

谷粒商城

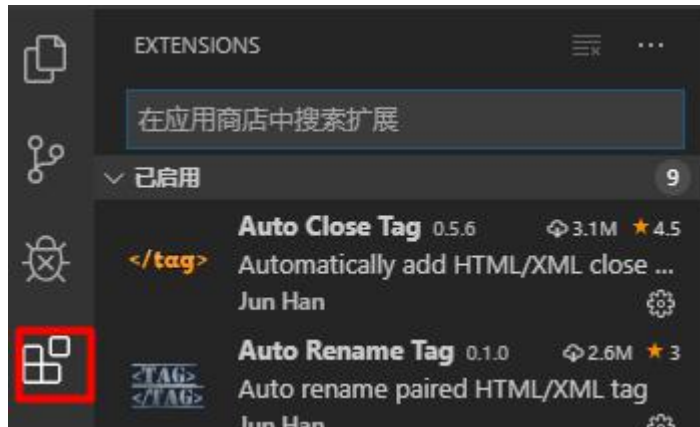
前端开发基础知识&快速入门



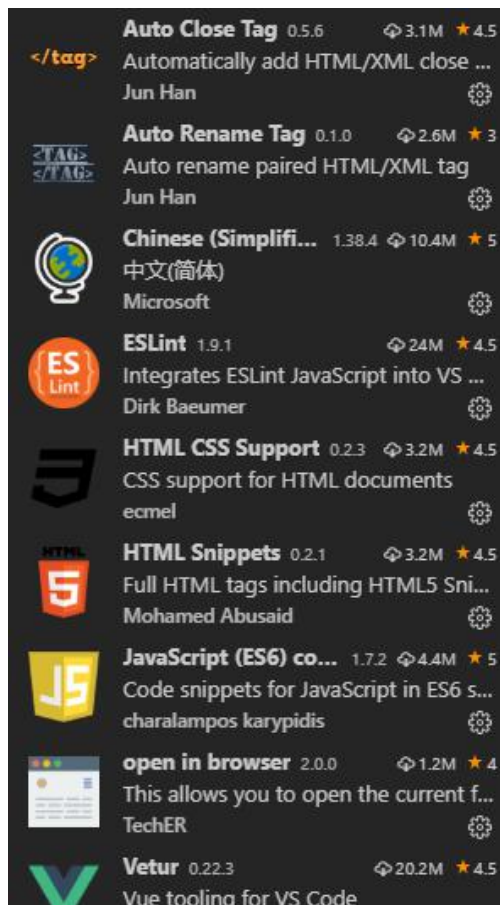
一、VSCode 使用

1、安装常用插件

切换到插件标签页



安装以下基本插件



2、创建项目

vscode 很轻量级，本身没有新建项目的选项，创建一个空文件夹就可以当做一个项目



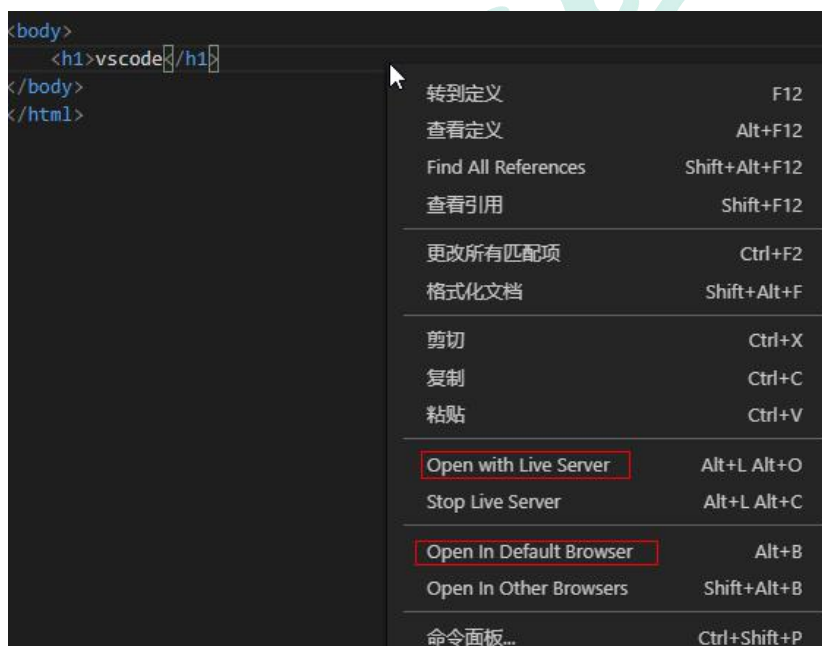
3、创建网页

创建文件，命名为 index.html

快捷键 !，快速创建网页模板

h1 + 回车，自动补全标签

4、运行效果

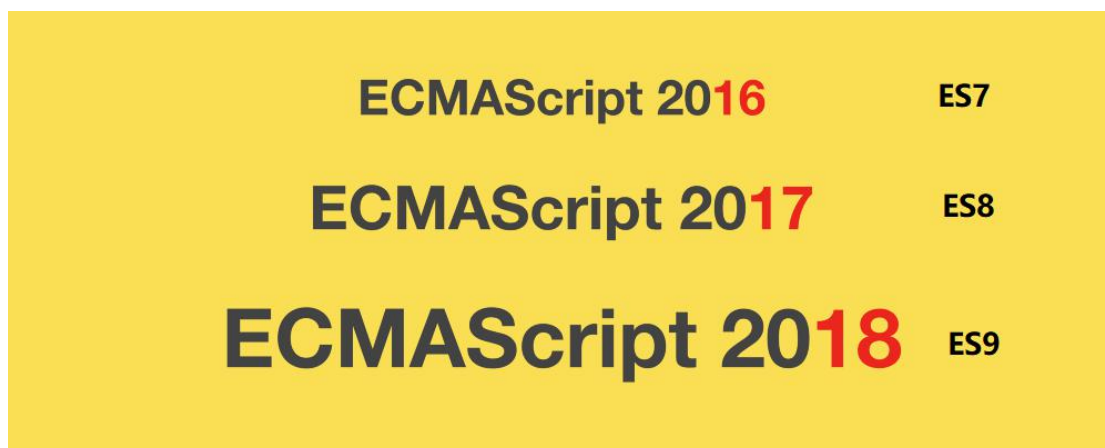


如果使用 live server，页面内容变化，保存以后，浏览器会自动变化；

二、ES6

1、简介

ECMAScript 6.0（以下简称 ES6，ECMAScript 是一种由 Ecma 国际(前身为欧洲计算机制造商协会,英文名称是 European Computer Manufacturers Association)通过 ECMA-262 标准化的脚本程序设计语言）是 **JavaScript 语言的下一代标准**，已经在 2015 年 6 月正式发布了，并且从 ECMAScript 6 开始，开始采用年号来做版本。即 ECMAScript 2015，就是 ECMAScript6。它的目标，是使得 JavaScript 语言可以用来编写复杂的大型应用程序，成为企业级开发语言。**每年一个新版本。**



2、什么是 ECMAScript

来看下前端的发展历程：

- web1.0 时代：

最初的网页以 HTML 为主，是纯静态的网页。网页是只读的，信息流只能从服务的到客户端单向流通。开发人员也只关心页面的样式和内容即可。

- web2.0 时代：

- 1995 年，网景工程师 Brendan Eich 花了 10 天时间设计了 JavaScript 语言。
- 1996 年，微软发布了 JScript，其实是 JavaScript 的逆向工程实现。
- 1996 年 11 月，JavaScript 的创造者 Netscape 公司，决定将 JavaScript 提交给标准化组织 ECMA，希望这种语言能够成为国际标准。
- 1997 年，ECMA 发布 262 号标准文件（ECMA-262）的第一版，规定了浏览器脚本语言的标准，并将这种语言称为 ECMAScript，这个版本就是 1.0 版。JavaScript 和 JScript 都是‘ECMAScript’的标准实现者，随后各大浏览器厂商纷纷实现了‘ECMAScript’标准。

所以，ECMAScript 是浏览器脚本语言的规范，而各种我们熟知的 js 语言，如 JavaScript 则是规范的具体实现。

3、ES6 新特性

1、let 声明变量

```
// var 声明的变量往往会越域
// let 声明的变量有严格局部作用域
{
    var a = 1;
    let b = 2;
}
console.log(a); // 1
console.log(b); // ReferenceError: b is not defined

// var 可以声明多次
// let 只能声明一次
var m = 1
var m = 2
let n = 3
// let n = 4
console.log(m) // 2
console.log(n) // Identifier 'n' has already been declared

// var 会变量提升
// let 不存在变量提升
console.log(x); // undefined
var x = 10;
console.log(y); //ReferenceError: y is not defined
let y = 20;
```

2、const 声明常量（只读变量）

```
// 1. 声明之后不允许改变
// 2. 一但声明必须初始化，否则会报错
const a = 1;
a = 3; //Uncaught TypeError: Assignment to constant variable.
```

3、解构表达式

1)、数组解构

```
let arr = [1,2,3];  
//以前我们想获取其中的值，只能通过角标。ES6 可以这样：  
const [x,y,z] = arr;// x, y, z 将与 arr 中的每个位置对应来取值  
// 然后打印  
console.log(x,y,z);
```

2)、对象解构

```
const person = {  
  name: "jack",  
  age: 21,  
  language: ['java', 'js', 'css']  
}  
// 解构表达式获取值，将 person 里面每一个属性和左边对应赋值  
const { name, age, language } = person;  
// 等价于下面  
// const name = person.name;  
// const age = person.age;  
// const language = person.language;  
// 可以分别打印  
console.log(name);  
console.log(age);  
console.log(language);
```

```
//扩展：如果想要将 name 的值赋值给其他变量，可以如下,nn 是新的变量名  
const { name: nn, age, language } = person;  
console.log(nn);  
console.log(age);  
console.log(language);
```

4、字符串扩展

1)、几个新的 API

ES6 为字符串扩展了几个新的 API:

- `includes()`: 返回布尔值, 表示是否找到了参数字符串。
- `startsWith()`: 返回布尔值, 表示参数字符串是否在原字符串的头部。
- `endsWith()`: 返回布尔值, 表示参数字符串是否在原字符串的尾部。

```
let str = "hello.vue";
console.log(str.startsWith("hello")); // true
console.log(str.endsWith(".vue")); // true
console.log(str.includes("e")); // true
console.log(str.includes("hello")); // true
```

2)、字符串模板

模板字符串相当于加强版的字符串, 用反引号 ```, 除了作为普通字符串, 还可以用来定义多行字符串, 还可以在字符串中加入变量和表达式。

```
// 1、多行字符串
let ss = `
    <div>
        <span>hello world</span>
    </div>
`;
console.log(ss)
```

// 2、字符串插入变量和表达式。变量名写在 `${}` 中, `${}` 中可以放入 JavaScript 表达式。

```
let name = "张三";
let age = 18;
let info = `我是${name}, 今年${age}岁了`;
console.log(info)
```

// 3、字符串中调用函数

```
function fun() {
    return "这是一个函数"
}
let sss = `0(n_n)0 哈哈~, ${fun()}`;
console.log(sss); // 0(n_n)0 哈哈~, 这是一个函数
```

5、函数优化

1)、函数参数默认值

```
//在 ES6 以前，我们无法给一个函数参数设置默认值，只能采用变通写法：
function add(a, b) {
    // 判断 b 是否为空，为空就给默认值 1
    b = b || 1;
    return a + b;
}
// 传一个参数
console.log(add(10));

//现在可以这么写：直接给参数写上默认值，没传就会自动使用默认值
function add2(a, b = 1) {
    return a + b;
}
// 传一个参数
console.log(add2(10));
```

2)、不定参数

不定参数用来表示不确定参数个数，形如，...变量名，由...加上一个具名参数标识符组成。具名参数只能放在参数列表的最后，并且有且只有一个不定参数

```
function fun(...values) {
    console.log(values.length)
}
fun(1, 2) //2
fun(1, 2, 3, 4) //4
```

3)、箭头函数

ES6 中定义函数的简写方式

- 一个参数时：


```
//以前声明一个方法
// var print = function (obj) {
//     console.log(obj);
// }
// 可以简写为:
var print = obj => console.log(obj);
// 测试调用
print(100);
```

- 多个参数:

```
// 两个参数的情况:
var sum = function (a, b) {
    return a + b;
}
// 简写为:
//当只有一行语句, 并且需要返回结果时, 可以省略 {}, 结果会自动返回。
var sum2 = (a, b) => a + b;
//测试调用
console.log(sum2(10, 10)); //20

// 代码不止一行, 可以用`{}`括起来
var sum3 = (a, b) => {
    c = a + b;
    return c;
};
//测试调用
console.log(sum3(10, 20)); //30
```

4)、实战: 箭头函数结合解构表达式

```
//需求, 声明一个对象, hello 方法需要对象的个别属性
//以前的方式:
const person = {
    name: "jack",
    age: 21,
    language: ['java', 'js', 'css']
}

function hello(person) {
    console.log("hello," + person.name)
```

```

    }
    //现在的方式
    var hello2 = ({ name }) => { console.log("hello," + name) };
    //测试
    hello2(person);

```

6、对象优化

1)、新增的 API

ES6 给 Object 拓展了许多新的方法，如：

- keys(obj): 获取对象的所有 key 形成的数组
- values(obj): 获取对象的所有 value 形成的数组
- entries(obj): 获取对象的所有 key 和 value 形成的二维数组。格式：`[[k1,v1],[k2,v2],...]`
- assign(dest, ...src) : 将多个 src 对象的值 拷贝到 dest 中。（第一层为深拷贝，第二层为浅拷贝）

```

const person = {
  name: "jack",
  age: 21,
  language: ['java', 'js', 'css']
}

console.log(Object.keys(person));//[ "name", "age", "language" ]
console.log(Object.values(person));//[ "jack", 21, Array(3) ]
console.log(Object.entries(person));//[ Array(2), Array(2), Arra
y(2) ]

```

```

const target = { a: 1 };
const source1 = { b: 2 };
const source2 = { c: 3 };
//Object.assign 方法的第一个参数是目标对象，后面的参数都是源对象。
Object.assign(target, source1, source2);
console.log(target)//{a: 1, b: 2, c: 3}

```

2)、声明对象简写

```

const age = 23
const name = "张三"

```

```
// 传统
const person1 = { age: age, name: name }
console.log(person1)

// ES6: 属性名和属性值变量名一样, 可以省略
const person2 = { age, name }
console.log(person2) //{age: 23, name: "张三"}
```

3)、对象的函数属性简写

```
let person = {
  name: "jack",
  // 以前:
  eat: function (food) {
    console.log(this.name + "在吃" + food);
  },
  // 箭头函数版: 这里拿不到 this
  eat2: food => console.log(person.name + "在吃" + food),
  // 简写版:
  eat3(food) {
    console.log(this.name + "在吃" + food);
  }
}
person.eat("apple");
```

4)、对象拓展运算符

拓展运算符 (...) 用于取出参数对象所有可遍历属性然后拷贝到当前对象。

```
// 1、拷贝对象 (深拷贝)
let person1 = { name: "Amy", age: 15 }
let someone = { ...person1 }
console.log(someone) //{name: "Amy", age: 15}

// 2、合并对象
let age = { age: 15 }
let name = { name: "Amy" }
let person2 = { ...age, ...name } //如果两个对象的字段名重复, 后面对象字段值会覆盖前面对象的字段值
console.log(person2) //{age: 15, name: "Amy"}
```

7、map 和 reduce

数组中新增了 map 和 reduce 方法。

1)、map

map(): 接收一个函数，将原数组中的所有元素用这个函数处理后放入新数组返回。

```
let arr = ['1', '20', '-5', '3'];
console.log(arr)

arr = arr.map(s => parseInt(s));
console.log(arr)
```

2)、reduce

语法:

`arr.reduce(callback,[initialValue])`

reduce 为数组中的每一个元素依次执行回调函数，不包括数组中被删除或从未被赋值的元素，接受四个参数：初始值（或者上一次回调函数的返回值），当前元素值，当前索引，调用 reduce 的数组。

callback （执行数组中每个值的函数，包含四个参数）

- 1、previousValue （上一次调用回调返回的值，或者是提供的初始值（initialValue））
- 2、currentValue （数组中当前被处理的元素）
- 3、index （当前元素在数组中的索引）
- 4、array （调用 reduce 的数组）

initialValue （作为第一次调用 callback 的第一个参数。）

示例:

```
const arr = [1,20,-5,3];
//没有初始值:
console.log(arr.reduce((a,b)=>a+b)); //19
console.log(arr.reduce((a,b)=>a*b)); // -300
```

```
//指定初始值:  
console.log(arr.reduce((a,b)=>a+b,1)); //20  
console.log(arr.reduce((a,b)=>a*b,0)); //-0
```

8、Promise

在 JavaScript 的世界中，所有代码都是单线程执行的。由于这个“缺陷”，导致 JavaScript 的所有网络操作，浏览器事件，都必须是异步执行。异步执行可以用回调函数实现。一旦有一连串的 ajax 请求 a,b,c,d... 后面的请求依赖前面的请求结果，就需要层层嵌套。这种缩进和层层嵌套的方式，非常容易造成上下文代码混乱，我们不得不非常小心翼翼处理内层函数与外层函数的数据，一旦内层函数使用了上层函数的变量，这种混乱程度就会加剧.....总之，这种‘层叠上下文’的层层嵌套方式，着实增加了神经的紧张程度。

案例：用户登录，并展示该用户的各科成绩。在页面发送两次请求：

1. 查询用户，查询成功说明可以登录
2. 查询用户成功，查询科目
3. 根据科目的查询结果，获取去成绩

分析：此时后台应该提供三个接口，一个提供用户查询接口，一个提供科目的接口，一个提供各科成绩的接口，为了渲染方便，最好响应 json 数据。在这里就不编写后台接口了，而是提供三个 json 文件，直接提供 json 数据，模拟后台接口：

user.json:

```
{  
  "id": 1,  
  "name": "zhangsan",  
  "password": "123456"  
}
```

user_corse_1.json:

```
{  
  "id": 10,  
  "name": "chinese"  
}
```

corse_score_10.json:

```
{  
  "id": 100,  
  "score": 90  
}
```

```
//回调函数嵌套的噩梦：层层嵌套。
```

```
$.ajax({
  url: "mock/user.json",
  success(data) {
    console.log("查询用户: ", data);
    $.ajax({
      url: `mock/user_corse_${data.id}.json`,
      success(data) {
        console.log("查询到课程: ", data);
        $.ajax({
          url: `mock/corse_score_${data.id}.json`,
          success(data) {
            console.log("查询到分数: ", data);
          },
          error(error) {
            console.log("出现异常了: " + error);
          }
        });
      },
      error(error) {
        console.log("出现异常了: " + error);
      }
    });
  },
  error(error) {
    console.log("出现异常了: " + error);
  }
});
```

我们可以通过 Promise 解决以上问题。

1)、Promise 语法

```
const promise = new Promise(function (resolve, reject) {
  // 执行异步操作
  if (/* 异步操作成功 */) {
    resolve(value); // 调用 resolve, 代表 Promise 将返回成功的结果
  } else {
    reject(error); // 调用 reject, 代表 Promise 会返回失败结果
  }
});
```

使用箭头函数可以简写为:

```
const promise = new Promise((resolve, reject) =>{
  // 执行异步操作
  if (/* 异步操作成功 */) {
    resolve(value); // 调用 resolve, 代表 Promise 将返回成功的结果
  } else {
    reject(error); // 调用 reject, 代表 Promise 会返回失败结果
  }
});
```

这样, 在 promise 中就封装了一段异步执行的结果。

2)、处理异步结果

如果我们想要等待异步执行完成, 做一些事情, 我们可以通过 promise 的 then 方法来实现。如果想要处理 promise 异步执行失败的事件, 还可以跟上 catch:

```
promise.then(function (value) {
  // 异步执行成功后的回调
}).catch(function (error) {
  // 异步执行失败后的回调
});
```

3)、Promise 改造以前嵌套方式

```
new Promise((resolve, reject) => {
  $.ajax({
    url: "mock/user.json",
    success(data) {
      console.log("查询用户: ", data);
      resolve(data.id);
    },
    error(error) {
      console.log("出现异常了: " + error);
    }
  });
}).then((userId) => {
  return new Promise((resolve, reject) => {
    $.ajax({
      url: `mock/user_corse_${userId}.json`,
```

```
        success(data) {
            console.log("查询到课程: ", data);
            resolve(data.id);
        },
        error(error) {
            console.log("出现异常了: " + error);
        }
    });
});
}).then((courseId) => {
    console.log(courseId);

    $.ajax({
        url: `mock/course_score_${courseId}.json`,
        success(data) {
            console.log("查询到分数: ", data);
        },
        error(error) {
            console.log("出现异常了: " + error);
        }
    });
});
});
```

4)、优化处理

优化：通常在企业开发中，会把 promise 封装成通用方法，如下：封装了一个通用的 get 请求方法：

```
let get = function (url, data) { // 实际开发中会单独放到 common.js 中
    return new Promise((resolve, reject) => {
        $.ajax({
            url: url,
            type: "GET",
            data: data,
            success(result) {
                resolve(result);
            },
            error(error) {
                reject(error);
            }
        });
    });
}
```



```
// 使用封装的 get 方法，实现查询分数
get("mock/user.json").then((result) => {
    console.log("查询用户: ", result);
    return get(`mock/user_corse_${result.id}.json`);
}).then((result) => {
    console.log("查询到课程: ", result);
    return get(`mock/corse_score_${result.id}.json`)
}).then((result) => {
    console.log("查询到分数: ", result);
}).catch(() => {
    console.log("出现异常了: " + error);
});
```

通过比较，我们知道了 Promise 的扁平化设计理念，也领略了这种‘上层设计’带来的好处。我们的项目中会使用到这种异步处理的方式：

9、模块化

1)、什么是模块化

模块化就是把代码进行拆分，方便重复利用。类似 java 中的导包：要使用一个包，必须先导包。而 JS 中没有包的概念，换来的是 **模块**。

模块功能主要由两个命令构成：‘export’和‘import’。

- ‘export’命令用于规定模块的对外接口。
- ‘import’命令用于导入其他模块提供的功能。

2)、export

比如我定义一个 js 文件:hello.js，里面有一个对象

```
const util = {
    sum(a,b){
        return a + b;
    }
}
```

我可以使用 export 将这个对象导出：

```
const util = {
    sum(a,b){
```

```
        return a + b;
    }
}

export {util};
```

当然，也可以简写为：

```
export const util = {
    sum(a,b){
        return a + b;
    }
}
```

‘export’不仅可以导出对象，一切 JS 变量都可以导出。比如：基本类型变量、函数、数组、对象。

当要导出多个值时，还可以简写。比如我有一个文件：user.js:

```
var name = "jack"
var age = 21
export {name,age}
```

省略名称

上面的导出代码中，都明确指定了导出的变量名，这样其它人在导入使用时就必须准确写出变量名，否则就会出错。

因此 js 提供了‘default’关键字，可以对导出的变量名进行省略

例如：

```
// 无需声明对象的名字
export default {
    sum(a,b){
        return a + b;
    }
}
```

这样，当使用者导入时，可以任意起名字

3)、import

使用‘export’命令定义了模块的对外接口以后，其他 JS 文件就可以通过‘import’命令加载这个模块。

例如我要使用上面导出的 util:

```
// 导入 util
```

```
import util from 'hello.js'  
// 调用 util 中的属性  
util.sum(1,2)
```

要批量导入前面导出的 name 和 age:

```
import {name, age} from 'user.js'  
console.log(name + " , 今年"+ age +"岁了")
```

但是上面的代码暂时无法测试，因为浏览器目前还不支持 ES6 的导入和导出功能。除非借助于工具，把 ES6 的语法进行编译降级到 ES5，比如`Babel-cli`工具我们暂时不做测试，大家了解即可。

三、Node.js

前端开发，少不了 node.js；Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行环境。

<http://nodejs.cn/api/>

我们关注与 node.js 的 npm 功能就行；

NPM 是随同 NodeJS 一起安装的**包管理**工具，JavaScript-NPM，Java-Maven；

1)、官网下载安装 node.js，并使用 node -v 检查版本

2)、配置 npm 使用淘宝镜像

npm config set registry <http://registry.npm.taobao.org/>

3)、大家如果 npm install 安装依赖出现 chromedriver 之类问题，先在项目里运行下面命令
npm install chromedriver --chromedriver_cdnurl=http://cdn.npm.taobao.org/dist/chromedriver
然后再运行 npm install

四、Vue

1、MVVM 思想

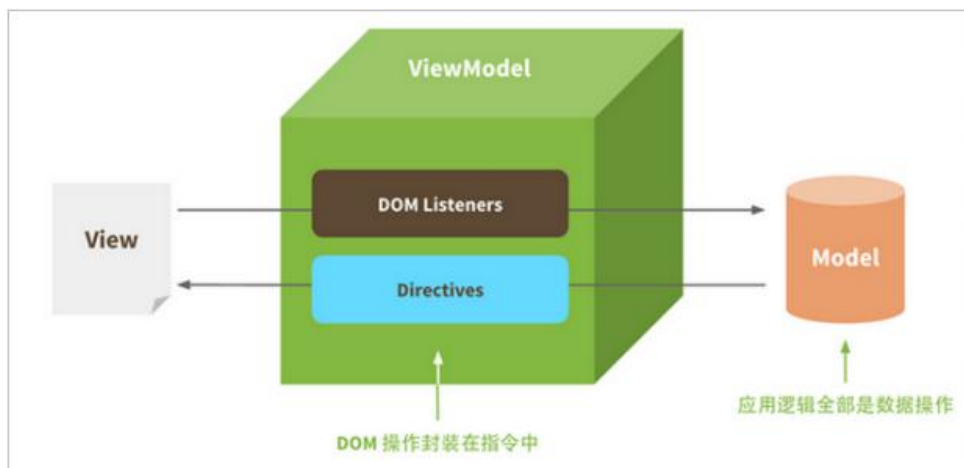
- M: 即 Model，模型，包括数据和一些基本操作
- V: 即 View，视图，页面渲染结果
- VM: 即 View-Model，模型与视图间的双向操作（无需开发人员干涉）

在 MVVM 之前，开发人员从后端获取需要的数据模型，然后通过 DOM 操作 Model 渲染到 View 中。而后当用户操作视图，我们还需要通过 DOM 获取 View 中的数据，然后同步到 Model 中。

而 MVVM 中的 VM 要做的事情就是把 DOM 操作完全封装起来，开发人员不用再关心 Model 和 View 之间是如何互相影响的：

- 只要我们 Model 发生了改变，View 上自然就会表现出来。
- 当用户修改了 View，Model 中的数据也会跟着改变。

把开发人员从繁琐的 DOM 操作中解放出来，把关注点放在如何操作 Model 上。



2、Vue 简介

Vue (读音 /vju:/，类似于 view) 是一套用于构建用户界面的渐进式框架。与其它大型框架不同的是，Vue 被设计为可以自底向上逐层应用。Vue 的核心库只关注视图层，不仅易于上手，还便于与第三方库或既有项目整合。另一方面，当与现代化的工具链以及各种支持类库结合使用时，Vue 也完全能够为复杂的单页应用提供驱动。

官网：<https://cn.vuejs.org/>

参考：<https://cn.vuejs.org/v2/guide/>

Git 地址：<https://github.com/vuejs>

尤雨溪，Vue.js 创作者，Vue Technology 创始人，致力于 Vue 的研究开发。

3、入门案例

1)、安装

官网文档提供了 3 中安装方式：

1. 直接 script 引入本地 vue 文件。需要通过官网下载 vue 文件。
2. 通过 script 引入 CDN 代理。需要联网，生产环境可以使用这种方式

3. 通过 npm 安装。这种方式也是官网推荐的方式，需要 nodejs 环境。
本课程就采用第三种方式

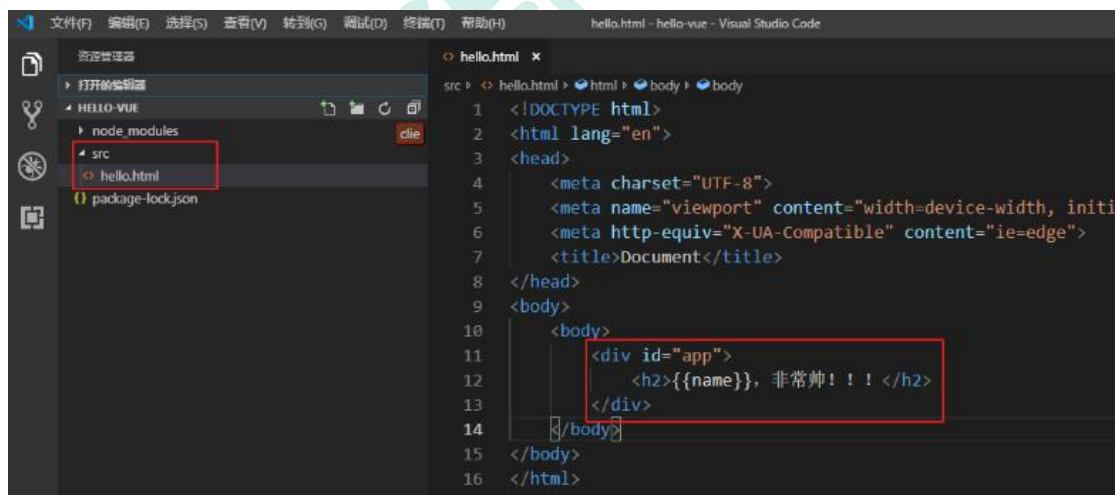
2)、创建示例项目

- 1、新建文件夹 hello-vue，并使用 vscode 打开
- 2、使用 vscode 控制台，npm install -y;
项目会生成 package-lock.json 文件，类似于 maven 项目的 pom.xml 文件。
- 3、使用 npm install vue，给项目安装 vue；项目下会多 node_modules 目录，并且在下面有一个 vue 目录。



3)、HelloWorld

在 hello.html 中，我们编写一段简单的代码。
h2 中要输出一句话：`xx 非常帅`。前面的`xx`是要渲染的数据。



4)、vue 声明式渲染

页面代码

```
<body>
  <div id="app">
```

```
<h1>{{name}}, 非常帅!!! </h1>
</div>

<script src="./node_modules/vue/dist/vue.min.js"></script>
<script>
  let vm = new Vue({
    el:"#app",
    data:{
      name: "张三"
    }
  });
</script>
</body>
```

- 首先通过 `new Vue()` 来创建 `Vue` 实例
- 然后构造函数接收一个对象，对象中有一些属性：
 - `el`: 是 `element` 的缩写，通过 `id` 选中要渲染的页面元素，本例中是一个 `div`
 - `data`: 数据，数据是一个对象，里面有很多属性，都可以渲染到视图中
 - ◆ `name`: 这里我们指定了一个 `name` 属性
- 页面中的 `'h2'` 元素中，我们通过 `{{name}}` 的方式，来渲染刚刚定义的 `name` 属性。

打开页面查看效果：



张三，非常帅!!!

更神奇的在于，当你修改 `name` 属性时，页面会跟着变化：

李四，非常帅!!!



5)、双向绑定

我们对刚才的案例进行简单修改：

```
<body>
  <div id="app">
    <input type="text" v-model="num">
    <h2>
      {{name}}, 非常帅!!! 有{{num}}个人为他点赞。
    </h2>
  </div>
  <script src="./node_modules/vue/dist/vue.js"></script>
  <script>
    // 创建 vue 实例
    let app = new Vue({
      el: "#app", // el 即 element, 该 vue 实例要渲染的页面元素
      data: { // 渲染页面需要的数据
        name: "张三",
        num: 5
      }
    });

  </script>
</body>
```

双向绑定：

效果：我们修改表单项，num 会发生变化。我们修改 num，表单项也会发生变化。为了实时观察到这个变化，我们将 num 输出到页面。

我们不需要关注他们为什么会建立起来关联，以及页面如何变化，我们只需要做好数据和视图的关联即可（MVVM）

张三,非常帅, 有1个人为他点赞

6)、事件处理

给页面添加一个按钮：

```
<body>
  <div id="app">
    <input type="text" v-model="num">
```

```

<button v-on:click="num++">关注</button>
<h2>
  {{name}}, 非常帅!!! 有{{num}}个人为他点赞。
</h2>
</div>
<script src="./node_modules/vue/dist/vue.js"></script>
<script>
  // 创建 vue 实例
  let app = new Vue({
    el: "#app", // el 即 element, 该 vue 实例要渲染的页面元素
    data: { // 渲染页面需要的数据
      name: "张三",
      num: 5
    }
  });

</script>
</body>

```

10

点赞

张三,非常帅, 有10个人为他点赞

- 这里用`v-on`指令绑定点击事件，而不是普通的`onclick`，然后直接操作 num
- 普通 click 是无法直接操作 num 的。
- 未来我们会见到更多 v-xxx，这些都是 vue 定义的不同功能的指令。

简单使用总结：

- 1)、使用 Vue 实例管理 DOM
- 2)、DOM 与数据/事件等进行相关绑定
- 3)、我们只需要关注数据，事件等处理，无需关心视图如何进行修改

4、概念

1、创建 Vue 实例

每个 Vue 应用都是通过用 Vue 函数创建一个新的 Vue 实例开始的：

```

let app = new Vue({
  // ...
});

```


在构造函数中传入一个对象，并且在对象中声明各种 Vue 需要的数据和方法，包括：

- el
- data
- methods

等等

接下来我们一一介绍。

2、模板或元素

每个 Vue 实例都需要关联一段 Html 模板，Vue 会基于此模板进行视图渲染。

我们可以通过 el 属性来指定。

例如一段 html 模板：

```
<div id="app">  
  
</div>
```

然后创建 Vue 实例，关联这个 div

```
let vm = new Vue({  
  el: "#app"  
})
```

这样，Vue 就可以基于 id 为`app`的 div 元素作为模板进行渲染了。在这个 div 范围以外的部分是无法使用 vue 特性的。

3、数据

当 Vue 实例被创建时，它会尝试获取在 **data** 中定义的所有属性，用于视图的渲染，并且监视 **data** 中的属性变化，当 **data** 发生改变，所有相关的视图都将重新渲染，这就是“响应式”系统。

html:

```
<div id="app">  
  <input type="text" v-model="name" />  
</div>
```

JS:

```
let vm = new Vue({  
  el: "#app",  
  data: {  
    name: "刘德华"
```

```
    }
  })
}
```

- name 的变化会影响到`input`的值
- input 中输入的值，也会导致 vm 中的 name 发生改变

4、方法

Vue 实例中除了可以定义 data 属性，也可以定义方法，并且在 Vue 实例的作用范围内使用。

Html:

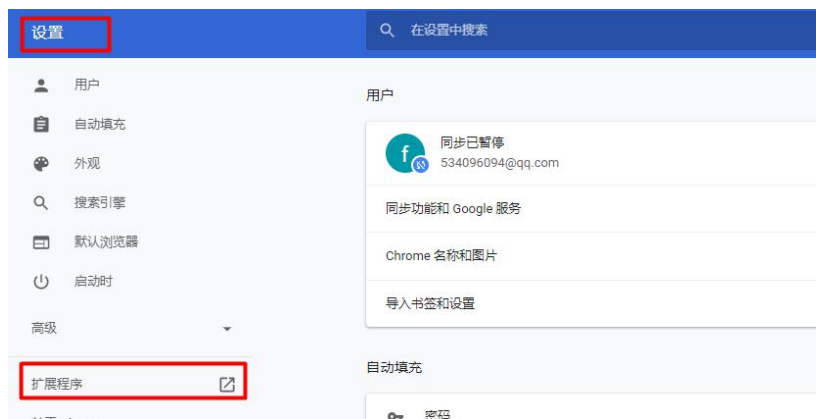
```
<div id="app">
  {{num}}
  <button v-on:click="add">加</button>
</div>
```

JS:

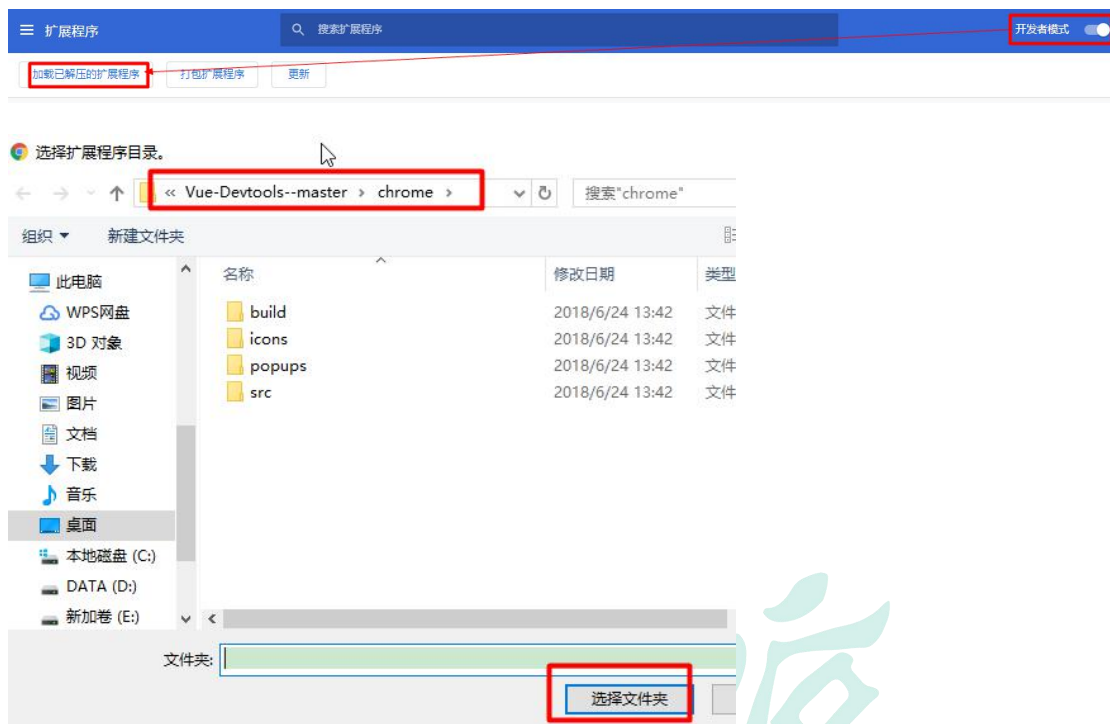
```
let vm = new Vue({
  el: "#app",
  data: {
    num: 0
  },
  methods: {
    add: function () {
      // this 代表的当前 vue 实例
      this.num++;
    }
  }
})
```

5、安装 vue-devtools 方便调试

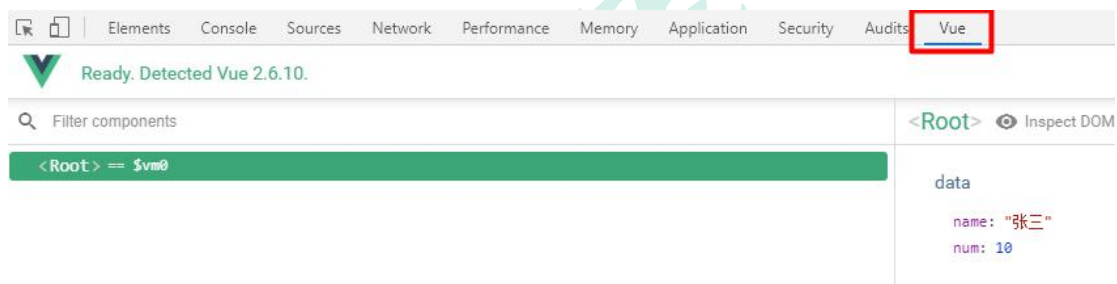
- 将软件包中的 vue-devtools 解压。
- 打开 chrome 设置->扩展程序



- 开启开发者模式，并加载插件



- 打开浏览器控制台，选择 vue



6、安装 vscode 的 vue 插件



安装这个插件就可以有语法提示

5、指令

什么是指令？

- 指令 (Directives) 是带有 `v-` 前缀的特殊特性。
- 指令特性的预期值是：**单个 JavaScript 表达式**。
- 指令的职责是，当表达式的值改变时，将其产生的连带影响，响应式地作用于 DOM。

例如我们在入门案例中的 `v-on`，代表绑定事件。

1、插值表达式

1)、花括号

格式: `{{表达式}}`

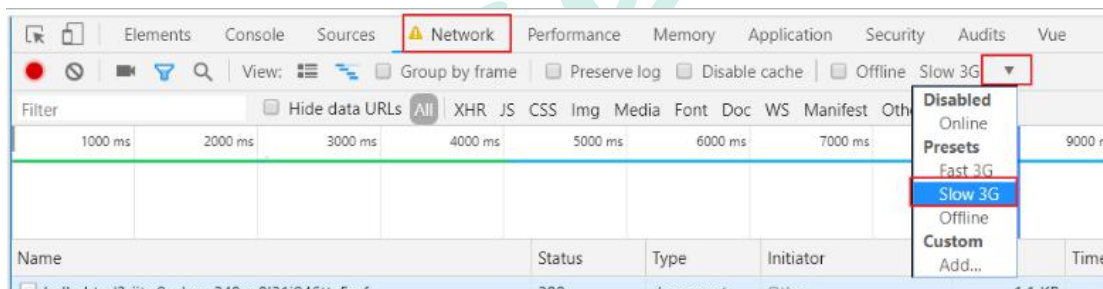
说明:

- 该表达式支持 JS 语法，可以调用 js 内置函数（必须有返回值）
- 表达式必须有返回结果。例如 `1 + 1`，没有结果的表达式不允许使用，如: `let a = 1 + 1;`
- 可以直接获取 Vue 实例中定义的数据或函数

2)、插值闪烁

使用 `{{}}` 方式在网速较慢时会出现问题。在数据未加载完成时，页面会显示出原始的 `{{}}`，加载完毕后才显示正确数据，我们称为插值闪烁。

我们将网速调慢一些，然后刷新页面，试试看刚才的案例：



3)、v-text 和 v-html

可以使用 `v-text` 和 `v-html` 指令来替代 `{{}}`

说明:

- `v-text`: 将数据输出到元素内部，如果输出的数据有 HTML 代码，会作为普通文本输出
- `v-html`: 将数据输出到元素内部，如果输出的数据有 HTML 代码，会被渲染

示例:

```
<div id="app">
  v-text:<span v-text="hello"></span> <br />
  v-html:<span v-html="hello"></span>
</div>
```

```
<script src="./node_modules/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      hello: "<h1>大家好</h1>"
    }
  })
</script>
```

效果:

v-text:<h1>大家好</h1>
v-html:

大家好

并且不会出现插值闪烁，当没有数据时，会显示空白或者默认数据。

2、v-bind

html 属性不能使用双大括号形式绑定，我们使用 v-bind 指令给 HTML 标签属性绑定值；而且在将 'v-bind' 用于 'class' 和 'style' 时，Vue.js 做了专门的增强。

1)、绑定 class

```
<div class="static" v-bind:class="{ active: isActive, 'text-danger'
: hasError }">
</div>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      isActive: true,
      hasError: false
    }
  })
</script>
```

2)、绑定 style

`v-bind:style` 的对象语法十分直观，看着非常像 CSS，但其实是一个 JavaScript 对象。style 属性名可以用驼峰式 (camelCase) 或短横线分隔 (kebab-case，这种方式记得用单引号括起来) 来命名。

例如：font-size-->fontSize

```
<div id="app" v-bind:style="{ color: activeColor, fontSize: fontSize + 'px' }"></div>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      activeColor: 'red',
      fontSize: 30
    }
  })
</script>
```

结果：<div style="color: red; font-size: 30px;"></div>

3)、绑定其他任意属性

```
<div id="app" v-bind:style="{ color: activeColor, fontSize: fontSize + 'px' }"
  v-bind:user="userName">
</div>
<script>
  let vm = new Vue({
    el: "#app",
    data: {
      activeColor: 'red',
      fontSize: 30,
      userName: 'zhangsan'
    }
  })
</script>
```

效果：

<div id="app" user="zhangsan" style="color: red; font-size: 30px;"></div>

4)、v-bind 缩写

```
<div id="app" :style="{ color: activeColor, fontSize: fontSize + 'px' }" :user="userName">
</div>
```

3、v-model

刚才的 v-text、v-html、v-bind 可以看做是单向绑定，数据影响了视图渲染，但是反过来就不行。接下来学习的 v-model 是双向绑定，视图（View）和模型（Model）之间会互相影响。

既然是双向绑定，一定是在视图中可以修改数据，这样就限定了视图的元素类型。目前 v-model 的可使用元素有：

- input
- select
- textarea
- checkbox
- radio
- components（Vue 中的自定义组件）

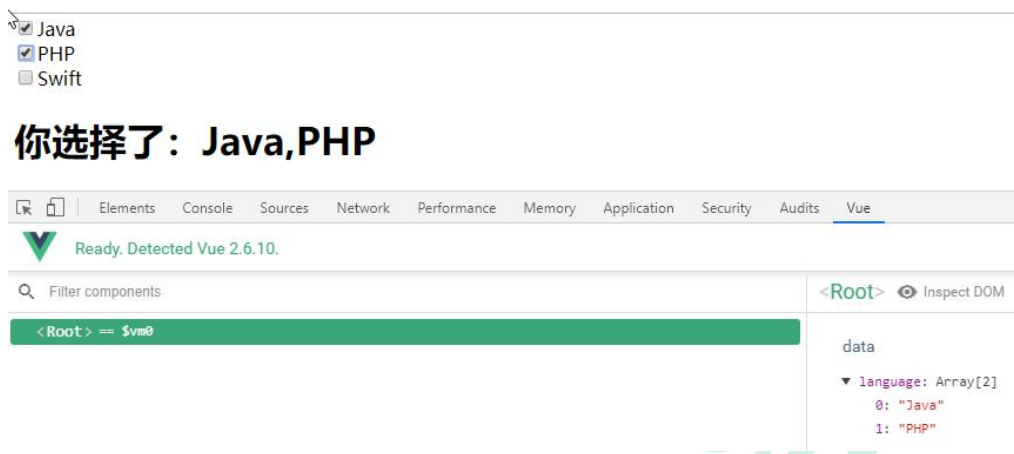
基本上除了最后一项，其它都是表单的输入项。

示例：

```
<div id="app">
  <input type="checkbox" v-model="language" value="Java" />Java<br />
  <input type="checkbox" v-model="language" value="PHP" />PHP<br />
  <input type="checkbox" v-model="language" value="Swift" />Swift<br />
  <h1>
    你选择了：{{language.join(',')}}
  </h1>
</div>
<script src="../../node_modules/vue/dist/vue.js"></script>
<script type="text/javascript">
  let vm = new Vue({
    el: "#app",
    data: {
      language: []
    }
  })
</script>
```

- 多个`CheckBox`对应一个 model 时, model 的类型是一个数组, 单个 checkbox 值默认是 boolean 类型
- radio 对应的值是 input 的 value 值
- `text` 和 `textarea` 默认对应的 model 是字符串
- `select` 单选对应字符串, 多选对应也是数组

效果:



4、v-on

1、基本用法

v-on 指令用于给页面元素绑定事件。

语法: `v-on:事件名="js 片段或函数名"`

示例:

```
<div id="app">
  <!--事件中直接写 js 片段-->
  <button v-on:click="num++">点赞</button>
  <!--事件指定一个回调函数, 必须是 Vue 实例中定义的函数-->
  <button v-on:click="decrement">取消</button>
  <h1>有{{num}}个赞</h1>
</div>

<script src="../node_modules/vue/dist/vue.js"></script>
<script type="text/javascript">
  let vm = new Vue({
    el: "#app",
    data: {
      num: 100
    },
    methods: {
```



```
        decrement() {
            this.num--; //要使用 data 中的属性，必须 this.属性名
        }
    }
})
</script>
```

另外，事件绑定可以简写，例如`v-on:click='add'`可以简写为`@click='add'`

2、事件修饰符

在事件处理程序中调用`event.preventDefault()`或`event.stopPropagation()`是非常常见的需求。尽管我们可以在方法中轻松实现这点，但更好的方式是：方法只有纯粹的数据逻辑，而不是去处理 DOM 事件细节。

为了解决这个问题，Vue.js 为`v-on`提供了事件修饰符。修饰符是由点开头的指令后缀来表示的。

- `.stop`：阻止事件冒泡到父元素
- `.prevent`：阻止默认事件发生
- `.capture`：使用事件捕获模式
- `.self`：只有元素自身触发事件才执行。（冒泡或捕获的都不执行）
- `.once`：只执行一次

```
<div id="app">
  <!--右击事件，并阻止默认事件发生-->
  <button v-on:contextmenu.prevent="num++">点赞</button>
  <br />
  <!--右击事件，不阻止默认事件发生-->
  <button v-on:contextmenu="decrement($event)">取消</button>
  <br />
  <h1>有{{num}}个赞</h1>
</div>
<script src="../../node_modules/vue/dist/vue.js"></script>
<script type="text/javascript">
  let app = new Vue({
    el: "#app",
    data: {
      num: 100
    },
    methods: {
      decrement(ev) {
        // ev.preventDefault();
        this.num--;
      }
    }
  });
```

```

    }
  }
})
</script>

```

效果：右键“点赞”，不会触发默认的浏览器右击事件；右键“取消”，会触发默认的浏览器右击事件）

3、按键修饰符

在监听键盘事件时，我们经常需要检查常见的键值。Vue 允许为 `v-on` 在监听键盘事件时添加按键修饰符：

```

<!-- 只有在 `keyCode` 是 13 时调用 `vm.submit()` -->
<input v-on:keyup.13="submit">

```

记住所有的 `keyCode` 比较困难，所以 Vue 为最常用的按键提供了别名：

```

<!-- 同上 -->
<input v-on:keyup.enter="submit">
<!-- 缩写语法 -->
<input @keyup.enter="submit">

```

全部的按键别名：

- `enter`
- `tab`
- `delete` (捕获“删除”和“退格”键)
- `esc`
- `space`
- `up`
- `down`
- `left`
- `right`

4、组合按钮

可以用如下修饰符来实现仅在按下相应按键时才触发鼠标或键盘事件的监听器。

- `ctrl`
- `alt`
- `shift`

```

<!-- Alt + C -->
<input @keyup.alt.67="clear">
<!-- Ctrl + Click -->

```

```
<div @click.ctrl="doSomething">Do something</div>
```

5、v-for

遍历数据渲染页面是非常常用的需求，Vue 中通过 v-for 指令来实现。

1、遍历数组

语法: **v-for="item in items"**

- items: 要遍历的数组，需要在 vue 的 data 中定义好。
- item: 迭代得到的当前正在遍历的元素

示例:

```
<div id="app">
  <ul>
    <li v-for="user in users">
      {{user.name}} - {{user.gender}} - {{user.age}}
    </li>
  </ul>
</div>
<script src="../../node_modules/vue/dist/vue.js"></script>
<script type="text/javascript">
  let app = new Vue({
    el: "#app",
    data: {
      users: [
        { name: '柳岩', gender: '女', age: 21 },
        { name: '张三', gender: '男', age: 18 },
        { name: '范冰冰', gender: '女', age: 24 },
        { name: '刘亦菲', gender: '女', age: 18 },
        { name: '古力娜扎', gender: '女', age: 25 }
      ]
    },
  })
</script>
```

效果:

- 柳岩 - 女 - 21
- 张三 - 男 - 18
- 范冰冰 - 女 - 24
- 刘亦菲 - 女 - 18
- 古力娜扎 - 女 - 25

2、数组角标

在遍历的过程中，如果我们需要知道数组角标，可以指定第二个参数：

语法： **v-for="(item,index) in items"**

- items：要迭代的数组
- item：迭代得到的数组元素别名
- index：迭代到的当前元素索引，从 0 开始。

示例：

```
<div id="app">
  <ul>
    <li v-for="(user, index) in users">
      {{index + 1}}. {{user.name}} - {{user.gender}} - {{user.age}}
    </li>
  </ul>
</div>
```

效果：

- 1. 柳岩 - 女 - 21
- 2. 张三 - 男 - 18
- 3. 范冰冰 - 女 - 24
- 4. 刘亦菲 - 女 - 18
- 5. 古力娜扎 - 女 - 25

3、遍历对象

v-for 除了可以迭代数组，也可以迭代对象。语法基本类似

语法：

v-for="value in object"

v-for="(value,key) in object"

v-for="(value,key,index) in object"

- 1 个参数时，得到的是对象的属性值
- 2 个参数时，第一个是属性值，第二个是属性名
- 3 个参数时，第三个是索引，从 0 开始

示例：

```
<div id="app">
  <ul>
    <li v-for="(value, key, index) in user">
      {{index + 1}}. {{key}} - {{value}}
    </li>
  </ul>
</div>
<script src="../../node_modules/vue/dist/vue.js"></script>
<script type="text/javascript">
  let vm = new Vue({
    el: "#app",
    data: {
      user: { name: '张三', gender: '男', age: 18 }
    }
  })
</script>
```

效果：

- 1. name - 张三
- 2. gender - 男
- 3. age - 18

4、Key

用来标识每一个元素的唯一特征，这样 Vue 可以使用“就地复用”策略有效的提高渲染的效率。

示例：

```
<ul>
  <li v-for="(item,index) in items" :key="index"></li>
</ul>

<ul>
  <li v-for="item in items" :key="item.id"></li>
</ul>
```

最佳实践：

如果 items 是数组，可以使用 index 作为每个元素的唯一标识
如果 items 是对象数组，可以使用 item.id 作为每个元素的唯一标识

6、v-if 和 v-show

1、基本用法

v-if，顾名思义，条件判断。当得到结果为 true 时，所在的元素才会被渲染。

v-show，当得到结果为 true 时，所在的元素才会被显示。

语法：**v-if="布尔表达式"**，**v-show="布尔表达式"**，

示例：

```
<div id="app">
  <button v-on:click="show = !show">点我呀</button>
  <br>
  <h1 v-if="show">
    看到我啦?!
  </h1>
  <h1 v-show="show">
    看到我啦?! show
  </h1>
</div>
<script src="../../node_modules/vue/dist/vue.js"></script>
<script type="text/javascript">
  let app = new Vue({
    el: "#app",
    data: {
      show: true
    }
  })
</script>
```

2、与 v-for 结合

当 v-if 和 v-for 出现在一起时，v-for 优先级更高。也就是说，会先遍历，再判断条件。

修改 v-for 中的案例，添加 v-if：

```
<ul>
  <li v-for="(user, index) in users" v-if="user.gender == '女'">
```

```
    {{index + 1}}. {{user.name}} - {{user.gender}} - {{user.age}}  
  </li>  
</ul>
```

效果：只显示女性

- 1. 柳岩 - 女 - 21
- 3. 范冰冰 - 女 - 24
- 4. 刘亦菲 - 女 - 18
- 5. 古力娜扎 - 女 - 25

7、v-else 和 v-else-if

v-else 元素必须紧跟在带 `v-if` 或者 `v-else-if` 的元素的后边，否则它将不会被识别。

示例：

```
<div id="app">  
  <button v-on:click="random=Math.random()">点我呀  
</button><span>{{random}}</span>  
  <h1 v-if="random >= 0.75">  
    看到我啦? ! v-if >= 0.75  
  </h1>  
  <h1 v-else-if="random > 0.5">  
    看到我啦? ! v-else-if > 0.5  
  </h1>  
  <h1 v-else-if="random > 0.25">  
    看到我啦? ! v-else-if > 0.25  
  </h1>  
  <h1 v-else>  
    看到我啦? ! v-else  
  </h1>  
</div>  
<script src="../../node_modules/vue/dist/vue.js"></script>  
<script type="text/javascript">  
  let app = new Vue({  
    el: "#app",  
    data: {  
      random: 1  
    }  
  })  
</script>
```

6、计算属性和侦听器

1、计算属性（computed）

某些结果是基于之前数据实时计算出来的，我们可以利用计算属性。来完成示例：

```
<div id="app">
  <ul>
    <li>西游记： 价格{{xyjPrice}}, 数量:
<input type="number" v-model="xyjNum"></li>
    <li>水浒传： 价格{{shzPrice}}, 数量:
<input type="number" v-model="shzNum"></li>
    <li>总价: {{totalPrice}}</li>
  </ul>
</div>
<script src="../../node_modules/vue/dist/vue.js"></script>
<script type="text/javascript">
  let app = new Vue({
    el: "#app",
    data: {
      xyjPrice: 56.73,
      shzPrice: 47.98,
      xyjNum: 1,
      shzNum: 1
    },
    computed: {
      totalPrice(){
        return this.xyjPrice*this.xyjNum + this.shzPrice*th
is.shzNum;
      }
    },
  })
</script>
```

效果：只要依赖的属性发生变化，就会重新计算这个属性

- 西游记： 价格56.73， 数量：
- 水浒传： 价格47.98， 数量：
- 总价： 370.86 动态计算

2、侦听（watch）

watch 可以让我们监控一个值的变化。从而做出相应的反应。

示例：

```
<div id="app">
  <ul>
    <li>西游记：价格{{xyjPrice}}, 数量:
<input type="number" v-model="xyjNum"></li>
    <li>水浒传：价格{{shzPrice}}, 数量:
<input type="number" v-model="shzNum"></li>
    <li>总价: {{totalPrice}}</li>
    {{msg}}
  </ul>
</div>
<script src="../../node_modules/vue/dist/vue.js"></script>
<script type="text/javascript">
  let app = new Vue({
    el: "#app",
    data: {
      xyjPrice: 56.73,
      shzPrice: 47.98,
      xyjNum: 1,
      shzNum: 1,
      msg: ""
    },
    computed: {
      totalPrice(){
        return this.xyjPrice*this.xyjNum + this.shzPrice*th
is.shzNum;
      }
    },
    watch: {
      xyjNum(newVal, oldVal){
        if(newVal >= 3){
          this.msg = "西游记没有更多库存了";
          this.xyjNum = 3;
        }else{
          this.msg = "";
        }
      }
    }
  })
</script>
```

效果：

- 西游记：价格56.73，数量：
 - 水浒传：价格47.98，数量：
 - 总价：218.17
- 西游记没有更多库存了 数量不能超过3

3、过滤器（filters）

过滤器不改变真正的`data`，而只是改变渲染的结果，并返回过滤后的版本。在很多不同的情况下，过滤器都是有用的，比如尽可能保持 API 响应的干净，并在前端处理数据的格式。

示例：展示用户列表性别显示男女

```
<body>
  <div id="app">
    <table>
      <tr v-for="user in userList">
        <td>{{user.id}}</td>
        <td>{{user.name}}</td>
        <!-- 使用代码块实现，有代码侵入 -->
        <td>{{user.gender===1? "男":"女"}}</td>
      </tr>
    </table>
  </div>
</body>
<script src="../../node_modules/vue/dist/vue.js"></script>
<script>

  let app = new Vue({
    el: "#app",
    data: {
      userList: [
        { id: 1, name: 'jacky', gender: 1 },
        { id: 2, name: 'peter', gender: 0 }
      ]
    }
  });

</script>
```

1、局部过滤器

注册在当前 vue 实例中，只有当前实例能用

```
let app = new Vue({
  el: "#app",
  data: {
    userList: [
      { id: 1, name: 'jacky', gender: 1 },
      { id: 2, name: 'peter', gender: 0 }
    ]
  },
  // filters 定义局部过滤器，只可以在当前 vue 实例中使用
  filters: {
    genderFilter(gender) {
      return gender === 1 ? '男~' : '女~'
    }
  }
});
```

<!-- | 管道符号：表示使用后面的过滤器处理前面的数据 -->

```
<td>{{user.gender | genderFilter}}</td>
```

2、全局过滤器

```
// 在创建 Vue 实例之前全局定义过滤器：
Vue.filter('capitalize', function (value) {
  return value.charAt(0).toUpperCase() + value.slice(1)
})
```

任何 vue 实例都可以使用： <td>{{user.name | capitalize}}</td>

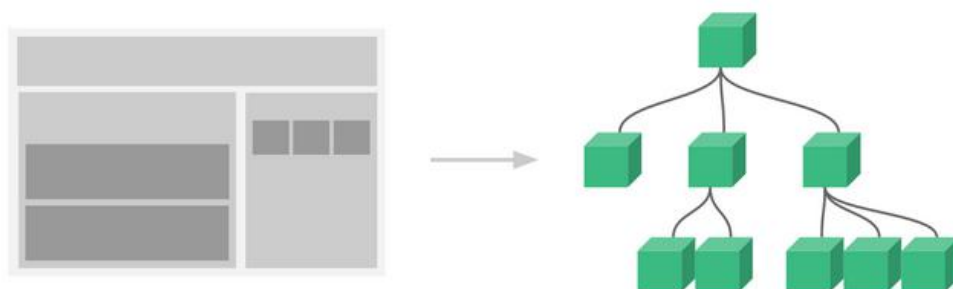
过滤器常用来处理文本格式化的操作。过滤器可以用在两个地方：[双花括号插值](#)和 [v-bind 表达式](#)

7、组件化

在大型应用开发的时候，页面可以划分成很多部分。往往不同的页面，也会有相同的部分。例如可能会有相同的头部导航。

但是如果每个页面都独自开发，这无疑增加了我们开发的成本。所以我们会把页面的不同部分拆分成独立的组件，然后在不同页面就可以共享这些组件，避免重复开发。

在 vue 里，所有的 vue 实例都是组件



例如，你可能会有页头、侧边栏、内容区等组件，每个组件又包含了其它的像导航链接、博文之类的组件。

1、全局组件

我们通过 Vue 的 `component` 方法来定义一个全局组件。

```
<div id="app">
  <!--使用定义好的全局组件-->
  <counter></counter>
</div>
<script src="../../node_modules/vue/dist/vue.js"></script>
<script type="text/javascript">
  // 定义全局组件，两个参数：1，组件名称。2，组件参数
  Vue.component("counter", {
    template: '<button v-on:click="count++">你点了我 {{ count }} 次，我记住了.</button>',
    data() {
      return {
        count: 0
      }
    }
  })
```

```
let app = new Vue({
  el: "#app"
})
</script>
```

- 组件其实也是一个 Vue 实例，因此它在定义时也会接收：data、methods、生命周期函数等
- 不同的是组件不会与页面的元素绑定，否则就无法复用了，因此没有 el 属性。
- 但是组件渲染需要 html 模板，所以增加了 template 属性，值就是 HTML 模板
- 全局组件定义完毕，任何 vue 实例都可以直接在 HTML 中通过组件名称来使用组件了
- data 必须是一个函数，不再是一个对象。

你点了我 7 次，我记住了。

2、组件的复用

定义好的组件，可以任意复用多次：

```
<div id="app">
  <!--使用定义好的全局组件-->
  <counter></counter>
  <counter></counter>
  <counter></counter>
</div>
```

组件的 data 属性必须是函数！

一个组件的 data 选项必须是一个函数，因此每个实例可以维护一份被返回对象的独立的拷贝；

否则：

<https://cn.vuejs.org/v2/guide/components.html#data-%E5%BF%85%E9%A1%BB%E6%98%AF%E4%B8%80%E4%B8%AA%E5%87%BD%E6%95%B0>

3、局部组件

一旦全局注册，就意味着即便以后你不再使用这个组件，它依然会随着 Vue 的加载而加载。因此，对于一些并不频繁使用的组件，我们会采用局部注册。

我们先在外部定义一个对象，结构与创建组件时传递的第二个参数一致：

```
const counter = {  
  template: '<button v-on:click="count++">你点了  
我 {{ count }} 次，我记住了.</button>',  
  data() {  
    return {  
      count: 0  
    }  
  }  
};
```

然后在 Vue 中使用它：

```
let app = new Vue({  
  el: "#app",  
  components: {  
    counter: counter // 将定义的对象注册为组件  
  }  
})
```

- components 就是当前 vue 对象子组件集合。
 - 其 key 就是子组件名称
 - 其值就是组件对象名
- 效果与刚才的全局注册是类似的，不同的是，这个 counter 组件只能在当前的 Vue 实例中使用

简写：

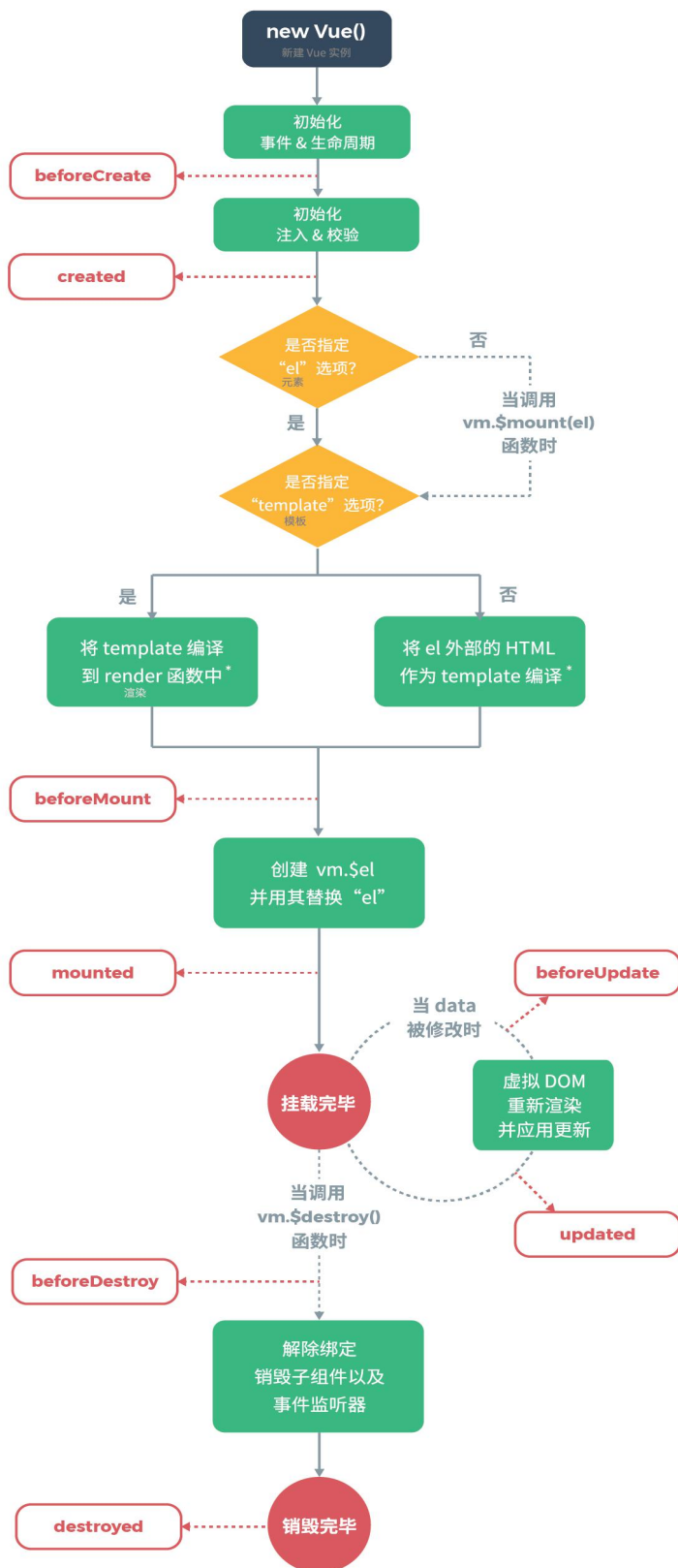
```
let app = new Vue({  
  el: "#app",  
  components: {  
    counter // 将定义的对象注册为组件  
  }  
})
```

8、生命周期钩子函数

1、生命周期

每个 Vue 实例在被创建时都要经过一系列的初始化过程：创建实例，装载模板，渲染模板等等。Vue 为生命周期中的每个状态都设置了钩子函数（监听函数）。每当 Vue 实例处于不同的生命周期时，对应的函数就会被触发调用。

生命周期：你不需要立马弄明白所有的东西。



* 如果使用构造生成文件（例如构造单文件组件），
模板编译将提前执行

2、钩子函数

- **beforeCreated:** 我们在用 Vue 时都要进行实例化，因此，该函数就是在 Vue 实例化时调用，也可以将他理解为初始化函数比较方便一点，在 Vue1.0 时，这个函数的名字就是 init。
- **created:** 在创建实例之后进行调用。
- **beforeMount:** 页面加载完成，没有渲染。如：此时页面还是{{name}}
- **mounted:** 我们可以将他理解为原生 js 中的 `window.onload=function({,..})`，或许大家也在用 jquery，所以也可以理解为 jquery 中的 `$(document).ready(function(){....})`，他的功能就是在 dom 文档渲染完毕之后将要执行的函数，该函数在 Vue1.0 版本中名字为 `compiled`。此时页面中的{{name}}已被渲染成张三
- **beforeDestroy:** 该函数将在销毁实例前进行调用。
- **destroyed:** 改函数将在销毁实例时进行调用。
- **beforeUpdate:** 组件更新之前。
- **updated:** 组件更新之后。

示例

```
<body>
  <div id="app">
    <span id="num">{{num}}</span>
    <button v-on:click="num++">赞! </button>
    <h2>
      {{name}}, 非常帅!!! 有{{num}}个人点赞。
    </h2>
  </div>
</body>
<script src="../node_modules/vue/dist/vue.js"></script>
<script>
  let app = new Vue({
    el: "#app",
    data: {
      name: "张三",
      num: 100
    },
    methods: {
      show() {
        return this.name;
      },
      add() {
        this.num++;
      }
    },
    beforeCreate() {
```



```
        console.log("=====beforeCreate=====");
        console.log("数据模型未加载: " + this.name, this.num);
        console.log("方法未加载: " + this.show());
        console.log("html 模板未加载:
" + document.getElementById("num"));
    },
    created: function () {
        console.log("=====created=====");
        console.log("数据模型已加载: " + this.name, this.num);
        console.log("方法已加载: " + this.show());
        console.log("html 模板已加载:
" + document.getElementById("num"));
        console.log("html 模板未渲染:
" + document.getElementById("num").innerText);
    },
    beforeMount() {
        console.log("=====beforeMount=====");
        console.log("html 模板未渲染:
" + document.getElementById("num").innerText);
    },
    mounted() {
        console.log("=====mounted=====");
        console.log("html 模板已渲染:
" + document.getElementById("num").innerText);
    },
    beforeUpdate() {
        console.log("=====beforeUpdate=====");
        console.log("数据模型已更新: " + this.num);
        console.log("html 模板未更新:
" + document.getElementById("num").innerText);
    },
    updated() {
        console.log("=====updated=====");
        console.log("数据模型已更新: " + this.num);
        console.log("html 模板已更新:
" + document.getElementById("num").innerText);
    }
});

</script>
```

9、vue 模块化开发

1、npm install webpack -g

全局安装 webpack

2、npm install -g @vue/cli-init

全局安装 vue 脚手架

3、初始化 vue 项目；

vue init **webpack** **appname**: vue 脚手架使用 webpack 模板初始化一个 appname 项目

4、启动 vue 项目；

项目的 package.json 中有 scripts，代表我们能运行的命令

npm start = npm run dev: 启动项目

npm run build: 将项目打包

5、模块化开发

1、项目结构

目录/文件	说明
build	项目构建(webpack)相关代码
config	配置目录，包括端口号等。我们初学可以使用默认的。
node_modules	npm 加载的项目依赖模块
src	<p>这里是我们要开发的目录，基本上要做的事情都在这个目录里。里面包含了几个目录及文件：</p> <ul style="list-style-type: none"> assets: 放置一些图片，如logo等。 components: 目录里面放了一个组件文件，可以不用。 App.vue: 项目入口文件，我们也可以直接将组件写这里，而不使用 components 目录。 main.js: 项目的核心文件。
static	静态资源目录，如图片、字体等。
test	初始测试目录，可删除
.xxxx文件	这些是一些配置文件，包括语法配置，git配置等。
index.html	首页入口文件，你可以添加一些 meta 信息或统计代码啥的。
package.json	项目配置文件。
README.md	项目的说明文档，markdown 格式



- 运行流程
 - 进入页面首先加载 index.html 和 main.js 文件。
 - main.js 导入了一些模块【vue、app、router】，并且创建 vue 实例，关联 index.html 页面的<div id="app">元素。使用了 router，导入了 App 组件。并且使用<App/>标签引用了这个组件
 - 第一次默认显示 App 组件。App 组件有个图片和<router-view>，所以显示了图片。但是由于<router-view>代表路由的视图，默认是访问/#/路径（router 路径默认使用 HASH 模式）。在 router 中配置的/是显示 HelloWorld 组件。
 - 所以第一次访问，显示图片和 HelloWorld 组件。
 - 我们尝试自己写一个组件，并且加入路由。点击跳转。需要使用<router-link to="/foo">Go to Foo</router-link>标签

2、Vue 单文件组件

Vue 单文件组件模板有三个部分：

<pre><template> <div class="hello"> <h1>{{ msg }}</h1> </div> </template> <script> export default { name: 'HelloWorld', data () { return { msg: 'Welcome to Your Vue.js App' } } } </script> <!-- Add "scoped" attribute to limit CSS to this component only --> <style scoped> h1, h2 { font-weight: normal; } </style></pre>
Template: 编写模板
Script: vue 实例配置
Style: 样式

3、vscode 添加用户代码片段（快速生成 vue 模板）

文件-->首选项-->用户代码片段-->点击新建代码片段--取名 vue.json 确定
<pre>{ "生成 vue 模板": { "prefix": "vue", "body": ["<template>", "<div></div>", "</template>",</pre>

```
    "",
    "<script>",
    "//这里可以导入其他文件（比如：组件，工具 js，第三方插件 js，json
文件，图片文件等等）",
    "//例如: import 《组件名称》 from '《组件路径》';",
    "",
    "export default {",
    "//import 引入的组件需要注入到对象中才能使用",
    "components: {},",
    "props: {},",
    "data() {",
    "//这里存放数据",
    "return {",
    "",
    "};",
    "},",
    "//计算属性 类似于 data 概念",
    "computed: {},",
    "//监控 data 中的数据变化",
    "watch: {},",
    "//方法集合",
    "methods: {",
    "",
    "},",
    "//生命周期 - 创建完成（可以访问当前 this 实例）",
    "created() {",
    "",
    "},",
    "//生命周期 - 挂载完成（可以访问 DOM 元素）",
    "mounted() {",
    "",
    "},",
    "beforeCreate() {}, //生命周期 - 创建之前",
    "beforeMount() {}, //生命周期 - 挂载之前",
    "beforeUpdate() {}, //生命周期 - 更新之前",
    "updated() {}, //生命周期 - 更新之后",
    "beforeDestroy() {}, //生命周期 - 销毁之前",
    "destroyed() {}, //生命周期 - 销毁完成",
    "activated() {}, //如果页面有 keep-alive 缓存功能,这个函数会触发
",
    "}",
    "</script>",
    "<style lang='scss' scoped>",
    "//@import url($3); 引入公共 css 类",
```

```
        "$4",  
        "</style>"  
    ],  
    "description": "生成 vue 模板"  
  }  
}
```

4、导入 element-ui 快速开发

- 1、安装 element-ui: `npm i element-ui`
- 2、在 main.js 中引入 element-ui 就可以全局使用了。
`import ElementUI from 'element-ui'`
`import 'element-ui/lib/theme-chalk/index.css'`

`Vue.use(ElementUI)`

- 3、将 App.vue 改为 element-ui 中的后台布局
- 4、添加测试路由、组件，测试跳转逻辑
 - (1)、参照文档 el-menu 添加 router 属性
 - (2)、参照文档 el-menu-item 指定 index 需要跳转的地址

五、Babel

Babel 是一个 JavaScript 编译器，我们可以使用 es 的最新语法编程，而不用担心浏览器兼容问题。他会自动转化为浏览器兼容的代码

六、Webpack

自动化项目构建工具。gulp 也是同类产品