

安装工具包后才能使用迁移命令

```
Install-Package Microsoft.EntityFrameworkCore.Tools
```

数据使用特性会替代约定，但会被 Fluent API 配置替代

EF core连接SQLserver

NuGet安装驱动包

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

安装工具包后才能使用迁移命令

```
Install-Package Microsoft.EntityFrameworkCore.Tools
```

EF core连接Oracle

```
Install-Package Oracle.EntityFrameworkCore
```

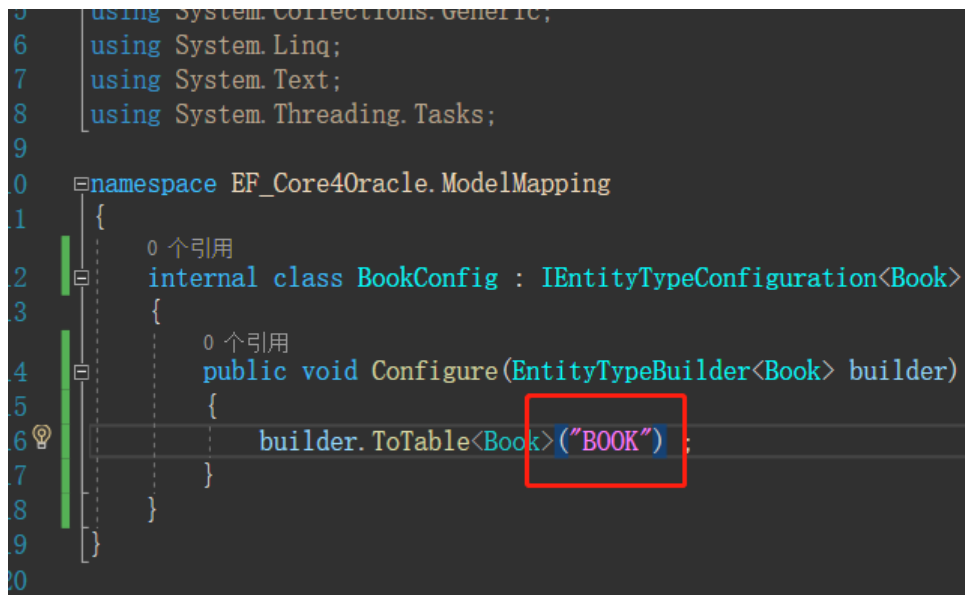
特别注意:由于EF Core生成的创表脚本带引号,导致表名和字段名区分大小写,在pl/sql查询不方便

现有以下方案:

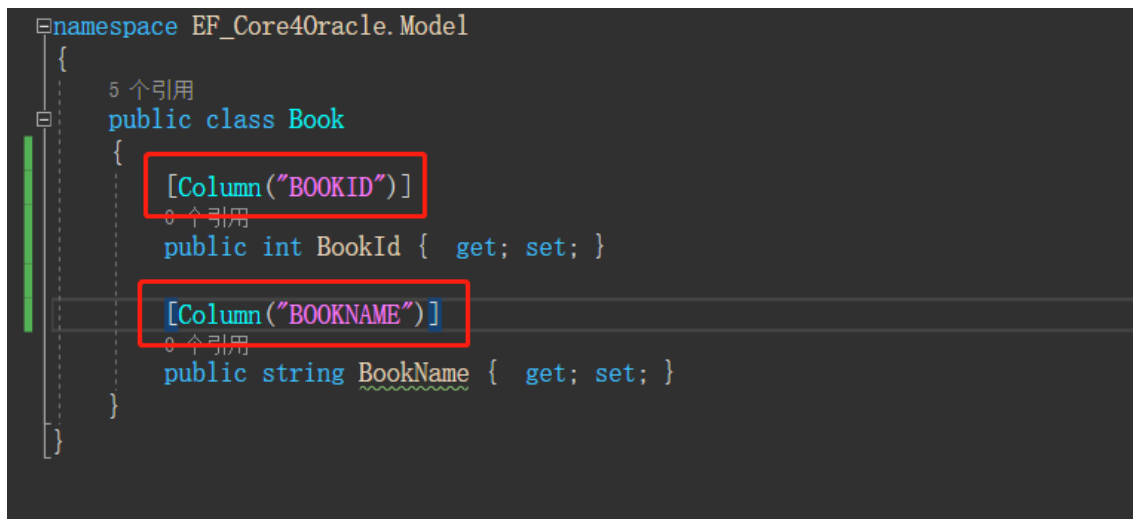
1. 使用DBFirst(反向工程),详情请见[反向工程 - EF Core | Microsoft Docs](#)

```
Scaffold-DbContext 'Data Source=(localdb)\MSSQLLocalDB;Initial Catalog=Chinook' Oracle.EntityFrameworkCore
```

2. 第二种方案是在表名指定大写并且属性上标记大写的字段名



```
5 using System.Collections.Generic;
6 using System.Linq;
7 using System.Text;
8 using System.Threading.Tasks;
9
10 namespace EF_Core4Oracle.ModelMapping
11 {
12     0 个引用
13     internal class BookConfig : IEntityTypeConfiguration<Book>
14     {
15         0 个引用
16         public void Configure(EntityTypeBuilder<Book> builder)
17         {
18             builder.ToTable<Book>("BOOK");
19         }
20     }
21 }
```



创建表结构对应的实体类Books

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Ef_CoreDeno
{
    public class Books
    {
        /// <summary>
        /// 主键
        /// </summary>
        public long Id { get; set; }

        /// <summary>
        /// 标题
        /// </summary>
        public string Title { get; set; }

        /// <summary>
        /// 发布时间
        /// </summary>
        public DateTime PubTime { get; set; }

        /// <summary>
        /// 单价
        /// </summary>
        public Double Price { get; set; }
    }
}
```

使用命令生成数据库

```
Add-Migration InitialCreate
```

```
Update-database
```

以上迁移命令适用于开发阶段，不适用于生产库，生产库应该生成sql语句进行迁移

下面的内容生成一个从空白数据库到最新迁移的 SQL 脚本：

```
Script-Migration
```

以下生成从给定的迁移到最新迁移的 SQL 脚本。(即生成从名字为 **AddNewTables** 的迁移版本到最新迁移版本的sql语句)

```
Script-Migration AddNewTables
```

生成从指定的 **AddNewTables** 版本迁移到**AddAuditTable**移的 SQL 脚本，
如果**AddNewTables** 的版本号比**AddAuditTable**的版本号高，则是回退版本

```
Script-Migration AddNewTables AddAuditTable
```

删除迁移(未更新到数据库时可使用以下命令)

注:更新到数据库后使用以下命令会报以下错误`The migration '20211023104245_InitialCreateThird' has already been applied to the database. Revert it and try again. If the migration has been applied to other databases, consider reverting its changes using a new migration instead.

```
Remove-Migration
```

、

回退数据库版本及迁移

回退数据库到最初始版本

```
Update-Database -Migration:0
```

回退数据库到某个版本

```
Update-Database -Migration:20211022145430_InitialCreate2
```

最后再删除最近一次迁移(需先回退数据库版本)

```
Remove-Migration
```

关系配置

- 注，SqlServer会自动开启级联删除，要先关掉

一对多

(Article类中定义一个Comment的list集合属性，Comment类中定义一个Article类型的属性)

以下配置一端即可

```
using Ef_CoreDeno.Model;  
using Microsoft.EntityFrameworkCore;
```

```

using Microsoft.EntityFrameworkCore.Metadata.Builders;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Ef_CoreDeno.Mapping
{
    public class ArticleEntryConfig : IEntityTypeConfiguration<Article>
    {
        public void Configure(EntityTypeBuilder<Article> builder)
        {
            builder.ToTable<Article>("T_Article");
            builder.HasMany<Comment>(a =>
a.comments).withOne(c=>c.articleMap).HasPrincipalKey(a=>a.title);//这边配置后另一边
可不用配置
        }
    }
}
//HasPrincipalKey是指定Article的某一列为Comment的外键值

```

```

using Ef_CoreDeno.Model;
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Ef_CoreDeno.Mapping
{
    internal class CommentEntryConfig : IEntityTypeConfiguration<Comment>
    {
        public void Configure(EntityTypeBuilder<Comment> builder)
        {
            builder.ToTable<Comment>("T_Comment");
            builder.HasOne<Article>(b => b.articleMap).withMany(a=>a.comments);
        }
    }
}

```

单向导航属性

一对多的配置HasOne(r => r.User).WithMany()中WithMany()方法不写任何参数即可(配置要写在引用的一方)

一对一

- 在一对多关系中，具有引用导航的实体是依赖实体，具有集合的实体是主体。但在一对一关系中，这种情况并不如此 -因此需要显式定义外键

```
HasOne<Husband>(w => w.Husband).WithOne(w=>w.Wife).HasForeignKey<Husband>(w=>w.HusbandId);
```

多对多

```
modelBuilder.Entity<Post>().HasMany(p => p.Tags).WithMany(p => p.Posts).UsingEntity(j => j.ToTable("PostTags"));
```

EF Core执行原生Sql

执行非查询语句

使用字符串内插可防止sql注入

```
using (DemoDbContext demoDbContext = new DemoDbContext())
{
    string Page = "66";
    double Price = 6.0;
    DateTime dateTime = DateTime.Now;
    string Title = "sda";
    string publisher = "123156";
    demoDbContext.Database.ExecuteSqlInterpolated(@$"INSERT INTO [T_Books] (
        [Page], [Price], [PubTime], [Title], [publisher] )
                                VALUES
                                ({Page}, {Price},
        {dateTime}, {Title}, {publisher})");
}
```

执行与DbContext实体相关的相关的sql语句

同样的使用字符串内插可防止sql注入

```
int Id = 1;
Books? books = demoDbContext.Set<Books>().FromSqlInterpolated($"SELECT *
FROM [dbo].[T_Books] where id={Id}").FirstOrDefault();
```

如果以上不能满足可以调用 `GetDbConnection()` 方法获得数据库连接对象，再使用ADO.net进行执行原生sql

```
System.Data.Common.DbConnection dbConnection =
demoDbContext.Database.GetDbConnection();
```

或者使用Dapper等轻量级框架

代码操作文档及仓库地址

[DapperLib/Dapper: Dapper - a simple object mapper for .Net \(github.com\)](https://github.com/DapperLib/Dapper)

Nuget安装Dapper包

```
Install-Package Dapper
```

```
using Dapper;
//Dapper是对IDbConnection进行方法扩展
System.Data.Common.DbConnection dbConnection =
demoDbContext.Database.GetDbConnection();
IEnumerable<Books> enumerable = dbConnection.Query<Books>(@"SELECT * FROM [dbo].[T_Books]");
```

EF Core的五种追踪状态

- Added(已添加):DbContext正在跟踪此实体, 但是数据库还没有这个数据
- Unchange(未改变):DbContext正在跟踪此实体,并且该实体的数据与数据库的一致
- Modified(已修改):DbContext正在跟踪此实体,与数据库中的数据不一致, 在调用SaveChanges()后会更新数据库中的数据
- Deleted(已删除):DbContext正在跟踪此实体,数据库中还有数据,在调用SaveChanges()后会将数据路中的数据删除
- Detached(分离):DbContext未跟踪此实体

使用EntityEntry的State属性来查看实体状态

```
// See https://aka.ms/new-console-template for more information
using Dapper;
using Ef_CoreDeno;
using Ef_CoreDeno.Model;
using Microsoft.EntityFrameworkCore;

using (DemoDbContext demoDbContext = new DemoDbContext())
{
    Books books = new Books();
    Microsoft.EntityFrameworkCore.ChangeTracking.EntityEntry<Books> entityEntry4
= demoDbContext.Entry(books);
    Console.WriteLine(entityEntry4.State.ToString());
    demoDbContext.Set<Books>().Add(books);
    Microsoft.EntityFrameworkCore.ChangeTracking.EntityEntry<Books> entityEntry
= demoDbContext.Entry(books);
    Console.WriteLine(entityEntry.State.ToString());
    Books books1 = demoDbContext.Set<Books>().FirstOrDefault();
    Microsoft.EntityFrameworkCore.ChangeTracking.EntityEntry<Books?>
entityEntry1 = demoDbContext.Entry(books1);
    Console.WriteLine(entityEntry1.State.ToString());
    books1.Title = "11111";
    Microsoft.EntityFrameworkCore.ChangeTracking.EntityEntry<Books> entityEntry2
= demoDbContext.Entry(books1);
    Console.WriteLine(entityEntry2.State.ToString());
    demoDbContext.Set<Books>().Remove(books1);
    Microsoft.EntityFrameworkCore.ChangeTracking.EntityEntry<Books> entityEntry3
= demoDbContext.Entry(books1);
    Console.WriteLine(entityEntry3.State.ToString());
}
```

```
Books books = new Books();
Microsoft.EntityFrameworkCore.ChangeTracking.EntityEntry<Books> entityEntry4 = demoDbContext.Entry(books);
Console.WriteLine(entityEntry4.State.ToString());
demoDbContext.Set<Books>().Add(books);
Microsoft.EntityFrameworkCore.ChangeTracking.EntityEntry<Books> entityEntry = demoDbContext.Entry(books);
Console.WriteLine(entityEntry.State.ToString());
Books books1 = demoDbContext.Set<Books>().FirstOrDefault();
Microsoft.EntityFrameworkCore.ChangeTracking.EntityEntry<Books?> entityEntry1 = demoDbContext.Entry(books1);
Console.WriteLine(entityEntry1.State.ToString());
books1.Title = "11111";
Microsoft.EntityFrameworkCore.ChangeTracking.EntityEntry<Books> entityEntry2 = demoDbContext.Entry(books1);
Console.WriteLine(entityEntry2.State.ToString());
demoDbContext.Set<Books>().Remove(books1);
Microsoft.EntityFrameworkCore.ChangeTracking.EntityEntry<Books> entityEntry3 = demoDbContext.Entry(books1);
Console.WriteLine(entityEntry3.State.ToString());
```

使用DebugView.LongView查看实体的具体状态和变化

```
//IEnumerable<Books> enumerable = dbConnection.Query<Books>(@"SELECT * FROM [dbo].[T_Books]");
Books books = new Books();
Microsoft.EntityFrameworkCore.ChangeTracking.EntityEntry<Books> entityEntry4 = demoDbContext.Entry(books);
//Console.WriteLine(entityEntry4.State.ToString());
demoDbContext.Set<Books>().Add(books);
Microsoft.EntityFrameworkCore.ChangeTracking.EntityEntry<Books> entityEntry = demoDbContext.Entry(books);
//Console.WriteLine(entityEntry.State.ToString());
Books books1 = demoDbContext.Set<Books>().FirstOrDefault();
Microsoft.EntityFrameworkCore.ChangeTracking.EntityEntry<Books?> entityEntry1 = demoDbContext.Entry(books1);
//Console.WriteLine(entityEntry1.State.ToString());
books1.Title = "11111";
Microsoft.EntityFrameworkCore.ChangeTracking.EntityEntry<Books> entityEntry2 = demoDbContext.Entry(books1);
//Console.WriteLine(entityEntry2.State.ToString());
Console.WriteLine(entityEntry2.DebugView.LongView);
demoDbContext.Set<Books>().Remove(books1);
Microsoft.EntityFrameworkCore.ChangeTracking.EntityEntry<Books?> entityEntry3 = demoDbContext.Entry(books1);
//Console.WriteLine(entityEntry3.State.ToString());
```

使用AsNoTracking()方法让DbContext不跟踪查出来的实体

```
using Dapper;
using Ef_CoreDeno;
using Ef_CoreDeno.Model;
using Microsoft.EntityFrameworkCore;

using (DemoDbContext demoDbContext = new DemoDbContext())
{
    Books? books = demoDbContext.Set<Books>().AsNoTracking().FirstOrDefault();
    Microsoft.EntityFrameworkCore.ChangeTracking.EntityEntry<Books?> entityEntry
    = demoDbContext.Entry(books);
    Console.WriteLine(entityEntry.State.ToString());
}
```

```
#endif
Books? books = demoDbContext.Set<Books>().AsNoTracking().FirstOrDefault();
Microsoft.EntityFrameworkCore.ChangeTracking.EntityEntry<Books?> entityEntry = demoDbContext.Entry(books);
Console.WriteLine(entityEntry.State.ToString());
```

全局查询筛选器

定义:EF在每次查询时会在sql的Where条件后自动增加条件(用途示例:默认查询软删除后有效的数据)

```
modelBuilder.Entity<Post>().HasQueryFilter(p => !p.IsDeleted);
```

可以在查询时使用IgnoreQueryFilters()方法禁用筛选器

```
blogs = db.Blogs
    .Include(b => b.Posts)
    .IgnoreQueryFilters()
    .ToList();
```

EF Core并发锁机制

- EF Core不自带悲观锁
- EF Core乐观锁

1. 并发令牌IsConcurrencyToken

原理:当更新标记为并发令牌的字段时,EF Core会在更新这个字段时关联条件带上旧值,进行更新,即

```
UPDATE T_Books SET Title = '新值' WHERE Title = '旧值' ;
```

当更新影响条数为0时,则说明产生并发,会报异常

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Books>()
        .Property(b => b.Title)
        .IsConcurrencyToken();
}
```

2. Timestamp/rowversion(支持SQLserver)

原理:每改动某一行中的某一列数据都会改动rowversion的值,关联条件带上rowversion的旧值,进行更新

```
internal class MyContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Blog>()
            .Property(p => p.Timestamp)
            .IsRowVersion(); //将byte数组属性标记为RowVersion
    }
}

public class Blog
{
    public int BlogId { get; set; }
    public string Url { get; set; }
    public byte[] Timestamp { get; set; } //定义一个byte数组属性
}
```


使用EFCore注意事项

- 不能使用DbContext嵌套遍历不同的实体类，否则会报错，原因是使用延迟加载时是使用DataReader从服务端不断取数据的而一个连接只能有一个已经打开状态的DataReader

```
foreach (Article article in demoDbContext.Set<Article>())
{
    Console.WriteLine($"{article.articleId}{article.title}");
    foreach (var item in demoDbContext.Set<Comment>())
    {
        Console.WriteLine($"{item.CommentId}{item.Message}
{item.articleMap.articleId}");
    }
}
```



- 如果DbContext被释放了，遍历IQueryable会报错

```
IQueryable<Article> articles = Test<Article>();
foreach (var item in articles)
{

}

static IQueryable<T> Test<T>() where T : class
{
    using (DemoDbContext demoDbContext = new DemoDbContext())
    {
        return demoDbContext.Set<T>().AsQueryable();
    }
}
```



- SaveChanges方法里会包含事务，如果再自己操作事务报错

