
学校的理想装备
电子图书·学校专集
校园网上的最佳资源
使用 Visual j++编程
BOOK
内部资料 非卖品

返回总目录

# 第二部分

# 使用 Visual J++编程

# 目　　录

# 第 11 章　WFC 编程基础

Windows Foundation Classes for Java（WFC）是与 Rapid Application Development（RAD）工具配合使用的类库集。它与 RAD 工具的配合使用使得用 Java 语言为 Microsoft 平台建立快速有效的应用程序和组件变得非常容易。WFC 可以用 Java 语言应用在 Win 32 平台和 Dynamic HTML（DHTML）对象模式中，DHTML 是由 Internet Explorer 4.0 支持的一个 W3C 标准。这就意味着可以使用 Java 来解决当前的一些实际问题，同时你已经拥有一条通向未来的坦途。

WFC 的目的是为 Java 提供完整的、吸引人的组件和编程模式。这是因为 Java 有一种在语言方面独特的优势。使用这种新的编程模型来开发 Windows 和 DHTML 的应用程序，Java 应当是非常好的选择。

利用 WFC，开发人员可以使用 Visual J++ Forms Designer 将控件拖放到窗体上、设置属性、生成事件处理程序。开发人员还可以很容易地从应用程序中访问服务器上的数据，并能像 Windows.EXE 文件或 Internet URL 一样展开应用程序。而且，它们可以通过使用设计器或通过直接在编辑器中输入代码的方法来编译自己的 WFC 组件。最后，开发人员可以使用 Java 类访问 DHTML 对象模型。

丰富的应用程序模型巧妙地利用了 J/Direct 技术访问 Win32 API。不管怎样，通过处理一些像 Windows 消息处理过程、消息转储、消息、窗

口句柄等细节，它会使编程更容易。应用程序模型是开放式的，因此，有经验的 Win32 程序员能将 J/Direct 和 WFC 的类结合在一起，在 Win32 平台中安装任何有效的功能。

如果你正找有关使用 Visual J++来开发 WFC 应用程序的快捷方便的介绍说明，那么应该花费一些事件浏览本书第 1 章中的“使用 WFC 创建 Windows 应用程序”一节，它能够一步步地教你建立和运行简单的 WFC 应用程序。

## 使用控件和模板开始

WFC 组件模型中的一个功能就是 Visual J++开发环境提供常用的大部分组件、窗体、菜单和对话框。这些组件通常可以满足需要，或者，用户可以创建自己的控件。因为 Visual J++提供了在 Win32 环境中常用的控件，所以，对于 Visual J++设计环境中可用的可视组件，你会感到非常熟悉。

Visual J++开发环境与 WFC 类紧密集成。因此，大部分使用控件、窗体和菜单的方法都可以在 Using Visual J++中找到。

## 使用窗体初步

在 WFC 编程中，多数应用程序的窗口作为窗体显示。无论是想要一个单独的叠加图形、图形组件（如主应用程序窗口）或对话框，都可以使

用窗体。窗体像一个容纳控件的容器，可以使用户以可视的方式编写应用程序。窗体具有自己的属性，这些属性可以在 Properties 窗口中设定。从语法上来说，窗体是 com.ms.wfc.ui.Form 类中分出的一个 Java 类。Forms 类展开 com.ms.wfc.ui.Control 类，窗体中的控件也是如此。控件封装了一个 Win32 窗口。

当建立新的项目时，并且从 New Project 对话框中选择了 Windows Application 之后，会自动创建一个窗体。用户通过在 Project 菜单中选择 Add Item（添加项）或 Add Form（添加窗体）来添加其他的窗体。

一旦窗体已经在项目中创建，用户可以在设计模式中或在 Text 编辑器中观察它。通过从 Project Explorer 中窗体的快捷菜单中选择 View Designer（查看设计器），可以使用 Forms Designer 来改变窗体的大小和设置窗体的属性。每个窗体都属于一个看不见的，包含应用程序主线程的 Application 对象。窗体和控件体现了可视的 Windows 组件。这种可视窗口和非可视窗口的集成完全由 WFC 结构处理。

当用户在新的窗体上打开 Text 编辑器时，会产生一个基于模板的类，它包含必要的 Form 类语法，包括一个构造器和一个带有实例化窗体代码的 main()方法。当在设计模式下添加控件、设计属性等时，Forms Designer 插入并修改类中相应的部分。参阅本书第 2 章中“创建窗体”一节来了解如何在项目中添加窗体。

## 添加控件

在添加控件时，先将控件从工具箱中拖放到窗体上，调整好大小，然后

在 Properties 窗口中设置属性。在 Visual J++中，ActiveX 和 WFC 控件都可以拖放到窗体上。参阅第 2 章中的“添加控件的窗体”一节来了解如何将控件从工具箱中添加到窗体上。
WFC 控件可分为三类，在使用时这三种控件没有区别：

- Intrinsic controls（内在控件）。基本的 Windows 控件，如按钮、复选框、编辑框和列表框等。
- Common controls（公共控件）。可以在 comctl32.dll 中找到的 Win32 公共控件。这些控件包括动画、工具栏、选项卡、状态栏和 Tree 查看控件等。
- WFC controls（WFC 控件）。特别为 WFC 框架编写的定制控件。

所有现有的 WFC 控件都是可以在 com.ms.wfc.ui 软件包中找到的 Java 类。
如果愿意的话，可以 WFC 软件包创建自己的控件，要么展开已有的控件，或者编写自己的窗体，并将其添加到工具箱中。利用 WFC 组件模型，可以使控件的属性和事件很容易地显示出来。使控件可以与 Visual J++ Forms Designer 紧密合作。

## 添加菜单

菜单可以通过与控件相似的方式来安装，可以像控件一样出现在工具箱中。从工具箱中拖动 MainMenu 控件放到窗体上来添加菜单。但菜单放到窗体上之后，可以通过输入的方式在 Forms Designer 中添加子菜单和

菜单项。每个菜单项具有一个事件处理程序与代码的开发阶段相联系。应用控件，在以可视方式创建菜单时，菜单和菜单选项的所有实现代码都可以自动创建。请参阅本书的第 2 章中“创建窗体菜单”一节，来获得如何将菜单添加到窗体上的更多信息。

## 添加代码

Forms Designer 帮助建立初始的窗体类，并提供许多像事件处理程序这样的骨架代码。但是，有些时候需要编写代码来使程序执行。

当某一事件从窗体上的用户界面元素触发时，例如单击一个控件，便调用事件处理程序。Forms Designer 可以为用户建立骨架代码，这样，通常在事件发生时，只需要填充要运行的代码即可。WFC 组件模型在所有事件处理程序的基础 Visual J++编译器中使用了一种新的 delegate 关键字。当使用 Forms Designer 将事件处理程序挂起时，这些委托是透明的，在更高级的方案中可以直接使用它们，例如源于自己的事件。由于它们基本上与其他语言中的功能指针一样，所以在很多方面它们是有用的。

除了控件和事件处理程序外，WFC 库中其他许多部分也是很有用的。

- Graphics support（图形支持）。在 com.ms.wfc.ui 软件包中的几个类，包括用来提供访问 Windows 图形服务的 Graphics 类。
- Dynamic HTML support（动态 HTML 支持）。com.ms.wfc.html 软件包提供由 Internet Explorer（4.0 或更高版本）访问到 DHTML 对象模

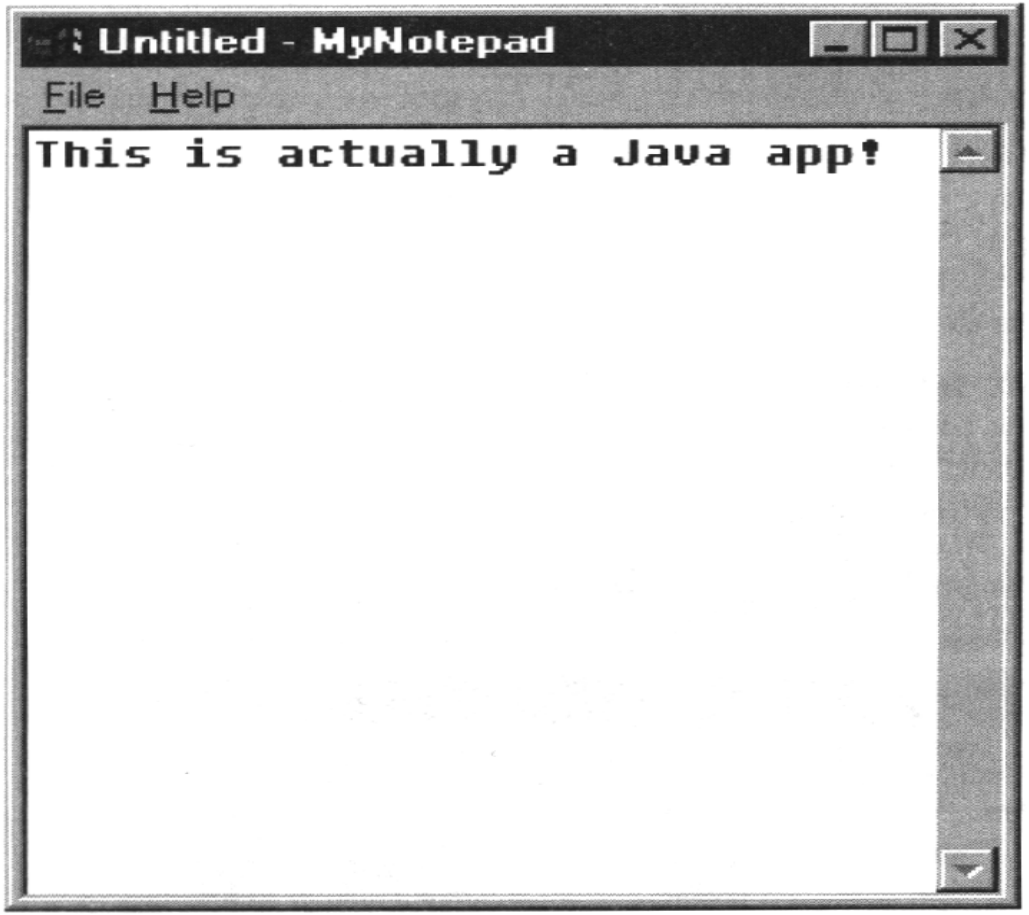
型的工具。

- Data binding support（数据绑定支持）。WFC被设计用来使用ActiveX Data Objects（ADO，ActiveX数据对象）组件来支持简单和复杂的数据绑定。使用DataBinder组件，用户可以从一个记录集中绑定字段到任何WFC组件的属性上。WFC还提供其他的复杂绑定组件用来直接作用于记录集。
- Localization support（本地化支持）。WFC让用户存储资源元素，如字符串、字体和位图到资源文件中，而且，这个资源文件符合特定的本地化特征，使用户可以轻松地进行本地化代码操作。这样就可以简化将用户界面转化为多种语言的操作。
- Direct Win32 API support（直接Win32 API支持）。由Visual J++提供的J/Direct技术允许用户从Java代码中调用任何动态链接库（DLL）。WFC建立在J/Direct对Win32库的调用层上（在com.ms.wfc.Win32和com.ms.wfc.OLE32软件包中的工具）。因此，如果用户习惯了标准的Windows编程，并想要直接访问这些库，WFC提供了基本的元素（如设备上下文和窗口句柄）来给用户提供最终的控件。这些在一些特殊的应用程序中是必须的，大部分程序员会发现WFC的服务是充足的。

## 简单的预演

这一部分简要概括了一个简单的Visual J++应用程序，这个程序以

Windows Notepad 应用程序为基础建立起来，名为 MyNotepad。MyNotepad 是一个 Text 编辑器，它的 File 菜单包括 New、Open、Save、Save As 和 Exit 菜单项。它包括了编辑器的大部分基本功能，如允许用户打开文件、进行编辑，然后将其存储到相同或不同的文件中等。




注意，这些语法与使用 Visual J++ Application Wizard 生成的 Jpad 应用程序很相似。但是，MyNotepad 并不是使用 Application Wizard 建立的，

它是专门用来清楚地演示几个基本概念的。当用户创建了 MyNotepad 之后，由 Application Wizard 生成的代码将更容易理解。因为它们使用了许多共同的原理。

MyNotepad 本质上是由带有编辑控件和菜单组成的单独的窗体（MyNotepad.java）。另一个窗体（NewDialog.java）是一个模态对话框，在用户打开新文件或关闭当前文件时它提示用户进行保存。在本章的最后列出了所有这些文件的完整代码。

该应用程序是利用 Visual J++ Forms Designer 和 Text 编辑器来设计和编码的。许多代码是由设计器自动生成的。在这里，指出了 Forms Designer 的功能、通过设计器创建该应用程序自动生成的代码和 WFC 应用程序编程的几个基本概念。它特别介绍了：

- 使用 Forms Designer 来创建应用程序窗体。
- 如何开始和终止应用程序。
- 剖析了 Visual J++ Forms 模板。
- 如何处理事件。
- 如何打开模态对话框和检索用户的结果。
- 如何使用 OpenFileDialog 和 SaveFileDialog 类来处理文件。
- 如何使用用于文件输入/输出（I/O）的 File 流类。

这个简单的预演结尾附有：

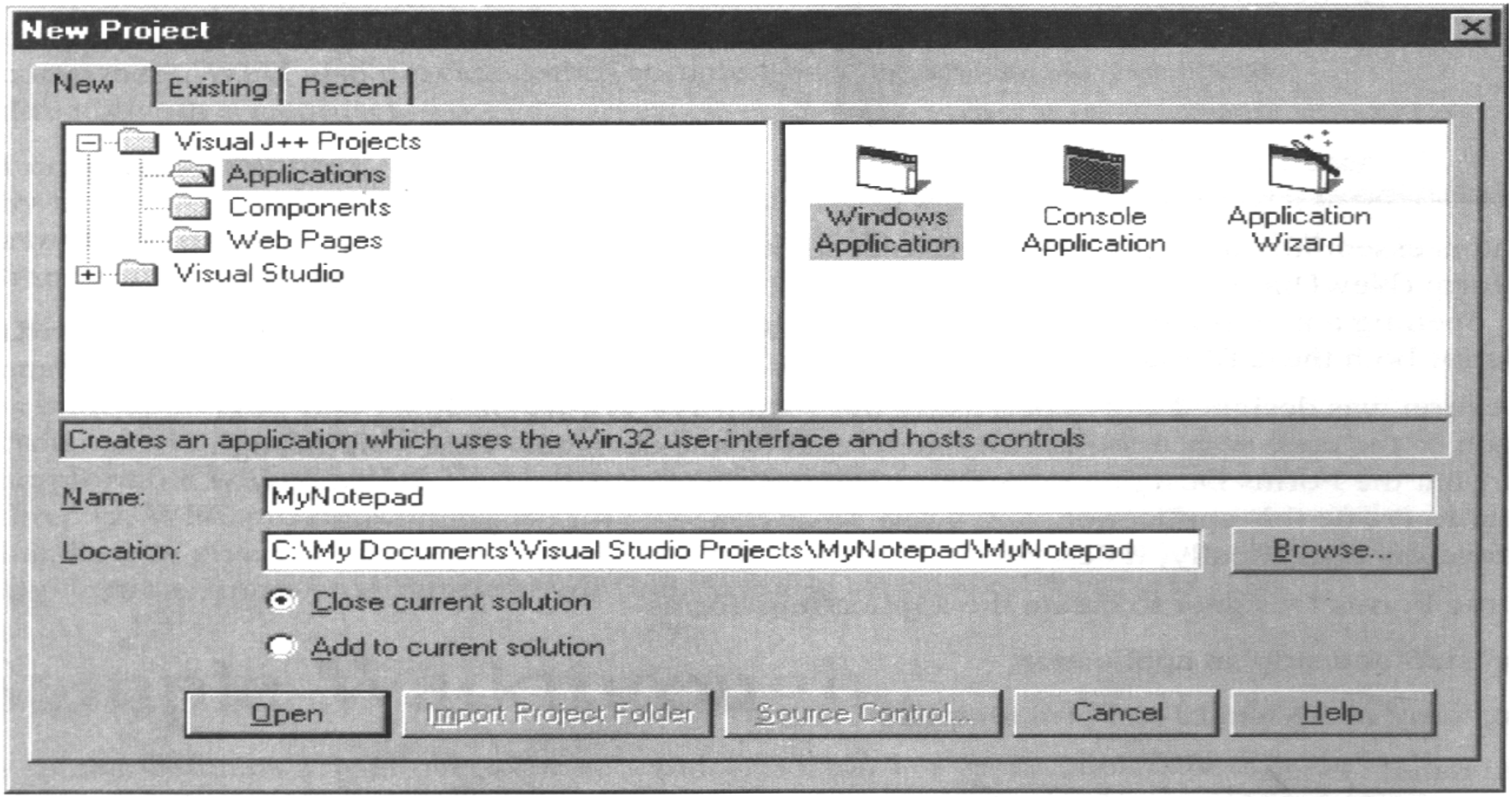
- 简单预演摘要。
- 代码列表。

## 使用Visual J++创建应用程序

这一部分的内容将介绍在 Visual J++中创建 MyNotepad 应用程序的步骤。在进入设计器为这些步骤生成的代码之前，先浏览一下这些步骤是很有好处的。

1. 创建主窗体

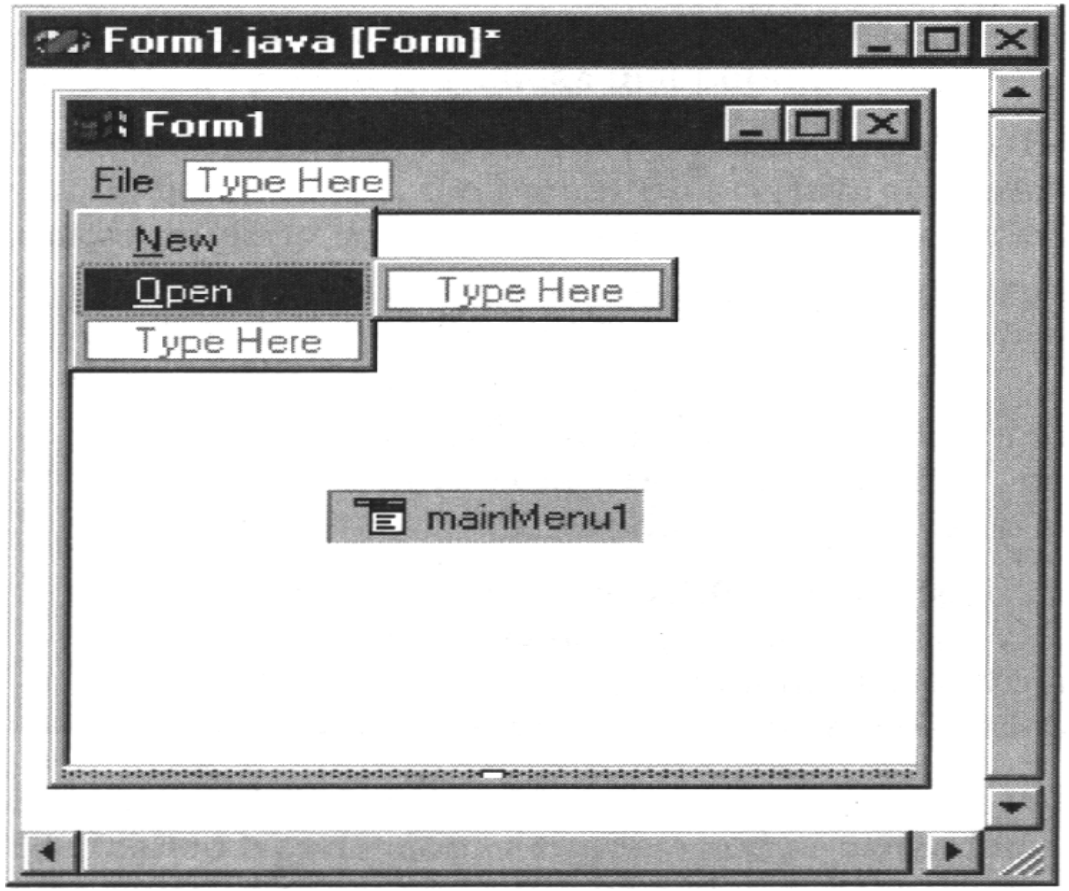
   可以使用第一次打开 Visual J++时出现的 New Project（新项目）对话框，或是从 File 菜单中选择 New Project 命令来创建主窗体。选择 Windows Application 图标，输入应用程序的名字（在这里指 MyNotepad），并且选择 Open，Visual J++使用该名字创建一个项目。该项目默认包含一个名为 Form1.java 的窗体，在后面的步骤中将要把它改名为 MyNotepad。




2. 添加控件和菜单到窗体中

Visual J++ Forms Designer 使得安排窗体内容非常容易。要在设计模式中打开该窗体，在 Project Explorer 中选择 Form1.java，然后从 View 菜单中选择 Designer（设计器）或者从快捷菜单中选择 View Designer（查看设计器）。当窗体显示时，用户可以从工具箱中添加控件。要

访问在工具箱中的 WFC 控件，可以在 Toolbox 选项卡上单击，或者从 View 菜单中选择 Toolbox，然后在工具箱中单击 WFC Control（WFC 控件）按钮来显示这些控件。
对于这个例子来说，一个编辑控件已经从工具箱中添加到窗体上。
添加菜单同样也很容易：从工具箱中拖动 MianMenu 控件到窗体上，并且将它放到一个地方；然后在第一个框中开始输入，并在该框的下面或右边继续添加菜单项。
需要注意的是，可以通过在某个字符前面输入一个“&”符号来在该菜单上创建一个快速访问键。该字符在菜单上变为下划线显示。




3. 在窗体和控件上设置属性

使用 Properties 窗口设置属性。在这种情况下，为了方便起见，许多属性设置为默认值。下面列出的在编辑控件上的属性被改变了：multiline(多行)属性设置为真、doc（文档）属性设置为 Fill（填充）、scrollBars（滚动条）属性设置为 Vertical（垂直）及 font name（字体名）属性设置为 Fixedsys，用来更好地模仿 Notepad。在窗体和控件中，可能还有用户想要设置的其他属性。

用户可能需要重新命名一些组件来使得标题更容易理解。选择窗体或控件，并设置名称属性来重新命名它们。在 MyNotepad 中，更改下面的名称：

| 默认名 | 新名 |
|---|---|
| Edit1 | EditBox |
| MainMenu1 | Menu |
| MenuItem1 | FileMenu |
| MenuItem2 | FileMenuNew |
| MenuItem3 | FileMenuOpen |
| MenuItem4 | FileMenuSave |
| MenuItem5 | FileMenuSaveAs |
| MenuItem6 | FileMenuExit |
| MenuItem7 | HelpMenu |
| MenuItem8 | HelpMenuAbout |

4. 改变 form1.java 的名称

用户可能想要使主窗体的名称不是 Form1.java。可以从 Project Explorer 中选择 Form1.java，右击并且从快捷菜单中选择 Rename（重命名），然后键入新的名称（在这里为 MyNotepad.java），这样，重命名操作就完成了。

如果改变名称，记住，必须将源代码中出现的所有 Form1 改变为 MyNotepad。要这样做，首先应该选择 Forms Designer。然后通过从快捷菜单中选择 View Code（查看代码）来打开源代码文件。从 Edit

菜单中选择 Find and replace（查找和替换）命令，并且，用新名称替换 Form1 的所有实例（例如，将 Form1 替换为 MyNotepad）。

5. 创建对话框

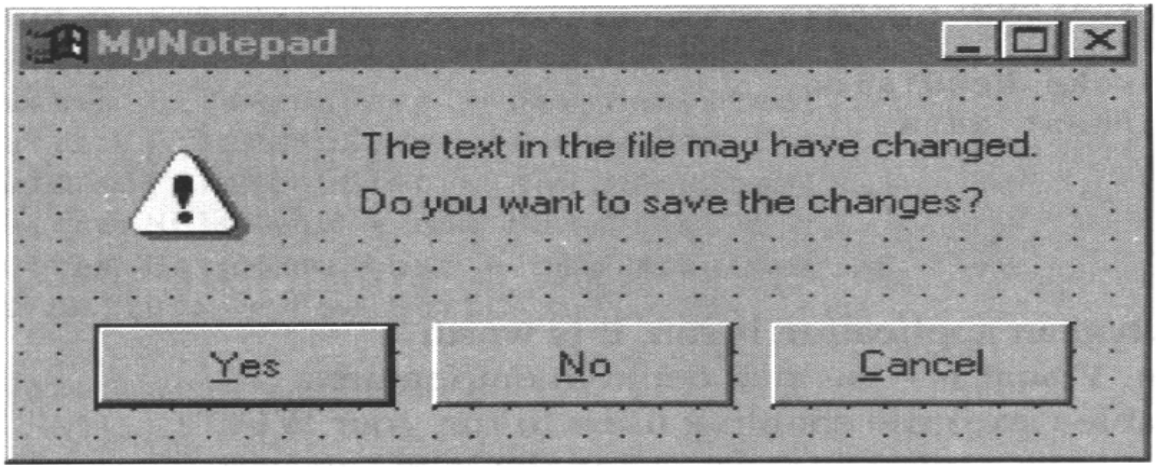
NewDialog 对话框就是在项目中的另外一个窗体。从 Project 菜单中选择 Add Form，在 Add Item 对话框中选择 Form，输入新窗体的名字（在这里，为 NewDialog.java）并单击 Open 来创建另外的窗体。

在这种情况下，添加了三个按钮，它们分别命名为 YesButton，NoButton 和 CancelButton，并具有适当的标签（&Yes，&No 和&Cancel）。按钮控件有一个 dialogResult 属性，在模态对话框中使用该按钮时，这个属性是很有用的。例如，如果 YesButton 控件的 dialogResult 属性设置为 Yes，并且当用户单击该按钮时，对话框关闭，并且返回 DialogResult 值 Yes。在这种情况下，dialogResult 属性设置为下面列出的值：

| 控件 | dialogresult 属性 |
|---|---|
| YesButton | Yes |
| Nobutton | No |
| CancelButton | Cancel |

有一点需要注意，Forms 类有一个 acceptbutton 属性，用来检测当用户按下 ENTER 键时单击了哪一个按钮。在这种情况下，acceptButton 属性设置到 YesButton 控件。Form 类还有一个 cancelButton 属性，用来检测当用户按下 ESC 键时单击了哪一个按钮。在这种情况下，该

属性设置到 CancelButton 控件。
同样，按钮标签中的&符号用来映射每个按钮的特定键（例如，由于 YesButton 按钮的标签是“&Yes”，所以，按 Y 键便相当于单击此按钮）。
随后添加两个附加的标签控件，用来显示该对话框的消息文本。最后，PictureBox 控件添加到窗体中，控件的图像属性设置为一个带有惊叹号的位图。




因为窗体中添加了一个图像，所以，当保存窗体时，Visual J++自动创建一个资源文件（称为 NewDialog.resources），并且将图像放到该文件中。资源文件提供了一种机制来在不同的语言环境中本地化窗体，虽然在这种情况下，它大部分用于进行打包。用户还可以设置窗体的本地化属性为真，以便使资源文件添加到项目中。在这种情况下，包括字符串在内的所有资源都添加到资源文件中。

## 启动和终止应用程序

com.ms.wfc.app 软件包包括静态的 Application 类，用来管理所有的 Win32 窗口处理，如注册、实例化、处理消息循环等等。创建的主应用程序窗口的过程是，调用 Application.run 方法，并将其传递到 Form 派生出的对象中，这个对象构成了窗口的可视化外观。这些调用发生在由 Visual J++生成的基于 Form 模板类的 main()方法中。对于 MyNotepad 应用程序，将生成下面的代码：

```
public static void main(String args[])
{
    Application.run(new MyNotepad());
}
```

默认情况下，由 Visual J++模板创建的基本应用程序使用在应用程序右上角的 Windows 退出按钮（X）来关闭。但是，用户可以在代码中的任何位置退出该应用程序，这只要调用 Application.exit 方法即可。例如，当单击 File 菜单上的 Exit 时，应用程序关闭。

```
Private void FileMenuExit_click(Object sender, Event e)
{
    // Call the new file handler to invoke NewDialog
    // to ask if user wants to save current data
    this.FileMenuNew_click(sender, e);
```

```
    Application.exit()
}
```

上面介绍了应用程序运行时调用的代码，用户如何运行应用程序是值得一提的。Visual J++有丰富的配置特性，其中一个就是用来创建 Windows 可执行文件（.exe），允许用户以与其他 Windows 应用程序类似的风格运行自己的 WFC 应用程序，假定用户的计算机上已经安装了 WFC 类。WFC 类包含在能够重新分布的 Virtual Machine for Java 中。

## Visual J++窗体模板的剖析

基本的 WFC 窗体是一个公共类，使用默认的构造器和 initForm 方法展开 FORM 类。当实例化 Form 类时，类构造器调用 initForm 方法，Forms Designer 将用于初始化窗体和控件属性的所有代码放入 initForm 方法中。应用程序的其他代码在构造器中调用 initForm 的后面。在 MyNotepad 应用程序中，应用程序名就放在这里（虽然它可能已经放在窗体的 Properties 窗口中）。用于 MyNotepad 应用程序的构造器是：

```
public MyNotepad()
{
    // Required for Visual J++ Form Designer support
    initForm();
    this.setBounds(100, 100, 300, 300);
    this.setText("Untitled - MyNotepad");
```

```
}
```

对于所有类主体中添加的控件，Visual J++ Forms Designer 在 initForm 方法前面添加声明。例如，这就构成 MyNotepad.java 窗体对象的声明：

```
/**
  * NOTE: The following code is required by the Visual J++ Forms
  * Designer. It can be modified using the Form editor. Do not
  * modify it using the Text editor.
  * /
Container components = new Container();
MainMenu Menu = new MainMenu();
MenuItem FileMenu = new MenuItem();
MenuItem FileMenuNew = new MenuItem();
MenuItem FileMenuOpen = new MenuItem();
MenuItem FileMenuSave = new MenuItem();
MenuItem FileMenuSaveAs = new MenuItem();
MenuItem FileMenuExit = new MenuItem();
MenuItem HelpMenu = new MenuItem();
MenuItem HelpMenuAbout = new MenuItem();
Edit editbox = new Edit();
private void initForm()
{
```

```
    ...
```

Visual J++ Forms Designer创建此声明代码和在 initForm 方法中的代码，这些代码设置窗体的属性，并且将控件放置到窗体上。处理事件的基础也与 Forms Designer紧密集成，它可以生成在 initForm 方法中映射的事件处理程序。

在 initForm 方法中的头两个语句用来示范 Forms Designer如何设置对象的属性（在这里是将菜单项的 Text 属性设置为“&New”）和使用对象的 AddOnClick 方法来为对象建立单击事件处理程序。

```
private void initForm()
{
    fileMenuNew.setText("&New");
    FileMenuNew.addOnClick(new Eventhandler(this.FileMenuNew_click));
```

## 处理事件

MyNotepad 应用程序中的多数代码出现在事件处理程序方法中，它们在单击菜单项时调用。Forms Designer 通过窗体上控件生成的事件，使得用户可以很容易地创建骨架事件处理程序。例如，要添加一个菜单单击事件的处理程序方法，用户只需在窗体上双击该菜单项（当直接在按钮或菜单项上双击时，会产生单击事件处理程序，也可以使用在 Properties 窗口中的 Events（事件）选项卡来轻松地创建处理程序的骨架代码）。Forms Designer 然后添加骨架事件处理方法，在其中可以添加代码、窗

体的类和插入用于在 initForm 中适当 MenuItem 类的 MenuItem.addOnClick 调用。
例如，单击名为 FileMenuNew 的菜单项时，在 initForm 中添加下面的代码，并且，在类中添加调用 FileMenuNew_click 的方法。

```
FileMenuNew.addOnClick(new EventHandler(this.FileMenuNew_click));
```

MenuItem.addOnClick 方法带有一个 EventHandler 对象。EventHandler 对象是在单击菜单项时使用调用方法的引用创建的。本来，MenuItem 对象监视鼠标单击，并当事件发生时调用每个事件处理程序添加到其中。
所有事件处理对象都是委托；在它们之间，不同的是它们传递到处理程序的事件对象。EventHandler 委托传递一个 Event 对象，该对象包含有关该事件的信息。但是，例如 KeyEventHandler 传递 KeyEvent 对象，该对象扩展键按下和键释放的事件（KeyEvent 包含一个附加的字段指定 UNICODE 字符，以及它是否与 CTRL，SHIFT 或 ALT 键组合使用）。多数 WFC 程序员不需要知道多少有关委托的知识，因为事件处理程序已经存在于 WFC 用于事件的软件包中，并且大部分要添加的代码已经由 Forms Designer 生成。
下面就是在 MyNotepad 应用程序中的事件处理程序，用来控制这些特殊菜单项的选择：

```
private void FileMenuNew_click(Object sender, Event e)
{
   // If edit control contains text, check if it should be saved
```

```
    if (editbox.getText(). length() ! = 0) {
        // Open NewDialog class as a modal dialog
        int result = new NewDialog(). showDialog(this);
        // Retrieve result
        // If Yes button was clicked open Save As dialog box
        if (result == DialogResult.YES)
            this.FileMenuSaveAs_click(sender, e);
        // If No button was clicked clear edit control and set title
        else if (result == DialogResult.NO) {
            editbox.setText("");
            this.setText("Untitled – My Notepad");
          }
    }
}
```

当然，Forms Designer 只创建骨架事件处理程序。打开定制模态对话框的代码还需要手工添加。

## 实现模态对话框

在 MyNotepad 应用程序中，用户单击在 File 菜单上的 New 命令时，在该事件处理程序中的代码检测是否在编辑控件中有文本。如果有，它将打开一个模态对话框，显示询问用户是否保存该文本的消息。如果用户

单击 Yes 按钮，则调用 MyNotepad.FileMenuSaveAs_ click 方法，该方法允许用户选择文件，并且存储当前文本。如果用户单击 No，编辑控件被清除，并且在主窗体上方的标题显示变为“Untitled-MyNotepad”。在 FileMenuNew_click 方法中，该对话框的请求和检索结果是在下面这一行中完成的：

```
int result = new NewDialog().showDialog(this);
```

当打开模态对话框时，对话框结果值从对话窗体中设置。DialogResult 类中包含用于此目的的整型常量，但是可以返回任何一个整数。在这种情况下，使用按钮的 dialogResult 属性，该属性可以设置 DialogResult 值。

作为例子，单击 yesButton 控件设置 DialogResult 为 Yes。当对话框关闭时，该结果从 ShowDialog 方法返回到调用的类中。由 showDialog 返回的整数结果随后用来检测下一步的动作。

NewDialog.java 窗体使用同主应用程序窗体一样的 Form 模板，作为一个新的窗体而创建。注意，主方法不需要模态对话框，并且在本例中删除（将其留下来不会导致错误，但也不是良好的编程风格）。还有，要删除与该方法无关的模板注释。

## 使用消息框作为模态对话框

在简单的情况下，用户还可以使用消息框来代替定制模态对话框。Help 菜单上 About MyNotepad 菜单项的单击事件处理程序使用如下所示的 MessageBox 对象：

```
private void HelpMenuAbout_click(Object sender, Event e)
{
messageBox.show("Version:Visual J++ 6.0", "MyNotepad");
}
```

## 实现文件对话框和文件输入/输出

在这个应用程序中，我们所关心的其余代码有用来操作文件输入/输出和使用 File Open 和 File Save 对话框的代码。它示范了 WFC 类如何简化了定位、打开、读取和写入文件的操作。下面简要介绍 WFC 类如何做这些工作。

在 WFC 类层次中，com.ms.wfc.ui.OpenFileDialog 和 com.ms.wfc.ui.SaveFileDialog 类都扩展 com.ms.wfc.ui.FileDialog。FileDialog 扩展 CommonDialog，它是用于 Win32 公用对话 API 的打包。所有公用对话使用如 setTitle 和 setFilter 这样的属性来设置，并且通过调用 showDialog 方法来运行。这些对话框允许用户为打开或保存的文件选择文件名。

com.ms.wfc.io 软件包包含基于流的输入/输出类。扩展 DataStream 的 File 类包含用于文件输入/输出的方法。在 MyNotepad 应用程序的情况中，需要所有这些来打开文件、将它们都读入到编辑控件中（或者将编辑控件中的内容写到文件中）和关闭文件。

在 MyNotepad 应用程序中，所有的输入/输出和文件对话代码都在 File 菜单上的 Open，Save 和 Save As 的事件处理方法中。我们将只注意其

中的 Open 菜单事件处理程序，因为它封装了公用对话和 File 输入/输出的功能。用于 FileMenuOpen_click 的代码为：

```
private void FileMenuOpen_click( Object sender, Event e)
{
   // Create an Open File dialog box
   OpenfileDialog ofd = new OpenFileDialog()
   // Set up filters and options
   ofd.setDefaultExt("Text Docs (*.txt)|*.txt|All Files (*.*)|*.*");
   ofd.setDefaultExt("txt");
   // Run the Open File dialog box
   int OK = ofd.showDialog();
   // Check result of dialog box after it closes
   if (OK == DialogResult.OK) {
        // Retrieve the filename entered
        fileName = ofd.getFileName();
        // Open a File stream on that filename
        currentDoc = File.open(fileName);
        // Retrieve the length of the file
        int ilength = (int)currentDoc.getLength();
        //Read in ANSI characters to edit buffer
        editbox.setText(currentDoc.readStringCharsAnsi(ilength));
        // Close the file handle
```

```
        currentDoc.close();
        fileOpen=true;
        // Set the application's caption
        this.setText(File.getName(fileName) + " - MyNotepad");
    }
}
```

当单击在 File 菜单上的 Open 命令时，将调用 FileMenuOpen_click 事件处理方法。代码中的前三行创建一个 OpenFileDialog 对象，并且设置对话框使用的过滤器和扩展名。当手工添加这些行时，用户可以使用 Forms Designer 做同样的事情，方法是添加一个 Openfiledialog 对象到窗体，并且设置它的属性（此时，初始化代码放置到 initForm 方法中）。

最后，调用 OpenFileDialog.showDialog 方法来打开对话框。当对话框关闭时，如果用户单击 OK 按钮，方法返回一个等于 DialogResult.OK 的整数，如果用户单击 Cancel 按钮，则返回 DialogResult.Cancel。如果单击了 OK 按钮，则文件名从 OpenFileDialog 对象中接收，并将其传递到 File.open 方法中，该方法返回一个打开的 File 流，这个 File 流是在该文件上以读写访问方式打开的。File.open 是一个实用程序函数，在使用下面的构造器创建 File 对象时，它也可以做同样的事情。

```
File ( fileName , File.OPEN, FileAccess.READWRITE， FileShare.NONE);
```

## 简单预演摘要

这样简短的教程不可能覆盖每个应用程序概念，即使是对 MyNotepad 这样简单的应用程序。但是，事件处理程序、使用对话框操作和简单的文件输入/输出的概念却是多数应用程序所公用的，并且是它们的主要问题。

在这个预演中，我们没有涉及 Save 和 Save As 菜单项事件处理程序的代码。但是，这些代码使用同样的原理，这将在其他地方讨论，所以，当看到列表时，不要感到惊奇。

除了 java.lang 之外，在该应用程序中使用的其他标准 Java 软件包没有太大的用处。诸如 com.ms.wfc.io 和 com.ms.wfc.ui 这样的软件包用来访问 Windows API 的基本功能。当用户知道目标环境将是 Win32 操作系统时，这将提供相当好的性能和可用性。

要记住，MyNotepad 应用程序是作为一个示范工具来编写的。为了要保持它的短小并突出要点，它没有提供更多的错误检查或像 Notepad 和 JPad 那样多的特性。但是，随着这次简要介绍的结束，用户应该知道使用 Visual J++和 WFC 创建自己的应用程序的方法。

## 代码列表

下面列出的代码提供了 MyNotepad 应用程序中的两个基于窗体的类。

# MyNotepad.java

```
/**********************************************************************
MyNotepad.java
This sample is provided as a companion to the Introduction to WFC
programming topic in the Visual J++ documentation.Read the section
titled MyNotepad Sample Walkthrough in conjunction with this sample.
**********************************************************************/

import com.ms.wfc.app.*;
import com.ms.wfc.core.*;
import com.ms.wfc.ui.*;
import com.ms.wfc.io.*;

public class MyNotepad extends Form
{
    private File currentDoc; // the I/O file stream
    private String fileName; //the most recently-used file name
    private boolean fileOpen = false; // set true after file opened

    public MyNotepad()
    {
        // Required for Visual J++ Forms Designer support
        initForm();
```

```
        this.setBounds(100, 100, 300, 300);
        this.setText(“Untitled – MyNotepad”);
}

private void HelpMenuAbout_click(Object sender, Event e)
{
        MessageBox.show("Version: Visual J++ 6.0 " , "MyNotepad");
}

private void FileMenuNew_click(Object sender, Event e)
{
        // If edit control contains text, check if it should be saved
        if(editbox.getText().length() != 0) {
           // Open NewDialog class as a modal dialog
           int result = new newDialog(). showDialog(this);
           // Retrieve result
           // If Yes button was clicked open Save As dialog box
           if (result == DialogResult.YES)
                this.FileMenuSaveAs_click(sender,e);
           // If No button was clicked clear edit control and set title
           else if (result == DialogResult.NO) {
              editbox.setText("");
              this.setText("Untitled – MyNotepad");
```

```
            }
        }
}
private void FileMenuOpen_click(Object sender, Event e)
{
    // Create an Open File dialog box
    OpenFileDialog ofd = new OpenFileDialog();
    // Set up filters and options
    ofd.setFilter("Text Docs (*.txt)|*.txt|All Files (*.*)|*.*");
    ofd.setDefaultExt("txt");
    // Run the Open File dialog box
    int OK = ofd.showDialog();
    // Check result of dialog box after it closes
    if (OK==DialogResult.OK) {
        // Retrieve the filename entered
    fileName = ofd.getFileName();
    // Open a File stream on that filename
    currentDoc =File.open(fileName);
    // Retrieve length of file
    int ilength = (int) currentDoc.getLength();
    // Read in ANSI characters to edit buffer
    editbox.setText(currentDoc.readStringCharsAnsi(ilength));
```

```
        // Close the file handle
        currentDoc.close();
        fileOpen=true;
        // Set the application's caption
        this.setText(File.getName(fileName)+ " – MyNotepad");
    }
}
private void FileMenuSave_click(Object sender, Event e)
{
        // If there has been a file opened or saved
        if (fileOpen){
            // Open the current file again
            currentDoc = File.open(fileName);
            // Write edit control contents to file
            currentDoc.writeStringCharsAnsi(editbox.getText());
            // Close file handle
            currentDoc.close();
        }
            else
                this.FileMenuSaveAs_click(sender, e);
}
```

```
private void FileMenuSaveAs_click(Object sender, Event e)
{
    SaveFileDialog sfd = new SaveFileDialog();
    // Set the options
    sfd.setFileName (fileName);
    sfd.setTitle("Save Text File");
    sfd.setFilter("Text Docs (*.txt)|*.txt|All Files (*.*)|*.*");
    sfd.setDefaultExt("txt");
        // Run the dialog box
        int result = sfd.showDialog();
        if (result == DialogResult.OK) {
            // Retrieve the filename entered in the dialog box
            fileName = sfd.getFileName();
            // Open a File stream with ability to create a file if needed
            currentDoc = new File(fileName, FileMode.OPEN_OR_CREATE);
            // Write the contents of the edit control to the file
            currentDoc.writeStringCharsAnsi(editbox.getText());
            // Close the file handle
            currentDoc.close();
            fileOpen = true;
            // Set the app's caption using the filename minus its path
            this.setText(File.getName(fileName) + " - MyNotepad");
```

```
            }
        }
    private void FileMenuExit_click(Object sender, Event e)
    {
        // Call the new file handler to invoke NewDialog
        // to ask if user wants to save current data
        this.FileMenuNew_click(sender, e);
        Application.exit();
    }
    /**
     * NOTE: The following code is required by the Visual J++ Forms
     * Designer. It can be modified using the Form editor. Do not
     * modify it using the Text editor.
     */
    Container components = new Container();
    MainMenu Menu = new MainMenu();
    MenuItem FileMenu = new MenuItem();
    MenuItem FileMenuNew = new MenuItem();
    MenuItem FileMenuOpen = new MenuItem();
    MenuItem FileMenuSave = new MenuItem();
    MenuItem FileMenuSaveAs = new MenuItem();
```

```
MenuItem FileMenuExit = new MenuItem();
MenuItem HelpMenu = new MenuItem();
MenuItem HelpMenuAbout = new MenuItem();
Edit editbox = new Edit();
private void initForm()
{
    FileMenuNew.setText("&New");
    FileMenuNew.addOnClick(new EventHandler(this.FileMenuNew_click));

    FileMenuOpen.setText("&Open");
    FileMenuOpen.addOnClick(new EventHandler(this.FileMenuOpen click));

    FileMenuSave.setText("&Save");
    FileMenuSave.addOnClick(new EventHandler(this.FileMenuSave_click));

    FileMenuSaveAs.setText("Save &As");
    FileMenuSaveAs.addOnClick(new
        EventHandler(this.FileMenuSaveAs_click));

    FileMenuExit.setText("E&xit");
    FileMenuExit.addOnClick(new EventHandler(this.FileMenuExit_click));

    FileMenu.setMenuItems(new MenuItem[] {
                                FileMenuNew,
                                FileMenuOpen,
```

```
                        FileMenuSave,
                        FileMenuSaveAs,
                        FileMenuExit});
FileMenu.setText("&File");

HelpMenuAbout.setText("&About MyNotepad... ");
HelpMenuAbout.addOnClick(new
    EventHandler(this.HelpMenuAbout_click));

HelpMenu.setMenuItems(new MenuItem[] {
                        HelpMenuAbout});
HelpMenu.setText("&Help");

Menu.setMenuItems(new MenuItem[] {
                    FileMenu,
                    HelpMenu});

this.setText("MyNotepad");
this.setVisible(false);
this.setAutoScaleBaseSize(13);
this.setClientSize(new Point(302,314));
this.setMenu(Menu);
editbox.setDock(ControlDock.FILL);
editbox.setFont(new Font("Fixedsys", 8.0f,
FontSize.POINTS,FontWeight.NORMAL,false,false,false,
```

```
        CharacterSet.DEFAULT, 0 ));
        editbox.setSize(new Point(302,314));
        editbox.setTabIndex(1);
        editbox.setText("");
        editbox.setMultiline(true);
        editbox.setScrollBars(ScrollBars.VERTICAL);

        this.setNewControls(new Control[] {
                                editbox});

         }

        /**
         * The main entry point for the application.
         *
         * @param args Array of parameters passed to the application
         * via the command line.
         */
        public static void main(String args[])
        {
           Application.run(new MyNotepad());
        }
}
```

# NewDialog.java

```
/**********************************************************************
NewDialog.java
This sample is provided as a companion to the Introduction to WFC
Programming topic in the Visual J++ documentation.Read the section
titled MyNotepad Sample Walkthrough in conjunction with this sample.
This form represents a simple modal dialog box.
**********************************************************************/

import com.ms.wfc.app.*;
import com.ms.wfc.core.*;
import com.ms.wfc.ui.*;
public class NewDialog extends Form
{
public NewDialog()
{
    // Required for Visual J++ Forms Designer support
    initForm();
    }
/**
```

```
 * NOTE: The following code is required by the Visual J++ Forms
 * Designer. It can be modified using the Form editor. Do not
 * modify it using the Text editor.
 */
Container components = new Container();
Label label1 = new Label();
Label lable2 = new Label();
Button yesButton = new Button();
Button noButton = new Button();
Button cancelButton = new Button();
PictureBox pictureBox1 = new PictureBox();
private void initForm()
{
    // NOTE: This form is storing resource information in an
    // external file. Do not modify the string parameter to any
    // resources.getObject() function call. For example, do not
    // modify "fool_location" in the following line of code
    // even if the name of the Foo object changes:
    // fool.setLocation((Point)resources.getObject(fool_location"));

    IResourceManager resources = new
            ResourceManager(this, "NewDialog");
```

```
label1.setLocation(new Point(90,20));
label1.setSize(new Point (210,20));
label1.setTabIndex(0);
label1.setTabStop(false);
label1.setText("The text in the file may have changed".);

label2.setLocation(new Point(90,40));
label2.setSize(new Point(190,20));
label2.setTabIndex(1);
label2.setTabStop(false);
label2.setText("Do you want to save the changes? ");

yesButton.setLocation(new Point(20,90));
yesButton.setSize(new Point(80,30));
yesButton.setTabIndex(2);
yesButton.setText("&Yes");
yesButton.setDialogResult(DialogResult.YES);

noButton.setLocation(new Point(110,90));
noButton.setSize(new Point(80,30));
noButton.setTabIndex(3);
noButton.setText("&No");
noButton.setDialogResult(DialogResult.NO);

cancelButton.setLocation(new Point(200,90));
```

```
cancelButton.setSize(new Point(80,30));
cancelButton.setTabIndex(4);
cancelButton.setText("&Cancel");
cancelButton.setDialogResult(DialogResult.CANCEL);

this.setText("MyNotepad");
this.setAcceptButton(yesButton);
this.setAutoScaleBaseSize(13);
this.setCancelButton(cancelButton);
this.setClientSize(new Point(297,136));

pictureBox1.setLocation(new Point(20,20));
pictureBox1.setSize(new Point(50,50));
pictureBox1.setTabIndex(5);
pictureBox1.setTabStop(false);
pictureBox1.setText("");

pictureBox1.setImage((Bitmap)resources.getObject
    ("pictureBox1_image"));

this.setNewControls(new Control[] {
                        pictureBox1,
                        cancelButton,
                        noButton,
                        yesButton,
```

```
                                label2,
                                label1});
        }
}
```

# 第 12 章　WFC 编程概念

Windows Foundation Classes for Java（WFC）提供了一种 Java 软件包的框架。这个软件包支持面向 Windows 操作系统和 Dynamic HTML 对象模型的组件。WFC 与 Visual J++开发环境紧密集成，并提供一整套用 Java 编写的 Windows 控件。使用这种紧密的集成和支持如 IntelliSense，Forms Designer，Application Wizard 和 Object Browser 的特性，建立 Windows 的 Java 应用程序会更加容易。当能够熟练应用这些 Visual J++的特性创建应用程序时，你可能会想了解组成 WFC 软件包和类的结构背后的结构和逻辑。

这一章的目的是为 WFC 软件包和类提供一个概念性的框架，并且解释一些基本的 WFC 模型。很多软件包是组件模型的技术，并且经常被热衷于使用 WFC 控件的开发人员所忽略。其他软件包可以很容易地通过 Visual J++ Forms Designer 访问。当开始研究 WFC 的库时，你就会想知道，对于特定的应用程序来说，哪些软件包和库是重要的。

本章包括下列内容：

- “WFC 软件包”高度概括了组成 WFC 的主软件包。
- “使用 WFC 的可视组件工作”描述了控件、窗体和 WFC 中的图形对象，包括下列内容：

    - Windows 可视组件
    - Dynamic HTML 可视组件
- “在 WFC 中处理事件”描述了处理事件的委托的作用。
- “本地化应用程序”描述了 Visual J++和 WFC 在各种语言中用于本地化应用程序的支持和方法。
- “使用 WFC 应用程序服务”描述了一些主要的应用程序的特点及下列一些内容：
    - 启动和退出应用程序
    - 处理应用程序事件
    - 访问系统信息
    - 执行剪贴板和拖放操作
- “使用 WFC 和 Java 线程”描述了 WFC 的线程模型，并介绍了如下几方面的内容：
    - 混合 Java 和 Win32 的线程模型
    - 创建和退出线程
    - 使用线程存储器
    - 使用线程异常

## WFC软件包

WFC 的基础是 Windows 和 Dynamic HTML。WFC 源于 Win32 Windows 编程模型，它能使用户通过使用 Java 来编译基于 Windows 的应用程序，

这些应用程序利用了 Windows 用户界面的控件、事件和系统服务功能。WFC 也源于 Dynamic HTML 模型，它可以使用户直接从 Java 中利用 Dynamic HTML 的功能来创建客户和服务器的 HTML 页。

这些技术的中心是本机动态链接库（DLLs），它提供了 WFC 基础的核心 API。由于使用了两种不同的技术：JActiveX 工具和 J/Direct，因此这些库对 Java 语言是有效的。如果 DLL 代表了 COM/ActiveX 组件，JActiveX 将创建映射 COM 对象到 Java 对象的打包类。如果 DLL 不是基于 COM 的，J/Direct 直接调用到 DLL,并配置 Java 和 DLL 本机语言（C 或 C++语言）之间的数据类型。这两种技术都利用了 JVC 编译器和 Microsoft Virtual Machine for Java 的内建支持和协作。

了解这些是很重要的，因为几个软件包都是由 COM 打包类（由 JActiveX 产生）或 J/Direct 类组成。这些类具有直接映射基本 API 的方法；在 WFC Reference 中没有说明它们，因为通常它们不能直接访问。但是对于其他软件包，它们是作为支持类来讨论的。

在 WFC 中，不包括本机的 API 支持软件包，有七个主要的软件包：

| 软件包 | 说明 |
| --- | --- |
| com.ms.wfc.app | 封装 Windows 应用程序操作的基类。Visual J++窗体组件模板使用这些类的服务。除了基本的 Windows 信息处理结构外，有适合 Windows 特点的支持，如剪贴板、注册表、线程、窗口句柄和系统信息等 |
| com.ms.wfc.core | 组件模型的基类。这个软件包支持包容器、事件、异常、属性，以及与 Visual J++特征相互操作的基础设 |

| | |
|---|---|
| | 施，如 Forms Designer |
| com.ms.wfc.data | 允许数据访问和数据绑定的 Active Data Object（ADO）Java 类。该软件包还包括 com.ms.wfc.data.ui，该软件包提供了 WFC 中数据绑定控件的基类 |
| com.ms.wfc.html | 用来实现 Java 中的 Dynamic HTML 类，这些类同时提供了客户端和服务器端的支持 |
| com.ms.wfc.io | 用来访问数据流、实现完整的数据包，用于读写串行流，文件访问和在数据流的不同类型之间映射 |
| com.ms.wfc.ui | 与 WFC 一起装载的控件的核心类。这些类还提供了对 Windows Graphics API 的访问 |
| com.ms.wfc.util | 用于各种排序窗体、实现哈希表等的各种窗体的实用程序类 |

下面是 WFC 中核心本地 API 支持类。

| 软件包 | 说明 |
|---|---|
| com.ms.wfc.ax | 提供了适用于 ActiveX 接口的 Java 打包类 |
| com.ms.wfc.html.om | 提供了适用于 Dynamic HTML 对象模型的 Java 打包类 |
| com.ms.wfc.ole32 | 提供了适用于 OLE 服务的 Java 打包类 |
| com.ms.wfc.win32 | 提供了适用于 Win32 API 的 Java 打包类 |

## 使用WFC的可视组件

虽然 WFC 中的 Win32 可视组件和 Dynamic HTML（DHTML）应用程

序之间存在着一些差别，但其间存在着很多相似的地方。这些相似性使得WFC和组件模型能非常容易地服务于上述两种应用程序。比如Win32和DHTML模型都包含基本的控件类型，如编辑框、复选框、按钮、单选框和组合框等。在Win32中最常用的控件中，会有一个DHTML控件以“Dh”开头,而且名字相同（如com.ms.wfc.ui.Edit类有对应的com.ms.wfc.html.DhEdit类）。字体、颜色常量和大多数事件类型也在Win32和DHTML模型中共享。

当然，每一种模型也存在着特殊的组件，如DHTML的表或Win32组件的列表查看控件。

可能WFC Win32和DHTML组件之间最明显的差别就在于DHTML组件在Forms Designer中是无效的。这就意味着用户必须在代码编辑器中创建、添加和修改DHTML元素。但是，在两种方式中最根本的Java代码看起来是非常相似的。

## Windows可视组件

WFC框架来源于它的可视组件：WFC与Visual J++这种可视化开发工具的集成，并位于适用于Windows操作系统的Win32 API的顶端。WFC中支持可视组件的主要软件包是com.ms.wfc.ui，WFC中大部分可视元素的基类是com.ms.wfc.ui.Control。大部分WFC控件扩展这个类，包括Form类，它是控件的可视包容器。

## 控件类

控件类中包含控制 Win32 窗口所有必需的基本属性、方法和逻辑。这些方法可分类如下：

- 用于设置和检索控件的属性，如显示的大小和客户矩形、前景色、背景色、相关的刷子、光标、文本、字体、位置等的方法。这些方法具有以 set 或 get 开头的名字（如 setBrush 和 GetBrush）。
- 事件方法。对于控件产生的每个事件，在控件中可以完成三个方法。例如，对于移动事件，有一个用来触发事件的 onMove 方法、一个将事件处理程序分配给移动事件的 addOnMove 方法和一个删除事件处理程序的 removeOnMove 方法。Control 对象处理大部分的基本控件事件。
- 处理控件父/子关系的方法。如添加方法添加一个子控件，并且 assignParent 和 getParent 分配并检索父控件。还有一些方法用来处理子控件的数组。
- 影响布局、z-次序、绘制和输入控件焦点的方法，如 bringToFront、sendToBack、updateZOrder、PerformLayout、焦点、显示、隐藏、更新、无效、createGraphics 和 createWindowGraphics。
- 低级事件处理方法。在 Win32 级，Windows 从系统中接收消息。在控件类中，每一个输入到控件的消息都有一个保护的进程方法（例如，processCmdKey、processCmdKeyEvent 或 processDialogKey）。如果扩展了基类的控件试图超越它们进行特殊处理，这些方法是非常重要

的。

- 关系到窗口处理、消息和线程启动的方法。这些方法对于编制过 Win32 应用程序的程序员来说是非常有用的。如 createHandle，destroyHandle，getRecreatingHandle，fromHandle，fromChildHandle 都涉及到窗口句柄，sendMessage 和 reflectMessage 允许控件访问到基本的窗口消息。有关线程启动方法的详细内容，见本章后面的“使用 WFC 和 Java”。

控件类扩展 com.ms.wfc.core.Component 类，它是所有 WFC 组件的基类。

## 使用窗体

窗体是应用程序或是与应用程序相关的定制对话框的主要可视元素。com.ms.wfc.ui.Form 类是在 WFC 中提供窗体的基础。

Visual J++ Forms Designer 启动时带有 Visual J++ 窗体模板，它提供类扩展 com.ms.wfc.ui.Form，并且帮助用户在窗体上设置属性，并将控件添加到窗体上。Form 派生类添加窗体中没有找到的 main 方法。当 com.ms.wfc.app.Application.run 方法从主方法中调用，并传递新的基于 Form 的类（代码已经在模板中存在）时，该窗体运行。更多信息见本章后面的“使用 WFC 应用程序服务”中的“启动和退出应用程序”部分。基于 Form 的类作为一个模态对话框来使用，可以通过调用 Form.showDialog 方法来打开（showDialog 也运行基于 com.ms.wfc.ui.Common- Dialog 框类的模态对话框）。基于窗体的非模

态定制对话框可以通过调用窗体的显示方法来打开，该方法使得窗体可见。

Forms 类扩展 com.ms.wfc.ui.Control，所以它具有所有的 Control 方法加上很多自己用来处理任务的方法，就像一个控件的包容器和窗口。

这些包含的方法用来：

- 添加和删除用于激活、关闭、结束、使无效、inputLangChange、inputLangChange- Request、MDIChildActivate、menuComplete、menuStart 和 ownedForm 事件的处理程序。
- 设置窗体的窗口属性，如设置窗体的初始大小或可视状态、开始位置和边界样式；或者用来检测窗体是否有自动滚动条、控制框或最小化框，以及窗体图标是否在任务栏上。
- 确定菜单、控件或其他放在窗体上的窗体的关系，如设置主菜单、安排控件、支持多文档界面（MDI）窗体，或确定窗体是否接收所有的控件键击事件。
- 当窗体作为模态对话框使用时设置属性，并且当窗体在应用程序中时，运行和检索对话框中的结果。它包括的方法用来设置对话框上的 Accept、Cancel 和 Help 按钮、启动对话框和设置，并检索由模态对话框返回的对话结果值。

所列出方法并不全面，但是，可以全面了解什么是窗体的概念。在 com.ms.wfc.ui 软件包中的其他类扩展窗体，并且它是 UserControl 类。UserControl 是用来创建自己复合的基于 Form 的控件，用户可以将其安

装到工具箱中。

## WFC 控件一览

所有可视的 WFC 控件都在 com.ms.wfc.ui 软件包中。在该软件包中有 240 多个类，决定要使用哪个类是很困难的。幸运的是，类划分如下几个主要的分类：

- 在 Visual J++工具箱中并且直接扩展 com.ms.wfc.ui Control 类或基本上源于 Control 类的是控件的类。
- 在 Visual J++工具箱中，并且使用 CommonDialog 作为基类（包括 CommonDialog 类自身）的是控件的类。CommonDialog 打包 Win32 CommonDialog API。
- 包含控件使用的常数值的类。这些类都扩展 com.ms.wfc.core.Enum。
- 表示事件并扩展 com.ms.wfc.core.Event 或是事件处理程序类（委托）的类。
- 表示内置 Windows 图形对象，如刷子、位图、颜色、字体、画笔、调色板、图标、区域和图像等的类。有关如何使用这些对象的消息，见下一部分“访问图形服务”。
- 像 Control 一样，扩展 com.ms.wfc.core.Component，但是不需要 Control 的可视运行时间开销的类。例如 ColumnHeader，Menu（和扩展它的 MainMenu，ContextMenu 和 MenuItem），RebarBand，StatusBarPanel，ToolTip 和 ImageList（和扩展它的 ImageListStreamer）。
- 用来打包其他 Windows 接口的混合类；这些类包括打包 Windows Help

引擎的 Help、打包浏览器的 HTMLControl 和打包 Windows 消息框的 MessageBox 等等。

下面列出的 com.ms.wfc.ui 类直接扩展 Control。在说明中列出了扩展这些类的控件。

| 类 | 说明 |
|---|---|
| Animation | 封装 Windows 动画控件，是用于播放 Audio-Video Interleaved(AVI)动画文件的矩形控件 |
| AxHost | 打包 ActiveX 控件，并且将该控件作为 WFC 控件公布 |
| Button | 封装 Windows 按钮控件 |
| Checkbox | 封装 Windows 复选框控件，该控件是一个用来选择或清除选项的、标识选中和放弃选中的标签框 |
| ComboBox | 封装 Windows 组合框控件 |
| DataTimePicker | 封装 Windows 日期和时间拾取器控件，该控件允许用户指定日期和时间消息 |
| Edit | 封装 Windows 编辑控件，该控件是一个矩形控件，用户可以在其中输入文本 |
| Form | 表示基本的顶层窗口 |
| GroupBox | 封装组框控件，该控件是包含其他控件的矩形框 |
| Label | 封装 Windows 标签控件，该控件显示用户不能编辑的文本串 |
| ListBox | 封装 Windows 列表框控件，该控件显示用户可以在其中 |

| | |
|---|---|
| | 选择一个或多个项的列表。ListItem 在该类中使用。CheckedListBox 控件扩展该类 |
| ListView | 封装 Windows 列表查看控件，该控件显示一个项目集，每个都由一个图标（来自图像列表）和一个标签组成 |
| MDIClient | 表示包含 MDI 子窗口的窗口 |
| MonthCalendar | 封装 Windows 月历控件，该控件提供一个简单的月历界面，在其中用户可以选择日期 |
| Panel | 表示可用来作为其他控件上层的可视的包容器。TabPage 类扩展 Panel |
| PictureBox | 封装用来包含位图的 Windows PictrueBox 控件 |
| ProgressBar | 封装 Windows ProgressBar 控件，该控件通过移动一个条来动态显示操作的进程 |
| RadioButton | 封装 Windows 单选（或选项按钮）控件，该控件显示可以被选择和清除的选项 |
| Rebar | 封装重条（rebar）控件，该控件在可移动、可重新设置大小的边界情况下包含其他控件。该类使用 RebarBand |
| RichEdit | 封装 Windows RichEdit 控件 |
| ScrollBar | 表示用于滚动条控件的基类。HScrollBar 和 VScrollBar 扩展 ScrollBar |
| Splitter | 封装拆分条控件，该控件允许用户在运行时重新设置停靠控件的大小 |
| StatusBar | 封装 Windows 状态栏控件。该类使用 StatusBarPanel |

| | |
|---|---|
| TabBase | 定义包含 Tab 类控件的公用功能的基类。TabControl（使用 TabPage）和 TabStrip（使用 TabItem）扩展该类 |
| ToolBar | 封装 ToolBar 定制控件。该类使用 ToolbarButton |
| TrackBar | 封装 Windows 轨迹条控件（也就是滑块控件），该控件包含一个用来在范围中选择值的滑块 |
| TreeView | 封装 Windows 树形视图控件。该类使用 TreeNode |
| UpDown | 封装上下箭头控件（有些情况下称为微调控件） |

## 访问图形服务

在 WFC 环境中，应用程序通过使用 Graphics 对象执行图形操作，这些对象封装了 Windows 操作系统本身的绘图能力。这些对象对于一般的绘图操作提供了灵活的支持，包括显示图像和图标，还有画线、画多边形及输入文本等。

图形对象通过打包 Windows 设备上下文来执行它的工作，Windows 设备上下文是定义系统图形对象、关联的属性以及影响设备输出的图形模式的系统数据结构。因为用户可以检索作为 Graphics 对象基础的设备上下文，所以可以将 Win32 绘图程序与 Graphics 对象合作使用。

所有扩展 control 对象的 WFC 对象都支持通过 createGraphics 方法创建 Graphics 对象。另外，所有扩展 Image 对象的对象，如 Bitmap、Icon 和 Metafile，都支持通过 getGraphics 方法创建和检索它们关联的 Graphics 对象。

有关如何使用对象的消息，见本书第 15 章。

## 动态HTML可视组件

Dynamic HTML 元素构成在 WFC 中第二个可视组件集合。在 com.ms.wfc.html 中的该控件基于 Dynamic HTML 对象模型。在 com.ms.wfc.html 中的类用来创建新的元素，并且还用来绑定到在 HTML 页上已有的元素。这些组件可以在客户的浏览器或是服务器上创建和操作，服务器将它们发送到客户浏览器。该对象模型存在于几个平台上。因此，尽管用户界面看起来有些相似，这是因为按钮、列表框、单选框等等标准设置在两端都预先发送了，但从根本上来说，该对象模型并非源于 Win32。

WFC 控件的这两种设置（Win32 和 Dynamic HTML）有一些相似，因为它们最终都是源于 com.ms.wfc.core.Component。组件是可放于包容器中的元素，并且支持 Icomponent 接口，该接口具有存放组件的方法。对于使用 WFC 的多数编程人员，并不怎么关心组件与包容器是如何关联的；但是，因为 com.ms.wfc.html 和 com.ms.wfc.ui 的元素都以组件为基础，它们的特性也类似。例如，使用 add 方法将所有组件添加到它们的父包容器中。

要更好地理解如何使用 com.ms.wfc.html 软件包，见本书第 14 章。

## 处理WFC事件

Control 基类和扩展它的类，如按钮和编辑框，会遇到如 click、keyPress、

mouseMove、DragDrop 和其他标准的 Windows 事件。用户可使用委托在应用程序中操纵事件。用户不需要十分清楚地理解委托，就可以在应用程序中编写事件处理程序。但是，如果正在创建控件、使用其他触发事件的应用程序，或者使用带有 WFC 组件的线程时，理解如何创建和使用委托是非常有用的。如果想要理解通过 Forms Designer 创建 Java 代码的细节，它也是很有用的。这一部分提供了一些有关委托的背景资料，然后介绍处理事件的实际情况。

什么是委托？委托声明定义一个扩展 com.ms.lang.Delegate 的类。JVC 编译器也将委托（delegate）作为一个关键字来识别，提供创建基于委托的类的快捷方式。委托实例可以调用有关对象的方法，并且传递数据到该方法。更重要的是，委托与它引用的对象隔离开来，并且不需要知道有关它的任何事情。那么，对于“匿名调用”来说，它是很理想的。在其他语言中，这种功能是作为函数指针来实现的，但是与函数指针不同的是，委托是面向对象的、类型安全的和可靠的。

在 WFC 中，委托经常用于将事件绑定到处理程序方法，如将按钮控件上的单击事件绑定到类的处理程序方法。但事件发生时，控件调用委托，传递给它一些事件消息。委托依次调用已注册的处理程序方法，并且将事件数据传递给它们。用户还可以使用委托来将一个事件绑定到多个方法上（叫做 multicasting）；当事件发生时，列表中的每个委托按着它们添加的顺序被调用。反过来说，不同事件的委托也可以分配给同一个处理程序方法（例如，工具栏按钮和菜单项都可以调用同一个处理程序）。

要使用在应用程序中的事件，当用于某个特殊控件的事件发生时，用户使用委托来为通知进行注册。要进行注册，应该调用控件的AddOn<event>方法，这里的<event>是所要处理的事件名称。例如，要为一个按钮的单击事件注册，应该调用该按钮对象的addOnClick方法。addOn<event>方法用一个委托的实例作为参数，通常，这是与特定事件数据相关联的现有WFC委托。在addOn<event>调用中，委托实例由方法的引用来创建，该方法就是要绑定事件的方法。下面的例子显示了如何将事件处理程序“btnOK_Click”（在当前类中）绑定到名为btnOK的按钮单击事件中。

```
Button btnOK = new Button();
btnOK.addOnMouseClick( new EventHandler ( this.btnOK_Click));
```

对于多数事件来说，用户可以创建并传递一个普通的EventHandler委托的实例，该委托将传递一个普通的event对象。但是，当某些事件包括附加的、事件指定的消息时，它们使用特殊的事件处理程序类。例如，典型的鼠标移动事件包括鼠标光标位置。要得到这种消息类型，用户创建一个MouseEventHandler类的实例，该类传递一个MouseEvent对象到处理程序。键盘事件需要KeyEventHandler来得到有关SHIFT键状态的消息等等（该处理程序传递一个KeyEvent对象）。
所有的WFC处理程序委托类扩展com.ms.lang.Delegate。它们中的大多数在com.ms.wfc.ui软件包中，并且以EventHandler字符结尾。所有的WFC事件扩展com.ms.wfc.core.Event，以Event字符结尾，并且可以在

com.ms.wfc.ui 软件包中找到它们。

> **提示：** 在 Forms Designer 中，用户可以使用 Properties 窗口中的 Events 视图来绑定事件到指定的方法上。Forms Designer 然后为用户创建适当的 addOn<event>方法和骨架处理程序。

当委托调用处理程序时，它传递两个参数。第一个参数是引起该事件的对象引用。第二个参数是包含该事件信息的事件对象。在前面的例子中，该委托的处理程序看起来应该如下所示：

```
private void btnOK_Click( Object source, Event e) {
    if (Source instanceof button) {
       String buttonName = ((button)source).getText();
       MessageBox.show("You clicked button" + buttonName;
    }
}
```

如果使用常规 EventHandler 类来绑定到方法中，应用程序中的 Event 对象将不包含任何重要的消息。但是，事件可用额外的信息，用户可以从具体的事件对象中将其抽取出来。下面列出的是用于鼠标移动事件的委托和处理程序的代码。MouseEvent 对象公布了允许用户得到鼠标位置的属性。

```
// This is the request for notification
Button btnTest = new Button();
```

```
// Note that the addOn<event>method uses the MouseEventHandler class
btnTest.addOnMouseMove( new MouseEventHandler(this.btnTestMouseMove));

// This is the handler for the mouse movement event
private void btnTestMouseMove(Object source, MouseEvent e) {
    edit1.setText( e.x + , + e.y);
}
```

如果想要为用于同样控件的多控件或多事件来处理事件，需要为每一个控件/事件的组合请求一个单独的通知。多个通知可以指定同一处理程序；例如，工具栏上的所有按钮为它们的单击事件可能调用同一个处理程序。用户可以使用源对象传递到事件处理程序，来得到单击了哪个按钮的详细情况（通常，用户将传递到处理程序中的对象放入适当的类中，以便能够从适当的类中调用方法）。
下面的例子显示了为工具栏定义按钮，为它们的单击事件请求通知，以及处理方法的代码：

```
private void initEventHandlers() {
    Button buttonNew = new Button();
    Button buttonSave = new Button();
    Button buttonExit = new Button();
    // All events are routed to the same handler
    buttonNew.addOnClick(new MouseEventHandler( this.toolbarClick) );
    buttonSave.addOnClick( new MouseEventHandler( this.toolbarClick) );
```

```
        buttonExit.addOnClick( new MouseEventHandler( this.toolbarClick);
    }
    // common event handler
    private void toolbarClick( Object source, Event e){
        String buttonName;
        if (source instanceof Button) {
            buttonname = new String((Button) source).getText());
            MessageBox.show("You clicked button" + buttonName);
        }
    }
```

## 本机化应用程序

WFC 和 Visual J++ Designer 为开发使用多种语言的应用程序提供了简便的方法。WFC 应用程序能够以几种本地化语言版本创建，版本之间的唯一不同是一个二进制资源文件。每个资源文件的命名约定都指出它所支持的语言，并且，根据用户的本地化设置，可以加载正确的资源。理解本地化概念有两部分：设计时实现和运行时支持。Visual J++要进行本地化，就要识别可视元素（窗体和控件）的某些属性。在设计时，Visual J++用来将这些本地化属性放入一个二进制资源文件中。例如，控件的文本、字体和大小可以在两种语言版本中改变。在运行时，当加载应用程序时，系统在确定用户的位置时，加载与客户线程对应的资源。

要创建应用程序的本地化版本，使用 Visual J++来设计可视布局，并将窗体的本地化属性设置为真，然后保存该窗体。Visual J++自动创建一个二进制资源文件，并将所有的本地化属性与它关联起来。

当窗体的本地化属性设置为真时，Visual J++总是将资源保存到名为 form.resources 的单一资源文件中，这里的 form 是主窗体的名称（例如，Form1.resources）。所创建的每个版本都将保存到该资源文件名。当创建每个新版本之后，用户使用 Windows Explorer 或 MS-DOS 命令将该资源文件复制一份，并且将其名称重命名为带有标准 Windows 本地后缀的适当的本地语言名（例如，对于日本版本来说，是 Fomr1_jpn_jpn.resources）。第一个本地后缀指定了主要的语言，第二个后缀指定了次要的语言。

> **重点：**在布局并保存任何本地化版本之前，一定要使用新名保存原始布局。这将是用户的主 .resources 文件。

例如，假定用户想要为主窗体名为 Zippo.java 的应用程序创建美国英语、法语和日语版本。还假定用户使用英语版本开始（尽管这不是必须的）。首先，按英语布局窗体，并将本地化属性设置为真。当保存该窗体时，便创建了文件 Zippo.resource。现在使用 Windows Explorer 或 MD-DOS 命令复制 Zippo.resource，并将复制的文件重命名为 Zippo_enu_enu.resources（enu 是用于美国英语的本地后缀，它是按照主要的和次要的语言指定的）。

接下来，在 Visual J++ Designer 中，改变默认的语言为法语，并且在法

语中使用属性本地化来布局控件。当结束时，再次保存窗体，覆盖zippo.resouces的以前版本。再次复制Zippo.resources，并且将其改名为Zippo_fra_fra.resources。

要测试这些版本，在Control Panel（控制面板）中的Regional Settings（区域设置）对话框中，此时选择区域为French(Standard)（不需要重新启动计算机，区域将在本地线程上改变）。

## 使用WFC应用程序服务

com.ms.wfc.app软件包包含了许多提供WFC应用程序服务的类。这些操作的大多数属于Application对象自身。这些操作主要用来创建线程、启动应用程序、处理应用程序事件等等。因为理解Java线程是很重要的，所以，在本章后面的“用WFC使用Java线程”一节中将专门讨论这一点。

作为应用程序服务限定的其他操作与Win32操作系统提供的操作相关。这些操作包括在其他操作之间访问Windows注册表、访问剪贴板数据和检索系统消息等。

## 启动和退出应用程序

Application.run方法启动WFC应用程序。它通常放在基于窗体的类的主方法中，该方法用来构造主应用程序窗体。Application.run具有不带

参数或带有一个参数的重载方法，这一个参数指定表示应用程序主窗口的窗体类。例如，下面就是该调用的一个典型窗体：

```
public static void main(String args[])
{
    Application.run(new MyMainWindow());
}
```

如果窗体传递到运行方法中，窗体的可视属性自动设置为真，并且onClosed事件处理程序添加到该窗体中。当窗体关闭时，onClosed事件处理程序调用 application.exitThread 方法。如果没有传递窗体，则应用程序会一直运行，直到调用 Application.exit，关闭应用程序上的所有线程和窗口，或运行到调用 exitThread，只关闭应用程序的当前线程。

## 处理应用程序事件

用户使用 Application 对象为 5 个不同的事件指定事件处理程序，这 5 个事件在应用程序的上下文中发生，它们是：applicationExit、idle、settingChange、systemShutdown 和 threadException。可以调用下面的addOn方法来为这些事件定义事件处理程序：

| 应用程序方法 | 说明 |
| --- | --- |
| addOnApplicationExit | 指定当应用程序退出时调用的处理程序。用户可以在这里清除不会由垃圾收集释放的应用程序资源（要 |

| | |
|---|---|
| | 强制应用程序不退出，需为窗体的关闭事件指定一个处理程序） |
| addOnIdle | 指定当应用程序的消息队列空闲时调用的处理程序。例如，执行后台操作或应用程序清除 |
| addOnSettingchange | 指定当用户改变窗口设置时调用的处理程序 |
| addOnSystemShutdown | 指定用户启动系统关闭之前立即调用的处理程序。这提供了一个保存数据的时机 |
| addOnThreadException | 指定当已经废弃未捕获的 Java 异常时调用的处理程序，允许应用程序完美地处理该异常。该事件处理程序获取 com.ms.wfc.app.ThreadExceptionEvent 对象，该对象中有一个用来表示废弃该异常的字段 |

所有这些“addOn”方法都有一个相对的“removeOn”方法，用来删除该事件处理程序。

## 访问系统信息

Win32 系统包含了大量的可以被 WFC 应用程序和组件访问的信息。这些访问多数是通过在 com.ms.wfc.app 软件包中的类进行的。这些消息多数存储在 Windows 的注册表中，并且通过 RegistryKey 和 Registry 类来访问。其他系统消息，如 Windows 显示元素的大小、操作系统设置、网络可用性和硬件性能等，使用 com.ms.wfc.app.SystemInfomation 类中的静态方法来访问。使用 com.ms.wfc.app.Time 类可以利用系统时间。

这一部分概括了 WFC 应用程序访问系统信息的方式。

## Windows注册表信息

com.ms.wfc.app 软件包中的 RegistryKey 类包含访问 Windows 系统注册表的方法。使用在该类中的方法来创建并删除子键、获得当前键的子键数量和名称以及检索、设置和删除分配到子键的值。

com.ms.wfc.app.Registry 类包含容纳表示注册表根键的 RegistryKey 对象的字段（以 HKEY_起始）。根 RegistryKey 对象还可以使用 getBaseKey 方法例示。方法可以在任何 RegistryKey 对象上调用，用来列举和操作在根对象下面的子键树中的键和键值。例如，下面的代码得到一个在 HKEY_CURRENT_USER 键下的 subKey 名数组和数组中的名称号。

```
int subKeyCount;
String[] subKeyNames;
SubKeyNames = Registry.CURRENT_USER.getSubKeyNames();
SubKeyCount = Registry.CURRENT_USER.getSubKeyCount();
```

同样，可以检索任何子键或设置给予它的路径和子键值名，并且，可以检索数据或设置给予它的值名。下面的例子显示了检索 Visual Studio 中最近使用的文件名，并在一个编辑框中显示它们。

```
String path; // Holds the path name.
String[] valueNames; // Holds array of MRU file names in the key.
```

```
int valueCount; // The number of MRU file names in valueNames.
path = new String(″Software\\Microsoft\\VisualStudio\\6.0\\FileMRUList″);
RegistryKey subKey = registry.CURRENT_USER.getSubKey( path);
// Get the file names and the number of file names.
valueNames = subKey.getValueNames();
valueCount = subKey.getValueCount();
if (valueCount > 0)
    for (int I = 0; I < valueCount; ++i){
        // Get the value, which is the actual file name.
        String value = new String((String)subKey.getValue(valueNames[i]));
        // Concatenate the name("1", "2", etc.) with the file name value.
        String valString = new String(valueNames[i] + "    " + value);
        // Add this to the edit box.
        edit1.setText(edit1.getText()+valString + "\r\n");
  }
```

用户还可以使用 createSubKey 方法创建新键，并使用 SetValue 方法在这些键中设置值。

## 位置信息

位置信息提供了在用户的计算机上设置的语言和区域设置的详细资料。系统中存储有许多有关语言和区域的特性。包括字符集、国际电话代码、

货币信息如何显示、使用哪种日历、度量系统等等。
这些信息一般使用在 Control Panel 中的 Regional Settings 对话框来设置，但也可以利用程序来设置。在 WFC 中提供这种访问的有：com.ms.wfc.app.Locale 类中的方法，以及与 Locale 方法相关的包含字段常量的许多 Local 子类。有关设置和检索这些信息的细节，见 Locale 类中的方法。

## 时间信息

时间是另外的一种系统信息。com.ms.wfc.app.Time 类提供了一个具有许多功能的 Time 对象，包括捕获系统时间：默认的构造器使用系统日期和时间创建一个 Time 对象。除了检索系统时间之外，Time 对象可以做许多其他事情，如比较 Date 和 Time 对象，将时间转换为各种格式，以及为以后的检索来存储 Time 对象等。
Time 对象一旦创建就不能改变。但是，Time 类提供了许多方法，以使用偏移时间来创建新对象（如 AddSeconds，addMinutes，addHours，addDays 和 addYears）。还有许多用来检索某种 Time 对象属性的方法，如秒、分钟、小时和天等等。
在 WFC 中，Time 对象从公元 100 年 1 月 1 日开始，以千万分之一秒为单位来存储时间。可以存储的最大值为公元 10000 年 12 月 31 日。转换 WFC Time 对象到其他格式（如 Strings，Variants，SYSTEMTIME 等）可能导致精度的丢失，并且不是所有的格式都可以存储这么大范围的值。

不要将 Time 类与其他调用 Timer 的 com.ms.wfc.app 类混为一谈。Timer 实际上是一个控件，它不在 com.ms.wfc.ui 软件包中的原因是它没有用户界面。

## 执行剪贴板和拖放操作

在 WFC 中，拖放操作是基于 Win32（OLE）模型的，这种模型实现用于复制或粘贴数据的快捷方式。当使用剪贴板时，必须执行几个步骤：选择数据，从上下文菜单中选择 Cut 或 Copy，移动到目标文件、窗口或应用程序中，然后从上下文菜单中选择 Paste（数据的来源称为源，而目的地称为目标）。

拖放特性不使用上下文菜单。它按下鼠标左键来捕获在源中选中的数据，然后在目标中放开鼠标按钮将其放入。拖放操作可以传输任何能放置到剪贴板中的数据；因此，用于拖放数据的格式与它们放入到剪贴板时的数据格式相同。例如，数据格式指定这些数据是否为文本、位图、HTML、.wav 等等。com.ms.wfc.app.DataFormats 类包含适合于每个剪贴板格式的字段。这些字段名（如 CF_TEXT）直接来源于 Win32 常量名。

剪贴板或拖放操作的数据存储在 com.ms.wfc.app 中名为 DataObject 的类中，该类实现 IDataObject 接口。IDataObjece 为设置和检索数据、在数据对象中得到数据格式的列表和查询已有的特殊数据格式定义方法。要使程序实现将数据放入剪贴板或从剪贴板中检索数据的操作，需使用 com.ms.wfc.app.Clipboard 中的静态方法。Clipboard.setDataObject 从剪

贴板中返回 IDataObject。目标位置必须确保可以使用在剪贴板中数据的格式。要做到这一点，它应该使用 IDataObject.getDataPresent 方法查询数据对象，传递可以接受的数据格式；如果数据类型可以接受，则 getDataPresent 返回真。

## 实现放入源

对于任何 WFC 控件组件（基于 com.ms.wfc.ui.Control），调用 Control.doDragDrop 方法来开始该操作。这些是响应用户按下左键移动鼠标的典型操作。因此，代码放置到 mouseMove 事件处理程序中，该程序检查 MouseEvent 对象查看鼠标左键是否被按下，指示拖动操作的开始。例如，下面是一个用于包含文件名的列表框控件的事件处理程序：

```
private void listFiles_mouseMove(Object source, MouseEvent e)
{
   //if the left button is down, do the drag/drop
   if(this.getMouseButtons()==MouseButton.LEFT)
{
      String data = (string)listFiles.getSelectedItem();
      listFiles.doDragDrop( data, DragDropEffect.ALL);
   }
}
```

doDragDrop 方法带有要转换的数据和 com.ms.wfc.ui.DragDropEffect 对象。对于拖放操作的模式，DragDropEffect 类包含下面使用位运算符 OR

组合的常量。

| DragDropEffect 方法 | 说明 |
| --- | --- |
| COPY | 指定在传递之后数据将不从源中删除 |
| MOVE | 指定在传递之后数据将从源中删除 |
| SCROLL | 指定在传递之后数据在目标中滚动 |
| ALL | 指定在传递之后数据将从源中删除，并且在目标中滚动（从本质上来说，是 COPY\|MOVE\|SCROLL） |
| NONE | 指定没有执行操作 |

在拖放操作中接收数据的目标检索包含 DragDropEffect 对象的 dragDrop 事件，所以它可以很容易地确定该操作的目的。

## 实现放入目标

拖放操作的放入部分作为事件处理。Control 类为拖放事件（dragDrop、dragEnter、dragLeave 和 dragOver）提供事件处理程序的基础。可以使用下面的方法来指定这些事件的处理程序。

| Control 方法 | 说明 |
| --- | --- |
| addOnDragDrop | 为拖入控件或窗口的数据指定一个处理程序（当鼠标左键放开时） |
| addOnDragEnter | 当光标第一次进入窗口时为放入数据指定一个处理程序 |
| addOnDragLeave | 当光标离开窗口时为放入数据指定一个处理程序 |
| addOnDragOver | 当光标拖过窗口时为放入数据指定一个处理程序 |

所有这些“addOn”方法都有与之相对的“removeOn”方法，用来删除该事件处理程序。当在 WFC 中使用所有的事件处理程序 addOn 和 removeOn 时，这些方法获取一个用处理程序方法的名称创建的委托（在此情况中，为 DragEventHandler）。例如，下面的一行添加 txtFile_dragDrop 方法作为一个 dragDrop 事件处理程序：

```
txtFile.addOnDragDrop( new DragEventHandler(this.txtFile_dragDrop));
```

对于所有的拖放事件，dragDrop 事件是最常处理的。不管处理这些事件中的哪个，处理程序中的代码必须至少做三件事。首先，它必须确定是否可以接收这个数据格式，如果可以，它必须复制数据，并且可以选择显示它（或者提供一些用户界面的反馈，表示数据已经放入）。
所有数据和信息都传递到 DragEvent 事件中。这包含了一个数据字段和一个效果字段。DragEvent.data 字段包含一个支持 IDataObject 的对象，该对象中有用来检索数据和数据格式及用来查询特殊格式存在的方法。因此，处理程序必须首先以它将接受的格式调用 DragEvent.data.getDataPresent 方法，然后确定是否它带有数据类型。如果有，它可以随后调用 DragEvent.data.getData 方法（传递到该数据类型）并且检索数据。数据如何显示取决于该控件。下面的例子说明了一个编辑控件 dragDrop 事件处理程序，来显示放入其中的文本数据。

```
Private void txtFile_dragDrop(Object source, DragEvent e)
{
   // If text is in the object, write it into the edit control.
```

```
    if (e.data.getDataPresent(DataFormats.CF_TEXT))
    {
        String filename=(String)e.data.getData(DataFormats.CF_TEXT);
        txtFile.setText(filename);
    }
}
```

## 使用具有WFC的Java线程

Java是一个自由线程的环境。这也就意味着任何对象在任何时间及任何线程中都可以调用任何其他对象。在编写对象时一定要注意，这些方法是不可分的（atomic），而且是线程安全的。
有几种类从自由线程中可以得到好处，并且WFC提供了锁定代码来使得这些对象具有线程安全性。这些类如下所示：

- com.ms.wfc.core.Component
- com.ms.wfc.core.Container
- com.ms.wfc.ComponentManager
- com.wfc.ui.Brush
- com.wfc.ui.Font
- com.wfc.ui.Pen

另一方面，任何源于com.ms.wfc.ui.Control的对象都是公寓式线程

（apartment-threaded），因为 Win32 窗口依赖于每个控件。加之在 com.ms.wfc.ui 软件包中的多数控件是非同步的，所以，它们也可以视为公寓式线程。同样，com.ms.wfc.io，com.ms.wfc.html 和 com.ms.wfc.util 软件包也不是线程安全的。

## 混合 Java和Win32线程模型

WFC 从 Java 对象中访问本机 Win32 结构（如窗口）。Win32 窗口管理器是公寓式线程，并且 Windows 根据需要自动从一个线程到另外一个线程进行调度。当一个自由线程对象调用到一个公寓式线程对象中时，该调用必须调度到该对象的公寓中。这就是说，当公寓式线程处理该请求时，该自由线程在一段时间中被阻塞。所有从自由线程对象到公寓的调用均将阻塞，直到公寓调用释放为止。因此，这样就会导致死锁。

那么，具有自由线程的 Java 对象应该使用 WFC 控件吗？当线程转换发生时，不是将其隐藏起来，WFC 认为请求该转换是程序员的责任。程序员可以设计一个阻止死锁的方法。通过在控件线程上调用一个委托，委托再依次调用其中指定的方法，就可以解决这个问题。

要在创建控件窗口句柄和包含信息循环的线程上执行给出的委托，从要求的控件中使用 Control.invoke 或 control.invokeAsync 方法。使用控件自己的线程是很重要的，以防止万一该控件因为各种原因要重建它的窗口句柄。invokeAsync 方法导致线程来调用回调方法而不等待回应。在这两种情况下，调用线程上的所有异常都传递到自己的控件上。

# 创建和退出线程

自由线程可以使用实现 java.lang.Runnable 接口的标准 Java 方法来创建。本例显示了实现 Runnable 和获取两个控件（跟踪条和标签）作为参数到它的构造器的类。从线程的运行方法中，它通过调用跟踪条的 invokeAsync 方法来转换到跟踪条线程。InvokeAsync 传递 tDelegate 委托（一个 com.ms.app.MethodInvoker 的实例）指定名为 tCallBack 的回调方法。在该方法内，控件的线程可以安全地操作控件的属性，此时是改变跟踪条的记号样式。这将导致重新创建任务栏窗口句柄。如果线程在这里没有按演示的那样转换，跟踪条将在新的线程上重新创建，而不是包含信息循环的线程；在此时，控件跟踪条将接收任何新消息，并且不能响应。

```
import com.ms.wfc.app.*;
import com.ms.wfc.core.*;
import com.ms.wfc.ui.*;

/**
 * Runnable is the interface you need to implement to make a new
 * java thread
 */
public class RunnableClass implements Runnable
{
    final int SLEEP = 500;
```

```
Label l;
TrackBar tb;

/**
 * This is the thread for our class.
 */
Thread thread;

/**
 * Makes a special delegate so WFC can call it from the control's
 * thread.
 */
MethodInvoker tDelegate = new MethodInvoker(tCallback);

/**
 * Make a new Java thread; tell it to begin running via the
 * start() method.
 */
public RunnableClass (TrackBar tb, Label 1)
{
      this.1 = 1;
      this.tb = tb;
      thread = new Thread(this, RunnableClass thread);
      thread.start();
```

```
    }
public void run()
{
    while (true)
    {
        /**
         * Call the specified method from the label's thread.
         */
         tb.invokeAsync (tCallback);
         try
         {
             Thread.sleep (SLEEP);
         }
         catch (InterruptedException e)
         {
         }
      }
  }
int nCount = 0;
int nTickStyles [] = {TickStyle.BOTH,
                      TickStyle.BOTTOMRIGHT,
```

```
                    TickStyle.NONE,
                    TickStyle.TOPLEFT};
/**
 * This code is executed on the trackbar's thread.
 */
private void tCallback()
{
    int nIndex = nCount % (nTickStyles.length);
    l.setText ("hello from tCallBack: " + nCount);
    tb.setTickStyle (nTickStyles [nIndex]);
    nCount++;

    int nValue = tb.getValue();
    if (nValue >= tb.getMaximum())
        tb.setValue(0);
    else
        tb.setValue (nValue + 1);
}
public void stopThread()
{
    thread.stop();
    }
```

}

在这种情况下，退出线程只是停止运行该线程的方法。在本例中，当处理创建 RunnableClasss 对象的 Form 类时，它调用对象的 stopThread 方法。下面的代码段演示了这一点。

```
...
import RunnableClass;

public class SimpleRunnable extends Form
{
    /**
     * This is the class that implements the Runnable interface.
     */
    RunnableClass runnableClass;

    public SimpleRunnable()
    {
        // Required for Visual J++ Forms Designer support.
        initForm();
        runnableClass = new RunnableClass (tb, 1);
    }

    public void dispose()
    {
```

```
        runnableClass.stopThread();
        super.dispose();
        components.dispose();
    }
    Container components = new Container();
    Edit eDescription = new Edit();
    TrackBar tb = new TrackBar();
    Label l = new Label();

    private void initForm()
    {
        // Code to initialize the controls omitted...
    }

    public static void main(String args[])
    {
        Application.run(new SimpleRunnable());
    }
}
```

另一种方法是，创建一个新的应用程序线程，而不必实现 Java Runnable 接口或是展开 java.lang.Thread，用户可以使用 Application.createThread 方法。createThread 方法用一个委托作为参数（MethodInvoker 经常使用，但是可以使用任何委托）。在这种情况下，在一个类中可以包含所有的

逻辑，通常这个类是应用程序基于 Form 的类。下面的代码段说明了它们的工作方式。

```
import com.ms.wfc.app.*;
import com.ms.wfc.core.*;
import com.ms.wfc.ui.*;
public class SimpleAppThread extends Form
{
    final int SLEEP = 700;
    Thread thread;

    // Specify the thread context to run a method on.
    MethodInvoker cbDelegate = new MethodInvoker( cdThrdCallback );

    public SimpleAppThread()
    {
      initForm();
      /**
       * Creates a new thread and runs the methodInvoker method
       * on the new thread. The returned thread object is needed
       * so we can stop the thread when this form is closed (disposed).
       * Note that thread.start() is called automatically.
       */
       thread = Application.createThread (new MethodInvoker (this.methodInvoker));
```

```
    }

    private void methodInvoker()
    {
       while (true)
       {
          // cbThrdCallback is called on the check box's thread.
          cb.invoke (cbDelegate);
          try
          {
             Thread.sleep (SLEEP);
          }
          catch (InterruptedException e)
          {
          }
       }
    }
int nCount = 0;
/**
 * Thread callback that sets check box alignment property.
 * This code is to be executed on the check box's thread.
 */
```

```
private void cbThrdCallback()
{
   cb.setText ("threadCallback loop:" + nCount++);
   if (nCount % 2 == 0)
       cb.setTextAlign (LeftRightAlignment.LEFT);
   else
       cb.setTextAlign (LeftRightAlignment.RIGHT);
}
public void dispose()
{
   thread.stop();
   super.dispose();
   components.dispose();
}
private void cbSuspend_click(Object sender, Event e)
{
   if (cbSuspend.getChecked())
   {
      cbSuspend.setText ("press to resume thread");
      thread.suspend();
   }
   else
```

```
    {
      cbSuspend.setText ("press to suspend thread");
      thread.resume();
    }
}
Container components = new Container();
CheckBox cbSuspend = new CheckBox();
CheckBox cb = new CheckBox();
private void initForm()
{
  // Code to initialize the controls here...
}
public static void main(String args[])
{
  Application.run(new SimpleAppThread());
}
}
```

这就是通过从窗体的处理方法中调用线程的停止方法来中止的线程。该线程不在一个单独的类中，所以，它的方法可以直接从基于 Form 的类中调用。

Application 类还包含 Application.exitThread 方法，该方法关闭线程的消

息循环，并关闭在线程上的所有窗口（注意，它并不中止或退出线程本身）。与之形成对比的是，Application.exit 关闭在所有线程上的消息循环，并关闭所有窗口。

## 使用线程存储器

WFC Application 类提供对 Thread Local Storage（TLS，线程局部存储器）的支持。每个线程可以线程特定的数据存储分配一个内存位置。调用 Application.allocThreadStorage 返回一个该内存位置的索引。要在 TLS 中设置一个值，使用该索引和想要设置的值来调用 Application.setThreadStorage 。要检索该值，需调用 Application.getThreadStorage。记住，要调用 Application.free.Thread-确良 Storage 来释放所有已分配的线程存储器。

## 使用线程异常

com.ms.wfc.app 类提供一个 ThreadExceptionDialog 类，当在线程中发生没有处理的程序异常时，它自动显示。用户可以通过使用 Application.addOnThreadException 方法指定自己的线程异常处理程序来得到对异常的控制。addOnThreadException 方法带有一个 ThreadExceptionEventHandler 委托，它是由事件处理程序方法和 ThreadExceptionEvent 类构成的。

通常，线程异常事件处理程序查询 ThreadExceptionEvent 对象的异常字

段，传递它来确定所要采取下一个动作。从这些线程异常处理程序中，用户可以使用与其他 WFC 对话框同样的方法来运行 ThreadExceptionDialog 并检索对话结果：使用 Form.show.Dialog 方法来启动对话框，并且与 com.ms.wfc.ui.DialogResult 类字段比较返回的结果。

# 第 13 章　WFC 控件开发

与在 Visual Basic，Web 页和其他支持标准 ActiveX 控件的主应用程序中一样，利用 Visual J++和 WFC，可以创建能在 Java 应用程序中使用的控件。WFC 提供合并多数控件功能的类，包括设计和实现属性的工具、处理和调用事件等等。

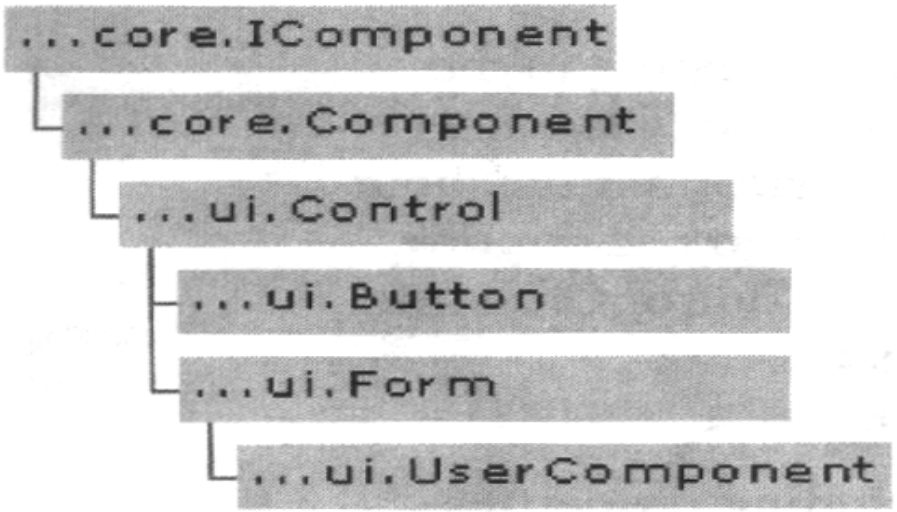
Visual J++还包括创建合成控件的可视工具，合成控件是由多个可视元素构成的控件，如复选框和列表框。利用可视工具，可以设计控件的布局，生成可以根据实现控件全部功能进行修正的适当代码。

创建控件的详细细节，参见下列列表的有关部分。

| 有关信息 | 参见 |
| --- | --- |
| WFC 中的控件技术，使用代码编写控件 | 编写 WFC 控件 |
| 使用可视工具创建合成控件 | 创建合成 WFC 控件 |

## 编写WFC控件

WFC 提供一个丰富的框架结构来开发自定义控件。下图显示 WFC 中可视控件的基本类层次。




com.ms.wfc.ui.Control 类是所有控件的基类，它可以提供控件的多数功能。

> **注意**：对合成控件来说——其他控件和事务逻辑组合创建的控件——使用的基本 WFC 软件包是 com.ms.wfc.ui。可以使用可视工具设计合成控件。更多信息参见本章后面的“创建合成 WFC 控件”。

编写控件时，可以扩展基本 Control 类，然后增加需要的成员（合成控件是通过扩展 UserControl 在可视设计器中创建的）。也可以根据需要超越从 Control 类继承的成员。下面提供有关创建控件的信息：

- **创建基本控件**。如何创建 Control 类的子类并且在设计阶段环境中放置控件。
- **定义控件属性**。如何定义并且公布控件的属性，如何为属性值指定自

定义编辑器。

- **使用控件事件。**如何为控件创建自定义事件，如何捕捉和使用标准事件。
- **定制控件。**管理控件的可视显示，指定动词和更多的信息。
- **使用控件。**这里需要做的是使控件可以应用到其他计算机的主应用程序上。

## 创建基本控件

这部分概括创建控件的过程。在这部分，可以学到如何定义基本控件，并为它的基本事件提供功能。

### 定义控件

自定义控件是 WFC Control 类的子类。为使控件在 Visual J++的工具箱中是可视的，也必须扩展 com.ms.wfc.ui.Control.ClassInfo 类。ClassInfo 类包含用于提供有关类设计信息的元数据和浏览器组件属性以及运行的事件。如果在工具箱中不需要控件是可视的，就不要扩展 com.ms.wfc.ui.Control.ClassInfo 。但是必须仍然有用以实现 com.ms.wfc.core.IClassInfo 的子类 Component.ClassInfo。这个内部类必须命名为 ClassInfo。

> **注意：**WFC Component Builder 创建了一个骨架控件，这个控件包括可以在里面填充内容的属性和方法。这部分有关如何建立控件的

信息可以帮助你手工建立控件，而且解释建立程序所做的一些事情。

下面列出一个骨架控件。如果编译它，这个控件将在计算机上注册，并且在 Customize Toolbox 对话框中变为可用的：

```
// MyControl.java
import com.ms.wfc.ui.*;
import com.ms.wfc.core.*;

public class MyControl extends Control {

    public static class ClassInfo extends Control.ClassInfo {

    }
}
```

如果把这个控件的一个实例添加到一个窗体中，在属性窗口中将会看到它的属性和事件。可以改变这些属性，也可以看到这个控件已经支持背景色、前景色、锚点、停靠、鼠标事件、焦点甚至更多。com.ms.wfc.ui.Control 类定义这些共同属性和事件的默认实现。

## 添加控件描述

可以为公布 ActiveX 控件的控件添加文本描述。然后，主应用程序可以查询和显示这个描述。为创建一个描述，需在 ClassInfo 类中添加一个

DescriptionAttribute 对象。下面的例子显示如何添加文本描述：

```
public static class ClassInfo extends Control.ClassInfo
{
    public void getAttributes(IAttributes attribs) {
    super.getAttributes(attribs);
    attribs.add(new DescriptionAttribute("This describes MyControl"));
}
```

## 为类事件提供功能

com.ms.wfc.ui.Control 类使用默认功能公布了一个共同的成员集。例如，如果如前所述创建了基本控件，则在 Properties 窗口中它的文本属性是可用的。

一般情况下，要向控件中添加功能，必须为通过基类公布的控件超越成员。WFC 中的大多数事件公布在它们带有保护的 on<事件名>成员的基类上，这可以让子类不必与事件处理程序联系就可以超越事件。在超越代码中，定义所需要事件的功能。通常，通过调用超类事件实现成员的默认功能。通过公布所调用的超类事件，可以指定事件触发的顺序。

> **注意：**当没有创建在超类中定义的事件的子类时，还有另外的方法接收事件。详情参见本章后面的“使用控件事件”。

下面举例说明怎样能超越这个保护的 onPaint 方法，来定义什么样的控件应该在运行时显示，在这个例子中，首先调用超类事件执行超类拥有

的绘图方法。控件拥有的代码通过调用 Graphics 对象的 drawString 方法来显示文本属性的值：

```
// MyControl.java
import com.ms.wfc.ui.*;
import com.ms.wfc.core.*;

public class MyControl extends Control {

    protected void onPaint (PaintEvent p) {
        super.onPaint(p);
        Graphics g = p.graphics;
        g.drawString(getText(), 0, 0);
    }

    public static class ClassInfo extends Control.ClassInfo {
    }
}
```

在这个例子中，通过调用控件的 getText 方法检索存储在控件中的文本，从而显示文本。当编译并且添加类的新版本时，现在，控件就包含了键入到文本属性的一切文本。因为 com.ms.wfc.ui.Control 类提供了文本属性，所以没有必要再创建一个新的。

OnPaint 的参数是 PaintEvent，它包含了事件特定的数据。当创建保护

成员的子类时，已经隐含知道了发送事件的对象，因为这个发送者是 this（类的当前实例）。事件数据随着事件的不同而不同。在这种情况下，PaintEvent 看起来如下所示：

```
Public class PaintEvent extends Event {
    //Graphics object with which painting should be done.
    Public final Graphics graphics;
     // Rectangle into which all painting should be done.
    Public final rectangle clipRect;
}
```

PaintEvent 的图形成员是指 com.ms.wfc.ui.Graphics 对象。这是绘图接口的 WFC 包装（wrapper），一个 Win32 设备的上下文。Graphics 对象公布方法来绘制字符串、行、位置、省略号等等。可以改变字体、前景色和背景色属性的值，这将正确显示，因为通过 PaintEvent 事件传入的 Graphics 对象是使用基于控件中设置的正确字体和刷子建立的。有关绘图的更详细资料参见本章后面的“修改可视显示”部分。

## 使用窗口句柄

Control 基类通过扩展 com.ms.wfc.app.Window 的私有成员封装 Win32 HWND（句柄）。Window 类管理为控件创建的 HWND 窗口过程子类。无论什么时候需要一个带有调用 Control.getHandle 的句柄，如果需要，就会创建这个句柄，或句柄存在时就返回句柄。创建句柄时，Control

类首先调用 getCreateParams 方法为创建句柄确定正确的窗口样式。如果想指定窗口样式或扩展窗口样式，可以超越 getCreateParams 方法。创建句柄后，马上激活 createHandle 事件。在撤消这个句柄之前，立即激活 destroyHandle 事件。因此在创建和撤消句柄事件期间，这个句柄是合法的。

任何给定句柄的生存期都是通过控件的实现来确定的。但是，任何控件的实例的生存期都不依赖于句柄的生存期。

WFC 允许以不同的方式操作需要在 Windows 中重建的控件。例如，可以使用控件，如标尺。当改变标尺的单位时，可能导致创建重建这个控件，使 HWND 无效，如下例所示：

```
{
h = x.getHandle();
x.units = "points"; // change property that results in recreation
// h is now invalid
}
```

通常，在使用控件之前，立即利用句柄得到它是最安全的。

有时，可能需要调用 recreateHandle 方法来重新创建句柄（先撤消然后立即再次创建）。如果调用 recreateHandle 方法，那么在这个句柄再次创建之前，getRecreatingHandle 将返回 true。这允许用户在 destroyHandle 事件中公布在句柄重建期间从 HWND 检索状态的特殊逻辑。一个例子是列表框，如果句柄重建，你可能想从破坏的句柄的列表框中保存所有

项。如果需要窗口格式重新应用程序到窗口，而不需要重建句柄，则可以调用 updateStyles。这将调用 getCreateParams 方法，并且在 HWND 中，重新应用这个格式和扩展格式。
注意，如果没有创建句柄，调用这些函数不会起任何作用。这在属性设定方法中是有用的，因为不想迫使句柄创建，而如果句柄已经创建，可能需要改变 HWND。通过这个约定，就可以不必为简单的属性变化而强制创建句柄了。较早强制创建句柄可能代价较大，因为用户可能设置需要句柄重建的其他属性。需要创建句柄的最好方法只能是显示控件的可视性，但不调整属性。

## WFC 控件中的线程

Java 语言通过类集和到控件线程创建与执行的 API 来支持多线程。Win32 窗口是固有的公寓线程。可以在任何线程上创建它们，但是它们不能交换线程，并且调用那个窗口的所有函数必须在它的主线程上出现。WFC 控件（从 com.ms.wfc.ui.Control 派生的所有东西）需要这种线程模型，而多数的 WFC 组件是自由线程。
Control 类提供两个函数，invoke 和 invokeAsync，将把所有代表调度到控件的主窗口线程上并且执行。如果试图在 Win32 HWND 下操作的控件上调用一个函数，这将通过 Windows 调度到正确的线程。通过直接调用 invoke 函数，可以成批调用到 HWND，并且限制所出现的交叉线程调度费用的数量。

**注意：** Windows 消息循环依赖于在上面创建 HWND 的线程。如果在一个新线程上创建控件的 HWND，则这个新线程不再接收信息。对更详细的资料，参见第 12 章“WFC 编程概念”中的“使用 WFC 和 Java 线程”。

虽然大多数 WFC 组件支持自由线程，但大多数方法调用不是同步的。同步调用非常昂贵，只有需要时才会这样做。例如，在对象上同步调用多个方法，将得到非常有用的执行结果。

## 定义控件属性

大多数控件公布一个允许用户确定控件行为和外表的设置或属性。在 WFC 控件中，可以使用 get 和 set 函数把属性创建为成员。但是 WFC 还提供类和接口帮助用户把控件集成到设计阶段环境中。
下面提供有关在控件中创建、公布和自定义属性的信息。

### 创建和公布属性

定义控件属性与向任何类中添加属性相似。除此之外，向 ClassInfo 类中添加成员可以使属性在 Properties 窗口中是可见的。

#### 添加属性定义

属性存储在控件的私有成员变量中。可以提供一个公共的 get<属性>方法来公布属性值。如果想使属性是可读写的，还可以提供公共的 set<属性>方法用来获取包含新属性值的参数。

注意：WFC Component Builder 将为用户创建属性骨架。

在 MyControl 控件中调用简单的整数属性 myProp 可以在下例中实现：

```
public class MyControl extends Control {
    private int myProp=0;  // 0 is the default value
    public getMyProp() {
        return myProp;
    }
    public setMyProp（int newValue）{
        myProp=newValue;
    }
}
```

注意：属性定义中使用的名字应该以一个小写字母开头，除非开始两个字母都是大写。紧跟在 get 或 set 后面的函数名字应该有一个大写字母。例如：对 text 函数应该有 getText 和 setText，而对于 MDIChild 应该有 getMDIChild 和 setMDIChild。

在设计阶段公布属性

为了在 Properties 窗口中使属性是可见的，可以做下列事情：

- 创建指出新属性的类、名字和数据类型的 PropertyInfo 类的一个静态最终实例。
- 超越超类的 getProperties 方法，并且添加超类的现有属性，其次是新

属性。

**注意：** getProperties 方法是用于公布方法、事件、扩展器（extender）和类属性的可以使用的几种方法之一。有关其他方法的细节参见本章后面的“使用控件事件”。

下例显示 myProp 属性的 ClassInfo 类：

```
public static class ClassInfo extends Control.ClassInfo {
    public static final PropertyInfo myProp =
        new propertynfo(MyControl.class, "myProp", int.class);
    public void getProperties(IProperties props) {
        // Add existing properties form parent class
        super.getProperties(props);
        props.add(myProp);     // adds custom property
    }
}
```

有关使用 PropertyInfo 类的更多信息，参见本章后面的“指定自定义控件属性”。

## 对齐属性举例

这部分提供 SuperLabel 类对齐属性的完整举例，它允许用户指定如何要求标签内的文本对齐：靠左，靠右和居中。这个例子说明如何做下列操作：

- 定义和公布属性。
- 使用枚举（enum）类为属性定义允许使用的值。
- 为新属性值提供验证。
- 画控件时使用对齐属性值。
- 在设计阶段公布属性。

这个例子的代码，参见本章后面的“一个完整例子”。

### 基本对齐属性

创建对齐属性的代码如下所示。在 setAlignment 方法中，调用 invalidate 函数，间接强制这个控件重画，以便它影响新属性值：

```
private int align=AlignStyle.LEFT;

public int getAlignment(){
    return align;
}

public void setAlignment(int value) {
    align = value;
    invalidate() ; //Repaint control when property changes
}
```

## 创建枚举属性值

因为属性只有三个可能的值，可以使用一个枚举（enum）定义它们。虽然 Java 没有枚举类型的支持，但是 WFC 提供了一个 com.ms.wfc.core.Enum 类。任何从 com.ms.wfc.core.Enum 得到的类都看成是 Enum 对象。所有 Enum 对象都遵守只包含公共静态最终整数的标准，而这个公共静态最终整数代表有效的选择。它们可以随意包含一个有效的方法，如果在 Enum 对象中传递的值是有效的，则这个方法返回 true。

枚举对象在 Properties 窗口中是可识别的。所以，当给 Enum 类分类时，在设计阶段的 Properties 窗口中将会自动得到一个有效项目的下拉表。

下面例子显示的是定义一个 AlignStyle 枚举的类：

```
//AlignStyle.java
import com.ms.wfc.core.*

public class AlignStyle extends Enum
{
    public static final int LEFT=1;
    public static final int RIGHT=2;
    public static final int CENTER=3;

     //Optional valid method to test passed value
    public static boolean valid(int value) {
        return (LEFT<=value &&value<=CENTER);
```

```
    }
}
```

**注意：**把有效的方法创建为静态成员允许用户调用它，而不必先创建一个类实例。

### 提供属性验证

设置属性时，为测试用户指定的有效对齐值，可以向 setAlignment 方法中添加验证。为测试这个值，可以调用 AlignStyle 枚举的有效方法。如果有错误，可以施加 WFCInvalidEnumException 异常。
下例显示使用添加的验证修改的 setAlignment 方法：

```
public void setAlignment（int value） {
    if（！ AlignStyle.valid(value)）
        throw new   WFCInvalidEnumException("value", value, AlignStyle.class);
     align=value;
    invalidate(); //Repaint control when property changes
}
```

**注意：**有关使用 invalidate 重画控件的信息，参见本章后面的“修改可视显示”。

通过施加 WFCInvalidEnumException ，可以得到非检查异常和与之关联的预定义信息。这个信息转换成多种语言，以便在不同场所运行这个

控件，自动获得有意义的错误信息。

## 画控件时使用属性值

对齐属性影响控件中文本的外表。所以，无论什么时候画控件，绘制代码必须检查属性值，从而使用它。

通过超越 onPaint 方法指定绘制行为。下例显示的是可以读对齐属性值，然后在 drawString 方法中使用这个值设置对齐。在 com.ms.wfc.ui 的 TextFormat 中定义 LEFT，RIGHT 等常数。

```
protected void onPaint(PaintEvent p) {
    Graphics g = p.graphics;
    int style = 0;
    switch (align) {
      case AlignStyle.LEFT:
          style = TextFormat.LEFT;
          break;
      case AlignStyle.RIGHT:
          style = TextFormat.RIGHT;
          break;
      case AlignStyle.CENTER:
          style = TextFormat.HORIZONTALCENTER;
          break;
    }
```

```
        g.drawString(getText(),getClientRect(), style);
}
```

## 在设计阶段公布属性

在设计阶段为使对齐属性可用，需要把 ClassInfo 分成首先定义属性的类（通过创建 PropertyInfo 的实例）。然后通过调用 getProperties 方法公布属性：

```
public static class ClassInfo extends Control.ClassInfo {
    public static final PropertyInfo alignment = new PropertyInfo(
        MyControl.class, "alignment", AlignStyle.class);
     Public void getProperties(IProperties props) {
        Super.getProperties(props);
        Props.add(alignment);
    }
}
```

> **注意：**传递到 PropertyInfo 的第三个参数（在这里是 AlignStyle.class）是属性的正常数据类型。然而，因为这个属性是一个枚举值，所以传递创建扩展 Enum 的类。

因为 AlignStyle 是从 Enum 类派生的，所以 Properties 窗口使用与 Enum 类相关的编辑器。结果是允许设置属性值的下拉列表。

注意：为创建不在设计阶段公布的属性（例如，只是运行时属性），使用 PropertyInfo 对象把 BrowsableAttribute 属性设置为 NO。详情参见本章后面的“指定自定义控件属性的属性”。

## 一个完整例子

下例显示使用了对齐属性的完整标签控件，这个对齐属性包括了前面讨论的所有特征：

```
// SuperLabel.java
import com.ms.wfc.ui.*;
import com.ms.wfc.core.*;

public class SuperLabel extends Control {
    private int align=AlignStyle.LEFT;
    public int getAlignment(){
        return align;
    }
     public void setAlignment(int value) {
        if (! AlignStyle.valid(value))
           throw new WFCInvalidEnumException("value", value, AlignStyle.class);
        align=value;
        invalidate(); //Repaint control when property changes
    }
```

```
 protected void onPaint(PaintEvent p) {
    Graphics g=p.graphics;
    int style=0;
    switch   (align) {
      case AlignStyle.LEFT:
          style=TextFormat.LEFT;
          break;
      case AlignStyle.RIGHT:
          style=TextFormat.RIGHT;
          break;
      case AlignStyle.CENTER:
          style=TextFormat.HORIZONTALCENTER;
              break;
    }
    g.drawString(getText(),getClientRect(),style);
}
 public static class ClassInfo extends Control.ClassInfo {
    public static final PropertyInfo alignment=new PropertyInfo(
      SuperLabel.class, "alignment", AlignStyle.class);
      public void getProperties(Iproperties props){
      super.getProperties(props);
```

```
            props.add(alignment);
        }
    }
}
```

## 指定自定义控件属性

当创建控件属性并且创建 PropertyInfo 对象来设置它的属性时,可以指定不同选项来自定义这个属性。

> **注意：**使用 PropertyInfo 类的详细信息已经在本章前面部分“定义控件属性”中提供了。

创建 PropertyInfo 对象的基本语法是：

```
new PropertyInfo (class owner, String name, Class dataType)
```

例如，下面是在 Mycontrol 控件中名叫 myProp 的属性的 PropertyInfo 对象：

```
public static final PropertyInfo myProp=
    new PropertyInfo (MyControl.class,”myProp”, int.class);
```

可以通过为 PropertyInfo 对象提供任何数量的附加成员属性来自定义属性。一个成员属性被定义成从 com.ms.wfc.core.MemberAttribute 得到任意类。在 PropertyInfo 类中默认的某一数量的属性是可用的，或者通过从 MemberAttribute 得到的属性可以创建和添加自己的属性。下表列出

在 PropertyInfo 类中预定义的属性。

| 属性 | 说明 |
| --- | --- |
| BrowsableAttribute | 指定在 Properties 窗口中属性是否是可见的。默认值是 YES。这个属性可以与 PersistableAttribute 属性（见下面）联合使用来定义控件属性，这个控件属性的值保存在控件中，但是在 Properties 窗口中不能编辑 |
| CategoryAttribute | 指定放入 Properties 窗口中的当前控件属性的类别，如 Appearance，Behavior 等等。默认类别是 Misc。可以通过使用 CategoryAttribute 中的一个静态成员在预定义类别中公布这个控件，或者通过构造一个新 CategoryAttribute 并且在需要调用的类别字串中传递它来创建一个新类别 |
| DataBindableAttribute | 指定控件属性是数据绑定的候选属性。默认值是.NO。当这个属性值设为.YES 时，DataBind 将在下拉列表中列出这个控件属性 |
| DefaultValueAttribute | 简单属性的默认值。当用户在 Properties 窗口中重设属性值时，使用这个值。Forms Designer 还比较默认值与当前值。如果它们匹配，则 Forms Designer 不生成设置属性值的代码，这可以减少为窗体生成的代码量<br>**注意**：还可以通过创建一个 reset<属性>方法来创建一个默认值。详情参见下一部分“指定动态默认值 |

| | |
|---|---|
| | 和永久性属性”。 |
| | 不同数据类型的许多公共值都预定义成静态值（例如，对布尔值，已经定义了值 TRUE 和 FALSE） |
| DescriptionAttribute | 当用户选择当前控件属性时，定义在 Properties 窗口的底部显示的文本。默认值是“” |
| LocalizableAttribute | 如果用户选择本地化一个窗体，则指定控件属性存储在源文件中。默认值是 NO。然后用户可以不必修改代码而本地化这个源文件 |
| PersistableAttribute | 在设计阶段保存控件属性的包容器时，指定是否保存这个控件属性的值。默认值是 YES。当不想保存控件属性的值时主要使用这个属性，例如，瞬时控件属性值，它的值依靠状态或计算结果。通常，把控件属性的这个属性设为 NO 时，这个控件属性也是不可见的（BrowsableAttribute 值是 NO） |
| ValueEditorAttribute | 为当前控件属性指定一个被 Properties 窗口使用的自定义编辑器。如果指定一个无值编辑器，则 Properties 窗口使用一个与控件属性的数据类型相关的编辑器。有关详情，参见本章后面的“创建自定义属性值编辑器” |

为了指定一个属性，需要创建一个所需属性的实例或者使用一个预定义实例（例如 BrowsableAttribute.NO），然后使用如下语法把它添加到 PropertyInfo 定义中：

```
new PropertyInfo(Class owner, String name, Class dataType,
    [MemberAttribute, [MemberAttribute, ...]])
```

下例显示如何使用预定义属性为 myProp 控件属性定义一个描述和默认值：

```
public static final PropertyInfo myProp=
    new PropertyInfo（MyControl.class,"myProp",int.class,
    new DescriptionAttribute("Test property"),
     new DefaultValueAttribute(DefaultValueAttribute.ONE);
```

只需把属性作为参数包括到 PropertyInfo 构造器中，就可以有总计六个属性。如果想指定多于五个的成员属性，可以为接收成员属性数组的 PropertyInfo 构造器使用语法。对更详细的信息，参见“Microsoft Visual J++6.0 库参考”一书中的 Microsoft Visual J++6.0 WFC 库参考的第一部分，Package com.ms.wfc.core 中的“PropertyInfo”。

## 指定动态默认值和永久性属性

与“指定自定义控件属性的属性”中描述的一样，当创建一个新的 PropertyInfo 对象时，为设置一个属性的默认值，可以指定一个 DefaultValueAttribute 属性。有时，可能想指定一个默认动态值，这指的是在运行时计算的值。同样，在运行时条件的基础上可能想指定一个永久属性值（在设计阶段保存）。

## 设置永久属性值

默认情况下，在设计阶段，当一个控件的属性值从默认值发生变化时（与在 Properties 窗口中一样），无论什么时候保存主控件都要保存控件的属性值。例如，如果在 Visual J++窗体中实例化控件，并且 Properties 窗口中如果改变属性的值，则保存窗体时，也就保存了这个属性的值。为了更有效率，如果属性值与它的默认值相匹配，则这个值不是永久的。在少量的实例中，可能要在运行时条件的基础上指定一个永久属性值。为使控件保持永久性，需超越 Control 类的 shouldPersist<属性>方法。条件测试后，这个方法应该返回一个表明自定义值是否被保存的布尔值。为避免保存这个值，从这个方法返回 false。还可以为从超类继承的属性超越 shouldPersist<属性>，与这个例子中一样，依靠字体是否改变来设置字体属性的永久性：

```
public boolean shouldPersistFont() {
    return font !=null;
}
```

> **注意：**比较上述说明，周围属性（它们的值是从父属性继承的）例如字体和背景色返回 null，代表它们被设置成默认值。但是，明确地获得这样属性的值——例如，调用 getFont 方法——则返回实际字体信息。

## 计算动态默认值

为了给在运行时属性创建一个默认值，需要为这个属性创建一个 reset<属性>方法。下面例子显示如何为名为 lastUpdate 的属性实现一个基于今天日期的默认值：

```
private Date dtLastUpdate;
public void setLastUpdate (Date d) {
      dtLastUpdate = d;
      invalidate(); //Repaint control when property changes
}
public void resetLastUpdate() {
    return new Date();
}
```

还可以超越从 Control 类继承的 reset<属性>方法。下面例子显示如何为控件超越 resetFont 方法：

```
public void resetFont() {
    Font f = new Font("Arial", 8.0f, FontSize.POINTS, FontWeight.BOLD,
        false, false, false);
     SetFont(f);
}
```

## 创建自定义属性值编辑器

Visual J++ 6.0 的 Properties 窗口为公布在控件中的属性值提供了编辑能力。事实上，Properties 窗口正好是一个真正的主编辑器。可以指定一个自定义编辑器，或者可以依靠 Properties 窗口的默认编辑能力。
这种机制是这样的：Properties 窗口首先检查你是否指定了自定义值编辑器。如果没有，它将在属性类型的类中寻找一个名叫 Edit 的内部类。如果没有发现编辑器，它遍历类型层寻找具有编辑器的任何超类。因为有一个与 java.lang.Object 关联的

ObjectEditor 类，所以总会发现一个编辑器。Properties 窗口将为所有简单类型提供一个默认编辑器，如字串编辑器、整数编辑器等等。
指定自定义值编辑器的原因有许多。典型的自定义属性编辑器任务包括：

- 文本向值的转换
- 值绘制
- 指定值的子属性
- 显示自定义对话框
- 显示自定义下拉列表
- 通过指定一个应该取代指定值的常数名来参与代码生成。

详情参见下一部分“定义自定义值编辑器。”

## 定义自定义值编辑器

为创建自定义值编辑器，必须实现 com.ms.wfc.core.IvalueEditor。一个方便的方法是扩展 com.ms.wfc.core.ValueEditor，用以提供所有成员的默认实现。

为创建在 Properties 窗口中显示的值（作为字串），超越 getTextFrom 方法。为改变在控件中受到影响的值，并且在 Properties 窗口中显示，超越 getValueFrom 方法。

下面例子说明一个在 Properties 窗口用“%”显示百分比值的简单自定义值编辑器，不过在存储这个值之前，除去了标点符号。这个例子利用了适合转换的 com.ms.wfc.util 类中的 Value 类。

```
import com.ms.wfc.core.*;
import com.ms.wfc.util.*;

public class PercentEditor extends ValueEditor {
    // Ctreates string as "nn%" for display in the Properties window
    public String getTextFromValue(object value) {
        return Value.toString(Value.toInt(value)) + "%";
    }
     // Converts string in format "nn%" to integer.This method
    // is required.
    public Object getValueFromText(String text) {
        String s= Utils.trimBlanks(text.replace("%","" ));
```

```
        return Value.toObject(Value.toInt(s));
    }
}
```

为使用正确的编辑器交换信息而通过 com.ms.wfc.core 提供的方法列表如下：

| 方法 | 说明 |
|---|---|
| editValue | 为编辑正确的值提供用户界面 |
| GetConstantName | 检索代替指定值的常数的全名 |
| GetStyle | 返回样式标志的位字段 |
| GetTextFromValue | 把值转换为字符串 |
| GetValues | 返回值的数组；通常，通过在循环中调用 getTextFromValue 可以在 Properties 窗口的一个列表中显示这些数组 |
| GetValueFromText | 从一个字符串返回一个值，必须实现这个方法 |
| PaintValue | 在指定的范围内描述一个值的表示法 |

两个附加方法——getSubProperties 和 getValueFromSubPropertyValues——允许使用对象属性工作，例如 Font 对象，可以作为构造器的一部分正常传递。

getStyle 方法用于获取和设置在值编辑器中指定行为的标志。有效的格式是：

- STYLE_NOEDITABLETEXT 表明值编辑器不支持在

getValueFromText 方法中接收任意文本。它应该接收从 getTextFromValue 返回的文本。一般情况下，这个格式将导致属性编辑器禁止在值编辑器类型的属性编辑框中键入字符。

- STYLE_PAINTVALUE 表明实现 paintValue 方法的值编辑器。一般情况下，这将导致属性编辑器描述值编辑器类型的属性表示法。
- STYLE_VALUES 表明实现 getValue 方法的值编辑器。一般情况下，这将导致属性编辑器显示值编辑器类型的属性下拉列表。
- STYLE_EDITVALUE 表明实现 editValue 方法的值编辑器。。一般情况下，这将导致属性编辑器显示一个值编辑器类型的属性省略按钮。
- STYLE_DROPDOWNARROW 表明如果也返回 STYLE_EDITVALUE，应该显示一个下拉箭头而不是省略按钮。如果值编辑器想在 editValue 中显示一个与启动模型对话框不同的下拉列表，这个格式是很有用的。
- STYLE_IMMEDIATE 表明这个类型的值适合修改，甚至当这个新值不完善的时候也可以修改。一般情况下，这将导致一个属性编辑器在编辑器发生的每一次变化后修改基本属性（例如，在编辑框中输入的每个字符）。
- STYLE_PROPERTIES 表明这个类型的值可以有能编辑的子属性，例如 Font 对象里的那些值。
- STYLE_NOARRAYEXPANSION 表明如果值是一个数组，则属性检查器不能进入数组。

- STYLE_NOARRAYMULTISELECT 表明如果值是一个数组，则属性检查器不应该有选择所有数组元素的入口。

下面例子显示怎样可以把一个模态对话框显示为自定义属性编辑器。在这个例子中，可以看到 IValueAccess 如何在当前正编辑的编辑器组件中提供一个获取和设置值的抽象方式。如果选择了一个以上的组件，这是很有用的。当通过 IValueAccess 调用获取和设置值的方法时，Properties 窗口可以在多个组件上正确设置值。为显示调用对话框的省略按钮，这个例子超越了 getSyle 方法并且在 getStyle 返回的位字段中设置 STYLE_EDITVALUE。

```
import com.ms.wfc.core.*;
import com.ms.wfc.util.*;

public class PercentEditor extends ValueEditor {
    public void editValue (IValueAccess valueAccess) {
        PercentDialog dialog=new PercentDialog();
        Int value=Value.toInt(valueAccess.getValue());
        dialog.setPercentValue(value);
        int result=dialog.showDialog();
        if (result= =DialogResult.OK) {
            valueAccess.setValue(dialog.getText());
        }
    }
```

```
    public int getStyle() {
        //Using OR operator sets bitfield to display ellipsis button
        return super.gettStyle() | STYLE_EDITVALUE;
    }
}
```

下面例子说明如何显示下拉对话框。当编辑属性时，控件调用IeditorHost.drop-DownDone。除此之外，控件还负责调用 IValueAccess 的 setValue 方法。如果用户以任何方式取消下拉对话框，则 Properties 窗口将关闭下拉列表：

```
import com.ms.wfc.core.*;
import com.ms.wfc.util.*;

public class PercentEditor extends ValueEditor {
    public void editValue(IvalueAccess valueAccess) {
        PercentControl control = new PercentControl();
        int value = Value.toInt(valueAccess.getValue());
        dialog.setPercentValue(value);
        IEditorHost host =
            (IEditorHost)valueAccess.getService(IEditorHost.class);
        control.sdtValueAccess(valueAccess);
        host.dropDownControl(control);
    }
```

```
    public int getStyle() {
        return super.getStyle() | STYLE_EDITVALUE | DROPDOWNARROW;
    }
}
```

# 使用控件事件

作为控件的子类，控件自动支持一个标准的 Windows 事件设置。可以在控件中捕获并且处理这些事件，并可选择把它们传递到主应用程序中。除此之外，还可以定义只有你的控件中才有的自定义事件，并且在需要时，激活它们。

## 用控件捕获用户交互

在标准事件送到主应用程序之前，控件可以捕获并且处理这些标准事件。交互作用的两个最常见的类型是鼠标和键盘事件。

### 捕获鼠标事件

WFC 中的 Control 类提供了大多数鼠标事件，这些事件包括 mouseMove，mouseUp，mouseDown，mouseWheel，mouseEnter 和 mouseLeave。除此之外，还可以选择接收单击和双击事件。

为了在控件中接收鼠标事件，可以超越所需的事件。为把鼠标事件传递到主应用程序中，需调用超类的相应事件方法。

下面这个例子说明如何超越 mouseMove 事件。处理程序直接显示鼠标

的位置，这个位置在传递到处理程序的 MouseEvent 对象的 x 和 y 属性中是可用的：

```
protected void onMouseMove( MouseEvent e) {
    String sMsg = "" + e.x +"," + e.y;
    This.setText(sMsg);
     Invalidate();   // Repaint control when property changes
    Super.onMouseMove( e ); // to make it visible in the host
}
```

## 捕获键盘事件

公布的标准键盘事件是 keyUp，keyDown，keyPress。这些事件是为常规键和系统键（例如，F1 到 F12）触发的。
输入到 WFC 的所有键盘都使用 Unicode 字符数据进行工作。当在一个不支持 Unicode 信息的操作系统下（像 Windows 95）运行时，WFC 框架将自动执行所需要的信息过滤生成 Unicode 事件。
下面例子说明在一个控件内如何捕获键盘事件。这里超越的 onKeyUp 方法在把事件传递到主应用程序之前先把数值字符过滤出来：

```
protected void onKeyUp (keyEvent e) {
    if (e.keydata<key.D0&& e.keyData>Key.D9) {
        super.onKeyUp(e);
      invalidate(); //Repaint control when property changes
    }
```

}

## 创建自定义事件

WFC 控件支持从 Control 类继承的事件的标准设置，如 click，mouseDown，keyPress 等等。控件自动公布这些事件，并且，在自己这一部分中，不用做特殊的处理就可以在主应用程序中使用它们，除非想超越这个事件来添加特殊功能。

> **注意：**有关创建事件处理程序的更多信息——在主应用程序中接收事件——参见第 12 章。有关如何捕获标准事件并且在控件中处理它的详情，参见本章前面的“使用控件捕获用户交互”。

有时，你可能想创建只在你的控件才有的自定义事件。例如，想让控件使用一个事件来通知状态发生变化的主应用程序，例如，初始化进程的完成，或者报告一个错误条件。

> **注意：**按照约定，为标明属性值的变化，需要利用下面描述的技术创建 on<属性>Changing 和 on<属性>Changed 方法。更多细节参见本章后面的“提供属性变化通知”。

自定义事件依赖于 Visual J++委托技术。为实现自定义事件，需要创建可以在应用程序中把事件处理程序绑定到事件的委托。

### 使用事件委托

WFC 的事件模型使用委托把事件绑定到处理它们的方法中。为了在

Visual J++中定义一个事件，给Even类分类，然后创建一个基于要绑定的com.ms.lang.Delegate的委托类。委托允许其他类通过指定处理程序方法为事件通知注册。事件出现时（被调用），委托调用绑定的方法。委托可以绑定到单个方法中，或绑定到多个方法中，这称做多点传送（multicasting）。为事件创建委托时，一般是创建一个多点传送事件。有少量的异常可以是导致指定进程（显示一个对话框）的事件，而这个指定进程不能感觉到每个事件重复多遍。

多点委托维护一个所绑定到的方法的调用表（invocation list）。多点委托支持一种组合的方法把一个方法添加到调用表中，并且添加一个用来删除这个新增方法的删除方法。

一个控件通过调用事件的委托可以激活这个事件。委托按次序调用绑定的方法。在大多数情况下，（多点委托）委托按次序调用调用表里的每个绑定方法，提供一个一对多的通告。这个策略意味着控件不需要维护一个用于事件通告的目标对象表——委托处理程序全部注册、通告和注销。

委托也允许把多个事件绑定到同一个方法中，允许一个多对一的通告。例如，按钮单击事件和菜单命令单击事件都可以调用相同的委托，然后委托调用单个方法用相同的方式处理这两个独立事件。

和委托一起使用的绑定结构是动态的——可以在运行时把一个委托绑定到调用号与事件处理程序相匹配的任何方法。这个特征允许靠条件建立或改变绑定方法，并且把事件处理程序动态附加到控件上。

## 创建自定义事件类

如果想使用事件传递事件指定数据（与 Control 类的 mouseMove 事件中 x 和 y 属性相似），必须创建一个自定义事件类并且定义它的数据。为创建自定义事件，需要给 Event 类建立子类。为使事件成为一个最高层的公共类，它必须是在一个独立的 java 文件中，在你的类中，可以定义成员来容纳事件数据。

下面例子显示怎样创建一个简单错误事件类，这个事件类有两个字段，一个用于错误号，另一个用于错误信息的文本：

```
// ErrorEvent.java
import com.ms.wfc.core.*;
public class ErrorEvent extends Event{
    public final int errorNumber;
    public final String errorText;
    public ErrorEvent( int eNum, String eTxt){
        this.errorNumber = eNum;
        this.errorText = eTxt;
    }
}
```

## 创建委托

为允许事件绑定，需要为事件创建一个用来声明指定调用号的委托处理程序类。如果想把委托绑定到多个方法中，需要把它声明为多个委托。

委托应该是最高层类，所以它应该在一个独立的文件中。
下面例子显示前面说明的 ErrorEvent 类的处理程序类：

```
// ErrorEventHandler.java
import com.ms.wfc.core.*;
public final multicast delegate void ErrorEvent(Object sender,
    ErrorEvent event);
```

多点委托必须返回 void，因为它们不能返回一个以上方法的结果。

## 实现自定义事件

为在控件中实现自定义事件，需要为主应用程序提供一个方式为事件注册。先创建一个私有委托实例。然后创建主应用程序能调用事件注册和绑定到指定方法的 addOn<事件>方法。如果是多点委托，在 addOn<事件>方法中调用委托的组合方法用来把用户方法添加到委托调用表中。
下面例子显示怎样为前面说明的 ErrorEvent 类执行这些步骤：

```
// Create instance of delegate
private ErrorEventHandler err = null;
// Call delegate's combine method to add binding to invocation list
public final void addOnErrorEvent(ErrorEventHandler handler){
    err = (ErrorEventHandler)Delegate.combine(err, handler);
}
```

一般情况下，还可以提供一个 removeOn<事件>方法，以便主应用程序

能为事件注册：

```
public final void removeOnErrorEvent (ErrorEventHandler handler) {
    err= (ErrorEventHandler)Delegate.remove(err,handler);
}
```

如果正使用非多点委托，则 addOn<事件>方法的语法很简单：

```
private ErrorEventHandler err=null;  //Create instance of delegate
public final void addOnErrorEvent (ErrorEventHandler handler) {
    err=handler;
}
```

> **注意：** 从控件得到的 Component 类包含实用程序代码用于添加、删除和激活事件。

在设计阶段公布事件

在设计阶段公布事件与公布属性相似。首先，创建标有新事件的类、名字和委托的 EventInfo 类的一个实例。然后超越超类的 getEventsEventInfo 方法，添加超类的存在事件，然后再添加新事件。

> **注意：** 事件名应该以一个小写字母开头（如 errorEvent），除非开头两个字母都是大写（例如，MIDIChildActivated）。

下面举例说明以前看到的 ErrorEvent 事件的 ClassInfo 类：

```
public static class ClassInfo extends Control.ClassInfo{
```

```
    public static EventInfo errEvt = new EventInfo(MyControl.class,
        "errorEvent", ErrorEventHandler.class);
    public void getEvents(IEvents events({
        super.getEvents(evetns);
        evetns.add(errEvt);
    }
}
```

## 激活事件

为激活事件，需要创建事件类的实例，然后把它传递到在类中定义的事件指定参数中。然后调用委托，如下例所示：

```
// the err delegate must already exist
int errNumber=1020;
String errText="Invalid value."
ErrorEvent e=new ErrorEvent (errNumber, errText);
err.invoke(this,e);
```

WFC 中的大多数事件包括一个 protected<事件名>成员，这个 protected<事件名>成员让子类超越事件，并且确定通过调用 super.on<事件名>的位置而触发的事件顺序。下面显示 ErrorEvent 的保护成员：

```
Protected void onErrorEvent (ErrorEvent event) {
    If (err ! =null) {
```

```
        err.invoke(this,event);
    }
}
```

如果包括保护成员，可以在控件内通过调用事件，而不是直接调用调用方法来激活这个事件，如下例所示：

```
if (value>20) {
    onErrorEvent(new ErrorEvent(1020,”Invalid value”));
}
```

一个完整的例子

下面显示一个完整的，包括前部分描述的 ErrorEvent 事件的简单控件。

```
// MyControl.java
import com.ms.wfc.ui.*;
import com.ms.wfc.core.*;
import com.ms.lang.Delegate;

public class MyControl extends Control {
   private int myProp = 1;
   public int getMyProp() {
       return myProp;
   }
   public void setMyProp(int value) {
```

```
        // Fires error event if property value exceeed 200
        if(value >200) {
                onErrorEvent(new ErrorEvent(1020, "Invalid value"));
      }
      myProp = value;
}
protected void onPaint( PaintEvent p) {
      super.onPaint(p);
      Graphics g=p.graphics;
      g.drawString( getText(), 0, 0);
}
// event setup to be able to fire event "ErrorEvent"
private ErrorEventHandler errDelegate = null;
public final void addOnErrowEvent(ErrorEventHandler handler) {
   errDelegate = (ErrorEventHandler)Delegate.combine(errDelegate,
        handler);
}
public final void removeOnErrorEvent(ErrorEventHandler handler){
```

```
    errDelegate = (ErrorEventHandler)Delegate.remove(errDelegate,
        handler);
    protected void onErrorEvent(ErriorEvent event) {
      if(errDelegate != null) {
        errDelegate.invoke(this,event);
      }
}
public static class ClassInfo extends Control.ClassInfo{
    public static final PropertyInfo myProp = new
        PropertyInfo(MyControl.class, "myProp", int.calss);
    public void getProperties(Iproperties props){
        super.getProperties(props);
        props.add(myProp);
    }
     public static EventInfo evt = new EventInfo(MyControl.class,
        "errorEvent,ErrorEventHandler.class);
     public void getEvents(IEvents events) {
        super.getEvents(events);
          events.add(evt);
```

```
    }
  }
}
```

## 提供属性变化通告

WFC 为标有属性值正在改变或已经改变的创建事件使用一个命名约定：

- <属性名>Changing 事件表明用来改变属性值的函数已经被调用。一般情况下，在 set<属性>方法的开始处激活这个事件。把它传递到 CancelEvent 对象；如果主应用程序已经有一个<属性名>Changing 事件处理程序，这个处理程序可以防止通过把 CancelEvent 对象的取消属性设定为 true 而改变这个属性。
- 为标明一个成功完成了的变化，可以激活<属性名>Changed 事件。

**注意：** 数据绑定需要实现<属性名>Changed 事件。

这些事件不是为提供数据有效性而设计的，但是允许控件越过被编辑的属性，例如防止一个控件修改它的值。数据有效性是在数据源上（例如一个记录设置）或通过在控件上自定义属性来处理的。

实现属性变化事件的方式是做一个自定义事件。对每个事件，可以提供一个 addOn<属性>Changing 和 addOn<属性>Changed 方法以及相应的 removeOn 方法。还可以创建保护的 on<属性名> Changing 和 on<属性>Changed 方法。on<属性名>Changing 通常使用一个 CancelEvent 对象

允许停止这个变化。有关创建这些方法的细节，参见本章前面的“创建自定义事件”。
下面举例说明怎样为调整属性包括属性通告：

```
public void setAlignment(int value) {
    if (alignment ! = value) {
        CancelEvent e = new CancelEvent();
        onAlignmentChanging(e);
        if (!e.cancel) {
            alignment = value;
            invalidate(); // Repaint control when property changes
            onAlignmentChanged(Event.EMPTY);
        }
    }
  }
}
private EventHandler eAlignChanged = null;
public final void addOnAlignmentChanged(EventHandler handler) {
    eAlignChanged = (EventHandler)Delegate.combine(eAlignChanged,
        handler);
}
public final void removeOnAlignmentChanged(EventHandler handler) {
```

```
        eAlighChanged = (EventHandler)Delegate.remove(eAlignChanged, handler);
}
protected void onAlignmentChanged(Event event) {
        if(eAlignChanged ! = null){
            eAlignChanged.invoke(this, event);
        }
}
private CancelEventHandler eAlignChangeing = null;
public final void addOnAlignmentChanging(CancelEventHandler handler) {
        eAlignChanging = (CancelEventHandler)Delegate.combine(eAlignChanging, handler);
}
public final void removeOnAlignmentChanging(CancelEventHandler handler){
        eAlignChanging = (CancelEventHandler)Delegate.remove(eAlignChanging, handlder);
}
protected void onAlignmentChanging(CancelEvent event){
    if(eAlignChanging != null){
        eAlignChanging.invoke(this.event);
    }
}
public static EventInfo eiAlignChanged = new EventInfo(MyControl.class,
```

```
        "onAlignmentChanged", EventHandler.class);
    public static EventInfo eiAlignChanging = new EventInfo(MyControl.class,
        "onAlignmentChanging", EventHandler.class);
    public void getEvents(IEvents events){
        super.getEvents(events);
        events.add(eiAlignChanged);
        events.add(eiAlignChanging);
    }
```

# 自定义控件

除了定义控件成员外，可以通过自定义控件来定义它的的功能。

## 定义控件的显示

在控件中,可以写代码来确定和修改控件的可视表示。以下部分提供有关控件显示的背景信息以及如何管理的信息。

### 控件维数

与控件关联的有三个主要维数：边界，客户和显示坐标。控件的边界是外部窗口控件的坐标。边界总是在父客户坐标中表示，这就是说，与父客户的维数有关。

客户坐标是控件可以画进去的区域范围。客户控件区域总是以（0，0）为起点并且在属于控件区域的内部客户的基础上扩大。客户坐标不包含

非客户区域如应用程序到窗口的边界（例如，WS_BORDER,WS_EX_CLIENTEDGE）以及最上层窗口的标题栏。

显示坐标是实际定义子控件显示区域的坐标。显示坐标能够定义比客户坐标大的区域。在客户坐标和显示坐标之间不同的很好的例子就是带有自动滚动的窗体。当自动滚动在窗体中启用时，窗体的显示大小能够变得比客户的尺寸大，在实际窗体中的结果比物理窗口要大一些。当在窗体上移动滚动条时，显示坐标改变。

> **注意：**因为控件的边界总是在父客户坐标中，当窗体滚动时，子控件的边界将改变，以反映它们相对于父客户坐标的实际位置。

在多数控件中，用户只需要使用客户坐标。如果编写转到主应用程序其他控件（如 TabControl 控件）的控件，并且想要启用停靠或其他布局机制时，显示坐标是很重要的。

位置和大小

控件的布局是由控件从 Control 类中继承的几个属性的值组合确定的。

- 位置属性包含设置控件的左上角 x 和 y 坐标 Point 类。
- 大小属性包含设置控件的高度和宽度的 Point 类。
- 锚点属性固定一个和多个控件的边到它的容器中。当该容器重新设置大小时，锚定的控件边也重新设置大小。
- 停靠属性指定控件停靠在主包容器的哪边。

这些属性可以在运行时间中设置，来改变控件的位置和大小。

如果控件的位置或大小在运行时间改变，控件接着通知事件。当在控件上改变的任何事情会导致它们重新应用任何布局时，将激活布局事件。示例包括添加子控件、改变控件的边界线或执行一些其他的由控件指定的事件，如改变属性值等。重置大小事件只有在控件的边界改变时激活。默认的 WFC 布局逻辑是在 Control.onLayout 和 Form.onLayoutv 中处理。停靠和子控件的锚定应用在父的布局事件中。因此，一个面板将布局所有的子控件，并且面板的父将布局所有的面板。

## 更新可视显示

所有可视控件必须提供它自己的表现。除非控件是其他控件的子类，用户必须通过超越 onPaint 方法添加定制绘制逻辑到控件中。调用超类的 OnPaint 方法来显示控件，然后添加自己的逻辑来定制显示。
onPaint 事件接收可以用来得到 Graphics 对象接口的 PaintEvent 对象。然后调用 Graphics 对象的方法来更新控件的显示。下面的代码显示了如何在控件中通过更新文本属性来显示文本的一个简单的例子。

```
protected void onPaint(PaintEvent p) {
    super.onPaint(p);
   Graphics g = p.graphics;
   g.drawString(getText(), 0, 0);
}
```

WFC 中的 Graphics 对象功能非常丰富和完全。用户几乎可以绘制任何基本结构，包括弧形、椭圆、矩形、多边形、直线和点。下表列出了 Graphics

对象的常用属性和方法。

| Graphics 对象成员 | 说明 |
| --- | --- |
| setPen 方法 | 指定一个用来定义如何绘制直线和对象周围的边框 Pen 对象 |
| setBrush 方法 | 指定一个填充控件的 Brush 对象（例如，在 clearRect 方法中） |
| setBackColor 方法 | 指定在文本后面显示的颜色 |
| setTextColor 方法 | 指定用于文本的前景色 |
| setFont 方法 | 指定将绘制的文本的字体 |
| drawArc 方法 | 绘制一个椭圆弧 |
| drawImage 方法 | 绘制一幅图像 |
| drawString 方法 | 显示一个串：包括支持字环绕、对齐、剪切等等 |

在 WFC 中使用标准的 Win32 模型，在控件的区域中绘制被禁止。这些有效地请求重新绘制，但是不会立即执行绘制进程。在下一个空闲循环中，绘制事件异步地送到该控件。当事件到达该控件时，控件首先调用 eraseBackgroud 来清除要绘制的区域。绘制事件随后发生，并剪取到无效区域。

绘制事件是合并的，所以，对于控件的多个无效区域，控件将只接收一个绘制事件。控件接收的合并绘制事件剪取到所有无效区域的结合部分。

下例解释了用户如何调用使无效（invalidate）的方法来请求控件重新绘

制它自己。在这个实例中，调用使无效的方法不用用来指出剪取部分的Rectangle对象参数，这样，整个的控件将重新绘制：

```
public void setAlignment(int value) {
    if (!AlignStyle.valid(value))
       throw new WFCInvalidEnumException("vlaue", value,
             AlignStyle.class);
    align = value;
    invalidate();   // Repaint control when property changes
}
```

> **注意：**如果修改多个属性（或两次修改同一属性），控件将不两次执行整个绘制操作。

要强制绘制操作同步发生，用户可以调用控件的更新方法，该方法强制控件立即执行所有未决定的绘制事件。

### 在绘制时消除闪烁

要消除控件中的闪烁，用户应该考虑超越 onEraseBackground 事件。这些事件的默认实现是使用 backColor 属性的当前值清除控件的背景。但是，并不总是需要重新绘制控件的整个区域，并且进行这些不需要的操作时将会导致闪烁。当控件有大区域或是复杂的绘制逻辑时，通常会出现这种情况。

在这个例子中，drawString 方法用来在控件中放置文本。如果背景没有

被 eraseBackground 方法清除，那么它将看起来就像从没有被绘制一样，产生一个“透明的”显示。

> **注意：**控件并不是真正透明。但是，因为屏幕区域中以前的内容没有重新绘制，它们是始终可见的。

要在没有闪烁的情况下创建实心的控件，应该避免绘制将要使用前景信息重新绘制的区域的背景。这样操作的最简单途径就是要确保 onPaint 方法为整个的 clientRect 区域计数。在如下所示的例子中，用户可以通过超越 onEraseBackground 事件，指定该事件已被处理，从而完全地跳过背景绘制操作：

```
protected void onEraseBackground(EraseBackgroundEvent event) {
    event.handled = true;
}
protected void onPaint(PaintEvent p) {
    Graphics g = p.graphics;
    g.clearRect(getClientRect());
    g.drawString(getText(), 0, 0);
}
```

这是在 oNPaint 方法中带有简单代码的小例子。在这种情况下，用户可能看不到超越 onEraseBackground 事件的好处。但是，在自己的控件中，绘制的代码可能会更复杂，使用示例中的技术，会极大地减少闪烁。

其他添加控件可视稳定性的技术是双缓冲用户在屏幕上的图像。在该技

术中，用户维护整个客户区域的位图，并且随后创建一个基于图像的Graphics对象。利用缓冲区更新图像的性能比正常的要快。
只是在已经超越 onEraseBackground 并优化绘制代码时，及始终感到有闪烁时，用户才应该使用双缓冲区技术。双缓冲区技术是需要许多资源的操作，因为用户要维护控件图像的附加副本。对于大的图像，这样做会需要大量的内存（附加内存的大小取决于图像大小和它的色深）。如果缓冲区存储一个大的区域，维护缓冲区还会减缓执行速度。

要在控件中使用双缓冲区，进行下列操作：

- 指定正在处理的 onEraseBackground 事件。
- 超越 OnResize 方法来创建一个新的缓冲区来匹配新的客户区域。对老的缓冲区进行处理，使之在资源中释放也是一个好的方法。用户还可以调用使无效的方法；默认情况下，Windows 将只是使在重设大小时直接影响区域的部分无效。
- 在绘制代码中，从缓冲区中创建一个 Graphics 对象，并且在该对象上执行所有的绘图操作。用户必须明确为基于这些定义的 Graphics 对象设置 backColor 和 pen 属性。

下例在控件中的客户区域中绘制一个简单的星形图案。如果不包括一个双缓冲区，控件的闪烁会非常引人注目：

```
// Star.java
import com.ms.wfc.ui.*;
import com.ms.wfc.core.*;
```

```
public class Star extends Control
{

    // Create buffer
    Bitmap buffer = null;
    // Override onResize in order to recreate buffer at new size
    protected void onResize(Event e) {
        if (buffer != null) {
            buffer.dispose(); // Frees resources
            buffer = null;
        }
        Point s = getClientSize();
        buffer = new Bitmap(s.x, s.y);
        invalidate(); // Forces Windows to redraw entire control
        super.onResize(e);
    }
    protected void onEraseBackground(EraseBackgroundEvent event) {
    event.handled = true;
    }
protected void onPaint(PaintEvent pe) {
    Graphics g = buffer.getGraphics();
    Rectangle client = getClicentRect();
```

```
// Explicitly set backColor and pen based on buffer's values
g.setBackColor(getBackColor());
g.setPen(new Pen(getForeColor()));
g.clearRect(client);
int x=0;
int y=0;
int xCenter = client.width/2;
int yCenter = client.heihgt/2;
// Draws a star
for (; x<client.width; x+=4) {
    g.drawLine(x,y,xCenter,yCenter);
}
for (; y<client.height; y+=4) {
    g.drawLine(x, y, xCenter,yCenter);
}
for (; x>=0; x-=4) {
    g.drawLine(x, y, xCenter, yCenter);
}
for (; y>=0; y-=4) {
    g.drawLine(x, y, xCenter, yCenter);
}
g.dispose();
```

```
        pe.graphics.drawImage(buffer, 0, 0);
}
public static class ClassInfo extends Control.ClassInfo {
}
}
```

## 为控件添加位图

作为控件的可视部分，用户可以指定一个16×16像素的位图到该控件。当控件可以使用时，位图显示在Visual J++的工具箱中。控件的位图还为没有运行时表示的控件显示，如计时器。

> **注意：**只有当控件作为一个WFC控件使用时，才使用位图。如果作为ActiveX控件使用，控件在工具栏中显示一个标准的、预定义的位图。

默认情况下，ClassInfo对象加载具有相同文件名作为控件的任何位图。如果控件命名为MyControl.java，用户可以只需添加Mycontrol.bmp到MyControl.class文件所在的目录中，就可以将位图与它关联。
作为选择，用户可以使用超越在ClassInfo子类中的getToolboxBitmap方法来指定一个特定的位图。下面的代码显示了如何将位图Gears.bmp分配到控件中：

```
public Bitmap getToolboxBitmap() {
    return = new Bitmap(MyControl.class , "Gears.bmp");
```

}

用户所指定的位图大小不能超过16×16像素，并且应该使用16色或更少的颜色。Visual J++能够自动重新设置尺寸不正确的位图大小，但是这种图形的结果看起来并不太好。

## 创建控件定制器

定制器（customizer）是每个实例设计时间元数据的类型。ClassInfo类定义指定到类的元数据；相反，定制器提供了访问到更高级的设计事件功能。通过创建一个Customizer对象，用户可以添加功能到其控件中，如指定一个设计时间活动、添加控件动词或创建设计页（定制属性页）。作为带有值的编辑器，这里有一个默认的com.ms.wfc.core.Icustomizer——com.ms.wfc.core.Customizer的实现。用户可以超越在ClassInfo中的getCustomizer方法来返回一个定制器的实例。

### 指定设计时间激活

WFC支持控件的设计时间激活，如设计时间滚动或项目扩充，通过简单的击中测试方案。当选中控件时，击中测试请求将传递到控件的定制器中。如果击中测试返回值为真，将认为控件在该位置是激活的。利用这个简单的结构，控件区域的激活便是无缝的，如选项卡条的选项卡区域。

下例举例说明了一个简单的击中测试方法，该方法允许控件在鼠标移动过控件顶部的50像素时来接收消息。

注意：控件只能在它是主选中组件时被激活（并且 getHitTest 事件被激活）。

```
import com.ms.wfc.ui.*;
import com.ms.wfc.core.*;
public class MyTabControl extends Control {
    public static class ClassInfo extends Control. ClassInfo {
        public ICustomizer getCustomzier(Object comp) {
            return ((MyTabControl)comp). new Customizer();
        }
    }
     public class Customizer extends com.ms.wfc.core.Customizer {
        public boolean getHitTest(Point pt) {
            if (pt.y < 50)
               return true;
            }
            return false;
        }
    }
}
```

## 指定控件动词

动词可定义设计时间中能够在控件上执行的动作。在设计器中，动词通常显示在对应于该控件的快捷菜单上。

要创建一个动词，需细分（subclass）Customizer 类，然后创建一个 CustomizerVerb 对象，该对象允许用户指定动词中的文本，并且创建一个绑定动词的动作到指定方法上的委托。

下例说明了如何创建一个调用 About 的动词。About 类提供一个动词，在上下文菜单中为“About”，并且显示一个简单的消息框。用来显示 Action 动词结果的方法也包含在该类中：

```
// About.java
import com.ms.wfc.ui.*;
import com.ms.wfc.core.*;

public class About extends Control {
    public static class ClassInfo extends Control.ClassInfo
        public ICustomizer getCustomizer(Object comp) {
            return ((About)comp).new Customizer();
        }
    }
    public class Customizer extends com.ms.wfc.core.Customizer {
        public CustomizerVerb[] getVerbs() {
            CustomizerVerb v = new CustomizerVerb(About,
```

```
                new VerbExecuteEventHandler(About.this.showAbout));
            return new CustomizerVerb[] {v};
        }
    }
    private void showAbout(Object sender, VerbExecuteEvent event) {
        MessageBox.show("This control was written in WFC", "About",
            MessageBox.OK);
    }
}
```

对于数据和委托，用户还可以指定这些项选中和启用状态，并将位图与这些项关联。尽管通常 Visual J++将不显示在定制器中指定的位图（例如，在快捷菜单中），其他宿主可能支持该特性。

### 指定设计页

设计页是 WFC 属性页的等价物。设计页可以在设计时间环境中编辑。要执行一个设计页，用户创建扩展 DesignPage 类的类，并且超越 OnReadProperty 和 onWriteProperty 方法。

> **注意：**尽管 WFC 支持设计页，但是我们还是建议通过定制编辑器、动词和设计事件激活来为控件提供功能。有关细节请见本章前面的“创建定制属性值编辑器”部分。

下例说明了如何为对齐属性创建一个设计页。对齐属性（左对齐、居中、

右对齐）是由一个选项按钮组来执行的。在 onReadProperty 方法中，代码检查对齐属性，并且返回对象在设计页中设置的属性值。在 onWriteProperty 方法中，代码再次检查对齐，并在设计页中显示它的值。用于所有这些单选按钮的句柄中，调用 setDirty 方法，该方法标记该设计页为无效页，并且启用在 Properties 窗口中的 Apply 按钮。

```
// SuperLabelDP.java
import com.ms.wfc.core.*;
import com.ms.wfc.ui.*;
public class SuperLabelDp extends DesignPage {
    public SuperLabelDP() {
        initForm();
    }
private void setAlign(int value) {
   switch (value) {
       case AlignStyle.LEFT:
          radioButton1.setChecked(true);
          break;
    case AlignStyle.CENTER;
          radioButton2.setChecked(true);
          break;
    case AlignStyle.RIGHT:
```

```
            radioButton3.setChecked(true);
            break;
    }
}

private int getAlign(){
    int align = AlignStyle.LEFT;
    if (radioButton1.getChecked()) {
       align = AlignStyle.LEFT;
    }
    else if (radioButton2.getChecked()) {
       align = AlignStyle.CENTER;
    }
    else if (radioButton3.getChecked()) {
       align = AlignStyle.RIGHT;
    }
    return align;
}
private void radioClicked(object sender, Event e) {
    setDirty();
}
```

```
protected Object onReadProperty(String name) {
    if (name.equals("alignment")) {
        return new Integer(getAlign());
    }
    return null;
}
protected void onWriteProperty(String name, Object value) {
    if (name.equals("alignment") && value instanceof Integer) {
       setAlign(((Integer)value).intValue());
    }
}
Container components = new Container();
GroupBox groupBox1 = new GroupBox();
RadioButton radioButton1 = new RadioButton();
RadioButton radioButton2 = new RadioButton();
RadioButton radioButton3 = new RadioButton();
private void initForm() {
    this.setText("Alignment");
    this.setAutoScaleBaseSize(13);
    this.setBorderStyle(FormBorderStyle.NONE);
    this.setClientSize(new Point(307, 131));
```

```
this.setControlBox(false);
this.setMaxButton(false);
this.setMinButton(false);
  groupBox1.setLocation(new Point(8,8));
groupBox1.setSize(new Point(128,112);
groupBox1.setTabIndex(0);
groupBox1.setTabStop(false);
groupBox1.setText("Alignment");
  radioButton1.setLocation(new Point(8,16));
radioButton1.setSize(new Point(112,25));
radioButton1.setTabIndex(0);
radioButton1.setTabStop(true);
radioButton1.setText("Left");
radioButton1.setChecked(true);
radioButton1.addOnClick(new EventHandler(this.radioClicked));
  radioButton2.setLocation(new Point(8,48));
radioButton2.setSize(new Point(112,25));
radioButton2.setTabIndex(1);
radioButton2.setText("Center");
radioButton2.addOnClick(new EventHandler(this.radioClicked));
  radioButton3.setLocation(new Point(8,80));
```

```
        radioButton3.setSize(new Point(112,25));
        radioButton3.setTabIndex(2);
        radioButton3.setText(Right);
        radioButton3.addOnClick(new EventHandler(This.radioCicked));
          this.setNewControls(new Control [] {
            groupBox1});
        groupBox1.setNewControls(new Control[] {
            radioButton3,
            radioButton2,
        radioButton1});
    }
}
```

## 使用控件

当创建了 WFC 控件之后，用户可以在主环境中使用想要使用的任何控件。

### 注册控件

控件必须在它将要运行的计算机上注册。控件必须包括注册进程可以读取，用来标识该控件的信息，并且这些信息将放入该控件将要运行的计算机的 Windows 注册表中。通过阅读注册信息，应用程序可以找到并

加载控件。

> **注意**：如果在编译控件项目的计算机上使用这些控件，用户将不需要注册这些控件；编译进程已经自动注册了这些控件。如果正在Visual J++中使用这些控件，用户也不需要明确地包括注册信息。

主应用程序也需要一个用于控件的类型库（.tlb 文件），该库中包含了有关控件成员的信息。应用程序在类型库中读取这些信息，用来知道控件支持何种属性、事件和方法。Visual J++可以自动为在编译进程中的控件生成类型库。

### 指定注册和类型库信息

注册需要下面的信息：

- 类 ID（clsid），这是一个唯一标识控件的 GUID（全局唯一的 ID）。控件自身和类型库中需要一个类 ID。
- 程序 ID（progID），当创建一个控件的实例时（如MyProject.MyControl），在主应用程序中所使用的名称。

如果已经使用了 WFC Component Builder 来创建控件，类 ID 自动生成。还可以在编译进程中通过指定该控件是一个 COM DLL 来自动获得 Visual J++创建的类 ID。可以在项目的 Properties 窗口中的 COM Classes 选项卡上选择该选项。有关编译 DLL 的细节，见本书第十七章。

最后，用户还可以手工创建并包含控件的类 ID。如果想要保证为控件留下相同的类 ID，或者为一些其他的目的想要知道类 ID，并且不想让

Visual J++来生成，用户可能要这样操作。可以使用 Uuidgen.exe 这样的程序生成一个 GUID，该程序是一个免费的实用程序，在 Microsoft Web 节点上可用。用户总共必须提供两个类 ID，一个用于控件，而另一个用于它的类型库。
要将手工注册信息添加到注册程序中，用户应该将其包括在控件的.java 文件中的命令内嵌注释中，格式为：

```
/*
 * @com.register(clsid = guid, typelib = guid)
 */
```

例如，一个注册块应该看起来如下所示（在例子中出现了折行是因为 classID 太长，但是在你的文件中不要将其折行）：

```
/*
 * @com.register ( clsid=d0702fa0-fb3b-11dl-8f88-00aa00600a54 , typelib = d3108a20-fb3b-11d1-8f88-00aa00600a54)
 */
```

当为控件编译项目时，编译进程查找该块。如果找到，编译进程为该控件生成一个类型库。它也将该控件和类型库注册到计算机上。

## 创建 progID

注册进程自动为使用下列格式的控件建立一个 progID：

ProjcetName.ControlName

例如，控件项目可能为 MyProject，并且控件的.java 文件可能是 MyControl.java。当注册之后，控件的 progID 将为 MyProject.Mycontrol。

### 运行注册进程

如果想要在编译某个控件的计算机上使用该控件，则不需要来注册它——编译进程已经自动为用户注册了。但是，如果将该控件发布到其他计算机上，则必须在那里注册该控件。

如果已经将控件作为 COM DLL 编译，用户可以使用 Windows Regsvr32.exe 程序，像对其他控件一样来注册它。在 Command 窗口使用下列语法：

```
regsvr32.exe path/controlName
```

如果控件简单地编译为一个.class 文件，则使用一个名为 Vjreg.exe 的命令行实用程序，该程序包含在 Visual J++中（不能使用 regsvr32.exe 来注册该控件，因为该程序不是设计用来注册.class 文件的）。使用下面的语法在命令行中注册控件：

```
Vjreg path/controlName
```

当运行此命令时，它注册 Msjaval.dll 作为该控件的服务器，控件的名字和路径作为参数。

## 在主应用程序中使用控件工作

可以使用控件在主应用程序中工作，如可以使用任何想要使用的控件在

Visual J++，Visual Basic 和 Internet Explorer 中工作。要运行在 Visual J++ 中创建的任何控件，下面列出的东西必须在主计算机上存在：

- 使用 Visual J++ 6.0 可用的 Microsoft Virtual Machine for Java (VM)。可分配的 VM 是可以利用文件 Msjavx86.exe。VM 的这些版本与 Internet Explorer 3.02 和 4.0 兼容，但是有些控件特性可能在 3.02 下不可用。

另外，对于每个控件，用户必须分配下列文件：

- 在项目中用于所有类的 .class 文件。
- 在编译进程中为项目生成的类型库（.tlb 文件）。
- 在编译进程中为控件生成的 .dat 文件。
- 包含注册信息的 VJPROJS.SRG 文件。

> **注意**：如果正在编译该控件的计算机上运行这些控件，所有这些文件都可以使用。

当在主机上安装了这些文件之后，用户必须像在本章前面的“注册控件”一部分介绍的那样注册该控件。随后可以创建该控件的实例。在许多主应用程序中，用户可以将该控件添加到工具栏中。例如，在 Visual Basic 中，用户可以右击工具箱，选择 Components，然后选择该控件。

### 在 Internet Explorer 中使用控件

要在 Internet Explorer 中使用控件，需要创建一个 <OBJECT〉 元素。在 CLASSID 属性中，按照下面列出的几种方法之一识别控件：

- 按 progID。如下所示使用 CLASSID 属性：

```
<OBJECT CLASSID = "progid:myProject.MyControl">
</OBJECT>
```

- 按 ClassID。如下所示使用 CLASSID 属性：

```
<OBJECT CLASSID= " clsid:d0702fa0-fb3b-lldl-8f88-00aa00600a54:>
</OBJECT>
```

- 使用控件名。这种语法需要 Microsoft Virtual Machine for Java 带有 Visual J++ 6.0 或更高的版本。按下面列出的方法使用包括 JAVA 名字的 CLASSID 属性：

```
<OBJECT CLASSID = "JAVA:MyControl">
</OBJECT>
```

## 创建组合的WFC控件

用户可以使用 Windows Foundation Classes for Java(WFC)组件模型来创建控件的两种类型：定制（customize）和组合（composite）。
定制控件源于 com.ms.wfc.ui.control 类。用户可以在一个退出 WFC 控件的整个和子类中设计定制的控件。有关创建定制控件的更多信息，见在本章前面的“编写 WFC 控件”一部分。
组合控件是包括其他控件的控件。所有组合控件均源于

com.ms.wfc.ui.Usercontrol 类。因为 UserControl 类是 com.ms.wfc.ui.Form 类的子类，所以用户可以使用 Forms Designer 来布局定义组合控件的控件。

在本部分中要创建控件，用户需要学习如何：

- 使用 Forms Designer 来定义控件的布局。
- 添加使用 WFC Component Builder 的属性和事件。
- 为控件添加支持代码。

## 创建控件对象

Visual J++ Control 模板提供在创建 WFC 控件的领先技术。模板提供源于 com.ms.wfc.ui.UserControl 的类，并且包含源于 Usercontrol.ClassInfo 的 ClassInfo 类。

> **注意：** 在开始下面的进程之前，关闭所有打开的项目（在 File 菜单上，单击 Close All）。

**使用 Control 模板创建控件项目：**

1. 在 File（文件）菜单上，单击 New Project（新项目）。
2. 在 New（新建）选项卡上，扩展 Visual J++ Projects 文件夹，单击 Components，然后选择 Control（控件）图标。
3. 在 Name（名称）框中，为项目输入一个名称。
   在此时，输入 GroupCheck。

4. 在 Location（位置）框中，输入要存储项目的路径，或者单击 Browse（浏览）按钮来定位到某个目录。
5. 单击 Open（打开 ）。
   该项目的折叠视图出现在 Project Explorer 中。
6. 在 Project Explorer 中，扩展该项目节点。
   一个带有默认名 Control1.java 的文件添加到项目中。
7. 要将该该控件源文件名重命名为 GroupCheck.java，在 Project Explorer 中可以右击该文件名，然后单击 Rename（重命名）。

   **注意：**重命名该文件不会重命名在源代码中与之关系的类，反之亦然。用户必须手工改变所有老的名称的实例（用户可以创建一个空的项目，并且添加 Control 类到该项目中。然后在控件创建之前命名该控件）。

下一步是来设计控件的布局。

## 设计控件的布局

使用 Forms Designer 来设计组合控件的布局。该进程与设计窗体布局的进程是一样的。用户可以从工具箱中添加控件、在 UserControl 的接口上移动控件和重置控件的大小、指定属性和创建事件处理程序。
该方案示范了如何创建一个组合控件，该控件中包含 GroupBox 控件和 CheckBox 控件的组合控件，CheckBox 控件在 GroupBox 控件的左上角。当组合控件被公布到窗体上时，用户还可以在 GroupBox 控件的一部分

添加控件。当 CheckBox 控件未被选取时，则禁用组合控件中的该组件；但选取 CheckBox 控件时，将启用组合控件中的该控件。

**添加控件到 UserControl**

1. 在 Project Explorer 中双击 GroupCheck.java 在 Forms Designer 中打开该控件。
2. 在工具箱中，选择 WFC Control（WFC 控件）选项卡。
   如果工具箱没有显示，在 View（视图）菜单中单击 Toolbox。
3. 要添加 GroupBox 控件到 UserControl 中，可以在工具箱中单击 GroupBox 控件，然后单击 UserControl 设计表面。
   GroupBox 控件以默认的名字 groupBox1 添加。
4. 要添加 CheckBox 控件到 UserControl，可以在工具箱中单击 CheckBox 控件，然后拖动该控件移动到先前添加的 GroupBox 控件的左上角。
   CheckBox 控件以默认的名字 checkBox1 添加。确保 CheckBox 控件包含在 GroupBox 控件中。

要节省屏幕的空间，重新设置 UserControl 大小是重要的。因为 UserControl 可以重置大小，则当 UserControl 的大小改变时，重新设置在 UserControl 中的控件的大小也是很重要的。

**设置控件的大小**

1. 重新设置环绕在 GroupBox 和 CheckBox 控件周围 UserControl 的设计表面来去除附加的空间（要重置设计表面的大小，选择该接口，并且拖动设置大小柄）。

当 UserControl 设计表面添加到窗体时，它的大小是控件的默认大小。

2. 要允许 GroupBox 控件在 UserControl 重置大小时也重置大小，则在 Forms Designer 中选择 GroupBox 控件，然后选择锚定属性为 Top（顶部）、Left（左）、Right（右）和 Bottom（底部）。
   GroupBox 控件现在被锚定到 UserControl 的边界上。
3. 要将 GroupBox 控件的正确显示提供给用户，修改已经添加到 UserControl 中的 GroupBox 和 CheckBox 的属性。

**设置控件的属性**

1. 要删除 GroupBox 控件的默认标题，可以在 Properties 窗口中选择 text 属性，并且清除它的内容。
2. 要设置 CheckBox 控件被选取的默认状态，可以在 Forms Designer 中选择该控件，并且在 Properties 窗口中设置 checked 属性为 true。

下一步是添加属性到控件中。

## 使用 WFC Component Builder添加定制属性

使用 WFC Component Builder 来从控件中添加和删除定制属性。通过使用 WFC Component Builder，在为控件定义属性中，用户会有较高的起点。

**使用 WFC Component Builder 添加属性**

1. 在 Project Explorer 中，右击控件的源文件，然后单击 View Code（查

看代码）。

文本编辑器打开，并且显示控件的源文件。

2. 在 Class Outline 中，右击控件的类名称，并且单击 WFC Component Builder（要显示 Class Outline 视图，在 View 菜单上，将鼠标指针指向 Other Windows，然后单击 Document Outline）。
3. 在 WFC Component Builder 的 Properties 部分，单击 Add。
4. 在 Add WFC Property（添加 WFC 属性）对话框中，为 GroupBox 控件按照下表格定义 checked 属性，然后单击 OK。

| 字段 | 值 |
| --- | --- |
| Name（名称） | checked |
| Data Type（数据类型） | Boolean |
| Category（分类） | Behavior |
| Description（说明） | Determines whether the control's check box is checked（检测是否控件的复选框被选取） |
| Read-only Property（只读属性） | unchecked |
| Declare Member Variable（声明成员变量） | unchecked |

5. 在 WFC Component Builder 中，单击 OK。

WFC Component Builder 添加 getChecked 和 setChecked 方法到代码中，并且属性定义到控件的 ClassInfo 类中。

下一步是添加代码到 getChecked 和 setChecked 方法中。

## 添加代码到属性方法中

WFC Component Builder 创建方法和字段需要定义和执行定制的属性。一般是用户自己修改代码来提供自己的实现操作。

### 添加代码到属性方法中

1. 对于要返回 CheckBox 控件的选取状态的 getChecked 方法，使用下面的一行代码替换由 WFC Component Builder 添加的代码：

```
return checkBox1.getChecked();
```

2. 对于要设置 CheckBox 控件的选取状态，并需要调用其他方法的 setChecked 方法，使用下面的几行代码替换由 WFC Component Builder 添加的代码：

```
checkBox1.setChecked(value);
onCheckedChanged(Event.EMPTY);
enableControls(this, value);
```

这些代码基于传递到该属性的值来设置 CheckBox 控件的选取状态。该代码还调用 onCheckedChanged 方法，并且生成到 enableControls 方

法的调用。onCheckChanged 方法的调用用来切换定制事件 checkedChanged 中，该事件在后面将添加到该方案中。
Event.EMPTY 值传递到 onCheckedChanged，定义一个空的 Event 对象，该对象分配到 checkedChanged 事件中。enableControls 方法的调用用来启用或禁用添加到 GroupCheck 控件中的控件。

这些方法也将在后面添加到该方案中。

## 使用 WFC Component Builder 添加事件

使用 WFC Component Builder 在控件中添加和删除定制事件。用户可以因此避免在控件的 ClassInfo 类中手工定义事件。

### 使用 WFC Component Builder 添加事件

1. 在 Class Outline 中，右击控件的类名称，然后单击 WFC Component Builder（要显示 Class Outline，在 View 菜单上，将鼠标指针指向 Other Windows，然后单击 Document Outline）。
2. 在 WFC Component Builder 的 Event（事件）部分，单击 Add。
3. 在 Add WFC Event（添加 WFC 事件）对话框中，按下表为 GroupCheck 控件定义 checkedChanged 事件，然后单击 OK。

| 字段 | 值 |
| --- | --- |
| Name（名称） | checkedChanged |
| Type（类型） | Event（事件） |

| | |
|---|---|
| Category（种类） | Action（活动） |
| Description（说明） | 当事件的复选框选取状态改变时发生 |

4. 在 WFC Component Builder 中，单击 OK。
   WFC Component Builder 添加 addOnCheckedchanged 、removeOnChecked-Changed 和 onCheckedChanged 方法到代码中，并且在控件的 ClassInfo 类中添加一个事件定义。WFC Component Builder 还添加一个 EventHandler 委托的实例，该委托由事件使用。

下一步是超越继承方法。

## 超越 UserControl的方法

使用 Class Outline 可以很容易地超越在控件继承类中的方法。由 Class Outline 创建的超越方法提供了实现 Control 类中该方法的领先技术。用户可以在超越方法代码中使用由 Class Outline 提供的注释，用以快速地检测添加自己代码的位置。

### 使用 Class Outline 超越方法

1. 在 Class Outline 中，扩展类的节点。
2. 扩展 Inherited member 节点，然后右击想要超越的方法名。
   如果该方法可以被超越，则快捷菜单上显示 Override Method（超越方法）。
3. 单击 Override Method。

Class Outline 为指定的方法添加一个方法定义到用户的源代码中。

4. 对于 GroupCheck 控件，为 add、getControl、getControlCount、getControls、remove 和 setText 方法创建超越的方法。

下一步是添加代码到超越方法中。

## 添加代码到超越方法

当已经使用 Class Outline 创建了超越方法之后，用户将实现代码提供到方法定义中。保留或是删除到超类的调用取决于用户如何执行超越方法。

### 添加代码到 add 方法

要添加控件到 GroupCheck 控件的 GroupBox 控件上，用户可以添加代码到 GroupCheck 的 add 方法中，该方法调用 GroupBox 的 add 方法。这将导致代替 GroupCheck 控件的 GroupBox 控件成为被添加控件的父控件。

#### 添加代码到 add 方法

1. 在添加代码代 add 方法之前，用户应该添加一个私有成员变量到 GroupCheck 类中，用来检测是否控件可以添加。将下面列出的代码添加到 GroupCheck 类中：

```
private boolean m_bReady = false;
```

2.添加下面的代码替代 add 方法的定义：

```
if ( m_bReady ){
    control.setEnabled ( checkBox1.getChecked());
    groupBox1.add(control);
}
else
    super.add(control);
```

这些代码检测 m_bReady 成员变量是否设置为真。该检查是用来预防控件 GroupBox 控件添加到它自身中。如果 m_bReady 的值为真，该代码调用控件的 setEnabled 方法作为参数传递到该方法中。setEnabled 方法传递 GroupCheck 控件的 CheckBox 控件的复选框状态。因为当控件未被选取时，该控件可以添加到 GroupCheck 控件中，所以当添加控件时，该控件的启用或禁用的状态是非常重要的。

这些代码然后调用 GroupBox 控件的 add 方法，并且传递控件参数来将该控件添加到 GroupBox 控件中，用以代替 UserControl。如果 m_bReady 成员变量设置为假，则调用转到 add 方法的超类版，并且带有作为参数传递的控件。

## 添加代码到控件相关的方法

所以，用户可以访问在 GroupBox 中的控件，用户在 getControl、getControlCount 和 getControls 方法中提供代码，用来调用 GroupBox 控件的这些代码的实现。

**添加代码到控件关系的方法**

1. 在 getControl 方法的定义中，输入下列代码来替换由 Class Outline 添加的代码：

```
return groupBox1.getControl(index);
```

2. 在 getControlCount 方法的定义中，输入下列代码来替换由 Class Outline 添加的代码：

```
return groupBox1.getControlCount();
```

3. 在 getControls 方法的定义中，输入下列代码来替换由 Class Outline 添加的代码：

```
return groupBox1.getControls();
```

## 添加代码到 remove 方法

控件可以从 GroupCheck 控件中删除，用户在 remove 方法中为 GroupCheck 提供代码，用来调用 GroupBox 控件的 remove 方法。

**添加代码到 remove 方法**

- 在 remove 方法中，输入下列代码来替换由 Class Outline 已添加的代码：

```
if(m_bReady) {
    groupBox1.remove( c);
```

```
}
else {
    super.remove( c);
}
```

这些代码检测 m_bReady 变量是否为真。如果为真，该代码调用 GroupBox 控件的 remove 方法的版本，并带有作为参数传递到该方法的控件。用于 m_bReady

的复选框为真，则阻止 CheckBox 或 GroupBox 控件被删除。如果 m_bReady 为假，则代码调用超类方法的译本，用来验证控件完全删除。

## 添加代码到 setText 方法

GroupBox 控件不提供自动设置控件的文本部分，用来计算显示文本的数量的方法。超越 setText 方法来检测正确的基于文本大小的宽度来显示文本，使 GroupCheck 控件适当显示它的文本。

### 添加代码到 setText 方法

- 在 setText 方法的定义中，输入下面的代码来替换由 Class Outline 添加的代码：

```
Graphics g = checkBox1.createGraphics();
CheckBox1.setWidth(g.getTextSize(value).x + 20);
```

```
g.dispose();
checkBox1.setText(value);
super.setText(value);
```

这些代码使用 Graphics 类方法来检测指定的文本大小。当检测文本的大小后，它的值增长 20，用来补偿在复选框和 CheckBox 控件文本部分之间的空格和复选框的大小。该代码随后调用 Graphics 类的 dispose 方法，用来释放任何被分配的资源、设置 CheckBox 控件的文本属性，并且调用 setText 方法的超类版本。

下一步是添加新的方法到该控件中。

## 添加方法到该控件

当正在开发控件时，用户必须经常提供方法以执行控件中的动作。对于 GroupCheck 控件，用户添加一个方法，用来根据 CheckBox 控件禁用或启用包含在 GroupBox 控件中的控件。此方法从 setChecked 方法中调用。

### 添加方法控件

- 添加下面的方法定义到 GroupCheck 控件的源代码中：

```
public void enableControls(control start. Boolean enable) {
    for (int i = ; i < start.getControlCount(); i++) {
        Control c = start.getControl(i)
        if(c == groupBox1 || c == checkBox1) {
```

```
            continue;
        }
        c.setEnabled(enable);
        enableControls(c, enable);'
    }
}
```

enableControls 方法接收一个控件和一个布尔值，用来确定所包含的控件是否应该禁用或启用。当该方法由 setchecked 方法调用时，它传递当前 GroupCheck 控件的一个实例。

enableControls 方法开始是循环通过在 start 控件参数中包含的控件。在 for 循环中，代码使用带有 for 循环当前索引的 getControl 方法得到包含的控件。如果控件不是 GroupCheck 控件的 GroupBox 或 CheckBox 控件，该代码启用或禁用基于启用参数的值的指定控件。任何包含在这些控件中的控件随后通过 enableControls 的递归调用被启用或禁用。

下一步是添加代码到构造器中。

## 添加代码到构造器

要为控件提供初始设置，可以添加代码到构造器中。对于 GroupCheck 控件，用户添加控件添加到窗体时设置属性的代码。

### 添加代码到构造器

- 使用下面的代码替换在构造器中用于 GroupCheck 控件的代码：

```
super();

initForm();

setStyle(this.STYLE_ACCEPTSCHILDREN, true);
m_bReady = true
```

这些代码将 GroupCheck 控件样式设置为接收子控件。代码还将私有变量 m_bReady 的值设置为 true，这样，GroupCheck 控件的 add 方法知道该控件已经初始化完毕，并且可以接收添加到 GroupBox 控件中的控件。

下一步是编译该控件。

## 编译控件

要使用该控件，用户必须编译它。当控件编译后，用户可以将其添加到工具箱中。

### 编译控件

1. 在 Build（编译）菜单上，单击 Build。
   所有编辑错误或消息出现在 Task List 中（双击在 Task List 中的某个错误，可以将插入点移动到在 Text 编辑器中产生该错误的代码处）。

2. 改正错误并且重新编译控件。

下一步是调试控件。

## 调试控件

当编译控件之后，可以测试和调试该控件来确保该控件是否如期操作。要做到这些，应该添加控件到工具箱中，添加窗体到项目中，然后添加该控件到该窗体。对于 GroupCheck 方案来说，用户还可以添加控件到 GroupCheck 控件，并添加一个用于该控件的事件处理程序，随后编译并且运行该控件。

### 添加控件的工具箱

当编译控件之后，为了要使用该控件，应该将其添加到工具箱中。

#### 添加控件到工具箱

1. 右击工具箱，然后单击 Customize Toolbox（定制工具栏）。
2. 单击 WFC Controls（WFC 控件）选项卡，并且选择要添加的控件名。对于本方案来说，单击 GroupCheck 控件。
3. 单击 OK。

### 添加窗体到项目

要测试和调试控件，用户应该添加一个窗体到项目中。

**添加窗体到项目**

1. 在 Project Explorer 中，右击该项目名称，将鼠标指向 Add，然后单击 Add Form（添加窗体）。
2. 单击 Form 图标。
3. 在 Name 框中，输入用于该窗体的名称。
   要确保输入的窗体名称不是 GroupCheck，这样，该窗体的源文件将不会与用户控件的名称有冲突。
4. 单击 Open。
   带有指定名称的窗体添加到项目中，并且在 Forms Designer 中打开。

## 添加控件到窗体

用户添加控件到窗体中来测试该控件。

**添加控件到窗体**

1. 选择该窗体。
2. 在工具箱中，双击要添加的控件。
   控件添加到窗体的中间。

## 添加控件到 GroupCheck 控件

要确保 GroupCheck 控件完全成为添加到其中的控件的父控件，用户可以添加 WFC 控件到 GroupCheck 控件中。

**添加其他控件到 GroupCheck 控件**

1. 在该窗体上，选择 GroupCheck 控件。
2. 在工具箱中，双击某个控件将指定的控件添加到 GroupCheck 控件的中间。

## 创建事件处理程序

GroupCheck 控件包含一个定制的事件处理程序，名为 checkedchanged。当包含在 GroupCheck 控件中的 CheckBox 控件被选中或放弃选中时，该事件被触发。要确定是否该事件是否适当触发，用户可以添加一个用于 checkedChanged 事件的事件处理程序到窗体中。

### 添加用于 checkedChange 事件的事件处理程序

1. 在 Properties 窗口中，单击 Events 工具栏按钮。
2. 要显示 GroupCheck 控件的事件，在窗体上选择 GroupCheck 控件，或者在 Properties 窗口中选择 GroupCheck 控件的名称。
3. 双击 checkedChanged 事件来创建带有默认方法名的事件处理程序。Text 编辑器打开一个空的事件处理程序。

要确定 checkedChanged 事件是否触发，可以添加代码到该事件处理程序中，用来在每次事件触发时在窗体上显示一个消息框。

### 为事件处理程序添加代码

- 在该事件处理程序的定义中，添加下面的一行代码：

```
messageBox.show(" The checkedChanged event was triggered . ")
```

## 编译并测试控件

当已经将控件添加到窗体、并添加控件到 GroupCheck 控件中、及添加用于控件的 checkedChanged 事件的事件处理程序之后，用户可以编译并运行该项目。

### 编译和运行窗体

1. 在 Build 菜单上，单击 Build（如果收到任何编译错误或消息，改正错误并重新编译项目）。
2. 单击在 Debug 菜单上的 Start。
   因为用户正在第一次运行该项目，并且由于该项目有两个.java 文件，则显示 Project Properties 对话框。
3. 在 Launch 选项卡上，选择 Default 选项按钮。
4. 当项目运行时，指定 Form1 将加载。
   关于 Project 属性的详细信息请参见第一章的“Project 选项”。

当该项目运行时，用户可以操作该控件用来确定该控件是否操作正确。

### 在运行时间测试该控件

- 在 GroupCheck 控件中单击复选框。
  一个消息框显示出来，通知用户 checkedChanged 事件触发。每次当复选框的选取状态改变时，该事件发生。包含在 GroupCheck 控件中的控件的启用或禁用取决于该控件的选取状态。

有关将 WFC 控件作为 ActiveX 控件输出的更多信息，见本书的第 16 章中的“创建和导入 ActiveX 控件”。

# 第 14 章　在 Java 中编制动态 HTML

通过使用 Microsoft Internet Explorer 4.0，Microsoft 推出了一种新型的 HTML 对象模型，内容提供者可以用它随时有效地操作 HTML 对象的属性。现在，这个对象模型基本上已经可以通过使用脚本技术来访问了。利用 Windows Foundation Classes for Java（WFC）框架的 com.ms.wfc.html 软件包，可以从一个 Java 类中访问 Web 页中的 Dynamic HTML（DHTML）。

本章包含下面的主题：

- com.ms.wfc.html 软件包介绍
- 使用 initForm 方法
- 理解 DhElement 类
- 使用包容器
- 处理事件
- 使用动态样式
- 使用动态表
- 在服务器上使用 com.ms.wfc.html 软件包

# 快速开始

为了帮助用户通过使用 com.ms.wfc.html 软件包来开始运行 Java 和 DHTML，这里给出了创建简单的 DHTML 对象和添加动态行为的基本执行步骤。然后，这并不是本节的全部内容，它还设置本主题的其余部分和样本的步骤。当使用 com.ms.wfc.html 软件包时，有如下 5 个基本步骤：

1. 通过从 File 菜单中选择 New Project，然后从 Web Pages 子菜单中选择 Code-behind HTML 命令来创建新项目。
   这就产生了一个包含名为 Class1 的类的对象，该对象扩展 DhDocument。这些类描述动态的 HTML 文档。加入初始化代码到 initForm 方法中可以控制文档的内容和行为。
   通过做下面的工作，可以扩展文档的行为：
2. 创建新的元素（例如 DhButtom）或创建描述文档中（HTML 页上的）现有元素的元素对象。
3. 对一些元素挂上事件处理程序。
4. 在 Class.initForm 方法中，使用 setNewElements 方法添加新的元素，并且使用 setBoundElements 方法连接任何一个现有元素。
5. 编写在步骤 3 中挂上的事件处理方法。

所创建的文档看起来如下所示：

```
import com.ms.wfc.html.*;
```

```
import com.ms.wfc.core.*;
import com.ms.wfc.ui.*;
public class Class1 extends DhDocument
{
    public Class1()
    {
        initForm();
    }
    // Step 2: create objects to represent a new elements...
    DhButton newElem = new DhButton();
    // ... as well as elements that already exist in the HTML page.
    DhText existElem = new DhText();
    private void initForm( )
    {
        // Set properties to existing elements and newly added elements.
        newElem.setText("hello world");
        existElem.setBackColor(Color.BLUE);
        // Step 3: hook up an event handler to your object
        newElem.addOnClick(new EventHandler(this.onClickButton);
        // Step 2: create an object to represent an existing element
        existElem = new DhText();
```

```
        // Step 4: call setNewElements with an array of new elements
        setNewElements(new Component[] { newElem });
    }
        // Step 4: call bindNewElements with an array of existing elements
        setBoundElements(newDhElement[] { existElem.setBindID("Sample") });
    }
    // Step 5: implement your event handler
    private void onClickButton(Object sender, Event e) {
        existElem.setText("Hello,world");
    }
}
```

用来练习的 Java 部分已经完成。其他部分是 HTML 的代码。下例显示了由 Code-behind HTML 对象模式产生的 HTML 文档的简化版本。这里有两个连接 HTML 到用户项目中的代码的元素。

1. <OBJECT>标记加载 com.ms.wfc.html.dhModule 类，该类通过 Java 的 Virtual Machine for Java 实例化。
2. <OBJECT>标记带有一个名为 CODECLASS 的参数。参数的值就是扩展 DhDocument 的用户类的名称。

```
<HTML>
<BODY>
<OBJECT classid="java:com.ms.wfc.html.DhModule"
```

```
        height=0 width=0... VIEWASTEXT>
<PARAM NAME=CABBASE VALUE=MyProject>
<PARAM NAME=CODECLASS VALUE=Class1>
</OBJECT>

<span id=Sample></span>
<!--Insert your own HTML here-- >

</BODY>
</HTML>
```

打开 Internet Explorer 4.0，将其指向用户的 HTML 文件，然后可以看到应用程序的运行。

## 使用 initForm 方法

在 WFC 中用于所有用户界面编程的编程模型中，initForm 方法起着重要的作用。当使用基于 Win32 应用程序的 Visual J++ Forms Designer 时，在代表用户主窗体的 Form 派生类中可以找到 initForm 方法。在 com.ms.wfc.html 软件包中，该方法可以在 DhDocument-derived 类中找到（例如，Class1 在有 Visual J++提供的代码在后的 HTML 模板）并且它由类的构造器调用。

用户将使用 initForm 方法来初始化 Java 组件，该组件代表用户想要访问和加入代码的 HTML 元素。当在 Form-derived 类中有 initForm 方法，

并从 DhDocument 中的 initForm 方法中调用 WFC 方法时有一些限制。规则是，用户只能调用设置属性的在 initForm 中的方法。甚至，用户将使用 setBoundElement 方法来连接 HTML 页中的元素。

特别是，这就意味着，在 initForm 中不支持调用重新设置或删除属性或元素的任何方法。这条规则也同样适用于试图定位到已有 HTML 页上的任何方法（例如 DhDocument.findElement）。

这是因为直到调用了 dhDocument.onDocumentLoad 方法，已有的 HTML 页中的文档才与用户的 DhDocument 派生类合并。用户可以使用 onDoucmentLoad 方法来恢复属性，并且操作或定位在已有的 HTML 文档中的元素。在服务器端的类中使用 initForm 和 onDocumentLoad 方法的信息，见本章后面的“在服务器上使用 com.ms.wfc.html”。

## 理解 DhElement类

元素是源于 DhElement 的对象，是在 com.ms.wfc.html 软件包中所有用户界面元素的超类。当使用源于 DhElement 的任何对象时，会有某种一致性。

- 每个事件都有一个空的构造器。因此，用户能够使用一个新的语句实例化任何元素，然后一致设置属性、挂接事件处理程序和调用方法。
- 事件是非模态的。设置属性或调用方法总是以任何顺序进行；并且在一些外部的状态或环境中它是无条件的。
- 每个包容器都有一个带有最适合的、安全类型元素的添加方法。

- 在浏览器环境中，直到用户添加元素（或是在它的父系中最上层的包容器元素）到文档中时，才可以见到它。然而，这只是一个假象，不是编程模型的一部分。因为它们不管是否可视，都始终以同样的方法来工作，所以用户不需要改变编程元素的方法。

当调用 DhDocument.onDocumentLoad 方法时，如果某个元素已经在页面上，用户可以调用该文档的 findElement 方法，并且开始对该元素编程。用户还可以从 initForm 中调用 setBoundElements，并使用在 DhDocument 派生类中的元素来合并页上的已知元素（findElememt 方法有较好的性能，但是要求首先调用 onDocumentLoad）。
findElement 和 setBoundElements 使用的搜索规则假定用户想要绑定到有特殊名字的 ID 属性值的元素。使用 findElement，用户还可以列举所有在文档中的元素，直到发现自己感兴趣的某个元素为止。

## 使用包容器

包容器就是能够容纳其他元素的元素。基本的例子就是<DIV>元素，它可以包含任意的其他 HTML 项。更复杂的例子包括表单元格和文档本身。在多数情况下，包容器能够任意嵌套，例如某个表嵌套到其他表的单元格中。
包容器与其他元素相似。它们使用 new 语句来创建，并且多数能够在页中进行位置和大小的选择。用户可以在一个包容器中选择元素的位置和大小，并且可以设置它们的 Z-次序关系。DHTML 的一个最重要特性

就是用户可以在自己的代码中改变这些元素的属性。
当然，用户也可以使用常用的 HTML 布局规则来设置包容器中元素的位置，调用元素的 setLocation 方法或是调用 setBounds 方法来设置它的绝对位置，或者调用 resetLocation 来允许 HTML 布局引擎选择位置（但是必须放在 HTML 流程布局中最后一个元素的后面）。
一旦创建了一个包容器元素，用户就可以通过使用 SetNewElements 或添加方法来添加元素到该包容器中。这种结构遵循规则的父-子关系模式：元素可以是其他包容器，添加到成为它的子包容器的包容器中。事实上，直到最顶部，不是任何其他包容器的一部分的包容器添加到文档时，才有元素和文档相联。
用户可以通过使用包容器的 setBound 方法来选择包容器的位置和大小。例如，创建包容器的语句如下：

```
DhForm myform = dhForm();
```

然后用户就可以在包容器上设置各种属性，包括当鼠标在面板上盘旋时显示的工具提示。

```
MyForm.setToolTip("This text appears when the mouse hovers");
MyForm.setFont("Arial", 10);
MyForm.setBackColor(Color.RED);
MyForm.setBounds(5, 5, 100, 100);
```

最后，用户可以将刚刚创建的包容器加入到在 DhDocument 派生类（如 Class1.java）中的文档中：

this.add(myForm)

当加入元素到包容器中时，用户能够通过使用 com.ms.wfc.html 软件包中提供的常量集中的某个来指定元素在 Z 次序中的位置。添加元素时使用默认的大小和位置。用户可以调用元素上的 setBounds 来指定不同的大小。

```
DhForm myOverLay1 = new DhForm();
DhForm myOverLay2 = new DhForm();
myOverLay1.setBackColor(Color.BLACK);
myOverLay1.setBounds(10,10.50.50);
myOverLay2.setBackColor(Color.BLUE);
myOverLay2.setBounds(20,25,50.50);
myForm.add(myOverLay1.null,DhInsertOptions.BEGINNING);
// Black on top of blue
myForm.add(myOverLay2.myOverLay1,DhInsertOptions.BEFORE);
// Blue on top of black (uncomment below and comment above)
// myForm.add(myOverLay2.myOverLay1,DhInsertOptions.AFTER);
```

在元素加入之后，还可以使用 setZIndex 方法围绕着 Z 次序中来移动元素。例如，下面的语法没有清楚地在添加的元素上设置 Z 次序，但是使用了默认的 Z 次序（在所有其他元素的顶部）。

```
myform.add(myText);
```

用户可以清楚地设置该元素为下面列出的形式。这里的 num 是整数集，用来描述在它的包容器中的元素关联的 Z 次序。

```
myText.setZIndex(num);
```

带有最小数字的元素是在 Z 次序的底部（其他元素覆盖了它）。带有最大的数字的元素是在 Z 次序的顶部（该元素覆盖了其他元素）。

## 处理事件

DHTML 程序中的许多元素都能触发事件。com.ms.wfc.html 软件包使用与 com.ms.wfc.ui 软件包相同的事件模型。如果用户熟悉此种技巧，就可以发现这两种软件包之间的小小不同之处。按钮就是一个很好的例子。假定用户想要处理单击页面中某个按钮时产生的事件，下面列出的就是处理的代码：

```
public class Class1 extends DhDocument
{
    Class1() { initForm();}
    DhButton myButton = new DhButton();
    private void initForm()
    {
       add(myButton);
       myButton.addOnClick(new EventHandler(this.myButtonClick));
```

```
    }
    void myButtonClick(object sender,Event e)
    {
       ((DhButton) sender).setText("I've been clicked");
    }
}
```

在这些代码中，无论何时按钮触发了 onClick 事件（也就是当其被单击时），便调用 myButtonClick 事件处理程序。在本例中，mybuttonClick 事件处理程序中的代码发挥了很小的作用，它仅仅设置按钮上的标题为新的文本。

多数事件按着包含树的所有路径进行传播；这就意味着按钮的单击事件可以由按钮的包容器和其本身看到。尽管在程序员一般在距离事件最近的包容器中处理事件，这些事件的模型可能在一些特殊的情况中很有用。它提供给程序员灵活地决定事件处理程序代码的最佳位置。

许多不同的事件能够被 DHTML 中的事件触发，并且能够以相同的方式捕获它们。例如，要确定鼠标通过按钮的事件，试试下面的代码，这些代码捕获用于按钮的 mouseEnter 和 mouseLeave 事件：

```
public class Class1 extends DhDocument
{
    DhButton button = new DhButton();
    private void initForm()
```

```
    {
        button.addOnMouseEnter(new MouseEventHandler(this.buttonEnter));
        button.addOnMouseLeave(new MouseEventHandler(this.buttonExit);
        setNewElements( enw DhElement[] { button } );
    }
    void buttonEnter(object sender, MouseEvent e)
    {
        button.setText("I can feel that mouse");
    }
    void buttonExit(Object sender, MouseEvent e)
    {
        button.setText("button");
    }
}
```

所有能够触发（和捕获）的事件都在基于 com.ms.wfc.core.Event 事件类中定义。

## 使用动态样式

用户可以将 Style（样式）对象想象为独立的属性集合。样式这个术语取自字处理操作，在这里，样式表的编辑与用户的文档无关。在库中使用和应用 Style 对象是同样的。

例如，老板告诉你新的公司徽标颜色为红色，你需要改变在 HTML 页中元素的颜色。当然，可以在元素上直接设置属性，这是 GUI 框架编程的传统模型：

```
// old way of doing things...
DhText t1 = new DhText();
DhText t2 = new DhText();
t1.setColor( Color.RED );
t1.setFont( "arial");
t2.setColor( Color.RED );
t2. setFont( "arial");
```

当然，也可以使用派生的方式来节省时间。例如，可以考虑使用下面的代码来改善这个问题：

```
// old way of doing things a little better...
public class MyText extends DhText
{
    public MyText()
    {
        setColor( Color.RED );
        SetFont( "arial");
     }
```

直到决定想要这些设置来设置按钮、标签、选项卡和文档等，上面这些

代码都还适用。当在其他程序中或程序的其他部分中使用这些设置时，用户就会发现还有很多的事情要做。
这个问题的答案就是 Style 对象。当使用这些库时，用户可以实例化一个 Style 对象，并在任意位置设置它的属性：

```
// STEP 1: Create style objects.

DhStyle myStyle = new DhStyle();

// STEP 2: Set Properties on style objects

myStyle.setColor( Color.RED );
myStyle.setFont( "arial" );
```

然后，在编码的其他时候，可以将该样式应用到任意数量的元素上：

```
DhText t1 = new DhText();
DhText t2 = new DhText();

// STEP 3: Apply styles using the setStyle method.

t1.setStyle( myStyle );
t2.setStyle( myStyle );
```

当在公司中使用高级设置策略的动态特性的时候，下面的程序行使用在它们上面的 myStyle 集来设置所有元素的实例化，用来改变它们的颜色：

myStyle.setColor(Color.BLUE);

这里才是真正重要的部分：在运行时，所有这些都是有用的。每次改变 Style 对象时，DHTML 动态运行，并且改变所有的 Style 对象应用的所有元素。

有关更多的信息，见下一节“理解 Style 继承性”。

## 理解 Style 继承性

如果样式与设置在页面上的设置有冲突，则 HTML 重现引擎可以确定所要使用的样式。例如，如果元素直接设置了颜色属性（DhElement.setcolor）,那么就使用颜色属性定义的颜色。然而，如果一个元素上有 Style 对象（DhElement.setStyle），并且该对象有颜色属性设置，那么将使用该对象的颜色设置。如果找不到某种颜色或样式，将使用带有元素的包容器（DhElement.getParent）相同的过程，如果还是找不到，则使用该包容器的包容器的过程，以此类推。

此过程向上一直到文档。如果文档没有颜色属性设置，环境（浏览器设置或一些其他环境设置）决定要使用的颜色。

因为属性层叠向下到包容层次，所以，该过程称为层叠样式。在 W3C 中，用于 DhStyle 对象的基础机制称为 Style Sheets（CSS，样式表）。

## 使用动态表

使用表通常与使用库中的任何其他部分没有什么不同；应用到表的原则和编程模型同样也应用到任何其他类型的元素。然而，表是值得一提的元素，它重要而且常用。

要使用表，应该建立一个 DhTable 对象，给它添加 DhRow 对象，然后在行上添加 DhCell 对象。下面是表使用规则：

- 用户只能添加 Dhrow 对象到 DhTable 对象。
- 用户只能添加 DhCell 对象到 DhRow 对象。
- 用户能添加元素的任何类型到 DhCell 对象。

尽管这可能有些严格，但用户可以使用下面的代码轻松地创建一个包容器，来模仿 gridBag 类：

```
import com.ms.wfc.html.*;

public class GridBag extends DhTable
{
    int       cols;
    int       currCol;
    DhRow currRow;

     public GridBag(int cols)
    {
```

```
        this.cols = cols;
        this.currCol = cols;
    }
     public void add(DhElement e)
    {
        if ( ++this .currCol >= cols )
        {
           this.currRow = new DhRow();
           super.add(currRow);
           this.currCol = 0;
        }
        DhCell c=new DhCell();
        c.add(e);
        this.currRow.add( c );
    }
}
```

要使用 GridBag 类，用户只需设置行和列的数字（必须与此实现相同），然后指定元素到单元格。下面的代码是使用 GridBag 的 DhDocument 派生类中的代码：

```
protected void initForm()
{
```

```
    GridBag myTable = new GridBag(5);
    for (int i = 0; i < 25; ++i) {
       myTable.add(new DhText("" + i));
    setNewElements( new DhElement[] { myTable } );
     }
}
```

使用库最强的功能之一就是表和 Style 对象的结合。这使得用户可以创建定制的报表生成器，该生成器功能强大，具有专业性的外观，并且容易编写代码。

## 数据绑定到表

表也有数据绑定的能力。使用 com.ms.wfc.data.ui.DataSource 对象，用户可以绑定数据到表中，如下面的简单代码所示：

```
.
.
.
import com.ms.wfc.data.*;
import com.ms.wfc.data.ui.*;
.
.
.
void private initForm( ){
```

```
    DhTable dataTable = new DhTable();
    dataTable.setBorder( 1 );
    dataTable.setAutoHeader( true );
    DataSource dataSource = new DataSource();
    dataSource.setConnectionString("DSN=Northwind");
    dataSource.setCommandText("SELECT * FROM Products" );
    // if you would like to use the table on the server,
    // call dataSurce.getRecordset() to force the DataSource
    //to synchronously create the recordset; otherwise,
    // call dataSource.begin(), and the table will be populated
    // when the recordset is ready, asynchronously.
    if ( !getServerMode() ){
       dataSource.begin();
       dataTable.setDataSource( dataSource );
    } else
        dataTable.setDataSource( dataSource.getRecordset() );
    setNewElements(new DhElement[] { dataTable } );
}
```

如果知道将要返回的数据格式，也可以指定表将要使用的模板（重复器）行，用来格式化要返回的数据。进行这些操作的步骤如下所示：

1.创建 DhTable 元素：

```
DhTable dataTable = new DhTable();
```

2. 创建模板行，并且将其设置到表中；用户还可以有选择地创建标题行。对于在模板单元格中用户要从记录集中接收数据的每项，为其创建一个 DataBinding。

```
.
.
.
DhRow repeaterRow = new DhRow();
RepeaterRow.setBackColor( Color.LIGHTGRAY );
RepeaterRow.setForeColor( Color.BLACK );
DataBinding[] bindings = new DataBinding[3];
DhCell cell = new DhCell();
DataBinding[0] = new DataBinding( cell, "text", "ProductID" );
repeaterRow.add( cell );
cell = new DhCell();.
DataBinding[1] = new DataBindig( cell, "text", "ProductName" );
cell = new DhCell();.
cell.setForeColor( Color.RED );
cell.add(new DhText("$" ) );
DhText price= new DhText();
price.setFont( Font.ANSI_FIXED );
DataBinding[2] = new DataBinding( price, "text", "UnitPrice" );
cell.add( price );
```

```
repeaterRow.add( cell );

// set up the table repeater row and bindings
table.setRepeaterRow( repeaterRow );
table.setDataBindings ( bindings) ;
// create and set the hander row
DhRow headerRow = new DhRow();
headerRow.add( new DhCell("ProductID") );
headerRow.add( new DhCell("Product Name") );
headerRow.add( new DhCell("Unit price" ) );
table.setHeaderRow( headerRow );
```

3.创建 DataSource 对象，并且设置它以希望的格式来恢复数据。

```
DataSource ds = new DataSource();
ds.setConnectionString("DSN=Northwind");
ds.setcommandText("SELECT ProductID, ProductName,
        UnitPrice FROM Products WHERE UnitPrice < 10" );
```

4.设置 DataSource 对象到 DhTable 对象中。

```
table.setDataSource(ds)
ds.begin();
```

5.添加 DhTable 到文档中。

```
setNewElements(new DhElement[] {table})'
```

```
// alternately: add(table)
```

现在表从记录集中放入了数据，并且这些数据以模板行格式进行了格式化。

## 在服务器上使用com.ms.wfc.html软件包

com.ms.wfc.html 也能够在服务器上使用，用来提供一种编程模型，来产生 HTML，并且将其送到客户页上。服务器端不像客户端 Dynamic HTML 模型那样，由于服务器 Java 类与客户机的文档没有相互作用，所以服务器端的模型是静态的。然而，服务器由 HTML 元素组成，并且当元素在 HTML 模板中使用时指定了一个模板，则服务器将频繁地将它们发送到客户中。

尽管这不是完全的动态模型，它还是一个重要的服务器特性。例如，用户能够应用 DhStyle 属性到一些模板的 HTML 代码的所有部分中，然后通过使用改变 DhStyle 属性来产生不同外观的显示页。用户不必通过程序来产生所有的单独样式改变。另外一个优点就是，用户能够使用相同的模型来为客户和服务器应用程序来产生动态 HTML，因此，HTML 生成器易学易记。

当前在服务器上有两个产生 HTML 的模型。这两种类型都使用 Active Server Pages(ASP)脚本和基于 com.ms.wfc.html 类的类。第一种类型是“主干”的方法，它更多地依赖于 ASP 脚本。而第二种类型则使用从 DhDocument 中衍生的类，并且，因为此类型在类中比在脚本中公布更

多的控件，所以与在客户机上使用的模型非常相似。

## 基于ASP的方法

此种方法在服务器页中使用两个 ASP 方法：getObject 和 Response.Write。GetObject 方法用来实例化基于 com.ms.wfc.html 类的类。Response.Write 方法将已生成的 HTML 串写到客户。com.ms.wfc.html.DhElement 类提供 getHTML 方法，用来创建 HTML 串；这些串然后使用 ASP Response.Write 方法送到客户页中。

例如，用户有一个名为 MyServer 的类，用来扩展 DhForm，并且合并一些 HTML 元素。在 ASP 脚本中，首先调用 getObject(“java:MyServer”)来创建一个 DHTML 对象。用户然后可以执行 ASP 脚本对象上的任意动作，如在对象中设置属性等。当完成这些操作之后，就可以调用对象的 getHTML 方法来产生串，并且将此结果传送到 ASP Response.Write 方法中，该方法将 HTML 送到客户上。下面的代码段显示了有关的 ASP 脚本和用来在 HTML 中创建 DhEdit 控件并将其送到客户中的 Java 代码。

ASP SCRIPT

```
Dim f, x
set f = getObject(java:dhFactory )
set x= f.createEdit
x.setText("I’ m an edit! ")
```

```
Response.Write( x.getHTML() )
Response.Write( f.createBreak().getHTML() )
.
.
.
```

JAVA CODE

```
public class dhFactory {
    public dhFactory() { }
     public DhBreak createBreak() {
        return new DhBreak();
    }
     public DhEdit createDdit(){
        return new DhEdit();
    }
}
```

## 基于HTML的方法

此种方法稍微复杂一些，并且与客户模型更接近一些。它使用一个ASP脚本来设置DhDocument类的地点，但是操作代码的其他部分却是在Java中。就像在客户模型一样，DhModule类在Web页中实例化为Java组件，并且自动调用从DhDocument中衍生的项目类中的initForm方法。

就像在客户模型中一样，用户能够在 initForm 调用中做所有的绑定设置。onDocumentLoad 函数也被服务器端的类调用。在此方法中，用户能够访问 IIS Response 和 Request 对象（使用 DhModule getResponse 和 getRequest 方法），并且还添加新的 DhElement 项到用户的文档流中。然而，理解用户不能使用文档级的函数（如 findElement）这一点是很重要的，而且，在服务器端文档中不能使用枚举操作，除非项目已经明确添加到自己的 DhDocument 派生类中。

按下列步骤使用基于 HTML 的方法：

1. 创建服务器 Java 类（从 DhDocument 中扩展）。
2. 在该类中，当想使用客户应用程序时执行 initForm 方法。
3. 从 ASP 中，调用 Server.CreateObject 方法。传递“DhModule”来创建 DhModule 对象。
4. 调用 DhModule.setCodeClass 方法，传递给它 Dhdocument-derived 类的名称。
5. 调用 DhModule.setHTMLDocument 方法，如果有该方法，则将服务器 Web 页的完全本地文件名传递给它。

   如果使用一个空串（""）调用 setHTMLDocument，那么 DhDocument 类运行，并且为在用户的 ASP 中调用 SetHTMLDocument 的位置处已添加的任意元素输出 HTML。然后用户就可以生成内嵌的 HTML 代码段。

   如果用户没有调用 setHTMLDocument，那么 DhDocument 类为该页输出完全的 HTML，并且包含<HTML>、<HEAD>和<BODY>标记。

下例显示了使用模板的 ASP 页：

```
<% Set mod = Server.CreateObject( "com.ms.wfc.html.DhModule" )
    mod.setCodeClass( "Class1" )
    mod.setHTMLDocument( "c:\inetpub\wwwroot\Page1.htm" )
%>
```

在运行期间，此结构能够识别在服务器用户类的运行和相应的动作。一旦实例化，用户就可以对 DhDocument 衍生类添加元素或文本。那些项目就可以添加到在</BODY>标记前指定的任意模板中。
下例说明了既工作在客户上，也工作在服务器上的类：

```
import com.ms.wfc.ui.*;
import com.ms.wfc.html.*;

public class Class1 extends DhDocument {
    public Class1(){
        initForm();
    }
    DhText txt1 = new DhText();
    DhForm sect = new DhForm();

     private void initForm() {

          // call getServerMode() to check
        // if this object is running on the server
```

```
        if ( getServerMode() ) {
          txt1.setText("Hello from the server!" );
        }else{
          txt1.setText("Hello from the client!" );
        }
          // size the section, set its background color
        // and add the txt1 element to it
        sect.setSize( 100, 100 );
        sect.setBackColor( Color.RED );
        sect.add( txt1 );
        add ( sect );
        setNewElements ( new DhElement[] {sect } );
    }
}
```

如果用户想要绑定到在该页上已有的 HTML 文档中，就像在客户上工作一样，使用 DhDocument.setBoundElement 方法。例如，如果 HTML 模板包含下面的 HTML：

```
<P>
The time is:<SPAN id=txt1></SPAN><BR>
<INPUT type=text id=edit1 value="">
</P>
```

用户的 initForm 方法应该如下所示：

```
DhText txt1 = new DhText();
DhEdit edit = new DhEdit();
DhComboBox cb = new DhComboBox();
private void initForm(){
    txt1.setText(com.ms.wfc.app.Time().formatShortTime());
     edit.setText("Hello,world!");
    edit.setBackColor(Color.RED );
     setBoundElements( new DhElement[]{ txt1.setBindID("txt1");
                                        edit.setBindID("edit1" ) } );
     // Create a combo box to be added after the bound items
    cb.addItem("One");
    cb.addItem("Two");
     // Add the items to the end of document.
    setNewElements( new DhElement[]{ cb } );
}
```

在客户和服务器上 HTML 类的执行几乎一样。然而，也有一个很重要的不同点。一旦元素写入（发送到客户），它们就不能像修改客户文档一样进行修改。如果企图再次修改此元素，在写入操作已经在元素上执行之后，会出现 DhCantModifyElement 只与服务器应用程序相关的异常

（这着重说明了在服务器 Java 类和客户文档之间没有实际互操作的事实，就像在客户上的 Java 类和文档之间一样）。

使用 DhDocument 派生方法的一个有利之处，就是能够实现用 com.ms.wfc.html 类识别的属性嵌入的 HTML 模板。首先，通过在文档中使用 ID 属性来装饰 HTML 元素，然后在源代码中使用 DhElement.setBindID 方法来设置相应的 ID，用户能够绑定到这些 HTML 元素中、在元素中设置属性、添加用户自己的 HTML 代码等。实际上，这就是允许用户提前编码和设计单独的模板，并在服务器请求文档时使用动态数据来传播模板。

# 第 15 章　图形服务器

通过图形设备接口（GDI），可以在 Microsoft Windows 中显示 Graphic 对象。它是一个与设备无关的图形输出模型，处理基于 Windows 应用程序的图形功能调用，并把这些调用传到合适的设备驱动程序。这些驱动程序在输出时执行硬件特定的函数。当在设备相关格式与设备相互作用时，GDI 在应用程序和输出设备之间充当一个缓冲区，对应用程序提供与设备无关的视图。

应用程序开发人员使用 GDI 的功能可以显示图像，绘制控件、形状和文本，创建和使用笔、刷子和字体。Windows Foundation Classes（WFC）Graphic 对象与其他 WFC 对象协调，诸如笔、文字和刷新对象等，将这些能力封装为基于 Java 的对象。

在 WFC 环境中，可通过 Graphic 对象输出图形。在创建或检索一个 Graphic 对象之后，可以关联其他基于图形的对象，诸如带对象的字体、笔和刷子，然后使用对象的各种绘制方法来重现显示输出。例如，要画具有特殊外观的线，可以使用 Graphic 对象的 setPen 方法来指定一个笔，对象将使用此笔来绘制，然后使用对象的 drawLine 方法来重现线条。还可以根据需要多次修改这些关联。

## 创建Graphic对象

WFC提供了几种方法来创建一个Graphic对象。
显式Graphic对象的创建：可以在任何可扩充Control类的对象上，通过调用creatGraphics方法，来创建一个显式的Graphic对象。
隐式显示Graphic对象的创建：Bitmap和Metafile对象通过getGraphics方法支持隐式Graphic对象的创建。

## 显式Graphic对象的创建

扩充Control类的所有类都支持createGraphics方法，可以使用它来创建一个Graphic对象实例。下面的程序段演示了如何从一个Form派生类中调用这种方法。

```
Graphics g = this.createGraphics();
```

显式Graphic对象创建的第二种方法是使用Win32设备场境句柄（HDC）。通常，当调用一个Win32方法返回HDC时，可以使用这种方式创建一个Graphic对象。
在Graphic对象中，如果只需要本机不支持的图形能力，可以使用对象的getHandle的方法来检索Win32设备场境的句柄，并且可以对适当的Win32方法透明传送句柄。
如果在以前的句柄上创建一个Graphic对象，则此对象假定不具有句柄的所有权。使用了此对象后，要负责使用适当的Win32函数释放句柄。

如果使用 Graphic 对象的 getHandle 方法来检索对象的基本句柄，对象保留句柄的所有权，就不应该释放它。
关于 Win32 句柄和 Graphic 对象的更多信息，请参看下一章的“执行基于句柄的操作”。

## 隐式 Graphic对象的创建

通过 getGraphics 方法，Bitmap 和 Metafile 对象支持隐式 Graphic 对象的创建，在下面的代码段中，使用 Bitmap 对象的 getGraphics 方法来创建一个 Graphic 对象，使用此对象可以对位图的表面画图：

```
Bitmap bmp = new Bitmap("c:\\MyImage.bmp");
Graphics gr = bmp.getGraphics();
```

最初，可以通过对象调用 getGraphics，此对象创建一个新的 Graphic 对象实例并返回它。通过原始的调用，在所建对象的回送中，通过相同对象结果建立 getGraphics 的后续调用。
另外，当通过采用对象的 getGraphics 方法创建的 Graphic 对象调用绘制方法时，此方法应用于通过它们创建的对象。例如，对 drawText 在调用中指定的 X、Y 的坐标（0，0）是与位图相关的，但不能控制位图重现。

## 检索Graphic对象

在窗体或控件的绘制事件中，可以检索和使用 Graphic 对象实例。绘制事件处理程序把 PaintEvent 对象作为一个参数，并且此对象包含了公共的图形成员。此成员是一个有效 Graphic 对象实例，可以如下调用：

```
protected void onPaint(PaintEvent e){
    e.graphics.drawString("Hello, world", new Point(10, 10));
}
```

关于如何创建绘制事件外的 Graphic 对象，请参考以前的章节“显式图形的创建”和“隐式图形的创建”。

## Graphic对象作用域

Graphic 对象有方法作用域。这意味着当使用 Graphic 对象返回某种方法时，自动调用此对象的清理方法，释放所有已分配给对象的资源。调用了清理之后，可尝试使用异常运行期间的对象结果。
如果在类级声明一个 Graphic 对象的实例，应使用 Form 对象的 createGraphics 方法来初始化每个使用它的方法中的对象。

```
Public class Form1 extends Form{
    Graphics g = new Graphics();
```

```
    private void Form1_resize(Object sender, Event e){
        // initialize object instance.
        g = this.createGraphics();
            // dispose automatically called...
    }
    private void Form1_click(Object sender, Event e){
        // Initialize object instance.
            g = this.createGraphics();
        // dispose method automatically called...
    }
}
```

尽管清理方法的调用是自动的，但在例程的最后，最好是使用 Graphic 对象显式调用它。在 Windows 95 平台上，这是非常重要的，因为系统分配的设备上下文数量是有限的。

## 维护边框矩形

可以画图的窗口区是窗口的客户区。在这个矩形区域，边框矩形定义了

在画 Graphic 对象中可视的矩形区域，并且可以包括窗口的整个客户区。当一个窗口丢失了，然后又恢复焦点时，则边框矩形部分被以前重现它的其他不自动重新显示的对象所覆盖。
为确保正确的显示，必须管理窗体或控件的重新绘制。绘制事件处理程序是放于进行管理的 Form 类代码中。在处理程序中，可以恢复边框矩形到正确的状态。
下面的例子将创建类级 Bitmap 对象，然后使用绘制事件处理程序来重画位图。窗体的客户区每次变成无效时，Windows 都调用这个处理程序，并且，图像重新绘制到窗体。

```
Bitmap bmp = new Bitmap("c:\\MyImage.bmp");
Protected void onPaint(PaintEvent e){
    e.graphics.drawImage(bmp, new Point(0, 0));
}
```

当第一次显示窗体时，每次都重新获得焦点，自动调用绘制事件处理程序。然而，如果窗体支持重设尺寸，则窗体维数的改变不会自动触发重新绘制。相反，必须添加对 Form 派生类的重设尺寸处理程序，然后，从处理程序中，调用使对象无效的方法。下面是在窗体绘制事件处理程序中调用使无效的方法：

```
protected void onResize(Event e){
    this.invalidate();
}
```

## 执行基于句柄的操作

句柄是操作系统中一些元素的唯一编号。例如，桌面上的每个窗口都有一个唯一的句柄，以使计算机能够把它与其他窗口区别开。每个设备上下文、刷子、笔和字体也都有一个唯一的句柄。

为了与 Win32 AIP 兼容，Graphic、Brush、Pen 和 Font 对象都支持基于句柄的操作。例如，如果使用 Win32 CreatPen 函数创建了一个刷子，此函数就会返回一个句柄。可以把这个句柄传给 Brush 对象构造器，而且，根据 Win32 笔的特征创建这个对象。

即使没有以句柄为基础创建 Brush、Pen 或 Font 对象，也可以检索基于 Brush, Pen 或 Font 对象的句柄。每个对象都支持 copyHandle 和 getHandle 方法，一旦使用这些方法复制或检索对象的句柄，就可以传送这个句柄给任意需要把句柄作为参数的 Win32 函数。

当使用 Graphic 对象执行基于句柄的操作时，应记住以下几条重要规则：

- 如果使用 Win32 方法创建基于以前分配过句柄的对象，对象不会假定具有句柄的所有权。例如，如果使用 Win32 GetDC 方法来检索窗体的设备上下文句柄（HDC），然后创建基于 HDC 的 Graphic 对象，则就拥有了句柄。这意味着，当调用 Graphic 对象的清理方法时，句柄在内存中不释放。相反，必须使用适当的 Win32 例程来清理句柄。下例说明了适当的句柄管理。本例使用 Win32 GetDC 来检索窗体设备上下文的句柄，然后创建一个基于此句柄的 Graphic 对象。使用 Graphic 对象在窗体上画了一条线以后，Win32 ReleaseDC 例程通过

GetDC 来释放已分配的设备上下文句柄：

```
// import the com.ms.wfc.Win32.Windows package.
Import com.ms.wfc.Win32.Windows;

int hDC = Windows.GetDC(this.getHandle());
Graphics g = new Graphics(hDC);

g.drawLine(new Point(0,0), new Point(100, 0));
g.dispose();

Windows.ReleaseDC(hDC);
```

注意，本例中使用设备上下文的句柄，这里说明的原则可用于笔、刷子、字体和位图。

- 多数 Graphic 对象支持 copyHandle 方法，通过这种方法，可以复制对象的句柄。如果使用 copyHandle 方法复制一个对象的句柄，则要负责释放句柄。
- 多数图形对象也支持 getHandle 方法，这种方法返回对象的句柄（与句柄的副本相反）。这种方法支持 Graphic 对象方法和 Win32 图形例程之间的兼容性。
  通过 getHandle 方法检索的句柄不能是对象基本句柄的副本。因此，不应该通过 getHandle 释放检索的句柄。当清理对象时可释放这些句柄。

# Graphic对象坐标系统

Graphic 对象支持的许多方法都依赖于数字坐标。此坐标可在 Graphic 对象中指定，在 Graphic 对象中，指定操作发生的区域，或者在 Point 对象中，指定操作发生的 X（水平）和 Y（垂直）坐标。
坐标系统指明了这样的对象指定的坐标如何映射到显示器或其他设备中。例如，假定调用 Graphic 对象的 drawString 方法在坐标 100，100 画文本：

```
Graphics g = this.createGraphics();
g.drawString("Hello, WFC, new Point(100, 100));"
```

本例中，Point 对象指定了画一字符串的 X 和 Y 坐标。然而，这次操作的实际结果取决于 Graphic 对象关联的坐标系统。
关联 Graphic 对象的坐标系统是在 CoordinateSystem 类中定义的。Graphic 对象的默认坐标系统是 CoordinateSystem.TEXT，这意味着，在 Point 对象的 X 和 Y 值增加时，文本（或位图或控件）将水平向右或垂直向下移动。
为了同带有 Graphic 对象的坐标系统关联，使用 setCoordinateSystem 方法如下：

```
Graphics gr = this.createGraphics();
Gr.setCoordinateSystem(CoordinateSystem.ANISOTROPIC);
```

下表列出了与 Graphic 对象关联的坐标系统，并描述了在绘制过程中，Point 对象中定义的 X 和 Y 值增加时绘制的方向如何。

| 坐标系统 | X 轴增长方向 | Y 轴增长方向 |
|---|---|---|
| CoordinateSystem.TEXT | 右 | 下 |
| CoordinateSystem.LOMETRIC | 右 | 上 |
| CoordinateSystem.HIMETRIC | 右 | 上 |
| CoordinateSystem.LOENGLISH | 右 | 上 |
| CoordinateSystem.HIENGLISH | 右 | 上 |
| CoordinateSystem.TWIPS | 右 | 上 |
| CoordinateSystem.ISOTROPIC | 用户定义 | 用户定义 |
| CoordinateSystem.ANISOTROPIC | 用户定义 | 用户定义 |

## 设置坐标原点

坐标原点定义了测量坐标的起始位置。

例如，如果通过应用程序的主窗体创建一个 Graphic 对象，则 Graphic 对象的坐标系统是 CoordinateSystem.TEXT 且图形坐标是从窗体的左上角（0，0）开始的。

然而，如果使用用户定义的坐标系统，诸如 CoordinateSystem.ANISOTROPIC 或 CoordinateSystem.ISOTROPIC，可以使用 Graphic 对象的 SetCoordinateOrigin 方法，定义坐标开始测量的点（对页面或设备）。例如，假设在应用程序的绘制事件中包括如下代

码：

```
private void Form1_paint(Object source, PaintEvent e)
{
// Set the coordinate system.

e.graphics.setCoordinateSystem(CoordinateSystem.ANISOTROPIC);
e.graphics.setCoordinateOrigin(new Point(20, 20, new Point(20,20));
e.graphics.setCoordinateScale(new Point(1,1), new Point(1,1));
e.graphics.drawLine(new Point(0,0), new Point(100,100));

}
```

代码设置坐标系统为 CoordinateSystem.ANISOTROPIC，并设置坐标原点为页面或设备的 20，20 坐标。在代码中以后出现的对 drawLine 的调用是从原点开始的。于是，直线将从坐标原点（10，10）开始到 100，100 结束。

## 映射逻辑坐标到设备坐标

Windows 对设备的映射页面坐标提供内置式支持，诸如打印机或显示器。

对于大多数通过 Windows 定义的坐标系统，由系统本身执行这种转换。例如，如果使用 CoordinateSystem.HIMETRIC 坐标系统关联 Graphic 对象，则对象上的每个逻辑单位在设备上转换为.001 英寸。

然而，使用用户定义的 ANISOTROPIC 和 ISOTORPIC 坐标系统，则必须使用 SetCoordinateScale 方法告诉系统如何执行这种转换，如下：

```
Graphics g = this.createGraphics();
g.setCoordinateSystem(CoordinateSystem.ANISOTROPIC);
g.setCoordinateScale(new Point(1,1), new Point(2,2));
```

在本例中，对 setCoordinateScale 的调用指示系统把一个逻辑水平或垂直单位转换为两个水平和垂直的设备单位。

## 绘制文本

Graphic 对象的 drawString 方法支持对象所属控件的文本输出。这个方法编写文本到指定的位置，如下：

```
Graphics g = this.createGraphics();
g.drawString("Hello, World", new Point(0, 0));
```

这次调用的可视结果依赖于诸多因素，包括 Graphic 对象目前的文本颜色、背景色、字体位置和坐标系统。
关于如何设置文本颜色的信息，参看下一节“设置文本颜色”。Graphic 对象与字体关联的信息，参看本章后面的“使用字体对象”。关于坐标系统的信息，参看本章后面的“Graphic 对象坐标系统”。

## 设置文本颜色

为设置文本中文本和背景的颜色，可在调用 drawString 之前调用 Graphic 对象的 setBackcolor 和 setTextColor：

```
// Set the text to white, the background to black.

g.setTextColor(new Color(0, 0, 0));

g.setBackColor(new Color(255, 255, 255));

// Draw the text...
```

setBackColor 方法只影响文本的背景色。为了设置背景或为其他对象填充颜色，如多边形或线，分别使用 setBrush 和 setPen 方法。

## 使用 Font对象

字体是共享共同设计的字符和符号的集合。字体的主要元素包括字样、效果和尺寸。
在 WFC 中，在字体对象中，Windows 字体是封装的。可以将对象与其他字体相关的对象组合使用，为了在 Graphic 对象上显示字体，定义一个可无限变化的字体。
下表列出了 WFC 支持的相关字体类别。

| 类 | 说明 |
| --- | --- |
| FontDescriptor | 提供有关 Font 对象的信息 |
| FontFamily | 定义一组常量，表示字体可能属于的系列 |
| FontMetrics | 定义字体映射到 Graphics 对象时的物理特性 |
| FontPitch | 定义字体的间距 |
| FontSize | 定义一组常量，在能被指定的字体大小单元中表示 |
| FontType | 定义一组常量，这些常量表示将重现字体的设备 |
| FontWeight | 定义常量，这个常量表示与字体关联的不同权重（weight） |

## 创建Font对象

创建 Font 对象可能非常简单，也可能非常复杂，这主要取决于所要定义的字体的粒度。下面的代码说明了一种创建 Font 对象的简单方法。

```
Font font = new Font("Times New Roman", 26);
```

这个例子使用了 Font 对象构造器，这只需要知道两项信息：字体名和字体尺寸。当创建 Font 对象时，也可以指定字体系列、类型、权重、方向和是否加粗、是否斜体、是否有下划线或是否有删除线。然而，一旦创建了一个 Font 对象，就不能再修改这个对象的属性。

## 在Graphic对象上设置字体

Graphic对象支持setFont方法，此方法与带有对象的字体相关联。当Font对象与Graphic对象相关联后，在Graphic对象矩形框内的所有文本都使用此字体。

下面的代码说明了把Graphic对象的背景色设置为白色，文本颜色设置为黑色，字体为26磅Times New Roman的过程。文本显示在指定的位置上：

```
Graphics g = this.createGraphics();
g.setFont(new Font("Times New Roman", 26));
g.setBackColor(new Color(255, 255, 255));
g.setTextColor(New Color(0, 0, 0));
g.drawString("Hello, World", 0 0);
```

## 枚举字体

在许多实例中，必须能枚举且检索可用字体的详细信息，并为某一特殊操作选择最合适的字体。

WFC FontDescriptor对象描述了字体，包括字体名、高度、方向等等。系统中所有可用字体的详细描述可使用Graphics对象的getFontDescriptor方法获得。这种方法返回FontDescriptor对象的一个数组，数组中的每个元素描述一种字体。

下面举例说明了如何使用 getFontDescriptor 方法。此例检索了可用字体的数组，然后，把字体名的唯一表列插入到列表框。

```
Graphics g = this.createGraphics();

// Create the array.
FontDescriptor rgFonts[] = g.getFontDescriptors();

for(int i = 0; i < rgFonts.length; i++){

    if(listBox1.findString(rgFonts[i].fullName == -1){

       ListBox1.addItem(rgFonts[i].fullName);
    }
}
```

## 使用笔

笔是一种图形工具，在 Microsoft Windows 中，用于绘制直线和曲线。笔可用于画徒手线。计算机辅助设计（CAD）应用程序使用笔绘制可见线、隐含线、剖面线、中心线等等。字处理和桌面出版应用程序使用笔绘制边框和标尺。电子数据表应用程序使用笔可在图中指明趋势，并且绘制条形图和饼图。

每个笔由三个属性构成：效果、宽度和颜色。如果对笔的宽度和颜色没有特殊限制，则操作系统必须支持笔的效果。

## WFC Pen对象

在 WFC Pen 和 PenStyle 对象中，Win32 笔的功能是封装的。以下代码段演示了如何创建 Pen 对象：

```
Pen p = new Pen(PenStyle.DASH);
```

笔效果是传递到构造器的常量。在 Windows 中支持七个内部笔效果，每一个都可以通过在 PenStyle 类中定义一个常量来表示。PenStyle 类是枚举类，这意味着它定义了一种方法（有效）用来决定指定值是否为 PenStyle 类的有效成员。
下表列出了 PenStyle 常量：

| 常量 | 说明 |
| --- | --- |
| PenStyle.DASH | 表示画破折号的笔 |
| PenSTYLE.DOT | 表示划点的笔 |
| PenStyle.DASHDOT | 表示画点划线的笔 |
| PenStyle.INSIDEFRAME | 表示这个笔在框架或封闭的形状内画一条线，这个形状是由 Graphic 对象指定边框矩形的输出函数产生的（例如，drawRect, drawPie 和 drawChord） |
| PenStyle.NULL | 表示一个空笔 |
| PenStyle.SOLID | 表示一个实心笔 |

另外，Pen 对象包括了一组用于指定要创建的笔种类的公共成员。这些成员中的任何一个都是 Pen 对象，用于模拟 Windows 用户界面的各种

本机特征。

例如，Pen 类定义了一个 WINDOWFARME 成员。当创建 WINDOWFARME 类型的笔时，使用笔画的线看起来与活动窗口的边框是一样的。

```
Pen pen = Pen.WINDOWFRAME;
```

下表列出了定义为 Pen 类公共成员的 Pen 对象。

| 对象 | 说明 |
| --- | --- |
| Pen.ACTIVEECAPTIONTEXT | 创建活动窗口标题文本颜色的笔 |
| Pen.CONTROLTEXT | 表示控件上文本颜色的笔 |
| Pen.GRAYTEXT | 表示禁用文本颜色的笔 |
| Pen.HIGHLIGHTTEXT | 表示高亮文本颜色的笔 |
| Pen.INACTIVECAPTIONTEXT | 表示不活动窗口标题文本颜色的笔 |
| Pen.INFOTEXT | 表示信息工具提示文本颜色的笔 |
| Pen.MENUTEXT | 表示菜单文本颜色的笔 |
| Pen.NULL | 表示一个空笔，空笔不做任何事情 |
| Pen.WINDOWFRAME | 表示活动窗口边框颜色的笔 |
| Pen.WINDOWTEXT | 表示活动窗口文本颜色的笔 |

注意，当根据系统常量使用笔时，例如活动窗口文本的颜色，Pen 对应的系统设置的改变将导致笔的变化。

## 在Graphic对象上设置笔

Pen对象自身包括了无颜色或绘制功能。它只描述了GDI功能的一个子集。当在窗体上使用Pen之前，先使用Graphic对象的setPen方法将Graphic对象与Pen关联：

```
Graphics g = this.createGraphics();
g.setPen(new Pen(PenStyle.DASH));
```

使用Graphic对象关联了Pen后，在Graphic对象矩形框内的所有线的绘制都使用这支笔。另外，可以在相同的Graphic对象上多次调用setPen方法。
下例演示了具有Graphic对象画线功能的Pen对象是如何工作的。本例中，在Pen对象中定义的笔效果存在于一个整数数组中。在类的绘制处理程序中，通过笔效果的数组使用for循环来进行迭代，使用任一效果绘制一条线。

```
public class Form1 extends Form
}
int [] rgStyles = { PenStyle.DASH, PenStyle.DASHDOT,
PenStyle.DASHDOTDOT,
        PenStyle.DOT,PenStyle.INSIDEFRAME,PenStyle.SOLID };
```

```
protected void onPaint(PaintEvent e)
{
    Rectangle rc = this.getClientRect();
      rc.y += 10;
    for(int i = 0; i <rgStyles.length; i++){
          e.graphics.setPen(new Pen(Color.BLACK,i));
        e.graphics.drawLine(new Point(0, rc.y),
                            new Point(rc.width, rc.y));
          rc.y += 10;
    }
}
// Rest of Form1 class...
```

## 使用刷子

刷子是一种图形工具，基于 Win32 的应用程序使用它可以绘制多边形、椭圆、轨迹的内部。画图应用程序使用刷子可以绘制形状，字处理应用程序使用刷子绘制标尺，计算机辅助设计（CAD）应用程序使用刷子绘制横断面视图的内部，电子数据表应用程序使用刷子绘制饼图和条形

图的断面。

刷子有两种类型：逻辑型和物理型。逻辑型刷子是在代码中定义的，是应用程序绘制形状使用的颜色和样式的理想组合。物理型刷子以逻辑型刷子为基础，由设备驱动程序创建。

## 刷子的原点

当某一应用程序调用绘制函数绘制形状时，Windows在绘制操作的开始定位一个刷子，并在刷子的位图中对客户区的窗口原点映射一个像素（窗口原点位于客户区窗口的左上角）。Windows映射的像素点坐标称为刷子的原点。

刷子原点的默认值是刷子位图的左上角（0，0）坐标。Windows随后拷贝经过客户区的刷子，形成与位图同样高的样式。拷贝操作不断逐行进行，直到填满整个客户区。然而，刷子样式只有在指定形状的边界才是可视的（这里，术语位图多用于字面意义——如同安排位一样　　而不专指存储在图像文件中的位）。

在不使用默认刷子原点时，有许多实例。例如，对于一个应用程序来说，可以有必要使用相同的刷子绘制父窗口和子窗口的背景，并把子窗口的背景与父窗口的背景相混合。

## 逻辑型刷子类型

逻辑型刷子的类型有三种：实心，影线，样式。

实心（solid）刷子由 Windows 用户界面的一些元素定义的颜色或样式组成（例如，可以通过 Windows 显示禁用按钮使用的常规颜色和样式来绘制一种形状）。
影线（hatched）刷子由颜色的组合和 Win32 定义的六种样式之一组成。
样式（pattern）刷子由位图组成，这个位图作为填充形状的样式的基础。它填充的区域比位图要大，在显示中以水平和垂直方向平铺位图。样式刷子能够创建由自定义样式组成的自定义刷子。

## WFC Brush对象

Win32 刷子的功能封装在 WFCBrush 对象和 BrushStyle 对象中，可以协调使用这些对象来创建实心、样式和影线型刷子。
Brush 对象定义了一组公用的最终成员，每个成员都是一个代表实心刷子的对象。BrushStyle 类代表了作为一组整型常量的影线刷子，每个常量代表刷子的不同类型。
下表列出了代表实心刷子的 Brush 对象。

| 常量 | 说明 |
|---|---|
| Brush.AVTIVEBORDER | 表示活动窗口边框颜色的 Brush 对象 |
| Brush.AVTIVECAPTION | 表示活动标题栏颜色的 Brush 对象 |
| Brush.APPWORKSPACE | 表示应用程序工作区颜色的 Brush 对象 |
| Brush.CONTROL | 表示控件颜色的 Brush 对象 |
| Brush.CONTROLDARK | 表示 3D 元素阴影部分颜色的 Brush 对象 |

| | |
|---|---|
| Brush.CONTROLDARKDARK | 表示 3D 元素最深颜色部分的 Brush 对象 |
| Brush.CONTROLLIGHT | 表示 3D 元素高亮部分颜色的 Brush 对象 |
| Brush.CONTROLLIGHTLIGHT | 表示 3D 元素最亮部分颜色的 Brush 对象 |
| Brush.DESKTOP | 表示桌面当前颜色的 Brush 对象 |
| Brush.HALFTONE | 表示标准半色调颜色的 Brush 对象 |
| Brush.HIGHLIGHT | 表示高亮元素背景色的 Brush 对象 |
| Brush.HOLLOW | 表示一个空刷子，什么也不绘制 |
| Brush.HOTTRACK | 表示指出热点跟踪颜色的 Brush 对象 |
| Brush.INACTIVEBORDER | 表示不活动窗口边框颜色的 Brush 对象 |
| Brush.INACTIVECAPTION | 表示不活动标题栏颜色的 Brush 对象 |
| Brush.INFO | 表示信息工具提示背景色的 Brush 对象 |
| Brush.MENU | 表示高亮元素背景色的 Brush 对象 |
| Brush.NULL | 表示一个空刷子 |
| Brush.SCROLLBAR | 表示滚动条背景色的 Brush 对象 |
| Brush.WINDOW | 表示窗口背景色的 Brush 对象 |

下表列出了代表影线刷子的 BrushStyle 常量。

| 常量 | 说明 |
|---|---|
| BrushStyle.BACKWARDDIAGONAL | 表示反对角刷子。从刷子原点左上角到右下角的平行线 |
| BrushStyle.DIAGONALCROSS | 表示一个交叉影线的刷子 |
| BrushStyle.FORWARDDIAGONA | 表示一个正斜线刷子。从刷子原点右下角到 |

| | |
|---|---|
| L | 左上角的平行线 |
| BrushStyle.HOLLOW | 表示一个中空刷子。一个中空刷子与空刷等同 |
| BrushStyle.HORIZONTAL | 表示由间隔均等的水平线构成的样式 |
| BrushStyle.PATTERN | 表示一个样式刷子 |
| BrushStyle.SOLID | 表示一个实心刷子 |
| BrushStyle.VERTICAL | 表示由间隔均等的垂直线构成的样式 |

## 创建Brush对象

如何创建 Brush 对象取决于所需刷子的类型。因为实心刷子是 Brush 对象，可以如下创建实心刷子：

```
Brush br = Brush.BLACK;
```

样式刷子可由作为整型常量的 Brush 对象来代替。创建如下：

```
Brush br = new Brush(Color.BLACK, BrushStyle.FORWARDDIAGONAL);
```

下面的代码段演示了基于位图创建刷子的一种方法。

```
Brush bmpBrush = new Brush(new Bitmap("c:\\mybitmap.bmp"));
```

本例中假定刷子样式所依赖的位图存储在磁盘上的文件中。然而，也可以使用 Bitmap 对象的 CreateBitmap 方法，在运行期间定义一个位图，并使用定义的位图作为刷子样式的基础。

## 在Graphic对象上设置刷子

如同 Pen 和 Font 对象一样，Brush 对象包含无颜色或绘制功能。它是 GDI 功能的一个子集。当使用刷子进行填充之前，必须先使用 setBrush 方法用 Graphic 对象关联。

```
Protected void onPaint(PaintEvent e)
{
    // Create a forward diagonal brush, and associate it with the
    // object.
    Brush br = new Brush(Color.BLACK, BrushStyle.FORWARDDIAGONAL);
    e.graphics.setBrush(br);
}
```

当 Brush 对象与 Graphic 对象建立关联后，使用 Graphic 对象实例绘制的所有多边形都由关联的刷子填充。如果想用 Graphic 对象关联一个新刷子，可以多次调用 setBrush。

## Brush对象举例

自定义刷子基于从文件加载或在内存创建的位图，CustomBrush 样本应用程序演示了如何创建和使用自定义刷子。
CustomBrush 的应用程序由两个面板和两个按钮组成。第一块面板

（panelGrid）由 64 个正方形组成。当用户单击这些正方形中的某一个时，应用程序寄存器注册已单击的正方形，将正方形绘制为黑色，并逐位对短整数数组的元素执行操作。整数的修改取决于单击的正方形。
当用户单击 Test 按钮时，CustomBrush 应用程序根据短整型的数组创建位图，根据这个位图创建一个刷子，然后使用刷子为 Test 按钮上显示的面板（panelRect）绘制。
在应用程序的启动上，CustomBrush 应用程序执行如下任务：

- 调用用户定义的 getRects 方法来创建一个 64 Rectangle 对象的数组，并为应用程序的网格面板绘制这些矩形（panelGrid）。
- 声明一个短整型数组（bBrushBits）。存于数组中的整数值将最后形成创建位图的基础。当用户单击 Test 按钮时，使用位图来创建一个刷子，这个刷子绘制 Test 按钮上显示的面板。
- 声明并初始化一个布尔型变量数组为假。此数组用于跟踪 panelGrid 中每个矩形的状态。当用户单击 PanelGrid 中的一个矩形时，数组中相应的变量设置为真：

```
public class Form1 extends Form
}

// Rectangles into which to divide the panel.
Rectangle [] rgRects = new Rectangle[64];

// Tracks the state of the rectangles.
boolean [] rgStates = new boolean[64];
```

```
int [] bBrushBits = new int [8];

public Form1()
{
    // set the state array to an initial value of false;.
    for(int i = 0; i < rgSTATES.LENGTH; i++){
        rgStates[i] = false;
    }
    // Initialize the from.
    initForm();
    // Divide the panel into 64 rectangles and paint the form.
    rgRects = getRects(panelGrid.getClientRect(), 8, 8);
    this.invalidate();
}

/**
   * This method divides a specified rectangular area into the specified
   * number of subrectangles (down and across), and returns an array
   * containing the subrectangles
   */
private Rectangle [] getRects(Rectangle rcClient, int nAcross, int nDown){
```

```
int deltax, deltaY;
int x, y, right, bottom;
int i;

Rectangle rgRects[] = new Rectangle[nAcross * nDown];

// Store the right and bottom of the rectangle.

right = rcClient.getRight();
bottom = rcClient.getBottom();

// Determine the height and width of each rectangle.

deltaX = (right – rcClient.x)/nAcross;
deltaY = (bottom – rcClient.y) / nDown;

// Initialize the array of cell rectangles.

for(y = rcClient.y, i = 0; y < bottom; y += deltaY);

    for (x= rcClient.x; x < (right - nAcross) && i < (nAcross * nDown);
          x += deltaX, i+){

          rgRects[i] = new Rectangle(x,y,deltaX,detaY);

    }
}

return rgRects;
```

```
}
```

当调用网格控制面板的绘制事件处理程序时，使用存储于应用程序矩形数组（rgRects）中的维数和存储于类类布尔型数组（rgStates）中的状态来绘制网格。否则，使用一个空刷子绘制。这使用户可以决定刷子的外观，此刷子是在网格面板上根据矩形样式最终创建的。

```
private void panelGrid_paint(Object source, PaintEvent e)
{
e.graphics.setPen(new Pen(Color.BLACK,PenStyle.SOLID, 1));
// Draw the grid to reflect the current state of the
// squares.
for(int i = 0 ; i < rgRects.lenght; i++){
    if(rgStates[i] == true){
        // If the square has been previously clicked, fill it.
      e.graphics.setBrush(new Brush(Color.BLACK));
     }else{
      e.graphics.setBrush(Brush.NULL);
     }
    e.graphics.drawRect(rgRects[i]);
```

```
    }
}
```

每次用户单击网格面板中的一个矩形时，都会激活面板单击事件。在网格面板的单击事例处理程序中，应用程序确定单击了哪一个矩形，并设置相应的布尔状态为真，然后调用 panelGrid.invalidate 强迫其重新绘制。另外，对存储在类级 bBrushBits 变量中适当的数组成员逐位执行操作：

```
private void panelGrid_mouseDown(Object source, MouseEvent e)
{
    Graphics gr = this.createGraphics();
    gr.setBrush(new Brush(Color.BLACK));
    Integer nBit = new Integer(0);
// Determine which cell was clicked.
for(int i = 0; i < 64; i++){
    Rectangle rc = new Rectangle(rgRects[i].x,
                    rgRects[i].y, rgRects[i].heigh,
                    rgRects[i].width);
    if(rc.contains(new Point(e.x, e.y))){
      // Set the appropriate boolean state variable.
       rgStates[i] = true;
```

```
// Perform a bitwise NOT operation on the appropriate
//integer in the array of shorts on which we'll base
// our bitmap.
  if(i % 8 == 0)
   bBNrushBits[i/8] = bBrushBits[i/8] 0x80;
if(i % 8 == 1)
   bBNrushBits[i/8] = bBrushBits[i/8] 0x40;
if(i % 8 == 2)
   bBNrushBits[i/8] = bBrushBits[i/8] 0x20;
if(i % 8 == 3)
   bBNrushBits[i/8] = bBrushBits[i/8] 0x10;
if(i % 8 == 4)
   bBNrushBits[i/8] = bBrushBits[i/8] 0x08;
if(i % 8 == 5)
   bBNrushBits[i/8] = bBrushBits[i/8] 0x04;
if(i % 8 == 6)
   bBNrushBits[i/8] = bBrushBits[i/8] 0x02;
if(i % 8 == 7)
   bBrushBits[i/8] = bBrushBits[i/8] 0x01;
  break;
}
```

```
    }
     // Repaint the grid.
    panelGrid.invalidate();
}
```

最后，当用户单击 Test 按钮时，Test 按钮的单击事件处理程序通过调用 panelRect.invalidate 对 Test 按钮上显示的面板(panelRect)进行重新绘制。

```
Private void btnTest_click(Object source, Event e)
{
    panelRect.invalidate();
}
```

在 panelRect 绘制处理程序中，应用程序根据短整型数组创建一个位图（bBrushBits），根据位图创建一个刷子，然后绘制 Test 按钮上显示的面板（panelRect）。

```
Private void panelRect_paint(Object source, PaintEvent e)
{
    short [] rgShorts = new short[8];
     for(int i = 0; i < bBrushBits.length; i++){
```

```
            rgshorts[i] = (short) bBrushBits[i];
        }
        e.graphics.setBrush(new Brush(new Bitmap(8,8,1,1,rgShorts)));
        e.graphics.fill(panelRect.getClientRect());
    }
```

## 绘制位图

位图是一个可绘制的表面。此表面可用于显示笔、刷子或图像，包括 Windows 位图、元文件、.gif 和 .jpg 图像。

Bitmap 对象支持对位图的创建和加载。可以使用此对象加载一个来自文件、流、或资源的位图图像，或在内存创建位图。另外，Bitmap 对象支持在位图中定义透明色。

创建 Bitmap 对象后，使用 Graphic 对象的 drawImage 方法，在显示器中重现 Bitmap 对象。

下面的代码创建了一个新的 Bitmap 对象，然后使用 Form 类的 createGraphics 方法创建 Graphic 对象来绘制图像：

```
Bitmap bmp = new Bitmap("c:\\MyImage.bmp");
Graphics g = this.createGraphics();

g.drawImage(bmp, new Point(10, 10));
```

## 缩小和放大位图

WFC 支持的所有图像（Metafile、Icon、Bitmap 和 Cursor）都支持 drawStretchTo 和 drawTo 方法。通过这些方法执行 Graphic 对象，可实现所有图像的重现。调用 Graphic 对象的 drawImage 方法时，对象决定是否在图像对象上调用 drawStretchTo 或 drawTo 方法。

DrawStretchTo 方法放大或缩小图像以适应特定的矩形区域，drawTo 方法放大图像以适合特定的区域。但如果对目标区域来讲图像太长，drawTo 方法会剪切图像。另外，两种方法都支持绘制图像部分到 Graphic 边框矩形的指定部分。

不能直接调用 DrawStretchTo 或 drawTo 方法。它们是在 WFC 图像类中作为保护对象而定义的。为决定调用哪一种方法，须调用 drawImage 方法的一种版本，且带有一个布尔标量参数。下面是 Graphic 对象定义的一种方法。

```
Public final void drawImage (Image I, Rectangle r, boolean scale)
```

如果 scale 参数是真，则放大或缩小图像以适合剪切矩形域，否则，则剪切图像。

## 透明重现映射

在一个图像中，Bitmap 对象支持一种或多种颜色的透明重现。

如果需要以透明的方式重现单一的颜色，可使用 setTransparentColor 和

setTransparent 方法。SetTransparentColor 方法须带有一个 Color 对象参数，此参数指定透明重现的位图颜色。SetTransparent 方法需带一个布尔值，以指定是否重现设定的透明颜色。
例如，以下代码段指定黑色作为透明色。当调用 Graphic 对象的 drawImage 方法来绘制图像时，图像中的黑色像素点并不重现：

```
protected void onPaint(PaintEvent e)
{
    Bitmap bmp = new Bitmap("c:\\MyImage.bmp");
    Bmp.setTransparentColor(Color.BLACK);
    Bmp.setTransparent(false);
    e.graphics.drawImage(bmp, new Point(0,0));
}
```

也可以根据位屏蔽创建 Bitmap 对象来获得透明性。在两个位屏蔽中，一个是彩色，另一个是单色。在彩色屏蔽中，透明绘制位图的每一部分都应为黑色。在单色屏蔽中，透明绘制位图的每一部分都应为白色。

## 光栅操作

光栅操作是对 GDI 图元显示的一种逻辑操作，例如刷子、笔、图像或形状，以获得可视效果。能够使用 Graphic 对象执行的光栅操作是在 RasterOp 对象中定义的。当检查 Graphic 对象支持的这种方法时，会发

现每一个基础操作，如画线或图像显示等，这些方法都带有一个 RasterOp 作为参数。

在最简单的情形中，Graphic 对象的绘制方法只对一些显示区支持白色或拷贝的像素，重写最近显示的内容。添加光栅操作后，重写就变得非常复杂。

例如，假设要对最近被图像覆盖的区域绘制一个黑色的矩形框，但想逻辑组合这些黑色像素，使之与目标图像中的像素对应，并对结果进行显示。光栅操作可以实现这些组合操作。

在逻辑上，RasterOp 对象支持的变化太多了，以整本书的篇幅都不能介绍完全。下面的语句演示了调用 RasterOp 对象的基本语法。这些语句设置了窗体的背景色，并且用背景色的反色绘制了一条线。

```
Protected void onPaint(PaintEvent e)
{
this.setBackColor(new Color(255, 255, 255);

e.graphics.drawLine(new Point(10, 10), new Point(100, 10),

RasterOp.TARGET.invert());
}
```

## 绘制形状

Graphic 对象支持绘制线、矩形、弦、弧线、弧线角和贝塞尔（Bezier）

样条。

## 直线

直线是显示器上一组高亮显示像素，可以用两点来确定：起点和终点。在 Windows 中，位于起点的像素总是包括在线中，而位于终点的像素总是排除在外的（这些线有时被称为包括-排除（inclusive-exclusive））。通过调用 Graphic 对象的 drawLine 方法，可以画一条线。这种方法带有两个参数，来指明线的起点和终点。

```
Protected void onPaint(PaintEvent e){

    Rectangle rcClient = this.getClientRect();

    // Draw lines that divide the screen area equally into four squares.

    e.graphics.drawLine(new Point(rcClient.x, rcClient.height / 2),
                        new Point(rcClient.width, rcClient.height / 2));

    e.graphics.drawLine(new Point(rcClient.width / 2, rcClient.y),
                        new Point(rcClient.Width / 2, rcClient.height));
}
```

## 矩形

为 Microsoft Windows 编写的应用程序使用矩形指定屏幕上或窗口中的

矩形区域。使用矩形描述窗口的客户区、屏幕的重绘制区或格式文本的显示区。也可以使用矩形来进行填充、画边框或转换带给定刷子的部分客户区，并检索窗口坐标或窗口客户区。
矩形区的维数在 WFC Rectangle 对象中描述。这个对象由描述矩形的屏幕位置和维数的 X，Y、高度和宽度整数组成。另外，使用对象的 getRight 和 getBottom 方法可以检索右边和底边的屏幕位置。
更多信息请参看下节“矩形操作”和“矩形举例”。

## 矩形操作

Rectangle 对象提供了许多使用矩形的方法。这个对象的等价方法确定是否有两个 Regtangle 对象是一样的——也就是说，它们是否具有相同的坐标。
InflateRect 方法增加或减少矩形的宽度或高度或两者。它可以从矩形的两个端点增加或减少宽度，也可以从矩形的两个顶部和底部增加或减少高度。
过载的包含方法可以确定一个矩形描述的区域是否存在于另一个矩形描述的区域中，或确定矩形中是否存在一个给定的点。
Intersects 和 intersectsWith 方法确定了两个 Rectangle 对象的相交结果。

## 矩形举例

本节的例子说明如何把应用程序的客户区分成几个子矩形以及在这些区内如何工作。

启动应用程序时，把主窗体的客户区分成 16×16 的子区。并在这些矩形区中，以指定的颜色显示每个阴影。为修改显示的矩形数量，可以使用 Dimensions 菜单项来显示一个对话框，在对话框中，可以指定显示矩形的新数量。

下例中给出了把屏幕划分成几个 Rectangle 对象的方法。此方法 getRects 带有三个参数：指定划分区域的一个矩形，两个整数表明划分矩形长和宽的单位。这种方法返回一个矩形数组，此数组中包括相应的坐标：

```
private Rectangle[] getRects(Rectangle rcClient, int nDown, int nAcross){
    int deltaX, deltaY; // The height and width of each cell.
    int x, y;
    int i;
     Rectangle rgRects[] = new Rectangle[nDown * nAcross];
    // Determine the height and width of each Rectangle.
     deltaX = (rcClient.getRight() – rcClietn.x) / nAcross;
    deltaY = (rcClietn.getBottom() – rcClient.y) / nDown;
     // Create and initialize the Rectangle array.
        for(y= rcClient.y, i = 0 < rcClient.getRight() – nAcross) &&
            i < (nAcross * nDown); x += deltaX, i++){
                rgRects[I] = new Rectangle(x,y, deltaX, deltaY);
        }
```

```
    }
     // Return the initialized array.
    return rgRects;
}
```

启动应用程序时，Form 类构造器初始化两个类级整数，*nAcross* 和 *nDown* 为 16，并调用以前列出的 getRects 方法：

```
public Form1(){
    initForm();
    nAcross = 16;
    nDown = 16;
    // Initialize class-level array.
    rgRects = getRects(this.getClientRect(), nAcross, nDown);
}
```

类构造器返回后，自动调用类绘制事件处理程序。

事件处理程序通过类级 rgRects 数组循环执行，使用 Graphic 对象的 setBrush 方法设置新的颜色，然后调用 drawRects 方法来绘制数组矩形：

```
protected void onPaint(PaintEvent e){
    for(int i = 0; i < rgRects.length; i++){
        e.graphics.setBrush(new Brush(new Color(0,0,i)));
```

```
            e.graphics.drawRect(rgRects[i]);
        }
    }
```

最后，Form 类的重设大小处理程序简单地重新初始化数组，然后强制窗体的客户区重新绘制：

```
protected void onResize(Event e){

    Rectangle rcClient = this.getClientRect();
    rgRects = getRects(rcClient, nDown, nAcross);
    this.invalidate();
}
```

## 弦

弦是椭圆和直线段（称为割线）的交点界限内的区域。我们可以使用当前的笔来画弦的轮廓，使用当前的刷子来填充。在割线的一边剪辑部分椭圆。

要画一条弦，可以使用 Graphics 对象的 drawChord 方法，语法如下：

drawChord（Rectangle *p1*，point *p2*，point *p3*）

drawChord 方法的 Rectangle 参数指定了绘制椭圆的区域。其他两个 Point 参数指定了割线与椭圆的两个交点。

下例是对 drawChord 的两次调用。第一次调用在显示器的左下角画了一

个弦，第二次调用在右上角画了一个弦。每次调用 drawChord，都有不同的刷子关联了 Graphic 对象来确保两个弦组成一个圆，分别绘制黑色和白色：

```
protected void onPaint(PaintEvent e){
    Rectangle rcClient = this.getClientRect();
     // Associate a black brush with the object and draw a chord.
    e.graphics.setBrush(new Brush(new Color(0,0,0)));
    e.graphics.drawChord(rcClient,new Point(rcClient.x, rcClient.y),
       new Point(rcClient.getRight(),rcClient.getBottom());
     // Associate a white brush with the object and draw a chord.
    e.graphics.setBrush(new Brush(new Color(255,255,255)));
    e.graphics.drawChord(rcClient, new Point(rcClient.getRight(),
       rcClient.getBottom()), new Point(rcClient.x, rcClient.y));
}
```

## 弧

通过调用 drawArc 方法，应用程序可以画一个椭圆或椭圆的一部分。这种方法可以在不可视矩形的周边画一条曲线，称之为边框矩形。椭圆的尺寸可以通过从矩形中心到矩形两边放大的两个可视半径来指定。

调用 drawArc 方法时，应用程序要指定带边框矩形的坐标和半径。
下例画了一条弧，填充了窗体的客户区，并使用 Graphic 对象的 drawLine 方法从弧的顶端到半径画了一条线，然后从半径到椭圆的最右边画了一条线。

```
protected void onPaint(PaintEvent e){

    Rectangle rcClient = this.getClientRect();

    e.graphics.drawArc(rcClient, new Point(rcClient.width /2,rcClient.y),
        new Point(rcClient.width, rcClient.height /2));
    e.graphics.drawLine(new Point(rcClient.width / 2, rcClient.y),
        new Point(rcClient.width / 2, rcClient.height / 2));
    e.graphics.drawLine(new Point(rcClient.width, rcClient.height / 2),
        new Point(rcClient.width / 2, rcClient.height / 2));
}
```

## 弧线角

弧线角类似于弧线。两者之间最基本的实用性区别是：当应用程序使用 drawArc 画一个弧时，须指明弧线径向的 X 和 Y 位置。
对照来说，应用程序使用 drawArcAngle 方法画一个弧线角，需指明角的度数。方法本身需要照管画线始点和终点的坐标及定位。
drawArAngle 方法的语法如下：

public final void drawAngleArc( Point *center*, int *radius*, float *startAngle*, float *endAngle*)

Point 参数定义了屏幕上的径向位置，此位置是在 radius 参数中指出的。StartAngle 指明了起始的度数，endAngle 参数指出了从 StartAngle 开始画多少度角。

下例说明了此种方法的使用方式。在 30 度角处开始，延伸了 300 度画了一个弧度角：

```
protected void onPaint(PaintEvent e){

    Rectangle rcClient = this.getClientRect();
    int x = rcClient.width / 2;
    int y = rcClient.height / 2;
    int radius = 100;
    float startAngle = 30;
    float endAngle = 300;

    e.graphics.drawAngleArc(new Point(x,y), radius, startAnlge, endAngle);

}
```

## 贝塞尔（Bezier）样条

用四点可确定一个 Bezier 样条，包括两个端点和两个控制点。这些点组合在一起定义一个曲线。两个端点定义曲线的开始和结束。控制点把曲线拉离由始点和终点形成的线段。

为绘制Bezier样条，需使用drawBezier曲线方法。这种方法需要一个Point对象数组作为参数。数组中的四个元素定义了样条的始点、两个控制点和终点。
要绘制多个样条，每个样条在第一个元素之后，包括三个数组元素。第一个样条的终点作为下一个的始点，三个数组元素定义了控制点和终点。
下面的代码使用 drawBezier 方法绘制了一条贝塞尔曲线。表明每次窗口重置的大小：

```
protected void onPaint(PaintEvent e)
{
    Point [] pt = new Point[4];
    Rectangle rc = this.getClientRect();
    int right = rc.getRight();
    int bottom = rc.getBottom();

    pt[0] = new Point(right / 4, bottom / 2);
    pt[1] = new Point(right / 2, bottom / 4);
    pt[2] = new Point(right / 2, 3 * bottom / 4);
    pt[3] = new point(3 * right / 4, bottom / 2);

    e.graphics.drawBezier(pt);
}

protected void onResize(Event e)
```

```
{
    this.invlaidate();
}
```

# 第 16 章　建立和导入 ActiveX 控件

ActiveX 技术的基础是组件对象模型（COM）。另外，为创建 Java 组件 Windows Foundation Classes（WFC），可以使用 Visual J++来建立和导入 ActiveX 控件。由于 ActiveX 是基于 COM 的，可以像其他 COM 对象一样很容易地合并 ActiveX。可以在其他开发环境中，诸如 Microsoft Visual Basic 和 Microsoft Visual J++，开发控件进行应用，并在 HTML 页中提供高级特征。此外，可以导入第三方 ActiveX 控件，来增强 WFC 应用程序。

在本章中，将学到：

- 如何从现有 WFC 组件建立 ActiveX 控件。
- 如何把 ActiveX 控件导入 WFC 应用程序。

## 建立ActiveX控件

使用 Java 的 WFC 的组件模型，可以创建 ActiveX 控件，用以在 WFC 应用程序中或其他开发环境中支持 ActiveX。为从 WFC 控件中创建一个 ActiveX 控件，以便注册 WFC 控件的类作为 COM 类。一旦注册了控件的类作为 COM 类，就可以为控件打包一个类文件到 COM DLL，

并在注册表中注册它为 ActiveX 控件。一旦注册了控件为 ActiveX 控件，就可以从 ActiveX 客户访问它。
在本方案中，可以使用第 1 章“创建控件”中说明的控件。如果没有创建它，则建立本节中说明的控件，然后继续这个过程。你将学到：

- 如何公布 WFC 控件作为 COM 对象。
- 如何通过其他应用程序打包控件到 COM DLL。
- 如何注册 COM DLL 作为 ActiveX 控件。
- 如何导入基于 WFC 的 ActiveX 控件到 Visual Basic。

> **注意**：下面的过程假设在 Visual J++中已存在一个打开的 WFC 组件项目。

## 定义WFC控件作为COM对象

为了从其他 ActiveX 客户中访问你的控件，要把控件定义为 COM 对象。使类作为 COM 对象公布，需要在类定义上放一个@COM.Register 注释标记。Visual J++为类提供了一种自动生成@COM.Register 注释的方法。

> **注意**：如果已使用 Control 模板创建了控件的项目，则控件已经包含了一个注释标记，来注册它作为 COM 对象。删除前斜杠（//）来启用注释标记。

### 定义 WFC 组件作为 COM 对象

1.在 Project 菜单上，单击< Project>Properties（这里的< Project>是控件

项目名）。

2. 在 < Project>Properties 对话框中，单击 COM Class 标记。
3. 在类列表中，选择控件的类。
4. 单击 Options 按钮。
5. 可选在 Type Library Options 对话框中，改变创建的类型库文件的名字，来定义与控件、库名、控件名的接口，作为开发环境和帮助文件的信息来显示。然后单击 OK。
6. 在 < Project>Properties 对话框中，单击 OK。

Visual J++在控件类定义的上端添加了一个注释标记，来注册此类为 COM 对象。

> **注意**：如果不需要定义多个类作为 COM 类，可以在 Class Properties 对话框中定义一个 COM 类。为显示 Class Properties 对话框，在 Class Outline 中，右击类名，然后单击 Class Properties。在 Class Properties 对话框中，选择 COM Class 复选框。

## 在 COM DLL 中打包控件

定义了 WFC 控件作为 COM 对象后，就可以打包控件的类文件到 COM DLL 文件中。控件必须在 COM DLL 文件中打包，才能作为 ActiveX 控件使用。COM DLL 提供 ActiveX 客户使用的接口，来访问控件及其成员。

**注意：**为了在 Internet 上发布 ActiveX 控件，可以在 CAB 文件中打包控件，而不是在 COM DLL 中。

**建立控件作为 COM DLL**

1. 在 Project 菜单上，单击< Project>Properties（这里< Project>是指控件项目名）。
2. 在< Project>Properties 对话框中单击 Output Format 选项卡。
3. 选择 Enable Packaging 复选框。
4. 选项卡上的其他控件都是启用的。
5. 在 Packaging type 下拉列表中，选择 COM DLL。
6. 在 File name 框中，键入 COM DLL 的名字（默认名是使用项目名创建的）。
7. 在关联的下拉列表中，选择 outputs of type 和 Java Classes & Resources 选项。
8. 单击 OK。

## 建立项目

为项目配置了打包选项后，需要建立项目。Visual J++为项目添加了一个类型库，此项目为控件定义 COM 接口。类型库也包含了注册表注册 COM 类作为控件时使用的信息。当生成类型库时，Visual J++使用创建的类型库文件在注册表中注册项目中的 COM 类。注册了控件的类之后，Visual J++打包项目的类文件和类型库到 COM DLL 中。

### 建立项目

- 在 Build 菜单上，单击 Build。

## 注册 COM DLL

一旦注册了 WFC 控件的类为 COM 类，然后打包它们到 COM DLL，就在系统注册表中注册了 COM DLL。为了做到这一点，可以使用 Regsvr32.exe 程序。因为项目的类型库标记 WFC 控件的 COM 类为一个控件，则 Regsvr32 注册 COM DLL 为一个 ActiveX 控件。当注册 COM DLL 时，其他应用程序可以看到系统上注册的 ActiveX 控件列表中的 WFC 控件。

### 注册 COM DLL

1. 单击 Start 按钮，然后单击 Run。
2. 在 Open 框中键入：

   Regsvr32.exe<DLL 路径和文件名>

   这里的<DLL 路径和文件名>是指控件的 DLL 路径和文件名。本方案中，键入：

   Regsvr32  c:\project\project1.dll
3. 单击 OK。

   如果收到注册失败的信息，请确认控件的 DLL 的路径是否正确，文件是否存在。

# 测试 Microsoft Visual Basic中的控件

为了测试 ActiveX 控件，可以为编程工具和支持 ActiveX 的应用程序添加控件。在本方案中，可以使用 Microsoft Visual Basic 5.0 版或以后版本添加一个控件，然后测试其特性。

## 为 Visual Basic 窗体添加基于 WFC 的 ActiveX 控件

1. 运行 Visual Basic。
2. 在 Visual Basic 中的 File 菜单上，单击 New Project。
3. 在 New Project 对话框中，单击 Standard EXE 图标，然后单击 OK 。
4. 右击工具箱，然后单击 Components。
5. 在 Components 对话框中，选择控件，然后单击 OK。
   对本方案，选择 Project1 控件。
6. 在工具箱中，双击控件，以在窗体中添加它。
   控件添加到窗体中心。
7. 按 F5，运行项目。
   则显示了带控件的窗体。

如果使用在第 1 章“创建控件”一节说明的控件，则可以滚动水平滚动条，控件中的文本随着滚动条的变化而变化。

关于在 WFC 应用程序中导入 ActiveX 控件的信息，请参看下一节“导入 ActiveX 控件”。

# 导入ActiveX控件

ActiveX 控件可以对 WFC 应用程序提供许多增强功能。许多第三方 ActiveX 控件可用于添加功能。诸如定制按钮的形状、电话学技术、图标、绘制图形和电子表格。可以使用 Visual J++ 中导入 ActiveX 控件，其过程与导入 COM 对象类似。

在本方案中，将导入用 Microsoft Internet Explorer 4.0 版安装的 Microsoft ActiveMovie 控件。将学到：

- 如何在系统注册表中注册 ActiveX 控件。
- 导入 ActiveX 控件时，如何创建项目。
- 如何把 ActiveX 控件导入到项目中。
- 如何为窗体添加 ActiveX 控件并设置其属性。
- 如何建立和运行项目来测试 ActiveX 控件。

## 注册控件

为了使 ActiveX 控件对 Visual J++可用，必须在系统注册表中注册此控件。

> **注意：** 本方案中，由于在安装 Internet Explorer 时已注册了 ActiveMovie 控件，所以不需要再注册。

### 注册 ActiveX 控件

1. 单击 Start 按钮，然后单击 Run。
2. 在 Open 框中，键入：

   Regsvr32.exe<Control 路径和文件名>

   这里的< Control 路径和文件名>是指控件的路径和文件名。本方案中，键入：

   Regsvr32 c:\Windows\system\AMOVIE.OCX。
3. 单击 OK。

   如果收到注册失败的信息，请确认控件的路径是否正确，文件是否存在。

## 创建 WFC 项目

当导入 ActiveX 控件时，Visual J++在项目中创建软件包目录，并在这些软件包目录中添加类封装器（wrapper）来访问 ActiveX 控件。因此，必须有有效的项目来导入 ActiveX 控件。

### 创建 WFC 控件

1. 在 File 菜单上，单击 New Project。
2. 在 New 选项卡上，展开 Visual J++ Project 文件夹，单击 Applications，然后单击 Windows Applications 图标。
3. 在 Name 框中，键入项目名。
4. 在 Location 框中，键入要保存项目的路径，或单击 Browse 来导入文件夹。

5.单击 Open。
则在 Project Explorer 中出现项目的折叠视图（如果 Project Explorer 没有显示，则在 View 菜单上，单击 Project Explorer）。
6.在 Project Explorer 中，展开项目节点。
添加到项目中的文件具有默认名 Form1.Java。
7. 在 Form Designer 中打开窗体，在 Project Explorer 中单击 Form1.Java。

## 导入 ActiveX 控件

创建了项目之后，就可以在项目中导入 ActiveX 控件。使用 Customize Toolbox 对话框，从系统中安装的 ActiveX 控件表中选择 ActiveX 控件。

> **注意：**一旦对某个工具箱添加了 ActiveX 控件，则控件会一直保留在工具箱中，其他所有项目都可以使用，直到删除它为止。

### 导入 ActiveX 控件

1.在工具箱中，单击 General 选项卡（如果工具箱没有出现，则单击 View 菜单上的 Toolbox）。
2.右击工具箱，然后单击 Customize Toolbox。
3.在 Customize Toolbox 对话框中，单击 ActiveX Controls 选项卡。
4.在 ActiveX 控件列表中，选择要导入的控件。也可以选择多个控件。
对本方案，选择 ActiveMovieControlObject。
5.单击 OK。

把选择的控件添加到工具箱。

## 为窗体添加控件

当对窗体添加 ActiveX 控件时，Visual J++在项目目录中创建一个或多个软件包，并为这些控件添加类封装器。通过 Visual J++使用类封装器，可以访问类及 ActiveX 控件的成员。如果要导入的 ActiveX 控件包含在某文件中，而此文件又包含多个 ActiveX 控件，则 Visual J++将对文件中的所有控件提供类封装器，但只把指定的控件添加到工具箱。

### 为窗体添加控件

- 在 Form Designer 中显示使用的窗体，双击要添加的控件。
  在本方案中，双击 ActiveX Movie 控件。此控件添加到窗体的中央。

## 设置控件的属性

为窗体添加了 ActiveX 控件后，就可以使用 Properties 窗口来设置属性，创建事件处理程序。

### 为 ActiveX Movie 控件设置属性

1. 在 Form Designer 中，选择 ActiveXMovie 控件。
2. 在 Properties 窗口中，选择 filename 属性。
3. 键入.avi 文件的路径和文件名，以在控件中显示。
   或

单击 filename 属性值部分的省略号按钮，显示对话框，浏览计算机中的文件。

## 建立项目

对窗体添加了 ActiveX 控件，并设置了其属性之后，就可以建立并运行此项目，来测试控件的函数性。

### 创建并运行窗体

1. 在 Build 菜单上，单击 Build。
   如果收到编译错误的信息，请改正错误后重新建立项目。
2. 为运行窗体，在 Debug 菜单上，单击 Start。
3. 显示窗体时，单击控件上的播放按钮。
   这就显示了 filename 属性中指定的影片。

# 第 17 章　建立和导入 COM 对象

组件对象模型（COM）是面向对象的体系结构，用于创建可从应用程序之间重用的对象。也可以使用 COM 编写可从其他编程环境中访问的对象，诸如 Microsoft Visual Basic 和 Microsoft Visual C++，以及 Microsoft Office 这样的应用程序。COM 对连接对象提供标准协议，即使它们是用不同的编程语言设计的。当建立了连接后，对象就可以通过标准的用户界面进行通信。

Visual J++支持建立和导入 COM 对象。通过在 Visual J++中建立 COM 对象，就能够提供可重用的组件，以便由许多应用程序以及用不同的语言编写的应用程序所共享。利用 COM 导入特征，可以从其他应用程序中使用对象，以便在 Windows Foundation Microsoft Office（WFC）应用程序中提供附加的特征。例如，可以使用 Microsoft Office 的 Microsoft Word 中的访问拼写检查特征或 Microsoft Excel 中数学函数这样的 COM 对象。通过在 Visual J++项目中使用 COM，在应用程序的开发中可以提供可重用的代码。在本章中，将学到：

- 如何建立 COM 对象。
- 如何导入现有 COM 对象到 WFC 应用程序中。

## 建立COM对象

Visual J++通过提供易用的界面，把要使用的项目选择类作为 COM 类，从而简化了 COM 对象的建立。任何公用的、非抽象的类都可用来建立 COM 对象。Visual J++使用类的公共成员作为对 COM 对象的接口。当建立项目时，选择类作为建立的 COM 类，并在系统中进行注册作为 COM 对象。一旦建立了 COM 对象，就可以在 COM DLL 中打包它们，并从其他编程环境或支持 COM 的程序中访问它们。

在本方案中，要建立 COM 对象，在 COM DLL 中打包，给出相关的活动，统计函数性。在此将学到：

- 创建具有作为 COM 对象给出的类的项目。
- 在 Project Properties 对话框中，使用 COM Classes 选项卡来选择在项目给出的作为 COM 对象的类。
- 为使用其他编程语言和应用程序，打包 COM 类到 COM DLL。
- 建立项目。

## 创建项目

Visual J++ 提供 COM DLL 模板，创建带有已注册为 COM 类的项目。这种模板在 COM DLL 中也可用于打包项目。至于更多的关于使用模板创建 COM DLL 项目的信息，请参看第 1 章的“创建 COM DLL ”。

尽管 COM DLL 模板可用，但本方案并不使用它，因为对于理解如何在

多个项目中选择类作为 COM 类是非常重要的。在本方案中，要创建空项目，并为项目加上 Java 类作为 COM 对象。

> **注意**：启动下列过程之前，关闭所有打开的项目（在 File 菜单，单击 Close All）。

### 创建空项目

1. 在 File 菜单上，单击 New Project。
2. 在 New 选项卡中，选择 Visual J++ Projects 文件夹，然后单击 Empty Project 图标。
3. 在 Name 框中，键入项目名。
   对于本方案，键入 Stats。
4. 在 Location 框中，键入项目的保存路径，或单击 Browse 来定位目录。
5. 单击 Open。
   在 Project Explorer 中显示项目，但不包含文件。

### 为项目加入类

1. 在 Project Explorer 中，右击项目名。
2. 在快捷菜单中，指向 Add，然后单击 Add Class。
3. 要添加空的 Java 类，选择 Class 图标。
4. 在 Name 框中，键入 Java 类名。
   在本方案中，类名为 Stats.Java。
5. 单击 Open。

## 对类添加代码

对于 COM DLL 客户，要操纵 COM 对象，必须在 COM 类中提供公共方法。Visual J++通过 COM 对象的接口给出了 Java 类的所有公共方法，包括那些从超类继承的方法。在本方案中，为类添加代码有两种方法。这些方法对 COM 对象的用户提供了与获得相关的统计函数。

Stats COM 对象需要的第一种公共方法是 WinlossPercentage 方法。这种方法用于计算某队在比赛中赢的百分比。

### 添加 WinLoss Percentage 方法

- 对 Stats 类，添加以下方法定义：

```
public float winLossPercentage(int gamesPlayed, int gamesWon)
{
    float returnValue = gamesWon % gamesPlayed * .100f;
    if (returnValue == 0.0f)
       return 1.0f;
    else
       return returnValue;
}
```

本代码采用模运算符来获得余数，此数是本队所赢比赛数与比赛总数相除得到的。然后将此数乘上.100f，转换成此队的显示值。此值确定后，代码检查它是否为 0，如是，则说明没有余数，此队赢了所有的

比赛，返回 1，否则，代码返回实际数值。

Stats COM 对象需要公布的最终公共方法是 goalsAgainstAverage 方法。这种方法计算射门的平均数。确定的方法是在所有比赛中，用射门次数除以比赛次数。

### 添加 goalsAgainstAverage 方法

- 对 Stats 类添加下列方法定义：

```
public float GoalsAgainstAverage(int gamesPlayed, int goalsAllowed)
{
    return (float) goalsAllowed/gamesPlayed;
}
```

本代码用射门次数除以比赛次数，并将结果转换为浮点数。

## 定义类作为 COM 类

当创建了类并定义了通过 COM 公布的公共方法之后，就定义了类作为 COM 类。在 Project Properties 对话框中使用 COM Classes 选项卡，来从项目中选择类作为 COM 类。

注意：如果不需要定义多个类作为 COM 类，可以在 Class Properties 对话框中定义 COM 类。为显示 Class Properties 对话框，在 Class Outline 中，右击类名，然后单击 Class Properties。在 Class Properties 对话框中，选择 COM Class 复选框。

**定义类作为 COM Class**

1. 在 Project 菜单中，单击<Project > Properties（<Project >是指控件项目的名字）。
2. 单击 COM Class 选项卡。
3. 在 COM Classes 选项卡中，选择 Automatically Generate Type Library 选项。
4. 在类列表中，选择想公布为 COM 对象的类。
   对这种方案，选择 stats 类。
5. 单击 OK 。
   Visual J++ 在类定义的上边添加了@com.register 注释标记。当编译项目时，编译器使用注释标记中的信息，在注册表中注册作为 COM 类的类。

## 打包项目作为COM DLL

为了使其他应用程序可以使用 COM 对象，可在 COM DLL 中打包它。

注意：如果想在 Internet 上分布 COM 对象，可以在 CAB 文件中打包控件来代替 COM DLL。

### 在 COM DLL 中打包 COM 对象

1. 在 Project 菜单中，单击<Project > Properties（<Project >是指控件项目的名字）。
2. 单击 Output Format 选项卡。
3. 选择 Enable Packaging 复选框。
4. 在 Packaging type 下拉菜单中，选择 COM DLL 。
5. 在 File Name 框中，键入 COM DLL 名字（默认名是使用项目名创建的）。
6. 选择 Outputs of type 和 Java Classes & Resources 选项。
7. 单击 OK。

## 建立项目

在建立过程中，Visual J++在 COM DLL 中打包 COM 类，并注册 DLL 和 COM 类作为 COM 对象。当类在注册表中时，可用于其他应用程序。关于在 Visual J++中导入 COM 对象的更多信息，请参看下节“导入 COM 对象”。

### 建立项目

- 在 Build 菜单上，单击 Build（如果收到一些编译错误信息，更正错误，然后重新建立项目）。

**注意：**注册了 COM 类后，就可以执行附加的注册。对 COM 类，可以添加一种名为 onCOMRegister 的用户定义的方法，可以使用这种方法执行诸如注册 Visual J++ 加载项到 Visual J++加载项表中这样的任务。在 COM 类在 COM 注册过程中和使用 Visual J++在 COM DLL 建立注册的过程中，Visual J++可调用此方法。onCOMRegister 方法的特征必须与下列特征相匹配。

```
Public static void onCOMRegister(boolean register)
{
    // Add your custom registration code here
}
```

## 导入COM对象

COM 对象提供多种方法来封装功能，并在多个应用程序中可重用它。可以使用 COM 对象在应用程序中公布具体的功能，以便在其他应用程序中使用，或者创建一组要在多个应用程序中使用的例程。因为 COM 不是针对具体的语言，所以，COM 对象可以集成到在各种编程语言中开发的应用程序中。可以使用 Visual J++来查看和导入在系统中注册的 COM 对象。在导入过程中，Visual J++创建类封装器，以使 COM 对象的访问与其他 Java 对象一样。

在本方案中，可以导入在本章前面的“建立 COM 对象”一节中建立的 COM 对象，在此将学到：

- 如何为 Java（WFC）项目导入 COM 对象。
- COM 对象的访问方法。
- 建立并运行项目来测试 COM 对象的功能。

**注意：**本节假设已在本章前面的“建立 COM 对象”中创建了 COM 对象，然后已关闭了所有打开的项目。

## 创建项目

当导入 COM 对象时，Visual J++在项目中创建目录，并在这些目录中加上类封装器以访问 COM 对象。为导入 COM 对象，必须有有效的 Java 项目。

### 创建项目

1. 在 File 菜单上，单击 New Project。
2. 在 New 选项卡中，选择 Visual J++ Projects 文件夹，然后单击 Applications，然后选择 Windows Applications 图标。
3. 在 Name 框中，键入项目名。
4. 在 Location 框中，键入项目的保存路径，或单击 Browse 来定位文件夹。
5. 单击 Open。
   在 Project Explorer 中显示项目的封装视图。
6. 在 Project Explorer 中，展开项目的节点。
   则在项目中添加了以 Form1.Java 为默认名的文件。

## 导入COM对象

创建了项目后，就可以为此项目导入 COM 对象。导入的 COM 对象在每个项目中必须是可访问的。当为项目导入了 COM 对象后，Visual J++ 创建类封装器来为访问 COM 对象提供接口。这些类封装器添加到项目目录下的软件包中（这取决于 COM DLL 中存储的对象数量，Visual J++ 可以创建多个软件包）。

> **注意：** 为了从另一个项目中访问特定的 COM 对象，可以通过在 Classpath 中放置 COM 封装器类，防止为每个项目封装对象。

### 导入 COM 对象

1. 在 Project 菜单上，单击 Add COM Wrapper。
   出现 COM Wrappers 对话框。
2. 在包含 COM DLLs 的表中，键入计算机上注册的库，选择想导入的 COM 对象的名字。
   在本方案中，选择 Stats COM 对象。
3. 单击 OK 。
   Visual J++为所选 COM DLL 中包含的每个 COM 对象添加软件包目录和类封装器。

## 为访问COM对象添加代码

Visual J++创建了类封装器后，就可以为 COM 对象的访问方法编写代

码。在本方案中，可以对项目窗体构造器添加代码。此代码创建了 COM 对象的实例，并生成对 COM 对象两种方法的调用。使用硬编码的值调用方法，然后在 Output 窗口显示结果。

### 为窗体构造器添加代码

1. 在 Project Explorer 中，右击 Form1，然后单击 View Code。
   在文本编辑器中显示 Form1 的源代码。
2. 在 Form1 构造器内部，调用 initForm，添加如下代码行：

```
statisticsobject.Stats stats = new statisticsobject.Stats();
System.out.println("Our team has a winning percentage of " +
        Stats.winLossPercentage(10, 4));
System.out.println("Our goaltender has a Goals Against Average of " +
        Stats.goalsAgainstAverage(12, 15));
```

这些代码创建了 Stats COM 对象的实例。在 Stats 对象之前，对 statisticsobject 的引用是创建 Stats 类封装器的软件包的名称。定义了 Stats 实例后，代码使用硬编码值调用 winLossPercentage 和 goalsAgainstAverage 方法，并通过调用 System.Out.println 发送结果到 Output 窗口。

## 建立和运行项目

当合并了调用 COM 对象的方法之后，就可以建立并运行此项目。

## 建立和运行项目

1. 在 Build 菜单上，单击 Build（如果收到一些编译错误或信息，更正错误后重新建立项目）。
2. 要运行窗体，单击 Debug 菜单上的 Start。

启动项目之后，就可以在 Output 窗口中看到来自对 Stats COM 对象调用的输出（为显示 Output 窗口，在 View 菜单上，单击 Other Windows，然后单击 Output）。

# 第 18 章　WFC 中的数据绑定

为了从 Java 项目中访问数据，通常可以使用 ActiveX Data Objects（ADO）组件，此组件定义了 WFC 应用程序的数据编程模型。ADO 对象的核心包括：Connection, Command 和 Recordset 对象。

利用 Connection 对象可连接到数据库。一旦建立了连接，就可以查询数据库，检索记录集。Recordset 对象表示查询返回的记录，可以使用 SQL 字符串或 Command 对象来指定查询。

ADO 也提供 DataSource 组件，它组合 Connection、Command 和 Recordset 对象的功能。关于使用 ADO 对象编程的信息，请参看 Microsoft ActiveX Data Objects 在线文档中的“ADO Tutorial（VJ++）”。在 com.ms.wfc.data 和 com.ms.wfc.data.ui 软件包中定义 ADO 类。

> **注意：** Form Designer 中的 Toolbox 只提供 DataSource 控件；在代码中只可使用 Connection、Command 和 Recordset 对象。

一旦通过 Recordset 对象或 DataSource 组件检索了记录集，就可以对 WFC 组件绑定记录集。WFC 支持简单的或复杂的数据绑定。说明如下：

| 绑定类型 | 说明 |
| --- | --- |
| 简单的 | 指记录集中的字段与 WFC 组件属性之间的关系，称之为简单绑定是因为组件不需要知道数据协议或数据提供者 |

复杂的　　指记录集和 WFC 组件之间的直接关系

注意，记录集的光标和锁定类型决定了记录集中的数据是否动态反映了数据库中的数据，以及记录集中的数据是否可以改变。

关于 ADO 的更多信息，请参看 Microsoft ActiveX Data Objects 在线文档中的“Getting Started with ADO 2.0”。

## 简单数据绑定

简单数据绑定指的是在记录集中的字段与 WFC 组件属性之间的关系。绑定了属性之后，就可以自动在字段与属性之间传递数据：

1. 如果改变了字段的值，则新值也传给属性。
2. 如果在记录集中导入了新的记录，则当前字段值也传给属性。
3. 如果属性的值改变了，并且组件支持属性改变的通知，则新值也传给记录集中的字段。

**注意：**如果想尝试改变只读记录集，则记录集将会产生 ADO 异常。可以捕捉这个异常，然后，把绑定的属性返回到原来的值；否则，属性的值和字段的值将是不一致的。

## 可绑定属性

通过 DataBinder 组件可执行 WFC 中的简单数据绑定。这个组件创建并

管理字段与属性间的绑定。DataBinder 组件可以通过与下面的设计模式相匹配的方法来绑定可访问的属性：

public <*PropertyType*> get<*PropertyName*>()

public void set<*PropertyName*>(<*PropertyType*>)

> **注意：** 如果属性也可以用 BindableAttribute.YES 在组件的 ClassInfo 中标记，则在 Properties 窗口和在 DataBinder 组件的设计页中，将列举出此属性（依旧可以在 Properties 窗口或设计页中手工键入名称或通过编程绑定属性，来绑定没有标记为可绑定的属性）。

## 属性改变通知

绑定属性的组件可以提供<*PropertyName*>Changed 事件来声明属性值已经改变了。当触发了此事件时，DataBinder 组件标记此绑定为无效。然后当用户定位到新记录或在代码中调用 DataBinder.commitChanges 方法时，DataBinder 组件决定哪些绑定是无效的，并在记录集上更新这些绑定。

> **注意：** 如果组件没有提供<*PropertyName*>Changed 事件来通过 DataBinder 组件绑定属性，则绑定是只读的，且将禁用组件。为在这种限制下工作，可以设法在代码中通过直接对记录集进行读写来管理绑定。

# DataBinder组件

ADO 定义了 DataBinder 组件来执行简单的数据绑定。这个组件在记录集中的字段和 WFC 组件的绑定属性间建立和管理记录。关于属性是否可以绑定的信息，请参看前面的“绑定属性”一节。

尽管单一的 DataBinder 组件可以管理多重绑定，但它只与单一的记录集关联。为在第二个记录集中绑定字段，必须使用另一个 DataBinder 组件。另外，DataBinder 组件只可用于绑定相同窗体上存在的组件。

如果绑定属性的组件支持属性的改变通知，则当属性值改变时，DataBinder 组件标记绑定无效，当用户定位到新记录或在代码中调用 DataBinder.commitChanges 方法时，就改变了记录集。更多的信息，请参看前面的一节“属性改变标记信息”。

## 示例

下例表明了如何编程带有 Data Binder 组件的简单数据绑定。注意，DataBinder 组件使用 DataBinding 对象的数组来管理绑定。

```
import com.ms.wfc.data.*;
import com.ms.wfc.data.ui.*;
import com.ms.wfc.ui.*;

/* Use the Connection and Recordset components to
   connect to a database and retrieve a recordset. */
Connection c=new Connection();
```

```
c.setConnectionString("dsn=myDSN; uid=myUID;pwd=myPWD");
c.open();
Recordset rs = new Recordset();
rs.setActiveConnection( c );
rs.setSource("select * from authors");
rs.open();
/* Create a DataBinder component. To associate this
   componet with rs, set the dataSource property. */
DataBinder db = new DataBinder();
db.setDataSource(rs);
/* Create the components whose properties will be bound. */
Edit edit1 = new Edit();
Edit edit2 = new Edit();
/* Set the DataBinder compoent's bindings property to
   an array of DataBinding objects. This array defines
   the bindings between the text property of each Edit
   component and the fields in the recordset. */
db.setBindings(new DataBinding[] {
        new DataBinding(edit1, "text", "firstName"),
        new DataBinding(edit2, "text", "lastName") } );
}
```

关于 DataBinder 组件 dataSource 属性的更多信息，请参看第 4 章。

## 复杂数据绑定

复杂数据绑定是指记录集之间直接相互作用的组件。复杂数据绑定组件提供 dataSource 和 dataMember 属性，这些属性标识了绑定的记录集。注意，复杂绑定组件的属性可以借助于 DataBinder 组件而仍然为简单的绑定。

## dataSource和dataMember属性

dataSource 属性确定了实现 IDataSource 接口的对象。此对象公布了一个或多个记录集。DataMember 属性随后指定组件目前绑定的记录集名称。例如：

```
/* Bind the component to the recordset named "Products",
   which is exposed by the data source, ds. */
dhComponent.setDataSource(ds);
dbcomponent.setDataMember("Products");
```

如果不能设置 dataMember 属性，则绑定数据源的默认记录集（可以通过设置 dataMember 属性为空，来明确指定默认记录集）。

```
/* bind the component to the default recordset
```

```
    exposed by the data source, ds. */
dbComponent.setDataSource(ds);
dbComponent.setDataMember(null); /* This line is optional. */
```

注意，Recordset 和 DataSource 组件已经实现了 IDataSource 接口。所以，可以直接设置 DataSource 属性为组件中的一员。本例中，不必设置 DataMember 属性：

```
/* set the dataSource property directly to the Recordset
   component,rs, without setting the dataMember property. */
dbComponent.setDataSource(rs);
```

## 在 Visual J++中的复杂绑定组件

Visual J++提供下列复杂数据绑定组件：

| 组件 | 说明 |
|---|---|
| DataBinder | 绑定记录集字段到 WFC 组件的属性（尽管使用 DataBinder 组件可用来执行简单的绑定，但组件本身公布了 DataSource 和 DataMember 属性） |
| DataGrid | 绑定记录集的多个字段，并以网格格式显示此数据 |
| DataNavigator | 允许用户改变记录集中的当前记录。绑定相同记录集合的其他组件可随后更新，以反应新的当前行 |

关于在 Form Designer 中使用这些组件的信息，请参看第 4 章。

# 第 19 章　使用 J/Direct 编写 Windows 应用程序

J/Direct 是 Microsoft Visual J++提供的一项新特征，可以很容易地对 Microsoft Windows 的动态链接库（DLLs）进行访问。使用 J/Direct 可以直接调用标准的 Win32 系统 DLLs（诸如 KERNEL32 和 USER32）和第三方 DLLs。使用 J/Direct 要比使用旧的 Raw Native Interface 和 Java Native Interface（分别为 RNI 和 JNI）要简单得多。这二者都要求编写专门的封装器 DLL。然后，自己执行所有数据类型转换。使用 J/Direct，通过简单声明并调用函数，就可以调用大多数以前存在的 DLL 函数。J/Direct 使用@dll.import 伪指令，此伪指令与 Visual Basic 的 DECLARE 功能很相似。

为使用 J/Direct，需要安装 Microsoft Visual J++和 Microsoft Internet Explorer 4.0 版或更高版本。为了快速建立利用 J/Direct 的本机能力的应用程序，需使用 J/Direct Call builder，它是 Visual J++开发环境的一部分。本章解释如何使用@dll.import 伪指令来调用来自 Java 的 DLL 函数。也提供了如何传送和接收任一数据类型的有关细节，并说明了如何使用@dll.import 伪指令来传送和接收来自 DLL 方法的结构。

# 消息框示例

本节提供使用@dll.import伪指令的消息框应用程序。
使用 J/Direct 可以直接从 Java 代码调用 Win32 DLLs。下面是显示消息框的 Java 应用程序。

```
Class ShowMsgBox {
    Public static void main(String args[])
    {
        MessageBox(0, "It worked!",
                    "This messagebox brought to you using J/Driect", 0);
    }
    /** @dll.import("USER32") */
    private static native int MessageBox(int hwndOwner, String text,
                                         String title, int fuStryle);
}
```

@dll.import 伪指令告诉编译器，MessageBox 方法将使用 J/Direct 协议链接到 Win32 USER32.DLL，而不使用以前版本支持的 Raw Native Interface（RNI）协议。另外，Java 的 Microsoft Virtual Machine（VM）提供自动类型度，从 String 对象到 C 中使用的以空字符结尾的字符串。没有必要指出是否调用 ANSI MessageBox（MessageBoxA）函数或 Unicode MessageBox 函数（MessageBoxW）。在上面的例子中，总是调用 ANSI 版本。在本章后面的“VM 如何在 ANSI 和 Unicode 之间进

行选择”一节，解释了如何使用 auto 修饰符来调用函数的最优版本，此函数依赖于支持应用程序的 Microsoft Windows 版本。

## J/Direct Call Builder

使用 J/Direct Call Builder，可以快速对 Win32 API 创建 J/Direct 调用。J/Direct Call Builder 对 Win32 API 元素自动插入 Java 定义到代码中，以及适当的 @dll.import 标记。

### 使用 J/Direct Call Builder 访问 Win32 API

1. 打开 J/Direct Call Builder，在 View 菜单上指向 Other Windows，然后单击 J/Direct Call Builder 。
2. 默认情况下，J/Direct Call Builder 显示下面 Win32 DLL 中定义的元素：advapi32.dll、gdi32.dll、kernel32.dll、shell32.dll、spoolss.dll、user32.dll 和 winmm.dll（这些 DLLs 都是通过在 Source 下选择 WIN32.TXT 文件来指定的）。
3. Target 框标识了包含 J/Direct 调用的类。默认类名为 Win32，它将添加到项目中，并包含这些调用（如果解决方案中包含了多个 Java 项目，Win32 类将添加到第一个项目中。
   要指定不同的目标类，单击省略号按钮。
   - 在 Select Class 对话框中，在 Project View 或 Class View 中，选择想要查看的类。项目视图显示了解决方案中 .Java 文件的分层列类，这里，每个文件节点都列出了此文件中包含的名称，类视图显示了

类名的分层列表。

- 一旦选择了视图，就可以选择包含 J/Direct 调用的类名（为了对 Windows Clipboard 而不是对类进行复制调用，选择 Clipboard）。
- 单击 OK。

4. 为了过滤 Win32 方法、结构和常量的显示，选择或清除 Methods，Structs 和 Constants 选项。
5. 现在，选择要插入的方法、结构和常量（可以使用 SHIFT 和 CTRL 键选择多个项）。注意，Win32 结构将被作为嵌套类添加到类中。

   **注意：**为通过首字符搜索项，在 Find 框中键入字符，这将自动选择匹配这些字符的第一项。

6. 在类中，插入关联的 Java 定义，单击 Copy To Target（也可以双击方法、结构或常量，把它插入到类中）。

当在 J/Direct Call Builder 中选择了 Win32 API 元素时，下面的预览窗格显示关联的 Java 定义。可以把文本拷贝或剪切到任何文件中。为了 Win32 API 元素快速查找在线参考信息，右击项，然后在快捷菜单上单击 Display API Help。

关于编译器默认设置的信息，请参看下一节“设置 J/Direct Call Builder 选项”。

# 设置J/Direct Call Builder选项

Visual J++允许用户设置两个选项，来定制J/Direct Call Builder的功能。

**设置 J/Direct Call Builder 选项**

1. 在 Tools 菜单上，单击 J/Direct Call Builder 选项。
2. 在 J/Direct Call Builder Options 对话框中，选择或清除以下设置：

| 选项 | 说明 |
| --- | --- |
| Show Target window when copy to target | 默认状态为选择。当在类中插入 J/Direct 调用时，编译器会自动在 Text 编辑器中打开 Java 文件（如果它没有打开的话） |
| Disable stack crawl security check | 默认状态为清除。VM 将对每个 J/Direct Check 调用启动堆栈安全检查。如果栈上的任何调用程序不是完全信任的话，则发出安全异常，然后就不再调用 J/Direct 调用。这主要影响 Web 页上的 Java 代码。<br>当选择这个选项来关闭安全检查时，@security（check DLLCalls=off）标记自动添加到下次插入 J/Direct 调用的类中（如果清除了这个选项，则任何现有@security 标记依然在文件中，但将不会添加新的实例） |

3. 单击 OK，保存设置。

# 快速语法参考

下面介绍@dll.import、@dll.struct和@dll.structmap伪指令的快速参考。对每个伪指令，都给出并解释所需的句法。

## @dll.import的语法

@dll.import伪指令应位于方法声明的上边。伪指令句法如下：

```
/**@dll.import("LibName",<Modifier>)*/
...method declaration...;
```

Libname是DLL的名称，它包含了要调用的函数。Modifier是可选项，其值可根据需要确定。在方法声明中，可以使用DLL使用的函数名，或通过使用别名来给出方法的不同名称。关于别名的信息，请参看本章后面的“别名（方法重命名）”一节。为参数和方法返回值选择的Java类型应该是映射到DLL函数的参数和返回值的类型。关于Java数据类型如何映射到本机类型的更多信息，请参看本章后面的“如何安排数据类型”一节。

下表给出了本节说明的几种形式的@dll.import句法。

| 情形 | 句法 | 解释 |
|---|---|---|
| 调用Win32 DLLs | /**@dll.import("Libname")*/ | |
| 调用OLE APIs | /**@dll.import("Libname", ole)*/ | |

| | | |
|---|---|---|
| 别名使用 | /**@dll.import("Libname", entrypoint="DLLFunctionName") */ | 在方法声明中，使用选择的 Java 名称 |
| 按 Ordinal 链接 | /**@dll.import("Libname", entrypoint="#ordinal")*/ | ordinal 是 16 位整数，以 10 进制的方式给出，指出正在导入的 DLL 函数 |

## @ dll.struct的句法

@dll.struct 伪指令应位于类声明的上边。@dll.struct 的句法是：

/**@dll.struct(<*LinkTypeModifier*>,>pack=*n*>)*/

... class declaration... ;

LinkTypeModifier 告诉编译器，在本机格式中，字符串和字符类型的字段是否表示 ANSI 或 Unicode 字符。LinkTypeModifier 可以是 ansi、unicode 或 auto。如果没有指定 LinkTypeModifier，将使用默认的 ANSI。参看本章后面的“VM 如何在 ANSI 和 Unicode 之间选择”一节，可以帮助了解关于可用 LinkTypeModifier 值的更多信息。
也可以指定 pack=n 来告诉编译器，设置结构的紧缩尺寸为 1，2，4 或 8，这主要依赖于指定的 n 值。如果省略了 pack=n，则紧缩尺寸默认为 8。关于设置紧缩尺寸的更多信息，请参看本章后面的“结构紧缩”一节。
在类声明中，需要提供字段，此字段的 Java 类型以它们在本机结构中

出现的次序映射为本机结构中字段的类型。关于选择字段数据类型的信息，请参看本章后面的“结构中类型之间的一致性”一节。
下表给出了可能使用的@dll.struct伪指令的几种形式的句法。

| 情形 | 需要的句法 | 解释 |
| --- | --- | --- |
| 声明结构 | /**@dll.struct()*/ | 在不指定 LinkTypeModifier 时，假定 char 或 String 字段表示 ANSI 字符 |
| 设置结构的紧缩尺寸 | /**@dll.struct(pack=n)*/ | n 可以是 1，2，4 或 8 |
| 声明具有字段或类型字符的结构 | /**@dll.struct(ansi)*/ | 字符字段表示一个 ANSI 字符 |
| 使用字段或类型字符串声明一个结构，并设置紧缩尺寸 | /**@dll.struct(unicode.pack=n)*/ | String 字段将以 Unicode 的格式，而且，紧缩尺寸根据 n 值设置为 1，2，4 或 8 |

## dll.structmap的语法

@dll.structmap 伪指令用于声明在结构中嵌入的定长字符串和数组。它位于使用@dll.struct声明的结构中，定长字符串或数组字段声明的上边。关于如何使用@dll.structmap 的更多信息，请参看本章后面的两节“在结构中嵌入定长字符串”和“在结构中嵌入定长标量数组”。
下表给出了@dll.structmap伪指令的句法：

| 情形 | 需要的句法 | 解释 |
| --- | --- | --- |
| 包含定长字符串的结构 | //**@dll.structmap([type=TCHAR[Size]])*/ | size 是十进制整数，它指出字符串中的字符数，包括空字符终止的空格 |
| 包含定长数组的结构 | //**@dll.structmap([type=FIXE-DARRAY,size=n])*/ | n 是实进制整数，它表示数组的大小 |

## 数据类型转换

Microsoft VM 计算 Java 参数，然后把它们转换成本机 C++类型。Microsoft VM 根据编译时声明的 Java 参数类型，推断每个参数和返回值的本机类型。例如，已声明为 Java 整数的参数，是作为 32 位整数传送的，已声明为 Java 字符 String 对象的参数，是作为空字符结尾的字符串传送的，等等。这没有不可见的属性来提供本机类型的信息。在 Java 中，所见即所得。

## 快速参考

下面两个表列出了每个 Java 类型对应的本机类型。第一张表描述了参数与返回值的映射，第二张表展示了@dll.struct 伪指令使用的映射。

**参数与返回值映射**

| Java | 本机 | 说明/限制 |
| --- | --- | --- |
| byte | BYTE 或 CHAR | |
| short | SHORT 或 WORD | |
| int | INT, UINT, LONG, ULONG, DWORD | |
| char | TCHAR | |
| long | __int64 | |
| float | float | |
| double | double | |
| Boolean | BOOL | |
| String | LPCTSTR | 除了 ole 模式外，不允许作为返回值。在 ole 模式中，String 映射为 LPWSTR。Microsoft VM 通过使用 CpTaskMemFree 来释放字符串 |
| byte[] | BYTE* | |
| short[] | WORD* | |
| char[] | TCHAR* | |
| int[] | DWORD* | |
| long[] | float* | |
| double[] | double* | |

| | | |
|---|---|---|
| long[] | __int64* | |
| Boolean[] | BOOL[] | |
| Object | 结构指针 | 在 ole 模式中，是传递一个 IUnknown* |
| Interface | COM 接口 | 使用 jactivex 或类似的工具生成接口文件 |
| com.ms.com.SafeArray | SAFEARRAY* | 不允许作为返回值 |
| com.ms.com._Guid | GUID, IID, CLSID | |
| com.ms.com.Variant | VARIANT* | |
| @dll.struct 类 | 结构的指针 | |
| @com.struct 类 | 结构的指针 | |
| void | VOID | 只作为返回值 |
| com.ms.dll.Callback | 函数指针 | 只作为参数 |

## 使用 @dll.struct 映射

| Java | 本机 |
|---|---|
| byte | BYTE 或 CHAR |
| char | TCHAR |
| short | SHORT 或 WORD |
| int | INT，UINT，LONG，ULONG 或 DWORD |

| | |
|---|---|
| float | _ _int64 |
| long | float |
| double | double |
| Boolean | BOOL[] |
| Java | Native |
| String | 字符串的指针，或嵌入的定长字符串 |
| 使用@dll.struct 标记的类 | 嵌入的结构 |
| char[] | 嵌入的 TCHAR 的数组 |
| byte[] | 嵌入的 BYTE 的数组 |
| short[] | 嵌入的 SHORT 的数组 |
| int[] | 嵌入的 LONG 的数组 |
| long[] | 嵌入的_ _int64 的数组 |
| float[] | 嵌入的 float 的数组 |
| double[] | 嵌入的 double 的数组 |

## 基本标量类型

下表给出了映射的基本标量类型：

| **Java 类型** | **DLL** |
|---|---|
| int | 有符号 32 位整数 |
| byte | 有符号 8 位整数 |
| short | 有符号 16 位整数 |

| | |
|---|---|
| long | 有符号 64 位整数 |
| float | 32 位浮点数 |
| double | 64 位双精度 |

注意，Java 中，没有无符号整型的直接表示。由于 Java 没有无符号类型，则可使用与整数类型相同长度的 Java 类型。例如对于常见的 DWORD 类型（无符号 32 位），可以使用 Java int 类型（如果在表示数据中没有什么丢失的话）。

## 字符型

如果 *unicodle* 或 *ole* @dll.struct 修饰符是无效的，则 Java 字符类型是 CHAR（8 位 ANSI 字符），在另一种情况中，它是 WCHAR（16 位 Unicode 字符）。

## 布尔值

Java Boolean 类型映射于 Win32 BOOL 类型，是 32 位类型，作为参数，Java 真为 1，假为 0，作为返回值，则非 0 为真。
注意，BOOL 和 VARIANT_BOOL（Microsoft Visual Basic 中的内部 Boolean 类型）是不能互换的。为了把 VARIANT_BOOL 传送到 Visual Basic DLL，必须使用 Java short 类型，使用-1 代表 VARIANT_TRUE，0 代表 VARIANT_FALSE。

# 字符串

本节解释在 ANSI 和 Unicode 格式中，如何把字符串传送到 DLL 函数，本节还讨论从 DLL 函数返回字符串的两种方法。

## 把字符串传送到 DLL 函数

为了把标准的空字符结尾的字符串传送到 DLL 函数，只传送 Java String。
例如，为了改变当前目录，可以如下访问 Kernel32 函数 CopyFile：

```
class showCopyFile
{
    public static void main(String args[])
    {
        CopyFile("old.txt", "new.txt", true);
    }
    /** @dll.import("KERNEL32") */
    private native static boolean CopyFile(String existingfile, String newFile, boolean f);
}
```

在 Java 中，字符串是只读的，所以 Microsoft VM 将只把 String 对象作为输入。为了允许虚拟机实现来转换字符串，而不用拷贝字符，String 对象参数不能传递到可修改字符串的 DLL 函数。如果 DLL 函数可以修改字符串，就可传递 StringBuffer 对象。

如果不使用 Unicode 或 ole 修饰符（在这种情况下，字符串以打算以 Unicode 格式传递），则字符串将转换为 ANSI。
除非在 ole 模式中，否则，字符串不能作为 DLL 函数的返回类型声明。在 ole 模式中，本机返回类型是使用 CoTaskMemAlloc 函数指定的 LPWSTR。

## 从 DLL 函数接收字符串

从函数中传回字符串有两种方法：一种是调用程序分配函数填充的缓冲区，另一种是函数分配字符串，然后把它返回给调用程序。大多数 Win32 函数使用第一种方法，但 OLE 函数使用第二种方法（参看本章后面的“调用 OLE API 函数”，可以了解 J/Direct 为调用 OLE 函数提供的特殊支持）。使用第一种方法的函数是 Kernel32 函数 GetTempPath,它具有如下原型：

```
DWORD GetTempPath(DWORD sizeofbuffer, LPTSTR buffer);
```

此函数只返回系统暂时文件目录的路径（诸如“c:\tmp\”）。Buffer 参数指向一个调用程序分配的缓冲区，在这个缓冲区中将接收路径。sizeofbuffer 指出了写入缓冲区的字符数（这不同于 Unicode 版本中的字节数）。在 Java 中，字符串是只读的，所以不能把字符 String 对象作为缓冲区来传递。相反，可以使用 Java 的 StringBuffer 类来创建可写的 StringBuffer 对象。下面是调用 GetTempPath 函数的例子：

```
class ShowGetTempPath
```

```
{
    static final int MAX_PATH = 260;
    public static void main(String args[])
    {
        StringBuffer temppath = new StringBuffer(MAX_PATH);
        GetTempPath(temppath.capacity(), temppath);
        System.out.println("Temppath = " + temppath);
    }
    /** @dll.import("KERNEL32") */
    private static native int GetTempPath(int sizeofbuffer, StringBuffer buffer);
}
```

要理解本例，区别 StringBuffer 的长度和容量是很重要的。长度是指在 StringBuffer 中当前存储字符串的逻辑字符个数。容量是指 StringBuffer 当前分配的实际存储量。执行下列语句之后：

```
Stringbuffer sb = new StringBuffer(259);
```

Sb.length 的值是 0，sb.capacity 的值是 259。当调用一种 DLL 方法传递 StringBuffer 时，Microsoft VM 检查 StringBuffer 的容量，并为空终止符将这个容量加 1，如果 Unicode 是默认字符长度，则将容量乘以 2，然后为传递到 DLL 函数的缓冲区分配多个字节的内存。换言之，可以使用容量，而不是长度，来设置缓冲区的尺寸。注意，只要不犯以下错误：

```
StringBuffer sb = new StringBuffer();   //Wrong!
    GetTempPath(MAX_PATH, sb);
```

不带参数调用 StringBuffer 构造器，创建容量为 16 的 StringBuffer 对象，此容量可能太小了。对 GetTempPath 方法传递 MAX_PATH，指出在缓冲区中是有足够的空间来容纳 260 个字符。这样，GetTempPath 缓冲区可能会溢出。如果想大量使用 GetTempPath，应该以如下方法在 Java 友好的封装器中封装它：

```
public static String GetTempPath()
{
    StringBuffer temppath = new StringBuffer(MAX_PATH-1);
    int res = GetTempPath)MAX_PATH, temppath);
    if (res == 0 || res > MAX_PATH) {
        throw new RuntimeException("GetTempPath error!");
    }
    return temppath.toSrring(); // can't return a StringBuffer
}
```

这种方法既方便又安全，并且它映射了 Java 异常下函数的错误返回值，注意，不能返回 StringBuffer 对象。

## 数组

J/Direct 自动处理标量的数组。下面的 Java 数组类型直接转换到本机指针类型：

| Java | 本机 | 每个元素的字节数 |
|---|---|---|
| Byte[] | BYTE* | 1 |
| Short[] | SHORT* | 2 |
| int[] | DWORD* | 4 |
| float[] | FLOAT* | 4 |
| double[] | DOUBLE* | 8 |
| long[] | __INT64* | 8 |
| Boolean[] | BOOL* | 4 |

如果 Unicode 修饰符无效，则 char[ ]数组类型转换为 CHAR*，如果有效，则转换为 WCHAR*。

通过调用程序，可以修改所有的标量数组参数（诸如[in，out]参数）。

数组类型不能作为返回类型。这里不支持对象或字符串数组。

通常，这种机制由 OLE 函数用来返回值（OLE 函数保留“函数”返回值来返回 HRESULT 错误代码）。参看本章后面的“调用 OLE API 函数”一节，会了解到 OLE 函数是如何获得返回值的。

## 结构

Java 语言不直接支持结构的概念。尽管使用 Java 类包含的字段可以在 Java 语言中模拟结构的概念，也不能使用普通的 Java 对象在本机 DLL 调用中模拟结构。这是因为 Java 语言不能保证字段的布局，因为无用单元收集程序可以在内存中自由移动对象。
然而，为了传递和接收来自 DLL 方法的结构，需要使用@dll.struct 编译器伪指令。当对 Java 类定义应用它时，这个伪指令将对内存块中分配的类产生类的所有实例，此内存块在无用单元回收期间是不可移动的。另外，使用 pack 修饰符，可以控制内存中字段的格式（参看本章后面的“结构封装”）。例如，Win32 SYSTEMTIME 结构，在 C 编程语言中有如下定义：

```
typedef struct {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME;
```

在 Java 中，此结构正确的声明如下：

```
/** @dll.struct() */
class SYSTEMTIME {
    public short wYear;
    public short wMonth;
    public short wDayOfWeek;
    public short wDay;
    public short wHour;
    public short wMinute;
    public short wSecond;
    public short wMilliseconds;
}
```

下面的例子在 DLL 方法调用中使用 SYSTEMTIME 结构：

```
class ShowStruct {
    /** @dll.import("KERNEL32") */
    static native void GetSystemTime(SYSTEMTIME pst);
     public static void main(String args[])
    {
        SYSTEMTIME systemtime = new SYSTEMTIME();
        GetSystemTime(systemtime);
        System.out.println("Year is " + systemtime.wYear);
```

```
        System.out.println("Month is " + systemtime.wMonth);
        // etc.
    }
}
```

**注意：**已声明的带有@dll.struct 的类是不安全的，不能被不信任的小应用程序使用。

## 结构内部各类型间的一致性

下表描述了在结构内部标量类型是如何映射的。

| Java | 本机 |
|---|---|
| byte | BYTE |
| vhar | TCHAR(CHAR 或 WCHAR，这取决于@dll.struct 定义) |
| short | SHORT |
| int | LONG |
| long | _ _int64 |
| float | float |
| double | double |
| boolean | BOOL(32 位 Boolean) |

引用类型（Java 对象和类）通常映射到嵌套结构和数组。下表给出各种支持的映射。

| Java | 本机 |
| --- | --- |
| string | 指针或字符串，或嵌套的定长字符串 |
| 使用@dll.struct标记的类 | 嵌套结构 |
| char[] | TCHAR（CHAR/WCHAR）的嵌套数组 |
| byte[] | BYTE的嵌套数组 |
| short[] | SHORT的嵌套数组 |
| int[] | LONG的嵌套数组 |
| long[] | __int64的嵌套数组 |
| float[] | 浮点的嵌套数组 |
| double[] | 双精度的嵌套数组 |

由于有许多种方法可完成引用对象的分配和清理，所以，这里不直接支持结构内部的指针。为了表示带嵌套指针的结构，需声明此指针字段为整数类型。需要对适当的分配函数生成显式DLL调用，然后初始化内存块（可以使用DllLib.ptrToStruct将这块内存映射到@dll.Struct类）。

## 嵌套的结构

结构可通过命名其他结构作为字段类型，来简单地嵌套另一个结构。例如，如下将Windows MSG结构嵌套一个POINT结构：

```
typedef struct {
    LONG x;
    LONG y;
```

```
} POINT;

typedef struct {
    int hwnd;
    int message;
    int wParam;
    int lParam;
    int time;
    POINT pt;
} MSG;
```

下面直接翻译进 Java：

```
/** @dll.Struct() */
class POINT {
    int x;
    int y;
}
/** @dll.struct() */
class MSG {
    public int hwnd;
    public int message;
    public int wParam;
    public int lParam;
```

```
    public int time;
    public POINT pt;
}
```

**提示：**尽管嵌套结构是很方便的，但实际上，Java 并不真正支持嵌套的对象，而只支持对象的嵌套引用。Microsoft VM 必须在每次传递嵌套结构时在两种格式之间进行转换。然而，在关键性的代码路径中，可以通过人为嵌套结构来改善性能（通过把嵌套结构的字段拷贝到包含结构中）。例如，MSG 结构中的 *Pt* 字段可以很容易地声明为两个整型字段，Pt_x 和 Pt_y。

## 在结构中嵌套定长字符串

在一些结构中可嵌套定长字符串。LOGFONT 结构定义如下：

```
typedef struct {
        LONG lfHeight;
        LONG lfWidth;
        /* <many other fields deleted for brevity> */
        TCHAR lfFaceName[32];
    } LOGFONT;
```

在 Java 中使用展开的句法指定长度来描述这种结构：

```
/** @dll.struct() */
```

```
class LOGFONT {
    int lfHeight;
    int lfWidth;
    /* <many other fields deleted for brevity> */
    /** @dll.structmap([type=TCHAR[32]]) */
    StringlfFaceName;
}
```

@dll.strutmap 伪指令指明了固定字符串的长度作为字符的测量尺度（对空字符结尾的串，包括空格）。
更多的信息请参看“在结构中嵌套定长标量数组”。

## 在结构中嵌套定长标量数组

使用@dll.structmap 伪指令，可以指定在结构中嵌套的定长标量数组。下面是包括定长标量数组的 C 语言结构：

```
struct EmbeddedArrays
{
BYTE            b[4]
CHAR            c[4]
SHORT           s[4]
INT             i[4]
__int64         l[4]
float           f[4]
```

```
doubl           d[4]
};
```

在下面的方法中，通过使用@dll.structmap可指定EmbeddedArrays结构。

```
/** @dll.struct() */
class EmbeddedArrays
{
    /** @dll.structmap([type=FIXEDARRAY,size=4]) */
    byte b[];

    /** @dll.structmap([type=FIXEDARRAY,size=4]) */
    char c[];

    /** @dll.structmap([type=FIXEDARRAY,size=4]) */
    short s[];

    /** @dll.structmap([type=FIXEDARRAY,size=4]) */
    int i[];

    /** @dll.structmap([type=FIXEDARRAY,size=4]) */
    long l[];

    /** @dll.structmap([type=FIXEDARRAY,size=4]) */
    float f[];

    /** @dll.structmap([type=FIXEDARRAY,size=4]) */
```

```
    double d[];
}
```

## 结构装填

在ANSI C草案3.5.2.1中，说明了结构字段的填充和对齐。可通过使用pack修饰符来设置装填长度：

```
/** @dll.struct(pack=n) */
```

这里的n可为1，2，4或8。默认为8。对于Microsoft Visual C++编译器的用户，“pack=n”等价于“#pragma pack(n)”。

## 理解@dll.struct和@com.struct之间的关系

@dll.struct伪指令与@com.struct伪指令非常类似，这个伪指令包括在jactivex工具并隐含包括在Javatlb工具中（Javatlb工具包含在以前的Microsoft Visual J++版本中，现在已经被jactivex所代替）。最重要的不同点是：默认类型映射适于Mcirosoft Windows函数调用，而不是COM对象调用。这样，就可以在类型库中描述这个结构，并使用jactivex工具生成Java类，以生成@dll.struct类。然而，通常情况下，手工生成类更快些。

# 指针

Java 不支持指针数据类型。然而，用户可以传递一个元素的数组，而不是传递指针。可以在 Java 整数中存储指针，然后从原始指针中读写数据。

## 返回值指针

通常，具有多个返回值的 Win32 函数可通过调用程序给更新的变量传递指针来进行处理。例如，GetDiskFreeSpace 函数具有如下原型：

```
BOOL GetDiskFreeSpace(LPCTSTR szRootPathName,
                      DWORD   *lpSectorsPerCluster,
                      DWORD   *lpBytesPerCluster,
                      DWORD   *lpFreeClusters,
                      DWORD   *lpClusters);
```

GetDiskFreeSpace 的典型调用如下：

```
DWORD sectorsPerCluster, bytesPerCluster, freeClusters, clusters;
GetDiskFreeSpace(rootname, &sectorsPerCluster,
                 &bytesPerCluster, &freeClusters, &clusters);
```

在 Java 中，这是传递元素标量数组的典型例子。下例显示了如何调用 GetDiskFreeSpace 函数：

```
class ShowGetDiskFreeSpace
{
    public static void main(String args[])
    {
        int sectorsPerCluster[] = {0};
        int bytesPerCluster[] = {0};
        int freeClusters[] = {0};
        int clusters[] = {0};
        GetDiskFreeSpace(c:\\", sectorsPerCluster" bytesPerCluster,
                                freeClusters, clusters);
        System.out.println("sectors/cluster = "+ sectorsPerCluster[0]);
        System.out.println("bytes/cluster = "+bytesPerCluster[0]);
        System.out.println("free clusters = "+freeClusters[0]);
        System.out.println("clusters= + "clusters[0]);
    }
     /** @dll.import(KERNEL32)*/
    private native static boolean GetDiskFreeSpace(String rootname,
                        int pSectorsPerCluster[], int pBytesPerCluster[],
                        int pFreeClusters[], int pClusters[]);
}
```

## 原始指针

未知的或难以实现的结构指针可存放于普通的 Java 整数中，如果应用程序只需存储指针，而不需要引用它，那就是是最简单且最有效的方法。可以使用此技术来存储分配内存块的 DLL 函数返回的指针。实际上，可以使用这种技术来存储任何 DLL 函数返回的指针。不用说，使用原始指针消除了许多 Java 安全性的优点。只要可能，都应使用其他方法。然而，还是有可能选择使用原始指针的情况。记住，从原始指针中读写数据有两种方式。

## 转换引用为 @Dll.Struct 类

通过原始指针中读写数据的一种方法是将原始指针转换为对 @dll.struct 类的引用。一旦完成了这一步，就可以使用正常字段访问句法来读写数据。例如，假设有一个要访问 RECT 的原始指针。可以使用系统方法 DllLib.ptrToStruct 如下：

```
/** @dll.struct() */
    class RECT {
       int left;
       int top;
       int right;
       int bottom;
    }
```

```
import com.ms.dll.*;
int rawptr = ... ;
RECT rect = (RECT)DllLib.ptrToStruct(RECT.class.rawptr);
rect.left = 0;
rect.top = 0;
ret.right = 10;
rect.bottom = 10;
```

在 RECT 实例中，ptrToStruct 方法封装原始指针。这个 RECT 实例与 New 运算符创建的实例不同，它不会通过无用单元收集来释放原始指针，因为 RECT 对象没有办法知道指针的位置。另外，由于本机内存在调用 ptrToStruct 时已经构成，所以不调用 RECT 类构造器。

## 使用 DllLib 拷贝方法

从原始指针中读写数据的另一种方法是在 DllLib 中使用重载拷贝方法。这些方法在各种类型的 Java 数组和原始指针之间拷贝数据。如果需要把原始指针作为指向字符串的指针（LPTSTR），可以使用 DllLib 方法 ptrToStringAnsi，ptrToStringUni 或 ptrToString 来从句法上分析字符串，并把它转换成 java..lang.String 对象。

```
Import com.ms.dll.*;
int rawptr = ... ;
String s = dllLib.ptrToStringAnsi(rawptr);
```

**警告：**所有 Java 对象都在内存中移动，或被无用单元收集程序回收。所以，不能试图通过调用具有常规类型转换的 DLL 函数来为 Java 数组获得指针。下例给出了获得指针的一种不正确的方法。

```
// Do not do this!
    /** @dll.import("MYDLL") */
    private native static int Cast(int javaarray[]);
    // Inside MYDLL.DLL
    LPVOID Case(LPVOID ptr)
    {
        // Do not do this!
        Return ptr; // comes in as a Java array; goes out as a Java int
    }
```

ptr 值只在调用 Cast 函数期间才能保证是有效的。这是因为允许 VM 实现通过拷贝来执行数组的传递，并且因为在对 Cast 函数的调用返回后，无用单元回收可能会使数组的物理存储位置变得不同。

## 多态参数

一些 Win32 函数声明参数的类型依赖于另一个参数的值。例如，WinHelp 函数声明如下：

```
BOOL WinHelp(int hwnd, LPCTSTR szHelpFile, UINT cmd, DWORD dwData);
```

看似简单的 dwData 参数实际上可以为以下几种形式：指向字符串的指针，指向 MULTIKEYHELP 结构的指针，指向 HELPWININFO 指针或普通整数，这取决于 cmd 参数的值。
J/Direct 提供两种方法来声明这种参数：

- 声明此参数作为 Object 类型。
- 对每种可能的类型，使用过载来声明分离方法。

关于这两种方法的比较，请参看本章后面的“两种方法的比较”。

## 声明参数为 Object 类型

下面给出了如何通过声明 dwData 为 Object 类型来声明 WinHelp。

```
/** @dll.import("USER32") */
static native boolean WinHelp(int hwnd, String szHelpFile,
                              int cmd, Object dwData);
```

当调用 WinHelp 时，J/Direct 将使用运行时间类型来决定如何传递 dwData。下表描述了类型是如何转换的。

| 类型 | 转换为 |
|---|---|
| java.lang.Integer | 4 字节整数 |
| java.lang.Boolean | 4 字节 BOOL |
| java.lang.Char | CHAR（如果 unicode 或 ole 修饰符有效，则是 WCHAR） |

| | |
|---|---|
| java.lang.short | 2 字节 SHORT |
| java.lang.Float | 4 字节 FLOAT |
| java.lang.String | LPCSTR(如果 unicode 或 ole 修饰符有效，则是 LPCWSTR) |
| java.lang.StringBuffer | LPSTR(如果 unicode 或 ole 修饰符有效，则是 LPWSTR) |
| byte[] | BYTE* |
| char[] | CHAR*(如果 unicode 或 ole 修饰符有效，则是 WCHAR) |
| short[] | SHORT* |
| int[] | INT*_ _int64 |
| float[] | float* |
| double[] | double* |
| @dll.struct | 结构指针 |

## 重载函数

声明 WinHelp 函数的另一种方法是对每种可能的类型进行重载函数：

```
/** @dll.import("USER32") */
static native boolean WinHelp(int hwnd, String szHelpFile,
                              int cmd, int dwData);

/** @dll.import("USER32") */
static native boolean WinHelp(int hwnd, String szHelpFile,
```

```
                                int cmd, String dwData);

/** @dll.import("USER32") */
static native boolean WinHelp(int hwnd, String szHelpFile,
                                int cmd, MULTIKEYHELP dwData);

/** @dll.import("USER32") */
static native boolean WinHelp(int hwnd, String szHelpFile,
                                int cmd, HELPWININFO dwData);
```

使用重载技术不能处理多态返回值，因为在单一返回值上不能重载 Java 方法。因此，必须给函数的每种变体起不同的 Java 名称，然后使用 entrypoint 修饰符来把它们链接到相同的 DLL 方法。要了解其他 DLL 方法的更多信息，请参看本章后面的“别名（方法重命名）”。

## 两种方法的比较

在大多数实例中，优先考虑重载方法，因为它运行性能高，还有较好的类型检查。另外，重载避免了在 Integer 对象内部封装整数参数。
然而，在有多个多态参数的情况中，声明参数为类型 Object 是很有用的。当要访问大量类型的服务时，诸如可以接受使用 @dll.struct 伪指令声明的任意对象的函数，也可以选择这种方法。

# 回调

几种 Win32 API 要求程序提供具有回调函数地址的 Windows。在 Java 中，可以通过展开系统类 com.ms.dll.callback 来写回调函数。

## 声明带有回调的方法

在 Java 中，为了表示回调参数，需声明 Java 类型为类型 com.ms.dll.Callback 或从中派生出来的类。例如，Microsoft Win32 EnumWindows 原型化如下：

```
BOOL EnumWindows(WNDENUMPROC wndenumproc, LPARAM lparam);
```

对应的 Java 原型为：

```
import com.ms.dll.Callback;
    /** @dll.import("USER32") */
    static native boolean EnumWindows(Callback wndenumproc, int lparam);
```

## 调用带回调的函数

为了调用带回调的函数，必须定义展开的类为 Callback。派生类必须公布一个非静态的方法，它的名称是 callback（全部小写）。WNDENUMPROC 的 C 语言定义如下：

```
BOOL CALLBACK EnumWindowsProc(HWND hwnd, LPARAM lparam);
```

为了在 Java 中创作 EnumWindowsProc，必须声明展开 Callback 的类，如下：

```
class EnumWindowsProc extends Callback
  {
    public boolean callback(int hwnd,int lparam)
    {
        StringBuffer text = new StringBuffer(50);
        GetWindowText(hwnd, text, text.capacity()+1);
          if (text.lengh() != 0){
        System.out.println("hwnd =" + Integer.toHexString(hwnd) +"h: Text =" + text);
        }
        return true; // return TRUE to continue enumeration.
    }
/** @dll.import("USER32") */
private static native int GetWindowText(int hwnd,StringBuffer text,int cch);
}
```

可以使用如下方法调用带有 callback 的 EnumWindows：

```
boolean result = EnumWindows(new EnumWindowsProc(), 0);
```

## 回调方法接受类型的限制

回调方法的返回类型必须是 void、int、boolean 或 short。当前允许的唯一参数类型是整型。幸运的是，这并不像听起来有那么多限制。可以使用 DllLib 方法 ptrToStringAnsi，ptrToStringUni 和 ptrToString 把参数作为 LPTSTR。可以使用 ptrToStruct 方法把参数作为指向 @dll.struct 类的指针。

## 用回调关联数据

经常需要把一些数据从函数的调用程序传到回调程序。这解释了 EnumWindows 带有额外的 lparam 参数的原因。大多数带有回调的 Win32 函数都有无条件接受传到回调函数的特殊的 32 位参数。使用回调机制，可以不必使用 lparam 参数来传递数据。因为回调方法是非静态的，所以可以在 EnumWindowsProc 对象中把数据作为字段来存储。

## 回调的生存期

需要注意，要完成本机函数之前，无用单元回收程序不回收回调。如果回调是短期的（只对函数调用的生存期可用），则不需要特殊的行为，因为对 DLL 函数传递的回调可以保证在调用过程中不必由无用单元回收程序来回收。

如果回调是长期的（整个函数调用过程中都使用），将必须在 Java 数据结构中通过存储它的引用防止回调被回收。也可以在本机数据结构中通过使用 com.ms.dll.root 类在根句柄内封装此回调，来存储回调的引用。

根句柄是 32 位句柄，它可以防止回调被回收，直到句柄显式释放为止。例如，WndProc 的根句柄可存储在 HWND 结构的应用数据区中，并且在 WM_NCDESTROY 消息中显式释放。

## 在结构内部嵌套回调

为把回调嵌套到结构内部，必须先调用 com.ms.dll.Root.alloc 方法以在根句柄中封装回调。然后把根句柄传递到 DllLib.addrOf 方法，以获得回调的实际（本机）地址。然后把这个地址作为整数存储。
例如，在 Java 中如下声明 WNDCLASS 结构：

```
/** @dll.struct()   */
    class WNDCLASS {
        int style;
        int lpfnWndProc;; // CALLBACK
        ... /*   <other fields deleted for brevity> */
    }
```

假设已经展开回调如下：

```
class WNDPROC extends Callback
    {
        public int callback(int hwnd, int msg, int wparam, int lparam)
        {
        ...
```

```
        }
    }
```

为了在WNCLASS对象内部存储指向回调的指针，使用如下序列：

```
import com.ms.dll.*;
        WNDCLASS wc = new WNDCLASS();
        int callbackroot =    Root.alloc(new WNDPROC());
        Wc.lpfnWndProc = DllLib.addrOf(callbackroot);
```

## 调用OLE API函数

本节提供关于OLE API函数的信息。

## OLE 模式句法

@dll.import伪指令包括为导入OLE API函数而设计的特殊模式。为使用这种模式，在下面的例子中只包括ole修饰符：

```
/** @dll.import("OLE32", ole) */
    public class OLE32 {
        ...
    }
```

## OLE函数与Win32函数的比较

理论上，OLE32.DLL 和 OLEAUT32.DLL 的导出函数不同于其他任何 DLL 函数，实际上，OLE 函数具有自己一致的调用风格，在以下几个方面，OLE 函数不同于 Win32 函数：

- OLE 函数只在 Unicode 中出现，这里没有 ANSIOLE 函数。
- 实际上，所有 OLE 函数都通过标准函数返回值来返回 32 位状态码，即 HRESULT。高位指出函数是成功（高位关闭）还是失败（高位打开）。少量函数可返回多个成功值（通常有 S_OK 和 S_FALSE），但大多数只返回一个成功值（S_OK）。
- 如果 OLE 函数除状态码外还返回多个值，则调用程序应提供指向变量的指针，这个变量将接收退出值。通常，返回值指针作为最后的参数。
- 返回字符串的 Win32 结构是填充在调用程序分配的缓冲区中。OLE 函数通过使用 CoTaskMemAlloc 来返回字符串，并期望调用程序使用 CoTaskMemFree 来释放它们。

## 比较Win32代码与OLE代码

简单 Add 函数的代码看起来像 Win32 编码结构中的代码：

```
int sum;
sum = Add(10, 20);
```

在 OLE 风格中，Add 函数如下：

```
HRESULT hr;
    int sum;
    hr = Add(10, 20, &sum);
    if (FAILED(hr)) {
        ... handle error..
    }
```

## 调用OLE函数

ole 模式利用一致性的编码风格为调用 OLE 函数提供一种友好的 Java 方法。从 Java 调用 OLE 风格的 Add 函数看起来像调用传统的 Win32 风格的函数。

```
/** dll.import("OLELIKEMATHDLL", ole)   */
    private native static int Add(int x, int y);
    int sum = Add(10, 20);
    // if we got here, Add succeeded.
```

## OLE模式的工作方式

由于 ole 修饰符，Microsoft VM 自动假设本机 Add 函数返回 HRESULT。VM 通知 Add 函数返回整数。当调用 Add 时，VM 自动分配整型的临

时变量，并插入指向它的指针作为第三个参数。本机 Add 函数返回后，VM 自动检查 HRESULT，并且，如果它指出了失败（高位打开），则发出类型 com.ms.com.ComFailException 的 Java 异常。如果 HRESULT 没有指出失败，则 VM 从它创建的临时变量中检索 Add 函数的真正返回值，并返回此值。

S_FALSE 与 Java/COM 的集成不同，它的返回值不会导致发出 ComSuccessException。如果要区别成功的结果，则必须使用正规的 DLL 调用模式，并把 HRESULT 作为整数返回值。

ole 模式修改 DLL 调用的语义概括如下：

1. 所有字符串和字符都假设为 Unicode。
2. 本机函数的函数返回值假设是 HRESULT，如果返回的 HRESULT 指出失败，则 Microsoft VM 发出 ComFailException。
3. 如果 Java 方法返回类型不是 void，则 Microsoft VM 将被假设本机函数通过指针返回附加结果，此指针是函数的最后一个参数。在调用获得附加返回值之后，VM 将提供这个指针的参数和引用，这个值将作为 Java 方法的值来返回。

## 传递和接收来自OLE函数的字符串

在 ole 模式函数上声明一个参数为 String 类型来传递 LPCOLESTR。Microsoft VM 也包括长度前缀，所以，这个字符串可以作为 BSTR。

在 ole 模式中声明返回值为 String 类型，可使 Microsoft VM 传递指向未

初始化的 LPCOLESTR 的指针。当本机函数返回时，Microsoft VM 将把返回的 LPCOLESTR 转换成字符串，然后调用 CoTaskFee 来释放字符串。

## 传递GUIDs（及IIDs和CLSIDs）

使用系统类 com.ms.com._Guids 来表示 GUIDs，作为参数传递_Guid 对象，也就是对本机函数传递了指向 GUID 的指针，声明_Guid 的返回类型，可使 Microsoft VM 传递指向函数填充的（只在 ole 模式中）未初始化的 GUID 的指针。

例如，OLE32 导出函数 CLSIDFromProgID 和 ProgIDFromCLSID，以在 CLSIDs 和通过 Visual Basic 函数 CreateObject 使用的可读名称之间进行映射。

这些方法具有如下原型：

```
HRESULT CLSIDFromProgID(LPCOLESTR szProgID, LPCLSID pclsid);
HRESULT ProgIDFromCLSID(REFCLSID clsid, LPOLESTR *lpszProgId);
```

在 Java 中，以如下方式声明这些方法：

```
import com.ms.com._Guid;
    class OLE {
        /** @dll.import("OLE32", ole) */
        public static native _Guid CLSIDFromProgID(String szProgID);
```

```
    /** @dll.import("OLE32", ole) */
    public static native String   ProgIDFromCLSID(_Guid clsid);
}
```

注意：com.ms.com._Guid 取代了 com.ms.com.Guid（去掉了下划线）。

## 传递VARIANTs

声明类型 com.ms.com.Variant 的参数来为本机函数传递指向 VARIANT 的指针。声明类型 com.ms.com.Variant（只对 ole 模式）的返回值，来为本机函数传递指向未被初始化的 Variant 的指针，以进行填充。

## 传递COM接口指针

为传递 COM 接口指针，必须使用诸如 jactivex.exe 之类的工具，来生成 Java/COM 接口类。然后，通过声明接口类型的参数来传递或接收 COM 接口。

例如，系统类 com.ms.com.IStream 是 Java/COM 接口。此接口代表了 structured Storage IStream *接口。OLE32 函数 CreatStreamOnHGlobal 的声明如下：

```
import com.ms.com.*;
/** @dll.import("OLE32", ole) */
```

```
public static native IStream CreateStreamOnHGlobal(int hglobal, boolean fDeleteOnRelease);
```

## 别名（方法重命名）

有时，可能想使用 Java 方法的名称，此名不同于导出函数时 DLL 使用的名称。例如，可能有以小写字母开头的 Java 名称，以便符合 Java 命名约定。为了做到这一点，在下例中使用带有 entrypoint 修饰符的 @dll.import 伪指令。

```
/** @dll.import("USER32", entrypoint="GetSysColor") */
    static native int getSysColor(int nIndex);
```

不需要用别名来执行 Win32 API 的 ANSI/Unicode 后缀。Microsoft VM 自动承担这些工作（请参看本节后面的“VM 如何在 ANSI 和 Unicode 之间进行选择”）。当访问通过 a.def 文件导出的函数时，也不必使用别名方式。通常这些名称是使用“破坏标准调用（stdcall manging）”的方式导出的。以下范例给出了_sample@8 重新命名的方法（这里的 8 表示函数接受的参数类型数量）：

```
extern "C"
    __declspec(dllexport)
    VOID sample(int x, int y){
        ...
    }
```

如果没有显式入口点，则 J/Direct 自动对破坏的标准调用（stdcall-mangled）名称进行绑定。Microsoft VM 自动为以下 KERNEL32 API 函数提供别名。

| Kernel32 Function | Alias |
|---|---|
| CopyMemory | RtlMoveMemory |
| MoveMemory | RtlMoveMemory |
| FillMemory | RtlFillMemory |
| ZeroMemory | RtlZeroMemory |

## 按序号链接

一些 DLL 是通过序号（16 位整数）而不是名称导出函数的。为了调用这样的 DLLs，需要使用方法级@dll.import 伪指令来确定序号。序号链接的句法是：

```
/** @dll.import.import("Libname", entryPoint="#ordinal") */
```

注意，ordinal 是以十进制形式指定的
例如，要以 DLL“MyDll.DLL”中的序号 82 链接导出的函数，其代码如下：

```
/** @dll.import("MyDll", entrypoint="#82") */
    public static native void MySample();
```

## 为整个类指定@dll.import

在类定义之前，可以使用@dll.import 伪指令来为所有在此类中声明的本机方法设置库名。以下声明是为整个类使用了@dll.import。

```
/** @dll.import ("KERNEL32") */
class EnvironmentStrings
{
    public static native int GetEnvironmentString();
    public static native int GetEnvironmentVariable(String name,
                                StringBuffer value, int ccbValue);
    pubic static native boolean SetEnvironmentVariable(String name,String value);
}
```

为每种方法指定@dll.import 是等效的，如下例：

```
class EnvironmentStrings
{
    /** @dll.import("KERNEL32") */
    public static native int GetEnvironmentStrings();
     /** @dll.import("KERNEL32") */
    public static native int GetEnvironmentVariable(String name,
                                StringBuffer value, int ccbValue);
```

```
    /** @dll.import("KERNEL32") */
    public static native boolean SetEnvironmentVariable(String name,String value);
}
```

在类级使用@dll.import 伪指令可以节省.class 文件中的空间，并消除了冗余的信息。然而，类级的@dll.import 伪指令不是通过子类来继承的。

## VM如何在ANSI和Unicode之间选择

考虑介绍 J/Direct 基础知识中 MessageBox 的例子，一个重要的事实是 USER32 并不会导出名为 MessageBox 的函数。因为 MessageBox 函数带有字符串，它（像所有 Win32 函数那样处理字符串）必须存在于两个版本中：ANSI 版本和 Unicode 版本（分别命名为 MessageBoxA 和 MessageBoxW）。当在 C 或 C++中调用 MessageBox 时，调用的 MessageBox“函数”实际上是一个宏，此宏根据是否定义了 UNICODE 宏而扩展到 MessageBoxA 或 MessageBoxW。

## 调用DLL函数的ANSI版本

默认情况下，Microsoft VM 假设 MessageBox 函数的 ANSI 版本是唯一需要的。如果使用@dll.import（不带修饰符）导入 MessageBox 函数，代码如下：

```
/** @dll.import("USER32") */
```

```
static native int MessageBox(int hwnd, String text, String   title, int style);
```

Microsoft VM 将采取如下步骤：

1. 字符串的“文本”和“标题”转换成 ANSI 空字符结尾的字符串。
2. VM 试图在 USER32.DLL 中找到名为 MessageBox 的出口。
3. 若此尝试失败，则 VM 在名称上追加“A”，然后寻找名为 MessageBoxA 的出口。
4. 若此尝试成功，则 VM 调用 MessageBoxA 函数。

## 调用 DLL 函数的 Unicode 版本

假设不想调用 MessgeBox 函数的 ANSI 版本，而是想调用 Unicode 版本，可以通过使用具有 @dll.import 伪指令的修饰符来完成。

```
/** @dll.import("USER32",unicode) */
    static antive int MessageBox(int hwnd, String text, String title, int style);
```

既然已给出了 Unicode 修饰符，则 Microsoft VM 将采取如下步骤：

1. 字符串的“文本”和“标题”转换成 Unicode 空字符结尾的字符串。
2. VM 试图在 USER32.DLL 中找到名为 MessageBox 的出口。
3. 若尝试失败，则 VM 在名称上追加“W”，然后寻找名为 MessageBoxW 的出口。
4. 若尝试成功，则 VM 调用 MessageBoxW 函数

## 调用DLL函数的最佳版本

不幸的是，调用DLL函数的ANSI版本或Unicode版本都是调用Win32函数的一种理想方法。使用默认ANSI模式允许代码在任何Win32平台上运行，但在全Unicode系统诸如Microsoft Windows NT上，会导致不必要的性能损失。使用Unicode修饰符可去除性能上的损失，但限制是只能在实现Unicode API的系统上运行。幸运的是，可以使用带有@dll.import伪指令的auto修饰符，来基于主操作系统调用DLL函数的最佳版本。

使用auto修饰符提供了这两种优点。下面的例子显示了如何调用MessageBox函数的最佳版本：

```
/** @dll.import("USER32",auto)   */
    static native int MessageBox(int hwnd, String text, String   title, int style);
```

当给定了auto修饰符时，Microsoft VM决定在运行期间基本平台是否支持Unicode APIs。如果支持Unicode，则VM的行为就像是已经指定了Unicode修饰符。否则，VM的行为就像是已经指定了ansi修饰符。这样，通过在给定的平台上使用最优的API集，auto修饰符就可以在ANSI和Unicode Windows系统上都能运行。

一般来说，无论何时调用Unicode API函数，都可以使用auto修饰符。如果调用自己的DLLs，则根据自己的需要选择ansi（默认）或Unicode。下表给出了当使用auto修饰符时VM如何决定使用ANSI或Unicode的细节：

1. VM 打开注册表键 HKEY_LOCAL_NACHINE\software\Microsoft\Java.VM，然后寻找 DWORD 命名的值 DllImportDefaultType，此值是下列几种之一：
   2—ANSI：一直使用 ANSI 版本。
   3—Unicode：一直使用 Unicode 版本。
   4—Platfort：根据平台确定使用 ANSI 或 Unicode。
2. 如果键不存在，或如果它已经设置为 4（表示平台），则 VM 调用 Win32 GetVersion 函数，然后检查高位来确定本平台是 Microsoft Windows 95，还是 Microsoft Windows NT。如果平台是 Windows 95，则使用 ANSI 模式，否则使用 Unicode 模式。

没有必要自己设置 DllImportDefaultType 注册表键。它基本上已存在了，所以安装程序可以在未来的 Windows 平台上设置最合适的选择。可以在平台上通过读取 com.ms.dll.dllLib.SystemDefaultCharSzie 字段，来编程查询首选模式。这个字段在 ANSI 系统上将设置为 1，在 Unicode 系统上将设置为 2。

Ansi, Unicode 和 auto 也可用于 @dll.struct 伪指令。

## 通过DLL函数获得错误代码

不调用 Win32 函数 GetLastError，而是通过另一个 DLL 调用，也可以获得错误代码。因为 Microsoft VM 可能在执行 Java 代码的过程中执行自身的函数调用，所以，在获得错误代码时，它可能已经被已经重写了。

为了可靠地通过 DLL 函数来访问错误代码，必须使用 setLastError 修饰符来指示 VM 在调用方法后立即捕获错误代码。由于性能的原因，这种行为不是默认的，随后可以调用 com.ms.dll.DllLib.getLastWin32Error 方法来检索错误代码，每个 Java 线程都独立保存这个值。
例如，FindNextFile 函数通过错误代码返回状态信息，FindNextFile 将声明如下：

```
/** @dll.import("KERNEL32",setLastError) */
    static native boolean FindNextFile(int    hFindFile, WIN32_FIND_DATA wfd);
```

典型的调用如下：

```
import com.ms.dll.DllLib;
boolean f = FindNextFile(hFindFile, wfd);
if (!f) {
    int errorcode = DllLib.getLastWin32Error();
}
```

## 动态加载和调用 DLL

有时，可能需要比 @dll.import 伪指令正常提供的更多控件来控制加载和链接过程。比如，需求可能包括：

- 加载库，它的名称和路径必须是在运行时计算的或从用户输入产生

的。

- 在处理终止之前释放库。
- 执行函数，它的名称和顺序必须在运行时计算。

Win32 APIs 一直提供了控制加载和链接的能力。LoadLibrary，LoadLibraryEx 和 FreeLibrary 函数显式控制 DLL 的加载和释放。GetProcAddress 函数允许链接到特定的出口。GetProcAddress 函数返回一个函数指针，所以，可通过函数指针调用任何语言，这样可以实现动态调用而毫无问题。

J/Direct 允许 Java 程序员以下列方式声明需要的函数。

> **注意：** 如果使用 com.ms.Win32 软件包，则这些声明也出现在 Kernel32 类中。

```
/** @dll.import("KERNEL32",auto)*/
public native static int LoadLibrary(String lpLibFileName);

/** @dll.import("KERNEL32",auto) */
public native static int LoadLibraryEx(String lpLibFileName,
                                       int hFile, int dwFlags);
/** @dll.import("KERNEL32",auto) */
public native static boolean FreeLibrary(int nLibModule);

/** @dll.import("KERNEL32",ansi) */
public native static int GetProcAddress(int hModule,String lpProcName);
```

注意，GetProcAddress 是使用 ansi 修饰符而不是 auto 来声明的。这是因为 GetProcAddress 是少数几个不带等效的 Unicode 的 Windows 函数之一。如果使用 auto 修饰符，则此函数在 Microsoft Windows 系统上是失败的。

调用函数唯一留下的问题是通过 GetProcAddress 获得的。为了方便起见，msjava.dll（实现 Microsoft VM 的 DLL）导出名为 call 的指定函数，call 函数的第一个参数是指向第二个函数的指针。所有调用所做的就是调用第二个函数，把它传递给其余参数。

下面是关于应用程序如何加载 DLL 和通过 DLL 调出 AFunction 的例子。

```
BOOL Afunction(LPCSTR argument);
```

```
/** @dll.import("msjava")*/
static native boolean call(int funcptr, String argument);
int hmod = LoadLibrary("... ");
int funcaddr = GetProcAddress(hmod, "Afunction");
boolean result = call(funcaddr, "Hello");
FreeLibrary(hmod);
```

## J/Direct与原始本机接口比较

J/Driect 与原始本机接口（RNI）是互补的技术。使用 RNI 要求 DLL 函数遵循严格的命名约定，要求 DLL 函数可以与 Java 无用单元回收程序

协调工作。也就是说，RNI函数必须保证在消耗时间的代码、起始（yield）代码、阻塞其他线程的代码、阻塞用户输入的代码周围调用 GCEnable 和 GCDisable。在 Java 环境中，必须特别设计 RNI 函数。作为回报，RNI 函数对 Java 对象内部和 Java 类加载程序可获得快速访问。

J/Direct 使用现有代码，诸如 Win32 API 函数，使 Java 链接，此函数不是为处理 Java 无用单元回收和 Java 运行时间环境的其他微妙的区别而设计的。然而，为了用户的利益，在无用单元回收程序中，J/Direct 自动调用 GCEnable，以便可以调用阻塞或执行 UI 的函数，而在无用单元回收中没有不好的影响。另外，J/Direct 自动转换普通数据类型（如字符串和结构）为 C 函数通常的格式，于是，就可以不必写很长的代码和包装 DLLs，折衷方案是 DLL 函数不能访问任意 Java 对象字段和方法。在这个版本中，它们只能访问使用@dll.struct 伪指令声明的对象的字段和方法。J/Direct 的另一个限制是不能从使用 J/Direct 调用的 DLL 函数调用 RNI 函数。这项限制的原因是无用单元回收可能与 DLL 函数并行运行。所以，任何通过 RNI 函数返回的或通过 DLL 函数操纵的对象句柄原本就是不稳定的。

幸运的是，可以使用 RNI 或 J/Direct（或两者）。编译器和 Microsoft VM 允许根据需要在相同的类中混合和匹配 J/Direct 和 RNI。

## 安全问题

对独立的 Java 应用程序和信任的内部网 Web 应用程序来说，尽管 J/Direct

是非常有力的特征，但它显然太强大了，以致于不能被 Web 上的普通 Java 小应用程序使用。本节描述 J/Direct 如何与 Microsoft VM 的安全系统一起防止不被信任的代码错误地访问 J/Direct 提供的功能。

## 信任与不信任的类

J/Direct 把所有加载的类分成了两种：

1. 完全信任（指出了最大权限）。
2. 不信任。

只有完全信任的类才允许使用 J/Direct。如果以下有一条语句是正确的，则认为 Java 类是完全信任的。

- 类进行数字签名，指出完全信任，这种类的例子有签名的小应用程序。
- 在目标计算机的 CLASSPATH 上安装的类，或通过软件包管理器安装的类，为在多个小应用程序之间共享而设计的数字签名库必须遵循这项准则。
- 使用 JVIEW 或 WJVIEW 应用程序把类作为非浏览器应用程序来运行。

另一方面，Web 上未签名的小应用程序，是不信任的 Java 代码。

## J/Direct方法调用的安全检查点

Microsoft VM 在三个不同的时间为 J/Direct 方法应用安全检查：

1. 连接时。
2. 第一次调用时。
3. 每次调用时。

只有在传递任意三个安全检查中的一个或显式禁用时，才尝试 J/Direct 调用。

### 链接时的安全检查

当 Java 类调用或访问（使用 Reflection API）另一类的成员时，才进行链接。在链接时，Microsoft VM 查看引用的类是否是可访问的，传递参数的类型和数目是否正确。如果类在相同的软件包中或已声明为公共的，则认为此类是可访问的。
使用标准的 Java 语言，对类的可访问性，限于两种选项：可以声明一个公共的类（每个人都可链接到它），或声明不带有公共修饰符的类（只有在相同软件包中的类才能链接到它）。然而，使用 Microsoft Internet Explorer 4.0，现在有第三种选项。可以声明此类为“只对完全信任的调用程序为公共的”。使用这种可访问性类型可以声明任何类，即使此类并不使用 J/Direct。为了声明此类，需在类的开头放置如下伪指令：

```
/** @security(checkClassLinding=on) */
```

重要的是要注意，这种安全检查只防止不信任的调用程序直接调用“保护的”类。并不防止间接调用。第三（完全信任的）类可以将不信任调用程序的调用转发到“保护的”类。然而，这是一种防护措施。中间类必须在目标计算机的 CLASSPATH 上安装，或必须已经为最大信任度进行数字签名并使用浏览器安装。

## 首次调用的安全检查

方法的第一次调用是指第一次从任一调用程序调用此方法。此时，对于本机关键字所标记的任何一种方法，Microsoft VM 确定此方法是否为完全信任类的成员。如果不是，则发出 SecurityException，并包括消息“Only fully trusted class can have native methods as members（只有完全信任的类才可以有本机方法作为成员）”。因为这项安全检查并不依赖于调用上下文，所以它只须一次执行。如果已传递，则在将来的调用中就不再进行检查。这里没有方法来禁止安全检查。

## 每次调用的安全检查

这是最严格的有效性检查。在每次调用，都检查整个调用栈。即使在调用栈上发现调用程序不是完全信任的，也发出 SecurityException。默认情况下，所有 J/Direct 方法都执行这项检查。RNI 方法不执行这项检查，这是由于向后兼容的需要。RNI 的设计允许从原始 JDK 1.0 本机接口很容易地进行移植，此接口不提供安全检查。

尽管此安全检查提供了最大的安全性，Microsoft 也提供了一种禁用它

的方法。提供此禁止机制是因为严格的安全检查有两点副作用：

| | |
|---|---|
| 可能的性能下降 | 这项安全性检查需要在每次调用 J/Direct 方法时扫描整个调用堆栈。在信任的小应用程序上，性能的下降最为显著，它通常是用现有的安全性管理器来运行的。另一方面，应用程序通常看不到大幅性能下降。这是由于 J/Direct 对没有安全管理器的应用程序忽略了堆栈扫描 |
| inflexibility | 这个安全机制强制使用最大权限，甚至是在一些情况下，只需要特定的权限。例如，要考虑使用 J/Direct 用安全方式公布单一权限到不信任的小应用程序。对于此库来说，应当关闭调用时间安全检查特征，并为具体的权限执行自己的安全检查 |

@security 伪指令禁用每次调用的安全检查，句法如下：

```
/** @security(checkDllCalls=off) */
```

@security 伪指令应用于整个类。在类中不能标记单个方法。下例显示了@security 伪指令的使用：

```
/** @security(checkDllCalls=off) */
    class FastJDirectMethods{
        /** @dll.import(...) */
        static native void func();
    }
```

注意，如果禁用这项安全检查，会将 Microsoft VM 的安全性责任转换到用户那里。要记住，即使禁止了安全性检查，对于最大信任的类，仍然必须数字签名。如果决定使用这条伪指令，请确保采用如下预防措施：

- 所有 J/Direct 方法都已声明为专用。
- 任何公共可访问的方法都不能盲目为 J/Direct 传递调用程序参数。用户必须负责保证只有有效的参数可以传递到本机代码。
- 类公布的能力不应当多于需要的能力，应当通过适当的安全检查防止对这些功能的访问。

**重点：**即使小应用程序是信任的，来自小应用程序的初始化、启动、停止或撤消方法中的调用也可能会触发 SecurityExcePtionEx。为了避免这种情况，应该通过如下代码来确定权限。

```
import com.ms.security.*;
    ...
PolicyEngine.assertPermission(PermissionID.SYSTEM);
```

## J/Direct结构的安全检查点

J/Direct 也在使用@dll.struct 伪指令标记的类上强加安全性限制。因为在例示结构时，此结构是不安全的，所以，这些安全性检查要比 J/Direct 方法使用的检查要有效得多。下面是在@dll.struct 类上执行的两种安全性检查。

| | |
|---|---|
| 加载时间 | 只有在上下文指出完全信任时，使用@dll.struct 伪指令标记的类才加载 |
| 链接时间 | 非完全信任的代码不能链接到使用@dll.struct 伪指令声明的类。如果已经作出了尝试，则 Microsoft VM 将发出一个 NoClassDefFoundError |

## 安全性与com.ms.Win32类

对于最大安全性，J/Direct方法在 com.ms.Win32 软件包中定义，在每次调用时都执行调用的堆栈检查。如果正使用这些 Java 应用程序的类（在 VIEW 或 WJVIEW 下运行），则性能开销是可以忽略的，如果使用来自信任类的 com.ms.Win32 类，并要求最大性能，则应该把需要的 J/Direct 声明拷贝到自己的类中，并禁止每次调用都执行安全检查。有关信息请参看本章前面的“每次调用时的安全检查”。

## 错误信息

当使用 J/Direct时，有几种 Microsoft VM 可能发出的异常类型。下面列出每一项运行期间的错误、导致错误信息的原因以及解决方案。

- Java.lang.SecurityException[class.method]
- Java.lang.IllegalAccessError
- Java.lang.SecurityException

- Java.lang.NoClassDefFoundError
- Com.ms.securityExceptionEx

## java.lang.SecurityException[class.method]

| | |
|---|---|
| 异常类 | java.lang.SecurityException |
| 消息 | class.method:Only fully trusted classed can have native methods as members（只有完全信任的类可以具有本机方法作为成员） |
| 可能的原因 | native 关键字已经在方法上使用，这个方法是不使用完全权限加载的类的成员（例如，未签名的小应用程序）。只有在尝试调用本机方式时，再发出此异常 |
| 可能的解决方案 | 数字签名请求完全权限的小应用程序 |

## Java.lang.IllegalAccessError

| | |
|---|---|
| 异常类 | java.lang.IllegalAccessError |
| 消息 | Class has been marked as nonpublic to untrusted code（类已经对不信任代码标记的非公共的） |
| 可能的原因 | 不信任的类已经尝试引用一个字段或另一个类的方法，这个类已经标记为只能信任使用。在 com.ms.com 和 com.ms.dll 软件包中，许多系统类已经以这种方式进行标记了。对不信任代码，类可以以如下方式使用@security 伪指令标记为非公共的：<br>/** @security(checkClassLinking=on) */ |

```
public class ForTrustedUseOnly{
    …
}
```

| | |
|---|---|
| 解决方法 | 数字签名需要完全权限的小应用程序 |

## java.lang.SecurityException

| | |
|---|---|
| 异常类 | java.lang.SecurityException |
| 消息 | J/Direct method has not been authorized for use on behalf of an untrusted caller（J/Direct 方法未授权代表不信任的调用程序使用） |
| 可能的原因 | 不信任的类已经调用了一个信任的方法试图生成 J/Direct 调用。即使生成实际 J/Direct 调用的类是信任的，如果调用堆栈中的任何方法属于不信任的类，安全管理程序也将发出一个 SecurityException。 |
| 可能的解决方案 | 数字签名请求完全权限的小应用程序。或者，可以禁用安全检查，方法是使用@security 伪指令标记尝试 J/Direct 调用的类，如下例所示 |

```
/** @security(checkDllCalls=off)   */
    public class SafeDllCalls{
        …
    }
```

**注意**：如果禁用这项安全检查，则为 Java 将安全性的责任从 Microsoft VM 传递给用户。要记住，即使禁用了这项安全检查，为了最大信任，还必须对这类进行数字签名。如果使用@security 伪指令，则应当保证下面的情形：

- 所有的 J/Direct 方法都声明为私有的
- 类只公布客户需要的功能
- 类可以使用适当的安全检查保护对这些功能的所有访问

## Java.lang.NoClassDefFoundError

| | |
|---|---|
| 异常类 | java.lang.NoClassDefFoundError |
| 消息 | 无 |
| 可能的原因 | 不信任的类已经尝试加载使用@dll.struct 伪指令标记的类，或者是使用 jactivex 工具生成的类。尽管这实际上违反了安全性（不是类加载程序的错误），为了向后兼容的目的，要发出一个 NoClassDefFoundError |
| 可能的解决方案 | 数字签名请求完全权限的小应用程序 |

## com.ms.security.SecurityExceptionEx

| | |
|---|---|
| 异常类 | com.ms.securityExceptionEx |
| 消息 | [host] J/Direct method has not been authorized for use on behalf of an untrusted caller（J/Direct 方法未授权用于代表不信任的调用程序） |

| | |
|---|---|
| 可能的原因 | 从小应用程序的初始化、启动、停止或撤消方法来尝试 J/Direct 调用 |
| 可能的解决方案 | 执行下面的代码来声明权限<br>import com.ms.security.*;<br>...<br>PolicyEngine.assertPermission(PermissionID.SYSTEM); |

## 故障排除提示

这一节描述了使用 J/Direct 时可能遇到的问题。对每一种情况，都提供了可能解决的方案。

## 调用方法时出现UnsatisifiedLinkError

- 检查编译器的版本，查看它是否是 Microsoft Visual J++的当前版本。如果编译器不支持 J/Direct，则 Microsoft VM 将试图使用 Raw Native Interface 来链接本机方法（并且将不会成功）。
- 确保 DLL 在系统路径上是可视的。将按下列位置（依次）查找 DLLs：
  1. 来自加载的应用程序的目录（通常为 JVIEW）。
  2. 当前目录。
  3. Windows 系统目录。
  4. Windows 目录。

5.PATH 环境变量中列出的目录。

Microsoft VM 将不再试图加载 DLL，直到真正调用需求的方法为止。然而，不能只因为 Java 类加载成功，就认为 DLL 加载成功。

- 检查方法限制词：使用 @dll.import 伪指令声明的方法必须是本机的，而且是静态的。它们可以拥有 Java 语言支持的任何访问级别（公共的，私有的等等）。
- 保证方法名称与 DLL 导出名称确切匹配，包括大小写。Win32 中的 DLL 链接机制是大小写相关的。
- 如果在链接方法时一直有问题，使用实用程序，诸如 dumpbin/export（Visual C++）来通过使用名称验证 DLL 导出了这个方法。有些 DLL 要求通过 ordinal（整数）而不是名称来链接导出的方法。在这种情况中，在方法上使用入口点超越，如下例所示，使用“#ordinal”语法。

```
// This method is exported as ordinal #34.
/** @dll.import("MyDll",entrypoint="#34") */
public static native void MySample();
```

- 要知道，所谓函数实际上是 C 宏，并且实际 DLL 导出名可能完全不同于宏名。

## 调用 DLL 方法或使用 @dll.struct 类时获得 SecurityException

DLL 调用和 @dll.struct 类的使用限于 Java 应用程序和签名的 Java 小应用程序。

## 从DLL函数的返回上截断StringBuffers

在调用DLL函数之前，为了包含所需的字符，必须确保SrtingBuffer的存储容量足够。在SrtingBuffer的构造器中可以指定容量，并且可以在DLL调用之前，使用SrtingBuffer.ensurecapacity方法保证最小容量。

## @dll伪指令内的语法错误

在@dll.import和@dll.struct伪指令内的额外空格将会导致语法错误。在@dll内部要避免使用空格。

## 编译器没有找到com.ms.dll软件包

可以使用旧版本的Classes.zip。在磁盘驱动器上重装Visual J++，试着保留所有Classes.zip的旧版本。

## @dll伪指令不能用于小应用程序（或只可用于Microsoft Visual J++环境）

因为J/Direct使用的安全性是折衷方式，所以，它只限于Java应用程序和签名的Java小应用程序。

## 使用 J/Direct 可产生不信任的类

在类中，通过不信任代码使用 J/Direct 都会导致类成为不安全的，或者不可使用的，实际上，即使 J/Direct 方法也不调用。

## 在调用本机函数之后，J/Direct 抛弃了 ParameterCountMismatchError

ParameterCountMismatchError 异常会警告用户，被调用函数消耗的参数（用堆栈）比 J/Direct 传递的参数或多或少。这个错误通常指出在 Java 方法中声明的参数与 DLL 函数期望的参数是不匹配的。
如果函数没有参数，则假设使用 cdecl 调用约定且不发出异常。即使 Java 方法声明了非零参数。

> **警告：** 不应当试图捕获和处理 ParameterCountMismatchError 异常。异常是为了辅助开发人员在开发阶段捕获错误而设计的。由于性能的原因，只有当应用程序是在 Java 调试器下运行时，才执行参数的计数检查。在函数调用完成后，记下 J/Direct 执行了这次检查也是很重要的。因为异常指出了为函数调用传递了一个或多个无效参数，所以它不能保证此进程可以恢复。

## J/Direct不能卸载 DLL

当 Microsoft VM 放弃了导入 DLL 的 Java 类时，J/Direct 卸载 DLL。对

于运行在 JVIEW 下的 Java 应用程序来说，直到进程退出，也不会发生这种情况。对于信任的 Java 小应用程序，在浏览器离开了包含小应用程序的页之后，会在某些不确定的时间发生，在重新访问页面的情况下，为了优化小应用程序的刷新时间，Microsoft VM 试图为后面的几页保留已加载的 Java 类。

如果需要显式控制 DLL 的加载和卸载，就必须显式调用 Windows 加载程序，并使用调用入口点来自动调用函数。关于如何做这些的详细信息，请参看本章前面的“动态加载和调用 DLLs”。