

Vue Js

<https://vuejs.org/>

前码狗们， 经历了些什么

- 三方面关注的分离， 是一把双刃剑
- Data $\leftarrow\rightarrow$ Dom，是永恒主题
- 人肉Dom操作， 催生了让人爱恨交加的JQuery生态
- 不摆脱人肉Dom操作， 前端永远是暗无天日

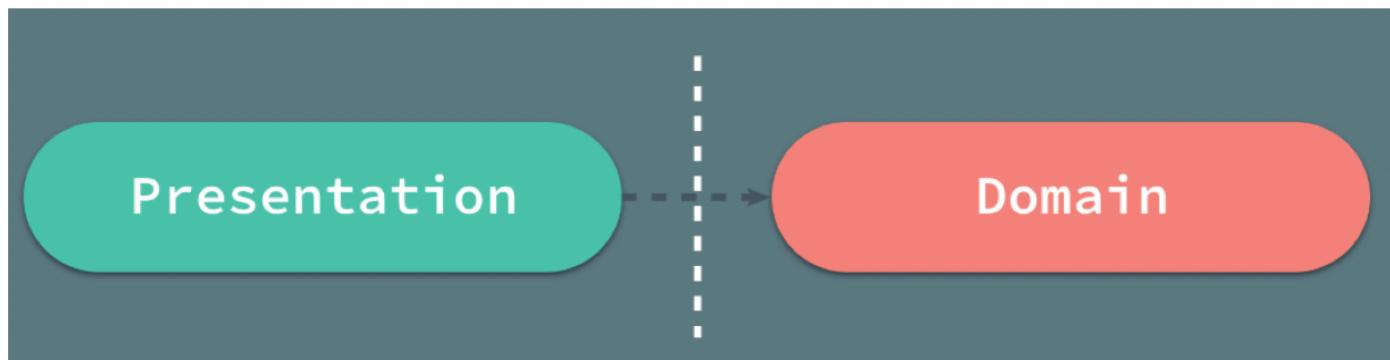
现代框架解决的问题：

- 不要操作Dom
- Data的变动， Dom自动改变
- Dom上的事件处理， 只改变Data，进而自动改变Dom
- 组件化Css,HTML,JS，并对组件进行组合，进而形成更大的组件
- 组件不会污染环境，并可以相互传递消息。
- 异步

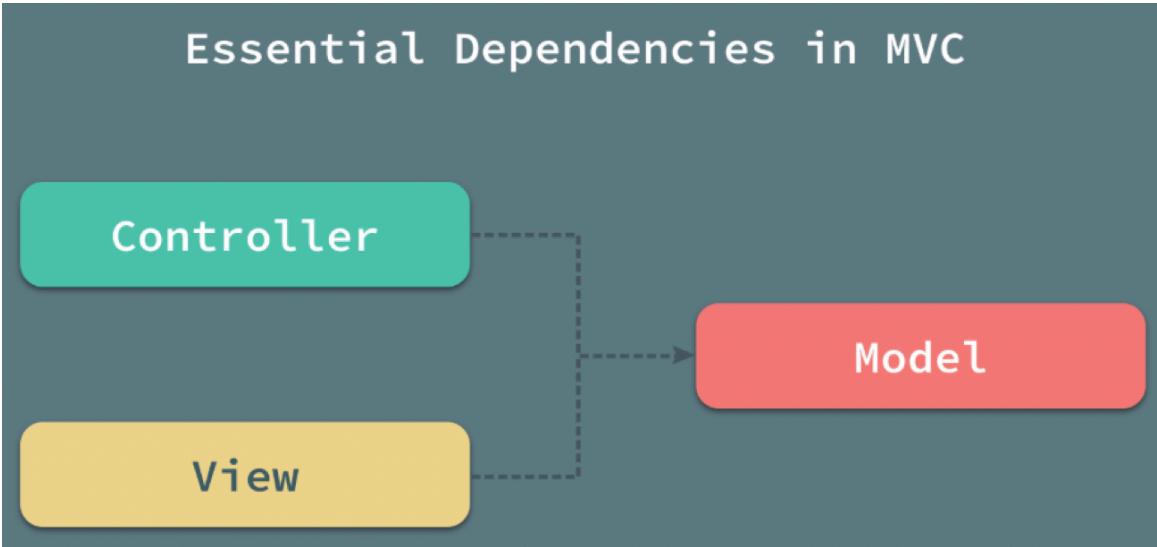
MVX架构

MVC到底是什么？

- 分离表现Martin Fowler

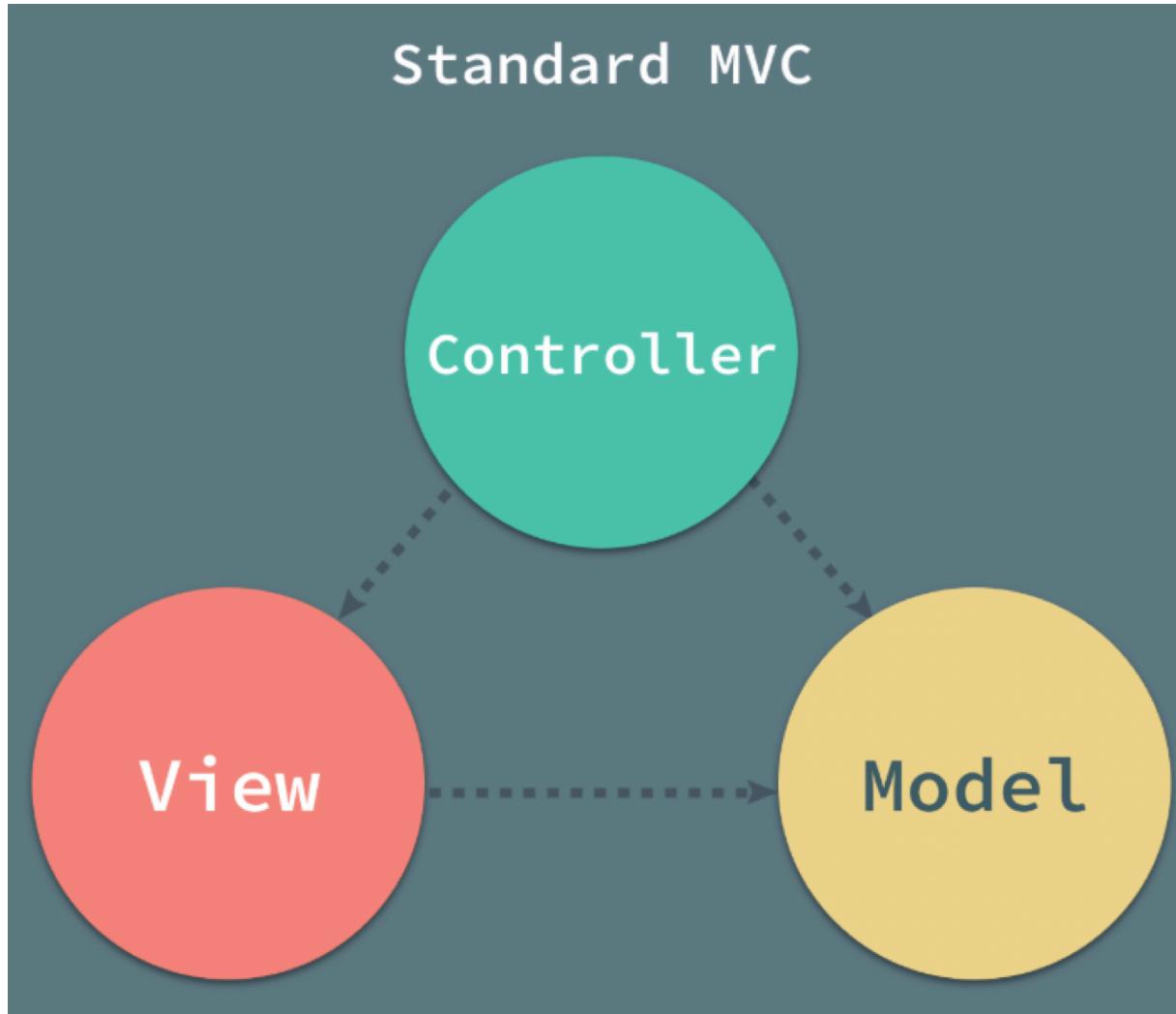


分工和分离是重点



MVC的提出，只是规定了每个部分负责的工作，并不涉及其交互的细节，所以导致了不同的对MVC的理解。

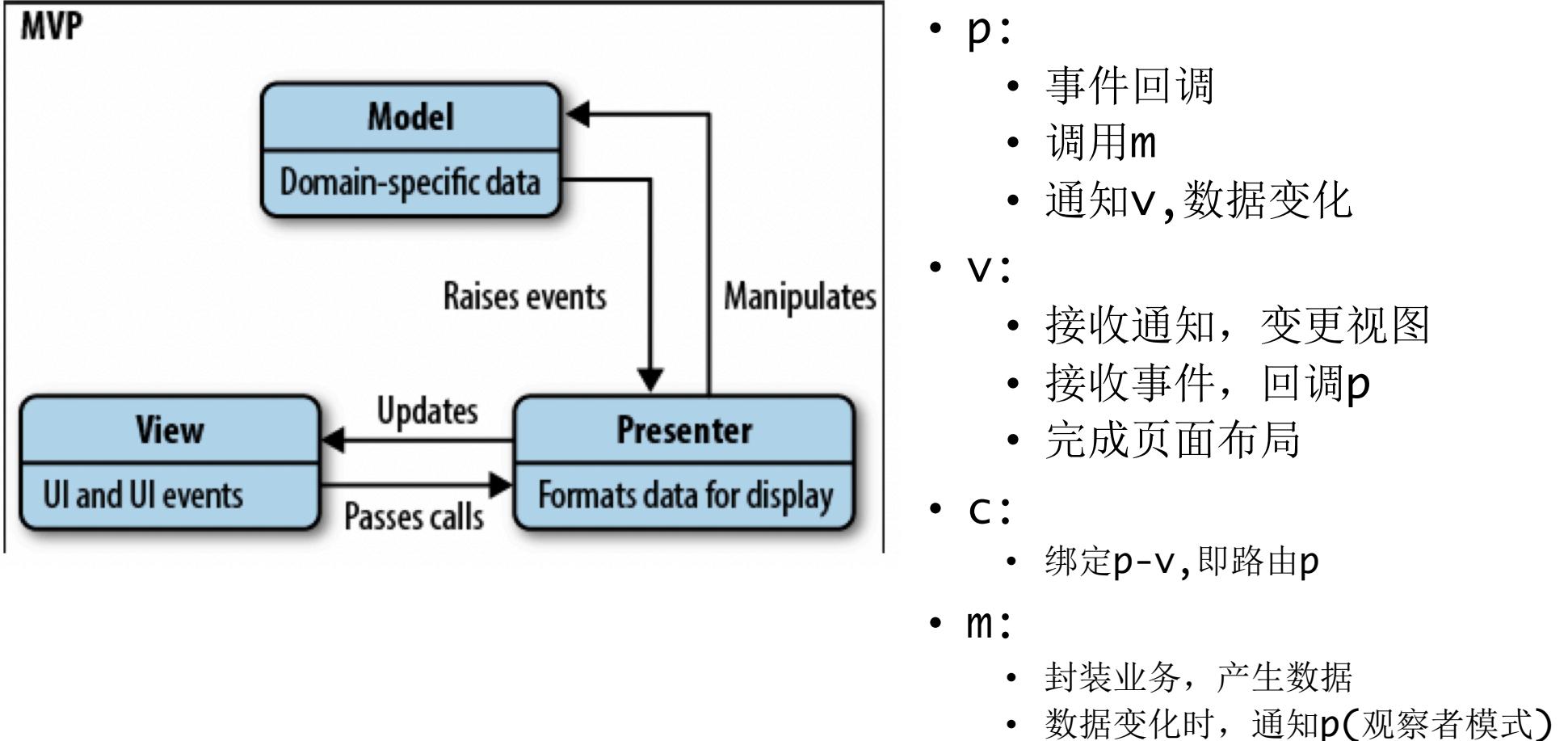
理论上是这样的



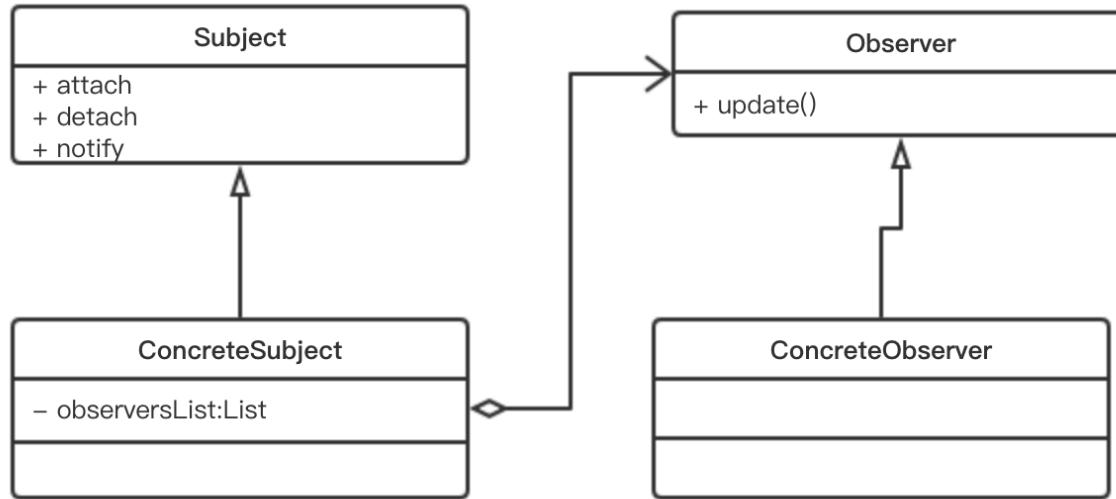
但这样，必然导致：

Massive Controller

要让M和V没有联系的MVP

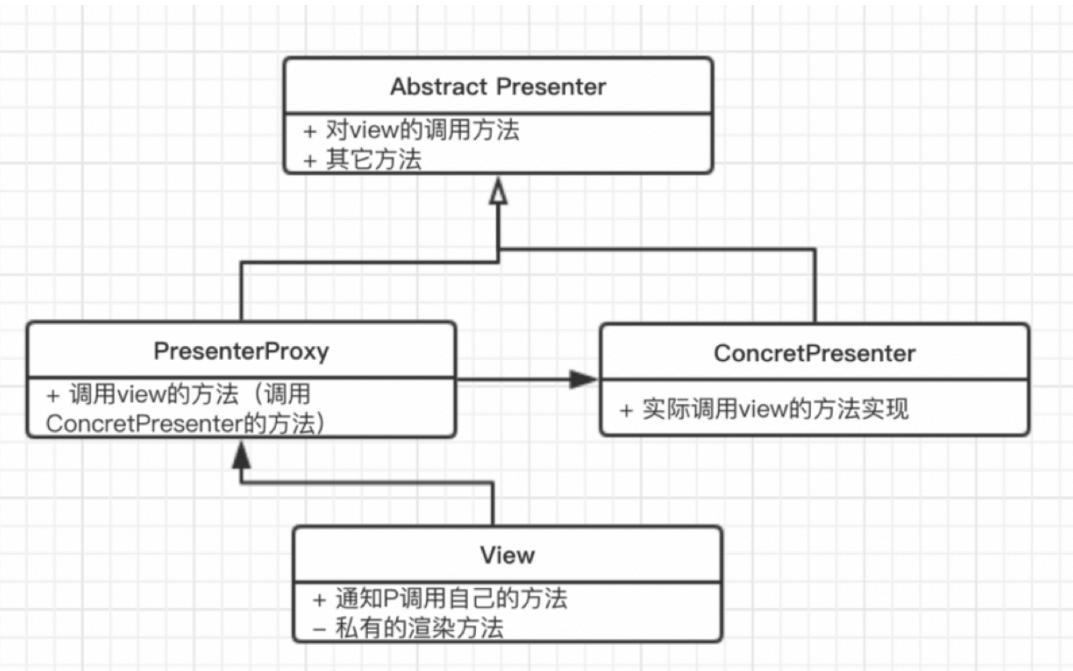


观察者模式



- 使View成为model的观察者
- 但这必须造成m-v的紧耦
- 所以，让p成为m的观察者关系，再去通知v
- 这就是“被动视图”

Passive View

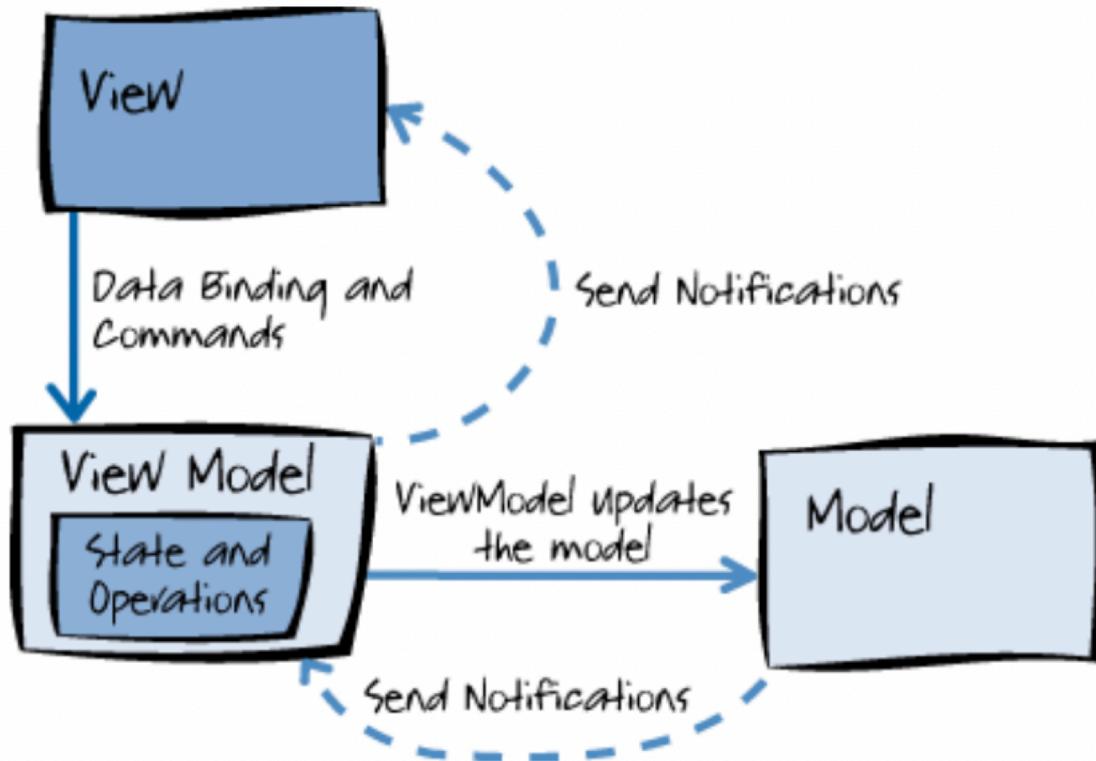


- 让 **v** 只依赖 **p** 的代理
- 这样更方便进行单元测试
- 但此方式仍然让 **view** 中有大量
的数据绑定代码
- 需要改良。 。 。

Supervising Controller模式

- model和view的双向关联
- 一方的变化，都会引起对方的改变
- 但此时仅限于简单的数据绑定，更复杂的渲染逻辑仍然在P中
- 它等待着专门的绑定语法出现。。

更强大的MV-VM



- 出现了模板语言和绑定语法
- 出现了模板和语法的解释器
- 它是一个看不见的**Binder**层
- 它背离了MVP的m-v分离
- 但由于**binder**自动完成，对于使用者，在逻辑层面上仍然是MVP



Vue的设计(虚实之间)

1. 组件化（**Vue类**），组件标签化，**Dom虚拟化**，**Progressive化**，**Reactive化**
2. 由一个**根组件对象**，建立**虚拟Dom**（根据el，或template）
3. 对**VDom**中的指令进行**内存渲染**，最后渲染成为**Dom**
4. 组件中的**Data**的**修改**，全部加入了**回调**，保证数据变动，自动更新到**VDom**中，并实时更新**Dom**
5. “2”步骤中的**template**包含**组件标签**，此时会**递归2, 3, 4**
6. 借助于**Babel**和**webpack**，使用**ES6语法**

魔鬼细节

实例化组件

```
//1.=====创造对象=====*/
```

```
new Vue({/* ... */});
```

```
//2.===== 创建构造器=====*/
```

```
let Profile=Vue.extend({/* ... */})  
// 创建 Profile 实例，并挂载到一个元素上。  
new Profile().$mount('#mount-point')
```

```
/*3.=====注册或获取全局组件=====*/
```

```
// 注册组件，传入一个扩展过的构造器
```

```
Vue.component('my-component', Vue.extend({ /* ... */}))
```

```
// 注册组件，传入一个选项对象（自动调用 Vue.extend）
```

```
Vue.component('my-component', { /* ... */})
```

```
// 获取注册的组件（始终返回构造器）
```

```
var MyComponent = Vue.component('my-component')
```

```
<!--局部注册-->  
<div id="app">  
  <hello-world></hello-world>  
</div>  
<script>  
  let HelloWorld={template:'<h1>Hello World</h1>'}  
  new Vue({  
    el:"#app",  
    components:{  
      "hello-world":HelloWorld  
    }  
  })
```

配置的分类

- data: data, methods, computed, watch, props
- dom: el, template, render
- hook: created, mounted, destroyed

指令

- v-text
- v-html
- v-show
- v-if
- v-else
- v-else-if
- v-for
- v-on
- v-bind
- v-model
- v-pre
- v-cloak
- v-once

实例属性、方法

- `vm.$data`
- `vm.$props`
- `vm.$el`
- `vm.$refs`

- `vm.$emit`
-

绑定技术

data->view

- 默认为单向绑定
 - 插值绑定 `{{data}}`
 - 属性绑定 `v-bind:[w3c属性/自定义属性]`
 - 事件绑定 `v-on:[w3c事件/自定义事件]`
- 对于表单也提供了双向绑定(`v-model`)
 - `text, textarea, number, email...`
 - `select`
 - `checkbox, radio`

计算属性

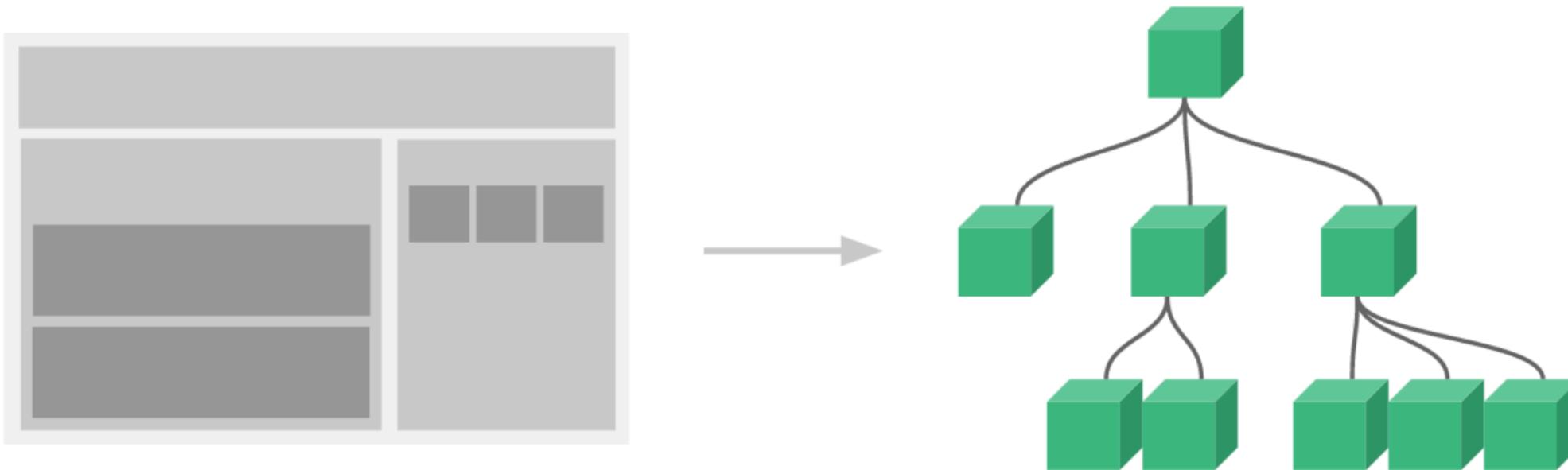
- 是**computed**下的函数，它只有在函数中出现的**data**变动时，才会更新**dom**(所谓的**lazy**更新)
- 适合于做批量操作，如统计操作

v系列

- **v-if/v-else-if/v-else** 用于条件判断元素是否出现
- **v-for** 用于遍历集合
- **v-html** 用于插入html数据（不要用于用户的输入-XXS）

父子组件及传参

组件化



组件化的几个关注点：

- 定义，实例化，注册，挂载
- 父向子组件的消息传递
- 子向父组件的消息传递
- 组件模块化

定义 实例化

注册 挂载

```
<div id="app">
    <!--*****实例化+挂载*****-->
    <sub-component></sub-component>
    <sub-component1></sub-component1>
    <sub-component2></sub-component2>
    <!--*****实例化+挂载*****-->

        <div id="app-inner"></div>
    </div>

<script>
    //定义组件 (字面量法)
    let SubComponent1={template:'<div>sub1</div>'}
    //定义组件 (Vue扩展法)
    let SubComponent2=Vue.extend({template:'<div>sub2</div>'})
        //手工实例化和挂载 (无注册)
        new SubComponent2().$mount("#app-inner")
    /*定义组件, 全局注册 (二合一法)
    Vue.component("sub-component"
        ,{template:'<div>sub</div>'})
    new Vue({
        el:"#app",
        //声明式注册 (局部注册)
        components:{
            'sub-component1':SubComponent1
            , 'sub-component2':SubComponent2
        }
    })
```

.vue组件

- Vue被设计成为组件化
- 组件采用嵌套结构组件
- 组件内容对外隔离
- 采用特殊方式来传参
- *此方法适用于webpack

```
<!--=====组件定义=====-->
<template>
  <!--结构定义-->
  <div id="app">
    <Root/><!-- 使用-->
  </div>
</template>

<script>
  /*=====行为定义=====*/
  import Root from './test1/Root' //导入
  export default { //App名字导出，为其它组件导入
    name: 'App',
    components: {Root} //注册
  }
</script>

<style>
  /*=====样式定义=====*/
</style>
```

父子传参（属性传参）

```
<!--父向子组件传参-->


<hello-world title="John"></hello-world>
    <hello-world title="Tom"></hello-world>



<script>
    let HelloWorld={template: `<h1>Hello {{title}}</h1>`  

        ,props:['title']} 子组件定义时的“形参”定义
    new Vue({  

        el:"#app",  

        components:{  

            "hello-world":HelloWorld  

        }  

    })
}
```

```
<!--复杂一点的父向子组件传参-->


<hello-world v-bind:customer="name: 'TomLi'"></hello-world>


<script>
    let HelloWorld= {
        template: `<h1>Hello {{customer.name}}</h1>`  

        ,props:{customer:{type:Object}}
    }
    ...

```

属性传参(父向子)

```
<template>
  <div>
    <Sub1 v-bind:name="name" v-bind:user="user"></Sub1>
  </div>
</template>
<script>
  import Sub1 from "./Sub1"
  export default {
    name: "root",
    components:{Sub1},
    data(){
      return {
        name:"John Yu",
        user:{name:"Mike"}
      }
    }
  }
</script>
```

```
<template>
  <div>
    {{name}},{{user.name}}
  </div>
</template>
<script>
  export default {
    name: "Sub1",
    props:{
      name:{type:String},
      user:{type:Object}
    }
  }
</script>
```

事件传参(子向父)

```
<template>
  <div>
    <button @click="sendToRoot">sendToRoot</button>
  </div>
</template>

<script>
  export default {
    name: "sub2",
    methods:{
      sendToRoot(){
        //sub2CallYou做为自定义事件名称，出现在Root中
        this.$emit("sub2CallYou", {name:"Sub2 name"})
      }
    }
  }
</script>
```

```
<template>
  <div>
    <!--rootProcess做为自定义事件的回调-->
    <Sub2 v-on:sub2CallYou="rootProcess"></Sub2>
    {{user.name}}
  </div>
</template>
<script>
  import Sub2 from "./Sub2"
  export default {
    name: "root",
    components:{Sub2},
    data(){
      return {
        user:{name:"Mike"}
      }
    },
    methods:{
      rootProcess(user){
        this.user.name=user.name;
      }
    }
  }
</script>
```

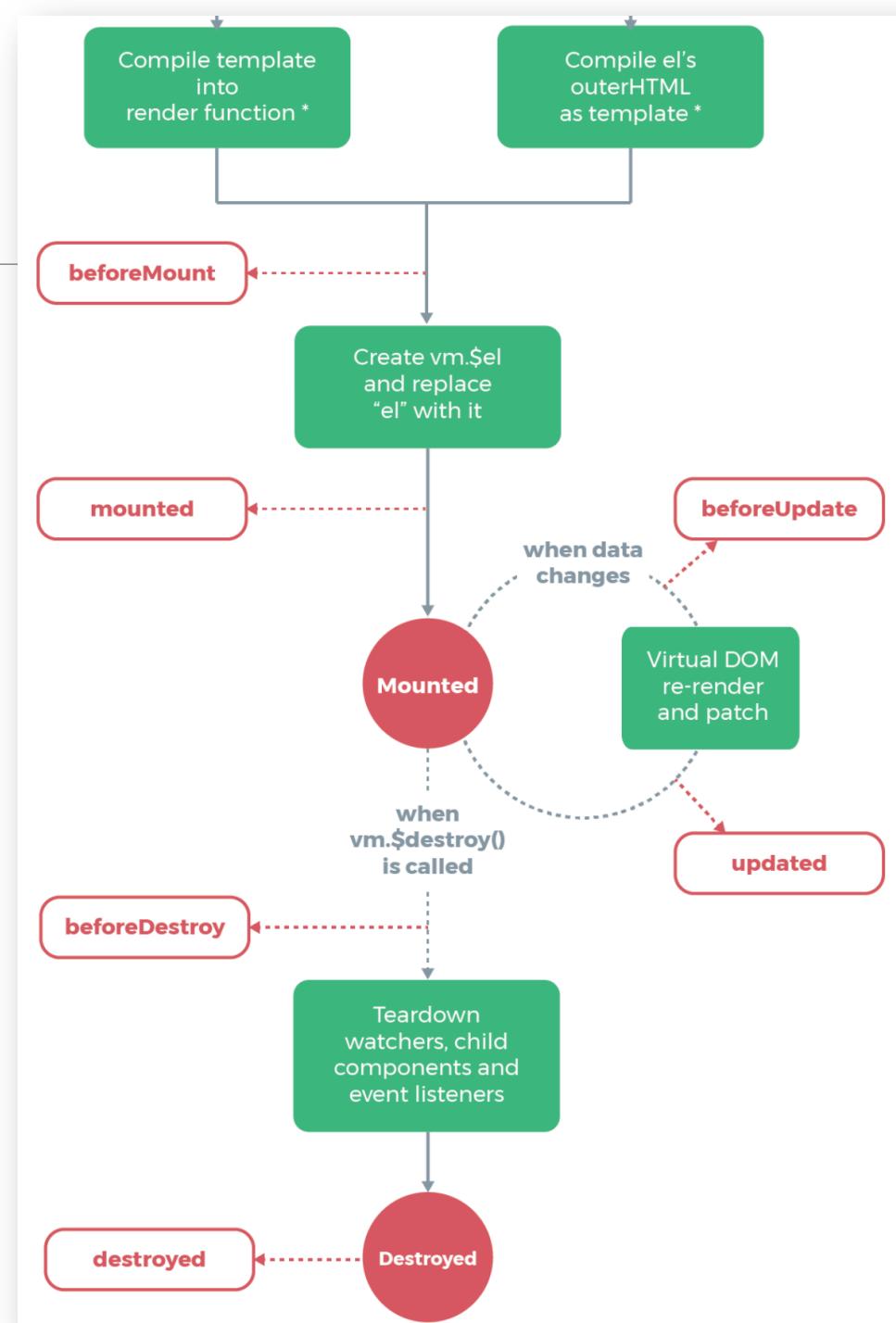
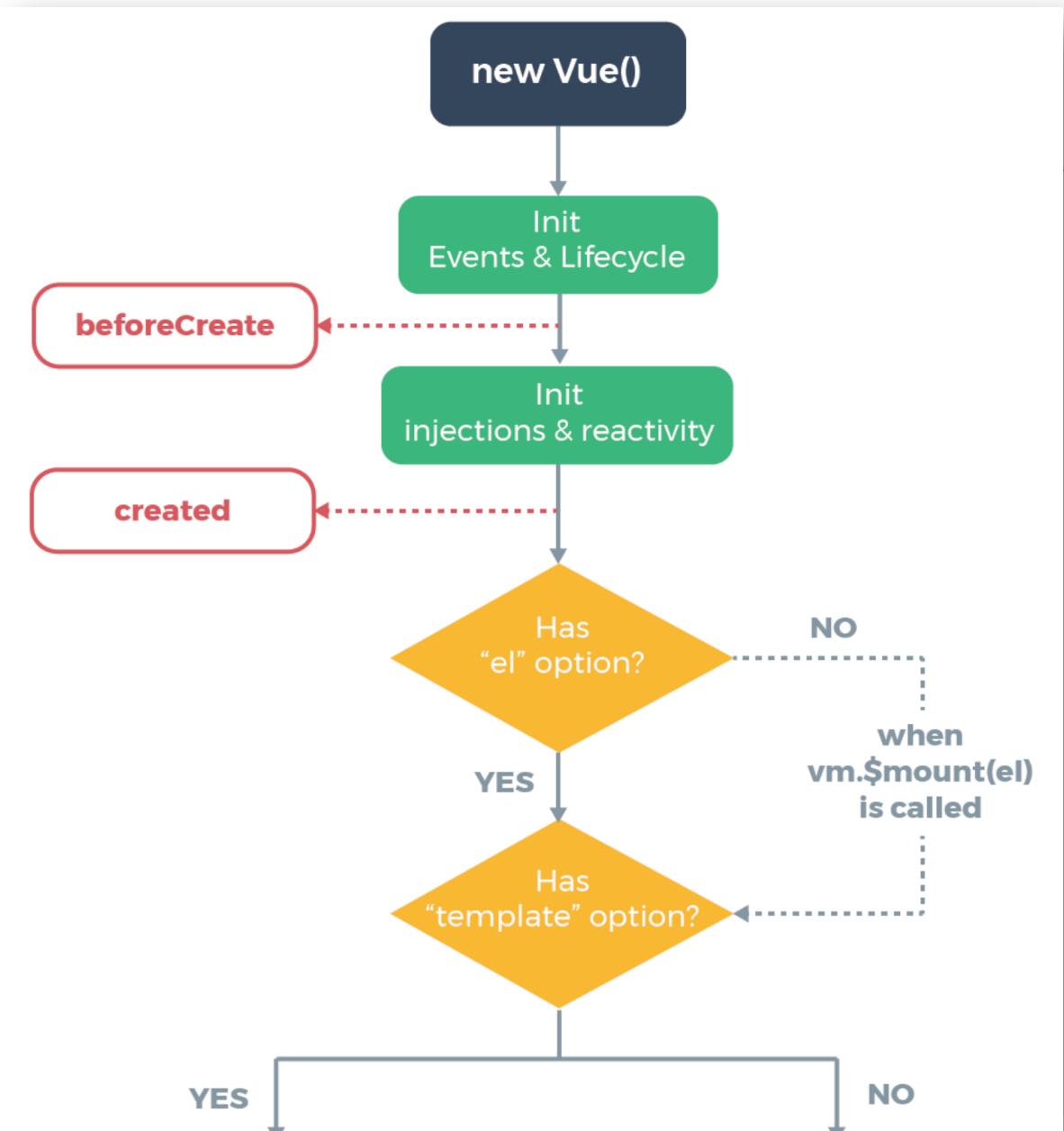
```
8 <body>
9   <div id="app">
10     <h2>{{book.bname}}</h2>
11     <book-adder @addbook="onAddBook($event)"></book-adder>
12   </div>
13 <script>
14   let BookAdder= {
15     template: `<div>Book Adder
16       <button @click="$emit('addbook',{bname:'Core Java'})">add book</button>
17     </div>
18     ,props:{customer:{type:Object}}
19   }
20   new Vue({
21     指定“事件代理”的回
22     el:"#app", 调处理
23     data:{book:{}},
24     methods:{
25       onAddBook:function (book) {
26         this.book=book
27       }
28     },
29     components:{
30       "book-adder":BookAdder
31     }
32   })
33 </script>
34 </body>
```

子组件事件的
“事件代理”

实参传递给“实参代理”，再传递
给形参

指定“事件代理”的回
调处理

Vue 对象的生成周期



Vue-Cli

使用原因

- 需要使用模块系统
- 需要打包工具
- 需要完成项目的代码检查，单元测试，集成测试，打包等一系列的构建过程
- 手工建立脚手架，需要一定的专业训练

建立过程

- `npm install -dev-save vue-cli` --安装工具
- `npx vue init webpack my-project` --建项目
- `npm run dev` --运行
- `npm build` --打包

服务器的异步交互

测试：https://gitee.com/bj_java_161221/codes/udc57ypqjhkegrso2wn6z20

设计思想

- 自身不提供与服务器交互的组件
- 需要借助于“**原生或其它库**”完成异步通讯
- 基本方式：在请求的回调中，更新**data**，进而自动更新**vdom**
- 可选方案：**xhr, fetch(), jquery, lodash, axios**

样例

//一次添加请求及回调处理

```
let myHeaders = new Headers();
myHeaders.append('Content-Type', 'application/json');
fetch(this.url,
  {method:"POST",
   headers: myHeaders,
   body:JSON.stringify(p)})
.then(response=>response.json())
.then(newProduct=>this.products.push(newProduct));
```

Vue-Router

<https://router.vuejs.org/zh/>

更多的是使用了....

我们完成一个案例

Element

<http://element-cn.eleme.io/#/zh-CN>

Element

- `npm i element-ui -S`
- `import ElementUI from 'element-ui'`
- `import 'element-ui/lib/theme-chalk/index.css'`
- `Vue.use(ElementUI)`