## [EF CodeFirst 初识](http://www.cnblogs.com/JasonShenW/p/5084066.html)

2015-12-28 21:26 by Gaea, 307 阅读, 4 评论, [收藏](http://www.cnblogs.com/JasonShenW/p/5084066.html), [编辑](https://i.cnblogs.com/EditPosts.aspx?postid=5084066)

随着EntityFramework的发展，原先的三种方式，{Code First ,Model First，Database First }  CodeFirst基本摆脱了另外两种方式

成为了 最受欢迎的一种编程模型。 我们通过先编写代码，然后自动生成数据库。 程序员只需专心的面向对象写代码，而无需再写烦人的各种Sql语句。

新建控制台项目，通过Nuget引入EF库

增加两个类

public class Blog

{

public string ID { get; set; }

public string BlogName { get; set; }

public string BlogAuthor { get; set; }

public virtual List < Post> Posts { get ; set ; } //导航属性

}

public class Post

{

public string ID { get; set; }

public string PostName { get; set; }

public string Content { get; set; }

public virtual Blog BelongBlog { get; set; } //导航属性

}

这两个类属于一对多的关系，一个博客可以有多个博文。

EF的核心在于DBContext类，可以说就是这个类负责着实体和数据库之间的交互

所以增加一个context类

public class BlogContext : DbContext

{

private readonly static string CONNECTIONSTRING = "Data Source=.;Initial Catalog=EFBlog;User ID=sa;Password=11111;Integrated Security=True;Pooling=False";

public BlogContext():base (CONNECTIONSTRING)

{

}

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

public DbSet < Post> Posts { get ; set ; }

}

注意需要连接字符串， 然后提供Dbset集合，以后的程序中，就可以通过这些集合获取数据了。

在main中如下

public static void Main( string [] args)

{

BlogContext ctx = new BlogContext ();

ctx.Blogs.Add( new Blog { ID = "1", BlogName = "Jason's Blog" , BlogAuthor = "Jason",

Posts = new List < Post> {

new Post {ID="1" ,PostName= "shenwei",Content= "今天的阳光真好，要晒被子！" },

new Post {ID="2" ,PostName= "shenwei",Content= "今天下雨了，都不能出去玩！" }

} });

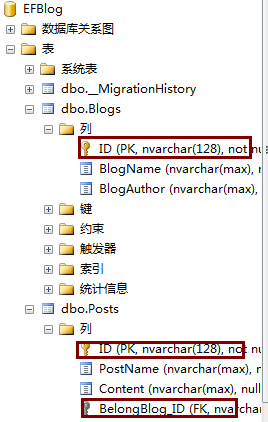
ctx.Blogs.Add( new Blog { ID="2" ,BlogName="Code Home",BlogAuthor= "zhangxiaomao"});

ctx.SaveChanges();

Console .ReadKey();

}

运行之后 数据库如下



**主键**     可以发现 主键每个类都各自对应一张表，而主键也自动有了，原因是 在EF默认规则下 EF会 自动把ID或者 类名+ID的字段设置为主键，如果不存在这样的字段则会报错

不过 我们当然也可以通过在DataAnnotations在字段上 标记[ key ] 特性，来指明 主键。

**外键**当codefirst发现我们的类之间存在一对多的关系时，会自动化在相应的表里生成外键，这样一来，类之间的关系就很清楚了。

select \* from posts

IMG_263

select \* from blogs

IMG_264

 ==========================================================================

## [EF Codefirst(二)数据注释](http://www.cnblogs.com/JasonShenW/p/5084363.html)

2015-12-28 23:17 by Gaea, 280 阅读, 0 评论, [收藏](http://www.cnblogs.com/JasonShenW/p/5084363.html), [编辑](https://i.cnblogs.com/EditPosts.aspx?postid=5084363)

CodeFirst通过分析我们在代码里编写的类，以及类之间的关系生成数据库表，以及表之间的各种关系。数据库的表会涉及到主键，外键，列是否为空，列类型等等。 我们要通过怎样的方式来暴露这些信息呢？

CodeFirst通过DataAnnotations（在 System.ComponentModel.DataAnnotations 命名空间中 )特性类标示这些信息。

常用的一些标识如下

## **主键**

    如果不存在符合EF默认规则的主键时，用key标注也可。个人不赞成这样，遵守默认规则就好，免去不必要麻烦。

public class Category

{

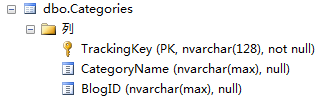
[ Key ]

public string TrackingKey { get; set; }

public string CategoryName { get; set; }

public string BlogID { get; set; }

}



## **自增长**

如果主键是int类型，EF为默认设置为增长。但如果是GUID类型，则要手动的设置自增长

public class User

{

[Key ,DatabaseGenerated (DatabaseGeneratedOption .Identity)]

public Guid UserGuid { get; set; }

public string UserName { get; set; }

}

查看数据表的创建脚本，我们会发现多了这样的一句话

ALTER TABLE [dbo]. [Users] ADD   DEFAULT ( newsequentialid ()) FOR[UserGuid]

## **必需 MaxLength 和 MinLength**

public class Category

{

[ Key ]

public string TrackingKey { get; set; }

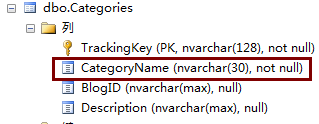
[ Required ][MaxLength (30), MinLength(5)]

public string CategoryName { get; set; }

public string BlogID { get; set; }

public string Description { get; set; }

}



## **NotMapped**

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

并且这个标示还可以用于对整个类进行标示，那这个类也就不会被映射了。

## **ComplexType**

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

public class Blog

{

public string ID { get; set; }

public string BlogName { get; set; }

public string BlogAuthor { get; set; }

public virtual List < Post> Posts { get ; set ; } //导航属性

public BlogDetails Detail { get; set; }

}

[ ComplexType]

public class BlogDetails

{

public string BlogDescription { get; set; }

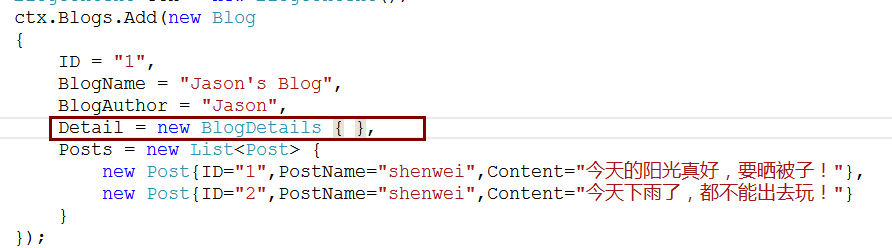
public string CreateTime { get; set; }

}

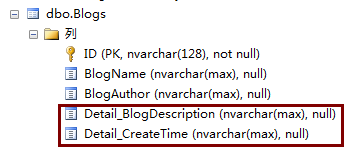


但是运行的时候却报错，提示Detail属性，不可以为空 。原来，虽然复杂类型的每个属性都可以为空，但当作为复杂属性出现时，却是不可以为空的，必须初始化！

需要修改如下



修改之后运行如下



会发现 复杂类型将自己的属性并入了所属类型，并以复杂类型名+属性名作为新列名

=========================================================================

## [EF CodeFirst(三) 并发处理](http://www.cnblogs.com/JasonShenW/p/5085382.html)

2015-12-29 13:10 by Gaea, 540 阅读, 0 评论, [收藏](http://www.cnblogs.com/JasonShenW/p/5085382.html), [编辑](https://i.cnblogs.com/EditPosts.aspx?postid=5085382)

并发分为两种，一种叫做悲观并发，一种叫乐观并发。 名字挺文艺

**悲观并发**

悲观并发是什么呢？ 就拿我们常用的代码版本控制来说。 有一个文档，A和B都要 获取这个文档并进行修改， 如果当A在读取这个文档数据时，就单独霸占了这个文档，B无法获取，只有当A读取修改完毕释放锁时,B才能获取这个文件，也就是一个人去的控制权的时候，其他人只能等待。这就是悲观锁。因为担心，多人同时操作造成的数据紊乱，大概是因为建立在这样的心态上，取名 “悲观锁”的。

悲观锁 通常用在频繁发生数据竞争的激烈环境下，以及 通过锁来保护数据所需的成本小于回滚事务的成本的时候。

**乐观并发**

乐观并发显然，“心理”上是反过来的，就是 “我不担心A和B同时取得控制权，A和B可以同时读取 同时修改，但是A修改保存后，B再修改保存，这时候系统会发现，当前文档和B进入系统时不一样了，就会报错。这显然也是一种不错的处理方式。

乐观锁 应用场景一般是 数据竞争并不是特别激烈，且 偶尔的数据争执所需的回滚事务的成本小于读取数据时锁定数据所需要的成本。乐观并发的初衷在于，不希望经常性的看到数据争执。

悲观并发如果加锁的成本较高的话，会很明显的降低效率，降低系统的并发性。

乐观并发 通常分为 三个阶段 读取阶段--校验阶段--写入阶段

在读取阶段，A和B分别将数据读入各自机器缓冲,此时并没有校验，在校验阶段，系统事务会对文件进行同步校验，如果不出现问题，则进入第三阶段写入，数据最终被提交，否则报错。

悲观锁还有一个常见的问题就是“死锁”，比如文档T1 和T2是内容相关的，但是不巧 T1被A锁住了 T2有被B锁住了， 那就会卡在这直到有一方先取消锁。

我们先看一下不控制并发时的场景

//未进行并发处理

User user = new User { UserName="shenwei" ,certID= "11111"};

using (BlogContext ctx= new BlogContext())

{

ctx.Users.Add(user);

ctx.SaveChanges();

} //首先插入一条数据 并提交

//定义两个context同时进行操作

BlogContext firContext = new BlogContext ();

User u1 = firContext.Users.FirstOrDefault();

BlogContext secContext = new BlogContext ();

User u2 = secContext.Users.FirstOrDefault();

u2.UserName = "zhangxiaomao" ; //改变名字 并提交 secContext.SaveChanges();

u1.UserName = "xxxxxx" ;

u1.certID = "22222" ; //另一个操作改变certid，也提交

firContext.SaveChanges();

数据库 查询select \* from Users ;

IMG_258

回到我们的EF codefirst . EntityFramework只支持乐观并发，也就是说EF其实并不希望经常性的看到数据冲突。

**针对整条记录的并发**

EF实现并发控制 需要借助 TimeStamp 标示 ，并且一个类只能有 一个此标示，标示的必须是byte[]类型

public class Blog

{

public string ID { get; set; }

public string BlogName { get; set; }

public string BlogAuthor { get; set; }

public virtual List <Post> Posts { get; set ; } //导航属性

public BlogDetails Detail { get; set; }

[ Timestamp]

public byte [] version { get; set; }

}

测试如下

//并发模拟

Blog b = new Blog

{

ID = "24",

BlogName = "Gaea",

BlogAuthor = "shenwei",

Detail = new BlogDetails { }

};

//先通过一个ctx插入数据并提交

using(BlogContext context=new BlogContext())

{

context.Blogs.Add(b);

context.SaveChanges();

}

//创建一个ctx取的第一条数据，修改 但是不提交

BlogContext fircontext = new BlogContext();

Blog firstblog = fircontext.Blogs.FirstOrDefault();

firstblog.BlogName = "哈哈，被改掉了" ;

//创建另一个ctx还是取第一条数据，修改并提交

BlogContext secContext = new BlogContext();

Blog secondBlog = secContext.Blogs.FirstOrDefault();

secondBlog.BlogAuthor = "JasonShen";

secContext.SaveChanges();

//这个时候再提交第一个ctx所做的修改

try

{

//这是后会发现现在的数据，已经和刚进入时发生了变化，故报错

fircontext.SaveChanges();

Console.WriteLine("保存成功" );

} catch(Exception e)

{

Console.WriteLine("保存失败" );

}

Console.ReadKey();

}

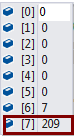
结果如下

IMG_263

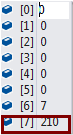
select \* from Blogs;

IMG_264  
之所以能捕捉到错误 是因为 EF这里的操作机制是将 被Timestamp标识的字段加入 where子句。

一开始插入一条数据之后，时间戳是这样的，初始版本的对象也就是这个样子



后来两个context 各自获取这个对象，其中一个进行修改，并提交，这个时候数据库中的时间戳标示的字段已经发生了改变。



这个时候 ，另一个context提交的时候 执行update .... where version=‘初始版本的version’ 然后会发现找不到，于是就报错！

也就是说依靠 timespan标示的字段来确认是否与初始版本发生了改动，若发生了，就报错，进行错误处理。

那如果捕捉到了异常，EF会怎么处理呢？使用Reload处理

## **Resolving optimistic concurrency exceptions with Reload**

　　使用Reload数据作为解决乐观并发异常的策略之一，除了Reload外，还有其他几种冲突解决策略，这里只讲下常用的Reload

　　微软Entity Framework 团队 推荐处理乐观并发冲突的策略之一是Reload数据，也就是EF检测到并发冲突时会抛出Db[update](http://www.07net01.com/tags-update-0.html" \t "http://www.cnblogs.com/JasonShenW/p/_blank)ConcurrencyException，这时解决冲突分为Client Wins或者Store Wins ，而Reload处理也就是Store Wins，意味着放弃当前内存中的实体，重新到数据库中加载当前实体，EF官方团队给出来的示例代码如下

捕捉到异常的时候

try

{

//这是后会发现现在的数据，已经和刚进入时发生了变化，故报错 fircontext.SaveChanges();

Console .WriteLine("保存成功" );

} catch (DbUpdateConcurrencyException e)

{

Console .WriteLine("保存失败" );

Console .WriteLine("Reload" );

e.Entries.Single().Reload();

Console .WriteLine(firstblog.BlogName); //会发现 变成了初始从数据库里加载的数据值

}

**针对单个字段的并发**

 有些时候并不需要控制针对整条记录的并发，只需要控制某个列的数据 不会出现脏操作就ok

这个时候 就使用ConcurrencyCheck 标示

public class User

{

[Key ,DatabaseGenerated (DatabaseGeneratedOption .Identity)]

public Guid UserGuid { get; set; }

public string UserName { get; set; }

[ ConcurrencyCheck ]

public string certID { get; set; }

}

//针对单个字段 标示的ConcurrencyCheck 的并发

User user = new User { UserName = "shenwei" , certID = "11111" };

using (BlogContext ctx = new BlogContext ())

{

ctx.Users.Add(user);

ctx.SaveChanges();

} //首先插入一条数据 并提交

//定义两个context同时进行操作

BlogContext firContext = new BlogContext ();

User u1 = firContext.Users.FirstOrDefault();

BlogContext secContext = new BlogContext ();

User u2 = secContext.Users.FirstOrDefault();

u2.certID= "22222" ; //改变名字 并提交 secContext.SaveChanges();

try

{

u1.certID = "33333" ; //另一个操作改变certid，也提交 firContext.SaveChanges();

} catch (Exception e)

{

Console .WriteLine("并发报错" );

}

当然可以同时用concurrentCheck标示多个字段，那被标示的每个都不能被同时修改了。这里背后的机制同样是where将被标示的字段作为了筛选条件。

**总结**

经过分析乐观锁 并不适合处理高并发的场景，少量的数据冲突才是乐观并发的初衷。 悲观锁同样也不适合处理高并发，特别在加锁成本比较大的时候。

如果项目并发量确实大， 那就可以考虑采用其他技术实现，比如 消息队列等。

## [EF CodeFirst(四) 关系](http://www.cnblogs.com/JasonShenW/p/5097814.html)

2016-01-04 09:07 by Gaea, 274 阅读, 0 评论, [收藏](http://www.cnblogs.com/JasonShenW/p/5097814.html), [编辑](https://i.cnblogs.com/EditPosts.aspx?postid=5097814)

数据库表之间有一对一  一对多 多对多关系。那同样，CodeFirst也要能分析这些类之间的这些关系。

CodeFirst可以自动通过分析类之间的属性导航属性 从而得出类之间的关系，自动确定外键。

**一对多**

一对多是最为常见的一种关系，符合怎样的规范会被CodeFirst识别为一对多的关系呢？

public class Blog

{

public string ID { get; set; }

public string BlogName { get; set; }

public string BlogAuthor { get; set; }

public virtual List <Post> Posts { get; set ; } //导航属性

public virtual List <Article> Articles { get; set; } //导航属性 [ Timestamp]

public byte [] version { get; set; }

}

public class Post

{

public string ID { get; set; }

public string PostName { get; set; }

public string Content { get; set; }

public virtual Blog BelongBlog { get; set; } //导航属性

}

public class Article

{

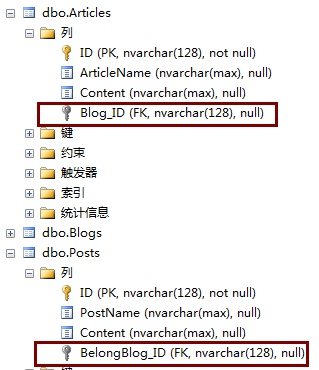
public string ID { get; set; }

public string ArticleName { get; set; }

public string Content { get; set; }

}

CodeFirst发现 Blog中有两个集合导航属性，Post中也有一个Blog类型的引用属性，分析出Blog和 Post以及Article之间都存在一对多的关系。



我们还可以发现 Article中并没有 Blog类型的导航属性，并且 Post和Article生成的主键名称不一样。

其实EF推断两个类之间存在一堆多的关系，只需要 两个类中的其中一个存在指向另一个类的导航属性即可。

并且 外键名称的生成规则是 首先是 导航属性名称+ 下划线\_ +ID 如果导航属性不存在 那就是 另一个类的类名+ID

**存在显示的符合规范的外键**

有些时候就是手贱，喜欢手动写上外键，那么属性名称符合什么样的规范会被CodeFirst识别为外键，而不是一般属性呢？（当然前提是已经存在导航属性）

当属性符合 [目标类型的键名]，[目标类型名称]+[目标类型键名称]，或[导航属性名称]+[目标类型键名称]的形式的时候，会被判定为外键。

当两个类之间存在多个关系时会怎样呢？在Post类中，有可能需要跟踪谁创建了它，以及谁编辑了它。那么Post类就需要增加两个导航属性

public Person CreatedBy { get; set; }

public Person UpdatedBy { get; set; }

那同样在我们的Person类之中，也需要加上两个属性，一个属性指向这个人创建的所有文章，另一个属性指向之个人更新的所有文章。

public virtual List<Post> PostsWritten { get; set; }

public virtual List <Post> PostsUpdated { get; set ; }



会发现这时候生成了四个外键，这是因为当类之间存在多个关系的时候，EF是无法准确分辨的，需要我们手动的添加代码来帮助EF分析。要想解决这个问题 就要用另一个数据注释 ： InverseProperty 代码赢改成如下

[InverseProperty( "CreatedBy")]

public virtual List <Post> PostsWritten { get; set ; }

[ InverseProperty("UpdatedBy" )]

public virtual List <Post> PostsUpdated { get; set ; }

要注意 InverseProperty括号内的名称是要与另一个类的相关的导航属性名称相匹配的，否则会报错。

这个时候数据库外键就正确了



**多对多**

如果两个类之间，各自都有集合导航属性指向对方，那这两个类之间的关键会被识别为多对多关系。

public class Student

{

public int StudentID { get; set; }

public string Name { get; set; }

public List <Course> Courses { get; set ; }

}

public class Course

{

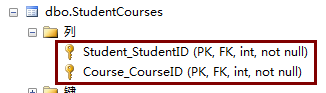
public int CourseID { get; set; }

public string CourseName { get; set; }

public List <Student> Students { get; set ; }

}

学生可以选择多门课程，每门课也有很多学生，这样的多对对关系，codefirst会自动生成第三张表，表里存放另外两张表的主键作为外键存放。



**一对一**  
在一对一和一对多的关系之后，我们回头再来讲一对一的关系。

如果需要将两个类之间的关系配置为一对一的关系，那么需要两个类互相有指向对方的引用属性

public class Book

{

public int BookID { get; set; }

public string Name { get; set; }

public BookCover Cover { get; set; }

}

public class BookCover

{

[ Key , ForeignKey ("Coverof" )]

public int BookID { get; set; }

public byte [] Photo { get; set; }

public Book Coverof { get; set; }

}

并且两个类的主键要是一样的，而且为了确认其中一个为关系中的依赖主体，必须用ForeignKey指明，也就是上面的BookCover表的BookID既是外键也必须是主键。否则会报错。