大前端

迷渡
@justjavac
https://github.com/justjavac

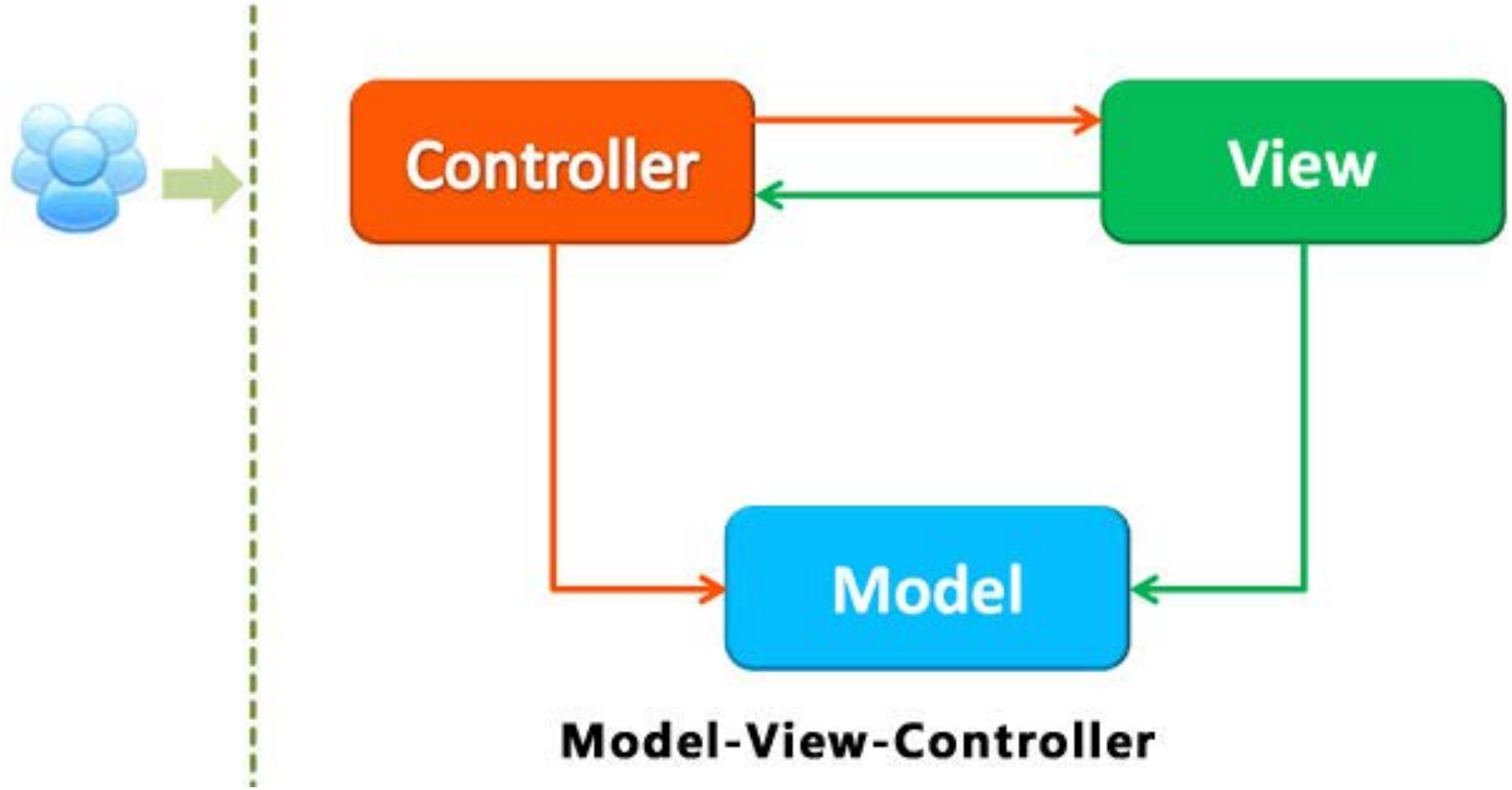
前端技术及框架的演进

每个框架都在尝试解决这样一些问题

- 在哪里（or 怎样）去维护 UI 当中的状态?
- 业务逻辑在应用的什么地方? 怎样被调用?
- 怎样保证 UI 跟数据的改变同步? 还有 UI 元素之间相互同步?
- 怎样保证对我们关心的代码做分离, 来降低可测试代码的耦合?

MVC

MVC



- 视图（View）：用户界面。
- 控制器（Controller）：业务逻辑
- 模型（Model）：数据保存

MVC的流程

- View（界面）触发事件
- --》Controller（业务）处理了业务，然后触发了数据更新
- --》更新Model的数据
- --》Model（带着数据）回到了View
- --》View更新数据

缺点

- 在MVC，当你有变化的时候你需要同时维护三个对象和三个交互，这显然让事情复杂化了

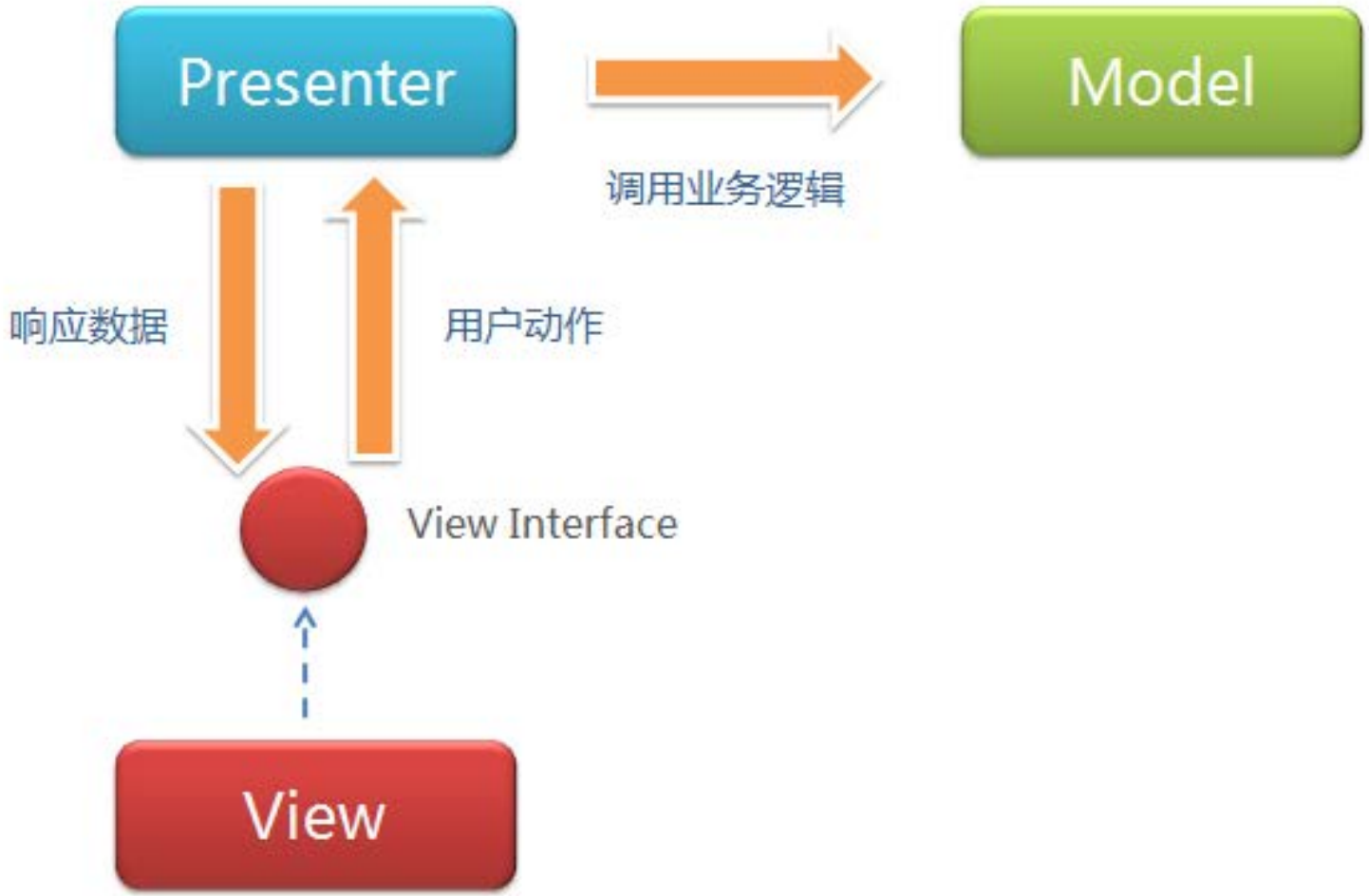
实际项目往往采用更灵活的方式

实例：Backbone

- 用户可以向 View发送指令（DOM 事件），再由 View直接要求 Model 改变状态。
- 用户也可以直接向 Controller发送指令（改变 URL 触发 hashChange 事件），再由 Controller发送给 View。
- Controller非常薄，只起到路由的作用，而 View非常厚，业务逻辑都部署在 View。所以，Backbone索性取消了 Controller，只保留一个 Router（路由器）。

MVP

前端的 MVP



改进了MVC

- 切断的View和Model的联系，让View只和Presenter（原Controller）交互，减少在需求变化中需要维护的对象的数量。

特点

- 各部分之间的通信，都是双向的。
- View与 Model不发生联系，都通过 Presenter传递。
- View非常薄，不部署任何业务逻辑，称为"被动视图"（Passive View），即没有任何主动性，而 Presenter非常厚，所有逻辑都部署在那里。

MVVM

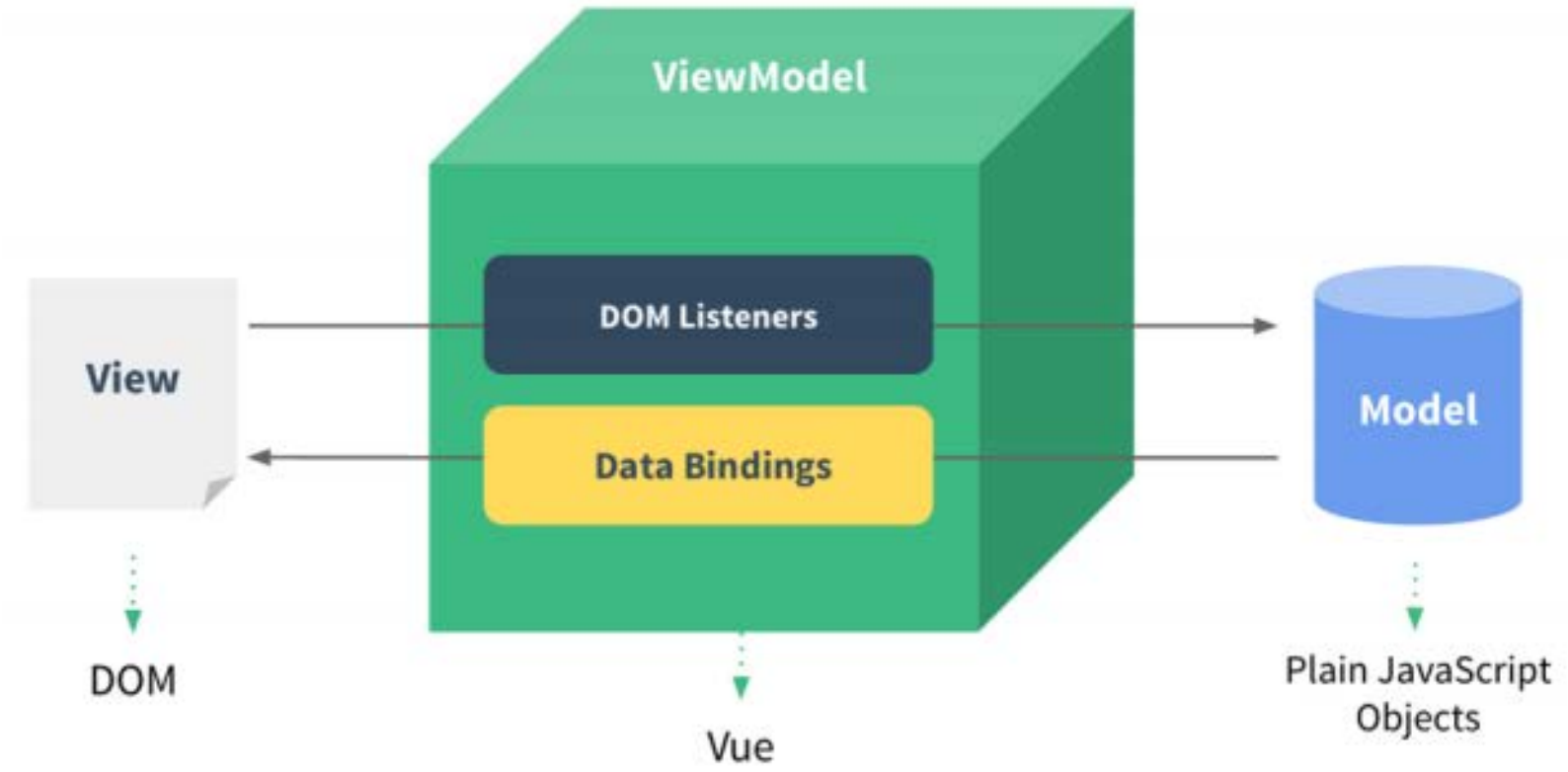
特点

- 双向绑定技术
- 自动同步更新

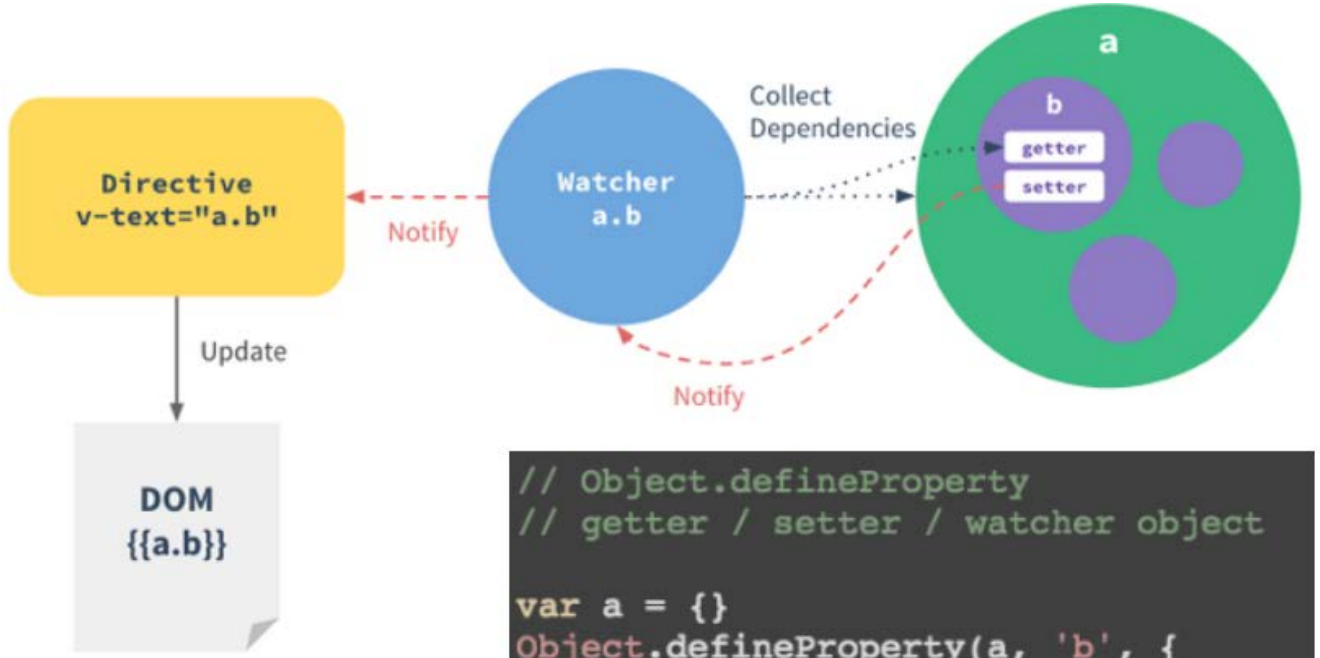
缺点

- 数据绑定使得 Bug 很难被调试
- 对于过大的项目，数据绑定需要花费更多的内存

vue的mvvm


ViewModel
DOM Listeners
Data Bindings
View
Model
DOM
Vue
Plain JavaScript
Objects

实现原理




```
// Object.defineProperty
// getter / setter / watcher object

var a = {}
Object.defineProperty(a, 'b', {
  enumerable: false,
  value: "static",
  get: () => { ... },
  set: () => { ... }
})
```

- ViewModel为View提供了一个访问Model的桥梁，但是View不是直接访问Model层。ViewModel为View提供了Model的特定于View的子集，可以包含状态和逻辑信息，而且无需向View暴露整个Model。
- View和ViewModel通过数据绑定和事件可以进行通信，因为ViewModel可以访问Model层，所以ViewModel为了数据绑定要暴露Model中的部分属性。

- 只要我们有一个JS的对象Model，修改它，页面就跟着改（Model -> View的单向绑定），并且满足Model相同页面就相同，这个框架就是一个MVVM的框架了

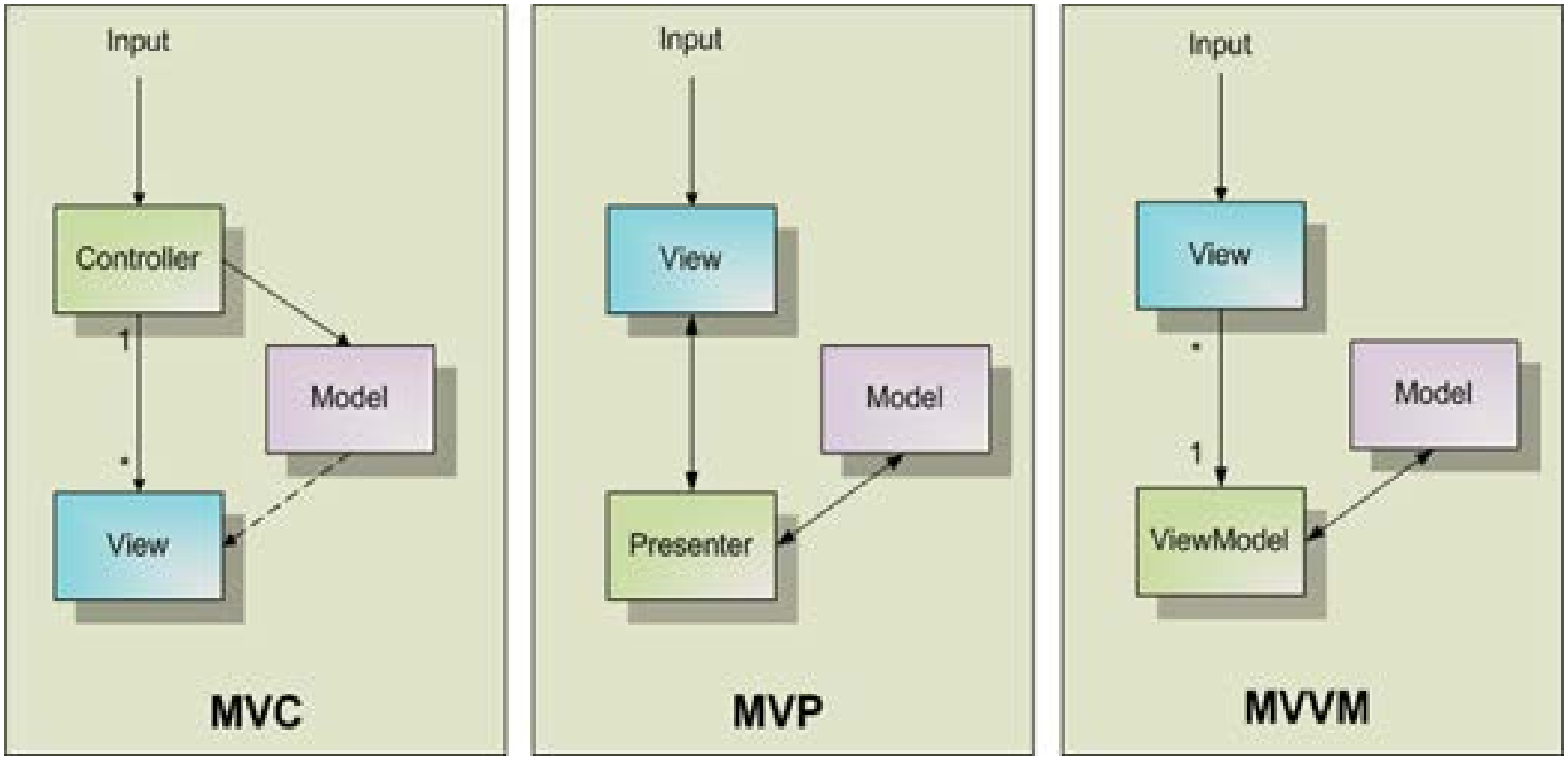
```
var tpl = '<p>这是一个任何模板引擎都可以用的模板：姓名：${name}</p>';

var model = {
    name: '张三'
};

var container = document.getElementById('container');
model = new ViewModel(container, model, tpl);

function ViewModel(elem, model, tpl) {
    var engine = new Engine(tpl);
    var vm = observer(model).on('change', function () {
        update(elem, engine.render(model));
    });
    return vm;
}
```

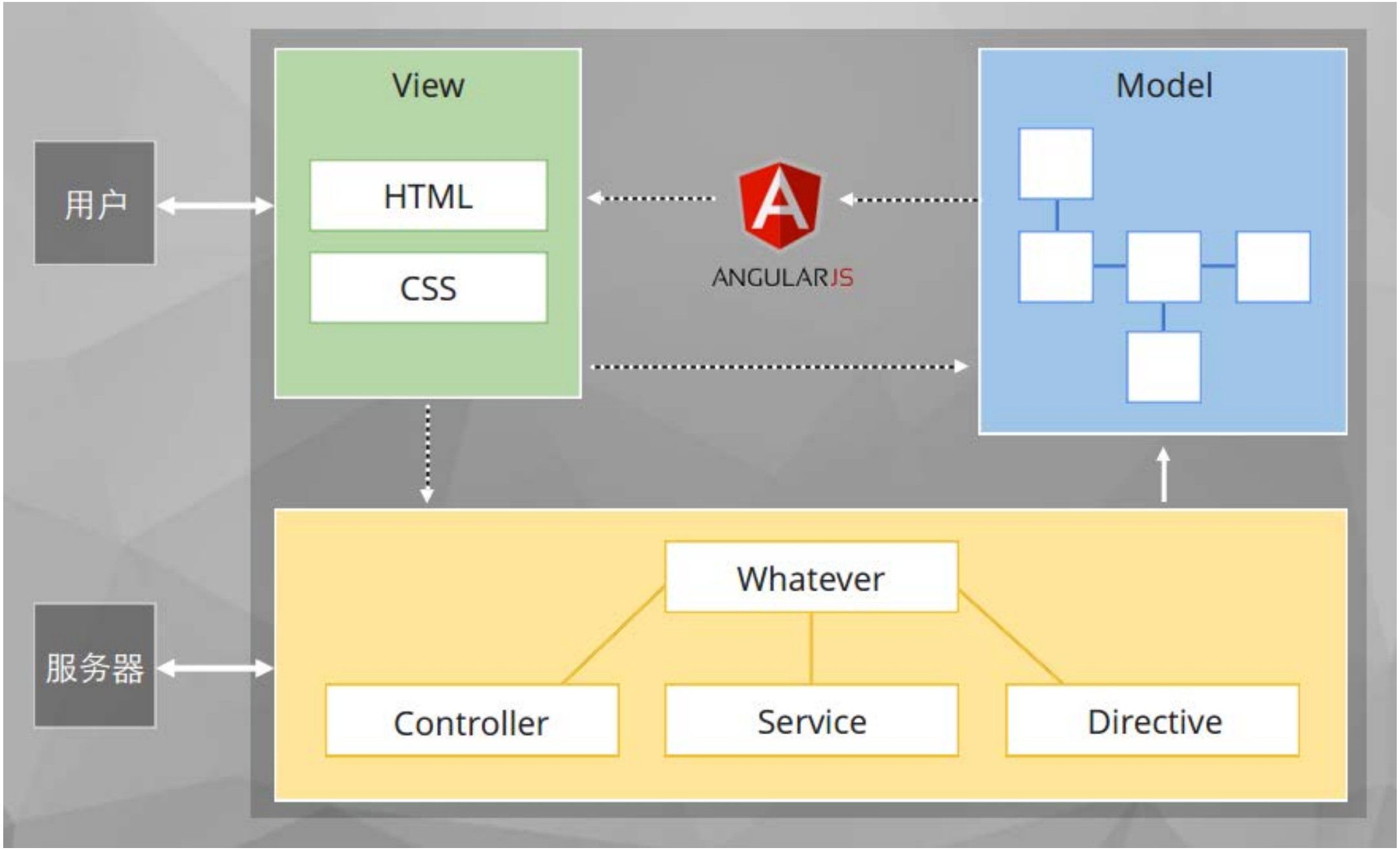
对比


Input
Controller
1
*
Model
View
MVC
Input
View
Model
Presenter
MVP
Input
View
*
1
Model
ViewModel
MVVM

AngularJS

- MV?

MVW


View
HTML
CSS
Model
用户
ANGULARJS
Whatever
服务器
Controller
Service
Directive

模块化

为什么要使用模块

- 可维护性
- 命名空间
- 可复用性

如何引入模块

- 匿名闭包函数
- 全局引入
- 对象接口
- 揭示模块模式
- CommonJS & AMD
- ES6

原始写法

- function m1(){
- //...
- }
- function m2(){
- //...
- }

- "污染"了全局变量
- 容易与其他模块发生变量名冲突
- 模块成员之间看不出直接关系

对象写法

```
var module1 = new Object({
    _count : 0,
    m1 : function (){
        //...
    },
    m2 : function (){
        //...
    }
});
```

立即执行函数写法

```
var module1 = (function(){
        var _count = 0;
        var m1 = function(){
            //...
        };
        var m2 = function(){
            //...
        };
        return {
            m1 : m1,
            m2 : m2
        };
    })();
```

输入全局变量

```
var module1 = (function ($, YAHOO) {
    //...
})(jQuery, YAHOO);
```

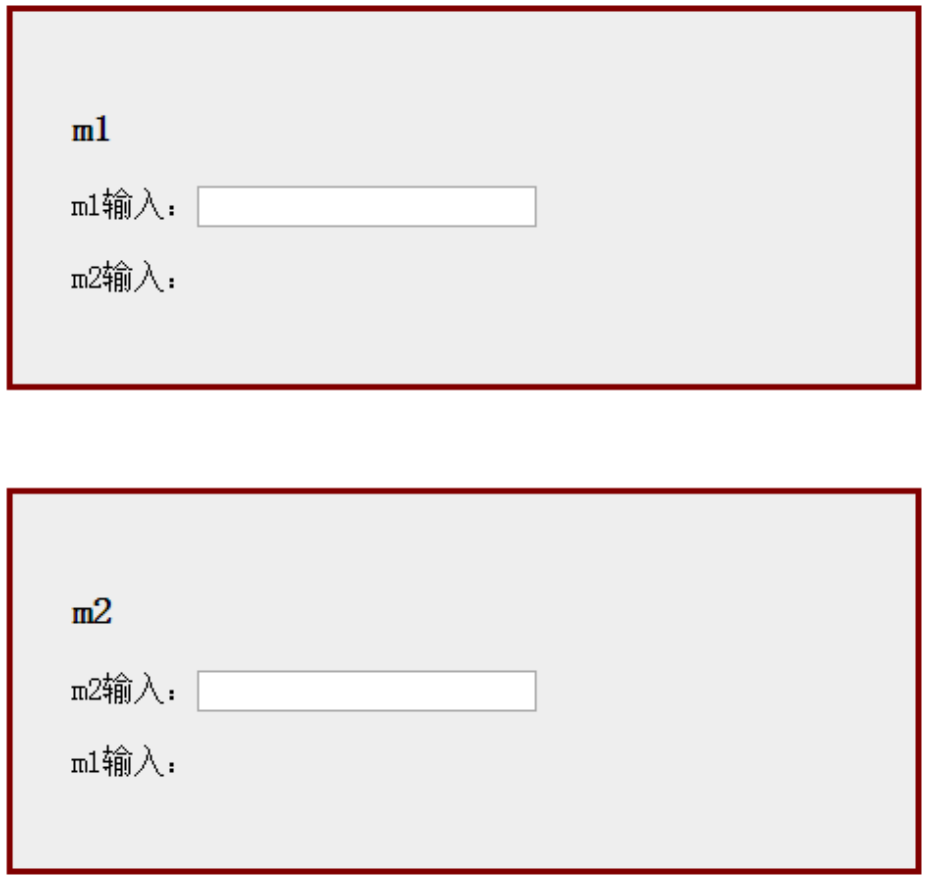
CommonJS

- /* math module */
- exports.add = function(a, b) {
- return a + b;
- };

- // 使用
- var math = require('math');

模块直接的耦合关系

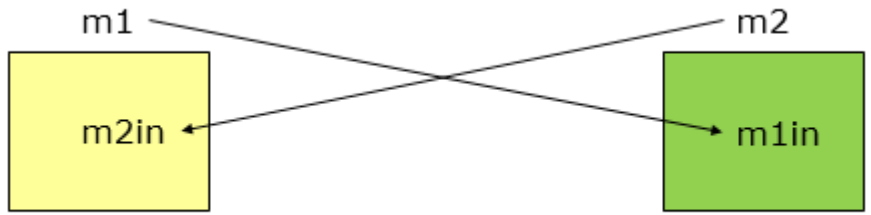
m1和m2是两个独立的模块，其中m2种会显示m1的输入，m1会显示m2的输入。



内容耦合

- // m1.js
- global.m2.m1input = this.value;
- m2.update();

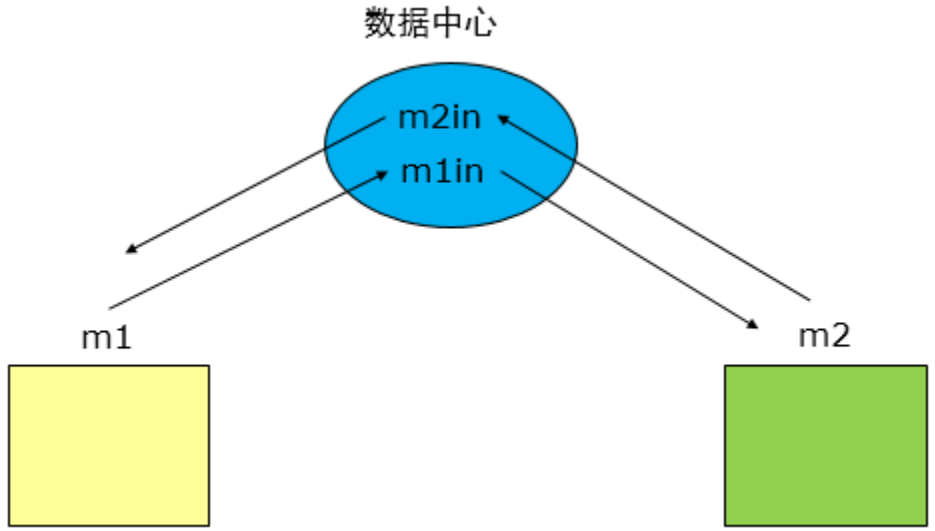
- // m2.js
- global.m1.m2input = this.value;
- m1.update();



公共耦合

- // m1.js
- global.data.m1input = this.value;
- m2.update();

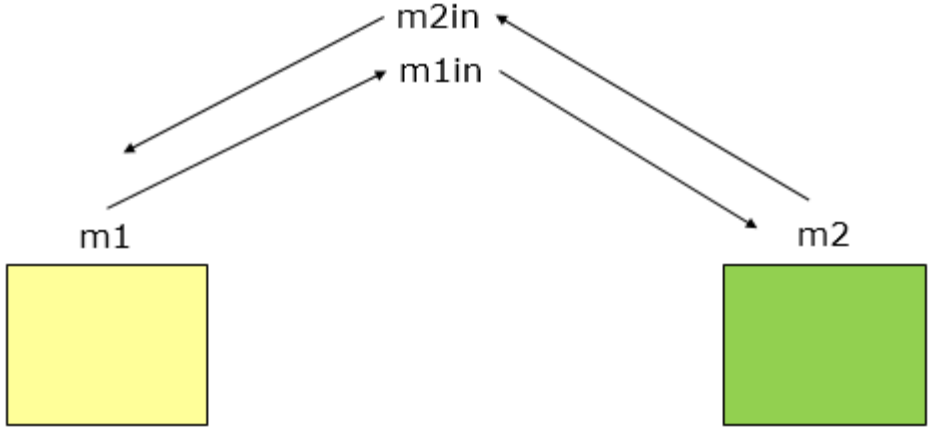
- // m2.js
- global.data.m2input = this.value;
- m1.update();



外部耦合

- // m1.js
- global.m1input = this.value;
- m2.update();

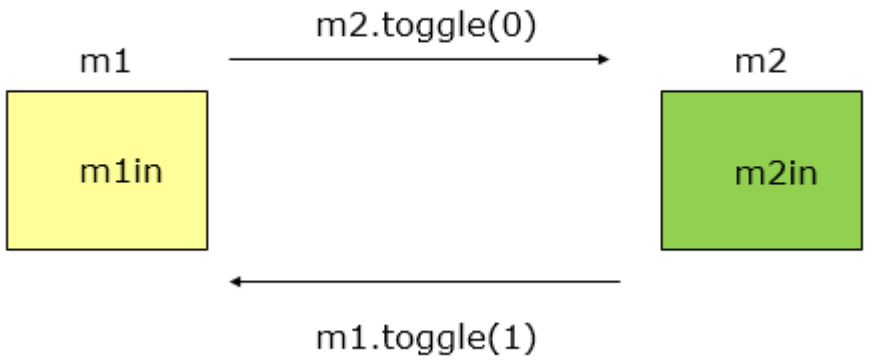
- // m2.js
- global.m2input = this.value;
- m1.update();



控制耦合

- // m1.js
- global.m1input = this.value;
- m2.update();

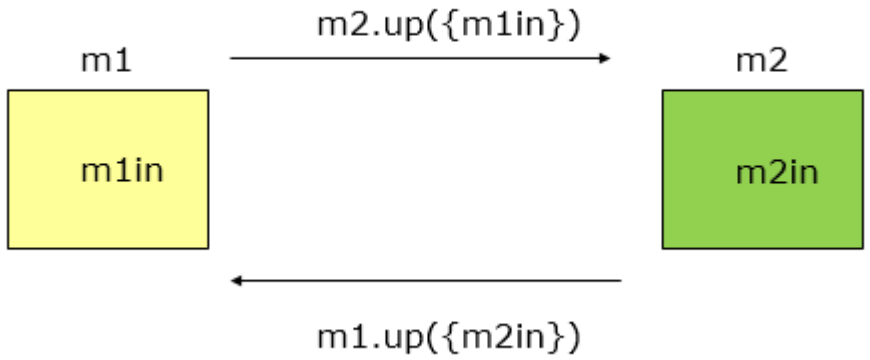
- m2.toggle(!!this.value); // 传递flag



标记耦合

- // m1.js
- me.m1input = this.value;
- m2.update(me); // 传递引用

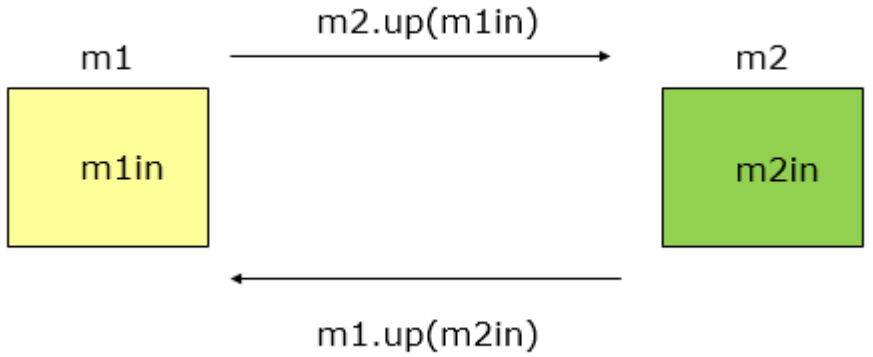
- // m2.js
- me.m2input = this.value;
- m1.update(me);



数据耦合

- // m1.js
- me.m1input = this.value;
- m2.update(me.m1input); // 传递值

- // m2.js
- me.m2input = this.value;
- m1.update(me.m2input);

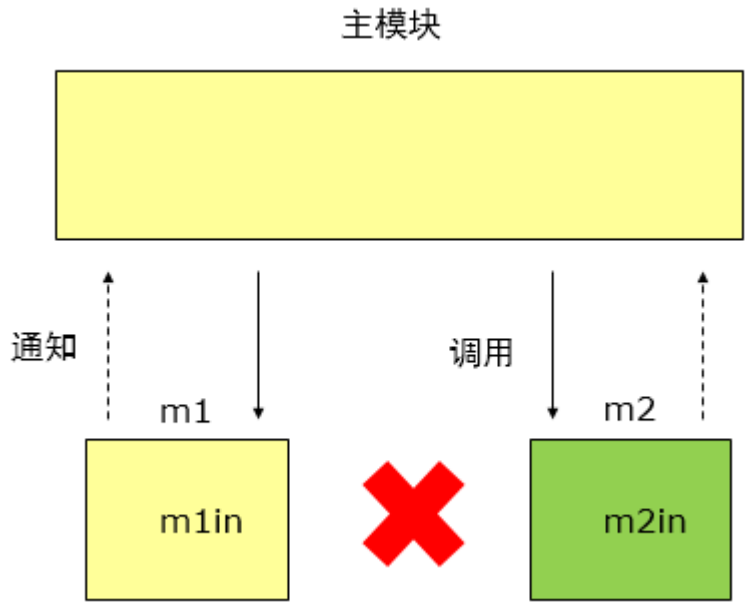


非直接耦合

- // index.js
- var m1 = roglobalot.m1;
- var m2 = global.m2;
- m1.init(function (str) {
- 	m2.update(str);
- });
- m2.init(function (str) {
- 	m1.update(str);
- });

- // m1.js
- me.m1input = this.value;
- inputcb(me.m1input); // inputcb是回调函数

- // m2.js
- me.m2input = this.value;
- inputcb(me.m2input);



前端与后端的职责划分

后端

- 提供数据
- 处理业务逻辑
- Server-side MVC架构
- 代码跑在服务器上

前端

- 接收数据，返回数据
- 处理渲染逻辑
- Client-side MV* 架构
- 代码跑在浏览器上

各层职责重叠，并且各玩各的

- Client-side Model 是 Server-side Model 的加工
- Client-side View 跟 Server-side是不同层次的东西
- Client-side的Controller 跟 Sever-side的Controller 各搞各的
- Client-side的Route 但是 Server-side 可能没有

性能问题

- 渲染，取值都在客户端进行，有性能的问题
- 需要等待资源到齐才能进行，会有短暂白屏与闪动
- 在移动设备低速网路的体验奇差无比

重用问题

- 模版无法重用，造成维护上的麻烦与不一致
- 逻辑无法重用，前端的校验后端仍须在做一次
- 路由无法重用，前端的路由在后端未必存在

跨终端问题

- 业务太靠前，导致不同端重复实现
- 逻辑太靠前，造成维护上的不易

SEO问题

- 渲染都在客户端，模版无法重用，SEO实现麻烦

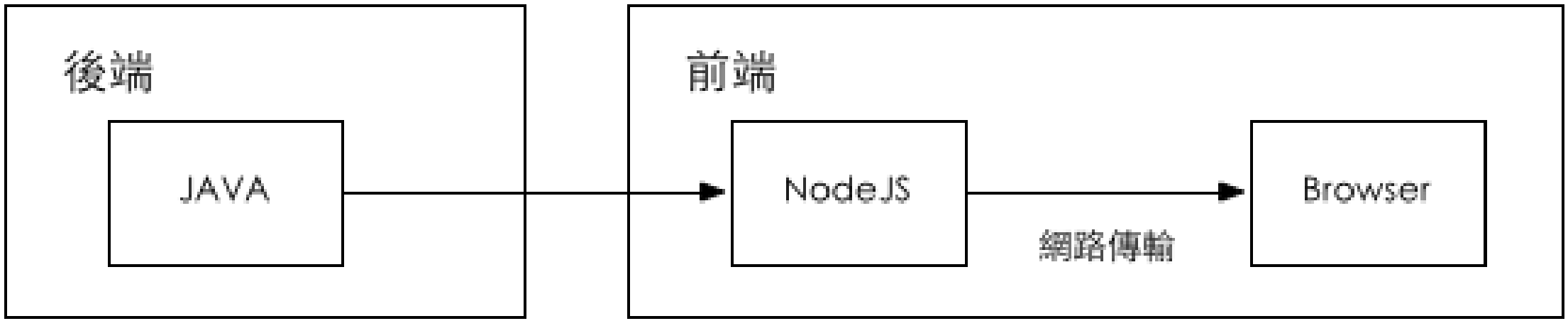
重新定义前后端


後端
JAVA
網路傳輸
前端
Browser

- 是依照**工作职责**来划分的前后端？
- 还是依照**软硬件环境**划分的前后端？

现在我们有了nodejs

重新定义的前后端



重新划分职责

NodeJS

- 跑在服务器上的JS
- 转发数据，串接服务
- 路由设计，控制逻辑
- 渲染页面，体验优化
- 更多的可能

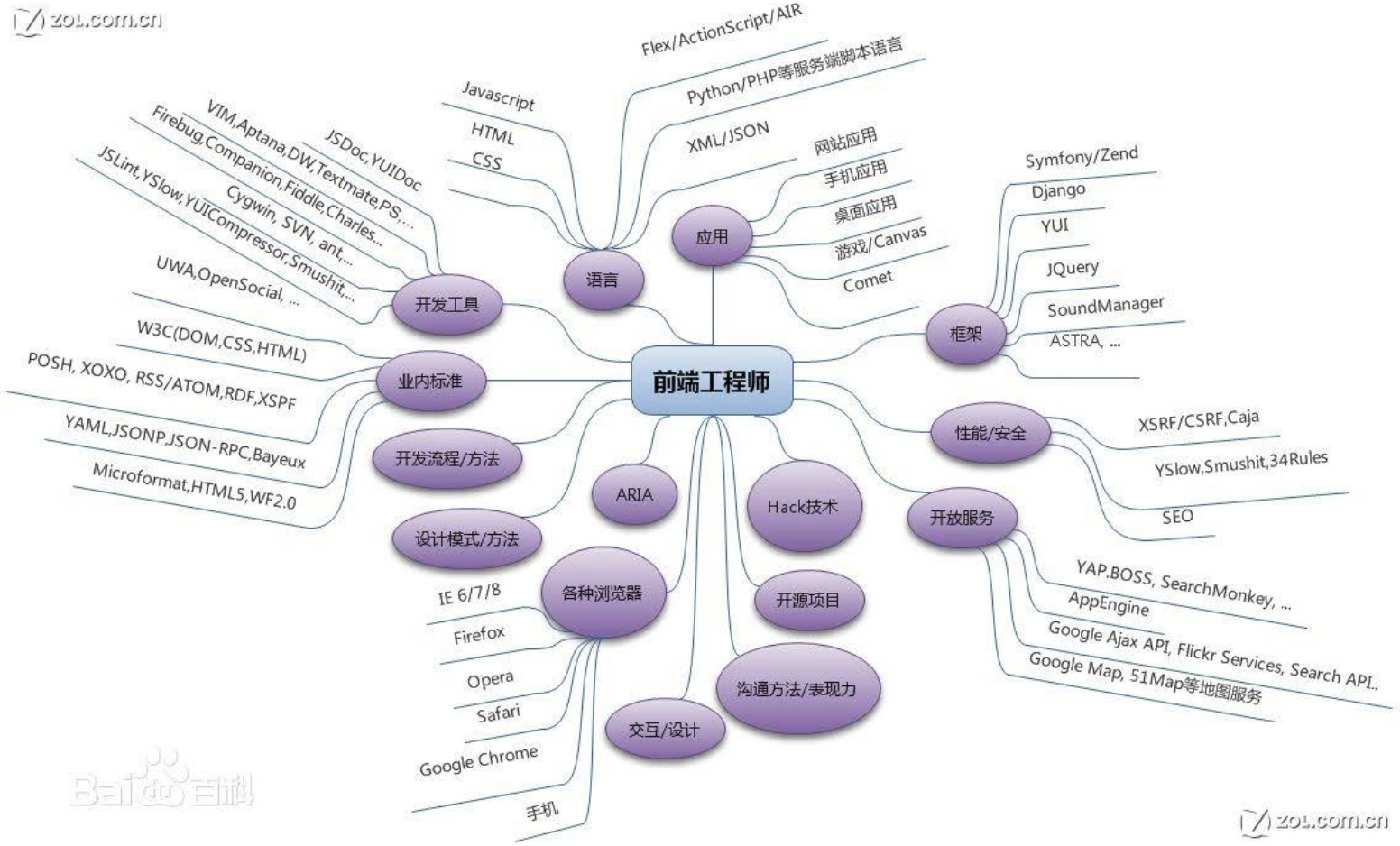
JS + HTML + CSS

- 跑在浏览器上的JS
- CSS、JS加载与运行
- DOM操作
- 任何的前端框架与工具
- 共用模版、路由

前后端分离

Why

前端变化远比后端变化频繁



数据逻辑与表现逻辑混杂不清

前端后端技能点差异很大

How？怎么做前后端分离

后端数据服务化

- SOA？
  - WebService(WSDL/SOAP)
  - ESB

- Micro Service
  - 快速迭代
  - 高可用
  - 高性能
  - 高并发

restful

- 把url映射为资源
- 把资源作为服务

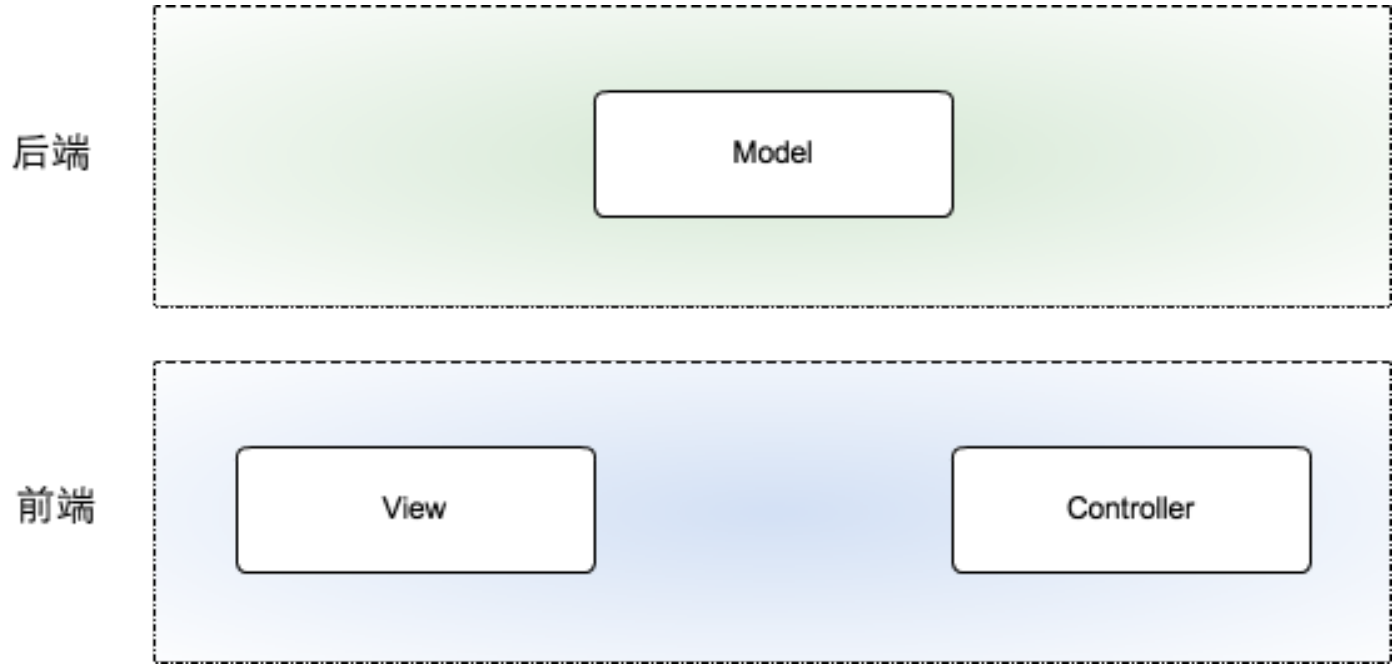
前端页面组件化

- 提高了开发效率
- 降低了维护成本
- 代码复用

前端组件的复用

- 控件
- 基础逻辑功能
- 公共样式
- 稳定的业务逻辑

- 前端：负责View和Controller层
- 后端：负责Model层，业务处理/数据等



小结

- 后端数据服务化
- 前端页面组件化

<questions />

by @jjc