系统架构师-基础到企业应用架构-表现层

一、前言

最近也许是由于假期的原因，我发布的文章的速度变慢了，对大家说下抱歉，这个系列的确我很难写，感谢大家对我的支持和关注，的确我在发布后得到大家的支

持和认可，让我有了更多的动力，之前发布的有些内容，可能对各层讲解的内容的广度还不够，当然这和我个人的水平面有关，还请各位多多提出宝贵意见和建议。

从本篇开始，我将会采用更加规范的格式，更严谨的求知态度，更加准确的表达，去将接下来的系列文章写完，并且与群中的很多朋友交流后，他们希望出一个总

的 PDF 电子书，这样可以方便阅读，的确谢谢各位的支持，我目前将以后每篇写的内容，放一份 PDF 格式的在群共享中，有需要的朋友可以进行相应的下载，由于本人

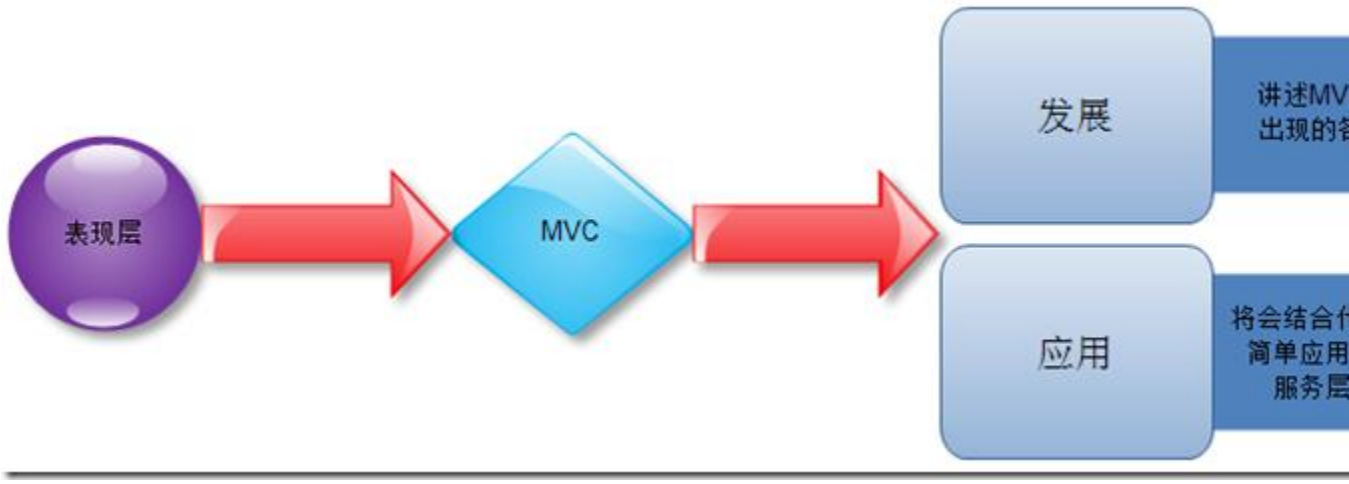
的写作水平有限,所以在书写的深度和书写的格式上还有很多的缺点,还希望大家多多指出。

二、开篇

本篇我们将针对系统架构分层中的表现层进行讲述，分析表现层的架构与设计模式，当然我们会结合目前比较流行的表现层模式进行分析讲解，主要是围绕 MVC 模

式的起源及发展的过程进行讲述，并且分析目前 MVC 模式在不同 UI 层中的应用设计，由于本人的水平有限，加之实际的项目中可能应用的理解和经验水平不足，可能在

某些分析的地方不对，还请大家提出。我们之前的写作惯例，我们先来看看本文的主要讲述的内容吧



本文将会以上面的 2 点为主线展开去讲解表现层的内容。

三、内容提要

四、表现层模式

4.1、表现层的职责

我们知道任何一个应用程序，如果想要更好的与客户交互，我们一般都是通过提供一个用户界面去完成与用户的交互，当然通过前面我们讲述的，服务层与业务逻

辑层的讲解，其实都是为了更好的为表现层服务的，通常，我们都不是特别的重视表现层，但是其实，表现层同样重要。其实通常我们在关注各个分层的时候，我们对

每个分层的重视程度会不同，可能是由于我们自身的爱好，态度，或者是专业技能所决定的，但是一个好的系统，必然要求我们不管哪个分层都要足够的重视。

我们在做表现层时，通常我们会关注下面几个组件，首先是用户界面。也就是客户能够看到的用户界面简称 UI，还有一部分就是与用户行为进行交互的组件，通常

我们叫做表现层逻辑，用户的所有操作都通过表现层逻辑来支持，表现逻辑层将负责其他层与 UI 层之间的交互。

根据我们前面讲述的各分层的设计我们知道，表现层逻辑将主要与服务层或者业务逻辑层进行交互。具体的关系如下：

通过上图我们知道表现逻辑层的位置，可以说如何组织好用户界面及表现层逻辑同其他层之间的关系，就是决定设计好坏的关键。也可以说表现层逻辑决定了用户

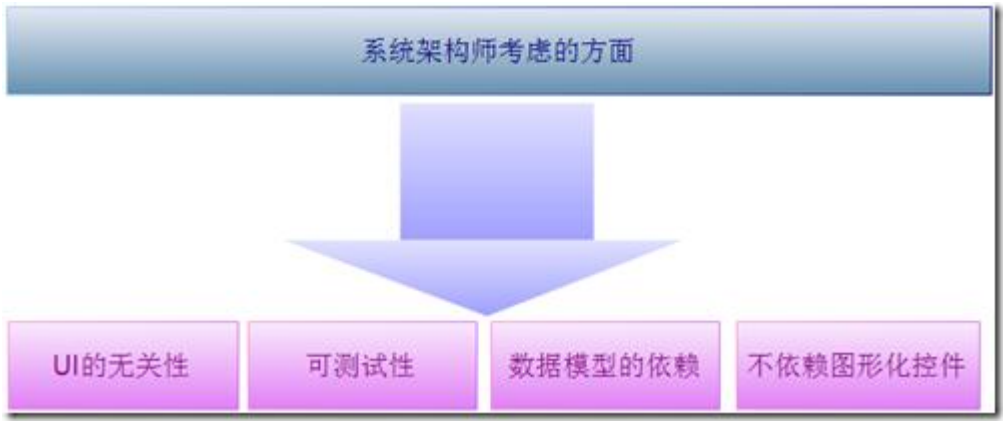
与系统交互。

下面我们来先看看表现层的职责吧

我们一般都把如下职责看作是表现层的职责，是不是你也有相同的看法？

其实，像图中说的验证，格式化，添加样式等这些更应该属于 UI 层组件，而不应该属于表现层，但

是作为架构师，你必须考虑的更全面，从更高的层次去考虑，比如说下面的几个方面：



只有更好的考虑这些因素，才能决定表现层的架构，我们针对上面的几个方面来简单的分析一下。

1、UI 的无关性：

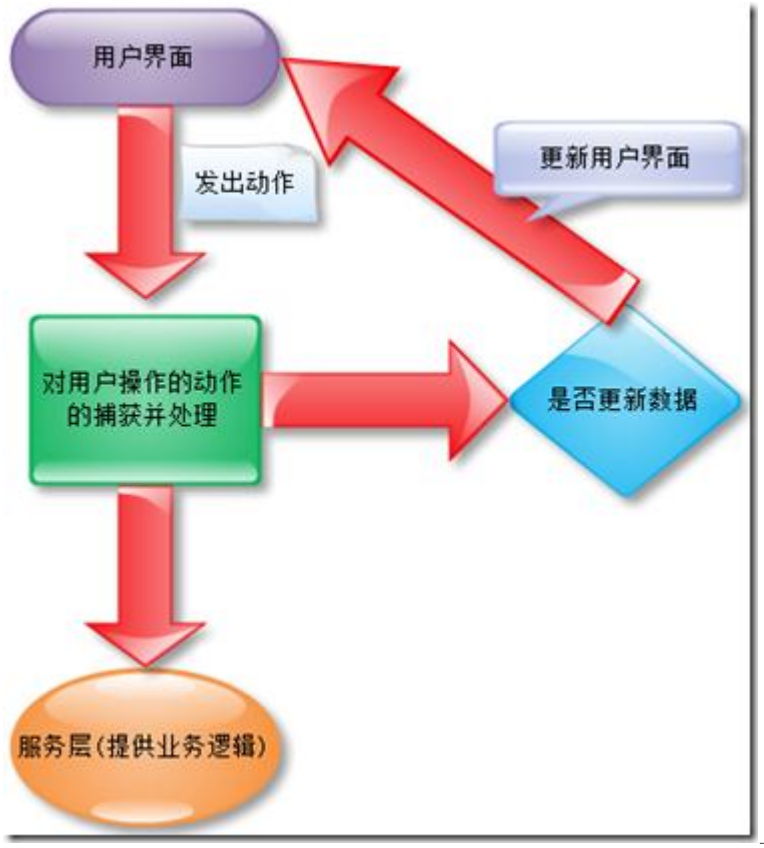
这里的 UI 的无关性就是指不依赖 UI 层的实现技术，不管是上面说的 Web 还是 Winform，还是其他的 WPF、WP 还是其他的任何 UI 表现形式。我们知道这样的表现层设计出来是理想的，我们只是想尽可能的提高可复用性，我们希望的是虽然不同的 UI 使用的实现技术不同，但是这些 UI 能够共用表现层逻辑，这时候我们就能做到很好的复用性。

我们需要知道，这里提出的代码重用只是说是尽可能大的做到表现层逻辑重用，但是有些情况我们没有办法做到，例如 Silverlight 与 WPF，由于他们使用不同的 CLR，所以我们只能说是做到代码重用，我们需要针对他们各自进行单独的编译。

2、可测试性

任何分层都需要能够测试，因为只有测试才能知道功能是否能够很好的满足需求。但是通常来说表现层的测试，相比其他分层要复杂一些，因为表现层需要考虑的因素很多，所以决定了测试的难度，所以更好的测试性就是我们架构设计的基本要求。

一般来说我们在表现层的处理流程是这样的：



上面的图中大概描述了，我们在表现层中的基本流程，我们来结合图形来说明下：用户界面通过用户

操作，将触发相应的行为事件，这些事件将会被某个特殊的控制器进行处理，控制器通过发送给指定的处理函数，然后将处理函数返回的结果，反馈给用户界面。

通过上面的流程我们知道，我们测试时必须模拟用户的行为，并且如何保证用户的行为事件能够得到正确的处理，这些都是我们必须测试的内容。我们如果将所有

的内容都写在表现层，那么我想可能测试起来不会那么容易，如果说我们的用户界面与表现层逻辑写在一个页面中，可能我们并不容易测试，幸好 APS.NET 中通过代码

后置的技术，实现设计层与表现层逻辑代码的分离。当然我们在之前就对架构设计的实现提出了一个观点，就是分离功能点，将关注的功能点进行分离，提出更高的抽

象，这样可以达到更好的测试的目的。

3、不依赖数据模型

在服务层、业务逻辑层中我们都提到了数据传输对象，并且把这个对象作为与表现层通信的载体，使用了数据传输对象后，那么我们就可以做到表现层不依赖于我

们的数据模型，但是我们也讲过，数据传输对象为项目带来了更多的类，更多的工程文件，所以这会对我们的开发工作带来额外的工作量。因为不同的表现层需要一个

与之对应的数据传输对象，这些都和前面我们介绍的情况是一致的。因此，我们在项目中是否使用数据传输对象，我们需要根据项目的需要来决定。

4、不依赖图形化元素

我们这里的图形化元素是指，开发工具给我们提供的控件和工具集合，我们通过这些图形化元素设计出与用户交互的界面，我们希望我们的表现层能够在不同的控

件上可以以不同的方式表现出来，我们同时希望图形化元素中的任意修改不会影响到表现层逻辑，举个例子，我们平时在博客园中的模板设计，其实就是个很好的例

子，我们选择不同的模板，我们的博客页面就会在皮肤，样式，主题等方面都发生变化，但是并不影响我们的表现层。

4.2、UI 层与表现层逻辑

4.2.1、用户界面的职责

我们接下来看看 UI 层的相关职责，我们知道，我们平时在系统设计过程中，一般架构师不会直接参与到用户的界面的设计中，架构师更侧重系统的可用性和可访问

性。我们认为 UI 层的主要功能包括下面几项：

下面我们来分别说说吧

1、数据显示

我们都知道 UI 层是用户使用系统的唯一入口，那么首先用户界面必须将系统中的一些信息进行展示，这些信息可能包括，普通信息、提示信息、系统中的相关数据

信息等，良好的设计 UI 能够很好的展示数据，用户界面需要支持本地化及全球化功能。

2、友好的交互

交互就体现在用户使用系统的功能，这样就会有这样的操作，用户输入相应的数据，通常来说一个好的 UI 层，会提供给用户一个比较人性化的输入页面，并且会根

据用户的输入，返回用户想得到的结果。不管 UI 层的具体技术是什么，好的用户交互页面都是系统中最重要的部分。输入的数据我们如何做到安全性方面的验证，是我

们必须重视的部分。

3、总体外观

系统的功能都是通过用户界面来作为统一入口，提供给用户使用的，所以一般来说我们在设计 UI 时必须要有精致的用户界面，软件的目的就是抓住用户，能够给用

户较好的用户体验，而且根据软件的用户群来进行不同的设计，如果是提供给大众使用的软件，那么我们必须达到设计精致的用户界面的要求，如果是提供给企业内部

使用，那么我们可以注重功能而不是视觉效果的震撼，当然不是说不重视 UI，只是没有那么重视。

4.2.2、表现层中容易产生的误区

我们来看看我们平时在表现层的设计与实现过程中可能存在的误区吧

我们来看看每个误区的说明吧

1、过度依赖工具

自从有了 Visual Studio 开发工具以来，就因为其提供了大量的控件，为我们开发简单的应用程序提供了很大的方便，我们通常都是通过拖拽控件来完成 UI 层的设

计。我们通过一些带有事件的控件，去完成与其他层之间的交互，我们可以在事件中直接处理逻辑，而且很方便很快捷。这也使得我们的开发效率很高，当然这是开发

简单的应用程序时我们通过这样来做，没有什么问题。但是，当我们开发大型的企业级应用时，这样的方式可能就不是理想的选择了，为什么这么说呢？我们不是说不

用开发工具提供的控件，而是应该将职责进行划分，比如说，用设计器完成界面的设计，然后其他的代码都是通过我们的手工边写，而不是通过开发工具提供的一些控

件去完成，比如说 Visio Studio 提供的数据源绑定控件等，而且如何将事件中的代码进行更好的抽象就是我们这里的需要考虑的问题。

2、表现层的边界

我们先要清楚表现层的职责是什么？它主要负责什么？我们知道，表现层中的 UI 层是个很薄弱的层，UI 层应该只负责与用户进行交互，表现层逻辑应该负责协调数

据流入/流出 UI 层，那么这里的表现逻辑层应该具有什么样的功能。我们在前面介绍业务逻辑层中也提到过，我们有时候将业务逻辑放在表现层中也是可以接收的，还是

看具体的系统需要。

有的时候我们很钟爱把业务逻辑层中的功能反正表现层逻辑中，我自己的很多项目中也是这样做的，因为我目前的项目中也有这样的代码充斥者，我们说过这不是

万恶的，但是当系统有一定规模时，表现逻辑层就会变成一个无所不有，无所不包的容器，难以维护和测试。

通过上面的分析我们知道，我们还是需要根据系统的实际需要来决定是不是把一些业务逻辑写在表现逻辑层中，对于大型的企业级项目来说，我们必须考虑这些问

题，因为好的设计可以提供测试行，维护性等，我们对表现逻辑层的要求也会比较高，我们期望通过 SOC 来实现，我们一般是通过分层来实现，因为分层来分离功能

点，我们通过分离关注点可以提供表现逻辑层的重构，提醒更好的设计结构和代码结构，提高复用性。

3、WUGWYW

大家估计出一看，还以为是什么意思呢，原谅我这里卖了个关子，意思就是说(what-user-get-is-what-you-want)用户得到的就是你想要的。

我们在使用图形化的界面设计工具时，为了能更好的提供开发效率，现在很多的开发工具提供了界面预览的功能，基本上是所见即所得，但是并不是全部，例如

Visio Studio 提供的 web 开发中的预览，但是有时候我们还需要自己动手去处理一些表现层的东西，所以我们会对界面通过 CSS 去控制 UI 层的显示，我这里解释这个的

意思就是，有时候我们需要多写一些代码来提供更好的功能，满足用户的需求。

4.3、MVC 模式的提出及演化

从本节开始我们就开始对表现层中的模式进行讲解，我们主要是针对 MVC 模式的起源及发展过程进行讲解。

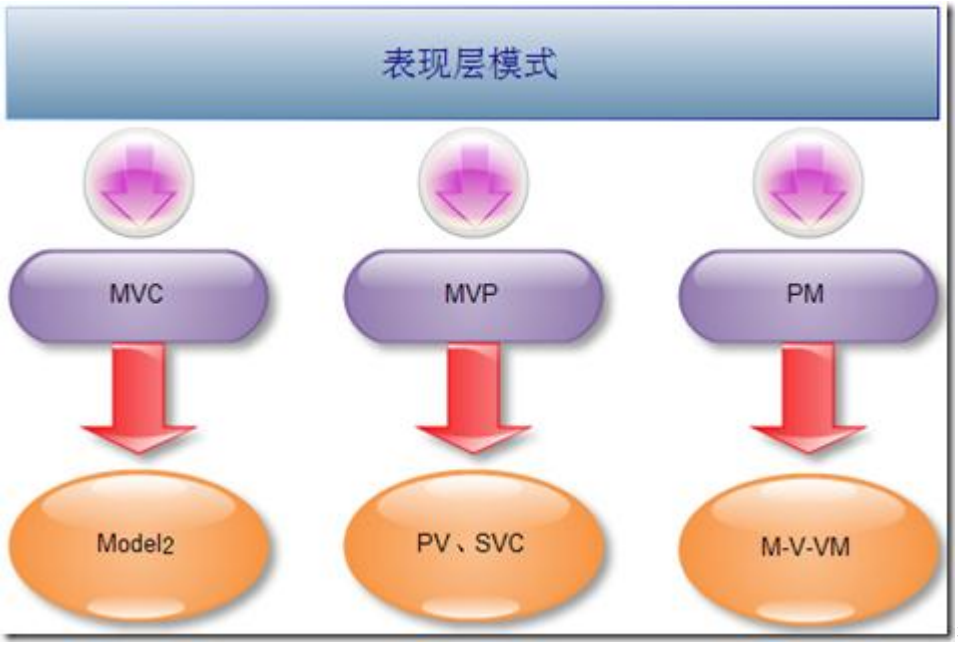
4.2.2、MVC 模式(Model-View-Controller)

MVC 模式起源于上个世纪 80 年代，目前已经有了 30 年的历史了，在现在的今天，可能我们没有办法再使用原来的定义去完成表现层的设计了，但是我们现在使用

的 MVC 模式都是从原来的 MVC 模式中演变过来的，以适应目前的软件开发需求。我们这里可能说的 MVC 模式也不是原始的定义，包括后面介绍的 MVP 模式等。

我们这里需要知道表现层都有哪些分类，每种分类可能演变出哪些哪些分类等，这对我们对表现层的模式有个大概的结构，对我们对表现层的设计有很大的帮助，

下面我们来看看吧:



我们将会详细的讲解这些模式及应用，本节将详细的讲解 MVC 的演变及原理，其中由 MVC 模式演变的 Model2 模式正式目前 ASP.NET MVC 的实现方式。

在早起我们开发表现层的时候，我想我们更多的是将表现层逻辑与视图放在一个文件中，可以看作是功能自治的视图。用户与视图交互，视图负责捕获用户的输入信息，并在内部处理，然后更新自己或者跳转到另外一个视图。这样的组织形式，不但难以维护，更难测试，因此更加让大家渴望有一种好的模式，从这样复杂的表现层中分离出来，由此 MVC 模式便提出来了。

MVC 模式的提出，将我们从视图自治的设计方式脱离出来，自治的视图内部的处理，可能包含业务层，数据访问层或者服务层等等，当然还有自身的设计层。MVC 模式，则让我们把自治视图的功能进行分离，将自治视图分解成，视图层，控制层，模型的形式，来分离关注点，通过关注点的分离，来降低系统的复杂度，进而提供系统的更好的设计性。

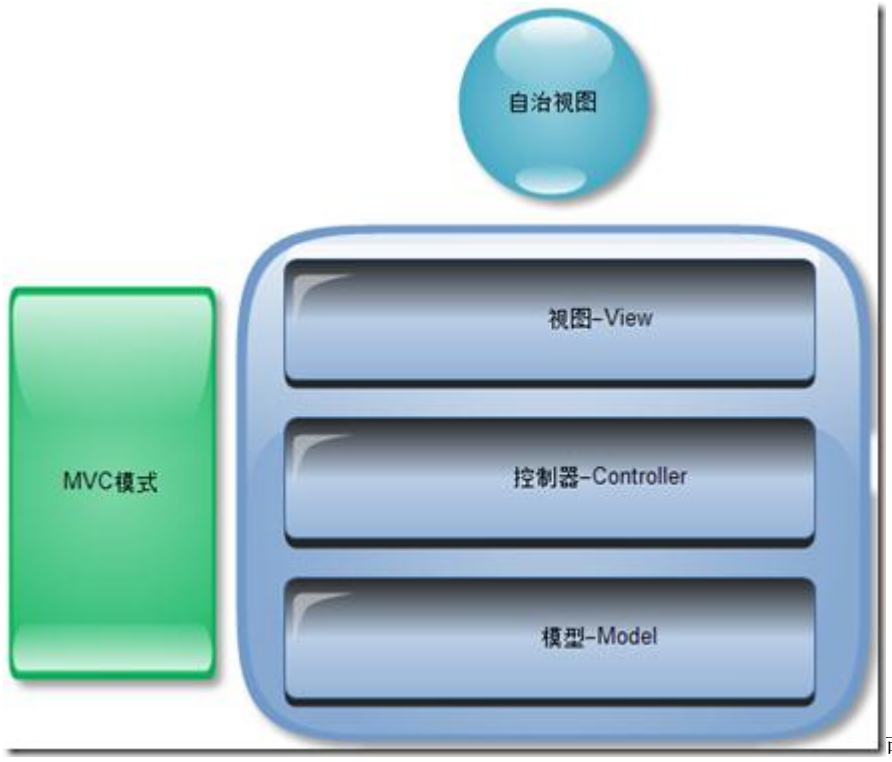
那么 MVC 模式，也有人称作模范，那么 MVC 到底是模式还是模范呢？我们来看看模式与模范的定义吧？可能就更能理解 MVC 了

模式在软件领域中是指一个被证明过的，具体的某类问题的解决方案。

模范是指某一类相似问题的解决方案，一般是指一类相似的模式。

通过上面的解释，我们知道 MVC 是模式。

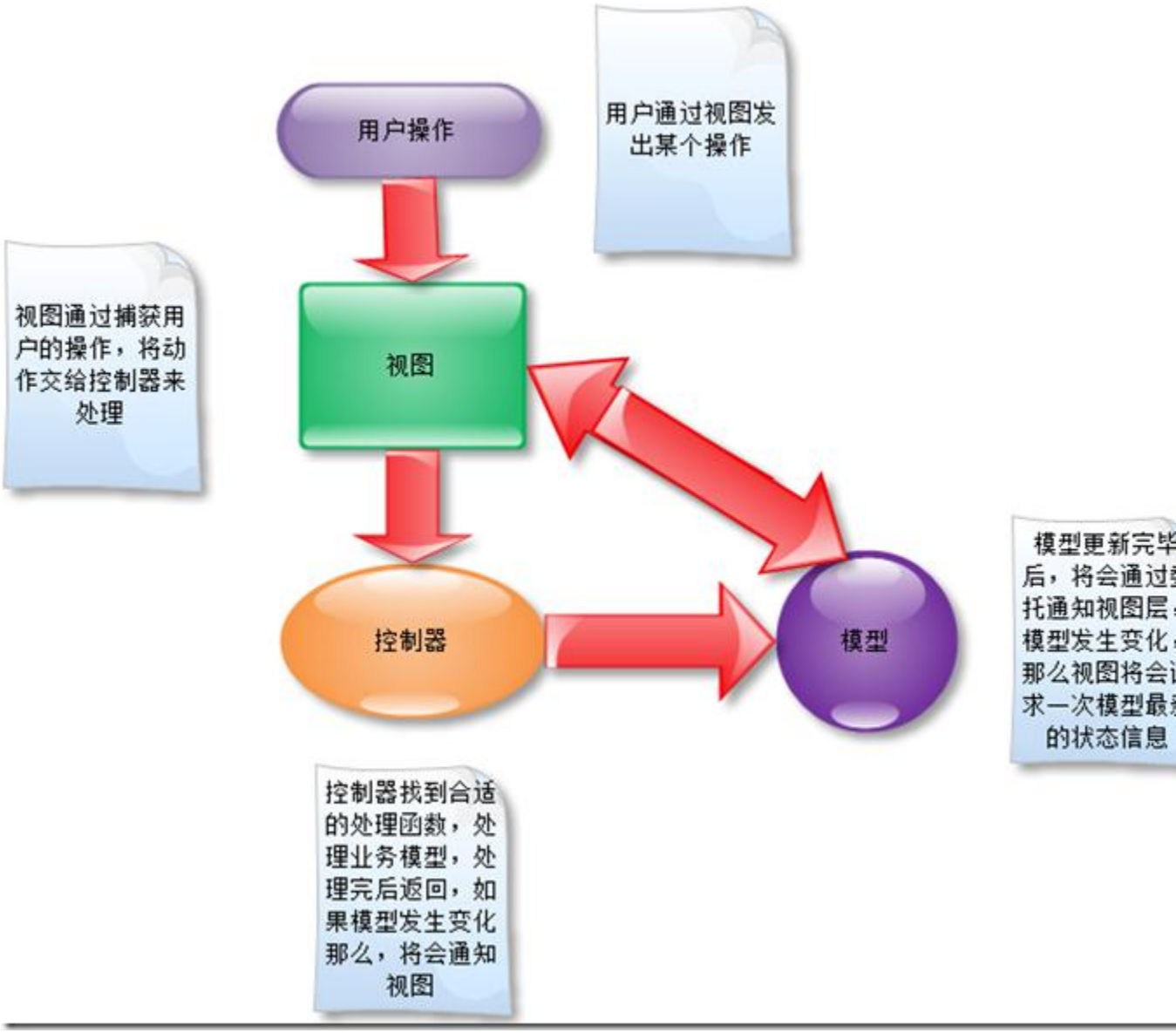
我们来看看自治视图与 MVC 的结构上的区别，MVC 模式我们知道将自治视图中的功能进行分离，分离成功能相对独立的组件，并且通过这些组件的交互完成表现层的工作。



可以这样说，自治视图将所有的功能放在一个容器中，而 MVC 模式关注的是功能点的分离，通过功能的分离开实现，我们通过将表现层划分成 3 个不同角色的组件，通过这些组件之间的配合来完成自治视图完成的工作。

MVC 的原始定义是什么呢？我们来看看，通过书上的记载称，MVC 的原始定义中，将模型作为业务逻辑的入口，模型中，我们能够看到程序的状态，我们需要将模型中的数据显示在视图中并且视图中接收用户的动作，然后通过模型来响应用户的操作行为。模型会处理控制器中发出的操作请求，一般来说这些请求都会对模型的状态有一定改变。

但是目前随着表现层模式的发展，现在 MVC 的定义已经不像原始定义的那样了，现在的模型只负责保持应用程序的状态，大多数的工作都是在视图和控制器来完成，控制器成为更核心的组件。我们现有的 MVC 模式中对模型的定义可能更多的采用是业务中的数据对象，或者专门针对领域模型来构建的领域对象或者是数据传输对象。当然还是根据我们在工作中的项目的需要来决定了。我们来看看 MVC 模式中的简单的这几个组件之间的调用关系吧



当然大家请不要诧异，这是最原始的 MVC 模式的处理流程。当今比较流行的 MVC 框架中流程已经不是这样处理的了，但是我们通过了解原始的定义将对我们更深入

的了解 MVC 有所帮助。MVC 当时的提出，并不是针对 WEB 来说的，但是 MVC 模式的发展，却是在 WEB 上流行起来的，像现在微软提供的 ASP.NET MVC 框架对 MVC 进

行很多的优化和修改。我们后面会讲述这个模式的原理。

我们来详细的看看 MVC 中的视图可能的代码：一般来说视图层通过一些界面设计的控件元素向用户展示相关的信息，用户通过界面层完成与系统的交互，那么我们

如何在视图中捕获用户的操作呢？视图通过一些列的控件去完成用户动作的捕获，比如说输入框，或者按钮等，视图的前台通过委托与后台的代码绑定，我们知道，微

软在处理.NET 的表现层中有很大的技巧在其中，通过视图与后台代码放在不同的类文件中去完成分离，这样，视图中用户发出的动作将会在代码后置类中捕获，然后进

行相应的处理。

我们先来看看控制器的可能代码

```
/// <summary>
/// 控制器类
/// </summary>
public class Controller
{
    public void Action(ActionType actionType)
    {

    }
}

/// <summary>
/// 控制器完成的动作
/// </summary>
public enum ActionType
{
    Add,
    Delete,
    Query,
    Update
}
```

视图中的可能代码：

```
private void button1_Click(object sender, RoutedEventArgs e)
{
    Controller controller = new Controller();
```

```
            controller.Action(ActionType.Add);
        }
```

代码中的控制器可以是任何特定技术的视图，控制器将会为视图中每个可能发生的操作，提供一个专门的方法，每个支持用户操作的动作都对应着一个与之相关的

处理服务。视图负责数据的呈现，控制器负责更新模型，如果模型发生变化，那么将会通过观察者模式通知视图其状态发生变化，然后视图将会决定是否将模型的变化

体现在视图上，如果需要更新，那么视图将会从模型中读取最新的数据信息。

MVC 中的模型我们上面罗列了几种可能，如果说我们这里的模型是业务逻辑层的话，那么我们就会直接调用业务逻辑对象中的方法完成模型状态的更新，这些都是

控制器来完成的，如果说 MVC 中的模型是数据传输对象，那么可能控制器将会通过业务逻辑层抽象出来的服务层的接口来访问与该数据传输对象相对应的公开方法去完

成模型的操作。如果说我们这里的模型是业务模型，那么控制器只是完成路由的功能，先要解析动作的发出者，然后要将这个动作解析交给哪个业务模型来处理，转发

完毕后就忘记了，其他的就有业务模型自动完成。

控制器有时候还要负责选择呈现视图的跳转功能，如果需要跳转时，那么控制器就会创建新的视图，控制器，模型。

我们有时候可能想要通过全局配置的方式去完成重定向，比如说通过 XML 文件去配置，类似工作流那样的方式来做的话。通过配置定向页面，那么我们如何来做

呢？我们就需要提供一个服务层完成定向，比如说通过一个通用的类去完成这样的操作。

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <route>
    <urls>
      <url key="key" value="value.aspx">
      </url>
    </urls>
  </route>
</configuration>
```

具体的路由代码如下：

```
    public class Route
```

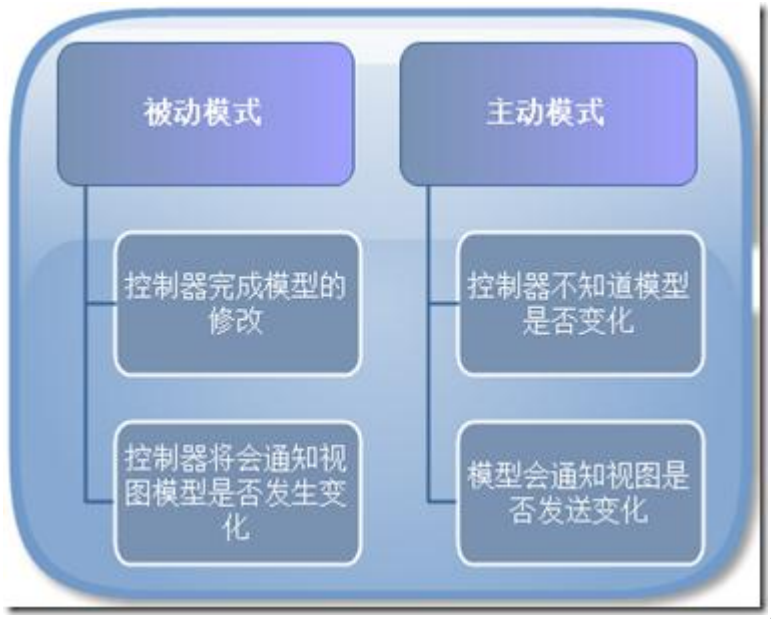
```
{
    public Route()
    {

    }

    /// <summary>
    /// 重定向服务
    /// </summary>
    public void RedirectUrl(string key)
    {
        string url = XMLHelper.GetValue(key);
    }
}
```

当然我这里只是给个可能的思路去完成服务地址的跳转，并不是比较好的方案。

通过上面的讲解我们知道，视图的更新并没有提出由谁来操作完成。通常来说有 2 中模式，我们来看看吧



后面我们会讲 s 解 ASP.NET MVC 模式中的视图更新的模式。

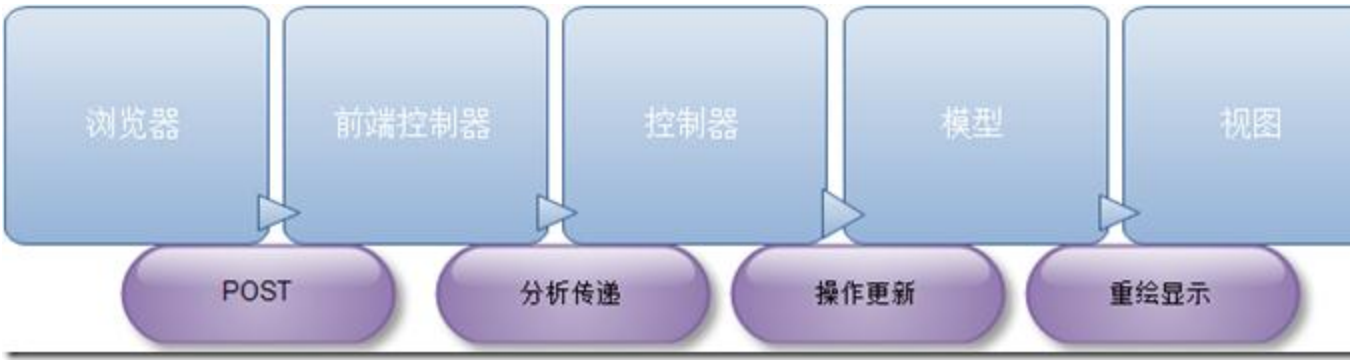
我们接下来看看 MVC 在 WEB 端开发的变体吧，那就是 Model2 模式，我们先来看看这种模式相比之前的 MVC 模式都有哪些变化吧

主要差别体现在以下几个方面，在 MVC 模式中，视图和模型有一定的关系，模型发生变化将会通过观察者模式通知视图，视图的更新，我们是通过视图主动请

求模型来完成更新的，而在 Model2 中，视图和模型是独立的，视图的更新是通过控制器来完成的，控制器根据模型的变化来通知视图的呈现及数据的变化，还有就是该

模式中，用户的操作不是通过视图来捕获的，是通过一个 web 组件，前端控制器来完成的，前端控制器负责拦截 HTTP 请求，然后根据这个请求的 URL 和 HTTP 头信息，

去决定使用哪个控制器去处理该请求。



基本上这个流程是 Model2 的处理流程。这个流

程也是 APS.NET MVC 框架背后用到的模式，相信大家对背后内容的理解，将更容易让我们在使用框架的过程中加深理解。

Model2 模式相比 MVC 模式都有什么样的优点呢?

可以肯定的是，Model2 具有更好的适应性，更好的可测试性，可维护性，并且相比原始的 MVC 模式，Model2 的效率更高。这个怎么说呢？

原始的 MVC 模式流程：我们的页面请求流程是这样的，用户的每个操作将会产生一个 HTTP 请求，然后服务器根据请求地址，将映射一个处理页面，然后该页面开

始一个生命周期，完成用户操作的请求。

Model2 模式的流程：通过上面的图形我们知道，用户的操作同样是通过 HTTP 请求，被前端控制器捕获，然后控制器将控制权交给具体的控制器去完成处理，而不

需要交给指定的页面去完成处理，这样还需要给页面创建一个生命周期，同时这种方式能够让视图做到非常的被动，而不是主动更新，我们将关注点转移到控制器中，

而不是视图上，测试时更容易测试。

我们需要注意的是 Model2 中的模型不是业务模型，也不是领域模型，这个模型是专门与视图进行交互的对象，我们叫做 ViewModel，MVC 框架中会为我们提供一

个与 ViewModel 对应的容器。容器负责创建各类的 ViewModel 对象。MVC 的流行也是因为它发挥了系统架构的原则：分离功能点的重要性，所以才会让它如此流行。

但是 MVC 还不完美，我们下面来看看改进 MVC 不足的另外一类模式 MVP。

4.3.2、MVP 模式

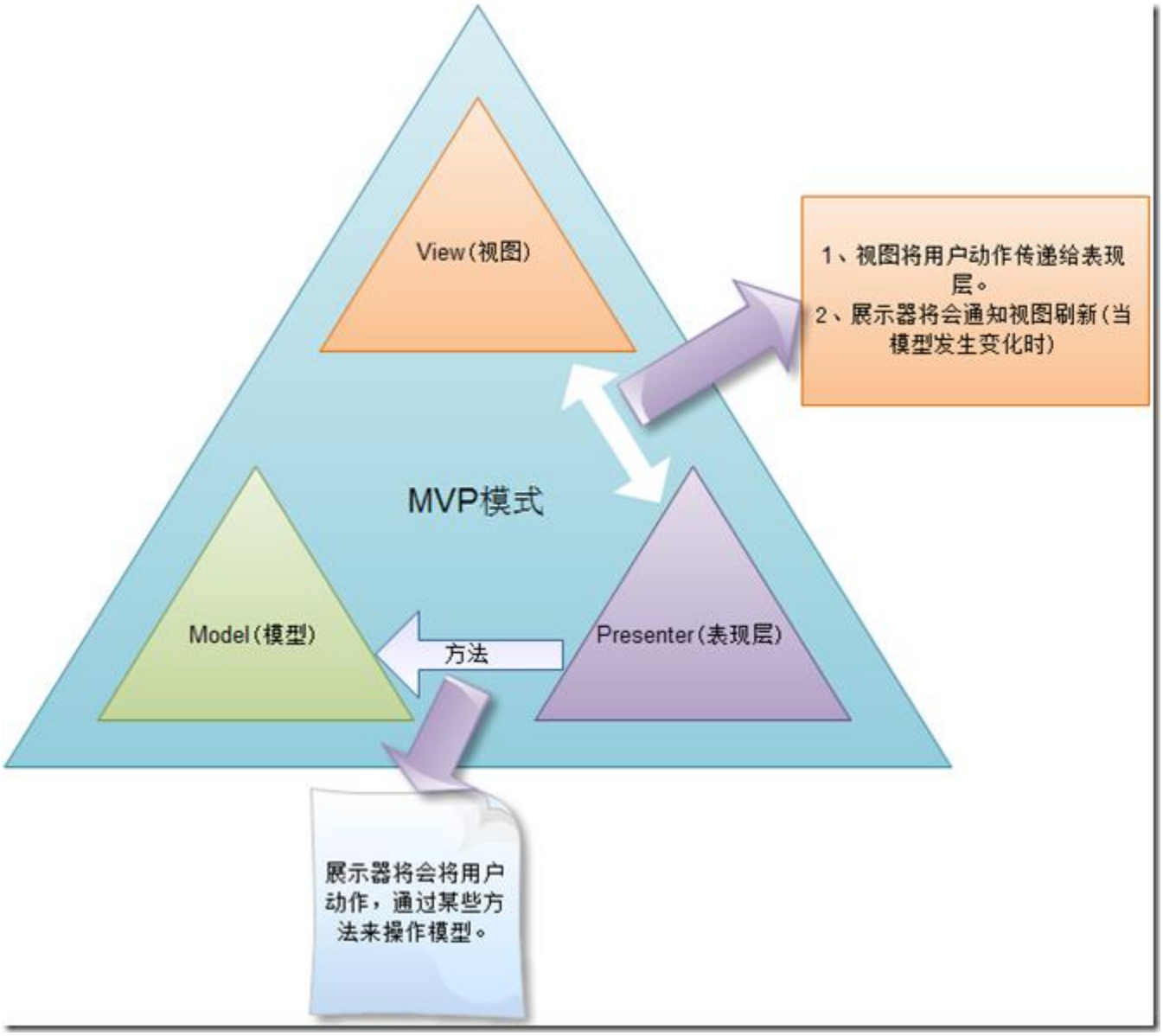
我们闲来看看 MVP 模式的解释，MVP 是将 MVC 模式中的控制器换成展示器,MVP 模式巧妙的将模型从视图/控制器中分离开来，我们将其叫做展示器，我们需要知道 MVP 模式是从 MVC 模式的基础上衍生出来，在 MVP 模式中，我们通常是这样去完成交互，展示器通过接口访问视图，这样做的目的是，展示器将不关心视图的实现形

式，只要实现接口，那么就能通过展示器来完成相应服务。当然如果我们再考虑展示器与模型之前的调用，如果也通过接口来完成，那么我们就将关注的中心放在展示

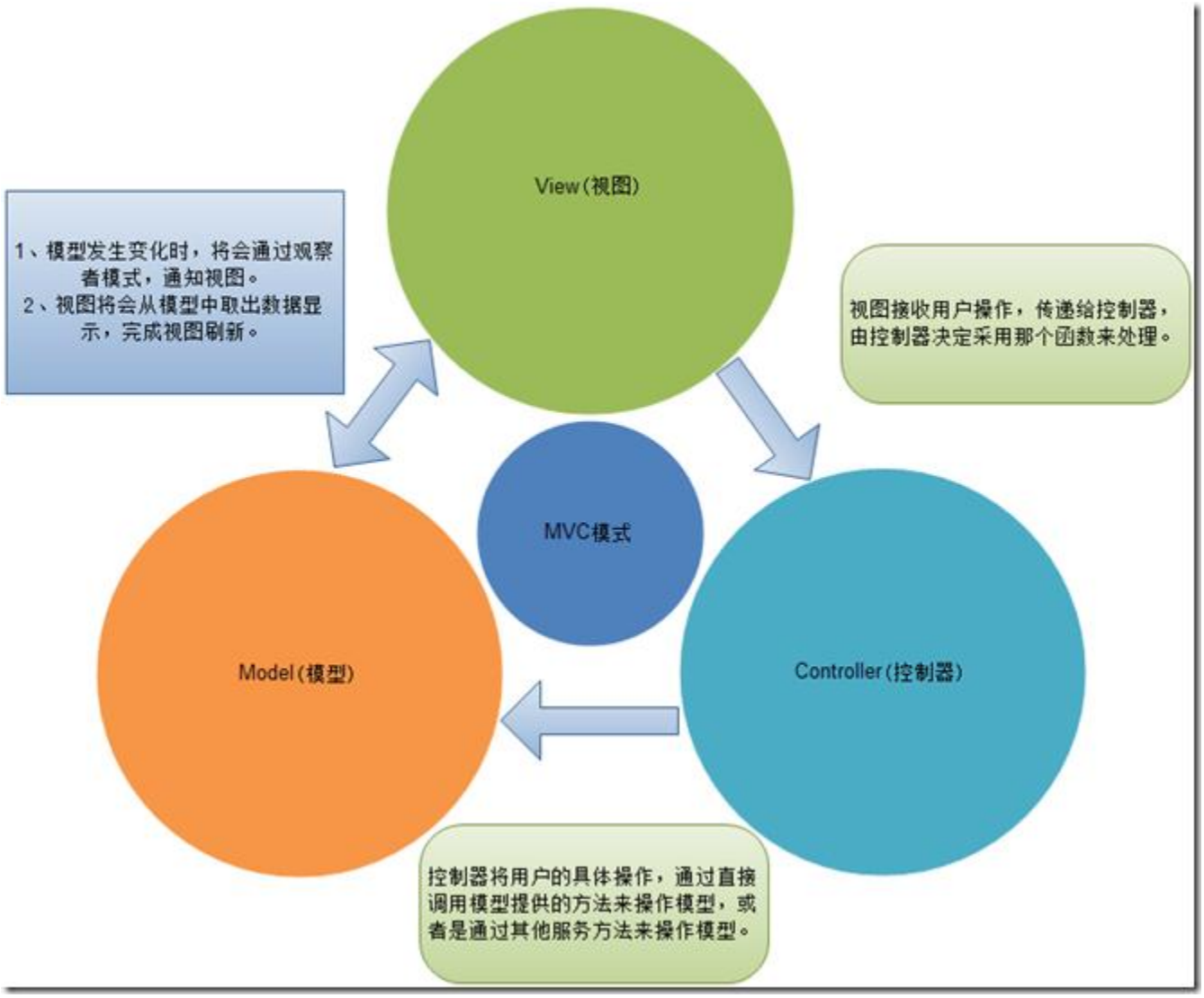
器上了,这样更容易测试,视图及模型都可以通过模拟来实现。这样将大大的提高可测试性。下面我们来看看 MVC 模式与 MVP 模式的差别，我们还是通过看图说话的形

式，这样更直观。

MVP 模式：



MVC 模式：

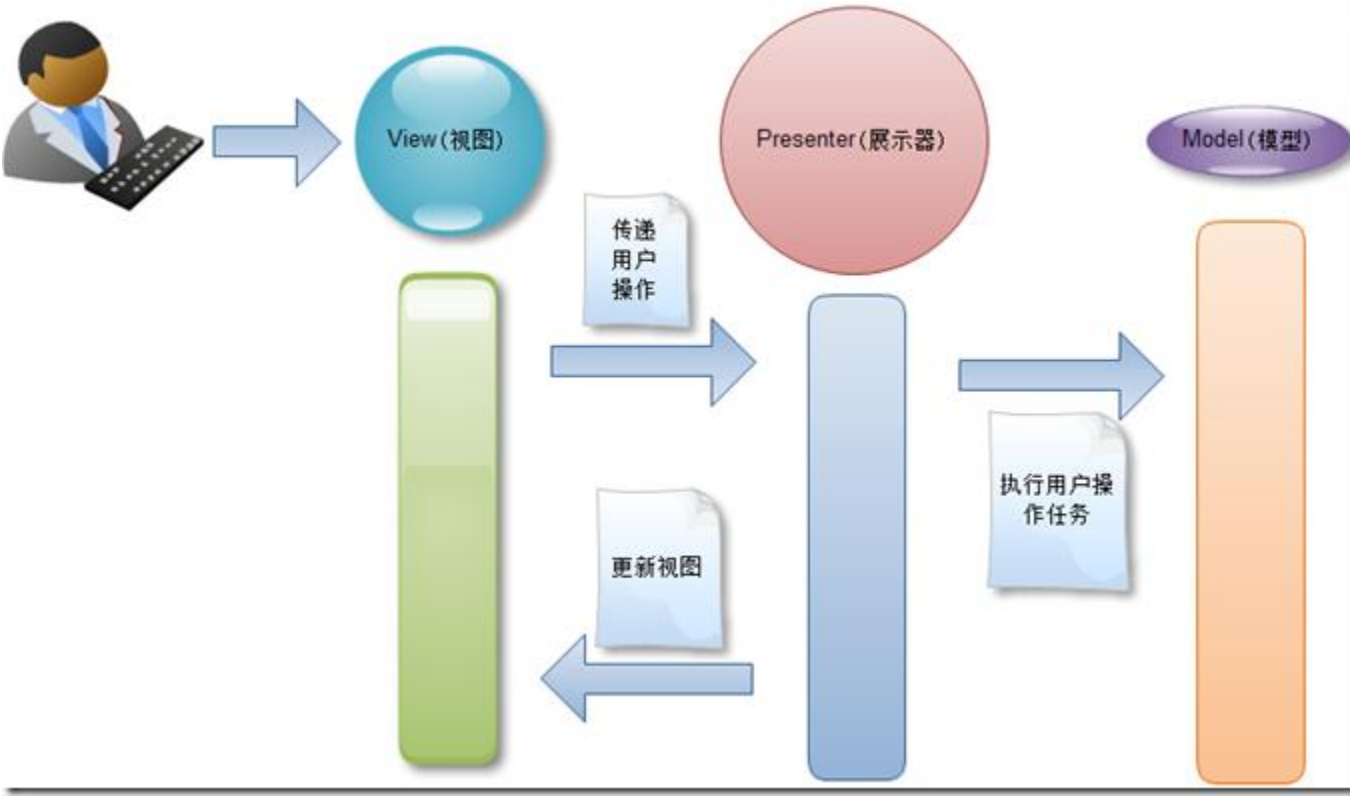


通过上面的图形，我们应该比较清楚 MVP 模式相对 MVC 模式的改进了吧？我这里就不多解释了。在 MVP 模式中，我们更关注展示器和视图之间的交互，我们使用该

模式的一个主要目的就是通过展示器与视图之间通过接口调用的形式，形成低耦合的形式，我们更关注展示器，然后不同技术实现的视图访问同一个访问器完成通用的

服务。这样的方式有点类似 SaaS 的形式，软件即是服务。

我们来看看 MVP 模式的工作流程：



其实，MVP 模式并不是一个很容易实现的模式，因为 MVP 模式需要为每个页面定义一个视图接口与展示器。当系统的页面有一定的规模时，这将是非常大的工作

量。因此我们需要根据项目的需要来决定采取的架构模式。

我们下面来看看 MVP 模式衍生出来的一类新模式：Presenter Model 模式，那么这个模式主要是应用在 WPF 中，也会叫做 Application Model ，那么这个么模式与

MVP 有什么区别呢？

我们来看看，总体来说区别不大，PM 更适合 WPF 和 silverlight 中构建表现层时采用的模式，与 MVP 模式相同，也是 3 个角色，视图、展示器、模型。

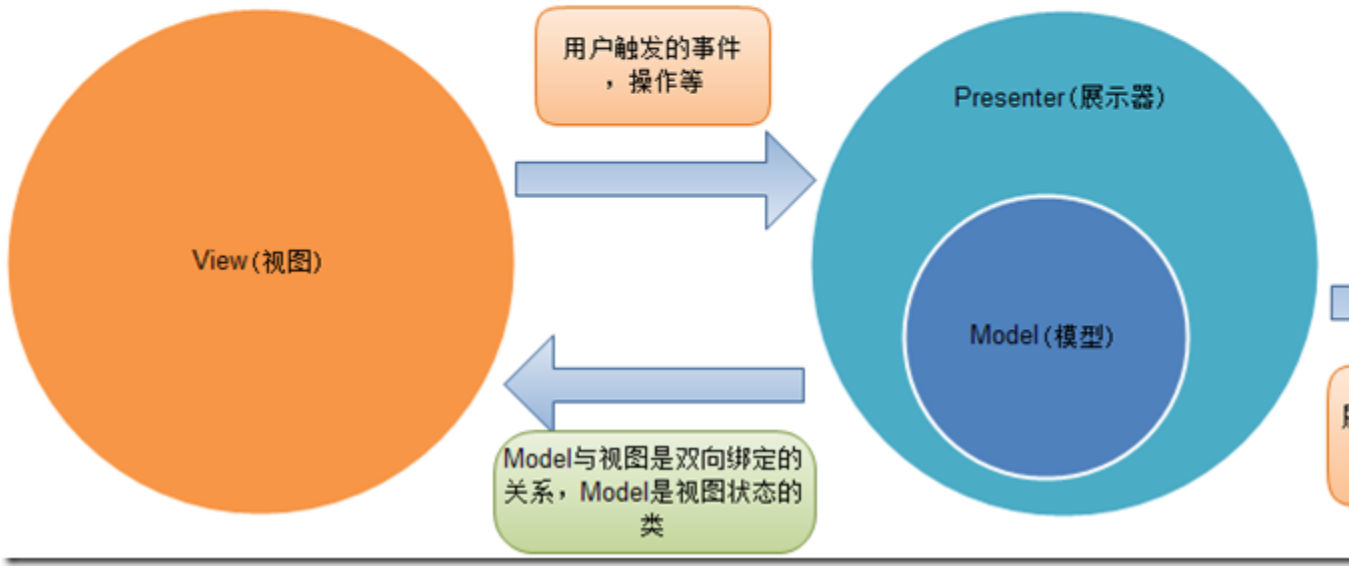
在 MVP 模式中，我们通过为视图定义接口，然后展示器通过接口调用的形式来和视图进行交互。而我们对视图的数据绑定则是，通过视图实现接口来完成的。那么

具体的实现技术可以有所不同。

在 PM 模式中，视图不会暴露任何的接口。在该模式中通过在展示器中引入与视图绑定的数据模型，通过这样的方式，视图就是被动的，当数据模型的状态发生改变

时，由于.NET FrameWork 已经提供了底层的数据绑定的同步，所以模型状态发送改变时，视图将会自动的同步更新。通过这样的双向绑定的方式来完成我们之前的

MVP 模式完成的功能，不过交互的结构发生变化，流程上也有一定的区别，我们来看看 PM 模式中 3 个角色的交互关系：



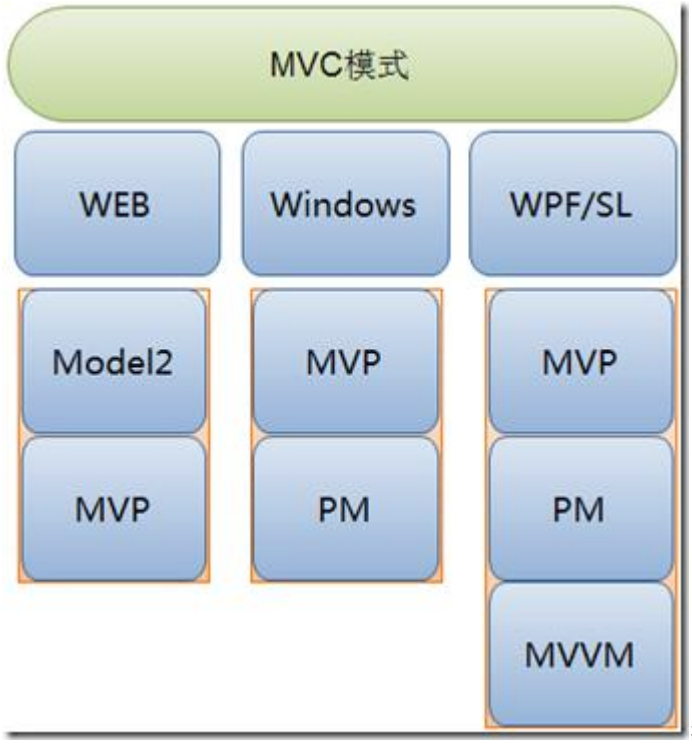
知道了这 3 个角色之间的关系，那么具体的流程是怎样呢？还是看图说话，更容易理解。

视图与模型是双向绑定，双方会自动根据某一方的变化来自动更

新，其实就是视图会被动的根据实体的变化来更新，展示器通过接收用户的动作，执行相应的业务逻辑，然后更新模型，模型发生变化，由于视图与模型绑定，那么视

图也被动变化。我们可以把这里模型就看做是数据传输对象那样的类，只是有存储数据的属性，而没有任何的行为的载体即可。PM 模式在 WPF 中该模式叫做 MVVM。

我们来总结下我们讲过的这些模式的不同 UI 的应用情形吧：



我们这里大概的总结下，不同模式的应用场景，希望能够对大家平时项目中的设计有所帮助。

五、结束语

本文主要讲述了系统架构中的表现层中的一些比较常见的基类表现层的模式，并且针对这些的模式的底层原理进行了简单的说明。这里并没有给出太多的实例代

码，是因为后面的一些实例中都会用到这里的一些模式，像 MVP，PM 的模式等，如果可以的话，我后面可以单独的开篇举例说明这些模式的应用。

六、系列进度

前篇

1、系统架构师-基础到企业应用架构系列之--开卷有益

2、系统架构师-基础到企业应用架构-系统建模[上篇]

3、系统架构师-基础到企业应用架构-系统建模[中篇](上)

4、系统架构师-基础到企业应用架构-系统建模[中篇](下)

5、系统架构师-基础到企业应用架构-系统建模[下篇]

6、系统架构师-基础到企业应用架构-系统设计规范与原则[上篇]

7、系统架构师-基础到企业应用架构-系统设计规范与原则[下篇]

8、系统架构师-基础到企业应用架构-设计模式[上篇]

9、系统架构师-基础到企业应用架构-设计模式[中篇]

10、系统架构师-基础到企业应用架构-设计模式[下篇]

中篇

11、系统架构师-基础到企业应用架构-企业应用架构

12、系统架构师-基础到企业应用架构-分层[上篇]

13、系统架构师-基础到企业应用架构-分层[中篇]

14、系统架构师-基础到企业应用架构-分层[下篇]

15、系统架构师-基础到企业应用架构-表现层

16、系统架构师-基础到企业应用架构-服务层

17、系统架构师-基础到企业应用架构-业务逻辑层

18、系统架构师-基础到企业应用架构-数据访问层

19、系统架构师-基础到企业应用架构-组件服务

20、系统架构师-基础到企业应用架构-安全机制

后篇

21、单机应用、客户端/服务器、多服务、企业数据总线全解析

22、系统架构师-基础到企业应用架构-单机应用(实例及 demo)

23、系统架构师-基础到企业应用架构-客户端/服务器(实例及 demo)

24、系统架构师-基础到企业应用架构-多服务(实例及 demo)

25、系统架构师-基础到企业应用架构-企业数据总线(实例及 demo)

26、系统架构师-基础到企业应用架构-性能优化(架构瓶颈)

27、系统架构师-基础到企业应用架构-完整的架构方案实例[上篇]

28、系统架构师-基础到企业应用架构-完整的架构方案实例[中篇]

29、系统架构师-基础到企业应用架构-完整的架构方案实例[下篇]

30、系统架构师-基础到企业应用架构-总结及后续

七、下篇预告

下一篇我们将会开始讲解系统架构中的分层-上-中-下进行相关讨论及设计方案分析，这些都是本人在工作中的经验总结，由于本人能力有限，不足之处，

还请大家多多指出。希望大家持续关注！