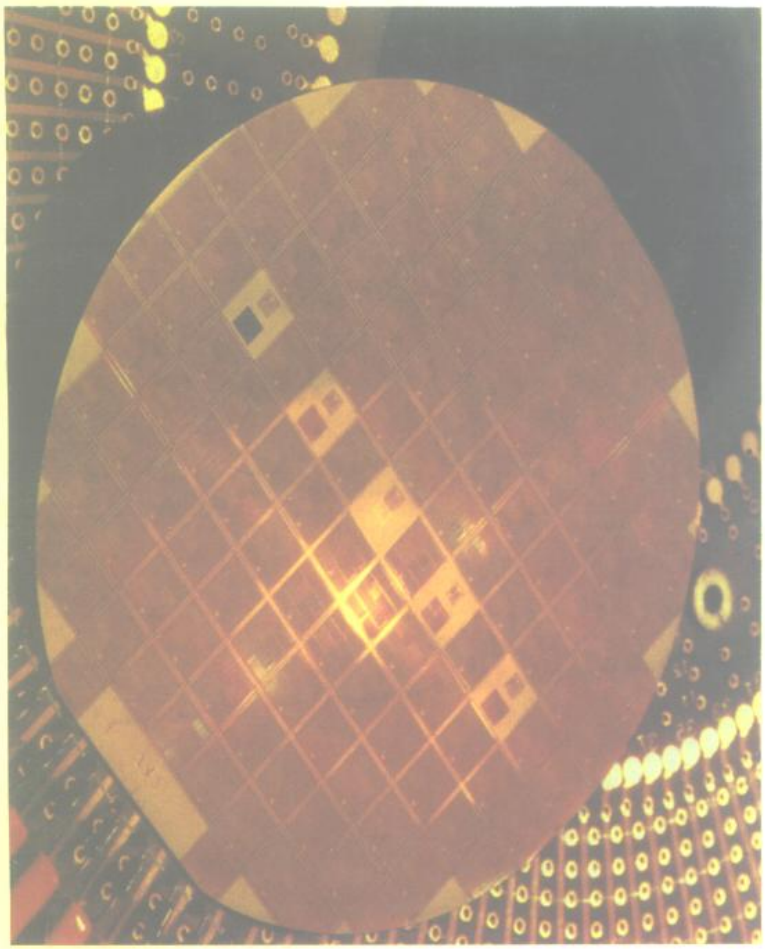
SCSD

软件开发文集

第七辑

《软件开发文集》编委会 编



- 用 MFC 编写 Windows 95 程序(Ⅱ)
  ——使用显示场境、画笔和画刷工作
- Visual FoxPro 中 SCREEN 对象的探讨
- 针对 Internet 的新的 OLE 准则
- Windows NT Server 3.51 评估指南(下)

科学出版社

软件开发文集

第七辑

《软件开发文集》编委会　编

科学出版社

1996

内 容 简 介

本书是为软件开发人员和计算机用户编写的《软件开发文集》系列书的第七辑，围绕专题内容，向读者提供了应用设计策略、开发技术规范、开发管理、开发平台等方面的技术资料。

本辑的重点文章是：用 MFC 编写 Windows 95 程序(Ⅱ)——使用显示场境、画笔和画刷工作，Visual FoxPro 中_SCREEN 对象的探讨，针对 Internet 的新的 OLE 准则，Windows NT Server 3.51 评估指南(下)。

本书适用于软件开发人员和广大计算机用户。

图书在版编目(CIP)数据

软件开发文集　第七辑/《软件开发文集》编委会编.
北京：科学出版社，1996.11
ISBN 7-03-005619-1

Ⅰ.软…　Ⅱ.软…　Ⅲ.软件开发-文集　Ⅳ.TP311.52-53

中国版本图书馆 CIP 数据核字(96)第 19643 号

科学出版社　出版
北京东黄城根北街 16 号
邮政编码：100717
北京管庄永胜印刷厂印刷
新华书店北京发行所发行　各地新华书店经售
*
1996 年 11 月第　一　版　　开本：787×1092　1/16
1996 年 11 月第一次印刷　　印张：5 1/2
印数：1—3500　　字数：110 000

定价：9.00 元

编委会名单

顾　问

杜家滨　冯玉琳　秦人华　林资山

主　编

郑茂松

副主编

（按姓氏笔画为序）

刘晓融　李　浩　廖恒毅

编　委

（按姓氏笔画为序）

王淑兰　巴建芬　华京梅　查良钿

由于时间仓促，《软件开发文集》难免存在这样或那样的问题，敬请广大的用户及读者提出宝贵意见和建议，以利改进，为我国软件产业的发展作出应有的贡献。

《软件开发文集》编委会

1996.3.25

编者的话

近年来，我国的计算机产业发展迅速，PC机销量已居世界第六位。同时，我国有一批人数众多、技术水平相当不错的软件开发人员，他们分布在全国各地、各个行业、各个领域。然而，由于计算机产业，特别是软件产业充满了变化，极富动态性，加之，我国幅员辽阔，通信技术又落后于较发达的国家，从而，软件开发人员面临着这样的问题：了解最新技术动态不及时、不方便，获取最新技术信息比较困难，所得到的信息往往不够全面、完整和深入，相互之间缺乏交流，因此，重复开发、重复别人走过的弯路的情况屡见不鲜。另一方面，全世界积累的资料浩如烟海，价格亦非常昂贵，鉴于国内经济承受能力有限，许多技术信息不能及时获得。有鉴于此，我们组织有关人员翻译、编写了以技术专集为特色的《软件开发文集》系列书，该套书将陆续出版、发行，及时向用户提供最新技术资料，力图为解决上述问题做一点尝试。

《软件开发文集》系列书的宗旨是为软件开发人员开辟一个相互交流经验和体会的园地，提供一条了解新技术、新工具、新产品的渠道，设立一个探讨软件开发理论和开发技术的论坛，帮助软件开发人员更快、更直接地获得有关软件开发的信息，提高软件开发人员的技术水平和工作效率，充分发挥软件的作用，提高软件的使用价值，促进我国软件产业与世界同步发展。

《软件开发文集》是面向软件开发人员的中、高档次的技术专集，所收集的文章向软件开发人员提供了最新科技动态，以及开发技术规范、开发经验和技巧、软件开发管理、应用设计策略、开发平台方面的内容等，同时，根据用户在软件使用和开发过程中遇到的难点、疑点给予一定的解答和帮助。《软件开发文集》资料主要来源于微软公司的“Microsoft System Journal”，“Developer News”，“Microsoft Development Library”，“TechNet”等，并收入国内软件开发人员撰写的有关文章。欢迎国内广大软件开发人员为《软件开发文集》撰稿，介绍自己的经验、心得、体会。特别要注重文章内容适合我国国情。

目前，在国内面向计算机最终用户的图书、杂志丰富多彩，但面向软件开发人员的图书、杂志却寥寥无几。我们期望《软件开发文集》能够弥补这方面的空白，并且不辜负广大用户的期望，把握好学术方向，确定好信息服务和技术支持的层次和深度，把《软件开发文集》办成深受广大用户喜爱的丛书。

目　录

1

用MFC编写Windows 95程序(Ⅱ)

——使用显示场景、画笔和画刷工作

在“用 MFC 编写 Windows 95 程序”系列的第 I 部分，你学会了怎样写一个简单的 MFC 应用程序，该应用程序可用于创建一个窗口，并把它显示在屏幕上。你也学会了 MFC 程序的结构，了解了 MFC 程序与用 C 语言写的传统应用程序之间的区别。本文将介绍一些用于在窗口中绘图的 MFC 类和成员函数。

在 Windows 中，负责图形输出的部分是图形设备接口(GDI)，GDI 服务由 GDI.EXE 和 GDI32.DLL 提供，它们分别拥有 GDI 的 16 位部分和 32 位部分。类似其他的核心系统 DLL，在 Windows 95 中，GDI 也被分成 16 位和 32 位两部分，以支持 16 位和 32 位的应用程序，同时这也提供了和以前 Windows 版本的高度兼容性。但是，你不必考虑 16/32 位代码的两分问题，只需简单地调用 GDI 服务或它的 MFC 等价物，Windows 将确保你的调用被发送到合适的地方。

在第 I 部分中，实例程序使用 CDC::DrawText 在窗口中输出文本。事实上，DrawText 只是 CDC 类提供文本和图形输出的许多成员函数中的一个。本文将更详细地讨论 CDC 类和它的派生，并且讨论如何使用画笔和画刷，然后编写一个例程在运行中显示这些对象；其次，增添滚动条；最后，将修改这个程序，使得 MFC 提供滚动逻辑。

(一)用 GDI 绘图

当基于 Windows 的程序把绘图输出到屏幕、打印机或其他输出设备时，程序并没有直接与设备通信，相反它绘到一个设备场境(DC)上。但首先它必须从 Windows 获得 DC 句柄，该句柄用作 GDI 输出函数的一个参数。

当你用 MFC 编写 Windows 程序时，DC 具有非常重要的意义。DC 对象封装程序调用以产生输出的 GDI 函数。在 MFC 中，你不用抓取 DC 句柄和调用 GDI 输出函数，至少不必直接这么做。相反，你获得一个指向 DC 对象的指针(或实例化一个 DC 对象)，并且用该指针去调用 DC 对象的成员函数。MFC 程序可以下面的方式从(0,0)点到(100,100)点画一条直线：

```
CDC* pDC=GetDC();
pDC->MoveTo(0,0);
pDC->LineTo(100,100);
```

CWnd::GetDC 返回一个指向代表 DC 的 CDC 对象的指针，并且该对象的成员函数被调

用来做绘图。可以容易地看到 MFC 在做什么，CDC 对象包裹着一个 DC 句柄，并且 CDC 对象的成员函数调用到 Windows API 中的 GDI 函数。在这种情况下，CDC::MoveTo 和 CDC::LineTo 直接映射到 API 函数::MoveToEx(::MoveTo 的 Win32 版本)和::LineTo。(提示：以 C++的全局域归结操作符::开始的函数名表示 Windows API 函数。)

在基于 Windows 的程序中，执行任何类型的图形输出的第一步是要获得到 DC 的入口，在 MFC 中，有多种方式来做它，GetDC 是其中之一。当你用完 DC 后，你必须释放它，这是因为 Windows 在系统中仅有 5 个 DC 可使用，且这 5 个 DC 必须被全部程序所共享(程序获得一个自己专用的 DC 是可能的，但那不是我们在这里讨论的特例)。如果你是通过调用 GetDC 获得了一个 DC 指针，那么你能调用 CWnd::Release DC 释放它：

```
CDC* pDC=GetDC();
//Do some drawing
ReleaseDC(pDC);
```

只要可能，总该释放 DC，并且 Release DC 总应该在 GetDC 被调入的同一消息的场境内调用。如果你用其他的方式获得 DC，如，通过实例化 CPaintDC 类或 CClient DC 类，当这些对象被删除时，MFC 为你释放 DC。你如何获得 DC(以及如何释放它)依赖于该调用周围的环境。

(二)在 OnPaint 中获得 DC

上辑中你学会了 MFC 程序使用透过消息映射图连接到消息系统的 OnPaint 处理函数处理 WM_PAINT 消息。OnPaint 时常是框架窗口类的成员函数，尽管从技术上说，它可以属于源于 CCmdTarget 的任何类。WM_PAINT 消息不同于其他消息的一个非常重要的方面是，如果 OnPaint 处理函数不能调用 Windows 的::Begin Paint 和::End Paint API，那么不管你的程序做了多少绘图，WM_PAINT 消息都不可能从消息队列中删除。其结果是程序阻塞于一遍又一遍地处理同一 WM_PAINT 消息的循环中。

为了在 OnPaint 处理函数中获得 DC，可用 MFC 的 CPaint DC 类。当一个 CPaint DC 对象被创建时，它自动地调用::Begin Paint 开始绘图过程。并且当它被销毁时，它调用::End Paint。下面的语句：

```
CPaintDC dc(this);
```

可在堆栈上创建一个名为 dc 的 DC 对象，该堆栈涉及本 DC 正在请求的窗口。用 CPaintDC 创建的 DC 时常作为绘图 DC 引用。用绘图 DC 画一条从(0,0)点到(100,100)点的直线的 OnPaint 处理函数可能看起来像下面这样：

```
void CMainWindow::OnPaint()
{
    CPaintDC dc(this);
    dc.MoveTo(0,0);
    dc.LineTo(100,100);
}
```

注意到，Release DC 没有被调用。当 OnPaint 结束并超出作用域时，CPaint DC 的解除程序

调用::EndPaint，接着释放 DC。如果你用 new 语句以下面的方式构建一个绘图 DC：

```
CPaintDC * pDC=new CPaintDC(this);
```

那么在你的 OnPaint 结束之前，要确保删除 pDC，否则::EndPaint 不会被调用。收到 WM_Paint 消息并不必定意味着程序的整个窗口需要重画。如果WM_Paint消息是由于另一窗口的位置改变暴露了窗口的角而产生的，那么仅需要更新那个暴露的角。WM_Paint 消息携带着描述无效矩形区的信息，窗口的这一无效矩形区需要重画。如果愿意，程序可以通过从 DC 中获得无效矩形区和仅更新窗口的相应区域来优化重画过程。CPaint 对象被构建后，无效矩形区的坐标可从以 m_ps 调用的 PAINTSTRUCT 结构的 rcPaint 域中获得，mps 是 CPaintDC 类的一个成员。rcPaint 是 RECT 类型的结构，按如下方式定义：

```
struct tagRECT{
    int left;
    int top;
    int right;
    int bottom;
}RECT;
```

rcPaint. left 和 rcPaint. right 容纳矩形区左、右边界的 x 坐标，rcPaint. top 和 rcPaint . bottom容纳上、下边界的 y 坐标。缺省地，窗口内的位置是以像素测量的。用户区的左上角像素是(0,0)点，x，y 随着你向右和向下移动而各自增加。如果测得窗口的用户区为 200 * 200 像素，并且无效矩形区为该窗口的右下四分之一，那么 rcPaint 将描述这样一个矩形区，它的左上角是(100,100)，右下角是(199,199)。如果你画出了无效矩形区，那也不是什么大问题。GDI 会自动裁剪输出，但裁剪会降低绘图速度。

(三)从 OnPaint 之外获得 DC

程序并不总是完全在 OnPaint 中绘图。假设你正在写一个画刷程序，当你单击鼠标钮时，该程序在屏幕上画对象。响应鼠标单击更新屏幕的一种方式是调用 CWnd:: InvalidateRect 去无效化窗口用户区的全部或者也许只是窗口用户区的一部分区域，让 On Paint 做绘图(当然，这是在更新窗口中对象绘图的你的程序内部表示之后，以使 OnPaint 知道画什么)。然而从速度上考虑，你可能宁愿通过得到非绘图 DC 立即更新屏幕，并用它做你的绘图。

如果你有指向窗口对象的指针，你总能用 GetDC 得到 DC。但是 MFC 提供了源于 CDC 的 CClientDC 类，使得它更容易，而不必使用下面的程序绘你的图：

```
CDC * pDC=GetDC();
//Do some drawing
ReleaseDC(pDC)
```

相反，你可以用下面这种方式做：

```
CClientDC dc(this);
//Do some drawing
```

现在 dc 表示你需要的 DC，并且你不必担心调用 Release DC。为什么呢？因为 CClient DC

的构造程序调用::GetDC，并且它的解除程序调用::Release DC。当用绘图 DC 时，如果由于某些原因你决定使用 new 实例化 CClientDC，那么你仅需关心释放 DC 就可以了。在那种情况下，在消息处理程序终止前，你必须调用 delete 以确保解除程序被调用。

在少数情况下，你可能不仅要画窗口的用户区，而且还要画非用户区(标题栏，窗口边界，等等)，MFC 提供 CWindow DC 类。CWindow DC 的工作方式就跟 CClient DC 一样，但是它所表示的 DC 包括了窗口边界之内的所有事物。程序员有时为了不同寻常的效果，如定制标题栏和带圆角的窗口，会使用 CWindow DC。CWindow DC 并不是你常常需要的东西。如果你要在窗口的非用户区做自己的绘图，那么你可使用 OnNcPaint 处理函数捕获 WM_NCPAINT 消息以决定什么时间非用户区需要绘图，不像 OnPaint、OnNcPaint 处理函数不必(并且也不应该)调用::Begin Paint 和::EndPaint。

在更少数的情况下，这时程序需要存取整个屏幕，你能构建一个 CClientDC 或 CWindowDC DC 对象，并传给构造程序一个 NULL 指针，以替代指向窗口对象的指针：

```
CClientDC dc(NULL);
```

下面这些语句可画一条从屏幕的左上角到右下角 100 个像素点的直线：

```
dc.MoveTo(0,0);
dc.LineTo(100,100);
```

屏幕捕获程序时常用这种方法获得访问整个屏幕。

(四)用 DC 绘图

一旦你拥有了用以工作的 DC，你就可以开始绘图了。你已经看到了 MFC 应用程序用于屏幕输出的很普通的 CDC 函数中的两个函数：MoveTo 和 LineTo。MoveTo 设置当前位置：一个在显示表面指定一个点的假想的指针。LineTo 从当前位置到调用中指定的位置画一条直线，并自动更新当前位置，以使下条线从前条线的结束点开始画。语句

```
dc.MoveTo(100,100);
dc.LineTo(200,100);
dc.LineTo(200,200);
dc.LineTo(100,200);
dc.LineTo(100,100);
```

可画一个每边为 100 单位的正方形。正像这个例子所表明的那样。如果当前位置就在你所要求的位置，那么就没有必要在每次调用 LineTo 之前调用 MoveTo。

MoveTo 和 LineTo 仅是 CDC 类提供在显示表面绘图的多个函数中的两个函数。表 1 列出了一些其他 CDC 绘图函数，你将会发现它们很有用。总的来说，CDC 类包括 150 种以上用于绘图、查询 GDI 信息、变换 DC 状态等等的成员函数。

表 1 包含了对当前画笔和画刷的引用。画笔和画刷由 MFC 类的 CPen 和 CBrush 表示。CDC::SelectObject 函数将一枝画笔或画刷选入关联的 DC。除非你调用 SelectObject 改变当前画笔或当前画刷，否则 GDI 一直使用缺省的画笔和画刷。缺省画笔可画 1 像素宽的实心黑线。缺省画刷以实心白色画区域。

表 1　有用的 CDC 绘图函数

函　数	描　述
ArcTo	用当前画笔画一段椭圆弧，并更新当前位置到弧的终点
Ellipse	在一个边界矩形内拟合一个椭圆，该椭圆以当前画笔画边，并以当前画刷填充
FillRect	用当前画刷填充一个矩形
FloodFill	以当前画刷填充任何形状和大小的边界区域
LineTo	以当前画笔画一条直线，并将当前位置更新到线的终点
MoveTo	设置当前位置，为绘图做准备
Pie	以当前画笔画一个带楔形边界的饼形图，并以当前画刷填充
PolyBezier	以当前画笔画一条或数条 Bezier 样条
Polygon	通过连接点队列画一个多边形，该多边形以当前画笔加边，以当前画刷填充
Rectangle	画一个矩形，该矩形以当前画笔加边，并以当前画刷填充
RoundRect	画一个圆角矩形，该圆角矩形以当前画笔加边，并以当前画刷填充
SetTextColor	设置文本输出颜色
TextOut	以当前文本色画文本(参见 SetTextColor)

你也能创建自己的画笔和画刷，并将它们选入 DC，以改变输出的属性。例如，你要画一个 10 像素宽的实心黑笔，和实心红画刷，并在调用 Ellipse 之前用 SelectObject 把它们选入 DC。假定 pPen 是一个指向该画笔对象的指针，pBrush 是一个指向该画刷对象的指针，并且 dc 代表 DC，那么画实心圆的代码可能看起来像下面这样：

```
dc. SelectObject(pPen);
de. SelectObject(pBrush);
Ellipse(0,0,100,100);
```

SelectObject 被重载以接收指向各种类型对象的指针，它的返回值是一个指向与前面选入 DC 同种类型的对象的指针。

当用 DC 绘图时，所有坐标都根据逻辑坐标指定。另一个 DC 属性是映射模式，它决定逻辑坐标怎样转换成设备坐标(屏幕上的像素)。GDI 支持 8 种映射模式，它们可以用 CDC::SetMapMode 函数设定，用 CDC::GetMapMode 函数查询。缺省映射模式是 MM_TEXT，在那里逻辑单位是像素(x 或 y 方向上的一个逻辑单位等同于一个像素)，原点(0，0)位于显示表面的左上角。对用户区 DC，这意味着窗口用户区左上角的像素是(0，0)。x 和 y 的值随着你向右和向下移动而增加(见图 1)。其他映射模式使用另外的把逻辑坐标转换成设备坐标的方法，并面向不同的坐标系统。例如，在 MM_LOMETRIC 映射模式中，一个逻辑单位对应显示表面的 0.1 毫米的长度，缺省的原点在左下角，x 和 y 的值随

着你向右和向上移动而增加。

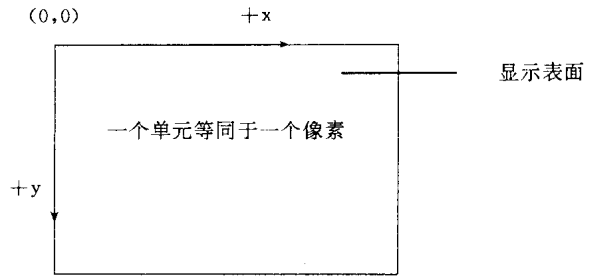


图1　缺省映射模式

在更详细地讨论画笔和画刷之前，有一点需要强调一下，因为 DC 是被频繁地请求和释放的共享资源，所以你不能把画笔和画刷选入 DC 后，期望当你在你的程序后面再次获得 DC 时，它们仍然处在被选状态。当一个 DC 被释放时，它的缺省属性被恢复。因此，每次你实例化 DC 对象时都必须重新初始化它。虽然有些方法可保护 DC 的属性，使得不必每次都初始化它们，但那是另一天的专题。一个绕过必须反复重新初始化 DC 的偷偷摸摸的方法，是在 OnCreate 中获得 DC，并且直到 WM_DESTROY 消息到达时才释放它。但这是不利的，因为可得到的 DC 数量是有限的。

(五)有关画笔的更多的信息

GDI 提供了一系列 API 函数，应用程序可以调用它们去创建在 DC 中使用的定制画笔和画刷，这些函数包括：:CreatePen，::CreateSolidBrush，::CreateHatchBrush 和 ::Create Pattern Brush。为简单起见，MFC 把这些函数裹进 CPen 和 CBrush 类，这使得画笔和画刷可以作为对象处理，而不是像 GDI 所提供的原始句柄那样处理。

MFC 用 CPen 类的对象表示画笔。创建 GDI 画笔的最简单方式是构建 CPen 对象，并传给它定义画笔的参数：

```
CPen pen(PS_SOLID,1,RGB(255,0,0));
```

第二种方法是构建一个未初始化的 CPen 对象，并且调用对象的 CreatePen 函数：

```
CPen pen();
pen.CreatePen(PS_SOLID,1,RGB(255,0,0));
```

第三种方法是构建一个未初始化的 CPen 对象，填写描述画笔的 LOGPEN 结构，然后调用 CreatePenIndirect 创建该画笔：

```
CPen pen();
LOGPEN lp;
lp.lopnStyle=PS_SOLID;
lp.lopnWidth=1;
lp.lopnColor=RGB(255,0,0);
pen.CreatePenIndirect(&lp);
```

如果画笔被成功地创建，CreatePen 和 CreatePenIndirect 就返回 TRUE(真)，否则返回 FALSE(假)。如果你允许该对象构造程序去创建画笔，那么假如该画笔不能被创建，则将产生一个 CResuarce Exception 类的异常事件。

一枝画笔有三种可定义的特征：风格、宽度和颜色。在以上三个创建画笔的例子中，画笔风格是 PS_SOLID，宽度是一个逻辑单位，颜色是亮红色的笔。传递给 CreatePen，CreatePenIndirect 或 CPen 构造程序的三个参数中的第一个参数指定画笔的风格，风格描述画笔将要绘的实心、点划和长划等等线的类型。PS_SOLID 创建画不间断线的实心画笔。特殊的 PS_INSIDEFRAME 风格画实心的直线或曲线，这些直线或曲线位于被画图形框架中的边界盒里面。例如，用 20 单位宽的 PS_SOLID 画笔画一个直径为 100 单位的圆，以圆的外边界上的任一点到直径所对的圆的另一外边界上的点，测量得到圆实际的直径将是 120 单位。为什么这样呢？这是因为被上面的画笔所画的边界在理论圆的每边上扩展了 10 个单位。用 PS_INSIDEFRAME 画笔画同样的圆，其直径将精确地为 100 单位(见图 2)。PS_INSIDEFRAME 风格不影响用 LineTo 和其他的不要求边界矩形的函数所画的直线和曲线。

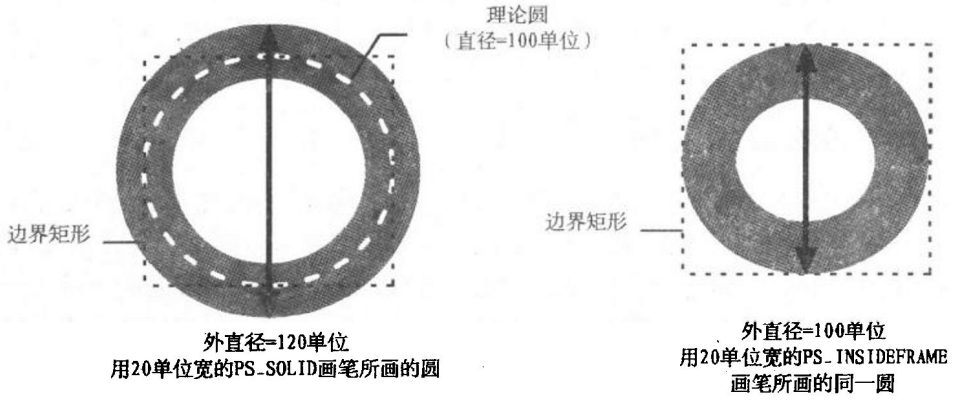


图 2　PS_INSIDEFRAME 风格

另一种画笔风格是 PS_NULL，它什么也不画。那你为什么要创建一个 NULL 画笔呢？因为它迟早会有用的，信不信由你。假设你要画一个无边界的实心红圆。如果你用 MFC 的 CDC::Ellipse 函数画这个圆，那么 Windows 会自动地使用当前选入 DC 的画笔加边界。你虽不能告诉 Ellipse 函数你不要边界，但你能选一个 NULL 画笔进入 DC，使得圆有一个不可见的边界。NULL 画刷也依类似的方式被使用。如果你要该圆有边界，而圆的内部透明，那么你可在绘图前把 NULL 画刷选入 DC。

传给 CPen 的画笔创建函数的第二个参数指定画笔宽度。画笔宽度以逻辑单位指定，它的物理意义随着 DC 的映射模式而变化。在缺省的 MM_TEXT 映射模式中，正如以前所说的那样，一个逻辑单位等同于一个像素。因此，由于缺省，你创建画笔时指定的宽度就是画笔的像素宽度，你能创建任何宽度的 PS_SOLID，PS_NULL 和 PS_INSIDEFRAME 画笔，但 PS_DASH，PS_DOT，PS_DASHDOT 和 PS_DASHDOTDOT 画笔只能是一个

逻辑单位宽。无论什么样的映射模式，为画笔指定 0 宽度都创建一个像素宽的画笔。

最后一个参数是画笔的颜色，Windows 使用 24 位(bit)RGB 颜色模型，在此颜色模型中，每种可能的颜色都用值从 0 到 255 的红、绿和蓝色来定义。值越高，颜色就越亮，并且越倾向于相应的颜色分量。经常被使用的宏 RGB 把由三个独立的颜色分量所指定的值结合成一个 Windows 识别的 COLORREF 值。因此，语句

```
CPen pen(PS_SOLID,1,RGB(255,0,0));
```

创建一支亮红色画笔(R＝255，G＝0 和 B＝0)，而语句

```
CPen pen(PS_SOLID,1,RGB(255,255,0));
```

通过组合红色和绿色创建一支黄色画笔。如果你的显示适配器不支持 24 位(bit)颜色，那么 Windows 会尽力混合它能显示的颜色来仿真它不能直接显示的颜色。只有比一个逻辑单位宽的 PS_INSIDEFRAME 画笔能使用杂色。如果必要，Windows 会自动把其他画笔的颜色映射到它能显示的最接近的实心颜色。通过把红、绿、蓝的值分别设成 192 或 255，你能得到想要的准确颜色。这些颜色是 Windows 程序进入每个显示适配器颜色寄存器的基本调色板的部分。

Windows 预定义了三个库存画笔，它们不用显式地创建就能够使用。GDI 库存对象用画笔对象从 CGdiObject 继承来的 CreateStockObject 函数来创建，库存画笔用常量 WHITE_PEN，BLACK_PEN 和 NULL_PEN 来标识，它们都是一个像素宽。

(六)有关画刷的更多的信息

MFC 的 CBrush 类封装了 GDI 画刷。画刷有三种基本变种：实心的、影线的和图案的。实心画刷用实心颜色绘图。实心颜色可能是一种真正的实心颜色，如果你的显示硬件不允许直接显示画刷的颜色，那么它就通过混合其他颜色来仿真。影线画刷使用六种模仿用在工程和建筑图中的图案的预定义影线风格之一绘图。图案画刷用位图绘图。CBrush 类为每个不同的画刷风格提供一个构造程序。

实心画刷能通过传递一个 COLORREF 值给 CBrush 构造程序经过一个步骤创建：

```
CBrush brush(RGB(255,0,0));
```

它能通过创建一个未初始化的 CBrush 对象并调用 CBrush::CreateSolidBrush 经过两个步骤创建：

```
CBrush brush();
brush. CreateSolidBrush(RGB(255,0,0));
```

以上两个例子都创建了一支画亮红色的实心画刷。如果愿意，画刷也能用 CBrush：：CreateBrushIndirect 间接地创建。同 CPen 一样，在画刷不可能被创建的不稳定的事件中，所有 CBrush 构造程序都抛出一个资源异常事件。

影线画刷可用接收影线索引和 COLORREF 值的构造程序创建，或用 CBrush::CreateHatchBrush 创建，语句

```
CBrush brush(HS_DIAGCROSS,RGB(255,0,0));
```

创建一个画刷，该画刷画与水平方向夹角为45°的交叉影线(或者，我猜是与垂直方向的夹角)。正像下面语句所做的一样：

```
CBrush brush();
brush. CreateHatchBrush(HS_ DIAGCROSS,RGB(255,0,0));
```

HS_DIAGCROSS是你可以从中选择的六种影线风格之一。当用影线画刷绘图时，影线之间的背景以实心白色填充，除非你通过使用CDC::SetBkColor改变背景色，或通过CDC::SetBkMode将背景模式OPAQUE改到TRANSPARENT来指定其他的颜色。下面的语句画一个100单位平方的矩形，用黑色交叉影线填充，并反衬淡灰色背景：

```
CBrush brush(HS_DIAGCROSS,RGB(0,0,0));
dc. SelectObject(&brush);
dc. SetBkColor(RGB(192,192,192));
Rectangle(0,0,100,100);
```

与此相对照，下面这些语句画反衬着现存背景的交叉影线矩形，而无论其他背景色可能是什么颜色(或颜色的组合)：

```
CBrush brush(HS_DIAGCROSS,RGB(0,0,0));
dc. SelectObject(&brush);
dc. SetBkMode(TRANSPARENT);
Rectangle(0,0,100,100);
```

背景色和背景模式也决定了Windows如何填充用给定风格的画笔(这种画笔不是实心的，例如，PS_DASH)所画线的间隙和文本串中字符的后面。

Windows也提供了七种库存画刷：BLACK_BRUSH，DKGRAY_BRUSH，GRAY_BRUSH，LTGRAY_BRUSH，HOLLOW_BRUSH，NULL_BRUSH和WHITE_BRUSH。所有这些都是实心的。HOLLOW_BRUSH和NULL_BRUSH是涉及同一事物的两种不同方式：都是一枝什么都不画的画刷。WHITE_BRUSH是DC的缺省画笔。像库存画笔一样，库存画刷能使用CGdiObject::CreateStockObject创建，或使用CDC::SelectStockObject直接选入DC。使用三个灰色库存画刷的好处是，这些画刷总是用实心的非杂色。创建一个你自己的灰色画刷，如果显示硬件太受限制以致于不能直接显示灰度浓淡，那么Windows就可能混合画刷颜色。

当使用杂色的实心画刷或影线画刷时，你应该知道的DC的一个独特属性是画刷的初始点。当Windows用影线或杂色画刷图案填充一个区域时，它在被影响区域内水平地和垂直地层叠8×8像素图案。由于缺省，这些图案的初始点，最好也认为是画刷的初始点，是对应于DC左上角的屏幕像素。这样一来的实际效果是，着眼于矩形的边界，画在从该初始点右下100像素处开始的矩形中的图案，将被排列得略微不同于从其左或右几个像素处开始的矩形(见图3)。在很多应用程序中，这并不算什么，用户可能不会注意到画刷排列上的这样细小的差别。然而，在一些情况下，这就是大问题。假设你正在用影线画刷填充一个矩形，并且通过反复地擦除它和向右或向左重画一个像素来动画该矩形的运动。如果你不在每次重画前把画刷的初始点重新设置到相对于该矩形位于同一位置的一些点，那么影线图案将在关联矩形区运动，产生一个令人非常不满意的结果。