# Code First 约定

借助 Code First，可通过使用 C# 或Visual Basic .NET 类来描述模型。模型的基本形状可通过约定来检测。约定是规则集，用于在使用 Code First 时基于类定义自动配置概念模型。约定是在 System.Data.Entity.ModelConfiguration.Conventions 命名空间中定义的。

可通过使用数据注释或Fluent API 进一步配置模型。优先级是通过 Fluent API 进行配置，然后是通过数据注释，再次是通过约定。

API 文档中提供了 Code First 约定的详细列表。本主题概述 Code First 使用的约定。

## 类型发现

当使用 CodeFirst 开发时，通常是从编写用来定义概念（域）模型的 .NET类开始。除了定义类之外，还需要让 DbContext 知道模型中要包含哪些类。为此，需要定义一个上下文类，此类派生自 DbContext 并公开需要成为模型一部分的类型的 DbSet 属性。Code First 将包含这些类型，还将包含任何引用类型，即使这些引用类型是在不同的程序集中定义的也是如此。

如果类型存在于继承层次结构中，则为基类定义 DbSet 属性就足够了，如果派生类型位于与基类相同的程序集中，则自动包含这些派生类型。

在下面的示例中，仅对SchoolEntities 类定义一个DbSet 属性 (Departments)。CodeFirst 使用此属性来发现并包含任何引用类型。

隐藏代码

```
public class  SchoolEntities: DbContext
{
  public DbSet<Department>Departments { get; set;}
}

public class  Department
{

  // Primary key
  public int DepartmentID { get;set; }
  public string Name { get; set; }

  // Navigationproperty
  public virtual ICollection<Course> Courses { get;set; }
}



public class  Course
{

  // Primary key
  public int CourseID { get; set; }
  public string Title { get; set; }
  public int Credits { get; set; }

  // Foreign key
  public int DepartmentID { get;set; }

  // Navigationproperties
  public virtual DepartmentDepartment {  get; set;}
}

public partial  class OnlineCourse :  Course
{
  public string URL { get; set; }
}

public partial  class OnsiteCourse :  Course
{
  public string Location { get;set; }
  public string Days { get; set; }
  public System.DateTime Time { get; set; }
}
```

如果要从模型排除类型，请使用 NotMapped 特性或DbModelBuilder.Ignore

## 主键约定

如果类的属性名为“ID”（不区分大小写）或类名的后面跟有“ID”，则 Code First 会推断该属性是主键。如果主键属性的类型为数值或 GUID，则将其配置为标识列。

隐藏代码

```
public class  Department
{
  // Primary key
  public int DepartmentID { get;set; }
}
```

## 关系约定

实体框架中的导航属性提供了一种在两个实体类型之间导航关系的方法。针对对象参与到其中的每个关系，各对象均可以具有导航属性。使用导航属性，可以在两个方向上导航和管理关系，返回引用对象（如果多重性为一或者零或一）或集合（如果多重性为多）。Code First 根据针对类型定义的导航属性来推断关系。

除导航属性外，建议还要包括表示依赖对象的类型的外键属性。任何数据类型与主体主键属性相同、遵循以下一种格式的属性都表示关系的外键：“<导航属性名称><主体主键属性>”、“<主体类名><主键属性名称>”或“<主体主键属性名称>”。如果找到多个匹配项，则优先级符合上面列出的顺序。外键检测不区分大小写。在检测外键属性时，Code First 基于外键的可空性推断关系的多重性。如果属性可以为 Null，则将关系注册为可选关系；否则，将关系注册为必需关系。

如果依赖实体上的外键不能为 Null，则 CodeFirst 对关系设置级联删除。如果依赖实体上的外键可以为 Null，则Code First 不对关系设置级联删除，并且在删除主体时，会将该外键设置为 Null。通过使用 Fluent API，可以覆盖由约定检测的多重性和级联删除行为。

隐藏代码

```
public class  Department
{
  // Primary key
  public int DepartmentID { get;set; }
  public string Name { get; set; }

  // Navigationproperty
  public virtual ICollection<Course> Courses { get;set; }
}

public class  Course
{
  // Primary key
  public int CourseID { get; set; }

  public string Title { get; set; }

  public int Credits { get; set; }

  // Foreign key
  public int DepartmentID { get;set; }

  // Navigationproperties
  public virtual DepartmentDepartment {  get; set;}
}
```

在下面的示例中，导航属性和外键用于定义 Department 类与Course 类之间的关系。

**注意：**如果相同类型间有多个关系（例如，假设定义 Person 和Book 类，其中，Person 包含ReviewedBooks 和AuthoredBooks 导航属性，而Book 类包含 Author 和Reviewer 导航属性），则需要使用数据注释或 Fluent API 手动配置关系。

## 复杂类型约定

当 CodeFirst 发现无法推断主键以及未通过数据注释或 Fluent API 注册主键的类时，类型会自动注册为复杂类型。复杂类型检测还要求类型不具有引用实体类型的属性，并且未被其他类型的集合属性引用。对于以下类定义，Code First 推断Details 是复杂类型，因为它没有主键。

隐藏代码

```
public partial  class OnsiteCourse :  Course
{
  public OnsiteCourse()
  {
    Details = newDetails();
  }
  public Details Details { get;set; }

}



public class  Details
{
  public System.DateTime Time { get; set; }

  public string Location { get;set; }

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

# 数据库配置约定

## 连接字符串约定

### 默认配置

如果您还没有在应用程序中进行任何其他配置，则对 DbContext 调用无参数构造函数将会导致 DbContext 使用按约定创建的数据库连接在 Code First 模式下运行。例如：

隐藏代码

```
namespaceDemo.EF
{
  public class BloggingContext: DbContext
  {
    publicBloggingContext()

    // C# will callbase class parameterless constructor by default
    {

    }
  }
}
```

在此示例中，DbContext使用派生上下文类 Demo.EF.BloggingContext 的命名空间限定名称作为数据库名称，并使用 SQL Express 或 LocalDb 为此数据库创建连接字符串。如果同时安装了这两个数据库，将使用 SQL Express。

默认情况下，VisualStudio 2010 包含 SQLExpress，VisualStudio 2012 包含LocalDb。安装期间，EntityFrameworkNuGet 包会检查哪个数据库服务器可用。随后 NuGet 包将设置按约定创建连接时 Code First 所使用的默认数据库服务器，以此更新配置文件。如果 SQL Express 正在运行，将使用它。如果 SQL Express 不可用，则 LocalDb 将注册为默认数据库。如果配置文件已包含默认连接工厂设置，则不会更改该文件。

### 指定数据库名称

如果您尚未在应用程序中进行任何其他配置，在通过要使用的数据库名称对 DbContext 调用字符串构造函数时，将会导致 DbContext 使用按约定创建的与该名称数据库的连接在 Code First 模式下运行。例如：

隐藏代码

```
namespaceDemo.EF
{
  public class BloggingContext: DbContext
  {
    public BloggingContext()
      : base("BloggingDatabase")
    {

    }
  }
}
```

在此示例中，DbContext使用“BloggingDatabase”作为数据库名称，并使用 SQL Express（随Visual Studio 2010 安装）或 LocalDb（随Visual Studio 2012 安装）为此数据库创建连接字符串。如果同时安装了这两个数据库，将使用 SQL Express。

### 指定连接字符串

可以选择将连接字符串放入 app.config 或web.config 文件中。例如：

隐藏代码

```
<configuration>
  <connectionStrings>
    <addname="BloggingCompactDatabase"
         providerName="System.Data.SqlServerCe.4.0"
         connectionString="Data Source=Blogging.sdf"/>
  </connectionStrings>
</configuration>
```

这是一种指示 DbContext 使用数据库服务器而非 SQL Express 或LocalDb 的简单方法 — 上例指定了 SQL Server Compact Edition 数据库。

如果连接字符串的名称与上下文的名称（带或不带命名空间限定）相同，则使用无参数构造函数时 DbContext 会找到该连接字符串。如果连接字符串名称与上下文名称不同，则可通过将连接字符串名称传递给 DbContext 构造函数，指示 DbContext 在 CodeFirst 模式下使用此连接。例如：

隐藏代码

```
public class BloggingContext: DbContext
{
  public BloggingContext()
```

```
    : base("BloggingCompactDatabase")
  {

  }
}
```

或者，也可以对传递给DbContext 构造函数的字符串使用 “name=<连接字符串名称>”格式。例如：

隐藏代码

```
public class BloggingContext: DbContext
{
  publicBloggingContext()
    : base("name=BloggingCompactDatabase")
  {

  }
}
```

使用此形式可以明确要求在配置文件中查找连接字符串。如果未找到具有给定名称的连接字符串，则将引发异常。

## 数据库初始化策略：

数据库创建是由策略来控制的，有如下四种策略：

1. CreateDatabaseIfNotExists：这是默认的策略。如果数据库不存在，那么就创建数据库。但是如果数据库存在了，而且实体发生了变化，就会出现异常。

2. DropCreateDatabaseIfModelChanges：此策略表明，如果模型变化了，数据库就会被重新创建，原来的数据库被删除掉了。

3. DropCreateDatabaseAlways：此策略表示，每次运行程序都会重新创建数据库，这在开发和调试的时候非常有用。

4. 自定制数据库策略：可以自己实现IDatabaseInitializer来创建自己的策略。或者从已有的实现了IDatabaseInitializer接口的类派生。

如下示例显示了如何应用数据库创建策略：

隐藏代码

```
public class  UserManContext : DbContext
{
    public UserManContext()
        : base("USMDBConnectionString")
    {
        Database.SetInitializer<UserManContext>(new CreateDatabaseIfNotExists<UserManContext>());
    }
}
```

下面的代码创建了一个自定义策略，什么也没有做，但是我们可以在Seed方法里添加我们的种子数据。

隐藏代码

```
public class  USMDBInitializer : DropCreateDatabaseAlways<UserManContext>
{
    protected override void Seed(UserManContext context)
    {
        base.Seed(context);
    }
}
```

虽然EF提供了在配置文件中配置策略的方法，如下所示：

隐藏代码

```
<appSettings>
    <addkey="DatabaseInitializerForType EFCodeFirstSample.UserManContext, EFCodeFirstSample"
        value="System.Data.Entity.DropCreateDatabaseAlways`1[[EFCodeFirstSample.UserManContext,EFCodeFirstSample]],
  </appSettings>
```

Key必须以DatabaseInitializerForType开始，后边加空格，之后是context类的全名称，包括带命名空间的类名和所在的程序集名。Value是策略的全名称。可以看见key和value都非常难读，还不如自己写配置来的好。

如果不想使用策略，就可以关闭策略，特别是默认策略。关闭策略的代码如下：

隐藏代码

```
public class  UserManContext : DbContext
{
    public UserManContext()
        : base("USMDBConnectionString")
    {
        Database.SetInitializer<UserManContext>(null);
    }
}
```

还可以在配置文件中关闭策略，如下：

隐藏代码

```
<addkey="DatabaseInitializerForTypeEFCodeFirstSample.UserManContext, EFCodeFirstSample" value="Disabled" />
```

### 为数据库添加种子数据

上面提高可以在自定义数据库初始化策略中添加种子数据，下面的示例说明如何添加种子数据：

隐藏代码

```
public class  USMDBInitializer : DropCreateDatabaseAlways<UserManContext>
{
    protected override void Seed(UserManContext context)
    {
        User admin =  new User();
        admin.Name = "admin";
        admin.DisplayName = "Administrator";
        admin.Status = 1;
        admin.LastModDate= DateTime.Now;
        context.Users.Add(admin);
        base.Seed(context);
    }
}
```

需要注意的是日期字段，数据库中的日期范围小于.NET中的日期范围，所以必须给一个合适的值，像DateTime.MinValue这样的值无法存储到数据库中。可以参考SqlDateTime类型来确定Sql数据库支持的时间范围。

## 移除约定

可以移除在System.Data.Entity.ModelConfiguration.Conventions 命名空间中定义的任何约定。下面的示例移除PluralizingTableNameConvention。

隐藏代码

```
publicclass  SchoolEntities: DbContext
{
  protected override voidOnModelCreating(DbModelBuilder modelBuilder)
  {
    // Configure CodeFirst to ignore PluralizingTableName convention
    // If you keepthis convention, the generated tables
    // will havepluralized names.
    modelBuilder.Conventions.Remove<PluralizingTableNameConvention>();
  }
}
```

## 可插入约定

可插入（自定义）约定目前不受支持，正在针对 EF6 进行开发。

# Code First 数据特性设置

通过实体框架Code First，可以使用您自己的域类表示 EF 执行查询、更改跟踪和更新函数所依赖的模型。Code First 利用称为“约定先于配置”的编程模式。这就是说，Code First 将假定您的类遵从 EF 所使用的约定。在这种情况下，EF 将能够找出自己工作所需的详细信息。但是，如果您的类不遵守这些约定，则可以向类中添加配置，以向 EF 提供它需要的信息。

Code First 为您提供了两种方法来向类中添加这些配置。一种方法是使用名为 DataAnnotations 的简单特性，另一种方法是使用 Code First 的Fluent API，该 API 向您提供了在代码中以命令方式描述配置的方法。

本文重点介绍如何使用DataAnnotations（在System.ComponentModel.DataAnnotations 命名空间中）对类进行配置，着重讲述常用的配置。很多 .NET 应用程序（如 ASP.NET MVC）都能够理解DataAnnotations，它允许这些应用程序对客户端验证使用相同的注释。

我将通过Blog 和 Post 这两个简单的类来说明 Code First DataAnnotations。

隐藏代码

```
public class  Blog
{
  public int Id { get; set; }
  public string Title { get; set; }
  public string BloggerName { get;set; }
  public virtual ICollection<Post> Posts { get;set; }
}
```

隐藏代码

```
public class  Blog
{
  public int Id { get; set; }
  public string Title { get; set; }
  public string BloggerName { get;set; }
  public virtual ICollection<Post> Posts { get;set; }
}
```

Blog 和 Post 类本身就遵守 Code First 约定，无需调整即可让EF 与之共同使用。但您也可以使用注释向 EF 提供有关类以及类所映射到的数据库的更多信息。

## 键

实体框架依赖于每个具有键值的实体，它使用键值来跟踪实体。Code First 依赖的一个约定是它在每一个 Code First 类中以何种方式表示哪一个属性是键。该约定是查找名为“Id”或类名与“Id”组合在一起（如“BlogId”）的属性。该属性将映射到数据库中的主键列。

Blog 和 Post 类都遵守此约定。但如果它们不遵守呢？如果 Blog 使用名称 PrimaryTrackingKey，甚至使用 foo 呢？如果 Code First 找不到符合此约定的属性，它将引发异常，因为实体框架要求必须要有一个键属性。您可以使用键注释来指定要将哪一个属性用作 EntityKey。

隐藏代码

```
public class  Blog
{
  [Key]
  public int PrimaryTrackingKey { get;set; }
  public string Title { get; set; }
  public string BloggerName { get;set; }
  public virtual ICollection<Post> Posts { get;set; }
}
```

如果您在使用Code First 的数据库生成功能，则Blog 表将具有名为 PrimaryTrackingKey 的主键列，该列默认情况下还定义为 Identity。

## 必需

Required 注释告诉 EF 某一个特定属性是必需的。

在 Title 属性中添加 Required 将强制 EF（和 MVC）确保该属性中包含数据。

隐藏代码

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

Required 特性将使被映射的属性不可为空来影响生成的数据库。请注意，Title 字段已经更改为“not null”。

## MaxLength 和MinLength

使用MaxLength 和MinLength 特性，您可以就像对Required 那样指定其他属性验证。

下面是具有长度要求的BloggerName。该示例也说明如何组合特性。

隐藏代码

```
[MaxLength(10),  MinLength(5)]
public string BloggerName { get;set; }
```

MaxLength 注释将通过把属性长度设置为 10 来影响数据库。MinLength属性不会对数据库产生影响。

## NotMapped

Code First 约定指示具有受支持数据类型的每个属性都要在数据库中有表示。但在您的应用程序中并不总是如此。例如，您可以在 Blog 类中使用一个属性来基于 Title 和BloggerName 字段创建代码。该属性可以动态创建，无需存储。您可以使用 NotMapped 注释来标记不映射到数据库的所有属性，如下面的 BlogCode 属性。

隐藏代码

```
[NotMapped]
public string BlogCode
{
  get
  {
    returnTitle.Substring(0, 1) +  ":" +BloggerName.Substring(0, 1);
  }
}
```

## ComplexType

跨一组类描述域实体，然后将这些类分层以描述一个完整实体的情况并不少见。例如，您可以向模型中添加一个名为 BlogDetails 的类。

public class BlogDetails { public DateTime? DateCreated { get;set; } [MaxLength(250)] public string Description { get;set; } }

请注意，BlogDetails 没有任何键属性类型。在域驱动的设计中，BlogDetails 称为值对象。实体框架将值对象称为复杂类型。复杂类型不能自行跟踪。

但是 BlogDetails 作为 Blog 类中的一个属性，将作为 Blog 对象的一部分被跟踪。为了让 Code First 认识到这一点，您必须将 BlogDetails 类标记为 ComplexType。

隐藏代码

```
[ComplexType]
public class  BlogDetails
{
  public DateTime? DateCreated { get;set; }
  [MaxLength(250)]
  public string Description { get;set; }

}
```

现在，您可以在Blog 类中添加一个属性来表示该博客的 BlogDetails。

隐藏代码

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

在数据库中，Blog表将包含该博客的所有属性，包括在其 BlogDetail 属性中所含的属性。默认情况下，每个属性都将添加复杂类型名称前缀 BlogDetail。

另外，有趣的是，虽然DateCreated 属性在类中定义为不可为空的 DateTime，但相关数据库字段是可为空的。如果想影响数据库架构，则必须使用 Required 注释。

## ConcurrencyCheck

ConcurrencyCheck 注释可用于标记要在用户编辑或删除实体时用于在数据库中进行并发检查的一个或多个属性。如果之前使用 EF 设计器，则这等同于将属性的 ConcurrencyMode 设置为 Fixed。

现在让我们将ConcurrencyCheck 添加到BloggerName 属性，看看它如何工作。

隐藏代码

```
[ConcurrencyCheck,  MaxLength(10),MinLength(5)]
public string BloggerName { get;set; }
```

调用SaveChanges 时，因为BloggerName 字段上具有ConcurrencyCheck 注释，所以在更新中将使用该属性的初始值。该命令将尝试通过同时依据键值和 BloggerName 的初始值进行筛选来查找正确的行。下面是发送到数据库的 UPDATE 命令的关键部分，在其中您可以看到该命令将更新 PrimaryTrackingKey 为 1 且BloggerName 为“Julie”（这是从数据库中检索到该博客时的初始值）的行。

```
where (([PrimaryTrackingKey]= @4) and([BloggerName] = @5)) @4=1,@5=N'Julie'
```

如果在此期间有人更改了该博客的博主姓名，则此更新将失败，并引发 DbUpdateConcurrencyException 并且需要处理该异常。

## TimeStamp

使用rowversion 或timestamp 字段来进行并发检查更为常见。但是比起使用 ConcurrencyCheck 注释，只要属性类型为字节数组，则不如使用更为具体的 TimeStamp 注释。Code First 将Timestamp 属性与ConcurrencyCheck 属性同等对待，但它还将确保 Code First 生成的数据库字段是不可为空的。在一个指定类中，只能有一个 timestamp 属性。

将以下属性添加到Blog 类：

隐藏代码

```
[Timestamp]
public Byte[] TimeStamp { get;set; }
```

这样，CodeFirst 将在数据库表中创建一个不可为空的 Timestamp 列。

## 表和列

如果您让Code First 创建数据库，则可能希望更改它创建的表和列的名称。也可以将 Code First 用于现有数据库。但是域中的类和属性的名称并不总是与数据库中表和列的名称相匹配。

我的类名为Blog，按照约定，Code First 将假定此类映射到名为 Blogs 的表。如果不是这样，您可以用 Table 特性指定该表的名称。举例来说，下面的注释指定表名称为 InternalBlogs，同时指定了schema，默认的schema就是dbo。

隐藏代码

```
[Table("InternalBlogs",Schema="dbo")]
public class  Blog
```

Column 注释更适于用来指定被映射列的特性。您可以规定名称、数据类型甚至列出现在表中的顺序。下面是 Column 特性的示例。

隐藏代码

```
[Column("BlogDescription",TypeName = "ntext")]
public String Description { get; set; }
```

下面是重新生成后的表。表名称已更改为 InternalBlogs，复杂类型的 Description 列现在是BlogDescription。因为该名称在注释中指定，Code First 不会使用以复杂类型名称作为列名开头的约定。

## DatabaseGenerated

一个重要的数据库功能是可以使用计算属性。如果您将 Code First 类映射到包含计算列的表，则您可能不想让实体框架尝试更新这些列。但是在插入或更新数据后，您的确需要 EF 从数据库中返回这些值。您可以使用 DatabaseGenerated 注释与 Computed 枚举一起在您的类中标注这些属性。其他枚举为 None 和Identity。

隐藏代码

```
[DatabaseGenerated(DatabaseGeneratedOption.Computed)]
public DateTime DateCreated { get; set; }
```

当 Code First生成数据库时，您可以对 byte 或timestamp 列使用此标记，否则您只应该在指向现有数据库时使用，因为 Code First 将不能确定计算列的公式。

您阅读过以上内容，知道默认情况下，整数键属性将成为数据库中的标识键。这与将 DatabaseGenerated 设置为 DatabaseGenerationOption.Identity 是一样的。如果不希望它成为标识键，则可以将该值设置为 DatabaseGenerationOption.None。

## 关系特性：InverseProperty和ForeignKey

Code First 约定将在您的模型中处理最常用的关系，但是在某些情况下它需要帮助。

在 Blog 类中更改键属性的名称造成它与 Post 的关系出现问题。

生成数据库时，CodeFirst 会在 Post 类中看到 BlogId 属性并识别出该属性，按照约定，它与类名加“Id”匹配，并作为 Blog 类的外键。但是在此Blog 类中没有 BlogId 属性。解决方法是，在 Post 中创建一个导航属性，并使用 Foreign DataAnnotation 来帮助 CodeFirst 了解如何在两个类之间创建关系（那就是使用 Post.BlogId 属性）以及如何在数据库中指定约束。

隐藏代码

```
public class  Post
{
  public int Id { get; set; }
  public string Title { get; set; }
  public DateTime DateCreated { get;set; }
  public string Content { get;set; }
  public int BlogId { get; set; }
  [ForeignKey("BlogId")]
  public Blog Blog { get; set; }
}
```

数据库中的约束显示InternalBlogs.PrimaryTrackingKey 与Posts.BlogId 之间的关系。

类之间存在多个关系时，将使用 InverseProperty。

在 Post 类中，您可能需要跟踪是谁撰写了博客文章以及谁编辑了它。下面是 Post 类的两个新的导航属性。

隐藏代码

```
public Person CreatedBy { get;set; }
public Person UpdatedBy { get;set; }
```

您还需要在这些属性引用的 Person 类中添加内容。Person类具有返回到 Post 的导航属性，一个属性指向该用户撰写的所有文章，一个属性指向该用户更新的所有文章。

隐藏代码

```
public class  Person
{
  public int Id { get; set; }
  public string Name { get; set; }
  public List<Post>PostsWritten { get; set;}
  public List<Post>PostsUpdated { get; set;}
}
```

Code First 不能自行使这两个类中的属性匹配。Posts 的数据库表应该有一个表示 CreatedBy 人员的外键，有一个表示 UpdatedBy 人员的外键，但是 Code First 将创建四个外键属性：Person_Id、Person_Id1、CreatedBy_Id 和UpdatedBy_Id。(针对每个导航属性创建一个外键)

要解决这些问题，您可以使用 InverseProperty 注释来指定这些属性的匹配。

隐藏代码

```
[InverseProperty("CreatedBy")]
publicList<Post>PostsWritten { get; set;}

[InverseProperty("UpdatedBy")]
publicList<Post>PostsUpdated { get; set;}
```

因为Person 中的PostsWritten 属性知道这指的是Post 类型，所以它将与 Post.CreatedBy 建立关系。同样，PostsUpdated 也将与 Post.UpdatedBy 建立关系。Code First 不会创建额外的外键。

## 总结

DataAnnotations 不仅可用于在 Code First 类中描述客户端和服务器端验证，还让您能够加强甚至更正 Code First 将基于其约定对您的类所作的假设。使用 DataAnnotations，您不仅能够推动数据库架构生成，还能将 Code First 类映射到预先存在的数据库。

虽然它们都非常灵活，但请记住，DataAnnotations 只提供您经常需要对 Code First 类进行的配置更改。要为一些边缘情况配置类，则应该采用另一种替代配置机制，那就是 Code First 的Fluent API。

# 使用Fluent API 配置/映射属性和类型

## 简介

通常通过重写派生DbContext 上的OnModelCreating 方法来访问Code First Fluent API。以下示例旨在显示如何使用 Fluent API 执行各种任务，您可以将代码复制出来并进行自定义，使之适用于您的模型。

## 属性映射

Property 方法用于为每个属于实体或复杂类型的属性配置特性。Property 方法用于获取给定属性的配置对象。配置对象上的选项特定于要配置的类型；例如，IsUnicode 只能用于字符串属性。

## 配置主键

要显式将某个属性设置为主键，可使用 HasKey 方法。在以下示例中，使用了 HasKey 方法对 OfficeAssignment 类型配置 InstructorID 主键。

隐藏代码

```
modelBuilder.Entity<OfficeAssignment>().HasKey(t =>t.InstructorID);
```

## 配置组合主键

以下示例配置要作为Department 类型的组合主键的DepartmentID 和 Name 属性。

隐藏代码

```
modelBuilder.Entity<Department>().HasKey(t => new { t.DepartmentID, t.Name });
```

## 关闭数值主键的标识

以下示例将DepartmentID 属性设置为System.ComponentModel.DataAnnotations.DatabaseGeneratedOption.None，以指示该值不由数据库生成。

隐藏代码

```
modelBuilder.Entity < Department > ().Property(t = >t.DepartmentID).HasDatabaseGeneratedOption(DatabaseGeneratedOpt
```

## 指定属性的最大长度

在以下示例中，Name属性不应超过 50 个字符。如果其值超过 50 个字符，则出现 DbEntityValidationException 异常。如果 Code First 基于此模型创建数据库，它还会将 Name 列的最大长度设置为50 个字符。

隐藏代码

```
modelBuilder.Entity<Department>().Property(t =>t.Name).HasMaxLength(50);
```

## 将属性配置为必需

在下面的示例中，Name属性是必需的。如果不指定 Name，则出现 DbEntityValidationException 异常。如果 Code First 基于此模型创建数据库，则用于存储此属性的列将不可为空。

隐藏代码

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

## 指定不将CLR 属性映射到数据库中的列

以下示例显示如何指定CLR 类型的属性不映射到数据库中的列。

隐藏代码

```
modelBuilder.Entity<Department>().Ignore(t => t.Budget);
```

## 将CLR 属性映射到数据库中的特定列

以下示例将Name CLR 属性映射到DepartmentName 数据库列。

隐藏代码

```
modelBuilder.Entity<Department>().Property(t =>t.Name).HasColumnName("DepartmentName");
```

## 重命名模型中未定义的外键

如果您选择不对CLR 类型定义外键，但希望指定它在数据库中应使用的名称，请编码如下：

隐藏代码

```
modelBuilder.Entity<Course>()
        .HasRequired(c => c.Department)
        .WithMany(t => t.Courses)
        .Map(m => m.MapKey("ChangedDepartmentID"));
```

## 配置字符串属性是否支持Unicode 内容

默认情况下，字符串为Unicode（SQLServer 中的nvarchar）。您可以使用IsUnicode 方法指定字符串应为varchar 类型。

隐藏代码

```
modelBuilder.Entity<Department>()
        .Property(t => t.Name)
        .IsUnicode(false);
```

## 配置数据库列的数据类型

HasColumnType 方法支持映射到相同基本类型的不同表示。使用此方法并不支持在运行时执行任何数据转换。请注意，IsUnicode 是将列设置为 varchar 的首选方法，因为它与数据库无关。

隐藏代码

```
modelBuilder.Entity<Department>()
        .Property(p => p.Name)
        .HasColumnType("varchar");
```

## 配置复杂类型的属性

对复杂类型配置标量属性有两种方法。

可以对ComplexTypeConfiguration 调用Property。

隐藏代码

```
modelBuilder.ComplexType<Details>()
        .Property(t => t.Location)
        .HasMaxLength(20);
```

也可以使用点表示法访问复杂类型的属性。

隐藏代码

```
modelBuilder.ComplexType<Details>()
        .Property(t => t.Location)
        .HasMaxLength(20);
```

## 将属性配置为用作乐观并发令牌

要指定实体中的某个属性表示并发令牌，可使用 ConcurrencyCheck 特性或 IsConcurrencyToken 方法。

隐藏代码

```
modelBuilder.Entity<OfficeAssignment>()
        .Property(t => t.Timestamp)
        .IsConcurrencyToken();
```

也可以使用IsRowVersion 方法将属性配置为数据库中的行版本。将属性设置为行版本会自动将它配置为乐观并发令牌。

隐藏代码

```
modelBuilder.Entity<OfficeAssignment>()
        .Property(t => t.Timestamp)
        .IsRowVersion();
```

## 类型映射

## 将类指定为复杂类型

按约定，没有指定主键的类型将被视为复杂类型。在一些情况下，Code First 不会检测复杂类型（例如，如果您有名为“ID”的属性，但不想将它用作主键）。在此类情况下，您将使用 Fluent API 显式指定某类型是复杂类型。

隐藏代码

```
modelBuilder.ComplexType<Details>();
```

## 指定不将CLR 实体类型映射到数据库中的表

以下示例显示如何排除一个 CLR 类型，使之不映射到数据库中的表。

隐藏代码

```
modelBuilder.Ignore<OnlineCourse>();
```

## 将CLR 实体类型映射到数据库中的特定表

Department 的所有属性都将映射到名为 t_ Department 的表中的列。

隐藏代码

```
modelBuilder.Entity().ToTable("t_Department");
```

您也可以这样指定架构名称：

```
modelBuilder.Entity().ToTable("t_Department", "school");
```

## 映射“每个层次结构一张表(TPH)”继承

在 TPH 映射情形下，继承层次结构中的所有类型都将映射到同一个表。鉴别器列用于标识每行的类型。使用 Code First 创建模型时，TPH 参与继承层次结构的类型所用的默认策略。默认情况下，鉴别器列将添加到名为“Discriminator”的表，且层次结构中每个类型的 CLR 类型名称都将用作鉴别器值。可以使用 Fluent API 修改默认行为。

隐藏代码

```
modelBuilder.Entity<Course>()
        .Map<Course>(m=> m.Requires("Type").HasValue("Course"))
        .Map<OnsiteCourse>(m=> m.Requires("Type").HasValue("OnsiteCourse"));
```

## 映射“每个类型一张表(TPT)”继承

在 TPT 映射情形下，所有类型分别映射到不同的表。仅属于某个基类型或派生类型的属性存储在映射到该类型的一个表中。映射到派生类型的表还会存储一个将派生表与基表联接的外键。

隐藏代码

```
modelBuilder.Entity<Course>().ToTable("Course");
modelBuilder.Entity<OnsiteCourse>().ToTable("OnsiteCourse");
```

## 映射“每个具体类一张表(TPC)”继承

在 TPC 映射情形下，层次结构中的所有非抽象类型分别映射到不同的表。映射到派生类的表与映射到数据库中基类的表并无关系。类的所有属性（包括继承属性）都将映射到相应表的列。

调用MapInheritedProperties 方法来配置每个派生类型。MapInheritedProperties 将继承自基类的所有属性重新映射到派生类的表中的新列。

注意：因为属于TPC 继承层次结构的表并不使用同一个主键，因此，如果您让数据库生成的值具有相同标识种子，则在映射到子类的表中执行插入操作时，会产生重复的实体键。要解决此问题，可以为每个表指定不同的初始种子值，或关闭主键属性的标识。当使用 Code First 时，标识就是整数键属性的默认值。

隐藏代码

```
modelBuilder.Entity<Course>()
  .Property(c => c.CourseID)
  .HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);

modelBuilder.Entity<OnsiteCourse>().Map(m =>
{
  m.MapInheritedProperties();
  m.ToTable("OnsiteCourse");
});

modelBuilder.Entity<OnlineCourse>().Map(m =>
{
  m.MapInheritedProperties();
  m.ToTable("OnlineCourse");
});
```

## 将实体类型的CLR 属性映射到数据库中的多个表（实体拆分）

实体拆分允许一个实体类型的属性分散在多个表中。在以下示例中，Department 实体拆分到两个表中：Department 和 DepartmentDetails。实体拆分通过多次调用 Map 方法将一部分属性映射到特定表。

隐藏代码

```
modelBuilder.Entity<Department>()
.Map(m=>
{
  m.Properties(t => new{ t.DepartmentID, t.Name });
  m.ToTable("Department");
})
.Map(m=>
{
  m.Properties(t=> new { t.DepartmentID, t.Administrator,t.StartDate, t.Budget });
  m.ToTable("DepartmentDetails");
 });
```

## 将多个实体类型映射到数据库中的一个表（表拆分）

以下示例将使用同一个主键的两个实体类型映射到同一个表。

隐藏代码

```
modelBuilder.Entity<OfficeAssignment>()
  .HasKey(t => t.InstructorID);
modelBuilder.Entity<Instructor>()
    .HasRequired(t => t.OfficeAssignment)
    .WithRequiredPrincipal(t =>t.Instructor);
modelBuilder.Entity<Instructor>().ToTable("Instructor");
modelBuilder.Entity<OfficeAssignment>().ToTable("Instructor");
```

# 使用Fluent API配置关系



## 简介

使用FluentAPI配置关系的时候，首先要获得一个EntityTypeConfiguration实例，然后使用其上的HasRequired, HasOptional或者 HasMany方法来指定当前实体参与的关系类型。HasRequired 和HasOptional方法需要一个lambda表达式来指定一个导航属性，HasMany方法需要一个lambda表达式指定一个集合导航属性。然后可以使用WithRequired, WithOptional和WithMany方法来指定反向导航属性，这些方法有不带参数的重载用来指定单向导航。

之后还可以使用HasForeignKey方法来指定外键属性。

## 配置【必须-可选】关系（1-0..1）

OfficeAssignment的键属性不符合命名约定，所以需要我们显式指定。下面的关系表明，OfficeAssignment的Instructor必须存在，但是Instructor的OfficeAssignment不是必须存在的。

隐藏代码

```
modelBuilder.Entity<OfficeAssignment>()
  .HasKey(t => t.InstructorID);

// Map one-to-zero or one relationship
modelBuilder.Entity<OfficeAssignment>()
  .HasRequired(t => t.Instructor)
  .WithOptional(t => t.OfficeAssignment);
```

## 配置两端都是必须的关系（1-1）

大多数情况下，EF都能推断哪一个类型是依赖项或者是主体项。然而当关系的两端都是必须的或者都是可选的，那么EF就不能识别依赖项或者是主体项。如果关系两端都是必须的，那么在HasRequired方法后使用WithRequiredPrincipal或者WithRequiredDependent来确定主体。如果关系两端都是可选的，那么在HasRequired方法后使用WithOptionalPrincipal和WithOptionalDependent。

隐藏代码

```
modelBuilder.Entity<OfficeAssignment>()
  .HasKey(t => t.InstructorID);

modelBuilder.Entity<Instructor>()
  .HasRequired(t => t.OfficeAssignment)
  .WithRequiredPrincipal(t => t.Instructor);
```

## 配置多对多关系

下面的代码配置了一个多对多关系，CodeFirst会使用命名约定来创建连接表，命名约定会使用Course_CourseID 和Instructor_InstructorID作为连接表的列。

隐藏代码

```
modelBuilder.Entity<Course>()
        .HasMany(t => t.Instructors)
        .WithMany(t => t.Courses);
```

如果想指定连接表的表名和列名，需要使用Map方法，如下：

隐藏代码

```
modelBuilder.Entity<Course>()
        .HasMany(t => t.Instructors)
        .WithMany(t => t.Courses)
```

```
        .Map(m =>
        {
          m.ToTable("CourseInstructor");
          m.MapLeftKey("CourseID");
          m.MapRightKey("InstructorID");
        });
```

## 配置单向导航属

所谓单向导航属性指的是只在关系的一端定义了导航属性。按照约定，CodeFirst将单向导航理解为一对多关系，如果需要一对一的单向导航属性，需要使用如下方法：

隐藏代码

```
modelBuilder.Entity<OfficeAssignment>()
  .HasKey(t => t.InstructorID);

modelBuilder.Entity<Instructor>()
  .HasRequired(t => t.OfficeAssignment)
  .WithRequiredPrincipal();
```

## 启用级联删除

使用WillCascadeOnDelete方法来配置关系是否允许级联删除。如果外键是不可空的，CodeFirst默认会设置级联删除；否则，不会设置级联删除，当主体被删除后，外键将会被置空。

可以使用如下代码移除此约定：

隐藏代码

```
modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();
modelBuilder.Conventions.Remove<ManyToManyCascadeDeleteConvention>();
```

下面的代码片段配置为外键不能为空，而且禁用了级联删除。

隐藏代码

```
modelBuilder.Entity<Course>()
  .HasRequired(t => t.Department)
  .WithMany(t => t.Courses)
  .HasForeignKey(d => d.DepartmentID)
  .WillCascadeOnDelete(false);
```

## 配置组合外键

下面的代码配置了组合外键

隐藏代码

```
modelBuilder.Entity<Department>()
  .HasKey(d => new{ d.DepartmentID, d.Name });

// Composite foreign key
modelBuilder.Entity<Course>()
    .HasRequired(c => c.Department)
    .WithMany(d => d.Courses)
    .HasForeignKey(d => new { d.DepartmentID, d.DepartmentName });
```

## 配置不符合命名约定的外键属性

SomeDepartmentID属性不符合外键命名约定，需要使用如下方法将其设置为外键属性：

隐藏代码

```
modelBuilder.Entity<Course>()
        .HasRequired(c => c.Department)
        .WithMany(d => d.Courses)
```

```
        .HasForeignKey(c =>c.SomeDepartmentID);
```



≪ 使用Fluent API 配置/映射属性和类型  定义DbSet ≫

# 定义DbSet

## DbContext 使用DbSet 属性

Code First 示例中显示的常见情况是让 DbContext 为模型实体类型使用公共自动 DbSet 属性。例如：

隐藏代码

```
public class  BloggingContext: DbContext
{
  public DbSet<Blog>Blogs { get; set;}
  public DbSet<Post>Posts { get; set;}
}
```

在 CodeFirst 模式下使用时，这会将 Unicorn、Princess、LadyInWaiting 和Castle 配置为实体类型，也将配置可从这些类型访问的其他类型。此外，DbContext 还将自动对其中每个属性调用 setter 以设置相应 DbSet 的实例。

## DbContext 使用IDbSet 属性

在创建 mock或 fake 等情况下，更适合使用接口来声明 set 属性。在这些情况下，可使用 IDbSet 接口替代 DbSet。例如：

隐藏代码

```
public class  BloggingContext: DbContext
{
  public IDbSet<Blog>Blogs { get; set;}
  public IDbSet<Post>Posts { get; set;}
}
```

此上下文的工作方式与对其 set 属性使用DbSet 类的上下文完全相同。

## DbContext 使用只读set 属性

如果不希望为DbSet 或 IDbSet 属性公开公共 setter，可以改为创建只读属性并自建 set 实例。例如：

隐藏代码

```
public class  BloggingContext: DbContext
{
  public DbSet<Blog>Blogs
  {
    get { return Set<Blog>();}
  }

  public DbSet<Post>Posts
  {
    get { return Set<Post>();}
  }
}
```

请注意，DbContext将缓存从 Set 方法返回的 DbSet 实例，以便每次调用其中每个属性时都返回同一实例。

搜索 CodeFirst 实体类型的工作方式与搜索具有公共 getter 和setter 的属性相同。

## DbContext使用注意事项

- 随着越来越多的对象和他们的引用进入内存，DbContext的内存消耗可能会迅速增长，这将会引起性能问题。
- 当不再使用context对象的时候，一定要释放它。
- 如果一个异常使得context进入了一个不可恢复的状态，整个应用可能会终止。

- 长时间使用的context会增加并发冲突的可能。

## DbSet使用注意事项

DbSet总是针对数据库执行查询，即使要查询的数据已经在上下文中，下面几种情况下会执行数据库查询。

- 执行foreach
- 调用ToArray, ToDictionary, ToList.
- 在最外层查询调用LINQ操作符First，Any等等。
- DbSet的扩展方法Load，DbEntityEntry.Reload，Database.ExecuteSqlCommand.

当数据库返回查询结果的时候，如果结果集中的对象在context中不存在，那么就会将对象attach到上下文中。如果对象已经存在（根据id来判断），那么就会返回在上下文中已经存在的对象，数据库的值不会覆盖当前对象database values。在这种情况下，如果我们长时间持有DbContext，那么我们在每次查询的时候得到就很有可能不是最新版本的对象。

在执行一个查询的时候，上下文中新添加但是还没有保存的对象不会作为查询结果返回，如果想访问这些对象，需要访问Local属性。下面是关于local属性的备注

1. Local属性不只是包含新添加的对象，它包含所有已经加载到context中的对象。
2. Local属性不包含那些已经被Remove的对象（上下文中remove了，但是还在数据库中）
3. 查询结果永远反应数据库的真实数据，在上下文中被Remove了但是还没有在数据库删除的对象，仍然可以查询到。

DbContext.ChangeTracker属性提供了DbChangeTracker的实例，该实例的Entries属性返回一个DbEntityEntry集合，可以找到所有当前上下文中跟踪的实体及其状态信息。

有时候在查询大量实体并只进行只读操作的时候，实体跟踪是没有任何意义的，禁用实体跟踪会提高查询性能，可以AsNoTracking方法来禁用实体跟踪，例如：

隐藏代码

```
using(var context = newBloggingContext())
{
  // Query for allblogs without tracking them
  var blogs1 =context.Blogs.AsNoTracking();

  // Query for someblogs without tracking them
  var blogs2 =context.Blogs
                     .Where(b =>b.Name.Contains(".NET"))
                     .AsNoTracking()
                     .ToList();
}
```

## 根据主键查找实体

DbSet.Find方法会根据主键来查找被上下文跟踪的实体。如果上下文中不存在此对象，那么将会对数据库进行查询来查找实体，如果没有找到实体，则返回null。Find方法可以查询到刚刚添加到上下文但是还没有被保存到数据库的实体，这与LINQ查询不同。

使用 Find 方法时必须考虑：

1. 如果对象没有在缓存中，则 Find 没有优势，但语法仍比按键进行查询简单。
2. 如果启用自动检测更改，则根据模型的复杂性以及对象缓存中的实体数量，Find 方法的成本可能会增加一个数量级，甚至更多。

此外，请注意Find 仅返回要查找的实体，它不会自动加载未在对象缓存中的关联实体。如果需要检索关联实体，可通过预先加载使用按键查询。

# 创建和修改关系

对于拥有外键属性的关系，修改关系是非常简单的，如下：

隐藏代码

```
course.DepartmentID = newCourse.DepartmentID;
```

下面的代码通过将外键设置为 null 删除了关系。请注意，外键属性必须可以为 Null。

隐藏代码

```
course.DepartmentID = null;
```

注意：如果引用处于已添加状态（在本例中为 course 对象），在调用 SaveChanges 之前，引用导航属性将不与新对象的键值同步。由于对象上下文在键值保存前不包含已添加对象的永久键，因此不发生同步。

通过将一个新对象分配给导航属性。下面的代码在 course 和department 之间创建关系。如果对象附加到上下文，course 也会添加到 department.Courses 集合中，course 对象的相应的外键属性设置为 department 的键属性值。

隐藏代码

```
course.Department = department;
```

要删除该关系，请将导航属性设置为 null。如果使用的是基于 .NET 4.0 的实体框架，则需要先加载相关端，然后再将其设置为 Null。例如：

隐藏代码

```
context.Entry(course).Reference(c = >c.Department).Load();
```

隐藏代码

```
course.Department = null;
```

从实体框架5.0（它基于 .NET 4.5）开始，不必加载相关端就可以将关系设置为 Null。也可以使用以下方法将当前值设置为 Null。

隐藏代码

```
context.Entry(course).Reference(c = >c.Department).CurrentValue = null;
```

通过在实体集合中删除或添加对象。例如，可以将 Course 类型的对象添加到 department.Courses 集合中。此操作将在特定 course 和特定 department 之间创建关系。如果对象附加到上下文，course 对象的 department 引用和外键属性将设置为相应的 department。

隐藏代码

```
department.Courses.Add(newCourse);
```

此处，如果course的departmentid不能为空，则可能会出现错误，对department.Courses集合不能有删除course的操作，否则会出现错误。因为如果从集合中移除了course，在SaveChanges过程中把该过程识别为更新关系，而那些被删除的course的departmentid又不能为空，所以save不会成功。

通过使用 ChangeRelationshipState方法更改两个实体对象间指定关系的状态。此方法是处理 N 层应用程序和独立关联 时最常用的方法（不能用于外键关联）。此外，要使用此方法，必须下拉到 ObjectContext，如下例所示。

在下面的示例中，Instructor和 Course 之间存在多对多关系。调用 ChangeRelationshipState 方法并传递 EntityState.Added 参数，使 SchoolContext 知道在这两个对象间添加了关系。

隐藏代码

```
((IObjectContextAdapter)context).ObjectContext.ObjectStateManager.
                ChangeRelationshipState(course, instructor, c => c.Instructor,EntityState.Added);
```

请注意，如果是更新（而不仅是添加）关系，添加新关系后必须删除旧关系：

隐藏代码

```
((IObjectContextAdapter) context).ObjectContext.ObjectStateManager.
ChangeRelationshipState(course, oldInstructor, c = >c.Instructor, EntityState.Deleted);
```

## 同步FK 和导航属性之间的更改

使用上述方法中的一种更改附加到上下文的对象的关系时，实体框架需要保持外键、引用和集合同步。实体框架使用代理自动管理 POCO 实体的这种同步（也称为关系修复）。

如果不通过代理使用POCO 实体，则必须确保调用 DetectChanges 方法同步上下文中的相关对象。请注意，下面的 API 会自动触发 DetectChanges 调用。

- DbSet.Add
- DbSet.Find
- DbSet.Remove
- DbSet.Local
- DbContext.SaveChanges
- DbSet.Attach
- DbContext.GetValidationErrors
- DbContext.Entry
- DbChangeTracker.Entries
- 对 DbSet 执行 LINQ 查询

如果context中有很多实体，而且你正在多次调用上述方法，那么就会造成很大的性能影响。可以使用下面的代码来的代码禁用自动检测：

隐藏代码

```
using(var context = newBloggingContext())
{
  try
  {
   context.Configuration.AutoDetectChangesEnabled = false;

    // Make manycalls in a loop
    foreach (var blog inaLotOfBlogs)
    {
      context.Blogs.Add(blog);
    }
  }
  finally
  {
   context.Configuration.AutoDetectChangesEnabled = true;
  }
}
```

除了以上方法，还可以调用context.ChangeTracker.DetectChanges方法来显式检测变化。需要小心使用这些高级方法，否则很容易在你的程序里引入微妙的bug。

# 加载相关对象

## 预加载（EagerlyLoading）

预加载表示在查询某类实体时一起加载相关实体，这是使用Include方法完成的，如下：

隐藏代码

```
using(var context = newBloggingContext())
{
  // Load all blogsand related posts
  var blogs1 =context.Blogs
.Include(b=> b.Posts)
.ToList();

  // Load one blogsand its related posts
  var blog1 =context.Blogs
.Where(b=> b.Name == "ADO.NET Blog")
.Include(b=> b.Posts)
.FirstOrDefault();

  // Load all blogsand related posts
  // using a stringto specify the relationship
  var blogs2 =context.Blogs
                      .Include("Posts")
                      .ToList();

  // Load one blogand its related posts
  // using a stringto specify the relationship
  var blog2 =context.Blogs
                    .Where(b => b.Name == "ADO.NET Blog")
                    .Include("Posts")
                    .FirstOrDefault();
}
```

注意：Include方法是一个扩展方法，在System.Data.Entity命名空间下，确保引用了此命名空间。

## 多级预加载

下面的代码显示了如何加载多级实体。

隐藏代码

```
using(var context = newBloggingContext())
{
  // Load all blogs,all related posts, and all related comments
  var blogs1 =context.Blogs
                    .Include(b =>b.Posts.Select(p => p.Comments))
                    .ToList();

  // Load all userstheir related profiles, and related avatar
  var users1 =context.Users
                     .Include(u =>u.Profile.Avatar)
                     .ToList();
```

```
  // Load all blogs,all related posts, and all related comments
  // using a stringto specify the relationships
  var blogs2 =context.Blogs
                     .Include("Posts.Comments")
                     .ToList();

  // Load all userstheir related profiles, and related avatar
  // using a stringto specify the relationships
  var users2 =context.Users
                      .Include("Profile.Avatar")
                      .ToList();
}
```

当前不支持在关联实体上进行查询，Include方法总是加载所有关联实体。

## 惰性加载

惰性加载指的是当第一访问导航属性的时候自动从数据库加载相关实体。这种特性是由代理类实现的，代理类派生自实体类，并重写了导航属性。所以我们的实体类的导航属性就必须标记为virtual，如下：

隐藏代码

```
public class  Blog
{
  public int ID { get; set; }
  public string Title { get; set; }
  public virtual ICollection<Post>Posts { get; set;}
}
```

可以对指定实体关闭惰性加载，如下：

隐藏代码

```
public class  Blog
{
  public int ID { get; set; }
  public string Title { get; set; }
  public ICollection<Post>Posts { get; set;}
}
```

也可以对所有实体关闭惰性加载，如下：

隐藏代码

```
public class  BloggingContext: DbContext
{
  public BloggingContext()
  {
    this.Configuration.LazyLoadingEnabled=  false;
  }
}
public class BloggingContext: DbContext {
    publicBloggingContext() {
        this.Configuration.LazyLoadingEnabled = false;
    }
}
```

## 显式加载

即使关闭了惰性加载，我们仍然可以通过显式调用来延迟加载相关实体，这是通过调用DbEntityEntry上的相关方法做到的，如下：

隐藏代码

```
using(var context = newBloggingContext())
{
```

```
  var post =context.Posts.Find(2);

  // Load the blogrelated to a given post
  context.Entry(post).Reference(p => p.Blog).Load();

  // Load the blogrelated to a given post using a string
  context.Entry(post).Reference("Blog").Load();

  var blog =context.Blogs.Find(1);

  // Load the postsrelated to a given blog
  context.Entry(blog).Collection(p =>p.Posts).Load();

  // Load the postsrelated to a given blog
  // using a stringto specify the relationship
  context.Entry(blog).Collection("Posts").Load();
}
```

注意：在外键关联中，加载依赖对象的相关端时，将根据内存中当前的相关外键值加载相关对象：

隐藏代码

```
// Get thecourse where currently DepartmentID = 1.
Course course2 =context.Courses.First(c=>c.DepartmentID == 2);
 // UseDepartmentID foreign key property
// to change theassociation.
course2.DepartmentID = 3;
 // Load therelated Department where DepartmentID = 3
context.Entry(course).Reference(c=> c.Department).Load();
```

在独立关联中，基于当前数据库中的外键值查询依赖对象的相关端。不过，如果修改了关系，并且依赖对象的引用属性指向对象上下文中加载的不同主对象，实体框架将尝试创建关系，就像它在客户端定义的那样。

Query方法提供了在加载相关实体的时候应用过滤条件的功能，引用导航属和集合导航属性都支持Query方法，但是大部分情况下都会针对集合导航属性使用Query方法，达到只加载部分相关实体的功能，如下：

隐藏代码

```
using(var context = newBloggingContext())
{
  var blog =context.Blogs.Find(1);

  // Load the postswith the 'entity-framework' tag related to a given blog
  context.Entry(blog)
      .Collection(b => b.Posts)
      .Query()
      .Where(p => p.Title.Contains("entity-framework"))
      .Load();

  // Load the postswith the 'entity-framework' tag related to a given blog
  // using a stringto specify the relationship
  context.Entry(blog)
      .Collection("Posts")
      .Query()
      .Cast<Post>()
      .Where(p => p.Title.Contains("entity-framework"))
      .Load();
}
```

注意，使用显式加载的时候，最好关闭惰性加载，避免引起混乱。Load方法是一个扩展方法，记得引用命名空间 System.Data.Entity.DbExtensions

使用Query查询相关实体个数，而不用加载相关实体，如下：

隐藏代码

```
using(var context = newBloggingContext())
{
  var blog =context.Blogs.Find(1);

  // Count how manyposts the blog has
  var postCount= context.Entry(blog)
                        .Collection(b =>b.Posts)
                        .Query()
                        .Count();
}
```

# 使用代理

为 POCO 实体类型创建实例时，实体框架常常为充当实体代理的动态生成的派生类型创建实例。此代理重写实体的某些虚拟属性，这样可在访问属性时插入挂钩，从而自动执行操作。例如，此机制用于支持关系的延迟加载。

## 禁止创建代理

有时需要禁止实体框架创建代理实例。例如，人们通常认为序列化非代理实例要比序列化代理实例容易得多。可通过清除 ProxyCreationEnabled 标记来关闭代理创建功能。上下文的构造函数便是可执行此操作的一个位置。例如：

隐藏代码

```
public class  BloggingContext: DbContext
{
  publicBloggingContext()
  {
    this.Configuration.ProxyCreationEnabled=  false;
  }

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

请注意，在无需代理执行任何操作的情况下，EF 不会为类型创建代理。这意味着，也可以通过使用封装和/或没有虚拟属性的类型，避免生成代理。

## 添加新实体

使用DbSet.Add方法添加实体

隐藏代码

```
using(var context = new BloggingContext())
{
  var blog = new Blog { Name = "ADO.NET Blog" };
  context.Blogs.Add(blog);
  context.SaveChanges();
}
```

修改Entry的State来添加实体

隐藏代码

```
using(var context = new BloggingContext())
{
  var blog = new Blog { Name = "ADO.NET Blog" };
  context.Entry(blog).State =EntityState.Added;
  context.SaveChanges();
}
```

设置导航属性来添加实体

隐藏代码

```
using(var context = new BloggingContext())
{
  // Add a new Userby setting a reference from a tracked Blog
  var blog =context.Blogs.Find(1);
  blog.Owner = newUser { UserName =  "johndoe1987" };

  // Add a new Postby adding to the collection of a tracked Blog
```

```
  var blog =context.Blogs.Find(2);

  blog.Posts.Add(newPost { Name = "Howto Add Entities" });


  context.SaveChanges();
}
```

所有被添加到上下文中的实体的引用实体，如果没有被跟踪，就会被当作新实体添加到上下文中，并在调用SaveChanges方法后被保存到数据库。

## 附加实体到上下文

如果实体在数据库中存在，但是没有被上下文跟踪，可是使用DbSet.Attach方法将其附加到上下文，附加之后，实体处于Unchanged状态。处于Unchanged状态的实体不会参与SaveChanges的逻辑。

```
                                                隐藏代码
var existingBlog = new Blog{ BlogId = 1, Name =  "ADO.NET Blog"};


using(var context = new BloggingContext())
{
  context.Blogs.Attach(existingBlog);


  // Do some morework...


  context.SaveChanges();
}
var existingBlog = new Blog {
    BlogId = 1,
    Name = "ADO.NET Blog"
};
using(var context = new BloggingContext()) {
    context.Blogs.Attach(existingBlog); // Do some morework...     context.SaveChanges();}
```

设置DbEntityEntry对象的State属性，也可以附加对象到上下文中，如下：

```
                                                隐藏代码
var existingBlog = new Blog{ BlogId = 1, Name =  "ADO.NET Blog"};


using(var context = new BloggingContext())
{
  context.Entry(existingBlog).State =EntityState.Unchanged;


  // Do some morework...


  context.SaveChanges();
}
```

使用上述两种方法附加到上下文的实体如果还引用其他实体，那么这些实体也会被附加到上下文中，状态为Unchanged

使用如下方法附加一个存在于数据库，但是还没有附加到上下文的已修改实体：

```
                                                隐藏代码
var existingBlog = new Blog{ BlogId = 1, Name =  "ADO.NET Blog"};


using(var context = new BloggingContext())
{
  context.Entry(existingBlog).State =EntityState.Modified;


  // Do some morework...


  context.SaveChanges();
}
```

如果把一个实体的状态置为Modified，那么该实体的所有属性都将被标记为已更改状态，当SaveChanges被调用时，所有的属性值都将被保存到数据库。如果不想保存所有值，可以单独为每个想要修改的属性设置IsModified属性，如下：

```
隐藏代码
using(var context = new BloggingContext())
{
    var blog =context.Blogs.Find(1);

    context.Entry(blog).Property(u =>u.Name).IsModified =  true;

    // Use a stringfor the property name
    context.Entry(blog).Property("Name").IsModified = true;
}
```

如果该实体还引用其他未被跟踪实体，那么这些实体将会作为Unchanged状态的实体附加到上下文。如果想修改这些实体，只能单独把每个引用实体设置为修改状态。

## 乐观并发模式

在尝试保存使用外键关联的实体期间，如果检测到乐观并发异常，SaveChanges 将引发 DbUpdateConcurrencyException。DbUpdateConcurrencyException的 Entries 方法为无法更新的实体返回 DbEntityEntry 实例。

## 使用DbContext执行原始SQL

使用SqlQuery方法执行SQL查询

```
隐藏代码
using(var context = new BloggingContext())
{
  var blogs =context.Blogs.SqlQuery("SELECT * FROMdbo.Blogs").ToList();
}
```

执行存储过程查询

```
隐藏代码
using(var context = new BloggingContext())
{
  var blogs =context.Blogs.SqlQuery("dbo.GetBlogs").ToList();
}
```

为存储过程传递参数

```
隐藏代码
using(var context = new BloggingContext())
{
  var blogId =1;

  var blogs =context.Blogs.SqlQuery("dbo.GetBlogById@p0", blogId).Single();
}
```

查询非实体类型

```
隐藏代码
using(var context = new BloggingContext())
{
  var blogNames= context.Database.SqlQuery<string>(
                    "SELECTName FROM dbo.Blogs").ToList();
}
```

返回是的对象将不会被跟踪，即使返回类型是实体类型。

执行SQL命令

```
隐藏代码
using(var context = new BloggingContext())
{
  context.Database.SqlCommand(
      "UPDATEdbo.Blogs SET Name = 'Another Name' WHERE BlogId = 1");
```

```
}
```



≪ 加载相关对象