系统架构师-基础到企业应用架构-数据访问层

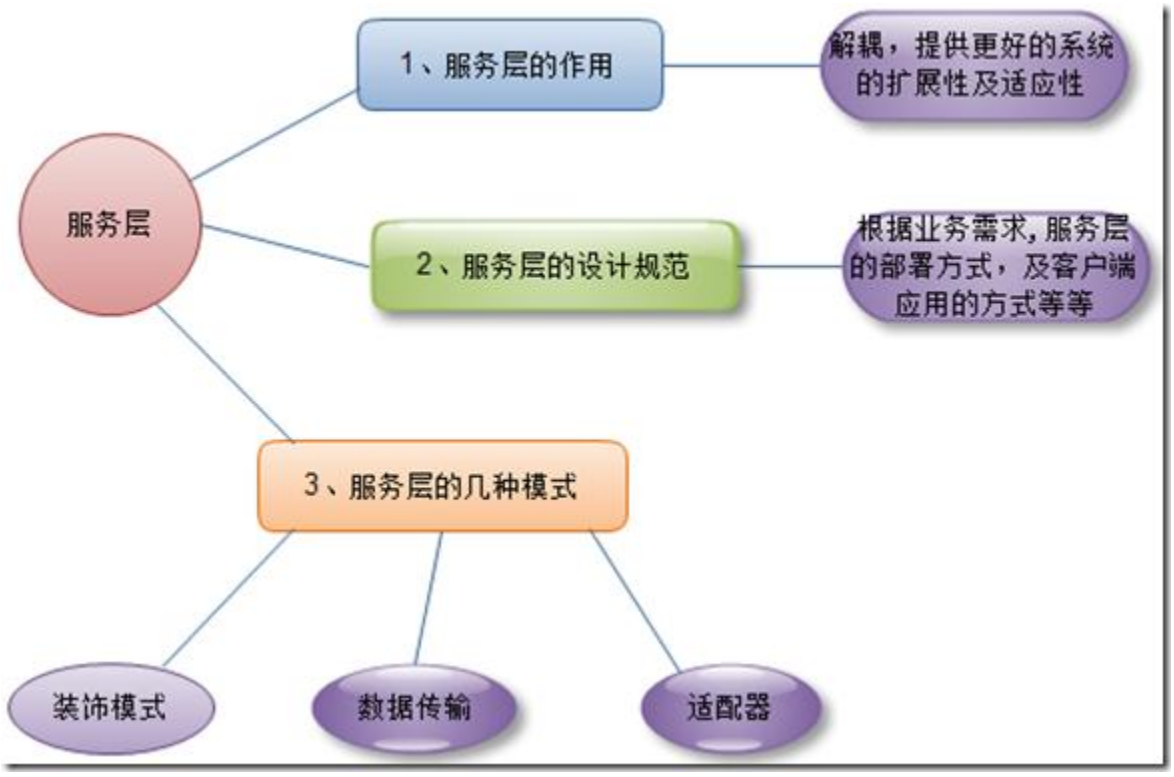
一、上章回顾

上篇我们简单讲述了服务层架构模式中的几种，并且讲解了服务层的作用及相关的设计规范，其实我们应该知道，在业务逻辑层中使用领域模型中使用服务层才

能发挥出最大的优势，如果说我们在业务逻辑层还是使用非领域模型的模式话，服务层的作用仅体现在解耦作用。其实在业务逻辑层采用领域模型时，我们前面说的持

久化透明的技术，其实我们可以通过服务层来做，我们在服务层中处理领域对象信息的持久化操作。当然本篇可能不会深入讨论持久化透明的具体实现，后面会单独开

篇来讲述，我们先来回顾下上篇讲解的内容：



上图大概描述了上篇我们讲解的内容，如果您想要详细的了解服务

层的相关内容，请参考：系统架构师-基础到企业应用架构-服务层，后续我们将会对一些前端的服务层还会进行扩展的讲解，请大家提出报告意见和建议。

二、摘要

本篇将主要以系统中与数据库存储介质进行交互的数据访问层进行详细的介绍讲解，我想这块也是大家比较熟悉也是经常在项目中一定会用到的部分，我们知道

数据访问层通常我们都把这块的内容提升出来，写成通用的类库，在我们前面讲解的分层架构的系统中，基本上可以说业务对象中的数据都要通过数据访问层将业

务数据持久化到存储介质中。其实目前有很多的好的 ORM 框架已经很好的实现了数据访问层，而且得到了很广泛的应用，当然我们本篇也会以这些通用的框架为例举

例说明数据访问层中的一些设计模式。本章将会以下列几个典型关注点展开去讲：

1、数据访问层的职责及与其他组件的交互。

2、如何设计自己的数据访问层。

3、实现数据访问层必须满足的 4 个基本要求，持久化 CRUD、查询服务、事务服务、实现并发等。

4、结合目前流行的几类框架分析框架提供的数据访问层功能的优劣。

下面我们将针对上面的几个关注点依次展开去说，希望能通过本文的讲解，让您对数据库访问层有个更深刻的认识和了解。

三、本章大纲

1、上章回顾。

2、摘要。

3、本章大纲。

4、数据访问层介绍。

5、如何设计数据访问层。

6、实现数据访问的四项原则。

7、本章总结。

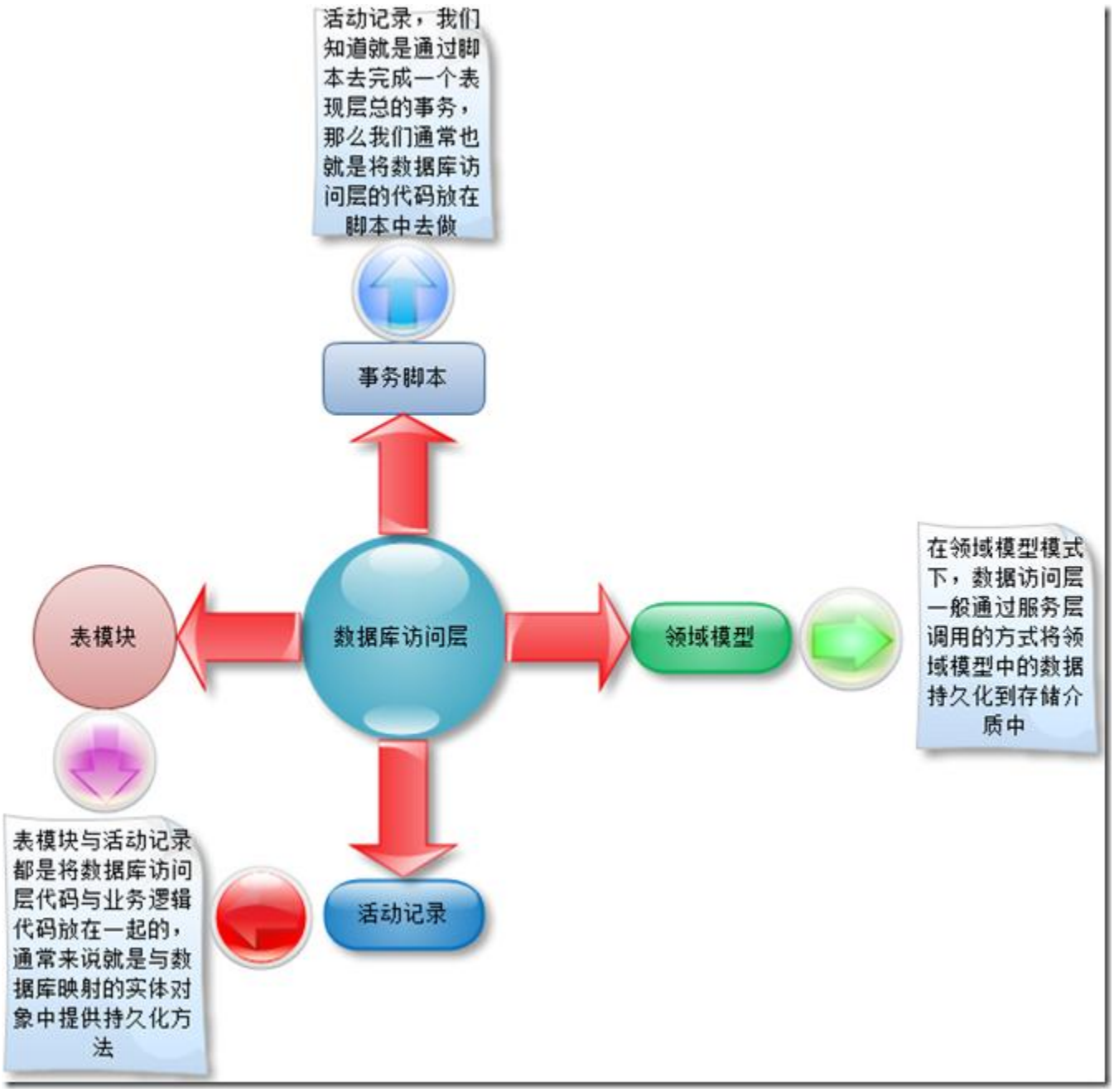
8、系列进度。

9、下篇预告。

四、数据库访问层简介

本节将会主要针对数据访问层的功能及职责进行讲解，分析之前在业务逻辑层中的四种模式与数据访问层之间的关系。我们闲来看看数据访问层与业务逻辑层中的

四种模式之间的关系。



我们在本节中的讲解主要是以领域模型为例进行分析讲解，因为只有领域模型模式，我们才能将数据访问层抽离出来，分成单独的层，这样能够做到领域对象持久

化透明。接下来我们来看看数据访问层都需要提供什么要的功能及数据访问层本身的职责是什么。

数据库访问层是唯一知道如何操作存储介质的入口，可以这么来说，基于数据访问层之上，我们调用数据库访问层提供的方法，我们就能完成数据的存储与读取，

所以我们可以知道，数据访问层应该是与数据库直接是独立的。还有就是我们的数据访问层如何能实现不同类型的数据库的动态的切换，而我们不需要修改任何的程序

功能等，可能我们在开发的过程中都会遇到这样的问题。所以我们希望可以对数据访问层完成动态的配置，通过不同的配置项完成对象数据库访问的切换，这里我想大

家都是比较熟悉的，通过 XML 配置文件来完成数据库的切换，前面我们说了我们的需求，是必须实现无缝的数据库的切换，那么我们如何实现呢，这里我们可以通过定

义一个数据库访问接口，然后通过实现不同的数据库的细节，来实现这样的切换。目前很多流行的框架都是采用这样的方式来实现数据库的动态切换。当然有时候我们

的项目中肯能不让我们使用开源的通用框架，这时候我们可能就需要自己去实现这些数据访问层的具体细节。

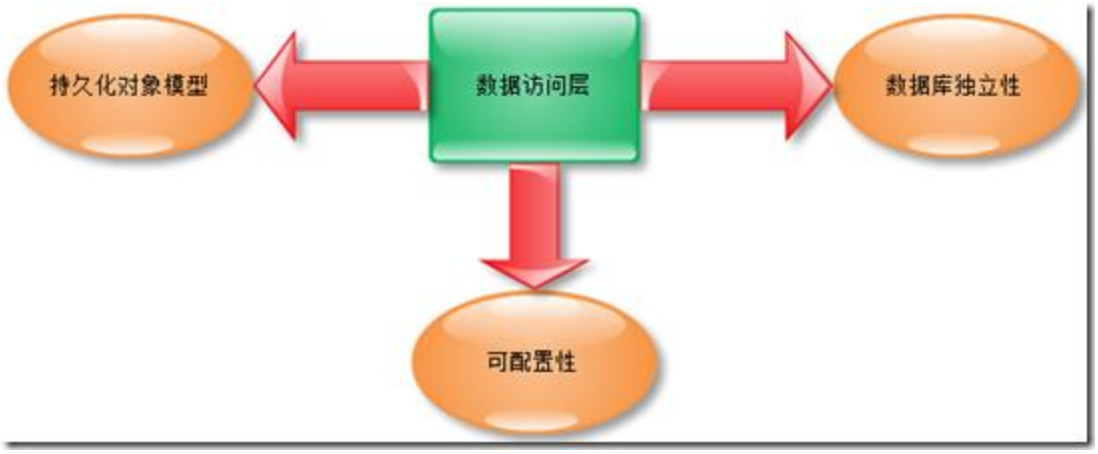
当然数据访问层都必须能够将应用程序中的数据持久化到存储介质中，通常我们使用的数据都是关系型的数据库，但是我们知道我们在程序的开发中，通常采用的

模型都是对象模型，那么如何实现对象模型与关系模型直接的互相的转换就显得非常的重要。当然这是数据访问层的重要功能。通常来说，业务逻辑层及服务层不了解

数据库访问的具体细节，他们都需要通过数据访问层来实现数据的交互。一般来说在领域模型中，数据访问通常都是在服务层中进行调用的，而业务逻辑层并不关注数

据持久化，所以我们前面说的持久化透明的方式也是由此方式来实现。

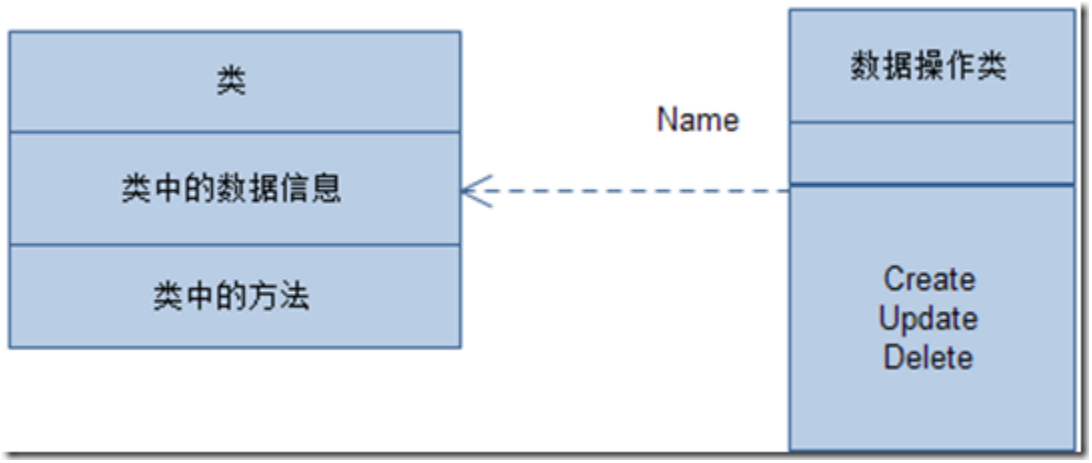
上面我们啰嗦了一大堆，基本上说了数据访问层的基本需求功能：



下面我们来看看数据访问层的几个基本职责

首先、数据访问层应该提供基本的持久化操作 CRUD 的操作，我们知道数据访问层是唯一能操作数据库的一层，因为我们在设计时需要注意，系统的其他层不能

包含操作数据库的相关功能

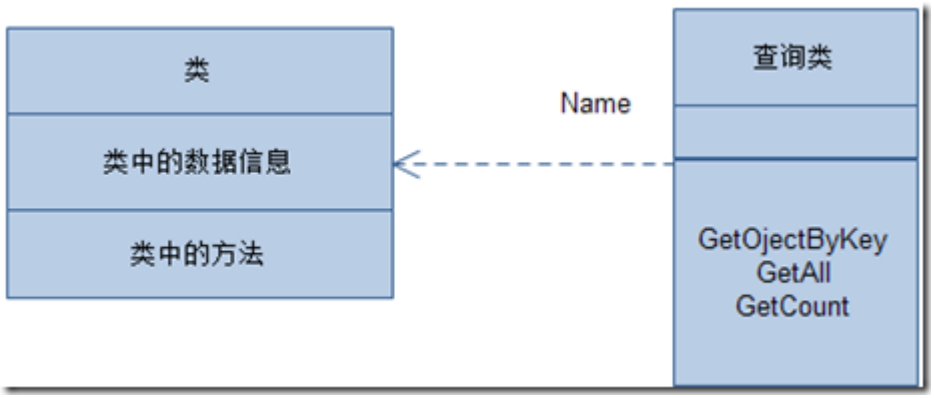


这里我们通过类图可以看到，通过提供与类对应的数据库操作类来提供

相应的持久化操作方法，当然这不是唯一方式，可行的方式有很多。我们后面会详细讨论。

其次、应该提供能够满足类中信息的读取操作，一般情况下来说我们经常使用的是，根据主键查询某个对象的信息，或者是查询所有的记录，或者是根据条件返

回满足的集合。



当然我们这里定义的查询类可能是通用的形式，通过泛型的形式来做。但是我们知道

领域模型中肯定会有引用对象的情况，那么对于这样的情况我们如何来做呢？我们一般是通过延迟加载的形式来处理这样的要求，我们后面会依次讲解。我们来看看上

面的通用形式的简单代码格式：

```
public class QueryHelper
{
    /// <summary>
    /// 根据主键返回对象
    /// </summary>
```

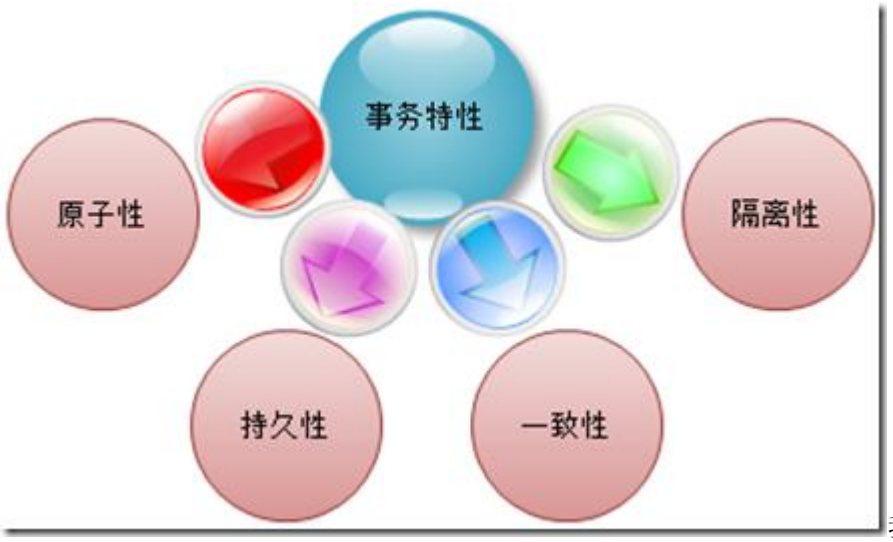
```
    /// <typeparam name="T"></typeparam>
    /// <param name="key"></param>
    /// <returns></returns>
    public T GetObjectByKey<T>(object key)
    {
        return default(T);
    }

    /// <summary>
    /// 获取指定类型对象的总记录数
    /// </summary>
    /// <typeparam name="T"></typeparam>
    /// <param name="key"></param>
    /// <returns></returns>
    public int GetCount<T>(object key)
    {
        return 0;
    }

   /// <summary>
   /// 返回指定类型的所有对象集合
   /// </summary>
   /// <typeparam name="T"></typeparam>
   /// <returns></returns>
    public List<T> GetAll<T>()
    {
        return new List<T>();
    }
}
```

再次、数据库访问必须提供事务的管理，可以说不提供事务操作的数据访问层就没有办法使用，因为这样的数据访问层构建的系统是不安全的。特别是批量持久

化的过程中，事务不但能够减少与数据库操作的次数，而且根据事务的四个特性可以提供更好的安全性。我们来回顾下事务的四个特性：



我想我这里就不用一一解释了，大家都明白的。我们在数据访问层的设计中是通过引入

“工作单元”来实现事务管理的，工作单元后面会讲述到工作单元内提供的方法及事务性。

最后、数据访问层必须提供处理并发的功能，我们在系统访问的人较多的情况时肯定会出现并发的情况，数据访问层如何处理这样的情况就显得极其重要了，在

一个多用户并发的系统中，通过前面提到的事务来处理，这时候可能就会出现数据库完整性的问题，例如这样的情况，一个用户现在在编辑自己的个人信息，例如将生

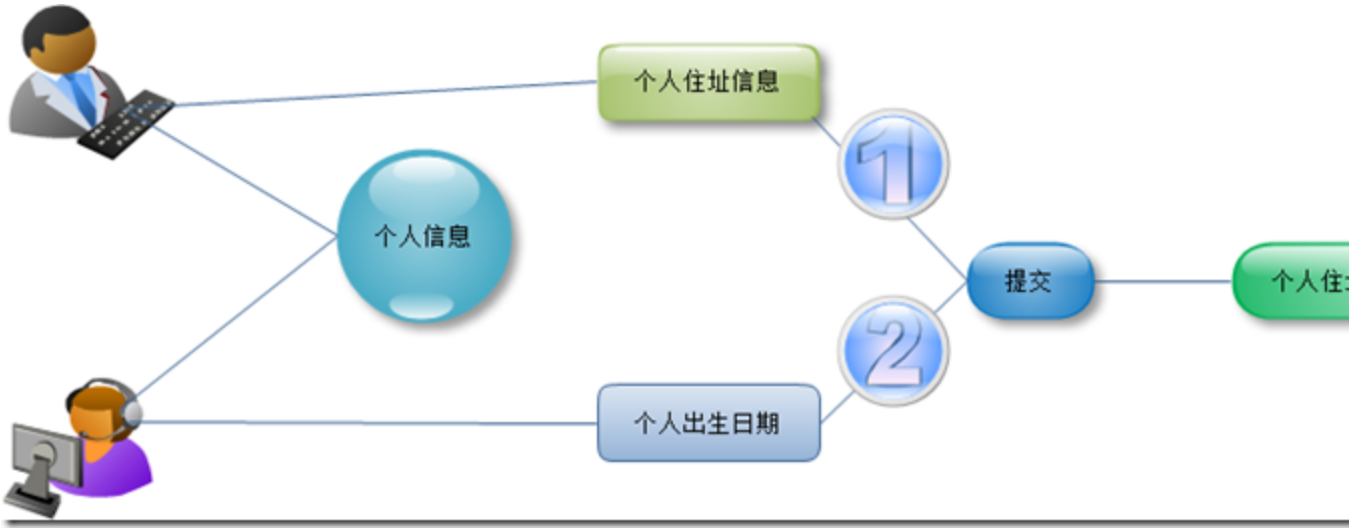
日修改为 1985 年 3 月 20 日，这个用户对应的 ID 是 298，这时候他只是修改了，但是还没有提交，此时管理员也修改了，比如说修改了 ID 为 298 的这个用户信息的地址或

者其他信息，并且提交，此时，用户将自己编辑的生日提交了，那么数据库中对应的 ID 为 298 的数据信息就会是最新修改的数据信息，那么之前管理员修改的数据信息

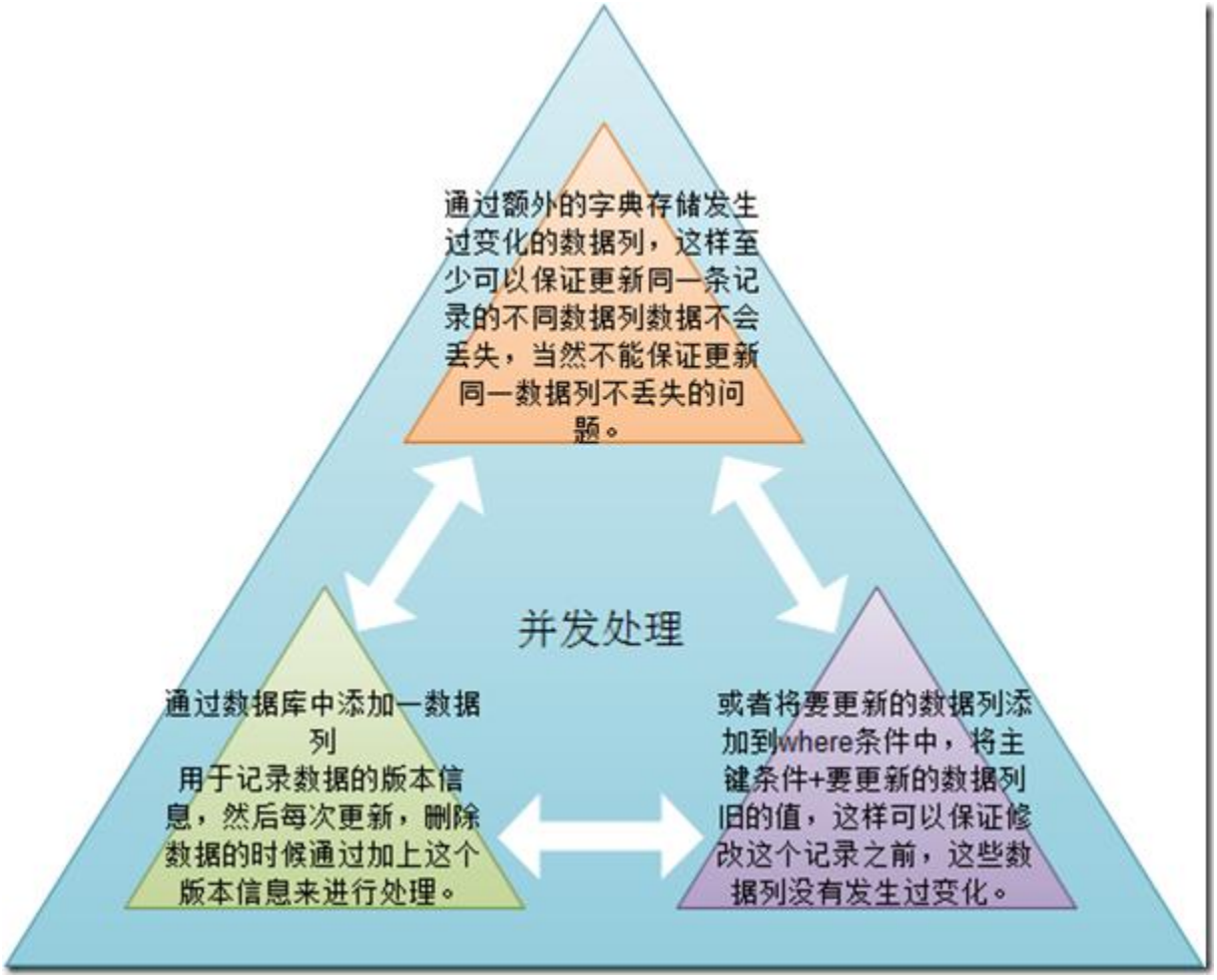
就会发生丢失，虽然是修改的可能字段不是同一个字段，这就和我们底层实现的数据访问层有关，当然如果说我们在数据访问层实现了，只更新修改过的数据列的值的

话，那么可能不会存在这样的情况，当然这就和我们底层实现的数据访问层的机制有关。

下面我们通过图形的方式来说明，更容易理解：



我们来看看可能的几种方案，可以对这样的并发做出相应的处理？



当然这里只是提供了几个简单可行的办法，当然如果大家还

有更好的办法，可以告诉我，不胜感激。

当然上面的四个基本职责是我们在数据访问层必须提供的，还应该提供缓存机制，延迟加载等等包括一些性能方面的优化的设计等，这些都在后面讲解吧。

下面我们来看看数据访问层与其他层直接的关系与交互，我们前面说过，在领域模型下，业务逻辑层中的数据的持久化都是通过服务层来完成的，下面我们来看

看各层之间的关系。我们先来看看服务层与数据访问层之间的关系。

服务层与数据访问层之间进行交互，服务层通过 DTO 与 UI 层进行交互，服务层通过组织业务逻辑层中的对象来实

现业务流，然后通过调用数据访问层将业务流中的相应数据进行持久化，通过数据访问层来初始化领域模型。置于直接在表现层中使用数据访问层的功能，我们通常是

不推荐这样做的，一般我们不会这么做的，我这里就不详细的阐述。

五、如何设计数据访问层

本节将详细的讲述如何设计出自己的数据访问层，满足上述的几个基本要求，那么可以说就算完成了基本的数据访问层的功能，其实如果我们从头开始开发一个

这样的数据访问层将是非常大的工作量，目前流行的很多的 ORM 框架已经提供了丰富的数据访问层的功能，能够非常好的满足上述的几项职责。当然本节还是会结合代

码来说说数据访问层的具体实现。

我们前面讲述了数据访问的 3 个基本的功能需求，数据库独立性，可配置性及持久化对象模式(对象模型与关系模型的转换)，我们这里先看如何实现数据库的独立

性，我们提出满足数据库的无缝迁移，通过 XML 配置文件，配置不同的数据库链接来实现这样的功能，那么首先我们需要定义针对不同数据库具体实现，才能完成后续

的操作，既然我们这里要降低耦合，那么根据我们前面的面向对象的设计原则与规范知道，我们推荐使用面向接口的编程的方式，来尽量的降低耦合性。我们来看看具

体的代码。首先我们定义一个通用的数据访问层的接口。

```
/// <summary>
/// 数据访问层统一接口
/// </summary>
public interface IDALInterface
{
    //CUD 持久化操作
    /// <summary>
    /// 创建新的对象
    /// </summary>
    /// <param name="model"></param>
    /// <returns></returns>
    int Create(object model);
    /// <summary>
    /// 更新对象
    /// </summary>
    /// <param name="model"></param>
    /// <returns></returns>
    int Update(object model);
    /// <summary>
    /// 删除对象
    /// </summary>
    /// <param name="model"></param>
    /// <returns></returns>
    int Delete(object model);
    //R 查询服务
    /// <summary>
    /// 查询所有记录
    /// </summary>
    /// <typeparam name="T">泛型模型</typeparam>
    /// <returns></returns>
```

```
        IList<T> GetAll<T>() where T : class,new();
        /// <summary>
        /// 查询满足条件的集合
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="whereCondition"></param>
        /// <returns></returns>
        IList<T> GetListByQuery<T>(WhereCondition whereCondition)
where T : class,new();
        /// <summary>
        /// 返回总行数
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <returns></returns>
        int GetCount<T>();
        /// <summary>
        /// 返回满足条件的总行数
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="whereCondition"></param>
        /// <returns></returns>
        int GetCount<T>(WhereCondition whereCondition);
        /// <summary>
        /// 根据主键返回对象模型
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="key"></param>
        /// <returns></returns>
        T GetModelByKey<T>(object key) where T : class,new();
        //事务
```

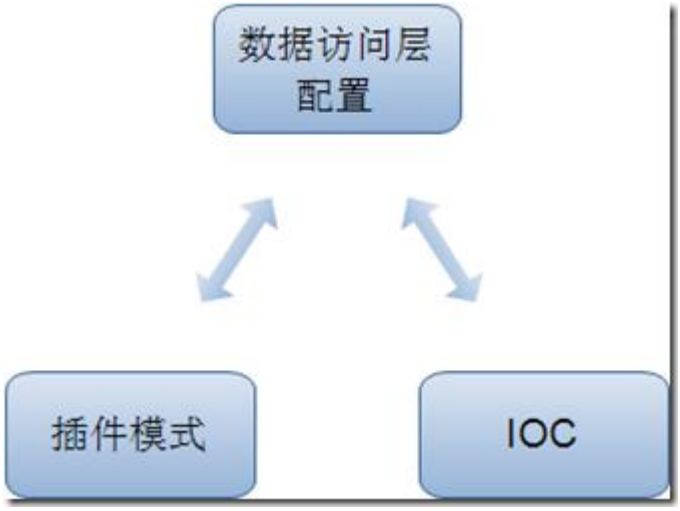
```
        /// <summary>
        /// 是否事务执行
        /// </summary>
        bool IsTransaction
        {
            get;
        }

        /// <summary>
        /// 开始事务
        /// </summary>
        void BeginTransaction();
        /// <summary>
        /// 提交事务
        /// </summary>
        void Commit();
        /// <summary>
        /// 回滚事务
        /// </summary>
        void Rollback();

    }
```

这里定义了基本的几个简单方法，当然其中并没有包括并发的处理，后面会讲到这块的处理方案的实现， 前面介绍了几种可行的实现方式。接口定义好了之后，

数据层的具体代码我这里就不一一的定义贴出来了，因为每种不同的数据库类型就要分别实现，我们这里讲解 2 中不同类型的实现思路吧，



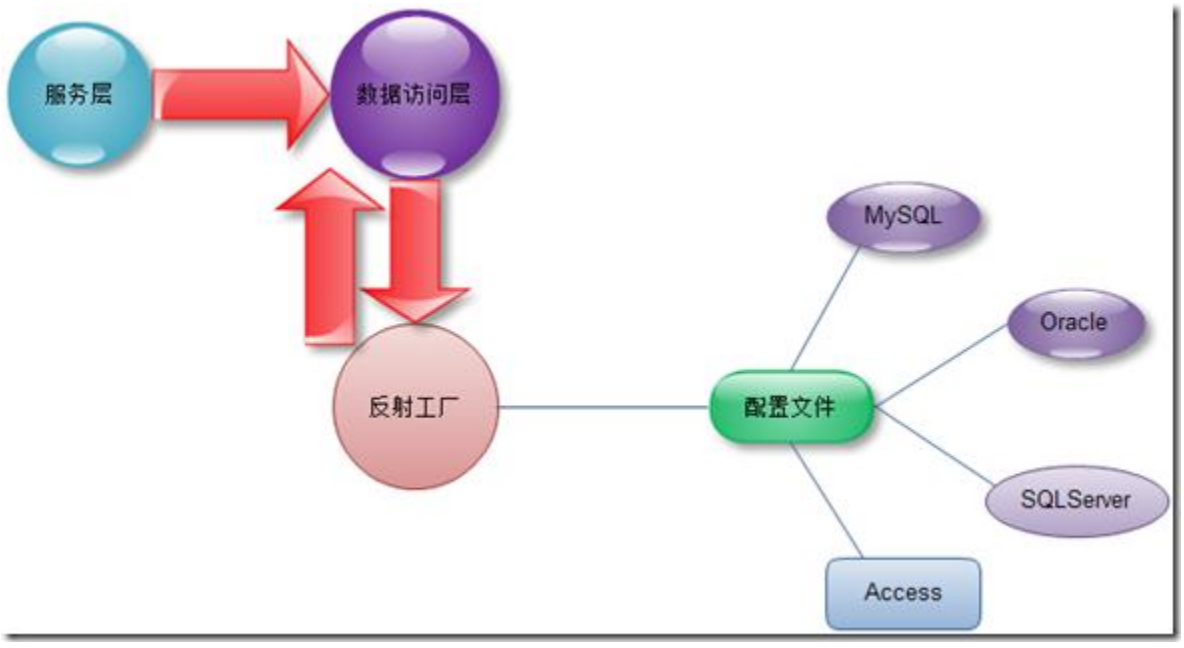
我们这里讲解 2 中实现动态创建具体数据访问组件的方式，我们先来讲讲插件模式。

插件模式

插件模式：插件模式就是通过外部配置文件中读取要创建的组件的类型信息，然后调用组件服务，插件模式的关键点就是服务不会与具体实现联系起来，在我们的

分层结构中的解释就是，服务层中调用数据访问层中的组件服务，服务层不关系具体的调用方式，服务层只关心服务。而具体的数据访问组件是通过配置文件来动态的

创建，当然这就需要使用.NET 中的反射的功能。我们来看个图形画的描述：



反射工厂通过读取配置文件中具体的数据配置项及数据访问的具

体服务组件类型，通过反射工厂来动态的创建，好了我们来看看实例代码及配置文件。

```
<?xml version="1.0" encoding="utf-8" ?>
<Connection>
  <ConnectionItem key="connectionString"
                  value="Data Source=.\SQLEXPRESS;Initial
Catalog=EasyStore;User ID=sa;Password=123456" />
  <DALType key="DALType"
                  value="DAL.SQLServer" />

<Assembly key="Assembly "
                  value="DAL.SQLServer" />
</Connection>
```

上面的配置文件中的 ConnectionItem 节点中配置了数据库访问的链接字符串，DALType 定义了数据访问层组件的类型。我们下来看看反射工厂的示例代码实现。

```
public class DALHelper
{
    private static IDALInterface instance;

    public static IDALInterface GetDAL()
    {
        string assambly = XmlHelper.getVlaue("Assembly");//这里应该是自定义的读取 XML 节点的方式
        string type = XmlHelper.getVlaue("DALType");

        Assembly asm = Assembly.Load(assambly);
        instance = (IDALInterface)asm.CreateInstance(type);

        return instance;
```

```
    }
}
```

我们接下来看看如何使用这个数据访问层去实现相应的持久化操作：

```
public class TestService
{
    private IDALInterface DAL;

    public TestService()
    {
        DAL = DALHelper.GetDAL();
    }

    public void Create(Test test)
    {
        //相应的判定操作

        //创建对象
        DAL.Create(test);
    }
}
```

这样就实现了在服务层对数据访问层的调用操作，这里是通过接口调用的方式来实现。我们再来看看控制反转的实现方式吧。

控制反转

控制反转我们在设计规范与原则中有过讲解，控制反正通过动态的将组件注入到引用该组件的对象中的形式，然后让引用该组件的对象使用组件的服务，DI 依赖注入可以看作是控制反转的一个应用实例，我们可以把控制反转看作是一个原则。

下面我们来看看我们如何通过控制反正的方式来实现数据访问层的平滑迁移。当然我们知道，肯定是通过动态注入的方式来实现，当然目前主流的也有很多的 IOC 动

态注入框架，下面我们将会借助一些框架来说明如何实现这样的功能。

本文将以 Enterprise Library5.0 为例进行讲解动态注入的形式。我们先来看看配置文件的设置


```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="unity" type="Microsoft.Practices.Unity.Configuration.UnityConfigurationSection,
        Microsoft.Practices.Unity.Configuration"/>
  </configSections>

  <unity>
    <containers>
      <container>
        <types>
          <type type="DAL.IDALInterface"
              mapTo="DAL.SQLServer" >
            </type>
        </types>
      </container>
    </containers>
  </unity>
</configuration>
```

我们来看看通过一个中间类去实现相应的注册代码：

```
/// <summary>
/// 用于动态完成代码注入的公共类
/// </summary>
public class IOCContainer
{
    private static IUnityContainer container;
    private UnityConfigurationSection section;
    public void InitIOC()
    {
        container = new UnityContainer();
```

```
            section =
(UnityConfigurationSection)System.Configuration.ConfigurationManager.
GetSection("unity");


            section.Configure(container);
        }


        public static IDALInterface GetDAL()
        {
            return container.Resolve<IDALInterface>();
        }
    }
```

通过上述代码我们实现了，动态的创建数据访问层组件实例，下面我们来看看依赖注入的方式去完成相应的持久化的功能。我们来看看服务层的代码

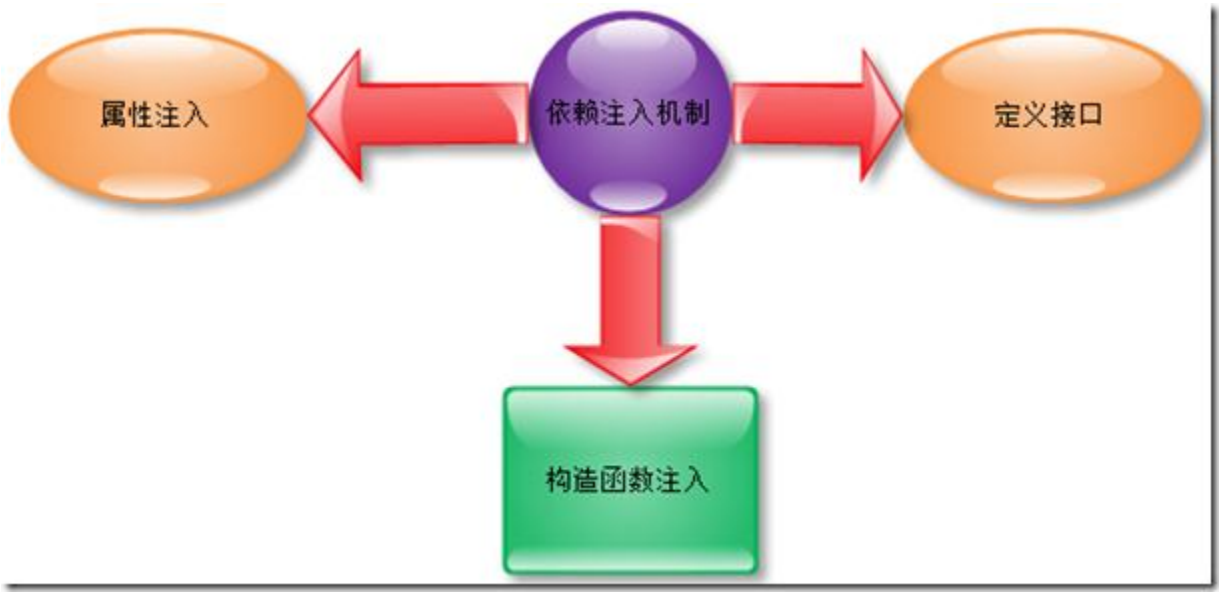
```
    /// <summary>
   /// 测试服务层
   /// </summary>
    public class TestService
   {
        private IDALInterface DAL;
        public TestService(IDALInterface dal)
        {
            DAL = dal;
        }


        public void Save(Test test)
        {
            DAL.Create(test);
        }
```

```
}
```

这里我们是采用构造函数注入的方式来实现数据访问层的动态注入的，当然除了构造函数注入，还有其他的方式，我们这里只是举例说明，一般来说依赖注入有

如下的几种形式



具体的我这里就不举例说明了，大家可以网上查查有很多的例

子，我们平时也常用这些方式。

六、实现数据访问的四项原则

在第四节中我们讲解了数据访问层的四个原则，那么我们在自己的数据访问层中如何实现这几个原则呢，我想针对第一个原则持久化的原则，我们前面只是简单

的讲解如何实现数据库的独立性，下面我们先来看看持久化的操作，也就是我们说的 CUD 的操作，并不包括具体的查询服务，查询服务也是我们数据访问层必须提供四

个原则之一，我们后面都会讲解，我们先来看看 CUD 的实现。我们在做持久化服务的时候，一般情况下，我们会定义一个统一的数据访问层接口，然后提供持久化服

务，事务等等，通常有一些数据访问层共性的部分，我们都通过一个抽象类来实现，抽象类将实现接口中的部分功能，然后通过定义一些抽象成员函数，让具体的数据

访问层去实现相应的功能。我们这里以上节我们定义的 IDALInterface 为例讲解基类的简单实现。

我们将原来的接口层进行相关的优化操作将 CUD 操作单独抽取出来，通过 ICUDMapper 接口来定义

```
/// <summary>
/// 数据库持久化访问器
/// </summary>
public interface ICUDMapper
{
    //CUD 持久化操作
    /// <summary>
    /// 创建新的对象
    /// </summary>
    /// <param name="model"></param>
    /// <returns></returns>
    int Create(object model);
    /// <summary>
    /// 更新对象
    /// </summary>
    /// <param name="model"></param>
    /// <returns></returns>
    int Update(object model);
    /// <summary>
    /// 删除对象
    /// </summary>
    /// <param name="model"></param>
    /// <returns></returns>
    int Delete(object model);
}
```

然后我们来看看基类接口层的简单实现，作为所有数据访问层的父类

```
public abstract class BaseDAL : IDALInterface,IDisposable
{
```

```
protected abstract ICUDMapper GetMapper();

//CUD 持久化操作
/// <summary>
/// 创建新的对象
/// </summary>
/// <param name="model"></param>
/// <returns></returns>
public int Create(object model)
{
    return GetMapper().Create(model);
}
/// <summary>
/// 更新对象
/// </summary>
/// <param name="model"></param>
/// <returns></returns>
public int Update(object model)
{
    return GetMapper().Update(model);
}
/// <summary>
/// 删除对象
/// </summary>
/// <param name="model"></param>
/// <returns></returns>
public int Delete(object model)
{
    return GetMapper().Delete(model);
}
```

```
        #region IDisposable 成员

        /// <summary>
        /// 是否数据库访问组件资源
        /// </summary>
        public void Dispose()
        {
        }

        #endregion
    }
```

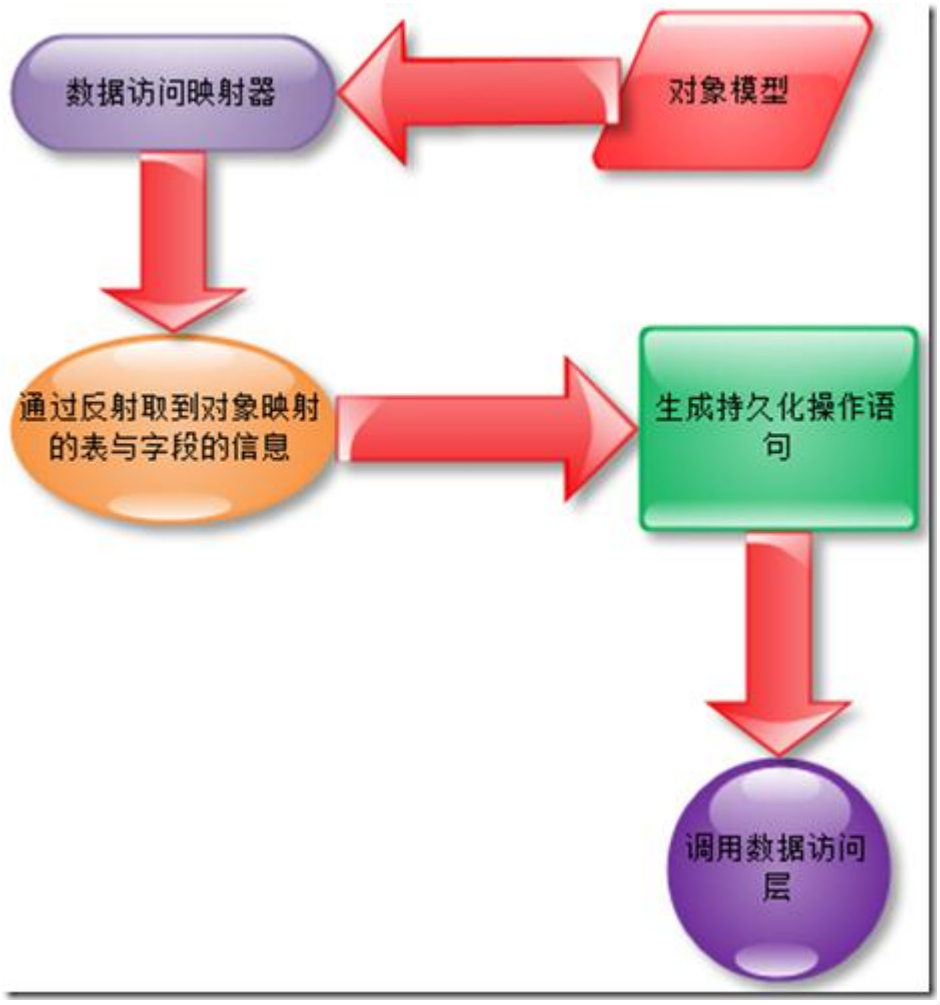
当然这里只贴出实例代码。当然我采用这样的方式，就是利用我们之前的一篇：Step by Step-构建自己的 ORM 系列-开篇 这篇中的反射的思想，大家可以看看

特性+反射的思路，我这里的数据持久化访问器也是类似的操作，可能底层的实现就是这样的方式。

具体的数据持久化访问器如何动态的生成 SQL 语句，缓存优化等各方面的内容，我们本篇可能不会深入的讲解，我还是想将这块放在 ORM 系类篇深入讲解。

当然我们其实可能极端的做饭就是为每个领域模型中的对象建立一个数据持久化映射器，完成映射，我这里则是通过创建数据库的统一模式，在具体的映射器中，

通过反射取得数据对象的映射信息。我们来看看实现的思路吧，具体代码我就不贴了



大体的流程就是上面说的了，细节肯定还有很多要注意的地方。

下面我们来看看查询服务的实现：

我想一般的系统 80%的时间数据库执行的操作是查询，而 20%的时间在完成写入和修改的操作，当然我这里不是绝对的说法。我们希望有一个工具帮我们自动完

成基本的查询服务，而不是我们手动的去书写，因为我们发现对大部分的数据集合而言，有一些共性的操作，例如获取某个主键值的对象的信息，或者是获取数据库表

中的总行数，或者是返回数据库表的所有记录，并且如何将关系数据库中的关系模型转换为对象模型，这都是查询服务中应该提供的基本功能。下面我们来看看简单实

现吧。

我想我们还是参考前面的方式，我们将 IDALInterface 层中的查询服务进行抽象分离，将查询服务单独提出来放在接口 IQuery 中。代码如下：

```
public interface IQuery
{
```

```
        /// <summary>
        /// 查询所有记录
        /// </summary>
        /// <typeparam name="T">泛型模型</typeparam>
        /// <returns></returns>
        IList<T> GetAll<T>() where T : class,new();
        /// <summary>
        /// 查询满足条件的集合
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="whereCondition"></param>
        /// <returns></returns>
        IList<T> GetListByQuery<T>(WhereCondition whereCondition) where
T : class,new();
        /// <summary>
        /// 返回总行数
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <returns></returns>
        int GetCount<T>();
        /// <summary>
        /// 返回满足条件的总行数
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="whereCondition"></param>
        /// <returns></returns>
        int GetCount<T>(WhereCondition whereCondition);
        /// <summary>
        /// 根据主键返回对象模型
        /// </summary>
```

```
    /// <typeparam name="T"></typeparam>
    /// <param name="key"></param>
    /// <returns></returns>
    T GetModelByKey<T>(object key) where T : class,new();
}
```

我们来看看在基类中的实现。查询服务的相关实现

```
    /// <summary>
    /// 查询服务组件
    /// </summary>
    /// <returns></returns>
    protected abstract IQuery GetQuery();

     #region IQuery 成员

     /// <summary>
     /// 查询所有记录
     /// </summary>
     /// <typeparam name="T">泛型模型</typeparam>
     /// <returns></returns>
     public IList<T> GetAll<T>() where T : class,new()
     {
         return GetQuery().GetAll<T>();
     }
     /// <summary>
     /// 查询满足条件的集合
     /// </summary>
     /// <typeparam name="T"></typeparam>
     /// <param name="whereCondition"></param>
     /// <returns></returns>
```

```
        public IList<T> GetListByQuery<T>(WhereCondition
whereCondition) where T : class,new()
        {
            return GetQuery().GetAll<T>();
        }
        /// <summary>
        /// 返回总行数
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <returns></returns>
        public int GetCount<T>()
        {
            return GetQuery().GetCount<T>();
        }
        /// <summary>
        /// 返回满足条件的总行数
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="whereCondition"></param>
        /// <returns></returns>
        public int GetCount<T>(WhereCondition whereCondition)
        {
            return GetQuery().GetCount<T>(whereCondition);
        }
        /// <summary>
        /// 根据主键返回对象模型
        /// </summary>
        /// <typeparam name="T"></typeparam>
        /// <param name="key"></param>
        /// <returns></returns>
```

```
        public T GetModelByKey<T>(object key) where T : class,new()
        {
            return GetQuery().GetModelByKey<T>(key);
        }

        #endregion
```

当然根据不同的数据库可能定义的查询语句的格式不同，但是返回的结果的形式却可以定义成通用的形式。这样我们就可以实现比较通用的查询服务，也有很好的通用型和扩展性。当然我们这里还可以添加分页的支持等，只是添加的条件有限制，实现方式还是相同。

下面我们来看看数据访问层功能必须职责之事务性，我们都知道事务性的几大特性，通过事务性来提供数据的安全性。我们这里给出一种思路去实现这样的事务性，我们在数据访问层中定义一组事务单元，通过一个列表维护这些事务单元，当执行提交时，我们将这个事务范围内的所有事务单元进行提交，否则不进行真正的提交操作。我们来看看吧，我们在之前的 IDALInterface 中已经定义了事务相关的几个方法，我们这里同样抽出来，进行分解，抽出来一个单独的接口 ITransation，具体代码如下：

```
public interface ITransaction
{
    /// <summary>
    /// 是否事务执行
    /// </summary>
    bool IsTransaction
    {
        get;
    }

    /// <summary>
    /// 开始事务
    /// </summary>
```

```
    void BeginTransaction();
    /// <summary>
    /// 提交事务
    /// </summary>
    void Commit();
    /// <summary>
    /// 回滚事务
    /// </summary>
    void Rollback();
}
```

基类中的代码如下：

```
    #region ITransaction

    /// <summary>
    /// 是否事务执行
    /// </summary>
    public bool IsTransaction
    {
        get
        {
            return GetTransaction().IsTransaction;
        }
    }

    /// <summary>
    /// 开始事务
    /// </summary>
    public void BeginTransaction()
```

```
{
    GetTransaction().BeginTransaction();
}
/// <summary>
/// 提交事务
/// </summary>
public void Commit()
{
    GetTransaction().Commit();
}
/// <summary>
/// 回滚事务
/// </summary>
public void Rollback()
{
    GetTransaction().Rollback();
}

/// <summary>
/// 返回事务单元列表
/// </summary>
List<TransationUnit> list
{
    get;
}

/// <summary>
/// 执行事务单元的操作，执行数据操作并提交
/// </summary>
/// <param name="unit"></param>
```

```
        void Excute(TransationUnit unit);

        #endregion

        /// <summary>
        /// 事务组件服务
        /// </summary>
        /// <returns></returns>
        protected abstract ITransaction GetTransaction();
```

事务组件中添加了特殊的事务单元，用来存储事务执行的操作 CUD，还有就是要事务执行的数据对象，当然事务对象中的 CRD 操作就是使用前面讲解的 CRD 操作的方式，我们来看看吧，我们来看看事务单元的形式。

```
/// <summary>
/// 事务单元
/// </summary>
public class TransationUnit
{
    /// <summary>
    /// CUD 枚举
    /// </summary>
    public enum CUDEnum
    {
        Create,
        Update,
        Delete
    }

    private CUDEnum _cudType;
```

```
        private object _model;


        public TransationUnit(object model, CUDEnum cudType)
        {
            _model = model;
            _cudType = cudType;
        }
    }
```

我们在事务处理中，我们只执行事务列表中的操作，置于非事务列表中的单元我们将不做任何处理，所以我们只要是事务执行的事务单元，我们必须将指定操作类

型，当然我们还可以更灵活，我们通过在事务单元中设置属性判定是否在事务中，如果不在事务中，我们执行数据持久化操作，如果在事务中，我们则添加到事务列表

中，因为我们在提交时总会循环执行事务列表中的事务单元。当然我这里只是一个简单的思路，抛砖引玉，希望大家有更好的想法，可以跟我交流，或者给我提出建

议，那样我就感激不尽了。下面我们来看看数据访问层中的并发的问题，我们如何去应对，有没有什么完美的方案呢？

前面我们举例子说过，有 3 中方式，我们这里举例说明：

通过字典存储值发生变化的列，完成更新变化列的操作。这样至少可以避免 2 人同时更新，造成最近更新的内容覆盖先前更新的内容，虽然可能更新的是不同的数

据列，我们这里只提供实例代码：

```
namespace DAL
{
    /// <summary>
    /// 实体
    /// </summary>
    public class Entity
    {
        IDictionary<object, object> updateColumn = new Dictionary<object, object>();

        //通过定义一个代理类对象来保存实体对象的旧值副本，然后通过在服务层对实体对象数据的对比，比较变化前后的值，或者
对象赋值时，判定旧值与新值是否相等，然后保存在要更新的列的字典中。
    }
}
```

另外 2 种实现方式也差不多，一种是在数据库表中添加一列版本号，然后在数据更新时必须加上之前取出来的对象的版本号，然后一并同主键作为条件，完成更

新，这样至少可以保证数据的完整性。另外一直就是通过对象代理，我们将要更新的对象的字段的旧值，放在代理中，然后在生成更新语句时，我们将对象代理中的旧

值也添加到 where 查询条件中，这样我们通过限制条件的方式，来维护数据的完整性。我们来看看简单代码：

```
public class EntityProxy
{
    public string Name
    {
        get;
        set;
    }
    public object Tag
    {
        get;
        set;
    }
    public int ID
    {
        get;
        set;
    }
    public string Unit
    {
        get;
        set;
    }
}
```

假设上面的几个数据列是我们要更新的数据库列,那么我们只需要在 UPdate 语句中，附加这四个数据库列字段条件就好了，这样如果发现更新结果返回的是 0，那

么代表更新失败。通过上面的几种形式，我们来看，如果允许的话，我们还是推荐在数据库表中添加一个版本号的形式来处理更新，这是最高效的形式，可以通过时间

戳的形式来生成版本号。

七、本章总结

本章主要简单阐述了，数据访问层的基本功能，必要的四个职责，及数据库访问层的简单的设计思路与实现思路，当然我这里没有提供完整的实现，具体的实

现，我想我会在 ORM 系列中提供完整的代码，当然目前可能必须讲完架构后，回头我会拾起来那部分进行详细的讲解，好了我想本文的内容都比较浅显易懂，大家都能

够迅速的掌握，如果您有更好的思路或者设计方案，那么很希望您能提出来交流，这将是我莫大的荣幸。

八、系列进度

前篇

1、系统架构师-基础到企业应用架构系列之--开卷有益

2、系统架构师-基础到企业应用架构-系统建模[上篇]

3、系统架构师-基础到企业应用架构-系统建模[中篇](上)

4、系统架构师-基础到企业应用架构-系统建模[中篇](下)

5、系统架构师-基础到企业应用架构-系统建模[下篇]

6、系统架构师-基础到企业应用架构-系统设计规范与原则[上篇]

7、系统架构师-基础到企业应用架构-系统设计规范与原则[下篇]

8、系统架构师-基础到企业应用架构-设计模式[上篇]

9、系统架构师-基础到企业应用架构-设计模式[中篇]

10、系统架构师-基础到企业应用架构-设计模式[下篇]

中篇

11、系统架构师-基础到企业应用架构-企业应用架构

12、系统架构师-基础到企业应用架构-分层[上篇]

13、系统架构师-基础到企业应用架构-分层[中篇]

14、系统架构师-基础到企业应用架构-分层[下篇]

15、系统架构师-基础到企业应用架构-表现层

16、系统架构师-基础到企业应用架构-服务层

17、系统架构师-基础到企业应用架构-业务逻辑层

18、系统架构师-基础到企业应用架构-数据访问层

19、系统架构师-基础到企业应用架构-组件服务

20、系统架构师-基础到企业应用架构-安全机制

后篇

21、单机应用、客户端/服务器、多服务、企业数据总线全解析

22、系统架构师-基础到企业应用架构-单机应用(实例及 demo)

23、系统架构师-基础到企业应用架构-客户端/服务器(实例及 demo)

24、系统架构师-基础到企业应用架构-多服务(实例及 demo)

25、系统架构师-基础到企业应用架构-企业数据总线(实例及 demo)

26、系统架构师-基础到企业应用架构-性能优化(架构瓶颈)

27、系统架构师-基础到企业应用架构-完整的架构方案实例[上篇]

28、系统架构师-基础到企业应用架构-完整的架构方案实例[中篇]

29、系统架构师-基础到企业应用架构-完整的架构方案实例[下篇]

30、系统架构师-基础到企业应用架构-总结及后续

九、下篇预告

下一篇我们将会开始讲解系统架构中的最后一个分层-表现层的相关介绍及设计方案，这些都是本人在工作中的经验总结，由于本人能力有限，不足之处，还请大家多多指出。希望大家持续关注！