vue第二天

课程目标

1. 文本渲染指令
2. 属性绑定指令
3. 事件处理指令
4. 条件渲染指令
5. 循环遍历指令

vue基础语法

一、文本渲染指令

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

在前面，我们一直使用的是字符串差值的形式渲染文本，但是除此方法之外，vue还提供了其他几种常见的文本渲染方式：

1. v-text：更新元素的innerText
2. v-html：更新元素的innerHTML

```
<div id="app">
  <div v-html="msg"></div>
  <div v-text="msg"></div>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {
      msg:'<p>Hello World!</p>'
    }
  });
</script>
```

在Vue中，我们可以使用{{}}将数据插入到相应的模板中，这种方法是一种文本插值。 使用这种方法，如果网络慢或者JavaScript出错的话，会将{{}}直接渲染到页面中。 值得庆幸的是，Vue还提供了v-text和v-html来渲染文本或元素。这样就避免了将{{}}直接渲染到页面中。

二、属性绑定指令

如果想让html标签中的属性，也能应用Vue中的数据，那么就可以使用vue中常用的属性绑定指令：v-bind

```
<div id="app">
  <div v-bind:title="msg">DOM元素属性绑定</div>
  <!-- v-bind的简写形式 -->
  <div :title="msg">DOM元素属性绑定</div>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {
      msg:'Hello World!'
    }
  });
</script>
```

上面展示的是v-bind的最基本的使用，第一种是完整语法，第二种是缩写方式。

除了将元素的title属性和vue实例的相关字段进行绑定外，还能将其他的属性字段进行绑定，最常见的是对于样式的绑定，即class和style属性。

2.1.绑定样式

使用v-bind指令绑定class属性，就可以动态绑定元素样式了。

```
<div id="app">
  <div class="one">DOM元素的样式绑定</div>
</div>
```

```
<head>
  <style>
    .one{
      color: red;
    }
    .two{
      color: blue;
    }
  </style>
</head>
<body>
  <div id="app">
    <div :class="className">DOM元素的样式绑定</div>
  </div>

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

```
  <script>
    let vm = new Vue({
      el: '#app',
      data: {
        className:'one'
      }
    });
  </script>
</body>
```

2.3.使用三目运算绑定样式

可以使用三目运算符，来动态绑定样式。

```
<head>
  <style>
    .one{
      color: red;
    }
  </style>
</head>
<body>
  <div id="app">
    <div :class="userId==1?className:''">DOM元素的样式绑定</div>
  </div>

  <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
  <script>
    let vm = new Vue({
      el: '#app',
      data: {
        userId:1,
        className:'one'
      }
    });
  </script>
</body>
```

2.4.直接绑定内联样式

也可以直接绑定内联样式。

```
<div id="app">
  <div :style="{color:colorValue,fontSize:fontSizeValue}">DOM元素的样式绑定</div>
</div>
```

```
<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {
      colorValue:'red',
      fontSizeValue:'48px'
    }
  });
</script>
```

> 注意：绑定style属性后，样式的书写要遵循javaScript规范。 也就是将 xxx-xxx 改写成驼峰命名方式 xxxXxxx

三、事件处理指令

我们可以用 v-on 指令绑定一个事件监听器，通过它调用我们 Vue 实例中定义的方法。

v-on指令可以简写为：@

```
<div id="app">
  <button v-on:click="pointme">点击我</button>
    <!-- v-on指令的简写 -->
  <button @click="pointyou">点击你</button>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {

    },
    methods:{
      pointme(){
        alert('点击了我');
      },
      pointyou(){
        alert('点击了你');
      }
    }
  });
</script>
```

一个案例：对一个数进行加减运算

```
<div id="app">
  {{num}}
  <button v-on:click="add(1)">加</button>
  <button @click="subtract(1)">减</button>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {
      num:1
    },
    methods:{
      add(value){
        this.num += value;
      },
      subtract(value){
        this.num -= value;
      }
    }
  });
</script>
```

四、条件渲染指令

条件渲染指令，可以根据条件判断，来设置元素的显示与隐藏。

4.1.v-if指令

当v-if的值为false时，网页中将不会对此元素进行渲染

```
▼<div id="app">
    <!---->
  </div>
```

```
<div id="app">
  <div v-if="isShow">DOM元素的样式绑定</div>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {
      isShow:false
    }
  });
</script>
```

4.2.v-else指令和v-else-if指令

我们可以使用 v-else 指令来表示 v-if 的“else 块”，v-else 元素必须紧跟在 v-if 或者 v-else-if 元素的后面——否则它将不会被识别。而v-else-if则是充当 v-if 的“else-if 块”，可以链式地使用多次。

```
<div id="app">
  <div v-if="type==1">A</div>
  <div v-else-if="type==2">B</div>
  <div v-else-if="type==3">C</div>
  <div v-else>D</div>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {
      type:2
    }
  });
</script>
```

4.3.v-if指令和v-show指令

v-show也是根据条件展示元素的指令。带有 v-show 的元素始终会被渲染并保留在 DOM 中。 v-show 是简单地切换元素的 CSS 属性 display 。

```
<div id="app">
  <div v-if="isShow">这里是v-if</div>
  <div v-show="isShow">这里是v-show</div>
```

```
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {
      isShow:false
    }
  });
</script>
```

```
▼<div id="app">
   <!---->
   <div style="display: none;">这里是v-show</div>
 </div>
```

通过上面的例子，我们不难发现两者的不同：

1. v-if是真正的条件渲染，因为它会确保在切换过程中条件块内的事件监听器和子组件适当地被销毁和重建。
2. v-if是惰性的，只有当条件为true时才会渲染，如果条件为false则什么都不做
3. v-if有很高的切换开销，适用于条件不太容易改变的时候
4. v-show不管条件是true还是false都会进行渲染。并且只是简单地基于 CSS 进行切换
5. v-show有很高的初始渲染开销，适用于非常频繁地切换

五、循环遍历指令

vue.js 的循环渲染是依赖于 v-for 指令，它能够根据 vue 的实例里面的信息，循环遍历所需数据，然后渲染出相应的内容。它可以遍历数组类型以及对象类型的数据，js 里面的数组本身实质上也是对象，这里遍历数组和对象的时候，方式相似但又稍有不同。

5.1.遍历对象属性

value 是遍历得到的属性值，key 是遍历得到的属性名，index 是遍历次序，这里的 key、index 都是可选参数，如果不需要，这个指令其实可以写成 v-for="value in user"；

0 : userId : 1

1 : userName : 张三

2 : userSex : 男

1

张三

男

```
<div id="app">
  <p v-for="(value,key,index) in user">{{index}}: {{key}}: {{value}}</p>
  <hr>
  <p v-for="value in user">{{value}}</p>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {
      user:{
        userId:1,
        userName:'张三',
        userSex:'男'
      }
    }
  });
</script>
```

5.2.遍历数组元素

value 是遍历得到的元素，index 是数组下标，这里的index 也是可选参数，如果不需要，这个指令其实可以写成 v-for="value in userArr"；

0,1,张三,男

1,2,李四,女

2,3,王五,男

```
<div id="app">
  <p v-for="(item,index) in userArr">
    {{item.userId}},{{item.userName}},{{item.userSex}}
      <button @click="operate(index)">操作</button>
  </p>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {
      userArr: [{
        userId: 1,
        userName: '张三',
```

```
        userSex: '男'
      }, {
        userId: 2,
        userName: '李四',
        userSex: '女'
      }, {
        userId: 3,
        userName: '王五',
        userSex: '男'
      }]
    },
    methods:{
      operate(index){
        console.log(this.userArr[index]);
      }
    }
  });
</script>
```

六、Vue的表单绑定

v-bind实现了数据的单向绑定，将vue实例中的数据同元素属性值进行绑定，接下来看一下vue中的数据双向绑定v-model。

1.v-model实现表单绑定

```
<div id="app">
  <h3>注册</h3>
  <form>
    用户名：<input type="text" v-model="myForm.username"/><br>
    密码：<input type="password" v-model="myForm.password" /><br>
    确认密码：<input type="password" v-model="myForm.beginpassword" /><br>
    性别：<input type="radio" v-model="myForm.sex" value="0" />男
       <input type="radio" v-model="myForm.sex" value="1" />女<br>
    爱好：<input type="checkbox" v-model="myForm.like" value="0" />读书
       <input type="checkbox" v-model="myForm.like" value="1" />体育
       <input type="checkbox" v-model="myForm.like" value="2" />旅游<br>
    国籍：<select v-model="myForm.nationality">
          <option value="0">中国</option>
        <option value="1">美国</option>
        <option value="2">英国</option>
       </select><br>
    个人简介：<textarea v-model="myForm.brief" rows="5" cols="30"></textarea><br>
       <input type="button" value="提交" @click="handler" />
    </form>
```

```
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {
      myForm: {
        username: '',
        password: '',
        beginpassword: '',
        sex: 0,
        like: [0, 1, 2],
        nationality: 0,
        brief: '',
        level: 0
      }
    },
    methods: {
      handler: function() {
        console.log(this.myForm.username);
        console.log(this.myForm.password);
        console.log(this.myForm.beginpassword);
        console.log(this.myForm.sex);
        console.log(this.myForm.like);
        console.log(this.myForm.nationality);
        console.log(this.myForm.brief);
      }
    }
  });
</script>
```

2.v-model修饰符

v-model中还可以使用一些修饰符来实现某些功能：

1. v-model.lazy 只有在input输入框发生一个blur时才触发，也就是延迟同步到失去焦点时。
2. v-model.trim 将用户输入的前后的空格去掉。
3. v-model.number 将用户输入的字符串转换成number。

```
<div id="app">
    <!-- 输入数据会延迟到失去焦点时才绑定 -->
  {{lazyStr}}<input type="text" v-model.lazy="lazyStr" /><br>
    <!-- 输入数据会自动转换为number，所以可以直接进行运算 -->
  {{numberStr+1}}<input type="text" v-model.number="numberStr" /><br>
    <!-- 输入数据会自动去除前后空格 -->
  {{trimStr}}<input type="text" v-model.trim="trimStr" /><br>
```

```
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {
      lazyStr: '',
      numberStr:'',
      trimStr:''
    }
  });
</script>
```

七、综合案例

编号	姓名	年龄	操作
1	张三	20	删除
2	李四	21	删除
3	王五	22	删除
	清空		

添加

姓名：
年龄：0
添加

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title></title>
    <style>
      #app{
        width: 500px;
      }
      table{
        width: 100%;
        border-collapse: collapse;
```

```
    }
    table tr th,table tr td{
      height: 35px;
      border-bottom: solid 1px #999;
      text-align: center;
    }
    .clear-btn{
      text-align: right;
    }
  </style>
</head>
<body>
  <div id="app">
    <table>
      <tr>
        <th>编号</th>
        <th>姓名</th>
        <th>年龄</th>
        <th>操作</th>
      </tr>
      <tr v-for="(obj,index) in arr">
        <td>{{obj.userId}}</td>
        <td>{{obj.userName}}</td>
        <td>{{obj.userAge}}</td>
        <td><button @click="del(index)">删除</button></td>
      </tr>
      <tr>
        <td colspan="4"><button @click="clear">清空</button></td>
      </tr>
    </table>

    <h3>添加</h3>
    姓名：<input type="text" v-model="userName"><br>
    年龄：<input type="text" v-model="userAge"><br>
    <button @click="add">添加</button>
  </div>

  <script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
  <script>
    let app = new Vue({
      el:'#app',
      data:{
        arr:[
          {
            userId:1,
            userName:'张三',
            userAge:20
          },{
            userId:2,
```

```
            userName:'李四',
            userAge:21
          },{
            userId:3,
            userName:'王五',
            userAge:22
          }
        ],
        userName:'',
        userAge:0
      },
      methods:{
        add(){
          let userId = 0;
          if(this.arr.length==0){
            userId = 1;
          }else{
            userId = this.arr[this.arr.length-1].userId+1;
          }
          this.arr.push({
            userId:userId,
            userName:this.userName,
            userAge:this.userAge
          });
        },
        del(index){
          this.arr.splice(index,1);
        },
        clear(){
          this.arr.splice(0,this.arr.length);
          //this.arr = [];
        }
      }
    });
  </script>
 </body>
</html>
```

vue进阶：方法、计算属性及监听器

在vue中处理复杂的逻辑的时候，我们经常使用计算属性、方法及监听器。

1. methods：方法：它们是挂载在Vue对象上的函数，通常用于做事件处理函数，或自己封装的自定义函数。
2. computed：计算属性：在Vue中，我们可以定义一个计算属性，这个计算属性的值，可以依赖于某个data中的数据。或者说：计算属性是对数据的再加工处理。
3. watch：监听器：如果我们想要在数据发生改变时做一些业务处理，或者响应某个特定的变化，我们就可以通过监听器，监听数据的变化，从而做出相应的反应。

一、computed 计算属性

计算属性是根据依赖关系进行缓存的计算，并且只在需要的时候进行更新。

```
<div id="app">
  <p>原数据：{{msg}}</p>
  <p>新数据：{{reversedMsg}}</p>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {
      msg:'hello world!'
    },
    computed:{
      reversedMsg(){
        return this.msg.split('').reverse().join('');
      }
    }
  });
</script>
```

一个案例：根据商品数量修改总价

```
<div id="app">
  <table width="100%" style="text-align: center;">
    <tr>
      <th>商品编号</th>
      <th>商品名称</th>
      <th>商品单价</th>
      <th>商品数量</th>
      <th>合计</th>
    </tr>
    <tr>
      <td>1</td>
      <td>小米10</td>
      <td>{{price}}</td>
      <td>
        <button @click="subtract">-</button>
        {{quantity}}
        <button @click="add">+</button>
      </td>
      <td>{{totalPrice}}</td>
    </tr>
  </table>
```

```
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {
      price:2999,
      quantity:1
    },
    computed:{
      totalPrice(){
        return this.price*this.quantity;
      }
    },
    methods:{
      add(){
        this.quantity++;
      },
      subtract(){
        this.quantity--;
      }
    }
  });
</script>
```

二、methods 方法

在使用vue的时候，可能会用到很多的方法，它们可以将功能连接到事件的指令，甚至只是创建一个小的逻辑就像其他函数一样被重用。接下来我们用方法实现上面的字符串反转。

```
<div id="app">
  <p>原数据：{{msg}}</p>
  <p>新数据：{{reversedMsg()}}</p>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {
      msg:'hello world!'
    },
    methods:{
      reversedMsg(){
        return this.msg.split('').reverse().join('');
      }
```

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

虽然使用computed和methods方法来实现反转，两种方法得到的结果是相同的，但本质是不一样的。

计算属性是基于它们的依赖进行缓存的。计算属性只有在它的相关依赖发生改变的时候才会重新求值，这就意味着只要message还没有发生改变，多次访问reversedMessage计算属性立即返回的是之前计算的结果，而不会再次执行计算函数。

而对于methods方法，只要发生重新渲染，methods调用总会执行该函数。

如果某个computed需要的遍历一个极大的数组和做大量的计算，可以减小性能开销，如果不希望有缓存，则用methods。

三、watch 监听器

watch能够监听数据的改变。监听之后会调用一个回调函数。
此回调函数的参数有两个：

1. 更新后的值（新值）
2. 更新前的值（旧值）

1.监听基本数据类型

下面使用watch来监听商品数量的变化。如果商品数量小于1，就重置成上一个值。

```
<div id="app">
  <table width="100%" style="text-align: center;">
    <tr>
      <th>商品编号</th>
      <th>商品名称</th>
      <th>商品单价</th>
      <th>商品数量</th>
      <th>合计</th>
    </tr>
    <tr>
      <td>1</td>
      <td>小米10</td>
      <td>{{price}}</td>
      <td>
        <button @click="subtract">-</button>
        {{quantity}}
        <button @click="add">+</button>
      </td>
      <td>{{totalPrice}}</td>
```

```
    </tr>
  </table>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {
      price: 2999,
      quantity: 1
    },
    computed: {
      totalPrice() {
        return this.price * this.quantity;
      }
    },
    methods: {
      add() {
        this.quantity++;
      },
      subtract() {
        this.quantity--;
      }
    },
    watch:{
      quantity(newVal,oldVal){
        console.log(newVal,oldVal);
        this.quantity = newVal<=0?oldVal:newVal
      }
    }
  });
</script>
```

2.深度监听

在上面的例子中，监听的简单的数据类型，数据改变很容易观察，但是当需要监听的数据变为对象类型的时候，上面的监听方法就失效了，因为上面的简单数据类型属于浅度监听，对应的对象类型就需要用到深度监听，只需要在上面的基础上加上deep: true就可以了。

```
<div id="app">
  <table width="100%" style="text-align: center;">
    <tr>
      <th>商品编号</th>
      <th>商品名称</th>
      <th>商品单价</th>
      <th>商品数量</th>
```

```
      <th>合计</th>
    </tr>
    <tr>
      <td>1</td>
      <td>小米10</td>
      <td>{{goods.price}}</td>
      <td>
        <button @click="subtract">-</button>
        {{goods.quantity}}
        <button @click="add">+</button>
      </td>
      <td>{{totalPrice}}</td>
    </tr>
  </table>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {
      goods:{
        price: 2999,
        quantity: 1
      }
    },
    computed: {
      totalPrice() {
        return this.goods.price * this.goods.quantity;
      }
    },
    methods: {
      add() {
        this.goods.quantity++;
      },
      subtract() {
        this.goods.quantity--;
      }
    },
    watch:{
      goods:{
        handler(newVal,oldVal){
                    /**
                     * 注意：虽然使用深度监听，可以监听到对象的改变。
                     *       但是，由于是对象类型，所以newVal与oldVal都指向同一个对象。
                     *       所以，newVal与oldVal中的quantity都是改变后的新值。
                     */
          console.log(newVal,oldVal);
          this.goods.quantity = newVal.quantity<=0?oldVal.quantity:newVal.quantity;
```

```
        },
        deep:true
      }
    }
  });
</script>
```

四、Vue操作数组时的注意事项

由于 JavaScript 语言本身的限制，当我们对数组进行某些操作时，Vue 可能不会检测到数组元素的变化，那么进而视图也不会响应。比如：

```
<div id="app">
  <p v-for="(item,index) in userArr">
    {{item.userId}},{{item.userName}},{{item.userSex}}
  </p>
  <button @click="clear">清空</button>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {
      userArr: [{
        userId: 1,
        userName: '张三',
        userSex: '男'
      }, {
        userId: 2,
        userName: '李四',
        userSex: '女'
      }, {
        userId: 3,
        userName: '王五',
        userSex: '男'
      }]
    },
    methods: {
      clear() {
        this.userArr.length = 0;
                console.log(this.userArr.length);    //0
      }
    }
  });
</script>
```

上面实例中，将数组的长度设置为0后，数组确实被清空了。但是Vue却不能检测到数组的变化，所以页面视图也不会响应。

为了解决这个问题，Vue给我们提供了如下解决方案：
Vue 提供一组观察数组的变异方法（就是这些方法会改变原始数组，所以才会被Vue检测到），使用它们就可以触发视图更新。
这些方法有7个：push()、pop()、shift()、unshift()、splice()、sort()、reverse()

```
<div id="app">
  <p v-for="(item,index) in userArr">
    {{item.userId}},{{item.userName}},{{item.userSex}}
  </p>
  <button @click="clear">清空</button>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  let vm = new Vue({
    el: '#app',
    data: {
      userArr: [{
        userId: 1,
        userName: '张三',
        userSex: '男'
      }, {
        userId: 2,
        userName: '李四',
        userSex: '女'
      }, {
        userId: 3,
        userName: '王五',
        userSex: '男'
      }]
    },
    methods: {
      clear() {
                //使用Vue中提供的变异方法
        this.userArr.splice(0,this.userArr.length);
      }
    }
  });
</script>
```

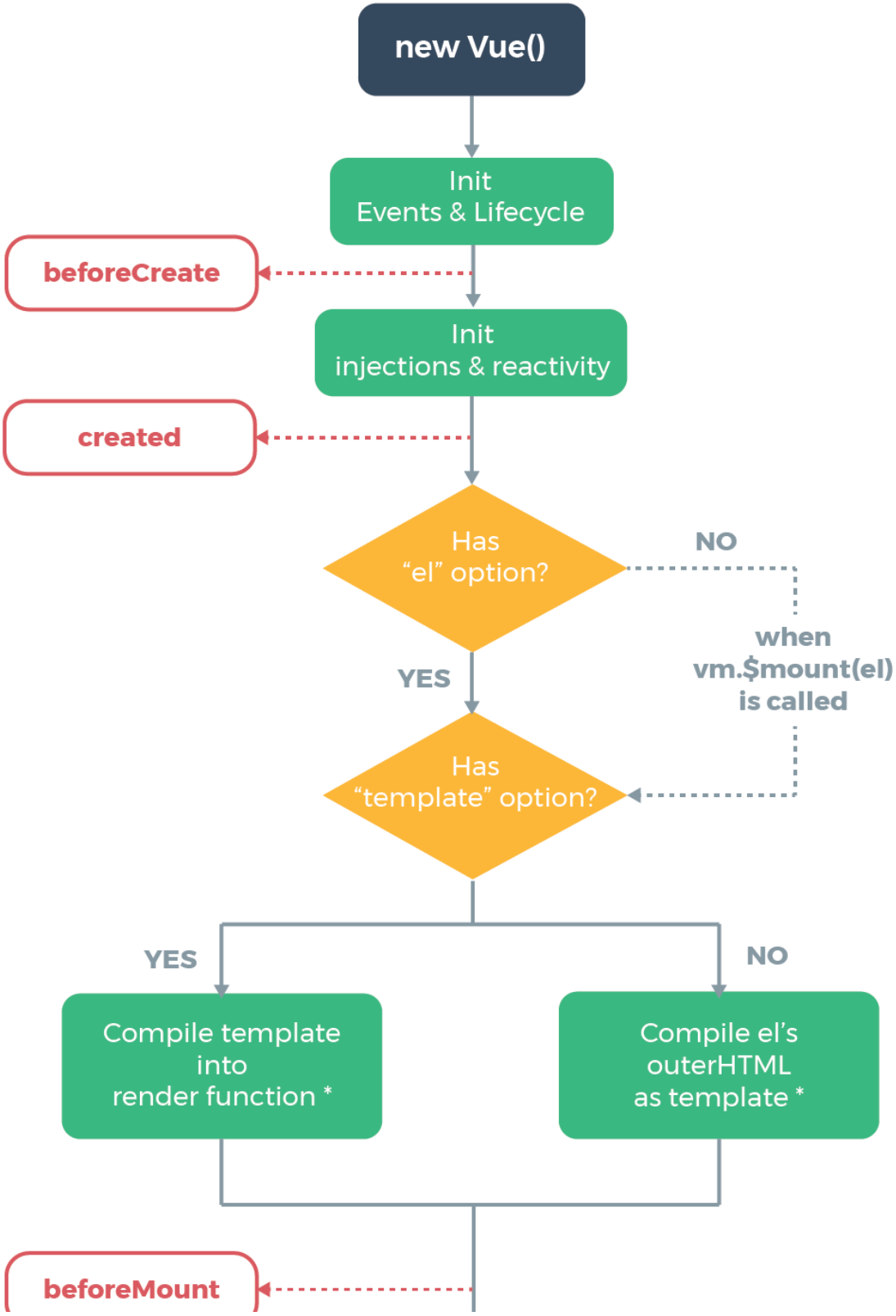
Vue生命周期

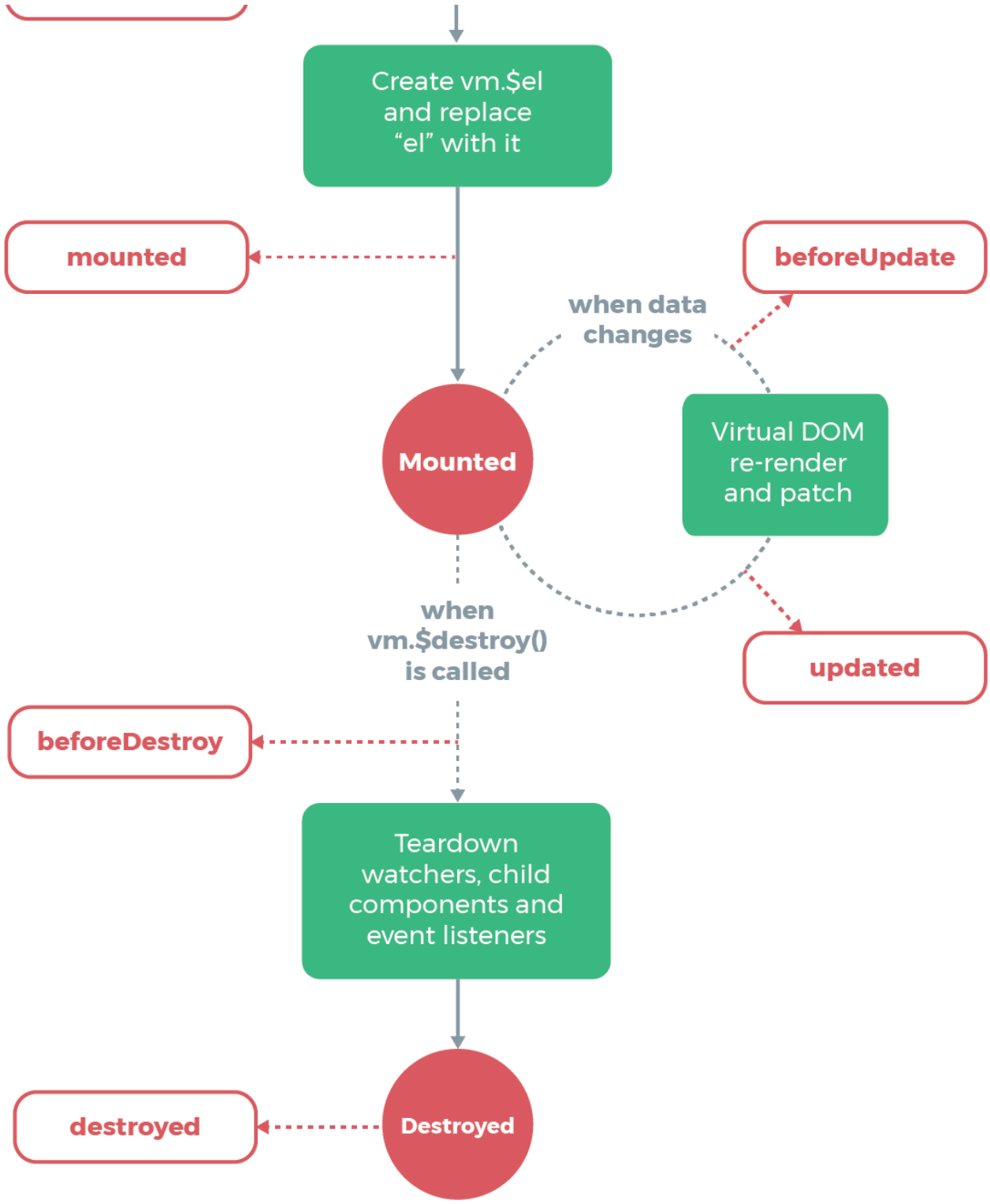
在使用vue进行日常开发中，我们总有这样的需求，想在页面刚一加载出这个表格组件时，就发送请求去后台拉取数据，亦或者想在组件加载前显示个loading图，当组件加载出来就让这个loading图消失等等这样或那样的需求。

要实现这些需求，最重要的一点就是我怎么知道这个组件什么时候加载，换句话说我该什么时候向后台发送请求，为了解决这种问题，组件的生命周期钩子函数就应运而生。

1.Vue生命周期图示

下面这张图，就是Vue官网给我们展示的Vue生命周期图：





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

这是官方文档给出的一个组件从被创建出来到最后被销毁所要经历的一系列过程，所以这个过程也叫做一个组件的生命周期图。从图中我们可以看到，一个组件从被创建到最后被销毁，总共要经历以下8个过程：

1. beforeCreate:组件实例创建之前
2. created：组件实例创建完毕
3. beforeMount：组件DOM挂载之前
4. mounted：组件DOM挂载完毕
5. beforeUpate：组件数据更新之前
6. updated：组件数据更新完毕
7. beforeDestoy：组件实例销毁之前
8. destoyed：组件实例销毁完毕

2.代码演示

```
<div id="app">
  <h1>{{num}}</h1>
  <button @click="add">加</button>
</div>

<script src="https://cdn.jsdelivr.net/npm/vue/dist/vue.js"></script>
<script>
  var app = new Vue({
    el: '#app',
    data: {
      num:10
    },
    //组件实例创建之前
    beforeCreate() {
      console.log('beforeCreate 组件实例创建前');
    },
    //组件实例创建完毕
    created() {
      console.log('created 组件实例创建完毕');
    },
    //组件DOM挂载之前
    beforeMount() {
      console.log('beforeMount 组件DOM挂载前');
    },
    //组件DOM挂载完毕
    mounted() {
      console.log('mounted 组件DOM挂载完毕');
    },
    //组件数据更新之前
    beforeUpdate() {
      console.log('beforeUpdate 组件数据更新前');
    },
    //组件数据更新完毕
    updated() {
      console.log('updated 组件数据更新完毕');
    },
```

```
    //组件实例销毁之前
    beforeDestroy() {
      console.log('beforeDestroy 组件实例销毁前');
    },
    //组件实例销毁完毕
    destroyed() {
      console.log('destroyed 组件实例销毁完毕');
    },
    methods:{
      //改变数据，可以演示beforeUpdate与updated
      add(){
        this.num++;
      }
    }
  })
  //使用$destroy()销毁当前实例，可以演示beforeDestroy与destroyed
  //app.$destroy();
</script>
```

3.总结

以上就是vue中组件生命周期钩子函数执行的各个过程以及执行的时机，但是这些钩子函数到底该怎么用呢？针对前言中提出的需求我们又该怎么解决呢？在这里，给大家举个例子：

> 例如有一个表格组件：
>
> 1. 我们想在表格加载之前显示个loading图，那么我们可以在组件实例创建之前的钩子函数beforeCreate里面将loading图显示。
> 2. 当组件实例加载出来，我们可以在created钩子函数里让这个loading图消失。
> 3. 当表格被加载好之后我们想让它马上去拉取后台数据，那么我们可以在组件DOM挂载之前的钩子函数beforeMount里面去发送请求。
> 4. 从后台拉取的数据要绑定在DOM中，那么就必须要在组件DOM挂载完毕的钩子函数mounted里面去做。
> 5. 表格中的数据在更新前和更新后，我们都需要做一个处理，那么这些工作就可以放在beforeUpdate和updated中去做。
> 6. 当应用程序结束后，或组件实例准备销毁时，有一些善后处理的工作（比如释放资源）就可以放在beforeDestroy和destroyed中去做。

Vue路由

一、Vue路由基础

Vue属于单页应用（SPA），即整个应用程序中只有一个html页面。

在单页应用中（SPA），由于只是更改DOM来模拟多页面，所以页面浏览历史记录的功能就丧失了。此时，就需要前端路由来实现浏览历史记录的功能。

```
<div id="app">
  <p>
    <!-- 使用 router-link 组件来导航. to属性指定导航地址-->
    <router-link to="/home">home</router-link>
    <router-link to="/news">news</router-link>
  </p>
  <!-- 路由出口（路由匹配到的组件将渲染在这里） -->
  <router-view></router-view>
</div>

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

<script type="text/javascript">
  // 1. 定义（路由）组件。
  const Home = {
    template: '<div>首页</div>'
  }
  const News = {
    template: '<div>新闻</div>'
  }

  // 2. 定义路由规则对象（每个路由应该映射一个组件）
  const routes = [
    {
      path: '/home',
      component: Home
    },{
      path: '/news',
      component: News
    }
  ]

  // 3. 创建 router 实例，然后传 `routes` 配置
  const router = new VueRouter({
    //如果路由规则对象名也为routes，那么就可以简写为 routes
    routes:routes
  })

  // 4. 将路由对象挂载到Vue实例上
  // 通过 router 配置参数注入路由，从而让整个应用都有路由功能
  var vm = new Vue({
    el: '#app',
    data: {},
```

```
    // 将路由添加到Vue中
    router
  })
</script>
```

> 注意：
>
> 上面代码中，router-link标签默认会被渲染成一个a标签
>
> ```
> ▼
>    <a href="#/home" class="router-link-exact-active router-link-active" aria-current="page">home</a>
>    <a href="#/news" class>news</a>
>  
> ```

路由重定向：上面代码中，我们应该设置打开浏览器就默认调整到 “首页”，所以需要把根路由/重定向到/home。
修改路由配置：

```
// 2. 定义路由规则对象（每个路由应该映射一个组件）
const routes = [
  {
    path: '/',              //根路由
    redirect: '/home'       //把根路由重定向到home
  },{
    path: '/home',
    component: Home
  },{
    path: '/news',
    component: News
  }
]
```

二、嵌套路由（子路由）

实际应用界面，通常由多层嵌套的组件组合而成。 比如，我们 “首页”组件中，还嵌套着 “登录”和 “注册”组件，那么URL对应就是/home/login和/home/reg。

```
<div id="app">
  <p>
    <!-- 使用 router-link 组件来导航. to属性指定导航地址-->
    <router-link to="/home">home</router-link>
    <router-link to="/news">news</router-link>
  </p>
  <!-- 路由出口（路由匹配到的组件将渲染在这里） -->
  <router-view></router-view>
</div>

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

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

<script type="text/javascript">
  // 1. 定义（路由）组件。
  const Home = {
    template: `<div>
            <h2>首页</h2>
            <router-link to="/home/login">登录</router-link>
            <router-link to="/home/reg">注册</router-link>
            <router-view></router-view>
          </div>`
  }

  const News = {
    template: '<div>新闻</div>'
  }

  const Login = {
    template: '<div>登陆</div>'
  }

  const Reg = {
    template: '<div>注册</div>'
  }

  // 2. 定义路由规则对象（每个路由应该映射一个组件）
  const routes = [
    {
      path: '/',                  //根路由
      redirect: '/home'           //把根路由重定向到home
    },{
      path: '/home',
      component: Home,
      children: [                 //配置子路由
        {
          path: '/home',
          redirect: '/home/login'
        },{
          path: '/home/login',
          component: Login
        },{
          path: '/home/reg',
          component: Reg
        }
      ]
    }, {
      path: '/news',
      component: News
    },
```

```
  ]

  // 3. 创建 router 实例，然后传 `routes` 配置
  const router = new VueRouter({
    //如果路由规则对象名也为routes，那么就可以简写为 routes
    routes: routes
  })

  // 4. 将路由对象挂载到Vue实例上
  // 通过 router 配置参数注入路由，从而让整个应用都有路由功能
  var vm = new Vue({
    el: '#app',
    data: {},
    // 将路由添加到Vue中
    router
  })
</script>
```

三、路由传参

路由传参有多种方式，这里我们学习两种：params与query。

3.1.params形式传参

```
<div id="app">
  <p>
    <router-link :to="{name:'Home',params:{msg:'hello world!'}}">home</router-link>
    <router-link :to="{name:'News',params:{id:id,name:name}}">news</router-link>
  </p>
  <router-view></router-view>
</div>

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

<script type="text/javascript">
  const Home = {
    template: '<div>{{$route.params.msg}},首页</div>'
  }
  const News = {
    template: `<div>新闻;
                参数1：{{$route.params.id}};
              参数2：{{$route.params.name}}
            </div>`
  }
```

```
  const routes = [{
    path: '/home',
    name:'Home',                //每个路由规则中必须要有一个name属性
    component: Home
  }, {
    path: '/news',
    name:'News',
    component: News
  }]

  const router = new VueRouter({
    routes
  })

  var vm = new Vue({
    el: '#app',
    data: {
      id:1,
      name:'zhangsan'
    },
    router
  })
</script>
```

> 注意：
>
> 1. 使用v-bind绑定to属性。
> 2. to属性的值是一个json对象，此对象有两个属性：name属性和params属性。
> 3. name属性就是要路由的对象。所以，在路由规则列表中，每一个路由规则都应用有一个name值。
> 4. params属性就是要传递的参数。也是一个json对象。
> 5. 组件接收参数时，使用 this.$route.params.参数名 的形式。

3.2.query形式传参

```
<div id="app">
  <p>
    <router-link :to="{path:'/home',query:{msg:'hello world!'}}">home</router-link>
    <router-link :to="{path:'/news',query:{id:id,name:name}}">news</router-link>
  </p>
  <router-view></router-view>
</div>

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

<script type="text/javascript">
```

```
  const Home = {
    template: '<div>{{$route.query.msg}},首页</div>'
  }
  const News = {
    template: `<div>新闻;
                参数1：{{$route.query.id}};
            参数2：{{$route.query.name}}
          </div>`
  }

  const routes = [{
    path: '/home',
    component: Home
  }, {
    path: '/news',
    component: News
  }]

  const router = new VueRouter({
    routes
  })

  var vm = new Vue({
    el: '#app',
    data: {
      id:1,
      name:'zhangsan'
    },
    router
  })
</script>
```

> 注意：
>
> 1. to属性的值仍然是一个josn对象，但是两个属性变了，一个是path，一个是query。
> 2. path属性就是路由地址，对应路由规则中的path值。
> 3. query属性就是要传递的参数。也是一个json对象。
> 4. 组件接收参数时，使用 this.$route.query.参数名 的形式。

3.3.params方式与query方式的区别

query方式传值：

home news

新闻; 参数1：1； 参数2：zhangsan

params方式传值：

home news

新闻; 参数1：1； 参数2：zhangsan

> 总结：params方式与query方式的区别：
>
> 1. query方式：
>
>   类似于get方式，参数会在路由中显示，可以用做刷新后仍然存在的参数。
>
>   利用路由规则中的path跳转。
> 2. params方式：
>
>   类似于post方式，参数不会在路由中显示，页面刷新后参数将不存在。
>
>   利用路由规则中的name跳转。

四、编程式路由

1、利用JS实现路由跳转

router-link标签可以实现页面超链接形式的路由跳转。但是实际开发中，在很多情况下，需要通过某些逻辑判断来确定如何进行路由跳转。也就是说：需要在js代码中进行路由跳转。此时可以使用编程式路由。

1. 使用this.$router.push方法可以实现路由跳转，方法的第一个参数可为string类型的路径，或者可以通过对象将相应参数传入。
2. 通过this.$router.go(n)方法可以实现路由的前进后退，n表示跳转的个数，正数表示前进，负数表示后退。
3. 如果只想实现前进后退可以使用this.$router.forward()（前进一页），以及this.$router.back()（后退一页）。

```
<div id="app">
  <p>
    <button @click="toHome">首页</button>
    <button @click="toNews">新闻</button>
    <button @click="toLogin">登陆</button>
    <button @click="doForward1">前进</button>
    <button @click="doForward2">前进</button>
```

```
    <button @click="doBack1">后退</button>
    <button @click="doBack2">后退</button>
  </p>
  <router-view></router-view>
</div>

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

<script type="text/javascript">
  const Home = {
    template: '<div>首页</div>'
  }
  const News = {
    template: '<div>新闻：{{$route.query.name}}</div>'
  }
  const Login = {
    template: '<div>登陆</div>'
  }

  const routes = [{
    path: '/',
    component: Home
  }, {
    path: '/home',
    component: Home
  }, {
    path: '/news',
    component: News
  }, {
    path: '/login',
    component: Login
  }]

  const router = new VueRouter({
    routes
  })

  var vm = new Vue({
    el: '#app',
    data: {},
    router,
    methods:{
      toHome(){
        //无参数时，push方法中直接写路由地址
        this.$router.push('/home');
      },
      toNews(){
        //有参数时，push方法中写一个json对象
```

```
        this.$router.push({path:'/news',query:{name:'zhangsan'}});
      },
      toLogin(){
        this.$router.push('/login');
      },
      doForward1(){
        this.$router.forward();
      },
      doForward2(){
        this.$router.go(1);
      },
      doBack1(){
        this.$router.back();
      },
      doBack2(){
        this.$router.go(-1);
      }
    }
  })
</script>
```

五、通过watch实现路由监听

通过watch属性设置监听$route变化，达到监听路由跳转的目的。

在上面代码中添加watch监听：

```
watch: {
  // 监听路由跳转。
  $route(newRoute, oldRoute) {
    console.log('watch', newRoute, oldRoute)
  }
}
```

六、导航守卫

路由跳转前做一些验证，比如登录验证，是网站中的普遍需求。
对此，vue-route 提供了实现导航守卫（navigation-guards）的功能。

你可以使用 router.beforeEach 注册一个全局前置守卫：

```
const router = new VueRouter({ ... })

router.beforeEach((to, from, next) => {
    // ...
})
```

每个守卫方法接收三个参数：

1. to：即将要进入的目标路由对象（去哪里），可以使用 to.path 获取即将要进入路由地址。
2. from：当前导航正要离开的路由对象（从哪来），可以使用 from.path 获取正要离开的路由地址。
3. next：一个函数，表示继续执行下一个路由。（如果没有next，将不会进入到下一个路由）

下面例子中实现了如下功能：

1. 列举需要判断登录状态的 “路由集合”，当跳转至集合中的路由时，如果“未登录状态”，则跳转到登录页面
2. 当直接进入登录页面LoginPage时，如果“已登录状态”，则跳转到首页HomePage；

```
<div id="box">
  <p>
    <router-link to="/home">home</router-link>
    <router-link to="/news">news</router-link>
    <router-link to="/music">music</router-link>
    <router-link to="/login">login</router-link>
  </p>
  <router-view></router-view>
</div>

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

<script type="text/javascript">
  const Home = {
    template: '<div>首页</div>'
  }
  const News = {
    template: '<div>新闻</div>'
  }
  const Music = {
    template: '<div>音乐</div>'
  }
  const Login = {
    template: '<div>登录</div>'
  }

  const routes = [{
    path: '/',
    component: Home
```

```
  }, {
    path: '/home',
    component: Home
  }, {
    path: '/news',
    component: News
  }, {
    path: '/music',
    component: Music
  }, {
    path: '/login',
    component: Login
  }]

  const router = new VueRouter({
    routes //  (缩写) 相当于 routes: routes
  })

  var vm = new Vue({
    el: '#box',
    data: {},
    router
  })

  // 添加全局路由守卫
  router.beforeEach((to, from, next) => {
    //创建守卫规则集合(这里表示'/news'与'/music'路径是需要权限验证的)
    const nextRoute = ['/news', '/music'];
    // 使用isLogin来模拟是否登录
    let isLogin = false;
    // 判断to.path(要跳转的路径)是否是需要权限验证的
    if (nextRoute.indexOf(to.path) >= 0) {
      if (!isLogin) {
        router.push({
          path: '/login'
        })
        location.reload(); //必须要有
      }
    }
    // 已登录状态；当路由到login时，跳转至home
    if (to.path === '/login') {
      if (isLogin) {
        router.push({
          path: '/home'
        });
        location.reload();
      }
    }
    next(); //必须要有
```

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