# Code First 数据注释

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

publicclass Blog

{

  public int Id { get; set; }

  public string Title { get; set; }

  public string BloggerName { get;set; }

  public virtual ICollection<Post> Posts { get;set; }

}

publicclass Post

{

  public int Id { get; set; }

  public string Title { get; set; }

  public DateTime DateCreated { get;set; }

  public string Content { get;set; }

  public int BlogId { get; set; }

  public ICollection<Comment>Comments { get; set;}

}

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

## 键

您可以使用键注释来指定要将哪一个属性用作 EntityKey。

publicclass Blog

{

  [Key]

  public int PrimaryTrackingKey { get;set; }

  public string Title { get; set; }

  public string BloggerName { get;set; }

  public virtual ICollection<Post> Posts { get;set; }

}

## 必需

[Required]

publicstring Title { get;set; }

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

## MaxLength 和MinLength

[MaxLength(10), MinLength(5)]

publicstring BloggerName { get;set; }

MaxLength 注释将通过把属性长度设置为 10 来影响数据库。

MinLength属性不会对数据库产生影响。

**Range**

[Range(0,10)]

数值范围

## NotMapped

您可以使用 NotMapped 注释来标记不映射到数据库的所有属性，如下面的 BlogCode 属性。

[NotMapped]

publicstring BlogCode

{

  get

  {

    returnTitle.Substring(0, 1) + ":" +BloggerName.Substring(0, 1);

  }

}

## ComplexType

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

publicclass BlogDetails

{

  public DateTime? DateCreated { get;set; }

  [MaxLength(250)]

  public string Description { get;set; }

}

请注意，BlogDetails 没有任何键属性类型，为复杂类型。复杂类型不能自行跟踪。

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

[ComplexType]

publicclass BlogDetails

{

  public DateTime? DateCreated { get;set; }

  [MaxLength(250)]

  public string Description { get;set; }

}

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

   public BlogDetails BlogDetail { get; set; }

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

## 

## ConcurrencyCheck

ConcurrencyCheck 注释可用于标记要在用户编辑或删除实体时用于在数据库中进行并发检查的一个或多个属性。现在让我们将ConcurrencyCheck 添加到BloggerName 属性，看看它如何工作。

[ConcurrencyCheck, MaxLength(10),MinLength(5)]

publicstring BloggerName { get;set; }

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

where (([PrimaryTrackingKey]= @4) and([BloggerName] = @5))

@4=1,@5=N'Julie'

## 

## TimeStamp

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

将以下属性添加到Blog 类：

[Timestamp]

publicByte[] TimeStamp { get;set; }

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

## 表和列

下面的注释指定表名称为 InternalBlogs，同时指定了schema，默认的schema就是dbo。

[Table("InternalBlogs",Schema="dbo")]

publicclass Blog

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

[Column("BlogDescription",TypeName = "ntext")]

publicString Description { get; set; }

## DatabaseGenerated

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

[DatabaseGenerated(DatabaseGeneratedOption.Computed)]

publicDateTime DateCreated { get; set; }

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

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

## 关系特性：InverseProperty和ForeignKey

ForeignKey用于指定BlogId作为Blog的外键

publicclass Post

{

  public int Id { get; set; }

  public string Title { get; set; }

  public DateTime DateCreated { get;set; }

  public string Content { get;set; }

  public int BlogId { get; set; }

  [ForeignKey("BlogId")]

  public Blog Blog { get; set; }

}

类之间存在多个关系时，将使用 InverseProperty。

Course表的Department1（表中的名字为Department1\_DepartmentID ） ，Department2 （表中的名字为Department2\_DepartmentID ）约束与Department表的主键DepartmentID

public class Department

{

// Primary key

public int DepartmentID { get; set; }

public string Name { get; set; }

[InverseProperty("Department1")]

public virtual ICollection<Course> Course1 { get; set; }

[InverseProperty("Department2")]

public virtual ICollection<Course> Course2 { get; set; }

## }

public class Course

{

// Primary key

public int CourseID { get; set; }

public string Title { get; set; }

public int Credits { get; set; }

public int ID { get; set; }

public virtual Department Department1 { get; set; }

public virtual Department Department2 { get; set;}

}

## 总结

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

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