2002上·增值版

programmer

程序员

增值合订本

适合开发者 项目经理 CTO&CIO 编程爱好者阅读收藏

专题应用 .NET/Java/Open Source/Borland开发技术

浓缩展现 2002年《程序员》杂志全年所有文章精粹

最新集锦 developerWorks中国网站权威文章/教程/工具

真挚奉供 全年杂志电子版/源代码/编程手册

赠 Delphi 7/DB2 8.1/WSAD 5.0

增值版 上册

合订版 下册

贺岁版 CD1

程序员版 CD2

電子工業出版社
Publishing House of Electronics Industry
http://www.phei.com.cn

www.csdn.net/magazine

程序员 增值合订本

2002 上 · 增值版

《程序员》总　　编：黄长著
《程序员》社　　长：张悦校
《程序员》副 社 长：蒋　涛
《程序员》执行副总编：熊池舟
《程序员》副 总 编：唐　琦
《程序员》技术总监：曾登高
《程序员》美术总监：关峻峰
《程序员》责任编辑：孟迎霞　汤　韬　熊　节　行　舟
张　里　闫　辉　孔祥利　吴志民
程　琰　俞　波　武　超

電子工業出版社
Publishing House of Electronics Industry
北京·BEIJING

京海工商广临字 2002053 号

Programmer 程序员 增值合订本（2002上·增值版）

图书在版编目（CIP）数据

程序员增值合订本（2002上·增值版）/《程序员》杂志社编. —北京：电子工业出版社，2003.1

ISBN 7-5053-8300-0

Ⅰ.程… Ⅱ.程… Ⅲ.程序设计—文集 Ⅳ.TP311.1-53
中国版本图书馆 CIP 数据核字（2002）第 098484 号

责任编辑：郭立
印　　刷：北京印刷三厂
出版发行：电子工业出版社　http://www.phei.com.cn
　　北京市海淀区万寿路173信箱　邮编100036
经　　销：各地新华书店
开　　本：850×1168　大1/16　印张：26　字数：800 千字　附光盘2张
版　　次：2003年1月第1版　2003年1月第1次印刷
印　　数：80 000册　　定价：39.00元（上、下册+2CD）

读者订购或发现装订错误、缺损、破损，请与《程序员》读者服务部联系。
联系地址：北京市朝阳区北三环东路静安中心26层（100028）
电话：010-84480948/84540231-33转279
传真：010-84540263

程序员增值合订本

内容简介

《程序员增值合订本》全套共分增值版、合订版两本图书，配专版、再片切版两张光盘。

- 增 值 版 （上册）：专题应用 .NET、Java、Open Source、Borland 开发技术
- 合 订 版 （下册）：浓缩展现 2002 年《程序员》杂志全年所有文章精粹
- 贺 岁 版 （CD-1）：最新集锦 developerWorks 中国网站权威文章、IBM 最新开发工具
- 程序员版 （CD-2）：真挚提供 全年杂志电子版 源代码 编程手册
- 特 别 附 赠 ：Delphi 7 DB2 8.1 WSAD 5.0

上册为“增值版”图书。从当前软件开发人员的需要出发，针对2002年最热门和使用最广泛的技术，精心整理了这套增值版技术图书，大部分内容都是在国内首次发表。

本书内容主要分为如下四大版块：

.NET版块：2002年最令软件开发者重视的无疑是.NET技术。.NET书籍目前已不少，大部分.NET的使用者已经理解了.NET的基础知识，但真正应用的还不多。

.NET版块得到了微软中国公司技术专家的大力帮助，共分七大专题：十余篇文章，分别为程序编C#讲座、FAQ（常见问题解答）、Web Services（MSDN专栏“All Your Services”中的精华文章）、Visual Studio .NET（Visual Studio .NET的高级特性介绍）、其他.NET的全面指南、Security 和 Case Study（如何用.NET技术开发真正的软件系统真实案例）专题。

Borland版块：Borland在国内最受欢迎的开发平台无疑是Delphi，内容共分为四大专题，即Delphi 7、Delphi 老版本用、数据库开发以及JBuilder高级指南。全部文章都由资深专家撰写，具有很高的实用价值。

Java版块：Java现在已经真正成为主流的开发语言，该版块共分六个专题，即：深入浅出读Java的数学魔术书、Java手机开发的4篇专栏、跨平台应用的角范JAX专题、详细论述Java垃圾收集、对象检查等JVM虚拟机底层内容的JVM专题、Persat JDO专题、以及Threading专题。

Open Source版块：开发源代码运动的浪潮一波高过一波，它带给软件开发者的影响深远而又巨大的，这次特地推出的Open Source版块，内容主要分为“大专题”wxWindows非常出色的跨平台GUI开发框架）、MemoryDB（内存数据库系统）以及Apache、PHP。

下册为“合订版”

本书浓缩了2002年《程序员》杂志的精华，全书共收录12期杂志中数百篇精彩文章，内容覆盖人物志、极道、管理、技术及服务四大版块，共分十五个专栏，分别为名人堂、创业者、CTO论道、名家专栏、技术专题、Python专栏、名家访谈、MSDN专栏、技术讲座、开发心得、源码分析、开发工具、编程指南、书评和广书推荐。此外，书后还附有全年12期杂志的目录索引。

全年12期杂志中最受读者欢迎的技术文章、最具指导性的管理、技术工具应用分析、最值得肯定的编程类图书评论，您都可以在这里找到。

CD-1 为贺岁版

本光盘内容包括developerWorks中国网站2002文章集锦，IBM最新开发工具（强大的通用数据库DB2 8.1版，还有集成Java、Web和企业应用于一体的开发环境WebSphere Studio Application Developer 5.0版），遗海数工具标志着IBM在开发工具市场从大企业延伸到中小企业，并保持着大企业的高性能。

CD-2 为程序员版

本光盘内容涵盖2002年程序员杂志的全部内容，另附送Borland Delphi 7完整试用版，用友华表Cell 5.0组件及Excel读写控件特别版等开发工具。此外，重点推出的“程序员编程手册”，包含了Java、VB、VC、Delphi、C++、SQL等开发工具的API、常用数、编程规范等，特别适合于开发者们学习、查阅和收藏。

希望读者能从这些精品文章和工具中得到真正有用的知识，学以致用，新年更上新台阶！

《developerWorks 2003 贺岁版》目录

- developerWorks 中国网站 2002 文章集萃
- 教程
- 工具包

  WebSphere Studio Application Developer 5.0Beta 版
- IBM软件全球专业认证

2002 年《程序员增值合订本》配套光盘目录

- 2002 全年 12 期《程序员》杂志电子版
- 《编程手册》：包含 Java、VB、VC、Delphi、C++ 等开发工具的 API、库函数、编码规范等
- 2002 全年《程序员》杂志文章配套源码
- 2002《程序员增值合订本》文章配套源码
- IBM DB2 通用数据库 8.1Beta版
- 用友华表 Cell 5.0 组件及 Excel 读写控件特别版
- Delphi 7 完整试用版
- 瑞星杀毒软件全功能查毒版

增值专题 .NET 版块

主持人 / 透明

导 读

在整个2002年中，最令中国软件开发者欣喜和重视的技术无疑是.NET技术。经过去年的前期宣传、Beta测试和今年的普及，.NET的使用者大多已经理解了.NET的基础知识，迫切希望获得更多更深入的.NET知识和经验。

在《程序员》杂志特别推出了.NET技术版块，大部分内容为首次发表，重点是在实际中的应用，深度探索.NET技术的奥妙。

".NET技术版块"中有三十余篇文章，共分七大专题：

☞ C#——本专题由台湾知名技术作家蔡学镛在《程序员》杂志上发表的专栏文章整理而成，介绍了学习C#编程的一些基础知识（例如.NET PE格式、Metadata、Assembly 等）。

☞ FAQ——本专题为.NET初学者准备，解答初学者最常提出的问题。共有两篇FAQ：

- 《.NET 框架 FAQ》
- 《写给 C++ 程序员的 C# FAQ》

☞ Web Services——本专题收录了深受读者欢迎的MSDN 专栏 "At Your Services" 中的精华文章，介绍了 Web Services 开发过程中的一些深度技巧。共有六篇文章：

- 《让我们相识》
- 《共享数据类型》
- 《在网络服务中使用 ASP.net 会话状态》
- 《通过.NET框架实现基于HTTP的异步Web服务调用》
- 《服务器端的异步 Web 方法》
- 《版本选择》

☞ Visual Studio .NET——本专题介绍了.NET 的集成开发环境 Visual Studio .NET 的一些高级特性。共有五篇文章：

- 《Visual Studio .NET 简介》
- 《Visual Studio .NET 下的 VC .NET 集成开发环境介绍》
- 《Visual Studio .NET 调试环境》
- 《自动化和扩展性概述》
- 《在 Visual Studio.NET 中自动化构建过程以及配置》

☞ 转向.NET——本专题介绍了从传统开发环境转移到.NET上的一些问题以及解决方案。共有六篇文章：

- 《Visual Basic 6.0 控件与.NET 控件的区别》
- 《将 ASP 转换为 ASP.NET》
- 《从.NET客户端调用 COM 组件》
- 《从COM组件中调用.NET 组件》
- 《在.NET 中使用 COM+ 服务》
- 《如何在.NET 中访问MySQL 数据库》
- 《如何在.NET 中访问 Oracle 数据库》

☞ Security——"安全"是软件开发中永远的话题。本专题介绍了 NET 中的安全性技术。共有七篇文章：

- 《CLR 安全基础设施提供证据、策略、权限和执行服务》
- 《Rich Client 之回归》
- 《Windows .NET Server 2003 加密文件系统》
- 《基于.NET 和 PKI 技术的单点登录系统》
- 《ASP .NET 中的身份验证》
- 《保护你的 Web Services》

☞ Case Study——读者最关心的，无疑是如何用.NET技术开发真正的软件系统。本专题介绍了三个使用.NET开发的真实案例：

- 《使用 Microsoft .NET 实现 Sun Microsystem 的 Java Pet Store J2EE 蓝图应用程序》
- 《一个.NET 平台的应用系统框架》
- 《利用.NET 框架开发应用系统》

希望读者能在我们的 ".NET技术版块" 中学到有用的知识，提高自己的技术水平。也欢迎读者将自己的开发心得整理成文投稿给《程序员》杂志，投稿信箱：editor@csdn.net。

祝您学习愉快！

C# 编程入门（一）

◎ 撰文 / 蒋学锦

.NET 已经正式上路，而.NET 平台中最重要的语言 C# 也广受瞩目。从最近许多读者的来信发现，大家有学习 C# 的需求，但是目前相关的中文入门资料相当缺乏。因为我在《程序员》杂志开设的 Java 专栏已接近尾声，同时为了响应大家对于 C# 的需求，所以我特别开辟新的专栏，以图文并茂及浅显易懂的方式，介绍 C# 编程。

关键词：.NET C# 编译期参考 调试 增量式编译

.NET Framework SDK

微软公司提供免费的.NET Framework SDK，让程序员可以编写.NET 程序，可以直接联线到网站（http://www.microsoft.com）下载它。你必须从此处下载两个文件：

1. .NET Framework SDK，约 131MB
2. .NET Framework Service Pack 1（英文版）

分别执行此二文件，即可安装（install）最新版的.NET Framework SDK。安装过程非常容易，不在此赘述。安装好之后，你就拥有了.NET 环境、C# 编译器 csc 以及相关的软件。

你的第一个 C# 程序

安装好.NET Framework SDK 之后，我们马上写一个 C# 程序来试试。请依照下图中的步骤来开发你的第一个 C# 程序：

图 1

csc 是 C# 的编译器（C Sharp Compiler），你可以在 C:\WINDOWS\Microsoft.NET\Framework\v1.0.3705 路径下找到 csc.exe。使用 csc 的时候，你可以利用一些选项（option）来进行调整。例如：

```
C:\>csc /nologo /out:HelloCS.exe Hello.cs
```

/nologo 的意思是“不要显示 logo”，/out:HelloCS.exe 的意思是“编译后的结果为 HelloCS.exe”；如果不使用 /out: 选项，默认的执行文件的文件名为源码文件名加上“.exe”扩展名。

如果你有太多选项想加进于 csc，有一个更方便的方式：把这些选项都放到一个文字文件中，然后再通过“@”选项来告诉 csc 去读取这个文件即可。例如，我把选项写在一个名为 options.txt 的文件中，内容如下：

```
/nologo
/out:HelloCS.exe
```

然后再利用下面的方式即可命令 csc 云读取这个文件的选项：

```
C:\>csc @options.txt Hello.cs
```

到目前为止，我们已经介绍了几个 csc 选项的用法，如果你想知道 csc 还有哪些选项可用，你可以通过下面的方式：

```
C:\>csc /help
```

本系列文章中，我会在适当的时机陆续介绍这些选项。

编译失败时……

第一次编译就成功，这样的情况不多见，写程序可不会都这么顺利。如果程序出错，导致编译失败，情况又会是如何呢？请看图 2 的试验：

编译时，如果出现错误消息，就表示编译失败，无法产生 exe 文件。这种情况，我们称为编译期错误（compile-time error）。

我们从错误消息的蛛丝马迹中，推断错误的原因，并回去修正源码，反复修正与编译，直到成功为止。如图 3 所示：

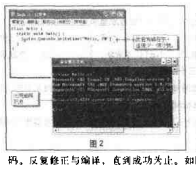


图 2

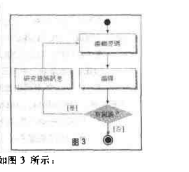


图 3

编译时，不要一看到错误消息就觉得很害怕。事实上，错误消息提供给我们许多有用的资料，可以帮助我们更快修正错误。这些信息包括：

- 文件名称和出错位置：告诉我们错误或警告的源码文件和位置。
- 错误或警告：告诉我们这是错误（error）还是警告（warning）。错误和警告的差别在于其严重等级，错误是很严重的，会导致无法编译，警告是不严重的，如果编译时只有警告没有任何错误，还是会编译成功，产生 exe 文件。
- 消息内容：告诉你错误或警告的原因。通常我们只要阅读这里的叙述，就可以知道如何修正错误。
- 消息编号：许多人都忽略了消息编号的重要性。当你在阅读完“消息内容”后，仍然一头雾水，不知道该如何修正此错误时，你需要通过这个消息编号来取得更详细的说明。

如果要得到“消息编号”的详细说明，你需要查询.NET Framework SDK 的文档（documentation）。这份文档会告诉你非常详细的原因，甚至提供了“错误示范”的范例程序，有了这么丰富的信息，很多问题就算都能迎刃而解，甚至在你对于 C# 语言还不熟悉的情况下，也能“有恃无恐”。

但是编译器毕竟无法猜测程序员的意图，所以有些时候会误判，提供一些让我们不以为然的信息。在下面的例子中，我故意在第五行犯了一个错误，但是编译器不但没找到此错误，还告诉我两个根本就不是错误的错误。我称这种错误为“代罪羔羊”。如图 4 所示：

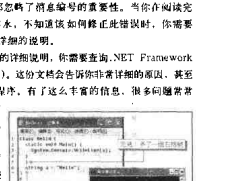


图 4

当你的编译结果是一连串的错误消息时，我建议你不要试图一次就修正全部的错误，你应该先搞定易下手的错误来修正，一次修正几个就好，修正后立刻编译，从错误消息中可以确定刚才的修正动作是正确的，错误数目减少了，心情也会比较笃实，一直重复这样的步骤，直到剩下的错误全是“莫名其妙”的错误，你无力解决的错误。

“莫名其妙”的错误有很大的机率是“代罪羔羊”，也就是说，本身并非错误，真正的错误“另有其人”。为代罪羔羊找出元凶的方式如下：

1. 挑出程序中最前面的第一个代罪羔羊。
2. 从这里开始往前一行一行地进行搜查，通常元凶就躲在附近。

来自编译器的警告

编译器除了可能会发出错误消息，也可能会发出警告消息。警告消息和错误消息很像，只不过错误消息的名称是“error CSxxxx”，而警告消息的名称是“warning CSxxxx”。

如果编译的过程只有警告没有错误，那么还是算编译成功，一样会产生exe文件。警告只是提醒程序员：程序某些地方虽然合乎C#语言的规定，但“不太对劲”。如果你能确定这些“不太对劲”的地方其实都在你的掌握之中，那么你可以不对这些警告做出任何反应。很多数的程序员都直接视现编译器的警告，完全不阅读警告消息。“反正可执行文件都产生出来了，通常都经繁文缛节？”

你可以选择不理会警告消息，也可以干脆来个“眼不见为净”，用“/nowarn:”选项可以连编译某些警告，不显示这些类型的警告。如下例所示：

```
C:\>csc /nowarn:28,1030 Hello.cs
```

这表示我们不希望编译器显示CS0028和CS1030的警告消息。

如果你想把所有的警告都过滤掉，那简要使用的是“/nowarn:”选项，而是“/warn:0”。关于此选项，稍后有更详细的说明。

你也可能和我一样神经质，很重视警告，常可以叫编译器把警告直接当作错误来看，一有警告出现就不允许编译成功。如果你想这么做，你需要“/warnaserror”选项，如下例所示：

```
C:\>csc /warnaserror:28,1030 Hello.cs
```

C#编译器把警告分成四个等级，最严重的警告是第一级（Level1），最轻微的警告是第四级，你可以利用“/warn:”选项来设定你想看的等级。“/warn:0”表示完全不想看到任何警告；“/warn:1”表示想看到第一级的警告；“/warn:2”表示想看到第一级和第二级的警告；“/warn:3”表示想看到第一至三级的警告；“/warn:4”表示想看到第一至四级的警告。如下图5所示：

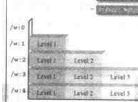


图5


图6

有些时候，警告可不是你说不理就不理的。警告有可能间接导致错误，造成编译失败。请看下面的例子如图6所示：

在这个例子中，编译时出现两个消息，第一个是警告消息，第二个是错误消息。而此错误消息是因为忽视警告所造成的。警告消息告诉我们此程序的Main()格式不对劲（在第二行的第十五个字符处），无法当作程序入口，编译器把Main()当作一般的method来处理，而不是当作程序入口来处理。但编译器又不确定程序员是不是故意要这样做的，所以列出一个警告消息来通知程序员。编译器将程序码编译完之后，发现完全没有任何程序进入点，所以无法编译成功，于是显示出一个错误消息。

编译期参考

请看下面的Hello范例程序如图7所示：

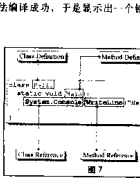


图7

程序中，“class”、“static”等文字是C#语言的关键词（Keyword），也就是有特殊作用的字。C#编译器懂得这些关键词。“Hello, C#”等文字是C#语言的字符串值（string literal）。编译器会原封不动地保留、不做处理。“System”、“Console”等文字则是标识符（identifier）。每个标识符都有其定义，有的用来代表一个数值，有的用来代表一组动作……

Hello正是一个标识符，用来代表一个class（因为前面跟着一个关键词class）。其定义是外层大括号所包围的部分。Main也是一个标识符，用来代表一个method（因为后面跟着一个组圆括号）。其定义是内层大括号所包围的部分。

“System”、“Console”、“WriteLine”也都是标识符，只是它们的定义不在这段源码中。换句话说，我们在本程序中只是使用这些标识符，但是却没有定义这些标识符。那么编译器又怎能知道“System”、“Console”、“WriteLine”的定义是放在哪一个文件中呢？如果编译器不知道它们的定义放在那个文件中，又怎能确定程序员这样的写法是正确的？如果编译器不能确定程序员这样的写法是正确的，又怎能享事地制造出exe文件，以编译成功收场即？

所以我们需要告诉编译器：这几个关键词的定义在哪里，好让编译器能有所参考。要指定编译期参考（compile-time reference），我们必须使用“/reference:”选项，如下所示：

```
C:\>csc /reference:Module1.dll;Module2.dll MyApp.cs
```

上述的编译方式，等于是告诉编译器：MyApp.cs源码中用到了一些外部定义的标识符，编译时请到Module1.dll文件或者Module2.dll文件中找寻其定义。

你也可以改用“/r”选项来取代“/reference:”选项。如果所参考的文件在特殊位置，你可以通过“/lib:”选项来指定文件位置，如下例所示：

```
C:\>csc /lib:c:\lib;c:\ext /r:Module.dll MyApp.cs
```

上述的编译方式，等于是告诉编译器，MyApp.cs源码中用到一些外部定义的标识符，编译时请到Module.dll文件中找寻其定义，其中Module.dll文件位于c:\lib或c:\ext中。

奇怪的是，我们的Hello程序中，虽然使用了外部定义的“System.Console.WriteLine”，而且我们编译时没有使用“/reference:”选项，但还是可以编译成功。那是因为“System.Console”的定义是在“mscorlib.dll”文件中，几乎所有的.NET程序都会需要参考到mscorlib.dll，所以C#编译器会直接去参考mscorlib.dll，不需我们再指示。

作本系列文章中，我们应该只会使用到mscorlib.dll，即使以后你的C#功力大增，可以写出很复杂的程序，在大部分的情况下，你仍不需要使用“/reference:”选项。因为C#编译器执行时，会先去读取“csc.RSP”文件，也就是csc的环境设定文件（configuration file），这个文件的内容如下：

```
r:Accessibility.dll
r:Microsoft.Vsa.dll
r:System.Configuration.Install.dll
r:System.Data.dll
r:System.Design.dll
r:System.DirectoryServices.dll
r:System.dll
r:System.Drawing.Design.dll
r:System.Drawing.dll
r:System.EnterpriseServices.dll
r:System.Management.dll
r:System.Messaging.dll
r:System.Runtime.Remoting.dll
r:System.Runtime.Serialization.Formatters.Soap.dll
r:System.Security.dll
r:System.ServiceProcess.dll
r:System.Web.dll
r:System.Web.RegularExpressions.dll
r:System.Web.Services.dll
r:System.Windows.Forms.dll
r:System.XML.dll
```

你看出端倪了吗？通过这个文件，C# 编译器会自动参考.NET 所有标准的程序库。除非你在程序中参考到了非.NET 标准的特殊程序库，才需要很明确地使用 "/reference:" 选项来指定程序库名称。

如果你不希望 C# 编译器自动使用 csc.RSP 的环境设定，可以利用 "/noconfig" 选项来关闭，如下所示：

```
C:\>csc /noconfig /r:Module.dll MyApp.cs
```

执行期 Bug

即使程序能通过编译器，没有任何错误，顺利地产生 exe 文件，也不能保证程序是对的。程序在执行的过程中，还是可能会有错误的状况出现，这样的状况我们称为 bug。

Y2K（Year 2000）就是一种 bug，它会造成你的银行存款数字异常，属于"逻辑错误"（logic error）。另外，程序执行时出现"异常"（exception），也是一种 bug。逻辑错误通常没有太明显的征兆，所以不易察觉；异常有讯息显示，所以比较容易发觉，也很容易 debug。

发现 bug 的人可能是程序员自己，可能是测试员（tester），也可能是使用者（end user）。发现 bug 的时机越早越好，最好是程序员自己发现 bug，然后立刻将它修正。如果程序员没发现 bug，那么最好在测试阶段由测试员发现 bug，并写了 bug report 间接送到程序员手中，让程序员进行修正。如果 bug 是由花钱买软件的使用者所发现，那可会引来拖想连连。如图 8 所示：

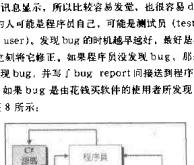


图 8

让我们来实际体会一下 debug 的过程。我写了一个 C# 程序，能进行除法计算，程序如图 9 所示：

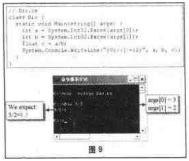


图 9

这个程序编译成功，执行结果却不对。因为 3 除以 2 应该是 1.5，而非这程序所算出来的 1，这就是逻辑错误。

吃烧饼不免会掉芝麻，写程序不免会有 bug。再怎么高明的程序员，也不敢保证写出来的程序不会有 bug。当发现程序有误，bug 隐身其中时，就是 debugger 出动的时候了。越是高明的程序员，越是能够灵巧地使用 debugger，让 bug 无所遁形。

图 10

.NET Framework SDK 提供了一套图形接口的 debugger，叫做"DbgCLR.exe"。要使用这个 debugger 来找寻 bug 之前，你必须准备好 .pdb 文件。利用 C# 编译器的 "/debug" 选项，即可在编译出 exe 文件的同时，产生 pdb 文件，如图 10 所示：

执行 C:\Program Files\Microsoft. NET\FrameworkSDK\GuiDebug\DbgCLR.exe，就可以激活 debugger。在 DbgCLR 的窗口中，选择"Debug"菜单（menu）中的"Program To Debug …"选项，接着出现一个对话框（dialog），在"Program"字段填入要 debug 的可执行文件"C:\Div.exe"，在"Arguments"字段填入要计算的被除数和除数（以空白隔开）"3 2"，按下"OK"按钮。接着，利用"Open File"来开启"C:\Div.cs"。

现在，DbgCLR 已经知道我们欲除错的程序是"Div.exe"，也知道其源码为"Div.cs"，也会去读取"Div.pdb"。一切的工作已经准备就绪，我们可以开始 debug 了！现在不是详细介绍 debug 方法的好时机，以后在适当的机会，我会陆续向各位介绍 debugger 的使用方式与技巧。


图 11

经由下面的修正，此程序终于摆脱此逻辑错误，如图 11 所示：

程序执行的过程中，如果有"异常"（exception）发生，debugger 可以自动激活。例如，我们利用刚刚修正好逻辑错误的程序，来执行"Div 3 a"，程序执行时出问题，产生意外状况，这时候出现的画面，如图 12 所示：


图 12、13

此对话框（dialog）告诉你程序发生了一个名为"System.FormatException"的异常。如果你想要通过 Debugger 来 debug，那么可以在菜单中选择一套 debugger，并按下"Yes"按钮，那么此 debugger 就会自动被激活；如果你不想等驾 debugger，请按下"No"按钮。画面如图 13 所示：

即使不使用 debugger，只要通过上述画面的信息，我们也可以很快视觉问题的症结点：原来是在"C:\Div.cs"的第 19 行出现了格式错误（Format Exception）的问题。

下面是再度修正后的版本，如图 14：

编译成功后，我们采用各种方式来测试执行，都没有问题，如图 15 所示：

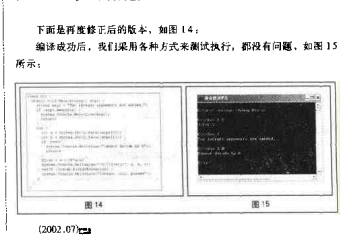


图 14　图 15

(2002.07)

C# 编程入门（二）

——.NET PE 格式

撰文／察学锦

阅读完上次的文章后，你已经知道如何编译C＃程序，也知道如何执行编译后的文件，但是，这一切内部的运作流程又是如何？

.NET程序编译后的文件是exe文件，完全符合原来PE（portable executable）文件的规范，并多了一些.NET特有的扩充。在本次的文章中，我将通过一个实际的例子，来介绍.NET PE文件的格式（format）。

关键词：C＃ .NET PE文件

范例源码

下面是我们的范例源码：

```
using System;

public class Hello {
    public static void Main(String[] args) {
        Console.WriteLine("Hello");}

}
```

利用下面的方式将其编译成exe文件：

csc Hello.cs

现在产生了Hello.exe文件，我们可以通过下列的工具来研究此文件：

- DumpBin（from Visual Studio）
- Depends（form Microsoft Platform SDK）
- PEBrowse Professional：http://www.smidgeonsoft.com（By SmidgeonSoft）
- PEDump：http://msdn.microsoft.com/msdnmag/code02.asp（By Matt Pietrek）
- ILDasm（By Serge Lidin）

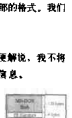
本文使用DumpBin以及ILDasm来研究Hello.exe内部的格式。我们先使用DumpBin，方式如下：

dumpbin /all Hello.exe > Hello.txt

现在你可以打开Hello.txt文件，详细观察。为了方便解说，我不将此文件内容一次列出，而是一边解说，一边列出相关的信息。

.NET PE的文件结构

.NET程序编译后的文件是exe文件，我称为.NET PE文件。它完全符合原来PE文件的规范，所以文件结构同PE文件一样，如右图所示：



下面各小节依序介绍.NET PE文件的结构。

MS-DOS Stub

.NET PE文件的最前面128（0x80）byte是MS-DOS的码，其内容如下：

```
[00000000] 4D 5A 90 00 03 00 00 00 04 00 00 00 FF FF 00 00  MZ?.........??..
[00000010] B8 00 00 00 00 00 00 00 40 00 00 00 00 00 00 00  ?.......@.......
[00000020] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
[00000030] 00 00 00 00 00 00 00 00 00 00 00 00 80 00 00 00  ............?...
[00000040] 0E 1F BA 0E 00 B4 09 CD 21 B8 01 4C CD 21 54 68  ..?..?.?!?.L?!Th
[00000050] 69 73 20 70 72 6F 67 72 61 6D 20 63 61 6E 6E 6F  is program canno
[00000060] 74 20 62 65 20 72 75 6E 20 69 6E 20 44 4F 53 20  t be run in DOS
[00000070] 6D 6F 64 65 2E 0D 0D 0A 24 00 00 00 00 00 00 00  mode....$.......
```

如果使用者是在Windows中执行此程序，就不会执行此部分的程序码。但是如果使用者并非在Windows中执行此程序，而是在DOS中执行此程序，就会执行到此部分的程序码，而显示出下列的警告讯息：

```
This program cannot be run in DOS mode
```

比较有趣的是：文件一开始的两个byte以ASCII来看是“MZ”，就是“Mark Zbikowski”的缩写，他是早期MS-DOS时代的重量工程师之一。另外，在文件0x0000003C到0x0000003F处的值是0x00000080，此值指向PE Signature的位置，在下一节我们就会看到PE Signature的内容。

PE Signature

PE Signature紧接在MS-DOS Stub之后出现，也就是在文件的0x00000080开始的四个byte，其ASCII值必须是“P”、“E”、“null”、“null”，来表示这是一个PE文件，.NET的执行文件完全遵守PE的格式。所以此处也必须符合这条规定，请看下面的dump内容，其中有下划线的部分是PE Signature。

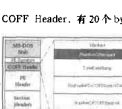
```
[00000080] 50 45 00 00 4C 01 03 00 D5 A6 1E 3D 00 00 00 00  PE..L...??.=....
```

通过DumpBin来观察，如果PE Signature正确的话，会显示下列的讯息：

```
PE signature found
File Type: EXECUTABLE IMAGE
```

COFF Header

紧接着PE Signature之后，就是COFF Header，有20个byte，其配置方式如下图所示：



将其dump出来的内容如下（有下划线的部分）：

```
[00000080] 50 45 00 00 4C 01 03 00 D5 A6 1E 3D 00 00 00 00  PE..L...??.=....
[00000090] E0 00 00 00 0E 01 0B 01 06 00 00 04 00 00 00 00  ....?...........
```

如果通过DumpBin来观察，结果如下：

```
FILE HEADER VALUES
             14C machine (x86)
               3 number of sections
        3D1EA6D5 time date stamp Sun Jun 30 14:36:05 2002
               0 file pointer to symbol table
               0 number of symbols
              E0 size of optional header
             10E characteristics
                   Executable
                   Line numbers stripped
                   Symbols stripped
                   32 bit word machine
```

COFF Header中第一个字段是Machine，对于.NET来说，Machine的值必须是0x014C，COFF Header中第二个字段是“Section的个数”，在此例中为3（0x0003），也就是说PE Header之后会有三个Section

Header，而且在Section Header之后会有三个Section。第二个字段是此文件被建立的时间。第四个字段和第五个字段在.NET PE中是没意义的，所以都设为零。

第六个字段是PE Header的大小，在此例中为224（0x00E0）。对于一般的PE文件来说，其PE Header是96加上Data Directory Table的大小（稍后会介绍Data Directory），但是对于.NET PE文件来说，Data Directory Table的大小固定为16*8=128，所以，.NET PE文件的PE Header固定为224（96+128）。关于PE Header以及Data Directory Table，稍后会有更详细的说明。

第七个字段，也就是最后一个字段的内容是PE文件的特征（Characteristics）。对于.NET的EXE文件来说，此处的值应该是0x010E，表示“可执行，无行号，无符号，32位”；对于.NET的DLL文件来说，此处的值应该是0x210E，表示“可执行，无行号，无符号，32位，DLL”。.NET PE文件一定不会有除错（debug）时需要的行号（Line Numbers）和符号（Symbols），因为这些信息被放在另一个文件中，也就是pdb文件。关于pdb文件，请参考上一期的文章。

PE Header

PE Header紧接着COFF Header之后出现。PE Header内有相当多字段，如下图所示：

我们已经提前在COFF Header中得知PE Header的长度为224（0x00E0）byte。下面先列出前面的96 byte，而把Data Directory Table的128 byte留在下一节讨论。

```
[00000090] 00 00 00 00 E0 00 0E
01 0B 01 06 00 00 04 00 00  ....
.....
[000000A0] 00 06 00 00 00 00 00 00 DE 22 00 00 00 20 00 00
.......?'... ..
[000000B0] 00 40 00 00 00 00 40 00 00 20 00 00 00 02 00 00  .@.....
@.. .....
[000000C0] 04 00 00 00 00 00 00 00 04 00 00 00 00 00 00 00
................
[000000D0] 00 80 00 00 00 02 00 00 00 00 00 00 03 00 00 00
?...............
[000000E0] 00 00 10 00 00 10 00 00 00 00 10 00 00 10 00 00
................
[000000F0] 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 00
................
```

利用DumpBin，得到的信息如下：

```
OPTIONAL HEADER VALUES
             10B magic # (PE32)
            6.00 linker version
             400 size of code
             600 size of initialized data
               0 size of uninitialized data
           22DE entry point (004022DE)
            2000 base of code
            4000 base of data
          400000 image base (00400000 to 00407FFF)
            2000 section alignment
             200 file alignment
            4.00 operating system version
            0.00 image version
            4.00 subsystem version
               0 Win32 version
            8000 size of image
             200 size of headers
               0 checksum
               3 subsystem (Windows CUI)
               0 DLL characteristics
          100000 size of stack reserve
            1000 size of stack commit
          100000 size of heap reserve
            1000 size of heap commit
               0 loader flags
              10 number of directories
```

第一个字段是Magic Number，0x010B表示32位的PE文件，.NET PE使用的是32位的PE文件格式，所以Magic Number必须是0x010B。第二个字段是linker的版本，Microsoft的C++ compiler和linker会将此字段设为7.0，Microsoft其它的编译器（包含C#）会将此字段设为6.0。

要解释第三、第四与第五个字段之前，我必须先说明Section的种类。Section分成Code和Data两大类。在后面讨论到Section Headers时，你会看到本范例共有三个Section，分别是“.text”、“.rsrc”和“.reloc”，占用文件体积分别是0x400、0x400、0x200。任何Section所占用的文件体积都必须是第十一个字段“File Alignment”的倍数。其中“.text”Section就是Code，内含程序码，件何其它名称的Section都是Data。还有一种名为“.bss”的Section，也是Data，但不会出现在文件中，只有可能会出现在内存中。

第三个字段是“.text”Section所占用的文件体积（后面会再详细解释“.text”Section），此处的值是0x00000400。此值必须是第十一个字段“File Alignment”的倍数，第四个字段是“已初始化数据”（initialized data）所占用的文件体积，也就是文件中所有非“.text”与非“.bss”的Section体积总和。此例中的值是0x00000600，也就是第二和第三个Section的体积总和（0x400 + 0x200），第五个字段是“未初始化数据”（uninitialized data）的大小，也就是“.bss”Section的体积。本例的值是0。所以Windows Loader不需要在内存中保留一块额外的空间作为“.bss”Section。

第六个字段是程序进入点（entry point），此例的值是0x000022DE。此值必须加上第九个字段的值（此例为0x00400000）才是内存地址，所以程序进入点的内存地址是0x004022DE。这个程序进入点可不是C#的“public static void Main()”，本文章最后对此会有更详细的解释。

第七个字段是Code（亦即“.text”Section）的起始位置。此例的值是0x00002000，必须加上第九个字段的值才是内存地址，所以“.text”的起始内存地址是0x00402000。

第八个字段是Data（亦即非“.text”的Section）的起始位置。此例的值是0x00004000，必须加上第九个字段的值才是内存地址，所以起始内存地址是0x00404000。

第九个字段是Image Base的位置，也就是这个文件被加载内存的位置。此例为0x00400000。第十六个字段是此文件被加载内存后所占用的体积。此例为0x00008000。所以整个Image的位置在内存的0x00400000～0x00407FFF。

你可以在编译C#程序时自行调整Image Base，只要通过“/baseaddress:”选项（option）即可，方法如下例所示。在下面的例子中，我将Image Base由预定（default）的0x00400000改成0x00500000：

```
csc /baseaddress:0x500000 Hello.cs
```

第十个字段是Section Alignment的大小，此例为0x00002000（8K）。此值必须符合下两点规范：

1. 是page size的倍数。x86的page size是0x1000（4 K）。
2. 是第十一个字段（File Alignment）的倍数。因为文件要被映像（map）到内存中。

第十一个字段是File Alignment的大小，此例为0x00000200（512）。Section在磁盘中所占用空间必须是此值的倍数，此值可以是512（0x200）、1024（0x400）、2048（0x800）、4096（0x1000）、8192（0x2000）（注意：.NET SDK的文档中，此处数字有误）。

你可以在编译时自行调整 Section Alignment 的值，只要通过 "/filealign:" 选项（option）即可。方法如下例所示。在下面的例子中，我将 Section Alignment 由预定的 0x200 改成 0x400：

csc /filealign:0x400 Hello.cs

第十二到第十七的字段只是版本号码。第十六个字段已经在前面介绍过了，是此文件被加载内存后所占用的体积。第十七个字段是所有 Header 所占用的文件体积之总和，包括了 MS-DOS Stub、PE Signature、COFF Header、PE Header（含 Data Directory Table）、Section Headers。特别值得注意的是：在 Section Headers 的后面、Sections 的前面，往往会有一些填补缝隙（padding）的 byte，好让第一个 Section 恰从 File Alignment 的倍数开始。这些 padding byte 也必须算进此字段中。所以我认为：第十七个字段如果改名为 "File Pointer to Sections"，会更名实相符。此例中，第十七个字段的值是 0x00000200。

第十八个字段是 checksum，并不使用，直接设为 0。

第十九个字段是 subsystem，如果这是 Console（文字）模式的程序，则值为 0x0003；如果这是 GUI（图形用户界面）模式的程序，则值为 0x0002。如果你希望此字段的值是 0x0002 的话，你可以利用下面的方式来编译：

csc /target:winexe Hello.cs

如果你希望此字段的值是 0x0003 的话，你可以利用下面的方式来编译：

csc /target:exe Hello.cs

如果编译时不注明 "/target"，则预设的方式是 "/target:exe"。

第二十个字段已经废弃不用，直接设为零。

程序在执行的过程中，每个 thread 都需要一份独立的内存，用来记录 Call Stack。所有的 thread 共需要一份内存，用来记录共享的数据，这份内存称为 Heap。第二十一个字段是 Stack 保留空间，第二十二个字段是 Stack 中有多少空间被设为 commit，第二十三个字段是 Heap 保留空间，第二十四个字段是 Heap 中有多少空间被设为 commit。

第二十五个字段已经废弃不用，直接设为 0。

第二十六个字段是 Data Directory Table 内的数据个数，对于 .NET PE 文件来说，此值一定为 0x00000010（16）。

Data Directory Table 属于 PE Header 的一部份，你可以将其视为第二十七个字段，但是因为 Data Directory Table 数据比较长，所以我将它独立成一个小节来详细介绍。

Data Directory Table

如前所述，.NET PE 的 Data Directory Table 内有 16 条 Data Directory。每一个 Data Directory 包含了 32 位的地址值和 32 位的体积值，如下图所示：

将此 dump 出来，内容如下（有下划线的部分）：

```
[00000f0] 00 00 00 00 00 10 00 00 00 00 00 00 00 00 00 00 ................
[00000100] 88 22 00 00 53 00 00 00 00 40 00 00 28 03 00 00 ."..S....@..(...
[00000110] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
[00000120] 00 60 00 00 0C 0C 00 00 00 00 00 00 00 00 00 00 .`..............
[00000130] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
[00000140] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
[00000150] 00 00 00 00 00 00 00 00 00 00 20 00 00 08 00 00 .......... .....
[00000160] 00 00 00 00 00 00 00 00 08 20 00 00 48 00 00 00 ......... ..H...
[00000170] 00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00 ........text...
```

利用 DumpBin，得到的信息如下：

```
0 [    0] RVA [size] of Export Directory
2288 [   53] RVA [size] of Import Directory
4000 [  328] RVA [size] of Resource Directory
0 [    0] RVA [size] of Exception Directory
0 [    0] RVA [size] of Certificates Directory
6000 [   C] RVA [size] of Base Relocation Directory
0 [    0] RVA [size] of Debug Directory
0 [    0] RVA [size] of Architecture Directory
0 [    0] RVA [size] of Global Pointer Directory
0 [    0] RVA [size] of Thread Storage Directory
0 [    0] RVA [size] of Load Configuration Directory
0 [    0] RVA [size] of Bound Import Directory
2000 [    8] RVA [size] of Import Address Table Directory
0 [    0] RVA [size] of Delay Import Directory
2008 [   48] RVA [size] of CLR Header Directory
0 [    0] RVA [size] of Reserved Directory
```

在本例中，只有五个 Data Directory 有数据，其余的 size 都是 0。这五个 directory 也是 .NET PE 中最常见的五个，其中 "CLR Header" 是 CLR 的 Header，在后面的文章中，我们会详细解说。(2002.08)

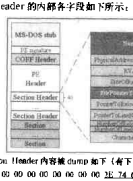
.NET 程序编译后的文件是 exe 文件，完全符合传统 PE（portable executable）文件的规范，并多了一些 .NET 特有的扩充。在本次的文章中，我将通过一个实际的例子，来介绍 .NET PE 文件的格式（format）。

关键词：C# .NET PE 文件

Section Headers

紧接在 Data Directory Table 之后的是整个 Section Header，每个 Section Header 占用 40 byte，至于 Section Header 个数则在前面的 COFF Header 中有指示，以本例来说，此值为 3。

Section Header 的内部各字段如下所示：



第一个 Section Header 内容被 dump 如下（有下划线的部分）：

```
[00000170] 00 00 00 00 00 00 00 00 2E 74 65 78 74 00 00 00 ........text...
[00000180] E4 02 00 00 00 20 00 00 00 04 00 00 00 02 00 00 ..... ..........
[00000190] 00 00 00 00 00 00 00 00 00 00 00 00 20 00 00 60 ............ ..`
```

如果通过 DumpBin 来观察，结果如下：

```
SECTION HEADER #1
.text name
   2E4 virtual size
  2000 virtual address (00402000 to 004022E3)
   400 size of raw data
   200 file pointer to raw data (00000200 to 000005FF)
     0 file pointer to relocation table
     0 file pointer to line numbers
     0 number of relocations
     0 number of line numbers
60000020 flags
         Code
         Execute Read
```

第二个 Section Header 内容被 dump 如下（有下划线的部分）：

```
[00009140]  2E 72 73 72 63 00 00 00  28 03 00 00 00 40 00 00   .rsrc...( ....@..
[00009150]  00 04 00 00 00 06 02 00  00 00 00 00 00 00 00 00   ........ ........
[00009160]  00 00 00 00 40 00 00 40  2E 72 65 6C 6F 63 00 00   ....@..@ .reloc..
```

如果通过 DumpBin 来观察，结果如下：

```
SECTION HEADER #2
   .rsrc name
    328 virtual size
   4000 virtual address (00404000 to 00404327)
    400 size of raw data
    600 file pointer to raw data (00000600 to 000009FF)
      0 file pointer to relocation table
      0 file pointer to line numbers
      0 number of relocations
      0 number of line numbers
40000040 flags
         Initialized Data
         Read Only
```

第三个 Section Header 内容被 dump 如下（有下划线的部分）：

```
[00009170]  00 00 00 00 40 00 00 40  2E 72 65 6C 6F 63 00 00   ....@..@ .reloc..
[00009180]  0C 00 00 00 00 60 00 00  00 02 00 00 00 0A 00 00   ........ ........
[00009190]  00 00 00 00 00 00 00 00  00 00 00 00 42 00 00 00   ........ ....B...
```

如果通过 DumpBin 来观察，结果如下：

```
SECTION HEADER #3
   .reloc name
       C virtual size
    6000 virtual address (00406000 to 0040600B)
     200 size of raw data
     A00 file pointer to raw data (00000A00 to 00000BFF)
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
42000040 flags
         Initialized Data
         Discardable
         Read Only
```

特别注意的是，在此三个 Section Header 的后面，还多了下面的 16 byte。

```
[000001F0]  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........ ........
```

这 16byte 并无实际作用，只是用作缝隙填补（padding）。前面提到过，本例的 File Alignment 为 0x200，有了这 16 byte，后面的 Section 就可以从 0x200 开始，符合 File Alignment 的要求。

这三个 Section Header 的名字分别是 ".text"、".rsrc"、".reloc"，我只解说 ".text" Section Header，因为 ".text" Section 是.NET PE 中最重要的 Section，另外两个大同小异，所以不在此赘述。

Section Header 的前八个 byte 用来记录 Section 的名称。第二个字段是占用的空间，此例的值是 0x000002E4，第三个字段是内存位置，必须加上 Image Base (0x00400000) 才是真正的内存地址，所以此 Section 真正占用的内存地址空间是在 0x00402000 到 0x004022E3。

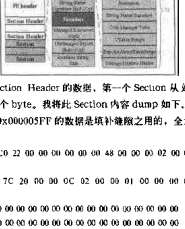
第四个字段是占用的文件体积，此例的值是 0x00000400。第五个字段是 Section 在文件中的起始位置，此例的值是 0x00000200，综合此二字段来看，Section 在文件中位置是在 0x00000200 到 0x000005FF。

请注意，第四个字段和第二个字段的值之所以不同，是因为 File Alignment 为 0x200，Section 在文件中占用体积必须是 0x200 的倍数，这一点前面已经提过了，这里再次强调。文件中用来填补缝隙的数据一律设定为 0，稍后讲解到 ".text" Section 时，我们会看到 ".text" Section 后面有一大块的数据全都是 0。

第六到第九的字段都是 0，我们不予理会。第十个字段是 0x60000020，是由0x40000000、0x20000000、0x00000020的和值，意思分别是"此Section可被读取"、"此 Section 可被执行"、"此 Section 含有可执行码"。

.text Section

在 Section Headers 之后，紧接着有数个 Section，以本例来说，有三个 Section，每个 Section 的内部结构大不相同，我在此只介绍第一个 Section，也就是 ".text" Section。因为 ".text" Section 是.NET PE 中最重要的 Section。".text" Section 的架构方式如下图所示：



根据前面 Section Header 的数据，第一个 Section 从文件的 0x200 开始，共有 0x400 个 byte。我将此 Section 内容 dump 如下，请注意：因为 0x000004E4 到 0x000005FF 的数据是填补缝隙之用的，全为 0，所以我将其省略不列。

```
[00000200]  C0 22 00 00 00 00 00 00  48 00 00 00 02 00 00 00   ?"......  H.......
[00000210]  7C 20 00 00 0C 02 00 00  01 00 00 00 01 00 00 06   |.......  ........
[00000220]  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........  ........
[00000230]  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........  ........
[00000240]  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ........  ........
[00000250]  13 30 01 00 0B 00 00 00  00 00 00 00 72 01 00 00   .0......  r...
[00000260]  70 23 02 00 00 0A 2A 00  13 30 01 00 07 00 00 00   p#....*.  .0... ..
[00000270]  00 00 00 00 02 28 03 00  00 0A 2A 00 42 53 4A 42   .....(..  *.BSJB
[00000280]  01 00 01 00 00 00 00 00  0C 00 00 00 76 31 2E 30   ........  v1.0
[00000290]  2E 33 37 30 35 00 00 00  00 00 05 00 6C 00 00 00   .3705...  l...
[000002A0]  DC 00 00 00 23 7E 00 00  48 01 00 00 7C 00 00 00   ?...#~..  H...|...
[000002B0]  23 53 74 72 69 6E 67 73  00 00 00 00 C4 01 00 00   #Strings  ....?...
[000002C0]  10 00 00 00 23 55 53 00  D4 01 00 00 10 00 00 00   ....#US.  ?.......
[000002D0]  23 47 55 49 44 00 00 00  E4 01 00 00 28 00 00 00   #GUID...  ?...(...
[000002E0]  23 42 6C 6F 62 00 00 00  00 00 00 00 01 00 00 01   #Blob...  ........
[000002F0]  47 15 00 00 09 00 00 00  00 FA 01 33 00 02 00 00   G.......  ?.3....
[00000300]  01 00 00 00 03 00 00 00  02 00 00 00 02 00 00 00   ........  ........
[00000310]  01 00 00 00 03 00 00 00  01 00 00 00 01 00 00 00   ........  ........
[00000320]  01 00 02 00 00 00 0A 00  01 00 00 00 00 00 00 00   ........  ........
[00000330]  24 00 1D 00 06 00 4F 00  3C 00 06 00 68 00 1D 00   $.....O.  <...h..
[00000340]  00 00 00 00 01 00 00 00  00 00 01 00 01 00 01 00   ........  ........
[00000350]  10 00 2B 00 00 00 05 00  01 00 01 00 50 20 00 00   ..+.....  P ..
[00000360]  00 00 96 00 31 00 0A 00  01 00 68 28 00 00 00 00   ..?.1...  h(....
[00000370]  86 18 36 00 10 00 02 00  00 00 01 00 63 00 11 00   ?.6.....  c...
[00000380]  36 00 14 00 19 00 70 00  1A 00 09 00 36 00 10 00   6.....p.  ....6...
[00000390]  2E 00 0B 00 1F 00 04 80  00 00 00 00 00 00 00 00   .....?..  ........
```

```
[000003A0] 00 00 00 00 00 00 00 00 2B 00 00 00 01 00 00 00 ........
+......
[000003B0] E4 0C 00 00 00 00 00 00 01 00 14 00 00 00 00 00 ?....  ....
[000003C0] 00 00 00 00 00 3C 4D 6F 64 75 6C 65 3E 00 48 65 ....
<Module>.He
[000003D0] 6C 6C 6F 2E 65 78 65 00 6D 73 63 6F 72 6C 69 62 llo.exe.
mscorlib
[000003E0] 00 53 79 73 74 65 6D 00 4F 62 6A 65 63 74 00 48 System
Object.H
[000003F0] 65 6C 6C 6F 00 4D 61 69 6E 00 2E 63 74 6F 72 00 ello.Main..
ctor.
[00000400] 53 79 73 74 65 6D 2E 44 69 61 67 6E 6F 73 74 69 System.
Diagnosti
[00000410] 63 73 00 44 65 62 75 67 67 61 62 6C 65 41 74 74 cs.
DebuggableAtt
[00000420] 72 69 62 75 74 65 00 61 72 67 73 00 43 6F 6E 73 ribute.args.
Cons
[00000430] 6F 6C 65 00 57 72 69 74 65 4C 69 6E 65 00 00 00 ole.
WriteLine...
[00000440] 00 0B 48 00 65 00 6C 00 6C 00 6F 00 00 00 00 00 ..H.e.l.l.
o......
[00000450] FF 42 1D 60 D6 B0 96 48 8B ED A7 D7 11 7D CF 34 ?B'???
H????.}?4
[00000460] 00 08 B7 7A 5C 56 19 34 E0 89 05 00 01 01 1D 0E ..?z\V.4
??......
[00000470] 03 20 00 01 05 20 02 01 02 02 04 00 01 01 0E 06 . ..
........
[00000480] 01 00 00 01 00 00 00 00 B0 22 00 00 00 00 00 00 ........?
".......
[00000490] 00 00 00 00 CE 22 00 00 00 20 00 00 01 00 00 00 ...?"...
..
[000004A0] 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
[000004B0] C0 22 00 00 00 00 00 00 00 00 0C 00 00 00 00 00 ?
"..........
[000004C0] 00 00 5F 43 6F 72 45 78 65 4D 61 69 6E 00 6D 73 ..
_CorExeMain.ms
[000004D0] 63 6F 72 65 65 2E 64 6C 6C 00 00 00 00 00 FF 25 coree.
dll.....?%
[000004E0] 00 20 40 00 00 00 00 00 00 00 00 00 00 00 00 00 . @.........
[000004F0] ~ [000005FF] 全为 00
```

又根据Section Header的数据，第一个Section会被map到内存0x2000处（必须加上Image Base 0x00400000），且占用内存0x2F4个byte。如果你通过DumpBin来观察，显示出来的正是内存的结果：

```
RAW DATA #1
  00402000: C0 22 00 00 00 00 00 00 48 00 00 00 02 00 00 00  A".....
H.......
  ... 略 ...
  004022E0: 00 20 40 00 00 00 00 00 00 00 00 00 00 00 00 00  .
@...........
```

我将内存内容省略不列出，因为都和前面文件dump的内容一样。由于".text" Section比较复杂，所以分成下面数小节来讨论。

Import Address Table

".text" Section中一开始先出现的是Import Address Table。前面Data Directory Table的第十三个字段正是指向此处，共有八个byte，如下所示（标示下划线处）。

```
00402000: C0 22 00 00 00 00 00 00 48 00 00 00 02 00 00 00
A"......H.......
```

CLR Header

紧接着是CLR Header。其内部组织方式如下：

前面Data Directory Table的第十五个字段正是指向此处，此例共有0x48个byte，如下所示（标示底线处）。

```
00402000: C0 22 00 00 00 00 00 00 48 00 00 00 02 00 00 00  A"....
H.....
00402010: 7C 20 00 00 0C 02 00 00 01 00 00 00 01 00 00 06
.... ......
00402020: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ........ ......
00402030: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ... ........
00402040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  .... .......
```

DumpBin对其解释如下：

```
clr Header:
          48 cb
        2.00 runtime version
        207C [     20C] RVA [size] of MetaData Directory
             flags
     6000001 entry point token
           0 [       0] RVA [size] of Resources Directory
           0 [       0] RVA [size] of StrongNameSignature Directory
           0 [       0] RVA [size] of CodeManagerTable Directory
           0 [       0] RVA [size] of VTableFixups Directory
           0 [       0] RVA [size] of ExportAddressTableJumps Directory
```

DumpBin省略了最后一个字段"Managed Native Header"没有列出，因为目前并不使用此字段，一律设为0。

特别注意第三个字段，这是Metadata的位置。此例的Metadata是位在内存中的0x0040207C到0x00402287。

对于.NET的程序来说，Metadata和IL Code是整个执行文件的重点所在。Metadata是程序结构以及相关数据，IL Code则是程序码。

第五个字段也很值得注意，它指向.NET程序进入点的method，也就是C#的"public static void Main()"。

Metadata

从前面得知Metadata是在内存中的0x0040207C到0x00402287，将其内存dump如下：

```
00402070: 00 00 00 00 02 28 03 00 00 0A 2A 00 42 53 4A 42  ....(...
*.BSJB
00402080: 01 00 01 00 00 00 00 00 0C 00 00 00 76 31 2E 30  ..... ......
v1.0
  .. 略...
00402270: 03 20 00 01 05 20 02 01 02 02 04 00 01 01 0E 06  . ...
..........
00402280: 01 00 00 01 00 00 00 00 B0 22 00 00 00 00 00 00  ........'
".....
```

为了节省篇幅，我将中间的部分省略。如有需要，你可以参考前面".text" Section小节中所dump的文件数据，等于是文件的0x0000027C到0x00000487。

Metadata的结构十分复杂，所以我无法在此详细讨论。一言以蔽之，Metadata就像是一个数据库（database），内涵许多table，且table之间互相reference。目前，.NET PE文件的Metadata中可以使用的table有下列38种：

Module、TypeRef、TypeDef、FieldPtr、Field、MethodPtr、Method、ParamPtr、Param、InterfaceImpl、MemberRef、Constant、CustomAttribute、FieldMarshal、DeclSecurity、ClassLayout、FieldLayout、StandAloneSig、EventMap、EventPtr、Event、PropertyMap、PropertyPtr、Property、MethodSemantics、MethodImpl、ModuleRef、TypeSpec、ImplMap、FieldRVA、ENCLog、ENCMap、Assembly、AssemblyRef、File、ExportedType、ManifestResource、NestedClass

我会在以后适当的时机，陆续向各位介绍这些Metadata Table。

IL Code

IL Code的区域并无很明确的边界，完全仰赖Metadata的Method Table来记录那一块内存地址是哪一个method的IL Code。以后介绍到Method Table时，这一点观念会更清楚。(2002.09)