Ｗeb前端开发职业技能标准

串讲教案

（高级）

工业和信息化部教育与考试中心

二Ｏ一九年制

第一篇 性能优化与自动化技术

[第一章 HTML与CSS代码优化 1](#_Toc13908318)

[第一节 前端命名规范 1](#_Toc13908319)

[第二节 HTML代码优化 2](#_Toc13908320)

[第三节 CSS代码优化 3](#_Toc13908321)

[第二章 前端资源优化 5](#_Toc13908322)

[第一节 Sprite 拼合图 5](#_Toc13908323)

[第二节 代码压缩 打包工具 7](#_Toc13908324)

[第三节 预加载 7](#_Toc13908325)

[第四节 懒加载 8](#_Toc13908326)

[第三章 JavaScript代码优化 9](#_Toc13908327)

[第一节 JavaScript代码可维护性 9](#_Toc13908328)

[第二节 JavaScript 代码可扩展性 9](#_Toc13908329)

[第三节 JavaScript代码可调试性 9](#_Toc13908330)

[第四节 JavaScript DOM优化 10](#_Toc13908331)

[第四章 webpack 入门 12](#_Toc13908332)

[第一节 简介 12](#_Toc13908333)

[第二节 安装 13](#_Toc13908334)

[第三节 基本应用 13](#_Toc13908335)

[第五章 配置文件入门 16](#_Toc13908336)

[第一节 核心概念 16](#_Toc13908337)

[第二节 简单的配置文件 17](#_Toc13908338)

[第六章 配置详解 18](#_Toc13908339)

[第一节 entry 和 output 18](#_Toc13908340)

[第二节 module 18](#_Toc13908341)

[第三节 plugins 20](#_Toc13908342)

[第四节 resolve 21](#_Toc13908343)

[第五节 其他配置项 21](#_Toc13908344)

[第七章 webpack-dev-server 22](#_Toc13908345)

[第一节 安装 22](#_Toc13908346)

[第二节 安装配置 22](#_Toc13908347)

[第三节 完整案例 23](#_Toc13908348)

[第四节 运行 24](#_Toc13908349)

[第八章 ES6基础 25](#_Toc13908350)

[第一节 概述 25](#_Toc13908351)

[第二节 symbol数据类型 25](#_Toc13908352)

[第三节 let与const 27](#_Toc13908353)

[第四节 变量的解构赋值 27](#_Toc13908354)

[第五节 Set与Map 28](#_Toc13908355)

[第六节 箭头函数 30](#_Toc13908356)

[第七节 ES6相对于ES5扩展 31](#_Toc13908357)

[第八节 ES6高级操作 33](#_Toc13908358)

第二篇 前端高效开发框架技术与应用

[第一章 vue基础 40](#_Toc13908359)

[第一节 框架简介 40](#_Toc13908360)

[第二节 基础语法 43](#_Toc13908361)

[第三节 渲染 51](#_Toc13908362)

[第二章 Vue组件 58](#_Toc13908363)

[第一节 组件概述 58](#_Toc13908364)

[第二节 事件通信 60](#_Toc13908365)

[第三节 组件类型 64](#_Toc13908366)

[第三章 Vue工程化工具 68](#_Toc13908367)

[第一节 搭建Node.js环境 68](#_Toc13908368)

[第二节 使用Vue CLI 74](#_Toc13908369)

[第三节 Vue项目配置 76](#_Toc13908370)

[第四节 js模块 78](#_Toc13908371)

[第五节 vue-devtools工作 80](#_Toc13908372)

[第四章 Express服务器开发 81](#_Toc13908373)

[第一节 概述 81](#_Toc13908374)

[第二节 Express 82](#_Toc13908375)

[第三节 请求响应对象 90](#_Toc13908376)

[第四节 Session使用 94](#_Toc13908377)

[第五节 Express中使用RESTful API 95](#_Toc13908378)

[第六节 规划目录结构 98](#_Toc13908379)

[第七节 Nodemon热刷新 98](#_Toc13908380)

[第五章 axios 网络交互 100](#_Toc13908381)

[第一节 axios库的概述 100](#_Toc13908382)

[第二节 axios库的使用 100](#_Toc13908383)

[第三节 发送get请求 103](#_Toc13908384)

[第四节 发送post请求 104](#_Toc13908385)

[第五节 Vue中使用axios 104](#_Toc13908386)

[第六节 axios跨域问题 104](#_Toc13908387)

[第六章 Vue-router 106](#_Toc13908388)

[第一节 SPA和Vue路由 106](#_Toc13908389)

[第二节 路由使用 106](#_Toc13908390)

[第三节 路由导航 108](#_Toc13908391)

[第四节 嵌套路由和命名路由视图 110](#_Toc13908392)

[第七章 Vuex 状态管理 112](#_Toc13908393)

[第一节 状态管理 112](#_Toc13908394)

[第二节 Vuex Store对象 113](#_Toc13908395)

[第三节 Vuex多模块 116](#_Toc13908396)

[第四节 Vuex插件使用 117](#_Toc13908397)

[第八章 Vue ui 118](#_Toc13908398)

[第一节 Vue ui库 118](#_Toc13908399)

[第二节 Element组件 119](#_Toc13908400)

[第三节 第三方扩展库Express 119](#_Toc13908401)

第三篇 移动web设计与开发

[第一章 多媒体与绘图 121](#_Toc13908402)

[第一节 HTML5多媒体支持 121](#_Toc13908403)

[第二节 HTMLAudioElement和HTMLVideoElement 122](#_Toc13908404)

[第三节 <audio>和<video>的事件 125](#_Toc13908405)

[第四节 HTML5的绘图支持 126](#_Toc13908406)

[第五节 SVG 130](#_Toc13908407)

[第二章 Web Storage 133](#_Toc13908408)

[第三章 Less的使用 134](#_Toc13908409)

[第一节 Less介绍 134](#_Toc13908410)

[第二节 Less的安装 134](#_Toc13908411)

[第三节 Less的使用 134](#_Toc13908412)

[第四章 jQuery Mobile基本 138](#_Toc13908413)

[第一节 jQuery Mobile的安装 138](#_Toc13908414)

[第二节 jQuery Mobile的使用 138](#_Toc13908415)

[第三节 jQuery Mobile主题 147](#_Toc13908416)

[第四节 jQuery Mobile事件 150](#_Toc13908417)

**第一篇 性能优化与自动化技术**

# HTML与CSS代码优化

## 第一节 前端命名规范

### 前端结构组织与文件命名规范

前端结构组织具有如下原则：

* 同一项目中代码的组织结构要清晰
* 同类型文件归类到相同的文件夹中
* 文件命名规则须统一并且命名要有意义

### HTML命名规范

* HTML代码所有的标签名和属性应该都为小写
* 属性值应该用引号括起来
* 元素的id与class按照特定规范命名
* 代码缩进4个空格
* 给HTML代码块添加必要注释

### CSS命名规范

* 尽量使用class选择器进行样式设定
* 类命名时取父元素的class名作为前缀，使用-连接
* 类名与样式之间以空格进行分割
* 添加CSS代码注释

### JavaScript命名规范

* 变量名区分大小写，第一个字符不允许是数字，不允许包含空格和其他标点符号
* 尽量使用有实际意义的命名
* 禁止使用JavaScript关键词、保留字全名
* 添加JavaScript代码注释

## 第二节 HTML代码优化

优化HTML代码的目的一方面是使网站对搜索引擎更友好，一个漂亮的前端网站是用户友好的，并且在各方面都进行了优化的网站是搜索引擎友好的，是理想的网站。另一方面也是对代码的维护提供便利。那么怎样的HTML代码是符合优化的规范写法呢？一个规范的网页的HTML代码应该尽可能满足如下的条件：

1. 正确闭合HTML标签，如 <div>盒子</div> 。
2. HTML代码层级间合理缩进，统一用两个或者四个空格缩进。
3. 属性值需要使用双引号，如<div id="mydiv">盒子</div>
4. 结构与样式进行有效的分离，即HTML和CSS文件的分离。
5. 结构与行为进行有效的分离，即HTML和JS文件的分离。
6. 使用语义化标签，如头部标签<header>。
7. 删除多余容器元素，代码层次少。
8. 避免使用table进行页面的布局，换成用DIV+CSS。

此外，还可以通过在线网站（http://validator.w3.org/）来对HTML代码进行格式化验证。

## 第三节 CSS代码优化

### 浏览器样式重置

在各个核心浏览器中，相同元素的解析结果不同，那么就需要手动重置一些样式，重置样式一般会放在公共CSS文件的头部，重置样式主要做以下操作：

1. 去除标签的默认样式，如p、li、input等。
2. HTML5新标签设置为display:block。
3. 重置一些元素的样式如超链接、字号等样式。

### CSS样式选择器、优先级

CSS样式选择器优先级高的会覆盖优先级低的选择器。以下是三类选择器的权重值：

1. ID选择器的权重值为100。
2. 类选择器的权重值为10。
3. 标签选择器的权重值为1。

此外，在这里还需要有几点注意事项需要提下：

1. CSS样式中尽量不使用ID选择器，会降低代码的复用性。
2. CSS样式中尽量不使用!important，会降低代码的复用性。
3. 尽量减少子选择器的层级。

### CSS样式多余样式去除、结构优化

常用的几点举措：

1. 定义简洁的CSS样式规则
2. 合并相关CSS样式规则，如margin-top、margin-right、margin-bottom和margin-left可以合并成margin、padding同理、background-repeat、background-size等可以合并成background。
3. 定义简洁的属性值，如颜色的十六进制#ffffff可以简写成#fff、字体的0.8em可以简写成.8em、0px可以简写成0;
4. 合并相同的，删除无效的。如p1、p2、p3都有background:red这个属性，那么可以给p1、p2和p3都加上p-red的class，然后给p-red赋值background:red的属性，就不需要给p1、p2、p3单独加了。

# 第二章 前端资源优化

## 第一节 Sprite 拼合图

CSS Sprite，中文也叫CSS精灵、雪碧图，是一种将零散的背景图合并成一张大图，再利用CSS的background-position属性进行背景的定位从而达到减少图片请求数量达到加快加载速度的网页应用处理方式。

### background属性

|  |  |  |
| --- | --- | --- |
| **值** | **描述** | **使用频率** |
| background-color | 规定要使用的背景颜色。 | 高 |
| background-position | 规定背景图像的位置。 | 高 |
| background-size | 规定背景图片的尺寸。 | 高 |
| background-repeat | 规定如何重复背景图像。 | 高 |
| background-origin | 规定背景图片的定位区域。 | 低 |
| background-clip | 规定背景的绘制区域。 | 低 |
| background-attachment | 规定背景图像是否固定或者随着页面的其余部分滚动。 | 中 |
| background-image | 规定要使用的背景图像 | 高 |

### background-position属性

|  |  |
| --- | --- |
| **值** | **描述** |
| top left top center top right center left center center center right bottom left bottom center bottom right | top代表垂直位置为最头部，bottom代表垂直位置为最底部，left代表水平位置为最左边，center代表水平位置为居中，right代表水平位置为最右边。 如果您仅规定了一个关键词，那么第二个值将是"center"。 默认值：0% 0%。 |
| x% y% | 第一个值是水平位置，第二个值是垂直位置。 左上角是 0% 0%。右下角是 100% 100%。 如果您仅规定了一个值，另一个值将是 50%。 |
| xpos ypos | 第一个值是水平位置，第二个值是垂直位置。 左上角是 0 0。单位是像素 (0px 0px) 或任何其他的 CSS 单位。 如果您仅规定了一个值，另一个值将是50%。 您可以混合使用 % 和 position 值。 |

### CSS Sprite的制作工具

CSS Sprite有三种制作方式，可以通过CSS Sprite制作工具、photoshop和打包工具（如webpack）来制作。

打包工具：现今流行的好多前端打包工具都支持CSS sprite的集成，如webpack中只需安装webpack-spritesmith依赖，然后在配置文件中引入依赖var SpritesmithPlugin = require('webpack-spritesmith')，最后在配置文件webpack.config.js添加如下代码:

model.export={  
 entry:'./src/main.js',//值可以是字符串、数组或对象  
 output:{  
 path:path.resolve(\_\_dirname,'./dist'),//webpack结果存储  
 publicPath:'/dist',//公共路径  
 filename:'build.js'  
 },  
 plugins:[  
 new SpriteswithPlugin({  
 //目标小图标  
 src:{  
 cwd:path.resolve(\_\_dirname,'./src/assets/imgs/icons'),  
 glob:'\*.png'  
 },  
 //输出CSS Sprite文件及样式文件  
 target:{  
 image:path.resolve(\_\_dirname,'./dist/sprites/sprite.png'),  
 css:path.resolve(\_\_dirname,'./dist/sprites/sprite.css')  
 },  
 //样式文件中调用CSS Sprite地址写法  
 apiOptions:{  
 cssImageRef:'../sprites/sprite.png'  
 },  
 spriteswithOptions:{  
 algorithm:'top-down'  
 }  
 })  
 ]  
}

## 第二节 代码压缩 打包工具

### 压缩JavaScript

目前最成熟的 JavaScript 代码压缩工具是[UglifyJS](https://github.com/mishoo/UglifyJS2)， 它会分析 JavaScript代码语法树，理解代码含义，从而能做到诸如去掉无效代码、去掉日志输出代码、缩短变量名等优化。

在 webpack 中接入 UglifyJS 需要通过插件的形式，UglifyJsPlugin是比较常用的插件，通过在webpack的配置文件webpack.config.js中加入以下代码即可

### 压缩CSS

CSS 代码也可以像 JavaScript 那样被压缩，以达到提升加载速度和代码混淆的作用。 目前比较成熟可靠的 CSS 压缩工具是CSSNANO，基于 PostCSS。CSSNANO由于能理解 CSS 代码的含义，故不仅仅是删掉空格，例如：margin:10px 20px 10px 20px 被压缩成 margin: 10px 20px；color: #ff0000 被压缩成 color:red。通JavaScript压缩一样，在 webpack 中接入CSSNANO去压缩代码需要引入css-loader 的 minimize 选项插件，通过在webpack的配置文件webpack.config.js中加入以下代码即可

## 第三节 预加载

预加载简单来说就是将所有所需的资源提前请求加载到本地，这样后面在需要用到时就直接从缓存取资源。

1. 使用HTML标签

* <img src="./icon886.png" style="display:none"/>

1. 使用Image对象

* var image= new Image();  
  image.src="./icon886.png";

1. 使用XMLHttpRequest对象

* xmlhttprequest.onreadystatechange=callback;  
  xmlhttprequest.onprogress=progressCallback;  
  xmlhttprequest.open("GET",http://image.baidu.com/mouse.jpg,true);  
  xmlhttprequest.send();  
  function callback(){  
   if(xmlhttprequest.readyState=4&&xmlhttprequest.status==200){  
   var responseText=xmlhttprequest.responseText;  
   }else{  
   console.log("Request was unsuccessful"+xmlhttprequest.status);  
   }  
  }  
  function progressCallback(e){  
   c=e||event;  
   if(e.lengthComputable){  
   console.log("Received"+e.loaded+"of"+e.total+"bytes");  
   }  
  }

1. 使用PreloadJS库

* var queue=new createjs.LoadQueue();  
  queue.on("complete",handleComplete,this);  
  queue.loadMainfest([  
   {id:"myImage",src:"http://pic26.nipic.com/20121213/6168183 0044449030002.jpg"},  
   {id:"myImage2",src:"http://pic9.nipic.com/20100814/2839526 1931471581702.jpg"}  
  ]);  
  function handleComplete(){  
   var image=queue.getResult("myImage");  
   document.body.appendChild(image);  
  }

## 第四节 懒加载

首屏加载，技术上显示要用的技术就是图片懒加载，即到可视区域再加载。

1. jquery.lazyload.js。该js是一个基于jquery的懒加载插件，里面封装了懒加载用到的方法及实现。
2. echo.js，一款非常简单实用轻量级的图片延时加载插件。

# 第三章 JavaScript代码优化

## 第一节 JavaScript代码可维护性

### 代码与结构分离

代码与结构分离，即把HTML和JavaScript进行有效分离，这里有两种意义上的分离，一种是就在HTML中分离JavaScript，另外一种是在JavaScript中分离HTML。

### 样式与结构的分离

样式和结构的分离，即把CSS和HTML进行有效分离，这里指在JavaScript中将CSS和HTML进行分离。

### 数据与代码分离

数据与代码分离，也可以认为是前后端分离的表现。后台接口只负责返回json格式的数据，不会返回带标签甚至是带样式或带JavaScript的组合数据。而且模拟数据可以用json文件或者相关插件如mock。这样的好处就是将数据从代码中抽离出来，当数据变化时不能影响代码。

## 第二节 JavaScript 代码可扩展性

一段好的代码应该是能灵活扩展的，随着前端和JavaScript的发展，在JavaScript中可扩展性的解决方法有AMD、CMD和ES6的模块化等。

* AMD：Asynchronous Module Definition，即异步模块加载机制。
* CMD:Common Module Definition，即通用模块定义规范。
* ES6模块化:ES6的模块化分为导出（export）与导入（import）两个模块。

## 第三节 JavaScript代码可调试性

可以通过console.log、debugger、alert和try...catch捕获异常来进行JavaScript的调试。

* console.log：即通过在JavaScript中添console.log(msg);msg为需要打印的信息，可以是变量、字符串，变量类型可以是数组、对象、数字等等。
* debugger：debugger 关键字用于停止执行 JavaScript，并调用调试函数。这个关键字与在调试工具中设置断点的效果是一样的。
* alert：和console.log一样，alert通过在JavaScript中添加alert(msg);msg为需要弹窗的信息，值得一提的是这个弹框是强制阻塞的，只有关闭该弹窗才能解除阻塞，因此需要谨慎使用。
* try...catch：用try...catch...finally来进行异常的捕获，try代码块表示可能发生异常的代码，catch表示捕获异常对象，finally无论是否发生异常都执行evalError、typeError、syntaxError、referenceError、rangeError、URLError。

## 第四节 JavaScript DOM优化

### 提升文件加载速度

* 合并JavaScript代码，尽可能少的使用script标签。
* 无堵塞加载JavaScript。通过给script标签增加 defer属性或者是 async 属性来实现<script src="file.js" defer></script>。
* 动态创建script标签来加载。

### JavaScript DOM操作优化

* DOM访问和修改。都说访问DOM耗性能，用循环访问更是如此。所以可以减少DOM访问来优化。
* HTML集合。

1. 缓存length属性，免得每次查询都要遍历HTML文档来获得length。（在没有增删节点的情况下）。
2. 利用局部变量访问HTML集合。
3. 元素节点的获取。
4. 选择器API。

* 重排和重绘。

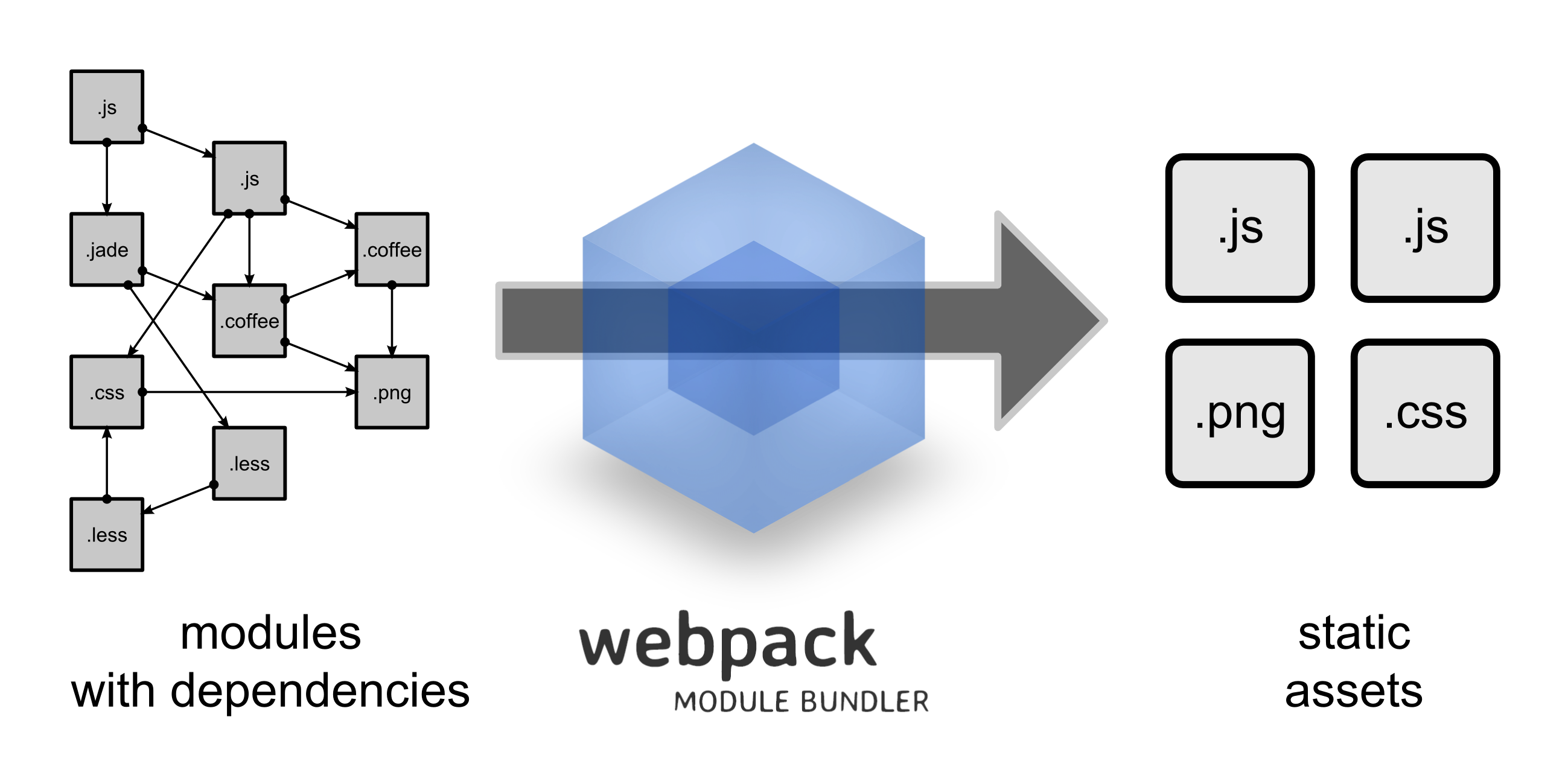
1. 用cssText改变样式。
2. 批量修改DOM。

### JavaScript DOM脚本加载优化

JavaScript通常会被称为脚本语言，加载脚本（js文件）的优化也至关重要，在一般情况下，许多人都是将<script>写在了<head>标签中，而许多浏览器都是使用单一的线程来加载JavaScript文件的，从上往下，从左往右。若是加载过程出错，那么网页就会阻塞，就像许多网站用Google的CDN库，而我们在墙内访问一样。页面不会出来，一直加载这个js文件，直到浏览器放弃加载为止。可以通过将JavaScript脚本文件放在body标签的后面和减少页面中包含script标签的数量（其实就是相当于合并请求）来优化。

# 第四章 webpack 入门

## 第一节 简介



[webpack](https://www.webpackjs.com/concepts/)是一个module bundler（模块构建工具），由于 JavaScript 应用程序的复杂性不断增加，构建工具已成为 web 开发中不可或缺的一部分。它帮助我们去打包、编译和管理项目需要的众多资源文件和依赖库。

webpack支持CommonJS、AMD和ES6模块系统，并且兼容多种JS书写规范，可以处理模块间的依赖关系，所以具有更强大的JS模块化的功能，它能压缩图片，对CSS、js文件进行语法检查、压缩、编译打包。

构建工具没有标准，现在开发者可以在 Webpack、Gulp、Bowserify、NPM scripts、Grunt 等类似的十余种构建工具。所有这些工具在在底层实现有所不同，但总体还是非常相似的，所以大多数时候你可以根据个人喜好和开发的项目，来选择合适的构建工具。

### 优点

* 可以很好的用于单页应用
* 同时支持 require() 和 import 模块语法
* 允许非常前沿的 code splitting（代码分割） 特性
* 热加载可以让 React、Vue.js 和其它类似框架的本地开发更快
* 它是目前最受欢迎的构建工具

### 缺点

* 不适合 web 开发的初学者
* 对于 CSS、图片和其它非 JS 资源文件时，需要先混淆处理
* 文档不够完善
* 变化很大，不同版本的使用方法存在较大差异

## 第二节 安装

1. 全局安装，用于全局使用命令行打包文件。

//安装全局webpack  
npm install webpack -g  
  
//安装全局webpack-cli  
npm install webpack-cli -g

1. 新建项目文件夹
2. 进入项目文件夹，打开cmd命令行窗口，运行如下命令：

webpack -v  
4.29.5 结果显示版本号，说明webpack安装成功，可以在全局任何目录下使用

## 第三节 基本应用

### hello world

**SPA**（single page web application）单页应用程序，是webpack打包的典型应用，一个典型的SPA应用，主要由以下几个部分组成：

|  |  |
| --- | --- |
| 文件 | 说明 |
| index.html | 主文件 |
| JS文件 | 可能有多个JS文件，可通过webpack合并打包为一个文件 |
| CSS文件 | 可能有多个CSS文件，可通过webpack合并打包为一个文件 |
| 图片 | 可通过webpack压缩优化 |

1. **新建src文件夹**

* 该文件夹存放开发用的文件（通常命名为：src 、dev、app），在src目录下创建文件：

//a.js  
var run=function(){  
 console.log("aaa");  
};  
//node CommonJS模块  
//module.exports.run=run;  
  
//ES6语法  
export default {run};

//b.js  
var play=function(arg){  
 console.log(arg);  
};  
//node CommonJS模块  
//module.exports.play=play;  
  
//ES6语法  
export default {play};

//index.js  
//node CommonJS 引入js模块  
//var a=require("./a.js");  
//var b=require("./b.js");  
  
//ES6 引入js模块  
import a from "./a.js";  
import b from "./b.js";  
  
var txt = "hello world";  
a.run();  
b.play(txt);

1. **新建dist文件夹**

该文件夹存放打包后的文件，可以先不创建，打包时可以自动创建，通常命名为：dis 、dist、bulit

<!--index.html-->  
<!DOCTYPE html>   
<html>   
 <head>   
 <meta charset="UTF-8">   
 <title></title>  
 <!--该文件暂时不存在，打包后自动创建该文件-->  
 <script src="./main.js"></script>  
 </head>   
 <body>   
 <h1>hxsd</h1>  
 </body>   
</html>

目录结构：

├──dist  
│ └── index.html  
├──src  
 ├── a.js   
 ├── b.js   
 └── index.js

1. **打包**

webpack --mode development

经过打包后，已经根据三个js依赖关系，打包合并为dist/main.js。

### 常用命令参数

|  |  |
| --- | --- |
| 常用命令 | 说明 |
| webpack --output 文件名 | 输出文件路径 |
| webpack --config 文件名 | 用于指定其他配置文件（默认为 webpack.config.js） |
| webpack --mode 模式 | 打包模式 <br />production：生产模式（压缩） <br />developement：开发模式（非压缩） |
| webpack --watch | 监听文件变化并自动打包 |
| webpack -p | 压缩混淆脚本（相当于 --mode production） |
| webpack -d | 生成map映射文件 |
| webpack --progress | 显示进度，打印出编译进度的百分比值 |
| webpack --color | 用不同颜色标记不同的信息 |
| webpack --profile | 显示每一步编译的具体时间，可帮助优化构建性能 |
| webpack --hot | 热替换 |
| webpack --bail | 如果编译过程出现error，立马停止编译 |

# 

# 第五章 配置文件入门

可通过定义配置文件进行复杂操作，默认的配置文件名为：**webpack.config.js**

## 第一节 核心概念

一个配置文件的基本结构如下：

//配置项  
module.exports={  
 //入口   
 entry:".....",  
  
 //输出配置   
 output:{......},  
  
 //模块   
 module: {......},  
  
 //解析   
 resolve：{......},   
  
 //插件（数组）  
 plugins:[......],  
  
 //开发服务器   
 devServer:{......}   
};

|  |  |
| --- | --- |
| 核心 | 说明 |
| entry | **入口** 定义入口文件，**默认入口文件：./src/index.js** |
| output | **输出** 定义出口文件，**默认出口文件：./dist/main.js** |
| resolve | **解析** 路径映射、省略后缀名等 |
| module | **模块** 定义不同loader，让 webpack 能够处理非 JavaScript 模块 |
| plugins | **插件** 扩展webpack功能 |
| devServer | **开发服务器** 用于配置webpack-dev-server选项 |

## 第二节 简单的配置文件

### 创建配置文件

在项目根目录下创建**webpack.config.js**，webpack运行时，会自动到项目根目录查找该文件并执行。

hello world案例，可以通过设置配置文件自动完成：

//webpack是基于node构建的，只支持CommonJS模块  
module.exports={   
 //入口配置  
 entry:'./src/js/main.js',   
  
 //出口配置  
 output:{   
 path:\_\_dirname +'/dist', //输出目录 \_\_dirname:本文件所在硬盘路径（node全局变量）  
 filename:'js/main.js' //文件名称（可以有子目录）  
 }  
};

### 修改webpack.json文件

在webpack.json中的"scripts"下增加：

"scripts": {   
 "dev": "webpack --mode development",  
 "build": "webpack --mode production"   
 },

### 执行打包

npm run dev

# 第六章 配置详解

## 第一节 entry 和 output

[entry](https://www.webpackjs.com/configuration/entry-context/#entry) **入口配置** 是指页面中的入口文件。也就是打包从哪个文件开始。**默认入口文件：./src/index.js**

[output](https://www.webpackjs.com/configuration/output/) **出口配置** 是指生成的文件输出到哪个地方去，**默认出口文件：./dist/main.js**，主要有以下属性：

|  |  |
| --- | --- |
| output属性 | 说明 |
| path | 输出路径 |
| filename | 输出文件名 |

//入口  
entry: {   
 index: './src/js/main.js',   
},  
//输出  
output: {  
 path: \_\_dirname + "/dist",//打包后的文件存放的地方  
 filename:"main.js"//打包后输出的文件名  
},

## 第二节 module

webpack 只能打包js文件（只理解 JavaScript语法），无法识别其他语法的文件，如果要让webpack打包其他文件，首先需要让webpack识别不同文件，这就需要特别的模块，这种模块统称为[loader](https://www.webpackjs.com/concepts/loaders/)。

### loader分类

|  |  |
| --- | --- |
| 分类 | 说明 |
| 转换编译 | script-loader，babel-loader，ts-loader，coffee-loader |
| 处理样式 | style-loader，css-loader，less-loader，sass-loader，postcss-loader |
| 处理文件 | raw--loader，url-loader，file-loader |
| 处理数据 | csv-loader，xml-loader |
| 处理模板语言 | html-loader，pug-loader，jade-loader，markdown-loader |
| 清理和测试 | mocha-loader，eslint-loader |

### 常用loader

|  |  |
| --- | --- |
| loader | 说明 |
| css-loader | 解析css语句 |
| style-loader | 将css-loader解析后的文本，添加<style>标签 |
| babel-loader | 将ES6+、JSX语法转成ES5低版本语法 |
| url-loader | url-loader对未设置或者小于limit byte设置的图片以base64的格式进行转换<br />对于大于limit byte的图片用file-loader进行解析 |
| file-loader | 解析项目中的url引入（包括img的src和background的url）<br />修改打包后文件引用路径，使之指向正确的文件 |
| less-loader | less编译器 |
| vue-loader | Vue也推出了自己的vue-loader，可以方便的打包 .vue文件 的代码。<br />在vue-cli（快速构建单页应用的脚手架）中得到应用。 |

#### css loader

在hello world案例中，可以在index.js中引入”style.css“文件，webpack可以一同打包：

//index.js  
import a from "./a.js";  
import b from "./b.js";  
var txt = "hello world";  
a.run();  
b.play(txt);  
  
//打包css文件  
import "./style.css"; //注意：是相对路径

安装loader（loader也是依赖包，需要安装）

npm install css-loader --save-dev  
npm install style-loader --save-dev

在module中定义ccs模块相关的配置：

module: {  
 rules: [  
 {  
 test: /.css$/, //正则  
 //把css添加到html的style标签里(style-loader要先加载)  
 loader: ['style-loader','css-loader'], //loader或者use  
 exclude: /node\_modules/, //正则 排除node\_modules目录  
 }  
 ]  
},

#### babel loader

[babel](https://www.babeljs.cn/)是一个 js 编译器，它通过语法转换器支持最新版本的 JavaScript （包括JSX、TypeScript等新语法）。 这些[插件](https://www.babeljs.cn/docs/plugins/)允许你**立刻**使用新语法，无需等待浏览器支持。

使用Babel首先要配置 **.babelrc** 文件，该文件用来设置转码规则和插件（json格式），存放在项目的根目录下。

tips：在linux系统中，rc结尾的文件通常代表运行时自动加载的文件、配置等等。

在.babelrc配置文件中，主要是对预设(presets) 和 插件(plugins) 进行配置。.babelrc配置文件一般为如下：

{  
 "presets": [  
 ["env",{"modules":false}] //modules是配置项  
 ],   
 "plugins": [  
 [ "transform-runtime",{"polyfill":false}] //polyfill是配置项  
 ]  
}

|  |  |
| --- | --- |
| 配置项 | 说明 |
| presets | **预设** 对js最新的语法糖进行编译，并不负责转译新增的api和全局对象。例如：let/const可以被编译，而String.includes、Object.assign等对象新增方法并不能被编译。<br />常用转译器：babel-preset-env、babel-preset-es2015（2016、2017）、babel-preset-latest等 |
| plugins | **插件** 控制如何转换代码，babel默认只转换新的js语法，而不转换新的API，比如 Set,，Maps，Symbol，Promise 等全局对象，transform-runtime 用来解决以上问题 |

## 第三节 plugins

插件（[plugins](https://www.webpackjs.com/configuration/plugins/)）可以扩展webpack的功能，loader不能做的处理都能交给plugin来做。

hello world案例中，只对三个js文件进行了打包压缩，对index.html并没有进行任何处理，对html文件的打包压缩，要比js 和css文件要复杂，webpack本身并不具备这种复杂的处理能力。webpack 自带一些插件 [官方插件列表](https://doc.webpack-china.org/plugins)，你可以通过npm 安装插件。

**例：HtmlWebpackPlugin**

[HtmlWebpackPlugin](https://www.npmjs.com/package/html-webpack-plugin) 插件简化了HTML文件的创建，可以通过模板文件，生成一个HTML文件，安装：

npm install html-webpack-plugin --save-dev  
//配置方法，参见后面的完整代码

## 第四节 resolve

[resolve](https://www.webpackjs.com/configuration/resolve/)（译：解析）配置webpack如何寻找模块对应的文件。webpack在启动后会从配置的入口模块触发，找出所有依赖的模块，默认会采用模块化标准里约定好的规则去寻找，你可以根据自己的需要修改默认的规则。

|  |  |
| --- | --- |
| 属性 | 说明 |
| alias | （译：别名）通过别名将原来导入路径映射成一个新的导入路径 |
| extensions | （译：扩展）数组 导入模块时，可以省略的文件后缀名 |

resolve: {  
 alias: {  
 "@": path.join(\_\_dirname,"./src") //将项目根目录下的src目录，映射为 @  
 },  
 extensions:[".js", ".json"]  
}

## 第五节 其他配置项

|  |  |
| --- | --- |
| 配置项 | 说明 |
| devtool | 是否生成以及如何生成sourcemap |
| devserver | 开启一个本地开发服务器 |
| watch | 监听文件变化并自动打包 |
| watchoption | 用来定制watch模式的选项 |
| performance | 打包后命令行如何展示性能提示，如果超过某个大小是警告还是报错 |

# 第七章 webpack-dev-server

## 第一节 安装

[webpack-dev-server](https://webpack.docschina.org/configuration/dev-server/)是一个小型的web服务器，可以自动监视项目文件的变化，自动刷新浏览器，其HMR（Hot Module Replacement 热模块替换）方式只替换更新的部分，而不是重载页面，大大提高了刷新效率。

需要本地安装 webpack和webpack-cli  
npm install webpack --save-dev  
npm install webpack-cli --save-dev  
npm install webpack-dev-server --save-dev

## 第二节 安装配置

|  |  |  |
| --- | --- | --- |
| 配置项 | 说明 | 默认值 |
| inline | 自动刷新<br />当我们对业务代码做了一些修改，保存（ctrl+s）后，页面会自动刷新，所做的修改会直接同步到页面上，不需要手动刷新页面或重启服务 | true |
| hot | 热模块替换<br /> 不用配置（通常是通过命令行 --hot 选项启动，会自动加载webpack.HotModuleReplacementPlugin插件） | true |
| host | 主机地址 |  |
| open | 自动打开浏览器，可以指定浏览器，例如：--open 'Chrome' | false |
| port | 端口 | 默认8080 |
| overlay | 编译出错的时候，在浏览器页面上显示错误 | false |
| stats | 用来控制编译的时候shell上的输出内容<br />stats: "errors-only" 只打印错误<br /> 还有"minimal"，"normal"，"verbose" |  |
| compress | true：对所有服务器资源采用gzip压缩 | false |
| contentBase | 指定了服务器资源的根目录，<br />如果不写入contentBase的值，那么contentBase默认是项目的目录 | "./" |
| historyApiFallback | 它使用的是**HTML5 History Api**，任意的跳转或404响应可以指向 index.html 页面 | true |

## 第三节 完整案例

一个具有完整基本功能的webpack.config.js配置文件：

let path=require("path");  
  
//HtmlWebpackPlugin插件  
let HtmlWebpackPlugin=require('html-webpack-plugin');  
let htmlPlugin=new HtmlWebpackPlugin({   
 filename:"index.html", //生成的新文件  
 template:\_\_dirname+"/src/index\_temp.html", //模板文件  
 minify:{ //压缩  
 removeComments:true, //刪除注釋  
 collapseWhitespace:true //合并空格  
 },  
});  
  
//配置项  
module.exports = {  
 //输入  
 entry:'./src/js/main.js',//主入口文件  
  
 //输出  
 output: {  
 path: \_\_dirname + "/dist", //打包后的文件存放的地方  
 filename:"main.js" //打包后输出的文件名  
 },  
  
 //模块  
 module: {  
 rules: [  
 {  
 test: /.css$/, //正则 解析css文件  
 //把css添加到html的style标签里(style-loader要先加载)  
 use: ['style-loader','css-loader'],  
 exclude: /node\_modules/, //正则 必须要写exclude！！  
 },  
 {  
 test: /\.js$/,  
 use: 'babel-loader',  
 exclude: /node\_modules/, //正则 必须要写exclude！！  
 },  
 ]  
 },  
 //插件  
 plugins:[  
 htmlPlugin   
 ],  
  
 //解析  
 resolve: {  
 alias: {  
 "@": path.join(\_\_dirname,"./src") //将项目根目录下的src目录，映射为 "@"  
 },  
 extensions:['.js', '.json']  
 },  
  
 //开发服务器  
 devServer: {  
 inline:true, //支持dev-server自动刷新  
 port:"8080", //端口  
 open:true, //自动打开默认浏览器  
 },  
}

scripts中增加命令（webpack-dev-serve.cmd是定义在.bin目录中的）

"scripts": {  
 "dev": "webpack-dev-server --hot",  
 "build": "webpack --mode production"  
}

## 第四节 运行

npm start

运行后，webpack-dev-server将开始运行，打开浏览器，直接输入**127.0.0.1:8080/index.html**，打包后的页面已经可以使用了，在console窗口会有如下显示：

**注意**：

index.html内无需引入main.js文件，打包后的index.html文件中会自动引入该文件。

webpack-dev-server运行后，浏览器中输出的页面，都是运行在内存中的，只有build以后，才会在dist目录中得到最终的结果文件。

# 第八章 ES6基础

## 第一节 概述

ECMAScript和JavaScript到底是什么关系？很多初学者会感到困惑，简单来说，ECMAScript是JavaScript语言的国际标准，JavaScript是ECMAScript的实现。

ECMAScript 6（以下简称ES6）是JavaScript语言的下一代标准。因为当前版本的ES6是在2015年发布的，所以又称ECMAScript 2015。也就是说，ES6就是ES2015。ES6的目标，是使得JavaScript语言可以用来编写大型的复杂的应用程序，成为企业级开发语言。

## 第二节 symbol数据类型

### Symbol定义

Symbol（译：标记、符号）是一种唯一标识符，可用作对象的唯一属性名，这样其他人就不会改写或覆盖你设置的属性值。它由全局 Symbol() 函数创建：

let s = Symbol();  
console.log(typeof s) //"symbol"

### Symbol作对象属性名

上面代码中，变量s就是一个独一无二的值。typeof运算符的结果，表明变量s是 Symbol 数据类型，而不是字符串之类的其他类型。

let s1 = Symbol('foo');  
let s2 = Symbol('bar');  
  
s1 // Symbol(foo)  
s2 // Symbol(bar)  
  
//可以转换为字符串  
s1.toString() // "Symbol(foo)"  
s2.toString() // "Symbol(bar)"

上面代码中s1和s2是两个 Symbol 值。如果不加参数，它们在控制台的输出都是Symbol()，不利于区分。Symbol函数可以接受一个字符串作为参数，表示对 Symbol 实例的描述，输出的时候就能够分清，到底是哪一个值。

Symbol永远不相等，即便创建它们的时候传入了相同值的参数，因此，可借助此特性解决属性名的冲突问题，这也是该数据类型存在的主要用途。

Symbol 值不能与其他类型的值进行运算（会报错），可以显式转为字符串。

### 使用场景

为某个对象添加属性，可能会遇到的问题：

1. 新添加的属性与原有属性重名，
2. 新添加的属性仅作为标记使用，不需要用遍历器遍历出来

另外，不同人都可以添加属性，不用考虑重名问题，此时，就可以使用Symbol数据类型。

let obj = {  
 "id": "box",  
 "width": 300,  
 "height": 100,  
 "symbol1": "redBox",  
 "symbol2": "ac"  
};  
  
//zhangsan定义了一个标记  
let symbol1 = Symbol("zhangsan");  
obj[symbol1] = "zhangsan\_symbol";//不能用点运算符定义Symbol属性，只能用object[symbol]格式  
  
//lisi定义了一个标记  
let symbol2 = Symbol("lisi");  
obj[symbol2] = "lisi\_symbol";  
  
//查看新增的symbol属性  
console.log(obj[symbol1]);// "zhangsan\_symbol"  
console.log(obj[symbol2]);// "lisi\_symbol"  
  
//obj原有属性不会被覆盖  
console.log(obj.symbol1);// "redBox"  
console.log(obj.symbol2);// "ac"  
  
//symbol数据类型，不会被遍历出来  
for(let key in obj){  
 console.log(key +":"+ obj[key]);  
}

## 第三节 let与const

用var声明的变量会造成全局污染，于是，就产生了新的声明方式。

### let

用let声明变量，**要先声明后使用**

for(let i=0; i<10; i++){  
 console.log(i);//0-9  
}  
console.log(i); //结果：undefined

### const

const声明一个只读的常量。一旦声明，常量的值就不能改变。

const PI = 3.1415;  
PI = 3;// 报错：TypeError: Assignment to constant variable.（类型错误：给常量赋值）

|  |  |  |  |
| --- | --- | --- | --- |
| 特性 | var | let | const |
| 作用域级别 | 函数级 | 块级 | 块级 |
| 初始值 | 否 | 否 | 声明时必须赋值 |
| 变量提升 | 是 | 否 | 否 |
| 重复声明 | 是 | 否 | 否 |

## 第四节 变量的解构赋值

如果我们想从复杂的数据结构（数组 对象）中获取某一个数据，可能需要大量的遍历操作才能够完成。解构赋值，可以将这一过程进行简化。

### 字符串的解构赋值

let [a, b, c, d] = "hxsd";  
console.log(a, b, c, d); // 结果：h x s d

### 数组的解构赋值

//旧  
var arr = ["zhangsan", 18];  
var name = arr[0];  
var age= arr[1];  
console.log(name,age);  
  
//ES6方法：结构一致，一一对应  
let [name,age] = ["zhangsan", 18];  
console.log(name,age);  
  
let [x,[y,z]] = [7, [8, 9]];  
console.log(l,m,n);

### 对象的解构赋值

let obj = {  
 name:{firstName:"zhang",lastName:"san"},  
 city: "beijing",  
 hobby: ["足球","篮球","游戏","读书"]  
};  
//变量名要与对象key一致，可以使用冒号':'定义新变量名  
let {name:new\_name,name:{firstName:fname,lastName},city, hobby} = obj;   
console.log(new\_name,fname,hobby);

## 第五节 Set与Map

JS原有2种数据结构：Array 和 Object，ES6新增两种数据解构：Set 和 Map

### Set数据结构

Set类似于数组，但是成员的值都是唯一的，没有重复的值。主要用于**数据去重**

Set本身是一个构造函数，用来生成 Set 数据结构。set实例是一个**类数组的对象**。

//声明的同时，传入数据（数组格式）  
let set = new Set([1, 1, 2, 2, 2, "s", "s"]);  
console.log(set); //返回的是一个对象  
  
set.add("s");//不能插入重复数据  
  
//set实例是一个类数组的对象，可用Array.from方法转换为数组  
let arr=Array.from(set);  
console.log(arr)//[1, 2, "s"]

|  |  |
| --- | --- |
| 方法 | 说明 |
| add(value) | 添加某个值，返回：Set结构本身 |
| delete(value） | 删除某个值，返回：bool（删除是否成功） |
| has(value) | 表示该值是否为Set的成员 返回：bool |
| clear() | 清除所有成员，返回：无 |

### Map数据结构

Map（译：映射）也是一个数据集合，与数组类似，是对Object的一个补充，Map的key可以是任意类型的（数组、对象、数字都可以），而传统对象的key必须是字符串

遍历Object得到的结果是无序的，遍历Map得到结果是有序的

let obj1={{a:1}: 1} //报错,key只能是字符串  
let obj2={"{a:1}" : 1}

//Map数据结构  
let map = new Map();  
  
//添加数据  
map.set("data","string");  
map.set(1,"num"); //key是任意类型  
console.log(map);  
  
//声明时赋值  
let map=new Map([ ["data","string"] , [1,"num"] ]);  
  
//map删除数据  
map.delete('data');  
console.log(map);

|  |  |
| --- | --- |
| 方法 | 说明 |
| clear | 从Map中移除所有元素 |
| delete | 从Map中移除指定的元素 |
| forEach | 对Map中的每个元素执行指定操作 |
| get | 返回Map中的指定元素 |
| has | 如果Map包含指定元素，则返回 true |
| set | 添加一个新建元素到Map |
| toString | 返回Map的字符串表示形式（序列化） |
| valueOf | 返回指定对象的原始值 |

### Array Set Map对比

|  |  |  |  |
| --- | --- | --- | --- |
| 类别 | Array | Set | Map |
| 长度 | arr.length | set.size | map.size |
| 增 | arr.push(新增值) | set.add(4) | map.set('t', 1) |
| 删 | arr.splice(索引,删除数量) | set.delete(2) | map.delete('t') |
| 改 | arr.splice(索引,删除数量,[新增值]) | 遍历 | map.set('t',2) |
| 查 | 遍历 | set.has(1) | map.has('t') |
| 清空 | arr = [] | set.clear() | map.clear() |

### Array Set Map共同遍历方法

|  |  |
| --- | --- |
| 方法 | 说明 |
| keys() | 返回键名的遍历器 |
| values() | 返回键值的遍历器 |
| entries() | 返回键值对的遍历器 |
| forEach() | 使用回调函数遍历每个成员 |

## 

## 第六节 箭头函数

箭头函数（arrow function）**相当于匿名函数**，并且简化了函数定义。

### 新增格式

箭头函数增加了一种简化格式：如果函数体只包含一个表达式，可以省略掉{ ... }和return

#### 只包含一个表达式

//旧  
var run=function(){alert("hxsd")}  
  
//ES6 可以省略{}  
let run=()=>alert("hxsd")

#### 只有return语句

//旧  
var sum = function() { return 1 + 2 }  
  
//ES6  
let sum=()=>1+2;

#### 返回对象

//旧  
var getObj=function() {  
 return {  
 id:1,  
 name:"zhangsan"  
 }  
}  
  
//ES6 返回对象用小括号  
let getObj=()=>({id:1, name: "zhangsan"});

### 参数格式

//无参数---空()  
let sum=()=>{...}; //相当于 var sum=function(){...}  
  
//单个参数---省略()  
let fn = val => val; //相当于： var fn = function(val) {return val}  
  
//多个参数---放在()内  
let sum = (a, b) => a + b; //相当于：var sum = function(a, b) {return a + b;}

### 箭头函数中的this

箭头函数的this，与父函数一致

<div id="box" style="width:200px;height:200px;background:red"></div>  
  
<script>  
 let oBox=document.getElementById("box");  
 oBox.onclick=function(){  
 setTimeout(()=>{  
 this.style.background="blue" //箭头函数的this，与父函数一致 },1000)  
 }  
</script>

## 

## 第七节 ES6相对于ES5扩展

### 函数的扩展

ES6中函数的扩展包含默认值、剩余运算符和扩展运算符。

function log(x,y){  
 y = y||'world';  
 console.log(x,y);  
}  
//参数可以有默认值  
function log(x,y='world'){  
 console.log(x,y);  
}

剩余运算符

function rest01(...arr) {  
 for (let item of arr) {  
 console.log(item);  
 }  
 }  
 rest01(1, 3, 5);

扩展运算符

function test(a,b,c){  
 console.log(a);  
 console.log(b);  
 console.log(c);  
}  
var arr = [1, 2, 3];  
test(...arr);

### 对象的扩展

ES6中允许向对象直接写入变量和函数，作为对象的属性和方法。

更加简洁的表现方式。ES6中允许使用表达式作为对象属性，并且函数名称定义也可以采用相同的方式。

setter和getter。JavaScript对象的属性是由名字、值和一组特性（可写、可枚举、可配置等）构成的。在ES6中，属性值可以用一个或两个方法代替，这两个方法就是getter和setter。具体的代码例子这里就不做详细讲解了。

其他操作方法如下表：

|  |  |
| --- | --- |
| **方法名称** | **方法描述** |
| Object.is() | 比较两个值是否相等 |
| Object.assign() | 用于将对象进行合并 |
| Object.getOwnPropertyDescriptor | 返回对象属性的描述 |
| Object.keys() | 返回一个数组，包括对象自身的所有的可枚举属性 |

### 数组的扩展

ES6中数组扩展了一些方法，常用的详见下表：

|  |  |
| --- | --- |
| **方法** | **方法描述** |
| copyWithin(target，start，end) | 在当前数组内部，将指定位置的成员复制到其他位置（会覆盖原有成员），然后返回当前数组。也就是说，使用这个方法，会修改当前数组。 参数说明：<br />target（必需）：从该位置开始替换数据。负值:倒数。 <br />start（可选）：从该位置开始读取数据，默认:0。负值:倒数。<br /> end（可选）：到该位置前停止读取数据，默认等于数组长度。负值:倒数。 |
| find() | 数组实例的find方法，用于找出第一个符合条件的数组成员。它的参数是一个回调函数，所有数组成员依次执行该回调函数，直到找出第一个返回值为true的成员，然后返回该成员。如果没有符合条件的成员，则返回undefined。 |
| findIndex() | findIndex方法的用法与find方法非常类似，返回第一个符合条件的数组成员的位置，如果所有成员都不符合条件，则返回-1。 |
| fill() | fill方法使用给定值，填充一个数组,fill方法用于空数组的初始化非常方便。数组中已有的元素，会被全部抹去，如果填充的类型为对象，那么被赋值的是同一个内存地址的对象，而不是深拷贝对象。 |
| includes() | 该方法返回一个布尔值，表示某个数组是否包含给定的值，与字符串的includes方法类似。 |

## 第八节 ES6高级操作

### promise对象

JavaScript本身是单线程，因此，为了应对一些单线程带来的问题，异步编程成为了JavaScript中非常重要的一部分。在异步编程中，许多操作都会放在回调函数（callback）中。同步与异步的混杂、过多的回调嵌套都会使得代码变得难以理解与维护，这也是常受人诟病的地方。

异步编程的回调地狱

//晚饭  
let state=1;//假设一个判断条件  
  
function dinner(){  
 setTimeout(()=>{  
 if(state==1){  
 console.log("1.做饭");  
 setTimeout(()=>{  
 if(state==1){  
 console.log("2.吃饭");  
 setTimeout(()=>{  
 if(state==1){  
 console.log("3.刷碗");  
 }else{  
 console.log("刷碗出错");  
 }  
 },1000)  
 }else{  
 console.log("吃饭出错");  
 }  
 },1000)  
 }else{  
 console.log("做饭出错");  
 }  
 },1000)  
}  
  
//调用  
dinner()

Promise对象用于处理异步操作的，异步处理成功了就执行成功的操作，异步处理失败了就捕获错误或者停止后续操作。 它的核心就是：一个promise代表一个任务结果，这个任务有可能完成有可能没完成。Promise模式唯一需要的一个接口是调用then方法，它可以用来注册当promise完成或者失败时调用的回调函数，可以把Promise对象看成是一条工厂的流水线， 使用promise对象之后可以使用一种链式调用的方式来组织代码；让代码更加的直观。

Promise 是一个类，需要用 new 实例化：

let obj=new Promise(function(resolve,reject){//resolve：决定 reject：拒绝  
 //做一些异步操作  
 //成功：resolve()  
 //失败：reject()  
})

then方法

then方法是Promise原型上的方法，它把原来的回调写法分离出来，在异步操作执行完后，用链式调用的方式执行回调函数。

1. then 方法接受两个参数：onFulfilled（译：完成的）和onRejected（译：被拒绝的）就是前次操作返回的resolve()和reject()
2. 回调函数中会把上一个then中返回的值当做参数值供当前then调用。每个then只可能使用前一个then的返回值。
3. then方法执行完毕后需要返回一个新的值给下一个then调用（没有返回值默认使用undefined）。

//promise  
let state=1;//假设一个判断条件  
  
function cook(){  
 return new Promise(function(resolve,reject){//resolve:决定 决心 reject:拒绝 抛弃  
 setTimeout(()=>{  
 if(state==1){  
 resolve("1.做饭");  
 }else{  
 reject("做饭出错");  
 }  
 },1000)  
 });  
}  
  
function eat(){  
 return new Promise(function(resolve,reject){  
 setTimeout(()=>{  
 if(state==1){  
 resolve("2.吃饭");  
 }else{  
 reject("吃饭出错");  
 }  
 },1000)  
 });  
}  
  
function wash(){  
 return new Promise(function(resolve,reject){  
 setTimeout(()=>{  
 if(state==1){  
 resolve("3.刷碗");  
 }else{  
 reject("刷碗出错");  
 }  
 },1000)  
 });  
}  
  
//实例化一个promise对象  
cook().then(  
 data=>{  
 console.log(data);  
 return eat();  
 }  
).then(  
 data=>{  
 console.log(data);  
 return wash();  
 }  
).then(  
 data=>{  
 console.log(data);  
 }  
)

### Iterator

Iterator（遍历器）是一种接口，为各种不同的数据结构提供统一的访问机制。任何数据结构只要部署Iterator接口，就可以完成遍历操作（即依次处理该数据结构的所有成员）。

Iterator的作用有三个：

1. 为各种数据结构，提供一个统一的、简便的访问接口；
2. 使得数据结构的成员能够按某种次序排列；
3. ES6创造了一种新的遍历命令for...of循环，Iterator接口主要供for...of消费。

**Array**、**Set**、**Map**、**String**都是可迭代对象，它们的原型中都有一个Symbol.iterator方法，通过调用Symbol.iterator方法来获取默认迭代器，这一过程是在JavaScript引擎背后完成的。

在下面的实例中就是为对象添加了Iterator接口。

let obj = {  
 data: [ 'hello', 'world' ],  
 [Symbol.iterator]() {  
 const self = this;  
 let index = 0;  
 return {  
 next() {  
 if (index < self.data.length) {  
 return {  
 value: self.data[index++],  
 done: false  
 };  
 } else {  
 return { value: undefined, done: true };  
 }  
 }  
 };  
 }  
};

### Generator

Generator是ES6提供的一种异步编程解决方案，在语法上，可以把它理解为一个状态机，内部封装了多种状态。执行Generator，会生成返回一个遍历器对象。返回的遍历器对象，可以依次遍历Generator函数的每一个状态。同时ES6规定这个遍历器是Generator函数的实例，也继承了Generator函数的prototype对象上的方法。

最简单的Generator函数,其实它就是一个普通的函数，但是它有两个特征。第一就是function关键字与函数名之间有一个\*号，其二就是函数体内使用yield表达式来遍历状态。

function\* newGenerator(){  
 yield 'hello';  
 yield 'world';  
 return 'ending';  
}

执行newGenerator函数之后，该函数并不会立即执行，返回的也不是函数运行结果，而是一个指向内部状态的指针对象。通常使用遍历器对象的next方法。使得指针移向下一个状态。每一次调用next()方法，内部指针就从函数头部或上一次停下里的地方开始执行，直到遇到下一个yield表达式位置，由此可以看出，Generator是分段执行的，yield表达式是暂停执行的标记，而next方法可以恢复执行。

* Generator中的yield表达式
* yield表达式在Generator中是作为一个暂停标志，当碰到yield时，函数暂停执行，等到下一次next()执行时，函数才从当前yield位置开始执行。

function\* gen() {  
 yield 'hello';  
 yield\* 'hello';  
}  
let f = gen();  
console.log(f.next().value);  
console.log(f.next().value);  
console.log(f.next().value);  
console.log(f.next().value);  
console.log(f.next().value);

* Generator中的next函数
* 通过next函数，可以执行对应的yield表达式，且next()函数还可以带参数，该参数可以作为上一次yield表达式的返回值，因为yield本身是没有返回值的，如果next()中不带参数，则yield每次运行之后的返回值都是为undefined;在下面的实例中，第一次运行next()，运行到第一个next()函数截止，第二个next运行时，传入的参数为'a';则运行到第二个yield地方截止，然后第一个yield运行的返回值为'a',依次类推。

function\* dataConsumer() {  
console.log('Started');  
console.log('1. ${yield}');  
console.log('2. ${yield}');  
return 'result';  
}  
let genObj = dataConsumer();  
genObj.next();//打印Started  
genObj.next('a');//打印1.a  
genObj.next('b');//打印2. b

### Class

传统的javascript中只有对象，没有类的概念。它是基于原型的面向对象语言。原型对象特点就是将自身的属性共享给新对象。

ES6引入了Class（类）概念，通过class关键字可以定义类。该关键字的出现使得其在对象写法上更加清晰，更像是一种面向对象的语言（ES6中的类只是语法糖，其实就是构造函数的另外一种写法）。

//1. 在class的{}区间内，只能写 构造函数、静态方法、实例方法  
//2. 实例属性和方法，写在构造函数内  
  
class Animal{  
 //构造函数  
 constructor(name,age){  
 this.name=name; //实例的属性  
 this.age=age;//实例的属性  
 }  
 //实例的方法  
 show(){  
 console.log(this.name);//实例方法中的this，指向实例对象  
 }  
  
 //类的静态方法  
 static getColor(){  
 console.log("类的静态方法："+this.color);//静态方法中的this 指向 类 本身  
 }  
}  
//定义类的属性（ES6规定，类里只可以有静态方法，不能有静态属性）  
Animal.color="白色";  
  
//调用类的静态方法  
Animal.getColor();  
  
//依然可以使用prototype添加方法（不推荐）  
Animal.prototype.eat=function(food){  
 console.log(food);  
}  
  
//实例化一个动物对象  
let dog=new Animal("小黑",2);  
dog.show();

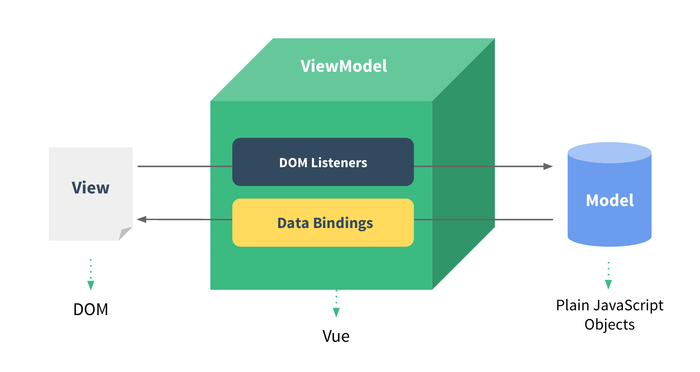
**第二篇 前端高效开发框架技术与应用**

# 第一章 Vue基础

## 第一节 框架简介

**mvx模式介绍**

介绍Vue框架之前，我们先了解一下MVX模式，X是未知数的意思。下面我们主要介绍一下MVC模式、MVVM模式、相互之间的区别以及Web前端开发经历这么几个阶段。

* MVC模式
* MVC模式是移动最广泛的软件架构之一，把应用程序强制性地划分为三部分：模型（Model）、视图（View）和控制器（Controller）。
* MVVM模式
* MVVM模式是把MVC模式的Controller改成ViewModel。View的变化会自动更新ViewModel，ViewModel的变化也会自动同步到View上显示。
* 

**Web前端开发阶段**

* 原生代码开发阶段：直接使用原生 HTML、CSS和JavaScript代码进行前端开发。
* 优点：学习成本较低，容易入手
* 缺点：代码结构混乱，代码冗余，浏览器兼容性不成熟，不利于分工合作
* 代码库开发阶段：使用成熟开源的扩展库进行前端开发（如在JS中可以使用jQuery、在CSS样式中可以使用Bootstrap响应式框架等）。
* 优点：开发快速，浏览器兼容性良好
* 缺点：视图层和数据层混合在一起，不处于团队分工合作
* 框架开发阶段：采用前端 MVC模式 或 MVVM模式开发。
* 优点：代码分层，便于团队合作，便于后续代码维护
* 缺点：学习成本较高，框架更新换代较快

**Vue框架概述**

Vue是一套构建用户界面的渐进式框架，Vue 只关注视图层，采用自底向上增量开发的设计，Vue 的目标是通过尽可能简单的 API 实现响应的数据绑定和组合的视图组件。Vue框架中主要内容：

|  |  |
| --- | --- |
| 模板 | 主要内容 |
| Vue基础知识 | Vue框架基础语法、Vue开发工具 |
| Vue组件 | Vue组件、Vue模块化开发 |
| Vue工程化开发 | npm/yarn和webpack开发工具、Vue-cli开发工具、单文件工具 |
| Node.js基础和Axios网络请求 | Node.js构建Web服务器、Axios发送ajax请求、postman网络调试工具 |
| Vue-Router路由 | 单页面应用SPA、Vue-Router实现路由机制 |
| Vuex状态管理 | Vuex状态管理、本地存储 |
| Vue-UI库 | Vue中的UI库 |

**主流框架优缺点和应用场景**

对于三大框架主要熟悉的它的优点，缺点以及应用使用场景如表所示。

|  |  |  |  |
| --- | --- | --- | --- |
|  | Angular | React | Vue |
| 发布时间 | 2009年 | 2013年 | 2014年 |
| 维护者 | 谷歌 | Facebook | 尤雨溪 |
| 视图引擎 | HTML | JSX | HTML |
| 学习曲线 | 陡峭 | 较复杂 | 简单 |
| 大小 | 500K+ | 130K+ | 50K+ |
| 功能复杂度 | 完善 | 完善 | 简单 |
| 文档 | 英文 | 英文 | 中文 |
| 应用场景 | 大型复杂应用 | 中大型应用、移动跨平台开发 | 中小型轻量级应用 |

**如何使用Vue.js**

**安装**

* 在Vue.js 的官网上直接下载vue.min.js 并用<script> 标签引入。
* CDN 方式引入 vue.js 文件
* Vue推荐使用的地址：https://unpkg.com/vue
* Staticfile CDN（国内）：https://cdn.staticfile.org/vue/2.2.2/vue.min.js
* unpkg：https://unpkg.com/vue/dist/vue.js
* cdnjs ：https://cdnjs.cloudflare.com/ajax/libs/vue/2.1.8/vue.min.js

**第一个Vue实例**

*<!--Vue的MVVM模式--->***<body>** *<!--View-->* **<div** id="app"**>** **<h1>**你好，{{ name }}**</h1>** **</div>****</body>** *<!--vue.js--->***<script** src="js/vue.js"**></script>****<script>** *//Model--必须是对象* **let** model = { name: 'vue application' } *//创建一个viewModel* **let** vm=**new** Vue({  el: '#app', data:model *//model* })**</script>**

## 

## 第二节 基础语法

**实例对象**

Vue实例对象是Vue框架的一个核心，对于Vue实例对象需要掌握Vue实例对象的基本使用、同时也要深刻的理解一下Vue实例对象包含的深刻思想，包括数据的双向绑定机制、实例对象的访问方法、设置方法等。

上面代码，使用 let vm=new Vue({……}) 创建了一个Vue实例，以对象方式传入配置项，每一个配置项代表一个功能模块，常用的配置项，包括：

|  |  |  |
| --- | --- | --- |
| 配置项 |  | 说明 |
| el | string | 把 Vue 实例挂载到DOM元素上，通过id绑定html元素 |
| data | object | 数据对象，Vue实例的数据（**注意：数据不要与methods中的方法重名**） |
| methods | object | 事件对象，包含事件所要触发的函数（**注意：方法名不要与data中的数据重名**） |
| computed | object | 计算属性 |
| watch | object | 监听器 |
| directives | object | 自定义指令 |
| 钩子（hook）函数（8个） | function | hook（钩子）函数，不同生命周期引发的动作 |
| 路由钩子函数（3个） | function | 路由组件在不同状态时触发 |
| components | object | 组件容器 |
| template | string | 定义模板，可以是字符串，也可以是”#“选择器 |
| props | array/object | 用于接收来自父组件的数据 |
| router | object | 路由 |
| store | object | vuex 状态 |

实例属性/方法

|  |  |
| --- | --- |
| 属性 | 说明 |
| vm.$el | Vue 实例使用的根 DOM 元素 |
| vm.$data | Vue的data配置项 |
| vm.$options | 用于当前 Vue 实例的初始化选项 |
| vm.$props | 当前组件接收到的 props 对象 |
| vm.$parent | 父实例（如果当前实例有的话） |
| vm.$root | 当前组件树的根 Vue 实例 |
| vm.$children | 当前实例的直接子组件 |
| vm.$refs | 原生DOM元素或子组件注册引用信息 |
| vm.$slots | 用来访问被插槽分发的内容 |
| vm.$router | 全局路由（vue-router插件） |
| vm.$store | vuex 状态对象（vuex插件） |
| 方法 | 说明 |
| vm.$emit() | 子组件可以使用 $emit 触发父组件的自定义事件 |
| vm.$set() | Vue.set的别名<br /> 设置对象的属性， 这个方法主要用于避开 Vue 不能检测属性被添加的限制 |
| vm.$watch | 侦听数据变化 |
| vm.$on() | 监听当前实例上的自定义事件。事件可以由*vm.$emit*触发。回调函数会接收所有传入事件触发函数的额外参数。 |
| vm.$mount | 可以使用 *vm.$mount()* 手动挂载（Vue 实例化时没有 el 选项） |
| vm.$destroy | 完全销毁一个实例。清理它与其它实例的连接，解绑它的全部指令及事件监听器。<br /> 触发 *beforeDestroy* 和 *destroyed* 的钩子。 |
| 属性绑定指令 | 说明 |
| v-bind | 动态改变dom标签上的属性<br />v-bind :class="" 简写 :class="" |

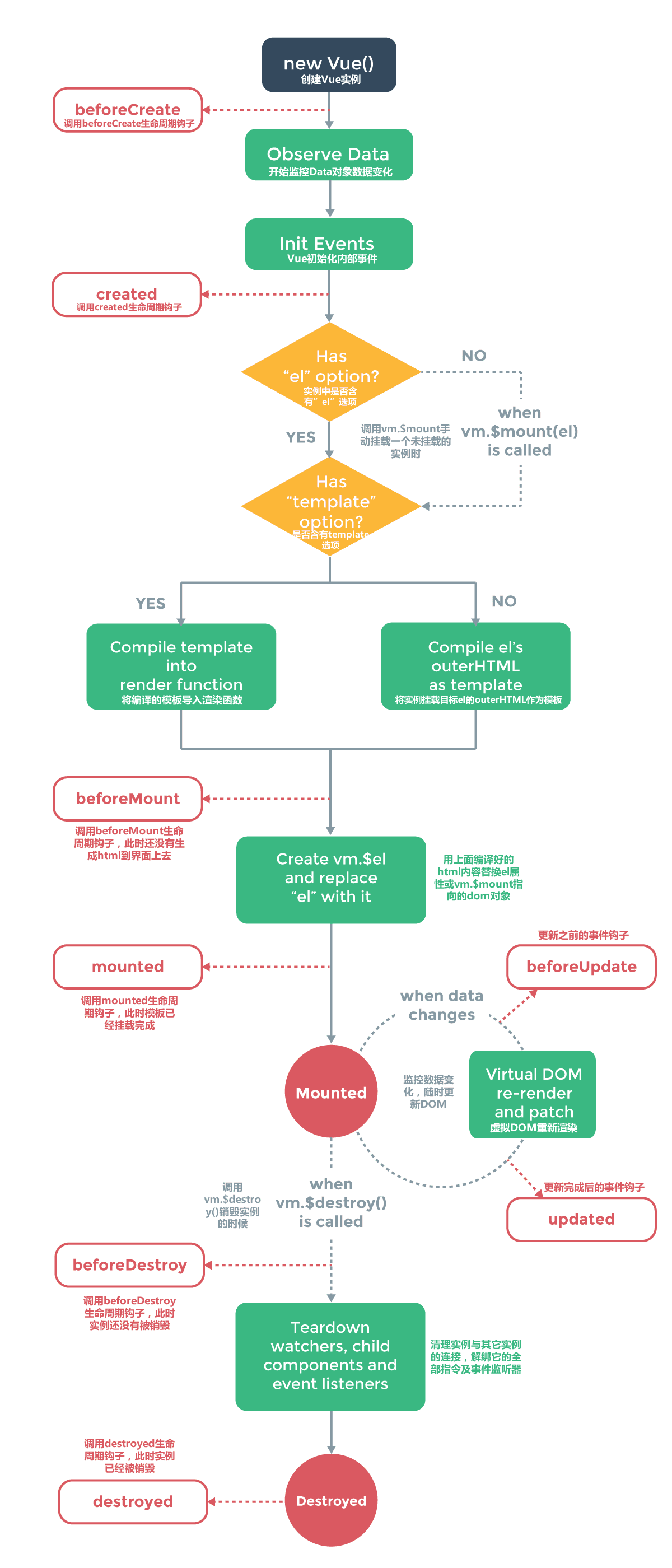
*<!--分写方式-->  
<div class="box" :class='bool? r\_cls:"" '></div>  
  
<!***--***数组方式-->  
<div :class='["box",bool? "red":""]'></div>  
  
<!***--***对象方式-->  
<div :class='{"box":true,"red":bool}'></div>  
  
<button @click="addClass">添加red类</button>  
  
data:{  
 r\_cls:"red",  
 bool:false  
},  
methods:{  
 addClass(){  
 this.bool=true;  
 }  
}*

*<!--行内样式-变量形式-->***<div** :style="{color:activeColor, fontSize:fontSize+'px'}"**>**创新时代**</div>**data: { activeColor:'red', fontSize: 30} *<!--行内样式-对象形式-->***<div** :style="style\_obj"**></div>**data: { style\_obj: { color:'red', fontSize:'13px' }}

*//拼字符串*<a :href="'http://'+url">{{link}}</a>data: { "url":"www.hxsd.com", "link":"创新时代"}

**生命周期**

Vue实例从开始创建、初始化数据、编译模板、挂载Dom、渲染→更新→渲染、卸载等一系列过程，我们称之为生命周期（life cycle）。这个过程可归纳为4个阶段：**创建→挂载→更新→销毁**



|  |  |  |
| --- | --- | --- |
| 生命周期钩子 | 运行时机 | 典型应用场景 |
| beforeCreate | 实例创建前 |  |
| created | 实例创建后 | 初始化数据（Ajax数据请求，获取Vuex状态、路由切换等） |
| beforeMount | 载入前 |  |
| mounted | 载入后 | 需要操作DOM时（应用第三方插件等） |
| beforeUpdate | 更新前 |  |
| updated | 更新后 | 通过事件修改数据、v-model引发数据变化、AJAX异步更新数据 |
| beforeDestroy | 实例销毁前 |  |
| destroyed | 实例销毁后 | 切换路由（组件将被销毁） |

**<div** id="box"**>** **<p** id="para"**>**{{msg}}**</p>** **<input** type="text" v-model="msg"**>****</div>****<script>** **let** vm = **new** Vue({ el:'#box', data:{ msg:'word', }, beforeCreate(){ console.log('beforeCreate---------------创建前，仅在内存开辟空间'); console.log('el:'+**this**.$el); console.log('data:'+**this**.$data); console.log('msg:'+**this**.msg); }, created(){ console.log('created--------------------创建完成'); console.log('el:'+**this**.$el); console.log('data:'+**this**.$data); console.log('msg:'+**this**.msg); }, beforeMount(){ console.log('beforeMount----------------挂载前'); console.log('el:'+**this**.$el); console.log('data:'+**this**.$data); console.log('msg:'+**this**.msg); **var** para = document.getElementById('para').innerText; console.log('p标签:'+para); }, mounted(){ console.log('mounted--------------------挂载完成'); console.log('el:'+**this**.$el); console.log('data:'+**this**.$data); console.log('msg:'+**this**.msg); **var** para = document.getElementById('para').innerText; console.log('p标签:'+para); }, beforeUpdate(){ console.log('beforeUpdate---------------更新前'); console.log('el:'+**this**.$el); console.log('data:'+**this**.$data); console.log('msg:'+**this**.msg); **var** para = document.getElementById('para').innerText; console.log('p标签:'+para); }, updated(){ console.log('updated--------------------更新完成'); console.log('el:'+**this**.$el); console.log('data:'+**this**.$data); console.log('msg:'+**this**.msg); **var** para = document.getElementById('para').innerText; console.log('p标签:'+para); }, beforeDestroy(){ console.log('beforeDestroy---------------销毁前'); console.log('el:'+**this**.$el); console.log('data:'+**this**.$data); console.log('msg:'+**this**.msg); }, destroyed(){ console.log('destroyed-------------------销毁完成'); console.log('el:'+**this**.$el); console.log('data:'+**this**.$data); console.log('msg:'+**this**.msg); } }) *//手动挂载* *//vm.$mount("#box");* *//销毁实例* *//vm.$destroy();***</script>**

**模板语法**

所谓模板语法主要是如何在HTML中输出Vue对象中的数据，具体而言包括：我们在HTML**元素**中输出Vue对象的变量，或者在HTML的**属性**中输出Vue对象的变量，在很多情况下，需要做一些简单的计算，针对简单的计算，我们可以输出一些JS表达式。

<!DOCTYPE html>**<html>****<head>** **<meta** charset="UTF-8"**>** **<title>**Vue模板语法</title> **<script** type="text/javascript"**></script>****</head>****<body>** **<div** id="app"**>** **<div>**name = {{name}}**</div>** **<div>** user.name = {{user.name}}**<br>** user.desc = {{user.desc}} **</div>** **<ul>** **<li>**{{names[0]}}**</li>** **<li>**{{names[1]}}**</li>** **<li>**{{names[2]}}**</li>** **</ul>** **</div>** **<script** src="https://unpkg.com/vue"**></script>** **<script>** *//准备双向绑定数据* **var** data={ name:'vue',*//普通字符串* user:{ *//对象* name:'vue', desc:'this is vue application' }, names:['vue','angular','react'], *//数组* } **var** vm = **new** Vue({ el:'#app', data:data }); **</script>****</body>****</html>**

**计算属性**

计算属性----顾名思义就是通过其他变量计算得来的另一个属性 。

计算属性具有缓存----多次访问计算属性会立即返回之前的计算结果，而不必再次执行函数。

**注意：computed中的函数，不能传参。**

<!DOCTYPE html>**<html>****<head>** **<meta** charset="UTF-8"**>** **<title>**Vue计算属性**</title>** **<script** type="text/javascript"**></script>****</head>****<body>** **<div** id="app"**>** **<div>** 订单金额：{{amount}} **</div>** **</div>** **<script** src="https://unpkg.com/vue"**></script>** **<script>** *//准备双向绑定数据* **var** data={ count:5, *//数量* price:12, *//单价* discount:0.8, *//折扣率* freight:5 *//运费* }; **var** vm = **new** Vue({ el:'#app', data:data, *//定义计算属性* computed:{ *//每一个计算属性值是函数形式，并且必须有返回值* amount:**function**(){ *//this对象表示当前Vue对象* *//this.$data 表示当前Vue对象装载的双向绑定数据* *//代理：this.$data.count<=>this.count* **return** **this**.$data.count\***this**.$data.price\* **this**.$data.discount+**this**.$data.freight; } } }); **</script>****</body>****</html>**

**Methods方法**

Vue对象中可以使用 methods属性 ，用来编写自定义函数，使用方法是methods 属性由一系列 JSON方法对组成（同computed一样）；在模板中使用是在 {{ }} 中调用方法即可（加小括号）。

methods:{ //订单金额方法：函数，需要返回值 getAmount:function(){ return this.$data.count\*this.$data.price\* this.$data.discount+this.$data.freight; }}**<div>** 订单金额(函数形式)：{{getAmount()}}**</div>**

## 

## 第三节 渲染

**列表渲染**

* **循环数组**

可以有 2 个参数

list = [1,7,15]；**<span** v-for="item in list"**>**{{ item }} **</span>**list = [1,7,15]；**<span** v-for="（item，index） in list"**>**{{index}}：{{item}} **</span>**

* **循环对象**

可以有 3 个参数

**<li** v-for="value in obj"**>** {{value}}**</li>****<li** v-for="(value,key) in obj"**>** {{value}}:{{key}}**</li>****<li** v-for="(value,key,index) in obj"**>** {{key}}:{{value}}:{{index}}**</li>**

**数组的更新检查**

|  |  |
| --- | --- |
| 方法 | 说明 |
| push()、pop()、shift()、unshift()<br />splice()、sort()、reverse() | 变异方法，可触发视图更新 |
| filter(), concat() 和 slice() | 非变异方法，不触发视图更新<br />它们可以返回新数组，用新数组替换旧数组，就可以刷新视图 |

受 JavaScript 的限制，Vue 不能检测以下变动的数组：

1. 当你利用索引直接设置一个项时，例如：*vm.items[i] = newValue*
2. 当你修改数组的长度时，例如：*vm.items.length = 0*

Vue.set(this.list, index, newValue)

**对象的更新检查**

受 JavaScript 的限制，Vue不能检测对象属性的 **添加** 或 **删除**：

Vue.set(this.object, key, value)

**key**

Vue 会尽可能高效地渲染元素，通常会复用已有元素而不是从头开始渲染。 如果数据项的顺序被改变，Vue 将不会移动 DOM 元素来匹配数据项的顺序， 而是简单复用此处每个元素，并且确保它在特定索引下显示已被渲染过的每个元素。

为了给 Vue 一个提示，以便它能跟踪每个节点的身份，从而重用和重新排序现有元素，你需要为每项提供一个唯一key 属性。理想的 *key* 值是每项都有的且唯一的 id。

**<div** id="app"**>** *<!--先勾选b，点击“改变位置”-->* **<p** v-for="item in list" **：key**="item.id" **>** **<label><input** type="checkbox"**/>**{{item.title}}**</label>** **</p>** **<button** **@click**="change"**>**改变位置**</button>****</div>** *<!--如果没有key，checkbox只修改值，位置不会发生变化-->***<script>** **const** app = **new** Vue({ el: "#app", data: { list: [ {id:1,title:"a"}, {id:2,title:"b"}, {id:3,title:"c"} ] }, methods: { change() { **this**.list= [ {id:2,title:"b"}, {id:1,title:"a"}, {id:3,title:"c"} ] } } })**</script>**

**条件渲染**

|  |  |
| --- | --- |
| 指令 | 说明 |
| v-if | 根据其后表达式的bool值进行判断是否渲染该元素 |
| v-else-if | *v-if* 的“else-if 块”，可以连续使用 |
| v-else | 表示 *v-if* 的“else 块”<br />*v-else* 元素必须紧跟在带 *v-if* 或者 *v-else-if* 的元素的后面，否则它将不会被识别。 |

**<div** id="box"**>** **<p** v-if="name.indexOf('san')>0"**>**姓名:{{name}}**</p>** **<p** v-if="man"**>**男**</p>** **<p** v-if="women"**>**女**</p>** **<p** v-if="age>25"**>**年龄:{{age}}**</p>****</div>****<div** v-if="city=='北京'"**>**城市：{{city}}**</div>****<div** v-else**>**城市 ：不在北京</div>let vm= new Vue({ el:"#box", data:{ name:'zhangsan', age:30, man:true, women: false, city:'北京', }})

**事件与表单**

**事件处理**

事件处理需要绑定事件处理函数，在Vue中绑定事件处理，直接在HTML元素上使用 v-on 指令绑定即可。语法格式为v-on:事件名，也可以使用简写形式"@事件名"绑定事件处理函数。可用的事件有鼠标事件和键盘事件。

|  |  |
| --- | --- |
| 指令 | 说明 |
| v-on:click | 简写: @click=""<br />click、mouseover、mouseout、mouseup、mousedown、<br />dblclick 、contextmenu、keydown、keyup等 |

可以用 *v-on* 指令监听 DOM 事件，并在触发时运行一些 JavaScript 代码：

**<div** id="app"**>** *<!--表达式 -->  
 <button @click="num++">add</button>  
  
 <!***--***事件方法 可传递参数 -->* **<button** **@click**="plus(10)"**></button>**  **<p>**{{ num }}**</p>****</div>****<script>** **new** Vue({ el: '#app', data: { num: 0 }， methods：{ *//事件方法* plus(i){ **this**.num+=i; } } })**</script>**

**事件对象**

事件处理的过程中会用到事件对象，事件处理过程中将会自动触发一个event事件对象，那么这个event 对象代表当前事件的状态；比如事件触发时的HTML元素是什么，event.target来得到触发事件的HTML元素；可以得到事件的类型，event.type得到当前的事件类型，是鼠标事件或者是键盘事件，如果是鼠标事件可以的到鼠标的位置，通过event的X和Y得到鼠标的位置，如果是键盘事件可以得到键盘的按键状态，得到当前按得是哪个按键等等信息。事件对象的属性介绍如下表所示。

|  |  |
| --- | --- |
| 属性和方法 | 描述 |
| event.type | 触发的事件类型 |
| event.target | 触发事件的HTML元素 |
| event.preventDefault( ) | 阻止事件的默认行为 |
| event.stopPropagation( ) | 阻止事件冒泡 |

**事件委派**

借助事件对象实现一个事件委托机制，从而提高我们应用程序的一个性能。对于上一章节事件对象最后存在问题，可以通过事件委派来解决。

什么是事件委派机制？事件委派机制在列表渲染过程中，大部分都是在v-for渲染过程中，借助 event 事件对象，在父元素之上绑定事件处理函数而不是在子元素之上绑定事件处理函数。针对上面的例子我们可以采用在table标签上面用传递事件对象，示例代码如下：

**<table** **@click**="editItem($event)"**>** **<tr>** **<th>**序号**</th>** **<th>**名称**</th>** **<th>**价格**</th>** **<th>**操作</th> **</tr>** **<tr** v-for="(goods,index) in goodslists"**>** **<td>**{{index+1}}**</td>** **<td>**{{goods.name}}**</td>** **<td>**{{goods.price}}**</td>** **<td><a** href="#"**>**编辑**</a></td>** **</tr>****</table>**

methods:{ editItem:**function**(event){ *//只有在<a>元素上单击，才触发单击事件* **if**(event.target.nodeName.toLowerCase()=='a'){ *//触发事件的HTML元素* console.log(event.target); } }}

访问原生 DOM 事件对象，可以用特殊变量 **$event** 把它传入方法：

*//同时传递参数 和 事件对象，$event放在最后*<button type="button" @click="btn\_fn(10,$event)">click</button>methods:{ *//事件函数如果不传参数，默认就有一个参数--ev* btn\_fn(i,ev){ console.log(i,ev); *//同时定义 参数 和 事件对象* }}

**表单处理**

在Vue框架中实现表单的双向绑定非常简单，直接使用v-model指令进行关联即可。v-model指令会根据控件类型自动选取正确的方法来更新元素。

|  |  |
| --- | --- |
| 指令 | 说明 |
| v-model | 双向数据绑定<br />支持标签：<input> <textarea> <select> |

*<!--text-->***<input** type="text" name="username" v-model="form.username"**>** *<!--radio-->***<input** type="radio" name="sex" value="m" v-model="form.sex" **>**男**<input** type="radio" name="sex" value="w" v-model="form.sex"**>**女 *<!--select-->***<select** name="sel" v-model="form.city"**>** **<option** value="1"**>**北京</option> **<option** value="2"**>**上海</option> **<option** value="3"**>**广州**</option>****</select>** *<!--checkbox-->***<input** type="checkbox" name="hobby[]" value="1" v-model="form.hobby"**/>**足球**<input** type="checkbox" name="hobby[]" value="2" v-model="form.hobby"**/>**篮球**<input** type="checkbox" name="hobby[]" value="3" v-model="form.hobby"**/>**游戏**<input** type="checkbox" name="hobby[]" value="4" v-model="form.hobby"**/>**读书data: { //表单的值类型必须是String类型 form:{ username:"zhangsan", sex:"m", city:"2", hobby:["1","3"] }},

表单事件处理就是当表单元素发生变化时，会触发相应事件，可以为表单元素绑定事件处理函数。常用的表单事件处理函数如下表所示。

|  |  |  |
| --- | --- | --- |
| 表单事件 | 触发控件 | 说明 |
| focus | input框、textarea框 | 输入框获取到输入焦点 |
| blur | input框、textarea框 | 输入框失去焦点 |
| click | 单选框、多选框 | 选中某一个单选项或复选框 |
| change | 下拉列表 | 切换下拉列表项 |
| submit | 提交按钮 | 单击提交按钮 |

# 

# 第二章 Vue组件

## 第一节 组件概述

组件（Component）是Vue.js最推崇，也是最强大的功能之一，核心目标是为了可重用性高，减少重复性的开发。将结构上（HTML）相近或相同的代码进行封装，成为一个高度可复用的部件称为组件。

Vue.js组件可以理解为预先定义好行为的ViewModel类。一个组件可以预定义很多选项，但最核心的是以下几个：

|  |  |
| --- | --- |
| 选项 | 说明 |
| 模板（template） | 模板声明了数据和最终展现给用户的DOM之间的映射关系。 |
| 初始数据（data） | 一个组件的初始数据状态。对于可以复用的组件来说，通常是私有的状态。 |
| 接受的外部参数（props） | 组件之间通过参数来进行数据的传递和共享。参数默认是单向绑定（由下往上），但也可以显示声明为双向绑定。 |
| 方法（methods） | 对数据的改动操作一般都在组件的方法内进行。可以通过v-on指令将用户输入事件和组件方法进行绑定。 |
| 生命周期钩子函数(lifecycle hooks） | 一个组件会触发多个生命周期钩子函数，比如created、attached、destroyed等。在这些钩子函数中，我们可以封装一些自定义的逻辑。和传统的MVC相比，这可以理解为Controller的逻辑被分散到了这些钩子函数中。 |

Vue组件的使用原则遵循"先注册后使用"。Vue组件既可以全局注册，也可以局部注册。

**创建组件**

因为组件是可复用的 Vue 实例，所以它们与 *new Vue* 接收相同的选项，例如 *data* *computed* *watch* *methods* 以及生命周期钩子等。（ *el* 是根实例特有选项）。

**let** myTemp={ template:'#temp', *//模板id* data: **function**(){ *//必须为函数(闭包)* **return** { *//必须有return，返回值为对象{}* title:"hxsd" } }};

**注册组件**

*//在components配置项中注册组件***let** app=**new** Vue({ el:"#box", components：{myTemp}}); *//全局注册组件，还可以使用Vue.component方法（仅限全局注册，不建议使用）*Vue.component('myTemp'，MyTemp）；

**使用组件**

*<!--在Vue实例中使用组件-->***<div** id='box'**>** *<!--组件名如果用驼峰定义，改为短横线命名-->* **<my-temp></my-temp>****</div>**

**注意事项**

模板：使用**id**绑定

命名：组件名**不要是html标准标签名**，如果用**驼峰**定义，html中**引用组件时**，改为**短横线**命名

**组件对象剖析**

使用**<template>**标签定义模板，<template>是html5新增了一个标签，就是用来声明”模板元素“，该元素默认就是（display:none）隐藏元素，并且可以在任何位置定义。

**<template** id="temp"**>** *<!--组件的内容需要包含在一个容器内-->* **<div>** **<h3>**{{title}}**</h3>** **</div>****</template>**

**注意：**

1. 将模板定义在Vue实例挂载元素（el绑定id的元素）的外面
2. 遵守html语法规定的标签嵌套规范。比如：*<ul>* *<ol>* *<table>* *<select>* 这样的元素里允许包含的元素有限制，而另一些像 *<option>* 这样的元素只能出现在某些特定元素的内部。

**Vue组件的data属性**

data属性表示Vue组件可用的响应式数据，是Vue 实例的数据对象。Vue 将会递归将 data 的属性转换为getter/setter，从而让 data 的属性能够响应数据变化。对象必须是纯粹的对象 (含有零个或多个的 key/value jian键值对)：浏览器API创建的原生对象，原型上的属性会被忽略。大概来说，data应该只能是数据。

一旦观察过，不需要再次在数据对象上添加响应式属性。因此推荐在创建实例之前，就声明所有的根级响应式属性。

实例创建之后，可以通过 vm.

data.a。

以 \_ 或

data.property 的方式访问这些属性。

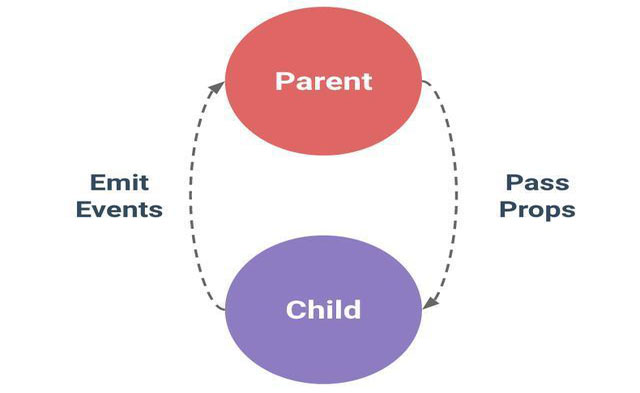
**Props传递数据**

父元素与组件之间数据通信，我们使用的是props属性，"props"是组件数据的一个字段，期望从父组件传下来数据。

## 第二节 事件通信

**父子通讯**

父子组件之间的数据操作，是通过**props**属性和**$emit()**方法来实现的。



**props**

当子组件需要读取父组件的数据（或者说，父组件要将数据传入子组件），子组件通过*props*自定义属性，来接收父组件的数据（或者说，父组件通过 props将数据下发到子组件），这是一个单向传递。

* 声明位置：props 属性**在子组件中声明**
* 属性值：**在组件模板中绑定**

**$emit**

子组件向父组件传值（或者说，子组件要修改父组件的数据），使用$emit触发父组件方法

声明位置：在子组件上声明**自定义事件**

*//定义组件***let** pagetmp=Vue.extend({ template:"#page",*//指定模板* props:["ap","p\_index"], *//ap:接收总页数 p\_index：接收分页索引* methods:{ paging(n){ **this**.$emit("p\_change",n);*//触发父级事件 参数1：自定义事件名称 参数2：传参* } }});

**注意：**

props属性名如果用**驼峰**定义，html标签中改为**短横线**命名

$emit自定义事件不能使用**驼峰**、**短横线**命名

二者均可以使用下划线命名

如果prop是一个对象或数组，子组件内部改变它会影响父组件的状态

**非父子通信**

在简单的场景下，可以使用一个空的 Vue 实例作为中央事件总线（event bus）。

本质上，是通过 **派发事件→监听事件** 从而更改值，（父子组件通信也可用这个方式，不同的是：父子组件通信不用一个空的Vue实例来做中转）

emit定义事件，可以使用驼峰命名

**<div** id="app"**>** *<!-- 不要使用<temp/>这样的单标签 -->* **<temp1></temp1>** **<temp2></temp2>****</div>****<script>** *//模板1* **let** temp1 = Vue.extend({ template:'<div><h1>{{msg}}</h1><button @click="run">修改</button></div>', data:()**=>**({msg:"hxsd"}), methods:{ run(){ **this**.msg=1234; bus.$emit('modify',**this**.msg);*//触发bus上的事件* } } }) *//模板2* **let** temp2 = Vue.extend({ template:'<div> <h1>{{txt}}</h1></div>', created(){ bus.$on('modify',val**=>**{ *//在bus实例上派发一个modify事件,并且开始监听* **this**.txt=val; }) }, data:()**=>**({txt:'6666'}), }) *//第三方实例* **let** bus=**new** Vue(); *//app实例* **let** app = **new** Vue({ el: '#app', components:{temp1, temp2} })**</script>**

**插槽**

为提高组件的灵活性，在定义组件时，不确定组件内容，在组件内预留一些slot，引用组件的时候传入内容

**<div** id="box"**>** *<!--引用组件-->* **<pc>** **<h2>**我的diy电脑**</h2>** **<p** slot="CPU"**>**intel i7**</p>** **<p** slot="memory"**>**16G**</p>** **<h4>**详细清单**</h4>** **</pc>** **<pc** v-for="item in list" :key="item.id"**>** *<!--没有对应name的内容，放在default slot中-->* **<h5>**{{title}}**</h5>** **<hr>** *<!--对应slot name 插入-->* **<p** v-for="(value,key) in item" :slot="key"**>**{{key}} ：{{value}}**</p>** **</pc>****</div>** *<!--模板-->***<template** id="pc"**>** **<div>** *<!--default slot（默认插槽，只能有一个）用于放置没有指定slot的元素-->* **<slot></slot>** *<!--具名插槽-->* **<slot** name="CPU"**></slot>** **<slot** name="memory"**></slot>** **<slot** name="SSD"**></slot>** **<slot** name="mouse"**></slot>** **</div>****</template>****<script>****let** pc=Vue.extend({ template:"#pc"， created(){ console.log(**this**.$slots)；*//查看插槽信息* }}) **let** app = **new** Vue({ el: '#box', data:{ title:"电脑配置单：", list:[ {id:1,"CPU":"intel i5","memory":"2G" }, {id:2,"CPU":"intel i7","memory":"8G","mouse":"蓝牙鼠标"}, {id:3,"CPU":"intel i3","memory":"8G","SSD":"256G"} ]  }, components:{pc},})**</script>**

## 

## 第三节 组件类型

**自定义组件v-model双向绑定**

组件中定义v-model。通过使用 v-bind:value 和 v-on:input 事件进行处理组件的数据。组件要接收名为 value 的prop数据，并使用v-bind:value绑定到表单元素上。示例代码如下：

<!DOCTYPE html>**<html>****<head>** **<meta** charset="UTF-8"**>** **<title>**Vue计算属性**</title>** **<script** type="text/javascript"**></script>****</head>****<body>****<div** id="app"**>** **<div>** **<h6>**v-model工作原理**</h6>** **<input** type="text" v-bind:value="name" v-on:input="name = $event.target.value"**>** **<div>**name = {{name}}**</div>** **</div>** **<div>** 父元素中使用组件的v-model： **<my-component** v-model="name"**></my-component>** **</div>****</div>****<template** id="my-component"**>** **<div>** 组件中的输入框： **<input** type="text" :value="value" **@input**="handleInput($event.target.value)"**>** **</div>****</template>****<script** src="vue.js"**></script>****<script>** *// 创建组件***const** MyComponent = { template: '#my-component', props: ['value'], methods: { handleInput: **function**(val) { *// 触发组件的自定义事件* **this**.$emit('input', val); } }}; *// 创建Vue实例对象***const** app = **new** Vue({ el: '#app', data: { name: '' }, components: {MyComponent}});**</script>****<body>****</html>**

**动态组件**

动态组件主要是指程序在使用过程中，可能需要动态的切换组件的显示内容，在这种场景下，我们需要使用到动态组件。

多个组件可以使用同一个挂载点，然后动态的在他们之间切换，使用保留的 *<component>* 标签添加动态组件， *<component>* 标签可以渲染其他任意组件。主要使用场景：程序中动态切换组件（选项卡、路由）。

**<div** id="app"**>** **<div>** **<button** **@click**="currentComponent='C1Component'"**>**c1-component**</button>** **<button** **@click**="currentComponent='C2Component'"**>**c2-component**</button>** **</div>** **<div>** *<!-- 使用动态组件 -->* **<keep-alive>** **<component** v-bind:is="currentComponent"**></component>** **</keep-alive>** **</div>****</div>****<template** id="c1"**>** **<div>** this is c1-component. **<div>**name: **<input** type="text"**></div>** **</div>****</template>****<template** id="c2"**>** **<div>**this is c2-component.**</div>****</template>****<script** src="vue.js"**></script>****<script>** *// 定义组件* **const** C1Component = { template: '#c1' }; **const** C2Component = { template: '#c2' }; *// 创建Vue对象* **const** app = **new** Vue({ el: '#app', data: { currentComponent: 'C1Component' }, components: {C1Component,C2Component} });**</script>**

**递归组件**

在组件内部可以嵌套调用其他组件，甚至可以递归地调用当前自身组件。

**<div** id="app"**>** **<menu-component** :menus="menus"**></menu-component>****</div>****<template** id="menu-component"**>** **<ul>** **<li** v-for="item in menus"**>** {{item.name}} *<!-- 递归调用本身 -->* **<menu-component** v-if="item.children" :menus="item.children"**></menu-component>** **</li>** **</ul>****</template>****<script** src="vue.js"**></script>****<script>** *// 创建组件* **const** MenuComponent = { name: 'MenuComponent', template: '#menu-component', props: ['menus'] }; *// 创建Vue对象* **const** app = **new** Vue({ el: '#app', data: { menus: [ {name: 'menu-1'}, { name: 'menu-2', children: [ {name: 'menu-2-1'}, { name: 'menu-2-2', children: [ {name: 'menu-2-2-1'}, {name: 'menu-2-2-2'} ] } ] } ] }, components: {MenuComponent} });**</script>**

# 第三章 Vue工程化工具

## 第一节 搭建Node.js环境

**Node.js简介**

Node.js 是一个让 JavaScript 运行在服务端的开发平台，它让 JavaScript 成为与PHP、Python、Perl、Ruby 等服务端语言平起平坐的脚本语言。 Node.js 发布于2009年5月，由Ryan Dahl开发，实质是对Chrome V8引擎进行了封装。 Node.js对一些特殊用例进行优化，提供替代的API，使得V8在非浏览器环境下运行得更好。V8引擎执行Javascript的速度非常快，性能非常好。Node.js是一个基于Chrome JavaScript运行时建立的平台，用于方便地搭建响应速度快、易于扩展的网络应用。Node.js 使用事件驱动，非阻塞I/O 模型而得以轻量和高效，非常适合在分布式设备上运行数据密集型的实时应用。 Node.js作为一个新兴的前端框架，后台语言，有很多吸引人的地方：

* RESTful API；
* 单线程
* Node.js可以在不新增额外线程的情况下，依然可以对任务进行并发处理--Node.js是单线程的。它通过事件循环（event loop）来实现并发操作，对此，我们应该要充分利用这一点 —— 尽可能的避免阻塞操作，取而代之，多使用非阻塞操作；
* 非阻塞I/O；
* V8虚拟机；
* 事件驱动。

Node.js使用Module模块去划分不同的功能，以简化应用的开发。每一个Node.js的类库都包含了十分丰富的各类函数，比如http模块就包含了和http功能相关的很多函数，可以帮助开发者很容易地对比如http,tcp/udp等进行操作，还可以很容易的创建http和tcp/udp的服务器。要在程序中使用模块是十分方便的，只需要如下：

**var** http = require('http');

在这里，引入了http类库，并且对http类库的引用存放在http变量中了。这个时候，Node.js会在我们应用中搜索是否存在node\_modules的目录，并且搜索这个目录中是否存在http的模块。如果Node.js找不到这个目录，则会到全局模块缓存中去寻找，用户可以通过相对或者绝对路径，指定模块的位置，比如：

**var** myModule = require('./myModule.js');

模块中包含了很多功能代码片断，在模块中的代码大部分都是私有的，意思是在模块中定义的函数方法和变量，都只能在同一个模块中被调用。当然，可以将某些方法和变量暴露到模块外，这个时候可以使用exports对象去实现。

**Node.js环境安装**

Node.js 安装包及源码下载地址为：[*https://nodejs.org/en/download/*](https://nodejs.org/en/download/) 。

**npm使用**

npm是随同Node.JS一起安装的包管理工具，能解决Node.JS代码部署上的很多问题，常见的使用场景有以下几种：

* 允许用户从npm服务器下载别人编写的第三方包到本地使用。
* 允许用户从npm服务器下载并安装别人编写的命令行程序到本地使用。
* 允许用户将自己编写的包或命令行程序上传到NPM服务器供别人使用。

可以通过输入 "npm -v" 来测试是否成功安装。

* 升级npm版本

npm install npm -g

* 使用淘宝镜像的命令：

npm install -g cnpm --registry=https://registry.npm.taobao.org

* 使用 npm 命令安装模块，npm 安装 Node.js 模块语法格式如下：

npm install <Module Name>

下面我们使用 npm 命令安装常用的 Node.js web框架模块 Express:

npm install express

安装好之后，Express 包就放在了工程目录下的 node\_modules 目录中，因此在代码中只需要通过 require('express') 的方式就好，无需指定第三方包路径。

var express = require('express');

npm 的包安装分为本地安装（local）、全局安装（global）两种，从敲的命令行来看，差别只是有没有-g而已，比如：

npm install express # 本地安装npm install express -g # 全局安装

如果出现以下错误：

npm err! Error: connect ECONNREFUSED 127.0.0.1:8087

解决办法为：

npm config set proxy null

本地安装：

将安装包放在 ./node*modules 下（运行 npm 命令时所在的目录），如果没有 node*modules 目录，会在当前执行 npm 命令的目录下生成 node\_modules 目录。

可以通过 require() 来引入本地安装的包。

全局安装：

将安装包放在 /usr/local 下或者 node 的安装目录。

可以直接在命令行里使用。

如果你希望具备两者功能，则需要在两个地方安装它或使用 npm link。接下来我们使用全局方式安装 Express

npm install express -g

* package.json简单介绍及使用
* package.json位于模块的目录下，用于定义包的属性。接下来让我们来看下Express 包的package.json 文件，位于 node\_modules/express/package.json，打开后显示内容如下所示：

{ "name": "npm", "version": "1.0.0", "main": "index.js", "dependencies": { "vue": "^2.6.10" }, "devDependencies": {}, "scripts": { "test": "echo \"Error: no test specified\" && exit 1" }, "author": "", "license": "ISC", "description": ""}

各项package.json属性解释如下表所示：

|  |  |
| --- | --- |
| 属性 | 说明 |
| name | 包名 |
| version | 包的版本号。 |
| description | 包的描述。 |
| homepage | 包的官网 url 。 |
| author | 包的作者姓名。 |
| contributors | 包的其他贡献者姓名。 |
| dependencies | 依赖包列表。如果依赖包没有安装，npm 会自动将依赖包安装在 node\_module 目录下。 |
| repository | 包代码存放的地方的类型，可以是 git 或 svn，git 可在 Github 上。 |
| main | 字段指定了程序的主入口文件，require('moduleName') 就会加载这个文件。这个字段的默认值是模块根目录下面的 index.js。 |
| kewords | 关键字 |

我们可以使用以下命令来**卸载** Node.js 模块

npm uninstall express

卸载后，你可以到 /node\_modules/ 目录下查看包是否还存在，或者使用**查看**命令：

npm ls

我们可以使用以下命令**更新**模块：

npm update express

使用以下来**搜索**模块：

npm search express

创建模块，package.json 文件是必不可少的。我们可以使用 NPM 生成 package.json 文件，生成的文件包含了基本的结果。

以上的信息，你需要根据你自己的情况输入。在最后输入 "yes" 后会生成 package.json 文件。接下来我们可以使用以下命令在 npm 资源库中注册用户（使用邮箱注册）：

$ npm adduser

Username: mcmohd

Password:

Email: (this IS public) mcmohd@gmail.com

最后用以下命令来发布模块：

npm publish

如果你以上的步骤都操作正确，你就可以跟其他模块一样使用 npm 来安装。

版本号定义规范：使用NPM下载和发布代码时都会接触到版本号。NPM使用语义版本号来管理代码，这里简单介绍一下。语义版本号分为X.Y.Z三位，分别代表主版本号、次版本号和补丁版本号。当代码变更时，版本号按以下原则更新：

如果只是修复bug，需要更新Z位。

如果是新增了功能，但是向下兼容，需要更新Y位。

如果有大变动，向下不兼容，需要更新X位。

版本号有了这个保证后，在申明第三方包依赖时，除了可依赖于一个固定版本号外，还可依赖于某个范围的版本号。例如"argv": "0.0.x"表示依赖于0.0.x系列的最新版argv。

NPM支持的所有版本号范围指定方式都可以查看[*官方文档*](https://npmjs.org/doc/files/package.json.html#dependencies)。

除了本章介绍的部分外，NPM还提供了很多功能，package.json里也有很多其它有用的字段。

除了可以在[*npmjs.org/doc/*](https://npmjs.org/doc/)查看官方文档外，这里再介绍一些NPM常用命令。NPM提供了很多命令，例如install和publish，使用npm help可查看所有命令。

|  |  |
| --- | --- |
| 命令 | 说明 |
| npm help | 可查看某条命令的详细帮助，例如npm help install。 |
| npm install <package>e -g | 在本地安装当前命令行程序，可用于发布前的本地测试。 |
| npm update <package> -g | 使用可以把全局安装的对应命令行程序更新至最新版。 |
| npm update <package> | 可以把当前node\_modules子目录里对应模块更新至最新版本 |
| npm cache clear | 可以清空NPM本地缓存<br />用于对付使用相同版本号发布新版本代码的人 |
| npm unpublish <package>@<version> | 可以撤销发布自己发布过的某个版本代码。 |

国内直接使用 npm 的官方镜像是非常慢的，这里推荐使用淘宝 npm镜像。淘宝 npm镜像是一个完整 npmjs.org 镜像，你可以用此代替官方版本(只读)，同步频率目前为 10分钟 一次以保证尽量与官方服务同步。你可以使用淘宝定制的 cnpm (gzip 压缩支持) 命令行工具代替默认的 npm：

npm install -g cnpm --registry=https://registry.npm.taobao.org

这样就可以使用 cnpm 命令来安装模块了：

cnpm install [name]

**yarn使用**

Yarn 是 Facebook, Google, Exponent 和 Tilde 开发的一款新的 JavaScript 包管理工具。它的目的是解决这些团队使用 npm 面临的少数问题，即：

* 安装的时候无法保证速度/一致性；
* 安全问题，因为 npm 安装时允许运行代码。

对比npm，yarn的优点主要体现以下几个方面：

* npm安装是串行，而yarn是并行，速度大大提升；
* 已经下载过的包会被缓存，无需重复下载,更关键的是，支持离线安装；
* 精准的版本控制，加上验证每个包的完整性，保证每次安装的npm包完全一致；
* 安全：下载前会检查签名及包的完整性；
* 网络优化：力求网络资源最大利用化，让资源下载完美队列执行，避免大量的无用请求，下载失败会自动重新请求，避免整个安装过程失败。

但它并没有试图完全取代 npm。Yarn 同样是一个从 npm 注册源获取模块的新的 CLI 客户端。注册的方式不会有任何变化，你同样可以正常获取与发布包。

进入[*官方*](file:///C:\Users\cjq\Documents\WeChat%20Files\cuixiaoming183015\FileStorage\File\2019-07\www.yarnpkg.com\zh-Hans\docs\install#windows-stable)下载页面进行安装，安装方法：直接双击安装即可。安装成功后，可以打开命令行，使用yarn -v查看当前yarn的版本。

yarn同样可以配置淘宝镜像，配置方法如下：

yarn config set registry https://registry.npm.taobao.org -g

Yarn常用的命令：

|  |  |
| --- | --- |
| 指令 | 说明 |
| yarn init | 初始化项目，生成package.json文件 |
| yarn add 包名 | 添加依赖包，示例：yarn add [package]、yarn add [package]@[version]、yarn add [package]@[tag] |
| yarn | 根据 package.json 安装全部依赖包，也可以使用 yarn install |
| yarn upgrade | 升级依赖包 |
| yarn remove | 移除依赖包 |

Yarn 是为了弥补 npm 的一些缺陷而出现的，Yarn 没想要完全替代 npm，它只是一个新的CLI 工具，拉取的 packages 依然来自npm仓库，所以yarn的命令与npm相似，但又有区别。yarn和npm命令对比，如下表所示：

|  |  |  |
| --- | --- | --- |
| Npm | Yarn | 说明 |
| npm init | yarn init | 初始化某个项目 |
| npm install/link | yarn install/link | 默认的安装依赖操作 |
| npm install taco -save | yarn add taco | 安装某个依赖，并默认保存到package |
| npm uninstall taco -save | yarn remove taco | 移除某个依赖项目 |
| npm install taco –save-dev | yarn add taco –dev | 安装某个开发时依赖项目 |
| npm updata taco -save | yarn upgrade taco | 更新某个依赖项目 |
| npm install taco --global | yarn global add taco | 安装某个全局依赖项目 |
| npm publish/login/logout | yarn publish/login/logout | 发布、登录、登出，一系列NPM Registry操作 |
| npm run test | yarn run test | 运行某个命令，可以在script脚本中去配置 |

## 

## 第二节 使用Vue CLI

**Vue CLI简介**

Vue CLI 是一个基于 Vue.js 进行快速开发的完整系统，提供：

* 通过 vue/cli 搭建交互式的项目脚手架；
* 通过 vue/cli + vue/cli-service-global 快速开始零配置原型开发；
* 一个运行时依赖 (vue/cli-service)，该依赖优点：
  + 可升级；
  + 基于webpack 构建，并带有合理的默认配置；
  + 可以通过项目内的配置文件进行配置；
  + 可以通过插件进行扩展。
* 一个丰富的官方插件集合，集成了前端生态中最好的工具；
* 一套完全图形化的创建和管理 Vue.js 项目的用户界面。

Vue CLI 需要[*Node.js*](https://nodejs.org/) 8.9 或更高版本，可以使用下列任一命令安装这个新的包：

npm install -g @vue/cli

yarn global add @vue/cli

**Vue CLI使用**

使用Vue CLI创建Vue项目非常简单，通过vue create 指令即可。当输入vue create时，Vue CLI交互式地提示输入项目的相关信息，不同版本的Vue CLI创建Vue项目的提示信息可能有差别，创建项目时根据不同Vue CLI版本信息，相应选择适合的信息选项。大致操作步骤有如下：

Vue项目创建成功后，项目目录结构如下表：

|  |  |
| --- | --- |
| 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 | 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： ² assets: 放置一些图片，如logo等 ² components: 目录里面放了一个组件文件，可以不用 ² 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 ² main.js: 项目的核心文件 |
| test | 初始测试目录，可删除 |
| static | 静态资源目录，如图片、字体等。 |
| 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 |

此时项目就已经全部搭建好并且运行了。进入到项目目录，若使用 npm方式，执行 npm run serve命令。

在浏览器中输入：[*http://localhost:8080*](http://localhost:8080/) Vuevue ui通过可视化网页形式创建应用程序。

## 第三节 Vue项目配置

使用 vue config 命令来审查或修改全局的 CLI 配置。本节主要介绍如何使用vue.config.js审查或修改全局的CLI配置。vue.config.js是一个可选的配置文件，如果项目的 (和package.json 同级的) 根目录中存在这个文件，那么它会被 vue/cli-service自动加载。

|  |  |  |  |
| --- | --- | --- | --- |
| 配置项 | 数据类型 | 说明 | 默认值 |
| publicPath | string | 基本路径 | '/' |
| outputDir | string | 输出文件目录 | 'dist' |
| assetsDir | string | 静态资源存放目录 | ' ' |
| indexPath | string | 输出路径 | 'index.html' |
| filenameHashing | Boolean | 生成的静态资源在它们的文件名中包含了hash 以便更好的控制缓存 | true |
| pages | Object | 在 multi-page 模式下构建应用。每个“page”应该有一个对应的 JavaScript 入口文件。 | undefined |
| lintOnSave | Boolean | 是否在开发环境下通过 [*eslint-loader*](https://github.com/webpack-contrib/eslint-loader) 在每次保存时 lint 代码。 | true |
| runtimeCompiler | Boolean | 是否使用包含运行时编译器的 Vue 构建版本 | false |
| transpileDependencies | Boolean | 是否生成source map文件，可以将其设置为 false 以加速生产环境构建。 | true |
| crossorigin | String | 设置生成的 HTML 中<link rel="stylesheet">和<script> 标签的 crossorigin 属性（跨域）。 | undefined |
| integrity | Boolean | 在生成的 HTML 中的 <link rel="stylesheet"> 和 <script> 标签上启用 Subresource Integrity(SRI：摘要签名机制，来保证外链资源的完整性)。 | flase |
| configureWebpack | Object/function | Object:通过 webpack-merge合并到最终的配置中<br />function:接收被解析的配置作为参数 |  |
| chainWebpack | Function | 用来表示是一个函数，会接收一个基于<br/>webpack-chain的 ChainableConfig 实例。允许对内部的 webpack 配置进行更细粒度的修改。 |  |
| css.modules | Boolean | 默认情况下，只有 *.module.[ext] 结尾的文件才会被视作 CSS Modules 模块。设置为true后你就可以去掉文件名中的.module并将所有\*.(css|scss|less|styel(us)?)文件视为CSS Modules文件。 | false |
| css.extract | Boolean/Object | 用来设置是否将组件中的 CSS 提取至一个独立的 CSS 文件中 (而不是动态注入到 JavaScript 中的 inline 代码)。 | 生产环境:true<br />开发环境:fals。 |
| css.sourceMap | Boolean | 是否为 CSS 开启 source map。 | false |
| css.loaderOptios | Object | 用来向 CSS 相关的 loader 传递选项 | {} |
| devServer | Object | 所有 webpack-dev-server 的选项都支持 |  |
| devServer.proxy | String/Object | 前端应用和后端 API 服务器没有运行在同一个主机上，需要在开发环境下将 API 请求代理到 API 服务器。 | proxy: '' |
| parallel | Boolean | 是否为Babel或 TypeScript 使用 thread-loader。该选项在系统的 CPU 有多于一个内核时自动启用，仅用于生产构建。 | require('os').cpus().length > 1 |
| pluginOptions | Object | 用来传递任何第三方插件选项。 |  |

## 

## 第四节 js模块

JS中容易出现变量被覆盖，方法被替代的情况（即被污染）。特别是存在依赖关系时，容易出现错误。这是因为JS缺少模块管理机制，来隔离实现各种不同功能的JS判断，避免它们相互污染。

模块主流实现方式有：

**CommonJS模块**

是node环境中采用的JS模块机制，具体使用介绍如下：

* 对于基本数据类型，属于复制。即会被模块缓存。同时，在另一个模块可以对该模块输出的变量重新赋值。
* 对于复杂数据类型，属于浅拷贝。由于两个模块引用的对象指向同一个内存空间，因此对该模块的值做修改时会影响另一个模块。
* 当使用require命令加载某个模块时，就会运行整个模块的代码。
* 当使用require命令加载同一个模块时，不会再执行该模块，而是取到缓存之中的值。也就是说，CommonJS模块无论加载多少次，都只会在第一次加载时运行一次，以后再加载，就返回第一次运行的结果，除非手动清除系统缓存
* 循环加载时，属于加载时执行。即脚本代码在require的时候，就会全部执行。一旦出现某个模块被"循环加载"，就只输出已经执行的部分，还未执行的部分不会输出。

*//a.js------------------------------------***var** txt="hello world";**let** run= **function**() { console.log(txt);}; *//输出对象*module.exports.hello=run;

*//b.js-------------------------------------***var** a=require("./a.js");*//引入a.js 注意：引入本地文件，必须用相对路径*a.hello();*//调用a模块中的hello方法*

**ES6模块**

面向未来的JS模块机制，模块标准。具体使用介绍如下：

* ES6模块中的值属于动态只读引用。
* 对于只读来说，即不允许修改引入变量的值，import的变量是只读的，不论是基本数据类型还是复杂数据类型。当模块遇到import命令时，就会生成一个只读引用。等到脚本真正执行时，再根据这个只读引用，到被加载的那个模块里面去取值。
* 对于动态来说，原始值发生变化，import加载的值也会发生变化。不论是基本数据类型还是复杂数据类型。
* 循环加载时，ES6模块是动态引用。只要两个模块之间存在某个引用，代码就能够执行

ES6移除了关于模块如何加载/执行的内容，只保留了输出、引入模块的语法。

每一个js文件都是一个模块，使用 **export** 或 **export default** 输出。

*//a.js 方式一 分别输出每一个，可以是变量 函数 对象 类*export **var** a=10;export **function** run(){ console.log("a模块的run函数");};

*//b.js 方式二 export default 对象方式输出全部***var** a = 1;**var** school = {name:'hxsd', age:20};**function** run(){console.log("b模块的run函数")};export default {a, school, run}; *//default 表示不给导出的对象起名字，名字交给引入方（引入时也无需查看名称），并且可以选择性导出*

使用**import(路径名)**引入模块，一定要使用 "/" "./" "../" 路径

*//在花括号中指明需使用的API，并且可以用as指定别名*import {a} from "./a.js";import {run as go} from './a.js';console.log("a模块的变量a："+a);go(); *//可以输入b.js中用export default定义的对象*import b from "./b.js";b.run(); *//使用通配符\*, 重新导出（转载）其他模块的接口*export \* from "./a.js";

## 第五节 vue-devtools工作

基于chrome游览器的插件，用于调试Vue应用，极大地提高调试Vue程序的效率，chrome浏览器商店安装：在中国境内不可用。

Vue-devtools的使用原则有如下几点：

* 必须在 http:// 协议中使用，所以必须打开开发者服务器
* 在Chrome的开发者工具中，选择 Vue 面板
* 显示当前组件的详细信息（层次关系、数据、事件等等）

# 第四章　 Express服务器开发

## 第一节 概述

简单来说， Node.js 就是运行在服务端（后端）的 JavaScript。

JavaScript语言的设计初衷是为了操作客户端（前端）页面的DOM元素，进行页面交互，对于语言本身来说，并不存在前后之分，如果操作的不是DOM对象，而是数据库对象（增删改查），是不是就变成后端语言了？

后端主要的工作就是响应用户请求，根据请求提供相应服务，包括：

文件操作、数据存取（操作数据库）、平台的稳定性、优化性能 等。

后端的主要开发程序有：PHP、JAVA 、ASP.net、python、ruby等等。这些程序的运行，都需要在服务器上部署不同的语言环境。

**第一个node应用程序**

*// 引入node模块***const** http = require("http"); *// 创建http服务器***const** server = http.createServer(**function**(request, response) { *// 当http服务器接收到http请求时，自动调用该回调函数* *// request表示请求对象* *//response表示响应对象* *// 处理请求和响应* **let** url = request.url; *//当前请求url信息* *//生成响应* response.setHeader('Content-Type', 'text/html'); *//响应消息头* response.write('hello nodejs'); *//生成响应消息主体内容* response.end(); *// 发送响应消息*}); *//启动http服务器  
// 给出启动端口号*server.listen(8080, **function**() { *// 回调函数：主要用来输出提示消息clg信息* *//nodejs程序中使用clg输出的提示消息，都是在命令行中查看* console.log('server is running on http://127.0.0.1:8080');});

## 第二节 Express

**创建Express 应用程序**

Express 是一个简洁而灵活的 node.js Web应用框架, 提供了一系列强大特性，帮助你创建各种 Web 应用和丰富的 HTTP 工具。使用 Express 可以快速地搭建一个完整功能的网站。

Express 框架核心特性：

* 可以设置中间件来响应 HTTP 请求。
* 定义了路由表用于执行不同的 HTTP 请求动作。
* 可以通过向模板传递参数来动态渲染 HTML

Express安装可以通过下面指令进行安装，并将其保存到依赖列表中：

npm install express --save

以下几个重要的模块是需要与 express 框架一起安装的：

* body-parser : node.js中间件，用于处理 JSON, Raw, Text 和 URL 编码的数据。
* cookie-parser : 是一个解析Cookie的工具。通过req.cookies可以取到传过来的cookie，并把它们转成对象。
* multer : node.js 中间件，用于处理 enctype="multipart/form-data"（设置表单的MIME编码）的表单数据。

npm install body-parser --savenpm install cookie-parser --savenpm install multer --save

Express安装成功后，我们可以创建第一个Express 框架实例

*// 引入node模块***const** express = require('express'); *// 创建expres程序***const** app = express(); *// 添加HTTP路由*app.get('/', **function**(request, response) { *//输出响应消息* response.send('hello express');}); *// 启动http服务器*app.listen(8080, **function**() { console.log('express app is running on http://127.0.0.1:8080');});

上面的示例代码中，Express 应用使用回调函数的参数值：request 和 response 对象来处理请求和响应的数据。

request 对象表示 HTTP 请求，包含了请求查询字符串，参数，内容，HTTP 头部等属性。常见属性如下表所示：

|  |  |
| --- | --- |
| 属性 | 说明 |
| req.app | callback为外部文件时，用req.app访问Express的实例 |
| req.baseUrl | 获取路由当前安装的URL路径 |
| req.body / req.cookies | 获得「请求主体」/ Cookies |
| req.fresh / req.stale | 判断请求是否还「新鲜」 |
| req.hostname / req.ip | 获取主机名和IP地址 |
| req.originalUrl | 获取原始请求URL |
| req.params | 获取路由的parameters |
| req.path | 获取请求路径 |
| req.protocol | 获取协议类型 |
| req.query | 获取URL的查询参数串 |
| req.route | 获取当前匹配的路由 |
| req.subdomains | 获取子域名 |
| req.accepts() | 检查可接受的请求的文档类型 |
| req.acceptsCharsets / req.acceptsEncodings / req.acceptsLanguages | 返回指定字符集的第一个可接受字符编码 |
| req.get() | 获取指定的HTTP请求头 |
| req.is() | 判断请求头Content-Type的MIME类型 |

response 对象表示 HTTP 响应，即在接收到请求时向客户端发送的 HTTP 响应数据。常见属性如下表所示：

|  |  |
| --- | --- |
| 属性 | 说明 |
| res.app | 同req.app一样 |
| res.append() | 追加指定HTTP头 |
| res.set() | 在res.append()后将重置之前设置的头 |
| res.cookie(name，value [，option]) | 设置Cookie |
| opition | domain / expires / httpOnly / maxAge / path / secure / signed |
| res.clearCookie() | 清除Cookie |
| res.download() | 传送指定路径的文件 |
| res.get() | 返回指定的HTTP头 |
| res.json() | 传送JSON响应 |
| res.jsonp() | 传送JSONP响应 |
| res.location() | 只设置响应的Location HTTP头，不设置状态码 或者close response |
| res.redirect() | 设置响应的Location HTTP头，并且设置状态码302 |
| res.render(view,[locals],callback) | 渲染一个view，同时向callback传递渲染后的字符串，如果在渲染过程中有错误发生next(err)将会被自动调用。callback将会被传入一个可能发生的错误以及渲染后的页面，这样就不会自动输出了。 |
| res.send() | 传送HTTP响应 |
| res.sendFile(path [，options] [，fn]) | 传送指定路径的文件 -会自动根据文件extension设定Content-Type |
| res.status() | 设置HTTP状态码 |
| res.type() | 设置Content-Type的MIME类型 |

**Express路由**

Express路由是指如何定义应用的端点（URIs）以及如何响应客户端的请求。

*// 引入node模块***const** express = require('express'); *// 创建expres程序***const** app = express(); *// 添加HTTP路由*app.get('/', **function**(request, response) { *//输出响应消息* response.send('hello express');});app.get('/users', **function**(req, res) { *//req 表示请求对象* *//res表示响应对象* res.send('users-page');}); *// 启动http服务器*app.listen(8080, **function**() { console.log('express app is running on http://127.0.0.1:8080');});

应用程序根据不同的HTTP请求方法和不同的HTTP请求URL，确定如何响应请求的过程。常用的HTTP请求方法如下表所示：

|  |  |
| --- | --- |
| 请求方法 | 说明 |
| GET | 请求一个指定资源的表示形式. 使用GET的请求应该只被用于获取数据. |
| HEAD | 请求一个与GET请求的响应相同的响应，但没有响应体. |
| POST | 用于将实体提交到指定的资源，通常导致状态或服务器上的副作用的更改. |
| PUT | 用请求有效载荷替换目标资源的所有当前表示。 |
| DELETE | 删除指定的资源。 |
| CONNECT | 建立一个到由目标资源标识的服务器的隧道。 |
| OPTIONS | 用于描述目标资源的通信选项。 |
| TRACE | 沿着到目标资源的路径执行一个消息环回测试。 |
| PATCH | 用于对资源应用部分修改。 |

常用路由路径如下表所示：

* 使用字符串的路由路径

*// 匹配 acd 和 abcd*app.get('/ab?cd', function(req, res) { res.send('ab?cd');});

* 使用字符串模式的路由路径

*// 匹配 acd 和 abcd*app.get('/ab?cd', function(req, res) { res.send('ab?cd');}); *// 匹配 /abe 和 /abcde*app.get('/ab(cd)?e', function(req, res) { res.send('ab(cd)?e');});

* 使用正则表达式的路由路径

*// 匹配 butterfly、dragonfly，不匹配 butterflyman、dragonfly man等*app.get(/.\*fly$/, function(req, res) { res.send('/.\*fly$/');});

路由句柄可以为请求处理提供多个回调函数，其行为类似中间件。唯一的区别是这些回调函数有可能调用 *next('route')* 方法而略过其他路由回调函数。可以利用该机制为路由定义前提条件，如果在现有路径上继续执行没有意义，则可将控制权交给剩下的路径。路由句柄有多种形式，可以是一个函数、一个函数数组，或者是两者混合。如下面实例：我们使用函数数组的形式，对根目录'/'的请求，提供四个回掉函数：

**let** cb1 = **function** (req, res, next) { console.log ('1'); next ();};**let** cb2 = **function** (req, res, next) { console.log ('2'); next ();};**let** cb3 = **function** (req, res, next) { console.log (3); next ();};**let** cb4 = **function** (req, res, next) { console.log (4); res.send ('Hello from 4');};app.get ('/', [cb1, cb2]);app.get ('/', [cb3, cb4]); *// 输出为1,2,3,4*

路由参数：app.param。即我们可以在路径中添加会变动的参数，例如我们可以将一个用户id放在url地址上，并通过req.params拿到这个用户id。示例代码如下：

*// 引入node模块***const** express = require('express'); *// 创建expres程序***const** app = express(); *// 添加HTTP路由*app.get('/', **function**(request, response) { *//输出响应消息* response.send('hello express');});app.get('/users', **function**(req, res) { *//req 表示请求对象* *// res表示响应对象* res.send('users-page');}); *// 动态路由*app.get('/users/id', **function**(req, res) { *// 接收id值* **let** id = req.params.id; *//获取动态参数* *// 返回响应* res.send('id = ' + id);}); *// 启动http服务器*app.listen(8080, **function**() { console.log('express app is running on http://127.0.0.1:8080');});

当请求的地址为localhost:8080/users/3时，可以通过req.params.id拿到当前用的id值。

使用app.param对路径参数做处理，可以理解为处理参数的一个中间件。下面对id这个参数进行的处理。示例代码如下：

*// 引入node模块***const** express = require('express'); *// 创建expres程序***const** app = express(); *// 添加HTTP路由*app.get('/', **function**(request, response) { *//输出响应消息* response.send('hello express');});app.get('/users', **function**(req, res) { *//req 表示请求对象* *// res表示响应对象* res.send('users-page');});app.param('id', (req, res, next) **=>** { console.log(req.params.id); **if** (req.params.id === '1') { next(); } **else** { res.sendStatus(404); }});app.get('/users/:id', (req, res) **=>** { res.send(`The current user ID is:${req.params.id}`);}); *// 启动http服务器*app.listen(8080, **function**() { console.log('express app is running on http://127.0.0.1:8080');});

**pug视图模板**

pug就是之前的jade，也就是一种通过缩进的方式来编写代码的过程，在编译的过程中，不需要考虑标签是否闭合的问题。此外，用这种编译方式，加快了我们写代码的速度，也为代码复用提供了便捷。

通过npm指令即可安装

npm install pug

Pug 的渲染操作一般来说是相当简单的。 pug.compile() 会把 Pug 代码编译成一个 JavaScript 函数，并且这个函数有一个参数可用于传入数据。

app.js中代码：

*// 引入node模块***const** express = require('express'); *// 创建expres程序***const** app = express(); *// 配置视图模板*app.set('view engine', 'pug');app.set('views', './views'); *//设置模板文件的路径  
  
// 添加HTTP路由*app.get('/', **function**(request, response) { *//输出响应消息* *//加载并解析index.pug文件，写入到响应消息主体中* response.render('index.pug');});app.get('/users', **function**(req, res) { *//req 表示请求对象* *// res表示响应对象* *//加载pug模板，并传递模板数据* res.render('users.pug', { title: 'users-title', users: [ { id: 1, name: '张三', age: 18 }, { id: 2, name: '李四', age: 20 } ] });}); *// 启动http服务器*app.listen(8080, **function**() { console.log('express app is running on http://127.0.0.1:8080');});

users.pug文件中代码：

doctype htmlhtml head meta(charset="utf-8") title #{title} body #app for user in users div p id = #{user.id} p name = #{user.name} p age = #{user.age} hr

Pug的常用属性介绍如下表所示：

|  |  |
| --- | --- |
| 属性 | 说明 |
| 属性 Attribute | 标签属性和 HTML 语法非常相似，但它们的值就是普通的 JavaScript 表达式。您可以用逗号作为属性分隔符，不过不加逗号也是允许的。 |
| 多行属性 | 如果您有很多属性，您可以把它们分几行写，如果您有一个很长属性，并且JavaScript 运行时引擎支持 ECMAScript 2015 [*模板字符串*](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Template_strings)（包括 Node.js 和 io.js v1.0.0 和更新的版本），您可以使用它来写属性值 |
| 用引号括起来的属性 | 如果您的属性名称中含有某些奇怪的字符，并且可能会与 JavaScript 语法产生冲突的话，请您将它们使用 "" 或者 '' 括起来。您还可以使用逗号来分割不同的属性。这种属性的例子有 Angular 2 中经常用到的 [] 和 () |
| 属性嵌入 | 如果您想要在您的属性当中使用变量的话，您可以这样做：直接使用 JavaScript 写那个属性；如果您的 JavaScript 运行时支持 ECMAScript 2015 [*模板字符串*](https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Template_strings)(包含在 Node.js/io.js 1.0.0 以及更新的版本当中)，那么您还可以使用这种方式来简化属性值。 |
| 不转义属性 | 在默认情况下，所有的属性都经过转义（即把特殊字符转换成转义序列）来防止诸如跨站脚本攻击之类的攻击方式。如果您非要使用特殊符号，您需要使用 != 而不是 =。未经转义的缓存代码十分危险。您必须正确处理和过滤用户的输入来避免[*跨站脚本攻击*](https://en.wikipedia.org/wiki/Cross-site_scripting)。 |
| 布尔值属性 | 在 Pug 中，布尔值属性是经过映射的，这样布尔值（true 和 false）就能接受了。当没有指定值的时候，默认是 true。 |
| 样式属性 | style（样式）属性可以是一个字符串（就像其他普通的属性一样）还可以是一个对象，这在部分样式是由 JavaScript 生成的情况下非常方便 |
| 类属性 | class（类）属性可以是一个字符串（就像其他普通的属性一样）还可以是一个包含多个类名的数组，这在类是由 JavaScript 生成的情况下非常方便 |
| 类的字面值 | 类可以使用 .classname 语法来定义 |
| ID的字面值 | ID 可以使用 #idname 语法来定义 |
| &attributes | &attributes 语法可以将一个对象转化为一个元素的属性列表。使用 &attributes 赋值的属性并不会经过自动转义过程。您必须正确处理和过滤用户的输入来避免[*跨站脚本攻击*](https://en.wikipedia.org/wiki/Cross-site_scripting)(XSS)。 但是如果您是从一个 mixin 调用中使用 attributes 的话，这个过程就是自动的 |

## 

## 第三节 请求响应对象

**处理GET请求和POST请求**

GET请求的参数在URL中，在原生Node中，需要使用url模块来识别参数字符串。在Express中，不需要使用url模块了，可以直接使用req.query对象。示例代码如下：

*// 引入node模块***const** express = require('express'); *// 创建expres程序***const** app = express(); *// 配置视图模板*app.set('view engine', 'pug'); *// 添加HTTP路由*app.get('/', **function**(req, res) { *//输出响应消息* req.send('hello express')});app.get('/users/:gender', **function**(req, res) { *//获取路由参数* **let** gender = req.params.gender; *//获取url参数* **let** name = req.query.name; *//输出响应* res.send('gender = ' + gender + ', name = ' + name);}); *// 启动http服务器*app.listen(8080, **function**() { console.log('express app is running on http://127.0.0.1:8080');});

POST请求在Express中不能直接获得，必须使用body-parser模块。使用后，将可以用req.body得到参数。但是如果表单中含有文件上传，那么还是需要使用formidable模块。

post请求要借助body-parser模块，借用req.body函数得到参数。使用body-parser模板引擎，处理表单提交。

form(action="/create", method="post") p label 用户名： input(type="text", name="name") p label 密码: input(type="password", name="password") p input(type="submit")

接下来就使用POST方式请求参数，引入 Express，body-parse。在这里必须要有ejs这个模块，示例代码如下：

*// 引入node模块***const** express = require('express');**const** bodyParser = require('body-parser'); *// 创建expres程序***const** app = express(); *// 配置视图模板*app.set('view engine', 'pug'); *//配置POSt消息主体解析方法*app.use(bodyParser.urlencoded({ extended: **true**}));app.get('/create', **function**(req, res) { res.render('create.pug');});app.post('/create', **function**(req, res) { *// 获取用户输出表单信息* **let** name = req.body.name; **let** pswd = req.body.password; res.send('name = ' + name + ", pswd = " + pswd);}); *// 启动http服务器*app.listen(8080, **function**() { console.log('express app is running on http://127.0.0.1:8080');});

**文件上传**

Express框架使用multer中间件实现上传文件功能:

upload.pug文件代码：

form(action="/upload", method="post", enctype="multipart/form-data") p label 用户名： input(type="text", name="name") p label 照片: input(type="file", name="photo") p input(type="submit")

server.js 文件代码：

*// 引入node模块***const** express = require('express');**const** bodyParser = require('body-parser');**const** multer = require('multer'); *// 创建expres程序***const** app = express(); *// 配置视图模板*app.set('view engine', 'pug'); *//配置POSt消息主体解析方法*app.use(bodyParser.urlencoded({ extended: **true**})); *//配置文件上传信息***const** upload = multer({ *// 文件上传路径* dest: 'uploads/'});app.get('/create', **function**(req, res) { res.render('create.pug');});app.post('/create', **function**(req, res) { *// 获取用户输出表单信息* **let** name = req.body.name; **let** pswd = req.body.password; res.send('name = ' + name + ", pswd = " + pswd);});app.get('/upload', **function**(req, res) { res.render('upload.pug');});app.post('/upload', upload.single('photo'), **function**(req, res) { *// 获取文件上传信息* **let** file = req.file; res.send(file);}); *// 启动http服务器*app.listen(8080, **function**() { console.log('express app is running on http://127.0.0.1:8080');});

## 第四节 Session使用

**Session概述**

在WEB开发中，服务器可以为每个用户浏览器创建一个会话对象（session对象）。在需要保存用户数据时，服务器程序可以把用户数据写到用户浏览器独占的session中，当用户使用浏览器访问其它程序时，其它程序可以从用户的session中取出该用户的数据，为用户服务。

Session和Cookie的主要区别如下表所示：

|  |  |
| --- | --- |
| Cookie | Session |
| Cookie是把用户的数据写给用户的浏览器，在浏览器端保存状态数据，每次访问服务器时，在HTTP请求消息中传输状态数据。缺点是：数据量太大，HTTP消息传输负担较大；用户可能会修改cookie信息，导致服务器不安全。 | Session技术把用户的数据写到用户独占的session中，不同用户通过不同的sessionid 辨别。 <br />1.把 sessionid 保存在 客户端cookie或本地存储中。 <br /> 2.浏览器发送HTTP请求时，附带 sessionid 的cookie信息，以便服务器端区分是哪一个用户。 Session技术把用户的数据写到用户独占的session中，不同用户通过不同的sessionid 辨别。 |

**Session使用**

Session的使用流程主要包括以下几个方面：

1. 安装express-session及session持久化中间件：

npm install express-session

1. 引入express-session：

var session = require("express-session");

1. 设置官方文档提供的中间件：

app.use(session({ secret: 'keyboard cat', resave: true, saveUninitialized: true}))

1. 使用：

req.session.username = "张三"; *//设置值*req.session.username *//获取值*

express-session的常用参数介绍：

|  |  |
| --- | --- |
| 参数 | 作用 |
| Secret | 一个String类型的字符串，作为服务器端创建session的签名 |
| Name | 返回客户端key的名称，默认为connect.sid，也可以自己设置 |
| Resave | 强制保存session，即使它没有变化。默认为true。建议设置成false。 |
| SaveUninitialized | 强制将未初始化的session存储。当新建了一个session且未设定属性或值时，它就处于未初始化状态。在设定一个cookie前，这对于登录验证，减轻服务的存储压力，极限控制是有帮助的（默认为true）。建议手动添加。 |
| Cookie | 设置返回到前端key的属性，默认值为{path: '/', httpOnly: true, secure: false, maxAge: null}。 |
| Rolling | 在每次请求时强行设置cookie，这将重置cookie过期时间，默认是false。 |

## 

## 第五节 Express中使用RESTful API

**构建RESTful API**

REST（Representational StateTransfer）即**表述性状态传递**，是Roy Fielding博士在2000年他的博士论文中提出来的一种软件架构风格。

表述性状态转移是一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是RESTful。需要注意的是，REST是设计风格而不是标准。REST通常基于使用HTTP，URI，和XML（标准通用标记语言下的一个子集）以及HTML（标准通用标记语言下的一个应用）这些现有的广泛流行的协议和标准。REST 通常使用 JSON 数据格式。

REST 基本架构的四个方法：

* GET - 用于获取数据。
* PUT - 用于更新或添加数据。
* DELETE - 用于删除数据。
* POST - 用于添加数据

数据

{ "user1" : { "name" : "mahesh", "password" : "password1", "profession" : "teacher", "id": 1 }, "user2" : { "name" : "suresh", "password" : "password2", "profession" : "librarian", "id": 2 }, "user3" : { "name" : "ramesh", "password" : "password3", "profession" : "clerk", "id": 3 }}

创建 RESTful API **listUsers**，用于读取用户的信息列表， server.js 文件代码如下所示：

**var** express = require('express');**var** app = express();**var** fs = require("fs");app.get('/listUsers', **function** (req, res) { fs.readFile( \_\_dirname + "/" + "users.json", 'utf8', **function** (err, data) { console.log( data ); res.end( data ); });})**var** server = app.listen(8081, **function** () { **var** host = server.address().address **var** port = server.address().port console.log("应用实例，访问地址为 http://%s:%s", host, port)})

创建 RESTful API **addUser**， 用于添加新的用户数据，server.js 文件代码如下所示：

**var** express = require('express');**var** app = express();**var** fs = require("fs"); *//添加的新用户数据***var** user = { "user4" : { "name" : "mohit", "password" : "password4", "profession" : "teacher", "id": 4 }}app.get('/addUser', **function** (req, res) { *// 读取已存在的数据* fs.readFile( \_\_dirname + "/" + "users.json", 'utf8', **function** (err, data) { data = JSON.parse( data ); data["user4"] = user["user4"]; console.log( data ); res.end( JSON.stringify(data)); });})**var** server = app.listen(8081, **function** () { **var** host = server.address().address **var** port = server.address().port console.log("应用实例，访问地址为 http://%s:%s", host, port)})

**postman工具**

Postman是一款功能超级强大的用于发送 HTTP 请求的Chrome插件。做web页面开发和测试的人员应该是无人不晓无人不用！其主要特点：

* 创建 + 测试：创建和发送任何的HTTP请求，请求可以保存到历史中再次执行
* Organize:使用Postman Collections为更有效的测试及集成工作流管理和组织APIs
* document:依据你创建的Clollections自动生成API文档,并将其发布成规范的格式
* collarorate:通过同步连接你的team和你的api，以及权限控制，API库

## 第六节 规划目录结构

Vue.js的目录结构，一个简单的Vue.js目录结构解析：

|  |  |
| --- | --- |
| 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 | 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： 1.sets: 放置一些图片，如logo等。 2.mponents: 目录里面放了一个组件文件，可以不用。 3.p.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。 4.in.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。 |

## 

## 第七节 Nodemon热刷新

Nodemon用来监视node.js应用程序中的任何更改并自动重启服务，非常适合用在开发环境中。Nodemon将监视启动目录中的文件，如果有任何文件更改，Nodemon将自动重新启动node应用程序。

Nodemon不需要对代码或开发方式进行任何更改。 Nodemon只是简单的包装你的node应用程序，并监控任何已经改变的文件。Nodemon只是node的替换包，只是在运行脚本时将其替换命令行上的node。

安装

npm install -g nodemonnpm install --save-dev nodemon

Nodemon常用指令说明如下表所示：

|  |  |
| --- | --- |
| 指令 | 说明 |
| nodemon -h或者 nodemon --help | 使用帮助 |
| nodemon -e | 默认的 nodemon监视 .js, .mjs, .coffee, litcoffee和Json文件，通过-e命令你可以指定你自己的查找列表 |
| nodemon --watch path | nodemon默认只会监视当前的工作路径，如果你想去监视其他路径上的文件，你可以使用如下命令： nodemon --watch app --watch libs app/server.js |
| nodemon --ignore | 忽视一些文件被监视 |
| nodemon --delay | 有时候你会修改许多文件，这时为了避免不必要的重启，你可以通过命令指定多少时间后再进行重启。 nodemon --delay 10 server.js |
| rs | 手动重启 |
| nodemon ./server.js localhost 8080 | 指定主机和端口 |
| nodemon --debug ./server.js 80 | 开启debug模式 |

# 

# 第五章 axios 网络交互

## 第一节 axios库的概述

[*axios*](https://github.com/axios/axios)（艾克瑟斯）是当前比较流行的，采用promise封装的Ajax异步请求工具（非jquery插件）。

在Vue程序应用中，向服务器端发送ajax请求同时获取服务器端响应的http请求响应库。axios库的主要优点是轻量级JS库，可以单独使用；支持PromiseAPI，解决JS"回调函数地狱"问题；支持发送cookie、HTTP认证、并发请求、请求和响应的拦截、取消请求等；自动转换JSON数据，特别适用于RESTful API 场景。

## 第二节 axios库的使用

{ *//url----------------------------------------------------------------* *//请求的服务器地址* url:'/user', *//method--------------------------------------------------------------* *//求资源的方式* method:'get', *//default* *//baseURL-------------------------------------------------------------* *//如果 url 不是绝对地址，那么 baseURL 将会加到 url 的前面* *//当 url 是相对地址的时候，设置`baseURL`会非常的方便* baseURL:'https://some-domain.com/api/', *//transformRequest-----------------------------------------------------* *//允许在请求发送到服务器之前对请求的数据做出一些改动,* *//该选项只适用于以下请求方式：put/post/patch* *//数组里面的最后一个函数必须返回一个字符串、-一个 ArrayBuffer 或者 Stream* transformRequest:[**function**(data){ *//根据需求改变数据* **return** data; }], *//transformResponse-----------------------------------------------------* *//允许在数据传送到 then/catch 方法之前对数据进行改动* transformResponse:[**function**(data){ *//根据需求改变数据* **return** data; }], *//headers----------------------------------------------------------------* *//自定义http请求头信息* headers: {'X-Requested-With':'XMLHttpRequest'}, *//params-----------------------------------------------------------------* *//要随请求一起发送的query参数,他的类型必须是一个纯对象或者是URLSearchParams对象* params: {ID:12345}, *//paramsSerializer-------------------------------------------------------* *//一个可选的函数，起作用是让参数（params）序列化* paramsSerializer: **function**(params){ **return** Qs.stringify(params,{arrayFormat:'brackets'}) }, *//data-------------------------------------------------------* *//作为一个请求体而需要被发送的数据* *//该选项只适用于方法：put/post/patch* *//当没有设置transformRequest选项时dada必须是以下几种类型之一：* *//string/plain/object/ArrayBuffer/ArrayBufferView/URLSearchParams* *//仅浏览器：FormData/File/Blob* *//仅node:Stream* data {firstName:"Fred"}, *//timeout----------------------------------------------------------* *//定义了请求发出的延迟毫秒数,如果请求超过延迟时间，请求会被终止* timeout:1000, *//withCredentails-------------------------------------------------------* *//表明是否是跨域请求* withCredentials:**false**,*//default* *//adapter----------------------------------------------------------------* *//适配器选项允许自定义处理请求，这会使得测试变得方便* *//返回一个promise,并提供验证返回* adapter: **function**(config){...}, *//auth------------------------------------------------------------------* *//表明HTTP基础的认证应该被使用，并提供证书* *//这会设置一个authorization头（header）,并覆盖你在header设置的Authorization头信息* auth: { username:"zhangsan", password: "s00sdkf" }, *//responseType---------------------------------------------------------* *//返回数据的格式,可选项是arraybuffer,blob,document,json,text,stream* responseType:'json',*//default* *//xsrfCookieName---------------------------------------------------------* *//是用作XSRF（跨站点请求伪造）令牌值的cookie的名称* xsrfCookieName: 'XSRF-TOKEN', *//default* *//xsrfHeaderName---------------------------------------------------------* *//是承载XSRF令牌值的HTTP头的名称* xsrfHeaderName: 'X-XSRF-TOKEN', *//default* *//onUploadProgress------------------------------------------------------* *//上传进度事件* onUploadProgress:**function**(progressEvent){...}, *//onDownloadProgress----------------------------------------------------* *//下载进度的事件* onDownloadProgress:**function**(progressEvent){...}, *//maxContentLength------------------------------------------------------* *//响应内容的最大值* maxContentLength:2000, *//validateStatus--------------------------------------------------------* *//定义了是否根据http相应状态码，来resolve或者reject promise* *//如果`validateStatus`返回true(或者设置为`null`或者`undefined`)* *//那么promise的状态将会是resolved,否则其状态就是rejected* validateStatus:**function**(status){ **return** status >= 200 && status <300;*//default* }, *//maxRedirects---------------------------------------------------------* *//定义了在nodejs中重定向的最大数量* maxRedirects: 5,*//default* *//httpAgent/httpsAgent-------------------------------------------------* *//定义了当发送http/https请求要用到的自定义代理，keeyAlive默认false* httpAgent: **new** http.Agent({keeyAlive:**true**}), httpsAgent: **new** https.Agent({keeyAlive:**true**}), *//proxy----------------------------------------------------------------* *//定义了主机名字和端口号，auth`表明http基本认证应该与proxy代理链接，并提供证书* *//这将会设置一个 Proxy-Authorization header,* *//并且会覆盖掉已经存在的 Proxy-Authorization header* proxy: { host:'127.0.0.1', port: 9000, auth: { username:'skda', password:'radsd' } }, *//cancelToken----------------------------------------------------------* *//定义了一个用于取消请求的cancel token* cancelToken: **new** cancelToken(**function**(cancel){...})}

调用axios，接收一个配置项参数（object）：

*// 发送一个 POST 请求*axios({ method: 'post', url: '/user/12345', data: { firstName: 'Fred', lastName: 'Flintstone' }});

为了方便起见，已经为所有支持的请求方法提供了别名：

* axios.get（url [，config]）
* axios.post（url [，data [，config]]）
* axios.request（config）
* axios.head（url [，config]）
* axios.delete（url [，config]）
* axios.put（url [，data [，config]]）
* axios.patch（url [，data [，config]]）

## 第三节 发送get请求

*// 发送一个 get 请求*axios.get（"/get\_data?name=zhangsan"）.then( res**=>**{ console.log(res.data); }, ()**=>**{ console.log("err"); })

## 第四节 发送post请求

*// 发送一个 post 请求*axios.post（"/post\_data",params:{name:"zhangsan",pwd:123456}）.then( res**=>**{ console.log(res.data); }, ()**=>**{ console.log("err"); })

## 第五节 Vue中使用axios

import './plugins/axios'; *// ajax网络请求获取数据* created: **function**() { **const** app = **this**; axios.get('http://127.0.0.1/users').then(res **=>** { app.users = res.data.data; }); }

## 第六节 axios跨域问题

在Vue中有两种方法来解决跨域问题，第一种方法是在前端，Vue框架中来配置跨域访问，第二种方法是通过服务器端，在node程序中修改来使它实现跨域访问。

* 在Vue框架中的vue.config.js配置文件中，配置代理服务器。vue.config.js文件：

module.exports = { devServer: { *// 若遇到未知请求，将会交给代理服务器处理* proxy: 'http://127.0.0.1' }}; *// ajax网络请求获取数据*created: **function**() { **const** app = **this**; axios.get('/users').then(res **=>** { app.users = res.data.data; });}

* 服务器端支持跨域访问

Express中开启CORS（跨域资源共享）支持，在这种情况下不使用代理服务器也可以实现获取跨域的数据。

首先安装cors跨域支持模块，

yarn add cors

constcors= require('cors');app.use(cors())，

# 第六章 Vue-router

## 第一节 SPA和Vue路由

是针对**SPA**（single page web application）单页面应用，根据URL地址中不同的 *#* 值，表示不同的访问状态，通过Ajax异步获取数据，然后进行页面渲染，用户的感觉象真的更换了页面一样。

SPA的好处是：

* 良好的交互体验：页面只是局部内容被刷新，不需要重新刷新整个页面，页面显示流畅。
* 前后端工作分离模式：服务器只提供数据，不用管展示逻辑和页面合成，减轻服务器压力。

vue-router是基于 Vue CLI 应用程序使用，安装非常简单，直接通过命令：vue add router安装即可。安装完成后选择 URL路由模式：

history模式：类似于普通的url，但是页面刷新时可能会有问题。

hash模式：使用 "#"号来分隔URL，这种比较丑陋。

## 第二节 路由使用

**定义路由信息**

定义路由主要是用routes，routes中是路由信息对象组成的数组，定义路由包括路由路径（path）、路由名称（name）、路由组件对象（component）。路由定义示例代码如下：

routes: [ { path: '/', *// 路由路径：说明当前路由匹配的路径是什么* name: 'home', *// 路由名称* component: Home *// 路由组件对象（说明当前的路由会交给单文件组件来处理）* }, { path: '/users', name: 'Users', component: UserComponent }, { path: '/about', name: 'about',  component: () **=>** import( './views/About.vue') } ]

**动态路由**

静态路由是路由路径和定义时路由信息保持完全一致。动态路由是指可以在路由定义中传入一些路由参数，这个路由参数可以在组件中通过props数据来获取。

routes: [ { path: '/users/:username/post/:postid', name: 'Users', component: UserComponent, } ]

下面对模式，匹配的路径和通过$route.params获取的信息进行举例说明，具体如表格所示。

|  |  |  |
| --- | --- | --- |
| 模式 | 匹配路径 | $route.params |
| /user/:username | /user/tom | {username:'tom'} |
| /user/:username/post/:postId | /user/tom/post/3 | {username:'tom',postId:'3'} |

**路由参数**

在路由参数中常用props传递路由参数。

routes: { path: '/users/:id', name: 'Users', component: UserComponent, *// 把当前路由参数作为props数据传递给路由组件* *// 降低了路由组件和当前路由定义的耦合性* props:**true** } ] ]

UserComponent.vue文件：

<template> <div class="user"> this is user-component id = {{id}} </div></template><script>export default { name: 'UserComponent', props: ['id']}</script>

**路由组件中的Ajax**

主要介绍动态路由和ajax请求之间的关系，介绍动态路由如何使用ajax,动态路由中传递不同的路由参数，根据路由参数向服务器端请求Ajax数据，显示不同的内容。示例代码如下：

module.exports = { devServer: { *// 代理服务器* proxy: 'http://127.0.0.1' }};

使用 axios 请求Ajax数据，axios请求方法，可以添加到路由组件的created( ) 生命周期回调方法中。示例代码如下：

created() { console.log('created被调用 id = ' + **this**.id); **const** app = **this**; *// 发送ajax请求，id值来自于当前的props中的id，传递给路由组件中，然后捕获响应，获取到响应以后，修改user的信息为当前的参数。* axios.get('/users/' + **this**.id).then(res **=>** { app.user = res.data.data; });}

## 第三节 路由导航

**路由导航守卫**

当动态路由发生切换时，可以使用路由导航守卫处理路由的变化。什么是路由导航守卫可以简单理解为路由组件的生命周期回调函数。

*// 路由导航守卫  
// 作用：在第一次进入当前路由组件之前被调用  
// 使用场景：获取ajax数据*beforeRouteEnter(to, from, next) { *// to：表示要进入的路由组件* *// from：表示将要离开的路由组件* *// next：表示后续操作函数* *// 此时还未进入到组件中，故不能使用this获取当前组件的实例* next(**function**(app) { *// 进入到当前组件后，才执行的回调* *// 此时回调参数 app 表示当前组件的实例对象* axios.get('/users/' + app.id).then(res **=>** { app.user = res.data.data; }); });}

beforeRouteUpdate(to, from, next) { *// 此时，可以使用this表示当前组件对象* **const** app = **this**; *// 发送ajax请求* *// this表示切换前的状态* *// to表示要切换到的路由对象 route* axios.get('/users/' + to.params.id).then(res **=>** { app.user = res.data.data; }); *// 执行后续* next();}

**编程式路由导航**

编程式路由导航也是路由的重要部分，之前实现路由跳转有两种方式，其一是在浏览器中输入地址实现路由跳转，其二是使用<router-link> 触发路由跳转。

methods: { login(){ **if**(登陆成功){ *//实现页面跳转* **this**.$router.push('/'); } }}

* push( )
* 跳转到指定的路由地址，并把当前地址写入到history中，参数可以是字符串路径或描述地址信息的对象

|  |  |
| --- | --- |
| 参数类型 | 参数 |
| 字符串 | router.push('home') |
| 对象 | router.push({path:'home'}) |
| 命名的路由 | router.push({name:user,params:{userId:1}}) |

* replace( )：跳转到指定路由，它不会向 history 添加新记录，而是替换掉当前的 history 记录。
* go( n )：前进或后退指定步
  + router.go(1）：前进一步，等同于 router.forward( )
  + router.go(-1) ：后退一步，等同于 router.back( )
  + router.go(0) ：刷新当前页面全局路由导航守卫

**全局路由导航守卫**

全局路由导航守卫可以应用于用户登陆校验，默认情况下显示的是首页，但是只有用户登陆以后才能进入首页，才能查看后台管理页面，否则用户没有处于登陆状态，需要跳转到登陆页面，针对这个问题使用全局路由导航守卫。全局路由导航守卫介绍如下：

*// 全局路由导航守卫* router.beforeEach((to, from, next) **=>** { });

## 

## 第四节 嵌套路由和命名路由视图

**嵌套路由**

是指在路由的定义中包含子路由的定义。实际使用中，很可能会需要在路由中嵌套子路由，指定路由 /user/:id ，可以在该路由组件内容渲染其它路由组件（如 /user/:id/posts 或 /user/:id/cart ）。

children: [ { path: "", component: 路由名 }, { path: "路径名", component: 路由名 }]

**命名视图**

使用<router-view> 可以使用 name属性为其设置名称，即命名路由的视图简称命名视图。若未添加name属性，则默认名称为default。

*<!-- router-view：路由页面显示的区域 -->* **<router-view/>** *<!--about -->* **<router-view** name="content"**></router-view>** *<!-- UserComponent -->*

import About from './views/About.vue';routes: [  { path: "/about", name: "about", components: { default: About, content: UserComponent } } ]

在上面实例中，About组件是默认的渲染视图，UserComponent组件是渲染的name为content的视图。

# 第七章 Vuex 状态管理

## 第一节 状态管理

**Vue组件共享数据的方式**

Vue组件共享数据的集中方式是在单页面应用程序中，路由和组件关联，采用单文件组件组织Vue程序。这种模式的优缺点如下：

优点：提高模块的独立性，便于代码重用和维护。

缺点：组件与组件（即不同的路由页面）之间必然需要进行数据通信，Vue实现组件间共享数据比较繁琐。

组件间共享数据的方式，主要有父子组件之间共享数据、非父子组件之间共享数据、以及借助本地存储来实现，下面进行介绍。

* 父子组件间

父组件向子组件传递props数据，子组件向父组件发送$emit自定义事件，告诉父组件更新数据。这种方式是目前最常用的方式，但是存在着一些缺点:

父子组件之间传递数据，只能对于父子组件有效，对于非父子组件间，兄弟组件之间是无效的。

状态数据更新位置不统一，不利于代码维护，换句话说，这种形式把状态数据和组件进行了强耦合，不利于后续代码的维护。

* 非父子组件间

非父子组件间使用全局事件总线形式。定义一个全局的bus事件总线，其实就是一个Vue对象，然后向单个组件发送事件，而某个组件监听事件消息，从而更新自己的数据消息。这种方式的缺点主要表现在：

操作繁琐，需要在每个组件的生命周期回调方法中监听事件

状态数据更新零散，不利于代码维护。

* 借助本地存储（如local Storage或sessionStorage）实现数据更新。

缺点：

操作繁琐，需要在每个组件的生命周期回调方法中获取数据：在本地方法的create获取本地存储的数据。

不能实现数据响应式更新：如用户未登陆时会存在登陆的链接，用户登陆之后会产生注销链接。

状态数据管理零散，不利于代码维护：可以在任意组件更新localstorage和sessionStorage，从而影响状态数据的改变。

**状态管理库Vuex**

Vue状态管理，Vue官方推荐使用Vuex进行状态管理，实现组件间数据共享。它的工作原理是Vuex把组件间的所有状态数据集中到一处进行统一管理，并提供统一的操作接口进行状态更新。

组件与组件之间需要共享数据，把所有的数据集中到Vuex中管理，在Vuex中提供统一的接口。比如提供state接口方便组件读取数据，提供actions和mutations方便组件提交数据或者更新数据。状态管理库Vuex中使用state存储状态数据，使用mutations更新数据。

Vuex安装

安装Vuex使用的命令为：

vue add vuex

## 第二节 Vuex Store对象

export default **new** Vuex.Store({ state: { message: 'this is store.state message' }, mutations: { }, actions: { }})

**全局注册store**

在根实例中注册store，该store实例会注入到所有子组件中，子组件能通过 *this.$store* 访问到。

**let** app=**new** Vue({ el:"#app", store,});

**获取store数据**

**let** page1=Vue.extend({ template:"#page1", created(){ console.log(store===**this**.$store); *//true* console.log(**this**.$store.state.num); *//获取store上的state数据* },});

**State**

在Vuex.Store对象中，使用 state属性存储状态数据。在组件中获取state信息时，可以使用 this.$store.state.\*\*\* 形式获取状态数据。state数据建议在组件的 computed 属性中使用，以实现响应式更新。

**getters**

如果多个组件都需要对store的数据进行computed，可以在getters中定义（不需要各组件分别computed），每个组件都可以读取和引用。

**<div** id="box"**>** **<router-link** to="/man1"**>**学生A**</router-link>** **<router-link** to="/man2"**>**学生B**</router-link>** **<router-view></router-view>****</div>**

**let** store=**new** Vuex.Store({ state:{num1:10,num2:20}, *//getters是store中的计算属性（computed）* getters:{ sum(state){ **return** state.num1+state.num2;*//返回相加结果* } }});**let** man1=Vue.extend({ template:"<div><h1>zhangsan</h1><h2>getter：{{num}}</h2></div>", data:()**=>**({num:0}), created(){ *//创建实例时，读取store内的num* **this**.num=**this**.$store.getters.sum; }})**let** man2=Vue.extend({ template:"<div><h1>lisi</h1><h2>getters：{{sum}}</h2></div>", computed:{ *//也可以使用cumputed接收getters* sum:**function**(){ **return** **this**.$store.getters.sum; } }})

**mutations**

更改store中state数据的唯一方法是提交 **mutation**。（引用数据类型赋值为其他对象，只是内存地址引用，会造成数据混乱）。

使用**store.commit**方法（commit 译：承诺、委托），调用mutations中的方法：

*//参数1：string 方法名  
//参数2：any 传递数据*store.commit('SET\_NUM',"123"); *//或者 this.$store.commit*

**actions**

actions可以将mutations里面的方法，变成可异步的处理数据的方法。

例如：在Ajax请求数据后，根据返回结果，再操作store中的数据。如果多个组件都需要这样操作，那每个组件都要调用Ajax，此时，就可以通过在actions定义方法调用Ajax，在Ajax的回调函数中直接commit mutations中的方法，也就是所谓：将mutations里面的方法，变成可异步处理数据的方法。

**<div** id="box"**>** **<router-link** to="/man1"**>**学生A**</router-link>** **<router-link** to="/man2"**>**学生B**</router-link>** **<router-view></router-view>****</div>**

*//vuex状态***let** store=**new** Vuex.Store({ state:{ age:0}, mutations:{ SET\_AGE(state,payload){ state.age=payload; }, }, actions:{ *//异步方法* get\_age(context,name){ *//context:代表store本身* axios.get("doAction.php?name="+name).then( (res)**=>**{ context.commit('SET\_AGE',res.data); }, ()**=>**{ console.log("err"); } ) } }});**let** man1 = Vue.extend({ template: "<div><h1>{{username}}</h1><h2>年龄：{{my\_age}}</h2></div>", data: () **=>** ({username: "zhangsan"}), created() { *//使用dispatch派发action中的方法* store.dispatch("get\_age",**this**.username); }, computed:{ *//用computed接收store数据* my\_age(){ **return** **this**.$store.state.age; } }})**let** man2 = Vue.extend({ template: "<div><h1>{{username}}</h1><h2>年龄：{{my\_age}}</h2></div>", data: () **=>** ({username: "lisi"}), created() { store.dispatch("get\_age",**this**.username); }, computed:{ my\_age(){ **return** **this**.$store.state.age; } }})

## 

## 第三节 Vuex多模块

多模块机制主要是当应用程序变大的时候，我们可以采取多模块的形式来组织状态数据。在大型Vuex应用程序中，可以使用modules 定义多模块的状态数据。定义多模块的状态数据非常简单，只需要定义模块对象，然后在Vuex.Store对象中引入modules即可，主要的步骤如下：

* 定义多模块对象：多模块对象的本质就是 Vuex.Store对象，包含自己的状态数据state、mutations操作、actions 异步操作等。
* 在组件中使用多模块，使用模块状态数据（state）：使用 $store.state.模块名.\*\*\*形式。

## 第四节 Vuex插件使用

Vuex插件使用方法很简单，使用 npm 或 yarn 安装 vuex-persistedstate。安装好插件之后，具体的使用步骤如下所示：

* 在store对象创建之前首先引入createPersistedState。

import createPersistedState from "vuex-persistedstate";

* 在 Store 对象中 使用 plugins。示例代码如下：

plugins: [ createPersistedState({ *//采用sessionStorage来保存我们的状态信息* storage: window.sessionStorage })]

# 第八章 Vue ui

## 第一节 Vue ui库

**Vue ui库概述**

Vue的成功很大一部分原因在于它的组件。与组件一起工作意味着您可以孤立地考虑每个UI片断，鼓励重用性和模块化，并帮助保持UI一致性。所有Vue组件也就是Vue的实例，然后接受同样的选择对象，提供相同的生命周期。

下面介绍一些非常受欢迎的基于Vue生态下的ui框架：

|  |  |  |  |
| --- | --- | --- | --- |
| UI框架 | 开发团队 | github star数 | 优缺点 |
| element ui（桌面版ui） | 饿了么团队开源的ui库 | 34K | 组件丰富，但是界面样式不太美观。 [*http://element-cn.eleme.io/#/zh-CN/component/installation*](http://element-cn.eleme.io/) |
| iView（桌面版ui） | 个人团队开源的ui库 | 19K | 组件丰富，界面样式美观，但使用 render 函数渲染，使用成本较高。 [*https://www.iviewui.com/docs/guide/install*](https://www.iviewui.com/docs/guide/install) |
| Vux（移动端ui） | 基于微信ui推出的移动端ui库 | 14.9K | [*https://doc.vux.li/zh-CN/*](https://doc.vux.li/zh-CN/) |
| mint-ui（移动端ui） | 饿了么团队推出的移动端ui库 | 13.2K | [*http://mint-ui.github.io/docs/#/zh-cn2*](http://mint-ui.github.io/docs/) |

**安装**

npm i element-ui –S

Element还为新版的 Vue CLI准备了相应的 Element 插件，我们可以用它们快速地搭建一个基于 Element 的项目。Element同时提供了通用的项目模板，我们可以直接使用。对于 Laravel 用户，Element也准备了相应的模板，同样可以直接下载使用。我们可以引入整个 Element，或是根据需要仅引入部分组件。下面介绍如何完整引入整个Element：

在 main.js 中写入以下内容：import Vue from 'vue';import ElementUI from 'element-ui';import 'element-ui/lib/theme-chalk/index.css';import App from './App.vue';Vue.use(ElementUI);**new** Vue({ el: '#app',});

## 

## 第二节 Element组件

Element ui的组件资源非常丰富，基本上涵盖了桌面ui开发所需的全部组件。官网有详细说明。

[*Element-UI*](http://element-cn.eleme.io/#/zh-CN/component/installation)，手机移动端有对应框架是 [*Mint UI*](https://mint-ui.github.io/#!/zh-cn) 。

## 第三节 第三方扩展库Express

**集成Express和Vue**

日常开发中往往是项目开始时是前后端分开开发的，但是当后台做好一部分要进行连接测试时问题就来了。前端部分在每次修改过就需要build一次，而后台也每次需要重启一次服务器才能进行调试。可想而知，这样的开发和调试效率是很低的。所以我们需要搭建一个Vue+Express前后端分离的开发环境，Express服务器端只提供 API接口，Vue客户端使用ajax请求服务器数据，express和Vue部署到同一个项目目录结构中。

Express和Vue部署到同一个项目目录结构中，我们必须重新规划项目目录结构，创建的Vue程序位于 src/ 目录下（默认）， 而Express程序位于 application/目录下，以达到前后端分离的开发环境。

**配置Express和Vue**

第一步 配置Express后端环境，使用 Express 提供的脚手架工具（Express application generator）搭建出最基本的应用。

第二步 配置Vue环境，同样地，推荐使用 Vue 自带的template 工具，生成基本代码。

第三步 配置webpack，我们计划让 Webpack 将前端文件打包成一个 build.js 文件，然后放在 /src/server/public/javascripts 中供 jade 模板使用。所以设置好 webpack 的路径部分，其他的保留原来的就好。

第四步 配置Nodemon，用Nodemon启动server并监听代码变动。而 Nodemon 默认会监听除了 .git 和 node\_modules 路径外的所有 js 代码，因为我们已经有 Webpack 监听前端代码了，所以必须设置相关配置项让Nodemon只监听某一块代码。在项目根路径下新增文件 nodemon.json，

第五步 配置package.json，在 script 中增加一个命令，用来启动整个应用

这样，只需要运行 npm run dev 这一个命令就可以启动server 同时进行开发了。到此我们就配置完成了一个Vue+Express前后端分离的项目开发环境。

**第三篇 移动web设计与开发**

# 第一章 多媒体与绘图

**px、em、rem、vh、 vw和% 移动端长度单位**

使用CSS单位px、em、rem、vh、 vw等实现页面布局。

- px：绝对单位，页面按精确像素展示

- em：相对单位，基准点为父节点字体的大小，如果自身定义了font-size按自身来计算（浏览器默认字体是16px），整个页面内1em不是一个固定的值。em会根据父级元素的大小而变化，但是如果嵌套了多个元素，要计算它的大小，是很容易出错的，这样就引申出了rem。

- rem：相对单位，可理解为”root em”, 相对根节点html的字体大小来计算，CSS3新加属性。

- %：% 百分比，相对长度单位，相对于父元素的百分比值

- vw、vh、vmin、vmax 主要用于页面视口大小布局

vw：viewpoint width，视窗宽度，1vw等于视窗宽度的1%。

vh：viewpoint height，视窗高度，1vh等于视窗高度的1%。

vmin：vw和vh中较小的那个。

vmax：vw和vh中较大的那个。

多媒体（Multimedia）是多种媒体的综合，一般包括文本、声音、音乐、图像、动画、视频等媒体形式。多媒体以多种方式存在。

## 第一节 HTML5多媒体支持

**<audio>和<video>**

audio：定义声音播放器，比如音乐或其他音频流，支持三种格式MP3、WAV、Ogg。

video：定义视频播放器，比如电影片段或其他视频流，支持三种格式MP4、WebM、Ogg。

这两个元素使用上大同小异，基本语法如下：

**<audio** src="\*\*\*" controls="controls"**>**  
 当前浏览器不支持audio  
**</audio>**  
**<video** width="800" height="" src="\*\*\*" controls="controls"**>**  
 当前浏览器不支持video  
**</video>**

其中src是设置多媒体文件的路径，controls是设置是否使用播放控制，如果给标签里写了controls="controls"，那么网页会显示audio自带的播放控件，如果没写就不会显示。开始标签和结束标签中间的内容是供不支持<audio>或<video>元素的浏览器显示的。

<audio>和<video>元素还支持一些其他属性：

|  |  |
| --- | --- |
| 属性名 | 说明 |
| autoplay | 如果设置为autoplay，表示音视频加载完成后会自动播放，默认不设置，通常需要配合muted属性来使用 |
| loop | 如果设置为loop，表示音视频播放完成后会再次重复播放，默认不设置 |
| muted | 如果设置为muted，表示音频输出为静音 |
| preload | 如果设置为auto，表示预加载音视频 <br />如果设置为metadate，表示预加载音视频的元数据，如大小，时间，第一帧等<br />如果设置为none，表示不执行预加载 |
| poster | 只对<video>有效，设置视频加载完成播放前，显示的图片，属性值为图片url |
| width | 只对<video>有效，设置视频播放器的宽度 |
| height | 只对<video>有效，设置视频播放器的高度 |

## 

## 第二节 HTMLAudioElement和HTMLVideoElement

除了在HTML页面中使用<audio>元素和<video>元素来播放音频和视频，更多时候需要使用JavaScript脚本来控制<audio>元素和<video>元素，在JavaScript中获取<audio>元素的对象为HTMLAudioElement，获取<video>元素的对象为HTMLVideoElement。

HTMLAudioElement和HTMLVideoElement支持的方法有如下几个：

|  |  |
| --- | --- |
| 方法名 | 说明 |
| paly() | 播放音视频 |
| pause() | 暂停音视频 |
| load() | 重新加载音视频 |
| canPlayType() | 判断支持的type类型，属性值可以是MIME字符串，也可以是codecs属性，也可以MIME字符串并带codecs属性。它的返回值有以下三种： <br />probably：表示该浏览器支持播放这种格式的音视频 <br />maybe：表示该浏览器可能支持播放这种格式的音视频 <br />空字符串：表示该浏览器不支持播放这种格式的音视频 |

HTMLAudioElement和HTMLVideoElement还有一系列的属性用于了解当前媒体的状态：

错误状态

|  |  |
| --- | --- |
| 属性名 | 说明 |
| error | 正常返回null，不正常返回MediaError对象 |
| MediaError.code | MEDIA*ERR*ABORTED(数值1)：用户中止 <br />MEDIA*ERR*NETWORK(数值2)：网络错误 <br />MEDIA*ERR*DECODE(数值3)：解码错误 <br />MEDIA*ERR*SRC*NOT*SUPPORTED(数值4)：URL无效 |

网络状态

|  |  |
| --- | --- |
| 属性名 | 说明 |
| currentSrc | 返回当前资源的URL |
| src | 返回或设置当前资源的URL |
| networkState | 返回音频/视频的当前网络状态<br />NETWORK*EMPTY(数值0)：处于初始状态 <br />NETWORK*IDLE(数值1)：处于空闲状态，还未建立网络连接 <br />NETWORK*LOADING(数值2)：正在加载音视频 <br />NETWORK*NO\_SOURCE(数值3)：没有找到或不支持 |
| preload | 设置或返回音频的preload属性的值 |

准备状态

|  |  |
| --- | --- |
| 属性名 | 说明 |
| readyState | 返回音视频当前的就绪状态： <br />HAVE*NOTHING(数值0)：还未获取任何数据 <br />HAVE*METADATA(数值1)：已获取音视频的元数据，但没有获取媒体数据，还不能播放 HAVE*CURRENT*DATA(数值2)：已经获取当前位置媒体数据，还未获取下一位置媒体数据，或者当前位置已经是最后了 <br />HAVE*FUTURE*DATA(数值3)：已经获取当前位置媒体数据，也获取下一位置媒体数据 HAVE*ENOUGH*DATA(数值4)：已经获取了足够多的位置媒体数据，播放器可以顺利的向后播放 |
| seeking | 返回是否正在定位到指定的时间点 |

播放状态

|  |  |
| --- | --- |
| 属性名 | 说明 |
| currentTime | 返回或设置当前播放的音视频所处的时间点，单位为秒 |
| startTime | 返回音视频的开始时间，一般为0，如果为流媒体或者不从0开始的资源，则不为0 |
| duration | 返回当前音视频的长度，单位为秒，如果未设置则返回NaN |
| defaultPlaybackRate | 设置或返回音视频的默认播放速度 |
| playbackRate | 设置或返回音视频的当前播放速度 |
| ended | 返回是否结束 |
| autoPlay | 返回或设置是否在加载完成后随即播放音视频 |
| loop | 返回或设置是否循环播放 |
| controls | 返回或设置是否有默认控制条 |
| volume | 返回或设置音量大小 |
| muted | 返回或设置静音，true：静音，false：不静音 |
| buffered | 返回TimeRanges区域对象，通过该对象可以获取浏览器已经缓存的媒体数据。缓冲范围指的是已缓冲音视频的时间范围，如果用户在音视频中跳跃播放，会得到多个缓冲范围。 |
| seekable | 返回TimeRanges区域对象，用于获取音视频可定位的时间段 |
| paused | 返回是否暂停，值为true或者false。 |
| played | 返回已经播放的区域对象TimeRanges |
| TimeRanges.length | 区域对象的段数 |
| TimeRanges.start(index) | 返回指定index段的开始时间 |
| TimeRanges.end(index) | 返回指定index段的结束时间 |

## 

## 第三节 <audio>和<video>的事件

<audio>和<video>元素除了标准的onclick、onfocus事件，还有一些特有的事件：

|  |  |
| --- | --- |
| 事件名 | 说明 |
| onabort | 在还未下载完媒体数据而被中断时触发 |
| oncanplay | 当文件就绪可以开始播放时触发，在播放中间可能需要缓冲 |
| oncanplaythrough | 当文件就绪可以开始播放时触发，在播放中间不需要缓冲 |
| ondurationchange | 当音视频的长度改变时触发 |
| onemptied | 当发生故障并且文件突然不可用时触发，如网络错误，加载错误等 |
| onended | 当音视频播放到结尾时触发 |
| onerror | 当在文件加载期间发生错误时触发 |
| onloadeddata | 当音视频媒体数据加载完成时触发 |
| onloadedmetadata | 当音视频元数据加载完成时触发 |
| onloadstart | 当音视频开始加载时触发 |
| onpause | 暂停音视频时触发 |
| onplay | 即将开始播放音视频时触发 |
| onplaying | 正在播放音视频时触发 |
| onprogress | 正在加载音视频媒体数据时触发 |
| onratechange | 当播放速率改变时触发 |
| onreadystatechange | 当readustate属性值发生改变时触发 |
| onseeked | 当成功定位到音视频的指定位置，且seeking属性值被设置为false时触发 |
| onseeking | 当seeking属性被设置为true时触发 |
| onstalled | 在浏览器不论何种原因未能取回音视频媒体数据时触发 |
| onsuspend | 在音视频媒体数据完全加载之前不论何种原因终止取回媒介数据时触发 |
| ontimeupdate | 当播放位置改变时触发 |
| onvolumechange | 当音量改变时时触发 |
| onwaiting | 当音视频由于因无后面内容正在缓冲而暂停时触发 |

## 第四节 HTML5的绘图支持

**<canvas>元素**

HTML5新增了<canvas>元素用来在HTML页面上动态的绘制图片。实际上<canvas>元素自身是不绘制图形，它只相当于一张空白的透明的画布，如果想要在<canvas>元素上绘制图形，需要使用JavaScript脚本进行绘制。

<canvas>元素除了id，class等通用属性以外，只有两个属性width和height，分别用来设置画布的宽和高。

<canvas id="canvas" width="200px" height="200px"></canvas>

**绘制图形**

通过JavaScript向<canvas>元素绘图，大致可以分为以下四步：

* 获取<canvas>元素的DOM对象，即Canvas对象。
* 调用Canvas对象的getContext()方法，获得一个CanvasRendingContext2D对象。
* 设置CanvasRendingContext2D对象的属性。
* 调用CanvasRendingContext2D对象的方法绘图。

CanvasRendingContext2D对象的属性主要用于控制绘图的风格，

CanvasRendingContext2D常用的属性如下表所示：

|  |  |
| --- | --- |
| 属性名 | 说明 |
| lineWidth | 设置或返回当前的线条宽度 |
| fillStyle | 设置或返回用于填充路径的颜色、渐变或模式，取值有：<br />1. 表示颜色的字符串，表示纯色填充 <br />2. CanvasGradient对象，表示使用渐变填充，需要配合createLinearGradient()方法、createRadialGradient()方法、addColorStop()方法。 <br />3. CanvasPattern对象，表示使用位图填充 |
| strokeStyle | 设置或返回绘制路径的颜色、渐变或模式，取值同fillStyle：<br />1. 表示颜色的字符串，表示纯色填充 <br />2. CanvasGradient对象，表示使用渐变填充 <br />3. CanvasPattern对象，表示使用位图填充 |
| lineCap | 设置或返回线条的结束端点样式，取值有： <br />1. butt：不绘制端点，线条结尾处直接结束，默认值 <br />2. round：绘制圆形端点，线条结尾绘制一个直径为线条宽度的圆形 <br />3. square：绘制方端点，线条结尾绘制半个边长为线条宽度的正方形，和butt类似 |
| lineJoin | 设置或返回两条线相交时，所创建的拐角类型，取值有： <br />1. meter：默认属性，尖角 <br />2. round：圆角 <br />3. bevel：斜角 |
| miterLimit | 设置或返回最大斜接长度，当lineJoin属性值为meter时有效 |
| shadowColor | 设置或返回用于阴影的颜色 |
| shadowBlur | 设置或返回用于阴影的模糊级别 |
| shadowOffsetX | 设置或返回阴影距形状的水平距离 |
| shadowOffsetY | 设置或返回阴影距形状的垂直距离 |

**绘制几何图形**

填充一个矩形区域：fillRect(float x, float y, float width, float height)

绘制一个矩形边框：strokeRect(float x, float y, float width, float height)

参数表示矩形的左上角坐标在(x,y)位置，宽为width，高为height。

**绘制路径**

CanvasRenderingContext2D提供了一系列方法来绘制路径：

moveTo(float x, float y)：将当前位置移动到坐标(x, y)。

lineTo(float x, float y)：从当前位置向坐标(x, y)画一条直线路径。如果不存在当前位置，相当于执行moveTo(x, y)，也就剩说在崭新的路径中没有执行过任何操作的情况下，默认是不存在当前位置的，所以一般在执行lineTo()之前，先执行moveTo()。

stroke()：对当前路径中的线段或曲线进行描边。描边的颜色由

strokeStyle决定，描边的粗细由lineWidth决定。另外与stroke相关的属性还有lineCap、lineJoin、miterLimit。

**绘制字符串**

CanvasRendingContext2D提供了两个方法用来绘制字符串：

填充字符串：fillText(String str, float x, float y [, float maxwidth])

绘制字符串边框：strokeText(String str, float x, float y [, float maxwidth])

参数表示在坐标(x,y)位置输出文字str，文字的最大宽度为maxwidth，如果文字输出长度超过maxwidth，将被压缩。

**清除绘制内容**

**var** canvas = document.getElementById("canvas");  
**var** ctx = canvas.getContext("2d");  
*// 在填充矩形内部清除*  
ctx.fillRect(10, 10, 90, 90);  
ctx.clearRect(30, 30, 50, 50);  
  
*// 在线段上清除*  
ctx.lineWidth = 10;  
ctx.moveTo(130, 10);  
ctx.lineTo(130, 130);  
ctx.stroke();  
ctx.clearRect(120, 60, 30, 30);  
  
*// 在文字上清除*  
ctx.font = "25px blod arial,serif";  
ctx.fillText("HTML5规范", 10, 130);  
ctx.clearRect(10, 120, 100, 10);

**绘制阴影html**

要用到四个属性，shadowColor、shadowBlur、shadowOffsetX、shadowOffsetY。设置好以后，可以用在几何图形，也可以用在文字，还可以用在路径中

<!DOCTYPE html>  
**<html>**  
 **<head>**  
 **<meta** charset="utf-8"**>**  
 **<title>**HTML5多媒体**</title>**  
 **</head>**  
 **<body>**  
 **<canvas** id="canvas" width="300px" height="150px" style="border: 1px black solid;"**></canvas>**  
 **<script** type="text/javascript"**>**  
 **var** canvas = document.getElementById("canvas");  
 **var** ctx = canvas.getContext("2d");  
 ctx.fillStyle = "red";  
 ctx.strokeStyle = "red";  
 ctx.shadowBlur = 5;  
 ctx.shadowOffsetX = 10;  
 ctx.shadowOffsetY = 10;  
 ctx.shadowColor = "black";  
 ctx.strokeRect(10, 10, 60, 60);  
 ctx.fillRect(100, 10, 60, 60);  
 ctx.font = "20px Arial";  
 ctx.fillText("新年快乐", 10, 100);  
 ctx.strokeText("新年快乐", 100, 100);  
 ctx.moveTo(10, 130);  
 ctx.lineTo(100, 130);  
 ctx.stroke();  
 **</script>**  
 **</body>**  
**</html>**

**绘制位图**

CanvasRendingContext2D提供了drawImage()方法在画布上绘制位图，这个方法还能绘制位图的某一部分，或者增加/减少图像的尺寸。

drawImage(Image img, float x, float y)

该方法表示在画布(x,y)位置绘制图像img。

drawImage(Image img, float x, float y, float width, float height)

该方法表示在画布(x,y)位置绘制图像img，绘制出来的图像宽度为width，高度为height。

drawImage(Image img, float sx, float sy, float swidth, float sheight, float x, float y, float width, float height)

该方法表示从img图像的(sx,sy)位置截取宽度为swidth高度为sheight的矩形区域，并且在画布(x,y)位置绘制图像，绘制出来的图像宽度为width，高度为height。

drawImage()方法需要一个Image对象，Image对象通常用元素或者自己创建并指定src。

**变形**

CanvasRendingContext2D提供了四个变形方法：

translate(float x, float y)：平移坐标系，表示坐标系水平方向移动x，垂直方向移动y。

scale(float x, float y)：缩放坐标系，表示坐标系水平方向缩放比为x，垂直方向上缩放比为y。

rotate(float angle)：旋转坐标系，表示坐标系顺时针旋转angle角度。

transform (float m11, float m12, float m21, float m22, float dx, float dy)：使用矩阵变幻，他的6个参数中，dx和dy表示水平方向和垂直方向移动的距离，m11、m12、m21、m22是一个矩阵

通常使用变形时，为了无须计算多次坐标变幻后的累加结果，通常会使用save()和restore()来保存当前的绘图状态或者恢复之前的绘图状态，这些绘图状态包括填充风格，线条风格，阴影风格，变形设置等。

## 第五节 SVG

有四种把SVG文件嵌入HTML页面的方法：

1. 使用<img>标签

**<img** src="rect.svg" width="300" height="100" **/>**

<img>标签只能用于静态SVG图像的导入。

1. 使用<embed>标签

**<embed** src="rect.svg" width="300" height="100" type="image/svg+xml" pluginspage="http://www.adobe.com/svg/viewer/install/" **/>**

1. 使用<object>标签

**<object** data="rect.svg" width="300" height="100" type="image/svg+xml"  
  
codebase="http://www.adobe.com/svg/viewer/install/" **/>**

<embed>标签和<object>标签使用上相同，不同的是<embed>标签可以使用脚本，<object>标签不可以使用脚本。这两种写法都或多或少存在一些问题，<embed>主要目的是实现Flash插件，而且XHTML标准中是没有<embed>标签。

1. 使用<iframe>标签

**<iframe** src="rect.svg" width="300" height="100"**>**\**</iframe>**

<iframe>标签在浏览器的兼容性是最好的，可以使用脚本，但<iframe>标签难以维护，且不会被搜索引擎索引，无法做搜索引擎优化。

**SVG的基本语法**

从本质上来说，SVG文档是XML文档，也就是说SVG文档具有XML基本属性。文档应该以XML声明<?xml version="1.0"?>开始，还需要指定DTD：<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">，然后在<svg>标签内编写矢量图的信息，这些信息很大程度上都类似CSS，然后保存即可完成一张矢量图。

**<svg>标签**

<svg>标签有如下属性：

|  |  |
| --- | --- |
| 属性名 | 说明 |
| width | 用来控制SVG视图的宽度 |
| height | 用来控制SVG视图的高度 |
| viewBox | 定义用户视野的位置以及大小，即定义用来观察SVG视图一个矩形区域，它的属性值是一个“x y width height”，四个数字用空格或逗号隔开，表示在左上角在(x,y)，宽度为width，高度为height的矩形 |

**<svg>内部标签**

SVG有一些预定义的标签可用在<svg>标签内部。

|  |  |
| --- | --- |
| 标签名 | 说明 |
| <rect> | 矩形标签 |
| <circle> | 圆形标签 |
| <ellipse> | 椭圆标签 |
| <line> | 线标签 |
| <polyline> | 折线标签 |
| <polygon> | 多边形标签 |
| <path> | 路径标签 |
| <text> | 文字标签 |
| <tspan> | 类似<span>用在<text>内部单独设置样式 |

**路径标签**

路径使用<path>标签进行绘制，使用d属性控制路径的类型及绘制，它和<canvas>的路径有着很多相似的地方。d属性值的书写格式有两种，可以使用逗号分隔坐标，如：d="M 10,10"，也可使用空格的形式，如：d="M 10 10"，通常建议使用逗号分格，不同的点用空格分隔。

|  |  |
| --- | --- |
| <canvas>方法 | <path>d的参数 |
| moveTo(10,20) | M 10,20 |
| lineTo(20,10) | L 20,10 |
| quadraticCurveTo(40,140,100,100) | Q 40,140 100,100 |
| bezierCurveTo(40,140,100,100,200,40) | C 40,140 100,100 200,40 |
| closePath() | Z |

**文字标签**

在svg中使用<text></text>标签绘制文字。它可以设置如下的属性：

x：文字的x坐标。

y：文字的y坐标。

dx：相对于当前位置x方向的距离。

dy：相对于当前位置y方向的距离。

textLength：文字的显示长度，不足则拉长，不然则压缩。

rotate：旋转角度，也可以使用transform="rotate(angle)"来实现

# 第二章 Web Storage

HTML5规范推出了Web Storage，目的是为了克服由cookie带来的一些问题，当数据需要被严格控制在客户端上时或者无须持续地将数据发回服务器。因此，Web Storage的两个主要目标是：

* 提供一种在cookie之外存储会话数据的途径。
* 提供一种存储大量可以跨会话存在的数据的机制。

Web Storage提供了两种客户端存储数据的方法：sessionStorage和localStorage。

sessionStorage是基于Session的Web Storage，它保存的数据的生存时间和Session的时间相同，用户Session结束时，sessionStorage保存的数据也就丢失了，因此，sessionStorage不是一个持久化的本地存储，仅仅是一个会话级别的存储。

localStorage是保存在用户磁盘的Web Storage，它和sessionStorage不同的是，localStorage保存的数据生存时间很长，除非用户或者程序显式的清楚这些数据，否则这些数据将一直存在。

它们的使用方法都很像，只是实现上有些不同，Web Storage有六个方法：

|  |  |
| --- | --- |
| 方法 | 说明 |
| length | 调用该属性返回Web Storage保存了多少组键值对。 |
| key(int index） | 调用该方法返回Web Storage中第index个键名。 |
| getItem(string key) | 调用该方法返回Web Storage中指定key键对应的值。 |
| setItem(string key, string value) | 调用该方法向Web Storage中存入指定的键值对key-value。 |
| removeItem(string key) | 调用该方法将从Web Storage中移除指定key键的键值对。 |
| clear() | 调用该方法清楚Web Storage的所有键值对 |

Web Storage是window对象的子对象，我们可以通过window.sessionStorage或者window.localStorage获取Web Storage对象，或者直接使用即可。

Web Storage可以存储一些用户配置信息，如网站风格，访问次数计数器等，总而言之，只要是cookie能做的工作，Web Storage都可以做。

# 第三章 Less的使用

## 第一节 Less介绍

Less是一门CSS预处理语言，它扩充了CSS语言，增加了诸如变量、混合、嵌套、函数等功能，使CSS更易维护和扩展。

## 第二节 Less的安装

直接在HTML页面中引入less.js，可以只使用CDN，也可以下载好本地直接指定。

## 第三节 Less的使用

**变量**

以@开头定义变量，并且使用时直接键入@+变量名称。

@color: #999;  
@width: 50%;  
#content{  
color: @color;  
width: @width;  
}

**嵌套**

在Less中，是支持嵌套的写法，如果在嵌套中使用了“&”，“&”代表的上一层选择器的名字，这时候就没有嵌套。

#header{  
 .title{  
 **margin**:20px;  
 }  
 &\_content{  
 **margin**:20px;  
 }  
}  
  
*/\* 生成的 CSS \*/*  
#header .title{ **//有嵌套**  
 **margin**:20px;  
}  
#header\_content{ **//无嵌套**  
 **margin**:20px;  
}

**混合**

无参数的混合

.card {  
 width: 200px;  
 height: 200px;  
 border: solid 1px black;  
};  
#con{  
 .card();  
}  
  
/\* 生成的 CSS \*/  
.card {  
 width: 200px;  
 height: 200px;  
 border: solid 1px black;  
};  
#con{  
 width: 200px;  
 height: 200px;  
 border: solid 1px black;  
}

有默认参数的混合Less

.border(@a:10px,@b:50px,@c:30px,@color:#000){  
 border:solid 1px @color;  
 box-shadow: @arguments; //指代的是全部参数  
}  
#main{  
 .border(0px,5px,30px,red); //必须带着单位  
}  
#title{  
 .border(0px);  
}  
#content{  
 .border; //等价于.border()  
}  
  
/\* 生成的 CSS \*/  
#main{  
 border:solid 1px red;  
 box-shadow:0px,5px,30px,red;  
}  
#wrap{  
 border:solid 1px #000;  
 box-shadow: 0px 50px 30px #000;  
}  
#content{  
 border:solid 1px #000;  
 box-shadow: 10px 50px 30px #000;  
}

**继承**

extend是Less的一个伪类。它可继承所匹配声明中的全部样式

.animation{  
 transition: all .3s ease-out;  
 .hide{  
 transform:scale(0);  
 }  
}  
#main{  
 &:extend(.animation);  
}  
#content{  
 &:extend(.animation .hide);  
}  
  
/\* 生成的 CSS \*/  
.animation, #main {  
 transition: all 0.3s ease-out;  
}  
.animation .hide, #content {  
 transform: scale(0);  
}

**函数**

Less提供了一系列的函数，我们常用的函数见下表:

|  |  |
| --- | --- |
| 函数 | 说明 |
| isnumber | 判断给定的值是否是一个数字 |
| iscolor | 判断给定的值是否是一个颜色 |
| isurl | 判断给定的值是否是一个URL |
| saturate | 增加一定数值的颜色饱和度 |
| lighten | 增加一定数值的颜色亮度 |
| darken | 降低一定数值的颜色亮度 |
| fade | 给颜色设定一定数值的透明度。 |
| mix | 根据比例混合两种颜色。 |
| ceil | 向上取整 |
| floor | 向下取整 |
| percentage | 将浮点数转换为百分比字符串 |
| round | 四舍五入 |
| sqrt | 平方根 |
| abs | 绝对值 |
| pow | 乘方 |

**导入**

Less可以导入其他写好的less文件，例如我们可以把一些常用的颜色，定义成color.less，常用的字体和字号，定义成font.less，然后在正式的less文件中直接导入即可。

导入的关键字是import，如果导入的是less文件，可以省略后缀。

@import "font.less";  
@import "color";

# 第四章 jQuery Mobile基本

由于移动设备的发展，移动端浏览器也在飞速发展，移动端浏览器的使用体验也在逐渐的赶上桌面端浏览器。在这种大的发展趋势上，jQuery框架也增加了一个组件，叫做jQuery Mobile。

## 第一节 jQuery Mobile的安装

从它的官方网站 <https://jquerymobile.com/download/>下载文件

<link rel="stylesheet" href="jquery.mobile-1.4.5.min.css">  
<script src="jquery-1.11.1.min.js"></script>  
<script src="jquery.mobile-1.4.5.min.js"></script>

## 第二节 jQuery Mobile的使用

**jQuery Mobile页面**

我们可以使用如下几个属性来定义页面：

1. data-role="page"：显示在浏览器中的页面。
2. data-role="header"：定义页面上方的标题栏。
3. data-role="content"：定义页面的内容，比如文本、图像、表单等。
4. data-role="footer"：定义页面下方的底栏。
5. data-rel="dialog"：创建链接使用对话框样式。

<!DOCTYPE html>  
**<html>**  
 **<head>**  
 **<meta** charset="utf-8"**>**  
 **<title>**jQuery Mobile页面**</title>**  
 **<link** rel="stylesheet" href="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.css" **/>**  
 **<script** src="http://code.jquery.com/jquery-1.11.1.min.js"**></script>**  
 **<script** src="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.js"**></script>**  
 **</head>**  
 **<body>**  
 **<div** data-role="page"**>**  
 **<div** data-role="header"**>**  
 **<h1>**主页**</h1>**  
 **</div>**  
 **<div** data-role="content"**>**  
 **<p>**jQuery Mobile页面**</p>**  
 **</div>**  
 **<div** data-role="footer"**>**  
 **<p>**联系我们**</p>**  
 **</div>**  
 **</div>**  
 **</body>**  
**</html>**

**jQuery Mobile过渡**

jQuery Mobile拥有一系列的关于如何从一个页面过渡到下一个页面的效果。值得注意的是浏览器必须支持CSS3的3D转换效果，才能实现过渡效果。jQuery Mobile使用data-transition属性来实现过渡效果，他可能的值有很多：

|  |  |
| --- | --- |
| 过渡属性值 | 说明 |
| fade | 默认值，淡入淡出到下一个页面 |
| flip | 以页面中线为轴，从右向左翻转到下一个页面 |
| flow | 缩小当前页面，向左滑动到下一个页面并放大至全屏 |
| pop | 以弹出形式跳转到下一个页面 |
| slide | 从右向左滑动到下一个页面 |
| slidefade | 从右向左滑动并淡入淡出到下一个页面 |
| slideup | 从下到上滑动到下一个页面 |
| slidedown | 从上到下滑动到下一个页面 |
| turn | 沿着页面左轴，空间旋转转向下一个页面 |
| none | 无过渡效果 |

**jQuery Mobile定位**

我们在使用jQuery Mobile的页眉页脚时，有时候会希望页眉或者页脚根据一定的条件来隐藏，我们可以使用data-position属性来设置页眉页脚的定位属性，它可以设置为两个值：

Inline：默认值，页眉和页脚与页面内容位于行内，并且页眉和页脚留在页面的顶部和底部，如果有滚动条，在滚动条中间部分，会出现既看不见页眉，也看不到页脚的情况。

Fixed：如果滚动条可用，点击屏幕将隐藏或显示页眉

页脚。效果会根据当前滚动条在页面上的位置而变化。当滚动条在顶部时可以点击屏幕将页脚显示出来，也可以隐藏页脚并放置在页面的底部；同理如果滚动条在底部时，可以点击屏幕将页眉显示出来，也可以隐藏页眉并放置在页面的顶部；如果滚动条在中间部分，可以点击屏幕将页眉页脚全部显示或者隐藏。如果想达到一次性将页眉页脚全部隐藏或者显示，可以使用data-fullscreen="true"属性，这种情况通常用于浏览图片等内容。

<!DOCTYPE html>  
**<html>**  
 **<head>**  
 **<meta** charset="utf-8"**>**  
 **<title>**jQuery Mobile定位</title>  
 **<link** rel="stylesheet" href="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.css" **/>**  
 **<script** src="http://code.jquery.com/jquery-1.11.1.min.js"**></script>**  
 **<script** src="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.js"**></script>**  
 **</head>**  
 **<body>**  
 **<div** data-role="page"**>**  
 **<div** data-role="header" data-position="inline"**>**行内页眉**</div>**  
 **<div** data-role="footer" data-position="inline"**>**行内页脚**</div>**  
 **</div>**  
 **<div** data-role="page"**>**  
 **<div** data-role="header" data-position="fixed"**>**定位页眉**</div>**  
 **<div** data-role="footer" data-position="fixed"**>**定位页脚**</div>**  
 **</div>**  
 **<div** data-role="page"**>**  
 **<div** data-role="header" data-position="fixed" data-fullscreen="true"**>**全面定位页眉**</div>**  
 **<div** data-role="footer" data-position="fixed" data-fullscreen="true"**>**全面定位页脚**</div>**  
 **</div>**  
 **</body>**  
**</html>**

**jQuery Mobile按钮**

HTML提供了两种创建按钮的办法，<button>元素和<input type="button">。在jQuery Mobile中创建按钮额外提供了一种办法，那就是给<a>元素使用属性data-role="button"，达到按钮的样式

<!DOCTYPE html>  
**<html>**  
 **<head>**  
 **<meta** charset="utf-8"**>**  
 **<title>**jQuery Mobile按钮**</title>**  
 **<link** rel="stylesheet" href="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.css" **/>**  
 **<script** src="http://code.jquery.com/jquery-1.11.1.min.js"**></script>**  
 **<script** src="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.js"**></script>**  
 **</head>**  
 **<body>**  
 **<div** data-role="page" id="menu"**>**  
 **<div** data-role="header"**>**  
 **<h1>**按钮**</h1>**  
 **</div>**  
 **<div** data-role="content"**>**  
 **<button>**button按钮**</button>**  
 **<input** type="button" value="input按钮" **/>**  
 **<a** href="" data-role="button"**>**a元素按钮**</a>**  
 **</div>**  
 **<div** data-role="footer"**>**  
 **<h1>**联系我们**</h1>**  
 **</div>**  
 **</div>**  
 **</body>**  
**</html>**

**jQuery Mobile图标**

|  |  |
| --- | --- |
| IconsReference | 说明 |
| arrow-l | 左箭头图标 |
| arrow-r | 右箭头图标 |
| arrow-u | 上箭头图标 |
| arrow-d | 下箭头图标 |
| plus | 加号图标 |
| minus | 减号图标 |
| delete | 删除图标 |
| check | 检查图标 |
| home | 首页图标 |
| info | 信息图标 |
| back | 后退图标 |
| forward | 向前图标 |
| refresh | 刷新图标 |
| grid | 网格图标 |
| gear | 齿轮图标 |
| search | 搜索图标 |
| star | 星形图标 |
| alert | 提醒图标 |

**jQuery Mobile导航栏**

jQuery Mobile导航栏是由一组水平排列的链接组成，通常用于页眉或者页脚中，这些链接将会默认加上data-role="button"属性，无须在手动添加该属性。jQuery Mobile使用data-role="navbar"属性来定义导航栏：

<!DOCTYPE html>  
**<html>**  
 **<head>**  
 **<meta** charset="utf-8"**>**  
 **<title>**jQuery Mobile导航栏**</title>**  
 **<link** rel="stylesheet" href="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.css" **/>**  
 **<script** src="http://code.jquery.com/jquery-1.11.1.min.js"**></script>**  
 **<script** src="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.js"**></script>**  
 **</head>**  
 **<body>**  
 **<div** data-role="page"**>**  
 **<div** data-role="header"**>**  
 **<div** data-role="navbar"**>**  
 **<ul>**  
 **<li><a>**首页**</a></li>**  
 **<li><a>**下载</a>**</li>**  
 **<li><a>**文档**</a></li>**  
 **</ul>**  
 **</div>**  
 **</div>**  
 **<div** data-role="content"**>**  
 **<div** data-role="navbar"**>**  
 **<a>**HTML5**</a>**  
 **<a>**CSS3**</a>**  
 **<a>**JavaScript**</a>**  
 **</div>**  
 **<div** data-role="navbar"**>**  
 **<ul>**  
 **<li><a>**jQuery Mobile 页面**</a></li>**  
 **<li><a>**jQuery Mobile 按钮**</a></li>**  
 **<li><a>**jQuery Mobile 图标**</a></li>**  
 **</ul>**  
 **</div>**  
 **</div>**  
 **<div** data-role="footer"**>**  
 **<div** data-role="navbar"**>**  
 **<ul>**  
 **<li><a>**网站地图**</a></li>**  
 **<li><a>**联系我们**</a></li>**  
 **<li><a>**关于我们**</a></li>**  
 **</ul>**  
 **</div>**  
 **</div>**  
 **</div>**  
 **</body>**  
**</html>**

**jQuery Mobile折叠**

jQuery Mobile对容器增加了折叠效果，用来设置隐藏或显示的内容。要创建可折叠的内容块，需要给容器（<div>）设置data-role="collapsible"属性，然后在容器中，添加一个标题元素（<h1>~<h6>），其后是折叠的内容。如果某容器设置了data-role="collapsible"属性，则默认是折叠的，如果想在页面加载时，展开折叠的内容，则需要设置data-collapsed="false"属性。

<!DOCTYPE html>  
**<html>**  
 **<head>**  
 **<meta** charset="utf-8"**>**  
 **<title>**jQuery Mobile折叠**</title>**  
 **<link** rel="stylesheet" href="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.css" **/>**  
 **<script** src="http://code.jquery.com/jquery-1.11.1.min.js"**></script>**  
 **<script** src="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.js"**></script>**  
 **</head>**  
 **<body>**  
 **<div** data-role="page"**>**  
 **<div** data-role="header"**>**  
 **<h1>**jQuery Mobile**</h1>**  
 **</div>**  
 **<div** data-role="content"**>**  
 **<div** data-role="collapsible" data-collapsed="false"**>**  
 **<h1>**jQuery Mobile页面**</h1>**  
 **<ul>**  
 **<li>**data-role="page"：显示在浏览器中的页面**</li>**  
 **<li>**data-role="header"：定义页面上方的标题栏**</li>**  
 **<li>**data-role="content"：定义页面的内容，比如文本、图像、表单等**</li>**  
 **<li>**data-role="footer"：定义页面下方的底栏**</li>**  
 **</ul>**  
 **</div>**  
 **<div** data-role="collapsible"**>**  
 **<h1>**jQuery Mobile按钮**</h1>**  
 **<pre>**  
&lt;button&gt;  
&lt;input type="button"&gt;  
&lt;a data-role="button"&gt;  
 **</pre>**  
 **</div>**  
 **</div>**  
 **<div** data-role="footer"**>**  
 **<h1>**联系我们**</h1>**  
 **</div>**  
 **</div>**  
 **</body>**  
**</html>**

**jQuery Mobile列布局**

jQuery Mobile提供了一套基于CSS的列布局方案。由于移动设备的屏幕宽度的限制，一般不使用列布局。这种列布局会等分最大宽度，显示上无边框、背景、外边距或内边距等效果，这些效果通常需要使用CSS样式单来定义。

|  |  |  |  |
| --- | --- | --- | --- |
| 列布局类名 | 列数 | 每列宽度 | 每一列对应类名 |
| ui-grid-a | 2 | 50% | ui-block-a|b |
| ui-grid-b | 3 | 33% | ui-block-a|b|c |
| ui-grid-c | 4 | 25% | ui-block-a|b|c|d |
| ui-grid-d | 5 | 20% | ui-block-a|b|c|d|e |

<!DOCTYPE html>  
**<html>**  
 **<head>**  
 **<meta** charset="utf-8"**>**  
 **<title>**jQuery Mobile列布局**</title>**  
 **<link** rel="stylesheet" href="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.css" **/>**  
 **<script** src="http://code.jquery.com/jquery-1.11.1.min.js"**></script>**  
 **<script** src="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.js"**></script>**  
 **</head>**  
 **<body>**  
 **<div** data-role="page"**>**  
 **<div** data-role="header"**>**  
 **<h1>**jQuery Mobile**</h1>**  
 **</div>**  
 **<div** data-role="content"**>**  
 **<div** class="ui-grid-b"**>**  
 **<div** class="ui-block-a" style="border: 1px solid black;padding: 10px;"**><span>**3\*3=9**</span></div>**  
 **<div** class="ui-block-b" style="border: 1px solid black;padding: 10px;"**><span>**3\*2=6**</span></div>**  
 **<div** class="ui-block-c" style="border: 1px solid black;padding: 10px;"**><span>**3\*1=3**</span></div>**  
 **<div** class="ui-block-a" style="border: 1px solid black;padding: 10px;"**><span>**2\*2=4**</span></div>**  
 **<div** class="ui-block-b" style="border: 1px solid black;padding: 10px;"**><span>**2\*1=2**</span></div>**  
 **<div** class="ui-block-a" style="border: 1px solid black;padding: 10px;"**><span>**1\*1=1**</span></div>**  
 **</div>**  
 **</div>**  
 **<div** data-role="footer"**>**  
 **<h1>**联系我们**</h1>**  
 **</div>**  
 **</div>**  
 **</body>**  
**</html>**

**jQuery Mobile列表**

jQuery Mobile中的列表是标准的HTML的列表，即有序列表(<ol>)和无序列表(<ul>)。创建列表给<ol>或<ul>元素添加data-role="listview"属性，如果给每个列表项添加圆角和外边距，需要添加data-inset="true"属性。

<!DOCTYPE html>  
**<html>**  
 **<head>**  
 **<meta** charset="utf-8"**>**  
 **<title>**jQuery Mobile列表**</title>**  
 **<link** rel="stylesheet" href="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.css" **/>**  
 **<script** src="http://code.jquery.com/jquery-1.11.1.min.js"**></script>**  
 **<script** src="http://code.jquery.com/mobile/1.4.5/jquery.mobile-1.4.5.min.js"**></script>**  
 **</head>**  
 **<body>**  
 **<div** data-role="page"**>**  
 **<div** data-role="header"**>**  
 **<h1>**jQuery Mobile**</h1>**  
 **</div>**  
 **<div** data-role="content"**>**  
 **<h1>**普通列表：</h1>  
 **<ul** data-role="listview"**>**  
 **<li><a>**jQuery Mobile 页面**</a></li>**  
 **<li><a>**jQuery Mobile 按钮**</a></li>**  
 **<li><a>**jQuery Mobile 图标**</a></li>**  
 **</ul><br>**  
 **<h1>**带有圆角和外边距的列表：</h1>  
 **<ul** data-role="listview" data-inset="true"**>**  
 **<li><a>**jQuery Mobile 页面**</a></li>**  
 **<li><a>**jQuery Mobile 按钮**</a></li>**  
 **<li><a>**jQuery Mobile 图标**</a></li>**  
 **</ul>**  
 **</div>**  
 **<div** data-role="footer"**>**  
 **<h1>**联系我们**</h1>**  
 **</div>**  
 **</div>**  
 **</body>**  
**</html>**

**jQuery Mobile表单**

jQuery Mobile 使用 CSS 为 HTML 表单元素添加样式，让它们更具吸引力，更易于使用。

**表单控件**

在 jQuery Mobile 中，您可以使用下列表单控件：

1. 文本输入框
2. 搜索输入框
3. 单选按钮
4. 复选框
5. 选择菜单
6. 滑动条
7. 翻转拨动开关

当使用 jQuery Mobile 表单时，您应当知道：

1. <form> 元素必须有一个 method 和一个 action 属性
2. 每个表单元素必须有一个唯一的 "id" 属性。id 必须是整个站点所有页面上唯一的。这是因为 jQuery Mobile 的单页导航机制使得多个不同页面在同一时间被呈现
3. 每个表单元素必须有一个<label>标签。设置标签的 **for** 属性来匹配元素的 id

**<form** method="post" action="demoform.html"**>**  
 **<label** for="fname"**>**姓名:**</label><input** type="text" name="fname" id="fname"**>**  
**</form>**

如需隐藏标签，请使用 class ui-hidden-accessible。这在您把元素的 placeholder 属性作为标签时经常用到：

**<form** method="post" action="demoform.html"**>**  
 **<label** for="fname" class="ui-hidden-accessible"**>**姓名:**</label>**  
 **<input** type="text" name="fname" id="fname" placeholder="姓名..."**>**  
 **</form>**

**Field 容器**

如需让标签和表单元素看起来更适应宽屏，请用带有 data-role="fieldcontain" 属性的 <div> 或<fieldset> 元素包围 label/form 元素：

**<form** method="post" action="demoform.html"**>**  
 **<div** data-role="fieldcontain"**>**  
 **<label** for="fname"**>**姓:**</label>**  
 **<input** type="text" name="fname" id="fname"**>**  
 **<label** for="lname"**>**名:**</label>**  
 **<input** type="text" name="lname" id="lname"**>**  
 **</div>**>  
 **</form>**

fieldcontain 属性基于页面的宽度为标签和表单控件添加样式。当页面的宽度大于 480px 时，它会自动把标签放置在与表单控件同一线上。当页面的宽度小于 480px 时，标签会被放置在表单元素的上面。

**提示：**为了防止 jQuery Mobile 为可点击元素自动添加样式，请使用 data-role="none" 属性：

**<label** for="fname"**>**姓名:**</label>**  
**<input** type="text" name="fname" id="fname" data-role="none"**>**

jQuery Mobile 中的表单提交： jQuery Mobile 通过 AJAX 自动处理表单提交，并将试图集成服务器响应到应用程序的 DOM 中。

## 第三节 jQuery Mobile主题

jQuery Mobile 提供了5种不同的主题样式, 从 "a" 到 "e" - 每一种主题的按钮，工具条，内容块等等颜色都不一致，每个主题的视觉效果也不一样。

通过设置元素的data-theme属性可以自定义应用的外观:

<div data-role="page" data-theme="a|b|c|d|e">

|  |  |
| --- | --- |
| 值 | 描述 |
| a | 默认。黑色背景白色文字 |
| b | 蓝色背景白色文字/ 黑色文字灰色背景 |
| c | 黑色文字浅灰色背景 |
| d | 黑色为主白色背景 |
| e | 黑色文字橙色背景 |

默认情况下， jQuery Mobile 使用 "a" 主题 (黑色) 来设置头部和底部 ， "c" 主题 (浅灰色) 设置页面内容。 但是，你可以自定义设置你喜欢的混合主题。

**主题头部、内容和底部**

**<div** data-role="header" data-theme="b"**></div>**  
**<div** data-role="content" data-theme="a"**></div>**  
**<div** data-role="footer" data-theme="e"**></div>**

**主题对话框**

**<a** href="#pagetwo" data-rel="dialog"**>**Go To The Themed Dialog Page**</a>**  
  
**<div** data-role="page" id="pagetwo" data-overlay-theme="e"**>**  
 **<div** data-role="header" data-theme="b"**></div>**  
 **<div** data-role="content" data-theme="a"**></div>**  
 **<div** data-role="footer" data-theme="c"**></div>**  
**</div>**

**主题按钮**

**<a** href="#" data-role="button" data-theme="a"**>**Button**</a>**  
**<a** href="#" data-role="button" data-theme="b"**>**Button**</a>**  
**<a** href="#" data-role="button" data-theme="c"**>**Button**</a>**

**主题图标**

**<a** href="#" data-role="button" data-icon="plus" data-theme="e"**>**Plus**</a>**

**头部和底部的主题按钮**

**<div** data-role="header"**>**  
 **<a** href="#" data-role="button" data-icon="home" data-theme="b"**>**Home**</a>**  
 **<h1>**Welcome To My Homepage**</h1>**  
 **<a** href="#" data-role="button" data-icon="search" data-theme="e"**>**Search**</a>**  
**</div>**  
  
**<div** data-role="footer"**>**  
 **<a** href="#" data-role="button" data-theme="b" data-icon="plus"**>**Button 1**</a>**  
 **<a** href="#" data-role="button" data-theme="c" data-icon="plus"**>**Button 2**</a>**  
 **<a** href="#" data-role="button" data-theme="e" data-icon="plus"**>**Button 3**</a>**  
**</div>**

**主题导航条**

**<div** data-role="footer" data-theme="e"**>**  
 **<h1>**Insert Footer Text Here**</h1>**  
 **<div** data-role="navbar"**>**  
 **<ul>**  
 **<li><a** href="#" data-icon="home" data-theme="b"**>**Button 1**</a></li>**  
 **<li><a** href="#" data-icon="arrow-r"**>**Button 2**</a></li>**  
 **<li><a** href="#" data-icon="arrow-r"**>**Button 3**</a></li>**  
 **<li><a** href="#" data-icon="search" data-theme="a" **>**Button 4**</a></li>**  
 **</ul>**  
 **</div>**  
**</div>**

**主题可折叠按钮和内容**

**<div** data-role="collapsible" data-theme="b" data-content-theme="e"**>**  
 **<h1>**Click me - I'm collapsible!**</h1>**  
 **<p>**I'm the expanded content.**</p>**  
**</div>**

**主题列表**

**<ul** data-role="listview" data-theme="e"**>**  
 **<li><a** href="#"**>**List Item**</a></li>**  
 **<li** data-theme="a"**><a** href="#"**>**List Item**</a></li>**  
 **<li** data-theme="b"**><a** href="#"**>**List Item**</a></li>**  
 **<li><a** href="#"**>**List Item**</a></li>**  
**</ul>**

**主题分割按钮**

**<ul** data-role="listview" data-split-theme="e"**>**

**主题可折叠列表**

**<div** data-role="collapsible" data-theme="b" data-content-theme="e"**>**  
 **<ul** data-role="listview"**>**  
 **<li><a** href="#"**>**Agnes**</a></li>**  
 **</ul>**  
**</div>**

**主题表单**

**<label** for="name"**>**Full Name:**</label>**  
**<input** type="text" name="text" id="name" data-theme="a"**>**  
  
**<label** for="colors"**>**Choose Favorite Color:**</label>**  
**<select** id="colors" name="colors" data-theme="b"**>**  
 **<option** value="red"**>**Red**</option>**  
 **<option** value="green"**>**Green**</option>**  
 **<option** value="blue"**>**Blue**</option>**  
**</select>**

**主题可折叠表单**

**<fieldset** data-role="collapsible" data-theme="b" data-content-theme="e"**>**  
 **<legend>**Click me - I'm collapsible!**</legend>**

**添加新主题**

jQuery Mobile 可以在移动页面添加新主题。

通过修改 CSS 文件来添加或编辑新主题(如果你已经下载了 jQuery Mobile)。你只需要拷贝样式模块，然后重命令字母类名（f-z）,并在样式中添加你喜欢的颜色和字体。

你也可以在 HTML 文档中添加主题的新样式 - 工具条添加类： ui-bar-(a-z) ，文本内容添加类: ui-body-(a-z) for the content:

<style>  
.ui-bar-f{**color**:green;**background-color**:yellow; }  
.ui-body-f{**font-weight**:bold;**color**:purple;}  
</style>

## 

## 第四节 jQuery Mobile事件

**jQuery Mobile页面事件**

在jQuery Mobile中页面事件主要分成三类：

页面初始化：有四个事件分别是：在页面初始化前、初始化时、初始化后、初始化失败，对应事件名称分别是pagebeforecreate、pagecreate、pageinit、pageloadfailed。

页面加载：有三个事件分别是：当外部页面加载前、加载后、加载失败，对应事件名称分别是pagebeforeload、pageload、pageloadfailed。

页面过渡：有四个事件分别是：前一页面过渡前、前一页面过渡后、下一页面过渡前、下一页面过渡后，对应事件名称分别是pagebeforehide、pagehide、pagebeforeshow、pageshow。

**jQuery Mobile触摸事件**

jQuery Mobile为移动浏览器增加的触摸事件有以下五种：

tap事件：触摸就触发。

taphold事件：触摸并保持一秒不松开则触发。

swipe事件：在某元素上水平滑动30px触发。

swipeleft事件：在某元素上向左水平滑动30px触发。

swiperight事件：在某元素上向右水平滑动30px触发。

**jQuery Mobile滚动事件**

JQuery Mobile提供了两个滚动事件：

scrollstart 事件：在滚动开始时触发。

scrollstop 事件：在滚动结束时触发。

**jQuery Mobile方向事件**

JQuery Mobile的方向事件是orientationchange事件，该事件只会在用户垂直或水平旋转移动设备时被触发。由于orientationchange事件与window对象绑定，可用使用window.orientation属性来得知屏幕的当前方向，如果是portrait视图（垂直）返回0或180，如果是landscape视图（水平）返回90或-90。