导航属性通常被定义为virtual，使他们能获得某些实体框架的功能，比如延迟加载的优势。

如果某个导航属性可以包含多个实体(如多对多或一对多关系)，它的类型必须可以进行增删改操作，比如ICollection。

如果一个属性的命名方式为导航属性名+主键属性名，实体框架便会将该属性视为外键属性。(例如，Student实体的主键为ID，则StudentID被 视为为Student导航属性的外键)。外键的属性也可以命名为简单的主键属性名(例如，Course实体的主键为CourseID)。

DatabaseGenerated特性。该特性可以让你来输入该实体的主键值，而不是让数据库自动生成它。

在一个数据模型中负责协调实体框架功能的主类被称为数据库上下文类。您可以通过派生自System.Data.Entity.DbContext类来创建。你可以在代码中指定那些实体被包含在数据模型中。您可以可以自定义某些实体框架的行为。

为每个实体集合创建了一个DbSet属性。在实体框架中，一个实体集对应数据库中的表，一个实体对应数据表中的一行。

连接字符串(稍后将被添加到web.config文件中)的名称被传递给构造函数。

1 public SchoolContext() : base("SchoolContext")

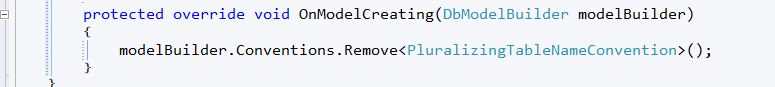
2 {

3 }

你同样可以通过传递连接字符串而不是存储在web.config文件的连接字符串名称本身来指定连接。如果你不指定连接字符串或一个明确的名称，实体框架 将假定连接字符串名称和类名称一致，即在本例中，默认的连接字符串名称为SchoolContext，同你显示声明的一致。

**指定表名**

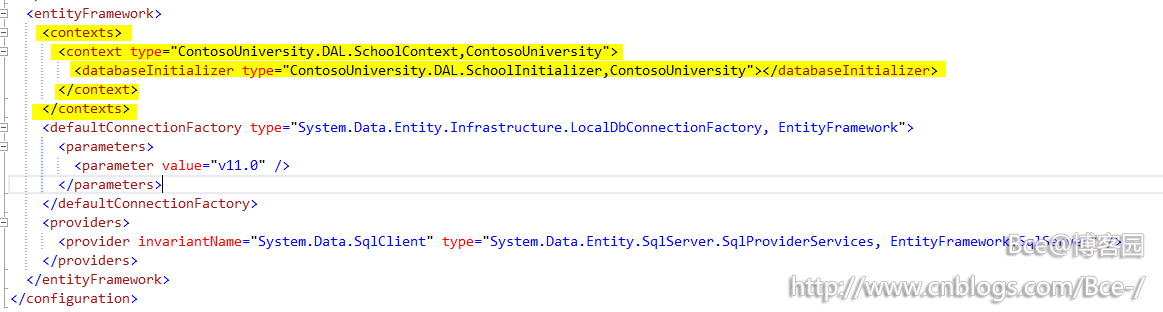
OnModelCreating方法中的modelBuilder.Convertions.Remove被用来防止生成复数表名。如果你不这样 做，在数据库中生成的数据表将被命名为Students,Courses及Entrollments。相反，在本例中我们的表名是 Student,Course及Enrollment。对于表名称是否应该使用复数或单数命名模式并没有明确的要求。在本教程中我们将使用单数形式。重要 的一点是，你可以选择任意的命名方式——通过是否注释掉该行代码。



当你运行程序时，实体框架可以自动创建(或自动删除并重新创建)数据表。你可以指定这应该在每次程序运行时进行或仅当模型发生了变化而不与现有的数据库同步时才进行。你也可以写一个Seed方法，以便在数据库初始化后自动填充测试数据到新的数据表中。

默认的行为是只有当该数据库不存在时才创建(当数据库已经存在时会抛出一个异常)。在本节中你将指定在每次模型发生变化时都删除旧数据库并建立一个 新的。在本例中这样做是适当的。Seed方法将在重新创建后自动填充测试数据。而在生产中通常不希望这样做从而丢失数据库中的所有数据。稍后您将看到如何 使用Code First Migration来改变数据库架构，而不是删除并重新创建。

Seed方法将数据库的上下文对象作为输入参数，并在方法中的代码使用该上下文对象将新的实体添加到数据库中。对于每个实体模型，代码创建新的实体集合添 加他们到相应的DbSet属性，然后将更改保存到数据库。它并不是必须在每组实体后立即调用SaveChanges方法。但这样做有助于你在发生数据库写 入异常时快速找到问题的根源。



context 的Type属性指定了上下文的类名。你应当使用完整的类名和程序集名。同样databaseInitializer的Type指定了初始化类的名称。(如果你不想使用EF初始化，你可以在上下文元素中设置disableDatabaseInitialization="true")。

作为一种在web.config中设置初始值设定项的替代方法，你可以通过在Global.asax.cs中Application\_Start方法中增加Database.SetInitializer语句来实现同样的功能。

现在应用程序已经设置为在程序首次运行时，对模型和数据库中的表进行比较，如果有区别，应用程序删除并重新创建该数据库。

**注意:**当你将应用程序部署到生产环境中时，你必须删除或禁用数据库重新创建代码。

LocalDB是SQL Server Express的一个轻量版本，非常适合用来进行本地测试，但不建议在生产中使用。

ValidateAntiForgeryToken属性有助于防止跨站请求伪造攻击，它需要在视图中相应地设置Html.AntiForgeryToken()语句，您将在后面看到。  
  
Bind 特性用于防止“过多发布”攻击。举例来说，假设Student实体中包含一个Secert字段，你不想让此属性由Web页面来进行更新，所以你没有在页面 上放置Secert的相应输入框。但黑客可以通过工具强行附加Secert字段即相应值到表单中并发送给服务器端。在没有使用Bind的默认情况下，模型 绑定器会自动遍历提交过来的所有表单值并尝试更新到实体中，所以Secert也会得到更新——使用黑客强行附加的值。  
  
安全的做法是使用 Bind特性的Include参数，可以让你指定那些字段是由模型绑定器来进行更新的，也可以相反地使用Exclude来排除你不想让模型绑定器来进行更 新的属性。我们推荐使用Include的理由是，如果对实体添加了新的属性，Exclude是不会自动更新的，新属性会默认被模型绑定器进行更新。  
  
另一种替代方法是使用ViewModel。ViewModel中仅包含你想要绑定的属性。在模型绑定器完成对ViewModel的更新后，将ViewModel中的属性复制到实体的实例已完成更新。  
  
try- catch块是除了Bind特性外您对脚手架代码所做的唯一更改。如果在保存时有一个源于DataException的异常被引发，一个通用的错误消息被 显示出来。由于DataException错误有时会由外部的应用程序引发，而不是程序编写的错误，所以建议用户进行再次尝试。此外，虽然该实例中没有实 现，在生产环境下，所有的应用程序错误都应该被记录下来。

实体状态和附加和调用SaveChanges方法

数据库上下文会跟踪内存中的实体是否与数据库中的行保持同步。并根据同步的信息来确定调用SaveChanges方法时会发生什么。例如，让你传递 一个新实体给Add方法，该实体的状态设置为Added。然后您调用SaveChanges方法时，数据库上下文会生成一个SQL Insert命令以插入数据。

一个实体可能处于以下状态之一：

* Added。该实体尚未在数据库中。SaveChanges方法将发出一个Insert语句。
* Unchanged。SaveChanges对该实体什么都不需要做。当你从数据库读出一个实体时，该实体就为这一状态。
* Modified。某些或所有实体的属性值已都被更改。SaveChanges将发出一个Update语句。
* Deleted。该实体已经被标志为删除。SaveChanges将发出一个Delete语句。
* Detached。该实体没有被跟踪的数据库上下文。

在桌面应用程序中，状态变化通常是自动设置的。在桌面型的应用程序中，你看到一个实体并更改它的一些属性值，将导致它的实体状态自动更改为Modified。然后你调用SaveChanges，实体框架生成一个SQL Update来更新你进行了变更的属性。  
  
Web 应用程序的断开连接性质不允许这种连续序列。数据库上下文在读取到实体并将其呈现在页面上，之后便被销毁。当HttpPost Edit动作方法被调用时，一个新请求被处理，你将获取一个新的数据库上下文的实例。所以你必须手动设置实体状态为Modified，然后你调用 SaveChanges，实体框架更新数据库中的所有的数据行，因为上下文没有办法知道那个属性是你进行了变更的。

如果你想在SQL Update语句只更新用户实际更改的字段，你可以以某种方式保存原来的值(比如隐藏字段)，这样在调用HttpPost Edit方法时就可以使用它们。然后，你可以使用原值来创建一个Student实体，调用原始版本的Attach方法更新实体的值到新值，然后调用 SaveChanges。更多信息请参见MSDN上的[Entity states and SaveChanges](http://msdn.microsoft.com/en-us/data/jj592676) 和 [Local Data](http://msdn.microsoft.com/en-us/data/jj592872)。

在一个Get请求响应中执行删除动作(或者创建、修改等对数据进行变更的动作)将带来安全风险。有关风险的详细信息请参见[ASP.NET MVC Tip #46 — Don't use Delete Links because they create Security Holes](http://stephenwalther.com/blog/archive/2009/01/21/asp.net-mvc-tip-46-ndash-donrsquot-use-delete-links-because.aspx)。

连接恢复功能只能针对特定的数据库服务进行正确的配置后才可用：

* 必须知道那些异常有可能是暂时的，您想要重试由于网络连接而造成的错误，而不是编程Bug带来的。
* 在失败操作的间隔中必须等待适当的时间。批量重试时在线用户可能会需要等待较长时间才能够获得响应。
* 需要设置一个适当的重试次数。在一个在线的应用程序中，您可能会进行多次重试。

DataType特性用于执行比数据库内部类型更加具体的数据类型。在本示例中，我们只想保持对日期的跟踪，而不是日期及时间。DataType枚 举提供了多种数据类型，比如日期，时间，电话号码，电子邮件等。DataType特性同样可以让应用程序来自动基于数据类型的特殊功能。例如 DataType.EmailAddress可以创建mailto:的超链接，DataType.Date特性可以在支持HTML5的浏览器中创建一个日 期选择器。DataType特性可以生成HTML5浏览器支持的HTML5 数据特性。要注意DataType特性并不提供任何验证。

DataType.Date不指定日期的显示格式。默认情况下， 数据字段的显示基于服务器本身的[CultureInfo](http://msdn.microsoft.com/en-us/library/vstudio/system.globalization.cultureinfo%28v=vs.110%29.aspx)的默认格式。

ApplyFormatInEditMode设置指定该值在文本框中进行编辑时也同样适用已指定的格式(某些情况下可能并不适用，比如针对一个货币值，您可能不希望在文本框中显示一个货币符号并对其编辑)。

您可以单独使用DisplayFormat特性，但通常一个好注意是同时使用DataType这两者。DataType特性所传达的是数据本身的表述而不是如何将它呈现在屏幕上。下面列出了一些您可以考虑不使用DisplayFormat的情况：

* 目标浏览器可以启用HTML5功能(比如显示日历控件，区域化的货币符号，电邮链接，一些客户端输入验证等。)。
* 默认情况下，浏览器将使用基于您的区域设置的正确格式老呈现数据。
* DataType特性可以让MVC自动选择正确的模板来呈现数据(DisplayFormat使用字符串模板)。

如果您在日期字段上使用DataType特性，您也应当指定DisplayFormat特性以确保该字段在Chrome浏览器中正确呈现。详细信息请参见[StackOverflow thread](http://stackoverflow.com/questions/12633471/mvc4-datatype-date-editorfor-wont-display-date-value-in-chrome-fine-in-ie)。

有关如何在MVC中处理其他数据类型，请参阅中[MVC 5 Introduction: Examining the Edit Methods and Edit View](http://www.asp.net/mvc/tutorials/mvc-5/introduction/examining-the-edit-methods-and-edit-view)的国际化部分。

**StringLength特性**

您还可以使用特新来指定数据验证规则和验证错误信息。StringLength特性设定设定数据库的最大长度并且提供ASP.NET MVC的客户端及服务器端验证。您还可以在此特性中指定字符串的最小长度，但最小值对数据库的架构没有任何影响。

假设您想要确保用户不能输入超过50个字符的名称，如果要添加该限制，将StringLength特性添加到LastName和FirstMidName属性

StringLength特性不会阻止用户在姓名中输入空白字符，但您可以使用正则表达式属性来进行该限制。例如下面的代码要求第一个字符必须是大写，其余的字符是字母。

[RegularExpression(@"^[A-Z]+[a-zA-Z''-'\s]\*$")]

实体框架会检测到数据模型已经进行了更改并且要求数据库架构也作出相应的改变。您将通过使用迁移来在不丢失数据的情况下升级架构。如果您更改了使用 Seed方法创建的数据，您在Seed方法中所使用的AddOrUpdate方法会更改回其原始状态(AddOrUpdate是一个相当 于"upsert"操作的数据库术语)。

在程序包管理器控制台中，输入以下命令：

add-migration MaxLengthOnNames

update-database

add-migration命令创建一个名为<时间戳>\_MaxLengthOnName.cs的文件，此文件包含用来更新数据库的Up方法，以匹配当前数据模型中的代码。update-database命令运行该代码。

实体框架使用有时间戳前缀的迁移文件名来进行迁移。您可以在运行update-database命令之前创建多个迁移，所有的迁移会按照它们创建的顺序来应用。

如果一个导航属性可以容纳多个实体，则它的类型必须实现ICollection<T>接口，例如List<T>，但不能是IEnumerable<T>，因为它不实现Add。

**Key特性**

Instructor和OfficeAssignment实体之间有一个对零或一对一的关系。办公室只和讲师之间存在关系，因此其主键也是其 Instructor实体的外键。但是实体框架不会自动将InstructorID识别为实体的主键，因为该命名不遵循实体框架约定。因此，我们使用 Key特性来标记该属性为实体的主键：

[Key]

[ForeignKey("Instructor")]

public int InstructorID { get; set; }

如果实体没有它自己的主键，但您想将属性名命名为类名-ID或ID以外的不同的名称，您同样可以使用Key特性。默认情况下实体框架将键视为非数据库生成的，因为该列用来标识关系。

**ForeignKey特性**

当两个实体之间存在有一对零或一对一关系时，实体框架无法自动辨认出那一端的关系是主体，那一端是依赖。一对一关系在每个类中使用导航属性来引用其 他类。ForeignKey特性可以应用于要建立关系的依赖类。如果您省略ForeignKey特性，当您尝试创建迁移时系统会出现一个无法确定实体间关 系的错误。

**添加迁移和更新数据库**

从程序包管理器控制台中，输入add-migration命令(先不要运行update-database命令)：

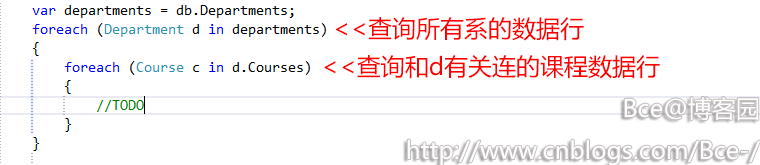
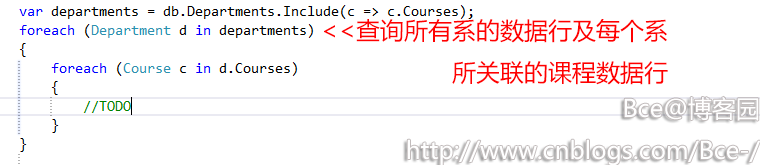
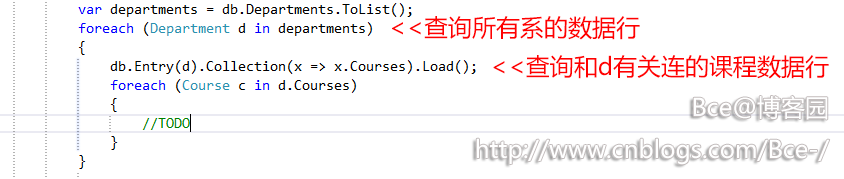
add-Migration ComplexDataModel

如果您试图再次运行update-database命令，您会收到一个外键冲突错误。

当您在保存现有数据的状态下执行迁移时，您需要将存根数据插入到数据库以满足外键约束要求，所以我们现在就来做这些。在 ComplexDataModel中的up方法生成的代码将为Course数据表添加一个非空DepartmentID外键。因为针对Course数据表 中的已有行执行代码时AddColumn操作将失败，因为SQL Server不知道使用何值来填充不可为空的列。因此，必须更改代码，提供一个默认值给新列，并创建一个"Temp"作为默认系的存根。因此，在Up方法 中使用Temp系来分配给Course的现有行。您可以在Seed方法中重新分配给它们正确的系

**延迟、预先和现实加载相关数据**

实体框架拥有多种将相关数据从一个实体的导航属性中进行加载的方法：

* 延迟加载(Lazy Loading)。当实体第一次被读取时，相关数据不会被获取。但是，当你第一次尝试存取导航属性时，该导航属性所需的数据会自动加载。结果会使用多个查 询发送到数据库——一次是读取实体本身，然后是每个相关的实体。DbContext类默认是使用延迟加载的。  
  
* 预先加载(Eager Loading)。当实体读取时立即获取与该实体相关的数据。这通常会导致在单个连接查询中检索出所有所需要的数据。您可以通过使用Include方法来指定预先加载。  
  
* 显式加载(Explicit Loading)。有点类似于延迟加载，只是你在代码中显式地获取相关数据。当您访问一个导航属性时，它不会自动加载。你需要通过使用实体的对象状态管理 器并调用集合上的Collection.Load方法或通过持有单个实体的属性的Reference.Load方法来手动加载相关数据。(在下面的示例 中，如果你想要加载管理员导航属性，你需要使用Reference(x => x.Administrator)来替换Collection(x => x.Courses))  
  

因为延迟加载和显式加载都不立即检索属性的值，所以它们也被称为推迟加载。

**性能注意事项**

如果你知道你立即需要每个实体的相关数据，预先加载通常提供最佳的性能。因为单个查询发送到数据库并一次性获取数据的效率通常比在每个实体上再发出 一次查询的效率更高。例如，在上面的示例中，假定每个系有十个相关的课程，预先加载会导致只有一个查询(join联合查询)往返于数据库。延迟加载和显式 加载两者都将造成11个查询和往返。在高延迟的情况下，额外的查询和往返通常是不利的。

另一方面，在某些情况下使用延迟加载的效率更高。预先加载可能会导致生成SQL Server不能有效处理的非常复杂的联接查询。或者，如果您正在 处理的是需要访问的某个实体的导航属性，该属性仅为实体集的一个子集，延迟加载可能比预先加载性能更好，因为预先加载会将所有的数据全部加载，即使你不需 要访问它们。如果应用程序的性能是极为重要的，你最好测试并在这两种方法之间选择一种最佳的。

延迟加载可能会屏蔽一些导致性能问题的代码。例如，代码没有指定预先或显式加载但在处理大量实体并时在每次迭代中都使用了导航属性的情况下，代码的 效率可能会很低(因为会有大量的数据库往返查询)。一个在开发环境下表现良好的应用程序可能会在移动到Windows Azure SQL数据库时由于增加了延迟导致延迟加载的性能下降。你应当分析并测试以确保延迟加载是否是适当的。详细信息，请参阅[Demystifying Entity Framework Strategies: Loading Related Data](http://msdn.microsoft.com/en-us/magazine/hh205756.aspx)和[Using the Entity Framework to Reduce Network Latency to SQL Azur](http://msdn.microsoft.com/en-us/magazine/gg309181.aspx)

**在序列化之前禁用延迟加载**

如果你在序列化期间启用了延迟加载，最终你可能会查询到比预期更多的数据。序列化一般会访问类的每个属性。而属性访问触发延迟加载，然后会将延迟加 载的实体也进行序列化。最终有可能会导致更多的延迟加载及属性的序列化，要防止这种链式反应，请在实体序列化之前禁用延迟加载。

有一种避免序列化问题的方式是序列化数据传输对象(DTO)而不是实体对象，如[Using Web API with Entity Framework](http://www.asp.net/web-api/overview/creating-web-apis/using-web-api-with-entity-framework/using-web-api-with-entity-framework,-part-6)教程所示。

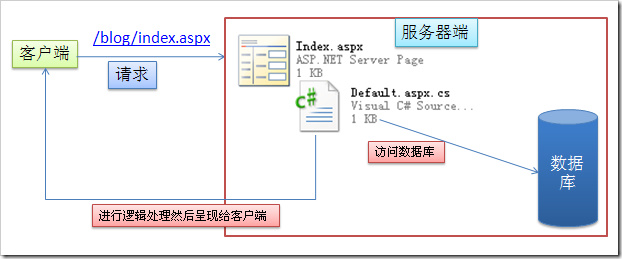
如果您不想使用DTO，您可以禁用延迟加载并避免通过[disabling proxy creation](http://msdn.microsoft.com/en-US/data/jj592886.aspx)来避免代理问题。

[这里](http://msdn.microsoft.com/en-US/data/jj574232)有一些禁用延迟加载的方式：

* 对于特定的导航属性，省略virtual关键字声明。
* 对于所有的导航属性，可以设置LazyLoadingEnabled为false，将下面的代码放在您上下文类的构造函数中：

this.Configuration.LazyLoadingEnabled = false;

**ASP.Net WebForm的开发模式**

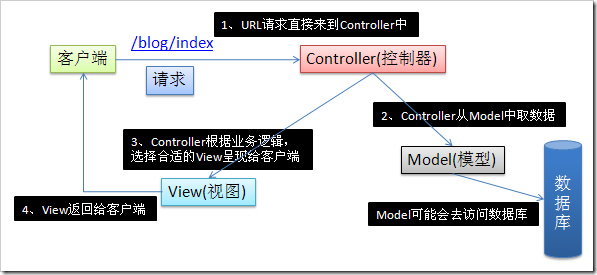


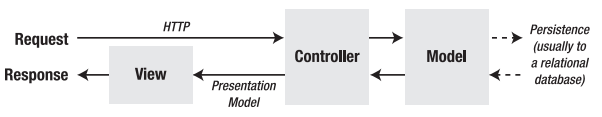
（1）处理流程

在传统的WebForm模式下，我们请求一个例如[http://www.aspnetmvc.com/blog/index.aspx](http://www.51mvc.com/blog/index.aspx)的 URL，那么我们的WebForm程序会到网站根目录下去寻找blog目录下的index.aspx文件，然后由index.aspx页面的 CodeBehind文件(.CS文件)进行逻辑处理，其中或许也包括到数据库去取出数据(其中的经过怎样的BLL到DAL这里就不谈了)，然后再由 index.aspx页面来呈现给用户。

综上所述，在WebForm模式下：**一个URL请求的是在服务器与该URL对应路径上的物理文件(ASPX文件或其他)，然后由该文件来处理这个请求并返回结果给客户端。**

**ASP.Net MVC的开发模式**





（1）处理流程

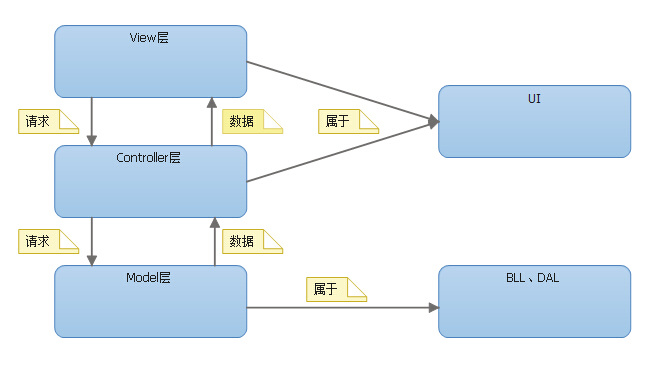
在ASP.Net MVC中，客户端所请求的URL是被映射到相应的Controller去，然后由Controller来处理业务逻辑，或许要从Model中取数据，然后 再由Controller选择合适的View返回给客户端。再说回前面我们运行的ASP.NET MVC程序访问的[http://www.aspnetmvc.com/](http://localhost:2176/Home/Index)**[Home/Index](http://localhost:2176/Home/Index)**这个URL，它访问的其实是**Home**Controller中的Index这个Action。

（2）显著特点  
• 2009年第一个开源项目版本发布，至今已过去5年，发展逐渐完善  
• 更加简洁，更加接近原始的“请求-处理-响应”  
• 更加开发、更多的新的特点、社区活跃  
• 不会取代WebForm  
• 底层跟WebForm都是一样的，只是管道上不同的处理而已

**MVC模式的两种不同解读**

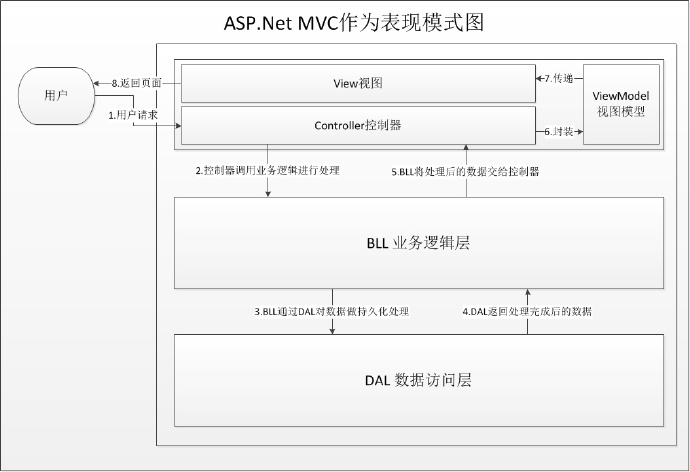
MVC 模式两种理解：一种是**表现模式**，另外一种是**架构模式**。它将应用程序分成三个主要组件即：视图（View）、控制器（Controller）和模型（Model）。现在，我们来看看M-V-C分别代表什么？

**M**：Model 主要是存储或者是处理数据的组件；Model其实是实现业务逻辑层对实体类相应数据库操作，如：CRUD。它包括数据、验证规则、数据访问和业务逻辑等应用程序信息。（补充：ViewModel：视图模型）  
**V**：View 是用户接口层组件。主要是将Model中的数据展示给用户。aspx和ascx文件被用来处理视图的职责；  
**C**： Controller 处理用户交互，从Model中获取数据并将数据传给指定的View；  
（1）MVC作为架构模式的理解

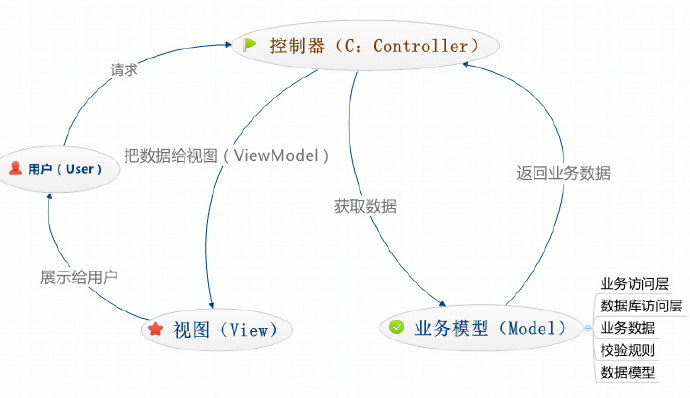


作为架构模式时，View的职责就是负责展示数据，而Controller则负责获取View传递来的数据，然后调用业务逻辑层处理完成的数据传递给 View进行展示。而Model则处理业务逻辑，并把结果返回给Controller。从传统三层架构上来看，View和Controller都属于UI 层，而Model则横跨BLL与DAL层。

（2）MVC作为表现模式的理解

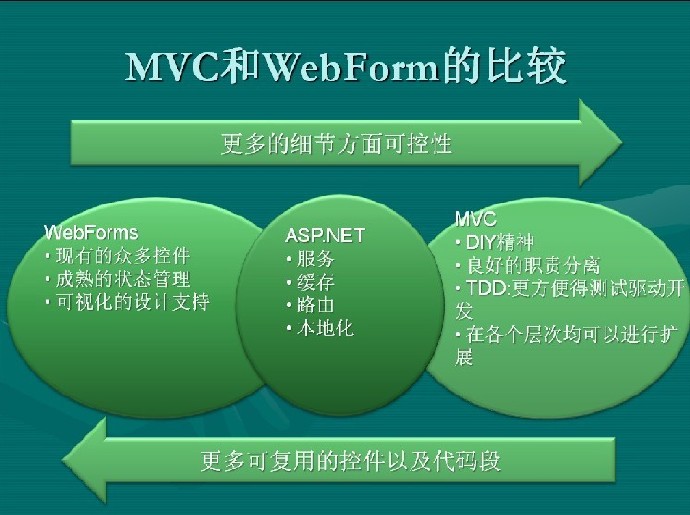


（3）MVC架构模式综合模型



可以看出，对于MVC的两种不同理解的区别就在于对于Model的理解上：**将Model作为业务模型（BLL、DAL等）还是作为视图模型（ViewModel）**。

**WebForm vs MVC**



（1）WebForm

**优点:** 1.提供了大量的服务器端控件，可以实现快速开发；

2.ViewState回传数据很方便；

3.学习成本低；

**缺点:**1. 封装太强，虽然学习成本低，很多底层东西让初学者不是很明白；

2. 自定义控制不灵活，不利于美工和开发人员的配合，往往那些服务器控件处理稍有不慎就会导致出错；

3. ViewState在页面中的传递会造成大量的流量消耗；

（2）MVC

**优点:** 1.很容易将复杂的应用分成Model(ViewModel)、View、Controller三个组件模型，将处理后台逻辑代码与前台展示逻辑进行了很好的分离，属于松耦合关系，在大项目应用中，更易于敏捷开发与测试驱动开发，有很强的可扩展性；

2.因为没有服务器端控件，所以程序员控制的会更加灵活，页面更加干净，没有ViewState；

3.通过修改路由规则，可以控制生成自定义的url，因此控制生成SEO友好的URL将更加容易；

4.强类型View实现、Razor视图、Model绑定机制、Model的验证机制，更安全高效；

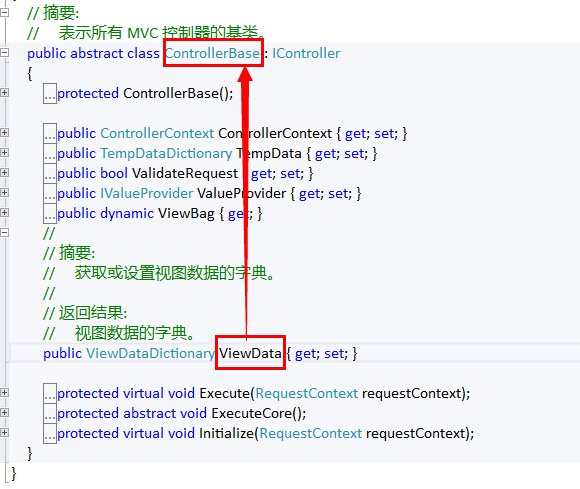
**缺点:**学习成本高，结构复杂，对未变化数据的不必要的频繁访问，也将损害操作性能。

**视图的相关约定**

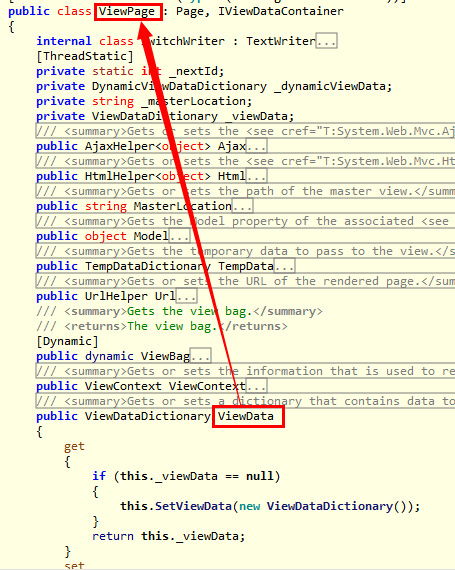
1. 所有的视图**必须放到*Views*目录下**  
   （2）不同控制器的视图用文件夹进行分割，**每个控制器都对应一个视图目录**  
   （3）一般视图名字跟控制器的Action相对应（非必须）  
   （4）多个控制器**公共的视图放到Shared：**例如公用的错误页、列表模板页、表单模板页等等；

**数据传递的桥梁-ViewData与ViewBag**

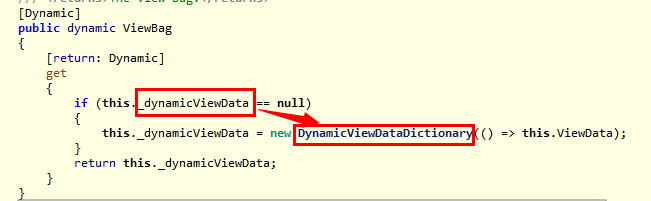
首先，ViewData是一个**Key/Value对的字典集合**数据结构，用于在Controller和View之间构建起传递数据的桥梁。  
（1）ViewData是Controller的属性，**此属性是继承ControllerBase而来**。



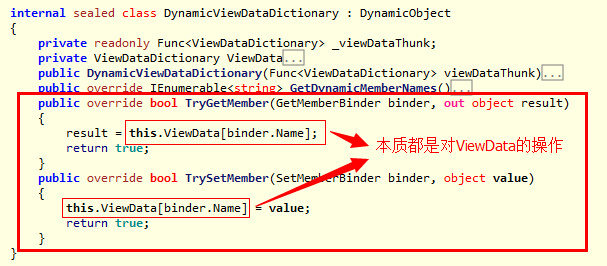
（2）ViewPage下也有一个ViewData的一个属性



（3）控制器的Action方法执行完成后，返回ViewResult，然后MVC框架在执行ExcuteResult方法时，Controller中的ViewData数据会传递给ViewPage类，其实就是**把Controller的ViewData赋值给ViewPage页面的ViewData属性**。  
（4）**ViewBag传递数据：**我们对ViewBag的动态属性进行赋值，**值实际上是存到了ViewData中**，动态属性的名存成了ViewDataDictionary的键，动态属性的值存成了ViewDataDictionary的值。



**PS：**ViweBag其实是就一个包含了一层Dynamic的ViewData，两个兄弟共用的是一个容器。



（5）ViewData与ViewBag的比较

|  |  |
| --- | --- |
| **ViewData** | **ViewBag** |
| 它是Key/Value字典集合 | 它是[dynamic](http://msdn.microsoft.com/en-us/library/dd264741.aspx)类型对像 |
| 从Asp.net MVC 1 就有了 | ASP.NET MVC3 才有 |
| 基于Asp.net 3.5 framework | 基于Asp.net 4.0与.net framework |
| ViewData比ViewBag快 | ViewBag比ViewData慢 |
| 在ViewPage中查询数据时需要转换合适的类型 | 在ViewPage中查询数据时不需要类型转换 |
| 有一些类型转换代码 | 可读性更好 |

**不要在 ViewModel 類別裡加入行為，或是加入 repository or service，這是因為 ViewModel 只是給 View 使用，只是作為資料的傳遞，**不希望因為 ViewModel 裡有行為或是可操作資料存取或是商業邏輯操作的媒介出現在 View 裡，這是避免 View 也做資料處理或是流程控制的動作，之前有看到很多人為了貪圖方便，然後在 ViewModel 類別裡建立方法或是加入 repository，就是想要在 View 裡去「寫程式」，太多的程式邏輯在 View 裡面並不恰當也不對，完全違反了 MVC 的基本原則。

**不建議的 ViewModel 使用方式**

相信有些人會在 Google 查詢 「ViewModel」，而查詢出來的第一筆的內容卻是讓我不以為然，他的作法就我所說的在 ViewModel 中含有「行為」，我在前面也有說過，在 View 的原始碼裡要避免對資料做直接的存取處理，因為 View 的責任在於顯示資料或是讓使用者輸入資料，不應該有任何的行為，否則使用 MVC 來開發網站就毫無意義了。

而有些人會說，我不會在 View 裡去做資料存取的處理，但在 ViewModel 加入 repository or service 是為了方便去判斷頁面上的資料顯示與否，例如：權限判斷，所以就必須在 ViewModel 加入 repository or service，這樣就可以在 View 裡去寫程式來做判斷，有時候為了貪圖方便或是取巧，往往會讓專案開發陷入不可預期的危機，尤其是多人開發的專案，你就會發現這樣取巧的做法，在有些開發人 員的手中是會玩出讓人驚訝的花樣。

**ASP.NET MVC 預設不會對 View 進行編譯，所以如果在 View 中加入了過多的程式邏輯，那麼在建置專案時並不會立即發現錯誤，往往要到執行後才會發現到問題與錯誤。**

一、不要為了偷懶就把好幾個類別都塞在同一個 cs 檔案中，尤其是 Repository 與 ViewModel 這兩種行為、責任截然不同的類別，這兩個根本就不應該放在一起，在日後做維護時會時常出現找不到正確類別在哪裡的問題，Respoitory 與 ViewModel 應該分開並且分別存放在不同目錄。

**二、ViewModel 是因為 View 的需求而被建立，所以是依據 View 的需求而建立不同的屬性，ViewModel 不適合使用繼承。**

三、ViewModel 內要使用 Property 而不要用 Field。

四、不要因為貪圖方便或是誤用「Reuseable」 的觀念而讓多個 View 共用一個 ViewModel，ViewModel 是因為 View 的需要而被建立，所以每個 View 都會有不同的資料需求，但實際的開發裡，真的會有很多頁面的 ViewModel 會有相同的情況，我的建議是不要共用！