分为几个大部分：

1. 控件
2. 容器（本质也是控件）
3. UI构建解析器（XML解析）
4. 窗体管理器（消息循环，消息映射，消息处理，窗口管理等）
5. 渲染引擎

**DuiLib 消息循环剖析**

DuiLib的消息循环非常灵活，但不熟悉的可能会觉得非常混乱，不知道该如何下手。所以，我总结了下DuiLib的各种消息响应的方式，帮助大家理解DuiLib和加快开发速度。

其消息处理架构较为灵活，基本上在消息能过滤到的地方，都给出了扩展接口。

看了DuiLib入门教程后，对消息机制的处理有些模糊，为了屏蔽Esc按键，都花了大半天的时间。究其原因，是因为对DuiLib消息过滤不了解。  
你至少应该看过上面提及的那篇入门教程，看过一些DuiLib的代码，但可能没看懂，那么这篇文章会给你指点迷津。

Win32消息路由如下：

1. 消息产生。
2. 系统将消息排列到其应该排放的线程消息队列中。
3. 线程中的消息循环调用GetMessage（or PeekMessage）获取消息。
4. 传送消息TranslateMessage and DispatchMessage to 窗口过程（Windows procedure）。
5. 在窗口过程里进行消息处理

我们看到消息经过几个步骤，DuiLib架构可以让你在某些步骤间进行消息过滤。首先，第1、2和3步骤，DuiLib并不关心。DuiLib对消息处理集中在**CPaintManagerUI**类中（也就是上面提到的**窗体管理器**）。DuiLib在发送到窗口过程的前和后都进行了消息过滤。

DuiLib的消息渠，也就是所谓的消息循环在**CPaintManagerUI::MessageLoop()**或者**CWindowWnd::ShowModal()**中实现。俩套代码的核心基本一致，以MessageLoop为例：

void CPaintManagerUI::MessageLoop()

{

MSG msg = { 0 };

while( ::GetMessage(&msg, NULL, 0, 0) ) {

// CPaintManagerUI::TranslateMessage进行消息过滤

if( !CPaintManagerUI::TranslateMessage(&msg) ) {

::TranslateMessage(&msg);

try{

::DispatchMessage(&msg);

} catch(...) {

DUITRACE(\_T("EXCEPTION: %s(%d)\n"), \_\_FILET\_\_, \_\_LINE\_\_);

#ifdef \_DEBUG

throw "CPaintManagerUI::MessageLoop";

#endif

}

}

}

}

3和4之间，DuiLib调用**CPaintManagerUI::TranslateMessage**做了过滤，类似MFC的**PreTranlateMessage**。

想象一下，如果不使用这套消息循环代码，我们如何能做到在消息发送到窗口过程前进行常规过滤（Hook等拦截技术除外）？答案肯定是做不到。因为那段循环 代码你是无法控制的。CPaintManagerUI::TranslateMessage将无法被调用，所以，可以看到DuiLib中几乎所有的 demo在创建玩消息后，都调用了这俩个消息循环函数。下面是TranslateMessage代码：

bool CPaintManagerUI::TranslateMessage(const LPMSG pMsg)

{

// Pretranslate Message takes care of system-wide messages, such as

// tabbing and shortcut key-combos. We'll look for all messages for

// each window and any child control attached.

UINT uStyle = GetWindowStyle(pMsg->hwnd);

UINT uChildRes = uStyle & WS\_CHILD;

LRESULT lRes = 0;

if (uChildRes != 0) // 判断子窗口还是父窗口

{

HWND hWndParent = ::GetParent(pMsg->hwnd);

for( int i = 0; i < m\_aPreMessages.GetSize(); i++ )

{

CPaintManagerUI\* pT = static\_cast<CPaintManagerUI\*>(m\_aPreMessages[i]);

HWND hTempParent = hWndParent;

while(hTempParent)

{

if(pMsg->hwnd == pT->GetPaintWindow() || hTempParent == pT->GetPaintWindow())

{

if (pT->TranslateAccelerator(pMsg))

return true;

// 这里进行消息过滤

if( pT->PreMessageHandler(pMsg->message, pMsg->wParam, pMsg->lParam, lRes) )

return true;

return false;

}

hTempParent = GetParent(hTempParent);

}

}

}

else

{

for( int i = 0; i < m\_aPreMessages.GetSize(); i++ )

{

CPaintManagerUI\* pT = static\_cast<CPaintManagerUI\*>(m\_aPreMessages[i]);

if(pMsg->hwnd == pT->GetPaintWindow())

{

if (pT->TranslateAccelerator(pMsg))

return true;

if( pT->PreMessageHandler(pMsg->message, pMsg->wParam, pMsg->lParam, lRes) )

return true;

return false;

}

}

}

return false;

}

bool CPaintManagerUI::PreMessageHandler(UINT uMsg, WPARAM wParam, LPARAM lParam, LRESULT& /\*lRes\*/)

{

for( int i = 0; i < m\_aPreMessageFilters.GetSize(); i++ )

{

bool bHandled = false;

LRESULT lResult = static\_cast<IMessageFilterUI\*>(m\_aPreMessageFilters[i])->MessageHandler(uMsg, wParam, lParam, bHandled); // 这里调用接口 IMessageFilterUI::MessageHandler 来进行消息过滤

if( bHandled ) {

return true;

}

}

…… ……

return false;

}

在发送到窗口过程前，有一个过滤接口：IMessageFilterUI，此接口只有一个成员：MessageHandler，我们的窗口类要提前过滤消息，只要实现这个IMessageFilterUI，调用**CPaintManagerUI::AddPreMessageFilter**，将我们的窗口类实例指针添加到**CPaintManagerUI::m\_aPreMessageFilters** 数组中。当消息到达窗口过程之前，就会会先调用我们的窗口类的成员函数：MessageHandler。

下面是AddPreMessageFilter代码：

bool CPaintManagerUI::AddPreMessageFilter(IMessageFilterUI\* pFilter)

{

// 将实现好的接口实例，保存到数组 m\_aPreMessageFilters 中。

ASSERT(m\_aPreMessageFilters.Find(pFilter)<0);

return m\_aPreMessageFilters.Add(pFilter);

}

我们从函数**CPaintManagerUI::TranslateMessage**代码中能够看到，这个过滤是在大循环：

for( int i = 0; i < m\_aPreMessages.GetSize(); i++ )

中被调用的。如果m\_aPreMessages.GetSize()为0，也就不会调用过滤函数。从代码中追溯其定义：

static CStdPtrArray m\_aPreMessages;

是个静态变量，MessageLoop，TranslateMessage等也都是静态函数。其值在CPaintManagerUI::Init中被初始化：

void CPaintManagerUI::Init(HWND hWnd)

{

ASSERT(::IsWindow(hWnd));

// Remember the window context we came from

m\_hWndPaint = hWnd;

m\_hDcPaint = ::GetDC(hWnd);

// We'll want to filter messages globally too

m\_aPreMessages.Add(this);

}

看来，m\_aPreMessages存储的类型为CPaintManagerUI\* ,也就说，这个静态成员数组里，存储了当前进程中所有的CPaintManagerUI实例指针，所以，如果有多个CPaintManagerUI实例， 也不会存在过滤问题，互不干扰，都能各自过滤。当然m\_aPreMessages不止用在消息循环中，也有别的用处。我觉得这个名字起得有点诡异。

然后再说，消息抵达窗口过程后，如何处理。首先，要清楚，窗口过程在哪儿？使用DuiLib开发，我们的窗口类无外呼，继承俩个基类：一个是功能简陋一点 的：CWindowWnd，一个是功能健全一点的：WindowImplBase（继承于CWindowWnd）。然后，我们实例化窗口类，调用这俩个基 类的Create函数，创建窗口，其内部注册了窗口过程：

LRESULT CALLBACK CWindowWnd::\_\_WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)

{

CWindowWnd\* pThis = NULL;

if( uMsg == WM\_NCCREATE ) {

LPCREATESTRUCT lpcs = reinterpret\_cast<LPCREATESTRUCT>(lParam);

pThis = static\_cast<CWindowWnd\*>(lpcs->lpCreateParams);

pThis->m\_hWnd = hWnd;

::SetWindowLongPtr(hWnd, GWLP\_USERDATA, reinterpret\_cast<LPARAM>(pThis));

}

else {

pThis = reinterpret\_cast<CWindowWnd\*>(::GetWindowLongPtr(hWnd, GWLP\_USERDATA));

if( uMsg == WM\_NCDESTROY && pThis != NULL ) {

LRESULT lRes = ::CallWindowProc(pThis->m\_OldWndProc, hWnd, uMsg, wParam, lParam);

::SetWindowLongPtr(pThis->m\_hWnd, GWLP\_USERDATA, 0L);

if( pThis->m\_bSubclassed ) pThis->Unsubclass();

pThis->m\_hWnd = NULL;

pThis->OnFinalMessage(hWnd);

return lRes;

}

}

if( pThis != NULL ) {

return pThis->HandleMessage(uMsg, wParam, lParam);

}

else {

return ::DefWindowProc(hWnd, uMsg, wParam, lParam);

}

}

里面，主要做了一些转换，细节自行研究，最终，他会调用pThis→HandleMessage(uMsg, wParam, lParam);。也即是说，HandleMessage相当于一个窗口过程（虽然它不是，但功能类似）。他是CWindowWnd的虚函数：

virtual LRESULT HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam);

所以，如果我们的窗口类实现了HandleMessage，就相当于再次过滤了窗口过程，HandleMessage代码框架如下：

LRESULT HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam)

{

if( uMsg == WM\_XXX ) {

… …

return 0;

}

else if( uMsg == WM\_XXX) {

… …

return 1;

}

LRESULT lRes = 0;

if( m\_pm.MessageHandler(uMsg, wParam, lParam, lRes) ) //CPaintManagerUI::MessageHandler

return lRes;

return CWindowWnd::HandleMessage(uMsg, wParam, lParam); // 调用父类HandleMessage

}

在注意：CPaintManagerUI::MessageHandler，名称为MessageHandler，而不是HandleMessage。  
没有特殊需求，一定要调用此函数，此函数处理了绝大部分常用的消息响应。而且如果你要响应Notify事件，不调用此函数将无法响应，后面会介绍。

好现在我们已经知道，俩个地方可以截获消息：

1. 实现IMessageFilterUI接口，调用CPaintManagerUI:: AddPreMessageFilter，进行消息发送到窗口过程前的过滤。
2. 重载HandleMessage函数，当消息发送到窗口过程中时，最先进行过滤。

下面继续看看void Notify(TNotifyUI& msg)是如何响应的。我们的窗口继承于INotifyUI接口，就必须实现此函数：

class INotifyUI

{

public:

    virtual void Notify(TNotifyUI& msg) = 0;

};

上面我说了，在我们的HandleMessage要调用CPaintManagerUI::MessageHandler来进行后续处理。下面是一个代码片段：

bool CPaintManagerUI::MessageHandler(UINT uMsg, WPARAM wParam, LPARAM lParam, LRESULT& lRes)

{

… …

TNotifyUI\* pMsg = NULL;

while( pMsg = static\_cast<TNotifyUI\*>(m\_aAsyncNotify.GetAt(0)) ) {

m\_aAsyncNotify.Remove(0);

if( pMsg->pSender != NULL ) {

if( pMsg->pSender->OnNotify ) pMsg->pSender->OnNotify(pMsg);

}

// 先看这里，其它代码先忽略；我们看到一个转换操作static\_cast<INotifyUI\*>

for( int j = 0; j < m\_aNotifiers.GetSize(); j++ ) {

static\_cast<INotifyUI\*>(m\_aNotifiers[j])->Notify(\*pMsg);

}

delete pMsg;

}

// Cycle through listeners

for( int i = 0; i < m\_aMessageFilters.GetSize(); i++ )

{

bool bHandled = false;

LRESULT lResult = static\_cast<IMessageFilterUI\*>(m\_aMessageFilters[i])->MessageHandler(uMsg, wParam, lParam, bHandled);

if( bHandled ) {

lRes = lResult;

return true;

}

}

… …

}

定义为CStdPtrArray m\_aNotifiers;数组，目前还看不出其指向的实际类型。看看，什么时候给该数组添加成员：

bool CPaintManagerUI::AddNotifier(INotifyUI\* pNotifier)

{

ASSERT(m\_aNotifiers.Find(pNotifier)<0);

return m\_aNotifiers.Add(pNotifier);

}

不错，正是AddNotifier，类型也有了：INotifyUI。所以，入门教程里会在响应WM\_CREATE消息的时候，调用 AddNotifier（this），将自身加入数组中，然后在CPaintManagerUI::MessageHandler就能枚举调用。由于 AddNotifer的参数为INotifyUI\*，所以，我们要实现此接口。   
所以，当HandleMessage函数被调用后，紧接着会调用我们的Notify函数。如果你没有对消息过滤的特殊需求，实现INotifyUI即可，在Notify函数中处理消息响应。

上面的Notify调用，是响应系统产生的消息。程序本身也能手动产生，其函数为：

void CPaintManagerUI::SendNotify(TNotifyUI& Msg, bool bAsync /\*= false\*/)

DuiLib将发送的Notify消息分为了同步和异步消息。同步就是立即调用(类似SendMessage)，异步就是先放到队列中，下次再处理。（类似PostMessage）。

void CPaintManagerUI::SendNotify(TNotifyUI& Msg, bool bAsync /\*= false\*/)

{

… …

if( !bAsync ) {

// Send to all listeners

// 同步调用OnNotify，注意不是Notify

if( Msg.pSender != NULL ) {

if( Msg.pSender->OnNotify ) Msg.pSender->OnNotify(&Msg);

}

// 还会再次通知所有注册了INotifyUI的窗口。

for( int i = 0; i < m\_aNotifiers.GetSize(); i++ ) {

static\_cast<INotifyUI\*>(m\_aNotifiers[i])->Notify(Msg);

}

}

else {

// 异步调用，添加到m\_aAsyncNotify array中

TNotifyUI \*pMsg = new TNotifyUI;

pMsg->pSender = Msg.pSender;

pMsg->sType = Msg.sType;

pMsg->wParam = Msg.wParam;

pMsg->lParam = Msg.lParam;

pMsg->ptMouse = Msg.ptMouse;

pMsg->dwTimestamp = Msg.dwTimestamp;

m\_aAsyncNotify.Add(pMsg);

}

}

我们CPaintManagerUI::MessageHandler在开始处发现一些代码：

TNotifyUI\* pMsg = NULL;

while( pMsg = static\_cast<TNotifyUI\*>(m\_aAsyncNotify.GetAt(0)) ) {

m\_aAsyncNotify.Remove(0);

if( pMsg->pSender != NULL ) {

if( pMsg->pSender->OnNotify ) pMsg->pSender->OnNotify(pMsg);

}

可以看到MessageHandler首先从异步队列中一个消息并调用OnNotify。OnNotify和上面的Notify不一样哦。

OnNotify是响应消息的另外一种方式。它的定义为：

CEventSource OnNotify;

属于CControlUI类。重载了一些运算符，如 operator()；要让控件响应手动发送（SendNotify）的消息，就要给控件的OnNotify，添加消息代理。在DuiLib的TestApp1中的OnPrepare函数里，有：

CSliderUI\* pSilder = static\_cast<CSliderUI\*>(m\_pm.FindControl(\_T("alpha\_controlor")));

if( pSilder ) pSilder->OnNotify += MakeDelegate(this, &CFrameWindowWnd::OnAlphaChanged);

至于代理的代码实现，我就不展示了，这里简单说明，就是将类成员函数，作为回调函数，加入到OnNotify中，然后调用 pMsg→pSender→OnNotify(pMsg)的时候，循环调用所有的类函数，实现通知的效果。代理代码处理的很巧妙，结合多态和模板，能将任 何类成员函数作为回调函数。

查阅CSliderUI代码，发现他在自身的DoEvent函数内调用了诸如：

m\_pManager->SendNotify(this, DUI\_MSGTYPE\_VALUECHANGED);

类似的代码，调用它，我们就会得到通知。

现在，又多了两种消息处理的方式：

1. 实现INotifyUI，调用CPaintManagerUI::AddNotifier，将自身加入Notifier队列。
2. 添加消息代理（其实就是将成员函数最为回到函数加入），MakeDelegate(this, &CFrameWindowWnd::OnAlphaChanged);，当程序某个地方调用了 CPaintManagerUI::SendNotify，并且Msg.pSender正好是注册的this，我们的类成员回调函数将被调用。

搜寻CPaintManagerUI代码，我们发现还有一些消息过滤再里面：

bool CPaintManagerUI::AddMessageFilter(IMessageFilterUI\* pFilter)

{  
    ASSERT(m\_aMessageFilters.Find(pFilter)<0);  
    return m\_aMessageFilters.Add(pFilter);

}

m\_aMessageFilters也是IMessageFilterUI array，和m\_aPreMessageFilters类似。

上面我们介绍的是CPaintManagerUI::AddPreMessageFilter，那这个又是在哪儿做的过滤？

还是CPaintManagerUI::MessageHandler中：

……

// Cycle through listeners

for( int i = 0; i < m\_aMessageFilters.GetSize(); i++ )

{

bool bHandled = false;

LRESULT lResult = static\_cast<IMessageFilterUI\*>(m\_aMessageFilters[i])->MessageHandler(uMsg, wParam, lParam, bHandled);

if( bHandled ) {

lRes = lResult;

return true;

}

}

… …

这个片段是在，异步OnNotify和Nofity消息响应，被调用后。才被调用的，优先级也就是最低。但它始终会被调用，因为异步OnNotify和 Nofity消息响应没有返回值，不会因为消息已经被处理，而直接退出。DuiLib再次给用户一个处理消息的机会。用户可以选择将bHandled设置 为True，从而终止消息继续传递。我觉得，这个通常是为了弥补OnNotify和Nofity没有返回值的问题，在m\_aMessageFilters 做集中处理。

处理完所有的消息响应后，如果消息没有被截断，CPaintManagerUI::MessageHandler继续处理大多数默认的消息，它会处理在其管理范围中的所有控件的大多数消息和事件等。

然后，消息机制还没有完，这只是CPaintManagerUI::MessageHandler中的消息机制，如果继承的是 WindowImplBase, WindowImplBase实现了DuiLib窗口的大部分功能。WindowImplBase继承了CWindowWnd，重载了 HandleMessage，也就是说，消息发送的窗口过程后，第一个调用的是WindowImplBase::HandleMessage：

LRESULT WindowImplBase::HandleMessage(UINT uMsg, WPARAM wParam, LPARAM lParam)

{

LRESULT lRes = 0;

BOOL bHandled = TRUE;

switch (uMsg)

{

case WM\_CREATE: lRes = OnCreate(uMsg, wParam, lParam, bHandled); break;

case WM\_CLOSE: lRes = OnClose(uMsg, wParam, lParam, bHandled); break;

case WM\_DESTROY: lRes = OnDestroy(uMsg, wParam, lParam, bHandled); break;

#if defined(WIN32) && !defined(UNDER\_CE)

case WM\_NCACTIVATE: lRes = OnNcActivate(uMsg, wParam, lParam, bHandled); break;

case WM\_NCCALCSIZE: lRes = OnNcCalcSize(uMsg, wParam, lParam, bHandled); break;

case WM\_NCPAINT: lRes = OnNcPaint(uMsg, wParam, lParam, bHandled); break;

case WM\_NCHITTEST: lRes = OnNcHitTest(uMsg, wParam, lParam, bHandled); break;

case WM\_GETMINMAXINFO: lRes = OnGetMinMaxInfo(uMsg, wParam, lParam, bHandled); break;

case WM\_MOUSEWHEEL: lRes = OnMouseWheel(uMsg, wParam, lParam, bHandled); break;

#endif

case WM\_SIZE: lRes = OnSize(uMsg, wParam, lParam, bHandled); break;

case WM\_CHAR: lRes = OnChar(uMsg, wParam, lParam, bHandled); break;

case WM\_SYSCOMMAND: lRes = OnSysCommand(uMsg, wParam, lParam, bHandled); break;

case WM\_KEYDOWN: lRes = OnKeyDown(uMsg, wParam, lParam, bHandled); break;

case WM\_KILLFOCUS: lRes = OnKillFocus(uMsg, wParam, lParam, bHandled); break;

case WM\_SETFOCUS: lRes = OnSetFocus(uMsg, wParam, lParam, bHandled); break;

case WM\_LBUTTONUP: lRes = OnLButtonUp(uMsg, wParam, lParam, bHandled); break;

case WM\_LBUTTONDOWN: lRes = OnLButtonDown(uMsg, wParam, lParam, bHandled); break;

case WM\_MOUSEMOVE: lRes = OnMouseMove(uMsg, wParam, lParam, bHandled); break;

case WM\_MOUSEHOVER: lRes = OnMouseHover(uMsg, wParam, lParam, bHandled); break;

default: bHandled = FALSE; break;

}

if (bHandled) return lRes;

lRes = HandleCustomMessage(uMsg, wParam, lParam, bHandled);

if (bHandled) return lRes;

if (m\_PaintManager.MessageHandler(uMsg, wParam, lParam, lRes))

return lRes;

return CWindowWnd::HandleMessage(uMsg, wParam, lParam);

}

WindowImplBase处理一些消息，使用成员函数On\*\*\*来处理消息，所以，可以重载这些函数达到消息过滤的目的。 然后，我们看到，有一个函数：WindowImplBase::HandleCustomMessage，它是虚函数，我们可以重载此函数，进行消息过滤，由于还没有调用m\_PaintManager.MessageHandler，所以在收到Notify消息之前进行的过滤。

有多了两种方式：

* 重载父类：WindowImplBase的虚函数
* 重载父类：WindowImplBase::HandleCustomMessage函数

最后，继承于WindowImplBase，还有一种过滤消息的方式，和Notify消息平级，实现方式是仿造的MFC消息映射机制： WindowImplBase实现了INotifyUI接口，并且AddNotify了自身，所以，它会收到Notify消息：

void WindowImplBase::Notify(TNotifyUI& msg)

{

    return CNotifyPump::NotifyPump(msg);

}

NotifyPump 调用 LoopDispatch，代码片段如下：

... ...

const DUI\_MSGMAP\_ENTRY\* lpEntry = NULL;

const DUI\_MSGMAP\* pMessageMap = NULL;

#ifndef UILIB\_STATIC

for(pMessageMap = GetMessageMap(); pMessageMap!=NULL; pMessageMap = (\*pMessageMap->pfnGetBaseMap)())

#else

for(pMessageMap = GetMessageMap(); pMessageMap!=NULL; pMessageMap = pMessageMap->pBaseMap)

#endif

{

#ifndef UILIB\_STATIC

ASSERT(pMessageMap != (\*pMessageMap->pfnGetBaseMap)());

#else

ASSERT(pMessageMap != pMessageMap->pBaseMap);

#endif

if ((lpEntry = DuiFindMessageEntry(pMessageMap->lpEntries,msg)) != NULL)

{

goto LDispatch;

}

}

... ...

代码量过多，这里进行原理说明，和MFC一样，提供了一些消息宏

DUI\_DECLARE\_MESSAGE\_MAP()

DUI\_BEGIN\_MESSAGE\_MAP(CKeyBoardDlg, CNotifyPump)

    DUI\_ON\_MSGTYPE(DUI\_MSGTYPE\_CLICK, OnClick)

    DUI\_ON\_MSGTYPE(DUI\_MSGTYPE\_WINDOWINIT, OnInitWindow)

DUI\_END\_MESSAGE\_MAP()

和MFC原理一样，声明一些静态变量，存储类的信息，插入一些成员函数，最为回调，最后生成一张静态表。当WindowImplBase::Notify有消息时，遍历表格，进行消息通知。

总结，DuiLib消息响应方式：

* 实现IMessageFilterUI接口，调用CPaintManagerUI::AddPreMessageFilter，进行消息发送到窗口过程前的过滤。
* 重载HandleMessage函数，当消息发送到窗口过程中时，最先进行过滤。
* 实现INotifyUI，调用CPaintManagerUI::AddNotifier，将自身加入Notifier队列。
* 添加消息代理（其实就是将成员函数最为回到函数加入），MakeDelegate(this, &CFrameWindowWnd::OnAlphaChanged);，当程序某个地方调用了 CPaintManagerUI::SendNotify，并且Msg.pSender正好是this，我们的类成员回调函数将被调用。
* 重载父类：WindowImplBase的虚函数
* 重载父类：WindowImplBase::HandleCustomMessage函数
* 使用类似MFC的消息映射