一、 基础

　　GDI的绘图函数基本上都是有状态的，所有的函数都要求一个HDC类型的句柄。这个HDC的获得有几个途径BeginPaint,GetWindowDC, GetDC.他们的参数都只需要一个HWND就差不多了。记得调用了BeginPaint后要调用EndPaint进行清理，调用GetWindowDC和GetDC后要调ReleaseDC进行清理。在MFC代码中常常遇到的CDC CPaintDC CWindowDC CClientDC。在这里稍作解释。

　　CDC :例如用GDI画矩形要Rectangle(hDC,...)，而使用CDC则是dc.Rectangle(...),由此可见CDC主要是把原本需要HDC作为参数的GDI函数封装了一下，HDC成了它的一个成员变量。

　　CPaintDC CWindowDC CClientDC:他们都是从CDC继承，分别是对上面所说的BeginPaint,GetWindowDC, GetDC调用对进行封装(CPaintDC构造时调用BeginPaint,析构时调用EndPaint,其余同理)。

　　BeginPaint:一般用在对WM\_PAINT的响应函数中使用

　　GetWindowDC:可获得整个Window的HDC，而GetDC仅能获得客户区的HDC，区别就在于--

　　前者有效地绘制区域是整个窗口(边框、标题栏、客户区的总和)。  
　　后者有效地绘制区域仅限于客户区。

　　两者的坐标系都是相对坐标而非屏幕坐标，原点是(0，0)。即以自己可绘制区域的左上角作为原点。

　　这里可以顺带的讲讲RECT了，RECT是一个结构，依次有4个成员left,top,right,bottom用来代表一个矩形区域。CRect从RECT继承，提供了一些常用的操作(例如说位移，缩小等等)，其实就是改变4个成员的值。完全不用CRect也可以。许多GDI函数都要求一个RECT作为参数，或者类似的用(x,y,cx,cy)作参数，其实也就是一个RECT变种，用了宽度和高度罢了。

二、 实例教程

　　基础知识介绍完毕，开始实例教程：

　我们以如何绘制一个具有平面风格的状态栏为例：

　　首先从CStatusBar继承一个类：CStatusBarNew。(如果无法通过类向导做这件事，而你又对MFC的MESSAGEMAP等等东西不熟悉，可以从CStatusBarCtrl继承一个，待生成代码后，把所有的CStatusBarCtrl改为CStatusBar)

　　在此，只需要重写WM\_PAINT和WM\_ERASEBKGND这两个消息的响应函数。

BOOL CStatusBarNew::OnEraseBkgnd(CDC\* pDC)   
{  
// TODO: Add your message handler code here and/or call default  
CRect rect;  
GetWindowRect(&rect);  
ScreenToClient(&rect);  
CBrush brush(0xf2f2f2);  
pDC->FillRect(&rect, &brush);  
return TRUE;  
}

　　上面函数把状态栏背景用0xf2f2f2这种颜色填充。

void CStatusBarNew::OnPaint()   
{  
CPaintDC cDC(this); // device context for painting  
// TODO: Add your message handler code here  
CRect rcItem;  
cDC.SetBkMode(TRANSPARENT);  
cDC.SelectObject (::GetStockObject (NULL\_BRUSH));//选入画刷

// 获取字体  
CFont\* pfont = GetFont();  
CFont\* def\_font;  
if (pfont)  
def\_font = cDC.SelectObject(pfont);//选入字体

CPen pen;  
pen.CreatePen(PS\_SOLID, 1, RGB(0xBD, 0xBA, 0xBD));  
CPen\* pOldPen = cDC.SelectObject(&pen);//选入画笔

CBrush br(0x00f2f2f2);  
for ( int i = 0; i < m\_nCount; i++ )  
{  
GetItemRect (i, rcItem);  
//填充面板背景  
cDC.FillRect(rcItem, &br);  
rcItem.bottom--;  
if(i == 0) rcItem.left += 2;

//对每个面板画圆角矩形  
cDC.RoundRect(rcItem, CPoint(5, 5));

//画面板上的文字  
UINT nNewStyle = GetPaneStyle(i);  
//如果style为SBPS\_DISABLED，则跳过不画  
if ((nNewStyle & SBPS\_DISABLED) != 0) continue;  
CString text = GetPaneText(i);  
UINT uFormat = DT\_SINGLELINE | DT\_NOPREFIX | DT\_TOP | DT\_LEFT;  
rcItem.left += 3;  
rcItem.top += 3;  
cDC.DrawText(text, rcItem, uFormat);  
}   
if (pfont)  
cDC.SelectObject(def\_font);//恢复字体

//画右下角小标志(这里画了六个小圆圈)  
if (GetStyle() & SBARS\_SIZEGRIP)  
{  
CRect rc;  
GetClientRect(&rc);  
rc.left = rcItem.right;  
rc.right--;  
rc.bottom--;  
rc.left = rc.right - rc.Width() / 4;  
rc.top = rc.bottom - rc.Width();  
int w = rc.Width();  
rc.top++;  
rc.left++;  
cDC.SelectObject(GetStockObject(GRAY\_BRUSH));  
cDC.Ellipse(&rc);  
rc.OffsetRect(-w, -w);  
cDC.Ellipse(&rc);  
rc.OffsetRect(w, 0);  
cDC.Ellipse(&rc);  
rc.OffsetRect(-w, w);  
cDC.Ellipse(&rc);  
rc.OffsetRect(-w, 0);  
cDC.Ellipse(&rc);  
rc.OffsetRect(2 \* w, -2 \* w);  
cDC.Ellipse(&rc);  
}

cDC.SelectObject(pOldPen);//恢复画笔

}

　　上面的函数我们可以多次看到SelectObject的调用，这就是前面所说的绘图函数基本上都是有状态的。这个状态保存在HDC中，而SelectObject则设置HDC的状态。通常称为选入。至于注释中的恢复是怎么回事呢？这要从CPen CBrush CFont等等说起了，它们是对GDI对象的封装。GDI对象通过CreatePen CreateBrush CreateFont等等函数创建，返回一个HGDIOBJ。这些对象不使用的时候需要销毁，用DeleteObject函数，但是如果一个HGDIOBJ被选入到一个HDC中的时候，它就不能被销毁，这样就造成了GDI资源的泄漏。解决这一问题通常有两种做法：

　　第一种，就是上面代码中看到的：

　　先保存原来的HGDIOBJ，def\_font = cDC.SelectObject(pfont);

　　用完了之后再恢复原来的 cDC.SelectObject(def\_font);

　　这样做，就保证了pfont能被正确销毁，至于原来的def\_font能不能被销毁，就不关我们的事了。

　　第二种，利用了系统的库存对象。库存GDI对象是windows系统预先创建的，不需要应用程序销毁。所以，不需要保存原来的HGDIOBJ，直接像这样

　　SelectObject (hdc, ::GetStockObject (NULL\_BRUSH));

　　或者cDC.SelectStockObject(NULL\_BRUSH);

　　就可以保证HDC中没有被选入任何我们自己创建的画刷了。

　　这两种方法各有好处，视情况选用。

　　另外上面说大部分GDI函数都是有状态的，有一个例外就是FillRect函数，它靠一个传给他的画刷进行填充。

三、 技巧

　　实例讲述完毕，接下来有一些补充技巧：

　　1. GDI绘图技巧的学习：通过阅读、运行、调试别人源代码获得经验这条路径是最快的。  
　　2．GDI程序的调试

　　调试GDI一般来说比其他程序困难，但是掌握了一些技巧也就没什么障碍了。调试GDI的时候，将IDE和代调试的程序窗口在桌面上尽量分开排列，不要重叠在一起。这样你能通过单步执行，看到每一步的绘图效果。

　　为配合上述策略，在应用程序初始化的时候加上下面一句：

　　#ifdef \_DEBUG   
　　GdiSetBatchLimit(1);  
　　#endif

　　这能保证调试时每一条GDI函数调用能马上产生效果。因为Windows为了性能优化，可能会分批处理GDI调用。

　　3.内存绘图

　　首先理解内存绘图，即把要绘制的东西先在内存中画好,然后一次性的画到屏幕上来。内存绘图经常用来防止闪烁。因为闪烁的原因是因为反差太大。例如你的绘图过程是先用白色擦除整个窗口，然后再将黑色的文字画到屏幕上来，这样在窗口重绘的时候，原本黑色文字区域就会白光一闪，然后再出现文字，也就是我们说的闪烁了。而内存绘图的过程呢，是先创建一个内存DC，然后在这个DC上把要绘制的图形画好，之后一次性的填到屏幕上去。

　　示例代码如下：

HDC hDestDC;  
RECT rc;  
//..此处得到目标的HDC和目标的RECT  
HDC hdc = ::CreateCompatibleDC (hDestDC);  
HBITMAP hBitmap = ::CreateCompatibleBitmap (hDestDC, rc.right, rc.bottom);  
HBITMAP hOldBitmap = ::SelectObject (hDC, hBitmap);  
//... 此处用hdc进行绘图  
//...  
::BitBlt (m\_hDestDC, rc.left, rc.top, rc.Width(), rc.Height(), hDC, rc.left, rc.top, SRCCOPY);  
::SelectObject (hDC, hOldBitmap);

　　当然，这样用起来不太方便，可以将这些操作封装到一个叫CMemDC的对象中，利用构造和析构自动进行这些操作。直接使用CMemDC还有一个好处，调试GDI时，如果图形都在内存中绘制，那么还是看不到绘图过程。

　　代码如果这样写：

CRect rc;  
GetWindowRect(&rc);  
#ifdef \_DEBUG  
CPaintDC dc;  
#else  
CPaintDC cdc;  
CMemDC dc(cdc.m\_hDC, &rc);  
#endif

　　那么就既能享受内存绘图的好处又能方便调试了。