DDD领域驱动设计初探（3）：仓储Repository（下） - 文章 - 伯乐在线

前言：上篇介绍了下仓储的代码架构示例以及简单分析了仓储了使用优势。本章还是继续来完善下仓储的设计。上章说了，仓储的最主要作用的分离领域层和具体的技术架构，使得领域层更加专注领域逻辑。那么涉及到具体的实现的时候我们应该怎么做呢，本章就来说说仓储里面具体细节方便的知识。

一、对仓储接口以及实现基类的完善

1、仓储实现基类的所有方法加上virtual关键字，方便具体的仓储在特定需求的时候override基类的方法。

C#

```
    //仓储的泛型实现类
        public class EFBaseRepository : IRepositorywhere TEntity : AggregateRoot
        {
                [Import(typeof(IEFUnitOfWork))]
                public IEFUnitOfWork UnitOfWork { get; set; }
                public EFBaseRepository()
                {
                        Regisgter.regisgter().ComposeParts(this);
                }
                public virtual IQueryable Entities
                {
                        get { return UnitOfWork.context.Set(); }
                }
                public virtual TEntity GetByKey(object key)
                {
                        return UnitOfWork.context.Set().Find(key);
                }
                public virtual IQueryable Find(Expressionbool>> express)
                {
                        Funcbool> lamada = express.Compile();
                        return UnitOfWork.context.Set().Where(lamada).AsQueryable();
                }
                public virtual int Insert(TEntity entity)
```

```
{
        UnitOfWork.RegisterNew(entity);
        return UnitOfWork.Commit();
}
public virtual int Insert(IEnumerable entities)
{
        foreach (var obj in entities)
        {
                UnitOfWork.RegisterNew(obj);
        }
        return UnitOfWork.Commit();
}
public virtual int Delete(object id)
{
        var obj = UnitOfWork.context.Set().Find(id);
        if (obj == null)
        {
                return 0;
        }
        UnitOfWork.RegisterDeleted(obj);
        return UnitOfWork.Commit();
}
public virtual int Delete(TEntity entity)
{
        UnitOfWork.RegisterDeleted(entity);
        return UnitOfWork.Commit();
}
public virtual int Delete(IEnumerable entities)
{
        foreach (var entity in entities)
        {
                UnitOfWork.RegisterDeleted(entity);
        }
        return UnitOfWork.Commit();
}
public virtual int Delete(Expressionbool>> express)
{
        Funcbool> lamada = express.Compile();
        var lstEntity = UnitOfWork.context.Set().Where(lamada);
        foreach (var entity in lstEntity)
        {
                UnitOfWork.RegisterDeleted(entity);
```

```
                }
                return UnitOfWork.Commit();
        }
        public virtual int Update(TEntity entity)
        {
                UnitOfWork.RegisterModified(entity);
                return UnitOfWork.Commit();
        }
}
```

2、查询和删除增加了传参lamada表达式的方法

仓储接口：

```
//仓储的泛型实现类
public class EFBaseRepository : IRepositorywhere TEntity : AggregateRoot
{
        //.............
        public virtual IQueryable Find(Expressionbool>> express)
        {
                Funcbool> lamada = express.Compile();
                return UnitOfWork.context.Set().Where(lamada).AsQueryable();
        }
        public virtual int Delete(Expressionbool>> express)
        {
                Funcbool> lamada = express.Compile();
                var lstEntity = UnitOfWork.context.Set().Where(lamada);
                foreach (var entity in lstEntity)
                {
                        UnitOfWork.RegisterDeleted(entity);
                }
                return UnitOfWork.Commit();
        }
//.............
}
```

增加这两个方法之后，对于单表的一般查询都可以直接通过lamada表示式的方法传入即可，并且返回值为IQueryable类型。

3、对于涉及到多张表需要连表的查询机制，我们还是通过神奇的Linq来解决。例如我们有一个通过角色取角色对应的菜单的接口需求。

在菜单的仓储接口里面：

C#

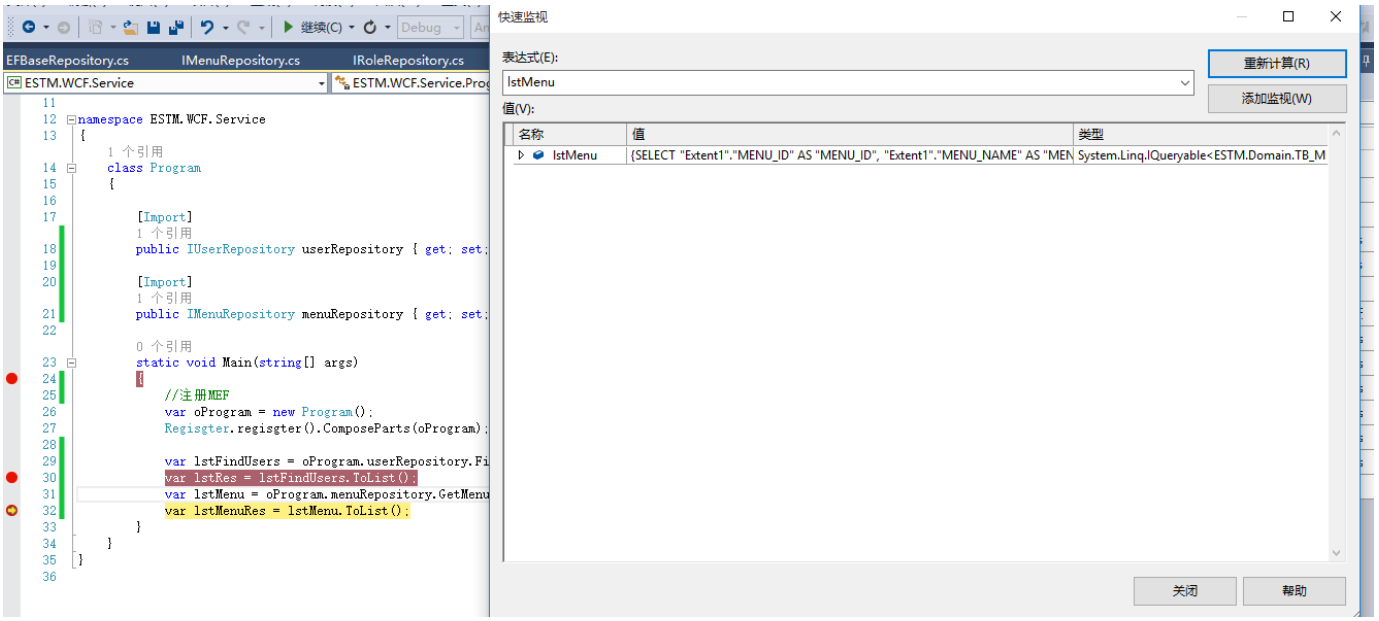
```
    [Export(typeof(IMenuRepository))]
        public class MenuRepository:EFBaseRepository,IMenuRepository
        {
                public IQueryable GetMenusByRole(TB_ROLE oRole)
                {
                        var queryrole = UnitOfWork.context.Set().AsQueryable();
                        var querymenu = UnitOfWork.context.Set().AsQueryable();
                        var querymenurole = UnitOfWork.context.Set().AsQueryable();
                        var lstres = from menu in querymenu
                                                from menurole in querymenurole
                                                from role in queryrole
                                                where menu.MENU_ID == menurole.MENU_ID &
                                                        menurole.ROLE_ID ==
role.ROLE_ID &&
                                                        role.ROLE_ID == oRole.ROLE_ID
                                                select menu;
                        return lstres;
                }
        }
```

这里也是返回的IQueryable接口的集合，千万不要小看IQueryable接口，它是一种表达式树，可以延迟查询。也就是说，在我们执行GetMenusByRole()之后，得到的是一个带有查询sql语句的表达式树结构，并没有去数据库执行查询，只有在我们ToList()的时候才会去查询数据库。我们来写个Demo测试下。

C#

```
     class Program
        {
                [Import]
                public IUserRepository userRepository { get; set; }
                [Import]
                public IMenuRepository menuRepository { get; set; }
                static void Main(string[] args)
                {
                        //注册MEF
                        var oProgram = new Program();
                        Regisgter.regisgter().ComposeParts(oProgram);
var lstFindUsers = oProgram.userRepository.Find(x => x.USER_NAME !=null);
                        var lstRes = lstFindUsers.ToList();
                        var lstMenu = oProgram.menuRepository.GetMenusByRole(new TB_ROLE()
{ ROLE_ID = "aaaa" });
                        var lstMenuRes = lstMenu.ToList();
                }
        }
```

来看执行过程：



当程序执行var lstMenu = oProgram.menuRepository.GetMenusByRole(new TB_ROLE() { ROLE_ID = “aaaa” })这一步的时候基本是不耗时的，因为这一步仅仅是在构造表达式树，只有在.ToList()的时候才会有查询等待。

在dax.net的系列文章中，提到了规约模式的概念，用于解决条件查询的问题。博主感觉这个东西设计确实牛叉，但实用性不太强，一般中小型的项目也用不上。

DDD领域驱动设计初探系列文章：