基础

安装

对于制作原型或学习，你可以这样使用最新版本：

```
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
```

对于生产环境，我们推荐链接到一个明确的版本号和构建文件，以避免新版本造成的不可预期的破坏：

```
<script src="https://cdn.jsdelivr.net/npm/vue@2.6.12"></script>
```

如果你使用原生 ES Modules，这里也有一个兼容 ES Module 的构建文件：

```
<script type="module">
  import Vue from
'https://cdn.jsdelivr.net/npm/vue@2.6.12/dist/vue.esm.browser.js'
</script>
```

你可以在 cdn.jsdelivr.net/npm/vue 浏览 NPM 包的源代码。

Vue 也可以在 unpkg 和 cdnjs 上获取 (cdnjs 的版本更新可能略滞后)。

请确认了解不同构建版本并在你发布的站点中使用**生产环境版本**，把 `vue.js` 换成 `vue.min.js`。这是一个更小的构建，可以带来比开发环境下更快的速度体验。

NPM

在用 Vue 构建大型应用时推荐使用 NPM 安装[1]。NPM 能很好地和诸如 webpack 或 Browserify 模块打包器配合使用。同时 Vue 也提供配套工具来开发单文件组件。

```
# 最新稳定版
$ npm install vue
```

简介

Vue.js 的核心是一个允许采用简洁的模板语法来声明式地将数据渲染进 DOM 的系统：

```
<div id="app">
  {{ message }}
</div>
var app = new Vue({
  el: '#app',
  data: {
    message: 'Hello Vue!'
  }
})
```

显示：Hello Vue!

我们已经成功创建了第一个 Vue 应用！看起来这跟渲染一个字符串模板非常类似，但是 Vue 在背后做了大量工作。现在数据和 DOM 已经被建立了关联，所有东西都是**响应式的**。我们要怎么确认呢？打开你的浏览器的 JavaScript 控制台 (就在这个页面打开)，并修改 `app.message` 的值，你将看到上例相应地更新。

条件与循环

条件：v-if

```
<div id="app-3">
  <p v-if="seen">现在你看到我了</p>
</div>
var app3 = new Vue({
  el: '#app-3',
  data: {
    seen: true
  }
})
```

循环：v-for

```
<div id="app-4">
  <ol>
    <li v-for="todo in todos">
      {{ todo.text }}
    </li>
  </ol>
</div>
```

```
var app4 = new Vue({
  el: '#app-4',
  data: {
    todos: [
      { text: '学习 JavaScript' },
      { text: '学习 Vue' },
      { text: '整个牛项目' }
    ]
  }
})
```

事件绑定

```
<div id="app-5">
  <p>{{ message }}</p>
  <button v-on:click="reverseMessage">反转消息</button>
</div>
```

```
var app5 = new Vue({
  el: '#app-5',
  data: {
    message: 'Hello Vue.js!'
  },
  methods: {
    reverseMessage: function () {
      this.message = this.message.split('').reverse().join('')
    }
  }
})
```

Vue实例

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

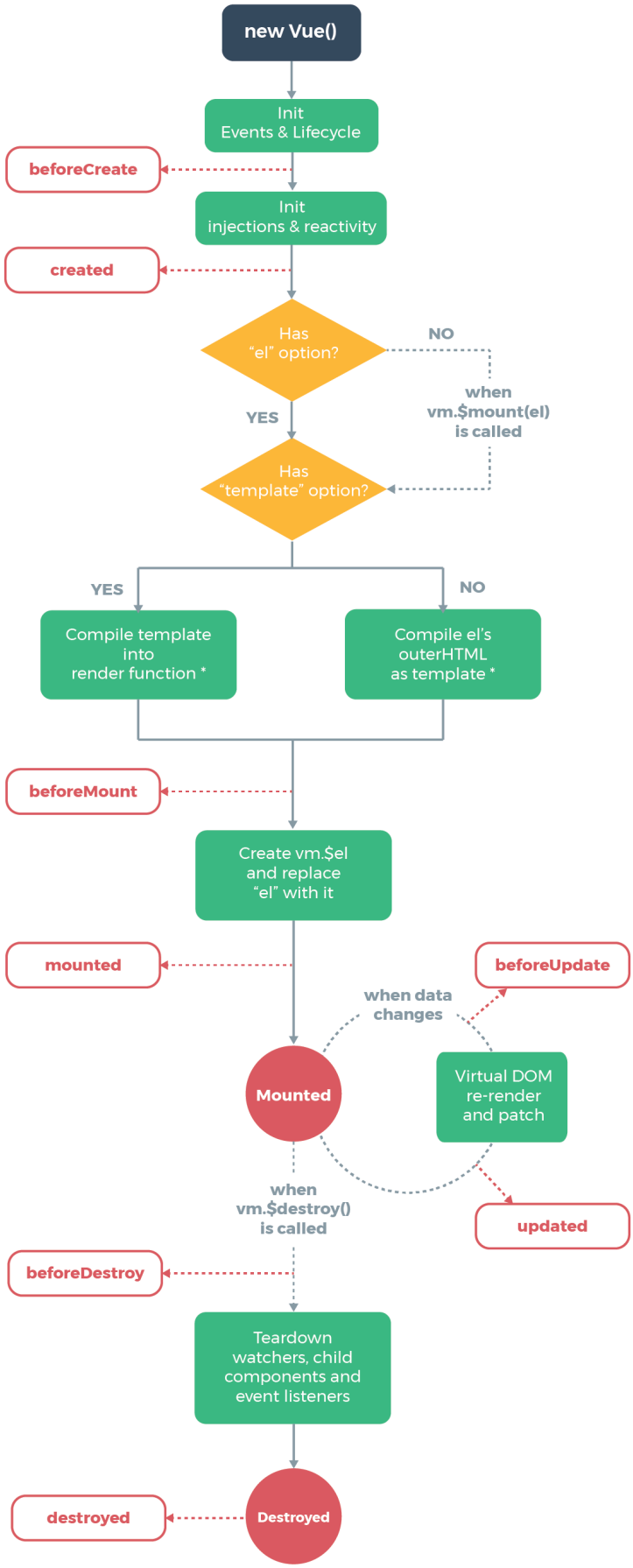
```
var vm = new Vue () {
    el: "", //定位dom元素
    data: { //需要传入的数据

    },
    methods: { //方法的定义

    }
}
```

生命周期

每个 Vue 实例在被创建时都要经过一系列的初始化过程——例如，需要设置数据监听、编译模板、将实例挂载到 DOM 并在数据变化时更新 DOM 等。同时在这个过程中也会运行一些叫做**生命周期钩子**的函数，这给了用户在不同阶段添加自己的代码的机会。

也就是说，用户可以通过不同的生命周期函数，在不同的生命周期完成自己的需求

```
new Vue({
  data: {
    a: 1
  },
  created: function () {
    // `this` 指向 vm 实例
    console.log('a is: ' + this.a)
  }
})
// => "a is: 1"
```



* template compilation is performed ahead-of-time if using a build step, e.g. single-file components

模板语法

Vue.js 使用了基于 HTML 的模板语法，允许开发者声明式地将 DOM 绑定至底层 Vue 实例的数据。所有 Vue.js 的模板都是合法的 HTML，所以能被遵循规范的浏览器和 HTML 解析器解析。

插值

文本

```
<span v-once>这个将不会改变: {{ msg }}</span>
```

HTML

双大括号会将数据解释为普通文本，而非 HTML 代码。为了输出真正的 HTML，你需要使用 `v-html` 指令：

```
<p>Using mustaches: {{ rawHtml }}</p>
<p>Using v-html directive: <span v-html="rawHtml"></span></p>
```

```
Using mustaches: <span style="color: red">This should be red.</span>
Using v-html directive: This should be red.
```

属性

Mustache 语法不能作用在 HTML attribute 上，遇到这种情况应该使用 `v-bind` 指令：

```
<div v-bind:id="dynamicId"></div>
```

对于布尔 attribute (它们只要存在就意味着值为 `true`)，`v-bind` 工作起来略有不同，在这个例子中：

```
<button v-bind:disabled="isButtonDisabled">Button</button>
```

如果 `isButtonDisabled` 的值是 `null`、`undefined` 或 `false`，则 `disabled` attribute 甚至不会被包含在渲染出来的 `<button>` 元素中。

语法糖：

```
<button :disabled="isButtonDisabled">Button</button>
```

表达式支持

对于所有的数据绑定，Vue.js 都提供了完全的 JavaScript 表达式支持。

```
{{ number + 1 }}

{{ ok ? 'YES' : 'NO' }}

{{ message.split('').reverse().join('') }}

<div v-bind:id="'list-' + id"></div>
```

这些表达式会在所属 Vue 实例的数据作用域下作为 JavaScript 被解析。有个限制就是，每个绑定都只能包含**单个表达式**，所以下面的例子都**不会**生效。

```
<!-- 这是语句，不是表达式 -->
{{ var a = 1 }}

<!-- 流控制也不会生效，请使用三元表达式 -->
{{ if (ok) { return message } }}
```

指令

参数

一些指令能够接收一个“参数”，在指令名称之后以冒号表示。例如，`v-bind` 指令可以用于响应式地更新 HTML attribute：

```
<a v-bind:href="url">...</a>
```

在这里 `href` 是参数，告知 `v-bind` 指令将该元素的 `href` attribute 与表达式 `url` 的值绑定。

另一个例子是 `v-on` 指令，它用于监听 DOM 事件：

```
<a v-on:click="doSomething">...</a>
```

在这里参数是监听的事件名。我们也会更详细地讨论事件处理。

v-bind 可以通过{}、[]等绑定诸如style、class等内容

对象语法

```
用法一：直接通过{}绑定一个类
<h2 :class="{'active': isActive}">Hello World</h2>

用法二：也可以通过判断，传入多个值
<h2 :class="{'active': isActive, 'line': isLine}">Hello World</h2>

用法三：和普通的类同时存在，并不冲突
注：如果isActive和isLine都为true，那么会有title/active/line三个类
<h2 class="title" :class="{'active': isActive, 'line': isLine}">Hello
World</h2>

用法四：如果过于复杂，可以放在一个methods或者computed中
注：classes是一个计算属性
<h2 class="title" :class="classes">Hello World</h2>
```

数组语法

```
用法一：直接通过{}绑定一个类
<h2 :class="['active']">Hello World</h2>

用法二：也可以传入多个值
<h2 :class="['active', 'line']">Hello World</h2>

用法三：和普通的类同时存在，并不冲突
注：会有title/active/line三个类
<h2 class="title" :class="['active', 'line']">Hello World</h2>

用法四：如果过于复杂，可以放在一个methods或者computed中
注：classes是一个计算属性
<h2 class="title" :class="classes">Hello World</h2>
```

计算属性 computed

通过computed引入计算属性

```
<body>
  <div id='app'>
    <h2>{{fullname}}</h2>
  </div>

  <script src="../vue.js"></script>
  <script>
    const app = new Vue({
      el: '#app',
      data: {
        firstName: 'Kobe',
        lastName: 'Brant'
      },
      computed: {
        fullname: {
          set: function (newValue) {
            const names = newValue.split(' ');
            this.firstName = names[0];
            this.lastName = names[1];
          },
          get: function () {
            return this.firstName + ' ' + this.lastName;
          }
        }
      }
    })
  </script>
```

默认情况下set属性可以不要，即

```
<body>
  <div id='app'>
    <h2>总价：{{totalPrice}}</h2>
  </div>

  <script src="../vue.js"></script>
```

```
  <script>
    const app = new Vue({
      el: '#app',
      data: {
        firstName: 'Lebron',
        lastName: 'James',
        books: [{
            price: 111
          },
          {
            price: 222
          },
          {
            price: 333
          }
        ]
      },
      computed: {
        fullName: function () {
          return this.firstName + '' + this.lastName;
        },
        totalPrice: function () {
          let total = 0;
          for (i = 0; i < this.books.length; i++) {
            total += this.books[i].price;
          }
          return total;
        }
      }
    })
  </script>
</body>
```

计算属性的效率比方法要高

事件监听 v-on

v-on

可以给标签**绑定事件**

```
<body>
  <div id='app'>
    <h2>{{counter}}</h2>
    <button v-on:click='increment()'>+</button>
    <button v-on:click='decrement()'>-</button>
  </div>

  <script src="../vue.js"></script>
  <script>
    const app = new Vue({
      el: '#app',
      data: {
        message: 'Hello',
        counter: 0
```

```
      },
      methods: {
        increment() {
          this.counter++;
        },
        decrement() {
          this.counter--;
        }
      }
    })
  </script>
</body>
```

语法糖：使用@

```
<button @click> + </button>
```

v-on 参数传递问题

```
<body>

<div id="app">
  <!--1.事件调用的方法没有参数-->
  <button @click="btn1Click()">按钮1</button>
  <button @click="btn1Click">按钮1</button>

  <!--2.在事件定义时，写方法时省略了小括号，但是方法本身是需要一个参数的，这个时候，Vue会默认将浏览器生产的event事件对象作为参数传入到方法-->
  <!--<button @click="btn2Click(123)">按钮2</button>-->
  <!--<button @click="btn2Click()">按钮2</button>-->
  <button @click="btn2Click">按钮2</button>

  <!--3.方法定义时，我们需要event对象，同时又需要其他参数-->
  <!-- 在调用方式，如何手动的获取到浏览器参数的event对象：$event-->
  <button @click="btn3Click(abc, $event)">按钮3</button>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊',
      abc: 123
    },
    methods: {
      btn1Click() {
        console.log("btn1Click");
      },
      btn2Click(event) {
        console.log('--------', event);
      },
      btn3Click(abc, event) {
        console.log('++++++++', abc, event);
```

```
      }
    }
  })

  // 如果函数需要参数,但是没有传入, 那么函数的形参为undefined
  // function abc(name) {
  //   console.log(name);
  // }
  //
  // abc()
</script>

</body>
```

v-on 修饰符

```
<body>

<div id="app">
  <!--1. .stop修饰符的使用 阻止事件冒泡-->
  <div @click="divClick">
    aaaaaaa
    <button @click.stop="btnClick">按钮</button>
  </div>

  <!--2. .prevent修饰符的使用 阻止默认行为-->
  <br>
  <form action="baidu">
    <input type="submit" value="提交" @click.prevent="submitClick">
  </form>

  <!--3. .监听某个键盘的键帽-->
  <input type="text" @keyup.enter="keyUp">

  <!--4. .once修饰符的使用 只触发一次-->
  <button @click.once="btn2Click">按钮2</button>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊'
    },
    methods: {
      btnClick() {
        console.log("btnClick");
      },
      divClick() {
        console.log("divClick");
      },
      submitClick() {
        console.log('submitClick');
      },
```

```
      keyUp() {
        console.log('keyUp');
      },
      btn2Click() {
        console.log('btn2Click');
      }
    }
  })
</script>

</body>
```

条件判断 v-if

if-else

```
<body>

  <div id="app">
    <h2 v-if="isShow">
      <div>abc</div>
      <div>abc</div>
      <div>abc</div>
      <div>abc</div>
      <div>abc</div>
      {{message}}
    </h2>
    <h2 v-else>isShow 为 false 时显示我</h2>
  </div>

  <script src="../vue.js"></script>
  <script>
    const app = new Vue({
      el: '#app',
      data: {
        message: '你好啊',
        isShow: true
      }
    })
  </script>

</body>
```

else-if

```
<body>

  <div id="app">
    <h2 v-if='score>=90'>优秀</h2>
    <h2 v-else-if='score>=80'>良好</h2>
    <h2 v-else-if='score>=60'>及格</h2>
    <h2 v-else>不及格</h2>
  </div>

```

```
  <script src="../vue.js"></script>
  <script>
    const app = new Vue({
      el: '#app',
      data: {
        message: '你好啊',
        score: 59
      }
    })
  </script>

</body>
```

v-if 和 v-show的区别

```
<body>

<div id="app">
  <!--v-if: 当条件为false时, 包含v-if指令的元素, 根本就不会存在dom中-->
  <h2 v-if="isShow" id="aaa">{{message}}</h2>

  <!--v-show: 当条件为false时, v-show只是给我们的元素添加一个行内样式: display: none-->
  <h2 v-show="isShow" id="bbb">{{message}}</h2>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊',
      isShow: true
    }
  })
</script>

</body>
```

循环遍历 v-for

遍历数组

```
<body>

<div id="app">
  <!--1.在遍历的过程中,没有使用索引值(下标值)-->
  <ul>
    <li v-for="item in names">{{item}}</li>
  </ul>

  <!--2.在遍历的过程中, 获取索引值-->
  <ul>
    <li v-for="(item, index) in names">
```

```
      {{index+1}}.{{item}}
    </li>
  </ul>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      names: ['why', 'kobe', 'james', 'curry']
    }
  })
</script>

</body>
```

遍历对象

```
<body>

<div id="app">
  <!--1.在遍历对象的过程中, 如果只是获取一个值, 那么获取到的是value-->
  <ul>
    <li v-for="item in info">{{item}}</li>
  </ul>


  <!--2.获取key和value 格式: (value, key) -->
  <ul>
    <li v-for="(value, key) in info">{{value}}-{{key}}</li>
  </ul>


  <!--3.获取key和value和index 格式: (value, key, index) -->
  <ul>
    <li v-for="(value, key, index) in info">{{value}}-{{key}}-{{index}}</li>
  </ul>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      info: {
        name: 'why',
        age: 18,
        height: 1.88
      }
    }
  })
</script>

</body>
```

状态维护

官方推荐我们在使用v-for时，给对应的元素或组件添加上一个:key属性。

为什么需要这个key属性呢?

这个其实和Vue的虚拟DOM的Diff算法有关系。

这里我们借用React's diff algorithm中的一张图来简单说明一下:

当某一层有很多相同的节点时，也就是列表节点时，我们希望插入一个新的节点

我们希望可以在B和C之间加一个F，Diff算法默认执行起来是这样的。

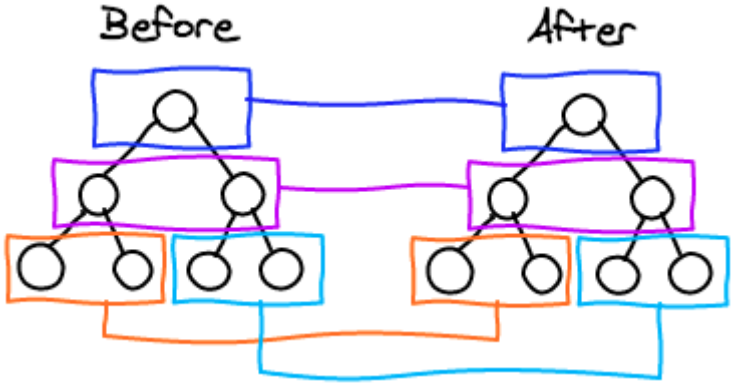
即把C更新成F，D更新成C，E更新成D，最后再插入E，是不是很没有效率?

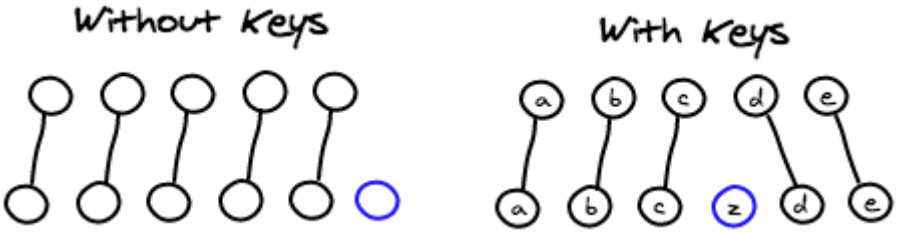
所以我们需要使用key来给每个节点做一个**唯一标识**（因此key必须是一个一一对应的值）

Diff算法就可以正确的识别此节点

找到正确的位置区插入新的节点。

所以一句话，**key**的作用主要是为了**高效的**更新虚拟DOM






```
<div id="app">
  <ul>
    <li v-for="item in letters" :key="item">{{item}}</li> //或者使用item.id
  </ul>
</div>
```

数组更新

有些操作数组方法不支持响应式

支持响应式的方法：push，pop，shift，unshift， splice，sort，reverse

注意：不要通过索引的方式修改数组，不支持响应式，用`array.splice(2, 1, 'aaa')`方式修改数组，或者用Vue自带的set方法修改

```
<body>
```

```
<div id="app">
  <ul>
    <li v-for="item in letters">{{item}}</li>
  </ul>
  <button @click="btnClick">按钮</button>
</div>

<script src="../js/vue.js"></script>
<script>
  const app = new Vue({
    el: '#app',
    data: {
      letters: ['a', 'b', 'c', 'd']
    },
    methods: {
      btnClick() {
        // 1.push方法
        // this.letters.push('aaa')
        // this.letters.push('aaaa', 'bbbb', 'cccc')

        // 2.pop(): 删除数组中的最后一个元素
        // this.letters.pop();

        // 3.shift(): 删除数组中的第一个元素
        // this.letters.shift();

        // 4.unshift(): 在数组最前面添加元素
        // this.letters.unshift()
        // this.letters.unshift('aaa', 'bbb', 'ccc')

        // 5.splice作用: 删除元素/插入元素/替换元素
        // 删除元素: 第二个参数传入你要删除几个元素(如果没有传,就删除后面所有的元素)
        // 替换元素: 第二个参数, 表示我们要替换几个元素, 后面是用于替换前面的元素
        // 插入元素: 第二个参数, 传入0, 并且后面跟上要插入的元素
        // splice(start)
        // splice(start):
        this.letters.splice(1, 3, 'm', 'n', 'l', 'x')
        // this.letters.splice(1, 0, 'x', 'y', 'z')

        // 5.sort()
        // this.letters.sort()

        // 6.reverse()
        // this.letters.reverse()

        // 注意: 通过索引值修改数组中的元素
        // this.letters[0] = 'bbbbbb';
        // this.letters.splice(0, 1, 'bbbbbb')
        // set(要修改的对象, 索引值, 修改后的值)
        Vue.set(this.letters, 0, 'bbbbbb')
      }
    }
  })


  // function sum(num1, num2) {
  //   return num1 + num2
```

```
  // }
  //
  // function sum(num1, num2, num3) {
  //   return num1 + num2 + num3
  // }
  // function sum(...num) {
  //   console.log(num);
  // }
  //
  // sum(20, 30, 40, 50, 601, 111, 122, 33)

</script>

</body>
```

表单输入绑定 v-model

双向数据绑定原理

v- model其实是一个语法糖，它的背后本质上是包含两个操作：

- v-bind 绑定一个 value 属性
- v-on 指令给当前元素绑定 input 事件，通过 methods 的方法修改参数

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

等同于

```
<input type="text" :value="message" @input="$event.target.value">
```

基本用法

v-model 结合 radio 使用

```
<body>
  <div id='app'>
    <label for="male">
      <input type="radio" id='male' name='sex' value='男' v-model='sex'>男
    </label>
    <label for="female">
      <input type="radio" id='female' name='sex' value='女' v-model='sex'>女
    </label>
    <h2>性别：{{sex}}</h2>
  </div>

  <script src="../vue.js"></script>
  <script>
    const app = new Vue({
      el: '#app',
      data: {
        message: 'Hello',
        sex: ''
      }
```

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

v-model 结合 checkbox 使用

单选

```
<body>
  <div id='app'>
    <label for="">
      <input type="checkbox" id="liscence" v-model='isAgree'>同意协议
    </label>
    <button :disabled="!isAgree">下一步</button>
    {{isAgree}}
  </div>

  <script src="../vue.js"></script>
  <script>
    const app = new Vue({
      el: '#app',
      data: {
        message: 'Hello',
        isAgree: false
      }
    })
  </script>
</body>
```

多选

```
<body>
  <div id='app'>
    <label for="">
      <input type="checkbox" value="篮球" v-model="hobbies">篮球
      <input type="checkbox" value="足球" v-model="hobbies">足球
      <input type="checkbox" value="羽毛球" v-model="hobbies">羽毛球
      <input type="checkbox" value="乒乓球" v-model="hobbies">乒乓球
    </label>
    {{hobbies}}
  </div>

  <script src="../vue.js"></script>
  <script>
    const app = new Vue({
      el: '#app',
      data: {
        message: 'Hello',
        isAgree: false,
        hobbies: []
      }
    })
  </script>
</body>
```

v-model 结合 select 使用

```
<body>
  <div id='app'>
    //单选
    <select name="abc" id="" v-model="fruit">
      <option value="苹果">苹果</option>
      <option value="香蕉">香蕉</option>
      <option value="榴莲">榴莲</option>
      <option value="葡萄">葡萄</option>
    </select>
    {{fruit}}
    //多选
    <select name="abc" id="" v-model="fruits" multiple>
      <option value="苹果">苹果</option>
      <option value="香蕉">香蕉</option>
      <option value="榴莲">榴莲</option>
      <option value="葡萄">葡萄</option>
    </select>
    {{fruits}}
  </div>

  <script src="../vue.js"></script>
  <script>
    const app = new Vue({
      el: '#app',
      data: {
        fruit: '苹果',
        fruits: []
      }
    })
  </script>
</body>
```

值绑定

需要用 v-for 从服务器动态获取值进行显示与绑定

```
<body>
  <div id='app'>
    <label v-for="item in hobbies" :for="item">
      <input type="checkbox" :value="item" v-model="hobbiesOut">{{item}}
    </label>
    {{hobbiesOut}}
  </div>

  <script src="../vue.js"></script>
  <script>
    const app = new Vue({
      el: '#app',
      data: {
        hobbies: ['篮球', '足球', '羽毛球'],
        hobbiesOut: []
      }
    })
  </script>
</body>
```

修饰符

- lazy: 默认情况下，v-model默认是在input事件中同步输入框的数据的。也就是说，一旦有数据发生改变对应的data中的数据就会自动发生改变。lazy修饰符可以让数据在失去焦点或者回车时才会更新
- number: 默认情况下，在输入框中无论我们输入的是字母还是数字，都会被当做字符串类型进行处理。但是如果我们希望处理的是数字类型，那么最好直接将内容当做数字处理。number修饰符可以让在输入框中输入的内容自动转成数字类型
- trim: 如果输入的内容首尾有很多空格，通常我们希望将其去除；trim修饰符可以过滤内容左右两边的空格

```
<body>
  <div id='app'>
    <input type="text" v-model.lazy="message">
    <h2>{{message}}</h2>

    <input type="text" v-model.number="age">
    <h2>{{age}} - {{typeof age}}</h2>

    <input type="text" v-model.trim='name'>
    <h2>{{name}}</h2>
  </div>

  <script src="../vue.js"></script>
  <script>
    const app = new Vue({
      el: '#app',
      data: {
        message: 'Hello',
        age: 18,
        name: 'zyy'
      }
    })
  </script>
</body>
```

组件

注册组件基本步骤

分为三步：创建组件构造器、注册组件、使用组件

```
<body>

<div id="app">
  <!--3.使用组件-->
  <my-cpn></my-cpn>
  <my-cpn></my-cpn>
  <my-cpn></my-cpn>
  <my-cpn></my-cpn>

  <div>
    <div>
```

```
      <my-cpn></my-cpn>
    </div>
  </div>
</div>

<my-cpn></my-cpn>

<script src="../js/vue.js"></script>
<script>
  // 1.创建组件构造器对象
  const cpnC = Vue.extend({
    template: `
      <div>
        <h2>我是标题</h2>
        <p>我是内容，哈哈哈哈</p>
        <p>我是内容，呵呵呵呵</p>
      </div>
    `
  })

  // 2.注册组件
  Vue.component('my-cpn', cpnC)

  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊'
    }
  })
</script>

</body>
```

步骤解析：

1. Vue.extend():
   - 调用Vue.extend()创建的是一个组件构造器
   - 通常在创建组件构造器时，传入template代表我们自定义组件的模板
   - 该模板就是在使用到组件的地方，要显示的HTML代码
   - 事实上，这种写法在Vue2.x的文档中几乎已经看不到了，它会直接使用下面我们会讲到的语法糖，但是在很多资料还是会提到这种方式，而且这种方式是学习后面方式的基础
2. Vue.component():
   - 调用Vue.component()是将刚才的组件构造器注册为一个组件，并且给它起一个组件的标签名称
   - p所以需要传递两个参数：1、注册组件的标签名 2、组件构造器
3. 组件必须挂载在某个Vue实例下，否则它不会生效

全局组件与局部组件

上述创建方式为全局组件，可以在多个Vue实例中使用，但局部组件只能在一个Vue实例中使用

局部组件的创建方法为：

```
<body>

  <div id='app'>
```

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

  <div id='app2'>
    <cpn></cpn>
  </div>

  <script src="../vue.js"></script>
  <script>
    const cpnC = Vue.extend({
      template: `
      <div>
        <h2>我是标题</h2>
        <p>我是内容</p>
      </div>
      `
    })
    // 局部组件在创建Vue实例时通过components引入
    const app = new Vue({
      el: '#app',
      data: {
        message: 'Hello',
      },
      components: {
        cpn: cpnC //cpn为使用组件时的标签名
      }
    })

    const app2 = new Vue({
      el: '#app2'
    })
  </script>

</body>
```

组件中数据存放

组件中不可以直接使用 Vue 实例中的数据，需要在注册组件时，通过`data(){}`函数，return一个对象来进行数据传递

```
<template id="cpn">
    <div>
      <h2>{{message}}</h2>
      <p>我是内容,呵呵呵</p>
    </div>
</template>
<script>
Vue.component('cpn', {
      template: '#cpn',
      data() {
        return {
          message: '这里存放组件的数据'
        }
      }
    })
</script>
```

父子组件

基本使用

示例：

cpn2 为父组件， cpn1 为子组件

- 首先创建两个组件实例
- 在父组件cpn2中通过components注册子组件cpn1
- 在Vue实例中注册父组件cpn2

```
<body>

  <div id='app'>
    <cpn2></cpn2>
  </div>

  <script src="../vue.js"></script>
  <script>
    const cpnC1 = Vue.extend({
      template: `
      <div>
        <h2>我是标题1</h2>
        <p>我是内容1111111111</p>
      </div>
      `
    })

    const cpnC2 = Vue.extend({
      template: `
      <div>
        <h2>我是标题2</h2>
        <p>我是内容22222222</p>
        <cpn1></cpn1>
      `,
      components: {
        cpn1: cpnC1
      }
    })

    const app = new Vue({
      el: '#app',
      data: {
        message: 'Hello',
      },
      components: {
        cpn2: cpnC2,
      }

    })
  </script>

</body>
```

语法糖：

省去了调用Vue.extend()的步骤，而是可以直接使用一个对象来代替

```
<body>

<div id="app">
  <cpn1></cpn1>
  <cpn2></cpn2>
</div>

<script src="../js/vue.js"></script>
<script>
  // 1.全局组件注册的语法糖
  // 1.创建组件构造器
  // const cpn1 = Vue.extend()

  // 2.注册组件
  Vue.component('cpn1', {
    template: `
      <div>
        <h2>我是标题1</h2>
        <p>我是内容，哈哈哈哈</p>
      </div>
    `
  })

  // 2.注册局部组件的语法糖
  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊'
    },
    components: {
      'cpn2': {
        template: `
          <div>
            <h2>我是标题2</h2>
            <p>我是内容，呵呵呵</p>
          </div>
    `
      }
    }
  })
</script>

</body>
```

模板抽离写法

```
<body>

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

```
<!--1.script标签, 注意:类型必须是text/x-template-->
<!--<script type="text/x-template" id="cpn">-->
<!--<div>-->
  <!--<h2>我是标题</h2>-->
  <!--<p>我是内容,哈哈哈</p>-->
<!--</div>-->
<!--</script>-->

<!--2.template标签-->
<template id="cpn">
  <div>
    <h2>我是标题</h2>
    <p>我是内容,呵呵呵</p>
  </div>
</template>

<script src="../js/vue.js"></script>
<script>

  // 1.注册一个全局组件
  Vue.component('cpn', {
    template: '#cpn'
  })

  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊'
    }
  })
</script>

</body>
```

父子组件通信

由于每一个组件相当于一个封闭的空间，因此需要参数传递功能，才能将参数在各个组件之间传递

父=>子 Props

父组件通过 Props 属性向子组件传递参数，下面以 Vue 实例为父组件 cpn 为子组件演示参数传递过程

- 在子组件中通过 props 属性创建数据
  - 可以通过数组`props: ['cmovies', 'cmessage']`
  - 可以通过对象，提供默认值
- 在使用组件时通过属性将子组件中的数据和父组件中数据关联`<cpn :cmessage="message" :cmovies="movies"></cpn>`

```
<body>

<div id="app">
  <!--<cpn v-bind:cmovies="movies"></cpn>-->
  <!--<cpn cmovies="movies" cmessage="message"></cpn>-->

  <cpn :cmessage="message" :cmovies="movies"></cpn>
```

```
</div>

<template id="cpn">
  <div>
    <ul>
      <li v-for="item in cmovies">{{item}}</li>
    </ul>
    <h2>{{cmessage}}</h2>
  </div>
</template>

<script src="../js/vue.js"></script>
<script>
  // 父传子: props
  const cpn = {
    template: '#cpn',
    // props: ['cmovies', 'cmessage'],
    props: {
      // 1.类型限制
      // cmovies: Array,
      // cmessage: String,

      // 2.提供一些默认值, 以及必传值
      cmessage: {
        type: String,
        default: 'aaaaaaaa',
        required: true
      },
      // 类型是对象或者数组时, 默认值必须是一个函数
      cmovies: {
        type: Array,
        default() {
          return []
        }
      }
    },
    data() {
      return {}
    },
    methods: {

    }
  }

  const app = new Vue({
    el: '#app',
    data: {
      message: '你好啊',
      movies: ['海王', '海贼王', '海尔兄弟']
    },
    components: {
      cpn
    }
  })
</script>

</body>
```

注意 props 的驼峰命名

如果在 props 里使用了驼峰命名

```
const cpn = {
   template: '#cpn',
   props: {
     childMyMessage: {
       type: String,
       default () {
         return 'a'
       }
     }
   }
}
```

那么在使用组件时，属性要使用`-`连接 `<cpn :child-my-message="message1"></cpn>`

子=>父 自定义事件

通过自定义事件实现子组件向父组件传递数据

- 首先在子组件中定义btnClick事件，获取item数据
- 在btnClick事件中通过`this.$emit('item-click', item)`通过自定义`item-click`事件传递 item 数据
- 在父组件中使用子组件时，监听`item-click`事件，并在负组件中注册`cpnClick`事件进行数据接收，往常未传入实参时，会默认接收到点击事件，但是此处默认接收到 emit 的参数，即 item

```
<body>

  <div id="app">
    <cpn @item-click="cpnClick"></cpn>
  </div>

  <template id="cpn">
    <div>
      <button v-for="item in categories" @click="btnClick(item)">
{{item.name}}</button>
    </div>
  </template>

  <script src="../vue.js"></script>
  <script>
    const cpn = {
      template: "#cpn",
      data() {
        return {
          categories: [{
              id: 'aaa',
              name: '热门推荐'
            },
            {
              id: 'bbb',
              name: '手机数码'
            },
          ]
        }
```

```
      },
      methods: {
        btnClick(item) {
          this.$emit('item-click', item);
        }
      },
    }

    const app = new Vue({
      el: '#app',
      data: {
        message: 'hello'
      },
      components: {
        cpn
      },
      methods: {
        cpnClick(item) {
          console.log(item)
        }
      },
    })
  </script>

</body>
```

父子组件访问

父子组件互相访问对方

ref如果绑定在组件中，那么通过`this.$ref.refname`获取到的是一个组件对象

ref如果绑定在元素中，那么通过`this.$ref.refname`获取到的是一个元素对象

父访问子 $children $refs

方法一：使用 $children

如果一个子组件使用了 n 次，就会产生 n 个children， 因此在使用`this.$children[0].name`时要指定具体访问哪个，但是一旦后续增加、减少了子组件，就会影响下标，因此不常用

```
<body>

  <div id="app">
    <cpn></cpn>
    <cpn></cpn>
    <cpn></cpn>
    <button @click="showChildren">点击</button>
  </div>

  <template id="cpn">
    <div>
      {{message}}
    </div>
  </template>
```

```
  <script src="../vue.js"></script>
  <script>
    const app = new Vue({
      el: '#app',
      data: {

      },
      components: {
        cpn: {
          template: '#cpn',
          data() {
            return {
              message: '我是子组件',
              name: 'zyy'
            }
          },
          methods: {
            showName() {
              console.log(this.name)
            }
          },
        }
      },
      methods: {
        showChildren() {
          console.log(this.$children[0]);
        }
      },
    })
  </script>

</body>
```

方法二：使用 $refs

需要提前在使用子组件时，添加 ref 属性，只有有了 ref 属性，才可以被`this.$refs`接收到

```
<div id="app">
    <cpn ref='aaa'></cpn>
    <cpn ref="bbb"></cpn>
    <cpn></cpn>
    <button @click="showChildren">点击</button>
</div>
```

```
showChildren() {
  console.log(this.$refs.aaa.name);
}
```

子访问父 $parent $root

```
<body>

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

```
6
7   <template id="cpn">
8     <div>
9       <h2>我是cpn组件</h2>
10      <ccpn></ccpn>
11    </div>
12  </template>
13
14  <template id="ccpn">
15    <div>
16      <h2>我是子组件</h2>
17      <button @click="btnClick">按钮</button>
18    </div>
19  </template>
20
21  <script src="../js/vue.js"></script>
22  <script>
23    const app = new Vue({
24      el: '#app',
25      data: {
26        message: '你好啊'
27      },
28      components: {
29        cpn: {
30          template: '#cpn',
31          data() {
32            return {
33              name: '我是cpn组件的name'
34            }
35          },
36          components: {
37            ccpn: {
38              template: '#ccpn',
39              methods: {
40                btnClick() {
41                  // 1.访问父组件$parent
42                  console.log(this.$parent);
43                  console.log(this.$parent.name);
44
45                  // 2.访问根组件$root
46                  // console.log(this.$root);
47                  // console.log(this.$root.message);
48                }
49              }
50            }
51          }
52        }
53      }
54    })
55  </script>
56
57  </body>
```

插槽

可以对组件的功能进行扩展

基本使用

`<slot></slot>`即为插槽，里面可以放默认值

- 如果有默认值，使用组件时显示默认值
- 如果使用组件时传入了其他值，则默认值会被覆盖
- 可以传入多个值

```
<body>

  <div id="app">
    <cpn></cpn>
    <cpn><span>123</span></cpn>
    <cpn>
      <span>123</span>
      <button>按钮</button>
    </cpn>
  </div>

  <template id="cpn">
    <div>
      <h2>我是组件</h2>
      <slot><button>按钮</button></slot>
    </div>
  </template>

  <script src="../vue.js"></script>
  <script>
    const app = new Vue({
      el: "#app",
      data: {
        message: '你好啊'
      },
      components: {
        cpn: {
          template: '#cpn'
        }
      }
    })
  </script>

</body>
```

具名插槽

```
<template id="cpn">
   <div>
     <slot name="left"><span>左边</span></slot>
     <slot name="middle"><span>中间</span></slot>
     <slot name="right"><span>右边</span></slot>
   </div>
</template>
```

使用

```
<div id="app">
  <cpn>
    <span slot="left">修改左边</span>
  </cpn>
  <cpn>
    <button slot="middle">中间</button>
  </cpn>
</div>
```

作用域插槽

父组件想要替换插槽里的标签，但是内容、数据由子组件提供

由于作用域的限制，父组件无法直接访问子组件的数据，需要借助作用域插槽

- 首先在插槽 slot 中添加属性 `:data='pLanguages'` data名字可以随意指定，暴露出pLanguages数据
- 在使用子组件时，添加 template 属性 `slot-scope='slot'` 即可在子组件中使用 `slot.data` 数据

```
<body>

  <div id="app">
    <cpn>
      <template slot-scope="slot">
        <span v-for="item in slot.data">{{item}} - </span>
      </template>
    </cpn>
  </div>

  <template id="cpn">
    <div>
      <slot :data="pLanguages">
        <ul>
          <li v-for="item in pLanguages">{{item}}</li>
        </ul>
      </slot>
    </div>
  </template>

  <script src="../vue.js"></script>
  <script>
    const app = new Vue({
      el: "#app",
      data: {
        message: '你好啊'
      },
      components: {
        cpn: {
          template: '#cpn',
          data() {
            return {
              pLanguages: ['JavaScript', 'Python', 'Swift', 'C++', 'C#',
'JAVA']
            }
          }
        }
      }
```

```
    })
  </script>

</body>
```

模块化导入导出

导出

```
// 1.导出方式一:
export {
  flag, sum
}

// 2.导出方式二:
export var num1 = 1000;
export var height = 1.88


// 3.导出函数/类
export function mul(num1, num2) {
  return num1 * num2
}

export class Person {
  run() {
    console.log('在奔跑');
  }
}
```

导入

```
// 1.导入的{}中定义的变量
import {flag, sum} from "./aaa.js";

if (flag) {
  console.log('小明是天才, 哈哈哈');
  console.log(sum(20, 30));
}

// 2.直接导入export定义的变量
import {num1, height} from "./aaa.js";

console.log(num1);
console.log(height);

// 3.导入 export的function/class
import {mul, Person} from "./aaa.js";
```

default导出

每个文件只能使用一次default导出，导出的对象可以被导入者**自定义名字**

```
//导出
export default function (argument) {
  console.log(argument);
}
// 导入 export default中的内容
import addr from "./aaa.js";
```

全部导入

```
// 统一全部导入
// import {flag, num, num1, height, Person, mul, sum} from "./aaa.js";
import * as aaa from './aaa.js'

console.log(aaa.flag);
console.log(aaa.height);
```

webpack

现在的js文件中使用了模块化的方式进行开发，他们可以直接使用吗？不可以

因为如果直接在`index.html`引入这两个js文件，浏览器并不识别其中的模块化代码

另外，在真实项目中当有许多这样的js文件时，我们一个个引用非常麻烦，并且后期非常不方便对它们进行管理

因此，需要使用webpack工具，对多个js文件进行打包

安装 `npm install webpack@3.6.0 -g`

打包 `webpack src/main.js dist/bundle.js` 前一部分是需要打包的js文件（入口），后一部分是打包后的js文件（出口），html文件引用打包后的js文件即可

配置

如果每次使用webpack的命令都需要写上入口和出口作为参数，就非常麻烦，有没有一种方法可以将这两个参数写到配置中，在运行时，直接读取呢？

创建一个`webpack.config.js`文件

```
const path = require('path')

module.exports = {
  // 入口：可以是字符串/数组/对象，这里我们入口只有一个,所以写一个字符串即可
  entry: './src/main.js',
  // 出口：通常是一个对象，里面至少包含两个重要属性，path 和 filename
  output: {
    path: path.resolve(__dirname, 'dist'), // 注意：path通常是一个绝对路径
    filename: 'bundle.js'
  }
}
```

这样一来在终端里直接输入`webpack`即可进行打包

局部安装webpack

一个项目往往依赖特定的webpack版本，全局的版本可能很这个项目的webpack版本不一致，导出打包出现问题，所以需要局部安装webpack

- 在文件夹路径下，执行`npm install webpack@3.6.0 --save-dev`
- 通过`node_modules/.bin/webpack`启动 webpack 打包

package.json中定义启动

每次执行都敲这么一长串有没有觉得不方便呢？我们可以在`package.json`的scripts中定义自己的执行脚本

```
{
  "name": "meetwebpack",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack"
  },
  "author": "",
  "license": "ISC"
}
```

之后，即可使用`npm run build`执行 webpack 命令

- 首先，会寻找本地的node_modules/.bin路径中对应的命令
- 如果没有找到，会去全局的环境变量中寻找

Loader

在开发中我们不仅仅有基本的js代码处理，我们也需要加载css、图片，也包括一些高级的将ES6转成ES5代码，将TypeScript转成ES5代码，将scss、less转成css，将.jsx、.vue文件转成js文件等等，对于webpack本身的能力来说，对于这些转化是不支持的，给webpack扩展对应的loader就可以了

使用过程：

- 通过npm安装需要使用的loader
- 在webpack.config.js中的modules关键字下进行配置

CSS文件处理

首先在 src 文件夹下创建 css 文件。之后在入口文件 main.js 中通过 require 引用

通过 npm 命令安装 loader

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

在webpack.config.js文件中进行配置

```
const path = require('path')

module.exports = {
  entry: './src/main.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js'
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        // css-loader只负责将css文件进行加载
        // style-loader负责将样式添加到DOM中
        // 使用多个loader时，是从右向左
        use: [ 'style-loader', 'css-loader' ]
      }
    ]
  }
}
```

less文件处理

- 安装loader `npm install --save-dev less-loader less`
- 配置

```
const path = require('path')

module.exports = {
  entry: './src/main.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'bundle.js'
  },
  module: {
    rules: [
      {
        test: /\.css$/,
        use: [ 'style-loader', 'css-loader' ]
      },
      {
        test: /\.less$/,
        use: [{
            loader: "style-loader" // creates style nodes from JS strings
        }, {
            loader: "css-loader" // translates CSS into CommonJS
        }, {
            loader: "less-loader" // compiles Less to CSS
```

```
            }]
        }
    ]
  }
}
```

图片文件处理

url-loader

```
body {
  background-color: red;
  background: url(../imgs/test01.jpeg)
}
```

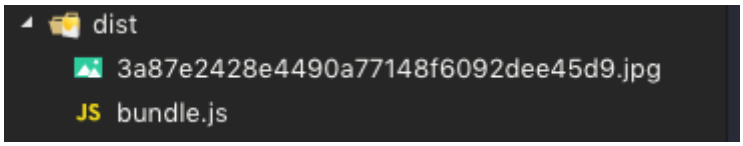
- 使用 url 引用图片
- `npm install --save-dev url-loader` 安装
- 配置 webpack.config.js 文件

```
{
        test: /\.(png|jpg|gif)$/,
        use: [
          {
            loader: 'url-loader',
            options: {
              limit: 8192
            }
          }
        ]
}
```

注意此处的 limit 意为大小小于8192的图片可以用 url-loader 转化成 base64 字符串进行显示，大于 limit 的图片将通过 file-loader 进行打包

file-loader

- `npm install --save-dev file-loader` 安装，无需配置
- 打包后 dist 文件夹下多以一个图片文件



- 我们可以在options中添加上如下选项：
  - img：文件要打包到的文件夹
  - name：获取图片原来的名字，放在该位置
  - hash:8：为了防止图片名称冲突，依然使用hash，但是我们只保留8位
  - ext：使用图片原来的扩展名

```
options: {
    limit: 8192,
    name: 'img/[name].[hash:8].[ext]'
}
```

- 我们发现图片并没有显示出来，这是因为图片使用的路径不正确.默认情况下，webpack会将生成的路径直接返回给使用者。但是，我们整个程序是打包在dist文件夹下的，所以这里我们需要在路径下再添加一个dist/

ES6转ES5

使用babel对应的loader即可将ES6转成ES5

- `npm install --save-dev babel-loader@7 babel-core babel-preset-es2015`
- 配置webpack.config.js文件

```
rules: [
    {
      test: /\.js$/,
      exclude: /(node_modules|bower_components)/,
      use: {
        loader: 'babel-loader',
        options: {
          presets: ['@babel/preset-env']
        }
      }
    }
  ]
```

- 重新打包，查看bundle.js文件

配置Vue

我们希望在项目中使用Vuejs，那么必然需要对其有依赖，所以需要先进行安装

- `npm install vue --save`，之后就可以按照之前学习的方式来使用Vue了
- 重新打包，运行程序，但是浏览器中会报错，这个错误说的是我们使用的是runtime-only版本的Vue
- 修改webpack的配置，添加如下内容，即可正常使用

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

组件抽离

首先要搞明白el和template的区别

```
new Vue({
  el: '#app',
  template: '<div id="app">{{message}}</div>',
  data: {
    message: 'coderwhy'
  }
})
```

如果Vue实例中同时指定了el和template，那么template模板的内容会替换掉挂载的对应el的模板，这样做之后我们就不需要在以后的开发中再次操作index.html，只需要在template中写入对应的标签即可

但是书写template模板非常麻烦，因此我们需要将template进行抽离

- 首先想到的，是将整个组件抽离到app.js中，之后在main.js中通过`import App from './vue/app.js'`使用App组件，但是在这种文件结构不太好，且没法编写样式

```
export default {
  template: `
  <div>
    <h2>{{message}}</h2>
    <button @click="btnClick">按钮</button>
    <h2>{{name}}</h2>
  </div>
  `,
  data() {
    return {
      message: 'Hello Webpack',
      name: 'coderwhy'
    }
  },
  methods: {
    btnClick() {

    }
  }
}
```

- 更好的方法是用.vue文件进行抽离，但是不能直接使用，需要`npm install vue-loader vue-template-compiler --save-dev`安装依赖，并修改webpack.config.js的配置文件，之后就可以使用了

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

```
<template>
  <div>
    <h2 class="title">{{message}}</h2>
    <button @click="btnClick">按钮</button>
    <h2>{{name}}</h2>
    <Cpn/>
  </div>
```

```
</template>

<script>
  import Cpn from './Cpn.vue'

  export default {
    name: "App",
    components: {
      Cpn
    },
    data() {
      return {
        message: 'Hello Webpack',
        name: 'coderwhy'
      }
    },
    methods: {
      btnClick() {

      }
    }
  }
</script>

<style scoped>
  .title {
    color: green;
  }
</style>
```

通过import导入即可使用该组件，且文件结构和逻辑都很清晰

Plugin

作用相当于插件

横幅BannerPlugin

```
const webpack = require('webpack')

module.exports = {
    plugins: [
        new webpack.BannerPlugin('最终版权由candy所有')
    ]
}
```

重新打包程序，查看bundle.js文件头部，可以看到版权信息

打包html的plugin

在真实发布项目时，发布的是dist文件夹中的内容，但是dist文件夹中如果没有index.html文件，那么打包的js等文件也就没有意义了。所以，我们需要将index.html文件打包到dist文件夹中，这个时候就可以使用HtmlWebpackPlugin插件

HtmlWebpackPlugin插件可以为我们做这些事情：自动生成一个index.html文件(可以指定模板来生成)；将打包的js文件，自动通过script标签插入到body中

- `npm install html-webpack-plugin --save-dev`安装HtmlWebpackPlugin插件
- 使用插件，修改webpack.config.js文件中plugins部分的内容如下：

```
plugins: [
  new webpack.BannerPlugin('最终版权归coderwhy所有'),
  new htmlWebpackPlugin({
    template: 'index.html'
  }),
]
```

  - 这里的template表示根据什么模板来生成index.html
  - 另外，我们需要删除之前在output中添加的publicPath属性
  - 否则插入的script标签中的src可能会有问题

js压缩的plugin

项目发布前，要对js等文件进行压缩处理

- `npm install uglifyjs-webpack-plugin@1.1.1 --save-dev`
- 修改webpack.config.js文件，使用插件：

```
const path = require('path')
const webpack = require('webpack')
const HtmlWebpackPlugin = require('html-webpack-plugin')
const UglifyjsWebpackPlugin = require('uglifyjs-webpack-plugin')

module.exports = {
  ...
  module: {
    plugins: [
        new webpack.BannerPlugin('最终版权归candy所有'),
        new HtmlWebpackPlugin({
        template: 'index.html'
        }),
        new UglifyjsWebpackPlugin()
    ],
}
```

搭建本地服务器

webpack提供了一个可选的本地开发服务器，这个本地服务器基于node.js搭建，内部使用express框架，可以实现我们想要的让浏览器自动刷新显示我们修改后的结果。

- 安装`npm install --save-dev webpack-dev-server@2.9.1`
- devserver也是作为webpack中的一个选项，选项本身可以设置如下属性：
  - contentBase：为哪一个文件夹提供本地服务，默认是根文件夹，我们这里要填写./dist
  - port：端口号
  - inline：页面实时刷新
  - historyApiFallback：在SPA页面中，依赖HTML5的history模式

```
module.exports = {
  ...
  devServer: {
    contentBase: './dist',
    inline: true
  }
}
```

- 在package.json文件中配置scripts，加一个dev，之后就可以用`npm run dev`使用，`--open`表示自动打开

```
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack",
    "dev": "webpack-dev-server --open"
  },
```

脚手架

安装`npm install -g @vue/cli`

上面安装的是Vue CLI3的版本，如果需要想按照Vue CLI2的方式初始化项目时不可以的。

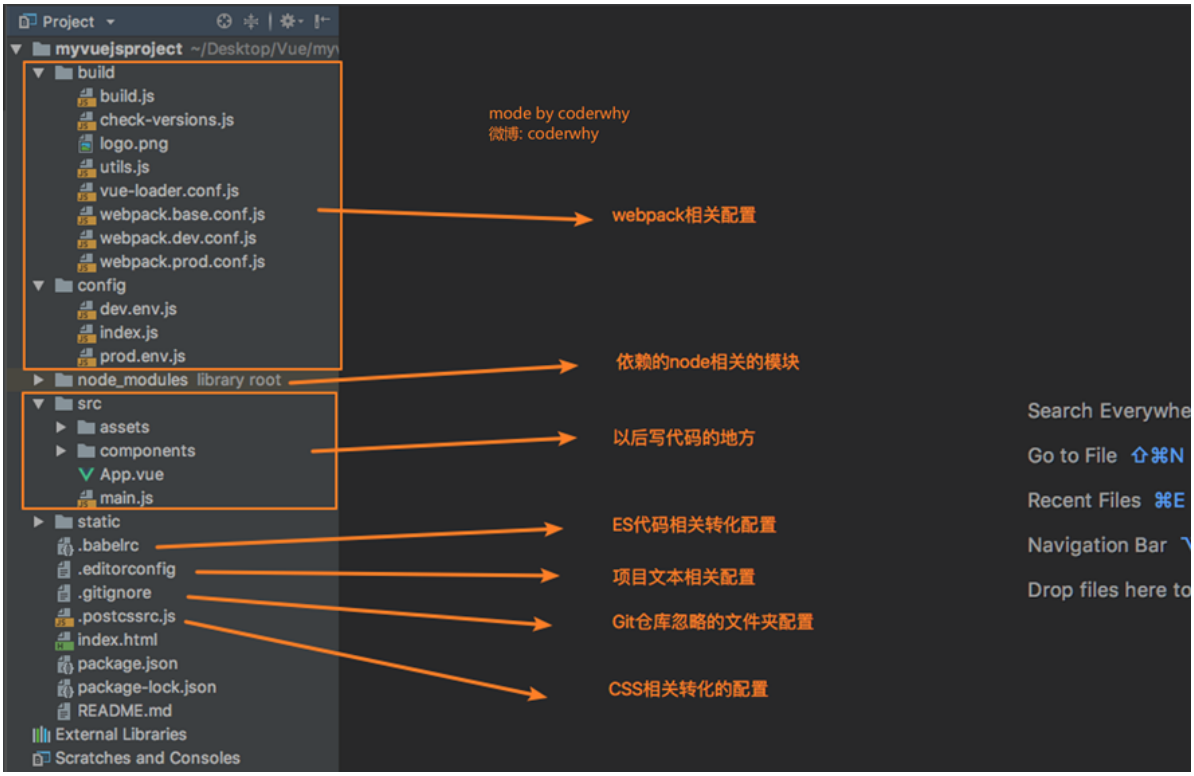
有时需要用CLI2，因此也需要拉取CLI2模板`npm install -g @vue/cli-init`

Vue CLI2初始化项目`vue init webpack my-project`

Vue CLI3初始化项目`vue create my-project`

CLI2

目录结构

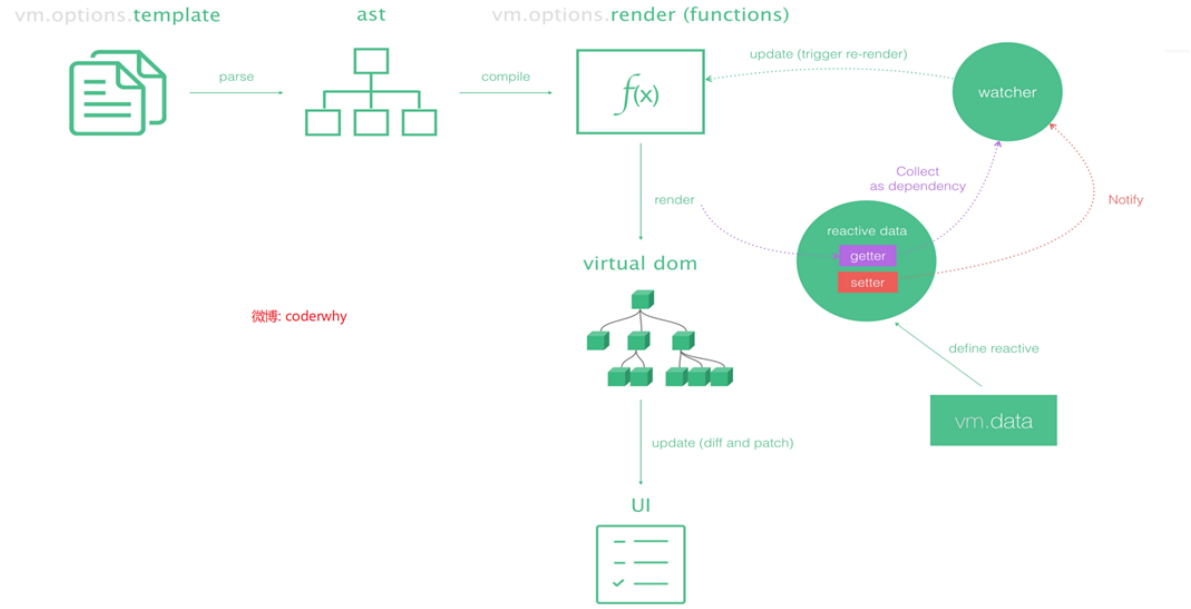


Runtime-compiler 和 Runtime-only 的区别

```
new Vue({
  el: '#app',
  components: { App },
  template: '<App/>'
})
```

```
new Vue({
  el: '#app',
  render: h => h(App)
})
```

Vue程序运行过程：



runtime-compiler编译过程：template -> ast -> render -> vdom -> UI

runtime-only编译过程：render -> vdom -> UI

对于每一个.vue文件（即组件）,Vue在编译的时候都会将template编译好，直接输出一个对象，对象里包含render方法，因此我们没有必要每次都通过template -> ast -> render步骤，所以使用runtime-only可以节省空间

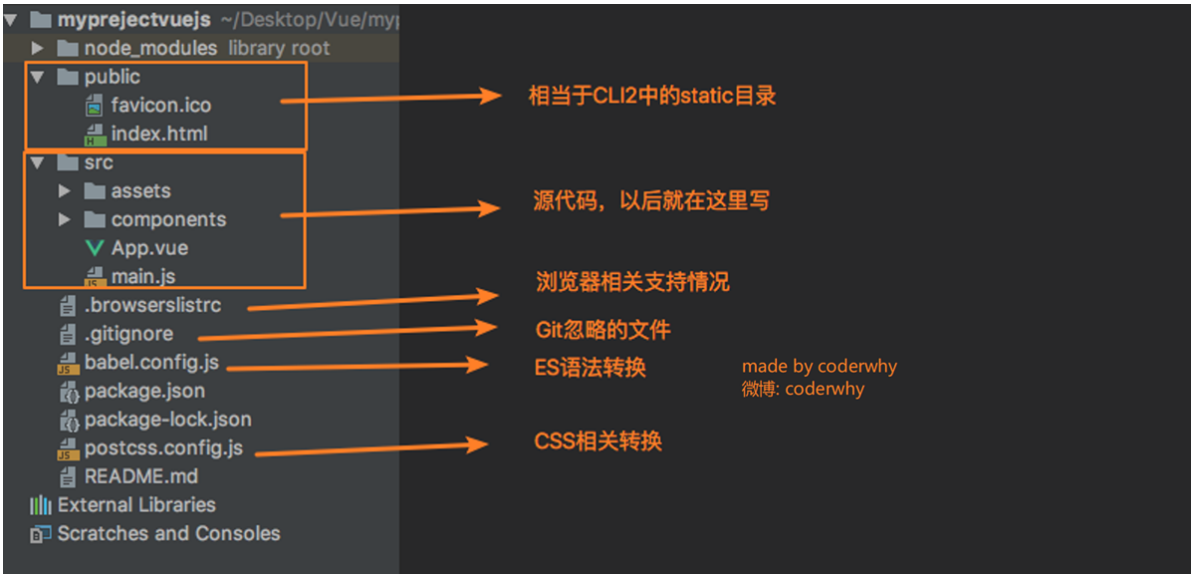
render函数的使用方法：

```
new Vue({
  el: '#app',
  render: (createElement) => {
    // 1.使用方式一:
    return createElement('标签', '相关数据对象(可以不传)', ['内容数组'])
    // 1.1.render函数基本使用
    return createElement('div', {class: 'box'}, ['coderwhy'])
    // 1.2.嵌套render函数
    return createElement('div', {class: 'box'}, ['coderwhy', createElement('h2', ['标题啊'])])
  }
})
```

```
new Vue({
  el: '#app',
  render: (createElement) => {
    // 2.使用方式二: 传入一个组件对象
    return createElement(cpn)
  }
})
```

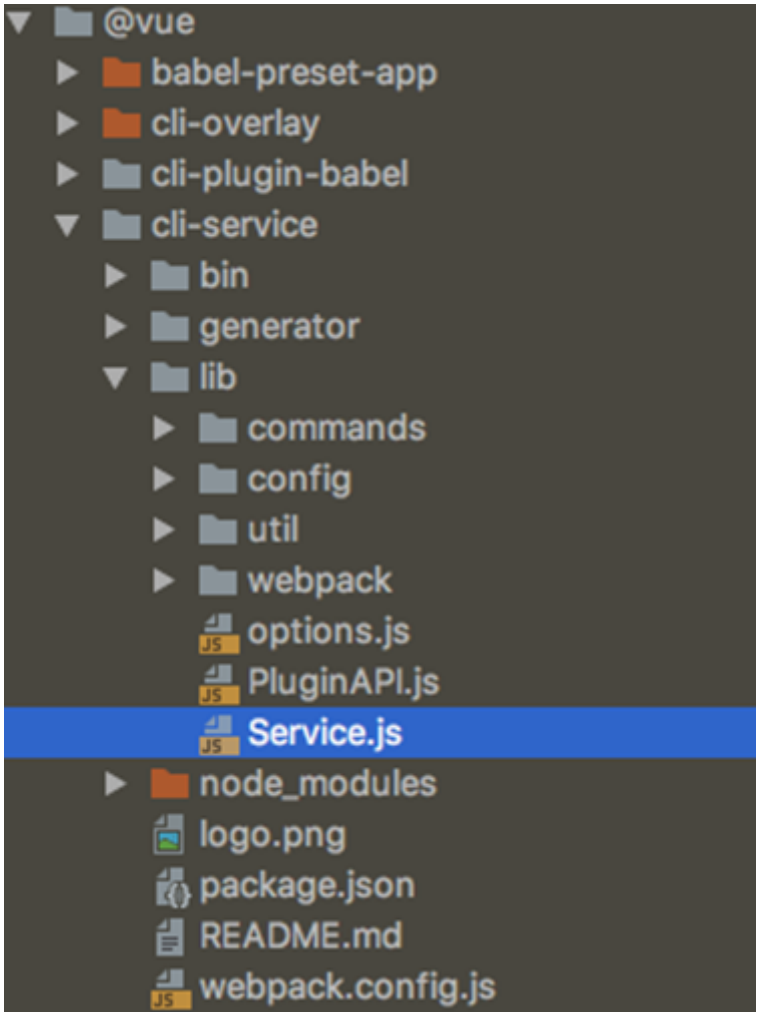
CLI3

目录结构



配置文件查看与修改

- 可以通过执行`vue ui`命令，通过ui的方式进行查看与修改
- CLI2的webpack配置被隐藏在了node_modules文件夹里



- 如果有配置文件需要自己修改，需要创建一个`vue.config.js`文件，名字必须是这个，在其中自定义配置

Vue-router

前端路由是通过一次请求资源，返回所有数据，通过js代码决定显示什么，由前端配置url，核心是改变URL，但是页面不进行整体的刷新

前端路由的规则

URL的hash

- URL的hash也就是锚点(#), 本质上是改变window.location的href属性.
- 我们可以通过直接赋值location.hash来改变href, 但是页面不发生刷新

```
> location.href
"http://192.168.1.101:8000/examples/urlChange/"
> location.hash = '/'
"/"
> location.href
"http://192.168.1.101:8000/examples/urlChange/#/"
> location.hash = '/foo'
"/foo"
> location.href
"http://192.168.1.101:8000/examples/urlChange/#/foo"
>
```


history

- history接口是HTML5新增的, 它有五种模式改变URL而不刷新页面.
- history.pushState()

```
> location.href
"http://192.168.1.101:8000/examples/urlChange/"
> history.pushState({}, '', '/foo')
undefined
> location.href
"http://192.168.1.101:8000/foo"
> history.pushState({}, '', '/')
undefined
> location.href
"http://192.168.1.101:8000/"
>
```


- history.replaceState()

```
> location.href
"http://192.168.1.101:8000/"
> history.replaceState({}, '', '/foo')
undefined
> location.href
"http://192.168.1.101:8000/foo"
> history.replaceState({}, '', '/foo/bar')
undefined
> location.href
"http://192.168.1.101:8000/foo/bar"
>
```


- history.go()

```
> location.href
"http://192.168.1.101:8000/"
> history.go(-1)
undefined
> location.href
"http://192.168.1.101:8000/foo"
> history.go(-1)
undefined
> location.href
"http://192.168.1.101:8000/examples/urlChange/"
> history.go(1)
undefined
> location.href
"http://192.168.1.101:8000/foo"
>
```


基础

安装和配置

- `npm install vue-router --save`
- 创建router文件夹，在其中创建index.js文件，编写如下内容，在routers里配置路由映射关系

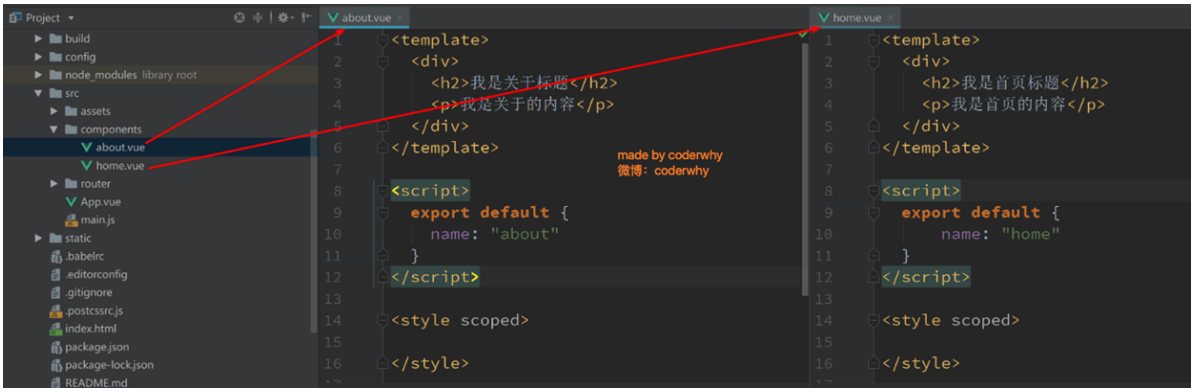
```
import Vue from 'vue'
import VueRouter from 'vue-router'

// 1.通过Vue.use(插件)，安装插件
Vue.use(VueRouter);

// 2.创建VueRouter对象
const routes = [

]

const router = new VueRouter({
  routes
})

// 3.将Router对象传入到Vue实例中
export default router

```

- 在main.js中进行挂载

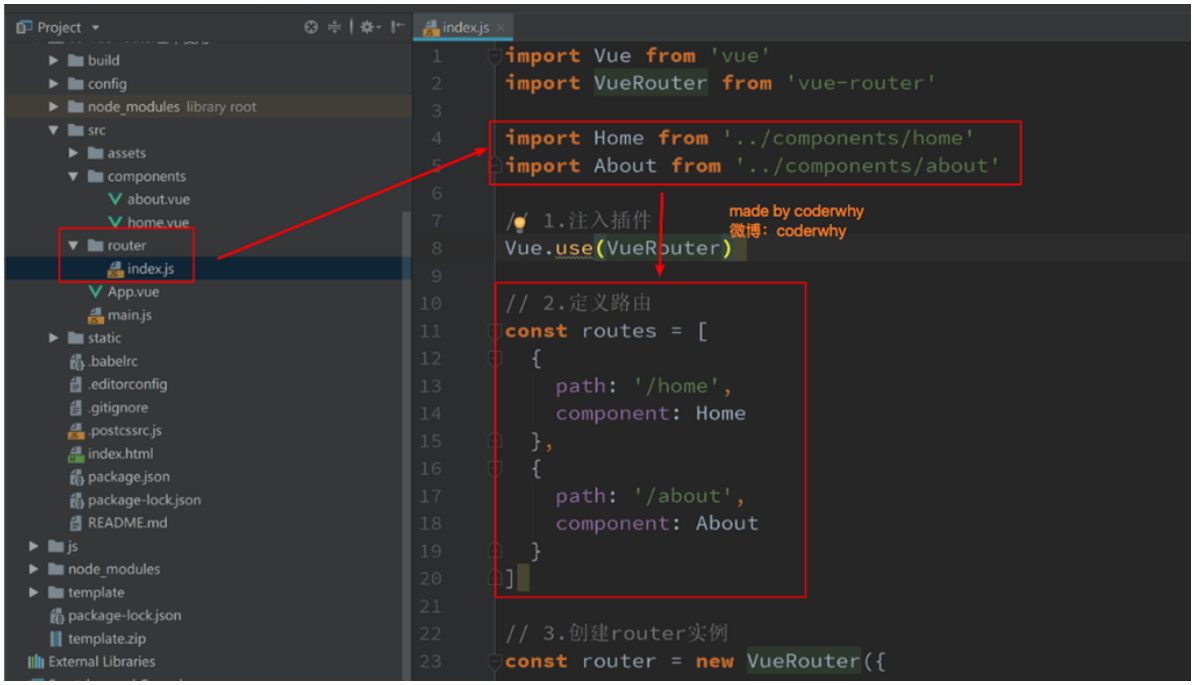
```
import Vue from 'vue'
import App from './App'
import router from './router'

Vue.config.productionTip = false

/* eslint-disable no-new */
new Vue({
  el: '#app',
  router,
  render: h => h(App)
})
```

使用

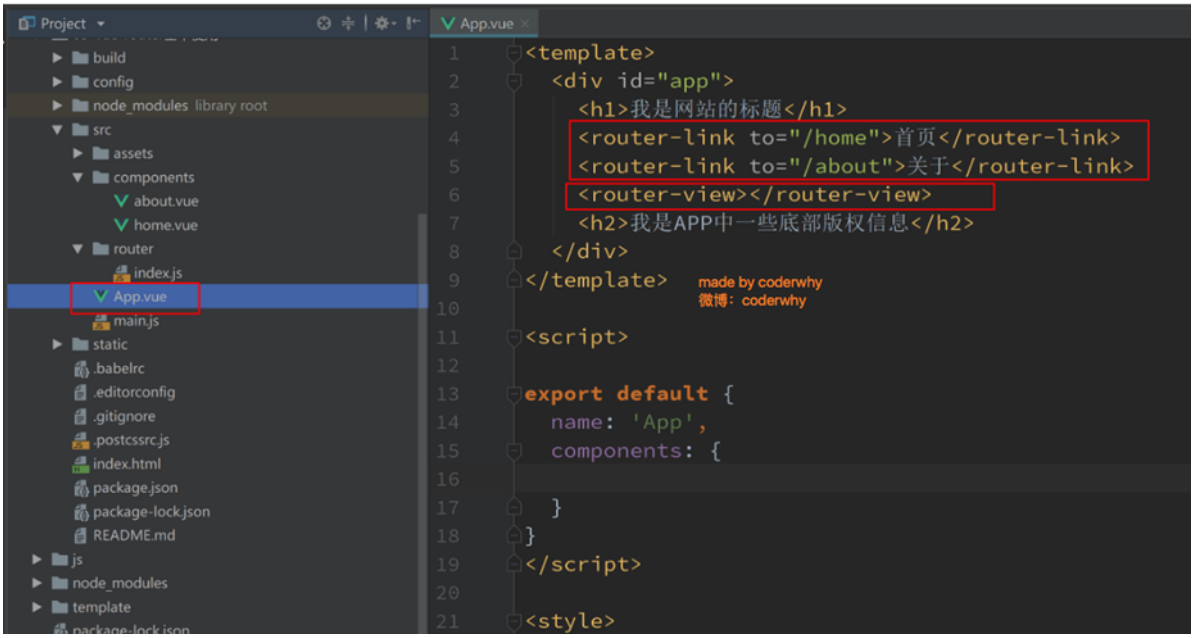
- 在components文件夹中创建路由组件



- 配置组件和路径的映射关系



- 使用路由

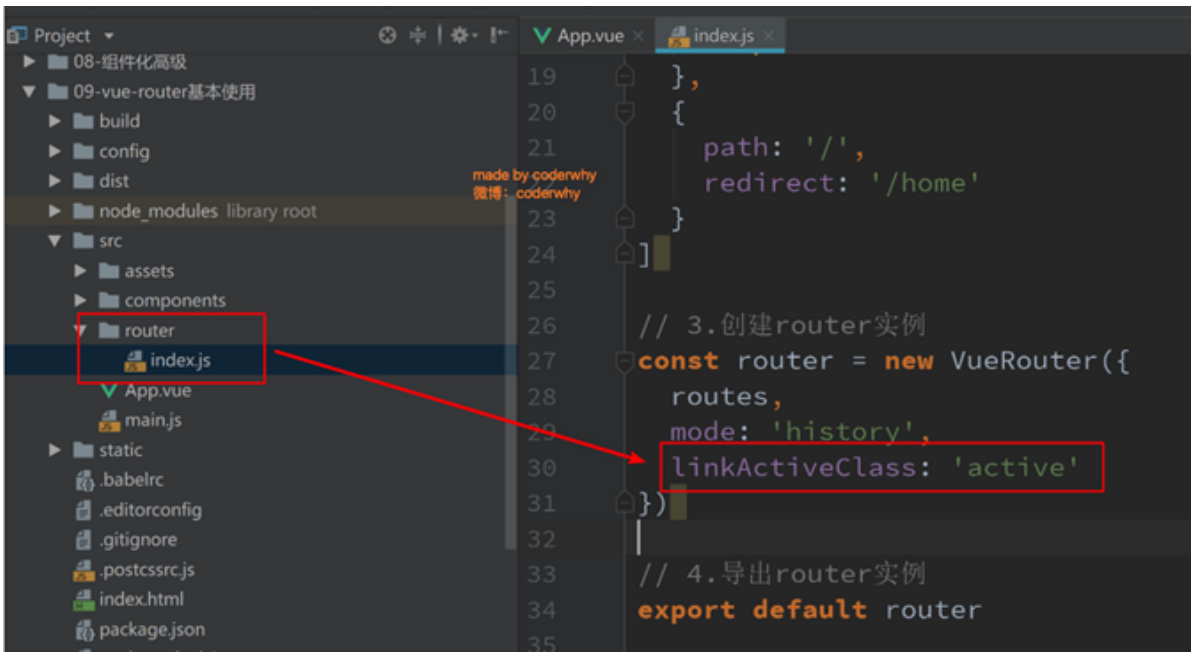


Vue默认使用hash规则，如果想要使用history，则需要修改

```
const router = new VueRouter({
  routes,
  mode: 'history'
})
```

router-link

- : 该标签是一个vue-router中已经内置的组件, 它会被渲染成一个标签，如果不想渲染成标签，可以添加tag属性`<router-link to="/about" tag="button">关于</router-link>`
- 如果不希望返回，使用replace模式，则`<router-link to="/home" replace>首页</router-link>`
- : 该标签会根据当前的路径, 动态渲染出不同的组件.
- 网页的其他内容, 比如顶部的标题/导航, 或者底部的一些版权信息等会和处于同一个等级.
- 在路由切换时, 切换的是挂载的组件, 其他内容不会发生改变
- 通过router-link选中的标签会有一个router-active-class属性，如果想改变class的名字，可以通过`<router-link to="/about" tag="button" active-class="active">关于</router-link>`改变，或者通过



通过代码跳转

通过this指针自带的方法$router.push进行跳转，同样，也可以使用replace方法

```
<template>
  <div id="app">
    <router-view></router-view>
    <button @click="homeClick">首页</button>
    <button @click="aboutClick">关于</button>
  </div>
</template>

<script>
export default {
  name: "App",
  methods: {
    homeClick() {
      this.$router.push("/home");
      console.log("homeclick");
    },
    aboutClick() {
      this.$router.push("/about");
      console.log("aboutClick");
    },
  },
};
</script>

<style>
</style>
```

动态路由

在某些情况下，一个页面的path路径可能是不确定的，比如我们进入用户界面时，希望是如下的路径：

- /user/aaaa或/user/bbbb
- 除了有前面的/user之外，后面还跟上了用户的ID
- 这种path和Component的匹配关系，我们称之为动态路由(也是路由传递数据的一种方式)

步骤如下：

- 首先我们创建一个User.vue文件，并在index.js中配置动态路由映射关系，冒号表示这是一个变量

```
const routes = [{
    path: '/',
    redirect: '/home'
  },
  {
    path: '/home',
    component: Home
  },
  {
    path: '/about',
    component: About
  },
  {
    path: '/user/:userId',
```

```
    component: User
  }
]
```

- 之后在App.vue文件中指定data中userId的值，并在router-link中进行拼接，即可跳转到动态的url地址

```
<template>
  <div id="app">
    <router-link to="/home">首页</router-link>
    <router-link to="/about">关于</router-link>
    <router-link :to="'/user/' + userId">用户</router-link>
    <router-view></router-view>
  </div>
</template>

<script>
export default {
  name: "App",
  data() {
    return {
      userId: "zhangsan",
    };
  },
};
</script>

<style>
</style>
```

- 下面需要考虑的问题是，对于不同的动态url，如何显示相应的userId呢？具体的做法是，在User.vue文件中，通过`this.$route`对象，这个`$route`对象是当前活跃的路由，获取动态的userId，这样就可以将userId显示在页面上。

```
<template>
  <div>
    <h2>我是用户</h2>
    <p>我是用户内容</p>
    <h2>{{ userId }}</h2>
  </div>
</template>

<script>
export default {
  name: "User",
  computed: {
    userId() {
      return this.$route.params.userId;
    },
  },
};
</script>

<style>
</style>
```

进阶

路由懒加载

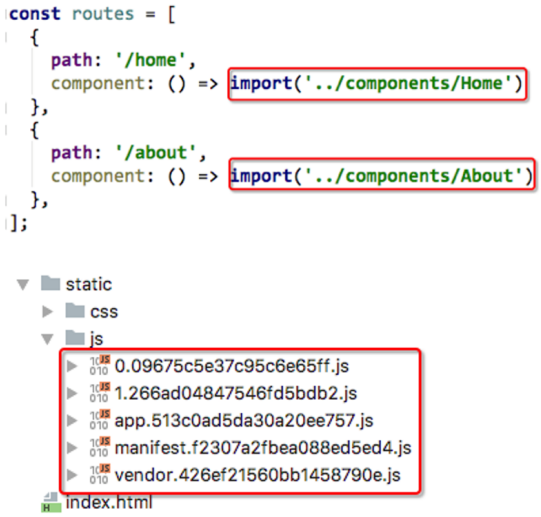
当打包构建应用时，Javascript 包会变得非常大，影响页面加载。

如果我们能把不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应组件，这样就更加高效了

路由懒加载的主要作用就是将路由对应的组件打包成一个个的js代码块，只有在这个路由被访问到的时候, 才加载对应的组件

路由懒加载只需要在index.js中，将各个组件引用的格式进行改变即可：

```
import Home from '../components/Home'
import About from '../components/About'
import User from '../components/User'
//改为如下的形式
const Home = () => import('../components/Home')
const About = () => import('../components/About')
const User = () => import('../components/User')
```



webpack打包后，会将不同的组件打包成不同的js文件

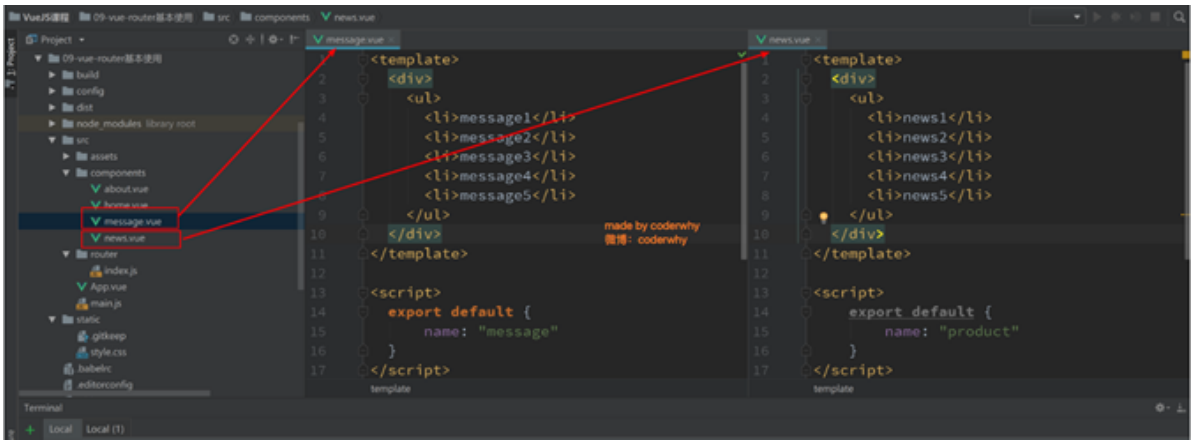
路由嵌套使用

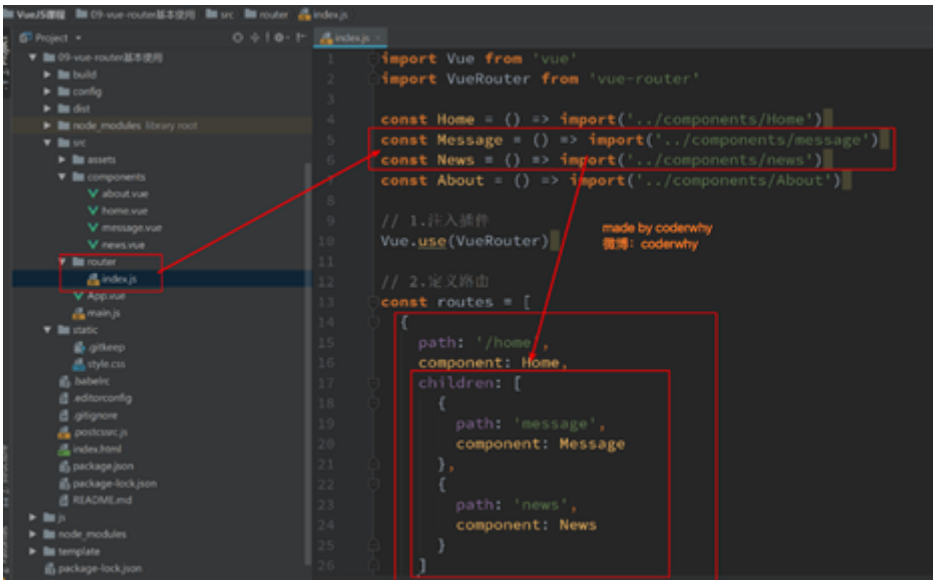
比如在home页面中, 我们希望通过/home/news和/home/message访问一些内容

路由和组件的关系如下：

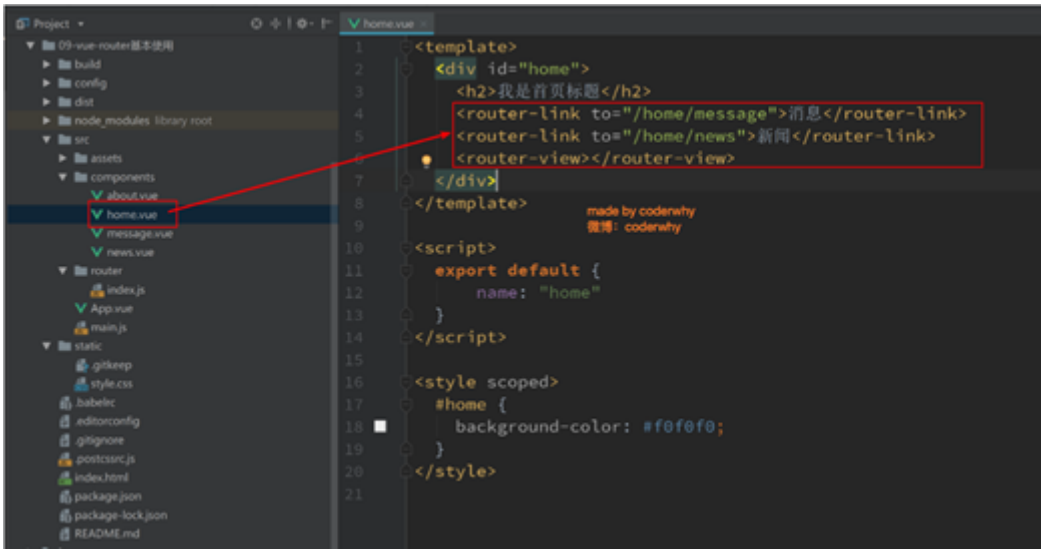
实现嵌套路由有两个步骤:

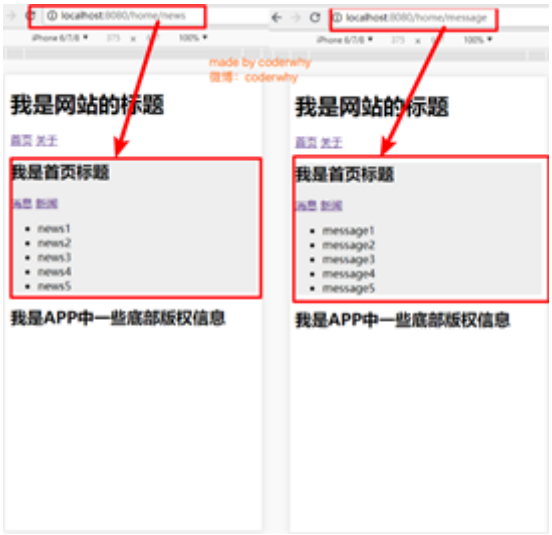
- 创建对应的子组件, 并且在路由映射中配置对应的子路由，在home路径下，通过添加children数组





- 在组件内部使用标签






```
const routes = [{
    path: '/',
    redirect: '/home',

  },
  {
    path: '/home',
    component: Home,
    children: [{
        path: '',
        redirect: 'news'
      },
      {
        path: 'news',
        component: HomeNews
      },
      {
        path: 'message',
        component: HomeMessage
      }
    ]
  },
]
```

```
<template>
```

```
  <div>
    <h2>我是首页</h2>
    <p>我是首页内容</p>
    <router-link to="/home/message">消息</router-link>
    <router-link to="/home/news">新闻</router-link>
    <router-view></router-view>
  </div>
</template>

<script>
export default {
  name: "Home",
};
</script>

<style>
</style>
```

参数传递

有时候url需要携带一些query数据，如何将query数据传递过去呢?

- 创建新的组件Profile.vue
- 配置路由映射
- 添加跳转的router-link，注意此时要通过对象的方式，指定path和query数据

```
<router-link :to="{ path: '/profile', query: { name: 'zhangsan' } }">档案
</router-link>
```

- 之后可以在组件中通过`$route.query`来获取query

```
<template>
  <div>
    <h2>我是profile</h2>
    <h2>{{ $route.query }}</h2>
  </div>
</template>

<script>
export default {
  name: "Profile",
};
</script>

<style>
</style>
```

还可以通过`this.$router.push()`进行路由跳转和参数传递

```
<template>
  <div id="app">
    <button @click="profileClick">档案</button>
    <router-view></router-view>
  </div>
</template>
```

```vue
<script>
export default {
  name: "App",
  methods: {
    profileClick() {
      this.$router.push({
        path: "/profile",
        query: { name: "zhangsan" },
      });
    },
  },
};
</script>

<style>
</style>
```

导航守卫

导航守卫主要用来监听监听路由的进入和离开的

考虑一个需求：在一个SPA应用中, 如何改变网页的标题呢?

可以使用导航守卫，在每一次路由改变时，执行钩子函数，vue-router提供了beforeEach和afterEach的钩子函数, 它们会在路由即将改变前和改变后触发

```vue
router.beforeEach((to, from, next) => {
  window.document.title = to.meta.title
  next()
})
```

每个守卫方法接收三个参数：

- `to: Route`: 即将要进入的目标 路由对象
- `from: Route`: 当前导航正要离开的路由
- `next: Function`: 一定要调用该方法来 **resolve** 这个钩子。执行效果依赖 `next` 方法的调用参数。
  - `next()`: 进行管道中的下一个钩子。如果全部钩子执行完了，则导航的状态就是 **confirmed** (确认的)。
  - `next(false)`: 中断当前的导航。如果浏览器的 URL 改变了 (可能是用户手动或者浏览器后退按钮)，那么 URL 地址会重置到 `from` 路由对应的地址。
  - `next('/')` 或者 `next({ path: '/' })`: 跳转到一个不同的地址。当前的导航被中断，然后进行一个新的导航。你可以向 `next` 传递任意位置对象，且允许设置诸如 `replace: true`、`name: 'home'` 之类的选项以及任何用在 `router-link` 的 `to` prop 或 `router.push` 中的选项。
  - `next(error)`: (2.4.0+) 如果传入 `next` 的参数是一个 `Error` 实例，则导航会被终止且该错误会被传递给 `router.onError()` 注册过的回调。

确保 `next` 函数在任何给定的导航守卫中都被严格调用一次。它可以出现多于一次，但是只能在所有的逻辑路径都不重叠的情况下，否则钩子永远都不会被解析或报错

同样的，还有**全局后置钩子、全局解析守卫**

如果想单独对某一路由配置守卫，有**路由独享守卫**

你可以在路由配置上直接定义 `beforeEnter` 守卫：

```
const router = new VueRouter({
  routes: [
    {
      path: '/foo',
      component: Foo,
      beforeEnter: (to, from, next) => {
        // ...
      }
    }
  ]
})
```

这些守卫与全局前置守卫的方法参数是一样的

keep-alive

keep-alive 是 Vue 内置的一个组件，可以使被包含的组件保留状态，或避免重新渲染

一般我们从一个路由跳转到另一个路由，组件都会被销毁，但是keep-alive可以避免销毁，保留状态，使用方法为：

```
<keep-alive>
   <router-view></router-view>
</keep-alive>
```

将 `router-view` 标签包含在 `keep-alive` 标签内即可

有了keep-alive，我们多了两个生命周期函数 `activated` 和 `deactivated`，可以用于判断当前组件处于活跃/非活跃状态，可以通过生命周期函数 `created` 和 `destroyed` 在创建和销毁时执行操作

现在有一个问题，如果想指定一个组件不被keep-alive约束，离开即销毁，该怎么操作呢？

- exclude：字符串或正则表达式，任何匹配的组件都不会被缓存

```
<keep-alive exclude="Profile,User">
   <router-view></router-view>
</keep-alive>
```

- include：字符串或正则表达，只有匹配的组件会被缓存

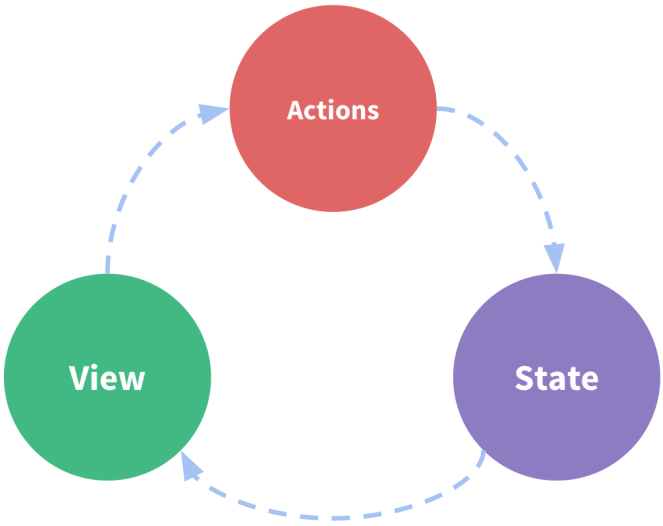
```
<keep-alive include="Profile,User">
   <router-view></router-view>
</keep-alive>
```

注意Profile和User之间的逗号，不能有空格

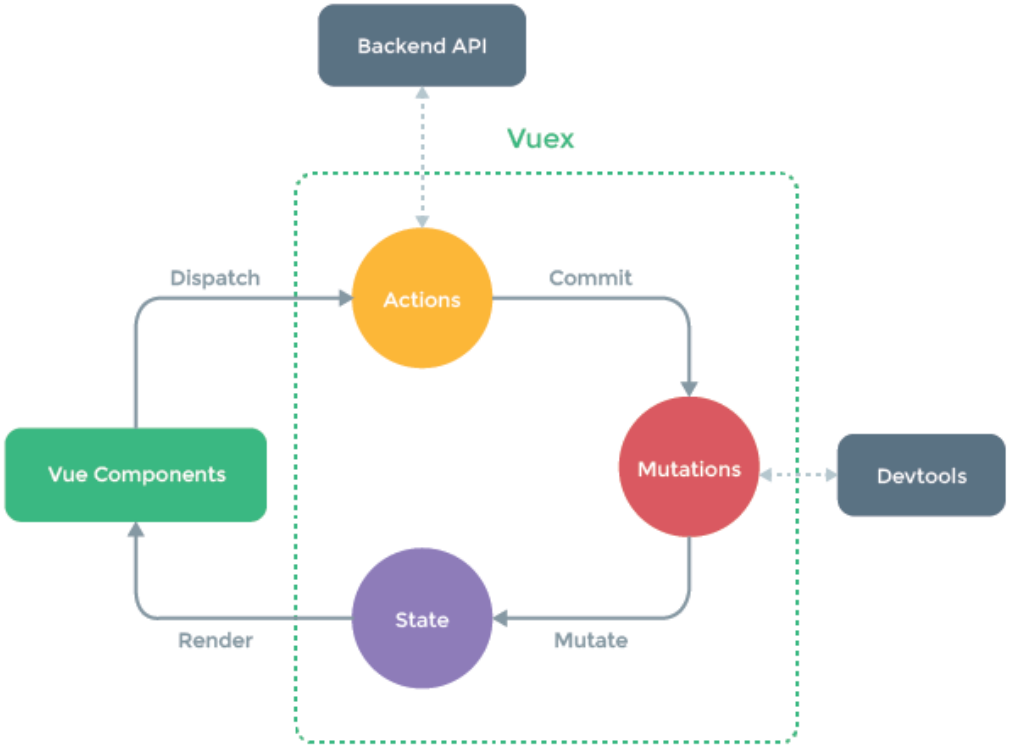
Vuex

Vuex 是一个专为 Vue.js 应用程序开发的**状态管理模式**

状态管理模式、集中式存储管理可以简单的将其看成把需要多个组件共享的变量全部存储在一个对象里面。然后，将这个对象放在顶层的Vue实例中，让其他组件可以使用。则多个组件就可以共享这个对象中的所有变量属性。



- State：是我们的状态。（姑且可以当做就是data中的属性）
- View：视图层，可以针对State的变化，显示不同的信息
- Actions：主要是用户的各种操作：点击、输入等等，会导致状态的改变



State

需求是定义一个counter变量，在父组件和子组件中都可以对其进行修改

- 首先`npm install vuex --save`安装插件
- 在src文件夹中建立一个store文件夹，建立index.js文件，大致和vue-router相同

```
import Vuex from 'vuex'
import Vue from 'vue'


//1.安装插件
Vue.use(Vuex)

//2.创建对象
const store = new Vuex.Store({
  state: {
    counter: 1000
  },
  mutations: {

  },
  actions: {

  },
  getters: {

  },
  modules: {

  }
})

//3.导出store对象
export default store
```

- 在main.js中进行挂载

```
import Vue from 'vue'
import App from './App'
import router from './router'
import store from './store'

Vue.config.productionTip = false

/* eslint-disable no-new */
new Vue({
  el: '#app',
  router,
  store,
  render: h => h(App)
})
```

- 随后在任何组件中都和通过`$store.state.counter`对此状态进行访问

```
<template>
  <div>
    <h2>{{ $store.state.counter }}</h2>
  </div>
</template>

<script>
export default {
```

```
  name: "HelloVuex",
};
</script>

<style>
</style>
```

Getters

有时候，我们需要从store中获取一些state处理后的状态，则需要通过getters（类似于计算属性）

在state中定义一个students数组，存放一些学生信息

- 需求为获取年龄大于20的学生，在getters中定义方法，return通过filter过滤后的数组
- 需求为获取年龄大于20的学生的人数，此时我们传入state和getters两个参数，调用第一个定义的getters，再获取它的length属性
- 需求为获取年龄大于age的学生，return一个函数，传入age参数

```
  getters: {
    morethan20(state) {
      return state.students.filter(s => s.age > 20)
    },
    morethan20Length(state, getters) {
      return getters.morethan20.length
    },
    morethanAge(state, getters) {
      return function (age) {
        return state.students.filter(s => s.age > age)
      }
    },

  },
```

```
    <h2>{{ $store.getters.morethan20 }}</h2>
    <h2>{{ $store.getters.morethan20Length }}</h2>
    <h2>{{ $store.getters.morethanAge(23) }}</h2>
```

Mutations

如果需要定义一些方法进行状态的改变，则需要在mutations中进行定义，下面以改变counter为例

- 在mutaions中定义增加和减少，需要**传入state变量**

```
const store = new Vuex.Store({
  state: {
    counter: 1000
  },
  mutations: {
    increment(state) {
      state.counter++;
    },
    decrement(state) {
      state.counter--;
    }
  },
})
```

- 在组件方法中定义两个点击时间，在点击事件中通过`this.$store.commit("increment")`进行提交，完成增加或减少动作

```
<template>
  <div id="app">
    <h2>{{ message }}</h2>
    <h2>{{ $store.state.counter }}</h2>
    <button @click="addition">+</button>
    <button @click="substraction">-</button>
    <hello-vuex></hello-vuex>
  </div>
</template>

<script>
import HelloVuex from "./components/HelloVuex.vue";

export default {
  name: "App",
  components: {
    HelloVuex,
  },
  data() {
    return {
      message: "App组件",
    };
  },
  methods: {
    addition() {
      this.$store.commit("increment");
    },
    substraction() {
      this.$store.commit("decrement");
    },
  },
};
</script>

<style>
</style>

```

参数传递

在通过mutation更新数据的时候, 有可能我们希望携带一些额外的参数，参数被称为是mutation的载荷(Payload)

mutations中代码：

- 同时传入state和需要的参数
- 如果参数不唯一，可以以对象的形式传递参数，再从对象中取出相关的信息

```
mutations: {
    addCount(state, num) {
      state.counter = state.counter + num
    },
    addStu(state, obj) {
      state.students.push(obj)
    }
  },
```

App中代码：

```
methods: {
    addCount(num) {
      this.$store.commit("addCount", num);
    },
    addStu() {
      const obj = { id: 123, name: "alex", age: 23 };
      this.$store.commit("addStu", obj);
    },
  },
```

```
<template>
  <div id="app">
    <h2>{{ $store.state.students }}</h2>
    <button @click="addCount(5)">+5</button>
    <button @click="addStu">addStu</button>
  </div>
</template>
```

还有另外一种提交方式：

```
methods: {
    addCount(num) {
      this.$store.commit({
          type: 'addCount',
          num,
      });
    },
  },
```

```
mutations: {
    addCount(state, payload) {
      state.counter = state.counter + payload.num
    },
  },
```

此时传入到mutations中的是一个对象，要从这个对象把num取出来

响应规则

Vuex的store中的state是响应式的, 当state中的数据发生改变时, Vue组件会自动更新，这就要求我们必须遵守一些Vuex对应的规则：

- 提前在store中初始化好所需的属性
- 当给state中的对象添加新属性时, 使用下面的方式：
  - 使用`Vue.set(obj, 'newProp', 123)`
  - 用新对象给旧对象重新赋值

否则，数据更新后不会被加入到响应式系统

举例，在state中添加一个info对象，想要给它添加一个address属性

```
  state: {
    info: {
      name: 'Kobe',
      age: '23',
      skill: 'basketball'
    }
  }
```

```
  methods: {
    addProperty() {
      this.$store.commit("addPro");
    },
  },
```

```
  mutations: {
    addPro(state) {
      Vue.set(state.info, 'address', '洛杉矶')//必须用这种方式添加
      Vue.delete(state.info, 'skill')//通过这种方式删除
      //通过普通的对象添加、删除属性的方式无法保证响应式
    }
  },
```

常量类型

在mutation中, 我们定义了很多事件类型(也就是其中的方法名称)，当我们的项目增大时, Vuex管理的状态越来越多, 需要更新状态的情况越来越多, 那么意味着Mutation中的方法越来越多，方法过多, 使用者需要花费大量的经历去记住这些方法, 甚至是多个文件间来回切换, 查看方法名称, 甚至如果不是复制的时候, 可能还会出现写错的情况，因此可以创建一个文件: mutation-types.js, 并且在其中定义我们的常量

index.js | mutation-types.js | App.vue

```
export const UPDATE_INFO = 'UPDATE_INFO'
```

```
import Vuex from 'vuex'
import Vue from 'vue'
import * as types from './mutation-types'

Vue.use(Vuex)

const store = new Vuex.Store({
  state: {
    info: {
      name: 'why', age: 18
    }
  },
  mutations: {
    [types.UPDATE_INFO](state, payload) {
      state.info = {...state.info, 'height': payload.height}
    }
  }
})

export default store
```

```
<script>
  import {UPDATE_INFO} from "./store/mutation-types";

  export default {
    name: 'App',
    components: {
    },
    computed: {
      info() {
        return this.$store.state.info
      }
    },
    methods: {
      updateInfo() {
        this.$store.commit(UPDATE_INFO, {height: 1.88})
      }
    }
  }
</script>
```

Actions

Mutation中不能进行异步操作，如果要在Vuex中进行一些异步操作, 比如网络请求，则需要用到actions，基本使用代码如下：

- 首先在mutations中定义事件
- 在actions中定义事件，传入context参数，并且把异步代码放到actions中，进行提交

```
const store = new Vuex.Store({
  state: {
    info: {
      name: 'Kobe',
      age: '23',
      skill: 'basketball'
    }
  },
  mutations: {
    updateInfo(state) {
      Vue.set(state.info, 'address', '洛杉矶')
    }
  },
  actions: {
    aupdateInfo(context) {
      setTimeout(() => {
        context.commit('updateInfo')
      }, 2000)
    }
  },
})
```

- 在App中对actions中的方法进行提交

```
  methods: {
    updateInfo() {
      this.$store.dispatch("aupdateInfo");
    },
  }
```

相当于绕了一个弯，先提交actions，再在action中提交mutations，完成异步操作

在Action中, 我们可以将异步操作放在一个Promise中, 并且在成功或者失败后, 调用对应的resolve或reject

```
  actions: {
    aupdateInfo(context, payload) {
      return new Promise((resolve, reject) => {
        setTimeout(() => {
          context.commit('updateInfo')
          console.log(payload)
          resolve('111111111111')
        }, 2000)
      })
    }
  },
```

```
 methods: {
   updateInfo() {
     this.$store.dispatch("aupdateInfo", "i'm the message").then((res) => {
       console.log("里面完成了提交");
       console.log(res);
     });
   },
 },
```

Modules

Module是模块的意思，Vue使用单一状态树,那么也意味着很多状态都会交给Vuex来管理，当应用变得非常复杂时,store对象就有可能变得相当臃肿，为了解决这个问题，Vuex允许我们将store分割成模块(Module)，而每个模块拥有自己的state、mutation、action、getters等

- 首先定义module对象，并在store中注册
- 访问state里对象的时候要先通过模块a，再进行访问
- 同样可以定义mutations和getters，并且依然通过 `this.$store` 进行直接调用
- 使用getters时，可以携带第三个参数 `rootState`，用以访问根store中的state对象

```
const moduleA = {
  state: {
    name: 'zhangsan'
  },
  mutations: {
    updateName(state, payload) {
      state.name = payload
    }
  },
  getters: {
    fullname1(state) {
      return state.name + '11111'
    },
    fullname2(state, getters) {
      return getters.fullname1 + '2222'
    },
    fullname3(state, getters, rootState) {
      return getters.fullname2 + rootState.counter
    }
  },
  actions: {

  }

}


const store = new Vuex.Store({
  state: {
    counter: 1000,
  },
  mutations: {

  },
  actions: {

```

```
  },
  getters: {

  },
  modules: {
    a: moduleA
  }
})
```

```
<h2>{{ $store.state.a.name }}</h2>
<button @click="updateName('lisi')">修改名字</button>
<h2>{{ $store.getters.fullname1 }}</h2>
<h2>{{ $store.getters.fullname2 }}</h2>
<h2>{{ $store.getters.fullname3 }}</h2>


<script>
import HelloVuex from "./components/HelloVuex.vue";

export default {
  name: "App",
  components: {
    HelloVuex,
  },
  data() {
    return {
      message: "App组件",
      counter: 0,
    };
  },
  methods: {
    updateName(name) {
      this.$store.commit("updateName", name);
    },
  },
};
</script>
```

Axios

Axios是用于发送异步的网络请求，主要有以下几种API

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

基本使用

首先要先通过`npm install axios --save`安装axios

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

Vue.config.productionTip = false

/* eslint-disable no-new */
new Vue({
  el: '#app',
  render: h => h(App)
})

//设定默认配置信息
//axios.defaults.baseURL = 'http://123.207.32.32:8000'
//axios.defaults.timeout = 5000


//get请求
axios.get('http://123.207.32.32:8000/home/multidata')
.then({
    console.log(res)
})

//axios支持promise，可以直接调用then
axios({
  url: 'http://123.207.32.32:8000/home/multidata',
}).then(res => {
  console.log(res);
})

//axios携带参数，可以用params，也可以直接放到url里
axios({
  url: 'http://123.207.32.32:8000/home/data',
  params: {
    type: 'pop',
    page: 1
  }
}).then(res => {
  console.log(res);
})

//发送并发请求
axios.all([axios({
  url: 'http://123.207.32.32:8000/home/multidata'
}), axios({
  url: 'http://123.207.32.32:8000/home/data',
  params: {
    type: 'sell',
    page: 3
  }
})]).then(results => {
  console.log(results);
})
//也可以写成
axios.all([axios({
```

```
  url: 'http://123.207.32.32:8000/home/multidata'
}), axios({
  url: 'http://123.207.32.32:8000/home/data',
  params: {
    type: 'sell',
    page: 3
  }
})]).then(axios.spread((res1, res2) => {
  console.log(res1);
  console.log(res2);
}))

```

封装

有时候不同的请求需要不同的baseUrl，所以我们可以创建多个axios实例

```
const instance1 = axios.create()
instance1.defaults.baseURL = 'http://152.136.185.210:7878/api/m5'
instance1({
  url: '/home/multidata'
}).then(res => {
  console.log(res);
})
```

首先可以通过回调函数进行封装，创建一个networks文件夹，里面创建一个requests.js文件

```
import axios from 'axios'

export function requests(config) {
  const instance1 = axios.create({
      baseURL: 'http://152.136.185.210:7878/api/m5',
      timeout: 5000
  })

  instance1(config.url).then(res => {
    config.success(res)
  }, err => {
    config.failure(err)
  })
}
```

使用：

```
import {
  requests
} from "./networks/requests";

requests({
  url: "/home/multidata",
  success(res) {
    console.log(res);
    this.message = res;
  },
  failure(err) {
```

```
    console.log(err);
  },
});
```

此时，requests.js文件还可以进一步封装，因为axios对象本身就支持promise，所以可以直接将其返回

```
export function requests(config) {
  const instance1 = axios.create({
    baseURL: 'http://152.136.185.210:7878/api/m5',
    timeout: 5000
  })

  return instance1(config)
}
```

使用：

```
requests({
  url: "/home/multidata",
}).then(res => {
  console.log(res);
}, err => {
  console.log(err);
})
```

拦截器

有时候需要对网络请求和响应进行拦截，则需要用到拦截器

```
//请求拦截
  instance1.interceptors.request.use(config => {
    console.log(config);

    //拦截过后需要再把config重新return回去，请求才能继续
    return config
  }, err => {
    console.log(err);
  })

  //响应拦截
  instance1.interceptors.response.use(res => {
    console.log(res);
    return res.data
  }, err => {
    console.log(err);
  })
```