# 究竟什么是 MVC

经常会听说 MVC 模式，MVC 模式是设计模式么？并不是。MVC 是框架模式。框架模式不是一门写代码的学问,而是一门管理与组织代码的学问.其本质是一种软件开发的模型.与设计模式不同,设计模式是在解决一类问题时总结抽象出的公共方法(工厂模式,适配器模式,单例模式,观察者模式 … …),他们与某种具体的技术栈无关.一种框架模式往往使用了多种设计模式,切不要把他们的关系搞混。

## 为什么需要 MVC 框架模式呢？

不久前，“数据丰富的 Web 应用程序”是一个矛盾的说法。今天，这些应用程序无处不在，你需要知道如何构建它们。

传统上，Web 应用程序将繁重的数据留给服务器，这些服务器在完整的页面加载中将 HTML 推送到浏览器。客户端 JavaScript 的使用仅限于改善用户体验。现在这种关系已经被颠倒了 - 客户端应用程序从服务器中提取原始数据，并在需要的时间和地点将其呈现在浏览器中。

想想在向购物篮添加商品时不需要在页面上刷新的 Ajax 购物车。最初，jQuery 成为这个范例的首选库。它的本质是制作 Ajax 请求然后更新页面上的文本等等。但是，使用 jQuery 的这种模式显示我们在客户端有隐式模型数据。由于服务器不再是唯一了解我们项目数量的地方，因此暗示了这种演变的自然紧张和拉动。

客户端的任意代码的增加可以与服务器通信但是它认为合适意味着客户端复杂性的增加。客户端上的良好架构已经从事后的想法变为必不可少的 - 您不能只是将一些 jQuery 代码混合在一起并期望它随着应用程序的增长而扩展。最有可能的是，你最终会遇到与业务逻辑交织在一起的噩梦般的 UI 回调，注定会被继承你代码的可怜的灵魂抛弃。

值得庆幸的是，有越来越多的 JavaScript 库可以帮助改进代码的结构和可维护性，从而可以更轻松地构建雄心勃勃的接口而无需花费大量精力。

接下来通过代码逻辑来看看我们写的代码怎么了？

阶段一：脚本式设计(无架构设计)

```
const a = document.createElement("a");
a.innerHTML = "www.baidu.com";
a.href = "//www.baidu.com";
a.style.position = "absolute";
a.style.top = 100;
a.onclick = () => {
        alert("baidu");
 };
 document.body.appendChild(a);
```

这样的代码,就是无任何设计模式的产物。短短的几行代码,包含了创建,样式,绑定,插入. balabala.....这种搞法虽有不少缺点,但麻雀虽小五脏俱全,所有功能一应俱全.早些年由于 UI 程序还处在一个懵懂期, 逻辑不算太复杂,代码量也不会太多.这样的搞法似乎也没有什么问题. 毕竟到达 A B 两点最短的距离就是直线,上述代码可以说是实现某功能的最短路径. 典型的例子就是 ASM (虽然汇编语言不是用来写 UI 的),他们共有的缺点是 ：

**入口单一 功能简单 不可维护。**

对于以上的内容进行修改一下：

```
function doCss(a) {
    a.style.position = "absolute";
    a.style.top = 100;
}
function doEvent(a) {
    a.onclick = () => {
        alert("baidu");
    };
}
function doAttribute(a){
    a.innerHTML = "www.baidu.com";
    a.href = "//www.baidu.com";
}
const a = document.createElement("a");
doCss(a);
doEvent(a);
doAttribute(a)
```

```
    document.body.appendChild(a);
```

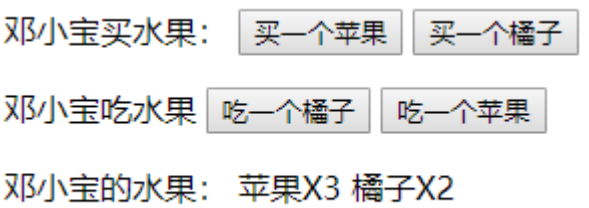
这里我们将一个功能拆分成了 3 个部分,即 外观 事件 和属性.函数将他们重新分离成一个个独立的逻辑块,这样一定程度上达到了分离复用的目的,比如你想修改外观,就去 doCss 函数里去找...就像有钱人追求更多的财富,权贵追求更多权利一样.人类总在思考同一个问题, 我们能不能做得更好?

第二个阶段：代码文件分离

```
<html>
<head>
    <link href="style.css" rel="stylesheet" />
</head>
<body>
    <script src="bundle.js" ></script>
</body>
</html>
```

各文件各司其职,编写的时候分离,在运行的时候合并.这样进一步降低了功能之间的耦合度. 视图看起来非常"清爽",对应的逻辑也被分离成一个个文件, 交由相应的开发人员处理。如果你涉猎的技术范围很广,你会发现其实已经出现在诸多成熟的技术栈中。

在接下来的这种应用场景就会出现问题。




我们想实现这样的一个效果。最基本的思路是：

`购买苹果按钮`绑定事件如下:

```
    苹果数变量 + 1;
    显示苹果数控件的值 = 苹果数变量;
```

`购买橘子按钮`绑定事件如下:

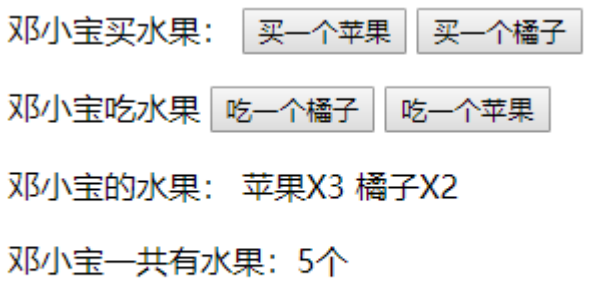
```
    橘子数变量 + 1;
    显示橘子数控件的值 = 橘子数变量;
```

...`吃苹果`

...`吃橘子`

就是如果显示苹果数的控件是另一个程序员开发，如何修改其值?
于是程序员 A 去找 程序员 B 寻求是否存在对应的 get/set 方法.程序员 B 说有,"有" 字还没落地，开发买梨的程序员 C 又踹门进来了,问了同样的问题，后来才知道 开发吃苹果功能的程序员 D 正在路上...于是程序员 B 不得不把接口的详细信息写到 wiki 中，于是 程序员 CDEFGHIJKMLN 都看了 wiki 懂了.完成这件事 程序员 B 写 wiki 花了 10 分钟，程序员 CDEFGHIJKMLN 看 wiki 每人花 1 分钟,一共团队成本 20 分钟.于是 wiki 中这个 get/set 接口函数出现在了每一个被绑定的函数里.一共 4 个 button 出现 4 次。
ps: 这个 get/set 接口本质上就是一个 view 刷新接口。

产品大爷发话了,要改成下面这样:



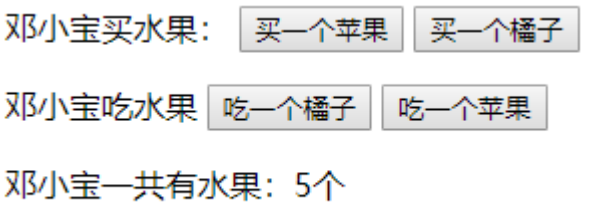

多了一个求和.于是思想变成了这样:

`购买苹果按钮`绑定事件如下:

```
苹果数变量 + 1;
总和变量 = 苹果数变量 + 梨数变量;
显示苹果数控件的值 = 苹果数变量;
显示总和控件的值 = 总和变量;
```

这次 程序员 B 为了人身安全,提前把 get/set 接口发布到了 wiki 上..
于是 总和控件的值 = 总和变量 这行代码出现了 4 次。
产品存在的意义,就是将程序狗虐到极致。

需求又有变动了：




去掉了一行的显示内容，于是 4 个函数中 所有相关代码都被删除...

**其核心问题在于,按照事件进行的业务模块划分,往往是不合理的,一个 button 的 click 可能横跨 N 个领域, N 个 UI 区域,这些在前期都是不可预见的(俗称 坑),这部分 逻辑到最后还是会耦合在一起,通过各种函数封装进行解耦,无疑是扬汤止沸,而我们需要的釜底抽薪。**

接下来说的就是我们的 MVC 模式了。

## 什么是 MVC 模式?

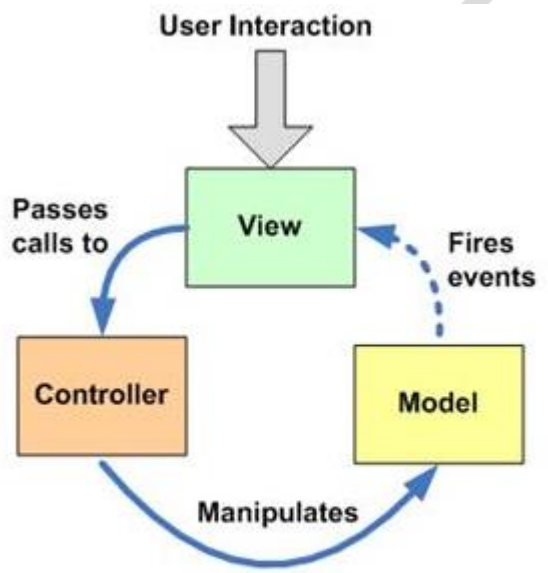
现在网上有很多关于 mvc 的介绍,让人纠结的是他们各不相同,而且有的根本就说的不对, 对于框架模式这东西,没有一个严格的规定说这样搞是 mvc 那样就不是. 甚至连 mvc 本身也有很多变种,我们只要从根源上理解这个东西就行.我就不扒祖坟了,咱们只需要知道它已经存在了 30 多年就行了.我们思考一下 UI(图形化用户界面) 的本质:为什么要有 UI, 在计算机眼中 一切即数据,其实要是深挖这个问题,数据与操作其实都是 0 1 组成的机器码,只不过 CPU 运行的时候用指定寄存器的数据当做指令罢了,也就是说 决定一个数据到底是数据还是指令 只取决于他所在的寄存器位置.数据的操作是抽象的,是专业人士干的事情, 计算机为了走进千家万户, 必须提供一种傻瓜式的操作方式,于是 UI 诞生了... 用一句话解释 UI 就是:他是数据到图像的一种映射程序;刚才说了它是一种映射程序,用户通过操作图像上的按钮,来达到操作数据的目的,数据被用户改变后,肯定需要从新生成映射。

许多现代 JavaScript 框架为开发人员提供了一种使用称为 MVC（模型 - 视图 - 控制器）模式的变体组织代码的简便途径。MVC 将应用程序中的问题分为三个部分：

模型表示应用程序中特定于域的知识和数据。可以将其视为您可以建模的“类型”数据 - 例如用户，照片或待办事项。模型可以在状态发生变化时通知观察者。
视图通常构成应用程序中的用户界面（例如，标记和模板），但不一定如此。他们观察模型，但不直接与它们通信。
控制器处理输入（例如，点击，用户操作）和更新模型。
因此，在 MVC 应用程序中，更新模型的控制器对用户输入起作用。视图观察模型并在发生更改时更新用户界面。

JavaScript MVC 框架并不总是严格遵循上述模式。一些解决方案（例如 Backbone.js）将 Controller 的职责合并到 View 中，而其他方法则将其他组件添加到组合中。

先上一张图：




Model-View-Controller

说说这里面 Modle View Controller 是干什么的:
1.View: 放置视图相关的代码,原则上里面不应该有任何业务逻辑.
2.Controller: 放置视图与模型之间的映射,原则上这里应该很薄,他只放一

些事件绑定相关的代码(router),但并不实现真正的功能,他只是一个桥梁.
3.Modle: 这里的 modle 不是说 实体类, 他是主要实现逻辑的地方.

那还是上面 买水果的例子,那么在 MVC 下该如何设计呢:
view 层放置界面代码,以及一些刷新逻辑 如数据中的 0 1 转成 男 女
controller 层放置一些绑定逻辑.完成 router,不实现函数体。
model 层接收 view 的注册,当自身数据变化时,执行 view 的刷新函数. 业务逻辑都在这里。

他是这样一个流程:

1.创建显示苹果数量的控件。

2.将上面控件注册到 model 中。(设置关联的数据,--苹果数变量)

3.修改 model 中 苹果数变量 。

4.由于苹果数变量被修改,触发所有绑定在上面的控件(view)从新执行刷新函数。

5.显示苹果数量的控件被更新。

## 简单的实现一个 JavaScript MVC 框架

MVC 的基础是观察者模式，这是实现 model 和 view 同步的关键
为了简单起见，每个 model 实例中只包含一个 primitive value 值。

```
// JavaScript 代码
function Model(value) {
    this._value = typeof value === 'undefined' ? '' : value;
    this._listeners = [];
}
Model.prototype.set = function (value) {
    var self = this;
    self._value = value;
    // model 中的值改变时，应通知注册过的回调函数
    // 按照 Javascript 事件处理的一般机制，我们异步地调用回调函数
    // 如果觉得 setTimeout 影响性能，也可以采用 requestAnimationFrame
    setTimeout(function () {
        self._listeners.forEach(function (listener) {
            listener.call(self, value);
```

```
            });
        });
};
Model.prototype.watch = function (listener) {
        // 注册监听的回调函数
        this._listeners.push(listener);
};

// html
<div id=”div1”></div>
// JavaScript
(function () {
        var model = new Model();
        var div1 = document.getElementById('div1');
        model.watch(function (value) {
            div1.innerHTML = value;
        });
        model.set('hello, this is a div');
})();
```

借助观察者模式，我们已经实现了在调用 model 的 set 方法改变其值的时候，模板也同步更新，但这样的实现却很别扭，因为我们需要手动监听 model 值的改变（通过 watch 方法）并传入一个回调函数，有没有办法让 view（一个或多个 dom node）和 model 更简单的绑定呢？

**实现 bind 方法，绑定 model 和 view**

```
Model.prototype.bind = function (node) {
        // 将 watch 的逻辑和通用的回调函数放到这里
        this.watch(function (value) {
            node.innerHTML = value;
        });
};

// html：
<div id="div1"></div>
<div id="div2"></div>
// JavaScript
```

```
(function () {
    var model = new Model();
    model.bind(document.getElementById('div1'));
    model.bind(document.getElementById('div2'));
    model.set('this is a div');
})();
```

通过一个简单的封装，view 和 model 之间的绑定已经初见雏形，即使需要在一个 model 上绑定多个 view，实现起来也很轻松。虽然绑定的复杂度降低了，这一步依然要依赖我们手动完成，有没有可能把绑定的逻辑从业务代码中彻底解耦呢？

**实现 controller，将绑定从逻辑代码中解耦**

细心的朋友可能已经注意到，虽然讲的是 MVC，但是上文中却只出现了 Model 类，View 类不出现可以理解，毕竟 HTML 就是现成的 View（事实上本文中从始至终也只是利用 HTML 作为 View，javascript 代码中并没有出现过 View 类），那 Controller 类为何也隐身了呢？别急，其实所谓的"逻辑代码"就是一个框架逻辑和业务逻辑耦合度很高的代码段，现在我们就来将它分解一下。

如果要将绑定的逻辑交给框架完成，那么就需要告诉框架如何来完成绑定。由于 JS 中较难完成 annotation（注解），我们可以在 view 中做这层标记——使用 html 的标签属性就是一个简单有效的办法。

```
// JavaScript
function Controller(callback) {
    var models = {};
    // 找到所有有 bind 属性的元素
    var views = document.querySelectorAll('[bind]');
    // 将 views 处理为普通数组
    views = Array.prototype.slice.call(views, 0);
    views.forEach(function (view) {
        var modelName = view.getAttribute('bind');
        // 取出或新建该元素所绑定的 model
        models[modelName] = models[modelName] || new Model();
        // 完成该元素和指定 model 的绑定
        models[modelName].bind(view);
    });
    // 调用 controller 的具体逻辑，将 models 传入，方便业务处理
```

```
        callback.call(this, models);
}

// html:
<div id="div1" bind="model1"></div>
<div id="div2" bind="model1"></div>
//JavaScript：
new Controller(function (models) {
        var model1 = models.model1;
        model1.set('this is a div');
});
```

**小例子：利用 MVC 方式实现一个电子表**

```
<span bind="hour"></span> : <span bind="minute"></span>
: <span bind="second"></span>

        <script>
                function Model(value) {
                        this._value = typeof value === 'undefined' ? '' : value;
                        this._listeners = [];
                }
                Model.prototype.set = function (value) {
                        var self = this;
                        self._value = value;
                        setTimeout(function () {
                                self._listeners.forEach(function (listener) {
                                        listener.call(self, value);
                                });
                        });
                };
                Model.prototype.watch = function (listener) {
                        this._listeners.push(listener);
                };
                Model.prototype.bind = function (node) {
                        this.watch(function (value) {
```

```
                node.innerHTML = value;
            });
        };
        function Controller(callback) {
            var models = {};
            var views = Array.prototype.slice.call(document.querySelectorAll('[bind]'), 0);
            views.forEach(function (view) {
                var modelName = view.getAttribute('bind');
                (models[modelName] = models[modelName] || new Model()).bind(view);
            });
            callback.call(this, models);
        }

        new Controller(function (models) {
        function setTime() {
                var date = new Date();
                models.hour.set(date.getHours());
                models.minute.set(date.getMinutes());
                models.second.set(date.getSeconds());
            }
            setTime();
            setInterval(setTime, 1000);
        });

    </script>
```