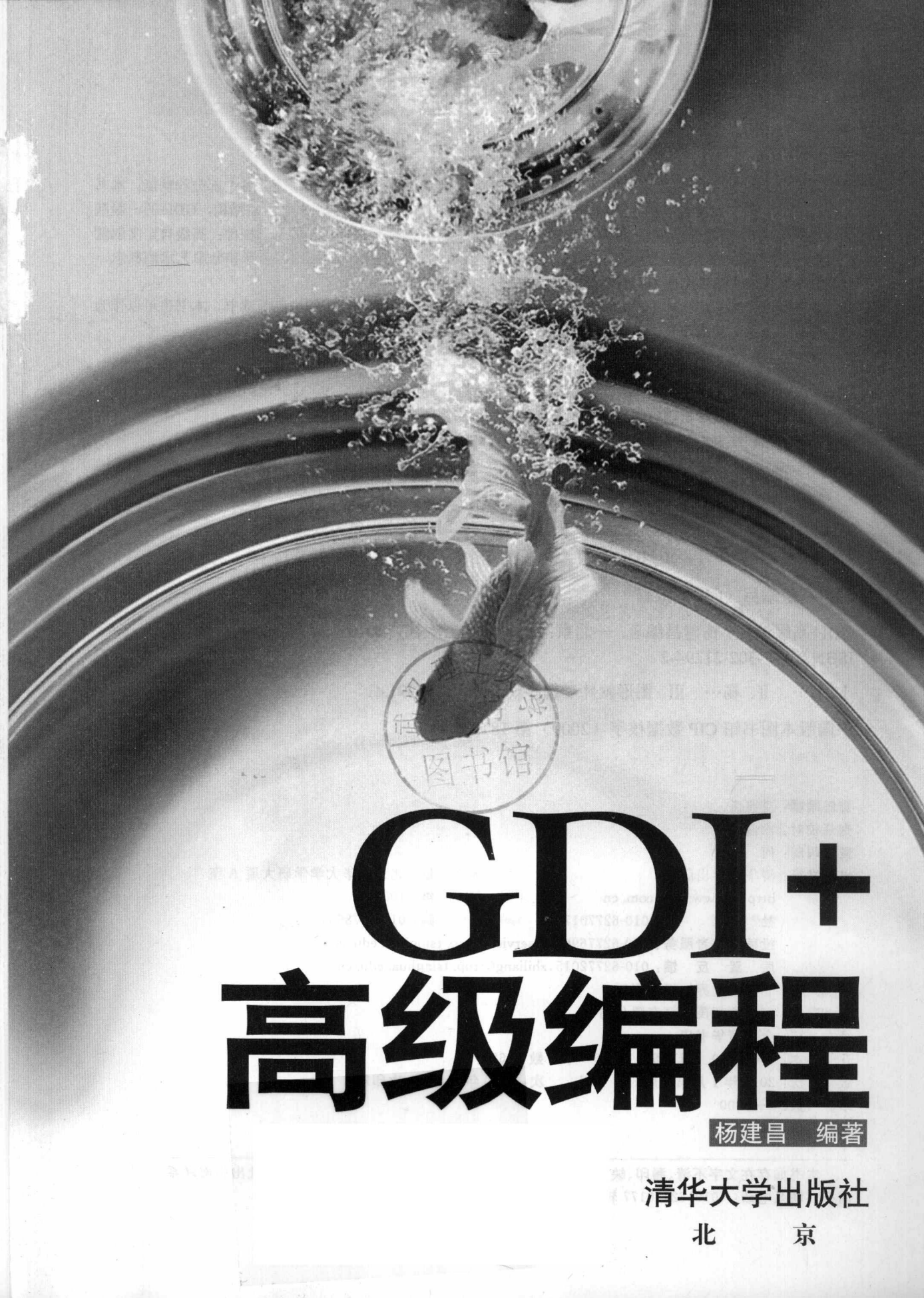
GDI+ 高级编程

杨建昌 编著

清华大学出版社



GDI+ 高级编程

杨建昌 编著

清华大学出版社
北 京

内 容 简 介

本书立足GDI+的巧妙运用，以制造实用的用户界面为目标来介绍GDI+图形库的各个部分和功能。本书共分为7章，每一章介绍了特定主题内容。全书主要内容包括GDI+的概念，以及层次结构，GDI+的一般基础性能应用——Windows基础控制的制作，包括带有视觉样式和没有视觉样式的基础控件；高级自定义创意控件的一般流程和方法；GDI+对字体和文字的支持；GDI+在处理动画上的应用；分形和分形艺术的概念，以及分形在现实生活中的应用；数字图像处理的相关技术。

本书采用C#语言描述，面向对.NET框架有一定认识并有一定编程经验的中高级读书。本书也可以作为对GDI+和C#感兴趣的在校师生的参考读物。

图书在版编目（CIP）数据

GDI+高级编程 / 杨建昌编著. —北京：清华大学出版社，2010.1
ISBN 978-7-302-21294-2

Ⅰ. G…　Ⅱ. 杨…　Ⅲ. 图形软件－程序设计　Ⅳ. TP391.41

中国版本图书馆CIP数据核字（2009）第181975号

责任编辑：夏兆彦
责任校对：徐俊伟
责任印制：何　芊
出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦A座
http://www.tup.com.cn　　**邮　　编**：100084
社　总　机：010-62770175　　**邮　　购**：010-62786544
投稿与读者服务：010-62776969，c-service@tup.tsinghua.edu.cn
质　量　反　馈：010-62772015，zhiliang@tup.tsinghua.edu.cn
印 刷 者：北京市世界知识印刷厂
装 订 者：三河市新茂装订有限公司
经　　销：全国新华书店
开　　本：190×260　**印　张**：34.75　**字　数**：862千字
版　　次：2010年1月第1版　　**印　　次**：2010年1月第1次印刷
印　　数：1～5000
定　　价：59.00元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：(010)62770177转3103　　产品编号：034384-01

FOREWORD

前言

GDI+（Graphical Device Interface Plus）伴随着.NET 诞生已经有好几年了。随着.NET 框架的升级，GDI+的功能也越来越完善。作为 Windows 的显示子系统，它负责完成 Windows 窗体和控件的绘制工作，是 Windows XP 以及 Windows 2003 等操作系统的用户界面层核心。与它的前身 GDI 图形系统相比，它引入了 2D 图形的反锯齿、渐变画刷、基数样条、浮点数坐标，以及 Alpha 混合支持，并支持多种图像格式等。但是，GDI+没有任何硬件加速，这在性能上造成了一些损失。因此，在某些场合仍然需要采用 GDI 与 GDI+混合编程的模式以获得最优的性能。在编程模式上，GDI+基于非状态的图形对象的用法使程序员可以简单地以面向对象的编程方式去完成每一个绘图任务，而不必花费大量精力去关注各种句柄和设备上下文的状态，这样的好处是显而易见的。

在 Windows Vista 操作系统中，提供了一种全新的显示子系统——WPF。WPF 提供了更加丰富和高效的多媒体应用。但是，GDI+作为当今 Windows Forms 应用程序的主流图形库仍然会在相当长的一段时间内存在。

GDI+图形库中包含了大量的类型、结构和枚举定义，本书没有逐一介绍每一个类，也没有介绍 GDI+图形库中的每一个相关类型。本书立足 GDI+的巧妙运用，以创建实用的、耐用的用户界面为例来介绍 GDI+图形库的各个部分和功能。本书采用 C#语言描述，不是面向.NET 的初级学者的，而是面向对.NET 框架有一定认识并有一定编程经验的中高级读者。另外，对 GDI+和 C#感兴趣的在校师生也可以选择阅读本书。

1. 本书内容

本书共分为 7 章，每一章以特定主题为中心、以贯穿始末的程序案例为主线讲解相关知识点。每一章中都给出了大量的源程序清单。其中，大部分源程序都具有极强的实用性，可以直接应用于相关的项目中。出于对篇幅的考虑，在书中介绍相关的知识点时，只会罗列核心的代码清单，完整的源程序可以在随书附带的光盘上找到。全部的源程序都已经在 Visual Studio 2005 中编译通过，在 Visual Studio 2008 上通常也可以直接运行。

第 1 章 介绍 GDI+的概念，以及层次结构，并介绍 Visual Studio 2005 集成开发环境的常用工具和操作，以及 C#语言 2.0 版本所提供的一些新特性。最后介绍 GDI+图形库的简单结构，并用一个小例子说明使用 GDI+图形库进行绘图的简单过程。

第 2 章 介绍 GDI＋的一般基础性应用——Windows 基础控件的制作，包括带有视觉样式和没有视觉样式的基础控件，介绍分层窗口的概念和应用，并给出利用分层窗口实现 Alpha 混合的异形窗口的简单封装。本章还用了比较大的篇幅来介绍有关颜色空间处理的知识，这可以用于增强 GDI+比较弱的颜色支持。本章使用的 GDI+技术包括线性渐变画刷、路径渐变画刷、路径、区域、剪辑和失效、自定义光标等。

第 3 章 介绍更加高级的自定义创意控件的一般流程和方法。与第 2 章介绍的传统的 Windows 基础控件相比，本章的内容更加新颖。利用本章介绍的知识，就可以使用 GDI+完成高级自定义控件的界面绘制和与用户交互的处理。然后介绍利用.NET 框架实现控件对 RAD 的支持，并简单介绍 Visual Studio IDE 的开放结构与创建自定义窗体设计器的步骤。最后，以创意控件的制作为目标介绍控件在用户交互时对音效的支持。本章涉及的内容较多，例如 GDI+坐标系统、矩阵的仿射变换、双倍缓存等以及相关的一些.NET 技术。

第 4 章 介绍 GDI+的另一个重要方面，即对字体、文本和打印的支持。首先介绍与字体相关的一些概念，以及当今常用的字体格式，例如 TrueType、OpenType，以及字体渲染技术。然后介绍如何使用 GDI+进行文本的格式化输出，包括文字修整和排版。接下来介绍如何绘制特效文本的技术：首先是对文字进行精确的大小测量，以便对文字的绘制坐标进行精确的计算，其次是建立 3D 的特效文本和将文字沿路径排版的技术。在本章的最后介绍使用 GDI+进行格式化打印的技术。

第 5 章 介绍 GDI+在处理动画上的应用。本章将介绍一些可以有效减少使用 GDI+处理动画过程中出现的闪烁现象的算法和技术，以及如何突破 GDI+性能瓶颈。具体的内容分为播放 GIF 动态图像、幻灯片效果实现、屏幕保护程序以及游戏。在本章提供的源程序清单中，包含大量实用的、有趣的、新颖的算法和技术。

第 6 章 介绍分形和分形艺术的概念，以及分形在现实生活中的应用。由于分形的快速发展，以及广阔的应用前景，希望通过本章的介绍能将读者引入探索奇幻莫测的分形世界的艺术殿堂。在内容的安排上，首先简单介绍分形的概念和相关的理论，然后结合 GDI+技术实现一个分形图像的生成模型，以及分形世界中最美轮美奂的复动力系统。

第 7 章 介绍数字图像处理的相关技术。本章中给出了大量的专业级数字图像处理应用。首先简单介绍 GDI+提供的数字图像处理功能接口，以及其一般用法，并在此基础上结合 Exif 规范实现通过 GDI+图形库对数码照片 Exif 信息的读取和写入。接下来介绍各种数字图像处理的原理和一般算法，例如图像的点运算，包括 Gamma 校正、亮度与对比度调节、色相与饱和度调节、ColorMatrix 的使用以及图像的直方图与直方图均衡算法等，并介绍滤波和压缩中使用较多的更加高级的图像变换原理和算法实现，以及图像的邻域运算和卷积操作。然后系统性地介绍图像色彩混合中使用的各种混合模式，这对于多个图像的混合叠加是非常强大而有用的功能。最后介绍图像后期处理中使用的一些特效滤镜的算法原理和实现，并探讨数字图像处理系统中相关的一些基本技术，如用户选区的平滑与羽化等。

2. 调试源程序

本书中介绍的完整源程序都可以在随书附带的光盘中找到。这些源程序全部由笔者亲自编写并经过笔者的调试，读者可以在自己的计算机上放心地运行。在运行这些编译好的源程

序时，还需要：

- Windows 2000 及以上的操作系统，例如 Windows XP、Windows 2003、Windows Vista 等。
- .NET Framework 2.0，或者在计算机上安装了 Visual Studio 2005 或后续版本。

本书中的源程序均是利用 Visual Studio 2005 开发的。若需要调试附书的源代码，则需要安装兼容调试的集成开发环境，例如 Visual Studio 2005、Visual Studio 2008，或者 SharpDevelop 2.0 以上版本等。

3. 附言

虽然作者在编写本书的过程中投入了大量的精力，但毕竟水平有限，所以书中难免存在不足之处。另外，本书在内容的安排上也不可能照顾到每一位读者的兴趣爱好而做到面面俱到，在此向大家表示深深的歉意，并恳请广大的读者朋友和高校师生批评赐教。

4. 作者简介

高级程序员、系统分析师，曾就职于国内某大型知名软件公司，现就职于国内某大型知名金融企业研发中心。对.NET 框架、数据库、图形图像编程有一些经验积累，在开发软件时提倡“怀疑一切”的编程思想，对于系统分析则注重开放式系统设计、数据和用户交互的智能化处理。

CONTENTS

目录

第1章 .NET Framework 2.0 图形设备接口

在 Microsoft 将操作系统升级到 Windows XP 以及 Windows 2003 后，原来的 GDI（Graphical Device Interface）也同时得到更新，升级为 GDI+（Graphic Device Interface Plus），在之前的操作系统中必须进行安装才能使用其提供的新特性。作为 Windows 操作系统的显示子系统，GDI+（GDI）的重要性是不言而喻的，其提供完成 Windows 操作系统从屏幕窗口显示（GUI）、图形图像渲染到打印机、绘图仪输出等一系列显示工作。

在 Windows 操作系统中，GDI/GDI+的结构如图 1.1 所示。GDI+（GDI）基于设备驱动程序接口（DDI），使得 GDI+（GDI）完全是设备无关的，因而调用者没有必要去关心图形硬件的类型和相应的驱动程序接口，可以使用同样的代码在不同的硬件平台上实现完全相同的功能和几乎一致的效果，Windows 会为用户实现与硬件的通信，这使得程序的编写变得极为容易。

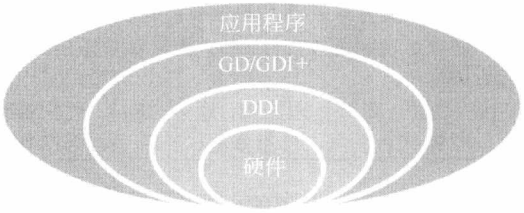


图 1.1　GDI/GDI+结构

随着 Visual Studio.NET 2005 的发布，GDI+也随之升级到 2.0 版本。.NET 对 GDI+进行了全面的封装，形成一个庞大而且功能比较完善的类库，直接体现为 System.Drawing.dll 程序集。本书将使用 Visual Studio .NET 2005 和 C＃作为开发工具。

1.1　GDI+Vs GDI、DirectX、OpenGL

作为 GDI 的升级版本，GDI+不仅提供了许多 GDI 所不具备的新特性，而且在性能、易用性以及 GDI 对象的管理上也做得比较出色。比如，在使用 GDI 时，必须获取设备环境——不必知道设备怎样实现渲染的细节，但是需要了解使用的设备。GDI+更加先进，使得使用者完全不用了解设备，而采用统一的 GDI+对象即可完成绘图任务。GDI+对 GDI 的改进可以概括为以下几个方面，如表 1.1 所示。

表 1.1 GDI 与 GDI+对比

区别	1. GDI 是硬件加速的；而 GDI+不是，而且 GDI+2.0 较 GDI+更快 2. GDI 是有状态的；而 GDI+是无状态的 3. GDI 绘图要使用设备环境和句柄；而 GDI+全部交由 Graphics 类管理 4. GDI 绘图时可以使用 SelectObject 频繁切换图形对象；而 GDI+对象是图形对象独立的 5. GDI 中存在一个“当前位置”，目的是提高绘图性能；而 GDI+取消了它，以避免绘图时不确定这个“当前位置”而带来非预期的错误 6. GDI 总是将画笔和画刷绑定在一起，即便不需要填充一个区域也必须指定一个画刷；而 GDI+则可以使用不同的函数分开使用画笔和画刷

除了 GDI+提供的 API 更灵活和更易于使用外，还提供了许多 GDI 所不支持的新特性，如表 1.2 所示。

表 1.2 GDI+新特性

新特性	1. 改进了颜色管理。GDI+不仅提供了更多可供选择使用的颜色，使其支持 Alpha 通道合成运算，而且还保持了与其他颜色的兼容性 2. 绘图支持反锯齿。通过设置 GDI+对象的相关属性，GDI+可以与相关的显示驱动程序搭配完成图形绘制时的反锯齿功能，使得绘制的图形更平滑、美观，而整个过程是由 GDI+对象自动计算完成的 3. 提供渐变画刷。GDI+拓展了 GDI 的功能，提供线性渐变和路径渐变画刷来填充一个图形、路径和区域，甚至也可用来绘制直线、曲线等 4. 独立的路径对象。GDI+使用 Graphics 对象来进行绘图操作，并将路径操作从 Graphics 对象分离出来，提供一个 GraphicsPath 类供用户使用，用户不必担心路径对象会受到 Graphics 对象操作的影响，从而可以使用同一个路径对象进行多次的路径绘制操作 5. 样条曲线。GDI+封装了绘制基数样条曲线和贝塞尔样条曲线的方法 6. 变形和矩阵运算。GDI+提供了功能强大的 Matrix 类来实现矩阵的旋转、错切、平移、比例等变换操作，以便产生复杂的新图形 7. 多图片格式的支持。GDI+改进了图像处理能力，通过 GDI+，用户能够访问多种格式的图片文件、转换文件格式等，还能进行图像重新着色、色彩修正、消除走样等图像处理

DirectX 也是由 Microsoft 开发的一套完整的、比 GDI 和 GDI+复杂得多的多媒体应用解决方案 API 库。DirectX 最初的出现可以追溯到 Windows 3.1 时代。当时的游戏大部分是运行在 DOS 平台上，而且每一种 DOS 游戏都只支持特定的显示芯片。软件开发商不得不根据不同的显卡来编写不同的代码，硬件开发商则不得不为那些大牌应用程序，如 AutoCAD 等开发驱动程序，所以当时没有什么可以保证显卡和应用程序能够一起工作。而且，当时的 Windows 3.1 对于图形图像的处理能力极弱，难堪大型游戏的应用。为了鼓励游戏厂商将游戏开发向 Windows 平台进行迁移，Microsoft 决定编写一套 API 提供高速的图形图像处理能力。首先，Microsoft 与硬件厂商达成协议，硬件厂商的职责除了包括制造芯片板卡以外还要提供一个驱动程序，这类驱动程序需提供一个一致的接口来调用或查询硬件所提供的功能，这个接口称为 DDK（Device Development Kit，设备开发工具）。其次，Microsoft 向软件开发商提供一套统一的开发接口，只要使用这种开发接口开发的软件运行在 Windows 平台下，无论计算机使用何种显卡，软件均能正常运行。这个开发接口就是 SDK（Software Development Kit，软件开发工具）。最后，就是 DirectX 本身，它结合自身提供的附加功能，将 DDK 的功能封装成可以直接使用的统一界面 SDK，提供给软件开发商，这样的好处是显而易见的，它统一了先前出现的混乱局面。但是，出于各种原因，直到 DirectX 5.0 出现，才获得广泛的应用。DirectX

发展到现在，功能体系越来越庞大和复杂，已经不再是一个单纯的图形函数库。它完全面向Windows 平台的多媒体编程，包含 2D/3D 图形输出、音频与音乐、视频回放与录制、输入设备管理、网络连接等支持组件，从大型游戏、流媒体播放到 CAD 应用，都能轻松自如地应付，并提供了许多特效处理功能。现在的最新版本为 DirectX 10.1。

OpenGL（Open Graphics Library）是由 SGI（Silicon Graphics Inc.）公司开发的一套开放图形库。它是一套跨平台、不依赖于编程语言的 2D/3D 图像编程接口，应用在包括 CAD、GIS、媒体娱乐、游戏开发、工程科学及虚拟现实等行业领域中。与 DirectX 相比，OpenGL 只是一个图形函数库，并不包含音视频处理、输入设备管理、网络连接等支持 API。OpenGL 以强大的功能、良好的移植性、高效的图像处理方法以及硬件扩展等特性在图形工作站等专业高端图形处理领域得到广泛应用而成为事实上的行业标准。由于 Microsoft 为了使 DirectX 在游戏开发市场竞争中更具有优势，于是采取了一系列措施来限制 OpenGL 的发展，例如拒绝在操作系统中发布支持 OpenGL 的驱动程序等。OpenGL 的发展一直处于一种近乎停滞的状况，而 DirectX 的发展则是蒸蒸日上，在 PC 领域成了完全的霸主。虽然如此，OpenGL 在高端图形处理领域仍旧是豪气十足，不可取代。

DirectX 与 OpenGL 虽然功能强大，但是其复杂程度是 GDI+远远不能比拟的。DirectX 与 OpenGL 属于重量级的图形库，而 GDI+只是轻量级的图形库，但是却提供了友好简单的编程界面，易学易用。DirectX 与 OpenGL 面向三维图像以及音视频等多媒体处理，直接面向显卡编程，比较适合大型多媒体应用程序，比如游戏、三维建模工具等的开发，在二维图像处理方面没有提供绘图接口和函数，只有位图操作。GDI+则集成了众多图形相关的类，并且提供了一组相当丰富的绘图函数，在 Windows 窗口以及桌面编程领域，可以完全胜任二维图元的绘制操作，也是 Windows 操作系统中窗口编程的绝对主力。

DirectX、OpenGL 以及 GDI+等图形库都有各自最适合的应用领域。由于 GDI+更注重系统的简洁性和兼容性，从而更易学易用。如果应用程序或游戏中不要求太多的 3D 效果，GDI+同样可以非常适合该类应用程序和很多类型游戏的开发。在常规的 Windows 窗口应用程序中，使用 GDI+是最好的选择。结合程序员对 GDI+图形库的扩展，GDI+可以在多个应用领域进行多种图形图像、地理信息、包装装潢、广告设计，以及游戏等应用的开发。

1.2 Visual Studio 2005 集成开发环境与 Visual C#

本书不是面向.NET 的初级学者的。阅读本书前读者应熟悉 Visual Studio 2005 IDE 的基本操作，并具备一定的 Visual C＃编程经验。本节将简要介绍 Visual Studio 2005 以及 Visual C＃2.0 的新特性，给读者一个快速学习 IDE 以及 C＃2.0 的参考。为了节省篇幅，本书后面的章节中将不再详细介绍所使用的超出 GDI+范围以外的知识，如果读者有兴趣，可以参阅其他的专业书籍。

Visual Studio 2005 集成开发环境提供了许多新的特性，例如增加了集成开发环境设置的导入导出，自定义代码段及模板，自动保存及恢复，文件修订，代码重构，类可视化设计器，可以引用.exe 程序集文件，编辑并继续调试等功能。

对象浏览器（图 1.2）是一个很好用的工具。庞大的.NET 框架包含超过 3000 个类，以及

数量众多的方法、属性、接口、委托以及事件等。对象浏览器提供了一种途径使得开发人员可以把精力放在所关心的.NET 程序集上，而不用花许多时间去检索大量的文档、书籍和编写大量测试代码，能够从各种组件中快速检查和发现对象及其成员。这些对象和成员甚至可以是隐藏的。使用对象浏览器不仅可以查看解决方案中的项目、引用、第三方.NET 类库，还可让开发人员根据命名空间、对象类型、字母顺序等条件来过滤与排序数据等。充分使用本工具可以大大提高学习某类库和撰写实用代码的效率。

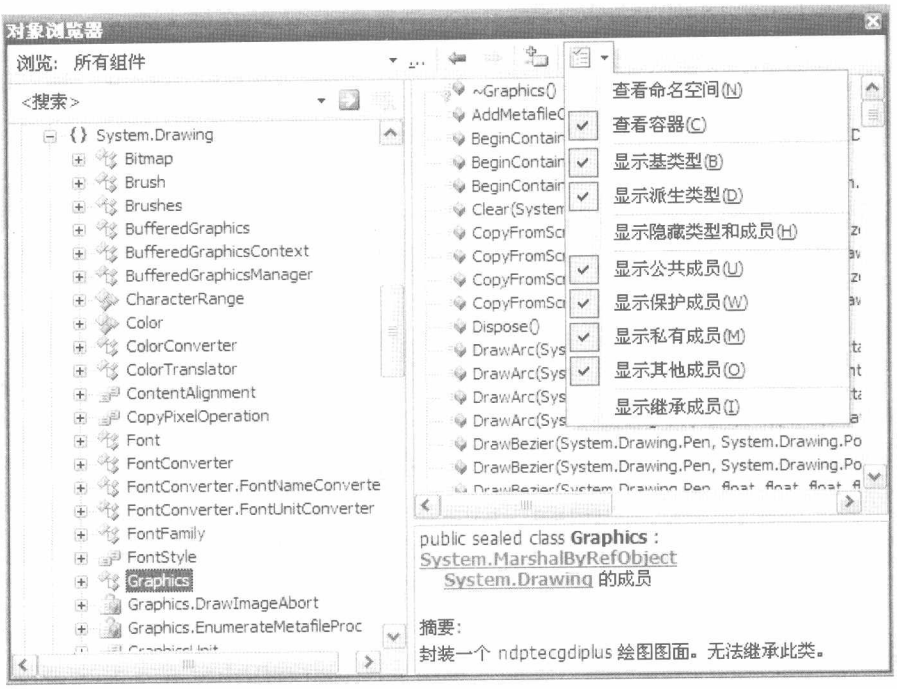


图 1.2　对象浏览器

类设计器（图 1.3）是 Visual Studio 2005 提供的又一个重要的工具，使得开发人员可以从软件架构的角度来考察和编辑程序代码。类设计器包含一些专门设计的功能，这些功能将有助于重构代码、方便地重命名标识符以及重载方法，可以自动生成类和结构，并通过自动生成存根实现接口。对类图的更改将直接反映到代码中，且对代码的更改会直接影响设计器的外观。设计器和代码之间的同步关系使得可视化创建和配置复杂的 CLR 类型变得容易。

类图不是项目的可编译部分，而是构建和编辑类的辅助工具。一旦创建了类图，通过将其从“类”视图拖到类设计表面，可以将现有类添加到类设计器。通过将其从工具箱拖到类设计器，也可添加新类和其他类型。一旦添加到类设计器，类和其他类型可以用图形表示，可以选择和操纵其图形。在类设计器中选择图形会使其相关信息显示在“类详细信息”窗口中。类图本身只存储可视信息，不包含任何关于代码内容的信息。删除类图文件不会丢失任何代码。

前面简要介绍了 Visual Studio 2005 集成开发环境的对象浏览器和类设计器，下面简要介绍 C#2.0 的新特性。

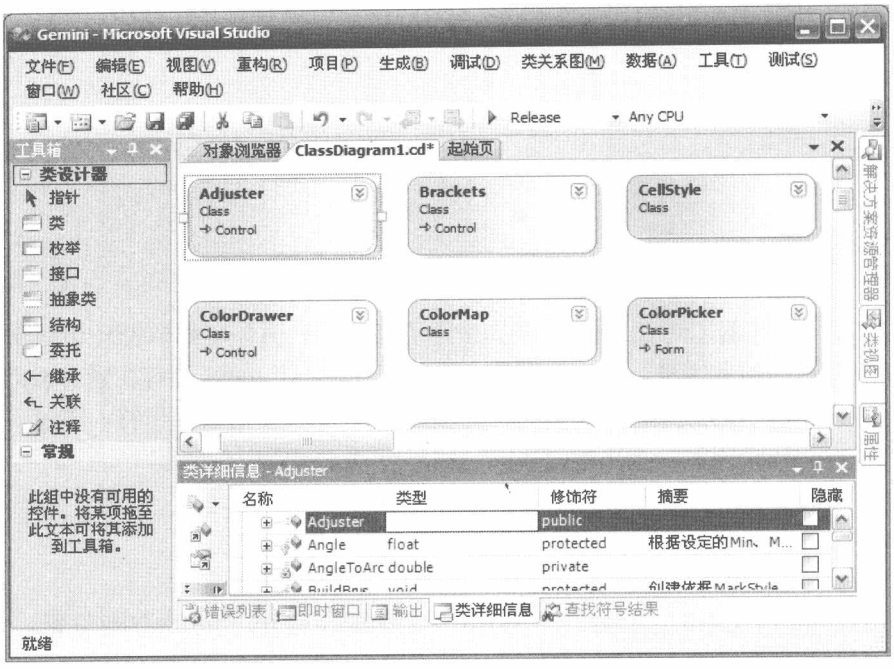


图 1.3 类设计器

（1）泛型。CLR 和 IL 都支持泛型（Generics）。泛型类和泛型方法同时具备可重用性、类型安全和效率，这是非泛型类和非泛型方法不具备的。泛型主要用在非类型特定的通用操作上，例如集合、栈、队列等。使用泛型能够显著提高性能并得到更高质量的代码，因为可以重用数据处理算法，而无须复制类型特定的代码。泛型类型还可以使用 where 子句将参数强制转换为特定的类型。在概念上，泛型类似于 C++ 模板。

下面是一个简单的泛型用法示例。

```
public class GenericClass<T>
{
    T _data = default(T);//初始化_data，如果T是值类型，_data初始化为0，否则初始化
                         为null
    string _name = "";

    public GenericClass(T data)
    {
        _data = data;
    }
    public GenericClass()
        : this(default(T))
    {
    }

    public T Data
    {
```

```
        get { return _data; }
    }
    public string Name
    {
        get { return _name; }
        set { _name = value; }
    }
}
```

（2）迭代器。迭代器（使用 yield 子句）规定了 foreach 循环将如何循环访问集合的内容。为了确保和现存程序的兼容性，yield 并不是一个保留字，并且 yield 只有在紧邻 return 或 break 关键词之前才具有特别的意义，而在其他上下文中，它可以被用做标识符。不过，yield 语句所能出现的地方有几个限制，如下所述。

① yield 语句不能出现在方法体、运算符体和访问器体之外。

② yield 语句不能出现在匿名方法之内。

③ yield 语句不能出现在 try 语句的 finally 语句中。

④ yield return 语句不能出现在包含 catch 子语句的任何 try 语句中任何位置。

下面是一个简单的迭代器用法示例。

```
public class IteratorDemo<T>
{
    T[] _data = default(T[]);

    public IteratorDemo(T[] data)
    {
        _data = data;
    }

    public IEnumerator<T> GetEnumerator()
    {
        for (int i = _data.Length; --i >= 0; )    //从数组尾到数组头遍历元素
        {
            yield return _data[i];
        }
    }
}
```

下面给出一个简单的测试示例，代码如下：

```
void Test()
{
    IteratorDemo<int> iter = new IteratorDemo<int>(new int[] { 1, 3, 2, 6, 7,
    9, 34, 2 });
    string s = "";
    foreach (int p in iter)
```

```
        {
            s += p.ToString() + ";";
        }
    }
```

（3）分部类型。分部类型（partial）定义允许将单个类型（比如某个类）拆分为多个文件。Visual Studio 设计器使用此功能将它生成的代码与用户代码分离。C#编译器在编译的时候会将各个部分的分部类型合并成一个完整的类。对编写程序而言，分部类型不存在特别的差异，只需将各部分当成统一的整体对待即可。分部类型只适用于类、结构或接口，不支持委托或枚举。同一个类型的各个部分必须都有修饰符 partial。使用分部类型时，一个类型的各个部分必须位于相同的名称空间中。一个类型的各个部分必须被同时编译。换言之，C#不支持先编译一个类型的某些部分，然后再编译一个类型的其他部分。

例如，有如下使用分部类型的代码：

```
[Attr1, Attr2("breeze")]
partial class C : IA, IB { }
[Attr3]
partial class C : IC { }
partial class C : IA, IB { }
```

与下面的代码是完全等价的：

```
[Attr1, Attr2("breeze"), Attr3]
class C : IA, IB, IC { }
```

（4）可空类型。可空类型（nullable，采用?来表示）允许变量包含未定义的值。在使用数据库和其他可能包含未含有具体值的元素的数据结构时，可以使用可空类型。可空类型的变量只能是值类型的。实际上，所有的可空类型均是 System.Nullable<T>的一个类型特定的版本。打开对象浏览器就能清楚地看到它的定义：

```
public struct Nullable<T> where T : struct
```

在 Nullable<T>中定义了 T 与 T?之间的隐式转换，例如有如下代码：

```
int? i = null;
int j = 9;
i = 9;              //等价于 i = (int?)9;
j = i;              //编译时错误
j = (int)i;         //正确
i = null;
j = (int)i;         //i 为 null，抛出 InvalidOperationException 异常
int k = i ?? j; //可空属性移除，等价于 int k = i == null ? j : (int)i;
```

可空类型之间的运算和比较稍微复杂，可以总结成一点，即只要有可空类型参与逻辑比较运算，那么逻辑比较的结果都是 false。鉴于篇幅不在此赘述，读者可查阅相关文档。

（5）匿名方法。匿名方法（Anonymous Method）实际上是内联的委托，而委托是一种指

向函数签名的指针。匿名方法是非常有用和强大的，它能减少由于实例化委托和减少分离方法所导致的代码开销。例如它匿名方法可以消除为一段使用不是非常频繁的代码建立函数时对函数命名的烦恼，同时也消除了使用小型函数产生的混乱，使得程序更易于理解和维护。匿名方法可以按如下方式使用：

```
delegate anonymous-method-signature block
```


其中的匿名方法签名是可选的，如下示例演示了匿名方法的使用：

```
delegate void AnonymousMethod1(int x);
AnonymousMethod1 amOne1 = delegate { };                    // 正确
AnonymousMethod1 amOne2 = delegate() { };                  // 错误，签名不匹配
AnonymousMethod1 amOne3 = delegate(long x) { };            // 错误，签名不匹配
AnonymousMethod1 amOne4 = delegate(int x) { };             // 正确
AnonymousMethod1 amOne5 = delegate(int x) { return; };  // 正确
AnonymousMethod1 amOne6 = delegate(int x) { return x; };
                                                           // 错误，返回值不匹配


delegate void AnonymousMethod2(out int x);
AnonymousMethod2 amTwo1=delegate{ };// 错误，AnonymousMethod2 带有一个输出参数
AnonymousMethod2 amTwo2 = delegate(out int x) { x = 1; };   // 正确
AnonymousMethod2 amTwo3 = delegate(ref int x) { x = 1; };   //错误，签名不匹配

delegate int AnonymousMethod3(params int[] a);
AnonymousMethod3 amThree1 = delegate { };                  // 错误，无返回值
AnonymousMethod3 amThree2 = delegate { return; };   // 错误，返回值类型不匹配
AnonymousMethod3 amThree3 = delegate { return 1; };// 正确
AnonymousMethod3 amThree4 = delegate { return "Hello"; };
                                                           // 错误，返回值类型不匹配

AnonymousMethod3 amThree5 = delegate(int[] a)              // 正确
{
    return a[0];
};

AnonymousMethod3 amThree6 = delegate(params int[] a)
                                                           //错误，多余的params修饰符
{
    return a[0];
};
```

（6）命名空间别名限定符。命名空间别名限定符（::）对访问命名空间成员提供了更多控制。global :: 别名将从全局命名空间中搜索类型，因而允许访问可能被代码中的实体隐藏的根命名空间，以避免不同命名空间中的类型名称冲突问题。

```
using Space = System.Windows;
Space::Forms.Form form = new Space::Forms.Form();
```