**前言**

谈起“消息机制”这个词，我们都会想到Windows的消息机制，系统将键盘鼠标的行为包装成一个Windows Message，然后系统主动将这些Windows Message派发给特定的窗口，实际上消息是被Post到特定窗口所在线程的消息队列，应用程序的消息循环再不断的从消息队列当中获取消息，然后再派发给特定窗口类的窗口过程来处理，在窗口过程中完成一次用户交互。其实，WPF的底层也是基于Win32的消息系统。

1. **WPF线程模式**

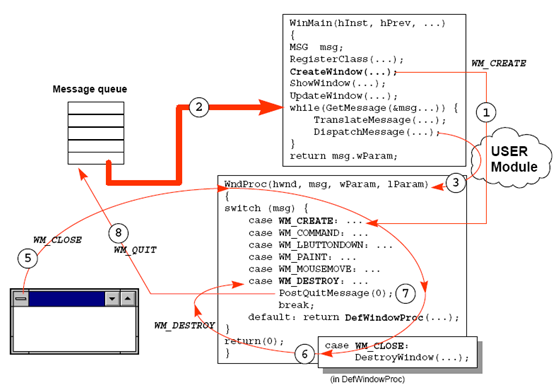
通常，WPF 应用程序从两个线程开始：一个用于处理呈现，一个用于管理 UI。呈现线程有效地隐藏在后台运行，而 UI 线程则接收输入、处理事件、绘制屏幕以及运行应用程序代码。

UI 线程对一个名为 [Dispatcher](http://msdn.microsoft.com/zh-cn/library/system.windows.threading.dispatcher.aspx) 的对象内的工作项进行排队。[Dispatcher](http://msdn.microsoft.com/zh-cn/library/system.windows.threading.dispatcher.aspx) 基于优先级选择工作项，并运行每一个工作项，直到完成。每个 UI 线程都必须至少有一个 [Dispatcher](http://msdn.microsoft.com/zh-cn/library/system.windows.threading.dispatcher.aspx)，并且每个 [Dispatcher](http://msdn.microsoft.com/zh-cn/library/system.windows.threading.dispatcher.aspx) 都只能在一个线程中执行工作项。

1. **应用程序启动过程**

谈到WPF的消息，首先应该知道DispactherObject以及Dispatcher在WPF系统中的作用。

作为一个Presentation的基架，WPF的使命就是要编写图形化的操作界面。而在Windows操作系统上，图形化界面是建立在消息机制这个基础上的，那么创建一个窗口，要经历哪些步骤呢？



　　1. 创建窗口类。 WNDCLASSEX wcex； RegisterClassEx(&wcex);

　　2. 创建窗口。CreateWindow(…); ShowWindow(…); UpdateWindow(…);

　　3. 建立消息泵。

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

　　{

　　　　TranslateMessage(&msg);

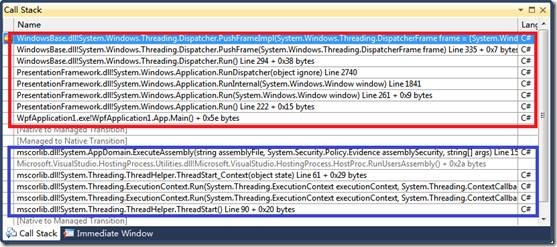
　　　　DispatchMessage(&msg);

　　}

　　 打个比方，我们在一个自动化的厂房里生产设备。基于正规，我们会首先定义好该设备的模板，这就是创建窗口类，这里”类”更多表示类别的意思。模板定义完毕，我们可以正式生产设备了，这就是创建窗口，这个CreateWindow的时候会通过字符串来匹配到我们定义的模板（窗口类）。创建成功后，我们要让设备动起来，就要像人一样，体内一定要有类似于血液的流传机制，把命令传达到设备的各个部分，这就是消息泵，这个泵就像我们的心脏一样，源源不断的通过GetMessage并Dispatch来分发血液（消息）。既然我们通过消息来对设备下达指令，那么就要有消息队列来存储消息，在Windows中，线程为基本的调度单位，这个消息队列就在线程上，当循环使用GetMessage时，就是在当前线程的消息队列中任劳任怨的取出消息，然后分发到对应的窗口中去。

那么具体到WPF，它又是一个怎么样的情况，WPF引入了Dispatcher的概念，这个Dispatcher的主要功能类似于Win32中的消息队列，在它的内部函数，仍然调用了传统的创建窗口类，创建窗口，建立消息泵等操作。Dispatcher本身是一个单例模式，构造函数私有，暴露了一个静态的CurrentDispatcher方法用于获得当前线程的Dispatcher。对于线程来说，它对Dispatcher是一无所知的，Dispatcher内部维护了一个静态的List<Dispatcher> \_dispatchers, 每当使用CurrentDispatcher方法时，它会在这个\_dispatchers中遍历，如果没有找到，则创建一个新的Dispatcher对象，加入到\_dispatchers中去。Dispatcher内部维护了一个Thread的属性，创建Dispatcher时会把当前线程赋值给这个Thread的属性，下次遍历查找的时候就使用这个字段来匹配是否在\_dispatchers中已经保存了当前线程的Dispatcher。

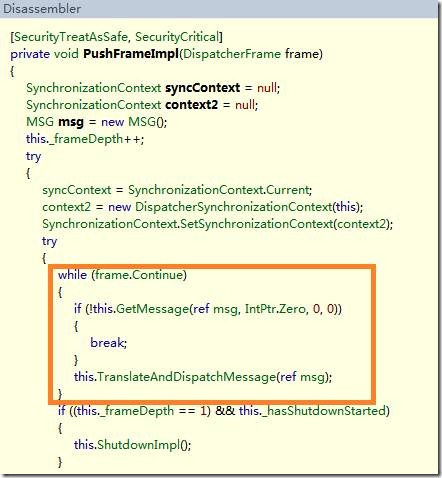
　　 那么这个创建窗口，建立消息泵又是什么时候被调用的呢？在Dispatcher内部，维护了一个HwndWrapper的字段，在Dispatcher的构造函数中，调用了HwndWrapper的构造函数，这个创建窗口类，创建窗口就是在这个函数中被调用的。这里实际的类是MessageOnlyHwndWrapper，这个Message-Only，是Windows编程中常用的伎俩，创建一个隐藏窗口，仅仅用来派发消息。那么循环读取消息的消息泵又是什么时候建立起来的呢？WPF的Application在启动之后都走了哪些逻辑：



通过调用堆栈可以看出，蓝色的部分是启动了一个线程，VisualStudio在Host的进程当中运行当前应用程序；红色的部分是从Application.Main函数开始执行，经过几个函数到达Dispatcher.Run()，最后到达Dispather.PushFrameInpl()方法。

　　 WPF大部分的对象都是从DispatcherObject派生的，从这里派生的对象具有一个明显的特征，那就是：修改对象时所在的线程，和创建对象时所在线程必须为同一个线程，这就是微软所谓的线程亲缘性（Thread affinity）的最简单理解。那么谁能保证线程亲缘性呢？那就是Dispacher了。从DispatcherObject派生的类型继承三个重要的成员：Dispatcher属性，CheckAccess(), VerifyAccess()方法。其中后面两个方法就是检验线程亲缘性的。按照WPF的实现，如果你自己定义了个WPF的类型，并且是DispatcherObject的子类，你就必须在public的成员定义的逻辑开始处，调用base.Dispatcher.VerifyAccess()，检验线程亲缘性。那么Dispatcher到底还做了什么事情呢？

那么一个Application在Run之后,为什么要调用Dispatcher.Run()呢，他做了些什么事情你？如果通过Reflector仔细查看Application.Run()，你会发现里面实际起作用的代码并不多，最后都是Dispatcher.Run在做事情。那么一个Application启动之后，按照以前对Win32的消息机制的理解，当应用程序启动后，必须进入消息循环，对于WPF，也是一样的。那么WPF应用程序是在什么地方进入消息循环呢？其实这就是Dispatcher.Run()做的事情。查看上图最后一步Dispacther.PushFrameImpl()的代码，你会看到有下面的一段代码：



很明显，橙色的部分是一个循环，跟Win32编程碰到的消息循环很像了，这就是WPF应用程序进入了消息循环。循环调用GetMessage方法从当前线程的消息队列当中不停的获取消息，取出一个msg之后，交给TranslateAndDispatchMessage方法Dispatch到不同的窗口过程去处理。这样以来，任何需要应用程序处理的消息通过这个过程，被不同的窗口处理了，应用程序就动起来了。

1. **消息处理过程**

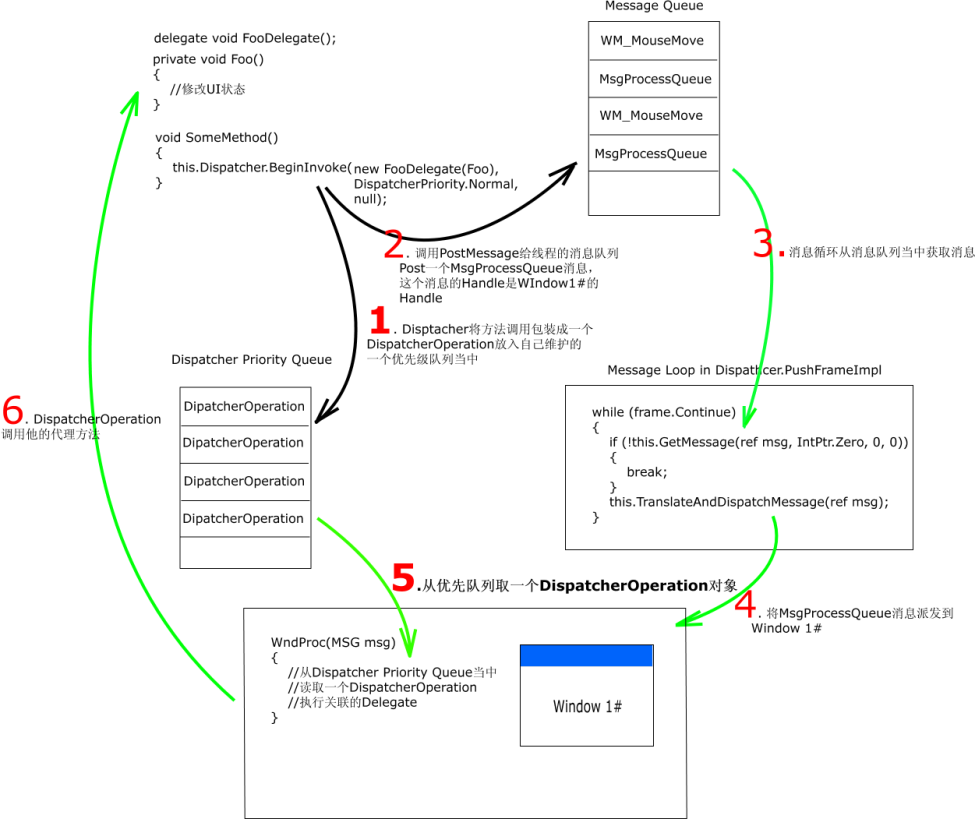
对于Windows系统来说，它是一个消息系统，消息系统的核心就是窗口。对于WPF来说也是如此。

运行WPF应用程序所在的线程就是WPF所谓的UI线程，在Application.Run之后，调用Dispatcher.Run时会检查当前线程是否已经存在了一个Dispatcher对象，如果没有就构造一个，在这里，一个线程对应一个Dispatcher。因此，WPF的对象在获取this.Dispatcher属性时，不同对象取的都是同一个Dispatcher实例。另外，前面提到的“消息循环”，“消息队列”等都是Win32应用程序的概念，我们知道，提起这些概念，必然会跟Win32的“窗口”，“Handle”，“WndProc”之类的概念离不开，那么WPF里面究竟有没有“窗体”，“Handle”，“WndProc”呢？

通常情况下，一个WPF应用程序在运行起来的时候，后台会创建5个Win32的窗口，帮助WPF系统来处理操作系统以及应用程序内部的消息。在这5个窗口中，只有一个是可见的，可以处理输入事件与用户交互，其他4个窗口都是不可见的，帮助WPF处理来自其他方面的消息。

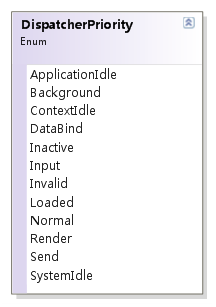
在Application的构造函数调用基类DispatcherObject的构造函数的时候，会创建一个Dispatcher对象，在Dispatcher的私有构造函数当中，会创建一个不可见的窗口（Window 1#），来进行消息处理，主要实现WPF线程模型的异步调用。

WinForm下，我们通常为了使一些花费较多时间的方法调用不影响UI的响应，会将这个操作分为很多步，然后使用BeginInvoke调用每一步，这样UI响应就不会被阻塞。BeginInvoke的本质是往消息队列当中PostMessage，而不是直接调用，与此同时，UI行为（MouseMove）导致系统也往消息队列当中PostMessage更新UI，但由于彼此花费的时间很短，就感觉两个消息是被同时处理似的，界面就不会觉得被阻塞了。WPF同样面临这样的问题，在这里Window 1#起着至关重要的作用。通过下面一副图我们来看看Window 1#在做什么事情？



WPF也是通过BeginInvoke来解决的，而Wpf的BeginInvoke是在Dispatcher上面暴露了，因为整个消息系统都是Dispatcher在协调。从上面图可以看出Dispatcher在调用BeginInvoke之后所经历的流程，最终是什么时候Foo()被真正执行的。

**第一步**，就是将调用的Delegate和优先级包装成一个DispatcherOperation放入Dispatcher维护的优先级队列当中，这个Queue是按DispatcherPriority排序的，总是高优先级的DispatcherOperation先被处理。WPF定义了这个优先级DispatcherPriority，有：



**第二步**，往当前线程的消息队列当中Post一个名为MsgProcessQueue的Message。这个消息是WPF自己定义的，见Dispatcher的静态构造函数当中的[\_msgProcessQueue](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://WindowsBase:3.0.0.0:31bf3856ad364e35/System.Windows.Threading.Dispatcher/_msgProcessQueue:Int32) = [UnsafeNativeMethods](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://WindowsBase:3.0.0.0:31bf3856ad364e35/MS.Win32.UnsafeNativeMethods).[RegisterWindowMessage](http://www.aisto.com/roeder/dotnet/Default.aspx?Target=code://WindowsBase:3.0.0.0:31bf3856ad364e35/MS.Win32.UnsafeNativeMethods/RegisterWindowMessage(String):Int32)("DispatcherProcessQueue"); 这个消息被Post到消息队列之前，还要设置MSG.Handle，这个Handle就是Window 1#的Handle。指定Handle是为了在消息循环Dispatch消息的时候，指定哪个窗口的WndProc（窗口过程）处理这个消息。在这里所有BeginInvoke引起的消息都是Window1#的窗口过程来处理的。

**第三步**，消息循环读取消息。

**第四步**，系统根据获取消息的Handle，发现跟Window1#的Handle相同，那么这个消息派发到Window1#的窗口过程，让其处理。

**第五步**，在窗口过程中，优先级队列当中取一个DispatcherOperation。

**第六步**，执行DispatcherOperation.Invoke方法，Invoke方法的核心就是调用DispatcherOperation构造时传入的Delegate，也就是Dispatcher.BeginInvoke传入的Delegate。最终这个Foo()方法就被执行了

通过上面的六步过程，一次Dispatcher.BeginInvoke就被处理完成。

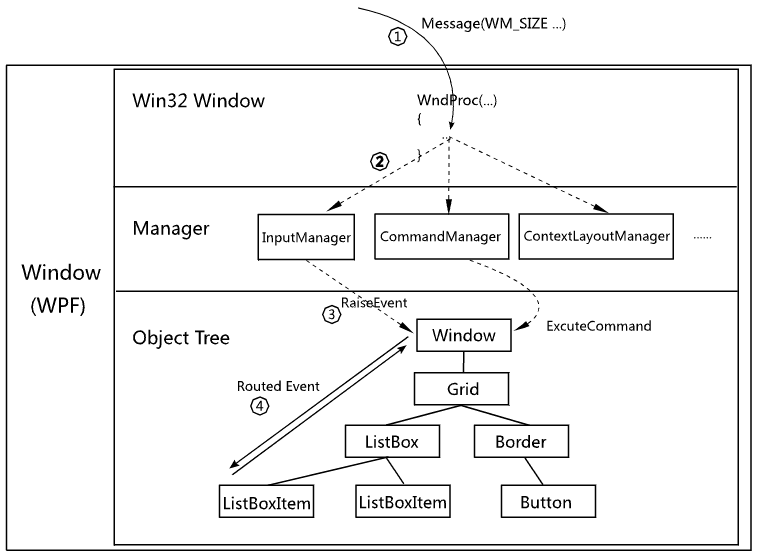
1. **深入WPF的Window**

WPF中的Window继承于ContentControl，内部可以承载一个Content，当然，借助于ItemsControl或Panel，Content也可以向下添加多个对象。这些对象都是WPF中的对象，也就是要承载的可视化数据。无论使用GDI绘制，或者使用DirectX绘制，在操作系统来看，Window都是一块持有句柄的有效区域。所有对该区域的操作，都会通过句柄来发送到Window对应的消息处理函数。也就是说，对外来看，WPF的Window依然是传统Win32的Window，对内它又把消息转化为Routed Event或者Command等来处理。关于这层处理和消息转化，要深入WPF的Window来谈起。

作为外界和可视化数据之间的桥梁，Window具有对内和对外两层作用。先说对内，Window内部可能会存在Button，ListBox等等控件，这些控件组成了一个对象树。树的子节点可能很多，但顶点只有一个，这个对象树是WPF的核心，Routed Event和Routed Command等都是依附于它的。抛开具体的对象树不说，我们要关注的是它的这种“众”字型的结构。如果你把这颗可视化数据组成的对象树想象成一个人的话，那么它的顶点就是它的头，我们对手臂和腿的操作只要对头喊话就可以了。换言之，对于WPF的Window，它对内最关心的就是找到对象树的头（RootVisual），然后通过头把操作传递下去。

从对外来看，操作系统关注的是注册Window的风格以及Rect。比如鼠标按键被按下时，按键消息被发送到系统的消息队列中，系统通过扫描所有注册窗口的Rect判断按键发生在哪个窗口中，再在适当的时机把按键消息从系统消息队列转移到创建窗口线程的消息队列中等待窗口处理。对于WPF的Window来说，同步这个Rect很重要，Window的UI是WPF的，但内部有个隐藏的使用CreateWindow创建的Win32-Window，当用户设置win.Width=60方法时要同步内部Window的Rect，反过来接收到WM\_SIZE时也需要调用RootVisual去执行WPF的Measure、Arrange流程。

用一个草图来表示Window的消息处理过程：



1. 系统将消息发给隐藏的Win32-Window，在Dispatcher中GetMessage并分发到对应的窗口过程处理函数WndProc。
2. WndProc里应是一个大的Switch-Case，用以处理不同的Window消息。按照消息的类别，WPF提供了不同的Manager来管理，这里的Manger并不是直接处理Window消息，并且并不是所有消息都经过WndProc再转到Manager的。   
   比如说WM\_KEYDOWN，Dispatcher调用GetMessage获得消息后， 调用了ComponentDispatcher的RaiseThreadMessage方法（关于ComponentDispatcher，可以参阅Nick的[文章](http://blogs.msdn.com/nickkramer/archive/2006/06/09/624468.aspx)），最终由KeyboardDevice产生Keyboard.PreviewKeyDownEvent这个路由事件（Routed Event）。
3. 仍然以WM\_KEYDOWN来说， InputManager找到这个Input发生的区域--Window，调用Window的RaiseEvent方法唤起Keyboard.PreviewKeyDownEvent这个路由事件。
4. 路由事件沿着对象树开始向下传递，方向是一去一回，由PreviewKeyDown到KeyDown。在这个传递过程中，相应的路由事件也被唤起，比如说如果此时焦点在Button上，当传递到Button时还会唤起Button的ClickEvent事件等。

这些Manager，其中像ContentLayoutManager，本身是Internal的，仅仅是在Measure和Arrange的内部使用，这里只是表示消息经由分类后最终由这些Manager来管理。这个过程比较有意思的是Input，简单的来谈一谈它。

# Input

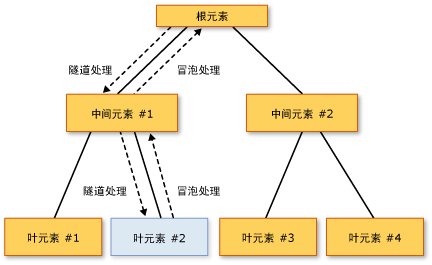
路由事件是WPF处理Input的核心，简略的说就是有一去一回从PreivewKeyDown到KeyDown这个过程，PreviewKeyDown的方向是从父到子，KeyDown的方向是从子到父。路由事件与一般事件的区别在于：路由事件是一种用于元素树的事件，当路由事件触发后，它可以向上或向下遍历可视树和逻辑树，他用一种简单而持久的方式在每个元素上触发，而不需要任何定制的代码（如果用传统的方式实现一个操作，执行过个事件的调用则需要执行代码将事件串连起来方可）。所谓的路由策略就是指：路由事件实现遍历元素的方式，总共有三种：Bubbling（冒泡：从子元素到根元素），Tunneling（隧道传递：从根元素到子元素），Direct（直接）

从图中可以看出，InputManager负责处理Input，一个Input，可能来自不同的设备--Mouse，Keyboard等等。InputManager要关注的地方有二：一，这个Input会转化成什么路由事件。二，这个Input作用在哪个UIElement上。第一个转化是由InputDevice来做的，这个InputDevice，具体有MouseDevice、KeyboardDevice等等。它会根据Window消息来生成对应的路由事件，然后把这些信息报告给InputManager。InputManager再根据这些信息找到作用的UIElement，然后唤起路由事件。

在 WPF 中，按照约定，隧道路由事件的名称以单词“Preview”开头。 输入事件通常成对出现，一个是冒泡事件，另一个是隧道事件。例如，[KeyDown](http://msdn.microsoft.com/zh-cn/library/system.windows.contentelement.keydown(v=vs.110).aspx) 事件和 [PreviewKeyDown](http://msdn.microsoft.com/zh-cn/library/system.windows.contentelement.previewkeydown(v=vs.110).aspx) 事件具有相同的签名，前者是冒泡输入事件，后者是隧道输入事件。 偶尔，输入事件只有冒泡版本，或者有可能只有直接路由版本。

实现成对出现的 WPF 输入事件的目的在于，使来自输入的单个用户操作（如按鼠标按钮）按顺序引发该对中的两个路由事件。 首先引发隧道事件并沿路由传播， 然后引发冒泡事件并沿其路由传播。 顾名思义，这两个事件会共享同一个事件数据实例，因为用来引发冒泡事件的实现类中的 [RaiseEvent](http://msdn.microsoft.com/zh-cn/library/system.windows.uielement.raiseevent(v=vs.110).aspx) 方法调用会侦听隧道事件中的事件数据，并在新引发的事件中重用它。 具有隧道事件处理程序的侦听器首先获得将路由事件标记为“已处理”的机会（首先是类处理程序，然后是实例处理程序）。 如果隧道路由中的某个元素将路由事件标记为“已处理”，则会针对冒泡事件发送已经处理的事件数据，而且将不调用为等效的冒泡输入事件附加的典型处理程序。 已处理的冒泡事件看起来好像尚未引发过。

为了说明输入事件处理的工作方式，请考虑下面的输入事件示例。 在下面的树插图中，leaf element #2 是先后发生的 PreviewMouseDown 事件和 MouseDown 事件的源。



事件的处理顺序如下所示：

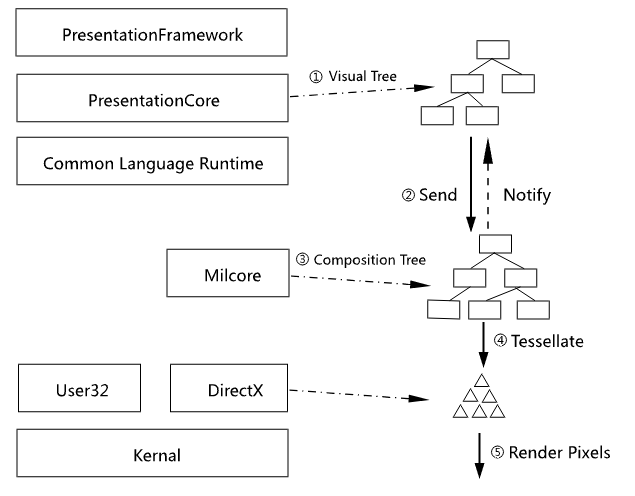
1. 针对根元素处理 PreviewMouseDown（隧道）。
2. 针对中间元素 1 处理 PreviewMouseDown（隧道）。
3. 针对源元素 2 处理 PreviewMouseDown（隧道）。
4. 针对源元素 2 处理 MouseDown（冒泡）。
5. 针对中间元素 1 处理 MouseDown（冒泡）。
6. 针对根元素处理 MouseDown（冒泡）。

所有的路由事件都共享一个公共的事件数据基类 [RoutedEventArgs](http://msdn.microsoft.com/zh-cn/library/system.windows.routedeventargs(v=vs.110).aspx)。 [RoutedEventArgs](http://msdn.microsoft.com/zh-cn/library/system.windows.routedeventargs(v=vs.110).aspx) 定义了一个采用布尔值的 [Handled](http://msdn.microsoft.com/zh-cn/library/system.windows.routedeventargs.handled(v=vs.110).aspx) 属性。 [Handled](http://msdn.microsoft.com/zh-cn/library/system.windows.routedeventargs.handled(v=vs.110).aspx) 属性的目的在于，允许路由中的任何事件处理程序通过将 [Handled](http://msdn.microsoft.com/zh-cn/library/system.windows.routedeventargs.handled(v=vs.110).aspx) 的值设置为 true 来将路由事件标记为“已处理”。[Handled](http://msdn.microsoft.com/zh-cn/library/system.windows.routedeventargs.handled(v=vs.110).aspx) 的值影响路由事件在沿路由线路向远处传播时的报告或处理方式。如果将 [Handled](http://msdn.microsoft.com/zh-cn/library/system.windows.routedeventargs.handled(v=vs.110).aspx) 设置为 true，以此将事件标记为“已处理”，则将“停止”隧道路由或冒泡路由，同时，类处理程序在某个路由点处理的所有事件的路由也将“停止”。

说过了Input，重点来看Presentation，所谓Windows Presentation Foundation，显示一定是它的重点。

# Presentation

在前面中，介绍到了需要被显示的可视化数据，在WPF中是以对象树（确切说是Visual Tree）来组织的。那么它又是如何被画出来的呢？从对象树到真正Render之间又发生了什么呢？



图例是WPF的架构图，其中重要的两个是PresentationCore和MilCore。在PresentationCore中，定义了Visual类，这个是WPF显示的核心，所有可以被显示的对象都直接或间接继承自Visual。当然，这里的Visual Tree就指Visual组成的树。Milcore（MIL -- Media Integration Layer），非托管代码，负责WPF和DirectX之间的通信，它主要由两部分组成：一，Composition Engine。二，Render Engine。前者负责创建Composition Tree，后者负责把Composition Tree转换成DirectX可以识别的Triangle并通知DirectX进行Render。

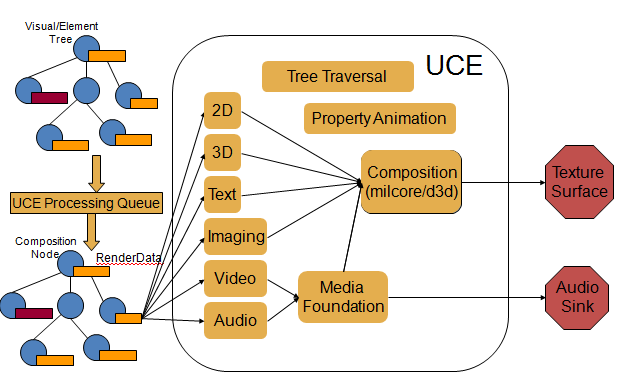
　　简单说一下Render的流程：

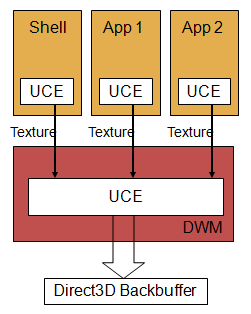
1. Visual被添加到Visual Tree上。
2. Visual Tree和Composite Engine通过Message Transport来进行通信，Message Transport包括Transport和Channel两部分。Transport定义了传输的细节，Channel作用在Transport上，用来建立一个双向的通信管道。这里，当Visual Tree被修改后，把被修改的Viusal数据通过Channel发送给Composition Engine。
3. Composition Engine接收到Visual数据后，创建对应的Composition Node，并加入到Composition Tree中去。
4. Composition Engine通知Render Engine开始绘制，Composition Tree中的节点是Rectangle，Ellipse等，DirectX不能识别这些数据，Render Engine要把这些数据转化为DirectX可以识别的三角形，这个过程叫做Tessellate。
5. Render Engine通知DirectX开始绘制（Render），DirectX在经过驱动（WDDM或者XPDM）通知显卡开始绘制像素到屏幕。

WPF中线程一分为二，有UI线程和Render线程。UI线程是托管代码，管理Visual  Tree，用于处理输入，事件等。Render线程是非托管代码，在MIL中，仅用于绘制，把从UI线程传入的Visual数据转化并添加到Composition Tree进行绘制。在这个过程中，Render线程是被动的，它等待着UI线程向它传输数据并下达命令，也会把操作的结果（绘制完成，错误）等通过Channel报告给UI线程。

这里要说说Viusal数据，也就是如何把Visual转化为Composition Node，在Avalon世界中，UCE（Unified Composition Engine）负责处理这层转化。当然，对UCE来说，它是不能识别WPF对象的，这种不能识别，就是说直接拿一个WPF的Line，它是不知道如何转化为相应Composition Node的，必须要WPF对象进行自描述，告诉UCE它对应什么Composition类型。UCE提供了IResource接口，这个接口定义了可以通过Channel传递到UCE的一系列方法。WPF的Visual实现了这一接口，Visual子类重写了其中的AddRefOnChannel方法并注明了其对应的Composition类型，比如说LineGeometry设置了它的类型是DUCE.Resource.Type\_LINEGEOMETRY。UCE通过这些信息，就可以把传递过来的Visual数据转化为相应的Composition Node了。

这里说到了UCE，每个WPF进程都有自己的UCE，并且在Avalon（Window Vista/Window 7）中，负责绘制桌面的DWM（Desktop Window Manager）也有它的UCE（也叫DUCE）。为了提供透明效果，桌面上的显示需要进行混合，DWM也是使用Composition Tree来管理窗口的，用两幅图来描述一下UCE的处理过程：





最终，DWM经过混合后得到了桌面最后的透明效果。

当然，整个过程不必细究，在WPF编程中也很少需要从UCE这个角度来考虑问题，只是帮助朋友们捋清一下思路，更好的理解WPF。讲过了这些底层的处理，把思路回归到Window上来，来看看Window是如何对这些进行整合的。

1. **深入了解Dispatcer**

新建一个Window程序，代码如下：

|  |
| --- |
| int WINAPI \_tWinMain(HINSTANCE hInstance,　 HINSTANCE hPrevInstance, LPTSTR lpCmdLine,　 int nCmdShow)  {  RegisterWindowClass(hInstance);　　　　 //1  HWND hWnd = CreateWindow(szWindowClass, szTitle, WS\_OVERLAPPEDWINDOW, //2  CW\_USEDEFAULT, 0, CW\_USEDEFAULT,z 0, NULL, NULL, hInstance, NULL);  ShowWindow(hWnd, nCmdShow);  MSG msg; //3  while (GetMessage(&msg, NULL, 0, 0))  {  TranslateMessage(&msg);  DispatchMessage(&msg);  }  return (int) msg.wParam;  } |

其中的RegisterWindowClass

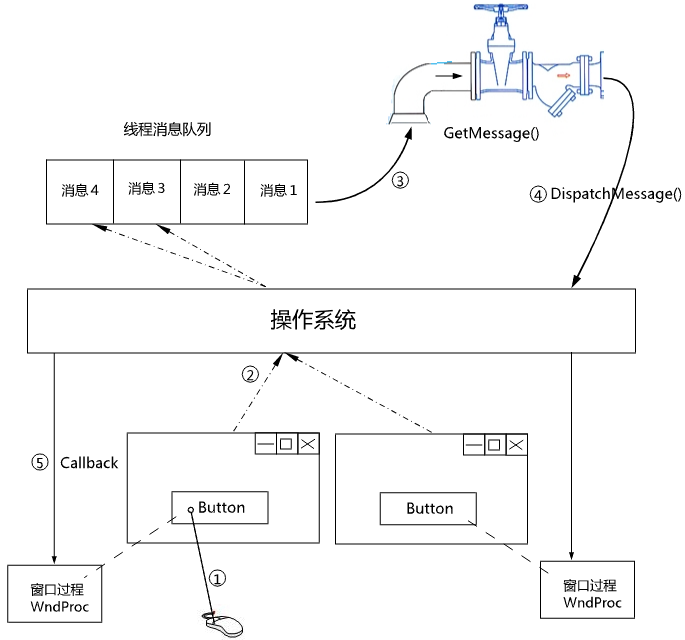
|  |
| --- |
| WORD RegisterWindowClass(HINSTANCE hInstance)  {  WNDCLASSEX wcex;  wcex.lpszClassName = szWindowClass;  wcex.lpfnWndProc = WndProc;  ...  }  LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)  {  switch (message)  {  case WM\_PAINT:  ...  } |

这个创建窗口并显示的过程如下：

1. 调用RegisterWindowClass注册窗口类，关联其中的窗口过程WndProc。
2. 调用CreateWindow创建窗口并显示。
3. （主线程）进入GetMessage循环，取得消息后调用DispatchMessage分发消息。

　　这里的GetMessage循环就是所谓的消息泵，它像水泵一样源源不断的从线程的消息队列中取得消息，然后调用DispatchMessage把消息分发到各个窗口，交给窗口的WndProc去处理。

　　用一副图来表示这个过程：



操作系统底层获知这次点击动作，根据点击位置遍历找到对应的Hwnd，构建一个Window消息MSG，把这个消息加入到创建该Hwnd线程的消息队列中去。

应用程序主线程处于GetMessage循环中，每次调用GetMessage获取一个消息，如果线程的消息队列为空，则线程会被挂起，直到线程消息队列存在消息线程会被重新激活。

调用DispatchMessage分发消息MSG，MSG持有一个Hwnd的字段，指明了消息应该发往的Hwnd，操作系统在第2步构建MSG时会设置这个值。

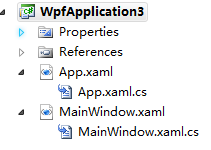
消息被发往Hwnd，操作系统回调该Hwnd对应的窗口过程WndProc，由WndProc来处理这个消息。

这是一个简略的Window消息处理流程，往具体说这个故事会很长，让我们把目光收回到WPF，看看WPF和即将介绍的Dispatcher在这个基础上都做了些什么，又有哪些出彩的地方。

# 从Main函数说起

作为应用程序的入口点，我们仍然从Main函数走进WPF。

新建一个WPF工程，如下：



默认的WPF工程中中是找不到传统的Program.cs文件的，它的App.xaml文件的编译动作为ApplicationDefinition，编译后，编译器会自动生成App.g.cs文件，包含了Main函数。如下：

|  |
| --- |
| [System.STAThreadAttribute()]  [System.Diagnostics.DebuggerNonUserCodeAttribute()]  public static void Main()  　　　　{  WpfApplication3.App app = new WpfApplication3.App();  app.InitializeComponent();  app.Run();  } |

这里出现了Application类，按MSDN上的解释，“Application 是一个类，其中封装了 WPF 应用程序特有的功能，包括：应用程序生存期；应用程序范围的窗口、属性和资源管理；命令行参数和退出代码处理；导航”等。

　　调用app.Run()之后，按照前面Win32的步骤，应用程序应进入到一个GetMessage的消息泵之中，那么对WPF程序来说，这个消息泵是什么样的呢？又和Dispatcher有什么关系呢？

Dispatcher的构造函数是私有的，调用Dispacher.CurrentDispatcher会获得当前线程的Dispatcher，Dispatcher内部持有一个静态的所有Dispatcher的List。因为构造函数私有，只能调用CurrentDispatcher来获得Dispatcher，可以保证对同一个线程，只能创建一个Dispatcher。

　　Dispatcher提供了一个Run函数，来启动消息泵，内部的核心代码是我们所熟悉的，如：

|  |
| --- |
| while (frame.Continue)  {  if (!GetMessage(ref msg, IntPtr.Zero, 0, 0))  break;  TranslateAndDispatchMessage(ref msg);  } |

　　这里出现了一个Frame的概念，暂且不谈，来看看Dispatcher相对于传统的消息循环，有哪些改进的地方。

在Winform的消息循环中，

1. 为了线程安全，调用Control的Invoke或者BeginInvoke方法可以在创建控件的线程上执行委托，方法的返回值分别为object和IAsyncResult。尽管可以使用IAsyncResult的IsCompleted和AsyncWaitHandle等方法来轮询或者等待委托的执行，但对于对任务的控制来讲，这个粒度是不够的，我们不能取消（Cancel）一个已经调用BeginInvoke的委托任务，也不能更换两个BeginInvoke的执行顺序。
2. 更为友好的接口支持，Windows编程中，在窗口消息循环中加入Hook是常见的需求，Dispatcher提供了DispatcherHooks类，以Event的形式对外提供了OperationAborted，OperationCompleted，OperationPosted等事件。

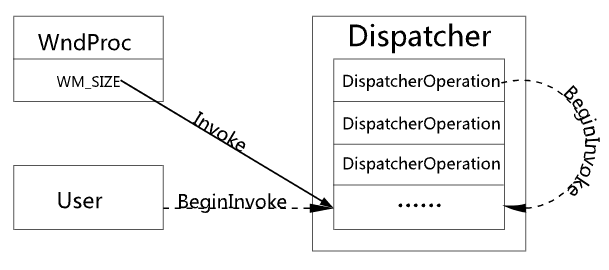
　　这里的Operation指的是DispatcherOperation，为了更好的控制消息循环，WPF引入了DispatcherOperation来封装Window消息，这个DispatcherOperation如下：

|  |
| --- |
| public sealed class DispatcherOperation  {  public Dispatcher Dispatcher { get; }  public DispatcherPriority Priority { get; set; }  public object Result { get; }  public DispatcherOperationStatus Status { get; }  public event EventHandler Aborted;  public event EventHandler Completed;  public bool Abort();  public DispatcherOperationStatus Wait();  public DispatcherOperationStatus Wait(TimeSpan timeout);  } |

　　 DispatcherOperation类看起来还是比较简单明了的，以属性的形式暴露了Result（结果），Status（状态），以及用事件来指出这个Operation何时结束或者取消。其中比较有意思的是Priority属性，从字面来看，它表示了DispatcherOperation的优先级，而且提供了get和set方法，也就是说，这个DispatcherOperation是可以在运行时更改优先级的。那么这个优先级是怎么回事，Dispatcher又是如何处理DispatcherOperation的呢，让我们深入DispatcherOperation，来看看它是如何被处理的。

所谓深入，也要有的放矢，从三个方面来谈一下DispatcherOperation：

1. DispatcherOperation是如何被创建的。
2. DispatcherOperation是何时被执行的。
3. DispatcherOperation是怎样被执行的。

　　 Dispatcher提供了BeginInvoke和Invoke两个方法，其中BeginInvoke的返回值是DispatcherOperation，Invoke函数的内部调用了BeginInvoke，也就是说，DispatcherOperation就是在这两个函数中被创建出来的。我们可以调用这两个函数创建新的DO，WPF内部也调用了这[](http://images.cnblogs.com/cnblogs_com/Zhouyongh/201101/201101120259509166.png)两个函数，把Window消息转化为DispatcherOperation，用一副图表示如下：

1. 窗口过程WndProc接收到Window消息，调用Dispatcher的Invoke方法，创建一个DispatcherOperation。Dispatcher内部持有一个DispatcherOperation的队列，用来存放所有创建出来的DispatcherOperation。默认一个DO（DispatcherOperation缩写）被创建出来后，会加入到这个队列中去。WndProc调用Invoke的时候比较特殊，他传递的优先级DispatcherPriority为Send，这是一个特殊的优先级，在Invoke时传递Send优先级WPF会直接执行这个DO，而不把它加入到队列中去。
2. 用户也可以随时调用Invoke或者BeginInvoke方法加入新的DO，在DispatcherOperation处理的时候也可能会调用BeginInvoke加入新的DO。

　　 DO被加入到Dispatcher的队列中去，那么这个队列又是何时被处理呢？Dispatcher在创建的时候，创建了一个隐藏的Window，在DO加入到队列后，Dispatcher会向自己的隐藏Window发送一个自定义的Window消息（DispatcherProcessQueue）。当收到这个消息后，会按照优先级和队列顺序取出第一个DO并执行：

1. 用户调用BeginInvoke。

如：public DispatcherOperation **BeginInvoke**(Delegate method, DispatcherPriority priority, params object[] args);

1. Dispatcher创建了一个DO，加入到DO队列中去，并向自己的隐藏窗口Post自定义消息（DispatcherProcessQueue）。
2. 创建隐藏窗口时会Hook它的消息，当收到的消息为DispatcherProcessQueue时，按照优先级取出队列中的一个DO，并执行。

　　每加入一个DO就会申请处理DO队列，在DO的优先级（DispatcherPriority）被改变的时候也会处理DO队列，DO在创建时声明了自己的优先级，这个优先级会影响到队列的处理顺序。

# DispatcherTimer

鉴于线程亲缘性，当需要创建Timer并访问UI对象时，多使用DispatcherTimer。DispatcherTimer的一个简单用法如下：

|  |
| --- |
| var dispatcherTimer = new System.Windows.Threading.DispatcherTimer();  dispatcherTimer.Tick += new EventHandler(dispatcherTimer\_Tick);  dispatcherTimer.Interval = new TimeSpan(0,0,1);  dispatcherTimer.Start(); |

　　在DispatcherTimer的内部，Timer的Tick事件处理也被包装成了DispatcherOperation，并调用BeginInvoke加入到Dispatcher中去。当这个DO被执行后，如果DispatcherTimer的状态仍然为Enable，DispatcherTimer会继续调用BeginInvoke加入新的DO。关于Timer的时间处理，Dispatcher会向自己的隐藏窗口调用SetTimer并计算时间间隔，当然，因为DispatcherOperation有优先级，不能保状正好在时间间隔时执行这个DO，这个执行的时间会比预计时间偏后而不会超前。

# UI线程和Dispatcher

　 通常，WPF启动时具有两个线程，一个处理呈现（Render），另一个用于管理UI。关于Render线程，请参见[前文](http://www.cnblogs.com/Zhouyongh/archive/2009/11/30/1613628.html)。这个管理UI的线程通常被称为UI线程。在WPF中，所有UI对象的基类为DispatcherObject，WPF在对所有DispatcherObject属性操作前进行了线程亲缘性校验，只有在创建UI对象的线程中才可以访问该UI对象。

　　前面提到，由于Dispatcher构造函数私有，一个线程最多只能有一个Dispatcher。对UI线程来说，Dispatcher的主要作用就是对任务项（DispatcherOperation）进行排队。对UI对象来说，DispatcherObject有一个Dispatcher属性，可以获得创建该UI对象线程的Dispatcher。这种设计通过Dispatcher统一了UI对象的操作，从使用上隔离了UI对象和线程间的关系。

多线程操作简单分为两种：多工作线程和多UI线程，当然，也可以有多工作多UI线程，思路是一样的，省去不谈。

　　程序启动时默认的主线程就是UI线程，它在调用Application.Run（也就是Dispatcher.Run）之后进入了一个GetMessage的循环中，对Window消息进行响应并构建执行一个个的DispatcherOperation。默认对UI对象的操作都是在这个主线程中，如果进行耗时很长的操作就会造成UI线程长时间不能继续响应Window消息，造成界面假死等一些的UI响应问题。对这种耗时较长的操作一般需要工作线程来帮忙，操作结束后再通过Dispatcher把结果Invoke到UI线程，如：

|  |
| --- |
| TextBlock textBlock = new TextBlock() { Text = "1" };  Thread thread = new Thread(new ThreadStart(() =>  {  //做一些耗时操作，这里用线程休眠10秒来模拟  Thread.Sleep(TimeSpan.FromSeconds(10));  textBlock.Dispatcher.Invoke(new Action(() =>  {  textBlock.Text = "2";  }));  }));  thread.Start(); |

　　当然，除了新建工作线程，也可以使用BackgroundWorker或者线程池中线程来进行耗时操作，操作结束后需要调用UI对象Dispatcher的Invoke或者BeginInvoke方法来操作UI，否则会抛出InvalidOperationException来提示不可跨线程访问UI对象。

　　这种多工作线程是很常见的，一般我们讨论的多线程大多指这种多工作线程单一UI线程，那么如何创建多UI线程的程序呢？

# 多UI线程

　　在谈多UI线程之前，先说说多UI线程使用的场景：

　　大多数情况下，我们是不需要多UI线程的，所谓多UI线程，就是指有两个或者两个以上的线程创建了UI对象。这种做法的好处是两个UI线程会分别进入各自的GetMessage循环，如果是需要多个监视实时数据的UI，或者说使用了DirectShow一些事件密集的程序，可以考虑新创建一个UI线程（GetMessage循环）来减轻单一消息泵的压力。当然，这样做的坏处也很多，不同UI线程中的UI对象互相访问是需要进行Invoke通信的，为了解决这个问题，WPF提供了VisualTarget来用于跨线程将一个对象树连接到另一个对象树，如：

|  |
| --- |
| public class VisualHost : FrameworkElement  {  public Visual Child  {  get { return \_child; }  set  {  if (\_child != null)  RemoveVisualChild(\_child);  \_child = value;  if (\_child != null)  AddVisualChild(\_child);  }  }  protected override Visual GetVisualChild(int index)  {  if (\_child != null && index == 0)  return \_child;  else  throw new ArgumentOutOfRangeException("index");  }  protected override int VisualChildrenCount  {  get { return \_child != null ? 1 : 0; }  }  private Visual \_child;  } |

　　在另一个UI线程下的VisualTarget：

|  |
| --- |
| Window win = new Window();  win.Loaded += (s, ex) =>  {  VisualHost vh = new VisualHost();  HostVisual hostVisual = new HostVisual();  vh.Child = hostVisual;  win.Content = vh;  Thread thread = new Thread(new ThreadStart(() =>  {  VisualTarget visualTarget = new VisualTarget(hostVisual);  DrawingVisual dv = new DrawingVisual();  using (var dc = dv.RenderOpen())  {  dc.DrawText(new FormattedText("UI from another UI thread",  System.Globalization.CultureInfo.GetCultureInfo("en-us"),  FlowDirection.LeftToRight,  new Typeface("Verdana"),  32,  Brushes.Black), new Point(10, 0));  }  visualTarget.RootVisual = dv;  Dispatcher.Run(); //启动Dispatcher    }));  thread.SetApartmentState(ApartmentState.STA);  thread.IsBackground = true;  thread.Start();  };  win.Show(); |

　　当然，这个多UI线程只是为了更好的捋清线程间的关系，实际使用中的例子并不是很常见。