

第2章 Gtk+/Gnome开发简介

2.1 安装Gtk+/Gnome库

要想用Gtk+/Gnome编程，首先要保证系统中已经安装了 Gtk+和Gnome库。

一般情况下，Linux发布版本的光盘中都应该包含了所需要的库的源代码。例如 Red Hat Linux 6.0/6.1和TurboLinux 4.0中都有Gtk+和Gnome的最新版本。在安装系统的过程中，当提示安装类型时，一般有“服务器”、“工作站”、“开发工作站”、“自定义”和“完全安装”等几个选项。选择“开发工作站”或“完全安装”，完全安装大多数用于软件开发的库、头文件和实用程序，如Gtk+库、Gnome库、automake、autoconfig、gcc编译器、gdb调试器等。

如果觉得系统中已有的库的版本已经过时，可以从 Internet上下载最新版本，然后安装。可以从Gtk+的Web站点<http://www.gtk.org>下载最新版本的Gtk+。Gtk+所使用的版本号与Linux的版本编号方法类似，偶数版本号（如 1.0和1.2）表示稳定的版本，而奇数版本号（如 0.9和1.1）表示正在开发的版本。有时还增加一个附加数字表示对这一版本进行了修正，如 1.2.1。当前Gtk+的最新版本是1.2.3。

从网上下载的文件名一般是 gtk+1.2.3.tar.gz或者其他类似形式，文件名中包含了该软件包的名称和版本号信息。因为它的后缀是 .tar.gz，所以它是一个归档的压缩文件。用 gunzip命令对它解压缩：

```
gunzip gtk+1.2.3.tar.gz
```

将会产生一个解压缩的以 .tar结尾的归档文件。用 tar命令将它扩展为它的目录结构：

```
tar -xvf gtk+1.2.3.tar
```

这个命令建立了建库所需要的目录结构。进入上面建立的目录，执行 configure脚本生成编译所需的makefile：

```
./configure
```

下面可以建立库了。输入 make命令：

```
make
```

建库后，需要安装刚才建立的库。输入以下命令：

```
make install
```

然后，还需要运行/sbin/ldconfig以使Gtk+能正常工作。

当然，完成上面工作的前提是系统上已经安装了 glib。不过，一般情况下都可以保证做到这一点。如果需要安装新版本的 glib库，操作步骤和安装Gtk+库一样。

在Gtk+源代码目录下的examples子目录下，有很多Gtk+构件示例。这些代码都有很详细的注释，通读这些代码对学习 Gtk+编程也是很有帮助的。本书中关于 Gtk+构件的演示代码都来自这些示例。

Gnome的最新版本可以从<http://www.gnome.org>下载。取得新版本软件后，解压缩和安装的方法与Gtk+类似。

2.2 第一个Gtk+应用程序

2.2.1 一个什么也不能做的窗口

用Gtk+库编程和同时使用Gtk+/Gnome库编程的方法完全相同，只是细节上略有不同。我们将从一个最简单的程序开始介绍GTK，然后用一个简单的例子介绍Gnome库编程的方法。

本程序将创建一个200×200像素的窗口，除了用shell命令kill以外，没有其他的退出程序的方法。

```
/* 例子开始 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(0);}
/*示例结束 */
```

上面的源代码中的/*和*/之间的内容都是注释。Linux中常用的C编译器能够识别/* */和//两种格式的注释。

可以用gcc编译上述程序：

```
gcc base.c -o base `gtk-config --cflags --libs`
```

注意，gtk-config --cflags--libs选项左右的符号不是单引号，而是反引号（键盘上“1”键左边的键）。编译选项的含义选项将在下面编译“Hello World”时解释。

编译结束后，输入以下命令来执行上面创建的应用程序（结果如图2-1所示）：

```
./base
```

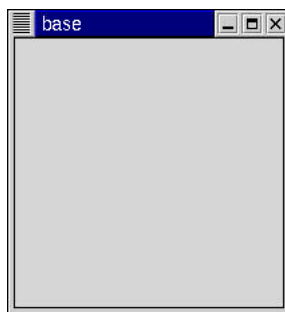


图2-1 第一个Gtk应用程序，它什么也不能做

2.2.2 示例代码的含义

在程序的源代码中，所有用到的函数和数据类型以及结构等都应该声明。也就是说，如果使用了一个按钮构件，应该包含按钮所对应的gtkbutton.h头文件。如果在程序中使用了多种构件和函数库，包含这些头文件将会是一件很麻烦的事，也很容易出错。有一个特殊的头文件，gtk/gtk.h，其中包含了Gtk+编程中所有需要的头文件，也包含gdk.h和glib.h等。在源文件的前面包含这个文件就可以了。后面会看到，如果要用到Gnome的构件和库函数，包含gnome.h就可以了。在gnome.h中已经包含了gtk.h文件和其他相关头文件。

第一行#include <gtk/gtk.h> 包含了所有需要的头文件。

下一行：

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

先调用函数gtk_init(gint *argc, gchar ***argv)。所有GTK应用程序都要调用该函数。它为我们设置一些缺省值例如视觉和颜色映射，然后调用 gdk_init(gint *argc, gchar ***argv)。这

个函数将函数库初始化,设置缺省的信号处理函数,并检查通过命令行传递给应用程序的参数。

主要检查下面所列的一些值:

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

它从变量表中删除以上参数,并分离出所有不识别的值,应用程序可以分析或忽略这些值,由此生成一些Gtk应用程序可以接受的标准参数。

下面两行代码将创建和显示一个窗口:

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

GTK_WINDOW_TOPLEVEL参数指明让窗口使用“窗口管理程序”指定的状态设置和位置布置。没有子窗口的窗口缺省设置为 200×200 像素大小,而不是 0×0 大小。gtk_widget_show()函数让Gtk知道那我们已经设置该构件(窗口)的属性,现在可以显示它了。

最后一行代码进入Gtk主处理循环:

```
gtk_main ();
```

gtk_main()是在每个Gtk应用程序都要调用的函数。当程序运行到这里时,Gtk将进入等待状态,等候X事件(比如点击按钮或按下键盘的某个按键)、Timeout或文件输入/输出发生。

在这个简单例子里,所有事件都被忽略。用鼠标点击窗口右上角的“×”按钮也不能将窗口关闭。唯一关闭它的办法就是使用Shell命令kill删除它的进程。

这里顺便指出:在代码中可以使用C语言风格的注释(/* */)和C++语言风格的注释(//)。不过,最好能够使用C语言风格注释,否则可能在某些平台上编译时会出现错误。

2.2.3 GTK的Hello World

上面的例子实在是太简陋了,它甚至什么也不能做。下面我们创建一个Gtk版本的“Hello World”以进一步说明Gtk+的编程方法。在这个例子中,我们在窗口里面放一个按钮构件。下面是示例的源代码。

1. 源代码

```
/*示例开始 helloworld helloworld.c */
#include <gtk/gtk.h>
/*回调函数在本例中忽略了传递给程序的所有参数。下面是回调函数 */
void hello( GtkWidget *widget, gpointer data )
{
    g_print ("Hello World\n");
```

```

}
gint delete_event( GtkWidget *widget, GdkEvent *event, gpointer data )
{
/*如果在"delete_event"处理程序中返回FALSE, GTK 将引发一个"destroy"
*信号, 返回TRUE意味着你不想关闭窗口。
* 这些在弹出"你真的要退出?"对话框时很有作用 */
g_print ( "delete event occurred\n");
/* 将TRUE改为FALSE, 主窗口就会用一个"delete_event"信号, 然后退出 */
return(TRUE);
}
/* 另一个回调函数 */
void destroy( GtkWidget *widget, gpointer data )
{
gtk_main_quit();
}
int main( int argc, char *argv[] )
{
/* GtkWidget是构件的存储类型 */
GtkWidget *window;
GtkWidget *button;
/* 在所有的Gtk应用程序中都应该调用。它的作用是解析由命令行传递
* 进来的参数并将它返回给应用程序 */
gtk_init(&argc, &argv);
/* 创建一个主窗口 */
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
/* 当给窗口一个"delete_event"信号时(这个信号是由窗口管
* 理器发出的, 通常是在点击窗口标题条右边的 "x" 按钮, 或
* 者在标题条的快捷菜单上选择 "close" 选项时发出的), 我们
* 要求调用上面定义的delete_event()函数传递给这个回调函数
* 的数据是NULL, 回调函数会忽略这个参数 */
gtk_signal_connect (GTK_OBJECT (window), "delete_event",
GTK_SIGNAL_FUNC (delete_event), NULL);
/* 这里, 我们给 "destroy" 事件连接一个信号处理函数,
* 当我们在窗口上调用gtk_widget_destroy()函数
* 或者在 "delete_event" 事件的回调函数中返回 FALSE
* 时会发生这个事件 */
gtk_signal_connect (GTK_OBJECT (window), "destroy",
GTK_SIGNAL_FUNC (destroy), NULL);
/* 设置窗口的边框宽度 */
gtk_container_set_border_width (GTK_CONTAINER (window), 10);
/* 创建一个标题为 "Hello World" 的按钮 */
button = gtk_button_new_with_label ("Hello World");
/* 当按钮接收到 "clicked" 时, 它会调用 hello() 函数,
* 传递的参数为 NULL。函数 hello() 是在上面定义的 */
gtk_signal_connect (GTK_OBJECT (button), "clicked",
GTK_SIGNAL_FUNC (hello), NULL);
/* 当点击按钮时, 通过调用gtk_widget_destroy(window)函数销毁窗口。
* 另外, "destroy" 信号可以从这里发出, 也可以来自于窗口管理器 */
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",
GTK_SIGNAL_FUNC (gtk_widget_destroy),
GTK_OBJECT (window));

```

```
/* 将按钮组装到窗口中 (一个gtk容器中) */
gtk_container_add (GTK_CONTAINER (window), button);
/* 最后一步就是显示新创建的构件 */
gtk_widget_show (button);
/* 显示窗口 */
gtk_widget_show (window);
/* 所有的GTK应用程序都应该有一个gtk_main()函数。
* 程序的控制权停在这里并等着事件的发生 (比如一次按键或鼠标事件) */
gtk_main ();
return(0); }
/* 示例结束 */
```

2. 编译 “Hello World” 应用程序

用以下语法编译：

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

上面使用了Gtk自带的gtk-config程序。这个程序“知道”用Gtk编译应用程序时需要使用什么编译程序开关选项。

gtk-config --cflags将输出一个包含编译器需要的查找路径的列表，并且 gtk-config --libs将输出编译程序要链接的库的列表和找到这些库的路径。在上面的例子中，这些选项可以写在一句话里，比如`gtk-config --cflags --libs`。

编译中通常要链接的库包含下面这些：

GTK库 (-lgtk)：构件库,建立在Gdk的基础上。

GDK库 (-lgdk)：包装的Xlib库。

gmodule库 (-lgmodule)：用于在加载运行时扩展函数。

glib库 (-lglib)：包含了各种实用函数。在这里仅仅使用了 g_print()。Gtk是在Glib的基础上创建的，所以需要用到glib库。

Xlib库 (-lX11)：GDK库要调用该库。

Xext库 (-lXext)：包含为共享内存中的pixmap图片和其他X扩展的代码。

数学库 (-lm)：被GTK调用。

2.2.4 Gtk+的信号和回调函数原理

在详述helloworld的细节之前，我们先讨论信号和回调函数。GTK是一种事件驱动工具包，这意味着它将在gtk_main函数中一直等待，直到事件发生和控制权被传递给相应的函数。

1. 信号 (signal)

控制权的传递是使用“信号”的方法。一旦事件发生，比如鼠标器按钮被按下，被按下的构件（按钮）将引发适当的信号。这是GTK实现其绝大多数工作的方法。

有一些信号是大多数构件都具备的，比如 destroy，还有一些是某些构件专有的，比如在开关按钮（togglebutton）的toggled信号。

要让一个按钮执行一个操作，我们需要建立一段信号处理程序，以捕获它的信号，然后调用相应的函数。这由类似以下所示的函数实现：

```
gint gtk_signal_connect( GtkObject *object, gchar *name,
                        GtkSignalFunc func, gpointer func_data );
```

第一个参数 *object 是将要发出信号的构件，第二个参数 *name 是希望捕获的信号的名称，第三个参数 func 是捕获信号时要调用的函数，第四个参数 func_data 是要传递给函数的用户数据参数。

在第三个参数里指定的函数称为“回调函数”，它的形式通常是：

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

这个函数的第一个参数是一个指向发出信号的构件的指针，第二个参数是一个指向传递给回调函数的用户数据的指针。

注意 上述对“信号”的回调函数的声明仅仅是一个通用的规则，因为一些构件的特殊“信号”产生不同调用参数。例如，Clist构件的select_row构件同时提供行和列参数。

另一个在 helloworld 例子里使用的函数调用是：

```
gint gtk_signal_connect_object( GObject *object,
                                gchar *name,
                                GtkSignalFunc func,
                                GObject *slot_object );
```

gtk_signal_connect_object() 跟 gtk_signal_connect() 一样，除了回调函数仅仅使用一个参数：指向GTK对象的指针。

这样，当用这个函数连接到一个信号时，回调函数将以下面的形式出现：

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

在这里，参数 object 通常是一个构件。

不过，我们通常不为 gtk_signal_connect_object() 设置回调函数。它们通常用于调用接受信号或对象作为参数的 Gtk 函数，就像在 helloworld 里的那样。有两种函数能连接到信号的目的仅仅是允许回调函数可以有不同数量的参数。

因为在 GTK 库里许多函数仅仅接受单个 GtkWidget 指针作为参数，所以可能想用 gtk_signal_connect_object() 连接到某些回调函数。在这种情况下，需要提供附加的数据传递到回调函数。

2. 事件

除了上面描述的信号机制之外，还有一套与 X 事件（X Window 中发生的动作，比如鼠标某个按键按下或弹起，鼠标移动等）机制相对应的事件。回调函数也可以与这些事件连接起来。

这些事件是：

button_press_event	button_release_event
motion_notify_event	delete_event
destroy_event	expose_event
key_press_event	key_release_event
enter_notify_event	leave_notify_event
configure_event	focus_in_event
focus_out_event	map_event unmap

要将回调函数连接到上面的某一个事件，需要使用 gtk_signal_connect 函数，并使用上面的事件名称作为命名参数。事件的回调函数与信号的回调函数在形式上略有不同：

```
Void func(
    GtkWidget *widget, GdkEvent *event,
    gpointer callback_data );
```

GdkEvent是 C中的联合体结构，其类型依赖于发生的事件是哪一个事件。要想知道哪一个事件已经引发，可以查看类型参数，因为每个可能的可选事件都有一个反映引发事件的类型参数。

事件结构的另一个组件依赖于事件类型。事件类型的可能取值有：

GDK_NOHING	GDK_DELETE	GDK_DESTROY
GDK_EXPOSE	GDK_MOTION_NOTIFY	GDK_BUTTON_PRESS
GDK_2BUTTON_PRESS	GDK_3BUTTON_PRESS	GDK_BUTTON_RELEASE
GDK_KEY_PRESS	GDK_KEY_RELEASE	GDK_ENTER_NOTIFY
GDK_LEAVE_NOTIFY	GDK_FOCUS_CHANGE	GDK_CONFIGURE
GDK_MAP	GDK_UNMAP	GDK_PROPERTY_NOTIFY
GDK_SELECTION_CLEAR	GDK_SELECTION_REQUEST	GDK_SELECTION_NOTIFY
GDK_PROXIMITY_IN	GDK_PROXIMITY_OUT	GDK_DRAG_BEGIN
GDK_DRAG_REQUEST	GDK_DROP_ENTER	GDK_DROP_LEAVE
GDK_DROP_DATA_AVAIL	GDK_CLIENT_EVENT	GDK_VISIBILITY_NOTIFY
GDK_NO_EXPOSE		
GDK_OTHER_EVENT	/* 最好不要使用它 */	

因此，要将回调函数与一个事件连接起来，需要使用以下形式的函数：

```
gtk_signal_connect( GTK_OBJECT(button),
    "button_press_event",
    GTK_SIGNAL_FUNC(button_press_callback),NULL);
```

这里假定button是一个按钮构件。现在，当鼠标移动到按钮上方，鼠标按钮按下时，将调用button_press_callback函数。

这个回调函数可以作如下声明：

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

注意，我们可以将第二个参数声明为 GdkEventButton类型，因为对被调用的函数来说，它已经知道将发生什么类型的事件。

函数返回的值指示事件是否由 GTK的事件处理机制做进一步的传播。

返回TRUE指示事件已被处理，并且不会进一步传播。

返回FALSE将继续正常的事件处理。

2.2.5 Hello World代码解释

我们已解释了 Gtk+编程中的事件、信号以及回调函数的机制，下面我们将详细解释helloworld程序。

下面是当clicked按钮时调用的回调函数。在这个例子里，我们忽略了构件和参数，但是实际处理它们并不难。在下一个例子中应用数据参数将告诉我们按下了哪一个按钮。

```
void hello( GtkWidget *widget, gpointer data )
```



```
{  
    g_print ("Hello World\n");  
}
```

下一个回调函数有一点特殊。当窗口管理程序将事件传递给应用程序时，`delete_event`事件发生。在这里，我们有机会选择对这些事件做些什么动作。我们可以忽略它们，也可以做出一些响应，或简单地退出应用程序。

回调函数返回的值让 GTK 知道发生了什么动作。返回 `TRUE`，意味着不想引发 `destory` 信号，让应用程序继续运行。返回 `FALSE`，要求引发 `destory` 信号，然后调用 `destory` 的信号处理函数。

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

下面是另一个回调函数，通过调用 `gtk_main_quit()` 使应用程序退出。这个函数告诉 GTK 当控制被交回时将从 `gtk_main` 退出。

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

`main()` 是应用程序的入口。与其他任何 C 语言程序一样，该程序应该有一个 `main()` 函数作为入口。

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

下一步声明了两个指向 `GtkWidget` 类型结构的指针。它们用于创建一个窗口和一个按钮：

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

下面又是 `gtk_init` 函数。和以前介绍的一样，这个函数初始化 GTK，并且分析在命令行中传递进来的参数。命令行中传递过来的任何参数，只要是它能识别的，都会从列表中删除，并且修改 `argc` 和 `argv` 的值，就像这些参数从不存在一样，然后应用程序分析剩余的参数。

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

下面的语句创建新窗口。这是相当直接了当的，因为 `GtkWidget *window` 结构分配内存，现在它指向了一个有效的结构类型。到这里为止，已经创建了一个新窗口，但是直到在程序结束前调用 `gtk_widget_show(window)`，它才会显示出来。

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

下面是将对象（窗口）和信号处理函数连接起来的两个例子。在此处捕获了 `delete_event` 和 `destory` 两个信号。当我们用窗口管理程序关闭窗口时，或当我们在窗口的某个构件中调用 `gtk_widget_destroy()` 销毁窗口时，将引发第一个信号。在 `delete_event` 处理函数中返回 `FALSE` 时，将引发第二个信号。`GTK_OBJECT` 和 `GTK_SIGNAL_FUNC` 是用于执行类型转换和检查的宏，它们还提高了代码的可读性。

```
gtk_signal_connect (GTK_OBJECT (window), "delete_event",  
                   GTK_SIGNAL_FUNC (delete_event), NULL);  
gtk_signal_connect (GTK_OBJECT (window), "destory",  
                   GTK_SIGNAL_FUNC (destory), NULL);
```

下面这个函数用于设置放容器对象的属性。这里将设置窗口的属性，让它内部没有构件

占据的位置有一个宽度为 10 像素的空白区域。在这一节我们还会看到其他一些类似的设置构件属性的函数。其中 GTK_CONTAINER 是用于类型转换的宏。

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

下面的函数将创建一个新按钮。它在内存中为一个新的 GtkWidget 结构类型分配空间，然后对它进行初始化，并且让按钮指针指向它。按钮显示时，它上面显示 “Hello World” 标签。

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

在这里，我们创建了一个按钮，并且让它做点儿有用的事。我们在按钮上添加一个信号处理函数，当它引发 clicked 信号时，调用 hello() 函数。在这里我们不想向函数传递参数，因而我们简单传递一个 NULL 给 hello() 回调函数。很明显，当我们用鼠标点击按钮时就会引发 clicked 信号。

```
gtk_signal_connect (GTK_OBJECT (button), "clicked",  
                    GTK_SIGNAL_FUNC (hello), NULL);
```

我们也用这个按钮退出应用程序。这说明了 destroy 信号可以来自窗口管理程序，也可以来自于应用程序。

当按钮被按下后，同上面所述一样，它首先调用 hello() 回调函数，然后按设置的次序调用其他函数。根据需要，可以有多个回调函数。它们会依据它们的连接次序依次执行。因为 gtk_widget_destroy() 函数仅仅接受 GtkWidget *widget 作为参数，在这里我们用 gtk_signal_connect_object() 函数代替 gtk_signal_connect()。

```
gtk_signal_connect_object (GTK_OBJECT (button), "clicked",  
                           GTK_SIGNAL_FUNC (gtk_widget_destroy),  
                           GTK_OBJECT (window));
```

下面的函数调用是一个组装调用，后面将会专门解释。它是相当容易理解的。它简单地告诉 GTK，按钮应该放置在窗口中，并且可以在窗口中显示。要注意，一个 GTK 容器构件仅仅能容纳一个子构件。还有一些其他构件，它们作为容器可以用多种方法容纳多个构件。

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

现在，我们已经拥有了创建一个应用程序所需要的所有方法。当设置了全部信号处理函数，并且将按钮放在窗口中恰当的地方后，我们要求 GTK 在屏幕上 “显示” 所有的构件。最好让窗口构件最后显示，这样整个窗口 - 包括里面的所有构件 - 将一起弹出来，而不是窗口先弹出来，然后窗口内的按钮再显示出来。不过，在这个简单的例子里，很难注意到这种区别。

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

然后，我们调用 gtk_main()，开始等候来自 X 服务器的事件发生。这些事件的发生，会使某个构件引发信号。

```
gtk_main ();
```

最后是 main() 函数的返回值。当调用 gtk_quit() 时，控制返回到这里。

```
return (0);
```

现在，当我们在 GTK 按钮上点击鼠标按键时，构件引发 clicked 信号。要使用这些信息，我们的程序中应设置一个信号处理函数以捕获这个信号。在本例子里，当我们创建的按钮被

“点击”时，将调用 `hello()` 函数，并给它传递一个 `NULL` 参数，然后调用这个信号的下一个处理函数。调用 `gtk_widget_destroy()` 函数，将窗口构件作为它的参数，运行结果是销毁窗口构件。这导致窗口引发 `destroy` 信号，当这个信号被捕获时，调用 `destroy()` 回调函数，简单退出 GTK。

另一种方法是用窗口管理程序关闭窗口时，会引发 `delete_event` 事件。它调用 `delete_event` 处理函数。如果返回 `TRUE`，将保留窗口，就像什么也没有发生一样。返回 `FALSE` 将导致 GTK 引发 `destroy` 信号，调用 `destroy` 信号的回调函数，退出 GTK。

2.2.6 运行helloworld

上面介绍了 `helloworld` 中代码的含义。编译完成后，在应用程序所在目录下输入以下命令，运行 `helloworld`（结果如图 2-2 所示）：

```
./helloworld
```

尝试一下，将鼠标放在窗口的边框处，按下鼠标左键，拖动鼠标以改变窗口大小，按钮会随之而改变大小。点击“Hello World”按钮会退出应用程序。



图2-2 Gtk 版本的“Hello World”

2.3 Gnome应用程序

上面我们通过两个例子介绍了使用 Gtk+ 库创建 Linux 应用程序的步骤。实际上，如果要用到 Gnome 库，在代码中只有一些细微的区别。

首先，应该在代码的头部包含 `gnome.h` 而不是 `gtk.h`：

```
#include <gnome.h>
```

`gnome.h` 头文件中已经包含了 `gtk.h` 文件。

其次，应该用 `gnome_init()` 函数替换 `gtk_init()` 函数。`Gnome_init()` 的声明如下：

```
gnome_init(const char* app_id,  
           const char* app_version,  
           int argc,  
           char** argv)
```

`gnome_init()` 函数会在内部调用 `gtk_init()`。`gnome_init()` 函数的第一个参数是应用程序的名称，第二个参数是代表应用程序版本的字符串。这些参数是 Gnome 库内部使用的（例如，由参数分析程序提供的一些缺省信息）。

与 `gtk_init()` 函数一样，`gnome_init()` 分析命令行参数；但是与 `gtk_init()` 不一样的是，它不会改变 `argc` 和 `argv` 的值。如果想分析特定的选项，应该使用 `gnome_init_with_popt_table()` 函数。我们会在后面的内容中介绍这个函数。

如果初始化失败，`gnome_init()` 函数的返回值应该是一个非 0 的值，但在当前的 Gnome 版本中它总返回 0（如果有什么问题，比如说没有找到 X 服务器，`gnome_init()` 会简单地异常终止）。通常的惯例是忽略它的返回值，至少在 Gnome 1.0 中是这样的，但是，无论如何，检查它的返回值是一个好主意，这样可以防止未来的 Gnome 版本返回一个错误。

最后，在 Gnome 应用程序中所有用户可见的字符串都应该为翻译做标记。翻译是用 GNU 的 `gettext` 实用程序完成的，这称为“国际化”。在程序的开头，必须调用 `bindtextdomain()` 和

textdomain() 函数。这两个函数的调用形式如下：

```
bindtextdomain (PACKAGE, PACKAGE_LOCALE_DIR);
textdomain (PACKAGE);
```

对国际化问题，我们将在后面另辟专题进行介绍。

除了上面的几点以外，使用 Gnome库和Gtk库没有什么两样，比如创建构件、连接信号和回调函数、主循环等等。

2.4 GNU C 编译器

目前最常用的GNU C编译器(gcc)是一个全功能的ANSI C兼容编译器。如果熟悉其他操作系统或硬件平台上的一种C编译器，将能很快地掌握 gcc。gcc可以用于编译C语言和用C++语言编的代码，不管它使用的是什么函数库和构件库。下面简要介绍如何使用 gcc 和一些 gcc 编译器最常用的选项。

2.4.1 使用 gcc

通常后跟一些选项和文件名来使用 gcc编译器。gcc 命令的基本用法如下：

```
gcc [options] [filenames]
```

命令行选项指定的操作将在命令行上每个给出的文件上执行。下面将介绍一些最常用到的选项。

2.4.2 gcc 选项

gcc 有超过100个的编译选项可用。这些选项中的许多选项一般根本不会用到，但一些主要的选项将会频繁用到。很多的 gcc选项包括一个以上的字符。因此必须为每个选项指定各自的连字符，并且就像大多数 Linux命令一样，不能在一个单独的连字符后跟一组选项。例如，下面的两个命令是不同的：

```
gcc -p -g test.c
gcc -pg test.c
```

第一条命令告诉 gcc编译test.c时为prof命令建立剖析信息并且把调试信息加入到可执行的文件里。第二条命令只告诉 gcc为gprof命令建立剖析信息。

如果编译一个程序时不使用任何选项， gcc 将会建立(假定编译成功)一个名为 a.out 的可执行文件。例如，下面的命令将在当前目录下产生一个名为 a.out的文件：

```
gcc test.c
```

可以使用-o编译选项来为即将产生的可执行文件指定一个文件名来代替 a.out。例如，将一个叫count.c的C程序编译为名叫count的可执行文件，可以输入下面的命令：

```
gcc -o count count.c
```

gcc 同样有指定编译器处理多少的编译选项。-c选项告诉 gcc 仅把源代码编译为目标代码而跳过汇编和连接的步骤。这个选项使用得非常频繁，因为它使编译多个 C程序时的速度更快并且更易于管理。缺省时 gcc建立的目标代码文件有一个.o的扩展名。

-S 编译选项告诉 gcc在为C代码产生了汇编语言文件后停止编译。 gcc 产生的汇编语言文件的缺省扩展名是.s。-E选项指示编译器仅对输入文件进行预处理。当使用这个选项时，预处

理器的输出被送到标准输出而不是储存在文件里。

1. 优化选项

用gcc编译C代码时，它会尝试用最少的时间完成编译并且使编译后的代码易于调试。易于调试意味着编译后的代码与源代码有同样的执行次序，编译后的代码没有经过优化。有很多选项可用于告诉 gcc 在耗费更多编译时间和牺牲易调试性的基础上产生更小更快的可执行文件。这些选项中最典型的是 -O和-O2选项。

-O选项告诉gcc对源代码进行基本优化。这些优化在大多数情况下都会使程序执行得更快。-O2选项告诉gcc产生尽可能小和尽可能快的代码。-O2选项将使编译的速度比使用-O时慢。但通常产生的代码执行速度会更快。

除了-O和-O2优化选项外，还有一些低级选项用于产生更快的代码。这些选项非常特殊，而且最好只有完全理解这些选项将会对编译后的代码产生什么样的效果时再去使用它们。

2. 调试和剖析选项

gcc 支持数种调试和剖析选项。在这些选项里最常用到的是 -g和-pg选项。

-g选项告诉gcc产生能被GNU调试器gdb使用的调试信息以便调试程序。gcc提供了一个很多其他C编译器里没有的特性，在gcc里可以将-g和-O (产生优化代码)联用。这一点非常有用，因为这样可以在与最终产品尽可能相近的情况下调试代码。在同时使用这两个选项时必须清楚所写的某些代码已经在优化时被gcc做了改动。

-pg选项告诉gcc在程序里加入额外的代码，执行时，将产生 gprof用的剖析信息以显示程序的耗时情况。

上面介绍的都是关于 gcc的最简单的知识。如果想详细了解 gcc，请参考GCC-HOWTO，或者在shell提示符下输入：

```
man gcc
```

这样可以浏览gcc的手册页。

当然，对规模较大的程序来说，仅仅一行编译指令是远远不够的。有多种 GNU实用工具可以帮助完成编译选项的设置，如 autoconf、automake和libtool等。GUI生成器Glade所生成的代码中有一个autogen.sh脚本，也可以用来完成这件工作。

2.5 初始化库

前面已经介绍过，在程序的 main函数一开始，应用程序必须调用 gtk_init()函数初始化 Gtk+库。对Gnome应用程序，要用gnome_init()函数代替gtk_init()函数(gnome_init()会在内部调用gtk_init())。

gnome_init()函数的第一个参数是应用程序的名称，第二个参数是代表应用程序版本的字符串。这些参数是Gnome库内部使用的(例如，由参数分析程序提供一些缺省信息)。

函数列表：初始化Gnome

```
#include <libgnomeui/gnome-init.h>
int gnome_init(const char* app_id,
               const char* app_version,
               int argc,
               char** argv)
```

与gtk_init()函数一样，gnome_init()分析命令行参数；但是与gtk_init()不一样的是，它不

会改变argc和argv的值。如果想分析特定的选项，应该使用 `gnome_init_with_popt_table()` 函数。

如果初始化失败，`gnome_init()` 函数的返回值应该是一个非 0 的值，但目前它总返回 0 (如果有什么问题，比如说没有找到 X 服务器，`gnome_init()` 会简单地异常终止)。通常的习惯是忽略它的返回值。但是，无论如何，检查它的返回值是一个好主意，这样可以防止未来的 Gnome 版本产生错误。

2.6 用popt分析参数

对基于 Gnome 的应用程序，如果允许用户从命令行带参数启动，应该在初始化的时候对传递到应用程序的参数和选项进行分析。

2.6.1 参数分析方法

Gnome 使用一个强劲的选项分析库 `popt` 来实现这一点。`popt` 处理所有的缺省 Gnome 选项。要查看 Gnome 的缺省选项，只需将 `--help` 选项传递给任何 Gnome 应用程序即可。还可以用定制的选项添加一个 “`popt` 表”。要做到这一点，用 `gnome_init_with_popt_table()` 函数代替 `gnome_init()` 函数。

函数列表：初始化库，进行参数分析

```
#include <libgnomeui/gnome-init.h>
int gnome_init_with_popt_table(const char* app_id,
                               const char* app_version,
                               int argc,
                               char** argv,
                               const struct poptOption* options,
                               int flags,
                               poptContext* return_ctx)
```

`popt`表是一个 `poptOption` 结构数组。下面是其定义：

```
struct poptOption {
    const char* longName;
    char shortName;
    int argInfo;
    void* arg;
    int val;
    char* descrip;
    char* argDescrip;
};
```

下面解释 `poptOption` 结构中各成员的含义。

前面的两个部分 `longName` 和 `shortName` 是选项的长名字和短名字，例如 “`help`” 和 “`h`” 对应于命令行选项的 `help` 和 `-h`。如果只想有一个选项名，它们可以设为 `NULL` 和 “`\0`”。

第三个成员 `arginfo` 告知表输入项是什么类型。可以是以下几种取值：

- `POPT_ARG_NONE` 表明选项只是一个简单的开关，它没有参数。
- `POPT_ARG_STRING` 表明选项有一个字符串参数，比如说 `--geometry="300 x 300+50+100"`。
- `POPT_ARG_INT` 表明选项带一个整数的参数，例如 `--columns=10`。

- POPT_ARG_LONG 表明选项带一个长整型的参数。
- POPT_ARG_INCLUDE_TABLE表明poptOption结构不指定一个选项，但是要包含另一个popt表。
- POPT_ARG_CALLBACK表明poptOption结构并不指定选项，而是一个分析表中选项的回调函数。入口种类应该在表的开头。
- POPT_ARG_INTL_DOMAIN表明poptOption结构指定这个表和任何子表的翻译范围。

arg的意义依赖于arginfo成员。对一个带参数的选项来说，arg应该指向一个参数类型的变量。popt会用参数填充所指向的变量。对 POPT_ARG_NONE，如果选项能在命令行上找到，*arg设置为TRUE。对所有情况，arg都可以设置为NULL，popt会忽略它。

对POPT_ARG_INCLUDE_TABLE，arg指向所包含的表；对 POPT_ARG_CALLBACK，它指向要调用的回调函数；对 POPT_ARG_INTL_DOMAIN，它应该是翻译范围字符串。

val成员是每个成员的标识符。它在 Gnome应用程序中一般没有什么用，但是如果使用一个回调函数，则它在回调函数中是有用的。如果不想用它，把它设置为 0。

最后两个成员用于对 --help选项自动生成输出信息。descrip用于描述选项；argDescrip 用于描述选项的参数。例如，--display选项的帮助是这个样子：

```
--display=DISPLAY          X display to use
```

在这里，argDescrip是“DISPLAY”；descrip是“X display to use”。要用_宏为这两个字符串标记以便翻译。

对POPT_ARG_INCLUDE_TABLE，descrip意义略有不同。在这种情况下，它在帮助输出中为一“组”选项加标题。例如，在下面输出中的“帮助选项”：

帮助选项

-?, --help 显示帮助信息

--usage 显示简要的用法消息

如果在popt表的开头放一个POPT_ARG_CALLBACK 类型条目，会对命令行中的每个选项的信息调用一个用户定义的回调函数。下面是回调函数应该有的类型：

```
typedef void (*poptCallbackType)(poptContext con,
                                  enum poptCallbackReason reason,
                                  const struct poptOption* opt,
                                  const char* arg,
                                  void* data);
```

poptContext对象是不透明的，它包含所有的 popt状态。这样就有可能在同一个程序中多次使用popt，或者同时分析一套以上的选项。还可以用由 popt提供的函数从popyContext中提取出当前分析状态的信息。

poptCallbackReason可取以下值：

- POPT_CALLBACK_REASON_PRE
- POPT_CALLBACK_REASON_POST
- POPT_CALLBACK_REASON_OPTION

回调函数将POPT_CALLBACK_REASON_OPTION 作为“reson”参数，对命令行中的每个选项调用一次。

根据要求，也可以在参数分析之前或之后调用。在这些情况下，“reson”参数会是

POPT_CALLBACK_REASON_PRE或 POPT_CALLBACK_REASON_POST。要指定回调函数在参数分析之前或之后调用，必须将上面两个标志与 POPT_ARG_CALLBACK 结合起来使用。例如，下面的poptOption结构初始化程序指定在参数分析之前和之后都调用回调函数：

```
{ NULL, '\0', POPT_ARG_CALLBACK|POPT_CBFLAG_PRE|POPT_CBFLAG_POST,
  &parse_an_arg_callback, 0, NULL}
```

回调函数的opt参数就是对应于最近看见的命令行选项的 poptOption结构。可以访问这个结构的val成员以判定只进行查找的选项是哪一个。arg参数是传递到命令行选项的任何参数的原文；data参数是回调函数的数据，这个数据就是在回调函数里指定的 popyOption结构的descrip成员。

在Gnome环境中gnome_init_with_popt_table()的flags参数基本上可以忽略掉，这个“flags”用处不大。

如果给gnome_init_with_popt_table()函数的最后一个参数return_ctx传递一个非空的指针，会返回当前的环境。可以用这个环境来提取那些不是选项的部分，比如文件名。这是使用poptGetArgs()完成的。下面是一个例子：

```
char** args;
poptContext ctx;
int i;

bindtextdomain (PACKAGE, GNOMELOCALEDIR);
textdomain (PACKAGE);
gnome_init_with_popt_table(APPNAME, VERSION, argc, argv,
                          options, 0, &ctx);

args = poptGetArgs(ctx);

if (args != NULL)
{
    i = 0;
    while (args[i] != NULL)
    {
        /* Do something with each argument */
        ++i;
    }
}

poptFreeContext(ctx);
```

注意，必须释放前面使用过的 poptContext变量。然而，如果对 return_ctx传递一个NULL值，库函数会释放它。还要记住，如果命令行没有参数，poptGetArgs()会返回NULL。

2.6.2 GnomeHello程序的参数分析

下面的示例代码来自于一个名为 GnomeHello的程序。这个程序中集中演示了 Gnome编程的各种技巧。GnomeHello的源代码见附录。

如果带有help启动程序，GnomeHello将输出下面的信息：

```
$ ./hello --help
```



```

Usage: hello [OPTION...]

GNOME Options
  --disable-sound           Disable sound server usage
  --enable-sound           Enable sound server usage
  --espeaker=HOSTNAME:PORT Host:port on which the sound server to use is
                           running

Help options
  -?, --help               Show this help message
  --usage                  Display brief usage message

GTK options
  --gdk-debug=FLAGS        Gdk debugging flags to set
  --gdk-no-debug=FLAGS     Gdk debugging flags to unset
  --display=DISPLAY        X display to use
  --sync                   Make X calls synchronous
  --no-xshm                Disable X shared memory extension
  --name=NAME              Program name as used by the window manager
  --class=CLASS            Program class as used by the window manager
  --gxid_host=HOST
  --gxid_port=PORT
  --xim-preedit=STYLE
  --xim-status=STYLE
  --gtk-debug=FLAGS        Gtk+ debugging flags to set
  --gtk-no-debug=FLAGS     Gtk+ debugging flags to unset
  --g-fatal-warnings       Make all warnings fatal
  --gtk-module=MODULE      Load an additional Gtk module

GNOME GUI options
  -V, --version

Help options
  -?, --help               Show this help message
  --usage                  Display brief usage message

Session management options
  --sm-client-id=ID        Specify session management ID
  --sm-config-prefix=PREFIX Specify prefix of saved configuration
  --sm-disable             Disable connection to session manager

GnomeHello options
  -g, --greet              Say hello to specific people listed on the
                           command line
  -m, --message=MESSAGE    Specify a message other than "Hello, World!"
  --geometry=GEOMETRY       Specify the geometry of the main window
$

```

对所有Gnome应用程序来说，所有这些选项差不多都是相同的，只有最后三个，标为“GnomeHello options”的是GnomeHello特有的。--greet或-g选项打开“问候模式”；GnomeHello期望在命令行上输入姓名列表，创建一个对话框向输入的每一个名字打招呼。--message选项期望输入一个字符串参数代替常见的“Hello, World!”消息；--geometry

选项期待一个标准的 X 几何字符串，指定主窗口的尺寸和位置。

下面是 GnomeHello 用作参数分析的变量和 popt 表：

```
static int greet_mode = FALSE;
static char* message = NULL;
static char* geometry = NULL;
struct poptOption options[] = {
    {
        "greet",
        'g',
        POPT_ARG_NONE,
        &greet_mode,
        0,
        N_("Say hello to specific people listed on the command line"),
        NULL
    },
    {
        "message",
        'm',
        POPT_ARG_STRING,
        &message,
        0,
        N_("Specify a message other than \"Hello, World!\""),
        N_("MESSAGE")
    },
    {
        "geometry",
        '\0',
        POPT_ARG_STRING,
        &geometry,
        0,
        N_("Specify the geometry of the main window"),
        N_("GEOMETRY")
    },
    {
        NULL,
        '\0',
        0,
        NULL,
        0,
        NULL,
        NULL
    }
};
```

下面是 main() 的第一部分，在这里 GnomeHello 检验参数是否已经正确组合并装配成一个要欢迎的人员名单列表：

```
GtkWidget* app;
poptContext pctx;
char** args;
int i;
```

```
GSList* greet = NULL;
GnomeClient* client;

bindtextdomain(PACKAGE, GNOMELOCALEDIR);
textdomain(PACKAGE);

gnome_init_with_popt_table(PACKAGE, VERSION, argc, argv,
                           options, 0, &pctx);

/* Argument parsing */

args = poptGetArgs(pctx);

if (greet_mode && args)
{
    i = 0;
    while (args[i] != NULL)
    {
        greet = g_slist_prepend(greet, args[i]);
        ++i;
    }
    /* Put them in order */
    greet = g_slist_reverse(greet);
}
else if (greet_mode && args == NULL)
{
    g_error(_("You must specify someone to greet."));
}
else if (args != NULL)
{
    g_error(_("Command line arguments are only allowed with --greet."));
}
else
{
    g_assert(!greet_mode && args == NULL);
}

poptFreeContext(pctx);
```

2.7 国际化

在Gnome应用程序中所有用户可见的字符串都应该为翻译做标记。翻译是用 GNU的gettext实用程序完成的。gettext只是一个简单的消息分类，它存储了键/值对，键是程序的硬编码字符串，值是翻译过的字符串（如果有的话），或者只有键（如果没有翻译，或者键的语言已经是正确的）。

作为程序员，不一定要提供软件的多国语言版本。不过，强烈建议将其中的字符串作为翻译标记，这样，gettext脚本能够提取出一个要翻译的字符串表。程序的用户可以根据他的实际情况决定是否编译一个对应于当前语言的版本。

宏列表：翻译宏

```
#include <libgnome/gnome-i18n.h>
_(string)
N_(string)
```

Gnome使用上面的两个宏来实现翻译标记。宏 `_()` 对字符串做翻译标记的同时还进行信息类别查找。应该在任何 C 准许函数调用的环境中使用它。宏 `N_()` 不做后一项操作，仅仅是为字符串做翻译标记。可以在不允许进行函数调用的场合使用它，例如在静态数组初始化程序中。如果使用宏 `N_()` 为字符串做翻译标记，最后必须对它调用宏 `_()` 以做实际的查找。

下面是一个简单的例子：

```
#include <gnome.h>
static char* a[] =
    N_("Translate Me"),
    N_("Me Too")
};

int main(int argc, char** argv)
{
    bindtextdomain(PACKAGE, GNOMELOCALEDIR);
    textdomain(PACKAGE);
    printf(_("Translated String\n"));
    printf(_(a[0]));
    printf(_(a[1]));
    return 0;
}
```

注意到字符串“Translate Me”和“Me Too”已经做了标志，这样，`gettext`就能发现它们，并产生一个要翻译的字符串列表。翻译程序将用这个表生成实际的翻译字符串。以后，宏 `_()` 包含了一个函数调用，用以对数组的每个成员进行翻译查找。因为当字符串文字“Translated String”引入时允许进行函数调用，所有的工作都是在一步内完成的。

在程序的开头，必须调用 `bindtextdomain()` 和 `textdomain()` 函数，如同上面的例子显示的那样。在上面的代码中，`PACKAGE` 是一个字符串，它代表程序找到的字符串包，典型情况下它是在 `config.h` 中定义的。还必须定义 `GNOMELOCALEDIR` 目录，典型情况下它是在 `Makefile.am` 中定义的（标准值应该是 `$(prefix)/share/locale`，或 `$(datadir)/locale`）。翻译是存储在 `GNOMELOCALEDIR` 路径下的。

当用字符串为翻译做标记时，必须保证字符串是可翻译的。要避免在运行时通过连接多个字符串生成一个字符串。例如，不要像下面这样做：

```
gchar* message = g_strconcat(_("There is an error on device "),
                               device, NULL);
```

问题是：在一些语言中，应该将设备名称放在前面（或放在中间）。如果使用 `g_snprintf()` 或者 `g_strdup_printf()` 函数而不是字符串连接函数，翻译程序就能够改变单词的次序。下面就是正确的方法：

```
gchar* message = g_strdup_printf(_("There is an error on device %s"),
                                   device);
```

现在，翻译程序能够根据需要移动 `%s` 的位置。

应该尽可能避免复杂的语法。例如，要翻译下面的字符串就很困难：

```
printf(_("There %s %d dog%s\n"),
        n_dogs > 1 ? _("were") : _("was"),
        n_dogs,
        n_dogs > 1 ? _("s") : "");
```

最好将条件移动到printf()的外面：

```
if (n_dogs > 0)
    printf(_("There were %d dogs\n"), n_dogs);
else
    printf(_("There was 1 dog\n"));
```

然而，即使这样也不一定能够正常工作。一些语言还有比“正好一个”和“一个以上”更多的分类(也就是，在英语的单复数形式以外，它们还有“正好两个”的说法)。可以使用以下做法：

```
static const char* ndogs_phrases[] = {
    N_("There were no dogs.\n"),
    N_("There was one dog.\n"),
    N_("There were two dogs.\n"),
    N_("There were three dogs.\n")
};
```

可是这样处理也太麻烦了。如果可能应该尽量避免出现这种情况。

当分析和显示特定种类的数据(包括日期和十进制数)时，也必须考虑国际化。通常，C库函数提供了足够的实用函数以处理这些情况，使用strftime()、strcoll()等函数处理这种情况。GlibGDate的实用函数在内部也是用strftime()函数处理日期数据的。

应避免的常见错误：当读和写文件时，不要使用依赖于特定地区的函数。例如，printf()和scanf()函数根据地区调整它们的小数位格式，所以不能在文件中使用这种格式。如果这样，欧洲的用户将不能读出在美国常见的文件。

2.8 保存配置信息

有时候，需要保留一些应用程序的配置信息。例如，在文件菜单里保存一个“最近打开的文件”列表，应用程序是否显示工具条、状态条等。这些功能都可以使用Gnome API函数实现。不过，有些设置值，比如上次窗口打开的位置、尺寸等，一般不这么处理，而是使用会话管理功能实现。

libgnome库有在普通文本配置文件里保存简单的键/值对的能力。提供的实用程序能够处理数值型和布尔型的值，可以透明地将变量值转换为文本文件，或者相反，从文本文件读出。Gnome配置文件的标准位置是~/.gnome，库函数文件用这个位置作为缺省位置。不过，库函数也可以使用任何其他文件。每个函数还有相应的变体函数，它们能将配置文件存到~/.gnome_private下，这个目录具有用户权限设置。通常也将这个libgnome模块称为gnome-config。不要将这个gnome-config与gnome-config脚本混淆，后者是Gnome程序用来报告编译和链接标志的。

gnome-config函数用路径作为参数。路径由三部分组成：

- 要用到的文件名，在~/.gnome或~/.gnome_private目录下。按惯例是应用程序的名字。
- 一个节，相关配置信息的逻辑子类。
- 一个键，键/值对的一半。键实际上是与一块配置数据相联系的。

路径作为一个字符串传给Gnome，并使用“/filename/section/key”的形式。如果想用一个不在标准Gnome目录下的文件名，可以将整个路径用“=”隔开，它将被解释为绝对路径。甚至可以将配置文件作为简单的数据文件格式（可以用作.desktop文件——设置在里面的文件会出现在Gnome的主菜单上）。然而，XML（也许要使用gnome-xml软件包）才是这种情况的更好的选择。对存储某些种类的配置信息，XML也许是一个更好的选择，libgnome配置函数库的主要优点就是简单。

gnome-config已经有很长的历史了，它最初是为WINE——Windows仿真器项目写的，然后用在GNU Midnight Commander文件管理器上，最后移植到Gnome库上。当前的计划是在下一个Gnome版本中用更强劲的库函数取代gnome-config，主要想支持按主机配置、LDAP（轻量级目录存取协议）后端，以及一些其他特性。然而，即使下层的引擎（函数库）发生剧烈变化，gnome-config API函数也总是会得到支持的。

2.8.1 读出存储的配置数据

从文件中获得数据很简单，只要调用一个函数获取给定键的值就可以了。取值的函数接受一个路径作为参数。例如，你可能询问用户是否想要看到一个对话框：

```
gboolean show_dialog;  
show_dialog =  
    gnome_config_get_bool("/myapp/General/dialog");
```

如果配置文件还不存在，或没有键名与你提供的键匹配，函数返回 0、FALSE或NULL。返回字符串的函数会返回一段分配的内存，应该用 g_free()函数将字符串释放。字符串矢量函数返回一个已分配空间的字符串数组（释放该矢量的最容易的方法是调用 g_strfreev()函数）。

如果给定键不存在，可以指定一个缺省值，在路径后加一个“=value”。例如：

```
gboolean show_dialog;  
show_dialog =  
    gnome_config_get_bool("/myapp/General/dialog=true");
```

每个函数都有一个 with_default式样的变体，这些函数告诉你返回的值是从配置文件取得的还是从指定的缺省值取得的。例如：

```
gboolean show_dialog;  
gboolean used_default;  
show_dialog =  
    gnome_config_get_bool_with_default("/myapp/General/dialog=true",  
                                       &used_default);  
  
if (used_default)  
    printf("Default value used for show_dialog\n");
```

gnome_config_push_prefix()和gnome_config_pop_prefix()函数可以用于避免每次都要指定整个路径。例如：

```
gboolean show_dialog;  
gnome_config_push_prefix("/myapp/General/");  
show_dialog = gnome_config_get_bool("dialog=true");  
gnome_config_pop_prefix();
```

这些函数在保存值时也起作用。

在名字中带一个private的配置函数使用具有权限限制的.gnome_private目录，如上面所讨论的。带translated_string后缀的函数限定用当前场所的名字限定给定的键，这些函数一般用

函数列表：从配置文件获取数据

[illegible]


```
gchar*** argvp,  
gboolean* was_default)
```

2.8.2 在配置文件中存储数据

保存数据是与获取数据相反的过程，用同样的方法给出一个路径“/file/section/key”，连带要存储的值。数据并不是立即写入的，必须调用 `gnome_config_sync()` 函数以保证文件写到磁盘上了。

函数列表：保存数据到配置文件

```
#include <libgnome/gnome-config.h>  
void gnome_config_set_string(const gchar* path,  
                             const gchar* value)  
void gnome_config_set_translated_string(const gchar* path,  
                                         const gchar* value)  
  
void gnome_config_set_int(const gchar* path,  
                          gint value)  
  
void gnome_config_set_float(const gchar* path,  
                            gdouble value)  
  
void gnome_config_set_bool(const gchar* path,  
                           gboolean value)  
  
void gnome_config_set_vector(const gchar* path,  
                             gint argc,  
                             const gchar* const argv[])  
  
void gnome_config_private_set_string(const gchar* path,  
                                     const gchar* value)  
  
void gnome_config_private_set_translated_string(const gchar* path,  
                                                const gchar* value)  
  
void gnome_config_private_set_int(const gchar* path,  
                                  gint value)  
  
void gnome_config_private_set_float(const gchar* path,  
                                    gdouble value)  
  
void gnome_config_private_set_bool(const gchar* path,  
                                   gboolean value)  
  
void gnome_config_private_set_vector(const gchar* path,  
                                     gint argc,  
                                     const gchar* const argv[])
```

2.8.3 配置文件迭代器

迭代器用于在给定文件中扫描节，或在给定节中扫描键。应用程序可以用这个特性存储

数据列表。具体实现方法是通过动态生成键或节的名字以保存数据，随后迭代它们以检查保存了些什么。

迭代器是一种不透明的数据类型；可以将“节”的名称传给 `gnome_config_init_iterator()` 函数，迭代一遍，并依次接收一个迭代器。然后调用 `gnome_config_iterator_next()` 函数从该节中获得键/值对。从 `gnome_config_iterator_next()` 函数返回的键/值对必须用 `g_free()` 函数释放，`gnome_config_iterator_next()` 函数返回的值是一个指向下一个迭代器的指针。当函数 `gnome_config_iterator_next()` 返回 NULL 时，说明已经遍历了所有的键/值对。

gnome-apt程序中的迭代示例

下面是一个来自 `gnome-apt` 的迭代示例，在 Debian 发布版本中用来管理软件包的 C++ 应用程序。`gnome-apt` 在一个树状视图中保存和加载一些栏的位置。栏是用 `GAptPkgTree::ColumnType` 枚举类型标识的。`GAptPkgTree::ColumnTypeEnd` 是栏枚举类型的最后一个元素，它等于有效栏类型的数目。

```
static void
load_column_order(vector<GAptPkgTree::ColumnType> & columns)
{
    gpointer config_iterator;
    guint loaded = 0;

    config_iterator = gnome_config_init_iterator("/gnome-apt/ColumnOrder");

    if (config_iterator != 0)
    {
        gchar * col, * pos;
        columns.reserve(GAptPkgTree::ColumnTypeEnd);

        loaded = 0;
        while ((config_iterator =
                gnome_config_iterator_next(config_iterator,
                                           &col, &pos)))
        {
            // shouldn't happen, but 'm paranoid
            if (pos == 0 || col == 0)
            {
                if (pos) g_free(pos);
                if (col) g_free(col);
                continue;
            }

            GAptPkgTree::ColumnType ct = string_to_column(col);

            gint index = atoi(pos);

            g_free(pos); pos = 0;
            g_free(col); col = 0;

            // the user could mangle the config file to make this happen
            if (static_cast<guint>(index) >= columns.size())
```

```

        continue;

        columns[index] = ct;

        ++loaded;
    }
}

if (loaded != static_cast<guint>(GAptPkgTree::ColumnTypeEnd))
{
    // Either there was no saved order, or something is busted - use
    // default order
    columns.clear();

    int i = 0;
    while (i < GAptPkgTree::ColumnTypeEnd)
    {
        columns.push_back(static_cast<GAptPkgTree::ColumnType>(i));
        ++i;
    }

    // Clean the section - otherwise an old entry could
    // remain forever and keep screwing us up in the future.
    gnome_config_clean_section("/gnome-apt/ColumnOrder");
    gnome_config_sync();
}

g_return_if_fail(columns.size() ==
                  static_cast<guint>(GAptPkgTree::ColumnTypeEnd));
}

```

下面是用于保存“列”位置的函数：

```

static void
save_column_order(const vector<GAptPkgTree::ColumnType> & columns)
{
    g_return_if_fail(columns.size() ==
                      static_cast<guint>(GAptPkgTree::ColumnTypeEnd));

    int position = 0;
    vector<GAptPkgTree::ColumnType>::const_iterator i = columns.begin();
    while (i != columns.end())
    {
        gchar key[256];
        g_snprintf(key, 255, "/gnome-apt/ColumnOrder/%s", column_to_string(*i));
        gchar val[30];
        g_snprintf(val, 29, "%d", position);
        gnome_config_set_string(key, val);

        ++position;
        ++i;
    }
}

```

```
}  
  
    gnome_config_sync();  
}
```

在这段代码中，将枚举值保存为字符串而不是整数值。`column_to_string()`和`string_to_column()`函数使用了一个简单的、由枚举值索引的栏名数组，用于来回转换。这么做有两个理由：在程序的未来版本中枚举值发生变化时，代码不会因此而发生故障，同时，它还使配置文件可以人工修改。

栏位置是用`gnome_config_set_string()`而不是用`gnome_config_set_int()`存储的。这是因为`gnome_config_iterator_next()`返回一个代表存储信息的字符串。更可能的情况是：`gnome_config_set_int()`函数将整数存储为`atoi()`函数能理解的字符串（它确实能理解），但是从技术上来说，API函数并未保证这一点。如果代码中使用了`gnome_config_set_int()`函数，它将从`gnome_config_iterator_next()`函数获得一个唯一的键，然后调用`gnome_config_get_int()`函数获得整数值。对获得的字符串值使用`atoi()`函数会对`gnome-config`的实现产生没有根据的假设。

2.8.4 节迭代器

`gnome_config_init_iterator_sections()`允许在一个文件中迭代所有的节，而不是在一节中迭代键。迭代节时，`gnome_config_iterator_next()`函数忽略它的值参数并将节名放在键参数位置上。

函数列表：配置文件迭代器

```
#include <libgnome/gnome-config.h>  
void* gnome_config_init_iterator(const gchar* path)  
void* gnome_config_private_init_iterator(const gchar* path)  
void* gnome_config_init_iterator_sections(const gchar* path)  
void* gnome_config_private_init_iterator_sections(const gchar* path)  
void* gnome_config_iterator_next(void* iterator_handle,  
                                gchar** key,  
                                gchar** value)
```

2.8.5 其他的配置文件操作

下面的函数列表列举了一些可用于操作配置文件的其他操作函数。这些函数中最重要的已经介绍过了。`gnome_config_sync()`将配置文件写到磁盘上，`gnome_config_push_prefix()`允许缩短传递到其他`gnome-config`函数中的路径长度。还有一些布尔测试函数，用于询问`gnome-config`给定的节是否存在。

这里面有两个新函数。“删除”一个文件或节意味着忘掉任何存储在内存中的相关信息，包括从文件加载的缓存值和还没有用`gnome_config_sync()`函数保存到磁盘上的值。要“清除”一个文件、节或键意味着将它的值清除，所以，一旦调用`gnome_config_sync()`函数，文件、节或键都不再存在。

`gnome_config_sync()`函数自动调用`gnome_config_drop_all()`函数并释放所有的`gnome-config`资源，因为信息已经安全地在磁盘上存在了。

还有可以从gnome-config路径取得一个配置文件的实际(文件系统级)路径的函数。这些在应用程序中没有多大用处。

函数列表：其他配置文件函数

```
#include <libgnome/gnome-config.h>
gboolean gnome_config_has_section(const gchar* path)
gboolean gnome_config_private_has_section(const gchar* path)
void gnome_config_drop_all()
void gnome_config_sync()
void gnome_config_sync_file(const gchar* path)
void gnome_config_private_sync_file(const gchar* path)
void gnome_config_drop_file(const gchar* path)
void gnome_config_private_drop_file(const gchar* path)
void gnome_config_clean_file(const gchar* path)
void gnome_config_private_clean_file(const gchar* path)
void gnome_config_clean_section(const gchar* path)
void gnome_config_private_clean_section(const gchar* path)
void gnome_config_clean_key(const gchar* path)
void gnome_config_private_clean_key(const gchar* path)
gchar* gnome_config_get_real_path(const gchar* path)
gchar* gnome_config_private_get_real_path(const gchar* path)
void gnome_config_push_prefix(const gchar* path)
void gnome_config_pop_prefix()
```

2.9 会话管理

术语“会话”是指用户桌面状态的快照：哪个应用程序是打开的，窗口放在桌面的什么地方，每个应用程序打开了什么窗口，这些窗口的尺寸是多大，打开了什么文档，当前鼠标光标的位置等等。用户能在退出前保存这些会话，并且在下次登录时尽可能地自动恢复上次的会话。要做到这一点，应用程序必须有能力和恢复其状态特征，这些状态特征不是由窗口管理器控制的。

当应用程序应该保存状态时，由一个称为会话管理器的特殊程序通知应用程序。Gnome桌面环境带有一个称为gnome-session的会话管理器，但是Gnome用的X会话管理规范已经过时好几年了。CDE(通用桌面环境)使用同样的规范，KDE(K桌面环境)也准备采用这个规范。一个通过Gnome接口实现了会话管理的应用程序应该在任何会话管理的桌面环境上正常工作。Gnome确实对基本的规范(特别是启动“优先级”)做了一些扩展，但是这些应该不会中断其他的会话管理器，并且KDE也可能会实现这些规范。

使用GnomeClient对象

Gnome将应用程序与原始的、随X系统自带的会话管理接口屏蔽开来。这是通过一个称为GnomeClient的GtkObject对象做到的。GnomeClient代表应用程序与会话管理器的连接。Gnome管理会话管理的大多数细节。对大多数应用程序来说，只需要对两个请求响应。

- 当保存一个会话时，会话管理器会要求每个客户保存足够的信息，以便在下次登录时恢复状态。应用程序应该保存尽可能多的状态：当前打开的文档、鼠标光标的位置、命令历史等等。应用程序不用保存当前窗口的几何参数，窗口管理器会做这些事。

- 有时，会话管理器会要求客户关闭并退出（典型情况是当用户退出时）。当接收到这样的信息时，应该完成一些必要的工作，然后退出应用程序。

当会话管理器要求应用程序完成某个动作时，GnomeClient对象引发一个适当的信号。两个重要的信号是save_yourself和die。当应用程序保存它的状态时会引发 save_yourself信号，当应用程序应该退出时，引发 die信号。save_yourself信号的回调函数相当复杂，有好几个参数。die信号的回调函数很简单。

下面是来自于GnomeHello中的代码。GnomeHello获得一个指向GnomeClient对象的指针，并设置了一个信号回调函数：

```
client = gnome_master_client ();
gtk_signal_connect (GTK_OBJECT (client), "save_yourself",
                    GTK_SIGNAL_FUNC (save_session), argv[0]);
gtk_signal_connect (GTK_OBJECT (client), "die",
                    GTK_SIGNAL_FUNC (session_die), NULL);
```

argv[0]将用于save_yourself信号的回调函数。

下面是GnomeHello中的die回调函数：

```
static void
session_die(GnomeClient* client, gpointer client_data)
{
    gtk_main_quit ();
}
```

它的作用是退出应用程序。

下面是save_yourself的回调函数：

```
static gint
save_session (GnomeClient *client, gint phase, GnomeSaveStyle save_style,
              gint is_shutdown, GnomeInteractStyle interact_style,
              gint is_fast, gpointer client_data)
{
    gchar** argv;
    guint argc;
    /* allocate 0-filled, so it will be NULL-terminated */
    argv = g_malloc0(sizeof(gchar*)*4);
    argc = 1;
    argv[0] = client_data;

    if (message)
    {
        argv[1] = "--message";
        argv[2] = message;
        argc = 3;
    }
    gnome_client_set_clone_command (client, argc, argv);
    gnome_client_set_restart_command (client, argc, argv);

    return TRUE;
}
```

save_yourself信号必须告诉会话管理器怎样重新启动并克隆应用程序（创建一个新的实例）。重新启动的应用程序应该记住尽可能多的状态，在 GnomeHello例子中，它记住了显示的消息。保存应用程序状态最简单的方法是生成一个命令行，就像 GnomeHello所做的。可以要求GnomeClient使用gnome-config API函数做这些工作，还可以将信息保存在一个按会话配置的文件中。带有重要状态信息的应用程序需要使用这种方法。

2.10 Gtk+的主循环

Gtk+主循环的首要目的就是在连接到 X服务器的文件描述符上监听事件，并将事件转发到构件上。本节解释怎样使用主循环，怎样给主循环添加新功能：当主循环在指定的时间间隔内空闲时、当一个文件描述符已经读或写就绪、以及当主循环退出时调用一个函数。

2.10.1 主循环基本知识

从根本上来说，主循环是由 glib实现的。Gtk+将glib主循环连接到 Gdk的X服务器，并提供一个方便的接口（glib循环是比Gtk+的循环更低层的）。

gtk_main()函数运行主循环。直到调用 gtk_main_quit()函数，gtk_main()才会退出。gtk_main()函数可以递归调用，每次调用一个 gtk_main_quit()就退出gtk_main()函数的一个实例。gtk_main_level()函数返回递归的层次，也就是：如果没有 gtk_main()运行，返回0；如果一个gtk_main()函数在运行，返回1，等等。

gtk_main()函数的所有实例功能都是一样的，它们都监视同一个与 X服务器的连接，都对同样的事件队列起作用。gtk_main()实例用于阻塞、遮断一个函数的控制流直到满足某些条件。所有的Gtk+程序都用这个技巧使应用程序正在运行时main()函数不能退出去。gnome_dialog_run()函数使用了一个递归的主循环，此循环直到用户点击对话框的按钮时它才会返回。

有时候想处理一些事件，又不想将控制交给 gtk_main()，可以调用gtk_main_iteration()函数对主循环进行迭代。例如，这样可以处理单独的一个事件，它依赖于想将什么任务挂起。可以检查是否有任何事件需要通过调用 gtk_events_pending()函数处理。同样地，这两个函数允许临时将控制交还给 Gtk+。例如，在一个很长的计算中，想显示一个进度条，必须允许Gtk+主循环周期性地返回，让Gtk+能重画进度条。可以使用下面的代码：

```
while (gtk_events_pending())
    gtk_main_iteration();
```

下面是有关主循环的函数：

```
#include <gtk/gtkmain.h>
void gtk_main()
void gtk_main_quit()
void gtk_main_iteration()
gint gtk_events_pending()
guint gtk_main_level()
```

2.10.2 退出函数

退出函数就是当调用 gtk_main_quit()函数时要调用的回调函数。换句话说，回调函数只在

gtk_main()返回之前运行。回调函数应该是一个像下面这样定义的 GtkFunction：

```
typedef gint (*GtkFunction) (gpointer data);
```

退出函数是用 gtk_quit_add()添加进去的。添加退出函数时，必须指定一个由 gtk_main_level()返回的主循环的级别。第二个和第三个参数指定一个回调函数和回调数据。

回调函数的返回值说明了回调函数是否应该再次调用。只要回调函数返回 TRUE，它会被重复调用。只要它返回 FALSE，将取消与主循环的连接，并且不会再次调用。所有的退出函数都返回FALSE时，gtk_main()会返回。

gtk_quit_add()函数返回一个ID号码，可以用于用 gtk_quit_remove()函数删除该退出函数。还可以通过将回调数据传递给 gtk_quit_remove_by_data()函数来删除退出函数。

函数列表：退出函数

```
#include <gtk/gtkmain.h>
guint gtk_quit_add(guint main_level,
                  GtkFunction function,
                  gpointer data)
void gtk_quit_remove(guint quit_handler_id)
void gtk_quit_remove_by_data(gpointer data)
```

2.10.3 Timeout函数

有时候可能想应该在 gtk_main主循环中怎样让 GTK做点什么。这时可以创建一个定时 (Timeout) 函数，隔一定时间(毫秒)就调用一次。Timeout类似于其他编程环境中的定时器控件。下面的函数用于添加一个Timeout函数。

```
#include <gtk/gtkmain.h>
gint gtk_timeout_add( guint32      interval,
                    GtkFunction function,
                    gpointer      data );
```

第一个参数调用定时函数的时间间隔，以毫秒计。第二个参数是要调用的函数，第三个是要传递给函数的参数。函数返回一个整数值“标志”。可以用下面的函数停止调用定时函数：

```
#include <gtk/gtkmain.h>
void gtk_timeout_remove( gint tag );
```

其中tag参数是前一个函数返回的“标志”值。

还可以让回调函数返回FALSE或0来停止调用定时函数。也就是说，要想让函数继续调用，必须让它返回一个非0值或TRUE。

定期调用的回调函数声明应该是下面的形式：

```
gint timeout_callback( gpointer data );
```

可以看到，Timeout函数类似于许多可视化编程工具中的Timer控件（计时器）。

2.10.4 idle函数

当Gtk+主循环没有其他事情做时，idle函数连续运行。只有在事件队列是空的，并且主循环正常空闲着，正等待着有什么事情发生时，idle函数才会运行。只要它们返回 TRUE，这个函数就会一次又一次地调用；当它们返回 FALSE时，函数会被删除，就像调用了

gtk_idle_remove()函数一样。

列在下面函数列表中的 idle 函数 API，与 timeout 以及退出函数 API 是一样的。不过，不能在 idle 函数中调用 gtk_idle_remove() 函数，因为它会破坏 Gtk+ 的函数列表。要返回 FALSE 来删除 idle 函数。

idle 函数在对“只此一次”的代码排队时通常是很有用的，这样的代码一般在所有的事件已经处理之后才运行。相对昂贵（耗费系统资源较多）的操作，比如 Gtk+ 的大小协商以及 GnomeCanvas 重绘一般在返回 FALSE 的 idle 函数中发生。这保证了代价昂贵的操作只会执行一次，即使多个连续的事件独立地要求重新执行。

Gtk+ 的主循环包含一个简单的调度程序。idle 函数有一个分配给它们的优先级，就像 UNIX 进程所做的一样，也可以分配一个非缺省的优先级给 idle 函数。

函数列表：idle 函数

```
#include <gtk/gtkmain.h>
guint gtk_idle_add(GtkFunction function,
                  gpointer data)
void gtk_idle_remove(guint idle_handler_id)
void gtk_idle_remove_by_data(gpointer data)
```

2.10.5 输入函数

输入函数用于检查文件描述符的数据（由 open(2) 或 socket(2) 返回的文件描述符）。输入函数是在 Gdk 级处理的。当给定的文件描述符已经读写就绪时，会调用输入函数。它们对网络应用程序特别有用。关于文件描述符请参考 Unix 编程方面的参考书。

要添加一个输入函数，需要指定要监视的文件描述符、要等待的状态（读或写就绪），以及一个回调函数/数据对。下面的函数列表列出了这些 API。该函数可以用 gdk_input_add() 返回的标识符删除。不像 quit、timeout 和 idle 函数，从输入函数里面调用 gdk_input_remove() 函数删除该函数是安全的，Gtk+ 不会处于输入函数列表的迭代状态。

要指定等待的条件，使用 GdkInputCondition 标志：

GDK_INPUT_READ

GDK_INPUT_WRITE

GDK_INPUT_EXCEPTION

可以用 OR 将一个以上的标志连在一起。这些标志对应于传到 select() 系统调用的三种文件描述符集。要了解详细的内容，请参考 UNIX 的编程参考书。如果所有条件都得到满足，就会调用输入函数。

回调函数应该是这个样子：

```
typedef void (*GdkInputFunction) (gpointer data,
                                  gint source_fd,
                                  GdkInputCondition condition);
```

它接受回调数据、被监视的文件描述符以及要满足的条件（可能是一个正在监视的条件的子集）。

函数列表：输入函数

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

```
gint gdk_input_add(gint source_fd,  
                  GdkInputCondition condition,  
                  GdkInputFunction function,  
                  gpointer data)  
  
void gdk_input_remove(gint tag)
```

2.11 编译应用程序

用前面所介绍的基本概念，已经可以编译全功能的 Gtk+/Gnome应用程序了。但是还有一个大问题：如何配置编译选项？一些实用工具如 automake、autoconf、libtool等，可以用来简化这一过程。

为了方便维护，同时，也是为了便于使用这些实用工具，应该在编写代码时遵从一些约定。如果要程序发布为自由软件，最好能使程序源代码的目录结构遵从“GNU项目编码标准”。即使应用程序是私有的商用程序，不想公开源代码，从技术上来说，这么做也是一个非常好的选择，因为这些标准都是经过实践检验，能够让你节省大量的时间和精力。另外还应该在程序代码中包含INSTALL、README的文件。

2.11.1 生成源代码树

差不多所有的Gnome应用程序都使用同样的基于GNU工具automake、autoconf和libtool的编译系统。Gtk+和Gnome提供了一套autoconf宏，用于生成可移植的、符合标准的编译设置。我们用一个称为GnomeHello的应用程序来演示Gnome的特性。

Gnome应用程序遵从一系列的约定来生成源代码树和发布的tar文件，大多数约定被自由软件社区广泛使用。这些约定的许多方面已经在“GNU项目编码标准”(GNU Project's Coding Standards：http://www.gnu.org/prep/standards_toc.html)和Linux文件系统层次标准(Linux Filesystem Hierarchy Standard：<http://www.pathname.com/fhs/>)中正式化了。

GNU工具集，包括automake和autoconf使遵从这些标准变得很容易。然而，有时候你可能不想使用GNU工具集，例如，你也许需要一个统一的在Windows和MacOS平台上都能工作的编译工具(一些工具确实能在Windows平台上工作，它们使用Cygnus的“Cygwin”环境，参看<http://sourceware.cygnus.com/cygwin>)。

如果使用了autoconf和automake，除了编译应用程序，用户并不需要有这些工具。使用这些工具的目的是创建能在用户环境使用的、可移植的shell脚本和Makefile文件。

Autoconf实际上是一个工具集，其中包含aclocal、autoheader和autoconf等可执行文件。这些工具生成一个可移植的shell脚本——configure，configure和软件包一起发布给用户。它探查编译系统，生成Makefile文件和一个特殊的头文件config.h。由configure生成的文件能适应用户系统的特定环境。configure脚本从一个称为Makefile.in的模板文件生成每个Makefile文件。automake由一个手写的Makefile.am生成Makefile.in文件。Makefile.in文件随软件一同发布，当用户运行configure时会自动生成Makefile。

Libtool软件包是第三个重要的GNU工具，它的作用是确定共享库在特定平台上的特性。因为共享库在不同平台上可能会有所不同。

下面有一些Gnome软件包应该具有的特征：

- 一个README文件，介绍软件包。

- 一个INSTALL文件，解释怎样编译、安装软件包。
- 一个configure脚本，能使程序自动适应特定平台的特征（或者该平台所缺乏的特性）。configure可以带一个参数--prefix，指定要安装的软件包的位置。
- 标准的make目标，比如clean等等。
- 一个COPYING文件，包含软件包的版权信息。
- 一个ChangeLog文件，记录了软件的变化。
- 打包文件，一般用gzip压缩，在名字中包含软件包的版本（例如foo-0.2.1.tar.gz）。它们应该解开到单个目录中，目录应该以软件包及其版本命名，比如foo-0.2.1。
- 国际化是由GNU gettext软件包提供的。将gettext软件包随应用程序提供给用户，这样用户没有gettext也能够实现国际化。

下面是创建Gtk+/Gnome应用程序源代码树框架的重要步骤：

1) 创建一个顶级目录，用以容纳应用程序的所有组件，包括编译文件、文档以及翻译文件。

2) 通常在顶级目录下创建一个src子目录，将所有的源代码放在该目录下，并与其他文件分开。

3) 在顶级目录下，创建AUTHORS、NEWS、COPYING和README文件。还可以创建一个空的ChangeLog文件。

4) 写一个configure.in文件；configure.in文件的主要作用是决定使用什么样的编译器、编译标志以及链接标志。configure.in还可以使用#define符号反映当前平台的特征；它把这些优先放在自动生成的config.h文件里。

5) 写一个acconfig.h文件。它是config.h.in文件要使用的模板文件。这个文件应该撤销每个可能在config.h中定义了的符号以避免重复定义（一般在config.h中用#define定义，用#undef撤销定义）。autoheader程序基于acconfig.h创建config.h.in文件，autoconf程序创建config.h文件。autoheader是autoconf软件包中的实用程序。

6) 创建一个空的stamp.h.in文件。在configure.in中的AM_CONFIG_HEADER宏会用到它。

7) 在顶级目录下，写一个Makefile.am文件，在其中列出每个包含源代码的子目录；在每个子目录中也写一个Makefile.am文件。

8) 运行gettext软件包中的gettextize程序。这样可以创建intl和po目录，这是软件国际化所需要的。在intl目录中包含GNU gettext源代码。如果编译程序的用户没有gettext，它们可以在执行configure脚本时传一个--with-included-gettext参数，让configure在intl目录下自动编译一个gettext的静态版本。在po容纳了翻译文件后，gettextize也会创建一个称为po/Makefile.in.in的文件，用于编译翻译文件。

9) 创建一个po/POTFILES.in文件，在其中列出应该扫描字符串以便翻译的源文件。最初的POTFILES.in文件可以是空的。

10) 从其他的Gnome模块中复制一个autogen.sh文件和它的宏目录。必须根据自己的软件包的名称修改autogen.sh文件。运行autogen.sh文件将调用libtoolize、aclocal、autoheader、automake以及autoconf。

11) autogen.sh用--add-missing参数调用文件automake。这会添加一些文件，比如带有通用安装指导的INSTALL文件。编辑INSTALL，在其中包含任何针对应用程序的安装指南。

autoconf.sh会在每个目录下创建一个 Makefile。

2.11.2 configure.in文件

autoconf处理configure.in文件，生成一个configure脚本。configure是一个可移植的shell脚本，它检查编译环境以决定哪些库可用，所用平台有什么特征，哪些库和头文件已经找到等等。基于这些信息，它修改编译标记，生成 Makefile文件，并/或输出一个包含已定义的预处理符号的config.h文件。configure并不需要运行autoconf，所以在发布应用程序之前生成这个文件，这样，用户就不必有 autoconf软件包。

眼前的任务就是写一个configure.in文件。文件基本上是一系列的 m4宏，根据传递给它们的参数，这些宏扩展为 shell脚本代码段。还可以手工书写 shell代码。要真正理解怎样写configure.in文件要求有一些 m4的知识以及一些 Bourne shell的知识。幸运的是，有省事的方法；可以找一个已有的 configure.in文件，然后修改它以适应你的应用程序。还有一个很全面的autoconf 手册，里面介绍了很多随 autoconf发布的预先写好的宏。

Gtk+和Gnome的开发者已经进一步简化了这些工作，提供了一些宏用于在用户的系统中定位Gtk+和Gnome。

下面是一个简单的configure.in文件，来自于Gnome版的“Hello, World”：

```
AC_INIT(src/hello.c)
AM_CONFIG_HEADER(config.h)
AM_INIT_AUTOMAKE(GnomeHello, 0.1)
AM_MAINTAINER_MODE
AM_ACLOCAL_INCLUDE(macros)
GNOME_INIT
AC_PROG_CC
AC_ISC_POSIX
AC_HEADER_STDC
AC_ARG_PROGRAM
AM_PROG_LIBTOOL
GNOME_COMPILE_WARNINGS
ALL_LINGUAS="de es fr no ru sv fi"
AM_GNU_GETTEXT
AC_SUBST(CFLAGS)
AC_SUBST(CPPFLAGS)
AC_SUBST(LDFLAGS)
AC_OUTPUT([
Makefile
macros/Makefile
src/Makefile
intl/Makefile
po/Makefile.in
pixmaps/Makefile
doc/Makefile
doc/C/Makefile
doc/es/Makefile
])
```

上面以 AC开头的宏来自 autoconf，以 AM开头的宏来自 automake。要从 autoconf或

automake中寻求帮助，这一点很有用。以GNOME开头的宏来自于Gnomemacros目录。这些宏都是用m4宏语言写的。如果将autoconf和automake安装在/usr目录下，autoconf和automake中的标准宏一般放在/usr/share/aclocal目录下。

- AC_INIT总是configure.in中的第一个宏。它扩展为许多可由其他 configure脚本共享的模板文件代码。这些代码解析传到 configure中的命令行参数。这个宏的一个参数是一个文件名，这个文件应该在源代码目录中，它用于健全性检查，以保证 configure脚本已正确定位源文件目录。
- AM_CONFIG_HEADER指定了要创建的头文件，差不多总是 config.h。创建的头文件包含由configure定义的C预处理符号。最低限度应该定义 PACKAGE和VERSION符号，这样可以将应用程序名称和版本传送到代码中，而无须对它们硬编码（非公用的源文件应该包含 config.h(#include <config.h>)以利用这些定义。然而，不要将 config.h文件安装到系统中，因为它有可能与其他的软件包冲突)。
- AM_INIT_AUTOMAKE初始化 automake。传到这个宏里的参数是要编译的应用程序的名称和版本号(这些参数成为 config.h中定义的 PACKAGE和VERSION值)。
- AM_MAINTAINER_MODE关闭缺省时仅供程序维护者使用的 makefile目标，并修改以使configure能理解--enable-maintainer-mode选项。--enable-maintainer-mode将maintainer-only目标重新打开。仅供维护者使用的 makefile目标允许最终用户清除自动生成的文件，比如configure，这意味着要修复编译故障，必须安装有 autoconf 和automake软件。注意，因为 autogen.sh脚本主要是给开发人员用的， autogen.sh会自动传递一个 --enable-maintainer-mode选项给configure。
- AM_ACLOCAL_INCLUDE指定一个附加的目录，用于搜索 m4宏。在这里，它指定为 macros子目录。在这个目录中应该有 Gnome宏的拷贝。
- GNOME_INIT给configure添加一个与 Gnome相关的命令行参数个数，并为 Gnome程序定义一些makefile变量，这些变量中包含了必要的预处理程序和链接程序标志。这些标志是由 gnome-config脚本取得的。安装 gnome-libs时会安装 gnome-config脚本。
- AC_PROG_CC定位C编译器。
- AC_ISC_POSIX 添加一些在某些平台上实现 POSIX兼容需要的标志。
- AC_HEADER_STDC检查当前平台上是否有标准的 ANSI头文件，如果有，则定义 STDC_HEADERS。
- AC_ARG_PROGRAM添加一些选项到configure中，让用户能够修改安装程序的名称（如果在用户系统上碰巧有一个与要安装的程序名称相同的程序，这是很有用的）。
- AM_PROG_LIBTOOL是由automake用来设置 libtool的用途的。只在计划编译共享库或动态可加载模块时才需要设置这个值。
- GNOME_COMPILE_WARNINGS给gcc命令行添加许多警告选项，但是在其他绝大多数的编译器上什么也不做。
- ALL_LINGUAS= “ es ” 不是一个宏，只是一句 shell代码。它包含一个由空格分隔的语言种类缩写表，对应于 po子目录下的 .po文件。 .po文件包含翻译成其他语言的文本，所以ALL_LINGUAS应该列出程序已经被翻译成的所有语言。
- AM_GNU_GETTEXT由automake使用，但是这个宏会随 gettext软件包发布。它让

automake执行一些与国际化相关的任务。

- AC_SUBST输出一个变量到由configure生成的文件中。具体内容将在后面说明。
- AC_OUTPUT列出由configure脚本创建的文件。这些文件都是由带.in后缀的同名文件生成的。例如，src/Makefile是由src/Makefile.in生成的，config.h是由config.h.in生成的。

在执行AC_OUTPUT宏时，configure脚本处理包含有两个@符号标志的变量（例如@PACKAGE@）的文件。只有用AC_SUBST输出了变量，它才能识别这些变量（许多在上面讨论过的预先写好的宏都用AC_SUBST定义变量）。这些特征用于将一个Makefile.in文件转换成一个Makefile文件。典型情况下，Makefile.in是由automake从Makefile.am生成的（不过，你可以只用autoconf，而不用automake，自己编写一个Makefile.in）。

2.11.3 Makefile.am文件

automake处理Makefile.am，生成一个符合标准的Makefile.in文件。automake会做很多工作：例如，它维护源文件之间的依赖关系；生成所有的标准目标，比如install和clean；它还生成更复杂的目标：如果Makefile.am是正确的，简单输入make dist就会创建一个标准的.tar.gz文件。

一般情况是在最上层目录下写一个Makefile.am，然后在每一个子目录下分别写一个Makefile.am文件。automake会从最上层开始递归处理各个Makefile.am，然后生成一个Makefile.in。

在最上层目录的Makefile.am通常都很简单，下面是一个例子：

```
SUBDIRS = macros po intl src pixmaps doc
EXTRA_DIST = \
    gnome-hello.desktop
Applicationsdir = $(datadir)/gnome/apps/Applications
Applications_DATA = gnome-hello.desktop
```

上面程序的第一行通知automake在给定的子目录中递归查找Makefile.am文件。在src子目录的Makefile.am是这样的：

```
INCLUDES = -I$(top_srcdir) -I$(includedir) $(GNOME_INCLUDEDIR) \
    -DG_LOG_DOMAIN=\"GnomeHello\"
-DGNOMELOCALEDIR=\"\"$(datadir)/locale\" \
-I../intl -I$(top_srcdir)/intl
bin_PROGRAMS = gnome-hello
gnome_hello_SOURCES = \
    app.c \
    hello.c \
    menus.c \
app.h \
hello.h \
menus.h
gnome_hello_LDADD = $(GNOMEUI_LIBS) $(GNOME_LIBDIR) $(INTLLIBS)
```

automake能够理解许多“不可思议的变量”，并用这些变量创建Makefile.in文件。在上面的小例子中，用到了下面的变量：

- INCLUDES指定了在编译阶段（与连接阶段相对）中传递给C编译器的标志。这一行用到的变量来自于2.11.2节中的configure.in文件。

- `bin_PROGRAMS`列出了要编译的程序。
- `hello_SOURCES`列出了要编译和连接的文件，这些文件是依赖生成 `hello`程序的。程序名必须列在`bin_PROGRAMS`中。在这个变量中的所有文件都被自动包含在发布包中。
- `hello_LDADD`列出了要传递给连接程序的标志。在这个例子中是由 `configure`决定的 Gnome库标志。
- `INCLUDES`行中有几个在所有的 Gnome程序中都应该用到的元素。应该定义 `G_LOG_DOMAIN`，来自与校验和断言代码中的错误信息会报告这个值，这样就能够判定错误是发生在什么地方(在代码中，还是在一个库中)。GNOMELOCALEDIR用于定位翻译文件。`intl`目录被添加到了头文件的搜索路径，这样应用程序就能够找到 `intl`头文件。

在`Makefile.am`中还可以做很多复杂的事，特别是，可以添加两端带有 `@`符号的、能带入到`configure`脚本中的变量。可以有条件地包含基于 `configure`校验的`Makefile`文件中的一部分，还可以建立库。`automake`的手册介绍了细节内容。

表2-1概括了由`automake`生成的最常见的`make`目标。当然，缺省的`make`目标是`all`，它编译整个程序。GNU代码标准(http://www.gnu.org/prep/standards_toc.html)中有这些`make`目标和GNU `Makefile`文件的详细信息。

表2-1 make目标

标准make目标	目标介绍
<code>Dist</code>	建立一个用于发布的tar压缩文件(.tar.gz)
<code>Distcheck</code>	建立一个用于发布的压缩文件，然后尝试编译
<code>Clean</code>	删除编译结果(目标文件和可执行文件)，但是也许不会删除一些由机器生成的文件
<code>Install</code>	如果需要，创建安装目录，将软件复制到目录里
<code>Uninstall</code>	反安装(删除已经安装文件)
<code>Distclean</code>	将执行 <code>configure</code> 脚本以及所有 <code>make</code> 过程的效果按相反的过程重做一遍，也就是，将一个tar文件还原为它原来的状态
<code>Mostlyclean</code>	和 <code>clean</code> 差不多，但是将那些极有可能不需重建的，目标文件留下来
<code>Maintainer-clean</code>	比 <code>clean</code> 更彻底，也许会删除一些需要由特殊工具软件重建的文件，比如由机器生成的源代码
<code>TAGS</code>	重建一个tag表，Emacs可以使用它
<code>Check</code>	如果有，则运行一个测试组件

2.11.4 安装支持文件

完整的Gnome应用程序还有许多代码以外的东西。它们有在线帮助(要列在 Gnome的主菜单上)，有界面翻译，还有一个桌面图标。它们也许带一个 `pixmap`以及一个用在“关于”对话框上的徽标、一个用于“向导”的图形或者一个用以帮助用户快速区别菜单项或列表元素的小图标。下面的内容介绍怎样发布这些文件。

1. 安装数据文件：文档和 `pixmap`

文档和`pixmap`的安装方法是差不多的。`automake`允许你将数据文件安装到任意位置，可以用配置文件中定义的变量决定将它们安装到哪里。

(1) `pixmap`

要从Makefile.am中安装数据文件，只需简单地为安装目标指定一个名字 (pixmap 就不错) 然后为该目录和要安装到目录里的文件分别创建一个变量。例如：

```
EXTRA_DIST = gnome-hello-logo.png
pixmapdir = $(datadir)/pixmaps
pixmap_DATA = gnome-hello-logo.png
fill
```

“ pixmap ” 字符串将 pixmap_DATA 变量和 pixmapdir 变量连接起来。 automake 解释 _DATA 前缀，并在 Makefile.in 中生成适当的安装规则。这个 Makefile.am 片断将 gnome-hello-logo.png 文件安装到 \$(datadir)/pixmaps 目录下，\$(datadir) 是由 configure 分配的变量。典型情况下，\$(datadir) 是 /usr/local/share (更精确的说，是 \$(prefix)/share)，这是独立于体系结构的数据文件 (也就是，几个具有不同二进制文件格式的系统共享的文件) 的标准位置。

Gnome 的 pixmap 图片的标准位置是 \$(datadir)/pixmaps，所以在例子中我们这样用。Gnome 项目鼓励在所有的 pixmap 图片中使用 PNG 格式，这个格式是 gdk_imlib (Gnome 图象加载库) 支持的。它的文件尺寸小，速度快，也不存在专利问题。

(2) 文档

安装文档使用同样的原则，不过稍有一点复杂。 Gnome 文档通常是用 DocBook 写的。DocBook 是一个 SGML DTD (Document Type Definition，文档类型定义)，就像 HTML 一样。然而，DocBook 的文档标签是为技术文档设计的。用 DocBook 写的文档可以转换为其他格式，包括 PostScript 和 HTML。依照标准，应该安装 HTML 格式的文档，用户就可以用 Web 浏览器或 Gnome 帮助浏览器阅读文档。

Gnome 库和帮助浏览器能理解一个名为 topic.dat 的文件，这个文件只是一个含有相应 URL 的帮助主题列表。它起应用程序帮助主题索引的作用。下面是一个例子，只有两条：

```
gnome-hello.html      GnomeHello manual
advanced.html          Advanced Topics
```

URL 路径相对于所安装的帮助文件的目录。

应该预先考虑文档可能会翻译为其他语言。最好为每一个地区建立一个子目录，例如，缺省地区 (C) 或 es (西班牙语)。使用这种方法翻译程序不会引起混乱。一般将 Gnome 帮助安装在以地区开头的目录下，这种做法用其他观点来看也是很方便的。文档目录看起来也许和 GnomeHello 示例程序差不多：

```
doc/
  Makefile.am
C/
  Makefile.am
  gnome-hello.sgml
  topic.dat
es/
  Makefile.am
  gnome-hello.sgml
  topic.dat
```

下面是 doc/C/Makefile.am：

```
gnome_hello_helpdir = $(datadir)/gnome/help/gnome-hello/C
```

```

gnome_hello_help_DATA = \
    gnome-hello.html \
    topic.dat
SGML_FILES = \
    gnome-hello.sgml
# files that aren't in a binary/data/library target have to be listed here
# to be included in the tarball when 'make dist'
EXTRA_DIST = \
    topic.dat \
    $(SGML_FILES)
## The - before the command means to ignore it if it fails. That way
## people can still build the software without the docbook tools
all:
gnome-hello.html: gnome-hello/gnome-hello.html
-cp gnome-hello/gnome-hello.html .
gnome-hello/gnome-hello.html: $(SGML_FILES)
-db2html gnome-hello.sgml
## when we make dist, we include the generated HTML so people don
## have to have the docbook tools
dist-hook:
mkdir $(distdir)/gnome-hello
-cp gnome-hello/*.html gnome-hello/*.css $(distdir)/gnome-hello
-cp gnome-hello.html $(distdir)
install-data-local: gnome-hello.html
$(mkinstalldirs) $(gnome_hello_helpdir)/images
-for file in $(srcdir)/gnome-hello/*.html $(srcdir)/gnome-hello/*.css; do \
basefile=`basename $$file`; \
$(INSTALL_DATA) $(srcdir)/$$file $(gnome_hello_helpdir)/$$basefile; \
done
gnome-hello.ps: gnome-hello.sgml
-db2ps $<
gnome-hello.rtf: gnome-hello.sgml
-db2rtf $<

```

需要特别注意的是生成的 HTML 文件的安装目录：\$(datadir)/gnome/help/gnome-hello/C。Gnome 库在这里查找帮助文件。每个应用程序的帮助都放在 \$(datadir)/gnome/help 下的它自己的目录下。每个地区的文档都安装在应用程序目录的对应于地区的子目录下。

2. .desktop 入口

Gnome 程序带有一个 .desktop 入口，它是一个以 desktop 为后缀的文本文件，描述应用程序应该放在 Gnome 主菜单的什么位置。安装一个 .desktop 入口文件可以让应用程序显示在 Gnome 面板菜单（主菜单）中。下面是 gnome-hello.desktop 文件：

```

[Desktop Entry]
Name=Gnome Hello
Name[es]=Gnome Hola
Name[fi]=GNOME-hei
Name[no]=Gnome hallo
Name[sv]=Gnome Hej
Comment=Hello World
Comment[es]=Hola Mundo

```

```
Comment[fi]=Hei, maaailma
Comment[sv]=Hej Världen
Comment[no]=Hallo verden
Exec=gnome-hello
Icon=gnome-hello-logo.png
Terminal=0
Type=Application
```

这个文件由键 - 值对组成。Name键指定在缺省地区场合的名称；任何键都可以有一个用于标明地区的字符串，放在一对方括号里面。当登录到 X窗口时，如果指定了不同的地区，系统会根据语言从这里读取相应的字符串。Comment键是一个“工具提示”或“暗示”，用以更详细地描述应用程序。Exec是用来执行程序的命令行。Terminal是布尔类型，如果为非0值，程序在一个终端内运行。在这里Type应该是“Application”

安装一个.desktop条目很简单。下面是GnomeHello中的最顶层的Makefile.am文件：

```
SUBDIRS = macros po intl src pixmaps doc
EXTRA_DIST = \
    gnome-hello.desktop
Applicationsdir = $(datadir)/gnome/apps/Applications
Applications_DATA = gnome-hello.desktop
```

注意，在\$(datadir)/gnome/apps/下有一个目录树，可以将应用程序安装到下面子目录所对应的类别中。GnomeHello将自己安装在“Applications”类中，而其他的应用程序也许会选择Games、Graphics、Internet或者其他合适的地方。尽量选择一个已经存在的类别，而不是自己建一个。

Makefile.am中的EXTRA_DIST变量列出了需要包含在发布软件包（压缩的tar文件）中的文件。最重要的文件会自动包含进来，例如，所有作为二进制或库的源文件的文件都会自动包含。然而，automake并不认识.desktop文件，或SGML文档，所有这些文件必须列在EXTRA_DIST中。如果将这些文件留在EXTRA_DIST之外，用make distcheck命令编译发布程序通常会失败。