**高DPI适配方案**

随着便携设备和高DPI显示器的普及，Windows自身对高DPI的支持也在不断更迭。对应用程序而言，如何与操作系统配合，实现DPI适配以达到最佳的显示效果愈发重要，我们平日常用的应用软件也陆续支持了DPI适配。本文将从概念原理、Windows DPI支持发展开始描述，并给出DPI适配方案的设计与实现.

**一、     背景与遇到的问题**

假设在一个19英寸的显示器上显示10像素x20像素的文字。若分辨率为1920x1080尚可正常阅读，但如果这是3840x2160分辨率的4K高分屏，便小到难以看清，相当于只有原来的1/4。此时我们则使用DPI来进行缩放调节，将该文字调整到对应高分辨率下的大小进行显示，如20像素x40像素，获得合适的视觉效果。

DPI（Dots Per Inch，每英寸点数）用于描述图像每英寸中含有多少像素点，故等大的图像，DPI越高，包含的像素点数越多，图像内容越丰富，看起来便越清晰。

在Windows下，屏幕单位长度中有多少像素点由实际屏幕物理大小与分辨率决定，而常说的DPI则成了缩放的参考值。如默认DPI（系统缩放设置为100%）为常量96，系统缩放设置为125%时对应DPI为96x125%=120。系统本身与应用程序均以此DPI作参照，实现缩放。

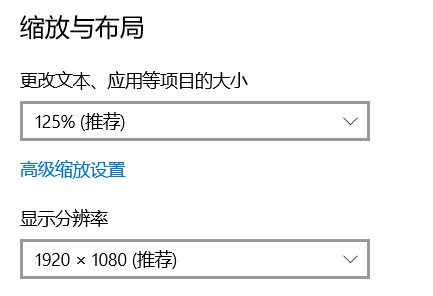


图1.1 Win10中缩放相关设置

了解背景以后，看似只要将各长度对应DPI进行等比缩放就能实现适配，比如原本需要绘制100x200单位矩形，在125%缩放比例下绘制125x250单位即可，并且系统已经有内置的缩放机制供默认使用。然而实际情况往往复杂得多，从Windows对DPI支持不断的更迭便可看出，主要有以下问题：

Q1：系统缩放容易导致应用程序模糊

若使用系统自带的DPI适配（DWM Scaling），应用程序一般无需做任何更改。系统始终告知应用程序DPI设置为默认值96（100%），但应用程序进行显示相关操作时，系统内部将为其自动应用DPI缩放。如同位图缩放，这样的缩放容易导致应用程序显示模糊，尤其是字体。下图是MSDN中对比示意：

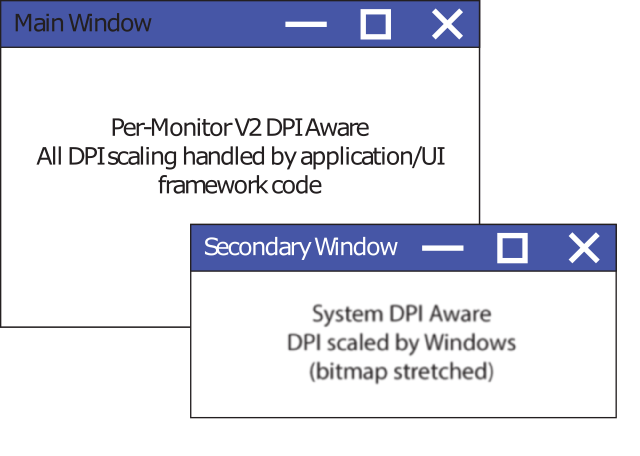


图1.2 MSDN 混合DPI缩放模式下，应用程序/UI框架缩放与系统缩放对比

以下是AlleyWind应用程序系统缩放（模糊）与应用程序适配DPI缩放（不模糊）的对比，系统缩放明显模糊、浑厚，应用程序缩放则显得清晰、锐利：



图1.3 系统缩放（125%）

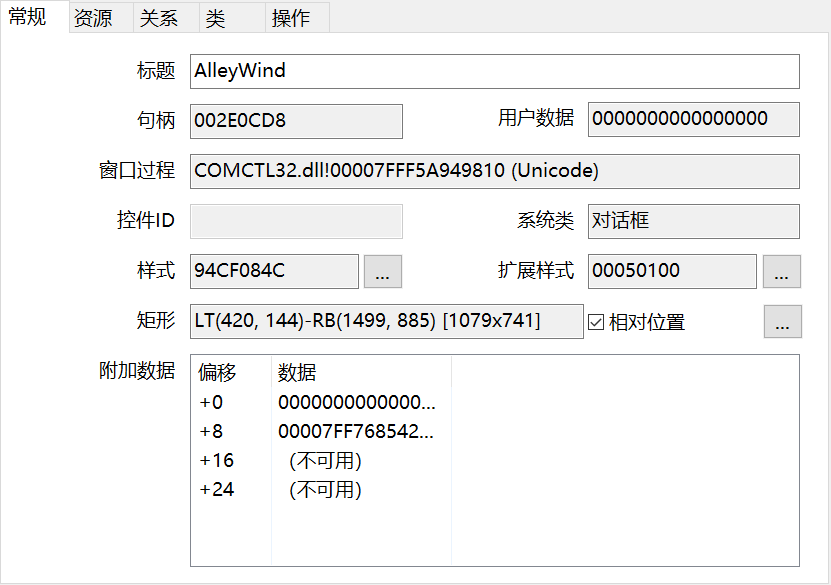


图1.4应用程序适配DPI（125%）

Q2：若应用程序自己控制缩放，界面编程繁复

需要应用程序在进行绘制时，自己将所有要绘制的内容全部按DPI进行缩放，并且依赖代码进行计算和控制。如今不论B/S还是C/S，我们都努力将界面与代码抽离开，而不是用代码去实现UI。

Q3：需要响应DPI变化，实时调节缩放

目前绝大多数支持DPI适配的常用软件，在DPI更改时都能正确调整布局，不需重新启动才生效。随着高DPI显示器的普及，这样的场景不再仅仅局限于用户手动更改系统DPI设置这样少见甚至不必考虑的场合——窗口游走于不同DPI显示器之间也会出现。所以比起在开局一次性缩放，如今更需要设计成能灵活响应DPI变化的模式。

**二、     Windows的支持**

1.Windows XP

早在Windows XP，便提供了缩放设置选项供用户设置。应用程序可以通过API获取当前DPI，补足系统缩放的缺陷。

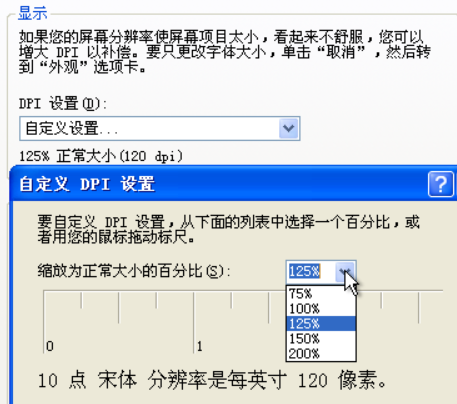


图2.1Windows XP中DPI设置

使用GetDeviceCaps获取传入DC的DPI设置，LOGPIXELSX与LOGPIXELSY对应X与Y方向的DPI，目前二者始终相等，默认值（缩放为100%）是96（Windows SDK里定义为USER\_DEFAULT\_SCREEN\_DPI）。

此时，面临上述三个问题，并且系统缩放效果不佳。

2.Windows Vista与Windows 7

随着DWM的引入，应用程序的系统缩放由其实现（DWM Scaling），系统缩放效果相较于XP好了很多。并且开始增加相关DPI函数，供应用程序自己实现缩放。

可以通过清单文件或者新增的SetProcessDPIAware函数设置应用程序DPI缩放模式，告诉系统本程序已考虑了DPI缩放问题，不要应用系统缩放。增加IsProcessDPIAware函数以获取此项设置。

Windows 7开始，DPI设置成为了每用户独立的设置。至此，虽然仍面临上述三个问题，但系统缩放效果得以改善，并且应用程序可以选择使用系统缩放还是自己实现。

3.Windows 8.1

支持Per-Monitor DPI缩放模式，不同显示器可以拥有不同DPI，新增GetDpiForMonitor函数获取特定显示器的DPI设置。由于多了DPI模式，原SetProcessDPIAware和IsProcessDPIAware的TRUE和FALSE已无法准确指明，并且相关DPI变更消息机制也应运而生。

可以使用清单文件或者新增的SetProcessDpiAwareness函数设置应用程序DPI缩放模式，告诉系统本程序使用Per-Monitor DPI缩放模式。与之前相同，系统不再为应用程序进行缩放，但当DPI变更时，操作系统会通知应用程序，让应用程序适时调整，实现窗口在不同DPI下来去自如。新增的GetProcessDpiAwareness函数可以获取此项设置。

除了新增上述函数支持Per-Monitor模式的设置，还新增了窗口消息以落实该机制。当顶层窗口DPI变更时，如用户修改了DPI设置，或窗口移动到了不同DPI的显示器上，操作系统会为其下发WM\_DPICHANGED消息，告诉窗口新的DPI与建议调整的位置和大小。看似提供了近乎保姆级别支持，但该消息只为顶层窗口派发，顶层窗口内部仍需自己进行计算和缩放子窗口。

至此，上述问题中DPI变化的场景得以有机会优雅地面对，多显示器不同DPI的场景需要得以满足。

 4.Windows 10 1607

DPI缩放模式设置从进程独立变为线程独立——从此同一个进程的不同线程可以拥有不同的DPI缩放模式。看似能让应用程序更自由地控制DPI缩放，但随之而来的又是一对新API——SetThreadDpiAwarenessContext与GetThreadDpiAwarenessContext。此外还新加入了GetAwarenessFromDpiAwarenessContext、GetDpiForWindow、GetDpiForSystem、GetSystemMetricsForDpi、SystemParametersInfoForDpi等函数供应用程序使用，看上去似乎更便捷，但考虑兼容性后应该就不会有这样的错觉了。还增加EnableNonClientDpiScaling函数为顶层窗口提供非客户区DPI缩放支持，这个很快便被即将到来的Per-Monitor v2模式吞并。

此外，微软还为自家的UWP与WPF提供了原生支持。至此，相比Windows 8.1变更也不少，但未必可圈可点。可以结合实际情况考虑，沿用之前的模式。

5.Windows 10 1703

随Win10 1703而来的，是Per-Monitor v2 DPI缩放模式，Win32应用程序DPI适配也终于完善。不出意外，又引入了新的DPI缩放模式读写API，SetProcessDpiAwarenessContext与GetProcessDpiAwarenessContext，与1607新增针对线程DPI模式的函数共用相同一套枚举值，函数命名也相似。

相较于Win8.1的Per-Monitor (v1)，Per-Monitor v2可谓众望所归：

WM\_DPICHANGED不仅发送给顶层窗口，也发送给其子窗口，还新增WM\_DPICHANGED\_BEFOREPARENT、WM\_DPICHANGED\_AFTERPARENT消息满足子窗口时序需要，新增WM\_GETDPISCALEDSIZE消息满足非线性缩放需要

增加或改进系统对话框、菜单与通用控件（ComCtl32.dll V6.0）对DPI的支持和响应

系统自动缩放非客户区，不需调用EnableNonClientDpiScaling

主题资源显示自动缩放

有了上述支持，经典Win32应用程序甚至只需在清单文件中声明或一行“SetProcessDpiAwarenessContext(DPI\_AWARENESS\_CONTEXT\_PER\_MONITOR\_AWARE\_V2);”设置DPI缩放模式为Per-Monitor v2，即可完美适配DPI，并响应DPI的变化。因为对话框及其菜单、内部的Win32通用控件都能自动响应并进行DPI缩放，应用程序无需干预。当然也可以使用SetDialogDpiChangeBehavior、SetDialogControlDpiChangeBehavior这样的函数修改默认行为，但一般少有需要，因为实际效果已经很理想。随着Win32通用控件对高DPI的适配，Windows Forms也在1703中提供了有限的支持。

至此，Windows基本完善对高DPI的支持，此版本具有重大意义。从系统层面解决了上述各问题，并满足相关使用场景，应用程序生态环境基本完善。从系统和应用程序角度上而言，通用控件原生支持了DPI适配，直接赋予Win32程序DPI适配的能力，甚至简单到应用程序开发者开启一个开关。UWP、WPF、Windows Forms、Direct2D、Win32等框架支持也顺势健全。

6.Windows 10 1803与1809

1703引入全新模式一举完善DPI的支持之后，1803与1809更多的是修订与补充，这里归并到一起简要描述。

1803主要引入新的API让同一线程的不同窗口可以具有不同DPI缩放模式，相关函数有如SetThreadDpiHostingBehavior、GetThreadDpiHostingBehavior、GetWindowDpiHostingBehavior。

1809主要增加DPI\_AWARENESS\_CONTEXT\_UNAWARE\_GDISCALED模式，该模式从GDI层面改善不支持DPI适配的应用程序系统缩放效果，对应应用程序兼容性的“系统（增强）”。

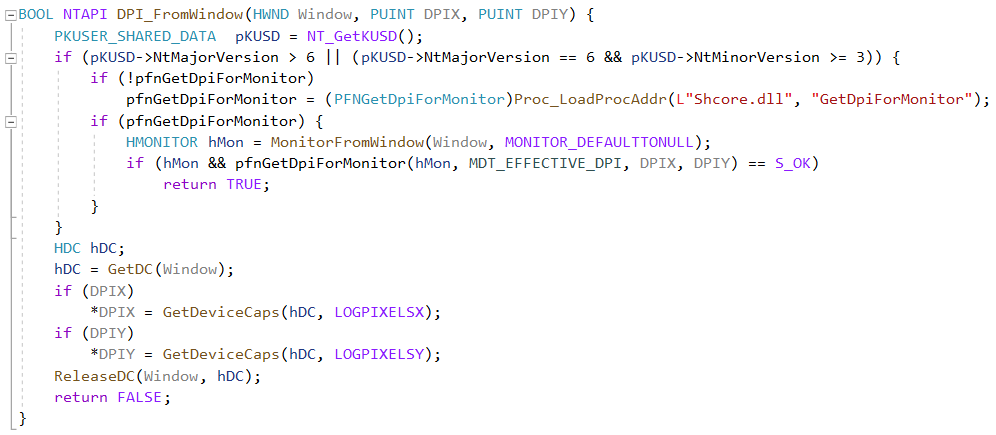
**三、     DPI适配方案的设计与实现**

在Windows的套路中兜兜转转，我们能否为自己的Win32应用程序走出一条自己的路，在更多的场景下获得更佳的效果，支持更多版本的系统？

设计时对系统兼容的考虑，我们不妨从Win8.1前后两个角度来看。Win8.1之前DPI设置无法独立到单显示器，所以也可不考虑DPI变化的场景。若必须考虑用户变更DPI，可以自行增加轮询机制。Win8.1开始，DPI支持针对到单个显示器，但系统也提供了WM\_DPICHANGED消息通知应用程序，我们只需要注意在收到这个消息时进行DPI缩放即可。

简而言之，我们的应用程序使用Per-Monitor (v1) DPI缩放模式，在程序初始化与收到WM\_DPICHANGED消息时进行DPI缩放操作，即可在最大程度上兼容各个操作系统。剩下的问题便是DPI缩放逻辑，主要包含窗口缩放、字体适配、图像资源缩放三个方面。

首先我们需要根据窗口获取DPI。



// 获取GetDpiForMonitor函数指针.

*HMODULE* hInst = nullptr;

hInst = *LoadLibrary*(L"Shcore.dll");

PFNGetDpiForMonitor pFNGetDpiForMonitor = *NULL*;

pFNGetDpiForMonitor = (PFNGetDpiForMonitor)*GetProcAddress*(hInst, "GetDpiForMonitor");

// 获取实际屏幕的DPI.

*UINT* iDPIX = 0;

*UINT* iDPIY = 0;

*HMONITOR* hMon = *MonitorFromWindow*(hWnd, *MONITOR\_DEFAULTTONULL*);

pFNGetDpiForMonitor(hMon, MDT\_EFFECTIVE\_DPI, &iDPIX, &iDPIY);

// 该方式获取的值不变 都为96.

*HDC* hdc = *GetDC*(hWnd);

int iX = *GetDeviceCaps*(hdc, *LOGPIXELSX*);

int iY = *GetDeviceCaps*(hdc, *LOGPIXELSY*);

*ReleaseDC*(hWnd, hdc);

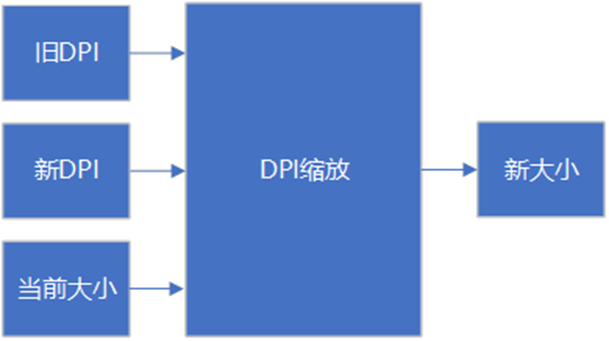
注意，这里首先考虑Per-Monitor模式，此模式下应调用GetDpiForMonitor获取窗口所在显示器的DPI，其返回值会随DPI设置的变化而变化；传统的GetDeviceCaps作为备选方案，它的返回值不随DPI设置相应变化。其次，X与Y始终相同，可不像以上二者兼顾。

接下来需要缩放窗口。目前使用较多的DPI缩放逻辑为下图所示：



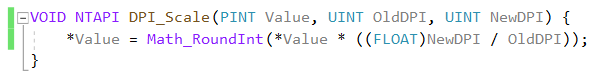
核心计算公式为NewValue = InitValue × DPI。如某长度为100单位，在150%缩放比例下新长度应为100 × 150% = 150。

虽然简单，但问题也很明显——在DPI变化的场景下，初始大小难以再直接获取。当然可以通过DPI的变化进行换算，此时与下面介绍的逻辑殊途同归：



核心计算公式为NewValue = OldValue × NewDPI ÷ OldDPI。如某长度为100单位，在150%缩放比例下新长度应为100 × 150% ÷ 100% = 150。

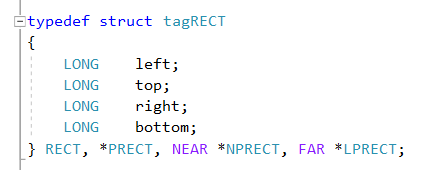
实现如下：



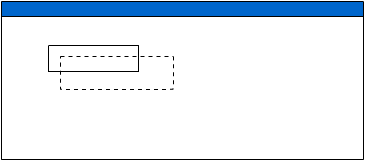
       结果记得四舍五入：

C:\Users\appeon\Desktop\[原创]Win32应用程序DPI适配的设计与实现-编程技术-看雪论坛-安全社区_安全招聘_bbs.pediy.com_files\853701_R37KEP5HY8CAPKR.jpg

我们据此缩放顶层窗口及其子窗口的位置和大小等值。在Windows下，窗口位置与大小一般使用RECT结构体描述：

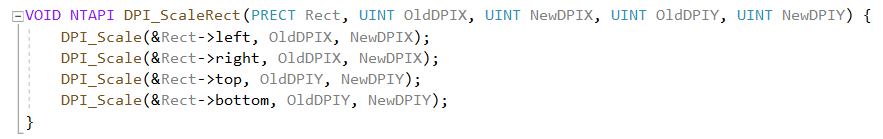


       四个成员值表示窗口区域的左、上、右、下。对于顶层窗口的子窗口而言，其位置相对于父窗口，直接对这四个值进行缩放，然后传给SetWindowPos函数即可。实现效果如图所示：

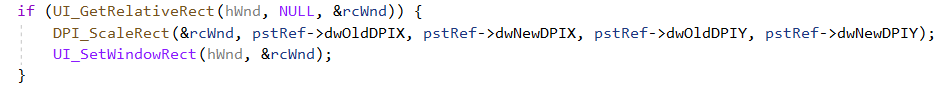


       DPI由100%调整到125%，则子窗口由实线示意的原位置缩放到虚线示意的新位置，其左、上、右、下均增加25%，反之亦然。

相关实现：

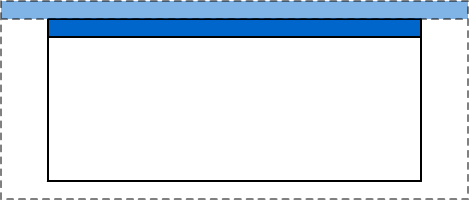


以上将RECT结构体的四个成员值根据新旧DPI进行缩放。



       以上获取窗口的相对位置，对其位置和大小进行缩放，然后应用。

       顶层窗口自身的缩放逻辑有别于其子窗口。由于顶层窗口的位置相对于整个屏幕，我们需要考虑其新位置是否合适。如若仍按照上述逻辑缩放，本处于屏幕正中的顶层窗口缩放后位置必然偏移，不再位于正中：若DPI增大，则造成顶层窗口向屏幕右下方偏移；若DPI减小，则造成顶层窗口向屏幕左上方偏移。处理逻辑很简单，根据DPI变化，将水平方向缩放产生的差平摊给左、右，垂直方向缩放产生的差平摊给上、下。实现效果如图所示：



以下是实现：



以上还考虑了缩放后左上超越屏幕边界的情况（尤其是标题栏出界可能会很不舒服），此时将其校正回左上角，但未考虑右下角，可作改进。

窗口缩放告一段落，接下来是字体缩放。若想提高性能减少资源开销，子窗口也不需要搞特殊时，可直接由顶层窗口接管各子窗口的字体对象，将统一字体应用到各子窗口。Windows对话框本身也有这样的机制，初始化时将选择合适的字体，应用到各个控件上。若不在意多一些GDI字体对象，或者子窗口的字体可能不同，则单独获取子窗口字体信息，缩放字体大小，然后创建新的GDI字体对象替换原对象。

不论哪者，都有不足，而且需要考虑GDI字体对象泄露和意外释放了其它地方仍在使用的GDI字体对象。比如我们经缩放创建了新的GDI字体对象，应用到子窗口时若不释放其原有GDI对象，可能造成泄露；若释放，可能该对象亦在其它地方被使用，造成影响。若有更好的方式欢迎多多讨论和指点指点，此设计仍需改进。

相关可能用到的API或消息有WM\_GETFONT（获取窗口字体）、WM\_SETFONT（设置窗口字体）、GetObject（获取字体对象信息）、CreateFontIndirect（创建字体）等。

整个响应DPI缩放流程实现如下：



当收到WM\_DPICHANGED消息时，先调整自身位置与大小，然后调整字体大小，替换旧字体对象。最后遍历子窗口，调整各个子窗口的位置与大小，为其应用新字体对象。

最后是图像资源缩放，如加载位图、光标、图标等。缩放思路没什么特别，只是要在相关操作时留意，加入缩放逻辑即可。也不用担心将固定大小的资源缩放产生较严重的失真，因为最终物理显示总是在期望的大小附近。

参考资料：

[《MSDN - DPI and Device-Independent Pixels》](https://docs.microsoft.com/en-us/windows/win32/learnwin32/dpi-and-device-independent-pixels)

[《MSDN - High DPI Desktop Application Development on Windows》](https://docs.microsoft.com/en-us/windows/win32/hidpi/high-dpi-desktop-application-development-on-windows)

[《Windows Blogs - Improving the high-DPI experience in GDI based Desktop Apps》](https://blogs.windows.com/windowsdeveloper/2017/05/19/improving-high-dpi-experience-gdi-based-desktop-apps/)