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
  - 在输入框中，v-model 默认是同步数据，使用 .lazy 会转变为在 change 事件中同步 ，也就是在失去焦点 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
```

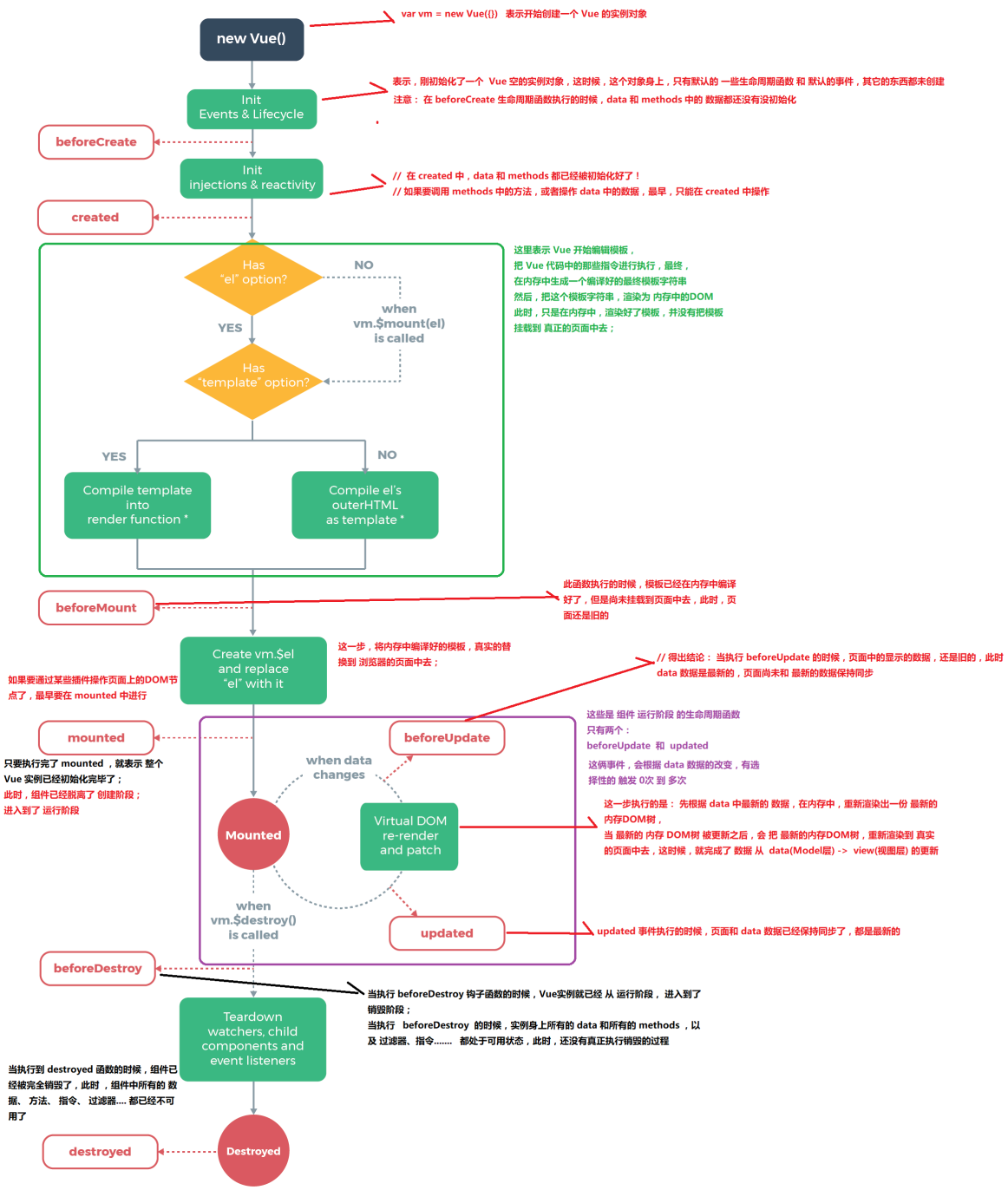
```
        // 第3个参数  b 对应 实参  arg2 字符串
        Vue.filter('filterA',function(n,a,b){
            if(n<10){
                return n+a;
            }else{
                return n+b;
            }
        });

        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


new Vue()
var vm = new Vue({}) 表示开始创建一个 Vue 的实例对象
Init
Events & Lifecycle
表示，刚初始化了一个 Vue 空的实例对象，这时候，这个对象身上，只有默认的 一些生命周期函数 和 默认的事件，其它的东西都未创建
注意： 在 beforeCreate 生命周期函数执行的时候，data 和 methods 中的 数据都还没有没初始化
beforeCreate
Init
injections & reactivity
// 在 created 中，data 和 methods 都已经被初始化好了！
// 如果要调用 methods 中的方法，或者操作 data 中的数据，最早，只能在 created 中操作
created
Has
"el" option?
NO
when
vm.$mount(el)
is called
YES
Has
"template" option?
YES
NO
Compile template
into
render function *
Compile el's
outerHTML
as template *
这里表示 Vue 开始编辑模板，
把 Vue 代码中的那些指令进行执行，最终，
在内存中生成一个编译好的最终模板字符串
然后，把这个模板字符串，渲染为 内存中的DOM
此时，只是在内存中，渲染好了模板，并没有把模板
挂载到 真正的页面中去；
beforeMount
此函数执行的时候，模板已经在内存中编译好了，但是尚未挂载到页面中去，此时，页面还是旧的
Create vm.$el
and replace
"el" with it
这一步，将内存中编译好的模板，真实的替换到 浏览器的页面中去；
如果要通过某些插件操作页面上的DOM节点了，最早要在 mounted 中进行
// 得出结论： 当执行 beforeUpdate 的时候，页面中的显示的数据，还是旧的，此时 data 数据是最新的，页面尚未和 最新的数据保持同步
mounted
只要执行完了 mounted ，就表示 整个 Vue 实例已经初始化完毕了；
此时，组件已经脱离了 创建阶段；
进入到了 运行阶段
beforeUpdate
这些是 组件 运行阶段 的生命周期函数
只有两个：
beforeUpdate 和 updated
这俩事件，会根据 data 数据的改变，有选择性的 触发 0次 到 多次
when data
changes
Mounted
Virtual DOM
re-render
and patch
这一步执行的是： 先根据 data 中最新的 数据，在内存中，重新渲染出一份 最新的内存DOM树，
当 最新的 内存 DOM树 被更新之后，会 把 最新的内存DOM树，重新渲染到 真实的页面中去，这时候，就完成了 数据 从 data(Model层) -> view(视图层) 的更新
when
vm.$destroy()
is called
updated
updated 事件执行的时候，页面和 data 数据已经保持同步了，都是最新的
beforeDestroy
当执行 beforeDestroy 钩子函数的时候，Vue实例就已经 从 运行阶段， 进入到了 销毁阶段；
当执行 beforeDestroy 的时候，实例身上所有的 data 和所有的 methods ，以及 过滤器、指令..... 都处于可用状态，此时，还没有真正执行销毁的过程
Teardown
watchers, child
components and
event listeners
当执行到 destroyed 函数的时候，组件已经被完全销毁了，此时 ，组件中所有的 数据、 方法、 指令、 过滤器.... 都已经不可用了
destroyed
Destroyed

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

```
beforeCreate() { // 这是我们遇到的第一个生命周期函数，表示实例完全被创建出来之前，会执行它
        // console.log(this.msg)
        // this.show()
        // 注意： 在 beforeCreate 生命周期函数执行的时候，data 和 methods 中的数据都还没有没初始化
      },
created() { // 这是遇到的第二个生命周期函数
        // console.log(this.msg)
        // this.show()
        //  在 created 中，data 和 methods 都已经被初始化好了！
        // 如果要调用 methods 中的方法，或者操作 data 中的数据，最早，只能在 created 中操作
      },
beforeMount() { // 这是遇到的第3个生命周期函数，表示模板已经在内存中编辑完成了，但是尚未把模板渲染到页面中
        // console.log(document.getElementById('h3').innerText)
        // 在 beforeMount 执行的时候，页面中的元素，还没有被真正替换过来，只是之前写的一些模板字符串
      },
mounted() { // 这是遇到的第4个生命周期函数，表示内存中的模板，已经真实的挂载到了页面中，用户已经可以看到渲染好的页面了
        // console.log(document.getElementById('h3').innerText)
        // 注意： mounted 是 实例创建期间的最后一个生命周期函数，当执行完 mounted 就表示，实例已经被完全创建好了，此时，如果没有其它操作的话，这个实例，就在我们的内存中
      },
      // 接下来的是运行中的两个事件
beforeUpdate() { // 这时候，表示 我们的界面还没有被更新【数据被更新了吗？　数据肯定被更新了】
        /* console.log('界面上元素的内容：' + document.getElementById('h3').innerText)
        console.log('data 中的 msg 数据是：' + this.msg) */
        // 得出结论： 当执行 beforeUpdate 的时候，页面中的显示的数据，还是旧的，此时data数据是最新的，页面尚未和最新的数据保持同步
      },
updated() {
        console.log('界面上元素的内容：' + document.getElementById('h3').innerText)
```

```
        console.log('data 中的 msg 数据是：' + this.msg)
        // updated 事件执行的时候，页面和 data 数据已经保持同步了，都是最新的
      }
```

es6常用方法

find()和 findIndex()

find()方法，用于找出第一个符合条件的数组成员。它的参数是一个回调函数，所有数组成员依次执行该回调函数，直到找出第一个返回值为true的成员，然后返回该成员。如果没有符合条件的成员，则返回undefined。

```
[1, 2, 5, -1, 9].find((n) => n < 0)
//找出数组中第一个小于 0 的成员
// -1
```

find()方法的回调函数可以接受三个参数，依次为当前的值、当前的位置和原数组

findIndex() 返回第一个符合条件的数组成员的位置，如果所有成员都不符合条件，则返回-1。

```
[1, 2, 5, -1, 9].findIndex((n) => n < 0)
//返回符合条件的值的位置（索引）
// 3
```

filter()

filter()方法使用指定的函数测试所有元素，并创建一个包含所有通过测试的元素的新数组。

filter 为数组中的每个元素调用一次 callback 函数，并利用所有使得 callback 返回 true 或 等价于 true 的值 的元素创建一个新数组。那些没有通过 callback 测试的元素会被跳过，不会被包含在新数组中。filter 不会改变原数组。

```
var arr = [10, 20, 30, 40, 50]
var newArr = arr.filter(item => item > 30);
console.log(newArr); //[40, 50]
```

```
//数组去重
var arr = [1, 2, 2, 3, 4, 5, 5, 6, 7, 7, 8, 8, 0, 8, 6, 3, 4, 56, 2];
var arr2 = arr.filter((x, index,self)=>self.indexOf(x) === index)
console.log(arr2); // [1, 2, 3, 4, 5, 6, 7, 8, 0, 56]
```

forEach()

遍历数组全部元素，利用回调函数对数组进行操作，自动遍历整个数组，且无法break中途跳出循环，不可控，不支持return操作输出，return只用于控制循环是否跳出当前循环。

回调有三个参数：第一个参数是遍历的数组内容，第二个参数是对应的数组索引，第三个参数是数组本身。

```
var arr = [1,2,3,4,5,] ;
arr.forEach(function(item,index){
  console.log(item);
});
```

这个方法是**没有返回值的**，仅仅是遍历数组中的每一项，不对原来数组进行修改；

但是可以自己通过数组的索引来**修改原来的数组**；

some() 和 every()

every()与some()方法都是JS中数组的迭代方法， 只返回布尔值。

`every()`

判断数组中是否每个元素都满足条件

只有都满足条件才返回true； 只要有一个不满足就返回false；

`some()`

判断数组中是否至少有一个元素满足条件 只要有一个满足就返回true 只有都不满足时才返回false

```
// 判断数组arr1是否全是偶数
// 判断数组arr2是否至少有一个偶数
var arr1=[1, 2, 3, 4, 5];
var arr2=[1, 4, 6, 8, 10];
var bool="";
    var bool=arr1.every(function(value, index, array){
        return value % 2 == 0;
    })
    // false
   var  bool=arr2.some(function(value, index, array){
        return value % 2 == 0;
    })
   // true
```

map()

map() 方法返回一个新数组，数组中的元素为原始数组元素调用函数处理后的值。

map() 方法按照原始数组元素顺序依次处理元素。

注意： map() 不会对空数组进行检测。

注意： map() 不会改变原始数组。

`array.map(function(currentValue,index,arr), thisValue)`

```
var data = [1, 2, 3, 4];
var arr = data.map(function (item) { //接收新数组
  return item * item;
});
alert(arr); // [1, 4, 9, 16]
```

reduce()

1.reduce()方法接收一个函数作为累加器(accumulator),数组中的每个值(从左到右)开始合并,最终为一个值.

array.reduce(callback, initialValue) 2.callback:执行数组中每个值的函数(也可以叫做reducer),包含4个参数.

1.previousValue:上一次调用回调返回的值,或者是提供的初始值(initialValue) 2.currentValue:数组中当前被处理的元素 3.index:当前元素在数组中的索引 4.array:调用reduce的数组

```
// 获取购物车中商品列表的价格总和
let goodList = [{id: 1, price: 10, qty: 5}, {id: 2, price: 15, qty: 2}, {id: 3, price: 20, qty: 1}]
let totalPrice = goodList.reduce((prev, cur) => {
    return prev + cur.price * cur.qty
}, 0)
console.log(totalPrice) // 100
```

```
var arrString = 'abcdaabc'
// 获取字符中中每个字母出现的次数
let count = arrString.split('').reduce(function(res, cur) {
    res[cur] ? res[cur]++ : res[cur] = 1
    return res
}, {})
console.log(count) // {a: 3, b: 2, c: 2, d: 1}
```

`	**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: ''
       }]
     }
   });
```

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>
```