# Vue.js - Day1

## 课程介绍

前5天： 都在学习Vue基本的语法和概念；打包工具 Webpack , Gulp 后5天： 以项目驱动教学；

### 什么是Vue.js

- Vue.js 是目前最火的一个前端框架，React是最流行的一个前端框架（React除了开发网站，还可以开发手机App， Vue语法也是可以用于进行手机App开发的，需要借助于Weex）
- Vue.js 是前端的**主流框架之一**，和Angular.js、React.js 一起，并成为前端三大主流框架！
- Vue.js 是一套构建用户界面的框架，**只关注视图层**，它不仅易于上手，还便于与第三方库或既有项目整合。（Vue有配套的第三方类库，可以整合起来做大型项目的开发）
- 前端的主要工作？主要负责MVC中的V这一层；主要工作就是和界面打交道，来制作前端页面效果；

## 为什么要学习流行框架

- 企业为了提高开发效率：在企业中，时间就是效率，效率就是金钱；
- 企业中，使用框架，能够提高开发的效率；

- 提高开发效率的发展历程：原生JS -> Jquery之类的类库 -> 前端模板引擎 -> Angular.js / Vue.js（能够帮助我们减少不必要的DOM操作；提高渲染效率；双向数据绑定的概念【通过框架提供的指令，我们前端程序员只需要关心数据的业务逻辑，不再关心DOM是如何渲染的了】）
- 在Vue中，一个核心的概念，就是让用户不再操作DOM元素，解放了用户的双手，让程序员可以更多的时间去关注业务逻辑；

- 增强自己就业时候的竞争力
- 人无我有，人有我优
- 你平时不忙的时候，都在干嘛？

## 框架和库的区别

- 框架：是一套完整的解决方案；对项目的侵入性较大，项目如果需要更换框架，则需要重新架构整个项目。
- node 中的 express；

- 库（插件）：提供某一个小功能，对项目的侵入性较小，如果某个库无法完成某些需求，可以很容易切换到其它库实现需求。
- 1. 从Jquery 切换到 Zepto
- 2. 从 EJS 切换到 art-template

# Node（后端）中的 MVC 与 前端中的 MVVM 之间的区别

- MVC 是后端的分层开发概念；
- MVVM是前端视图层的概念，主要关注于 视图层分离，也就是说：MVVM把前端的视图层，分为了 三部分 Model, View , VM ViewModel
- 为什么有了MVC还要有MVVM

# Vue.js 基本代码 和 MVVM 之间的对应关系

# Vue之 - 基本的代码结构 和 插值表达式 、 v-cloak

# Vue指令之 v-text 和 v-html

# Vue指令之 v-bind 的三种用法

1. 直接使用指令 `v-bind`
2. 使用简化指令 `:`
3. 在绑定的时候，拼接绑定内容： `:title="btnTitle + '，这是追加的内容'"`

# Vue指令之 v-on 和 跑马灯效果

## 跑马灯效果

1. HTML结构：

```
<div id="app">
    <p>{{info}}</p>
    <input type="button" value="开启" v-on:click="go">
    <input type="button" value="停止" v-on:click="stop">
  </div>
```

2. Vue实例：

```
    // 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      data: {
        info: '猥琐发育，别浪~! ',
        intervalId: null
      },
      methods: {
        go() {
          // 如果当前有定时器在运行，则直接return
          if (this.intervalId != null) {
            return;
          }
          // 开始定时器
          this.intervalId = setInterval(() => {
            this.info = this.info.substring(1) + this.info.substring(0, 1);
          }, 500);
        },
        stop() {
```

```
          clearInterval(this.intervalId);
        }
      }
    });
```

## Vue指令之v-on的缩写和事件修饰符

### 事件修饰符:

- .stop 阻止冒泡
- .prevent 阻止默认事件
- .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）触发时触发回调
- .once 事件只触发一次

## Vue指令之v-model和双向数据绑定

## 简易计算器案例

1. HTML 代码结构

```
<div id="app">
  <input type="text" v-model="n1">
  <select v-model="opt">
```

```
      <option value="0">+</option>
      <option value="1">-</option>
      <option value="2">*</option>
      <option value="3">÷</option>
    </select>
    <input type="text" v-model="n2">
    <input type="button" value="=" v-on:click="getResult">
    <input type="text" v-model="result">
  </div>
```

2. Vue实例代码：

```
  // 创建 Vue 实例，得到 ViewModel
  var vm = new Vue({
    el: '#app',
    data: {
      n1: 0,
      n2: 0,
      result: 0,
      opt: '0'
    },
    methods: {
      getResult() {
        switch (this.opt) {
          case '0':
            this.result = parseInt(this.n1) + parseInt(this.n2);
            break;
          case '1':
```

```
              this.result = parseInt(this.n1) - parseInt(this.n2);
              break;
            case '2':
              this.result = parseInt(this.n1) * parseInt(this.n2);
              break;
            case '3':
              this.result = parseInt(this.n1) / parseInt(this.n2);
              break;
          }
        }
      }
    });
```

# 在Vue中使用样式

## 使用class样式

1. 数组

```
<h1 :class="['red', 'thin']">这是一个邪恶的H1</h1>
```

2. 数组中使用三元表达式

```
<h1 :class="['red', 'thin', isactive?'active':'']">这是一个邪恶的H1</h1>
```

3. 数组中嵌套对象

```
<h1 :class="['red', 'thin', {'active': isactive}]">这是一个邪恶的H1</h1>
```

4. 直接使用对象

```
<h1 :class="{red:true, italic:true, active:true, thin:true}">这是一个邪恶的H1</h1>
```

## 使用内联样式

1. 直接在元素上通过 `:style` 的形式，书写样式对象

```
<h1 :style="{color: 'red', 'font-size': '40px'}">这是一个善良的H1</h1>
```

2. 将样式对象，定义到 `data` 中，并直接引用到 `:style` 中

- 在data上定义样式：

```
data: {
        h1StyleObj: { color: 'red', 'font-size': '40px', 'font-weight': '200' }
}
```

- 在元素中，通过属性绑定的形式，将样式对象应用到元素中：

```
<h1 :style="h1StyleObj">这是一个善良的H1</h1>
```

3. 在 `:style` 中通过数组，引用多个 `data` 上的样式对象

- 在data上定义样式：

```
data: {
        h1StyleObj: { color: 'red', 'font-size': '40px', 'font-weight': '200' },
        h1StyleObj2: { fontStyle: 'italic' }
}
```

- 在元素中，通过属性绑定的形式，将样式对象应用到元素中：

```
<h1 :style="[h1StyleObj, h1StyleObj2]">这是一个善良的H1</h1>
```

# Vue指令之 v-for 和 key 属性

1. 迭代数组

```
<ul>
  <li v-for="(item, i) in list">索引: {{i}} --- 姓名: {{item.name}} --- 年龄: {{item.age}}
</li>
</ul>
```

2. 迭代对象中的属性

```
<!-- 循环遍历对象身上的属性 -->

<div v-for="(val, key, i) in userInfo">{{val}} --- {{key}} --- {{i}}</div>
```

3. 迭代数字

```
<p v-for="i in 10">这是第 {{i}} 个P标签</p>
```

> 2.2.0+ 的版本里，**当在组件中使用** v-for 时，key 现在是必须的。

当 Vue.js 用 v-for 正在更新已渲染过的元素列表时，它默认用 “**就地复用**” 策略。如果数据项的顺序被改变，Vue将**不是移动 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。

# Vue指令之`v-if`和`v-show`

> 一般来说，v-if 有更高的切换消耗而 v-show 有更高的初始渲染消耗。因此，如果需要频繁切换 v-show 较好，如果在运行时条件不大可能改变 v-if 较好。

# 品牌管理案例

## 添加新品牌

## 删除品牌

## 根据条件筛选品牌

1. 1.x 版本中的filterBy指令，在2.x中已经被废除：

filterBy - 指令

```
<tr v-for="item in list | filterBy searchName in 'name'">

  <td>{{item.id}}</td>

  <td>{{item.name}}</td>

  <td>{{item.ctime}}</td>

  <td>

    <a href="#" @click.prevent="del(item.id)">删除</a>

  </td>

</tr>
```

2. 在2.x版本中手动实现筛选的方式：

- 筛选框绑定到 VM 实例中的 `searchName` 属性：

```
<hr> 输入筛选名称：

<input type="text" v-model="searchName">
```

- 在使用 `v-for` 指令循环每一行数据的时候，不再直接 `item in list`，而是 `in` 一个 过滤的methods 方法，同时，把过滤条件 `searchName` 传递进去：

```
<tbody>

      <tr v-for="item in search(searchName)">

        <td>{{item.id}}</td>
```

```
          <td>{{item.name}}</td>
          <td>{{item.ctime}}</td>
          <td>
            <a href="#" @click.prevent="del(item.id)">删除</a>
          </td>
        </tr>
      </tbody>
```

- `search` 过滤方法中，使用 数组的 `filter` 方法进行过滤：

```
search(name) {
  return this.list.filter(x => {
    return x.name.indexOf(name) != -1;
  });
}
```

# Vue调试工具`vue-devtools`的安装步骤和使用

Vue.js devtools - 翻墙安装方式 - 推荐

# 过滤器

概念：Vue.js 允许你自定义过滤器，**可被用作一些常见的文本格式化。**过滤器可以用在两个地方：**mustache 插值和 v-bind 表达式。**过滤器应该被添加在 JavaScript 表达式的尾部，由“管道”符指示；

## 私有过滤器

1. HTML元素：

```
<td>{{item.ctime | dataFormat('yyyy-mm-dd')}}</td>
```

2. 私有 `filters` 定义方式：

```
filters: { // 私有局部过滤器，只能在 当前 VM 对象所控制的 View 区域进行使用
    dataFormat(input, pattern = "") { // 在参数列表中 通过 pattern="" 来指定形参默认值，防止报错
      var dt = new Date(input);
      // 获取年月日
      var y = dt.getFullYear();
      var m = (dt.getMonth() + 1).toString().padStart(2, '0');
      var d = dt.getDate().toString().padStart(2, '0');

      // 如果 传递进来的字符串类型，转为小写之后，等于 yyyy-mm-dd，那么就返回 年-月-日
      // 否则，就返回  年-月-日 时：分：秒
      if (pattern.toLowerCase() === 'yyyy-mm-dd') {
        return `${y}-${m}-${d}`;
      } else {
        // 获取时分秒
        var hh = dt.getHours().toString().padStart(2, '0');
        var mm = dt.getMinutes().toString().padStart(2, '0');
        var ss = dt.getSeconds().toString().padStart(2, '0');

        return `${y}-${m}-${d} ${hh}:${mm}:${ss}`;
      }
    }
  }
```

> 使用ES6中的字符串新方法 String.prototype.padStart(maxLength, fillString='') 或 String.prototype.padEnd(maxLength, fillString='')来填充字符串；

## 全局过滤器

```
// 定义一个全局过滤器
Vue.filter('dataFormat', function (input, pattern = '') {
  var dt = new Date(input);
  // 获取年月日
  var y = dt.getFullYear();
  var m = (dt.getMonth() + 1).toString().padStart(2, '0');
  var d = dt.getDate().toString().padStart(2, '0');

  // 如果 传递进来的字符串类型，转为小写之后，等于 yyyy-mm-dd，那么就返回 年-月-日
  // 否则，就返回  年-月-日 时：分：秒
  if (pattern.toLowerCase() === 'yyyy-mm-dd') {
    return `${y}-${m}-${d}`;
  } else {
    // 获取时分秒
    var hh = dt.getHours().toString().padStart(2, '0');
    var mm = dt.getMinutes().toString().padStart(2, '0');
    var ss = dt.getSeconds().toString().padStart(2, '0');

    return `${y}-${m}-${d} ${hh}:${mm}:${ss}`;
  }
});
```

> 注意：当有局部和全局两个名称相同的过滤器时候，会以就近原则进行调用，即：局部过滤器优先于全局过滤器被调用！

# 键盘修饰符以及自定义键盘修饰符

## 1.x中自定义键盘修饰符【了解即可】

```
Vue.directive('on').keyCodes.f2 = 113;
```

## 2.x中自定义键盘修饰符

1. 通过`Vue.config.keyCodes.名称 = 按键值`来自定义案件修饰符的别名：

```
Vue.config.keyCodes.f2 = 113;
```

2. 使用自定义的按键修饰符：

```
<input type="text" v-model="name" @keyup.f2="add">
```

# 自定义指令

1. 自定义全局和局部的 自定义指令：

```
    // 自定义全局指令 v-focus，为绑定的元素自动获取焦点：
    Vue.directive('focus', {
      inserted: function (el) { // inserted 表示被绑定元素插入父节点时调用
        el.focus();
      }
    });
```

```
    // 自定义局部指令 v-color 和 v-font-weight，为绑定的元素设置指定的字体颜色 和 字体粗细：
      directives: {
        color: { // 为元素设置指定的字体颜色
          bind(el, binding) {
            el.style.color = binding.value;
          }
        },
        'font-weight': function (el, binding2) { // 自定义指令的简写形式，等同于定义了 bind 和 update 两个钩子函数
          el.style.fontWeight = binding2.value;
        }
      }
```

2. 自定义指令的使用方式：

```
<input type="text" v-model="searchName" v-focus v-color="'red'" v-font-weight="900">
```

# Vue 1.x 中 自定义元素指令【已废弃,了解即可】

```
Vue.elementDirective('red-color', {
  bind: function () {
    this.el.style.color = 'red';
  }
});
```

使用方式：

```
<red-color>1232</red-color>
```

# 相关文章

1. vue.js 1.x 文档

2. vue.js 2.x 文档
3. String.prototype.padStart(maxLength, fillString)
4. js 里面的键盘事件对应的键码
5. Vue.js双向绑定的实现原理

# Vue.js - Day2

## 品牌管理案例

### 添加新品牌

### 删除品牌

### 根据条件筛选品牌

1. 1.x 版本中的filterBy指令，在2.x中已经被废除：

filterBy - 指令

```
<tr v-for="item in list | filterBy searchName in 'name'">
  <td>{{item.id}}</td>
  <td>{{item.name}}</td>
  <td>{{item.ctime}}</td>
  <td>
    <a href="#" @click.prevent="del(item.id)">删除</a>
  </td>
</tr>
```

2. 在2.x版本中手动实现筛选的方式：

- 筛选框绑定到 VM 实例中的 `searchName` 属性：

```
<hr> 输入筛选名称：
<input type="text" v-model="searchName">
```

- 在使用 `v-for` 指令循环每一行数据的时候，不再直接 `item in list`，而是 `in` 一个 过滤的methods 方法，同时，把过滤条件 `searchName` 传递进去：

```
<tbody>
```

```
      <tr v-for="item in search(searchName)">
        <td>{{item.id}}</td>
        <td>{{item.name}}</td>
        <td>{{item.ctime}}</td>
        <td>
          <a href="#" @click.prevent="del(item.id)">删除</a>
        </td>
      </tr>
    </tbody>
```

- `search` 过滤方法中，使用 数组的 `filter` 方法进行过滤：

```
search(name) {
  return this.list.filter(x => {
    return x.name.indexOf(name) != -1;
  });
}
```

# Vue调试工具`vue-devtools`的安装步骤和使用

Vue.js devtools - 翻墙安装方式 - 推荐

# 过滤器

概念：Vue.js 允许你自定义过滤器，**可被用作一些常见的文本格式化。**过滤器可以用在两个地方：**mustache 插值和 v-bind 表达式。**过滤器应该被添加在 JavaScript 表达式的尾部，由“管道”符指示；

## 私有过滤器

1. HTML元素：

```
<td>{{item.ctime | dataFormat('yyyy-mm-dd')}}</td>
```

2. 私有 `filters` 定义方式：

```
filters: { // 私有局部过滤器，只能在 当前 VM 对象所控制的 View 区域进行使用

    dataFormat(input, pattern = "") { // 在参数列表中 通过 pattern="" 来指定形参默认值，防止报
错

      var dt = new Date(input);

      // 获取年月日

      var y = dt.getFullYear();

      var m = (dt.getMonth() + 1).toString().padStart(2, '0');

      var d = dt.getDate().toString().padStart(2, '0');


      // 如果 传递进来的字符串类型，转为小写之后，等于 yyyy-mm-dd，那么就返回 年-月-日

      // 否则，就返回  年-月-日 时：分：秒

      if (pattern.toLowerCase() === 'yyyy-mm-dd') {

        return `${y}-${m}-${d}`;

      } else {

        // 获取时分秒

        var hh = dt.getHours().toString().padStart(2, '0');

        var mm = dt.getMinutes().toString().padStart(2, '0');

        var ss = dt.getSeconds().toString().padStart(2, '0');


        return `${y}-${m}-${d} ${hh}:${mm}:${ss}`;

      }

    }

  }
```

> 使用ES6中的字符串新方法 String.prototype.padStart(maxLength, fillString='') 或 String.prototype.padEnd(maxLength, fillString='')来填充字符串；

## 全局过滤器

```
// 定义一个全局过滤器
Vue.filter('dataFormat', function (input, pattern = '') {
  var dt = new Date(input);
  // 获取年月日
  var y = dt.getFullYear();
  var m = (dt.getMonth() + 1).toString().padStart(2, '0');
  var d = dt.getDate().toString().padStart(2, '0');

  // 如果 传递进来的字符串类型，转为小写之后，等于 yyyy-mm-dd，那么就返回 年-月-日
  // 否则，就返回  年-月-日 时：分：秒
  if (pattern.toLowerCase() === 'yyyy-mm-dd') {
    return `${y}-${m}-${d}`;
  } else {
    // 获取时分秒
    var hh = dt.getHours().toString().padStart(2, '0');
    var mm = dt.getMinutes().toString().padStart(2, '0');
    var ss = dt.getSeconds().toString().padStart(2, '0');

    return `${y}-${m}-${d} ${hh}:${mm}:${ss}`;
  }
});
```

> 注意：当有局部和全局两个名称相同的过滤器时候，会以就近原则进行调用，即：局部过滤器优先于全局过滤器被调用！

# 键盘修饰符以及自定义键盘修饰符

## 1.x中自定义键盘修饰符【了解即可】

```
Vue.directive('on').keyCodes.f2 = 113;
```

## 2.x中自定义键盘修饰符

1. 通过`Vue.config.keyCodes.名称 = 按键值`来自定义案件修饰符的别名：

```
Vue.config.keyCodes.f2 = 113;
```

2. 使用自定义的按键修饰符：

```
<input type="text" v-model="name" @keyup.f2="add">
```

# 自定义指令

1. 自定义全局和局部的 自定义指令：

```
    // 自定义全局指令 v-focus，为绑定的元素自动获取焦点：
    Vue.directive('focus', {
      inserted: function (el) { // inserted 表示被绑定元素插入父节点时调用
        el.focus();
      }
    });
```

```
    // 自定义局部指令 v-color 和 v-font-weight，为绑定的元素设置指定的字体颜色 和 字体粗细：
      directives: {
        color: { // 为元素设置指定的字体颜色
          bind(el, binding) {
            el.style.color = binding.value;
          }
        },
        'font-weight': function (el, binding2) { // 自定义指令的简写形式，等同于定义了 bind 和 update 两个钩子函数
          el.style.fontWeight = binding2.value;
        }
      }
```

2. 自定义指令的使用方式：

```
<input type="text" v-model="searchName" v-focus v-color="'red'" v-font-weight="900">
```

# Vue 1.x 中 自定义元素指令【已废弃,了解即可】

```
Vue.elementDirective('red-color', {
  bind: function () {
    this.el.style.color = 'red';
  }
});
```

使用方式：

```
<red-color>1232</red-color>
```

# vue实例的生命周期

- 什么是生命周期：从Vue实例创建、运行、到销毁期间，总是伴随着各种各样的事件，这些事件，统称为生命周期！
- 生命周期钩子：就是生命周期事件的别名而已；

- 生命周期钩子 = 生命周期函数 = 生命周期事件
- 主要的生命周期函数分类：

- 创建期间的生命周期函数：

- beforeCreate：实例刚在内存中被创建出来，此时，还没有初始化好 data 和 methods 属性
- created：实例已经在内存中创建OK，此时 data 和 methods 已经创建OK，此时还没有开始 编译模板
- beforeMount：此时已经完成了模板的编译，但是还没有挂载到页面中
- mounted：此时，已经将编译好的模板，挂载到了页面指定的容器中显示

- 运行期间的生命周期函数：

- beforeUpdate：状态更新之前执行此函数， 此时 data 中的状态值是最新的，但是界面上显示的 数据还是旧的，因为此时还没有开始重新渲染DOM节点
- updated：实例更新完毕之后调用此函数，此时 data 中的状态值 和 界面上显示的数据，都已经完成了更新，界面已经被重新渲染好了！

- 销毁期间的生命周期函数：

- beforeDestroy：实例销毁之前调用。在这一步，实例仍然完全可用。
- destroyed：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。

# vue-resource 实现 get, post, jsonp请求

除了 vue-resource 之外，还可以使用 `axios` 的第三方包实现实现数据的请求

1. 之前的学习中，如何发起数据请求?
2. 常见的数据请求类型？ get post jsonp
3. 测试的URL请求资源地址：

- get请求地址： http://vue.studyit.io/api/getlunbo
- post请求地址：http://vue.studyit.io/api/post
- jsonp请求地址：http://vue.studyit.io/api/jsonp

4. JSONP的实现原理

- 由于浏览器的安全性限制，不允许AJAX访问 协议不同、域名不同、端口号不同的 数据接口，浏览器认为这种访问不安全；
- 可以通过动态创建script标签的形式，把script标签的src属性，指向数据接口的地址，因为script标签不存在跨域限制，这种数据获取方式，称作JSONP（注意：根据JSONP的实现原理，知晓，JSONP只支持Get请求）；
- 具体实现过程：

- 先在客户端定义一个回调方法，预定义对数据的操作；
- 再把这个回调方法的名称，通过URL传参的形式，提交到服务器的数据接口；
- 服务器数据接口组织好要发送给客户端的数据，再拿着客户端传递过来的回调方法名称，拼接出一个调用这个方法的字符串，发送给客户端去解析执行；
- 客户端拿到服务器返回的字符串之后，当作Script脚本去解析执行，这样就能够拿到JSONP的数据了；

- 带大家通过 Node.js ，来手动实现一个JSONP的请求例子；

```
    const http = require('http');
    // 导入解析 URL 地址的核心模块
```

```
    const urlModule = require('url');

    const server = http.createServer();
    // 监听 服务器的 request 请求事件，处理每个请求
    server.on('request', (req, res) => {
      const url = req.url;

      // 解析客户端请求的URL地址
      var info = urlModule.parse(url, true);

      // 如果请求的 URL 地址是 /getjsonp ，则表示要获取JSONP类型的数据
      if (info.pathname === '/getjsonp') {
        // 获取客户端指定的回调函数的名称
        var cbName = info.query.callback;
        // 手动拼接要返回给客户端的数据对象
        var data = {
          name: 'zs',
          age: 22,
          gender: '男',
          hobby: ['吃饭', '睡觉', '运动']
        }
        // 拼接出一个方法的调用，在调用这个方法的时候，把要发送给客户端的数据，序列化为字符串，作为参数传递给这个调用的方法：
        var result = `${cbName}(${JSON.stringify(data)})`;
        // 将拼接好的方法的调用，返回给客户端去解析执行
        res.end(result);
      } else {
        res.end('404');
      }
    });

    server.listen(3000, () => {
      console.log('server running at http://127.0.0.1:3000');
    });
```

5. vue-resource 的配置步骤：

- 直接在页面中，通过 `script` 标签，引入 `vue-resource` 的脚本文件；
- 注意：引用的先后顺序是：先引用 `Vue` 的脚本文件，再引用 `vue-resource` 的脚本文件；

6. 发送get请求：

```
getInfo() { // get 方式获取数据
  this.$http.get('http://127.0.0.1:8899/api/getlunbo').then(res => {
    console.log(res.body);
  })
}
```

7. 发送post请求：

```
postInfo() {
  var url = 'http://127.0.0.1:8899/api/post';
  // post 方法接收三个参数:
  // 参数1:  要请求的URL地址
  // 参数2:  要发送的数据对象
  // 参数3:  指定post提交的编码类型为 application/x-www-form-urlencoded
  this.$http.post(url, { name: 'zs' }, { emulateJSON: true }).then(res => {
    console.log(res.body);
  });
}
```

8. 发送JSONP请求获取数据:

```
jsonpInfo() { // JSONP形式从服务器获取数据
  var url = 'http://127.0.0.1:8899/api/jsonp';
  this.$http.jsonp(url).then(res => {
    console.log(res.body);
  });
}
```

# 配置本地数据库和数据接口API

1. 先解压安装 `PHPStudy`;
2. 解压安装 `Navicat` 这个数据库可视化工具，并激活;
3. 打开 `Navicat` 工具，新建空白数据库，名为 `dtcmsdb4`;
4. 双击新建的数据库，连接上这个空白数据库，在新建的数据库上`右键` -> `运行SQL文件`，选择并执行 `dtcmsdb4.sql` 这个数据库脚本文件；如果执行不报错，则数据库导入完成;
5. 进入文件夹 `vuecms3_nodejsapi` 内部，执行 `npm i` 安装所有的依赖项;
6. 先确保本机安装了 `nodemon`, 没有安装，则运行 `npm i nodemon -g` 进行全局安装，安装完毕后，进入到 `vuecms3_nodejsapi` 目录 -> `src` 目录 -> 双击运行 `start.bat`
7. 如果API启动失败，请检查 PHPStudy 是否正常开启，同时，检查 `app.js` 中第 `14行` 中数据库连接配置字符串是否正确；PHPStudy 中默认的 用户名是root，默认的密码也是root

# 品牌管理改造

## 展示品牌列表

## 添加品牌数据

## 删除品牌数据

# Vue中的动画

为什么要有动画：动画能够提高用户的体验，帮助用户更好的理解页面中的功能;

## 使用过渡类名

1. HTML结构:

```
<div id="app">
    <input type="button" value="动起来" @click="myAnimate">
    <!-- 使用 transition 将需要过渡的元素包裹起来 -->
    <transition name="fade">
      <div v-show="isshow">动画哦</div>
    </transition>
  </div>
```

2. VM 实例：

```
// 创建 Vue 实例，得到 ViewModel
var vm = new Vue({
  el: '#app',
  data: {
    isshow: false
  },
  methods: {
    myAnimate() {
      this.isshow = !this.isshow;
    }
  }
});
```

3. 定义两组类样式：

```
/* 定义进入和离开时候的过渡状态 */
    .fade-enter-active,
    .fade-leave-active {
      transition: all 0.2s ease;
      position: absolute;
    }

    /* 定义进入过渡的开始状态 和 离开过渡的结束状态 */
    .fade-enter,
    .fade-leave-to {
      opacity: 0;
      transform: translateX(100px);
    }
```

## 使用第三方 CSS 动画库

1. 导入动画类库：

```
<link rel="stylesheet" type="text/css" href="./lib/animate.css">
```

2. 定义 transition 及属性：

```
<transition
    enter-active-class="fadeInRight"
    leave-active-class="fadeOutRight"
    :duration="{ enter: 500, leave: 800 }">
    <div class="animated" v-show="isshow">动画哦</div>
</transition>
```

## 使用动画钩子函数

1. 定义 transition 组件以及三个钩子函数：

```
<div id="app">
    <input type="button" value="切换动画" @click="isshow = !isshow">
    <transition
    @before-enter="beforeEnter"
    @enter="enter"
    @after-enter="afterEnter">
      <div v-if="isshow" class="show">OK</div>
    </transition>
  </div>
```

2. 定义三个 methods 钩子方法：

```
methods: {
        beforeEnter(el) { // 动画进入之前的回调
          el.style.transform = 'translateX(500px)';
        },
        enter(el, done) { // 动画进入完成时候的回调
          el.offsetWidth;
          el.style.transform = 'translateX(0px)';
          done();
        },
        afterEnter(el) { // 动画进入完成之后的回调
          this.isshow = !this.isshow;
        }
      }
```

3. 定义动画过渡时长和样式：

```
.show{
      transition: all 0.4s ease;
    }
```

## v-for 的列表过渡

1. 定义过渡样式：

```
<style>
    .list-enter,
    .list-leave-to {
      opacity: 0;
      transform: translateY(10px);
    }

    .list-enter-active,
    .list-leave-active {
      transition: all 0.3s ease;
    }
</style>
```

2. 定义DOM结构，其中，需要使用 transition-group 组件把v-for循环的列表包裹起来：

```
  <div id="app">
    <input type="text" v-model="txt" @keyup.enter="add">

    <transition-group tag="ul" name="list">
      <li v-for="(item, i) in list" :key="i">{{item}}</li>
    </transition-group>
  </div>
```

3. 定义 VM中的结构：

```
    // 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      data: {
        txt: '',
        list: [1, 2, 3, 4]
      },
      methods: {
        add() {
          this.list.push(this.txt);
          this.txt = '';
        }
      }
    });
```

## 列表的排序过渡

`<transition-group>` 组件还有一个特殊之处。不仅可以进入和离开动画，**还可以改变定位。**要使用这个新功能只需了解新增的 `v-move` 特性，**它会在元素的改变定位的过程中应用。**

- `v-move` 和 `v-leave-active` 结合使用，能够让列表的过渡更加平缓柔和：

```
.v-move{
  transition: all 0.8s ease;
}
.v-leave-active{
  position: absolute;
}
```

## 相关文章

1. vue.js 1.x 文档
2. vue.js 2.x 文档
3. String.prototype.padStart(maxLength, fillString)
4. js 里面的键盘事件对应的键码
5. pagekit/vue-resource
6. navicat如何导入sql文件和导出sql文件
7. 贝塞尔在线生成器

# Vue.js - Day3

## 定义Vue组件

什么是组件： 组件的出现，就是为了拆分Vue实例的代码量的，能够让我们以不同的组件，来划分不同的功能模块，将来我们需要什么样的功能，就可以去调用对应的组件即可； 组件化和模块化的不同：

- 模块化： 是从代码逻辑的角度进行划分的；方便代码分层开发，保证每个功能模块的职能单一；
- 组件化： 是从UI界面的角度进行划分的；前端的组件化，方便UI组件的重用；

### 全局组件定义的三种方式

1. 使用 Vue.extend 配合 Vue.component 方法：

```
var login = Vue.extend({
      template: '<h1>登录</h1>'
    });
    Vue.component('login', login);
```

2. 直接使用 Vue.component 方法：

```
Vue.component('register', {
      template: '<h1>注册</h1>'
    });
```

3. 将模板字符串，定义到script标签种：

```
<script id="tmpl" type="x-template">
      <div><a href="#">登录</a> | <a href="#">注册</a></div>
    </script>
```

同时，需要使用 Vue.component 来定义组件：

```
Vue.component('account', {
      template: '#tmpl'
    });
```

> 注意： 组件中的DOM结构，有且只能有唯一的根元素（Root Element）来进行包裹！

### 组件中展示数据和响应事件

1. 在组件中，`data`需要被定义为一个方法，例如：

```
Vue.component('account', {
      template: '#tmpl',
      data() {
        return {
          msg: '大家好! '
        }
      },
      methods:{
        login(){
          alert('点击了登录按钮');
        }
      }
    });
```

2. 在子组件中，如果将模板字符串，定义到了script标签中，那么，要访问子组件身上的`data`属性中的值，需要使用`this`来访问；

## 【重点】为什么组件中的data属性必须定义为一个方法并返回一个对象

1. 通过计数器案例演示

## 使用`components`属性定义局部子组件

1. 组件实例定义方式：

```
<script>
    // 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      data: {},
      methods: {},
      components: { // 定义子组件
        account: { // account 组件
          template: '<div><h1>这是Account组件{{name}}</h1><login></login></div>', // 在这
里使用定义的子组件
          components: { // 定义子组件的子组件
            login: { // login 组件
              template: "<h3>这是登录组件</h3>"
            }
          }
        }
      }
    });
  </script>
```

2. 引用组件：

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

## 使用 flag 标识符结合 v-if 和 v-else 切换组件

1. 页面结构：

```
<div id="app">
    <input type="button" value="toggle" @click="flag=!flag">
    <my-com1 v-if="flag"></my-com1>
    <my-com2 v-else="flag"></my-com2>
  </div>
```

2. Vue实例定义：

```
<script>
    Vue.component('myCom1', {
      template: '<h3>奔波霸</h3>'
    })

    Vue.component('myCom2', {
      template: '<h3>霸波奔</h3>'
    })

    // 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      data: {
        flag: true
      },
      methods: {}
    });
  </script>
```

## 使用 :is 属性来切换不同的子组件,并添加切换动画

1. 组件实例定义方式：

```
  // 登录组件
    const login = Vue.extend({
      template: `<div>
        <h3>登录组件</h3>
      </div>`
    });
    Vue.component('login', login);

    // 注册组件
    const register = Vue.extend({
      template: `<div>
        <h3>注册组件</h3>
      </div>`
    });
    Vue.component('register', register);
```

```
    // 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      data: { comName: 'login' },
      methods: {}
    });
```

2. 使用 `component` 标签，来引用组件，并通过 `:is` 属性来指定要加载的组件：

```
  <div id="app">
    <a href="#" @click.prevent="comName='login'">登录</a>
    <a href="#" @click.prevent="comName='register'">注册</a>
    <hr>
    <transition mode="out-in">
      <component :is="comName"></component>
    </transition>
  </div>
```

3. 添加切换样式：

```
  <style>
    .v-enter,
    .v-leave-to {
      opacity: 0;
      transform: translateX(30px);
    }

    .v-enter-active,
    .v-leave-active {
      position: absolute;
      transition: all 0.3s ease;
    }

    h3{
      margin: 0;
    }
  </style>
```

# 父组件向子组件传值

1. 组件实例定义方式，注意：一定要使用 `props` 属性来定义父组件传递过来的数据

```
<script>
    // 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      data: {
        msg: '这是父组件中的消息'
      },
      components: {
```

```
        son: {
          template: '<h1>这是子组件 --- {{finfo}}</h1>',
          props: ['finfo']
        }
      }
    });
  </script>
```

2. 使用`v-bind`或简化指令，将数据传递到子组件中：

```
<div id="app">
    <son :finfo="msg"></son>
  </div>
```

# 子组件向父组件传值

1. 原理：父组件将方法的引用，传递到子组件内部，子组件在内部调用父组件传递过来的方法，同时把要发送给父组件的数据，在调用方法的时候当作参数传递进去；
2. 父组件将方法的引用传递给子组件，其中，`getMsg`是父组件中`methods`中定义的方法名称，`func`是子组件调用传递过来方法时候的方法名称

```
<son @func="getMsg"></son>
```

3. 子组件内部通过`this.$emit('方法名', 要传递的数据)`方式，来调用父组件中的方法，同时把数据传递给父组件使用

```
<div id="app">
    <!-- 引用父组件 -->
    <son @func="getMsg"></son>

    <!-- 组件模板定义 -->
    <script type="x-template" id="son">
      <div>
        <input type="button" value="向父组件传值" @click="sendMsg" />
      </div>
    </script>
  </div>

  <script>
    // 子组件的定义方式
    Vue.component('son', {
      template: '#son', // 组件模板Id
      methods: {
        sendMsg() { // 按钮的点击事件
          this.$emit('func', 'OK'); // 调用父组件传递过来的方法，同时把数据传递出去
        }
      }
    });

    // 创建 Vue 实例，得到 ViewModel
```

```
  var vm = new Vue({
    el: '#app',
    data: {},
    methods: {
      getMsg(val){ // 子组件中，通过 this.$emit() 实际调用的方法，在此进行定义
        alert(val);
      }
    }
  });
</script>
```

## 评论列表案例

目标：主要练习父子组件之间传值

## 使用 `this.$refs` 来获取元素和组件

```
<div id="app">
  <div>
    <input type="button" value="获取元素内容" @click="getElement" />
    <!-- 使用 ref 获取元素 -->
    <h1 ref="myh1">这是一个大大的H1</h1>

    <hr>
    <!-- 使用 ref 获取子组件 -->
    <my-com ref="mycom"></my-com>
  </div>
</div>

<script>
  Vue.component('my-com', {
    template: '<h5>这是一个子组件</h5>',
    data() {
      return {
        name: '子组件'
      }
    }
  });

  // 创建 Vue 实例，得到 ViewModel
  var vm = new Vue({
    el: '#app',
    data: {},
    methods: {
      getElement() {
        // 通过 this.$refs 来获取元素
        console.log(this.$refs.myh1.innerText);
        // 通过 this.$refs 来获取组件
        console.log(this.$refs.mycom.name);
      }
    }
```

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

# 什么是路由

1. 对于普通的网站，所有的超链接都是URL地址，所有的URL地址都对应服务器上对应的资源；
2. 对于单页面应用程序来说，主要通过URL中的hash(#号)来实现不同页面之间的切换，同时，hash有一个特点：HTTP请求中不会包含hash相关的内容；所以，单页面程序中的页面跳转主要用hash实现；
3. 在单页面应用程序中，这种通过hash改变来切换页面的方式，称作前端路由（区别于后端路由）；

# 在 vue 中使用 vue-router

1. 导入 vue-router 组件类库：

```
<!-- 1. 导入 vue-router 组件类库 -->
  <script src="./lib/vue-router-2.7.0.js"></script>
```

2. 使用 router-link 组件来导航

```
<!-- 2. 使用 router-link 组件来导航 -->
<router-link to="/login">登录</router-link>
<router-link to="/register">注册</router-link>
```

3. 使用 router-view 组件来显示匹配到的组件

```
<!-- 3. 使用 router-view 组件来显示匹配到的组件 -->
<router-view></router-view>
```

4. 创建使用 `Vue.extend` 创建组件

```
    // 4.1 使用 Vue.extend 来创建登录组件
    var login = Vue.extend({
      template: '<h1>登录组件</h1>'
    });

    // 4.2 使用 Vue.extend 来创建注册组件
    var register = Vue.extend({
      template: '<h1>注册组件</h1>'
    });
```

5. 创建一个路由 router 实例，通过 routers 属性来定义路由匹配规则

```
// 5. 创建一个路由 router 实例，通过 routers 属性来定义路由匹配规则
    var router = new VueRouter({
      routes: [
        { path: '/login', component: login },
        { path: '/register', component: register }
      ]
    });
```

6. 使用 router 属性来使用路由规则

```
// 6. 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      router: router // 使用 router 属性来使用路由规则
    });
```

# 设置路由高亮

# 设置路由切换动效

# 在路由规则中定义参数

1. 在规则中定义参数：

```
{ path: '/register/:id', component: register }
```

2. 通过 `this.$route.params` 来获取路由中的参数：

```
var register = Vue.extend({
      template: '<h1>注册组件 --- {{this.$route.params.id}}</h1>'
    });
```

# 使用 `children` 属性实现路由嵌套

```
  <div id="app">
    <router-link to="/account">Account</router-link>

    <router-view></router-view>
  </div>

  <script>
    // 父路由中的组件
    const account = Vue.extend({
      template: `<div>
        这是account组件
        <router-link to="/account/login">login</router-link> |
        <router-link to="/account/register">register</router-link>
```

```
        <router-view></router-view>
      </div>`
    });

    // 子路由中的 login 组件
    const login = Vue.extend({
      template: '<div>登录组件</div>'
    });

    // 子路由中的 register 组件
    const register = Vue.extend({
      template: '<div>注册组件</div>'
    });

    // 路由实例
    var router = new VueRouter({
      routes: [
        { path: '/', redirect: '/account/login' }, // 使用 redirect 实现路由重定向
        {
          path: '/account',
          component: account,
          children: [ // 通过 children 数组属性，来实现路由的嵌套
            { path: 'login', component: login }, // 注意，子路由的开头位置，不要加 / 路径符
            { path: 'register', component: register }
          ]
        }
      ]
    });

    // 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      data: {},
      methods: {},
      components: {
        account
      },
      router: router
    });
  </script>
```

# 命名视图实现经典布局

1. 标签代码结构：

```
<div id="app">
    <router-view></router-view>
    <div class="content">
      <router-view name="a"></router-view>
      <router-view name="b"></router-view>
    </div>
  </div>
```

2. JS代码：

```
<script>
    var header = Vue.component('header', {
      template: '<div class="header">header</div>'
    });

    var sidebar = Vue.component('sidebar', {
      template: '<div class="sidebar">sidebar</div>'
    });

    var mainbox = Vue.component('mainbox', {
      template: '<div class="mainbox">mainbox</div>'
    });

    // 创建路由对象
    var router = new VueRouter({
      routes: [
        {
          path: '/', components: {
            default: header,
            a: sidebar,
            b: mainbox
          }
        }
      ]
    });

    // 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      data: {},
      methods: {},
      router
    });
  </script>
```

3. CSS 样式：

```
  <style>
    .header {
      border: 1px solid red;
    }

    .content{
      display: flex;
    }
    .sidebar {
      flex: 2;
      border: 1px solid green;
      height: 500px;
    }
```

```
    .mainbox{
      flex: 8;
      border: 1px solid blue;
      height: 500px;
    }
  </style>
```

## watch属性的使用

考虑一个问题：想要实现 名 和 姓 两个文本框的内容改变，则全名的文本框中的值也跟着改变；（用以前的知识如何实现？？？）

1. 监听data中属性的改变：

```
<div id="app">
    <input type="text" v-model="firstName"> +
    <input type="text" v-model="lastName"> =
    <span>{{fullName}}</span>
  </div>

  <script>
    // 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      data: {
        firstName: 'jack',
        lastName: 'chen',
        fullName: 'jack - chen'
      },
      methods: {},
      watch: {
        'firstName': function (newVal, oldVal) { // 第一个参数是新数据，第二个参数是旧数据
          this.fullName = newVal + ' - ' + this.lastName;
        },
        'lastName': function (newVal, oldVal) {
          this.fullName = this.firstName + ' - ' + newVal;
        }
      }
    });
  </script>
```

2. 监听路由对象的改变：

```
<div id="app">
    <router-link to="/login">登录</router-link>
    <router-link to="/register">注册</router-link>

    <router-view></router-view>
  </div>

  <script>
```

```
    var login = Vue.extend({
      template: '<h1>登录组件</h1>'
    });

    var register = Vue.extend({
      template: '<h1>注册组件</h1>'
    });

    var router = new VueRouter({
      routes: [
        { path: "/login", component: login },
        { path: "/register", component: register }
      ]
    });

    // 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      data: {},
      methods: {},
      router: router,
      watch: {
        '$route': function (newVal, oldVal) {
          if (newVal.path === '/login') {
            console.log('这是登录组件');
          }
        }
      }
    });
  </script>
```

## `computed` 计算属性的使用

1. 默认只有 `getter` 的计算属性：

```
<div id="app">
    <input type="text" v-model="firstName"> +
    <input type="text" v-model="lastName"> =
    <span>{{fullName}}</span>
  </div>

  <script>
    // 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      data: {
        firstName: 'jack',
        lastName: 'chen'
      },
      methods: {},
      computed: { // 计算属性； 特点：当计算属性中所以来的任何一个 data 属性改变之后，都会重新触发 本计算属性 的重新计算，从而更新 fullName 的值
```

```
        fullName() {
          return this.firstName + ' - ' + this.lastName;
        }
      }
    });
  </script>
```

2. 定义有`getter`和`setter`的计算属性：

```
<div id="app">
    <input type="text" v-model="firstName">
    <input type="text" v-model="lastName">
    <!-- 点击按钮重新为 计算属性 fullName 赋值 -->
    <input type="button" value="修改fullName" @click="changeName">

    <span>{{fullName}}</span>
  </div>

  <script>
    // 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      data: {
        firstName: 'jack',
        lastName: 'chen'
      },
      methods: {
        changeName() {
          this.fullName = 'TOM - chen2';
        }
      },
      computed: {
        fullName: {
          get: function () {
            return this.firstName + ' - ' + this.lastName;
          },
          set: function (newVal) {
            var parts = newVal.split(' - ');
            this.firstName = parts[0];
            this.lastName = parts[1];
          }
        }
      }
    });
  </script>
```

## `watch`、`computed`和`methods`之间的对比

1. `computed`属性的结果会被缓存，除非依赖的响应式属性变化才会重新计算。主要当作属性来使用；
2. `methods`方法表示一个具体的操作，主要书写业务逻辑；

3. `watch` 一个对象，键是需要观察的表达式，值是对应回调函数。主要用来监听某些特定数据的变化，从而进行某些具体的业务逻辑操作；可以看作是 `computed` 和 `methods` 的结合体；

## `nrm` 的安装使用

作用：提供了一些最常用的NPM包镜像地址，能够让我们快速的切换安装包时候的服务器地址； 什么是镜像：原来包刚一开始是只存在于国外的NPM服务器，但是由于网络原因，经常访问不到，这时候，我们可以在国内，创建一个和官网完全一样的NPM服务器，只不过，数据都是从人家那里拿过来的，除此之外，使用方式完全一样；

1. 运行 `npm i nrm -g` 全局安装 `nrm` 包；
2. 使用 `nrm ls` 查看当前所有可用的镜像源地址以及当前所使用的镜像源地址；
3. 使用 `nrm use npm` 或 `nrm use taobao` 切换不同的镜像源地址；

## 相关文件

1. URL中的hash（井号）

# Vue.js - Day4

## 父组件向子组件传值

1. 组件实例定义方式，注意：一定要使用 `props` 属性来定义父组件传递过来的数据

```
<script>
    // 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      data: {
        msg: '这是父组件中的消息'
      },
      components: {
        son: {
          template: '<h1>这是子组件 --- {{finfo}}</h1>',
          props: ['finfo']
        }
      }
    });
  </script>
```

2. 使用 `v-bind` 或简化指令，将数据传递到子组件中：

```
<div id="app">
    <son :finfo="msg"></son>
  </div>
```

## 子组件向父组件传值

1. 原理：父组件将方法的引用，传递到子组件内部，子组件在内部调用父组件传递过来的方法，同时把要发送给父组件的数据，在调用方法的时候当作参数传递进去；
2. 父组件将方法的引用传递给子组件，其中，`getMsg` 是父组件中 `methods` 中定义的方法名称，`func` 是子组件调用传递过来方法时候的方法名称

```
<son @func="getMsg"></son>
```

3. 子组件内部通过 `this.$emit('方法名', 要传递的数据)` 方式，来调用父组件中的方法，同时把数据传递给父组件使用

```
<div id="app">
    <!-- 引用父组件 -->
    <son @func="getMsg"></son>

    <!-- 组件模板定义 -->
    <script type="x-template" id="son">
```

```
    <div>
      <input type="button" value="向父组件传值" @click="sendMsg" />
    </div>
  </script>
</div>

<script>
  // 子组件的定义方式
  Vue.component('son', {
    template: '#son', // 组件模板Id
    methods: {
      sendMsg() { // 按钮的点击事件
        this.$emit('func', 'OK'); // 调用父组件传递过来的方法，同时把数据传递出去
      }
    }
  });

  // 创建 Vue 实例，得到 ViewModel
  var vm = new Vue({
    el: '#app',
    data: {},
    methods: {
      getMsg(val){ // 子组件中，通过 this.$emit() 实际调用的方法，在此进行定义
        alert(val);
      }
    }
  });
</script>
```

# 组件中data和props的区别

# 评论列表案例

目标：主要练习父子组件之间传值

# 使用 `this.$refs` 来获取元素和组件

```
<div id="app">
  <div>
    <input type="button" value="获取元素内容" @click="getElement" />
    <!-- 使用 ref 获取元素 -->
    <h1 ref="myh1">这是一个大大的H1</h1>

    <hr>
    <!-- 使用 ref 获取子组件 -->
    <my-com ref="mycom"></my-com>
  </div>
</div>

<script>
  Vue.component('my-com', {
```

```
    template: '<h5>这是一个子组件</h5>',
    data() {
      return {
        name: '子组件'
      }
    }
  });

  // 创建 Vue 实例，得到 ViewModel
  var vm = new Vue({
    el: '#app',
    data: {},
    methods: {
      getElement() {
        // 通过 this.$refs 来获取元素
        console.log(this.$refs.myh1.innerText);
        // 通过 this.$refs 来获取组件
        console.log(this.$refs.mycom.name);
      }
    }
  });
</script>
```

# 什么是路由

1. **后端路由：**对于普通的网站，所有的超链接都是URL地址，所有的URL地址都对应服务器上对应的资源；
2. **前端路由：**对于单页面应用程序来说，主要通过URL中的hash(#号)来实现不同页面之间的切换，同时，hash有一个特点：HTTP请求中不会包含hash相关的内容；所以，单页面程序中的页面跳转主要用hash实现；
3. 在单页面应用程序中，这种通过hash改变来切换页面的方式，称作前端路由（区别于后端路由）；

# 在 vue 中使用 vue-router

1. 导入 vue-router 组件类库：

```
<!-- 1. 导入 vue-router 组件类库 -->
  <script src="./lib/vue-router-2.7.0.js"></script>
```

2. 使用 router-link 组件来导航

```
<!-- 2. 使用 router-link 组件来导航 -->
<router-link to="/login">登录</router-link>
<router-link to="/register">注册</router-link>
```

3. 使用 router-view 组件来显示匹配到的组件

```
<!-- 3. 使用 router-view 组件来显示匹配到的组件 -->
<router-view></router-view>
```

4. 创建使用 `Vue.extend` 创建组件

```
    // 4.1 使用 Vue.extend 来创建登录组件
    var login = Vue.extend({
      template: '<h1>登录组件</h1>'
    });

    // 4.2 使用 Vue.extend 来创建注册组件
    var register = Vue.extend({
      template: '<h1>注册组件</h1>'
    });
```

5. 创建一个路由 router 实例，通过 routers 属性来定义路由匹配规则

```
// 5. 创建一个路由 router 实例，通过 routers 属性来定义路由匹配规则
    var router = new VueRouter({
      routes: [
        { path: '/login', component: login },
        { path: '/register', component: register }
      ]
    });
```

6. 使用 router 属性来使用路由规则

```
// 6. 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      router: router // 使用 router 属性来使用路由规则
    });
```

# 使用tag属性指定router-link渲染的标签类型

# 设置路由重定向

# 设置路由高亮

# 设置路由切换动效

# 在路由规则中定义参数

1. 在规则中定义参数：

```
{ path: '/register/:id', component: register }
```

2. 通过 `this.$route.params` 来获取路由中的参数：

```
var register = Vue.extend({
      template: '<h1>注册组件 --- {{this.$route.params.id}}</h1>'
    });
```

## 使用 `childrens` 属性实现路由嵌套

```
<div id="app">
  <router-link to="/account">Account</router-link>

  <router-view></router-view>
</div>

<script>
  // 父路由中的组件
  const account = Vue.extend({
    template: `<div>
      这是account组件
      <router-link to="/account/login">login</router-link> |
      <router-link to="/account/register">register</router-link>
      <router-view></router-view>
    </div>`
  });

  // 子路由中的 login 组件
  const login = Vue.extend({
    template: '<div>登录组件</div>'
  });

  // 子路由中的 register 组件
  const register = Vue.extend({
    template: '<div>注册组件</div>'
  });

  // 路由实例
  var router = new VueRouter({
    routes: [
      { path: '/', redirect: '/account/login' }, // 使用 redirect 实现路由重定向
      {
        path: '/account',
        component: account,
        children: [ // 通过 children 数组属性，来实现路由的嵌套
          { path: 'login', component: login }, // 注意，子路由的开头位置，不要加 / 路径符
          { path: 'register', component: register }
        ]
      }
    ]
  });

  // 创建 Vue 实例，得到 ViewModel
  var vm = new Vue({
    el: '#app',
    data: {},
    methods: {},
    components: {
      account
    },
```

```
    router: router
  });
</script>
```

# 命名视图实现经典布局

1. 标签代码结构：

```
<div id="app">
    <router-view></router-view>
    <div class="content">
      <router-view name="a"></router-view>
      <router-view name="b"></router-view>
    </div>
  </div>
```

2. JS代码：

```
<script>
    var header = Vue.component('header', {
      template: '<div class="header">header</div>'
    });

    var sidebar = Vue.component('sidebar', {
      template: '<div class="sidebar">sidebar</div>'
    });

    var mainbox = Vue.component('mainbox', {
      template: '<div class="mainbox">mainbox</div>'
    });

    // 创建路由对象
    var router = new VueRouter({
      routes: [
        {
          path: '/', components: {
            default: header,
            a: sidebar,
            b: mainbox
          }
        }
      ]
    });

    // 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      data: {},
      methods: {},
      router
    });
```

```
  </script>
```

3. CSS 样式：

```
  <style>
    .header {
      border: 1px solid red;
    }

    .content{
      display: flex;
    }
    .sidebar {
      flex: 2;
      border: 1px solid green;
      height: 500px;
    }
    .mainbox{
      flex: 8;
      border: 1px solid blue;
      height: 500px;
    }
  </style>
```

## `watch` 属性的使用

考虑一个问题：想要实现 `名` 和 `姓` 两个文本框的内容改变，则全名的文本框中的值也跟着改变；（用以前的知识如何实现？？？）

1. 监听 `data` 中属性的改变：

```
<div id="app">
    <input type="text" v-model="firstName"> +
    <input type="text" v-model="lastName"> =
    <span>{{fullName}}</span>
  </div>

  <script>
    // 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      data: {
        firstName: 'jack',
        lastName: 'chen',
        fullName: 'jack - chen'
      },
      methods: {},
      watch: {
        'firstName': function (newVal, oldVal) { // 第一个参数是新数据，第二个参数是旧数据
          this.fullName = newVal + ' - ' + this.lastName;
        },
```

```
        'lastName': function (newVal, oldVal) {
          this.fullName = this.firstName + ' - ' + newVal;
        }
      }
    });
  </script>
```

2. 监听路由对象的改变：

```
<div id="app">
    <router-link to="/login">登录</router-link>
    <router-link to="/register">注册</router-link>

    <router-view></router-view>
  </div>

  <script>
    var login = Vue.extend({
      template: '<h1>登录组件</h1>'
    });

    var register = Vue.extend({
      template: '<h1>注册组件</h1>'
    });

    var router = new VueRouter({
      routes: [
        { path: "/login", component: login },
        { path: "/register", component: register }
      ]
    });

    // 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      data: {},
      methods: {},
      router: router,
      watch: {
        '$route': function (newVal, oldVal) {
          if (newVal.path === '/login') {
            console.log('这是登录组件');
          }
        }
      }
    });
  </script>
```

## `computed` 计算属性的使用

1. 默认只有 `getter` 的计算属性：

```
<div id="app">
    <input type="text" v-model="firstName"> +
    <input type="text" v-model="lastName"> =
    <span>{{fullName}}</span>
  </div>

  <script>
    // 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      data: {
        firstName: 'jack',
        lastName: 'chen'
      },
      methods: {},
      computed: { // 计算属性； 特点：当计算属性中所以来的任何一个 data 属性改变之后，都会重新触发本计算属性 的重新计算，从而更新 fullName 的值
        fullName() {
          return this.firstName + ' - ' + this.lastName;
        }
      }
    });
  </script>
```

2. 定义有`getter`和`setter`的计算属性：

```
<div id="app">
    <input type="text" v-model="firstName">
    <input type="text" v-model="lastName">
    <!-- 点击按钮重新为 计算属性 fullName 赋值 -->
    <input type="button" value="修改fullName" @click="changeName">

    <span>{{fullName}}</span>
  </div>

  <script>
    // 创建 Vue 实例，得到 ViewModel
    var vm = new Vue({
      el: '#app',
      data: {
        firstName: 'jack',
        lastName: 'chen'
      },
      methods: {
        changeName() {
          this.fullName = 'TOM - chen2';
        }
      },
      computed: {
        fullName: {
          get: function () {
            return this.firstName + ' - ' + this.lastName;
          },
```

```
            set: function (newVal) {
              var parts = newVal.split(' - ');
              this.firstName = parts[0];
              this.lastName = parts[1];
            }
          }
        }
      });
    </script>
```

## `watch`、`computed`和`methods`之间的对比

1. `computed`属性的结果会被缓存，除非依赖的响应式属性变化才会重新计算。主要当作属性来使用；
2. `methods`方法表示一个具体的操作，主要书写业务逻辑；
3. `watch`一个对象，键是需要观察的表达式，值是对应回调函数。主要用来监听某些特定数据的变化，从而进行某些具体的业务逻辑操作；可以看作是`computed`和`methods`的结合体；

## `nrm`的安装使用

作用：提供了一些最常用的NPM包镜像地址，能够让我们快速的切换安装包时候的服务器地址； 什么是镜像：原来包刚一开始是只存在于国外的NPM服务器，但是由于网络原因，经常访问不到，这时候，我们可以在国内，创建一个和官网完全一样的NPM服务器，只不过，数据都是从人家那里拿过来的，除此之外，使用方式完全一样；

1. 运行`npm i nrm -g`全局安装`nrm`包；
2. 使用`nrm ls`查看当前所有可用的镜像源地址以及当前所使用的镜像源地址；
3. 使用`nrm use npm`或`nrm use taobao`切换不同的镜像源地址；

## 相关文件

1. URL中的hash（井号）

# Vue.js - Day5 - Webpack

## 在网页中会引用哪些常见的静态资源?

- JS
- .js .jsx .coffee .ts（TypeScript 类 C# 语言）
- CSS
- .css .less .sass .scss
- Images
- .jpg .png .gif .bmp .svg
- 字体文件（Fonts）
- .svg .ttf .eot .woff .woff2
- 模板文件
- .ejs .jade .vue【这是在webpack中定义组件的方式，推荐这么用】

## 网页中引入的静态资源多了以后有什么问题？？？

1. 网页加载速度慢， 因为 我们要发起很多的二次请求；
2. 要处理错综复杂的依赖关系

## 如何解决上述两个问题

1. 合并、压缩、精灵图、图片的Base64编码
2. 可以使用之前学过的requireJS、也可以使用webpack可以解决各个包之间的复杂依赖关系；

## 什么是webpack?

webpack 是前端的一个项目构建工具，它是基于 Node.js 开发出来的一个前端工具；

## 如何完美实现上述的2种解决方案

1. 使用Gulp， 是基于 task 任务的；
2. 使用Webpack， 是基于整个项目进行构建的；

- 借助于webpack这个前端自动化构建工具，可以完美实现资源的合并、打包、压缩、混淆等诸多功能。
- 根据官网的图片介绍webpack打包的过程
- webpack官网

## webpack安装的两种方式

1. 运行 `npm i webpack -g` 全局安装webpack，这样就能在全局使用webpack的命令

2. 在项目根目录中运行 `npm i webpack --save-dev` 安装到项目依赖中

# 初步使用webpack打包构建列表隔行变色案例

1. 运行 `npm init` 初始化项目，使用npm管理项目中的依赖包
2. 创建项目基本的目录结构
3. 使用 `cnpm i jquery --save` 安装jquery类库
4. 创建 `main.js` 并书写各行变色的代码逻辑：

```
    // 导入jquery类库
    import $ from 'jquery'

    // 设置偶数行背景色，索引从0开始，0是偶数
    $('#list li:even').css('backgroundColor','lightblue');
    // 设置奇数行背景色
    $('#list li:odd').css('backgroundColor','pink');
```

5. 直接在页面上引用 `main.js` 会报错，因为浏览器不认识 `import` 这种高级的JS语法，需要使用webpack进行处理，webpack默认会把这种高级的语法转换为低级的浏览器能识别的语法；
6. 运行 `webpack 入口文件路径 输出文件路径` 对 `main.js` 进行处理：

```
webpack src/js/main.js dist/bundle.js
```

# 使用webpack的配置文件简化打包时候的命令

1. 在项目根目录中创建 `webpack.config.js`
2. 由于运行webpack命令的时候，webpack需要指定入口文件和输出文件的路径，所以，我们需要在 `webpack.config.js` 中配置这两个路径：

```
    // 导入处理路径的模块
    var path = require('path');

    // 导出一个配置对象，将来webpack在启动的时候，会默认来查找webpack.config.js，并读取这个文件中导出的配置对象，来进行打包处理
    module.exports = {
        entry: path.resolve(__dirname, 'src/js/main.js'), // 项目入口文件
        output: { // 配置输出选项
            path: path.resolve(__dirname, 'dist'), // 配置输出的路径
            filename: 'bundle.js' // 配置输出的文件名
        }
    }
```

# 实现webpack的实时打包构建

1. 由于每次重新修改代码之后，都需要手动运行webpack打包的命令，比较麻烦，所以使用 `webpack-dev-server` 来实现代码实时打包编译，当修改代码之后，会自动进行打包构建。
2. 运行 `cnpm i webpack-dev-server --save-dev` 安装到开发依赖

3. 安装完成之后，在命令行直接运行`webpack-dev-server`来进行打包，发现报错，此时需要借助于`package.json`文件中的指令，来进行运行`webpack-dev-server`命令，在`scripts`节点下新增`"dev": "webpack-dev-server"`指令，发现可以进行实时打包，但是dist目录下并没有生成`bundle.js`文件，这是因为`webpack-dev-server`将打包好的文件放在了内存中

- 把`bundle.js`放在内存中的好处是：由于需要实时打包编译，所以放在内存中速度会非常快
- 这个时候访问webpack-dev-server启动的`http://localhost:8080/`网站，发现是一个文件夹的面板，需要点击到src目录下，才能打开我们的index首页，此时引用不到bundle.js文件，需要修改index.html中script的src属性为:`<script src="../bundle.js"></script>`
- 为了能在访问`http://localhost:8080/`的时候直接访问到index首页，可以使用`--contentBase src`指令来修改dev指令，指定启动的根目录：

```
  "dev": "webpack-dev-server --contentBase src"
```

同时修改index页面中script的src属性为`<script src="bundle.js"></script>`

# 使用`html-webpack-plugin`插件配置启动页面

由于使用`--contentBase`指令的过程比较繁琐，需要指定启动的目录，同时还需要修改index.html中script标签的src属性，所以推荐大家使用`html-webpack-plugin`插件配置启动页面.

1. 运行`cnpm i html-webpack-plugin --save-dev`安装到开发依赖
2. 修改`webpack.config.js`配置文件如下：

```
    // 导入处理路径的模块
    var path = require('path');
    // 导入自动生成HTML文件的插件
    var htmlWebpackPlugin = require('html-webpack-plugin');

    module.exports = {
        entry: path.resolve(__dirname, 'src/js/main.js'), // 项目入口文件
        output: { // 配置输出选项
            path: path.resolve(__dirname, 'dist'), // 配置输出的路径
            filename: 'bundle.js' // 配置输出的文件名
        },
        plugins:[ // 添加plugins节点配置插件
            new htmlWebpackPlugin({
                template:path.resolve(__dirname, 'src/index.html'),//模板路径
                filename:'index.html'//自动生成的HTML文件的名称
            })
        ]
    }
```

3. 修改`package.json`中`script`节点中的dev指令如下：

```
 "dev": "webpack-dev-server"
```

4. 将index.html中script标签注释掉，因为`html-webpack-plugin`插件会自动把bundle.js注入到index.html页面中!

# 实现自动打开浏览器、热更新和配置浏览器的默认端口号

**注意：热更新在JS中表现的不明显，可以从一会儿要讲到的CSS身上进行介绍说明！**

## 方式1：

- 修改 `package.json` 的script节点如下，其中 `--open` 表示自动打开浏览器， `--port 4321` 表示打开的端口号为4321， `--hot` 表示启用浏览器热更新：

```
"dev": "webpack-dev-server --hot --port 4321 --open"
```

## 方式2：

1. 修改 `webpack.config.js` 文件，新增 `devServer` 节点如下：

```
devServer:{
        hot:true,
        open:true,
        port:4321
    }
```

2. 在头部引入 `webpack` 模块：

```
var webpack = require('webpack');
```

3. 在 `plugins` 节点下新增：

```
new webpack.HotModuleReplacementPlugin()
```

# 使用webpack打包css文件

1. 运行 `cnpm i style-loader css-loader --save-dev`
2. 修改 `webpack.config.js` 这个配置文件：

```
module: { // 用来配置第三方loader模块的
        rules: [ // 文件的匹配规则
            { test: /\.css$/, use: ['style-loader', 'css-loader'] }//处理css文件的规则
        ]
    }
```

3. 注意： `use` 表示使用哪些模块来处理 `test` 所匹配到的文件； `use` 中相关loader模块的调用顺序是从后向前调用的；

# 使用webpack打包less文件

1. 运行 `cnpm i less-loader less -D`
2. 修改 `webpack.config.js` 这个配置文件：

```
{ test: /\.less$/, use: ['style-loader', 'css-loader', 'less-loader'] },
```

# 使用webpack打包sass文件

1. 运行 `cnpm i sass-loader node-sass --save-dev`
2. 在 `webpack.config.js` 中添加处理sass文件的loader模块：

```
{ test: /\.scss$/, use: ['style-loader', 'css-loader', 'sass-loader'] }
```

# 使用webpack处理css中的路径

1. 运行 `cnpm i url-loader file-loader --save-dev`
2. 在 `webpack.config.js` 中添加处理url路径的loader模块：

```
{ test: /\.(png|jpg|gif)$/, use: 'url-loader' }
```

3. 可以通过 `limit` 指定进行base64编码的图片大小；只有小于指定字节（byte）的图片才会进行base64编码：

```
{ test: /\.(png|jpg|gif)$/, use: 'url-loader?limit=43960' },
```

# 使用babel处理高级JS语法

1. 运行 `cnpm i babel-core babel-loader babel-plugin-transform-runtime --save-dev` 安装babel的相关loader包
2. 运行 `cnpm i babel-preset-es2015 babel-preset-stage-0 --save-dev` 安装babel转换的语法
3. 在 `webpack.config.js` 中添加相关loader模块，其中需要注意的是，一定要把 `node_modules` 文件夹添加到排除项：

```
{ test: /\.js$/, use: 'babel-loader', exclude: /node_modules/ }
```

4. 在项目根目录中添加 `.babelrc` 文件，并修改这个配置文件如下：

```
{
    "presets":["es2015", "stage-0"],
    "plugins":["transform-runtime"]
}
```

5. **注意：语法插件 `babel-preset-es2015` 可以更新为 `babel-preset-env`，它包含了所有的ES相关的语法；**

# 相关文章

babel-preset-env：你需要的唯一Babel插件 Runtime transform 运行时编译es6

# Vue.js - day6

## 注意：

有时候使用`npm i node-sass -D`装不上，这时候，就必须使用`cnpm i node-sass -D`

## 在普通页面中使用render函数渲染组件

## 在webpack中配置.vue组件页面的解析

1. 运行`cnpm i vue -S`将vue安装为运行依赖；
2. 运行`cnpm i vue-loader vue-template-compiler -D`将解析转换vue的包安装为开发依赖；
3. 运行`cnpm i style-loader css-loader -D`将解析转换CSS的包安装为开发依赖，因为.vue文件中会写CSS样式；
4. 在`webpack.config.js`中，添加如下`module`规则：

```
module: {
    rules: [
      { test: /\.css$/, use: ['style-loader', 'css-loader'] },
      { test: /\.vue$/, use: 'vue-loader' }
    ]
  }
```

5. 创建`App.js`组件页面：

```
    <template>
      <!-- 注意：在 .vue 的组件中，template 中必须有且只有唯一的根元素进行包裹，一般都用 div 当作唯一的根元素 -->
      <div>
        <h1>这是APP组件 - {{msg}}</h1>
```

```
    <h3>我是h3</h3>
  </div>
</template>

<script>
// 注意：在 .vue 的组件中，通过 script 标签来定义组件的行为，需要使用 ES6 中提供的 export default 方式，导出一个vue实例对象
export default {
  data() {
    return {
      msg: 'OK'
    }
  }
}
</script>

<style scoped>
h1 {
  color: red;
}
</style>
```

6. 创建`main.js`入口文件：

```
// 导入 Vue 组件
import Vue from 'vue'

// 导入 App组件
```

```
import App from './components/App.vue'


// 创建一个 Vue 实例，使用 render 函数，渲染指定的组件
var vm = new Vue({
  el: '#app',
  render: c => c(App)
});
```

# 在使用webpack构建的Vue项目中使用模板对象?

1. 在 `webpack.config.js` 中添加 `resolve` 属性:

```
resolve: {
    alias: {
      'vue$': 'vue/dist/vue.esm.js'
    }
  }
```

# ES6中语法使用总结

1. 使用 `export default` 和 `export` 导出模块中的成员; 对应ES5中的 `module.exports` 和 `export`
2. 使用 `import ** from **` 和 `import '路径'` 还有 `import {a, b} from '模块标识'` 导入其他模块
3. 使用箭头函数: `(a, b)=> { return a-b; }`

# 在vue组件页面中，集成vue-router路由模块

vue-router官网

1. 导入路由模块:

```
import VueRouter from 'vue-router'
```

2. 安装路由模块:

```
Vue.use(VueRouter);
```

3. 导入需要展示的组件:

```
import login from './components/account/login.vue'

import register from './components/account/register.vue'
```

4. 创建路由对象:

```
var router = new VueRouter({
  routes: [
    { path: '/', redirect: '/login' },
    { path: '/login', component: login },
    { path: '/register', component: register }
  ]
});
```

5. 将路由对象，挂载到 Vue 实例上:

```
var vm = new Vue({
  el: '#app',
  // render: c => { return c(App) }
  render(c) {
    return c(App);
  },
  router // 将路由对象，挂载到 Vue 实例上
});
```

6. 改造App.vue组件，在 template 中，添加 `router-link` 和 `router-view`:

```
    <router-link to="/login">登录</router-link>

    <router-link to="/register">注册</router-link>


    <router-view></router-view>
```

## 组件中的css作用域问题

## 抽离路由为单独的模块

## 使用 饿了么的 MintUI 组件

Github 仓储地址

Mint-UI官方文档

1. 导入所有MintUI组件：

```
import MintUI from 'mint-ui'
```

2. 导入样式表：

```
import 'mint-ui/lib/style.css'
```

3. 在 vue 中使用 MintUI：

```
Vue.use(MintUI)
```

4. 使用的例子：

```
<mt-button type="primary" size="large">primary</mt-button>
```

# 使用 MUI 组件

官网首页

文档地址

1. 导入 MUI 的样式表：

```
import '../lib/mui/css/mui.min.css'
```

2. 在 `webpack.config.js` 中添加新的loader规则：

```
{ test: /\.(png|jpg|gif|ttf)$/, use: 'url-loader' }
```

3. 根据官方提供的文档和example，尝试使用相关的组件

# 将项目源码托管到oschina中

1. 点击头像 -> 修改资料 -> SSH公钥 如何生成SSH公钥
2. 创建自己的空仓储，使用 `git config --global user.name "用户名"` 和 `git config --global user.email ***@**.com` 来全局配置提交时用户的名称和邮箱
3. 使用 `git init` 在本地初始化项目
4. 使用 `touch README.md` 和 `touch .gitignore` 来创建项目的说明文件和忽略文件；
5. 使用 `git add .` 将所有文件托管到 git 中
6. 使用 `git commit -m "init project"` 将项目进行本地提交
7. 使用 `git remote add origin 仓储地址` 将本地项目和远程仓储连接，并使用origin最为远程仓储的别名
8. 使用 `git push -u origin master` 将本地代码push到仓储中

# App.vue 组件的基本设置

1. 头部的固定导航栏使用 `Mint-UI` 的 `Header` 组件；
2. 底部的页签使用 `mui` 的 `tabbar`；
3. 购物车的图标，使用 `icons-extra` 中的 `mui-icon-extra mui-icon-extra-cart`，同时，应该把其依赖的字体图标文件 `mui-icons-extra.ttf`，复制到 `fonts` 目录下！
4. 将底部的页签，改造成 `router-link` 来实现单页面的切换；
5. Tab Bar 路由激活时候设置高亮的两种方式：

- 全局设置样式如下：

```
.router-link-active{
    color:#007aff !important;
}
```

- 或者在 `new VueRouter` 的时候，通过 `linkActiveClass` 来指定高亮的类：

```
// 创建路由对象
var router = new VueRouter({
  routes: [
    { path: '/', redirect: '/home' }
  ],
  linkActiveClass: 'mui-active'
});
```

# 实现 tabbar 页签不同组件页面的切换

1. 将 tabbar 改造成 `router-link` 形式，并指定每个连接的 `to` 属性；
2. 在入口文件中导入需要展示的组件，并创建路由对象：

```
// 导入需要展示的组件
import Home from './components/home/home.vue'
import Member from './components/member/member.vue'
import Shopcar from './components/shopcar/shopcar.vue'
import Search from './components/search/search.vue'

// 创建路由对象
var router = new VueRouter({
  routes: [
```

```
        { path: '/', redirect: '/home' },
        { path: '/home', component: Home },
        { path: '/member', component: Member },
        { path: '/shopcar', component: Shopcar },
        { path: '/search', component: Search }
      ],
      linkActiveClass: 'mui-active'
    });
```

# 使用 mt-swipe 轮播图组件

1. 假数据：

```
lunbo: [
        'http://www.itcast.cn/images/slidead/BEIJING/2017440109442800.jpg',
        'http://www.itcast.cn/images/slidead/BEIJING/2017511009514700.jpg',
        'http://www.itcast.cn/images/slidead/BEIJING/2017421414422600.jpg'
      ]
```

2. 引入轮播图组件：

```
<!-- Mint-UI 轮播图组件 -->
    <div class="home-swipe">
      <mt-swipe :auto="4000">
        <mt-swipe-item v-for="(item, i) in lunbo" :key="i">
          <img :src="item" alt="">
        </mt-swipe-item>
      </mt-swipe>
```

```
    </div>
  </div>
```

## 在 .vue 组件中使用 vue-resource 获取数据

1. 运行 `cnpm i vue-resource -S` 安装模块
2. 导入 vue-resource 组件

```
import VueResource from 'vue-resource'
```

3. 在vue中使用 vue-resource 组件

```
Vue.use(VueResource);
```

# day7

## 使用mui的`tab-top-webview-main`完成分类滑动栏

### 兼容问题

1. 和 App.vue 中的 `router-link` 身上的类名 `mui-tab-item` 存在兼容性问题，导致tab栏失效，可以把`mui-tab-item`改名为`mui-tab-item1`，并复制相关的类样式，来解决这个问题；

```
.mui-bar-tab .mui-tab-item1.mui-active {
  color: #007aff;
}

.mui-bar-tab .mui-tab-item1 {
  display: table-cell;
  overflow: hidden;
  width: 1%;
  height: 50px;
  text-align: center;
  vertical-align: middle;
  white-space: nowrap;
  text-overflow: ellipsis;
  color: #929292;
}

.mui-bar-tab .mui-tab-item1 .mui-icon {
  top: 3px;
  width: 24px;
  height: 24px;
  padding-top: 0;
  padding-bottom: 0;
}

.mui-bar-tab .mui-tab-item1 .mui-icon~.mui-tab-label {
  font-size: 11px;
  display: block;
  overflow: hidden;
  text-overflow: ellipsis;
}
```

2. `tab-top-webview-main`组件第一次显示到页面中的时候，无法被滑动的解决方案：

- 先导入 mui 的JS文件:

```
import mui from '../../../lib/mui/js/mui.min.js'
```

- 在 组件的 `mounted` 事件钩子中，注册 mui 的滚动事件：

```
    mounted() {
        // 需要在组件的 mounted 事件钩子中，注册 mui 的 scroll 滚动事件
        mui('.mui-scroll-wrapper').scroll({
          deceleration: 0.0005 //flick 减速系数，系数越大，滚动速度越慢，滚动距离越小，默认值
0.0006
        });
    }
```

3. 滑动的时候报警告：`Unable to preventDefault inside passive event listener due to target being treated as passive. See https://www.chromestatus.com/features/5093566007214080`

```
解决方法，可以加上* { touch-action: none; } 这句样式去掉。
```

原因：（是chrome为了提高页面的滑动流畅度而新折腾出来的一个东西） http://www.cnblogs.com/pearl07/p/6589114.html https://developer.mozilla.org/zh-CN/docs/Web/CSS/touch-action

# 移除严格模式

babel-plugin-transform-remove-strict-mode

# vue-preview

一个Vue集成PhotoSwipe图片预览插件

# Day10

## 开启Apache的gzip压缩

要让apache支持gzip功能，要用到deflate_Module和headers_Module。打开apache的配置文件httpd.conf，大约在105行左右，找到以下两行内容：（这两行不是连续在一起的）

```
#LoadModule deflate_module modules/mod_deflate.so
#LoadModule headers_module modules/mod_headers.so
```

然后将其前面的“#”注释删掉，表示开启gzip压缩功能。开启以后还需要进行相关配置。在httpd.conf文件的最后添加以下内容即可：

```
<IfModule deflate_module>
    #必须的，就像一个开关一样，告诉apache对传输到浏览器的内容进行压缩
    SetOutputFilter DEFLATE
    DeflateCompressionLevel 9
</IfModule>
```

最少需要加上以上内容，才可以生gzip功能生效。由于没有做其它的额外配置，所以其它相关的配置均使用Apache的默认设置。这里说一下参数“DeflateCompressionLevel”，它表示压缩级别，值从1到9，值越大表示压缩的越厉害。

## 使用ngrok将本机映射为一个外网的Web服务器

注意：由于默认使用的美国的服务器进行中间转接，所以访问速度炒鸡慢，访问时可启用FQ软件，提高网页打开速度！