## **一、仓储的定义**

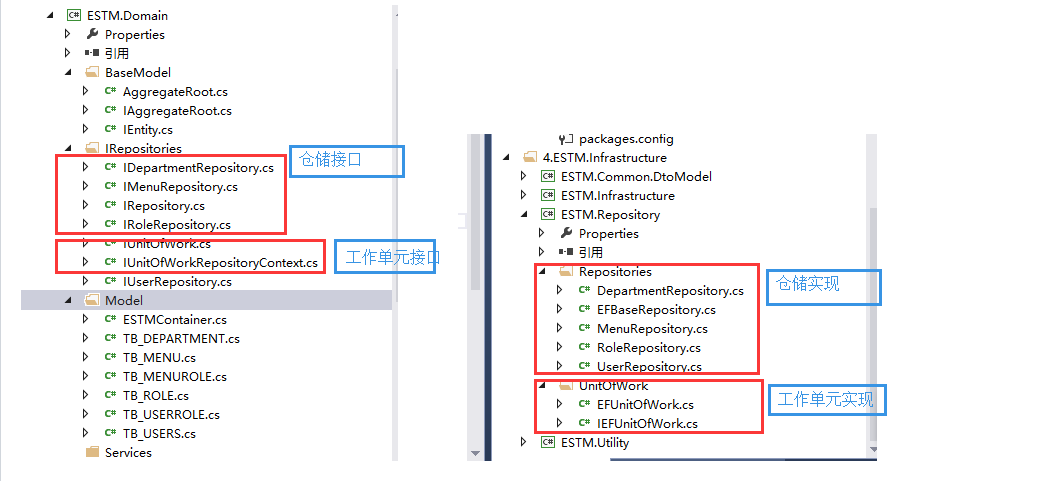
按照DDD设计原则，仓储的作用对象的领域模型的聚合根，也就是说每一个聚合都有一个单独的仓储。

## **二、使用仓储的意义**

**1、仓储作为领域层和基础结构层的连接组件，使得领域层不必过多的关注存储细节。在设计时，将仓储接口放在领域层，而将仓储的具体实现放在基础结构层2、站在架构的层面，仓储解耦了领域层和ORM之间的联系，这一点也就是很多人设计仓储模式的原因。**

## **三、代码示例**

### **1、解决方案结构图**



### **2、仓储接口**

　　 /// <summary>

/// 仓储接口，定义公共的泛型GRUD

/// </summary>

/// <typeparam name="TEntity">泛型聚合根，因为在DDD里面仓储只能对聚合根做操作</typeparam>

public interface IRepository<TEntity> where TEntity : AggregateRoot

{

#region 属性

IQueryable<TEntity> Entities { get; }

#endregion

#region 公共方法

int Insert(TEntity entity);

int Insert(IEnumerable<TEntity> entities);

int Delete(object id);

int Delete(TEntity entity);

int Delete(IEnumerable<TEntity> entities);

int Update(TEntity entity);

TEntity GetByKey(object key);

#endregion

}

/// <summary>

/// 部门聚合根的仓储接口

/// </summary>

public interface IDepartmentRepository:IRepository<TB\_DEPARTMENT>

{

}

/// <summary>

/// 菜单这个聚合根的仓储接口

/// </summary>

public interface IMenuRepository:IRepository<TB\_MENU>

{

IEnumerable<TB\_MENU> GetMenusByRole(TB\_ROLE oRole);

}

/// <summary>

/// 角色这个聚合根的仓储接口

/// </summary>

public interface IRoleRepository:IRepository<TB\_ROLE>

{

}

/// <summary>

/// 用户这个聚合根的仓储接口

/// </summary>

public interface IUserRepository:IRepository<TB\_USERS>

{

IEnumerable<TB\_USERS> GetUsersByRole(TB\_ROLE oRole);

}

### **3、仓储实现类**

　　//仓储的泛型实现类

public class EFBaseRepository<TEntity> : IRepository<TEntity> where TEntity : AggregateRoot

{

[Import(typeof(IEFUnitOfWork))]

private IEFUnitOfWork UnitOfWork { get; set; }

public EFBaseRepository()

{  
　　　　　　 //注册MEF

Regisgter.regisgter().ComposeParts(this);

}

public IQueryable<TEntity> Entities

{

get { return UnitOfWork.context.Set<TEntity>(); }

}

public int Insert(TEntity entity)

{

UnitOfWork.RegisterNew(entity);

return UnitOfWork.Commit();

}

public int Insert(IEnumerable<TEntity> entities)

{

foreach (var obj in entities)

{

UnitOfWork.RegisterNew(obj);

}

return UnitOfWork.Commit();

}

public int Delete(object id)

{

var obj = UnitOfWork.context.Set<TEntity>().Find(id);

if (obj == null)

{

return 0;

}

UnitOfWork.RegisterDeleted(obj);

return UnitOfWork.Commit();

}

public int Delete(TEntity entity)

{

UnitOfWork.RegisterDeleted(entity);

return UnitOfWork.Commit();

}

public int Delete(IEnumerable<TEntity> entities)

{

foreach (var entity in entities)

{

UnitOfWork.RegisterDeleted(entity);

}

return UnitOfWork.Commit();

}

public int Update(TEntity entity)

{

UnitOfWork.RegisterModified(entity);

return UnitOfWork.Commit();

}

public TEntity GetByKey(object key)

{

return UnitOfWork.context.Set<TEntity>().Find(key);

}

}

仓储的泛型实现类里面通过MEF导入工作单元，工作单元里面拥有连接数据库的上下文对象。

　　[Export(typeof(IDepartmentRepository))]

public class DepartmentRepository : EFBaseRepository<TB\_DEPARTMENT>,IDepartmentRepository

{

}

[Export(typeof(IMenuRepository))]

public class MenuRepository:EFBaseRepository<TB\_MENU>,IMenuRepository

{

public IEnumerable<TB\_MENU> GetMenusByRole(TB\_ROLE oRole)

{

throw new Exception();

}

}

[Export(typeof(IRoleRepository))]

public class RoleRepository:EFBaseRepository<TB\_ROLE>,IRoleRepository

{

}

[Export(typeof(IUserRepository))]

public class UserRepository:EFBaseRepository<TB\_USERS>,IUserRepository

{

public IEnumerable<TB\_USERS> GetUsersByRole(TB\_ROLE oRole)

{

throw new NotImplementedException();

}

}

仓储是4个具体实现类里面也可以通过基类里面导入的工作单元对象UnitOfWork去操作数据库。

### **4、工作单元接口**

　　 //工作单元基类接口

public interface IUnitOfWork

{

bool IsCommitted { get; set; }

int Commit();

void Rollback();

}

//仓储上下文工作单元接口，使用这个的一般情况是多个仓储之间存在事务性的操作（将增删改查功能移至工作单元中）

public interface IUnitOfWorkRepositoryContext:IUnitOfWork,IDisposable

{

/// <summary>

/// 将聚合根的状态标记为新建，但EF上下文此时并未提交

/// </summary>

/// <typeparam name="TEntity"></typeparam>

/// <param name="obj"></param>

void RegisterNew<TEntity>(TEntity obj)

where TEntity : AggregateRoot;

/// <summary>

/// 将聚合根的状态标记为修改，但EF上下文此时并未提交

/// </summary>

/// <typeparam name="TEntity"></typeparam>

/// <param name="obj"></param>

void RegisterModified<TEntity>(TEntity obj)

where TEntity : AggregateRoot;

/// <summary>

/// 将聚合根的状态标记为删除，但EF上下文此时并未提交

/// </summary>

/// <typeparam name="TEntity"></typeparam>

/// <param name="obj"></param>

void RegisterDeleted<TEntity>(TEntity obj)

where TEntity : AggregateRoot;

}

### **5、工作单元实现类**

//表示EF的工作单元接口，因为DbContext是EF的对象

public interface IEFUnitOfWork : IUnitOfWorkRepositoryContext

{

DbContext context { get; }

}

**为什么要在这里还设计一层接口？因为博主觉得，工作单元要引入EF的Context对象，同理**。

/// <summary>

/// 工作单实现类

/// </summary>

[Export(typeof(IEFUnitOfWork))]

public class EFUnitOfWork : IEFUnitOfWork

{

//通过工作单元向外暴露的EF上下文对象

public DbContext context { get { return EFContext; } }

[Import(typeof(DbContext))]

public DbContext EFContext { get; set; }

public EFUnitOfWork()

{

//注册MEF

Regisgter.regisgter().ComposeParts(this);

}

public void RegisterNew<TEntity>(TEntity obj) where TEntity : AggregateRoot

{

var state = context.Entry(obj).State;

if (state == EntityState.Detached)

{

context.Entry(obj).State = EntityState.Added;

}

IsCommitted = false;

}

public void RegisterModified<TEntity>(TEntity obj) where TEntity : AggregateRoot

{

if (context.Entry(obj).State == EntityState.Detached)

{

context.Set<TEntity>().Attach(obj);

}

context.Entry(obj).State = EntityState.Modified;

IsCommitted = false;

}

public void RegisterDeleted<TEntity>(TEntity obj) where TEntity : AggregateRoot

{

context.Entry(obj).State = EntityState.Deleted;

IsCommitted = false;

}

public bool IsCommitted { get; set; }

public int Commit()

{

if (IsCommitted)

{

return 0;

}

try

{

int result = context.SaveChanges();

IsCommitted = true;

return result;

}

catch (DbUpdateException e)

{

throw e;

}

}

public void Rollback()

{

IsCommitted = false;

}

public void Dispose()

{

if (!IsCommitted)

{

Commit();

}

context.Dispose();

}

}

**这里要想导入DbContext，那么EF的上下文对象就要Export**。

[Export(typeof(DbContext))]

public partial class ESTMContainer:DbContext

{

}

EF如下：

　　public partial class ESTMContainer : DbContext

{

public ESTMContainer()

: base("name=ESTMContainer")

{

}

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

throw new UnintentionalCodeFirstException();

}

public DbSet<TB\_DEPARTMENT> TB\_DEPARTMENT { get; set; }

public DbSet<TB\_MENU> TB\_MENU { get; set; }

public DbSet<TB\_MENUROLE> TB\_MENUROLE { get; set; }

public DbSet<TB\_ROLE> TB\_ROLE { get; set; }

public DbSet<TB\_USERROLE> TB\_USERROLE { get; set; }

public DbSet<TB\_USERS> TB\_USERS { get; set; }

}

上文中多个地方用到了注册MEF的方法

Regisgter.regisgter().ComposeParts(this);

是因为我们在基础结构层里面定义了注册方法



namespace ESTM.Infrastructure.MEF

{

public class Regisgter

{

private static object obj =new object();

private static CompositionContainer \_container;

public static CompositionContainer regisgter()

{

lock (obj)

{

try

{

if (\_container != null)

{

return \_container;

}

AggregateCatalog aggregateCatalog = new AggregateCatalog();

string path = AppDomain.CurrentDomain.BaseDirectory;

var thisAssembly = new DirectoryCatalog(path, "\*.dll");

if (thisAssembly.Count() == 0)

{

path = path + "bin\\";

thisAssembly = new DirectoryCatalog(path, "\*.dll");

}

aggregateCatalog.Catalogs.Add(thisAssembly);

\_container = new CompositionContainer(aggregateCatalog);

return \_container;

}

catch (Exception ex)

{

return null;

}

}

}

}

}

### **6、Demo测试**

namespace ESTM.WCF.Service

{

class Program

{

[Import]

public IUserRepository userRepository { get; set; }

static void Main(string[] args)

{

var oProgram = new Program();

Regisgter.regisgter().ComposeParts(oProgram);

var lstUsers = oProgram.userRepository.Entities.ToList();

}

}

}

 运行得到结果：

