第16章 GDK 基 础

16.1 GDK和Xlib

GTK是用于实现图形用户接口的函数库。在 Linux平台上，GUI（图形用户接口）使用的是称为X 窗口（X Window）的系统。X窗口系统是1984年由美国麻省理工学院（MIT）开发的。在Linux上使用的 X窗口系统是一种称为 XFree86的X版本。 X窗口系统与 Microsoft Windows的图形用户接口有所不同，它是基于客户 /服务器的。X服务器在计算机上运行，控制监视器、鼠标和键盘。 X客户通过网络与服务器通讯。 X服务器为X客户提供图形显示服务。也就是说，X客户和X服务器可能在同一台计算机上运行，也可能在不同的计算机上运行。

X窗口系统带有一套低级的库函数，称为 Xlib。Xlib提供了许多对X窗口的屏幕进行操作的函数。当然，使用 Xlib函数在屏幕上创建构件是很复杂的。 GTK要在屏幕上绘制各种构件，就需要与 X服务器打交道。但是 GTK提供的构件库并未直接使用 Xlib，而是使用了一个称为GDK的库。GDK的意思是GIMP Drawing Toolkit，亦即GIMP绘图工具包。差不多每个 Gdk函数都是一个相应 Xlib函数的封装。但是 Xlib的某些复杂性(和功能)被隐藏起来了。这样是为了简化编程，使 Gdk更容易移植到其他窗口系统 (有一个在Windows平台上的 Gdk版本)。被隐藏的Xlib功能一般是程序员极少用到的，例如， Xlib的许多特性只有窗口管理器才会用到，所以没有封装到Gdk当中。如果需要，可以在应用程序中直接调用 Xlib函数，只要在文件头部包含 gdk/gdkx.h头文件就可以了。

一般情况下，如果要创建普通的图形接口应用程序，使用 GTK就可以了。Gtk+和Gnome构件库提供了极为丰富的构件，足以构造非常复杂的用户界面。但是，如果需要开发新构件，或者要创建绘图程序，仅使用 GTK就不够了。这时可以采用 Xlib，更好的方法是使用 GDK库，它可以应付绝大多数的编程需要。

本章介绍了关于GDK的一些基本知识，这些也是创建构件和绘图的基础。更多的 GDK细节内容，请参考 gdk.h头文件。

如果想了解 Gdk函数的实现细节（比如它对应于 Xlib的哪一个函数），可以看一下 Gdk的源代码以确定它所封装的 Xlib函数，然后用 man指令参看该函数的手册页。例如，下面是 gdk_draw_point()函数的实现代码：

```
void
gdk_draw_point (GdkDrawable *drawable,
                GdkGC       *gc,
                gint         x,
                gint         y)
{
  GdkWindowPrivate *drawable_private;
  GdkGCPrivate *gc_private;

  g_return_if_fail (drawable != NULL);
  g_return_if_fail (gc != NULL);
```

```
    drawable_private = (GdkWindowPrivate*) drawable;
    if (drawable_private->destroyed)
      return;
    gc_private = (GdkGCPrivate*) gc;

    XDrawPoint (drawable_private->xdisplay, drawable_private->xwindow,
                gc_private->xgc, x, y);
}
```

每一个数据结构都被转换给它的一个“私有”版本，该“私有”版本包含了与 GDK正在使用的特定窗口系统的相关信息，这样可以将与特定窗口系统相关的函数声明排除在gdk/gdk.h 头文件外。每个数据结构的“私有”版本都包含一个封装的 Xlib数据结构，且这个数据结构被传递到 XDrawPoint()函数中，所以 XDrawPoint()函数的文档也适用于gdk_draw_point()函数。

16.2 GdkWindow

GdkWindow是Xlib窗口对象的封装。一个GdkWindow 代表屏幕上的一个区域，可以显示或隐藏起来(在Xlib里面称为映射或反映射窗口)，也可以捕获GdkWindow接收到的事件，还可以在里面绘制图像，移动或调整图像的尺寸。 GdkWindow 是以树状结构组织的，也就是说，每一个窗口都可以有子窗口。子窗口是相对于父窗口的位置定位的，当父窗口移动时，子窗口也会移动。子窗口不会在父窗口边界外的区域绘出 (也就是说，它们会被父窗口剪裁) 。

所谓GdkWindow窗口的树状组织并不是针对每个应用程序的，实际上有一个由 X服务器和窗口管理器控制的窗口的全局树。根窗口没有父窗口，所有窗口都是从它派生而来的。作为桌面背景，根窗口的全部或部分总是可见的。每个窗口都可以为不同的 Linux进程所拥有，一些窗口是由窗口管理器所创建的，还有一些来自于用户的应用程序。

GdkWindow和GtkWindow是完全不同的东西。GtkWindow是一个Gtk+构件，用于表示顶级(toplevel)窗口 (顶级窗口是在窗口层次中由应用程序控制的最高级别的窗口)。典型情况下，窗口管理器为顶级窗口创建各种装饰，包括标题条、关闭按钮以及窗口外观等。

要理解X窗口首先要知道它是 X服务器上的一个对象，这一点很重要。 X客户对每一个窗口获得一个独一无二的整数 ID号，并用ID号码引用该窗口。这样，所有的窗口操作都发生在服务器上，并且所有与 X窗口打交道的函数都要通过网络传输。

GdkWindow是由X返回的整数ID号的一个封装。它确实保存一些信息的本地拷贝 (比如说窗口的尺寸)，所以一些Gdk操作比相应的Xlib操作效率更高。还有， GdkWindow本质上是服务器端对象的一个句柄。许多Gdk对象都是相似的，字体、像素映射图片、鼠标光标等等也是服务器端对象的句柄。

16.2.1 GdkWindow和GtkWidget

许多GtkWidget子类都有一个相关联的 GdkWindow窗口。理论上， Gtk+应用程序可以只创建一个顶级窗口，并将所有构件画在里面。然而，这么做并没有什么实际意义，因为GdkWindow允许X窗口系统自动处理许多细节。例如， Gdk所接收到的事件都使用它们所发生的窗口标志，这使得Gtk+能很快确定每个事件是哪个构件发生的。

有一些构件是没有与之相关联的 GdkWindow窗口的，它们称为“无窗口”的构件，用一

个GTK_NO_WINDOW标记来标志它们(可以用GTK_WIDGET_NO_WINDOW()宏来测试它们)。没有窗口的构件会将自身绘制在其父构件容器的 GdkWindow窗口上。无窗口构件相对较小，占用资源较少，一般将它们称为“轻量级”的， GtkLabel构件就是一个最常见的例子。因为事件总是由GdkWindow窗口接收的，无窗口构件不能接收事件。如果想让无窗口构件捕获事件，可以使用GtkEventBox容器构件。

16.2.2 GdkWindow属性

创建GdkWindow时，gdk_window_new() 函数允许指定窗口的所有属性，这些属性以后也可以再改变。要指定多个属性，可以向函数传递一个 GdkWindowAttr对象。GdkWindowAttr对象的内容就是GdkWindow窗口的属性。下面是GdkWindowAttr结构的定义：

```
typedef struct _GdkWindowAttr GdkWindowAttr;
struct _GdkWindowAttr
{
  gchar *title;
  gint event_mask;
  gint16 x, y;
  gint16 width;
  gint16 height;
  GdkWindowClass wclass;
  GdkVisual *visual;
  GdkColormap *colormap;
  GdkWindowType window_type;
  GdkCursor *cursor;
  gchar *wmclass_name;
  gchar *wmclass_class;
  gboolean override_redirect;
};
```

因为GdkWindowAttr结构的一些成员是可选的，所以 gdk_window_new()函数用一个attributes_mask参数指定哪一个可选成员里包含有效的数据。 Gdk 只检查在屏蔽值里面指定的成员，这样可以让不感兴趣的成员保留缺省值。下面的函数列表简要概括了这些值。没有attributes_mask标志的成员必须指定，因为它们没有缺省值。

gdk_window_new()函数最典型的是用在构件的实现中，用来创建构件的 GdkWindow。在其他场合下极少用到它。 gdk_window_destroy()函数销毁GdkWindow窗口。

函数列表： GdkWindow

```
#include <gdk/gdk.h>
GdkWindow* gdk_window_new(GdkWindow* parent,
              GdkWindowAttr* attributes,
              gint attributes_mask)

void gdk_window_destroy(GdkWindow* window)
```

下面简要介绍GdkWindowAttr 结构中各个成员的意义。

第一个成员title是GdkWindow的标题，它只对顶级窗口才有实际意义，大多数窗口管理器把它放在标题条上。通常，不要在创建 GdkWindow窗口时指定 title值，应该让用户调用gtk_window_set_title()函数来指定它。

第二个成员event_mask是窗口的事件屏蔽，它决定这个窗口接收什么事件。后面会有详细的介绍。

第三个和第四个成员x、y是窗口的X、Y坐标，它们是以像素度量的；这两个坐标是相对于父窗口原点的坐标。每个窗口的原点都是它的左上角 (西北角)。注意，它们是16位的有符号整数。X窗口最大可以是32 768像素，它可以是负数值，不过会被它的父窗口剪裁掉 (只有在父窗口内部的区域才是可见的)。第五个和第六个成员width、height是窗口的宽度和高度，它们是以像素度量的，也是16位的有符号整数。

第七个成员GdkWindowClass可取以下两种值：

- GDK_INPUT_OUTPUT：GdkWindow是一个普通窗口
- GDK_INPUT_ONLY：GdkWindow是一个窗口，它有一个位置，能接收事件，但没有视觉上的表示，也就是说它是不可见的。它的子窗口也必须是这种类型的。你可以为这种窗口设置光标和其他的属性，但是没有办法把窗口画出来 (因为它是不可见的)。这种窗口有时候用于捕获事件或改变两个普通窗口重叠区域的鼠标光标。

第八个成员visual（视件）描述了一个显示器的颜色处理特征；第九个成员 colormap（颜色表）包含了用于绘画的颜色。

第十个成员window_type指定GdkWindow的类型。窗口可以是下面几种不同类型之一，由GdkWindowType枚举类型指定：

- GDK_WINDOW_ROOT：是根窗口的Gdk封装类型，在初始化时创建。
- GDK_WINDOW_TOPLEVEL：是一个顶级窗口。在这种情况下， gdk_window_new()函数的parent参数应该设为NULL。Gdk自动使用根窗口作为它的父窗口。
- GDK_WINDOW_CHILD：是一个在顶级窗口中的下级窗口。
- GDK_WINDOW_DIALOG：基本上与顶级窗口是一样的。它的父窗口应该是 NULL，并且Gdk会替换根窗口。应该设置一个窗口类提示（ wmclass_name）告诉窗口管理器这个窗口是一个对话框，一些窗口管理器会考虑这些情况并作适当的处理。
- GDK_WINDOW_TEMP：用于弹出菜单或其他类似的东西。它是一个只需暂时存在的窗口。它是一个顶级窗口，所以它的父窗口应该是 NULL。这种窗口的鼠标光标总是与它们父窗口的一样。所以它们会忽略属性结构中的相关成员值。
- GDK_WINDOW_PIXMAP：根本就不是窗口。在 Gdk中GdkPixmap和GdkWindow差不多是同样处理的，所以Gdk用同样的结构表示它们；它们可以视为 GdkDrawable类型。
- GDK_WINDOW_FOREIGN：标识一个不是由Gdk创建的窗口的封装。

对gdk_window_new()来说，只有GDK_WINDOW_TOPLEVEL、GDK_WINDOW_CHILD、GDK_WINDOW_TEMP和GDK_WINDOW_DIALOG是有效的。库用户不会创建一个GDK_WINDOW_ROOT。pixmap(GDK_WINDOW_PIXMAP)是用gdk_pixmap_new()创建的。外来窗口 (GDK_WINDOW_FOREIGN)是在Gdk外部创建并用 gdk_window_foreign_new()封装的X窗口。

第十一个成员cursor指定在这个窗口中的鼠标的指针 (光标)形状。

第十二个成员wmclass_name前面已经介绍过了。写构件时，通常不设置类提示，它只与顶级窗口有关。Gtk+提供了gtk_window_set_wmclass()函数，程序员可以将它设置为具有确切含义的值。

GdkWindowAttr的最后一个成员 override_redirect决定窗口是否“替换重定向”的。通常，窗口管理器截获所有对顶级窗口的显示、隐藏、移动或尺寸调整请求。你可以重定向或取消这些请求，让顶级窗口按窗口管理器的布局策略行事。将 override_redirect设置为TRUE，禁止窗口管理器对窗口的管理。因为窗口管理器不能移动设置了这个标志的窗口，通常也不会为这样的窗口设置标题条或其他的装饰。注意，所有的 GDK_WINDOW_TEMP窗口都将这个成员设为 TRUE；GDK_WINDOW_TEMP通常用作弹出菜单，窗口管理器不能控制它。

一般不应该改变override_redirect成员。如果指定了正确的 GdkWindowqType，缺省值差不多总是正确的。不过还是有一些例外，例如 Gnome面板应用程序设置了这个成员。

16.3 视件和颜色表

硬件之间总存在差别。最原始的 X服务器只支持两种颜色，每一个像素只能是 on或off(开或关)。这就是“每像素一位”(bpp)显示模式。每像素一位的显示模式称为深度为 1。多数高级的X服务器支持每像素 24或32位，还允许以窗口为基础指定不同的深度。每像素 24位允许2^{24}(16 777 216)种像素，包含了比人眼能分辨的还要多的颜色。

从概念上说，位图显示由一个矩形的像素网格组成。每个像素由一些固定的位数组成；像素以一种硬件相关的方法映射为可视的颜色。考虑这种概念的一种方法就是想象一个二维的整数数组，整数的大小等于要求的位数。换一种说法，可以想象一种显示就像一个“位平面”栈，或“位”的二维数组。如果每个平面都平行于其他平面，那么一个像素就是一根在相同坐标处穿过每个平面的垂线，并且从每一个平面处获得一位。这就是术语“深度”的起源，因为每个像素的位数等于位平面栈的深度。

在X窗口系统中，像素代表在一个颜色查找表中的入口。一种颜色就是一组红、绿、蓝(RGB)值 —— 监视器以一定比率混合红绿蓝光以显示每个像素。例如，考虑一种八位的显示模式：八位不足以为现实中的颜色编码，只可能为很少部分的 RGB值编码。作为替代，数据位被解释为整数，用于为RGB颜色值做索引。这个颜色表称为 colormap(颜色表)，有时，你也可以修改它以包含要使用的颜色，虽然这样做是硬件相关的 (一些colormap是只读的)。

视件（visual）用来决定像素的位模式如何转换为一个可见的颜色。因而，视件还定义了颜色表如何工作。在八位显示中， X服务器也许将每个像素解释为一个包含 256种可能颜色值的颜色表的索引。典型情况下， 24位视件有三个颜色表：一个是红色的浓淡值，一个是绿色的浓淡值，一个是蓝色的浓淡值。每个颜色表用一个八位的值索引；三个八位值组成 24位的像素。视件定义了像素内容的含义，还定义了颜色表是只读的还是可修改的。

简而言之，视件就是特定 X服务器的颜色容量的描述。在 Xlib中，你得围绕视件做很多罗嗦的事，但是Gdk和Gtk+能极大地简化了这些繁琐工作。

16.3.1 GdkVisual

Xlib能报告一个所有可用的视件以及相关信息的列表； Gdk在一个称为GdkVisual的结构中保持一个这些信息的客户端拷贝。 Gdk能报告可用的视件，并将它们以不同的方式分级。在多数时候，只需用 gdk_visual_get_system()函数就可以了，它返回一个指向缺省视件的指针。如果正在写一个GtkWidget构件， gtk_widget_get_visual()函数将返回应该使用的视件。返回的

视件不是一个拷贝，所以不需要释放它， Gdk将永久保存视件。

获取缺省视件

```
#include <gdk/gdk.h>
GdkVisual* gdk_visual_get_system()
```

下面是GdkVisual结构的定义，大多数成员都用于由颜色计算像素值。

```
typedef struct _GdkVisual GdkVisual;
struct _GdkVisual
{
  GdkVisualType type;
  gint depth;
  GdkByteOrder byte_order;
  gint colormap_size;
  gint bits_per_rgb;

  guint32 red_mask;
  gint red_shift;
  gint red_prec;

  guint32 green_mask;
  gint green_shift;
  gint green_prec;

  guint32 blue_mask;
  gint blue_shift;
  gint blue_prec;
};
```

16.3.2 视件的类型

视件用不同的度量方法加以区分。它们可以是灰度级或者 RGB值，颜色表可以是可修改的或者固定的，像素值可以是单个的颜色表或者包含了压缩的红、绿、蓝的索引值。下面是GdkVisualType的可能取值：

- GDK_VISUAL_STATIC_GRAY：意味着显示器是单色的或者灰度的，颜色表不能修改。一个像素值只是一个灰度级别，每个像素都是“ 硬编码 ”，分别代表一个确定的屏幕上的颜色。
- GDK_VISUAL_GRAYSCALE：意味着显示器是可修改的，但是只有灰度的级别才可能修改。像素代表在颜色表中的一个入口，所以给定的像素在不同的时候能够代表不同的灰度级别。
- GDK_VISUAL_STATIC_COLOR：代表一种颜色显示，它使用单个只读的颜色表而不是红、绿、蓝每种颜色各一个单独的颜色表。这种显示差不多就是 12位或更少(使用单个颜色表的24位显示器需要一个带 2^{24}个入口的颜色表，差不多有 500MB)。这是一种恼人的视件，因为它的可用颜色太少，而且不能改变成它们的实际颜色。
- GDK_VISUAL_PSEUDO_COLOR：从很多年前开始，就一直是低端PC硬件的常用视件。如果有一个1MB显存、256色的显示卡，这极有可能就是你的 X服务器的视件。它代表一种具有读/写颜色表的颜色显示。像素只对单个颜色表索引。

- GDK_VISUAL_TRUE_COLOR：是带三个只读颜色表的颜色显示，红、绿、蓝每种颜色有一个颜色表。一个像素包含三个索引，每个颜色表一个。在像素值和 RGB三元组之间有固定的数学关系，可以用下面的公式从 [0，255]间的红、绿、蓝值获取像素值：gulong pixel = (gulong)(red*65536 + green*256 + blue)。
- GDK_VISUAL_DIRECT_COLOR：是一种有三个读写颜色表的颜色显示。如果使用Gdk颜色处理例程，它们只是简单地填充所有三个颜色表以模拟真彩显示。

16.3.3 颜色和GdkColormap

GDK使用GdkColor存储RGB值和像素值。红、绿、蓝值是以 16位无符号整数给出的，取值范围为0到65535。像素的内容依赖于视件。下面是 GdkColor结构定义：

```
typedef struct _GdkColor GdkColor;
struct _GdkColor
{
  gulong  pixel;
  gushort red;
  gushort green;
  gushort blue;
};
```

在用一种颜色绘画时，必须：

- 保证像素值包含合适的值。
- 保证颜色值在要使用的可绘区的颜色表中存在 (可绘区是一个可以在上面绘画的窗口或pixmap)。

在Xlib中，这是一个非常复杂的过程，因为需要对每种不同的视件做不同的工作。 Gdk对这个过程作了大量的简化。你只需调用 gdk_colormap_alloc_color()函数来填充像素值并将颜色值添加到颜色表中即可。下面是一个例子，它使用一个现有的颜色表，这个颜色表应该是要绘画的可绘区的颜色表：

```
GdkColor color;
/* 纯红色*/
color.red   = 65535;
color.green = 0;
color.blue  = 0;

if (gdk_colormap_alloc_color(colormap, &color, FALSE, TRUE))
  {
    /* 成功！*/
  }
```

如果gdk_colormap_alloc_color()函数返回TRUE，然后分配了一个颜色， color.pixel中包含了一个有效的值，这个颜色就可以用于绘画了。 gdk_colormap_alloc_color()函数中的两个布尔型参数指定了这个颜色是否可写，以及当颜色不能分配时是否尽量找到一个“最匹配的”颜色。如果使用了最匹配的颜色而不是分配一个新颜色，颜色的 RGB值会变成最匹配的值。如果为一个不可写的颜色表请求一个最匹配值，颜色分配不应该会失败。因为即使是在黑白显示器中，黑或者白也会是最好的匹配，只有空的颜色表会导致失败。获得空颜色表的唯一方法就是自己创建一个定制的颜色表。如果不要求最好匹配，在有限颜色数的显示器中极有

可能会失败。在可写的颜色表中很容易会发生失败（此时“最匹配”颜色并没有意义，因为该颜色表能被修改)。

“可写”的颜色表是可以在任何时间改变的颜色表，一些视件支持它，还有一些视件不支持。可写颜色表的作用是改变屏幕的颜色而不用重绘图形。一些硬件在它的颜色查找表中将像素作为索引存储，所以改变查找表就改变了像素的显示。可写颜色表的缺点非常多。最特别的几点：不是所有的视件支持它们，并且可写的颜色表不能由其他应用程序所用，而只读的颜色表可以共享，因为其他的应用程序知道颜色会保持不变。因而，应该尽量避免分配可写的颜色值。在较新的硬件中，使用“可写”颜色表的坏处比获得的好处更多；与直接重绘相比，它并不能显著提高绘图的速度。

使用完一种颜色后，应该用gdk_colormap_free_colors()函数将它从颜色表中删除。这只对伪彩色和灰度级的视件很重要，在这两种情况下，颜色不够使用，颜色表能被客户修改。 Gdk会自动对每一种视件类型做适当的事情，所以调用这个函数就可以了。

获得RGB值最方便的方法是 gdk_color_parse()函数。这个函数采用 X颜色规范，填充GdkColor的红、绿、蓝值。X颜色规范可以有多种形式，一种可能形式是 RGB字符串：

```
RGB:FF/FF/FF
```

这指定了一个白色 (红绿蓝全部是全亮度)。“ RGB: ”指定一个“颜色空间”，并决定后面数字的意义。X系统还能理解其他更晦涩的“颜色空间”。如果颜色规范字符串不是以一个可识别的“颜色空间”开头， X系统假定它是一个颜色名，并在一个颜色名称数据库中查找。所以可以写下面这样的代码：

```
GdkColor color;
if (gdk_color_parse("orange", &color))
  {
    if (gdk_colormap_alloc_color(colormap, &color, FALSE, TRUE))
      {
        /* 得到橙色！*/
      }
  }
```

可以看到，如果gdk_color_parse()函数认出了传给它的字符串，它会返回 TRUE。只能使用很少的颜色名，比如“ orange”，但是绝大多数颜色都没有相对应的颜色名，也没有办法知道什么名字在数据库里面能找到，所以应该检查函数的返回值。

函数列表：颜色分配

```
#include <gdk/gdk.h>
gboolean gdk_colormap_alloc_color(GdkColormap* colormap,
                        GdkColor* color,
                        gboolean writeable,
                        gboolean best_match)

void gdk_colormap_free_colors(GdkColormap* colormap,
                        GdkColor* colors,
                        gint ncolors)

gint gdk_color_parse(gchar* spec,
                GdkColor* color)
```

16.3.4 获得颜色表

如果正在写一个 GtkWidget子类，获得颜色表的正确方法就是使用 gtk_widget_get _colormap()函数。另外，系统（缺省）的颜色表通常就是所想要的，调用 gdk_colormap_get _system()函数时不需要参数，它返回缺省的颜色表。

GdkRGB模块是另一种处理颜色的方法，在它的其他功能中，它能用 RGB值设置S图形上下文的背景色和前景色。相关的函数是 gdk_rgb_gc_set_foreground()和gdk_rgb _gc_set _background()。GdkRGB有一个预先分配的颜色表，用来拾取一个最匹配的颜色，使用它意味着应用程序能够与其他使用 GdkRGB的应用程序(比如GIMP)共享有限的颜色表资源。你还能获得GdkRGB的颜色表，可以直接使用它。

16.4 可绘区和pixmap

一个pixmap（像素映射图片）是一个屏幕外的缓冲，可以在里面绘图。当图片存在pixmap中之后，可以将它复制到一个窗口，当窗口可见时，让它显示在屏幕上。当然，还可以直接在窗口中绘制。使用 pixmap作为缓冲可以快速更新屏幕而不需要重复一系列原始的绘图操作。pixmap用来存储从磁盘加载的图像数据，比如图标和徽标。然后可以将图像复制到窗口中。在Gdk中，pixmap类型称为GdkPixmap。每个像素只有一位pixmap称为位图，在Gdk中用GdkBitmap代表位图。位图(Bitmap)并不真是一种单独的类型，从 X系统的观点来说，它只是一个深度为1 的pixmap。像窗口一样， Bitmap是服务器端资源。

在X的术语学中，一个可绘区就是所有可以在上面绘图的东西。在 Gdk中有一个相应的类型，称为GdkDrawable。可绘区包括窗口、pixmap以及位图。下面是在Gdk里面的类型定义：

```
typedef struct _GdkWindow GdkWindow;
typedef struct _GdkWindow GdkPixmap;
typedef struct _GdkWindow GdkBitmap;
typedef struct _GdkWindow GdkDrawable;
```

在客户端， pixmap和位图只是类型为 GDK_WINDOW_PIXMAP 的GdkWindow窗口。GdkDrawable用在可接受窗口或者 pixmap为参数的函数声明中。绘制图形的函数使用这两种类型作为参数，移动或者设置“ 窗口管理器提示 ”的函数只接受窗口为参数。只有窗口能够接收事件。GDK_INPUT_ONLY窗口是一种特殊情况，它们不是可绘区，不能在上面绘画。

实际上，上面四种可绘区有三种逻辑组合：

	可绘制的	不可绘制的
具有窗口特性	普通窗口	只能输入的窗口
无窗口特性	Pixmap/Bitmap	---

不幸的是，所有这三种逻辑上截然不同的情形从类型检查的观点来看都是一样的。所以要小心使用，不要弄错了。还有，要记住普通窗口在实际显示在屏幕上之前并不是可绘区，应该等到接收到一个“ expose ”事件后才能开始绘画。

像GdkWindow一样，一个GdkPixmap仅仅是位于X服务器上的对象的客户端句柄。因为这一点，某些操作从性能的观点来看是完全不可行的，例如，如果正在做任何要求对单个像素进行大量操作的事情，可绘区反应会非常慢。另一方面，复制一个 pixmap到窗口中并没有想象的慢，因为两个对象都是在同一台机器上的。

创建一个pixmap比创建一个窗口要简单得多，因为大多数窗口属性与 pixmap是不相关的。用gdk_pixmap_new()函数创建pixmap。它以初始尺寸和位深度为参数。如果深度值是 -1，从它的GdkWindow参数中复制深度。不能为深度随意指定一个值 —— 服务器不支持所有的深度，并且pixmap的深度必须与要复制到的窗口的深度一致。调用 gdk_pixmap_unref()函数可以销毁一个pixmap。

传递到gdk_pixmap_new()函数中的GdkWindow参数不是严格需要的。不过，该函数中封装了一个XCreatePixmap()函数，它以一个X窗口作为参数。它用这个参数决定在哪个屏幕上创建窗口，一些X服务器有多个显示器。屏幕是一个完全由 Gdk隐藏起来的Xlib概念，Gdk一次只支持一个屏幕。这样，从 Gdk的观点来看，gdk_pixmap_new()函数中的window参数很神秘。

函数列表：GdkPixmap构建函数

```
#include <gdk/gdk.h>
GdkPixmap*
gdk_pixmap_new(GdkWindow* window,
               gint width,
               gint height,
               gint depth)
void gdk_pixmap_unref(GdkPixmap* pixmap)
```

16.5 事件

事件传送到应用程序中，以指明在一个 GdkWindow中的变化或者有意义的用户动作。所有的事件都与一个 GdkWindow相关联。它们也与一个 GtkWidget相关联，Gtk+主循环将事件从Gdk传递给Gtk+构件树。

16.5.1 事件类型

事件GDK中有多种类型，可以使用 GdkEvent联合体代表任何类型。有一个特殊的事件类型GdkEventAny，它包含了所有事件都有的三个成员，任何事件都可以转换为 GdkEventAny。GdkEventAny中的第一个成员是一个类型标志， GdkEventType。GdkEvent联合体中也包含GdkEventType。下面是GdkEventAny结构定义：

```
struct _GdkEventAny
{
  GdkEventType type;
  GdkWindo-w *window;
  gint8 send_event;
};
```

下面是GdkEvent 联合体的定义：

```
union _GdkEvent
{
  GdkEventType              type;
  GdkEventAny               any;
  GdkEventAny               any;
  GdkEventExpose            expose;
  GdkEventNoExpose          no_expose;
```

```
    GdkEventVisibility        visibility;
    GdkEventMotion            motion;
    GdkEventButton            button;
    GdkEventKey               key;
    GdkEventCrossing          crossing;
    GdkEventFocus             focus_change;
    GdkEventConfigure         configure;
    GdkEventProperty          property;
    GdkEventSelection         selection;
    GdkEventProximity         proximity;
    GdkEventClient            client;
    GdkEventDND               dnd;
};
```

每个事件类型都以GdkEventAny中的三个成员作为它的头三个成员。这样，有多种方式引用一个事件的类型（假设GdkEvent*称为一个事件）：

```
event->type
event->any.type
event->button.type
((GdkEventAny*)event)->type
((GdkEventButton*)event)->type
```

在Gtk+的源代码中有可能看到上面的各种方法。当然，每个事件子类型都有它自己唯一的成员，type成员指出什么子类型是有效的。

GdkEventAny的window成员是事件要传送到的GdkWindow窗口。如果send_event标志是TRUE，事件就会由另一个应用程序（或本应用程序）合成；如果是 FALSE，事件由X服务器引发。Gdk并不传送事件的X接口（XSendEvent()）。不过，Gtk+经常通过声明一个静态的event结构来“虚构”一个事件，在结构中填充相应的值，然后引发与事件相对应的构件信号。这些合成的事件中send_event设置为TRUE。

对GdkEventType，有比GdkEvent 联合体中更多的可能值。许多事件类型共用同样的数据。例如，因为当鼠标按键按下和弹起时，会传递同样的信息，GDK_BUTTON_PRESS和GDK_BUTTON_RELEASE都使用GdkEvent的button成员。表16-1显示了GdkEventType枚举类型和相应的GdkEvent成员中所有的可能取值。每个事件的类型在本节稍后介绍。

表16-1 GdkEventType值

值	GdkEvent成员
GDK_NOTHING	None
GDK_DELETE	GdkEventAny
GDK_DESTROY	GdkEventAny
GDK_EXPOSE	GdkEventExpose
GDK_MOTION_NOTIFY	GdkEventMotion
GDK_BUTTON_PRESS	GdkEventButton
GDK_2BUTTON_PRESS	GdkEventButton
GDK_3BUTTON_PRESS	GdkEventButton
GDK_BUTTON_RELEASE	GdkEventButton
GDK_KEY_PRESS	GdkEventKey
GDK_KEY_RELEASE	GdkEventKey
GDK_ENTER_NOTIFY	GdkEventCrossing
GDK_LEAVE_NOTIFY	GdkEventCrossing

(续)

值	GdkEvent成员
GDK_FOCUS_CHANGE	GdkEventFocus
GDK_CONFIGURE	GdkEventConfigure
GDK_MAP	GdkEventAny
GDK_UNMAP	GdkEventAny
GDK_PROPERTY_NOTIFY	GdkEventProperty
GDK_SELECTION_CLEAR	GdkEventSelection
GDK_SELECTION_REQUEST	GdkEventSelection
GDK_SELECTION_NOTIFY	GdkEventSelection
GDK_PROXIMITY_IN	GdkEventProximity
GDK_PROXIMITY_OUT	GdkEventProximity
GDK_DRAG_ENTER	GdkEventDND
GDK_DRAG_LEAVE	GdkEventDND
GDK_DRAG_MOTION	GdkEventDND
GDK_DRAG_STATUS	GdkEventDND
GDK_DROP_START	GdkEventDND
GDK_DROP_FINISHED	GdkEventDND
GDK_CLIENT_EVENT	GdkEventClient
GDK_VISIBILITY_NOTIFY	GdkEventVisibility
GDK_NO_EXPOSE	GdkEventNoExpose

16.5.2 事件屏蔽

每个GdkWindow窗口都有一个与之相关联的事件屏蔽，它决定 X服务器将哪一个事件传递到应用程序。你可以在创建 GdkWindow窗口时将它作为GdkWindowAttr结构的一部分指定事件屏蔽。以后还可以用 gdk_window_set_events()和gdk_window_get_events()函数存取或改变事件屏蔽。如果要设置的 GdkWindow属于一个构件，则不应该直接改变事件屏蔽，应该使用gtk_widget_set_events()或gtk_widget_add_events()函数。gtk_widget_set_events() 函数在构件已经实现后才能使用，它可以在任何时候用 gtk_widget_add_events()函数向一个已存在的事件屏蔽添加事件。

函数列表：GdkWindow事件屏蔽

```
#include <gdk/gdk.h>
GdkEventMask gdk_window_get_events(GdkWindow* window)
void gdk_window_set_events(GdkWindow* window,
                           GdkEventMask event_mask)
```

函数列表：构件的事件屏蔽

```
#include <gtk/gtkwidget.h>
gint gdk_widget_get_events(GtkWidget* widget)
void gtk_widget_add_events(GtkWidget* widget,
                           gint event_mask)
void gtk_widget_set_events(GtkWidget* widget,
                           gint event_mask)
```

表16-2显示了什么事件要求什么样的事件屏蔽值。有些事件不用指定就会接收到，特别是：

• Map，unmap，destroy和configure事件是用GDK_STRUCTURE_MASK指定的，但是对

任何新窗口，Gdk都会自动选中它们。不过，Xlib并不这样做。

- 选择、拖放以及删除事件没有事件屏蔽，因为它们会被自动选中 (对所有的窗口，Xlib都选中它们)。

表16-2 事件与屏蔽类型

事 件 屏 蔽	屏蔽事件类型
GDK_EXPOSURE_MASK	GDK_EXPOSE
GDK_POINTER_MOTION_MASK	GDK_MOTION_NOTIFY
GDK_POINTER_MOTION_HINT_MASK	N/A
GDK_BUTTON_MOTION_MASK	GDK_MOTION_NOTIFY(鼠标键按下时)
GDK_BUTTON1_MOTION_MASK	GDK_MOTION_NOTIFY(鼠标左键按下时)
GDK_BUTTON2_MOTION_MASK	GDK_MOTION_NOTIFY(鼠标右键按下时)
GDK_BUTTON3_MOTION_MASK	GDK_MOTION_NOTIFY(鼠标中间键按下时)
GDK_BUTTON_PRESS_MASK	GDK_BUTTON_PRESS
GDK_BUTTON_RELEASE_MASK	GDK_BUTTON_RELEASE
GDK_KEY_PRESS_MASK	GDK_KEY_PRESS
GDK_KEY_RELEASE_MASK	GDK_KEY_RELEASE
GDK_ENTER_NOTIFY_MASK	GDK_ENTER_NOTIFY
GDK_LEAVE_NOTIFY_MASK	GDK_LEAVE_NOTIFY
GDK_FOCUS_CHANGE_MASK	GDK_FOCUS_IN, GDK_FOCUS_OUT
GDK_STRUCTURE_MASK	GDK_CONFIGURE
GDK_PROPERTY_CHANGE_MASK	GDK_PROPERTY_NOTIFY
GDK_VISIBILITY_NOTIFY_MASK	GDK_VISIBILITY_NOTIFY
GDK_PROXIMITY_IN_MASK	GDK_PROXIMITY_IN
GDK_PROXIMITY_OUT_MASK	GDK_PROXIMITY_OUT
GDK_SUBSTRUCTURE_MASK	对子窗口接收到GDK_STRUCTURE_MASK
GDK_ALL_EVENTS_MASK	所有事件

16.5.3 在Gtk+中接收Gdk事件

在一个Gtk+程序里面不会直接接收Gdk事件。相反，所有事件都是传递到一个 GtkWidget构件，由它引发一个相应的信号。通过连接一个信号处理函数到信号上来处理事件。

X服务器给每个X客户发送一个事件流。事件按它们发生的次序被发送和接收。 Gdk将它接收到的每一个XEvent转换为一个GdkEvent事件，然后将事件放在一个队列里面。对每个被接收的事件来说，它决定哪一个构件 (如果有的话)接收事件。GtkWidget基类对大多数事件类型(比如说button_press_event)都定义了信号；它也定义了一个普通的“事件”信号。 Gtk+主循环调用gtk_widget_event()函数将事件传递给构件，这个函数首先引发一个“事件”信号，然后只对特定事件类型 (如果合适)引发一个信号。一些事件是用特殊的方法处理的；例如，drag-and-drop(拖放)事件并不是直接对应着drag-and-drop信号。

通常，构件的事件属于事件所发生的GdkWindow。不过，还有下面几种特殊情况：

- 如果构件具有独占性(也就是说，如果调用了gtk_grab_add()函数)，某些事件只会转发给具有独占性的构件，或者这个构件的子构件。发生在其他构件上的事件会被忽略。只有由用户引发的事件，比如鼠标按键事件和键盘事件才会受独占影响。
- 构件的敏感性也会影响事件传递的位置。用户交互事件不会传递到不敏感的构件上。

可以预见，没有相关联的 GdkWindow窗口的构件（比如 GtkLabel）不会产生事件； X只

给窗口传递事件。有一个例外：容器为它们的无窗口子构件合成 expose事件。

Gtk+主循环将事件从子构件传给它们的父容器。也就是，对每个事件，信号先由它的子构件引发，然后是它紧挨着的父构件，然后是父构件的父构件，等等。依次向上递归。例如，如果点击一个GtkMenuItem，它会忽略按钮按下事件，让它的上层菜单处理它。有一些事件是不会传播的。

如果一个构件“处理”了事件，事件传播就会结束。这保证了对任何用户动作，只会发生一个用户可见的变化。构件的事件信号处理程序必须返回一个 gint型整数值。最后一个要运行的信号处理程序决定了信号引发的返回值。所有的事件信号都是 GTK_RUN_LAST类型的，所以返回值来自于下面几种：

- 如果有的话，用gtk_signal_connect_after()连接的最后一个处理程序。
- 否则，构件的缺省信号处理程序，如果有的话。
- 否则，用gtk_signal_connect()连接的信号处理程序(如果有)。
- 否则，缺省返回值是FALSE。

如果一个信号引发返回值是 TRUE，Gtk+主循环会停止当前事件的传播。如果它返回FALSE，主循环会将事件传播给它的父构件。每个构件都会导致两种信号引发：一种普通的“事件”信号和一种特殊的信号 (比如button_press_event或key_press_event)。 如果两种引发都返回TRUE，事件传播将停止。普通“事件”信号的返回值还有例外的效果：如果是 TRUE，第二个信号（特殊信号）不会引发。

表16-3概括了GtkWidget信号与事件类型的对应关系。该事件受活动的“鼠标独占”的影响，并沿父构件到子构件的路径传播。所有事件信号处理程序都返回一个 gint类型的整数值，带有三个参数：引发信号的构件、触发信号的事件、用户数据指针。

表16-3 GtkWidget事件

事件类型	GtkWidget信号	是否传播	是否独占
GDK_DELETE	"delete_event"	否	否
GDK_DESTROY	"destroy_event"	否	否
GDK_EXPOSE	"expose_event"	否	否
GDK_MOTION_NOTIFY	"motion_notify_event"	是	是
GDK_BUTTON_PRESS	"button_press_event"	是	是
GDK_2BUTTON_PRESS	"button_press_event"	是	是
GDK_3BUTTON_PRESS	"button_press_event"	是	是
GDK_BUTTON_RELEASE	"button_release_event"	是	是
GDK_KEY_PRESS	"key_press_event"	是	是
GDK_KEY_RELEASE	"key_release_event"	是	是
GDK_ENTER_NOTIFY	"enter_notify_event"	否	是
GDK_LEAVE_NOTIFY	"leave_notify_event"	否	是*
GDK_FOCUS_CHANGE	"focus_in_event","focus_out_event"	否	否
GDK_CONFIGURE	"configure_event"	否	否
GDK_MAP	"map_event"	否	否
GDK_UNMAP	"unmap_event"	否	否
GDK_PROPERTY_NOTIFY	"property_notify_event"	否	否
GDK_SELECTION_CLEAR	"selection_clear_event"	否	否
GDK_SELECTION_REQUEST	"selection_request_event"	否	否
GDK_SELECTION_NOTIFY	"selection_notify_event"	否	否

(续)

事件类型	GtkWidget信号	是否传播	是否独占
GDK_PROXIMITY_IN	"proximity_in_event"	是	是
GDK_PROXIMITY_OUT	"proximity_out_event"	是	是
GDK_CLIENT_EVENT	"client_event"	否	否
GDK_VISIBILITY_NOTIFY	"visibility_notify_event"	否	否
GDK_NO_EXPOSE	"no_expose_event"	否	否

* 如果正被“独占”的构件接收到了相应的GDK_ENTER_NOTIFY事件，GDK_LEAVE_NOTIFY事件会发生在该“独占”的构件。这保证了“进入”事件（GDK_ENTER_NOTIFY）和“离开”事件（GDK_LEAVE_NOTIFY）是成对发生的。

16.5.4 鼠标按键事件

在GdkEventButton中有四种事件类型：

- GDK_BUTTON_PRESS 表明一个鼠标按键被按下。
- GDK_BUTTON_RELEASE 表明一个鼠标按键按下后按键被释放了。不一定是按下鼠标的地方发生这个事件，如果用户将鼠标移动到一个不同的 GdkWindow窗口，新的窗口会接收这个事件，而不是按下鼠标的窗口。
- GDK_2BUTTON_PRESS 表明鼠标按键在一个较短的时间内连续点击了两次：双击事件。该事件总是在GDK_BUTTON_PRESS和GDK_BUTTON_RELEASE事件的第一次点击之后发生。
- GDK_3BUTTON_PRESS表明鼠标键在很短时间间隔内连续点击了三次：“三击”事件。该事件总是在两个 GDK_BUTTON_PRESS/GDK_BUTTON_RELEASE事件对和GDK_2BUTTON_PRESS之后发生。

如果在同一个GdkWindow窗口上快速点击了三次鼠标，下面的事件会按次序接收到：

- GDK_BUTTON_PRESS
- GDK_BUTTON_RELEASE
- GDK_BUTTON_PRESS
- GDK_2BUTTON_PRESS
- GDK_BUTTON_RELEASE
- GDK_BUTTON_PRESS
- GDK_3BUTTON_PRESS
- GDK_BUTTON_RELEASE

当鼠标按键被按下时，X服务器自动发生指针独占，释放按键时，解除指针独占。这意味着按键释放事件总是发生在接收“按键按下”事件的同一个窗口上。 Xlib允许改变这种行为，但是Gdk不允许(在Xlib文档中，这种自动独占称为“被动独占”。这与用gdk_pointer_grab()函数主动独占窗口是截然不同的)。

按键事件是像下面这样定义的：

```
typedef struct _GdkEventButton GdkEventButton;
struct _GdkEventButton
{
```

```
  GdkEventType type;
  GdkWindow *window;
  gint8 send_event;
  guint32 time;
  gdouble x;
  gdouble y;
  gdouble pressure;
  gdouble xtilt;
  gdouble ytilt;
  guint state;
  guint button;
  GdkInputSource source;
  guint32 deviceid;
  gdouble x_root, y_root;
};
```

按钮事件在X服务器中是用一个时间标记来标志的。时间是按服务器端时间、以毫秒度量的，每隔几周，整数就会溢出，然后时间标记就会从 0重新开始。因而，不应该依赖于绝对的时间值；它一般用于判定事件之间的相对时间。

GdkEventButton结构中包含的x、y是鼠标指针相对于事件发生的窗口的 x和y坐标。记住，鼠标指针可能会在窗口的外边 (如果指针独占有效)。如果鼠标在窗口的外边，坐标值可能是负值或比窗口的尺寸还大。坐标是以双精度值给出的，而不是整数，因为一些输入设备，例如图形输入板有比像素还小的度量单位。对大多数情况，可能要将双精度值转换为整数值。一些输入设备还有 pressure（压力）、xtilt(x方向斜移)以及ytilt(y方向斜移)等特征，大多数情况下都可以忽略它们。

GdkEventButton的state成员指示在按钮按下之前哪个组合键或鼠标按键是按下的。它是一个位域，设置为下表中的一个或多个标志值。因为组合键值是在按键按下之前读出的，它只会使用在“ 按键按下 ”之前的 state的值，但是“ 按键释放 ”事件不是这样的。

应该细心检查确切的位屏蔽值，而不是检查 state域的精确值。也就是说，尽量使用这样的代码：

```
if ( (state & GDK_SHIFT_MASK) = = GDK_SHIFT_MASK )
```

避免使用这样的代码：

```
if ( state == GDK_SHIFT_MASK )
```

如果检查state成员的精确值，如果应用程序的 Num Locks 键或其他较较难注意到的按键是打开的，应用程序有可能不会按所设想的方式工作，并且也很难检查出问题所在。

表16-4 鼠标按键和键盘按键的修改键屏蔽值

事 件	修改键含义
GDK_SHIFT_MASK	Shift键
GDK_LOCK_MASK	CapsLock键
GDK_CONTROL_MASK	Ctrl键
GDK_MOD1_MASK	Mod1(通常是Meta或Alt键)
GDK_MOD2_MASK	Mod2键
GDK_MOD3_MASK	Mod3键
GDK_MOD4_MASK	Mod4键

(续)

事　件	修改键含义
GDK_MOD5_MASK	Mod5键
GDK_BUTTON1_MASK	鼠标按键1
GDK_BUTTON2_MASK	鼠标按键2
GDK_BUTTON3_MASK	鼠标按键3
GDK_BUTTON4_MASK	鼠标按键4
GDK_BUTTON5_MASK	鼠标按键5
GDK_RELEASE_MASK	键盘按键释放

GdkEventButton 的button成员指明是哪个按键触发了事件 (也就是，哪个按键被按下或释放)。按键是从1到5编号的，大多数场合，按键 1是左键，按键 2是中间键，按键 3是右键。左撇子也许会将按键颠倒过来。按键 4和按键5事件是由滚轮鼠标在旋转滚轮时生成的， Gtk+会捕获这些事件并移动旁边的滚动条。有的鼠标没有中间键，也没有滚轮，因而，最好不要写一些依赖于按键2、4或5引发的事件的程序。

三个标准的鼠标键在 Gnome中有约定的含义。按键 1用于选择、拖放以及操作构件等最常见的任务。按键3用于激活一个弹出菜单。按键 2通常用于移动对象，比如面板。有时按键 1也用于移动对象，例如桌面图标可以用按键 1或按键3移动。应该尽量与其他应用程序保持一致。

GdkEventButton 的source和deviceid成员用于决定是哪一个设备触发了事件，例如，用户也许同时连接了一块图形输入板和一个鼠标。除非要写一个能利用非鼠标输入设备的应用程序，可以忽略这两个成员。

GdkEventButton 的最后两个成员， x_root和y_root，是相对于根窗口的x和y的坐标，而不是相对于接收事件的窗口的坐标。可以用这些“ 绝对 ”坐标来比较来自于两个不同窗口的事件。

16.5.5 键盘事件

只有两种类型的键盘事件： GDK_KEY_PRESS和GDK_KEY_RELEASE。一些硬件不会产生“ 放开按键 ”事件；不应该写依赖于 GDK_KEY_RELEASE事件的代码，虽然如果有构件接收到该事件，代码也会正常反应。

下面是键盘事件的类型定义：

```
typedef struct _GdkEventKey GdkEventKey;
struct _GdkEventKey
{
  GdkEventType type;
  GdkWindow *window;
  gint8 send_event;
  guint32 time;
  guint state;
  guint keyval;
  gint length;
  gchar *string;
};
```

前三个成员是GdkEventAny事件中的标准成员， time和state成员与GdkEventButton中的是一样的。

第六个成员keyval包含一个键符。X服务器保留了一个全局的翻译表，它将实际按键和控制键的组合转换成键符。例如，键盘上的“ A ”键，产生不带控制键的 GDK_a键符，和按下shift键的GDK_A 键符。用户可以改变物理按键和键符之间的映射关系，例如，可以重新排列按键，创建一个Dvorak键盘(更普通的情况：可以将 Ctrl和Caps Lock键交换，或将 Alt键用作Meta键)。键符是在gdk/gdkkeysyms.h头文件中定义的。如果要使用 keyval成员，应该包含这个文件。键符是用字符串来表示的。例如， GDK_a键符映射为字符串“ a ”。不过，X服务器允许修改键符到字符串的映射。

GdkEventKey的string成员包含一个键符的字符串描述， length成员包含字符串的长度。length值可能是0 (缺省状态，许多非字母的按键并没有字符串描述)。

通常，如果为了将用户打的字显示出来而读取键盘事件，应该使用 GdkEventKey的string成员。例如， GtkEntry和GtkText构件都使用了string成员。字处理程序也应该使用这个成员。如果因为其他原因而读取键盘事件 (比如说键盘快捷键)，或者对缺省没有字符串描述的按键 (比如说功能键或方向键)感兴趣，需要使用在 gdk/gdkkeysyms.h头文件里面定义的 keyval和keysym。

下面是一个简单的按键事件回调函数示例，它演示了怎样从按键事件中获取信息。对任何构件，将这个回调函数连接到key_press_event信号都是很合适的：

```
static gint
key_press_cb(GtkWidget* widget, GdkEventKey* event, gpointer data)
{
  if (event->length > 0)
    printf("The key event's string is `%s'\n", event->string);
  printf("The name of this keysym is `%s'\n",
         gdk_keyval_name(event->keyval));
  switch (event->keyval)
    {
    case GDK_Home:
      printf("The Home key was pressed.\n");
      break;
    case GDK_Up:
      printf("The Up arrow key was pressed.\n");
      break;
    default:
      break;
    }

  if (gdk_keyval_is_lower(event->keyval))
    {
      printf("A non-uppercase key was pressed.\n");
    }
  else if (gdk_keyval_is_upper(event->keyval))
    {
      printf("An uppercase letter was pressed.\n");
    }
}
```

gdk_keyval_name()函数对调试很有用，它返回不带 GDK_前缀的键符。例如，如果向它

传递的是GDK_Home，则返回“ Home ”。字符串是静态分配内存的。如果键符是大写的，gdk_keyval_is_lower()函数将返回FALSE。对小写字母、数字以及所有的非字母的字符，它返回TRUE；它只对大写字母返回 FALSE。gdk_keyval_is_upper()与gdk_keyval_is_lower()刚好相反。

16.5.6 鼠标移动事件

当鼠标在屏幕上移动时，可以使用鼠标移动事件跟踪它的移动。移动事件是当鼠标指针在窗口内移动时发生的，穿越事件是在鼠标指针进入或离开 GdkWindow窗口时发生的。移动事件中的典型成员是GDK_MOTION_NOTIFY。有两种类型的穿越事件：GDK_ENTER_NOTIFY和GDK_LEAVE_NOTIFY。

有两种方法跟踪鼠标移动事件。如果在窗口的事件屏蔽中指定了 GDK_POINTER _MOTION_MASK，可以接收到X服务器能产生的尽可能多的事件。如果用户快速移动指针，程序会被移动事件淹没，必须快速处理它们，否则应用程序在处理大量事件时会反应迟钝。如果还指定了GDK_POINTER_MOTION_HINT_MASK，那么每次只发送一个移动事件。也要调用gdk_window_get_pointer()函数、鼠标指针离开并重新进入窗口，或者鼠标按键或键盘事件发生时，事件才会发送出去。也就是，每次接收到一个移动事件必须调用gdk_window_get_pointer()函数取得鼠标指针的当前位置，并通知 X服务器已经可以接收另一个事件了。

选择何种模式依赖于应用程序。如果需要精确跟踪指针的轨迹，就得捕获所有的移动事件。如果仅仅关心最近的鼠标指针位置，为了将网络流量最小化，使应用程序的响应能力最大化，则应在应用程序的事件屏蔽中包含 GDK_POINTER_MOTION_HINT_MASK。gdk_window_get_pointer()函数要求在服务器和客户之间传输数据以获得鼠标指针的位置，所以它对应用程序的响应能力有很大的限制。如果能尽可能快地处理鼠标运动事件而不被大量的事件淹没，应用程序看起来会比没有设置事件屏蔽 GDK_POINTER _MOTION_HINT_MASK 要快些。鼠标移动事件不可能在一秒钟内发生 200次 ，所以如果能在5毫秒内处理好鼠标移动事件，情况就会不错。

如果想在一个或多个鼠标按键按下时才接收鼠标事件，可以用 GDK_BUTTON_MOTION _MASK代替 GDK_POINTER _MOTION_MASK。还可以用 GDK_POINTER_MOTION _HINT_MASK和GDK_BUTTON_MOTION_MASK一起使用来限制要接收的事件的数量，就像和GDK_POINTER_MOTION_MASK一起使用一样。如果仅仅对某个特定的鼠标按键按下时的鼠标移动事件感兴趣，可以使用与特定鼠标键相关的事件屏蔽 GDK_BUTTON1_MOTION _MASK、GDK_BUTTON2_MOTION_MASK以及 GDK_BUTTON3_MOTION_MASK。这三个事件屏蔽的任何组合都是允许的。它们也可以与GDK_POINTER_MOTION_HINT_MASK联合使用以限制事件发生的数量。

总而言之，可以用下面五个事件屏蔽值选择在什么按键状态下接收什么鼠标移动事件：

- GDK_POINTER_MOTION_MASK：不管按键状态，接收所有事件。
- GDK_BUTTON_MOTION_MASK：有按键按下时接收所有事件。
- GDK_BUTTON1_MOTION_MASK：按键1按下时接收所有事件。
- GDK_BUTTON2_MOTION_MASK: 按键2按下时接收所有事件。

• GDK_BUTTON3_MOTION_MASK: 按键3按下时接收所有事件。

缺省状态下，应用程序会被X服务器能生成的事件所淹没， 最好在事件屏蔽中添加一个GDK_POINTER_MOTION_HINT_MASK，让事件“一段时间内发生一次”。

移动事件用GdkEventMotion结构描述：

```
typedef struct _GdkEventMotion GdkEventMotion;
struct _GdkEventMotion
{
  GdkEventType type;
  GdkWindow *window;
  gint8 send_event;
  guint32 time;
  gdouble x;
  gdouble y;
  gdouble pressure;
  gdouble xtilt;
  gdouble ytilt;
  guint state;
  gint16 is_hint;
  GdkInputSource source;
  guint32 deviceid;
  gdouble x_root, y_root;
};
```

大多数成员与GdkEventButton中的成员是类似的。事实上，唯一与 GdkEventMotion不同的成员是is_hint标志。如果它是TRUE，GDK_POINTER_MOTION_HINT_MASK标志会选中。如果要写一个供其他人使用的构件，并且想让他们选择怎样接收移动事件，需要设置这个成员的值。在移动事件处理程序中，可以这么做：

```
  double x, y;
  x = event->motion.x;
  y = event->motion.y;
  if (event->motion.is_hint)
    gdk_window_get_pointer(event->window, &x, &y, NULL);
```

也就是说，只在必要时调用gdk_window_get_pointer()函数。

如果正在使用GDK_POINTER_MOTION_HINT_MASK，给定事件的坐标应该使用从gdk_window_get_pointer()函数取得的值，因为它们是更新过的。如果正在接收每个事件，调用gdk_window_get_pointer()就没有什么意义，因为这样做太慢了，并且会引起严重事件积压——这样，最终会得到所有的事件，但是应用程序的性能会很糟糕。

穿越事件是在鼠标指针进入或离开一个窗口时发生的。如果在应用程序的 GdkWindow之间快速移动， Gdk会为每一个被穿越的窗口产生穿越事件。不过， Gtk+会试图删除在多个“穿越”过程中的事件，只将第一个离开窗口事件和最后一个进入窗口事件传给构件。这种优化能够提高程序的响应速度。如果感觉到应该发生了进入 /离开事件，可实际上并没有发生，可能是上面的优化造成的。

GdkEventCrossing结构的定义如下：

```
typedef struct _GdkEventCrossing GdkEventCrossing;
struct _GdkEventCrossing
```

```
{
  GdkEventType type;
  GdkWindow *window;
  gint8 send_event;
  GdkWindow *subwindow;
  guint32 time;
  gdouble x;
  gdouble y;
  gdouble x_root;
  gdouble y_root;
  GdkCrossingMode mode;
  GdkNotifyType detail;
  gboolean focus;
  guint state;
};
```

上面所示结构中的很多成员都应该是很熟悉的，其中 x和y坐标是相对于发生穿越事件的窗口的坐标；x_root和y_root坐标是相对于根窗口的；time成员指明事件的时间；state成员指明发生事件时按下的鼠标按键和组合键。结构中的前三个成员是 GdkEventAny 中的三个标准成员。不过，这里还有几个新成员。

GdkEventCrossing结构中的window成员是一个指向鼠标指针要进入或离开的窗口指针， x和y坐标就是相对于这个窗口的。但是，在“离开”事件发生之前，鼠标指针也许已经在接收事件的窗口的子窗口中存在了；当“进入”事件发生时，鼠标指针也许在一个子窗口中消失了。在这些情况下， subwindow成员应该设置为这个子窗口。否则，可以将 subwindow设置为NULL。注意，如果在子窗口的事件屏蔽值中设置有 GDK_ENTER_NOTIFY_MASK或GDK_LEAVE_NOTIFY_MASK，子窗口也会接收到它自己的“进入”或者“离开”事件。

GdkEventCrossing 中的mode成员指出事件是正常引发的还是作为指针独占的一部分。当指针被独占或解除独占时，指针也许会移动。由一个独占引起的鼠标移动事件的 mode成员的值为GDK_CROSSING_GRAB， 由解除独占引起的的移动事件的mode成员为GDK_CROSSING_UNGRAB，所有其他情况mode都为GDK_CROSSING_NORMAL。

GdkEventCrossing 中的detail成员很少用到。它给出了关于要离开和进入的窗口在 X系统窗口树中的相对位置的一些信息。它有两个很简单、很有用的值：

- GDK_NOTIFY_INFERIOR 标志一个当指针移进或移出一个子窗口时，由父窗口接收到的穿越事件。
- GDK_NOTIFY_ANCESTOR 标志一个当指针移进或者移出一个父窗口时，由子窗口接收到的穿越事件。

其他几种值也是可能的： GDK_NOTIFY_VIRTUAL，GDK_NOTIFY_INFERIOR，GDK_NOTIFY_NONLINEAR，GDK_NOTIFY_NONLINEAR_VIRTUAL以及GDK_NOTIFY_UNKNOWN。不过，它们绝不会用到，因为它们太复杂了。

键盘焦点

GdkEventCrossing的focus成员指出是事件窗口还是它的子窗口得到键盘输入焦点。键盘焦点是一个X概念，用于决定哪一个窗口应该接收按键事件。窗口管理器决定哪一个顶级窗口应该获得焦点。通常，获得焦点的窗口是高亮显示的，并在最前面显示。大多数窗口管理器会让你在“跟随鼠标获得焦点”和“点击获得焦点”间作出选择。当应用程序具有焦点时，

可以将焦点在它的子窗口间自由移动（例如，在不同的 GtkText构件间移动）。不过， Gtk+并不对子窗口使用X的焦点机制。顶级GtkWindow构件是唯一接收X焦点的窗口。 因而，它们从X服务器(通过Gdk)接收所有的原始按键事件。 Gtk+实现了它自己的构件焦点概念，它和 X的窗口焦点概念是类似的，但实际上是截然不同的。当一个顶级窗口接收到按键事件，它会将事件传给具有Gtk+焦点的构件。

简而言之，如果包含事件窗口的顶级 GtkWindow窗口目前具有 X焦点， focus成员的值就会是TRUE。Focus成员与Gtk+的构件焦点概念没有直接的关系。

16.5.7 焦点变更事件

前面直接解释了 Gtk+和X/Gdk的键盘焦点概念的区别。只有一种类型的焦点事件，GDK_FOCUS_CHANGE，当一个窗口或获得或失去键盘焦点，就会发生这个事件。从 Gdk的观点来看，只有顶级 GtkWindow构件才能获得或失去焦点，这样这个事件好象没什么用。不过，每个GtkWindow窗口构件维护一个当前“ 焦点构件 ”，并将键盘事件传递给这个构件。当焦点构件改变时，它也合成一个 Gdk风格的焦点事件。这样，即使没有用到 Gdk风格的焦点事件，构件以用到这种事件相同的方法接收到焦点。有一点细微的区别：不论构件的GdkWindow的事件屏蔽值是否设置包含了GDK_FOCUS_CHANGE_MASK ，它都会接收到焦点变更事件。只有顶级构件需要指定这个屏蔽值。

下面是焦点事件GdkEventFocus结构的定义。焦点事件本身很简单。当一个构件获得键盘焦点，它接收到一个焦点事件，同时它的 in成员设置为TRUE，当它失去焦点时，它也接收到这个焦点事件，它的in成员设置为FALSE。除此之外，焦点事件只包含 GdkEventAny 中的三个标准成员：

```
typedef struct _GdkEventFocus GdkEventFocus;
struct _GdkEventFocus
{
  GdkEventType type;
  GdkWindow *window;
  gint8 send_event;
  gint16 in;
};
```

16.6 鼠标指针

鼠标在屏幕上用一个称为“ 光标 ”的位图显示。普通情况下光标是一个箭头形状，但是它可以以窗口为基础改变。鼠标指针移动时，它产生移动事件，并在屏幕上移动光标，向用户反馈信息。

16.6.1 指针定位

可以用gdk_window_get_pointer()函数查询指针的位置。这个函数以指针的 x和y坐标作为参数；坐标是相对与第一个参数“ window ”的。它还获得当前的活动修改键 (包括修改键和按钮；这个成员与几个其他事件，比如按钮事件中 state成员是一样的)。如果给x，y或state参数传递NULL值，这个参数会被忽略。

函数列表：查询指针位置

```
#include <gdk/gdk.h>
GdkWindow* gdk_window_get_pointer(GdkWindow* window,
                    gint* x,
                    gint* y,
                    GdkModifierMask* state)
```

16.6.2 独占指针

可以独占指针，这意味着在独占期间，所有的鼠标指针事件都会发生在独占的窗口上。通常指针事件都发生在指针所在的窗口上。例如当用户正在用“点击－拖动”来选择一个矩形的区域时，应该独占指针。如果点击鼠标后将指针拖到窗口外边，应该继续跟踪鼠标的位置，并据此改变选择的范围。独占也保证了指针事件不会发送到其他的应用程序中。

调用gdk_pointer_grab()函数独占指针。函数的第一个参数是被独占的窗口，在独占期间，这个窗口将接收到事件。下一个参数应该是 TRUE或FALSE，它指定事件是否只发生在独占的窗口，或者还包括它的子窗口。 confine_to参数指定一个限制指针的窗口。用户不能将指针移出这个窗口。在独占期间，可以指定不同的光标形状。如果不想改变光标，将 cursor设置为NULL。因为光标是一种服务器端资源，直到独占结束， X不会释放它， 调用完gdk_pointer_grab()后可以立即销毁光标。

最后一个参数time指定独占何时生效，它是指服务器时间。它主要用于解决当两个客户试图同时独占指针时的冲突， time必须在上次独占之后，并且不能是在将来的某个时间。通常，总是将 time成员设置为正在处理事件的时间，或者使用 GDK_CURRENT_TIME宏。GDK_CURRENT_TIME让X服务器代入当前时间。

如果成功，gdk_pointer_grab()返回TRUE。如果独占窗口或confine_to是隐藏的，另一个客户已经独占，或者任何参数是无效的时，它也有可能失败。遗憾的是很少有应用程序检查这个返回值，这是一个缺陷。不过，因为很难触发这个错误，忽略这个问题一般不会引起严重后果。

调用gdk_pointer_ungrab()函数可以取消指针独占，参数time和gdk_pointer_grab()中的time参数一样。 可以用gdk_pointer_is_grabbed()函数来检查指针是否被独占了。当使用完之后，必须将其取消独占，否则在指针被独占时用户不能使用其他的应用程序。

注意，Gdk级的独占指针和Gtk+级的独占指针的概念是不一样的。 Gtk+中的独占将某些事件重定向到具有独占构件，生成一个“模态”构件，比如对话框。 Gtk+的独占只影响当前的应用程序，只有发生在当前应用程序的某个构件上的事件才会被重定向。 Gdk的独占范围更宽，包含了整个X服务器，而不仅仅是一个应用程序。

函数列表：独占指针

```
#include <gdk/gdk.h>
gint gdk_pointer_grab(GdkWindow* window,
                gint owner_events,
                GdkWindow* confine_to,
                GdkCursor* cursor,
                guint32 time)

void gdk_pointer_ungrab(guint32 time)

gint gdk_pointer_is_grabbed()
```

16.6.3 改变光标

在任何时候都可以改变光标的形状，光标形状是用 gdk_window_set_cursor() 函数设置的，它与特定的窗口相关。缺省时，窗口使用它们父窗口的光标。将窗口的光标设置为 NULL，就可以恢复它的缺省光标。

有两种方法创建光标。最简单的方法就是从 X中的光标字体中挑选一个。光标字体包含光标而不是字符，可以用 xfd -fn命令察看这些光标字体。每个光标形状在 gdk/gdkcursors.h中都定义了一个常数。gdk_cursor_new()可以接受这种常数作为它唯一的参数：

```
GdkCursor* cursor;
cursor = gdk_cursor_new(GDK_CLOCK);
gdk_window_set_cursor(window, cursor);
gdk_cursor_destroy(cursor);
```

注意，一旦将光标连接到窗口，就可以销毁光标了。 GdkCursor是服务器端资源的客户端句柄，只要它还在使用， X就会保持服务器端资源。

如果光标字体中没有一种适合需要，可以用位图创建一个定制的光标。实际上需要两个位图：源pixmap和屏蔽位图。因为它们是位图，每个像素都是 on或off(0或1)。如果在屏蔽位图中像素是 1，这个像素点会是透明的。如果像素在 pixmap中也是 1，它会以传递到gdk_cursor_new_from_pixmap()函数中的fg(前景色)颜色显示。如果像素点在屏蔽位图中是 1，但是在源pixmap中是0，它会以bg(背景色)颜色显示。源和屏蔽 pixmap的大小必须是一样的，它们的深度必须都是1。

前景色和背景色应该明暗对比强烈，这样光标才会在任何背景上都是可见的。大多数光标都用前景色绘制，用背景色作为轮廓 (要看到这一点，在黑色的背景上移动 X光标，可以看到在边缘上是白色的轮廓)。要做到这一点，屏蔽位图应该和源 pixmap形状一样，尺寸稍大一些。

gdk_cursor_new_from_pixmap()函数的最后两个参数是光标热点的坐标。这个点应该是鼠标指针指向点的位置 —— 箭头光标的尖端，或者十字交叉光标的中心。如果光标热点不在位图内部，调用gdk_cursor_new_from_pixmap()函数创建光标会失败。

函数列表： GdkCursor

```
#include <gdk/gdk.h>
GdkCursor* gdk_cursor_new(GdkCursorType cursor_type)

GdkCursor* gdk_cursor_new_from_pixmap(GdkPixmap* source,
                        GdkPixmap* mask,
                        GdkColor* fg,
                        GdkColor* bg,
                        gint x,
                        gint y)
void gdk_cursor_destroy(GdkCursor* cursor)

void gdk_window_set_cursor(GdkWindow* window,
                    GdkCursor* cursor)
```

16.7 字体

X字体是一种服务器端资源。本质上来说，字体是位图的集合，这些位图代表了不同的字

符。一种字体中不同字符的位图具有相似的尺寸和风格。 Gdk允许用一种称为GdkFont的客户端句柄来操纵字体。

要获得一种GdkFont，调用gdk_font_load() 函数(或者使用先前存在的GtkStyle中的字体)。字体是用字体名加载的，字体名是一个相当棘手的话题。字体名遵从称为“ X 逻辑字体描述”或XLFD的约定。要了解 XLFD的最好的方法是使用随 X发布的xfontsel实用程序。还可以用 xlsfonts 程序得到一个X服务器上的字体名列表。

字体名是一个用连字号分隔开的域组成的字符串。每个域描述了字体的一些方面的信息。例如：

```
-misc-fixed-medium-r-normal--0-0-75-75-c-0-iso8859-1
```

或

```
 -adobe-new century schoolbook-bold-i-normal--11-80-100-100-p-66-iso8859-1
```

十四个域分别是：

Foundry：字体供应商的名字，例如 Adobe或Sony。对随X发布的普通字体，用 misc。

Family：字体的字样或风格，例如 Courier、Times、Helvetica等等。

Weight：粗体、半粗体、一般体等。

Slant：斜体、罗马体、倾斜体 (缩写为I，r，或o)。

Set Width：字体的“ 比例宽度 ”，包括普通、紧缩、半紧缩等几种可能取值。

Add Style：字体的任何附加信息都可以放在这里，它用于区别名字相同的两种字体。此处的字符串没有限制。

Pixels：字体的像素数目。

Point：字体的点数，以十分之一点为基准。一点是 1/72英寸，点和像素的关系是由 X服务器所识别的显示器的分辨率 (每英寸的字点数)决定的。典型情况，人们并不配置 X服务器以适应监视器的特性，所以X服务器对当前分辨率的设置也许是不准确的。

Horizontal Resolution：水平显示分辨率，以要显示的字体的每英寸字点数计量。

Vertical Resolution：字体的垂直分辨率。

Spacing：可为两种取值－Monospace (缩写为 m)或proportional (缩写为p)。指明所有的字符都是相同的宽度或是不同的字符可以有不同的宽度。

Average Width：在字体中所有字体的平均宽度，以 1/10像素计。

Character Set Registry：定义字符集的组织或标准。

Character Set Encoding：指定具体字符集的编码方式。最后两个域结合起来指定了字符集。对欧洲语言来说总是使用 iso8859-1。这是“ Latin-1 ”字符集，它是一种八位编码，其中包含 ASCII作为它的子集。

使用字体时，不必指定所有 14个域。可以使用“ 通配符 ”：*代表任意个字符；?代表任意一个字符。例如， 160点粗体Roman Helvetica字体可以像下面这样：

```
-*-helvetica-bold-r-*-*-*-160-*-*-*-*-*-*
```

当传递字体名到gdk_font_load()函数时，应该只将它作为一种缺省选择。其他国家的用户可能想使用适合于他们的语言的字体，美国和欧洲用户可能也想自己定义字体。还有，有些字体在其他服务器也许并不存在。因而，应该提供一种方法来定制要用到的任何字体。最容易的方法就是使用从构件的GtkStyle中取得的字体。

如果没有找到与所给字体名匹配的字体， gdk_font_load()函数会返回NULL。当使用过一种字体后，应该调用gdk_font_unref()函数释放它。

加载字体时，至少要指定字体名、 weight、slant、size等属性 —— 否则，通配符*也许会随机加载一个粗体、斜体的字体，这可能不是实际需要的。 Xlib编程手册建议总是要指定字体的点数，这样用户在不同的监视器上都能得到正确的显示效果。不过， X服务器不一定能正确识别出显示分辨率，所以这样做的理论意义多于实际意义。也许更好的方法是指定像素值，因为可以知道所显示的其他元素的像素大小。没有完美的方法，最好使应用程序的字体是可配置的。

函数列表：GdkFont

```
#include <gdk/gdk.h>
GdkFont* gdk_font_load(const gchar* font_name)
void gdk_font_unref(GdkFont* font)
```

字体规格

要使用字体，典型情况下需要知道字体规格的详细信息。字体规格用于字符之间的定位，以及决定用字体绘制字符串时的大小。最基本的规格是字体的上行和下行。文本放在一条基线上，基线就像笔记本的纸上的一把尺子。每个字符的底部都挨着基线。一些字符 (比如小写的“ p ”和“ y ”)向基线下延伸。字体的下行就是字符在基线下的最大距离。它的上行值就是在基线之上的距离。字体的高度是上行值和下行值的和。绘制多行字体时，至少得在每条基线间留出字体高度这样的距离。

在GdkFont结构定义中有上行(ascent)和下行(descent)成员：

```
typedef struct _GdkFont GdkFont;
struct _GdkFont
{
  GdkFontType type;
  gint ascent;
  gint descent;
};
```

第一个成员type将字体从字体集中区别开来；字体集用于显示非欧洲语言。

在字体中，个别字符有自己的上行值和下行值；字符的上行值和下行值必须小于或等于字体的上行值和下行值。Gdk能计算特定字符串的最大上行值和下行值的和，而不是计算整个字体的。该高度应该小于或等于字体的高度。相关的函数是 gdk_string_height()、gdk_text_height()和gdk_char_height()。gdk_text_height()函数与gdk_string_height()略有不同，前者接受字符串的长度作为一个参数，而 gdk_string_height()直接调用strlen()。如果已经知道了字符串的长度，应该使用gdk_text_height()函数。

除了垂直规格，在字体中，每个字符还有三个描述它的水平尺寸的规格。字符的 width是从一个字符的左边起点到下一个字符开始的距离。注意， width不是到字符左边起点到最右边像素点的距离，在一些字体中，特别是斜体中，字符可能会倾斜超过下一个字符的开始点。left-side-bearing或lbearing是从字符左边起点到最左边像素点的距离； right-side-bearing或rbearing是从字符起点到最右边像素点的距离 。因而， rbearing可能会比width值还大。在斜体字中，字符倾斜越过了下一个字符的起始点。

所有返回字符或字符串宽度的Gdk函数都返回字符的宽度，或者返回在字符串内字符宽度

的总和。如果最右边的字符的rbearing值比它的宽度大，字符串也许需要比gdk_string_width(), gdk_text_width()，或gdk_char_width()所返回值更多的空间。对测量高度的函数， _string_函数变体计算字符串的长度，而_text_变体函数接受一个预先计算好的长度作为参数。

通常，以_measure结尾的函数才是编程所需要的。对一个有N个字符串的函数，这些函数返回前N－1个字符的宽度之和，加上最后一个字符的 rbearing值。也就是，它们会考虑rbearing值也许会比 width值大。如果要决定为绘制一个字符串留多少空间，可能要用gdk_string_measure()、gdk_text_measure()或gdk_char_measure()函数。不过，有时不应该考虑字符的rbearing值，例如，如果要居中对齐字符串，也许使用字符的宽度值更合适（因为如果不使用宽度，一个很小的、超过字符串宽度的斜体装饰并不会“填充”空白区域，字符串看起来会略为中央靠左）。

gdk_text_extents()和gdk_string_extents()返回字符串的所有规格，包括lbearing、rbearing、width、ascent和descent等。函数返回的left-side-bearing是字符串最左边像素点的， right-side-bearing是最右边的像素点的，和gdk_text_measure()返回的一样。返回的width值是字符宽度的总和，和gdk_text_width()返回的值一样。

所有的字体规格都是在客户端计算的，所以与其他的绘图函数相比，这些函数占用资源还不算昂贵。

函数列表：字体规格

```
#include <gdk/gdk.h>
gint gdk_string_width(GdkFont* font,
                const gchar* string)
gint gdk_text_width(GdkFont* font,
              const gchar* string,
              gint string_length)
gint gdk_char_width(GdkFont* font,
              gchar character)
gint gdk_string_measure(GdkFont* font,
                  const gchar* string)
gint gdk_text_measure(GdkFont* font,
                const gchar* string,
                gint string_length)
gint gdk_char_measure(GdkFont* font,
                gchar character)
gint gdk_string_height(GdkFont* font,
                 const gchar* string)
gint gdk_text_height(GdkFont* font,
               const gchar* string,
               gint string_length)
gint gdk_char_height(GdkFont* font,
               gchar character)
void gdk_string_extents(GdkFont* font,
                  const gchar* string,
                  gint* lbearing,
                  gint* rbearing,
                  gint* width,
                  gint* ascent,
```

```
                    gint* descent)
void gdk_text_extents(GdkFont* font,
                      const gchar* string,
                      gint string_length,
                      gint* lbearing,
                      gint* rbearing,
                      gint* width,
                      gint* ascent,
                      gint* descent)
```

16.8 图形上下文

一个图形上下文，或者GC（Graphics Context），是一套在绘图时要用到的参数(比如颜色、剪裁屏蔽值、字体等等)。它是一种服务器端资源，就像pixmap和窗口一样。GC减少了Gdk绘图函数的参数个数，也减少了每个绘图请求从客户到服务器间传递的参数的数目。

与GdkWindowAttr类似，图形上下文可以用GdkGCValues结构创建。结构中包含了图形上下文中所有的特性，还可以传递gdk_gc_new_with_values()标志指出哪一个成员是有效的。其他的成员保留其缺省值。还可以用gdk_gc_new()函数(这种方法通常更容易)创建一个全为缺省值的GC。创建GC之后，还有一些函数用来改变GC的设置，但是要记住，每次改变GC设置值都需要一条消息传递到X服务器。

所有的GC都是不可以互换的，它们都与特定的深度和视件相关联。 GC的深度和视件必须与要绘图的可绘区的深度和视件相匹配。 GC的深度和视件是从传递到gdk_gc_new()函数的GdkWindow*参数中获得的，所以处理这种问题的最容易的方法就是在要绘图的窗口上创建GC。

下面是GdkGCValues结构的定义：

```
typedef struct _GdkGCValues GdkGCValues;
struct _GdkGCValues
{
  GdkColor           foreground;
  GdkColor           background;
  GdkFont           *font;
  GdkFunction        function;
  GdkFill            fill;
  GdkPixmap         *tile;
  GdkPixmap         *stipple;
  GdkPixmap         *clip_mask;
  GdkSubwindowMode   subwindow_mode;
  gint               ts_x_origin;
  gint               ts_y_origin;
  gint               clip_x_origin;
  gint               clip_y_origin;
  gint               graphics_exposures;
  gint               line_width;
  GdkLineStyle       line_style;
  GdkCapStyle        cap_style;
  GdkJoinStyle       join_style;
};
```

前景色（foreground成员）是画线、圆或其他形状时的“画笔颜色”。背景色（background成员）

的用处依赖于特定的绘画操作。这些颜色必须是用gdk_color_alloc()函数在当前颜色表中分配的。

font成员没有用到：在Xlib中绘制文本时用它指定字体。在Gdk以前的版本中，它也有同样的作用；但是新的绘制文本的Gdk程序都要求一个GdkFont*参数。一个Xlib图形上下文只能存储无格式的字体，但是GdkFont能够代表一个字体集(用以绘制一些外语文字)。

function成员指定要画的像素点与可绘区上已有的像素点如何结合起来。有许多种可能取值，但是只有两种是最常用的：

- GDK_COPY：缺省值。它忽略已存在的像素点(只是将新的像素点画在上面)。
- GDK_XOR：将旧的和新的像素点一种可反转的方式结合起来。也就是，如果执行两次GDK_OR操作，头一次绘图就会被第二次操作取消。GDK_XOR通常用于“擦除”，可以恢复可绘区原来内容。

GdkGCValues的fill成员决定如何使用GdkGCValues中的tile和stipple成员。其中tile成员是一个与目的可绘区深度相同的pixmap图片，它被反复复制到目的可绘区，将它们拼贴起来——第一次拼贴的原点是(ts_x_origin, ts_y_origin)。而stipple成员是一个位图(深度为1的pixmap)；它也是从(ts_x_origin, ts_y_origin)开始拼贴的。下面是fill的可能取值：

- GDK_SOLID：忽略tile和stipple成员。绘图形状是用前景色和背景色绘制的。
- GDK_TILED：绘图形状用tile成员指定的pixmap图片绘制，而不是用前景色和背景色。用GDK_TILED模式绘画会擦除可绘区上的任何内容，显示由tile成员指定的图片的拼贴图形。
- GDK_STIPPLED：用stipple中定义的位绘制图形。也就是，在stipple成员中未设置的位不会绘出。
- GDK_OPAQUE_STIPPLED：用前景色绘制在stipple中设置的位，没有在stipple中设置的位用背景色绘制。

有些X服务器并没有有效实现上面所有的fill模式值，所以使用时可能会很慢。

clip_mask成员是可选的，它是一个位图，只有在这个位图中设置了的位才会画出。从clip_ mask到可绘区的映射是由clip_x_origin和clip_y_origin值决定的，这些定义了与clip_mask中(0,0)对应的可绘区的坐标。也可以设置一个剪裁矩形(最常用的、也是最有用的形式)或一个剪裁区域(区域就是在屏幕上的任意范围，典型情况是一个多边形或矩形列表)。用gdk_gc_set_clip_rectangle()设置剪裁矩形：

```
GdkRectangle clip_rect;
clip_rect.x = 10;
clip_rect.y = 20;
clip_rect.width = 200;
clip_rect.height = 100;
gdk_gc_set_clip_rectangle(gc, &clip_rect);
```

要关闭“剪裁”，将剪裁的矩形、区域或剪裁屏蔽值设置为NULL。

GC的subwindow_mode只与可绘区是否为一个窗口有关。缺省设置是GDK_CLIP_BY_CHILDREN；这意指子窗口不会被在父窗口上的绘图影响。这会造成一个假象：子窗口在父窗口的“上面”，并且子窗口是不透明的。GDK_INCLUDE_INFERIORS在所有在“上面”的子窗口上绘制，改写子窗口上包含的任何图形——通常不使用这种模式。如果确实在使用GDK_INCLUDE_INFERIORS模式，可能要使用GDK_XOR作为绘图函数，因为它允许恢复子窗口原先的内容。

graphics_exposures是一个布尔值，缺省是TRUE，它决定gdk_window_copy_area()是否产生expose事件。

GC的最后四个值决定怎样画线。这些值用于画线，包括未填充多边形的边框以及弧线。line_width域决定线的宽度(以像素计)。宽度为0的线称为一条“细线”，细线是一个像素宽的线，绘制得非常快(通常使用硬件加速)，但是画的具体像素依赖于所使用的X服务器。为了一致性，最好使用宽度为1的线。

line_style域可以是下面三种值：

- GDK_LINE_SOLID是缺省值；一条实线。
- GDK_LINE_ON_OFF_DASH用前景色画一条虚线，将虚线的off(关闭)部分空着。
- GDK_LINE_DOUBLE_DASH用前景色画一条虚线，但是虚线的off(关闭)部分用背景色绘制。

虚线是gdk_gc_set_dashes()指定的；GdkGCValues中并不包括这个域。 gdk_gc_set_dashes()需要三个参数：

- dash_list是一个虚线长度的数组。偶数号的长度是“ on”（打开）部分，它们是用前景色绘制的；奇数号的长度是“ off”（关闭）部分，它们不画出，或者用背景色绘制，具体绘制方法依赖于line_style。长度值不能是0，所有的值必须是正数。
- dash_offset是在虚线列表中第一个像素的索引号。也就是，如果在 dash_list中指定了5个on和5个off，并且dash_offset是3，绘制的线将从第3个on虚线开始。
- N是在dash_list中的元素的个数。

可以设置一个古怪的虚线模式，例如：

```
gchar dash_list[] = { 5, 5, 3, 3, 1, 1, 3, 3 };
gdk_gc_set_dashes(gc, 0, dash_list, sizeof(dash_list));
```

缺省的dash_ list是{4, 4}，偏移量是0。

图16-1显示了一些用GDK_LINE_DOUBLE_DASH画的虚线。图形上下文的前景色是黑色，背景色是亮灰色。头5根线是缺省的{4, 4}虚线模式，偏移量分别是0、1、2、3和4。记住，缺省值是0。图16-2显示了这5根线的放大图。最后的一根线就是上面提到的古怪虚线模式，它的放大图显示在图16-3。

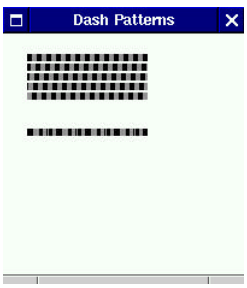

图16-1 5根虚线，线型为GDK_LINE_DOUBLE_DASH

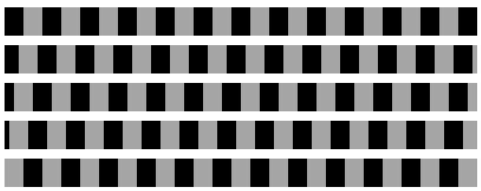

图16-2 缺省的虚线模式，偏移量不同

cap_style决定X怎样画线的端点(或虚线端点)。它有4种可能取值：

- GDK_CAP_BUTT：是缺省值，它意味着线的端点是正方形的。

- GDK_CAP_NOT_LAST：对应一个像素宽度的线，最后一个像素忽略不画。其他与GDK_CAP_BUTT一样。
- GDK_CAP_ROUND：在线的端点画一个小弧线，由线的端点向两边延伸。弧线的中心是线的端点，半径是线宽的一半。对一个像素宽的线，它没有什么效果 (因为没有办法画一个像素宽的弧线)。
- GDK_CAP_PROJECTING：将线延伸，越过它的终点半个线宽。它对一个像素的线没有效果。

图16-3 复杂的虚线模式

join_style参数影响画多边形或在一个函数中画多条线时，各线之间如何连接。如果把线想象成一个细长的矩形，就很容易弄清楚线之间并不是平滑连接的。在连接的两个端点之间有一个凹槽。对这个凹槽有三种处理方法，也就是 join_style的三种可能取值：

- GDK_JOIN_MITER：是缺省值，在线交叉的地方画一个尖角。
- GDK_JOIN_ROUND：在交叉的凹槽处画一个弧线，画一个圆形的转角。
- GDK_JOIN_BEVEL ：用最小的可能的形状填充凹槽，画一个平坦的转角。

函数列表： GdkGC

```
#include <gdk/gdk.h>
GdkGC* gdk_gc_new(GdkWindow* window)
GdkGC* gdk_gc_new_with_values(GdkWindow* window,
                    GdkGCValues* values,
                    GdkGCValuesMask values_mask)
void gdk_gc_set_dashes(GdkGC* gc,
                gint dash_offset,
                gchar dash_list,
                gint n)
void gdk_gc_unref(GdkGC* gc)
```

GC 属 性

属性	GdkGCValuesMask	Modifying	函 数	缺省值
GdkColor	foreground	GDK_GC_FOREGROUND	gdk_gc_set_foreground()	黑色
GdkColor	background	GDK_GC_BACKGROUND	gdk_gc_set_background()	白色
GdkFont	*font	GDK_GC_FONT	gdk_gc_set_font	依赖于X服务器
GdkFunction	function	GDK_GC_FUNCTION	gdk_gc_set_function()	GDK_COPY
GdkFill	fill	GDK_GC_FILL	gdk_gc_set_fill()	GDK_SOLID
GdkPixmap	*tile	GDK_GC_TILE	gdk_gc_set_tile()	用前景色填充的Pixmap
GdkPixmap	*stipple	GDK_GC_STIPPLE	gdk_gc_set_stipple()	所有位都是on(打开)的位图
GdkPixmap	*clip_mask	GDK_GC_CLIP_MASK	gdk_gc_set_clip_mask()	none
GdkSubwindowMode	Subwindow_mode	GDK_GC_SUBWINDOW	gdk_gc_set_subwindow()	GDK_CLIP_BY_CHILDREN

(续)

属性	GdkGCValuesMask	Modifying	函 数	缺省值
Gint	ts_x_origin	GDK_GC_TS_X_ORIGIN	gdk_gc_set_ts_origin()	0
Gint	ts_y_origin	GDK_GC_TS_Y_ORIGIN	gdk_gc_set_ts_origin()	0
Gint	clip_x_origin	GDK_GC_CLIP_X_ORIGIN	gdk_gc_set_clip_origin()	0
Gint	clip_y_origin	GDK_GC_CLIP_Y_ORIGIN	gdk_gc_set_clip_origin()	0
Gint	graphics_exposures	GDK_GC_EXPOSURES	gdk_gc_set_exposures()	TRUE
Gint	line_width	GDK_GC_LINE_WIDTH	gdk_gc_set_line_attributes()	0
GdkLineStyle	line_style	GDK_GC_LINE_STYLE	gdk_gc_set_line_attributes()	GDK_LINE_SOLID
GdkCapStyle	cap_style	GDK_GC_CAP_STYLE	gdk_gc_set_line_attributes()	GDK_CAP_BUTT
GdkJoinStyle	join_style	GDK_GC_JOIN_STYLE	gdk_gc_set_line_attributes()	GDK_JOIN_MITER
Gchar	dash_list[]	none	gdk_gc_set_dashes()	{4, 4}
Gint	dash_offset	None	gdk_gc_set_dashes()	0

16.9 绘图

一旦理解了绘图区、颜色、视件、图形上下文和字体，绘图就会变得很简单了。本节简要介绍了Gdk的相关绘图函数。要注意的是绘图是一种服务器端的操作，例如，如果要画一条线，Xlib会将线的端点传给服务器，服务器用特定的 GC(GC也是一种服务器端资源)做实际的绘图操作。创建绘图应用程序时通常要考虑程序的性能。

16.9.1 画点

可以用gdk_draw_point()函数画一个点，或用gdk_draw_points()函数画多个点。点是用当前的前景颜色画的。多个点是由一个 GdkPoint数组给出的。GdkPoint结构如下所示：

```
typedef struct _GdkPoint GdkPoint;
struct _GdkPoint
{
  gint16 x;
  gint16 y;
};
```

注意，X坐标是从绘图区的左上角开始的，是一个有符号的 16位整数。

函数列表：画点

```
#include <gdk/gdk.h>
void gdk_draw_point(GdkDrawable* drawable,
                    GdkGC* gc,
                    gint x,
                    gint y)

void gdk_draw_points(GdkDrawable* drawable,
                     GdkGC* gc,
                     GdkPoint* points,
                     gint npoints)
```

16.9.2 画线

用gdk_draw_line()函数画一条线，将线的端点作为它的参数。要画几条连接着的线，可调用gdk_draw_lines()函数，并将点的列表作为参数传到函数中。 Gdk会将点连接起来。要画多条不必连接起来的线，可调用 gdk_draw_segments()函数，将线段列表作为参数传到函数中。

GdkSegment结构是这样定义的：

```
typedef struct _GdkSegment GdkSegment;
struct _GdkSegment
{
  gint16 x1;
  gint16 y1;
  gint16 x2;
  gint16 y2;
};
```

如果在同一个绘画请求中画出的线或者线段的端点是相交的，它们会以 GC中的“连接风格”连接起来。

函数列表：画线

```
#include <gdk/gdk.h>
void gdk_draw_line(GdkDrawable* drawable,
            GdkGC* gc,
            gint x1,
            gint y1,
            gint x2,
            gint y2)

void gdk_draw_lines(GdkDrawable* drawable,
             GdkGC* gc,
             GdkPoint* points,
             gint npoints)

void gdk_draw_segments(GdkDrawable* drawable,
                GdkGC* gc,
                GdkSegment* segments,
                gint nsegments)
```

16.9.3 矩形

用gdk_draw_rectangle()函数画矩形。其中 filled参数指明是否填充矩形，设为 TRUE意味着填充它， FALSE则不填充。

函数列表：画矩形

```
#include <gdk/gdk.h>
void gdk_draw_rectangle(GdkDrawable* drawable,
                 GdkGC* gc,
                 gint filled,
                 gint x,
                 gint y,
                 gint width,
                 gint height)
```

16.9.4 画弧

gdk_draw_arc()函数画一个椭圆或椭圆弧。弧可以是填充的，也可以是不填充的，第 3个参数切换填充状态。第 4到第7个参数描绘了一个矩形，椭圆内切于这个矩形； angle1参数是画弧的起始角，它以时钟三点钟位置为 0°；angle2是绕圆弧旋转的角度，如果值是正的，逆时针旋转，否则顺时针旋转。参数 angle1和angle2都是以1/64度来指定的，所以360°就是360×64。这样可以更精确地指定弧形的尺寸和形状，而不需使用浮点数。参数 angle2不要超过360°，因为对椭圆来说旋转角度超过360°没有什么意义。

要画一个圆，在正方形内从0到360×64画一个弧：

```
gdk_draw_arc(drawable, gc, TRUE,
             0, 0,
             50, 50,
             0, 360*64);
```

要画一个半椭圆，改变长轴和短轴的比例，角度从 0到180×64：

```
gdk_draw_arc(drawable, gc, TRUE,
             0, 0,
             100, 50,
             0, 180*64);
```

有许多X服务器在用来画填充圆弧的边缘时是很难看的，特别是很小的圆看起来会不太圆。要解决这个问题，可以在画圆时把圆的轮廓也画出来。

函数列表：画弧

```
#include <gdk/gdk.h>
void gdk_draw_arc(GdkDrawable* drawable,
            GdkGC* gc,
            gint filled,
            gint x,
            gint y,
            gint width,
            gint height,
            gint angle1,
            gint angle2)
```

16.9.5 多边形

gdk_draw_polygon()函数用来画一个填充或不填充的多边形。注意，也可以用gdk_draw_lines()函数画一个不填充的多边形。这那种方法并没有什么实质区别。gdk_draw_polygon()函数的参数与gdk_draw_lines()中的参数完全一样。多边形不一定必须是凸多边形。它的边可以是交叉的(自交)。自交的多边形用一种“奇－偶规则”来填充，规则是重叠奇数次的多边形区域是不填充的。也就是，如果多边形不是重叠的，它就是完全填充的；如果一个区域重叠了一次，它不会被填充；如果它重叠了两次，它会被填充等等。

函数列表：画多边形

```
#include <gdk/gdk.h>
void gdk_draw_polygon(GdkDrawable* drawable,
              GdkGC* gc,
              gint filled,
```

```
                GdkPoint* points,
                gint npoints)
```

16.9.6 文本

有两个函数可用于绘制字符串： gdk_draw_text()和gdk_draw_string()。其中gdk_draw_text()是最好的方法。 gdk_draw_text()用字符串的长度作为参数，而gdk_draw_string()函数用strlen()函数计算字符串的长度。函数中的x和y坐标指定要绘制的文本基线的左边位置。文本是用前景颜色绘出的。

在Gdk中没有方法画出缩放的或旋转的文本。 GnomeCanvasText构件提供了一个较慢的、质量较低的方法来渲染缩放的和旋转的文本。如果需要高质量的按比例缩放的和旋转的文本，则需要使用额外的库函数，比如说对 Type 1 fonts 字体使用t1lib 库函数，或对True Type fonts字体使用FreeType库函数，或者使用Display Postscript的X扩展(XDPS)。GNU项目组正致力于开发一个免费的XDPS的Linux版本。作为gnome-print 库的一部分， Gnome项目组也正在开发一个文本绘图方案。

函数列表：画文本

```
#include <gdk/gdk.h>
void gdk_draw_string(GdkDrawable* drawable,
                GdkFont* font,
                GdkGC* gc,
                gint x,
                gint y,
                const gchar* text)

void gdk_draw_text(GdkDrawable* drawable,
               GdkFont* font,
               GdkGC* gc,
               gint x,
               gint y,
               const gchar* text,
               gint text_length)
```

16.9.7 pixmap像素映射图形

gdk_draw_pixmap()从一个像素映射图形中复制一个区域到另一个绘图区 (像素映射或窗口)。绘图区的源和目的必须有相同的深度和视件（ visual)。如果给width或height参数传递-1值，会将源pixmap图片全部复制过去。源图片可以是任何绘图区，包括窗口，但是如果源是一个窗口，使用gdk_window_copy_area()函数将使代码更清晰。下面是 gdk_draw_pixmap()函数的声明形式。

函数列表：画pixmaps图形

```
#include <gdk/gdk.h>
void gdk_draw_pixmap(GdkDrawable* drawable,
                GdkGC* gc,
                GdkDrawable* src,
                gint xsrc,
                gint ysrc,
```

```
gint xdest,
gint ydest,
gint width,
gint height)
```

16.9.8 RGB缓冲

Gdk的GdkRGB模块允许将一个图像数据的客户端缓冲复制到一个绘图区。如果要大量操作图像，或将图像数据复制到服务器，最好使用这种方法。因为 pixmap是一种服务器端对象，不能直接操纵GdkPixmap。用gdk_draw_point()函数复制图像数据到服务器中是非常慢的，因为每一个点都要求一个服务器请求(也许还不止一个，因为还需要为每个点更改其 GC)。

本质上来说， GdkRGB用一个称为GdkImage的对象在一个请求内将图象数据复制到服务器。这样还是有点慢(大量的数据需要复制)，但是GdkRGB是已经对此进行了优化，并且如果客户和服务器碰巧是在同一台机器上，它还会使用共享内存。所以在 X结构中，这是完成这个任务最快的方法。它还会处理一些微妙的问题 (比如说适应给定X服务器上的颜色表和视件)。

GdkRGB函数是在一个单独的头文件 gdk/gdkrgb.h.里面定义的。在使用任何 GdkRGB函数前，必须用gdk_rgb_init()函数初始化GdkRGB模块，它设置一些要用到的视件和颜色表，以及一些内部的数据结构。

要将RGB缓冲复制进去的绘图区必须使用 GdkRGB的视件和颜色表。如果绘图区是构件的一部分，保证这一点最简单的方法就是在创建构件时将 GdkRGB视件和颜色表压入构件的视件和颜色表栈中：

```
GtkWidget* widget;
gtk_widget_push_visual(gdk_rgb_get_visual());
gtk_widget_push_colormap(gdk_rgb_get_cmap());
widget = gtk_whatever_new();
gtk_widget_pop_visual();
gtk_widget_pop_colormap();
```

如果创建包含绘图区的顶级窗口时像上面那么做，而不是在只在创建绘图区时做，当前的Gtk+版本表现还是较好的。不过，原则上可以只对绘图区做这样的工作。

GdkRGB能理解几种类型的图像数据，包括 24位和32位RGB数据、8位灰度级，以及 8位的按RGB值数组索引的数据(一种客户端GdkRgbCmap)。本节只介绍最简单的24位 RGB数据，这种缓冲数据是用gdk_draw_rgb_image()函数渲染的。有一些单独的函数用于渲染其他缓冲类型，但是所有其他函数本质上的工作原理都是一样的。

24位RGB缓冲是一个一维的字节数组，每个字节三个一组组成像素值 (0位是红色， 1位是绿色， 2位是蓝色)。三个数字描述了数组的大小和字节的位置：

width是图像每行的像素数 (三个字节)。

height是图像的行数。

Rowstride(行跨距)是行之间的字节数。也就是说，对一个 rowstride为r的缓冲，如果第n行从数组索引i开始，那么第n+1行从数组索引i+r开始。 rowstride不一定是缓冲宽度的三倍；如果源指针和rowstride都与4字节边界对齐， GdkRGB会较块。指定一个 rowstride允许用填充量来达到这个效果。

gdk_rgb_draw_image()中的x、y、width和height参数定义了目标可绘区中的一个区域，

RGB缓冲就复制到这个区域。RGB缓冲至少要有width列、height行。RGB缓冲的第0行第0列将复制到可绘区的(x, y)。

gdk_draw_rgb_image 函数中的dither（抖动）参数在有限调色板的显示器上模拟很多颜色。dither只在8位和16位的显示器上管用，24位的显示器的调色板没有什么限制。dither参数是一个枚举类型，有下列几种可能取值：

- GDK_RGB_DITHER_NONE 规定没有抖动。它对画文本或用较少的颜色画线很合适，但是对处理照片不合适。
- GDK_RGB_DITHER_NORMAL 规定在8位的显示器上抖动。但在16位的显示器上不抖动。这通常是质量/性能的一个折衷。
- GDK_RGB_DITHER_MAX 规定在8位和16位的显示器上都抖动。在16位显示器上使用GDK_RGB_DITHER_MAX能够使绘图质量获得提高，但是与性能损失相比可能不值得这么做。

gdk_draw_rgb_image()的gc参数被简单地传递到gdk_draw_image()函数中(GdkRGB在内部使用GdkImage)。最好使用有意义的gc值(比如剪裁屏蔽值、绘图函数，以及 subwindow模式)。

函数列表：GdkRGB

```
#include <gdk/gdkrgb.h>
void gdk_rgb_init()
GdkColormap* gdk_rgb_get_cmap()
GdkVisual* gdk_rgb_get_visual()
void gdk_draw_rgb_image(GdkDrawable* drawable,
                GdkGC* gc,
                gint x,
                gint y,
                gint width,
                gint height,
                GdkRGBDither dither,
                guchar* rgb_buf,
                gint rowstride)
```