源代码所在目录的 locale 子目录下，需要说明的是，我们将其中的英文翻译为中文，中文的 locale 目录为 zh_CN，而翻译包要放在此目录的 LC_MESSAGES 目录下，所以我们一定要创建好这个目录，并将做好的 inter.mo 文件复制到此目录下。

(2) gettext 工具

在编程时按上面的做好后，可以用 gettext 软件包提供的两个重要工具之一 xgettext 来生成国际化信息包(*.po 文件)，翻译好后再用 msgfmt 工具来生成最终国际化信息翻译包(*.mo 文件)。

工具 xgettext 用到了两个命令行参数，其中-o 用于指定输出文件的名称，-k 用于指定要搜索的字符串的前缀(我们这里是下划线)，后面加入程序源文件的名称即可。

在翻译信息包时，只须将 msgstr“”的引号内加入翻译内容即可。

工具 msgfmt 的功能是将翻译好的国际化信息包进行进一步的编译，使程序中的 gettext 函数能正确地找到相应的翻译信息，它的-o 参数用来指定输出的文件名称，它的输入文件为翻译好的*.po 文件。

本节示例是用 GTK+2.0 进行国际化编程的一个简单尝试，只实现了英文和中文简体两种语言，有能力的读者完全可以将输出的*.po 文件的内容翻译为其他语种，从而进行更广泛的国际化支持。

实例分析

(1) 国际化编程

在进行国际化编程时需要 gettext 软件包的支持，这也是 GTK+2.0 软件包安装的前提，所以一般不必担心是否安装有此软件包。在程序的源代码中多数包含下面程序段：

```
#include <libintl.h>          //gettext的包含文件
#include <locale.h>      //设定国际化信息翻译包的包含文件
#cefine _(string)   gettext(string) //定义选择国际化信息的函数gettext为以_
开始的宏
#define N_(string)  string  //反向定义
```

如此之后在程序中需要进行国际化输出的地方就可以用这两个宏来替代，这样做有两个好处，一是可以简化编程，二是可以非常方便地输出*.po 文件，同时这也是 GTK+2.0 代码风格的一个标准。

函数 gettext 的功能就是从国际化翻译包中提取相关的输出信息，函数 bindtextdomain 的功能是将相应的目录设定为国际化翻译包所在的目录，函数 textdomain 设定此软件的国际化翻译信息包的名称，省略末尾的 mo。

一般情况下 linux 系统的国际化翻译包均放在/usr/share/locale 目录的相应子目录下，本示例将其设定为此程序的源代码所在目录的 locale 子目录下，需要说明的是，我们将其中的英文翻译为中文，中文的 locale 目录为 zh_CN，而翻译包要放在此目录的 LC_MESSAGES 目录下，所以我们一定要创建好这个目录，并将做好的 inter.mo 文件复制到此目录下。

(2) gettext 工具

在编程时按上面的做好后，可以用 gettext 软件包提供的两个重要工具之一 xgettext 来生成国际化信息包(*.po 文件)，翻译好后再用 msgfmt 工具来生成最终国际化信息翻译包(*.mo 文件)。

工具 xgettext 用到了两个命令行参数，其中-o 用于指定输出文件的名称，-k 用于指定要搜索的字符串的前缀(我们这里是下划线)，后面加入程序源文件的名称即可。

在翻译信息包时，只须将 msgstr“”的引号内加入翻译内容即可。

工具 msgfmt 的功能是将翻译好的国际化信息包进行进一步的编译，使程序中的 gettext 函数能正确地找到相应的翻译信息，它的-o 参数用来指定输出的文件名称，它的输入文件为翻译好的*.po 文件。

本节示例是用 GTK+2.0 进行国际化编程的一个简单尝试，只实现了英文和中文简体两种语言，有能力的读者完全可以将输出的*.po 文件的内容翻译为其他语种，从而进行更广泛的国际化支持。