Vue技術點之間的聯繫

Vue.js技術棧開發前端PC端項目：

* Vue.js語法
* Vue-router
* Axios
* Webpack
* Element-UI
* Vuex

Vue.js技術棧開發前端移動端項目：

* Vue.js語法
* Vue-router
* Axios
* Webpack
* Vant-UI
* Vuex

Vue基礎

## 1 Vue概述

### （1）創始

尤雨溪是Vue.js的創建者

2014年2月，Vue.js正是發佈

2015年10月27日，正是發佈1.0.0

2016年4月27日，發佈2.0的預覽版本

官網：https://cn.vuejs.org/v2/guide/

Vue很高效：20kb運行大小，超快虛擬DOM。

### （2）Vue是漸進式JavaScript框架

#### ① 框架和庫的區別

框架以提供基礎性的服務為主。比如Vue有雙向數據綁定的支撐，可以減少實際開發時的代碼量。

庫以提供API為主。比如jQuery。

現在框架和庫的界限不是很明顯了。所以很多時候把兩者理解成一個東東。

Vue很靈活，在一個庫和一套完整框架之間自如伸縮。

#### ② 漸進式

聲明式渲染→組件系統→客戶端路由→集中式狀態管理→項目構建

## 2 Vue基本使用

  <body>

    <div id="app">

      <!-- {{}} 是插值表達式，作用是把數據填充到HTML標籤裡 -->

      <div>{{msg}}</div>

      <!-- 差值表達式支持基本的計算操作。比如下面的標籤裡顯示的數字是3。-->

      <div>{{1+2}}</div>

      <!-- 只要符合JS語法規則的表達式都可以放在這裡。 -->

      <div>{{msg + '------' + 123}}</div>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      new Vue({

        el: "#app", //元素的掛載位置，值可以是CSS選擇器或者DOM元素。掛載是指把數據關聯到頁面的某個標籤裡。

        // data是模型數據，即開發中遇到的業務相關數據，值是一個對象。

        data: {

          msg: "Hello World by Vue",

        },

      });

    </script>

  </body>

Vue不需要直接操作DOM了。

Vue代碼運行原理分析

Vue.js把Vue代碼編譯為原生JS代碼，以便瀏覽器解析。

## 3 Vue模板語法

### （1）前端渲染

Vue是用來做前端渲染的。

前端渲染即把從數據庫請求回來的數據填充到HTML標籤中。

前端渲染的產物是靜態HTML內容，這內容最終被瀏覽器呈現給用戶。

### （2）前端渲染的方式

#### ① 原生js拼接字符串

缺點：不同開發人員的代碼風格差別很大；隨著業務越來越複雜，後期的維護變得逐漸困難起來。

#### ② 使用前端模板引擎，比如art-template

優點：大家都遵循同樣的規則寫代碼，代碼可讀性明顯提高，方便後期維護。

缺點：沒有專門提供事件機制。

#### ③ 使用vue特有的模板語法

模板語法概覽：

* 插值表達式
* 指令
* 事件綁定
* 屬性綁定
* 樣式綁定
* 分支循環結構

### （3）什麼是指令？

指令的本質就是自定義屬性。

比如下面的abc=123就是一個自定義屬性。標準屬性例子：a標籤中的href屬性。

<div id="app" abc=123>

指令的格式：以v-開始，後面跟個字符串，比如v-cloak。

### （4）v-cloak指令用法

* 差值表達式存在的問題：閃動（如果網絡不好，頁面會先顯示花括號，再把花括號替換為相關的數據內容，這個過程就叫做閃動，給用戶造成不好的體驗。）
* 如何解決該問題？使用v-cloak指令

    <style type="text/css">

      [v-clock] {

        display: none;

      }

    </style>

      <div v-cloak>{{msg}}</div>

* 解決該問題的原理：先通過樣式隱藏內容，然後在內存中進行值的替換，替換好值之後再顯示最終的結果。

### （5）數據綁定指令v-text，v-html，v-pre

1. v-text 填充純文本
   * 沒有閃動問題
   * 比插值表達式更加簡潔
   * 推薦用v-text而非插值表達式
2. v-html 填充HTML片段
   * 存在安全問題
   * 本網站內部數據可以使用，來自第三方的數據不可以用
3. v-pre 填充原始信息
   * 顯示原始信息，跳過編譯過程（分析編譯過程）

範例代碼：

  <body>

    <div id="app">

      <div v-text="msg"></div>

      <div v-html="msg1"></div>

      <div v-pre>{{msg}}</div>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      new Vue({

        el: "#app",

        data: {

          msg: "Hello World by Vue",

          msg1: "<h1>HTML標籤h1</h1>",

        },

      });

    </script>

  </body>

顯示的樣子：



### （6）數據響應式和v-once

響應式類別：

* HTML5中的響應式（屏幕尺寸的變化導致樣式的變化）
* 數據的響應式（數據的變化導致頁面內容的變化）

數據綁定：把數據填充到標籤中（v-text，v-html，v-pre的功能，它們綁定的數據默認都是響應式的）

v-once只編譯一次，顯示內容之後不再具有響應式功能，可以提高性能。

### （7）雙向綁定數據v-model

在input中添加v-model功能，可以修改data中msg的值，進而影響div中的顯示內容

  <body>

    <div id="app">

      <div>{{msg}}</div>

      <input type="text" v-model="msg" />

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      new Vue({

        el: "#app",

        data: {

          msg: "Hello World by Vue",

        },

      });

    </script>

  </body>

### （8）與雙向數據綁定相關的MVVM設計思想

* M（model）
  + Plain JavaScript Objects，比如上面的例子中data中的數據
* V（view）
  + 本質上是DOM元素
* VM（view-model）
  + 實現控制邏輯，把上面兩者結合在一起。



### （9）事件綁定

v-on指令用法

<input type="button" v-on:click="num++" />

v-on指令簡寫

<input type="button" @click="num++" />

事件函數的調用方式：

* 直接綁定函數名稱

<button v-on:click="say">Hello</button>

* 調用函數（要傳遞參數的話，只能用這種方式）

<button v-on:click="say()">Hi</button>

函數定義在methods屬性中

示例代碼

  <body>

    <div id="app">

      <div>{{num}}</div>

      <button v-on:click="say">Hello</button>

      <button v-on:click="say()">Hi</button>

      <button v-on:click="inc">點擊+1</button>

      <button v-on:click="inc()">點擊+1</button>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          msg: "Hello World by Vue",

          num: 1,

        },

        methods: {

          say: function () {

            console.log("LOL");

          },

          inc: function () {

            // this === vm。在這裡，this指的是Vue的實例對象vm

            this.num++;

          },

        },

      });

    </script>

  </body>

### （10）事件函數參數傳遞

<button v-on:click="say('hi', $event)">Hi</button>

如果沒用到$event，傳參數的時候可以不寫$event。

下面這種直接綁定的方式默認了會傳遞$event參數

<button v-on:click="say">Hello</button>

示例代碼

  <body>

    <div id="app">

      <!-- 傳遞1個參數 -->

      <button v-on:click="say1('LOL1')">Hi 1個參數</button>

      <!-- 傳遞多個參數 -->

      <button v-on:click="say2('LOL2','lala2', $event)">Hi 多個參數</button>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          msg: "Hello World by Vue",

          num: 1,

        },

        methods: {

          say1: function (*p1*) {

            console.log(p1);

          },

          say2: function (*p1*, *p2*, *event*) {

            console.log(p1, p2, event);

          },

        },

      });

    </script>

  </body>

### （11）事件修飾符

.stop阻止冒泡

<a v-on:click.stop="handle">跳轉</a>

.prevent阻止默認行為

<a href="http://www.bing.com" v-on:click.prevent="handle">跳轉</a>

事件修飾符可以連著寫。寫的先後順序不同，有時會有不同的效果。

既阻止冒泡又阻止默認行為

<a href="http://www.bing.com" v-on:click.stop.prevent="handle">跳轉</a>

### （12）按鍵修飾符

.enter回車鍵

<input type="text" v-on:keyup.enter="handle" />

.delete刪除鍵

<input type="text" v-on:keyup.delete="handle" />

### （13）自定義按鍵修飾符

全局config.keyCodes對象

比如，按下a，會觸發handle函數

<input type="text" v-on:keyup.65="handle" />

### （14）屬性綁定v-bind

<a v-bind:href="url">跳轉</a>

縮寫形式

<a :href="url">跳轉</a>

url是寫在data裡面的數據

  <body>

    <div id="app">

      <a :href="url">跳轉</a>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          url: "http://www.bing.com",

        },

        methods: {

          handle: function () {

            console.log("handle");

          },

        },

      });

    </script>

  </body>

### （15）v-model底層實現原理

  <body>

    <div id="app">

      <!-- 下面兩種寫法都可以 -->

      <input type="text" v-on:input="handle" />

      <input type="text" v-on:input="msg=$event.target.value" />

      <div>{{msg}}</div>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          msg: "天朗氣清",

        },

        methods: {

          handle: function (*event*) {

            this.msg = event.target.value;

          },

        },

      });

    </script>

  </body>

### （16）樣式綁定

#### ① 對象語法

<div v-bind:class="{active:isActive}"></div>

示例代碼

    <style type="text/css">

      .active {

        width: 100px;

        height: 50px;

        border: 5px solid skyblue;

      }

      .error {

        background-color: yellow;

      }

    </style>

  <body>

    <div id="app">

      <div v-bind:class="{active:isActive1}">1</div>

      <div v-bind:class="{active:isActive, error:isError}">2</div>

      <button v-on:click="handle1">操縱邊框1</button>

      <button v-on:click="handle2">操縱邊框2</button>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          isActive1: true,

          isActive: true,

          isError: true,

        },

        methods: {

          handle1: function () {

            this.isActive1 = !this.isActive1;

          },

          handle2: function () {

            this.isActive = !this.isActive;

            this.isError = !this.isError;

          },

        },

      });

    </script>

  </body>

#### ② 數組語法

<div v-bind:class="[activeClass, errorClass]"></div>

示例代碼

    <style type="text/css">

      .active {

        width: 100px;

        height: 50px;

        border: 5px solid skyblue;

      }

      .error {

        background-color: yellow;

      }

    </style>

  <body>

    <div id="app">

      <div v-bind:class="[activeClass, errorClass]"></div>

      <button v-on:click="handle">操作背景色</button>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          activeClass: "active",

          errorClass: "error",

        },

        methods: {

          handle: function () {

            this.errorClass = "";

          },

        },

      });

    </script>

  </body>

#### ③ 對象綁定和數組綁定可以結合使用

    <style type="text/css">

      .active {

        width: 100px;

        height: 50px;

        border: 5px solid skyblue;

      }

      .error {

        background-color: yellow;

      }

      .test {

        color: salmon;

      }

    </style>

  <body>

    <div id="app">

      <div v-bind:class="[activeClass, errorClass,{test: isTest}]">test</div>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          activeClass: "active",

          errorClass: "error",

          isTest: true,

        },

        methods: {},

      });

    </script>

  </body>

#### ④ class綁定的值可以簡化操作

    <style type="text/css">

      .active {

        width: 100px;

        height: 50px;

        border: 5px solid skyblue;

      }

      .error {

        background-color: yellow;

      }

      .test {

        color: salmon;

      }

    </style>

  <body>

    <div id="app">

      <div v-bind:class="arrClasses">test</div>

      <div v-bind:class="objClasses">test</div>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          arrClasses: ["active", "error", "test"],

          objClasses: {

            active: true,

            error: true,

            test: true,

          },

        },

        methods: {},

      });

    </script>

  </body>

#### ⑤ 處理默認的class

原有的class不會被覆蓋。在原有的class基礎上追加class。

    <style type="text/css">

      .active {

        width: 100px;

        height: 50px;

        border: 5px solid skyblue;

      }

      .error {

        background-color: yellow;

      }

      .test {

        color: salmon;

      }

      .base {

        font-size: 28px;

      }

    </style>

  <body>

    <div id="app">

      <div class="base" v-bind:class="arrClasses">test</div>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          arrClasses: ["active", "error", "test"],

        },

        methods: {},

      });

    </script>

  </body>

#### ⑥ style樣式處理

方式1

  <body>

    <div id="app">

      <div

        v-bind:style="{border: borderStyle, width:widthStyle, height:heightStyle}"

      >

        test

      </div>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          borderStyle: "1px solid skyblue",

          widthStyle: "200px",

          heightStyle: "100px",

        },

        methods: {},

      });

    </script>

  </body>

方式2

  <body>

    <div id="app">

      <div v-bind:style="objStyles">test</div>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          objStyles: {

            border: "1px solid skyblue",

            width: "200px",

            height: "100px",

          },

        },

        methods: {},

      });

    </script>

  </body>

方式3，用數組的方式把樣式對象羅列上，前面有的樣式會被覆蓋，沒有的會加上

  <body>

    <div id="app">

      <div v-bind:style="[objStyles,overrideStyles]">test</div>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          objStyles: {

            border: "1px solid skyblue",

            width: "200px",

            height: "100px",

          },

          overrideStyles: {

            border: "5px solid #355c7d",

            backgroundColor: "pink",

          },

        },

        methods: {},

      });

    </script>

  </body>

### （17）分支結構

分支結構：

* v-if
* v-else
* v-else-if
* v-show

v-if不會渲染標籤；v-show會把標籤渲染出來，如果不需要顯示，則把標籤設置為display:none，如果需要顯示，則不給標籤設置display:none。

### （18）循環結構

v-for遍歷數組

<li v-for=’item in list’>{{item}}</li>

<li v-for=’(item, index) in list’>{{item}} + ‘---’ + {{index}}</li>

（其中item和index都是自定義的，愛叫啥叫啥）

key的作用：幫助vue區分不同的元素，從而提高性能

<li :key=’item.id’ v-for=’(item, index) in list’>{{item}} + ‘---’ + {{index}}</li>

<li :key=’index’ v-for=’(item, index) in list’>{{item}} + ‘---’ + {{index}}</li>

v-for遍歷對象

<div v-for=’(value, key, index) in object’></div>

v-if和v-for結合使用遍歷對象：滿足了某個條件才渲染標籤。同理，v-if可以和上述的v-for遍歷數組結合使用。

<div v-if=’value===12’ v-for=’(value, key, index) in object’></div>

（其中value、key、index都是自定義的，愛叫啥叫啥）

## 4 基礎案例

4-1 Vue選項卡案例

聲明式編程：模板的結構和最終顯示的效果基本一致。

## 5 Vue常用特性

### （1）表單操作

* input單行文本
* radio單選框
* checkbox多選框
* textarea多行文本
* select下拉多選

用:prevent阻止表單的默認提交操作

<input type="submit" value="提交" @click:prevent="handle" />

複選框的寫法示例代碼：

  <body>

    <div id="app">

      <span>爱好：</span>

      <input type="checkbox" id="ball" value="1" v-model="hobbies" />

      <label for="ball">篮球</label>

      <input type="checkbox" id="sing" value="2" v-model="hobbies" />

      <label for="sing">唱歌</label>

      <input type="checkbox" id="code" value="3" v-model="hobbies" />

      <label for="code">写代码</label>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          hobbies: ["2", "3"],

        },

        methods: {},

      });

    </script>

  </body>

下拉框-單選 示例代碼

  <body>

    <div id="app">

      <span>职业：</span>

      <select v-model="occupation">

        <option value="0">请选择职业...</option>

        <option value="1">教师</option>

        <option value="2">软件工程师</option>

        <option value="3">律师</option>

      </select>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          occupation: 2,

        },

        methods: {},

      });

    </script>

  </body>

下拉框-多選 示例代碼

  <body>

    <div id="app">

      <span>职业：</span>

      <select v-model="occupation" multiple="true">（也可以把=“true”省略）

        <option value="0">请选择职业...</option>

        <option value="1">教师</option>

        <option value="2">软件工程师</option>

        <option value="3">律师</option>

      </select>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          occupation: ["2", "3"],

        },

        methods: {},

      });

    </script>

  </body>



手動多選是shift+鼠標左鍵單擊

### （2）表單域修飾符

* number: 轉化為數值
* trim: 去掉開始和結尾的空格
* lazy: 將input事件切換為change事件

input事件是每次輸入內容時，都會被觸發；change事件是失去焦點時才被觸發。

示例代碼

  <body>

    <div id="app">

      <input type="number" v-model.number="age" />

      <input type="text" v-model.trim="info" />

      <input type="text" v-model.lazy="msg" />

      <div>{{msg}}</div>

      <button @click="handle">点击</button>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          age: "",

          info: "",

          msg: "",

        },

        methods: {

          handle: function () {

            console.log(this.info.length);

          },

        },

      });

    </script>

  </body>

### （3）自定義指令

#### ① 不帶參數

在內置指令不滿足需求的情況下，自定義指令應運而生。

自定義指令的語法規則，比如獲取元素焦點：

在script裡定義指令：

      Vue.directive("focus", {

        inserted: function (*el*) {

          //獲取元素的焦點

          // el表示指令綁定的元素

          el.focus();

        },

      });

在標籤裡使用指令：

      <input type="text" v-focus />

#### ② 攜帶參數

inserted和bind大多數情況下通用，只不過觸發時機不同。兩者使用的參數也是一樣的。

inserted：被綁定元素插入父節點時調用（僅保證父節點存在，但不一定已被插入文檔中）。

bind：只調用一次，指令第一次綁定到元素時調用。可以進行一次性的初始化設置。

在script裡定義

      Vue.directive("color", {

        bind: function (*el*, *binding*) {

          console.log(binding.value.color);

          el.style.backgroundColor = binding.value.color;

        },

      });

在標籤裡使用

<input type="text" v-color='{color:"skyblue"}' />

或者這樣：

  <body>

    <div id="app">

      <input type="text" v-color="msg" />

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      Vue.directive("color", {

        bind: function (*el*, *binding*) {

          el.style.backgroundColor = binding.value.color;

        },

      });

      var vm = new Vue({

        el: "#app",

        data: {

          msg: {

            color: "skyblue",

          },

        },

        methods: {},

      });

    </script>

  </body>

#### ③ 局部指令

局部指令只能在本組件中使用，而全局指令沒有這個限制。

  <body>

    <div id="app">

      <input type="text" v-color="msg" />

      <input type="text" v-focus />

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          msg: {

            color: "skyblue",

          },

        },

        methods: {},

        directives: {

          color: {

            bind: function (*el*, *binding*) {

              el.style.backgroundColor = binding.value.color;

            },

          },

          focus: {

            inserted: function (*el*) {

              el.focus();

            },

          },

        },

      });

    </script>

  </body>

### （4）計算屬性

為何需要計算屬性？

表達式的計算邏輯可能會比較複雜，使用計算屬性可以使模板內容更加簡潔。

* 寫函數的時候記得return
* 使用的時候直接寫函數的名字（比如下面的reverseString），不用帶括號

  <body>

    <div id="app">

      <div>{{msg}}</div>

      <div>{{reverseString}}</div>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          msg: "Hello",

        },

        methods: {},

        computed: {

          reverseString: function () {

            return this.msg.split("").reverse().join("");

          },

        },

      });

    </script>

  </body>

### （5）計算屬性和方法的區別

* 計算屬性是基於它們的依賴（即data中的數據）進行緩存的。（只要data中相關的數據不變化，計算屬性只會計算一次，然後存在緩存中，接下來每次調用都用緩存中的結果。）
* 方法（methods）不存在緩存一說。（每次都會重新調用方法）
* 調用計算屬性時不需要加括號。
* 調用方法需要加括號。

### （6）偵聽器

#### ① 偵聽器基礎

偵聽器用於偵聽data。一旦data變化，偵聽器通知其綁定的方法。

偵聽器應用場景：數據變化時執行異步或開銷較大的操作。

示例：

修改input裡面的姓或者名，下面div裡顯示的全名跟著變化。

  <body>

    <div id="app">

      <div>

        <span>姓：</span>

        <span>

          <input type="text" v-model="familyName" />

        </span>

      </div>

      <div>

        <span>名：</span>

        <span>

          <input type="text" v-model="givenName" />

        </span>

      </div>

      <div>{{fullName}}</div>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          familyName: "Zhu",

          givenName: "Yan",

          fullName: "Zhu Yan",

        },

        watch: {

          familyName: function (*val*) {

            this.fullName = val + " " + this.givenName;

          },

          givenName: function (*val*) {

            this.fullName = this.familyName + " " + val;

          },

        },

      });

    </script>

  </body>

#### ② 偵聽器案例：檢測用戶名是否可用

需求：輸入框中輸入姓名，失去焦點時驗證是否存在。如果存在，提示重新輸入；如果不存在，提示可以使用。

步驟：

1. 通過v-model實現數據綁定
2. 需要提供提示信息
3. 需要偵聽器監聽輸入信息的變化
4. 需要修改觸發的事件

示例代碼

  <body>

    <div id="app">

      <div>

        <span>用戶名：</span>

        <span>

          <input type="text" v-model.lazy="uname" />

        </span>

        <span>{{tip}}</span>

      </div>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          uname: "",

          tip: "",

        },

        methods: {

          checkName: function (*uname*) {

            // 調用接口，可以使用定時任務的方式模擬接口調用

            var that = this;

            setTimeout(function () {

              // 模擬接口調用

              if (uname == "admin") {

                that.tip = "用戶名已存在，請更換一個";

              } else {

                that.tip = "用戶名可以使用";

              }

            }, 1000);

          },

        },

        watch: {

          /\*

          偵聽器

          1 採用偵聽器監聽用戶名的變化

          2 調用後台接口進行驗證

          3 根據驗證的結果調整提示信息

          \*/

          uname: function (*val*) {

            // 修改提示信息

            this.tip = "正在驗證...";

            // 調用後台接口驗證用戶名是否合規

            this.checkName(val);

          },

        },

      });

    </script>

  </body>

### （7）過濾器

#### ① 過濾器基本用法

格式化數據，比如將字符串格式化為首字母大寫，將日期格式化為指定的格式等。

自定義過濾器

Vue.filter(‘過濾器名稱’, function(value){

// 過濾器業務邏輯

})

使用過濾器

<div>{{msg | upper }}</div>

<div>{{msg | upper | lower }} </div> （可以同時使用多個過濾器，把前面的過濾器返回的結果作為下一個過濾器接收的參數）

<div v-bind:id= “id | formatId”></div>

支持局部過濾器

filters: {

capitalize: function(){}

}

全局過濾器在所有地方都可以使用，局部過濾器只在組件內部使用。

  <body>

    <div id="app">

      <input type="text" v-model="msg" />

      <div>{{msg | upper}}</div>

      <div :abc="msg | upper">testing</div>

      <div>{{msg | lower}}</div>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      // 過濾器

      Vue.filter("upper", function (*val*) {

        return val.charAt(0).toUpperCase() + val.slice(1);

      });

      var vm = new Vue({

        el: "#app",

        data: {

          msg: "",

        },

        filters: {

          lower: function (*val*) {

            return val.charAt(0).toLowerCase() + val.slice(1);

          },

        },

      });

    </script>

  </body>

#### ② 帶參數的過濾器案例

定義過濾器

Vue.filter(‘format’, function(value, arg1){

// value是下面的date，arg1是下面的’yyyy-mm-dd’

}

使用過濾器

<div>{{date | format(‘yyyy-mm-dd’)}}</div>

案例：使用過濾器格式化日期

  <body>

    <div id="app">

      <div>{{date | format('yyyy-mm-dd')}}</div>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      // 過濾器

      Vue.filter("format", function (*value*, *arg*) {

        if (arg === "yyyy-mm-dd") {

          var result = "";

          var mm = value.getMonth() + 1;

          mm = mm < 10 ? "0" + mm : mm;

          var dd = value.getDate();

          dd = dd < 10 ? "0" + dd : dd;

          return (result = value.getFullYear() + "-" + mm + "-" + dd);

        }

      });

      var vm = new Vue({

        el: "#app",

        data: {

          date: new Date(),

        },

      });

    </script>

  </body>

### （8）生命週期





上面的生命週期函數又叫“鉤子”

重點關注mounted。這個函數一旦被觸發，表示頁面完成了初始化，頁面的模板內容存在，可以向模板裡填充數據了。

  <body>

    <div id="app">

      <div>{{msg}}</div>

      <button @click="update">更新</button>

      <button @click="destroy">銷毀</button>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          msg: "生命週期",

        },

        methods: {

          update: function () {

            this.msg = "hello";

          },

          destroy: function () {

            this.$destroy();

          },

        },

        beforeCreate: function () {

          console.log("beforeCreate");

        },

        created: function () {

          console.log("created");

        },

        beforeMount: function () {

          console.log("beforeMount");

        },

        mounted: function () {

          console.log("mounted");

        },

        beforeUpdate: function () {

          console.log("beforeUpdate");

        },

        updated: function () {

          console.log("updated");

        },

        beforeDestroy: function () {

          console.log("beforeDestroy");

        },

        destroyed: function () {

          console.log("destroyed");

        },

      });

    </script>

  </body>

## 6 綜合案例

### （1）數組相關API

#### ① 變異方法和替換數組

變異方法：數組的原生方法不是響應式的，Vue把原生的數組API處理成了響應式，這樣只要我們操作數組中的數據，頁面中的模板內容會隨之變化。

變異方法會修改原有數據

* push()
* pop()
* shift()
* unshift()
* splice()
* sort()
* reverse()

替換數組：不會影響原始數組，總是返回一個新數組。

* filter()
* concat()
* slice()

示例代碼

  <body>

    <div id="app">

      <div>

        <span>

          <input type="text" v-model="fname" />

          <button @click="add">添加</button>

          <button @click="del">刪除</button>

          <button @click="change">替換</button>

        </span>

      </div>

      <ul>

        <li :key="index" v-for="(item,index) in list">{{item}}</li>

      </ul>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          fname: "",

          list: ["apple", "orange", "banana"],

        },

        methods: {

          add: function () {

            this.list.push(this.fname);

          },

          del: function () {

            this.list.pop();

          },

          change: function () {

            this.list = this.list.slice(0, 2); //從0開始，截取2個

          },

        },

      });

    </script>

  </body>

#### ② 修改響應式數據

如下2個API既可以處理數組，也可以處理對象，都是響應式的。兩者形式不同，作用一樣。

Vue.set(vm.items, indxOfItem, newValue)

vm.$set(vm.items, indxOfItem, newValue)

參數1表示要處理的數字名稱

參數2表示要處理的數組的索引

參數3表示要處理的數組的值

  <body>

    <div id="app">

      <div>{{info.gender}}</div>

      <ul>

        <li :key="index" v-for="(item,index) in list">{{item}}</li>

      </ul>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var vm = new Vue({

        el: "#app",

        data: {

          fname: "",

          list: ["apple", "orange", "banana"],

          info: {

            name: "list",

            age: 12,

          },

        },

      });

      // vm.list[1]='lemon';//用索引修改了的數據不是響應式的，這樣修的話，雖然data有變化，但是頁面不會有變化。

      Vue.set(vm.list, 2, "lemon"); //這種方法可取，還可以用下面的方法

      // vm.$set(vm.list, 1, "lemon");

      vm.$set(vm.info, "gender", "female");

    </script>

  </body>

### （2）項目目標

1. 實現靜態列表效果
2. 基於數據實現模板效果
3. 處理每行的操作按鈕

### （3）添加圖書

1. 實現表單的靜態效果
2. 添加圖書表單域數據綁定
3. 添加按鈕事件綁定
4. 實現添加業務邏輯

### （4）修改圖書

1. 點擊修改按鈕，把圖書信息填充到input表單
2. 修改後提交表單（重複利用添加的方法）

注意this的指代

下面是Vue實例的methods中的一段代碼

addBook: function () {

            if (this.flag) {

              // 編輯圖書

              // 箭頭函數中的this指的是定義這個函數的父級作用域中（這裡是addBook的作用域）的this（addBook中的this指向的是Vue實例），而普通函數會指向window，所以這裡要用箭頭函數

              this.books.some((*item*) => {

                if (item.id === this.id) {

                  item.name = this.name;

                  this.flag = false;

                  this.id = "";

                  this.name = "";

                  // 完成更新操作後，即可終止循環，以提高性能。

                  return true;

                }

              });

            } else {

              // 添加圖書

              var book = {};

              book.id = this.id;

              book.name = this.name;

              book.date = "";

              this.books.push(book);

              // 清空表單

              this.id = "";

              this.name = "";

            }

          },

### （5）刪除圖書

1. 刪除按鈕綁定事件處理方法
2. 實現刪除業務邏輯

### （6）常用特性應用場景

1. 過濾器（格式化日期）
2. 自定義指令（獲取表單焦點）
3. 計算屬性（統計圖書數量）
4. 偵聽器（驗證圖書是否存在）
5. 生命週期（圖書數據處理）

## 7 Vue組件

### （1）組件化開發思想

* 標準
* 分治
* 重用
* 組合

組件規範化：Web Components

* 盡可能多重用代碼
* 自定義組件的方式不大容易（html，css和js）
* 多次使用組件可能導致衝突

Web Components通過創建封裝好功能的定制元素解決上述問題

官網：<https://developer.mozilla.org/zh-CN/docs/Web/Web_Components>

Vue部分實現了上述規範

### （2）組件註冊

#### ① 全局組件註冊語法

Vue.component(組件名稱, {

data: 組件數據,

template: 組件模板內容

})

組件可以多次使用

  <body>

    <div id="app">

      <button-counter></button-counter>

      <!-- 組件可以多次使用 -->

      <!-- 每个button-counter組件的數據都是相互獨立的 -->

      <button-counter></button-counter>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      // 定義一個名為button-counter的新組件

      Vue.component('button-counter',{

        data:function(){

          return{

            count:0

          }

        },

        // template:'<button v-on:click="count++">點擊了{{count}}次</button>'//这样写可以，或者把click后面换成函数

        template:'<button v-on:click="handle">點擊了{{count}}次</button>',

        methods:{

          handle:function(){

            this.count++;

          }

        }

      })

        // Vue實例vm本質上也是組件

      var vm = new Vue({

        el: "#app",

        data: {

          date: new Date(),

        },

      });

    </script>

  </body>

#### ② 組件註冊注意事項

1. data必須是一個函數 VS vue實例的data是個對象

函數會形成一個閉包的環境，這樣保證每個組件都擁有獨立的數據

1. 組件模板內容必須是單個根元素

上面的實例代碼中，template中的button，只能是1個，不能有同級節點。如果一定要加同級節點，必須在最外層包裹一個單個根元素（比如1個div）。

1. 組件模板內容可以是模板字符串

模板字符串需要瀏覽器提供支持（ES6語法）

        // template的內容如果比較複雜，可以換成模板字符串

        template:`

        <div>

          <button v-on:click="count++">點擊了{{count}}次</button><span>--可</span>

        </div>

        `,

1. 組件命名方式

* 短橫線方式（更推薦這種）

Vue.component(‘my-component’, { /\* … \*/ })

* 駝峰方式

駝峰方式命名的組件不能用在HTML頁面的body部分的普通標籤中，只能用於插入其它組件的字符串模板中

Vue.component(‘MyComponent’, { /\* … \*/ })

  <body>

    <div id="app">

      <button-counter></button-counter>

      <!-- 組件可以多次使用 -->

      <!-- 每个button-counter組件的數據都是相互獨立的 -->

      <button-counter></button-counter>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      // 定義一個名為button-counter的新組件

      Vue.component('button-counter',{

        data:function(){

          return{

            count:0

          }

        },

        // template的內容如果比較複雜，可以換成模板字符串

        template:`

        <div>

          <button v-on:click="count++">點擊了{{count}}次</button><span>--可</span>

          <HelloWorld></HelloWorld>

        </div>

        `,

        methods:{

          handle:function(){

            this.count++;

          }

        }

      })

      // 定義一個名為HelloWorld的組件

      Vue.component('HelloWorld', {

        data: function(){

          return {

            msg: 'HelloWorld'

          }

        },

        template:'<div>{{msg}}</div>'

      })

      var vm = new Vue({

        el: "#app",

        data: {

          date: new Date(),

        },

      });

    </script>

  </body>

#### ③ 局部組件註冊

局部組件只能在父組件中使用。自定義指令和自定義過濾器有類似的特性，也分為全局和局部兩種。

var ComponentA = { /\* … \*/ }

var ComponentB = { /\* … \*/ }

var ComponentC = { /\* … \*/ }

new Vue({

el: ‘#app’,

components: {

‘component-a’: ComponentA,

‘component-b’: ComponentB,

‘component-c’: ComponentC,

}

})

  <body>

    <div id="app">

      <hello-world></hello-world>

      <hello-linda></hello-linda>

      <hello-lizzy></hello-lizzy>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      var HelloWorld = {

        data: function(){

          return {

            msg: 'HelloWorld'

          }

        },

        template: '<div>{{msg}}</div>'

      }

      var HelloLinda = {

        data: function(){

          return {

            msg: 'HelloLinda'

          }

        },

        template: '<div>{{msg}}</div>'

      }

      var HelloLizzy = {

        data: function(){

          return {

            msg: 'HelloLizzy'

          }

        },

        template: '<div>{{msg}}</div>'

      }

      var vm = new Vue({

        el: "#app",

        data: {

        },

        components: {

        'hello-world': HelloWorld,

        'hello-linda': HelloLinda,

        'hello-lizzy': HelloLizzy,

        }

      });

    </script>

  </body>

### （3）Vue調試工具用法

安裝調試工具

在vue官網上面，“生態系統”的下拉菜單裡



點擊Devtools後，會進入一個GitHub頁面。

README.md區域的Documentation裡面有Chrome的Vue.js devtools插件的安裝步驟。



安裝好Chrome的Vue.js devtools後，在Chrome中進入插件的詳情頁





設置好下面兩個重要選項



然後，只要是Vue頁面，瀏覽器的URL右邊的這個圖標都會亮起來，比如bilibili，比如自己寫的vue頁面。



而對於自己有控制權的Vue頁面，F12後，Chrome的控制台裡會有vue的選項卡



### （4）組件間數據交互——父組件向子組件傳值

#### ① 基本用法

組件內部通過props接收傳遞過來的值

Vue.component(‘menu-item’, {

props:[‘title’],

template: ‘<div>{{ title }}</div>’

})

父組件通過屬性將值傳遞給子組件

<menu-item title=”來自父組件的數據”></menu-item> （值可以是固定的）

<menu-item :title=”title”></menu-item> （值也可以是動態綁定的）

  <body>

    <div id="app">

      <div>{{pmsg}}</div>

      <menu-item title="來自父組件"></menu-item>

      <menu-item :title="dongtai"></menu-item>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

Vue.component('menu-item',{

  props:['title'],

  data: function(){

    return {

      msg:'子組件本身的數據'

    }

  },

  template:'<div>{{msg + "-----" + title}}</div>'

})

      var vm = new Vue({

        el: "#app",

        data: {

          pmsg:'父組件中的內容',

          dongtai:'動態綁定屬性'

        },

      });

    </script>

  </body>

#### ② props屬性名規則

1. 在props中使用駝峰形式，模板中需要使用短橫線的形式（因為HTML中不區分大小寫）
2. 字符串形式的模板中沒有這個限制

Vue.component(‘menu-item’, {

// 在JavaScript中是駝峰式的

props: [‘menuTitle’],

template: <div>{{ menuTitle }}</div>

})

<!- 在html中是短橫線方式的 🡪

<menu-item menutitle=”nihao”></menu-item>

  <body>

    <div id="app">

      <div>{{pmsg}}</div>

      <menu-item :menu-title="dongtai"></menu-item>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      Vue.component('third-com',{

        props:['testTitle'],

        template:'<div>{{testTitle}}</div>'

      })

      Vue.component('menu-item',{

        props:['menuTitle'],

        template:'<div>{{menuTitle}} <third-com testTitle="hello">{{testTitle}}</third-com testTitle="hello"> </div>'

      })

      var vm = new Vue({

        el: "#app",

        data: {

          pmsg:'父組件中的內容',

          dongtai:'動態綁定屬性'

        },

      });

    </script>

  </body>

#### ③ props屬性值類型

* 字符串String
* 數值Number
* 布爾值Boolean
* 數組Array
* 對象Object

  <body>

    <div id="app">

      <div>{{pmsg}}</div>

      <!-- pnum前面加冒號，pnum是number；不加冒號，是string。同理，pboo前面加冒號，pboo是boolean；不加冒號，是string。parr和pobj前面加冒號，parr和pobj都是object，不加冒號，兩者都是string。 -->

      <menu-item :pstr="pstr" :pnum='12' :pboo='true' :parr='parr' :pobj='pobj'></menu-item>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      Vue.component('menu-item',{

        props:['pstr','pnum','pboo','parr','pobj'],

        template:`

          <div>

            <div>{{pstr}}</div>

            <div>{{'在pnum前面加了冒號，那麼pnum是'+ typeof pnum}}</div>

            <div>{{'在pboo前面加了冒號，那麼pboo是'+ typeof pboo}}</div>

            <ul>

              <li :key='index' v-for='(item, index) in parr'>{{item}}</li>

            </ul>

            <div>

              <span>{{ pobj.name }}</span>

              <span>{{ pobj.age }}</span>

            </div>

          </div>

        `

      })

      var vm = new Vue({

        el: "#app",

        data: {

          pmsg:'父組件中的內容',

          pstr:'hello',

          parr:['apple','orange','lemon','banana'],

          pobj: {

          name: 'lisi',

          age: 21

        }

        },

      });

    </script>

  </body>

### （5）組件間數據交互——子組件向父組件傳值

#### ① 子組件直接操控父組件裡面的值

#### ② 子組件通過自定義事件向父組件傳遞信息-基本用法

<button v-on:click=’$emit(“enlarge-text”)’>放大字體</button>

父組件監聽子組件的事件

<menu-item v-on:enlarge-text=’fontSize += 0.1’></menu-item>

  <body>

    <div id="app">

      <div :style='{fontSize: fontSize + "px"}'>{{pmsg}}</div>

      <menu-item :parr='parr' @enlarge-text='handle'></menu-item>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

     // 在Vue中，子組件不要直接通過props操作父組件的數據，因為Vue是單向數據流。

      Vue.component('menu-item',{

        props:['pstr','pnum','pboo','parr','pobj'],

        template:`

          <div>

            <button @click='parr.push("lemon")'>點擊添加lemon</button>

            <button v-on:click='$emit("enlarge-text")'>放大字體</button>

            <ul>

              <li :key='index' v-for='(item, index) in parr'>{{item}}</li>

            </ul>

          </div>

        `

      })

      var vm = new Vue({

        el: "#app",

        data: {

          pmsg:'父組件中的內容',

          parr:['apple','orange','lemon','banana'],

          fontSize:10

        },

        methods:{

          handle: function(){

          // 放大字體

          this.fontSize +=5;

        }

        }

      });

    </script>

  </body>

#### ③ 子組件通過自定義事件向父組件傳遞信息-攜帶參數

<button v-on:click=’$emit(“enlarge-text”, 0.1)’>放大字體</button>

父組件監聽子組件的事件

<menu-item v-on:enlarge-text=’fontSize += $event’></menu-item>

  <body>

    <div id="app">

      <div :style='{fontSize: fontSize + "px"}'>{{pmsg}}</div>

      <menu-item :parr='parr' @enlarge-text='handle($event)'></menu-item>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

     // 在Vue中，子組件不要直接通過props操作父組件的數據，因為Vue是單向數據流。

      Vue.component('menu-item',{

        props:['pstr','pnum','pboo','parr','pobj'],

        template:`

          <div>

            <button @click='parr.push("lemon")'>點擊添加lemon</button>

            <button v-on:click='$emit("enlarge-text", 5)'>放大字體</button>

            <ul>

              <li :key='index' v-for='(item, index) in parr'>{{item}}</li>

            </ul>

          </div>

        `

      })

      var vm = new Vue({

        el: "#app",

        data: {

          pmsg:'父組件中的內容',

          parr:['apple','orange','lemon','banana'],

          fontSize:10

        },

        methods:{

          handle: function(*val*){

          // 放大字體

          this.fontSize += val;

        }

        }

      });

    </script>

  </body>

### （6）組件間數據交互——同級別組件傳值

原理：



單獨的事件中心管理組件間的通信

var eventHub = new Vue()

監聽事件與銷毀事件

eventHub.$on(‘add-todo’, addTodo)

eventHub.$off(‘add-todo’)

觸發事件（可以攜帶參數，比如下面的id）

eventHub.$emit(‘add-todo’, id)

  <body>

    <div id="app">

      <div>父組件</div>

      <div><button @click='handle'>銷毀事件</button></div>

      <test-tom></test-tom>

      <test-jerry></test-jerry>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

    // 提供事件中心

    var hub = new Vue();

    // 子組件test-tom

      Vue.component('test-tom',{

        data: function(){

          return{

            num:0

          }

        },

        template:`

          <div>

            <div>TOM:{{num}}</div>

            <div>

              <button @click='handle'>點擊</button>

            </div>

          </div>

        `,

        methods: {

          handle: function(){

            // 觸發同級組件jerry的監聽事件

            hub.$emit('jerry-event',2);

          }

        },

        mounted: function(){

          // 監聽事件

          hub.$on('tom-event',(*val*)=>{

            this.num += val;

          })

        }

      })

    // 子組件test-jerry

      Vue.component('test-jerry',{

        data: function(){

          return{

            num:0

          }

        },

        template:`

          <div>

            <div>Jerry:{{num}}</div>

            <div>

              <button @click='handle'>點擊</button>

            </div>

          </div>

        `,

        methods: {

          handle: function(){

            // 觸發同級組件tom的監聽事件

            hub.$emit('tom-event',1);

          }

        },

        mounted: function(){

          // 監聽事件

          hub.$on('jerry-event',(*val*)=>{

            this.num += val;

          })

        }

      })

      var vm = new Vue({

        el: "#app",

        data: {},

        methods:{

          handle:function(){

            hub.$off('tom-event');

            hub.$off('jerry-event');

          }

        }

      });

    </script>

  </body>

### （7）組件插槽

#### ① 基本用法

父組件向子組件傳遞模板的內容

插槽位於子組件的模板中，下面的slot標籤

Vue.component(‘alert-box’, {

template:`

<div class=”demo-alert-box”>

<strong>Error!</strong>

<slot></slot>

</div>

`

})

插槽內容

<alert-box>There is an error.</alert-box>

如果slot標籤中有默認內容，而父組件沒有給這個插槽傳遞內容，就會顯示slot中的默認內容；否則顯示父組件傳遞過來的內容。

  <body>

    <div id="app">

      <div>父組件</div>

      <alert-box>有個bug</alert-box>

      <alert-box>有個警告</alert-box>

      <alert-box></alert-box>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      // 子組件

      Vue.component('alert-box', {

      template:`

        <div class=”demo-alert-box”>

          <strong>Error!</strong>

          <slot>默認內容</slot>

        </div>

        `

      })

      var vm = new Vue({

        el: "#app",

        data: {}

      });

    </script>

  </body>

#### ② 具名插槽

定義插槽

<div class=”container”>

<header>

<slot name=”header”></slot>

</header>

<main>

<slot></slot>

</main>

<footer>

<slot name=”footer”></slot>

</footer>

</div>

插槽內容

<base-layout>

<h1 slot=”header”>標題內容</h1> （這裡會填充到上面name=”header”的slot中）

<p>主要內容1</p>

<p>主要內容2</p>

<p slot=”footer”>底部內容</p>（這裡會填充到上面name=”footer”的slot中）

</base-layout>





  <body>

    <div id="app">

      <base-layout>

        <!-- 這裡會填充到name="header"的slot中 -->

          <h1 slot="header">標題內容</h1>

        <p>主要內容1</p>

        <p>主要內容2</p>

      <!-- 這裡會填充到name="footer"的slot中 -->

        <p slot="footer">底部內容</p>

      </base-layout>

      <div>——————分割線——————</div>

      <base-layout>

      <!-- template標籤是Vue特有的API標籤，作用是臨時包裹它裡面的內容。template不會被渲染到頁面。它的作用是把被包裹的內容都放入相關的slot中 -->

        <template slot="header">

          <h1 >標題內容1</h1>

          <h1 >標題內容2</h1>

        </template>

        <p>主要內容1</p>

        <p>主要內容2</p>

        <template slot="footer">

          <p >底部內容1</p>

          <p >底部內容2</p>

        </template>

      </base-layout>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      // 子組件

      Vue.component('base-layout', {

      template:`

        <div class="container">

          <header>

            <slot name='header'></slot>

          </header>

          <main>

            <slot></slot>

          </main>

          <footer>

            <slot name='footer'></slot>

          </footer>

        </div>

        `

      })

      var vm = new Vue({

        el: "#app",

        data: {}

      });

    </script>

  </body>

#### ③ 作用域插槽

父組件對子組件的內容進行加工處理

## 8 基於組件的案例——購物車案例

### （1）根據業務功能進行組件劃分

1. 標題組件（展示文本）
2. 列表組件（列表展示，商品數量變更，刪除商品）
3. 結算組件（計算商品總額）

CartList更新商品數量時，不要用v-model。因為CartList中的值是props接收的父組件傳過來的值。一般不要去修改props中的值。在methods中設置的changeNum，把修改後的值傳遞給my-cart父組件，在父組件中完成值的修改。

Vue全家桶-前後端交互

## 1 前後端交互模式

### （1）數據的流轉

前端申請後端數據，後端返回相關數據給前端後，前端把數據渲染到頁面上。



前端調用接口請求後端數據的方式

* 原生ajax
* 基於jQuery的ajax
* fetch
* axios

### （2）傳統形式的URL

格式 schema://host:port/path?query#fragment

* schema：協議。例如http、https、ftp等
* host：域名或者IP地址
* port：端口，http默認端口80，可以省略
* path：路徑，例如/abc/a/b/c
* query：查詢參數，例如uname=lisi&age=12
* fragment：錨點（哈希Hash），用於定位頁面的某個位置

傳統格式URL例子：

* <http://www.itcast.cn>
* <http://www.itcast.cn/java/web>
* <http://www.itcast.cn/java/web?flag=1>
* <http://www.itcast.cn/java/web?flag=1#function>

### （3）Restful形式的URL

HTTP請求方式：

* GET 查詢
* POST 添加
* PUT 修改
* DELETE 刪除

Restful格式URL例子：

* <http://www.hello.com/books> GET方式提交
* <http://www.hello.com/books> POST方式提交
* <http://www.hello.com/books/123> PUT方式提交
* <http://www.hello.com/books/123> DELETE方式提交

Restful的傳參方式是直接在URL地址中寫斜杠/+參數。

傳統URL地址中是問號+參數來傳參。

## 2 Promise用法

Promise是es6引入的新語法，專門用來處理異步編程（ajax也是異步編程）。

### （1）異步調用

異步函數：

* 定時任務（setTimeout和setInterva）
* Ajax
* 事件函數

異步函數特性：

* 多次異步調用的結果順序不確定。
* 比如有多個ajax，每個ajax有success回調函數，這些回調函數的執行先後不確定，取決於請求結果返回的快慢，而不取決於ajax請求代碼的先後順序。
* 異步調用結果如果存在醫療關係，需要嵌套。
* 比如有2個ajax，第2個ajax必須在第1個ajax拿到返回數據後才發送請求，第2個ajax要寫在第1個ajax的success函數裡面。
* 上述例子存在一個問題，如果這樣在回調函數success中嵌套的層次特別多，代碼就很亂，有個詞描述這種狀況叫“回調地獄”。為了解決這個問題，Promise應運而生。

### （2）Promise概述

Promise是異步編程的一種解決方案，從語法上講，Promise是一個對象，從它可以獲取異步操作的消息。

使用Promise的好處：

* 避免多層回調函數異步調用的嵌套問題（避免“回調地獄”）
* Promise對象提供了簡介的API，使得控制異步操作更加容易

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise>

### （3）Promise基本用法

* 實例化Promise對象，構造函數中傳遞函數，該函數中處理異步任務。
* resolve和reject兩個參數用於處理成功和失敗兩種情況，並通過p.then獲取處理結果。

語法：

      var p = new Promise(function (*resolve*, *reject*) {

        // 成功時調用 resolve()

        // 失敗時調用 reject()

      });

      p.then(

        function (*ret*) {

          // 從resolve得到正常結果

        },

        function (*ret*) {

          // 從reject得到錯誤信息

        }

      );

示例代碼：

      var p = new Promise(function (*resolve*, *reject*) {

        // 實現異步任務

        setTimeout(function () {

          // var flag = true;

          var flag = false;

          if (flag) {

            // 正常情況

            resolve("right");

          } else {

            // 異常情況

            reject("wrong");

          }

        });

      });

      p.then(

        function (*data*) {

          // 從resolve得到正常結果

          console.log(data);

        },

        function (*info*) {

          // 從reject得到錯誤信息

          console.log(info);

        }

      );

### （4）Promise發送ajax請求並處理回調地獄問題

      // 基於Promise發送ajax請求

      function queryData(*url*) {

        var p = new Promise(function (*resolve*, *reject*) {

          var xhr = new XMLHttpRequest();

          xhr.onreadystatechange = function () {

            if (xhr.readyState != 4) return;

            if (xhr.readyState == 4 && xhr.status == 200) {

              // 處理正常的情況

              resolve(xhr.responseText);

            } else {

              // 處理異常情況

              reject("服務器錯誤");

            }

          };

          xhr.open("get", url);

          xhr.send(null);

        });

        return p;

      }

      queryData("http://localhost:3000/data").then(

        function (*data*) {

          console.log(data);

        },

        function (*info*) {

          console.log(info);

        }

      );

發送多個ajax請求並且保證順序

      // 基於Promise發送ajax請求

      function queryData(*url*) {

        var p = new Promise(function (*resolve*, *reject*) {

          var xhr = new XMLHttpRequest();

          xhr.onreadystatechange = function () {

            if (xhr.readyState != 4) return;

            if (xhr.readyState == 4 && xhr.status == 200) {

              // 處理正常的情況

              resolve(xhr.responseText);

            } else {

              // 處理異常情況

              reject("服務器錯誤");

            }

          };

          xhr.open("get", url);

          xhr.send(null);

        });

        return p;

      }

      queryData("http://localhost:3000/data")

        .then(function (*data*) {

          console.log(data);

          return queryData("http://localhost:3000/data1");

        })

        .then(function (*data*) {

          console.log(data);

          return queryData("http://localhost:3000/data2");

        })

        .then(function (*data*) {

          console.log(data);

        });

上面代碼中，return的是一個新的promise對象，調用then的，就是返回的promise對象。

### （5）Promise的then方法參數中的函數的返回值

#### ① 返回Promise實例對象

返回的該實例對象會調用下一個then

#### ② 返回普通值

返回的普通值會直接傳遞給下一個then，通過then參數中函數的參數接受該值

      // 基於Promise發送ajax請求

      function queryData(*url*) {

        return new Promise(function (*resolve*, *reject*) {

          var xhr = new XMLHttpRequest();

          xhr.onreadystatechange = function () {

            if (xhr.readyState != 4) return;

            if (xhr.readyState == 4 && xhr.status == 200) {

              // 處理正常的情況

              resolve(xhr.responseText);

            } else {

              // 處理異常情況

              reject("服務器錯誤");

            }

          };

          xhr.open("get", url);

          xhr.send(null);

        });

      }

      queryData("http://localhost:3000/data")

        .then(function (*data*) {

          console.log(data);

          return queryData("http://localhost:3000/data1");

        })

        .then(function (*data*) {

          console.log(data); //這是上一個then return回來的promise對象

          return new Promise(function (*resolve*, *reject*) {

            setTimeout(function () {

              resolve(123);

            }, 1000);

          });

        })

        .then(function (*data*) {

          console.log(data); //123

          return "hello"; // 如果這裡返回的是個普通值，這個then會產生一個默認的promise對象來調用下一個then

        })

        .then(function (*data*) {

          console.log(data); //hello

        });

### （6）Promise常用API-實例方法

這些位於函數原型（prototype）中：

* p.then() 得到異步任務的正確結果
* p.catch() 獲取異常信息
* p.finally() 成功與否都會執行（尚未稱為正式標準）

位於函數原型（prototype）中的方法都是實例方法。

語法：

queryData()

.then(function(data){

console.log(data);

})

.catch(function(data){

console.log(data);

})

.finally(function(){

console.log(‘finished’);

});

示例代碼：

      // Promise常用API實例方法

     function foo() {

       return new Promise(function(*resolve*, *reject*){

         setTimeout(function(){

          //  resolve(123);

          reject('error');

         },100);

       })

     }

     foo()

      .then(function(*data*){

        console.log(data); //123

      })

      .catch(function(*data*){

        console.log(data) //error

      })

      .finally(function(){

        console.log('finished') //finished

      });

示例代碼2：

下面的then的第2個參數（函數），相當於上面的catch

      // Promise常用API實例方法

     function foo() {

       return new Promise(function(*resolve*, *reject*){

         setTimeout(function(){

          //  resolve(123);

          reject('error');

         },100);

       })

     }

     foo()

      .then(function(*data*){

        console.log(data); //123

      },function(*data*){

        console.log(data); //error

      })

      .finally(function(){

        console.log('finished') //finished

      });

### （7）Promise常用API-對象方法

Promise.all() 並發處理多個異步任務，所有任務都執行完成才能得到結果。

Promise.race() 並發處理多個異步任務，只要有一個任務完成就能得到結果。

語法（傳入的參數是個數組，其中p1，p2，p3都是實例對象。每一個實例對象完成一個異步操作。）：

Promise.all([p1, p2, p3]).then((result) => {

console.log(result)

})

Promise.race([p1, p2, p3]).then((result) => {

console.log(result)

})

前端HTML頁面body中的語法

  <body>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script type="text/javascript">

      // Promise常用API對象方法

      function queryData(*url*){

        return new Promise(function(*resolve*, *reject*){

          var xhr = new XMLHttpRequest();

          xhr.onreadystatechange = function(){

            if(xhr.readyState !=4) return;

            if(xhr.readyState ==4 && xhr.status  ==200) {

              // 處理正常的情況

              resolve(xhr.responseText);

            }else{

              // 處理異常情況

              reject('服務器錯誤')

            }

          }

          xhr.open('get',url);

          xhr.send(null);

        });

      }

      var p1 = queryData('http://localhost:3000/a1')

      var p2 = queryData('http://localhost:3000/a2')

      var p3 = queryData('http://localhost:3000/a3')

      Promise.all([p1,p2,p3]).then(function(*result*){

        console.log(result);

      })

    </script>

  </body>

後台index.js中的語法

const express = require('express')

const app = express()

// 配置解析 application/json 格式數據的內置中間件

app.use(express.json())

// 配置解析 application/x-www.form-urlencoded格式數據的內置中間件

app.use(express.urlencoded({ extended: false }))

// 設置允許跨域訪問該服務

app.all('\*', function (*req*, *res*, *next*) {

  res.header("Access-Control-Allow-Origin", "\*");

  res.header('Access-Control-Allow-Methods', 'PUT, GET, POST, DELETE, OPTIONS');

  next();

});

// 路由

app.get('/a1', (*req*, *res*) => {

  setTimeout(function () {

    res.send('Hello a1')

  }, 500);

})

app.get('/a2', (*req*, *res*) => {

  setTimeout(function () {

    res.send('Hello a2')

  }, 1000);

})

app.get('/a3', (*req*, *res*) => {

  setTimeout(function () {

    res.send('Hello a3')

  }, 1500);

})

app.listen(3000, () => {

  console.log('express server running at http://127.0.0.1:3000')

})

Console控制台打印的數據：



## 3 接口調用-fetch用法

### （1）fetch基本用法

fetch是window的一個方法，用來請求數據的。

ajax和fetch的區別 ：

1. ajax是利用瀏覽器提供的JavaScript對象XMLHttpRequest（簡稱XHR）來請求數據，而fetch是window的一個方法。
2. fetch可以看作是xhr的升級版，fetch基於Promise實現
3. fetch比ajax有著更方便的寫法。
4. fetch只對網絡請求報錯，對400，500都當做成功的請求，需要做相關封裝處理。
5. fetch沒有辦法監測請求的進度，而XHR可以。

語法結構：

fetch(url).then(fn2)

.then(fn3)

…

.catch(fn)

基本用法

  <script>

    fetch('/abc').then(*data*=>{

      return data.text(); //text()屬於fetchAPI的一部分，data.text()的返回值是個promise實例對象，用於獲取後台返回的數據。

    }).then(*ret*=>{

      // 注意這裡得到的才是最終的數據

      console.log(ret);

    })

  </script>

Fetch API網址：<https://developer.mozilla.org/en-US/docs/web/api/fetch_api>

### （2）常用配置選項

* method(String)：HTTP請求方法有GET，POST，PUT，DELETE，默認為GET。
* body(String)：HTTP的請求參數
* headers(Object)：HTTP的請求頭，默認為{}；必須設置，否則body中的數據無法被傳遞到服務器。

範例代碼

前端fetch.html頁面中的JavaScript代碼

<body>

  <script>

    // 1 傳統的GET傳參方式

    fetch('http://localhost:3000/conventional-get?id=1',{

      method:'get' //如果是get請求，這裡method可以不寫

    }).then(function(*data*){

      return data.text();

    }).then(function(*data*){

      console.log(data)

    })

    // 2 Restful的GET傳參方式

    fetch('http://localhost:3000/restful-get/2',{

      method:'get'

    }).then(function(*data*){

      return data.text();

    }).then(function(*data*){

      console.log(data)

    })

    // 3 Restful的DELETE傳參方式

    fetch('http://localhost:3000/restful-delete/3',{

      method:'delete'

    }).then(function(*data*){

      return data.text();

    }).then(function(*data*){

      console.log(data)

    })

    // 4-1 Restful的POST傳參方式x-www-form-urlencoded

    fetch('http://localhost:3000/restful-post-form',{

      method:'post',

      body:'pwd=4-1&uname=lisi',

      headers:{

        'Content-Type':'application/x-www-form-urlencoded'

      }

    }).then(function(*data*){

      return data.text();

    }).then(function(*data*){

      console.log(data)

    })

    // 4-2 Restful的POST傳參方式JSON

    fetch('http://localhost:3000/restful-post-json',{

      method:'post',

      body:JSON.stringify({

        uname:'lisi',

        pwd:'4-2'

      }),

      headers:{

        'Content-Type':'application/json'

      }

    }).then(function(*data*){

      return data.text();

    }).then(function(*data*){

      console.log(data)

    })

    // 5 Restful的PUT傳參方式JSON，用於修改數據

    fetch('http://localhost:3000/restful-put-json/5',{

      method:'put',

      body:JSON.stringify({

        uname:'wangwu',

        pwd:'5demima'

      }),

      headers:{

        'Content-Type':'application/json'

      }

    }).then(function(*data*){

      return data.text();

    }).then(function(*data*){

      console.log(data)

    })

  </script>

</body>

後台的JavaScript代碼：

const express = require('express')

const app = express()

// 配置解析 application/json 格式數據的內置中間件

app.use(express.json())

// 配置解析 application/x-www.form-urlencoded格式數據的內置中間件

app.use(express.urlencoded({ extended: false }))

// 設置允許跨域訪問該服務

app.all('\*', function (*req*, *res*, *next*) {

  res.header("Access-Control-Allow-Origin", "\*");

  res.header('Access-Control-Allow-Headers', 'Content-Type');

  res.header('Access-Control-Allow-Methods', 'PUT, GET, POST, DELETE, OPTIONS');

  next();

});

// 路由

app.get('/conventional-get', (*req*, *res*) => {

  setTimeout(function () {

    res.send('傳統URL-GET傳遞參數' + req.query.id)

  }, 500);

})

app.get('/restful-get/:id', (*req*, *res*) => {

  setTimeout(function () {

    res.send('Restful-URL-GET傳遞參數' + req.params.id)

  }, 500);

})

app.delete('/restful-delete/:id', (*req*, *res*) => {

  setTimeout(function () {

    res.send('Restful-URL-DELETE傳遞參數' + req.params.id)

  }, 500);

})

app.post('/restful-post-form', (*req*, *res*) => {

  setTimeout(function () {

    res.send('Restful-POST傳遞參數，格式x-www-form-urlencoded：' + req.body.uname + '---' + req.body.pwd)

  }, 500);

})

app.post('/restful-post-json', (*req*, *res*) => {

  setTimeout(function () {

    res.send('Restful-POST傳遞參數，格式json：' + req.body.uname + '---' + req.body.pwd)

  }, 500);

})

app.put('/restful-put-json/:id', (*req*, *res*) => {

  setTimeout(function () {

    res.send('Restful-PUT傳遞參數，格式json：' + req.body.uname + '---' + req.body.pwd + '---' + req.params.id)

  }, 500);

})

app.listen(3000, () => {

  console.log('express server running at http://127.0.0.1:3000')

})

### （3）fetch回來的數據的格式處理

可以用text()：將返回體處理成字符串類型。

也可以用json()：將返回體處理成JSON格式，結果和JSON.parse(responseText)。

用json()更方便。

範例代碼：

  <script>

    fetch('http://localhost:3000/get-json',{

    }).then(function(*data*){

      return data.json();

    }).then(function(*data*){

      console.log(typeof data) //object

    })

    fetch('http://localhost:3000/get-json',{

    }).then(function(*data*){

      return data.text();

    }).then(function(*data*){

      console.log(typeof data) //string。如果要獲取其中的數據，必須把字符串轉換為JSON格式

      var objData = JSON.parse(data)

      console.log(typeof objData) //object

    })

  </script>

## 4 接口調用-axios用法

### （1）axios基本用法

axios是第三方庫，用於調用接口。

axios基於promise，可用於瀏覽器和node.js，能攔截請求和響應，自動轉換JSON數據。

官網：<https://github.com/axios/axios>

基本用法：

    <script>

      axios.get("/adata").then((*ret*) => {

        // data屬性名稱是固定的，用於獲取後台響應的數據

        console.log(ret.data);

      });

    </script>

使用axios前，要引入axios。先去官網下載zip壓縮包，解壓後，在這個文件夾裡axios-master\dist，根據情況引入下面兩個文件。



### （2）axios常用API

* get 查詢數據
  + get傳遞參數：①通過URL，②通過params
* post 添加數據
* put 修改數據
* delete 刪除數據

範例代碼

前端代碼

  <body>

    <script type="text/javascript" src="/axios.js"></script>

    <script type="text/javascript">

      // 1-1 get 傳統url 問號傳參

      axios.get("http://localhost:3000/get?id=1-1").then((*ret*) => {

        console.log(ret.data);

      });

      // 1-2 get Restful url傳參

      axios.get("http://localhost:3000/gett/1-2").then((*ret*) => {

        console.log(ret.data);

      });

      // 1-3 get params屬性傳參 （三種get，這種方式最方便）

      axios

        .get("http://localhost:3000/gettt", {

          params: {

            id: "1-3",

          },

        })

        .then((*ret*) => {

          console.log(ret.data);

        });

      // 2 delete的傳參方式和get一樣，有上面3種

      // 3 post有2種傳參方式，具體用那種，取決於後台支持那種。如果後台兩種都支持，則用JSON更方便。

      // 3-1 post通過選項傳遞參數，默認傳遞的是JSON格式的數據

      axios

        .post("http://localhost:3000/post", {

          uname: "zs",

          pwd: "3-1",

        })

        .then((*ret*) => {

          console.log(ret.data);

        });

      // 3-2 post通過URLSearchParams傳遞參數（application/x-www-form-urlencoded）

      const params = new URLSearchParams();

      // params.append("param1", "value1");

      // params.append("param2", "value2");

      params.append("aname", "linjiangxian");

      params.append("pwd", "3-2");

      axios.post("http://localhost:3000/postt", params).then((*ret*) => {

        console.log(ret.data);

      });

      // 4 put的傳參方式和post類似，有上面2種。注意，與post不同的是，put一般會在url中攜帶1個參數

      // 4-1 put通過選項傳遞參數，默認傳遞的是JSON格式的數據

      axios

        .post("http://localhost:3000/put/4-1", {

          uname: "zs",

          pwd: "put",

        })

        .then((*ret*) => {

          console.log(ret.data);

        });

      // 4-2 put通過URLSearchParams傳遞參數（application/x-www-form-urlencoded）

      const paramsPut = new URLSearchParams();

      // params.append("param1", "value1");

      // params.append("param2", "value2");

      paramsPut.append("aname", "dianjiangchun");

      paramsPut.append("pwd", "putt");

      axios.post("http://localhost:3000/putt/4-2", paramsPut).then((*ret*) => {

        console.log(ret.data);

      });

      axios

        .post("http://localhost:3000/put/4-1", {

          uname: "zs",

          pwd: "put",

        })

        .then((*ret*) => {

          console.log(ret.data);

        });

    </script>

  </body>

後台代碼

const express = require('express')

const app = express()

// 配置解析 application/json 格式數據的內置中間件

app.use(express.json())

// 配置解析 application/x-www.form-urlencoded格式數據的內置中間件

app.use(express.urlencoded({ extended: false }))

// 設置允許跨域訪問該服務

app.all('\*', function (*req*, *res*, *next*) {

  res.header("Access-Control-Allow-Origin", "\*");

  res.header('Access-Control-Allow-Headers', 'Content-Type');

  res.header('Access-Control-Allow-Methods', 'PUT, GET, POST, DELETE, OPTIONS');

  next();

});

// 路由

app.get('/get', (*req*, *res*) => {

  res.send('1-1 axios get 傳統url問號傳參：' + req.query.id)

})

app.get('/gett/:id', (*req*, *res*) => {

  res.send('1-2 axios get Restful url傳參：' + req.params.id)

})

app.get('/gettt', (*req*, *res*) => {

  res.send('1-3 axios get params屬性傳參：' + req.query.id)

})

app.post('/post', (*req*, *res*) => {

  res.send('3-1 axios post json傳參：' + req.body.uname + req.body.pwd)

})

app.post('/postt', (*req*, *res*) => {

  res.send('3-2 axios post URLSearchParams傳參：' + req.body.aname + req.body.pwd)

})

app.post('/put/:id', (*req*, *res*) => {

  res.send('4-1 axios put json傳參：' + req.body.uname + req.body.pwd + req.params.id)

})

app.post('/putt/:id', (*req*, *res*) => {

  res.send('4-2 axios put URLSearchParams傳參：' + req.body.aname + req.body.pwd + req.params.id)

})

app.listen(3000, () => {

  console.log('express server running at http://127.0.0.1:3000')

})

### （3）axios響應結果

響應結果的主要屬性：

* data：實際響應回來的數據（絕大多數情況下是JSON）
* headers：響應頭信息
* status：響應狀態碼
  + 如果status===200，狀態正常。
* statusText：響應狀態信息

後台可以用json()返回JSON數據

app.get('/json', (*req*, *res*) => {

  res.json({

    uname: 'qaz',

    age: 12

  })

})

### （4）axios的全局配置

axios.defaults.timeout = 3000; //超時時間

axios.defaults.baseURL = ‘http://localhost:3000/app’; //默認地址

axios.defaults.headers[ ‘mytoken’ ] = ‘d2kjklj546kl234kjlk5645kj234lkj567’ //設置請求頭

前端代碼

    <script type="text/javascript">

      axios.defaults.baseURL = "http://localhost:3000/";

      // 配置請求頭

      axios.defaults.headers["mytoken"] = "kna2s7d";

      axios.get("json").then((*ret*) => {

        console.log(ret.data);

      });

    </script>

前端代碼裡在headers中發送了”mytoken”字段，那麼後台要允許前端在headers中發送“mytoken”這個字段，否則，會觸發跨域問題：

Access to XMLHttpRequest at 'http://localhost:3000/json' from origin 'http://localhost:8080' has been blocked by CORS policy: Request header field mytoken is not allowed by Access-Control-Allow-Headers in preflight response.



### （5）axios攔截器

#### ① 請求攔截器

在請求發出之前設置一些信息

前端

    <script type="text/javascript" src="/axios.js"></script>

    <script type="text/javascript">

      axios.interceptors.request.use(

        function (*config*) {

          console.log(config.url);

          // 一個應用場景：可以根據url來決定是否設置token

          config.headers.mytoken = "nihao";

          return config;

        },

        function (*err*) {

          console.log(err);

        }

      );

      axios.get("http://localhost:3000/adata").then(function (*res*) {

        console.log(res.data);

      });

    </script>

後台

const express = require('express')

const app = express()

// 配置解析 application/json 格式數據的內置中間件

app.use(express.json())

// 配置解析 application/x-www.form-urlencoded格式數據的內置中間件

app.use(express.urlencoded({ extended: false }))

// 設置允許跨域訪問該服務

app.all('\*', function (*req*, *res*, *next*) {

  res.header("Access-Control-Allow-Origin", "\*");

  res.header('Access-Control-Allow-Headers', 'Content-Type');

  res.header('Access-Control-Allow-Headers', 'mytoken');

  res.header('Access-Control-Allow-Methods', 'PUT, GET, POST, DELETE, OPTIONS');

  next();

});

// 路由

app.get('/adata', (*req*, *res*) => {

  res.json({

    uname: 'qaz',

    age: 12

  })

})

app.listen(3000, () => {

  console.log('express server running at http://127.0.0.1:3000')

})

#### ② 響應攔截器

在獲取數據之前對數據做一些加工處理

前端代碼

    <script type="text/javascript" src="/axios.js"></script>

    <script type="text/javascript">

      axios.interceptors.response.use(

        function (*res*) {

          var data = res.data; //1 把res中的data取出來

          return data; //2 然後返回

        },

        function (*err*) {

          console.log(err);

        }

      );

      axios.get("http://localhost:3000/adata").then(function (*data*) {

        console.log(data); // 3 這裡接收到的是上面返回的data

      });

    </script>

## 5 接口調用-async/await用法

### （1） async/await基本用法

async/ await是ES7引入的新語法，可以更加方便地進行異步操作

async關鍵字用於函數上（async函數的返回值是Promise實例對象）

await關鍵字用於async函數內（await可以得到異步的結果）

    <script type="text/javascript" src="/axios.js"></script>

    <script type="text/javascript">

      async function queryData() {

        const ret = await axios.get("http://localhost:3000/adata");// 有了await，axios不再需要then和回調函數就能直接拿到返回值，代碼更簡潔。

        // console.log(ret);

        return ret.data;

      }

      queryData().then(function (*data*) {

        console.log(data);

      });

    </script>

await後面可以跟一個promise實例對象，得到異步結果。

    <script type="text/javascript" src="/axios.js"></script>

    <script type="text/javascript">

      async function queryData() {

        const ret = await new Promise(function (*resolve*, *reject*) {

          setTimeout(function () {

            resolve("nihao");

          }, 500);

        });

        return ret;

      }

      queryData().then(function (*data*) {

        console.log(data);

      });

    </script>

### （2） async/await處理多個異步請求

前端

    <script type="text/javascript" src="/axios.js"></script>

    <script type="text/javascript">

      axios.defaults.baseURL = "http://localhost:3000";

      async function queryData() {

        const info = await axios.get("async1");

        const ret = await axios.get("async2?info=" + info.data);

        return ret.data;

      }

      queryData().then(function (*info*) {

        console.log(info);

      });

    </script>

後台

const express = require('express')

const app = express()

// 配置解析 application/json 格式數據的內置中間件

app.use(express.json())

// 配置解析 application/x-www.form-urlencoded格式數據的內置中間件

app.use(express.urlencoded({ extended: false }))

// 設置允許跨域訪問該服務

app.all('\*', function (*req*, *res*, *next*) {

  res.header("Access-Control-Allow-Origin", "\*");

  res.header('Access-Control-Allow-Headers', 'Content-Type');

  res.header('Access-Control-Allow-Headers', 'mytoken');

  res.header('Access-Control-Allow-Methods', 'PUT, GET, POST, DELETE, OPTIONS');

  next();

});

// 路由

app.get('/async1', (*req*, *res*) => {

  res.send('1')

})

app.get('/async2', (*req*, *res*) => {

  res.json({

    msg: 'async2传来的信息',

    info: '请求时的info--' + req.query.info

  })

})

app.listen(3000, () => {

  console.log('express server running at http://127.0.0.1:3000')

})

## 6 基於接口的案例-圖書管理項目

### （1）案例分析

圖書相關的操作基於後台接口數據進行

需要調用接口的功能點

|  |  |  |
| --- | --- | --- |
| 1 圖書列表數據加載 | GET | http://localhost:3000/books |
| 2 驗證圖書id是否存在 | GET | http://localhost:3000/books/bookId/:id |
| 3 驗證圖書名稱是否存在 | GET | http://localhost:3000/books/book/:name |
| 4 添加圖書 | POST | http://localhost:3000/books |
| 5 編輯圖書-提交圖書信息 | PUT | http://localhost:3000/books/bookId/:id |
| 6 刪除圖書 | DELETE | http://localhost:3000/books/bookId/:id |

Vue全家桶-前端路由

## 1 路由的基本概念與原理

路由的本質是對應關係。

路由分為後端路由和前端路由。

### （1）後端路由

概念：根據不同的用戶URL請求，返回不同的內容

本質：URL請求地址與服務器資源之間的對應關係

### （2）SPA（Single Page Application）

後端渲染存在性能問題。

Ajax前端渲染能提高性能，但是不支持瀏覽器的前進後退操作。

SPA（Single Page Application）單頁面應用程序：整個網站只有一個頁面，內容的變化通過ajax局部更新實現，同時支持瀏覽器地址欄的前進後退操作。

SPA實現原理之一：基於URL地址的hash（hash的變化會導致瀏覽器記錄訪問歷史的變化，但是hash的變化不會觸發新的URL請求）

在實現SPA的過程中，最核心的技術點就是前端路由。

### （3）前端路由

概念：根據不同的用戶事件，顯示不同的頁面內容

本質：用戶事件與事件處理函數之間的對應關係



### （4）實現簡易前端路由

基於URL中的hash實現：點擊菜單的時候改變URL的hash，根據hash的變化控制組件的切換。

    <script>

      // 監聽window的onhashchange事件，根據獲取到的最新hash值，切換要顯示的組件的名稱

      window.onhashchange = function () {

        // 通過location.hash 獲取到最新的hash值

      };

    </script>

頁面樣式：



代碼：

  <body>

    <!-- 被vue實例控制的div區域 -->

    <div id="app">

      <!-- 切換組件的超鏈接 -->

      <a href="#/homepage">主頁</a>

      <a href="#/tech">科技</a>

      <a href="#/finance">財經</a>

      <a href="#/entertainment">娛樂</a>

      <!-- 根據:is屬性指定的組件名稱，把對應的組件渲染到component標籤所在的位置 -->

      <!-- 可以把component標籤當做是組件的佔位符 -->

      <component :is="showName"></component>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script>

      // #region定義需要被切換的4個組件

      // 主頁組件

      const homepage = {

        template: "<h1>主頁</h1>",

      };

      // 科技組件

      const tech = {

        template: "<h1>科技</h1>",

      };

      // 財經組件

      const finance = {

        template: "<h1>財經</h1>",

      };

      // 娛樂組件

      const entertainment = {

        template: "<h1>娛樂</h1>",

      };

      // #region vue實例對象

      const vm = new Vue({

        el: "#app",

        data: {

          showName: "homepage",

        },

        // 註冊私有組件

        components: {

          homepage,

          tech,

          finance,

          entertainment,

        },

      });

      // // 監聽window的onhashchange事件，根據獲取到的最新hash值，切換要顯示的組件的名稱

      window.onhashchange = function () {

        // 通過location.hash 獲取到最新的hash值

        console.log(location.hash);

        switch (location.hash.slice(1)) {

          case "/homepage":

            vm.showName = "homepage";

            break;

          case "/tech":

            vm.showName = "tech";

            break;

          case "/finance":

            vm.showName = "finance";

            break;

          case "/entertainment":

            vm.showName = "entertainment";

            break;

        }

      };

    </script>

  </body>

## 2 vue-router的基本使用

### （1）介紹

Vue Router（官網：https://router.vuejs.org/zh/）是Vue.js官方的路由管理器。

它和Vue.js的核心深度集成，可以非常方便地用於SPA應用程序的開發。

Vue Router包含的功能有：

* 支持HTML5歷史模式或hash模式
* 支持嵌套路由
* 支持路由參數
* 支持編程式路由
* 支持命名路由

### （2）基本使用步驟

1. 引入相關的庫文件

先導入vue.js（為全局window對象掛載Vue構造函數），再導入vue-router.js（為全局有window對象掛載VueRouter構造函數），因為vue-router依賴vue。

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script src="https://unpkg.com/vue-router@2.0.0/dist/vue-router.js"></script>

1. 添加路由鏈接

router-link是vue中提供的標籤，默認會被渲染為a標籤

to屬性默認會被渲染為href屬性

to屬性的值默認會被渲染為#開頭的hash地址

      <router-link to="/user">User</router-link>

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

1. 添加路由填充位

    <!-- 路由填充位，也叫路由佔位符 -->

    <!-- 將來通過路由規則匹配到的組件，將會渲染到router-view所在的位置 -->

    <router-view></router-view>

1. 定義路由組件

      var User = {

        template: "<div>User</div>",

      };

      var Register = {

        template: "<div>Register</div>",

      };

1. 配置路由規則並創建路由實例

只要導入了vue-router.js，就會為window掛載VueRouter構造函數

      // 創建路由實例對象

      var router = new VueRouter({

        // routes是路由規則數組

        routes: [

          // 每個路由規則都是一個配置對象，其中至少包含path和component兩個屬性：

          // path表示當前路由規則匹配的hash地址。router-link to=指向那個地址，path就寫那個地址

          // component表示當前路由規則對應要展示的組件

          { path: "/user", component: User },

          { path: "/register", component: Register },

        ],

      });

1. 把路由掛載到Vue跟實例中

      const vm = new Vue({

        el: "#app",

        // 為了能夠讓路由規則生效，必須把路由對象掛載到vue實例對象上

        router, // 也可以寫成router: router。在es6中，如果屬性名和屬性值一致，都是router，可以簡寫成router。

      });

範例代碼

（下面script中的代碼塊順序不能調轉，否則報錯）

  <body>

    <!-- 被vue實例控制的div區域 -->

    <div id="app">

      <router-link to="/user">User</router-link>

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

      <!-- 路由填充位，也叫路由佔位符 -->

      <!-- 將來通過路由規則匹配到的組件，將會渲染到router-view所在的位置 -->

      <router-view></router-view>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script src="https://unpkg.com/vue-router@2.0.0/dist/vue-router.js"></script>

    <script>

      var User = {

        template: "<div>User</div>",

      };

      var Register = {

        template: "<div>Register</div>",

      };

      // 創建路由實例對象

      var router = new VueRouter({

        // routes是路由規則數組

        routes: [

          // 每個路由規則都是一個配置對象，其中至少包含path和component兩個屬性：

          // path表示當前路由規則匹配的hash地址

          // component表示當前路由規則對應要展示的組件

          { path: "/user", component: User },

          { path: "/register", component: Register },

        ],

      });

      const vm = new Vue({

        el: "#app",

        // 為了能夠讓路由規則生效，必須把路由對象掛載到vue實例對象上

        router,

      });

    </script>

  </body>

### （3）路由重定向

路由重定向指：用戶訪問地址A時，被強制跳轉到B地址的頁面。

通過路由規則的redirect屬性，指定一個新的路由地址，可以很方便地設置路由重定向。

      var router = new VueRouter({

        // routes是路由規則數組

        routes: [

          // 下面的第一條表示：如果用戶訪問根目錄，則被強制跳轉到/user目錄

          { path: "/", redirect: "/user" },

          { path: "/user", component: User },

          { path: "/register", component: Register },

        ],

      });

## 3 vue-router嵌套路由

點擊父級路由鏈接顯示模板內容

模板內容中有子級路由鏈接

點擊子級路由鏈接顯示子級模板內容

### （1）父級路由鏈接和父組件路由填充位

      <p>

        <router-link to="/user">User</router-link>

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

      </p>

      <div>

        <router-view></router-view>

      </div>

### （2）子級路由鏈接和子級路由填充位

      var Register = {

        template: `

        <div>

          <h1>Register組件</h1>

          <hr />

          <router-link to="/register/tab1">Tab1</router-link>

          <router-link to="/register/tab2">Tab2</router-link>

          <!-- 子路由填充位 -->

          <router-view />

        </div>

        `,

      };

### （3）範例代碼

  <body>

    <!-- 被vue實例控制的div區域 -->

    <div id="app">

      <p>

        <router-link to="/user">User</router-link>

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

      </p>

      <div>

        <router-view></router-view>

      </div>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script src="https://unpkg.com/vue-router@2.0.0/dist/vue-router.js"></script>

    <script>

      var User = {

        template: "<div>User</div>",

      };

      var Register = {

        template: `

        <div>

          <h1>Register組件</h1>

          <hr />

          <router-link to="/register/tab1">Tab1</router-link>

          <router-link to="/register/tab2">Tab2</router-link>

          <!-- 子路由填充位 -->

          <router-view />

        </div>

        `,

      };

      const Tab1 = {

        template: "<h3>tab1子組件</h3>",

      };

      const Tab2 = {

        template: "<h3>tab2子組件</h3>",

      };

      // 創建路由實例對象

      var router = new VueRouter({

        // routes是路由規則數組

        routes: [

          // 下面的第一條表示：如果用戶訪問根目錄，則被強制跳轉到/user目錄

          { path: "/", redirect: "/user" },

          { path: "/user", component: User },

          // chilren數組表示子路由規則

          {

            path: "/register",

            component: Register,

            children: [

              { path: "/register/tab1", component: Tab1 },

              { path: "/register/tab2", component: Tab2 },

            ],

          },

        ],

      });

      const vm = new Vue({

        el: "#app",

        // 為了能夠讓路由規則生效，必須把路由對象掛載到vue實例對象上

        router,

      });

    </script>

  </body>

## 4 vue-router動態路由匹配

### （1）基本用法

      var router = new VueRouter({

        // routes是路由規則數組

        routes: [

          //  動態路徑參數，以冒號開頭

          { path: "/user/:id", component: User },

        ],

      });

      var User = {

        // 路由組件中通過$route.params獲取路由參數

        template: "<div>User {{ $route.params.id }}</div>",

      };

範例代碼

  <body>

    <!-- 被vue實例控制的div區域 -->

    <div id="app">

      <router-link to="/user/1">User1</router-link>

      <router-link to="/user/2">User2</router-link>

      <router-link to="/user/3">User3</router-link>

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

      <router-view></router-view>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script src="https://unpkg.com/vue-router@2.0.0/dist/vue-router.js"></script>

    <script>

      var User = {

        // 路由組件中通過$route.params獲取路由參數

        template: "<div>User {{ $route.params.id }}</div>",

      };

      var Register = {

        // 路由組件中通過$route.params獲取路由參數

        template: "<div>Register</div>",

      };

      // 創建路由實例對象

      var router = new VueRouter({

        // routes是路由規則數組

        routes: [

          { path: "/", redirect: "/user" },

          //  動態路徑參數，以冒號開頭

          { path: "/user/:id", component: User },

          { path: "/register", component: Register },

        ],

      });

      const vm = new Vue({

        el: "#app",

        // 為了能夠讓路由規則生效，必須把路由對象掛載到vue實例對象上

        router,

      });

    </script>

  </body>

### （2）路由組件傳遞參數

$route與對應路由高度耦合，不夠靈活，可以使用props將組件和路由解耦。

#### ① props值為布爾類型

相關語法

      var router = new VueRouter({

        // routes是路由規則數組

        routes: [

          { path: "/", redirect: "/user" },

          //  如果props被設置為true，route.params將會被設置為組件屬性

          { path: "/user/:id", component: User, props: true },

          { path: "/register", component: Register },

        ],

      });

      var User = {

        props: ["id"], //使用props接收路由參數

        template: "<div>User {{ id }}</div>", //使用路由參數

      };

範例代碼：

  <body>

    <!-- 被vue實例控制的div區域 -->

    <div id="app">

      <router-link to="/user/1">User1</router-link>

      <router-link to="/user/2">User2</router-link>

      <router-link to="/user/3">User3</router-link>

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

      <router-view></router-view>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script src="https://unpkg.com/vue-router@2.0.0/dist/vue-router.js"></script>

    <script>

      var User = {

        props: ["id"], //使用props接收路由參數

        template: "<div>User {{ id }}</div>", //使用路由參數

      };

      var Register = {

        // 路由組件中通過$route.params獲取路由參數

        template: "<div>Register</div>",

      };

      // 創建路由實例對象

      var router = new VueRouter({

        // routes是路由規則數組

        routes: [

          { path: "/", redirect: "/user" },

          //  如果props被設置為true，route.params將會被設置為組件屬性

          { path: "/user/:id", component: User, props: true },

          { path: "/register", component: Register },

        ],

      });

      const vm = new Vue({

        el: "#app",

        // 為了能夠讓路由規則生效，必須把路由對象掛載到vue實例對象上

        router,

      });

    </script>

  </body>

#### ② props為對象類型

  <body>

    <!-- 被vue實例控制的div區域 -->

    <div id="app">

      <router-link to="/user/1">User1</router-link>

      <router-link to="/user/2">User2</router-link>

      <router-link to="/user/3">User3</router-link>

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

      <router-view></router-view>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script src="https://unpkg.com/vue-router/dist/vue-router.js"></script>

    <script>

      var User = {

        props: ["id", "uname", "age"], //使用props接收路由參數

        template: "<div>User {{id}}--- {{uname}}  --- {{age}}</div>", //使用路由參數，其中id的值為空

      };

      var Register = {

        // 路由組件中通過$route.params獲取路由參數

        template: "<div>Register</div>",

      };

      // 創建路由實例對象

      var router = new VueRouter({

        // routes是路由規則數組

        routes: [

          { path: "/", redirect: "/user" },

          {

            path: "/user/:id",

            component: User,

            props: { uname: "lisi", age: 12 }, //用這種方式傳參，id無法傳遞

          },

          { path: "/register", component: Register },

        ],

      });

      const vm = new Vue({

        el: "#app",

        // 為了能夠讓路由規則生效，必須把路由對象掛載到vue實例對象上

        router,

      });

    </script>

  </body>

#### ③ props為函數類型

  <body>

    <!-- 被vue實例控制的div區域 -->

    <div id="app">

      <router-link to="/user/1">User1</router-link>

      <router-link to="/user/2">User2</router-link>

      <router-link to="/user/3">User3</router-link>

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

      <router-view></router-view>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script src="https://unpkg.com/vue-router/dist/vue-router.js"></script>

    <script>

      var User = {

        props: ["id", "uname", "age"], //使用props接收路由參數

        template: "<div>User {{id}}--- {{uname}}  --- {{age}}</div>", //使用路由參數

      };

      var Register = {

        // 路由組件中通過$route.params獲取路由參數

        template: "<div>Register</div>",

      };

      // 創建路由實例對象

      var router = new VueRouter({

        // routes是路由規則數組

        routes: [

          { path: "/", redirect: "/user" },

          {

            path: "/user/:id",

            component: User,

            props: (*route*) => ({ uname: "lisi", age: 12, id: route.params.id }),//  這樣id的值也可以傳過去

          },

          { path: "/register", component: Register },

        ],

      });

      const vm = new Vue({

        el: "#app",

        // 為了能夠讓路由規則生效，必須把路由對象掛載到vue實例對象上

        router,

      });

    </script>

  </body>

## 5 vue-router命名路由

為了更加方便地表示路由的路徑，可以給路由規則起一個別名，即為“命名路由”。

  <body>

    <!-- 被vue實例控制的div區域 -->

    <div id="app">

      <router-link to="/user/1">User1</router-link>

      <router-link :to="{name:'user', params:{id:222}}">User2</router-link>

      <router-link to="/user/3">User3</router-link>

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

      <router-view></router-view>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script src="https://unpkg.com/vue-router/dist/vue-router.js"></script>

    <script>

      var User = {

        props: ["id", "uname", "age"], //使用props接收路由參數

        template: "<div>User {{id}}--- {{uname}}  --- {{age}}</div>", //使用路由參數

      };

      var Register = {

        // 路由組件中通過$route.params獲取路由參數

        template: "<div>Register</div>",

      };

      // 創建路由實例對象

      var router = new VueRouter({

        // routes是路由規則數組

        routes: [

          { path: "/", redirect: "/user" },

          //  如果props被設置為true，route.params將會被設置為組件屬性

          {

            path: "/user/:id",

            component: User,

            name: "user",

            props: (*route*) => ({ uname: "lisi", age: 12, id: route.params.id }), //  這樣id的值也可以傳過去

          },

          { path: "/register", component: Register },

        ],

      });

      const vm = new Vue({

        el: "#app",

        // 為了能夠讓路由規則生效，必須把路由對象掛載到vue實例對象上

        router,

      });

    </script>

  </body>

## 6 vue-router編程式導航

頁面導航的兩種方式：

* 聲明式導航：通過點擊鏈接實現導航的方式。例子：
  + 普通頁面中的<a></a>鏈接，或者vue中的<router-link></router-link>
* 編程式導航：通過調用JavaScript形式的API實現導航的方式。例子：
  + 普通網頁中的location.href
  + Vue中常用的編程式導航API：
    - this.$router.push(‘hash地址’)
    - this.$router.go(n) 實現前進後退

### （1）基本用法

  <body>

    <!-- 被vue實例控制的div區域 -->

    <div id="app">

      <router-link to="/user/3">User3</router-link>

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

      <router-view></router-view>

    </div>

    <script src="https://cdn.jsdelivr.net/npm/vue@2/dist/vue.js"></script>

    <script src="https://unpkg.com/vue-router/dist/vue-router.js"></script>

    <script>

      var User = {

        template:

          "<div>User頁面<button @click='goRegister'>去註冊</button></div>", //使用路由參數

        methods: {

          goRegister: function () {

            // 用編程的方式控制路由跳轉

            this.$router.push("/register");

          },

        },

      };

      var Register = {

        template:

          "<div>Register頁面<button @click='goBack'>後退</button></div>",

        methods: {

          goBack: function () {

            this.$router.go(-1);

          },

        },

      };

      // 創建路由實例對象

      var router = new VueRouter({

        // routes是路由規則數組

        routes: [

          { path: "/", redirect: "/user" },

          {

            path: "/user/:id",

            component: User,

          },

          { path: "/register", component: Register },

        ],

      });

      const vm = new Vue({

        el: "#app",

        // 為了能夠讓路由規則生效，必須把路由對象掛載到vue實例對象上

        router,

      });

    </script>

  </body>

### （2）router.push()方法的參數規則

      // 字符串（路徑名稱）

      router.push("/home");

      // 對象

      router.push({ path: "/home" });

      // 命名的路由（傳遞參數）

      router.push({ name: "/user", params: { userId: 123 } });

      // 帶查詢參數，編程 /register?uname=lisi

      router.push({ path: "/register", query: { uname: "lisi" } });

## 7 基於vue-router的案例

### （1）用到的路由技術要點：

* 路由的基礎用法
* 嵌套路由
* 路由重定向
* 路由傳參
* 編程式導航

### （2）步驟

1. 抽離並渲染App根組件
2. 將左側菜單改造為路由鏈接
3. 創建左側菜單對應的路由組件
4. 在右側主體區域添加路由佔位符
5. 添加子路由規則
6. 通過路由重定向默認渲染用戶組件
7. 渲染用戶列表數據
8. 編程時導航跳轉到用戶詳情頁
9. 實現後退

前端工程化

## 1 模塊化相關規範

### （1）模塊化的好處

傳統開發模式的主要問題：

1. 命名衝突
2. 文件依賴

通過模塊化解決上述問題

模塊化是吧單獨的一個功能封裝到一個模塊（文件）中，模塊之間相互隔離，但是可以通過特定的接口公開內部成員，也可以依賴別的模塊。

模塊化開發的好處：方便代碼重用，方便後期維護。

### （2）ES6模塊化

瀏覽器端模塊化規範：AMD和CMD（已落伍，不推薦使用）

服務器端模塊化規範：CommonJS

1. 模塊分為單文件模塊與包
2. 模塊成員到處：module.exports和exports
3. 模塊成員導入：require(‘模塊標識符’)

ES6模塊化規範是瀏覽器端和服務器端通用的模塊化開發規範。

ES6模塊化規範定義：

1. 每個js文件都是一個獨立的模塊
2. 導入模塊成員使用import關鍵字
3. 暴露模塊成員使用export關鍵字

### （3）在Node.js中通過babel體驗ES6模塊化

#### ① 安裝babel

Node.js默認支持commonJS，但是對ES6模塊化支持得不是特別好，所以需要結合babel這個第三方插件在node中來體驗高級的ES6特性。

Babel是個語法轉換工具，可以把高級的、有兼容性問題的JavaScript代碼轉換為更低一級的、沒有兼容性問題的代碼。

1. npm install –save-dev @babel/core @babel/cli @babel/preset-env @babel/node
2. npm install –save @babel/polyfill
3. 項目跟目錄創建文件babel.config.js
4. babel.config.js文件內容如下面的代碼

    const presets = [

      ["@babel/env",{

        targets:{

          edge:"17",

          firefox:"60",

          chrome:"67",

          safari:"11.1"

        }

      }]

    ];

    module.exports = { presents };

1. 通過npx babel-node index.js執行代碼

上面的安裝步驟是2019年視頻的內容，有些已經過時了。最新的安裝方法看官網的：

<https://www.babeljs.cn/docs/usage>

#### ② 默認導出和導入

默認導出語法export default

每個模塊（js文件）只允許有1個export default。

// 當前文件模塊為m1.js

// 定義私有成員a、b、show，

let a = 10;

let b = 20;

let d = 40;

function show() {

  console.log('這是m1.js')

}

// 將本模塊中的私有成員暴露出去，供其它模塊使用。d沒有暴露出去，所以外界訪問不到d。

export default {

  a,

  b,

  show

}

默認的導入語法import 接收名稱from ‘模塊標識符’。接收名稱可以任意命名，只要合法即可。

如果導入的模塊沒有任何export default，會導入一個空對象。

在每個模塊中，可以多次按需導出。

// 在m2.js中導入模塊成員

import m1 from './m1.js'

console.log(m1);

// 打印輸出的結果為：

// { a: 10, b: 20, show: [Function: show] }

#### ③ 按需導出和導入

按需導出語法 export let s1 = 10;

按需導入語法import {s1} from ‘模塊標識符’

// 當前文件模塊為m1.js

let d1 = 'd1';

const watch = function () {

  console.log('watch')

}

// 默認導出

export default {

  d1,

  watch

}

// 向外按需導出變量s1

export let s1 = 'aaa'

// 向外按需導出變量s2

export let s2 = 'bbb'

// 向外按需導出方法say

// export const say = function () { //或者寫成

export function say() {

  console.log('m1.js裡的function say')

}

// 當前文件為m2.js

// 導入模塊成員

import m1, { s1, s2 as ss2, say } from './m1.js' //m1是默認導入的成員，花括號裡是按需導入的成員。花括號裡，把s2重命名為ss2

console.log(m1); //{ d1: 'd1', watch: [Function: watch] }

console.log(s1); // aaa

console.log(ss2); // bbb

console.log(say);// [Function: say]

#### ④ 直接導入並執行模塊代碼

有時候，我們只想單純地執行某個模塊中的代碼，並不需要得到模塊中向外暴露的成員。此時，可以直接導入並執行模塊代碼。

// 當前文件為m2.js

// 在當前模塊中執行一個for循環操作

for (let i = 0; i < 10; i++) {

  console.log(i);

}

// 當前文件模塊為m1.js

// 直接導入並執行模塊代碼

import './m2.js'

## 2 webpack用法

### （1）webpack概述

Webpack是一個流行的前端項目構建工具（打包工具），優點：

* 支持模塊化
* 梳理簡化錯綜複雜的依賴關係
* 壓縮代碼
* 處理js兼容問題

絕大多數企業的前端項目用webpack打包構建。

### （2）webpack各行變色項目

步驟

1. 新建項目空白目錄，並運行npm init -y命令，初始化包管理配置文件package.json
2. 新建src文件夾，以後開發者自己寫的代碼都放在這個文件夾中
3. 在src中建立index.html文件作為項目首頁
4. 初始化首頁基本結構
5. 運行npm install jquery -S 命令，安裝jQuery
6. 通過模塊化的形式，實現列表各行變色效果

按照上面的步驟做完，遇到問題：瀏覽器無法識別import命令。這時需要webpack來解決問題

### （3）安裝和配置webpack

1. 運行npm install webpack webpack-cli -D命令，安裝webpack相關的包

總結：

npm install xxx --save == npm install xxx -S 生產環境的所需依賴也就是咱們的線上環境。

npm install xxx --save-dev == npm install xxx -D 開發和測試所需的依賴也就是咱們本地環境。

1. 在項目根目錄中，創建名為webpack.config.js的webpack配置文件
2. 在webpack的配置文件中初始化如下基本配置：

module.exports = {

mode: ‘development’

//mode用來指定構建模式是開發模式（development）還是生產模式（production）。development模式的代碼不會被壓縮，這樣轉換速度會快很多。在開發階段，一般把mode的值設置為development。

}

1. 在package.json配置文件中的scripts節點下，新增dev腳本如下：

“scripts”:{

“dev”: “webpack” //script節點下的腳本可以通過npm run執行

}

1. 在終端中運行npm run dev命令，啟動webpack進行項目打包。

最新的安裝和配置方式參看官網：<https://webpack.docschina.org/guides/getting-started/>

（2）和（3）提到的項目的文件結構



其中dist/main是運行npm run dev後打包後自動生成的

src/index.html文件代碼：

<!DOCTYPE html>

<html lang="en">

  <head>

    <meta charset="UTF-8" />

    <meta http-equiv="X-UA-Compatible" content="IE=edge" />

    <meta name="viewport" content="width=device-width, initial-scale=1.0" />

    <title>隔行變色</title>

    <!-- <script src="./index.js"></script> 直接引用index.js會報錯，因為瀏覽器無法識別index.js中的import語法。通過webpack打包後，生成的main.js就解決了這個問題-->

    <script src="../dist/main.js"></script>

  </head>

  <body>

    <ul>

      <li>第1個li</li>

      <li>第2個li</li>

      <li>第3個li</li>

      <li>第4個li</li>

      <li>第5個li</li>

      <li>第6個li</li>

      <li>第7個li</li>

      <li>第8個li</li>

      <li>第9個li</li>

    </ul>

  </body>

</html>

src/index.js代碼

import $ from 'jquery'

$(function () {

  $('li:odd').css('backgroundColor', '#ffd000')

  $('li:even').css('backgroundColor', '#00314f')

})

package.json代碼

{

  "name": "test-webpack",

  "version": "1.0.0",

  "description": "",

  "main": "index.js",

  "scripts": {

    "test": "echo \"Error: no test specified\" && exit 1",

    "dev": "webpack"

  },

  "keywords": [],

  "author": "",

  "license": "ISC",

  "devDependencies": {

    "webpack": "^5.52.0",

    "webpack-cli": "^4.8.0"

  },

  "dependencies": {

    "jquery": "^3.6.0"

  }

}

webpack.config.js代碼

module.exports = {

  // 編譯模式

  mode: 'development'

}

### （4）配置webpack打包的入口與出口

webpack的4.x版本中默認約定：

打包的入口文件為src/index.js

打包的輸入文件為dist/main.js

如果要修改打包的入口與出口，可以在webpack.config.js中新增如下配置信息：

const path = require(‘path’) //導入node.js中專門操作路徑的模塊

module.exports = {

entry: path.join(\_\_dirname, ‘./src/index.js’), //打包入口文件的路徑

output: {

path: path.join(\_\_dirname, ‘./dist’), //輸出文件的存放路徑

filename: ‘bundle.js’ //輸出文件的名稱

}

}

### （5）配置webpack的自動打包功能

1. 運行npm install webpack-dev-server -D命令，安裝支持項目自動打包的工具包。
2. 修改package.json的scripts找那個的dev命令如下（這裡和官網的說明有出入 <https://github.com/webpack/webpack-dev-server> ）：

“scripts”{

“dev”: “webpack-dev-server” //script節點下的腳本，可以通過npm run 執行

}

在webpack-dev-server後面加個 –open，表示編譯完成後，會自動打開主頁<http://localhost:8080>

"dev": "webpack-dev-server --open"

1. 將src/index.html中script腳本的引用路徑修改為”/bundle.js”

    <!-- webpack-dev-server會虛擬一個bundle.js放在內存中根目錄中，而磁盤根目錄中看不見這個bundle.js -->

1. webpack.config.js中要加入下列高亮代碼，否則打開的主頁會顯示Cannot GET，console裡面會顯示404。

const path = require('path') //導入node.js中專門操作路徑的模塊

module.exports = {

  // 編譯模式

  devServer: { static: path.resolve(\_\_dirname, './src') },

  mode: 'development',

  entry: path.join(\_\_dirname, './src/index.js'), //打包入口文件的路徑

  output: {

    path: path.join(\_\_dirname, './dist'), //輸出文件的存放路徑

    filename: 'bundle.js', //輸出文件的名稱

  },

}

上面的高亮代碼也可以改為：

  devServer: { static: \_\_dirname },

這樣<http://localhost:8080>就被定位到了webpack.config.js所在的文件夾

1. 運行npm run dev命令，重新打包
2. 在瀏覽器中訪問<http://localhost:8080>，查看自動打包效果

### （6）配置html-webpack-plugin生成預覽頁面

1. 運行npm install html-webpack-plugin -D命令，安裝生成預覽頁面的插件
2. 修改webpack.config.js文件頭部區域，添加如下配置信息

// 導入生成預覽頁面的插件，得到一個構造函數

const HtmlWebpackPlugin = require('html-webpack-plugin')

const htmlPlugin = new HtmlWebpackPlugin({ //创建插件的实例对象

  template: './src/index.html',//指定要用到的模板文件

  filename: 'index.html' //指定生成的文件的名称，该文件存在于内存中，在目录中不显示

})

1. 修改webpack.config.js文件向外暴露的配置對象，新增如下配置節點

在module.exports = {}中添加：

plugins: [htmlPlugin] //plugins数组是webpack打包期间会用到的一些插件列表

### （7）打包相關的參數

package.json中的配置

-- open 打包完成後自動打開瀏覽器頁面

-- host 配置IP地址

-- port 配置端口

  "scripts": {

    "dev": "webpack-dev-server --open --host 127.0.0.1 --port 6688"

  },

### （8）webpack中的加載器-通過loader打包非js模塊

在實際開發過程中，webpack默認只能打包處理以.js為後綴名的模塊，其它非.js後綴名結尾的模塊，webpack處理不了，需要調用loader加載器才可以正常打包

loader加載器可以協助webpack打包處理特定的文件模塊，比如：

* less-loader可以打包處理.less相關的文件
* sass-loader可以打包處理.scss相關的文件
* url-loader可以打包處理css中與url路徑相關的文件
* babel-loader用來處理ES6語法，將其編譯為瀏覽器可以執行的js語法

loader的調用過程



### （9）loader打包處理css文件

1. 運行npm i style-loader css-loader -D命令，安裝處理css文件的loader
2. 在webpack.config.js的module->rules數組中，添加loader規則如下（高亮部分）：

module.exports = {

  // 編譯模式

  // devServer: { static: path.resolve(\_\_dirname, './src') },

  devServer: { static: \_\_dirname },

  mode: 'development',

  entry: path.join(\_\_dirname, './src/index.js'), //打包入口文件的路徑

  output: {

    path: path.join(\_\_dirname, './dist'), //輸出文件的存放路徑

    filename: 'bundle.js', //輸出文件的名稱

  },

  plugins: [htmlPlugin], //plugins数组是webpack打包期间会用到的一些插件列表

  module: {

    rules: [

      { test: /\.css$/, use: ['style-loader', 'css-loader'] }//test表示文件匹配的類型，接收一個正則表達式，\.表示匹配後綴名的點號，css$表示以css結尾的文件類型。use表示對應要調用的loader。只要test匹配到了以css結尾的文件，都去調用style-loader和css-loader

    ]

  }

}

注意：

* 寫代碼時，use數組中指定的loader順序是固定的。必須先寫style-loader，再寫css-loader，順序不能顛倒。
* 而系統調用多個loader的順序是從後往前調用的。調用的時候，會先調用css-loader來處理代碼，處理好以後，再調用style-loader來處理代碼，然後style-loader把處理結果返回給webpack，讓webpack來打包合併。

### （10）loader打包處理less文件

1. 運行npm i less-loader less -D命令
2. 在webpack.config.js的module->rules數組中，添加loader規則如下：

  module: {

    rules: [

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

    ]

  }

### （11）loader打包處理scss文件

1. 運行npm install sass-loader sass webpack --save-dev
2. 在webpack.config.js的module->rules數組中，添加loader規則如下：

  module: {

    rules: [

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

    ]

  }

注意：文件後綴名是scss，而loader的名字是sass-loader

### （12）postCSS自動添加css的兼容前綴

有些低版本的瀏覽器（比如IE）不兼容某些css樣式。

例如

    <input type="text" placeholder="ceshi" />

把這個輸入框的placeholder的字體顏色改為紅色

::placeholder {

  color: red;

}

在Chrome可以正常顯示，但是在IE無法正常顯示。通常情況下，是在::placeholder前面加上瀏覽器的前綴。但是如果每個都手動去加，就太麻煩了。

可以通過配置postCSS自動添加css的兼容前綴。

1. 運行npm i postcss-loader autoprefixer -D
2. 在項目根目錄中創建postcss的配置文件postcss.config.js，並初始化如下配置：

const autoprefixer = require('autoprefixer') //導入自動添加前綴的插件

module.exports = {

  plugins: { autoprefixer } //掛載插件

}

1. 在webpack.config.js的module->rules數組中，修改css的loader規則如下（添加postcss-loader）：

  module: {

    rules: [

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

    ]

  }

### （13）打包樣式表中的圖片和字體文件

1. 運行npm i url-loader file-loader -D （**webpack5已廢棄使用url-loader和file-loader**）
2. 在webpack.config.js的module->rules數組中，添加loader規則如下：

  module: {

    rules: [

      { test: /\.jpg|png|gif|bmp|ttf|eot|svg|woff|woff2$/, use: 'url-loader?limit=16940' }

    ]

  }

其中，？之後的是loader的參數項。limit用來指定圖片的大小，單位是字節（bite）。只有小於limit大小的圖片，才會被轉為base64圖片。

將小的，即低於limit的圖片進行base64編碼，以減少http請求

webpack裏的url-loader中limit作用：

將圖片文件轉換為base64編碼並載入瀏覽器能夠減少http請求數，但是增大了js或html文件的體積，如果圖片在項目中的重用度較高，那麽每處引用都會生成base64編碼，造成了代碼的冗余。通過http請求載入到瀏覽器的文件可以緩存到本地，當圖片在項目中的重用度較高時，會為圖片的訪問增加緩存的便利性，下次訪問更快。因此要平衡考慮。

是不是可以讓所有的圖片都展示base64，這肯定不行，因為上面那張編碼過的png只有不到5kb，圖片太大的話base64字符串會更長，所以還不如用http請求，這也就是為什麽默認limit是10000了，就是讓8kb之內的圖片才編碼

### （14）打包處理js文件中的高級語法

1. 安裝babel轉換器相關的包：npm i babel-loader @babel/core @babel/runtime -D
2. 安裝babel語法插件相關的包：npm i @babel/preset-env @babel/plugin-transform-runtime @babel/plugin-proposal-class-properties -D
3. 在項目根目錄中，創建babel配置文件babel.config.js並初始化基本配置如下：

module.exports = {

  presets: ['@babel/preset-env'],

  plugins: ['@babel/plugin-transform-runtime', '@babel/plugin-proposal-class-properties']

}

1. 在webpack.config.js的module->rules數組中，添加loader規則如下：

  module: {

    rules: [

      // exclude為排除項，表示babel-loader不需要處理node\_modules中的js文件

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

    ]

  }

### （15）Vue單文件組件

文件後綴名.vue，文件由3部分構成：

* template 組件的模板區域
* script 業務邏輯區域
* style 樣式區域（為每個style添加scoped，防止組件樣式衝突）

<template>

  <!-- 定義vue組件的模板內容 -->

  <div>

    <h1>APP根組件</h1>

  </div>

</template>

<script>

export default {

  // 定義vue組件的業務邏輯

  export default{

    data:(){return {}}, //私有數據

    methods:{} //處理函數

    // 其它業務邏輯

  }

}

</script>

<style scoped>

/\* 定義組件的樣式 \*/

h1{

  color:yellow

}

</style>

### （15）Webpack中vue組件加載器

1. 運行npm install -D vue-loader vue-template-compiler
2. 在webpack.config.js配置文件中，添加vue-loader的配置項如下：

const VueLoaderPlugin = require('vue-loader/lib/plugin')

module: {

    rules: [

      // ……其它規則

      { test: /\.vue$/, loader: 'vue-loader' }

    ]

  },

  plugins: [

//……其它插件

new VueLoaderPlugin()], //plugins数组是webpack打包期间会用到的一些插件列表

Vue腳手架

Element-UI的基本使用