**[从抽象谈起（一）：工厂模式与策略模式](http://www.cnblogs.com/mad/p/3160557.html)**

抽象的意思是，抽取不同事物的共性而成的一种新事物。为什么用事物一词？因为抽象未必抽的是物，也可能是事。  
抽象是编程的重要思想之一，其主要目的是为了减少代码重复，使其更易维护。  
抽象就是让变化的事物得到一致的处理方式。

**抽象是如何应用的？我们怎么去抽象？**

当我们面临有共同特性的事物时，需要对它们统一处理，那么就需要抽象。而这种共性的事物在实际项目中会经常碰到。而且在我们使用的各种框架中应用广泛。比如说，用户打开不同的网页，都需要去展现页面，那么所有的网页都有一个共性就是展现，而不同的网页又具有不同的行为；所以在处理网页展现时，只需要处理网页们的抽象的东西——展现。这个“处理网页展现”的代码一般在框架内部实现。他对所有的网页处理都是调用抽象网页的展现代码，所以他的代码是一致不变的。再比如说我们点击某一些按钮，会触发各种事件，点击按钮的行为都是一致的，而事件的内容缺各不相同。那么在点击的这个行为上的处理也是一致的，就是触发事件的内容，至于事件内容的本身，那就是具体的实现问题，跟处理点击没有关系。我们把统一处理抽象事物的代码叫上层代码。抽象就是为了上层代码的一致性，不需要因为具体事物的改变而改变。

**抽象与模式**

也许大家都知道设计模式，这是经典的实际应用中碰到的各种常见问题而归纳出来的编程技巧，其中大多数都离不开抽象这一概念。掌握的抽象的思想，再去理解他们更容易些。

**工厂模式**  
工厂模式是最易理解的模式之一，他是通过一个工厂类，创建抽象对象（其实是具体的实际对象），因为是抽象对象，所以其他代码在使用这些抽象对象的共性时只需要通过工厂类获取对象即可，而不需要具体new每个实际对象。  
代码示例：

|  |
| --- |
| public interface IUserRepository  {      IEnumerable<Users> GetUsers();  }    namespace DataRepository.MySql  {      public class UserRepository : IUserRepository      {          private MySql \_db;          public UserRepository()          {              \_db = new MySql(ConnectionStringManager.Get("MySql"));          }            public IEnumerable<Users> GetUsers()          {              return \_db.ExecuteSql("sql").ToList<Users>();          }      }  }    namespace DataRepository.SqlServer  {      public class UserRepository : IUserRepository      {          public IEnumerable<Users> GetUsers()          {              return GetDataContext(ConnectionStringManager.Get("SqlServer")).Users.AsEnumerable();          }      }  }  public class RepositoryFactory  {      public IUserRepository GetUserRepository()      {          return CreateInsnace(AppSettings.Get("CurrentDatabase"),"UserRepository");      }  }    public class UserManager()  {      public IEnumerable<Users> GetUsers()      {          return RepositoryFactory.GetUserRepository().GetUsers();      }  } |

上面的代码看起来比较简单，继承自接口，反射实例化具体子类便可。但抽象意味着是具体实现，而不是继承，继承只是实现的一种。所以在继承上要慎用。而大多数的模式也都是采用各种组合。其不外乎就是抽象出共同的接口，组合接口的实现。

**策略模式**  
策略模式的应用场景我们几乎都碰到过，比如现在都比较流行SinaWeibo登录和QQ登录，再加上自己的Email登录，每种登录方式都有不同的实现，因为Sina和QQ这种OAuth的登录都需要回调网页(其实就是用来验证用户有效性的)，而我们的Email登录也需要验证，总不能写3个登录验证页面吧？~~当然，写三个也不是不可以，但是如果我们还是Wap站，那就是6个，如果再有其他的站点，那就不知道要写多少个了。至少页面数量会很多。如果用MVC框架的话，倒是可以用公共的Controller来省去，那至少也要三个Action，而这三个页面其实也有共性，必然存在代码重复。我们需要消除这种重复，万一哪天再来个开心、人人等登录实现，又会增加很多页面；~~所以要把因为变化牵扯出来的变化保持不变。

可以看下面的实现代码，实现了3种登录方式的登录、验证和退出。

|  |
| --- |
| //登录地址策略接口  public interface ILogin  {      public string GetLoginUrl(HttpContextBase context);  }  //第三方用户的验证策略接口  public interface IAuthenticate  {      public TrdUser Authenticate(HttpContextBase context);  }  //退出地址策略接口  public interface ILogout  {      public string GetLogoutUrl(HttpContextBase context);  }  //新浪Auth的策略实现  public class SinaAuth : ILogin,IAuthenticate,ILogout  {      public string GetLoginUrl(HttpContextBase context)      {          return "<http://weibo.com/oauth/login>";      }        public TrdUser Authenticate(HttpContextBase context)      {          return new TrdUser { Name = "我来自新浪" };      }        public string GetLogoutUrl(HttpContextBase context)      {          return "<http://weibo.com/oauth/logout>";      }  }  //QQAuth的策略实现  public class QQAuth : ILogin,IAuthenticate  {      public string GetLoginUrl(HttpContextBase context)      {          return "<http://qq.com/oauth/login>";      }        public TrdUser Authenticate(HttpContextBase context)      {          return new TrdUser { Name = "我来自QQ" };      }  }  //本网站默认策略的实现  public class DefaultAuth :ILogin, IAuthenticate,ILogout  {      public string GetLoginUrl(HttpContextBase context)      {          return "<http://mysite.com/oauth/login>";      }        public TrdUser Authenticate(HttpContextBase context)      {          return new TrdUser { Name = "我来自Email" };      }        public string GetLogoutUrl(HttpContextBase context)      {          return "<http://mysite.com/oauth/logout>";      }  }  //策略组装类  public class Login  {      private HttpContextBase \_context;      public Login(HttpContextBase context)      {          \_context = context;      }      //获取登录地址      public string GetLoginUrl(ILogin login)      {          return login.GetLoginUrl(\_context);      }      //获取退出地址      public string GetLogoutUrl(ILogout logout)      {          return logout.GetLogoutUrl(\_context);      }      //获取本站用户信息      public User Authenticate(IAuthenticate auth)      {          var trdUser = auth.Authenticate(\_context);          return xx.GetUser(trdUser);      }  }  //策略工厂，根据不同的type创建不同的策略  public class AuthFactory  {      private string \_authType;      public AuthFactory(string authType)      {          \_authType = authType;      }        public ILogin GetLogin()      {          return (ILogin)CreateInstance(typeof(ILogin));      }        public IAuthenticate GetAuth()      {          return (IAuthenticate)CreateInstance(typeof(IAuthenticate));      }        public ILogout GetLogoutUrl()      {          return (ILogout)CreateInstance(typeof(ILogout));      }        private object CreateInstance(Type type)      {          //通过反射获取具体的类型          var instance = ....;          //若是没有实现，就使用默认的。          if(instance == null) return new DefaultAuth();      }  }    //本站的页面具体代码  public AccountController : Controller  {      //登录地址      public ActionResult Login(string authType)      {          var login = new Login(HttpContext);          var factory = new AuthFactory(authType);            return Redirect(login.GetLoginUrl(factory.GetLogin()));      }        //验证地址，不管是Email还是第三方登录回调，均使用该地址。      public ActionResult Authenticate(string authType)      {          var login = new Login(HttpContext);          var factory = new AuthFactory(authType);            var user = login.Authenticate(factory.GetAuth());          //....      }        //退出地址      public ActionResult Logout(string authType)      {          var login = new Login(HttpContext);          var factory = new AuthFactory(authType);            return Redirect(login.GetLogoutUrl(factory.GetLogin()));      }  } |

因为我们在获取具体策略的时候依然要判断该使用哪种策略，所以用工厂模式来创建具体的策略。但是仍然能看到Controller代码的丑陋之处，这就需要AOP的实现来避免重复的代码，后面我们会讲到。

观察者模式又叫发布订阅模式，有订阅者和发布者；发布者可以包含了多个订阅者订阅的事件，一旦发布者执行，会执行所有的订阅者订阅的事件。我觉得这么讲还是很迷糊。其实就是说“发布者”是一段上层代码，他知道他所需要执行的过程中会发生一些事情，而这些事情具体逻辑自己又不知道，就算知道所有的逻辑，要用条件分支判断执行，这总归的是不好的，所以才有了这个模式。这是一个非常棒的模式。他使得发布者的代码保持不变。而订阅者的事件可以散步在他们自己的代码中。

我们实际应用中最常见的就是页面中的按钮点击事件。当我们双击webform中的按钮后会自动生成一个btn\_OnClick的方法，然后在里面编写一些逻辑，同时也生成了btn.Click+= new EventHandler(btn\_OnClick)代码(只是2.0之后这个代码就被隐藏起来了)，这就是给按钮btn(订阅者)订阅了一个事件。这些逻辑理当属于按钮所在的页面，而不是需要执行这个方法的代码中。

当按钮点击之后，会触发页面的提交，webform框架可以获取是哪个按钮被点击过，然后执行btn.Click()，就可以执行我们具体的逻辑了。 设想如果不用这个模式，按钮的Click方法是不是要写很多switch来判断是哪个按钮，然后调用该有的逻辑。

那么假如说我们不用.NET的这套事件机制，该如何漂亮的抽象出Click的代码呢？

其实只要相当上一节的策略模式，我们只要给Click接受一个IClickEvent接口，然后button类再包含一组IClickEvent的成员，就可以遍历这些成员执行了。订阅的代码就变成了button.AddEvent(new xxClickEvent());即可。

在.net中，我们没必要用IClickEvent接口的形式，因为我们有委托这个方法代理（或者叫方法指针），他可以说是一个只具有一个方法的接口，而.net中的事件本身也是委托。只是事件形式的委托是封闭的，不可在外部直接赋值操作，只能订阅和删除订阅。

综上来看，观察者模式不过是一个处理未知方法的模式，他漂亮的把具体逻辑分散到他该属于地方。

在web前端的Javascript中，这种情况就更为普遍。比如我们用jQuery时，给一个按钮增加一个onClick方法，只需要$(“#btn”).click(function(){})即可。浏览器会知道具体的哪个按钮被点击，甚至我们随便点击页面的一个地方，都会被浏览器截获，假如我们有相应的OnClick方法，他会执行调用，并传值给我们当前鼠标的位置等。 在我们发起一个ajax请求时，会有一个参数是callback方法，在判断完XmlHttpRequest的readyState == 4后调用，每个ajax的请求完后的callback都不一致。所以说他也是观察者模式。我们说这种叫做回调模式是不是更好？所谓的“回”就是使用之前的代码，而不是当前的代码。

**所以说委托或者回调(方法指针)也是一种抽象，在不具备这种能力的语言里可用接口来代替。重复上节的话，抽象提取变化事物的共性，不管是面向对象还是过程式还是函数式，都离不开抽象的思想。**

说了这么多，不知道表达清楚没有，我们来讲一个实际应用不依赖于框架的。

比如我们发布一篇文章，常用逻辑就是保存文章。如果哪天来了新的需求，比如说跟某某公司合作，发表完文章之后需要给用户增加一些奖励。又过了几天又来一个新的需求，所最近抓的紧，需要对文章审核，一旦有违禁的关键词不允许发布。这两个逻辑之前都不存在。我们是不是要修改代码呢？这两个需求都是临时的。假如修改的保存逻辑，回头合作取消和风头过后又要取消掉。这太不合适了。我们应让代码的修改的范围缩到最小。

如果我们在保存逻辑前后增加事件，比如PreSave和EndSave，然后对应增加相关的订阅代码，就不需要修改Save的逻辑。但是调用Save的代码依然需要增加两个方法和订阅代码。

|  |
| --- |
| public class ArticleController : Controller  {      public ActionResult Save(Article article)      {          var articleManager = new ArticleManager();          articleManager.PreSaveEvent +=  x =>          {              var shouldBlock = BlockWords.Filter(article);              if(shouldBlock)              {                  throw new SecretExcepetion("对不起，您的文章中包含违禁关键词，请检查");                  //封杀用户              }          };            articleManager.EndSaveEvent += x=>          {              //奖励用户          }            articleManager.Save(article);      }  }    public class ArticleManager  {      public event Action<Article> PreSaveEvent;      public event Action<Article> EndSaveEvent;        public void Save(Article article)      {          if(PreSaveEvent!=null)          {              PreSaveEvent(article);          }          var db = new ArticleRepository();          db.Save(article);          if(EndSaveEvent!=null)          {              EndSaveEvent(article);          }      }  }    public class ArticleRepository  {      public void Save(Article article)      {          database.Save(article);      }  } |

虽然上面的用的是事件的方式，如果用委托作为Save的参数也可以，只是作为参数对以后的重构可能会带来麻烦。

这样只需要修改Controller的代码就能改变这些需求，如果不连Controller的代码也不想改怎么办呢？后面再说吧。

【转载请注明出处】

AOP（Aspect oriented programming）面向切面编程。说成切面不容易理解，代码哪里有切面？又不是三维物体。概念不管，我们从其思想来理解这个名词吧。 AOP的主要思想是把相同、相似的并且零散的逻辑抽离出来，统一处理；这样不仅维护起来方便，也让代码更加关注自己本身，清晰明了。

比如我们常见的权限检查、日志记录、异常处理等都是散乱在系统各个地方，比如发表一篇文章的代码：

1. **public** **void** Post(Article article)
2. {
3. **if**(currentUser **is** **null**)
4. **throw** **new** AuthException("您还没有登录");
5. **else**
6. ArticleManager.Save(article);
7. }

本来一句话ArticleManager.Save就能搞定的事情，现在要加上if else 还要处理异常，代码显得异常难看也难以维护。假如换成

1. [Authorize]
2. **public** **void** Post(Article article)
3. {
4. ArticleManager.Save(article);
5. }

用AuthroizeAttribute来处理权限问题，这样代码清晰很多，而且可以复用这个Attribute，这么好的思想就是AOP思想。  
当然Attribute只是一种实现方式，Attribute也是调用Post方法前，通过反射得到Attribute，然后执行其代码，  
平时我们用的最多的AOP就在ASP.NET MVC框架里，这个AuthorizeAttribute就是MVC自带的。我们可以重写他的一些方法达到自己想要的功能（比如权限等级等）。

再举一个处理异常的例子，如果我们有一个统一的处理异常的逻辑，那么就可以在逻辑代码里不用try/catch，而是直接throw exception，这会让代码更加整洁。  
（PS：比不是说有了统一处理就再也不用try/catch了，有些异常该吃掉的还是要吃掉，这要看具体的业务需求。另外捕获不到的异常比如线程里的要注意catch。）  
比如例子：

1. **public** **void** Post(Article article)
2. {
3. **if**(String.IsNullOrEmpty(article.Title))
4. {
5. **throw** **new** ArgumentMissException("title");
6. }
7. ...
8. ArticleManager.Save(article);
9. }
11. **protected** **override** **void** OnException(ExceptionContext filterContext)
12. {
13. //..异常处理代码
14. }

而只要重写MVC提供好的OnException方法就能处理所有的异常，或者是展现给用户友好的错误界面，或者是发送异常日志都很方便，而不用在每个地方都写处理的代码。  
延伸考虑一下，如果统一处理异常信息，那么对异常的善后处理的把握就要看异常类的设计和运用了。

另外，再看上面的代码，如果有多个地方需要Post(Article)，但又完全不一致，可是字段检查是一致的，那就会造成字段检查的代码重复，这种情况要么用一个单独的方法来验证Article的有效性，然后各个地方调用，要么就用MVC提供的ModelBinder，这也是通过参数的Attribute来实现的AOP方法；MVC考虑的太周到了，不得不赞。

1. **public** **void** Post([ArticleBinder]Article article)
2. {
3. //这里就不需要再写关于Article的验证代码了
4. ArticleManager.Save(article);
5. }
7. **public** **class** ArticleBinderAttribute : CustomModelBinderAttribute
8. {
9. **class** ArticleBinder : IModelBinder
10. {
11. **public** **object** BindModel(ControllerContext controllerContext, ModelBindingContext bindingContext)
12. {
13. //虽然自定义了binder很惬意，但这里获取参数值时就没有用Action的参数那么舒服了。
14. var title = controllerContext.RouteData.Values["title"].ToString();
15. **if** (**string**.IsNullOrEmpty(title))
16. {
17. **throw** **new** ArgumentNullException("title");
18. }
19. **return** **new** Article
20. {
21. Title = title
22. };
23. }
24. }
26. **public** **override** System.Web.Mvc.IModelBinder GetBinder()
27. {
28. **return** **new** ArticleBinder();
29. }
30. }

运用框架提供的AOP很简单也很惬意，但我们在用别人的框架时不能只做应用级程序员，我们要领会其思想，掌握其本质，明白其实现。  
所幸，MVC是开源的，我们可以看他的源代码，关注提供AOP的地方。

代码的设计或者说框架的设计，总是想让代码写起来干净利落，松耦合，不拖泥带水。  
所以清晰的设计就是从程序的开始，一步步规划其运行步骤，并在适当的地方提供一些供用户处理方法。框架就是把用户框在自己设定的圈子里，但又尽可能的给用户自由。  
我们看MVC的大致流程，  
1、从输入Url回车那一刻起，先是通过UrlRouting路由，来判断用户访问哪个Controller/Action;  
2、通过controllerfactory获取具体的controller实例，用户可自定义factory。  
3、通过actioninvoke调用action，在调用之前需要先获取action的filter，这些filter就是AOP的拦截器。用户都可以自定义各种filter(就是自定义的attribute)。  
4、根据执行顺序执行action和filter，onactionexecuting、onactionexecuted等。  
5、找到action对应的view，用户可自定义viewenginer。  
6、呈现页面结束。

当出错的时候invoke的catch调用onexception的用户的实现。

我本想罗列一些MVC的源码，但感觉没必要，有心人自己去下载看吧。如果不清楚的可以加群交流。

就是这样简单，AOP的思想是想把和当前逻辑不相干的代码抽离的一种实现，我们如果追求代码的美感，就会更加在意整体代码的设计，AOP通常用于一个系统的外围搭建处。  
只有我们把架子、外围都搭建的漂亮，代码写起来才更美。