

Vue学习看这篇就够

Vue - 渐进式JavaScript框架

介绍

- [vue 中文网](#)
- [vue github](#)
- Vue.js 是一套构建用户界面(UI)的渐进式JavaScript框架

库和框架的区别

- [我们所说的前端框架与库的区别?](#)

Library

库，本质上是一些函数的集合。每次调用函数，实现一个特定的功能，接着把 **控制权** 交给使用者

- 代表：jQuery
- jQuery这个库的核心：DOM操作，即：封装DOM操作，简化DOM操作

Framework

框架，是一套完整的解决方案，使用框架的时候，需要把你的代码放到框架合适的地方，框架会在合适的时机调用你的代码

- 框架规定了自己的编程方式，是一套完整的解决方案
- 使用框架的时候，由框架控制一切，我们只需要按照规则写代码

主要区别

- You call Library, Framework calls you
- 核心点：谁起到主导作用（控制反转）
 - 框架中控制整个流程的是框架
 - 使用库，由开发人员决定如何调用库中提供的方法（辅助）
- 好莱坞原则：Don't call us, we'll call you.

- 框架的侵入性很高（从头到尾）

MVVM的介绍

- MVVM，一种更好的UI模式解决方案
- [从Script到Code Blocks、Code Behind到MVC、MVP、MVVM - 科普](#)

MVC

- M: Model 数据模型（专门用来操作数据，数据的CRUD）
- V: View 视图（对于前端来说，就是页面）
- C: Controller 控制器（是视图和数据模型沟通的桥梁，用于处理业务逻辑）

MVVM组成

- MVVM ==> M / V / VM
- M: model数据模型
- V: view视图
- VM: ViewModel 视图模型

优势对比

- MVC模式，将应用程序划分为三大部分，实现了职责分离
- 在前端中经常要通过 JS代码 来进行一些逻辑操作，最终还要把这些逻辑操作的结果现在页面中。也就是需要频繁的操作DOM
- MVVM通过 **数据双向绑定** 让数据自动地双向同步
 - V (修改数据) -> M
 - M (修改数据) -> V
 - 数据是核心
- Vue这种MVVM模式的框架，不推荐开发人员手动操作DOM

Vue中的MVVM

虽然没有完全遵循 MVVM 模型，Vue 的设计无疑受到了它的启发。因此在文档中经常会使用 `vm` (`ViewModel` 的简称) 这个变量名表示 Vue 实例

学习Vue要转化思想

- 不要在想着怎么操作DOM，而是想着如何操作数据！！！

起步 - Hello Vue

- 安装: `npm i -S vue`

```
<!-- 指定vue管理内容区域，需要通过vue展示的内容都要放到找个元素中 通常我们也把它叫做边界 数据只在边界内部解析-->
<div id="app">{{ msg }}</div>

<!-- 引入 vue.js -->
<script src="vue.js"></script>

<!-- 使用 vue -->
<script>
var vm = new Vue({
  // el: 提供一个在页面上已存在的 DOM 元素作为 Vue 实例的挂载目标
  el: '#app',
  // Vue 实例的数据对象，用于给 View 提供数据
  data: {
    msg: 'Hello Vue'
  }
})
</script>
```

Vue实例

- 注意 1: 先在data中声明数据，再使用数据
- 注意 2: 可以通过 `vm.$data` 访问到data中的所有属性，或者 `vm.msg`

```
var vm = new Vue({
  data: {
    msg: '大家好，...'
  }
})

vm.$data.msg === vm.msg // true
```

数据绑定

- 最常用的方式: `Mustache(插值语法)`，也就是 `{{}}` 语法
- 解释: `{{}}` 从数据对象 `data` 中获取数据
- 说明: 数据对象的属性值发生了改变，插值处的内容都会更新
- 说明: `{{}}` 中只能出现JavaScript表达式 而不能解析js语句
- 注意: **Mustache 语法不能作用在 HTML 元素的属性上**

```
<h1>Hello, {{ msg }}.</h1>
<p>{{ 1 + 2 }}</p>
<p>{{ isOk ? 'yes' : 'no' }}</p>

<!-- ! ! ! 错误示范! ! ! -->
<h1 title="{{ err }}"></h1>
```

双向数据绑定 Vue two way data binding

- 双向数据绑定：将DOM与Vue实例的data数据绑定到一起，彼此之间相互影响
 - 数据的改变会引起DOM的改变
 - DOM的改变也会引起数据的变化
- 原理：`Object.defineProperty` 中的 `get` 和 `set` 方法
 - `getter` 和 `setter`：访问器
 - 作用：指定 `读取或设置` 对象属性值的时候，执行的操作
- [Vue - 深入响应式原理](#)
- [MDN - Object.defineProperty\(\)](#)

```
/* defineProperty语法 介绍 */
var obj = {}
Object.defineProperty(obj, 'msg', {
  // 设置 obj.msg = "1" 时set方法会被系统调用 参数分别是设置后和设置前的值
  set: function (newVal, oldVal) { },
  // 读取 obj.msg 时get方法会被系统调用
  get: function (newVal, oldVal) {}
})
```

Vue双向绑定的极简实现

- [剖析Vue原理&实现双向绑定MVVM](#)

```
<!-- 示例 -->
<input type="text" id="txt" />
<span id="sp"></span>

<script>
var txt = document.getElementById('txt'),
    sp = document.getElementById('sp'),
    obj = {}

// 给对象obj添加msg属性，并设置setter访问器
Object.defineProperty(obj, 'msg', {
  // 设置 obj.msg 当obj.msg发生改变时set方法将会被调用
  set: function (newVal) {
    // 当obj.msg被赋值时 同时设置给 input/span
    txt.value = newVal
    sp.innerText = newVal
  }
})

// 监听文本框的改变 当文本框输入内容时 改变obj.msg
txt.addEventListener('keyup', function (event) {
  obj.msg = event.target.value
})
```

```
})
</script>
```

动态添加数据的注意点

- 注意：只有 `data` 中的数据才是响应式的，动态添加进来的数据默认为非响应式
- 可以通过以下方式实现动态添加数据的响应式
 - 1 `Vue.set(object, key, value)` - 适用于添加单个属性
 - 2 `Object.assign()` - 适用于添加多个属性

```
var vm = new Vue({
  data: {
    stu: {
      name: 'jack',
      age: 19
    }
  }
})

/* Vue.set */
Vue.set(vm.stu, 'gender', 'male')

/* Object.assign 将参数中的所有对象属性和值 合并到第一个参数 并返回合并后的对象*/
vm.stu = Object.assign({}, vm.stu, { gender: 'female', height: 180 })
```

异步DOM更新

- 说明：Vue 异步执行 DOM 更新，监视所有数据改变，一次性更新DOM
- 优势：可以去除重复数据，对于避免不必要的计算和 避免重复 DOM 操作上，非常重要
- 如果需要那到更新后dom中的数据 则需要通过 `Vue.nextTick(callback)`：在DOM更新后，执行某个操作（属于DOM操作）
 - 实例调用 `vm.$nextTick(function () {})`

```
methods: {
  fn() {
    this.msg = 'change'
    this.$nextTick(function () {
      console.log('$nextTick中打印：', this.$el.children[0].innerText);
    })
    console.log('直接打印：', this.$el.children[0].innerText);
  }
}
```

指令

- 解释：指令 (Directives) 是带有 `v-` 前缀的特殊属性
- 作用：当表达式的值改变时，将其产生的连带影响，响应式地作用于 DOM

v-text

- 解释：更新DOM对象的 `textContent`

```
<h1 v-text="msg"></h1>
```

v-html

- 解释：更新DOM对象的 `innerHTML`

```
<h1 v-html="msg"></h1>
```

v-bind

- 作用：当表达式的值改变时，将其产生的连带影响，响应式地作用于 DOM
- 语法：`v-bind:title="msg"`
- 简写：`:title="msg"`

```
<!-- 完整语法 -->
<a v-bind:href="url"></a>
<!-- 缩写 -->
<a :href="url"></a>
```

v-on

- 作用：绑定事件
- 语法：`v-on:click="say" or v-on:click="say('参数', $event)"`
- 简写：`@click="say"`
- 说明：绑定的事件定义在 `methods`

```
<!-- 完整语法 -->
<a v-on:click="doSomething"></a>
<!-- 缩写 -->
<a @click="doSomething"></a>
```

事件修饰符

- `.stop` 阻止冒泡，调用 `event.stopPropagation()`
- `.prevent` 阻止默认行为，调用 `event.preventDefault()`
- `.capture` 添加事件侦听器时使用事件 `捕获` 模式
- `.self` 只当事件在该元素本身（比如不是子元素）触发时，才会触发事件

- `.once` 事件只触发一次

v-model

- 作用：在表单元素上创建双向数据绑定
- 说明：监听用户的输入事件以更新数据
- 案例：计算器

```
<input type="text" v-model="message" placeholder="edit me">
<p>Message is: {{ message }}</p>
```

v-for

- 作用：基于源数据多次渲染元素或模板块

```
<!-- 1 基础用法 -->
<div v-for="item in items">
  {{ item.text }}
</div>

<!-- item 为当前项，index 为索引 -->
<p v-for="(item, index) in list">{{item}} -- {{index}}</p>
<!-- item 为值，key 为键，index 为索引 -->
<p v-for="(item, key, index) in obj">{{item}} -- {{key}}</p>
<p v-for="item in 10">{{item}}</p>
```

key属性

- 推荐：使用 `v-for` 的时候提供 `key` 属性，以获得性能提升。
- 说明：使用 `key`，VUE会基于 `key` 的变化重新排列元素顺序，并且会移除 `key` 不存在的元素。
- [vue key](#)
- [vue key属性的说明](#)

```
<div v-for="item in items" :key="item.id">
  <!-- 内容 -->
</div>
```

样式处理 -class和style

- 使用方式：`v-bind:class="expression"` or `:class="expression"`
- 表达式的类型：字符串、数组、对象（重点）
- 语法：

```
<!-- 1 -->
<div v-bind:class="{ active: true }"></div> ===> 解析后
<div class="active"></div>
```

```

<!-- 2 -->
<div :class="['active', 'text-danger']"></div> ===>解析后
<div class="active text-danger"></div>

<!-- 3 -->
<div v-bind:class="[{ active: true }, errorClass]"></div> ===>解析后
<div class="active text-danger"></div>

--- style ---
<!-- 1 -->
<div v-bind:style="{ color: activeColor, 'font-size': fontSize + 'px' }"></div>
<!-- 2 将多个 样式对象 应用到一个元素上-->
<!-- baseStyles 和 overridingStyles 都是data中定义的对象 -->
<div v-bind:style="[baseStyles, overridingStyles]"></div>

```

v-if 和 v-show

- [条件渲染](#)
- [v-if](#)：根据表达式的值的真假条件，销毁或重建元素
- [v-show](#)：根据表达式之真假值，切换元素的 display CSS 属性

```

<p v-show="isShow">这个元素展示出来了吗？？？</p>
<p v-if="isShow">这个元素，在HTML结构中吗？？？</p>

```

提升性能：v-pre

- 说明：vue会跳过这个元素和它的子元素的编译过程。可以用来显示原始 Mustache 标签。跳过大段没有指令的节点会加快编译。

```
<span v-pre>{{ this will not be compiled }}</span>
```

提升性能：v-once

- 说明：vue只渲染元素和组件一次。随后的重新渲染，元素/组件及其所有的子节点将被视为静态内容并跳过。这可以用于优化更新性能。

```
<span v-once>This will never change: {{msg}}</span>
```

过滤器 filter

- 作用：文本数据格式化
- 过滤器可以用在两个地方：[{{}}](#) 和 v-bind 表达式
- 两种过滤器：1 全局过滤器 2 局部过滤器

全局过滤器

- 说明：通过全局方式创建的过滤器，在任何一个vue实例中都可以使用
- 注意：使用全局过滤器的时候，需要先创建全局过滤器，再创建Vue实例
- 显示的内容由过滤器的返回值决定

```
Vue.filter('filterName', function (value) {
  // value 表示要过滤的内容
})

• 示例：

<div>{{ dateStr | date }}</div>
<div>{{ dateStr | date('YYYY-MM-DD hh:mm:ss') }}</div>

<script>
  Vue.filter('date', function(value, format) {
    // value 要过滤的字符串内容，比如: dateStr
    // format 过滤器的参数，比如: 'YYYY-MM-DD hh:mm:ss'
  })
</script>
```

局部过滤器

- 说明：局部过滤器是在某一个vue实例的内容创建的，只在当前实例中起作用

```
{
  data: {},
  // 通过 filters 属性创建局部过滤器
  // 注意：此处为 filters
  filters: {
    filterName: function(value, format) {}
  }
}
```

按键值修饰符

- 说明：在监听键盘事件时，Vue 允许为 `v-on` 在监听键盘事件时添加关键修饰符
- [键盘事件 - 键值修饰符](#)
- 其他：修饰键（.ctrl等）、鼠标按键修饰符（.left等）

```
// 只有在 keyCode 是 13 时调用 vm.submit()
@keyup.13="submit"
// 使用全局按键别名
@keyup.enter="add"

---
```

```
// 通过全局 config.keyCode 对象自定义键值修饰符别名
Vue.config.keyCode.f2 = 113
// 使用自定义键值修饰符
@keyup.enter.f2="add"
```

监视数据变化 - watch

- 概述： `watch` 是一个对象，键是需要观察的表达式，值是对应回调函数
- 作用：当表达式的值发生变化后，会调用对应的回调函数完成响应的监视操作
- [VUE \\$watch](#)

```
new Vue({
  data: { a: 1, b: { age: 10 } },
  watch: {
    a: function(val, oldVal) {
      // val 表示当前值
      // oldVal 表示旧值
      console.log('当前值为: ' + val, '旧值为: ' + oldVal)
    },
    // 监听对象属性的变化
    b: {
      handler: function (val, oldVal) { /* ... */ },
      // deep : true 表示是否监听对象内部属性值的变化
      deep: true
    },
    // 只监视 user 对象中 age 属性的变化
    'user.age': function (val, oldVal) {
    },
  }
})
```

计算属性

- 说明：计算属性是基于它们的依赖进行缓存的，只有在它的依赖发生改变时才会重新求值
- 注意：Mustache语法 (`{}`) 中不要放入太多的逻辑，否则会让模板过重、难以理解和维护
- 注意： `computed` 中的属性不能与 `data` 中的属性同名，否则会报错
- [Vue computed 属性原理](#)

```
var vm = new Vue({
  el: '#app',
  data: {
    firstname: 'jack',
    lastname: 'rose'
  },
  computed: {
    fullname() {
      return this.firstname + '.' + this.lastname
    }
  }
})
```

```
    }  
}  
})
```

实例生命周期

- 所有的 Vue 组件都是 Vue 实例，并且接受相同的选项对象即可 (一些根实例特有的选项除外)。
- 实例生命周期也叫做：组件生命周期

生命周期介绍

- [vue生命周期钩子函数](#)
- 简单说：一个组件从开始到最后消亡所经历的各种状态，就是一个组件的生命周期

生命周期钩子函数的定义：从组件被创建，到组件挂载到页面上运行，再到页面关闭组件被卸载，这三个阶段总是伴随着组件各种各样的事件，这些事件，统称为组件的生命周期函数！

- 注意：Vue在执行过程中会自动调用 [生命周期钩子函数](#)，我们只需要提供这些钩子函数即可
- 注意：钩子函数的名称都是Vue中规定好的！

钩子函数 - beforeCreate()

- 说明：在实例初始化之后，数据观测 (data observer) 和 event/watcher 事件配置之前被调用
- 注意：此时，无法获取 data中的数据、methods中的方法

钩子函数 - created()

- 注意：这是一个常用的生命周期，可以调用methods中的方法、改变data中的数据
- [vue实例生命周期 参考1](#)
- [vue实例生命周期 参考2](#)
- 使用场景：发送请求获取数据

钩子函数 - beforeMounted()

- 说明：在挂载开始之前被调用

钩子函数 - mounted()

- 说明：此时，vue实例已经挂载到页面中，可以获取到el中的DOM元素，进行DOM操作

钩子函数 - beforeUpdated()

- 说明：数据更新时调用，发生在虚拟 DOM 重新渲染和打补丁之前。你可以在这个钩子中进一步地更改状态，这不会触发附加的重渲染过程。
- 注意：此处获取的数据是更新后的数据，但是获取页面中的DOM元素是更新之前的

钩子函数 - updated()

- 说明：组件 DOM 已经更新，所以你现在可以执行依赖于 DOM 的操作。

钩子函数 - beforeDestroy()

- 说明：实例销毁之前调用。在这一步，实例仍然完全可用。
- 使用场景：实例销毁之前，执行清理任务，比如：清除定时器等

钩子函数 - destroyed()

- 说明：Vue 实例销毁后调用。调用后，Vue 实例指示的所有东西都会解绑定，所有的事件监听器会被移除，所有的子实例也会被销毁。

axios

- Promise based HTTP client for the browser and node.js
 - 以Promise为基础的HTTP客户端，适用于：浏览器和node.js
 - 封装ajax，用来发送请求，异步获取数据
- 安装：`npm i -S axios`
- [axios](#)

// 在浏览器中使用，直接引入js文件使用下面的GET/POST请求方式即可

```
// 1 引入 axios.js
// 2 直接调用axios提供的API发送请求
created: function () {
  axios.get(url)
    .then(function(resp) {})
}
```

```
// 配合 webpack 使用方式如下：
import Vue from 'vue'
import axios from 'axios'
// 将 axios 添加到 Vue.prototype 中
Vue.prototype.$axios = axios
```

```
// 在组件中使用：
methods: {
  getData() {
    this.$axios.get('url')
      .then(res => {})
      .catch(err => {})
  }
}
```

// API使用方式：

```
axios.get(url[, config])
axios.post(url[, data[, config]])
axios(url[, config])
axios(config)
```

Get 请求

```
const url = 'http://vue.studyit.io/api/getnewslist'

// url中带有query参数
axios.get('/user?id=89')
  .then(function (response) {
    console.log(response);
  })
  .catch(function (error) {
    console.log(error);
  });

// url和参数分离，使用对象
axios.get('/user', {
  params: {
    id: 12345
  }
})
```

Post 请求

- [不同环境中处理 POST 请求](#)
- 默认情况下， axios 会将JS对象序列化为JSON对象。为了使用 `application/x-www-form-urlencoded` 格式发送请求，我们可以这样：

```
// 使用 qs 包，处理将对象序列化为字符串
// npm i -S qs
// var qs = require('qs')
import qs from 'qs'
qs.stringify({ 'bar': 123 }) ===> "bar=123"
axios.post('/foo', qs.stringify({ 'bar': 123 }))

// 或者：
axios.post('/foo', 'bar=123&age=19')

const url = 'http://vue.studyit.io/api/postcomment/17'
axios.post(url, 'content=点个赞不过份')

axios.post('/user', qs.stringify({
  firstName: 'Fred',
  lastName: 'Flintstone'
}))
```

```
.then(function (response) {
  console.log(response);
})
.catch(function (error) {
  console.log(error);
});
```

全局配置

```
// 设置请求公共路径:
axios.defaults.baseURL = 'http://vue.studyit.io'
```

拦截器

- 拦截器会拦截发送的每一个请求，请求发送之前执行 `request` 中的函数，请求发送完成之后执行 `response` 中的函数

```
// 请求拦截器
axios.interceptors.request.use(function (config) {
  // 所有请求之前都要执行的操作

  return config;
}, function (error) {
  // 错误处理

  return Promise.reject(error);
});

// 响应拦截器
axios.interceptors.response.use(function (response) {
  // 所有请求完成后都要执行的操作

  return response;
}, function (error) {
  // 错误处理
  return Promise.reject(error);
});
```

自定义指令

- 作用：进行DOM操作
- 使用场景：对纯 DOM 元素进行底层操作，比如：文本框获得焦点
- [vue 自定义指令用法实例](#)
- 两种指令：1 全局指令 2 局部指令

全局自定义指令

```

// 第一个参数: 指令名称
// 第二个参数: 配置对象, 指定指令的钩子函数
Vue.directive('directiveName', {
  // bind中只能对元素自身进行DOM操作, 而无法对父级元素操作
  // 只调用一次 指令第一次绑定到元素时调用。在这里可以进行一次性的初始化设置。
  bind( el, binding, vnode ) {
    // 参数详解
    // el: 指令所绑定的元素, 可以用来直接操作 DOM 。
    // binding: 一个对象, 包含以下属性:
    //   name: 指令名, 不包括 v- 前缀。
    //   value: 指令的绑定值, 等号后面的值。
    //   oldValue: 指令绑定的前一个值, 仅在 update 和 componentUpdated 钩子中可用。无论值是否改变都可用。
    //   expression: 字符串形式的指令表达式 等号后面的字符串 形式
    //   arg: 传给指令的参数, 可选。例如 v-my-directive:foo 中, 参数为 "foo"。
    //   modifiers: 指令修饰符。例如: v-directive.foo.bar中, 修饰符对象为 { foo: true, bar: true }。
    //   vnode: Vue 编译生成的虚拟节点。
    //   oldVnode: 上一个虚拟节点, 仅在 update 和 componentUpdated 钩子中可用。
  },
  // inserted这个钩子函数调用的时候, 当前元素已经插入页面中了, 也就是说可以获取到父级节点了
  inserted ( el, binding, vnode ) {},
  // DOM重新渲染前
  update(el, binding, vnode,oldVnode) {},
  // DOM重新渲染后
  componentUpdated ( el, binding, vnode,oldVnode ) {},
  // 只调用一次, 指令与元素解绑时调用
  unbind ( el ) {
    // 指令所在的元素在页面中消失, 触发
  }
})
// 简写 如果你想在 bind 和 update 时触发相同行为, 而不关心其它的钩子:
Vue.directive('自定义指令名', function( el, binding ) {})
// 例:
Vue.directive('color', function(el, binding) {
  el.style.color = binding.value
})
// 使用 注意直接些会被当成data中的数据“red” 需要字符串则嵌套引号"'red'"
<p v-color="'red'"></p>

```

局部自定义指令

```

var vm = new Vue({
  el : "#app",
  directives: {
    directiveName: { }
  }
})

```

- [vue 剖析Vue原理&实现双向绑定MVVM](#)

组件

组件系统是 Vue 的另一个重要概念，因为它是一种抽象，允许我们使用小型、独立和通常可复用的组件构建大型应用。仔细想想，几乎任意类型的应用界面都可以抽象为一个组件树

- 创建组件的两种方式：1 全局组件 2 局部组件

全局组件

- 说明：全局组件在所有的vue实例中都可以使用
- 注意：**先注册组件，再初始化根实例**

```
// 1 注册全局组件
Vue.component('my-component', {
  // template 只能有一个根元素
  template: '<p>A custom component!</p>',
  // 组件中的 `data` 必须是函数 并且函数的返回值必须是对象
  data() {
    return {
      msg: '注意：组件的data必须是一个函数！！！'
    }
  }
})
```

// 2 使用：以自定义元素的方式

```
<div id="example">
  <my-component></my-component>
</div>
```

// =====> 渲染结果

```
<div id="example">
  <p>A custom component!</p>
</div>
```

// 3 template属性的值可以是：

- 1 模板字符串
 - 2 模板id template: '#tpl'
- ```
<script type="text/x-template" id="tpl">
 <p>A custom component!</p>
</script>
```

- **extend**：使用基础 Vue 构造器，创建一个“子类”。参数是一个包含组件选项的对象。

// 注册组件，传入一个扩展过的构造器

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

// 注册组件，传入一个选项对象（自动调用 Vue.extend）

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

```
var Home = Vue.extend({
 template: '',
 data() {}
```

```
})
Vue.component('home', Home)
```

## 局部组件

- 说明：局部组件，是在某一个具体的vue实例中定义的，只能在这个vue实例中使用

```
var Child = {
 template: '<div>A custom component!</div>'
}

new Vue({
 // 注意：此处为 components
 components: {
 // <my-component> 将只在当前vue实例中使用
 // my-component 为组件名 值为配置对象
 'my-component': {
 template: ``,
 data () { return { } },
 props : []
 }
 }
})
```

## is特性

在某些特定的标签中只能存在指定表恰 如ul > li 如果要浏览器正常解析则需要使用is

```
<!-- 案例 -->
<ul id="app">
 <!-- 不能识别 -->
 <my-li></my-li>
 正常识别
 <li is="my-li">

<script>
 var vm = new Vue({
 el: "#app",
 components : {
 myLi : {
 template : `内容`
 }
 }
 })
</script>
```

## 组件通讯

## 父组件到子组件

- 方式：通过子组件 `props` 属性来传递数据 `props` 是一个数组
- 注意：属性的值必须在组件中通过 `props` 属性显示指定，否则，不会生效
- 说明：传递过来的 `props` 属性的用法与 `data` 属性的用法相同

```
<div id="app">
 <!-- 如果需要往子组件总传递父组件data中的数据 需要加v-bind="数据名称" -->
 <hello v-bind:msg="info"></hello>
 <!-- 如果传递的是字面量 那么直接写-->
 <hello my-msg="abc"></hello>
</div>

<!-- js -->
<script>
 new Vue({
 el: "#app",
 data : {
 info : 15
 },
 components: {
 hello: {
 // 创建props及其传递过来的属性
 props: ['msg', 'myMsg'],
 template: '<h1>这是 hello 组件，这是消息: {{msg}} --- {{myMsg}}</h1>'
 }
 }
 })
</script>
```

## 子组件到父组件

方式：父组件给子组件传递一个函数，由子组件调用这个函数

- 说明：借助vue中的自定义事件 (`v-on:customFn="fn"`)

步骤：

- 1、在父组件中定义方法 `parentFn`
- 2、在子组件 组件引入标签 中绑定自定义事件 `v-on:自定义事件名="父组件中的方法" ==> @pfn="parentFn"`
- 3、子组件中通过 `$emit()` 触发自定义事件事件 `this.$emit(pfn,参数列表。。。)`

```
<hello @pfn="parentFn"></hello>

<script>
 Vue.component('hello', {
 template: '<button @click="fn">按钮</button>',
 methods: {
 // 子组件：通过$emit调用
```

```

fn() {
 this.$emit('pfn', '这是子组件传递给父组件的数据')
}
})
new Vue({
 methods: {
 // 父组件: 提供方法
 parentFn(data) {
 console.log('父组件: ', data)
 }
 }
})
</script>

```

## 非父子组件通讯

在简单的场景下，可以使用一个空的 Vue 实例作为事件总线

- `$on()` : 绑定自定义事件

```

var bus = new Vue()

// 在组件 B 绑定自定义事件
bus.$on('id-selected', function (id) {
 // ...
})

// 触发组件 A 中的事件
bus.$emit('id-selected', 1)

```

- 示例：组件A ---> 组件B

```

<!-- 组件A: -->
<com-a></com-a>
<!-- 组件B: -->
<com-b></com-b>

<script>
 // 中间组件
 var bus = new Vue()
 // 通信组件
 var vm = new Vue({
 el: '#app',
 components: {
 comB: {
 template: '<p>组件A告诉我: {{msg}}</p>',
 data() {
 return {
 msg: ''
 }
 },
 props: []
 }
 }
 })

```

```

created() {
 // 给中间组件绑定自定义事件 注意:如果用到this 需要用箭头函数
 bus.$on('tellComB', (msg) => {
 this.msg = msg
 })
},
comA: {
 template: '<button @click="emitFn">告诉B</button>',
 methods: {
 emitFn() {
 // 触发中间组件中的自定义事件
 bus.$emit('tellComB', '土豆土豆我是南瓜')
 }
 }
}
})
</script>

```

## 内容分发

- 通过<slot></slot> 标签指定内容展示区域

案例:

```

<!-- html代码 -->
<div id="app">
 <hello>
 <!-- 如果只有一个slot插槽 那么不需要指定名称 -->
 <p slot="插槽名称">我是额外的内容</p>
 </hello>
</div>

```

```

// js代码
new vue({
 el : "#app",
 components : {
 hello : {
 template : `
 <div>
 <p>我是子组件中的内容</p>
 <slot name="名称"></slot>
 </div>
 `
 }
 }
})

```

## 获取组件（或元素） - refs

- 说明: `vm.$refs` 一个对象, 持有已注册过 ref 的所有子组件 (或HTML元素)
- 使用: 在 HTML元素 中, 添加 `ref` 属性, 然后在JS中通过 `vm.$refs.属性` 来获取
- 注意: 如果获取的是一个子组件, 那么通过ref就能获取到子组件中的data和methods

```
<div id="app">
 <div ref="dv"></div>
 <my res="my"></my>
</div>

<!-- js -->
<script>
 new Vue({
 el : "#app",
 mounted() {
 this.$refs.dv //获取到元素
 this.$refs.my //获取到组件
 },
 components : {
 my : {
 template: `<a>sss`
 }
 }
 })
</script>
```

## SPA -单页应用程序

---

### SPA: Single Page Application

单页Web应用 (single page application, SPA), 就是只有一个Web页面的应用, 是加载单个HTML页面, 并在用户与应用程序交互时动态更新该页面的Web应用程序。

- 单页面应用程序:

- 只有第一次会加载页面, 以后的每次请求, 仅仅是获取必要的数据.然后, 由页面中js解析获取的数据, 展示在页面中

- 传统多页面应用程序:

- 对于传统的多页面应用程序来说, 每次请求服务器返回的都是一个完整的页面

### 优势

- 1 减少了请求体积, 加快页面响应速度, 降低了对服务器的压力
- 2 更好的用户体验, 让用户在web app感受native app的流畅

### 实现思路和技术点

- 1 ajax
- 2 锚点的使用 (`window.location.hash #`)
- 3 hashchange 事件 `window.addEventListener("hashchange",function () {})`
- 4 监听锚点值变化的事件，根据不同的锚点值，请求相应的数据
- 5 原本用作页面内部进行跳转，定位并展示相应的内容

## 路由

- 路由即：浏览器中的哈希值 (# hash) 与展示视图内容 (template) 之间的对应规则
- vue中的路由是：hash 和 component的对应关系

在 Web app 中，通过一个页面来展示和管理整个应用的功能。

SPA往往是功能复杂的应用，为了有效管理所有视图内容，前端路由 应运而生！

简单来说，路由就是一套映射规则（一对一的对应规则），由开发人员制定规则。

当URL中的哈希值 (# hash) 发生改变后，路由会根据制定好的规则，展示对应的视图内容

## 基本使用

- 安装：`npm i -S vue-router`

```
<div id="app">
 <!-- 5 路由入口 指定跳转到只定入口 -->
 <router-link to="/home">首页</router-link>
 <router-link to="/login">登录</router-link>

 <!-- 7 路由出口：用来展示匹配路由视图内容 -->
 <router-view></router-view>
</div>

<!-- 1 导入 vue.js -->
<script src="./vue.js"></script>
<!-- 2 导入 路由文件 -->
<script src="./node_modules/vue-router/dist/vue-router.js"></script>
<script>

 // 3 创建两个组件
 const Home = Vue.component('home', {
 template: '<h1>这是 Home 组件</h1>'
 })
 const Login = Vue.component('login', {
 template: '<h1>这是 Login 组件</h1>'
 })

 // 4 创建路由对象
 const router = new VueRouter({
 routes: [
 // 路径和组件一一对应
 { path: '/home', component: Home },
 { path: '/login', component: Login }
]
 })

```

```

var vm = new Vue({
 el: '#app',
 // 6 将路由实例挂载到vue实例
 router
})
</script>

```

## 重定向

```

// 将path 重定向到 redirect
{ path: '/', redirect: '/home' }

```

## 路由其他配置

- 路由导航高亮
  - 说明：当前匹配的导航链接，会自动添加router-link-exact-active router-link-active类
  - 配置：linkActiveClass
- 匹配路由模式
  - 配置：mode

```

new Router({
 routers:[],
 mode: "hash", //默认hash | history 可以达到隐藏地址栏hash值 | abstract, 如果发现没有浏览器的 API 则强制进入
 linkActiveClass : "now" //当前匹配的导航链接将被自动添加now类
})

```

## 路由参数

- 说明：我们经常需要把某种模式匹配到的所有路由，全都映射到同一个组件，此时，可以通过路由参数来处理
- 语法：/user/:id
- 使用：当匹配到一个路由时，参数值会被设置到 this.\$route.params
- 其他：可以通过 \$route.query 获取到 URL 中的查询参数 等

```

// 方式一
<router-link to="/user/1001">如果你需要在模版中使用路由参数 可以这样 {{ $route.params.id }}</router-link>
// 方式二
<router-link :to="{path: '/user', query: {name: 'jack', age: 18}}>用户 Rose</router-link>

<script>
// 路由
var router = new Router({
 routers : [

```

```
// 方式一 注意 只有/user/1001这种形式能被匹配 /user | /user/ | /user/1001/ 都不能被匹配
// 将来通过$router.params获取参数返回 {id:1001}
{ path: '/user/:id', component: User },
// 方式二
{ path: "user" , component: User}

])
})

// User组件:
const User = {
 template: `<div>User {{ $route.params.id }}</div>`
}
</script>
<!-- 如果要在vue实例中获取路由参数 则使用this.$router.params 获取路由参数对象 -->
<!-- {{ $router.query}} 获取路由中的查询字符串 返回对象 --></pre>

```

## 嵌套路 - 子路由

- 路由是可以嵌套的，即：路由中又包含子路由
- 规则：父组件中包含 router-view，在路由规则中使用 children 配置

```
// 父组件:
const User = Vue.component('user', {
 template: `
 <div class="user">
 <h2>User Center</h2>
 <router-link to="/user/profile">个人资料</router-link>
 <router-link to="/user/posts">岗位</router-link>
 <!-- 子路由展示在此处 -->
 <router-view></router-view>
 </div>
 `
})

// 子组件[简写]
const UserProfile = {
 template: '<h3>个人资料: 张三</h3>'
}
const UserPosts = {
 template: '<h3>岗位: FE</h3>'
}

// 路由
var router =new Router({
 routers : [
 { path: '/user', component: User,
 // 子路由配置:
 children: [
 {
 // 当 /user/profile 匹配成功,
 // UserProfile 会被渲染在 User 的 <router-view> 中
 path: 'profile',
 }
]
 }
]
})
```

```

 component: UserProfile
 },
 {
 // 当 /user/posts 匹配成功
 // UserPosts 会被渲染在 User 的 <router-view> 中
 path: 'posts',
 component: UserPosts
 }
]
}
])
})

```

## 前端模块化

### 为什么需要模块化

- 1 最开始的js就是为了实现客户端验证以及一些简单的效果
- 2 后来，js得到重视，应用越来越广泛，前端开发的复杂度越来越高
- 3 旧版本的js中没有提供与模块（module）相关的内容

## 模块的概念

- 在js中，一个模块就是实现特定功能的文件（js文件）
- 遵循模块的机制，想要什么功能就加载什么模块
- 模块化开发需要遵循规范

## 模块化解决的问题

- 1 命名冲突
- 2 文件依赖（加载文件）
- 3 模块的复用
- 4 统一规范和开发方式

## JS实现模块化的规范

- AMD 浏览器端
  - requirejs
- CommonJS nodejs
  - 加载模块：require()
  - 导出模块：module.exports = {} / exports = {}
- ES6 中的 import / export

- CMD 浏览器端
  - 玉伯（阿里前端大神） -> seajs
- UMD 通用模块化规范，可以兼容 AMD、CommonJS、浏览器中没有模块化规范 等这些语法

## AMD 的使用

**Asynchronous Module Definition**: 异步模块定义，浏览器端模块开发的规范 代表：require.js 特点：  
模块被异步加载，模块加载不影响后面语句的运行

### 1、定义模块

```
// 语法:define(name, dependencies?, factory);
// name表示: 当前模块的名称, 是一个字符串 可有可无
// dependencies表示: 当前模块的依赖项, 是一个数组无论依赖一项还是多项 无则不写
// factory表示: 当前模块要完成的一些功能, 是一个函数

// 定义对象模块
define({})

// 定义方法模块
define(function() {
 return {}
})

// 定义带有依赖项的模块
define(['js/a'], function() {})
```

### 2、加载模块

```
// - 注意: require的第一个参数必须是数组

// 参数必须是数组 表示模块路径 以当前文件为基准,通过回调函数中的参数获取加载模块中的变量 参数与模块按照顺序一一对应
require(['a', 'js/b'], function(a, b) {
 // 使用模块a 和 模块b 中的代码
})
```

### 3、路径查找配置

- requirejs 默认使用 baseUrl+paths 的路径解析方式
- 可以使用以下方式避开此设置：
  - 1 以.js结尾
  - 2 以 / 开始
  - 3 包含协议: <https://> 或 <http://>

```
// 配置示例
// 注意配置应当在使用之前
require.config({}
```

```

 baseUrl: './js' // 配置基础路径为: 当前目录下的js目录
 })
 require(['a']) // 查找 基础路径下的 ./js/a.js

// 简化加载模块路径
require.config({
 baseUrl: './js',
 // 配置一次即可, 直接通过路径名称 (template || jquery) 加载模块
 paths: {
 template: 'assets/artTemplate/template-native',
 jquery: 'assets/jquery/jquery.min'
 }
})
// 加载jquery template模块
require(['jquery', 'template'])

```

## 4、非模块化和依赖项支持

- 1 添加模块的依赖模块, 保证加载顺序 (deps)
- 2 将非模块化模块, 转化为模块化 (exports)

```

// 示例
require.config({
 baseUrl: './js',
 paths: {
 // 配置路径
 noModule: 'assets/demo/noModule'
 },
 // 配置不符合规范的模块项
 shim: {
 // 模块名称
 noModule: {
 deps: [], // 依赖项
 exports: 'sayHi' // 导出模块中存在的函数或变量
 }
 }
});

// 注意点 如果定义模块的时候, 指定了模块名称, 需要使用该名称来引用模块
// 定义 这个模块名称与paths中的名称相同
define('moduleA', function() {})
// 导入
require.config({
 paths: {
 // 此处的模块名: moduleA
 moduleA: 'assets/demo/moduleA'
 }
})

```

## 5、路径加载规则

- 路径配置的优先级:

- 1 通过 config 配置规则查找
- 2 通过 data-main 指定的路径查找
- 3 以引入 requirejs 的页面所在路径为准查找

```
<!--
设置data-main属性
1 data-main属性指定的文件也会同时被加载
2 用于指定查找其他模块的基础路径
-->
<script src="js/require.js" data-main="js/main"></script>
```

## Webpack

- [webpack 官网](#)
- bundle ['**ndl**'] 捆绑，收集，归拢，把...塞入

1 webpack 将带有依赖项的各个模块打包处理后，变成了独立的浏览器能够识别的文件  
 2 webpack 合并以及解析带有依赖项的模块

### 概述

webpack 是一个现代 JavaScript 应用程序的模块打包器(特点 module、 bundler)  
 webpack 是一个模块化方案 (预编译)  
 webpack 获取具有依赖关系的模块，并生成表示这些模块的静态资源

- 四个核心概念：入口(entry)、输出(output)、加载器loader、插件(plugins)

对比

模块化方案：webpack 和 requirejs (通过编写代码的方式将前端的功能，划分成独立的模块)

browserify 是与 webpack 相似的模块化打包工具

webpack 预编译 (在开发阶段通过webpack进行模块化处理，最终项目上线，就不在依赖于 webpack)  
 requirejs 线上的编译( 代码运行是需要依赖与 requirejs 的 )

## webpack起源

- webpack解决了现存模块打包器的两个痛点：
  - 1 **Code Splitting** - 代码分离 按需加载
  - 2 **静态资源的模块化处理方案**

## webpack与模块

- [前端模块系统的演进](#)

- 在webpack看来：所有的静态资源都是模块
- webpack 模块能够识别以下等形式的模块之间的依赖：
- JS的模块化规范：

- ES2015 `import export`
- CommonJS `require() module.exports`
- AMD `define` 和 `require`

- 非JS等静态资源：

- css/sass/less 文件中的 `@import`
- 图片连接，比如：样式 `url(...)` 或 HTML `<img src=...>`
- 字体 等

## webpack文档和资源

- [webpack 中文网](#)
  - [webpack 1.0](#)
  - [webpack 2.x+](#)
  - [入门Webpack, 看这篇就够了](#)
- 

## 安装webpack

- 全局安装：`npm i -g webpack`
  - 目的：在任何目录中通过CLI使用 `webpack` 这个命令
- 项目安装：`npm i -D webpack`
  - 目的：执行当前项目的构建

## webpack的基本使用

- 安装：`npm i -D webpack`
- webpack的两种使用方式：1 命令行 2 配置文件（`webpack.config.js`）

## 命令行方式演示 - 案例：隔行变色

- 1 使用 `npm init -y` 初始化`package.json`，使用npm来管理项目中的包
- 2 新建 `index.html` 和 `index.js`，实现隔行变色功能
- 3 运行 `webpack src/js/index.js dist/bundle.js` 进行打包构建，语法是：`webpack` 入口文件 输出文件
- 4 注意：需要在页面中引入 输出文件 的路径（此步骤可通过配置webpack去掉）

```

/*
src/js/index.js
*/

// 1 导入 jQuery
import $ from 'jquery'
// 2 获取页面中的li元素
const $lis = $('#ulList').find('li')
// 3 隔行变色
// jQuery中的 filter() 方法用来过滤jquery对象
$lis.filter(':odd').css('background-color', '#def')
$lis.filter(':even').css('background-color', 'skyblue')

//命令行运行 `webpack src/js/index.js dist/bundle.js` 目录生成在命令行运行目录
/*
运行流程：
1、webpack 根据入口找到入口文件
2、分析js中的模块化语法
3、将所有关联文件 打包合并输出到出口
*/

```

## webpack-dev-server 配置

---

### 一、 package.json配置方式

- 安装: `npm i -D webpack-dev-server`
- 作用: 配合webpack, 创建开发环境 (启动服务器、监视文件变化、自动编译、刷新浏览器等), 提高开发效率
- 注意: 无法直接在终端中执行 `webpack-dev-server`, 需要通过 `package.json` 的 `scripts` 实现
- 使用方式: `npm run dev`

```

// 参数解释 注意参数是无序的 有值的参数空格隔开
// --open 自动打开浏览器
// --contentBase ./ 指定浏览器 默认打开的页面路径中的 index.html 文件
// --open 自动打开浏览器
// --port 8080 端口号
// --hot 热更新, 只加载修改的文件(按需加载修改的内容), 而非全部加载
"scripts": {
 "dev": "webpack-dev-server --open --contentBase ./ --port 8080 --hot"
}

```

### 二、 webpack.config.js 配置方式(推荐)

```

var path = require('path')
module.exports = {
 // 入口文件
 entry: path.join(__dirname, 'src/js/index.js'),

```

```
// 输出文件
output: {
 path: path.join(__dirname, 'dist'), // 输出文件的路径
 filename: 'bundle.js' // 输出文件的名称
}

}

const webpack = require('webpack')

devServer: {
 // 服务器的根目录 Tell the server where to serve content from
 // https://webpack.js.org/configuration/dev-server/#devserver-contentbase
 contentBase: path.join(__dirname, './'),
 // 自动打开浏览器
 open: true,
 // 端口号
 port: 8888,

 // ----- 1 热更新 -----
 hot: true
},

plugins: [
 // ----- 2 启用热更新插件 -----
 new webpack.HotModuleReplacementPlugin()
]
```

- [html-webpack-plugin 插件](#)

- 安装: `npm i -D html-webpack-plugin`
- 作用: 根据模板, 自动生成html页面
- 优势: 页面存储在内存中, 自动引入 `bundle.js`、`css` 等文件

```
/* webpack.config.js */
const htmlWebpackPlugin = require('html-webpack-plugin')
plugins: [
 new htmlWebpackPlugin({
 // 模板页面路径
 template: path.join(__dirname, './index.html'),
 // 在内存中生成页面路径, 默认值为: index.html
 filename: 'index.html'
 })
]
```

## Loaders (加载器)

- [webpack - Loaders](#)
- [webpack - 管理资源示例](#)

webpack enables use of loaders to preprocess files. This allows you to bundle any static resource way beyond JavaScript.

- webpack只能处理JavaScript资源
- webpack通过loaders处理非JavaScript静态资源

## 1、CSS打包

- 安装: `npm i -D style-loader css-loader`
- 注意: use中模块的顺序不能颠倒, 加载顺序: 从右向左加载

```
/* 在index.js 导入 css 文件*/
import './css/app.css'

/* webpack.config.js 配置各种资源文件的loader加载器*/
module: {
 // 配置匹配规则
 rules: [
 // test 用来配置匹配文件规则 (正则)
 // use 是一个数组, 按照从后往前的顺序执行加载
 {test: /\.css$/, use: ['style-loader', 'css-loader']},
]
}
```

## 2、使用webpack打包sass文件

- 安装: `npm i -D sass-loader node-sass`
- 注意: `sass-loader` 依赖于 `node-sass` 模块

```
/* webpack.config.js */
// 参考: https://webpack.js.org/loaders/sass-loader/#examples
// "style-loader" : creates style nodes from JS strings 创建style标签
// "css-loader" : translates CSS into CommonJS 将css转化为CommonJS代码
// "sass-loader" : compiles Sass to CSS 将Sass编译为css
module:{
 rules:[
 {test: /\.scss|sass$/}, use: ['style-loader', 'css-loader', 'sass-loader']},
]
}
```

## 3、图片和字体打包

- 安装: `npm i -D url-loader file-loader`
- `file-loader` : 加载并重命名文件 (图片、字体 等)
- `url-loader` : 将图片或字体转化为base64编码格式的字符串, 嵌入到样式文件中

```
/* webpack.config.js */
module: {
```

```

rules:[
 // 打包 图片文件
 { test: /\.(jpg|png|gif|jpeg)$/, use: 'url-loader' },

 // 打包 字体文件
 { test: /\.(woff|woff2|eot|ttf|otf)$/, use: 'file-loader' }
]
}

```

## 图片打包细节

- **limit** 参数的作用：（单位为：字节(byte)）
  - 当图片文件大小（字节）**小于** 指定的limit时，图片被转化为base64编码格式
  - 当图片文件大小（字节）**大于等于** 指定的limit时，图片被重命名以url路径形式加载（此时，需要 **file-loader** 来加载图片）
- 图片文件重命名，保证相同文件不会被加载多次。例如：一张图片（a.jpg）拷贝一个副本（b.jpg），同时引入这两张图片，重命名后只会加载一次，因为这两张图片就是同一张
- 文件重命名以后，会通过MD5加密的方式，来计算这个文件的名称

```

/* webpack.config.js */

module: {
 rules: [
 // {test: /\.(jpg|png|gif|jpeg)$/, use: 'url-loader?limit=100'},
 {
 test: /\.(jpg|png|gif|jpeg)$/,
 use: [
 {
 loader: 'url-loader',
 options: {
 limit: 8192
 }
 }
]
 }
]
}

```

## 字体文件打包说明

- 处理方式与图片相同，可以使用：**file-loader** 或 **url-loader**

## babel

- [babel](#)
- [es2015-loose](#)
- [babel全家桶](#)

- 安装: `npm i -D babel-core babel-loader`
- 安装: `npm i -D babel-preset-env`

## 基本使用 (两步)

- 第一步:

```
/* webpack.config.js */

module: {
 rules: [
 // exclude 排除，不需要编译的目录，提高编译速度
 {test: /\.js$/, use: 'babel-loader', exclude: /node_modules/}
]
}
```

- 第二步: 在项目根目录中新建 `.babelrc` 配置文件

```
/* 创建 .babelrc 文件*/
// 将来babel-loader运行的时候，会检查这个配置文件，并读取相关的语法和插件配置
{
 "presets": ["env"]
}
```

## babel的说明

- babel的作用:
  - 1 语法转换: 将新的ES语法转化为浏览器能识别的语法 (`babel-preset-*`)
  - 2 polyfill浏览器兼容: 让低版本浏览器兼容最新版ES的API

### **babel-preset-\***

Babel通过语法转换器，能够支持最新版本的JavaScript语法  
`babel-preset-*` 用来指定我们书写的是什么版本的JS代码

- 作用: 将新的ES语法转化为浏览器能识别的ES5代码
- [ES6语法提案的批准流程](#)
  - ES2015 也就是 ES6, 下一个版本是ES7, 从 ES6 到 ES7之间经历了 5 个阶段
  - `babel-preset-es2015` 转换es6的语法
  - `babel-preset-stage-0` 转换比es6更新的语法

Stage 0 - Strawman (展示阶段)

Stage 1 - Proposal (征求意见阶段)

Stage 2 - Draft (草案阶段)

Stage 3 - Candidate (候选人阶段)

Stage 4 - Finished (定案阶段)

```
Stage 0 is "i've got a crazy idea",
stage 1 is "this idea might not be stupid",
stage 2 is "let's use polyfills and transpilers to play with it",
stage 3 is "let's let browsers implement it and see how it goes",
stage 4 is "now it's javascript".
```

## babel-polyfill 和 transform-runtime

- 作用：实现浏览器对不支持API的兼容（兼容旧环境、填补）
  - 在低版本浏览器中使用高级的ES6或ES7的方法或函数，比如：`'abc'.padStart(10)`
- 方式一 [polyfill](#)
- 方式二 [transform-runtime](#)
- 方式一：`npm i -S babel-polyfill`
- 方式二：`npm i -D babel-plugin-transform-runtime` 和 `npm i -S babel-runtime`
  - 注意：babel-runtime包中的代码会被打包到你的代码中（-S）

区别：

`polyfill` 所有兼容性问题，都可以通过`polyfill`解决（包括：实例方法）、污染全局环境

`runtime` 除了实例方法以外，其他兼容新问题都能解决、不污染全局环境

`polyfill`：如果想要支持全局对象（比如：``Promise``）、静态方法（比如：``Object.assign``）或者\*\*实例方法\*\*（比如：``String.prototype``）

`babel-runtime`：提供了兼容旧环境的函数，使用的时候，需要我们自己手动引入

比如：`const Promise = require('babel-runtime/core-js/promise')`

存在的问题：

1 手动引入太繁琐

2 多个文件引入同一个`helper`（定义），造成代码重复，增加代码体积

`babel-plugin-transform-runtime`：

1 自动引入`helper`（比如，上面引入的 `Promise`）

2 `babel-runtime`提供`helper`定义，引入这个`helper`即可使用，避免重复

3 依赖于 `babel-runtime` 插件

`transform-runtime`插件的使用：

直接在 `.bablerc` 文件中，添加一个 `plugins` 的配置项即可！！！

```
"plugins": [
 "transform-runtime"
]
```

```
/*
 babel-polyfill 的使用步骤：
 1 main.js
*/
```

```
// 第一行引入
require("babel-polyfill")

var s = 'abc'.padStart(4)
console.log(s)

// 2 webpack.config.js 配置
module.exports = {
 entry: ['babel-polyfill', './js/main.js']
}
```

## 总结

`babel-core` babel核心包

`babel-loader` 用来解析js文件

`babel-preset-*` 新ES语法的解析和转换

`transform-runtime / babel-polyfill` 兼容旧浏览器，到达支持新API目的

```
// 判断浏览器是否兼容 padStart 这个 API
if (!String.prototype.padStart) {
 // 如果不兼容，就自己模拟 padStart的功能实现一份
 String.prototype.padStart = function padStart(targetLength,padString) {
 }
}
```

## vue单文件组件

- [vue-loader](#)
- single-file components(单文件组件)
- 后缀名: `.vue`，该文件需要被预编译后才能在浏览器中使用
- 注意: 单文件组件依赖于两个包 **vue-loader / vue-template-compiler**
- 安装: `npm i -D vue-loader vue-template-compiler`

```
<!-- App.vue 示例代码: -->
<template>
 <div>
 <h1>VUE 单文件组件示例 -- App.vue</h1>
 <p>这是 模板内容</p>
 </div>
</template>

<script>
 // 组件中的逻辑代码
 export default {}
</script>
```

```

<style>
/* 组件样式 */
h1 {
 color: red;
}
</style>

// webpack.config.js 配置:
module: {
 rules: [
 {
 test: /\.vue$/,
 loader: 'vue-loader'
 }
]
}

```

## 使用单文件组件

```

/* main.js */

import Vue from 'vue'
// 导入 App 组件
import App from './App.vue'

const vm = new Vue({
 el: '#app',
 // 通过 render 方法，渲染App组件
 render: c => c(App)
})

```

## 单文件组件使用步骤

- 1 安装: `npm i -D vue-loader vue-template-compiler`
- 2 在 `webpack.config.js` 中配置 `.vue` 文件的loader
  - `{ test: /\.vue$/, use: 'vue-loader' }`
- 3 创建 `App.vue` 单文件组件，注意：App可以是任意名称
- 4 在 `main.js` 入口文件中，导入 `vue` 和 `App.vue` 组件，通过 `render` 将组件与实例挂到一起

## 单文件组件+路由

- [vue - Vue.use](#)
- [Vue.use 和 路由](#)

```

import Vue from 'vue'
import App from './App.vue'

// ----- vue路由配置 开始 -----
import Home from './components/home/Home.vue'
import Login from './components/login/Login.vue'

// 1 导入 路由模块
import VueRouter from 'vue-router'
// 2 ** 调用use方法使用插件 **
Vue.use(VueRouter)
// 3 创建路由对象
const router = new VueRouter({
 routes: [
 { path: '/home', component: Home },
 { path: '/login', component: Login }
]
})
// ----- vue路由配置 结束 -----

const vm = new Vue({
 el: '#app',
 render: c => c(App),
 // 4 挂载到 vue 实例中
 router
})

```

## Mint-UI

---

- 基于 Vue.js 的移动端组件库
- [Mint-UI](#)

## 快速开始

- 安装: `npm i -S mint-ui`

```

// 1 导入 mint-ui模块
import MintUI from 'mint-ui'
// 2 导入 样式
import 'mint-ui/lib/style.css'
// 3 注册插件
Vue.use(MintUI)

```

## MUI

---

- [MUI](#)
- MUI 也是移动端的UI库

- 使用：从github下载包，找到dist文件夹，只需要导入样式即可

```
// 只需要导入 MUI的样式 即可，根据MUI的例子，直接使用HTML结果即可
// 导入样式
import './lib/mui/css/mui.min.css'
```

## ElementUI

- 这是PC端的UI组件库
- 安装：`npm i -S element-ui`
- [饿了吗 - ElementUI](#)

```
{
 "presets": [
 ["es2015", { "modules": false }], "stage-0"
],
 "plugins": [
 ["component", [
 {
 "libraryName": "mint-ui",
 "style": true
 },
 {
 "libraryName": "element-ui",
 "styleLibraryName": "theme-default"
 }
]]
]
}
```

## Webpack 发布项目

- [webpack 打包的各种坑](#)
- `webpack` 命令能够生成dist目录到磁盘中，最终，把打包后的代码，部署服务器中去
- `webpack-dev-server` 仅是在内存中生成的文件，并没有写到磁盘中，所以，只能在开发期间使用

## 创建项目发布配置文件

- 开发期间配置文件：`webpack.config.js`
- 项目发布配置文件：`webpack.prod.js`（文件名称非固定 production 生产环境）
- 命令：`webpack --config webpack.prod.js` 指定配置文件名称运行webpack
- 参数：`--display-error-details` 用于显示webpack打包的错误信息

```
/* package.json */
```

```
"scripts": {
```

```

 "build": "webpack --config webpack.prod.js"
}

```

- 1 在项目根目录中创建 `webpack.prod.js` 文件
- 2 在 `package.json` 中，配置一个 `scripts`
- 3 在 终端中 通过 `npm run build` 对项目进行打包

## 打包处理过程

- 1 删除掉 `devServer` 相关的配置项
- 2 将图片和字体文件输出到指定的文件夹中
- 3 自动删除 `dist` 目录
- 4 分离第三方包（将使用的 `vue` 等第三方包抽离到 `vender.js` 中）
- 5 压缩混淆 JS 以及 指定生成环境
- 6 抽取和压缩 CSS 文件
- 7 压缩 HTML 页面
- 8 配合 `vue` 的异步组件，实现按需加载功能

## 处理图片路径

- 注意：如果 `limit` 小于比图片大，那么图片将被转化为 `base64` 编码格式
- [name参数介绍](#)

```

/* webpack.prod.js */
// 处理URL路径的loader

{
 test: /\.(jpg|png|gif|bmp|jpeg)$/,
 use: {
 loader: 'url-loader',
 options: {
 limit: 8192,
 name: 'images/[hash:7].[ext]' // 作用：将图片输出到images文件夹中，文件名采用7位的哈希值（MD5），并且保持原来的后缀名
 // name: 指定文件输出路径和输出文件命令规则
 // [hash:7]: 表示使用7位哈希值代表文件名称
 // [ext]: 表示保持文件原有后缀名
 // name: 'imgs/img-[hash:7].[ext]'
 }
 }
},

```

## 自动删除dist目录

- 安装：`npm i -D clean-webpack-plugin`
- 作用：每次打包之前，删除上一次打包的 `dist` 目录

```
/* webpack.prod.js */
const cleanWebpackPlugin = require('clean-webpack-plugin')

plugins: [
 // 创建一个删除文件夹的插件, 删除dist目录
 new cleanWebpackPlugin(['./dist'])
]
```

## 分离第三方包

- 目的: 将公共的第三方包, 抽离为一个单独的包文件, 这样防止重复打包!
  - 例如: main.js、router、vuex中都引入了vue, 不分离的话, vue会被打包3次
  - 抽离后, vue文件只会被打包一次, 用到的地方仅仅是引用

```
/* webpack.prod.js */

// 1 入口 -- 打包文件的入口
entry: {
 // 项目代码入口
 app: path.join(__dirname, './src/js/main.js'),
 // 第三方包入口
 vendor: ['vue', 'vue-router', 'axios']
},
output: {
 // 2 修改输出文件路径和命名规则
 filename: 'js/[name].[chunkhash].js',
},
plugins: [
 // 3 抽离第三方包
 new webpack.optimize.CommonsChunkPlugin({
 // 将 entry 中指定的 ['vue', 'vue-router', 'axios'] 打包到名为 vendor 的js文件中
 // 第三方包入口名称, 对应 entry 中的 vendor 属性
 name: 'vendor',
 }),
]
```

## 压缩混淆JS

- 注意: uglifyjs 无法压缩ES6的代码

```
plugins: [
 // 优化代码
 // https://github.com/webpack-contrib/uglifyjs-webpack-plugin/tree/v0.4.6
 new webpack.optimize.UglifyJsPlugin({
 // 压缩
 compress: {
```

```

 // 移除警告
 warnings: false
 }
),

// 指定环境为生产环境: vue会根据这一项启用压缩后的vue文件
new webpack.DefinePlugin({
 'process.env': {
 'NODE_ENV': JSON.stringify('production')
 }
})
]

```

## 抽取和压缩CSS文件

- 安装: 抽离 `npm i -D extract-text-webpack-plugin`
- 安装: 压缩 `npm i -D optimize-css-assets-webpack-plugin`
- [webpack 抽离CSS文档](#)
- [压缩抽离后的CSS](#)

压缩和抽离CSS报错的说明:

```
Error processing file: css/style.css
postcss-svgo: Error in parsing SVG: Unquoted attribute value
```

原因: 压缩和抽离CSS的插件中只允许 SVG 使用双引号

```

/* webpack.prod.js */

// 分离 css 到独立的文件中
const ExtractTextPlugin = require("extract-text-webpack-plugin");
// 压缩 css 资源文件
const OptimizeCssAssetsPlugin = require('optimize-css-assets-webpack-plugin')

// bug描述: 生成后面的css文件中图片路径错误, 打开页面找不到图片
// 解决: google搜索 webpack css loader 样式图片路径
output: {
 // ...

 // https://doc.webpack-china.org/configuration/output/#output-publicpath
 // 设置公共路径
 publicPath: '/',
}

module: {
 rules: [
 {
 test: /\.css$/,
 use: ExtractTextPlugin.extract({
 fallback: "style-loader",
 use: "css-loader"
 })
 }
]
}

```

```

},
{
 test: /\.scss$/,
 use: ExtractTextPlugin.extract({
 fallback: "style-loader",
 use: ['css-loader', 'sass-loader']
 })
},
],
},
plugins: [
 // 通过插件抽离 css (参数)
 new ExtractTextPlugin("css/style.css"),
 // 抽离css 的辅助压缩插件
 new OptimizeCssAssetsPlugin()
]

```

## 压缩HTML页面

- 详细的配置可以参考[html-minifier](#)

```

new htmlWebpackPlugin({
 // 模板页面
 template: path.join(__dirname, './index.html'),
 // 压缩HTML
 minify: {
 // 移除空白
 collapseWhitespace: true,
 // 移除注释
 removeComments: true,
 // 移除属性中的双引号
 removeAttributeQuotes: true
 }
}),

```

## vue配合webpack实现路由按需加载

- [Vue 路由懒加载](#)
- [Vue 异步组件](#)
- [Vue 组件懒加载浅析](#)
- [Vue.js路由懒加载[译]]([www.jianshu.com/p/abb0...](http://www.jianshu.com/p/abb0...))

## 步骤

- 1 修改组件的引用方式

```

// 方式一: require.ensure()
const NewsList = r => require.ensure([], () => r(require('../components/news/newslist.vue')), 'news')

```

```
// 方式二: import() -- 推荐
// 注意: /* webpackChunkName: "newsinfo" */ 是一个特殊的语法, 表示生成js文件的名称
const NewsInfo = () => import(/* webpackChunkName: "newsinfo" */ '../components/news/newsinfo.vue')
```

- 2 修改 webpack 配置文件的output

```
output: {
 // -----添加 chunkFilename, 指定输出js文件的名称-----
 chunkFilename: 'js/[name].[chunkhash].js',
},
```

Vue.js    Webpack    JavaScript    Babel

---