day01

Vue 是什么?

- **Vue (读音 /vju:/，类似于 view) 是一套用于构建用户界面的渐进式框架**
- vue 的核心库只关注视图层，不仅易于上手，还便于与第三方库或既有项目整合

使用Vue将helloworld 渲染到页面上

```
<body>
    <div id="app">
        <div> {{msg}} </div>
        <div> {{1 + 2}} </div>
        <div> {{msg + '----' + 123}}  </div>
    </div>
    <script type="text/javascript" src="js/vue.js"></script>

    <script type="text/javascript">
        var vm = new Vue({
            el: '#app',
            data: {
                msg: 'Hello Vue'
            }
        });
    </script>

</body>
```

1、需要提供标签用于填充数据

2、引入vue.js库文件

3、使用vue的语法做功能

获取元素 如果是id 传入# 如果是类则传入 .

4、把vue提供的数据填充到标签里面

data 中 存放要渲染到页面上的数据

msg 中 存储的值 hello Vue

会将 HTML中 插值语法中的msg 替换掉

指令

- 本质就是自定义属性
- Vue中指定都是以 v- 开头

v-cloak

- 防止页面加载时出现闪烁问题

```
<style type="text/css">
 /*
```

```
    1、通过属性选择器 选择到 带有属性 v-cloak的标签  让他隐藏
 */
  [v-cloak]{
    /* 元素隐藏    */
    display: none;
  }
  </style>
<body>
  <div id="app">
    <!-- 2、 让带有插值 语法的   添加 v-cloak 属性
         在 数据渲染完场之后，v-cloak 属性会被自动去除，
         v-cloak一旦移除也就是没有这个属性了  属性选择器就选择不到该标签
         也就是对应的标签会变为可见
    -->
    <div  v-cloak  >{{msg}}</div>
  </div>
  <script type="text/javascript" src="js/vue.js"></script>
  <script type="text/javascript">
    var vm = new Vue({
      //  el   指定元素 id 是 app 的元素
      el: '#app',
      //  data  里面存储的是数据
      data: {
        msg: 'Hello Vue'
      }
    });
</script>
</body>
</html>
```

v-text

- v-text指令用于将数据填充到标签中，作用于插值表达式类似，但是没有闪动问题
- 如果数据中有HTML标签会将html标签一并输出
- 注意：此处为单向绑定，数据对象上的值改变，插值会发生变化；但是当插值发生变化并不会影响数据对象的值

```
<div id="app">
    <!--
        注意:在指令中不要写插值语法  直接写对应的变量名称
        在 v-text 中 赋值的时候不要在写 插值语法
        一般属性中不加 {{}}  直接写 对应 的数据名
    -->
    <p v-text="msg"></p>
    <p>
        <!-- Vue  中只有在标签的 内容中 才用插值语法 -->
        {{msg}}
    </p>
</div>

<script>
    new Vue({
```

```
        el: '#app',
        data: {
            msg: 'Hello Vue.js'
        }
    });

</script>
```

v-html

- 用法和v-text 相似 但是他可以将HTML片段填充到标签中
- 可能有安全问题, 一般只在可信任内容上使用 `v-html`，**永不**用在用户提交的内容上
- 它与v-text区别在于v-text输出的是纯文本，浏览器不会对其再进行html解析，但v-html会将其当html标签解析后输出。

```
<div id="app">
    <p v-html="html"></p> <!-- 输出：html标签在渲染的时候被解析 -->

     <p>{{message}}</p> <!-- 输出：<span>通过双括号绑定</span> -->

    <p v-text="text"></p> <!-- 输出：<span>html标签在渲染的时候被源码输出</span> -->
</div>
<script>
    let app = new Vue({
    el: "#app",
    data: {
        message: "<span>通过双括号绑定</span>",
        html: "<span>html标签在渲染的时候被解析</span>",
        text: "<span>html标签在渲染的时候被源码输出</span>",
    }
 });
</script>
```

v-pre

- 显示原始信息跳过编译过程
- 跳过这个元素和它的子元素的编译过程。
- **一些静态的内容不需要编译加这个指令可以加快渲染**

```
    <span v-pre>{{ this will not be compiled }}</span>
    <!--  显示的是{{ this will not be compiled }}  -->
    <span v-pre>{{msg}}</span>
     <!--   即使data里面定义了msg这里仍然是显示的{{msg}}  -->
<script>
    new Vue({
        el: '#app',
        data: {
            msg: 'Hello Vue.js'
        }
    });

</script>
```

v-once

- 执行一次性的插值【当数据改变时，插值处的内容不会继续更新】

```
  <!-- 即使data里面定义了msg 后期我们修改了 仍然显示的是第一次data里面存储的数据即 Hello Vue.js  -->
     <span v-once>{{ msg}}</span>
<script>
    new Vue({
        el: '#app',
        data: {
            msg: 'Hello Vue.js'
        }
    });
</script>
```

双向数据绑定

- 当数据发生变化的时候，视图也就发生变化
- 当视图发生变化的时候，数据也会跟着同步变化

v-model

- **v-model**是一个指令，限制在 `<input>、<select>、<textarea>、components` 中使用

```
 <div id="app">
      <div>{{msg}}</div>
      <div>
          当输入框中内容改变的时候，   页面上的msg   会自动更新
        <input type="text" v-model='msg'>
      </div>
  </div>
```

mvvm

- MVC 是后端的分层开发概念； MVVM是前端视图层的概念，主要关注于 视图层分离，也就是说：MVVM把前端的视图层，分为了 三部分 Model, View , VM ViewModel
- m model
    - 数据层 Vue 中 数据层 都放在 data 里面
- v view 视图
    - Vue 中 view 即 我们的HTML页面
- vm （view-model） 控制器 将数据和视图层建立联系
    - vm 即 Vue 的实例 就是 vm

v-on

- 用来绑定事件的
- 形式如：v-on:click 缩写为 @click;

```
<div id="app">
    <div>{{num}}</div>
    <div>
        <!--通过v-on来给元素绑定事件   -->
                注意点:  在HTML中使用data里面的数据不需要加this
        <button v-on:click='num++'>点击</button>

        <!-- v-on 可以用 @ 来替代 -->
        <button @click='num++'>点击1</button>
        <!--
          开发中许多事件处理逻辑会更为复杂,
          所以直接把 JavaScript 代码写在 v-on 指令中是不可行的。
          因此 v-on 还可以接收一个需要调用的方法名称
        -->
                                    这里事件处理名称要和 methods 中一致
        <button @click='handle'>点击2</button>
        <!--
          注意:  这里虽然写的是 handle()并不会调用函数
          只有点击的时候才会调用函数   -->

        <button @click='handle()'>点击3</button>
    </div>
</div>

<script type="text/javascript">

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

      num: 0

    },  // 注意点:  这里不要忘记加逗号
    // methods  中 主要是定义一些函数
    methods: {

      handle: function(){
        // 这里的this是Vue的实例对象+
        console.log(this === vm)
        //    在函数中 想要使用data里面的数据 一定要加this
        this.num++;
      }
```

这2种方案能将 num 值改变但是不推荐这么写

v-on事件函数中传入参数

```
<body>
    <div id="app">
        <div>{{num}}</div>
        <div>
            <!-- 如果事件直接绑定函数名称，那么默认会传递事件对象作为事件函数的第一个参数 -->
            <button v-on:click='handle1'>点击1</button>
            <!-- 2、如果事件绑定函数调用，那么事件对象必须作为最后一个参数显示传递，
                 并且事件对象的名称必须是$event
            -->
            <button v-on:click='handle2(123, 456, $event)'>点击2</button>
        </div>
    </div>
    <script type="text/javascript" src="js/vue.js"></script>
    <script type="text/javascript">
        var vm = new Vue({
            el: '#app',
            data: {
                num: 0
            },
            methods: {
                handle1: function(event) {
                    console.log(event.target.innerHTML)
                },
                handle2: function(p, p1, event) {
                    console.log(p, p1)
                    console.log(event.target.innerHTML)
                    this.num++;
                }
            }
        });
    </script>
```

事件修饰符

- 在事件处理程序中调用 `event.preventDefault()` 或 `event.stopPropagation()` 是非常常见的需求。
- Vue 不推荐我们操作DOM 为了解决这个问题，Vue.js 为 `v-on` 提供了**事件修饰符**
- 修饰符是由点开头的指令后缀来表示的

```
<!-- 阻止单击事件继续传播 -->
<a v-on:click.stop="doThis"></a>

<!-- 提交事件不再重载页面 -->
<form v-on:submit.prevent="onSubmit"></form>

<!-- 修饰符可以串联    即阻止冒泡也阻止默认事件 -->
<a v-on:click.stop.prevent="doThat"></a>

<!-- 只当在 event.target 是当前元素自身时触发处理函数 -->
<!-- 即事件不是从内部元素触发的 -->
<div v-on:click.self="doThat">...</div>
```

使用修饰符时，顺序很重要；相应的代码会以同样的顺序产生。因此，用 v-on:click.prevent.self 会阻止所有的点击，而 v-on:click.self.prevent 只会阻止对元素自身的点击。

按键修饰符

- 在做项目中有时会用到键盘事件，在监听键盘事件时，我们经常需要检查详细的按键。Vue 允许为 `v-on` 在监听键盘事件时添加按键修饰符

```
<!-- 只有在 `keyCode` 是 13 时调用 `vm.submit()` -->
<input v-on:keyup.13="submit">

<!-- -当点击enter 时调用 `vm.submit()` -->
<input v-on:keyup.enter="submit">

<!--当点击enter或者space时  时调用 `vm.alertMe()`   -->
<input type="text" v-on:keyup.enter.space="alertMe" >

常用的按键修饰符
.enter =>    enter键
.tab => tab键
.delete（捕获“删除”和“退格”按键）=>  删除键
.esc => 取消键
.space =>  空格键
.up =>  上
.down =>  下
.left =>  左
.right =>  右

<script>
    var vm = new Vue({
        el:"#app",
        methods: {
              submit:function(){},
              alertMe:function(){},
        }
    })

</script>
```

自定义按键修饰符别名

- 在Vue中可以通过`config.keyCodes`自定义按键修饰符别名

```
<div id="app">
    预先定义了keycode 116（即F5）的别名为f5，因此在文字输入框中按下F5，会触发prompt方法
    <input type="text" v-on:keydown.f5="prompt()">
</div>

<script>

    Vue.config.keyCodes.f5 = 116;
```

```
    let app = new Vue({
        el: '#app',
        methods: {
            prompt: function() {
                alert('我是 F5! ');
            }
        }
    });
</script>
```

v-bind

- v-bind 指令被用来响应地更新 HTML 属性
- v-bind:href 可以缩写为 :href;

```
<!-- 绑定一个属性 -->
<img v-bind:src="imageSrc">

<!-- 缩写 -->
<img :src="imageSrc">
```

绑定对象

- 我们可以给v-bind:class 一个对象，以动态地切换class。
- 注意：v-bind:class指令可以与普通的class特性共存

```
1、 v-bind 中支持绑定一个对象
    如果绑定的是一个对象 则 键为 对应的类名  值 为对应data中的数据
<!--
    HTML最终渲染为 <ul class="box textColor textSize"></ul>
    注意:
        textColor, textSize  对应的渲染到页面上的CSS类名
        isColor, isSize  对应vue data中的数据  如果为true 则对应的类名 渲染到页面上


        当 isColor 和 isSize 变化时, class列表将相应的更新,
        例如, 将isSize改成false,
        class列表将变为 <ul class="box textColor"></ul>
-->

<ul class="box" v-bind:class="{textColor:isColor, textSize:isSize}">
    <li>学习Vue</li>
    <li>学习Node</li>
    <li>学习React</li>
</ul>
  <div v-bind:style="{color:activeColor,fontSize:activeSize}">对象语法</div>

<sript>
var vm= new Vue({
    el:'.box',
    data:{
```

```
        isColor:true,
        isSize:true,
        activeColor:"red",
        activeSize:"25px",
    }
})
</sript>
<style>

    .box{
        border:1px dashed #f0f;
    }
    .textColor{
        color:#f00;
        background-color:#eef;
    }
    .textSize{
        font-size:30px;
        font-weight:bold;
    }
</style>
```

绑定class

```
2、  v-bind 中支持绑定一个数组    数组中classA和 classB 对应为data中的数据

这里的classA  对用data 中的  classA
这里的classB  对用data 中的  classB
<ul class="box" :class="[classA, classB]">
    <li>学习Vue</li>
    <li>学习Node</li>
    <li>学习React</li>
</ul>
<script>
var vm= new Vue({
    el:'.box',
    data:{
        classA:‘textColor‘,
        classB:‘textSize‘
    }
})
</script>
<style>
    .box{
        border:1px dashed #f0f;
    }
    .textColor{
        color:#f00;
        background-color:#eef;
    }
    .textSize{
        font-size:30px;
        font-weight:bold;
```

```
    }
</style>
```

绑定对象和绑定数组 的区别

- 绑定对象的时候 对象的属性 即要渲染的类名 对象的属性值对应的是 data 中的数据
- 绑定数组的时候数组里面存的是data 中的数据

绑定style

```
 <div v-bind:style="styleObject">绑定样式对象</div>'

<!-- CSS 属性名可以用驼峰式 (camelCase) 或短横线分隔 (kebab-case，记得用单引号括起来)     -->
 <div v-bind:style="{ color: activeColor, fontSize: fontSize,background:'red' }">内联样式
</div>

<!--组语法可以将多个样式对象应用到同一个元素 -->
<div v-bind:style="[styleObj1, styleObj2]"></div>


<script>
    new Vue({
      el: '#app',
      data: {
        styleObject: {
          color: 'green',
          fontSize: '30px',
          background:'red'
        },
        activeColor: 'green',
        fontSize: "30px"
      },
      styleObj1: {
             color: 'red'
       },
       styleObj2: {
            fontSize: '30px'
       }

</script>
```

分支结构

v-if 使用场景

- 1- 多个元素 通过条件判断展示或者隐藏某个元素。或者多个元素
- 2- 进行两个视图之间的切换

```
<div id="app">
        <!--  判断是否加载，如果为真，就加载，否则不加载-->
        <span v-if="flag">
           如果flag为true则显示,false不显示!
```

```
        </span>
</div>

<script>
    var vm = new Vue({
        el:"#app",
        data:{
            flag:true
        }
    })
</script>

------------------------------------------------------------

    <div v-if="type === 'A'">
       A
    </div>
  <!-- v-else-if紧跟在v-if或v-else-if之后    表示v-if条件不成立时执行-->
    <div v-else-if="type === 'B'">
       B
    </div>
    <div v-else-if="type === 'C'">
       C
    </div>
  <!-- v-else紧跟在v-if或v-else-if之后-->
    <div v-else>
       Not A/B/C
    </div>

<script>
    new Vue({
      el: '#app',
      data: {
        type: 'C'
      }
    })
</script>
```

v-show 和 v-if的区别

- v-show本质就是标签display设置为none，控制隐藏
  - v-show只编译一次，后面其实就是控制css，而v-if不停的销毁和创建，故v-show性能更好一点。
- v-if是动态的向DOM树内添加或者删除DOM元素
  - v-if切换有一个局部编译/卸载的过程，切换过程中合适地销毁和重建内部的事件监听和子组件

循环结构

v-for

- 用于循环的数组里面的值可以是对象，也可以是普通元素

```
<ul id="example-1">
   <!-- 循环结构-遍历数组
    item 是我们自己定义的一个名字   代表数组里面的每一项
    items对应的是 data中的数组-->
  <li v-for="item in items">
    {{ item.message }}
  </li>

</ul>
<script>
 new Vue({
  el: '#example-1',
  data: {
    items: [
      { message: 'Foo' },
      { message: 'Bar' }
    ],

  }
})
</script>
```

- **不推荐**同时使用 `v-if` 和 `v-for`
- 当 `v-if` 与 `v-for` 一起使用时，`v-for` 具有比 `v-if` 更高的优先级。

```
   <!--  循环结构-遍历对象
        v 代表    对象的value
        k  代表对象的 键
        i  代表索引
    --->
     <div v-if='v==13' v-for='(v,k,i) in obj'>{{v + '---' + k + '---' + i}}</div>

<script>
 new Vue({
  el: '#example-1',
  data: {
    items: [
      { message: 'Foo' },
      { message: 'Bar' }
    ],
    obj: {
        uname: 'zhangsan',
        age: 13,
        gender: 'female'
    }
  }
})
</script>
```

- key 的作用
    - **key来给每个节点做一个唯一标识**

- **key的作用主要是为了高效的更新虚拟DOM**

```
<ul>
  <li v-for="item in items" :key="item.id">...</li>
</ul>
```

案例选项卡

1、 HTML 结构

```
`
    <div id="app">
        <div class="tab">
            <!--  tab栏  -->
            <ul>
                <li class="active">apple</li>
                <li class="">orange</li>
                <li class="">lemon</li>
            </ul>
              <!--  对应显示的图片 -->
            <div class="current"><img src="img/apple.png"></div>
            <div class=""><img src="img/orange.png"></div>
            <div class=""><img src="img/lemon.png"></div>
        </div>
    </div>


`
```

2、 提供的数据

```
        list: [{
                    id: 1,
                    title: 'apple',
                    path: 'img/apple.png'
                }, {
                    id: 2,
                    title: 'orange',
                    path: 'img/orange.png'
                }, {
                    id: 3,
                    title: 'lemon',
                    path: 'img/lemon.png'
                }]
```

3、 把数据渲染到页面

- 把tab栏 中的数替换到页面上
  - 把 data 中 title 利用 v-for 循环渲染到页面上
  - 把 data 中 path利用 v-for 循环渲染到页面上

```
    <div id="app">
        <div class="tab">
            <ul>
                  <!--
                    1、绑定key的作用 提高Vue的性能
                    2、 key 需要是唯一的标识 所以需要使用id, 也可以使用index ,
                        index 也是唯一的
                    3、 item 是 数组中对应的每一项
                    4、 index 是 每一项的 索引
                -->
                   <li :key='item.id' v-for='(item,index) in list'>{{item.title}}</li>
              </ul>
              <div  :key='item.id' v-for='(item, index) in list'>
                    <!-- :  是 v-bind 的简写   绑定属性使用 v-bind -->
                    <img :src="item.path">
              </div>
        </div>
    </div>
<script>
    new  Vue({
        //  指定 操作元素 是 id 为app 的
        el: '#app',
            data: {
                list: [{
                    id: 1,
                    title: 'apple',
                    path: 'img/apple.png'
                }, {
                    id: 2,
                    title: 'orange',
                    path: 'img/orange.png'
                }, {
                    id: 3,
                    title: 'lemon',
                    path: 'img/lemon.png'
                }]
            }
    })

</script>
```

4、 给每一个tab栏添加事件,并让选中的高亮

- 4.1 、让默认的第一项tab栏高亮
  - tab栏高亮 通过添加类名active 来实现 （CSS active 的样式已经提前写好）

        - 在data 中定义一个 默认的 索引 currentIndex 为 0
        - 给第一个li 添加 active 的类名
            - 通过动态绑定class 来实现 第一个li 的索引为 0 和 currentIndex 的值刚好相等
            - currentIndex === index 如果相等 则添加类名 active 否则 添加 空类名
- 4.2 、让默认的第一项tab栏对应的div 显示
    - 实现思路 和 第一个 tab 实现思路一样 只不过 这里控制第一个div 显示的类名是 current

```
  <ul>
       <!-- 动态绑定class    有 active    类名高亮   无 active    不高亮-->
       <li  :class='currentIndex==index?"active":""'
           :key='item.id' v-for='(item,index) in list'
           >{{item.title}}</li>
  </ul>
    <!-- 动态绑定class    有 current   类名显示   无 current   隐藏-->
  <div :class='currentIndex==index?"current":""'

       :key='item.id' v-for='(item, index) in list'>
        <!-- :  是 v-bind 的简写    绑定属性使用 v-bind -->
        <img :src="item.path">
  </div>

<script>
    new  Vue({
        el: '#app',
            data: {
                currentIndex: 0, // 选项卡当前的索引  默认为 0
                list: [{
                    id: 1,
                    title: 'apple',
                    path: 'img/apple.png'
                }, {
                    id: 2,
                    title: 'orange',
                    path: 'img/orange.png'
                }, {
                    id: 3,
                    title: 'lemon',
                    path: 'img/lemon.png'
                }]
            }
    })

</script>
```

- 4.3 、点击每一个tab栏 当前的高亮 其他的取消高亮
    - 给每一个li添加点击事件
    - 让当前的索引 index 和 当前 currentIndex 的 值 进项比较
    - 如果相等 则当前li 添加active 类名 当前的 li 高亮 当前对应索引的 div 添加 current 当前div 显示 其他隐藏

```
    <div id="app">
        <div class="tab">
            <ul>
                <!--  通过v-on 添加点击事件    需要把当前li 的索引传过去
                -->
                <li v-on:click='change(index)'
                    :class='currentIndex==index?"active":""'
                    :key='item.id'
                    v-for='(item,index) in list'>{{item.title}}</li>
            </ul>
            <div :class='currentIndex==index?"current":""'
                 :key='item.id' v-for='(item, index) in list'>
                <img :src="item.path">
            </div>
        </div>
    </div>

<script>
    new  Vue({
        el: '#app',
            data: {
                currentIndex: 0, // 选项卡当前的索引  默认为 0
                list: [{
                    id: 1,
                    title: 'apple',
                    path: 'img/apple.png'
                }, {
                    id: 2,
                    title: 'orange',
                    path: 'img/orange.png'
                }, {
                    id: 3,
                    title: 'lemon',
                    path: 'img/lemon.png'
                }]
            },
            methods: {
                change: function(index) {
                    // 通过传入过来的索引来让当前的  currentIndex  和点击的index 值 相等
                    //  从而实现 控制类名
                    this.currentIndex = index;
                }
            }

    })

</script>
```

day02

Vue常用特性

表单基本操作

- 获取单选框中的值
  - 通过v-model

```
    <!--
        1、 两个单选框需要同时通过v-model 双向绑定 一个值
        2、 每一个单选框必须要有value属性  且value 值不能一样
        3、 当某一个单选框选中的时候 v-model  会将当前的 value值 改变 data 中的 数据

        gender 的值就是选中的值，我们只需要实时监控他的值就可以了
    -->
   <input type="radio" id="male" value="1" v-model='gender'>
   <label for="male">男</label>

   <input type="radio" id="female" value="2" v-model='gender'>
   <label for="female">女</label>

<script>
    new Vue({
         data: {
             // 默认会让当前的 value 值为 2 的单选框选中
                gender: 2,
            },
    })

</script>
```

- 获取复选框中的值
  - 通过v-model
  - 和获取单选框中的值一样
  - 复选框 `checkbox` 这种的组合时 data 中的 hobby 我们要定义成数组 否则无法实现多选

```
    <!--
        1、 复选框需要同时通过v-model 双向绑定 一个值
        2、 每一个复选框必须要有value属性  且value 值不能一样
        3、 当某一个单选框选中的时候 v-model  会将当前的 value值 改变 data 中的 数据

        hobby 的值就是选中的值，我们只需要实时监控他的值就可以了
    -->

<div>
   <span>爱好: </span>
```

```
   <input type="checkbox" id="ball" value="1" v-model='hobby'>
   <label for="ball">篮球</label>
   <input type="checkbox" id="sing" value="2" v-model='hobby'>
   <label for="sing">唱歌</label>
   <input type="checkbox" id="code" value="3" v-model='hobby'>
   <label for="code">写代码</label>
 </div>
<script>
    new Vue({
         data: {
                // 默认会让当前的 value 值为 2 和 3 的复选框选中
                hobby: ['2', '3'],
            },
    })
</script>
```

- 获取下拉框和文本框中的值
  - 通过v-model

```
   <div>
      <span>职业: </span>
       <!--
            1、 需要给select  通过v-model 双向绑定 一个值
            2、 每一个option  必须要有value属性  且value 值不能一样
            3、 当某一个option选中的时候 v-model  会将当前的 value值 改变 data 中的 数据
             occupation 的值就是选中的值，我们只需要实时监控他的值就可以了
        -->
       <!-- multiple  多选 -->
      <select v-model='occupation' multiple>
          <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>
         <!-- textarea 是 一个双标签   不需要绑定value 属性的  -->
        <textarea v-model='desc'></textarea>
  </div>
<script>
    new Vue({
         data: {
                // 默认会让当前的 value 值为 2 和 3 的下拉框选中
                 occupation: ['2', '3'],
                 desc: 'nihao'
            },
    })
</script>
```

表单修饰符

- .number 转换为数值
  - 注意点:

    - 当开始输入非数字的字符串时，因为Vue无法将字符串转换成数值
    - 所以属性值将实时更新成相同的字符串。即使后面输入数字，也将被视作字符串。
- .trim 自动过滤用户输入的首尾空白字符
    - 只能去掉首尾的 不能去除中间的空格
- .lazy 将input事件切换成change事件
    - .lazy 修饰符延迟了同步更新属性值的时机。即将原本绑定在 input 事件的同步逻辑转变为绑定在 change 事件上
- 在失去焦点 或者 按下回车键时才更新

```
<!-- 自动将用户的输入值转为数值类型 -->
<input v-model.number="age" type="number">

<!--自动过滤用户输入的首尾空白字符    -->
<input v-model.trim="msg">

<!-- 在“change”时而非“input”时更新 -->
<input v-model.lazy="msg" >
```

自定义指令

- 内置指令不能满足我们特殊的需求
- Vue允许我们自定义指令

Vue.directive 注册全局指令

```
<!--
  使用自定义的指令，只需在对用的元素中，加上'v-'的前缀形成类似于内部指令'v-if', 'v-text'的形式。
-->
<input type="text" v-focus>
<script>
// 注意点:
//   1、 在自定义指令中  如果以驼峰命名的方式定义 如  Vue.directive('focusA',function(){})
//   2、 在HTML中使用的时候 只能通过 v-focus-a 来使用

// 注册一个全局自定义指令 v-focus
Vue.directive('focus', {
    // 当绑定元素插入到 DOM 中。 其中 el为dom元素
    inserted: function (el) {
            // 聚焦元素
            el.focus();
    }
});
new Vue({
   el:'#app'
});
</script>
```

Vue.directive 注册全局指令 带参数

```
  <input type="text" v-color='msg'>
```

```
<script type="text/javascript">
    /*
      自定义指令-带参数
      bind - 只调用一次，在指令第一次绑定到元素上时候调用

    */
    Vue.directive('color', {
      // bind声明周期，只调用一次，指令第一次绑定到元素时调用。在这里可以进行一次性的初始化设置
      // el 为当前自定义指令的DOM元素
      // binding 为自定义的函数形参    通过自定义属性传递过来的值 存在 binding.value 里面
      bind: function(el, binding){
        // 根据指令的参数设置背景色
        // console.log(binding.value.color)
        el.style.backgroundColor = binding.value.color;
      }
    });
    var vm = new Vue({
      el: '#app',
      data: {
        msg: {
          color: 'blue'
        }
      }
    });
  </script>
```

自定义指令局部指令

- 局部指令，需要定义在 directives 的选项 用法和全局用法一样
- 局部指令只能在当前组件里面使用
- 当全局指令和局部指令同名时以局部指令为准

```
<input type="text" v-color='msg'>
 <input type="text" v-focus>
 <script type="text/javascript">
    /*
      自定义指令-局部指令
    */
    var vm = new Vue({
      el: '#app',
      data: {
        msg: {
          color: 'red'
        }
      },
      //局部指令，需要定义在  directives 的选项
      directives: {
        color: {
          bind: function(el, binding){
            el.style.backgroundColor = binding.value.color;
          }
        },
        focus: {
```

```
        inserted: function(el) {
          el.focus();
        }
      }
    }
  });
</script>
```

计算属性 computed

- 模板中放入太多的逻辑会让模板过重且难以维护 使用计算属性可以让模板更加的简洁
- **计算属性是基于它们的响应式依赖进行缓存的**
- computed比较适合对多个变量或者对象进行处理后返回一个结果值，也就是数多个变量中的某一个值发生了变化则我们监控的这个值也就会发生变化

```
<div id="app">
    <!--
       当多次调用 reverseString  的时候
       只要里面的 num 值不改变 他会把第一次计算的结果直接返回
       直到data 中的num值改变 计算属性才会重新发生计算
    -->
   <div>{{reverseString}}</div>
   <div>{{reverseString}}</div>
    <!-- 调用methods中的方法的时候  他每次会重新调用 -->
   <div>{{reverseMessage()}}</div>
   <div>{{reverseMessage()}}</div>
 </div>
 <script type="text/javascript">
   /*
     计算属性与方法的区别:计算属性是基于依赖进行缓存的，而方法不缓存
   */
   var vm = new Vue({
     el: '#app',
     data: {
       msg: 'Nihao',
       num: 100
     },
     methods: {
       reverseMessage: function(){
         console.log('methods')
         return this.msg.split('').reverse().join('');
       }
     },
     //computed  属性 定义 和 data 已经 methods 平级
     computed: {
       //  reverseString   这个是我们自己定义的名字
       reverseString: function(){
         console.log('computed')
         var total = 0;
         //  当data 中的 num 的值改变的时候  reverseString  会自动发生计算
         for(var i=0;i<=this.num;i++){
           total += i;
```

```
            }
            // 这里一定要有return 否则 调用 reverseString 的 时候无法拿到结果
            return total;
          }
        }
      });
  </script>
```

侦听器 watch

- 使用watch来响应数据的变化
- 一般用于异步或者开销较大的操作
- watch 中的属性 一定是data 中 已经存在的数据
- **当需要监听一个对象的改变时，普通的watch方法无法监听到对象内部属性的改变，只有data中的数据才能够监听到变化，此时就需要deep属性对对象进行深度监听**

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

 <script type="text/javascript">
       /*
             侦听器
           */
       var vm = new Vue({
           el: '#app',
           data: {
               firstName: 'Jim',
               lastName: 'Green',
               // fullName: 'Jim Green'
           },
            //watch  属性 定义 和 data 已经 methods 平级
           watch: {
               //   注意:   这里firstName  对应着data 中的 firstName
               //   当 firstName 值 改变的时候  会自动触发 watch
               firstName: function(val) {
                   this.fullName = val + ' ' + this.lastName;
               },
               //   注意:   这里 lastName 对应着data 中的 lastName
               lastName: function(val) {
```

```
                    this.fullName = this.firstName + ' ' + val;
                }
            }
        });
    </script>
```

过滤器

- Vue.js允许自定义过滤器，可被用于一些常见的文本格式化。
- 过滤器可以用在两个地方：双花括号插值和v-bind表达式。
- 过滤器应该被添加在JavaScript表达式的尾部，由“管道”符号指示
- 支持级联操作
- 过滤器不改变真正的`data`，而只是改变渲染的结果，并返回过滤后的版本
- 全局注册时是filter，没有s的。而局部过滤器是filters，是有s的

```
  <div id="app">
    <input type="text" v-model='msg'>
      <!-- upper 被定义为接收单个参数的过滤器函数，表达式  msg  的值将作为参数传入到函数中 -->
    <div>{{msg | upper}}</div>
    <!--
      支持级联操作
      upper  被定义为接收单个参数的过滤器函数，表达式msg 的值将作为参数传入到函数中。
      然后继续调用同样被定义为接收单个参数的过滤器 lower ，将upper 的结果传递到lower中
    -->
    <div>{{msg | upper | lower}}</div>
    <div :abc='msg | upper'>测试数据</div>
  </div>

<script type="text/javascript">
   //  lower  为全局过滤器
   Vue.filter('lower', function(val) {
      return val.charAt(0).toLowerCase() + val.slice(1);
    });
    var vm = new Vue({
      el: '#app',
      data: {
        msg: ''
      },
       //filters  属性 定义 和 data 已经 methods 平级
       //  定义filters 中的过滤器为局部过滤器
      filters: {
        //   upper  自定义的过滤器名字
        //    upper 被定义为接收单个参数的过滤器函数，表达式  msg  的值将作为参数传入到函数中
        upper: function(val) {
         //  过滤器中一定要有返回值 这样外界使用过滤器的时候才能拿到结果
          return val.charAt(0).toUpperCase() + val.slice(1);
        }
      }
    });
  </script>
```

过滤器中传递参数

```
<div id="box">
    <!--
        filterA 被定义为接收三个参数的过滤器函数。
        其中 message 的值作为第一个参数，
        普通字符串 'arg1' 作为第二个参数，表达式 arg2 的值作为第三个参数。
    -->
    {{ message | filterA('arg1', 'arg2') }}
</div>
<script>
    // 在过滤器中 第一个参数 对应的是  管道符前面的数据   n  此时对应 message
    // 第2个参数  a 对应 实参  arg1 字符串
    // 第3个参数  b 对应 实参  arg2 字符串
    Vue.filter('filterA',function(n,a,b){
        if(n<10){
            return n+a;
        }else{
            return n+b;
        }
    });

    new Vue({
        el:"#box",
        data:{
            message: "哈哈哈"
        }
    })

</script>
```

生命周期

- 事物从出生到死亡的过程
- Vue实例从创建 到销毁的过程， 这些过程中会伴随着一些函数的自调用。我们称这些函数为钩子函数

常用的 钩子函数

beforeCreate	**在实例初始化之后，数据观测和事件配置之前被调用 此时data 和 methods 以及页面的DOM结构都没有初始化 什么都做不了**
created	在实例创建完成后被立即调用此时data 和 methods已经可以使用 但是页面还没有渲染出来
beforeMount	在挂载开始之前被调用 此时页面上还看不到真实数据 只是一个模板页面而已
mounted	el被新创建的vm.$el替换，并挂载到实例上去之后调用该钩子。 数据已经真实渲染到页面上 在这个钩子函数里面我们可以使用一些第三方的插件
beforeUpdate	数据更新时调用，发生在虚拟DOM打补丁之前。 页面上数据还是旧的
updated	由于数据更改导致的虚拟DOM重新渲染和打补丁，在这之后会调用该钩子。 页面上数据已经替换成最新的
beforeDestroy	实例销毁之前调用
destroyed	实例销毁后调用

数组变异方法

- 在 Vue 中，直接修改对象属性的值无法触发响应式。当你直接修改了对象属性的值，你会发现，只有数据改了，但是页面内容并没有改变
- 变异数组方法即保持数组方法原有功能不变的前提下对其进行功能拓展

`push()`	**往数组最后面添加一个元素，成功返回当前数组的长度**
`pop()`	删除数组的最后一个元素，成功返回删除元素的值
`shift()`	删除数组的第一个元素，成功返回删除元素的值
`unshift()`	往数组最前面添加一个元素，成功返回当前数组的长度
`splice()`	有三个参数，第一个是想要删除的元素的下标（必选），第二个是想要删除的个数（必选），第三个是删除 后想要在原位置替换的值
`sort()`	sort() 使数组按照字符编码默认从小到大排序,成功返回排序后的数组
`reverse()`	reverse() 将数组倒序，成功返回倒序后的数组

替换数组

- 不会改变原始数组，但总是返回一个新数组

filter	**filter() 方法创建一个新的数组，新数组中的元素是通过检查指定数组中符合条件的所有元素。**
concat	concat() 方法用于连接两个或多个数组。该方法不会改变现有的数组
slice	slice() 方法可从已有的数组中返回选定的元素。该方法并不会修改数组，而是返回一个子数组

动态数组响应式数据

- Vue.set(a,b,c) 让 触发视图重新更新一遍，数据动态起来
- a是要更改的数据 、 b是数据的第几项、 c是更改后的数据

图书列表案例

- 静态列表效果
- 基于数据实现模板效果
- 处理每行的操作按钮

1、 提供的静态数据

- 数据存放在vue 中 data 属性中

```
var vm = new Vue({
     el: '#app',
     data: {
       books: [{
         id: 1,
         name: '三国演义',
         date: ''
       },{
         id: 2,
         name: '水浒传',
         date: ''
       },{
         id: 3,
         name: '红楼梦',
         date: ''
       },{
         id: 4,
         name: '西游记',
         date: ''
       }]
     }
   }); var vm = new Vue({
     el: '#app',
     data: {
       books: [{
         id: 1,
         name: '三国演义',
         date: ''
       },{
         id: 2,
         name: '水浒传',
         date: ''
       },{
         id: 3,
         name: '红楼梦',
         date: ''
       },{
         id: 4,
         name: '西游记',
         date: ''
```

```
        }]
      }
    });
```

2、 把提供好的数据渲染到页面上

- 利用 v-for循环 遍历 books 将每一项数据渲染到对应的数据中

```
 <tbody>
    <tr :key='item.id' v-for='item in books'>
       <!-- 对应的id 渲染到页面上 -->
       <td>{{item.id}}</td>
        <!-- 对应的name 渲染到页面上 -->
       <td>{{item.name}}</td>
       <td>{{item.date}}</td>
       <td>
         <!-- 阻止 a 标签的默认跳转 -->
         <a href="" @click.prevent>修改</a>
         <span>|</span>
          <a href="" @click.prevent>删除</a>
       </td>
     </tr>
</tbody>
```

3、 添加图书

- 通过双向绑定获取到输入框中的输入内容
- 给按钮添加点击事件
- 把输入框中的数据存储到 data 中的 books 里面

```
<div>
  <h1>图书管理</h1>
  <div class="book">
       <div>
         <label for="id">
           编号:
         </label>
          <!-- 3.1、通过双向绑定获取到输入框中的输入的 id  -->
         <input type="text" id="id" v-model='id'>
         <label for="name">
           名称:
         </label>
           <!-- 3.2、通过双向绑定获取到输入框中的输入的 name  -->
         <input type="text" id="name" v-model='name'>
            <!-- 3.3、给按钮添加点击事件  -->
         <button @click='handle'>提交</button>
       </div>
  </div>
</div>
  <script type="text/javascript">
    /*
      图书管理-添加图书
```

```
  */
  var vm = new Vue({
    el: '#app',
    data: {
      id: '',
      name: '',
      books: [{
        id: 1,
        name: '三国演义',
        date: ''
      },{
        id: 2,
        name: '水浒传',
        date: ''
      },{
        id: 3,
        name: '红楼梦',
        date: ''
      },{
        id: 4,
        name: '西游记',
        date: ''
      }]
    },
    methods: {
      handle: function(){
        // 3.4 定义一个新的对象book 存储 获取到输入框中 书 的id和名字
        var book = {};
        book.id = this.id;
        book.name = this.name;
        book.date = '';
       // 3.5 把book  通过数组的变异方法 push 放到    books 里面
        this.books.push(book);
        //3.6 清空输入框
        this.id = '';
        this.name = '';
      }
    }
  });
</script>
```

4 修改图书-上

- 点击修改按钮的时候 获取到要修改的书籍名单
  - 4.1 给修改按钮添加点击事件， 需要把当前的图书的id 传递过去 这样才知道需要修改的是哪一本书籍
- 把需要修改的书籍名单填充到表单里面
  - 4.2 根据传递过来的id 查出books 中 对应书籍的详细信息
  - 4.3 把获取到的信息填充到表单

```
<div id="app">
  <div class="grid">
    <div>
```

```
        <h1>图书管理</h1>
        <div class="book">
          <div>
            <label for="id">
              编号:
            </label>
            <input type="text" id="id" v-model='id' :disabled="flag">
            <label for="name">
              名称:
            </label>
            <input type="text" id="name" v-model='name'>
            <button @click='handle'>提交</button>
          </div>
        </div>
      </div>
      <table>
        <thead>
          <tr>
            <th>编号</th>
            <th>名称</th>
            <th>时间</th>
            <th>操作</th>
          </tr>
        </thead>
        <tbody>
          <tr :key='item.id' v-for='item in books'>
            <td>{{item.id}}</td>
            <td>{{item.name}}</td>
            <td>{{item.date}}</td>
            <td>
              <!---
                4.1  给修改按钮添加点击事件,   需要把当前的图书的id 传递过去
                这样才知道需要修改的是哪一本书籍
                --->
              <a href="" @click.prevent='toEdit(item.id)'>修改</a>
              <span>|</span>
              <a href="" @click.prevent>删除</a>
            </td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
<script type="text/javascript">
    /*
      图书管理-添加图书
    */
    var vm = new Vue({
      el: '#app',
      data: {
        flag: false,
        id: '',
        name: '',
```

```
      books: [{
        id: 1,
        name: '三国演义',
        date: ''
      },{
        id: 2,
        name: '水浒传',
        date: ''
      },{
        id: 3,
        name: '红楼梦',
        date: ''
      },{
        id: 4,
        name: '西游记',
        date: ''
      }]
    },
    methods: {
      handle: function(){
        // 3.4 定义一个新的对象book 存储 获取到输入框中 书 的id和名字
        var book = {};
        book.id = this.id;
        book.name = this.name;
        book.date = '';
       // 3.5 把book  通过数组的变异方法 push 放到    books 里面
        this.books.push(book);
        //3.6 清空输入框
        this.id = '';
        this.name = '';
      },
      toEdit: function(id){
        console.log(id)
        //4.2  根据传递过来的id 查出books 中 对应书籍的详细信息
        var book = this.books.filter(function(item){
          return item.id == id;
        });
        console.log(book)
        //4.3 把获取到的信息填充到表单
        // this.id   和  this.name 通过双向绑定 绑定到了表单中  一旦数据改变视图自动更新
        this.id = book[0].id;
        this.name = book[0].name;
      }
    }
  });
</script>
```

5 修改图书-下

- 5.1 定义一个标识符， 主要是控制 编辑状态下当前编辑书籍的id 不能被修改 即 处于编辑状态下 当前控制书籍编号的输入框禁用
- 5.2 通过属性绑定给书籍编号的 绑定 disabled 的属性 flag 为 true 即为禁用
- 5.3 flag 默认值为false 处于编辑状态 要把 flag 改为true 即当前表单为禁用

- 5.4 复用添加方法 用户点击提交的时候依然执行 handle 中的逻辑如果 flag为true 即 表单处于不可输入状态 此时执行的用户编辑数据数据

```
<div id="app">
    <div class="grid">
      <div>
        <h1>图书管理</h1>
        <div class="book">
          <div>
            <label for="id">
              编号:
            </label>
              <!-- 5.2 通过属性绑定 绑定 disabled 的属性  flag 为 true 即为禁用      -->
            <input type="text" id="id" v-model='id' :disabled="flag">
            <label for="name">
              名称:
            </label>
            <input type="text" id="name" v-model='name'>
            <button @click='handle'>提交</button>
          </div>
        </div>
      </div>
      <table>
        <thead>
          <tr>
            <th>编号</th>
            <th>名称</th>
            <th>时间</th>
            <th>操作</th>
          </tr>
        </thead>
        <tbody>
          <tr :key='item.id' v-for='item in books'>
            <td>{{item.id}}</td>
            <td>{{item.name}}</td>
            <td>{{item.date}}</td>
            <td>
              <a href="" @click.prevent='toEdit(item.id)'>修改</a>
              <span>|</span>
              <a href="" @click.prevent>删除</a>
            </td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
<script type="text/javascript">
        /*图书管理-添加图书*/
        var vm = new Vue({
            el: '#app',
            data: {
                // 5.1  定义一个标识符,  主要是控制 编辑状态下当前编辑书籍的id 不能被修改
                // 即 处于编辑状态下 当前控制书籍编号的输入框禁用
```

```
            flag: false,
            id: '',
            name: '',

        },
        methods: {
            handle: function() {
               /*
                 5.4  复用添加方法    用户点击提交的时候依然执行 handle 中的逻辑
                      如果 flag为true 即 表单处于不可输入状态 此时执行的用户编辑数据数据
               */
                if (this.flag) {
                    // 编辑图书
                    // 5.5  根据当前的ID去更新数组中对应的数据
                    this.books.some((item) => {
                        if (item.id == this.id) {
                            // 箭头函数中 this 指向父级作用域的this
                            item.name = this.name;
                            // 完成更新操作之后，需要终止循环
                            return true;
                        }
                    });
                    // 5.6 编辑完数据后表单要处以可以输入的状态
                    this.flag = false;
                //  5.7  如果 flag为false  表单处于输入状态 此时执行的用户添加数据
                } else {
                    var book = {};
                    book.id = this.id;
                    book.name = this.name;
                    book.date = '';
                    this.books.push(book);
                    // 清空表单
                    this.id = '';
                    this.name = '';
                }
                // 清空表单
                this.id = '';
                this.name = '';
            },
            toEdit: function(id) {
                 /*
                 5.3  flag 默认值为false   处于编辑状态 要把 flag 改为true 即当前表单为禁
用
                 */
                this.flag = true;
                console.log(id)
                var book = this.books.filter(function(item) {
                    return item.id == id;
                });
                console.log(book)
                this.id = book[0].id;
                this.name = book[0].name;
            }
```

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

6 删除图书

- 6.1 给删除按钮添加事件 把当前需要删除的书籍id 传递过来
- 6.2 根据id从数组中查找元素的索引
- 6.3 根据索引删除数组元素

```
  <tbody>
          <tr :key='item.id' v-for='item in books'>
            <td>{{item.id}}</td>
            <td>{{item.name}}</td>
            <td>{{item.date}}</td>
            <td>
              <a href="" @click.prevent='toEdit(item.id)'>修改</a>
              <span>|</span>
               <!--  6.1 给删除按钮添加事件 把当前需要删除的书籍id 传递过来  -->
              <a href="" @click.prevent='deleteBook(item.id)'>删除</a>
            </td>
          </tr>
</tbody>
  <script type="text/javascript">
    /*
      图书管理-添加图书
    */
    var vm = new Vue({
      methods: {
        deleteBook: function(id){
          // 删除图书
          #// 6.2 根据id从数组中查找元素的索引
          // var index = this.books.findIndex(function(item){
          //   return item.id == id;
          // });
          #// 6.3 根据索引删除数组元素
          // this.books.splice(index, 1);
          // -------------------------
         #// 方法二：通过filter方法进行删除

          # 6.4  根据filter 方法 过滤出来id 不是要删除书籍的id
          # 因为 filter 是替换数组不会修改原始数据 所以需要 把 不是要删除书籍的id  赋值给 books
          this.books = this.books.filter(function(item){
            return item.id != id;
          });
        }
      }
    });
  </script>
```

常用特性应用场景

1 过滤器

- Vue.filter 定义一个全局过滤器

```
 <tr :key='item.id' v-for='item in books'>
            <td>{{item.id}}</td>
            <td>{{item.name}}</td>
            <!-- 1.3  调用过滤器 -->
            <td>{{item.date | format('yyyy-MM-dd hh:mm:ss')}}</td>
            <td>
              <a href="" @click.prevent='toEdit(item.id)'>修改</a>
              <span>|</span>
              <a href="" @click.prevent='deleteBook(item.id)'>删除</a>
            </td>
</tr>

<script>
        #1.1  Vue.filter  定义一个全局过滤器
        Vue.filter('format', function(value, arg) {
              function dateFormat(date, format) {
                if (typeof date === "string") {
                  var mts = date.match(/(\/Date\((\d+)\)\/)/);
                  if (mts && mts.length >= 3) {
                    date = parseInt(mts[2]);
                  }
                }
                date = new Date(date);
                if (!date || date.toUTCString() == "Invalid Date") {
                  return "";
                }
                var map = {
                  "M": date.getMonth() + 1, //月份
                  "d": date.getDate(), //日
                  "h": date.getHours(), //小时
                  "m": date.getMinutes(), //分
                  "s": date.getSeconds(), //秒
                  "q": Math.floor((date.getMonth() + 3) / 3), //季度
                  "S": date.getMilliseconds() //毫秒
                };
                format = format.replace(/([yMdhmsqS])+/g, function(all, t) {
                  var v = map[t];
                  if (v !== undefined) {
                    if (all.length > 1) {
                      v = '0' + v;
                      v = v.substr(v.length - 2);
                    }
                    return v;
                  } else if (t === 'y') {
                    return (date.getFullYear() + '').substr(4 - all.length);
                  }
                  return all;
```

```
                });
                return format;
              }
              return dateFormat(value, arg);
            })
#1.2  提供的数据 包含一个时间戳    为毫秒数
   [{
          id: 1,
          name: '三国演义',
          date: 2525609975000
        },{
          id: 2,
          name: '水浒传',
          date: 2525609975000
        },{
          id: 3,
          name: '红楼梦',
          date: 2525609975000
        },{
          id: 4,
          name: '西游记',
          date: 2525609975000
        }];
</script>
```

2 自定义指令

- 让表单自动获取焦点
- 通过Vue.directive 自定义指定

```
<!-- 2.2  通过v-自定义属性名 调用自定义指令 -->
<input type="text" id="id" v-model='id' :disabled="flag" v-focus>

<script>
    # 2.1   通过Vue.directive 自定义指定
    Vue.directive('focus', {
      inserted: function (el) {
        el.focus();
      }
    });

</script>
```

3 计算属性

- 通过计算属性计算图书的总数
    - 图书的总数就是计算数组的长度

```
 <div class="total">
        <span>图书总数: </span>
        <!-- 3.2 在页面上 展示出来 -->
        <span>{{total}}</span>
```

```
</div>

  <script type="text/javascript">
    /*
      计算属性与方法的区别:计算属性是基于依赖进行缓存的，而方法不缓存
    */
    var vm = new Vue({
      data: {
        flag: false,
        submitFlag: false,
        id: '',
        name: '',
        books: []
      },
      computed: {
        total: function(){
          // 3.1  计算图书的总数
          return this.books.length;
        }
      },
    });
  </script>
```

生命周期

day03

组件

- 组件 (Component) 是 Vue.js 最强大的功能之一
- 组件可以扩展 HTML 元素，封装可重用的代

组件注册

全局注册

- Vue.component('组件名称', { }) 第1个参数是标签名称，第2个参数是一个选项对象
- **全局组件**注册后，任何**vue实例**都可以用

组件基础用

```
<div id="example">
  <!-- 2、 组件使用 组件名称 是以HTML标签的形式使用  -->
  <my-component></my-component>
</div>
<script>
    //   注册组件
    // 1、 my-component 就是组件中自定义的标签名
    Vue.component('my-component', {
      template: '<div>A custom component!</div>'
    })

    // 创建根实例
    new Vue({
      el: '#example'
    })

</script>
```

组件注意事项

- 组件参数的data值必须是函数同时这个函数要求返回一个对象
- 组件模板必须是单个根元素
- 组件模板的内容可以是模板字符串

```
  <div id="app">
     <!--
        4、   组件可以重复使用多次
          因为data中返回的是一个对象所以每个组件中的数据是私有的
          即每个实例可以维护一份被返回对象的独立的拷贝
    -->
    <button-counter></button-counter>
    <button-counter></button-counter>
    <button-counter></button-counter>
```

```
      <!-- 8、必须使用短横线的方式使用组件 -->
     <hello-world></hello-world>
  </div>

<script type="text/javascript">
    //5  如果使用驼峰式命名组件，那么在使用组件的时候，只能在字符串模板中用驼峰的方式使用组件，
    // 7、但是在普通的标签模板中，必须使用短横线的方式使用组件
     Vue.component('HelloWorld', {
      data: function(){
        return {
          msg: 'HelloWorld'
        }
      },
      template: '<div>{{msg}}</div>'
    });



    Vue.component('button-counter', {
      // 1、组件参数的data值必须是函数
      // 同时这个函数要求返回一个对象
      data: function(){
        return {
          count: 0
        }
      },
      //  2、组件模板必须是单个根元素
      //  3、组件模板的内容可以是模板字符串
      template: `
        <div>
          <button @click="handle">点击了{{count}}次</button>
          <button>测试123</button>
            #  6 在字符串模板中可以使用驼峰的方式使用组件
           <HelloWorld></HelloWorld>
        </div>
      `,
      methods: {
        handle: function(){
          this.count += 2;
        }
      }
    })
    var vm = new Vue({
      el: '#app',
      data: {

      }
    });
  </script>
```

局部注册

- 只能在当前注册它的vue实例中使用

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


<script>
    // 定义组件的模板
    var Child = {
      template: '<div>A custom component!</div>'
    }
    new Vue({
      //局部注册组件
      components: {
        // <my-component> 将只在父模板可用  一定要在实例上注册了才能在html文件中使用
        'my-component': Child
      }
    })
 </script>
```

Vue 调试工具

Vue组件之间传值

父组件向子组件传值

- 父组件发送的形式是以属性的形式绑定值到子组件身上。
- 然后子组件用属性props接收
- 在props中使用驼峰形式，模板中需要使用短横线的形式字符串形式的模板中没有这个限制

```
  <div id="app">
    <div>{{pmsg}}</div>
     <!--1、menu-item  在 APP中嵌套着 故 menu-item   为  子组件       -->
     <!-- 给子组件传入一个静态的值 -->
    <menu-item title='来自父组件的值'></menu-item>
    <!-- 2、 需要动态的数据的时候 需要属性绑定的形式设置 此时 ptitle  来自父组件data 中的数据 .
          传的值可以是数字、对象、数组等等
    -->
    <menu-item :title='ptitle' content='hello'></menu-item>
  </div>

  <script type="text/javascript">
    Vue.component('menu-item', {
      // 3、 子组件用属性props接收父组件传递过来的数据
      props: ['title', 'content'],
```

```
    data: function() {
      return {
        msg: '子组件本身的数据'
      }
    },
    template: '<div>{{msg + "----" + title + "-----" + content}}</div>'
  });
  var vm = new Vue({
    el: '#app',
    data: {
      pmsg: '父组件中内容',
      ptitle: '动态绑定属性'
    }
  });
</script>
```

子组件向父组件传值

- 子组件用 `$emit()` 触发事件
- `$emit()` 第一个参数为 自定义的事件名称 第二个参数为需要传递的数据
- 父组件用v-on 监听子组件的事件

```
<div id="app">
  <div :style='{fontSize: fontSize + "px"}'>{{pmsg}}</div>
   <!-- 2 父组件用v-on 监听子组件的事件
      这里 enlarge-text  是从 $emit 中的第一个参数对应   handle 为对应的事件处理函数
  -->
  <menu-item :parr='parr' @enlarge-text='handle($event)'></menu-item>
</div>
<script type="text/javascript" src="js/vue.js"></script>
<script type="text/javascript">
  /*
    子组件向父组件传值-携带参数
  */

  Vue.component('menu-item', {
    props: ['parr'],
    template: `
      <div>
        <ul>
          <li :key='index' v-for='(item,index) in parr'>{{item}}</li>
        </ul>
          ###  1、子组件用$emit()触发事件
          ### 第一个参数为 自定义的事件名称    第二个参数为需要传递的数据
        <button @click='$emit("enlarge-text", 5)'>扩大父组件中字体大小</button>
        <button @click='$emit("enlarge-text", 10)'>扩大父组件中字体大小</button>
      </div>
    `
  });
  var vm = new Vue({
    el: '#app',
    data: {
```

```
        pmsg: '父组件中内容',
        parr: ['apple','orange','banana'],
        fontSize: 10
      },
      methods: {
        handle: function(val){
          // 扩大字体大小
          this.fontSize += val;
        }
      }
    });
  </script>
```

兄弟之间的传递

- 兄弟之间传递数据需要借助于事件中心，通过事件中心传递数据
  - 提供事件中心 var hub = new Vue()
- 传递数据方，通过一个事件触发hub.$emit(方法名，传递的数据)
- 接收数据方，通过mounted(){} 钩子中 触发hub.$on()方法名
- 销毁事件 通过hub.$off()方法名销毁之后无法进行传递数据

```
<div id="app">
    <div>父组件</div>
    <div>
      <button @click='handle'>销毁事件</button>
    </div>
    <test-tom></test-tom>
    <test-jerry></test-jerry>
  </div>
  <script type="text/javascript" src="js/vue.js"></script>
  <script type="text/javascript">
    /*
      兄弟组件之间数据传递
    */
    //1、 提供事件中心
    var hub = new Vue();

    Vue.component('test-tom', {
      data: function(){
        return {
          num: 0
        }
      },
      template: `
        <div>
          <div>TOM:{{num}}</div>
          <div>
            <button @click='handle'>点击</button>
          </div>
        </div>
```

```
    `,
    methods: {
      handle: function(){
        //2、传递数据方，通过一个事件触发hub.$emit(方法名，传递的数据)    触发兄弟组件的事件
        hub.$emit('jerry-event', 2);
      }
    },
    mounted: function() {
     // 3、接收数据方，通过mounted(){} 钩子中   触发hub.$on(方法名
      hub.$on('tom-event', (val) => {
        this.num += val;
      });
    }
  });
  Vue.component('test-jerry', {
    data: function(){
      return {
        num: 0
      }
    },
    template: `
      <div>
        <div>JERRY:{{num}}</div>
        <div>
          <button @click='handle'>点击</button>
        </div>
      </div>
    `,
    methods: {
      handle: function(){
        //2、传递数据方，通过一个事件触发hub.$emit(方法名，传递的数据)    触发兄弟组件的事件
        hub.$emit('tom-event', 1);
      }
    },
    mounted: function() {
      // 3、接收数据方，通过mounted(){} 钩子中   触发hub.$on()方法名
      hub.$on('jerry-event', (val) => {
        this.num += val;
      });
    }
  });
  var vm = new Vue({
    el: '#app',
    data: {

    },
    methods: {
      handle: function(){
        //4、销毁事件 通过hub.$off()方法名销毁之后无法进行传递数据
        hub.$off('tom-event');
        hub.$off('jerry-event');
      }
    }
```

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

组件插槽

- 组件的最大特性就是复用性，而用好插槽能大大提高组件的可复用能力

匿名插槽

```
<div id="app">
  <!-- 这里的所有组件标签中嵌套的内容会替换掉slot  如果不传值 则使用 slot 中的默认值  -->
  <alert-box>有bug发生</alert-box>
  <alert-box>有一个警告</alert-box>
  <alert-box></alert-box>
</div>

<script type="text/javascript">
  /*
    组件插槽：父组件向子组件传递内容
  */
  Vue.component('alert-box', {
    template: `
      <div>
        <strong>ERROR:</strong>
      # 当组件渲染的时候，这个 <slot> 元素将会被替换为“组件标签中嵌套的内容”。
      # 插槽内可以包含任何模板代码，包括 HTML
        <slot>默认内容</slot>
      </div>
    `
  });
  var vm = new Vue({
    el: '#app',
    data: {

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

具名插槽

- 具有名字的插槽
- 使用 中的 "name" 属性绑定元素

```
<div id="app">
  <base-layout>
     <!-- 2、 通过slot属性来指定，这个slot的值必须和下面slot组件得name值对应上
```

```
                如果没有匹配到 则放到匿名的插槽中    -->
      <p slot='header'>标题信息</p>
      <p>主要内容1</p>
      <p>主要内容2</p>
      <p slot='footer'>底部信息信息</p>
    </base-layout>

    <base-layout>
      <!-- 注意点：template临时的包裹标签最终不会渲染到页面上      -->
      <template slot='header'>
        <p>标题信息1</p>
        <p>标题信息2</p>
      </template>
      <p>主要内容1</p>
      <p>主要内容2</p>
      <template slot='footer'>
        <p>底部信息信息1</p>
        <p>底部信息信息2</p>
      </template>
    </base-layout>
  </div>
  <script type="text/javascript" src="js/vue.js"></script>
  <script type="text/javascript">
    /*
      具名插槽
    */
    Vue.component('base-layout', {
      template: `
        <div>
          <header>
            ### 1、 使用 <slot> 中的 "name" 属性绑定元素 指定当前插槽的名字
            <slot name='header'></slot>
          </header>
          <main>
            <slot></slot>
          </main>
          <footer>
            ###  注意点:
            ###  具名插槽的渲染顺序，完全取决于模板，而不是取决于父组件中元素的顺序
            <slot name='footer'></slot>
          </footer>
        </div>
      `
    });
    var vm = new Vue({
      el: '#app',
      data: {

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

作用域插槽

- 父组件对子组件加工处理
- 既可以复用子组件的slot，又可以使slot内容不一致

```
<div id="app">
  <!--
      1、当我们希望li 的样式由外部使用组件的地方定义，因为可能有多种地方要使用该组件，
      但样式希望不一样 这个时候我们需要使用作用域插槽

  -->
  <fruit-list :list='list'>
     <!-- 2、 父组件中使用了<template>元素,而且包含scope="slotProps",
          slotProps在这里只是临时变量
      --->
    <template slot-scope='slotProps'>
      <strong v-if='slotProps.info.id==3' class="current">
          {{slotProps.info.name}}
       </strong>
      <span v-else>{{slotProps.info.name}}</span>
    </template>
  </fruit-list>
</div>
<script type="text/javascript" src="js/vue.js"></script>
<script type="text/javascript">
  /*
    作用域插槽
  */
  Vue.component('fruit-list', {
    props: ['list'],
    template: `
      <div>
        <li :key='item.id' v-for='item in list'>
          ###  3、 在子组件模板中,<slot>元素上有一个类似props传递数据给组件的写法msg="xxx",
          ###    插槽可以提供一个默认内容，如果如果父组件没有为这个插槽提供了内容，会显示默认的内容。
                    如果父组件为这个插槽提供了内容，则默认的内容会被替换掉
          <slot :info='item'>{{item.name}}</slot>
        </li>
      </div>
    `
  });
  var vm = new Vue({
    el: '#app',
    data: {
      list: [{
        id: 1,
        name: 'apple'
      },{
        id: 2,
        name: 'orange'
      },{
```

```
          id: 3,
          name: 'banana'
        }]
      }
    });
  </script>
</body>
</html>
```

购物车案例

1. 实现组件化布局

- 把静态页面转换成组件化模式
- 把组件渲染到页面上

```
<div id="app">
   <div class="container">
     <!-- 2、把组件渲染到页面上 -->
     <my-cart></my-cart>
   </div>
 </div>
 <script type="text/javascript" src="js/vue.js"></script>
 <script type="text/javascript">
   # 1、 把静态页面转换成组件化模式
   # 1.1  标题组件
   var CartTitle = {
     template: `
       <div class="title">我的商品</div>
     `
   }
   # 1.2  商品列表组件
   var CartList = {
     #  注意点 :   组件模板必须是单个根元素
     template: `
       <div>
         <div class="item">
           <img src="img/a.jpg"/>
           <div class="name"></div>
           <div class="change">
             <a href=""> - </a>
             <input type="text" class="num" />
             <a href=""> + </a>
           </div>
           <div class="del">x</div>
         </div>
         <div class="item">
           <img src="img/b.jpg"/>
           <div class="name"></div>
           <div class="change">
```

```
            <a href="">-</a>
            <input type="text" class="num" />
            <a href="">+</a>
          </div>
          <div class="del">x</div>
        </div>
        <div class="item">
          <img src="img/c.jpg"/>
          <div class="name"></div>
          <div class="change">
            <a href="">-</a>
            <input type="text" class="num" />
            <a href="">+</a>
          </div>
          <div class="del">x</div>
        </div>
        <div class="item">
          <img src="img/d.jpg"/>
          <div class="name"></div>
          <div class="change">
            <a href="">-</a>
            <input type="text" class="num" />
            <a href="">+</a>
          </div>
          <div class="del">x</div>
        </div>
        <div class="item">
          <img src="img/e.jpg"/>
          <div class="name"></div>
          <div class="change">
            <a href="">-</a>
            <input type="text" class="num" />
            <a href="">+</a>
          </div>
          <div class="del">x</div>
        </div>
      </div>
    `
}
# 1.3  商品结算组件
var CartTotal = {
  template: `
    <div class="total">
      <span>总价: 123</span>
      <button>结算</button>
    </div>
  `
}
## 1.4  定义一个全局组件 my-cart
Vue.component('my-cart',{
  ##  1.6 引入子组件
  template: `
    <div class='cart'>
```

```
          <cart-title></cart-title>
          <cart-list></cart-list>
          <cart-total></cart-total>
        </div>
      `,
      # 1.5  注册子组件
      components: {
        'cart-title': CartTitle,
        'cart-list': CartList,
        'cart-total': CartTotal
      }
    });
    var vm = new Vue({
      el: '#app',
      data: {

      }
    });

  </script>
```

2、 实现 标题和结算功能组件

- 标题组件实现动态渲染
  - 从父组件把标题数据传递过来 即 父向子组件传值
  - 把传递过来的数据渲染到页面上
- 结算功能组件
  - 从父组件把商品列表list 数据传递过来 即 父向子组件传值
  - 把传递过来的数据计算最终价格渲染到页面上

```
<div id="app">
    <div class="container">
      <my-cart></my-cart>
    </div>
  </div>
  <script type="text/javascript" src="js/vue.js"></script>
  <script type="text/javascript">
     # 2.2  标题组件      子组件通过props形式接收父组件传递过来的uname数据
    var CartTitle = {
      props: ['uname'],
      template: `
        <div class="title">{{uname}}的商品</div>
      `
    }
    # 2.3  商品结算组件  子组件通过props形式接收父组件传递过来的list数据
    var CartTotal = {
      props: ['list'],
      template: `
        <div class="total">
```

```
      <span>总价: {{total}}</span>
      <button>结算</button>
    </div>
  `,
  computed: {
    # 2.4    计算商品的总价   并渲染到页面上
    total: function() {
      var t = 0;
      this.list.forEach(item => {
        t += item.price * item.num;
      });
      return t;
    }
  }
}
Vue.component('my-cart',{
  data: function() {
    return {
      uname: '张三',
      list: [{
        id: 1,
        name: 'TCL彩电',
        price: 1000,
        num: 1,
        img: 'img/a.jpg'
      },{
        id: 2,
        name: '机顶盒',
        price: 1000,
        num: 1,
        img: 'img/b.jpg'
      },{
        id: 3,
        name: '海尔冰箱',
        price: 1000,
        num: 1,
        img: 'img/c.jpg'
      },{
        id: 4,
        name: '小米手机',
        price: 1000,
        num: 1,
        img: 'img/d.jpg'
      },{
        id: 5,
        name: 'PPTV电视',
        price: 1000,
        num: 2,
        img: 'img/e.jpg'
      }]
    }
  },
  #  2.1  父组件向子组件以属性传递的形式 传递数据
```

```
    #   向 标题组件传递 uname 属性    向 商品结算组件传递 list  属性
    template: `
      <div class='cart'>
        <cart-title :uname='uname'></cart-title>
        <cart-list></cart-list>
        <cart-total :list='list'></cart-total>
      </div>
    `,
    components: {
      'cart-title': CartTitle,
      'cart-list': CartList,
      'cart-total': CartTotal
    }
  });
  var vm = new Vue({
    el: '#app',
    data: {

    }
  });

</script>
```

3. 实现列表组件删除功能

- 从父组件把商品列表list 数据传递过来 即 父向子组件传值
- 把传递过来的数据渲染到页面上
- 点击删除按钮的时候删除对应的数据
  - 给按钮添加点击事件把需要删除的id传递过来
    - 子组件中不推荐操作父组件的数据有可能多个子组件使用父组件的数据 我们需要把数据传递给父组件让父组件操作数据
    - 父组件删除对应的数据

```
<div id="app">
  <div class="container">
    <my-cart></my-cart>
  </div>
</div>
<script type="text/javascript" src="js/vue.js"></script>
<script type="text/javascript">

  var CartTitle = {
    props: ['uname'],
    template: `
      <div class="title">{{uname}}的商品</div>
    `
  }
  #  3.2 把列表数据动态渲染到页面上
  var CartList = {
    props: ['list'],
```

```
  template: `
    <div>
      <div :key='item.id' v-for='item in list' class="item">
        <img :src="item.img"/>
        <div class="name">{{item.name}}</div>
        <div class="change">
          <a href="">-</a>
          <input type="text" class="num" />
          <a href="">+</a>
        </div>
        # 3.3  给按钮添加点击事件把需要删除的id传递过来
        <div class="del" @click='del(item.id)'>x</div>
      </div>
    </div>
  `,
  methods: {
    del: function(id){
       # 3.4 子组件中不推荐操作父组件的数据有可能多个子组件使用父组件的数据
      #       我们需要把数据传递给父组件 让父组件操作数据
      this.$emit('cart-del', id);
    }
  }
}
var CartTotal = {
  props: ['list'],
  template: `
    <div class="total">
      <span>总价: {{total}}</span>
      <button>结算</button>
    </div>
  `,
  computed: {
    total: function() {
      // 计算商品的总价
      var t = 0;
      this.list.forEach(item => {
        t += item.price * item.num;
      });
      return t;
    }
  }
}
Vue.component('my-cart',{
  data: function() {
    return {
      uname: '张三',
      list: [{
        id: 1,
        name: 'TCL彩电',
        price: 1000,
        num: 1,
        img: 'img/a.jpg'
      },{
```

```
          id: 2,
          name: '机顶盒',
          price: 1000,
          num: 1,
          img: 'img/b.jpg'
        },{
          id: 3,
          name: '海尔冰箱',
          price: 1000,
          num: 1,
          img: 'img/c.jpg'
        },{
          id: 4,
          name: '小米手机',
          price: 1000,
          num: 1,
          img: 'img/d.jpg'
        },{
          id: 5,
          name: 'PPTV电视',
          price: 1000,
          num: 2,
          img: 'img/e.jpg'
        }]
      }
    },
    # 3.1 从父组件把商品列表list 数据传递过来 即 父向子组件传值
    template: `
      <div class='cart'>
        <cart-title :uname='uname'></cart-title>
        #  3.5  父组件通过事件绑定 接收子组件传递过来的数据
        <cart-list :list='list' @cart-del='delCart($event)'></cart-list>
        <cart-total :list='list'></cart-total>
      </div>
    `,
    components: {
      'cart-title': CartTitle,
      'cart-list': CartList,
      'cart-total': CartTotal
    },
    methods: {
      # 3.6    根据id删除list中对应的数据
      delCart: function(id) {
        // 1、找到id所对应数据的索引
        var index = this.list.findIndex(item=>{
          return item.id == id;
        });
        // 2、根据索引删除对应数据
        this.list.splice(index, 1);
      }
    }
  });
  var vm = new Vue({
```

```
      el: '#app',
      data: {

      }
    });

  </script>
</body>
</html>
```

4. 实现组件更新数据功能 上

- 1. 将输入框中的默认数据动态渲染出来
- 2. 输入框失去焦点的时候 更改商品的数量
- 3 子组件中不推荐操作数据 把这些数据传递给父组件 让父组件处理这些数据
- 4 父组件中接收子组件传递过来的数据并处理

```
 <div id="app">
    <div class="container">
      <my-cart></my-cart>
    </div>
  </div>
  <script type="text/javascript" src="js/vue.js"></script>
  <script type="text/javascript">

    var CartTitle = {
      props: ['uname'],
      template: `
        <div class="title">{{uname}}的商品</div>
      `
    }
    var CartList = {
      props: ['list'],
      template: `
        <div>
          <div :key='item.id' v-for='item in list' class="item">
            <img :src="item.img"/>
            <div class="name">{{item.name}}</div>
            <div class="change">
              <a href="">-</a>
                # 1. 将输入框中的默认数据动态渲染出来
                # 2. 输入框失去焦点的时候 更改商品的数量  需要将当前商品的id 传递过来
              <input type="text" class="num" :value='item.num' @blur='changeNum(item.id,
$event)'/>
              <a href="">+</a>
            </div>
            <div class="del" @click='del(item.id)'>x</div>
          </div>
        </div>
      `,
      methods: {
        changeNum: function(id, event){
```

```
          # 3 子组件中不推荐操作数据  因为别的组件可能也引用了这些数据
          #  把这些数据传递给父组件 让父组件处理这些数据
          this.$emit('change-num', {
            id: id,
            num: event.target.value
          });
        },
        del: function(id){
          // 把id传递给父组件
          this.$emit('cart-del', id);
        }
      }
    }
    var CartTotal = {
      props: ['list'],
      template: `
        <div class="total">
          <span>总价: {{total}}</span>
          <button>结算</button>
        </div>
      `,
      computed: {
        total: function() {
          // 计算商品的总价
          var t = 0;
          this.list.forEach(item => {
            t += item.price * item.num;
          });
          return t;
        }
      }
    }
    Vue.component('my-cart',{
      data: function() {
        return {
          uname: '张三',
          list: [{
            id: 1,
            name: 'TCL彩电',
            price: 1000,
            num: 1,
            img: 'img/a.jpg'
          }]
      },
      template: `
        <div class='cart'>
          <cart-title :uname='uname'></cart-title>
            # 4  父组件中接收子组件传递过来的数据
          <cart-list :list='list' @change-num='changeNum($event)' @cart-
del='delCart($event)'></cart-list>
          <cart-total :list='list'></cart-total>
        </div>
      `,
```

```
    components: {
      'cart-title': CartTitle,
      'cart-list': CartList,
      'cart-total': CartTotal
    },
    methods: {
      changeNum: function(val) {
        //4.1 根据子组件传递过来的数据，跟新list中对应的数据
        this.list.some(item=>{
          if(item.id == val.id) {
            item.num = val.num;
            // 终止遍历
            return true;
          }
        });
      },
      delCart: function(id) {
        // 根据id删除list中对应的数据
        // 1、找到id所对应数据的索引
        var index = this.list.findIndex(item=>{
          return item.id == id;
        });
        // 2、根据索引删除对应数据
        this.list.splice(index, 1);
      }
    }
  });
  var vm = new Vue({
    el: '#app',
    data: {

    }
  });

</script>
```

5. 实现组件更新数据功能 下

- 1

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Document</title>
  <style type="text/css">
    .container {
    }
    .container .cart {
      width: 300px;
      margin: auto;
    }
    .container .title {
```

```
  background-color: lightblue;
  height: 40px;
  line-height: 40px;
  text-align: center;
  /*color: #fff;*/
}
.container .total {
  background-color: #FFCE46;
  height: 50px;
  line-height: 50px;
  text-align: right;
}
.container .total button {
  margin: 0 10px;
  background-color: #DC4C40;
  height: 35px;
  width: 80px;
  border: 0;
}
.container .total span {
  color: red;
  font-weight: bold;
}
.container .item {
  height: 55px;
  line-height: 55px;
  position: relative;
  border-top: 1px solid #ADD8E6;
}
.container .item img {
  width: 45px;
  height: 45px;
  margin: 5px;
}
.container .item .name {
  position: absolute;
  width: 90px;
  top: 0;left: 55px;
  font-size: 16px;
}

.container .item .change {
  width: 100px;
  position: absolute;
  top: 0;
  right: 50px;
}
.container .item .change a {
  font-size: 20px;
  width: 30px;
  text-decoration:none;
  background-color: lightgray;
  vertical-align: middle;
```

```
    }
    .container .item .change .num {
      width: 40px;
      height: 25px;
    }
    .container .item .del {
      position: absolute;
      top: 0;
      right: 0px;
      width: 40px;
      text-align: center;
      font-size: 40px;
      cursor: pointer;
      color: red;
    }
    .container .item .del:hover {
      background-color: orange;
    }
  </style>
</head>
<body>
  <div id="app">
    <div class="container">
      <my-cart></my-cart>
    </div>
  </div>
  <script type="text/javascript" src="js/vue.js"></script>
  <script type="text/javascript">

    var CartTitle = {
      props: ['uname'],
      template: `
        <div class="title">{{uname}}的商品</div>
      `
    }
    var CartList = {
      props: ['list'],
      template: `
        <div>
          <div :key='item.id' v-for='item in list' class="item">
            <img :src="item.img"/>
            <div class="name">{{item.name}}</div>
            <div class="change">
              # 1.  + - 按钮绑定事件
              <a href="" @click.prevent='sub(item.id)'> - </a>
              <input type="text" class="num" :value='item.num' @blur='changeNum(item.id,
$event)'/>
              <a href="" @click.prevent='add(item.id)'> + </a>
            </div>
            <div class="del" @click='del(item.id)'>x</div>
          </div>
        </div>
      `,
```

```
      methods: {
        changeNum: function(id, event){
          this.$emit('change-num', {
            id: id,
            type: 'change',
            num: event.target.value
          });
        },
        sub: function(id){
          # 2 数量的增加和减少通过父组件来计算    每次都是加1 和 减1 不需要传递数量    父组件需要一个类型来判断 是 加一 还是减1  以及是输入框输入的数据  我们通过type 标识符来标记 不同的操作
          this.$emit('change-num', {
            id: id,
            type: 'sub'
          });
        },
        add: function(id){
          # 2 数量的增加和减少通过父组件来计算    每次都是加1 和 减1 不需要传递数量    父组件需要一个类型来判断 是 加一 还是减1  以及是输入框输入的数据  我们通过type 标识符来标记 不同的操作
          this.$emit('change-num', {
            id: id,
            type: 'add'
          });
        },
        del: function(id){
          // 把id传递给父组件
          this.$emit('cart-del', id);
        }
      }
    }
    var CartTotal = {
      props: ['list'],
      template: `
        <div class="total">
          <span>总价: {{total}}</span>
          <button>结算</button>
        </div>
      `,
      computed: {
        total: function() {
          // 计算商品的总价
          var t = 0;
          this.list.forEach(item => {
            t += item.price * item.num;
          });
          return t;
        }
      }
    }
    Vue.component('my-cart',{
      data: function() {
        return {
          uname: '张三',
```

```
          list: [{
            id: 1,
            name: 'TCL彩电',
            price: 1000,
            num: 1,
            img: 'img/a.jpg'
          },{
            id: 2,
            name: '机顶盒',
            price: 1000,
            num: 1,
            img: 'img/b.jpg'
          },{
            id: 3,
            name: '海尔冰箱',
            price: 1000,
            num: 1,
            img: 'img/c.jpg'
          },{
            id: 4,
            name: '小米手机',
            price: 1000,
            num: 1,
            img: 'img/d.jpg'
          },{
            id: 5,
            name: 'PPTV电视',
            price: 1000,
            num: 2,
            img: 'img/e.jpg'
          }]
        }
      },
      template: `
        <div class='cart'>
          <cart-title :uname='uname'></cart-title>
        # 3 父组件通过事件监听    接收子组件的数据
          <cart-list :list='list' @change-num='changeNum($event)' @cart-
del='delCart($event)'></cart-list>
          <cart-total :list='list'></cart-total>
        </div>
      `,
      components: {
        'cart-title': CartTitle,
        'cart-list': CartList,
        'cart-total': CartTotal
      },
      methods: {
        changeNum: function(val) {
          #4 分为三种情况：输入框变更、加号变更、减号变更
          if(val.type=='change') {
            // 根据子组件传递过来的数据，跟新list中对应的数据
            this.list.some(item=>{
```

```
              if(item.id == val.id) {
                item.num = val.num;
                // 终止遍历
                return true;
              }
            });
          }else if(val.type=='sub'){
            // 减一操作
            this.list.some(item=>{
              if(item.id == val.id) {
                item.num -= 1;
                // 终止遍历
                return true;
              }
            });
          }else if(val.type=='add'){
            // 加一操作
            this.list.some(item=>{
              if(item.id == val.id) {
                item.num += 1;
                // 终止遍历
                return true;
              }
            });
          }
        }
      }
    });
    var vm = new Vue({
      el: '#app',
      data: {

      }
    });

  </script>
</body>
</html>
```

接口调用方式

- 原生ajax
- 基于jQuery的ajax
- fetch
- axios

异步

- JavaScript的执行环境是「单线程」
- 所谓单线程，是指JS引擎中负责解释和执行JavaScript代码的线程只有一个，也就是一次只能完成一项任务，这个任务执行完后才能执行下一个，它会「阻塞」其他任务。这个任务可称为主线程
- 异步模式可以一起执行**多个任务**
- JS中常见的异步调用
  - 定时任何
  - ajax
  - 事件函数

promise

- 主要解决异步深层嵌套的问题
- promise 提供了简洁的API 使得异步操作更加容易

```
<script type="text/javascript">
  /*
   1. Promise基本使用
         我们使用new来构建一个Promise  Promise的构造函数接收一个参数，是函数，并且传入两个参数:
resolve, reject,  分别表示异步操作执行成功后的回调函数和异步操作执行失败后的回调函数
  */


  var p = new Promise(function(resolve, reject){
    //2. 这里用于实现异步任务  setTimeout
    setTimeout(function(){
      var flag = false;
      if(flag) {
        //3. 正常情况
        resolve('hello');
      }else{
        //4. 异常情况
        reject('出错了');
      }
    }, 100);
  });
  //  5 Promise实例生成以后，可以用then方法指定resolved状态和reject状态的回调函数
```

```
  //  在then方法中，你也可以直接return数据而不是Promise对象，在后面的then中就可以接收到数据了
  p.then(function(data){
    console.log(data)
  },function(info){
    console.log(info)
  });
</script>
```

基于Promise发送Ajax请求

```
<script type="text/javascript">
  /*
    基于Promise发送Ajax请求
  */
  function queryData(url) {
   #   1.1 创建一个Promise实例
    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) {
          # 1.2 处理正常的情况
          resolve(xhr.responseText);
        }else{
          # 1.3 处理异常情况
          reject('服务器错误');
        }
      };
      xhr.open('get', url);
      xhr.send(null);
    });
    return p;
  }
  # 注意:   这里需要开启一个服务
  # 在then方法中，你也可以直接return数据而不是Promise对象，在后面的then中就可以接收到数据了
  queryData('http://localhost:3000/data')
    .then(function(data){
      console.log(data)
      #  1.4 想要继续链式编程下去 需要 return
      return queryData('http://localhost:3000/data1');
    })
    .then(function(data){
      console.log(data);
      return queryData('http://localhost:3000/data2');
    })
    .then(function(data){
      console.log(data)
    });
</script>
```

Promise 基本API

实例方法

.then()

- 得到异步任务正确的结果

.catch()

- 获取异常信息

.finally()

- 成功与否都会执行（不是正式标准）

```
<script type="text/javascript">
  /*
    Promise常用API-实例方法
  */
  // console.dir(Promise);
  function foo() {
    return new Promise(function(resolve, reject){
      setTimeout(function(){
        // resolve(123);
        reject('error');
      }, 100);
    })
  }
  // foo()
  //   .then(function(data){
  //     console.log(data)
  //   })
  //   .catch(function(data){
  //     console.log(data)
  //   })
  //   .finally(function(){
  //     console.log('finished')
  //   });

  // --------------------------
  // 两种写法是等效的
  foo()
    .then(function(data){
      # 得到异步任务正确的结果
      console.log(data)
    },function(data){
      # 获取异常信息
      console.log(data)
    })
    # 成功与否都会执行（不是正式标准）
    .finally(function(){
      console.log('finished')
    });
</script>
```

静态方法

.all()

- `Promise.all`方法接受一个数组作参数，数组中的对象（p1、p2、p3）均为promise实例（如果不是一个promise，该项会被用`Promise.resolve`转换为一个promise)。它的状态由这三个promise实例决定

.race()

- `Promise.race`方法同样接受一个数组作参数。当p1, p2, p3中有一个实例的状态发生改变（变为`fulfilled`或`rejected`），p的状态就跟着改变。并把第一个改变状态的promise的返回值，传给p的回调函数

```
  <script type="text/javascript">
    /*
      Promise常用API-对象方法
    */
    // console.dir(Promise)
    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) {
            // 处理正常的情况
            resolve(xhr.responseText);
          }else{
            // 处理异常情况
            reject('服务器错误');
          }
        };
        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){
       //   all 中的参数  [p1,p2,p3]   和 返回的结果一 一对应["HELLO TOM", "HELLO JERRY",
"HELLO SPIKE"]
       console.log(result) //["HELLO TOM", "HELLO JERRY", "HELLO SPIKE"]
     })
    Promise.race([p1,p2,p3]).then(function(result){
      // 由于p1执行较快，Promise的then()将获得结果'P1'。p2,p3仍在继续执行，但执行结果将被丢弃。
      console.log(result) // "HELLO TOM"
    })
  </script>
```

fetch

- Fetch API是新的ajax解决方案 Fetch会返回Promise
- **fetch不是ajax的进一步封装，而是原生js，没有使用XMLHttpRequest对象。**

- fetch(url, options).then()

```
<script type="text/javascript">
  /*
    Fetch API 基本用法
      fetch(url).then()
      第一个参数请求的路径    Fetch会返回Promise    所以我们可以使用then 拿到请求成功的结果
  */
  fetch('http://localhost:3000/fdata').then(function(data){
    // text()方法属于fetchAPI的一部分，它返回一个Promise实例对象，用于获取后台返回的数据
    return data.text();
  }).then(function(data){
    //    在这个then里面我们能拿到最终的数据
    console.log(data);
  })
</script>
```

fetch API 中的 HTTP 请求

- fetch(url, options).then()
- HTTP协议，它给我们提供了很多的方法，如POST，GET，DELETE，UPDATE，PATCH和PUT
  - 默认的是 GET 请求
  - 需要在 options 对象中 指定对应的 method method:请求使用的方法
  - post 和 普通 请求的时候 需要在options 中 设置 请求头 headers 和 body

```
 <script type="text/javascript">
      /*
            Fetch API 调用接口传递参数
      */
     #1.1 GET参数传递 - 传统URL  通过url  ?  的形式传参
      fetch('http://localhost:3000/books?id=123', {
              # get 请求可以省略不写 默认的是GET
              method: 'get'
          })
          .then(function(data) {
              # 它返回一个Promise实例对象，用于获取后台返回的数据
              return data.text();
          }).then(function(data) {
              # 在这个then里面我们能拿到最终的数据
              console.log(data)
          });

     #1.2  GET参数传递  restful形式的URL  通过/ 的形式传递参数  即  id = 456 和id后台的配置有关
      fetch('http://localhost:3000/books/456', {
              # get 请求可以省略不写 默认的是GET
              method: 'get'
          })
          .then(function(data) {
              return data.text();
          }).then(function(data) {
              console.log(data)
```

```
    });

#2.1  DELETE请求方式参数传递      删除id  是  id=789
 fetch('http://localhost:3000/books/789', {
        method: 'delete'
    })
    .then(function(data) {
        return data.text();
    }).then(function(data) {
        console.log(data)
    });

#3 POST请求传参
 fetch('http://localhost:3000/books', {
        method: 'post',
        # 3.1  传递数据
        body: 'uname=lisi&pwd=123',
        #  3.2  设置请求头
        headers: {
            'Content-Type': 'application/x-www-form-urlencoded'
        }
    })
    .then(function(data) {
        return data.text();
    }).then(function(data) {
        console.log(data)
    });

# POST请求传参
 fetch('http://localhost:3000/books', {
        method: 'post',
        body: JSON.stringify({
            uname: '张三',
            pwd: '456'
        }),
        headers: {
            'Content-Type': 'application/json'
        }
    })
    .then(function(data) {
        return data.text();
    }).then(function(data) {
        console.log(data)
    });

 # PUT请求传参     修改id 是 123 的
 fetch('http://localhost:3000/books/123', {
        method: 'put',
        body: JSON.stringify({
            uname: '张三',
            pwd: '789'
        }),
        headers: {
```

```
                    'Content-Type': 'application/json'
                }
            })
            .then(function(data) {
                return data.text();
            }).then(function(data) {
                console.log(data)
            });
    </script>
```

fetchAPI 中 响应格式

- 用fetch来获取数据，如果响应正常返回，我们首先看到的是一个response对象，其中包括返回的一堆原始字节，这些字节需要在收到后，需要我们通过调用方法将其转换为相应格式的数据，比如`JSON`，`BLOB`或者`TEXT`等等

```
    /*
      Fetch响应结果的数据格式
    */
    fetch('http://localhost:3000/json').then(function(data){
      // return data.json();    //  将获取到的数据使用 json 转换对象
      return data.text(); //  //  将获取到的数据 转换成字符串
    }).then(function(data){
      // console.log(data.uname)
      // console.log(typeof data)
      var obj = JSON.parse(data);
      console.log(obj.uname,obj.age,obj.gender)
    })
```

axios

- 基于promise用于浏览器和node.js的http客户端
- 支持浏览器和node.js
- 支持promise
- 能拦截请求和响应
- 自动转换JSON数据
- 能转换请求和响应数据

axios基础用法

- get和 delete请求传递参数
    - 通过传统的url 以 ? 的形式传递参数
    - restful 形式传递参数
    - 通过params 形式传递参数
- post 和 put 请求传递参数
    - 通过选项传递参数
    - 通过 URLSearchParams 传递参数

```
# 1. 发送get 请求
axios.get('http://localhost:3000/adata').then(function(ret){
  #  拿到 ret 是一个对象       所有的对象都存在 ret 的data 属性里面
  // 注意data属性是固定的用法，用于获取后台的实际数据
  // console.log(ret.data)
  console.log(ret)
})
# 2.  get 请求传递参数
# 2.1  通过传统的url  以 ? 的形式传递参数
axios.get('http://localhost:3000/axios?id=123').then(function(ret){
  console.log(ret.data)
})
# 2.2  restful 形式传递参数
axios.get('http://localhost:3000/axios/123').then(function(ret){
  console.log(ret.data)
})
# 2.3  通过params  形式传递参数
axios.get('http://localhost:3000/axios', {
  params: {
    id: 789
  }
}).then(function(ret){
  console.log(ret.data)
})
#3 axios delete 请求传参      传参的形式和 get 请求一样
axios.delete('http://localhost:3000/axios', {
  params: {
    id: 111
  }
}).then(function(ret){
  console.log(ret.data)
})

# 4  axios 的 post 请求
# 4.1  通过选项传递参数
axios.post('http://localhost:3000/axios', {
  uname: 'lisi',
  pwd: 123
}).then(function(ret){
  console.log(ret.data)
})
# 4.2  通过 URLSearchParams  传递参数
var params = new URLSearchParams();
params.append('uname', 'zhangsan');
params.append('pwd', '111');
axios.post('http://localhost:3000/axios', params).then(function(ret){
  console.log(ret.data)
})

#5  axios put 请求传参   和 post 请求一样
axios.put('http://localhost:3000/axios/123', {
  uname: 'lisi',
  pwd: 123
```

```
    }).then(function(ret){
      console.log(ret.data)
    })
```

axios 全局配置

```
#  配置公共的请求头
axios.defaults.baseURL = 'https://api.example.com';
#  配置 超时时间
axios.defaults.timeout = 2500;
#  配置公共的请求头
axios.defaults.headers.common['Authorization'] = AUTH_TOKEN;
# 配置公共的 post 的 Content-Type
axios.defaults.headers.post['Content-Type'] = 'application/x-www-form-urlencoded';
```

axios 拦截器

- 请求拦截器
  - 请求拦截器的作用是在请求发送前进行一些操作
    - 例如在每个请求体里加上token，统一做了处理如果以后要改也非常容易
- 响应拦截器
  - 响应拦截器的作用是在接收到响应后进行一些操作
    - 例如在服务器返回登录状态失效，需要重新登录的时候，跳转到登录页

```
# 1. 请求拦截器
axios.interceptors.request.use(function(config) {
  console.log(config.url)
  # 1.1  任何请求都会经过这一步    在发送请求之前做些什么
  config.headers.mytoken = 'nihao';
  # 1.2  这里一定要return    否则配置不成功
  return config;
}, function(err){
   #1.3 对请求错误做点什么
  console.log(err)
})
#2. 响应拦截器
axios.interceptors.response.use(function(res) {
  #2.1  在接收响应做些什么
  var data = res.data;
  return data;
}, function(err){
  #2.2 对响应错误做点什么
  console.log(err)
})
```

async 和 await

- async作为一个关键字放到函数前面
  - 任何一个`async`函数都会隐式返回一个`promise`
- `await`关键字只能在使用`async`定义的函数中使用
  - await后面可以直接跟一个 Promise实例对象
  - await函数不能单独使用
- **async/await 让异步代码看起来、表现起来更像同步代码**

```
# 1.  async 基础用法
# 1.1 async作为一个关键字放到函数前面
async function queryData() {
  # 1.2 await关键字只能在使用async定义的函数中使用       await后面可以直接跟一个 Promise实例对象
  var ret = await new Promise(function(resolve, reject){
    setTimeout(function(){
      resolve('nihao')
    },1000);
  })
  // console.log(ret.data)
  return ret;
}
# 1.3 任何一个async函数都会隐式返回一个promise    我们可以使用then 进行链式编程
queryData().then(function(data){
  console.log(data)
})

#2.  async    函数处理多个异步函数
axios.defaults.baseURL = 'http://localhost:3000';

async function queryData() {
  # 2.1  添加await之后 当前的await 返回结果之后才会执行后面的代码

  var info = await axios.get('async1');
  #2.2  让异步代码看起来、表现起来更像同步代码
  var ret = await axios.get('async2?info=' + info.data);
  return ret.data;
}

queryData().then(function(data){
  console.log(data)
})
```

图书列表案例

1. 基于接口案例-获取图书列表

- 导入axios 用来发送ajax
- 把获取到的数据渲染到页面上

```
<div id="app">
      <div class="grid">
          <table>
              <thead>
```

```
                <tr>
                    <th>编号</th>
                    <th>名称</th>
                    <th>时间</th>
                    <th>操作</th>
                </tr>
            </thead>
            <tbody>
                <!-- 5.  把books  中的数据渲染到页面上   -->
                <tr :key='item.id' v-for='item in books'>
                    <td>{{item.id}}</td>
                    <td>{{item.name}}</td>
                    <td>{{item.date }}</td>
                    <td>
                        <a href="">修改</a>
                        <span>|</span>
                        <a href="">删除</a>
                    </td>
                </tr>
            </tbody>
        </table>
    </div>
</div>
<script type="text/javascript" src="js/vue.js"></script>
1.  导入axios
<script type="text/javascript" src="js/axios.js"></script>
<script type="text/javascript">
    /*
         图书管理-添加图书
     */
    # 2   配置公共的url地址  简化后面的调用方式
    axios.defaults.baseURL = 'http://localhost:3000/';
    axios.interceptors.response.use(function(res) {
        return res.data;
    }, function(error) {
        console.log(error)
    });

    var vm = new Vue({
        el: '#app',
        data: {
            flag: false,
            submitFlag: false,
            id: '',
            name: '',
            books: []
        },
        methods: {
            # 3 定义一个方法 用来发送 ajax
            # 3.1  使用 async  来 让异步的代码  以同步的形式书写
            queryData: async function() {
                // 调用后台接口获取图书列表数据
                // var ret = await axios.get('books');
```

```
                    // this.books = ret.data;
                    # 3.2  发送ajax请求  把拿到的数据放在books 里面
                    this.books = await axios.get('books');
                }
            },

            mounted: function() {
                #  4 mounted  里面 DOM已经加载完毕  在这里调用函数
                this.queryData();
            }
        });
    </script>
```

2 添加图书

- 获取用户输入的数据 发送到后台
- 渲染最新的数据到页面上

```
methods: {
    handle: async function(){
          if(this.flag) {
            // 编辑图书
            // 就是根据当前的ID去更新数组中对应的数据
            this.books.some((item) => {
              if(item.id == this.id) {
                item.name = this.name;
                // 完成更新操作之后，需要终止循环
                return true;
              }
            });
            this.flag = false;
          }else{
            # 1.1  在前面封装好的 handle 方法中  发送ajax请求
            # 1.2  使用async  和 await 简化操作 需要在 function 前面添加 async
            var ret = await axios.post('books', {
              name: this.name
            })
            # 1.3  根据后台返回的状态码判断是否加载数据
            if(ret.status == 200) {
             # 1.4  调用 queryData 这个方法  渲染最新的数据
              this.queryData();
            }
          }
          // 清空表单
          this.id = '';
          this.name = '';
        },
}
```

3 验证图书名称是否存在

- 添加图书之前发送请求验证图示是否已经存在

- 如果不存在 往后台里面添加图书名称
    - 图书存在与否只需要修改submitFlag的值即可

```
watch: {
       name: async function(val) {
         // 验证图书名称是否已经存在
         // var flag = this.books.some(function(item){
         //   return item.name == val;
         // });
         var ret = await axios.get('/books/book/' + this.name);
         if(ret.status == 1) {
           // 图书名称存在
           this.submitFlag = true;
         }else{
           // 图书名称不存在
           this.submitFlag = false;
         }
       }
},
```

4. 编辑图书

- 根据当前书的id 查询需要编辑的书籍
- 需要根据状态位判断是添加还是编辑

```
methods: {
       handle: async function(){
         if(this.flag) {
           #4.3 编辑图书    把用户输入的信息提交到后台
           var ret = await axios.put('books/' + this.id, {
             name: this.name
           });
           if(ret.status == 200){
             #4.4  完成添加后 重新加载列表数据
             this.queryData();
           }
           this.flag = false;
         }else{
           // 添加图书
           var ret = await axios.post('books', {
             name: this.name
           })
           if(ret.status == 200) {
             // 重新加载列表数据
             this.queryData();
           }
         }
         // 清空表单
         this.id = '';
         this.name = '';
       },
       toEdit: async function(id){
```

```
        #4.1  flag状态位用于区分编辑和添加操作
        this.flag = true;
        #4.2  根据id查询出对应的图书信息   页面中可以加载出来最新的信息
        # 调用接口发送ajax 请求
        var ret = await axios.get('books/' + id);
        this.id = ret.id;
        this.name = ret.name;
      },
```

5 删除图书

- 把需要删除的id书籍 通过参数的形式传递到后台

```
deleteBook: async function(id){
       // 删除图书
       var ret = await axios.delete('books/' + id);
       if(ret.status == 200) {
         // 重新加载列表数据
         this.queryData();
       }
}
```