

实体框架
专业人员笔记

实体框架

Entity Framework

Notes for Professionals



80+ 页
专业提示和技巧

80+ pages
of professional hints and tricks

目录

关于	1
第1章：Entity Framework入门	2
第1.1节：安装Entity Framework NuGet包	2
第1.2节：从C#使用实体框架（代码优先）	4
第1.3节：什么是实体框架？	5
第2章：代码优先约定	6
第2.1节：移除约定	6
第2.2节：主键约定	6
第2.3节：类型发现	6
第2.4节：DecimalProperty约定	7
第2.5节：关系约定	9
第2.6节：外键约定	10
第3章：Code First 数据注解	11
第3.1节：[Column] 属性	11
第3.2节：[DatabaseGenerated] 属性	11
第3.3节：[Required] 属性	12
第3.4节：[MaxLength] 和 [MinLength] 属性	12
第3.5节：[InverseProperty(string)] 属性	13
第3.6节：[ComplexType] 属性	14
第3.7节：[ForeignKey(string)] 属性	15
第3.8节：[Range(min,max)] 属性	15
第3.9节：[NotMapped] 属性	16
第3.10节：[Table] 属性	17
第3.11节：[Index] 属性	17
第3.12节：[Key] 属性	18
第3.13节：[StringLength(int)] 属性	19
第3.14节：[Timestamp] 属性	19
第3.15节：[ConcurrencyCheck] 属性	20
第4章：实体框架代码优先	21
第4.1节：连接到现有数据库	21
第5章：实体框架代码优先迁移	23
第5.1节：启用迁移	23
第5.2节：添加你的第一个迁移	23
第5.3节：在代码中执行“Update-Database”	25
第5.4节：迁移期间的数据种子	25
第5.5节：Entity Framework Code First初始迁移步骤	26
第5.6节：迁移期间使用Sql()	27
第6章：EntityFramework（Code First）中的继承	29
第6.1节：层次表	29
第6.2节：每种类型的表	29
第7章：代码优先 - Fluent API	31
第7.1节：映射模型	31
第7.2节：复合主键	32
第7.3节：最大长度	33
第7.4节：主键	33
第7.5节：必需属性（非空）	34

Contents

About	1
Chapter 1: Getting started with Entity Framework	2
Section 1.1: Installing the Entity Framework NuGet Package	2
Section 1.2: Using Entity Framework from C# (Code First)	4
Section 1.3: What is Entity Framework?	5
Chapter 2: Code First Conventions	6
Section 2.1: Removing Conventions	6
Section 2.2: Primary Key Convention	6
Section 2.3: Type Discovery	6
Section 2.4: DecimalPropertyConvention	7
Section 2.5: Relationship Convention	9
Section 2.6: Foreign Key Convention	10
Chapter 3: Code First DataAnnotations	11
Section 3.1: [Column] attribute	11
Section 3.2: [DatabaseGenerated] attribute	11
Section 3.3: [Required] attribute	12
Section 3.4: [MaxLength] and [MinLength] attributes	12
Section 3.5: [InverseProperty(string)] attribute	13
Section 3.6: [ComplexType] attribute	14
Section 3.7: [ForeignKey(string)] attribute	15
Section 3.8: [Range(min,max)] attribute	15
Section 3.9: [NotMapped] attribute	16
Section 3.10: [Table] attribute	17
Section 3.11: [Index] attribute	17
Section 3.12: [Key] attribute	18
Section 3.13: [StringLength(int)] attribute	19
Section 3.14: [Timestamp] attribute	19
Section 3.15: [ConcurrencyCheck] Attribute	20
Chapter 4: Entity Framework Code First	21
Section 4.1: Connect to an existing database	21
Chapter 5: Entity framework Code First Migrations	23
Section 5.1: Enable Migrations	23
Section 5.2: Add your first migration	23
Section 5.3: Doing “Update-Database” within your code	25
Section 5.4: Seeding Data during migrations	25
Section 5.5: Initial Entity Framework Code First Migration Step by Step	26
Section 5.6: Using Sql() during migrations	27
Chapter 6: Inheritance with EntityFramework (Code First)	29
Section 6.1: Table per hierarchy	29
Section 6.2: Table per type	29
Chapter 7: Code First - Fluent API	31
Section 7.1: Mapping models	31
Section 7.2: Composite Primary Key	32
Section 7.3: Maximum Length	33
Section 7.4: Primary Key	33
Section 7.5: Required properties (NOT NULL)	34

第7.6节：显式外键命名	34
第8章：使用实体框架代码优先映射关系：一对一及变体	36
第8.1节：映射一对零或一	36
第8.2节：一对一映射	39
第8.3节：映射一对零或一对零	40
第9章：使用实体框架代码优先映射关系：一对多和多对多	41
第9.1节：映射一对多	41
第9.2节：一对多映射：违背惯例	42
第9.3节：零或一对多映射	43
第9.4节：多对多映射	44
第9.5节：多对多关系：自定义连接表	45
第9.6节：多对多：自定义连接实体	46
第10章：数据库优先模型生成	49
第10.1节：从数据库生成模型	49
第10.2节：向生成的模型添加数据注解	50
第11章：复杂类型	52
第11.1节：代码优先复杂类型	52
第12章：数据库初始化器	53
第12.1节：如果不存在则创建数据库	53
第12.2节：如果模型更改则删除并创建数据库	53
第12.3节：DropCreateDatabaseAlways	53
第12.4节：自定义数据库初始化器	53
第12.5节：将数据库迁移到最新版本	54
第13章：跟踪与非跟踪	55
第13.1节：非跟踪查询	55
第13.2节：跟踪查询	55
第13.3节：跟踪与投影	55
第14章：事务	57
第14.1节：Database.BeginTransaction()	57
第15章：管理实体状态	58
第15.1节：设置单个实体的Added状态	58
第15.2节：设置对象图的Added状态	58
第16章：加载相关实体	60
第16.1节：急切加载	60
第16.2节：显式加载	60
第16.3节：延迟加载	61
第16.4节：投影查询	61
第17章：模型约束	63
第17.1节：一对多关系	63
第18章：使用PostgreSQL的实体框架	65
第18.1节：使用Npgsql数据提供程序配合Entity Framework 6.1.3和PostgreSQL的前置步骤	65
NpgsqlDbContextProvider	65
第19章：使用SQLite的实体框架	66
第19.1节：设置项目以使用带有SQLite提供程序的实体框架	66
第20章：实体框架中的.t4模板	69
第20.1节：动态添加接口到模型	69

Section 7.6: Explicit Foreign Key naming	34
Chapter 8: Mapping relationship with Entity Framework Code First: One-to-one and variations	36
Section 8.1: Mapping one-to-zero or one	36
Section 8.2: Mapping one-to-one	39
Section 8.3: Mapping one or zero-to-one or zero	40
Chapter 9: Mapping relationship with Entity Framework Code First: One-to-many and Many-to-many	41
Section 9.1: Mapping one-to-many	41
Section 9.2: Mapping one-to-many: against the convention	42
Section 9.3: Mapping zero or one-to-many	43
Section 9.4: Many-to-many	44
Section 9.5: Many-to-many: customizing the join table	45
Section 9.6: Many-to-many: custom join entity	46
Chapter 10: Database first model generation	49
Section 10.1: Generating model from database	49
Section 10.2: Adding data annotations to the generated model	50
Chapter 11: Complex Types	52
Section 11.1: Code First Complex Types	52
Chapter 12: Database Initialisers	53
Section 12.1: CreateDatabaseIfNotExists	53
Section 12.2: DropCreateDatabaseIfModelChanges	53
Section 12.3: DropCreateDatabaseAlways	53
Section 12.4: Custom database initializer	53
Section 12.5: MigrateDatabaseToLatestVersion	54
Chapter 13: Tracking vs. No-Tracking	55
Section 13.1: No-tracking queries	55
Section 13.2: Tracking queries	55
Section 13.3: Tracking and projections	55
Chapter 14: Transactions	57
Section 14.1: Database.BeginTransaction()	57
Chapter 15: Managing entity state	58
Section 15.1: Setting state Added of a single entity	58
Section 15.2: Setting state Added of an object graph	58
Chapter 16: Loading related entities	60
Section 16.1: Eager loading	60
Section 16.2: Explicit loading	60
Section 16.3: Lazy loading	61
Section 16.4: Projection Queries	61
Chapter 17: Model Constraints	63
Section 17.1: One-to-many relationships	63
Chapter 18: Entity Framework with PostgreSQL	65
Section 18.1: Pre-Steps needed in order to use Entity Framework 6.1.3 with Npgsql using NpgsqlDbContextProvider	65
Chapter 19: Entity Framework with SQLite	66
Section 19.1: Setting up a project to use Entity Framework with an SQLite provider	66
Chapter 20: .t4 templates in entity framework	69
Section 20.1: Dynamically adding Interfaces to model	69

第20.2节：向实体类添加XML文档	69
第21章：高级映射场景：实体拆分，表拆分	71
第21.1节：实体拆分	71
第21.2节：表拆分	72
第22章：实体框架的最佳实践（简单与专业）	73
第22.1节：1- 实体框架 @ 数据层（基础）	73
第22.2节：2- 实体框架 @ 业务层	76
第22.3节：3- 在表示层（MVC）使用业务层	79
第22.4节：单元测试层的4-实体框架	81
第23章：EF中的优化技术	85
第23.1节：使用AsNoTracking	85
第23.2节：尽可能在数据库中执行查询，而非内存中	85
第23.3节：仅加载所需数据	85
第23.4节：异步并行执行多个查询	86
第23.5节：处理存根实体	87
第23.6节：禁用更改跟踪和代理生成	88
鸣谢	89
你可能还喜欢	90

Section 20.2: Adding XML Documentation to Entity Classes	69
Chapter 21: Advanced mapping scenarios: entity splitting, table splitting	71
Section 21.1: Entity splitting	71
Section 21.2: Table splitting	72
Chapter 22: Best Practices For Entity Framework (Simple & Professional)	73
Section 22.1: 1- Entity Framework @ Data layer (Basics)	73
Section 22.2: 2- Entity Framework @ Business layer	76
Section 22.3: 3- Using Business layer @ Presentation layer (MVC)	79
Section 22.4: 4- Entity Framework @ Unit Test Layer	81
Chapter 23: Optimization Techniques in EF	85
Section 23.1: Using AsNoTracking	85
Section 23.2: Execute queries in the database when possible, not in memory	85
Section 23.3: Loading Only Required Data	85
Section 23.4: Execute multiple queries async and in parallel	86
Section 23.5: Working with stub entities	87
Section 23.6: Disable change tracking and proxy generation	88
Credits	89
You may also like	90

请随意免费分享此 PDF，
本书的最新版可从以下网址下载：
<https://goalkicker.com/EntityFrameworkBook>

本Entity Framework 专业人士笔记一书汇编自StackOverflow 文档，内容
由 StackOverflow 的优秀人士撰写。文本内容采用知识共享署名-相同方式共享许可
协议发布，详见本书末尾对各章节贡献者的致谢。图片版权归各自所有者所有，除非
另有说明

这是一本非官方的免费书籍，旨在教育用途，与官方 Entity Framework 组
织或公司及 StackOverflow 无关。所有商标和注册商标均为其各自公司所
有者所有

本书所提供的信息不保证正确或准确，使用风险自负

请将反馈和更正发送至web@petercv.com

Please feel free to share this PDF with anyone for free,
latest version of this book can be downloaded from:
<https://goalkicker.com/EntityFrameworkBook>

This *Entity Framework Notes for Professionals* book is compiled from [Stack Overflow Documentation](#), the content is written by the beautiful people at Stack Overflow. Text content is released under Creative Commons BY-SA, see credits at the end of this book whom contributed to the various chapters. Images may be copyright of their respective owners unless otherwise specified

This is an unofficial free book created for educational purposes and is not affiliated with official Entity Framework group(s) or company(s) nor Stack Overflow. All trademarks and registered trademarks are the property of their respective company owners

The information presented in this book is not guaranteed to be correct nor accurate, use at your own risk

Please send feedback and corrections to web@petercv.com

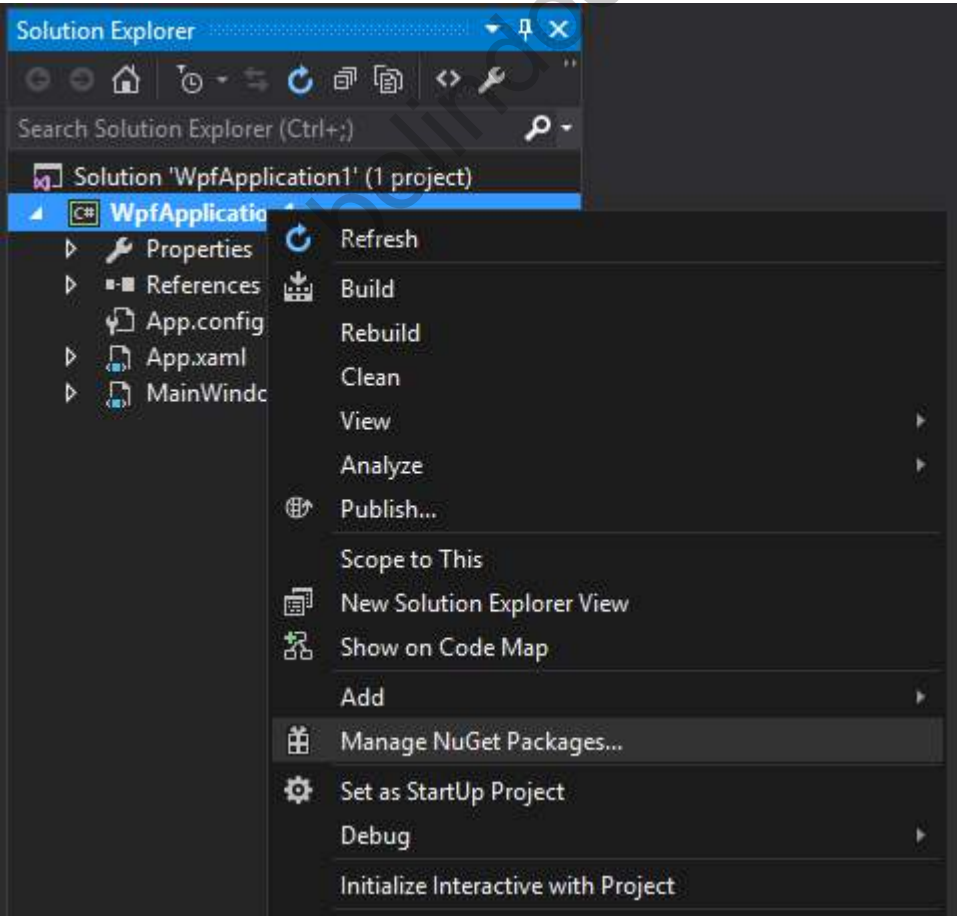
第1章：开始使用 Entity Framework

版本	发布日期
1.0	2008-08-11
4.0	2010-04-12
4.1	2011-04-12
4.1 更新 1	2011-07-25
4.3.1	2012-02-29
5.0	2012-08-11
6.0	2013-10-17
6.1	2014-03-17
核心 1.0	2016-06-27

发行说明：<https://msdn.microsoft.com/en-ca/data/jj574253.aspx>

第 1.1 节：安装实体框架 NuGet 包

在你的 Visual Studio 中打开解决方案资源管理器窗口，然后 右键点击 你的项目，然后选择菜单中的 *管理 NuGet 包*：



在打开的窗口右上角的搜索框中输入EntityFramework。

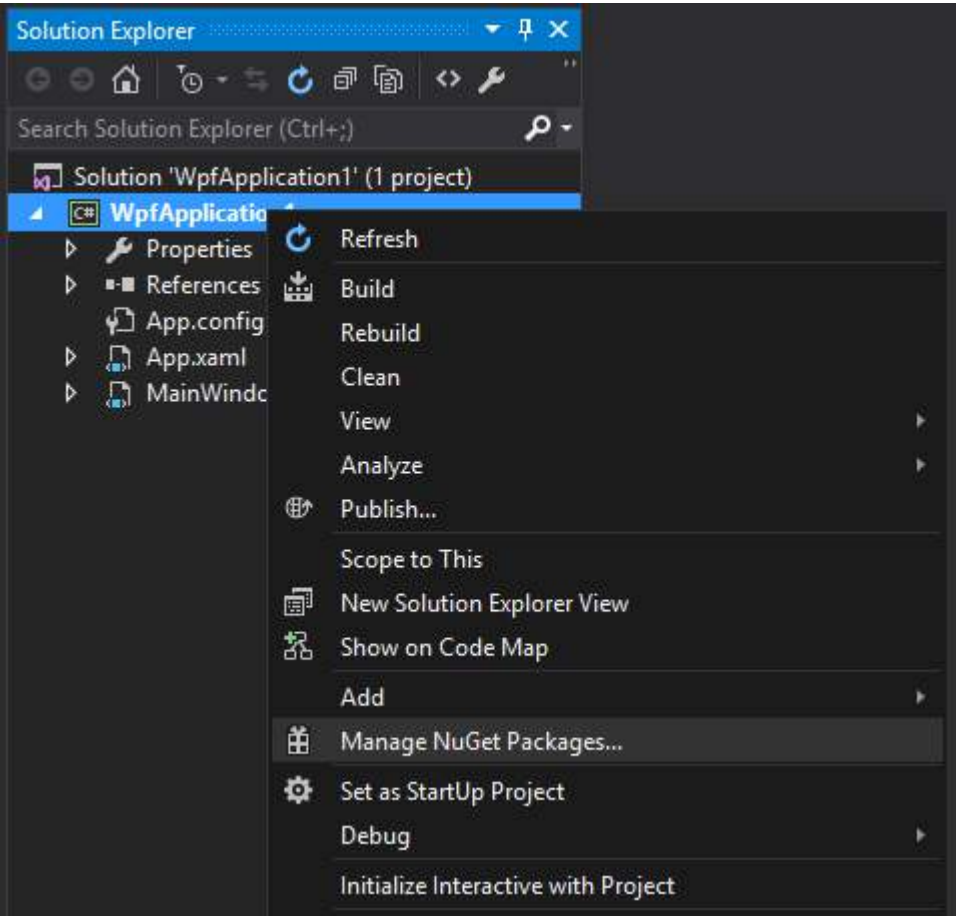
Chapter 1: Getting started with Entity Framework

Version	Release Date
1.0	2008-08-11
4.0	2010-04-12
4.1	2011-04-12
4.1 Update 1	2011-07-25
4.3.1	2012-02-29
5.0	2012-08-11
6.0	2013-10-17
6.1	2014-03-17
Core 1.0	2016-06-27

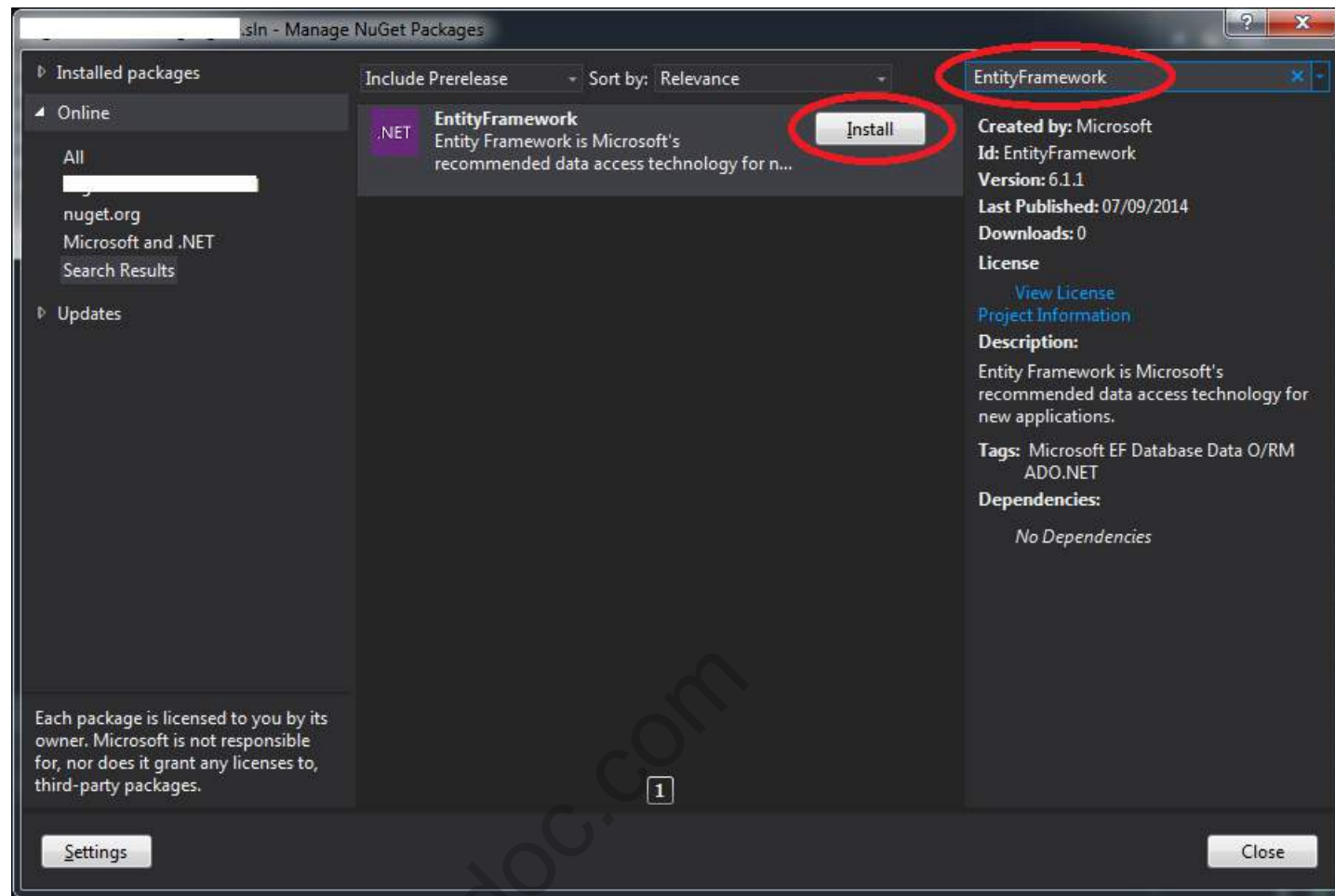
Release Notes: <https://msdn.microsoft.com/en-ca/data/jj574253.aspx>

Section 1.1: Installing the Entity Framework NuGet Package

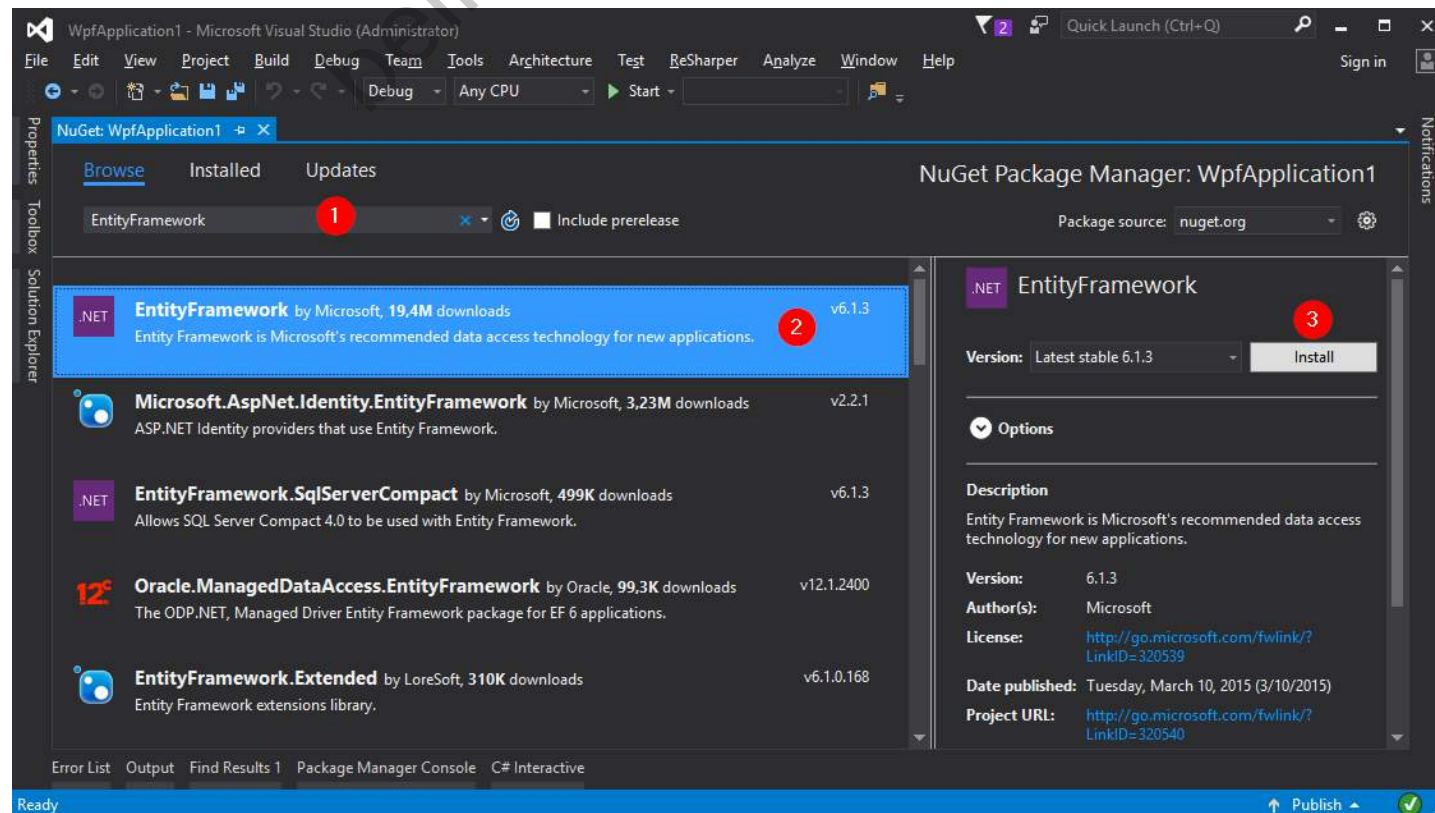
In your Visual Studio open the **Solution Explorer** window then right click on your project then choose *Manage NuGet Packages* from the menu:



In the window that opens type EntityFramework in the search box in the top right.

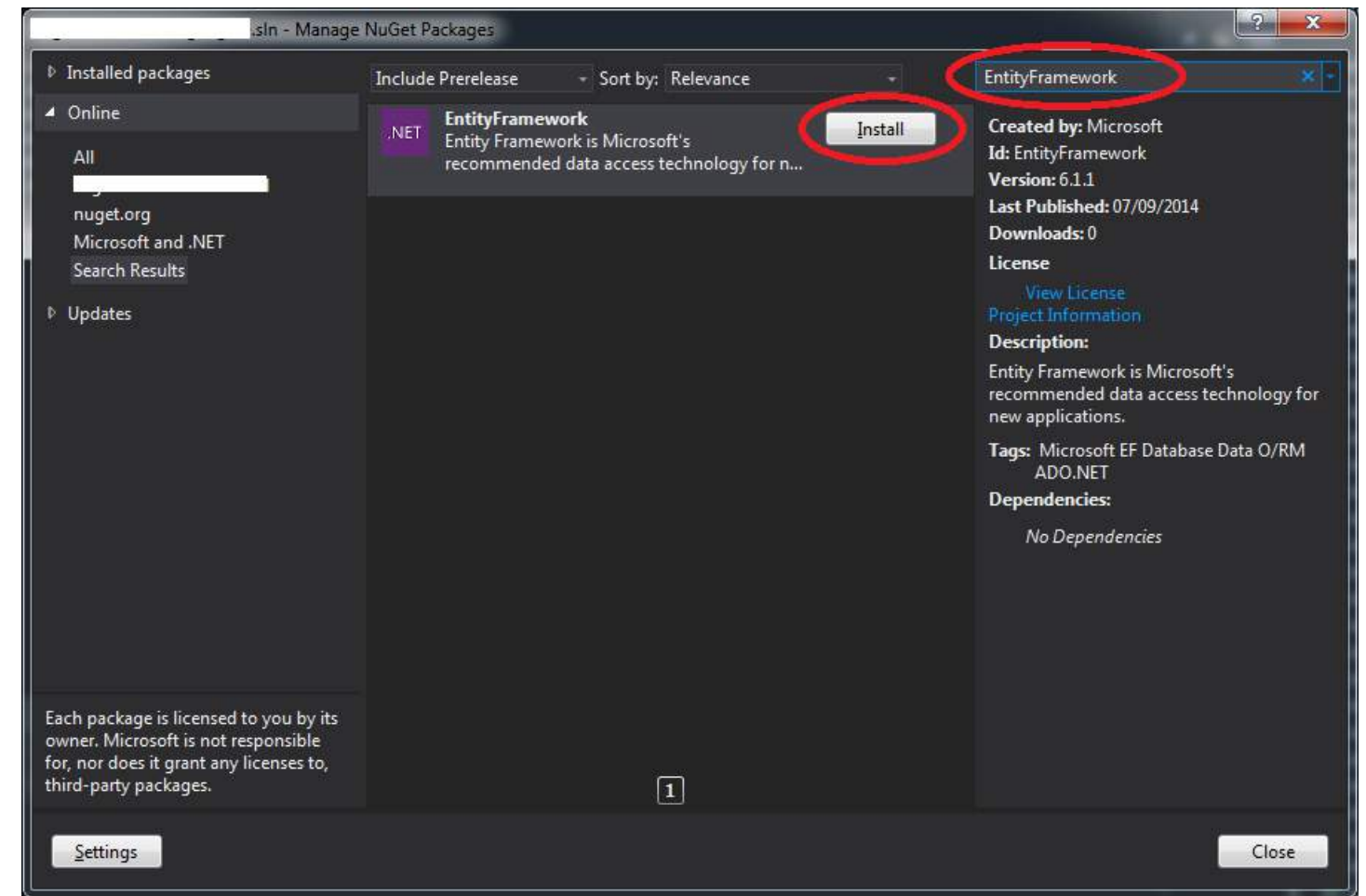


或者如果你使用的是 Visual Studio 2015, 你会看到类似这样的界面：

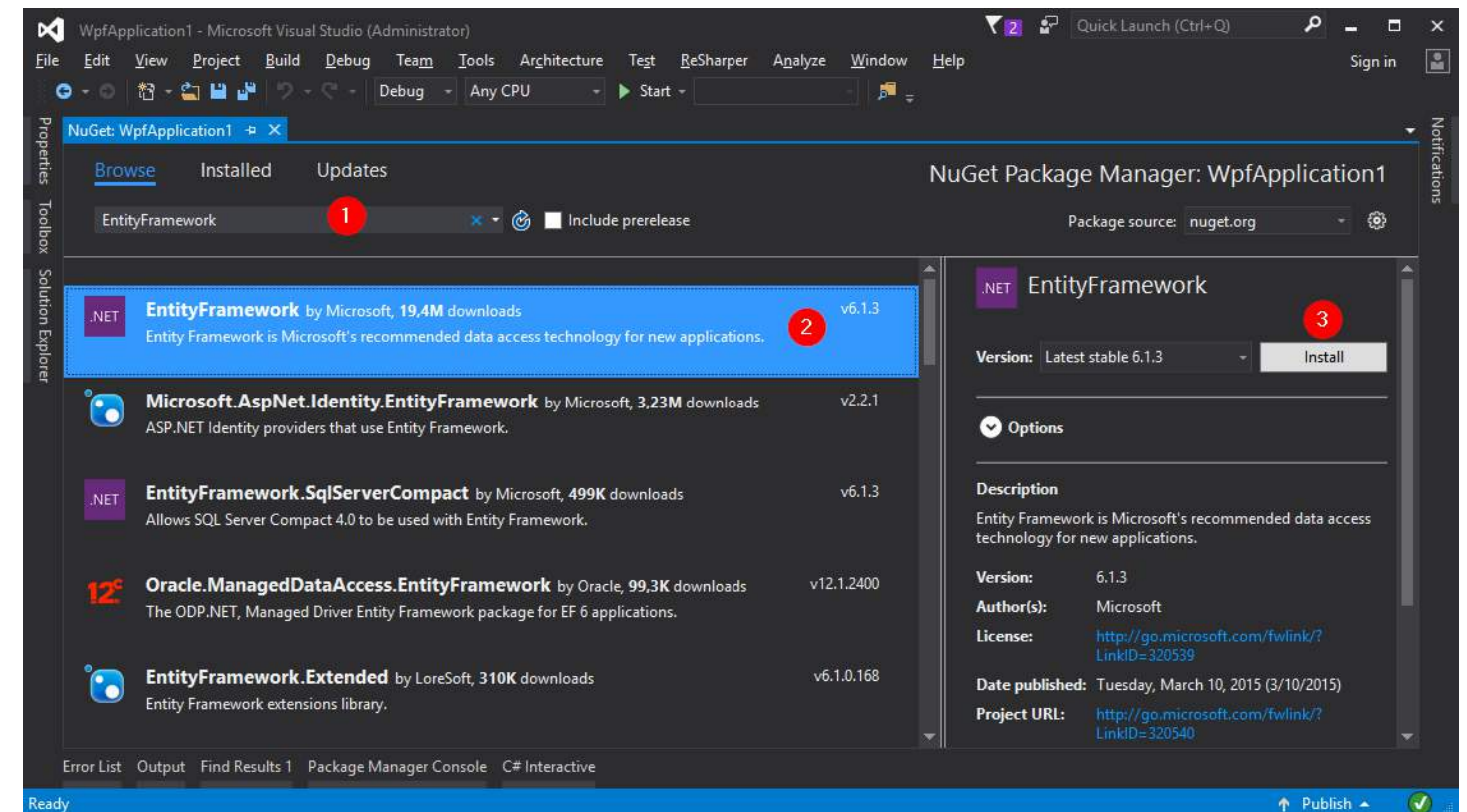


然后点击安装。

我们也可以使用包管理器控制台安装实体框架。为此，你首先需要通过
工具菜单 -> NuGet 包管理器 -> 包管理器控制台打开它，然后输入以下命令：



Or if you are using Visual Studio 2015 you'll see something like this:



Then click Install.

We can also install entity framework using the package manager console. To do you have first to open it using the
Tools menu -> NuGet Package Manager -> Package Manager Console then enter this:

```
Package Manager Console
Package source: nuget.org Default project: WpfApplication1
Type 'get-help NuGet' to see all available NuGet commands.

PM> Install-Package EntityFramework

Attempting to gather dependency information for package 'EntityFramework.6.1.3' with respect to project 'WpfApplication1', targeting '.NETFramework,Version=v4.6'
Gathering dependency information took 580,37 ms
Attempting to resolve dependencies for package 'EntityFramework.6.1.3' with DependencyBehavior 'Lowest'
Resolving dependency information took 0 ms
Resolving actions to install package 'EntityFramework.6.1.3'
Resolved actions to install package 'EntityFramework.6.1.3'
Retrieving package 'EntityFramework 6.1.3' from 'nuget.org'.
Adding package 'EntityFramework.6.1.3' to folder 'c:\dev\so\WpfApplication1\packages'
Added package 'EntityFramework.6.1.3' to folder 'c:\dev\so\WpfApplication1\packages'
Added package 'EntityFramework.6.1.3' to 'packages.config'
Executing script file 'c:\dev\so\WpfApplication1\packages\EntityFramework.6.1.3\tools\init.ps1'
Executing script file 'c:\dev\so\WpfApplication1\packages\EntityFramework.6.1.3\tools\install.ps1'

Type 'get-help EntityFramework' to see all available Entity Framework commands.
Successfully installed 'EntityFramework 6.1.3' to WpfApplication1
Executing nuget actions took 7,94 sec
Time Elapsed: 00:00:09.3130546
PM>
```

这将安装实体框架并自动在您的项目中添加对程序集的引用。

第1.2节：从C#使用实体框架（代码优先）

代码优先允许您在不使用GUI设计器或.edmx文件的情况下创建实体（类）。它被称为代码优先，因为您可以先创建模型first，实体框架将根据映射自动为您创建数据库。或者，您也可以使用这种方法配合现有数据库，这称为代码优先使用现有数据库。例如，如果您想要一个表来保存行星列表：

```
public class 行星
{
    public string 名称 { get; set; }
    public decimal 平均距离太阳 { get; set; }
}
```

现在创建您的上下文，它是实体类和数据库之间的桥梁。给它一个或多个DbSet<> 属性：

```
using System.Data.Entity;

public class 行星上下文 : DbContext
{
    public DbSet<行星> 行星集合 { get; set; }
}
```

我们可以通过以下方式使用它：

```
using(var context = new PlanetContext())
{
    var jupiter = new Planet
    {
```

```
Package Manager Console
Package source: nuget.org Default project: WpfApplication1
Type 'get-help NuGet' to see all available NuGet commands.

PM> Install-Package EntityFramework

Attempting to gather dependency information for package 'EntityFramework.6.1.3' with respect to project 'WpfApplication1', targeting '.NETFramework,Version=v4.6'
Gathering dependency information took 580,37 ms
Attempting to resolve dependencies for package 'EntityFramework.6.1.3' with DependencyBehavior 'Lowest'
Resolving dependency information took 0 ms
Resolving actions to install package 'EntityFramework.6.1.3'
Resolved actions to install package 'EntityFramework.6.1.3'
Retrieving package 'EntityFramework 6.1.3' from 'nuget.org'.
Adding package 'EntityFramework.6.1.3' to folder 'c:\dev\so\WpfApplication1\packages'
Added package 'EntityFramework.6.1.3' to folder 'c:\dev\so\WpfApplication1\packages'
Added package 'EntityFramework.6.1.3' to 'packages.config'
Executing script file 'c:\dev\so\WpfApplication1\packages\EntityFramework.6.1.3\tools\init.ps1'
Executing script file 'c:\dev\so\WpfApplication1\packages\EntityFramework.6.1.3\tools\install.ps1'

Type 'get-help EntityFramework' to see all available Entity Framework commands.
Successfully installed 'EntityFramework 6.1.3' to WpfApplication1
Executing nuget actions took 7,94 sec
Time Elapsed: 00:00:09.3130546
PM>
```

This will install Entity Framework and automatically add a reference to the assembly in your project.

Section 1.2: Using Entity Framework from C# (Code First)

Code first allows you to create your entities (classes) without using a GUI designer or a .edmx file. It is named *Code first*, because you can create your models *first* and *Entity framework* will create database according to mappings for you automatically. Or you can also use this approach with existing database, which is called *code first with existing database* For example, if you want a table to hold a list of planets:

```
public class Planet
{
    public string Name { get; set; }
    public decimal AverageDistanceFromSun { get; set; }
}
```

Now create your context which is the bridge between your entity classes and the database. Give it one or more DbSet<> properties:

```
using System.Data.Entity;

public class PlanetContext : DbContext
{
    public DbSet<Planet> Planets { get; set; }
}
```

We can use this by doing the following:

```
using(var context = new PlanetContext())
{
    var jupiter = new Planet
    {
```



```
Name = "木星",
    AverageDistanceFromSun = 778.5
};

context.Planets.Add(jupiter);
context.SaveChanges();
}
```

在此示例中，我们创建了一个新的Planet对象，其Name属性值为"木星"，AverageDistanceFromSun属性值为778.5

然后，我们可以使用DbSet的Add()方法将该Planet添加到上下文中，并通过SaveChanges()方法将更改提交到数据库。

或者，我们可以从数据库中检索行：

```
using(var context = new PlanetContext())
{
    var jupiter = context.Planets.Single(p => p.Name == "木星");
    Console.WriteLine($"木星距离太阳有 {jupiter.AverageDistanceFromSun} 百万公里。");
}
```

第1.3节：什么是实体框架？

编写和管理用于数据访问的ADO.Net代码是一项繁琐且单调的工作。微软提供了一个名为“Entity Framework”的O/RM框架，用于自动化应用程序的数据库相关操作。

Entity Framework是一种对象/关系映射（O/RM）框架。它是对ADO.NET的增强，为开发者提供了一种自动化机制，用于访问和存储数据库中的数据。

什么是O/RM？

ORM是一种工具，用于将领域对象中的数据以自动化方式存储到关系型数据库（如MS SQL Server）中，几乎无需编程。O/RM包括三个主要部分：

- 1. 领域类对象
- 2. 关系型数据库对象
- 3. 关于领域对象如何映射到关系型数据库对象（例如表、视图和存储过程）的映射信息

ORM允许我们将数据库设计与领域类设计分开。这使得应用程序更易维护和扩展。它还自动化了标准的CRUD操作（创建、读取、更新和删除），开发者无需手动编写这些操作。

```
Name = "Jupiter",
    AverageDistanceFromSun = 778.5
};

context.Planets.Add(jupiter);
context.SaveChanges();
}
```

In this example we create a new Planet with the Name property with the value of "Jupiter" and the AverageDistanceFromSun property with the value of 778.5

We can then add this Planet to the context by using the DbSet's Add() method and commit our changes to the database by using the SaveChanges() method.

Or we can retrieve rows from the database:

```
using(var context = new PlanetContext())
{
    var jupiter = context.Planets.Single(p => p.Name == "Jupiter");
    Console.WriteLine($"Jupiter is {jupiter.AverageDistanceFromSun} million km from the sun.");
}
```

Section 1.3: What is Entity Framework?

Writing and managing ADO.Net code for data access is a tedious and monotonous job. Microsoft has provided an O/RM framework called "Entity Framework" to automate database related activities for your application.

Entity framework is an Object/Relational Mapping (O/RM) framework. It is an enhancement to ADO.NET that gives developers an automated mechanism for accessing & storing the data in the database.

What is O/RM?

ORM is a tool for storing data from domain objects to the relational database like MS SQL Server, in an automated way, without much programming. O/RM includes three main parts:

- 1. Domain class objects
- 2. Relational database objects
- 3. Mapping information on how domain objects map to relational database objects(e.x tables, views & stored procedures)

ORM allows us to keep our database design separate from our domain class design. This makes the application maintainable and extendable. It also automates standard CRUD operation (Create, Read, Update & Delete) so that the developer doesn't need to write it manually.

第2章：代码优先约定

第2.1节：移除约定

您可以通过重写OnModelCreating方法，移除System.Data.Entity.ModelConfiguration.Conventions命名空间中定义的任何约定。

下面的示例移除了PluralizingTableNameConvention约定。

```
public class EshopContext : DbContext
{
    public DbSet<Product> Products { set; get; }
    ...

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    }
}
```

默认情况下，EF会创建以实体类名加“s”后缀的数据库表。在此示例中，代码优先配置为忽略PluralizingTableName约定，因此将创建 dbo.Product表，而不是 dbo.Products表。

第2.2节：主键约定

默认情况下，如果类中的属性名为“ID”（不区分大小写），或者属性名为类名后跟“ID”，则该属性为主键。如果主键属性的类型是数字或GUID，则会被配置为标识列。简单示例：

```
public class Room
{
    // 主键
    public int RoomId{ get; set; }
    ...
}
```

第2.3节：类型发现

默认情况下，Code First 会将以下内容包含在模型中

- 1. 在上下文类中定义为 DbSet 属性的类型。
- 2. 即使定义在不同的程序集，引用类型也会包含在实体类型中。
- 3. 派生类，即使只有基类被定义为 DbSet 属性。

下面是一个示例，我们只在上下文类中将Company作为DbSet<Company>添加：

```
public class Company
{
    public int Id { set; get; }
    public string Name { set; get; }
    public virtual ICollection<Department> Departments { set; get; }
}

public class Department
{
    public int Id { set; get; }
```

Chapter 2: Code First Conventions

Section 2.1: Removing Conventions

You can remove any of the conventions defined in the System.Data.Entity.ModelConfiguration.Conventions namespace, by overriding OnModelCreating method.

The following example removes PluralizingTableNameConvention.

```
public class EshopContext : DbContext
{
    public DbSet<Product> Products { set; get; }
    ...

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
    }
}
```

By default EF will create DB table with entity class name suffixed by 's'. In this example, Code First is configured to ignore PluralizingTableName convention so, instead of dbo.Products table dbo.Product table will be created.

Section 2.2: Primary Key Convention

By default a property is a primary key if a property on a class is named “ID” (not case sensitive), or the class name followed by "ID". If the type of the primary key property is numeric or GUID it will be configured as an identity column. Simple Example:

```
public class Room
{
    // Primary key
    public int RoomId{ get; set; }
    ...
}
```

Section 2.3: Type Discovery

By default Code First includes in model

- 1. Types defined as a DbSet property in context class.
- 2. Reference types included in entity types even if they are defined in different assembly.
- 3. Derived classes even if only the base class is defined as DbSet property

Here is an example, that we are only adding Company as DbSet<Company> in our context class:

```
public class Company
{
    public int Id { set; get; }
    public string Name { set; get; }
    public virtual ICollection<Department> Departments { set; get; }
}

public class Department
{
    public int Id { set; get; }
```

```
public string Name { set; get; }
public virtual ICollection<Person> Staff { set; get; }
}

[Table("Staff")]
public class Person
{
    public int Id { set; get; }
    public string Name { set; get; }
    public decimal Salary { set; get; }
}

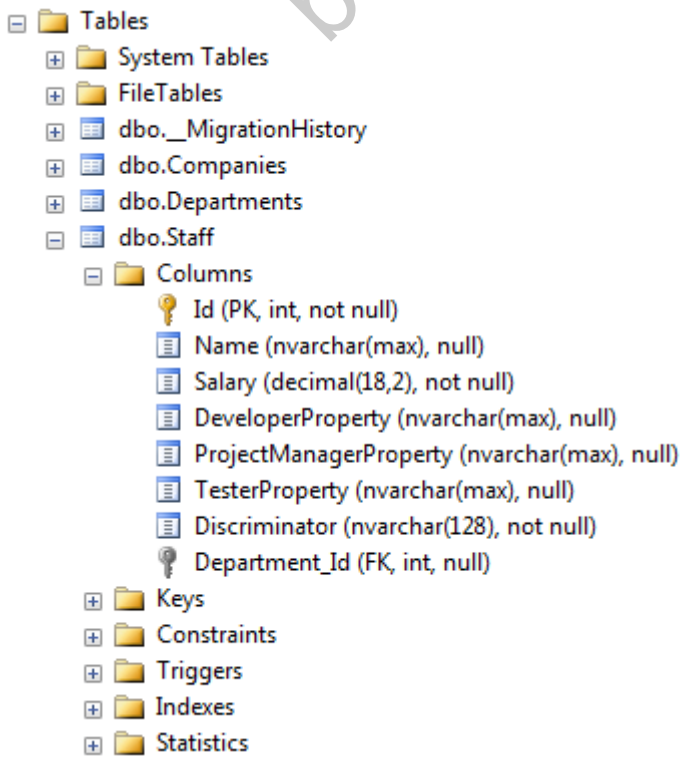
public class ProjectManager : Person
{
    public string ProjectManagerProperty { set; get; }
}

public class 开发者 : 人员
{
    public string 开发者属性 { set; get; }
}

public class 测试员 : 人员
{
    public string 测试员属性 { set; get; }
}

public class 应用数据库上下文 : DbContext
{
    public DbSet<公司> 公司集合 { set; get; }
}
```

我们可以看到所有类都包含在模型中



第2.4节：DecimalPropertyConvention（十进制属性约定）

默认情况下，Entity Framework 将十进制属性映射到数据库表中的 decimal(18,2) 列。

```
public string Name { set; get; }
public virtual ICollection<Person> Staff { set; get; }
}

[Table("Staff")]
public class Person
{
    public int Id { set; get; }
    public string Name { set; get; }
    public decimal Salary { set; get; }
}

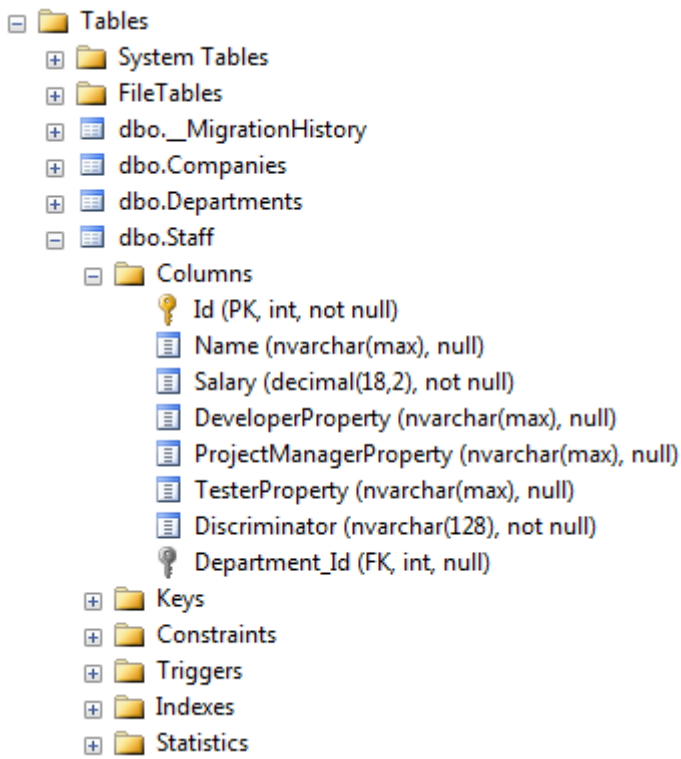
public class ProjectManager : Person
{
    public string ProjectManagerProperty { set; get; }
}

public class Developer : Person
{
    public string DeveloperProperty { set; get; }
}

public class Tester : Person
{
    public string TesterProperty { set; get; }
}

public class ApplicationDbContext : DbContext
{
    public DbSet<Company> Companies { set; get; }
}
```

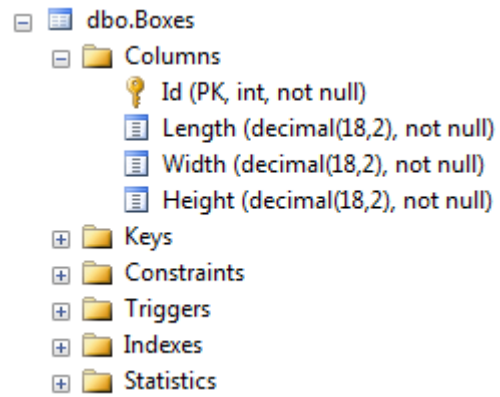
We can see that all the classes are included in model



Section 2.4: DecimalPropertyConvention

By default Entity Framework maps decimal properties to decimal(18,2) columns in database tables.

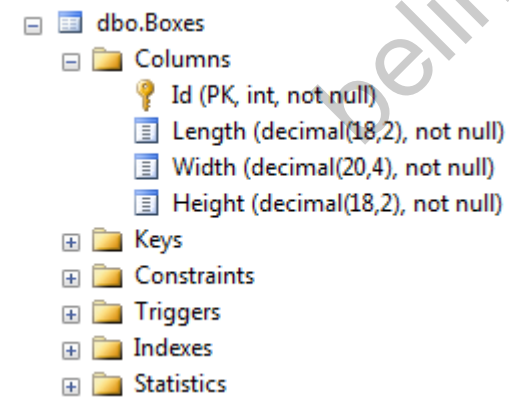
```
public class Box
{
    public int Id { set; get; }
    public decimal Length { set; get; }
    public decimal Width { set; get; }
    public decimal Height { set; get; }
}
```



我们可以更改 decimal 属性的精度：

1. 使用 Fluent API：

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Box>().Property(b => b.Width).HasPrecision(20, 4);
}
```

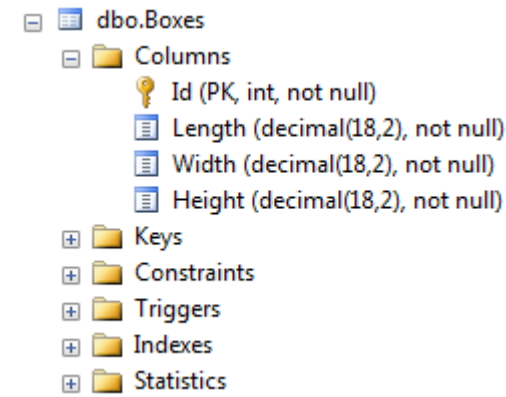


只有“Width”属性映射为 decimal(20, 4)。

2. 替换约定：

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Conventions.Remove<DecimalPropertyConvention>();
    modelBuilder.Conventions.Add(new DecimalPropertyConvention(10, 4));
}
```

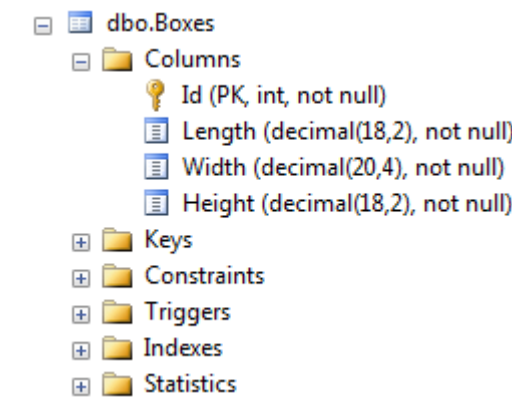
```
public class Box
{
    public int Id { set; get; }
    public decimal Length { set; get; }
    public decimal Width { set; get; }
    public decimal Height { set; get; }
}
```



We can change the precision of decimal properties:

1. Use Fluent API:

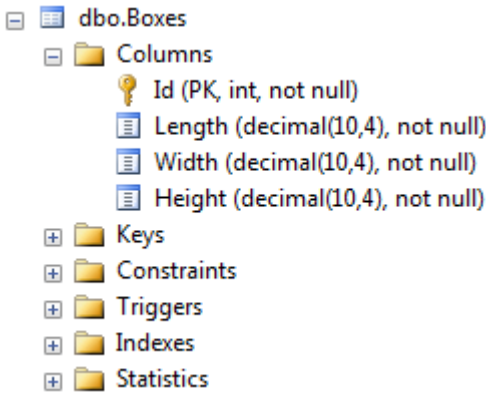
```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Box>().Property(b => b.Width).HasPrecision(20, 4);
}
```



Only "Width" Property is mapped to decimal(20, 4).

2. Replace the convention:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Conventions.Remove<DecimalPropertyConvention>();
    modelBuilder.Conventions.Add(new DecimalPropertyConvention(10, 4));
}
```

每个 decimal 属性都映射为 decimal(10,4) 列。

第2.5节：关系约定

代码优先通过导航属性推断两个实体之间的关系。该导航属性可以是简单的引用类型或集合类型。例如，我们在学生类中定义了标准导航属性，在标准类中定义了 ICollection 导航属性。因此，代码优先通过在学生表中插入 Standard_StandardId 外键列，自动在标准和学生数据库表之间创建了一对多关系。

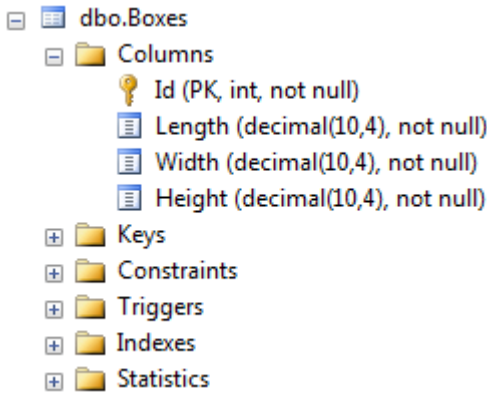
```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime DateOfBirth { get; set; }

    //导航属性
    public Standard Standard { get; set; }
}

public class Standard
{
    public int StandardId { get; set; }
    public string StandardName { get; set; }

    //集合导航属性
    public IList<Student> Students { get; set; }
}
```

上述实体使用 Standard_StandardId 外键创建了以下关系。



Every decimal property is mapped to decimal(10,4) columns.

Section 2.5: Relationship Convention

Code First infer the relationship between the two entities using navigation property. This navigation property can be a simple reference type or collection type. For example, we defined Standard navigation property in Student class and ICollection navigation property in Standard class. So, Code First automatically created one-to-many relationship between Standards and Students DB table by inserting Standard_StandardId foreign key column in the Students table.

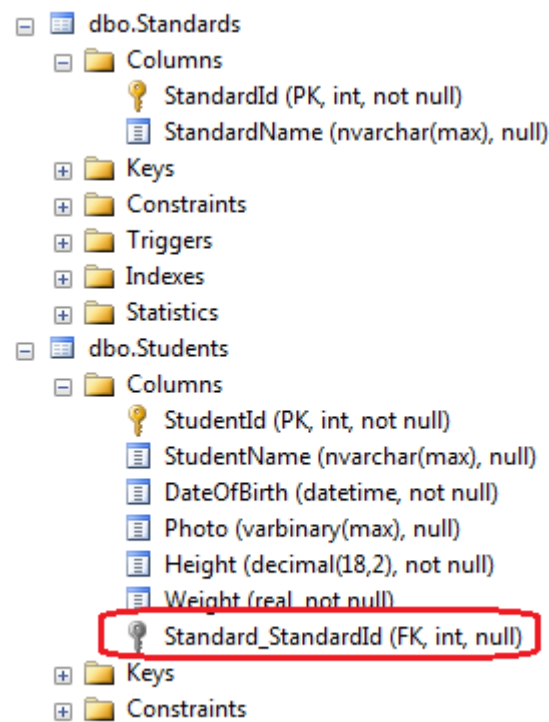
```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public DateTime DateOfBirth { get; set; }

    //Navigation property
    public Standard Standard { get; set; }
}

public class Standard
{
    public int StandardId { get; set; }
    public string StandardName { get; set; }

    //Collection navigation property
    public IList<Student> Students { get; set; }
}
```

The above entities created the following relationship using Standard_StandardId foreign key.



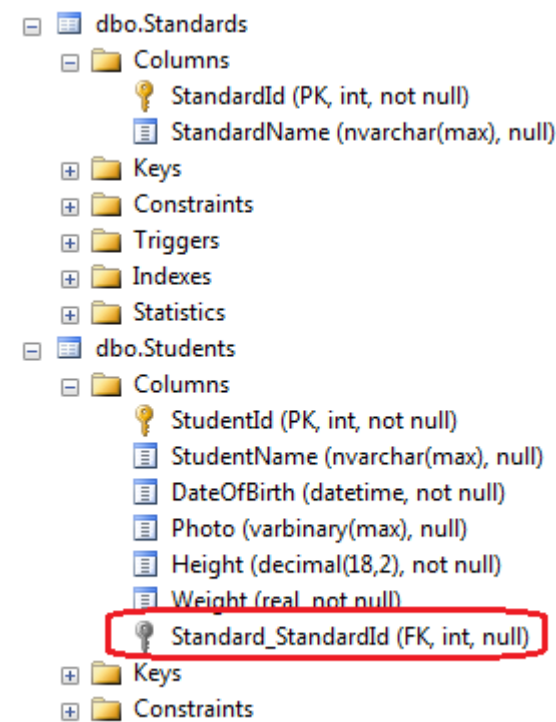
第2.6节：外键约定

如果类A与类B存在关联关系，并且类B具有与A的主键同名且同类型的属性，则EF会自动假定该属性是外键。

```
public class Department
{
    public int DepartmentId { set; get; }
    public string Name { set; get; }
    public virtual ICollection<Person> Staff { set; get; }
}

public class Person
{
    public int Id { set; get; }
    public string Name { set; get; }
    public decimal Salary { set; get; }
    public int DepartmentId { set; get; }
    public virtual Department Department { set; get; }
}
```

在这种情况下，DepartmentId 是未显式指定的外键。



Section 2.6: Foreign Key Convention

If class A is in relationship with class B and class B has property with the same name and type as the primary key of A, then EF automatically assumes that property is foreign key.

```
public class Department
{
    public int DepartmentId { set; get; }
    public string Name { set; get; }
    public virtual ICollection<Person> Staff { set; get; }
}

public class Person
{
    public int Id { set; get; }
    public string Name { set; get; }
    public decimal Salary { set; get; }
    public int DepartmentId { set; get; }
    public virtual Department Department { set; get; }
}
```

In this case DepartmentId is foreign key without explicit specification.

第3章：Code First 数据注解

第3.1节：[Column] 属性

```
public class Person
{
    public int PersonID { get; set; }

    [Column("NameOfPerson")]
    public string PersonName { get; set; }
}
```

告诉实体框架使用特定的列名，而不是使用属性名。你还可以指定数据库的数据类型和表中列的顺序：

```
[Column("NameOfPerson", 类型名称 = "varchar", 顺序 = 1)]
public string PersonName { get; set; }
```

第3.2节：[DatabaseGenerated] 属性

指定数据库如何为属性生成值。有三种可能的值：

- 1. None 指定值不是由数据库生成的。
- 2. Identity 指定该列是一个标识列，通常用于整数主键。
- 3. Computed 指定数据库为该列生成值。

如果值不是None，Entity Framework 将不会将该属性所做的更改提交回数据库。

默认情况下（基于StoreGeneratedIdentityKeyConvention），整数键属性将被视为标识列。要覆盖此约定并强制将其视为非标识列，可以使用DatabaseGenerated属性并将值设置为None。

```
using System.ComponentModel.DataAnnotations.Schema;

public class Foo
{
    [Key]
    public int Id { get; set; } // 标识列（自动递增）
}

public class Bar
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int Id { get; set; } // 非标识列
}
```

以下 SQL 创建了一个带有计算列的表：

```
CREATE TABLE [Person] (
    Name varchar(100) PRIMARY KEY,
    DateOfBirth Date NOT NULL,
    Age AS DATEDIFF(year, DateOfBirth, GETDATE())
)
```

Chapter 3: Code First DataAnnotations

Section 3.1: [Column] attribute

```
public class Person
{
    public int PersonID { get; set; }

    [Column("NameOfPerson")]
    public string PersonName { get; set; }
}
```

Tells Entity Framework to use a specific column name instead using the name of the property. You can also specify the database data type and the order of the column in table:

```
[Column("NameOfPerson", TypeName = "varchar", Order = 1)]
public string PersonName { get; set; }
```

Section 3.2: [DatabaseGenerated] attribute

Specifies how the database generates values for the property. There are three possible values:

- 1. None specifies that the values are not generated by the database.
- 2. Identity specifies that the column is an identity column, which is typically used for integer primary keys.
- 3. Computed specifies that the database generates the value for the column.

If the value is anything other than None, Entity Framework will not commit changes made to the property back to the database.

By default (based on the StoreGeneratedIdentityKeyConvention) an integer key property will be treated as an identity column. To override this convention and force it to be treated as a non-identity column you can use the DatabaseGenerated attribute with a value of None.

```
using System.ComponentModel.DataAnnotations.Schema;

public class Foo
{
    [Key]
    public int Id { get; set; } // identity (auto-increment) column
}

public class Bar
{
    [Key]
    [DatabaseGenerated(DatabaseGeneratedOption.None)]
    public int Id { get; set; } // non-identity column
}
```

The following SQL creates a table with a computed column:

```
CREATE TABLE [Person] (
    Name varchar(100) PRIMARY KEY,
    DateOfBirth Date NOT NULL,
    Age AS DATEDIFF(year, DateOfBirth, GETDATE())
)
```

要为上述表中的记录创建实体，您需要使用值为Computed的DatabaseGenerated属性。

```
[Table("Person")]
public class Person
{
    [Key, StringLength(100)]
    public string Name { get; set; }
    public DateTime DateOfBirth { get; set; }
    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
    public int Age { get; set; }
}
```

第3.3节：[Required] 属性

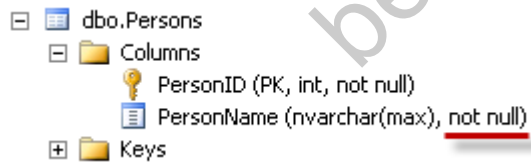
当应用于领域类的属性时，数据库将创建一个 NOT NULL 列。

```
using System.ComponentModel.DataAnnotations;

public class Person
{
    public int PersonID { get; set; }

    [Required]
    public string PersonName { get; set; }
}
```

生成的带有 NOT NULL 约束的列：



注意：它也可以作为验证属性用于 asp.net-mvc。

第3.4节：[MaxLength] 和 [MinLength] 属性

[MaxLength(int)] 属性可以应用于领域类的字符串或数组类型属性。实体框架将把列的大小设置为指定的值。

```
using System.ComponentModel.DataAnnotations;

public class Person
{
    public int PersonID { get; set; }

    [最小长度(3), 最大长度(100)]
    public string PersonName { get; set; }
}
```

指定列长度后生成的列：

To create an entity for representing the records in the above table, you would need to use the DatabaseGenerated attribute with a value of Computed.

```
[Table("Person")]
public class Person
{
    [Key, StringLength(100)]
    public string Name { get; set; }
    public DateTime DateOfBirth { get; set; }
    [DatabaseGenerated(DatabaseGeneratedOption.Computed)]
    public int Age { get; set; }
}
```

Section 3.3: [Required] attribute

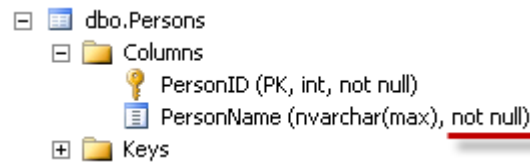
When applied to a property of a domain class, the database will create a NOT NULL column.

```
using System.ComponentModel.DataAnnotations;

public class Person
{
    public int PersonID { get; set; }

    [Required]
    public string PersonName { get; set; }
}
```

The resulting column with the NOT NULL constraint:



Note: It can also be used with asp.net-mvc as a validation attribute.

Section 3.4: [MaxLength] and [MinLength] attributes

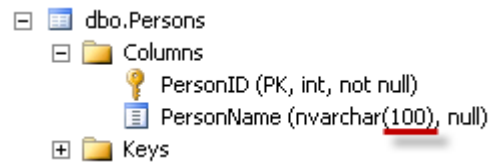
[MaxLength(int)] attribute can be applied to a string or array type property of a domain class. Entity Framework will set the size of a column to the specified value.

```
using System.ComponentModel.DataAnnotations;

public class Person
{
    public int PersonID { get; set; }

    [MinLength(3), MaxLength(100)]
    public string PersonName { get; set; }
}
```

The resulting column with the specified column length:



[MinLength(int)] 属性是一个验证属性，它不会影响数据库结构。如果我们尝试插入/更新一个 PersonName 长度少于 3 个字符的 Person，该提交将失败。我们'll 会收到一个需要处理的 DbUpdateConcurrencyException。

```
using (var db = new ApplicationDbContext())
{
    db.Staff.Add(new Person() { PersonName = "ng" });
    try
    {
        db.SaveChanges();
    }
    catch (DbEntityValidationException ex)
    {
        //错误信息 = "字段 PersonName 必须是字符串或数组类型，且最小长度为 '3'。"
    }
}
```

Both [MaxLength] and [MinLength] 属性也可以作为验证属性与 asp.net-mvc 一起使用。

第3.5节：[InverseProperty(string)] 属性

```
using System.ComponentModel.DataAnnotations.Schema;

public class 部门
{
    ...

    public virtual ICollection<员工> 主要员工 { get; set; }
    public virtual ICollection<员工> 次要员工 { get; set; }
}

public class 员工
{
    ...

    [InverseProperty("主要员工")]
    public virtual 部门 主要部门 { get; set; }

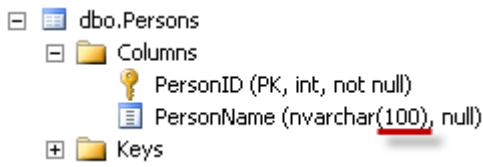
    [InverseProperty("次要员工")]
    public virtual 部门 次要部门 { get; set; }
}
```

当两个实体之间存在多个双向关系时，InverseProperty 可用于识别双向关系。

它告诉实体框架应将哪些导航属性与另一方的属性匹配。

当两个实体之间存在多个双向关系时，实体框架不知道哪个导航属性映射到另一侧的哪些属性。

它需要相关类中对应导航属性的名称作为参数。



[MinLength(int)] attribute is a validation attribute, it does not affect the database structure. If we try to insert/update a Person with PersonName with length less than 3 characters, this commit will fail. We'll get a DbUpdateConcurrencyException that we'll need to handle.

```
using (var db = new ApplicationDbContext())
{
    db.Staff.Add(new Person() { PersonName = "ng" });
    try
    {
        db.SaveChanges();
    }
    catch (DbEntityValidationException ex)
    {
        //ErrorMessage = "The field PersonName must be a string or array type with a minimum length of '3'."
    }
}
```

Both **[MaxLength]** and **[MinLength]** attributes can also be used with asp.net-mvc as a validation attribute.

Section 3.5: [InverseProperty(string)] attribute

```
using System.ComponentModel.DataAnnotations.Schema;

public class Department
{
    ...

    public virtual ICollection<Employee> PrimaryEmployees { get; set; }
    public virtual ICollection<Employee> SecondaryEmployees { get; set; }
}

public class Employee
{
    ...

    [InverseProperty("PrimaryEmployees")]
    public virtual Department PrimaryDepartment { get; set; }

    [InverseProperty("SecondaryEmployees")]
    public virtual Department SecondaryDepartment { get; set; }
}
```

InverseProperty can be used to identify *two way* relationships when **multiple** *two way* relationships exist between two entities.

It tells Entity Framework which navigation properties it should match with properties on the other side.

Entity Framework doesn't know which navigation property map with which properties on the other side when multiple bidirectional relationships exist between two entities.

It needs the name of the corresponding navigation property in the related class as its parameter.

这也可以用于与同一类型的其他实体有关系的实体，形成递归关系。

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

public class TreeNode
{
    [Key]
    public int ID { get; set; }
    public int ParentID { get; set; }

    ...

    [ForeignKey("ParentID")]
    public TreeNode ParentNode { get; set; }
    [InverseProperty("ParentNode")]
    public virtual ICollection<TreeNode> ChildNodes { get; set; }
}
```

还要注意使用ForeignKey属性来指定表中用作外键的列。在第一个示例中，Employee类上的两个属性可以应用ForeignKey属性来定义列名。

第3.6节：[ComplexType]属性

```
using System.ComponentModel.DataAnnotations.Schema;

[ComplexType]
public class BlogDetails
{
    public DateTime? DateCreated { get; set; }

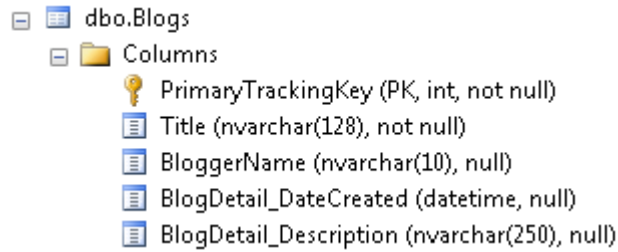
    [MaxLength(250)]
    public string Description { get; set; }
}

public class Blog
{
    ...

    public BlogDetails BlogDetail { get; set; }
}
```

将类标记为实体框架中的复杂类型。

复杂类型（或领域驱动设计中的值对象）不能单独被跟踪，但它们作为实体的一部分被跟踪。这就是为什么示例中的 BlogDetails 没有主键属性的原因。



当描述跨多个类的领域实体并将这些类分层为

This can also be used for entities that have a relationship to other entities of the same type, forming a recursive relationship.

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

public class TreeNode
{
    [Key]
    public int ID { get; set; }
    public int ParentID { get; set; }

    ...

    [ForeignKey("ParentID")]
    public TreeNode ParentNode { get; set; }
    [InverseProperty("ParentNode")]
    public virtual ICollection<TreeNode> ChildNodes { get; set; }
}
```

Note also the use of the ForeignKey attribute to specify the column that is used for the foreign key on the table. In the first example, the two properties on the Employee class could have had the ForeignKey attribute applied to define the column names.

Section 3.6: [ComplexType] attribute

```
using System.ComponentModel.DataAnnotations.Schema;

[ComplexType]
public class BlogDetails
{
    public DateTime? DateCreated { get; set; }

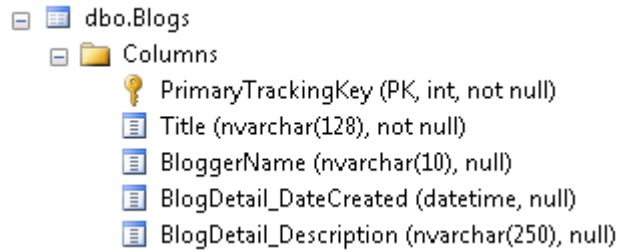
    [MaxLength(250)]
    public string Description { get; set; }
}

public class Blog
{
    ...

    public BlogDetails BlogDetail { get; set; }
}
```

Mark the class as complex type in Entity Framework.

Complex Types (Or *Value Objects* In Domain Driven Design) cannot be tracked on their own but they are tracked as part of an entity. This is why BlogDetails in the example does not have a key property.



They can be useful when describing domain entities across multiple classes and layering those classes into a

第3.7节：[ForeignKey(string)] 属性

如果需要自定义外键名称且不遵循实体框架的约定，则指定自定义外键名称。

```
public class Person
{
    public int IdAddress { get; set; }

    [ForeignKey(nameof(IdAddress))]
    public virtual Address HomeAddress { get; set; }
}
```

当你对同一实体类型有多个关系时，也可以使用这种方式。

```
using System.ComponentModel.DataAnnotations.Schema;

public class Customer
{
    ...

    public int MailingAddressID { get; set; }
    public int BillingAddressID { get; set; }

    [ForeignKey("MailingAddressID")]
    public virtual Address MailingAddress { get; set; }
    [ForeignKey("BillingAddressID")]
    public virtual Address BillingAddress { get; set; }
}
```

如果没有ForeignKey属性，EF 可能会混淆它们，在获取MailingAddress时使用BillingAddressID的值，或者它可能会根据自己的命名规则（比如Address_MailingAddress_Id）为列生成不同的名称并尝试使用它（如果你在使用现有数据库，这将导致错误）。

第3.8节：[Range(min,max)] 属性

为属性指定数值的最小值和最大值范围

```
using System.ComponentModel.DataAnnotations;

public partial class Enrollment
{
    public int EnrollmentID { get; set; }

    [范围(0, 4)]
    public Nullable<decimal> Grade { get; set; }
}
```

如果我们尝试插入/更新一个超出范围的 Grade 值，该提交将失败。我们'll 得到一个需要处理的 DbUpdateConcurrencyException。

```
using (var db = new ApplicationDbContext())
{
    db.Enrollments.Add(new Enrollment() { Grade = 1000 });
}
```

complete entity.

Section 3.7: [ForeignKey(string)] attribute

Specifies custom foreign key name if a foreign key not following EF's convention is desired.

```
public class Person
{
    public int IdAddress { get; set; }

    [ForeignKey(nameof(IdAddress))]
    public virtual Address HomeAddress { get; set; }
}
```

This can also be used when you have multiple relationships to the same entity type.

```
using System.ComponentModel.DataAnnotations.Schema;

public class Customer
{
    ...

    public int MailingAddressID { get; set; }
    public int BillingAddressID { get; set; }

    [ForeignKey("MailingAddressID")]
    public virtual Address MailingAddress { get; set; }
    [ForeignKey("BillingAddressID")]
    public virtual Address BillingAddress { get; set; }
}
```

Without the ForeignKey attributes, EF might get them mixed up and use the value of BillingAddressID when fetching the MailingAddress, or it might just come up with a different name for the column based on its own naming conventions (like Address_MailingAddress_Id) and try to use that instead (which would result in an error if you're using this with an existing database).

Section 3.8: [Range(min,max)] attribute

Specifies a numeric minimum and maximum range for a property

```
using System.ComponentModel.DataAnnotations;

public partial class Enrollment
{
    public int EnrollmentID { get; set; }

    [Range(0, 4)]
    public Nullable<decimal> Grade { get; set; }
}
```

If we try to insert/update a Grade with value out of range, this commit will fail. We'll get a DbUpdateConcurrencyException that we'll need to handle.

```
using (var db = new ApplicationDbContext())
{
    db.Enrollments.Add(new Enrollment() { Grade = 1000 });
}
```

```
try
{
db.SaveChanges();
}
catch (DbEntityValidationException ex)
{
    // 一个或多个实体的验证失败
}
```

它也可以作为验证属性与 asp.net-mvc 一起使用。

结果：

Grade

The field Grade must be between 0 and 4.

第3.9节：[NotMapped] 属性

根据代码优先（Code-First）约定，实体框架（Entity Framework）会为每个公共属性创建一个列，该属性必须是受支持的数据类型且同时具有 getter 和 setter。[NotMapped] 注解必须应用于任何我们不希望在数据库表中有对应列的属性。

一个我们可能不想存储在数据库中的属性示例是学生的全名（FullName），它是基于他们的名字和姓氏计算得出的。这个可以动态计算，无需存储在数据库中。

```
public string FullName => string.Format("{0} {1}", FirstName, LastName);
```

“FullName”属性只有 getter，没有 setter，因此默认情况下，实体框架不会为其创建列。

另一个我们可能不想存储在数据库中的属性示例是学生的“AverageGrade”（平均成绩）。我们不需要按需获取 AverageGrade；相反，我们可能在其他地方有一个例程来计算它。

```
[NotMapped]
public float AverageGrade { set; get; }
```

“AverageGrade”必须标记为[NotMapped]注解，否则实体框架会为其创建列。

```
using System.ComponentModel.DataAnnotations.Schema;

public class Student
{
    public int Id { set; get; }

    public string FirstName { set; get; }

    public string LastName { set; get; }

    public string FullName => string.Format("{0} {1}", FirstName, LastName);

    [NotMapped]
    public float AverageGrade { set; get; }
}
```

对于上述实体，我们将在 DbMigration.cs 文件中查看

```
try
{
    db.SaveChanges();
}
catch (DbEntityValidationException ex)
{
    // Validation failed for one or more entities
}
```

It can also be used with asp.net-mvc as a validation attribute.

Result:

Grade

The field Grade must be between 0 and 4.

Section 3.9: [NotMapped] attribute

By Code-First convention, Entity Framework creates a column for every public property that is of a supported data type and has both a getter and a setter. **[NotMapped]** annotation must be applied to any properties that we do **NOT** want a column in a database table for.

An example of a property that we might not want to store in the database is a student's full name based on their first and last name. That can be calculated on the fly and there is no need to store it in the database.

```
public string FullName => string.Format("{0} {1}", FirstName, LastName);
```

The "FullName" property has only a getter and no setter, so by default, Entity Framework will **NOT** create a column for it.

Another example of a property that we might not want to store in the database is a student's "AverageGrade". We do not want to get the AverageGrade on-demand; instead we might have a routine elsewhere that calculates it.

```
[NotMapped]
public float AverageGrade { set; get; }
```

The "AverageGrade" must be marked **[NotMapped]** annotation, else Entity Framework will create a column for it.

```
using System.ComponentModel.DataAnnotations.Schema;

public class Student
{
    public int Id { set; get; }

    public string FirstName { set; get; }

    public string LastName { set; get; }

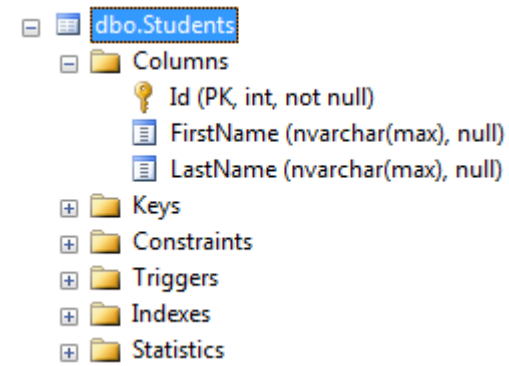
    public string FullName => string.Format("{0} {1}", FirstName, LastName);

    [NotMapped]
    public float AverageGrade { set; get; }
}
```

For the above Entity we will see inside DbMigration.cs


```
CreateTable(
    "dbo.Students",
    c => new
    {
        Id = c.Int(nullable: false, identity: true),
        FirstName = c.String(),
        LastName = c.String(),
    })
    .PrimaryKey(t => t.Id);
```

以及在 SQL Server Management Studio 中



第3.10节：[Table] 属性

```
[Table("People")]
public class Person
{
    public int PersonID { get; set; }
    public string PersonName { get; set; }
}
```

告诉实体框架使用特定的表名，而不是自动生成（例如Person或Persons）

我们也可以使用[Table]属性指定表的架构

```
[Table("People", Schema = "domain")]
```

第3.11节：[Index] 属性

```
public class Person
{
    public int PersonID { get; set; }
    public string PersonName { get; set; }

    [Index]
    public int Age { get; set; }
}
```

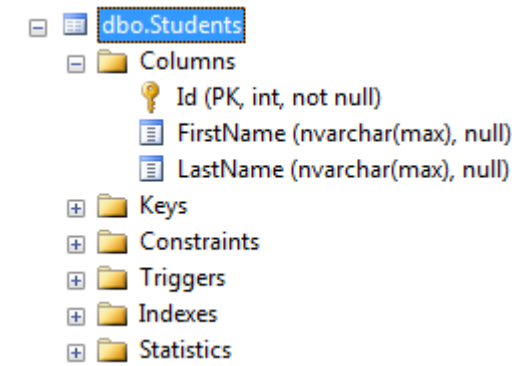
为一列或多列创建数据库索引。

```
[Index("IX_Person_Age")]
public int Age { get; set; }
```

这将创建一个具有特定名称的索引。

```
CreateTable(
    "dbo.Students",
    c => new
    {
        Id = c.Int(nullable: false, identity: true),
        FirstName = c.String(),
        LastName = c.String(),
    })
    .PrimaryKey(t => t.Id);
```

and in SQL Server Management Studio



Section 3.10: [Table] attribute

```
[Table("People")]
public class Person
{
    public int PersonID { get; set; }
    public string PersonName { get; set; }
}
```

Tells Entity Framework to use a specific table name instead of generating one (i.e. Person or Persons)

We can also specify a schema for the table using [Table] attribute

```
[Table("People", Schema = "domain")]
```

Section 3.11: [Index] attribute

```
public class Person
{
    public int PersonID { get; set; }
    public string PersonName { get; set; }

    [Index]
    public int Age { get; set; }
}
```

Creates a database index for a column or set of columns.

```
[Index("IX_Person_Age")]
public int Age { get; set; }
```

This creates an index with a specific name.

```
[Index(IsUnique = true)]
public int Age { get; set; }
```

这将创建一个唯一索引。

```
[Index("IX_Person_NameAndAge", 1)]
public int Age { get; set; }

[Index("IX_Person_NameAndAge", 2)]
public string PersonName { get; set; }
```

这将使用两列创建一个复合索引。为此，您必须指定相同的索引名称并提供列顺序。

注意: Index 属性是在 Entity Framework 6.1 中引入的。如果您使用的是更早版本，本节中的信息不适用。

第3.12节：[Key] 属性

Key 是数据库表中唯一标识每一行/记录的字段。

使用此属性来覆盖默认的 **Code-First 约定** 如果应用于属性，它将被用作该类的主键列。

```
using System.ComponentModel.DataAnnotations;

public class Person
{
    [Key]
    public int PersonKey { get; set; }      // <- 将被用作主键

    public string PersonName { get; set; }
}
```

如果需要复合主键，[Key] 属性也可以添加到多个属性上。复合键中列的顺序必须以 [Key, Column(Order = x)] 的形式提供。

```
using System.ComponentModel.DataAnnotations;

public class Person
{
    [Key, Column(Order = 0)]
    public int PersonKey1 { get; set; }    // <- 将作为主键的一部分使用

    [Key, Column(Order = 1)]
    public int PersonKey2 { get; set; }    // <- 将作为主键的一部分使用

    public string PersonName { get; set; }
}
```

如果没有 [Key] 属性，EntityFramework 将回退到默认约定，即使用类中名为 "Id" 或 "{ClassName}Id" 的属性作为主键。

```
public class Person
{
    public int PersonID { get; set; }      // <- 将作为主键使用
}
```

```
[Index(IsUnique = true)]
public int Age { get; set; }
```

This creates a unique index.

```
[Index("IX_Person_NameAndAge", 1)]
public int Age { get; set; }

[Index("IX_Person_NameAndAge", 2)]
public string PersonName { get; set; }
```

This creates a composite index using 2 columns. To do this you must specify the same index name and provide a column order.

Note: The Index attribute was introduced in Entity Framework 6.1. If you are using an earlier version the information in this section does not apply.

Section 3.12: [Key] attribute

Key is a field in a table which uniquely identifies each row/record in a database table.

Use this attribute to **override the default Code-First convention**. If applied to a property, it will be used as the **primary key column** for this class.

```
using System.ComponentModel.DataAnnotations;

public class Person
{
    [Key]
    public int PersonKey { get; set; }      // <- will be used as primary key

    public string PersonName { get; set; }
}
```

If a composite primary key is required, the [Key] attribute can also be added to multiple properties. The order of the columns within the composite key must be provided in the form **[Key, Column(Order = x)]**.

```
using System.ComponentModel.DataAnnotations;

public class Person
{
    [Key, Column(Order = 0)]
    public int PersonKey1 { get; set; }    // <- will be used as part of the primary key

    [Key, Column(Order = 1)]
    public int PersonKey2 { get; set; }    // <- will be used as part of the primary key

    public string PersonName { get; set; }
}
```

Without the [Key] attribute, EntityFramework will fall back to the default convention which is to use the property of the class as a primary key that is named "Id" or "{ClassName}Id".

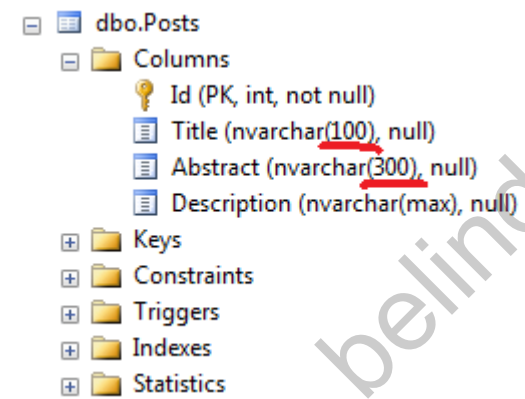
```
public class Person
{
    public int PersonID { get; set; }      // <- will be used as primary key
}
```

```
public string PersonName { get; set; }  
}
```

第3.13节：[StringLength(int)] 属性

```
using System.ComponentModel.DataAnnotations;  
  
public class Post  
{  
    public int Id { get; set; }  
  
    [StringLength(100)]  
    public string Title { get; set; }  
  
    [StringLength(300)]  
    public string Abstract { get; set; }  
  
    public string Description { get; set; }  
}
```

定义字符串字段的最大长度。



注意: 它也可以作为验证属性用于 asp.net-mvc。

第3.14节：[Timestamp] 属性

[TimeStamp] 属性只能应用于给定实体类中的一个字节数组属性。实体框架将在数据库表中为该属性创建一个非空的时间戳列。实体框架将自动使用此 TimeStamp 列进行并发检查。

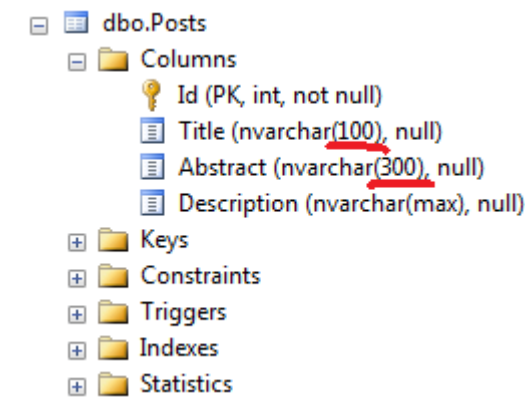
```
using System.ComponentModel.DataAnnotations.Schema;  
  
public class Student  
{  
    public int Id { set; get; }  
  
    public string FirstName { set; get; }  
  
    public string LastName { set; get; }  
  
    [Timestamp]  
    public byte[] RowVersion { get; set; }  
}
```

```
public string PersonName { get; set; }  
}
```

Section 3.13: [StringLength(int)] attribute

```
using System.ComponentModel.DataAnnotations;  
  
public class Post  
{  
    public int Id { get; set; }  
  
    [StringLength(100)]  
    public string Title { get; set; }  
  
    [StringLength(300)]  
    public string Abstract { get; set; }  
  
    public string Description { get; set; }  
}
```

Defines a maximum length for a string field.

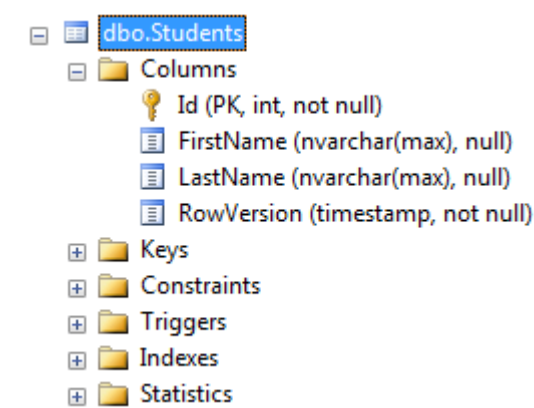


Note: It can also be used with asp.net-mvc as a validation attribute.

Section 3.14: [Timestamp] attribute

[TimeStamp] attribute can be applied to only one byte array property in a given Entity class. Entity Framework will create a non-nullable timestamp column in the database table for that property. Entity Framework will automatically use this TimeStamp column in concurrency check.

```
using System.ComponentModel.DataAnnotations.Schema;  
  
public class Student  
{  
    public int Id { set; get; }  
  
    public string FirstName { set; get; }  
  
    public string LastName { set; get; }  
  
    [Timestamp]  
    public byte[] RowVersion { get; set; }  
}
```



第3.15节：[ConcurrencyCheck] 属性

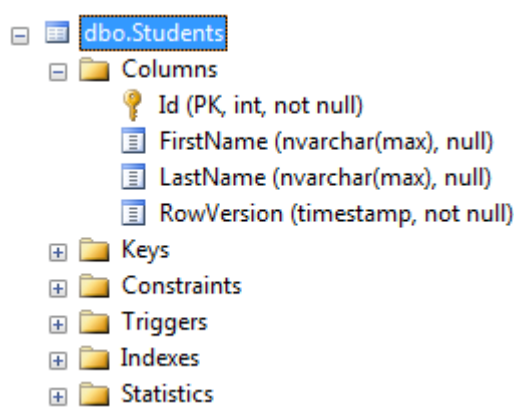
此属性应用于类属性。当您想使用现有列进行并发检查，而不是使用单独的时间戳列进行并发时，可以使用 ConcurrencyCheck 属性。

```
using System.ComponentModel.DataAnnotations;

public class 作者
{
    public int 作者Id { get; set; }

    [ConcurrencyCheck]
    public string 作者名 { get; set; }
}
```

从上面的例子来看，ConcurrencyCheck 特性被应用于 Author 类的 AuthorName 属性。因此，Code-First 会在更新命令的（where 子句中）包含 AuthorName 列，以检查乐观并发。



Section 3.15: [ConcurrencyCheck] Attribute

This attribute is applied to the class property. You can use ConcurrencyCheck attribute when you want to use existing columns for concurrency check and not a separate timestamp column for concurrency.

```
using System.ComponentModel.DataAnnotations;

public class Author
{
    public int AuthorId { get; set; }

    [ConcurrencyCheck]
    public string AuthorName { get; set; }
}
```

From above example, ConcurrencyCheck attribute is applied to AuthorName property of the Author class. So, Code-First will include AuthorName column in update command (where clause) to check for optimistic concurrency.

第4章：实体框架 Code First

第4.1节：连接到现有数据库

要实现实体框架中最简单的任务——连接到本地 MSSQL 实例上的现有数据库 ExampleDatabase，只需实现两个类。

第一个是实体类，它将映射到我们的数据库表 dbo.People。

```
class 人员
{
    public int PersonId { get; set; }
    public string FirstName { get; set; }
}
```

该类将使用 Entity Framework 的约定，并映射到表 dbo.People，该表预计具有主键 PersonId 和一个 varchar(max) 类型的属性 FirstName。

其次是上下文类，它继承自 System.Data.Entity.DbContext，负责在运行时管理实体对象，从数据库中填充它们，处理并发并将其保存回数据库。

```
class Context : DbContext
{
    public Context(string connectionString) : base(connectionString)
    {
        Database.SetInitializer<Context>(null);
    }

    public DbSet<Person> People { get; set; }
}
```

请注意，在上下文的构造函数中需要将数据库初始化器设置为 null——我们不想让 Entity Framework 创建数据库，我们只是想访问它。

现在你可以操作表中的数据，例如，从控制台应用程序中这样更改数据库中第一个人的 FirstName：

```
class Program
{
    static void Main(string[] args)
    {
        using (var ctx = new Context("DbConnectionString"))
        {
            var firstPerson = ctx.People.FirstOrDefault();
            if (firstPerson != null) {
                firstPerson.FirstName = "John";
                ctx.SaveChanges();
            }
        }
    }
}
```

在上面的代码中，我们使用参数 "DbConnectionString" 创建了 Context 的实例。这个参数必须要在我们的 app.config 文件中这样指定：

```
<connectionStrings>
```

Chapter 4: Entity Framework Code First

Section 4.1: Connect to an existing database

To achieve the simplest task in Entity Framework - to connect to an existing database ExampleDatabase on your local instance of MSSQL you have to implement two classes only.

First is the entity class, that will be mapped to our database table dbo.People.

```
class Person
{
    public int PersonId { get; set; }
    public string FirstName { get; set; }
}
```

The class will use Entity Framework's conventions and map to table dbo.People which is expected to have primary key PersonId and a varchar(max) property FirstName.

Second is the context class which derives from System.Data.Entity.DbContext and which will manage the entity objects during runtime, pupulate them from database, handle concurrency and save them back to the database.

```
class Context : DbContext
{
    public Context(string connectionString) : base(connectionString)
    {
        Database.SetInitializer<Context>(null);
    }

    public DbSet<Person> People { get; set; }
}
```

Please mind, that in the constructor of our context we need to set database initializer to null - we don't want Entity Framework to create the database, we just want to access it.

Now you are able manipulate data from that table, e.g. change the FirstName of first person in the database from a console application like this:

```
class Program
{
    static void Main(string[] args)
    {
        using (var ctx = new Context("DbConnectionString"))
        {
            var firstPerson = ctx.People.FirstOrDefault();
            if (firstPerson != null) {
                firstPerson.FirstName = "John";
                ctx.SaveChanges();
            }
        }
    }
}
```

In the code above we created instance of Context with an argument "DbConnectionString". This has to be specified in our app.config file like this:

```
<connectionStrings>
```

```
<add name="DbConnectionString"
connectionString="Data Source=.;Initial Catalog=ExampleDatabase;Integrated Security=True"
providerName="System.Data.SqlClient"/>
</connectionStrings>
```

belindoc.com

```
<add name="DbConnectionString"
connectionString="Data Source=.;Initial Catalog=ExampleDatabase;Integrated Security=True"
providerName="System.Data.SqlClient"/>
</connectionStrings>
```

第5章：实体框架 Code First 迁移

第5.1节：启用迁移

要在实体框架中启用 Code First 迁移，请使用以下命令

```
Enable-Migrations
```

在 包管理控制台 中。

您需要有一个有效的 DbContext 实现，其中包含由 EF 管理的数据库对象。在此示例中，数据库上下文将包含对象 BlogPost 和 Author：

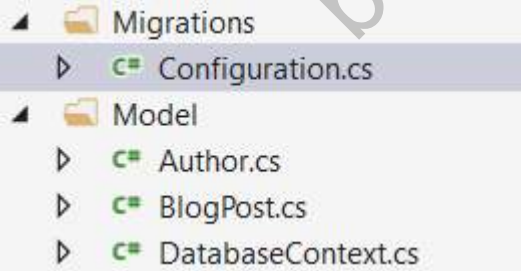
```
internal class DatabaseContext: DbContext
{
    public DbSet<Author> Authors { get; set; }

    public DbSet<BlogPost> BlogPosts { get; set; }
}
```

执行命令后，应显示以下输出：

```
PM> Enable-Migrations
正在检查 上下文是否 目标为现有数据库...
Code First 迁移已为项目 <YourProjectName> 启用。
PM>
```

此外，应该会出现一个名为Migrations的新文件夹，里面有一个名为Configuration.cs的文件：



下一步是创建你的第一个数据库迁移脚本，该脚本将创建初始数据库（见下一个示例）。

第5.2节：添加你的第一个迁移

启用迁移后（请参阅此示例），你现在可以创建第一个迁移，包含所有数据库表、索引和连接的初始创建。

可以使用以下命令创建迁移

```
Add-Migration <migration-name>
```

该命令将创建一个新类，包含两个方法Up和Down，用于应用和撤销迁移。

现在根据上述示例执行命令，创建一个名为Initial的迁移：

Chapter 5: Entity framework Code First Migrations

Section 5.1: Enable Migrations

To enable Code First Migrations in entity framework, use the command

```
Enable-Migrations
```

on the *Package Manager Console*.

You need to have a valid DbContext implementation containing your database objects managed by EF. In this example the database context will contain to objects BlogPost and Author:

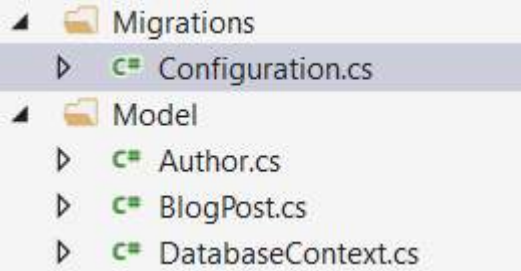
```
internal class DatabaseContext: DbContext
{
    public DbSet<Author> Authors { get; set; }

    public DbSet<BlogPost> BlogPosts { get; set; }
}
```

After executing the command, the following output should appear:

```
PM> Enable-Migrations
Checking if the context targets an existing database...
Code First Migrations enabled for project <YourProjectName>.
PM>
```

In addition, a new folder Migrations should appear with a single file Configuration.cs inside:



The next step would be to create your first database migration script which will create the initial database (see next example).

Section 5.2: Add your first migration

After you've enabled migrations (please refer to this example) you are now able to create your first migration containing an initial creation of all database tables, indexes and connections.

A migration can be created by using the command

```
Add-Migration <migration-name>
```

This command will create a new class containing two methods Up and Down that are used to apply and remove the migration.

Now apply the command based on the example above to create a migration called *Initial*:

PM> 添加-迁移初始
脚手架迁移'初始'。
此迁移文件的设计器代码包含您当前 Code First 模型的快照。该快照用于在您生成下一次迁移时计算模型的更改。如果您对模型进行了额外更改并希望将其包含在此迁移中，则可以通过运行命令重新生成迁移。

再次运行 'Add-Migration Initial'。

创建了一个新文件 timestamp_Initial.cs（这里只显示重要内容）：

```
public override void Up()
{
    CreateTable(
        "dbo.Authors",
        c => new
        {
            AuthorId = c.Int(nullable: false, identity: true),
            Name = c.String(maxLength: 128),
        })
        .PrimaryKey(t => t.AuthorId);

    CreateTable(
        "dbo.BlogPosts",
        c => new
        {
            Id = c.Int(nullable: false, identity: true),
            Title = c.String(nullable: false, maxLength: 128),
            Message = c.String(),
            Author_AuthorId = c.Int(),
        })
        .PrimaryKey(t => t.Id)
        .ForeignKey("dbo.Authors", t => t.Author_AuthorId)
        .Index(t => t.Author_AuthorId);
}

public override void Down()
{
    DropForeignKey("dbo.BlogPosts", "Author_AuthorId", "dbo.Authors");
    DropIndex("dbo.BlogPosts", new[] { "Author_AuthorId" });
    DropTable("dbo.BlogPosts");
    DropTable("dbo.Authors");
}
```

如您所见，在方法Up()中，创建了两个表Authors和BlogPosts，并相应地创建了字段。此外，通过添加字段Author_AuthorId建立了两个表之间的关系。另一方面，方法Down()尝试撤销迁移操作。

如果您对迁移操作有信心，可以使用以下命令将迁移应用到数据库：

Update-Database

所有待处理的迁移（在本例中为Initial迁移）都会被应用到数据库，随后会执行seed方法（相应的示例）

PM> update-database
指定'-Verbose'标志以查看正在应用到目标数据库的SQL语句。
正在应用显式迁移: [201609302203541_Initial]。
正在应用显式迁移: 201609302203541_Initial。

PM> Add-Migration Initial
Scaffolding migration 'Initial'.
The Designer Code for this migration file includes a snapshot of your current Code First model. This snapshot is used to calculate the changes to your model when you scaffold the next migration. If you make additional changes to your model that you want to include in this migration, then you can re-scaffold it by running 'Add-Migration Initial' again.

A new file timestamp_Initial.cs is created (only the important stuff is shown here):

```
public override void Up()
{
    CreateTable(
        "dbo.Authors",
        c => new
        {
            AuthorId = c.Int(nullable: false, identity: true),
            Name = c.String(maxLength: 128),
        })
        .PrimaryKey(t => t.AuthorId);

    CreateTable(
        "dbo.BlogPosts",
        c => new
        {
            Id = c.Int(nullable: false, identity: true),
            Title = c.String(nullable: false, maxLength: 128),
            Message = c.String(),
            Author_AuthorId = c.Int(),
        })
        .PrimaryKey(t => t.Id)
        .ForeignKey("dbo.Authors", t => t.Author_AuthorId)
        .Index(t => t.Author_AuthorId);
}

public override void Down()
{
    DropForeignKey("dbo.BlogPosts", "Author_AuthorId", "dbo.Authors");
    DropIndex("dbo.BlogPosts", new[] { "Author_AuthorId" });
    DropTable("dbo.BlogPosts");
    DropTable("dbo.Authors");
}
```

As you can see, in method Up() two tables Authors and BlogPosts are created and the fields are created accordingly. In addition, the relation between the two tables is created by adding the field Author_AuthorId. On the other side the method Down() tries to reverse the migration activities.

If you feel confident with your migration, you can apply the migration to the database by using the command:

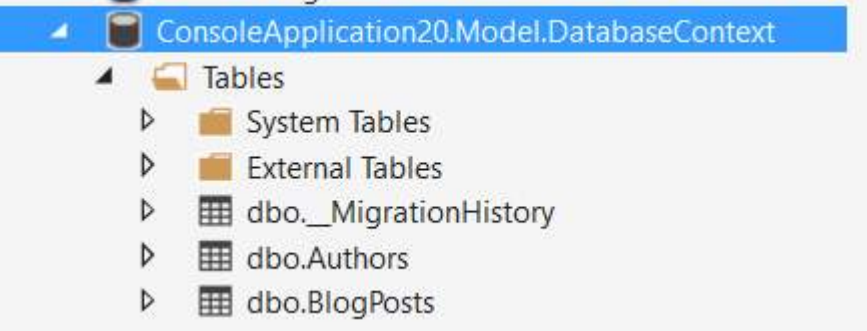
Update-Database

All pending migrations (in this case the Initial-migration) are applied to the database and afterwards the seed method is applied (the appropriate example)

PM> update-database
Specify the '-Verbose' flag to view the SQL statements being applied to the target database.
Applying explicit migrations: [201609302203541_Initial].
Applying explicit migration: 201609302203541_Initial.

正在运行Seed方法。

您可以在SQL资源管理器中查看活动结果：



对于命令Add-Migration和Update-Database，有多种选项可用来调整这些操作。要查看所有选项，请使用

get-help 添加-迁移

和

get-help 更新-数据库

第5.3节：在代码中执行“Update-Database”

运行在非开发环境中的应用程序通常需要数据库更新。在使用Add-Migration命令创建数据库补丁后，需要在其他环境中运行更新，然后在测试环境中也运行更新。

常见的挑战有：

- 生产环境中未安装Visual Studio，且
- 实际中不允许连接到连接/客户环境。

一种解决方法是以下代码序列，它检查需要执行的更新，并按顺序执行。请确保适当的事务和异常处理，以防出错时数据不会丢失。

```
void UpdateDatabase(MyDbConfiguration configuration) {
    DbMigrator dbMigrator = new DbMigrator( configuration);
    if ( dbMigrator.GetPendingMigrations().Any() )
    {
        // 有待处理的迁移，运行迁移任务
        dbMigrator.Update();
    }
}
```

where MyDbConfiguration 是你在源码中某处的迁移配置：

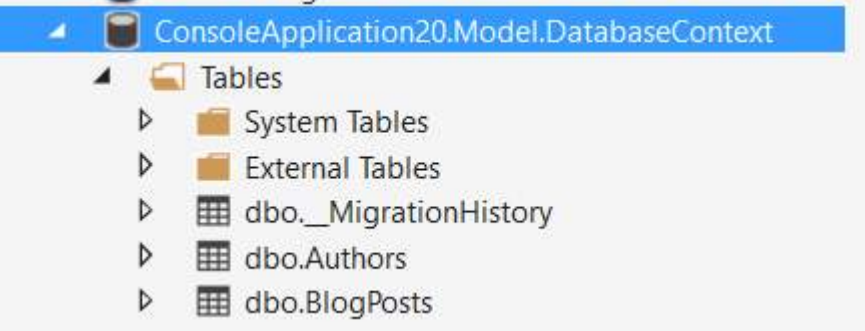
```
public class MyDbConfiguration : DbMigrationsConfiguration<ApplicationDbContext>
```

第5.4节：迁移期间的数据填充

启用并创建迁移后，可能需要初始填充或迁移数据库中的数据。这里有几种可能的方法，但对于简单的迁移，你可以使用调用 enable-migrations 后生成的 Configuration 文件中的 'Seed()' 方法。

Running Seed method.

You can see the results of the activities in the SQL explorer:



For the commands Add-Migration and Update-Database several options are available which can be used to tweak the activities. To see all options, please use

get-help Add-Migration

and

get-help Update-Database

Section 5.3: Doing "Update-Database" within your code

Applications running in non-development environments often require database updates. After using the Add-Migration command to create your database patches there's the need to run the updates on other environments, and then the test environment as well.

Challenges commonly faced are:

- no Visual Studio installed on production environments, and
- no connections allowed to connection/customer environments in real life.

A workaround is the following code sequence which checks for updates to be performed, and executes them in order. Please ensure proper transactions & exception handling to ensure no data gets lost in case of errors.

```
void UpdateDatabase(MyDbConfiguration configuration) {
    DbMigrator dbMigrator = new DbMigrator( configuration);
    if ( dbMigrator.GetPendingMigrations().Any() )
    {
        // there are pending migrations run the migration job
        dbMigrator.Update();
    }
}
```

where MyDbConfiguration is your migration setup somewhere in your sources:

```
public class MyDbConfiguration : DbMigrationsConfiguration<ApplicationDbContext>
```

Section 5.4: Seeding Data during migrations

After enabling and creating migrations there might be a need to initially fill or migrate data in your database. There are several possibilities but for simple migrations you can use the method 'Seed()' in the file Configuration created after calling enable-migrations.

Seed()函数以数据库上下文作为唯一参数，你可以在此函数内执行EF操作：

```
protected override void Seed(Model.DatabaseContext context);
```

你可以在Seed()中执行各种操作。如果发生任何失败，整个事务（包括已应用的补丁）都会被回滚。

一个仅在表为空时创建数据的示例函数可能如下所示：

```
protected override void Seed(Model.DatabaseContext context)
{
    if (!context.Customers.Any()) {
        Customer c = new Customer{ Id = 1, Name = "Demo" };
        context.Customers.Add(c);
    }
    context.SaveChanges();
}
```

EF 开发者提供的一个很好的功能是扩展方法 AddOrUpdate()。该方法允许基于主键更新数据，或者如果数据不存在则插入数据（示例取自 Configuration.cs 的生成源代码）：

```
protected override void Seed(Model.DatabaseContext context)
{
    context.People.AddOrUpdate(
        p => p.FullName,
        new Person { FullName = "Andrew Peters" },
        new Person { FullName = "Brice Lambson" },
        new Person { FullName = "Rowan Miller" }
    );
}
```

请注意，Seed() 在最后一个补丁应用后被调用。如果需要在补丁期间进行迁移或数据填充，则需要使用其他方法。

第5.5节：Entity Framework Code First 迁移步骤详解

1. 创建一个控制台应用程序。
2. 通过在“包管理器”中运行 Install-Package EntityFramework 来安装 EntityFramework NuGet 包控制台
3. 在 app.config 文件中添加您的连接字符串，包含 providerName="System.Data.SqlClient" 非常重要。
providerName="System.Data.SqlClient" 应包含在您的连接字符串中。
4. 创建一个公共类，名称随意，比如 "Blog"
5. 创建一个继承自 DbContext 的上下文类，比如 "BlogContext"
6. 在您的上下文中定义一个 DbSet 类型的属性，类似如下：

```
public class Blog
{
    public int Id { get; set; }

    public string Name { get; set; }
}
```

The Seed() function retrieves a database context as it's only parameter and you are able to perform EF operations inside this function:

```
protected override void Seed(Model.DatabaseContext context);
```

You can perform all types of activities inside Seed(). In case of any failure the complete transaction (even the applied patches) are being rolled back.

An example function that creates data only if a table is empty might look like this:

```
protected override void Seed(Model.DatabaseContext context)
{
    if (!context.Customers.Any()) {
        Customer c = new Customer{ Id = 1, Name = "Demo" };
        context.Customers.Add(c);
        context.SaveChanges();
    }
}
```

A nice feature provided by the EF developers is the extension method AddOrUpdate(). This method allows to update data based on the primary key or to insert data if it does not exist already (the example is taken from the generated source code of Configuration.cs):

```
protected override void Seed(Model.DatabaseContext context)
{
    context.People.AddOrUpdate(
        p => p.FullName,
        new Person { FullName = "Andrew Peters" },
        new Person { FullName = "Brice Lambson" },
        new Person { FullName = "Rowan Miller" }
    );
}
```

Please be aware that Seed() is called after the **last** patch has been applied. If you need to migration or seed data during patches, other approaches need to be used.

Section 5.5: Initial Entity Framework Code First Migration Step by Step

1. Create a console application.
2. Install EntityFramework nuget package by running Install-Package EntityFramework in "Package Manager Console"
3. Add your connection string in app.config file , It's important to include providerName="System.Data.SqlClient" in your connection.
4. Create a public class as you wish , some thing like "Blog"
5. Create Your ContextClass which inherit from DbContext , some thing like "BlogContext"
6. Define a property in your context of DbSet type , some thing like this：

```
public class Blog
{
    public int Id { get; set; }

    public string Name { get; set; }
}
```

```
public class BlogContext : DbContext
{
    public BlogContext() : base("name=Your_Connection_Name")
    {
    }

    public virtual DbSet<Blog> Blogs { get; set; }
}
```

7. 在构造函数中传递连接名称（这里是 Your_Connection_Name）非常重要
8. 在程序包管理器控制台运行 Enable-Migration 命令，这将在你的项目
9. 运行 Add-Migration Your_Arbitrary_Migraiton_Name 命令，这将在 migrations 文件夹中创建一个包含 Up() 和 Down() 两个方法的迁移类
10. 运行 Update-Database 命令以创建包含 blog 表的数据库

第 5.6 节：迁移过程中使用 Sql()

例如：您打算将现有列从非必填迁移为必填。在这种情况下，您可能需要在迁移中为那些被更改字段实际为NULL的行填充一些默认值。如果默认值很简单（例如“0”），您可以在列定义中使用default或defaultSql属性。如果情况不那么简单，您可以在迁移的Up()或Down()成员函数中使用Sql()函数。

这是一个例子。假设有一个包含电子邮件地址作为数据集一部分的Author类。现在我们决定将电子邮件地址设为必填字段。为了迁移现有列，业务部门有一个smart的想法，即创建类似fullname@example.com的虚拟电子邮件地址，其中fullname是作者的全名且不含空格。给字段Email添加[Required]属性将生成如下迁移：

```
public partial class AuthorsEmailRequired : DbMigration
{
    public override void Up()
    {
        AlterColumn("dbo.Authors", "Email", c => c.String(nullable: false, maxLength: 512));
    }

    public override void Down()
    {
        AlterColumn("dbo.Authors", "Email", c => c.String(maxLength: 512));
    }
}
```

如果数据库中存在某些NULL字段，则此操作会失败：

无法将值NULL插入到列'Email'，表'App.Model.DatabaseContext.dbo.Authors'；
该列不允许为空。更新失败。

在AlterColumn命令之前添加如下内容会有所帮助：

```
Sql(@"Update dbo.Authors
set Email = REPLACE(name, ' ', '') + N'@example.com'
where Email is null");
```

该update数据库调用成功，表格如下所示（示例数据）：

```
public class BlogContext : DbContext
{
    public BlogContext() : base("name=Your_Connection_Name")
    {
    }

    public virtual DbSet<Blog> Blogs { get; set; }
}
```

7. It's important to pass the connection name in constructor (here Your_Connection_Name)
8. In Package Manager Console run Enable-Migration command , This will create a migration folder in your project
9. Run Add-Migration Your_Arbitrary_Migraiton_Name command , this will create a migration class in migrations folder with two method Up() and Down()
10. Run Update-Database command in order to create a database with a blog table

Section 5.6: Using Sql() during migrations

For example: You are going to migrate an existing column from non-required to required. In this case you might need to fill some default values in your migration for rows where the altered fields are actually **NULL**. In case the default value is simple (e.g. "0") you might use a **default** or **defaultSql** property in your column definition. In case it's not so easy, you may use the **Sql()** function in **Up()** or **Down()** member functions of your migrations.

Here's an example. Assuming a class *Author* which contains an email-address as part of the data set. Now we decide to have the email-address as a required field. To migrate existing columns the business has the *smart* idea of creating dummy email-addresses like fullname@example.com, where full name is the authors full name without spaces. Adding the [Required] attribute to the field Email would create the following migration:

```
public partial class AuthorsEmailRequired : DbMigration
{
    public override void Up()
    {
        AlterColumn("dbo.Authors", "Email", c => c.String(nullable: false, maxLength: 512));
    }

    public override void Down()
    {
        AlterColumn("dbo.Authors", "Email", c => c.String(maxLength: 512));
    }
}
```

This would fail in case some NULL fields are inside the database:

Cannot insert the value NULL into column 'Email', table 'App.Model.DatabaseContext.dbo.Authors';
column does not allow nulls. UPDATE fails.

Adding the following like **before** the AlterColumn command will help:

```
Sql(@"Update dbo.Authors
set Email = REPLACE(name, ' ', '') + N'@example.com'
where Email is null");
```

The update-database call succeeds and the table looks like this (example data shown):

dbo.Authors [Data] 201610071531268_A...sEmailRequired.cs Output

Max Rows: 1000

	AuthorId	Name	Email
▶	1	Stephen Reindl	StephenReindl@example.com
	2	DemoUser	DemoUser@example.com
	3	Test User 2	TestUser2@example.com
	4	Field user	demo@demo.com
*	NULL	NULL	NULL

其他用法

您可以使用Sql()函数来执行数据库中的所有类型的DML和DDL操作。它作为迁移事务的一部分执行；如果SQL执行失败，整个迁移将失败并回滚。

dbo.Authors [Data] 201610071531268_A...sEmailRequired.cs Output

Max Rows: 1000

	AuthorId	Name	Email
▶	1	Stephen Reindl	StephenReindl@example.com
	2	DemoUser	DemoUser@example.com
	3	Test User 2	TestUser2@example.com
	4	Field user	demo@demo.com
*	NULL	NULL	NULL

Other Usage

You may use the Sql() function for all types of DML and DDL activities in your database. It is executed as part of the migration transaction; If the SQL fails, the complete migration fails and a rollback is done.

第6章：使用 EntityFramework（代码优先）的继承

第6.1节：每层表（Table per hierarchy）

这种方法将在数据库中生成一个表来表示所有的继承结构。

示例：

```
public abstract class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime BirthDate { get; set; }
}

public class Employee : Person
{
    public DateTime AdmissionDate { get; set; }
    public string JobDescription { get; set; }
}

public class Customer : Person
{
    public DateTime LastPurchaseDate { get; set; }
    public int TotalVisits { get; set; }
}

// 在 DbContext 上
public DbSet<Person> People { get; set; }
public DbSet<Employee> Employees { get; set; }
public DbSet<Customer> Customers { get; set; }
```

生成的表将是：

表：People 字段：Id Name BirthDate Discriminator AdmissionDate JobDescription LastPurchaseDate TotalVisits

其中 'Discriminator' 将保存继承中子类的名称，'AdmissionDate'、'JobDescription'、'LastPurchaseDate'、'TotalVisits' 是可空的。

优点

- 性能更好，因为不需要连接，尽管对于很多列，数据库可能需要多次分页操作。
- 使用 and 创建简单
- 易于添加更多子类 and 字段

缺点

- 违反第三范式[维基百科：第三范式](#)
- 产生大量可为空字段

第6.2节：每类型一表

该方法将在数据库中生成(n+1)个表来表示所有继承结构，其中n是

Chapter 6: Inheritance with EntityFramework (Code First)

Section 6.1: Table per hierarchy

This approach will generate one table on the database to represent all the inheritance structure.

Example:

```
public abstract class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime BirthDate { get; set; }
}

public class Employee : Person
{
    public DateTime AdmissionDate { get; set; }
    public string JobDescription { get; set; }
}

public class Customer : Person
{
    public DateTime LastPurchaseDate { get; set; }
    public int TotalVisits { get; set; }
}

// On DbContext
public DbSet<Person> People { get; set; }
public DbSet<Employee> Employees { get; set; }
public DbSet<Customer> Customers { get; set; }
```

The table generated will be:

Table: People Fields: Id Name BirthDate Discriminator AdmissionDate JobDescription LastPurchaseDate TotalVisits

Where 'Discriminator' will hold the name of the subclass on the inheritance and 'AdmissionDate', 'JobDescription', 'LastPurchaseDate', 'TotalVisits' are nullable.

Advantages

- Better performance since no joins are required although for too many columns the database might require many paging operations.
- Simple to use and create
- Easy to add more subclasses and fields

Disadvantages

- Violates the 3rd Normal Form [Wikipedia: Third normal form](#)
- Creates lots of nullable fields

Section 6.2: Table per type

This approach will generate (n+1) tables on the database to represent all the inheritance structure where n is the

子类的数量。

操作方法：

```
public abstract class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime BirthDate { get; set; }
}

[Table("Employees")]
public class Employee : Person
{
    public DateTime AdmissionDate { get; set; }
    public string JobDescription { get; set; }
}

[Table("Customers")]
public class Customer : Person
{
    public DateTime LastPurchaseDate { get; set; }
    public int TotalVisits { get; set; }
}

// 在 DbContext 上
public DbSet<Person> People { get; set; }
public DbSet<Employee> Employees { get; set; }
public DbSet<Customer> Customers { get; set; }
```

生成的表将是：

- 表：People 字段：Id Name BirthDate
- 表：Employees 字段：PersonId AdmissionDate JobDescription
- 表：Customers 字段：PersonId LastPurchaseDate TotalVisits

其中所有表中的'PersonId'将作为主键并约束于People.Id

优点

- 规范化表格
- 易于添加列和子类
- 无可空列

缺点

- 需要连接以检索数据
- 子类推断更耗费资源

number of subclasses.

How to:

```
public abstract class Person
{
    public int Id { get; set; }
    public string Name { get; set; }
    public DateTime BirthDate { get; set; }
}

[Table("Employees")]
public class Employee : Person
{
    public DateTime AdmissionDate { get; set; }
    public string JobDescription { get; set; }
}

[Table("Customers")]
public class Customer : Person
{
    public DateTime LastPurchaseDate { get; set; }
    public int TotalVisits { get; set; }
}

// On DbContext
public DbSet<Person> People { get; set; }
public DbSet<Employee> Employees { get; set; }
public DbSet<Customer> Customers { get; set; }
```

The table generated will be:

- Table: People Fields: Id Name BirthDate
- Table: Employees Fields: PersonId AdmissionDate JobDescription
- Table: Customers: Fields: PersonId LastPurchaseDate TotalVisits
- Where 'PersonId' on all tables will be a primary key and a constraint to People.Id

Advantages

- Normalized tables
- Easy to add columns and subclasses
- No nullable columns

Disadvantages

- Join is required to retrieve the data
- Subclass inference is more expensive

第7章：代码优先 - Fluent API

第7.1节：映射模型

EntityFramework Fluent API 是一种强大且优雅的方式，将你的**代码优先**领域模型映射到底层数据库。这也可以用于**已有数据库的代码优先**。使用*Fluent API*时，你有两个选项：你可以直接在OnModelCreating方法中映射你的模型，或者你可以创建继承自EntityTypeConfiguration的映射类，然后在OnModelCreating方法中将这些模型添加到modelBuilder。第二种选项是我更喜欢的，也是我将要展示示例的方式。

第一步：创建模型。

```
public class Employee
{
    public int Id { get; set; }
    public string Surname { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public short Age { get; set; }
    public decimal MonthlySalary { get; set; }

    public string FullName
    {
        get
        {
            return $"{姓} {名} {姓氏}";
        }
    }
}
```

第二步：创建映射类

```
public class 员工映射
: EntityTypeConfiguration<员工>
{
    public 员工映射()
    {
        // 主键
        this.HasKey(m => m.Id);

        this.Property(m => m.Id)
            .HasColumnType("int")
            .HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity);

        // 属性
        this.Property(m => m.Surname)
            .HasMaxLength(50);

        this.Property(m => m.FirstName)
            .IsRequired()
            .HasMaxLength(50);

        this.Property(m => m.LastName)
            .HasMaxLength(50);

        this.Property(m => m.Age)
            .HasColumnType("smallint");

        this.Property(m => m.MonthlySalary)
            .HasColumnType("number")
    }
}
```

Chapter 7: Code First - Fluent API

Section 7.1: Mapping models

EntityFramework Fluent API is a powerful and elegant way of mapping your **code-first** domain models to underlying database. This also can be used with *code-first with existing database*. You have two options when using *Fluent API*: you can directly map your models on *OnModelCreating* method or you can create mapper classes which inherits from *EntityTypeConfiguration* and then add that models to *modelBuilder* on *OnModelCreating* method. *Second* option is which I prefer and am going to show example of it.

Step one: Create model.

```
public class Employee
{
    public int Id { get; set; }
    public string Surname { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public short Age { get; set; }
    public decimal MonthlySalary { get; set; }

    public string FullName
    {
        get
        {
            return $"{Surname} {FirstName} {LastName}";
        }
    }
}
```

Step two: Create mapper class

```
public class EmployeeMap
: EntityTypeConfiguration<Employee>
{
    public EmployeeMap()
    {
        // Primary key
        this.HasKey(m => m.Id);

        this.Property(m => m.Id)
            .HasColumnType("int")
            .HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity);

        // Properties
        this.Property(m => m.Surname)
            .HasMaxLength(50);

        this.Property(m => m.FirstName)
            .IsRequired()
            .HasMaxLength(50);

        this.Property(m => m.LastName)
            .HasMaxLength(50);

        this.Property(m => m.Age)
            .HasColumnType("smallint");

        this.Property(m => m.MonthlySalary)
            .HasColumnType("number")
    }
}
```

```
        .HasPrecision(14, 5);

        this.Ignore(m => m.FullName);

        // 表和列映射
        this.ToTable("TABLE_NAME", "SCHEMA_NAME");
        this.Property(m => m.Id).HasColumnName("ID");
        this.Property(m => m.Surname).HasColumnName("SURNAME");
        this.Property(m => m.FirstName).HasColumnName("FIRST_NAME");
        this.Property(m => m.LastName).HasColumnName("LAST_NAME");
        this.Property(m => m.Age).HasColumnName("AGE");
        this.Property(m => m.MonthlySalary).HasColumnName("MONTHLY_SALARY");
    }
}
```

让我们解释映射：

- **HasKey** - 定义主键。也可以使用复合主键。例如：`this.HasKey(m => new { m.DepartmentId, m.PositionId })`。
- **Property** - 允许我们配置模型属性。
- **HasColumnType** - 指定数据库级别的列类型。请注意，不同数据库如Oracle和MS SQL可能不同。
- **HasDatabaseGeneratedOption** - 指定属性是否在数据库级别计算。数值型主键默认是DatabaseGeneratedOption.Identity，如果不希望如此，应指定DatabaseGeneratedOption.None。
- **HasMaxLength** - 限制字符串长度。
- **IsRequired** - 标记属性为必填。
- **HasPrecision** - 允许我们为小数指定精度。
- **Ignore** - 完全忽略属性，不映射到数据库。我们忽略了FullName，因为不希望该列出现在表中。
- **.ToTable** - 指定模型的表名和（可选的）架构名。
- **HasColumnName** - 将属性与列名关联。当属性名和列名相同时不需要此项。

第三步：将映射类添加到配置中。

我们需要告诉 EntityFramework 使用我们的映射器类。为此，我们必须将其添加到 `modelBuilder.Configurations` on `OnModelCreating` 方法中：

```
public class DbContext()
    : base("Name=DbContext")
{
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Configurations.Add(new EmployeeMap());
    }
}
```

就是这样。我们已经准备好开始了。

第7.2节：复合主键

通过使用 `.HasKey()` 方法，可以显式地将一组属性配置为实体的复合主键。

```
using System.Data.Entity;
```

```
        .HasPrecision(14, 5);

        this.Ignore(m => m.FullName);

        // Table & column mappings
        this.ToTable("TABLE_NAME", "SCHEMA_NAME");
        this.Property(m => m.Id).HasColumnName("ID");
        this.Property(m => m.Surname).HasColumnName("SURNAME");
        this.Property(m => m.FirstName).HasColumnName("FIRST_NAME");
        this.Property(m => m.LastName).HasColumnName("LAST_NAME");
        this.Property(m => m.Age).HasColumnName("AGE");
        this.Property(m => m.MonthlySalary).HasColumnName("MONTHLY_SALARY");
    }
}
```

Let us explain mappings:

- **HasKey** - defines the primary key. *Composite primary keys* can also be used. For example: `this.HasKey(m => new { m.DepartmentId, m.PositionId })`.
- **Property** - lets us to configure model properties.
- **HasColumnType** - specify database level column type. Please note that, it can be different for different databases like Oracle and MS SQL.
- **HasDatabaseGeneratedOption** - specifies if property is calculated at database level. Numeric PKs are *DatabaseGeneratedOption.Identity* by default, you should specify *DatabaseGeneratedOption.None* if you do not want them to be so.
- **HasMaxLength** - limits the length of string.
- **IsRequired** - marks the property as required.
- **HasPrecision** - lets us to specify precision for decimals.
- **Ignore** - Ignores property completely and does not map it to database. We ignored FullName, because we do not want this column at our table.
- **ToTable** - specify table name and schema name (optional) for model.
- **HasColumnName** - relate property with column name. This is not needed when property names and column names are identical.

Step three: Add mapping class to configurations.

We need to tell EntityFramework to use our mapper class. To do so, we have to add it to `modelBuilder.Configurations` on `OnModelCreating` method:

```
public class DbContext()
    : base("Name=DbContext")
{
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Configurations.Add(new EmployeeMap());
    }
}
```

And that is it. We are all set to go.

Section 7.2: Composite Primary Key

By using the `.HasKey()` method, a set of properties can be explicitly configured as the composite primary key of the entity.

```
using System.Data.Entity;
```

```
// ..

public class PersonContext : DbContext
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..

        modelBuilder.Entity<Person>().HasKey(p => new { p.FirstName, p.LastName });
    }
}
```

第7.3节：最大长度

通过使用 .HasMaxLength() 方法，可以为属性配置最大字符数。

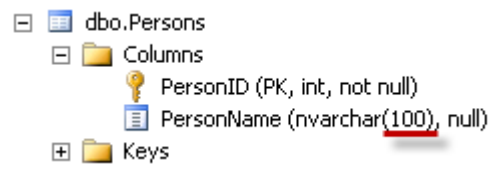
```
using System.Data.Entity;
// ..

public class PersonContext : DbContext
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..

        modelBuilder.Entity<Person>()
            .Property(t => t.Name)
            .HasMaxLength(100);
    }
}
```

指定列长度后生成的列：



第7.4节：主键

通过使用 .HasKey() 方法，可以显式地将属性配置为实体的主键。

```
using System.Data.Entity;
// ..

public class PersonContext : DbContext
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..

        modelBuilder.Entity<Person>().HasKey(p => p.PersonKey);
    }
}
```

```
// ..

public class PersonContext : DbContext
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..

        modelBuilder.Entity<Person>().HasKey(p => new { p.FirstName, p.LastName });
    }
}
```

Section 7.3: Maximum Length

By using the .HasMaxLength() method, the maximum character count can be configured for a property.

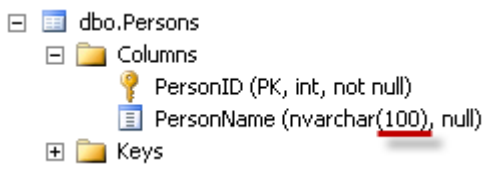
```
using System.Data.Entity;
// ..

public class PersonContext : DbContext
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..

        modelBuilder.Entity<Person>()
            .Property(t => t.Name)
            .HasMaxLength(100);
    }
}
```

The resulting column with the specified column length:



Section 7.4: Primary Key

By using the .HasKey() method, a property can be explicitly configured as primary key of the entity.

```
using System.Data.Entity;
// ..

public class PersonContext : DbContext
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..

        modelBuilder.Entity<Person>().HasKey(p => p.PersonKey);
    }
}
```

```
}
```

第7.5节：必填属性（非空）

通过使用 `.IsRequired()` 方法，可以将属性指定为必填，这意味着该列将具有 NOT NULL 约束。

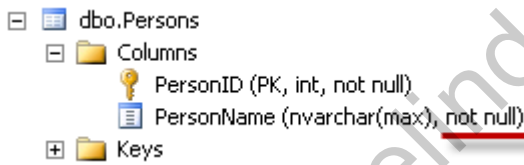
```
using System.Data.Entity;
// ..

public class PersonContext : DbContext
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..

        modelBuilder.Entity<Person>()
            .Property(t => t.Name)
            .IsRequired();
    }
}
```

生成的带有 NOT NULL 约束的列：



第7.6节：显式外键命名

当模型中存在导航属性时，实体框架会自动创建一个外键列。如果需要指定特定的外键名称，但该名称未作为模型中的属性包含，则可以使用 Fluent API 明确设置。通过在建立外键关系时使用 `Map` 方法，可以为外键使用任何唯一名称。

```
public class Company
{
    public int Id { get; set; }
}

public class Employee
{
    property int Id { get; set; }
    property Company Employer { get; set; }
}

public class EmployeeContext : DbContext
{
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Employee>()
            .HasRequired(x => x.Employer)
            .WithRequiredDependent()
            .Map(m => m.MapKey("EmployerId"));
    }
}
```

```
}
```

Section 7.5: Required properties (NOT NULL)

By using the `.IsRequired()` method, properties can be specified as mandatory, which means that the column will have a NOT NULL constraint.

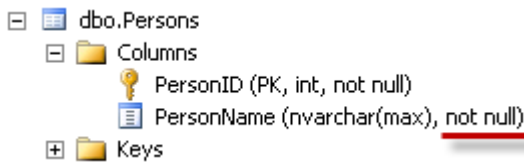
```
using System.Data.Entity;
// ..

public class PersonContext : DbContext
{
    // ..

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        // ..

        modelBuilder.Entity<Person>()
            .Property(t => t.Name)
            .IsRequired();
    }
}
```

The resulting column with the NOT NULL constraint:



Section 7.6: Explict Foreign Key naming

When a navigation property exist on a model, Entity Framework will automatically create a Foreign Key column. If a specific Foreign Key name is desired but is not contained as a property in the model, it can be set explicitly using the Fluent API. By utilizing the `Map` method while establishing the Foreign Key relationship, any unique name can be used for Foreign Keys.

```
public class Company
{
    public int Id { get; set; }
}

public class Employee
{
    property int Id { get; set; }
    property Company Employer { get; set; }
}

public class EmployeeContext : DbContext
{
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Employee>()
            .HasRequired(x => x.Employer)
            .WithRequiredDependent()
            .Map(m => m.MapKey("EmployerId"));
    }
}
```



```
}
```

指定关系后，Map 方法允许通过执行 MapKey 显式设置外键名称
MapKey。在此示例中，原本会生成的列名 Employer_Id 现在变为 EmployerId。

belindoc.com

```
}
```

After specifying the relationship, the Map method allows the Foreign Key name to be explicitly set by executing MapKey. In this example, what would have resulted in a column name of Employer_Id is now EmployerId.

第8章：使用 Entity Framework Code First 映射关系：一对一及其变体

本主题讨论如何使用 Entity Framework 映射一对一类型的关系。

第8.1节：一对零或一的映射

那么我们再说一次，你有以下模型：

```
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
}

public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
}

public class MyDemoContext : DbContext
{
    public DbSet<Person> People { get; set; }
    public DbSet<Car> Cars { get; set; }
}
```

现在你想设置这样一个规范：一个人可以拥有一辆或没有车，并且每辆车恰好属于一个人（关系是双向的，所以如果CarA属于PersonA，那么PersonA“拥有”CarA）。

所以我们稍微修改一下模型：添加导航属性和外键属性：

```
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
    public int CarId { get; set; }
    public virtual Car Car { get; set; }
}

public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public int PersonId { get; set; }
    public virtual Person Person { get; set; }
}
```

配置如下：

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
    public CarEntityTypeConfiguration()
    {
    }
```

Chapter 8: Mapping relationship with Entity Framework Code First: One-to-one and variations

This topic discusses how to map one-to-one type relationships using Entity Framework.

Section 8.1: Mapping one-to-zero or one

So let's say again that you have the following model:

```
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
}

public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
}

public class MyDemoContext : DbContext
{
    public DbSet<Person> People { get; set; }
    public DbSet<Car> Cars { get; set; }
}
```

And now you want to set it up so that you can express the following specification: one person can have one or zero car, and every car belongs to one person exactly (relationships are bidirectional, so if CarA belongs to PersonA, then PersonA 'owns' CarA).

So let's modify the model a bit: add the navigation properties and the foreign key properties:

```
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
    public int CarId { get; set; }
    public virtual Car Car { get; set; }
}

public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public int PersonId { get; set; }
    public virtual Person Person { get; set; }
}
```

And the configuration:

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
    public CarEntityTypeConfiguration()
    {
    }
```

```

        this.HasRequired(c => c.Person).WithOptional(p => p.Car);
    }
}

```

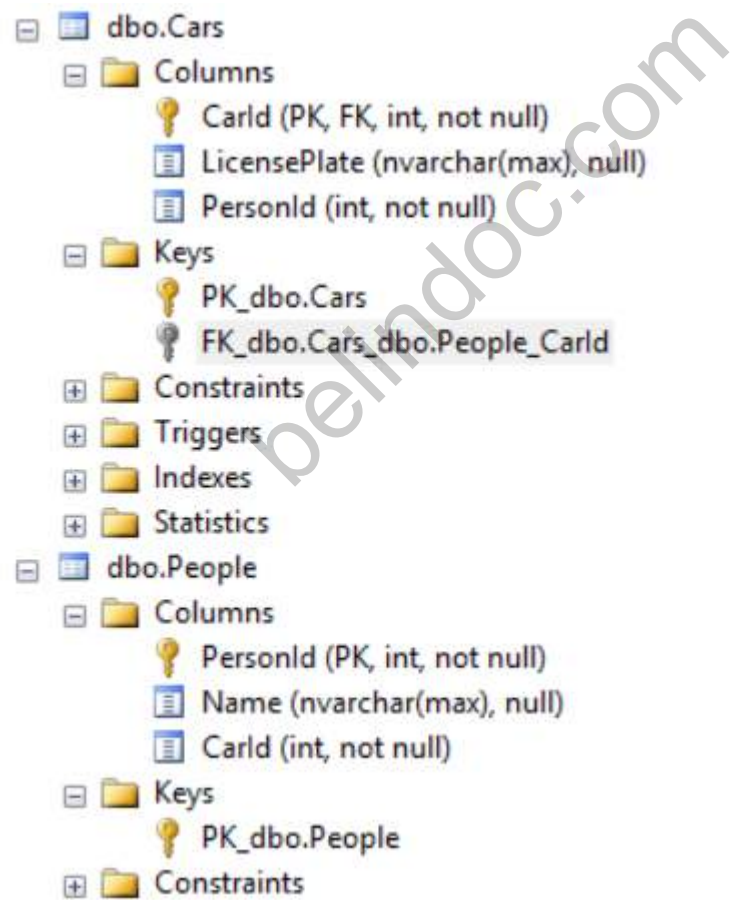
到这里应该不言自明。汽车有一个必需的人员（HasRequired()），而人员有一个可选的汽车（WithOptional()）。同样，无论你从哪一方配置这个关系都无所谓，只要在使用 Has/With 和 Required/Optional 的正确组合时小心即可。从Person这一方看，代码如下：

```

public class PersonEntityTypeConfiguration : EntityTypeConfiguration<Person>
{
    public PersonEntityTypeConfiguration()
    {
        this.HasOptional(p => p.Car).WithOptional(c => c.Person);
    }
}

```

现在让我们来看一下数据库模式：



仔细看：你可以看到在People中没有外键指向Car。而且，Car中的外键不是PersonId，而是CarId。以下是外键的实际脚本：

```

ALTER TABLE [dbo].[Cars] WITH CHECK ADD CONSTRAINT [FK_dbo.Cars_dbo.People_CarId] FOREIGN
KEY([CarId])
REFERENCES [dbo].[People] ([PersonId])

```

这意味着我们模型中存在的CarId和PersonId外键属性基本上被忽略了。它们存在于数据库中，但并不是外键，不像预期的那样。这是因为一对一映射不支持在你的EF模型中添加外键。这是因为一对一映射在关系数据库中相当有问题。

```

        this.HasRequired(c => c.Person).WithOptional(p => p.Car);
    }
}

```

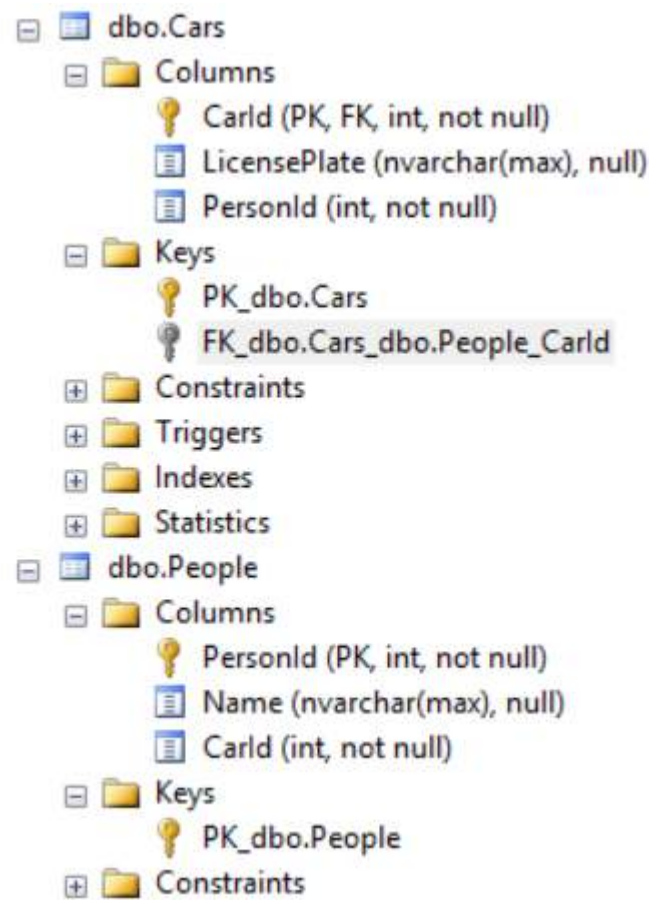
By this time this should be self-explanatory. The car has a required person ([HasRequired\(\)](#)), with the person having an optional car ([WithOptional\(\)](#)). Again, it doesn't matter which side you configure this relationship from, just be careful when you use the right combination of Has/With and Required/Optional. From the Person side, it would look like this:

```

public class PersonEntityTypeConfiguration : EntityTypeConfiguration<Person>
{
    public PersonEntityTypeConfiguration()
    {
        this.HasOptional(p => p.Car).WithOptional(c => c.Person);
    }
}

```

Now let's check out the db schema:



Look closely: you can see that there is no FK in People to refer to Car. Also, the FK in Car is not the PersonId, but the CarId. Here's the actual script for the FK:

```

ALTER TABLE [dbo].[Cars] WITH CHECK ADD CONSTRAINT [FK_dbo.Cars_dbo.People_CarId] FOREIGN
KEY([CarId])
REFERENCES [dbo].[People] ([PersonId])

```

So this means that the CarId and PersonId foreign key properties we have in the model are basically ignored. They are in the database, but they are not foreign keys, as it might be expected. That's because one-to-one mappings does not support adding the FK into your EF model. And that's because one-to-one mappings are quite problematic in a relational database.

其想法是每个人最多只能有一辆车，而那辆车只能属于那个人。或者可能存在没有关联车辆的人员记录。

那么这如何用外键来表示呢？显然，Car中可以有一个PersonId，People中可以有一个CarId。为了强制每个人只能有一辆车，PersonId必须在Car中唯一。但如果PersonId在People中唯一，那么如何添加两个或更多PersonId为NULL的记录（即多辆没有主人的车）呢？答案是：不行（实际上，你可以在SQL Server 2008及更高版本中创建过滤唯一索引，但暂且不谈这个技术细节；更不用说其他关系数据库管理系统了）。更不用说你同时指定关系两端的情况.....

唯一真正能强制执行此规则的方法是People和Car表具有“相同”的主键（连接记录中值相同）。为此，Car中的CarId必须既是主键又是指向People主键的外键。这会使整个模式变得混乱。当我使用这种方式时，我更倾向于将Car中的主键/外键命名为PersonId，并相应地进行配置：

```
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
    public virtual Car Car { get; set; }
}

public class Car
{
    public string LicensePlate { get; set; }
    public int PersonId { get; set; }
    public virtual Person Person { get; set; }
}

public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
    public CarEntityTypeConfiguration()
    {
        this.HasRequired(c => c.Person).WithOptional(p => p.Car);
        this.HasKey(c => c.PersonId);
    }
}
```

这不是理想的方案，但可能稍微好一点。不过，使用这个解决方案时必须保持警惕，因为它违背了常规的命名规范，可能会让你迷失方向。以下是从该模型生成的架构：

The idea is that every person can have exactly one car, and that car can only belong to that person. Or there might be person records, which do not have cars associated with them.

So how could this be represented with foreign keys? Obviously, there could be a PersonId in Car, and a CarId in People. To enforce that every person can have only one car, PersonId would have to be unique in Car. But if PersonId is unique in People, then how can you add two or more records where PersonId is NULL (more than one car that don't have owners)? Answer: you can't (well actually, you can create a filtered unique index in SQL Server 2008 and newer, but let's forget about this technicality for a moment; not to mention other RDBMS). Not to mention the case where you specify both ends of the relationship...

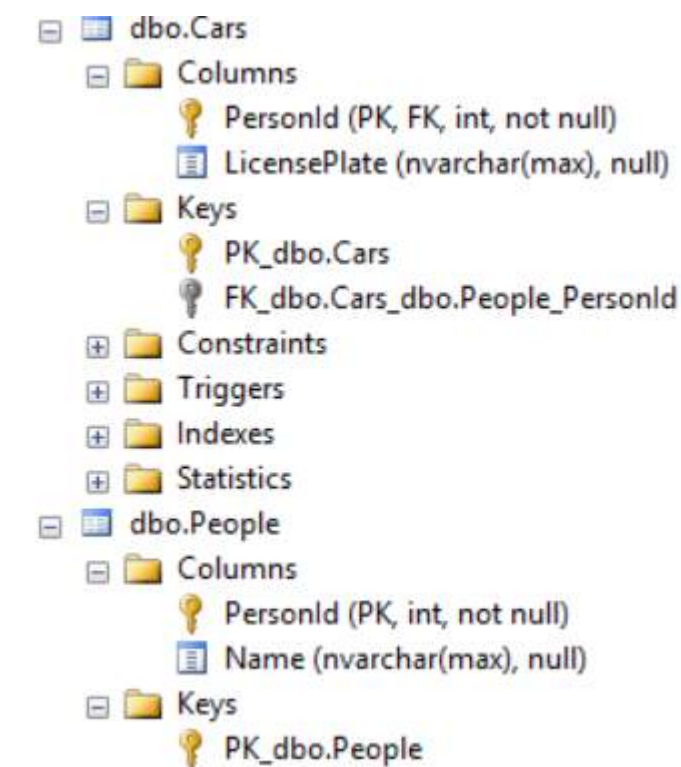
The only real way to enforce this rule if the People and the Car tables have the 'same' primary key (same values in the connected records). And to do this, CarId in Car must be both a PK and an FK to the PK of People. And this makes the whole schema a mess. When I use this I rather name the PK/FK in Car PersonId, and configure it accordingly:

```
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
    public virtual Car Car { get; set; }
}

public class Car
{
    public string LicensePlate { get; set; }
    public int PersonId { get; set; }
    public virtual Person Person { get; set; }
}

public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
    public CarEntityTypeConfiguration()
    {
        this.HasRequired(c => c.Person).WithOptional(p => p.Car);
        this.HasKey(c => c.PersonId);
    }
}
```

Not ideal, but maybe a bit better. Still, you have to be alert when using this solution, because it goes against the usual naming conventions, which might lead you astray. Here's the schema generated from this model:



所以这个关系不是由数据库架构强制执行的，而是由实体框架（Entity Framework）本身强制执行的。这就是为什么你在使用时必须非常小心，不要让任何人直接篡改数据库。

第8.2节：一对一映射

当双方都是必需时，一对一映射也是一件棘手的事情。

让我们想象一下这如何用外键来表示。再次说明，People表中有一个CarId，指向Car表中的CarId，而Car表中有一个PersonId，指向People表中的PersonId。

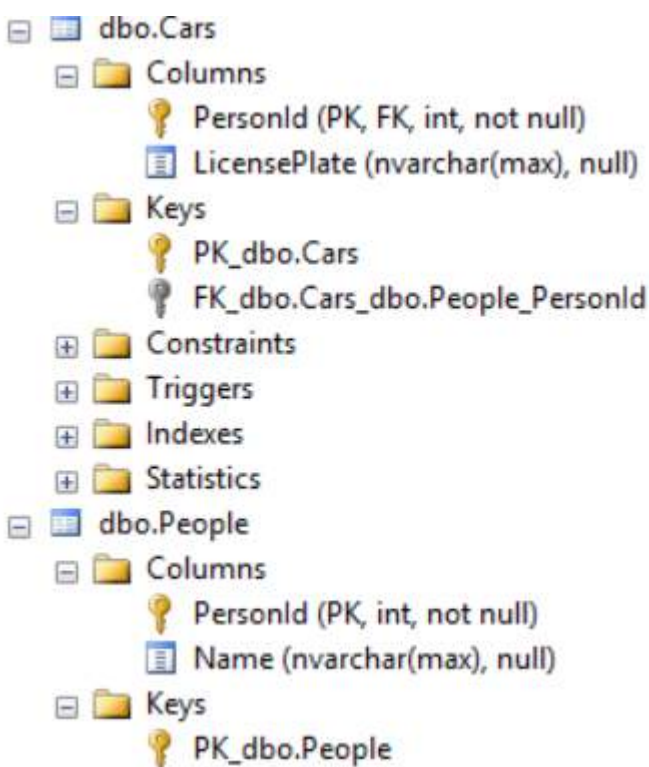
现在如果你想插入一条汽车记录，会发生什么？为了成功，必须在这条汽车记录中指定一个PersonId，因为这是必需的。并且为了使这个PersonId有效，对应的People表中的记录必须存在。好的，那么我们继续插入人员记录。但为了成功，人员记录中必须有一个有效的CarId——但那辆车还没有插入！这不可能，因为我们必须先插入被引用的人员记录。但我们又不能插入被引用的人员记录，因为它又引用回汽车记录，所以汽车记录必须先插入（外键套外键 :)）。

所以这也不能用“逻辑”的方式表示。你必须放弃其中一个外键。放弃哪一个由你决定。保留外键的一方称为“依赖方”，没有外键的一方称为“主方”。同样，为了确保依赖方的唯一性，主键必须是外键，因此添加一个外键列并将其导入模型是不被支持的。

配置如下：

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
    public CarEntityTypeConfiguration()
    {
        this.HasRequired(c => c.Person).WithRequiredDependent(p => p.Car);
        this.HasKey(c => c.PersonId);
    }
}
```

到现在你应该已经理解了它的逻辑 :) 只要记住你也可以选择另一边，



So this relationship is not enforced by the database schema, but by Entity Framework itself. That's why you have to be very careful when you use this, not to let anybody temper directly with the database.

Section 8.2: Mapping one-to-one

Mapping one-to-one (when both sides are required) is also a tricky thing.

Let's imagine how this could be represented with foreign keys. Again, a CarId in People that refers to CarId in Car, and a PersonId in Car that refers to the PersonId in People.

Now what happens if you want to insert a car record? In order for this to succeed, there must be a PersonId specified in this car record, because it is required. And for this PersonId to be valid, the corresponding record in People must exist. OK, so let's go ahead and insert the person record. But for this to succeed, a valid CarId must be in the person record — but that car is not inserted yet! It cannot be, because we have to insert the referred person record first. But we cannot insert the referred person record, because it refers back to the car record, so that must be inserted first (foreign key-ception :)).

So this cannot be represented the 'logical' way either. Again, you have to drop one of the foreign keys. Which one you drop is up to you. The side that is left with a foreign key is called the 'dependent', the side that is left without a foreign key is called the 'principal'. And again, to ensure the uniqueness in the dependent, the PK has to be the FK, so adding an FK column and importing that to your model is not supported.

So here's the configuration:

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
    public CarEntityTypeConfiguration()
    {
        this.HasRequired(c => c.Person).WithRequiredDependent(p => p.Car);
        this.HasKey(c => c.PersonId);
    }
}
```

By now you really should have gotten the logic of it :) Just remember that you can choose the other side as well, just

但要小心使用 WithRequired 的依赖方/主方版本（而且你仍然需要在 Car 中配置主键）。

```
public class PersonEntityTypeConfiguration : EntityTypeConfiguration<Person>
{
    public PersonEntityTypeConfiguration()
    {
        {
            this.HasRequired(p => p.Car).WithRequiredPrincipal(c => c.Person);
        }
    }
}
```

如果你查看数据库架构，会发现它和一对一或零对一的情况完全相同。这是因为这不是由架构强制执行的，而是由 EF 本身强制执行的。所以再次提醒，务必小心 :)

第8.3节：映射一对零或一对零

最后，我们简要看看双方都是可选的情况。

到现在你应该对这些例子感到厌烦了 :)，所以我不会深入细节，也不会探讨拥有两个外键的想法及其潜在问题，并提醒你
不要仅在 EF 本身而非架构中强制执行这些规则的危险。

这是您需要应用的配置：

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
    public CarEntityTypeConfiguration()
    {
        {
            this.HasOptional(c => c.Person).WithOptionalPrincipal(p => p.Car);
            this.HasKey(c => c.PersonId);
        }
    }
}
```

同样，你也可以从另一方进行配置，只要注意使用正确的方法 :)

be careful to use the Dependent/Principal versions of WithRequired (and you still have to configure the PK in Car).

```
public class PersonEntityTypeConfiguration : EntityTypeConfiguration<Person>
{
    public PersonEntityTypeConfiguration()
    {
        {
            this.HasRequired(p => p.Car).WithRequiredPrincipal(c => c.Person);
        }
    }
}
```

If you check the DB schema, you'll find that it's exactly the same as it was in the case of the one-to-one or zero solution. That's because again, this is not enforced by the schema, but by EF itself. So again, be careful :)

Section 8.3: Mapping one or zero-to-one or zero

And to finish off, let's briefly look at the case when both sides are optional.

By now you should be really bored with these examples :)，so I'm not going into the details and play with the idea of having two FK-s and the potential problems and warn you about the dangers of not enforcing these rules in the schema but in just EF itself.

Here's the config you need to apply:

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
    public CarEntityTypeConfiguration()
    {
        {
            this.HasOptional(c => c.Person).WithOptionalPrincipal(p => p.Car);
            this.HasKey(c => c.PersonId);
        }
    }
}
```

Again, you can configure from the other side as well, just be careful to use the right methods :)

第9章：使用Entity Framework Code First 映射关系：一对多和多对多

本章讨论如何使用 Entity Framework Code

第9.1节：映射一对多

假设你有两个不同的实体，类似如下：

```
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
}

public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
}

public class MyDemoContext : DbContext
{
    public DbSet<Person> People { get; set; }
    public DbSet<Car> Cars { get; set; }
}
```

你想在它们之间建立一对多关系，也就是说，一个人可以拥有零个、一个或多个汽车，而一辆汽车恰好属于一个人。每个关系都是双向的，所以如果一个人有一辆车，这辆车就属于那个人。

为此，只需修改你的模型类：

```
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
    public virtual ICollection<Car> Cars { get; set; } // 别忘了初始化 (使用 HashSet)
}

public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public int PersonId { get; set; }
    public virtual Person Person { get; set; }
}
```

就是这样 :) 你已经设置好了关系。在数据库中，这当然是通过外键来表示的。

Chapter 9: Mapping relationship with Entity Framework Code First: One-to-many and Many-to-many

The topic discusses how you can map one-to-many and many-to-many relationships using Entity Framework Code First.

Section 9.1: Mapping one-to-many

So let's say you have two different entities, something like this:

```
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
}

public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
}

public class MyDemoContext : DbContext
{
    public DbSet<Person> People { get; set; }
    public DbSet<Car> Cars { get; set; }
}
```

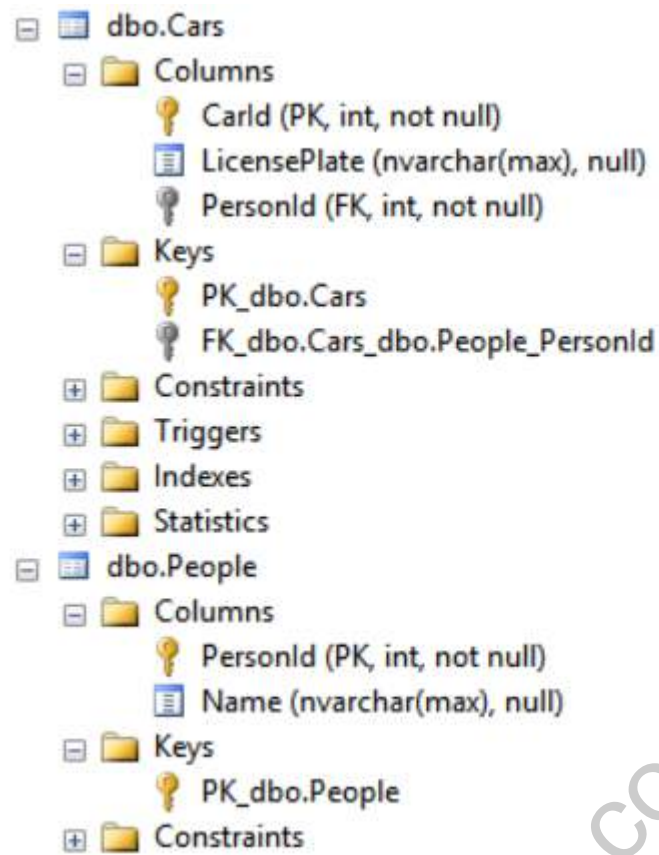
And you want to setup a one-to-many relationship between them, that is, one person can have zero, one or more cars, and one car belongs to one person exactly. Every relationship is bidirectional, so if a person has a car, the car belongs to that person.

To do this just modify your model classes:

```
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
    public virtual ICollection<Car> Cars { get; set; } // don't forget to initialize (use HashSet)
}

public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public int PersonId { get; set; }
    public virtual Person Person { get; set; }
}
```

And that's it :) You already have your relationship set up. In the database, this is represented with foreign keys, of course.



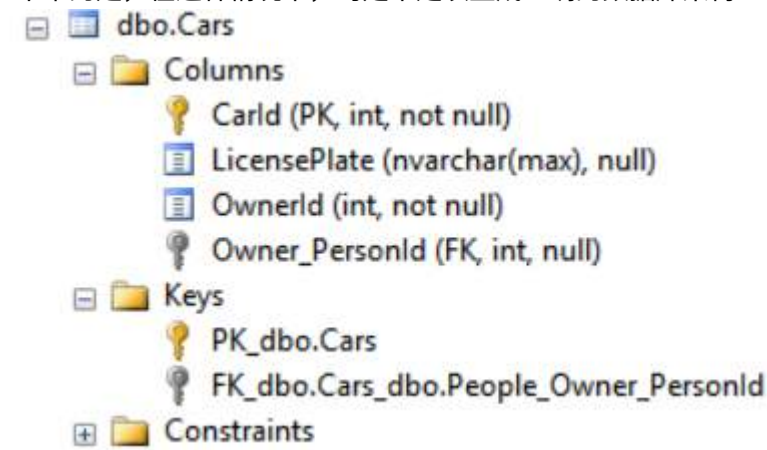
第9.2节：映射一对多：违背约定

在上一个例子中，你可以看到 EF 会自动识别哪个列是外键以及它应该指向哪里。怎么做到的？通过使用约定。拥有一个类型为Person且命名为Person的属性，并且有一个PersonId属性，会让 EF 认为PersonId是外键，并且它指向由类型Person表示的表的主键。

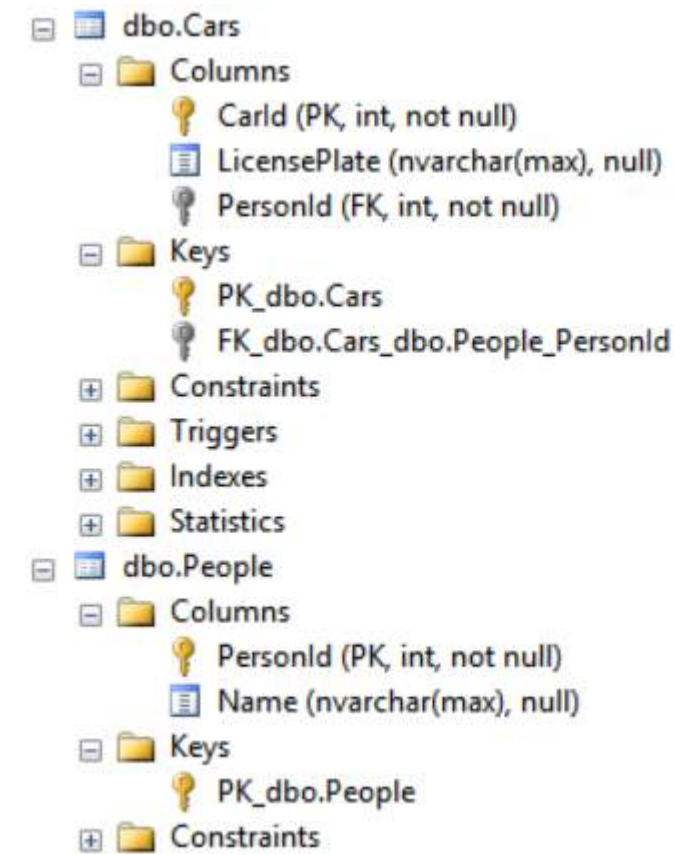
但是如果你把PersonId改成OwnerId，把Person改成Owner，在Car类型中会怎样？

```
public class Car { public int CarId { get; set; } public string LicensePlate { get; set; } public int OwnerId { get; set; } public virtual Person Owner { get; set; } }
```

嗯，不幸的是，在这种情况下，约定不足以生成正确的数据库架构：



别担心；你可以通过在模型中给 EF 一些关于关系和键的提示来帮助它。只需配置你的 Car 类型使用OwnerId属性作为外键。创建一个实体类型配置并在你的 OnModelCreating():



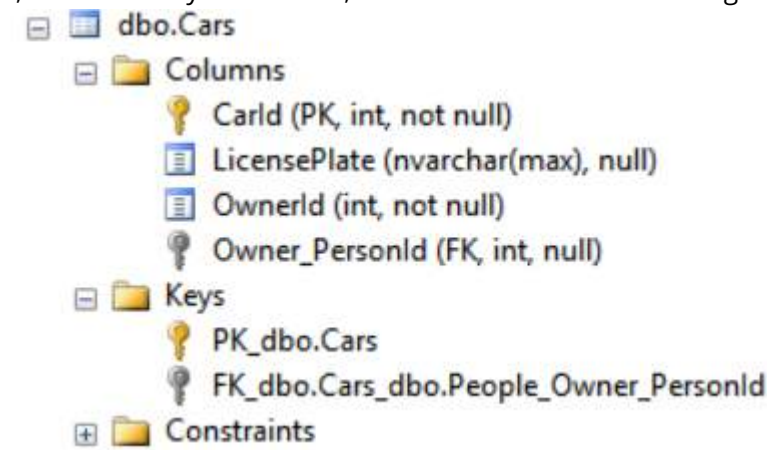
Section 9.2: Mapping one-to-many: against the convention

In the last example, you can see that EF figures out which column is the foreign key and where should it point to. How? By using conventions. Having a property of type Person that is named Person with a PersonId property leads EF to conclude that PersonId is a foreign key, and it points to the primary key of the table represented by the type Person.

But what if you were to change *PersonId* to *OwnerId* and *Person* to *Owner* in the **Car** type?

```
public class Car { public int CarId { get; set; } public string LicensePlate { get; set; } public int OwnerId { get; set; } public virtual Person Owner { get; set; } }
```

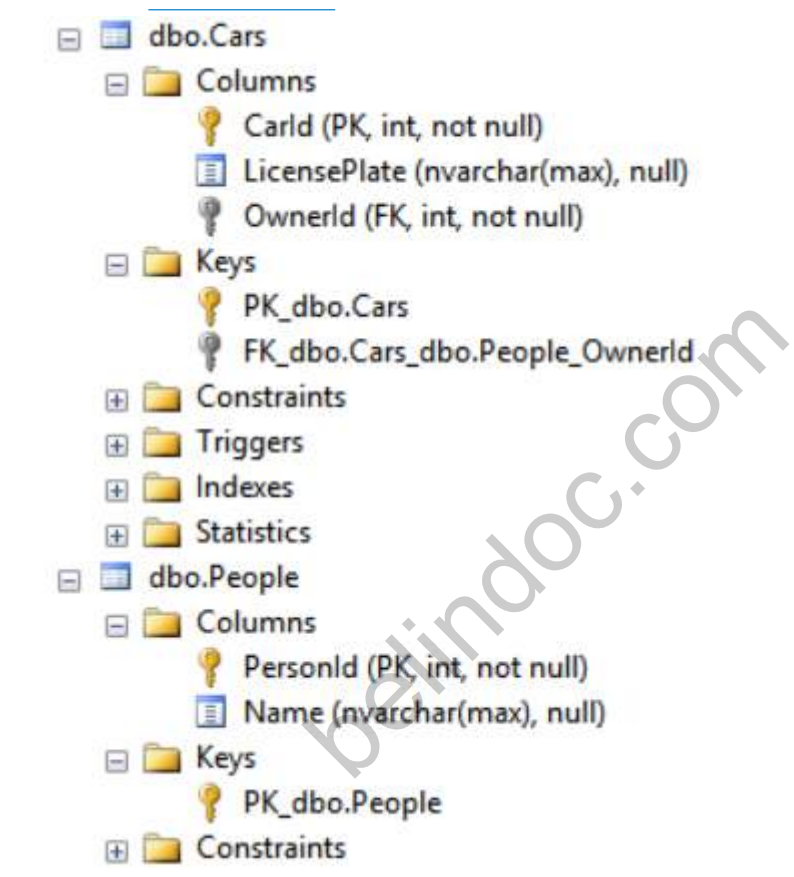
Well, unfortunately in this case, the conventions are not enough to produce the correct DB schema:



No worries; you can help EF with some hints about your relationships and keys in the model. Simply configure your Car type to use the OwnerId property as the FK. Create an entity type configuration and apply it in your OnModelCreating():

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
    public CarEntityTypeConfiguration()
    {
        this.HasRequired(c => c.Owner).WithMany(p => p.Cars).HasForeignKey(c => c.OwnerId);
    }
}
```

这基本上表示Car有一个必需的属性Owner（HasRequired()），在Owner类型中，Cars属性用于引用回汽车实体（WithMany()）。最后指定了表示外键的属性（HasForeignKey()）。这给出了我们想要的模式：



你也可以从Person端配置这个关系：

```
public class PersonEntityTypeConfiguration : EntityTypeConfiguration<Person>
{
    public PersonEntityTypeConfiguration()
    {
        this.HasMany(p => p.Cars).WithRequired(c => c.Owner).HasForeignKey(c => c.OwnerId);
    }
}
```

思路是一样的，只是两端不同（注意你可以这样理解整个表达：“这个人有多辆车，每辆车都有一个必需的车主”）。无论你是从Person端还是Car端配置关系都没关系。你甚至可以同时包含两者，但在这种情况下要小心确保两端指定的是同一个关系！

第9.3节：映射零或一对多关系

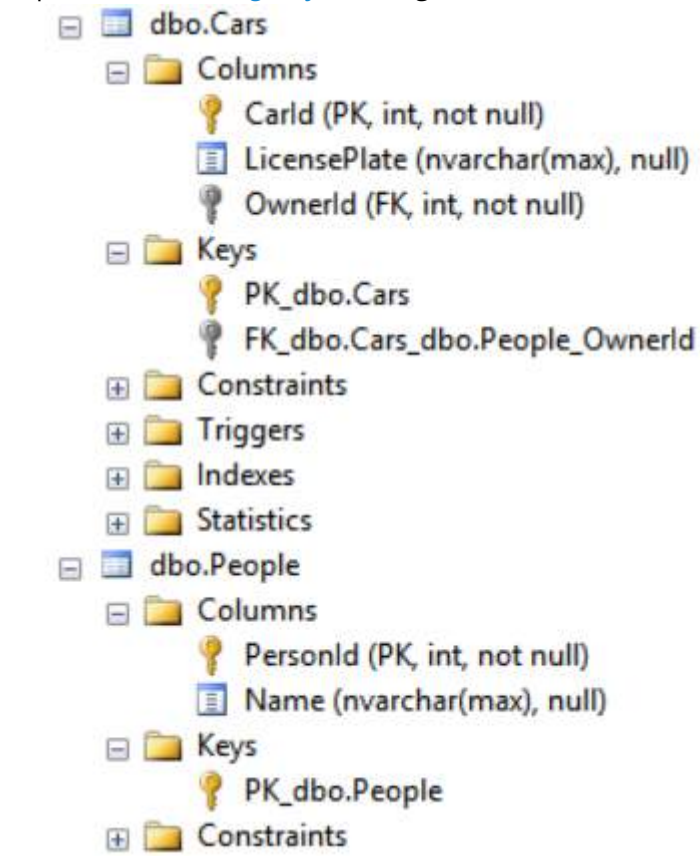
在之前的例子中，汽车不能没有人。如果你想让汽车端的人是可选的呢？这其实很简单，知道如何做一对多关系即可。只需将Car中的PersonId改为可空：

```
public class Car
{

```

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
    public CarEntityTypeConfiguration()
    {
        this.HasRequired(c => c.Owner).WithMany(p => p.Cars).HasForeignKey(c => c.OwnerId);
    }
}
```

This basically says that Car has a required property, Owner ([HasRequired\(\)](#)) and in the type of Owner, the Cars property is used to refer back to the car entities ([WithMany\(\)](#)). And finally the property representing the foreign key is specified ([HasForeignKey\(\)](#)). This gives us the schema we want:



You could configure the relationship from the Person side as well:

```
public class PersonEntityTypeConfiguration : EntityTypeConfiguration<Person>
{
    public PersonEntityTypeConfiguration()
    {
        this.HasMany(p => p.Cars).WithRequired(c => c.Owner).HasForeignKey(c => c.OwnerId);
    }
}
```

The idea is the same, just the sides are different (note how you can read the whole thing: 'this person has many cars, each car with a required owner'). Doesn't matter if you configure the relationship from the Person side or the Car side. You can even include both, but in this case be careful to specify the same relationship on both sides!

Section 9.3: Mapping zero or one-to-many

In the previous examples a car cannot exist without a person. What if you wanted the person to be optional from the car side? Well, it's kind of easy, knowing how to do one-to-many. Just change the PersonId in Car to be nullable:

```
public class Car
{

```

```

public int CarId { get; set; }
public string LicensePlate { get; set; }
public int? PersonId { get; set; }
public virtual Person Person { get; set; }
}

```

然后使用 `HasOptional()` (或根据配置的端不同使用 `WithOptional()`) :

```

public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
    public CarEntityTypeConfiguration()
    {
        this.HasOptional(c => c.Owner).WithMany(p => p.Cars).HasForeignKey(c => c.OwnerId);
    }
}

```

第9.4节：多对多

让我们继续讨论另一种情况，每个人可以拥有多辆车，每辆车也可以有多个车主（但关系仍然是双向的）。这是一种多对多关系。最简单的方法是让EF通过约定来完成它的魔法。

只需这样更改模型：

```

public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
    public virtual ICollection<Car> Cars { get; set; }
}

public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public virtual ICollection<Person> Owners { get; set; }
}

```

以及架构：

```

public int CarId { get; set; }
public string LicensePlate { get; set; }
public int? PersonId { get; set; }
public virtual Person Person { get; set; }
}

```

And then use the [HasOptional\(\)](#) (or [WithOptional\(\)](#), depending from which side you do the configuration):

```

public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
    public CarEntityTypeConfiguration()
    {
        this.HasOptional(c => c.Owner).WithMany(p => p.Cars).HasForeignKey(c => c.OwnerId);
    }
}

```

Section 9.4: Many-to-many

Let's move on to the other scenario, where every person can have multiple cars and every car can have multiple owners (but again, the relationship is bidirectional). This is a many-to-many relationship. The easiest way is to let EF do it's magic using conventions.

Just change the model like this:

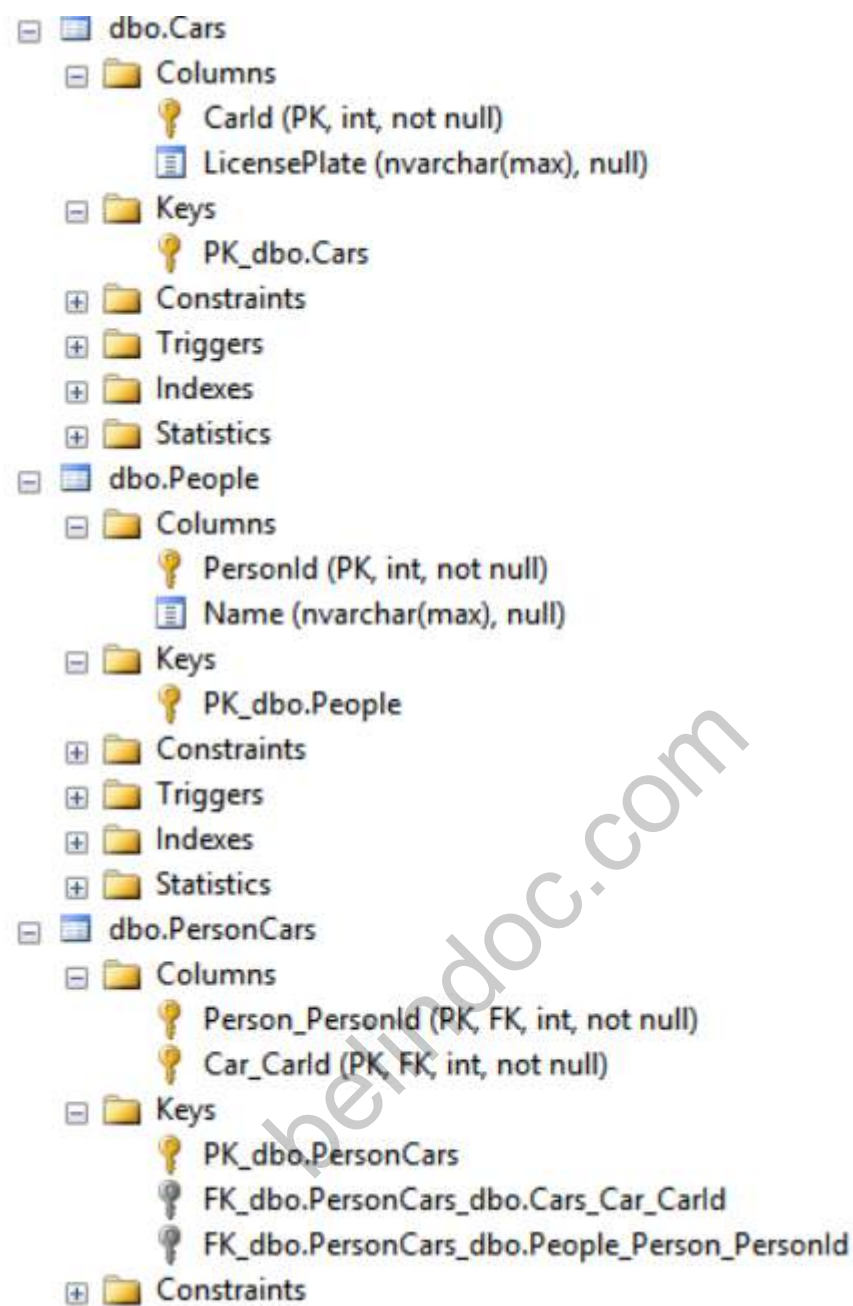
```

public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
    public virtual ICollection<Car> Cars { get; set; }
}

public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public virtual ICollection<Person> Owners { get; set; }
}

```

And the schema:



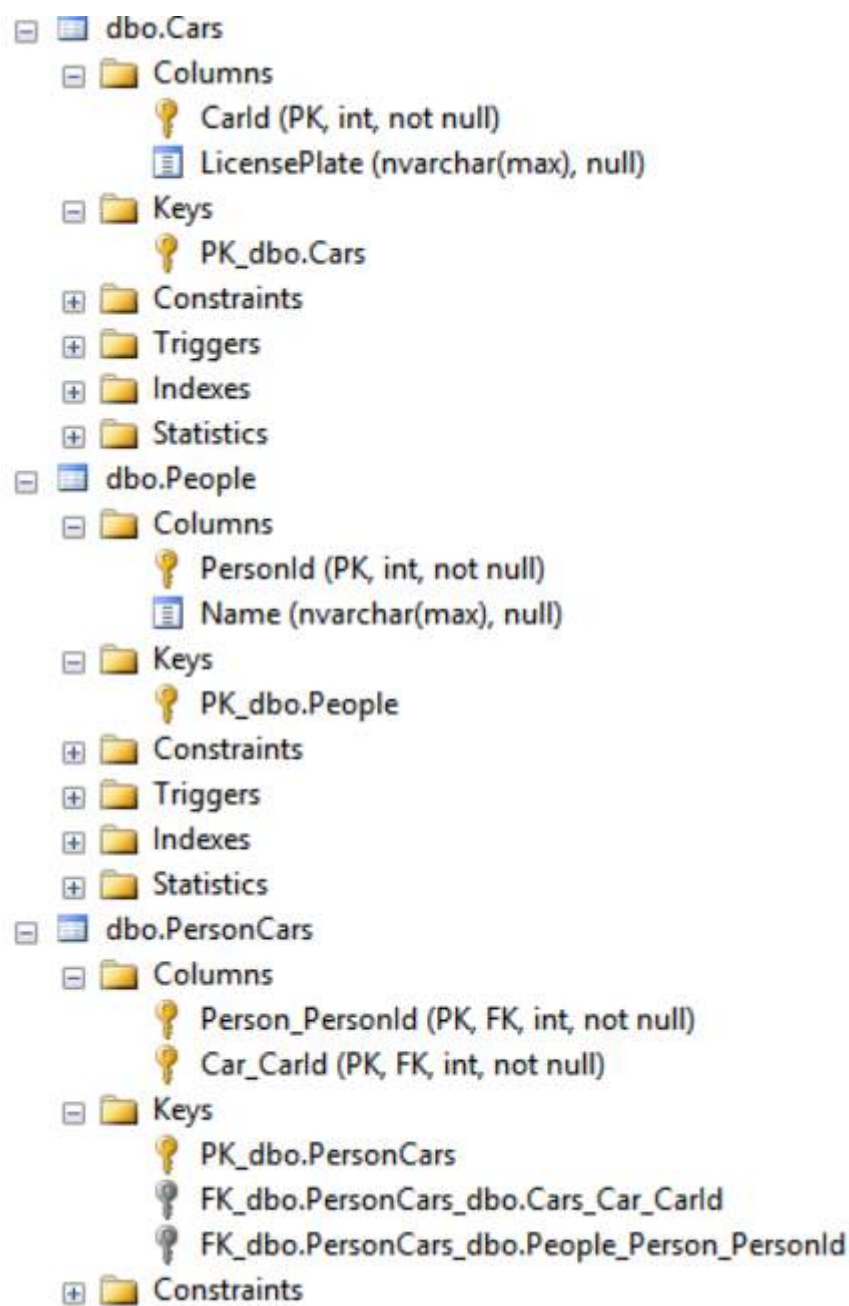
几乎完美。如你所见，EF

识别出了需要一个连接表，用来跟踪人和车的配对关系。

第9.5节：多对多：自定义连接表

你可能想给连接表中的字段重命名，使其更友好。你可以使用常规的配置方法来实现（同样，无论从哪一方进行配置都没关系）：

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
    public CarEntityTypeConfiguration()
    {
        this.HasMany(c => c.Owners).WithMany(p => p.Cars)
            .Map(m =>
            {
                m.MapLeftKey("OwnerId");
                m.MapRightKey("CarId");
                m.ToTable("PersonCars");
            });
    }
}
```



Almost perfect. As you can see, EF

recognized the need for a join table, where you can keep track of person-car pairings.

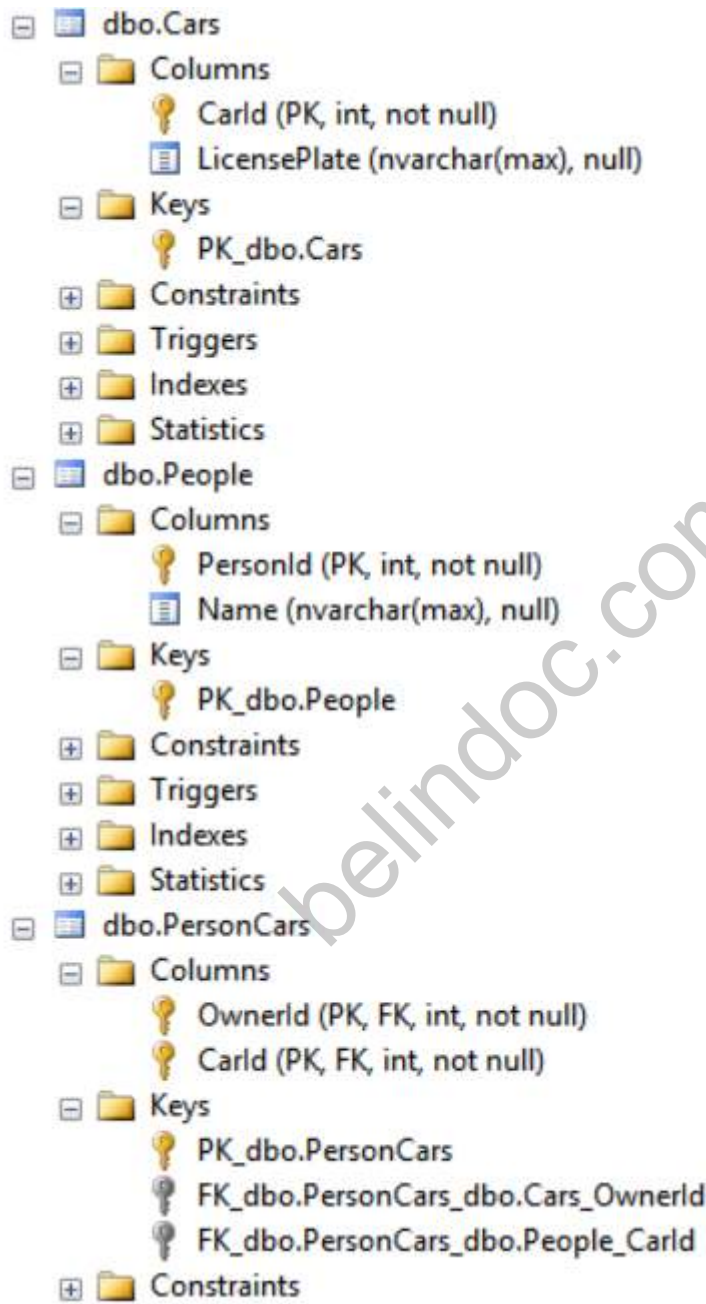
Section 9.5: Many-to-many: customizing the join table

You might want to rename the fields in the join table to be a little more friendly. You can do this by using the usual configuration methods (again, it doesn't matter which side you do the configuration from):

```
public class CarEntityTypeConfiguration : EntityTypeConfiguration<Car>
{
    public CarEntityTypeConfiguration()
    {
        this.HasMany(c => c.Owners).WithMany(p => p.Cars)
            .Map(m =>
            {
                m.MapLeftKey("OwnerId");
                m.MapRightKey("CarId");
                m.ToTable("PersonCars");
            });
    }
}
```

}

即使是这样也很容易理解：这辆车有很多车主（`HasMany()`），每个车主也有很多车（`WithMany()`）。将其映射，使左键映射到 `OwnerId`（`MapLeftKey()`），右键映射到 `CarId`（`MapRightKey()`），整个映射到表 `PersonCars`（`ToTable()`）。这正好给你这样的模式：



第9.6节：多对多：自定义连接实体

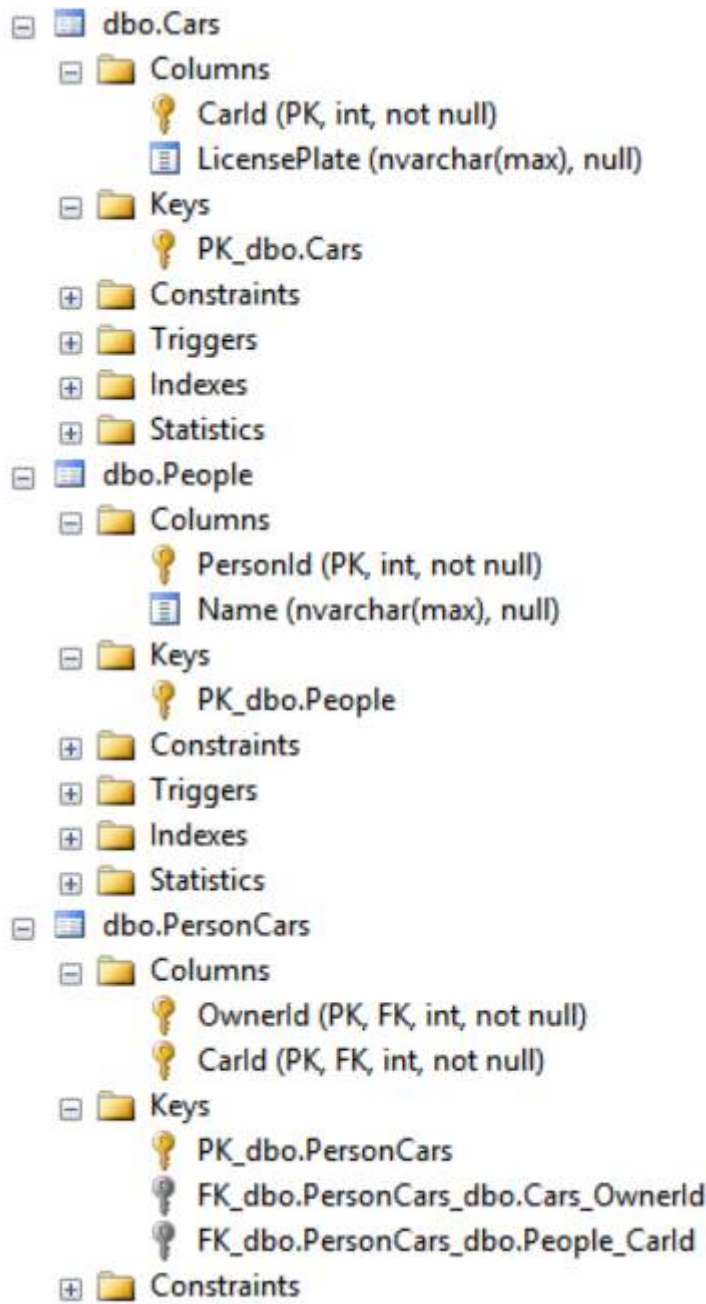
我必须承认，我并不喜欢让 EF 在没有连接实体的情况下推断连接表。你无法跟踪人与车关联的额外信息（比如有效日期），因为你无法修改该表。

此外，连接表中的 `CarId` 是主键的一部分，所以如果家庭买了新车，你必须先删除旧的关联再添加新的。EF 会帮你隐藏这些，但这意味着你必须执行这两个操作，而不是简单的更新（更不用说频繁的插入/删除可能导致索引碎片—好在有简单的解决方法）。

在这种情况下，你可以创建一个连接实体，该实体引用一个特定的汽车和一个特定的人。基本上，你将多对多关联视为两个一对多关联的组合：

}

Quite easy to read even: this car has many owners (`HasMany()`), with each owner having many cars (`WithMany()`). Map this so that you map the left key to `OwnerId` (`MapLeftKey()`), the right key to `CarId` (`MapRightKey()`) and the whole thing to the table `PersonCars` (`ToTable()`). And this gives you exactly that schema:



Section 9.6: Many-to-many: custom join entity

I have to admit, I'm not really a fan of letting EF infer the join table without a join entity. You cannot track extra information to a person-car association (let's say the date from which it is valid), because you can't modify the table.

Also, the `CarId` in the join table is part of the primary key, so if the family buys a new car, you have to first delete the old associations and add new ones. EF hides this from you, but this means that you have to do these two operations instead of a simple update (not to mention that frequent inserts/deletes might lead to index fragmentation — good thing [there is an easy fix](#) for that).

In this case what you can do is create a join entity that has a reference to both one specific car and one specific person. Basically you look at your many-to-many association as a combinations of two one-to-many associations:

```

public class PersonToCar
{
    public int PersonToCarId { get; set; }
    public int CarId { get; set; }
    public virtual Car Car { get; set; }
    public int PersonId { get; set; }
    public virtual Person Person { get; set; }
    public DateTime ValidFrom { get; set; }
}

public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
    public virtual ICollection<PersonToCar> CarOwnerShips { get; set; }
}

public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public virtual ICollection<PersonToCar> Ownerships { get; set; }
}

public class MyDemoContext : DbContext
{
    public DbSet<Person> People { get; set; }
    public DbSet<Car> Cars { get; set; }
    public DbSet<PersonToCar> PersonToCars { get; set; }
}

```

这让我拥有了更多的控制权，也更加灵活。我现在可以向关联中添加自定义数据，并且每个关联都有自己的主键，因此我可以更新其中的汽车或车主引用。

```

public class PersonToCar
{
    public int PersonToCarId { get; set; }
    public int CarId { get; set; }
    public virtual Car Car { get; set; }
    public int PersonId { get; set; }
    public virtual Person Person { get; set; }
    public DateTime ValidFrom { get; set; }
}

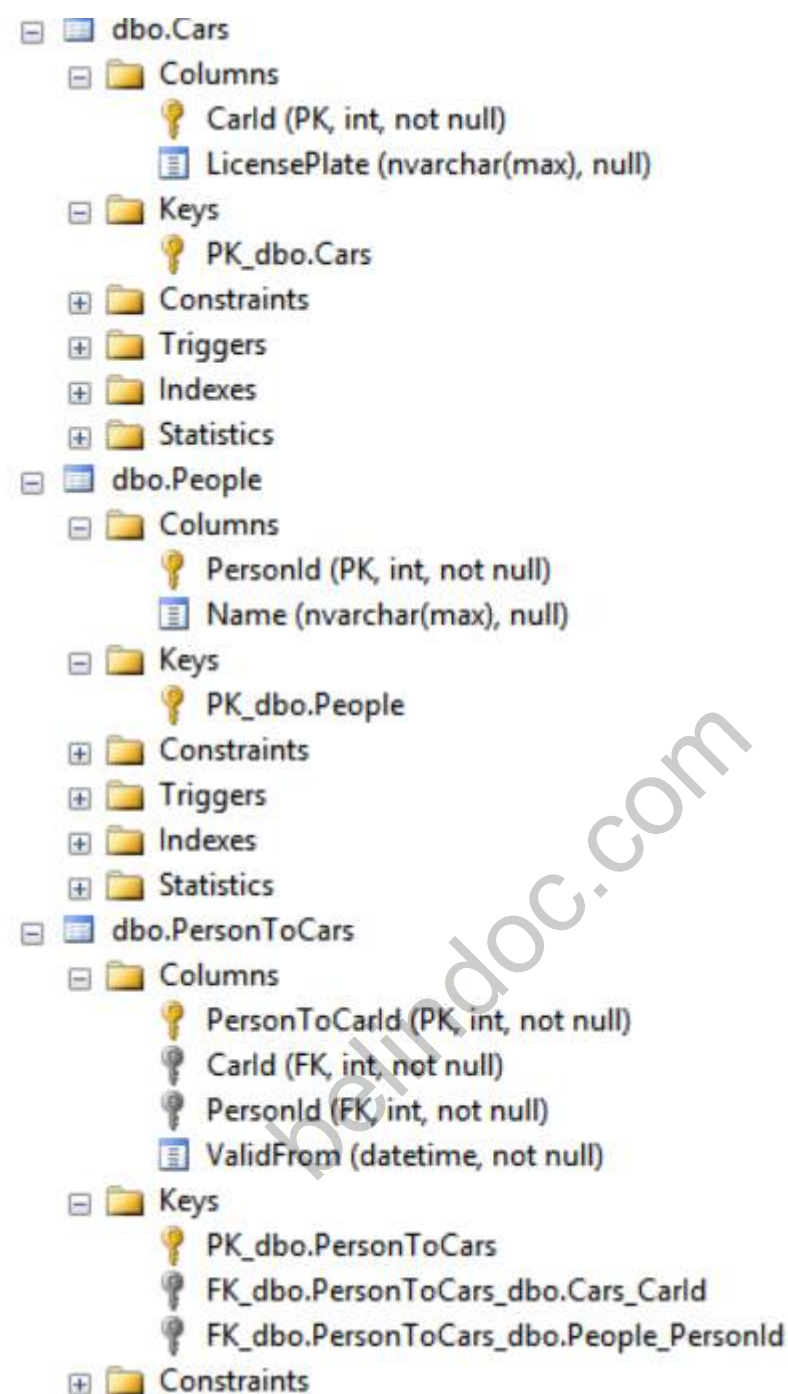
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
    public virtual ICollection<PersonToCar> CarOwnerShips { get; set; }
}

public class Car
{
    public int CarId { get; set; }
    public string LicensePlate { get; set; }
    public virtual ICollection<PersonToCar> Ownerships { get; set; }
}

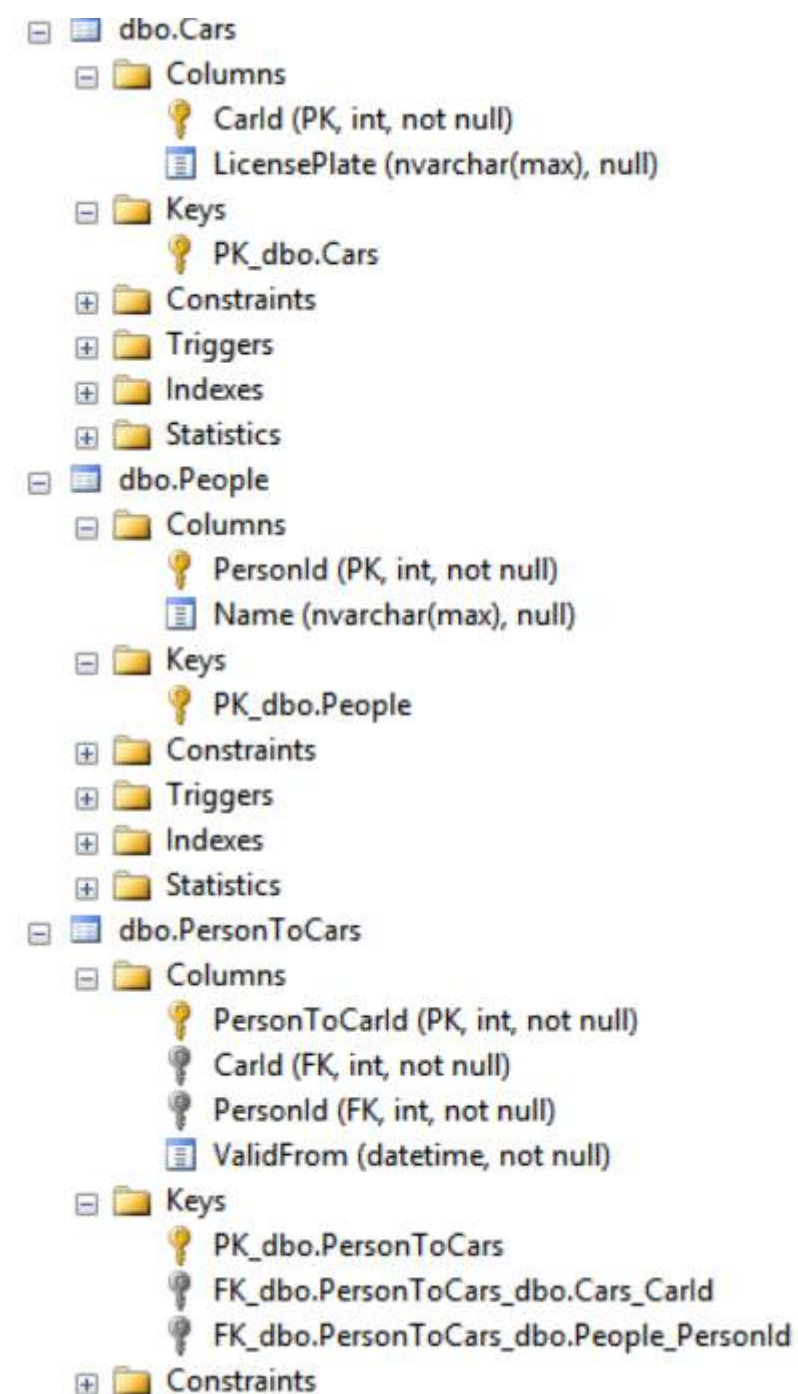
public class MyDemoContext : DbContext
{
    public DbSet<Person> People { get; set; }
    public DbSet<Car> Cars { get; set; }
    public DbSet<PersonToCar> PersonToCars { get; set; }
}

```

This gives me much more control and it's a lot more flexible. I can now add custom data to the association and every association has its own primary key, so I can update the car or the owner reference in them.



请注意，这实际上只是两个一对多关系的组合，因此你可以使用前面示例中讨论的所有配置选项。

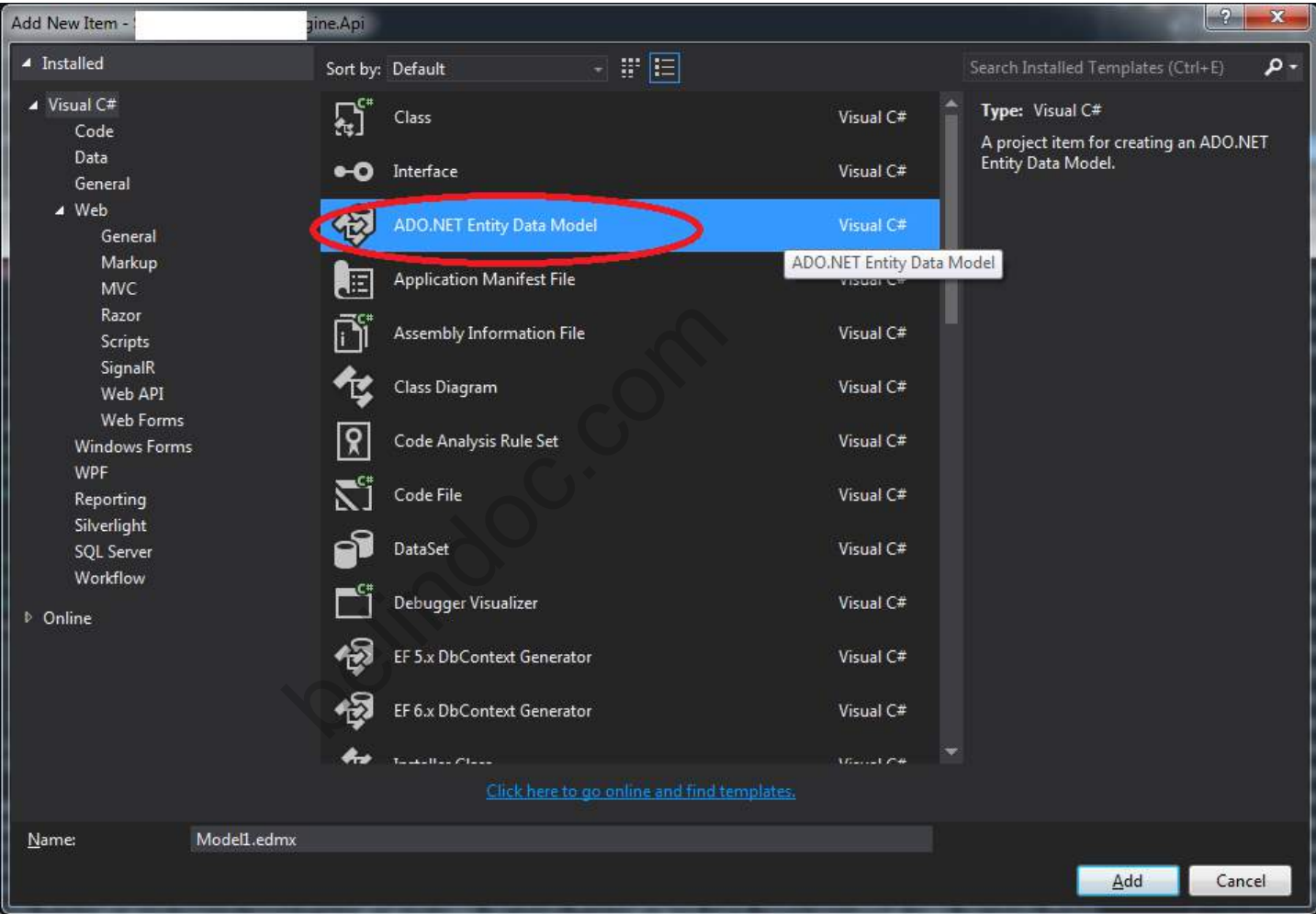


Note that this really is just a combination of two one-to-many relationships, so you can use all the configuration options discussed in the previous examples.

第10章：数据库优先模型生成

第10.1节：从数据库生成模型

在Visual Studio中，进入你的Solution Explorer，然后点击Project，你将添加模型 右键鼠标。
选择ADO.NET实体数据模型

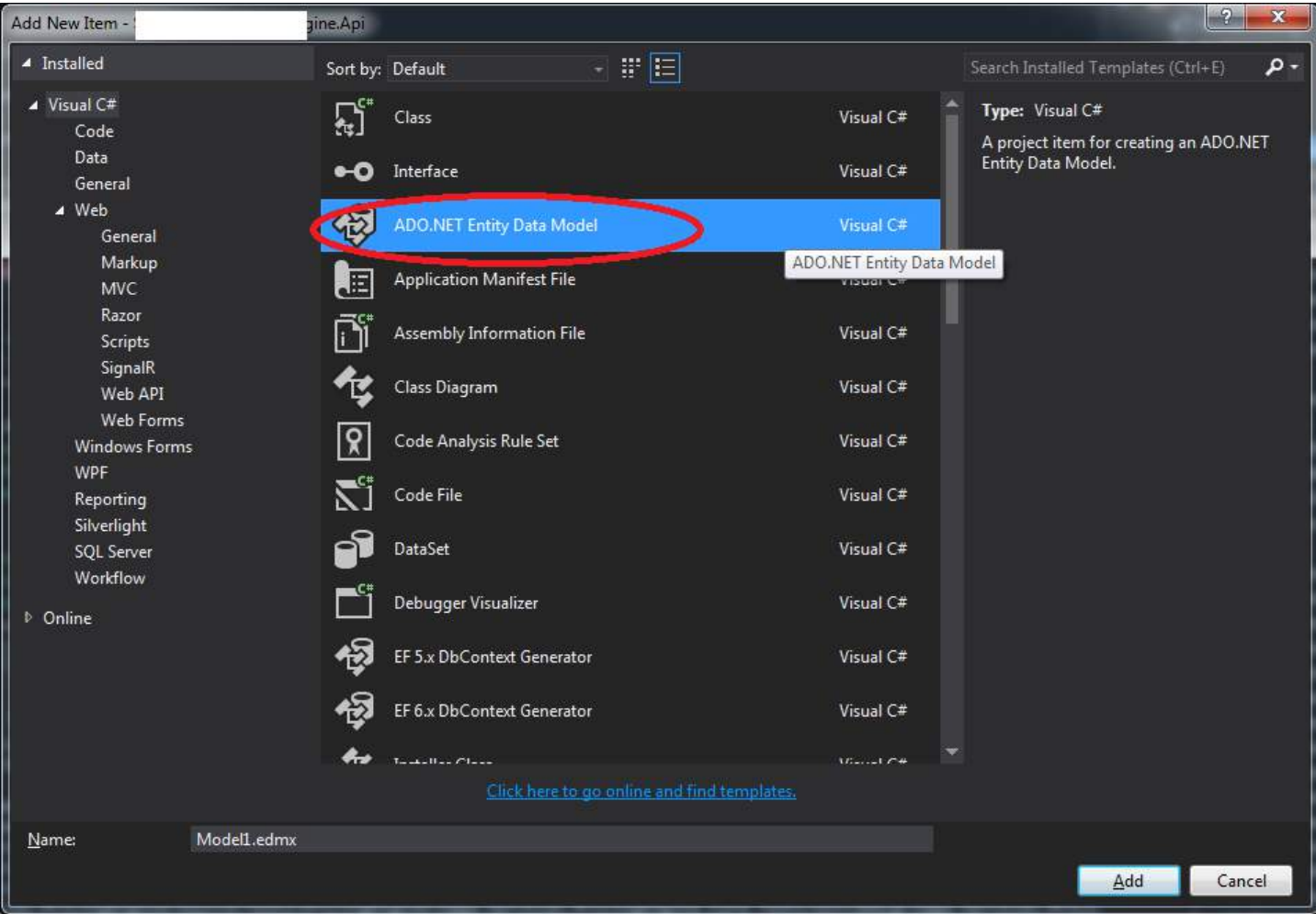


然后选择从数据库生成并点击下一步，在下一个窗口点击新建连接...并指向你想要生成模型的数据库（可以是MSSQL、MySQL或Oracle）

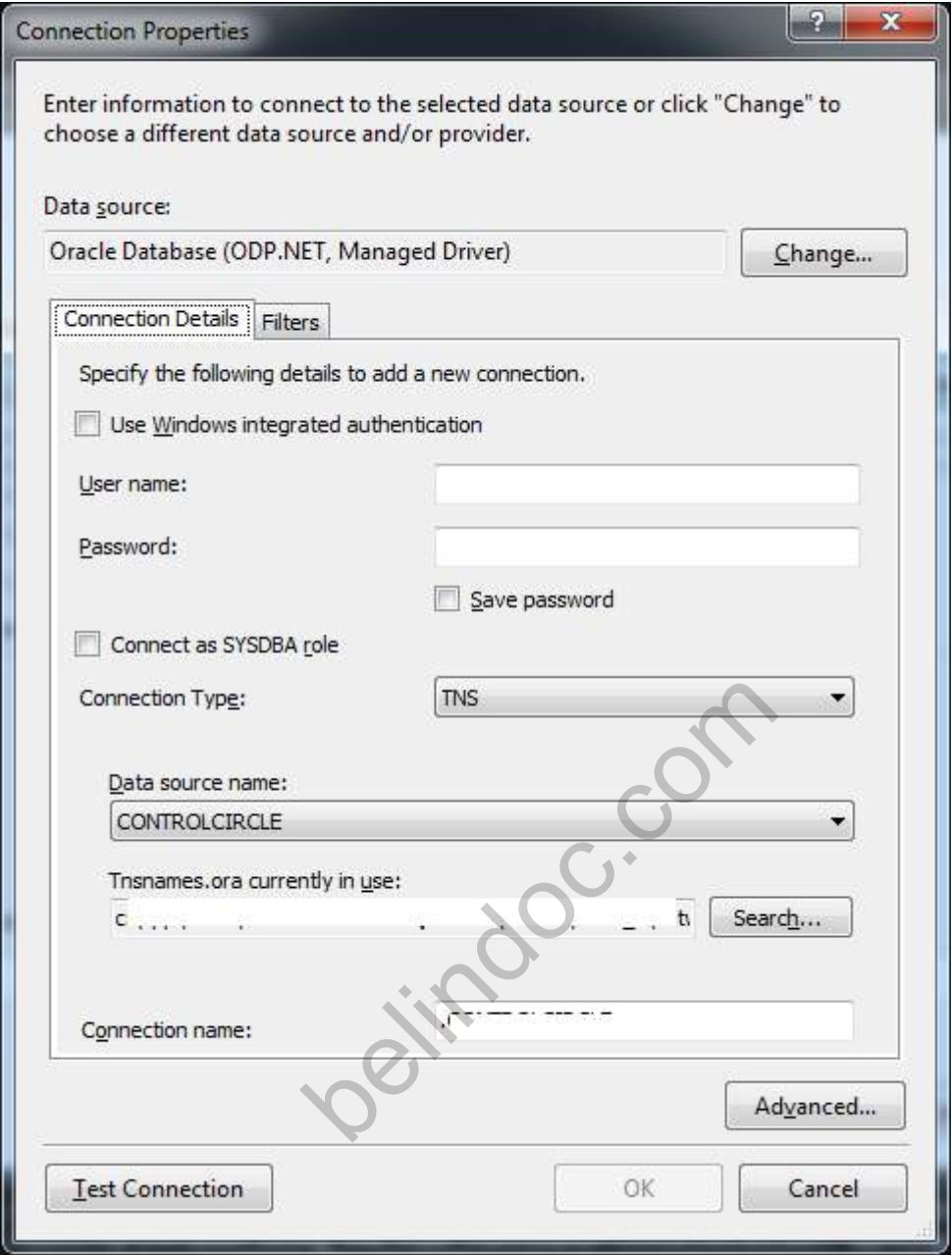
Chapter 10: Database first model generation

Section 10.1: Generating model from database

In Visual Studio go to your Solution Explorer then click on Project you will be adding model Right mouse.
Choose ADO.NET Entity Data Model



Then choose Generate **from** database and click Next in next window click **New** Connection... and point to the database you want to generate model from (Could be MSSQL, MySQL or Oracle)



完成后点击测试连接以检查连接是否配置正确（如果这里失败，请不要继续操作）。

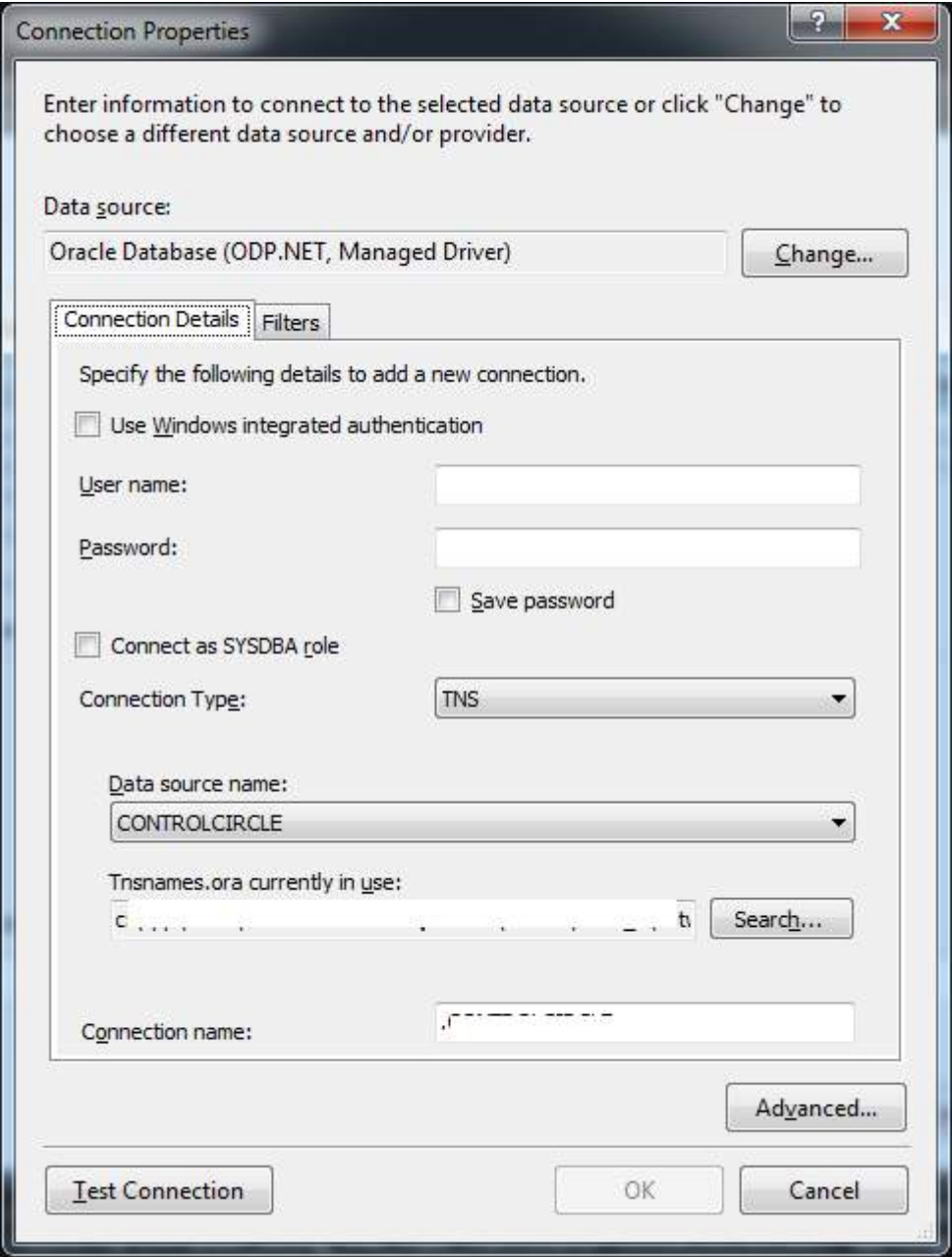
点击下一步，然后选择你想要的选项（例如生成实体名称的样式或添加外键）。

再次点击下一步，此时你应该已经从数据库生成了模型。

第10.2节：向生成的模型添加数据注解

在Entity Framework 5及更高版本使用的T4代码生成策略中，默认不包含数据注解属性。要在每次模型重新生成时在某些属性上包含数据注解，请打开EDMX附带的模板文件（扩展名为.tt），然后在UsingDirectives方法下添加如下using语句：

```
foreach (var entity in typeMapper.GetItemsToGenerate<EntityType>
(itemCollection))
{
    fileManager.StartNewFile(entity.Name + ".cs");
    BeginNamespace(code);
#>
<#=#codeStringGenerator.UsingDirectives(inHeader: false)#>
```



After you done this click Test Connection to see if you have configured connection properly (do not go any further if it fails here).

Click Next then choose options that you want (like style for generating entity names or to add foreign keys).

Click Next again, at this point you should have model generated from database.

Section 10.2: Adding data annotations to the generated model

In T4 code-generation strategy used by Entity Framework 5 and higher, data annotation attributes are not included by default. To include data annotations on top of certain property every model regeneration, open template file included with EDMX (with .tt extension) then add a using statement under UsingDirectives method like below:

```
foreach (var entity in typeMapper.GetItemsToGenerate<EntityType>
(itemCollection))
{
    fileManager.StartNewFile(entity.Name + ".cs");
    BeginNamespace(code);
#>
<#=#codeStringGenerator.UsingDirectives(inHeader: false)#>
```

```
using System.ComponentModel.DataAnnotations; // --> 添加此行
```

举例来说，假设模板应包含表示主键属性的 `KeyAttribute`。要在重新生成模型时自动插入 `KeyAttribute`，找到包含 `codeStringGenerator.Property` 的代码部分，如下所示：

```
var simpleProperties = typeMapper.GetSimpleProperties(entity);
if (simpleProperties.Any())
{
    foreach (var edmProperty in simpleProperties)
    {
#>
<#codeStringGenerator.Property(edmProperty)#>
<#
    }
}
```

然后，插入一个 `if` 条件来检查主键属性，如下所示：

```
var simpleProperties = typeMapper.GetSimpleProperties(entity);
if (simpleProperties.Any())
{
    foreach (var edmProperty in simpleProperties)
    {
        if (ef.IsKey(edmProperty)) {
#> [Key]
<# }
#>
<#codeStringGenerator.Property(edmProperty)#>
<#
    }
}
```

通过应用上述更改，所有生成的模型类在从数据库更新模型后，其主键属性上都会带有 `KeyAttribute`。

之前

```
using System;

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

之后

```
using System;
using System.ComponentModel.DataAnnotations;

public class Example
{
    [Key]
    public int Id { get; set; }
    public string Name { get; set; }
}
```

```
using System.ComponentModel.DataAnnotations; // --> add this line
```

As an example, suppose the template should include `KeyAttribute` which indicates a primary key property. To insert `KeyAttribute` automatically while regenerating model, find part of code containing `codeStringGenerator.Property` as below:

```
var simpleProperties = typeMapper.GetSimpleProperties(entity);
if (simpleProperties.Any())
{
    foreach (var edmProperty in simpleProperties)
    {
#>
<#codeStringGenerator.Property(edmProperty)#>
<#
    }
}
```

Then, insert an `if`-condition to check key property as this:

```
var simpleProperties = typeMapper.GetSimpleProperties(entity);
if (simpleProperties.Any())
{
    foreach (var edmProperty in simpleProperties)
    {
        if (ef.IsKey(edmProperty)) {
#> [Key]
<# }
#>
<#codeStringGenerator.Property(edmProperty)#>
<#
    }
}
```

By applying changes above, all generated model classes will have `KeyAttribute` on their primary key property after updating model from database.

Before

```
using System;

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

After

```
using System;
using System.ComponentModel.DataAnnotations;

public class Example
{
    [Key]
    public int Id { get; set; }
    public string Name { get; set; }
}
```

第11章：复杂类型

第11.1节：代码优先复杂类型

复杂类型允许你将数据库表的选定字段映射为主类型的子类型。

```
[ComplexType]
public class Address
{
    public string Street { get; set; }
    public string Street_2 { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string ZipCode { get; set; }
}
```

该复杂类型随后可以在多个实体类型中使用。它甚至可以在同一个实体类型中使用多次。

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

    ...
    public Address ShippingAddress { get; set; }
    public Address BillingAddress { get; set; }
}
```

该实体类型随后将存储在数据库中的一个表中，表结构大致如下。

🔑 Id (PK, int, not null)
📄 Name (varchar(max), null)
📄 ShippingAddress_Street (varchar(max), null)
📄 ShippingAddress_Street_2 (varchar(max), null)
📄 ShippingAddress_City (varchar(max), null)
📄 ShippingAddress_State (varchar(max), null)
📄 ShippingAddress_ZipCode (varchar(max), null)
📄 BillingAddress_Street (varchar(max), null)
📄 BillingAddress_Street_2 (varchar(max), null)
📄 BillingAddress_City (varchar(max), null)
📄 BillingAddress_State (varchar(max), null)
📄 BillingAddress_ZipCode (varchar(max), null)

当然，在这种情况下，1:n 关联（客户-地址）将是首选模型，但该示例展示了复杂类型的使用方法。

Chapter 11: Complex Types

Section 11.1: Code First Complex Types

A complex type allows you to map selected fields of a database table into a single type that is a child of the main type.

```
[ComplexType]
public class Address
{
    public string Street { get; set; }
    public string Street_2 { get; set; }
    public string City { get; set; }
    public string State { get; set; }
    public string ZipCode { get; set; }
}
```

This complex type can then be used in multiple entity types. It can even be used more than once in the same entity type.

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

    ...
    public Address ShippingAddress { get; set; }
    public Address BillingAddress { get; set; }
}
```

This entity type would then be stored in a table in the database that would look something like this.

🔑 Id (PK, int, not null)
📄 Name (varchar(max), null)
📄 ShippingAddress_Street (varchar(max), null)
📄 ShippingAddress_Street_2 (varchar(max), null)
📄 ShippingAddress_City (varchar(max), null)
📄 ShippingAddress_State (varchar(max), null)
📄 ShippingAddress_ZipCode (varchar(max), null)
📄 BillingAddress_Street (varchar(max), null)
📄 BillingAddress_Street_2 (varchar(max), null)
📄 BillingAddress_City (varchar(max), null)
📄 BillingAddress_State (varchar(max), null)
📄 BillingAddress_ZipCode (varchar(max), null)

Of course, in this case, a 1:n association (Customer-Address) would be the preferred model, but the example shows how complex types can be used.

第12章：数据库初始化器

第12.1节：CreateDatabaseIfNotExists

实现了IDatabaseInitializer接口，该接口是EntityFramework默认使用的。顾名思义，它会在数据库不存在时创建数据库。但当你更改模型时，它会抛出异常。

用法：

```
public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(new CreateDatabaseIfNotExists<MyContext>());
    }
}
```

第12.2节：DropCreateDatabaseIfModelChanges

此 IDatabaseInitializer 的实现会在模型更改时自动删除并重新创建数据库。

用法：

```
public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(new DropCreateDatabaseIfModelChanges<MyContext>());
    }
}
```

第12.3节：DropCreateDatabaseAlways

此 IDatabaseInitializer 的实现会在每次应用程序域中使用上下文时删除并重新创建数据库。请注意，由于数据库被重新创建，可能会导致数据丢失。

用法：

```
public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(new DropCreateDatabaseAlways<MyContext>());
    }
}
```

第12.4节：自定义数据库初始化器

您可以创建自己的 IDatabaseInitializer 实现。

初始化器的示例实现，它将数据库迁移到版本0，然后迁移到最新的迁移版本（例如在运行集成测试时非常有用）。为此，你还需要一个 DbMigrationsConfiguration 类型。

```
public class RecreateFromScratch<TContext, TMigrationsConfiguration> :
    IDatabaseInitializer<TContext>
where TContext : DbContext
where TMigrationsConfiguration : DbMigrationsConfiguration<TContext>, new()
{
    private readonly DbMigrationsConfiguration<TContext> _configuration;
```

Chapter 12: Database Initialisers

Section 12.1: CreateDatabaseIfNotExists

Implementation of IDatabaseInitializer that is used in EntityFramework by default. As the name implies, it creates the database if none exists. However when you change the model, it throws an exception.

Usage:

```
public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(new CreateDatabaseIfNotExists<MyContext>());
    }
}
```

Section 12.2: DropCreateDatabaseIfModelChanges

This implementation of IDatabaseInitializer drops and recreates the database if the model changes automatically.

Usage:

```
public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(new DropCreateDatabaseIfModelChanges<MyContext>());
    }
}
```

Section 12.3: DropCreateDatabaseAlways

This implementation of IDatabaseInitializer drops and recreates the database every time your context is used in applications app domain. Beware of the data loss due to the fact, that the database is recreated.

Usage:

```
public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(new DropCreateDatabaseAlways<MyContext>());
    }
}
```

Section 12.4: Custom database initializer

You can create your own implementation of IDatabaseInitializer.

Example implementation of an initializer, that will migrate the database to 0 and then migrate all the way to the newest migration (usefull e.g. when running integration tests). In order to do that you would need a DbMigrationsConfiguration type too.

```
public class RecreateFromScratch<TContext, TMigrationsConfiguration> :
    IDatabaseInitializer<TContext>
where TContext : DbContext
where TMigrationsConfiguration : DbMigrationsConfiguration<TContext>, new()
{
    private readonly DbMigrationsConfiguration<TContext> _configuration;
```

```

    public RecreateFromScratch()
    {
        _configuration = new TMigrationsConfiguration();
    }

    public void InitializeDatabase(TContext context)
    {
        var migrator = new DbMigrator(_configuration);
        migrator.Update("0");
        migrator.Update();
    }
}

```

第12.5节：将数据库迁移到最新版本

一个实现了IDatabaseInitializer接口的类，它将使用Code First迁移来更新数据库到最新版本。要使用此初始化程序，您还必须使用DbMigrationsConfiguration类型。

用法：

```

public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(
            new MigrateDatabaseToLatestVersion<MyContext, Configuration>());
    }
}

```

```

    public RecreateFromScratch()
    {
        _configuration = new TMigrationsConfiguration();
    }

    public void InitializeDatabase(TContext context)
    {
        var migrator = new DbMigrator(_configuration);
        migrator.Update("0");
        migrator.Update();
    }
}

```

Section 12.5: MigrateDatabaseToLatestVersion

An implementation of IDatabaseInitializer that will use Code First Migrations to update the database to the latest version. To use this initializer you have to use DbMigrationsConfiguration type too.

Usage:

```

public class MyContext : DbContext {
    public MyContext() {
        Database.SetInitializer(
            new MigrateDatabaseToLatestVersion<MyContext, Configuration>());
    }
}

```


第13章：跟踪与无跟踪

第13.1节：无跟踪查询

- 无跟踪查询在结果用于只读场景时非常有用
- 它们执行速度更快，因为不需要设置变更跟踪信息

示例：

```
using (var context = new BookContext())
{
    var books = context.Books.AsNoTracking().ToList();
}
```

使用 EF Core 1.0, 您还可以在上下文实例级别更改默认的跟踪行为。

示例：

```
using (var context = new BookContext())
{
    context.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;

    var books = context.Books.ToList();
}
```

第13.2节：跟踪查询

- 默认情况下，返回实体类型的查询是**跟踪的**
- 这意味着您可以对这些实体实例进行更改，并通过 SaveChanges()

示例：

- 对book评分的更改将在SaveChanges()期间被检测并保存到数据库中。

```
using (var context = new BookContext())
{
    var book = context.Books.FirstOrDefault(b => b.BookId == 1);
    book.Rating = 5;
    context.SaveChanges();
}
```

第13.3节：跟踪与投影

- 即使查询的结果类型不是实体类型，如果结果包含实体类型，它们仍将默认被跟踪

示例：

- 在以下查询中，返回一个匿名类型，结果集中Book的实例将被跟踪

```
using (var context = new BookContext())
{
    var book = context.Books.Select(b => new { Book = b, Authors = b.Authors.Count() });
}
```

Chapter 13: Tracking vs. No-Tracking

Section 13.1: No-tracking queries

- No tracking queries are useful when the results are used in a read-only scenario
- They are quicker to execute because there is no need to setup change tracking information

Example：

```
using (var context = new BookContext())
{
    var books = context.Books.AsNoTracking().ToList();
}
```

With EF Core 1.0 you are also able to change the default tracking behavior at the context instance level.

Example：

```
using (var context = new BookContext())
{
    context.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;

    var books = context.Books.ToList();
}
```

Section 13.2: Tracking queries

- By default, queries that return entity types are **tracking**
- This means you can make changes to those entity instances and have those changes persisted by SaveChanges()

Example：

- The change to the book rating will be detected and persisted to the database during SaveChanges().

```
using (var context = new BookContext())
{
    var book = context.Books.FirstOrDefault(b => b.BookId == 1);
    book.Rating = 5;
    context.SaveChanges();
}
```

Section 13.3: Tracking and projections

- Even if the result type of the query isn't an entity type, if the result contains entity types they will still be tracked by default

Example：

- In the following query, which returns an anonymous type, the instances of Book in the result set will be tracked

```
using (var context = new BookContext())
{
    var book = context.Books.Select(b => new { Book = b, Authors = b.Authors.Count() });
}
```

```
}
```

- 如果结果集不包含任何实体类型，则不会执行跟踪

示例：

- 在以下查询中，返回一个匿名类型，包含实体的一些值（但不包含实际实体类型的实例），不会执行跟踪

```
using (var context = new BookContext())
{
    var book = context.Books.Select(b => new { Id = b.BookId, PublishedDate = b.Date });
}
```

```
}
```

- If the result set does not contain any entity types, then no tracking is performed

Example :

- In the following query, which returns an anonymous type with some of the values from the entity (but no instances of the actual entity type), there is **no tracking** performed.

```
using (var context = new BookContext())
{
    var book = context.Books.Select(b => new { Id = b.BookId, PublishedDate = b.Date });
}
```

belindoc.com

第14章：事务

第14.1节：Database.BeginTransaction()

多个操作可以在单个事务中执行，以便如果任何

操作失败，可以回滚更改。

```
using (var context = new PlanetContext())
{
    using (var transaction = context.Database.BeginTransaction())
    {
        try
        {
            //假设这能正常工作
            var jupiter = new Planet { Name = "Jupiter" };
            context.Planets.Add(jupiter);
            context.SaveChanges();

            //然后这将抛出异常
            var neptune = new Planet { Name = "Neptune" };
            context.Planets.Add(neptune);
            context.SaveChanges();

            //没有这行代码，数据库不会应用任何更改
            transaction.Commit();
        }
        catch (Exception ex)
        {
            //这里不需要调用 transaction.Rollback(), 因为事务对象
            //将超出作用域，释放时会自动回滚
        }
    }
}
```

请注意，开发者可能有显式调用transaction.Rollback()的习惯，因为这样代码更具自我说明性。此外，可能存在一些（不太知名的）Entity Framework查询提供程序未正确实现Dispose，这也需要显式调用transaction.Rollback()

Chapter 14: Transactions

Section 14.1: Database.BeginTransaction()

Multiple operations can be executed against a single transaction so that changes can be rolled back if any of the operations fail.

```
using (var context = new PlanetContext())
{
    using (var transaction = context.Database.BeginTransaction())
    {
        try
        {
            //Lets assume this works
            var jupiter = new Planet { Name = "Jupiter" };
            context.Planets.Add(jupiter);
            context.SaveChanges();

            //And then this will throw an exception
            var neptune = new Planet { Name = "Neptune" };
            context.Planets.Add(neptune);
            context.SaveChanges();

            //Without this line, no changes will get applied to the database
            transaction.Commit();
        }
        catch (Exception ex)
        {
            //There is no need to call transaction.Rollback() here as the transaction object
            //will go out of scope and disposing will roll back automatically
        }
    }
}
```

Note that it may be a developers' convention to call transaction.Rollback() explicitly, because it makes the code more self-explanatory. Also, there *may* be (less well-known) query providers for Entity Framework out there that don't implement Dispose correctly, which would also require an explicit transaction.Rollback() call.

第15章：管理实体状态

第15.1节：设置单个实体的Added状态

EntityState.Added 可以通过两种完全等效的方式设置：

- 1. 通过设置其在上下文中的条目状态：

```
context.Entry(entity).State = EntityState.Added;
```

- 2. 通过将其添加到上下文的DbSet中：

```
context.Entities.Add(entity);
```

调用SaveChanges时，实体将被插入到数据库中。当它有一个标识列（自动设置的、自增的主键）时，在调用SaveChanges之后，实体的主键属性将包含新生成的值，即使该属性之前已有值。

第15.2节：设置对象图的Added状态

将object graph（一组相关实体）的状态设置为Added，与将单个实体设置为Added不同（参见此示例）。

在示例中，我们存储行星及其卫星：

类模型

```
public class 行星
{
    public Planet()
    {
        Moons = new HashSet<Moon>();
    }
    public int ID { get; set; }
    public string Name { get; set; }
    public ICollection<Moon> Moons { get; set; }
}

public class Moon
{
    public int ID { get; set; }
    public int PlanetID { get; set; }
    public string Name { get; set; }
}
```

上下文

```
public class PlanetDb : DbContext
{
    public property DbSet<Planet> Planets { get; set; }
}
```

我们使用该上下文的一个实例来添加行星及其卫星：

Chapter 15: Managing entity state

Section 15.1: Setting state Added of a single entity

EntityState.Added can be set in two fully equivalent ways:

- 1. By setting the state of its entry in the context:

```
context.Entry(entity).State = EntityState.Added;
```

- 2. By adding it to a DbSet of the context:

```
context.Entities.Add(entity);
```

When calling SaveChanges, the entity will be inserted into the database. When it's got an identity column (an auto-set, auto-incrementing primary key), then after SaveChanges, the primary key property of the entity will contain the newly generated value, *even when this property already had a value*.

Section 15.2: Setting state Added of an object graph

Setting the state of an *object graph* (a collection of related entities) to Added is different than setting a single entity as Added (see this example).

In the example, we store planets and their moons:

Class model

```
public class Planet
{
    public Planet()
    {
        Moons = new HashSet<Moon>();
    }
    public int ID { get; set; }
    public string Name { get; set; }
    public ICollection<Moon> Moons { get; set; }
}

public class Moon
{
    public int ID { get; set; }
    public int PlanetID { get; set; }
    public string Name { get; set; }
}
```

Context

```
public class PlanetDb : DbContext
{
    public property DbSet<Planet> Planets { get; set; }
}
```

We use an instance of this context to add planets and their moons:

示例

```
var mars = new Planet { Name = "Mars" };
mars.Moons.Add(new Moon { Name = "Phobos" });
mars.Moons.Add(new Moon { Name = "Deimos" });

context.Planets.Add(mars);

Console.WriteLine(context.Entry(mars).State);
Console.WriteLine(context.Entry(mars.Moons.First()).State);
```

输出：

```
Added
Added
```

这里我们看到的是，添加一个Planet也会将一个卫星的状态设置为Added。

当将实体的状态设置为Added时，其导航属性中的所有实体（导航属性是指“导航”到其他实体的属性，如Planet.Moons）也会被标记为Added，除非它们已经附加到上下文中。

Example

```
var mars = new Planet { Name = "Mars" };
mars.Moons.Add(new Moon { Name = "Phobos" });
mars.Moons.Add(new Moon { Name = "Deimos" });

context.Planets.Add(mars);

Console.WriteLine(context.Entry(mars).State);
Console.WriteLine(context.Entry(mars.Moons.First()).State);
```

Output:

```
Added
Added
```

What we see here is that adding a Planet also sets the state of a moon to Added.

When setting an entity's state to Added, all entities in its navigation properties (properties that "navigate" to other entities, like Planet .Moons) are also marked as Added, *unless they already are attached to the context.*

第16章：加载相关实体

第16.1节：急切加载

急加载 让你一次性加载所有需要的实体。如果你希望通过一次数据库调用获取所有要操作的实体，那么急加载是最佳选择。它还允许你加载多级关联。

你有两种选项来加载相关实体，可以选择强类型或字符串重载的Include方法。

强类型。

```
// 加载一个包含创始人和地址详情的公司
int companyId = ...;
Company company = context.Companies
    .Include(m => m.Founder)
    .Include(m => m.Addresses)
    .SingleOrDefault(m => m.Id == companyId);

// 加载5个包含地址详情的公司，同时检索地址的国家和城市
// 信息
List<Company> companies = context.Companies
    .Include(m => m.Addresses.Select(a => a.Country));
    .Include(m => m.Addresses.Select(a => a.City))
    .Take(5).ToList();
```

此方法自 Entity Framework 4.1 起可用。确保你已引用using System.Data.Entity; 命名空间。

字符串重载。

```
// 加载一个包含创始人和地址详情的公司
int companyId = ...;
Company company = context.Companies
    .Include("Founder")
    .Include("Addresses")
    .SingleOrDefault(m => m.Id == companyId);

// 加载5个包含地址详情的公司，同时检索地址的国家和城市
// 信息
List<Company> companies = context.Companies
    .Include("Addresses.Country");
    .Include("Addresses.City"))
    .Take(5).ToList();
```

第16.2节：显式加载

关闭延迟加载后，可以通过显式调用条目的Load方法来延迟加载实体。单个导航属性使用Reference加载，而集合使用Collection获取。

```
Company company = context.Companies.FirstOrDefault();
// 加载创始人
context.Entry(company).Reference(m => m.Founder).Load();
// 加载地址
context.Entry(company).Collection(m => m.Addresses).Load();
```

由于是急加载，您可以使用上述方法的重载通过名称加载实体：

Chapter 16: Loading related entities

Section 16.1: Eager loading

Eager loading lets you load all your needed entities at once. If you prefer to get all your entities to work on in one database call, then Eager loading is the way to go. It also lets you load multiple levels.

You have two options to load related entities, you can choose either strongly typed or string overloads of the Include method.

Strongly typed.

```
// Load one company with founder and address details
int companyId = ...;
Company company = context.Companies
    .Include(m => m.Founder)
    .Include(m => m.Addresses)
    .SingleOrDefault(m => m.Id == companyId);

// Load 5 companies with address details, also retrieve country and city
// information of addresses
List<Company> companies = context.Companies
    .Include(m => m.Addresses.Select(a => a.Country));
    .Include(m => m.Addresses.Select(a => a.City))
    .Take(5).ToList();
```

This method is available since Entity Framework 4.1. Make sure you have the reference using System.Data.Entity; set.

String overload.

```
// Load one company with founder and address details
int companyId = ...;
Company company = context.Companies
    .Include("Founder")
    .Include("Addresses")
    .SingleOrDefault(m => m.Id == companyId);

// Load 5 companies with address details, also retrieve country and city
// information of addresses
List<Company> companies = context.Companies
    .Include("Addresses.Country");
    .Include("Addresses.City"))
    .Take(5).ToList();
```

Section 16.2: Explicit loading

After turning Lazy loading off you can lazily load entities by explicitly calling Load method for entries. Reference is used to load single navigation properties, whereas Collection is used to get collections.

```
Company company = context.Companies.FirstOrDefault();
// Load founder
context.Entry(company).Reference(m => m.Founder).Load();
// Load addresses
context.Entry(company).Collection(m => m.Addresses).Load();
```

As it is on Eager loading you can use overloads of above methods to load entiteis by their names:

```
Company company = context.Companies.FirstOrDefault();
// 加载创始人
context.Entry(company).Reference("Founder").Load();
// 加载地址
context.Entry(company).Collection("Addresses").Load();
```

过滤相关实体。

使用Query方法我们可以过滤已加载的相关实体：

```
Company company = context.Companies.FirstOrDefault();
// 加载位于巴库的地址
context.Entry(company)
    .Collection(m => m.Addresses)
    .Query()
    .Where(a => a.City.Name == "Baku")
    .Load();
```

第16.3节：延迟加载

默认启用延迟加载。延迟加载是通过创建派生的代理类并重写虚拟导航属性来实现的。延迟加载发生在属性首次被访问时。

```
int companyId = ...;
Company company = context.Companies
    .First(m => m.Id == companyId);
Person founder = company.Founder; // 加载了创始人

{
    // 地址详情是逐个加载的。
}
```

要关闭特定导航属性的延迟加载，只需从属性声明中移除 `virtual` 关键字：

```
public Person Founder { get; set; } // 已移除 "virtual" 关键字
```

如果你想完全关闭延迟加载，则需要更改配置，例如，在 *Context* 构造函数中：

```
public class MyContext : DbContext
{
    public MyContext(): base("Name=ConnectionString")
    {
        this.Configuration.LazyLoadingEnabled = false;
    }
}
```

注意： 如果你使用序列化，请记得关闭延迟加载。因为序列化器会访问每个属性，这会导致你从数据库加载所有属性。此外，你可能会遇到导航

属性之间的循环引用问题。

第16.4节：投影查询

如果需要以非规范化类型获取相关数据，或者例如只需要部分列，可以使用投影查询。如果没有使用额外类型的理由，也可以将值合并到一个匿名类型中。

```
Company company = context.Companies.FirstOrDefault();
// Load founder
context.Entry(company).Reference("Founder").Load();
// Load addresses
context.Entry(company).Collection("Addresses").Load();
```

Filter related entities.

Using *Query* method we can filter loaded related entities:

```
Company company = context.Companies.FirstOrDefault();
// Load addresses which are in Baku
context.Entry(company)
    .Collection(m => m.Addresses)
    .Query()
    .Where(a => a.City.Name == "Baku")
    .Load();
```

Section 16.3: Lazy loading

Lazy loading is enabled by default. Lazy loading is achieved by creating derived proxy classes and overriding virtual navigation proeptrties. Lazy loading occurs when property is accessed for the first time.

```
int companyId = ...;
Company company = context.Companies
    .First(m => m.Id == companyId);
Person founder = company.Founder; // Founder is loaded
foreach (Address address in company.Addresses)
{
    // Address details are loaded one by one.
}
```

To turn Lazy loading off for specific navigation properties just remove virtual keyword from property declaration:

```
public Person Founder { get; set; } // "virtual" keyword has been removed
```

If you want to completely turn off Lazy loading, then you have to change Configuration, for example, at *Context* constructor:

```
public class MyContext : DbContext
{
    public MyContext(): base("Name=ConnectionString")
    {
        this.Configuration.LazyLoadingEnabled = false;
    }
}
```

Note: Please remember to *turn off* Lazy loading if your are using serialization. Because serializers access every property you are going to load all of them from database. Additionally, you can run into loop between navigation properties.

Section 16.4: Projection Queries

If one needs related data in a denormalized type, or e.g. only a subset of columns one can use projection queries. If there is no reason for using an extra type, there is the possibility to join the values into an [anonymous type](#).

```

var dbContext = new MyDbContext();
var denormalizedType = from company in dbContext.Company
                        where company.Name == "MyFavoriteCompany"
                        join founder in dbContext.Founder
on company.FounderId equals founder.Id
                        select new
                        {
CompanyName = company.Name,
                        CompanyId = company.Id,
                        FounderName = founder.Name,
                        FounderId = founder.Id
                        };

```

或者使用查询语法：

```

var dbContext = new MyDbContext();
var denormalizedType = dbContext.Company
                        .Join(dbContext.Founder,
c => c.FounderId,
                        f => f.Id ,
                        (c, f) => new
                        {
CompanyName = c.Name,
                        公司编号 = c.Id,
                        创始人姓名 = f.Name,
                        创始人编号 = f.Id
                        })
                        .Select(cf => cf);

```

```

var dbContext = new MyDbContext();
var denormalizedType = from company in dbContext.Company
                        where company.Name == "MyFavoriteCompany"
                        join founder in dbContext.Founder
on company.FounderId equals founder.Id
                        select new
                        {
CompanyName = company.Name,
                        CompanyId = company.Id,
                        FounderName = founder.Name,
                        FounderId = founder.Id
                        };

```

Or with query-syntax:

```

var dbContext = new MyDbContext();
var denormalizedType = dbContext.Company
                        .Join(dbContext.Founder,
c => c.FounderId,
                        f => f.Id ,
                        (c, f) => new
                        {
CompanyName = c.Name,
                        CompanyId = c.Id,
                        FounderName = f.Name,
                        FounderId = f.Id
                        })
                        .Select(cf => cf);

```

第17章：模型约束

第17.1节：一对多关系

UserType 属于多个用户 <-> 用户拥有一个 UserType

带有必需条件的单向导航属性

```
public class UserType
{
    public int UserId {get; set;}
}
public class User
{
    public int UserId {get; set;}
    public int UserId {get; set;}
    public virtual UserType UserType {get; set;}
}

Entity<User>().HasRequired(u => u.UserType).WithMany().HasForeignKey(u => u.UserId);
```

单向导航属性，外键必须为可空类型（可选）

```
public class UserType
{
    public int UserId {get; set;}
}
public class User
{
    public int UserId {get; set;}
    public int? UserId {get; set;}
    public virtual UserType UserType {get; set;}
}

Entity<User>().HasOptional(u => u.UserType).WithMany().HasForeignKey(u => u.UserId);
```

双向导航属性（必需/可选，根据需要更改外键属性）

```
public class UserType
{
    public int UserId {get; set;}
    public virtual ICollection<User> Users {get; set;}
}
public class User
{
    public int UserId {get; set;}
    public int UserId {get; set;}
    public virtual UserType UserType {get; set;}
}
```

必需

```
Entity<User>().HasRequired(u => u.UserType).WithMany(ut => ut.Users).HasForeignKey(u => u.UserId);
```

可选

Chapter 17: Model Restraints

Section 17.1: One-to-many relationships

UserType belongs to many Users <-> Users have one UserType

One way navigation property with required

```
public class UserType
{
    public int UserId {get; set;}
}
public class User
{
    public int UserId {get; set;}
    public int UserId {get; set;}
    public virtual UserType UserType {get; set;}
}

Entity<User>().HasRequired(u => u.UserType).WithMany().HasForeignKey(u => u.UserId);
```

One way navigation property with optional (foreign key must be Nullable type)

```
public class UserType
{
    public int UserId {get; set;}
}
public class User
{
    public int UserId {get; set;}
    public int? UserId {get; set;}
    public virtual UserType UserType {get; set;}
}

Entity<User>().HasOptional(u => u.UserType).WithMany().HasForeignKey(u => u.UserId);
```

Two way navigation property with (required/optional change the foreign key property as needed)

```
public class UserType
{
    public int UserId {get; set;}
    public virtual ICollection<User> Users {get; set;}
}
public class User
{
    public int UserId {get; set;}
    public int UserId {get; set;}
    public virtual UserType UserType {get; set;}
}
```

Required

```
Entity<User>().HasRequired(u => u.UserType).WithMany(ut => ut.Users).HasForeignKey(u => u.UserId);
```

Optional

```
Entity<User>().HasOptional(u => u.UserType).WithMany(ut => ut.Users).HasForeignKey(u =>
u.UserId);
```

belindoc.com

```
Entity<User>().HasOptional(u => u.UserType).WithMany(ut => ut.Users).HasForeignKey(u =>
u.UserId);
```


第18章：使用PostgreSQL的实体框架

第18.1节：使用Npgsql数据提供程序将Entity Framework 6.1.3与PostgreSQL配合使用所需的预备步骤

1) 从位置 C:\Windows\Microsoft.NET\Framework\v4.0.30319\Config 和

C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Config 备份了 Machine.config 文件

2) 将它们复制到不同的位置并进行编辑

a) 在 <system.data> <DbProviderFactories> 下定位并添加

```
<add name="Npgsql 数据提供程序" invariant="Npgsql" support="FF"
description=".Net 框架用于 PostgreSQL 服务器的数据提供程序"
type="Npgsql.NpgsqlFactory, Npgsql, Version=2.2.5.0, Culture=neutral,
PublicKeyToken=5d8b90d52f46fda7" />
```

b) 如果上述条目已存在，检查版本并更新。

- 3. 用更改后的文件替换原始文件。
- 4.以管理员身份运行 VS2013 的开发者命令提示符。
 - 5. 如果已安装 Npgsql，请使用命令 "gacutil -u Npgsql" 卸载，然后安装新版本的 Npgsql 2.5.0 通过命令 "gacutil -i [dll路径]" 安装
- 6. 对 Mono.Security 4.0.0.0 执行上述操作
- 7. 下载 NpgsqlDdexProvider-2.2.0-VS2013.zip 并运行其中的 NpgsqlDdexProvider.vsix（请关闭所有 Visual Studio 实例）
- 8. 找到 EFTools6.1.3-beta1ForVS2013.msi 并运行它。
- 9. 创建新项目后，从管理 NuGet 包中安装 EntityFramework(6.1.3)、Npgsql(2.5.0) 和 Npgsql.EntityFramework(2.5.0) 版本。10) 完成，继续...在你的 MVC 项目中添加新的实体数据模型

Chapter 18: Entity Framework with PostgreSQL

Section 18.1: Pre-Steps needed in order to use Entity Framework 6.1.3 with PostgresSql using Npgsqlddexprovider

1)Took backup of Machine.config from locations C:\Windows\Microsoft.NET\Framework\v4.0.30319\Config and

C:\Windows\Microsoft.NET\Framework64\v4.0.30319\Config

2)Copy them to different location and edit them as

a)locate and add under <system.data> <DbProviderFactories>

```
<add name="Npgsql Data Provider" invariant="Npgsql" support="FF"
description=".Net Framework Data Provider for Postgresql Server"
type="Npgsql.NpgsqlFactory, Npgsql, Version=2.2.5.0, Culture=neutral,
PublicKeyToken=5d8b90d52f46fda7" />
```

b)if already exist above entry, check verison and update it.

- 3. Replace original files with changed ones.
- 4. run Developer Command Prompt for VS2013 as Administrator.
- 5. if Npgsql already installed use command " gacutil -u Npgsql " to uninstall then install new version of Npgsql 2.5.0 by command " gacutil -i [path of dll] "
- 6. Do above for Mono.Security 4.0.0.0
- 7. Download NpgsqlDdexProvider-2.2.0-VS2013.zip and run NpgsqlDdexProvider.vsix from it(Do close all instances of visual studio)
- 8. Found EFTools6.1.3-beta1ForVS2013.msi and run it.
- 9. After crating new project, Install version of EntityFramework(6.1.3), Npgsql(2.5.0) and Npgsql.EntityFramework(2.5.0) from Manage Nuget Packages.10)Its Done go ahead...Add new Entity Data Model in your MVC project

第19章：使用 SQLite 的实体框架

SQLite 是一个自包含、无服务器、支持事务的 SQL 数据库。它可以通过使用免费提供的 .NET SQLite 库和实体框架 SQLite 提供程序，在 .NET 应用程序中使用。本章节将介绍实体框架 SQLite 提供程序的设置和使用。

第19.1节：设置项目以使用带有 SQLite 提供程序的实体框架

实体框架库仅自带 SQL Server 提供程序。要使用 SQLite 需要额外的依赖项和配置。所有必需的依赖项均可通过 NuGet 获得。

安装 SQLite 托管库

所有托管依赖项都可以通过 NuGet 包管理器控制台安装。运行命令 `Install-Package System.Data.SQLite`。

```
PM> Install-Package System.Data.SQLite
Attempting to gather dependency information for package 'System.Data.SQLite.1.0.104' with respect to project
Attempting to resolve dependencies for package 'System.Data.SQLite.1.0.104' with DependencyBehavior 'Lowest'
Resolving actions to install package 'System.Data.SQLite.1.0.104'
Resolved actions to install package 'System.Data.SQLite.1.0.104'
Adding package 'EntityFramework.6.0.0' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Projects
Added package 'EntityFramework.6.0.0' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Projects\
Added package 'EntityFramework.6.0.0' to 'packages.config'
Executing script file 'c:\users\jason.tyler\documents\visual studio 2015\Projects\SQLiteSetup\packages\Entit
Executing script file 'c:\users\jason.tyler\documents\visual studio 2015\Projects\SQLiteSetup\packages\Entit

Type 'get-help EntityFramework' to see all available Entity Framework commands.
Successfully installed 'EntityFramework 6.0.0' to SQLiteSetup
Adding package 'System.Data.SQLite.Core.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 201
Added package 'System.Data.SQLite.Core.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015
Added package 'System.Data.SQLite.Core.1.0.104' to 'packages.config'
Successfully installed 'System.Data.SQLite.Core 1.0.104' to SQLiteSetup
Adding package 'System.Data.SQLite.EF6.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015
Added package 'System.Data.SQLite.EF6.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015\
Added package 'System.Data.SQLite.EF6.1.0.104' to 'packages.config'
Executing script file 'c:\users\jason.tyler\documents\visual studio 2015\Projects\SQLiteSetup\packages\Syste
Successfully installed 'System.Data.SQLite.EF6 1.0.104' to SQLiteSetup
Adding package 'System.Data.SQLite.Linq.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 201
Added package 'System.Data.SQLite.Linq.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015
Added package 'System.Data.SQLite.Linq.1.0.104' to 'packages.config'
Successfully installed 'System.Data.SQLite.Linq 1.0.104' to SQLiteSetup
Adding package 'System.Data.SQLite.1.0.104', which only has dependencies, to project 'SQLiteSetup'.
Adding package 'System.Data.SQLite.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Pro
Added package 'System.Data.SQLite.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Proj
Added package 'System.Data.SQLite.1.0.104' to 'packages.config'
Successfully installed 'System.Data.SQLite 1.0.104' to SQLiteSetup
```

如上所示，安装System.Data.SQLite时，会同时安装所有相关的托管库。这包括System.Data.SQLite.EF6，即SQLite的EF提供程序。该项目现在还引用了使用SQLite提供程序所需的程序集。

Chapter 19: Entity Framework with SQLite

SQLite is a self-contained, serverless, transactional SQL database. It can be used within a .NET application by utilizing both a freely available .NET SQLite library and Entity Framework SQLite provider. This topic will go into setup and usage of the Entity Framework SQLite provider.

Section 19.1: Setting up a project to use Entity Framework with an SQLite provider

The Entity Framework library comes only with an SQL Server provider. To use SQLite will require additional dependencies and configuration. All required dependencies are available on NuGet.

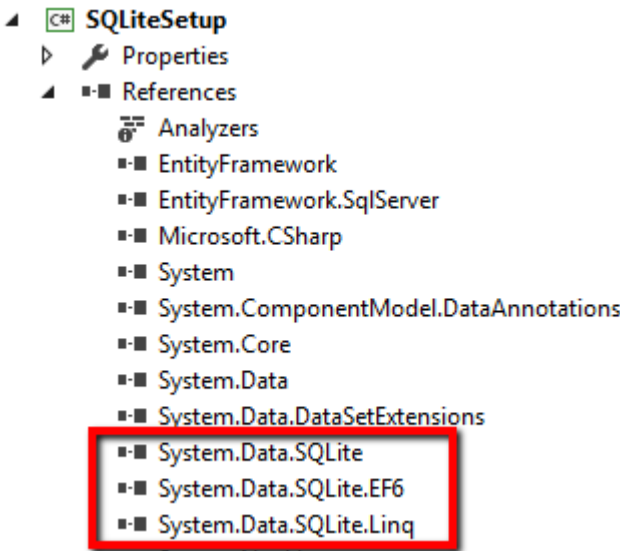
Install SQLite Managed Libraries

All of the mananged depedencies can be installed using the NuGet Package Manager Console. Run the command `Install-Package System.Data.SQLite`.

```
PM> Install-Package System.Data.SQLite
Attempting to gather dependency information for package 'System.Data.SQLite.1.0.104' with respect to project
Attempting to resolve dependencies for package 'System.Data.SQLite.1.0.104' with DependencyBehavior 'Lowest'
Resolving actions to install package 'System.Data.SQLite.1.0.104'
Resolved actions to install package 'System.Data.SQLite.1.0.104'
Adding package 'EntityFramework.6.0.0' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Projects
Added package 'EntityFramework.6.0.0' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Projects\
Added package 'EntityFramework.6.0.0' to 'packages.config'
Executing script file 'c:\users\jason.tyler\documents\visual studio 2015\Projects\SQLiteSetup\packages\Entit
Executing script file 'c:\users\jason.tyler\documents\visual studio 2015\Projects\SQLiteSetup\packages\Entit

Type 'get-help EntityFramework' to see all available Entity Framework commands.
Successfully installed 'EntityFramework 6.0.0' to SQLiteSetup
Adding package 'System.Data.SQLite.Core.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 201
Added package 'System.Data.SQLite.Core.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015
Added package 'System.Data.SQLite.Core.1.0.104' to 'packages.config'
Successfully installed 'System.Data.SQLite.Core 1.0.104' to SQLiteSetup
Adding package 'System.Data.SQLite.EF6.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015
Added package 'System.Data.SQLite.EF6.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015\
Added package 'System.Data.SQLite.EF6.1.0.104' to 'packages.config'
Executing script file 'c:\users\jason.tyler\documents\visual studio 2015\Projects\SQLiteSetup\packages\Syste
Successfully installed 'System.Data.SQLite.EF6 1.0.104' to SQLiteSetup
Adding package 'System.Data.SQLite.Linq.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 201
Added package 'System.Data.SQLite.Linq.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015
Added package 'System.Data.SQLite.Linq.1.0.104' to 'packages.config'
Successfully installed 'System.Data.SQLite.Linq 1.0.104' to SQLiteSetup
Adding package 'System.Data.SQLite.1.0.104', which only has dependencies, to project 'SQLiteSetup'.
Adding package 'System.Data.SQLite.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Pro
Added package 'System.Data.SQLite.1.0.104' to folder 'c:\users\jason.tyler\documents\visual studio 2015\Proj
Added package 'System.Data.SQLite.1.0.104' to 'packages.config'
Successfully installed 'System.Data.SQLite 1.0.104' to SQLiteSetup
```

As shown above, when installing System.Data .SQLite, all related managed libraries are installed with it. This includes System.Data .SQLite .EF6, the EF provider for SQLite. The project also now references the assemblies required to use the SQLite provider.



包含非托管库

SQLite 托管库依赖于一个名为SQLite.Interop.dll的非托管程序集。该程序集包含在随 SQLite 包下载的程序集中，并且在构建项目时会自动复制到你的构建目录中。然而，由于它是非托管的，因此不会包含在你的引用列表中。但请注意，该程序集必须随应用程序一起分发，SQLite 程序集才能正常工作。

注意：该程序集依赖于位数，这意味着你需要为计划支持的每种位数（x86/x64）包含特定的程序集。

编辑项目的 App.config

在使用 SQLite 作为实体框架提供程序之前，app.config 文件需要进行一些修改。

必需的修正

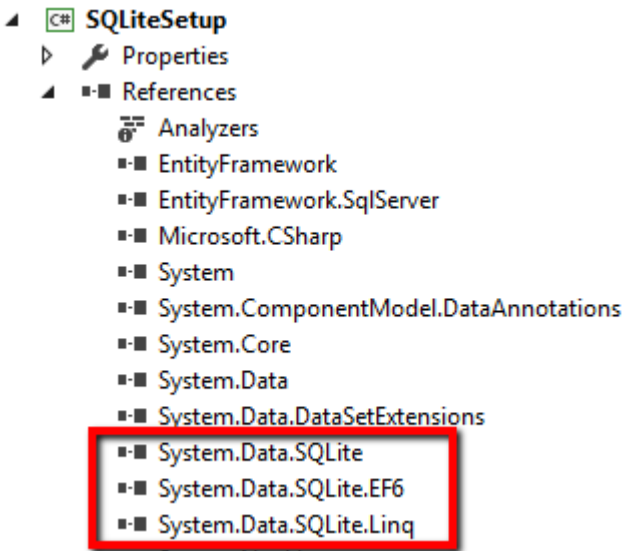
安装包时，app.config 文件会自动更新，包含 SQLite 和 SQLite EF 所需的条目。不幸的是，这些条目包含一些错误。需要修改它们才能正确工作。

首先，定位配置文件中的 DbProviderFactories 元素。它位于 system.data 元素内，内容如下

```
<DbProviderFactories>
  <remove invariant="System.Data.SQLite.EF6" />
  <add name="SQLite 数据提供程序 (Entity Framework 6)" invariant="System.Data.SQLite.EF6"
description=".NET 框架 SQLite 数据提供程序 (Entity Framework 6)"
type="System.Data.SQLite.EF6.SQLiteProviderFactory, System.Data.SQLite.EF6" />
  <remove invariant="System.Data.SQLite" /><add name="SQLite 数据提供程序"
invariant="System.Data.SQLite" description=".NET 框架 SQLite 数据提供程序"
type="System.Data.SQLite.SQLiteFactory, System.Data.SQLite" />
</DbProviderFactories>
```

这可以简化为只包含一个条目

```
<DbProviderFactories>
  <add name="SQLite 数据提供程序" invariant="System.Data.SQLite.EF6" description=".NET 框架
SQLite 数据提供程序" type="System.Data.SQLite.SQLiteFactory, System.Data.SQLite" />
</DbProviderFactories>
```



Including Unmanaged Library

The SQLite managed libraries are dependent on an unmanaged assembly named SQLite.Interop.dll. It is included with the package assemblies downloaded with the SQLite package, and they are automatically copied into your build directory when you build the project. However, because it's unmanaged, it will not be included in your reference list. But make note, this assembly must be distributed with the application for the SQLite assemblies to work.

Note: This assembly is bit-dependent, meaning you will need to include a specific assembly for each bitness you plan to support (x86/x64).

Editing the project's App.config

The app.config file will require some modifications before SQLite can be used as an Entity Framework provider.

Required Fixes

When installing the package, the app.config file is automatically updated to include the necessary entries for SQLite and SQLite EF. Unfortunately these entries contain some errors. They need to be modified before it will work correctly.

First, locate the DbProviderFactorieselement in the config file. It is within the system.data element and will contain the following

```
<DbProviderFactories>
  <remove invariant="System.Data.SQLite.EF6" />
  <add name="SQLite Data Provider (Entity Framework 6)" invariant="System.Data.SQLite.EF6"
description=".NET Framework Data Provider for SQLite (Entity Framework 6)"
type="System.Data.SQLite.EF6.SQLiteProviderFactory, System.Data.SQLite.EF6" />
  <remove invariant="System.Data.SQLite" /><add name="SQLite Data Provider"
invariant="System.Data.SQLite" description=".NET Framework Data Provider for SQLite"
type="System.Data.SQLite.SQLiteFactory, System.Data.SQLite" />
</DbProviderFactories>
```

This can be simplified to contain a single entry

```
<DbProviderFactories>
  <add name="SQLite Data Provider" invariant="System.Data.SQLite.EF6" description=".NET Framework
Data Provider for SQLite" type="System.Data.SQLite.SQLiteFactory, System.Data.SQLite" />
</DbProviderFactories>
```

通过此设置，我们已指定 EF6 SQLite 提供程序应使用 SQLite 工厂。

添加 SQLite 连接字符串

连接字符串可以添加到配置文件的根元素内。添加一个用于访问 SQLite 数据库的连接字符串。

```
<connectionStrings>
  <add name="TestContext" connectionString="data source=testdb.sqlite;initial
catalog=Test;App=EntityFramework;" providerName="System.Data.SQLite.EF6" />
</connectionStrings>
```

这里需要注意的重要一点是provider。它被设置为System.Data.SQLite.EF6。这告诉EF，当我们使用这个连接字符串时，我们想使用SQLite。指定的data source只是一个示例，具体取决于你的SQLite数据库的位置和名称。

你的第一个SQLite DbContext

完成所有安装和配置后，你现在可以开始使用一个DbContext，它将作用于你的SQLite数据库。

```
public class TestContext : DbContext
{
    public TestContext()
        : base("name=TestContext") { }
}
```

通过指定name=TestContext，我表明应该使用位于app.config文件中的TestContext连接字符串来创建上下文。该连接字符串已配置为使用SQLite，因此此上下文将使用SQLite数据库。

With this, we have specified the EF6 SQLite providers should use the SQLite factory.

Add SQLite connection string

Connection strings can be added to the configuration file within the root element. Add a connection string for accessing an SQLite database.

```
<connectionStrings>
  <add name="TestContext" connectionString="data source=testdb.sqlite;initial
catalog=Test;App=EntityFramework;" providerName="System.Data.SQLite.EF6" />
</connectionStrings>
```

The important thing to note here is the provider. It has been set to System.Data.SQLite.EF6. This tells EF that when we use this connection string, we want to use SQLite. The data source specified is just an example and will be dependent on the location and name of your SQLite database.

Your first SQLite DbContext

With all the installation and configuration complete, you can now start using a DbContext that will work on your SQLite database.

```
public class TestContext : DbContext
{
    public TestContext()
        : base("name=TestContext") { }
}
```

By specifying name=TestContext, I have indicating that the TestContext connection string located in the app.configfile should be used to create the context. That connection string was configured to use SQLite, so this context will use an SQLite database.

第20章：Entity

Framework中的.t4模板

第20.1节：动态向模型添加接口

在处理现有模型时，如果模型相当庞大且经常重新生成，而在需要抽象的情况下，手动为模型添加接口可能代价较高。在这种情况下，可能希望为模型生成添加一些动态行为。

以下示例将展示如何自动为具有特定列名的类添加接口：

在你的模型中，打开.tt文件，按以下方式修改EntityClassOpening方法，这将为具有POLICY_NO列的实体添加IPolicyNumber接口，为UNIQUE_ID添加IUniqueId接口

```
public string EntityClassOpening(EntityType entity)
{
    var stringsToMatch = new Dictionary<string,string> { { "POLICY_NO", "IPolicyNumber" }, {
"UNIQUE_ID", "IUniqueId" } };
    return string.Format(
CultureInfo.InvariantCulture,
        "{0} {1}partial class {2}{3}{4}",
Accessibility.ForType(entity),
_code.SpaceAfter(_code.AbstractOption(entity)),
_code.Escape(entity),
_code.StringBefore(" : ", _typeMapper.GetTypeName(entity.BaseType)),
        stringsToMatch.Any(o => entity.Properties.Any(n => n.Name == o.Key)) ? " : " +
string.Join(" ", stringsToMatch.Join(entity.Properties, l => l.Key, r => r.Name, (l,r) =>
l.Value)) : string.Empty);
}
```

这是一个具体的例子，但它展示了修改.tt模板的强大能力。

第20.2节：向实体类添加XML文档

在每个生成的模型类中，默认情况下不会添加文档注释。如果您想为每个生成的实体类使用 XML文档注释，请在 [model name].tt 文件中找到以下部分（modelname 是当前 EDMX 文件名）：

```
foreach (var entity in typeMapper.GetItemsToGenerate<EntityType>(itemCollection))
{
    fileManager.StartNewFile(entity.Name + ".cs");
    BeginNamespace(code); // 用于写入模型命名空间
#>
<#=#codeStringGenerator.UsingDirectives(inHeader: false)#>
```

您可以在 UsingDirectives 行之前添加 XML 文档注释，如下面示例所示：

```
foreach (var entity in typeMapper.GetItemsToGenerate<EntityType>(itemCollection))
{
    fileManager.StartNewFile(entity.Name + ".cs");
    BeginNamespace(code);
#>
    /// <summary>
    /// <#=#entity.Name#> 模型实体类。
    /// </summary>
```

Chapter 20: .t4 templates in entity framework

Section 20.1: Dynamically adding Interfaces to model

When working with existing model that is quite big and is being regenerated quite often in cases where abstraction needed it might be costly to manually go around redecorating model with interfaces. In such cases one might want to add some dynamic behavior to model generation.

Following example will show how automatically add interfaces on classes that have specific column names:

In your model go to .tt file modify the EntityClassOpening method in following way, this will add IPolicyNumber interface on entities that have POLICY_NO column, and IUniqueId on UNIQUE_ID

```
public string EntityClassOpening(EntityType entity)
{
    var stringsToMatch = new Dictionary<string,string> { { "POLICY_NO", "IPolicyNumber" }, {
"UNIQUE_ID", "IUniqueId" } };
    return string.Format(
CultureInfo.InvariantCulture,
        "{0} {1}partial class {2}{3}{4}",
Accessibility.ForType(entity),
_code.SpaceAfter(_code.AbstractOption(entity)),
_code.Escape(entity),
_code.StringBefore(" : ", _typeMapper.GetTypeName(entity.BaseType)),
        stringsToMatch.Any(o => entity.Properties.Any(n => n.Name == o.Key)) ? " : " +
string.Join(" ", stringsToMatch.Join(entity.Properties, l => l.Key, r => r.Name, (l,r) =>
l.Value)) : string.Empty);
}
```

This is one specific case but it shows a power of being able to modify .tt templates.

Section 20.2: Adding XML Documentation to Entity Classes

On every generated model classes there are no documentation comments added by default. If you want to use XML documentation comments for every generated entity classes, find this part inside [modelname].tt (modelname is current EDMX file name):

```
foreach (var entity in typeMapper.GetItemsToGenerate<EntityType>(itemCollection))
{
    fileManager.StartNewFile(entity.Name + ".cs");
    BeginNamespace(code); // used to write model namespace
#>
<#=#codeStringGenerator.UsingDirectives(inHeader: false)#>
```

You can add the XML documentation comments before UsingDirectives line as shown in example below:

```
foreach (var entity in typeMapper.GetItemsToGenerate<EntityType>(itemCollection))
{
    fileManager.StartNewFile(entity.Name + ".cs");
    BeginNamespace(code);
#>
    /// <summary>
    /// <#=#entity.Name#> model entity class.
    /// </summary>
```



```
<#=codeStringGenerator.UsingDirectives(inHeader: false)#>
```

生成的文档注释应包含实体名称，如下所示。

```
/// <summary>
/// 示例模型实体类。
/// </summary>
public partial class 示例
{
    // 模型内容
}
```

```
<#=codeStringGenerator.UsingDirectives(inHeader: false)#>
```

The generated documentation comment should be includes entity name as given below.

```
/// <summary>
/// Example model entity class.
/// </summary>
public partial class Example
{
    // model contents
}
```

第21章：高级映射场景：实体拆分，表拆分

如何配置您的EF模型以支持实体拆分或表拆分。

第21.1节：实体拆分

假设您有一个这样的实体类：

```
public class 人员
{
    public int PersonId { get; set; }
    public string 姓名 { get; set; }
    public string 邮编 { get; set; }
    public string 城市 { get; set; }
    public string 地址行 { get; set; }
}
```

然后假设您想将这个人员实体映射到两个表中——一个包含PersonId和姓名，另一个包含地址详细信息。当然，您也需要在这里包含PersonId以识别地址属于哪个人员。基本上，您想要的是将实体拆分成两个（甚至更多）部分。因此得名实体拆分。你可以通过将每个属性映射到不同的表来实现这一点：

```
public class MyDemoContext : DbContext
{
    public DbSet<Person> Products { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Person>().Map(m =>
        {
            m.Properties(t => new { t.PersonId, t.Name });
            m.ToTable("People");
        }).Map(m =>
        {
            m.Properties(t => new { t.PersonId, t.AddressLine, t.City, t.ZipCode });
            m.ToTable("PersonDetails");
        });
    }
}
```

这将创建两个表：People 和 PersonDetails。Person 有两个字段，PersonId 和 Name，PersonDetails 有四列，分别是 PersonId、AddressLine、City 和 ZipCode。在 People 表中，PersonId 是主键。在 PersonDetails 表中，主键也是 PersonId，但它同时是一个外键，引用 Person 表中的 PersonId。

如果你查询 People DbSet，EF 会根据 PersonId 进行连接，从两个表中获取数据以填充实体。

你也可以更改列的名称：

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Person>().Map(m =>
    {
        m.Properties(t => new { t.PersonId });
        m.Property(t => t.Name).HasColumnName("PersonName");
    });
}
```

Chapter 21: Advanced mapping scenarios: entity splitting, table splitting

How to configure your EF model to support entity splitting or table splitting.

Section 21.1: Entity splitting

So let's say you have an entity class like this:

```
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
    public string ZipCode { get; set; }
    public string City { get; set; }
    public string AddressLine { get; set; }
}
```

And then let's say that you want to map this Person entity into two tables — one with the PersonId and the Name, and another one with the address details. Of course you would need the PersonId here as well in order to identify which person the address belongs to. So basically what you want is to split the entity into two (or even more) parts. Hence the name, entity splitting. You can do this by mapping each of the properties to a different table:

```
public class MyDemoContext : DbContext
{
    public DbSet<Person> Products { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Person>().Map(m =>
        {
            m.Properties(t => new { t.PersonId, t.Name });
            m.ToTable("People");
        }).Map(m =>
        {
            m.Properties(t => new { t.PersonId, t.AddressLine, t.City, t.ZipCode });
            m.ToTable("PersonDetails");
        });
    }
}
```

This will create two tables: People and PersonDetails. Person has two fields, PersonId and Name, PersonDetails has four columns, PersonId, AddressLine, City and ZipCode. In People, PersonId is the primary key. In PersonDetails the primary key is also PersonId, but it is also a foreign key referencing PersonId in the Person table.

If you query the People DbSet, EF will do a join on the PersonIds to get the data from both tables to populate the entities.

You can also change the name of the columns:

```
protected override void OnModelCreating(DbModelBuilder modelBuilder)
{
    modelBuilder.Entity<Person>().Map(m =>
    {
        m.Properties(t => new { t.PersonId });
        m.Property(t => t.Name).HasColumnName("PersonName");
    });
}
```

```
m.ToTable("People");
}).Map(m =>
{
m.Property(t => t.PersonId).HasColumnName("ProprietorId");
m.Properties(t => new { t.AddressLine, t.City, t.ZipCode });
m.ToTable("PersonDetails");
});
}
```

这将创建相同的表结构，但在 People 表中将有一个 PersonName 列代替 Name 列，而在 PersonDetails 表中将有一个 ProprietorId 代替 PersonId 列。

第21.2节：表拆分

现在假设你想做实体拆分的相反操作：不是将一个实体映射到两个表，而是想将一个表映射到两个实体。这称为表拆分。假设你有一个包含五个列的表：PersonId、Name、AddressLine、City、ZipCode，其中 PersonId 是主键。然后你想创建如下的 EF 模型：

```
public class 人员
{
    public int PersonId { get; set; }
    public string Name { get; set; }
    public Address Address { get; set; }
}

public class Address
{
    public string 邮政编码 { get; set; }
    public string 城市 { get; set; }
    public string 地址行 { get; set; }
    public int 个人编号 { get; set; }
    public Person Person { get; set; }
}
```

有一件事很明显：Address 中没有 AddressId。这是因为这两个实体映射到同一张表，所以它们必须有相同的主键。如果你进行表拆分，这就是你必须处理的事情。因此，除了表拆分之外，你还必须配置 Address 实体并指定主键。方法如下：

```
public class MyDemoContext : DbContext
{
    public DbSet<Person> Products { get; set; }
    public DbSet<Address> Addresses { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Address>().HasKey(t => t.PersonId);
        modelBuilder.Entity<Person>().HasRequired(t => t.Address)
            .WithRequiredPrincipal(t => t.Person);

        modelBuilder.Entity<Person>().Map(m => m.ToTable("People"));
        modelBuilder.Entity<Address>().Map(m => m.ToTable("People"));
    }
}
```

```
m.ToTable("People");
}).Map(m =>
{
m.Property(t => t.PersonId).HasColumnName("ProprietorId");
m.Properties(t => new { t.AddressLine, t.City, t.ZipCode });
m.ToTable("PersonDetails");
});
}
```

This will create the same table structure, but in the People table there will be a PersonName column instead of the Name column, and in the PersonDetails table there will be a ProprietorId instead of the PersonId column.

Section 21.2: Table splitting

And now let's say you want to do the opposite of entity splitting: instead of mapping one entity into two tables, you would like to map one table into two entities. This is called table splitting. Let's say you have one table with five columns: PersonId, Name, AddressLine, City, ZipCode, where PersonId is the primary key. And then you would like to create an EF model like this:

```
public class Person
{
    public int PersonId { get; set; }
    public string Name { get; set; }
    public Address Address { get; set; }
}

public class Address
{
    public string ZipCode { get; set; }
    public string City { get; set; }
    public string AddressLine { get; set; }
    public int PersonId { get; set; }
    public Person Person { get; set; }
}
```

One thing jumps right out: there is no AddressId in Address. That's because the two entities are mapped to the same table, so they must have the same primary key as well. If you do table splitting, this is something you just have to deal with. So besides table splitting, you also have to configure the Address entity and specify the primary key. And here's how:

```
public class MyDemoContext : DbContext
{
    public DbSet<Person> Products { get; set; }
    public DbSet<Address> Addresses { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Address>().HasKey(t => t.PersonId);
        modelBuilder.Entity<Person>().HasRequired(t => t.Address)
            .WithRequiredPrincipal(t => t.Person);

        modelBuilder.Entity<Person>().Map(m => m.ToTable("People"));
        modelBuilder.Entity<Address>().Map(m => m.ToTable("People"));
    }
}
```

第22章：Entity Framework最佳实践 (简单与专业)

本文旨在介绍一种简单且专业的Entity Framework使用方法。

简单：因为它只需要一个类（带一个接口）

专业：因为它应用了[SOLID架构原则](#)

我不想多说.....让我们开始吧！

第22.1节：1- Entity Framework @ 数据层（基础）

本文将使用一个名为“Company”的简单数据库，包含两张表：

[dbo].[Categories]([CategoryID], [CategoryName])

[dbo].[Products]([ProductID], [CategoryID], [ProductName])

1-1 生成实体框架代码

在此层中，我们生成实体框架代码（在项目库中）（参见这篇文章了解如何操作），然后你将拥有以下类

```
public partial class CompanyContext : DbContext
public partial class Product
public partial class Category
```

1-2 创建基础接口

我们将为基础功能创建一个接口

```
public interface IDbRepository : IDisposable
{
    #region 表和视图功能

    IQueryable<TResult> GetAll<TResult>(bool noTracking = true) where TResult : class;
    TEntity Add<TEntity>(TEntity entity) where TEntity : class;
    TEntity 删除<TEntity>(TEntity 实体) where TEntity : class;
    TEntity 附加<TEntity>(TEntity 实体) where TEntity : class;
    TEntity 如果未附加则附加<TEntity>(TEntity 实体) where TEntity : class;

    #endregion 表和视图函数

    #region 事务函数

    int Commit();
    Task<int> CommitAsync(CancellationToken cancellationToken = default(CancellationToken));

    #endregion 事务函数

    #region 数据库存储过程和函数

    TResult Execute<TResult>(string functionName, params object[] parameters);

    #endregion 数据库存储过程和函数
```

Chapter 22: Best Practices For Entity Framework (Simple & Professional)

This article is to introduce a simple and professional practice to use Entity Framework.

Simple: because it only needs one class (with one interface)

Professional: because it applies [SOLID architecture principles](#)

I don't wish to talk more.... let's enjoy it!

Section 22.1: 1- Entity Framework @ Data layer (Basics)

In this article we will use a simple database called “Company” with two tables:

[dbo].[Categories]([CategoryID], [CategoryName])

[dbo].[Products]([ProductID], [CategoryID], [ProductName])

1-1 Generate Entity Framework code

In this layer we generate the Entity Framework code (in project library) (see [this article](#) in how can you do that) then you will have the following classes

```
public partial class CompanyContext : DbContext
public partial class Product
public partial class Category
```

1-2 Create basic Interface

We will create one interface for our basics functions

```
public interface IDbRepository : IDisposable
{
    #region Tables and Views functions

    IQueryable<TResult> GetAll<TResult>(bool noTracking = true) where TResult : class;
    TEntity Add<TEntity>(TEntity entity) where TEntity : class;
    TEntity Delete<TEntity>(TEntity entity) where TEntity : class;
    TEntity Attach<TEntity>(TEntity entity) where TEntity : class;
    TEntity AttachIfNot<TEntity>(TEntity entity) where TEntity : class;

    #endregion Tables and Views functions

    #region Transactions Functions

    int Commit();
    Task<int> CommitAsync(CancellationToken cancellationToken = default(CancellationToken));

    #endregion Transactions Functions

    #region Database Procedures and Functions

    TResult Execute<TResult>(string functionName, params object[] parameters);

    #endregion Database Procedures and Functions
```

1-3 实现基本接口

```

/// <summary>
/// 实现基本的表、视图、存储过程、函数和事务函数
/// 查询（获取所有）、插入（添加）、删除和附加
/// 无编辑（修改）功能（可以在不调用函数的情况下修改附加的实体）
/// 执行数据库存储过程或函数（Execute）
/// 事务函数（提交）
/// 如有需要，可添加更多函数
/// </summary>
/// <typeparam name="TEntity">实体框架表或视图</typeparam>
public class DbRepository : IRepository
{
    #region 受保护成员

    protected DbContext _dbContext;

    #endregion 受保护成员

    #region 构造函数

    /// <summary>
    /// 仓储构造函数
    /// </summary>
    /// <param name="dbContext">实体框架数据库上下文</param>
    public DbRepository(DbContext dbContext)
    {
        _dbContext = dbContext;

        ConfigureContext();
    }

    #endregion 构造函数

    #region IRepository 实现

    #region 表和视图函数

    /// <summary>
    /// 查询所有
    /// 将 noTracking 设置为 true 以仅进行选择（只读查询）
    /// 将 noTracking 设置为 false 以便在选择后进行插入、更新或删除
    ///
    public virtual IQueryable<TResult> GetAll<TResult>(bool noTracking = true) where TResult : class
    {
        var entityDbSet = GetDbSet<TResult>();

        if (noTracking)
            return entityDbSet.AsNoTracking();

        return entityDbSet;
    }

    public virtual TEntity Add<TEntity>(TEntity entity) where TEntity : class
    {
        return GetDbSet<TEntity>().Add(entity);
    }
}

```

1-3 Implementing basic Interface

```

/// <summary>
/// Implementing basic tables, views, procedures, functions, and transaction functions
/// Select (GetAll), Insert (Add), Delete, and Attach
/// No Edit (Modify) function (can modify attached entity without function call)
/// Executes database procedures or functions (Execute)
/// Transaction functions (Commit)
/// More functions can be added if needed
/// </summary>
/// <typeparam name="TEntity">Entity Framework table or view</typeparam>
public class DbRepository : IRepository
{
    #region Protected Members

    protected DbContext _dbContext;

    #endregion Protected Members

    #region Constructors

    /// <summary>
    /// Repository constructor
    /// </summary>
    /// <param name="dbContext">Entity framework database context</param>
    public DbRepository(DbContext dbContext)
    {
        _dbContext = dbContext;

        ConfigureContext();
    }

    #endregion Constructors

    #region IRepository Implementation

    #region Tables and Views functions

    /// <summary>
    /// Query all
    /// Set noTracking to true for selecting only (read-only queries)
    /// Set noTracking to false for insert, update, or delete after select
    /// </summary>
    public virtual IQueryable<TResult> GetAll<TResult>(bool noTracking = true) where TResult : class
    {
        var entityDbSet = GetDbSet<TResult>();

        if (noTracking)
            return entityDbSet.AsNoTracking();

        return entityDbSet;
    }

    public virtual TEntity Add<TEntity>(TEntity entity) where TEntity : class
    {
        return GetDbSet<TEntity>().Add(entity);
    }
}

```



```

    /// <summary>
    /// 删除已加载（附加）或未加载（分离）的实体
    /// 删除时无需加载对象
    /// 创建新的 TEntity 对象并设置 id，然后调用 Delete 函数
    /// </summary>
    /// <param name="entity">TEntity</param>
    /// <returns></returns>
    public virtual TEntity 删除<TEntity>(TEntity 实体) where TEntity : class
    {
        if (_dbContext.Entry(实体).State == EntityState.Detached)
        {
            _dbContext.Entry(实体).State = EntityState.Deleted;
            return 实体;
        }
        else
        {
            return GetDbSet<TEntity>().Remove(实体);
        }
    }

    public virtual TEntity 附加<TEntity>(TEntity 实体) where TEntity : class
    {
        return GetDbSet<TEntity>().Attach(实体);
    }

    public virtual TEntity 如果未附加则附加<TEntity>(TEntity 实体) where TEntity : class
    {
        if (_dbContext.Entry(实体).State == EntityState.Detached)
            return 附加(实体);

        return entity;
    }

#endregion 表和视图函数

#region 事务函数

    /// <summary>
    /// 将此上下文中所做的所有更改保存到基础数据库。
    /// </summary>
    /// <returns>写入基础数据库的对象数量。</returns>
    public virtual int Commit()
    {
        return _dbContext.SaveChanges();
    }

    /// <summary>
    /// 异步将此上下文中所做的所有更改保存到基础数据库。
    /// </summary>
    /// <param name="cancellationToken">等待任务完成时用于观察的 System.Threading.CancellationToken。</param>

    /// <returns>表示异步保存操作的任务。任务结果包含写入基础数据库的对象数量。</returns>

    public virtual Task<int> CommitAsync(CancellationToken cancellationToken =
default(CancellationToken))
    {
        return _dbContext.SaveChangesAsync(cancellationToken);
    }

#endregion 事务函数

#region 数据库存储过程和函数

    /// <summary>

```

```

    /// <summary>
    /// Delete loaded (attached) or unloaded (Detached) entity
    /// No need to load object to delete it
    /// Create new object of TEntity and set the id then call Delete function
    /// </summary>
    /// <param name="entity">TEntity</param>
    /// <returns></returns>
    public virtual TEntity Delete<TEntity>(TEntity entity) where TEntity : class
    {
        if (_dbContext.Entry(entity).State == EntityState.Detached)
        {
            _dbContext.Entry(entity).State = EntityState.Deleted;
            return entity;
        }
        else
        {
            return GetDbSet<TEntity>().Remove(entity);
        }
    }

    public virtual TEntity Attach<TEntity>(TEntity entity) where TEntity : class
    {
        return GetDbSet<TEntity>().Attach(entity);
    }

    public virtual TEntity AttachIfNot<TEntity>(TEntity entity) where TEntity : class
    {
        if (_dbContext.Entry(entity).State == EntityState.Detached)
            return Attach(entity);

        return entity;
    }

#endregion Tables and Views functions

#region Transactions Functions

    /// <summary>
    /// Saves all changes made in this context to the underlying database.
    /// </summary>
    /// <returns>The number of objects written to the underlying database.</returns>
    public virtual int Commit()
    {
        return _dbContext.SaveChanges();
    }

    /// <summary>
    /// Asynchronously saves all changes made in this context to the underlying database.
    /// </summary>
    /// <param name="cancellationToken">A System.Threading.CancellationToken to observe while waiting
    for the task to complete.</param>
    /// <returns>A task that represents the asynchronous save operation. The task result contains
    the number of objects written to the underlying database.</returns>
    public virtual Task<int> CommitAsync(CancellationToken cancellationToken =
default(CancellationToken))
    {
        return _dbContext.SaveChangesAsync(cancellationToken);
    }

#endregion Transactions Functions

#region Database Procedures and Functions

    /// <summary>

```

```
/// 在上下文中执行任何函数
/// 用于调用数据库存储过程和函数
/// </summary>>
/// <typeparam name="TResult">返回函数类型</typeparam>
/// <param name="functionName">上下文函数名称</param>
/// <param name="parameters">上下文函数参数，顺序相同</param>
public virtual TResult Execute<TResult>(string functionName, params object[] parameters)
{
    MethodInfo method = _dbContext.GetType().GetMethod(functionName);

    return (TResult)method.Invoke(_dbContext, parameters);
}

#endregion 数据库存储过程和函数

#endregion IRepository 实现

#region IDisposable 实现

public void Dispose()
{
    _dbContext.Dispose();
}

#endregion IDisposable 实现

#region 受保护的函数

/// <summary>
/// 设置上下文配置
/// </summary>
protected virtual void ConfigureContext()
{
    // 设置您推荐的上下文配置
    _dbContext.Configuration.LazyLoadingEnabled = false;
}

#endregion 受保护的函数

#region 私有函数

private DbSet<TEntity> GetDbSet<TEntity>() where TEntity : class
{
    return _dbContext.Set<TEntity>();
}

#endregion 私有函数

}
```

第22.2节：2- 实体框架 @ 业务层

在此层我们将编写应用程序业务逻辑。

建议针对每个展示界面，创建包含该界面所需所有功能的业务接口和实现类。

下面我们以产品界面业务为例进行编写

```
/// <summary>
```

```
/// Executes any function in the context
/// use to call database procedures and functions
/// </summary>>
/// <typeparam name="TResult">return function type</typeparam>
/// <param name="functionName">context function name</param>
/// <param name="parameters">context function parameters in same order</param>
public virtual TResult Execute<TResult>(string functionName, params object[] parameters)
{
    MethodInfo method = _dbContext.GetType().GetMethod(functionName);

    return (TResult)method.Invoke(_dbContext, parameters);
}

#endregion Database Procedures and Functions

#endregion IRepository Implementation

#region IDisposable Implementation

public void Dispose()
{
    _dbContext.Dispose();
}

#endregion IDisposable Implementation

#region Protected Functions

/// <summary>
/// Set Context Configuration
/// </summary>
protected virtual void ConfigureContext()
{
    // set your recommended Context Configuration
    _dbContext.Configuration.LazyLoadingEnabled = false;
}

#endregion Protected Functions

#region Private Functions

private DbSet<TEntity> GetDbSet<TEntity>() where TEntity : class
{
    return _dbContext.Set<TEntity>();
}

#endregion Private Functions

}
```

Section 22.2: 2- Entity Framework @ Business layer

In this layer we will write the application business.

It is recommended for each presentation screen, you create the business interface and implementation class that contain all required functions for the screen.

Below we will write the business for product screen as example

```
/// <summary>
```

```

/// 包含产品业务功能
/// </summary>
public interface IProductBusiness
{
    Product SelectById(int productId, bool noTracking = true);
    Task<IEnumerable<dynamic>> SelectByCategoryAsync(int categoryId);
    Task<Product> InsertAsync(string productName, int categoryId);
    Product InsertForNewCategory(string productName, string categoryName);
    Product Update(int productId, string productName, int categoryId);
    Product Update2(int productId, string productName, int categoryId);
    int DeleteWithoutLoad(int productId);
    int DeleteLoadedProduct(Product product);
    IEnumerable<GetProductsCategory_Result> GetProductsCategory(int categoryId);
}

```

```

/// <summary>
/// 实现产品业务功能
/// </summary>
public class ProductBusiness : IProductBusiness
{
    #region 私有成员

    private IDbRepository _dbRepository;

    #endregion 私有成员

    #region 构造函数

    /// <summary>
    /// 产品业务构造函数
    /// </summary>
    /// <param name="dbRepository"></param>
    public ProductBusiness(IDbRepository dbRepository)
    {
        _dbRepository = dbRepository;
    }

    #endregion 构造函数

    #region IProductBusiness 功能

    /// <summary>
    /// 根据Id选择产品
    /// </summary>
    public Product 根据Id选择(int productId, bool noTracking = true)
    {
        var products = _dbRepository.GetAll<Product>(noTracking);

        return products.FirstOrDefault(pro => pro.ProductID == productId);
    }

    /// <summary>
    /// 异步根据类别Id选择产品
    /// 要使用异步方法, 请添加对EntityFramework 6或更高版本dll的引用
    /// 还需要包含命名空间 "System.Data.Entity"
    /// </summary>
    /// <param name="CategoryId">类别Id</param>
    /// <returns>返回你想返回的任何对象</returns>
    public async Task<IEnumerable<dynamic>> 异步根据类别选择(int CategoryId)
    {

```

```

/// Contains Product Business functions
/// </summary>
public interface IProductBusiness
{
    Product SelectById(int productId, bool noTracking = true);
    Task<IEnumerable<dynamic>> SelectByCategoryAsync(int categoryId);
    Task<Product> InsertAsync(string productName, int categoryId);
    Product InsertForNewCategory(string productName, string categoryName);
    Product Update(int productId, string productName, int categoryId);
    Product Update2(int productId, string productName, int categoryId);
    int DeleteWithoutLoad(int productId);
    int DeleteLoadedProduct(Product product);
    IEnumerable<GetProductsCategory_Result> GetProductsCategory(int categoryId);
}

```

```

/// <summary>
/// Implementing Product Business functions
/// </summary>
public class ProductBusiness : IProductBusiness
{
    #region Private Members

    private IDbRepository _dbRepository;

    #endregion Private Members

    #region Constructors

    /// <summary>
    /// Product Business Constructor
    /// </summary>
    /// <param name="dbRepository"></param>
    public ProductBusiness(IDbRepository dbRepository)
    {
        _dbRepository = dbRepository;
    }

    #endregion Constructors

    #region IProductBusiness Function

    /// <summary>
    /// Selects Product By Id
    /// </summary>
    public Product SelectById(int productId, bool noTracking = true)
    {
        var products = _dbRepository.GetAll<Product>(noTracking);

        return products.FirstOrDefault(pro => pro.ProductID == productId);
    }

    /// <summary>
    /// Selects Products By Category Id Async
    /// To have async method, add reference to EntityFramework 6 dll or higher
    /// also you need to have the namespace "System.Data.Entity"
    /// </summary>
    /// <param name="CategoryId">CategoryId</param>
    /// <returns>Return what ever the object that you want to return</returns>
    public async Task<IEnumerable<dynamic>> SelectByCategoryAsync(int CategoryId)
    {

```

```

var products = _dbRepository.GetAll<Product>();
var categories = _dbRepository.GetAll<Category>();

var result = (from pro in products
              join cat in categories
on pro.CategoryID equals cat.CategoryID
              where pro.CategoryID == categoryId
              select new
              {
                ProductId = pro.ProductID,
                ProductName = pro.ProductName,
                CategoryName = cat.CategoryName
              });

return await result.ToListAsync();
}

/// <summary>
/// 为给定类别异步插入新产品
///
public async Task<Product> InsertAsync(string productName, int categoryId)
{
    var newProduct = _dbRepository.Add(new Product() { ProductName = productName, CategoryID =
categoryId });

    await _dbRepository.CommitAsync();

    return newProduct;
}

/// <summary>
/// 插入新产品和新类别
/// 在一个事务中执行多个数据库操作
/// 每个 _dbRepository.Commit(); 将提交一个事务
///
public Product InsertForNewCategory(string productName, string categoryName)
{
    var newCategory = _dbRepository.Add(new Category() { CategoryName = categoryName });
    var newProduct = _dbRepository.Add(new Product() { ProductName = productName, Category =
newCategory });

    _dbRepository.Commit();

    return newProduct;
}

/// <summary>
/// 使用跟踪更新指定产品
/// </summary>
public Product Update(int productId, string productName, int categoryId)
{
    var product = SelectById(productId, false);
    product.CategoryID = categoryId;
    product.ProductName = productName;

    _dbRepository.Commit();

    return product;
}

/// <summary>

```

```

var products = _dbRepository.GetAll<Product>();
var categories = _dbRepository.GetAll<Category>();

var result = (from pro in products
              join cat in categories
on pro.CategoryID equals cat.CategoryID
              where pro.CategoryID == categoryId
              select new
              {
                ProductId = pro.ProductID,
                ProductName = pro.ProductName,
                CategoryName = cat.CategoryName
              });

return await result.ToListAsync();
}

/// <summary>
/// Insert Async new product for given category
/// </summary>
public async Task<Product> InsertAsync(string productName, int categoryId)
{
    var newProduct = _dbRepository.Add(new Product() { ProductName = productName, CategoryID =
categoryId });

    await _dbRepository.CommitAsync();

    return newProduct;
}

/// <summary>
/// Insert new product and new category
/// Do many database actions in one transaction
/// each _dbRepository.Commit(); will commit one transaction
/// </summary>
public Product InsertForNewCategory(string productName, string categoryName)
{
    var newCategory = _dbRepository.Add(new Category() { CategoryName = categoryName });
    var newProduct = _dbRepository.Add(new Product() { ProductName = productName, Category =
newCategory });

    _dbRepository.Commit();

    return newProduct;
}

/// <summary>
/// Update given product with tracking
/// </summary>
public Product Update(int productId, string productName, int categoryId)
{
    var product = SelectById(productId, false);
    product.CategoryID = categoryId;
    product.ProductName = productName;

    _dbRepository.Commit();

    return product;
}

/// <summary>

```

```

/// 使用无跟踪和附加功能更新指定产品
/// </summary>
public Product Update2(int productId, string productName, int categoryId)
{
    var product = SelectById(productId);
    _dbRepository.Attach(product);

    product.CategoryID = categoryId;
    product.ProductName = productName;

    _dbRepository.Commit();

    return product;
}

/// <summary>
/// 删除产品而不加载它
///
public int DeleteWithoutLoad(int productId)
{
    _dbRepository.Delete(new Product() { ProductID = productId });

    return _dbRepository.Commit();
}

/// <summary>
/// 删除产品, 先加载它
///
public int DeleteLoadedProduct(Product product)
{
    _dbRepository.Delete(product);

    return _dbRepository.Commit();
}

/// <summary>
/// 假设数据库中有以下存储过程
/// PROCEDURE [dbo].[GetProductsCategory] @CategoryID INT, @OrderBy VARCHAR(50)
///
public IEnumerable<GetProductsCategory_Result> GetProductsCategory(int categoryId)
{
    return
    _dbRepository.Execute<IEnumerable<GetProductsCategory_Result>>("GetProductsCategory", categoryId,
    "ProductName DESC");
}

#endregion IProductBusiness 功能
}

```

第22.3节：3- 在表示层使用业务层 (MVC)

在本例中，我们将在表示层使用业务层。我们将以MVC作为表示层的示例（但你也可以使用任何其他表示层）。

我们首先需要注册IoC（我们将使用Unity，但你可以使用任何IoC），然后编写我们的表示层

3-1 在MVC中注册Unity类型

```

/// Update given product with no tracking and attach function
/// </summary>
public Product Update2(int productId, string productName, int categoryId)
{
    var product = SelectById(productId);
    _dbRepository.Attach(product);

    product.CategoryID = categoryId;
    product.ProductName = productName;

    _dbRepository.Commit();

    return product;
}

/// <summary>
/// Deletes product without loading it
/// </summary>
public int DeleteWithoutLoad(int productId)
{
    _dbRepository.Delete(new Product() { ProductID = productId });

    return _dbRepository.Commit();
}

/// <summary>
/// Deletes product after loading it
/// </summary>
public int DeleteLoadedProduct(Product product)
{
    _dbRepository.Delete(product);

    return _dbRepository.Commit();
}

/// <summary>
/// Assuming we have the following procedure in database
/// PROCEDURE [dbo].[GetProductsCategory] @CategoryID INT, @OrderBy VARCHAR(50)
/// </summary>
public IEnumerable<GetProductsCategory_Result> GetProductsCategory(int categoryId)
{
    return
    _dbRepository.Execute<IEnumerable<GetProductsCategory_Result>>("GetProductsCategory", categoryId,
    "ProductName DESC");
}

#endregion IProductBusiness Function
}

```

Section 22.3: 3- Using Business layer @ Presentation layer (MVC)

In this example we will use the Business layer in Presentation layer. And we will use MVC as example of Presentation layer (but you can use any other Presentation layer).

We need first to register the IoC (we will use Unity, but you can use any IoC), then write our Presentation layer

3-1 Register Unity types within MVC

3-1-1 添加“ASP.NET MVC的Unity引导程序”NuGet包

3-1-2 在Global.asax.cs文件的Application_Start()函数中添加UnityWebActivator.Start();

3-1-3 按照以下方式修改UnityConfig.RegisterTypes函数

```
public static void RegisterTypes(IUnityContainer container)
{
    // 数据访问层
    container.RegisterType<DbContext, CompanyContext>(new PerThreadLifetimeManager());
    container.RegisterType(typeof(IDbRepository), typeof(DbRepository), new
    PerThreadLifetimeManager());

    // 业务层
    container.RegisterType<IProductBusiness, ProductBusiness>(new PerThreadLifetimeManager());
}
```

3-2 在表示层（MVC）中使用业务层

```
public class ProductController : Controller
{
    #region 私有成员

    IProductBusiness _productBusiness;

    #endregion 私有成员

    #region 构造函数

    public ProductController(IProductBusiness productBusiness)
    {
        _productBusiness = productBusiness;
    }

    #endregion 构造函数

    #region 操作函数

    [HttpPost]
    public ActionResult InsertForNewCategory(string productName, string categoryName)
    {
        try
        {
            // 你可以使用任何 IProductBusiness 的函数
            var newProduct = _productBusiness.InsertForNewCategory(productName, categoryName);

            return Json(new { success = true, data = newProduct });
        }
        catch (Exception ex)
        { /* 记录异常 */
            return Json(new { success = false, errorMessage = ex.Message });
        }
    }

    [HttpDelete]
    public ActionResult SmartDeleteWithoutLoad(int productId)
    {
        try
        {
            // 删除产品但不加载
        }
    }
}
```

3-1-1 Add “Unity bootstrapper for ASP.NET MVC” NuGet package

3-1-2 Add UnityWebActivator.Start(); in Global.asax.cs file (Application_Start() function)

3-1-3 Modify UnityConfig.RegisterTypes function as following

```
public static void RegisterTypes(IUnityContainer container)
{
    // Data Access Layer
    container.RegisterType<DbContext, CompanyContext>(new PerThreadLifetimeManager());
    container.RegisterType(typeof(IDbRepository), typeof(DbRepository), new
    PerThreadLifetimeManager());

    // Business Layer
    container.RegisterType<IProductBusiness, ProductBusiness>(new PerThreadLifetimeManager());
}
```

3-2 Using Business layer @ Presentation layer (MVC)

```
public class ProductController : Controller
{
    #region Private Members

    IProductBusiness _productBusiness;

    #endregion Private Members

    #region Constructors

    public ProductController(IProductBusiness productBusiness)
    {
        _productBusiness = productBusiness;
    }

    #endregion Constructors

    #region Action Functions

    [HttpPost]
    public ActionResult InsertForNewCategory(string productName, string categoryName)
    {
        try
        {
            // you can use any of IProductBusiness functions
            var newProduct = _productBusiness.InsertForNewCategory(productName, categoryName);

            return Json(new { success = true, data = newProduct });
        }
        catch (Exception ex)
        { /* log ex */
            return Json(new { success = false, errorMessage = ex.Message });
        }
    }

    [HttpDelete]
    public ActionResult SmartDeleteWithoutLoad(int productId)
    {
        try
        {
            // deletes product without load
        }
    }
}
```

```

        var deletedProduct = _productBusiness.DeleteWithoutLoad(productId);

        return Json(new { success = true, data = deletedProduct });
    }
    catch (Exception ex)
    {
        /* 记录异常 */
        return Json(new { success = false, errorMessage = ex.Message });
    }
}

public async Task<ActionResult> SelectByCategoryAsync(int categoryId)
{
    try
    {
        var results = await _productBusiness.SelectByCategoryAsync(categoryId);

        return Json(new { success = true, data = results }, JsonRequestBehavior.AllowGet);
    }
    catch (Exception ex)
    {
        /* log ex */
        return Json(new { success = false, errorMessage = ex.Message }, JsonRequestBehavior.AllowGet);
    }
}

#endregion Action Functions
}

```

第22.4节：4- 实体框架 @ 单元测试层

在单元测试层，我们通常测试业务层的功能。为了做到这一点，我们将移除数据层（实体框架）的依赖。

现在的问题是：如何移除实体框架的依赖，以便对业务层函数进行单元测试？

答案很简单：我们将为 IDbRepository 接口创建一个假的实现，然后就可以进行单元测试了

4-1 实现基本接口（伪实现）

```

class FakeDbRepository : IDbRepository
{
    #region 受保护成员

    protected Hashtable _dbContext;
    protected int _numberOfRowsAffected;
    protected Hashtable _contextFunctionsResults;

    #endregion 受保护成员

    #region 构造函数

    public FakeDbRepository(Hashtable contextFunctionsResults = null)
    {
        _dbContext = new Hashtable();
        _numberOfRowsAffected = 0;
        _contextFunctionsResults = contextFunctionsResults;
    }

    #endregion 构造函数
}

```

```

        var deletedProduct = _productBusiness.DeleteWithoutLoad(productId);

        return Json(new { success = true, data = deletedProduct });
    }
    catch (Exception ex)
    {
        /* log ex */
        return Json(new { success = false, errorMessage = ex.Message });
    }
}

public async Task<ActionResult> SelectByCategoryAsync(int categoryId)
{
    try
    {
        var results = await _productBusiness.SelectByCategoryAsync(categoryId);

        return Json(new { success = true, data = results }, JsonRequestBehavior.AllowGet);
    }
    catch (Exception ex)
    {
        /* log ex */
        return Json(new { success = false, errorMessage = ex.Message }, JsonRequestBehavior.AllowGet);
    }
}

#endregion Action Functions
}

```

Section 22.4: 4- Entity Framework @ Unit Test Layer

In Unit Test layer we usually test the Business Layer functionalities. And in order to do this, we will remove the Data Layer (Entity Framework) dependencies.

And the question now is: How can I remove the Entity Framework dependencies in order to unit test the Business Layer functions?

And the answer is simple: we will a fake implementation for IDbRepository Interface then we can do our unit test

4-1 Implementing basic Interface (fake implementation)

```

class FakeDbRepository : IDbRepository
{
    #region Protected Members

    protected Hashtable _dbContext;
    protected int _numberOfRowsAffected;
    protected Hashtable _contextFunctionsResults;

    #endregion Protected Members

    #region Constructors

    public FakeDbRepository(Hashtable contextFunctionsResults = null)
    {
        _dbContext = new Hashtable();
        _numberOfRowsAffected = 0;
        _contextFunctionsResults = contextFunctionsResults;
    }

    #endregion Constructors
}

```

```

#region IRepository 实现

#region 表和视图功能

public IQueryable<TResult> GetAll<TResult>(bool noTracking = true) where TResult : class
{
    return GetDbSet<TResult>().AsQueryable();
}

public TEntity Add<TEntity>(TEntity entity) where TEntity : class
{
    GetDbSet<TEntity>().Add(entity);
    ++_numberOfRowsAffected;
    return entity;
}

public TEntity Delete<TEntity>(TEntity entity) where TEntity : class
{
    GetDbSet<TEntity>().Remove(entity);
    ++_numberOfRowsAffected;
    return entity;
}

public TEntity Attach<TEntity>(TEntity entity) where TEntity : class
{
    return Add(entity);
}

public TEntity AttachIfNot<TEntity>(TEntity entity) where TEntity : class
{
    if (!GetDbSet<TEntity>().Contains(entity))
        return Attach(entity);

    return entity;
}

#endregion 表和视图函数

#region 事务函数

public virtual int Commit()
{
    var numberOfRowsAffected = _numberOfRowsAffected;
    _numberOfRowsAffected = 0;
    return numberOfRowsAffected;
}

public virtual Task<int> CommitAsync(Cancellation_token cancellation_token =
default(Cancellation_token))
{
    var numberOfRowsAffected = _numberOfRowsAffected;
    _numberOfRowsAffected = 0;
    return new Task<int>(() => numberOfRowsAffected);
}

#endregion 事务函数

#region 数据库存储过程和函数

public virtual TResult Execute<TResult>(string functionName, params object[] parameters)
{

```

```

#region IRepository Implementation

#region Tables and Views functions

public IQueryable<TResult> GetAll<TResult>(bool noTracking = true) where TResult : class
{
    return GetDbSet<TResult>().AsQueryable();
}

public TEntity Add<TEntity>(TEntity entity) where TEntity : class
{
    GetDbSet<TEntity>().Add(entity);
    ++_numberOfRowsAffected;
    return entity;
}

public TEntity Delete<TEntity>(TEntity entity) where TEntity : class
{
    GetDbSet<TEntity>().Remove(entity);
    ++_numberOfRowsAffected;
    return entity;
}

public TEntity Attach<TEntity>(TEntity entity) where TEntity : class
{
    return Add(entity);
}

public TEntity AttachIfNot<TEntity>(TEntity entity) where TEntity : class
{
    if (!GetDbSet<TEntity>().Contains(entity))
        return Attach(entity);

    return entity;
}

#endregion Tables and Views functions

#region Transactions Functions

public virtual int Commit()
{
    var numberOfRowsAffected = _numberOfRowsAffected;
    _numberOfRowsAffected = 0;
    return numberOfRowsAffected;
}

public virtual Task<int> CommitAsync(Cancellation_token cancellation_token =
default(Cancellation_token))
{
    var numberOfRowsAffected = _numberOfRowsAffected;
    _numberOfRowsAffected = 0;
    return new Task<int>(() => numberOfRowsAffected);
}

#endregion Transactions Functions

#region Database Procedures and Functions

public virtual TResult Execute<TResult>(string functionName, params object[] parameters)
{

```

```

        if (_contextFunctionsResults != null && _contextFunctionsResults.Contains(functionName))
            return (TResult)_contextFunctionsResults[functionName];

        throw new NotImplementedException();
    }

#endregion 数据库存储过程和函数

#endregion IRepository 实现

#region IDisposable 实现

public void Dispose()
{

}

#endregion IDisposable 实现

#region 私有函数

private List<TEntity> GetDbSet<TEntity>() where TEntity : class
{
    if (!_dbContext.Contains(typeof(TEntity)))
        _dbContext.Add(typeof(TEntity), new List<TEntity>());

    return (List<TEntity>)_dbContext[typeof(TEntity)];
}

#endregion 私有函数
}

```

4-2 运行你的单元测试

```

[TestClass]
public class ProductUnitTest
{
    [TestMethod]
    public void TestInsertForNewCategory()
    {
        // 初始化仓储
        FakeDbRepository _dbRepository = new FakeDbRepository();

        // 初始化业务对象
        IProductBusiness productBusiness = new ProductBusiness(_dbRepository);

        // 执行测试方法
        productBusiness.InsertForNewCategory("Test Product", "Test Category");

        int _productCount = _dbRepository.GetAll<Product>().Count();
        int _categoryCount = _dbRepository.GetAll<Category>().Count();

        Assert.AreEqual<int>(1, _productCount);
        Assert.AreEqual<int>(1, _categoryCount);
    }

    [TestMethod]
    public void 测试过程函数调用()
    {
        // 初始化过程/函数结果
        Hashtable _contextFunctionsResults = new Hashtable();
    }
}

```

```

        if (_contextFunctionsResults != null && _contextFunctionsResults.Contains(functionName))
            return (TResult)_contextFunctionsResults[functionName];

        throw new NotImplementedException();
    }

#endregion Database Procedures and Functions

#endregion IRepository Implementation

#region IDisposable Implementation

public void Dispose()
{

}

#endregion IDisposable Implementation

#region Private Functions

private List<TEntity> GetDbSet<TEntity>() where TEntity : class
{
    if (!_dbContext.Contains(typeof(TEntity)))
        _dbContext.Add(typeof(TEntity), new List<TEntity>());

    return (List<TEntity>)_dbContext[typeof(TEntity)];
}

#endregion Private Functions
}

```

4-2 Run your unit testing

```

[TestClass]
public class ProductUnitTest
{
    [TestMethod]
    public void TestInsertForNewCategory()
    {
        // Initialize repositories
        FakeDbRepository _dbRepository = new FakeDbRepository();

        // Initialize Business object
        IProductBusiness productBusiness = new ProductBusiness(_dbRepository);

        // Process test method
        productBusiness.InsertForNewCategory("Test Product", "Test Category");

        int _productCount = _dbRepository.GetAll<Product>().Count();
        int _categoryCount = _dbRepository.GetAll<Category>().Count();

        Assert.AreEqual<int>(1, _productCount);
        Assert.AreEqual<int>(1, _categoryCount);
    }

    [TestMethod]
    public void TestProceduresFunctionsCall()
    {
        // Initialize Procedures / Functions result
        Hashtable _contextFunctionsResults = new Hashtable();
    }
}

```

```

_contextFunctionsResults.Add("GetProductsCategory", new List<GetProductsCategory_Result> {
    new GetProductsCategory_Result() { ProductName = "产品 1", ProductID = 1,
    CategoryName = "类别 1" },
    new GetProductsCategory_Result() { ProductName = "产品 2", ProductID = 2,
    CategoryName = "类别 1" },
    new GetProductsCategory_Result() { ProductName = "产品 3", ProductID = 3,
    CategoryName = "类别 1" }});

    // 初始化仓储
FakeDbRepository _dbRepository = new FakeDbRepository(_contextFunctionsResults);

    // 初始化业务对象
IProductBusiness productBusiness = new ProductBusiness(_dbRepository);

    // 执行测试方法
var results = productBusiness.GetProductsCategory(1);

Assert.AreEqual<int>(3, results.Count());
    }
}

```

```

_contextFunctionsResults.Add("GetProductsCategory", new List<GetProductsCategory_Result> {
    new GetProductsCategory_Result() { ProductName = "Product 1", ProductID = 1,
    CategoryName = "Category 1" },
    new GetProductsCategory_Result() { ProductName = "Product 2", ProductID = 2,
    CategoryName = "Category 1" },
    new GetProductsCategory_Result() { ProductName = "Product 3", ProductID = 3,
    CategoryName = "Category 1" }});

    // Initialize repositories
FakeDbRepository _dbRepository = new FakeDbRepository(_contextFunctionsResults);

    // Initialize Business object
IProductBusiness productBusiness = new ProductBusiness(_dbRepository);

    // Process test method
var results = productBusiness.GetProductsCategory(1);

Assert.AreEqual<int>(3, results.Count());
    }
}

```

belindoc.com

第23章：EF中的优化技术

第23.1节：使用AsNoTracking

错误示例：

```
var location = dbContext.Location
    .Where(l => l.Location.ID == location_ID)
    .SingleOrDefault();

return location;
```

由于上述代码仅返回实体而未修改或添加它，我们可以避免跟踪开销。

正确示例：

```
var location = dbContext.Location.AsNoTracking()
    .Where(l => l.Location.ID == location_ID)
    .SingleOrDefault();

return location;
```

当我们使用函数AsNoTracking()时，我们明确告诉实体框架这些实体不会被上下文跟踪。当从数据存储中检索大量数据时，这尤其有用。然而，如果你想对未跟踪的实体进行更改，必须记得在调用SaveChanges之前先附加它们。

第23.2节：尽可能在数据库中执行查询，而不是在内存中

假设我们想统计德克萨斯州有多少个县：

```
var counties = dbContext.States.Single(s => s.Code == "tx").Counties.Count();
```

该查询是正确的，但效率低下。 States.Single(...) 会从数据库加载一个州。接着， Counties 会在第二个查询中加载所有254个县及其所有字段。然后 .Count() 在内存中对加载的Counties 集合执行。

我们加载了很多不需要的数据，实际上可以做得更好：

```
var counties = dbContext.Counties.Count(c => c.State.Code == "tx");
```

这里我们只执行一个查询，在SQL中相当于一个计数和连接。我们只从数据库返回计数——节省了返回行、字段和对象创建的开销。

通过查看集合类型很容易判断查询是在何处执行： IQueryable<T> 与 IEnumerable<T>。

第23.3节：仅加载所需数据

代码中常见的一个问题是加载所有数据。这会大大增加服务器负载。

假设我有一个名为“location”的模型，它包含10个字段，但并非所有字段都需要同时使用。假设我只想要该模型的“LocationName”参数。

错误示例

Chapter 23: Optimization Techniques in EF

Section 23.1: Using AsNoTracking

Bad Example:

```
var location = dbContext.Location
    .Where(l => l.Location.ID == location_ID)
    .SingleOrDefault();

return location;
```

Since the above code is simply returning an entity without modifying or adding it, we can avoid tracking cost.

Good Example:

```
var location = dbContext.Location.AsNoTracking()
    .Where(l => l.Location.ID == location_ID)
    .SingleOrDefault();

return location;
```

When we use function AsNoTracking() we are explicitly telling Entity Framework that the entities are not tracked by the context. This can be especially useful when retrieving large amounts of data from your data store. If you want to make changes to un-tracked entities however, you must remember to attach them before calling SaveChanges.

Section 23.2: Execute queries in the database when possible, not in memory

Suppose we want to count how many counties are there in Texas:

```
var counties = dbContext.States.Single(s => s.Code == "tx").Counties.Count();
```

The query is correct, but inefficient. States.Single(...) loads a state from the database. Next, Counties loads all 254 counties with all of their fields in a second query. .Count() is then performed *in memory* on the loaded Counties collection.

We've loaded a lot of data we don't need, and we can do better:

```
var counties = dbContext.Counties.Count(c => c.State.Code == "tx");
```

Here we only do one query, which in SQL translates to a count and a join. We return only the count from the database - we've saved returning rows, fields, and creation of objects.

It is easy to see where the query is made by looking at the collection type: IQueryable<T> vs. IEnumerable<T>.

Section 23.3: Loading Only Required Data

One problem often seen in code is loading all the data. This will greatly increase the load on the server.

Let's say I have a model called "location" that has 10 fields in it, but not all the fields are required at the same time. Let's say I only want the 'LocationName' parameter of that model.

Bad Example

```
var location = dbContext.Location.AsNoTracking()
    .Where(l => l.Location.ID == location_ID)
    .SingleOrDefault();

return location.Name;
```

正确示例

```
var location = dbContext.Location
    .Where(l => l.Location.ID == location_ID)
    .Select(l => l.LocationName);
    .SingleOrDefault();

return location;
```

“正确示例”中的代码只会获取“LocationName”，不会获取其他字段。

注意，由于此示例中没有实体被实例化，AsNoTracking()并非必要。反正也没有任何内容需要被跟踪。

使用匿名类型获取更多字段

```
var location = dbContext.Location
    .当(l => l.Location.ID == location_ID)
    .选择(l => new { 名称 = l.LocationName, 区域 = l.LocationArea })
    .SingleOrDefault();

返回 location.Name + " 的面积是 " + location.Area;
```

与之前的示例相同，只是从数据库中检索字段“LocationName”和“LocationArea”，匿名类型可以包含任意数量的值。

第23.4节：异步并行执行多个查询

使用异步查询时，可以同时执行多个查询，但不能在同一个上下文中执行。如果一个查询的执行时间是10秒，错误示例的总时间将是20秒，而正确示例的时间将是10秒。

错误示例

```
IEnumerable<TResult1> result1;
IEnumerable<TResult2> result2;

使用(var context = new Context())
{
    result1 = await context.Set<TResult1>().ToListAsync().ConfigureAwait(false);
    result2 = await context.Set<TResult1>().ToListAsync().ConfigureAwait(false);
}
```

正确示例

```
public async Task<IEnumerable<TResult>> GetResult<TResult>()
{
    使用(var context = new Context())
    {
        return await context.Set<TResult1>().ToListAsync().ConfigureAwait(false);
    }
}
```

```
var location = dbContext.Location.AsNoTracking()
    .Where(l => l.Location.ID == location_ID)
    .SingleOrDefault();

return location.Name;
```

Good Example

```
var location = dbContext.Location
    .Where(l => l.Location.ID == location_ID)
    .Select(l => l.LocationName);
    .SingleOrDefault();

return location;
```

The code in the "good example" will only fetch 'LocationName' and nothing else.

Note that since no entity is materialized in this example, AsNoTracking() isn't necessary. There's nothing to be tracked anyway.

Fetching more fields with Anonymous Types

```
var location = dbContext.Location
    .Where(l => l.Location.ID == location_ID)
    .Select(l => new { Name = l.LocationName, Area = l.LocationArea })
    .SingleOrDefault();

return location.Name + " has an area of " + location.Area;
```

Same as the example before, only the fields 'LocationName' and 'LocationArea' will be retrieved from the database, the Anonymous Type can hold as many values you want.

Section 23.4: Execute multiple queries async and in parallel

When using async queries, you can execute multiple queries at the same time, but not on the same context. If the execution time of one query is 10s, the time for the bad example will be 20s, while the time for the good example will be 10s.

Bad Example

```
IEnumerable<TResult1> result1;
IEnumerable<TResult2> result2;

using(var context = new Context())
{
    result1 = await context.Set<TResult1>().ToListAsync().ConfigureAwait(false);
    result2 = await context.Set<TResult1>().ToListAsync().ConfigureAwait(false);
}
```

Good Example

```
public async Task<IEnumerable<TResult>> GetResult<TResult>()
{
    using(var context = new Context())
    {
        return await context.Set<TResult1>().ToListAsync().ConfigureAwait(false);
    }
}
```

```
IEnumerable<TResult1> result1;
IEnumerable<TResult2> result2;

var result1Task = GetResult<TResult1>();
var result2Task = GetResult<TResult2>();

await Task.WhenAll(result1Task, result2Task).ConfigureAwait(false);

var result1 = result1Task.Result;
var result2 = result2Task.Result;
```

第23.5节：使用存根实体

假设我们有Product和Category之间的多对多关系：

```
public class Product
{
    public Product()
    {
        Categories = new HashSet<Category>();
    }
    public int ProductId { get; set; }
    public string ProductName { get; set; }
    public virtual ICollection<Category> Categories { get; private set; }
}

public class Category
{
    public Category()
    {
        Products = new HashSet<Product>();
    }
    public int CategoryId { get; set; }
    public string CategoryName { get; set; }
    public virtual ICollection<Product> Products { get; set; }
}
```

如果我们想给一个Product添加一个Category，我们必须加载该产品并将类别添加到它的Categories中，例如：

错误示例：

```
var product = db.Products.Find(1);
var category = db.Categories.Find(2);
product.Categories.Add(category);
db.SaveChanges();
```

(其中db是一个DbContext子类)。

这将在Product和Category之间的连接表中创建一条记录。然而，该表只包含两个Id值。为了创建一条微小的记录而加载两个完整的实体是资源的浪费。

一种更高效的方法是使用存根实体（stub entities），即在内存中创建的实体对象，只包含最基本的数据，通常只有一个Id值。示例如下：

好例子：

```
// 创建两个存根实体
```

```
IEnumerable<TResult1> result1;
IEnumerable<TResult2> result2;

var result1Task = GetResult<TResult1>();
var result2Task = GetResult<TResult2>();

await Task.WhenAll(result1Task, result2Task).ConfigureAwait(false);

var result1 = result1Task.Result;
var result2 = result2Task.Result;
```

Section 23.5: Working with stub entities

Say we have Products and Categorys in a many-to-many relationship:

```
public class Product
{
    public Product()
    {
        Categories = new HashSet<Category>();
    }
    public int ProductId { get; set; }
    public string ProductName { get; set; }
    public virtual ICollection<Category> Categories { get; private set; }
}

public class Category
{
    public Category()
    {
        Products = new HashSet<Product>();
    }
    public int CategoryId { get; set; }
    public string CategoryName { get; set; }
    public virtual ICollection<Product> Products { get; set; }
}
```

If we want to add a Category to a Product, we have to load the product and add the category to its Categories, for example:

Bad Example:

```
var product = db.Products.Find(1);
var category = db.Categories.Find(2);
product.Categories.Add(category);
db.SaveChanges();
```

(where db is a DbContext subclass).

This creates one record in the junction table between Product and Category. However, this table only contains two Id values. It's a waste of resources to load two full entities in order to create one tiny record.

A more efficient way is to use *stub entities*, i.e. entity objects, created in memory, containing only the bare minimum of data, usually only an Id value. This is what it looks like:

Good example:

```
// Create two stub entities
```

```
var product = new Product { ProductId = 1 };
var category = new Category { CategoryId = 2 };

// 将根实体附加到上下文
db.Entry(product).State = System.Data.Entity.EntityState.Unchanged;
db.Entry(category).State = System.Data.Entity.EntityState.Unchanged;

product.Categories.Add(category);
db.SaveChanges();
```

最终结果相同，但避免了两次往返数据库。

防止重复

如果你想检查关联是否已存在，一个简单的查询就足够了。例如：

```
var exists = db.Categories.Any(c => c.Id == 1 && c.Products.Any(p => p.Id == 14));
```

同样，这不会将完整的实体加载到内存中。它实际上是查询连接表并仅返回一个布尔值。

第23.6节：禁用变更跟踪和代理生成

如果您只是想获取数据，而不修改任何内容，可以关闭变更跟踪和代理创建。这将提升性能，同时防止延迟加载。

错误示例：

```
使用(var context = new Context())
{
    return await context.Set<MyEntity>().ToListAsync().ConfigureAwait(false);
}
```

正确示例：

```
使用(var context = new Context())
{
    context.Configuration.AutoDetectChangesEnabled = false;
    context.Configuration.ProxyCreationEnabled = false;

    return await context.Set<MyEntity>().ToListAsync().ConfigureAwait(false);
}
```

通常会在上下文的构造函数中关闭这些设置，尤其是当您希望在整个解决方案中应用这些设置时：

```
public class MyContext : DbContext
{
    public MyContext()
        : base("MyContext")
    {
        Configuration.AutoDetectChangesEnabled = false;
        Configuration.ProxyCreationEnabled = false;
    }

    //snip
}
```

```
var product = new Product { ProductId = 1 };
var category = new Category { CategoryId = 2 };

// Attach the stub entities to the context
db.Entry(product).State = System.Data.Entity.EntityState.Unchanged;
db.Entry(category).State = System.Data.Entity.EntityState.Unchanged;

product.Categories.Add(category);
db.SaveChanges();
```

The end result is the same, but it avoids two roundtrips to the database.

Prevent duplicates

It you want to check if the association already exists, a cheap query suffices. For example:

```
var exists = db.Categories.Any(c => c.Id == 1 && c.Products.Any(p => p.Id == 14));
```

Again, this won't load full entities into memory. It effectively queries the junction table and only returns a boolean.

Section 23.6: Disable change tracking and proxy generation

If you just want to get data, but not modify anything, you can turn off change tracking and proxy creation. This will improve your performance and also prevent lazy loading.

Bad Example:

```
using(var context = new Context())
{
    return await context.Set<MyEntity>().ToListAsync().ConfigureAwait(false);
}
```

Good Example:

```
using(var context = new Context())
{
    context.Configuration.AutoDetectChangesEnabled = false;
    context.Configuration.ProxyCreationEnabled = false;

    return await context.Set<MyEntity>().ToListAsync().ConfigureAwait(false);
}
```

It is particularly common to turn these off from within the constructor of your context, especially if you wish these to be set across your solution:

```
public class MyContext : DbContext
{
    public MyContext()
        : base("MyContext")
    {
        Configuration.AutoDetectChangesEnabled = false;
        Configuration.ProxyCreationEnabled = false;
    }

    //snip
}
```

致谢

非常感谢所有来自Stack Overflow Documentation的人员提供此内容，
更多更改可发送至web@petercv.com以发布或更新新内容

阿迪尔·马马多夫	第1、7和16章
阿科什·纳吉	第8、9和21章
安舒尔·尼甘	第23章
CptRobby	第3、11和14章
丹尼尔·莱姆克	第3章和第7章
大卫·G	第1章、第3章和第23章
迭戈	第3章
埃尔多	第1章
弗洛里安·海德	第16章
格特·阿诺德	第3、11、14、15和23章
雅各布·林尼	第1章
杰森·泰勒	第7和19章
乔希特	第16章
约瑟夫·拉čný	第3、4和12章
科比	第23章
lucavgobbi	第6和23章
MacakM	第2章
马克·谢甫琴科	第3章
马塔斯·瓦伊特凯维休斯	第1、3、10和20章
米娜·马塔	第22章
莫斯塔法	第5章
纳斯雷丁	第1章
帕尔斯·帕特尔	第1、2和3章
皮奥特雷克	第3章
桑帕斯	第13章
skj123	第18章
SOfanatic	第17章
斯蒂芬·赖因德尔	第5章
山本哲也	第10章和第20章
tmg	第1、2、3和13章
图沙尔·帕特尔	第3章
wertzui	第23章

Credits

Thank you greatly to all the people from Stack Overflow Documentation who helped provide this content,
more changes can be sent to web@petercv.com for new content to be published or updated

Adil Mammadov	Chapters 1, 7 and 16
Akos Nagy	Chapters 8, 9 and 21
Anshul Nigam	Chapter 23
CptRobby	Chapters 3, 11 and 14
Daniel Lemke	Chapters 3 and 7
DavidG	Chapters 1, 3 and 23
Diego	Chapter 3
Eldho	Chapter 1
Florian Haider	Chapter 16
Gert Arnold	Chapters 3, 11, 14, 15 and 23
Jacob Linney	Chapter 1
Jason Tyler	Chapters 7 and 19
Joshit	Chapter 16
Jozef Lačný	Chapters 3, 4 and 12
Kobi	Chapter 23
lucavgobbi	Chapters 6 and 23
MacakM	Chapter 2
Mark Shevchenko	Chapter 3
Matas Vaitkevicius	Chapters 1, 3, 10 and 20
Mina Matta	Chapter 22
Mostafa	Chapter 5
Nasreddine	Chapter 1
Parth Patel	Chapters 1, 2 and 3
Piotrek	Chapter 3
Sampath	Chapter 13
skj123	Chapter 18
SOfanatic	Chapter 17
Stephen Reindl	Chapter 5
Tetsuya Yamamoto	Chapters 10 and 20
tmg	Chapters 1, 2, 3 and 13
Tushar patel	Chapter 3
wertzui	Chapter 23

你可能也喜欢



You may also like

