通过采用MVC模式，我们可以将可视化UI元素的呈现、UI处理逻辑和业务逻辑分别定义在View、Controller和Model中，但是对于三者之间的交互，MVC并没有进行严格的限制。最为典型的就是允许View和Model绕开Controller进行直接交互，View不仅仅可以通过调用Model获取需要呈现给用户的数据，Model也可以直接通知View让其感知到状态的变化。当真正地将MVC应用于具体的项目开发中，不论是基于GUI的桌面应用还是基于Web UI的Web应用，如果不对Model、View和Controller之间的交互进行更为严格的限制，我们编写的程序可以比自治视图更为难以维护。

今天我们将MVC视为一种模式（Pattern），但是作为MVC最初提出者的Trygve M. H. Reenskau实际是将MVC视为一种范例（Paradigm），这可以从它在《Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC)》中对MVC的描述可以看出来：*In the MVC****paradigm****the user input, the modeling of the external world, and the visual feedback to the user are explicitly separated and handled by three types of object, each specialized for its task.*

模式和范例的区别在于前者可以直接应用到具体的应用上，而后者则仅仅提供一些基本的指导方针。在我看来MVC是一个很宽泛的概念，任何基于Model、View和Controller对UI应用进行分解的设计都可以成为MVC。当我们采用MVC的思想来设计UI应用的时候，应该根据应用框架（比如Windows Forms、WPF和Web Forms）的特点对Model、View和Controller的界限以及相互之间的交互设置一个更为严格的规则。在软件设计的发展历程中出现了一些MVC的变体（Varation），它们遵循定义在MVC中的基本原理。我们现在来简单地讨论一些一个常用的MVC变体。

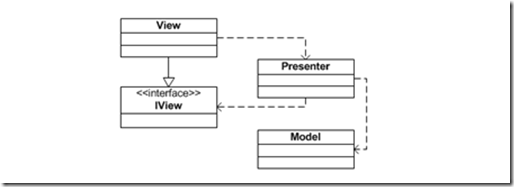
目录   
MVP   
    PV与SoC   
    View和Presenter交互的规则（针对SoC模式）   
    实例演示：SoC模式的应用   
Model 2

ASP.NETMVC与Model2

**一、 MVP**

MVP是一种广泛使用的基于架构模式，使用与基于事件驱动的应用框架，比如ASP.NET Web Forms和Windows Forms应用。MVP中的M和V对应中MVC的Model和View，而P（Presenter）则自然代替了MVC中的Controller。但是MVP并非仅仅体现在从Controller到Presenter的转换，更对地体现在Model、View和Presenter之间的交互上。

MVC模式中元素之间混乱的交互只要体现在允许View和Model绕开Controller进行单独“交流”，这在MVP模式中得到了彻底地解决。如下图所示，能够与Model直接进行交互的仅限于Presenter，View只能间接地通过Presenter调用Model。Model的独立性在这里得到了真正的体现，它不仅仅与可视化元素的呈现无关（View）和与UI处理逻辑（Presenter）无关。使用MVP的应用是用户驱动的而非Model驱动的，所以Model不需要主动通知View以提醒状态发生了改变。

[](http://images.cnblogs.com/cnblogs_com/artech/201203/201203082136509318.png)

MVP不仅仅避免了View和Model之间的耦合，更进一步地降低Presenter对View的依赖。如图1-2所示，Presenter依赖的是一个抽象化的View，即View实现的接口IView。这带来的最直接的好处就是使定义在Presenter中的UI处理逻辑变得易于测试。由于Presenter对View的依赖行为定义在接口IView中，我们只需要Mock一个实现了该接口的View就能对Presenter进行测试。

构成MVP三要素之间的交互体现在两个方面，即View/Presenter和Presenter/Model。Presenter和Model之间的交互很清晰，仅仅体现在Presenter对Model的单向调用。而View和Presenter之间该采用怎样的交互方式是整个MVP的核心，MVP针对关注点分离的初衷能否体现在具体的应用中很大程度上取决于两者之间的交互方式是否正确。按照View和Presenter之间的交互方式以及View本身的职责范围，Martin Folwer将MVP可分为PV（Passive View）和SoC（Supervising Controller）两种模式。

**PV与SoC**

解决View难以测试的最好的办法就是让它无须测试，如果View不需要测试，其先决条件就是让它尽可能不涉及到UI处理逻辑，而这就是PV模式目的所在。顾名思义，PV（Passive View）是一个被动的View，针对包含其中的UI元素（比如控件）的操作不是由View自身来操作，而交给Presenter来操控。

如果我们纯粹地采用PV模式来设计View，意味着我们需要将定义View中的UI元素通过属性的形式暴露出来。具体来说，当我们在为View定义接口的时候，需要定义基于UI元素的属性以使Presenter可以对View进行细粒度地操作，但这并不是意味着我们直接将View上的控件暴露出来。举个简单的例子，我们开发的HR系统 中具有如下图所示的Web页面用于根据部门获取员工列表。

[](http://images.cnblogs.com/cnblogs_com/artech/201203/201203082136527418.png)

现在通过ASP.NET Web Form应用来涉及这个页面，我们来讨论一下如果采用PV模式View的接口该如何定义。对于Presenter来说，View供它操作的控件有两个，一个是包含所有部门列表的DropDownList，另一个则是显示员工列表的GridView。在页面加载的时候，Presenter将部门列表绑定在DropDownList上，与此同时包含所有员工的列表被绑定到GridView。当用户选择某个部门并点击“查询”按钮后，View将包含筛选部门在内的查询请求转发给Presenter，后者筛选出相应的员工列表之后将其绑定到GridView。

如果我们为该View定义一个接口IEmployeeSearchView，我们不能像如下的代码所示将上述这两个控件直接以属性的形式暴露出来。针对数据绑定对控件类型的选择属于View的内部细节（比如说针对部门列表的显示，我们可以选择DropDownList也可以选择ListBox），不能体现在表示用于抽象View的接口中。在另一方面，理想情况下定义在Presenter中的UI处理逻辑应该是与具体的技术平台无关的，如果在接口中涉及到了控件类型，这无疑将Presenter也具体的技术平台绑定在了一起。

1: public interface IEmployeeSearchView

2: {

3: DropDownList Departments { get;}

4: GridView Employees { get; }

5: }

正确的接口和实现该接口的View（一个Web页面）应该采用如下的定义方式。Presenter通过属性Departments和Employees进行赋值进而实现对DropDownList和GridView进行绑定，通过属性SelectedDepartment得到用户选择的筛选部门。为了尽可能让接口只暴露必须的信息，我们特意将对属性的读写作了控制。

1: public interface IEmployeeSearchView

2: {

3: IEnumerable<string> Departments { set; }

4: string SelectedDepartment { get; }

5: IEnumerable<Employee> Employees { set; }

6: }

7:

8: public partial class EmployeeSearchView: Page, IEmployeeSearchView

9: {

10: //其他成员

11: public IEnumerable<string> Departments

12: {

13: set

14: {

15: this.DropDownListDepartments.DataSource = value;

16: this.DropDownListDepartments.DataBind();

17: }

18: }

19: public string SelectedDepartment

20: {

21: get { return this.DropDownListDepartments.SelectedValue;}

22: }

23: public IEnumerable<Employee> Employees

24: {

25: set

26: {

27: this.GridViewEmployees.DataSource = value;

28: this.GridViewEmployees.DataBind();

29: }

30: }

31: }

虽然从可测试性的角度来说PV模式是一种不错的选择，因为所有的UI处理逻辑全部定义在Presenter上，意味着所有的UI处理逻辑都可以被测试。但是我们需要将View可供操作的UI元素定义在对应的接口中，对于一些复杂的富客户端（Rich Client）View来说，接口成员将会变得很多，这无疑会提升编程所需的代码量。从另一方讲，由于Presenter需要在控件级别对View进行细粒度的控制，这无疑会提供Presenter本身的复杂度，往往会使原本简单的逻辑复杂化，在这种情况下我们往往采用SoC模式。

在SoC（Supervising Controller）模式下，为了降低Presenter的复杂度，我们将诸如数据绑定和格式化这样简单的UI处理逻辑逻辑转移到View中，这些处理逻辑会体现在View实现的接口中。尽管View从Presenter中接管了部分UI处理逻辑，但是Presenter依然是整个三角关系的驱动者，View被动的地位依然没有改变。对于用户作用在View上的交互操作，View本身并不进行响应，而是直接将交互请求转发给Presenter，后者在独立完成相应的处理流程（可能涉及到针对Model的调用）之后会驱动View或者创建新的View作为对用户交互操作的响应。

**View和Presenter交互的规则（针对SoC模式）**

View和Presenter之间的交互是整个MVP的核心，能够正确地应用MVP模式来架构我们的应用极大地取决于能够正确地处理View和Presenter两者之间的关系。在由Model、View和Presenter组成的三角关系的核心不是View而是Presenter，Presenter不是View调用Model的中介，而是最终决定如何响应用户交互行为的决策者。

打个比方，View是Presenter委派到前端的客户代理，而作为客户的自然就是最终的用户。对于以鼠标/键盘操作体现的交互请求应该如何处理，作为代理的View并没有决策权，所以它会将请求汇报给委托人Presenter。View向Presenter发送用户交互请求应该采用这样的口吻：“我现在将用户交互请求发送给你，你看着办，需要我的时候我会协助你”，而不应该是这样：“我现在处理用户交互请求了，我知道该怎么办，但是我需要你的支持，因为实现业务逻辑的Model只信任你”。

对于Presenter处理用户交互请求的流程，如果中间环节需要涉及到Model，它会直接发起对Model的调用。如果需要View的参与（比如需要将Model最新的状态反应在View上），Presenter会驱动View完成相应的工作。

对于绑定到View上的数据，不应该是View从Presenter上“拉”回来的，应该是Presenter主动“推”给View的。从消息流（或者消息交换模式）的角度来讲，不论是View向Presenter完成针对用户交互请求的同志，还是Presenter在进行交互请求处理过程中驱动View完成相应的UI操作，都是单向（One-Way）的。反应在 应用编程接口的定义上就意味着不论是定义在Presenter中被View调用的方法，还是定义在IView接口中被Presenter调用的方法最好都是没有返回值得。如果不采用方法调用的形式，我们也可以通过事件注册的方式实现View和Presenter的交互，事件机制体现的消息流无疑是单向的。

View本身仅仅实现单纯的、独立的UI处理逻辑，它处理的数据应该是Presenter实时推送给它的，所以View尽可能不维护数据状态。定义在IView的接口最好只包含方法，而避免属性的定义，Presenter所需的关于View的状态应该在接收到View发送的用户交互请求的时候一次得到，而不需要通过View的属性去获取。

**实例演示：SoC模式的应用**

为了让读者对MVP模式，尤其是该模式下的View和Presenter之间的交互方式具有一个深刻的认识，我们现在来进行一个简单的实例演示。本实例采用上面提及的关于员工查询的场景，并且采用ASP.NET Web Form来建立这个简单的应用，最终呈现出来的效果如上图所示。前面我们已经演示了采用PV模式下的IView应该如何定义，现在我们来看看SoC模式下的IView有何不同。先来看看表示员工信息的数据类型如何定义，我们通过具有如下定义的数据类型Employee来表示一个员工。简单起见，我们仅仅定义了表示员工基本信息（ID、姓名、性别、出生日期和部门）的5个属性。

1: public class Employee

2: {

3: public string Id { get; private set; }

4: public string Name { get; private set; }

5: public string Gender { get; private set; }

6: public DateTime BirthDate { get; private set; }

7: public string Department { get; private set; }

8:

9: public Employee(string id, string name, string gender, DateTime birthDate, string department)

10: {

11: this.Id = id;

12: this.Name = name;

13: this.Gender = gender;

14: this.BirthDate = birthDate;

15: this.Department = department;

16: }

17: }

作为包含应用状态和状态操作行为的Model通过如下一个简单的EmployeeRepository类型还体现。如代码所示，表示所有员工列表的数据通过一个静态字段来维护，而GetEmployees返回指定部门的员工列表。如果没有指定筛选部门或者指定的部门字符为空，则直接返回所有的员工列表。

1: public class EmployeeRepository

2: {

3: private static IList<Employee> employees;

4: static EmployeeRepository()

5: {

6: employees = new List<Employee>();

7: employees.Add(new Employee("001", "张三", "男", new DateTime(1981, 8, 24), "销售部"));

8: employees.Add(new Employee("002", "李四", "女", new DateTime(1982, 7, 10), "人事部"));

9: employees.Add(new Employee("003", "王五", "男", new DateTime(1981, 9, 21), "人事部"));

10: }

11: public IEnumerable<Employee> GetEmployees(string department = "")

12: {

13: if (string.IsNullOrEmpty(department))

14: {

15: return employees;

16: }

17: return employees.Where(e => e.Department == department).ToArray();

18: }

19: }

接下来我们来看作为View接口的IEmployeeSearchView的定义。如下面的代码片断所示，该接口定义了BindEmployees和BindDepartments两个方法，分别用于绑定基于部门列表的DropDownList和基于员工列表的DataView。除此之外，IEmployeeSearchView接口还定义了一个事件DepartmentSelected，该事件会在用户选择了筛选部门后点击“查询”按钮时触发。DepartmentSelected事件参数类型为自定义的DepartmentSelectedEventArgs，属性Department表示用户选择部门。

1: public interface IEmployeeSearchView

2: {

3: void BindEmployees(IEnumerable<Employee> employees);

4: void BindDepartments(IEnumerable<string> departments);

5: event EventHandler<DepartmentSelectedEventArgs> DepartmentSelected;

6: }

7:

8: public class DepartmentSelectedEventArgs : EventArgs

9: {

10: public string Department { get; private set; }

11: public DepartmentSelectedEventArgs(string department)

12: {

13: Guard.ArgumentNotNullOrEmpty(department, "department");

14: this.Department = department;

15: }

16: }

作为MVP三角关系核心的Presenter通过具有如下定义的EmployeeSearchPresenter表示。如下面的代码片断所示，表示View的只读属性类型为IEmployeeSearchView接口，而另一个只读属性Repository则表示作为Model的EmployeeRepository对象，两个属性均在构造函数中初始化。

1: public class EmployeeSearchPresenter

2: {

3: public IEmployeeSearchView View { get; private set; }

4: public EmployeeRepository Repository { get; private set; }

5:

6: public EmployeeSearchPresenter(IEmployeeSearchView view)

7: {

8: this.View = view;

9: this.Repository = new EmployeeRepository();

10: this.View.DepartmentSelected += OnDepartmentSelected;

11: }

12: public void Initialize()

13: {

14: IEnumerable<Employee> employees = this.Repository.GetEmployees();

15: this.View.BindEmployees(employees);

16: string[] departments = new string[] { "销售部", "采购部", "人事部", "IT部" };

17: this.View.BindDepartments(departments);

18: }

19:     protected void OnDepartmentSelected(object sender, DepartmentSelectedEventArgs args)

20: {

21: string department = args.Department;

22: var employees = this.Repository.GetEmployees(department);

23: this.View.BindEmployees(employees);

24: }

25: }

在构造函数中我们注册了View的DepartmentSelected事件，作为事件处理器的OnDepartmentSelected方法通过调用Repository（即Model）实现了针对所选部门的筛选，而返回的员工列表通过调用View的BindEmployees方法实现了在View上的数据绑定。在Initialize方法中，我们通过调用Repository获取了表示所有员工的列表，并通过View的BindEmployees方法显示在界面上；通过调用View的BindDepartments方法将作为筛选条件的部门列表绑定在View上。

最后我们来看看作为View的Web页面如何定义，如下所示的是作为页面主体部分的HTML，核心部分之包括一个用于绑定筛选部门列表的DropDownList和一个绑定员工列表的GridView。

1: <html xmlns="http://www.w3.org/1999/xhtml">

2: <head runat="server">

3: ...

4: </head>

5: <body>

6: <form id="form1" runat="server">

7: <div id="page">

8: <div class="top">

9: 选择查询部门：

10: <asp:DropDownList ID="DropDownListDepartments" runat="server" />

11: <asp:Button ID="ButtonSearch" runat="server" Text="查询" OnClick="ButtonSearch\_Click" />

12: </div>

13: <asp:GridView ID="GridViewEmployees" runat="server" AutoGenerateColumns="false" Width="100%">

14: <Columns>

15: <asp:BoundField DataField="Name" HeaderText="姓名" />

16: <asp:BoundField DataField="Gender" HeaderText="性别" />

17: <asp:BoundField DataField="BirthDate" HeaderText="出生日期" DataFormatString="{0:dd/MM/yyyy}" />

18: <asp:BoundField DataField="Department" HeaderText="部门"/>

19: </Columns>

20: </asp:GridView>

21: </div>

22: </form>

23: </body>

24: </html>

如下所示的是该Web页面的后台代码的定义。它实现了定义在IEmployeeSearchView接口的两个方法（BindEmployees和BindDepartments）和一个事件（DepartmentSelected）。表示Presenter的同名属性在构造函数中被初始化。在页面加载的时候（Page\_Load方法）Presenter的Initialize方法被调用，而在“查询”按钮被点击的时候（ButtonSearch\_Click）事件DepartmentSelected被触发。

1: public partial class Default : Page, IEmployeeSearchView

2: {

3: public EmployeeSearchPresenter Presenter { get; private set; }

4: public event EventHandler<DepartmentSelectedEventArgs> DepartmentSelected;

5: public Default()

6: {

7: this.Presenter = new EmployeeSearchPresenter(this);

8: }

9: protected void Page\_Load(object sender, EventArgs e)

10: {

11: if (!this.IsPostBack)

12: {

13: this.Presenter.Initialize();

14: }

15: }

16: public void BindEmployees(IEnumerable<Employee> employees)

17: {

18: this.GridViewEmployees.DataSource = employees;

19: this.GridViewEmployees.DataBind();

20: }

21: public void BindDepartments(IEnumerable<string> departments)

22: {

23: this.DropDownListDepartments.DataSource = departments;

24: this.DropDownListDepartments.DataBind();

25: }

26: protected void ButtonSearch\_Click(object sender, EventArgs e)

27: {

28: string department = this.DropDownListDepartments.SelectedValue;

29: DepartmentSelectedEventArgs eventArgs = new DepartmentSelectedEventArgs(department);

30: if (null != DepartmentSelected)

31: {

32: DepartmentSelected(this, eventArgs);

33: }

34: }

35: }

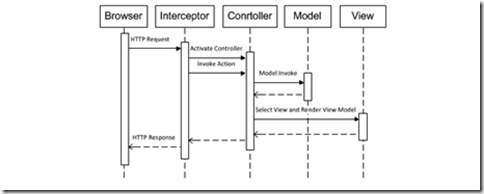
**二、Model2**

Trygve M. H. Reenskau当初提出的MVC是作为基于GUI的桌面应用的架构模式，并不太适合Web本身的特性。虽然MVC/MVP也可以直接用于ASP.NET Web Form应用，但这是因为微软基于桌面应用的编程模式 来设计基于Web Form的ASP.NET应用框架的。Web应用不同于GUI桌面应用在于用户是通过浏览器与应用进行交互，交互请求和相应是通过HTTP请求和回复来完成的。

为了让MVC能够Web应用提供原生的支持，另一个被称为Model2 的MVC变体被提出来，Model2来源于基于Java的Web应用架构模式。Java Web应用具有两种基本的架构模式，分别被称为Model1和Model2。Model1类似于我们前面提及的自治试图模式，它将数据的可视化呈现和用户交互操作的处理逻辑合并在一起。Model1使用于那些比较简单的Web应用，对于相对复杂的应用应该采用Model 2。

为了让开发者采用相应的编程模式进行GUI桌面应用和Web应用的开发，微软通过ViewState和Postback对背后的HTTP请求和回复机制进行了封装，使我们能够像编写Windows Forms应用一样采用事件驱动的方式进行ASP.NET Web Forms应用的编程。而Model 2采用完全不同的设计，它让开发者直接面向Web，让他们关注HTTP的请求和回复流程，所以Model 2提供对Web应用原生的支持。

对于Web应用来说，和用户直接交互的UI界面由浏览器来提供，接下来我们详细讨论作为MVC的三要素是如何相互协作对从浏览器发出的用户交互请求的响应的，下图所示的序列图体现了整个流程的全过程。

[](http://images.cnblogs.com/cnblogs_com/artech/201203/201203082142456061.png)

Model 2种一个HTTP请求的目标是Controller中的某个Action，后者体现为定义在Controller类型中的某个方法，所以对请求的处理最终体现在对Controller对象的激活和对Action方法的执行。一般来说，Controller、Action以及作为Action方法的部分参数（针对HTTP-GET）可以直接通过请求的URL解析出来。

如上图所示，我们通过一个拦截器（Interceptor）对抵达Web服务器的HTTP请求进行拦截。一般的Web应用框架都提供了针对这样一种拦截机制，对于ASP.NET来说，我们可以以HttpModule的形式来定义这么一个拦截器。拦截器根据请求解析出目标Controller和对应的Action，Controller被激活之后Action方法被执行。对于需要传入Action方法的输入参数，则来源于请求地址或/和Post的数据。

在Controller的Action方法被执行过程中，它可以调用Model获取或者改变其状态。在Action方法执行的最后阶段会选择相应的View，绑定在View上的数据来源Model或者基于显示要求进行得简单逻辑计算，我们有时候它们成为VM（View Model），即基于View的Model（MVC中的Model是与UI无关的）。生成的View最终写入HTTP回复并最终呈现在用户的浏览器中。

和MVP一样，Model 2完全隔断了View和Model之间的联系。Controller作为支配者地位在Model 2体现尤为明显，用户交互请求不再由View报告给Controller（Presenter），而是由拦截器直接转发给Controller。Controller不仅仅决定着Model的调用，还决定了View的选择和生成。ASP.NET MVC就是基于Model 2模式设计的。

**三、ASP.NETMVC与Model2**

凭着读者对ASP.NET MVC的了解，通过上面对Model2模式的介绍，应该很清楚地认识到ASP.NET MVC就是根据Model2模式设计的。基于HTTP请求的拦截机制是通过一个自定义的HttpModule和一个自定义HttpHandler来实现的，在本章的最后我们会通过一个例子来模拟ASP.NET MVC的工作原理。

在上面我们多次强调MVC的Model是维持应用状态提供业务功能的领域模型，或者是多层架构中进入业务层的入口或者业务服务的代理，但是ASP.NET MVC中的Model还是这个Model吗？稍微了解ASP.NET MVC的读者都知道ASP.NET MVC的Model仅仅是绑定到View上的数据而已，它和MVC模式中的Model并不是一回事。由于ASP.NET MVC中的Model是基于View的，我们可以将其称为View Model。

由于ASP.NET MVC只有View Model，所以ASP.NET MVC应用框架本社实际上仅仅关于View和Controller，真正的Model以及Model和Controller之间的交互体现在我们如何来设计Controller。我个人觉得将用于构建ASP.NET MVC的MVC模式成为**M(Model)-V(View)-VM(View Model)-C(Controller)**也许更为准确。

**参考资料：**1、Dino Esposito，Andrea Saltarello《Micorsoft .NET Architecting Applications for Enterprise》   
2、Adam Freeman， Steven Sanderson《Pro ASP.NET MVC 3 Framework》   
3、Martin Fowler 《GUI Architectures》： <http://martinfowler.com/eaaDev/uiArchs.html>   
4、Martin Fowler 《Passive View》：<http://martinfowler.com/eaaDev/PassiveScreen.html>   
5、Martin Fowler 《Supervising Controller》：<http://martinfowler.com/eaaDev/SupervisingPresenter.html>   
6、Mike Potel，VP & CTO，Taligent, Inc. 《MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java》   
7、Model View Controller- Wikipedia, the free encyclopedia：<http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller>   
8、Model View Presenter- Wikipedia, the free encyclopedia：<http://en.wikipedia.org/wiki/Model_View_Presenter>   
9、Steve Burbeck, Ph.D. 《Applications Programming in Smalltalk-80(TM):How to use Model-View-Controller (MVC)》：<http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html>