课程简介

简介

随着前端的发展及现代浏览器的普及，只为解决兼容性问题并且有些臃肿的jQuery逐渐没落，Angular，Vue和React这些新生的前端框架给我们带来的完全不一样的体验，让前端程序员只关注自己的业务及数据的变化，不再需要操作繁琐的DOM，前端也渐渐工程化，大大的提高的开发及维护的效率。

本系列课程带领大家逐步完成完整的前端MVVM框架，今天主要实现基本的核心逻辑

目标

```
<div id="app">
    <p>{{price}}元</p>
    <p v-text="msg"></p>
    <button @click="addHandle">add</button>
</div>
```

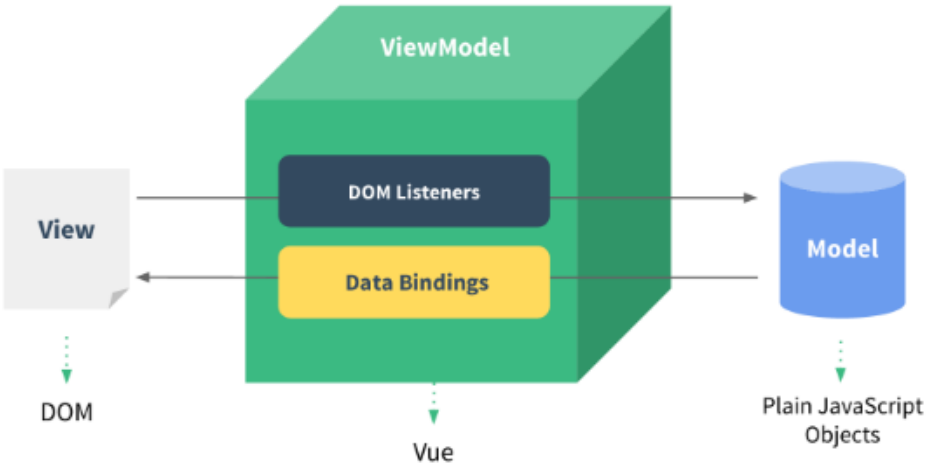
```
var vm = new Vue({
    el: '#app',
    data: {
        price: 100,
        msg: 'hello world'
    },
    methods: {
        addHandle: function() {
            this.price++
        }
    }
})
```


什么是 MVVM

先来说说我对 MVVM 的理解

介绍

MVVM 分别对应的是 Model View ViewModel ，对Vue了解的，应该知道Model和View是什么，关键是ViewModel；我们之前课程中也是按照以下这张图讲解的；View即视图，Model即模型，是数据，ViewModel就是关联起模型和数据的部分，在前端框架中，主要担任的数据绑定和监听页面事件的角色，在Vue中，el就是模板view，data就是数据model



前端框架带来哪些改变

jquery也可以实现我们现在页面的全部功能，但是需要手动绑定事件到DOM上，页面变化的时候我们也需要操作DOM，什么事都得操作DOM，这样我们容易把MVC中的三个都耦合起来，容易搞成意大利面一样的程序。

MVVM模式可以让我们程序员更加关注我们业务的实现，代码只关注数据的变化，View让它自动更新即可，相比手动操作DOM，我们只需要修改Model的数据就行，降低耦合的同时，更加符合人的逻辑习惯。

所以我们现在不仅要学会使用这些框架，在我们想提高代码能力，增加面试成功率的时候，就需要研究源码，弄清框架的核心逻辑和本质，那我们这样的课程，就是带领大家，完成一个前端MVVM框架的，今天，我们只实现最基础的核心逻辑，并且我们参照Vue1.0版本的源码来实现，因为2.0以后，增加了虚拟DOM，和Diff算法，在之后的系列课程，我们都会去实现它！

核心要素/模块

响应式

MVVM的核心就是View与Model分离后，修改Model中的数据，View也随之自动更新，而不是手动的去修改DOM；用户在View中进行了操作，Model中的数据也随之更新，其实就是我们说的数据双向绑定；

而想要完成响应式，则需要一套完善的响应机制，就是观察者模式，也叫发布-订阅模式。我们在初始化页面的时候，监听页面的数据以及数据所对应的View元素，一旦数据发生变化，立即更新View。

数据劫持

```
<div id="app"></div>
<input type="text" id="input">
```

```
// 定义一个空的obj对象
var obj = {};
// 定义一个空数据
var value = "";
// 我们使用Object.defineProperty进行数据劫持，
// 给obj添加一个名叫msg的属性
// 改函数接收三个参数
```

```
// 1.第一个参数，是一个对象
// 2.第二个参数，是给该对象上设置的属性名
// 3.第三个参数，是一个配置对象，可以配置该属性的set/get方法
Object.defineProperty(obj, "msg", {
    set: function(newValue) {
        console.log("执行了set函数");
        // 当数据发生变动时，我们重新渲染对应的html
        document.querySelector("#app").innerHTML = newValue;
        value = newValue;
    },
    get: function() {
        console.log("执行了get函数");
        return value;
    }
});
// 监听input事件，当发生变化时
// 重新设置obj的msg属性，触发set函数，则view自动更新
document.querySelector("#input").addEventListener("input", function(e) {
    obj.msg = e.target.value;
});
```

观察者模式/发布订阅模式

不论在程序的世界里，还是现实世界里，发布订阅模式，都被广泛应用，我们可以先看一个现实中的例子。

小明最近看上了一套房子，到了售楼处之后才被告知，该楼盘的房子已经售罄。好在销售员告诉小明，不久后还有一些尾盘推出，开发商正在办理相关手续，手续办好后就可以购买，但到底什么时候，没人知道。

于是小明记住了售楼处的电话，以后每天都会打电话过去询问是不是已经可以购买了，但是，除了小明，还有其他很多人小红，小强，小龙等也会每天打电话咨询同样的问题，一个星期后，售楼MM辞职了，原因是厌烦了每天回答成千上万个相同的电话问题。

当然，现实中不会有这么笨的公司和人，真实的故事应该是这样的，小明离开之前，把自己电话号码留在了售楼处，销售员答应他，尾盘一推出，就马上发短信通知小明，而小红，小强，小龙也同样把电话留在了售楼处的客户名册上，现在直到尾盘退出的时候，销售员就会翻开客户名册，遍历上面的电话号码，依次发送一条短信通知。

这，就是观察者模式，或者叫，发布订阅模式；那我们可以看实际的代码示例来进一步的了解。

```
<button onclick="sell()">尾盘开售了</button>
```

```
// 定义一个事件处理中心
// 该对象中包含subs的数组，存放需要遍历执行的函数
// 相当于客户名册，只不过需要提供一个添加和通知的功能
function EventHandle() {
    var subs = [];
    // 添加需要执行的函数
    this.addSub = function(sub) {
        sub && subs.push(sub);
    };
    // 遍历通知
    this.notify = function() {
        subs.forEach(function(sub) {
            sub.update();
```

```
        });
    };
}
// 实例化
var ev = new EventHandle();

// 尾盘开售了
// 发布者
function sell() {
    console.log("尾盘开售了");
    ev.notify();
}

// 订阅者客户小明
(function Ming() {
    var myName = "小明";
    ev.addSub({
        update: function() {
            console.log(myName + "接收到了通知");
        }
    });
})();

// 订阅者客户小红
(function Hong() {
    var myName = "小红";
    ev.addSub({
        update: function() {
            console.log(myName + "接收到了通知");
        }
    });
})();
```


模板解析

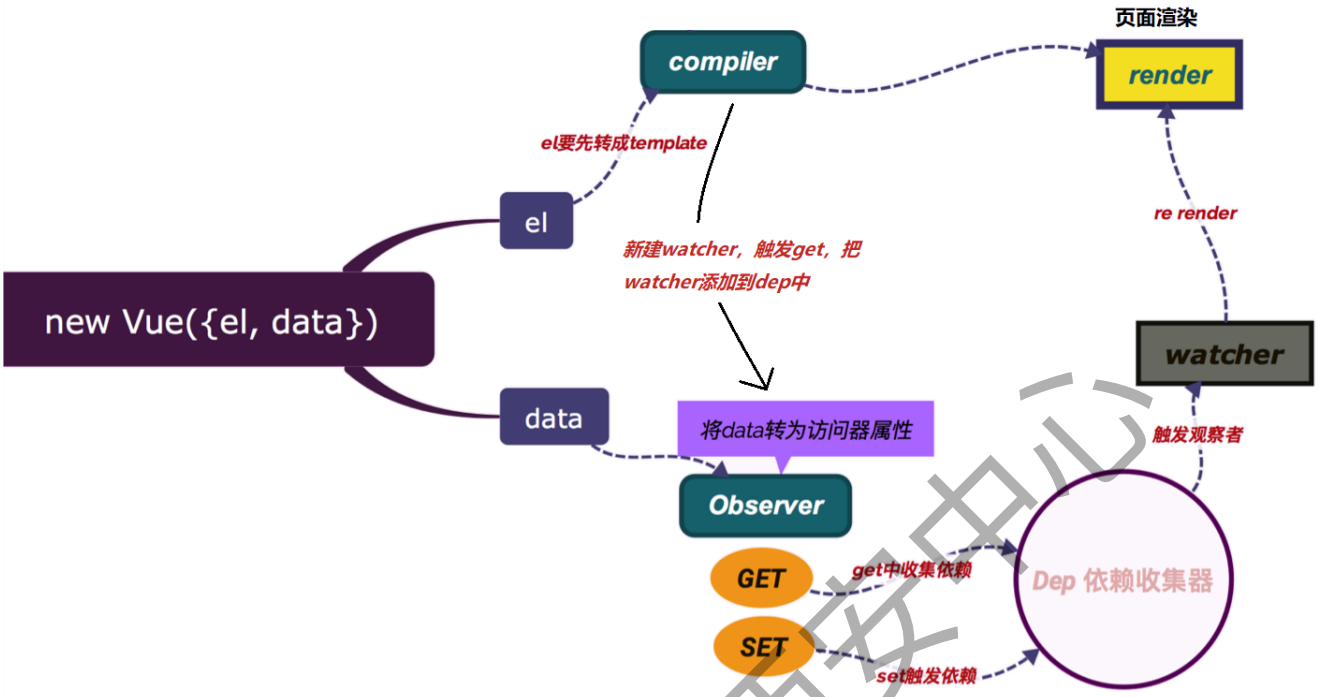
首先模板就是一段HTML代码片段，但是还有很多的指令以及{{}}，这些东西浏览器本身是不识别不认识的，所以我们需要把模板解析成浏览器认识的真正的html代码。

Vue1.x的模板引擎并没有对虚拟DOM的支持，而Vue2.0已经增加；我们这里主要参照Vue1.x来完成模板的解析。

虚拟DOM

在本系列的下个课程中，我们会完成从模板到AST，再到虚拟DOM，渲染页面的整个流程，今天我们先不添加虚拟DOM的功能，故今天也不涉及Diff算法的讲解。

完整流程



核心代码实现

创建项目

- 新建目录src，dist，example
- example目录中新建demo.html，new Vue，打开页面展示报错
- src中新建index.js，webpack打包，html中引入文件
- 展示webpack --watch打包过后的控制台输出

```
class Vue {
  constructor(options) {
    console.log(options);
  }
}
window.Vue = Vue
```

框架壳子-vue.js

```
import Observer from './observer'
import Complier from './compiler'

class Vue {
  constructor(options) {
    // 获取元素DOM对象
    this.$el = document.querySelector(options.el)
    // 转存数据
```

```
    this.$data = options.data || {};

    // 数据劫持
    new Observer(this.$data);
    // 编译模板
    new Complier(this);
  }
}
window.Vue = Vue
```

Observer

```
export default class Observer {
  constructor(data) {
    // 数据转存
    this.data = data;
    // 遍历对象完成数据劫持
    this.walk(data);
  }

  /**
   * 遍历对象
   * @param {*} data data总数据
   */
  walk(data) {
    if (!data || typeof data !== "object") {
      return;
    }
    Object.keys(data).forEach(key => {
      // 数据劫持函数
      this.defineReactive(data, key, data[key]);
    });
  }

  /**
   * 动态响应式数据
   * @param {*} data 对象
   * @param {*} key 键
   * @param {*} value 值
   */
  defineReactive(data, key, value) {
    Object.defineProperty(data, key, {
      // 可遍历
      enumerable: true,
      // 不可再配置
      configurable: false,
      get: () => {
        console.log('触发了get')
        return value;
      },
      set: newValue => {
        if (value === newValue) {
          return;
```

```
        }
        console.log('触发了set')
        // TODO 触发View的变化
        value = newValue;
      }
    });
    // 继续往下层编译添加响应式
    this.walk(value)
  }
}
```

Data&Method代理

```
添加到index.js中
这两行要添加在数据劫持之前
this._proxyData(this.$data);
this._proxyMethods(options.methods);

_proxyData(data) {
  Object.keys(data).forEach(key => {
    Object.defineProperty(this, key, {
      set(newValue) {
        data[key] = newValue;
      },
      get() {
        return data[key];
      }
    });
  });
}

_proxyMethods(methods) {
  if (methods && typeof methods === "object") {
    Object.keys(methods).forEach(key => {
      this[key] = methods[key];
    });
  }
}
```

Complier

```
export default class Compiler {
  constructor(context) {
    this.$el = context.$el;
    this.context = context;
    // 判断是否存在$el
    if (this.$el) {
      // 原始DOM对象转换为documentfragment
      this.$fragment = this.nodeToFragment(this.$el);
      // 编译模板
      this.compiler(this.$fragment);
      // 添加文档碎片到页面中
```

```
      this.$el.appendChild(this.$fragment);
    }
  }

  /**
   * 把id为app的div下的所有元素转换为文档片段
   * @param {*} node DOM节点
   */
  nodeToFragment(node) {

  }

  /**
   * 编译模板
   * @param {*} node
   */
  compiler(node) {

  }
}
```

查看MDN文档：document.createDocumentFragment()

`DocumentFragments` 是DOM节点。它们不是主DOM树的一部分。通常的用例是创建文档片段，将元素附加到文档片段，然后将文档片段附加到DOM树。在DOM树中，文档片段被其所有的子元素所代替。

因为文档片段存在于**内存中**，并不在DOM树中，所以将子元素插入到文档片段时不会引起页面回流（对元素位置和几何上的计算）。因此，使用文档片段通常会带来更好的性能。

nodeToFragment函数

```
/**
   * 把id为app的div下的所有元素转换为文档片段
   * @param {*} node DOM节点
   */
  nodeToFragment(node) {
    // 创建文档片段
    var fragment = document.createDocumentFragment();
    if (node.childNodes && node.childNodes.length) {
      // 循环Node
      node.childNodes.forEach(child => {
        // 判断是否为换行和注释节点
        if (!this.isIgnorable(child)) {
          fragment.appendChild(child);
        }
      });
    }
    return fragment;
  }

  /**
   * 判断是否为注释或元素节点
   * @param {*} node 单个元素
```

```
 */
isIgnorable(node) {
  // 匹配所有tab回车换行
  var regIgnorable = /^[\t\n\r]+/;
  // console.log打印查看nodeType的值
  // 查看MDN文档
  // nodeType==8为注释节点
  // nodeType==3为text节点
  return (
    node.nodeType == 8 ||
    (node.nodeType == 3 && regIgnorable.test(node.textContent))
  );
}
```

compiler函数

```
/**
  * 编译模板
  * @param {*} node
  */
 compiler(node) {
   if (node.childNodes && node.childNodes.length) {
     // 循环所有的节点
     node.childNodes.forEach(child => {
       if (child.nodeType === 1) {
         // 当nodeType为1时，说明是元素节点
         this.compileElementNode(child);
       } else if (child.nodeType === 3) {
         // 当nodeType为3时，说明是文本节点
         this.compileTextNode(child);
       }
     });
   }
 }

 /**
  * 编译element节点
  * @param {*} node
  */
 compileElementNode(node) {
   // 当是element节点时，继续调用compiler函数，深入编译
   this.compiler(node);
 }

 /**
  * 编译文本节点
  * @param {*} node
  */
 compileTextNode(node) {}
```

compileTextNode函数

```
/**
   * 编译文本节点
   * @param {*} node
   */
  compileTextNode(node) {
    let text = node.textContent.trim();
    if (text) {
      // 获取表达式

      // 添加订阅者，计算表达式的值
      // 当表达式依赖的数据变化时
      // 1.重新计算表达式的值
      // 2.node.textContent给到最新的值
      // 即可完成Model->View的响应式
    }
  }
```

- node.textContent 改变，页面实时动态变更
- 获取表达式
  - let exp = this.parseTextExp(text);
- parseTextExp函数

```
/**
   * 通过文本获取表达式
   * 1111{{msg+'----'}}2222的内容，则表达式应该为：
   * '1111' + msg + '----' + '2222'
   * @param {*} text
   */
  parseTextExp(text) {
    //  (非贪婪匹配) 匹配插值表达式正则
    var regText = /\{\{(.+?)\}\}/g;
    // 分割插值表达式前，后的内容
    var pieces = text.split(regText);
    // 匹配插值表达式
    var matches = text.match(regText);
    // 设置表达式数组
    var tokens = [];
    // 循环分割后的数组
    pieces.forEach(item => {
      // 当匹配到插值表达式，则只需要把数据添加到数组里即可，加上()是为了优先计算
      if (matches && matches.indexOf("{{" + item + "}}") > -1) {
        tokens.push("(" + item + ")");
      } else {
        // 当不是插值表达式时，则给数据加上``，当成字符串，添加到表达式数组
        tokens.push("`" + item + "`");
      }
    });
    // 最后数组使用+拼接，即可获取完整的表达式
    return tokens.join("+");
  }
```

- 完成compileTextNode函数

```
if (text) {
    // 获取表达式
    let exp = this.parseTextExp(text);
    // 添加订阅者，计算表达式的值
    // 当表达式依赖的数据变化时
    // 1.重新计算表达式的值
    // 2.node.textContent给到最新的值
    // 即可完成Model->View的响应式

    // 传递三个参数
    // 表达式，作用域，回调函数
    new Watcher(exp, this.context, (newValue) => {
        node.textContent = newValue
    })
}
```

Watcher

```
var $uid = 0;
export default class Watcher {
  constructor(exp, scope, cb) {
    this.exp = exp;
    this.scope = scope;
    this.cb = cb;
    this.uid = $uid++;
    this.update();
  }

  /**
   * 通过exp表达式求值
   */
  get() {
    // 定义求值函数
    var value = Watcher.computeExpression(this.exp, this.scope)
    return value
  }

  /**
   * 完成回调函数的调用
   * 把求出来的最新的值传递给回调函数
   */
  update() {
    var newValue = this.get()
    this.cb && this.cb(newValue)
  }

  static computeExpression(exp, scope) {
    // 创建函数
    // scope当作函数参数传递
    // 函数内部使用with指明作用域
```

```
    // 函数返回表达式计算后的值
    var fn = new Function("scope", "with(scope){return " + exp + "}");
    return fn(scope);
  }
}
```

Dep

```
export default class Dep {
  constructor() {
    // 存放所有watcher
    this.subs = {};
  }

  // 添加watcher到subs中
  addSub(target) {
    if (!this.subs[target.uid]) {
      this.subs[target.uid] = target;
    }
  }

  // 循环所有subs，执行update方法
  notify() {
    for (let uid in this.subs) {
      this.subs[uid].update();
    }
  }
}
```

改造Observer和Watcher

给每一个被劫持的数据，都添加一个Dep依赖列表，因为每一个数据可能在很多地方都会被调用

所以数据与模板，是一对多的关系

数据劫持到，当数据被set时，执行dep.notify方法

现在的问题是，什么时候往dep中添加watcher，这里Vue用了一个小技巧：我们想要的效果是，数据每当被页面用到了，那我们就把当前的watcher添加到dep，而我们的数据会表达式计算求值，然后再绑定到页面上，所以，当我们表达式求值的时候，把watcher添加到dep即可。

```
// watcher.js
get() {
    // 定义求值函数
    Dep.target = this;
    var value = Watcher.computeExpression(this.exp, this.scope);
    Dep.target = null;
    return value
}

// observer.js
defineReactive(data, key, value) {
```

```
    var dep = new Dep();
    Object.defineProperty(data, key, {
        // 可遍历
        enumerable: true,
        // 不可再配置
        configurable: false,
        get: () => {
            console.log('触发了get')
            Dep.target && dep.addSub(Dep.target)
            return value;
        },
        set: newValue => {
            if (value === newValue) {
                return;
            }
            console.log('触发了set')
            // TODO 触发View的变化
            value = newValue;
            // 数据更新后，通知所有依赖watcher，重新计算页面展示
            dep.notify();
        }
    });
    // 继续往下层编译添加响应式
    this.walk(value)
}
```

到现在为止，我们基本完成了基本核心功能，但是，指令，函数我们还没有添加，现在我们添加两个指令

v-text

```
  compileElementNode(node) {
    // 指令即是自定义属性，解构赋值，获取所有属性
    var attrs = [...node.attributes];
    attrs.forEach(attr => {
      // 解构赋值，获取属性的名称和值
      let { name: attrName, value: attrValue } = attr;
      if (attrName.indexOf("v-") > -1) {
        // 截取v-后面的部分
        let dirName = attrName.slice(2);
        switch (dirName) {
          case "text":
            // 当是v-text，则属性的value就是表达式
            new Watcher(attrValue, this.context, newValue => {
              node.textContent = newValue;
            });
            break;
        }
      }
    });
    // 当是element节点时，继续调用compiler函数，深入编译
    this.compiler(node);
  }
```

v-model

```
case "model":
// 先做一个判断，是不是input，其他情况暂时不考虑
if (node.tagName.toLowerCase() === "input") {
    new Watcher(attrValue, this.context, newValue => {
        // 因为是input，这里是value
        // 这时候可以做到数据变化页面变化
        node.value = newValue;
    });
    // 添加事件监听
    node.addEventListener("input", e => {
        that.context[attrValue] = e.target.value;
    })
}
break;
```

click事件函数

```
// 在if (attrName.indexOf("v-") > -1)下添加
// 解析事件
if (attrName.indexOf("@") === 0) {
    this.compileMethods(this.context, node, attrName, attrValue);
}

 // 外层书写
 /**
   * 解析事件函数
   * @param {*} node
   * @param {*} attrName
   * @param {*} attrValue
   */
  compileMethods(scope, node, attrName, attrValue) {
    // 获取事件类型
    let type = attrName.slice(1);
    // 获取函数，先只考虑不传参数的情况
    let fn = scope[attrValue]
    // 给对应的node节点添加事件
    node.addEventListener(type, fn.bind(scope))
  }
```