EF Core-关系

实体间的关系

一对一、一对多（在子中指定父Id）、多对多（使用第三方表）。

EF Core不仅支持单实体操作，更支持多实体的关系操作。

三部曲：实体类中关系属性；FluentAPI关系配置（重点）；使用关系操作。

一对多

比如，文章实体类Article，评论实体类Comment。一篇文章对应多条评论。

1）实体类中关系属性

建议List<>属性都初始化。

```
public class Article
{
    public long Id { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }
    public List<Comment> Comments { get; set; }=new List<Comment>();
}
```

```
public class Comment
{
    public long Id { get; set; }
    public string Message { get; set; }
    public Article Article { get; set; }
}
```

2）FluentAPI关系配置

EF Core中实体之间关系的配置套路：

```
HasXXX(...).WithXXX(...);//有XXX、反之带有XXX，XXX可选One、Many。
HasOne(...).WithMany(...);//一对多，在多表中配置外键指向一表
HasOne(...).WithOne(...);//一对一
HasMany(...).WithMany(...);//多对多
```

```
public class ArticleConfig:IEntityTypeConfiguration<Article>
{
    public void Configure(EntityTypeBuilder<Article> builder)
    {
        builder.Property(a => a.Title).IsRequired().IsUnicode().HasMaxLength(255);
        builder.Property(a => a.Content).IsRequired().IsUnicode();
    }
}
```

```
public class CommentConfig:IEntityTypeConfiguration<Comment>
{
    public void Configure(EntityTypeBuilder<Comment> builder)
    {
        builder.Property(c => c.Message).IsRequired().IsUnicode();
        builder.HasOne<Article>(c => c.Article).WithMany(a => a.Comments).IsRequired();
    }
}
```

3）使用关系操作

不需要显式为Comment对象的Article属性赋值，也不需要显式的创建Comment类型的对象添加到DbContext中，EF Core会“顺竿爬”。

```
await using TestDbContext context = new TestDbContext();

Article a1=new Article{Title = "felix被评为读者之星",Content = "据报道。。。。"};
await context.Articles.AddAsync(a1);

Comment c1=new Comment{Message = "太牛了"};
Comment c2=new Comment{Message = "吹吧你"};
a1.Comments.Add(c1);
a1.Comments.Add(c2);

await context.SaveChangesAsync();
```

获取一对多数据，使用Include

正向，反向都可以查。

```
var a= await context.Articles.Include(a=>a.Comments).FirstOrDefaultAsync(a=>a.Id==1);
Console.WriteLine(a!.Title);
foreach (var c in a.Comments)
{
    Console.WriteLine(c.Message);
}
```

```
var c=await context.Comments.Include(c=>c.Article).FirstOrDefaultAsync(c=>c.Id==1);
Console.WriteLine(c!.Message);
Console.WriteLine(c.Article.Title);
```

额外的外键字段

为什么需要外键属性？

1）EF Core会在数据表中建外键列。

2）如果需要获取外键列的值，就需要做关联查询，效率低。

3）需要一种不需要Join直接获取外键列的值的方式。

比如，如果想在Comment中只想获取Article的Id信息，而不想获取全部的Article内容。

在数据库中尽量不要select * from table，这样会获取全部table的字段，效率低。

```
//var a1=await context.Articles.FirstAsync(); //生成的SQL中包含了Content
//SELECT TOP(1) [a].[Id], [a].[Content], [a].[Title] FROM[Articles] AS[a]
var a1 = await context.Articles.Select(a=>new{a.Id,a.Title}).FirstAsync();//优化查询
//SELECT TOP(1) [a].[Id], [a].[Title]FROM[Articles] AS[a]
Console.WriteLine($"id:{a1.Id},title:{a1.Title}");
```

从comment中，使用Select依然会生成带Join的SQL语句，如何生成不带Join的SQL来提高效率呢？

```
var c1=await context.Comments.Select(c=>new{c.Id,AId=c.Article.Id}).FirstAsync();
// SELECT TOP(1) [c].[Id], [a].[Id] AS [AId] FROM[Comments] AS[c]
//INNER JOIN[Articles] AS [a] ON[c].[ArticleId] = [a].[Id]
Console.WriteLine($"comment Id:{c1.Id},article id:{c1.AId}");
```

设置外键属性

解决办法时使用额外的外键字段。

```
public class Comment
{
    public long Id { get; set; }
    public string Message { get; set; }
    //导航属性
    public Article Article { get; set; }
    //额外外键字段
    public long ArticleId { get; set; }
}
```

然后在关系配置中，通过`HasForeignKey(c=>c.ArticleId)`，指定这个属性为外键。除非必要，否则不用声明，否则会引入重复。

```
public class CommentConfig:IEntityTypeConfiguration<Comment>
{
    public void Configure(EntityTypeBuilder<Comment> builder)
    {
        builder.Property(c => c.Message).IsRequired().IsUnicode();
        builder.HasOne<Article>(c => c.Article).WithMany(a => a.Comments).HasForeignKey(c=>c.ArticleId).IsRequired();
    }
}
```

然后可以直接查询到对应的ArticleId，并且SQL语句中没有Join关联，提高查询效率。

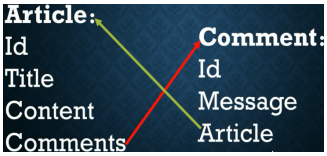
```
var c1 = await context.Comments.FirstAsync();
//SELECT TOP(1) [c].[Id], [c].[ArticleId], [c].[Message] FROM[Comments] AS[c]
Console.WriteLine($"comment Id:{c1.Id},article id:{c1.ArticleId}");
```

通常情况下只需关注Linq查询而无需关注SQL语句，只是在遇到查询性能问题时，再来看看生成的SQL语句，再进一步优化查询。EF Core大部分查询比大部分普通程序员写出来的SQL性能高，有少部分自动生成SQL语句可能不尽人意，但是影响不大。

单向导航属性

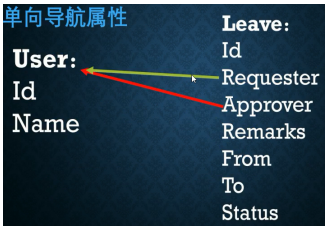
导航属性，由该属性能访问到另一个类型的实体。

再之前的Article和Comment两个类中，都定义了对方的导航属性，这种方式叫做双向导航属性。双向导航属性给类的访问非常方便。



但是有的时候，数据结构非常复杂，而且有时不需要双向导航。

比如User表，会被非常多的表格引用，而且一个表格中可能多个属性都需要指向User，如果都双向指定，会导致User表太复杂。这种情况下只需要在请假单表、报销表、采购单表、提货单表、离职单等表中指向User即可，而User中无需再指定表格。比如如下的请假单：




```
public class User
{
    public long Id { get; set; }
    public string Name { get; set; }
}
```

```
public class Leave
{
    public long Id { get; set; }
    public string Remarks { get; set; }

    public User Requester { get; set; }//申请人
    public User? Approver { get; set; } //审批人，注意标记可空
}
```

配置，因为User不知道由Leave存在，因此只能在Leave中配置User的一对多。

配置时必须写WithMany()，但不设置参数即可。

```
public class LeaveConfig:IEntityTypeConfiguration<Leave>
{
    public void Configure(EntityTypeBuilder<Leave> builder)
    {
        builder.HasOne<User>(l => l.Requester).WithMany().IsRequired();
        builder.HasOne<User>(l => l.Approver).WithMany();
    }
}
```

由于只有Leave指向User，因此在插入数据时只能从Leave开始，而不能从User开始。

模拟填写请假单。

```
User u1 = new User {Name = "小峰"};
Leave l1=new Leave{Remarks = "回家处理拆迁事宜",Requester = u1};
await context.Leaves.AddAsync(l1);
await context.SaveChangesAsync();
```

模拟批准请假单。

```
User u2 = new User { Name = "朱总" };
User u1 =await context.Users.SingleAsync(u=>u.Name== "小峰");
Leave l1 = await context.Leaves.Where(l => l.Requester == u1).FirstAsync();
l1.Approver = u2;
await context.SaveChangesAsync();
```

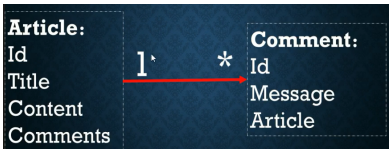
对于**主从结构**的一对多关系，一般时声明双向导航属性。而对于其他一对多关系，如果表属于被很多表引用的基础表，则用单向导航属性。

主从结构，如采购单与采购明细，购物车与商品，文章与评论等。

关系配置在任何一方都可以

正反的概念，比如一条高速路，在吉安的人叫吉石高速，在石城的人叫石吉高速。

一对多，多对一，站的角度不一样，其实描述的是同一件事。



因此在双向导航中，正反配置都可以。但是单项导航只能配置到有导航属性的类中。

```
//CommentConfig，Connent有一个Article，一个Article有多个Comment
builder.HasOne<Article>(c => c.Article).WithMany(a => a.Comments).HasForeignKey(c=>c.ArticleId).IsRequired();
```

```
//ArticleConfig，Articel有多个Comment，一个Comment有一个Article
builder.HasMany<Comment>(a=>a.Comments).WithOne(c=>c.Article).HasForeignKey(c=>c.ArticleId).IsRequired();
```

考虑到单项导航属性的可能，推荐都在One中配置，简化问题。`HasOne().WithMany();`

自引用的组织结构树

相同类型的一对多关系，例如组织结构，组织单元之间有父子结构。节点与子节点有一对多的关系。

```
public class OrgUnit
{
    public long Id { get; set; }
    public string Name { get; set; }
```

```
    public OrgUnit? Parent { get; set; }
    public List<OrgUnit> Children { get; set; }=new List<OrgUnit>();
}
```

```
public class OrgUnitConfig:IEntityTypeConfiguration<OrgUnit>
{
    public void Configure(EntityTypeBuilder<OrgUnit> builder)
    {
        builder.Property(o => o.Name).IsRequired().HasMaxLength(50);
        //因为根节点没有parent，因此不能修饰为IsRequired()
        builder.HasOne<OrgUnit>(o => o.Parent).WithMany(o=>o.Children);
    }
}
```

测试添加自引用组织结构树：

```
//既可以设置一个ou的Parent，也可以把节点加入父节点的Children.Add(...)，推荐使用后者
OrgUnit ouRoot=new OrgUnit{Name = "全球总部"};
OrgUnit ouAsia = new OrgUnit {Name = "亚洲总部"};
OrgUnit ouChina = new OrgUnit { Name = "中国分部"};
OrgUnit ouSg = new OrgUnit { Name = "新加坡分部"};
OrgUnit ouAmerica = new OrgUnit {Name = "美洲总部"};
OrgUnit ouUsa = new OrgUnit {Name = "美国分部"};
OrgUnit ouCan = new OrgUnit {Name = "加拿大分部"};
ouRoot.Children.Add(ouAsia);
ouRoot.Children.Add(ouAmerica);
ouAsia.Children.Add(ouChina);
ouAsia.Children.Add(ouSg);
ouAmerica.Children.Add(ouUsa);
ouAmerica.Children.Add(ouCan);
await context.OrgUnits.AddAsync(ouRoot);
await context.SaveChangesAsync();
```

推荐使用指定父节点的方式，然后使用AddRangeAsync()。

```
OrgUnit ouRoot = new OrgUnit { Name = "全球总部" };
OrgUnit ouAsia = new OrgUnit { Name = "亚洲总部",Parent = ouRoot};
OrgUnit ouChina = new OrgUnit { Name = "中国分部",Parent = ouAsia};
OrgUnit ouSg = new OrgUnit { Name = "新加坡分部",Parent = ouAsia};
OrgUnit ouAmerica = new OrgUnit { Name = "美洲总部",Parent = ouRoot};
OrgUnit ouUsa = new OrgUnit { Name = "美国分部",Parent = ouAmerica};
OrgUnit ouCan = new OrgUnit { Name = "加拿大分部" ,Parent = ouAmerica};
await context.OrgUnits.AddRangeAsync(ouRoot, ouAsia, ouChina, ouSg, ouAmerica, ouUsa, ouCan);
await context.SaveChangesAsync();
```

测试递归缩进打印

```
//首先找到根节点，父节点为空的节点为根节点
var ouRoot = context.OrgUnits.Single(o => o.Parent == null);
Console.WriteLine(ouRoot.Name);
PrintChildren(1, ouRoot);


//测试递归缩进打印
void PrintChildren(int identLevel, OrgUnit parent)
{
    //先找到父节点的子节点
    var children = context.OrgUnits.Where(o => o.Parent == parent);
    foreach (var child in children)
    {
        Console.WriteLine(new string('\t', identLevel)+child.Name);//identLevel为\t的重复次数
        PrintChildren(identLevel++, child);//递归，打印子节点的子节点
    }
}
```

程序报错：There is already an open DataReader associated with this Connection which mu……

解决方法：数据库连接字符串加上“MultipleActiveResultSets = true”。

很多数据库的ADO.NET Core Provider是不支持多个DataReader同时执行的。

```
var connStr = @"Server=PDMSERVER\SQLEXPRESS; Database=EFCoreRelationDB; User Id = sa; Password=Epdm2018;TrustServerCertificate=true;Mu
```

一对一

采购申请单与采购订单，订单Order与快递单Delivery。

必须**显式的**在其**中一个**实体类中声明一个外键属性，如在Delivery中声明OrderId外键属性，最好在后产生的实体类中指定先产生的实体类的Id。

```
public class Order
{
    public long Id { get; set; }
    public string Name { get; set; }
    public string Address { get; set; }
    public Delivery Delivery { get; set; }
}
```

```
public class Delivery
{
    public long Id { get; set; }
    public string CompanyName { get; set; }
    public string Number { get; set; }
    public Order Order { get; set; }
    public long OrderId { get; set; }
}
```

配置关系，HasOne<>().WithOne().HasForeignKey<>()。

```
public class DeliveryConfig:IEntityTypeConfiguration<Delivery>
    {
        public void Configure(EntityTypeBuilder<Delivery> builder)
        {
            builder.HasOne<Order>(d=>d.Order).WithOne(o=>o.Delivery).HasForeignKey<Delivery>(d=>d.OrderId);
        }
    }
```

一对一数据的插入

```
Order o1=new Order{Name = "书"};
Delivery d1=new Delivery {CompanyName = "峰峰快递", Number = "fengfeng0001", Order = o1};
//存Delivery才能找到Order，否则存Order找不到Delivery,如果搞不清楚关系，则全部加进去
await context.Deliveries.AddAsync(d1);
await context.SaveChangesAsync();
```

多对多

多对多关系，比如老师和学生。

EF Core5.0开始，正式支持多对多。但是无论如何，在数据库层面必须有一张中间关系表存在。

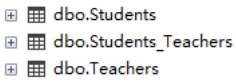
双方都建立对方的List导航属性。

```
public class Student
{
    public long Id { get; set; }
    public string Name { get; set; }
    public List<Teacher> Teachers { get; set; }=new List<Teacher>();
}
```

```
public class Teacher
{
    public long Id { get; set; }
    public string Name { get; set; }
    public List<Student> Students { get; set; }=new List<Student>();
}
```

在其中一方配置多对多关系，然后指定中间表的名字（如果不指定，EFCore会自动创建默认表）。建议不要手动指定表明，使用系统默认表明就好了。

```
public class StudentConfig:IEntityTypeConfiguration<Student>
{
    public void Configure(EntityTypeBuilder<Student> builder)
    {
        //ToTable指定中间表的名字
        builder.HasMany<Teacher>(s => s.Teachers).WithMany(t => t.Students)
            .UsingEntity(r => r.ToTable("Students_Teachers"));
    }
}
```



多对多数据插入。

```
Student s1=new Student{Name = "张三"};
Student s2=new Student{Name = "李四"};
Student s3=new Student{Name = "王五"};
Teacher t1=new Teacher{Name = "jack"};
Teacher t2=new Teacher{Name = "tom"};
Teacher t3=new Teacher{Name = "jerry"};
s1.Teachers.Add(t1);
s1.Teachers.Add(t2);
s2.Teachers.Add(t2);
s2.Teachers.Add(t3);
s3.Teachers.Add(t1);
s3.Teachers.Add(t2);
s3.Teachers.Add(t3);
//单向添加，不要再反向添加，否则容易将关系搞混
//添加到数据库时，将实体全部加入，防止遗漏
await context.Students.AddRangeAsync(s1, s2, s3);
await context.Teachers.AddRangeAsync(t1, t2, t3);
await context.SaveChangesAsync();
```

查询老师，并列出他们的学生。

```
var teachers=context.Teachers.Include(t => t.Students);
foreach (var teacher in teachers)
{
    Console.WriteLine(teacher.Name);
    foreach (var student in teacher.Students)
    {
        Console.WriteLine($"\t{student.Name}");
    }
}
```