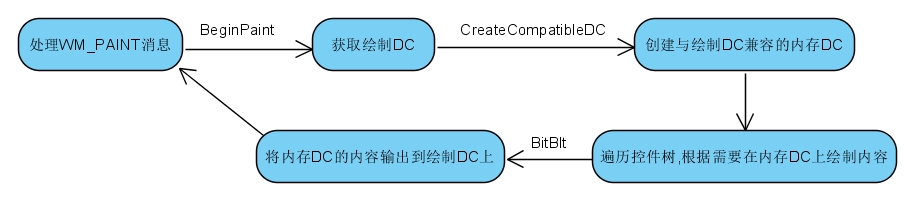
转载请注明出处：http://www.cnblogs.com/fangkm/p/3943896.html

**常见的UI库的绘制逻辑**

任何一个成熟的界面框架都有一个相当复杂的结构，消息循环的处理、控件的布局与绘制、焦点的管理以及资源的存取等等，Chromium里的界面框架也不例外，尤其采用的MVC设计方式更是增添了代码结构的复杂度。这里并不打算讨论Chromium的界面框架，本文感兴趣的只是Chromium的UI绘制部分，确切地说应该是引入Aura架构之后控件渲染的硬件加速支持。

在常见的DirectUI实现（Windows平台）中，绘制逻辑一般如下：最外层带HWND句柄的的Window在接收到WM\_PAINT消息时，会通过BeginPaint函数获得一个窗口对应的绘制DC，然后根据这个绘制DC创建一个相应的内存DC，之后遍历Window内自绘的控件（呈树型结构），每个控件计算需要更新区域与自己区域的交集，在该内存DC上绘制交集区域对应的控件部位，最后由BitBlt系列的API将内存DC拷贝到绘制DC上，从而完成一整套的渲染逻辑。没有语言功底的人用文字来描述流程逻辑，会显得相当苍白无力，还是附上一张简单的流程图吧。

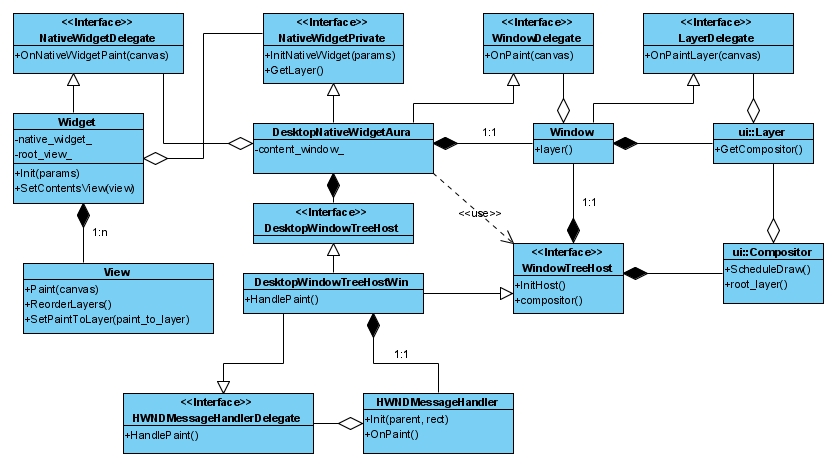


整体流程结构上很清晰明了，但是实现起来很是繁琐，尤其是处理控件的层级、可视性、绘制区域的计算等逻辑。Chromium中UI框架的软件渲染流程也大致是这个流程，只是引入了硬件渲染后，增加了合成层的概念，为了结构上的一致性，软件渲染流程也经过cc层，路由一大圈，最终绘制在软件渲染提供的OutputSurface上，具体可参见SoftwareOutputDeviceWin实现，Chromium的软件渲染部分在探究CC库的时候会深入研究。

**引入Aura框架后的Chromium UI库的绘制逻辑**

          在普遍的DirectUI库实现中，一般都是采用软件渲染，也就是通过一系列的GDI函数在DC上绘制，软件渲染一旦遇到刷新频率比较高的情况下，比如视频画面、游戏画面等，绘制效率就力不从心了，流畅度体验下降的厉害，所以播放视频或者游戏都用GPU进行硬件渲染，然而Windows下硬件渲染一般都需要提供窗口句柄HWND，比如DirectShow中的渲染组件VMR Filter， 但是在DirectUI框架中加入一个带HWND的子窗口是种很要命的情况，这样就完全破坏了自绘体系，任何完全自绘的控件都无法位于HWND子窗口之上，焦点逻辑、消息路由等体系都会遭受不同程度的破坏。退一步讲，就算一个成熟的DirectUI库对包容子HWND窗口逻辑做的足够好，那试想一下如果一个ui界面中如果有多处需要HWND的情况,那也是很糟糕的事。总之,在我看来,在DirectUI框架中加入带HWND的子窗口完全就是悖论，就是对实际应用场景的一种妥协。还有，DirectUI的一大特色就是完全自绘，这样就容易开发出绚丽多彩的界面，酷炫界面当然少不了动画逻辑，动画的渲染如果能用上GPU也是一大助力。废话了这么多，总之，能够支持硬件渲染对DirectUI库来说是很有必要的。

          非常荣幸，Chromium UI库就支持硬件渲染，为此还引入了Aura框架结构来管理UI的渲染底层。Chromium UI框架渲染逻辑涉及到的结构图大致如下：



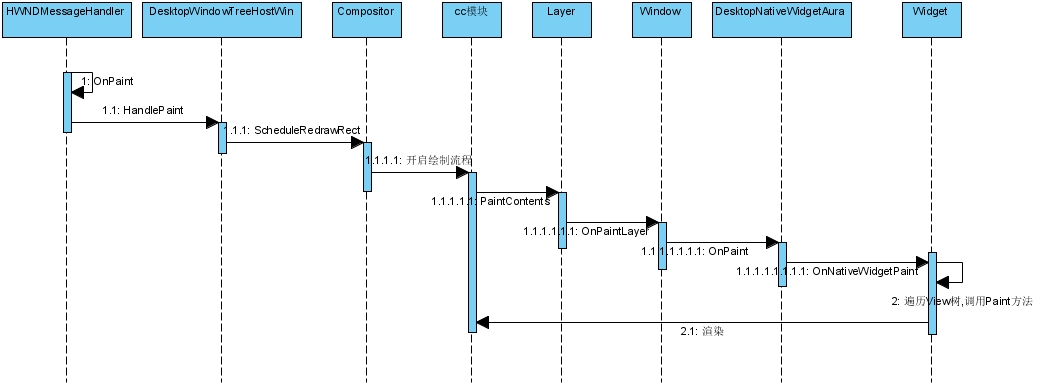
在介绍渲染流程之前之前，有必要理解一下Aura结构，依据我的理解， Aura中的Window类应该就是Aura框架中的窗口的概念，Window同样采取树型组织结构，从而WindowTreeHost类的作用就不难理解了，就是作为Window树的宿主，WindowTreeHost内部创建一个顶层的Window作为所有的Window树的根节点。WindowTreeHost在不同平台下有着不同的实现，在windows下的实现类是DesktopWindowTreeHostWin， 其维护一个HWNDMessageHandler对象，用来处理windows平台下的窗口的创建、消息处理等逻辑，HWNDMessageHandler从 WindowImpl派生，负责处理庞大的Windows消息处理，并将处理逻辑通过HWNDMessageHandlerDelegate接口传递给DesktopWindowTreeHostWin处理，比如，WM\_PAINT消息的处理逻辑就是通过这种方式委托给DesktopWindowTreeHostWin的HandlePaint函数来处理。

在引入Aura之前，Widget类直接HWNDMessageHandler进行对接，引入Aura之后，Widget与HWND脱离开来，不过逻辑上仍然作为对外开发接口中的顶层窗口。

views框架为Widget配备一个DesktopNativeWidgetAura对象作为与Aura层的适配，如图所示，DesktopNativeWidgetAura逻辑上只维护了一个Window对象，即content\_window\_成员，该Window对应整个的Widget窗口层级，也就是说整个Widget包括其内部的View树都属于Aura中的一个窗口层。

          在Aura体系中，Window和Layer为一一对应的关系，从而很好地表示出每个窗口就是一个渲染层的概念。当渲染引擎渲染之前，需要收集每个Layer的绘制逻辑时，Layer对象通过LayerDelegate接口的OnPaintLayer方法来将逻辑委托给与其对应的Window类。同样，DesktopNativeWidgetAura为Widget维护了一个Window对象，它就有义务从Window对象中将绘制逻辑通过WindowDelegate接口承接过来，然后通过NativeWidgetDelegate接口将该逻辑抛给Widget对象，Widget对象响应OnNativeWidgetPaint函数，遍历自身的View树，往画布上添砖加瓦。

          在这里需要补充一下，Widget的View也可以持有Layer对象，刚才的结构图中实在不便标出，View同样实现了LayerDelegate接口，控件可以根据需要创建属于自己的Layer层，并将其添加到DesktopNativeWidgetAura成员content\_window\_对应的Layer层的子节点中（再次嵌套补充一下，这里说的Layer是ui层的Layer，添加到其子节点，为的是添加到与ui层Layer对应的cc层的Layer的树层级中，cc层的Layer树才是真正的渲染树，ui层的Layer只是为了方便Aura访问做的一层适配。有点绕了，以后讨论cc库的时候再细说吧）。通过调用View的SetPaintToLayer方法，View就会创建自己的Layer层。此时View直接实现OnPaintLayer方法来处理绘制即可，而且绘制的层级在不创建Layer的View控件之上。如果要创建一个显示视频的控件，可以考虑创建一个属于自己的Layer层，从而在独立的层上绘制视频数据。啰嗦了这么多，还是配上一张序列图啊，一图抵千言：



cc库的逻辑以后再开始重点研究，这里姑且就当黑盒处理。这个流程中，绘制的驱动由windows消息WM\_PAINT的响应来驱动，这是典型的软件渲染流程，另外可以调用View控件的SchedulePaint方法来重绘该控件，重绘的流程是找到与View绑定的Layer对象，如果没有绑定则一直向上找，最顶层的Widget可以通过GetLayer方法访问到DesktopNativeWidgetAura成员content\_window\_对应的Layer。找到Layer后调用Layer的SchedulePaint函数，从而驱动Compositor出发绘制流程。

本文就先介绍到这里，其实这里面还有很多逻辑可扒，比如绘制流程一旦触发，怎么确定单个Layer或View是否需要重新绘制等。时间有限，精力有限，还是留点激情稍后去研究CC模块为好。