wxWidgets实践指南

**摘要：**本文介绍了开源框架wxWidgets的基本概念，包括窗口、事件处理和控件布局，主要着眼于界面这个层面，也包含一些实践的经验。为了便于理解，全篇贯穿以一个自定义控件的例子。

**关键字：**C++ wxWidgets GUI 界面 控件 框架

**简介**

wxWidgets是一个类似于MFC的跨平台GUI框架。刚接触的时候，没什么好感。随着这几年慢慢熟悉下来，觉得它真的不错。几乎每一篇介绍wxWidgets的文档都会在开头列举这样几条特色或优点，我们也不例外：

1. No license constraints for commercial products
2. Keep native look & feel of target OS
3. Easy internationalization & unicode support
4. Easy to learn (similarity to MFC)
5. Easy arrangement of UI elements

本文的原意是想分享一些项目开发中积累的经验。但是一味的罗列观点而失去行文的连贯性实在不是一个好的主意。所谓指南也并非要求面面俱到，所以这里主要着眼于GUI这个层面（窗口，事件和控件的布局），而舍弃了诸如数据结构、多线程等辅助性的东西，后者往往本身就比较独立，实际应用中也可能常常被其它程序库所替代（比如wxWidgets的数据结构设计得很糟糕，便用STL和boost替代）。

**窗口 WINDOW**

GUI程序中最直接的东西就是窗口，它是程序留给用户的接口。不同的操作系统中，窗口的概念基本是一致的，封装到wxWidgets这一层后，大致分为以下几类（括号内为对应的类名）：

* Frame (wxFrame)
* Dialog (wxDialog)
* Panel (wxPanel)
* Control (wxControl)

Frame和dialog属于顶层窗口，一般有标题栏，边框，可以移动，等等；panel和control属于非顶层窗口，只能作为顶层窗口的子窗口。还有一些窗口，比如wxScrolledWindow，就是加了滚动条的panel，没有本质区别。

其实从稍低的层面来看，比如从Win32的角度来看，不同窗口的差别并不那么明显。实践的时候，可以通过设定不同的窗口风格（window style）来灵活应用。比如想自定义一个tool tip窗口，以支持多行文字、改变字体和颜色并使用半透明效果，便可以让这个tool tip继承自wxFrame，然后加上这样几个窗口风格：

* wxFRAME\_TOOL\_WINDOW（wxPOPUP\_WINDOW on MAC） => 不出现在任务栏，不抢焦点
* wxSTAY\_ON\_TOP => 总在最上面
* wxBORDER\_NONE => 没有边框

下面说说各类窗口之间的区别，不过这区别只是就一般情况而言，像上面给frame加了某个风格后就不会出现在任务栏的情况，另当别论。

**FRAME vs. DIALOG**

* Frame会出现在任务栏上，dialog不会；（MAC没有任务栏，除外）
* Frame不能直接包含control，必须先包含panel，dialog可以直接包含control；
* Dialog会阻塞命令式事件（见下文）的传输，frame则不会；
* Frame可以有menu，dialog不能有menu；
* Dialog有模态非模态之分，被关闭时并不会销毁自己，可以调用Show/ShowModal再次弹出。

**PANEL vs. DIALOG**

Panel可以看作是非顶层的dialog，或者dialog可以看作是顶层的panel。

**PANEL vs. CONTROL**

* Panel可以包含panel和control，control不可以再包含其他窗口；
* Control放在panel/dialog里，panel/dialog响应tab事件可以对control进行遍历。

如今的GUI程序，尤其是Windows平台，倾向于隐藏缺省的标题栏，自己绘制窗口，自定义控件的外观。这是一种低俗的审美。自定义的窗口和控件一般都不如系统缺省的好用和稳定，只是看上去更花哨一些，而且当系统升级后，它也不能"免费地"享受到新系统在GUI上的改进。然而用户的品味已经被"培养"成这样了，程序长什么样也不再是程序员能说了算。通常的情况是，一些专门从事美工的人先绘出一些酷炫的图片，然后程序员按照这些图片中描述的程序的样子加以实现。这种GUI程序的设计方式真的很不好，但很多项目确实就是这样做的。

自定义窗口相对比较容易，就是上面提到的几点：隐藏缺省的标题栏和边框，然后自己绘制和处理事件，顶多再弄个圆角。自定义控件有点复杂，值得一说。本文将始终贯穿一个自定义控件的例子，逐步完善，借这个例子可以看到wxWidgets的一些细节。现在假如你想自定义一个控件，叫ToolbarButton，顾名思义，就是放在toolbar上的button，功能如下：

* 它有4种不同的状态：normal，hover，pressed，disabled；
* 它有自定义的外观，由图片绘制，每种状态对应一张图片；
* 每次被点击后，它会抛出一个事件。

基于以上对窗口的介绍，我们先让ToolbarButton继承自wxControl：

class ToolbarButton : public wxControl {

DECLARE\_DYNAMIC\_CLASS(ToolbarButton)

DECLARE\_NO\_COPY\_CLASS(ToolbarButton)

public:

enum { Image\_Normal = 0, Image\_Hover, Image\_Pressed, Image\_Disabled, Image\_Count };

void SetImage(int id, const wxImage& image) { m\_images[id] = image; }

const wxImage& GetImage(int id) const { return m\_images[id]; }

ToolbarButton() {}

ToolbarButton(wxWindow\* parent, int id) { Create(parent, id); }

virtual ~ToolbarButton() {}

bool Create(wxWindow\* parent, int id) {

if (!wxControl::Create(parent, id)) {

return false;

}

this->SetCursor(wxCursor(wxCURSOR\_HAND));

return true;

}

private:

wxImage m\_images[Image\_Count];

};

可以看到，ToolbarButton里保存了一个wxImage的数组，对应于每种状态的图片，这些图片可以通过SetImage进行设置。这里只写了构造相关的函数，和RTTI的一些宏（RTTI和MFC的完全一样，不做赘述），至于事件处理和其他细节，会在接下来逐步加入。

**事件 EVENT**

wxWidgets里的事件等价于Win32里的消息（message），Win32里有SendMessage和PostMessage这样的API，wxWidgets里便也封装有wxSendEvent和wxPostEvent这样的函数。不要将这种事件跟Win32 API CreateEvent所创建出来的"事件"相混淆，Win32的事件是一种内核对象，可以对线程进行同步，而wxWidgets的事件（或Win32的消息）主要和窗口应用相关（虽然也常用于线程间通信）。

**EVENT HANDLER 事件处理器**

事件的接收者或处理者叫event handler。在wxWidgets里有一个类叫wxEvtHandler，它是对event handler这个角色的一个抽象，因为事件主要为窗口系统服务，所以所有的窗口类都继承自这个wxEvtHandler。每一个event handler类都可以有（也可以没有）一张（静态的）事件表，这个事件表里存放了事件到事件处理函数的映射信息，还有一个指向父类的事件表的指针。所以对一个event handler的继承结构来说，事件表是像链表一样从子类到基类依次串起来的。和RTTI一样，事件表的机制也类似于MFC。

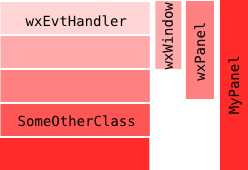
值得注意的是，当使用多继承时，wxEvtHandler必须排在最先，如下：

class MyPanel

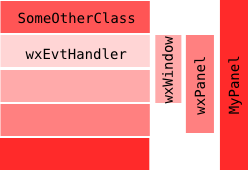
: public wxPanel // 必须排在最先

, public SomeOtherClass

这一点很重要。把wxEvtHandler放在最先，就保证了在内存布局上MyPanel的wxEvtHandler部分也在最先，这就使整体上的MyPanel对象的this指针和wxEvtHandler那部分的this指针保持一致。简单的内存布局如下图：



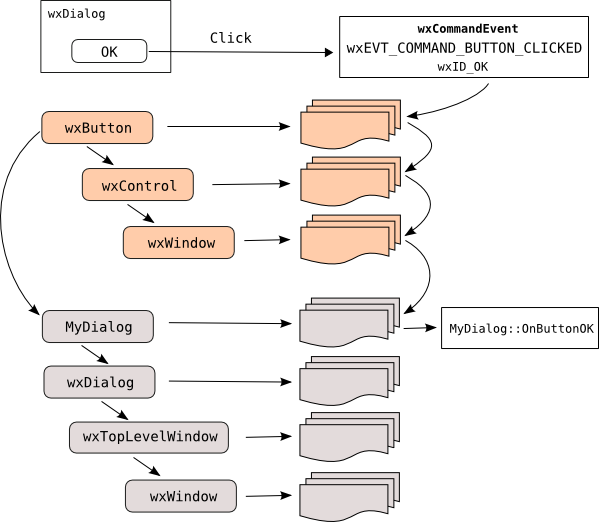
在这里，this指针扮演了两种角色。一方面，在MyPanel的事件处理函数里它应该指向MyPanel整体对象的起始位置；另一方面，它又被wxEvtHandler用来调用匹配到的事件处理函数。假如这个最终被调用的事件处理函数定义在MyPanel里，而不是某个父类里，那这两个this必须一致，否则一旦在这个函数里访问this便会crash。下图是不一致的情形：



**事件的分类**

事件主要分两种，命令式（command）和非命令式。Button被点击，check box被勾选，menu的某个条目被选中，等等，这些都是命令式事件；相反，诸如窗口的激活与否，得到或失去焦点，关闭，背景擦除或绘制，移动，尺寸改变，鼠标和键盘消息，则属于非命令式事件。

这两种事件的区别在于，命令式事件会递归地路由到父类和父窗口，这个递归路由的过程用文字描述起来比较麻烦，而下面这个图就容易理解多了：



图中，click事件依次经过wxButton类层次里的事件表的筛选，没有找到相应的事件处理函数，继而又经过父窗口MyDialog的类层次里的事件表，最终找到事件处理函数MyDialog::OnButtonOK，遂停止路由。这是命令式事件典型的的处理过程。

非命令式事件不会递归地路由到上层窗口，因为没有这个必要，逻辑上也不合理。试想如果父窗口能收到子窗口的移动、尺寸改变或键盘鼠标消息，那么事情会是怎样的混乱。

不过，无论是命令式还是非命令式事件，都可以手动地进行路由，即调用事件的Skip函数。比如前面button click事件的最终处理函数MyDialog::OnButtonOK如果调用了Skip，那么这个click事件还会继续往上路由。

void MyDialog::OnButtonOK(wxCommandEvent& evt) {

// 某些处理...

evt.Skip(); // 继续路由

}

对于非命令式事件，建议调用Skip，这样缺省的事件处理便能执行（如下size event的示例）；对于命令式事件，一般不调用Skip，因为命令式事件通常只应该在一个地方处理。

void MyDialog::OnSize(wxSizeEvent& evt) {

// 某些处理...

evt.Skip(); // 交给缺省的处理

}

至此，可以为ToolbarButton加上一些事件处理了：

class ToolbarButton : public wxControl {

DECLARE\_EVENT\_TABLE()

protected:

void PostEvent();

void OnPaint(wxPaintEvent& evt);

void OnMouseLeftDown(wxMouseEvent& evt);

void OnMouseLeftUp(wxMouseEvent& evt);

void OnMouseEnter(wxMouseEvent& evt);

void OnMouseLeave(wxMouseEvent& evt);

private:

bool m\_pressed; // mouse left down but not up yet.

};

BEGIN\_EVENT\_TABLE(ToolbarButton, wxControl)

EVT\_PAINT(ToolbarButton::OnPaint)

EVT\_LEFT\_DOWN(ToolbarButton::OnMouseLeftDown)

EVT\_LEFT\_UP(ToolbarButton::OnMouseLeftUp)

EVT\_ENTER\_WINDOW(ToolbarButton::OnMouseEnter)

EVT\_LEAVE\_WINDOW(ToolbarButton::OnMouseLeave)

END\_EVENT\_TABLE()

void ToolbarButton::PostEvent() {

// TODO

}

void ToolbarButton::OnPaint(wxPaintEvent& evt) {

wxPaintDC dc(this);

dc.Clear();

int id = Image\_Normal;

if (this->IsEnabled()) {

if (m\_pressed) { id = Image\_Pressed; }

else if (is\_mouse\_in\_window(this)) { id = Image\_Hover; }

} else {

id = Image\_Disabled;

}

wxRect rect = this->GetClientRect();

dc.DrawBitmap(wxBitmap(GetImage(id)), rect.x, rect.y, true);

}

void ToolbarButton::OnMouseLeftDown(wxMouseEvent& evt) {

m\_pressed = true;

if (!this->HasCapture()) {

this->CaptureMouse();

}

this->Refresh();

evt.Skip();

}

void ToolbarButton::OnMouseLeftUp(wxMouseEvent& evt) {

m\_pressed = false;

if (this->HasCapture()) {

this->ReleaseMouse();

}

PostEvent();

this->Refresh();

evt.Skip();

}

void ToolbarButton::OnMouseEnter(wxMouseEvent& evt) {

this->Refresh();

evt.Skip();

}

void ToolbarButton::OnMouseLeave(wxMouseEvent& evt) {

this->Refresh();

evt.Skip();

}

为了处理好pressed这个状态，加了一个额外的bool成员变量m\_pressed，在mouse left down时capture mouse可以保证之后的mouse left up一定会被收到，所以这里的m\_pressed便有了保证。此外，在mouse left up时调用的PostEvent尚未实现，在此需要抛出一个事件，但是具体抛什么事件，有待决定。

在wxWidgets里（或者说一般的系统里，因为wxWidgets只是做了封装而已），toolbar是一个control，toolbar上的tool可以是一般的button，也可以是check box或radio button，但是怎么创建这些tool却是透明的，没有办法定制。Toolbar本质上只是menu的另外一种形式，所以连事件也是共用的。我们写的这个ToolbarButton最终会放在一个panel或dialog上，跟系统的toolbar不一样，我们只是在模拟。

**自定义事件**

自定义控件时常犯的一个错误是，顺便也自定义了事件。虽然我们自定义的ToolbarButton跟系统缺省的相比差别很大，但它们的行为是相似的——点击后抛出一个事件，并且概念上也是一致的——都是tool bar相关，所以完全没必要去自定义事件——从wxCommandEvent继承一个子类，wxCommandEvent就已经符合我们的需求。最终PostEvent函数的实现如下：

void ToolbarButton::PostEvent() {

wxCommandEvent evt(wxEVT\_COMMAND\_TOOL\_CLICKED, this->GetId());

evt.SetEventObject(this);

this->GetParent()->AddPendingEvent(evt);

}

这里使用了预定义的事件类型（event type）wxEVT\_COMMAND\_TOOL\_CLICKED。事件类型在wxWidgets里实际上是个整形的值，相当廉价。所以如果非要自定义事件，通常也只是需要自定义事件类型，而不是事件类。如下代码给wxCommandEvent定义了一个新的事件类型：

BEGIN\_DECLARE\_EVENT\_TYPES()

DECLARE\_EVENT\_TYPE(wxEVT\_COMMAND\_TOOLBAR\_BUTTON\_CLICKED, 801)

END\_DECLARE\_EVENT\_TYPES()

#define EVT\_COMMAND\_TOOLBAR\_BUTTON\_CLICKED(id, func)\

DECLARE\_EVENT\_TABLE\_ENTRY(wxEVT\_COMMAND\_TOOLBAR\_BUTTON\_CLICKED, id, -1, wxCommandEventHandler(func), (wxObject\*)NULL),

DEFINE\_EVENT\_TYPE(wxEVT\_COMMAND\_TOOLBAR\_BUTTON\_CLICKED)

**UPDATE UI事件**

Toolbar tool或menu item的状态（是否disabled/checked）通常都会由程序的状态来决定。对于ToolbarButton来说，需要在normal和disabled两种状态间进行切换。比如一个undo button，会根据undo列表是否为空来决定自己是否应该处于disabled状态。这种逻辑混杂着界面的琐事，幸亏有update UI事件才得以化繁为简。如下示例把undo button放在MyToolbar 里，然后绑定update UI事件到OnUpdateUIUndo，后者会将是否可以undo的信息设置给wxUpdateUIEvent，至于undo button，看上去则什么都不用操心。

BEGIN\_EVENT\_TABLE(MyToolbar, wxDialog)

EVT\_UPDATE\_UI (ID\_TOOL\_UNDO, MyToolbar::OnUpdateUIUndo)

END\_EVENT\_TABLE()

void MyToolbar::OnUpdateUIUndo(wxUpdateUIEvent& event) {

event.Enable(get\_undoable());

}

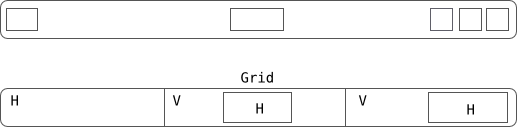
实际上wxWindow有个虚函数叫DoUpdateWindowUI，上面代码里那句Enable会导致undo button的DoUpdateWindowUI函数的调用，ToolbarButton并没有覆写DoUpdateWindowUI，缺省的行为即已满足。

**布局 LAYOUT**

控件创建好了，要放到panel或dialog里，并且排列上要整齐美观，符合用户的习惯。很多Windows上的软件在这一点上做得很差，因为MFC里没有像GTK+或Java Swing里layout那样的东西。早些年用Win32写GUI，在RC文件里要调一个button的位置实在是麻烦的事，用VC的拖拽界面来做也是一样，控件的位置都是死的。后来在某些软件的代码里看到有人给MFC写了个类似的东西叫ResizableLib，可以在sourceforge找到。

在wxWidgets里layout相关的问题由sizer来处理，sizer这个名字有点莫名其妙，不如layout形象。Sizer主要分box sizer和grid sizer，box sizer又分为horizontal和vertical，分别按横向和纵向排列窗口，而grid sizer一般用于类似表格的窗口排列，现实中也常常将box和grid两种sizer组合起来使用。

下图是一个比较复杂的例子，理解了这个例子基本也就没有大的问题了。这个图的上半部分表示想要的layout：一个panel分左中右三个区域，左边居左放一个控件，中间居中放一个控件，右边居右放三个控件；然后下半部分给出了sizer的用法：一个1行3列的grid sizer平分这个panel，在左边的grid里放一个horizontal sizer，在中间的grid里放一个vertical sizer并居中嵌套一个horizontal sizer，在右边的grid里放一个vertical sizer并居右嵌套一个horizontal sizer。



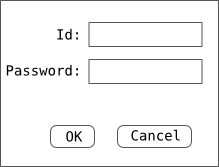
这个例子不用grid sizer也能做，但是要更麻烦一些，不做赘述。

**BEST SIZE**

使用sizer时，不需要指定被layout（非顶层）窗口的大小，甚至在创建这些窗口时也不需要指定大小，窗口的大小应该是灵活的，顶层窗口resize后，子窗口自然也要跟着进行调整，所以指定大小一般是没有意义的。

从layout的角度来看，非顶层窗口的大小可以分为两类：以自身内容决定大小，和以父窗口大小决定大小。以自身内容决定大小的窗口应该覆写DoGetBestSize函数，通过它来返回自己的best size，有了best size的窗口不应该被sizer拉伸（stretch），否则这个best size就等于白设了。以父窗口大小决定大小的窗口不需要覆写DoGetBestSize函数，它们一般会被sizer拉伸。无论是best size还是sizer的拉伸，都可以只是横向或只是纵向，因为有些窗口高度由自身内容决定宽度却由父窗口大小决定，或者反之。

下图在一个panel里放了两个text control和两个button，两个text control在横向上被sizer拉伸了，这样可以让它们大小一致保持对齐，两个button则根据自己的label来决定大小，并且没有被sizer拉伸。



也许让这两个button大小一致会更好些，其实像button这样的控件应该设置最小宽度，通过SetMinSize设置最小宽度后，sizer会优先考虑最小宽度，这样两个button的大小不至于差别太大。

就ToobarButton而言，它的大小应该由其自身内容（图片）来决定，所以给它加上DoGetBestSize：

wxSize ToolbarButton::DoGetBestSize() const {

const wxImage& normal = GetImage(Image\_Normal);

if (!normal.IsOk()) {

return wxDefaultSize;

}

return wxSize(normal.GetWidth(), normal.GetHeight());

}

至此，自定义控件ToolbarButton就算告一段落了。这篇指南也该结束了。可以看出，除了知识性的内容，本文也试着做了一些经验性的论断。这些论断可能微不足道，也难免疏漏，但如果能引起你的思考，目的便已达到。