Beta1-GTK-3-Chinese-Reference-Manual

Author:
MingpeiZheng , truehyp

2015 年 2 月 17 日

1 GTK+ 概览

GTK+ 是用来创造图形界面的库，它可以运行在许多类 UNIX 系统，Windows 和 OS X。GTK+ 按照 GNU LGPL 许可证发布，这个许可证对程序来说相对宽松。GTK+ 有一个基于 C 的面向对象的灵活架构，它有对于许多其他语言的版本，包括 C++，Objective-C，Guile/Scheme, Perl, Python, TOM, Ada95, Free Pascal 和 Eiffel。GTK+ 依赖于以下库：

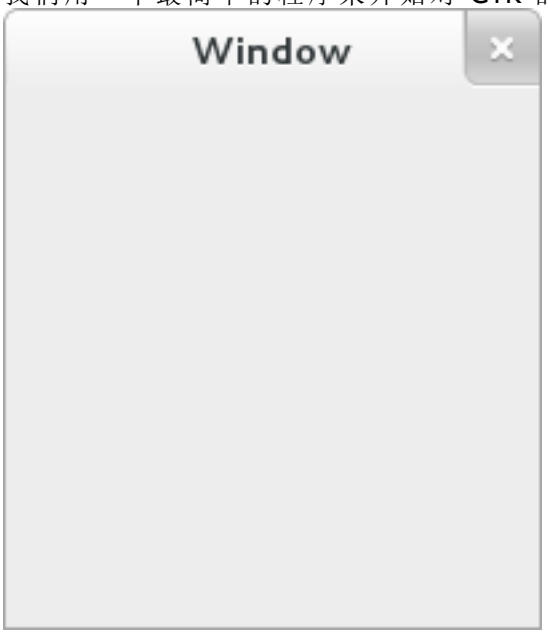
- GLib 是一个多方面用途的库, 不仅仅针对图形界面。GLib 提供了有用的数据类型、宏、类型转换，字符串工具，文件工具，主循环抽象等等。
- GObject 是一个提供了类型系统、包括一个元类型的基础类型集合、信号系统的库。
- GIO 是一个包括文件、设备、声音、输入输出流、网络编程和 DBus 通信的现代的易于使用的 VFS 应用程序编程接口。
- cairo Cairo 是一个支持复杂设备输出的 2D 图形库。
- Pango Pango 是一个国际化正文布局库。它围绕一个表现正文段落的 PangoLayout object。Pango 提供 GtkTextView、GtkLabel、GtkEntry 和其他表现正文的引擎。
- ATK ATK 是一个友好的工具箱。它提供了一个允许技术和图形用户界面交互的界面的集合。例如，一个屏幕阅读程序用 ATK 去发现界面上的文字并为盲人用户阅读。GTK ＋部件已经被制作方便支持 ATK 框架。
- GdkPixbuf 是一个允许你从图像数据或图像文件创建 GdkPixbuf(``pixel buffer'') 的小的库。用一个 GdkPixbuf 与显示图像的 GtkImage 结合。
- GDK GDK 是一个允许 GTK ＋支持复杂图形系统的抽象层。GDK 支持 X11、Windows 和 OS X 的图形系统工具。
- GTK+ 是 GTK+ 库本身包含的部件，确切的说是 GUI 零件，比如 GtkButton 或者 GtkTextView。

2 开始使用 GTK+

这一章包含一些让你开始学习 GTK+ 编程的指导信息，假设你已经安装了 GTK+，依赖库和 C 编译器。如果你想构建 GTK+ 本身，可以参考编译 GTK+ 的部分。

2.1 基础

我们用一个最简单的程序来开始对 GTK 的介绍，下面的程序将创造一个 200×200 像素的窗体。



新建一个名为 example-0.c 的文件，写入如下内容：

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

int main (int   argc, char *argv[])
{
    GtkWidget *window;
    gtk_init (&argc, &argv);
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Window");
    g_signal_connect (window, "destroy", G_CALLBACK (gtk_main_quit), NULL);
    gtk_widget_show (window);
    gtk_main ();
    return 0;
}
```

然后在终端输入以下命令用 GCC 编译程序：

```
gcc `pkg-config --cflags gtk+-3.0` -o example-0 example-0.c `pkg-config --libs gtk+-3.0`
```

注：要查找更多编译 GTK 程序的信息，请查看手册中编译 GTK+ 应用的部分。

所有的 GTK+ 程序必须包括 gtk/gtk.h，这个头文件声明了 GTK+ 程序需要的函数、类和宏。

注：即使 GTK+ 安装了多种头文件，只有顶层的 gtk/gtk.h 能被第三方代码直接引入。如果引入任意一个其他的头文件，编译器都会报错。

我们接下来进入 main 函数，将会声明一个 GtkWidget 类型的指针变量 window。下面一行将会调用 gtk_init() 函数，这个函数是 GTK+ 程序的初始化函数，它将设置 GTK+、类系统和与窗口环境的连接。

注: 要查找更多 GTK+ 程序的命令参数，请查看手册中运行 GTK+ 程序的部分。

调用 gtk_window_new() 函数将会创造一个新的 GtkWindow 并将其储存在 window 变量中。并且，这个窗体的类型是 GTK_WINDOW_TOPLEVEL，这也就意味着这个 GtkWindow 将会被当前的系统管理：这个窗体将会根据不同的系统平台产生一个框架、一个标题栏和窗口控件。

当 GtkWindow 被破坏时，我们将“destroy”信号连接到 gtk_main_quit() 函数以终止这个程序。这个函数将会在之后终止由 gtk_main() 函数启动的 GTK+ 程序的主循环。“destroy”信号会在一个窗口部件被破坏时触发，也会是在调用 gtk_widget_destroy() 或者在这个窗口部件失去母体控件时触发。最顶端的 GtkWindows 会在关闭按钮被点击时被破坏。

GtkWidgets 默认是隐藏的，通过在一个控件上调用 gtk_widget_show()，我们将能设置其为可见。所有这些工作都将在主循环开始后被完成。

最后一行调用了 gtk_main()。这个函数就会启动 GTK+ 程序的主循环并且在 gtk_main_quit() 函数被调用之前都阻止 main() 的控制流。

当程序运行时，GTK+ 一直接收事件。有一些输入事件是由用户与程序互动时产生的，但也有一些事件，比如来自窗口管理器或者其他程序的信息。GTK+ 处理这些事件和信息，然后触发信号。为这些信号连接 handles 就是让你的程序为用户输入做出正确响应的方法。

下面这个例子有点复杂，它将展示 GTK+ 的能力。按照程序设计语言和库的古老传统，这个程序也叫 Hello, World。

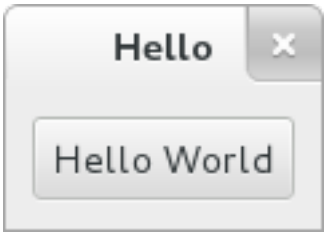


Figure 1: hello_worldpng

Example 1. Hello World in GTK+

新建一个名为 example-1.c 的文件，写入如下内容：

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

/* This is a callback function. The data arguments are ignored
 * in this example. More on callbacks below.
 */
static void print_hello (GtkWidget *widget,
        gpointer   data)
{
    g_print ("Hello World\n");
}
```

```c
static gboolean on_delete_event (GtkWidget *widget,
        GdkEvent  *event,
        gpointer   data)
{
    /* If you return FALSE in the "delete_event" signal handler,
     * GTK will emit the "destroy" signal. Returning TRUE means
     * you don't want the window to be destroyed.
     *
     * This is useful for popping up 'are you sure you want to quit?'
     * type dialogs.
     */
    g_print ("delete event occurred\n");
    return TRUE;
}

int main (int  argc, char *argv[])
{
    /* GtkWidget is the storage type for widgets */
    GtkWidget *window;
    GtkWidget *button;

    /* This is called in all GTK applications. Arguments are parsed
     * from the command line and are returned to the application.
     */
    gtk_init (&argc, &argv);

    /* create a new window, and set its title */
    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Hello");

    /* When the window emits the "delete-event" signal (which is emitted
     * by GTK+ in response to an event coming from the window manager,
     * usually as a result of clicking the "close" window control), we
     * ask it to call the on_delete_event() function as defined above.
     *
     * The data passed to the callback function is NULL and is ignored
     * in the callback function.
     */
    g_signal_connect (window, "delete-event", G_CALLBACK (on_delete_event), NULL);

    /* Here we connect the "destroy" event to the gtk_main_quit() function.
     * This signal is emitted when we call gtk_widget_destroy() on the window,
     * or if we return FALSE in the "delete_event" callback.
     */
```

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

    /* Sets the border width of the window. */
    gtk_container_set_border_width (GTK_CONTAINER (window), 10);

    /* Creates a new button with the label "Hello World". */
    button = gtk_button_new_with_label ("Hello World");

    /* When the button receives the "clicked" signal, it will call the
     * function print_hello() passing it NULL as its argument.
     * The print_hello() function is defined above.
     */
    g_signal_connect (button, "clicked", G_CALLBACK (print_hello), NULL);

    /* The g_signal_connect_swapped() function will connect the "clicked" signal
     * of the button to the gtk_widget_destroy() function; instead of calling it
     * using the button as its argument, it will swap it with the user data
     * argument. This will cause the window to be destroyed by calling
     * gtk_widget_destroy() on the window.
     */
    g_signal_connect_swapped (button, "clicked", G_CALLBACK (gtk_widget_destroy), window);

    /* This packs the button into the window. A GtkWindow inherits from GtkBin,
     * which is a special container that can only have one child
     */
    gtk_container_add (GTK_CONTAINER (window), button);

    /* The final step is to display this newly created widget... */
    gtk_widget_show (button);

    /* ... and the window */
    gtk_widget_show (window);

    /* All GTK applications must have a gtk_main(). Control ends here
     * and waits for an event to occur (like a key press or a mouse event),
     * until gtk_main_quit() is called.
     */
    gtk_main ();
    return 0;
}
```

然后在终端输入以下命令用 GCC 编译程序：

```
gcc `pkg-config --cflags gtk+-3.0` -o example-1 example-1.c `pkg-config --libs gtk+-3.0`
```

2.2 填充

当创建一个应用时，你将会想将多个控件放入一个窗口控件。我们的第一个 helloworld 范例仅仅使用了一个控件，因而我们可以只是简单地调用一个 `gtk_container_add()` 将控件填充到一个窗口控件。但是当你想要向窗口控件中放置超过一个控件时，控制每一个控件的位置和大小就变得很重要了。这就是接下来要讲的填充。

GTK+ 自带了大量各种布局的容器，这些容器的目的是控制被添加到他们的子控件的布局。具体可以参考布局容器的概述。下面的示例显示了 GtkGrid 容器如何让你如何安排几个按钮：

Figure 2: grid_packingpng

Example 2. Packing buttons

新建一个名为 `example-2.c` 的文件，写入如下内容:

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

static void
print_hello (GtkWidget *widget,
             gpointer   data)
{
    g_print ("Hello World\n");

}

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

    /* This is called in all GTK applications. Arguments are parsed
    * from the command line and are returned to the application.
    */
    gtk_init (&argc, &argv);

    /* create a new window, and set its title */
```

```
window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
gtk_window_set_title (GTK_WINDOW (window), "Grid");
g_signal_connect (window, "destroy", G_CALLBACK (gtk_main_quit), NULL);
gtk_container_set_border_width (GTK_CONTAINER (window), 10);

/* Here we construct the container that is going pack our buttons */
grid = gtk_grid_new ();

/* Pack the container in the window */
gtk_container_add (GTK_CONTAINER (window), grid);

button = gtk_button_new_with_label ("Button 1");
g_signal_connect (button, "clicked", G_CALLBACK (print_hello), NULL);

/* Place the first button in the grid cell (0, 0), and make it fill
* just 1 cell horizontally and vertically (ie no spanning)
*/
gtk_grid_attach (GTK_GRID (grid), button, 0, 0, 1, 1);

button = gtk_button_new_with_label ("Button 2");
g_signal_connect (button, "clicked", G_CALLBACK (print_hello), NULL);

/* Place the second button in the grid cell (1, 0), and make it fill
* just 1 cell horizontally and vertically (ie no spanning)
*/
gtk_grid_attach (GTK_GRID (grid), button, 1, 0, 1, 1);

button = gtk_button_new_with_label ("Quit");
g_signal_connect (button, "clicked", G_CALLBACK (gtk_main_quit), NULL);

/* Place the Quit button in the grid cell (0, 1), and make it
* span 2 columns.
*/
gtk_grid_attach (GTK_GRID (grid), button, 0, 1, 2, 1);

/* Now that we are done packing our widgets, we show them all
* in one go, by calling gtk_widget_show_all() on the window.
* This call recursively calls gtk_widget_show() on all widgets
* that are contained in the window, directly or indirectly.
*/
gtk_widget_show_all (window);

/* All GTK applications must have a gtk_main(). Control ends here
* and waits for an event to occur (like a key press or a mouse event),
* until gtk_main_quit() is called.
```

```
    */
    gtk_main ();

    return 0;
}
```

然后在终端输入以下命令用 GCC 编译程序：

```
gcc `pkg-config --cflags gtk+-3.0` -o example-2 example-2.c `pkg-config --libs gtk+-3.0`
```

2.3 绘制

许多插件，比如 buttons，自己就做了它们所有的绘制工作。比如你仅仅需要告诉它们你想看到的标签、你想它们使用的字体、绘制按钮的轮廓和焦点矩形。有时候，有必要做些自定义的绘制。在这种情况下，一个 GtkDrawingArea 控件可能是正确的选择，这个控件提供了一个画布，在这个画布上你可以绘制并且将其连接到”draw“信号。

控件的内容常常需要被部分或者全部重新绘制。比如，当另一个窗口控件被移动并且露出控件的一部分，或者当包含它的窗口重新调整大小时，也会导致控件的部分或者全部被重新绘制。通过调用 gtk_widget_queue_draw() 或者它的变体，GTK+ 提供一个现成的 cairo 给绘制信号从而实现众多细节。

下面的程序将会展示一个绘制信号句柄。这个例子比之前的略微复杂，因为它也通过 button_press 和 motion_notify 句柄显示出输入活动。

Figure 3: drawingpng

Example 3. Drawing in response to input

新建一个名为 example-3.c 的文件，写入如下内容：

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

/* Surface to store current scribbles */
static cairo_surface_t *surface = NULL;
```

```
static void clear_surface (void)
{
    cairo_t *cr;

    cr = cairo_create (surface);

    cairo_set_source_rgb (cr, 1, 1, 1);
    cairo_paint (cr);

    cairo_destroy (cr);

}

/* Create a new surface of the appropriate size to store our scribbles */
static gboolean
configure_event_cb (GtkWidget         *widget,
                    GdkEventConfigure *event,
                    gpointer           data)
{
    if (surface)
    cairo_surface_destroy (surface);

    surface = gdk_window_create_similar_surface (gtk_widget_get_window (widget),
                                                 CAIRO_CONTENT_COLOR,
                                                 gtk_widget_get_allocated_width (widget),
                                                 gtk_widget_get_allocated_height (widget));

    /* Initialize the surface to white */
    clear_surface ();

    /* We've handled the configure event, no need for further processing. */
    return TRUE;

}

/* Redraw the screen from the surface. Note that the ::draw
*  * signal receives a ready-to-be-used cairo_t that is already
*   * clipped to only draw the exposed areas of the widget
*    */
static gboolean draw_cb (GtkWidget *widget,
         cairo_t   *cr,
         gpointer   data)
{
    cairo_set_source_surface (cr, surface, 0, 0);
    cairo_paint (cr);
```

```
    return FALSE;

}

/* Draw a rectangle on the surface at the given position */
static void draw_brush (GtkWidget *widget,
            gdouble    x,
            gdouble    y)
{
    cairo_t *cr;

    /* Paint to the surface, where we store our state */
    cr = cairo_create (surface);

    cairo_rectangle (cr, x - 3, y - 3, 6, 6);
    cairo_fill (cr);

    cairo_destroy (cr);

    /* Now invalidate the affected region of the drawing area. */
    gtk_widget_queue_draw_area (widget, x - 3, y - 3, 6, 6);

}

/* Handle button press events by either drawing a rectangle
* or clearing the surface, depending on which button was pressed.
* The ::button-press signal handler receives a GdkEventButton
* struct which contains this information.
*/
static gboolean button_press_event_cb (GtkWidget      *widget,
                       GdkEventButton *event,
                       gpointer        data)
{
    /* paranoia check, in case we haven't gotten a configure event */
    if (surface == NULL)
    return FALSE;

    if (event->button == GDK_BUTTON_PRIMARY)
    {
        draw_brush (widget, event->x, event->y);
    }
    else if (event->button == GDK_BUTTON_SECONDARY)
    {
        clear_surface ();
```

```
        gtk_widget_queue_draw (widget);

    }

    /* We've handled the event, stop processing */
    return TRUE;

}

/* Handle motion events by continuing to draw if button 1 is
* still held down. The ::motion-notify signal handler receives
* a GdkEventMotion struct which contains this information.
*/
static gboolean
motion_notify_event_cb (GtkWidget      *widget,
                        GdkEventMotion *event,
                        gpointer        data)
{
    /* paranoia check, in case we haven't gotten a configure event */
    if (surface == NULL)
    return FALSE;

    if (event->state & GDK_BUTTON1_MASK)
    draw_brush (widget, event->x, event->y);

    /* We've handled it, stop processing */
    return TRUE;

}

 static void close_window (void)
 {
     if (surface)
     cairo_surface_destroy (surface);

     gtk_main_quit ();

 }

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

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

    window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
    gtk_window_set_title (GTK_WINDOW (window), "Drawing Area");

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

    gtk_container_set_border_width (GTK_CONTAINER (window), 8);

    frame = gtk_frame_new (NULL);
    gtk_frame_set_shadow_type (GTK_FRAME (frame), GTK_SHADOW_IN);
    gtk_container_add (GTK_CONTAINER (window), frame);

    da = gtk_drawing_area_new ();
    /* set a minimum size */
    gtk_widget_set_size_request (da, 100, 100);

    gtk_container_add (GTK_CONTAINER (frame), da);

    /* Signals used to handle the backing surface */
    g_signal_connect (da, "draw",
                      G_CALLBACK (draw_cb), NULL);
    g_signal_connect (da,"configure-event",
                      G_CALLBACK (configure_event_cb), NULL);

    /* Event signals */
    g_signal_connect (da, "motion-notify-event",
                      G_CALLBACK (motion_notify_event_cb), NULL);
    g_signal_connect (da, "button-press-event",
                      G_CALLBACK (button_press_event_cb), NULL);

    /* Ask to receive events the drawing area doesn't normally
    * subscribe to. In particular, we need to ask for the
    * button press and motion notify events that want to handle.
    */
    gtk_widget_set_events (da, gtk_widget_get_events (da)
                           | GDK_BUTTON_PRESS_MASK
                           | GDK_POINTER_MOTION_MASK);

    gtk_widget_show_all (window);

    gtk_main ();

    return 0;
}
```

然后在终端输入以下命令用 GCC 编译程序：

```
gcc `pkg-config --cflags gtk+-3.0` -o example-3 example-3.c `pkg-config --libs gtk+-3.0`
```

2.4 构建用户界面

当我们构建一个更加复杂的带有成百控件的用户界面时，用 C 程序做这些控件的所有设置工作是非常麻烦的，而且也让做些调整变得几乎不可能。谢天谢地，GTK+ 支持将用户界面布局从业务逻辑中分离。这是一种通过 XML 格式实现的 UI 描述，它可以通过 `Gtkuilder` 类进行解析。

Example 4. Packing buttons with GtkBuilder

新建一个名为 `example-4.c` 的文件，写入如下内容：

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

static void print_hello (GtkWidget *widget,
             gpointer   data)
{
    g_print ("Hello World\n");

}

int main (int   argc, char *argv[])
{
    GtkBuilder *builder;
    GObject *window;
    GObject *button;

    gtk_init (&argc, &argv);

    /* Construct a GtkBuilder instance and load our UI description */
    builder = gtk_builder_new ();
    gtk_builder_add_from_file (builder, "builder.ui", NULL);

    /* Connect signal handlers to the constructed widgets. */
    window = gtk_builder_get_object (builder, "window");
    g_signal_connect (window, "destroy", G_CALLBACK (gtk_main_quit), NULL);

    button = gtk_builder_get_object (builder, "button1");
    g_signal_connect (button, "clicked", G_CALLBACK (print_hello), NULL);

    button = gtk_builder_get_object (builder, "button2");
    g_signal_connect (button, "clicked", G_CALLBACK (print_hello), NULL);

    button = gtk_builder_get_object (builder, "quit");
```

```
    g_signal_connect (button, "clicked", G_CALLBACK (gtk_main_quit), NULL);

    gtk_main ();

    return 0;
}
```

新建一个名为 builder.ui 的文件，写入如下内容：

```
<interface>
    <object id="window" class="GtkWindow">
        <property name="visible">True</property>
        <property name="title">Grid</property>
        <property name="border-width">10</property>
        <child>
            <object id="grid" class="GtkGrid">
                <property name="visible">True</property>
                <child>
                    <object id="button1" class="GtkButton">
                        <property name="visible">True</property>
                        <property name="label">Button 1</property>
                    </object>
                    <packing>
                        <property name="left-attach">0</property>
                        <property name="top-attach">0</property>
                    </packing>
                </child>
                <child>
                    <object id="button2" class="GtkButton">
                        <property name="visible">True</property>
                        <property name="label">Button 2</property>
                    </object>
                    <packing>
                        <property name="left-attach">1</property>
                        <property name="top-attach">0</property>
                    </packing>
                </child>
                <child>
                    <object id="quit" class="GtkButton">
                        <property name="visible">True</property>
                        <property name="label">Quit</property>
                    </object>
                    <packing>
                        <property name="left-attach">0</property>
                        <property name="top-attach">1</property>
                        <property name="width">2</property>
```

```
                    </packing>
                </child>
            </object>
            <packing>
            </packing>
        </child>
    </object>
</interface>
```

然后在终端输入以下命令用 GCC 编译程序：

```
gcc `pkg-config --cflags gtk+-3.0` -o example-4 example-4.c `pkg-config --libs gtk+-3.0`
```

注意 GtkBuilder 也可以用来构建非控件的对象，例如树结构，调节器。这也是我们这里使用的方法叫做 gtk_builder_get_object() 并且返回值为 GObject* 而不是 GtkWidget* 的原因。一般情况下，你将把一个完整路径传递给 gtk_builder_add_from_file() 使你的程序不依赖于当前路径运行。一个常用的放置 UI 描述和类似数据的目录是/usr/share/appname。

也可以将 UI 描述以字符串的形式嵌入到源代码中，然后使用 gtk_builder_add_from_string() 加载。但是将 UI 描述放置在一个单独的文件有几个好处：首先，这让我们在对 UI 进行调整时不需要重新编译程序，而且，更重要的是，一些 UI 编辑器比如 glade 可以加载这种文件并且允许你通过点击就能够创建和修改你的 UI。

2.5 构建应用程序

一个普通的应用程序由以下文件组成：

- 二进制文件 > 这个安装在 /usr/bin。
- 一个桌面文件 > 这个桌面文件向 shell 提供关于这个程序的重要信息, 例如名称、图标、D-Bus 名称，启动的命令行。安装在 /usr/share/applications.
- 一个图标 > 这个图标安装在 /usr/share/icons/hicolor/48x48/apps, 无论当前背景是什么系统都会到这里查找图标。
- 一个设置框架 > 如果应用使用了 GSettings, 它会将它的 schema 安装在 /usr/share/glib-2.0/schemas, 这样 dconf-editor 之类的工具就能够找到它。
- 其他资源 > 其他文件, 例如 GtkBuilder ui 文件, 最好从应用二进制文件自身储存的资源中加载。如果有需要，许多文件会按照惯例放置在/usr/share。

GTK+ includes application support that is built on top of GApplication. 在这篇教程中，我们从头开始构建一个简单的应用，然后逐渐一点一点增加功能。在这个过程中, 我们将会了解到 GtkApplication, templates, resources, application menus, settings, GtkHeaderBar, GtkStack, GtkSearchBar, GtkListBox 和更多东西。

完整的源文件可以在 GTK+ source distribution 的范例根目录下找到，或者可以在 GTK+ 的 git 仓库在线查看。

2.5.1 一个小应用

当使用 GtkApplication，main 主函数非常简单。我们仅仅调用了 `g_application_run()` 并给出一个应用范例。

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

int
main (int argc, char *argv[])
{
  return g_application_run (G_APPLICATION (example_app_new ()), argc, argv);
}
```

所有的应用程序逻辑都在 GtkApplicaton 的子类中。我们的范例还没有任何有趣的功能。它所做的只是当它没有传递参数而被激活时打开一个窗口，在传递了参数被激活时打开给定的文件。

为了处理这两种情况，我们重载了 activate()vfunc，当应用程序被加载没有命令行参数时它被调用，当应用程序被加载并带有命令行参数时，调用 open()vfunc。

想知道更多关于 GApplication 入口知识，请查看 GIO 文档。

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

#include "exampleapp.h"
#include "exampleappwin.h"

struct _ExampleApp
{
  GtkApplication parent;
};

struct _ExampleAppClass
{
  GtkApplicationClass parent_class;
};

G_DEFINE_TYPE(ExampleApp, example_app, GTK_TYPE_APPLICATION);

static void
example_app_init (ExampleApp *app)
{
}

static void
example_app_activate (GApplication *app)
{
```

```
  ExampleAppWindow *win;

  win = example_app_window_new (EXAMPLE_APP (app));
  gtk_window_present (GTK_WINDOW (win));
}

static void
example_app_open (GApplication  *app,
                  GFile        **files,
                  gint           n_files,
                  const gchar   *hint)
{
  GList *windows;
  ExampleAppWindow *win;
  int i;

  windows = gtk_application_get_windows (GTK_APPLICATION (app));
  if (windows)
    win = EXAMPLE_APP_WINDOW (windows->data);
  else
    win = example_app_window_new (EXAMPLE_APP (app));

  for (i = 0; i < n_files; i++)
    example_app_window_open (win, files[i]);

  gtk_window_present (GTK_WINDOW (win));
}

static void
example_app_class_init (ExampleAppClass *class)
{
  G_APPLICATION_CLASS (class)->activate = example_app_activate;
  G_APPLICATION_CLASS (class)->open = example_app_open;
}

ExampleApp *
example_app_new (void)
{
  return g_object_new (EXAMPLE_APP_TYPE,
                       "application-id", "org.gtk.exampleapp",
                       "flags", G_APPLICATION_HANDLES_OPEN,
                       NULL);
}
```

应用程序中另一个受 GTK+ 支持的重要的类是 GtkApplicationWindow。它通常也是子类。我们的子类不做任何事，因此我们只得到一个空的窗口。

```
#include "exampleapp.h"
#include "exampleappwin.h"
#include <gtk/gtk.h>

struct _ExampleAppWindow
{
  GtkApplicationWindow parent;
};

struct _ExampleAppWindowClass
{
  GtkApplicationWindowClass parent_class;
};

G_DEFINE_TYPE(ExampleAppWindow, example_app_window, GTK_TYPE_APPLICATION_WINDOW);

static void
example_app_window_init (ExampleAppWindow *app)
{
}

static void
example_app_window_class_init (ExampleAppWindowClass *class)
{
}

ExampleAppWindow *
example_app_window_new (ExampleApp *app)
{
  return g_object_new (EXAMPLE_APP_WINDOW_TYPE, "application", app, NULL);
}

void
example_app_window_open (ExampleAppWindow *win,
                         GFile            *file)
{
}
```

作为我们应用程序初始化中的一部分，我们创建一个图标和一个桌面文件。

```
[Desktop Entry]
Type=Application
```

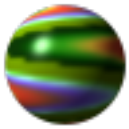


Figure 4: exampleapp.png

```
Name=Example
Icon=exampleapp
StartupNotify=true
Exec=@bindir@/exampleapp
```

注意 @bindir@ 需要被实际的二进制文件路径替代，这样桌面文件才能使用。

这就是目前我们实现的:

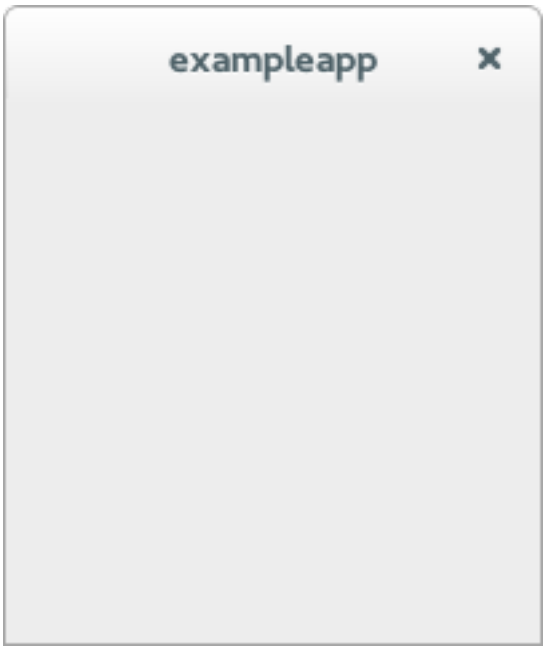


Figure 5: getting-started-app1.png

至今我们的程序并没那么瞩目，但是它已经在会话总线上出现，它有单个实例，而且它接受文件作为命令行参数。

2.5.2 填充窗口

在这节中，我们用 GtkBuilder 模板结合一个 GtkBuilder ui 文件和我们的应用程序窗口类。

我们简单的 ui 文件把 GtkHeaderBar 放在 GtkStack 部件顶端。头栏包括一个显示 GtkStack 页面分页的一行的独立部件 ------GtkStackSwitcher。

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
```

```
  <!-- interface-requires gtk+ 3.8 -->
  <template class="ExampleAppWindow" parent="GtkApplicationWindow">
    <property name="title" translatable="yes">Example Application</property>
    <property name="default-width">600</property>
    <property name="default-height">400</property>
    <child>
      <object class="GtkBox" id="content_box">
        <property name="visible">True</property>
        <property name="orientation">vertical</property>
        <child>
          <object class="GtkHeaderBar" id="header">
            <property name="visible">True</property>
            <child type="title">
              <object class="GtkStackSwitcher" id="tabs">
                <property name="visible">True</property>
                <property name="margin">6</property>
                <property name="stack">stack</property>
              </object>
            </child>
          </object>
        </child>
        <child>
          <object class="GtkStack" id="stack">
            <property name="visible">True</property>
          </object>
        </child>
      </object>
    </child>
  </template>
</interface>
```

为了在我们的应用程序中使用这个文件，我们回到我们的 GtkApplicationWindow 子类，从类初始化函数中调用 gtk_widget_class_set_template_from_resource() 来把 ui 文件设为这个类的模板。在实例初始化函数中我们增加 gtk_widget_init_template() 去为我们的类的个体实例化模板。

``` ...

static void example_app_window_init (ExampleAppWindow *win) { gtk_widget_init_template (GTK_WIDGET (win)); }

static void example_app_window_class_init (ExampleAppWindowClass *class) { gtk_widget_class_set_template_from (GTK_WIDGET_CLASS (class), ``/org/gtk/exampleapp/window.ui''); }

...
```

(full source)

你也许注意到了，我们在函数中用了变量 _from_resource() 来设定一个模板。现在我们需要用 GLib 的资

源功能在二进制文件中包含一个 ui file。通常是在.gresource.xml 中列出所有资源，就像这样：

```
<?xml version="1.0" encoding="UTF-8"?>
<gresources>
  <gresource prefix="/org/gtk/exampleapp">
    <file preprocess="xml-stripblanks">window.ui</file>
  </gresource>
</gresources>
```

这个文件必须被转换成一个 C 源文件，这样它才能和其他源文件一起被编译链接进应用程序中。因此，我们使用了 glib-complie-resources：

```
glib- compile- resources exampleapp.gresource.xml -- target=resources.c --
generate-source
```

如今我们的应用程序就像这样：

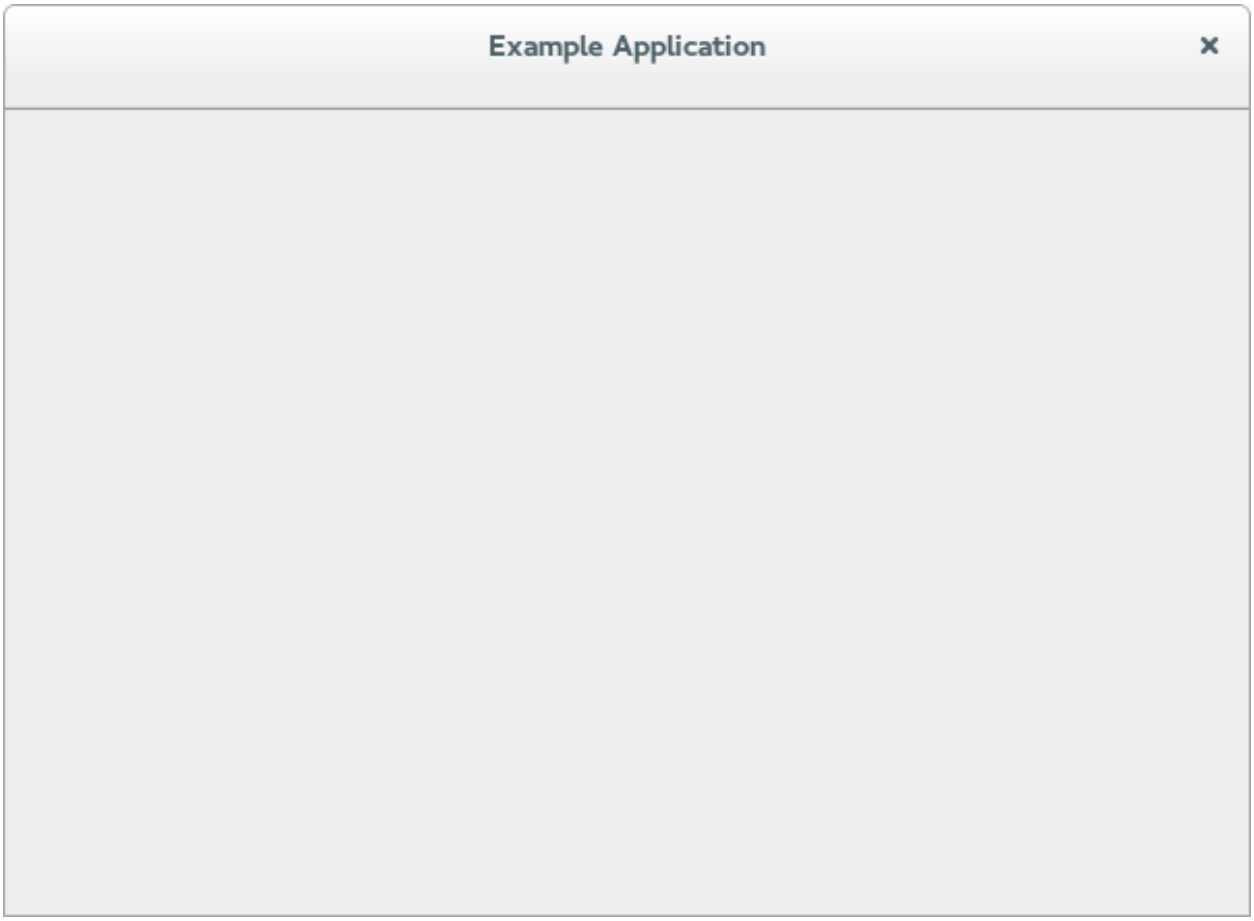


Figure 6: getting-started-app2.png

2.5.3 打开文件

在这节，我们使我们的应用程序展示命令行传来的文件的正文。

在这后面，我们为我们的应用程序的窗口子类增加了一个私有的结构体，结构体内是一个指向 GtkStack 的指针。`gtk_widget_class_bind_template_child_private()` 函数使得在实例化模板后，私有结构体中的 stack 成员会指向模板中的同名部件。

```
...

struct _ExampleAppWindowPrivate
{
  GtkWidget *stack;
};

G_DEFINE_TYPE_WITH_PRIVATE(ExampleAppWindow, example_app_window, GTK_TYPE_APPLICATION_WINDO

...

static void
example_app_window_class_init (ExampleAppWindowClass *class)
{
  gtk_widget_class_set_template_from_resource (GTK_WIDGET_CLASS (class),
                                               "/org/gtk/exampleapp/window.ui");
  gtk_widget_class_bind_template_child_private (GTK_WIDGET_CLASS (class), ExampleAppWindow,
}

...
```

(full source)

现在我们重新看一下在每个命令行参数中都会被调用的 `example_app_window_open()` 函数，然后构建 `GtkTextView`，它在后来的 stack 中作为一页被添加。

```
...

void
example_app_window_open (ExampleAppWindow *win,
                         GFile            *file)
{
  ExampleAppWindowPrivate *priv;
  gchar *basename;
  GtkWidget *scrolled, *view;
  gchar *contents;
  gsize length;

  priv = example_app_window_get_instance_private (win);
```

```
  basename = g_file_get_basename (file);

  scrolled = gtk_scrolled_window_new (NULL, NULL);
  gtk_widget_show (scrolled);
  gtk_widget_set_hexpand (scrolled, TRUE);
  gtk_widget_set_vexpand (scrolled, TRUE);
  view = gtk_text_view_new ();
  gtk_text_view_set_editable (GTK_TEXT_VIEW (view), FALSE);
  gtk_text_view_set_cursor_visible (GTK_TEXT_VIEW (view), FALSE);
  gtk_widget_show (view);
  gtk_container_add (GTK_CONTAINER (scrolled), view);
  gtk_stack_add_titled (GTK_STACK (priv->stack), scrolled, basename, basename);

  if (g_file_load_contents (file, NULL, &contents, &length, NULL, NULL))
    {
      GtkTextBuffer *buffer;

      buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (view));
      gtk_text_buffer_set_text (buffer, contents, length);
      g_free (contents);
    }

  g_free (basename);
}

...
```

(full source)

注意我们不一定非要接触 stack switcher。它从它属于的 stack 得到了自己所有的信息。在这里，我们传递 `gtk_stack_add_titled()` 函数的最后一个参数来显示每个文件的标签。

我们的程序打开后就像这样：

2.5.4 一个应用菜单

就像窗口模板，在一个 ui file 中我们指定了我们的应用程序菜单，然后作为资源向二进制文件中添加。

```
<?xml version="1.0"?>
<interface>
  <!-- interface-requires gtk+ 3.0 -->
  <menu id="appmenu">
    <section>
      <item>
        <attribute name="label" translatable="yes">_Preferences</attribute>
        <attribute name="action">app.preferences</attribute>
```

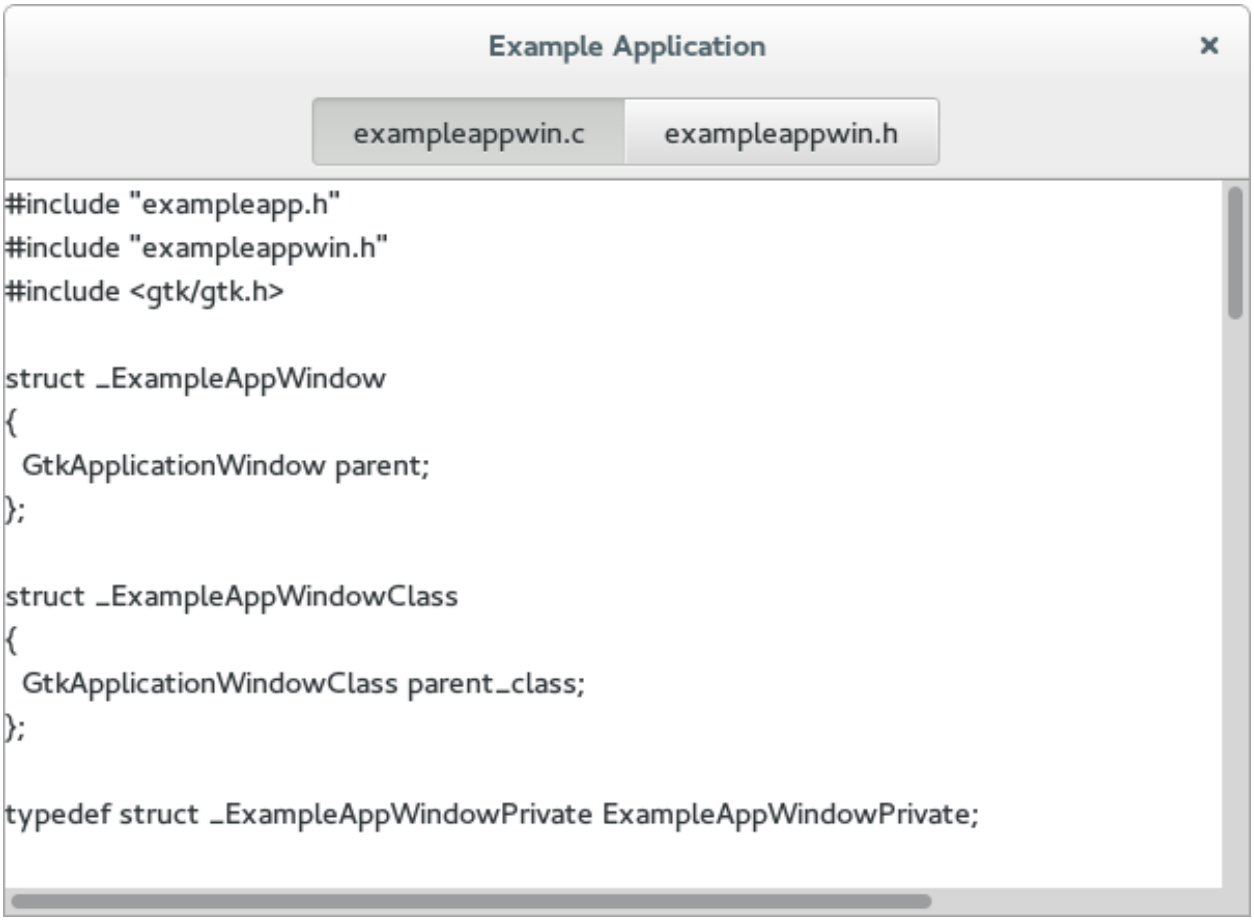


Figure 7: getting-started-app3.png

```
      </item>
    </section>
    <section>
      <item>
        <attribute name="label" translatable="yes">_Quit</attribute>
        <attribute name="action">app.quit</attribute>
      </item>
    </section>
  </menu>
</interface>
```

为了关联应用程序和应用菜单，我们必须调用 `gtk_application_set_app_menu()`。y 因为应用菜单被活动的 GActions 激活，所以必须为应用程序增加一个合适的设定。

所有这些任务最好在 startup() 函数中做完，因为 startup() 函数被保证在每个应用程序实例中只被调用一次。

```
...

static void
preferences_activated (GSimpleAction *action,
                       GVariant      *parameter,
                       gpointer       app)
{
}

static void
quit_activated (GSimpleAction *action,
                GVariant      *parameter,
                gpointer       app)
{
  g_application_quit (G_APPLICATION (app));
}

static GActionEntry app_entries[] =
{
  { "preferences", preferences_activated, NULL, NULL, NULL },
  { "quit", quit_activated, NULL, NULL, NULL }
};

static void
example_app_startup (GApplication *app)
{
  GtkBuilder *builder;
  GMenuModel *app_menu;
  const gchar *quit_accels[2] = { "<Ctrl>Q", NULL };
```

```
  G_APPLICATION_CLASS (example_app_parent_class)->startup (app);

  g_action_map_add_action_entries (G_ACTION_MAP (app),
                                   app_entries, G_N_ELEMENTS (app_entries),
                                   app);
  gtk_application_set_accels_for_action (GTK_APPLICATION (app),
                                         "app.quit",
                                         quit_accels);

  builder = gtk_builder_new_from_resource ("/org/gtk/exampleapp/app-menu.ui");
  app_menu = G_MENU_MODEL (gtk_builder_get_object (builder, "appmenu"));
  gtk_application_set_app_menu (GTK_APPLICATION (app), app_menu);
  g_object_unref (builder);
}

static void
example_app_class_init (ExampleAppClass *class)
{
  G_APPLICATION_CLASS (class)->startup = example_app_startup;
  ...
}

...
```

(full source)

菜单首选项如今并不能作任何事，但是 Quit 菜单选项的功能是正常的。注意它也可以被快捷键 `Ctrl-Q` 激活。这个快捷方式已经在 `gtk_application_set_accels_for_action()` 中被添加。

我们的应用菜单如下：

2.5.5 一个偏好对话框

一个典型的应用程序应该有一些偏好设置，在每次打开时都能被记住。即使是为这个小范例程序，我们也将想改变正文的字体。

我们将用 GSettings 来保存偏好设置，GSettings 需要一个描述我们设置的模式。

```
<?xml version="1.0" encoding="UTF-8"?>
<schemalist>
  <schema path="/org/gtk/exampleapp/" id="org.gtk.exampleapp">
    <key name="font" type="s">
      <default>'Monospace 12'</default>
      <summary>Font</summary>
      <description>The font to be used for content.</description>
    </key>
```

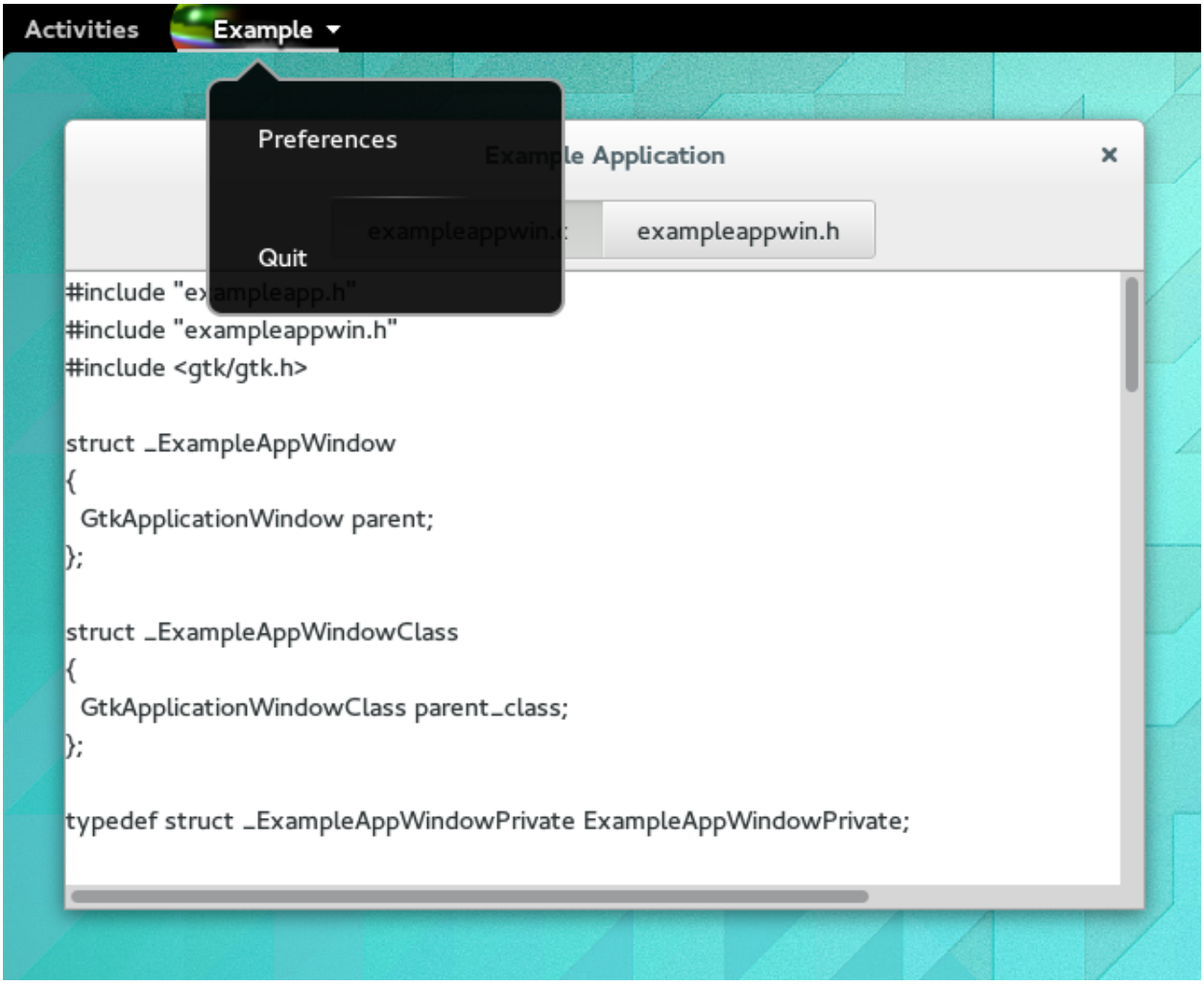


Figure 8: getting-started-app4.png

```
    <key name="transition" type="s">
      <choices>
        <choice value='none'/>
        <choice value='crossfade'/>
        <choice value='slide-left-right'/>
      </choices>
      <default>'none'</default>
      <summary>Transition</summary>
      <description>The transition to use when switching tabs.</description>
    </key>
  </schema>
</schemalist>
```

当我们在应用程序中使用这个模式之前，我们需要从 GSettings 中将这编译进二进制文件。GIO 提供 macros 来在工程中做这件事。

接着，我们需要连接 settings 和我们的目标部件。一个简便的方法是用 GSettings bind 函数绑定设定关键词和目标属性，就像我们这里为转换设置做的。

```
...

static void
example_app_window_init (ExampleAppWindow *win)
{
  ExampleAppWindowPrivate *priv;

  priv = example_app_window_get_instance_private (win);
  gtk_widget_init_template (GTK_WIDGET (win));
  priv->settings = g_settings_new ("org.gtk.exampleapp");

  g_settings_bind (priv->settings, "transition",
                   priv->stack, "transition-type",
                   G_SETTINGS_BIND_DEFAULT);
}

...
```

(full source)

这个连接字体设置的代码有点儿复杂，因为我们没有对应的简单的目标属性，我们本没打算这么做。

至此，如果我们改变一个设置，程序将会有反应，比如用 gsettings 命令行工具。当然，我们希望应用程序提供一个偏好对话框。所以干吧，我们的偏好对话框是 GtkDialog 的子类，我们将使用我们已经用过的技术：templates,private structs, settingbindings。

让我们从模板开始。

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
```

```
<!-- interface-requires gtk+ 3.8 -->
<template class="ExampleAppPrefs" parent="GtkDialog">
  <property name="title" translatable="yes">Preferences</property>
  <property name="resizable">False</property>
  <property name="modal">True</property>
  <child internal-child="vbox">
    <object class="GtkBox" id="vbox">
      <child>
        <object class="GtkGrid" id="grid">
          <property name="visible">True</property>
          <property name="margin">6</property>
          <property name="row-spacing">12</property>
          <property name="column-spacing">6</property>
          <child>
            <object class="GtkLabel" id="fontlabel">
              <property name="visible">True</property>
              <property name="label">_Font:</property>
              <property name="use-underline">True</property>
              <property name="mnemonic-widget">font</property>
              <property name="xalign">1</property>
            </object>
            <packing>
              <property name="left-attach">0</property>
              <property name="top-attach">0</property>
            </packing>
          </child>
          <child>
            <object class="GtkFontButton" id="font">
              <property name="visible">True</property>
            </object>
            <packing>
              <property name="left-attach">1</property>
              <property name="top-attach">0</property>
            </packing>
          </child>
          <child>
            <object class="GtkLabel" id="transitionlabel">
              <property name="visible">True</property>
              <property name="label">_Transition:</property>
              <property name="use-underline">True</property>
              <property name="mnemonic-widget">transition</property>
              <property name="xalign">1</property>
            </object>
            <packing>
              <property name="left-attach">0</property>
```

```
              <property name="top-attach">1</property>
            </packing>
          </child>
          <child>
            <object class="GtkComboBoxText" id="transition">
              <property name="visible">True</property>
              <items>
                <item translatable="yes" id="none">None</item>
                <item translatable="yes" id="crossfade">Fade</item>
                <item translatable="yes" id="slide-left-right">Slide</item>
              </items>
            </object>
            <packing>
              <property name="left-attach">1</property>
              <property name="top-attach">1</property>
            </packing>
          </child>
        </object>
      </child>
    </object>
  </child>
 </template>
</interface>
```

接下来是对话框子类。

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

#include "exampleapp.h"
#include "exampleappwin.h"
#include "exampleappprefs.h"

struct _ExampleAppPrefs
{
  GtkDialog parent;
};

struct _ExampleAppPrefsClass
{
  GtkDialogClass parent_class;
};

typedef struct _ExampleAppPrefsPrivate ExampleAppPrefsPrivate;

struct _ExampleAppPrefsPrivate
{
```

```
  GSettings *settings;
  GtkWidget *font;
  GtkWidget *transition;
};

G_DEFINE_TYPE_WITH_PRIVATE(ExampleAppPrefs, example_app_prefs, GTK_TYPE_DIALOG)

static void
example_app_prefs_init (ExampleAppPrefs *prefs)
{
  ExampleAppPrefsPrivate *priv;

  priv = example_app_prefs_get_instance_private (prefs);
  gtk_widget_init_template (GTK_WIDGET (prefs));
  priv->settings = g_settings_new ("org.gtk.exampleapp");

  g_settings_bind (priv->settings, "font",
                   priv->font, "font",
                   G_SETTINGS_BIND_DEFAULT);
  g_settings_bind (priv->settings, "transition",
                   priv->transition, "active-id",
                   G_SETTINGS_BIND_DEFAULT);
}

static void
example_app_prefs_dispose (GObject *object)
{
  ExampleAppPrefsPrivate *priv;

  priv = example_app_prefs_get_instance_private (EXAMPLE_APP_PREFS (object));
  g_clear_object (&priv->settings);

  G_OBJECT_CLASS (example_app_prefs_parent_class)->dispose (object);
}

static void
example_app_prefs_class_init (ExampleAppPrefsClass *class)
{
  G_OBJECT_CLASS (class)->dispose = example_app_prefs_dispose;

  gtk_widget_class_set_template_from_resource (GTK_WIDGET_CLASS (class),
                                               "/org/gtk/exampleapp/prefs.ui");
  gtk_widget_class_bind_template_child_private (GTK_WIDGET_CLASS (class), ExampleAppPrefs,
  gtk_widget_class_bind_template_child_private (GTK_WIDGET_CLASS (class), ExampleAppPrefs,
}
```

```
ExampleAppPrefs *
example_app_prefs_new (ExampleAppWindow *win)
{
  return g_object_new (EXAMPLE_APP_PREFS_TYPE, "transient-for", win, "use-header-bar", TRU
}
```

现在我们再看 `preferences_activated()` 函数，使它打开一个偏好对话框。

```
...

static void
preferences_activated (GSimpleAction *action,
                       GVariant      *parameter,
                       gpointer       app)
{
  ExampleAppPrefs *prefs;
  GtkWindow *win;

  win = gtk_application_get_active_window (GTK_APPLICATION (app));
  prefs = example_app_prefs_new (EXAMPLE_APP_WINDOW (win));
  gtk_window_present (GTK_WINDOW (prefs));
}

...
```

(full source)

完成所有这些工作后，我们的应用程序现在可以像这样显示一个偏好对话框：

2.5.6 增加搜索条

我们继续充实我们应用程序的功能。如今，我们添加搜索。GTK+ 在 `GtkSearchEntry` 和 `Gtksearchbar` 中支持这个功能。搜索条是一个可以嵌入顶端来展现搜索输入。

我们在头栏增加一个开关按钮，他可以用来滑出头栏下的搜索条。

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <!-- interface-requires gtk+ 3.8 -->
  <template class="ExampleAppWindow" parent="GtkApplicationWindow">
    <property name="title" translatable="yes">Example Application</property>
    <property name="default-width">600</property>
    <property name="default-height">400</property>
    <child>
      <object class="GtkBox" id="content_box">
```

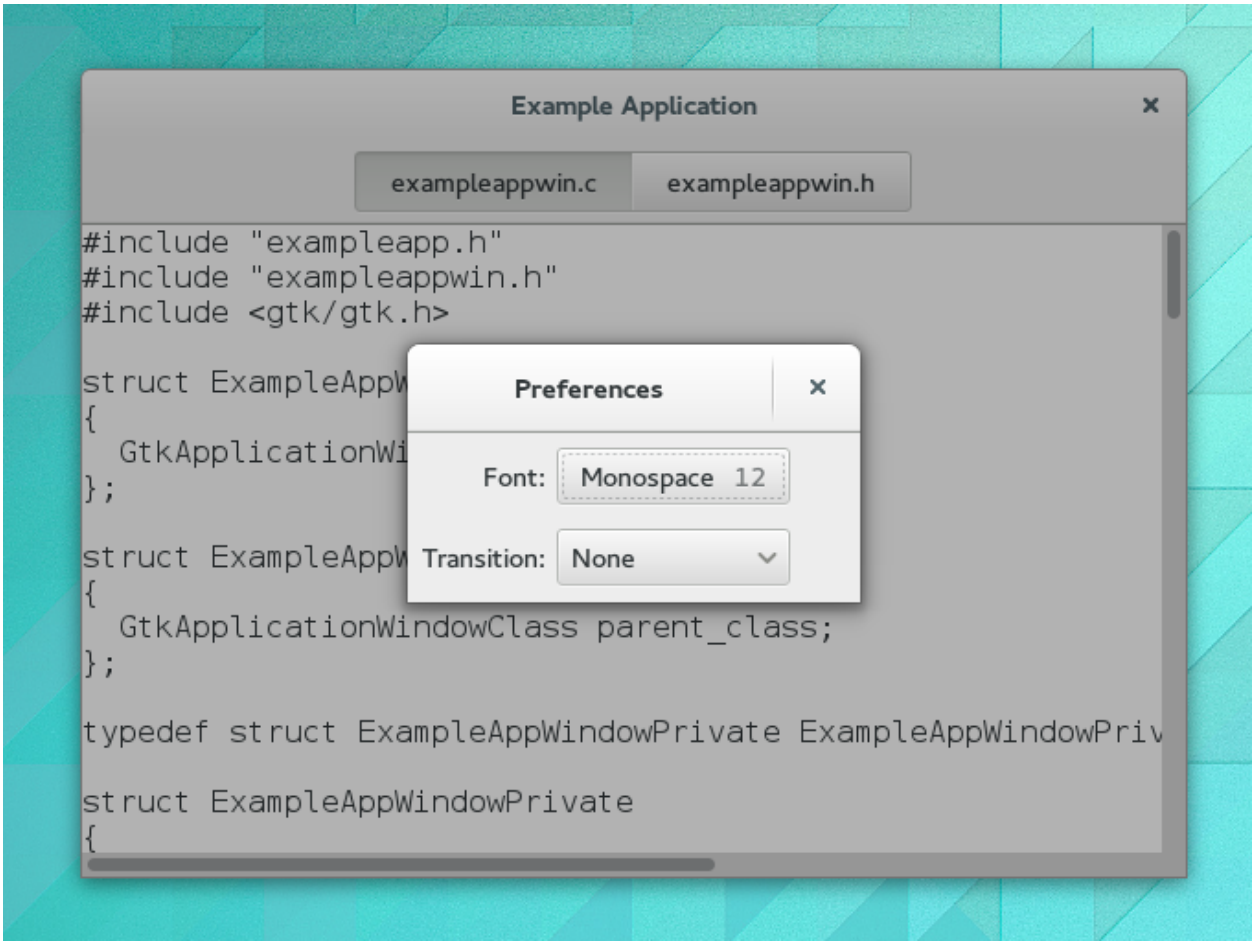


Figure 9: getting-started-app6.png

```
<property name="visible">True</property>
<property name="orientation">vertical</property>
<child>
  <object class="GtkHeaderBar" id="header">
    <property name="visible">True</property>
    <child type="title">
      <object class="GtkStackSwitcher" id="tabs">
        <property name="visible">True</property>
        <property name="margin">6</property>
        <property name="stack">stack</property>
      </object>
    </child>
    <child>
      <object class="GtkToggleButton" id="search">
        <property name="visible">True</property>
        <property name="sensitive">False</property>
        <style>
          <class name="image-button"/>
        </style>
        <child>
          <object class="GtkImage" id="search-icon">
            <property name="visible">True</property>
            <property name="icon-name">edit-find-symbolic</property>
            <property name="icon-size">1</property>
          </object>
        </child>
      </object>
      <packing>
        <property name="pack-type">end</property>
      </packing>
    </child>
  </object>
</child>
<child>
  <object class="GtkSearchBar" id="searchbar">
    <property name="visible">True</property>
    <child>
      <object class="GtkSearchEntry" id="searchentry">
        <signal name="search-changed" handler="search_text_changed"/>
        <property name="visible">True</property>
      </object>
    </child>
  </object>
</child>
<child>
```

```
          <object class="GtkStack" id="stack">
            <signal name="notify::visible-child" handler="visible_child_changed"/>
            <property name="visible">True</property>
          </object>
        </child>
      </object>
    </child>
  </template>
</interface>
```

实现搜索条需要更改一点我们还没打算完成的代码。搜索实现的核心是一个监听搜索条文字变化的信号句柄。

```
...

static void
search_text_changed (GtkEntry         *entry,
                     ExampleAppWindow *win)
{
  ExampleAppWindowPrivate *priv;
  const gchar *text;
  GtkWidget *tab;
  GtkWidget *view;
  GtkTextBuffer *buffer;
  GtkTextIter start, match_start, match_end;

  text = gtk_entry_get_text (entry);

  if (text[0] == '\0')
    return;

  priv = example_app_window_get_instance_private (win);

  tab = gtk_stack_get_visible_child (GTK_STACK (priv->stack));
  view = gtk_bin_get_child (GTK_BIN (tab));
  buffer = gtk_text_view_get_buffer (GTK_TEXT_VIEW (view));

  /* Very simple-minded search implementation */
  gtk_text_buffer_get_start_iter (buffer, &start);
  if (gtk_text_iter_forward_search (&start, text, GTK_TEXT_SEARCH_CASE_INSENSITIVE,
                                    &match_start, &match_end, NULL))
    {
      gtk_text_buffer_select_range (buffer, &match_start, &match_end);
      gtk_text_view_scroll_to_iter (GTK_TEXT_VIEW (view), &match_start,
                                    0.0, FALSE, 0.0, 0.0);
    }
```

```
}

static void
example_app_window_init (ExampleAppWindow *win)
{

...

  gtk_widget_class_bind_template_callback (GTK_WIDGET_CLASS (class), search_text_changed);

...

}

...
```

(full source)

加上了搜索条，我们的应用程序现在是这样的：

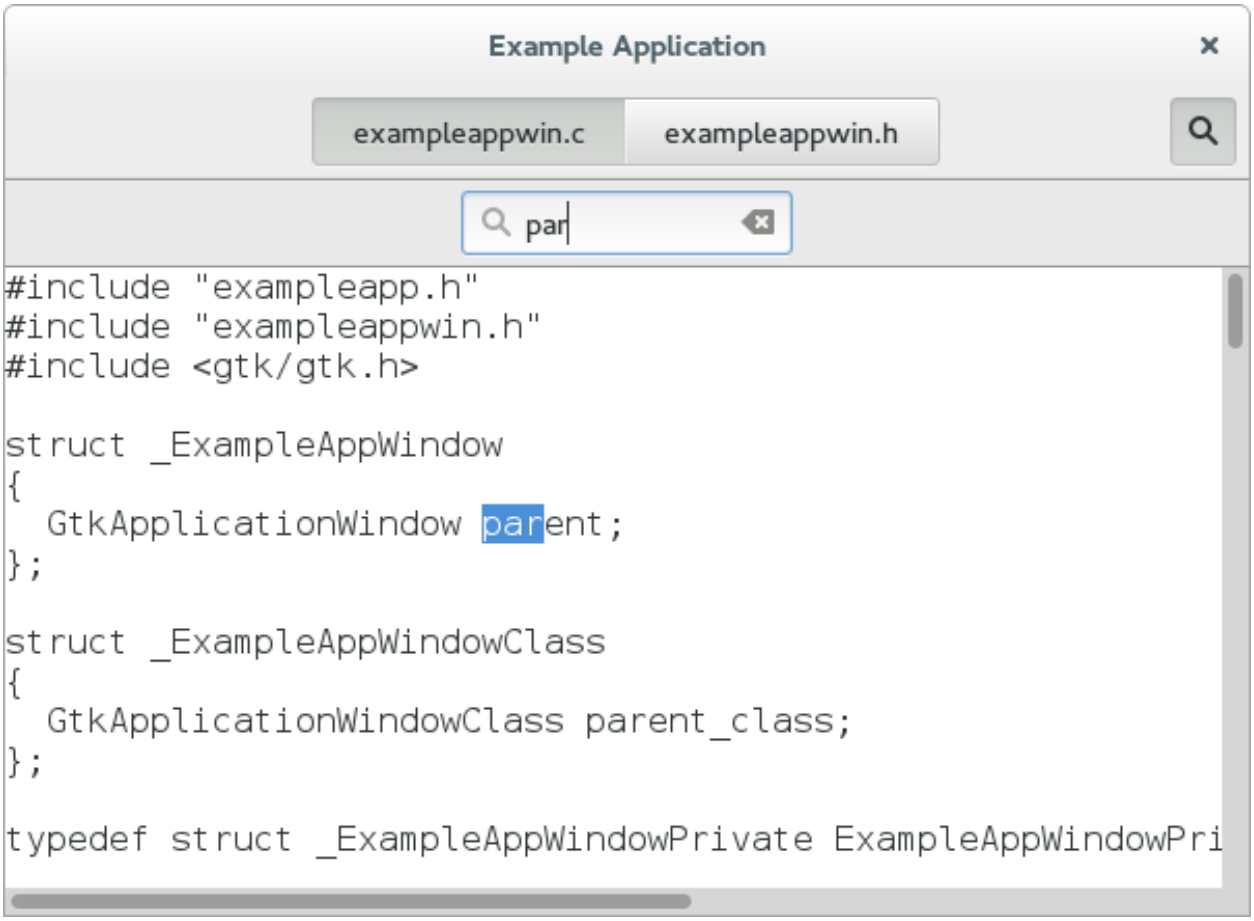


Figure 10: getting-started-app7.png

2.5.7 增加侧边栏

作为另一个实用的功能，我们增加一个显示 GtkMenuButton,GtkRevealer 和 GtkListBox 的侧边条。

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <!-- interface-requires gtk+ 3.8 -->
  <template class="ExampleAppWindow" parent="GtkApplicationWindow">
    <property name="title" translatable="yes">Example Application</property>
    <property name="default-width">600</property>
    <property name="default-height">400</property>
    <child>
      <object class="GtkBox" id="content_box">
        <property name="visible">True</property>
        <property name="orientation">vertical</property>
        <child>
          <object class="GtkHeaderBar" id="header">
            <property name="visible">True</property>
            <child type="title">
              <object class="GtkStackSwitcher" id="tabs">
                <property name="visible">True</property>
                <property name="margin">6</property>
                <property name="stack">stack</property>
              </object>
            </child>
            <child>
              <object class="GtkToggleButton" id="search">
                <property name="visible">True</property>
                <property name="sensitive">False</property>
                <style>
                  <class name="image-button"/>
                </style>
                <child>
                  <object class="GtkImage" id="search-icon">
                    <property name="visible">True</property>
                    <property name="icon-name">edit-find-symbolic</property>
                    <property name="icon-size">1</property>
                  </object>
                </child>
              </object>
              <packing>
                <property name="pack-type">end</property>
              </packing>
            </child>
            <child>
```

```
        <object class="GtkMenuButton" id="gears">
          <property name="visible">True</property>
          <property name="direction">none</property>
          <property name="use-popover">True</property>
          <style>
            <class name="image-button"/>
          </style>
        </object>
        <packing>
          <property name="pack-type">end</property>
        </packing>
      </child>
    </object>
  </child>
  <child>
    <object class="GtkSearchBar" id="searchbar">
      <property name="visible">True</property>
      <child>
        <object class="GtkSearchEntry" id="searchentry">
          <signal name="search-changed" handler="search_text_changed"/>
          <property name="visible">True</property>
        </object>
      </child>
    </object>
  </child>
  <child>
    <object class="GtkBox" id="hbox">
      <property name="visible">True</property>
      <child>
        <object class="GtkRevealer" id="sidebar">
          <property name="visible">True</property>
          <property name="transition-type">slide-right</property>
          <child>
           <object class="GtkScrolledWindow" id="sidebar-sw">
             <property name="visible">True</property>
             <property name="hscrollbar-policy">never</property>
             <property name="vscrollbar-policy">automatic</property>
             <child>
               <object class="GtkListBox" id="words">
                 <property name="visible">True</property>
                 <property name="selection-mode">none</property>
               </object>
             </child>
           </object>
          </child>
```

```
            </object>
          </child>
          <child>
            <object class="GtkStack" id="stack">
              <signal name="notify::visible-child" handler="visible_child_changed"/>
              <property name="visible">True</property>
            </object>
          </child>
        </object>
      </child>
    </object>
  </child>
</template>
</interface>
```

这些代码将每个文件中相关的词做成按钮显示在侧边条上。但我们将考虑用这些代码去添加一个工具菜单。像我们所希望的，这个工具菜单在一个 `GtkBuilder` ui file 中被指定。

```
<?xml version="1.0"?>
<interface>
  <!-- interface-requires gtk+ 3.0 -->
  <menu id="menu">
    <section>
      <item>
        <attribute name="label" translatable="yes">_Words</attribute>
        <attribute name="action">win.show-words</attribute>
      </item>
    </section>
  </menu>
</interface>
```

为了连接菜单项和 show-words 设置，我们用了 `GAction` 对应于给定的 `GSettings`。

```
...

static void
example_app_window_init (ExampleAppWindow *win)
{

...

  builder = gtk_builder_new_from_resource ("/org/gtk/exampleapp/gears-menu.ui");
  menu = G_MENU_MODEL (gtk_builder_get_object (builder, "menu"));
  gtk_menu_button_set_menu_model (GTK_MENU_BUTTON (priv->gears), menu);
  g_object_unref (builder);
```

```
  action = g_settings_create_action (priv->settings, "show-words");
  g_action_map_add_action (G_ACTION_MAP (win), action);
  g_object_unref (action);
}

...
```

(full source)

我们的应用程序如今是这样的：

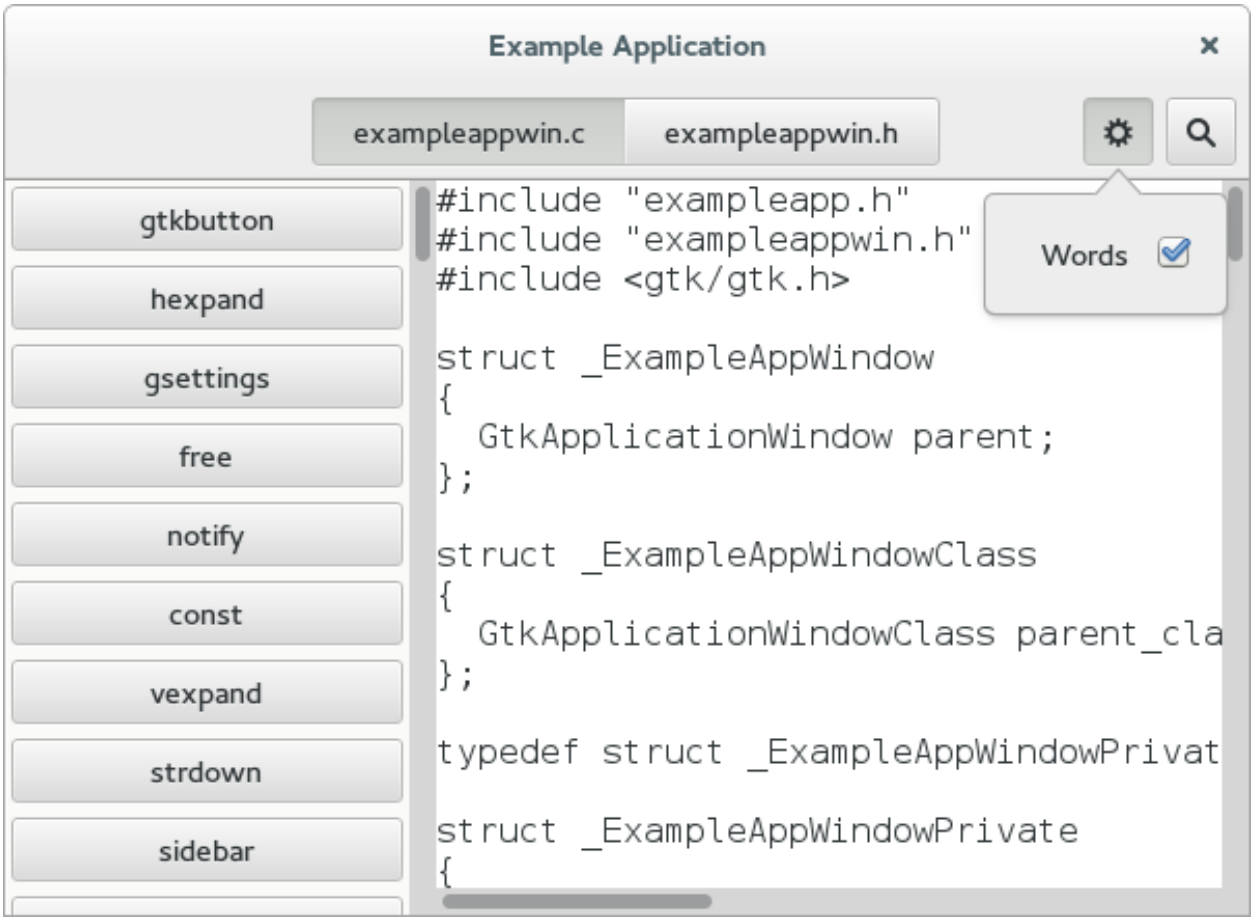


Figure 11: getting-started-app8.png

2.5.8 属性

部件和其他的对象有许多有用的属性。

这里我们展示一些灵活的新方法来使用它们，可以通过 `GPropertyAction` 包装在 action 中，也可以用 `GBinding` 来绑定它们。

着手干吧，我们在窗口模板头栏增加两个 lable，分别为 lines_label 和 lines，然后在一个私有结构体中将它们和结构体成员绑定，就像我们前 2 次做的一样。

我们在工具菜单上增加一个新的 Lines 菜单项，它负责触发 show-lines 动作。

```
<?xml version="1.0"?>
<interface>
  <!-- interface-requires gtk+ 3.0 -->
  <menu id="menu">
    <section>
      <item>
        <attribute name="label" translatable="yes">_Words</attribute>
        <attribute name="action">win.show-words</attribute>
      </item>
      <item>
        <attribute name="label" translatable="yes">_Lines</attribute>
        <attribute name="action">win.show-lines</attribute>
      </item>
    </section>
  </menu>
</interface>
```

为了使这个菜单项起作用，我们为 lines label 的可见属性添加了一个属性动作，然后将它添加进了窗口动作。效果就是，每次 lable 一可见，该动作就被触发。

因为我们希望所有的 label 都能一起显示和消失，我们将 lines-label 部件的可见属性和 lines 部件相同属性绑定。

```
...

static void
example_app_window_init (ExampleAppWindow *win)
{
  ...

  action = (GAction*) g_property_action_new ("show-lines", priv->lines, "visible");
  g_action_map_add_action (G_ACTION_MAP (win), action);
  g_object_unref (action);

  g_object_bind_property (priv->lines, "visible",
                          priv->lines_label, "visible",
                          G_BINDING_DEFAULT);
}

...
```

(full source)

我们需要一个计算当前活动标签行数的函数，然后更新 lines label。如果你对细节感兴趣，请看全部源代码。这使我们的范例程序如下所示：

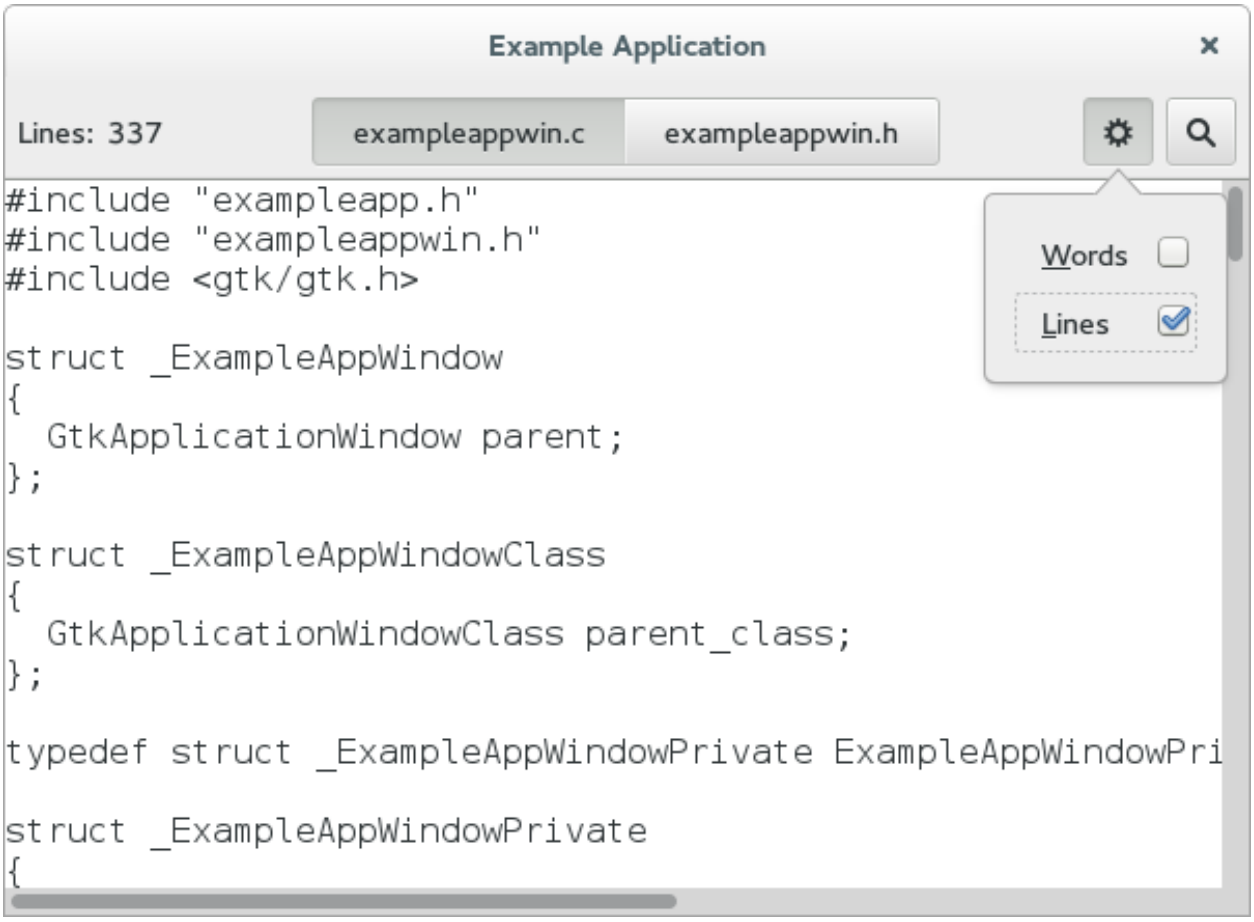


Figure 12: getting-started-app9.png

2.5.9 标题栏

我们的应用程序已经用了 `GtkHeaderBar`，但至今它仍然只在顶端显示一个 `正常' 的 window titlebar。这有点多余，我们现在要用 header bar 来替代 titlebar。为了达到目的，我们将 header bar 移到窗口的直接子成员中，并把它设为 titlebar。

```
<?xml version="1.0" encoding="UTF-8"?>
<interface>
  <!-- interface-requires gtk+ 3.8 -->
  <template class="ExampleAppWindow" parent="GtkApplicationWindow">
    <property name="title" translatable="yes">Example Application</property>
    <property name="default-width">600</property>
    <property name="default-height">400</property>
        <child type="titlebar">
          <object class="GtkHeaderBar" id="header">
```

```
<property name="visible">True</property>
<property name="show-close-button">True</property>
<child>
  <object class="GtkLabel" id="lines_label">
    <property name="visible">False</property>
    <property name="label" translatable="yes">Lines:</property>
  </object>
  <packing>
    <property name="pack-type">start</property>
  </packing>
</child>
<child>
  <object class="GtkLabel" id="lines">
    <property name="visible">False</property>
  </object>
  <packing>
    <property name="pack-type">start</property>
  </packing>
</child>
<child type="title">
  <object class="GtkStackSwitcher" id="tabs">
    <property name="visible">True</property>
    <property name="margin">6</property>
    <property name="stack">stack</property>
  </object>
</child>
<child>
  <object class="GtkToggleButton" id="search">
    <property name="visible">True</property>
    <property name="sensitive">False</property>
    <style>
      <class name="image-button"/>
    </style>
    <child>
      <object class="GtkImage" id="search-icon">
        <property name="visible">True</property>
        <property name="icon-name">edit-find-symbolic</property>
        <property name="icon-size">1</property>
      </object>
    </child>
  </object>
  <packing>
    <property name="pack-type">end</property>
  </packing>
</child>
```

```
          <child>
            <object class="GtkMenuButton" id="gears">
              <property name="visible">True</property>
              <property name="direction">none</property>
              <property name="use-popover">True</property>
              <style>
                <class name="image-button"/>
              </style>
            </object>
            <packing>
              <property name="pack-type">end</property>
            </packing>
          </child>
        </object>
      </child>
    <child>
      <object class="GtkBox" id="content_box">
        <property name="visible">True</property>
        <property name="orientation">vertical</property>
        <child>
          <object class="GtkSearchBar" id="searchbar">
            <property name="visible">True</property>
            <child>
              <object class="GtkSearchEntry" id="searchentry">
                <signal name="search-changed" handler="search_text_changed"/>
                <property name="visible">True</property>
              </object>
            </child>
          </object>
        </child>
        <child>
          <object class="GtkBox" id="hbox">
            <property name="visible">True</property>
            <child>
              <object class="GtkRevealer" id="sidebar">
                <property name="visible">True</property>
                <property name="transition-type">slide-right</property>
                <child>
                 <object class="GtkScrolledWindow" id="sidebar-sw">
                   <property name="visible">True</property>
                   <property name="hscrollbar-policy">never</property>
                   <property name="vscrollbar-policy">automatic</property>
                   <child>
                     <object class="GtkListBox" id="words">
                       <property name="visible">True</property>
```

```
                    <property name="selection-mode">none</property>
                  </object>
                </child>
              </object>
            </child>
          </object>
        </child>
        <child>
          <object class="GtkStack" id="stack">
            <signal name="notify::visible-child" handler="visible_child_changed"/>
            <property name="visible">True</property>
          </object>
        </child>
      </object>
    </child>
  </object>
</child>
</template>
</interface>
```

使用 header bar 的一个额外的好处是我们免费得到了一个回退项。如果这回退应用了，我们的应用程序将如下显示。

如果我们为窗口设定了图标，那么菜单按钮就是设定好的图标，而不是你现在看到的样子。

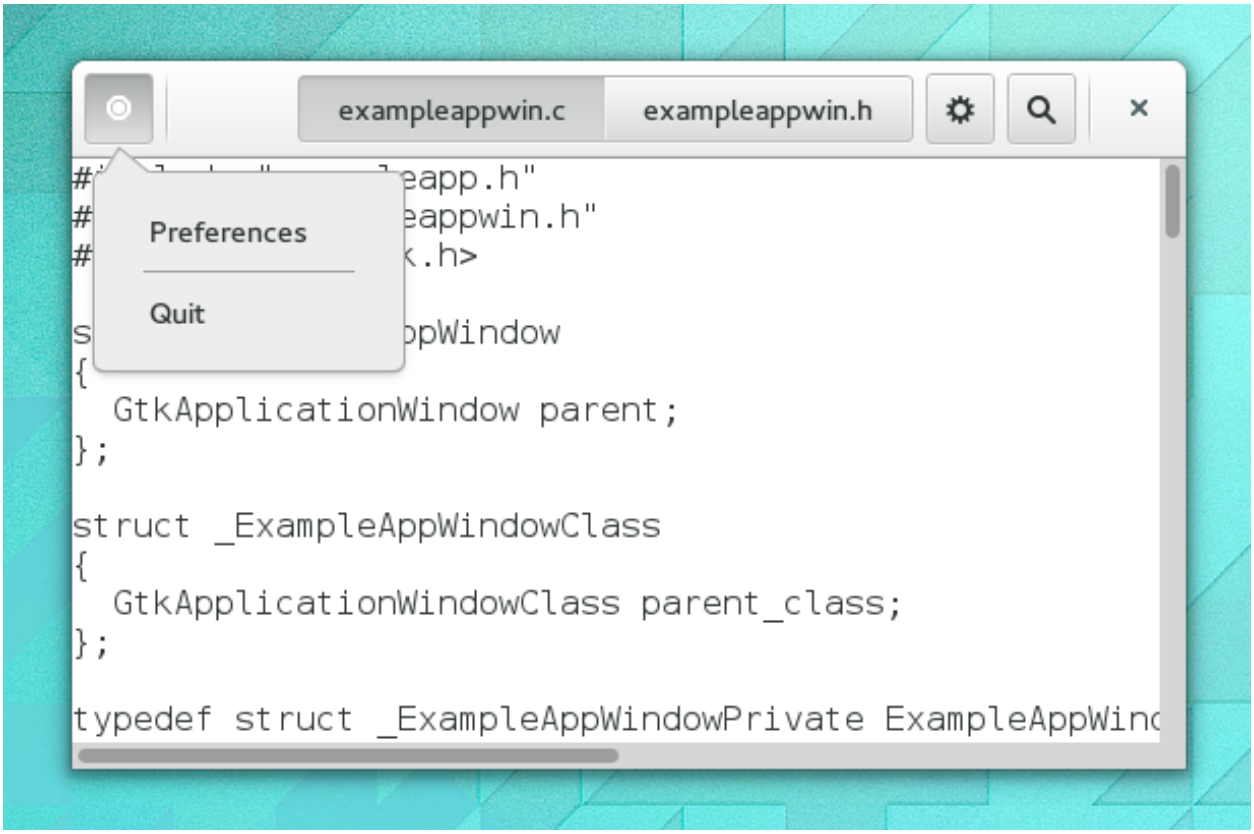


Figure 13: getting-started-app10.png