MVC 的弊端是什么，该如何解决

MVC 框架有怎么样的弊端？

先看一个例子（加减计数器）：

Model 层：
Model 层用于封装和应用程序的业务逻辑相关的数据以及对数据的处理方法。这里我们把需要用到的数值变量封装在 Model 中，并定义了 add、sub、getVal 三种操作数值方法。Model 层用来存储业务的数据，一旦数据发生变化，模型将通知有关的视图。

```
// JS 代码
        var myapp = {}; //  创建这个应用对象
        myapp.Model = function () {
            var val = 0;
            this.add = function (v) {
                if (val < 100) {
                    val += v;
                }
            };

            this.sub = function (v) {
                if (val > 0) {
                    val -= v;
                }
            };

            this.getVal = function () {
                return val;
            };
```

```
        //  观察者模式
        var self = this,
            views = [];

        this.register = function (view) {
            views.push(view);
        };

        this.notify = function () {
            for (var i = 0; i < views.length; i++) {
                views[i].render(self);
            }
        };
    };
```

Model 和 View 之间使用了观察者模式，View 事先在此 Model 上注册，进而观察 Model，以便更新在 Model 上发生改变的数据。

View 层：

View 作为视图层，主要负责数据的展示。view 和 controller 之间使用了策略模式，这里 View 引入了 Controller 的实例来实现特定的响应策略，比如这个例子中按钮的 click 事件：

```
myapp.View = function(controller) {
    var $num = $('#num'),
        $incBtn = $('#increase'),
        $decBtn = $('#decrease');

    this.render = function(model) {
        $num.text(model.getVal() + 'rmb');
    };

    /*  绑定事件  */
    $incBtn.click(controller.increase);
    $decBtn.click(controller.decrease);
};
```

现在通过 Model&View 完成了数据从模型层到视图层的逻辑。但对于一个应用程序，这远远是不够的，我们还需要响应用户的操作、同步更新 View 和 Model。于是，在 MVC 中加入了控制器 controller，让它来定义用户界

面对用户输入的响应方式，它连接模型和视图，用于控制应用程序的流程，处理用户的行为和数据上的改变。如果要实现不同的响应的策略只要用不同的 Controller 实例替换即可。

```
myapp.Controller = function() {
    var model = null,
        view = null;

    this.init = function() {
        /*  初始化 Model 和 View */
        model = new myapp.Model();
        view = new myapp.View(this);

        /* View 向 Model 注册，当 Model 更新就会去通知 View */
        model.register(view);
        model.notify();
    };

    /*  让 Model 更新数值并通知 View 更新视图  */
    this.increase = function() {
        model.add(1);
        model.notify();
    };

    this.decrease = function() {
        model.sub(1);
        model.notify();
    };
};
```

这里我们实例化 View 并向对应的 Model 实例注册，当 Model 发生变化时就去通知 View 做更新，这里用到了观察者模式。

当我们执行应用的时候，使用 Controller 做初始化：

```
(function() {
    var controller = new myapp.Controller();
    controller.init();
})();
```

可以明显感觉到，MVC 模式的业务逻辑主要集中在 Controller，而前端的 View 其实已经具备了独立处理用户事件的能力，主要问题在于：**当每个事**

件都流经 Controller 时，这层会变得十分臃肿。而且 MVC 中 View 和 Controller 一般是一一对应的，捆绑起来表示一个组件，视图与控制器间的过于紧密的连接让 Controller 的复用性成了问题。

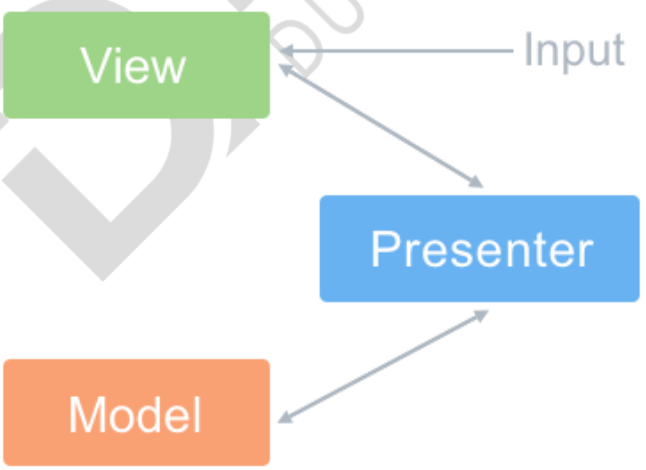
总结一下：

Controller 测试困难。因为视图同步操作是由 View 自己执行，而 View 只能在有 UI 的环境下运行。在没有 UI 环境下对 Controller 进行单元测试的时候，应用逻辑正确性是无法验证的：Model 更新的时候，无法对 View 的更新操作进行断言。

View 无法组件化。View 是强依赖特定的 Model 的，如果需要把这个 View 抽出来作为一个另外一个应用程序可复用的组件就困难了。因为不同程序的的 Domain Model 是不一样的

MVC 模式变种一：MVP 模式（了解）

王者荣耀 MVP 了解一下。不扯了，这里要说的一个改进的模式被称为 MVP 模式。MVP（model-view-Presenter）是经典 MVC 设计模式的一种衍生模式，是在 1990 年代 Taligent 公司创造的，一个用于 C++ CommonPoint 的模型。背景上不再考证，直接上图看一下与 MVC 的不同。



和 MVC 模式一样，用户对 View 的操作都会从 View 交移给 Presenter。Presenter 会执行相应的应用程序逻辑，并且对 Model 进行相应的操作；而这时候 Model 执行完业务逻辑以后，也是通过观察者模式把自己变更的消息传递出去，但是是传给 Presenter 而不是 View。Presenter 获取到 Model 变更的消息以后，通过 View 提供的接口更新界面。

关键点：

View 不再负责同步的逻辑，而是由 Presenter 负责。Presenter 中既有应用程序逻辑也有同步逻辑。

View 需要提供操作界面的接口给 Presenter 进行调用。（关键）

对比在 MVC 中，Controller 是不能操作 View 的，View 也没有提供相应的接口；而在 MVP 当中，Presenter 可以操作 View，View 需要提供一组对界面操作的接口给 Presenter 进行调用；Model 仍然通过事件广播自己的变更，但由 Presenter 监听而不是 View。

MVP 优缺点

优点：

便于测试。Presenter 对 View 是通过接口进行，在对 Presenter 进行不依赖 UI 环境的单元测试的时候。可以通过 Mock 一个 View 对象，这个对象只需要实现了 View 的接口即可。然后依赖注入到 Presenter 中，单元测试的时候就可以完整的测试 Presenter 应用逻辑的正确性。这里根据上面的例子给出了 Presenter 的单元测试样例。

View 可以进行组件化。在 MVP 当中，View 不依赖 Model。这样就可以让 View 从特定的业务场景中脱离出来，可以说 View 可以做到对业务完全无知。它只需要提供一系列接口提供给上层操作。这样就可以做到高度可复用的 View 组件。

缺点：

Presenter 中除了应用逻辑以外，还有大量的 View->Model，Model->View 的手动同步逻辑，造成 Presenter 比较笨重，维护起来会比较困难。 ‘’

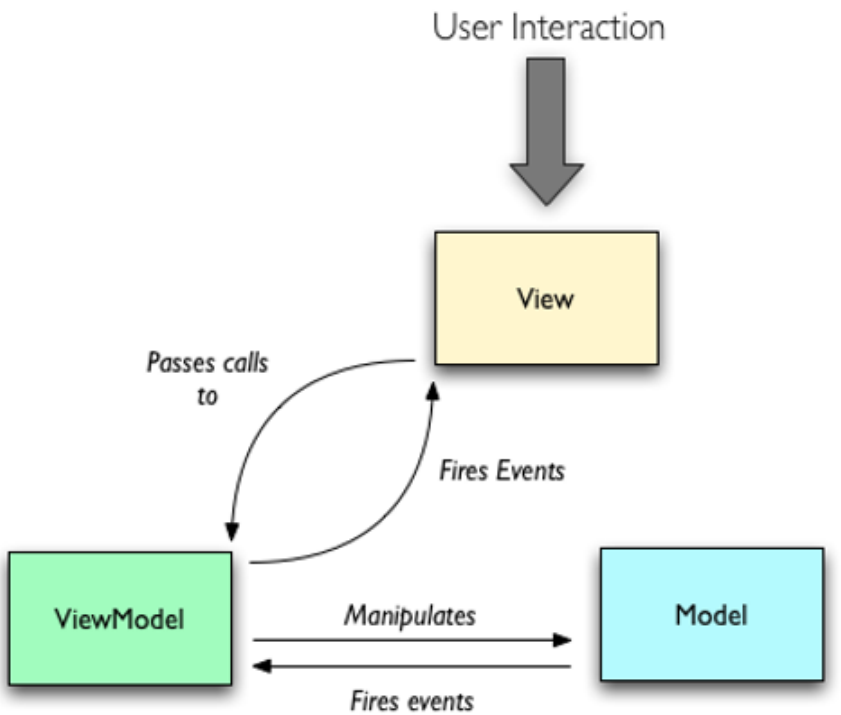
Web 前端的 MVP 案例：Jsviews（Web 前端采用 MVP 架构模式设计的框架相对较少，而在 Android 的开发中相对较多，例如 MVPArms、MvpApp 和 Android-tech-frontier 等）。

MVP 作为对于 MVC 的变种模式之一。在 Android 中使用较多，在前端领域内用的确实不多，大家了解一下即可。

MVC 模式变种二：MVVM 模式

什么是 MVVM 模式?

Model-View-ViewModel 就是将其中的 View 的状态和行为抽象化，让我们可以将 UI 和业务逻辑分开。当然这些工作 ViewModel 已经帮我们做了，它可以取出 Model 的数据同时帮忙处理 View 中由于需要展示内容而涉及的业务逻辑。



MVVM 模式是通过以下三个核心组件组成，每个都有它自己独特的角色：
Model - 包含了业务和验证逻辑的数据模型
View - 定义屏幕中 View 的结构，布局和外观
ViewModel - 扮演“View”和“Model”之间的使者，帮忙处理 View 的全部业务逻辑。
那这和我们曾经用过的 MVC 模式有什么不同呢？以下是 MVC 的结构

View 在 Controller 的顶端，而 Model 在 Controller 的底部
Controller 需要同时关注 View 和 Model
View 只能知道 Model 的存在并且能在 Model 的值变更时收到通知
MVVM 模式和 MVC 有些类似，但有以下不同：

ViewModel 替换了 Controller，在 UI 层之下
ViewModel 向 View 暴露它所需要的数据和指令对象
ViewModel 接收来自 Model 的数据

你可以看到这两种模式有着相似的结构，但新加入的 ViewModel 是用不同的方法将组件们联系起来的，它是双向的，而 MVC 只能单向连接。

MVVM 模式是在 MVP 模式的基础上进行了改良，在 MVP 中，派发器 P 需要手动调用 View 组件化后的接口才能实现视图的更新，而在 MVVM 模式中，将 P 改良成抽象视图（ViewModel）。ViewModel 中有一个 Binder（ Data-binding Engine），在 Presenter 中负责的 View 和 Model 的数据同步逻辑交由 Binder 处理（注意不包括应用逻辑）。
注意点：MVVM 实现了 View 和 Model 的同步逻辑自动化。MVP 中 Presenter 负责的 View 和 Model 同步需要手动进行操作，但是在 MVVM 中同步逻辑交由 Binder 进行负责（一旦 Model 发生变化，Binder 会对 Model 对应的数据进行变化监听，并实现相应视图的更新）。
优点：提高可维护性、简化测试
缺点：简单 GUI 应用会产生额外的性能损耗、复杂 GUI 的 ViewModel 构建和维护成本高
Web 前端的 MVVM 案例： Vue、React。

接下来的文章我们深入探究一下 MVVM 模式案例的 Vue ，在这样的框架中中体验 MVVM 模式给我们带来的快感。