# 第一章：欢迎来到Code First

(译者注：为方便起见，不再直译Code First)

微软的ADO.NET实体框架（Entity Framework），泛称EF，把外部（out-of-the-box，译成“外部”不一定对，我是这样理解的：the-box可能意指.NET框架，合起来的意思就可能是“外部”了 — 译者ro1cn注）的对象关系映射（ORM）引到了.NET和Visual Studio。实体框架的核心是实体数据模型，这是应用程序域向数据库架构进行映射的一个概念模型。这个概念模型描述了应用程序中的核心类。实体框架在查询数据库，或根据数据创建对象，然后把修改持久化回数据库时，都会使用这个概念模型。（译者注：此段全文引用r01cn园友的译文，确实有深度，佩服！）

## 在编写代码前构建EF模型

在使用.Net 3.5和Visual Studio 2008构建的第一代EF中，给予了开发者通过将现存数据库转化为XML文件的形式来创建概念模型的能力。这种XML文件使用EDMX扩展名，可以使用设计器来查看和定制模型以便更好地适应程序域。Visual Studio 2010 and .NET 4带来了EF的新版本：称之为Entity Framework 4 (EF4)。EF4还引入了一种称为Model First的功能，通过此功能可以先创建概念模型然后根据此模型再创建数据库。

Model First 允许开发者在没有传统的数据库时就可以开始一个新的项目，这是EF框架带来的好处。开发者可以通过设计概念模型而首先关注应用程序域，从而让数据库的创建工作水到渠成。

无论是通过database-first还是通过model-first设计EDMX，下一步都是为了创建您的域以便让自动代码生成器基于实体以及找到的各种关系来创建类。通过这种方式，开发人员拥有强类型的类来指向域对象，无论它们是客户，棒球卡，还是童话人物，都可以围绕着这些类来愉快地开发软件应用。

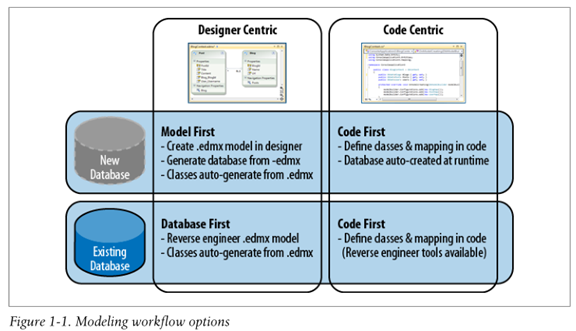
另一个关键的变化在EF4。在.NET 3 5，实体框架只能够管理处于内存中的对象。这些对象的类都继承自实体框架的EntityObject对象。EntityObject对象向实体框架通知变更，并由实体框架跟踪这些变化并最终持久化到数据库中。NET 4中不仅仍有这些功能，还推出了 POCO（Plain Old CLR Object，简单传统CLR对象），支持实体框架跟踪简单类的变化而不需要EntityObject的参与，这使开发者可以自由使用自己的类，从而独立于实体框架。EF在运行时可以通过监视对象在内存中的行为保持对这些类的响应和跟踪。

## Code First的发起

基于前面介绍的有关EF4的片断，我们可以看到，微软已经提供了多途径的建模方式，其中有一种新的方式是在EF发布之初，开发人员就所期待的。这种新型的建模方法就是所谓的Code First. Code first 首先让您定义域模型，而不是使用基于XML的edmx文件。即使采用Model First或Database First来生成代码，开发者也必需使用设计器或类生成器来辅助工作。而使用Code First您可以直接通过使用POCO类来定义域模型，而无需依赖于EF框架。Code First可以通过构建的类模型推断出大量信息。你也可以提供额外的配置，进一步描述有关模型的信息或者是覆写Code Firstr推断的信息。这种配置也是用代码来定义的，不需要XML文件或设计器。

使用设计器工作的EF4也支持POCO技术。EF开发团队提供了一种POCO模板来生成POCO类（可以使用NuGet下载这种T4模板--译者注）。这些生成的类可以随着设计器的更改而自动更新。也可以使用你自己创建的POCO类。但如果你决定这样做，就需要负责保持您的类与EDMX文件的同步。这意味着任何变更都必须在两个位置进行。Code First的一个最大好处就是让你的类变成了模型。这意味着改变模型就只需要在一处（你自已的POCO类）中作出更改即可。

Code First, Database First, 和 Model First都只是一种建立可用于EF框架中以进行数据访问的实体数据模型的方式。一旦模型构建，EF框架在运行时的表现是相同的，与如何构建模型无关。选择从设计器开始还从代码开始完全取决于您的选择。图1-1列出了不同的可选项。

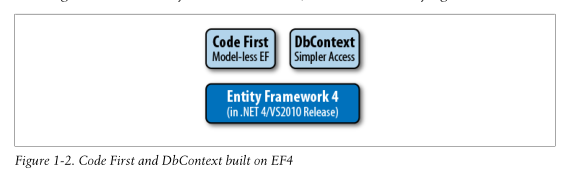


微软将Database First, Model First, Code First分别形成了可选的工作流程。这是因为每种选项都是一系列步骤，无论这些步骤是手工实施还是自动进行。例如，使用Database First工作流，你需要将数据库转化为工程，然后让代码生成器创建类。而Code First工作流则开始于类的代码构建，然后用代码生成所用的数据库。

## 在.NET发行版本中间获取Code First

Code First 在.Net4发布时尚未准备好。微软不想将其在.Net5发布时才带给开发者，就在2011年4月推出了"特别"版，称为EF4.1.版本号随后在更新时顺延。2011年10月，EF4.2推出 ，替代了EF4.1，作为包含Code first的最新版本。核心API，System.Data.Entity.dll，仍然是.Net Framework的一部分，没有与EF4.1，4.2紧密连接。EF4.2还包含了另一个重要的功能，称之为DbContext API.DbContext是这个API的核心，除此之外还包含其他支持类。DbContext是一个轻量级的EF ObjectContext.这是一个包装过的ObjectContext,只暴露了那些微软认为大多数开发人员最常用的EF功能。DbContext也支持通过代码模式访问由ObjectContext所提供的复杂功能。DbContext提供了大量的通用任务，可以使开发者用较少的代码完成这些任务，这在使用Code First时尤其明显。由于微软推荐使用DbContext来进行Code First操作，您会看到本书将贯穿始终介绍此方法。但是，另一本书，叫做Programming Entity Framework: DbContext，将深入探讨DbContext，DbSet，验证API以及DbContext带来的其他特征。

图1-2帮助你查看如何通过构建EF4 API核心添加Code First和DbContext功能。



## 灵活的发布日程

微软将继续通过VS提供的NuGet工具发布基于EF4.2框架的新功能。EF的核心库将会随着.Net的新版本的发布而内置在其中。但是诸如CodeFirst 和DbContext等核心功能会在EF的NuGet包中不断得到更新。

## 先输入代码。。。

Code First 是个好听的名字：先写代码，然后继续。让我们先看看一些基本的默认功能，不要管各种你可能会遇到的场景（后续的内容我们会专注于此）。

我们并不期望你重建本章的代码案例。这些案例只是作一概览，并不会深入。在第2章，你将会逐步深入。你将会跟随我们在VS中的操作进行，然后尝试你想实现的功能。

当然，首先需要一些代码，以便足以描述业务域。本案例是关于一个宠物医院收治患畜的业务模型。

Example 1-1 Domain classes

using System;

using System.Collections.Generic;

namespace ChapterOneProject

{

class Patient

{

public Patient()

{

Visits = new List<Visit>();

}

public int Id { get; set; }

public string Name { get; set; }

public DateTime BirthDate { get; set; }

public AnimalType AnimalType { get; set; }

public DateTime FirstVisit { get; set; }

public List<Visit> Visits { get; set; }

}

class Visit

{

public int Id { get; set; }

public DateTime Date { get; set; }

public String ReasonForVisit { get; set; }

public String Outcome { get; set; }

public Decimal Weight { get; set; }

public int PatientId { get; set; }

}

class AnimalType

{

public int Id { get; set; }

public string TypeName { get; set; }

}

}

Code First 的核心是约定，这些默认的规则使我们可以用我们自己的类来创建模型。EF框架要求一个类必须有一个键属性。规则约定如果一个属性名为Id或者是类名+Id的形式（如PatientId），这一属性就被自动配置为键。如果无法找到满足这一规则的属性，将会在运行时抛出一个异常告诉你没有找到Key.其他约定包括确定字符串的默认长度，或者默认表结构，以及当类相互继承时如何在数据库内建表等等。

如果Code First完全依赖于这些规则去创建自己的类，我们需要做的工作是很有限的。但是Code First并不强制你按此要求设计类，这些规则的设计是为了Code First能够自动处理一些通用的场景。如果你的类遵从这些规则，Code First不会需要更多信息。EF框架会以你的类直接工作。如果不遵从这些规则，你就必须通过Code First的一些配置选项来提供一些附加信息以确保你的类能够被Code First 所理解。

I代码1-1的三个类，Id属性均符合键属性的规则要求。使用这三个类按Code First工作不需要增加任何配置。

## 使用DbContext管理对象

前述的域与EF框架无关。Code First很美好，你直接就使用你自己的类。如果你从其他项目中现存的域类开始将获得特别的好处。

使用Code First,您应该定义一个继承自DbContext的类。此类的其中一个角色，应被指作为context，以便让Code First知道有关类需要用于创建模型。这就是EF框架能够感知类以及如何保持跟踪这些类的原因。这需要引入另一个新类：DbSet。正如DbContext是一个简单包装的ObjectContext, DbSet是一个包装过的EF4的ObjectSet对象，同样简化了正常使用ObjectSet的编码任务量。

例1-2展示了这种context类的基本样式。注意到其中有一个用于Patients 和Visits的DbSet属性。DbSets允许你使用类型查询。如果我们不希望直接查询AnimalTypes，就不需要一个AnmialType的DbSet.Code First 足够聪明知道Patient使用AnimalType类，会包含在模型内。

Example 1-2. VetContext class which derives from DbContext

using System.Data.Entity;

namespace ChapterOneProject

{

class VetContext : DbContext

{

public DbSet<Patient> Patients { get; set; }

public DbSet<Visit> Visits { get; set; }

}

}

## 使用数据层和域类

现在你可能会有一些惊讶：这些可以满足数据层的需要了吗？假如你100%按照Code First的约定就是这样。

这里没有数据库连接字符串，甚至连数据库也没有。但你已经做好使用数据层的准备。例1-3显示了一个创建新病患对象的方法。这个方法同时创建了该病患的信息以及第一次治疗记录；

然后我们将此实例装入context，添加到DbSet<Patient>(Patinets)中（前已定义为context）,最后调用DbContext的SaveChanges方法保存数据。

Example 1-3. Adding a patient to the database with the VetContext

private static void CreateNewPatient()

{

var dog = new AnimalType { TypeName = "Dog" };

var patient = new Patient

{

Name = "Sampson",

BirthDate = new DateTime(2008, 1, 28),

AnimalType = dog,

Visits = new List<Visit>

{

new Visit

{

Date = new DateTime(2011, 9, 1)

}

}

};

using(var context = new VetContext())

{

context.Patients.Add(patient);

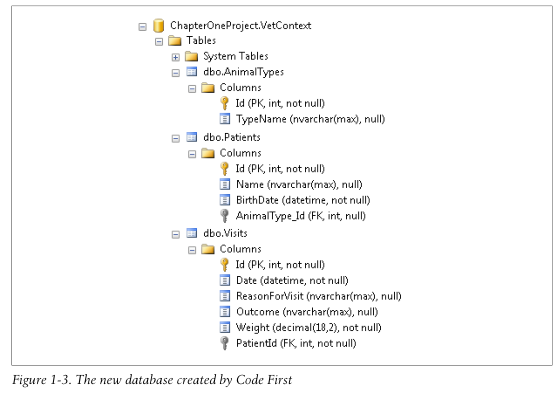
context.SaveChanges();

}

}

记住目前还没有任何连接字符串，也没有任何数据库。然后运行这些代码后，我们会在本地的SQL Server Express找到名字与context类完全匹配的新的数据库：ChapterOneProject.VetContext

有关此数据的结构图示见图1-3.



将数据库构架与例1中定的类相对比，表与类，字段与属性几乎完全匹配，唯一的区别是外键，尽管在Patient类中并没有外键属性，但Patients.AnimalType\_Id外键依然创建了。Code First得到此外键是根据类中表现出的关系确定的。（注意到Patient对AnimalType有一个引用），在数据库中外键用来维持表间的关系。在这种处理关系的方面Code First提供了很多便利。在类中有很多方式来表达类间的关系。Code First能够方便地识别这些关系。注意，PatientId字段，在类中本来是明确指向Visit类的属性，是非空的，而来自于导航属性的AnimalType\_Id字段是可空的。约定再一欠确定了外键的可空性，如果想要修改这些字段，让Code First按照您想法来表达，就需要使用附加的配置。

## 从类到数据库

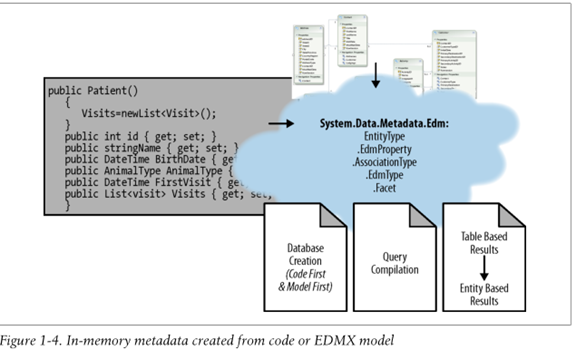
如果您在EF框架下工作过，一定会熟悉采用EDMX文件表达的可视化模型。你也可以意识到EDMX文件实际上就是一个XML文件，但是设计器使其工作更加容易。使用XML来描述模型需要专门的构架元素，没有设计器处理XML源文件是相当耗费脑筋的。

很显然显示在设计器上的内容并非是XML文件内容的全部。这一文件中还有一些数据库构架的描述，以使类能够映射到类似图1-3中相应的表和列中。模型XML，数据库构架XML和映射XML都被引用为元数据。

在运行时，EF框架读取前述三部分的XML文件并在内存中创建元数据的映像。内存中的影像不再是XML，而是强类型的对象，如EntityType, EdmProperty和ssociationType.EF框架与这种在内存中的类对象进行交互，然后在需要时再与数据库进行交互。

Code First并没有XML文件，其内存中的数据对象是根据您的域类集合直接生成的。这里是约定与配置在起作用。Code First有一个称为DbModelBuilder的类。这个类读取域中的类集合，然后构建内存模型。由于其也可以构建代表数据库构架的元数据，因此可以用来创建数据库。如果增加配置信息帮助模型构建器确定哪一个模型和数据库构架应该看起来一致，模型构建器就会检查这些类，并将有关信息集成进模型中，使数据库构架与设想的一致。

图1-4显示了EF框架如何从代码或XML文件（由设计器维护和操作）中创建内存模型。一旦内存模型得以构建，EF框架不再需要知道模型是如何创建的。也可使用内存的模型来确定数据库架构是怎样的，可以构建查询访问数据，将查询结果返回对象，将对象的变化更新回数据库。



## 使用配置

在某些场合下需要协助Code First理解你的意图时，你有两个选择来实施配置：Data Annotations（数据注解）和Code First的Fluent API.根据个人喜好可以任选其一。有一些高级配置只有在使用Fluent API时才能实现。

Code First允许您配置各种属性，关系，继承，级别和数据库映射。现在我们给出一个示例，先睹为快。本书将把精力和时间放在解释约定和对你有用的配置属性上去。

## 使用Data Annotations配置

多开发者喜欢使用Data Annotations配置方式，因为其非常简单。Data Annotations将一些特性直接应用于想要影响的类或属性上。这些特性可以在System.ComponentModel.DataAnnotations命名空间里找到。

例如，如果你想确保一个属性应总是有一个值，你就可以使用Required特性。例1-4在AnimalType类的TypeName属性上应用了这一特性。

Example 1-4. Using an annotation to mark a property as required

class AnimalType

{

public int Id { get; set; }

[Required]

public string TypeName { get; set; }

}

这会有两个效果。第一个是数据库中的TypeName字段将会设置为非空。第二个将会实现由EF框架提供的验证，得益于EF4.1框架引入的验证API。默认情况下，当保存更改时，EF框架将检查确保被标记为Required的属性是非空的。如果为空，将会抛出异常。

Required特性影响数据库列的各个方面和属性验证。许多特性以用于数据库映射。例如，Table特性告诉Code First该类映射到一个具有某个名子的表中。应用程序中的AnimailType数据完全可以存储在称为Species的表中。Table特性允许指定这种映射。

Example 1-5. Specifying a table name to map to

[Table("Species")]

class AnimalType

{

public int Id { get; set; }

[Required]

public string TypeName { get; set; }

}

## 使用Fluent API配置

使用Data Annotations非常简单，在域中的类中指定元数据可能不适应你的开发风格。有一个替代的方法来添加配置，就是使用Code First's Fluent API.使用Fluent API，你的域类始终保持"清洁"。你的配置信息通过重写DbContext暴露的方法OnModelCreating来实现。例1-6展示了与前述相同的配置，这个示例使用的是Fluent APIn。在每个配置中，代码指出了model bulider如何配置AnimalType.

Example 1-6. Configuring the model using the Fluent API

class VetContext : DbContext

{

public DbSet<Patient> Patients { get; set; }

public DbSet<Visit> Visits { get; set; }

protected override void OnModelCreating

(DbModelBuilder modelBuilder)

{

modelBuilder.Entity<AnimalType>()

.ToTable("Species");

modelBuilder.Entity<AnimalType>()

.Property(p => p.TypeName).IsRequired();

}

}

第一个使用Fluent API的配置等价于Data Annotation 的Table特性，现在是ToTable方法，传递了一个表名指定了AnimalType类应该生成为该表；第二个配置使用了一个Lambda表达式指定AnimalType的一个属性应用IsRequired方法。

这只是一个建立配置的方法。你将会在后续的章节中学到更多有关使用Data Annotations 和Fluent API的知识，用于配置属性、关系，继承，层级和数据库映射等。

## 创建或指向数据库

前面，你已经看到默认代码创建了一个SQL Server Express数据库。Code First完全自动调用了内置的连接字符串生成了数据库。

你将在第6章全面了解Code First是如何与你的数据库进行交互的。

本书中的示例将逐步介绍如何配置数据库映射。这些概念同样适用于生成数据库和映射到一个已经存在的数据库。当生成数据库时，会影响生成的构架。当映射到存在的数据库时，数据库定义的构架必须与EF框架在运行时生成的保持一致。

在我们探索Code First的约定和配置时，我们也会允许Code First创建数据库。这就使得运行应用程序的每个步骤都可以看到数据库构架的变化。如果映射到一个已经存在的数据库，唯一的区别是为数据库指定Code First。最容易的办法在"使用配置文件控制数据库位置"（第6章），也可以看看"逆向Code First"(第8章)。

## Code First 不支持的情况

Code First 是新加入到EF框架中的技术，有几个特性目前尚不支持。EF开发团队指出他们正在计划在未来版本中加入这些特性。

数据迁移（Database migrations）

在写这本书的时候，Code First尚不支持数据迁移，换句话说，修改数据反映在模型的变化上。但是这一特性在不久的将来就可能发布。也可以在该团队的blog上获取有关数据迁移的早期预览版。

映射到视图

Code First目前只支持映射到表。这一不幸意味着不能将Code First直接映射到存储过程，视图或其他数据库对象。如果你正使用Code First来生成数据库，没有办法直接将这些特性加到数据库，必须手工添加在Code First已经创建的数据里。如果映射到一个现存的数据库，有一些其他的技术可以用来获取数据库中非表格数据。

这些技术描述为"映射到非表格数据库对象"，见第7章。

## 架构定义查询

EF实体框架包含一个定义查询功能，允许您指定数据库查询直接访问XML元数据。还有一个查询视图功能，可让您使用概念模型定义一个用来加载实体的查询。这允许您指定的查询提供独立的数据库。Code First不支持这些功能。

## 单类型多实体集（Multiple Entity Sets per Type (MEST)）

Code First尚不支持MEST。MEST允许使用同一个类映射到不同的表中。这是EF构架的一个隐含特性，很少用到。EF团队表示，为了确保Code First API的简化，他们不打算加入对MEST的支持。

## 条件列映射

继承层次结构时，Code First要求一个属性始终具有相同的名称映射到列，而不能实现条件列映射。例如，有一个Person基类具有一个NationalIdentifier属性。American和Australian类都是派生自Person基类分别映射到数据库中单独的Americans和Australians表中。当使用设计器时，你可映射NationalIdentifier属性到Americans表中的SSN列，映射到Australians表中的PassportNumber列。Code First 并不支持这种情况。映射到各个表中的NationalIdentifier必须具有同一名称。

## 选择Code First

现在你已经知道什么是Code First,你可能还在犹豫是否这是一种正确的应用程序开发模型工作流。好消息是这完全取决于你和你团队的开发风格。

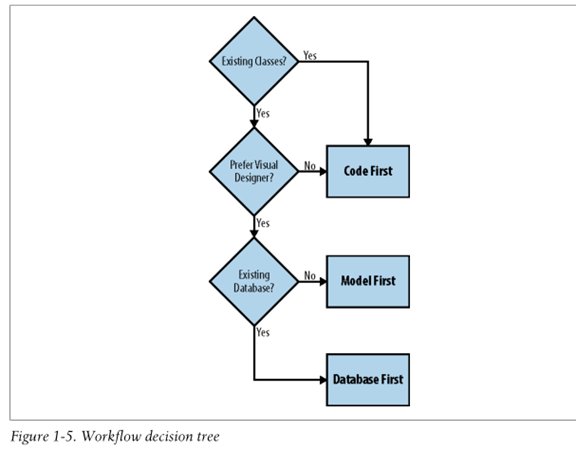
如果你喜欢书写自己的POCO类然后使用代码定义如何映射到数据库，Code First是你要选择的。正如前面提到的，Code First可以为你生成数据库或者映射到已存在的数据库；

如果你更喜欢用设计器定义你的类，并且用可视化的方式映射到数据库，你可能不会使用Code First.如果你需要映射到一个现存的数据库，你可能会想用Database First来从database逆向生成为模型。这可以使用VS的实体数据模型向导生成基于数据库的EDMX文件。你可以使用设计器查看和编辑生成的模型。如果尚没有数据库而仍然要使用设计器，你应该使用设计器以Model First方式定义你的模型。根据创建的模型再创建数据库。这些方法工作良好，在设计器中你会很好地应用你的类。

最后，如果你想针对现存的类使用EF框架，你可能会选择Code First，即便你第一选择可能是基于模型的设计器。如果选择了设计器，你需要在设计器对模型和类作出调整。这种方式低效易出错，而使用Code First可能会获得好处。在Code First中，你的类就是模型，因此模型变更只需要在一处，不需要所谓的同步操作。

EF团队正在对设计器进行改进以加入附加功能：数据库逆向生成支持Code First和Fluent配置的 类。这一工具将为已经拥有数据库但是更想使用Code First的开发者提供。你可以在第8章获取更多信息。

EF工作流的选择总结如图1-5.



### 从本书中可以学到什么？

本书关注采用Code First构建和配置模型。这是一种采用EF框架编程的扩展，在本书背后你会找到很多参考，不会在这复制近900页的有关EF框架的详细信息，而是关注于如何实现查询，更新，如何在各种应用类型中使用，如何自动测试，如何处理异常，安全，数据库连接和事务。采用Code First 创建模型只是EF框架中诸多特性中的一个。事实上，当进入第二章，你会发现宠物医院的域模型将会变为可编为应用程序的业务模型，称之Break Away Geek Adventures（奇客冒险之旅）。另一本书名为《编程实体框架：DbContext》的书，将集中于DbContext，DbSet，验证API，并使用NuGet包装的Enfity Framework的一部分的功能，这些不是本书的重点。

# 第二章：Code First概览

如果你使用第一、二版的EF框架工作过，你会回想起书中的业务案例：Break Away Geek Adventures, 简称BAGA。BAGA共享了很多像我们这样的奇客的冒险旅行。但是几年过去了，业务又在增长，到了需要更新应用程序的时候了。既然BAGA为软件奇客服务，他们不能拒绝尝试使用新技术，如EF的Code First.

在这一章里，我们从一个小例子来开始学习Code First的默认行为，然后逐步向这个例子里添加影响这种行为的信息。

我们将开始于BAGA业务域的一个小片断：包括我们旅行的目的地和我们的奇客们在这次旅行的住所。

Code First的美妙在于域类的定义代码与EF数据模型所依赖的代码是一样的。我们只需要开始于代码就可以了，例2-1，分别展示了Destination类和Loadging类。在开始的案例中，我们要保持类的简洁；这些类包含了一些自动属性，并没有什么逻辑。

Example 2-1. The domain model

public class Destination

{

public int DestinationId { get; set; }

public string Name { get; set; }

public string Country { get; set; }

public string Description { get; set; }

public byte[] Photo { get; set; }

public List<Lodging> Lodgings { get; set; }

}

public class Lodging

{

public int LodgingId { get; set; }

public string Name { get; set; }

public string Owner { get; set; }

public bool IsResort { get; set; }

public Destination Destination { get; set; }

}

Destination类描述了一个特定的BAGA旅行的目的地，对于任何给定的目的地，从Aspen到Zimbabwe，BAGA都会安排各种住所，从床位到五星级酒店，以便到时暂住。因此目的地对象要包含一个或多个居所，使用了List<lodging>来表达；

## EF域类介绍

本质上,这些类与EF和Code First无关。这里只简单描述了域的一部分。

为了让EF框架能够找到这些类，需要使用EF框架的context来服务、管理和持久化数据至数据库。EF框架有两种context工具可供选择，一个是ObjectContext,这一工具从EF第一次发布就一直是EF框架的一部分，而随着EF4.1的发布，伴随Code First推出轻量级的DbContext。两种工具都可选用，但更通用（推荐）的是使用新的DbContext,也就是我们马上就要使用的。本书第7章，你将会学习如何使用ObjectContex进行Code First.

我们的BreakAwayContext类，就继承自DbContext，可以获得DbContext的所有功能。除此以外，还需要返回Destination类和Lodging类的可查询数据集DbSets，以暴露类中的属性

Example 2-2 The BreakAwayContext class

public class BreakAwayContext : DbContext

{

public DbSet<Destination> Destinations { get; set; }

public DbSet<Lodging> Lodgings { get; set; }

}

 这个小小的类就代表了你在应用程序中使用的完整的数据层.感谢DbContext,你可查询,修改,跟踪和保存目的地和住所数据.下面创建一个控制台程序使用这个数据层来做一些工作,你会看到我们并非言过其实.

## 将片断放入程序中

为了查看有关行为,下面将带您实现一个小的VS解决方案,在此你可将这些类放进去，然后创建一个简单的控制台程序来测试你的新数据层.确保你已经开始了正确的路径,本示例将把应用程序的各个层组织在不同的项目中.

1.在VS中创建一个新的解决方案;

2.添加一个类库项目到解决方案,命名为Model.

3.在这个项目里,添加一个新的类命名为Destination.

4.修改Destination类以使其与例2-3一致:

Example 2-3. The Destination class

using System.Collections.Generic;

namespace Model

{

public class Destination

   {

public int DestinationId { get; set; }

public string Name { get; set; }

public string Country { get; set; }

public string Description { get; set; }

public byte[] Photo { get; set; }

public List<Lodging> Lodgings { get; set; }

    }

}

5.添加另一个类,命名为Lodging,使其与例2-4一致.

Example2-4.The Lodging class

namespace Model

{

public class Lodging

  {

public int LodgingId { get; set; }

public string Name { get; set; }

public string Owner { get; set; }

public bool IsResort { get; set; }

public Destination Destination { get; set; }

  }

}

这就是Model项目的内容.现在,转到数据层,因为域类尚未连接到EF框架上,我们数据层将完全依赖于EF框架.

小贴士：将NuGet安装到VS中就可以将类库方便地添加到项目中.在第1章您已学到EF可以通过NuGet获取.为了完成如下步骤,你需要为VS安装NuGet扩展.为了安装NuGet,选择工具--扩展管理….会出现扩展管理对话框,然后查找NuGet.选择NuGet扩展管理包,点击下载按钮,按照提示安装.

一旦安装完成,您就可以以一种新的方式使用VS了.你可通过VS的工具菜单--添加类库来添加所需的类库,另一种方式是使用上下文菜单来添加,方法是在解决方案的指定项目上点击右键.在下面的操作步骤中,我们使用后者

1.添加另一个类库项目,命名为DataAcess;

2.在项目资源管理器中右键单击新建立的项目选择"Add Livray Package Reference"

3.在弹出的对话框中,选择Online,并搜索Entity Framework;

4.在EF安装包上点击安装按钮,这将把Code First的运行时组件(EntityFramework.dll)添加到你的项目中;

5.右键单击新的项目并选择添加引用;

6.选择项目选项卡将Model proect加入.这将会使context能够访问刚刚在Model Project上创建的域类;

7.添加一个新类,BreakAwayContext到项目中;

8.配置这个新类如代码2-5;

Example 2-5. The BreakAwayContext class

using System.Data.Entity;

using Model;

namespace DataAccess

{

public class BreakAwayContext : DbContext

   {

public DbSet<Destination> Destinations { get; set; }

public DbSet<Lodging> Lodgings { get; set; }

   }

}

注意类顶部的using语句.一个是System.Data.Entity命名空间.这个命名空间使得你可以访问DbContext和DbSet类;这可能有些令人困惑,命名空间为EntityFramework.dll的程序集是 EntityFramework.dll,而不是System.Data.Entity.dll。System.Data.Entity.dll包含了EF框架的核心，已经是.Net框架的一部分.

现在你的数据访问层已经建好了.可以测试你的数据访问层了.

请等一下,我们还没有告诉数据访问层数据库在哪.还没有连接字符串,没有配置文件,没有关联的数据库.我们要在此利用Code First的第一个功能,数据库初始化.Code First有一系列的步骤来找到数据库并且初始化它.我们现在先使用默认行为,在后面第6章,将会学到更多关于数据库初始化的知识。当然,也可以开始于一个现存的数据库,只是我们现在不这么做。

1.添加一个新的控制台应用程序项目到解决方案,命名为BreakAwayConsole.

2.右键单击新的项目,设置为启动项目;

3.右键单击项目,选择添加类库引用,添加对EF框架的引用;

4.再次右键单击项目,选择添加引用,将Model和DataAcess项目添加至项目;

5.在新项目中,打开Program类;

6.添加两个using语句在文件顶部:

using Model;

using DataAccess;

7.在类中添加一个方法,命名为InsertDestination(代码2-6);

Example 2-6 The InsertDestination Methoed

private static void InsertDestination()

{

var destination = new Destination

   {

Country = "Indonesia",

Description = "EcoTourism at its best in exquisite Bali",

Name = "Bali"

   };

using (var context = new BreakAwayContext())

  {

context.Destinations.Add(destination);

context.SaveChanges();

  }

}

8.在类中调用这个方法(代码2-7)

Example 2-7. Calling the InsertDestination method

static void Main()

{

InsertDestination();

}

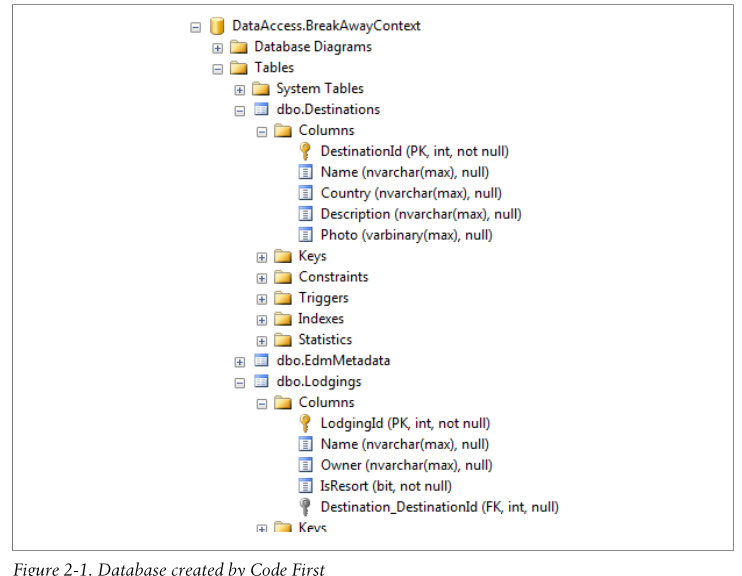
完成！对这两个域类你只用了少量代码,两行数据访问类以及这个小的应用类,你已经开始准备运行应用程序了。

这个控制台程序并不会显示任何信息.有趣的是发生数据库里的事情：数据库刚才并不存在。

Code First 通过检视Destination 和 Lodging 类确定了数据模型,然后根据类属性推断出数据库的架构模型.既然我们没有提供连接字符串,它就使用默认的约定,查找本地SQL Server Express实例(localhost\SQLEXPERSS)作为数据库宿主,然后以context类的类名DataAcess.BreakAwayContext匹配数据库名.没有找到这样的数据库,Code First就创建它,然后使用它在模型中发现的类依据约定创建表和表的字段。

我们来看看SQL Server Express中Code First创建的数据库(图2-1).

可以看到,DataAccess.BreakAwayContext 有三个表:Destinations ,Lodgings和EdmMetadata,后者用于Code First的数据库初始化,后面我们会提到.

[](file:///C:\Users\æå¨\AppData\Local\Temp\WindowsLiveWriter-148658618\supfilesB9B689E\image8.png)

## 关于表,构架,和字段名的约定

 Code First 约定表名使用EF框架的复数化服务(由EF4引入)，即使用英语语法的类名复数形式来命名表名。（这个功能可以关闭，见第7章，译者注）默认情况下,每个表都使用dbo构架创建。Code First 属性映射的列使用与类中属性一致的名字命名.

## 关于主键的约定

再次观察数据库表,你会找到Code First已经使用的默认规则。例如,Code First知道Detination类的DestinationId和Lodging类的LodgingId意思是键值,然后将这些键映射为数据库的主键。注意这些键都是非空主键(PK,notnull)。Code First默认约定将命名为Id或[类名]Id的属性视为类的键。DestinationId 和LodgingId显然符合这一规则。由于这些属性都是整型类型,Code First还会将它们配置为数据库的标识字段。这表示在插入数据时数据库会为这一字段自动生成值。

## 字符串属性的约定

字符串约定为映射到不限长度的非空列中。由数据库引擎来负责确定映射到何种类型。对于SQL Server而言，默认数据类型为nvarchar(max)。这就是你在图2-1看到的所有的字符串属性都映射成了nvarchar(max)列的缘故---Destination.Name,Country,Description等等.--同时还允许存储空值;

## 针对Byte数组的约定

Destination类有一个Photo属性,定义为byte[].Code First 约定byte数组映射到不限长度的非空列中.对SQL Server而言就是varbinary(max)类型.

## 布尔值的约定

Lodging类中的IsResort属性是一个bool类型.由于bool是一个值类型,不能分配给其一个null值.Code First强制要求此列不能为空.对SQL Server而言将bool属性映射为bit 数的库类型.

## 一对多关系的约定

一个目的地会有很多居所,因此Destination类有一个List<Lodging>属性允许你获取特定目的地的所有居所信息.同时,Lodging类也有一个Destination类型的属性,,因此你可看到某一居所与某一特定的目的地相连接.Code First将这种情况视为一对多关系，约定Lodging表有一个外键约束目的地与居所间的所有关系.

注意尽管在Lodging类中没有外键属性指向Destination类(如DestinationId)，Code First仍然使用默认规则创建了一个,形如[Name of navigation property]\_[Primary Key of related class](即Destination\_DestinationId)，这是Code First所为。使得EF框架知道在从数据库中查询或保存时要使用外键.

在目的地和住宿类提供给Code First的导航属性不是一个，而是两个。如果我们只提供了其中之一，关系仍然明显，Code First还是会在数据库中创建外键。还有很多提供外键的约定适用于某些场景,如多对多关系等,在后面我们会深入讨论。

## 使用配置来覆写约定

正如在第1章学到的,Code First允许你覆写约定,方法是添加配置.可以选择使用Data Annotation的特性标记也可以使用强类型的Fluent API来配置.

您提供的任何配置将会作为模型的一部分，EF框架使用此模型在运行时解析数据。这不仅影响数据库构架，也会影响DbContext内置的验证功能。例如，如果你告诉Code First一个属性是必备的，验证API将让你知道，如果该属性没有被填充的后果。在后续我们将看到这一行为。

Data Annotations是最简单的配置方式，直接应用到你的类和类属性上。这些特性位于System.ComponentModel.DataAnnotations命名空间，目前分布在System.ComponentModel.DataAnnotations.dll和EntityFramework.dll两个程序集中。在未来版本的.NET Framework中，EntityFramework.dll中的Annotations将迁移到System.ComponentModel.DataAnnotations.dll。所以需要在域类的项目中引用上述程序集（取决于您使用的Annotations）。注意Data Annotations能够完成常用的配置，但并非所有Code First配置都可以使用Data Annotation来完成。一些情况下只能使用另一种风格的配置方式:Fluent API。

因为Code First没有自动发现Destination和Lodging类中的某些意图，需要用一些Data Annotations提供额外的配置细节。

http://images.cnblogs.com/cnblogs_com/qouoww/201112/201112312122158003.png

小贴士：在C＃和Visual Basic应用特性

如果你第一次使用特性，在C＃中，特性应使用方括号。例如，使用Annotation 特性Key,在c#中使用[Key]，而在Visual Basic中，使用尖括号（<Key>）。当一个特性使用参数时，在C＃中是表达了一个等号表达([Table(Schema="baga")]),而Visual Basic使用冒号等号表示（<Table(Schema:="baga")>） 要知道.NET代码中使用特性的更多信息，请参阅MSDN主题"应用特性"，在<http://msdn.microsoft.com/en-us/library/>bfz783fz.aspx。

让我们从Destination类开始.关于这个类我想要作三个调整:

* 必须提供目的地的名字
* 限制描述字段的文本内容在500个字符以内
* 保存照片到SQL Server数据库应为image类型,而不是varbinary(max).

上述需要的Data Annotations特性有两个包含在System.ComponentModel.DataAnnotations.dll程序集中,这是.Net 4的一部分,有一个包含在EntityFramework.dll程序集里，所以需要对两个程序集都要添加引用：

1.在Model项目中,添加对System.ComponentModel.DataAnnotations 程序集的引用.

2.通过类库包装引用添加对EntityFramework程序集的引用.

记住:你必须对每个项目都运行NuGet类库添加向导,即使类库包已经添加到解决方案中的某个项目中.使用同样的步骤将EntityFramework.dll添加到DataAccess项目中.

3.在Destination类顶部,添加System.ComponentModel.DataAnnotations命名空间.

4.修改类如代码2-8所示Example 2-8. Modified Destination class

using System.Collections.Generic;

using System.ComponentModel.DataAnnotations;

namespace Model

{

public class Destination

   {

public int DestinationId { get; set; }

[Required]

public string Name { get; set; }

public string Country { get; set; }

[MaxLength(500)]

public string Description { get; set; }

[Column(TypeName = "image")]

public byte[] Photo { get; set; }

public List<Lodging> Lodgings { get; set; }

   }

}

Required特性标记不需要附加信息,而MaxLength和Column特性均需要提供参数.这些参数对如何映射到数据库的列进行了明确。我们想要图片储存在SQL Server的image字段里.

小贴士：只要有可能强制将数据库中字段的类型设定为指定类型(例如将byte[]强制指定为image),你可配置数据类型特性。所有的三个特性将会影响数据库的构架,其中之二,Required和MaxLength,也被用于EF框架的验证之用.在观察效果之前,也对Lodging类作些类似的修改.

Annotation特性标记可组合使用,也就是在一个类或属性上可以附加多个annotations特性.我们准备在Lodging.Name属性附加多个配置特性.

添加下述三个注解到Lodging类的Name属性:

[Required]

[MaxLength(200)]

[MinLength(10)]

MinLenght是一个有趣的annotation特性.MaxLength在数据库有对应的含义,而MinLength并不有.MinLength将会用于EF框架的验证,并不会影响数据库.

MinLength只能通过Data Annotations来进行配置,在Fluent API 中无对应项.

http://images.cnblogs.com/cnblogs_com/qouoww/201112/201112312122165986.png

### 理解模型变化如何影响数据库初始化

如果再次运行控制台程序,你会得到一个InvalidOperationException异常.我们刚刚对模型作的修改并没有什么错误,问题在于Code First数据库初始化的默认行为.因此我们需要在继续探索配置时需要修复这些问题.

下面是异常的信息：

*http://images.cnblogs.com/cnblogs_com/qouoww/201112/201112312122162364.png*

The model backing the 'BreakAwayContext' context has changed since the database was created. Either manually delete/update the database, or call Database.SetInitializer with an IDatabaseInitializer instance. For example, the DropCreateDatabaseIfModelChangesstrategy will automatically delete and recreate the database, and optionally seed it with new data.

自上次数据库创建以来,'BreakAwayContext'的上下文已经发生变化.要么手工删除或更新数据库,要么调用IDatabaseInitializer接口的实例 Database.SetInitializer.例如 DropCreateDatabaseIfModelChanges策略将自动删除和再创建数据库,并且可选择使用新数据作为种子.

http://images.cnblogs.com/cnblogs_com/qouoww/201112/201112312122161808.png

.默认情况下,Code First只有当数据库不存在的时才创建数据库,但目前数据库已经有了，该怎么办呢？

是否还记得EdmMetadata表?如果没有,回头看一下图2-1.这个表包含了Cord First创建数据模型的部分快照。默认情况下,它会伴随着Code First第一次使用模型建立数据库时创建。然后它就会将表中的版本信息与内存中的模型相比较(这可以从EdmMetadata表中读出）。当Code First识别到上一模型的元数据模型与新建立的模型不匹配,就无法将模型映射到数据库.

您有很多选择.一个是简单删除数据库(包括其中的所有数据),然后让Code First使用默认规则(无数据库--创建新数据库)来使用更新模型创建数据库.这可能很痛苦,特别是你在开发过程中,可能会遇到某些情况导致文件锁定而无法删除.另一种方法,我们马上要用,Code First在遇到模型变化时有一套初始化策略可供使用，该行为删除数据库并重建，默认是封装在名为CreateDatabaseIfNotExists的类中，你可告知正在执行中的程序(本处是指控制台程序)使用哪个策略。

修改Main方法使之与例2-9一样.也需要在添加一个using语句,引用System.Data.Entity.

Example 2-9. Adding Database Initialization to the Main method

static void Main(string[] args)

{

Database.SetInitializer(

new DropCreateDatabaseIfModelChanges<BreakAwayContext>());

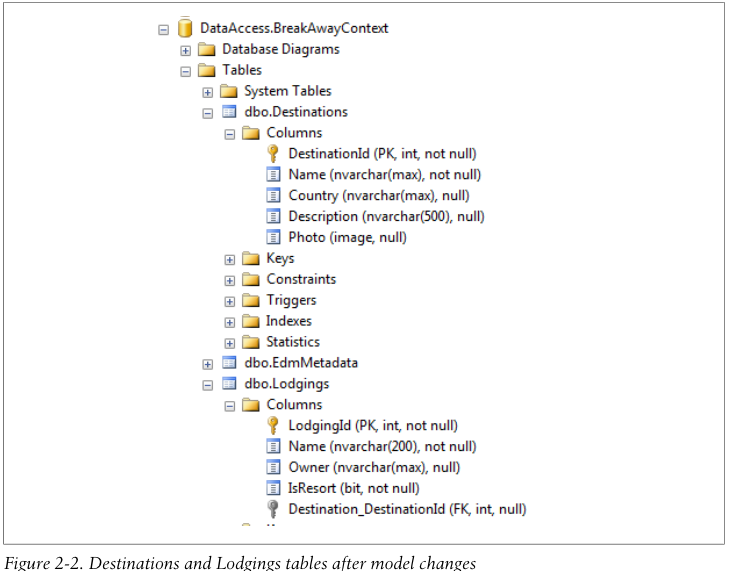
InsertDestination();

}

 在此代码中我告诉Code First使用初始化器并且指定使用何种策略.(这里使用了DropCreateDatabaseIfModelChanges)应用于上下文(BrakAwayCOntext).现在当你返回应用程序,Code First将识别新旧模型的区别,通知初始化器,删除和重建数据库.

如果你已经在其他地方打开了数据库表读取数据，（例如，在Visual Studio的服务器资源管理器），Code First将不能够删除数据库。在这种情况下，在EF框架尝试删除时会有一个延迟，最终因为无法删除而抛出一个异常。似乎这样做太繁琐，特别是当我在用户组和会议展示时。我遇到的一个常见的场景是，我已打开了SQL Server Management Studio（SSMS）中，对数据库进行一些查询。你不得不关闭SSMS的数据库完全释放对数据库的锁定。

http://images.cnblogs.com/cnblogs_com/qouoww/201112/201112312122166235.png

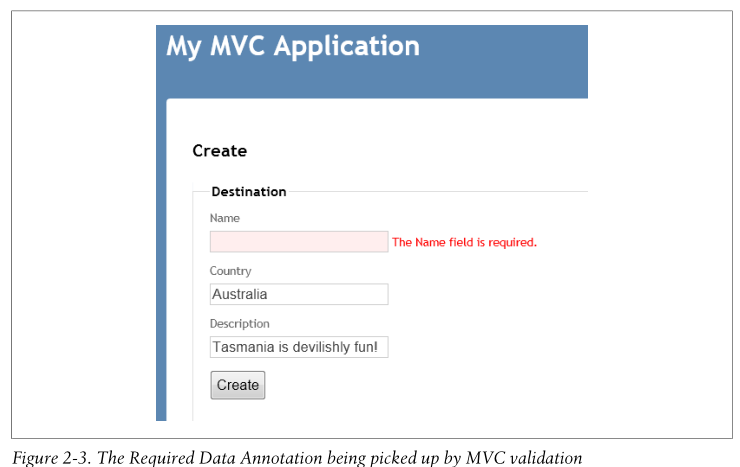
[](file:///C:\Users\æå¨\AppData\Local\Temp\WindowsLiveWriter-148658618\supfilesB9B689E\image11.png)

Destination类之Destinations表有三个可见的改变,Name设定为必须,在数据库表现为非空字段.Description现在是nvarchar(500)而不再是max,Photo也已经是一个image数据类型.Lodging表也得到影响,Name现在限制为200个字符且为非空型,第三个annotation:MinLength在数据库构架中没有等效值,因此在此忽略.但是,EF框架将会对此进行验证.

### Data Annotations和前端验证提示

NET 4中引入的Data Annotations，可以用于ASP.NET(含.Net MVC)作为用户界面的动态数据验证工具来使用。许多Code First 使用的annotations特性标记都来自源于System.ComponentModel.DataAnnotations程序集。因此，这些数据验证功能是.Net程序集的的内置特性。

例如，"Required"特性在.NET4的程序集中，而不在EntityFramework.dll程序集里。因此，如果你在MVC应用程序中使用的Code First类，并且忽略填写一个必须输入的属性，MVC用户界面验证会响应，你可以在图2-3看到。

[](file:///C:\Users\æå¨\AppData\Local\Temp\WindowsLiveWriter-148658618\supfilesB9B689E\image14.png)

http://images.cnblogs.com/cnblogs_com/qouoww/201112/201112312122175297.png

DbContext还提供了一个验证API，具有服务器端验证的能力，这与使用Data Annotations特性标识还是使用Fluent API来配置类是没有关系的。但是，使用Data Annotations时,MVC与特性更好地配合，实现一些验证API的功能。验证API的细节内容不包括在这本书中，本书的重点在于Code First建模，你可在另一本书:Entity Framework:DbContext找到与验证API有关的信息。

http://images.cnblogs.com/cnblogs_com/qouoww/201112/201112312122171153.png

## 使用Fluent API 配置Code First

使用Data Annotations配置很简单，也可能是你想要的。但是Data Annotations只能获得一部分配置的功能。这时,Fluent API可以提供更多的功能,基于这个原因你可能会愿意用它.

小贴士：Fluent API是什么?

Fluent API并非专用于Code First或EF框架。Fluent API的基本思路是使用链方法调用程序代码，这样代码很容易被开发者所阅读。每一个调用的返回类型都定义为下个调用的有效方法.例如,在Code First的Fluent API中,你可以使用Entity方法选择一个实体来配置.智能感知将会显示出所可以用来配置的方法.如果你使用属性方法选择属性来配置,你就看到所有为特定属性所配置的方法.

http://images.cnblogs.com/cnblogs_com/qouoww/201112/201112312122179724.png

还有另一个原因解释为什么开发者首选Fluent API。在将标记应用到你的美妙域类上时,需要定义越来越多的烦人的标记。目前还只是一些验证逻辑,当你学到更多Data Annotations配置特性的时候,你就会发现需要更多的信息指导类如何映射到数据库。如果你喜欢清洁的类,不想让类包含那么多的标记，就可以选择Fluent API。Code First的一个优势就是能够使用自己的类构建EF框架，一个充斥着数据库表名或列类型的类显然不够简洁，Fluent API解决了这一问题,我们来看这是如何实现的。

http://images.cnblogs.com/cnblogs_com/qouoww/201112/201112312122182756.png

小贴士：跟随Data Annotations还是跟随Fluent API  ？

在本书后续内容中,你将会看到很多例子来说明如何配置映射,有Data Annotations的,也有使用Fluent API的.

许多映射只能通过Fluent API获得.如果你在VS中跟随我们学习,强烈推荐您创建两个单独的解决方案。否则,当使用了Data Annotation中还想看看在Fluent API中同样的配置效果,你就不得不注释掉有关代码。然后当你使用另一个Data Annotation特性标记，你又不得不注释掉Fluent API代码.在注释与取消注释之间来回调整,万一出现错误就会导致抛出异常,要么是配置重复要么是配置丢失，可能会非常麻烦。

这就意味着在创建新类或添加新方法时需要进行许多必要的复制/粘贴操作.也意味着Data Annotations 只能应用于一个解决方案而Fluent 配置被强制用于另一个.

如果你遵循此建议,还有一个附加的建议.确保在两个解决方案使用不同的命名空间。如果其中一个解决方案使用了DataLayerForAnnotations命名空间,其数据库将为DataLayerForAnnotations.BrakeAwayContext。另一解决方案使用了DataLayerForFluent,命名空间,其数据库将为DataLayerForFluent.BrakeAwayContext。你会看到有两个独立的数据库，很容易理解哪个解决方案影响了哪个数据库。

http://images.cnblogs.com/cnblogs_com/qouoww/201112/201112312122181643.png

DbContext首先在类中查找可以获取的信息。这时，context已经准备好解析模型,但对开发者来说有一个机会来中断context和执行附加配置的连接.这得益于DbContext.OnModelCreating方法，这是在模型创建前被上下文所调用的方法。方法是虚拟的,因此你可以覆写并加入你自已的逻辑代码.这就Fluent API进行配置的入口.

此方法的声明方式如下：

protected override void OnModelCreating(DbModelBuilder modelBuilder)

 提供给OnModelCreating的参数DbModelBuilder是你需要添加配置的类。DbModelBuilder使用泛型和lambda表达式，所以编码是强类型的可以协助您进行配置的设置。首先要告诉DbModerBuilder对哪个类(实体)实施配置:

modelBuilder.Entity<Destination>() ;

 你可以配置类映射到数据库的表名:

 modelBuilder.Entity<Destination>().ToTable("a\_table\_name");

你也可配置类的属性.如果想要配置属性,应进一步演进:

modelBuilder.Entity<Destination>()  
.Property(d => d.Description).HasMaxLength(500);

代码2-10重写了前面使用data annotations建立的配置。OnModelCreating是DbContext的一个方法,请确保它在BrakAwayContext类里。

Example 2-10. Configuring with the Fluent API

[复制代码](javascript:void(0);)

protected override void OnModelCreating(DbModelBuilder modelBuilder)  
 {  
   modelBuilder.Entity<Destination>()  
    .Property(d => d.Name).IsRequired();  
   modelBuilder.Entity<Destination>()  
    .Property(d => d.Description).HasMaxLength(500);  
   modelBuilder.Entity<Destination>()  
   .Property(d => d.Photo).HasColumnType("image");  
   modelBuilder.Entity<Lodging>()  
   .Property(l => l.Name).IsRequired().HasMaxLength(200);

[复制代码](javascript:void(0);)

  }

如果你疑惑HasMinLength哪里去了,在Fluent配置方法里没有最小长度的配置,因为这不会对数据库的行产生影响.

除非要配合使用Data Annotations 和 Fluent API为了保持代码一致最好选择其一。为了让代码更流畅,我们已经移除了所有的Data Annotation标记.实际上Model项目也不需要对System.Component Model.DataAnnotations 或者EntityFramework程序集的引用.

再次运行代码,Code First将对比模型和数据库中的EdmMeta表,尽管我们修改了配置代码,最终结果还是一样的.同样的destination再次加入数据库中结束与匹配的记录.在图2-4你可以看到复制到的数据.

[](file:///C:\Users\æå¨\AppData\Local\Temp\WindowsLiveWriter-148658618\supfilesB9B689E\image17.png)

## 组织Fluent配置

如果你有很多配置需要执行,OnModelCreating 可能很快不堪重负(代码太多)。应该使用位于EntityTypeConfiguration的实体类来分组配置，然后在OnModelCreating方法中告诉DbModelBuilde有些实体类r。DbModelBulider有一个Configruation属性可以来增加EntityTypeConfigurations(实体类型配置).

例2-111展示了对Destinaton类和Lodging类的分组情况:

Example 2-11. Organizing configs into separate EntityTypeConfiguration classes

[复制代码](javascript:void(0);)

using System.Data.Entity.ModelConfiguration;  
using Model;  
public class DestinationConfiguration :  
EntityTypeConfiguration<Destination>  
 {  
    public DestinationConfiguration()  
     {  
        Property(d => d.Name).IsRequired();  
        Property(d =>   
         .Description).HasMaxLength(500);  
        Property(d => d.Photo).HasColumnType("image");  
     }  
 }  
public class LodgingConfiguration :  
EntityTypeConfiguration<Lodging>  
 {  
    public LodgingConfiguration()  
    {  
        Property(l => l.Name).IsRequired().HasMaxLength(200);  
    }  
 }

[复制代码](javascript:void(0);)

这些代码在OnModelCreating方法内部的时候，由DbModelBuilder开始，后面跟着Entity方法来确定哪个实体进行配置。在EntityConfiguration类中，是由继承于EntityTypeConfiguration类的类开始的，这里的实体类型已经指定。例如，modelBuilder.Entity<Destination>().Property这个语句是对属性进行设置，而调用的modelBuilder.Entity<Destination>()实际上是创建了一个EntityTypeConfiguration<Destination>对象并返回给你，然后在属性上配置。所以无论使用哪种方式，访问的是同一个API。

代码2-12,你会看到修改的OnModelCrating方法,来使用这些类.

Example 2-12. Adding the configuration classes in OnModelCreating

protected override void OnModelCreating(DbModelBuilder modelBuilder)  
  {  
    modelBuilder.Configurations.Add(new DestinationConfiguration());  
    modelBuilder.Configurations.Add(new LodgingConfiguration());

   }http://images.cnblogs.com/cnblogs_com/qouoww/201112/201112312122184085.png

随着我们转向Fluent配置的例子，我们有时会将配置放在EntityTypeConfiguration类,有时放置在modelBulider中。这仅仅是为了让你继续看到这两种语法。但是，在最终产品中，很显然不能混用配置。你更希望有一个统一的方式，无论是将所有配置都写在OnModelCrating方法内部还是组织在EntityTypeConfiguration类中。你会看到有几个配置操作不是类型相关的，必须要直接在OnModelCreating方法中调用。

## 小结

本章对Code First作了基础介绍.我们看到Code First的约定规则非常智能,能够通过类的构建获知您的意图。当约定无法正确推断时,你需要使用配置明确控制Code First如何构建模型和数据库构架。我们介绍可以直接在类上采用Data Annotations的特性来实现配置。而对那些更愿意让域类独立的情形,你还学到可以使用替代的Fluent API在DbContext类中来实施配置.

Code First 可以自动为你构建数据库。你已经看到如何施加控制到其响应以对模型作出调整。期待重建数据库来匹配新的模型.现在你应该已经喜欢上了Code First,我们将进一步深入。扩展模型,学习使用Code First的来龙去脉.

# 第三章：对属性使用约定和配置

在第2章，对Code First的约定以及如何通过配置覆写默认约定行为进行了大致的介绍。学习了如何使用Data Annotations进行配置，也学习了如何使用Fluent API作出相同的配置，并对两者进行了对比。

在本章乃至以后几章里，将深入各种用于配置模型的领域。对每个主题会看到Code First如何通过默认规则进行工作，也会学到如何通过Data Annotations和Fluent API来覆写这些规则。前已指出，在Fluent API中可以实现的很多配置在Data Annotations无法实现。我们会在适当的时机指出这些差异。

本章专注于对类中属性的配置，以观察默认规则和配置对数据库列的影响。你将会学习到诸如如何控制字符串长度，byte 数组，数值的精度等方面的知识。你也可以学到键属性以及所谓的“开放式并发属性”。最后，您还可以学到有关Code First检测一个属性是否是复杂类型（aka值类型），如果Code First无法从您的域类中推断出复杂类型时，我们将都会您如何对Code First提供帮助以识别复杂类型。

## 在Code First中使用属性

在第2章里，您已经看到一些应用于字符串属性的规则和配置选项，在进入新的选择我们快速回顾一下。

## Length

## 字长

|  |  |
| --- | --- |
| Convention  默认规则 | max (type specified by database)  max(类型由数据库指定) |
| Data Annotation | MinLength(nn)  MaxLength(nn)  StringLength(nn) |
| Fluent | Entity<T>.Property(t=>t.PropertyName).HasMaxLength(nn) |

字长用于描述数组的长度。包括对字符串和byte数组。

Code First的默认规则string 或者byte数组的长度应为最大。根据不同的数据库类型确定在数据库最终的类型。对SQL Server而言，string 会生成nvarchar(max)，而byte数组会生成varbinary(max).

你可以覆写默认长度来设置在数据库中的实际字长。长度的最大值会在EF框架将更新数据存入数据库之前进行验证。如果使用Data Annotation来配置，还可以为数组配置MinLength(最小长度)特性。最小长度特性也会得到EF验证API的验证，但不会影响数据库。

#### 数据类型

|  |  |
| --- | --- |
| Convention  默认规则 | The default column data type is determined by the database provider you are using. For SQL Server some example default data types are:  默认的列数据类型由数据库决定，对SQL Server而言如下：  String : nvarchar(max)  Integer:int  Byte Array:varbinary(max)  Boolen:bit |
| Data Annotation | Column(TypeName="XXX") |
| Fluent | Entity<T>.Property(t=>t.PropertyName).HasColumnType("XXX") |

第2章，您已经看到了几个如何映射.Net类型到数据库数据类型的例子。Destination和Lodging类包含有整型，字符串，Byte数组，布尔型变量。Code First通知数据库选择合适的数据类型匹配每一列。由于使用的是SQL Server数据库，因此分别映射到nvarchar(max), int, varbinary(max)和bit类型。

根据您选择的配置当然也可以指定到基他类型。例如，将字符串映射到数据库的int数据类型，运行时DbModelBuilder就会抛出一个错误告知映射非法，然后会给出如何进行纠正的细节指示。

#### 可空性和必需项配置

|  |  |
| --- | --- |
| Convention  默认规则 | Key Properties : not null in database  键属性：在数据库中为非空  Reference Types (String, arrays): null in the database  引用类型（String,数组）：在数据库中可空  Value Types (all numeric types, DateTime, bool, char) : not null in database  值类型（所有数字类型，日期，布尔，字符）：在数据库为非空  Nullable<T> Value Types : null in database  Nullable<T>值类型（可空类型）：在数据库可空 |
| Data Annotation | Required |
| Fluent | Entity<T>.Property(t=>t.PropertyName).IsRequired |

默认规则约定确保非可空的.Net类型要映射到数据库的非可空字段，除此以外，任何键属性都只能映射到非可空数据库字段。

如果你使用.Net的泛型Nullable<T>指定一个值类型（如int）为可空，将会映射到数据库的一个可空字段。

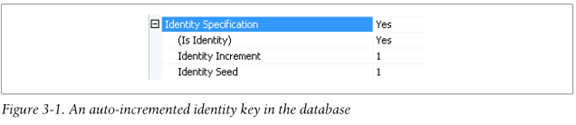
在第2章您已看到如何使用配置指定一个属性为必须项。使用Data Annotation的Required标记和Fluent的IsRequired属性都可强制Lodging.Name属性为必须项。在保存数据到数据库之前，EF运行时会对必须属性进行验证；如果属性没有赋值就会抛出一个异常。另一个效果是，数据库相应字段为非空。

## 映射键

|  |  |
| --- | --- |
| Convention  默认规则 | Properties named Id  属性名为Id  Properties named [TypeName] + Id  属性名为[类型名]+Id |
| Data Annotation | Key |
| Fluent | Entity<T>.HasKey(t=>t.PropertyName |

EF框架要求每个实体都有一个键。这个键用于上下文以保持每个独立对象的跟踪。键是唯一的而且经常由数据库生成。Code First默认规则作出了同样的预设。

回忆一下由Destination和Lodging类生成的数据库，DestinationId和LodgingId的整型字段都被标记为主键和非空字段。如果进一步观察二者的列属性，你会发现这些字段是自增长的标识字段，如图3-1所示，这是默认规则将整型量作为主键来管理。



大多数情况下，数据库中的主键不是int就是GUID类型，尽管任意类型都可以作为键属性。数据库中的主键会是多个表的组成字段，类似地，一个实体的键也是某个类中的多个属性之一。在本节结束的时候，你会看到如何配置复合键。

## Code First默认规则对不合规键属性的响应

如果在我们的类中我们意指的键碰巧满足Code First默认规则，那么一切顺利。但是如果不满足规则呢？

我们向模型添加一个新类，Trip，见代码3-1.Trip类没有任何满足实体键默认规则的属性，但我们的意图是Identifier属性应该作为键。

Example 3-1. The Trip class without an obvious key property

 public class Trip

[复制代码](javascript:void(0);)

{  
   public Guid Identifier { get; set; }  
   public DateTime StartDate { get; set; }  
   public DateTime EndDate { get; set; }  
   public decimal CostUSD { get; set; }  
}

[复制代码](javascript:void(0);)

伴随这个新类，我们需要在BrakAwayContext中添加一个DbSet<Trip>数据集：

public DbSet<Trip> Trips { get; set; }

 我们再次运行程序，在尝试从类中创建模型时DbModel Builder抛出一个异常：

在模型生成过程中检测到一个或多个验证错误：

System.Data.Edm.EdmEntityType: :

实体类型"Trip"还没有定义key。请为这个实体类型定义Key.

由于没有找到期望的默认Key属性（Id或TripId）,Code First无法继续创建模型。需要明确的是，类型（GUID）与这个问题无关。如前所述，您可以使用任何的原始类型作为键。

## 使用Data Annotations配置Key

Data Annotation标识一个键只需要简单的一个Key.Key特性位于System.ComponentModel.DataAnnotations.dll，由于它已经被添加到了.Net4之中，也被其他API所使用（如ASP.Net MVC使用的Key）.如果你的项目尚未包含此程序集的引用，你就不能添加它。对这个特定的特性不需要引用EntityFramwork.dll。

[Key]  
public Guid Identifier { get; set; }

## 在Flurent API中使用HasKey来配置Key属性

使用Fluent API来配置Key属性与前面几个Fluent配置不同。这一配置直接添加到实体上。为了配置一个key，你需要使用HasKey方法，如代码3-2。

Example 3-2. The HasKey Fluent configuration in OnModelCreating

modelBuilder.Entity<Trip>().HasKey(t => t.Identifier)

如果将代码配置进EntityTypeConfiguration类中，正如你在第2章学到的，应该开始于HasKey或This.HasKey(代码3-3)

Example 3-3. HasKey inside of an EntityTypeConfiguration class

HasKey(t => t.Identifier)

## 配置数据库生成的属性

|  |  |
| --- | --- |
| Convention  默认规则 | Integer keys:Identity  整型键值：标识列 |
| Data Annotation | DatabaseGenerated(DatabaseGeneratedOption) |
| Fluent | Entity<T>.Property(t=>t.PropertyName)  .HasDatabaseGeneratedOption(DatabaseGeneratedOption) |

在前面部分里，你已经看到默认情况整型键值会被EF框架生成标识字段，由数据库生成值。而我们自己创建的Guid型的键值怎么办？Guid需要特殊的处置，包含在DatabaseGenerated配置。

为了展示，我们添加一个新方法，InsertTrip(代码3-4)到控制台程序，然后在主模型进行调用。

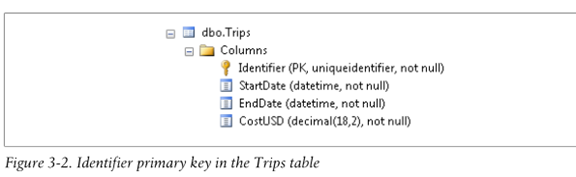
Example 3-4. The InsertTrip method

[复制代码](javascript:void(0);)

private static void InsertTrip()  
{  
   var trip = new Trip  
   {  
     CostUSD = 800,  
     StartDate = new DateTime(2011, 9, 1),  
     EndDate = new DateTime(2011, 9, 14)  
   };  
   using (var context = new BreakAwayContext())  
    {  
      context.Trips.Add(trip);  
      context.SaveChanges();  
    }  
}

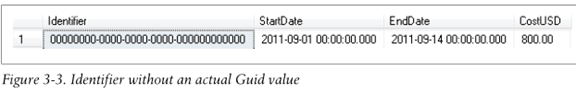
[复制代码](javascript:void(0);)

运行程序会导致数据库卸载并增加新的Trips表后重新创建，如图3-2.Identifier是主键，唯一标识，非空列。



回到本章前面的内容，你知道值类型默认是required。在此也会看到同样的效果，StartDate,EndDtat和CostUSD属性都是值类型，默认情况下，在数据库也都是非空字段。

然后在新行中我们看到Guid值被填充为很多个0.如图3-3



数据库和EF框架都不知道我们想让他们之一为新添加的Trips生成一个新的Guid。由于这个属性没有一个生成新Guid的逻辑方法，就会默认以0值填入。

如果你尝试以同样的值插入另一个记录，数据库会抛出一个错误，因为期待一个唯一值。当然可以配置数据库自动生成一个新的Guid（通过设置默认值为newid()。不管你在数据库中手动操作还希望CodeFirst插入此逻辑，你必须让Code First知道数据库将要处理Guid.

解决方案是让Code First 知道数据库将要生成这个键值通过使用另一个annotation:DatabaseGenerated.这一配置有三个选项—None,Identity和Computed.我们想要Identifier字段被标识为Identity，才能确保数据库在加入新行时自动生成标识字段的值，正如整型类型的键值自动生成一样。

## 使用Data Annotations配置数据库-生成选项

修改类代码告诉Code First让数据库生成一个唯一的键值：

[Key,DatabaseGenerated(DatabaseGeneratedOption.Identity)]  
public Guid Identifier { get; set; }

当键字段为整数时，Code First默认选择DatabaseGeneratedOption.Identity。而对Guid，你需要显示进行配置。这是唯一一种可以通过Identify来配置Code First的数据类型。如果映射到一现有的数据库，任何在插入数据可生成值的列都可以标识为Identify.

再次运行程序，如图3-4，输出了新生成的标识。



你可能对查看SQL语句感兴趣，代码3-5显示的EF框架发送给数据库的INSERT语句，其中要求数据库为Idenfifier属性生成Guid值

Example 3-5. SQL for inserting a new Trip

[复制代码](javascript:void(0);)

declare @generated\_keys table([Identifier] uniqueidentifier)  
insert [dbo].[Trips]([StartDate], [EndDate], [CostUSD])  
output inserted.[Identifier] into @generated\_keys  
values (@0, @1, @2)  
select t.[Identifier]  
from @generated\_keys as g  
join [dbo].[Trips] as t on g.[Identifier] = t.[Identifier]  
where **@@ROWCOUNT** > **0**',  
N'@0 datetime2(**7**),@1 datetime2(**7**),@2 decimal(**18**,**2**)',  
@0='**2011**-**09**-**01** **00**:**00**:**00**',@1='**2011**-**09**-**14** **00**:**00**:**00**',@2=800.00

[复制代码](javascript:void(0);)

DatabaseGeneratedOption还有两个枚举值:None和Coumputed。下面就有一个示例证明None是有用的。代码3-6显示了另一个新类，Person，SocialSecurityNumber属性已经被配置为此类的键属性。

Example 3-6. Person class with unconventional key property

[复制代码](javascript:void(0);)

using System.ComponentModel.DataAnnotations;  
namespace Model  
{  
   public class Person  
  {  
    [Key]  
    public int SocialSecurityNumber { get; set; }  
    public string FirstName { get; set; }  
    public string LastName { get; set; }  
  }  
}

[复制代码](javascript:void(0);)

记得要在BreakAwayContext类中添加DbSet<Person>

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

最后，将一个新方法，InsertPerson（见代码3-7）添加到控制台程序中，在Main方法中调用这个方法，就会向数据库中添加一个新的person。

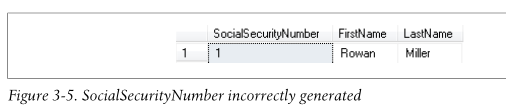
Example 3-7. InsertPerson method

[复制代码](javascript:void(0);)

private static void InsertPerson()  
{  
   var person = new Person  
  {  
     FirstName = "Rowan",  
     LastName = "Miller",  
     SocialSecurityNumber = **12345678**  
  };  
   using (var context = new BreakAwayContext())  
  {  
     context.People.Add(person);  
     context.SaveChanges();  
  }  
}

[复制代码](javascript:void(0);)

再次运行程序，让我们再看看数据库新添加的一行，如图3-5.

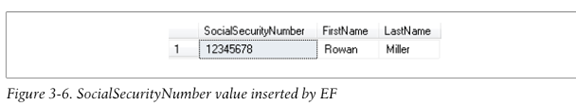


SocialSecurityNumber 的值是 1, 不是12345678.为什么？由于Code First根据key是一个整型这个事实告知数据库这是一个标识字段，因此在INSERT语句中EF框架没有提供正确的SocialSecurityNumber 的值，而是让数据库自行生成。从而在SaveChanges完成后查看 person实例中SocialSecurityNumber 的值，此值已经被更新为数据库生成的值，1.

为修正这一点，我们需要添加一些配置覆写默认标识规则，在这种情况下，DatabaseGeneratedOption.Identity是不对的，应该用None:

[Key, DatabaseGenerated(DatabaseGeneratedOption.None)]  
public int SocialSecurityNumber { get; set; }

 然后再运行程序，如图3-6，数据库正确插入了有关数据。



DatabaseGeneratedOption.Computed用于指定一个映射到数据库的字段是通过计算得到的。例如，如果有一个FullName字段在People表中，是用一个公式将FirstName和LastName组合起来得到的，你就应该让EF框架知道以便其不会尝试存储数据到此列中。你不能指定一个公式用来计算Code First中列的值，因此当映射到一个现存的数据库中你只能使用Computed。要不然，在试图创建数据库时如果遇到Computed配置，数据库引擎就会抛出运行时异常。

## 使用Fluent API来配置数据库生成选项

DatabaseGeneratedOption可以配置为一种特殊的属性，你可以将配置附加HasKey后面，如：

modelBuilder.Entity<Trip>()  
.HasKey(t => t.Identifier)  
.Property(t => t.Identifier)  
.HasDatabaseGeneratedOption(DatabaseGeneratedOption.Identity);

或者创建一个独立的语句：

[复制代码](javascript:void(0);)

modelBuilder.Entity<Person>()  
.HasKey(p => t.SocialSecurityNumber);  
modelBuilder.Entity<Person>()  
.Property(p => p.SocialSecurityNumber)  
.HasDatabaseGeneratedOption(DatabaseGeneratedOption.None);

[复制代码](javascript:void(0);)

你会注意到DatabaseGeneratedOption枚举位于System.ComponentModel.DataAnnotations名称空间，在EntityFramework.dll中。需要在context类的文件头部添加using引用。

## 为开放式并发环境配置时间戳或行版本字段

|  |  |
| --- | --- |
| Convention  默认规则 | None  无 |
| Data Annotation | TimeStamp |
| Fluent | Entity<T>.Property(t=>t.PropertyName).IsRowVersion() |

EF框架从第一版本开始就支持开放式并发环境。[Programming Entity Framework](http://shop.oreilly.com/product/9780596807252.do)这本书的第二版在第23章深入探讨了开放式并发。在这里我我们教你如何配置类映射到RowVersion（或称作TimeStamp，时间戳）字段，同时通知EF框架在进行更新或删除数据库操作时使用这些字段进行并发检查。

使用Code First你需要指定一个字段使用开放式并发检查，与映射到数据库的类型无关，或者你可以进一步指定并发的字段映射到一个TimeStamp字段。

一个类只能有一个属性可以配置为TimeStamp特性。

RRowVersion和TimeStamp是两个具有相同类型的项。Sql Server使用TimeStamp,而其他数据库使用更恰当的名称为RowVersion.d SQL Server2008中，timestamp数据类型也调整为rowversion，但是大多数工具（如Sql Server Management Studio,vs等）仍然显示为timestamp.

## Code First的默认规则与TimeStamp字段

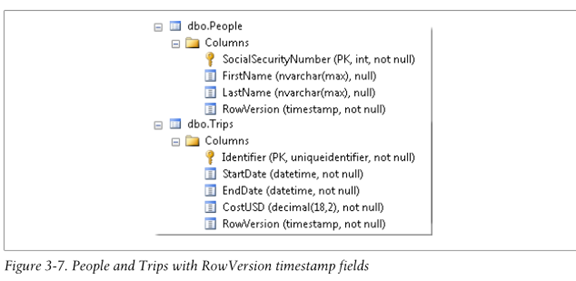
默认情况下，Code First并不识别时间戳属性，因此没有默认约定行为，获得此行为必须配置此属性。

## 使用Data Annotations配置时间戳

并非任何属性都可以映射到一个timestamp数据库类型。必须是byte数组才可以。配置过程很简单，将TimeStamp特性加到Trip和Personal类中的下列属性中。

[Timestamp]  
public byte[] RowVersion { get; set; }

 然后运行控制台程序，确保InserTrip和InsertPerson方法都在Main方法中进行调用。在数据库中你会看到新生成的RowVersion列（图3-7），类型为非可空timestamp类型。



任何时候行内数据被修改时数据库都会自动为此属性创建新值。但TimeStamp不仅影响数据库的映射，还会导致属性被EF框架视作并发的令牌。如果你使用EDMX文件，这就等同于设置了一个属性的ConcurrencyMode(并发模式)。EF框架在执行插入、更新或删除数据库时，就会考虑并发字段，返回每个INSERT和UPDATE更新数据库的值，并传回到每个UPDATE和DELETE的相关属性的原始位置。

例3-8显示了当执行InsertPerson方法后保存设置时的SQL语句：

Example 3-8. INSERT combined with SELECT to return new RowVersion

[复制代码](javascript:void(0);)

exec sp\_executesql N  
'insert [dbo].[People]([SocialSecurityNumber], [FirstName], [LastName])  
values (@0, @1, @2)  
select [RowVersion]  
from [dbo].[People]  
where @@ROWCOUNT > 0 and [SocialSecurityNumber] = @0',

[复制代码](javascript:void(0);)

N'@0 int,@1 nvarchar(max) ,@2 nvarchar(max) ',@0=**12345678**,@1=N'Rowan',@2=N'Miller'

EF框架不仅通知数据库执行**插入，**而且还请求返回RowVersion的值。一旦属性被标记为并发，EF就总会这样做，即使它并不是一个timestamp类型数据。对**更新和删除语句**更是如此，因为在这会有并发检查产生。我们添加一个新的方法，UpdatePerson到程序中，见代码3-9

Example 3-9. The UpdateTrip method

[复制代码](javascript:void(0);)

private static void UpdateTrip()  
{  
   using (var context = new BreakAwayContext())  
  {  
    var trip = context.Trips.FirstOrDefault();  
    trip.CostUSD = 750;  
    context.SaveChanges();  
  }

[复制代码](javascript:void(0);)

}

代码3-10显示了当调用UpdatePerson时的SQL语句：

Example 3-10. UPDATE that filters on original RowVersion and returns new RowVersion

[复制代码](javascript:void(0);)

exec sp\_executesql N'update [dbo].[Trips]  
set [CostUSD] = @0  
where (([Identifier] = @1) and ([RowVersion] = @2))  
select [RowVersion]  
from [dbo].[Trips]  
where @@ROWCOUNT > 0 and [Identifier] = @1',  
N'@0 decimal(18,2),@1 uniqueidentifier,@2 binary(8)',  
@0=**750.00**,@1='D1086EFE-5C5B-405D-9F09-688981BB5B41',@2=**0x0000000000001773**

[复制代码](javascript:void(0);)

注意谓词Where用于定位trip的语句被更新—过滤器包括了Identifier和Rowversion两个参数。如果另外的人更改了行程就会被我们的方法检索到，由于RowVersion已经更改，将不会再有行匹配过滤器。更新就会失败，EF框架会抛出OptimisticConcurrencyException的异常。

## 使用Fluent API配置TimeStamp/RowVersion

Fluent 使用RowVersion来配置，要指定一个RowVersion属性，需要将IsRowVersion()方法附加到属性上。

使用DbModelBuilder,需要对属性作如下配置：

[复制代码](javascript:void(0);)

modelBuilder.Entity<Person>()  
.Property(p => p.RowVersion).IsRowVersion();  
 在EnityTypeConfiguration<T>类中配置如下：

[复制代码](javascript:void(0);)

Property(p=>p.RowVersion).IsRowVersion();

## 配置并发非时间戳字段

|  |  |
| --- | --- |
| Convention  默认规则 | None  无 |
| Data Annotation | ConcurrencyCheck |
| Fluent | Entity<T>.Property(t=>t.PropertyName).IsConcurrencyToken() |

一个不太常见的方式是并发检查是通过字段为非行版本类型进行的。例如，许多数据库可能并没有行版本数据类型。因此你不能指定一个行版本属性，但你仍需要对一个或多个数据库字段进行并发检查。

Person类当前使用属性SocialSecurityNumber作为其标识键。设想类使用了PersionId属性作为标识键而将SocialSecurityNumber简单地视作整型数据而不作为标识跟踪。在这种情况下，你可能想有一种方法避免在SocialSecurityNum ber进行改变时的冲突，因为在美国，每个公民的社会保险号码是唯一的。因此，如果一个一个用户编辑了一个人的记录，可能更改的FirstName的拼写，但同时，另外的人想更改此人的社会保险号码，前者在尝试存储更改时就会遇到一个冲突。指定SocialSecurityNumber属性为一个并发检查字段将提供这种检查（避免这种事情发生）。

## 使用Data Annotations配置开放式并发

代码3-11显示了修改的类为SocialSecurityNumber配置并发检查

Example 3-11. Modified Person class with a ConcurrencyCheck

[复制代码](javascript:void(0);)

public class Person  
 {  
   public int PersonId { get; set; }  
   [ConcurrencyCheck]  
   public int SocialSecurityNumber { get; set; }  
   public string FirstName { get; set; }  
   public string LastName { get; set; }  
}

[复制代码](javascript:void(0);)

例3-12显示了一个方法试图更新一个Person.如果调用这个方法，就需要先调用InsertPerson以确保数据库内存在一个Person数据。

Example 3-12. The UpdatePerson method

[复制代码](javascript:void(0);)

private static void UpdatePerson()  
{  
   using (var context = new BreakAwayContext())  
  {  
    var person = context.People.FirstOrDefault();  
    person.FirstName = "Rowena";  
    context.SaveChanges();  
  }  
}

[复制代码](javascript:void(0);)

正如您在Trip.RowVersion字段中看到的（代码3-10），当一个更新或删除请求发送以数据库时，SLQ语句（见代码3-13）不仅查找匹配的Key（PersonId），还要匹配原始并发字段值（SocialSecurityNumber）.

Example 3-13. SQL providing concurrency checking on SocialSecurityNumber

exec sp\_executesql N'update [dbo].[People]  
set [FirstName] = @0  
where (([PersonId] = @1) and ([SocialSecurityNumber] = @2))  
',N'@0 nvarchar(max) ,@1 int,@2 int',@0=N'Rowena',@1=**1**,@2=**12345678**

如果匹配没有发现（也就是说SocialSecurityNumber已经在数据库中变更了），更新失败抛出OptimisticConcurrencyException异常。

## 使用Fluent API的开放式并发配置

Fluent API使用IsConcurrencyToken方法配置并发，并应用于属性。如代码3-14所示

Example 3-14. Configuring concurrency checking fluently

[复制代码](javascript:void(0);)

public class PersonConfiguration : EntityTypeConfiguration<Person>  
{  
     public PersonConfiguration()  
  {  
     Property(p => p.SocialSecurityNumber).IsConcurrencyToken();  
  }  
}

[复制代码](javascript:void(0);)

我们为Person提供其自己的配置类，也就是这个新类。不要忘记在OnModelCreating方法中将PersonConfiguration添加到modelBuilder.Configurations集合里。

### 映射到非-Unicode数据库类型

|  |  |
| --- | --- |
| Convention  默认规则 | All strings map to Unicode-encoded database types  所有的字符串都映射到Unicode数据库类型 |
| Data Annotation | 不可用 |
| Fluent | Entity<T>.Property(t=>t.PropertyName).IsUnicode(boolean) |

默认情况下，Code First会将所有字符串都映射到数据库中的Unicode字符串类型。

你可以使用IsUnicod方法指定一个字符串是否映射到数据库Unicode字符串类型。下列代码添加到LodgingConfiguation中告诉Code First不要将Owner属性作为Unicode 类型：

Property(l=>l.Owner).IsUnicode(false);

## 对Decimal固定有效位数和小数位数的影响

|  |  |
| --- | --- |
| Convention  默认规则 | Decimals are 18, 2 |
| Data Annotation | 不可用 |
| Fluent | Entity<T>.Property(t=>t.PropertyName).HasPrecision(n,n) |

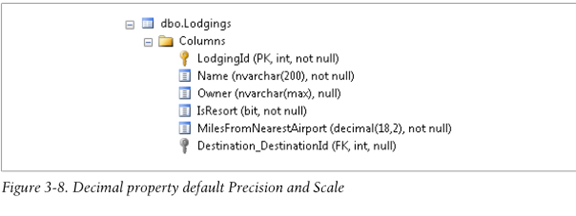
固定有效位数（一个数字中数的位数）和小数位数（小数点右侧的位数）可以使用Fluent API进行配置，而不能用Data Annotations配置。

为了观察其如何工作，我们向Lodging 类中添加一个新的decmial属性：MilesFromNearestAirport:

public decimal MilesFromNearestAirport { get; set; }

## 默认设置

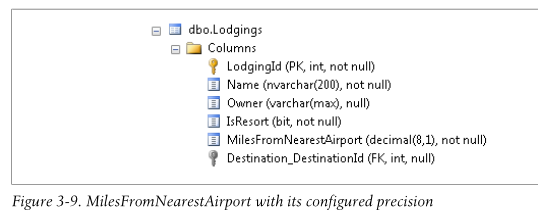
默认情况下，固定有效位数为18，小数位为2，如图3-8所示。



使用Flurent API,可以对固定有效位和小数位进行配置，使用的是HasPrecison方法。即使默认值之一是想要设置的的，也需要将两个值都指定：

Property(l => l.MilesFromNearestAirport).HasPrecision(**8**, **1**);

 图3-9显示了MilesFromNearestAirport有效位和小数位的更改情况。



## 在Code First使用复杂类型

EF框架从第一版开始就支持复杂类型。复杂类型也可视作值类型（？）可以作为附加属性添加到其他类。复杂类型与实体类型的区别在于复杂类型没有其自己的键。它是依赖于其"宿主"类型跟踪变化 和持久化。

一个没有Key属性的类型，并且作为属性映射到一个或多个类型中，Code First就会将其视作为复杂类型。Code First将预设复杂类型的属性出现在宿主类型映射到数据库的表中。

在People表中如何将Person中的Address包含进来，将Address的属性都映射到People表中？可以直接将所有相关属性都纳入Person类中，见代码3-15：

Example 3-15. Individual properties representing an address in Person

[复制代码](javascript:void(0);)

   public class Person  
   {  
    public int PersonId { get; set; }  
    public int SocialSecurityNumber { get; set; }  
    public string FirstName { get; set; }  
    public string LastName { get; set; }  
    public string StreetAddress { get; set; }  
    public string City { get; set; }  
    public string State { get; set; }  
    public string ZipCode { get; set; }  
    }

[复制代码](javascript:void(0);)

但在你的模型中如果使用Address类作为分割类，就可以简化Person类，如代码3-16所示：

Example 3-16. Address type as a property of Person

[复制代码](javascript:void(0);)

public class Address  
{  
public int AddressId { get; set; }  
public string StreetAddress { get; set; }  
public string City { get; set; }  
public string State { get; set; }  
public string ZipCode { get; set; }  
}  
public class Person  
{  
public int PersonId { get; set; }  
public int SocialSecurityNumber { get; set; }  
public string FirstName { get; set; }  
public string LastName { get; set; }  
public Address Address { get; set; }

[复制代码](javascript:void(0);)

}

但是如果这样分割，使用默认规则，会产生一个单独的表：Addresses。而我们目标是让People表中拥有一系列地址字段。如果Adress是一个复杂类型就可以达到这个目的。如果你有其他表中也包含相同的属性，你也可以在那些类中使用Address复杂类型。

## 定义默认复杂类型

最方便的方法将Address转化为复杂类型是移除AddressId 属性。现在注释掉它：

// public int AddressId { get; set; }

在重新运行程序之前，你需要考虑InsertPerson方法（代码3-7）之前 Address是否存在。因为Address属性没有处理将会成为null值，将会造成SaveChanges抛出DbUpdateException异常。现在可以向代码中插入一个新的Person,并且在Person类中实例化一个新的Address。

Example 3-17. Instantiating the Address property in the constructor of the Person class

[复制代码](javascript:void(0);)

public class Person  
{  
public Person()  
{  
Address = new Address();  
}  
//…  
}

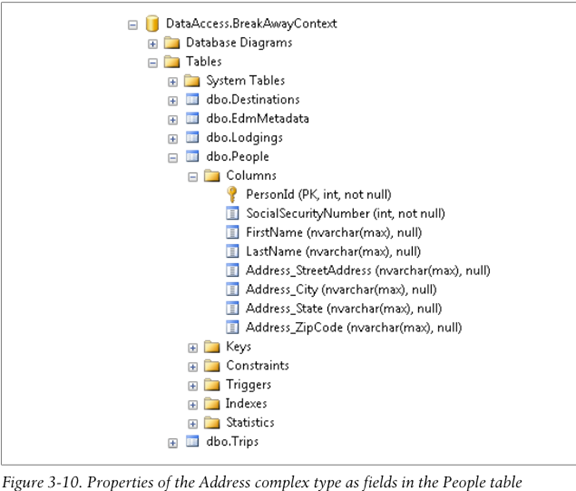
[复制代码](javascript:void(0);)

除了复杂类型不能有Key以外，Code First中还有两条规则用于检测复杂类型是否满足要求。复杂类型在应用于其他类时只能包含原始属性，只能被用作为非集合类型。换句话说，如果想要Person类中有一个List<Address>或其他Address类型的集合类型属性，Address不能作为复杂类型。

复杂类型的默认规则

1. 复杂类型无Key属性
2. 复杂类型只包含原始属性
3. 用作其他类的属性时，属性必须是一个单一实例，不能用于集合类型

运行程序后，图3-10显示了Address字段成为了People表的一部分。Code First认定Address是一个复杂类型，在新模型生成得到响应：



注意Address 字段的命名：HostPropertyName\_Property。这是Code First 的默认设置。第5章，你就会学到如何为复杂属性配置列名。

### 配置非默认复杂类型

如果要使用复杂类型，必须要遵循这些规则吗？可能您想要有一个AddressId属性，尽管你知道一个单独的地址实例不会变更，也不需要EF框架对其跟踪。

如果我们添加AddressId属性重新运行程序，Code First不能推断出你的意图，然后会创建一个单独的Addersses表，并建立与People表的主外键关系。你可以显示地配置复杂类型来修正。

## 使用Data Annotations指定复杂类型

Data Annotation提供了ComplexType特性应用于类上。

Example 3-18. Address with AddressId reinstated and a ComplexType configuration

[复制代码](javascript:void(0);)

[ComplexType]  
public class Address  
{  
public int AddressId { get; set; }  
public string StreetAddress { get; set; }  
public string City { get; set; }  
public string State { get; set; }  
public string ZipCode { get; set; }  
}

[复制代码](javascript:void(0);)

使用这种方式，再次运行程序，模型将重建，最终的数据库架构再一次同图3-10一致，另外附加了一个新的int字段，命名为Address\_AddressId.

## 使用Fluent API指定复杂类型

为了通过Fluent API向Code First指明一个类型为复杂类型，你必须使用DbModelBuilder.ComplexType 方法。

modelBuilder.ComplexType<Address>();

代码3-19显示了对OnModelCreating方法的修改：

Example 3-19. Specifying a complex type fluently

[复制代码](javascript:void(0);)

protected override void OnModelCreating(DbModelBuilder modelBuilder)  
{  
modelBuilder.Configurations.Add(new DestinationConfiguration());  
modelBuilder.Configurations.Add(new LodgingConfiguration());  
modelBuilder.Configurations.Add(new PersonConfiguration());  
modelBuilder.Configurations.Add(new TripConfiguration());  
modelBuilder.ComplexType<Address>();  
}

[复制代码](javascript:void(0);)

本modelBuilder的配置故意将新增加的配置代码放在后面。那些直接通过内建在OmodelCreating方法内部建立的modelBuilder类的实例，称为内联配置，必须将这些代码写在添加的配置类集合的后面。

## 处理更多的杂乱的复杂类型

回想默认配置的复杂类型规定类型只能包含原始类型。如果你的复杂类型不符合这一规范，必须进行配置，这里有一些例子。

我们创建了两个新类，PersonalInfo和Measurement,见代码3-20. PersonalInfo包含有两个Measurement属性。注意到在两个类都没有标识属性。我们的意图是两个类都成为复杂类型。PersonalInfo复杂类型使用Measurment复杂类型，这就是所谓的嵌套复杂类型。

Example 3-20. New classes: PersonalInfo and Measurement

[复制代码](javascript:void(0);)

public class PersonalInfo  
{  
public Measurement Weight { get; set; }  
public Measurement Height { get; set; }  
public string DietryRestrictions { get; set; }  
}  
public class Measurement  
{  
public decimal Reading { get; set; }  
public string Units { get; set; }  
}

[复制代码](javascript:void(0);)

当我们向Person类中添加新的PersonInfo属性后：

public PersonalInfo Info { get; set; }

 也需要添加一些逻辑到Person的构造器，用具体实例说明这些属性：

[复制代码](javascript:void(0);)

public Person()  
{  
Address = new Address();  
   Info = new PersonalInfo  
  {  
    Weight = new Measurement(),  
    Height = new Measurement()  
  };

}

[复制代码](javascript:void(0);)

如果此时继续运行程序，Model builder会抛出异常：

实体类"PersonInfo"没有定义键。请为此实体类定义键。

Code First 并没有将PersonalInfo识虽为复杂类型。原因是我们打破了规则：复杂类型必须只包含原生类型。在PersonalInfo类中有两个Measurement类型的属性。由于这是非原生类型，规则不能将PersonalInfo作为复杂类型。

如果添加ComplexType配置到PersonalInfo类，Code First就能够将属性建立到模型中。你不必配置Measurement类，因为它遵循复杂类的规则。

## 配置复杂类型的属性

Code First将复杂类型属性与其他类型以相同的方式处理，你可以用Data Annotations或Fluently来配置。

## 使用Data Annotations来配置复杂类型

与Code First默认命名列类似：ComplexTypeName\_PropertyName(见图3-10)。你可以应用Data Annotations在复杂类型上，正如你在其他类上使用的那样。例3-21使用了一个你熟悉的特性标记，MaxLength，去影响 Address类型的中的属性。

Example 3-21. Configuring the StreetAddress property of the Address

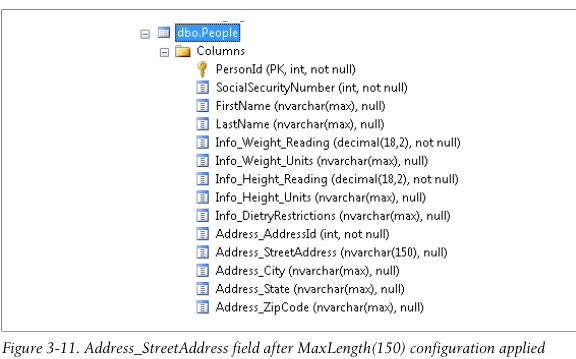
[复制代码](javascript:void(0);)

[ComplexType]  
public class Address  
{  
public int AddressId { get; set; }  
[MaxLength(150)]  
public string StreetAddress { get; set; }  
public string City { get; set; }  
public string State { get; set; }  
public string ZipCode { get; set; }  
}

[复制代码](javascript:void(0);)

Figure 3-11 shows the People table of the database with the modified Address\_StreetAddress field. You can also see the Address\_AddressId field that came from reinstating AddressId and the fields added as a result of the PersonalInfo complex type and its Measurement subtype.

图3-11显示了数据库的People表,其Address\_streetAddress字段已经被修改.你也可以看到Address\_AddressId字段来自于AddressId,作为结果,新添加的了PersonalInfo复杂类型和它的Measurement子类型.



第5章,我们重新检视复杂类型的列名,然后你会学习如何通过配置调整列名.

## 使用Fluent API来配置复杂类型

使用Fluent API有两种方法配置复杂类型属性.你可以开始于宿主实体也可以开始于复杂类型本身.根据EF构架团队的报告,后者是更好的配置方式.MaxLength属于这一类.当我们在第5章讨论列名时,我们会从Person实体中进行配置,以期对Person映射到字段的名字产生影响.

Model builder能够识别复杂类和实体类的差异.

如果你想直接从DbModelBuilder配置,必须开始于Compex<T>方法,而不是一直在用的Entity<T>方法.代码3-22展示了直接在OnCreatingModel中的ModelBuilder实例配置复杂类型.

Example 3-22. Configuring a property of the Address complex type

modelBuilder.ComplexType<Address>()

.Property(p => p.StreetAddress).HasMaxLength(150);

如果你更喜欢封装配置,你需要继承自ComplexTypeConfiguation类而不是EntityTypeConfigration,如代码3-23.

Example 3-23. Configuring the length of StreetAddress in the Address ComplexType

[复制代码](javascript:void(0);)

public class AddressConfiguration :  
ComplexTypeConfiguration<Address>  
{  
public AddressConfiguration()  
{  
Property(a => a.StreetAddress).HasMaxLength(150);  
}  
}

[复制代码](javascript:void(0);)

你还要确保在模型中添加AddressConfiguration:

modelBuilder.Configurations.Add(new AddressConfiguration());

## 小结

I在本章中你已经看到很多Code First创建的预设模型,这些模型都是基于你自己的类创建的.Strings在SQL Server中变成nvarchar(max).数字的固定位数设置为18位(可以确保你可以跟踪1024的数据)和2位小数.这些及其他默认值在很广泛的场合中很有关,你也可以根据需要通过应用配置指定位数.你已经学到如何确保EF框架知道如何将值设定为timestamps或至少是一个并发字段.你已经开始接触到复杂类型,这些类型没有键,只有它们是其他类的属性时才能用到.

Code First的约定在大量通用场景都发挥了很好的作用,但通过你的配置来覆写这些约定还可以控制如何由EF框架来管理你的类.

# 第四章：对关系使用默认规则与配置

在第3章，你已经掌握了默认规则与配置对属性以及其在数据库映射的字段的影响。在本章，我们把焦点放在类之间的关系上面。这包括类在内存如何关联，还有数据库中的外键维持等。你将了解控制多重性关系，无论是否是必须的，还将学习级联删除操作。你会看到默认行为以及如何使用Data Annnotations和Fluent API来控制关系。

你会看到很多只能使用Fluent API而不能使用Data Annotations的情况。上一章我们介绍过"映射到非Unicode数据库类型"就只能在Fluent API中找到。在前几章你已经看到了几个有关默认关系的例子，如代码4-1，就是通过建立类型为List<Lodging>的Lodging属性与炻Destination建立了联系。

Example 4-1. The Destination class with a property that points to the Lodging class

public class Destination

{

public int DestinationId { get; set; }

public string Name { get; set; }

public string Country { get; set; }

public string Description { get; set; }

public byte[] Photo { get; set; }

public List<Lodging> Lodgings { get; set; }

}

在Lodging类（代码4-2）也有一个Destination属性代表单个Destination实例。

Example 4-2. The Lodging class with its reference back to the Destination class

public class Lodging

{

public int LodgingId { get; set; }

public string Name { get; set; }

public string Owner { get; set; }

public bool IsResort { get; set; }

public decimal MilesFromNearestAirport { get; set; }

public Destination Destination { get; set; }

}

Code First观察到您既定义了一个引用又有一个集合导航属性，因此引用默认规则将其配置为一对多关系。基于此，Code First可以确定Lodging（外键）与Destination（主键）具有依赖关系。因此获表Lodging需要要一个外键映射到Destination的主键。在第2章你已经看到，在Lodgings表中确实建立了Destination\_DestinationId外键字段。

本章将全面解析Code First在处理关系的默认规则以及如何按我们的意图覆写这些规则。

应用程序逻辑中的关系

一旦Code First已经创建了模型与关系，EF框架就会将这些关系视为与使用EDMX文件映射的POCO是类似的。所有你在使用POCO对象对EF框架编程的方法和规则仍然适用。例如，如何有一个外键属性和一个导航属性关系，EF框架就会保持他们的同步。如果存在双向关系，EF框架也同样会保持他们的同步。EF框架在什么点上同步值取决于您是否在利用动态代理。没有代理，EF框架将会隐式或显示调用DetectChanges。使用代理，同步的响应发生在属性值变更的时候。事实上你不需要关心是否调用DetectChanges因为DbConext将会在你调用任何依赖同步的方法自动调用。EF框架开发团队建议你如果需要只用动态代理；通常这都是围绕着性能调优进行的。没有代理的POCO类通常使交互关系理简化，因为你没必要知道代理相关的附加行为。

## 多重性关系

如前所述，Code First在看到导航属性和可选的外键属性时将创建关系。有关导航属性和外键属性的细节将帮助我们来确定多重关系的每一端。本章对外键将关注更多一点；现在我们来看看在类中没有外键属性定义的情况。

Code First在处理多重性关系时应用了一系列规则。规则使用导航属性确定多重性关系。即可以是一对导航属性互相指定（双向关系），也可以是单个导航属性（单向关系）。

•如果你的类中包含一个引用和一个集合导航属性，Code First视为一对多关系；

• 如果你的类中仅在单边包含导航属性（即要么是集合要么是引用，只有一种），Code First也将其视为一对多关系；

• 如果你的类包含两个集合属性，Code First默认会使用多对多关系；

•如果你的类包含两个引用属性，Code First会视为一对一关系；

•在一对一关系中，你需要提供附加信息以使Code First获知何为主何为辅。本章后面会在"一对一关系"中提到。如果没有在类中定义外键属性，Code First将设定关系为可选（即一端的关系实际是零对一或恰好相反，零指的是可空---译者注）。

•在本章的"外键"小节，你会看到当在类中定义外键属性，Code First会使用属性的可空性来确定关系是必须的还是可选的。

回顾我们刚刚重温的Lodging与destination的关系,你会看到上述规则。由于有集合和引用属性，Code First就将其视为一对多关系。同时也看到，通过默认规则，Code First将经将其配置为可选关系（optional）。但是在我们的场景里，确实没有想让一个Lodging(住所)不从属于一个Destination(目的地)。因此我们来看看如何确保这种关系是必须的。

## 使用Data Annotations配置多重关系

大多数多重关系配置都需要使用Fluent API。但是我可以使用Data Annotations来指定一些关系是必须的。只需要简单地将Required标记放在你需要定义为必须项的引用属性上。修改Lodging类的代码将Required特性标记放在Destination属性上（代码4-3）：

Example 4-3. Required annotation added to Destination property

public class Lodging

{

public int LodgingId { get; set; }

public string Name { get; set; }

public string Owner { get; set; }

public bool IsResort { get; set; }

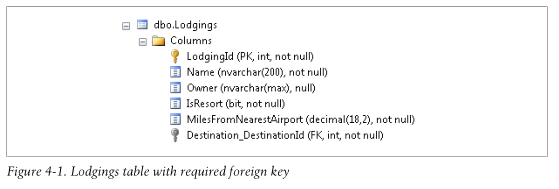
public decimal MilesFromNearestAirport { get; set; }

[Required]

public Destination Destination { get; set; }

}

运行程序，数据库重新创建，你会看到Lodgings表中Destination\_DestinationId不再允许空值（图4-1）。这是因为关系现在是必须的（Required）。



使用Fluent API配置多重性关系

如果没有花时间去理解基本原理，使用Fluent API配置关系会让人感到迷惑。

当使用Data Annotations修改关系时，可以将特性直接放在了导航属性上。这与Fluent API不同，Fluent API并不直接在属性上配置关系。为了达到相同的目的，必须先确定关系。有时在一端就足够，但更多的需要对全部关系进行描述。

为了确定关系，你必须指明导航属性。不管从哪端开始，都要使用这样的代码模板：

Entity.Has[Multiplicity](Property).With[Multiplicity](Property)

多重性关系可以是Optional（一个属性可拥有一个单个实例或没有），Required（一个属性必须拥有一个单个实例）或很多的（一个属性可以拥有一个集合或一个单个实例）。

Has方法包括如下几个：

• HasOptional

• HasRequired

• HasMany

在多数情况还需要在Has方法后面跟随如下With方法之一：

• WithOptional

• WithRequired

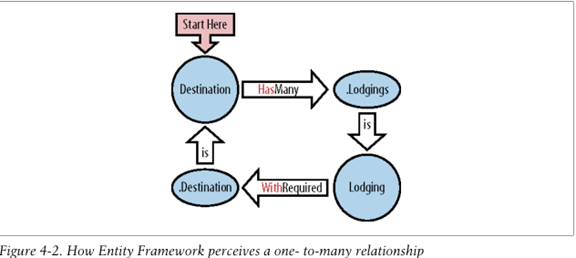
• WithMany

代码4-4显示了一个 使用现有的Destination和Lodging之间的一对多关系的实例。这一配置并非真的做任何事，因为这会被Code First通过默认规则同样进行配置。本章后面会看到识别这种关系然后作进一步的配置，实现外键关系和级联删除功能。

Example 4-4. Specifying an optional one-to-many relationship

modelBuilder.Entity<Destination>()  
.HasMany(d => d.Lodgings)  
.WithOptional(l => l.Destination);

这一代码确定Destination的Has关系。有很多由Lodgings的属性所定义的关系。Lodgings端到Destination的关系是可选的。图4-2尝试帮你观察这种关系建立的过程。



我们来看看如何使用Fluent API建立Required关系。在DestinationConfiguration添加代码4-6：

Example 4-6. Configuring a required relationship with the Fluent API

HasMany(d => d.Lodgings)  
.WithRequired(l => l.Destination);

这看起来非常类似于代码4-5，只不过调用了HasRequired来取代HasOptional。这会使Code First知晓你想建立一个必须的（Required）一对多关系。运行程序你会看到数据库与图4-1显示的一样，与使用Data Annotations的Required标记产生效果一致。如果你想在两端配置全必须的一对一或全可选的一对一关系，Code First会需要更多的信息来获知何为主何为辅。这种Fluent API代码会让人很迷惑！好消息是你可能不需要经常这么做。这一议题将在"1-1关系"中详细讲述。

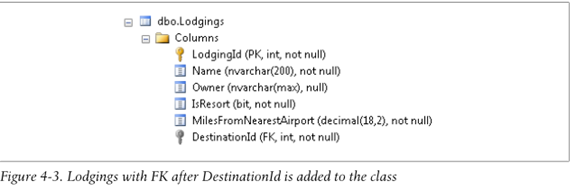
## 使用外键

到目前为止，我们只是看了在类中没有外键属性的关系。例如，Lodging只包含一个引用属性到Destination，但没有属性来存储它指向Destination的键值。在这种情况下，我们已经看到，Code First会为你的数据库引入外键。我们来看看在如果在类本身引入键属性时会发生什么。

在上一节中，通过添加一些配置，建立了Lodging与Destination的Required关系。请删除此配置，以使我们可以观察到Code First的约定行为。配置删除后，添加到一个DestinationId属性到Lodging类中：

public int DestinationId { get; set; }

一旦你添加外键属性到Lodging类，继续运行您的应用程序。该数据库将回应你刚才的改变重新创建。如果您检查Lodgings表列，你会发现，Code First自动检测DestinationId是一个外键，对应于Lodging与Destination的关系，不再产生Destination\_DestinationId外键（图4-3）。



正如您现在可能期望的，Code First有一个设置或规则得到了应用，结果就是发现了一个关系尝试并找到一个外键属性。规则使用的是属性的名称。按照默认规则，一旦发现外键属性，就被命名为“[目标类型的键名]，[目标类型名称]+[目标类型键名称]”，或“[导航属性名称]+[目标类型键名称]”的形式。前面提到的三个规则中的第一个与您添加的属性DestinationId相匹配。名称匹配是区分大小写的，所以你可以有一个名为DestinationID，DeStInAtIoNiD，或任何其他变化的属性，（将不会被匹配，译者注）。如果没有检测到外键，也没有配置，Code First会自动在数据库中设置一个。

为什么要使用外键属性？

在编写代码时，要找出一个与其他类的关系。例如，您可能会创建一个新的Lodging，要指定Lodging与哪个Destination相关。如果特定的Destination在内存中，你就可以通过导航属性的设置关系：

myLodging.Destination=myDestinationInstance;

但是，如果Destination不在内存中，这将要求你先执行对数据库的查询，检索Destination，让你可以设置该属性。有时，你可能没有在内存中的对象，但你想访问该对象的键值。

带有外键的属性，你可以简单地使用键值而不依赖于内存中的实例：

myLodging.DestinationId=3;

此外，在特定情况下，如果Lodging是新建的，您可以附加到原有的Destination实例上，有些情况下，实体框架还要设定Destination的状态为新增，即使所需的Destination实例已经

存在于数据库中。如果你只与外键进行工作，就能避免这个问题。

还有一些有趣的现象会在添加外键属性时会发生。没有 DestinationId外键属性时，Code First的约定规则允许Lodging.Destination是Optional，这意味着你可以添加没有Destination的Lodging。如果回到第2章中的图2-1，你会看到，在Lodgings表中Destination\_DestinationId字段为可空类型。现在DestinationId属性加入，数据库中的字段不再是可空的，你会发现，你不再可以保存没有Destination，或没有DestinationId属性填充的Lodging数据。这是因为DestinationId是int类型，这是一个值类型，不能分配null值。如果DestinationId类型是Nullable<int>的，这种关系将保持Optional状态。事实上，Code First默认约定就是根据类中外键属性的可空性，来确定是否关系是Required还是Optional的。

Code First允许你定义的类中不使用外键属性建立关系，只是使用外键属性更容易建立关系。然而，由于没有外键属性可以依赖，开发者在与实体框架中的相关数据工作时会遇到一些混乱的行为。如果有外键属性，EF框架会在执行插入时检查关系约束，可以避免插入无效数据。没有外键属性来对一个要求为Required的主实体进行跟踪时（例如，要求一个特定的destination必须对应特定的lodging）时，就需要开发者自己来确保以某种方式提供所需信息给EF（很显然，这样更麻烦---译者注）。您还可以了解更多有关"缺少外键下工作"的信息，见2012年1月号（http://msdn.com/magazine） 。

## 指定非规则命名的外键

如果有一个不遵循规则的外键会怎么样呢？

我们来引入一个新的InternetSpecial类，来跟踪一些各种lodging的特定价格（代码4-7）。这个类即有导航属性（Accommodation），又有外键属性（AccommodationId）,都是为同一关系设立的。

Example 4-7. The new InternetSpecial class

using System;

namespace Model

{

public class InternetSpecial

{

public int InternetSpecialId { get; set; }

public int Nights { get; set; }

public decimal CostUSD { get; set; }

public DateTime FromDate { get; set; }

public DateTime ToDate { get; set; }

public int AccommodationId { get; set; }

public Lodging Accommodation { get; set; }

}

}

在Lodging中需要一个新的属性来包含每个logding的特定报价。

public List<InternetSpecial> InternetSpecials { get; set; }

Code First看到Lodging有很多InternetSpecials，而InternetSpecials又有一个Lodging（称之为Accommodation）.尽管没有设置DbSet<InternetSpecial>, InternetSpecial也可以通过Lodging而包含在模型里。

再次运行程序，将会创建如图4的表。不仅有不是外键的AccommodataionId列，也新增了一个外键列，Accommodation\_LodgingId。



你会看到Code First引入一个外键。Code First根据Accommodation导航属性，检测到了一个对应对Lodging的关系然后使用默认规则创建了Accommodation\_LodgingId字段。默认规则无法将AccommodationId推断为外键，因为Code First检查了默认规则对外键属性名称的三个要求没有在类中找到匹配项，就创建了自己的外键。

### 使用Data Annotations修改外键

你可以使用 Data Annotations 的配置外键特性ForeignKey来声明外键属性。在AccommodtaionId上添加ForeignKey特性告知Code First哪个导航属性是外键，来修复这个问题。

[ForeignKey("Accommodation")]

public int AccommodationId { get; set; }

public Lodging Accommodation { get; set; }

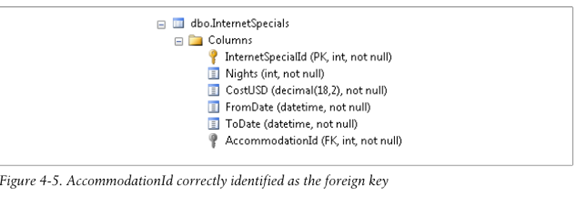
你也可以将ForeignKey特性放在导航属性上来通知哪个属性是关系的外键。

public int AccommodationId { get; set; }

[ForeignKey("AccommodationId")]

public Lodging Accommodation { get; set; }

两种写法都可以。与此同时，你获得的正确的的数据库外键：AccommodtationId,如图4-5所示。



### 使用Fluent API来修改外键

Fluent API并没有提供配置属性作为外键的简单方法。你要使用专门的关系API来配置正确的外键。而且你不能简单地配置关系的片断，你需要首先指定你想配置的关系类型（前面已经提到）然后才能应用修改。

为了指定关系，需要从IneternetSpecial实体开始，我们直接在modelBuilder中进行配置，当然也你可以在EntityTypeConfiguration类中为InternetSpecial创建一个实例。

在这种情况下，我们先要设置关系而不打破Code First建立的默认关系。代码4-8指出了这种关系：

Example 4-8. Identifying the relationship to be configured

modelBuilder.Entity<InternetSpecial>()

.HasRequired(s => s.Accommodation)

.WithMany(l => l.InternetSpecials)

我们想要改变的，是在这种关系下的外键。Code First期待外键属性命名为LodgingId或者是其他的默认名称。因此我们需要告诉AccommodationId 属性才是真正的外键：.代码4-9添加了HasForeignKey方法来为关系指定外键：

Example 4-9. Specifying a foreign key property when it has an unconventional name

modelBuilder.Entity<InternetSpecial>()

.HasRequired(s => s.Accommodation)

.WithMany(l => l.InternetSpecials)

.HasForeignKey(s => s.AccommodationId);

效果与图4-5一致。

## Working with Inverse Navigation Properties

## 使用逆导航属性

Code First到目前为止一直能够解析我们定义的两个导航属性，虽然它们处于不同端，实际上是同一关系。它之所以能做到这一点，因为两端至少有一个可能的匹配关系。例如，Lodging只包含一个单一的属性，指向目的地（Lodging.Destination）;同样地，目的地Destination只包含一个属性引用住所（Destination.Lodgings）。

虽然并不十分普遍，您可能会遇到这样一种情况：实体之间存在多个关系。在这种情况下，Code First将不能够与相关导航属性相匹配。您将需要提供一些额外的配置。

例如，如果你想跟踪每个住所的两个联系人怎么办？这就需要在Lodging类中有一个PromaryContact和一个SecondaryContact属性。我们先将这两个属性添加到类中：

public Person PrimaryContact { get; set; }

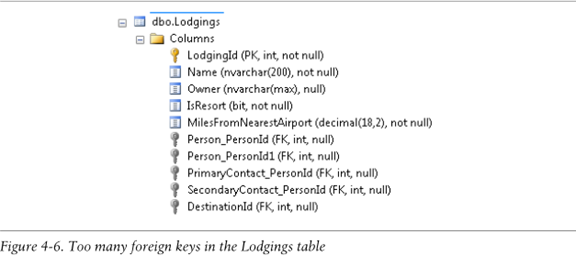
public Person SecondaryContact { get; set; }

在关系的另一端我们也需要引入导航属性。这需要让你从Person类导航到Lodging实例，知道第一联系人和第二联系人连接到哪里去。添加如下两个属性到Person类：

public List<Lodging> PrimaryContactFor { get; set; }

public List<Lodging> SecondaryContactFor { get; set; }

Code First默认约定将对你刚才添加的这些新的关系进行错误的假设。因为有两套导航属性，Code First无法确定他们如何匹配，它会创建单独为每个属性创建关系。图4-6显示的Code First创建的基于您刚才添加的导航属性的四个关系。



Code First默认规则可以识别双向关系，但不能识别在两个实体中多个双向关系。原因如图4-6所示，多个外键，使得Code First无法确定在Lodging中的两个返回Person实体的属性连接到Person类的哪个List<Lodging>属性。

你可以添加配置（使用Data Annotations或Fluent API）到modelBuilder来表明这种关系。使用Data Annotations，你需要使用一个特性标记叫做InverseProperty。使用Fluent API，需要合并使用Has/With方法指定这些关系正确的端点。

你可将特性标记放在关系的任何一端（或两端都放）。我们将其放在Lodging类中（代码4-10）。InverseProperty特性标记需要相关类中相应导航属性作为参数。

Example 4-10. Configuring multiple bidirectional relationships from Lodging to Person

[InverseProperty("PrimaryContactFor")]

public Person PrimaryContact { get; set; }

[InverseProperty("SecondaryContactFor")]

public Person SecondaryContact { get; set; }

使用Fluent API，你需要使用Has/With语句来指定关系的两端。见代码4-11 ，第一个配置的一端为Lodging.PrimaryContact，另一端为Person.Primary ContactFor。第二个配置是针对SecondaryContact和SecondaryContactFor两者关系建立的，方法类似。

Example 4-11. Configuring multiple relationships fluently

modelBuilder.Entity<Lodging>()

.HasOptional(l => l.PrimaryContact)

.WithMany(p => p.PrimaryContactFor);

modelBuilder.Entity< Lodging >()

.HasOptional(l => l.SecondaryContact)

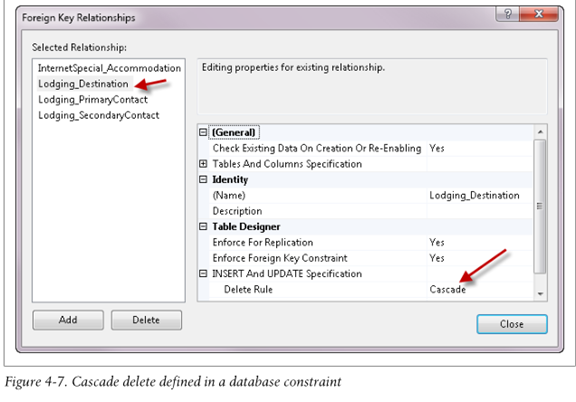
.WithMany(p => p.SecondaryContactFor);

## 使用级连删除

级联删除允许主记录被删除时相关联的依赖性数据也被删除。例如，如果你删除Destinantion，相关的Lodgings会被自动删除。EF框架支持对内存中和数据库中的数据进行级联删除。在"用EF框架编程"第二版第19章，推荐为模型实体配置级联删除，所映射的数据库对象也会具有级联删除的定义。

默认规则约定，Code First会对Required的关系设置级联删除。当一个级联删除定义后，Code First会在数据库中为其创建级联删除。在本章前面我们已经将Lodging和Destination的关系设定为Required 。换句话说，没有Destination，Lodging也不存在。因此，如果删除一个Destination，任何相关联的Lodging（在内存中且被上下文所跟踪）也会被删除。当提交SaveChanges，数据库会删除任何保存在Lodgings表中的相关行，使用的就是级联删除行为。

再看数据库，你会看到Code First实施了级联删除并且在数据库之间的关系上添加了约束。请注意图4-7的删除规则设定了级联。



代码4-12是一个新方法叫做DeleteDestinationInMemoryAndDbCascade，用于展示内存中和数据库中的级联删除。

Example 4-12. A method to explore cascade deletes

private static void DeleteDestinationInMemoryAndDbCascade()

{

int destinationId;

using (var context = new BreakAwayContext())

{

var destination = new Destination

{

Name = "Sample Destination",

Lodgings = new List<Lodging>

{

new Lodging { Name = "Lodging One" },

new Lodging { Name = "Lodging Two" }

}

};

context.Destinations.Add(destination);

context.SaveChanges();

destinationId = destination.DestinationId;

}

using (var context = new BreakAwayContext())

{

var destination = context.Destinations

.Include("Lodgings")

.Single(d => d.DestinationId == destinationId);

var aLodging = destination.Lodgings.FirstOrDefault();

context.Destinations.Remove(destination);

Console.WriteLine("State of one Lodging: {0}",

context.Entry(aLodging).State.ToString());

context.SaveChanges();

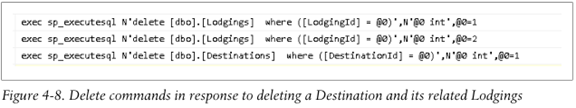
}

}

代码使用context插入了一个新Destination，有两个Lodging.然后将这些Lodging储存进数据库然后记录了新添加的Destination。在一个单独的context里，代码取出Destination和其相关的Lodging，然后使用Remove方法标记Destination实例为删除，我们使用Console.WriteLine来检测相关Lodging实例在内存中状态，这使用了一个DbContext的Entry方法。Entry方法能够让我们访问EF施加给给定对象的状态信息。最后，调用SaveChanges方法持久化删除信息到数据库。

调用Destination的Remove方法，Lodging的状态显示在控制台窗口。尽管我们并没有显示地要求删除任何Lodging，但仍显示出了删除命令。这是因为当我们显示地删除Destination时，EF框架使用客户端的级联删除功能删除了依赖的Lodging。

下一步，当SaveChanges方法调用时，EF框架发送三个DELERE命令到数据库。如图4-8所示，前两删除命令是对相关Lodging实例进行删除，第三个才是删除Destination,



现在我们来改变一下方法。我们将要要随同Destination一起删除以前存入的Loading数据。我们删除与Lodging提到的所有相关代码。由于内存中无Lodging，就不会有客户端的级联删除，而数据库却清除了任何孤立的Lodgings数据，这是因为在数据库中定义了级联删除。（见图4-7）修改的方法见代码4-13.

Example 4-13. Modified DeleteDestinationInMemoryAndDbCascade code

private static void DeleteDestinationInMemoryAndDbCascade()

{

int destinationId;

using (var context = new BreakAwayContext())

{

var destination = new Destination

{

Name = "Sample Destination",

Lodgings = new List<Lodging>

{

new Lodging { Name = "Lodging One" },

new Lodging { Name = "Lodging Two" }

}

};

context.Destinations.Add(destination);

context.SaveChanges();

destinationId = destination.DestinationId;

}

using (var context = new BreakAwayContext())

{

var destination = context.Destinations

.Single(d => d.DestinationId == destinationId);

context.Destinations.Remove(destination);

context.SaveChanges();

}

using (var context = new BreakAwayContext())

{

var lodgings = context.Lodgings

.Where(l => l.DestinationId == destinationId).ToList();

Console.WriteLine("Lodgings: {0}", lodgings.Count);

}

}

运行后，发送到数据库的唯一命令是删除destination。数据库级联删除响应的相关Lodging。当在Lodgings端查询时，由于数据库删除了lodgings,查询不会返回结果，lodgings变量成为一空的列表。

### 使用Fluent API配置打开或关闭客户端级联删除功能

你可能会在现有的数据库上工作，不使用级联删除或者你可能有一个规则必须显示删除数据，不允许在数据库中自动删除。如果从Lodging到Destination之间的关系是Optional，这不是一个问题，因为按照默认规则，Code First不能在可选的关系上使用级联删除。但你可能需要即有Required的关系，又不想使用级联删除功能。

例如需要在应用程序中试图删除一个Destination时向用户返回一个错误，这是在没有显示地删除或重新给这个Destination分配Lodging实例时出现的。在这种情况下，你就需要一个Required关系而不需要级联删除，你可以显示地覆写默认规则通过使用Fluent API来对级联删除进行配置。这个功能Data Annotations不支持。

记住，如果建立这样的模型，应用程序代码可以实现定制地删除数据，或在必要时重新分配相关的数据。

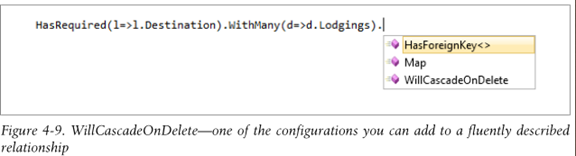
Fluent API使用的方法是WillCascadeOnDelete，以一个布尔值作为参数。此配置适用于所有关系，因此首先需要使用指定一个配对的关系，然后调用WillCascadeOnDelete方法。

在LodgingConfiguration类中，关系定义为：

HasRequired(l=>l.Destination)

.WithMany(d=>d.Lodgings)

在这里，有三个可能的配置可供添加。WillCascadeOnDelete是其中之一，如图4-9所示。



现在你可以设置此关系的WillCascadeOnDelete为false：

HasRequired(l=>l.Destination)

.WithMany(d=>d.Lodgings)

.WillCascadeOnDelete(false)

这就使得Code First生成的数据库架构将不会包含级联删除。图4-7所示的级联删除规则将不会出现。

在关系为Required的场景下，这种逻辑会创建一个冲突，例如，目前在Lodging和Destination的Required关系中，需要一个Lodging实例有一个Destination或一个DestinationId。如果你有一个正在变化的跟踪，并删除了相关的Destination，这将导致Lodging.Destination为空。调用SaveChanges时，实体框架将尝试同步Lodging.DestinationId，设置为NULL。但是，这是不行的，异常将抛出下面的详细信息：

The relationship could not be changed because one or more of the foreign-key properties is non-nullable. When a change is made to a relationship, the related foreign-key property is set to a null value. If the foreign-key does not support null values, a new relationship must be defined, the foreign-key property must be assigned another non-null value, or the unrelated object must be deleted.

关系不能更新，因为一个或多个外键的属性非空。对关系进行更新时，相关的外键的属性设置为空值。如果外键不支持空值，必须定义一个新的关系，外键的属性必须指派另一个非空值，或删除非关联的对象。

这表明，如果已经控制了级联删除设置，就要为避免或解决验证没有级联删除可能引起的冲突负责。

### 对不被数据库所支持的场合关闭级联删除

许可数据库（包括SQL Server）不支持指定级联删除指向到同一个表的多重关系（原文为：Some databases (including SQL Server) don't support multiple relationships that specify cascade delete pointing to the same table.不知如何翻译---译者注 ）。由于Code First配置的Required关系包括级联删除，这就造成如果有两个Required关系指向同一个实体就会出现错误。你可以使用WillCascadeOnDelete(false)来关闭一个或多个联删除设置。代码4-14显示了如果不进行正确配置来自于SQL Server的异常信息。

Example 4-14. Exception message when Code First attempts to create cascade delete where multiple relationships exist

System.InvalidOperationException was unhandled

Message=The database creation succeeded, but the creation of the database objects

did not.

See InnerException for details.

InnerException: System.Data.SqlClient.SqlException

Message=Introducing FOREIGN KEY constraint 'Lodging\_SecondaryContact' on table

'Lodgings' may cause cycles or multiple cascade paths. Specify ON DELETE

NO ACTION or ON UPDATE NO ACTION, or modify other FOREIGN KEY

constraints. Could not create constraint. See previous errors.

### 客户端级联删除对性能的影响

本身就会实施级联删除。如果你将所有相关对象调入内存，让客户端级联删除这些对象，就会导致调用SaveChanges方法时，会发送很多针对这些相关对象的Delete命令到数据库，从而造成暂时的命令拥塞。当然有的情况下，这些相关对象是在内存中的，也希望这些对象能够被删除，这种情况另当别论。但是，如果数据并没载入内存，完全可以依靠数据库做级联删除，而避免将这些对象载入内存。

## 探索多对多关系

EF框架支持多对多关系。让我们来看看Code First是如何在生成数据库时响应类间的多对多关系。

在使用database first策略时如果有多对多关系，EF框架可以创建多对多映射，条件是数据库内联表只包含相关实体的主键。这种映射规则也适用于Code First。

我们添加一个新的类：Acitivity到模型中，如代码4-15，将于Trip类相关联。一个Trip类可以有一些Activites日程，而一个Activity日程又可以计划好几个trips(行程)。因此Trip和Activity就会有多对多关系。

Example 4-15. A new class, Activity

using System.ComponentModel.DataAnnotations;

using System.Collections.Generic;

namespace Model

{

public class Activity

{

public int ActivityId { get; set; }

[Required, MaxLength(50)]

public string Name { get; set; }

public List<Trip> Trips { get; set; }

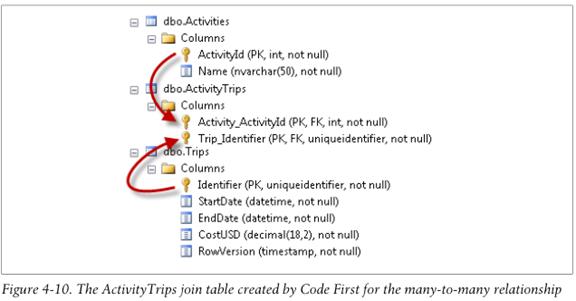
}

}

在Activity类中有一个List<Trip>，我们也添加了一个List<Activity>到Trip类到另一端形成多对多关系。

public List<Activity> Activities { get; set; }

再次运行程序，因为模型变化Code First将重新创建数据库。Code First根据默认规则识别出了多对多关系，建立了内联表，并配置了合适的键。两个内联表的主键都作为外键指向了内联表，如图4-10所示。



注意到Code First的默认规则创建的表名合并使用了两个类的类名。它也使用了我们在前面创建外键使用的模式来创建外键。在第5章，我们将关注于表和列的映射，到时你会学习到如何使用配置为内联表指定表名和列名。

一旦多对对关系建立，其行为就与EF早期版本中多对多关系所表现出来的是一样的。你可以通过类属性查询，添加和删除相关对象。在后台，EF框架将使用它的内置特性来协助数据库创建集成的内联表的select,insert,update和delete命令。

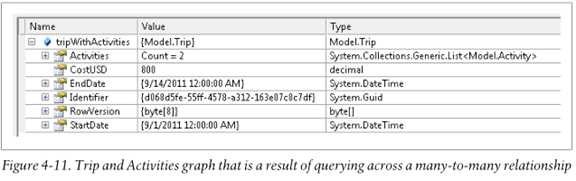
例如，如下的查询寻找一次单独的trip和计划实施的相关Activities.

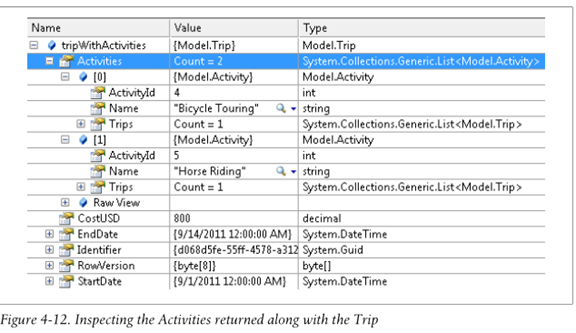
var tripWithActivities = context.Trips

.Include("Activities").FirstOrDefault();

查询是对类进行的，没有必要关心trip和activities是怎样在数据库连接的。EF框架会自行配置SQL语句执行内联，并返所有适合于第一条trip的所有activities记录。虽然不需要自行构建SQL语句，但一定要记住不管你的类的结构或数据库构架如何，EF框架构建的SQL都是可以通用的。

输出的结果是trip和其activities的图。图4-11显示了Trip类在一个调试窗口的信息。你可以看到其包含两个activites，都最从数据库中提取出来匹配这次Trip的。





不必知道它的存在，EF框架会维护内联表并通过来组织表之间的Join。同样地，任何时候你进行插入，更新或删除操作，EF框架将制定出正确的内联SQL语句，不用在你的代码中作任何关注。

## 使用单边导航的关系

到目前为止我们已经观察了导航属性已经定义在两个类中的关系。但是，这并不是EF框架能够工作所必须的。

在你的域中，从Destination导航到其相关的Lodging选项是一种通常的情况，但是可能很少需要从Lodging导航回Destination.让我们将Destination从Lodging类中移走（代码4-16）。

Example 4-16. Navigation property removed from Lodging class

public class Lodging

{

public int LodgingId { get; set; }

public string Name { get; set; }

public string Owner { get; set; }

public bool IsResort { get; set; }

public decimal MilesFromNearestAirport { get; set; }

public int DestinationId { get; set; }

//public Destination Destination { get; set; }

public List<InternetSpecial> InternetSpecials { get; set; }

public Person PrimaryContact { get; set; }

public Person SecondaryContact { get; set; }

}

EF框架可以处理这种情况。这里清晰地定义了从Lodging到Destination之间的关系，依据的是Destination类中的Lodgings属性。这仍然会使用模型构建器到Lodging类中去寻找外键Lodging.DestinationId满足默认规则。

现在我们前进一步，将Lodging类中的外键属性删除，如代码4-17.

Example 4-17. Foreign key commented out

public class Lodging

{

public int LodgingId { get; set; }

public string Name { get; set; }

public string Owner { get; set; }

public bool IsResort { get; set; }

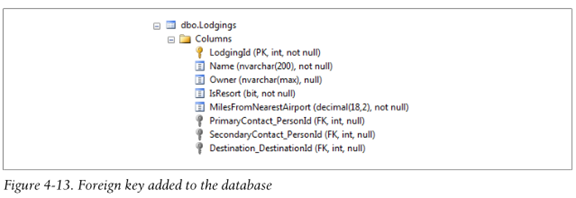
public decimal MilesFromNearestAirport { get; set; }

//public int DestinationId { get; set; }

//public Destination Destination { get; set; }

}

是否还记得如果不定义一个外键在你的类中Code First默认规则会自动引入一个？同样的规则适用于在单边定义的导航属性。Destination仍然有一个属性定义 了到Lodging的关系。图4-13显示了有一个Destination\_DestinationId列加入到的Lodgings表中。这可能会使你回想起有关外键列的命名规则：[Navigation Property Name] + [Primary Key Name]。但是我们在Lodgin类里不再有一个导航属性。如果在依赖实体中没有导航属性加以定义，Code First将会使用[Principal Type Name] + [Primary KeyName].在这种情况下，使用了同一个名字。



那么如果我们试图在另一个类中只定义外键而没有导航属性呢，EF框架本身支持这种情况，但Code First不支持。Code First需要至少一个导航属性来创建关系。如果你移除了两边的导航属性，Code First将只将外键属性作为任何类中的其他属性而不会在数据库中创建约束。

现在我将外键属性调整为默认规则无法检测到的情况。我们用LocationId替代DestinationId,如代码4-16.记住我们没有导航属性，仍然被注释着。

Example 4-18. Foreign key with unconventional name

public class Lodging

{

public int LodgingId { get; set; }

public string Name { get; set; }

public string Owner { get; set; }

public bool IsResort { get; set; }

public decimal MilesFromNearestAirport { get; set; }

public int LocationId { get; set; }

//public Destination Destination { get; set; }

public List<InternetSpecial> InternetSpecials { get; set; }

public Person PrimaryContact { get; set; }

public Person SecondaryContact { get; set; }

}

感谢Destination.Lodgings，Code First知道两个类中存在关系。但它无法找到一个符合约定的外键。我们之前已经铺了路，现在还需要一些配置来帮助Code First识别外键。

在前面的例子里，我们将ForeignKey特性标记放在依赖类的导航属性或者将其放在外键属性上，告知哪个导航属性属于它。但我们在依赖类中不再有一个导航属性。幸运的是，我们可以将Data Annotations的标记放在导航属性上（Destination.Lodgings）。Code First知道Lodging是关系中的依赖类，因此它会为外键在此类中寻找有关字段：

[ForeignKey("LocationId")]

public List<Lodging> Lodgings { get; set; }

Fluent API也能为这种单侧导航属性创建关系。配置的Has部分必须指定一个导航属性，而With部分如果没有反向导航属性就留空。一旦指定了Has和With语句，就可以调用HasForeignKey方法：

modelBuilder.Entity<Destination>()

.HasMany(d => d.Lodgings)

.WithRequired()

.HasForeignKey(l => l.LocationId);

在我们需要创建单边关系时，很多情况下我们想要从Lodging导航回相应的Destination。我们恢复对Lodging类的调整。取消对Destination属性的注释并将外键属性恢复，如代码4-19.你也需要将ForegnKey标记从Destination.Lodging上移除,删除上述刚刚添加的Fluent API配置。

Example 4-19. Lodging class reverted to include navigation property and conventional foreign key

public class Lodging

{

public int LodgingId { get; set; }

public string Name { get; set; }

public string Owner { get; set; }

public bool IsResort { get; set; }

public decimal MilesFromNearestAirport { get; set; }

public int DestinationId { get; set; }

public Destination Destination { get; set; }

public List<InternetSpecial> InternetSpecials { get; set; }

public Person PrimaryContact { get; set; }

public Person SecondaryContact { get; set; }

}

## 使用一对一关系

There is one type of relationship that Code First will always require configuration for: one-to-one relationships. When you define a one-to-one relationship in your model, you use a reference navigation property in each class. If you have a reference and a collection, Code First can infer that the class with the reference is the dependent and should have the foreign key. If you have two collections, Code First knows it's many-to-many and the foreign keys go in a separate join table. However, when Code First just sees two references, it can't work out which class should have the foreign key.

Let's add a new PersonPhoto class to contain a photo and a caption for the people in the Person class. Since the photo will be for a specific person, we'll use PersonId as the key property. And since that is not a conventional key property, it needs to be configured as such with the Key Data Annotation (Example 4-20).

有一种关系Code First必须进行配置后才能工作，这种关系就是一对一关系。当你在模型中定义一对一关系，你需要在每个类中都要使用引用导航。如果你有一个引用和一个集合，Code First就会将引用视为依赖类，推测应该有一个外键。如果有两个集合，Code First视为多对多关系，将外键放在一个单独的内联表中。但是，Code First看到两个引用时，它无法识别哪个类应该有一个外键。

我们添加一个新的PersonPhoto类，包含一个针对属于Person类中的people的photo和caption属性。由于photo将会指定给特定的person，我们使用PersonId作为键属性。并有没有一个默认的键属性，需要如下所示的Data Annotations配置（代码4-20）：

Example 4-20. The PersonPhoto class

using System.ComponentModel.DataAnnotations;

namespace Model

{

public class PersonPhoto

{

[Key]

public int PersonId { get; set; }

public byte[] Photo { get; set; }

public string Caption { get; set; }

public Person PhotoOf { get; set; }

}

}

我们在Person中也添加一个Photo属性，这样可以在两端都可以导航。

public PersonPhoto Photo { get; set; }

记住在这种情况下Code First无法确认哪个类是依赖类。当其尝试构建模型时，就会抛出一个异常，告知你它需要更多信息：

Unable to determine the principal end of an association between the types 'Model.PersonPhoto' and 'Model.Person'. The principal end of this association must be explicitly configured using either the relationship fluent API or data annotations.  
无法确认在类型'Model.PersonPhoto'和'Model.Person'之间联系的主端。这种联系的主端必须使用Fluent API或data annotations进行显示配置；

个问题可以很容易地使用ForeignKey特性标记来解决，将标记放在依赖类上指出其包含外键。当配置一对一关系时，EF框架需要依赖类的主键也应是外键。在我们的案例中，PersonPhoto是依赖类，而其键，PersonPhoto.PersonId，也应是一个外键。我们将ForeignKey标记加在PersonPhoto.PersonId属性上，如代码4-21，记住在加入ForeignKey时要为关系指定导航属性。

Example 4-21. Adding the ForeignKey annotation

public class PersonPhoto

{

[Key]

[ForeignKey("PhotoOf")]

public int PersonId { get; set; }

public byte[] Photo { get; set; }

public string Caption { get; set; }

public Person PhotoOf { get; set; }

}

运行程序会成功创建新数据库标，尽管你会看到EF框架并没有很好地处理单词"Photo",还是将其复数化（第5章你会学习如何为表指定名称），但是PersonId现在即是PK又是FK。如果你观察PersonPhoto\_PhotoOf外键约束细节，还可以看到这里显示People.PersonId在关系中是主表/列，而PersonPhotoes.PersonId是外键表/列（图4-14）：



在本章前面，介绍过可以将ForeignKey标记放在导航属性上，也可以指定外键属性的名称（在本例中，就是PersonId）.由于两个类都包含PersonId属性，Code First仍不能确认哪个类包含外键，因此你不能用这样的方式来为此种场景配置。

当然，我们也可以Fluent API来进行配置。我们假定这时的关系是一对零或一对一，也就是PersonPhoto必须有一个Person对应而一个Person不必一定有一个PersonPhoto对应。我们使用HasRequired和WithOptinal联合使用来指定这种情况：

modelBuilder.Entity<PersonPhoto>()

.HasRequired(p => p.PhotoOf)

.WithOptional(p => p.Photo);

这足以让Code First将PersonPhoto视作依赖类。我们想要将Person类作为主类而PersonPhoto辅助类，因为一个Person可以存在没有PersonPhoto的情况，但是一个PersonPhoto必须有一个Person.

注意你没有必要使用HasForeignKey来指定PersonPhot.PersonId作为外键。这是因为EF框架可以直接将依赖项的主键作为外键使用。由于没有选择，Code First会将这种唯一情况推断出来。事实上，Fluent API也不会让你使用HasForeignKey，在HasRequired和WithOptional方法后的智能感知里该方法根本不可用。

### 当两端都是Required时配置一对一关系

现在我们来告诉Code First一个Person必须有一个PersonPhoto（即也是Required）。使用Data Annotations，可以将Rrequired标记放在任何类型的属性上来实现（不一定非是原生类型）：

[Required]

public PersonPhoto Photo { get; set; }

现在更新Main方法来调用InserPerson方法（见第3章），运行程序。在运行SaveChanges时会抛出异常，EF框架的验证API报告对PersonPhoto的Required要求验证失败。

### Ensuring that the sample code honors the required Photo

### 如何修正代码？

如果你想让Photo属性为Required又要避免验证错误，可以修改InsertPerson和UpdatePerson方法以便可将数据添加到Photo字段中。为了保持代码的简洁，我们只填充一个单一的字节到Photo的byte数组里而不是使用实际的图片。

在InsertPerson方法里，修改代码实例化一个新的Person对象添加Photo属性，如代码4-22：

Example 4-22. Modifying the InsertPerson method to add a Photo to the new Person

var person = new Person

{

FirstName = "Rowan",

LastName = "Miller",

SocialSecurityNumber = 12345678,

Photo = new PersonPhoto { Photo = new Byte[] { 0 } }

};

在UpdatePerson方法中，我们添加了一些代码来保证任何已添加的Person数据都会在更新时同时获得一个Photo。修改UpdatePerson方法见代码4-23：

Example 4-23. Modification to UpdatePerson to ensure existing Person data has a Photo

private static void UpdatePerson()

{

using (var context = new BreakAwayContext())

{

var person = context.People.Include("Photo").FirstOrDefault();

person.FirstName = "Rowena";

if (person.Photo == null)

{

person.Photo = new PersonPhoto { Photo = new Byte[] { 0 } };

}

context.SaveChanges();

}

}

更新方法使用Include方法来获取数据库中Person的图片。然后检查Person对象是否有Photo数据，如果没有就添加一个新的。现在Person类中的Photo的Required要求得到满足，就可以在任何时候成功执行InsertPerson和UpdatePerson方法。

### 使用Fluent API配置一对一关系

毫无疑问，也可以使用Fluent API来配置同样的关系。但首先需要让Code First知道哪个类为主哪个类为辅。如果两端均为Required，不能简单地从多重关系上推测出来。

你可以跟随在WidthRequired后面来调用HasRequired方法 。但是如果你开始于HasRequired，你会在WithReuired的位置有两个附加选择：WithRequiredPrincipal 和WithRequiredDependent。这些方法将你要配置的实体考虑了进去（就是你选择的基于模型构建器或者EntityTypeConfiguration类建立的实体）。选择WithRequiredPrincipal将会使实体配置为主类，意味着该类包含有关系的主键。选择WithRequiredDependent会使实体配置为辅助类，意味着该类包含有关系的外键。

假设你想将PersonPhoto配置为依赖类，应该使用下列配置代码：

modelBuilder.Entity<PersonPhoto>()

.HasRequired(p => p.PhotoOf)

.WithRequiredDependent(p => p.Photo());

配置两端都是Optional的一对一的关系其方法是类似的，除了开始于HasOptional外还应该选择是WithOptionalPrincipal 还是 WithOptionalDependent。

## 小结

在本章，你已经看到Code First在处理关系上很智能。Code First的默认规则能够发现任何多样性的关系，并适时提供外键配置。但也有一些场景可能你并不想完全遵循默认规则，Code First完全支持这种场景。你也学习了如何使用Data Annotations和Fluent API来定制模型。你应该已经很好地理解了如何在在Fluent API中使用基于Has/With的语句来处理关系。

在下一章，我们来看看Code First的另一套映射，就是类如何映射到数据库，包括如何映射到各种继承架构等。

# 第五章：对数据库映射使用默认规则与配置

    到目前为止我们已经领略了Code First的默认规则与配置对属性、类间关系的影响。在这两个领域内，Code First不仅影响模型也影响数据库。在这一章，你将让默认规则与配置的目光聚焦在类映射到数据库上而不影响概念模型。

     我们从简单的映射开始，设法指定数据库的表名，构架与属性。在此你将掌握如何让多个类映射到一个通用表中，或将单个类映射到多个表中。最后，带您漫步各种继承架构的配置。

## 将类名映射到数据库表名和构架名

EF框架使用模型的类名的复数形式来生成数据库表名—Destination变成Destinations，Person变成People等。默认的类命名规则可能与您设定的命名规则不一致，例如需要指定非复数形式的表名（例如PersonPhoto类不要映射到PersonPhotoes表上去），或者你想映射到一些现有的表，其表名恰好与Code First的规则不同。

可以使用Data Annotations 来确保Code First将你的类映射到正确的表名上。使用Table标记，就可改变数据库构架的表名。

数据库表命名对其他映射至关重要，正如你将在本章看到的，包括实体分割，继承层次结构，甚至多对多映射。

默认规则里Code First复数化类名，然后使用复数化后的结果作为类映射的表名。另外，默认情况下所有的表都被安置到dbo构架里。

### 使用Data Annotations 配置表和构架名

     Table标记允许你改变类映射的表名。在上一章，PersonPhoto被不恰当地复数化了，表名为PersonPhotoes。尽管这样仍可以正常工作，但直接使用PersonPhoto也是可行的:

[Table("PersonPhotos")]

public class PersonPhoto

另一个例子是原始的BrakAway数据库,在Programming Entity Framework的第一,二版都使用过.包含相关目的地信息的表名为Locations。如果你映射到此表,就必须指定表名:

[Table("Locations")]

public class Destination

Table标记还有一个参数可以指定构架名.这里是一个与表名共同指定的例子:

[Table("Locations", Schema="baga")]

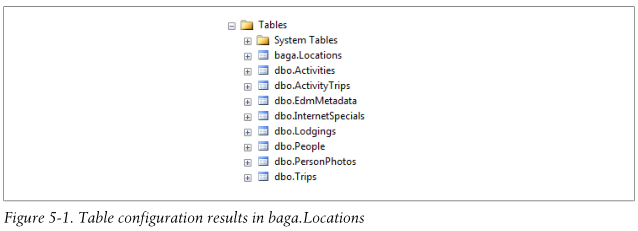
public class Destination

记住VB的语法有些不同:

<Table("Locations", Schema:="baga")>

Public Class Destination

图5-1显示了baga构架下的Locations表:

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/201201031906062669.png)

第二个参数不是必须的,使用Table("Locations") 标记是完全合法的.但是,如果你只想指定构架不想指定表名,仍然需要提供第一个参数,这个参数不能是empty,空格或null,因此你必须提供表名.

### 使用Fluent API配置表和构架名

Fluent API有一个ToTable方法用于指定表名称构架名.需要两个参数,第一个是表名,第二个是构架名.如同Data Annotations 一样,你可以提供表名参数而不提供构架名,但不能在指定构架时不包含表名:

### 映射属性名到数据库列

      不仅可以重新映射表名，也可修改预设的数据库列名。根据默认规则，Code First使用属性名作为映射的列名，但这并不总是符合需要的。例如，在原有的BreakAway数据库中，不仅包含有目的地信息的表名区别于默认设置，而且主键字段也被称为LocationID,而不是DestinationId.并且字段中还包含了一个称为LocationName的字段。

### 使用Data Annotations 修改默认的列名

你可能会回想起使用Column标记去修改列的数据类型。同样的标记也可以用来对列名进行修改，如代码5-1

Example 5-1. Specifying column names for properties using Data Annotations

[Column("LocationID")]

public int DestinationId { get; set; }

[Required, Column("LocationName")]

public string Name { get; set; }

### 使用Fluent API修改默认列名

HasColumnName是一个用于指定属性在数据库中列名的Fluent方法，如代码5-2所示，效果与图5-1所示的相同。bgga.Locations表配置结果合适地映射了Destination类。注意HasColumnName可以附加到现有的属性配置上。

Example 5-2. Specifying column names for properties using the Fluent API

public class DestinationConfiguration :

EntityTypeConfiguration<Destination>

{

public DestinationConfiguration()

{

Property(d => d.Nam

.IsRequired().HasColumnName("LocationName");

Property(d => d.DestinationId).HasColumnName("LocationID");

### 对复杂类型的列名施加影响

第3章，我们已经创建了一个来自于Address类的复杂类型，将在Person类中添加了一个Address属性。你可能会想起Code First对Person.Address映射列的命名方式为Address\_StreetAddress 或 Address\_State等。

你可以使用前面对Destination类类似的配置方法来调整Adderss复杂类型在数据库中的列名。

Example 5-3. Configuring column names to be used in the table of any class that hosts the complex type

[ComplexType]

public class Address

{

public int AddressId { get; set; }

[MaxLength(150)]

[Column("StreetAddress")]

public string StreetAddress { get; set; }

[Column("City")]

public string City { get; set; }

[Column("State")]

public string State { get; set; }

[Column("ZipCode")]

public string ZipCode { get; set; }

}

现在任何在Address复杂类型中的属性都可以使用这些Data Annotations 标记来指定列名了。

如果你需要流畅地配置列名，你可以在两种方式内选择：1）任何以Address作为属性的类中统一进行命名；2）显示地在每个“宿主”类中进行单独配置；

代码5-4和5-5显示Fluent API配置应用于Address复杂类型上的情况。这与使用Data Annotations 进行配置时，效果相同，所有使用Address的类都会使用这些列名。

Example 5-4. Configuring a complex type column name from the modelBuilder

modelBuilder.ComplexType<Address>()

.Property(p => p.StreetAddress).HasColumnName("StreetAddress");

Example 5-5. Configuring the complex type column name from a configuration class

public class AddressConfiguration :

ComplexTypeConfiguration<Address>

{

public AddressConfiguration()

{

Property(a => a.StreetAddress).HasColumnName("StreetAddress");

}

}

这种配置的效果使得Person表中的Address\_StreetAddress简化为SteetAddress.

代码5-6显示了一个配置Person实体的导航属性来控制StreetAddress列名的例子。首先，配置是分别在OnModelCreating方法和configuration类中进行暴露。在Person类中的配置将会只影响People表的列名。如果在另一个类中也有Address属性，它的表不会使用这种列名。

Example 5-6. Configuring the StreetAddress column name to be used when Address is a property of Person

modelBuilder.Entity<Person>()

.Property(p => p.Address.StreetAddress)

.HasColumnName("StreetAddress");

public class PersonConfiguration : EntityTypeConfiguration<Person>

{

public PersonConfiguration()

{

Property(p => p.Address.StreetAddress)

.HasColumnName("StreetAddress");

}

}

让多个实体映射到同一个表：aka表切分

      通常一个数据库表中虽然有很多列，但在很多场景只需要使用其中的一部分，其他的只是一些附加的数据。当我们映射一个实体到这样的表以后，你会发现要浪费资源来处理一些无用的数据。表切分技术可以解决这个问题，这种技术允许在一个单独表中访问多个实体。下面就介绍如何使用Code First来配置表切分功能。

       在映射到一个现存的数据库时你可能更想使用表切分技术，因为这种情况下你会发现场景与上面描述的很相似，在这种情况下你甚至都想让Code First来会创建一个新的数据库。

      我们更想person的photo和caption直接储存在People表中而不是存在单独的PersonPhotos表中。由于我们访问person的姓名与个人信息频率要远高于photo，让photo放在一个单独的类中会工作得很好。代码5-7回顾了PersonPhoto的类，也回顾了Person类中的Photo属性。我们来配置Photo属性为一个image列，正如我们在第2章对Destination.Photo属性所做的一样。

Example 5-7. The PersonPhoto class with Data Annotations

[Table("PersonPhotos")]

public class PersonPhoto

{

[Key , ForeignKey("PhotoOf")]

public int PersonId { get; set; }

[Column(TypeName="image")]

public byte[] Photo { get; set; }

public string Caption { get; set; }

public Person PhotoOf { get; set; }

}

使用默认规则，Code First会将PersonPhoto映射到它自已的表上，表名依照我们的配置命名为PersonPhotos.

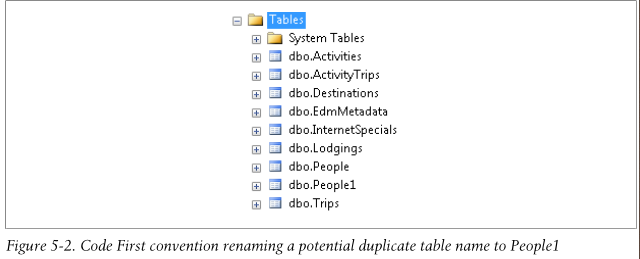
为了将多个实体映射到一个通用的表中，实体必须遵循如下规则：

* 实体必须是一对一关系
* 实体必须共享一个通用键

Person和PersonPhoto类符合这些条件；（原文中PersonTable应有误，在此更改为PersonPhoto,译者注）

### 使用Data Annotations 映射到通用表

      设置这种映射需要使用Data Annotations 的Table标记。因为我们知道Person实体映射到People表，你可以配置PersonPhoto类也映射到这个表中。但是你需要对所有相关类指定表名，否则，EF框架就会使用另一个默认规则来避免表名重复。你可以在图5-2看到这种错误的结果。因为Table标记的配置，PersonPhoto表被命名为People，因此当Code First尝试自动为Person类命名表名时，发现People已经使用过了，只好命名为People1.

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/201201041733528171.png)

因此我们必须对两个类作同一命名：

[Table("People")]

public class Person

[Table("People")]

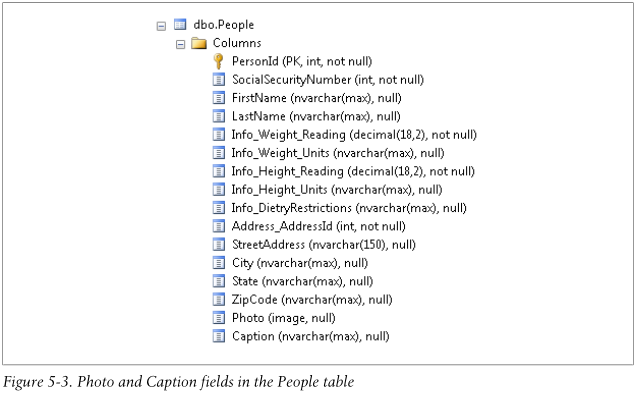
public class PersonPhoto

随着这种对模型的调整，Code First会再次重建数据库。People表现在有了Photo和Caption字段（见图5-3），不再有PersonPhotos表。

如果你回到InsertPerson和UpdatePerson方法，你会发现不需要任何修改方法仍然能够工作。类仍然是单独的。EF框架可以识别表的映射，对数据库执行正确的命令。

更有意思的是当你查询一个实体时不用再浪费资源从其他实体中抽取数据了。

再次执行对Person类的查询，如context.People.ToList()，EF框架项目只将这些列映射到Person类而没有映射到PersonPhoto的任何字段。（代码5-8）。

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/201201041733527233.png)

Example 5-8. SQL Query retrieving subset of table columns

SELECT

[Extent1].[PersonId] AS [PersonId],

[Extent1].[SocialSecurityNumber] AS [SocialSecurityNumber],

[Extent1].[FirstName] AS [FirstName],

[Extent1].[LastName] AS [LastName],

[Extent1].[Info\_Weight\_Reading] AS [Info\_Weight\_Reading],

[Extent1].[Info\_Weight\_Units] AS [Info\_Weight\_Units],

[Extent1].[Info\_Height\_Reading] AS [Info\_Height\_Reading],

[Extent1].[Info\_Height\_Units] AS [Info\_Height\_Units],

[Extent1].[Info\_DietryRestrictions] AS [Info\_DietryRestrictions],

[Extent1].[StreetAddress] AS [StreetAddress],

[Extent1].[City] AS [City],

[Extent1].[State] AS [State],

[Extent1].[ZipCode] AS [ZipCode]

FROM [dbo].[People] AS [Extent1]

感谢在Person和PersonPhoto类之间建立的关系，你可以轻松地加载photo数据，例如使用context.People.Include(“Photo”)或者显示地在需要时加载，或者使用延迟加载。这些都是不能使用标量属性来实现的功能。

## 延迟加载分割表数据

    虽然DbContext默认情况下启用 延迟加载，但我们目前尚未讨论如何使你的代码类利用延迟加载。事实上，由于在EF4引入了POCO的支持，任何EF框架中的简单类都可具有这项功能。任何使用virturl关键字（在Visual Basic为Overridable）的导航属性都会在第一次数据库检索时自动应用延迟加载。  
例如，你可以改变Photo的属性，以便它可以延迟加载：

[Required]

public virtual PersonPhoto Photo { get; set; }

如下的代码示例显示了Photo的延迟加载。一个查询返回了数据库中所有的Person数据，但相关的Photo数据没有在查询中使用。然后代码会对某个Person执行一些任务。在最后一行，代码显示Person的图片Caption被导航到了Person.Photo.Caption.由于此时Photo在内存中还没有加载，此调用将触发实体框架运行后台的查询，并从数据库中检索数据。就代码而言，Photo就在那里。如果Photo属性不是虚拟的，或延迟加载被上下文显示地禁用，最后一行代码将抛出一个异常，因为照片的属性将会为null：

var people = context.People.ToList();

var firstPerson = people[0];

SomeCustomMethodToDisplay(firstPerson.Photo.Caption);

### 使用Fluent API切分表

正如Data Annotations 一样，你只需要为类指定表名即可，但是必须要为所有包含的类指定同样的表名。如下代码显示了使用Fluent API配置切分表的方法，直接使用了OnModelCreating方法中的modelBuilder实例,你也可以使用相应的EntityTypeConfiguration类来配置：

modelBuilder.Entity<Person>().ToTable("People");

modelBuilder.Entity<PersonPhoto>().ToTable("People");

### 映射一个单独的实体到多个表

       现在，我们将完全翻转最后一个场景。我们不需要经常看到照片，因为我们只需要看名字就可以。另一个比较常见的情况是当我们检索目的地时希望能够随时看到目的地的照片。如果你映射到一个现有的数据库，有可能照片已经被存储在一个数据库单独的表中，原因可能是为了规范化，性能，或其他什么原因。但您的域模型表示所有的数据都是在一个单独的类中。这就需要将单个的Destination类分配到两个数据库的表中，以获取所有的详细信息。这种映射称为实体分割。

这个映射的关键是Code First配置一系列属性映射到一个特定的表的能力。实现这个功能不能使用Data Annotations ，因为Data Annotations 没有子属性的概念。

Fluent API有一个Map方法可以使用你分配一个属性的列表到一个表名。我们将使用这个方法映射部分Destination的属性到Locations表和其他一些属性映射到名为LocationPhotos的表。务必不要跳过任何属性！  
关于在Destinaion类中进行了一系列的配置，代码5-9展示了在DestinationConfiguration类中的配置方法：

Example5-9. DestinationConfiguration with Entity Splitting mapping at the end

public class DestinationConfiguration :

EntityTypeConfiguration<Destination>

{

public DestinationConfiguration()

{

Property(d => d.Name)

.IsRequired().HasColumnName("LocationName");

Property(d => d.DestinationId).HasColumnName("LocationID");

Property(d => d.Description).HasMaxLength(500);

Property(d => d.Photo).HasColumnType("image");

// ToTable("Locations", "baga");

Map(m =>

{

m.Properties(d => new

{d.Name, d.Country, d.Description });

m.ToTable("Locations");

});

Map(m =>

{

m.Properties(d => new { d.Photo });

m.ToTable("LocationPhotos");

});

}

}

Map配置中的Lambda表达式是多行表达式语句，你看到里面的Lambda表达式有分号。多行的Lambda语句VB也是支持的，相同的配置类在VB中如下所示：

Example 5-10. DestinationConfiguration using Visual Basic syntax

Public Class DestinationConfiguration

Inherits EntityTypeConfiguration(Of Destination)

Public Sub New()

Me.Property(Function(d) d.Name)

.IsRequired().HasColumnName("LocationName")

Me.Property(Function(d) d.DestinationId)

.HasColumnName("LocationID")

Me.Property(Function(d) d.Description).HasMaxLength(500)

Me.Property(Function(d) d.Photo).HasColumnType("image")

Me.Ignore(Function(d) d.TodayForecast)

' Me.ToTable("Locations") REM replaced by table mapping below

Me.Map(Sub(m)

m.Properties(Function(d) New With

{Key d.Name, Key d.Country, Key d.Description})

m.ToTable("Locations")

End Sub)

Map(Sub(m)

m.Properties(Function(d) New With {Key d.Photo})

m.ToTable("LocationPhotos")

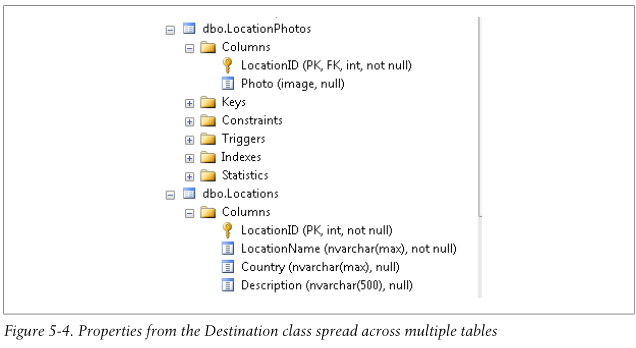
End Sub)

End Sub

End Class

最后，我们可以看到在图5-4数据库的效果。

请注意即使我们只有Photo属性映射到LocationPhotos表，Code First也会共享该表的主键和外键。这也创建了Location和LocationPhotos之间的PK / FK约束。有趣的是，没有为LocationPhotos表中定义级联删除。但是，EF框架知道，如果你删除一个目的地，它必须建立一个跨越两个表的删除命令。让我们来看看EF框架在对Destination对象的各种CRUD操作生成的SQL。  
代码5-12显示了我们在第2章中插入一个Destination对象方法，然后调用了SaveChanges方法保存数据到数据库。

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/201201041733533579.png)

Example 5-12. Insert a single object that maps to two database tables

private static void InsertDestination()

{

var destination = new Destination

{

Country = "Indonesia",

Description = "EcoTourism at its best in exquisite Bali",

Name = "Bali"

};

using (var context = new BreakAwayContext())

{

context.Destinations.Add(destination);

context.SaveChanges();

}

}

exec sp\_executesql

N'insert [dbo].[Locations]([LocationName], [Country], [Description])

values (@0, @1, @2)

select [LocationID]

from [dbo].[Locations]

where @@ROWCOUNT > 0 and [LocationID] = scope\_identity()',

N'@0 nvarchar(max) ,@1 nvarchar(max) ,@2 nvarchar(500)',

@0=N'Bali',@1=N'Indonesia',@2=N'EcoTourism at its best in exquisite Bali'

exec sp\_executesql

N'insert [dbo].[LocationPhotos]([LocationID], [Photo])

values (@0, null)',

N'@0 int',@0=1

第一个SQL语句向Locations表中插入数据，此语句中的代码返回了新生成的LocationID的值。第二个SQL语句，插入一个新行到LocationPhotos，包含了从第一个命令返回LocationID。由于该方法没有提供任何照片信息，照片字段的值插入到该表是空的。

例5-13显示了代码来查询，更新和删除Destination数据。你可以看到通过SQL，EF框架如何在响应SaveChanges方法时处理多个表的关系的。

Example 5-13. Query, update and delete a Destination

using (var context = new BreakAwayContext())

{

var destinations = context.Destinations.ToList();

var destination = destinations[0];

destination.Description += "Trust us, you'll love it!";

context.SaveChanges();

context.Destinations.Remove(destination);

context.SaveChanges();

}

—-RESPONSE TO QUERY

SELECT

[Extent1].[LocationID] AS [LocationID],

[Extent2].[LocationName] AS [LocationName],

[Extent2].[Country] AS [Country],

[Extent2].[Description] AS [Description],

[Extent1].[Photo] AS [Photo]

FROM [dbo].[LocationPhotos] AS [Extent1]

INNER JOIN [dbo].[Locations] AS [Extent2] ON [Extent1].[LocationID] = [Extent2].

[LocationID]

—RESPONSE TO UPDATE

exec sp\_executesql

N'update [dbo].[Locations]

set [Description] = @0

where ([LocationID] = @1)',

N'@0 nvarchar(500),@1 int',

@0='Trust us, you''ll love it!',@1=1

--RESPONSE TO DELETE

exec sp\_executesql N'delete [dbo].[LocationPhotos]

where ([LocationID] = @0)',N'@0 int',@0=1

exec sp\_executesql N'delete [dbo].[Locations]

where ([LocationID] = @0)',N'@0 int',@0=1

现在，我们来看看实体分割的结果，把所有的Destination数据再存到一个表中。如果您已经在Visual Studio中，删除了我们刚才添加的实体分割配置，恢复单个ToTable调用映射，从Destination映射到Locations表：

ToTable("Locations", "baga");

### 控制到映射到数据库中数据类型

每次你添加一个类模型，你也得添加到一个DbSet到BreakAwayContext 。 DbSet提供两个功能。首先，它返回一个特定类型的可查询数据集。其次可以让DbModelBuilder知道，每个数据集中引用的类型应包括在模型中。

但是确保一个类型成为你的模型的一部分，这不是唯一的途径。包括一个类型到模型中有三种方式：  
1。在上下文暴露类型的DbSet。  
2。另一个已经映射到类型中被引用（即，该类型可以通过另一种类型与模型接触）。  
3。从任何Fluent API调用的DbModelBuilder引用一个类型。  
你已经见过第一种方式。下面介绍其他两种方式。  
我们将添加一个新的类到模型中：Reservation。

Example 5-14. The new Reservation class

namespace Model

{

public class Reservation

{

public int ReservationId { get; set; }

public DateTime DateTimeMade { get; set; }

public Person Traveler { get; set; }

public Trip Trip { get; set; }

public DateTime PaidInFull { get; set; }

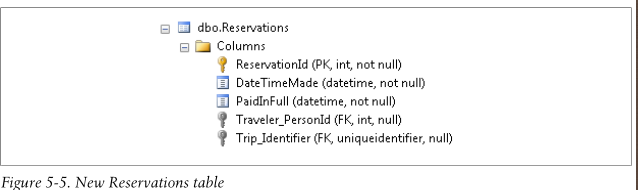
}

}

如果你运行程序，DbModelBuilder并不知道这个类的存在。因为即没有DbSet<Reservation>也没有其他两种形式的配置。因此，模型必须进行更改，数据库才会重新创建。如果你现在看数据库，里面不会有Reservation表。现在我们进入Person类，添加一个属性以便我们可以看到所有由Person设置的Resvervations。

public List<Reservation> Reservations { get; set; }

再次运行应用程序，现在你会看到数据库中有了一个Reservations表，如图5-5所示：

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/201201041733537101.png)

根据默认规则，由于Person是模型中的类，而Person找到了Reservationod ,Code First就会将Reservation类放进模型中。

现在来看看第三种默认规则：通过提供一个配置将类包括在模型中。

首先我们将Person类中的Reservation属性注释掉：

// public List<Reservation> Reservations { get; set; }

为reservation添加Fluent API配置。将配置封装到一个EntityTypeConfiguration类中和直接调用modelBuilder.Configurations的OnModelCreating方法效果是一样。只是为了让事情井井有条，我们将创建一个单独的配置类，示例5-15中所示。

Example 5-15. An empty configuration class for Reservation

public class ReservationConfiguration :

EntityTypeConfiguration<Reservation>

{

}

注意我们除了声明这个类以外什么都没做。里面还没有代码。这足以让你向DbModelBuilder配置中添加类，确保Reservation包含进模型中将映射到数据库Reservations表。

modelBuilder.Configurations.Add(new ReservationConfiguration());

现在你已经看到Code First包含一个类到模型的默认方式，现在我们来学习如何配置模型来排降一个类。

### 避免类型包含进模型中

您可能在应用程序中为了某些目的定义了一些类，但并不要求它些在数据库被持久化。即使你不为它们定义一个DbSet或任何配置，它们也可能的被另一种已经映射的类型所访问而被拉入模型，在查询或更新数据库时被非预期创建在数据库中。

但是，您可以明确地告诉Code First忽略一个类，不要使其成为模型的一部分。

### 使用Data Annotations 来忽略类型

NotMapped特性标记可以用来指导Code First从模型中排队类型：

[NotMapped]

public class MyInMemoryOnlyClass

### 使用Fluent API配置来忽略类型

使用Fluent API，你需要使用Ignore方法来避免类型被扯入模型中。如果你想忽略一个类，你需要直接从DbModelBuilder来实现，不能在EntityTypeConfigraiton内进行：

modelBuilder.Ignore<MyInMemoryOnlyClass>();

### 理解属性映射和可访问性

有各种因素可以影响你的类是否能被Code First所识别和映射。下面是一些在定义类时需要注意的规则，可以来了解默认规则需要什么，如何通过配置来改变默认映射。

### 标量属性映射

如果属性性可以转化为EDM支持的类型称为标量属性，映射是唯一的。

合法的EDM类型有：Binary, Boolean, Byte, DateTime, DateTimeOffset, Decimal, Double, Guid, Int16, Int32, Int64, SByte, Single, String, Time.

标量属性不能映射到一个被忽略的EDMonton类型（如枚举类型或无符号整型数）；

### 属性的可访问性，Getters和Setters

1.public属性将会被Code First自动映射。

2.Set访问器可以用更严格的访问规则界定，但get访问器必须保持public才能被自动映射；

3.非公开的属性必须使用Fluent API配置才能被Code First所映射；

对于非公开属性，这意味只有执行配置的位置才能访问该属性。

例如，如果Person类有一个internal Name的属性，使用与PersonContext类相同的程序集，你可以在Person context的OnModelCreating方法中调用modelBuilder.Entity<Person>().Property(p => p.Name)。这就会使用该属性包含进你的方法。但是如果Person和PersonContext定义在单独的程序集中，你就需要添加一个PersonConfiguration类（EntityConfiguration<Person>)到Person的同一程序集中，以执行配置类内的配置。这要求包含有域类的程序集必须添加对EntityFramework.dll的引用。PersonCOnfig配置类可以在PersonContext的OnModelCreating方法中被注册。

类似的方法可以用于受保护的和私有的属性。但是，配置类必须嵌套在类内部，成为模型的一部分，这样才能访问私有或受保护的属性。下面是一个这样的例子，使用private隐藏了Name属性，但是允许外部代码使用CreatePerson方法对Name属性进行设置。嵌套的PersonConfig类可以访问本地复制的Name属性。

public class Person

{

public int PersonId { get; set; }

private string Name { get; set; }

public class PersonConfig : EntityTypeConfiguration<Person>

{

public PersonConfig()

{

Property(b => b.Name);

}

}

public string GetName()

{

return this.Name;

}

public static Person CreatePerson(string name)

{

return new Person { Name = name };

}

}

当我们配置类为为嵌套类时，可以使用下列代码：

modelBuilder.Configurations.Add(new Person.PersonConfig());

一个常见的场景是为了避免开发者在代码中修改某个特定的属性（如PersonId)，使用set访问器来将属性设置为private或internal.这种场景的实现归功于上述所列的第二个规则：Set访问器可以用更严格的访问规则界定，但get访问器必须保持public才能被自动映射；EF框架必须使用反射才能访问非公开的set访问器，但当运行于中等信任的模式时这并不提供支持。除了中等信任的情况以外，这意味着当真实对象作为查询或插入的结果时，上下文将能够填充受限的属性。上下文也能够以查询或插入的数据为属性设置值--即使上下文和域类处于单独的程序集或名称空间里。这即可以工作在有键值的情况，也可以工作在没有键值的情况下。

### 避免属性被包含在模型中

默认规则里，所有同时拥有get和set访问器的公开属性都会包含进模型中。

Code First使用使用相同的配置方法[—Data Annotations](mailto:%E2%80%94@DA)的NotMapped标记或Fluent API的Ignore配置方法---来排除类中的属性。

一个典型的不想储存在数据库中的属性例子是在类中使用其他属性计算出来的属性。例如，你可能想很容易地访问一个人的full name，这是根据First Name和Last Name合并计算得到的。类可以计算它而没有必要将其存进数据库。

如果一个属性吸有get或set访问器，也不会被包含进模型中。

如果你在Person类中有FullName属性，可以只设置get访问器而不设置set访问器来实现排除在映射之外。

public string FullName

{

get { return String.Format("{0} {1}",

FirstName.Trim(), LastName); }

}

但是，你可以还会有一些同时拥有get和set访问器的属性不想持久化到数据库。例如，Destination类可能会有一个字符串包含有当前的天气预报。但你可能想在程序的什么地方弹出预报内容，而不想对预报本身作出什么探究。

private string \_todayForecast;

public string TodayForecast

{

get { return \_todayForecast; }

set { \_todayForecast = value; }

}

这种情况下你不想持久化天气预报信息到数据库。EF框架在执行映射到Destination的查询或修改数据库表的时候不能包含这个属性。

### 使用Data Annotations 忽略属性

使用Data Annotations ，必须应用NotMapped特性标记：

[NotMapped]

public string TodayForecast

### 使用Fluent API忽略属性

在Fluent API，你需要配置实体忽略一个属性。下列是一个在DestinationConfiguration类中使用Ignore方法的例子：

Ignore(d => d.TodayForecast);

注意：在使用NotMapped或Ignore对private属性进行设置时有一个已知的bug。你可以MSDN找到关于此bug的描述。2011-8-18，微软评论声称：此bug已修复，并且会在下一个主要版本的Code First中发布。

## 映射到继承层次结构

EF框架支持各种模型中的继承层次结构。无论你使用Code First，Model First还是Database First来定义模型都不用担心继承的类型问题，也不用考虑EF框架如何使用这些类型进行查询，跟踪变更和更新数据。

使用Code First的默认继承：每层次结构映射表（TPH）

TPH描述了映射继承类关系到独立数据库表的方法，这种方法使用鉴别列来识别是否为子类型。这是Code First默认规则使用的表映射方法。为了观察这种行为，我们对模型做两处修改。首先我们从Lodging类里移除IsResort属性，然后创建一个单独的Resort类继承自Lodging类。代码5-16显示了这些类：

Example 5-16. Modified Lodging class and a new Resort class that derives from Lodging

public class Lodging

{

public int LodgingId { get; set; }

[Required]

[MaxLength(200)]

[MinLength(10)]

public string Name { get; set; }

public string Owner { get; set; }

// public bool IsResort { get; set; }

public decimal MilesFromNearestAirport { get; set; }

[InverseProperty("PrimaryContactFor")]

public Person PrimaryContact { get; set; }

[InverseProperty("SecondaryContactFor")]

public Person SecondaryContact { get; set; }

public int DestinationId { get; set; }

public Destination Destination { get; set; }

public List<InternetSpecial> InternetSpecials { get; set; }

}

public class Resort : Lodging

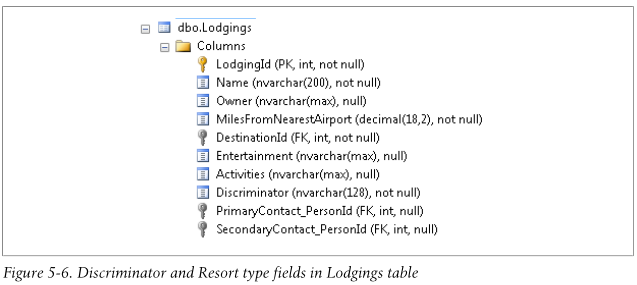
{

public string Entertainment { get; set; }

public string Activities { get; set; }

}

图5-6显示了数据库使用默认规则后的变化：

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/201201041733547351.png)

Resort信息储存在Lodgings表中，Code First创建了一个列命名为Didcriminator。注意这是一个非可空列，类型为nvarchar(128)。默认情况下，Code First会使用每个类型在继承层次中的类名作为discrimnator列的存储值。例如，如果你添加和运行InsertLogdging方法（代码5-17），由EF框架生成的INSERT语句会将字符串“Lodging”放进新加入行的Discriminator列中。

Example 5-17. Code to insert a new Lodging type

private static void InsertLodging()

{

var lodging = new Lodging

{

Name = "Rainy Day Motel",

Destination=new Destination

{

Name="Seattle, Washington",

Country="USA"

}

};

using (var context = new BreakAwayContext())

{

context.Lodgings.Add(lodging);

context.SaveChanges();

}

}

作为可选方案，代码5-18显示了一个新的Resort类型的实例：

Example 5-18. Code to insert a new Resort type

private static void InsertResort()

{

var resort = new Resort {

Name = "Top Notch Resort and Spa",

MilesFromNearestAirport=30,

Activities="Spa, Hiking, Skiing, Ballooning",

Destination=new Destination{

Name="Stowe, Vermont",

Country="USA"}

};

using (var context = new BreakAwayContext())

{

context.Lodgings.Add(resort);

context.SaveChanges();

}

}

这一次，EF框架将会在Discriminator列中插入字符串“Resort”.

这种默认规则的行为基于你可能会加入更多的衍生Lodging类型。如果Lodging类只有有Resort和没有这两种情况，此时discrimainator列就可指定为Boolen型，同时也就没有扩展层次的空间了。这种灵活性在默认规则中工作得很好。

### 使用Fluent API定制TPH区分符字段

你可以通过指定配置来定制discriminator列的类型和命名，方法是使用Fluent API（Data Annotations 没有标记可用于定制TPH）。

使用Map配置方法来实现，前已述及，Map可用于实体分割。这个方法需要包含几个配置：

Example 5-19. Configuring the discriminator column name and possible values

Map(m =>

{

m.ToTable("Lodgings");

m.Requires("LodgingType").HasValue("Standard");

})

.Map<Resort>(m =>

{

m.Requires("LodgingType").HasValue("Resort");

});

注意我们看到了几个新的配置方法-Requires和HasValue.Requires是一个配置，在此用于指定一个discriminator列。HasValue也用于指定配置discriminator.你可以使用HasValue来指定在一个特定类型中使用什么样的值。我们告诉Code First使用LodgingType作为discriminator列的列名而不是使用默认的名称：Discriminator.默认规则规定，Code First使用类名作为Discriminator值（例如：“Lodging”).我们将其调整为“Standard”(Lodging基类适用）和“Resort”(派生类适用）。

你可能知道Loding的唯一派生类就是Resort，因此决定使用Boolean型数据，如用IsResort表示，这也是可以的。在这种情况下，值将是Boolean型。你不需要告知Code First这个事实，只需要在提供期望的值即可，Code First将正确地识别discriminator列的类型。

修改为布尔型的discriminator的代码见代码5-20.

Example 5-20. Configuring a discriminator column to be a boolean

Map(m =>

{

m.ToTable("Lodging");

m.Requires("IsResort").HasValue(false);

})

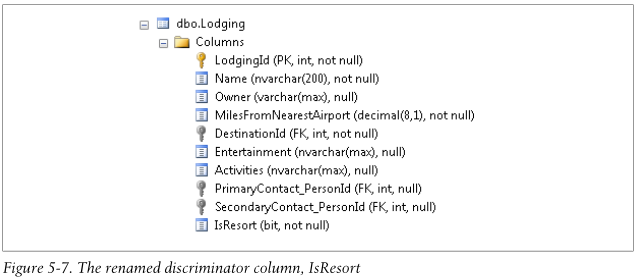
.Map<Resort>(m =>

{

m.Requires("IsResort").HasValue(true);

});

结果见图5-7：

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/20120104173355873.png)

配置每个类型映射表（TPT）

      TPH将所有层次的类都放在了一个表里，而TPT在一个单独的表中储存来自基类的属性。在派生类定义的附加属性储存在另一个表里，并使用外键与主表相连接。如果你的数据库构架使用单独的表表达层次关系，你需要对派生类进行显示的配置。下面是一个简单配置的例子，你需要指定派生类的表名，可以使用Data Annotations 也可以使用Fluent API来完成这项工作。

下面是对Resort类型的配置：

[Table("Resorts")]

public class Resort : Lodging

{

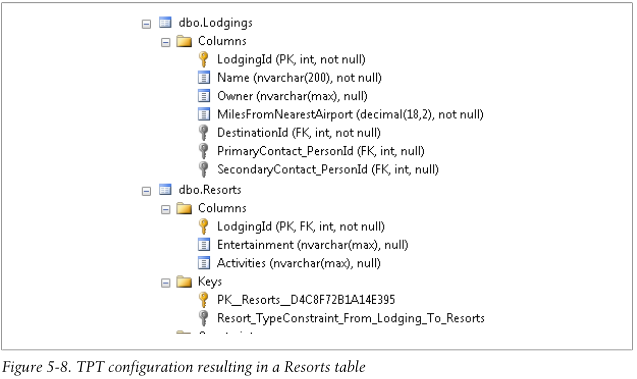
public string Entertainment { get; set; }

public string Activities { get; set; }

}

联合使用继承和Data Annotation 的Table特性标记将告知Code First要为Resort类型创建一个新表，由于该表继承自Lodging，它将继承Lodging的键属性。

图5-8显示了Lodgings和新建的Resorts表。注意：Lodgings不再包含discriminator也没有Resort的字段(Entertainment and Activities).

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/20120104173356217.png)

新增的Resorts表中有一个LodgingId类，这是一个主键也是一个外键，后者的名字被区分为：Resort\_TypeConstraint\_From\_Lodging\_To\_Resorts.

值得好奇的是，在Resort\_Type Constraint\_From\_Lodging\_To\_Resorts 键上没有定义级联删除。EF框架将在必要的时候来维护删除的数据。

当你添加一个新的Resort并调用 SaveChanges方法时，就触发EF框架首先在Lodging表添加以合适的值，然后返回一个新LodgingId值，然后使用这个LodgingId值在Resorts表中插入一个新行。

使用Fluent API你可以使用ToTable方法来获得TPT映射。你只需要指定派生实体的表名，Resort，这样Code First就会创建附加表和约束，如您在较长5-8所见。代码5-21显示了只接通过modelBuilder实例调用的例子。

Example 5-21. In-line ToTable mapping used for TPT inheritance

modelBuilder.Entity<Resort>().ToTable("Resorts");

你也可以从基类开始配置，使用Map方法得到Resort类型：  
Example 5-22. Mapping ToTable within the Map method

modelBuilder.Entity<Lodging>()

.Map<Resort>(m =>

{

m.ToTable("Resorts");

}

);

如果你想显示操作，你可以指定每个层级的表名。在这种情况下，Lodgings表已经被默认规则所指定，但是为了更明确的配置，我们使用了更为清晰的代码，如代码5-23：

Example 5-23. Mapping ToTable for a TPT inheritance from base entity

modelBuilder.Entity<Lodging>().Map(m =>

{

m.ToTable("Lodgings");

}).Map<Resort>(m =>

{

m.ToTable("Resorts");

});

这最后的变化很有趣，你可以从它的基类配置映射的派生类。但无法从实体<Resort>开始，然后添加<Lodging>的映射。

创建这个映射的所有三个变化达到同样的目的。如果需要，你可能想作出选择以更好地适应您的编码风格。

Configuring for Table Per Concrete Type (TPC) Inheritance

## 配置每概念层级类型一个表（TPC）

     TPC类似于TPT，除了每个类型的所有属性都储存在单独表里以外。在层级中没有一个核心表包含通用于所有类型的数据。这就使用需要映射一个继承结构层次到层叠（通用）的字段上。这对你将历史数据存存储在备用表时特别有用。可我们会将Lodgings和Resorts映射到一个resort表里，这个表里也包含Name,Ownert MilesFromNearestAirport字段。你可以使用Fluent API来配置你的结构层次映射到这样的表中。（Data Annotations 不支持TPC）。

    下面对Lodging/Resort结构层次再做些改变。

     TPC使用MapInheritedProperties方法来进行配置，只能在Map方法里才能访问。既然我们需要为派生类提供单独的表（这会复制继承的属性），我们需要联合设置Table和MapInheritedProperties方法来进行配置。

     注意你必须使用ToTable方法包含到基类实体的映射。虽然这在TPT不是必须的，但TPC是：

modelBuilder.Entity<Lodging>()

.Map(m =>

{

m.ToTable("Lodgings");

})

.Map<Resort>(m =>

{

m.ToTable("Resorts");

m.MapInheritedProperties();

});

MapInheritedProperties方法告知Code First它想要重映射所有继承自基类的属性到派生类表的新列里。

### 避免TPC里的映射异常

as a reminder in Example 5-24.

如果你尝试运行当前程序来检查配置，你会得到一个异常，描述存在映射冲突，因为DbModelBuilder 尝试创建新模型。这个冲突来自于Lodging类。

TPC要求任何TPC层级内的类通过一个显示定义的外键属性来建立关系。观察Lodging类，如代码5-24所示：

Example 5-24. A reminder of the Lodging class

public class Lodging

{

public int LodgingId { get; set; }

public string Name { get; set; }

public string Owner { get; set; }

public decimal MilesFromNearestAirport { get; set; }

public List<InternetSpecial> InternetSpecials { get; set; }

public Person PrimaryContact { get; set; }

public Person SecondaryContact { get; set; }

public int DestinationId { get; set; }

public Destination Destination { get; set; }

}

While the navigation to Destination is complemented by the DestinationId property, there are two navigation reference properties that do not have a foreign key  property: PrimaryContact and SecondaryContact. Code First leverages the database foreign key fields to take care of persisting the relationship. If you’ve been using Entity Framework since the first version, you may recognize this as independent associations, which were the only option for building relationships in Visual Studio 2008. Foreign Key associations, where we can have a foreign key property such as DestinationId in the class, were introduced to Entity Framework in Visual Studio 2010 and .NET 4. TPC can’t work with classes that have independent associations in them.

虽然到Destinaition的导航，是由DestinationId属性补充的，虽然有两个导航引用属性，但是没有外键属性：PrimaryContact和SecondaryContact。Code First利用数据库外键字段来维护关系的持久化。如果您使用过EF框架的第一个版本，您可以将此视为独立的关系，是Visual Studio 2008中建立关系的唯一选项。而外键关系，在这里我们可以使用外键属性如类中的DestinationId属性，被引入到Visual Studio 2010和.NET4下的EF框架中。 TPC不能与带有独立关联的类一同工作。（译者注：此处不知所云，请高手赐教！）

      为了解决这个问题，你必须给Lodging添加外键属性。对某些开发人员而言，这是一个被迫吞下的痛苦的药丸---为了让类加入而被迫遵守EF框架的约定。不幸的是，正如您可能已经关注到的，实现Code First的很多映射就需要有外键可用。

      请记住，在我们的域中，可能是一个Lodging既没有PrimaryContact，也没有SecondaryContact。当我们在第4章增加PrimaryContact和SecondaryContact导航属性时，Code First通过约定推断它们是可空类型（Optional）。新的外键的属性是整数，默认情况下，非可空。这样冲突就出现了，因为如果在外键中有一个值，Contact就不是可选的。因此，我们将创建新的具有可空性的外键属性。注意使用Nullable<T>泛型来创建PrimaryContactId和SecondaryContactId两个新属性，如代码5-25所示。

Example 5-25. Lodging class with nullable foreign keys

abstract public class Lodging

{

public int LodgingId { get; set; }

public string Name { get; set; }

public string Owner { get; set; }

public decimal MilesFromNearestAirport { get; set; }

public List<InternetSpecial> InternetSpecials { get; set; }

public Nullable<int> PrimaryContactId { get; set; }

public Person PrimaryContact { get; set; }

public Nullable<int> SecondaryContactId { get; set; }

public Person SecondaryContact { get; set; }

public int DestinationId { get; set; }

public Destination Destination { get; set; }

}

       我们还没有完成。记住Code First无法在没有协助的情况下识别出非常规的外键属性。这些新的属性不匹配任何三种Code First检测外键的可能模式（例如，PersonId）。所以你需要使用HasForeignKey来进行映射,如同例4-3。  
代码5-26显示了对这些属性在LodgingConfiguration类的两个现有配置。我们通过加入HasForeignKey映射对其进行了修改。

Example 5-26. Fixing up the model for unconventional foreign key properties

HasOptional(l => l.PrimaryContact)

.WithMany(p => p.PrimaryContactFor)

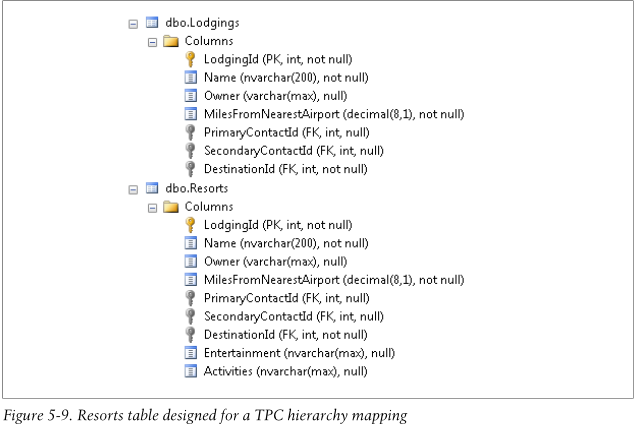
.HasForeignKey(p=>p.PrimaryContactId);

HasOptional(l => l.SecondaryContact)

.WithMany(p => p.SecondaryContactFor)

.HasForeignKey(p => p.SecondaryContactId);

最后所有的片断都安置在TPC所需的正确位置。模型将会对其进行验证，如图5-9所示。事实上，所有自Lodging类的继承字段现在都在Resorts表中。由我们协助Code First识别外键，这些外键也在Resorts表中进行了合适的配置。

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/201201041733572103.png)

### 使用抽象基类

     所有我们前面使用的继承都是基类为实体类的情况，我们也可以使用抽象类来进行工作。在使用Code First来构建抽象基类的模型之前我们先作一个快速的了解。

我们将Lodging类修改成为抽象基类。这意味着我们不再能直接使用Lodging类。它不能被实例化。我们只能使用派生于它的类。见例5-27，我们添加了第二个派生类：Hostel.

代码5-27列出了所有的三个类。  
Example 5-27. The abstract base class, Lodging, with its derived classes, Resort and Hostel

abstract public class Lodging

{

public int LodgingId { get; set; }

public string Name { get; set; }

public string Owner { get; set; }

public decimal MilesFromNearestAirport { get; set; }

public List<InternetSpecial> InternetSpecials { get; set; }

public Nullable<int> PrimaryContactId { get; set; }

public Person PrimaryContact { get; set; }

public Nullable<int> SecondaryContactId { get; set; }

public Person SecondaryContact { get; set; }

public int DestinationId { get; set; }

publicDestination Destination { get; set; }

}

public class Resort : Lodging

{

public string Entertainment { get; set; }

public string Activities { get; set; }

}

public class Hostel: Lodging

{

public int MaxPersonsPerRoom { get; set; }

public bool PrivateRoomsAvailable { get; set; }

}

当你将Lodging类变成抽象类，这意味着不能在程序中实例化Lodging类。如果控制台程序中有这样的代码将会导致编译错误。因此你应该将有关代码注释掉。一个好的方法是将方法使用#if/#endif包括起来，例如：

#if false

private static void InsertLodging()

{

var lodging = new Lodging

{

Name = "Rainy Day Motel",

Destination = new Destination

{

Name = "Seattle, Washington",

Country = "USA"

}

};

using (var context = new BreakAwayContext())

{

context.Lodgings.Add(lodging);

context.SaveChanges();

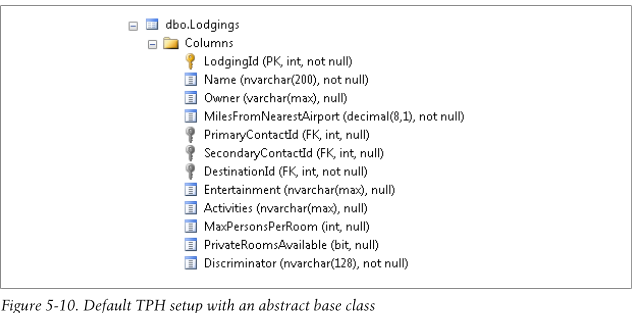
}

}

#endif

为了重新使用代码，可以将其改为#if true

我们已经移除了TPC的配置使模型和数据库表完全基于Code First的默认配置，这表示继承结构重新成为TPH。所有派生类的字段都会包含在Lodgings表中。在图5-10中可以看到尽管Lodging类是一个抽象类，对数据库的构建与其不是抽象类没有什么影响。但是，既然我们有了另一个派生类，还是有一些新的属性包含了进去，这些属性是Hostel类型提供的。

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/2012010417574583.png)

代码5-28显示了一系列插入新的Resort和新的Hostel的方法，然后从数据库中查出所有Lodgings来观察。

Example 5-28. Code to insert a Resort, then insert a Hostel, and finally to query Lodgings

private static void InsertResort()

{

var resort = new Resort

{

Name = "Top Notch Resort and Spa",

MilesFromNearestAirport = 30,

Activities = "Spa, Hiking, Skiing, Ballooning",

Destination = new Destination { Name = "Stowe, Vermont",

Country = "USA" }

};

using (var context = new BreakAwayContext())

{

context.Lodgings.Add(resort);

context.SaveChanges();

}

}

private static void InsertHostel()

{

var hostel = new Hostel

{

Name = "AAA Budget Youth Hostel",

MilesFromNearestAirport = 25,

PrivateRoomsAvailable=false,

Destination = new Destination {

Name = "Hanksville, Vermont",

Country = "USA" }

};

using (var context = new BreakAwayContext())

{

context.Lodgings.Add(hostel);

context.SaveChanges();

}

}

private static void GetAllLodgings()

{

var context = new BreakAwayContext();

var lodgings = context.Lodgings.ToList();

foreach (var lodging in lodgings)

{

Console.WriteLine("Name: {0} Type: {1}",

lodging.Name, lodging.GetType().ToString());

}

Console.ReadKey();

}

当EF框架发送INSERT命令到数据库，它使用“Resort”和“Hostel”分别填充了Discriminator列的数据。当调出所Lodgings的数据时，它过滤了Resort和Hostel识别标记，如代码5-29所示的SQL语句：

Example 5-29. SQL to retrieve all of the known types that derive from Lodging

SELECT

[Extent1].[Discriminator] AS [Discriminator],

[Extent1].[LodgingId] AS [LodgingId],

[Extent1].[Name] AS [Name],

[Extent1].[Owner] AS [Owner],

[Extent1].[MilesFromNearestAirport] AS [MilesFromNearestAirport],

[Extent1].[PrimaryContactId] AS [PrimaryContactId],

[Extent1].[SecondaryContactId] AS [SecondaryContactId],

[Extent1].[DestinationId] AS [DestinationId],

[Extent1].[Entertainment] AS [Entertainment],

[Extent1].[Activities] AS [Activities],

[Extent1].[MaxPersonsPerRoom] AS [MaxPersonsPerRoom],

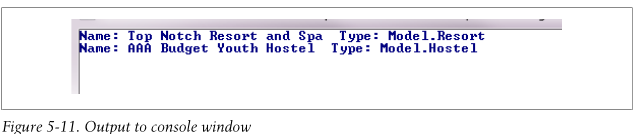
[Extent1].[PrivateRoomsAvailable] AS [PrivateRoomsAvailable]

FROM [dbo].[Lodgings] AS [Extent1]

WHERE [Extent1].[Discriminator] IN ('Resort','Hostel')

为什么要使用discriminators而不是简单地返回所有的Lodging数据？这覆盖了其他类型在数据库中不是模型的一部分的场景。

图5-11显示了前两个方法插入数据后调用GetAllLodgings方法的控制台输出结果。

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/201201042026184591.png)

      可以修改映射变成TPT或TPC。例如，如果你同时给Resort和Hostel类指定表明而Lodging类是抽象的，你就会得到三个数据库表，Resorts,Hosetls和Lodgings。代码5-28在没有任何修改的情况下可以正常工作。SQL命令将在必要时组合内联有关表，正如Lodging不是抽象的一样。所有模型中围绕着抽象基类的行为都简单遵从了EF框架第一版以来的行为。唯一的区别就是以新的方式定义了模型。

现在我们已经探索了抽象基类，现在将将abstract关键字从Lodging类移走，以便我们可以再次创建Lodging类的实例。也可将前面已经注释掉的任何方法重新启用：

public class Lodging

## 映射关系

到目前为止我们已经掌握了如何控制类或其属性的映射方法;最后我们来看看关系如何来映射.这包括控制外键列的命名以及多对对关系内联表的命名.第4章对关系的默认规则和配置进行了全面介绍.你应该对映射已经很熟悉了,这一部分将提供一些方式去控制映射的一些细节部分.

### 控制包含在类中的外键

你已经看到两个类的关系通过添加导航属性得到了创建.可以选择性地在一个单独类中设置外键.默认情况,Code First将会使用属性名作为列名.在第4章我们了解到,当添加DestinationId到Lodging类,Code First将添加了一个DestinationId列到数据库里并将其配置为一个外键 .

将外键属性的列名进行更改类似于更改其他属性的列名.改变外键属性名不会影响Code First检查是否为外键的能力.外键检测只针对属性名而不是属性映射到数据库的列名.

假如想把列名更为destination\_id.你可以使用Column的Data Annotations 标记直接将外键属性进行更改.

[Column("destination\_id")]

public int DestinationId { get; set; }

当然也可使用Fluent API来配置LodgingConfiguration类达成同样的目的:

Property(l => l.DestinationId).HasColumnName("destination\_id");

### 控制由Code First生产的外键

如果类中没有包含外键属性Code First就会自动创建一下.例Lodging类,一个Destination\_DestinationId列就被加到数据库中,就是一个自动添加的外键列.我们从Lodging类中移走DestinationId外键属性,让Code First自动创建一下.(如代码3-10).

Example 5-30. Foreign key property commented out

public class Lodging

{

public int LodgingId { get; set; }

public string Name { get; set; }

public string Owner { get; set; }

public decimal MilesFromNearestAirport { get; set; }

//public int DestinationId { get; set; }

public Destination Destination { get; set; }

public List<InternetSpecial> InternetSpecials { get; set; }

public Nullable<int> PrimaryContactId { get; set; }

public Person PrimaryContact { get; set; }

public Nullable<int> SecondaryContactId { get; set; }

public Person SecondaryContact { get; set; }

}

变更外键列名只能通过Fluent API.你可以使用Map方法来控制的映射,也可使用Map方法来控制关系的映射.

代码5-31显示了如何添加Map方法到关系来给外键名进行指定.

Example 5-31. Generated foreign key column configured

HasRequired(l => l.Destination)

.WithMany(d => d.Lodgings)

.Map(c => c.MapKey("destination\_id"));

现在你就已经观察到这些行为.将所有注释掉的代码恢复过来,移除上述Fluent API配置.你也可以取消DeleteDestinationInMemoryAndDbCascade方法的注释:

public int DestinationId { get; set; }

public Destination Destination { get; set; }

### 控制使用实体切分生成的外键

前面已经学习过有关实体拆分的技术。可以让一个类分配其属性到多个数据库表中。在此可以按意图对表中添加的外键列进行控制。

默认情况，外键将添加到你在实体分割配置中指定的第一个表中。你可以通过附加在生成外键的ToTable方法后来变更这种情况。例如，将Lodging实体划分为Lodgings表和LodgingInfo表。如果想要将外键放在LodgingInfo表的相关destination上，你将应按如下方式添加配置：

Example 5-32. Generated foreign key column configured

HasRequired(l => l.Destination)

.WithMany(d => d.Lodgings)

.Map(c => c.MapKey("destination\_id").ToTable("LodgingInfo"));

### 控制多对多关系中的内联表

前面我们已经在Acitvity和Trip之间引入了多对多关系，最终在数据库中生成ActivityTrips内联表（见图4-10）。

但是，在我们的域中将表名命名为TripActivites可能含义更明了。幸运的是可以使用Map方法配置这个表名。添加的配置设置见代码5-33：

Example 5-33. Many-to-many join table name changed

HasMany(t => t.Activities)

.WithMany(a => a.Trips)

.Map(c => c.ToTable("TripActivities"));

映射首先使用HasMany和WithMany方法来识别要配置的关系。一旦关系得到识别，就可以使用Map方法来指定映射。在映射关系中，你可使用ToTable方法来指定表名。我们来看看Map方法最终得到的结果。内联表更新后如图5-12所示：

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/201201050919276232.png)

你可以想要外键名叫做TripIdentifier和ActivityId.可以通过Map方法来实现列名的指定。

Example 5-34. Changing the many-to-many column names

HasMany(t => t.Activities)

.WithMany(a => a.Trips)

.Map(c =>

{

c.ToTable("TripActivities");

c.MapLeftKey("TripIdentifier");

c.MapRightKey("ActivityId");

});

注意MapLeftKey和MapRightKey方法用来指定列名。MapLeftKey影响指向配置的类的外键。你可以添加配置到TripConfiguration类，因此Trips是被配置的实体。这样Trip被视为左实体，Activity视为右实体。图5-13显示了更改列名后的效果：

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/201201050919281149.png)

## 小结

本章学习了Code First控制类与属性映射到数据库的技术。通过学习，您应掌握控制列，表乃至构架的方法。还学习了如何配置类映射到多个表以及如何将多个类指向单一的表的技术。然后花了很多时间研究了如何配置继承层次关系 ，很多配置只能使用Fluent API 来进行，最后你还学到如何调整到关系的映射。

总之，本章所学已经能够使Code First实现使用EDMX进行设计所实现的几乎全部功能 。通过Code Fi，你有能力将域类插入到EF框架中，而不使用设计器或任何附加模型。

# 第六章：控制数据库位置，创建过程和种子数据

前面我们已经提到默认规则和配置可以用于影响模型和数据库构架。本章你会看到如何使用Code First来控制数据库。

你将会学到Code First默认配置如何选择数据库，也会掌握如何修改默认规则或指定上下文使用真正的数据库。我们覆盖的主题将包括指向其他数据库引擎，部署应用程序，执行数据库有关的任何等。

你也可以学习到数据库初始化器可以用于控制数据库生成过程，添加种子数据到数据库中等。这在进行自动测试的场景时特别有用。

## 控制数据库位置

到目前为止我们都是引用了Code First的默认规则来选择应用程序的数据库目标。默认情况，Code First会使用localhost\SQLEXPRESS作为目标数据库引擎，并使用context类的全名作为数据库名（即命名空间+类名）。如果不符合要求你需要覆写默认规则然后告知Code First想要连接到哪个数据库。你可以Code First的连接工厂来选择数据库，修改或替换默认规则。可选择地，你也可以使用DbContext构造器或应用程序配置文件来告知Code First对某个特定的上下文使用指定的数据库。

小贴士：Code First可以在SQL Azure上使用与任何本地数据库相同的方法进行创建和初始化工作。你可在<http://www.windowsazure.com/zh-cn/develop/overview/>找到相关的文章。目前数据库供应商正在努力修改他们的产品已提供对Code First的支持。在使用Code First与第三方数据库引擎工作前一定要检查是否支持。

## 使用配置文件控制数据库位置

控制数据库连接最简单也最可靠的方法是使用配置文件。配置文件可以帮你绕过所有与数据库位置相关的约定，并能指定到你想使用的确切数据库。这种方法是非常有用的，如果你想改变你的上下文的连接字符串指向一个生产数据库，为您部署应用程序。

默认情况下，您添加到您的配置文件的连接字符串应该与context有相同的名称。连接字符串的名称可以是类型名称或完全限定的类型名称。在后面“使用DbContext构造器控制连接字符串名称”，你会看到如何让连接字符串的名称不匹配上下文的名称。添加一个App.config文件到BreakAwayConsole应用程序，内中包含BreakAwayContext的连接字符串，如例6-1所示。

<?xml version="1.0"?>

<configuration>

<connectionStrings>

<add name="BreakAwayContext"

providerName="System.Data.SqlClient"

connectionString="Server=.\SQLEXPRESS;

Database=BreakAwayConfigFile;

Trusted\_Connection=true" />

</connectionStrings>

</configuration>

小贴士：对熟悉使用EDMX文件创建连接字符串的人，请注意这不是一个EntityConnection String 而是一个简单的数据库连接字符串。使用Code First，你不需要引用元数据文件或System.Data.Entity Client 名称空间。

修改Main方法，调用InsertDestination方法，如代码6-2所示。

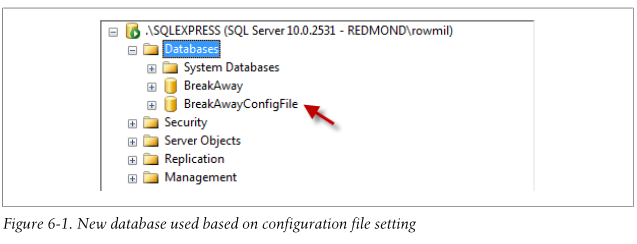
static void Main()

{

InsertDestination();

}

运行应用程序，你会发现，BreakAwayConfigFile数据库实例已在您的本地SQL Express上创建了（图6-1）。

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/201201051545263199.png)

Code First使用配置文件中的BreakAwayContext连接字符串匹配名为BreakAwayContext的上下文。因为在配置文件中发现了一个条目，就不再使用约定来定位数据库。连接字符串项，也已经被命名为DataAccess.BreakAwayContext，这是上下文的全名。

使用DbContext的构造器控制数据库名称

你已经看到如何通过配置文件中的连接字符串设置上下文，现在让我们来看看使用代码来控制数据库连接的方法。到目前为止，您只使用过DbContext的默认构造函数，还有一些其他可用的构造函数可供使用。其中大部分是更高级的方法，这将在这本书中进行介绍，有两个构造函数允许你影响连接到的数据库。

小贴士：如果您添加了一个连接字符串到您的配置文件，如在上一节所示，开始本节之前，一定要去掉它。记住，配置文件压倒一切，包括在本节的功能。

DbContext有一个构造函数使用一个字符串参数来控制数据库的名称。如果您使用此构造器，您提供的值将被用来代替context的全名。添加到BreakAwayContext构造函数接受一个数据库名称的字符串值，并把它传递给基构造器（例6-3）。请注意，您也必须要加入默认的构造器，以确保所有现有的代码从前面的章节继续工作。

public BreakAwayContext()

{ }

public BreakAwayContext(string databaseName)

: base(databaseName)

{ }

修改Main方法调用 SpecifyDatabaseName方法 (Example 6-4).

Example 6-4. SpecifyDatabaseName method added to application

static void Main()

{

SpecifyDatabaseName();

}

private static void SpecifyDatabaseName()

{

using (var context =

new BreakAwayContext("BreakAwayStringConstructor"))

{

context.Destinations.Add(new Destination { Name = "Tasmania" });

context.SaveChanges();

}

}

新方法使用的构造器，就是你刚刚添加的，用于指定数据库名称。使用的此名称，就不是上下文的全名了。运行应用程序，你会看到一个名为BreakAwayStringConstructor数据库已在您的本地SQL Express实例中创建。

### 使用DbContext构造器控制连接字符串

在本章的前面，你看到，你可以通过在配置文件中加入与你的上下文的名称相同的连接字符串指定一个数据库。如果您使用的DbContext构造函数接受一个数据库名，EF框架就会寻找一个与连接字符串的名称相匹配的数据库的名称。换句话说，默认的构造函数，实体框架会寻找名为BreakAwayContext的连接字符串，而使用与示例6-4中使用的构造函数，它会期望一个名为BreakAwayStringConstructor的连接字符串。  
您还可以强制上下文从配置文件中所提供的name= [connection string name]获取连接字符串。这样，你就不需要依靠名称匹配，因为你明确地提供了一个连接字符串。如果没有找到有具体指定名称的连接字符串，就会抛出一个异常。  
例6-5显示了如何修改breakAwayContext默认构造器，以确保连接字符串始终是从配置文件加载。

public BreakAwayContext()

:base("name=BreakAwayContext")

{ }

### 重用数据库连接

DbContext另一个构造器，允许您提供一个DbConnection的实例。这可能是有用的，如果你有其他的应用程序逻辑与DbConnection相关，或者如果你想重用在多个环境下的同一个连接。要看到这种行为，添加另一个构造器BreakAwayContext接受一个DbConnection，然后通过DbConnection基构造器传递值，如例6-6所示。你还会发现，你需要指定一个contextOwnsConnection的值。此参数控制context是否拥有连接的所有权。如果设置为true，连接将会随上下文一起被释放。如果设置为false，您的代码将需要关注连接的释放问题。

小贴士：添加这个构造器你需要引用System.Data.Common名称空间。

public BreakAwayContext(DbConnection connection)

: base(connection, contextOwnsConnection: false)

{ }

修改Main方法调用新的ReuseDbConnection方法，如Example 6-7所示：

小贴士：你需要引用System.Data.SqlClient名称空间，因为此代码使用SqlConnection类型。

static void Main()

{

ReuseDbConnection();

}

private static void ReuseDbConnection()

{

var cstr = @"Server=.\SQLEXPRESS;

Database=BreakAwayDbConnectionConstructor;

Trusted\_Connection=true";

using (var connection = new SqlConnection(cstr))

{

using (var context = new BreakAwayContext(connection))

{

context.Destinations.Add(new Destination { Name = "Hawaii" });

context.SaveChanges();

}

using (var context = new BreakAwayContext(connection))

{

foreach (var destination in context.Destinations)

{

Console.WriteLine(destination.Name);

}

}

}

}

ReuseDbConnection构造一个SqlConnection，然后重用它来构造两个单独的BreakAwayContext实例。在这个例子中，在SqlConnection是在代码中定义的连接字符串。然而，Code First并不关心你否获得连接。您可以从资源文件该连接字符串。您也可以使用一些现有的组件，让您获得现有的DbConnection的实例。

### 使用连接工厂控制数据库位置

控制所使用的数据库的一个最后的选择是通过更换Code First使用默认约一。Code First使用约定是通过Database.DefaultConnectionFactory来进行。连接工厂实现IDbConnectionFactory接口，并负责上下文的命名，并指明为要使用的数据库创建一个DbConnection。EF框架包含两个连接工厂实现，你也可以创建自己的。

### 使用SqlConnectionFactory

Code First的默认连接工厂是SqlConnectionFactory。此连接工厂将使用SQL Client（System.Data.SqlClient的）数据库引擎连接到数据库。默认的行为，将选择在localhost\ SQLEXPRESS创建数据库，并使用上下文类型的完全限定名作为数据库的名称。集成身份验证，将用于与数据库服务器进行身份验证。  
你可以通过指定的连接字符串段，来覆写默认规则。这些片段使用SqlConnectionFactory构造函数相同的语法，在连接字符串中使用。例如，如果你想使用不同的数据库服务器，您可以指定服务器段的连接字符串：

Database.DefaultConnectionFactory =

new SqlConnectionFactory("Server=MyDatabaseServer");

可选地，你也可使用不同的验证方式来连接数据库服务器：

Database.DefaultConnectionFactory =

new SqlConnectionFactory("User=MyUserName;Password=MyPassWord;");

Working with SqlCeConnectionFactory

使用SqlCeConnectionFactory  
EF框架还包括SqlCeConnectionFactory，它使用SQL Compact Client 连接到SQL ServerCompact Edition数据库。默认情况下，数据库文件名匹配上下文类的完全限定名，创建在| ApplicationData|目录。（对可执行程序而言，| ApplicationData|位于应用程序所在目录，对web应用程序，伴于网站根目录下的的App\_Data子目录内。

小贴士：安装SQL Server Compact Edition

在使用SQL Server Compact Edition前，需要进行安装。可以通过NuGet来进行安装。安装SqlServerCompact 的NuGet包的到你的BreakAwayConsole项目的方法是：右键单击项目并选择：Add Library Package Reference….，从弹出的对话框中选择Online并查找SqlServerCompact.

修改Main方法，如代码6-8所示，设置SqlCeConnectionFactory，然后调用InsertDestination方法（第2章创建）。连接工厂包含在System.Data.Entity.Infrastructure名称空间，需要添加对其的引用。在运行代码前请读完本节。

static void Main()

{

Database.SetInitializer(

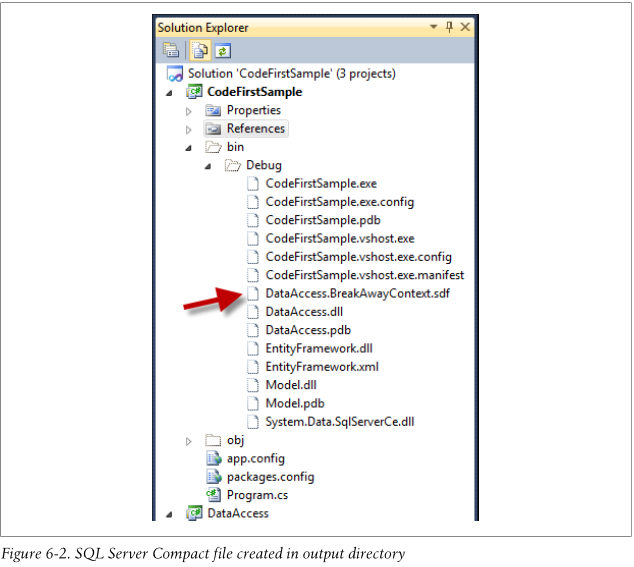
new DropCreateDatabaseIfModelChanges<BreakAwayContext>());

Database.DefaultConnectionFactory =

new SqlCeConnectionFactory("System.Data.SqlServerCe.4.0");

InsertDestination();

}

请注意，您需要指定一个字符串，标识数据库引擎（称为provider invariant name）。这个字符串是数据库供应商提供的唯一标识。大多数供应商保持不同版本之间使用相同的标识符，但SQL Compact为每个版本使用不同的标识符。这是因为SQL Compact数据库引擎并不向后兼容的，例如，您不能使用4.0引擎连接到一个3.5数据库。SqlCeConnectionFactory需要知道provider使用的版本，所以它需要你提供这个字符串。  
如果你想测试一下这个代码，你需要到你的模型一个小的变化。早在第3章，我们配置Trip.Identifier为数据库生成的key。标识符是一个GUID属性，在SQL Server下没有任何问题。 SQL Compact，不能产生的GUID列的值。如果你想运行该应用程序，删除或注释掉Data Annotations或Fluent API配置的Trip.Identifier（作为数据库生成列）。  
一旦你做出了这种变化，你可以运行该应用程序，你会发现一个DataAccess.BreakawayContext.sdf文件是在您的应用程序的输出目录（图6-2）创建。现在，你已经看到SQL默认规则的行为，继续前进，重新启用以前的配置，使Trip.Identifier能够在数据库里生成。  
[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/201201051545261705.png)

### 写一个定制的连接工厂

到目前为止，您已经看到，连接工厂已经包含EF框架中，但你也可以通过实现IDbConnectionFactory接口来创建自已的连接工厂。  
这个接口很简单，包含了一个单一的创建连接的方法，它接受上下文的名称，并返回一个DbConnection。  
在本节中，您将构建一个自定义的连接工厂，与SqlConnectionFactory非常相似，但它将只使用上下文类的名称，而不是使用全名作为数据库的名称。您还可以定制这个工厂，当其发现名称中包含Context字符串时将删除它。  
加入一个CustomConnectionFactory类到DataAccess项目（代码6-9）。

using System.Data.Common;

using System.Data.Entity.Infrastructure;

using System.Data.SqlClient;

using System.Linq;

namespace DataAccess

{

public class CustomConnectionFactory : IDbConnectionFactory

{

public DbConnection CreateConnection(

string nameOrConnectionString)

{

var name = nameOrConnectionString

.Split('.').Last()

.Replace("Context", string.Empty);

var builder = new SqlConnectionStringBuilder

{

DataSource = @".\SQLEXPRESS",

InitialCatalog = name,

IntegratedSecurity = true,

MultipleActiveResultSets = true

};

return new SqlConnection(builder.ToString());

}

}

}

CustomConnectionFactory使用Split方法取得上下文的名称的最后一段（以“.”划分）作为数据库名称。然后，它将Context字符串替换为空字符串字（如果有的话）。然后，它使用SqlConnection的StringBuilder创建一个连接字符串，将其用于构造一个SqlConnection。  
有了这个方法，你可以修改Main方法，使用刚刚创建的自定义连接工厂（例6-10）。这样就DefaultConnectionFactory或其他代码之前，让上下文设置使用自定义的ConnectionFactory。

static void Main()

{

Database.SetInitializer(

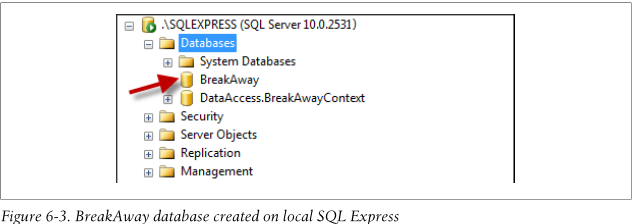
new DropCreateDatabaseIfModelChanges<BreakAwayContext>());

Database.DefaultConnectionFactory = new CustomConnectionFactory();

InsertDestination();

}

运行程序你会在SQL Express实例中发现一个新“BreakAway”数据库创建了。定制工厂已经替你将数据库名的名称空间和后缀Context删除。

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/201201051545271606.png)

## 数据库初始化

在第2章，你已经学习到可以使用Database.SetInitializer方法来为上下文类型设置初始化。设置初始化器可以清除并在模型变化时重建数据库：

Database.SetInitializer(

new DropCreateDatabaseIfModelChanges<BreakAwayContext>());

初始化包括两个主要步骤。首先，使用Code First在内存中根据默认规则和配置创建模型。其次，使用已设置的数据库初始化器将用于存储数据的数据库初始化。默认情况下，这个初始化将使用Code First创建一个数据库架构的模型。初始化会发生在每一个.NET Framework应用程序的实例上。应用程序的实例也被称为一个AppDomain。当上下文被使用时，初始化第一次被引发。初始化是延迟加载的，所以创建一个实例的是不完全满足初始化发生的条件的。必须执行对模型的操作，如查询或添加实体才会发生。  
初始化过程是线程安全的，所以在同一AppDomain中的多个线程可以使用相同的上下文类型。 DbContext本身不是线程安全的，因此，必须只能在一个单独的线程中使用一个给定的上下文类型实例。

### 在数据库初始化产生时进行控制

有的情况下，您可能希望控制初始化的发生，而不是让它自动发生在应用程序实例中第一次使用上下文对象时。初始化可以使用DbContext.Database.Initialize方法触发。这个方法接受一个名为force的布尔参数。该参数为false将导致初始化只发生在尚未在当前AppDomain触发的情况。请记住，每个AppDomain运行初始化一次，就会执行默认行为一次。force设置为true时将会使初始化过程运行，即使它已经在当前AppDomain发生。因为上下文也触发初始化，此代码需要运行在上下文被AppDomain使用之前。  
为什么你想手动触发数据库初始化？您可能需要通过手动触发初始化，使模型的创建和数据库初始化过程中发生的任何错误可以被捕获，并在一个地方处理。强制初始化发生的另一个原因是为了前端加载大型和/或复杂的模型。  
让我们来看看这个行为。修改Main方法，在代码中加入强制数据库初始化的配置，处理模型构建时发生的任何异常（代码6-11）。

static void Main()

{

Database.SetInitializer(

new DropCreateDatabaseIfModelChanges<BreakAwayContext>());

using (var context = new BreakAwayContext())

{

try

{

context.Database.Initialize(force: false);

}

catch (Exception ex)

{

Console.WriteLine("Initialization Failed...");

Console.WriteLine(ex.Message);

}

}

}

现在我们做些更改以使用初始化失败。这个错误发生在Code First映射一个数值属性到字符串列中。这样做会使模型创建失败发生在试图创建数据库构架之前。

修改Activity类加入一个Data Annotations的Column特性标记指定AcitivityId属性使用varchar数据类型。（代码6-12）

public class Activity

{

[Column(TypeName = "varchar")]

public int ActivityId { get; set; }

[Required, MaxLength(50)]

public string Name { get; set; }

public List<Trip> Trips { get; set; }

}

运行程序将显示异常信息，表示不能将整形数据映射到varchar类型：

移除刚刚添加到DestinationId上的特性标记，再次运行程序。这次就没有问题了。

### 关闭数据库初始化功能

当然，并不是所有场景都需要自动调用初始化，EF框架满足所有情况。例如，如果你映射到一个现有的数据库，可能在不能连接到数据库时需要让Code First发生错误而不是魔法般地创建一下。你可以通过传递一个null参数到Database.SetInitializer来关闭初始化功能:

Database.SetInitializer(null);

当初始化器被设置为null后,DbContext.Database.Initialize 仍可用于模型的创建过程.

### 将数据库初始化器包含在EF框架

你会发现，Database.SetInitializer接受IDatabaseInitializer<TContext>的一个实例。在EF框架中有三个针对此接口的实现。这些实现是抽象的，所以你可以从其中派生或自定义行为。后面我们将引导您逐步创建自己的实现。  
CreateDatabaseIfNotExists  
除非Database.SetInitializer指定了替代的初始化器,所有上下文都会被设置给默认初始化器。这是最安全的初始化，数据库将永远不会被自动删除，造成数据丢失。我们看到在第2章，如果模型是从数据库时创建后发生的改变，在初始化期间会抛出异常：

由于默认初始化器的存在,如果需要执行下列代码,你不需要做任何设置:

Database.SetInitializer(

new CreateDatabaseIfNotExists<BreakAwayContext>());

从前面几章你已经看到要确保数据库总是匹配当前的模型.如果Code First检测到二者不匹配,数据库就人被删除并且重新创建以便可以满足匹配关系.在开发时这很有用,但是显然不在在程序部署中使用,这样数的会丢失.我们已经看到这样的设置初始化的代码:

Database.SetInitializer(

new DropCreateDatabaseIfModelChanges<BreakAwayContext>());

这一初始化器将不管模型与数据库匹配与否都会删除和重建数据库.你可能会疑惑为什么要这么做.如果你集成测试的整个应用程序,就会需要在运行测试前将数据库重置到一个已知的状态.修改Main方法(代码6-13),运行一些代表测试的代码,这会在应用程序中插入一个Destinaion.

static void Main()

{

Database.SetInitializer(

new DropCreateDatabaseAlways<BreakAwayContext>());

RunTest();

}

private static void RunTest()

{

using (var context = new BreakAwayContext())

{

context.Destinations.Add(new Destination { Name = "Fiji" });

context.SaveChanges();

}

using (var context = new BreakAwayContext())

{

if (context.Destinations.Count() == 1)

{

Console.WriteLine(

"Test Passed: 1 destination saved to database");

}

else

{

Console.WriteLine(

"Test Failed: {0} destinations saved to database",

context.Destinations.Count());

}

}

}

由于初始化器被设置为每次都删除并重建,你会知道在测试开始前数据库是空的.你不用总是去考虑在运行测试前数据库是否为空,后而我们会看到放置一些种子数据在里面的例子.运行程序,我们会看到测试通过.就目前为止我们只执行一个单一的测试,通常一个应用程序里面需要进行多个测试.更新Main方法以便使其可以在一行里运行两次测试(代码6-14):

static void Main(string[] args)

{

Database.SetInitializer(

new DropCreateDatabaseAlways<BreakAwayContext>());

RunTest();

RunTest();

}

运行程序,你会看到第一个方法通过而第二个失败,表明数据库中有两个destinations.第二个测试失败的原因是第一次执行的结果已经在数据库中了.进一步的原因是AppDomaing默认情况每次程序运行只执行一次初始化.

本章前面介绍可以使用Database.Initialize强制初始化,不管当前的AppDomain是否已经初始化过.修改RunTest方法包含一个调用Database.Initialize强制初始化的方法确保每次测试前都会重置数据库(代码6-15),再次运行程序你会发现测方式现在通过了在每次测试执行前.数据库先删除又以已知的状态进行重建.

static void RunTest()

{

using (var context = new BreakAwayContext())

{

context.Database.Initialize(force: true);

context.Destinations.Add(new Destination { Name = "Fiji" });

context.SaveChanges();

}

using (var context = new BreakAwayContext())

{

if (context.Destinations.Count() == 1)

{

Console.WriteLine(

"Test Passed: 1 destination saved to database");

}

else

{

Console.WriteLine(

"Test Failed: {0} destinations saved to database",

context.Destinations.Count());

}

}

}

删除和重建数据库是将数据库状态保持在一个已知状态的很容易的方法,但是如果运行一系集成的测试,系统开销过大.考虑使用System.Transactions.TransactionScope作为避免在每一次测试时永久存储对数据库的修改.

### 创建一个定制的数据库初始化器

到目前为止,我们一直在使用EF框架中包含的初始化器.有时不想按照已有的初始化器的逻辑进行工作.Database.SetInitializer 接受IDatabaseInitializer 接口,你可以通过实现这个接口来定制逻辑.

小贴士:除了自已写定制的初始化器,也可以引用别人创建的.有一个例子EFCodeFirst.CreateTablesOnly NuGet 包.这个初始化器允许你在已经存在的数据库进行删除和创建操作,而不是删除和创建数据库实体本身.当你将数据库指向一个宿主数据库而又没有权限删除和创建整个数据库时特别有用.

你想实现自己的初始化器的原因可能有很多。我们来看一个简单的场景:在数据库删除并重新创建之前给开发者一个提示。Database属性暴露了各种方法与数据库进行交互,可以实现检查是否存在，是否创建，或是否删除等功能。API中包含的初始化器包含的逻辑利用了这些方法。在你自己的类，你也可以将这些方法整合在逻辑里。这就是下面这个例子要做的。添加PromptForDropCreateDatabaseWhenModelChages类到您的DataAccess项目（例6-16）。

using System;

using System.Data.Entity;

namespace DataAccess

{

public class PromptForDropCreateDatabaseWhenModelChages<TContext>

: IDatabaseInitializer<TContext>

where TContext : DbContext

{

public void InitializeDatabase(TContext context)

{

// If the database exists and matches the model

// there is nothing to do

var exists = context.Database.Exists();

if (exists && context.Database.CompatibleWithModel(true))

{

return;

}

// If the database exists and doesn't match the model

// then prompt for input

if (exists)

{

Console.WriteLine

("Existing database doesn't match the model!");

Console.Write

("Do you want to drop and create the database? (Y/N): ");

var res = Console.ReadKey();

Console.WriteLine();

if (!String.Equals(

"Y",

res.KeyChar.ToString(),

StringComparison.OrdinalIgnoreCase))

{

return;

}

context.Database.Delete();

}

// Database either didn't exist or it didn't match

// the model and the user chose to delete it

context.Database.Create();

}

}

}

PromptForDropCreateDatabaseWhenModelChages类实现单一的InitializeDatabase方法。首先，它检查数据库是否存在以及是否与当前的模型相匹配。如果是这样，什么也不做,初始化器返回。如果该数据库存在，但不匹配当前的模型，会提示你是否想删除和创建数据库。如果您决定不重新创建数据库，初始化器返顺,EF框架将尝试按现有的数据库模式再次运行。如果您决定重新创建数据库，现有的数据库将被删除。最后一行代码简单地创建数据库中，只会在数据库不存在，或者我们选择重新创建数据库才会得到执行。  
自定义的初始器在需要在EF框架内注册;修改Main方法（例6-17）。你会注意到Main方法调用了我们在第2章更新的InsertDestination方法:

static void Main()

{

Database.SetInitializer(new

PromptForDropCreateDatabaseWhenModelChages<BreakAwayContext>());

InsertDestination();

}

我们对模型作些修改使其不再与数据库匹配.修改Destinaton类中的Name属性,在其上附加一个Data Annotations标记:MaxLength.

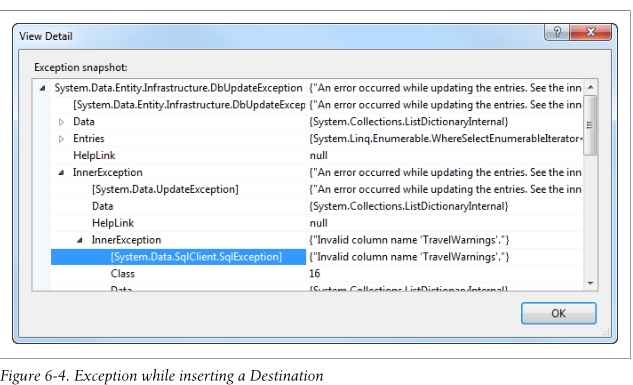
[MaxLength(200)]

public string Name { get; set; }

现在运行的应用程序，将提示您，询问您是否要删除并创建数据库。答否（N），告诉我们的自定义初始化器不理会数据库。你会注意到，应用程序仍然成功完成。这是因为您所做的更改不会阻止EF框架使用当前模型访问过时的数据库架构。EF框架预期Destination Names应为200个字符或更少。由于数据库没有改变，它没有强制执行的最大长度，所以EF框架给它发送的INSERT语句可以执行。  
现在，让我们做出改变，会影响INSERT语句。修改目标类，包括一个新的TravelWarnings属性：

public string TravelWarnings { get; set; }

再次运行应用程序。像以前一样，你会被提示，询问您是否要删除并创建数据库。选择不创建数据库，这个时候你会得到一个DbUpdateException。你需要通过内部异常链去找到错误的真正原因（图6-4）。  
顶层异常的内部异常是UpdateException，该内部异常是一个SQLException。最后的SQLException的消息，说明发生了什么：“无效的列名称”TravelWarnings，“问题发生的原因是EF框架试图执行示例6-18中所示的SQL语句，但TravelWarnings列在数据库中不存在。

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/201201051545282935.png)

insert [dbo].[Destinations]([Name], [Country], [Description],

[TravelWarnings], [Photo])

values (@0, @1, @2, null, null)

select [DestinationId]

from [dbo].[Destinations]

where @@ROWCOUNT > 0 and [DestinationId] = scope\_identity()

再次运行程序,这次选择删除并重建数据库,程序成功执行.

### 在配置文件中设置数据库初始化器

在代码中设置的初始化是一种简单的方法，但部署用程序时，您可能希望有一个更简单的方式设置，而无需修改代码。想要应用程序部署设置DropCreateDatabaseIfModelChanges的初始化器，这是极不可能的！将appSettings节添加到BreakAwayConsole项目的配置文件中，其中包括了初始化器的设置,见示例6-19中所示。  
Example 6-19. Initializer set in configuration file

<?xml version="1.0"?>

<configuration>

<appSettings>

<add key="DatabaseInitializerForType DataAccess.BreakAwayContext, DataAccess"

value="System.Data.Entity.DropCreateDatabaseIfModelChanges`1

[[DataAccess.BreakAwayContext, DataAccess]], EntityFramework" />

</appSettings>

</configuration>

小贴士:代码示例有一行断裂的代码,在实际的App.config文件中应该删除.value值必须在同一行才能工作.

还应有很多配置行，我们打破配置结构来分别研究。关键部分开始于DatabaseInitializerForType，后跟一个空格，然后配置正确的上下文名称以便初始化器能够为其设置。在我们的例子就是DataAccess.BreakAwayContext，DataAccess，仅仅意味着DataAccess.BreakAwayContext类型的定义是在DataAccess程序集。Value部分是配置数据库初始化器要使用名称。它看起来复杂，因为我们使用泛型类型，我们使用了EF框架程序集中定义的DropCreateDatabaseIfModelChanges<BreakAwayContext> 方法来进行设置。  
还需要修改Main方法，以便它不再设置在代码中的初始化：

static void Main()

{

InsertDestination();

}

现在对模型作些改变以便测试配置文件是否得到应用.修改Destination类包含一个新的ClimateInfor属性:

public string ClimateInfo { get; set; }

运行程序,你会看到数据库被删除重建,新增ClimateInfo列,现在，如果你要部署你的应用程序，你可能变更初始化器为CreateDatabaseIfNotExists，以便永远不会导致数据丢失。您也可能工作在别人为您创建的DBA上,如果数据库在Code First工作流之外创建，你会想禁用数据库的初始化。你可以通过改变配置文件来指定初始化器的禁用(代码6-20）.

<?xml version="1.0"?>

<configuration>

<appSettings>

<add key="DatabaseInitializerForType DataAccess.BreakAwayContext, DataAccess"

value="Disabled" />

</appSettings>

</configuration>

现在我探索了有关在配置文件设置初始化器的方法,请移除已经添加的任何设置.

数据库数据库初始化器添加种子数据在本章中，你已经看到数据库的初始化可以被用来控制Code First何时以及如何创建数据库。到目前为止，Code First创建的数据库一直是空的，但也有一些需要Code First创建一些种子数据的情况。您可能有一些预定义的数据，如性别或国家的查找表。或者你可能只是想在本地工作时,在数据库中放一些示例数据，从而可以看到应用程序的行为。  
种子数据可以用另一种情况是运行集成测试。在上一节中，我们写了一个测试，依靠的是一个空的数据库，现在让我们进行一个依赖于包含一些已知数据的数据库的测试。  
让我们开始写要运行的测试。修改Main方法来运行测试，以验证“Great Barrier Reef”是否为数据库中的Destination条目（例6-21）.确保您已经删除在上一节添加到任何设置的配置文件。

static void Main()

{

Database.SetInitializer(

new DropCreateDatabaseAlways<BreakAwayContext>());

GreatBarrierReefTest();

}

static void GreatBarrierReefTest()

{

using (var context = new BreakAwayContext())

{

var reef = from destination in context.Destinations

where destination.Name == "Great Barrier Reef"

select destination;

if (reef.Count() == 1)

{

Console.WriteLine(

"Test Passed: 1 'Great Barrier Reef' destination found");

}

else

{

Console.WriteLine(

"Test Failed: {0} 'Great Barrier Reef' destinations found",

context.Destinations.Count());

}

}

}

运行应用程序，你会看到测试失败，说明“Great Barrier Reef”在数据库中没有任何条目与之匹配。这是有道理的，因为你设置了DropCreateDatabaseAlways初始化，这将创建和清空数据库。  
测试真正需要的是，在创建了数据库后，将插入一些种子数据,能够DropCreateDatabaseAlways的变化来实现。包括在EF框架中的三个初始化器不是sealed的，这意味着你可以通过派生其中之一来创建自己的初始化器。所有三个初始化器都包括一个名为Seed的abstract方法（在Visual Basic中为Overridable），这意味着它可以被覆写。Seed方法有一个空的实现，但是，初始化器可以在适当的时候插入您提供的种子数据。  
要检查此功能，您的DataAccess项目添加DropCreateBreakAwayWithSeedData类。提供种子数据的关键是覆写初始化种子的方法，如例6-22所示。

using System.Data.Entity;

using Model;

namespace DataAccess

{

public class DropCreateBreakAwayWithSeedData :

DropCreateDatabaseAlways<BreakAwayContext>

{

protected override void Seed(BreakAwayContext context)

{

context.Destinations.Add(new Destination

{ Name = "Great Barrier Reef" });

context.Destinations.Add(new Destination

{ Name = "Grand Canyon" });

}

}

}

小贴士:注意我们在这里没有调用SaveChanges方法.Seed的基方法会在定制方法之后调用.如果你让VS的编辑器自动实现覆写方法,就会包括一个对base.Seed(context)的调用.你可以不去管他,但是记住要将这行代码放在方法的最后一行.

现在你已经创建了一个能够插入种子数据的初始化器,它需要在EF框架中注册后才能被使用.这可以以我们在前面包含初始化器相同的方式进行--通过Database.SetInitializer方法

修改Main方法以注册DropCreateBreakAwayWithSeedData类:  
static void Main()

{

Database.SetInitializer(new DropCreateBreakAwayWithSeedData());

GreatBarrierReefTest();

}

再次运行应用程序，本次测试通过，因为Code First现在使用DropCreateBreakAwayWithSeedData初始化数据库。由于此初始化派生自DropCreateDatabaseAlways，它会删除数据库并重新创建一个空数据库。覆写的Seed方法，随后被调用，您指定的种子数据插入到了新创建的数据库里。

代码6-24中的Seed方法或许有些简单化,但这让我们很好地观察了这个方法的创建过程。可以插入各类数据及相关数据。例如使用一个有效率的LINQ方法利用Code First检查我的博客,将相关文章的全部图片作为种子插入数据库。

## 使用数据库初始化进一步影响数据库构架

除了使用Code First在数据库中创建种子数据以外,你也可不使用配置或种子数据达到相同目的.你如,你可以想创建Lodgings表中Name字段的索引以加快使用name查询的速度.

你可以通过调用DbContext.Database.ExecuteSqlCommand 方法来达到目的,这个方法会在Seed方法内部构造创建索引的SQL语句.代码6-24显示了对Seed方法的修改,强制数据插入时创建索引.

protected override void Seed(BreakAwayContext context)

{

context.Database.ExecuteSqlCommand

("CREATE INDEX IX\_Lodgings\_Name ON Lodgings (Name)");

context.Destinations.Add(new Destination

{ Name = "Great Barrier Reef" });

context.Destinations.Add(new Destination

{ Name = "Grand Canyon" });

}

## 小结

在这一章中，你看到了默认情况Code First如何与数据库进行交互，也学习到如何覆写此默认行为。你已经学会了如何控制数据库，Code First连接到数据库时是如何初始化的。您还学到如何将数据库的初始化用于情景测试中,如何在初始化时插入种子数据。  
在这本书中，你已经看到Code First根据您的域类和配置创建了一个模型，然后，你也看到Code First是如何定义和初始化被模型用来访问的数据库。在下一章中，您将学习一些不太常用的先进的理念，但这些理念有时会很有用。

# 第七章：高级概念

到目前为止你已经看到Code First的建模功能应足以让你和运行大多数应用。然而，Code First还包括一些更高级的功能，您可能有需要。在这本书中，您已经看到Code First的默认行为，但如果有一个或多个你不喜欢的约定，Code First允许你删除它们。您可能还希望得到摆脱EdmMetadata表添加到您的数据库。默认情况下Code First需要缓存模型，您可以覆写这种行为来解决类似在一个应用程序中指向多个数据库的这种情况。本章将涵盖更多的这些主题。

## 映射到非表数据库对象

您已经学习了使用Code First映射到表，无论是生成一个数据库还是映射到一个现有的数据库中的表，都是针对有表的数据库进行。但数据库支持许多其他类型的对象，包括存储过程和视图。  
对EF框架4.2而言，Code First只内置表支持，这意味着它只能生产生包含表的构架。因此，如果您  
使用Code First生成数据库，就被限制在使用表工作上面。  
不过，如果你是映射到一个现有的数据库，可能有视图，存储过程，并映射到数据库中的其他对象。让我们看看如何与这些对象进行交互。

小贴士：你可以选择在Code First生成数据库构架后手工编辑数据库来包含非表对象。在本节中你就可以使用这样的技术。

EF框架开发团队声称他们已经计划在未来的版本技持非表数据库对象的映射。

## 映射到可更新的视图

在某些情况下，你可能想简单地将实体映射到视图，而不是表。例如，您可能需要映射到一个数据库，它有一个非常大的和混乱的构架。为了简单，数据库可能包含一个视图，使用相比实体更容易理解的列名暴露数据。如果视图是可更新的，您可以使用EF框架来插入，更新和删除数据以及选择数据。幸运的是，大多数数据库，包括SQL Server，使用相同的SQL语法与视图交互，就好像与表格进行交互一样。这意味着你可以简单地“欺骗[”Code First](mailto:%E2%80%9D@CF)，并告诉它那个视图就是一个表。您可以通过使用命名表中相同的配置的方法来控制它。

小贴士：不懂可更新视图？请到MSDN中的 SQL Server’s “CREATE VIEW (Transact-SQL)”去看看，会有所帮助。

例如，假设有一个Destination数据是来自于一个可更新的视图，名为my\_destination\_view。你可以使用Table标记为其指定视图名称：

[Table("my\_detination\_view")]

public class Destination

你也可能使用Fluent API的ToTable方法来映射到视图：

modelBuilder.Entity<Destination>().ToTable("my\_detination\_view");

## 使用视图填充对象

并非所有的情况下都可调用实体直接映射到一个可更新的视图。在某些特殊场景里，如果一个类已经映射到一个表，而需要视图能够来调用一组这样的类。举个例子，假设你已经让Destination表映射到了Destinations表，而在您的应用程序的一个区域里需要从TopTenDestination视图中加载所有的destinations。您就可以使用DbSet的SqlQuery方法来加载你写的一些以SQL为基础的实体：

var destinations = context.Destinations

.SqlQuery("SELECT \* FROM dbo.TopTenDestinations");

在上面的代码中，我们使用一个SQL语句，绕过实体框架取回所需的目标对象。好消息是，一旦这些对象从数据库中检索，它们被视为与其他方式加载的对象相同。这意味着你仍然能为加载的目标对象获得变化跟踪，延迟加载，和其他DbContext功能。

SqlQuery函数的方法依赖于在查询结果集的列名和对象属性名的精确匹配。由于目标类包含DestinationId，Name和其他属性，视图必须返回与其相同名称的列。如果视图没有与类属性相同的列名，需要在SELECT语句中的为列设置别名。  
例如，TopTenDestinations视图使用Id而不是DestinationId作为主键的名称。在SQL Server中，可以使用AS关键字为Id列指定DestinationId列的别名，满足@EF的要求，代码7-1对此进行了解释：

var destinations = context.Destinations

.SqlQuery(@"SELECT

Id AS DestinationId,

Name,

Country,

Description,

Photo

FROM dbo.TopTenDestinations");

请注意，列与属性的名称匹配并未考虑任何到帐户的映射。例如，如果你映射DestinationId属性到Desintaions表中的Id列，SqlQuery方法不会使用这种映射。 sqlquery函数的方法总是尝试其于属性名来匹配列-属性关系。因此，仍然需要在结果集中将此列称为DestinationId。

## 使用视图来填充非模型对象

我们已经了解了两个技术，都可以使用视图来填充一个对象，成为模型的一部分。一旦这些对象被创建，将会被上下文跟踪，任何更改都会写回数据库。您可能会发现视图的结果返回的是只读的对象集。视图的结果可能会将多个表中的数据结合起来，因此不能直接映射到一个作为模型一部分的实体中。  
例如，你可能有一个视图，命名为DestinationSummaryView，组合了Destinations和Lodgings表的数据。这个视图可能有DestinationId，Name，LodgingCount，和ResortCount列。这些列不匹配任何在BAGA模型内的实体，但它能够将结果返回一个专用的对象，你可以在程序中自由使用。

DestinationSummary类类似代码7-2所示：

public class DestinationSummary

{

public int DestinationId { get; set; }

public string Name { get; set; }

public int LodgingCount { get; set; }

public int ResortCount { get; set; }

}

由于此类不是BAGA模型的一部分，不能使用DbSet来查询结果。相反，需要使用SqlQuery方法查询DbContext.Database：

var summary = context.Database.SqlQuery<DestinationSummary>(

"SELECT \* FROM dbo.DestinationSummaryView");

作为回应，EF框架将运行由您提供的SQL访问DestinationSummaryView视图。然后将这些结果尝试使用DestinationSummary类的属性名匹配您在SqlQuery方法的范型参数指定的列名。因为列名和属性名相匹配，我们可以从DestinationSummary对象的集合中得到的查询结果。  
在例7-1中，我们并没有通过DbSet创建DestinationSummary对象，因此不会被上下文跟踪到。因此，如果你改变任何属性，EF框架将不会关注任何在调用SaveChanges时发生的任何变化。

## 使用存储过程

Code First并不支持直接通过存储过程来映射插入，更新和删除命令，但是你可以在设计器中使用这些命令。

小贴士：EF框架开发团队已经接到了大量的用户要求支持此功能，声明在后续版本中可能会加入支持。

使用你刚刚在视图中同样的技术，也可使用存储过程从数据库中生成结果。例如，有一个名为TopTenDestinations 的存储过程通过一个单一的参数来指定在哪个国家内查找目的地。你可以使用DbSet的SqlQuery方法来生成这个过程：

var country = "Australia";

var destinations = context.Destinations

.SqlQuery("dbo.GetTopTenDestinations @p0", country);

注意到SqlQuery接受一个参数，见后面“使用SqlQuery参数避免SQL注入”获取更多信息。

与视图类似，你可以使用DbContext.Database.SqlQuery方法获得存储过程返回的数据，这些存储过程并不与模型中的实体相匹配。假设有一个GetDestinationSummary存储过程，想要获取的结果是DestinationSummary类的集合（代码7-2）。现在给存储过程提供两个参数，一个为country，另一个为某个关键词：

var country = "Australia";

var keyWords = "Beach, Sun";

var destinations = context.Database.SqlQuery<DestinationSummary>(

"dbo.GetDestinationSummary @p0, @p1", country, keyWords);

在上述代码中，我们为参数使用了索引名。@EF将会封装这些参数为DbParameter对象以避免任何SQL注入问题。由存储过程返回的列名将会匹配DestinationSummary的属性名。由于DestinationSummary并不是BAGA模型的一部分，结果不会被跟踪，任何更改也不会返回到数据库中。

小贴士：使用SqlQuery参数防止SQL注入

SqlQuery方法允许指定参数。EF框架将封装参数为DbParameter对象以防止SQL注入式攻击。使用对参数@p前缀后附一个整数索引号来表达。EF框架将以这些索引号匹配在查询字符串中提供的参数值。正如你在前面看到的基于视图的案例，查询结果可以被上下文所跟踪，像其他查询结果一样使用。

## 删除默认规则

从前面的章节中可以看到，Code First包含了一系列的约定帮助你创建模型。你也看到如何设置或覆写默认规则，可以使用Data Annoations，也可使用@FA。还有一个选项可以关闭一个或多个默认约定。

每个Code First的默认约定都被配置为一个类，这个类位于System.Data.Entity.ModelConfiguration.Conventions 名称空间。Code First目前只允许移除一个多个包含的默认规则。

小贴士：Code First的预览版本支持编写自己的“约定”代码。但是，EF框架团队在正式版取消了此功能，因为他们认为他们没有时间去完善并不是太过令信期待的功能。很可能在未来的版本会再次提供此功能。

有关类列表取自微软网站，<http://msdn.microsoft.com/zh-cn/library/gg696316(v=VS.103).aspx>，目前尚无中文版本。

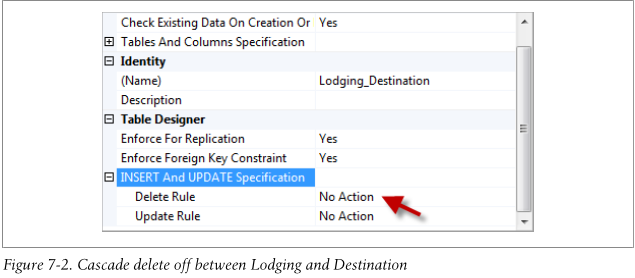
我们可以移除上述列表中的任何一个默认规则。这里我们只尝试对OneToManyCascadeDelete进行移除作为展示。这个默认规则对所有必须的关系创建级联删除。

当然可以覆写每个必须关系的级联行为。但是如有很多关系，可能需要同时禁用默认规则。

关闭默认规则在context的OnModelCreating方法内进行，通过调用DbModelBuilder.Conventions.Remove方法就可以实现。在BreakAwayContext类中的OnmodelCrating方法加入如下代码：

modelBuilder.Conventions.Remove<OneToManyCascadeDeleteConvention>();

在模型中有一个必须的关系：Lodging与Destination.当前，Code First自动为这个关系添加了一个级联删除。使用新代码后，运行程序，级联删除就从模型和数据库中被移除了（见图7-2）。任何其他必须关系的级联删除都被去除了。

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/201201062025519901.png)

关闭默认约定后，你可能还想在许多关系中重新引入级联删除行为。你可以使用Fluent API来进行设置。（见第4章）

在BAGA模型中，需要在必须的关系中具有级联删除功能，请删除上述代码，恢复  
OneToManyCascadeDeleteConvention默认规则。

## 控制模型缓存

在这本书中，你已经看到Code First管理了很多东西，但是，你可以在需要控制它，改变它的行为。模型缓存也不例外，事实上，你可能没想到，Code First为你进行了模型的缓存。在扫描类并使用默认约定与配置后，Code First就会使模型的一个版本保持在内存中，以便可以被应用程序的实例重用。这是因为应用程序实例的每一个Dbcontext只调用一次OnModelCreating方法。在本节中，您将学习模型缓存是什么，如果需要重写约定，应如何去做。

## 理解模型缓存

在前面的章节中，你已经看到，Code First会自动发现和建立一个模型，这个模型基于你在context中暴露的DbSet属性而创建。模型创建过程中涉及到提取实体类型设置，配置Code First约定以及启用任何由@DA或@FA实施的附加配置。这个过程很耗费资源，可能会需要一些时间，尤其是在模型很大和/或复杂的时候。为了避免每次创建上下文的一个实例所需的开销过大，Code First每运行一次这个过程，就会缓存上下文类型的最后一个模型。模型缓存发生在AppDomain级别。  
通过在context中调用OnModelCreating方法可以看到模型缓存行为，在OnModelCreating方法中添加一行如下的代码，只要被调用就会写入控制台：  
Console.WriteLine("OnModelCreating called!");  
2.

  修改Main方法调用InsertDestination方法一定次数（代码7-3）。InsertDestination方法见第2章。

static void Main()

{

Database.SetInitializer(

new DropCreateDatabaseIfModelChanges<BreakAwayContext>());

InsertDestination();

InsertDestination();

InsertDestination();

}

运行程序，你会发现尽管代码构造和使用了三个BreakAwayContext实例，OnModelCrating方法只被调用了一次。这是因为Code First只有当第一次context实例创建模型时和进行调用；在那以后，最后一个模型就被加入缓存，在后面的BreakAwayContext需要时被重用。

## 覆写默认的模型缓存

有许多情况需要控制模式缓存。只要AppDomain中的每一个实例中上下文类型的模型保持相同，默认行为就会按预期方式工作。使用默认行为，可以获得最好的性能，因为模型的创建将只发生一次。  
有某些情况下，给定的上下文类型的模型，在同一AppDomain中实例之间可能会有所不同。一个例子是使用多租赁数据库。多租赁数据库会涉及到在同一物理数据库具有相同的模式的实例会复制多闪。例如，可能有一个模型用于存存储blog文章，同时也用于在网站上予以显示。您的网站可能包含个人和工作博客，都使用相同的模型。在数据库中你可能将这一模型复制存付两个单独构架中的表中。工作博客使用的表可能位于work构架中（如work.Posts，work.Comments等），而个人博客可能位于personal构架中（如personal.Posts,personal.Comments等）。这些表被称为租客。数据库模式只是一种区分租户的方法;还有许多其他的模式，如表前缀，。  
如果您的应用程序需要访问相同的AppDomain多个租户，类和表之间的映射将是不同的，取决于你的目标租户。不同的映射意味着不同的模型，这反过来又意味着默认模型缓存不会为你工作。

另一个例子是同一AppDomain内，使用相同的上下文指向不同数据库引擎的相同模型。不同的数据库引擎意味着数据库列中有不同的数据类型，而这又意味着不同的模型。让我们看看在这种情况下，如何处理模型缓存。  
添加例7-4中所示的TargetMultipleProviders方法。此方法使用  
同样的上下文访问SQL Server和SQL Server Compact Edition数据库。

小贴士：

完成本节需要安装SQL ServerCompact Edition运行时。如果您在第6章，已经安装了运行时，请继续。如果没有，请参阅前述的“安装SQL Server Compact Edition”。您可能还记得，在第6章中，我们不得不改变我们的模型，指向SQL Compact Edition。如果你想测试一下这个代码，你需要在这里再次作出同样的变化。早在第3章，我们配置数据库生成的键Trip.Identifier标识符是一个GUID属性，使用在SQL Server上没有任何问题。 SQL Compact不能产生的GUID列的值。的如果你想运行应用程序，删除数据库生成Trip.Identifier的Data Annoations或Fluent API配置代码。

static void Main(string[] args)

{

Database.SetInitializer(new

DropCreateDatabaseIfModelChanges<BreakAwayContext>());

TargetMultipleProviders();

}

private static void TargetMultipleProviders()

{

var sqlString = @"Server=.\SQLEXPRESS;

Database=DataAccess.BreakAwayContext;

Trusted\_Connection=true";

using (var connection = new SqlConnection(sqlString))

{

using (var context = new BreakAwayContext(connection))

{

context.Destinations.Add(new Destination { Name = "Hawaii" });

context.SaveChanges();

}

}

var sqlCeString =

@"Data Source=|AppData|\DataAccess.BreakAwayContext.sdf";

using (var connection = new SqlCeConnection(sqlCeString))

{

using (var context = new BreakAwayContext(connection))

{

context.Destinations.Add(new Destination { Name = "Hawaii" });

context.SaveChanges();

}

}

}

运行程序，在试图使用上下文实例指向SQL Server Compact Edition时会得到一个异常。这是一个NotSupportedException，言明：“不支持在不同类型的数据服务中使用同一DbCompiledModel创建contexts.可以为每种服务器创建一个单独的context类型”。

为了在同一AppDomain中使用同一context指向不同的模型，你需要为每个模型构建一个DbCompiledModel，然后使用这个模型构建不同的上下文实例。DbContext暴露了一系构造器允许你支持模型这样使用。添加一个构造器到BreakAwayContext类中，允许支持DbCompiledModel 和DbConnection：

public BreakAwayContext(DbConnection connection,

DbCompiledModel model)

: base(connection, model, contextOwnsConnection: false)

{ }

代码7-5显示了一个更新的TargetMultipleProviders方法展示如何利用这个构造器来指向不同的数据库引擎，为各自使用不同的模型：

Example 7-5. Code updated to work with multiple providers

private static void TargetMultipleProviders()

{

var sql\_model = GetBuilder().Build(

new DbProviderInfo("System.Data.SqlClient", "2008"))

.Compile();

var ce\_model = GetBuilder().Build(

new DbProviderInfo("System.Data.SqlServerCe.4.0", "4.0"))

.Compile();

var sql\_cstr = @"Server=.\SQLEXPRESS;

Database=DataAccess.BreakAwayContext;

Trusted\_Connection=true";

using (var connection = new SqlConnection(sql\_cstr))

{

using (var context =

new BreakAwayContext(connection, sql\_model))

{

context.Destinations.Add(new Destination { Name = "Hawaii" });

context.SaveChanges();

}

}

var ce\_cstr =

@"Data Source=|DataDirectory|\DataAccess.BreakAwayContext.sdf";

using (var connection = new SqlCeConnection(ce\_cstr))

{

using (var context = new BreakAwayContext(connection, ce\_model))

{

context.Database.Initialize(force: true);

context.Destinations.Add(new Destination { Name = "Hawaii" });

context.SaveChanges();

}

}

}

private static DbModelBuilder GetBuilder()

{

var builder = new DbModelBuilder();

builder.Entity<EdmMetadata>().ToTable("EdmMetadata");

builder.Entity<Activity>();

builder.Entity<Destination>();

builder.Entity<Hostel>();

builder.Entity<InternetSpecial>();

builder.Entity<Lodging>();

builder.Entity<Person>();

builder.Entity<PersonPhoto>();

builder.Entity<Reservation>();

builder.Entity<Resort>();

builder.Entity<Trip>();

builder.ComplexType<Address>();

builder.ComplexType<Measurement>();

builder.ComplexType<PersonalInfo>();

return builder;

}

我们来看看TargetMultipleProviders方法中的代码做了什么。 GetBuilder方法负责创建一个DbModelBuilder，使用builder注册所有的类。示例代码中的每个类都使用DbModelBuilder.Entity和DbModelBuilder.ComplexType方法进行了注册。这种方法将正常工作在已经使用Data Annoations配置过的类上。如果使用Fluent API，应该从OnModelCreating方法复制的代码来替换这部分代码。请注意，还需要包括EdmMetadata类，并将其映射到EdmMetadata表中，这使Code First可以到检测到模型和数据库是否同步。当由DbContext负责创建模型时，会对此类自动维护。

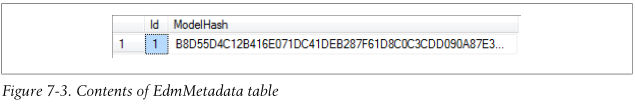
下一步要为两个需要指向的数据库引擎建立和编译模型。在这个例子中，不变的名称和数据库引擎令牌都提供给Build方法。作为一种替代方法，Builder的另一个重载，接受一个DbConnection获得数据库引擎信息。

在编译过的模型创建后，他们现在可以用来访问两个不同的数据库。请记住，数据库初始化只在每个AppDomain发生一次，所以只有第一个要使用的数据库将自动初始化。调用Database.Initialize是针对第二个数据库的，确保第二个数的也得到初始化。  
最后，新的Destination被添加到两个不同的数据库中，使用的是相同的类和相同的配置设置定义了重复的模型。现在，我们使用过SQL Compact了，重新启用以往配置，以使Trip.Identifier键由数据库生成。

小贴士：记住构建和编译模型是高开销的操作。结果编译模型应被缓存并且在同一方面的所有context实例中被重用。

## 使用EdmMetadata表

早在第2章中，您就学习到，默认情况下，Code First会将EdmMetadata表添加到您的数据库。允许Code First在数据库中创建这个表有一定的好处，但您也可以选择删除它。在本节中，你会看到如何从数据库中删除EdmMetadata表。您还可以了解删除它的影响。  
EdmMetadata表的存在只有一个目的，那就是存储被用来创建数据库的模型快照。快照允许Code First检查当前模型是否匹配当前数据库。快照以model数据库部分的SHA256哈希形式来存储。你可以看到在图7-3 EdmMetadata表总是包含一个存储有哈希值的单列。

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/201201062025527883.png)

## 使用EdmMetadata编码

Code First在包含的数据库初始化器中使用EdmMetadata表，但是你也可以使用编程方式与其交互，通过在@EFAPI中EdmMetadata类来实现。修改Main方法，调用一个新的UseEdmMetadataTable方法，如代码7-6所示：  
static void Main()

{

Database.SetInitializer(

new DropCreateDatabaseIfModelChanges<BreakAwayContext>());

UseEdmMetadataTable();

}

private static void UseEdmMetadataTable()

{

using (var context = new BreakAwayContext())

{

var modelHash = EdmMetadata.TryGetModelHash(context);

Console.WriteLine("Current Model Hash: {0}", modelHash);

var databaseHash =

context.Set<EdmMetadata>().Single().ModelHash;

Console.WriteLine("Current Database Hash: {0}", databaseHash);

var compatible =

context.Database.CompatibleWithModel(throwIfNoMetadata: true);

Console.WriteLine("Model Compatible With Database?: {0}",

compatible);

}

}

此代码开始使用静态的EdmMetadata.TryGetModelHash方法，找到当前模型的哈希值。此方法会始终工作在Code First模型中，但如果您尝试与设计器创建的模型进行工作，将返回null。 EdmMetadata类作为你的模型的一部分，所以你可以用你的DbContext与它交互。  
代码的第二部分创建了为EdmMetadata类创建了DbSet，然后索取一行数据以便可以读取哈希值。最后，还有一个DbContext.Database.bvgt65方法，它简单地检查模型和数据库是否匹配。这是包含在EF框架内部的数据库初始化器的方法。如果EdmMetadata表从数据库中已排除，为throwIfNoMetadata参数指定true将导致抛出一个异常。如果排除表，指定false将导致该方法返回false。您可以运行代码并看到一切当前的匹配。

### 在Code First中使用ObjectContext

至此，Code First都在使用DbContext API进行工作，这是Code First的推荐API。DbContext是一个轻型工具，在EF框架4.1中引入，封装了现有的其他EF框架组件，更具生产力。可以使用ObjectContext代替DbContext。本章，你会看到如何使用ObjcetContext来创建Code First模型。

小贴士：DbContext还是ObjectContext？

DbContext封装自ObjectContext，并对相关类进行了简化。如果你需要一些只有ObjectContext才能提供的高级功能 ，可以强制DbContext实现 IObjectContextAdapter接口访问底层的ObjectContext。这种方法允许您访问ObjectCon文本的功能，同时仍然能够对新的DbContext写你的代码。您可能会考虑使用ObjectContext的代码，首先，如果您有现有的，是基于ObjectContext和你交换首先从型号的第一或数据库代码的应用。

类似于基于DbContext的context，你可以开始于创建一个派生的context，只不过这一次我们是派生自ObjectContext，暴露ObjectSet的属性而不是DbSet的属性。在代码7-8就使用了ObjectContext，相比DbContext需要多写一点代码。必须暴露一个构造器接受一个EntityConnection.ObjcetSet也CreaeObjectSet方法进行初始化；这些DbContext都替您完成了。

添加一个新的BreakAwayObjectContext类到DataAccess项目。

using System.Data.EntityClient;

using System.Data.Objects;

using Model;

namespace DataAccess

{

public class BreakAwayObjectContext : ObjectContext

{

public BreakAwayObjectContext(EntityConnection connection)

: base(connection)

{

this.Destinations = this.CreateObjectSet<Destination>();

this.Lodgings = this.CreateObjectSet<Lodging>();

this.Trips = this.CreateObjectSet<Trip>();

this.People = this.CreateObjectSet<Person>();

this.PersonPhotos = this.CreateObjectSet<PersonPhoto>();

}

public ObjectSet<Destination> Destinations { get; private set; }

public ObjectSet<Lodging> Lodgings { get; private set; }

public ObjectSet<Trip> Trips { get; private set; }

public ObjectSet<Person> People { get; private set; }

public ObjectSet<PersonPhoto> PersonPhotos { get; private set; }

}

}

DbContext能够自动扫描DbSet属性并创建一个基于这些属性的模型。但是ObjectContext没有为Code First内建支持。Code First提供了一个方法来填补这个鸿沟，这个方法就是DbModelBuilder.UseObjectContext. 下面，我们来看如何使用这个方法在Code First模型中创建ObjectContext.

修改Main方法使用一个新的UseObjectContext方法，如代码7-9所示：

static void Main()

{

UseObjectContext();

}

private static void UseObjectContext()

{

var builder = GetBuilder();

var cstr = @"Server=.\SQLEXPRESS;

Database=BreakAwayObjectContext;

Trusted\_Connection=true";

using (var connection = new SqlConnection(cstr))

{

var model = builder.Build(connection).Compile();

using (var context =

model.CreateObjectContext<BreakAwayObjectContext>(connection))

{

if (!context.DatabaseExists())

{

context.CreateDatabase();

}

context.Destinations.AddObject(

new Destination { Name = "Ayers Rock" });

context.SaveChanges();

}

}

以往我们都是先创建DbModelBuilder，使用GetBuilder方法添加上下文。然后使用模型构建器创建模型并进行编译。注意在创建模型时必须提供数据库连接或引擎信息，因为数据库引擎类型影响最终结果数据的类型。使用编译过的模型，也可使用CreateObjectContext方法来构建ObjectContext。这个方法依赖于你暴露的构造器，这个构造器接受一个单一的实体连接参数。ObjectContext不支持数据库初始化器，因此必须编写代码检查数据库是否存在并在不存在的时候去创建它。注意ObjectContext不支持EdmMetadata，因此无法检测模型和数据库是否兼容。

## 小结

本章提供了Code First提供的一些高级特征，这些特性确保在默认行为不能满足您工作场景需要时能够覆写Code First的默认设置。大多数应用程序者不需要你使用这些特征，但是理解这些特征可以作为不时之需。

到目前为止我们已经全面介绍了Code First。我们开始于一个高级别的概览。然后学习了如何使用Code First构建模型，如何使用配置定制模型。还学习了如何影响Code First连接哪一个数据库，如何让数据库初始化。最后在本章中，我们讨论了一些高级话题。本书的最后一章将帮您为未来的新版本发布作些准备。

# 第八章：Code First将走向哪里？

到目前为止，本书已经覆盖了Code First组件的所有功能（包含截止本书写作时的最后一个正式发布版本）。但是还有很多你应该知道的显著功能已经纳入了预览版本。这些功能包括通过迁移数据库构架作为Code First的模型，从现在数据库中逆生成Code First模型和许多其他的有用的功能。

这些特性已经制成Entity Framework NuGget包作为插件进行了发布，可以分别单独下载。目前就有两个可用的插件。第一个，EntityFramework.Migrations NuGet package,为Code First添加了数据库迁移能力。第二个Entity Framework Power Tools，提供了许多Code First的设计时工具，可用于Visural Studio中。

## Code First的数据库迁移

本书中我们使用数据库初始化器在每次模型变化时来删除和重建数据库。这不够理想，因为这样做在每次模型变化时都会丢失数据。在本地开发时这也许是可接受的，但是一旦想将变化放在产品中这绝对不是一个可行的解决方案。现在你被强制使用构架比较 工作或手写脚本推动数据库变更到产品数据库。

自从@CF发布后，来自开发者社区的多个呼声指向迁移解决方法。迁移允许的数据库构架随着Code First模型的变化 逐步演变。现在有很多可用的迁移解决方案，但没有一个集成在Code First上。多数解决方案在数据库迁移功能提供了类似的方法。任何一套对数据库变更都会表达代码文件中称之为迁移。迁移是有序的，通常使用一个时间戳和一个数据库中的表保持跟何种踪移已应用到数据库。  
@EF团队一直致力于提供一个迁移解决方案，能够紧密集成在Code First。Code First迁移Alpha3已经在2011年9月月初发布。Alpha3发布的详细信息，可以在<http://blogs.msdn.com/b/adonet/archive/2011/09/21/code-first-migrations-alpha-3-released.aspx>找到。这种早期预览版聚焦于在Visual Studio的开发体验，并在Code First模型发生改变时允许迁移到已创建和执行过的数据库上。  
Code First目前已经发布了EntityFramework.Migrations NuGet包。一旦安装完毕，它增加了一组命令到包管理器控制台，可用于生成和运行迁移命令。  
当Code First模型变化后，你可以要求@CF迁移到创建一个迁移，应用相应的变化到数据库。迁移使用代码表示，一旦迁移已经创建，您可以自由编辑为您生成的代码。例8-1给出了迁移的代码样例。

namespace BreakAway.Migrations

{

using System.Data.Entity.Migrations;

public partial class SampleChanges : DbMigration

{

public override void Up()

{

AddColumn(

"Destinations",

"Description",

c => c.String());

ChangeColumn(

"Destinations",

"Name",

c => c.String(maxLength: 100));

}

public override void Down()

{

ChangeColumn(

"Destinations",

"Name",

c => c.String());

DropColumn(

"Destinations",

"Description");

}

}

}

迁移用类似于Fluent API的API来表达。在上述例子中，你可以看到一个描述列被添加到Destinations表中，而Name列的最大长度改为50。provider模型是用来将代码中的具体操作转化成数据库特定的SQL代码。  
Code First迁移还支持自动迁移，允许简单的变化，如​​列增加，执行，可以在项目没有迁移存在的情况下执行。您可以同一解决方案中同时使用自动迁移和基于代码的迁移，让你所做的每个更改都处于正确的控制水平。

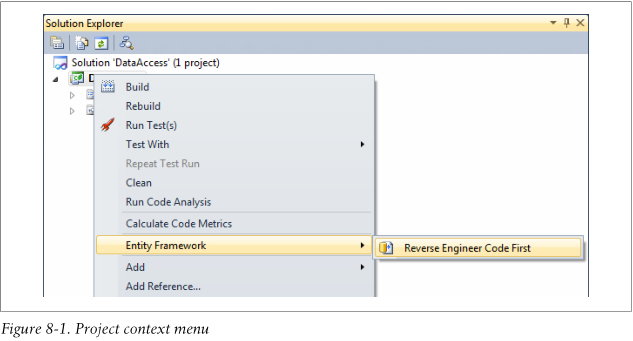
## Entity Framework工具

@EF Power Tools社区预览版（CTP1）于2011年5月发布。这个安装包可以通过VS扩展管理器进行安装，也可 以直接从VS Gallery下载：<http://visualstudiogallery.msdn.microsoft.com/72a60b14-1581-4b9b-89f2-846072eff19d>.

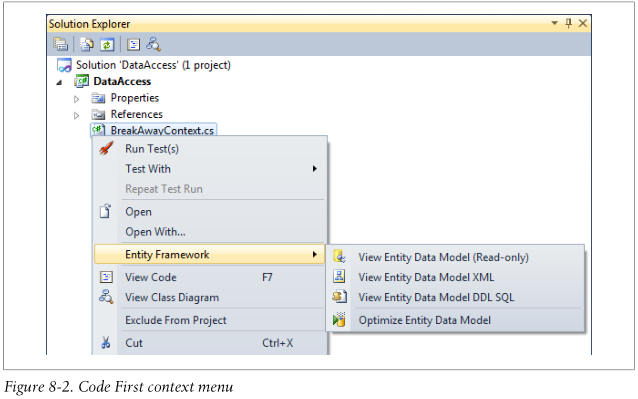
一旦安装，Power Tools会在VS中添加一些设计时工作，可以通过VS解决方案浏览器的上下文菜单来访问。

## Code First逆向工程

设计工具的安装后，你会发现在解决方案资源管理器中项目的上下文菜单中出现了一个新的菜单选项—“Enity Framework”（图8-1）。新项下，有一个”Reverse Engineer Code First“子菜单项。选择此选项，会提示你指向一个现有的数据库，这个数据库就是您想使用Code First类来映射的数据库。该工具将会使用数据库的架构产生一个Code First的context，一系列POCO类，以及POCO类映射到数据库的配置设置。在博客文章“Quick Look at Reverse Engineer DB into Code First Classes”，你可以看到这些类看起来像什么。  
逆向工程过程的设计为对现有数据库一次性实施代码生成。一旦生成代码，您可能需要调整，以满足您的的数据库架构。例如，CTP1不会对dbo架构生成正确的代码。如果你有另一个构架中的表，您将需要编辑ToTable调用相关的配置对象来指定正确的构架。  
Power Tools CTP1只支持逆向工程到C＃。Entity Framework团队将在未来版本中引入生成VB.NET的能力。

[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/201201071452069213.png)

## 查看Code First模型

@EF Power Tools还在“Entity Framework”项中添加了针对继承自DbContext的类的上下文菜单项（图8-2）。当在解决方案资源管理器左键单击代码文件时，Entity Framework 就有了四个选项，前三个提供了模型的不同查看方式：  
  
[](http://images.cnblogs.com/cnblogs_com/qouoww/201201/20120107145207226.png)

## 查看实体数据模型（只读）

“View Entity Data Model(Read Only）”选项将启动Entity Data Model设计器显示定义在context中的模型。Power Tools通过创建了一个临时的代表模型的EDMX文件实现此功能并将其在设计器中打开。

这是对模型的只读查看，任何在设计器的更改不会对代码有任何改变。你能做修改是由于设计品是VS的一部分，没有只读模式。查看模型能力在你试图解析Code First如何表达类与配置时是有用的。设计器也显示映射，因此你可以看到Code First如何映射你的类和属性到数据库的表和列中。

## 查看实体数据模型XML

“View Entity Data Model XML”选项在XML编辑器中载入等价于@CF模型的EDMX文件。

这个选项主要用来给Entity Framework团队识别Code First生成模型存在的问题。

## 查看实体数据模型DDL SQL

“View Entity Data Model DDL SQL ”选项用于生成与Code First用来从模型生成数据库所用的SQL脚本相同的脚本。在使用Code First开发中可能会用到，因为目前需要手写SQL脚本来创建最终产品数据库。

## 优化实体数据模型

在@EF中的上下文菜单项目的第四个选项是“优化实体数据模型。”从而允许您使用Code First模型执行视图生成。

视图生成是一个过程，这个过程中Entity Framework执行一些SQL语句计算，这些语句可用于在模型中对每个类型进行选择，插入，更新，删除。这个过程通常发生在您第一次在应用程序进程中使用上下文时。如果你有一个大的和/或复杂的模型，视图生成是一项耗费资源的操作，非常大的模型在需要几分钟甚至几小时。  
  
为了避免产生这种每次应用程序运行发生的问题，你可以使用预编译的视图。这正是此选项要做的工作：它预先计算 SQL并存储在一个项目中的代码文件中。@EF将在运行时执行这些代码，使用预计算的视图，而不是执行视图生成。执行视图生成后如果模型发生改变，需要重新运行此选项再次计算视图。  
预编译视图是相当先进的功能，这里不建议您使用它们，除非你遇到了有关于视图生成相关的性能问题。  
预编译和视图生成不只存在于Code First中。这些功能已在EDMX文件中实现，并在Entity Framework第一个版本就有了。Power Tools简单地让你能够在Code First中使用这些功能。你可以阅读更多有关预编译视图和视图生成的内容，见Programing Entity Framework，第二版的第20章。

追随EF团队  
有一些方法可以使你保持获取Entity Framework团队开发的的新功能---甚至影响他们的工作的方向。The ADO.NET团队博客与EF团队共同分享最新发布的公告信息。@EF团队也有专用的设计博客，在那里可以早期获取新功能 的想法以及将要开始开发的功能。这允许您在开发团阶段实施和发布预览版本之前将您的设计需求输入到Entity Framework的设计特征中去。最后，EF的团队有一个用户之声网站，您可以添加和为新功能要求投票。

# 来源：

http://www.cnblogs.com/qouoww/