第 3 章

模型和数据操作

本章导读：

MVC 的第一个字母 M 是 Model，它承载着 View 层和 Controller 之间的数据传输，是数据传输的载体，通过 Model 层解耦了视图和控制器。MVC 框架中 Model 的主要关注点是如何把请求的数据自动装配成 Action 所需要的实体类，除此之外，框架 Model 层还可以提供复合实体类自动装配、输入校验、本地化及国际化、字符集编码转换、多重输出等功能。

在本书中，数据操作是通过 Entity Framework（以下简称 EF）完成的。EF 是微软在 ADO.NET 开发以来十分重要的数据访问框架，它不但提供了标准的数据连接自动管理，也提供了对象关系映射的能力，简化程序员在编写数据访问程序时所需要的 SQL 指令编写工作，以及将查询结果转换成强类型对象的需求。随着 EF 不断进步，它也提供了在部署数据库时的辅助功能。

本章要点：

本章首先介绍 Model 的概念，然后介绍 ORM 和 EF 框架的概念，并对 EF 框架的 3 种模型进行详细的讲解，当数据库在使用的过程中需要对数据库中的表结构进行修改时，则需要使用数据库迁移完成。

首先在项目实践部分使用 EF 框架中的 Code First 方式导入现有数据库，然后创建图书销售系统其他数据模型，这些类在 ASP.NET MVC 应用程序中扮演 Model 的角色，最后使用数据迁移功能同步到数据库中。

3.1 Model 的概念

Model，即“模型”，也就是程序中的“数据”。程序是由数据和算法组成的，在 MVC 应用程序中，算法由 Controller 提供，而数据由 Model 提供。但是，Model 并不指程序内的局部变量、全局变量或常数，而是指由程序外部所提供的数据。程序外部的数据有很多种，凡是数据库、文件、Web Service（网页服务）、其他的应用程序或系统，乃至于由不同程序所演算出来的结果等，都算是 Model。所以，Model 并不仅是来自数据库的数据，也包括来自外部系统或文件的数据。

Model 本身基本上并不属于哪个应用程序项目，在大中型应用系统的设计上，Model 并不只归属于某一个应用程序，而是会特别将 Model 剥离到架构层面上，让 Model 可以被大部分应用程序所共享，并且在 Model 和实际数据源之间插入一个中介层，由这个中

介层负责与数据源进行互动，包括所熟知的 CRUD 动作。如此一来，上层应用程序只需在关注 Model 操作的情况下访问 Model，而不必担心数据源的数据管理与读写方式，这种架构在具有弹性与模块化的应用程序中相当常见。

在 ASP.NET MVC 应用程序中，Model 通常会放在项目的 Models 文件夹内，以便和其他程序进行区分，这是基于 ASP.NET MVC 应用程序所强调的“约定优于配置”原则，这个原则适合小型应用程序。若一开始就知道要开发的是大型应用程序，Models 文件夹就不适合放系统层次的 Model 对象，而只能放针对应用程序本身的 Modle，甚至不放。

Model 的类型如下。

Model 在具体实现中使用的方式有很多种，基于不同的功能与设计需求，Model 通常不会只有一种类型，而是会有多种类型。

最常见的 Model 多半是从数据源的结构而来，这些 Model 可能由 DataSet 或 DataTable 构成，用来加载与数据源互动的数据，并将数据提供给应用程序使用。此类 Model 的结构会和数据库架构相似，或是使用与数据库相同的命名规则设计，常见于以数据库为中心(Database-Centric)的 MVC 应用程序。

另一种 Model 依存于特定的程序或显示接口，此类 Model 会按照程序所需要的数据结构进行设计，而不是针对数据源，最常见的是与显示接口相互沟通的 Model，如 ViewModel，它们的存在与程序紧密结合，但通常与数据源无关，也就是数据是来自程序的处理与计算，而不是来自数据源。

ASP.NET 的大师级人物 Dino Esposito 将 ASP.NET MVC 内使用的 Model 分为三种，分别是 Domain Model、View Model 与 Input Model。

- Domain Model：与前面所述的以数据源为主的 Model 很相似，但融入了领域驱动设计的概念。
- View Model：与 View 紧密结合的 Model 类型。
- Input Model：是由用户端或外部系统端输入的 Model，Input Model 会和 MVC 的 Model Binding 机制协同合作，以提供简便的数据输入绑定方式。

无论何种类型，都可以看到一个共同的含义，就是 Model 并不限于数据，也可以是外部服务或是程序(例如商业逻辑层)，所以，在进行 Model 的设计时不能只就数据面思考，尤其是当程序和其他服务连接时。

3.2 ORM

程序员除了掌握基本的程序设计语言外，若需要访问数据库，还必须额外学习 SQL 指令，才能处理数据库端的数据访问。而各种数据库都有自己的 SQL 语法，像 Oracle 是 PL/SQL，SQL Server 是 Transact-SQL 等，SQL 本身又带有很强的关系型代数，以及基于架构(Schema-based)的特性，再加上 SQL 本身的指令和一般程序在概念上又不太相似(SQL 是一种以集合为主的语言)，造成程序员在学习上的障碍。另外，数据库端可能还会有数据库管理员(DataBase Administrator，DBA)，通常 DBA 会严格监控程序员所写的 SQL，因为 SQL 写得好与坏关乎数据库的访问性能，而这也是令很多程序员十分棘手的问题。

若能将对象自动与数据库进行映射，程序员就不必担心数据访问时还要写 SQL 的问题，用他们所熟悉的方法就可方便地访问数据库，对象关系映射（Object Relational Mapping，ORM）的概念就是在这个基础上开发出来的。

ORM 是指对象结构（Structure）和数据库架构（Schema）间的映射。使用 ORM 所设计的系统，其内部对象和数据库架构都有一定的映射规则，程序员可以不必编写 SQL，通过 ORM 提供的 Mapping Services（映射服务），将上层的对象访问指令转换成对数据库的 SQL 操作指令后，由数据库执行，再将运行结果封装成对象风格后返回给程序，所以程序员只要熟悉对象的使用就能访问数据库，无须另外学习使用 SQL。

3.3 Entity Framework 概述

Entity Framework（简称 EF）是一种 ORM 的数据访问框架，它主要为程序员提供更轻松访问数据源的功能，并试着以 ORM 的架构让程序员不必为了编写 SQL 而费心。EF 结合了 LINQ 以及 Expression 的功能，在核心层实现 SQL 生成器，以程序通过 IQueryable＜T＞指令传入的 Expression 的内容，决定如何产生 SQL 指令，而且这些指令都经过 SQL Server 工程团队的设计，特别为 SQL Server 进行了优化，所以程序员只要会应用 LINQ，就能很便捷地查询 SQL Server 数据库，而不必特别为了访问数据库而学习一堆 SQL 指令。

虽然前面只讲了 SQL Server，但其实 EF 本身是具有扩展性的框架，只要其他数据库（如 Oracle、MySQL 或是 DB2）按照 EF 提供的接口实现自己的 SQL 生成器与连接管理机制，就可以让 EF 支持不同的数据库，事实上也已经有不少数据库提供商宣布或实现支持 EF 的数据提供程序。

EF 利用数据库建模（Database Modeling）的方法将程序代码与数据库架构结合起来，以支持对象与数据库架构的串接，程序员可利用 Visual Studio 内的 ADO.NET 实体数据模型（Entity DataModel）文件调用 EF 的实体数据模型向导。

EF 包含三种建模方法，分别是数据库优先建模法（DataBase First）、模型优先建模法（Model First），以及完全以程序代码建模的程序代码优先建模法（Code First）。起初 EF 用于开发时，只提供了数据库优先的建模方法，由现有的数据库产生建模映射；到 4.0 版时，加入了模型优先建模法，允许程序员直接在项目中定义数据模型，再使用 T4 模板的方法产生数据库的设计模型，之后通过 DataContext 的生成器将设计好的模型送到数据库系统内以产生数据库。

Entity Framework 5.0 版中推出了前面提到的程序代码优先模型，程序员只需要定义出要使用的数据模型，然后在程序代码中设置数据模型间的关系，以及设置各属性的特征（例如它是 IDENTITY，或是允许 NULL 等），然后利用 DbContext 对象按照这些设置建立数据库，至于要用什么 SQL 指令，就不是程序员担心的了。

实际使用哪种类型的数据建模法，要由团队的成员配置与职务而定。若是新项目，又是以程序员为主的团队，则使用 Code First 方法较佳；若有数据库管理员，则应考虑使用 DataBase First 或是 Model First 方法，由数据库管理员进行设计，再由程序员根据模型编

写程序。另外，若使用的数据库是现有的，建议使用 DataBase First 或是 Code First 方法，若是新建数据库，则考虑使用 Model First 或 Code First 方法。

在 Visual Studio 开发环境下学习数据库应用编程时，用它自带的 SQL Server Express LocalDB 数据库实现即可，这种数据库的优点是用法简单，而且将项目和数据库从一台计算机复制到另一台计算机上时，不需要对数据库做任何单独的额外操作。

LocalDB 数据库实际上并不是为 IIS 设计的，但是，在开发环境下，由于 LocalDB 数据库使用方便，而且开发完成后，将 LocaIDB 数据库移植到其他版本的数据库中也非常容易（只需要修改项目根目录下 Web.config 中的数据库连接字符串即可，其他代码不需要做任何改变）。因此，本书中开发 Web 应用程序项目，采用 LocaIDB 数据库实现。

3.3.1　DataBase First

DataBase First 模式称为“数据库优先”，该模式假设已经有相应的数据库，可以使用 EF 设计工具根据数据库生成数据类，并使用 Visual Studio 模型设计器修改这些模型之间的对应关系。

首先创建一个控制台应用程序，然后创建实体模型，添加新建项，选择“ADO.NET 实体数据模型”表示要使用实体数据模型，并且使用 Entity Model Designer 编辑。选择 ADO.NET 实体数据模型如图 3.1 所示。

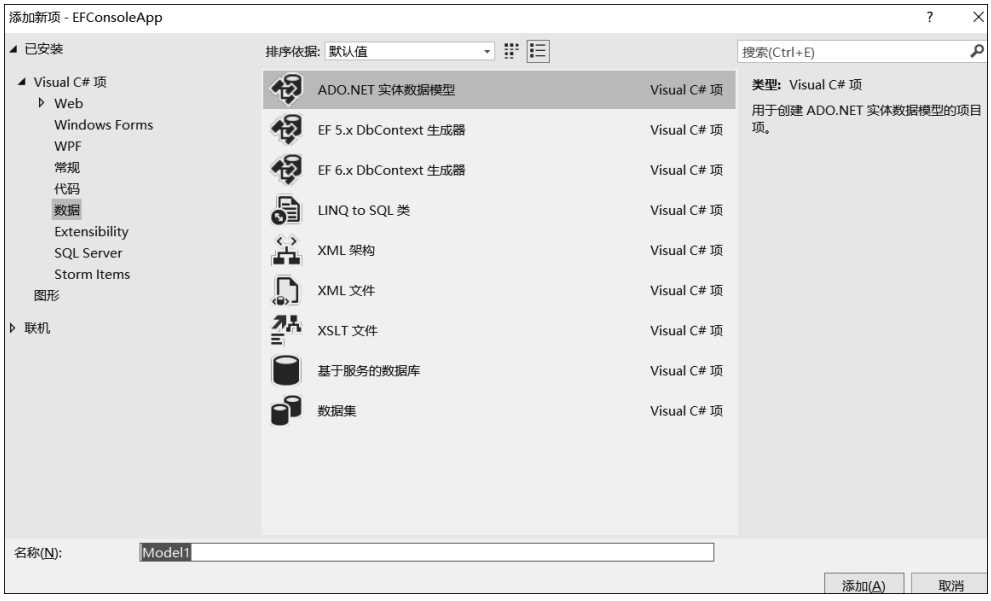

图 3.1　选择 ADO.NET 实体数据模型

下面需要与现有的数据库进行连接生成 EF 实体，在进行这一步之前，首先确定是否已经有现成的数据库，单击“添加”按钮，进入实体数据模型向导页面，如图 3.2 所示。

若使用 DataBase First 方式，其实体数据模型选择“来自数据库的 EF 设计器”，单击“下一步”按钮进入设置数据库连接字符串的屏幕显示页面，如图 3.3 所示。

图 3.2　实体数据模型向导

图 3.3　选择数据连接

若“您的应用程序应使用哪个数据连接与数据库进行连接?”下拉菜单为空,请单击右边的“新建连接”按钮设置新的数据连接。数据源可以是 SQL Server 数据库,也可以是数据库文件,如图 3.4 所示。

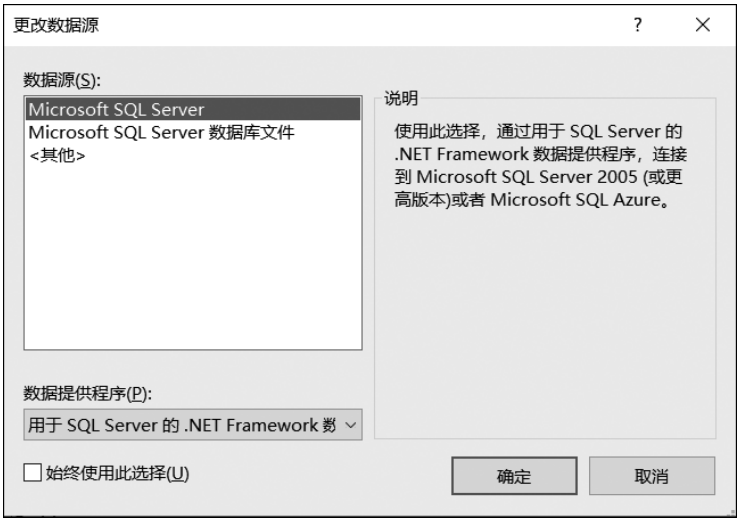

图 3.4　设置数据源

连接设置完成时,会回到“连接属性”对话框,并且默认存储连接字符串到 App.config(若是 ASP.NET 应用程序则是 Web.config),名称默认为 Model 的名称加上 Entities 字样,以本例来说就是 BookManagerEntities,单击“下一步”按钮会出现选用何种 EF 版本的对话框,如图 3.5 所示。

图 3.5　选择实体框架版本

选择最新版本实体框架 6.x,单击“完成”按钮,弹出“选择您的数据库对象和设置”页面,如图 3.6 所示。

图 3.6　选择数据库对象

在“选择您的数据库对象和设置”页面中首先需要选择在模型中包括哪些数据库对象，选择项包括数据“表”“视图”及“存储过程和函数”，此外还有三个选项：第一个选项是“确定所生成对象名称的单复数形式”，这个选项的功能是要求 EF 在生成类对象时，分析数据库的名称决定 Entity、EntitySet 及 Navigation Property 三者的名称是单数或是复数。第二个选项是“在模型中包括外键列”，表示要在生成出来的 Model 中加入 Navigation Property，不过只限于数据表内有明确设置 Foreign Key Constraint 的才会加入。第三个选项是“将所选存储过程和函数导入实体模型中”，表示若选择了存储过程和函数，向导会将这些存储过程和函数的设置加到实体模型内，这样就可以利用 ObjectContext＜T＞的 ExecuteFunction()调用，而不需使用 ExecuteStoreCommand()调用。

选择好要导入模型的数据库对象和三个选项后，如果不需要修改模型命名空间，单击“完成”按钮，系统会帮助生成数据库实体类及 EDMX 的定义文件。

生成的文件目录如图 3.7 所示。

创建完实体模型后，会自动生成 Books、BookTypes、Roles、Users 和 Carts 实体类和一个 BookManagerEntities 数据库上下文操作类，双击 Model1.edmx 会显示如图 3.8 所示的实体关系图。

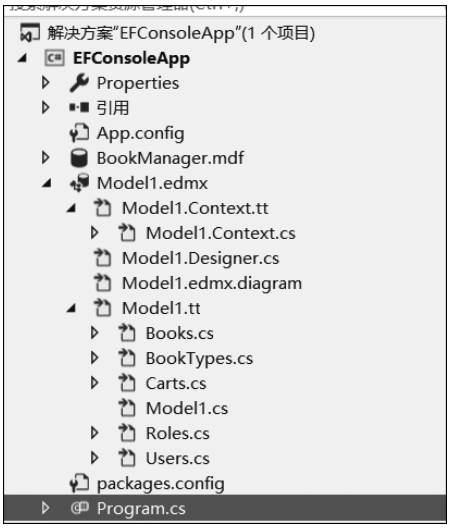

图 3.7　生成的文件目录

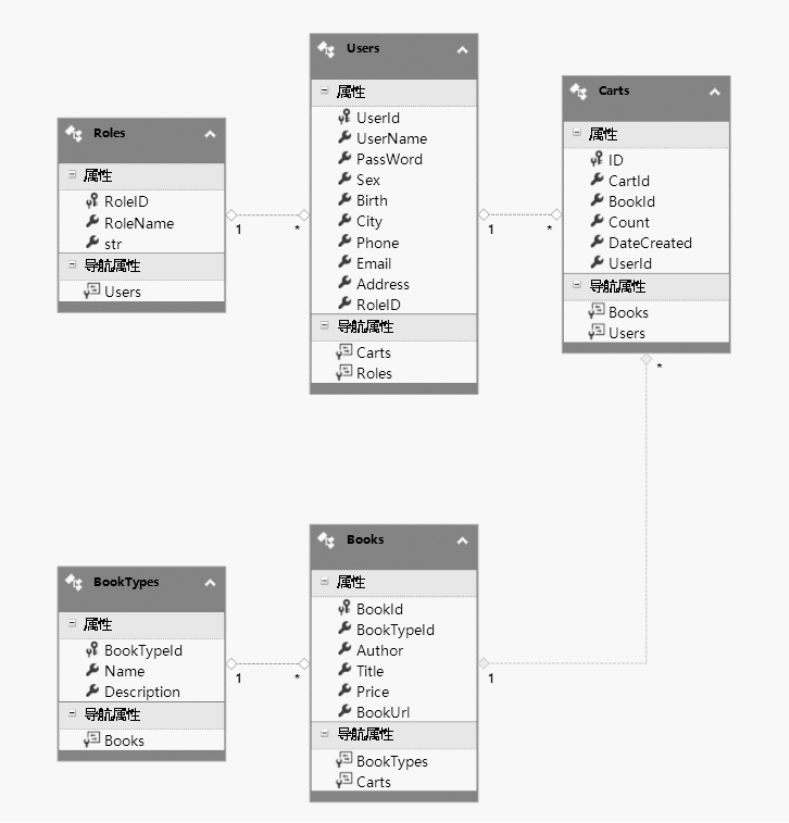

图 3.8　实体关系图

使用 DataBase First 所生成的模型，会使用 T4（Text Template Transformation Toolkit）模板进行转换，在项目内的.tt 文件就是文字模板（Text Template）文件，它是用来生成数据库对象所对应的程序代码文件。

这样就拥有访问数据库的模型了，只要编写简单的数据访问代码，就可以对数据库进行增加、删除、修改、查找操作了。下面简单看一下如何使用 EF 进行数据查询，通过下面的代码可以看到 EF 对于数据的操作非常容易。

```
static void Main(string[] args)
{
    using (var db = new BookManagerEntities())
    {
        IQueryable<Books> books = from book in db.Books
                                  where book.Author == "王雪"
                                  select book;
        foreach (Books p in books)
        {
            Console.WriteLine("书名是 :{0}", p.Title);
        }
    }
    Console.ReadLine();
}
```

在程序入口的 Main()函数中实例化 BookManagerEntities 数据操作类，使用 Linq 语句对 Books 中的数据做投影查询，过滤条件为作者“王雪”，最后输出到控制台。运行结果如图 3.9 所示。

图 3.9　运行结果

3.3.2　Model First

Model First 是 EF 4 开始新增的功能，主要提供给目前没有数据库，但又需要使用 EF 设计模型的程序员使用。正如其名，程序员要先在 Designer 内设计好模型的结构，再利用这个结构生成数据库。

首先，与 DataBase First 一样，新增一个 ADO.NET 实体数据模型，但这次选择的是“空 EF 设计器模型”，然后单击“完成”按钮，如图 3.10 所示。

这时会出现一个空白的 Designer，并且工具箱也会出现必要的模板，就像使用 Windows Forms 的窗体设计器一样，由工具箱拖拉出模板放到 Designer 的空间内，以此建立模型。建立模型空窗体如图 3.11 所示。

图 3.10　选择实体数据模型

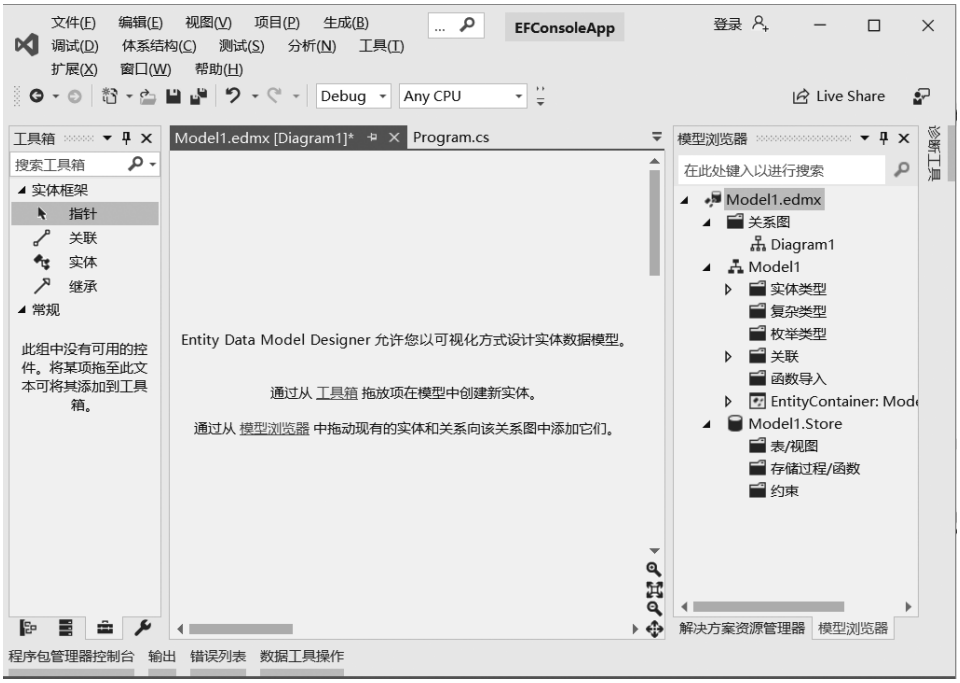

图 3.11　建立模型空窗体

接着，建立数据模型，方法很简单，从左边的“工具箱”中将“实体”拖放到 Designer 内，就会产生一个新的模型，然后将名称改为 Roles 和 Users，并在模型上右击，从弹出的快捷菜单中选择“新增”→“标量属性”命令，并给其命名，设计好实体后可以添加关联关系，如图 3.12 所示。

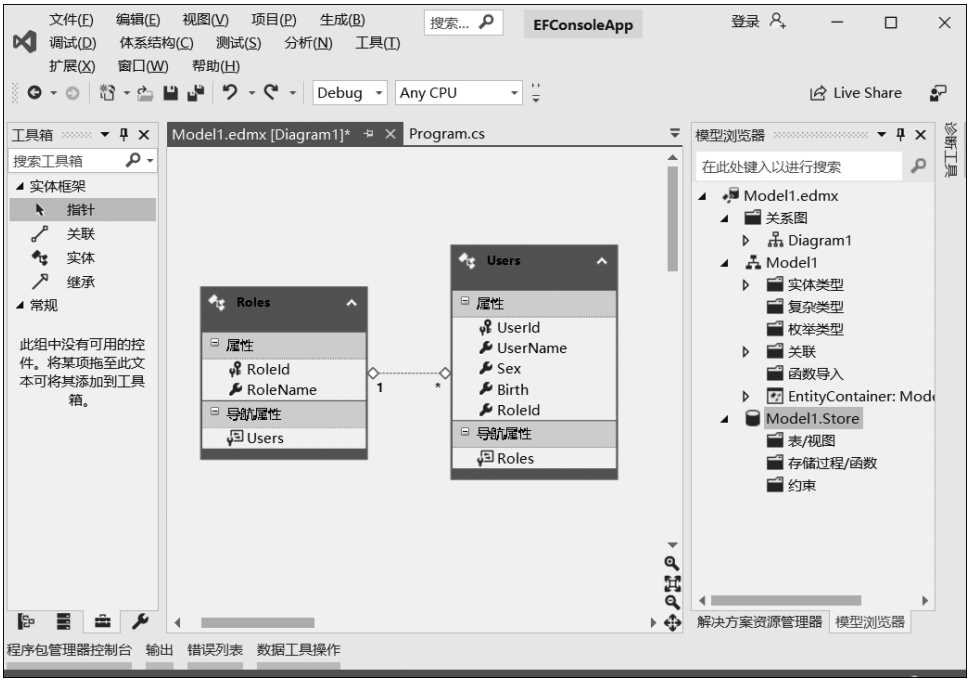

图 3.12　建立模型及关联关系

下面设置每个字段的数据类型，在字段上右击，从弹出的快捷菜单中选择“属性”命令，在“属性”窗口的“类型”中选择所需的类型即可，如图 3.13 所示。

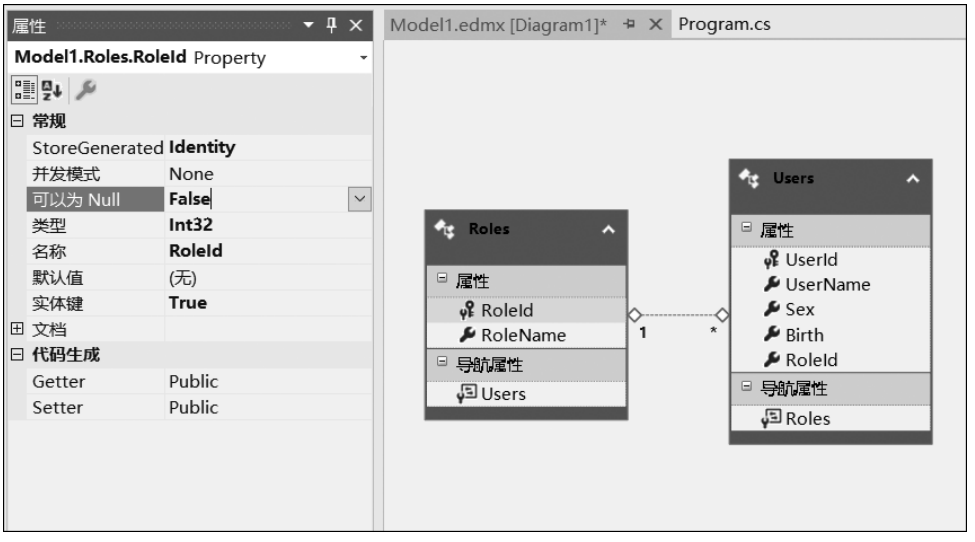

图 3.13　设置字段属性

在建立关联的同时，还会产生“导航属性”选项，通过导航属性，就能直接浏览关联好的对象。设置完成后，在 Designer 的空白处单击，可以看到属性窗口内会出现关于这个 Model 的设置，将“以复数形式表示新对象”设为 true，会让 User 在产生数据表时将名称设为 Users，Role 会设为 Roles。至此，设计已经完成，但要将它生成为数据库之前，还需要做一个设置，就是需要一个 DbContext 对象。要产生 DbContext 对象的方法也不难，只要在 Designer 上右击，选择“添加代码生成项”命令即可，如图 3.14 所示。

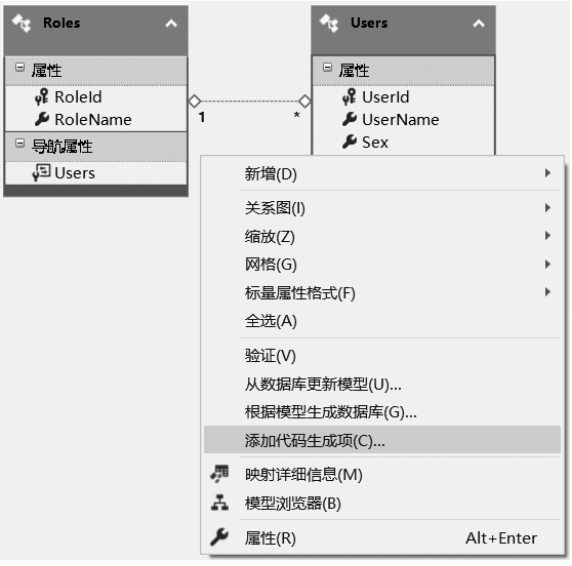

图 3.14　根据模型生成代码

此时会出现添加 DbContext 生成器的选项，选择“EF 6.x DbContext 生成器”，可以修改 Model 的名字，再单击“添加”按钮，如图 3.15 所示。

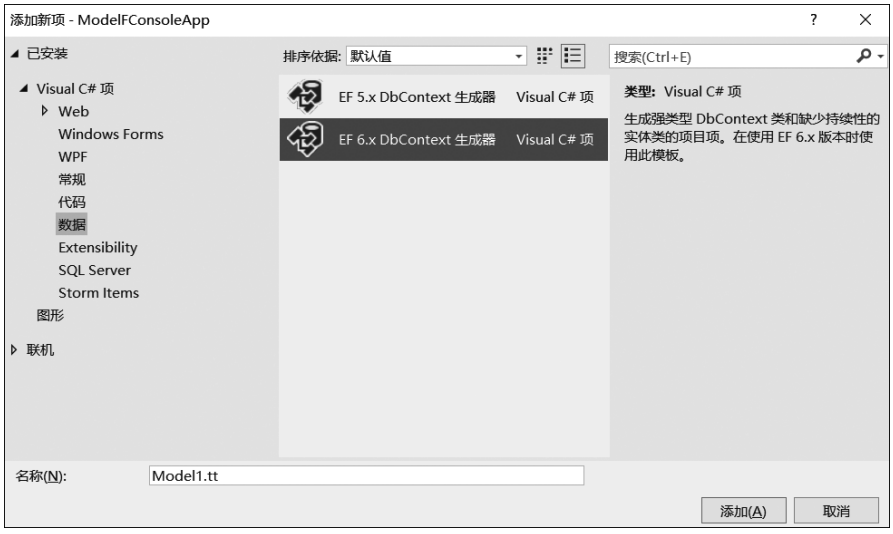

图 3.15　选择生成 DbContext 的生成器

此时就能在项目中看到 Model1.edmx 下出现了 Model1.context.tt 文件，以及 Model1.tt 下包含了 Model1.Context.cs 及在 Designer 中所设计的 Model 的程序代码文件。在 Designer 的空白处右击，从弹出的快捷菜单中选择“根据模型生成数据库”命令，如图 3.16 所示。

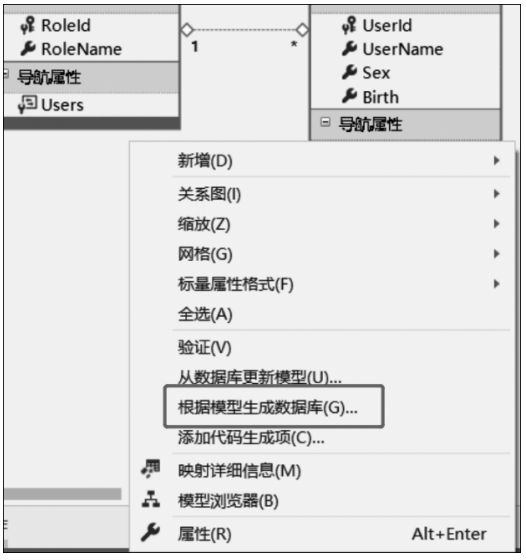

图 3.16　根据模型生成数据库

下一步是设置数据库连接属性，数据源选择 Microsoft SQL Server 数据库文件(SqlClient)，输入数据库文件名后，单击“确定”按钮，如图 3.17 所示。

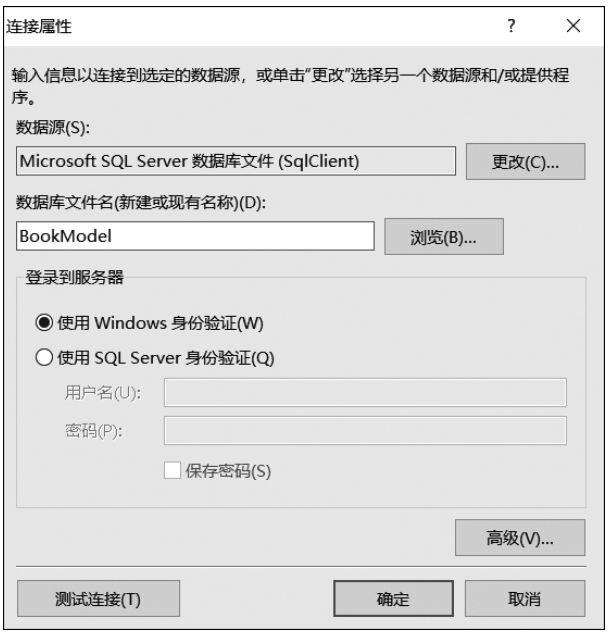

图 3.17　设置数据库连接属性

在数据连接的窗口中设置要连接的数据库，在“使用那个数据库连接与数据库进行连接”下拉菜单中输入 BookModel，然后单击“确定”按钮，这时会出现是否要建立新数据库的对话框，单击“是”按钮，建立新数据库。接着会出现数据库架构脚本生成的屏幕显示画面，如图 3.18 所示。

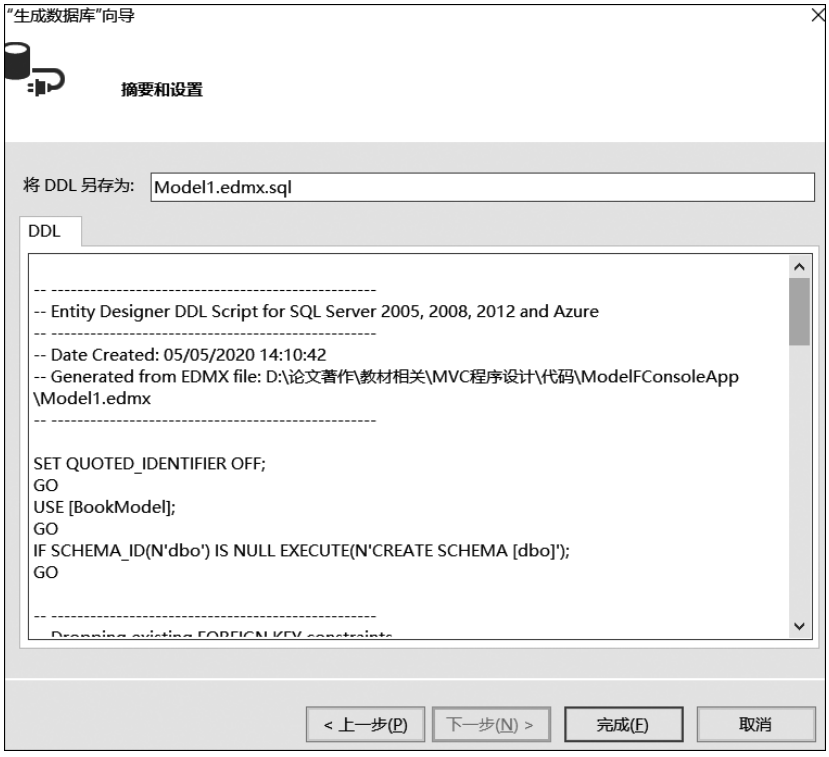

图 3.18　设置数据库连接属性

单击“完成”按钮，Visual Studio 将自动生成 DDL 文件并打开，如图 3.19 所示。

Model First 生成数据库及数据对象后，就可以使用程序代码处理数据库的访问工作了，这部分与 Database First、Code First 都一样，所以留在 Code First 的部分一起说明。

3.3.3　Code First

Code First 模式最早是从 EF 4 开始的，利用 EF 6 模板和已存在的数据库创建实体模型在前面的学习中已经介绍，EF 6（Entity Framework 6）提供的 Code First 模式分为两种情况：一种情况是数据库已经存在，如果系统开发前已经存在数据库，则选择“来自数据库的 Code First”，步骤和 Database First 类似，在该模式中取消了 edmx 模型和 T4 模板，直接生成 EF 上下文和相应的类，该模式出现在 Visual Studio 2015 版本以后。

另一种情况是还不存在数据库，那么选择“空 Code First 模型”，添加完成后，Visual Studio 会打开生成好的 Code First 程序代码。正如 Code First，定义的规则全部由程序代码处理，所以这也意味着使用 Code First 方式进行建模时，可以直接使用类的程序代码，而不一定用 ADO.NET 实体数据模型的方式产生。

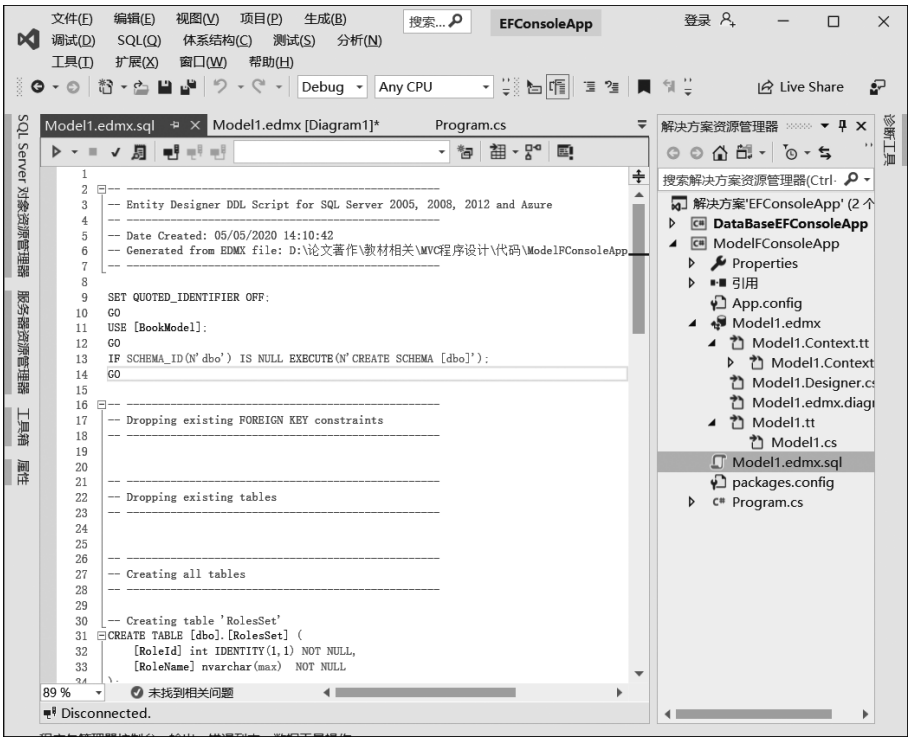

图 3.19 DDL 文件

创建一个控制台应用程序，然后添加新建项，选择“ADO.NET 实体数据模型”，弹出实体数据模型向导，如图 3.20 所示。

图 3.20 实体数据模型向导

EF 6 提供的“来自数据库的 Code First”模板适用于数据库已经存在的情况，利用它可自动生成 C＃模型类代码和 C＃数据上下文类代码。或者说，利用该模板既可以根据生成的代码理解数据库表中字段类型和 C＃数据类型之间的对应关系，又可以简化 Code First 代码编写的工作量。

“空 Code First 模型”和“来自数据库的 Code First”用法相似，适用于不存在数据库，但是希望借助它先帮助自动生成部分 C＃代码的情况。和前面的 DataBase First 与 Model First 一样，可以通过添加 ADO.NET 实体数据模型生成模型。

本节主要介绍数据库不存在时 Code First 模式的基本用法，目的是让读者理解相关的概念。但是一定要记住，Code First 模式既可用于现有数据库，也可以创建数据库。在实体数据模型向导中选择“空 Code First 模型”，系统会自动生成一个类。注意，自动生成的模型都需要继承自 DbContext，可以看到，微软已经给了很多提示，例如，需要自己配置连接数据库的字符串；创建的模型类都需要在 DbContext 实体类中添加 DbSet，代码如下所示。

```
两个引用
public class Model1 : DbContext
{
    //您的上下文已配置为从您的应用程序的配置文件(App.config 或 Web.config)
    //使用“Model1”连接字符串。默认情况下，此连接字符串针对您的 LocalDb 实例上的
    //“CRConsoleApp.Model1”数据库
    //
    //如果您想针对其他数据库和/或数据库提供程序，请在应用程序配置文件中修改“Model1”
    //连接字符串
    1 个引用
    public Model1()
        : base("name=Model1")
    {
    }
    //为您要在模型中包含的每种实体类型都添加 DbSet。有关配置和使用 Code First  模型
    //的详细信息，请参阅 http://go.microsoft.com/fwlink/?LinkId=390109。

    // public virtual DbSet<MyEntity> MyEntities { get; set; }
}

//public class MyEntity
//{
//    public int Id { get; set; }
//    public string Name { get; set; }
//}
```

几乎所有的管理软件都具有权限管理。下面新建两个模型类用户模型 Users 和角色模型 Roles，用于对图书馆里系统的权限进行管理。需要根据提示引入相应的命名空间，使用构造函数设置属性初值，并且需要用 Key 为自己的表指定主键。

```
public class Roles
{
      public Roles()
        {
            Users = new HashSet<Users>();
        }
        [key]
        public int RoleID { get; set; }
        public string RoleName { get; set; }
        public virtual ICollection<Users> Users;
```

```
    }

public class Users
{
    public Users()
    {
            IsValid = true;
            Birth = DateTime.Now;
            Sex = "男";
            RoleID = 3;
    }
    [key]
    public int UserId { get; set; }
    public string UserName { get; set; }
    public string Password { get; set; }
    public string Sex { get; set; }
    public DateTime? Birth { get; set; }
    public string City { get; set; }
    public string Phone { get; set; }
    public string Email { get; set; }
    public string Address { get; set; }
    public bool IsValid{ get; set; }
    public int RoleID { get; set; }
    public Roles Roles { get; set; }
}
```

在 Roles 和 Users 模型中，除了将数据库中的字段定义为类的属性外，还为 Roles 添加了类型为 ICollection＜Users＞的 Users 对象，并在构造方法中实例化；为 Users 模型添加类型为 Roles 的 Roles 对象，表明 Roles 和 Users 模型的关系为一对多。

定义好模型后，需要按照微软提供的模板将这两个类添加到 DbSet。这里的“name = Model1”表示使用名字为 Model1 的字符串连接数据库。

```
public class Model1:DbContext
{
    //连接字符串
    public Model1()
        : base("name=Model1")
    {
    }
    public virtual DbSet<Roles> Roles { get; set; }
    public virtual DbSet<Users> Users { get; set; }
}
```

下面是项目根目录下的 App.config 中数据库连接字串的配置，名为 Model1 的连接字串对应的数据库名称为 EFConsoleApp.Model1。

```
<connectionStrings>
<add name="Model1" connectionString="data source=(LocalDb)\MSSQLLocalDB;
initial catalog = EFConsoleApp. Model1; integrated security = True;
MultipleActiveResultSets=True;App=EntityFramework" providerName="System.
Data.SqlClient" />
</connectionStrings>
```

然后在 Program 的 Main()函数中写操作数据库的代码，用来生成数据库。

```
static void Main(string[] args)
{
    using (var db = new Model1())
    {
        var role = new Roles();
        role.RoleName = "超级管理员";
        db.Roles.Add(role);
        role = new Roles();
        role.RoleName = "管理员";
        db.Roles.Add(role);
        role = new Roles();
        role.RoleName = "普通用户";
        db.Roles.Add(role);
        db.SaveChanges();
        IQueryable<Roles> roles = from r in db.Roles
                                  select r;
        foreach (Roles r in roles)
        {
            Console.WriteLine("角色是 :{0}", r.RoleName);
        }
    }
    Console.ReadLine();
}
```

定义并实例化数据库操作类 Model1，为 Roles 表添加超级管理员、管理员和普通用户后，执行 db.SaveChanges()语句将数据保存在数据库中。运行程序，可以看到运行结果如图 3.21 所示。

图 3.21　运行结果

程序运行后，在控制台中查询到三条角色列表，说明使用 Code First 方式创建了数据库，并且包含 Roles 和 Users 两张表，Roles 表中插入了以上三条数据，那么创建的数据库文件在哪里呢？

可以打开 Visual Studio 视图菜单下的 SQL Server 对象资源管理器查看数据库文件，使用 Code First 方式生成的数据库如图 3.22 所示。

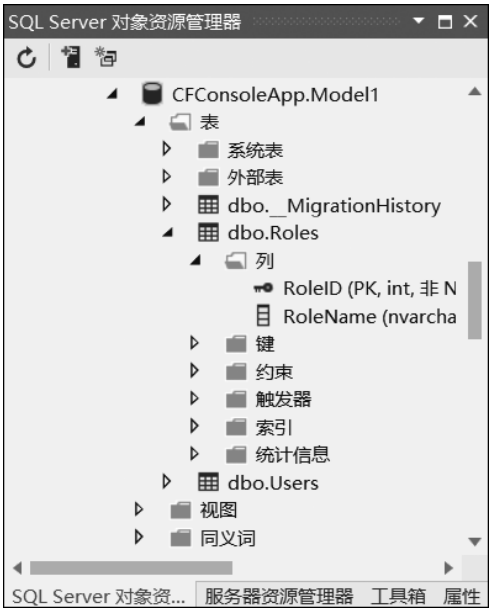

图 3.22　使用 Code First 方式生成的数据库

3.3.4　数据库初始化

在系统开发初期，通常不会有数据库，要自行创建数据库之后，才能进行后续程序的开发。按照 ORM 的概念，程序员应该不需要编写 CREATE DATABASE 语句，所以 DbContext 提供了两个方法创建数据库，分别是 Create()、CreateIfNotExists()，前者可创建数据库，但若数据库已存在，则会弹出提示；后者会判断数据库是否已存在，若存在，则不做任何动作，否则会创建数据库。有新增数据库的指令，当然也会有删除数据库的指令，若要删除数据库，只要调用 Delete()即可。

```
db.Database.Create();
db.Database.CreateIfNotExists();
db.Database.Delete();
```

不过，对于程序员来说，数据库的新增和删除不是重点，重点是在新增数据库时要额外做一些数据新增的工作，这在部署系统时很常见，每次在部署有数据库的系统时，都要产生大量 SQL 指令带到目标环境执行，对程序员来说并不十分方便，因此 ORM 通常会提供一些方法处理这部分工作。EF 在这部分提供了数据库初始化器（DataBase Initializer）的功能，内建了以下 4 种方法。

- 在数据库不存在时创建数据库

```
Database.SetInitializer<Model1>(new CreateDatabaseIfNotExists< Model1>());
```

- 在模型更改时创建数据库

```
Database.SetInitializer<Model1>(new DropCreateDatabaseIfModelChanges
< Model1>());
```

- 每次启动应用程序时创建数据库

```
Database.SetInitializer<Model1>(new DropCreateDatabaseAlways < Model1>());
```

- 从不创建数据库

```
Database.SetInitializer< Model1>(null);
```

除此之外，EF 也提供了 IDatabaseInitializer＜TContext＞接口供程序员使用，以开发出适合自己数据库初始化器的功能，并且可加入一些所需的新增数据。IDatabase-Initialize＜TContex＞只有一个方法 InitializeDatabase()，只实现这个方法即可，例如：

```
public class DbInit:IDatabaseInitializer< Model1>
    {
        public void InitializeDatabase(Model1 context)
        {
            context.Database.CreateIfNotExists();
            context.Users.Add(
                new Users()
                {
                    UserName = "Test",
                    Password = "t123",
                    Phone = "18695553888",
                    Email = "wangyy@163.com"
                });
            context.SaveChanges();
        }
  }
```

若要驱动 DbInit 类，需要两步：首先是设置要使用的初始化器，可用 Database.SetInitializer()实现；然后是在 DbContext 的生命周期内调用 DbContext.Initialize()，并传入是否要强制执行 Database Initializer 的参数。

```
Database.SetInitializer<Model1>(new DbInit());
using (var db = new Model1())
{
    db.Database.Initialize(true);
}
```