Windows RT Introduction

Part 1-the architecture

1 概述

这篇的标题叫做 Windows RT Introduction 而非 Windows 8 Introduction 是想强调此次介绍是从开发人员的角度而不是普通用户的角度出发的。同时，我们关注的是 Metro Style 应用而不是传统的 Win32 应用程序的开发。

实际上，使用 C#或者 HTML + Javascript 书写一个 Hello world 应用的代码例子已经在网上泛滥了。但是仅有一个 Hello world 并不能够说你掌握了 Win RT 的开发。从 Pro 的角度来说我们应该弄清楚整件事情的细节。那么首先就应当是他的架构。这样写起程序来才能心定。

2 Windows 8 Metro 与 Desktop 模式

2.1 两种模式

Windows 8 的应用程序显示模式目前有两种，定义在 METRO_MONITOR_MODE 中：即传统的桌面模式（MMM_Desktop）以及 Metro 模式（MMM_Metro）。如果你是 Windows Phone 用户的话可能就会对 Metro 比较熟悉。事实上，微软在 2009 年启动 Windows 8 的研发工作时目标是创造一个完全不同以往的操作系统，完全不以之前的操作系统为蓝本。而后发现 Desktop 应用是不可或缺的部分而将两个部分进行合并。一开始用可能会有些别扭，但是我估计开发人员半天之内就能够熟练使用这个系统了。

2.2 Metro 和 Desktop 的一些不同

既然有两种模式那么我们自然就会关注他们的不同点。这个问题应该从架构图上做一下说明但是我们可以先有一些直观的认识。

2.2.1 Message Loop

消息处理的编程是传统 Desktop 应用程序的重要部分。你需要书写维护 Message Loop 的代码。例如：在 WinMain 调用（或者其子例程中）你需要书写类似

```
while (::GetMessage(&message, NULL, 0, 0)) {
    ::TranslateMessage(&message);
    ::DispatchMessage(&message);
}
```

而在 Window 创建之前候你一定指定了

```
WNDCLASS wndClass;
// ...
wndClass.lpfnWndProc = WndProc;
```

这样你就可以在 WndProc 函数中决定特定 message 的流向了。对于绘图来说，你一定是接受了 WM_PAINT 消息，然后执行了区域重绘。

但在 Metro App 中这些都已经隐藏了，而且消息的细节也可能发生了变化。Metro App 中你看不到消息循环。一切关于界面消息的分发都隐藏在了 CoreDispatcher 中。因此如果你用 Spy++去试探 Metro App 的消息循环那么你什么都抓不到。

2.2.2 Display

在传统的 Desktop 应用程序中，绘图可能通过 GDI，GDI+，DirectDraw，DirectX 进行。同样通过捕获 WM_PAINT 消息或者当系统处于 IDLE 的时候进行绘图（对于游戏编程来说）。
而 Metro App 不会再支持 GDI 和 GDI+，在 Metro App 中绘图只能通过 DirectX 来进行。确切的说是 Direct3D 和新公布的 Direct2D、Direct Write API。因此 Metro 应用的所有绘图都是希望是硬件加速的。这种绘图更高效，解放 CPU，而且一般不需要处理复杂的 Dirty Region Repaint。

2.2.3 Life Cycle

Metro App 并没有关闭窗口这种按钮。其生命周期是由系统托管的。系统会决定仅仅是挂起应用执行还是需要完全销毁应用进程。这和一般意义上的 Desktop 应用程序不一样。（当然，你也可以使用 Alt+F4 显示的结束 Metro App 的执行）。

2.2.4 Share & Communication

传统的桌面应用程序有多种手段进行公共组建的公用或 IPC。但是在 Metro App 中，隔离是一个很重要的概念，应用的可执行部分，运行库，Isolated Storage 都是独立的，不能够共用。同样，不能够使用传统的 IPC 机制。应用程序的互动仅仅可以通过内置的 Contracts 进行，关于这一部分内容可以查看 MSDN：
http://msdn.microsoft.com/en-us/library/windows/apps/hh464906.aspx

2.2.5 Portability

传统的 Desktop 应用程序的支持大多为 x86/64 架构的处理器。由于 Metro 环境可以完整运行在 ARM 处理器上是一个重要的特性，因此 Metro App 可以运行在 ARM 处理器上，即同时部署在 PC 和移动设备上。

2.2.6 OS Access

当然为了 Portability 的要求，必然要求应用不能够越过 Win RT 的抽象，因此 Metro 是不能像 Desktop App 那样访问所有的 Windows API 的。

3 从 Windows 8 API 的架构图看 Windows RT

我们对 Windows RT 的介绍都将围绕着这个图展开。
在这个图中，最底层的是 NT 的内核；在其上是 Windows 子系统。实际上 NT 至少有三个子系统，Windows 子系统，POSIX 子系统（Unix）和 OS/2 子系统。POSIX 子系统和 OS/2 子系统实际还是在使用 Windows 子系统。 在 Windows 子系统上划分了不同的运行时（橙色）和程序库（浅蓝色），最上面的绿色是我们使用的各种开发语言。
这个架构图实际上说明了一切。并且消除了很多误解：

（1） 第一个误解是 INFOQ 指出的 Windows RT 和 Win32 是完全分开的。这源于微软发布的一幅饱受批评的架构图，在那张架构图中，Windows RT 和 Windows 子系统竟然是并排排列的。这是很荒谬的，Windows RT 实际上基于 Windows 子系统。首先 Windows RT 完全基于 COM；其次 Windows RT 利用了一部分现有的 Win32 API；其余的部分 Windows RT 则直接访问 NT 内核。

（2） 第二个误解是 C++/CX。C++/CX 是微软推荐的开发 Windows RT 的方式。他主要隐藏了 COM 的复杂性。关于这个问题我们后续会有说明。这个误解是 C++/CX 实际就是 C++ CLI。实际上这是两个完全不同的东西，C++ CLI 是运行在托管环境下的，而 C++/CX 完全是 Native 的。

3.1 Windows RT 仅用于 Metro 应用

从架构图中可以看出，Win RT 仅仅用于 Metro 应用。并秉承了我们刚才介绍的，简单部署，没有共享的组件，没有 IPC，等等。

3.2 Windows RT 构建与 COM 之上

这也是为什么说 Windows RT 是构建与 Win32 之上，因为 COM 是 Win32 重要的组成部分。这意味着：

（1） 你可以用之前所有的消费 COM 的方式来使用 Windows RT，你可以用 C，你可以用 ATL 或者新的 WRL；
（2） WRL 完全符合传统的 C++语法，这意味着你可以使用不同的编译器（例如 Intel C++编译器）来构建 Metro 应用。但是微软显然希望大家都来使用 C++/CX，WRL 的文档跟没有差不多，现在也看不到一个完整的例子出现。

3.3 Windows RT 限制了系统 API 的调用

Win RT 是基于 COM 的，但是 COM 仅仅是一个二进制协议而已。在 COM Interface 实现中从技术上讲还是在调用 Win32 API。但由于前面介绍的 Win RT 的设计要求，系统 API 的调用需要受到严格的限制。仅仅支持有限的 API 调用，因此在你希望使用一个 Win32 API 时，一定要查询 MSDN 上的 Applied To 一节，看看是否是 Metro Style App | desktop App。

同样的道理，.NET 的某些方法也在进行着系统调用，因此在使用.NET 开发 Metro Style 应用程序的时候也并不是所有的程序集都能够支持。当然，如果使用 P-Invoke 的方式调用 Win32 API 那么危险性就会更大。

总之，在 Metro 应用中调用不支持的 Win32 API 会有如下的后果：

（1） 发生一个 Runtime Exception；
（2） 应用程序失去响应，尤其是在使用和消息循环相关的代码时。例如对 Metro App 进程使用 WaitForSingleObject(hProcess)。
（3） 调用成功，但是你的 Metro App 应用会被 Windows Store 驳回。

按照上述分析，那么即使你存在相当可观的 COM 代码库，也需要巨大的努力才能够保证他们在 Metro App 上正确运行（消除非法的系统调用）。对于新的应用来说，为了避免书写大量的 COM 开发代码，最好使用 C++/CX 进行开发了。

3.4 C++/CX

为什么会有 C++/CX 呢？这可以联想 n 年前我们为了避免 C++开发 COM 的冗长的代码，转而使用 C 开发关键程序，而使用 Visual Basic 创建 COM 组件。现在时间到了 2012 年，VB6 已经不在考虑范围之内了，于是 C++/CX 取代了他的位置。

C++/CX 是 Native 的，但是它的语法为什么能够和 C++ CLI 保持近乎一致呢？这是因为 Win RT 本身虽然是 Native 的，但它以.NET 兼容的方式暴露了元数据。但是我们在编程中要时刻想到，我们在操作实打实的 Native 对象。根本没有什么垃圾收集器在帮助我们。

那么为什么不单纯使用.NET 开发 Metro App 呢？这是因为对于移动设备来说，CPU 的速度和电池是两大局限，因此在近一年，Go Native 的大潮终于袭来。目前：

（1） iOS 使用 Objective-C 进行程序开发，而且在移动设备上也是没有垃圾收集器的，需要手动释放使用的内存；
（2） Android 一开始使用 Java 进行开发，但是在糟糕的性能和社区的强大压力下，终于开放了 C/C++开发接口；
（3） WP7/8 也出现了类似 Android 的情况。

目前客户端应用向更薄（核心应用向服务器移动），更快（运行速度快，耗电小），交互更丰富（没有动画你都对不起观众）的方向发展。因此开放 Native 接口是大势所趋，C/C++顺理成章的在 Windows 8 强势回归了。

但是，用.NET 开发 Metro 应用也是一个不错的选择，尤其你的应用没有密集的运算（游戏）的情况下。你可以参考幻灯片中的 Cheat Sheet。