

第一回. 真的了解.NET Compact Framework 吗?

作为系列文章的开篇, 有必要先详细了解一下基于 CE.NET 的.NET Compact Framework(以后简称.NET CF), 本文叙述了.NET CF 的设计目标, 架构特征和执行环境。

.NET CF 的目标在哪里?

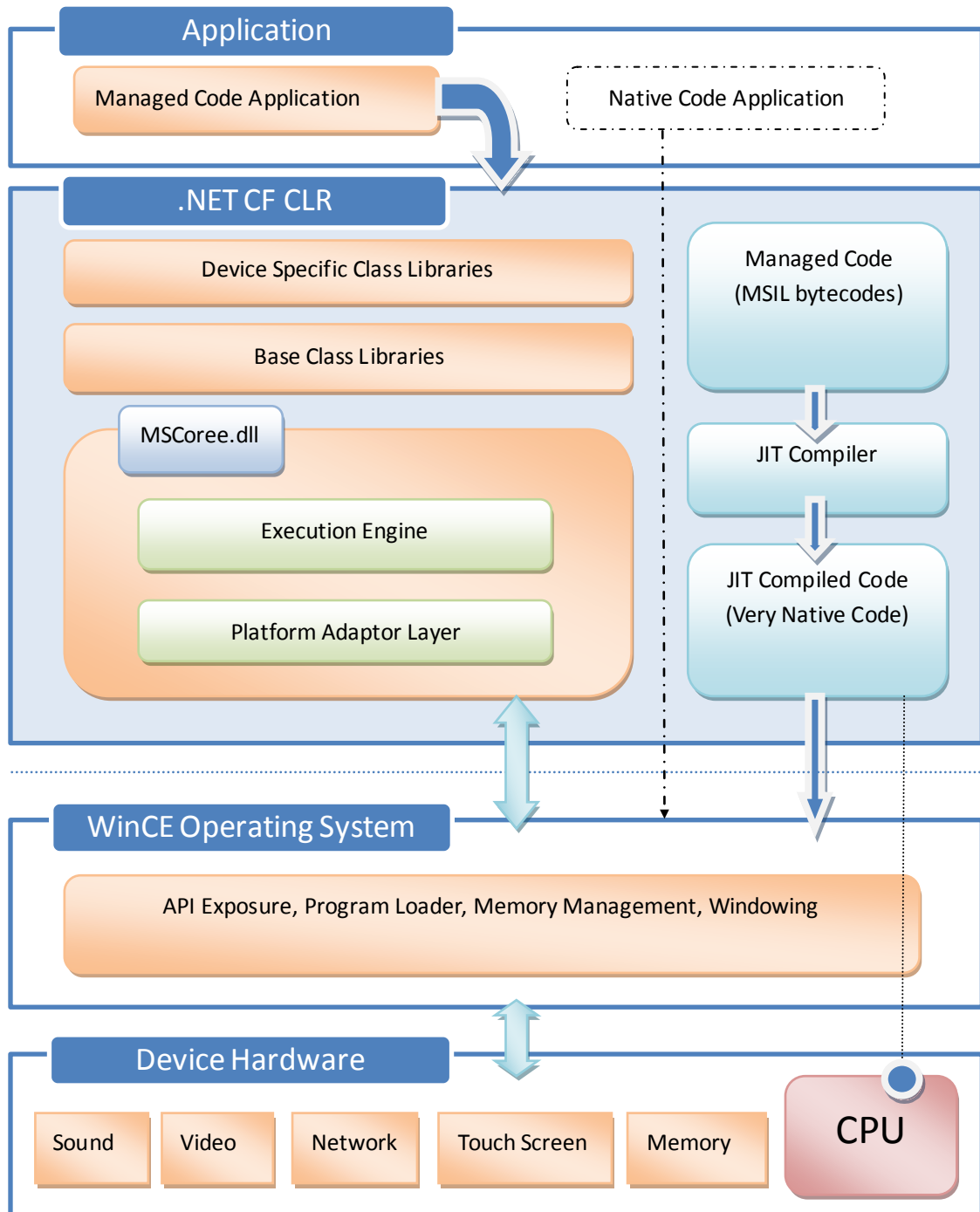
1. 专为设备设计的便携式小型.NET CLR
具有.NET Framework 的子集属性, 支持多种语言开发。我们知道, 在英文里面“便携式”对应的单词是 **Portable**, 这个 **Portable** 我们可以从两方面理解: 一方面, .NET CF 工作在一个灵活的, 移动的, 资源有限的环境下; 另一层意思则体现在.NET CF 本身的特性上, 比如说它与 OS 的宽松耦合, OS 管与 CLR 托管的不同就在这里。从编程的角度 **Portable** 体现在很多“和谐”的方面, 比如 I/O 内存映射, 比如仅支持 **Unicode** 编码等等。
2. 与 Visual Studio 系列 IDE 高度兼容
不仅仅是编译, 调试托管和非托管的代码, 在 **Visual Studio 2008** 中你还可以通过 **Device Security Manager** 来为已连接的设备管理证书和设置安全级别。甚至可以编程访问模拟器资源。
3. 与主机的操作系统有良好的共存性
这个共存性是多方面的, 包括应用程序的执行模型, 内存管理, 用户输入和 **UI** 接口。这些在后面的文章中您都会接触到。

当然还有一些要求是.NET CF 做不到的, 暂时也不是它的目标, 为了不使大家对.NET CF 的要求太“苛刻”, 我觉得必须把这些“非目标”也列举出来:

1. **Compact VS. Full**
.NET CF 不是对桌面版本.NET Framework 的部分简单平移, 把.NET Framework 完整移植到移动设备上并不是.NET CF 的目标, 尽管表面上看起来有些内容和完整版的.NET Framework 是一致的, 但是其实现方式可能很不一样。
2. **实时性**
Windows Mobile 是一个 32 位的民用操作系统, 你不能要求它和 **VxWorks** 一样工作! .NET CF 也并没有提供对强实时性的支持(问题是您真的需要那么高的实时性吗?)。
3. **语言支持**
.NET CF 目前支持的开发语言并不像完整版本的那么丰富, 目前比较流行的是 **C/C++**, **C#** 和 **VB**。但是.NET CF 完全支持精简版本的 **ECMA CLI Profile**, 这意味着你也可以为更多的语言编写针对.NET CF 的编译器。

.NET CF 的结构模型

.NET CF 的架构跟完整版的.NET Fx 有相似之处，同时又具有自己特色，如图表 1 所示。



图表 2 .NET CF Architecture

最下面的是硬件层，由于图幅有限，我仅列出了主要的一些硬件，而这所有的硬件都是由 Windows CE 操作系统所控制的，WinCE 提供了内存管理规范和用于加载可执行文件的 Program Loader，Program Loader 负责将可执行文件 Push 到内存中并启动它。当然，WinCE 作为操作系统还有很多其他职能，比如线程管理，绘制窗体，相应来自 GUI 的事件，管理网络连接等等。

作为使用 .NET CF 的程序员，我们主要关注的是图中虚线以上的部分，现在来看看 .NET CF 的 Common Language Runtime (CLR)，图中灰色背景的方框表示 .NET CF CLR。

用 Native Code (如 C/C++) 编写的基于 WinCE 的程序，直接被编译成 CPU 可以识别的指令，但是依赖 WinCE 去加载它们并提供所需的服务。而 CLR 是一个用来托管应用程序的平台，托管的应用程序被编译成 Microsoft Intermediate Language (MSIL)，IL 在这里提供了一个与 CPU 松耦合的机会，CLR 根据不同的 CPU 体系结构将 IL 编译成不同的 CPU 指令，这一点在行为上与桌面版的 CLR 是一致的。

有一点要弄清的是托管的 EXEs 或者 DLLs 是由 IL 构成的，并不能被 CPU 直接执行，而是需要被 CLR 编译成适用于本地 CPU 的指令或者是本地代码。可见，CLR 的工作是执行托管代码，这个过程就是托管代码本地化并被执行的过程，简单来讲，它包括以下步骤：

1. 将 IL 从文件系统加载到内存中。
2. 这些 IL 中的部分或全部将被转化成本地代码供 CPU 执行
3. 如果这些 IL 引用了某些 DLL 的内容，则当 DLL 被找到并正确加载后，这些被用到的部分也会被转化成本地代码并被执行。

可见这个 Just In Time Compiler 在 CLR 的运行中扮演着十分重要的角色。在 .NET Compact Framework 中有两种形式的 JIT 编译器，iJIT 和 sjIT：

iJIT 适用于所有的 CPU，如 ARM，MIPS，SHx 和 X86 等。iJIT 较简单，编译的速度也比较快，但是它编译出来的本地代码并不像 sjIT 那样经过优化的。

sjIT 编译器是 ARM 指令体系下特有的，它充分利用了 ARM 处理器的优势。虽然编译速度不及 iJIT，但是编译后的代码在第二次运行的时候会迅速得多。

所以在编写应用程序的时候你应当考虑到你的程序是否是专为基于 ARM 的设备而设计，或是有考虑到用户的机器可能是一台性能一般的 MIPS。

默认状况下，仅当无法使用 sjIT Compiler 的时候，CLR 才会选择使用 iJIT 的方式。这样做的的原因是，通常在程序执行过程中，花在 JIT 编译上的时间和执行程序的时间相比是微不足道的。

需要注意的是在我们的程序当中应当避免额外的 JIT 行为，因为有些情况下，这会明显影响到应用程序的运行速度，当然，要做到这一点需要您对 CLR 有一定的认识。JIT 按需编译，并尝试对编译后的代码在程序生命周期内进行保留，这样下一次调用的时候就不必再执行 JIT 了。说是尝试是不考虑内存的缘故，这样的缓存不会无限制的进行下去，当内存不够用时，CLR 会逐方法的将 JIT 过的代码清除掉，这就是所谓的 Code Pitching。清除掉之后再次调用该方法 CLR 就会重新进行 JIT 编译。有趣的是这个 Pitch 的过程也是智能化的，哪些最不常用的方法的 JITed Code 会被最先清除。

如果我们的程序编写不当，在某些极端情况下，重复的 JIT 可能会出现在一个循环中，每一次循环都将重新 JIT 一次，这显然会使性能大打折扣，而且很可能使你的程序因此 down 掉。

性能问题在今后的文章中还会专门介绍。

用一句话概括图表 1，可以说“是.NET CF CLR 使得.NET 应用程序得以运行在各种不同的基于 Windows CE.NET 的移动设备上”。

Compact CLR VS. Full CLR

对于习惯了桌面应用的程序员，有必要了解一下精简版的.NET CF 与完整版本的.NET CF 有哪些不同。

首先从体系结构上，.NET CF CLR 与桌面版本的 CLR 不尽相同。从图表一我们看到，.NET CF CLR 构建在 Platform Abstraction Layer(PAL)之上，PAL 位于执行引擎与 OS 之间，将 CLR 从硬件的层面抽象出来，如果您需要将 CLR 移植到其他平台，只需要改变 PAL 层，并为该平台的 CPU 编写相应的 JIT Compiler。正是这种灵活性，使得.NET CF CLR 也随着日新月异的 Mobile 硬件设备不断前行。

在 JIT 编译之后，.Net CF CLR 对生成的本地代码的处理也与桌面版本不同。在桌面版本的 CLR 中，JIT 过后的代码有时候会在程序退出之后依然存在，这样在它下次加载的时候会快一些。但是.Net CF CLR 仅仅在程序运行期间保存 JIT 生成的代码。每一次程序启动，JIT 编译必然再次发生。

在对程序集的定义上面，.Net CF CLR 和桌面版本的 CLR 也有所不同，桌面版本的程序集支持多文件构成一个程序集。而.NET CF CLR 不支持这一性质，这在移植桌面应用程序到.NET CF 下的时候是需要注意的。

这些不同都是与移动设备本身的特性密不可分的，如果您想了解更多.NET CF 与.NET Framework 的不同，请参照[这里](#)

.Net CF 应用程序的执行环境

我们通常在调试程序的时候主要有两种执行环境---模拟器和真实设备。要注意，这里模拟器并不是说就是在 Windows(x86)下面工作的用来模仿基于 WinCE 操作系统行为的一个程序，而是一个真正的 CE.NET 或者 Windows Mobile 操作系统的镜像，只是它是由 x86 的操作系统所编译并运行。而在真实设备上运行的程序则是.NET CF CLR 所掌管的一个实例。

前面讲 JIT 的时候已经提到过应用程序的执行了。我们说“是.NET CF CLR 使得.NET 应用程序得以运行在各种不同的基于 Windows CE.NET 的移动设备上”。CLR 无疑是.NET 最重要的组成部分，它负责将已编译成 MSIL 的托管程序集装配到应用程序域中，以 JIT 的编译方式将他们编译成本地代码供宿主 CPU 执行，同时它还要在运行过程中管理内存分配，垃圾回收以及加载其他类库等。

从组成上可以把.NET CF CLR 分成两部分：执行引擎和基础类库。

执行引擎与底层操作系统提供的各种服务接口打交道(这离不开 PAL 的作用)，而基础类库则是构成.NET 应用程序的基本程序单元。

其中基础类库发展到现在已经十分丰富，想必大家也比较熟悉，在此无需也无法作多的介绍。

下面看看执行引擎(Execution Engine)。

执行引擎为托管代码的执行提供了众多基本服务，比如：

- 程序集的 Loader 和全局程序集缓存(GAC)
 - 元数据引擎/缓存
 - 对类层次模型的描述
 - “反射”技术
 - (关于程序集的加载，后续的文章中也会介绍)
- JIT 的编译和校对机制
- 安全的执行体系
 - 异常探知,
 - 本地代码互操作,
 - OS 安全性保障
- 垃圾回收器
- 对调试的支持
 - 为 Debug 版本的程序生成可方便调试(如断点)的代码
- 对某些托管的 API(Class Libs)采用本地化的实现

执行引擎(EE)是.NET 的核心组件，它掌管着.NET CF 的所有其他东西。其本身是一个本地可执行的文件，它通过平台抽象层(PAL)与底层操作系统进行交互，他们共同被安置在一个可执行文件中(Mscoree.dll)，你可以在 Windows/System32 目录下面找到它。

PAL 层(the Platform Adaptation Layer)就如同基于 WinNT 的操作系统(Windows NT 4.0, 2000, XP, 2003)的硬件适配层(HAL)，用于在常规代码和不同硬件水平的 CPU 之间做一个适配。正是 PAL 的存在，使得.NET CF 的程序集能运行在使用任何 CPU 的任何 WinCE 兼容的设备上，而以往用经典的 EVC 开发的应用程序可能还需要为不同的 OS 和 CPU 单独编译。

另外，.NET CF 同样支持 GAC，你可以以可复用的方式部署你的应用程序，它是通过预加载某些基础类库，你可以在你的应用中通过应用的方式来调用他们，这样减少了代码量，也提高了性能。

在异常处理方面，.NET CF 还有一个有趣的特点，我们知道，通常一个 error 发生的时候，一条错误消息会随之而来，开发者可以捕获到这个消息，并选择怎么处理。在.NET CF 中，基于对内存问题的考虑，微软摘录了所有这些错误信息，并为各种支持的语言把它们分别单独放在一个字符串文件中 (SYSTEM.SR.dll)，部署的时候，你可以选择是否将这些错误信息文件随你的应用程序一同部署到设备上。当然选择不部署这个文件在 load 的时候会节省一些性能，在[这篇文章](#)也有提到这个问题。

之前一再提到 Portable 这个词，设备是 Portable 的，.NET CF 也是 Portable 的，移植到新的平台只需重新编写 JIT Compiler 和 PAL 层，执行引擎是无需改变的，不同的 PAL 导致了 MSCoree.dll 不同的实现方式。执行引擎用标准 C 语言编写，这也是 Portable 的体现。

总结

.NET Compact Framework 使得熟悉 .NET Framework 的程序员得以用他们熟知的 C# 或者 VB 来开发移动设备上的程序。

.NET CF 和完整版的 .NET Framework 不单纯是子集关系。开发的时候除了要注意，用户可能使用不同处理器，还应考虑到 PPC 与 PC 的区别，Smartphone 与 PPC 的区别。

.NET CF CLR (the CLR Designed for .NET Compact Framework) 是专为设备设计的代码托管运行机构，在垃圾回收机制，异常处理和安全性等方面继承了 .NET 托管应用程序的优势。同时它又有自身的特性，比如 PAL。两种 JIT 的方式中，sJIT 是仅适用于 ARM 处理器的。

执行环境方面，MSCore.dll 是应用程序执行的基础，它由执行引擎 (EE) 和平台抽象层 (PAL) 构成。

对于 .NET CF 3.5 中的新特性，可以参考 [这里](#)

华中科技大学控制科学与工程系 黄季冬

hjd.click AT gmail.com

2008 年 2 月 25 日