通过XML创建界面---对象的动态创建以及属性的设置

为了界面可配置化和换肤，需要界面元素可以根据XML动态创建和设置属性。

在BKLib中，CBkObject类就完成了这样的功能，主要负责类的创建和属性的设置。因为对象都是从XML动态创建的，动态的创建是一个类最基本的属性，所以其他类都从CBkObject派生。

来看看这个类的四个方法：

BOOL IsClass(LPCSTR lpszName)：判断是不是这个类的对象。纯虚方法，也就是从CBkObject继承来的类都要实现这个方法才行，同时，这个方法在不同的类上面会有不同的表现。所以上层定义接口，下层提供实现。这个方法可以在运行时检测类实例的实际类型。

LPCSTR GetObjectClass()：同上一个方法，用于获取类的名字。

BOOL Load(TiXmlElement\* pXmlElem)：从XML中获取属性并将属性设置到对象中，在基类中仅仅是将一个XML元素的属性设置到对象中。当然，如果子类对象有更复杂的实现，比如一个对象对应的XML元素还有子节点，就需要Load其子节点，这些都可以在子类中通过覆盖父类方法来实现。

HRESULT SetAttribute(CStringA strAttribName, CStringA strValue, BOOL bLoading)：设置属性的方法，CBkObject是纯虚类，在XML中不会有对应的节点，自然也没有相应的属性，所以其实现仅仅放回了一个E\_FAIL，没有其他操作。

接着我们就看到了在bkobject.h里面一大堆的宏定义：

宏定义一般是为了简洁，而这些宏的用途多为子类使用。

BKOBJ\_DECLARE\_CLASS\_NAME：获取类名，判断是否是某个类的对象，还有CheckAndNew，用于动态创建

以下的宏定义主要用于属性的设置和映射(XML节点属性和对象属性的对应)

BKWIN\_DECLARE\_ATTRIBUTES\_BEGIN

BKWIN\_DECLARE\_ATTRIBUTES\_END

BKWIN\_CHAIN\_ATTRIBUTE

BKWIN\_CUSTOM\_ATTRIBUTE

BKWIN\_INT\_ATTRIBUTE

BKWIN\_UINT\_ATTRIBUTE

BKWIN\_DWORD\_ATTRIBUTE

BKWIN\_STRING\_ATTRIBUTE

BKWIN\_TSTRING\_ATTRIBUTE

BKWIN\_HEX\_ATTRIBUTE

BKWIN\_COLOR\_ATTRIBUTE

BKWIN\_FONT\_ATTRIBUTE

BKWIN\_ENUM\_ATTRIBUTE

BKWIN\_ENUM\_VALUE

BKWIN\_ENUM\_END

BKWIN\_STYLE\_ATTRIBUTE

BKWIN\_SKIN\_ATTRIBUTE

现在我们看一个例子，继承自CBkObject的控件对象CBkProgress是如何完成从XML动态创建的。

首先，在类的定义中包含宏BKOBJ\_DECLARE\_CLASS\_NAME(CBkProgress, "progress")，将宏展开如下：

public:

 static CBkProgress\* CheckAndNew(LPCSTR lpszName)

 {

 if (strcmp(GetClassName(), lpszName) == 0)

 return new CBkProgress;

 else

 return NULL;

 } //通过传入名称创建对应的类，在解析XML中按照节点名字创建对应类的实例

 static LPCSTR GetClassName()

 {

 return “progress”;

 }

 virtual LPCSTR GetObjectClass()

 {

 return “progress”;

 } //覆盖父类方法，返回类实例对应的XML节点名字

 virtual BOOL IsClass(LPCSTR lpszName)

 {

 return strcmp(GetClassName(), lpszName) == 0;

 } //覆盖父类方法，根据XML节点名字检查类实例是否是此节点

另外，包含设置节点属性的宏，如下

 BKWIN\_DECLARE\_ATTRIBUTES\_BEGIN()

 BKWIN\_SKIN\_ATTRIBUTE("bgskin", m\_pSkinBg, TRUE)

 BKWIN\_DWORD\_ATTRIBUTE("min", m\_dwMinValue, FALSE)

 BKWIN\_UINT\_ATTRIBUTE("showpercent", m\_bShowPercent, FALSE)

 BKWIN\_DECLARE\_ATTRIBUTES\_END()

将宏展开如下：

public:

 virtual HRESULT SetAttribute(

 CStringA strAttribName,

 CStringA strValue,

 BOOL bLoading) //添加SetAttribute方法，在Load中循环调用设置属性

 {

 HRESULT hRet = \_\_super::SetAttribute( //首先设置父类定义的属性

 strAttribName,

 strValue,

 bLoading

 );

 if (SUCCEEDED(hRet))

 return hRet;

 if ("bgskin" == strAttribName) //属性名称是strAttribName

 {

 m\_pSkinBg = BkSkin::GetSkin(strValue); //属性的值是strValue

 hRet = TRUE ? S\_OK : S\_FALSE; //是否全部重绘

 }

 else

 if ("min" == strAttribName)

 {

 m\_dwMinValue = (DWORD)::StrToIntA(strValue);

 hRet = FALSE ? S\_OK : S\_FALSE;

 }

 else

 if ("showpercent" == strAttribName)

 {

 m\_bShowPercent = (UINT)::StrToIntA(strValue);

 hRet = FALSE ? S\_OK : S\_FALSE;

 }

 else

 return E\_FAIL;

 return hRet;

 }

现在我们来看看CheckAndNew和SetAttribute这两个方法是如何被调用的，看调用栈：



1.在实窗口的Create方法中(DoModal中调用Create)调用实窗口的Load和SetXml方法装载XML，在SetXML方法中查找XML中存在的"header"、"body"、"footer"节点调用各自的Load方法并设置相应属性。这三个节点开始就是虚窗口了，调用其Load方法就进入了虚窗口的创建体系。

2.在这三者的调用中会调用CBkPanel::Load方法

 virtual BOOL Load(TiXmlElement\* pTiXmlElem)

 {

 if (!CBkWindow::Load(pTiXmlElem)) //调用父类load方法，主要进行自身属性设置

 return FALSE;

 return LoadChilds(pTiXmlElem->FirstChildElement()); //load子节点

}

在CBkPanel::LoadChilds方法中，顺序调用每个子节点的创建方法并调用Load方法。

3. 在CBkPanel::LoadChilds方法中进行了如下调用：

CBkWindow \*pNewChildWindow = \_CreateBkWindowByName(pXmlChild->Value());

在\_CreateBkWindowByName函数根据从XML解析出的节点名称调用

pNewWindow = CBkDialog::CheckAndNew(lpszName); // CBkDialog为需要动态创建类的名称

创建出对应节点对象。安装我们展开的CheckAndNew方法，如果节点名称相同，创建类对象并返回，否则返回空。

至此，按照XML节点名称动态创建类对象的过程就完成了。

**统一的资源管理**

为了对界面资源进行管理，同时也为了方便替换，需要对使用的资源进行统一的管理。

在BKLib中，资源管理主要由以下几种：

BkBmpPool：HBITMAP资源池，用于统一管理HBITMAP，单例。

BkFontPool：FONT资源池，用于统一管理FONT，单例。

BkPngPool：PNG图片资源池，用于统一管理PNG图片，单例，使用GDI+。

BkString：CString资源池，用于统一管理String，单例，从XML中获取。

BkColor：HLS&RGB颜色处理工具类

CBkImage：图像处理类

BkResManager：资源处理器，单例，用于获取资源

BkSkin：Skin资源池，单例，统一管理Skin (CBkSkinBase)，从XML中获取

BkStyle：Style资源池，单例，统一管理Style，从XML中获取

在窗口创建之前需要加载相应的资源

 BkString::Load(IDR\_BK\_STRING\_DEF); // 加载字符串

 BkFontPool::SetDefaultFont(BkString::Get(IDS\_APP\_FONT), -12); // 设置字体

 BkSkin::LoadSkins(IDR\_BK\_SKIN\_DEF); // 加载皮肤

 BkStyle::LoadStyles(IDR\_BK\_STYLE\_DEF); // 加载风格

 //其中输入参数为XML文件的名称

在程序关闭后应释放对应资源。

使用时只要按照ID在资源池中查找对应资源即可，如

 if ("bgskin" == strAttribName) //属性名称是strAttribName

 {

 m\_pSkinBg = BkSkin::GetSkin(strValue); //属性的值是strValue

 hRet = TRUE ? S\_OK : S\_FALSE; //是否全部重绘

 }

根据ID(strValue)就可在BkSkin中获取相应的Skin。

**真实窗口的封装以及实窗口到虚窗口的转化**

所谓的DUI库，windowless都是在一个窗口体系内虚拟出来虚窗口概念，并且通过接管界面布局、消息传递和分发以及界面绘制来完成更优秀的界面效果。不过这些的根基却又都要落到真实的窗口上，所以在界面库中需要对真实窗口进行封装，并将真实窗口纳入到我们创建的控件体系当中，并在这个过程中完成windows消息的传递，鼠标键盘事件的分发处理，实窗口上的虚窗口的布局排版和绘制操作。

首先我们看看BKLib中的实窗口体系：



BKLib中并没有自己对windows窗口进行封装，而是使用了WTL的CWindowImpl类，这个类对于windows窗口，以及消息分发、窗口属性设置等进行了封装。对于windows消息分发中有一个重要的地方就是注册的窗口过程是按照窗口句柄进行处理，但是在我们的程序中是窗口类的一个成员方法，如何将窗口句柄和C++类实例之间进行映射就成了一个重要的话题，WTL中主要使用trunk技术，而MFC则使用链表进行查找，具体的细节大家可以在网上查询，这里就不赘述了。

这里我们看一下上面类图中其他几个类的作用：

CBkDialog:无窗口控件，它是我们虚窗口体系的一部分

CBkViewImpl:这个算是一个接口类，不过它提供了我们需要包含虚窗口体系的实窗口所必须具备的一些方法(这句话真绕)

CBkDialogViewImpl：它是一个实窗口了，因为他从封装了实窗口的 CWindowImpl类继承而来，同时他也具有包含虚窗口所必须实现的方法(从CBkViewImpl继承而来)，同时他还聚合了我们虚窗口体系的一部分，也就是CBkDialog，但是我们实际使用的类并不是它。

CBkDialogView：这个类在我们的程序中会创建一个真正的窗口的，从CBkDialogViewImpl继承而来，具有了所有的能力，算是这里的中坚力量了。

CBkDialogImpl：它也是一个实窗口，我们自定义窗口就是从此继承创建，同时它还聚合了上面的包含容器窗口(CBkDialogView)，所以在BKLib中创建的窗口是有两层窗口的，上面的用于承载控件体系，下面的则是我们需要自定义的窗口。为什么要创建两个窗口我们之后再研究，这里就先不解释了。

了解了上面的窗口体系后，我们就来看看在一个窗口创建过程中，各种消息、排版、绘制是如何从我们封装的实窗口想我们的虚窗口——控件体系上转化的吧。

创建过程：



布局过程(WM\_SIZE消息处理)



绘制过程(WM\_PAINT消息处理)



在实窗口到虚窗口的转化过程中，主要在于CBkDialogViewImpl类中包含三个成员变量：m\_bkHeader,m\_bkBody,m\_bkFooter，这三者的类型均是CBkDialog，也就是我们控件体系的一部分。BkLib中将一个实窗口划分为三个虚窗口：head、body、foot，对于界面的创建、布局、绘制等操作也由这三者传递到虚窗口体系中，并通过递归调用来对所有成员进行处理。

**如何创建一个模态对话框**

我们创建的窗口类从CBkDialogImpl继承而来，这个窗口就是一个模态的窗口，我们需要调用其DoModal方法，但是在界面库里面是如何实现的一个模态的对话框呢。

核心就在这个类中的\_ModalMessageLoop方法，我们来研究一下。

 void \_ModalMessageLoop()

 {

 BOOL bRet;

 MSG msg;

 for(;;)

 {

 if (::PeekMessage(&msg, NULL, 0, 0, PM\_NOREMOVE))

 {

 if (WM\_QUIT == msg.message)

 break;

 }

 if (m\_bExitModalLoop || NULL == m\_hWnd || !::IsWindow(m\_hWnd))

 break;

 bRet = ::GetMessage(&msg, NULL, 0, 0);

 if (bRet == -1)

 {

 continue; // error, don't process

 }

 else if (!bRet)

 {

 ATLTRACE(L"Why Receive WM\_QUIT here?\r\n");

 break; // WM\_QUIT, exit message loop

 }

 ::TranslateMessage(&msg);

 ::DispatchMessage(&msg);

 }

}

在这个函数里面建立了一个消息处理循环。

首先进行WM\_QUIT消息的检测，并且采用PM\_NOREMOVE的方式，如果获得了这个消息，那么退出消息循环，在之后的DoModal函数中模态窗口就被销毁了。

然后检测bExitModalLoop标志，如果这个标志位为TRUE的话那么也退出处理。我们来看一下退出模态的函数：

 void EndDialog(UINT uRetCode)

 {

 m\_uRetCode = uRetCode;

 m\_bExitModalLoop = TRUE;

 // DestroyWindow里面直接Send了WM\_DESTROY，所以不会跑到DoModal的消息循环里，所以有了下面那行代码

 // DestroyWindow();

 // 这句非常重要，可以让DoModal消息循环再跑一次，防止卡死在GetMessage，泪奔

 ::PostThreadMessage(::GetCurrentThreadId(), WM\_NULL, 0, 0);

 }

可见就是设置m\_bExitModalLoop这个标志位，使窗口退出。

 之后就是GetMessage，TranslateMessage和DispatchMessage了，就完成了常见的消息处理。当我们创建窗口并运行在ModalMessageLoop函数当中时就形成了模态窗口的效果，也就是接管了消息的分发和处理，其他的窗口就被模住了。

**无窗口模式---逻辑树结构的建立**

对于无窗口的模式，各种控件之间的关系需要我们自己来维护，因为我们要进行消息传递，排版布局，创建等操作时都需要沿着各种包含关系来逐个调用，这样对于窗口中的控件就形成了一个逻辑上的树形结构。我们看一下CBkPanel类，这个类的名字就表示它是一个有包含功能的类，可以有自己的子节点。这个类有一个成员CAtlList<CBkWindow \*> m\_lstWndChild，也就是每个CBkPanel类都有一个链表，在这个链表中存储了它的子节点控件。

那么这个链表是何时被填充的呢，我们看一下LoadChilds方法，就是在这个函数中进行的填充操作，从前面我们知道，这个函数是在用xml 初始化界面是调用的。其中对于当前节点下的所有xml子节点进行下面的处理：

CBkWindow \*pNewChildWindow = \_CreateBkWindowByName(pXmlChild->Value()); //创建控件

if (!pNewChildWindow)

continue; //创建失败，不进行后续设置

pNewChildWindow->SetParent(m\_hBkWnd); //设置父节点HBKWND

pNewChildWindow->SetContainer(m\_hWndContainer); //设置容器窗口的HWND

pNewChildWindow->Load(pXmlChild); //Load这个节点的子节点

m\_lstWndChild.AddTail(pNewChildWindow); **//将自己加入到父节点的链表中**

在CBkPanel的OnDestroy函数中进行子节点的销毁操作

 POSITION pos = m\_lstWndChild.GetHeadPosition();

 while (pos != NULL)

 {

 CBkWindow \*pBkWndChild = m\_lstWndChild.GetNext(pos);

 pBkWndChild->BkSendMessage(WM\_DESTROY);

 delete pBkWndChild;

 }

 m\_lstWndChild.RemoveAll();

之后进行消息传递，绘制，排版操作都可以使用这个树来进行处理，循环调用所有的节点。如在OnPaint函数中：

 POSITION pos = m\_lstWndChild.GetHeadPosition();

 BOOL bDisabled = IsDisabled(), bIsChildDisabled = FALSE;

 while (pos != NULL)

 {

 //……此处略去

 pBkWndChild->BkSendMessage(WM\_PAINT, (WPARAM)(HDC)dc);

 //……此处略去

}

可见也是循环向每个子节点发送WM\_PAINT消息。

同时，在界面库中，还使用了BkWnds来记录所有的控件，以便可以便捷地进行获取操作：

CBkWindow类有HBKWND类型的成员变量m\_hBkWnd，这是一个虚拟的窗口句柄。

BkWnds是一个控件池，创建的控件会在这里注册，并通过一个HBKWND类型变量标示，以后就可以通过控件的m\_hBkWnd来获取这个控件。

**消息和事件的传递、分发、相应**

既然没有真实的窗口，那么也就不能使用windows的根据句柄来分发消息的方式了，我们需要创建自己的消息和事件分发体系。主要应该包括这几个方面：

1. 接收真实窗口的消息，并将其转化虚窗口体系中的处理
2. 虚窗口体系内有一套独立的消息分发机制，可以讲系统消息发至该接收的控件
3. 虚窗口需要能够抛出事件的能力，因为虚窗口之间也需要有相互的通知和相应的能力，那么就需要虚窗口可以对于指定窗口抛出事件
4. 对于3中所抛出的事件，可以传递到指定的控件，并且控件内部应该有一套指定额相应体系

在CBkDialogViewImpl中使用WTL的消息分发方法用于系统消息的分发：

 BEGIN\_MSG\_MAP\_EX(CBkDialogViewImpl)

 MESSAGE\_RANGE\_HANDLER\_EX(WM\_MOUSEFIRST, WM\_MOUSELAST, OnToolTipEvent)

 MSG\_WM\_SIZE(OnSize)

 MSG\_WM\_PRINT(OnPrint)

 MSG\_WM\_PAINT(OnPaint)

 //……从略

 NOTIFY\_CODE\_HANDLER\_EX(BKINM\_INVALIDATERECT, OnBKINMInvalidateRect)

 REFLECT\_NOTIFY\_CODE(NM\_CUSTOMDRAW)

 MESSAGE\_HANDLER\_EX(WM\_NOTIFY, OnChildNotify)

 MESSAGE\_HANDLER\_EX(WM\_COMMAND, OnChildNotify)

 MESSAGE\_HANDLER\_EX(WM\_VSCROLL, OnChildNotify)

 MESSAGE\_HANDLER\_EX(WM\_HSCROLL, OnChildNotify)

 REFLECT\_NOTIFICATIONS\_EX()

END\_MSG\_MAP()

下面的是对于WM\_SIZE消息的分发过程：



在CBkWindow类中创建了用于发送通知的函数：

 // Send a message to BkWindow

 LRESULT BkSendMessage(UINT Msg, WPARAM wParam = 0, LPARAM lParam = 0)

 {

 LRESULT lResult = 0;

 if ( Msg < WM\_USER

 && Msg != WM\_DESTROY

 && Msg != WM\_CLOSE

 )

 {

 TestMainThread();

 }

 SetMsgHandled(FALSE);

 ProcessWindowMessage(NULL, Msg, wParam, lParam, lResult);

 return lResult;

 }

这个函数很像windows的消息发送函数SendMessage，

LRESULT SendMessage(HWND hWnd, UINT Msg, WPARAM wParam, LPARAM IParam)

需要向那个对象发送消息，那么首先需要获取这个对象，然后调用这个对象的BkSendMessage函数，用消息类型和附属参数填充，就可以完成事件通知的过程了。

在这个函数中调用了ProcessWindowMessage方法，但是我们并没有看见这个函数的实现，我们来找一下哈~~



我们点击调用栈里面的ProcessWindowMessage方法，程序转到了CBkWindow类的这块：

 BKWIN\_BEGIN\_MSG\_MAP()

 MSG\_WM\_CREATE(OnCreate)

 MSG\_WM\_PAINT(OnPaint)

 MSG\_WM\_DESTROY(OnDestroy)

 MSG\_WM\_WINDOWPOSCHANGED(OnWindowPosChanged)

 MSG\_WM\_NCCALCSIZE(OnNcCalcSize)

 MSG\_WM\_SHOWWINDOW(OnShowWindow)

BKWIN\_END\_MSG\_MAP\_BASE()

难道就是这串宏实现了实现了通知的相应机制，我们来分析一下这个几个宏的实现。

// BkWindow Message Map Define

// Use WTL Message Map Ex (include atlcrack.h)

#define BKWIN\_BEGIN\_MSG\_MAP() \

protected: \

 virtual BOOL ProcessWindowMessage( \

 HWND hWnd, UINT uMsg, WPARAM wParam, \

 LPARAM lParam, LRESULT& lResult) \

 { \

#define BKWIN\_END\_MSG\_MAP() \

 if (!IsMsgHandled()) \

 return \_\_super::ProcessWindowMessage( \

 hWnd, uMsg, wParam, lParam, lResult); \

 return TRUE; \

 } \

#define BKWIN\_END\_MSG\_MAP\_BASE() \

 return TRUE; \

 } \

上面的是自定义的，而其他则是使用的WTL的宏，如：

// int OnCreate(LPCREATESTRUCT lpCreateStruct)

#define MSG\_WM\_CREATE(func) \

 if (uMsg == WM\_CREATE) \

 { \

 SetMsgHandled(TRUE); \

 lResult = (LRESULT)func((LPCREATESTRUCT)lParam); \

 if(IsMsgHandled()) \

 return TRUE; \

 }

我们将宏展开看一下就是：

protected:

 virtual BOOL ProcessWindowMessage(

 HWND hWnd, UINT uMsg, WPARAM wParam,

 LPARAM lParam, LRESULT& lResult)

 {

 if (uMsg == WM\_CREATE)

 {

 SetMsgHandled(TRUE);

 lResult = (LRESULT) OnCreate ((LPCREATESTRUCT)lParam);

 if(IsMsgHandled())

 return TRUE;

 }

 //......此处略去其他分发函数

 return TRUE;

 }

可见就是给每个虚窗口类增加了一个函数ProcessWindowMessage，而这个函数的作用就是根据发送通知的不同来选择处理的函数，也就是BkSendMessage🡪ProcessWindowMessage🡪根据消息的不同选择不同的处理函数，并且这个函数和消息我们可以自己进行搭配的，可能是为了简便使用了系统消息宏，其实我们也可以自己进行定义的。比如定义一个消息为BKM\_CREATE，那么定义一个宏：

#define MSG\_BKM\_CREATE(func) \

 if (uMsg == BKM\_CREATE) \

 { \

 SetMsgHandled(TRUE); \

 lResult = (LRESULT)func((LPCREATESTRUCT)lParam); \

 if(IsMsgHandled()) \

 return TRUE; \

 }

就可以自己来定义通知的类型了。

对于CBkWindow，因为它是所有虚窗口的基类，所以消息处理到此为止，ProcessWindowMessage函数最后什么都没做，使用了BKWIN\_END\_MSG\_MAP\_BASE宏。

在CBkWindow子类中，我们将BKWIN\_END\_MSG\_MAP\_BASE替换为BKWIN\_END\_MSG\_MAP，那么宏展开了就是：

protected:

 virtual BOOL ProcessWindowMessage(

 HWND hWnd, UINT uMsg, WPARAM wParam,

 LPARAM lParam, LRESULT& lResult)

 {

 if (uMsg == WM\_CREATE)

 {

 SetMsgHandled(TRUE);

 lResult = (LRESULT)func((LPCREATESTRUCT)lParam);

 if(IsMsgHandled())

 return TRUE;

 }

 //......此处略去其他分发函数

 if (!IsMsgHandled())

 return \_\_super::ProcessWindowMessage(

 hWnd, uMsg, wParam, lParam, lResult);

 return TRUE;

 }

\_\_super::ProcessWindowMessage完整的链式消息传递啊~~，对于子类处理之后的消息还行传递给父类进行处理就可以这么搞了~~~

**如何进行排版**

前面我们分析过对于WM\_SIZE消息的处理过程，CBkDialogViewImpl窗口类在处理WM\_SIZE消息时，调用了自身的\_RepositionItems方法，在这个方法内对窗口上的虚拟控件进行处理，现在我们来详细看一下控件体系内是如何进行排版操作的。

在\_RepositionItems方法中定义了4个CRect变量，就是rcClient, rcHeader, rcFooter和rcBody，其中rcHeader, rcFooter和rcBody分别对应m\_bkHeader，m\_bkFooter和m\_bkBody的位置，rcClient是rcHeader，m\_bkFooter和m\_bkBody之和。

在控件中哪些变量表示控件的位置信息呢：

在CBkWindow中，用m\_rcWindow这个CRect类型的变量来表示每个控件的基本位置信息(控件的长宽以及左上角相对于实窗口的pos)，用m\_dlgpos这个BKDLG\_POSITION类型的变量来表示 从Xml获取的控件位置信息(相对于父控件的距离)。

CBkWindow类中的BKDLG\_POSITION类型包含了控件上下左右的边界位置，而且可以表示控件边界相对于父控件边界是左对齐，右对齐还是居中对齐，以及具体的偏移量。

而在CBkWindow类中也设置了一系列的枚举值：

 enum {

 // Specify by "width" attribute

 SizeX\_Mask = 0x0007UL,

 SizeX\_Specify = 0x0001UL, // width > 0

 SizeX\_FitContent = 0x0002UL, // width <= 0

 SizeX\_FitParent = 0x0004UL, // width = "full" default

 // Specify by "height" attribute

 SizeY\_Mask = 0x0070UL,

 SizeY\_Specify = 0x0010UL, // height > 0

 SizeY\_FitContent = 0x0020UL, // height <= 0 default

 SizeY\_FitParent = 0x0040UL, // height = "full" default

 // Specify by "float" attribute

 Position\_Mask = 0x0300UL,

 Position\_Relative = 0x0100UL, // float = 0 default

 Position\_Float = 0x0200UL, // float = 1

 // Specify by "valign" and "align" attribute, only using at float = 1 or panel control (Vert-Align)

 Align\_Mask = 0xF000UL,

 VAlign\_Top = 0x0000UL, // valign = top

 VAlign\_Middle = 0x1000UL, // valign = middle

 VAlign\_Bottom = 0x2000UL, // valign = bottom

 Align\_Left = 0x0000UL, // align = left

 Align\_Center = 0x4000UL, // align = center

 Align\_Right = 0x8000UL, // align = right

};

用来表示控件是按照内容填充，按照父控件填充，还是按照固定的值进行填充，也可以定义对齐的方式，这些描述可以比上面的BKDLG\_POSITION类型更加丰富，对于在xml中描述控件位置也更加灵活。

控件中用于处理位置信息的方法有哪些呢：

CBkWindow：
OnWindowPosChanged：处理WM\_WINDOWPOSCHANGED消息，发送WM\_NCCALCSIZE消息获取自身长宽，通过传入的Pos和前面取得的长宽设置m\_rcWindow变量

OnNcCalcSize：处理WM\_NCCALCSIZE消息，根据父控件的长宽信息以Xml文件中自身位置信息的描述来计算自身对于父控件的位置信息。

CBkPanel：

OnWindowPosChanged：处理WM\_WINDOWPOSCHANGED消息，调用CBkWindow类的OnWindowPosChanged方法，调用\_ComposingPanel方法。

CBkDialog：

OnWindowPosChanged：处理WM\_WINDOWPOSCHANGED消息，调用CBkWindow类的OnWindowPosChanged方法，调用\_RepositionChilds方法(依次调用每个子节点的RepositionChild方法)。

所以对控件发送WM\_WINDOWPOSCHANGED消息时需要传入WINDOWPOS结构体，其中x和y为控件的位置，而cx和cy为父窗口的长宽，控件根据cx和cy以及xml中的描述来计算自身的长宽。

现在我们来看一下控件位置调整的过程:



CBkWindow类负责自身的计算工作，所有控件的自身计算工作都是由调用父类CBkWindow的OnWindowPosChanged方法来进行的，在这个方法中通过xml描述和传入的父控件长宽计算自身大小，同时根据传入的pos设置控件的m\_rcWindow。CBkDialog类计算自身，并计算其子节点的大小，设置子节点的pos并向其发送WM\_WM\_WINDOWPOSCHANGED消息，令其计算自身。

**如何进行绘制**

首先我们来看一下CBkDialogViewImpl类的绘制过程：

CBkDialogViewImpl含有一个CBkImage类成员m\_imgMem用于双缓冲绘制，在WM\_SIZE消息处理函数中进行创建：

m\_imgMem.CreateBitmap(rcClient.Width(), rcClient.Height(), RGB(0, 0, 0));

CBkDialogViewImpl含有一个CRgn类成员m\_rgnInvalidate用于脏区域描述，在\_InvalidateRect方法内创建(绘制之前)：

m\_rgnInvalidate.CreateRectRgnIndirect(rcInvalidate);

我们来看看CBkDialogViewImpl类的OnPrint方法，在处理WM\_PAINT消息时会调用这个函数，首先获取一个兼容DC，并将双缓冲的图像渲染兼容DC：

 CDC dcMem;

 CDCHandle dcMemHandle;

 HDC hDCDesktop = ::GetDC(NULL);

 dcMem.CreateCompatibleDC(hDCDesktop);

 ::ReleaseDC(NULL, hDCDesktop);

 HBITMAP hbmpOld = dcMem.SelectBitmap(m\_imgMem);

然后设置兼容DC的脏区域：

 dcMem.SelectClipRgn(m\_rgnInvalidate);

之后在这个DC上面进行绘制操作，

 if (m\_bHasHeader)

 m\_bkHeader.RedrawRegion(dcMemHandle, m\_rgnInvalidate);

 if (m\_bHasBody)

 m\_bkBody.RedrawRegion(dcMemHandle, m\_rgnInvalidate);

 if (m\_bHasFooter)

 m\_bkFooter.RedrawRegion(dcMemHandle, m\_rgnInvalidate);

最后将这个双缓冲图像整体绘制到窗口DC上面：

 m\_imgMem.Draw(dc, 0, 0);

CBkDialogViewImpl的方法\_InvalidateRect用于绘制指定区域，合并现有脏区域，重绘指定区域。\_InvalidateControl方法用于重绘指定控件，即获取此控件位置并重绘此位置

可见现在绘制转入了控件层的RedrawRegion方法。

控件中用于处理绘制的方法有哪些呢：

CBkWindow：

RedrawRegion(只有控件在重绘区域内时才进行)：调用DrawBkgnd，发送WM\_PAINT消息

DrawBkgnd：在兼容DC中绘制背景，如果Skin存在，用Skin绘制，否则用Style中颜色绘制。

OnPaint：只有绘制文字的过程，一般不被调用

CBkPanel：

RedrawRegion：调用父类CBkWindow的RedrawRegion方法，重绘自身背景，循环调用每个子节点的RedrawRegion方法。

OnPaint：原来有处理，现在被return掉了。

在CBkPanel 类的RedrawRegion方法中调用了BeforePaint和AfterPaint两个方法，BeforePaint用来设置BkMode、BkColor、Font和TextColor并保存原始信息，AfterPaint用于将这些环境重置回原值。

**渲染层的封装和隔离：BkSkin**

控件的绘制工作基本都封装到了Skin里面，如果控件对应的Skin存在的话，那么会按照Skin的描述进行绘制，在CBkWindow的DrawBkgnd方法中，使用Skin进行了绘制：

 CBkSkinBase\* pSkin = BkSkin::GetSkin(m\_style.m\_strSkinName);

 if (pSkin)

 {

 pSkin->Draw(dc, m\_rcWindow, m\_dwState);

 }

看一下Skin的继承关系：



对于CBkSkinBase类，只需要绘制的区域，绘制的状态以及绘制的DC就可以。CBkSkinBase是一个虚基类，从CBkSkinBase继承的子类需要实现Draw方法，在这个方法中实现具体的绘制操作。

同时CBkSkinBase类也提供了一下这些工具方法，用来辅助绘制操作：

HorzExtendDraw

FrameDraw

GradientFillRectV

GradientFillRectH

GradientFillRectV

GradientFillRectH

我们来看个CBkSkinButton类，当然主要看Draw方法：

virtual void Draw(CDCHandle dc, CRect rcDraw, DWORD dwState)

{

 CPen penFrame;

 CRect rcBg = rcDraw;

 dc.FillSolidRect(rcDraw, m\_crBg); //填充背景颜色

 rcBg.DeflateRect(2, 2);

 if (BkWndState\_Disable == (BkWndState\_Disable & dwState)) //如果Disable状态，不进行绘制

 {

 }

 else

 GradientFillRectV(

 dc, rcBg,

 IIF\_STATE3(dwState, m\_crBgUpNormal, m\_crBgUpHover, m\_crBgUpPush),

 IIF\_STATE3(dwState, m\_crBgDownNormal, m\_crBgDownHover, m\_crBgDownPush));

 penFrame.CreatePen( //创建画笔，用于边框绘制

 PS\_SOLID,

 1,

 m\_crBorder

 );

 HPEN hpenOld = dc.SelectPen(penFrame);

 HBRUSH hbshOld = NULL, hbshNull = (HBRUSH)::GetStockObject(NULL\_BRUSH); //空画刷

 hbshOld = dc.SelectBrush(hbshNull);

 dc.Rectangle(rcDraw); //绘制矩形边框

 dc.SelectBrush(hbshOld);

 dc.SelectPen(hpenOld);

}

在上面的绘制中使用了控件的状态，控件状态定义如下：

// State Define

enum {

 BkWndState\_Normal = 0x00000000UL,

 BkWndState\_Hover = 0x00000001UL,

 BkWndState\_PushDown = 0x00000002UL,

 BkWndState\_Check = 0x00000004UL,

 BkWndState\_Invisible = 0x00000008UL,

 BkWndState\_Disable = 0x00000010UL,

};

这个IIF\_STATE3是个宏定义，对于指定的状态，返回相应的背景，具体如下：

#define IIF\_STATE2(the\_state, normal\_value, hover\_value) \

 (((the\_state) & BkWndState\_Hover) ? (hover\_value) : (normal\_value))

#define IIF\_STATE3(the\_state, normal\_value, hover\_value, pushdown\_value) \

 (((the\_state) & BkWndState\_PushDown) ? (pushdown\_value) : IIF\_STATE2(the\_state, normal\_value, hover\_value))

#define IIF\_STATE4(the\_state, normal\_value, hover\_value, pushdown\_value, disable\_value) \

 (((the\_state) & BkWndState\_Disable) ? (disable\_value) : IIF\_STATE3(the\_state, normal\_value, hover\_value, pushdown\_value))

**鼠标消息、窗口状态的管理**

首先来看CBkDialogViewImpl类的鼠标消息处理方法

OnMouseLeave方法：

如果现在处于非跟踪的状态，那么调用\_TrackMouseEvent并设置m\_bTrackFlag；

之后获取现在鼠标所悬停的控件，通过调用header,body和footer的BkGetHWNDFromPoint方法来判断，因为这三者覆盖整个客户区域

如果当前悬停控件和记录的hover控件不一致，那么进行更新操作

OnMouseLeave方法：

设置不跟踪状态

如果hover控件存在，那么重置状态

如果pushdown控件存在，那么重置状态

OnLButtonDown方法：

查看是否是点击标题栏，如果是，调用最大化，最小化和恢复处理

在非标题栏点击情况下，设置capture状态

OnLButtonUp方法

ReleaseCapture，设置控件状态

OnLButtonDblClk方法

在双击标题栏情况下最大化或者恢复窗口

在类CBkWindow 中，使用m\_dwState变量来标示控件的状态，主要由以下几种状态：

// State Define

enum {

 BkWndState\_Normal = 0x00000000UL, //正常

 BkWndState\_Hover = 0x00000001UL, //悬停

 BkWndState\_PushDown = 0x00000002UL, //按下

 BkWndState\_Check = 0x00000004UL, //选中

 BkWndState\_Invisible = 0x00000008UL, //不可见

 BkWndState\_Disable = 0x00000010UL, //失活

};

主要使用ModifyState方法来改变控件的状态。