Entity Framework Code First Learning Note

1. LocalDB 介绍

安装完成SQL Server Express 后，

Express 2012默认会创建一个 V11.0 的本地数据库。

Express 2016 则默认创建 MSSQLLocalDB 的本地数据库

* 1. 连接字符串

.Net 连接 LocalDB 其实没有什么不同的地方， 主要就是Server 参数使用 (LocalDB)

<connectionStrings>

<add name="CustomerContext"

providerName="System.Data.SqlClient"

connectionString="Server=(LocalDB)\MSSQLLocalDB;Database=CustomerDB;Trusted\_Connection=True;Integrated Security=True;Persist Security Info=True;" />

</connectionStrings>

Database – Initial Catalog 相同

* 1. 连接安全， 虽然LocalDB 也支持 SQL User, Password 登录，如果用户名和密码不正确则出错

<connectionStrings>

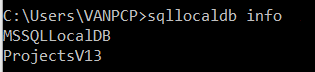
<add name="CustomerContext"

providerName="System.Data.SqlClient" connectionString="Server=(LocalDB)\MSSQLLocalDB;User=lwh;Password=lwh123;Initial Catalog=CustomerDB;Trusted\_Connection=false;Integrated Security=False;Persist Security Info=False;" />

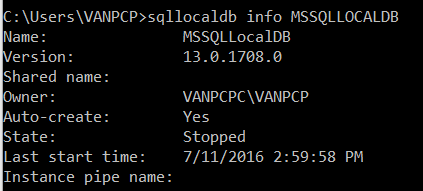
</connectionStrings>

* 1. SqlLocalDB.exe 工具
     1. SqlLocalDB info

列出所有LocalDB 实例，当然是属于登录Windows用户下的所有本地数据库

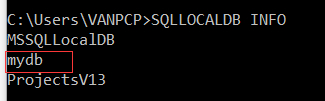


SqlLocalDB info MSSQLLocalDB



* + 1. SqlLocalDB create [instanceName]





创建完以后，状态是 Stopped

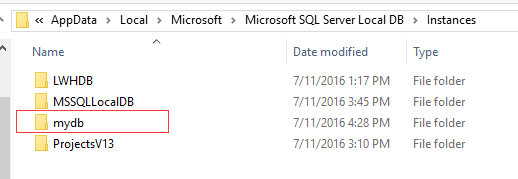
SQLLocalDB Create [InstanceName]

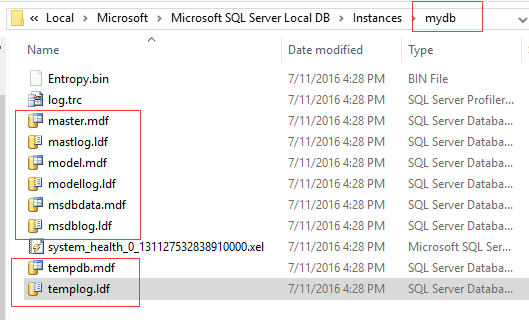
运行这个命令之后，只是创建了SQL Server的一个实例，此时只有系统数据库，并没有用户自定义的数据库。 而这些系统数据库的位置默认位置是

C:\Users\Will\AppData\Local\Microsoft\Microsoft SQL Server Local DB\Instances

系统会自动创建系统数据库在用户目录下：

C:\Users\VANPCP\AppData\Local\Microsoft\Microsoft SQL Server Local DB\Instances

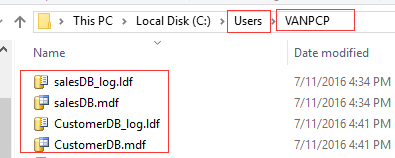




注意：

1. 以上都是系统数据库，
2. 而用户数据库的存放位置，默认是C:\Users\UserName

C:\Users\VANPCP



1. 如何指定用户数据库的位置:

如果是 code first 自动创建用户数据库，则可以在连接字符串里指定位置和文件名：

<connectionStrings>

<add name="CustomerContext"

providerName="System.Data.SqlClient" connectionString="Server=(LocalDB)\mydb;

Initial Catalog=CustomerDB;

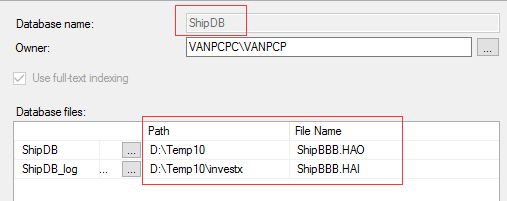
Trusted\_Connection=True;

Integrated Security=True;

AttachDbFileName=D:\Temp8\custt.mdf" />

</connectionStrings>

如果是通过SQL Management Client Tool



路径和文件名都可以使用自定义的名字

1. 如何通过 Code First 附加用户数据库， 可以通过连接字符串

<connectionStrings>

<add name="CustomerContext"

providerName="System.Data.SqlClient" connectionString="Server=(LocalDB)\MetroTown;

Initial Catalog=wm\_maindb;

Trusted\_Connection=True;

Integrated Security=True;

AttachDbFileName=D:\Temp12\wm\_maindb.mdf" />

</connectionStrings>

第一）可以把 xxx.mdf 复制到本地目录

第二）指定数据库名和原来名字一样， 表格名称不一样的话，则会在此基础上创建新表。

第三）Code First , 如果是DropCreateDatabaseIfModelChanges ， 任何表的定义发生改变，则删除整个数据再重新创建新数据库， 原来的表格和数据都会丢失

<connectionStrings>

<add name="CustomerContext"

providerName="System.Data.SqlClient" connectionString="Server=(LocalDB)\VanDB;

Initial Catalog=DownTown;

Trusted\_Connection=True;

Integrated Security=True;

AttachDbFileName=D:\Temp12\wm\_maindb.mdf" />

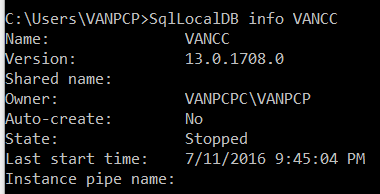
</connectionStrings>

如果数据库名称和原来的不一样， 则会完全覆盖掉 xxx.mdf 的所有定义，然后创建新的用户数据库， 写入到 xxx.mdf 里

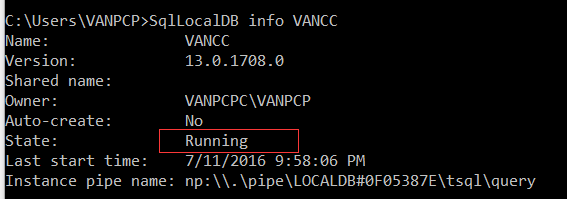
* + 1. Attach User Database to SQL Instance

sqlcmd -S (LocalDB)\xxxInstanceName

* Use Master
* Go
* CREATE DATABASE XXTest ON (FILENAME=N'D:\Temp\MyDB.mdf') FOR ATTACH
  + 1. SqlLocalDB start instanceName



SqlLocalDB start VANCC

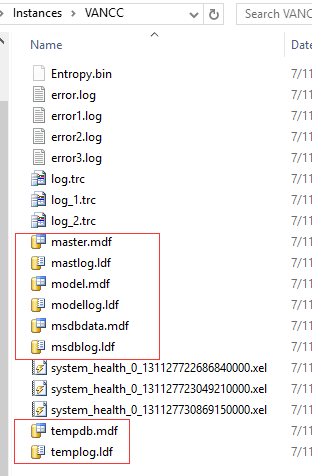
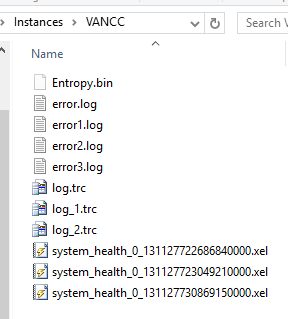


* + 1. SqlLocalDB stop instanceName
    2. SqlLocalDB delete instanceName

如果是正在运行状态的， 必须要停止， 才能够删除。

注意： 只会删除该实例的系统数据库， 并不会删除创建过的用户数据库。

也不会删除记录过的错误日志等文件， 如果想彻底删除干净，必须到相关目录手工删除

 删除以后 

# SqlLocalDB 公用程式

 適用於: SQL Server 2016

使用 **SqlLocalDB** 公用程式可建立 MicrosoftSQL Server 2016 Express**LocalDB** 的執行個體。**SqlLocalDB** 公用程式 (SqlLocalDB.exe) 是一個簡單的命令列工具，可讓使用者和開發人員建立及管理 SQL Server Express**LocalDB** 的執行個體。 如需有關如何使用 **LocalDB** 的詳細資訊，請參閱＜[SQL Server 2016 Express LocalDB](https://msdn.microsoft.com/zh-tw/library/hh510202.aspx)＞。

## [語法](javascript:void(0))

SqlLocalDB.exe

{

[ create | c ] <instance-name><instance-version> [-s ]

| [ delete | d ] <instance-name>

| [ start | s ] <instance-name>

| [ stop | p ] <instance-name> [ -i ] [ -k ]

| [ share | h ] ["<user\_SID>" | "<user\_account>" ] "<private-name>""<shared-name>"

| [ unshare | u ] "<shared-name>"

| [ info | i ] <instance-name>

| [ versions | v ]

| [ trace | t ] [ on | off ]

| [ help | -? ]

}

## [引數](javascript:void(0))

[ **create** | **c** ] <instance-name> <instance-version> [**-s** ]  
建立 SQL Server Express**LocalDB** 的新執行個體。**SqlLocalDB** 會使用 <instance-version> 引數所指定之 SQL Server Express 二進位檔的版本。 使用至少一個十進位數的數字格式指定版本號碼。 次要版本號碼 (Service Pack) 為選擇性。 例如，下列兩個版本號碼都可接受：11.0 或 11.0.1186。 電腦上必須安裝指定的版本。 如果未指定，版本號碼會預設為 **SqlLocalDB** 公用程式的版本。 加入 **–s** 會啟動新的 **LocalDB** 執行個體。

[ **share** | **h** ]  
使用指定的共用名稱來共用指定的 **LocalDB** 私用執行個體。 如果省略使用者 SID 或帳戶名稱，會預設為目前的使用者。

[ **unshared** | **u** ]  
停止共用指定的 **LocalDB** 共用執行個體。

[ **delete** | **d** ] <instance-name>  
刪除指定的 SQL Server Express**LocalDB** 執行個體。

[ **start** | **s** ] " <instance-name> "  
啟動指定的 SQL Server Express**LocalDB** 執行個體。 當成功的陳述式傳回 **LocalDB** 的具名管道位址時。

[ **stop** | **p** ] <instance-name> [**-i** ] [**-k** ]  
停止指定的 SQL Server Express**LocalDB** 執行個體。 加入 **–i** 會要求使用 **NOWAIT** 選項關閉此執行個體。 加入 **–k** 會在未經連絡的情況下終止執行個體處理序。

[ **info** | **i** ] [ <instance-name> ]  
列出目前使用者擁有的所有 SQL Server Express**LocalDB** 執行個體。

<instance-name> 會傳回名稱、版本、狀態 (執行中或已停止)、指定之 SQL Server Express**LocalDB** 執行個體的上次啟動時間，以及 **LocalDB** 的本機管道名稱。

[ **trace** | **t** ] **on** | **off**  
**trace on** 會針對目前使用者啟用 **SqlLocalDB** API 呼叫的追蹤。**trace off** 停用追蹤。

**-?**  
傳回每一個 **SqlLocalDB** 選項的簡短描述。

## [備註](javascript:void(0))

instance name 引數必須遵循 SQL Server 識別碼的規則，或者必須以雙引號括住。

不使用任何引數執行 SqlLocalDB 會傳回說明文字。

啟動以外的作業只能在屬於目前登入之使用者的執行個體上執行。 SQLLOCALDB 執行個體，當共用時，只能由該執行個體的擁有者啟動和停止。

## [範例](javascript:void(0))

### A. 建立 LocalDB 的執行個體

下列範例會使用 SQL Server 2016 二進位檔建立名為 DEPARTMENT 的 SQL Server Express **LocalDB** 執行個體，並啟動此執行個體。

SqlLocalDB.exe create "DEPARTMENT" 12.0 -s

### B. 使用 LocalDB 的共用執行個體

使用管理員權限開啟命令提示字元。

SqlLocalDB.exe create "DeptLocalDB"

SqlLocalDB.exe share "DeptLocalDB" "DeptSharedLocalDB"

SqlLocalDB.exe start "DeptLocalDB"

SqlLocalDB.exe info "DeptLocalDB"

REM The previous statement outputs the Instance pipe name for the next step

sqlcmd –S np:\\.\pipe\LOCALDB#<use your pipe name>\tsql\query

CREATE LOGIN NewLogin WITH PASSWORD = 'Passw0rd!!@52';

GO

CREATE USER NewLogin;

GO

EXIT

使用 NewLogin 登入執行以下程式碼以連接到 **LocalDB** 的共用執行個體。

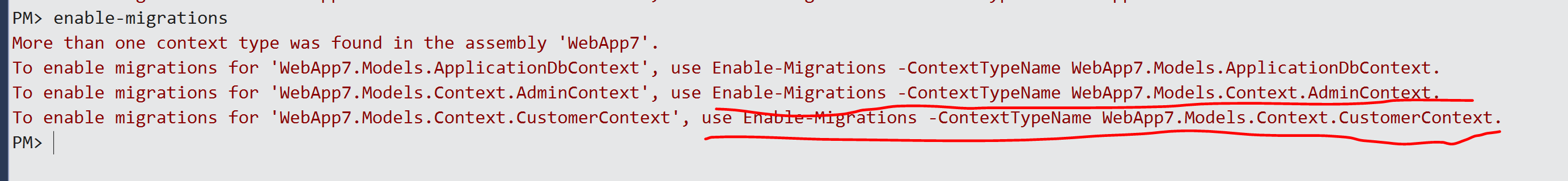
sqlcmd –S (localdb)\.\DeptSharedLocalDB -U NewLogin -P Passw0rd!!@52

### C. 启用数据库迁移 – Visual Studio环境下对数据上下文迁移：

### 

### 运行指令：

### enable-migrations–会列出所有的数据库上下文，选择我们需要升级的数据库上下文



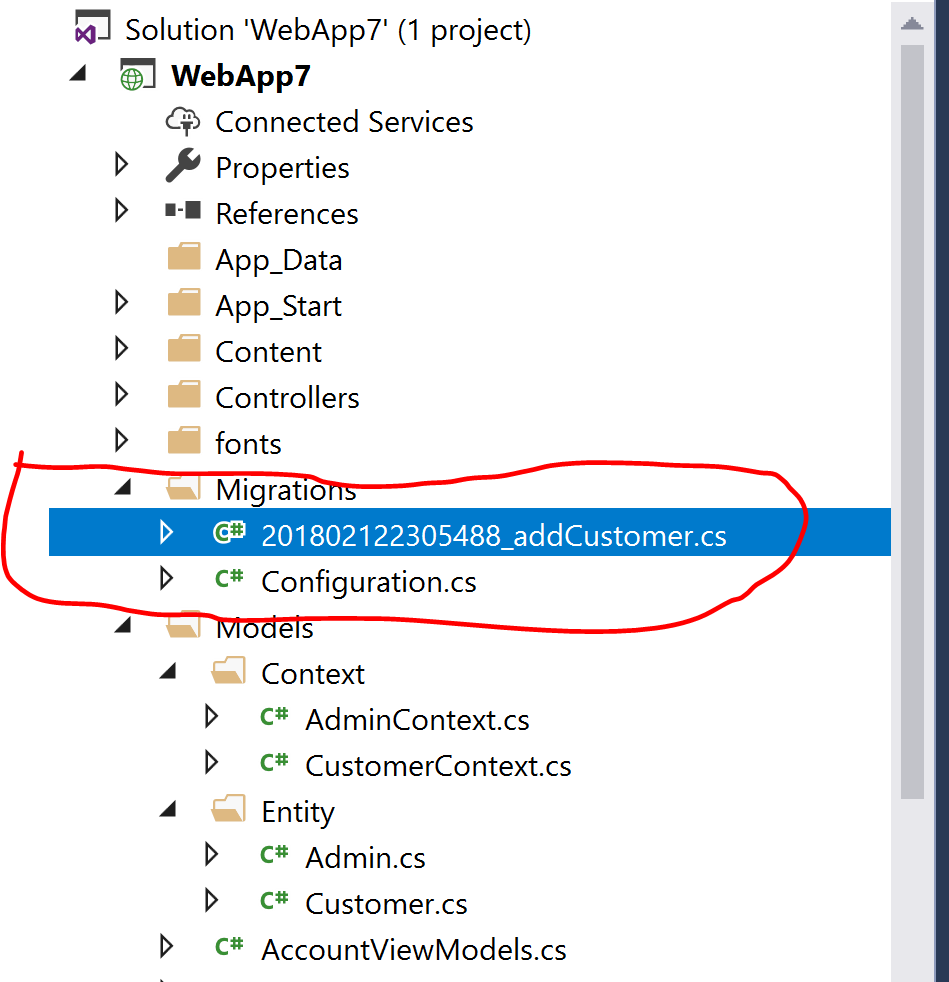
选择以后，运行命令：

Enable-Migrations -ContextTypeName WebApp7.Models.Context.CustomerContext

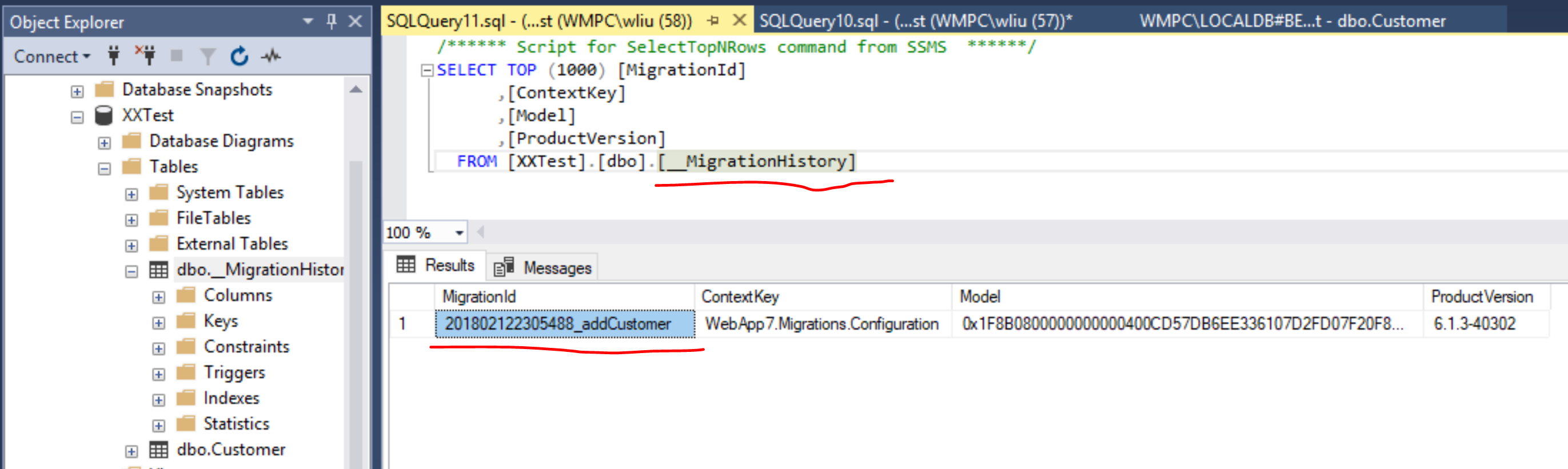
如果我们将CustomerContext 定义好了: 可以运行一次升级数据库，初始化SQL 数据库

add-migration addCustomer 名字可以随意定义

运行完我们会看到在项目里会自动生成一个目录：



此时可以运行：update-database 提交更新到数据库



如果我们再次对Customer 修改

[Table("Customer")]

public class Customer

{

[Key]

public int Id { get; set; }

public string Name { get; set; }

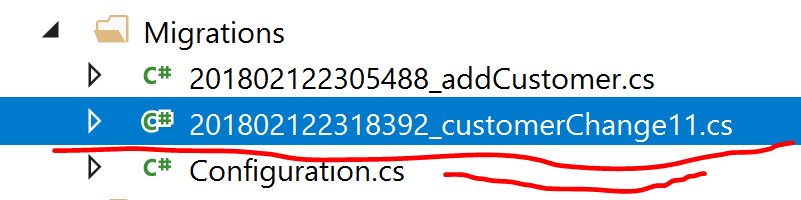
public int Level { get; set; } --新增

public decimal Price { get; set; } --新增

}

再次运行：

add-migration customerChange11



此文件的内容是：

namespace WebApp7.Migrations

{

using System;

using System.Data.Entity.Migrations;

public partial class customerChange11 : DbMigration

{

public override void Up() **-- 升级**

{

AddColumn("dbo.Customer", "Level", c => c.Int(nullable: false));

AddColumn("dbo.Customer", "Price", c => c.Decimal(nullable: false, precision: 18, scale: 2));

}

public override void Down() **-- 降级**

{

DropColumn("dbo.Customer", "Price");

DropColumn("dbo.Customer", "Level");

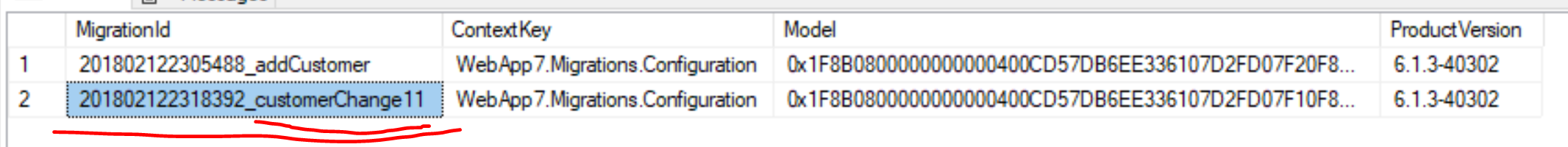
}

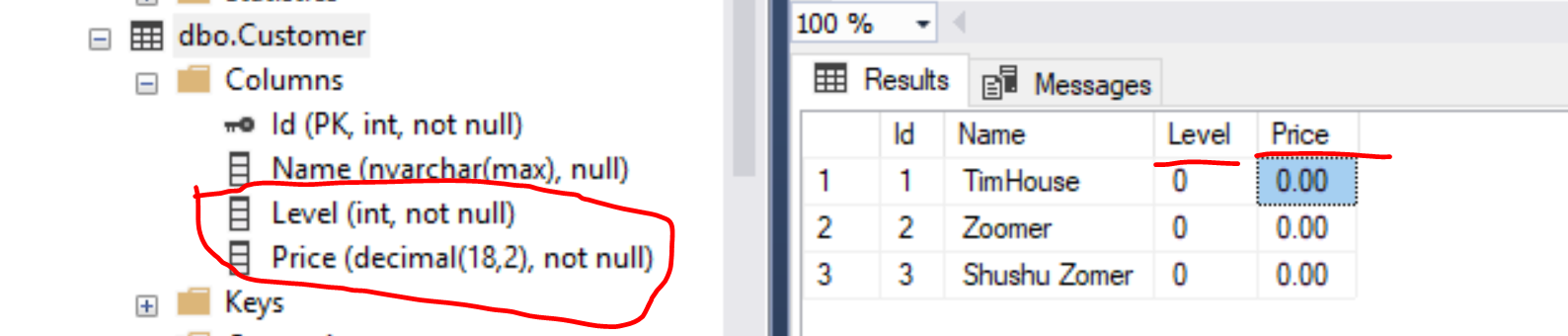
}

}

提交数据库升级：运行命令

update-database





注意如何避免数据库被自动创建和自动升级：

public class CustomerContext : DbContext

{

static CustomerContext() – 只运行一次

{

Database.SetInitializer<AdminContext>(null); -- 传入null

}

public CustomerContext() : base("mydbcon")

{

}

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

}

public DbSet<Customer> Customers { get; set; }

}

1. DbContext & DbSet

**注意**：在使用Code First开发时要特别注意，若要使用“延迟载入”，必须在属性上声明“Virtual”关键字，才会启用“延迟载入”特性。

* 1. 定义实体 Entity

[Table("Employee")] – Table 用于映射到数据库的表格, 如果不存在用于创建表格时的表格名称

public class EmployeeEntity

{

public int empid { get; set; }

public string firstName { get; set; }

public string lastName { get; set; }

}

注意每个Entity 必须要有主键：否则出错

如果没有指定key, 会自动寻找“id”当做key. 默认是 Identity seed 1, increase 1

* 1. Model Metadata – 数据模型用于映射的相关的属性

using System.ComponentModel.DataAnnotations.Schema;

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

public class EmployeeEntity

{

[Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]

[Column("employeeID", Order =0, TypeName = "BigInt")]

public int eid { get; set; }

[Key]

[Column(Order = 1), DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public int pid { get; set; }

public Address addr { get; set; }

[Column("Fname", Order =3, TypeName ="Varchar")]

public string firstName { get; set; }

[Column("Lname", Order = 2, TypeName = "Varchar")]

public string lastName { get; set; }

public ICollection<TaskList> tasklist;

}

[Table("tablename")] – 用于映射到数据库的表

[Column("colName", Order =0, TypeName = "Int")]

[NotMapped] - 不映射到数据库， 在数据库端也不创建字段,可以应用于 class, property

[DatabaseGenerated(DatabaseGeneratedOption.Identity)] – 一个表只能有一个identity 字段,

可以指定给其他非主键字段， 这时主键字段默认不会再设置 Identity特性。

* colName 用于映射数据库表的字段，
* Order 数值越小越排前， 数值必须唯一不能重复， 数值可以任意不一定要连续
* TypeName 是数据库的类型定义： 如： Varchar, NVarchar, Image, BigInt, SmallInt
  1. Model Metadata – 数据模型相关的属性
     1. [Key]

using System.ComponentModel.DataAnnotations;

[Key] – 设置主键， 如果类型是 Int, BigInt且没有其他Identity字段，会自动设置 Identity Seed = 1;

increment = 1; (其他类型不行, decimal, float, numeric),

[Key]

[Column("employeeID", Order =1, TypeName = "Bigint")]

public long eid { get; set; }

复合主键如何设置：

[Table("Employee")]

public class EmployeeEntity

{

[Key]

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

[Column("empid", Order =1, TypeName = "int")]

public int eid { get; set; }

[Key]

[Column("pagid", Order = 9)]

public int pid { get; set; }

public Address addr { get; set; }

[Column("Fname", Order =7, TypeName ="Varchar")]

[StringLength(200, MinimumLength = 10), Required]

public string firstName { get; set; }

[Column("Lname", Order = 8, TypeName = "Varchar")]

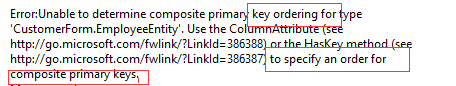
public string lastName { get; set; }

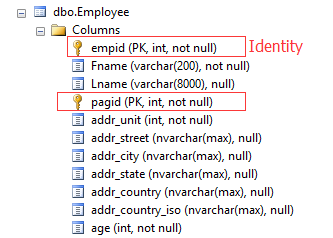
public int age { get; set; }

public ICollection<TaskList> tasklist;

}

提示：可以设置两个或以上[Key], 前提是必须有[Column(Order =1)],如果不设置 Order 则出错





其他类型的 Key : Guid

[Key,DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public Guid Identifier { get; set; }

[Table("Member")]

public class memberEntity

{

[Key, Column("member\_id")] - 默认不会自动创建 new\_id()

public Guid ggid { get; set; }

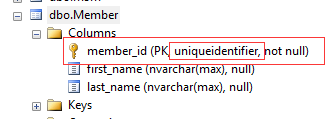
[Column("first\_name")]

public string firstName { get; set; }

[Column("last\_name")]

public string lastName { get; set; }

}



memberEntity m1 = new memberEntity { firstName = "William", lastName = "Liu" };

this.db.Members.Add(m1);

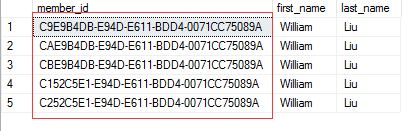
this.db.SaveChanges();



第一次插入数据没问题， 之后再插入数据就有问题，

[Key,DatabaseGenerated(DatabaseGeneratedOption.Identity), Column("member\_id")]

public Guid ggid { get; set; }



其他类型的 Key : [Timestamp]

public class memberEntity

{

[Key,Column("member\_id")]

[Timestamp]

public byte[] ggid { get; set; }

[Column("first\_name")]

public string firstName { get; set; }

[Column("last\_name")]

public string lastName { get; set; }

}

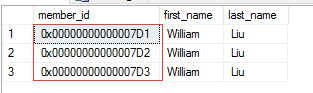
如果[Timestamp] 作为主键使用， 则不会自动生成时间戳， 必须要明确指明：

[Key,Timestamp,DatabaseGenerated(DatabaseGeneratedOption.Identity)]

memberEntity m1 = new memberEntity { firstName = "William", lastName = "Liu"};

this.db.Members.Add(m1);

this.db.SaveChanges();



public class memberEntity

{

[Key, Column("member\_id")]

public int mmid { get; set; }

[Timestamp]

public byte[] ggid { get; set; }

[Column("first\_name")]

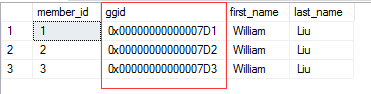
public string firstName { get; set; }

[Column("last\_name")]

public string lastName { get; set; }

}

[Timestamp]: 会自动生成时间戳



* + 1. [Index("idx\_uk\_oid", IsClustered = false, Order = 1, IsUnique = true)] – 定义索引

[Table("Teacher")]

public class TeacherEntity

{

[Key]

[Column("teacher\_id")]

public int tid { get; set; }

[Index("idx\_uk\_oid", IsClustered = false, Order = 1, IsUnique = true)]

[Column("other\_id")]

public int otherID { get; set; }

public string tname { get; set; }

}

一个表格可以有多个索引， 但是 Clustered 索引只有一个， 如果定义了[Key] 它就是

注意：

在 Entity Framework 里， 每个Entity 必须要有一个[Key]

[Index] 则不一定是必须的

public class TeacherEntity

{

[Key]

[Column("teacher\_id")]

public int tid { get; set; }

[Index("idx\_oid\_sid",4, IsUnique =true)]

public int oid { get; set; }

[Index("idx\_oid\_sid", 2, IsUnique = true)]

public int sid { get; set; }

public string tname { get; set; }

public ICollection<StudentEntity> students { get; set; }

}

可以设置复合索引， 只要指定同名，并指定不同的Order顺序

* + 1. [ForeignKey()] – 定义外键，用于定义表与表之间的关系

[Table("Teacher")]

public class TeacherEntity

{

[Key]

[Column("teacher\_id")]

public int tid { get; set; }

[Index("idx\_uk\_oid", IsClustered = false, Order = 1, IsUnique = true)]

[Column("other\_id")]

public int otherID { get; set; }

public string tname { get; set; }

}

[Table("Student")]

public class StudentEntity

{

[Key]

public int sid { get; set; }

public string sname { get; set; }

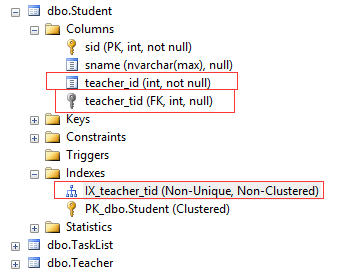
public int teacher\_id { get; set; }

public TeacherEntity teacher { get; set; }

}

在这个表关系中， 我们并没有定义 ForeignKey,

由于public TeacherEntity teacher { get; set; }， Entity Framework 自动添加外键



在没有指定外键的情况下， 自动创建了一个外键字段

“teacher” + “tid” = teacher\_tid

复合的主键，复合的外键情况：

[Table("Teacher")]

public class TeacherEntity

{

[Key]

[Column("teacher\_id",Order = 1)]

public int tid { get; set; }

[Key]

[Column("other\_id", Order = 2)]

public int otherID { get; set; }

public string tname { get; set; }

}

[Table("Student")]

public class StudentEntity

{

[Key]

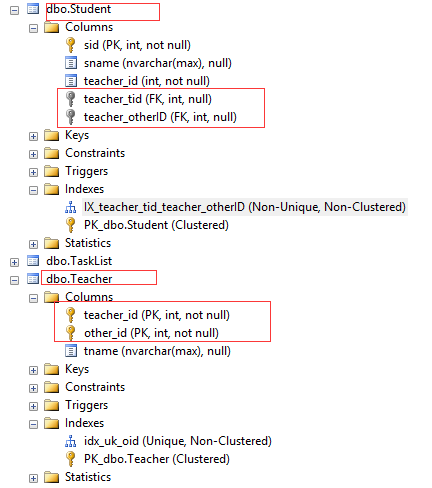
public int sid { get; set; }

public string sname { get; set; }

public int teacher\_id { get; set; }

public TeacherEntity teacher { get; set; }

}



复合外键可以自动添加

[Table("Student")]

public class StudentEntity

{

[Key]

public int sid { get; set; }

public string sname { get; set; }

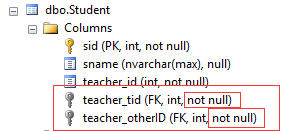
public int teacher\_id { get; set; }

[Required]

public TeacherEntity teacher { get; set; }

}

如果指定[Required]



那么如何指定字段呢？ 有两个方法都可以使用

1. 在字段上指定 外键的集合名称; 对于复合外键来说，[Column(Order = x)] 是必须的

[Table("Student")]

public class StudentEntity

{

[Key]

public int sid { get; set; }

public string sname { get; set; }

[ForeignKey("**teacher**")]

[Column("fk\_teaid",Order = 3)]

public int fk\_tid { get; set; }

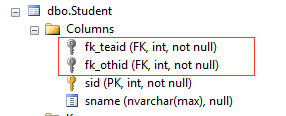
[ForeignKey("**teacher**")]

[Column("fk\_othid", Order = 5)]

public int fk\_oid { get; set; }

public TeacherEntity **teacher** { get; set; }

}



1. 在集合上指定 外键字段； 因为在集合上只能指定一个外键字段， 不能用于复合情况

[Table("Teacher")]

public class TeacherEntity

{

[Key]

[Column("teacher\_id")]

public int tid { get; set; }

public string tname { get; set; }

public ICollection<StudentEntity> students { get; set; }

}

[Table("Student")]

public class StudentEntity

{

[Key]

public int sid { get; set; }

public string sname { get; set; }

public int fk\_tid { get; set; }

[ForeignKey("fk\_tid")]

public TeacherEntity teacher { get; set; }

}

* + 1. [MinLength(10)], [MaxLength(5)], [StringLength(200, MinimumLength = 10)]

只能应用于 string, byte[] 类型， 否则出错

string – 对应的类型是 nvarchar(max), null

byte[] – 对应的类型是 varbinary(max), null , 加了Required 是 not null

byte - 对应的类型是 tinyint, not null

bool - 对应的类型是 bit, not null

int - 对应的类型是 int, not null

DateTime - 对应的类型是 datetime, not null

TimeSpan - 对应的类型是 time(7), not null

[Column("colName", TypeName ="SQL DataType")] – 可以指定兼容的SQL DataType, 如果不兼容，则出错

如：[Column(TypeName ="Image")]

public Image img { get; set; } - C# Image 不能直接映射到 SQL Image

正确做法：byte[] - Image

[Column(TypeName ="Image")]

public byte[] img { get; set; }

[MinLength(10), MaxLength(5)]

public string firstName { get; set; }

MaxLength(5) – 对数据库起作用, 数据库的类型为 Varchar(5)

MinLength(10) – 对数据库没有任何影响和约束， 但会对前端的数据有效性验证有作用

[Column("Fname")]

[StringLength(30, MinimumLength = 15)]

[MinLength(10),MaxLength(8)]

[Required]

public string firstName { get; set; }

[Column("Lname")]

[StringLength(12)]

public string lastName { get; set; }

StringLength(30, MinimumLength = 15) – 的优先级别比MaxLength(5) 低, 但是对于验证，则是两个同时起作用

StringLength() – 对于ASP.NET MVC 和 Dynamic Data， 有效， 因为他们不认 [MinLength(10),MaxLength(8)]

* + 1. [Required( AllowEmptyStrings =true,

ErrorMessage ="Name is requred",

ErrorMessageResourceName ="First Name",

ErrorMessageResourceType = typeof(string)

)]

反应到数据库对应 not null

[Required]

[Column("Status")]

public int? status { get; set; }

[Required] 优先级别比Nullable<int> 高

* + 1. [EmailAddress] [Phone] [MembershipPassword] [Range(12, 25)]

这些验证不会对数据库后端有任何影响， 只是用于客户端输入数据有效性验证。

[Range(typeof(string), "A", "Z")]

public string sta { get; set; }

[Range(1.5, 3.6)] - 这里指定是 decimal

public int sta { get; set; } – 这里指定是 int ,如果录入是decimal 则会提示类型不兼容错误

[Range(10, 20)]

public int sta { get; set; }

* + 1. [RegularExpression(@"^[1-9]\*$")] – 正则表达式

只是前端输入验证， 对数据库没有影响

* + 1. 自定义验证： 继承自RegularExpression

public class CellAttribute: RegularExpressionAttribute

{

public CellAttribute() : base(@"^(604|778)-[0-9]\*$") { }

}

[Cell(ErrorMessage ="Cell Number invalid")]

public string cell { get; set; }

* + 1. 自定义验证： 继承自ValidationAttribute

public class TellAttribute: ValidationAttribute

{

private string \_some;

public TellAttribute() { }

public TellAttribute(string some)

{

this.\_some = some;

}

public override bool IsValid(object value)

{

if (value.ToString() == this.\_some)

return true;

else

return false;

}

}

[Tell("Hello", ErrorMessage ="Cell Number = Hello")]

public string cell { get; set; }

* + 1. [Compare("pass2", ErrorMessage ="Pass2 <> Pass1")]

[Compare("pass2", ErrorMessage ="Pass2 <> Pass1")]

public string pass1 { get; set; }

[NotMapped]

public string pass2 { get; set; }

用于比较密码很方便

* + 1. [MembershipPassword] – 符合成员资格提供者当前的密码需求，通过Membership提供者所定义的密码复杂

程度要求进行检查

using System.Web.Security;

[MembershipPassword]

[Compare("pass2", ErrorMessage ="Pass1 != Pass2")]

public string pass1 { get; set; }

[NotMapped]

public string pass2 { get; set; }

* + 1. [Remote] – ASP.Net MVC 通过Ajax送到指定的Action做验证，可以验证如验证用户账号是否被使用过
    2. [DataType] – 只应用于 ASP.NET MVC 应用于 View HTML helper 输入控件

[DataType(DataType.EmailAddress, ErrorMessage ="Invalid Email Me")]

public string email { get; set; }

* + 1. 其他应用于 ASP.NET MVC 的属性通常和 View 的输入控件有关

[DisplayName("Email Address")]

[DisplayColumn("email", "sortColumn")]

[DisplayFormat]

* + 1. Partial class - [MetadataType(typeof(CustomerMeta))] 将 Metadata 定义分开来写

[Table("Customer3")]

public partial class Customer

{

[Browsable(false)]

[DisplayName("Customer ID")]

public int CustomerID { get; set; }

public Guid gcode { get; set; }

[DisplayName("C Name")]

public string CustomerName { get; set; }

[DisplayName("Phone Number")]

[Column("Phone")]

public string phone { get; set; }

[Required]

[EmailAddress(ErrorMessage ="Email Me Required", ErrorMessageResourceName ="MailMe")]

[Column("Email")]

public string email { get; set; }

[Range(10, 20)]

public int sta { get; set; }

[Column(TypeName ="Image")]

public Byte[] photo { get; set; }

}

[MetadataType(typeof(CustomerMeta))]

public partial class Customer

{

private class CustomerMeta

{

[Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]

[Column("CustomerID")]

public object CustomerID { get; set; }

[DatabaseGenerated(DatabaseGeneratedOption.Identity)]

public object gcode { get; set; }

[RegularExpression(@"^[1-9]\*$", ErrorMessage = "dIGI NUMBER ONLY")]

[Required, MaxLength(30)]

public object CustomerName { get; set; }

[Required] - 这个会出错，因为没有定义过

public int shuxue { get; set; }

}

}

* 1. [ForeignKey(["fk\_col"|"fk\_entity"])] 与 [InverseProperty("fk\_entity")] 的使用

[Table("Teacher")]

public class TeacherEntity

{

[Key]

[Column("TeacherID")]

public int teacher\_id { get; set; }

public string tname { get; set; }

}

[Table("Student")]

public class StudentEntity

{

[Key]

[Column("StudentID")]

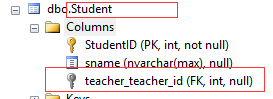
public int sid { get; set; }

public string sname { get; set; }

public TeacherEntity teacher { get; set; }

}

默认会自动创建 Foreign Key 在数据库 Student 表上， Student表上会自动增加一个外键字段 “teacher” + “teacher\_id”

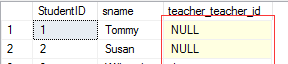


StudentEntity stu2 = new StudentEntity { sname = "Tommy" };

StudentEntity stu3 = new StudentEntity { sname = "Susan" };

this.db.Students.Add(stu2);

this.db.Students.Add(stu3);



TeacherEntity tch = new TeacherEntity { tname = "Luo Laoshi" };

StudentEntity stu = new StudentEntity { sname = "William Liu" };

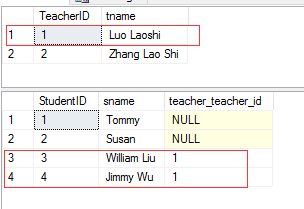
stu.teacher = tch;

StudentEntity stu1 = new StudentEntity { sname = "Jimmy Wu" };

stu1.teacher = tch;

this.db.Students.Add(stu);

this.db.Students.Add(stu1);



现在我们指定外键，不使用自动创建的外键

[Table("Teacher")]

public class TeacherEntity

{

[Key]

[Column("TeacherID")]

public int teacher\_id { get; set; }

public string tname { get; set; }

public ICollection<StudentEntity> students { get; set; }

}

[Table("Student")]

public class StudentEntity

{

[Key]

[Column("StudentID")]

public int sid { get; set; }

public string sname { get; set; }

[Column("fk\_tid")]

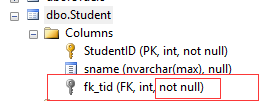
[ForeignKey("teacher")] - 或者在外键上指定实体名称

public int tid { get; set; } - 注意外键 not null

// [ForeignKey("tid")] – 或者在实体上指定外键字段

public TeacherEntity teacher { get; set; }

}

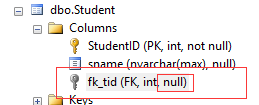


[Column("fk\_tid")]

public int? tid { get; set; }

[ForeignKey("tid")]

public TeacherEntity teacher { get; set; }

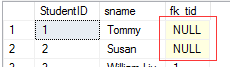


StudentEntity stu2 = new StudentEntity { sname = "Tommy" };

StudentEntity stu3 = new StudentEntity { sname = "Susan" };

this.db.Students.Add(stu2);

this.db.Students.Add(stu3);



才不会出错，fk\_tid 允许为 null

[Table("Teacher")]

public class TeacherEntity

{

public TeacherEntity()

{

this.students = new HashSet<StudentEntity>();

}

[Key]

[Column("TeacherID")]

public int teacher\_id { get; set; }

public string tname { get; set; }

public ICollection<StudentEntity> students { get; set; }

}

[Table("Student")]

public class StudentEntity

{

[Key]

[Column("StudentID")]

public int sid { get; set; }

public string sname { get; set; }

[Column("fk\_tid")]

public int tid { get; set; }

[ForeignKey("tid")]

public TeacherEntity teacher { get; set; } - 对于实体则不需要构建函数实例化

}

注意：1) 集合实体，必须要在构建函数里初始实例化， 否则使用时会说 object is null

TeacherEntity tch1 = new TeacherEntity { tname = "Zhang Lao Shi" };

StudentEntity stu2 = new StudentEntity { sname = "Tommy" };

StudentEntity stu3 = new StudentEntity { sname = "Susan" };

tch1.students.Add(stu2); - 如果不初始化 this.students = new HashSet<StudentEntity>() 出错

tch1.students.Add(stu3); - 如果不初始化 this.students = new HashSet<StudentEntity>() 出错

this.db.Teachers.Add(tch1);

this.db.SaveChanges();

2）对于实体 ，则不需要再构造函数初始实例化。

如：public TeacherEntity teacher { get; set; }

多个表的外键关系：

[Table("Teacher")]

public class TeacherEntity

{

public TeacherEntity()

{

this.students = new HashSet<StudentEntity>();

this.subjects = new HashSet<SubjectEntity>();

}

[Key]

[Column("TeacherID")]

public int teacher\_id { get; set; }

public string tname { get; set; }

public ICollection<StudentEntity> students { get; set; }

public ICollection<SubjectEntity> subjects { get; set; }

//如果从Teacher 实体的角度出发，不需要操作集合students, subjects， 可以把这两个集合删除掉

}

[Table("Student")]

public class StudentEntity

{

[Key]

[Column("StudentID")]

public int sid { get; set; }

public string sname { get; set; }

[Column("fk\_tid")]

public int tid { get; set; }

[ForeignKey("tid")]

public TeacherEntity teacher { get; set; }

//如果不希望自动创建的外键字段， 则teacher实体是必须的，有实体才能指定外键

}

[Table("Subject")]

public class SubjectEntity

{

[Key]

[Column("SubjectID")]

public int sid { get; set; }

public string subject { get; set; }

[Column("fk\_tid")]

public int tid { get; set; }

[ForeignKey("tid")]

public TeacherEntity teacher { get; set; }

//如果不希望自动创建的外键字段， 则teacher实体是必须的，有实体才能指定外键

}



Teacher - Student one - many

Teacher - Subject one - many

一个表格对应多个属性 ， 如学生有第一主修课， 第二主修课。主要是以属性的形式出现

[Table("Student")]

public class StudentEntity

{

[Key]

[Column("StudentID")]

public int sid { get; set; }

public string sname { get; set; }

[Column("fk\_tid")]

public int tid { get; set; }

[ForeignKey("tid")]

public TeacherEntity teacher { get; set; }

public SubjectEntity first\_subject { get; set; }

public SubjectEntity second\_subject { get; set; }

}

[Table("Subject")]

public class SubjectEntity

{

[Key]

[Column("SubjectID")]

public int sid { get; set; }

public string subject { get; set; }

[Column("fk\_tid")]

public int tid { get; set; }

[ForeignKey("tid")]

public TeacherEntity teacher { get; set; }

}

在没有指定任何外键的情况下， 会自动创建外键 “first\_subject\_sid” “second\_subject\_sid”

此情况是 StudentEntity 包含两个实体first\_subject，second\_subject

SubjectEntity 没有包含实体集合



[Table("Subject")]

public class SubjectEntity

{

public SubjectEntity()

{

this.first\_students = new HashSet<StudentEntity>();

this.second\_students = new HashSet<StudentEntity>();

}

[Key]

[Column("SubjectID")]

public int sid { get; set; }

public string subject { get; set; }

[Column("fk\_tid")]

public int tid { get; set; }

[ForeignKey("tid")]

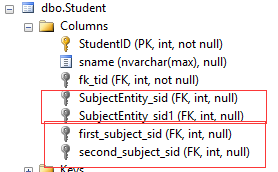
public TeacherEntity teacher { get; set; }

public ICollection<StudentEntity> first\_students { get; set; }

public ICollection<StudentEntity> second\_students { get; set; }

}

SubjectEntity 包含实体集合first\_students，second\_students



这时Student表格自动产生了 4 个外键， 其实这并不是我们想要的结果， 我们想要2个外键即可。

如何解决这个问题呢？[InverseProperty] 属性该派上用场了

[Table("Teacher")]

public class TeacherEntity

{

public TeacherEntity()

{

this.students = new HashSet<StudentEntity>();

this.subjects = new HashSet<SubjectEntity>();

}

[Key]

[Column("TeacherID")]

public int teacher\_id { get; set; }

public string tname { get; set; }

public ICollection<StudentEntity> students { get; set; }

public ICollection<SubjectEntity> subjects { get; set; }

}

[Table("Student")]

public class StudentEntity

{

[Key]

[Column("StudentID")]

public int sid { get; set; }

public string sname { get; set; }

[Column("fk\_tid")]

public int tid { get; set; }

[ForeignKey("tid")]

public TeacherEntity teacher { get; set; }

[InverseProperty("first\_students")]

public SubjectEntity first\_subject { get; set; }

[InverseProperty("second\_students")]

public SubjectEntity second\_subject { get; set; }

}

[Table("Subject")]

public class SubjectEntity

{

public SubjectEntity()

{

this.teachers = new HashSet<TeacherEntity>();

this.first\_students = new HashSet<StudentEntity>();

this.second\_students = new HashSet<StudentEntity>();

}

[Key]

[Column("SubjectID")]

public int sid { get; set; }

public string subject { get; set; }

public ICollection<TeacherEntity> teachers { get; set; }

[InverseProperty("first\_subject")]

public ICollection<StudentEntity> first\_students { get; set; }

[InverseProperty("second\_subject")]

public ICollection<StudentEntity> second\_students { get; set; }

}

SubjectEntity - [InverseProperty("first\_subject")] 对应

StudentEntity - public SubjectEntity first\_subject { get; set; }

SubjectEntity - [InverseProperty("second\_subject")] 对应

StudentEntity - public SubjectEntity second\_subject { get; set; }

[InverseProperty(实体或者实体集合名称)] – 可以指定在任何一边， 或者两边同时指定都可以

所以在 Student 表格里会只看到 2 个自动创建的外键（而不是之前的4个）

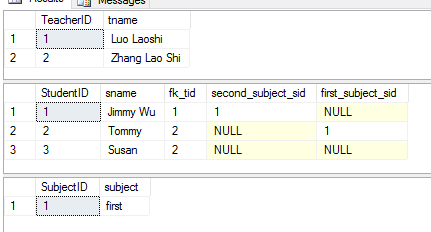


Teacher – Subject 是多对多的关系：

TeacherEntity - public ICollection<SubjectEntity> subjects { get; set; }

SubjectEntity - public ICollection<TeacherEntity> teachers { get; set; }

自动创建了中间表： 名称是 TeacherEntity + SubjectEntity



如何使用自定义的外键呢？ 注意外键是在StudentEntity 里添加的

[Table("Student")]

public class StudentEntity

{

[Key]

[Column("StudentID")]

public int sid { get; set; }

public string sname { get; set; }

[Column("fk\_tid")]

public int tid { get; set; }

[ForeignKey("tid")]

public TeacherEntity teacher { get; set; }

[Column("first\_id")]

public int first\_subject\_id { get; set; }

[Column("second\_id")]

public int second\_subject\_id { get; set; }

[ForeignKey("first\_subject\_id")]

public SubjectEntity first\_subject { get; set; }

[ForeignKey("second\_subject\_id")]

public SubjectEntity second\_subject { get; set; }

}



当我们运行时创建数据库时，会出现以上错误，外键设计到级联操作,

原因是删除SubjectEntity 会涉及到 StudentEntity 的两个外键

解决方案：

1. 设置外键可以为 null

[Column("first\_id")]

public int? first\_subject\_id { get; set; } - 注意外键是 int? 允许 null

[Column("second\_id")]

public int? second\_subject\_id { get; set; } - 注意外键是 int? 允许 null

[ForeignKey("first\_subject\_id")]

public SubjectEntity first\_subject { get; set; }

[ForeignKey("second\_subject\_id")]

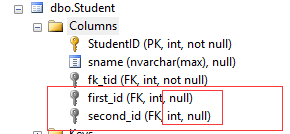
public SubjectEntity second\_subject { get; set; }

1. 级联操作：OnModelCreating(DbModelBuilder modelBuilder) { } 方法里去除级联操作

modelBuilder.Entity<StudentEntity>().HasRequired(p => p.first\_subject).WithMany().WillCascadeOnDelete(false);

modelBuilder.Entity<StudentEntity>().HasRequired(p => p.second\_subject).WithMany().WillCascadeOnDelete(false);

1. 建议用第一个方案：也就是允许null 值





* 1. One to One Relationship
     1. One to One ( 1 – 1 )

[Table("Country")]

public class CountryEntity

{

[Key,Column("country\_id")]

public int id { get; set; }

[Column("country\_name")]

public string name { get; set; }

[Column("country\_iso"), StringLength(4,MinimumLength = 2), Required]

public string iso { get; set; }

[Required]

public CountryInfoEntity countryInfo { get; set; }

}

[Table("CountryInfo")]

public class CountryInfoEntity

{

[Key,Column("country\_id")]

[ForeignKey("country")]

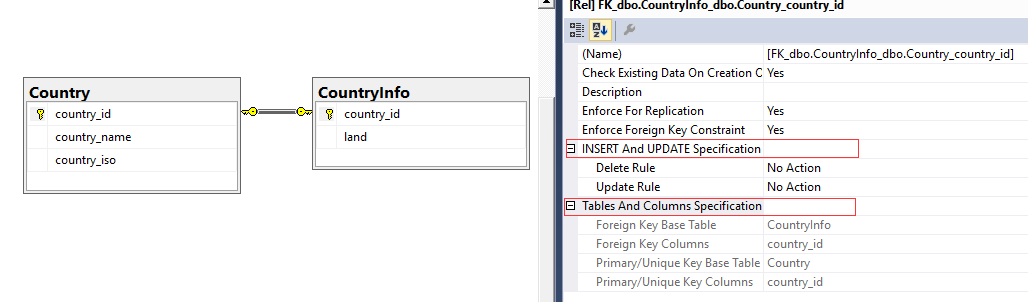
public int id { get; set; }

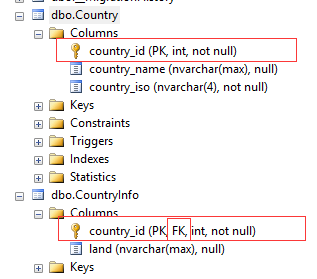
public string land { get; set; }

[Required]

public CountryEntity country { get; set; }

}





在数据库端，并不存在 1 – 1的强制关系， 只有 1 – 0..1 的关系

CountryEntity ccc = new CountryEntity { name = "China", iso = "CN" };

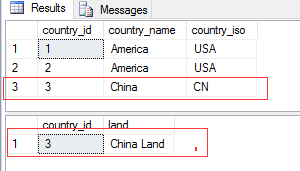
CountryInfoEntity cinfo = new CountryInfoEntity { land = "China Land" };

ccc.countryInfo = cinfo; - 这个必须的，

因为 [Required] public CountryInfoEntity countryInfo { get; set; }， 否则出错

this.db.Countries.Add(ccc);

this.db.SaveChanges();



* + 1. Master to Slave ( 1 – 0..1 )

[Table("Country")]

public class CountryEntity

{

[Key,Column("country\_id")] - 默认主键是 Identity(1,1)

public int id { get; set; }

[Column("country\_name")]

public string name { get; set; }

[Column("country\_iso"), StringLength(4,MinimumLength = 2), Required]

public string iso { get; set; }

//[Required] – 注意这里没有必须的属性

public CountryInfoEntity countryInfo { get; set; }

}

[Table("CountryInfo")]

public class CountryInfoEntity

{

[Key,Column("country\_id")] – 主键上如果同时又是外键， 默认将不再是 Identity(1,1)

[ForeignKey("country")]

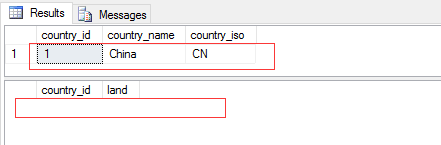
public int id { get; set; }

public string land { get; set; }

[Required]

public CountryEntity country { get; set; }

}



public class CountryInfoEntity

{

[Key,Column("country\_id"), DatabaseGenerated(DatabaseGeneratedOption.Identity)]

[ForeignKey("country")]

public int id { get; set; }

public string land { get; set; }

public CountryEntity country { get; set; }

}

注意：如果强行将主键设置为 Identity(1,1) ，不出错，数据库端也会创建， 但是在插入关系数据时，将出错

CountryEntity ccc = new CountryEntity { name = "China", iso = "CN" };

this.db.Countries.Add(ccc);

this.db.SaveChanges();

CountryInfoEntity cinfo = new CountryInfoEntity { land = "China Land" };

this.db.CountryInfos.Add(cinfo);

this.db.SaveChanges();

即使是单独插入数据也是一样出错， 因为外键与主键的自动增量是冲突的

public class CountryInfoEntity

{

[Key,Column("country\_id")]

[ForeignKey("country")]

public int id { get; set; }

public string land { get; set; }

[Required] – 不管有没有，因为主键同时也是外键，且不为空，即使客户端通过，数据库端也出错

public CountryEntity country { get; set; }

}

如果有[Required]

//CountryEntity ccc = new CountryEntity { name = "China", iso = "CN" };

CountryInfoEntity cinfo = new CountryInfoEntity { land = "China Land" };

//cinfo.country = ccc;

this.db.CountryInfos.Add(cinfo);

this.db.SaveChanges();

则错误提示： 字段 country 是必须的

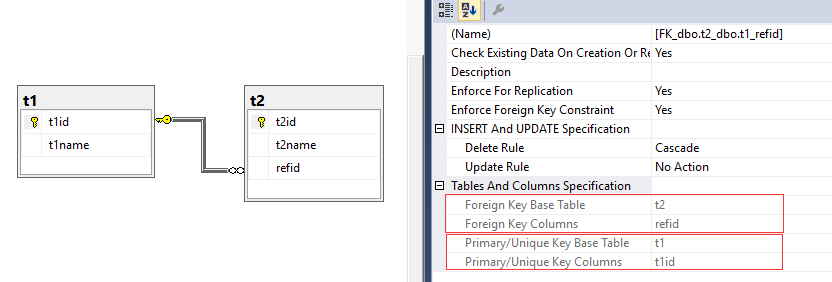
Additional information: entity:CountryInfoEntity property:country Message:The country field is

required.

如果没有指定 //[Required] 则错误来自于底层数据库操作

An error occurred while updating the entries. See the inner exception for details

* 1. 数据操作的误区



One to Many Relationship:

[Table("t1")]

public class TeacherEntity

{

public TeacherEntity()

{

this.students = new HashSet<StudentEntity>();

}

[Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]

[Column("t1id")]

public int id { get; set; }

[Column("t1name"), MaxLength(10)]

public string taname { get; set; }

public ICollection<StudentEntity> students { get; set; }

}

[Table("t2")]

public class StudentEntity

{

[Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]

[Column("t2id")]

public int id { get; set; }

[Column("t2name"),MaxLength(10)]

public string studentName { get; set; }

[Column("refid")]

public int teacher\_id { get; set; }

[ForeignKey("teacher\_id")]

public TeacherEntity teacher { get; set; }

}

TeacherEntity t = new TeacherEntity { taname = "William" };

StudentEntity s1 = new StudentEntity { studentName = "Susuan" };

StudentEntity s2 = new StudentEntity { studentName = "Tommy" };

//t.students.Add(s1);

//t.students.Add(s2);

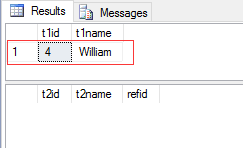
s1.teacher = t;

s2.teacher = t;

this.db.Teachers.Add(t);

this.db.SaveChanges();

此时的结果是：



TeacherEntity t = new TeacherEntity { taname = "William" };

StudentEntity s1 = new StudentEntity { studentName = "Susuan" };

StudentEntity s2 = new StudentEntity { studentName = "Tommy" };

t.students.Add(s1);

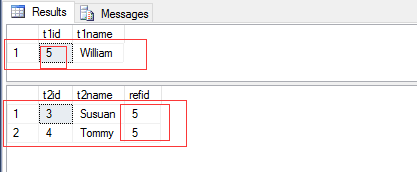
t.students.Add(s2);

//s1.teacher = t;

//s2.teacher = t;

this.db.Teachers.Add(t);

this.db.SaveChanges();



这个结果是我们所期待的， 所以可以比较一下细微的差别

也可以这样写：

TeacherEntity t = new TeacherEntity { taname = "William" };

StudentEntity s1 = new StudentEntity { studentName = "Susuan" };

StudentEntity s2 = new StudentEntity { studentName = "Tommy" };

//t.students.Add(s1);

//t.students.Add(s2);

s1.teacher = t;

s2.teacher = t;

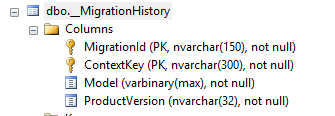
this.db.Students.Add(s1);

this.db.Students.Add(s2);

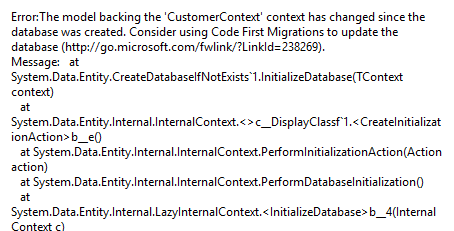
this.db.SaveChanges();

* 1. 创建数据库错误

如果我们的数据库是先用code first 创建的。则在数据库端会自动生成一个升级的表格



那么日后我们的更改了Entity Framework 的metadata ，或者使用database first 方式时

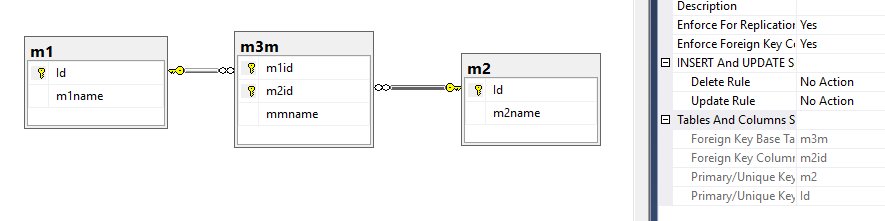


如何屏蔽升级检查呢？

方法一：**初始化器** Database.SetInitializer<CustomerContext>(null);

方法二： 在数据库端， 直接将升级表删除掉

* 1. Many to Many 使用代码映射到数据库
     1. 数据库关系图， 多对多关系需要中间表



* + 1. 创建映射关系：

public class m1

{

public m1()

{

this.m2sets = new HashSet<m2>();

}

[Key]

public int id { get; set; }

public string m1name { get; set; }

public ICollection<m2> m2sets { get; set; }

}

public class m2

{

public m2()

{

this.m1sets = new HashSet<m1>();

}

[Key]

public int id { get; set; }

public string m2name { get; set; }

public ICollection<m1> m1sets { get; set; }

}

public class m3m

{

[Key, Column(Order =1)]

protected int m1id { get; set; }

[ForeignKey("m1id")]

public m1 m3m1 { get; set; }

[Key, Column(Order =2)]

protected int m2id { get; set; }

[ForeignKey("m2id")]

public m2 m3m2 { get; set; }

}

必须使用代码来设置中间表映射，使用自定义中间表，只能通过代码来设置映射

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Entity<m1>()

.HasEntitySetName("m3m")

.HasMany(p => p.m2sets)

.WithMany(p => p.m1sets)

.Map(m => {

m.MapLeftKey("m1id");

m.MapRightKey("m2id");

m.ToTable("m3m");

});

}

public DbSet<m1> mothers { get; set; }

public DbSet<m2> children { get; set; }

m1 m11 = new m1 { m1name = "William11" };

m1 m12 = new m1 { m1name = "William12" };

m2 m21 = new m2 { m2name = "Lilian21" };

m2 m22 = new m2 { m2name = "Lilian22" };

m11.m2sets.Add(m21);

m11.m2sets.Add(m22);

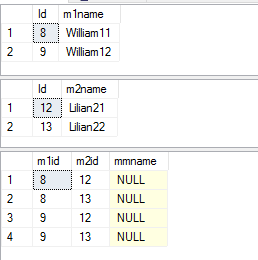
m12.m2sets.Add(m21);

m12.m2sets.Add(m22);

this.db.mothers.Add(m11);

this.db.mothers.Add(m12);

this.db.SaveChanges();



* + 1. 多对多关系， 中间表有额外其他字段，可以考虑中间表

1. 中间表也必须是一个 DBSet<中间表> , 可以利用查询找到记录，对字段赋值修改。
2. 可以把多对多关系， 拆分：左表对中间表：一对多； 右表对中间表：一对多

后面有专门的文章详细描述

* 1. [ComplexType] 复合类型

public class m1

{

public m1()

{

this.m2sets = new HashSet<m2>();

}

[Key]

public int id { get; set; }

public string m1name { get; set; }

public m1info addt { get; set; }

public ICollection<m2> m2sets { get; set; }

}

[ComplexType]

public class m1info

{

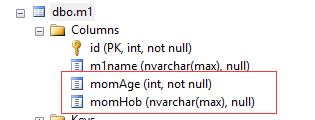
[Column("momAge")]

public int age { get; set; }

[Column("momHob")]

public string horby { get; set; }

}



如果只是单纯类

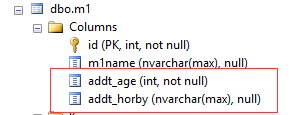
public class m1info

{

public int age { get; set; }

public string horby { get; set; }

}



如果字段有映射： [ComplexType]可有可无， 关键是映射字段名称[Column("colName")]

public class m1info

{

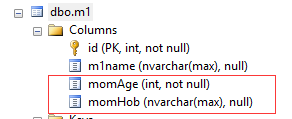
[Column("momAge")]

public int age { get; set; }

[Column("momHob")]

public string horby { get; set; }

}



* 1. 表格分裂：将一个表分成两实体或多个实体

[Table("m1")]

public class m1

{

public m1()

{

this.m2sets = new HashSet<m2>();

}

[Key]

public int id { get; set; }

public string m1name { get; set; }

[Required] - 必须有, 只是决定必须提供 m1info 实体, 并不决定字段是否是 not null

public m1info mmfo { get; set; }

public ICollection<m2> m2sets { get; set; }

}

[Table("m1")] - 必须指明同一个表

public class m1info

{

[Key]

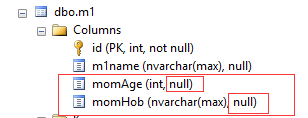
[ForeignKey("fkset")] - 必须主键和外键：即建立 1 – 1 的关系

public int fkid { get; set; }

public m1 fkset { get; set; } - 虽然没有[Required] , 但是赋值是必须的，否则出错

[Column("momAge")] - 字段是否是 not null, 由 int? 来决定

public int? age { get; set; }

 [Column("momHob")]

public string horby { get; set; }

}

m1 m11 = new m1 { m1name = "William11" };

m1info minfo = new m1info { age= 38, horby="giid" };

m11.mmfo = minfo; - 必须要赋值， 否则出错

this.db.mothers.Add(m11);

this.db.SaveChanges();

另外一种方法

m1 m11 = new m1 { m1name = "William11" };

m1info minfo = new m1info { age= 38, horby="giid" };

minfo.fkset = m11; - 必须要赋值， 否则出错

this.db.motherInfo.Add(minfo);

this.db.SaveChanges();

1. DbContext & DbSet

当我们有了实体以后，需要在数据库端进行数据库的映射：DbContext & DbSet<ClassEntity>

public class CustomerContext: DbContext

{

// static constructor only verify database schema,

// if database not exists or changed, it will drop and create database again.

static CustomerContext()

{

Database.SetInitializer<CustomerContext>(null);

//Database.SetInitializer<CustomerContext>(new CreateDatabaseIfNotExists<CustomerContext>());

//Database.SetInitializer<CustomerContext>( new CustomerContextInitilizer() );

}

public CustomerContext():base("name=CustomerContext") {

this.Database.Initialize(true);

}

public CustomerContext(string connect\_str) : base(connect\_str)

{

this.Database.Initialize(true);

}

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Entity<m1>()

.HasEntitySetName("m3m")

.HasMany(p => p.m2sets)

.WithMany(p => p.m1sets)

.Map(m => {

m.MapLeftKey("m1id");

m.MapRightKey("m2id");

m.ToTable("m3m");

});

base.OnModelCreating(modelBuilder);

//modelBuilder.Filter("customerFilter", (Customer c, string match) => (c.CustomerName.StartsWith(match)), "T");

}

public DbSet<CountryEntity> Countries { get; set; }

public DbSet<CountryInfoEntity> CountryInfos { get; set; }

public DbSet<Customer> Customers { get; set; }

public DbSet<TeacherEntity> Teachers { get; set; }

public DbSet<StudentEntity> Students { get; set; }

public DbSet<m1> mothers { get; set; }

public DbSet<m1info> motherInfo { get; set; }

public DbSet<m2> children { get; set; }

}

1. Dynamic Filter – 动态过滤

NuGet Package: [EntityFramework.DynamicFilters](https://www.nuget.org/packages/EntityFramework.DynamicFilters)

using EntityFramework.DynamicFilters;

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Filter("customerFilter",

(Customer c, string match) => (c.CustomerName.StartsWith(match)), "T");

base.OnModelCreating(modelBuilder);

}

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Filter("customerFilter",

(Customer c, string match) => (c.CustomerName.StartsWith(match)), "T");

modelBuilder.Filter("customerFilter1", (Customer c) => c.sta, 25);

modelBuilder.DisableFilterGlobally("customerFilter1"); - 定义完，可以直接失效

base.OnModelCreating(modelBuilder);

}

* 1. Define filters

modelBuilder.Filter("IsDeleted", (ISoftDelete d) => d.IsDeleted, false);

修改变量值

context.SetFilterScopedParameterValue("IsDeleted", true);

modelBuilder.Filter("Notes\_CurrentUser",

(Note n) => n.PersonID,

() => GetPersonIDFromPrincipal(Thread.CurrentPrincipal)

);

modelBuilder.Filter("Notes\_CurrentUser",

(Note n) => n.PersonID,

(MyContext ctx) => ctx.CurrentPersonID

);

多个变量:这里是两个变量 accountID, isDeleted

modelBuilder.Filter("BlogEntryFilter",

(BlogEntry b, Guid accountID, bool isDeleted) =>

(b.AccountID == accountID) && (b.IsDeleted == isDeleted),

() => GetPersonIDFromPrincipal(Thread.CurrentPrincipal),

() => false);

修改变量值

context.SetFilterScopedParameterValue("BlogEntryFilter", "accountID", 12345);

var values = new List<int> { 1, 2, 3, 4, 5 };

modelBuilder.Filter("ContainsTest",

(BlogEntry b, List<int> valueList) => valueList.Contains(b.IntValue.Value),

() => values

);

* 1. Enable & Disable Filters

To disable a filter, use the DisableFilter extension method like this:

context.DisableFilter("IsDeleted");

Filters can also be globally disabled after they are created in OnModelCreating:

modelBuilder.DisableFilterGlobally("IsDeleted");

You can also mass enable/disable all filters within a DbContext at once:

context.DisableAllFilters();

context.EnableAllFilters();

If you will never require the need to enable or disable filters at any time during

the application life cycle, you can prevent this condition entirely using these 2

methods: 如果你想过滤器在程序运行期间，不能被激活和失效

// disable a single filter

modelBuilder.PreventDisabledFilterConditions("IsDeleted");

// disable all filters defined up to calling this

modelBuilder.PreventAllDisabledFilterConditions();

* 1. Conditonal Enable Filters: 有条件激活过滤器：

modelBuilder.EnableFilter("BlogEntryFilter", () => !UserIsAdmin(Thread.CurrentPrincipal) );

Func<bool>

* 1. 动态过滤器的使用注意：

但需要注意的是，如果在同一个上下文DbContext实例中，运用过虑器之前，过虑器有被禁用过，而数

据被缓存时，过滤器就不会起任何效果，所有使用时，你一定要避免在同一个上下文中因更改过滤器而

影响结果的情况。

也就是说：不能把过滤器，当做查询过滤来使用， 因为数据被缓存以后就不会有任何效果。过滤器最好设定好一次就不再更改，

也不再Enable/Disable. 而是做为一种业务逻辑来使用， 如对记录进行软删除（使用标识字段来标记删除）

1. DBContext 的构造函数
   1. 构造函数的连接数据库

<connectionStrings> - 在 App.config 或者 Web.config里指定连接字符串

<add name="CustomerContext"

providerName="System.Data.SqlClient"

connectionString="Server=(LocalDB)\sohu;

Initial Catalog=MyOrder;

Trusted\_Connection=True;

Integrated Security=True;

AttachDbFileName=D:\Temp12\history.mdf" />

</connectionStrings>

public CustomerContext():base("name=CustomerContext") { - 默认使用连接字段串的名称

}

public CustomerContext(string connect\_str) : base(connect\_str)

{}

* 1. Database Initialization

using System.Data.Entity;

There are four different database initialization strategies:

1. **CreateDatabaseIfNotExists:** This is **default** initializer. As the name suggests, it will create the database if none exists as per the configuration. However, if you change the model class and then run the application with this initializer, then it will throw an exception.
2. **DropCreateDatabaseIfModelChanges:** This initializer drops an existing database and creates a new database, if your model classes (entity classes) have been changed. So you don't have to worry about maintaining your database schema, when your model classes change.
3. **DropCreateDatabaseAlways:** As the name suggests, this initializer drops an existing database every time you run the application, irrespective of whether your model classes have changed or not. This will be useful, when you want fresh database, every time you run the application, like while you are developing the application.
4. **Custom DB Initializer:** You can also create your own custom initializer, if any of the above doesn't satisfy your requirements or you want to do some other process that initializes the database using the above initializer.
5. **Null :** 不执行任何初始化任务

public CustomerContext():base("CustomerContext")

{ }

public CustomerContext(string connect\_str) : base(connect\_str)

{ }

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{ }

如果没有代码，则默认执行的是**CreateDatabaseIfNotExists**

* 不执行任何初始化：

Database.SetInitializer<CustomerContext>(null);

this.db = new CustomerContext();

* 如果数据库不存在则创建数据库：

Database.SetInitializer<CustomerContext>(new CreateDatabaseIfNotExists<CustomerContext>());

this.db = new CustomerContext();

* 自定义数据库初始化：需要继承上面 1 – 3 的任何一种情况

public class **CustomerContextInitilizer** : DropCreateDatabaseIfModelChanges<CustomerContext>

{

// 覆盖填充数据的方法，即可为数据库的表添加初始化数据

protected override void Seed(CustomerContext context)

{ // 还可以执行一些数据库操作

context.Database.ExecuteSqlCommand("CREATE TABLE LWH(id int, lwhname varchar(20))");

IList<Customer> custList = new List<Customer>();

custList.Add(new Customer() { CustomerName="Test Test",

phone="604-888-8888", sta=15,

email = "william@hotmail.com"

});

foreach (Customer cust in custList)

context.Customers.Add(cust);

base.Seed(context);

}

}

Database.SetInitializer<CustomerContext>(new **CustomerContextInitilizer**());

this.db = new CustomerContext();

this.db.Database.Initialize(true);

In the above code snippet you are calling the Initialize() method immediately after creating a context instance. In this case, the database will be created immediately after calling the Initialize() method instead of waiting until the context is used for the first time. The Initialize() method takes a boolean parameter that controls whether the initialization process should re-run if it has already run for the application. Specifying false will skip the initialization process if it has already executed. A value of true will initialize the database again even if it was already initialized.

* 1. 如何创建简洁的数据库初始化。

using System.Data.Entity.ModelConfiguration.Conventions;

public class CustomerContext: DbContext

{

// static constructor only verify database schema,

// if database not exists or changed, it will drop and create database again.

static CustomerContext() - 静态构造函数，只是当第一个实例化时执行一次，其后的实例化对象不再执行

{

Database.SetInitializer<CustomerContext>(null);

Database.SetInitializer<CustomerContext>(new CreateDatabaseIfNotExists<CustomerContext>());

Database.SetInitializer<CustomerContext>(new DropCreateDatabaseAlways<CustomerContext>());

Database.SetInitializer<CustomerContext>(new DropCreateDatabaseIfModelChanges<CustomerContext>());

Database.SetInitializer<CustomerContext>( new CustomerContextInitilizer() );

初始化数据库策略只能选一个

}

public CustomerContext():base("CustomerContext") {

this.Database.Initialize(true); - 每次实例化时即初始化Context，不用等待调用时在初始化

}

public CustomerContext(string connect\_str) : base(connect\_str)

{

this.Database.Initialize(true);

}

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{ // 可以添加或者删除， 数据库关系的级联操作

modelBuilder.Conventions.Add<OneToManyCascadeDeleteConvention>();

modelBuilder.Entity<m1>().HasEntitySetName("m3m")

.HasMany(p => p.m2sets).WithMany(p => p.m1sets)

.Map(m => {

m.MapLeftKey("m1id");

m.MapRightKey("m2id");

m.ToTable("m3m");

});

modelBuilder.Filter("customerFilter",

(Customer c, string match) => (c.CustomerName.StartsWith(match)),

"T");

modelBuilder.Filter("customerFilter1", (Customer c) => c.sta, 25);

//modelBuilder.PreventAllDisabledFilterConditions();

//modelBuilder.DisableFilterGlobally("customerFilter1");

base.OnModelCreating(modelBuilder);

}

public DbSet<CountryEntity> Countries { get; set; }

public DbSet<CountryInfoEntity> CountryInfos { get; set; }

public DbSet<Customer> Customers { get; set; }

public DbSet<TeacherEntity> Teachers { get; set; }

public DbSet<StudentEntity> Students { get; set; }

public DbSet<m1> mothers { get; set; }

public DbSet<m1info> motherInfo { get; set; }

public DbSet<m2> children { get; set; }

}

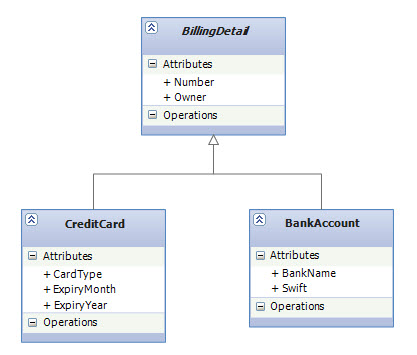
* 1. DbSet<> 需要注意的事项

public DbSet<memberEntity> Members { get; set; }

public DbSet<memberEntity> Member1s { get; set; } – 同一个实体类型不能定义多个实体对象集合



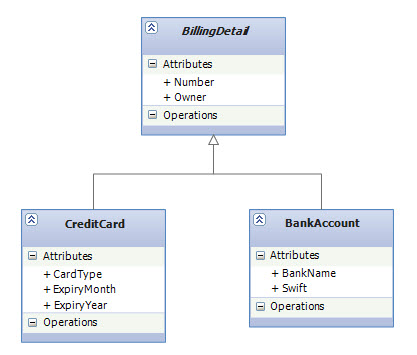
1. Inheritance with EF Code First:

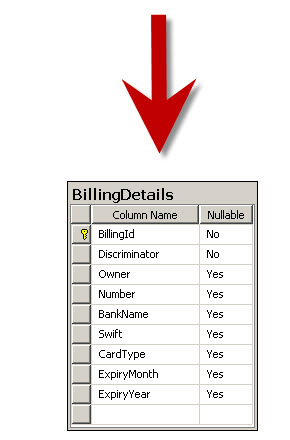


同样的类继承模型， 可以创建出有三种不同的 ”数据库表格” 结果， 因为在数据库端，表格之只有关系(即外键关联)， 而没有继承关系.

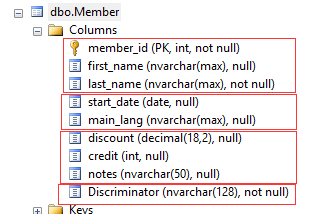
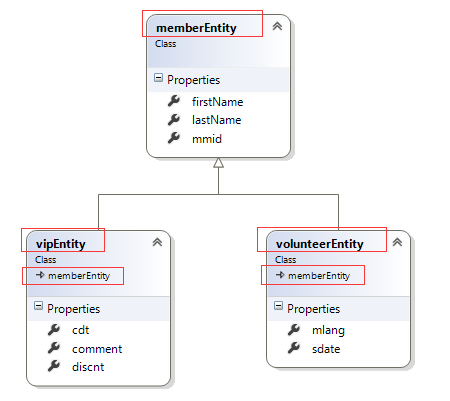
所以在C# Entity Framework Entity Class 之间是有类的继承关系的， 所以如何对应到数据库定义，这里有三种方式可以选择

* 1. Table per Hierarchy (TPH)





* + 1. 使用Entity Framework Code First



在 C# Entity Framework 我们创建了 3 个Entities, 但是在数据库端却只有一个表

public class CustomerContext: DbContext

{

public DbSet<memberEntity> Members { get; set; }

}

[Table("Member")]

public class **memberEntity** - 可以是抽象类，也可以是普通类

{

[Key, Column("member\_id", Order =1)]

public int mmid { get; set; }

[Column("first\_name", Order =2)]

public string firstName { get; set; }

[Column("last\_name", Order =3)]

[Required]

public string lastName { get; set; }

}

public class vipEntity : **memberEntity** - 继承自**memberEntity**, 且没有指明其他[Table]名称

{

[Column("discount", Order = 30)]

public decimal discnt { get; set; }

[Column("credit", Order =31)]

[Required]

public int cdt { get; set;}

[Column("notes", Order = 32), MaxLength(50)]

public string comment { get; set; }

}

public class volunteerEntity: **memberEntity** - 继承自**memberEntity**, 且没有指明其他[Table]名称

{

[Column("start\_date", Order = 20, TypeName ="Date")]

public DateTime sdate { get; set; }

[Column("main\_lang", Order = 21)]

public string mlang { get; set; }

}

添加数据：

memberEntity m1 = new memberEntity { firstName = "William", lastName = "Liu"};

this.db.Members.Add(m1);

vipEntity v1 = new vipEntity { lastName = "Su", discnt = (decimal)10.58, comment = "Good Vip" };

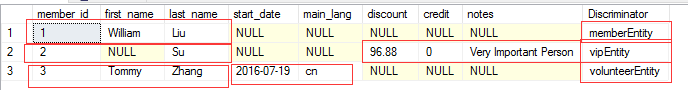
this.db.Members.Add(v1);

volunteerEntity volunt = new volunteerEntity { firstName = "Tommy", mlang = "cn", sdate = DateTime.Now, lastName =

"Zhang" };

this.db.Members.Add(volunt);

this.db.SaveChanges();



注意： 1） Discriminator 字段是自动创建的

修改数据：OfType<vipEntity>() - 通过OfType<>() 来访问继承的类，必须兼容的，如果不是继承关系则出错

vipEntity ve = this.db.Members.OfType<vipEntity>().FirstOrDefault();

if(ve != null)

{

ve.firstName = "VIP Susan";

ve.lastName = "Vip Last";

ve.discnt = (decimal)96.88;

ve.comment = "Very Important Person";

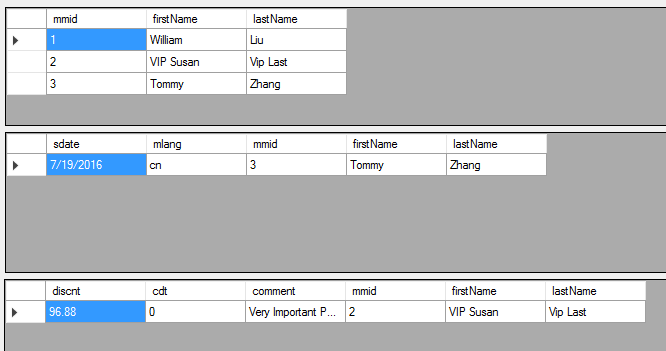
this.db.SaveChanges();

}

this.dataGridView1.DataSource = this.db.Members.OfType<memberEntity> ().ToList(); -可以不用OfType

this.dataGridView2.DataSource = this.db.Members.OfType<volunteerEntity>().ToList();

this.dataGridView3.DataSource = this.db.Members.OfType<vipEntity>().ToList();



继承类：也可以直接创建实体集合来操作，但是不会影响数据库创建表格，仍然是一个表格Member

public DbSet<memberEntity> Members { get; set; }

public DbSet<vipEntity> Vips { get; set; }

public DbSet<volunteerEntity> Volunteers { get; set; }

memberEntity m1 = new memberEntity { firstName = "William", lastName = "Liu"};

this.db.Members.Add(m1);

vipEntity v1 = new vipEntity { lastName = "Su", discnt = (decimal)10.58, comment = "Good Vip" };

this.db.Vips.Add(v1);

volunteerEntity volunt = new volunteerEntity { firstName = "Tommy", mlang = "cn", sdate =

DateTime.Now, lastName = "Zhang" };

this.db.Volunteers.Add(volunt);

this.db.SaveChanges();

其他操作是一样的， 无论使用哪个实体集合

vipEntity ve = this.db.Members.OfType<vipEntity>().FirstOrDefault();

if(ve != null)

{

ve.firstName = "VIP Susan";

ve.lastName = "Vip Last";

ve.discnt = (decimal)96.88;

ve.comment = "Very Important Person";

this.db.SaveChanges();

}

this.dataGridView1.DataSource = this.db.Members.ToList();

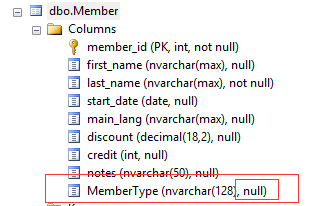
this.dataGridView2.DataSource = this.db.Vips.ToList();

this.dataGridView3.DataSource = this.db.Volunteers.ToList();

* + 1. 使用Fluent API 定义 TPH : 我们使用的是 MemberType 来替换Discriminator 字段

[Table("Member")]

public class memberEntity

 {

[Key, Column("member\_id", Order =1)]

public int mmid { get; set; }

[Column("first\_name", Order =2)]

public string firstName { get; set; }

[Column("last\_name", Order =3)]

public string lastName { get; set; }

}

public class vipEntity : memberEntity

{

[Column("discount", Order = 30)]

public decimal discnt { get; set; }

[Column("credit", Order =31)]

public int cdt { get; set;}

[Column("notes", Order = 32),MaxLength(50)]

public string comment { get; set; }

}

public class volunteerEntity: memberEntity

{

[Column("start\_date", Order = 20, TypeName ="Date")]

public DateTime sdate { get; set; }

[Column("main\_lang", Order = 21)]

public string mlang { get; set; }

}

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

modelBuilder.Entity<memberEntity>()

.Map<vipEntity>(m => m.Requires("MemberType").HasValue("Vip Person"))

.Map<volunteerEntity>(m => m.Requires("MemberType").HasValue("Volunteer Person"));

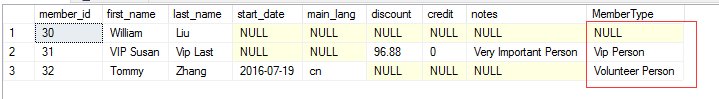
}

modelBuilder.Entity<memberEntity>()

.HasDiscriminator<string>("MemberType") - 没有这个？？？

.HasValue<vipEntity>("Vip Person")

.HasValue<volunteerEntity>("Volunteer Person");



注意:MemberType 为什么要定义为 nullable, 否则not null , 指定default 值

memberEntity m1 = new memberEntity { firstName = "William", lastName = "Liu" };

this.db.Members.Add(m1);

//如果 MemeberType 不是 null， 则出错，如果上图结果， 如果指定not null,则最好指定default值

vipEntity v1 = new vipEntity { lastName = "Su", firstName="Ling",

discnt = (decimal)10.58, comment = "Good Vip" };

this.db.Members.Add(v1);

//this.db.Vips.Add(v1);

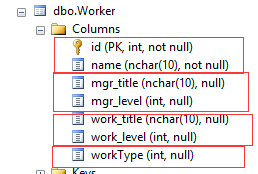
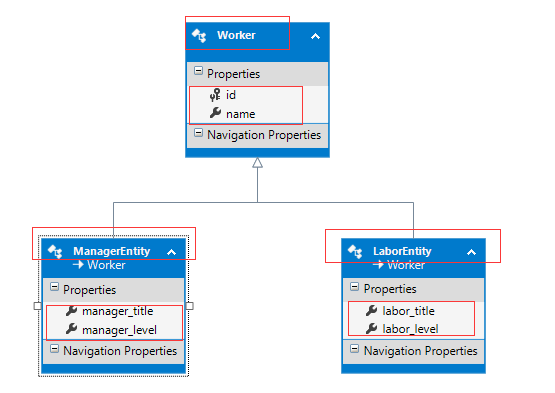
volunteerEntity volunt = new volunteerEntity { firstName = "Tommy", mlang = "cn",

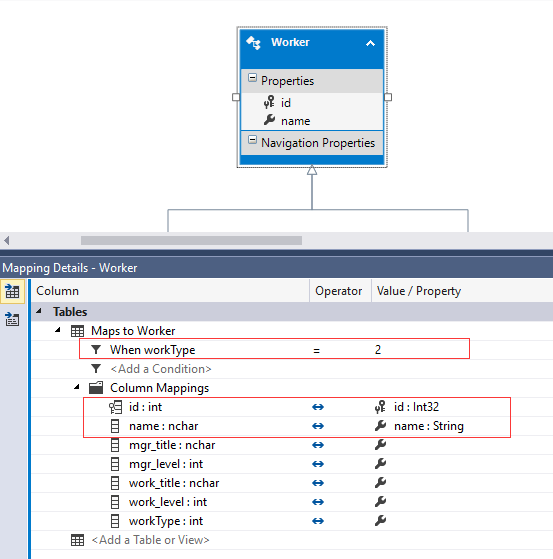
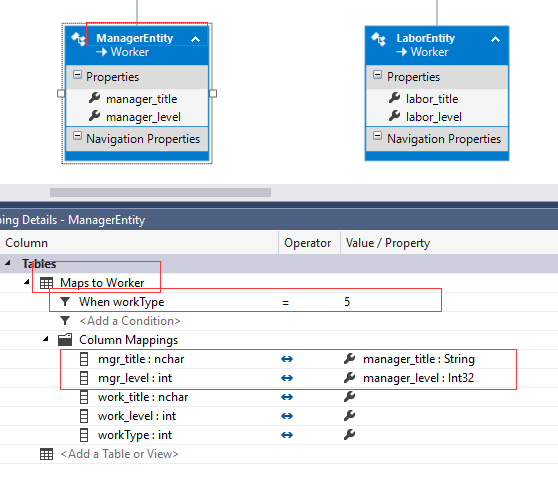
sdate = DateTime.Now, lastName = "Zhang" };

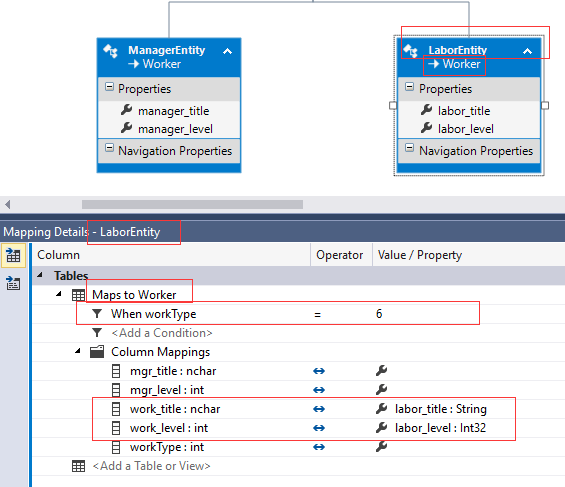
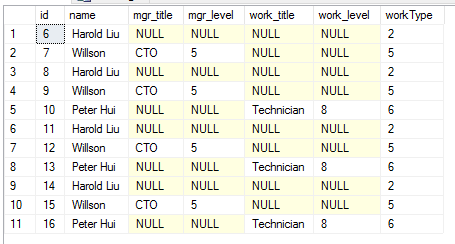
this.db.Members.Add(volunt);

this.db.SaveChanges();

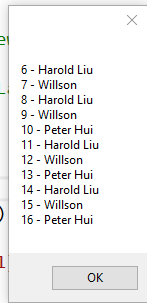
* + 1. 使用Entity Framework Database First Model Designer

注意

1. 对于 base class : Worker 可以是abstract = true | false, 如果不是抽象类，则可以有实体集合
2. base class: worker 的条件 workType =2, 对实体集合 DbSet<Worker> 不起过滤作用

MyOrder myo = new MyOrder();

string str = string.Empty;

foreach (Worker el in myo.Workers)

{

str += string.Format("{0} - {1}\n", el.id, el.name);

}

MessageBox.Show(str);

1. 继承类的条件：workType = 5 , 6 是起到过滤作用， 只有符合条件的集合列出
2. workType=2,5,6 对于新增数据时， 会赋予相应的值到数据库。而且代码也不允许直接对它赋值
3. base class 可以用来操作继承类的集合， 但是会失去继承类里的信息

string str = string.Empty;

foreach (Worker el in myo.Workers.OfType<LaborEntity>() )

{

str += string.Format("{0} - {1}\n", el.id, el.name);

}

MessageBox.Show(str);

1. DbContext 的 Set<xxEntity>, Entry 的使用

foreach (Worker el in myo.Set<Worker>() )

{

str += string.Format("{0} - {1} - {2}\n", el.id, el.name, myo.Entry(el).State);

//myo.Entry(el).Reload();

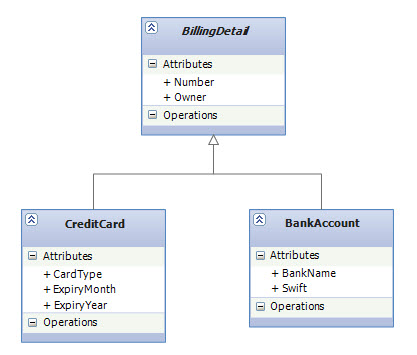
}

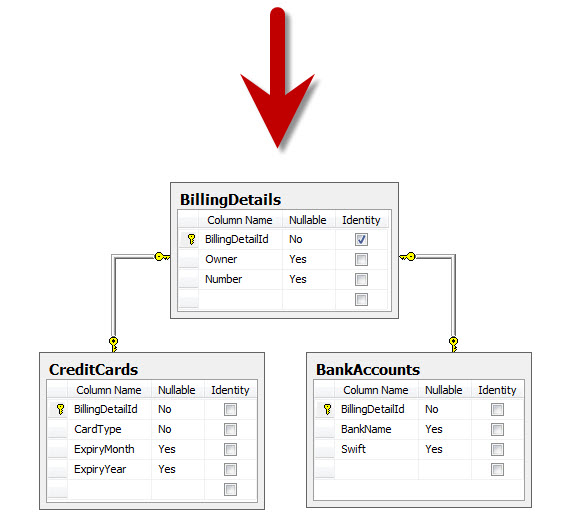
Worker fw = myo.Workers.FirstOrDefault();

fw.name = "Modify Me";

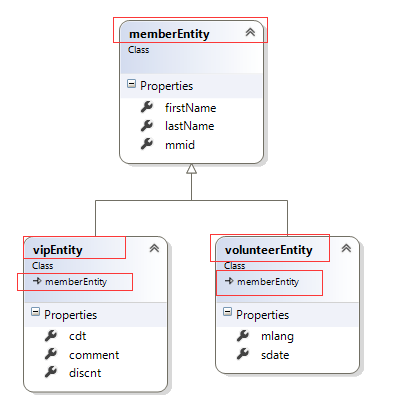
myo.Entry(fw).State = EntityState.Unchanged;

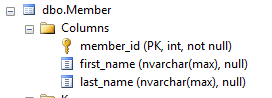
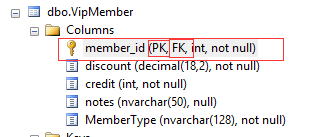
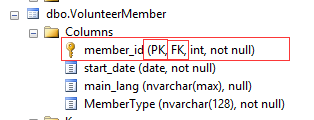
* 1. Table per Type (TPT)





* + 1. Entity Framework Code First

在 C# Entity Framework 我们创建了 3 个Entities, 但是在数据库端也创建了3个表，并且继承的实体类在数据库表自动创建主键和外键， 而且主键和外键是合一的

注意： 不需要Discriminator 字段或者自定义的Discriminator 字段如：memberType

会在继承表里，自动创建（主键+外键）和基表一样的主键名称。

这种继承关系，和 1 – 1 关系类似但是又不同，

在1-1关系里，必须要自己定义主键和外键, 另外类之间没有继承关系

[Table("Member")]

public class memberEntity

{

[Key, Column("member\_id", Order =1)]

public int mmid { get; set; }

[Column("first\_name", Order =2)]

public string firstName { get; set; }

[Column("last\_name", Order =3)]

public string lastName { get; set; }

}

[Table("VipMember")]

public class vipEntity : memberEntity

{

[Column("discount", Order = 30)]

public decimal discnt { get; set; }

[Column("credit", Order =31)]

public int cdt { get; set;}

[Column("notes", Order = 32), MaxLength(50)]

public string comment { get; set; }

}

[Table("VolunteerMember")]

public class volunteerEntity: memberEntity

{

[Column("start\_date", Order = 20, TypeName ="Date")]

public DateTime sdate { get; set; }

[Column("main\_lang", Order = 21)]

public string mlang { get; set; }

}

数据操作：

vipEntity v1 = new vipEntity { discnt = (decimal)10.68, comment = "Good VipPerson" };

this.db.Members.Add(v1);

volunteerEntity t1 = new volunteerEntity { firstName = "Todu", lastName = "Sha", mlang = "CN",

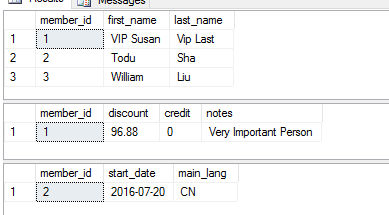
sdate = DateTime.Now };

this.db.Members.Add(t1);

memberEntity m1 = new memberEntity { firstName = "William", lastName = "Liu" };

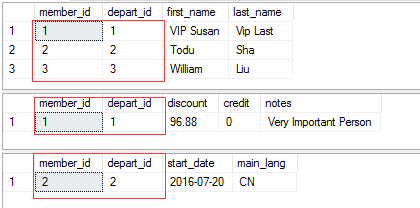
this.db.Members.Add(m1);

this.db.SaveChanges();



复合主键也是自动维护关系的

[Table("Member")]

public class memberEntity

{

[Key, Column("member\_id", Order =1)]

public int mmid { get; set; }

[Key, Column("depart\_id", Order = 2)]

public int ddid { get; set; }

[Column("first\_name", Order =3)]

public string firstName { get; set; }

[Column("last\_name", Order =4)]

public string lastName { get; set; }

}

* + 1. Fluent API

protected override void OnModelCreating(DbModelBuilder modelBuilder)

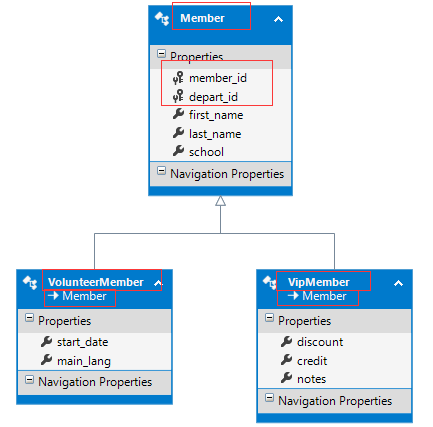
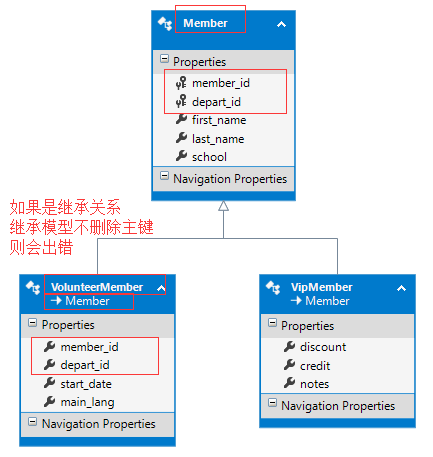
{

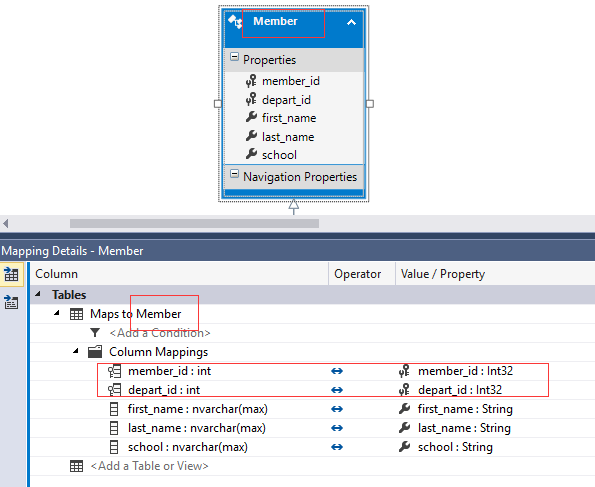
modelBuilder.Entity<vipEntity>().ToTable("VipMember");

modelBuilder.Entity<volunteerEntity>().ToTable("VolunteerMember");

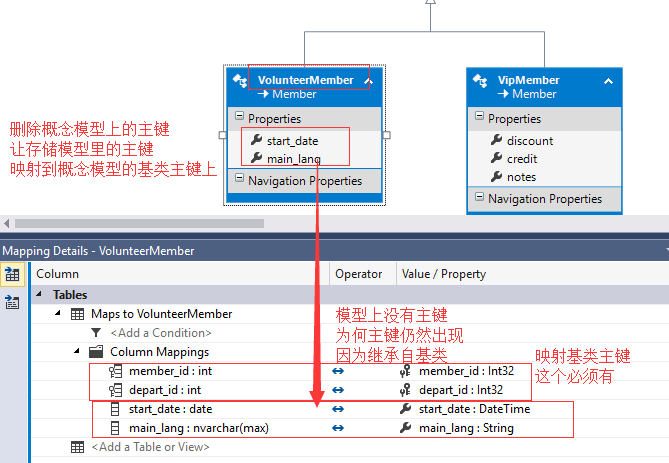
}

* + 1. Model First

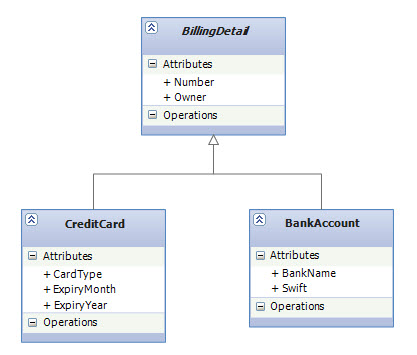
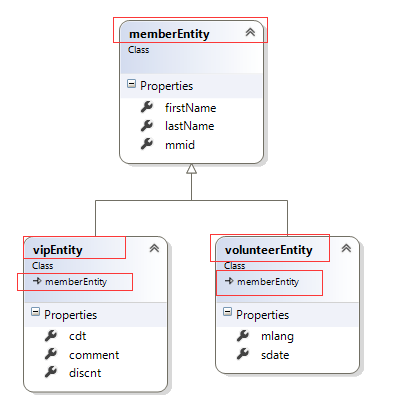
  

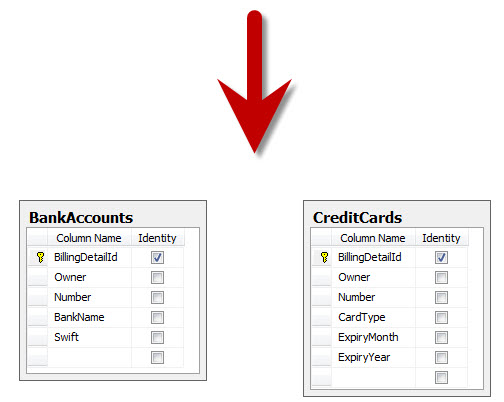


继承模型的映射关系



* 1. Table per Concrete Class (TPC) - 实际应用来说，TPC用处不大，数据冗余大





* + 1. Entity Framework Code First

public class memberEntity

{

[Key, Column("member\_id", Order =1)]

public int mmid { get; set; }

[Column("first\_name", Order =3)]

public string firstName { get; set; }

[Column("last\_name", Order =4)]

public string lastName { get; set; }

public string school { get; set; }

}

[Table("PPPMember")]

public class vipEntity : memberEntity

{

[Column("discount", Order = 30)]

public decimal discnt { get; set; }

[Column("credit", Order =31)]

public int cdt { get; set;}

[Column("notes", Order = 32), MaxLength(50)]

public string comment { get; set; }

public string school { get; set; }

}

[Table("VVVMember")]

public class volunteerEntity: memberEntity

{

[Column("start\_date", Order = 20, TypeName ="Date")]

public DateTime sdate { get; set; }

[Column("main\_lang", Order = 21)]

public string mlang { get; set; }

}

public class CustomerContext: DbContext

{

// public DbSet<memberEntity> BaseMembers { get; set; }

public DbSet<vipEntity> VipMembers { get; set; }

public DbSet<volunteerEntity> VoluteerMembers { get; set; }

}

这样的方案： 不是好的结构

1. 数据库会产生 2 个表， 客户端则产生 2 个集合。 这不是好的结构
2. 如果加上public DbSet<memberEntity> BaseMembers,则数据库3个表， 客户端 3 个集合

正确的 TPC 方案：

public abstract class memberEntity

{

[Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]

[Column("mem\_id")]

public int mmid { get; set; }

[Column("first\_name", Order =3)]

public string firstName { get; set; }

[Column("last\_name", Order =4)]

public string lastName { get; set; }

}

[Table("PPPMember")]

public class vipEntity : memberEntity

{

[Column("discount", Order = 30)]

public decimal discnt { get; set; }

[Column("credit", Order =31)]

public int cdt { get; set;}

[Column("notes", Order = 32), MaxLength(50)]

public string comment { get; set; }

}

[Table("VVVMember")]

public class volunteerEntity: memberEntity

{

[Column("start\_date", Order = 20, TypeName ="Date")]

public DateTime sdate { get; set; }

[Column("main\_lang", Order = 21)]

public string mlang { get; set; }

}

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{ // 如果不设置成 TPC, 则会创建 TPT

modelBuilder.Entity<vipEntity>().Map(m => { m.MapInheritedProperties(); });

modelBuilder.Entity<volunteerEntity>().Map(m = { m.MapInheritedProperties(); });

}

public class CustomerContext: DbContext

{ // 设置成 TPC, 数据库就不创建基表

public DbSet<memberEntity> BaseMembers { get; set; }

}

新增数据： 通过抽象类的集合来统一添加数据。 由于客户端两个实体共享基类集合， 所有主键会有出现重

复的情况，如果出现则出错。

vipEntity v1 = new vipEntity { firstName = "VVVPPP", lastName = "VVPPLLL", cdt = 120, discnt =

(decimal)89.87, comment = "VIP"};

this.db.BaseMembers.Add(v1);

volunteerEntity t1 = new volunteerEntity { firstName = "VVTTT1", lastName = "VVTTLL", mlang = "CN",

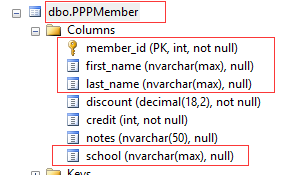
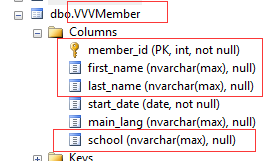
sdate = DateTime.Now };

this.db.BaseMembers.Add(t1);

this.db.SaveChanges();

注意：

1. 以上不定义基类的集合， 这样就不会在数据库端创建该表。
2. 继承的表格，会完整创建表格，包括基表的主键和所有字段

1. 基表实体上有没有[Table] ,都没关系，只要不在数据库端生成表格即可。

[Table("BaseMember")]

public class memberEntity

1. 如果想主键是Identity , 必须在基表指定, 否则继承表的主键不会是Identity

[Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]

1. 在客户端， 基表是被两个数据模型公用的，所以在前端操作时会出现键值重复错误

memberEntity m1 = new memberEntity { firstName = "AAA", lastName = "AAALL"};

this.db.BaseMembers.Add(m1);

//this.db.SaveChanges();

vipEntity v1 = new vipEntity { firstName = "VVVPPP", lastName = "VVPPLLL", cdt = 120,

discnt = (decimal)89.87, comment = "VIP"};

this.db.BaseMembers.Add(v1);

//this.db.SaveChanges();

volunteerEntity t1 = new volunteerEntity { firstName = "VVTTT1", lastName = "VVTTLL",

mlang = "CN", sdate = DateTime.Now };

this.db.BaseMembers.Add(t1);

this.db.SaveChanges();

重要的注意：

1. this.db.SaveChanges()是否能成功， 取决于运气， 是否三者产生的 Identity 有重复 ,

因为在数据库端是三个不同的表， Identity 也是由三个表各自产生。 有很大概率重复。

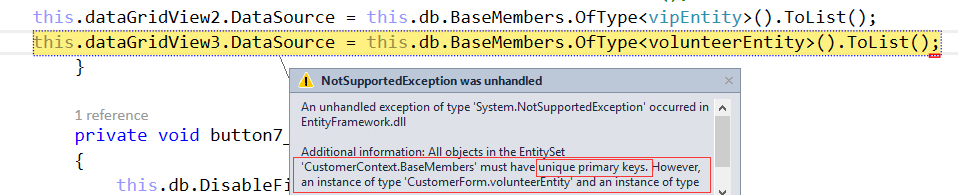
这样数据回到客户端的实体集合时， 基类的主键就会有机会出现重复值而出错。

1. 另外当我们读取数据时，情况也相同， 如下

this.dataGridView2.DataSource = this.db.BaseMembers.OfType<vipEntity>().ToList();

this.dataGridView3.DataSource = this.db.BaseMembers.OfType<volunteerEntity>().ToList();

错误提示如下：



* + 1. Fluent API

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

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

{

m.MapInheritedProperties();

m.ToTable("VIPMember"); - 优先级别比 [Table("PPMM")] 高

});

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

{

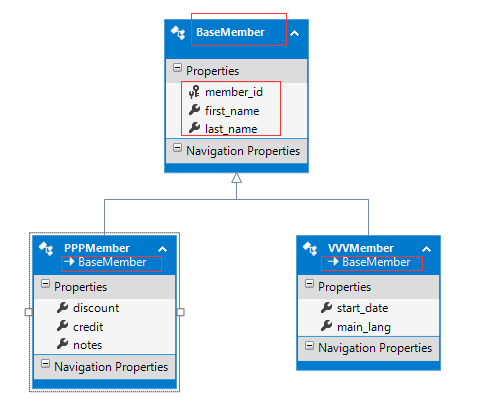
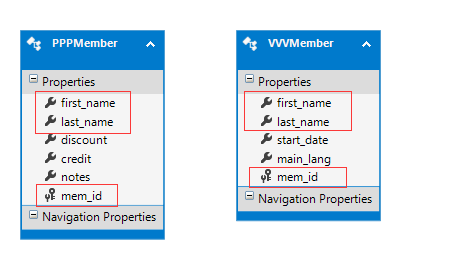
m.MapInheritedProperties();

m.ToTable("VVVMember");

});

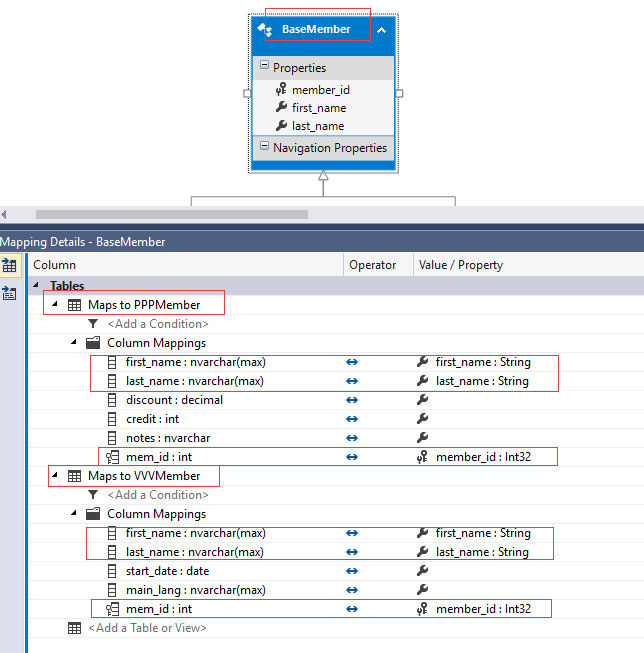
}

* + 1. Model Designer

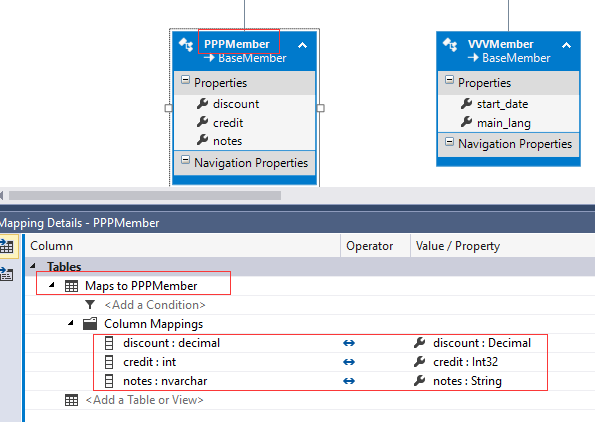
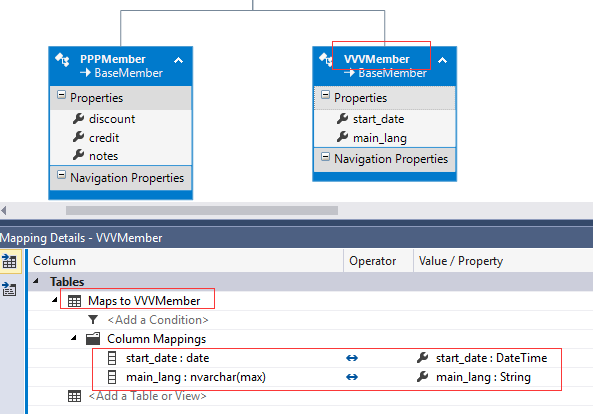


步骤：

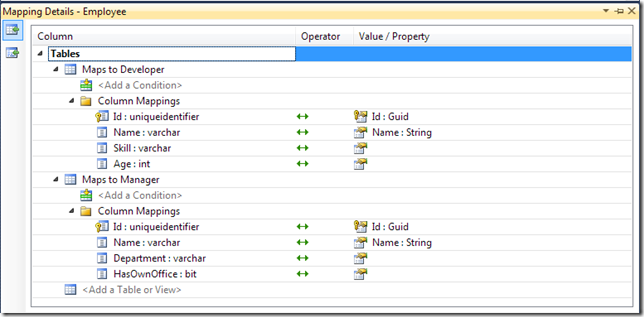
1. 手工添加 Base Entity , 将这两个表的 Base Type 改成新建的 Base Entity, 两个表的公共字段删除.
2. 将基表实体的字段，同时映射到两个继承表的存储模型上。



1. 剩下继承表实体，保留默认映射：

Hit compile and…. it doesn’t work. The first problem you’ll get is that you need to map the Id and Name properties from the base Employee entity to both physical tables. How do we do that? Well, you can try to go to mapping of Employee in the designer and doing the following:

[](https://cockneycoder.files.wordpress.com/2010/11/image1.png)

It won’t work. You still need to map the unique identifier column (ID) to both the Developer and Manager entities – but you can’t do this in the designer.

So you open up the model in the XML Editor (Open with…) and you get a whole load of XML.

* **SSDL**: The database defintions; you won’t need to modify this.
* **CSDL**: The conceptual model i.e. the entities you see in the designer. You won’t need to modify this.
* **CS Mapping**: The mapping between the above two models.
* **Designer content**: Ignore this; it contains the designer surface details e.g. where on the screen the entities should be displayed etc.

So, you go into the CS Mapping section, and create a new scalar property in both Manager and Developer mapping fragments for ID e.g.

<EntityContainerMapping StorageEntityContainer="TestModelStoreContainer" CdmEntityContainer="TestEntities">

<EntitySetMapping Name="Employees">

<EntityTypeMapping TypeName="IsTypeOf(TestModel.Developer)">

<MappingFragment StoreEntitySet="Developer">

<ScalarProperty Name="Skill" ColumnName="Skill" />

<ScalarProperty Name="Age" ColumnName="Age" />

</MappingFragment>

</EntityTypeMapping>

becomes

<EntityContainerMapping StorageEntityContainer="TestModelStoreContainer" CdmEntityContainer="TestEntities">

<EntitySetMapping Name="Employees">

<EntityTypeMapping TypeName="IsTypeOf(TestModel.Developer)">

<MappingFragment StoreEntitySet="Developer">

**<ScalarProperty Name="Id" ColumnName="Id" />**

<ScalarProperty Name="Skill" ColumnName="Skill" />

<ScalarProperty Name="Age" ColumnName="Age" />

</MappingFragment>

</EntityTypeMapping>

OK, let’s try to compile again. Still no luck – VS now complains about the lack of the mapping for the Name property, too! In other words:

**you need to manually map across all shared properties from the base class to all derived classes**

Once you create another mapping for Name (just like I did for ID), your code will compile, and you can do code like the following: –

var context = new TestEntities();

context.AddToEmployees(new Manager());

context.AddToEmployees(new Developer());

context.SaveChanges();

 You can also query the Employees collection on the context in order to get all Employees, or do a Employees.OfType<Developer>().

* 1. 默认情况

public class memberEntity

public class vipEntity : memberEntity

public class volunteerEntity: memberEntity

public DbSet<memberEntity> BaseMembers { get; set; }

或者

public DbSet<memberEntity> BaseMembers { get; set; }

public DbSet<vipEntity> VipMembers { get; set; }

public DbSet<volunteerEntity> VoluteerMembers { get; set; }

此时： TPH 在数据库端只有一个表

public DbSet<vipEntity> VipMembers { get; set; }

public DbSet<volunteerEntity> VoluteerMembers { get; set; }

此时： TPC 在数据库端只有两个表

[Table("BaseMM")]

public class memberEntity

[Table("PPMM")]

public class vipEntity : memberEntity

[Table("VVMM")]

public class volunteerEntity: memberEntity

public DbSet<memberEntity> BaseMembers { get; set; }

或者

public DbSet<memberEntity> BaseMembers { get; set; }

public DbSet<vipEntity> VipMembers { get; set; }

public DbSet<volunteerEntity> VoluteerMembers { get; set; }

此时： TPT 在数据库端有三个表

public DbSet<vipEntity> VipMembers { get; set; }

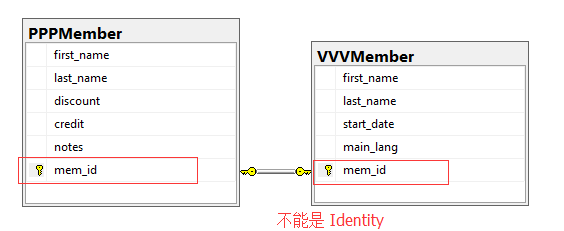
public DbSet<volunteerEntity> VoluteerMembers { get; set; }

此时： TPC 在数据库端只有两个表

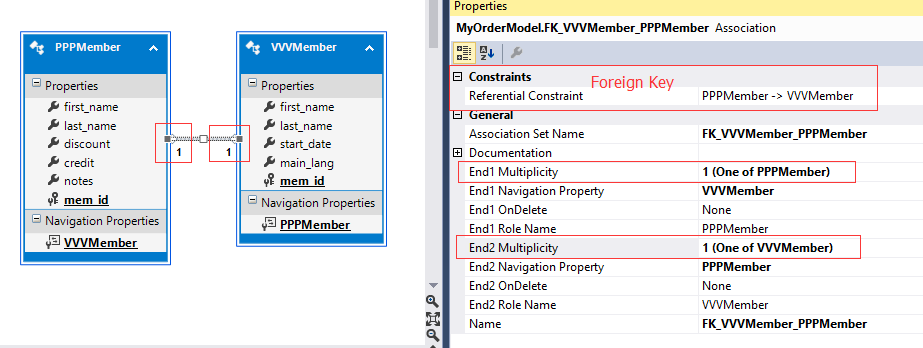
1. One to One Relationship ( 1 – 1 )

前提：

1. 数据的主表与附表，是通过主键关联的, 附表的主键即是主键也是外键。
2. 数据库主表，不管主表的主键是 Identity 字段与否， 附表的主键都不能是Identity 字段。



Entity Framework Model Designer:



1. 我们人为将附表属性 1- 0..1 改成 1 – 1, 实际效果还是 1 – 0..1

看代码： 并没有要求必须输入 VVVMember 实体记录

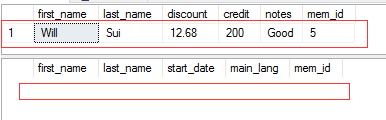
hh = new HowDB();

PPPMember pm = new PPPMember { first\_name="Will", last\_name="Sui", credit=200,

discount=(decimal)12.68, notes="Good" };

hh.PPPMembers.Add(pm);

hh.SaveChanges();



1. 只能通过客户端来强制要求输入 VVVMember 来保证 1-1 关系，

但是对于数据库端则没有强制：

using System.ComponentModel.DataAnnotations;

public partial class PPPMember

{

public string first\_name { get; set; }

public string last\_name { get; set; }

public Nullable<decimal> discount { get; set; }

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

public string notes { get; set; }

public int mem\_id { get; set; }

[Required] - **人为添加强制要求，来实现 1-1关系**

public virtual VVVMember VVVMember { get; set; }

}

再执行上面的代码，则出错：



hh = new HowDB();

PPPMember pm = new PPPMember { first\_name="Will", last\_name="Sui", credit=200,

discount=(decimal)12.68, notes="Good" };

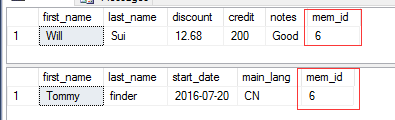
VVVMember vm = new VVVMember {mem\_id = pm.mem\_id, first\_name = "Tommy", last\_name = "finder",

main\_lang = "CN", start\_date = DateTime.Now };

pm.VVVMember = vm; - **必须要赋值，才不出错**

hh.PPPMembers.Add(pm);

hh.SaveChanges();

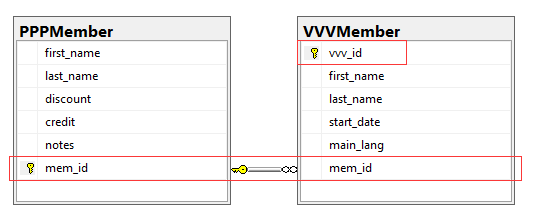


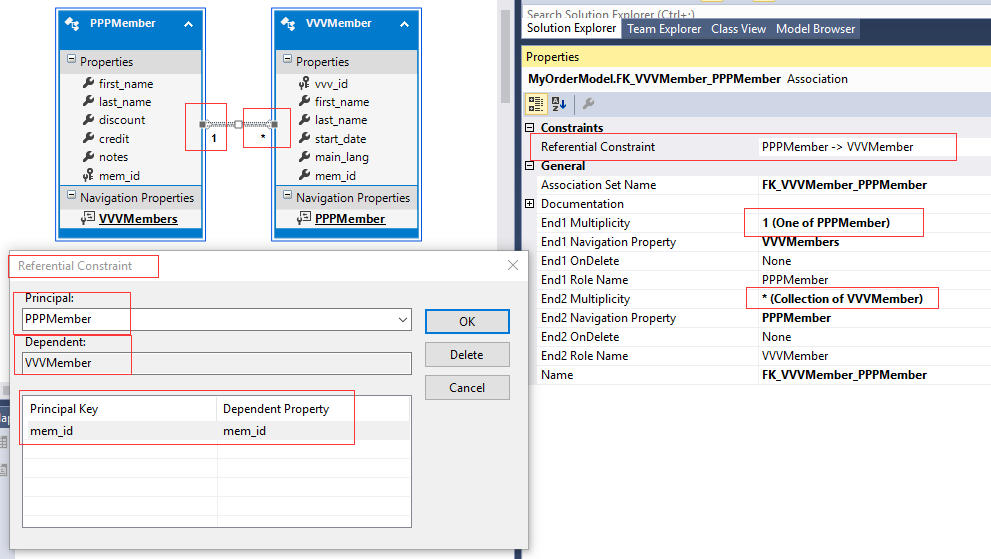
1. Master to Slave Relationship ( 1 – 0..1 )

同上， 只要在主表上对附表不强制输入[Required]即可，和模型 1-1 ， 1-0..1 关系不大 .

和数据库的主表和附表，通过主键关联， 附表的主键即是主键也是外键

1. One to Many ( 1 – Many )





hh = new HowDB();

PPPMember pm = new PPPMember { first\_name="Will", last\_name="Sui", credit=200, discount=(decimal)12.68, notes="Good" };

VVVMember vm1 = new VVVMember { first\_name = "Tommy", last\_name = "finder", main\_lang = "CN", start\_date = DateTime.Now };

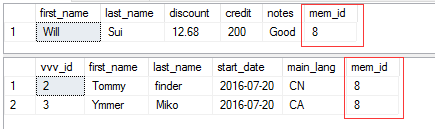
VVVMember vm2 = new VVVMember { first\_name = "Ymmer", last\_name = "Miko", main\_lang = "CA", start\_date = DateTime.Now };

pm.VVVMembers.Add(vm1);

pm.VVVMembers.Add(vm2);

hh.PPPMembers.Add(pm);

hh.SaveChanges();



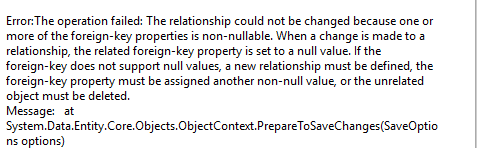
删除与级联删除：

设置级联删除， 只能再 1 的一方设置， 不能在 Many 一方设置

在客户端的模型关系里，如果设置了级联删除。 则不管数据库的关联是否设置了级联删除， 一样会同时在客户端和数据库端进行级联删除。

如果客户端模型没有设置级联删除， 而数据库端有级联删除。

则在客户端就弹出错误信息：

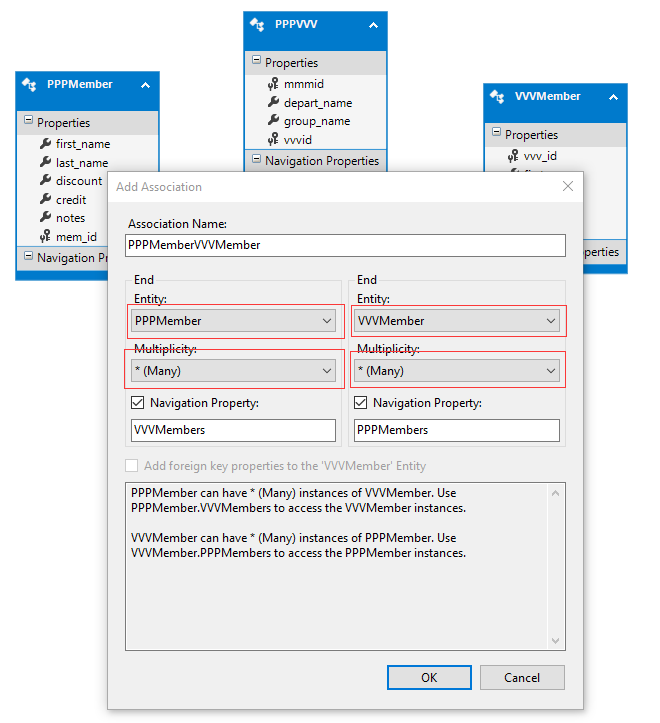


1. Many to Many Relationship

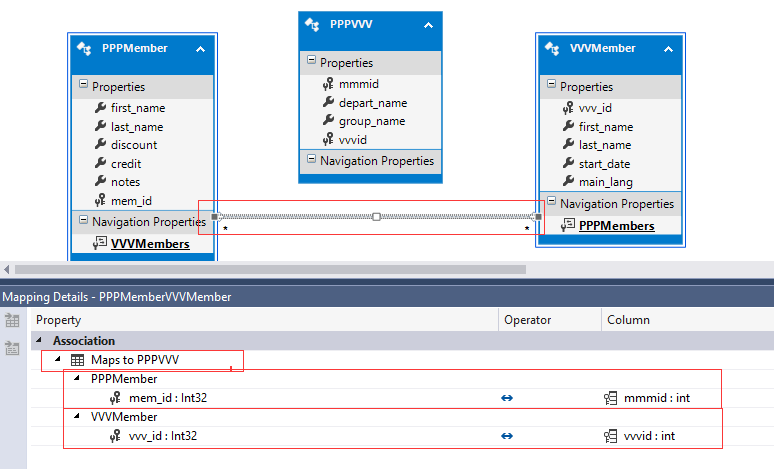


手工建立多对多关系：

1）两个表之间添加多对多关联：

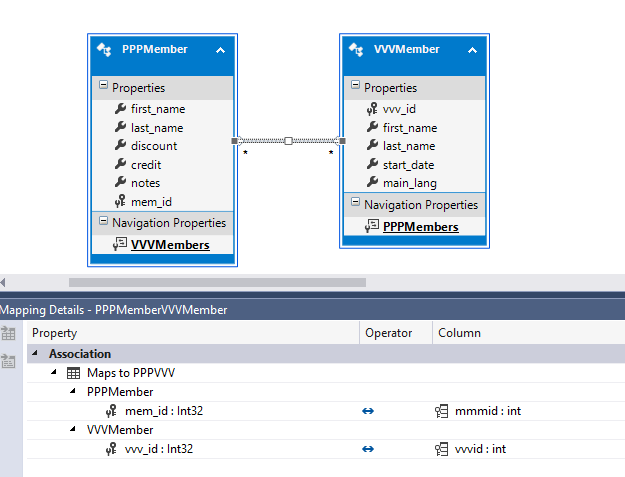


1. 重点就是中间表的映射关系：



做完关联映射以后， 必须把中间表删除，否则出错。

1. 删除中间表，完成



插入数据操作：

PPPMember pm1 = new PPPMember { first\_name="Will", last\_name="Liu", credit=200,

discount=(decimal)12.68, notes="Good" };

PPPMember pm2 = new PPPMember { first\_name = "Lilian", last\_name = "Liu", credit = 200,

discount = (decimal)12.68, notes = "Good" };

VVVMember vm1 = new VVVMember { first\_name = "Tommy", last\_name = "finder", main\_lang = "CN",

start\_date = DateTime.Now };

VVVMember vm2 = new VVVMember { first\_name = "Ymmer", last\_name = "Miko", main\_lang = "CA",

start\_date = DateTime.Now };

pm1.VVVMembers.Add(vm1); - pm1.vm1

pm1.VVVMembers.Add(vm2); - pm1.vm2

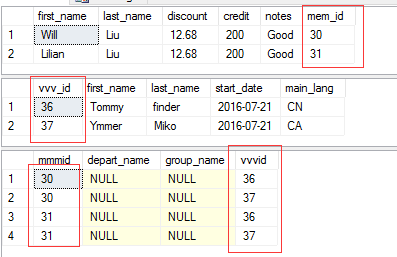
vm1.PPPMembers.Add(pm2); - 实际是 pm1.vm1.pm2

vm2.PPPMembers.Add(pm2); - 实际是 pm1.vm2.pm2

hh.PPPMembers.Add(pm1); - 注意：只是添加了一个实体 pm1, 却可以把 2 x 2 = 4 个记录插入

hh.SaveChanges();

神奇的插入数据效果



多对多不能设置级联删除：因为不能再Many一方设置级联删除

但是可以在数据库端，设置级联操作，两个表对中间表的级联操作。

PPPMember pm = this.hh.PPPMembers.FirstOrDefault();

if (pm != null)

{

this.hh.PPPMembers.Remove(pm);

this.hh.SaveChanges();

}

VVVMember vm = this.hh.VVVMembers.FirstOrDefault();

if( vm!=null)

{

this.hh.VVVMembers.Remove(vm);

this.hh.SaveChanges();

}

新增以后删除有问题，本地会有脏数据存在。 因为如果数据库有级联操作，数据库会把中间表删除， 而本地没有级联操作，

内存数据仍然有中间表数据？？？

I'm working on an ASP.NET MVC 5 project in VS2013, .NET 4.5.1, that uses Entity Framework 6 Code-First. I have a decent size database built out and somewhat working (project is about two weeks old). I want to integrate user authentication now, but I'm not sure how to approach it. After spending most of the day researching I've decided to give the new ASP.NET Identity framework a shot over having to write custom Membership or Role providers. What I'm confused about is how to make it all work with the existing database/model I have.

Currently I have an object called Employee that holds basic employee information (for now). After having pondered the question all day, I decided to decouple authentication from it into a User object, which is what Identity wants anyway. That being said how do I make it all work?

Here's my Employee class:

public class Employee : Person {

public int EmployeeId { get; set; }

public byte CompanyId { get; set; }

public string Name {

get {

return String.Format("{0} {1}", this.FirstName, this.LastName);

}

}

public string Password { get; set; }

public bool IsActive { get; set; }

public virtual ICollection<Address> Addresses { get; set; }

public virtual Company Company { get; set; }

public virtual ICollection<Email> Emails { get; set; }

public virtual ICollection<Phone> Phones { get; set; }

public Employee() {

this.Addresses = new List<Address>();

this.Emails = new List<Email>();

this.Phones = new List<Phone>();

}

}

And my DbContext derived class:

public class DatabaseContext : DbContext {

static DatabaseContext() {

Database.SetInitializer<DatabaseContext>(new DatabaseInitializer());

}

public DatabaseContext()

: base("Name=DatabaseContext") {

this.Database.Initialize(true);

}

public DatabaseContext(

string connectionString)

: base(connectionString) {

this.Database.Initialize(true);

}

/// DbSets...

public override int SaveChanges() {

try {

return base.SaveChanges();

} catch (DbEntityValidationException e) {

IEnumerable<string> errors = e.EntityValidationErrors.SelectMany(

x =>

x.ValidationErrors).Select(

x =>

String.Format("{0}: {1}", x.PropertyName, x.ErrorMessage));

throw new DbEntityValidationException(String.Join("; ", errors), e.EntityValidationErrors);

}

}

protected override void OnModelCreating(

DbModelBuilder modelBuilder) {

modelBuilder.Ignore<Coordinate>();

/// Configs...

base.OnModelCreating(modelBuilder);

}

}

Error Handle

public partial class SomethingSomethingEntities

{

public override int SaveChanges()

{

try

{

return base.SaveChanges();

}

catch (DbEntityValidationException ex)

{

// Retrieve the error messages as a list of strings.

var errorMessages = ex.EntityValidationErrors

.SelectMany(x => x.ValidationErrors)

.Select(x => x.ErrorMessage);

// Join the list to a single string.

var fullErrorMessage = string.Join("; ", errorMessages);

// Combine the original exception message with the new one.

var exceptionMessage = string.Concat(ex.Message, " The validation errors are: ", fullErrorMessage);

// Throw a new DbEntityValidationException with the improved exception message.

throw new DbEntityValidationException(exceptionMessage, ex.EntityValidationErrors);

}

}

}

List<string> errorMessages = new List<string>();

foreach (DbEntityValidationResult validationResult in ex.EntityValidationErrors)

{

string entityName = validationResult.Entry.Entity.GetType().Name;

foreach (DbValidationError error in validationResult.ValidationErrors)

{

errorMessages.Add(entityName + "." + error.PropertyName + ": " + error.ErrorMessage);

}

}

[HashSet<T>类](http://www.cnblogs.com/kissdodog/archive/2013/02/02/2889887.html)

HashSet<T>类主要是设计用来做高性能集运算的，例如对两个集合求交集、并集、差集等。集合中包含一组不重复出现且无特性顺序的元素。

HashSet<T>的一些特性如下:

1、HashSet<T>中的值不能重复且没有顺序。

2、HashSet<T>的容量会按需自动添加。

构造方法：

HashSet()　默认相等比较器创建一个空的新实例。

HashSet(IEnumerable<T> collection)　　把指定集合中的collection中的数据复制到集中

HashSet(IEqualityComparer<T> comparer)　　使用指定的相等比较器创建一个空的新实例

HashSet(IEnumerable<T> collection,IEqualityComparer<T> comparer)　　使用指定的比较器实例化数据，且将指定集合中的元素复制到集合中。

因为HashSet<T>是专门设计来做集合运算的，因此它提供的方法中有不少是和集合运算相关的。

以下给出它的一些常用方法介绍

成员　　　　　　　　类型　　　　　　　　说明

Add　　　　　　　　方法　　　　　　　　将指定的元素添加到集合中

Clear　　　　　　　 方法　　　　　　　　 清空集合中的所有元素

Contains　　　　    方法　　　　　　　　 确定某元素是否在HashSet<T>中

Exists　　　　　　  方法　　　　　　　　 确定HashSet<T>是否包含于指定条件相匹配的元素

ExceptWith　　　　方法　　　　　　　　 从当前HashSet<T>移除指定集合中的所有元素

IntersectWith　　   方法　　　　　　　　修改当前的HashSet<T>对象，以仅包含该对象和指定集合中存在的元素

IsProperSubsetOf　 方法　　　　　　　　确定HashSet<T>对象是否为指定集合的真子集

IsProperSupersetOf 方法　　　　　　　　确定HashSet<T>对象是否为指定集合的真超集

IsSunsetOf　　　　 方法　　　　　　　　 确定HashSet<T>对象是否为指定集合的子集

IsSupersetOf　　　 方法　　　　　　　　 确定HashSet<T>对象是否为指定集合的超集

Remove　　　　　　方法　　　　　　　　 从HashSet<T>对象中移除指定的元素

RemoveWhere　　  方法　　　　　　　　 从HashSet<T>集合中移除与指定谓词所定义的条件相匹配的所有元素

SetEquals　　　　　方法　　　　　　　　 确定HashSet<T>对象与指定的集合中是否包含相同的元素

SynmmetricExceptWith　　方法　　　　 修改当前的HashSet<T>对象，以仅包含该对象或指定集合中存在的元素

TrimExcess　　　　方法　　　　　　　　 将HashSet<T>对象的容量设置为它所包含的元素的实际个数，向上舍入为接近的特性与实现的值。

UnionWith　　　　 方法　　　　　　　　 修改当前的HashSet<T>对象，以包含该对象本身和指定集合中存在的所有元素

 给个简单的例子，写不完的，总之记得HashSet<T>主要的作用是用来进行，交集、并集等运算的就OK了。

[复制代码](javascript:void(0);)

static void Main(string[] args)

{

HashSet<string> hs = new HashSet<string>();

hs.Add("你");

hs.Add("好");

hs.Add("吗");

HashSet<string> hs1 = new HashSet<string>();

hs1.Add("你");

hs1.Add("好");

bool b = hs1.IsProperSubsetOf(hs); //确定hs1是否是hs的真子集

Console.WriteLine(b); //输出True

HashSet<string> hs2 = new HashSet<string>();

hs2.Add("爱你");

IEnumerable<string> list = hs.Union(hs2); //返回并集

foreach (string str in list)

{

Console.WriteLine(str); //输出 你 好 吗 爱你

}

Console.ReadKey();

}

HashSet<T> vs List<T> : 基本相同， HashSet.UnionWith(otherList) ; HashSet.RemoveWhere( predict<>() )

# [Entity Framework多对多关系实践(many-to-many)](http://www.cnblogs.com/artwl/archive/2011/11/03/2234359.html)

　　Entity Framework中有三种关系，一对一(one-to-one)，一对多(one-to-many)，多对多(many-to-many)，前两种就不说了，园子里这方面的文章很多（dudu的：[Entity Framework 实践系列](http://www.cnblogs.com/dudu/tag/Entity%20Framework%20%E5%AE%9E%E8%B7%B5%E7%B3%BB%E5%88%97/)，杨延成的：[EF框架step by step](http://www.cnblogs.com/yangyancheng/archive/2011/05/17/2048794.html)，郝冠军的：[Entity Framework系列文章](http://www.cnblogs.com/haogj/archive/2011/05/06/2038965.html)），看过之后简单的使用基本没什么问题，这里要说的是第三种：多对多(many-to-many)。

　　这里单独把多对多关系拿出来说，不是因为上述系列文章中没有，只不过需求不同， 我的需求用上述系列文章中的方法实现不了。这里先用一个例子说一下我的需求吧：我要用EF处理 question(QID,Title)与tag(TID,TagName)之间的关系，这是一个多对多关系(一个问题有多个标签，一个标签有多个问 题)，因此在数据库中除了question与tag表外应该还有他们的关系表question\_tag表，问题就出在question\_tag表上。

　　如果我的question\_tag表仅仅只有两个字段QID与TID，那么用上面系列文章中提到的方法就可以实现，关键代码如下：

1 [Table("Question")]  
 2 public class Question  
 3 {  
 4 [Key]  
 5 [DatabaseGenerated(DatabaseGeneratedOption.Identity)]  
 6 public int QID { get; set; }  
 7 public string Title { get; set; }  
 8 public virtual ICollection<Tag> Tags { get; set; }  
 9 }  
10   
11 [Table("Tag")]  
12 public class Tag  
13 {  
14 [Key]  
15 [DatabaseGenerated(DatabaseGeneratedOption.Identity)]  
16 public int TID { get; set; }  
17 public string TagName { get; set; }  
18 public virtual ICollection<Question> Questions { get; set; }  
19 }  
20   
21 public class BlogDbContext : DbContext  
22 {  
23 protected override void OnModelCreating(DbModelBuilder modelBuilder)  
24 {  
25 modelBuilder.Entity<Question>()  
26 .HasMany(q => q.Tags)  
27 .WithMany(t => t.Questions)  
28 .Map  
29 (  
30 m =>  
31 {  
32 m.MapLeftKey("QID");  
33 m.MapRightKey("TID");  
34 m.ToTable("Question\_Tag");  
35 }  
36 );  
37 base.OnModelCreating(modelBuilder);  
38 }  
39 public IDbSet<Question> Questions { get; set; }  
40 public IDbSet<Tag> Tags { get; set; }  
41 }

　　现在的问题是我的question\_tag表为了业务需求不仅仅只有这两个字段（这个需求应该很常见，本例中增加一个时间字段DateAdded作为示例），因此用上面的方案就不行了。那么要怎么处理呢，找了好多资料都不行，没办法只好自己动手，丰衣足食。

　　首先想到的是，既然question\_tag表中还有其它字段，那么这个实体肯定要表现出来。然后想到的是，按原来的方法question跟tag是直接产生联系的，EF根据question和tag的定义可以判断出是多对多关系，但现在加了一个关系实体question\_tag，question跟question\_tag的关系是一对多，tag跟question\_tag的关系也是一对多，因此可以通过question\_tag来联接question跟tag（数据库中这个表的存在本来就是这个意思），也就是说question跟tag不直接产生联系。有了上面的想法，经过多次尝试后，我把实体间的关系修改为如下形式：

1 [Table("Question")]  
 2 public class Question  
 3 {  
 4 [Key]  
 5 [DatabaseGenerated(DatabaseGeneratedOption.Identity)]  
 6 public int QID { get; set; }  
 7 public string Title { get; set; }  
 8 public virtual ICollection<QuestionTag> QuestionTags { get; set; }  
 9 }  
10   
11 [Table("Tag")]  
12 public class Tag  
13 {  
14 [Key]  
15 [DatabaseGenerated(DatabaseGeneratedOption.Identity)]  
16 public int TID { get;set;}  
17 public string TagName { get; set; }  
18 public virtual ICollection<QuestionTag> QuestionTags { get; set; }  
19 }  
20   
21 [Table("QuestionTag")]  
22 public class QuestionTag  
23 {  
24 [Key]  
25 [Column(Order=0)]  
26 [ForeignKey("Question")]  
27 public int QID { get; set; }  
28   
29 [Key]  
30 [Column(Order = 1)]  
31 [ForeignKey("Tag")]  
32 public int TID { get; set; }  
33   
34 public DateTime DateAdded { get; set; }  
35   
36 public virtual Question Question { get; set; }  
37   
38 public virtual Tag Tag { get; set; }  
39 }

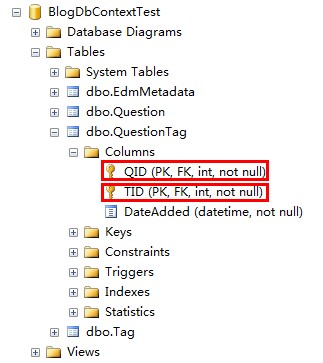
　　有了上面的定义后，DbContext的定义就很简单了，不需要重写OnModelCreating：

1 public class BlogDbContext : DbContext  
2 {  
3 public IDbSet<Question> Questions { get; set; }  
4 public IDbSet<Tag> Tags { get; set; }  
5 public IDbSet<QuestionTag> QuestionTags { get; set; }  
6 }

好了，下面开始写测试代码，就是增删改查操作：

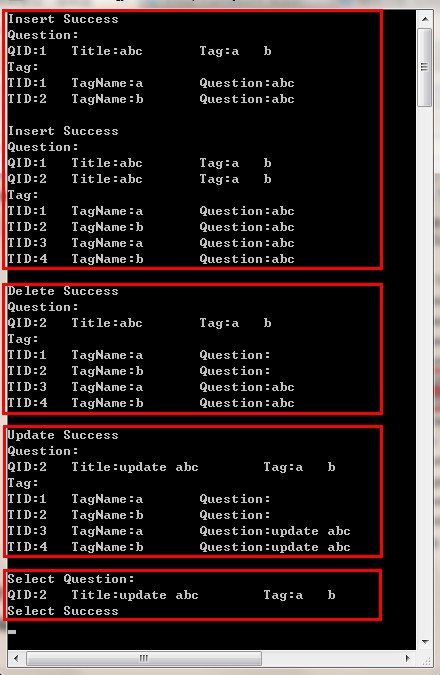
1 public class EFTest  
 2 {  
 3 public void Insert()  
 4 {  
 5 using (var db = new BlogDbContext())  
 6 {  
 7 //添加一个question，两个tag  
 8 var question = new Question() { Title = "abc" };  
 9 var tagA = new Tag() { TagName = "a" };  
 10 var tagB = new Tag() { TagName = "b" };  
 11 var qes = db.Questions.Add(question);  
 12 var tA = db.Tags.Add(tagA);  
 13 var tB = db.Tags.Add(tagB);   
 14 db.SaveChanges();  
 15   
 16 //添加question\_tag  
 17 var questiontaga = new QuestionTag() { QID = qes.QID, TID = tA.TID, DateAdded = DateTime.Now };  
 18 var questiontagb = new QuestionTag() { QID = qes.QID, TID = tB.TID, DateAdded = DateTime.Now };  
 19 var qtA = db.QuestionTags.Add(questiontaga);  
 20 var qtB = db.QuestionTags.Add(questiontagb);  
 21 db.SaveChanges();  
 22 Console.WriteLine("Insert Success");  
 23   
 24 //显示数据  
 25 Show();  
 26 }  
 27 }  
 28   
 29 public void Delete()  
 30 {  
 31 using (var db = new BlogDbContext())  
 32 {  
 33 var qes = db.Questions.SingleOrDefault(q => q.QID == 1);  
 34 if (qes != null)  
 35 {  
 36 db.Questions.Remove(qes);  
 37 db.SaveChanges();  
 38 Console.WriteLine("Delete Success");  
 39   
 40 Show();  
 41 }  
 42 }  
 43 }  
 44   
 45 public void Update()  
 46 {  
 47 using (var db = new BlogDbContext())  
 48 {  
 49 var qes = db.Questions.SingleOrDefault(q => q.QID == 2);  
 50 if (qes != null)  
 51 {  
 52 qes.Title = "update abc";  
 53 db.SaveChanges();  
 54 Console.WriteLine("Update Success");  
 55   
 56 Show();  
 57 }  
 58 }  
 59 }  
 60   
 61 public void Select()  
 62 {  
 63 using (var db = new BlogDbContext())  
 64 {  
 65 var qes = db.Questions.SingleOrDefault(q => q.QID == 2);  
 66 if (qes != null)  
 67 {  
 68 Console.WriteLine("Select Question:");  
 69 Console.Write("QID:"+qes.QID + "\tTitle:" + qes.Title+"\tTag:");  
 70 qes.QuestionTags.ForEach(t => Console.Write(t.Tag.TagName+"\t"));  
 71 }  
 72 Console.WriteLine("\nSelect Success");  
 73 }  
 74 }  
 75   
 76 public void Show()  
 77 {  
 78 using (var db = new BlogDbContext())  
 79 {  
 80 //显示question  
 81 var qes = db.Questions;  
 82 if (qes != null)  
 83 {  
 84 Console.WriteLine("Question:");  
 85 qes.ForEach(q =>  
 86 {  
 87 Console.Write("QID:"+q.QID+"\tTitle:"+q.Title+"\tTag:");  
 88 q.QuestionTags.ForEach(t =>  
 89 {  
 90 Console.Write(t.Tag.TagName+"\t");  
 91 });  
 92 Console.WriteLine();  
 93 });  
 94 }  
 95   
 96 //显示tag  
 97 var tag = db.Tags;  
 98 if (tag != null)  
 99 {  
100 Console.WriteLine("Tag:");  
101 tag.ForEach(t =>  
102 {  
103 Console.Write("TID:" + t.TID + "\tTagName:" + t.TagName + "\tQuestion:");  
104 t.QuestionTags.ForEach(q =>  
105 {  
106 Console.Write(q.Question.Title + "\t");  
107 });  
108 Console.WriteLine();  
109 });  
110 }  
111   
112 Console.WriteLine();  
113 }  
114 }  
115 }

　　很幸运地通过了，运行后数据库中生成的表如下：

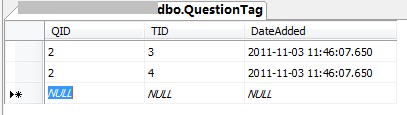


　　要注意的是这里QID和TID不仅仅是PK，也是FK。

　　程序的输出如下：



　　可以看到程序能很好地满足我的需求。在上面的删除代码中删除了QID为1的question后，数据库中question\_tag表中的数据如下：



　　我们可以看到，question\_tag表中QID为1的数据也同时删除了，正是我们需要的结果。

**最后做个小结吧：**

　　一直以来都是看的多，写的少，从老鸟那里吸收的多，贡献的少，总怕自己写的不好，以后争取慢慢改变这种状况，把自己学到的知识总结出来，争取能给新手一些帮助吧。

　　希望这篇文章对大家有所帮助，当然了，限于水平欢迎大家拍砖，提出更好的解决方案。