笔记来源：《2019年最全最新Vue、Vuejs教程，从入门到精通》

一、邂逅Vuejs

- vue 读音 /vju:/ ，类似于 view
- vue是一个渐进式的框架，渐进式意味着你可以将vue作为你应用的一部分嵌入其中，带来更丰富的交互体验
- vue有很多特点和web开发中常见的高级功能
  - 解耦视图和数据
  - 可复用的组件
  - 前端路由技术
  - 状态管理
  - 虚拟DOM

1.1、vue.js安装

安装Vue的方式有很多：

方式一：直接CDN引入

```
<!-- 开发环境版本，包含了有帮助的命令行警告 -->
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>

<!-- 生产环境版本，优化了尺寸和速度 -->
<script src="https://cdn.jsdelivr.net/npm/vue"></script>
```

方式二：下载和引入

```
开发环境 https://vuejs.org/js/vue.js

生产环境 https://vuejs.org/js/vue.min.js
```

方式三：NPM安装

后续通过 `webpack` 和 `CLI` 的使用，我们使用该方式

Vue的其他插件安装
在使用 Vue 时，我们推荐在你的浏览器上安装 Vue Devtools。它允许你在一个更友好的界面中审查和调试 Vue 应用。

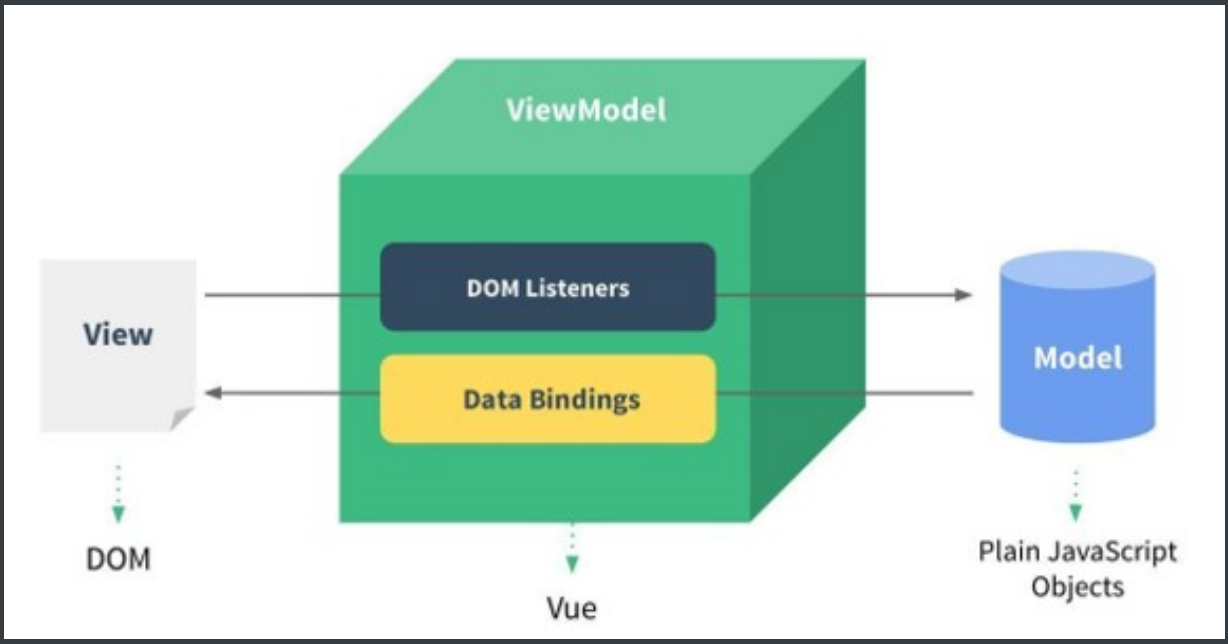
1.2、小案例-计数器

通过vscode打开文件夹，创建html 文件，英文输入法下输入感叹号，按tab键，即可快捷创建html文件。

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">  <!-- 定义一个 div 元素，id 为 app，将 Vue 实例挂载到这个元素上 -->
    <div class="input-num">
        <button @click="sub">-</button>
        <span>Current Num :{{num}}</span>
        <button @click="add">+</button>
    </div>
</div>
<script>
    var app = new Vue({
        el: '#app',
        data: {
            num: 1
```

```
        },
        methods:{
            add:function(){
                this.num++
            },
            sub:function(){
                this.num--
            }
        }}
        )
</script>
```

1.3、mvvm编程思想



Vue 的 MVVM 编程思想是 Model-View-ViewModel 模式的实现，用于简化和组织前端开发。以下是 MVVM 模式的主要概念：

- Model（模型）：表示应用程序的数据和业务逻辑。它负责处理数据的获取、保存和更新，通常与服务器交互。
- View（视图）：表示用户界面（UI），即DOM元素。它的作用是展示数据和接受用户输入。

- ViewModel（视图模型）：是 View 和 Model 之间的桥梁。它负责将 Model 的数据绑定到 View 中，并监听 View 中的事件。当 Model 发生变化时，ViewModel 会自动更新 View；当 View 中的事件被触发时，ViewModel 会相应地更新 Model。

Vue 通过数据绑定（Data Binding）和 DOM 监听（DOM Listener）实现了 ViewModel 的功能，确保数据和视图的同步更新，使得开发者能够更专注于业务逻辑和界面设计。

1.4、Vue的options选项

你会发现，我们在创建Vue实例的时候，传入了一个对象options。

这个options中可以包含哪些选项呢？详细解析：选项-数据

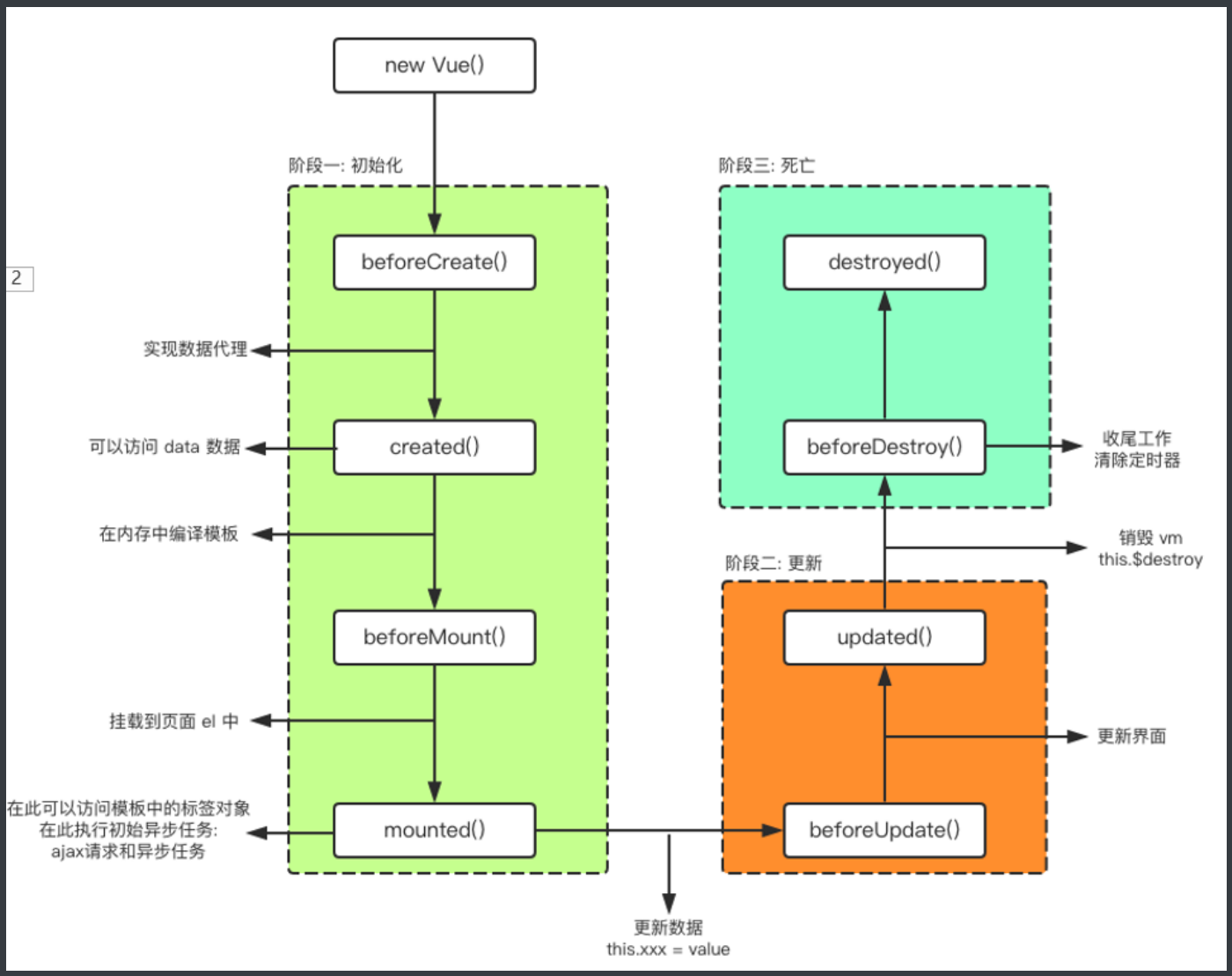
目前掌握这些选项：

el：决定之后 Vue 实例会管理哪一个DOM

data：Vue实例对应的数据对象（组件当中 data 必须是一个函数）

methods：定义属于 Vue 的一些方法，可以在其他地方调用，也可以在指令中使用

1.5、Vue的生命周期



Vue的生命周期包括从实例创建到销毁的整个过程，在此过程中会触发一系列钩子函数。根据上面的图片，Vue的生命周期可以分为三个阶段：初始化、更新和销毁。

阶段一：初始化

1. beforeCreate()：实例初始化之后调用，在这一步，实例的 data 和 methods 都还没有初始化。

   > 详细：在这个阶段，数据是获取不到的，并且真实dom元素也是没有渲染出来的

2. created()：实例已经创建完成，属性已经绑定，可以进行数据的操作，但尚未挂载到DOM中。

   > 详细：在这个阶段，可以访问到数据了，但是页面当中真实dom节点还是没有渲染出来，在这个钩子函数里面，可以进行相关初始化事件的绑定、发送请求操作

3. beforeMount()：在挂载之前调用，相关的render函数首次被调用。

> 详细：代表 DOM 马上就要被渲染出来了，但是却还没有真正的渲染出来，这个钩子函数与created钩子函数用法基本一致，可以进行相关初始化事件的绑定、发送 ajax 操作

4. mounted()：实例挂载到DOM上之后调用，此时可以访问到真实的DOM元素。

> 详细：挂载阶段的最后一个钩子函数,数据挂载完毕，真实dom元素也已经渲染完成了,这个钩子函数内部可以做一些实例化相关的操作

阶段二：更新

1. beforeUpdate()：当响应式数据更新时调用，这时还没有更新DOM。

> 详细：这个钩子函数初始化的不会执行,当组件挂载完毕的时候，并且当数据改变的时候，才会立马执行,这个钩子函数获取DOM的内容是更新之前的内容

2. updated()：数据更新导致的DOM重新渲染和更新之后调用。

> 详细：这个钩子函数获取DOM的内容是更新之后的内容生成新的虚拟DOM，新的虚拟DOM与之前的虚拟DOM进行比对，差异之后，就会进行真实DOM渲染。在updated钩子函数里面就可以获取到因diff算法比较差异得出来的真实dom渲染了

阶段三：销毁

1. beforeDestroy()：实例销毁之前调用，这一步可以执行一些清理工作，比如清除定时器等。

> 详细：当组件销毁的时候，就会触发这个钩子函数代表销毁之前，可以做一些善后操作,可以清除一些初始化事件、定时器相关的东西。

2. destroyed()：实例销毁之后调用，此时所有的事件监听器会被移除，所有的子实例也会被销毁。

> 详细：Vue实例失去活性，完全丧失功能

通过这些生命周期钩子函数，开发者可以在Vue实例的不同阶段执行特定的操作，从而实现更加灵活和强大的功能。

1.6、Vue 的核心功能

基础功能：页面渲染、表单处理提交、帮我们管理DOM(虚拟DOM)节点
组件化开发：增强代码的复用能力，复杂系统代码维护更简单

前端路由：更流畅的的用户体验、灵活的在页面切换已渲染组件的显示，不需与后端做多余的交互
状态集中管理：MVVM 响应式模型基础上实现多组件之间的状态数据同步与管理
前端工程化：结合 webpack 等前端打包工具，管理多种静态资源，代码，测试，发布等，整合前端大型项目。

二、基础语法

2.1、插值操作

2.1.1、Mustache语法

通常被称为 “插值”。它使用双大括号 {{ }} 包裹变量或表达式，将数据绑定到 HTML 中。这种语法使得模板中的动态内容能够根据 Vue 实例的数据进行更新。

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
        <h2>hello,{{name}}</h2>
    <h2>{{firstName}} {{lastName}}</h2>
        <h2>{{counter * 2}}</h2>
 </div>
 <script src="../js/vue.js"></script>
 <script>
       let vm = new Vue({
```

```
            el: '#app',
            data: {
              name:'VueJS',
              firstName:'Kobe',
              lastName: 'Bryant',
              counter:100
            }
          })
    </script>
```

2.1.2、v-once

当你使用 v-once 指令时，Vue.js 会跳过该元素和子元素的重新渲染。这在性能优化方面非常有用，特别是当你知道某个部分的内容不会改变时。不支持表达式。

具体作用如下：

1.**性能优化**：通过使用 v-once，可以避免不必要的重新渲染，从而提高应用的性能。

2.**静态内容**：适用于那些在组件的生命周期内不会变化的静态内容。

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
        <h2 v-once="message"></h2>
 </div>
 <script src="../js/vue.js"></script>
 <script>
       let vm = new Vue({
        el: '#app',
        data: {
          message: 'Hello World! '
        }
      })
 </script>
```

就算对变量重新赋值，页面显示也不会改变

```
vm.message = 'java gogogo'
```

2.1.3、v-html

v-html 指令用于将包含 HTML 的字符串插入到元素中，并将其作为真正的 HTML 渲染。这与 Mustache 语法 {{ }} 不同，后者会将内容作为纯文本插入，并对 HTML 标签进行转义。

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
        <h2 v-html="link"></h2>
        <h2>{{link}}</h2>
 </div>

 <script>
       let vm = new Vue({
        el: '#app',
        data: {
          link: '<a href="http://wwww.baidu.com">百度一下</a>'
        }
      })
 </script>
```

2.1.4、v-text

- v-text作用和Mustache比较相似：都是用于将数据显示在界面中
- v-text通常情况下，接受一个string类型

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
        <h2 v-text="message"></h2>
        <h2>{{message}}</h2>
```

```
    </div>

    <script>
          let vm = new Vue({
           el: '#app',
           data: {
             message: 'Hello World! '
           }
         })
    </script>
```

2.1.5、v-pre

v-pre用于跳过这个元素和它子元素的编译过程，用于显示原本的Mustache语法。

比如下面的代码：

- 第一个h2元素中的内容会被编译解析出来对应的内容
- 第二个h2元素中会直接显示{{message}}

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
    <h2>{{message}}</h2>
    <h2 v-pre>{{message}}</h2>
</div>

<script>
    let vm = new Vue({
        el: '#app',
        data: {
            message: 'Hello World!'
        }
    })
</script>
```

2.1.6、v-cloak

- 网络慢时，如果 Vue.js 尚未加载完成，页面可能会显示出未渲染的 Vue 模板代码。为了避免这种情况，可以使用 v-cloak 指令来隐藏未编译的 Mustache 标签 ({{ }})，提升用户体验并防止页面闪烁。
- cloak: 斗篷

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
    <h2 v-cloak>Hello {{name}}</h2>
</div>

<script>
    setTimeout(()=>{
        let vm = new Vue({
            el: '#app',
            data: {
                name: 'VueJS!'
            }
        })
    },10000)
</script>

<style>
    [v-cloak]{
        dispaly:none;
    }
</style>
```

添加了 `v-cloak` 会渲染完成后再显示

2.2、绑定属性

2.2.1、v-bind基础语法

通过 v-bind，你可以将数据绑定到 HTML 属性，CSS 类，样式以及其他组件的 prop。它允许你在模板中绑定一个或多个属性，并在数据发生变化时自动更新这些属性，或者向另一个组件传递props值(这个学到组件时再介绍)。

比如通过 Vue 实例中的 data 绑定元素的 src 和 href，代码如下：

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
    <a v-bind:href="link">vue官网</a>
    <img v-bind:src="logoURL" alt="">


    <!--语法糖，省略写法-->
    <a :href="link">vue官网</a>
    <img :src="logoURL" alt="">
</div>


<script>
    let vm = new Vue({
        el: '#app',
        data: {
            link:'https://cn.vuejs.org/index.html',
            logoURL:'https://cn.vuejs.org/images/logo.png'
        }
    })
</script>
```

2.2.2、例如绑定css类

很多时候，我们希望动态的来切换class，比如：

- 当数据为某个状态时，字体显示红色；当数据另一个状态时，字体显示黑色。

绑定class有两种方式：

- 对象语法
- 数组语法

绑定方式：对象语法

- 对象语法的含义是:class后面跟的是一个对象。

对象语法有下面这些用法：

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

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

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

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

演示案例：

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<style>
```

```
    .active{
        color: red;
    }

    .line{
        text-decoration:underline;
    }
</style>

<div id="app">
    <h2 :class="{'active': isActive,line:this.isLine}">{{message}}
</h2>
    <h2 :class="getClasses()">{{message}}</h2>
    <button v-on:click="btnClick">按钮</button>
</div>

<script>
    let vm = new Vue({
        el: '#app',
        data: {
            message:'Hello World',
            isActive:true,
            isLine:true
        },
        methods:{
            btnClick:function () {
                this.isActive=!this.isActive;
                this.isLine=!this.isLine;
            },
            getClasses:function () {
                return {active:this.isActive,line:this.isLine}
            }
        }
    })
</script>
```

绑定方式：数组语法

- 数组语法的含义是:class后面跟的是一个数组。

数组语法有下面这些用法：

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

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

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

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

演示案例:

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<style type="text/css">
    .active {
        color: red;
    }
    .line {
        text-decoration:underline;
    }
</style>

<div id="app">
```

```
    <div v-bind:class='[activeClass, lineClass]'>{{message}}</div>
    <button v-on:click='handle'>切换</button>
</div>

<script type="text/javascript">
    var vm = new Vue({
        el: '#app',
        data: {
            message:'Hello World',
            activeClass: 'active',
            lineClass: 'line'
        },
        methods: {
            handle: function(){
                this.activeClass = '';
                this.lineClass = '';
            }
        }
    });
</script>
```

2.2.3、例如绑定style

我们可以利用 v-bind:style 来绑定一些 CSS 内联样式。

在写 CSS 属性名的时候，比如 font-size

- 我们可以使用驼峰式 (camelCase) fontSize
- 或短横线分隔 (kebab-case，记得用单引号括起来) ‘font-size’

绑定 style 有两种方式：

绑定方式：对象语法

```
:style="{color: currentColor, fontSize: fontSize + 'px'}"
```

style 后面跟的是一个对象类型
对象的 key 是 CSS 属性名称
对象的 value 是具体赋的值，值可以来自于 data 中的属性

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
    <div :style="{color: currentColor, fontSize: fontSize + 'px'}">
{{message}}</div>
    <button v-on:click='handle'>切换</button>
</div>

<script type="text/javascript">
    var vm = new Vue({
        el: '#app',
        data: {
            message:'Hello World',
            currentColor: 'red',
            fontSize: 100
        },
        methods: {
            handle: function(){
                this.currentColor = 'blue';
                this.fontSize =   this.fontSize * 2;
            }
        }
    });
</script>
```

绑定方式：数组语法

```
<div v-bind:style="[baseStyles, overridingStyles]"></div>
```

style后面跟的是一个数组类型
多个值以，分割即可

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
    <div :style="[baseStyles,overridingStyles]">{{message}}</div>
    <button v-on:click='handle'>切换</button>
</div>

<script type="text/javascript">
    var vm = new Vue({
        el: '#app',
        data: {
            message:'Hello World',
            baseStyles: {
                color: 'green',
                fontSize: '30px'
            },
            overridingStyles: {
                'font-weight': 'bold'
            }
        },
        methods: {
            handle: function(){
                this.baseStyles.color = 'blue';
                this.baseStyles.fontSize=  '60px';
            }
        }
    });
</script>
```

2.3、计算属性

基础含义

我们知道，在模板中可以直接通过插值法显示一些 data 中的数据

但是在某些情况下，我们需要对数据进行一些转化后再显示，或者需要将多个数据结合起来进行显示

- 比如我们有 firstName 和 lastName 两个变量，我们需要显示完整的名称
- 但是如果多个地方需要显示完整的名称，我们就需要写多个 {{firstName }}{{lastName }}

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
    <h2>{{firstName}} {{lastName}}</h2>
</div>

<script>
    let vm = new Vue({
        el: '#app',
        data: {
            firstName:'Kobe',
            lastName: 'Bryant',

        }
    })
</script>
```

- 我们可以将上面的代码换成计算属性，我们发现计算属性是写在实例的 `computed` 选项中的。

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
```

```
<div id="app">
    <h2>{{getfullname}}</h2>
</div>

<script>
    let vm = new Vue({
        el: '#app',
        data: {
            firstName:'Kobe',
            lastName: 'Bryant',

        },
        computed:{
            getfullname : function(){
                return this.firstName+" "+this.lastName
            }
        }
    })
</script>
```

计算属性可以在 Vue 组件的 computed 选项中定义。它们通常用来代替在模板中复杂的表达式计算，提供更清晰和更易于维护的代码。

计算属性的复杂用法

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
    <h2>总价格:{{totalPrice}}</h2>
    <h2>平均价格:{{average}}</h2>
</div>

<script>
    let vm = new Vue({
        el: '#app',
```

```
        data: {
            books:[{id:110,name:'Unix编程艺术',price:119},
                {id:111,name:'代码大全',price:105},
                {id:112,name:'深入理解计算机原理',price :98},
                {id:113,name:'现代操作系统',price:85}
            ]
        },
        computed:{
            totalPrice : function(){
                let result = 0
                for(let i=0;i<this.books.length;i++){
                    result += this.books[i].price
                    console.log(result);
                }
                return result
            },
            average:function(){
                return Math.round((this.totalPrice/3));
            }
        }
    })
</script>
```

计算属性setter和getter

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id='app'>
    <!--通过定义computed,显示四行，但是对应方法只调用一次，建议多采用这种方式，性能更高-->
    <h2>{{fullName}}</h2>
    <h2>{{fullName}}</h2>
    <h2>{{fullName}}</h2>
    <h2>{{fullName}}</h2>
```

```
    <!--通过定义methods,显示四行，但是对应方法会调用四次-->
    <h2>{{getFullName()}}</h2>
    <h2>{{getFullName()}}</h2>
    <h2>{{getFullName()}}</h2>
    <h2>{{getFullName()}}</h2>
</div>


<script>
    let vm = new Vue({
    el: '#app',
    data:{
        firstName:'Kobe',
        lastName:'Bryant'
    },
    computed:{
        fullName:{
            // 计算属性一般没有set方法，只有get只读方式
            get:function(){
                console.log("by computed......");
                return this.firstName + ' ' + this.lastName
            }
        }
    },
    // 也可以使用 methods 方法
    methods:{
            getFullName(){
                 console.log("by methods.......");
                 return this.firstName + ' ' + this.lastName
             }
    }
})
</script>
```

2.4、ES6补充

ECMAScript 6.0 简介

ECMAScript 6.0 (以下简称 ES6), ECMAScript 是一种由 Ecma 国际 (前身为欧洲计算机制造商协会,英文名称是 European Computer Manufacturers Association) 通过 ECMA-262 标准化的脚本程序设计语言) 一种脚本语 言的标准,已经在 2015 年 6 月正式发布了,并且从 ECMAScript6 开始,开始采用年号来做版本。即 ECMAScript 2015,就是 ECMAScript6 它的目标,是使得 JavaScript 语言可以用来编写复杂的大型应用程序,成为企业级开发语 言。每年一个新版本。

ES6新特性-let&const

- var声明的变量往往会越域 , let声明的变量有严格的局部作用域

```
{
 var a=1
 let b=2
}
 console.log(a)
 console.log(b)
//Uncaught ReferenceError: b is not defined at let&const.html:19 11
</script>
```

- var 可以声明多次,let只可以声明一次

```
 var a=1
 var a=3
 let b=2
 let b=4
 console.log(a)
 console.log(b)
 //Uncaught SyntaxError: Identifier 'b' has already been declared
```

- var会变量提升 , let不会变量提升

```
    console.log(a)
    var a=1
    console.log(b)
    let b=2
    // Uncaught
    // ReferenceError: Cannot access 'b' before initialization
```

- const 声明之后不允许改变，一旦声明必须初始化， 否则报错

```
    const a=3
    a=4
   // Uncaught
   // TypeError: Assignment to constant variable.
```

ES6新特性-解构&字符串

```
//数组解构赋值
 let arr=[1,2,3];// 初始变量赋值

 let d=arr[0];   // 一、传统的赋值方式，将数组 arr 的每个元素赋值给变量 d, b, 和 c
 let b=arr[1];
 let c=arr[2];


 let [d,b,c]=arr; // 二、ES6 解构赋值

 console.log(d,b,c); // 输出:  1, 2 , 3
 console.log(d,b);   // 输出:  1, 2
```

```
//对象解构
let person={
    name: "jack",
    age: 21,
    language: ['java','js','css'],
}

 let {name,age,language}=person  // 使用ES6 的对象解构赋值语法取代了传统的let name = person.namge 的写法

 console.log(name,age,language)
```

```
// ES6 引入了一些字符串方法来扩展和增强字符串操作的能力

let str="hello.vue";

console.log(str.startsWith("hello"))// 检查字符串 str 是否以 "hello" 开头 ： true
console.log(str.endsWith(".vue"))   // 检查字符串 str 是否以 ".vue" 结尾： true
console.log(str.includes("e"));     // 检查字符串 str 是否包含字符 "e" : true
console.log(str.includes("hello"))  // 检查字符串 str 是否包含子字符串 "hello" ： true
```

```
  // ES6 引入了模板字符串（template literals），也称为模板字面量，是一种增强版的字符串，用反引号（``）包围，可以包含多行字符串和插入变量。以下是你的代码及其详细解释：

  let ss=`<div>
            <a>11</a>
          </div>
          `
  console.log(ss) // <div><a>11</a></div>
```

```
let person={
      name: "jack",
      age: 21,
      language: ['java','js','css'],
}

// 解构赋值从对象中提取属性
let {name,age,language}=person

// 定义一个函数
function fun(){
   return "这是一个函数"
}

// 在模板字符串中插入变量和表达式
let info=`我是${name},今年${age+10},我想说${fun()}`


console.log(name,age,language) // 输出:  jack 21 ['java', 'js', 'css']
console.log(info)              // 输出:  我是jack,今年31,我想说这是一个函数
```

ES6新特性-函数优化

```
//函数默认值
//在ES6以前，我们无法给一个函数参数设置默认值，只能采用变通写法:
function add(a, b) {
  // 判断b是否为空，为空就给默认值1
  b=b||1;
  return a + b;
}
```

```
// 传一个参数
console.log(add(10));    // 输出: 11

//现在可以这么写:直接给参数写上默认值，没传就会自动使用默认值
function add2(a, b = 1) {
   return a + b;
}
console.log(add2(20));  // 输出: 21



//可变长度参数
function fun(...values){
    console.log(values.length)
}

fun(5)   // 输出:  1
fun(5,5,6) // 输出 : 3



//简单的箭头函数
//    function fun(a,b){
//        return a+b;
//    }
var sum=(a,b) => a+b
console.log(sum(11,11))



//箭头函数
//        const person={                    // 定义一个对象
//          name: "jack",
//          age: 21,
//          language: ['java','js','css'],
//        }
```

```
//        function hello (person) {       // 定义一个普通函数
//             console.log(person.name)
//        }

let hellos=(obj) => console.log(obj.name) // 使用箭头函数定义
hellos(person)                            // 调用箭头函数，输出：  jack
```

ES6新特性-对象优化

```
  // 对象的内置函数
   const person = {
      name: "jack",
      age: 21,
      language: ['java', 'js', 'css']
   }

   console.log(Object.keys(person));//["name", "age", "language"]
  数组存放key
   console.log(Object.values(person));//["jack", 21, Array(3)]
  数组存放value
   console.log(Object.entries(person));//[Array(2), Array(2),
Array(2)] 数组存在对象

   // 1）、使用 Object.assign 方法将多个对象的属性合并到一个目标对象中
   const target = { a: 1 };
   const source1 = { b: 2 };
   const source2 = { c: 3 };


    Object.assign(target, source1, source2);
    console.log(target);//{a:1,b:2,c:3}  把三个对象合并为一个

   //2）、声明对象简写
   const age = 23   // 定义变量
```

```
const name = ""
const person1 = { age: age, name: name } //使用传统方式声明对象
const person2 = { age, name } //使用简写方式声明对象
console.log(person2);  // { age: 23, name: "" }

//3）、声明对象书写方式
let person3={
     name: "xushu",

    // 使用传统的函数表达式定义方法
     eat: function(food){
           console.log("我吃了"+food)
     },

    // 使用箭头函数定义方法
    eat1: food => console.log("我吃了"+food),

    // 使用简洁方法语法定义方法（ES6）
    eat3(food){
        console.log("我吃了"+food)
     }
 }

 person3.eat("香蕉")       // 我吃了香蕉
 person3.eat1("苹果")      // 我吃了苹果
 person3.eat3("肥肠")      // 我吃了肥肠


 //4）、对象拓展运算符

 //4.1 拷贝对象
 let p1={
     name1: "zlj",
     age: 19
 }
```

```
    // 使用对象拓展运算符拷贝对象，... 将 p1 对象的所有属性拷贝到新对象
someone 中
    let someone={...p1}   // { ...p1 } 会展开 p1 对象的所有属性并将其复制
到新对象中
    console.log(someone)  // {name1: "zlj", age: 19}


   //4.2 对象合并
    let name1={name1: "zlj"}
    let age={age: 19}

    // 使用对象拓展运算符将 name1 和 age 对象的属性合并到一个新对象 someone1
中。
    let someone1={...name1,...age} // { ...name1, ...age } 会将 name1
和 age 对象的所有属性展开并合并到新对象中
    console.log(someone1) //{name1: "zlj", age: 19}
```

filter 函数的使用

```
    // 定义数组
    const nums =[10,20,111,222,444,40,50]

    //使用 filter 方法
    let newNums = nums.filter(function(n){
        return n <100
    })
    //  • filter 方法用于创建一个新数组，其中包含所有通过回调函数测试的元素。
    //  • 回调函数 function(n) 接受一个参数 n，表示当前正在处理的数组元素。
    //  • 在回调函数中，条件 n < 100 用于测试当前元素 n 是否小于 100。
    //  • 如果条件为 true，则该元素被保留在新数组中。
    //  • 如果条件为 false，则该元素被过滤掉。


    console.log(newNums);// 输出：[10, 20, 40, 50]
```

map 函数的使用

```
let newNums = [10, 20, 40, 50];
let new2Nums = newNums.map(function(n){
  return n*2
})

//  • map 方法用于创建一个新数组，其中包含对原数组的每个元素调用提供的回调函数后返回的结果。
//  • 回调函数 function(n) 接受一个参数 n，表示当前正在处理的数组元素。
//  • 在回调函数中，表达式 n * 2 将当前元素 n 的值乘以 2，并返回结果。
//  • map 方法会将每个回调函数返回的结果收集到一个新数组中，并最终返回这个新数组。


console.log(new2Nums) // 输出：[20, 40, 80, 100]
```

reduce 函数的使用

```
    let new2Nums = [20, 40, 80, 100];
    let total = new2Nums.reduce(function(preValue,n){
        return preValue + n;
    },0)

// • reduce 方法用于对数组中的所有元素进行汇总，并将其累积为一个单一的结果。
// • 回调函数 function(preValue, n) 接受两个参数：
//    preValue：上一次回调函数执行时的返回值，或者初始值（在第一次执行时）。
//    n：当前正在处理的数组元素。
// • 回调函数中，表达式 preValue + n 将上一次累积的值与当前元素 n 相加，并返回结果。
// • 0 是 reduce 方法的初始值，在第一次调用回调函数时作为 preValue 的值。

    console.log(total); // 输出：240
```

上面的三个方法可以合并为以下：

```
    const nums =[10,20,111,222,444,40,50]
    let total = nums.filter(function(n){
        return n<100
    } ).map(function(n){
        return n*2
    }).reduce(function(preValue,n){
        return preValue + n
    },0)

    console.log(total);// 240
```

还可以继续简写为：

```
let total = nums.filter(n => n<100) .map( n => n*2) .reduce((pre,n) => pre+n);

console.log(total); // 240
```

输出结果与上面一致。

2.5、事件监听

在前端开发中，我们经常需要与用户交互。为此，我们必须监听用户操作事件，例如点击、拖拽、键盘事件等。

在 Vue 中，我们可以使用 v-on 指令来监听这些事件。

基本使用

一个监听按钮的点击事件案例：

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id='app'>
        <h2>点击次数：{{counter}}</h2>
        <button v-on:click="btnClick"> 按钮点击 </button>
    </div>
</div>

    <script>
        let app = new Vue({
            el:'#app',
            data:{
                counter:0
            },
            methods:{
```

```
                btnClick(){
                    this.counter++;
                }
            }
        })
    </script>
```

注：v-on也有对应的语法糖：

v-on:click 可以写成 @click

```
<div id="app">
    <h2>点击次数：{{counter}}</h2>
    <button  @click="btnClick"> 按钮点击</button>
</div>
```

v-on的参数传递问题

当在 methods 中定义方法，以供@click调用时，需要注意参数问题：

- 情况一：如果该参数不需要额外参数，那么方法后的（）可以不添加，但是注意，如果方法本身中有一个参数，那么默认将原生事件event参数传递进去
- 情况二：如果需要传入某个参数的同时需要event时，可以通过$event传入事件。

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id='app'>
    <h2>点击次数：{{counter}}</h2>
    <button @click="handleAdd"> +1 </button>
    <button @click="handleAddTen(10,$event)"> +10 </button>
</div>


    <script>
```

```
        let app = new Vue({
            el:'#app',
            data:{
                counter:0
            },
            methods:{
                // vue 会默认将浏览器产生的even事件对象作为参数传入到方法中
                handleAdd(event){
                    console.log("event:"+event);
                    this.counter++
                },
                handleAddTen(count,event){
                    console.log("event:"+event);
                    this.counter +=10;
                }
            }
        })
    </script>
```

输出：

```
event:[object PointerEvent]
```

v-on 修饰符

在某些情况下，我们拿到event的目的可能是进行一些事件处理vue提供了修饰符来帮助我们方便的处理一些事件：

- .stop　　调用 event.stopPropagetion() ， 例如，@click.stop ="btnClick" ,按钮点击不会生效
- .prevent　　调用 event.perventDefault() , 例如，@click.prevent ="btnClick" ,按钮点击可以输出日志等，但是不提交对应表单请求
- .{keyCode | keyAlias}　　只当事件是从特定键触发时才触发回调，例如，@keyup.enter="btnClick", 输入回车才会触发点击方法
- .native　监听组件根元素的原生事件

- .onece　只触发一次回调

v-if 、v-else-if、v-else

更推荐使用 `computed` 以避免页面复杂

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id='app'>
    <h2>{{result}}</h2>
</div>

<script>
    const app = new Vue({
        el:'#app',
        data:{
            score:99
        },
        computed:{
            result(){
                let showMessage ='';
                if(this.score >=90){
                    showMessage ="优秀"
                }else {
                    showMessage ="不优秀"
                }
                return showMessage
            }
        }
    })
</script>
```

输出：

```
优秀
```

条件渲染案例

我们来做一个简单的小案例：

- 用户在登录时，可以切换使用用户账户登录还是邮箱地址登录

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id='app'>
  <span v-if="isUser">
      <!--点击label时，因为for指向input，光标自动聚集在输入框上-->
        <label for="username">用户账号：</label>
        <input type ="text" id="username" placeholder="用户账号"
key="username">
        <!--placeholder 输入框的文字提示-->
  </span>
  <span v-else>
      <label for="email">邮箱地址：</label>
        <!--key 如果于与上面一样，上面如果输入了123 ，那么这里输入框里还是会展示
123-->
        <!--这是因为vue在进行DOM渲染时，出于性能考虑，会尽可能复用已经存在的元
素-->
        <!--如果不希望vue出现类似重复利用的问题，可以给对应的input添加key-->
        <!--并且保证key的值不相同-->
      <input  type ="text" id="email" placeholder="邮箱地址"
key="email">
  </span>
    <button @click="isUser = !isUser"> 切换类型 </button>
</div>
```

```
<script>
    let app= new Vue({
        el:'#app',
        data:{
            isUser:true
        }
    })
</script>
```

key的 作用主要是为了高效的更新虚拟DOM

v-show

v-show: 就会在标签中添加display样式, 如果vlaue为true, display=block, 否则是none

```
<button id="show" v-show="deleteButton">我是删除按钮，我通过v-show控制显隐
</button>
<button v-on:click="deleteButton = true">设置显示</button>
<button v-on:click="deleteButton = false">设置隐藏</button>
```

v-show 的用法和 v-if非常相似，也用于决定一个元素是否渲染

v-if 和 v-show 都可以决定一个元素是否渲染，那么开发中我们如何选择呢?

- v-if 当条件为 false时，压根不会有对应的元素出现在DOM中
- v-show当条件为false时，仅仅是将元素的display属性设置为none而已

开发中如何选择呢?

- 当需要在显示和隐藏之间切换很频繁时，使用v-show
- 当只有一次切换时，通过使用v-if

2.6、条件和循环

v-for 遍历数组

当我们有一组数据需要进行渲染时，我们就可以使用v-for来完成

- v-for 的语法类似于 javascript 中的 for 循环
- 格式如下：iterm in iterms 的形式

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id='app'>
    <!--1.在遍历数组的过程中，没有使用索引值（下标值）-->
    <ul>
        <li v-for="xxx in names"> {{xxx}}</li>
    </ul>
    <!--2.在遍历的过程中，使用索引值（下标值）-->
    <ul>
        <li v-for="(xxx ,index) in names"> {{index+1}} - {{xxx}}</li>
    </ul>
</div>


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

输出：

```
why
kobe
james
curry

1 - why
2 - kobe
3 - james
4 - curry
```

v-for 遍历对象

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id='app'>
    <!--1.在遍历对象的过程中，没有使用索引值（下标值）-->
    <ul>
        <li v-for="xxx in info"> {{xxx}}</li>
    </ul>
    <!--2.在遍历对象的过程中，获取key和value格式：(value,key)-->
    <ul>
        <li v-for="(value ,key) in info"> {{key}} : {{value}}</li>
    </ul>
</div>

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

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

输出:

```
VueJS
18
1.88

name : VueJS
age : 18
height : 1.88
```

哪些数组的方法是响应式的

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id = "app">
    <ul>
        <li v-for = "xxx in letters" > {{xxx}}
    </ul>
    <button @click="btnClick"> 按钮 </button>
</div>


<script>
    const app = new Vue({
        el:'#app',
        data:{
            letters :['a','b','c','d']
        },
        methods:{
```

```
            btnClick(){
                // 一、通过数组方法
                // 1. push 方法
                // 页面效果：当点击按钮时，最下面会添加一个 aaa，点一次添加一次
                this.letters.push('aaa')

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

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

                // 4. unshift(): 在数组最前面添加元素 ， 可以一次添加多个
("aaa","bbb")
                this.letters.shift("aaaa");

                // 5. slice作用：删除元素、插入元素、替换元素
                // 删除元素：第二个参数传入你要删除几个元素
                const start =2
                this.letters.splice(start,this.letters.length -start )
                // 替换元素：第二个参数传入你要替换几个元素，后面是用来替换前面
的元素
                this.letters.splice(1,3,'m','n','l','x')
                // 插入元素：第二个参数传入0,并且后面跟上要插入的元素
                this.letters.splice(1,0,'x','y','z')

                // this.letters.sort() 排序
                // this.letters.reverse() 反转

                // 二、通过索引值修改数组中的元素，可能导致bug
                this.letters[0] = 'bbbbbbb'

                // 页面效果：点击按钮页面显示的数组元素没有发生变化，控制台显示
第一个元素已经变成 'bbbbbbb'
                console.log(this.letters[0]);
            }
```

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

点击变色案例

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<style>
    .active{
        color:red;
    }
</style>

<div id="app">
    <ul>
        <li v-for="(item,index) in movies"
            :class = "{active:currentIndex === index}" @click
="liClick(index)">
            {{index}} - {{item}}
        </li>
    </ul>
</div>


<script>
    const app = new Vue({
        el:'#app',
        data:{
            movies:['海王','海贼王','加勒比海盗','海尔兄弟'],
            currentIndex:0
        },
        methods:{
            liClick(index){
                this.currentIndex = index
```

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

输出：

```
· 0 - 海王
· 1 - 海贼王
· 2 - 加勒比海盗
· 3 - 海尔兄弟
```

2.7、表单绑定

v-model

在实际开发中，表单控件非常常见，尤其是用户信息提交时，Vue 使用 v-model 指令实现表单元素和数据的双向绑定。

用于 input 元素

通过 v-model 绑定 message 实现了输入内容与 DOM 中显示的 message 值的双向绑定。

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
    <input type="text" v-model="message">
    <h2>输入的内容是：{{message}}</h2>
</div>


<script>
    let app = new Vue({
        el:'#app',
        data:{
```

```
            message:''
        }
    })
</script>
```

用于 textarea 元素

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
    <textarea type="text" v-model="message"> </textarea>
    <h2>输入的内容是: {{message}}</h2>
</div>

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

用于 radio 元素

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
     <!--如果有 v-model属性，那么男和女的选择就可以是互斥的，所以可以删除 name属性-->
     <label for="male">
      <input type="radio" id="mal" name ="sex" value ="男" v-model ="sex">男
     </label>
    <label for="female">
```

```
        <input type="radio" id="female" name ="sex" value ="女" v-model
="sex">女
      </label>
     <h2>您选择的性别是：{{sex}}</h2>
</div>


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

页面效果：

```
比如选择的是男，页面显示：您选择的性别是：男
```

用于 checkbox 元素

单选框：

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
     <!--如果有label属性，页面点击label文字也可以选择到复选框-->
     <label for="male">
      <input type="checkbox" id="agree"  v-model ="isAgree">同意协议
     </label>

     <h2>您选择的是：{{isAgree}}</h2>
     <!-- 复选框选择同意，下一步按钮才可以被点击-->
     <button :disabled="!isAgree">下一步</button>
```

```
    </div>

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

多选框:

```
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <div id="app">
        <input type="checkbox" value ="篮球" v-model ="hobbies">篮球
        <input type="checkbox" value ="足球" v-model ="hobbies">足球
        <input type="checkbox" value ="乒乓球" v-model ="hobbies">乒乓球
        <input type="checkbox" value ="羽毛球" v-model ="hobbies">羽毛球

        <h2>您的爱好是: {{hobbies}}</h2>
    </div>
    <script src="../js/vue.js"></script>
    <script>
        const app = new Vue({
            el:'#app',
            data:{
                hobbies:[]
            }
        })
    </script>
```

页面输出效果:

```
您的爱好是：[篮球,足球]
```

用于 select 元素

单选

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
    <select name = "abc" id="" v-model ="sport">
        <option  value ="篮球">篮球</option>
        <option  value ="足球">足球</option>
        <option  value ="乒乓球">乒乓球</option>
        <option  value ="羽毛球">羽毛球</option>
    </select>
    <h2>您选择的运动是：{{sport}}</h2>
</div>
<script src="../js/vue.js"></script>
<script>
    const app = new Vue({
        el:'#app',
        data:{
            sport:'羽毛球'
        }
    })
</script>
```

多选

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
    <select name = "abc" id="" v-model ="sport" multiple>
        <option  value ="篮球">篮球</option>
        <option  value ="足球">足球</option>
        <option  value ="乒乓球">乒乓球</option>
        <option  value ="羽毛球">羽毛球</option>
```

```
    </select>
    <h2>您选择的运动是: {{sport}}</h2>
</div>
<script src="../js/vue.js"></script>
<script>
    const app = new Vue({
        el:'#app',
        data:{
            sport:[]
        }
    })
</script>
```

值绑定

复选框的值来源于data

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
    <lable v-for ="item in originHobbies" :for="item">
        <input type="checkbox" :value="item" :id="item" v-
model="hobbies"> {{item}}
    </lable>

    <h2>您的爱好是: {{hobbies}}</h2>
</div>
<script src="../js/vue.js"></script>
<script>
    const app = new Vue({
        el:'#app',
        data:{
            hobbies:[],
            originHobbies:['篮球','足球','乒乓球','羽毛球','台球','高尔夫
球']
        }
    })
```

```
</script>
```

修饰符

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

三、组件开发

组件化是 Vue.js 中的重要思想，它提供了一种抽象方式，让我们可以开发独立、可复用的小组件来构建应用，任何应用都可以被抽象成一个组件树。应用组件化思想时，应充分利用这一特性，将页面拆分成小的、可复用的组件，使代码更易组织和管理，并具备更强的扩展性。使用组件分为三个步骤：创建组件构造器、注册组件、使用组件。

简单案例

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

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
    <!--3. 使用组件-->
    <cpn></cpn>
    <cpn></cpn>
</div>

<script>
    // 1.创建组件构造器对象
    const cpnC = Vue.extend({
        template:`
            <div>
                <h2>我是标题</h2>
                <p>我是内容</p>
            </div>
        `   // 注意这里的 ` 是tab键上面的按钮符号
    })


    // 2.注册组件(全局组件)
    Vue.component('cpn',cpnC)

    var app = new Vue({
        el: '#app',

        // 或者使用下面的这种方式注册组件（局部组件）
        // components:{
        //     cpn:cpnC
        // }
    })
</script>
```

推荐下面这种语法糖的写法：

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

```
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
    <!--使用组件-->
    <cp2></cp2>
</div>

<script>
    // 注册全局组件的语法糖
    Vue.component('cp2',{
        template:`
            <div>
                <h2>我是标题1</h2>
                <p>我是内容1</p>
            </div>
        `
    })

    var app = new Vue({
        el: '#app'
    })
</script>
```

父组件与子组件

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">     <!-- 定义一个 div 元素，id 为 app，将 Vue 实例挂载到这个元素上 -->
    <!--使用组件-->
    <cpn2></cpn2>  <!-- 使用了两个 cpn2 组件，占位符将会被实际的组件内容替代 -->
    <cpn2></cpn2>
</div>
```

```
<script>
    // 1. 创建第一个组件构造器（子组件）
    const cpnC1 = Vue.extend({  <!-- 使用 Vue.extend 方法创建一个组件构造器 cpnC1 -->
        template:`
            <div>
                <h2>我是标题1</h2>
                <p>我是内容1</p>
            </div>
        `                          <!-- 定义模板 template，内容是一个包含标题和段落的 div 元素 -->
    })

    // 2.创建第二个组件构造器（父组件）
    const cpnC2 = Vue.extend({
        template:`
            <div>
                <h2>我是标题2</h2>
                <p>我是内容2</p>
                <cpn1></cpn1>
            </div>
         `,
        components:{          <!-- 在 components 选项中注册 cpn1 组件，这意味着 cpnC2 组件中可以使用 cpn1 组件 -->
            cpn1:cpnC1
        }
    })

    // 创建 Vue 实例并挂载到 #app 元素上
    const app = new Vue({
        el: '#app',
        components:{
            cpn2:cpnC2
        }
    })
```

```
</script>
```

输出：

```
我是标题2
我是内容2

我是标题1
我是内容1

我是标题2
我是内容2

我是标题1
我是内容1
```

组件模板抽离的写法

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
    <!--使用组件-->
    <cp2></cp2>
    <cp2></cp2>
</div>

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

```
</script>

<script>
    // 注册组件
    Vue.component('cp2',{
        template:'#templateID'
    })
    const app = new Vue({
        el: '#app'
    })
</script>
```

或者使用下面的 template 标签，效果与上面是一样的

```
<template id="cpn">
    <div>
        <h2>我是标题1</h2>
        <p>我是内容1</p>
    </div>
</template>
```

为什么组件data必须是函数

组件的 data 属性必须是一个函数，因为每个组件实例需要独立的数据对象。组件对象的 data 属性是一个返回对象的函数，该对象内部保存着组件的数据，同时组件对象也可以包含 methods 等其他属性。

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
```

```
        <!--使用组件-->
        <cp2></cp2>
        <cp2></cp2>
    </div>

    <template id="templateID">
        <div>
            <h2>{{title}}</h2>
            <p>我是内容1</p>
        </div>
    </template>

    <script>
        // 注册组件
        Vue.component('cp2',{
            template:'#templateID',
            data(){
                return {
                    title:'我是标题'
                }
            }
        })

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

输出：

```
我是标题
我是内容1

我是标题
我是内容1
```

data 使用函数，每一次创建组件实例的时候，都会创建一个仅属于当前实例的 data 数据域，这样每一个组件实例都有自己的数据状态。

如果是下面这种方式，组件每一次复用则是公有data域了：

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
     <!--使用组件-->
     <cp2></cp2>
     <cp2></cp2>
</div>

<template id="templateID">
    <div>
        <h2>{{title}}</h2>
        <p>我是内容1</p>
    </div>
</template>

<script>

    //这里的obj是公用数据域（即对象）可以通过多个组件实例共享，主要是因为JavaScript 中的对象和数组是引用类型
    const obj={
           title:'我是标题'
    }

     // 注册组件
     Vue.component('cp2',{
         template:'#templateID',
         data(){
             return obj
         }
     })
```

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

- **JavaScript 对象是引用类型：** 当你创建一个对象 `obj` 并赋值给多个组件的 `data` 选项时，实际上是将同一个对象的引用传递给了这些组件。在 JavaScript 中，对象被存储在内存中，并且当你将对象赋值给多个变量或者组件实例时，它们实际上引用的是同一个对象，而不是对象的副本
- **Vue 组件中的数据共享：** 在 Vue 中，每个组件实例的 `data` 选项应当返回一个对象。当你在多个组件中使用相同的对象字面量（如这里的 `obj` 变量），这些组件实际上在访问和修改同一个对象。因此，当一个组件修改了 `obj` 对象中的属性，其他组件也会立即看到这些修改，因为它们引用的是同一个对象。

父子组件的通信

如何进行父子组件间的通信呢？Vue官方提到

- 通过props向子组件传递数据
  - 方式一：字符串数组，数组中的字符串就是传递数据的载体
  - 方式二：对象，对象可以设置传递时的类型，也可以设置默认值等
- 通过事件向父组件发送数据

通过props向子组件传递数据

方式一案例：我们这里将vue实例当作父组件，创建一个名为cpn的子组件

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<!--vue实例代码,父组件模板-->
```

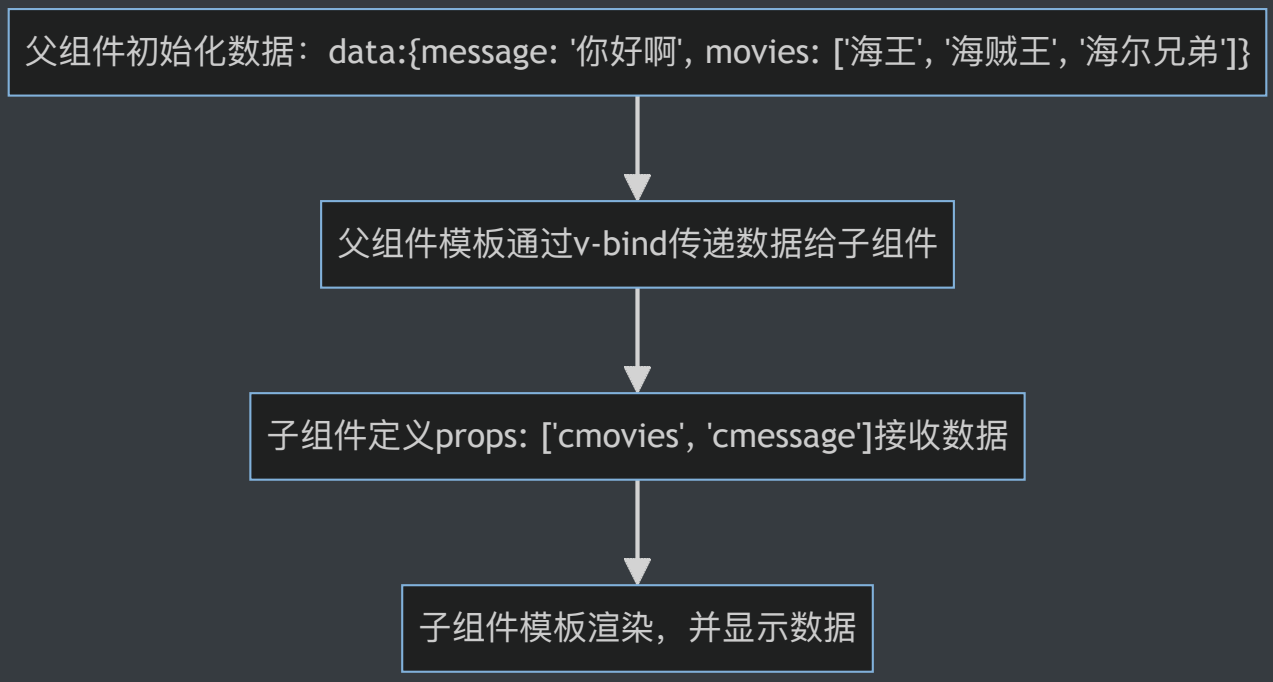
```
<div id="app">  <!--在#app中使用了cpn子组件，并通过v-bind: 将movies和message数据作为props传递给cpn组件 -->

    <cpn v-bind:cmovies="movies" :cmessage="message"></cpn>

    <!--cmovies 这里不支持驼峰写法，比如写成cMovies，这样就获取不到值，如果要写成驼峰，这里要转换写成: c-movies -->
    <!--又比如变量: cheildMyMessage,则转换写成: child-my-message -->
    <!--父级 prop 的更新会向下流动到子组件中，但是反过来则不行。-->
</div>

<!--组件模板-->
    <template id="templateID">
        <div>
            <h2>{{cmovies}}</h2>    <!--这里可以使用驼峰-->
            <p>{{cmessage}}</p>     <!--这里可以使用驼峰-->
        </div>
    </template>

    <script>
        const cpn={
            template: '#templateID',
            props:['cmovies','cmessage'],     // props 属性定义了组件接收的属性名
            data(){
                return{}
            }
        }
        const app = new Vue({
            el: '#app',
            data:{
                message:'你好啊',
                movies:['海王','海贼王','海尔兄弟']
            },
            components:{
                cpn
```

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

输出结果：

```
['海王','海贼王','海尔兄弟']
你好啊
```

请求流程如下（输入 ```**mermaid** 然后敲击回车，即可初始化一张空白图）：



props 数据验证：

- String
- Number
- Boolean
- Array
- Object
- Date

- Function
- Symbol
- 当我们有自定义构造函数时，验证也支持自定义的类型

```
    props:{
            //1.类型限制
            // cmovies:Array,
            // cmessage:String,

            //2.提供一些默认值，以及必传值
            cmessage:{
                type:String,
                default:'aaaaaa',
                required:true
            },

            cmovies:{
              type: Array,
              default(){          // 类型是对象或者数组时，默认值必须是一个函数
                  return []
              }
            }
        },
```

通过事件向父组件发送数据

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

<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<div id="app">
    <!--父组件模板-->
    <cpn @item-click="cpnClick"></cpn>
```

```
    <!--@item-click 是 v-on:item-click 的缩写，监听子组件触发的 item-click 事件，并调用父组件的 cpnClick 方法-->
</div>

<!--子组件模板-->
<template id="templateID">
    <div>
        <button v-for="item in categories" @click="btnClick(item)">
            {{item.name}}
        </button>
    </div>
</template>

<script>
    const cpn={
        template: '#templateID',
        data(){   <!--子组件的数据域-->
            return{
                categories:[
                    {id:'aaa',name:"热门推荐"},
                    {id:'bbb',name:"手机数码"},
                    {id:'ccc',name:"家用家电"},
                    {id:'ddd',name:"电脑办公"},
                ]
            }
        },
        methods:{
            btnClick(item){
                // 向父组件发送事件
                this.$emit('item-click',item)
            }
        }
    }

    const app = new Vue({
```

```
        el: '#app',
        data:{    <!--父组件的数据域-->
            message:'你好啊',
            movies:['海王','海贼王','海尔兄弟']
        },
        components:{
            cpn
        },
        methods: {
            cpnClick(item){
                console.log("cpnClick",item.name);
            }
        }
    })
</script>
```

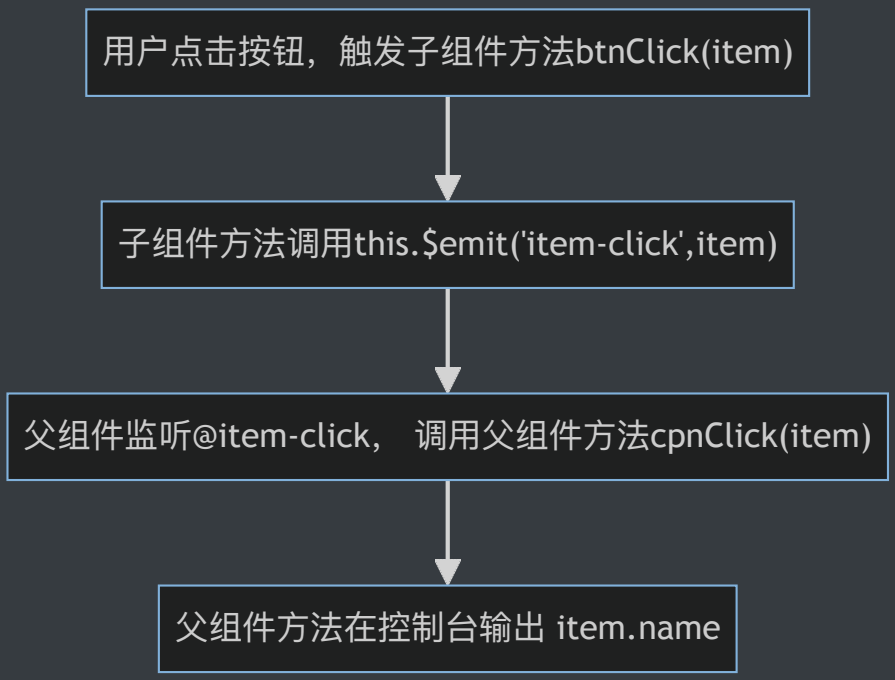
页面样式：

```
热门推荐 手机数码 家用家电 电脑办公
```

比如点击最后一个，浏览器控制台输出：

```
cpnClick 电脑办公
```

请求流程如下（输入 ```**mermaid** 然后敲击回车，即可初始化一张空白图）：



结合双向绑定案例

需求：子组件里修改后的数据再传回给vue实例对象里

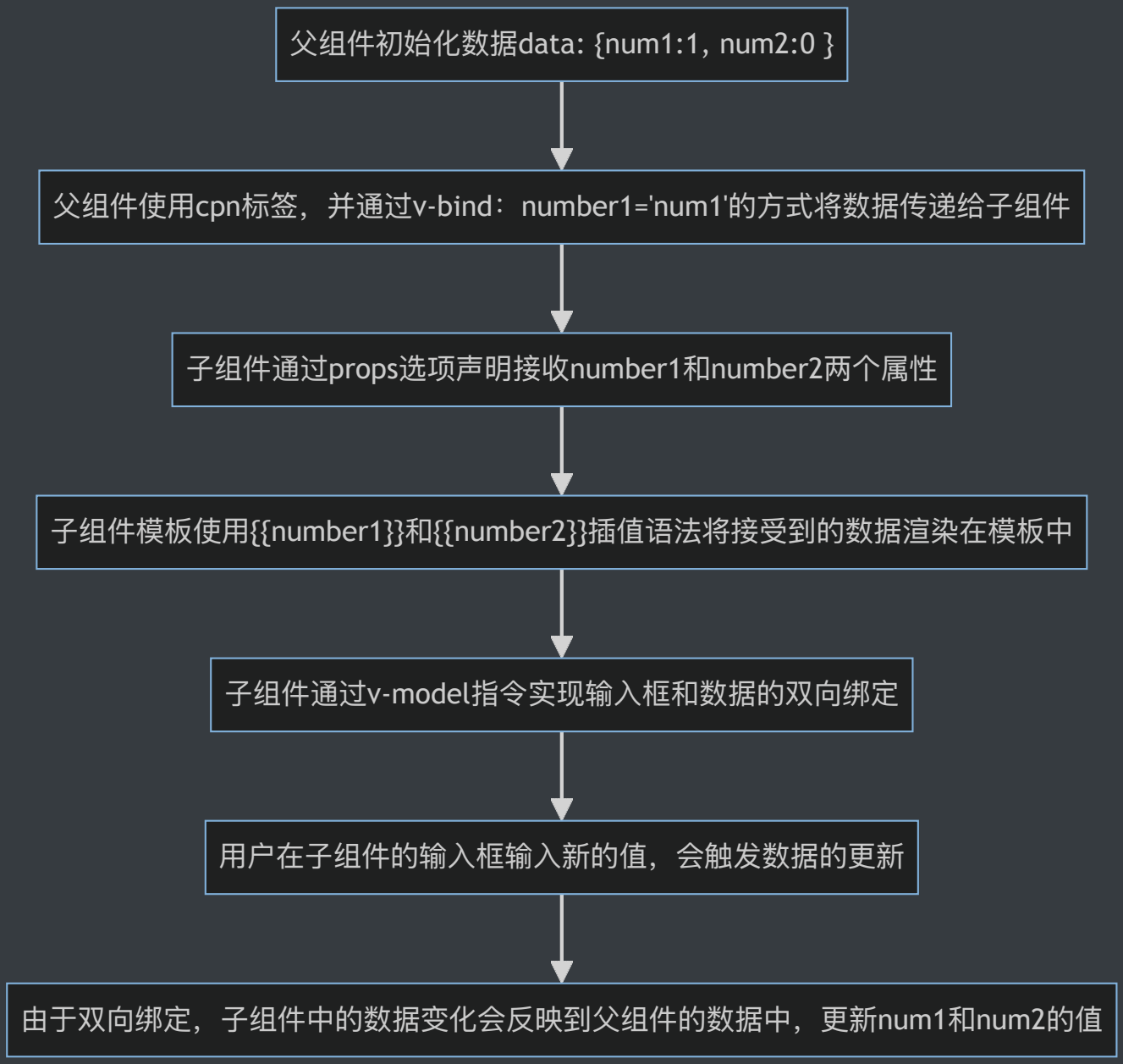
```
<div id="app">
    <cpn :number1="num1" :number2="num2"/>
</div>

<template id="templateID">
    <div>
        <h2>{{number1}}</h2>
        <input type="text" v-model ="number1" >
        <h2>{{number2}}</h2>
        <input type="text" v-model ="number2" >
    </div>
</template>

<script>
    const app = new Vue({
        el: '#app',
        data:{
            num1:1,
            num2:0
```

```
        },
        components:{
            cpn:{
                template: '#templateID',
                props:{
                    number1:Number,
                    number2:Number
                }
            }
        }

    })
</script>
```

浏览器控制台**报错**:

```
[Vue warn]: Avoid mutating a prop directly since the value will be overwritten whenever the parent component re-renders. Instead, use a data or computed property based on the prop's value. Prop being mutated: "number1"
```

请求流程如下（输入 ```**mermaid** 然后敲击回车，即可初始化一张空白图）：



意思是不应该通过子组件里面页面录入的信息去改变number1 ， 这个值应该是通过父组件里面的num1来改变的。

如果在子组件中，页面录入的数据可以改变number1 ， 父组件通过num1 传递到子组件也可以改变number1，容易造成程序的混乱。

解决：创建组件自己的data域

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

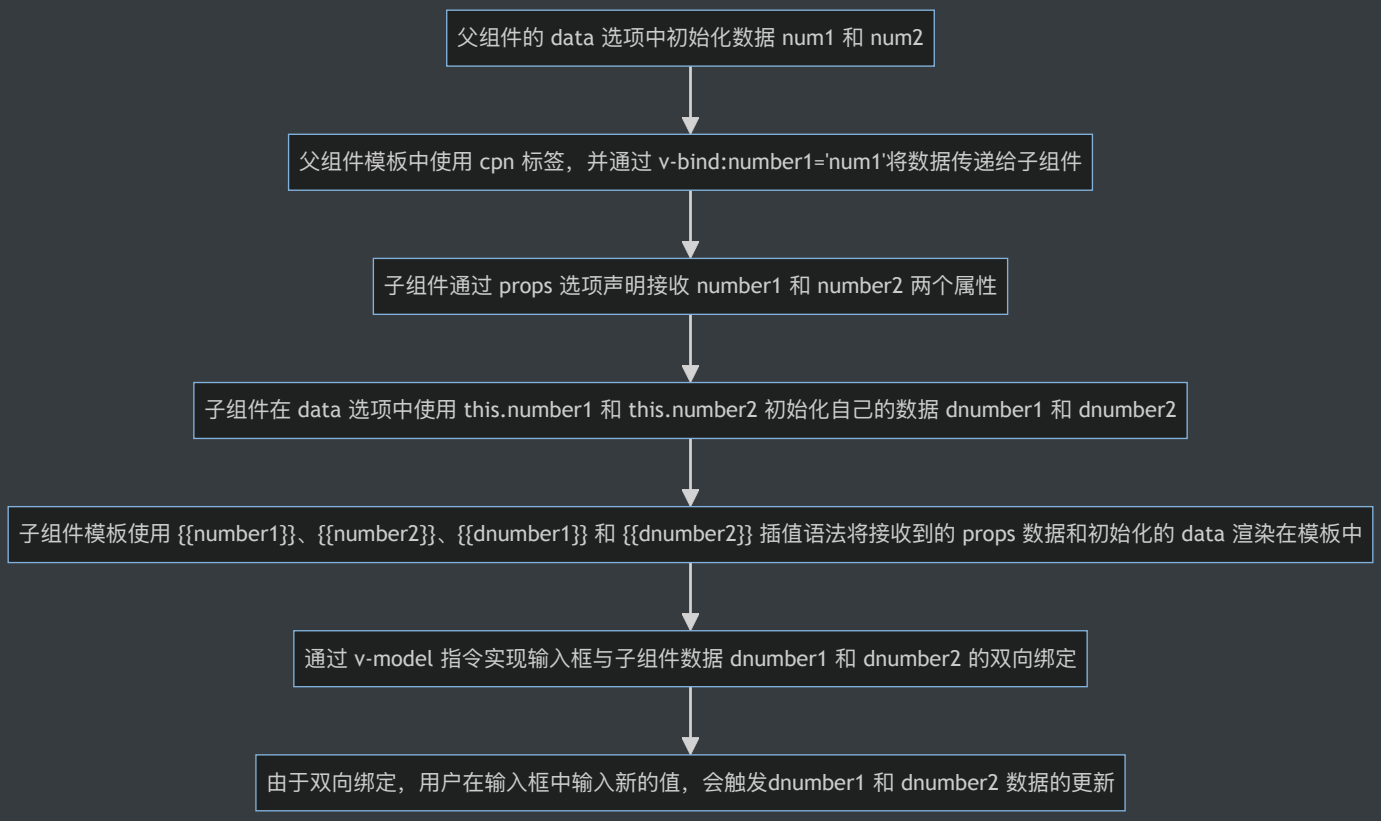
```
<div id="app">
    <cpn :number1="num1" :number2="num2"/>
</div>

<template id="cpn">
    <div>
        <h2>props:{{number1}}</h2>
        <h2>data:{{dnumber1}}</h2>
        <input type="text" v-model ="dnumber1" >
        <h2>props:{{number2}}</h2>
        <h2>data:{{dnumber2}}</h2>
        <input type="text" v-model ="dnumber2" >
    </div>
</template>

<script>
    const app = new Vue({
        el: '#app',
        data:{
            num1:1,
            num2:0
        },
        components:{
            cpn:{
                template: '#cpn',
                props:{
                    number1:Number,
                    number2:Number
                },
                data(){
                    return{
                        dnumber1: this.number1,
                        dnumber2: this.number2
                    }
                }
```

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

页面效果：

> 子组件中 data 里面的此时子组件里面输入框录入值，发现子组件中只有 data 里面的数据发生了变化，props 里面的数据没有变化
>
> 说明子组件页面输入的数据已经和data做了绑定了

请求流程如下：



那么子组件里面录入的值，如何传递回给父组件呢?

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

```
<div id="app">
    <cpn :number1="num1"
         :number2="num2"
    @num1change="num1ch"
    @num2change="num2ch"/>
</div>

<template id="cpn">
    <div>
        <h2>props:{{number1}}</h2>
        <h2>data:{{dnumber1}}</h2>
        <input type="text" :value="dnumber1" @input="num1Input">
        <h2>props:{{number2}}</h2>
        <h2>data:{{dnumber2}}</h2>
        <input type="text" :value="dnumber2" @input="num2Input">
    </div>
</template>


<script>
    const app = new Vue({
        el: '#app',
        data:{
            num1:1,
            num2:0
        },
        methods: {
            num1ch(value){
                console.log(typeof value); //string
                this.num1=parseFloat(value)
            },
            num2ch(value){
                console.log(typeof value);//string
                this.num2=parseFloat(value)
            }
        },
```

```
        components:{
            cpn:{
                template: '#cpn',
                props:{
                    number1:Number,
                    number2:Number
                },
                data(){
                    return{
                        dnumber1: this.number1,
                        dnumber2: this.number2
                    }
                },
                methods:{
                    num1Input(event){
                        // 1.将 input 中的value赋值给dnumber中
                        this.dnumber1 = event.target.value,

                        // 2.为了让父组件可以修改值，发出一个事件
                        this.$emit('num1change',this.dnumber1)

                        // 3.同时修改dnumber2的值
                        this.dnumber2 = this.dnumber1 *2
                        this.$emit('num2change',this.dnumber2)
                    },
                    num2Input(event){
                        this.dnumber2 = event.target.value
                        this.$emit('num2change',this.dnumber2)

                        // 3.同时修改dnumber1的值
                        this.dnumber1 = this.dnumber2 / 2
                        this.$emit('num1change',this.dnumber1)
                    }
                }

            }
```

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

请求流程如下：

父子组件的访问方式

有时候我们需要父组件直接访问子组件，子组件直接访问父组件

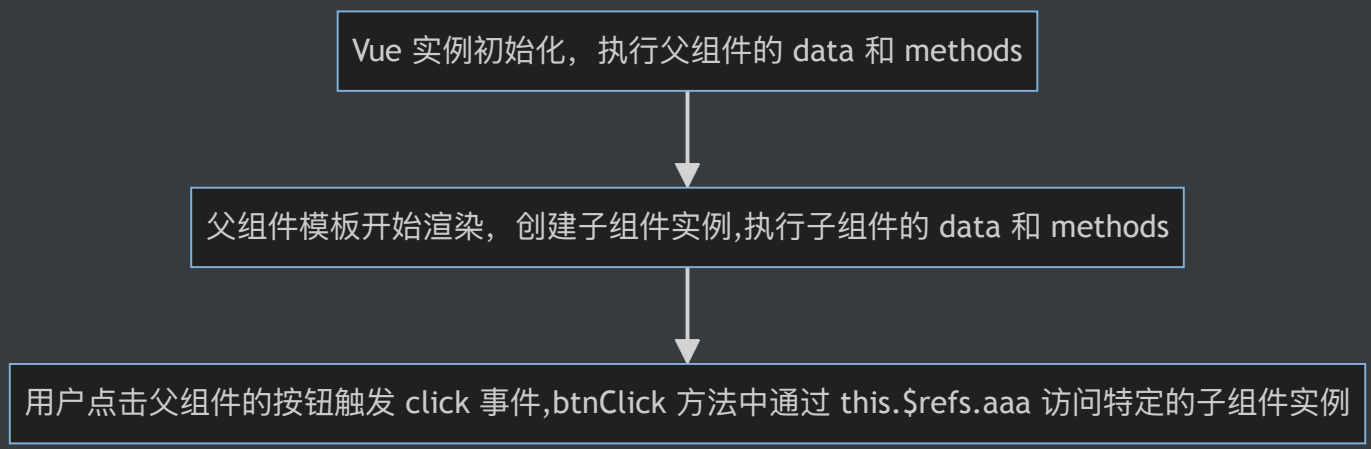
- 父组件访问子组件：使用$children 或$refs
- 子组件访问父组件：使用$parent

父组件访问子组件

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

<div id="app">
    <div>
        <cpn ref="aaa"></cpn>
        <cpn ref="bbb"></cpn>
        <cpn ref="ccc"></cpn>
        <button @click="btnClick">点击按钮获取子组件名称</button>
    </div>
</div>

<template id="templateID">
    <div>我是子组件</div>
</template>


<script>
    const app = new Vue({
        el: '#app',
        data:{

        },
        methods:{
            btnClick(){
                // 1.$children
                console.log(this.$children);
                for(let c of this.$children){
                    console.log(c.name);
                    console.log(c.showMessage());
                }
               console.log("------------------------------");

                // 2.$refs
                console.log(this.$refs.bbb.name);
```

```
            }
        },
        components:{
            cpn:{
                template:'#templateID',
                data(){
                    return{
                        name:'我是子组件的name'
                    }
                },
                methods: {
                    showMessage(){
                        console.log('子组件的方法输出');
                    }
                }
            }
        }
    })
</script>
```

请求流程如下：



子组件访问父组件

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

<div id="app">
```

```
        <cpn ref="aaa"></cpn>
        <cpn ref="bbb"></cpn>
        <cpn ref="ccc"></cpn>
        <button @click="btnClick">点击按钮获取子组件名称</button>
</div>

<template id="templateID">
    <div>
        <div>我是子组件</div>
        <button @click="accessParent">访问父组件</button>
    </div>
</template>
<script>
   const app = new Vue({
       el: '#app',
       data: {
           message: 'Hello from parent'
       },
       methods: {
           btnClick(){
               console.log(this.$refs.aaa.name);
           }
       },
       components: {
           cpn: {
               template: '#templateID',
               data() {
                   return {
                       name: '我被父组件调用啦，输出子组件属性：name'
                   };
               },
               methods: {
                   accessParent() {
                       // 访问父组件的 data
                       console.log(this.$parent.message);
                       // 调用父组件的方法
```

```
                        this.$parent.btnClick();
                    }
                }
            }
        }
    });
</script>
```

slot – 插槽的基本使用

组件的插槽就像是给一个容器留了个小洞，让用户可以根据自己的需要往里面放东西。这种封装方式非常适合，因为它允许我们把组件的通用部分整合在一起，同时又能让用户在使用时灵活地决定具体显示什么内容。简单来说，我们设计好组件的框架，留下一些空间给用户填充自己想要的内容，这样就能满足各种不同的需求了。

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

<div id="app">
    <!--使用组件-->
    <cp2></cp2>
    <cp2><lab>这是特殊的展示</lab></cp2>
</div>

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

<script>
    // 注册组件
```

```
        Vue.component('cp2',{
            template:'#templateID'
        })
        const app = new Vue({
            el: '#app'
        })
    </script>
```

- **父组件模板**：在 `#app` 容器中使用了两次 `cp2` 组件。其中一次没有任何内容（默认使用子组件中的内容），另一次在组件标签之间插入了一个自定义的 `<lab>` 标签及其内容。
- **子组件模板**：在子组件的模板中，使用了 `<slot></slot>` 标签。这个标签表示一个插槽，用于放置父组件传入的内容。如果父组件没有传入任何内容，插槽会保持为空。

父组件模板的所有东西都会在父级作用域内编译；子组件模板的所有东西都会在子级作用域内编译。即组件是封装的逻辑单元，每个组件都有独立的作用域，父组件和子组件的模板内容互不干扰。为了在父子组件之间通讯，通常使用props 和 ¥ emit事件，或者使用Vuex进行状态管理。

插槽是一个特殊情况，插槽内容在父级作用域内编译，但渲染在子组件中。

作用域插槽

作用域插槽（Scoped Slots）是 Vue.js 提供的一种高级插槽技术。与普通插槽不同，作用域插槽允许子组件将数据传递给父组件的插槽内容，从而使父组件能够根据子组件的数据自定义渲染内容。简而言之，作用域插槽使父组件可以“作用”于子组件的数据。

案例：

- 子组件包括一组数据，比如：pLanguages:['JavaScript','Python','Swify','Go','C++']
- 需要在多个页面展示：某些页面水平展示，某些页面列表展示，某些页面直接展示为数组

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

```
<div id="app">
    <cpn></cpn>
    <cpn>
        <!--目的是获取子组件中的pLanguages-->
        <template slot-scope="slot">
            <span v-for="item in slot.xxx">{{item}} # </span>
        </template>
    </cpn>
    <cpn>
        <template slot-scope="slot">
            <span>{{slot.xxx.join(' * ')}}</span>
        </template>
    </cpn>
 </cp2>
</div>

<!--子组件模板 -->
<script type="text/x-template" id = "templateID">
    <div>
        <slot :xxx="pLanguages">
            <ul>
                <li v-for="item in pLanguages">{{item}}</li>
            </ul>
        </slot>

    </div>
</script>

<script>
    const app = new Vue({
        el: '#app',
        components:{
            cpn:{
                template:'#templateID',
                data(){
```

```
                    return {
                        pLanguages:
['JavaScript','Python','Swify','Go','C++']
                    }
                }
            }
        }
    })
</script>
```

- **父组件模板**：包含三个 `cpn` 组件实例，其中两个通过作用域插槽获取子组件的数据 `pLanguages` 。
- **子组件模板**：使用 `<slot>` 标签，绑定 `pLanguages` 数据到插槽中，提供给父组件使用
- **子组件数据**：在子组件中定义了一个数据 `pLanguages` ，包含多种编程语言
- **作用域插槽使用**：

  第一个 `cpn` 实例没有自定义插槽内容，所以会显示默认内容，即一个包含编程语言列表的 `<ul>` 。

  第二个 `cpn` 实例自定义了插槽内容，通过 `slot-scope="slot"` 获取子组件的 `pLanguages` 数据，并使用 `v-for` 指令渲染每个编程语言，结果为每个语言后面加上 `#` 。

  第三个 `cpn` 实例主要是展示如何将子组件的数据以特定的格式连接成一个字符串并渲染出来。

四、ES模块化的导入和导出

模块化是将代码进行拆分以便重复利用。类似于 Java 中的导包，使用一个包前必须先导包。在 JavaScript 中没有包的概念，而是使用模块。在 ES6 中，每个模块就是一个文件，文件中的变量、函数、对象等在外部无法直接访问。模块功能主要通过两个命令实现： `export` 和 `import` 。 `export` 用于暴露模块中的内容，使外部可以读取，包括基本类型、变量、函数、数组和对象等； `import` 用于导入模块。

案例：

index.html 、aaa.js 、bbb.js 都在同一个文件夹下

index.html

```
<body>
    <script src="aaa.js" type="module"></script>
    <script src="bbb.js" type="module"></script>
</body>

<!--这个 HTML 文件包含两个 `<script>` 标签，每个标签引用一个 JavaScript 文件并指定 `type="module"`，这意味着这些文件是 ES6 模块-->
```

aaa.js

```
var name ='小明'
var age =18
var flag = true   // 定义了三个变量

function sum(num1,num2){  //定义了一个函数，用于计算两个数的和
    return num1+num2;
}

if(flag){ // 判断条件，输出 `sum(20, 30)` 的结果到控制台
    console.log("aaa.js的输出："+sum(20, 30));
}

// 1.导出方式一：使用 `export { flag, sum }` 导出变量和函数
export {
    flag, sum
}

// 2.导出方式二：使用 `export var` 语法直接导出变量 `num1` 和 `height`
```

```
export var num1 =1000;
export var height =1.888

// 3.导出函数/类 : 使用 `export function` 和 `export class` 导出函数 `mul` 和类 `Person`
export function mul(num1,num2) {
    return num1 * num2
}

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

// 4、某些情况下，一个模块包含某个功能，我们并不希望给这个功能命名，而且让导入者可以自己来命名，使用 export default
const  address ='北京市'
export  default  address
```

bbb.js

```
// 1. 使用 `import { ... }` 语法从 `aaa.js` 导入特定变量和函数
import {flag,sum} from "./aaa.js";

if(flag){  // 检查 `flag`，输出 "小明是天才，哈哈哈" 和 `sum(20, 30)` 的结果
    console.log('小明是天才，哈哈哈');
    console.log("bbb.js的输出: "+sum(20, 30));
}

//2.导入 `num1` 和 `height`，并在控制台输出
import  {num1,height} from "./aaa.js";
console.log(num1);
console.log(height);

//3.导入 export 的函数 `mul` 和类 `Person`，然后调用它们
```

```
import  {mul,Persion} from "./aaa.js";

console.log(mul(30, 50));

const p = new Persion();
p.run();

// 4.导入 export default 中的内容，并在控制台输出
import addr from "./aaa.js";
console.log(addr);

// 5.使用 `import * as aaa` 导入 `aaa.js` 中所有的导出内容，并访问
`aaa.flag
import  * as aaa from './aaa.js'

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

浏览器控制台输出：

```
bbb.js:10 aaa.js的输出: 50
bbb.js:5 小明是天才，哈哈哈
bbb.js:6 bbb.js的输出: 50
bbb.js:11 1000
bbb.js:12 1.888
bbb.js:17 1500
aaa.js:29 在奔跑
bbb.js:24 北京市
bbb.js:29 true
```

如果出现下面错误：

```
Access to script at 'file:///C:/Users/d1348681/Grey/Invoke/aa/bbb.js'
from origin 'null' has been blocked by CORS policy: Cross origin
requests are only supported for protocol schemes: http, data,
isolated-app, chrome-extension, chrome-untrusted, https, edge.
```

则安装 Node.js，可以使用 `http-server` 模块。在你的项目文件夹中打开命令行，然后运行

```
npm install -g http-server
http-server
```

这时会启动一个本地 HTTP 服务器，你可以通过浏览器访问 `http://localhost:8080/index.html` 来查看你的网页，即可避免 CORS 的问题得到上面的控制台输出结果。

五、webpack

webpack 可以看作是一个打包工具，它可以分析你的项目结构，然后找到js 模块以及一些浏览器不能直接执行的一些语言比如 Less 、TypeScript 等，并将其转换和打包为一个合法的格式以供浏览器使用。webpack从3.0之后还担负起了优化项目的功能。

1、安装 Node.js 和 npm

官网下载node -x86.msi。安装完成后，可以通过以下命令验证安装是否成功：

```
node -v
npm -v
```

2、创建项目目录

```
mkdir my-webpack-project
cd my-webpack-project
```

3、初始化 npm 项目

在项目目录中运行以下命令来初始化一个新的 npm 项目。这会创建一个 `package.json` 文件：

```
npm init -y
```

4、安装 Webpack 和 Webpack CLI

接下来，安装 Webpack 和 Webpack CLI 作为开发依赖：

```
npm install --save-dev webpack webpack-cli
```

5、创建 Webpack 配置文件

在项目根目录中创建一个名为 `webpack.config.js` 的配置文件。这个文件用于配置 Webpack 的打包行为：

```
const path = require('path');
module.exports = {
 entry: './src/index.js', // 入口文件
 output: {
   filename: 'bundle.js', // 输出文件名
   path: path.resolve(__dirname, 'dist'), // 输出目录
 },
 mode: 'development', // 模式，可以是 'development' 或 'production'
};
```

6、创建项目目录结构

在项目根目录中创建 `src` 目录，并在 `src` 目录中创建 `index.js` 文件：

```
mkdir src
echo console.log('Hello, Webpack!'); > src/index.js
```

7、配置 npm 脚本

在 `package.json` 文件中添加一个 `scripts` 部分，用于运行 Webpack：

```
{
 "name": "my-webpack-project",
 "version": "1.0.0",
 "description": "",
 "main": "index.js",
 "scripts": {
   "build": "webpack"   //这里是具体的修改内容
 },
 "keywords": [],
 "author": "",
 "license": "ISC",
 "devDependencies": {
   "webpack": "^5.x.x",
   "webpack-cli": "^4.x.x"
 }
}
```

8、运行 Webpack

现在，你可以运行以下命令来打包你的项目：

```
npm run build
```

如果一切正常，你应该会看到 Webpack 生成的 `dist` 目录，其中包含 `bundle.js` 文件。这个文件是 Webpack 将 `src/index.js` 打包后的输出结果。

9、创建 HTML 文件

为了在浏览器中查看打包结果，可以在项目根目录中创建一个简单的 `index.html` 文件，并引用 `dist/bundle.js`：

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Webpack Project</title>
</head>
<body>
    <h1>vue 的标题</h1>
  <script src="./dist/bundle.js"></script>
</body>
</html>
```

到这里，在浏览器打开index.html 即可看到控制台输出： `Hello，Webpack!`

webpack中使用css文件

`loader` 是 `webpack` 中一个非常核心的概念

`webpack` 用来做什么呢?

- 在我们之前的实例中，我们只要是用 `webpack` 来处理我们写的 js 代码，并且 `webpack` 会自动处理 js 之间的依赖
- 但是，在开发中我们不仅仅有基本的js代码处理，我们也需要 `css` 、图片，也包括一些高级的将 ES6 转成ES5代码，将 `TypeScript` 转成ES5代码，将scss、less转成scc，将.jsx、.vue文件转成js文件等等
- 对应 `webpack` 本身的能力来说，对于这些转化是不支持的
- 那怎么办呢? 给 `webpack` 扩展对应的 `loader` 就可以啦

`loader` 使用过程:

步骤一：通过npm安装需要使用的loader (官方文档)

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

步骤二：在 `webpack.config.js` 中的modules关键字下进行配置

```
module.exports = {
  ......
  module: {
    rules: [
            //css-loader 只负责将css文件进行加载
            //style-loader负责将样式添加到DOM中
            //使用多个loader时，是从右向左读取
            { test: /\.css$/, use: ['style-loader','css-loader'] },
          ],
  },
  ......
};module.exports = {
  ......
  module: {
    rules: [
            //css-loader 只负责将css文件进行加载
            //style-loader负责将样式添加到DOM中
            //使用多个loader时，是从右向左读取
            { test: /\.css$/, use: ['style-loader','css-loader'] },
          ],
  },
  ......
};
```

步骤三、在 `src` 文件夹中创建 `styles.css` 文件，并添加一些简单的样式：

```
body {
 background-color: lightblue;
}
h1 {
 color: white;
 text-align: center;
}
```

在 index.js 引入 css 文件：

```
import './styles.css';
console.log('Webpack is working!');
```

步骤四、最后执行 npx webpack ，在浏览器打开index.html 即可看到带有蓝色背景和白色文本的网页。

webpack中使用图片

步骤一：安装依赖

```
npm install --save-dev url-loader file-loader
```

步骤二：更新 webpack.config.js 配置

```
const path = require('path');
module.exports = {
 entry: './src/index.js', // 入口文件
 output: {
```

```
    filename: 'bundle.js', // 输出文件名
    path: path.resolve(__dirname, 'dist'), // 输出目录
  },
  module: {
   rules: [
                //css-loader 只负责将css文件进行加载
                //style-loader负责将样式添加到DOM中
                //使用多个loader时，是从右向左读取
                { test: /\.css$/, use: [
                        {loader:'style-loader'},
                        {loader:'css-loader',options:{esModule:false}} //
解决背景图乱码，生成多余文件问题
                        ]
                },

                {
                  test: /\.(png|jpg|gif|jpeg)$/i,
                  use: [
                      {
                          loader: 'url-loader',
                          options: {
                              // 当加载的图片，小于limit时，会将图片编译成
base64字符串形式，使用url-loader
                              // 当加载的图片，大于limit时，需要使用file-
loader模块进行加载
                              limit: 8192, // 文件大小限制 (8 KB)
                              name:'images/[name].[hash:8].[ext]',
                              esModule: false, // 兼容CommonJS模块
                          },
                      },
                  ],
                }
            ],
},
  mode: 'development', // 模式，可以是 'development' 或 'production'
};
```

其中，

img：文件要打包到的文件夹

name：获取图片原来的名字，放在该位置

hash8:为了防止图片名称冲突，依然使用hash，但是我们只保留8位

ext：使用图片原来的扩展名

步骤三：添加图片文件

在 `src` 文件夹中创建一个 `images` 文件夹，并添加一张图片，例如 `1.png` 。

步骤四：使用图片

在 `src/styles.css` 文件中使用这张图片：

```
body {
 background-color: lightblue;
}
h1 {
 color: white;
 text-align: center;
}
.background {
 background-image: url('./images/1.png');
 background-size: cover;
 height: 400px;
 width: 100%;
}
```

在 src/index.js 文件中添加一个元素来应用背景图片：

```
import './styles.css';
console.log('Webpack is working!');
const div = document.createElement('div');
div.className = 'background';
document.body.appendChild(div);
```

步骤五：构建项目

```
npx webpack
```

引入vue.js

安装vue:

```
npm install vue --save
```

查看 vue 的版本号

```
npm list vue
```

一、在main.js中初步引入vue

步骤一：修改index.html

```
<div id="app">
    <h2>{{message}}</h2>
</div>

<script src="./dist/bundle.js"></script>
```

index.js

```
import { createApp } from 'vue'

createApp({data(){
    return {
        message:"Hello Webpack"
    }
}}).mount('#app')
```

步骤二：`webpack.config.js` 下新增 `resolve`：

```
const path = require('path');
module.exports = {
 ......................参考上面内容.........................
 resolve: {
    alias: {
      'vue$': 'vue/dist/vue.esm-browser.js' // 使用包含编译器的 Vue 构建版本
    },
    extensions: ['.*', '.js', '.vue', '.json']
 },
 mode: 'development', // 模式，可以是 'development' 或 'production'
};
```

步骤三： 构建项目

```
npx webpack
```

二、将index.js样式封装成组件

步骤一：修改index.html

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

<script src="./dist/bundle.js"></script>
```

修改index.js

```
import { createApp } from 'vue'

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

```
}).mount('#app')
```

上面的问题是模板和js太耦合了，如何进行拆分呢，看下面的最终方案。

三、将样式拆分到App.vue

步骤一：安装依赖

```
npm install --save-dev vue-loader vue-template-compiler
```

步骤二：创建vue组件

在 `src` 目录下创建 `components` 目录，创建 `App.vue` 和 `Cpn.vue` 两个文件

`App.vue`：

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

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

    export default {
        name: "App",
        //注册子组件
        components:{
            Cpn
        },
```

```
        data(){
            return {
                message:'Hello Webpack',
                name:'Grey'
            }
        },
        methods:{
            btnClick(){
            }
        }
    }
</script>

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

Cpn.vue：

```
<template>
    <div>
        <h2>我是CPN组件的标题</h2>
        <p>我是CPN组件的内容</p>
        {{name}}
    </div>
</template>

<script>
    export default {
        name: "Cpn",
        data(){
            return{
                name:'CPN组件的name'
            }
```

```
        }
    }
</script>

<style scoped>

</style>
```

修改 index.js

```
import { createApp } from 'vue'
import App from './components/App.vue'

createApp({
    template:'<App/>',
    components:{
        App
    }
}).mount('#app')
```

index.html保持不变。

步骤三：更新 webpack.config.js 配置

```
const path = require('path');
const { VueLoaderPlugin } = require('vue-loader')


module.exports = {
 entry: './src/index.js', // 入口文件
 output: {
   filename: 'bundle.js', // 输出文件名
   path: path.resolve(__dirname, 'dist'), // 输出目录
 },
 module: {
```

```
  rules: [
              //css-loader 只负责将css文件进行加载
              //style-loader负责将样式添加到DOM中
              //使用多个loader时，是从右向左读取
              { test: /\.css$/, use: [
                     {loader:'style-loader'}, //style-lader 应该在css-
loader之前
                     {loader:'css-loader',options:{esModule:false}} //
解决背景图乱码，生成多余文件问题
                     ]
              },

              {
                test: /\.(png|jpg|gif|jpeg)$/i,
                use: [
                    {
                        loader: 'url-loader',
                        options: {
                            // 当加载的图片，小于limit时，会将图片编译成
base64字符串形式
                            // 当加载的图片，大于limit时，需要使用file-
loader模块进行加载
                            limit: 8192000, // 文件大小限制 (8 KB)
                            name:'images/[name].[hash:8].[ext]',
                            esModule: false, // 兼容CommonJS模块
                        },
                    },
                ],
              },

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

        ],
```

```
  },

    plugins: [
      new VueLoaderPlugin()
    ],
    resolve: {
      alias: {
        'vue$': 'vue/dist/vue.esm-browser.js' // 使用包含编译器的 Vue 构建版
本
      },
      extensions: ['.*', '.js', '.vue', '.json']
    },

    mode: 'development', // 模式，可以是 'development' 或 'production'
};
```

步骤四：构建项目

```
npx webpack
```

通过浏览器打开src 目录下的index.html 即可看到页面效果。

步骤五：打包html文件到dist目录

```
npm install html-webpack-plugin
```

修改 `webpack.config.js` 文件中的plugins部分内容：

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

module.exports = {
   ..............重复内容,参考上面代码.....................
 plugins:[
       new HtmlWebpackPlugin({
           template:'index.html'
       })
      ]
 }
```

通过浏览器打开dist 目录下的index.html 可看到同样页面效果。

步骤六：通过本地服务器运行项目

你可以使用一个本地开发服务器来查看你的项目，如 `webpack-dev-server` 插件。首先安装它：

```
npm install --save-dev webpack-dev-server
```

```
npm list webpack-cli   查看版本
```

`webpack.config.js` 文件添加下面内容：

```
const path = require('path');
const { VueLoaderPlugin } = require('vue-loader')
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
 entry: './src/index.js', // 入口文件
 ..............重复内容,参考上面代码.....................
  devServer: {
```

```
        static: {
          directory: path.join(__dirname, './'),   //为哪一个文件夹提供本地
    服务，默认是根文件夹，我们这里填写./
        },
        compress: true,
        port: 9000,
        inline:true // 页面实时刷新
      },
    };
```

`package.json` 下添加 `dev`：

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

执行：

```
    npm run dev
```

访问 `http://localhost:9000` 查看同样的结果

配置文件的分离

`webpack.config.js` 可以分离到 `config` 文件夹下的三个文件

```
-- config
  -- base.config.js   // 存放基本配置，开发阶段和生产阶段都需要的配置
  -- dev.config.js    // 存放只在开发阶段用到的配置
  -- prod.config.js   // 存放只在生产阶段用到的配置
```

现在需要将三个配置文件联系到一起，安装 `webpack-merge` 插件，

```
npm config set registry https://registry.npmjs.org/   解决安装插件慢的问题

npm install --save-dev webpack-merge
```

`base.config.js`

```
const path = require('path');
const { VueLoaderPlugin } = require('vue-loader')
const HtmlWebpackPlugin = require('html-webpack-plugin');

module.exports = {
 entry: './src/index.js', // 入口文件
 output: {
   filename: 'bundle.js', // 输出文件名
   path: path.resolve(__dirname, '../dist'), // 输出目录
 },
 module: {


  rules: [
              //css-loader 只负责将css文件进行加载
              //style-loader负责将样式添加到DOM中
              //使用多个loader时，是从右向左读取
              { test: /\.css$/, use: [
                     {loader:'style-loader'}, //style-lader 应该在css-loader之前
```

```
                    {loader:'css-loader',options:{esModule:false}} //
解决背景图乱码，生成多余文件问题
                    ]
            },

            {
              test: /\.(png|jpg|gif|jpeg)$/i,
              use: [
                  {
                      loader: 'url-loader',
                      options: {
                          // 当加载的图片，小于limit时，会将图片编译成
base64字符串形式
                          // 当加载的图片，大于limit时，需要使用file-
loader模块进行加载
                          limit: 8192000, // 文件大小限制 (8 KB)
                          name:'images/[name].[hash:8].[ext]',
                          esModule: false, // 兼容CommonJS模块
                      },
                  },
              ],
            },


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

        ],

},
  plugins: [
    new VueLoaderPlugin(),
    new HtmlWebpackPlugin({template:'index.html'}),
  ],

  resolve: {
```

```
    alias: {
      'vue$': 'vue/dist/vue.esm-browser.js' // 使用包含编译器的 Vue 构建版本
    },
    extensions: ['.*', '.js', '.vue', '.json']
  },

  mode: 'development', // 模式，可以是 'development' 或 'production'
};
```

dev.config.js

```
//dev.config.js
let webpackMerge = require('webpack-merge');
let baseConfig = require('./base.config')

module.exports = webpackMerge(baseConfig,{
    devServer:{ //这个配置只在开发阶段有用，打包生成最终代码的时候，这个配置就没有用了
          static: {
            directory: path.join(__dirname, './'),
          },
          compress: true,
          port: 9000,
    }
})
```

prod.config.js

```
// prod.config.js
let UglifyWebpackPlugin = require('uglifyjs-webpack-plugin')
let webpackMerge = require('webpack-merge') //1.引入webpackMerge
let baseConfig = require('./base.config')   //2.引入base.config.js

module.exports = webpackMerge(baseConfig,{  //3.使用webpackMerge进行合并
    plugins:[
        new UglifyWebpackPlugin() //打包阶段才需要压缩js代码
    ]
})
```

在 package.json 文件里的scripts中配置命令：

```
"scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "build": "webpack --config config/prod.config.js",
    "dev": "webpack server --config config/dev.config.js"
  },
```

- 在执行 `npm run build` 的时候，会使用prod.config.js配置文件
- 在执行 `npm run dev` 的时候，会使用dev.config.js配置文件

另外，注意一下在 base.config.js 中 output 的配置

```
module.exports = {
  entry: './src/main.js',
  output: {
   //注意这个地方是 ../dist，如果直接写 dist ，它会在build文件夹下生成dist文件夹
    path: path.join(__dirname, '../dist'),
    filename: 'bundle.js',
  }
}
```

六、路由

认识路由

路由是一个网路工程里面的术语。

路由（routing）就是通过互联网的网络把信息从源地址传输到目的地址的活动。--维基百科

在生活中，我们有没有听过路由的概念呢？当然了，路由器嘛

路由器是做什么的?

路由器提供了两种机制：路由和转送

- 路由是决定数据包从来源到目的地的路径
- 转送将输入端的数据转移到合适的输出端

路由中有一个非常重要的概念叫路由表

- 路由表本质上就是一个映射表，决定了数据包的指向

后端路由阶段

早期的网站开发整个HTML页面是由服务器来渲染的

- 服务器直接生产渲染好对应的HTML页面，返回给客户端进行展示

但是，一个网站这么多页面服务器如何处理呢?

- 一个页面有自己对应的网址，也就是URL

- URL会发送到服务器，服务器会通过正则对该URL进行匹配，并且最后交给一个Controller进行处理
- Controller进行各种处理，最终生成HTML或者数据，返回给前端
- 这就完成了一个IO操作

上面的这种操作，就是后端路由

- 当我们页面中需要请求不同的路径内容时，交给服务器来进行处理，服务器渲染好整个页面，并且将页面返回给客户端
- 这种情况下渲染好的页面，不需要单独加载任何的js和css，可以直接交给浏览器展示，这样也有利于SEO的优化

后端路由的缺点

- 一种情况是整个页面的模块由后端人员来编写和维护的
- 另一种情况是前端开发人员如果要开发页面，需要通过PHP和Java等语言来编写页面代码
- 而且通常情况下HTML代码和数据以及对应的逻辑会混子一起，编写和维护都是非常糟糕的事情

前端分离阶段

随着Ajavx的出现，有了前后端分离的开发模式

后端只提供API来返回数据，前端通过Ajava获取数据，并且可以通过JavaScript将数据渲染到页面中。

这样做最大的优点就是前后端责任的清晰，后端专注于数据上，前端专注于交互和可视化上

并且当移动端（ios/Android）出现后，后端不需要进行任何处理，依然使用之前的一套API即可

目前很大的网站依然采用这种模式开发

单页面富应用阶段

其实SPA最主要的特地就是在前后端分离的基础上加了一层前端路由

也就是前端来维护一套路由规则

前端路由的核心是改变URL，但是页面不进行整体的刷新。

怎么实现呢?

URL的hash

URL的hash也就是锚点（#），本质上是改变window.location的href属性

我们可以通过直接赋值location.hash来改变href，但是页面不刷新

```
location.gref
"http://192.167.3.123:8080/examplels/urlChange/"
location.hash='aaa'
"aaa"
```

HTML5的history模式：pushState

```
location.gref
"http://192.167.3.123:8080/examplels/urlChange/"
history.pushState({},'','/foo')
undefinded
history.back()   // 出栈
```

vue-router基本使用

vue-router是vue.js官方的路由插件，它和vue.js是深度集成的，适合用于构建单页面应用。

步骤一：安装vue-router

```
npm install vue-router --save
```

步骤二：在模块化工程中使用它（因为是一个插件，所以可以通过Vue.use()来安装路由功能）

- 第一步：导入路由对应，并且调用Vue.use(VueRouter)

- 第二步：创建路由实例，并且传入路由映射配置
- 第三步：在Vue实例中挂载创建的路由实例

使用vue-router的步骤

- 第一步：创建路由组件
- 第二步：配置路由映射：组件和路径映射关系
- 第三步：使用路由：通过 和

案例

创建项目：

```
vue init webpack vueclie2Router
```

目录结构：

```
src
  --assets
  --components
    About.vue
    Home.vue
  --router
    index.js
  App.vue
  main.js
```

`index.js`

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

```
import Home from "../components/Home";
import About from "../components/About";

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

//2、创建VueRouter对象
const routes =[
  {
    // 设置首页默认路由
    path:'',
    // redirect 重定向
    redirect :'/home'
  },
  {
    path:'/home',
    component:Home
  },
  {
    path:'/about',
    component:About
  }
]
const router = new VueRouter({
  // 配置路由和组件之间的应用关系
  routes,
  mode:'history'    // 修改为history模式
})

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

App.vue

```
<template>
  <div id="app">

    <!--router-link 是vue-router中已经内置的组件，它会被渲染成一个<a>标签-->
    <!--router-view 根据当前的路径，动态渲染出不同的组件-->
    <!--网页的其他内容，比如顶部的标题/导航，或者底部的一些版权信息等会和<router-view>处于同一个等级-->
    <!--在路由切换时，切换的时<router-view>挂载的组件，其他内容不会发生变化-->
    <router-link to="/home">首页</router-link>
    <router-link to="/about">关于</router-link>
    <router-view></router-view>
  </div>
</template>

<script>
export default {
  name: 'App'
}
</script>

<style>
</style>
```

About.vue

```
<template>
    <div>
      <h2>我是关于</h2>
      <p>我是关于内容。。。</p>
    </div>
</template>

<script>
    export default {
        name: "About"
    }
```

```
</script>

<style scoped>

</style>
```

Home.vue

```
<template>
   <div>
     <h2>我是首页</h2>
     <p>我是首页内容。。。</p>
   </div>
</template>

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

<style scoped>
</style>
```

main.js

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

Vue.config.productionTip = false

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

页面效果：点击关于会展示下面内容

```
[首页] [关于]
我是关于
我是关于内容。。。
```

router-link补充

在前面的 中，我们只是使用了一个属性：to，用于指定跳转的路径

还有一些其他属性：

- tag : tag 可以指定之后渲染成什么组件，比如下面的代码会被渲染成一个
- 元素，而不是

```
<router-link to='/home' tag='li'>
```

- replace : replace 在浏览器不会留下history记录，所有指定replace 的情况下，后退键返回不能返回到上一个页面中。

```
<router-link to='/home' tag='li' replace>
```

- active-class:当对应的路由匹配成功时，会自动给当前元素设置一个 `router-link-active` 的class，设置active-class可以修改默认的名称
  - 在进行高亮显示的导航菜单或底部tabbar时，会使用到该类
  - 但是通常不会修改类的属性，会直接使用默认的 `router-link-acive` 即可

  浏览器打开 `Elements` 查看

  ```
  <a href="/about" class="router-link-exact-active router-link-active" aria-current="page">关于</a>
  ```

  比如修改名称 `router-link-active` 为 `active`，这样就可以针对该class配置单独的css样式：

  ```
  const router = new VueRouter({
    // 配置路由和组件之间的应用概念性
    routes,
    mode:'history',
    linkActiveClass:'active'
  })
  ```

通过代码跳转路由

修改 `App.vue`

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

<script>
export default {
```

```
    name: 'App',
    methods:{
      homeClick(){
        this.$router.push('/home')
        // 浏览器不留访问历史：this.$router.replace('/home')
        console.log('homeclick');
      },
      aboutClick(){
        this.$router.push('/about')
        console.log('aboutClick');
      }
    }
  }
  </script>

  <style>
  </style>
```

如果浏览器控制台报错：

```
vue-router.esm.js?fe87:2065 Uncaught (in promise)
NavigationDuplicated: Avoided redundant navigation to current
location: "/home".
```

则在 `router\index.js` 中添加下面代码：

```
const originalPush = VueRouter.prototype.push

VueRouter.prototype.push = function push(location) {
  return originalPush.call(this, location).catch(err => err)
}
```

位置在：

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

const originalPush = VueRouter.prototype.push

VueRouter.prototype.push = function push(location) {
  return originalPush.call(this, location).catch(err => err)
}


//2、创建VueRouter对象
const routes =[
 {
    path:'',
    // redirect 重定向
    redirect :'/home'
  },
  {
    path:'/home',
    component:Home
  },
  {
    path:'/about',
    component:About
  }
]
```

vue-router基本使用

首页有个参数，如何传递到关于页面页面呢?

`App.vue`：

```
<template>
  <div id="app">
    <router-link to="/home">首页</router-link>
```

```
      <router-link v-bind:to="'/about/'+userId">关于</router-link>

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

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

  <style>
  </style>
```

index.js

```
  //2、创建VueRouter对象
  const routes =[
   {
      path:'',
      // redirect 重定向
      redirect :'/home'
    },
    {
      path:'/home',
      component:Home
    },
    {
      path:'/about/:aaa',
```

```
      component:About
    }
  ]
```

About.vue：

```
<template>
    <div>
      <h2>我是关于</h2>
      <p>我是关于内容。。。</p>
      {{userId}}
    </div>
</template>

<script>
    export default {
        name: "About",
      computed:{
          userId(){
            // 当前哪一个路由处于活跃状态，拿到的就是哪一个路由
            // 注意不是router
            return this.$route.params.aaa
          }
      }
    }
</script>

<style scoped>

</style>
```

页面效果：点击关于，展示zhangsan

```
[首页] [关于]
我是关于
我是关于内容。。。

zhangsan
```

认识路由的懒加载

官方给出了解释

- 当打包构建应用时，Javascript 包会变得非常大，影响页面加载
- 如果我们能把不同路由对应的组件分割成不同的代码块，然后当路由被访问的时候才加载对应组件，这样就更加高效了

路由懒加载多了什么?

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

vue路由懒加载写法:

```
import Home from "../components/Home";
import About from "../components/About";

// 懒加载写法:
const Home = () => import('../components/Home')
const About = () => import('../components/About')
```

vue-router嵌套路由

修改 `router/index.js` :

```
{
    path:'/home'
```

```
        component:Home,
        children:[
            {
                path:'news',
                component:HomeNews
                // 在要展示子路由的界面配置上 <router-view></router-view>
                // <router-link to="/home/news/">新闻</router-link>
            },
            {
                path:'message',
                component:HomeMessage
            }
        ]
    }
```

vue-router参数传递

传递参数主要有两种类型：params 和 query

vue-router导航守卫

我们来考虑一个需求：在一个SPA应用中，如何改变网页的标题呢?

使用导航守卫就可以达到这个目标。

- vue-router提供的导航守卫主要是用来监听监听路由的进入和离开的
- vue-router提供了beforeEach 和afterEach的钩子函数，它们会在路由即将改变前和改变后触发

我们可以利用beforeEach来完成标题的修改

- 首先，我们可以在钩子当中定义一些标题，可以利用meta来定义
- 其次，利用导航守卫，修改我们的标题

```
    //2、创建VueRouter对象
```

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

  },
  {
    path:'/home',
    component:Home,
    meta:{
      titel:'首页'
    }
  },
  {
    path:'/about/:aaa',
    component:About,
    meta: {
      titel: '关于'
    }
  }
]
const router = new VueRouter({
  // 配置路由和组件之间的应用概念性
  routes,
  mode:'history',
  linkActiveClass:'active'
})

// 前置守卫 (guard)
router.beforeEach((to,from ,next) =>{
  // 从 from 跳转到 to
  document.title = to.matched[0].meta.titel
  next() //确保要调用 next 方法，否则钩子就不会被 resolved
})
```

如果是后置钩子，也就是afterEach，不需要主动调用next()函数

```
//后置钩子（hook）
router.afterEach((to,from) =>{
    console.log('------');
})
```

keep-alive

keep-alive 是 vue 内置的一个组件，可以使被包含的组件保留状态，或避免重新选择。

- include 字符串或正则表达式，只有匹配的组件会被缓存
- exclude 字符串或正则表达式，任何匹配的组件都不会被缓存

router-view 也是一个组件，如果直接被包在keep-alive 里面，所有路径匹配到的视图组件都会被缓存。

```
<keep-alive>
      <router-view>
        <!--所有路径匹配到的视图组件都会被缓存-->
      </router-view>
</keep-alive>
```

七、Promise

什么情况下会用到Promise?

一般情况下是有异步操作的，使用Promis对这个异步操作进行封装

new -> 构造函数（1.保存了一些状态信息 2.执行传入的函数）

在执行传入的回调函数时，会传入两个参数，resolve，reject ，本身又是函数

```
new Promise((resolve,reject) =>{
    setTimeout(()=>{
        // 成功的时候调用resolve
        // resolve('Hello World')

        // 失败的时候调用reject
        reject('error message')
    },1000)

}).then((data) =>{
    // 成功处理代码
    ...
}).catchs((err) =>{
    console.log(err);
})
```

Promise三种状态

首先，当我们开发中有异步操作时，就可以给异步操作包装一个Promis

异步操作之后会有三种状态：

- pending:等待状态。比如正在进行网络请求，或者定时器没有到时间
- fulfill:满足状态，当我们主动回调了resolve时，就处于该状态，并且会回调.then()
- reject:拒绝状态，当我们主动回调了reject时，就处于该状态，并且会回调.catch

八、Vuex

Vuex 是一个专为 Vue.js 应用程序开发的状态管理模式。

- 它采用集中式存储管理应用的所有组件的状态，并以相应的规则保证状态以一种可预测的方式发生变化。
- Vuex 也集成到Vue 的官方调试工具 `devtools extension`,提供了诸如零配置的 `time-travel` 调试、状态快照导入导出等高级调试功能

状态管理到底是什么?

- 其实，你可以简单的将其看成把需要多个组件共享的变量全部存储在一个对象里面
- 然后，将这个对象放在顶层的Vue实例中，让其他组件可以使用
- 那么，多个组件就可以共享这个对象中的所有变量属性了，但是这样做不到响应式
- Vuex 就是为了提供这样一个多组件间共享状态的响应式插件

安装插件:

```
npm install vuex --save
```

使用插件，新建一个store的文件夹，创建index.js

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

// 1.安装插件
Vue.use(Vuex)

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

```
        decrement(state){
            state.counter--
        }
    },
    actions:{...},
    getters:{...},
    modules:{...}
})

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

在 App.vue 页面上修改公共变量:

```
<script>
    export default{
  name:'App',
    ...
    methods:{
        addition(){
            this.$store.commit('increment')
        },
        subtraction(){
            this.$store.commit('decrement')
        }
    }
}
</script>
```