Overview of the .NET Framework

The .NET Framework is a technology that supports building and running the next generation of apps and XML Web services. The .NET Framework is designed to fulfill the following objectives:

- To provide a consistent object-oriented programming environment whether object code is stored and executed locally, executed locally but Internet-distributed, or executed remotely.
- To provide a code-execution environment that minimizes software deployment and versioning conflicts.
- To provide a code-execution environment that promotes safe execution of code, including code created by an unknown or semi-trusted third party.
- To provide a code-execution environment that eliminates the performance problems of scripted or interpreted environments.
- To make the developer experience consistent across widely varying types of apps, such as Windows-based apps and Web-based apps.
- To build all communication on industry standards to ensure that code based on the .NET Framework integrates with any other code.

The .NET Framework consists of the common language runtime (CLR) and the .NET Framework class library. The common language runtime is the foundation of the .NET Framework. Think of the runtime as an agent that manages code at execution time, providing core services such as memory management, thread management, and remoting, while also enforcing strict type safety and other forms of code accuracy that promote security and robustness. In fact, the concept of code management is a fundamental principle of the runtime. Code that targets the runtime is known as managed code, while code that doesn't target the runtime is known as unmanaged code. The class library is a comprehensive, object-oriented collection of reusable types that you use to develop apps ranging from traditional command-line or graphical user interface (GUI) apps to apps based on the latest innovations provided by ASP.NET, such as Web Forms and XML Web services.

The .NET Framework can be hosted by unmanaged components that load the common language runtime into their processes and initiate the execution of managed code, thereby creating a software environment that exploits both managed and unmanaged features. The .NET Framework not only provides several runtime hosts but also supports the development of third-party runtime hosts.

For example, ASP.NET hosts the runtime to provide a scalable, server-side environment for managed code. ASP.NET works directly with the runtime to enable ASP.NET apps and XML Web services, both of which are discussed later in this topic.

Internet Explorer is an example of an unmanaged app that hosts the runtime (in the form of a MIME type extension). Using Internet Explorer to host the runtime enables you to embed

managed components or Windows Forms controls in HTML documents. Hosting the runtime in this way makes managed mobile code possible, but with significant improvements that only managed code offers, such as semi-trusted execution and isolated file storage.

The following illustration shows the relationship of the common language runtime and the class library to your apps and to the overall system. The illustration also shows how managed code operates within a larger architecture.

Features of the common language runtime

The common language runtime manages memory, thread execution, code execution, code safety verification, compilation, and other system services. These features are intrinsic to the managed code that runs on the common language runtime.

Regarding security, managed components are awarded varying degrees of trust, depending on a number of factors that include their origin (such as the Internet, enterprise network, or local computer). This means that a managed component might or might not be able to perform file-access operations, registry-access operations, or other sensitive functions, even if it's used in the same active app.

The runtime also enforces code robustness by implementing a strict type-and-code-verification infrastructure called the common type system (CTS). The CTS ensures that all managed code is self-describing. The various Microsoft and third-party language compilers generate managed code that conforms to the CTS. This means that managed code can consume other managed types and instances, while strictly enforcing type fidelity and type safety.

In addition, the managed environment of the runtime eliminates many common software issues. For example, the runtime automatically handles object layout and manages references to objects, releasing them when they are no longer being used. This automatic memory management resolves the two most common app errors, memory leaks and invalid memory references.

The runtime also accelerates developer productivity. For example, programmers write apps in their development language of choice yet take full advantage of the runtime, the class library, and components written in other languages by other developers. Any compiler vendor who chooses to target the runtime can do so. Language compilers that target the .NET Framework make the features of the .NET Framework available to existing code written in that language, greatly easing the migration process for existing apps.

While the runtime is designed for the software of the future, it also supports software of today and yesterday. Interoperability between managed and unmanaged code enables developers to continue to use necessary COM components and DLLs.

The runtime is designed to enhance performance. Although the common language runtime provides many standard runtime services, managed code is never interpreted. A feature called just-in-time (JIT) compiling enables all managed code to run in the native machine language of the system on which it's executing. Meanwhile, the memory manager removes the possibilities of fragmented memory and increases memory locality-of-reference to further increase performance.

Finally, the runtime can be hosted by high-performance, server-side apps, such as Microsoft SQL Server and Internet Information Services (IIS). This infrastructure enables you to use managed code to write your business logic, while still enjoying the superior performance of the industry's best enterprise servers that support runtime hosting.

.NET Framework class library

The .NET Framework class library is a collection of reusable types that tightly integrate with the common language runtime. The class library is object oriented, providing types from which your own managed code derives functionality. This not only makes the .NET Framework types easy to use but also reduces the time associated with learning new features of the .NET Framework. In addition, third-party components integrate seamlessly with classes in the .NET Framework.

For example, the .NET Framework collection classes implement a set of interfaces for developing your own collection classes. Your collection classes blend seamlessly with the classes in the .NET Framework.

As you would expect from an object-oriented class library, the .NET Framework types enable you to accomplish a range of common programming tasks, including tasks such as string management, data collection, database connectivity, and file access. In addition to these common tasks, the class library includes types that support a variety of specialized development scenarios. Use the .NET Framework to develop the following types of apps and services:

1. Console apps. See Building Console Applications.
2. Windows GUI apps (Windows Forms). See Windows Forms.
3. Windows Presentation Foundation (WPF) apps. See Windows Presentation Foundation.
4. ASP.NET apps. See Web Applications with ASP.NET.
5. Windows services. See Introduction to Windows Service Applications.
6. Service-oriented apps using Windows Communication Foundation (WCF). See Service-Oriented Applications with WCF.
7. Workflow-enabled apps using Windows Workflow Foundation (WF). See Windows Workflow Foundation.

The Windows Forms classes are a comprehensive set of reusable types that vastly simplify Windows GUI development. If you write an ASP.NET Web Form app, you can use the Web

Forms classes.

WPF overview

Windows Presentation Foundation (WPF) lets you create desktop client applications for Windows with visually stunning user experiences.

The core of WPF is a resolution-independent and vector-based rendering engine that is built to take advantage of modern graphics hardware. WPF extends the core with a comprehensive set of application-development features that include Extensible Application Markup Language (XAML), controls, data binding, layout, 2D and 3D graphics, animation, styles, templates, documents, media, text, and typography. WPF is included in the .NET Framework, so you can build applications that incorporate other elements of the .NET Framework class library.

This overview is intended for newcomers and covers the key capabilities and concepts of WPF.

Program with WPF

WPF exists as a subset of .NET Framework types that are for the most part located in the System.Windows namespace. If you have previously built applications with .NET Framework using managed technologies like ASP.NET and Windows Forms, the fundamental WPF programming experience should be familiar; you instantiate classes, set properties, call methods, and handle events, all using your favorite .NET programming language, such as C# or Visual Basic.

WPF includes additional programming constructs that enhance properties and events: dependency properties and routed events.

Markup and code-behind

WPF lets you develop an application using both markup and code-behind, an experience with which ASP.NET developers should be familiar. You generally use XAML markup to implement the appearance of an application while using managed programming languages (code-behind) to implement its behavior. This separation of appearance and behavior has the following benefits:

- Development and maintenance costs are reduced because appearance-specific markup is not tightly coupled with behavior-specific code.
- Development is more efficient because designers can implement an application's appearance simultaneously with developers who are implementing the application's behavior.
- Globalization and localization for WPF applications is simplified.

Markup

XAML is an XML-based markup language that implements an application's appearance declaratively. You typically use it to create windows, dialog boxes, pages, and user controls, and to fill them with controls, shapes, and graphics.

The following example uses XAML to implement the appearance of a window that contains a single button.

XAML 复制

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    Title="Window with Button"
    Width="250" Height="100">

  <!-- Add button to window -->
  <Button Name="button">Click Me!</Button>

</Window>
```

Specifically, this XAML defines a window and a button by using the Window and Button elements, respectively. Each element is configured with attributes, such as the Window element's Title attribute to specify the window's title-bar text. At run time, WPF converts the elements and attributes that are defined in markup to instances of WPF classes. For example, the Window element is converted to an instance of the Window class whose Title property is the value of the Title attribute.

Code-behind

The main behavior of an application is to implement the functionality that responds to user interactions, including handling events (for example, clicking a menu, tool bar, or button) and calling business logic and data access logic in response. In WPF, this behavior is implemented in code that is associated with markup. This type of code is known as code-behind. The following example shows the updated markup from the previous example and the code-behind.

XAML 复制

```
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.AWindow"
    Title="Window with Button"
    Width="250" Height="100">

  <!-- Add button to window -->
  <Button Name="button" Click="button_Click">Click Me!</Button>

</Window>
```

In this example, the code-behind implements a class that derives from the Window class. The x:Class attribute is used to associate the markup with the code-behind class. InitializeComponent is called from the code-behind class's constructor to merge the UI that is defined in markup with the code-behind class. (InitializeComponent is generated for you

when your application is built, which is why you don't need to implement it manually.) The combination of x:Class and InitializeComponent ensure that your implementation is correctly initialized whenever it is created. The code-behind class also implements an event handler for the button's Click event. When the button is clicked, the event handler shows a message box by calling the System.Windows.MessageBox.Show method.

Controls

The user experiences that are delivered by the application model are constructed controls. In WPF, control is an umbrella term that applies to a category of WPF classes that are hosted in either a window or a page, have a user interface, and implement some behavior.

Input and commands

Controls most often detect and respond to user input. The WPF input system uses both direct and routed events to support text input, focus management, and mouse positioning.

Applications often have complex input requirements. WPF provides a command system that separates user-input actions from the code that responds to those actions.

Layout

When you create a user interface, you arrange your controls by location and size to form a layout. A key requirement of any layout is to adapt to changes in window size and display settings. Rather than forcing you to write the code to adapt a layout in these circumstances, WPF provides a first-class, extensible layout system for you.

The cornerstone of the layout system is relative positioning, which increases the ability to adapt to changing window and display conditions. In addition, the layout system manages the negotiation between controls to determine the layout. The negotiation is a two-step process: first, a control tells its parent what location and size it requires; second, the parent tells the control what space it can have.

The layout system is exposed to child controls through base WPF classes. For common layouts such as grids, stacking, and docking, WPF includes several layout controls:

- Canvas: Child controls provide their own layout.
- DockPanel: Child controls are aligned to the edges of the panel.
- Grid: Child controls are positioned by rows and columns.
- StackPanel: Child controls are stacked either vertically or horizontally.
- VirtualizingStackPanel: Child controls are virtualized and arranged on a single line that is either horizontally or vertically oriented.
- WrapPanel: Child controls are positioned in left-to-right order and wrapped to the next line when there are more controls on the current line than space allows.

Data binding

Most applications are created to provide users with the means to view and edit data. For WPF applications, the work of storing and accessing data is already provided for by technologies such as SQL Server and ADO .NET. After the data is accessed and loaded into an application's managed objects, the hard work for WPF applications begins. Essentially, this involves two things:

1. Copying the data from the managed objects into controls, where the data can be displayed and edited.
2. Ensuring that changes made to data by using controls are copied back to the managed objects.

To simplify application development, WPF provides a data binding engine to automatically perform these steps. The core unit of the data binding engine is the Binding class, whose job is to bind a control (the binding target) to a data object (the binding source). This relationship is illustrated by the following figure:

The WPF data binding engine provides additional support that includes validation, sorting, filtering, and grouping. Furthermore, data binding supports the use of data templates to create custom user interface for bound data when the user interface displayed by the standard WPF controls is not appropriate.

Graphics

WPF introduces an extensive, scalable, and flexible set of graphics features that have the following benefits:

- Resolution-independent and device-independent graphics. The basic unit of measurement in the WPF graphics system is the device-independent pixel, which is 1/96th of an inch, regardless of actual screen resolution, and provides the foundation for resolution-independent and device-independent rendering. Each device-independent pixel automatically scales to match the dots-per-inch (dpi) setting of the system it renders on.
- Improved precision. The WPF coordinate system is measured with double-precision floating-point numbers rather than single-precision. Transformations and opacity values are also expressed as double-precision. WPF also supports a wide color gamut (scRGB) and provides integrated support for managing inputs from different color spaces.
- Advanced graphics and animation support. WPF simplifies graphics programming by managing animation scenes for you; there is no need to worry about scene processing, rendering loops, and bilinear interpolation. Additionally, WPF provides hit-testing support and full alpha-compositing support.

- Hardware acceleration. The WPF graphics system takes advantage of graphics hardware to minimize CPU usage.

WPF Data Binding with LINQ to XML

Overview

This topic introduces the dynamic data binding features in the System.Xml.Linq namespace. These features can be used as a data source for user interface (UI) elements in Windows Presentation Foundation (WPF) apps. This scenario relies upon special dynamic properties of System.Xml.Linq.XAttribute and System.Xml.Linq.XElement.

XAML and LINQ to XML

The Extensible Application Markup Language (XAML) is an XML dialect created by Microsoft to support .NET Framework 3.0 technologies. It is used in WPF to represent user interface elements and related features, such as events and data binding. In Windows Workflow Foundation, XAML is used to represent program structure, such as program control (workflows). XAML enables the declarative aspects of a technology to be separated from the related procedural code that defines the more individualized behavior of a program.

There are two broad ways that XAML and LINQ to XML can interact:

- Because XAML files are well-formed XML, they can be queried and manipulated through XML technologies such as LINQ to XML.
- Because LINQ to XML queries represent a source of data, these queries can be used as a data source for data binding for WPF UI elements.

This documentation describes the second scenario.

Data Binding in the Windows Presentation Foundation

WPF data binding enables a UI element to associate one of its properties with a data source. A simple example of this is a Label whose text presents the value of a public property in a user-defined object. WPF data binding relies on the following components:

A dependency property is a concept specific to WPF that represent a dynamically computed property of a UI element. For example, dependency properties often have default values or values that are provided by a parent element. These special properties are backed by instances of the DependencyProperty class (and not fields as with standard properties). For more information, see Dependency Properties Overview.

Dynamic Data Binding in WPF

By default, data binding occurs only when the target UI element is initialized. This is called one-time binding. For most purposes, this is insufficient; typically, a data-binding solution requires that the changes be dynamically propagated at run time using one of the following:

- One-way binding causes the changes to one side to be propagated automatically. Most commonly, changes to the source are reflected in the target, but the reverse can sometimes be useful.
- In two-way binding, changes to the source are automatically propagated to the target, and changes to the target are automatically propagated to the source.

For one-way or two-way binding to occur, the source must implement a change notification mechanism, for example by implementing the INotifyPropertyChanged interface or by using a PropertyNameChanged pattern for each property supported.

Dynamic Properties in LINQ to XML Classes

Most LINQ to XML classes do not qualify as proper WPF dynamic data sources. Some of the most useful information is available only through methods, not properties, and properties in these classes do not implement change notifications. To support WPF data binding, LINQ to XML exposes a set of dynamic properties.

These dynamic properties are special run-time properties that duplicate the functionality of existing methods and properties in the XAttribute and XElement classes. They were added to these classes solely to enable them to act as dynamic data sources for WPF. To meet this need, all these dynamic properties implement change notifications.

Many of the standard public properties, found in the various classes in the System.Xml.Linq namespace, can be used for one-time data binding. However, remember that neither the source nor the target will be dynamically updated under this scheme.

.NET Framework 概述

.NET Framework 是一种技术，该技术支持生成和运行下一代应用和 XML Web Services。 .NET Framework 旨在实现下列目标:

- 提供一个一致的面向对象的编程环境，而无论对象代码是在本地存储和执行，还是在本地执行但在 Internet 上分布，或者是在远程执行的。
- 提供一个将软件部署和版本控制冲突最小化的代码执行环境。
- 提供一个可提高代码（包括由未知的或不完全受信任的第三方创建的代码）执行安全性的代码执行环境。
- 提供一个可消除脚本环境或解释环境的性能问题的代码执行环境。
- 使开发人员的经验在面对类型大不相同的应用（如基于 Windows 的应用和基于 Web 的应用）时保持一致。
- 按照工业标准生成所有通信，确保基于 .NET Framework 的代码可与任何其他代码集成。

.NET Framework 包括公共语言运行时 (CLR) 和 .NET Framework 类库。 公共语言运行时是 .NET Framework 的基础。 可将运行时看作一个在执行时管理代码的代理，它提供内存管理、线程管理和远程处理等核心服务，并且还强制实施严格的类型安全以及可提高安全性和可靠性的其他形式的代码准确性。 事实上，代码管理的概念是运行时的基本原则。 以运行时为目标的代码称为托管代码，而不以运行时为目标的代码称为非托管代码。 类库是一个综合性的面向对象的可重用类型集合，可使用它来开发多种应用，这些应用程序包括传统的命令行或图形用户界面 (GUI) 应用，还包括基于 ASP.NET 提供的最新创新的应用（如 Web 窗体和 XML Web Services）。

.NET Framework 可由非托管组件承载，这些组件将公共语言运行时加载到它们的进程中并启动托管代码的执行，从而创建一个同时利用托管和非托管功能的软件环境。 .NET Framework 不但提供若干个运行时主机，而且还支持第三方运行时主机的开发。

例如，ASP.NET 承载运行时以为托管代码提供可伸缩的服务器端环境。 ASP.NET 直接使用运行时以启用 ASP.NET 应用和 XML Web Services。

Internet Explorer 是承载运行时（以 MIME 类型扩展的形式）的非托管应用的一个示例。使用 Internet Explorer 承载运行时使您能够在 HTML 文档中嵌入托管组件或 Windows 窗体控件。 以这种方式承载运行时可使托管移动代码成为可能，不过它需要进行只有托管代码才能提供的重大改进（如不完全受信任的执行和独立的文件存储）。

下面的插图显示公共语言运行时和类库与应用之间以及与整个系统之间的关系。 该插图还显示托管代码如何在更大的结构内运行。

公共语言运行时的功能

公共语言运行时管理内存、线程执行、代码执行、代码安全验证、编译以及其他系统服务。这些功能是在公共语言运行时上运行的托管代码所固有的。

至于安全性，取决于包括托管组件的来源（如 Internet、企业网络或本地计算机）在内的一些因素，托管组件被赋予不同程度的信任。 这意味着即使用在同一活动应用中，托管组件既可能能够执行文件访问操作、注册表访问操作或其他须小心使用的功能，也可能不能够执行这些功能。

运行时还通过实现称为常规类型系统 (CTS) 的严格类型验证和代码验证基础结构来加强代码可靠性。 CTS 确保所有托管代码都是可以自我描述的。 各种 Microsoft 编译器和第三方语言编译器都可生成符合 CTS 的托管代码。 这意味着托管代码可在严格实施类型保真和类型安全的同时使用其他托管类型和实例。

此外，运行时的托管环境还消除了许多常见的软件问题。 例如，运行时自动处理对象布局并管理对对象的引用，在不再使用它们时将它们释放。 这种自动内存管理解决了两个最常见的应用错误：内存泄漏和无效内存引用。
运行时还提高了开发人员的工作效率。 例如，程序员用他们选择的开发语言编写应用，却仍能充分利用其他开发人员用其他语言编写的运行时、类库和组件。 任何选择以运行时为目标的编译器供应商都可以这样做。 以 .NET Framework 为目标的语言编译器使得用该语言编写的现有代码可以使用 .NET Framework 的功能，这大大减轻了现有应用的迁移过程的工作负担。

尽管运行时是为未来的软件设计的，但是它也支持现在和以前的软件。 托管和非托管代码之间的互操作性使开发人员能够继续使用所需的 COM 组件和 DLL。

运行时旨在增强性能。 尽管公共语言运行时提供许多标准运行时服务，但是它从不解释托管代码。 一种称为实时 (JIT) 编译的功能使所有托管代码能够以它在其上执行的系统的本机语言运行。 同时，内存管理器排除了出现零碎内存的可能性，并增大了内存引用区域以进一步提高性能。

最后，运行时可由高性能的服务器端应用（如 Microsoft SQL Server 和 Internet Information Services (IIS)）承载。 此基础结构使您在享受支持运行时承载的行业最佳企业服务器的优越性能的同时，能够使用托管代码编写业务逻辑。

.NET Framework Class Library — .NET Framework 类库

.NET Framework 类库是一个与公共语言运行时紧密集成的可重用的类型集合。该类库是面向对象的，并提供某些类型，可供你自己的托管代码从中派生功能。 这不但使 .NET Framework 类型易于使用，而且还减少了学习 .NET Framework 的新功能所需要的时间。此外，第三方组件与 .NET Framework 中的类无缝集成。

.NET Framework 类库是一个与公共语言运行时紧密集成的可重用的类型集合。该类库是面向对象的，并提供某些类型，可供你自己的托管代码从中派生功能。 这不但使 .NET Framework 类型易于使用，而且还减少了学习 .NET Framework 的新功能所需要的时间。此外，第三方组件与 .NET Framework 中的类无缝集成。

例如，.NET Framework 集合类实现一组用于开发自己的集合类的接口。你的集合类与 .NET Framework 中的类无缝地混合。

正如您对面向对象的类库所希望的那样，.NET Framework 类型使您能够完成一系列常见编程任务（包括诸如字符串管理、数据收集、数据库连接以及文件访问等任务）。 除这些常规任务之外，类库还包括支持多种专用开发方案的类型。 使用 .NET Framework 开发下列类型的应用和服务：

- Console 应用。 请参阅生成控制台应用程序。
- Windows GUI 应用（Windows 窗体）。 请参阅 Windows 窗体。
- Windows Presentation Foundation (WPF) 应用。 请参阅 Windows Presentation Foundation。
- ASP.NET 应用。 请参阅使用 ASP.NET 的 Web 应用程序。
- Windows 服务。 请参阅 Windows 服务应用程序简介。
- 使用 Windows Communication Foundation (WCF) 的面向服务的应用。 请参阅使用 WCF 的面向服务的应用程序。
- 使用 Windows Workflow Foundation (WF) 的启用工作流程的应用。 请参阅 Windows Workflow Foundation。

Windows 窗体类是一组综合性的可重用的类型，它们大大简化了 Windows GUI 的开发。 如果要编写 ASP.NET Web 窗体应用，可使用 Web 窗体类。

Wpf 概述

使用 Windows Presentation Foundation (WPF)，你可以创建适用于 Windows 且具有非凡视觉效果的桌面客户端应用程序。

WPF 的核心是一个与分辨率无关且基于矢量的呈现引擎，旨在充分利用现代图形硬件。 WPF 通过一套完善的应用程序开发功能对该核心进行了扩展，这些功能包括可扩展应用程序标记语言 (XAML)、控件、数据绑定、布局、二维和三维图形、动画、样式、模板、文档、媒体、文本和版式。 WPF 包含在 .NET Framework 中，因此你可以生成整合其他 .NET Framework 类库元素的应用程序。
本概述适用于新用户，介绍了 WPF 的主要功能和概念。

使用 WPF 进行编程

WPF 作为大部分位于 System.Windows 命名空间中的 .NET Framework 类型的一个子集存在。 如果你之前使用托管技术（如 ASP.NET 和 Windows 窗体）通过 .NET Framework 生成过应用程序，则不会对基本的 WPF 编程体验感到陌生；你可以使用最喜欢的 .NET 编程语言（如 C# 或 Visual Basic）来完成实例化类、设置属性、调用方法以及处理事件等所有操作。

WPF 还包括增强属性和事件的其他编程构造： 依赖项属性 和 路由事件。

标记和代码后置

通过 WPF，可以使用标记和代码隐藏开发应用程序，这是 ASP.NET 开发人员已经熟悉的体验。 通常使用 XAML 标记实现应用程序的外观，同时使用托管编程语言（代码隐藏）来实现其行为。 这种外观和行为的分离具有以下优点：

- 降低了开发和维护成本，因为特定于外观的标记与特定于行为的代码不紧密耦合。
- 开发效率更高，因为设计人员在实现应用程序外观的同时，开发人员可以实现应用程序的行为。
- WPF 应用程序的全球化和本地化 得以简化。

标记

XAML 是一种基于 XML 的标记语言，以声明形式实现应用程序的外观。 通常用它创建窗口、对话框、页和用户控件，并填充控件、形状和图形。
下面的示例使用 XAML 来实现包含一个按钮的窗口的外观。

XAML 复制

```xaml
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    Title="Window with Button"
    Width="250" Height="100">

  <!-- Add button to window -->
  <Button Name="button">Click Me!</Button>

</Window>
```

具体而言，此 XAML 通过分别使用 Window 和 Button 元素来定义窗口和按钮。 每个元素均配置了特性（如 Window 元素的 Title 特性）来指定窗口的标题栏文本。 在运行时，WPF 会将标记中定义的元素和特性转换为 WPF 类的实例。 例如， Window 元素被转换为 Window 类的实例，该类的 Title 属性是 Title 特性的值。

代码后置

应用程序的主要行为是实现响应用户交互的功能，包括处理事件（例如，单击菜单、工具栏或按钮）以及相应地调用业务逻辑和数据访问逻辑。 在 WPF 中，在与标记相关联的代码中实现此行为。 此类代码称为代码隐藏。 下面的示例演示上一个示例的更新标记和代码隐藏。

XAML 复制

```xaml
<Window
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    x:Class="SDKSample.AWindow"
    Title="Window with Button"
    Width="250" Height="100">

  <!-- Add button to window -->
  <Button Name="button" Click="button_Click">Click Me!</Button>

</Window>
```

在此示例中，代码隐藏实现派生自 Window 类的类。 x:Class 特性用于将标记与代码隐藏类相关联。 从代码隐藏类的构造函数调用 InitializeComponent，以将标记中定义的 UI 与代码隐藏类合并在一起。 （生成应用程序时将为你生成 InitializeComponent，因此你无需手动实现它。）x:Class 和 InitializeComponent 的组合可确保你的实现无论在何时创建都会得到正确初始化。代码隐藏类还可实现按钮的 Click 事件的事件处理程序。单击该按钮后，事件处理程序会通过调用 System.Windows.MessageBox.Show 方法显示一个消息框。

控件

应用程序模型带来的用户体验是构造的控件。 在 WPF 中，控件是适用于 WPF 类这一类别的总括术语，这些类托管在窗口或页中、具有用户界面并实现一些行为。

输入和命令

最常检测和响应用户输入的控件。 WPF 输入系统 使用直接事件和路由事件来支持文本输入、焦点管理和鼠标定位。

应用程序通常具有复杂的输入要求。 WPF 提供了命令系统，用于将用户输入操作与对这些操作做出响应的代码分隔开来。

布局

创建用户界面时，按照位置和大小排列控件以形成布局。 任何布局的一项关键要求都是适应窗口大小和显示设置的变化。 WPF 为你提供一流的可扩展布局系统，而不强制你编写代码以适应这些情况下的布局。

布局系统的基础是相对定位，这提高了适应不断变化的窗口和显示条件的能力。 此外，该布局系统还可管理控件之间的协商以确定布局。 协商是一个两步过程：首先，控件将需要的位置和大小告知父级；其次，父级将控件可以有的空间告知控件。

该布局系统通过基 WPF 类公开给子控件。 对于通用的布局（如网格、堆叠和停靠），WPF 包括若干布局控件：

- Canvas：子控件提供其自己的布局。
- DockPanel：子控件与面板的边缘对齐。
- Grid：子控件按行和列放置。
- StackPanel：子控件垂直或水平堆叠。
- VirtualizingStackPanel：子控件在水平或垂直的行上进行虚拟化和排列。
- WrapPanel：子控件按从左到右的顺序放置，在当前行上的控件超出允许的空间时，换行到下一行。

数据绑定

大多数应用程序旨在为用户提供查看和编辑数据的方法。 对于 WPF 应用程序，已对存储和访问数据的工作提供技术（如 SQL Server 和 ADO.NET）。 访问数据并将数据加载到应用程序的托管对象后，WPF 应用程序的复杂工作开始。 从根本上来说，这涉及到两件事：

1. 将数据从托管对象复制到控件，在控件中可以显示和编辑数据。
2. 确保使用控件对数据所做的更改将复制回托管对象。

为了简化应用程序开发，WPF 提供了一个数据绑定引擎来自动执行这些步骤。 数据绑定引擎的核心单元是 Binding 类，其工作是将控件（绑定目标）绑定到数据对象（绑定源）。 下图阐释了这种关系：

WPF 数据绑定引擎提供了额外支持，包括验证、排序、筛选和分组。 此外，数据绑定支持在标准 WPF 控件显示的用户界面不恰当时，使用数据模板来为数据绑定创建自定义的用户界面。

图形

WPF 引入了一组广泛、可伸缩的灵活图形功能，具有以下优点：

- 图形与分辨率和设备均无关。 WPF 图形系统中的基本度量单位是与设备无关的像素（即 1/96 英寸），且不考虑实际屏幕分辨率，并为实现与分辨率和设备无关的呈现提供了基础。 每个与设备无关的像素都会自动缩放，以匹配呈现它的系统的每英寸点数 (dpi) 设置。
- 精度更高。 WPF 坐标系统使用双精度浮点数字度量，而不是单精度数字。 转换和不透明度值也表示为双精度数字。 WPF 还支持广泛的颜色域 (scRGB)，并集成了对管理来自不同颜色空间的输入的支持。
- 高级图形和动画支持。 WPF 通过为你管理动画场景简化了图形编程，你无需担心场景处理、呈现循环和双线性内插。 此外，WPF 还提供了点击测试支持和全面的 alpha 合成支持。

- 硬件加速。 WPF 图形系统充分利用图形硬件来尽量降低 CPU 使用率。

使用 LINQ to XML 进行 WPF 数据绑定概述

本主题介绍 System.Xml.Linq 命名空间中的动态数据绑定功能。这些功能可用作 Windows Presentation Foundation (WPF) 应用中用户界面 (UI) 元素的数据源。 此方案依赖于 System.Xml.Linq.XAttribute 和 System.Xml.Linq.XElement 的特殊动态属性。
XAML 和 LINQ to XML

可扩展应用程序标记语言 (XAML) 是由 Microsoft 创建的 XML 方言，支持 .NET Framework 3.0 技术。 它在 WPF 中用于表示用户界面元素和相关功能，如事件和数据绑定。 在 Windows Workflow Foundation 中，XAML 用于表示程序结构，如程序控制（工作流）。 XAML 使技术的声明性方面与相关的过程性代码分离，从而可定义更具个性化的程序行为。
XAML 和 LINQ to XML 的交互有两种主要方式：

- 由于 XAML 文件是格式良好的 XML，因此可以通过 XML 技术（如 LINQ to XML）查询和操作。
- 由于 LINQ to XML 查询表示数据的源，因此这些查询可用作 WPF UI 元素数据绑定的数据源。

Windows Presentation Foundation 中的数据绑定

WPF 数据绑定可使 UI 元素将其一个属性与一个数据源相关联。 这种情况的一个简单示例是 Label，其文本表示用户定义对象中一个公共属性的值。 WPF 数据绑定依赖于下列组件：

依赖项属性是特定于 WPF 的概念，它表示 UI 元素的动态计算的属性。 例如，依赖项属性通常具有默认值或具有由父元素提供的值。 DependencyProperty 类的实例（而不是支持标准属性的字段）支持这些特殊属性。 有关详细信息，请参阅依赖项属性概述。

WPF 中的动态数据绑定

默认情况下，仅在初始化目标 UI 元素时，才会发生数据绑定。 这称为“一次性”绑定。 这不能满足多数用途的需要；通常，数据绑定解决方案要求使用以下方式之一在运行时动态传播更改：

- 单向绑定，这种方式会自动传播对一侧所做的更改。 最常见的情况是对源所做的更改会反映在目标中，但有时需要相反的情况。
- 双向绑定，在这种方式中，对源所做的更改会自动传播到目标，而且对目标的更改也会自动传播到源。

LINQ to XML 类中的动态属性

大多数 LINQ to XML 类都不适合作为适当的 WPF 动态数据源。 一些最有用的信息只能通过方法（而不是属性）提供，并且这些类中的属性不实现更改通知。 为了支持 WPF 数据绑定，LINQ to XML 公开了一组动态属性。

这些动态属性是特殊的运行时属性，它们重复 XAttribute 和 XElement 类中现有方法和属性的功能。 将这些属性添加到这些类中只是为了使这些类能够充当 WPF 的动态数据源。 为了满足这一要求，所有这些动态属性都要实现更改通知。

System.Xml.Linq 命名空间的各个类中的很多标准公共属性都可用于一次性数据绑定。但请记住，在这种方案下，源和目标都不会动态更新。