# 分层架构

## 架构

架构一般是针对整个系统的，并非针对某个单独的问题（单独的问题可以用模式来解决）

针对整个系统的“一张蓝图”，对系统的抽象。构架与具体的语言平台无关。

## 模式

软件开发中遇到的一些特定问题，前人总结出来的特定的经验、解决方法(复制某企业的成功模式。)

## 框架

架构设计、模式应用的经验积累的具体代码实现，方便以后的复用。Asp.net mvc、NHibernate、NSpring、…

## 分层

分为逻辑分层与物理分层。

### 逻辑分层（N-Layer）

* 逻辑上将系统中的不同功能模块、不同子系统等进行分层
* 好的逻辑分层可以让后续选择物理架构更灵活，选择性更大
* ”三层架构”属于逻辑分层（three-layer）

### 物理分层（N-Tier）

物理部署时将系统的不同模块部署在不同的服务器上

# 三层架构

三层是个比较好的软件架构：

* 表现层（UI）

用来搜集数据与展示数据。

* 业务逻辑层(BLL)

业务处理和业务计算：例如返回订单信息让表示层展示，会有个必要的数据——订单总额。这个数据需要首先用各个订单项的单价乘以数量，然后加和。那么，这个数据应该在业务层完成计算直接返回，总之不应让表示层进行任何业务处理和计算操作

数据合法性及完整性验证：与业务关系紧密的验证，最好放在业务层。甚至有些验证只能在业务层验证，如“当前用户名不能与已有用户名重复”，这种验证需要访问持久化数据，需要由业务层完成

* 数据访问层（DAL）

## （补充）什么是业务逻辑

记得几个月前，在一次北京博客园俱乐部的活动上，最后一个环节是话题自由讨论。就是提几个话题，然后大家各自加入感兴趣的话题小组，进行自由讨论。当时金色海洋同学提出了一个话题——“什么是业务逻辑”。当时我和大家讨论ASP.NET MVC的相关话题去了，就没能加入“业务逻辑”组的讨论，比较遗憾。

其实，一段时间内，我脑子里对“业务逻辑”的概念也是非常模糊的。但在不断地阅读、思考和实践过程中，这个概念及其相关的内容才在我脑子里渐渐清晰。我想，很多朋友也许也对这个概念不是很了解，所以愿意结合既有资料和自己的思考，总结一篇关于业务逻辑的概述性文章，一则与朋友们分享探讨，二则也是为自己对业务逻辑的学习做一个总结和提升。因为我还不敢说对业务逻辑内涵及外延理解的非常充分，所以文中如有不当之处，还请各位不用客气，尽管批评就好！

**内容提要**

**===================前篇=====================**

前言

内容提要

1、我把业务逻辑丢了！——找回丢失的业务逻辑

2、细说业务逻辑

  2.1、业务逻辑到底是什么

  2.2、业务逻辑的组成结构

   2.2.1、领域实体（Domain Entity）

   2.2.2、业务规则（Business Rules）

   2.2.3、完整性约束（Validation）

   2.2.4、业务流程及工作流（Business Processes and Workflows）

  2.3、业务逻辑层职责相关争议

   2.3.1、争议一：数据的格式化

   2.3.2、争议二：数据合法性及完整性验证

   2.3.3、争议三：CRUD

   2.3.4、争议四：存储过程

**===================后篇=====================**

3、业务逻辑的架构模式及实现

  3.1、Transcaton Script

   3.1.1、概述

   3.1.2、分析

  3.2、Table Module

   3.2.1、概述

   3.2.2、分析

  3.3、Active Record

   3.3.1、概述

   3.3.2、分析

  3.4、Domain Model

   3.4.1、概述

   3.4.2、分析

  3.5、各种架构模式的比较及选择

4、结束语

参考文献

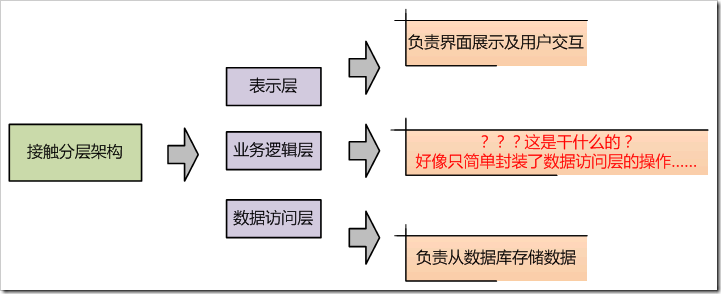
**1、我把业务逻辑丢了！——找回丢失的业务逻辑**

相信朋友们基本都是软件开发人员。不论身处什么职位，我们的工作都有一个共同的目标——制作软件产品。而所谓的软件产品，一定是在某个领域内去实现某些业务 。如此看来，“业务逻辑”本应和“软件产品”是紧紧绑在一起的，没有业务逻辑，何来软件产品？

但是，我发现一个奇怪的现象，一说业务逻辑，很多人就无法形成清晰地印象。例如，经典的三层架构：表示层、业务逻辑层和数据访问层，一提到表示层或数据访问层，大家脑子里马上能产生出清晰的概念，但一提到业务逻辑层，就有点模糊了，或者完全不知道其是什么，或者有个模糊的轮廓，但对其具体的职责、结构不是很清楚。真是奇了怪了！我们天天和业务逻辑打交道，搞不清业务逻辑是什么。

对于这个奇怪的现象，我思前想后，结合自身的教训（我也曾很长时间搞不清业务逻辑），终于弄清楚了其原因——这和我们接触这个概念的途径和认知结构有莫大关系。

不知道有多少人和我一样，首次接触“业务逻辑”这个概念是从分层架构中的“业务逻辑层”概念开始的，我相信不在少数。事情坏就坏在这里！为了让朋友们直观看清“业务逻辑”的概念是怎么被我们丢掉的，请大家看一个图，这个图展示了很多人对“业务逻辑”的认知过程。



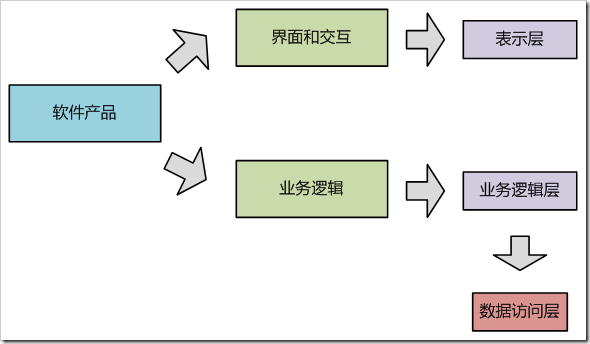
**图 1-1、狭义的认知分解过程**

如图1-1所示，我们先接触了分层架构，然后对每个层产生了初步的认识。其中，由于表示层和数据访问层的代码职责清晰明确，基本能正确认识。但是，由于我们接触的分层架构的Demo大多业务极其简单，又基本是CRUD操作集中型的业务。所以，我们脑子中就产生了疑问：这个所谓的业务逻辑层是干什么的？怎么就简单封装了一下数据访问层的操作？这有存在的必要吗？由于有了这种“先入为主”的误导，使得很多朋友脑中将“业务逻辑”和“业务逻辑层”两个概念混淆了，始终想不明白这东西到底是什么，做什么用的。再加上很多朋友所看的、所做的系统都是CRUD操作集中型的，就形成了“业务逻辑貌似就是对数据访问操作的简单封装”这一片面概念。

到底这一概念有没有错呢？其实没错，因为在简单的、CRUD操作集中型软件中，业务逻辑基本就是对数据访问简单的封装。但是，无错不代表全面，这是一种狭义的业务逻辑理解，而且是狭义中的狭义。为什么这么说呢？因为我们不但是在“业务逻辑层”这么一个狭义范围内去理解业务逻辑，而且还是CRUD集中型操作这种“非常瘦”的业务逻辑层范围内去理解，所以，可谓是在狭义的基础上的狭义。

当我们把这么一个“狭义中的狭义业务逻辑”与“业务逻辑”等同起来时，误会、迷茫、困惑、不屑就出现了。这就如同，给你一只温顺的哈巴狗，还是病怏怏的、无精打采的小哈巴狗，而你把这只“病怏怏的小哈巴狗”与“狗”的概念等同起来了。那么你一定就会为有人养狗看家和警察养狗当警犬抓坏人而困惑：这东西这么弱小，我一脚就踩死了，怎么弄用来看家和抓坏人呢？进而可能会产生“狗狗无用论”，“狗狗废品”等观念。当然，在现实中，很少有人只见过小哈巴狗而没见过狼狗等其它狗类，所以，故事中的误会对 “狗”一般是不存在的。但在现实中，确实有很多人只见过业务逻辑中的“小哈巴狗”，却没有见过业务逻辑中的“狼狗”、“藏獒”，所以，这种误会在对“业务逻辑”的理解上广泛存在。

那么，广义的情况究竟是怎么样的？请看下图。



**图 1-2、广义的认知分解过程**

（注意！凡是不特别说明，下文中所有“数据”一词都指需要持久化的数据，而不包括内存中的临时数据。请各位留心。）

如图1-2所示，广义的认知分解应该是这样的：软件产品都是在某个领域内实现某些特定业务，所以，软件产品天生应该分解为界面交互部分和业务逻辑部分，其中业务逻辑部分是软件产品的核心，它客观存在于软件产品内部，但是无法对使用者产生直观刺激，因此业务逻辑不能与使用者直接交互。而界面交互部分是业务逻辑与使用者进行交流的接口，使用者通过界面交互部分，与业务进行交流，从而使得软件产品发挥其作用。

而在具体实现系统时，界面交互部分演化成表示层，业务逻辑部分演化成业务逻辑层。所以，可以认为，数据访问层不是软件产品自然演化的直接产物，之所以出现数据访问层，是因为某些产品的业务属于“数据操作集中型”业务，为了实现隔离、复用等目的，架构师从业务逻辑中分离出了频繁使用的数据访问业务，形成了单独的数据访问层。从广义来说，可以认为数据访问隶属于业务逻辑，因为，数据访问操作实际上也是业务逻辑的一部分。

总结一下几个要点：（这几个要中的业务逻辑均指广义业务逻辑）

1）软件产品自然的可分为界面交互部分和业务逻辑部分。

2）从空间结构上看，业务逻辑和数据访问不是并列关系，而是隶属关系——数据访问隶属于业务逻辑。虽然在具体系统实现层面，数据访问层和业务逻辑层是并列存在，但从概念本质层面上分析，两者是隶属关系。

3）从时间结构上看，应该是先有业务逻辑的概念，才有数据访问的概念。业务逻辑衍生自软件本身，数据访问衍生自业务逻辑。

4）因为业务逻辑是软件产品自然的一部分，所以拥有业务逻辑是软件产品的必要条件（读者可以试着举出一个不包含业务逻辑的软件）。但是一个软件可以没有数据访问，如“计算器”、“不带存档的小游戏”等。

利用以上论述要点和认知分解，朋友们可以试试在脑中重新构筑狭义和广义“业务逻辑”的概念。看能不能把我们丢掉的业务逻辑概念找回来。关于业务逻辑更多的细节，将在下文中讨论。

**2、细说业务逻辑**

**2.1、业务逻辑到底是什么**

在第一大节里说了那么多，相信各位基本已经形成“业务逻辑”的概念了。如果我在这里再啰嗦什么，我不嫌累各位也要嫌烦了。所以，这里我仅给出两个定义。

广义上的义务逻辑——软件本身固有的一种品性，自然存在于软件产品内部，是软件具有的在某个业务领域内的逻辑，是软件的核心和灵魂。软件产品除界面和交互外的一切都可看作是广义业务逻辑。

狭义上的业务逻辑——等同于分层架构中“业务逻辑层”的职责，是软件中处理与业务相关任务的部分，一般狭义上的业务逻辑不包含数据持久化，而只关注领域内的相关业务。

对于以上两种定义，希望朋友们不要割裂开来看，而 要辩证统一的去看，这样，才能构建一个完整而辩证统一的“业务逻辑”概念。在下文中，将不再明确区分狭义和广义，“业务逻辑”一词将代表两者的辩证统一体。

**2.2、业务逻辑的组成结构**

业务逻辑作为一个高层次概念，其内在结构也是非常丰富的，下面我们深入其里，去探寻一下业务逻辑都是由哪些更底层的部分构成的。

**2.2.1、领域实体（Domain Entity）**

通俗的说，领域实体就是这个领域内有哪些东西。例如，银行业领域内有账户、支票、前台营业员等实体；B2C电子商务领域有商品、订单、交易等实体；魔兽世界游戏的领域内有角色、种族、道具、魔法等实体；高等代数领域有矩阵、行列式等实体。

领域实体是某个领域内各种对象的抽象，可以用名词表示（可以是具体名词或抽象名词，甚至动名词，只要其具有名词性），构成了整个业务逻辑的骨骼和静态模型。一般每个领域实体有自己的一些属性和行为。 顺便说一句，领域实体的存在时OOA&D的基础。

在具体的软件系统中，领域实体往往会根据架构的不同有不同的映射存在形式。

其中一种叫做Business Object（BO），即业务对象，某些文献称其为“充血实体类”，这种对象完整抽象了领域内的某个实体，封装了此实体相关属性和行为。在面向对象的设计和架构中，这种实体类很常见。

另一种叫做Data Transfer Object（DTO），某些文献称其为“贫血实体类”，其特点是仅有属性，不存在行为。这种实体类主要负责整体性传递数据。另外，与BO不同的是，DTO可以不抽象领域实体的全部属性，而只根据需要抽象一部分。例如，某个“User”实体存在很多属性，但如果某个方法仅需要其联系方式，可以设计一个DTO，仅有id，email，address，phone等就够了。在面向过程的设计和架构中，这种实体设计比较常见。

**2.2.2、业务规则（Business Rules）**

业务规则就是某个领域内运作的规则，构成了整个业务逻辑的灵魂和动态模型。业务规则作用于领域实体，领域实体遵从业务规则进行运作。

如：在银行领域内，“转账时从A账户扣除相应款项，在B账户添加相应款项，并从A账户扣除相应手续费，并通过某些途径通知A和B账户的户主”就是一条规则。需要注意的是，业务规则比较抽象，并不是需求，需求需要具体且无二义性，而业务规则只是抽象的一种描述，例如，通知户主的途径是什么？电子邮件？电话？短信？并没有具体描述，但在规则中有“通知”这一项，因此不能将业务规则等同于需求。

**2.2.3、完整性约束（Validation）**

领域实体和业务规则构建了业务逻辑的主体，但在这主体之上，还存在着一个限制，这就是完整性约束。

完整性约束是对业务领域中的数据、规则的强制性规定与约束。这种约束是系统正常运转的保证。

如“账户密码不能为空”，“身份证号必须符合具体格式规定”，“转账流程必须具有原子性，A账户扣钱、B账户存钱、A账户扣除手续费、通知户主四项操作必须要么都做，要么都不做”，都是完整性约束。

**2.2.4、业务流程及工作流（Business Processes and Workflows）**

有了上述三项，业务逻辑还不能正常工作，因为还没有“启动器”和“过程托管器”。设想我们有了各种实体类，它们有各自的属性和行为，也有定义好的业务规则和完整性约束。现在实体类仅仅具有实现业务规则的能力，但它们如何启动并交互协调完成业务规则呢？因此我们需要有东西去触发和协调实体。

业务流程或工作流是启动及托管协调领域实体完成既定规则的过程。例如，“在线订购”是一个业务流程，它包括“用户登录-选择商品-结算-下订单-付款-确认收货”这一系列流程。各个实体如会员、订单、商品等已经包含了完成在线订购必要的行为，但仍需一个流程，才能真正完成业务。

具体到程序中，业务流程也许通过一个方法来实现，这个方法负责启动并协调各个实体类，完成一个流程。

**2.3、业务逻辑层职责及相关争议**

**2.3.1、数据的格式化**

关于数据的格式化应该放在业务层进行还是表示层进行一直存在争议。我个人的意见是这样的：

业务层送给表示层的数据应该具备以下要求。1）返回的数据应该完成了所有必要的业务处理和业务计算。 例如，若返回订单信息让表示层展示，会有个必要的数据——订单总额。这个数据需要首先用各个订单项的单价乘以数量，然后加和。那么，这个数据应该在业务层完成计算直接返回，总之不应让表示层进行任何业务处理和计算操作。2）一次性返回所有需要的数据，避免表示层再一个Action里调用多次业务。 打个比方，例如订单中有个“客户姓名”，这个数据不保存在订单表中，而是通过外键关联的，那么，业务层应该将“客户姓名”一并取出返回给表示层。总之，避免表示层在一个Action里多次调用业务层。3）不携带任何格式信息，仅仅是结构良好的纯净数据，如DTO形式。 因为，数据如何展示，是表示层的职责，如何在业务层中返回了过多格式信息，就会造成表示层的修改困难。例如，我曾听说过所里承接的一个实际项目，开始是使用B/S，当时他们的业务层返回的数据全都附带了html代码。后来，客户嫌B/S响应不够迅速（可能是客户公司的网络条件不好），要求改成C/S，当时全傻眼了，貌似几乎修改了整个业务层。那个项目相当庞大，7个子系统，投入200人开发了1年多，想想修改的难度吧。

**2.3.2、数据合法性及完整性验证**

一般做系统，都避免不了数据验证。上文曾经提到，完整性约束是业务逻辑的一部分。如此看来，数据验证一般应该放在业务层。但是，实际情况并不尽然。个人认为数据验证的方式，目前没有统一标准，可以根据需要放在表示层或业务层。但是，我个人不提倡在“表示层的服务端”放置过多完整性验证。因为，表示层的职责应该仅仅是接收数据并传递给业务层，不应对数据是否合法负责。过多的数据验证，不但令表示层代码臃肿，而且使得表示层职责变得不明确。

可以在“表示层的服务端”放置一些简单的验证，如空值验证，两次输入密码是否一致等，但业务关系紧密的验证，最好放在业务层。甚至有些验证只能在业务层验证，如“当前用户名不能与已有用户名重复”，这种验证需要访问持久化数据，需要由业务层完成。

这里之所以强调“表示层的服务端”，是因为一般在B/S系统中，都会在JavaScript里加入一些基本的数据验证，如空值检查，格式正则匹配等。这主要是为了减轻服务器负担，将大多数显然包含不合法数据的请求拒绝掉，而不发给服务端验证。当然，因为可能会出现JS被屏蔽或黑客恶意攻击行为，所以，所有验证不论JS中是否验证过，服务端（可能是表示层的服务端部分或业务层）一定要再进行验证。

**2.3.3、CRUD**

CRUD，即常说的增删改查操作。关于CRUD是否是业务层的职责，一直也是争议不断。因为目前并没有权威的定义，所以这里我斗胆说一下我对这个问题的看法。还请大家批判性阅读。

一说到“增删改查”，大家一定会觉得这理所当然是数据访问层的职责。我认为这个理解是对的，但是只对了一半！之所以这么说，是因为“增删改查”有两个层次含义。

第一个层次，是数据访问层次上的。在这个层次上，“增删改查”只是单纯的数据库操作，“增删改查”可以理解为“插入一条记录，删除一条记录，更新一条记录的信息，获取一条或多条记录”四个操作，其意义和着眼点完全是数据访问层面上的，不带有任何业务成分和业务知觉。这个层面上的CRUD应该属于数据访问层的职责。

第二个层次，是业务逻辑层次上的。在这个层次上，“增删改查”是业务领域内实体的变化以及一系列相关反应，“增删改查”可以理解为“领域内新增一个业务实体，领域内去掉一个业务实体，领域内一个业务实体更新了信息，得到领域内一个或多个业务实体的信息”。

两者最大的不同，是业务层面上的增删改查往往不是单纯的增加减少，还包括实体变化后相关的业务流程。下面举个例子：

“添加一个新的订单”——这是一条典型的“增”操作。在数据访问层面上，它的意义是“在表示订单的数据表里增加一条记录”；而在业务逻辑层面上，它的意义除了“领域内多了一个订单实体”外，还可能包括“根据业务规则判断是否是重复下单，根据金额对下订单客户的等级做相应提升、发送Email和短信通知客户等”。可以看到，业务层面上的“增”可能不仅是简单封装一个简单的插入记录，可能还要去做其他数据访问——提升用户等级，以及做一些非CRUD的业务操作 ——发送短信通知。

在许多稍微复杂的系统中，业务往往不仅仅是封装了一条数据访问操作，而是还有很多如计算等业务处理，一个业务操作期间可能要多次使用数据访问操作。退一步说，即使某个业务仅仅封装了一条数据访问操作，其意义和层面也是不同的，在数据访问层面，仅仅是多了一条记录，而业务逻辑层面，是领域内多了一个业务实体。也许其本质上都是往数据库插入一条记录，但人类的抽象思维可以将之在不同层面上区分，这也是人类思维层面的一种抽象能力的表现。例如，我们知道太阳升起不过是地球自转使得从背阴面转到了向阳面，但当人们看日出时，很少有人会说“看！我们从背阴面转到向阳面了！”，我们会说“看！日出！”，这就是同一事物的不同层次表现。

**2.3.4、存储过程**

也许是性能上的诱惑，许多人喜欢在数据库系统中写很复杂的存储过程。这样，许多业务操作就被写到存储过程中去了。我个人建议，除非对性能要求极高，否则最好还是不要用存储过程实现业务。例如，在一般的系统中，某个业务操作可能需要1秒，而是用了存储过程只用0.1秒，看上去存储过程将效率提高了10倍。但对大多数用户来说，1秒和0.1秒的差别并不大，但是这样做的话，业务会变得十分不容易维护。所以，我个人觉得，除非十分必要，还是不要用存储过程实现业务。

**3、业务逻辑的架构模式及实现**

Martin Fowler在《Patterns of Enterprise Application Architecture》一书中，总结了四种企业应用中业务逻辑的组织方式 ：Transcation Script，Domain Model，Table Module及Service Layer，另外，本书的第十章“Data Source Architecture Patterns”中包含一种模式——Active Record。结合软件体系结构的近期发展及个人的理解，我更倾向将Active Record归入业务逻辑的组织模式，而Service Layer应该不算做业务逻辑特有的模式，所以，在本文中，将介绍四种模式：Transcation Script，Table Module，Active Record及Domain Model。

**3.1、Transction Script**

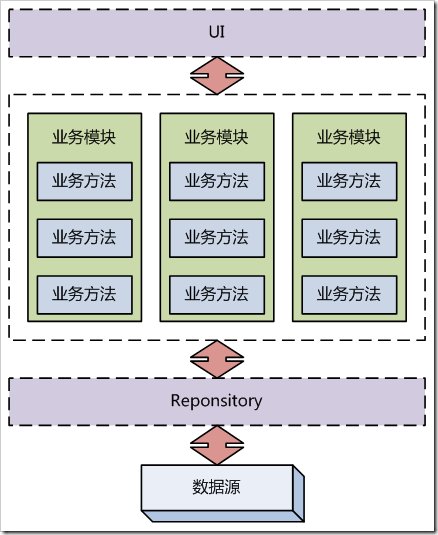
**3.1.1、概述**

Transction Script（以下简称TS）是一种面向过程的业务逻辑组织方式。这里首先要强调一点，这里的Transction一词与数据库系统中表示“事务”的 Transction没有任何联系。TS是将领域中的业务分解为一个个业务过程，每个过程实现一项业务功能，具体到程序中，一个业务过程往往映射到一个方法。TS是完全面向过程的业务组织模式，适合应用于业务逻辑较简单的场合。

应该说，我们见到的绝大多数系统都是以TS组织业务的。例如PetShop及Oxite等经典示例。有时为了方便维护，开发者会将同一领域实体相关的业务方法集中到一个类中，这里虽然用到了领域实体和类的概念，但和面向对象没有任何关系，完全是面向过程的。

当使用TS时，可以不需要数据访问层，而是将数据操作执行代码（如执行SQL或存储过程的代码）直接嵌入在业务方法中，有时为了复用性和维护性可以编写一个helper类封装数据库的操作。当然这并不是说TS不能配合数据访问层使用，但由于应用TS的场合一般业务非常简单，如果配合Repository或 ORM使用，业务逻辑层往往就会变得非常“瘦”，看起来仅仅是对数据访问层的封装。一般在需要支持多数据库的场合，要配合Repository和 Abstract Factory使用。

TS的示意图如下所示：



**图3-1、 Transcation Script架构示意**

可以看到，在TS中，业务层并没有面向对象的东西。也许会用到类，但类只是组织业务方法的模块，每个模块中有一个个业务方法，每个业务方法完成一个业务流程，完全按面向过程结构组织。

**3.1.2、分析**

* 什么时候可以用TS？

应该说，如果具备以下条件之一 ，你可以考虑TS：

1）系统业务十分简单直观，并且频繁变动的可能性不大

2）工期很紧，需要尽量压缩设计的时间，尽快投入编码

3）不能熟练掌握和使用OO进行系统的设计与开发

4）厌恶OO，就是喜欢面向过程

* TS的优点？

1）设计阶段投入较小，启动耗费低。因为TS较容易掌握，使用起点低，所以使用TS的初期投入较少

2）在业务比较简单直观的情况下，TS结构的代码直观易懂，具有良好的可维护性

* TS的缺点？

1）容易造成代码冗余。因为各个业务自行组织流程，所以减少了复用的机会，可能产生重复性代码

2）因为TS天生不适合业务复杂的系统，当系统业务较复杂时，可能会令业务层代码繁杂不堪

**3.2、Table Module**

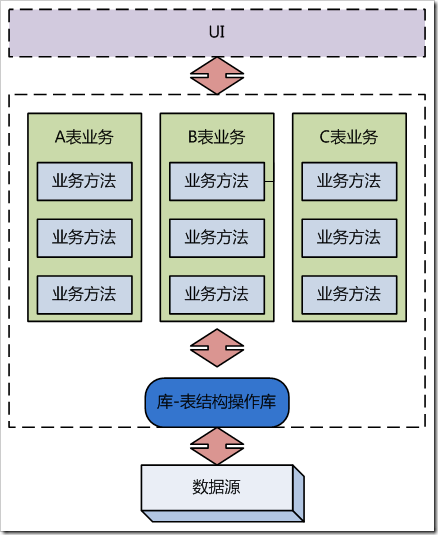
**3.2.1、概述**

Table Module（以下简称TM）同样是一种面向过程的业务逻辑组织方式，与TS不同的是，TM更贴近关系型数据库结构。在TS中，一般使用DTO等进行数据表示和传递，其着眼点一般在单个对象。而TM一般根据数据表组织业务模块，每个模块对应一个表，其中包含了这个表的相应处理。并且在业务层内，使用库-表结构的对象进行数据操作，做到最大限度与数据表的对应。业务组织一般按照面向过程组织。

一般当业务相对简单且业务基本集中在CRUD操作时，可以考虑TM。使用TM意味着使用数据驱动设计。通常自己实现一套库-表结构操作对象的库是难度比较大的，所以一般选用TM时，所使用的平台应该包括这么一套库。如.NET平台上的ADO.net就内置了丰富的库-表操作，DataSet，DataTable，DataAdapter等在TM架构的实现中可以起到非常方便的作用。

使用TM后，一般不需要再配合Reponsitory或ORM，因为此时的业务层也是面向过程和面向关系型结构的，无须映射。

TM的示意图如下：



**图3-2、Table Module架构示意**

在使用TM后，业务代码中往往有各种对象对应数据库中的库、表、记录、字段等元素，并提供类似关系数据库的操作。

**3.2.2、分析**

* 什么时候可以用TM？

如果同时 具备以下条件，你可以考虑TM：

1）系统业务较直观，以CRUD操作比较集中

2）整个开发的指导思想是数据驱动

3）所选用的平台有成熟的库-表操作库支持

* TM的优点？

1）类似关系数据库的数据操作方式非常直观，使得设计和编写数据操作功能的代码简单高效

* TM的缺点？

1）TM需要完全的数据驱动，从业务到UI传递、存放数据都要以表结构形式，造成一定程度上的不灵活

2）当业务并非CRUD集中型操作，特别是领域模型和数据库表模型差异较大时，使用TM组织业务的难度非常大

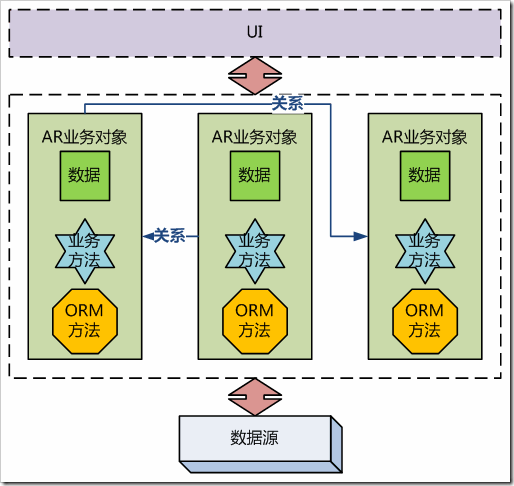
**3.3、Active Record**

**3.3.1、概述**

Active Record（以下简称AR）是一种面向对象的业务逻辑组织方式。AR适用于在业务较简单的情况下，应用面向对象思想进行设计。它的基本思想就是将领域中每个实体抽象出一个业务类（BO），然后，将这个实体的数据和行为封装成类的属性和方法。特别的，将CRUD功能也封装进BO中。也就是说，AR中的BO同时具备业务方法和持久化功能。其本身具有ORM的特性，其内部要处理关系实体间的关联问题。

使用AR时，一般最好有相应框架支持，否则完全手工实现AR有点麻烦。像Castle框架中就有AR功能，Linq to sql也有AR的意思。使用AR后，一般不需要再单独使用数据访问层。

AR的组织架构如下图：



**图3-3、Active Record架构示意**

从图3-3中可以看出，AR对业务领域进行了一个简单的OO抽象，将各个实体抽象为AR业务对象，AR业务对象内含有数据、业务方法及数据访问相关的 ORM方法。另外，AR业务对象要维护实体间简单的一对多和多对多等关系。

**3.3.2、分析**

* 什么时候可以用AR？

如果同时 具备以下条件，你可以考虑AR：

1）系统业务较直观

2）想尝试使用或习惯于使用OO进行系统设计与实现

3）平台上有成熟的AR框架可以用

* AR的优点？

1）使用OO的方式进行设计与实现，能在一定程度上避免冗余代码问题）

2）使用AR后，与某个实体相关的数据和业务全部集中于AR业务对象中，模块内聚性好，便于维护

3）实践证明，AR结构的业务层编码效率很高

* AR的缺点？

1）AR仍需要关注数据之间的关联，在一定程度上带有数据表和影子，没有完全摆脱数据驱动，所以当业务领域和数据库结构差距大时，实施困难

2）AR的CRUD是以个体为粒度的，当进行批量操作时，如一次查数千个数据，如果严格尊从AR就需要生成数千个AR业务对象，这简直是场灾难。所以在有大规模查询的情况下，可以考虑使用TS配合AR

3）如果业务非常复杂，AR将力不从心

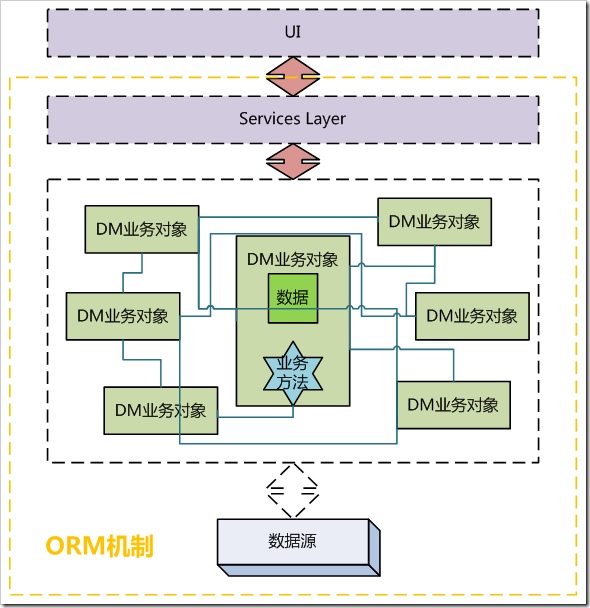
**3.4、Domain Model**

**3.4.1、概述**

Domain Model（以下简称DM）是一种适合领域驱动和为复杂业务系统组织业务的面向对象业务逻辑组织方式。前面三种架构模式都有一个共同的缺点——不适合业务复杂的系统。那么何为复杂何为简单？很抱歉，我给不出明确答案，而且我估计世界上任何一个人都很难给出标准的无争议答案。因为软件系统中的复杂和简单本身就是一个难以量化的指标，很多时候，只能靠专业人员的经验了。

我个人估计，世界上95%的软件系统其业务难度都不会超出上述三种模式的能力范围，而若你不幸遇到剩下的5%，恐怕目前只有Domain Model能帮你了。Domain Model是一种纯面向对象的业务架构模式，它的核心思想是获取领域中的各种实体抽象，然后完全按照现实领域中的情况去建模和运行。并且业务对象是“持久化无知”的。 关于“持久化无知”下面细讨论。这个模式十分复杂和难以掌握，但一旦掌握并使用，其能力绝对会超乎你的想象。

下面看一下DM的架构示意图：



**图3-4、Domain Model架构示意**

从图3-4中可以看出，DM看上去是个十分纠结的模式，而实际上，它确实很纠结！实际上，我认为如果能熟练掌握并运用DM进行业务逻辑的组织，那这人绝对是架构师中的大师级人物（我目前是做不到）。

还是先结合图示分析一下DM中的要点。

第一，DM中的业务对象是纯业务对象，不含数据访问操作。这个可以和AR中的业务对象对比一下。也就是说，DM中的业务对象是纯业务对象，它们只关注与业务的实现。

第二，DM的组织内部对象多，关系复杂，而这种关系不再只是那种简单的一对一、一对多的关系，而是领域中的各种依赖和关联的抽象，关系类型多，非常复杂。

第三，DM需要业务部分“持久化无知”。所谓持久化无知，指业务部分只需执行业务功能，而不必关系持久化。在使用DM时，必须设计一套ORM机制（注意这里用到了“机制”一词，而不是“框架”或“库”），使得在业务系统运行时，自动在必要的时候执行数据持久化操作。这也是为什么上图数据源和业务层间的箭头是虚线的关系。

上文曾说过，DM要最大程度模拟现实情况。而现实世界和软件世界最大的区别就是现实世界是“内存无限大、永不停机的”，可以把现实世界看成在一个无限大内存里永不停止运行的程序。而软件世界不同，它的内存有限制，我们不能将所有对象都放在内存，而且一旦掉电，它就会停止运行，正因如此，我们才需要持久化机制去配合DM模拟现实世界。为了让业务更接近现实，它必须对持久化过程毫无感觉。而一套持久化机制默默为其营造了一个好似内存无限大、永不停机的环境，因此DM才得以发挥威力。

第四，DM往往需要Services Layer的配合。因为DM内部仅有一个个业务对象，它们互相调用，并没有提供一个友好的接口与UI交互，所以在使用DM时，往往在其上对各种UI需要的服务进行封装（回顾一下Facade模式），形成一个Services Layer，以方便与UI交互。

**3.4.2、分析**

* 什么时候可以用DM？

如果同时 具备以下条件，你可以考虑DM：

1）系统业务极为复杂

2）有功底扎实和经验丰富的精通OO的架构及设计师

3）项目经费和时间充足

4）贯彻领域驱动设计

* DM的优点？

1）完全的OO思想运用，将使你享受到OO的所有优势

2）应付复杂业务的强力杀手锏。如果DM运用得当，将会使得复杂业务被高效解决

* DM的缺点？

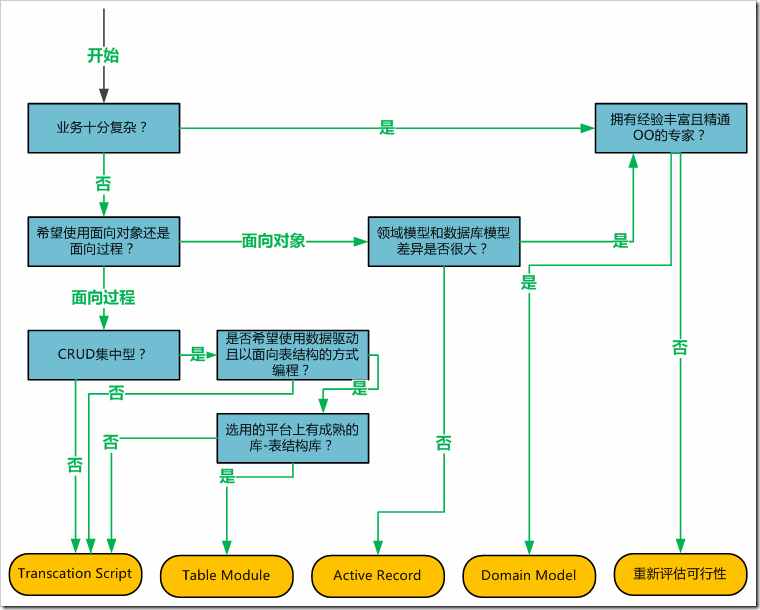
1）使用门槛极高，难度极大，如果团队中没有精通OO和系统架构且经验丰富的专家很难实施

2）设计过程极为复杂，可能会导致设计瘫痪

3）如何设计良好的ORM机制辅助DM是一大难题

**3.5、各种架构模式的比较及选择**

相信看过上文内容后，各位一定对各种业务组织模式及其特点、优劣、应用场景有了清晰地认识，如果我在这里再喋喋不休讨论各种模式的比较及如何选择，难免有侮辱各位智商之嫌O(∩\_∩)O~，所以这里我只给大家呈现一幅决策网络图，以期起到一个梳理和归纳总结的作用。



**图3-5、业务架构模式决策网络**

**（郑重声明：图3-5为本人原创，并非摘录自已有文献，因此此图的选型流程仅代表个人意见。由于笔者水平有限，不能保证此图一定合理和正确。因此在实际选型时请多多参考已有文献及咨询相关专家，此图只起总结归纳和探讨作用，不作为任何指导和规范。若因遵循此图选型而给项目带来的任何经济及其他方面损失，笔者不承担任何责任。）**

**4、结束语**

本文通过两篇文章的篇幅，先后介绍了业务逻辑的定义、相关理论及经典的业务逻辑相关的架构模式。本文中阐述了不少已有理论，亦掺杂诸多个人理解及看法。因此请各位在阅读时多进行批判吸收，同时参考以后经典文献及书目综合理解业务逻辑，切勿仅看我一家之言。

另外，由于本文仅仅是综述性文章，不能具名业务逻辑的各个方面，在深度上也基本是浅尝辄止。因此，若希望深入理解业务逻辑，可以看到相关经典书籍及文献。

**参考文献**

[1] [意]Dino Esposito, Andrea Saltarello, .NET软件架构之美英文版(原名Microsoft .NET Architecting Application for the Enterprise), 人民邮电出版社, 2009

[2] [美]Martin Fowler, 企业应用架构模式影印版(原名Patterns of Enterprise Application Architecture), 中国电力出版社, 2004

[3] [美]Mclaughlin, Pollice, West, 深入浅出面向对象分析与设计影印版(原名Head First OOA&D), 东南大学出版社, 2007

[4] Google, [www.google.com](http://www.google.com/)

## 三层架构图



## 三层实现步骤

1. 分析需求

例如：输入用户名，获取电话号码

1. 根据需求确定Sql语句

select CPhoneNum from TContactInfo where CName=@name;

1. 在DAL编写执行SQl语句的方法

在编写该方法前确定该方法的参数与返回值:

参数：执行Sql语句的时候需要外界传入值吗？如果需要则有参数，如果不需要则不需要参数

返回值：该Sql语句在数据库中执行完毕后数据库返回什么值，就将当前的方法的返回值设置为对应的类型(实际上最好是通用类型)。

1. 编写逻辑层方法

根据表现层的需求编写业务逻辑层方法。例如，用户名不存在，表现层会进行提示。在业务层中就需要进行用户名是否存在的判断。

在编写业务逻辑层的方法前，确定方法的参数与返回值。

确定参数：根据表现层采集到的数据，确定参数的个数与类型。

确定返回值：根据表现层需要的返回结果确定方法的返回值。

1. 表现层调用逻辑层方法实现功能就行了。

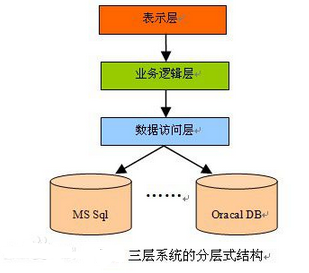
调用业务层的方法实现提示功能。

# 搭建三层

三层架构(3-tier architecture) 通常意义上的三层架构就是将整个业务应用划分为：

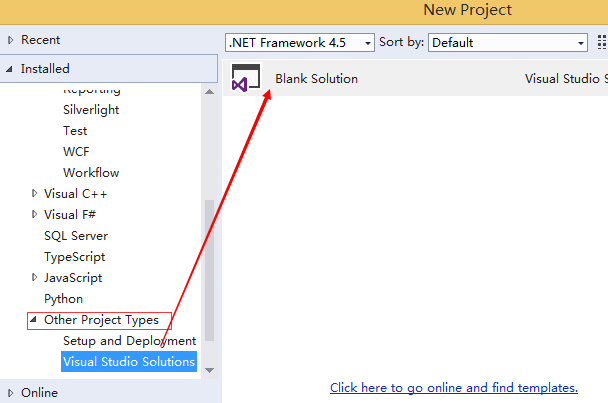
* 数据访问层（Data access layer）：主要是对原始数据（数据库或者文本文件等存放数据的形式）的操作层，而不是指原始数据，也就是说，是对数据的操作，而不是数据库，具体为业务逻辑层或表示层提供数据服务．
* 业务逻辑层（Business Logic Layer）：主要是针对具体的问题的操作，也可以理解成对数据层的操作，对数据业务逻辑处理，如果说数据层是积木，那逻辑层就是对这些积木的搭建。
* 表现层（Presentation layer）：主要表示WEB方式，也可以表示成WINFORM方式，WEB方式也可以表现成：aspx，如果逻辑层相当强大和完善，无论表现层如何定义和更改，逻辑层都能完善地提供服务。

区分层次的目的即为了“高内聚低耦合”的思想。在软件体系架构设计中，分层式结构是最常见，也是最重要的一种结构。

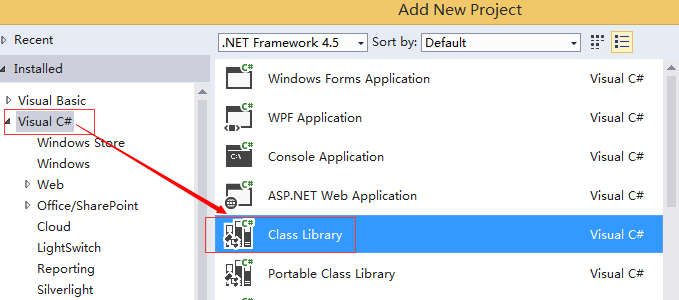


## 创建三层架构

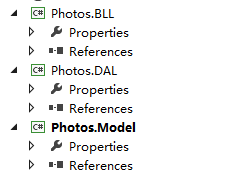
1. 新建空解决方案



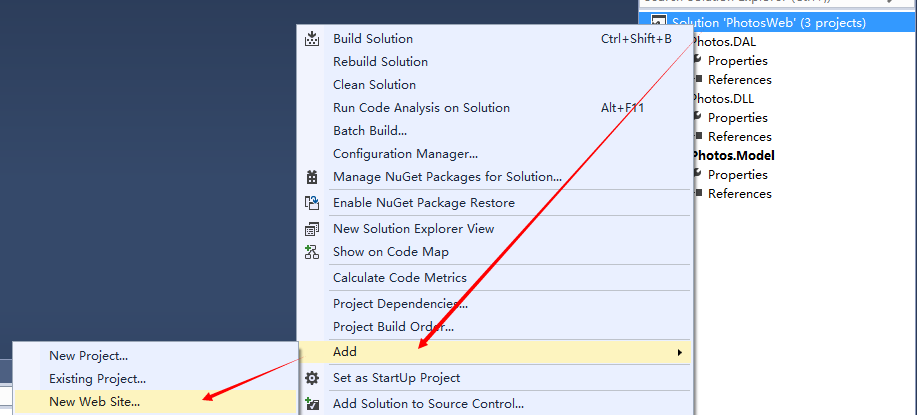
1. 新建三个类库项目



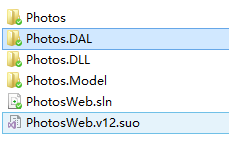
1. 这三个类库分别为：Model、DAL、BLL层。自带的class1.cs都删除掉。



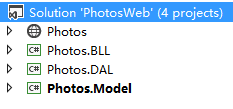
1. 新建一个website作为UI层。



1. 文件夹架构如下：

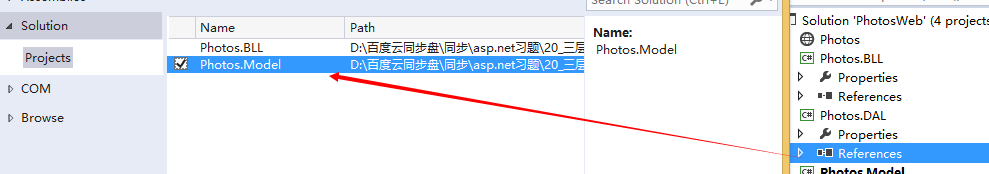


1. 项目架构如下：



## 添加引用

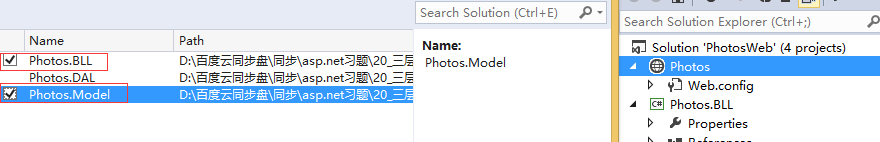
1. DAL引用Model



1. BLL引用DAL和Model

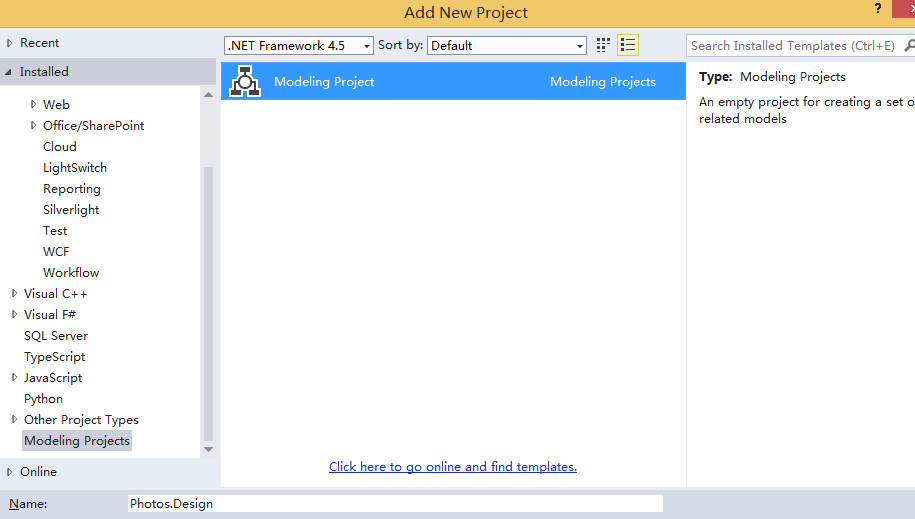


1. UI引用BLL和Model

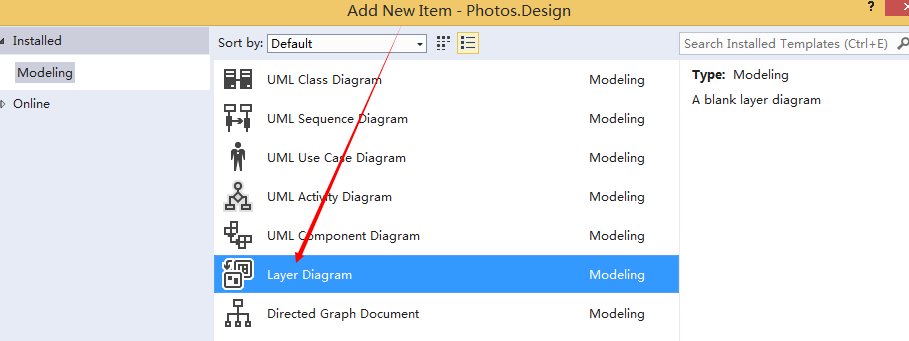


## 设计建模

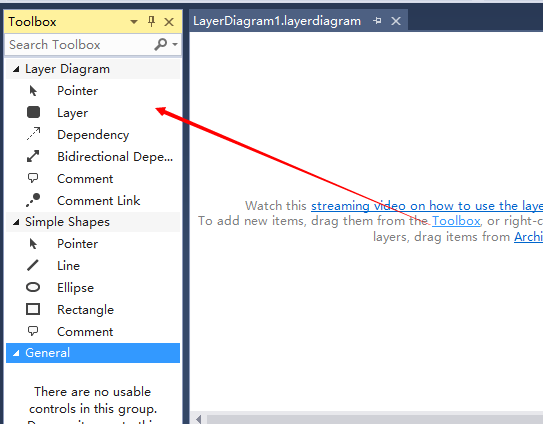
1. 添加建模项目



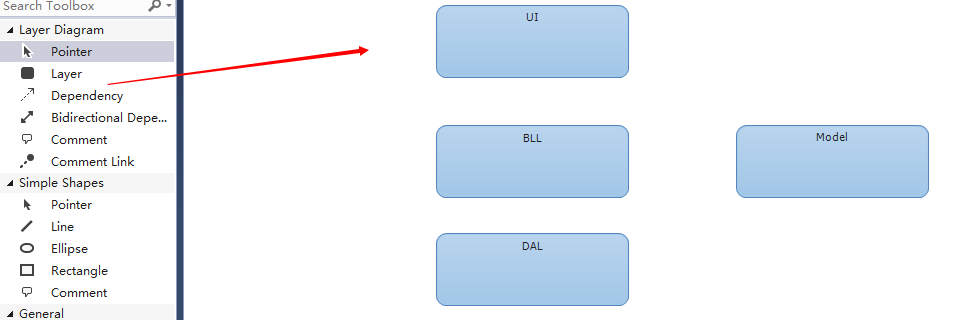
1. 添加一个层图



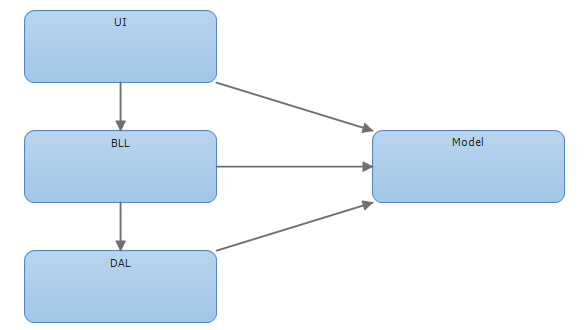
1. 显示工具箱



1. 拖到层到面板。总共添加4个层，双击命名



1. 添加依赖关系（Dependency）。



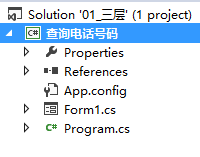
1. UI依赖于BLL和Model；BLL依赖于DAL和Model；DAL依赖于Model。

## 案例：输入用户名查询电话号码

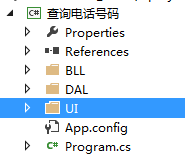
### 三层搭建

三层的搭建方式：可以在一个项目中分为三个文件夹，或者三个层分为三个项目（大的项目）。我们在这里用三个文件夹来分层。

首先创建一个winform项目



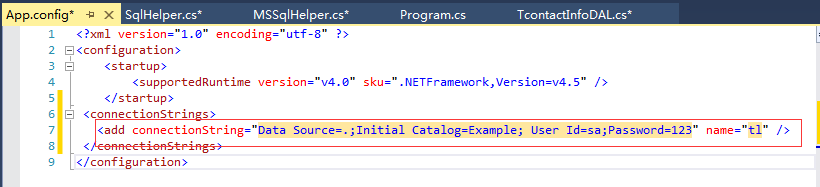
在该项目中创建三个文件夹，删除Form1.cs。



### DAL层编写

1. 分析需求，在DAL中实现查询

首先进行app.config的配置



在DAL中，创建cs文件，编写SqlParamsDictionary类，用来封装泛型

|  |
| --- |
| using *System*;  using *System*.*Collections*.*Generic*;  using *System*.*Linq*;  using *System*.*Text*;  using *System*.*Threading*.*Tasks*;  namespace 查询电话号码.DAL  {  public sealed class SqlParamsDictionary:*Dictionary*<string,object>  {  }  } |

编写SqlHelper父类，用来封装sql语句。

|  |
| --- |
| using *System*;  using *System*.*Collections*.*Generic*;  using *System*.*Data*;  using *System*.*Data*.*SqlClient*;  using *System*.*Linq*;  using *System*.*Text*;  using *System*.*Threading*.*Tasks*;  namespace 查询电话号码.DAL  {  public abstract class SqlHelper  {  #region Fileds  private string \_connectionString;  public string ConnectionString  {  get { return \_connectionString; }  set { \_connectionString = value; }  }  #endregion  protected SqlHelper(string connectionStrings)  {  this.ConnectionString = connectionStrings;  }  public abstract object ExecuteScalar(string sql, params SqlParamsDictionary[] spc);  public abstract object ExecuteScalar(string sql, *CommandType* type, params SqlParamsDictionary[] spc);  public abstract int ExecuteNonQuery(string sql, params SqlParamsDictionary[] spc);  public abstract int ExecuteNonQuery(string sql, *CommandType* type, params SqlParamsDictionary[] spc);  public abstract *DataTable* ExecuteDataTable(string sql, params SqlParamsDictionary[] spc);  public abstract *DataTable* ExecuteDataTable(string sql, *CommandType* type, params SqlParamsDictionary[] spc);  }  } |

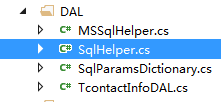
编写实现SqlHelper的类，MSSqlHelper用来连接Sql server数据库，并实现Sql语句增删改查。

|  |
| --- |
| using *System*;  using *System*.*Collections*.*Generic*;  using *System*.*Data*;  using *System*.*Data*.*SqlClient*;  using *System*.*Linq*;  using *System*.*Text*;  using *System*.*Threading*.*Tasks*;  namespace 查询电话号码.DAL  {  public sealed class MSSqlHelper : SqlHelper  {  public MSSqlHelper(string connectionStringName)  : base(connectionStringName)  { }  /// <summary>  /// 创建连接  /// </summary>  /// <returns>返回连接对象</returns>  private *SqlConnection* getConn()  {  string connStr = *System*.*Configuration*.*ConfigurationManager*.*ConnectionStrings*[ConnectionString].ConnectionString;  return new *SqlConnection*(connStr);  }  /// <summary>  /// 返回首行首列  /// </summary>  /// <param name="sql">要执行的sql语句</param>  /// <param name="type">执行的sql类型，sql语句、存储过程</param>  /// <param name="spc">sql参数</param>  /// <returns></returns>  public override object ExecuteScalar(string sql, *CommandType* type, SqlParamsDictionary[] spc)  {  object obj = null;  using (*SqlConnection* conn = getConn())  {  using (*SqlCommand* cmd = new *SqlCommand*(sql, conn))  {  cmd.*CommandType* = type;  if (spc.*Length* > 0 || spc != null)  {  for (int i = 0; i < spc.*Length*; i++)  {  foreach (var item in spc[i])  {  cmd.*Parameters*.*Add*(new *SqlParameter*(item.*Key*, item.*Value*));  }  }  }  conn.*Open*();  obj = cmd.ExecuteScalar();  }  }  return obj;  }  public override object ExecuteScalar(string sql, params SqlParamsDictionary[] spc)  {  return ExecuteScalar(sql, *CommandType*.*Text*, spc);  }  public override int ExecuteNonQuery(string sql, params SqlParamsDictionary[] spc)  {  return ExecuteNonQuery(sql, *CommandType*.*Text*, spc);  }  /// <summary>  /// 返回受影响的行数  /// </summary>  /// <param name="sql">执行的Sql语句</param>  /// <param name="type">定义sql语句的类型：一般语句，存储过程</param>  /// <param name="spc">sql可变参数</param>  /// <returns></returns>  public override int ExecuteNonQuery(string sql, *CommandType* type, params SqlParamsDictionary[] spc)  {  int nums = -1;  using (*SqlConnection* conn = getConn())  {  using (*SqlCommand* cmd = new *SqlCommand*(sql, conn))  {  cmd.*CommandType* = type;  if (spc.*Length* > 0 || spc != null)  {  for (int i = 0; i < spc.*Length*; i++)  {  foreach (var item in spc[i])  {  cmd.*Parameters*.*Add*(new *SqlParameter*(item.*Key*, item.*Value*));  }  }  }  conn.*Open*();  nums = cmd.ExecuteNonQuery();  }  }  return nums;  }  public override *System*.*Data*.*DataTable* ExecuteDataTable(string sql, params SqlParamsDictionary[] spc)  {  return ExecuteDataTable(sql, *CommandType*.*Text*, spc);  }  /// <summary>  /// 返回Datatable  /// </summary>  /// <param name="sql">执行的Sql语句</param>  /// <param name="type">定义sql语句的类型：一般语句，存储过程</param>  /// <param name="spc">sql可变参数</param>  /// <returns></returns>  public override *DataTable* ExecuteDataTable(string sql, *CommandType* type, params SqlParamsDictionary[] spc)  {  *DataTable* dt = new *DataTable*();  using (*SqlConnection* conn = getConn())  {  using (*SqlCommand* cmd = new *SqlCommand*(sql, conn))  {  cmd.*CommandType* = type;  if (spc.*Length* > 0 || spc != null)  {  for (int i = 0; i < spc.*Length*; i++)  {  foreach (var item in spc[i])  {  cmd.*Parameters*.*Add*(new *SqlParameter*(item.*Key*, item.*Value*));  }  }  using (*SqlDataAdapter* adapter = new *SqlDataAdapter*(cmd))  {  conn.*Open*();  adapter.*Fill*(dt);  }  }  }  return dt;  }  }  }  } |

最后创建TContactInfoDAL，用来实现对TContactInfo表的具体操作。

|  |
| --- |
| using *System*;  using *System*.*Collections*.*Generic*;  using *System*.*Linq*;  using *System*.*Text*;  using *System*.*Threading*.*Tasks*;  namespace 查询电话号码.DAL  {  class TcontactInfoDAL  {  /// <summary>  /// 根据名称获得电话号码  /// </summary>  /// <param name="name"></param>  /// <returns></returns>  object GetPhoneNum(string name)  {  object obj = null;  string sql = "select CPhoneNum from TContactInfo where Cname = @name";  SqlParamsDictionary spdic = new SqlParamsDictionary();  spdic.*Add*("@name", name);  SqlHelper sqlhelper = new MSSqlHelper("tl");  obj = sqlhelper.ExecuteScalar(sql, spdic);  return obj;  }  }  } |

DAL结构如下：



### BLL层编写

根据表现层需求进行用户名是否存在的校验以及返回需要的数据。

创建TconatactInfoBLL:

|  |
| --- |
| using *System*;  using *System*.*Collections*.*Generic*;  using *System*.*Linq*;  using *System*.*Text*;  using *System*.*Threading*.*Tasks*;  using 查询电话号码.DAL;  namespace 查询电话号码.BLL  {  class TconatactInfoBLL  {  //判断用户是否存在  public bool IsExist(string name)  {  bool isExist = false;  isExist = getPhoneNum(name) != null;  return isExist;  }  /// <summary>  /// 返回表示层进行展示需要的数据类型  /// </summary>  /// <param name="name"></param>  /// <returns></returns>  public string getPhoneNum(string name)  {  TcontactInfoDAL dal = new TcontactInfoDAL();  string str = (string)dal.GetPhoneNum(name);  return str;  }  }  } |

### UI层编写

在UI层新建winform窗体。

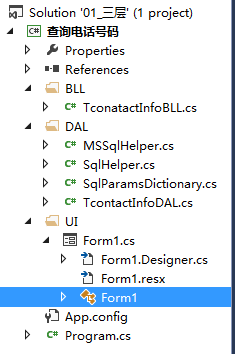
并在program.cs中修改

|  |
| --- |
| *Application*.*Run*(new Form1()); |

UI层进行数据收集、提示、展示工作。

|  |
| --- |
| private void button1\_Click(object sender, *EventArgs* e)  {  //1.采集数据  string name = txt\_name.*Text*.*Trim*();  TconatactInfoBLL bll = new TconatactInfoBLL();  if(!bll.IsExist(name))  {  *MessageBox*.*Show*("用户名不存在");  }  //展示数据  txt\_phoneNum.*Text* = bll.getPhoneNum(name);  } |

最后的分层如下：



## 案例：用户登录

三层搭建方式与：案例：输入用户名查询电话号码相似。不再赘述。

### 创建T\_UserInfo的DAL层

|  |
| --- |
| namespace 登录与修改密码.DAL  {  class T\_UserInfoDAL  {  public int GetUserLogin(string name, string pwd)  {  SqlHelper sqlhelper = new MSSqlHelper("tl");  string sql = "select count(\*) from T\_UserInfo where Uname=@name and UPassword=@pwd";  SqlParamsDictionary spdic = new SqlParamsDictionary();  spdic.*Add*("@name", name);  spdic.*Add*("@pwd", pwd);  return sqlhelper.ExecuteNonQuery(sql, spdic);  } |

### 创建T\_UserInfo的BLL层

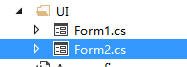
|  |
| --- |
| namespace 登录与修改密码.BLL  {  class T\_UserInfoBLL  {  public bool LoginStatus(string name,string pwd)  {  T\_UserInfoDAL dal = new T\_UserInfoDAL();  return dal.GetUserLogin(name,pwd)>0;  }  }  } |

### UI层进行数据采集与信息提示

|  |
| --- |
| private void button1\_Click(object sender, *EventArgs* e)  {  string name = txt\_name.*Text*.*Trim*();  string pwd = txt\_pwd.*Text*.*Trim*();  T\_UserInfoBLL bll = new T\_UserInfoBLL();  if (bll.LoginStatus(name,pwd))  {  *MessageBox*.*Show*("登录失败");  }else  {  *MessageBox*.*Show*("登录失败");  }  } |

## 案例：修改密码

新建一个winform放到UI层中



### 修改Form2的构造函数接受传入的数据

|  |
| --- |
| public Form2(string name):this()  {  Uname = name;  lbl\_username.*Text* = "欢迎您： "+Uname;  }  string Uname; |

### 修改Form1使登录成功跳转到Form2

|  |
| --- |
| if (bll.LoginStatus(name,pwd))  {  this.*Hide*();  Form2 form2 = new Form2(name);  form2.*Show*();    }else  {  *MessageBox*.*Show*("登录失败");  } |

### 为DAL层的T\_UserInfoDAL添加修改密码的方法

|  |
| --- |
| public int ChangePwd(string name,string pwd)  {  SqlHelper sqlhelper = new MSSqlHelper("tl");  string sql = "update T\_UserInfo set UPassword=@pwd where Uname=@name";  SqlParamsDictionary spdic = new SqlParamsDictionary();  spdic.*Add*("@name", name);  spdic.*Add*("@pwd", pwd);  return sqlhelper.ExecuteNonQuery(sql, spdic);  } |

### 为BLL层的T\_UserInfoDAL数据访问情况进行校验，并调用方法执行sql语句

|  |
| --- |
| public bool ChangePwd(string name,string pwd)  {  T\_UserInfoDAL dal = new T\_UserInfoDAL();  return dal.ChangePwd(name, pwd)>0;  } |

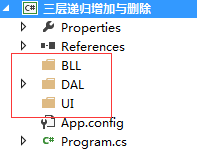
### 在From2中进行数据收集、展示

|  |
| --- |
| private void button1\_Click(object sender, *EventArgs* e)  {  string pwd = txt\_pwd.*Text*.*Trim*();  string pwdAgain = txt\_pwdAgain.*Text*.*Trim*();  if(pwd == pwdAgain)  {  T\_UserInfoBLL bll = new T\_UserInfoBLL();  if(bll.ChangePwd(Uname,pwdAgain))  {  lbl\_username.*Text* =Uname + ":密码修改成功！";  }  }else  {  *MessageBox*.*Show*("密码不一致！请重新输入");  }  } |

# 案例：递归增加与删除

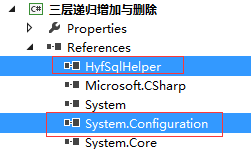
## 搭建三层

新建一个winform窗体项目，创建三个文件夹。并删除Form1

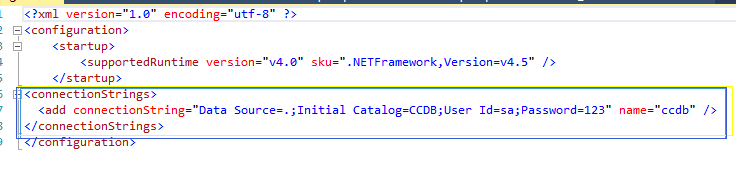


## 引入程序集并配置文件

引入以下两个程序集：



进行文件配置：



## 需求分析

1. 递归读取城市信息到到treeview上。
2. 删除所选的节点代表的省市及其下的节点代表的城市。

、

## 根据需求构建Sal语句

根据AreaPid读取：”select AreaId, AreaName from TblArea where AreaPid = @pid and IsDeleted = 0“;

根据AreaId软删除：”update TblArea set IsDeleted = 1 where Areaid=@id”;

## 确定Sql返回值

返回值为单行，有1个返回值时：可以用int、string、object等通用类型。

返回值为单行，有2个返回值：可以用Dictionary

返回值为单行，有多个返回值： 一般创建model，即一个类用属性来存值。

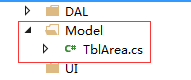
有多行的返回值: 用list<>;

在这个案例中，每一行有2个返回值，有多行，可以用dictionary<>或者List<>

这里我们用List<>来实现（只有2个值，dictionary更好，但list扩展性好）。

## 创建Model

新建一个model层，并添加一个TblArea类



数据库中每一个字段作为类的一个属性：

|  |
| --- |
| namespace 三层递归增加与删除.Model  {  class TblArea  {  public int AreaId { get; set; }  public string AreaName { get; set; }  public int AreaPid { get; set; }  public int IsDeleted { get; set; }  }  } |

## 在DAL层中创建TblAreaDal类进行数据访问

|  |
| --- |
| using *System*;  using *System*.*Collections*.*Generic*;  using *System*.*Linq*;  using *System*.*Text*;  using *System*.*Threading*.*Tasks*;  using HyfSqlHelper;  using 三层递归增加与删除.Model;  using *System*.*Data*.*SqlClient*;  namespace 三层递归增加与删除.DAL  {  class TblAreaDal  {  /// <summary>  /// 根据id获得名称  /// </summary>  /// <param name="pid">id</param>  /// <returns>返回类型为TblArea类型的list</returns>  public *List*<TblArea> GetAreaByPid(int pid)  {  *List*<TblArea> list = new *List*<TblArea>();  string sql = "select AreaId,AreaName from TblArea where AreaPid = @pid and IsDeleted=0";  SqlHelper sqlhelper = new MSSqlHelper("ccdb");  *SqlParameter* p = new *SqlParameter*("@pid", pid);  using (*SqlDataReader* reader = sqlhelper.*ExecuteReader*(sql, p))  {  if (reader.*HasRows*)  {  while (reader.*Read*())  {  TblArea model = new TblArea();  model.AreaId = reader.*GetInt32*(reader.*GetOrdinal*("AreaId"));  model.AreaName = reader.*GetString*(reader.*GetOrdinal*("AreaName"));  list.*Add*(model);  }  }  }  return list;  }  }  } |

## 在BLL中，添加TblAreaBll

|  |
| --- |
| using *System*;  using *System*.*Collections*.*Generic*;  using *System*.*Linq*;  using *System*.*Text*;  using *System*.*Threading*.*Tasks*;  using 三层递归增加与删除.DAL;  using 三层递归增加与删除.Model;  namespace 三层递归增加与删除.BLL  {  class TblAreaBLL  {  //因为递归的方法需要TreeNodeCollection，而它只在winform中有效，所以不适合在BLL中。这里的BLL只是访问数据的中转站。  public *List*<TblArea> GetAreaById(int pid)  {  TblAreaDal dal = new TblAreaDal();  return dal.GetAreaByPid(pid);  }  }  } |

## 表现层中进行递归增加

|  |
| --- |
| private void button1\_Click(object sender, *EventArgs* e)  {  *TreeNodeCollection* c = treeView1.*Nodes*;  int i = 0;  LoadArea(i, c);  }  /// <summary>  /// 方法需要传入TreeNodeCollection参数，因此不能在BLL层中。  /// </summary>  /// <param name="pid"></param>  /// <param name="nodes"></param>  static void LoadArea(int pid, *TreeNodeCollection* nodes)  {  TblAreaBLL bll = new TblAreaBLL();  *List*<TblArea> modes = bll.GetAreaById(pid);  foreach (TblArea item in modes)  {  *TreeNode* node = nodes.*Add*(item.AreaName);  node.*Tag* = item.AreaId;  LoadArea(item.AreaId, node.*Nodes*);  }  } |

## 递归删除

删除的步骤与递归添加类似。但是删除时不需要TreeNodeCollection。所以递归逻辑可以放到BLL层。

### DAL层

|  |
| --- |
| public int DeleteAreaByID(int id)  {  string sql = "update TblArea set IsDeleted = 1 where Areaid=@id ";  *SqlParameter* sp = new *SqlParameter*("@id", id);  SqlHelper sh = new MSSqlHelper("ccdb");  return sh.*ExecuteNonQuery*(sql, sp);  } |

### BLL层

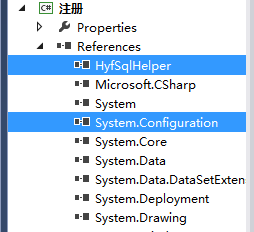
|  |
| --- |
| public void DeleteById(int id)  {  TblAreaDal dal = new TblAreaDal();  *List*<TblArea> models = dal.GetAreaByPid(id);  foreach (TblArea item in models)  {  //递归  DeleteById(item.AreaId);  }  dal.DeleteAreaByID(id);  } |

### UI层

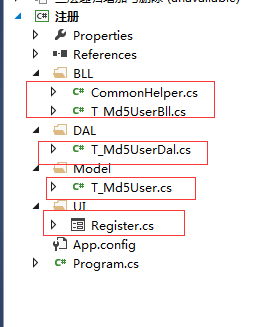
|  |
| --- |
| private void button2\_Click(object sender, *EventArgs* e)  {  *TreeNode* node = treeView1.*SelectedNode*;  int id = (int)node.*Tag*;  TblAreaBLL bll = new TblAreaBLL();  bll.DeleteById(id);  node.*Remove*();  } |

# 案例：用户注册

## 进行文件配置与添加引用



## 三层与类的结构



## 在model层的类T\_Md5User中创建属性

为T\_Md5User的每一个字段创建一个属性。

|  |
| --- |
| //Mid, Mname, MPassword, MGender, MErroTimes, LastErrorTimes  class T\_Md5User  {  public int Mid { get; set; }  public string Mname { get; set; }  public string MPassword { get; set; }  public bool? MGender { get; set; }  public int ErrorTimes { get; set; }  public *DateTime*? LastErrorTimes { get; set; }  } |

## 在Dal层进行插入数据与用户名查询

|  |
| --- |
| class T\_Md5UserDal  {  /// <summary>  /// 插入注册信息  /// </summary>  /// <param name="model">携带用户注册信息的model类</param>  /// <returns></returns>  public int UserRegister(T\_Md5User model)  {  *SqlHelper* sqlhp = new *MSSqlHelper*("example");  string sql = "insert into T\_Md5User(Mname,MPassword,MGender) values (@name,@pwd,@gender)";  *SqlParameter*[] sp = new *SqlParameter*[] {  new *SqlParameter*("@name", model.Mname) ,  new *SqlParameter*("@pwd",model.MPassword),  new *SqlParameter*("@gender",model.MGender==null?*DBNull*.*Value*:(object)model.MGender)  };  return sqlhp.*ExecuteNonQuery*(sql, sp);  }  /// <summary>  /// 根据用户名来查找数据库中该用户名的数量。  /// </summary>  /// <param name="name"></param>  /// <returns></returns>  public object getUserCount(string name)  {  *String* sql = "select count(\*) from T\_Md5User where Mname=@name";  *SqlParameter* sp = new *SqlParameter*("@name", name);  *SqlHelper* sqlhelper = new *MSSqlHelper*("example");  return sqlhelper.*ExecuteScalar*(sql, sp);  }  } |

## 在BLL层进行业务处理

判断用户名是否存在、密码是否一致、用户名输入是否合法；

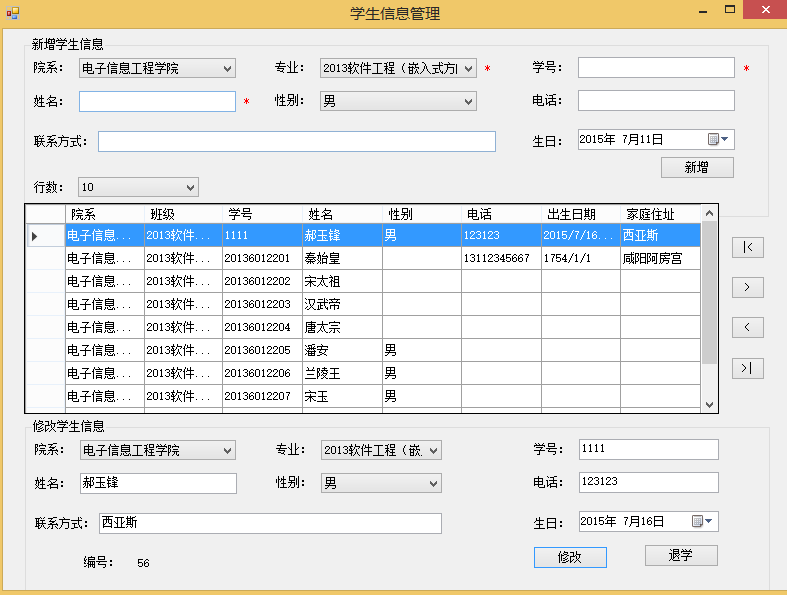
|  |
| --- |
| namespace 注册.BLL  {  class T\_Md5UserBll  {  /// <summary>  /// 传入的密码进行MD5编码，并调用DAL进行注册  /// </summary>  /// <param name="model"></param>  /// <returns></returns>  private int UserRegister(T\_Md5User model)  {  model.MPassword = CommonHelper.GetMD5FromString(model.MPassword);  T\_Md5UserDal dal = new T\_Md5UserDal();  return dal.UserRegister(model);  }  /// <summary>  /// 返回用户名数量  /// </summary>  /// <param name="name"></param>  /// <returns></returns>  private int GetUserCount(string name)  {  T\_Md5UserDal dal = new T\_Md5UserDal();  return (int)dal.getUserCount(name);  }  /// <summary>  /// 判断用户名是否存在、密码是否一致、用户名输入是否合法。  /// </summary>  /// <param name="model"></param>  /// <param name="pwdAgain"></param>  /// <returns>返回值为枚举类型</returns>  public RegistResult GetRegistResult(T\_Md5User model, string pwdAgain)  {  //数量大于0表示用户名已经存在  if (GetUserCount(model.Mname) > 0)  {  return RegistResult.NameRepeater;  }  else  {  //判断密码是否一致。  if (model.MPassword == pwdAgain)  {  //正则表达式判断用户名是否合法  if (*Regex*.*IsMatch*(model.Mname, "^[\_0-9a-zA-Z]+$"))  {  //进行注册  UserRegister(model);  return RegistResult.Success;  }  else  {  return RegistResult.NameTooShort;  }  }  else  {  return RegistResult.PasswordDiff;  }  }  }  }  /// <summary>  /// 枚举保存注册的状态。  /// </summary>  enum RegistResult  {  NameRepeater,  NameTooShort,  PasswordDiff,  Success  }  } |

## 在UL层搜集数据，反馈注册结果

|  |
| --- |
| namespace 注册.UI  {  public partial class Register : *Form*  {  public Register()  {  InitializeComponent();  }  private void button1\_Click(object sender, *EventArgs* e)  {  //搜集数据-----------------------------  T\_Md5User model = new T\_Md5User();  model.Mname = txt\_name.*Text*.*Trim*();  model.MPassword = txt\_pwd.*Text*;  string pwdagin = txt\_pwdagain.*Text*;  if (rbman.*Checked* == true)  {  model.MGender = true;  }  else if (rbwoman.*Checked* == true)  {  model.MGender = true;  }  else  {  model.MGender = null;  }  //-------------注册结果进行相应的反馈--------------------  T\_Md5UserBll bll = new T\_Md5UserBll();  RegistResult result = bll.GetRegistResult(model, pwdagin);  switch (result)  {  case RegistResult.Success:  *MessageBox*.*Show*("注册成功！");  break;  case RegistResult.NameRepeater:  *MessageBox*.*Show*("用户名已存在");  break;  case RegistResult.PasswordDiff:  *MessageBox*.*Show*("密码不一致");  break;  case RegistResult.NameTooShort:  *MessageBox*.*Show*("包含非法字符");  break;  }  }  }  } |

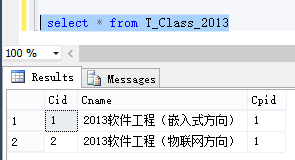
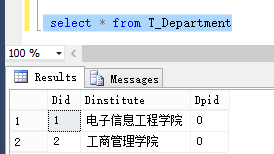
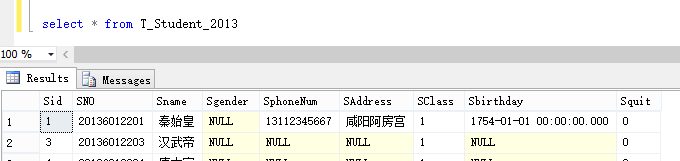
# 综合：学生信息管理

## 效果



## 数据库设计

三个表



其中 SClass=cid，Cpid=Did；（外键约束）

## Model层

模拟学生表

|  |
| --- |
| using *System*;  using *System*.*Collections*.*Generic*;  using *System*.*Linq*;  using *System*.*Text*;  using *System*.*Threading*.*Tasks*;  using HyfSqlHelper;  using Model;  using *System*.*Data*.*SqlClient*;  using *System*.*Data*;  namespace DAL  {  public class ElectronicInformationDal  {  /// <summary>  /// 分页查询学生信息  /// </summary>  /// <param name="instuitute">所在院系</param>  /// <param name="bindex">开始的索引</param>  /// <param name="eindex">结束的索引</param>  /// <returns>ElectronicInformation实体list集合</returns>  public *List*<ElectronicInformation> getStuInfoByInstitute(string instuitute, int bindex, int eindex)  {  *List*<ElectronicInformation> list = new *List*<ElectronicInformation>();  //分页查询：开窗函数  string sql = @"select \* from ( select T\_Student\_2013.\* ,Cname,Dinstitute ,row\_number() over (order by SNO ) as paging from T\_Student\_2013  join T\_Class\_2013 on T\_Student\_2013.SClass = T\_Class\_2013.Cid  join T\_Department on T\_Department.Did = T\_Class\_2013.Cpid) stuinfo where Dinstitute=@institute and squit=0 and paging between @bindex and @eindex ";  //参数集合  *SqlParameter*[] sp = new *SqlParameter*[] {  new *SqlParameter*("@institute", instuitute),  new *SqlParameter*("@bindex",bindex),  new *SqlParameter*("@eindex",eindex)  };  SqlHelper sqlhelper = new MSSqlHelper("SIAS");  *SqlDataReader* reader = sqlhelper.ExecuteReader(sql, sp);  //如果有数据  if (reader.*HasRows*)  {  while (reader.*Read*())  {  ElectronicInformation ei = new ElectronicInformation();  //获取的值赋予实体  ei.Sid = reader.*GetInt32*(reader.*GetOrdinal*("Sid"));  ei.SNO = reader.*GetString*(reader.*GetOrdinal*("SNO"));  ei.Sname = reader.*GetString*(reader.*GetOrdinal*("Sname"));  ei.Sgender = reader.*IsDBNull*(reader.*GetOrdinal*("Sgender")) ? null : reader.*GetString*(reader.*GetOrdinal*("Sgender"));  ei.SphoneNum = reader.*IsDBNull*(reader.*GetOrdinal*("SphoneNum")) ? null : reader.*GetString*(reader.*GetOrdinal*("SphoneNum"));  ei.SAddress = reader.*IsDBNull*(reader.*GetOrdinal*("SAddress")) ? null : reader.*GetString*(reader.*GetOrdinal*("SAddress"));  ei.Sbirthday = reader.*IsDBNull*(reader.*GetOrdinal*("SBirthday")) ? null : (*DateTime*?)reader.*GetDateTime*(reader.*GetOrdinal*("SBirthday"));  //模拟的外键关系的关联的实体  T\_Class classmodel = new T\_Class();  classmodel.Cname = reader.*GetString*(reader.*GetOrdinal*("Cname"));  ei.ClassModel = classmodel;  T\_Department deparmentModel = new T\_Department();  deparmentModel.Dinstitute = reader.*GetString*(reader.*GetOrdinal*("Dinstitute"));  ei.DepartmentModel = deparmentModel;  list.*Add*(ei);  }  }  return list;  }  /// <summary>  /// 查询表中所有院系  /// </summary>  /// <returns>院系实体集合</returns>  public *List*<T\_Department> getDepartment()  {  *List*<T\_Department> model = new *List*<T\_Department>();  string sql = "select Did,Dinstitute from T\_department ";  SqlHelper sqlhelper = new MSSqlHelper("SIAS");  *SqlDataReader* reader = sqlhelper.ExecuteReader(sql);  if (reader.*HasRows*)  {  while (reader.*Read*())  {  T\_Department dm = new T\_Department();  dm.Did = reader.*GetInt32*(0);  dm.Dinstitute = reader.*GetString*(1);  model.*Add*(dm);  }  }  return model;  }  /// <summary>  /// 查询某个院系的所有的班级  /// </summary>  /// <param name="institute">院系</param>  /// <returns>班级实体的集合</returns>  public *List*<T\_Class> getClass(string institute)  {  *List*<T\_Class> model = new *List*<T\_Class>();  string sql = @"select cid ,Cname from T\_Class\_2013 join T\_department on T\_department.did= T\_Class\_2013.cpid where Dinstitute= @institute ";  *SqlParameter* sp = new *SqlParameter*("@institute", institute);  SqlHelper sqlhelper = new MSSqlHelper("SIAS");  *SqlDataReader* reader = sqlhelper.ExecuteReader(sql, sp);  if (reader.*HasRows*)  {  while (reader.*Read*())  {  T\_Class cm = new T\_Class();  cm.Cid = reader.*GetInt32*(0);  cm.Cname = reader.*GetString*(1);  model.*Add*(cm);  }  }  return model;  }  /// <summary>  /// 查询T\_Student\_2013班级表中学生的总数  /// </summary>  /// <returns>object类型的总数</returns>  public object GetStuNums()  {  string sql = "select count(\*) from T\_Student\_2013";  SqlHelper sqlhelper = new MSSqlHelper("SIAS");  object obj = sqlhelper.ExecuteScalar(sql);  return obj;  }  /// <summary>  /// 添加学生信息到表中  /// </summary>  /// <param name="model">学生实体</param>  /// <returns></returns>  public int AddStuToTalbe(ElectronicInformation model)  {  //@gender,@phoneNum,@addreass,@birityday,  //,Sgender,SPhoneNum,SAddress,Sbirthday  string sql = @"insert into T\_Student\_2013(SNO,Sname,Sgender,SPhoneNum,SAddress,Sbirthday,SClass) values (@no,@name,@gender,@phoneNum,@addreass,@birityday,@class)";  *SqlParameter*[] sp = new *SqlParameter*[]  {  new *SqlParameter*("@no", model.SNO),  new *SqlParameter*("@gender",model.Sgender),  new *SqlParameter*("@name", model.Sname),  //三元运算符两个结果值需要能相互转换  new *SqlParameter*("@phoneNum",model.SphoneNum!=null?(*Object*)model.SphoneNum:*DBNull*.*Value*),  new *SqlParameter*("@addreass",model.SAddress != null?(object)model.SAddress:*DBNull*.*Value*),  new *SqlParameter*("@birityday",model.Sbirthday == null ?*DBNull*.*Value*:(object)model.Sbirthday),  new *SqlParameter*("@class",model.ClassModel.Cid),  };  SqlHelper sqlhelper = new MSSqlHelper("SIAS");  return sqlhelper.ExecuteNonQuery(sql, sp);  }  /// <summary>  /// 修改学生信息  /// </summary>  /// <param name="model"></param>  /// <returns></returns>  public int ModifyStuInfo(ElectronicInformation model)  {  string sql = "update T\_Student\_2013 set SNO = @no,Sname=@name,Sgender=@gender,SphoneNum=@phoneNum,SClass=@class,Sbirthday=@birthday,SAddress=@address where sid=@id";  T\_Class cm = new T\_Class();  *SqlParameter*[] sps = new *SqlParameter*[]  {  new *SqlParameter*("@no",model.SNO),  new *SqlParameter*("@name",model.Sname),  new *SqlParameter*("@gender",model.Sgender),  new *SqlParameter*("@phoneNum",model.SphoneNum),  new *SqlParameter*("@class",model.ClassModel.Cid),  new *SqlParameter*("@address",model.SAddress),  new *SqlParameter*("@birthday",model.Sbirthday),  new *SqlParameter*("@id",model.Sid),  };  SqlHelper sqlhelper = new MSSqlHelper("SIAS");  return sqlhelper.ExecuteNonQuery(sql, sps);  }  /// <summary>  /// 软删除  /// </summary>  /// <param name="sid"></param>  /// <returns></returns>  public int DeleteStu(int sid)  {  string sql = @"update T\_Student\_2013 set Squit=1 where Sid=@id";  *SqlParameter* sp = new *SqlParameter*("@id", sid);  SqlHelper sqlhelper = new MSSqlHelper("SIAS");  return sqlhelper.ExecuteNonQuery(sql, sp);  }  }  } |

模拟院系表

|  |
| --- |
| using *System*;  using *System*.*Collections*.*Generic*;  using *System*.*Linq*;  using *System*.*Text*;  using *System*.*Threading*.*Tasks*;  namespace Model  {  public class T\_Department  {  public int Did { get; set; }  public string Dinstitute { get; set; }  public int DPid { get; set; }  }  } |

模拟班级表

|  |
| --- |
| using *System*;  using *System*.*Collections*.*Generic*;  using *System*.*Linq*;  using *System*.*Text*;  using *System*.*Threading*.*Tasks*;  namespace Model  {  public class T\_Class  {  public int Cid { get; set; }  public string Cname { get; set; }  public int Cpid { get; set; }  }  } |

## Dal层

|  |
| --- |
| using *System*;  using *System*.*Collections*.*Generic*;  using *System*.*Linq*;  using *System*.*Text*;  using *System*.*Threading*.*Tasks*;  using HyfSqlHelper;  using Model;  using *System*.*Data*.*SqlClient*;  using *System*.*Data*;  namespace DAL  {  public class ElectronicInformationDal  {  /// <summary>  /// 分页查询学生信息  /// </summary>  /// <param name="instuitute">所在院系</param>  /// <param name="bindex">开始的索引</param>  /// <param name="eindex">结束的索引</param>  /// <returns>ElectronicInformation实体list集合</returns>  public *List*<ElectronicInformation> getStuInfoByInstitute(string instuitute, int bindex, int eindex)  {  *List*<ElectronicInformation> list = new *List*<ElectronicInformation>();  //分页查询：开窗函数  string sql = @"select \* from ( select T\_Student\_2013.\* ,Cname,Dinstitute ,row\_number() over (order by SNO ) as paging from T\_Student\_2013  join T\_Class\_2013 on T\_Student\_2013.SClass = T\_Class\_2013.Cid  join T\_Department on T\_Department.Did = T\_Class\_2013.Cpid) stuinfo where Dinstitute=@institute and squit=0 and paging between @bindex and @eindex ";  //参数集合  *SqlParameter*[] sp = new *SqlParameter*[] {  new *SqlParameter*("@institute", instuitute),  new *SqlParameter*("@bindex",bindex),  new *SqlParameter*("@eindex",eindex)  };  SqlHelper sqlhelper = new MSSqlHelper("SIAS");  *SqlDataReader* reader = sqlhelper.ExecuteReader(sql, sp);  //如果有数据  if (reader.*HasRows*)  {  while (reader.*Read*())  {  ElectronicInformation ei = new ElectronicInformation();  //获取的值赋予实体  ei.Sid = reader.*GetInt32*(reader.*GetOrdinal*("Sid"));  ei.SNO = reader.*GetString*(reader.*GetOrdinal*("SNO"));  ei.Sname = reader.*GetString*(reader.*GetOrdinal*("Sname"));  ei.Sgender = reader.*IsDBNull*(reader.*GetOrdinal*("Sgender")) ? null : reader.*GetString*(reader.*GetOrdinal*("Sgender"));  ei.SphoneNum = reader.*IsDBNull*(reader.*GetOrdinal*("SphoneNum")) ? null : reader.*GetString*(reader.*GetOrdinal*("SphoneNum"));  ei.SAddress = reader.*IsDBNull*(reader.*GetOrdinal*("SAddress")) ? null : reader.*GetString*(reader.*GetOrdinal*("SAddress"));  ei.Sbirthday = reader.*IsDBNull*(reader.*GetOrdinal*("SBirthday")) ? null : (*DateTime*?)reader.*GetDateTime*(reader.*GetOrdinal*("SBirthday"));  //模拟的外键关系的关联的实体  T\_Class classmodel = new T\_Class();  classmodel.Cname = reader.*GetString*(reader.*GetOrdinal*("Cname"));  ei.ClassModel = classmodel;  T\_Department deparmentModel = new T\_Department();  deparmentModel.Dinstitute = reader.*GetString*(reader.*GetOrdinal*("Dinstitute"));  ei.DepartmentModel = deparmentModel;  list.*Add*(ei);  }  }  return list;  }  /// <summary>  /// 查询表中所有院系  /// </summary>  /// <returns>院系实体集合</returns>  public *List*<T\_Department> getDepartment()  {  *List*<T\_Department> model = new *List*<T\_Department>();  string sql = "select Did,Dinstitute from T\_department ";  SqlHelper sqlhelper = new MSSqlHelper("SIAS");  *SqlDataReader* reader = sqlhelper.ExecuteReader(sql);  if (reader.*HasRows*)  {  while (reader.*Read*())  {  T\_Department dm = new T\_Department();  dm.Did = reader.*GetInt32*(0);  dm.Dinstitute = reader.*GetString*(1);  model.*Add*(dm);  }  }  return model;  }  /// <summary>  /// 查询某个院系的所有的班级  /// </summary>  /// <param name="institute">院系</param>  /// <returns>班级实体的集合</returns>  public *List*<T\_Class> getClass(string institute)  {  *List*<T\_Class> model = new *List*<T\_Class>();  string sql = @"select cid ,Cname from T\_Class\_2013 join T\_department on T\_department.did= T\_Class\_2013.cpid where Dinstitute= @institute ";  *SqlParameter* sp = new *SqlParameter*("@institute", institute);  SqlHelper sqlhelper = new MSSqlHelper("SIAS");  *SqlDataReader* reader = sqlhelper.ExecuteReader(sql, sp);  if (reader.*HasRows*)  {  while (reader.*Read*())  {  T\_Class cm = new T\_Class();  cm.Cid = reader.*GetInt32*(0);  cm.Cname = reader.*GetString*(1);  model.*Add*(cm);  }  }  return model;  }  /// <summary>  /// 查询T\_Student\_2013班级表中学生的总数  /// </summary>  /// <returns>object类型的总数</returns>  public object GetStuNums()  {  string sql = "select count(\*) from T\_Student\_2013";  SqlHelper sqlhelper = new MSSqlHelper("SIAS");  object obj = sqlhelper.ExecuteScalar(sql);  return obj;  }  /// <summary>  /// 添加学生信息到表中  /// </summary>  /// <param name="model">学生实体</param>  /// <returns></returns>  public int AddStuToTalbe(ElectronicInformation model)  {  //@gender,@phoneNum,@addreass,@birityday,  //,Sgender,SPhoneNum,SAddress,Sbirthday  string sql = @"insert into T\_Student\_2013(SNO,Sname,Sgender,SPhoneNum,SAddress,Sbirthday,SClass) values (@no,@name,@gender,@phoneNum,@addreass,@birityday,@class)";  *SqlParameter*[] sp = new *SqlParameter*[]  {  new *SqlParameter*("@no", model.SNO),  new *SqlParameter*("@gender",model.Sgender),  new *SqlParameter*("@name", model.Sname),  //三元运算符两个结果值需要能相互转换  new *SqlParameter*("@phoneNum",model.SphoneNum!=null?(*Object*)model.SphoneNum:*DBNull*.*Value*),  new *SqlParameter*("@addreass",model.SAddress != null?(object)model.SAddress:*DBNull*.*Value*),  new *SqlParameter*("@birityday",model.Sbirthday == null ?*DBNull*.*Value*:(object)model.Sbirthday),  new *SqlParameter*("@class",model.ClassModel.Cid),  };  SqlHelper sqlhelper = new MSSqlHelper("SIAS");  return sqlhelper.ExecuteNonQuery(sql, sp);  }  /// <summary>  /// 修改学生信息  /// </summary>  /// <param name="model"></param>  /// <returns></returns>  public int ModifyStuInfo(ElectronicInformation model)  {  string sql = "update T\_Student\_2013 set SNO = @no,Sname=@name,Sgender=@gender,SphoneNum=@phoneNum,SClass=@class,Sbirthday=@birthday,SAddress=@address where sid=@id";  T\_Class cm = new T\_Class();  *SqlParameter*[] sps = new *SqlParameter*[]  {  new *SqlParameter*("@no",model.SNO),  new *SqlParameter*("@name",model.Sname),  new *SqlParameter*("@gender",model.Sgender),  new *SqlParameter*("@phoneNum",model.SphoneNum),  new *SqlParameter*("@class",model.ClassModel.Cid),  new *SqlParameter*("@address",model.SAddress),  new *SqlParameter*("@birthday",model.Sbirthday),  new *SqlParameter*("@id",model.Sid),  };  SqlHelper sqlhelper = new MSSqlHelper("SIAS");  return sqlhelper.ExecuteNonQuery(sql, sps);  }  /// <summary>  /// 软删除  /// </summary>  /// <param name="sid"></param>  /// <returns></returns>  public int DeleteStu(int sid)  {  string sql = @"update T\_Student\_2013 set Squit=1 where Sid=@id";  *SqlParameter* sp = new *SqlParameter*("@id", sid);  SqlHelper sqlhelper = new MSSqlHelper("SIAS");  return sqlhelper.ExecuteNonQuery(sql, sp);  }  }  } |

## BLL层

|  |
| --- |
| using DAL;  using Model;  using *System*;  using *System*.*Collections*.*Generic*;  using *System*.*Linq*;  using *System*.*Text*;  using *System*.*Threading*.*Tasks*;  namespace BLL  {  //案例缺少数据有效性验证。这里可以用正则表达式等。------  public class ElectronicInformationBll  {  private *List*<ElectronicInformation> GetStuInfoByInstitute(string institute, int bindex, int eindex)  {  return new ElectronicInformationDal().getStuInfoByInstitute(institute, bindex, eindex);  }  /// <summary>  /// 分页重载  /// </summary>  /// <param name="institute"></param>  /// <param name="paging"></param>  /// <param name="nums"></param>  /// <returns></returns>  public *List*<ElectronicInformation> GetStuInfoByInstitute(string institute, int paging, string nums)  {  int bindex = (paging - 1) \* *Convert*.*ToInt32*(nums) + 1;  int eindex = paging \* *Convert*.*ToInt32*(nums);  return new ElectronicInformationDal().getStuInfoByInstitute(institute, bindex, eindex);  }  public *List*<T\_Department> getDepartment()  {  return new ElectronicInformationDal().getDepartment();  }  public *List*<T\_Class> getClass(string institute)  {  return new ElectronicInformationDal().getClass(institute);  }  public int MaxPaging(string ppnums)  {  ElectronicInformationDal dal = new ElectronicInformationDal();  int Snums = (int)dal.GetStuNums();  return (int)*Math*.*Ceiling*(Snums / *Convert*.*ToDouble*(ppnums));  }  public int AddStuToTalbe(ElectronicInformation model)  {  return new ElectronicInformationDal().AddStuToTalbe(model);  }  public int ModifyStuInfo(ElectronicInformation model)  {  return new ElectronicInformationDal().ModifyStuInfo(model);  }  public int DeleteStu(int sid)  {  return new ElectronicInformationDal().DeleteStu(sid);  }  }  } |

## UI层

|  |
| --- |
| using BLL;  using Model;  using *System*;  using *System*.*Collections*.*Generic*;  using *System*.*ComponentModel*;  using *System*.*Data*;  using *System*.*Drawing*;  using *System*.*Linq*;  using *System*.*Text*;  using *System*.*Threading*.*Tasks*;  using *System*.*Windows*.*Forms*;  namespace StuInfo  {  public partial class Form1 : *Form*  {  int paging = 1;  int maxPaging = 1;  public Form1()  {  InitializeComponent();  }  /// <summary>  /// 窗口加载时进行初始化  /// </summary>  /// <param name="sender"></param>  /// <param name="e"></param>  private void Form1\_Load(object sender, *EventArgs* e)  {  //性别默认选第一项  combo\_gender.*SelectedIndex* = 0;  //定义默认的每页显示的行数  combo\_Rows.*SelectedIndex* = 1;  //加载院系信息  LoadDeparment();  //默认选择第一个院系  combo\_instuitute.*SelectedIndex* = 0;  //加载班级信息  LoadClass(combo\_instuitute, combo\_class);  //加载学生信息到DataGridview  LoadDataGridView();  //dataGridView默认不选择  dataGridView1.*Rows*[0].*Cells*[0].*Selected* = false;  //班级默认选择第一项  combo\_class.*SelectedIndex* = 0;  // combo\_class.Text = "";  //根据每页显示的条数计算最大页数。  ElectronicInformationBll bll = new ElectronicInformationBll();  maxPaging = bll.MaxPaging(combo\_Rows.*SelectedItem*.*ToString*());  }  private void LoadClass(*ComboBox* institute, *ComboBox* combo)  {  //初始化  combo.*Text* = "";  //如果院系有选项  if (institute.*SelectedItem*.*ToString*() != "")  {  //根据院系选项加载该院系所欲的班级  ElectronicInformationBll bll = new ElectronicInformationBll();  *List*<T\_Class> cm = bll.getClass(institute.*SelectedItem*.*ToString*());  //显示的是对象的CName属性  combo.*DisplayMember* = "CName";  //值是对象的Cid属性  combo.*ValueMember* = "Cid";  //绑定对象到combox上  combo.*DataSource* = cm;  }  }  private void LoadDeparment()  {  //初始化  combo\_instuitute.*Items*.*Clear*();  txtMInstitute.*Items*.*Clear*();  //查询所有系  ElectronicInformationBll bll = new ElectronicInformationBll();  *List*<T\_Department> dmodel = bll.getDepartment();  //循环添加对象的属性到combox上。也可以用LoadClass方法中的绑定对象到combox上。  foreach (T\_Department item in dmodel)  {  combo\_instuitute.*Items*.*Add*(item.Dinstitute);  txtMInstitute.*Items*.*Add*(item.Dinstitute);  }  }  private void LoadDataGridView()  {  //查询电子信息工程学院所有学生信息  ElectronicInformationBll bll = new ElectronicInformationBll();  *List*<ElectronicInformation> model = bll.GetStuInfoByInstitute("电子信息工程学院", paging, combo\_Rows.*SelectedItem*.*ToString*());  //关闭自动产生列  dataGridView1.*AutoGenerateColumns* = false;  //把对象集合绑定到gridview上。  dataGridView1.*DataSource* = model;  }  /// <summary>  ///gridview格式化  /// </summary>  /// <param name="sender"></param>  /// <param name="e"></param>  private void dataGridView1\_CellFormatting(object sender, *DataGridViewCellFormattingEventArgs* e)  {  //如果第一列绑定的是T\_Department类型，显示其Dinstitute属性  if (e.*ColumnIndex* == 0 && e.*Value* != null && e.*Value* is T\_Department)  {  T\_Department modeld = e.*Value* as T\_Department;  if (modeld != null)  {  //e.value定义在GridView上显示的数据  e.*Value* = modeld.Dinstitute;  }  }  if (e.*ColumnIndex* == 1 && e.*Value* != null && e.*Value* is T\_Class)  {  T\_Class modelclass = e.*Value* as T\_Class;  if (modelclass != null)  {  e.*Value* = modelclass.Cname;  }  }  }  /// <summary>  /// 当院系combox选项改变时，调用LoadClass方法自动绑定班级combox上的数据。  /// </summary>  /// <param name="sender"></param>  /// <param name="e"></param>  private void combo\_instuitute\_SelectedIndexChanged(object sender, *EventArgs* e)  {  *ComboBox* combo = sender as *ComboBox*;  if (combo.*Name* == "combo\_instuitute")  {  LoadClass(combo\_instuitute, combo\_class);  }  else  {  LoadClass(txtMInstitute, txtMClass);  }  }  /// <summary>  /// 每页显示条目改变时，重新加载LoadDataGridView  /// </summary>  /// <param name="sender"></param>  /// <param name="e"></param>  private void combo\_Rows\_SelectedIndexChanged(object sender, *EventArgs* e)  {  LoadDataGridView();  }  /// <summary>  /// 下一页按钮  /// </summary>  /// <param name="sender"></param>  /// <param name="e"></param>  private void button4\_Click(object sender, *EventArgs* e)  {  paging++;  if (paging > maxPaging)  {  paging = maxPaging;  return;  }  //刷新gridview  LoadDataGridView();  }  private void button5\_Click(object sender, *EventArgs* e)  {  paging--;  if (paging <= 0)  {  paging = 1;  return;  }  LoadDataGridView();  }  private void button6\_Click(object sender, *EventArgs* e)  {  paging = 1;  LoadDataGridView();  }  private void button7\_Click(object sender, *EventArgs* e)  {  paging = maxPaging;  LoadDataGridView();  }  /// <summary>  /// 提交学生信息到数据库  /// </summary>  /// <param name="sender"></param>  /// <param name="e"></param>  private void button1\_Click(object sender, *EventArgs* e)  {  if (txt\_name.*Text*.*Trim*() == string.*Empty* || txt\_NO.*Text*.*Trim*() == string.*Empty*)  {  *MessageBox*.*Show*("带\*号的是必填项");  return;  }  //采集数据  ElectronicInformation model = new ElectronicInformation();  model.SNO = txt\_NO.*Text*;  model.Sname = txt\_name.*Text*;  model.Sgender = combo\_gender.*SelectedItem*.*ToString*();  model.SphoneNum = txt\_phone.*Text*.*Trim*().*Length* > 0 ? txt\_phone.*Text*.*Trim*() : null;  model.Sbirthday = dtp\_birth.*Value*;  model.SAddress = txt\_address.*Text*.*Trim*();  model.Sbirthday = dtp\_birth.*Value*;  T\_Class c = new T\_Class();  c.Cid = (int)combo\_class.*SelectedValue*;  model.ClassModel = c;  //搜集数据提交结果  ElectronicInformationBll bll = new ElectronicInformationBll();  int result = bll.AddStuToTalbe(model);  //提示信息  if (result > 0)  {  *MessageBox*.*Show*("新增成功");  }  }  /// <summary>  /// 当切换行时，显示信息到：修改学生信息区  /// </summary>  /// <param name="sender"></param>  /// <param name="e"></param>  private void dataGridView1\_RowEnter(object sender, *DataGridViewCellEventArgs* e)  {  T\_Department dmodel = dataGridView1.*Rows*[e.*RowIndex*].*Cells*[0].*Value* as T\_Department;  T\_Class Cmodel = dataGridView1.*Rows*[e.*RowIndex*].*Cells*[1].*Value* as T\_Class;  txtMInstitute.*Text* = dmodel.Dinstitute;  txtMClass.*Text* = Cmodel.Cname;  txtMNo.*Text* = dataGridView1.*Rows*[e.*RowIndex*].*Cells*[2].*Value*.*ToString*();  txtMName.*Text* = dataGridView1.*Rows*[e.*RowIndex*].*Cells*[3].*Value*.*ToString*();  //如果为空，则以空字符填充。  txtMGender.*SelectedItem* = dataGridView1.*Rows*[e.*RowIndex*].*Cells*[4].*Value* == null ? "" : dataGridView1.*Rows*[e.*RowIndex*].*Cells*[4].*Value*.*ToString*();  txtMPhone.*Text* = dataGridView1.*Rows*[e.*RowIndex*].*Cells*[5].*Value* == null ? "" : dataGridView1.*Rows*[e.*RowIndex*].*Cells*[5].*Value*.*ToString*();  txtMAddrewss.*Text* = dataGridView1.*Rows*[e.*RowIndex*].*Cells*[7].*Value* == null ? "" : dataGridView1.*Rows*[e.*RowIndex*].*Cells*[7].*Value*.*ToString*();  dtBirthday.*Text* = dataGridView1.*Rows*[e.*RowIndex*].*Cells*[6].*Value* == null ? "" : dataGridView1.*Rows*[e.*RowIndex*].*Cells*[6].*Value*.*ToString*();  lblMid.*Text* = dataGridView1.*Rows*[e.*RowIndex*].*Cells*[8].*Value*.*ToString*();  }  /// <summary>  /// 修改学生信息  /// </summary>  /// <param name="sender"></param>  /// <param name="e"></param>  private void button2\_Click(object sender, *EventArgs* e)  {  //1. 搜集数据  ElectronicInformation model = new ElectronicInformation();  model.Sid = *Convert*.*ToInt32*(lblMid.*Text*);  model.SNO = txtMNo.*Text*.*Trim*();  model.Sname = txtMName.*Text*.*Trim*();  model.Sgender = txtMGender.*SelectedItem*.*ToString*();  model.SAddress = txtMAddrewss.*Text*.*Trim*();  model.Sbirthday = dtBirthday.*Value*;  model.SphoneNum = txtMPhone.*Text*.*Trim*();  T\_Class cm1 = new T\_Class();  cm1.Cid = (int)txtMClass.*SelectedValue*;  model.ClassModel = cm1;  //2. 搜集数据提交返回结果  int result = new ElectronicInformationBll().ModifyStuInfo(model);  //3.提示信息  if (result > 0)  {  LoadDataGridView();  *MessageBox*.*Show*("修改成功");  }  }  /// <summary>  /// 软删除  /// </summary>  /// <param name="sender"></param>  /// <param name="e"></param>  private void button3\_Click(object sender, *EventArgs* e)  {  //根据sid进行删除  int sid = int.*Parse*(lblMid.*Text*);  //返回结果  ElectronicInformationBll bll = new ElectronicInformationBll();  int result = bll.DeleteStu(sid);  //提示  if (result > 0)  {  *MessageBox*.*Show*("退学申请完成");  LoadDataGridView();  }  }  }  } |

# 业务层与数据访问层解耦

抽象工厂用来解耦，业务层与数据访问层通过接口分离，改变数据库不会影响业务层

# Sprint.NET(表现层与业务层解耦)

Sprint.Net有两种方式：

* IOC（inversion of control）：控制翻转，就是创建对象的权利由开发人员自己控制new的对象，转到了由容器来控制。相当于抽象工厂。
* DI（Dependency Injection）：通过容器来创建工厂的时候，在对象初始化时可以给属性、构造方法的参数注入默认值。

Sprint.Net可以完成面向切片（AOP）编程。在mvc中通过过滤器实现。

## IOC

业务逻辑层代码如下：

|  |
| --- |
| using System;  using System.Collections.Generic;  using System.Linq;  using System.Text;  using System.Threading.Tasks;  namespace UserInfoService  {  public interface IUserInfoService  {  string ShowMsg();  }  public class Contacts  {  public string PhoneNum { get; set; }  public string Address { get; set; }  }  public partial class UserInfoService : IUserInfoService  {  public string Name { get; set; }  public int Age { get; set; }  public Contacts Contact { get; set; }  public UserInfoService(int age)  {  Age = age;  }  public string ShowMsg()  {  return "My Name is " + Name + "," + Age + "岁了。我来自于" + Contact.Address;  }  }  } |

在UI层中使用UserInfoService类的方式通常如下：

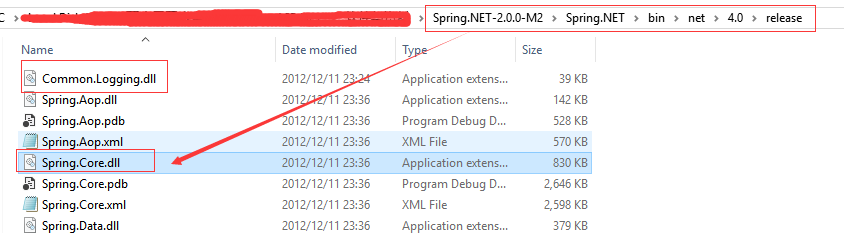
|  |
| --- |
| UserInfoService.UserInfoService useinfo = new UserInfoService.UserInfoService(); |

UI层与业务层是紧耦合，当UserInfoService类名称或命名空间改变后，需要在UI层进行修改。使用IOC可以实现解耦，命名空间改变后，只需修改配置文件即可。

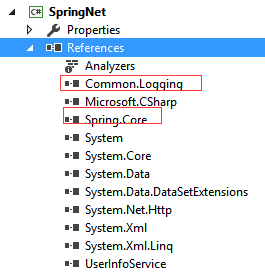
Spring.Net是一个可以完成IOC操作的框架。

### 引入程序集

在UI层引入程序集



结果



### 配置文件

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <configuration>  <configSections>  <sectionGroup name="spring">  <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>  <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />  </sectionGroup>  </configSections>  <spring>  <context>  <resource uri="config://spring/objects"/>  </context>  <objects xmlns="http://www.springframework.net">  <description>An example that demonstrates simple IoC features.</description>  <!--type中逗号左边是类的完整名称，右边是类的程序集的名称-->  <object name="UserInfoManager" type="UserInfoService.UserInfoService,UserInfoService">  <!--<constructor-arg name="name" value="hyf" />-->  <!--<constructor-arg index="0" value="hao"/>  <constructor-arg index="1" value="10"/>-->  <!--<constructor-arg index="0" value="hao"/>-->  <!--<constructor-arg type="String" value="hao"/>-->  <!--<constructor-arg type="int" value="10"/>-->  <property name="Name" value="hao"/>  <property name="Age" value="10"/>  <property name="Contact" ref="contact1" />  </object>  <object name="contact1" type="UserInfoService.Contacts,UserInfoService">  <property name="Address" value="zzsias"/>  </object>  </objects>  </spring>  </configuration> |

### 使用容器

在UI层中

|  |
| --- |
| static void Main(string[] args)  {  //1、创建容器对象  IApplicationContext container = ContextRegistry.GetContext();  //通过容器创建对象  IUserInfoService userInfo = container.GetObject<IUserInfoService>("UserInfoManager");  //调用对象的方法  string str = userInfo.ShowMsg();  Console.WriteLine(str);  Console.ReadKey();  } |

结果：



## DI

在8.1.2进行配置时，已经实现了依赖注入。有三种形式的注入。

### 属性注入

属性Name初始化为hao

|  |
| --- |
| <property name="Name" value="hao"/> |

### 构造器参数注入

构造方法的参数age接收值为18。 Index=”0”表示为构造方法的第一个参数。

|  |
| --- |
| <constructor-arg name="age" value="18" /> <!--等同于<constructor-arg index="0" value="18"/>--> |

### 复杂类型属性注入

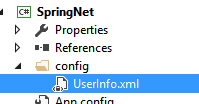
|  |
| --- |
| <property name="Contact" ref="contact1" />  </object>  <object name="contact1" type="UserInfoService.Contacts,UserInfoService">  <property name="Address" value="zzsias"/>  </object> |

属性Contact是复杂类型，用ref执行对象的名称，添加object节点对其名称、类型、属性进行设置。

## 配置分离

当对象特别多时，在配置文件中添加大量object节点，会降低文件的可读性，spring.net支持把object提取到独立的xml文件中。

第一步，创建config文件夹，并创建xml文件。如下图：



第二步，把object移到UserInfo.xml中

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <!--type中逗号左边是类的完整名称，右边是类的程序集的名称-->  <objects xmlns="http://www.springframework.net">  <object name="UserInfoManager" type="UserInfoService.UserInfoService,UserInfoService">  <constructor-arg name="age" value="18" />  <!--<constructor-arg index="0" value="18"/>-->  <property name="Name" value="hao"/>  <property name="Contact" ref="contact1" />  </object>  <object name="contact1" type="UserInfoService.Contacts,UserInfoService">  <property name="Address" value="zzsias"/>  </object>  </objects> |

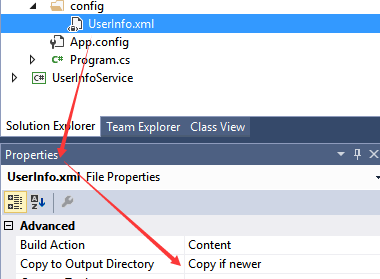
第三步，修改配置文件

|  |
| --- |
| <spring>  <context>  <resource uri="file://~/config/UserInfo.xml"/>  </context>  </spring> |

配置文件清晰明了：

|  |
| --- |
| <?xml version="1.0" encoding="utf-8" ?>  <configuration>  <configSections>  <sectionGroup name="spring">  <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>  <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />  </sectionGroup>  </configSections>  <spring>  <context>  <resource uri="file://~/config/UserInfo.xml"/>  </context>  </spring>  </configuration> |

第四步：xml文件复制到输出目录



# Sprint.NET面向切面编程

面向切面编程(也叫面向方面)：Aspect Oriented Programming(AOP),是目前软件开发中的一个热点，也是Spring框架中的一个重要内容。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

AOP是OOP的延续，是（Aspect Oriented Programming）的缩写，意思是面向切面（方面）编程。

主要的功能是：日志记录，性能统计，安全控制，事务处理，异常处理等等。  
　　主要的意图是：将日志记录，性能统计，安全控制，事务处理，异常处理等代码从业务逻辑代码中划分出来，通过对这些行为的分离，我们希望可以将它们独立到非指导业务逻辑的方法中，进而改  变这些行为的时候不影响业务逻辑的代码。

可以通过预编译方式和运行期动态代理实现在不修改源代码的情况下给程序动态统一添加功能的一种技术。AOP实际是GoF设计模式的延续，设计模式孜孜不倦追求的是调用者和被调用者之间的解耦，AOP可以说也是这种目标的一种实现。

假设把应用程序想成一个立体结构的话，OOP的利刃是纵向切入系统，把系统划分为很多个模块（如：用户模块，文章模块等等），而AOP的利刃是横向切入系统，提取各个模块可能都要重复操作的部分（如：权限检查，日志记录等等）。由此可见，AOP是OOP的一个有效补充。

注意：AOP不是一种技术，实际上是编程思想。凡是符合AOP思想的技术，都可以看成是AOP的实现。

## 什么是切面编程

在考虑对象及对象与其他对象的关系时，我们通常会想到继承这个术语。例如，定义某一个抽象类 Dog 类。在标识相似的一些类但每个类又有各自的独特行为时，通常使用继承来扩展功能。举例来说，如果标识了 Poodle，则可以说一个 Poodle 是一个 Dog，即 Poodle 继承了 Dog。到此为止都似乎不错，但是如果定义另一个以后标识为 Obedient Dog 的独特行为又会怎样呢？当然，不是所有的 Dogs 都很驯服，所以 Dog 类不能包含 obedience 行为。此外，如果要创建从 Dog 继承的 Obedient Dog 类，那么 Poodle 放在这个层次结构中的哪个位置合适呢？Poodle 是一个 Dog，但是 Poodle 不一定 obedient；那么 Poodle 是继承于 Dog 还是 Obedient Dog 呢？都不是，我们可以将驯服看作一个方面，将其应用到任何一类驯服的 Dog，我们反对以不恰当的方式强制将该行为放在 Dog 层次结构中。

## 与OOP面向对象编程的区别

AOP、OOP在字面上虽然非常类似，但却是面向不同领域的两种设计思想。OOP（面向对象编程）针对业务处理过程的实体及其属性和行为进行抽象封装，以获得更加清晰高效的逻辑单元划分。  
　　而AOP则是针对业务处理过程中的切面进行提取，它所面对的是处理过程中的某个步骤或阶段，以获得逻辑过程中各部分之间低耦合性的隔离效果。这两种设计思想在目标上有着本质的差异。  
　　上面的陈述可能过于理论化，举个简单的例子，对于“雇员”这样一个业务实体进行封装，自然是OOP/OOD的任务，我们可以为其建立一个“Employee”类，并将“雇员”相关的属性和行为封装其中。而用AOP设计思想对“雇员”进行封装将无从谈起。  
　　同样，对于“权限检查”这一动作片断进行划分，则是AOP的目标领域。而通过OOD/OOP对一个动作进行封装，则有点不伦不类。

换而言之，OOD/OOP面向名词领域，AOP面向动词领域。

## AOP 的基本概念

在面向对象编程中，类，对象，封装，继承，多态等概念是描述面向对象思想主要术语。与此类似，在面向方面编程中，同样存在着一些基本概念：

联结点（JointPoint） ：一个联结程序执行过程中的一个特定点。典型的联结点有：调用一个方法；方法执行这个过程本身；类初始化；对象初始化等。联结点是 AOP 的核心概念之一，它用来定义在程序的哪里通过 AOP 加入新的逻辑。

切入点（Pointcut） ：一个切入点是用来定义某一个通知该何时执行的一组联结点。通过定义切入点，我们可以精确地控制程序中什么组件接到什么通知。上面我们提到，一个典型的联结点是方法调用，而一个典型的切入点就是对某一个类的所在方法调用的集合。通常我们会通过组建复杂的切入点来控制通知什么时候被执行。

通知（Advice） ：在某一个特定的联结点处运行的代码称为“通知”。通知有很多种，比如

在联结点之前执行的前置通知（before advice）和在联结点之后执行的后置通知（after advice） 。

方面（Aspect） ：通知和切入点的组合叫做方面，所以，方面定义了一段程序中应该包括的逻辑，以及何时应该执行该逻辑。

织入（Weaving） ：织入是将方面真正加入程序代码的过程。对于静态 AOP 方案而言，织入是在编译时完成的，通常是在编译过程中增加一个步骤。类似的，动态 AOP 方案则是在程序运行是动态织入的。

目标（Target） ：如果一个对象的执行过程受到某一个 AOP 的修改，那么它就叫一个目标对象。目标对象通常也称为被通知对象。

引入（Introduction） ： 通过引入，可以在一个对象中加入新的方法或属性，以改变它的结构，这样即使该对象的类没有实现某一个接口，也可以修改它，使之成为该接口的一个实现。

静态和动态：静态 AOP 和动态 AOP 两者之间的区别主要在于什么时间织入，以及如何织入。最早的 AOP 实现大多都是静态的。在静态 AOP 中，织入是编译过程的一个步骤。用Java 的术语说，静态 AOP 通过直接对字节码进行操作，包括修改代码和扩展类，来完成织入过程。显然，这种办法生成的程序性能很好，因为最后的结果就是普通的 Java 字节码，在运行时不再需要特别的技巧来确定什么时候应该执行通知。这种方法的缺点是，如果想对方面做什么修改，即使只是加入一个新的联结点，都必须重新编译整个程序。AspectJ 是静态 AOP 的一个典型例子。与静态 AOP 不同，动态 AOP 中织入是在运行时动态完成的。织入具体是如何完成的，各个实现有所不同。Spring AOP 采取的方法是建立代理，然后代理在适当的时候执行通知。动态 AOP 的一个弱点就在于，其性能一般不如静态 AOP。而动态AOP 的主要优点在于可以随时修改程序的所有方面，而不需重新编译目标。

### 横切技术

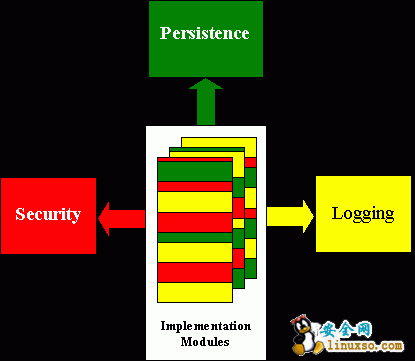
 “横切”是AOP的专有名词。它是一种蕴含强大力量的相对简单的设计和编程技术，尤其是用于建立松散耦合的、可扩展的企业系统时。横切技术可以使得AOP在一个给定的编程模型中穿越既定的职责部分（比如日志记录和性能优化）的操作。  
        如果不使用横切技术，软件开发是怎样的情形呢？在传统的程序中，由于横切行为的实现是分散的，开发人员很难对这些行为进行逻辑上的实现或更改。例如，用于日志记录的代码和主要用于其它职责的代码缠绕在一起。根据所解决的问题的复杂程度和作用域的不同，所引起的混乱可大可小。更改一个应用程序的日志记录策略可能涉及数百次编辑——即使可行，这也是个令人头疼的任务。  
       在AOP中，我们将这些具有公共逻辑的，与其他模块的核心逻辑纠缠在一起的行为称为“横切关注点（Crosscutting Concern）”，因为它跨越了给定编程模型中的典型职责界限。

### 横切关注点

一个关注点（concern）就是一个特定的目的，一块我们感兴趣的区域，一段我们需要的逻辑行为。从技术的角度来说，一个典型的软件系统包含一些核心的关注点和系统级的关注点。举个例子来说，一个信用卡处理系统的核心关注点是借贷/存入处理，而系统级的关注点则是日志、事务完整性、授权、安全及性能问题等，许多关注点——即横切关注点（crosscutting concerns）——会在多个模块中出现。如果使用现有的编程方法，横切关注点会横越多个模块，结果是使系统难以设计、理解、实现和演进。AOP能够比上述方法更好地分离系统关注点，从而提供模块化的横切关注点。

例如一个复杂的系统，它由许多关注点组合实现，如业务逻辑、性能，数据存储、日志和调度信息、授权、安全、线程、错误检查等，还有开发过程中的关注点，如易懂、易维护、易追查、易扩展等，

 1 .**由不同模块实现的一批关注点组成一个系统，即把模块作为一批关注点来实现**，如图：

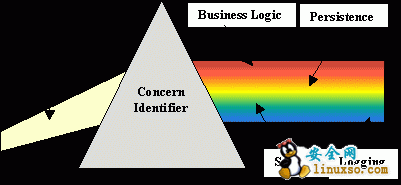


 通过对系统需求和实现的识别，我们可以将模块中的这些关注点分为：核心关注点和横切关注点。对于核心关注点而言，通常来说，实现这些关注点的模块是相互独立的，他们分别完成了系统需要的商业逻辑，这些逻辑与具体的业务需求有关。而对于日志、安全、持久化等关注点而言，他们却是商业逻辑模块所共同需要的，这些逻辑分布于核心关注点的各处。在AOP中，诸如这些模块，都称为横切关注点。应用AOP的横切技术，关键就是要实现对关注点的识别。

**2 .识别关注点**

如果将整个模块比喻为一个圆柱体，那么关注点识别过程可以用三棱镜法则来形容，穿越三棱镜的光束（指需求），照射到圆柱体各处，获得不同颜色的光束，最后识别出不同的关注点。

**1 ). 关注点识别：三棱镜法则，如图所示：**

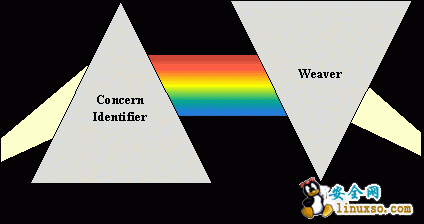


      上图识别出来的关注点中，Business Logic属于核心关注点，它会调用到Security，Logging，Persistence等横切关注点。

1. **public** **class** BusinessLogic {
2. **public** **void** SomeOperation() {
3. //验证安全性；Securtity关注点；
4. //执行前记录日志；Logging关注点；
5. DoSomething();
6. //保存逻辑运算后的数据；Persistence关注点；
7. //执行结束记录日志；Logging关注点；
8. }
9. }

**3. 将横切关注点织入到核心关注点中**

        AOP的目的，就是要将诸如Logging之类的横切关注点从BusinessLogic类中分离出来。利用AOP技术，可以对相关的横切关注点封装，形成单独的“aspect”。这就保证了横切关注点的复用。由于BusinessLogic类中不再包含横切关注点的逻辑代码，为达到调用横切关注点的目的，可以利用横切技术，截取BusinessLogic类中相关方法的消息，例如SomeOperation()方法，然后将这些“aspect”织入到该方法中。将横切关注点织入到核心关注点中,如图：



       通过利用AOP技术，改变了整个系统的设计方式。在分析系统需求之初，利用AOP的思想，分离出核心关注点和横切关注点。在实现了诸如日志、事务管理、权限控制等横切关注点的通用逻辑后，开发人员就可以专注于核心关注点，将精力投入到解决企业的商业逻辑上来。同时，这些封装好了的横切关注点提供的功能，可以最大限度地复用于商业逻辑的各个部分，既不需要开发人员作特殊的编码，也不会因为修改横切关注点的功能而影响具体的业务功能。

参看：<http://blog.csdn.net/hguisu/article/details/7586704>

## 实践

比如有个继承了接口ITestInterface的AopTest类，该类有2个方法，一个构造方法，一个BeingTest方法。

|  |
| --- |
| public interface ITestInterface  {  void BeingTest();  }  public class AopTest : ITestInterface  {  public AopTest()  {  Console.WriteLine("--------AopTest对象被创建");  }  public void BeingTest()  {  System.Console.WriteLine("-----------------BeingTest执行，开始测试-----------------");  }  } |

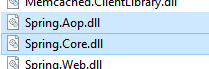
在面向对象编程中，我们一般用下面的方式调用：

|  |
| --- |
| ITestInterface t1 = new AopTest ();  t1.aopTest(); |

但是要扩展AopTest的功能时，我们就不得不修改接口与AopTest类，最后还需要重新编译。而aop可以在不修改原来代码的基础上实现功能的扩展。

### 引入程序集

引入spring.Aop、Spring.Core程序集



### 创建通知

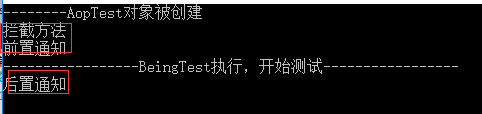
创建一个Advice.cs文件，里面实现了3种通知：

|  |
| --- |
| public class BeforeAdvice : Spring.Aop.IMethodBeforeAdvice  {  public void Before(MethodInfo method, object[] args, object target)  {  Console.WriteLine("前置通知");  }  }  public class AfterReturningAdvice : Spring.Aop.IAfterReturningAdvice  {  public void AfterReturning(object returnValue, MethodInfo method, object[] args, object target)  {  Console.WriteLine("后置通知"); ;  }  }  public class Interceptor : IMethodInterceptor  {  public object Invoke(IMethodInvocation invocation)  {  Console.WriteLine("拦截方法");  invocation.Proceed();  return "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*";  }  } |

### 通过代理工厂创建对象

|  |
| --- |
| static void Main(string[] args)  {  //创建代理工厂，构造方法将执行  ProxyFactory factory = new ProxyFactory(new AopTest());  //添加通知  factory.AddAdvice(new Interceptor());//拦截  factory.AddAdvice(new BeforeAdvice());//前置  factory.AddAdvice(new AfterReturningAdvice());//后置  ITestInterface proxy = factory.GetProxy() as ITestInterface;//创建ITestInterface的代理  //调用  proxy.BeingTest();//代理对象负责调用方法  Console.ReadKey();  } |

执行结果：



在不修改AopTest的环境下，AopTest功能获得了扩展。