第 3 章 Vue 基础语法

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，因此对于大多数开发者来讲上手是非常容易的。在 Vue 的分层设计思想中，模板属于视图层，有了模板功能，开发者可以方便地将项目组件化，也可以方便地封装定制化的通用组件。在编写组件时，模板的作用是让开发者将重心放在页面布局渲染上，而不需要关心数据逻辑。同样，在 Vue 组件内部编写数据逻辑代码时，也无须关心视图的渲染。

3.1 ES6 新特性

当前版本的 ES6 是在 2015 年 6 月正式发布的，所以又称为 ECMAScript 2015。它的目标是使 JavaScript 语言可以用来编写复杂的程序，成为企业级脚本语言。

3.1.1 ES6 简介

1997 年，为了统一各种不同的 Script 脚本语言，ECMA（欧洲计算机制造商协会）以 JavaScript 为基础制定了 ECMAScript 标准规范，也就是 ECMAScript 1.0。JavaScript 和 JScript 都是 ECMAScript 的标准实现者，随后各大浏览器厂商纷纷实现了 ECMAScript 标准。

由此可知，ECMAScript 是浏览器脚本语言的规范，而我们熟知的 JavaScript 则是规范的具体实现。

ECMAScript 发布的历史版本见表 3-1。

表 3-1 ECMAScript 的历史版本

版　本	年　份	描　述
第 1 版	1997 年	制定了语言的基本语法
第 2 版	1998 年	较小改动
第 3 版	1999 年	引入了正则、异常处理、格式化输出等。IE 开始支持
第 4 版	2007 年	过于激进，对 ES3 做了彻底升级，未发布
第 5 版	2009 年	引入了严格模式、JSON，扩展对象、数组、原型、字符串、日期方法
第 6 版	2015 年	模块化、面向对象语法、Promise、箭头函数、let、const、数组解构赋值等

ES6 提供了许多新特性，但并不是所有的浏览器都能够完美支持。好在目前各大浏览器自身也加快速度兼容 ES6 的新特性，其中对 ES6 新特性支持最友好的是 Chrome 和 Firefox 浏览器。

为什么要学习 ES6？

（1）ES6 的版本变动内容最多，具有里程碑意义。

（2）ES6 加入许多新的语法特性，编程实现更简单、高效。

（3）ES6 是前端发展趋势，是就业的必备技能。

3.1.2　let 和 const

1. 基本用法

ES6 新增了 let 和 const 命令，用于声明变量。

在 ES5 之前，JavaScript 定义变量常用的关键字是 var，var 有一个问题，就是定义的变量有时会莫名其妙地成为全局变量，示例代码如下：

```
{
  var b = 1;
  let a = 10;
}
console.log(b);              //1
console.log(a);              //a is not defined
```

在上面的示例代码中，分别用 var 和 let 声明了变量，然后在代码块之外调用这两个变量，结果 let 声明的变量报错，而 var 声明的变量返回了正确值。这表明，let 声明的变量只在它所在的代码块有效，而 var 在全局变量中都有效。

在 for 循环的计数器中，var 和 let 的区别更加明显，示例代码如下：

```
for (var i = 0; i < 5; i++) {
      console.log("for 循环(内)部:" + i)
  }
  console.log("for 循环(外)部:" + i);
```

输出的结果如图 3-1 所示。

图 3-1　计数器中使用 var 声明变量

在上面的代码中,变量 i 是在 for 循环的内部声明的,而循环体的外部也能访问,其值为5,证明 i 是全局变量。

let 所声明的变量,只在 let 命令所在的代码块内有效。把上例中的 var 改为 let,代码如下:

```
for (let i = 0; i < 5; i++) {
      console.log("for 循环(内)部:" + i)
  }
  console.log("for 循环(外)部:" + i);
```

输出的结果如图 3-2 所示。

图 3-2　计数器中使用 let 声明变量

在上面的代码中,变量 i 只在 for 循环体内有效,如果在循环体外引用就会报错。

const 声明的变量是常量,不能被修改。

常量指的是无法在程序正常运行的过程中修改值。一方面无法通过重新赋值进行修改,另一方面也无法进行重新声明。在 JavaScript 中,常量通过关键字 const 声明,示例代码如下:

```
const number = 66;
console.log(number)
number++;
console.log(number)
```

输出的结果如图 3-3 所示。

图 3-3　const 声明的常量不能被修改

const 实际上保证的并不是常量的值不得改动,而是常量指向的那个内存地址所保存的数据不得改动。

2. 不允许重复声明

let 不允许在相同作用域内重复声明同一个变量，代码如下：

```
//报错
function f() {
  let a = 10;
  var a = 1;
}

//报错
function f() {
  let name = 'admin';
  let name = 'beixi';
}
```

3. ES5 的块级作用域

ES5 只有全局作用域和函数作用域，没有块级作用域，这带来很多不合理的场景。如下面的实例，内层变量可能会覆盖外层变量，代码如下：

```
var tmp = new Date();
function f() {
  console.log(tmp);
  if (false) {
      var tmp = 'hello world';
  }
}

f();                //输出: undefined
```

上面代码的原意是，if 代码块的外部使用外层的 tmp 变量，内部使用内层的 tmp 变量，但是，函数 f 执行后，输出结果为 undefined，原因在于变量提升，从而导致内层的 tmp 变量覆盖了外层的 tmp 变量。

let 命令实际上为 JavaScript 新增了块级作用域，代码如下：

```
function f() {
  let age = 18;
  if (true) {
    let age = 66;
  }
  console.log(age);
}
f();                        //输出: 18
```

在上面的示例代码中声明了两个 age 变量，运行后输出18。这表示外层代码块不受内层代码块的影响。

3.1.3 变量的解构赋值

ES6 允许按照一定模式,从数组和对象中提取值,对变量进行赋值,这被称为解构。本节主要讲解数组的解构赋值、对象的解构赋值、字符串的解构赋值、数值和布尔值的解构赋值及函数参数的解构赋值等。

1. 数组的解构赋值

在 ES6 之前想要为变量赋值,只能指定其值,代码如下:

```
let a = 1;
let b = 2
```

而在 ES6 中可以写成这样,代码如下:

```
let [a,b] = [1,2]
//a = 1, b = 2
```

如此,只要等号两边的模式一样,就可以成功地解构赋值。如果两边的模式相同,但是右边少值,就会解构不成功,导致左边的变量值为 undefined,代码如下:

```
let [a,b,c] = [1,2]
a = 1, b = 2, c = undefined
```

注意:*只有左右两边的格式一定要对等,数量可以不对等。*

还有一种情况,等号左边为数组,但是等号右边为其他值,这将会报错,代码如下:

```
let [a] = 1;
let [a] = false;
let [a] = NaN;
let [a] = undefined;
let [a] = null;
let [a] = {};

//以上都会报错
```

在数组解构赋值中允许指定默认值。当一个位置没有值时,也就是当模式相同,但是右边没有值时可以指定默认值,代码如下:

```
let [a,b=4,c] = [1, , 3];
console.log(a);                    //输出: 1
console.log(b);                    //输出: 4
console.log(c);                    //输出: 3
```

2. 对象的解构赋值

变量的解构赋值和数组的解构赋值不太一样,数组的元素必须和赋值的元素的位置一致,这样才能正确地赋值,而对象的解构赋值则是等号两边的变量和属性同名即可取到正确

的值，否则值为 undefined，代码如下：

```
let {a,b} = {a:'23',b:'3'}
let {a,b} = {b:'3',a:'23'}

//上面两个的值都是 a: 23 b: 3

let {a,b} = {a:'3',c:'d'}
//a: 3 b: undefined
```

如果等号左边的变量名不能和等号右边的对象的属性名一致，则必须写成如下格式：

```
let {a:b} = {a:'ss'}                    //b:ss
//a 是属性名,b 才是实际赋值的变量名
```

对象的解构也可以指定默认值，代码如下：

```
let {x = 3} = {}
//x: 3
let {x,y = 5} = {x : 1}
//x: 1, y: 5
let {x: y = 5} = {}
//y = 5
let {x: y = 5} = {x : 4}
//y = 4
let {x: y = 'hhhh'} = {}
//y = 'hhhh'
```

3. 字符串的解构赋值

如果赋值的对象是字符串，则字符串将被分割为数组的格式，一一对应赋值，代码如下：

```
let [a, b, c, d, e, f] = 'string';
console.log(a);                         //输出：s
console.log(b);                         //输出：t
console.log(c);                         //输出：r
console.log(d);                         //输出：i
console.log(e);                         //输出：n
console.log(f);                         //输出：g
```

因为字符串具有 length 这个属性，因此还可以对该属性进行解构赋值，代码如下：

```
let { length: len } = 'string';
console.log(len);                  //输出：6
```

4. 数值和布尔的解构赋值

数值和布尔值也能进行解构赋值，此时它们都会被转换为对象，代码如下：

```
let { toString: tos1 } = 456;
let { toString: tos2 } = false;
```

```
console.log(tos1 === Number.prototype.toString);              //输出: true
console.log(tos2 === Boolean.prototype.toString);             //输出: true
```

5. 函数的解构赋值

函数的参数也可以进行解构赋值,这是一个解构赋值运用比较多的场景,其实就是对之前所讲的数组、对象、布尔值、数值解构赋值的一种实际使用,代码如下:

```
function add([a, b]) {
   return a + b;
}
console.log(add([2, 3]));                  //输出: 5
```

6. 用途

变量的解构赋值使用场景很多。

1) 场景 1: 交换变量的值

示例代码如下:

```
let x = 1;
let y = 2;
[x,y] = [y,x];
 //x = 2,y = 1;
```

上面代码交换了 x 和 y 的值,这样的写法不仅简洁,而且易读,语义非常清晰。

2) 场景 2: 从函数返回多个值

函数只能返回一个值,如果要返回多个值,则只能将它们放在数组或对象里返回。有了解构赋值,取出这些值就非常方便了,代码如下:

```
//返回一个数组
function example() {
  return [1, 2, 3];
}
let [a, b, c] = example();

//返回一个对象
function example() {
  return {
    foo: 1,
    bar: 2
  };
}
let { foo, bar } = example();
```

3) 场景 3: 函数参数的定义

解构赋值可以方便地将一组参数与变量名对应起来,代码如下:

```
//参数是一组有次序的值
function f([x, y, z]) { ... }
f([1, 2, 3]);

//参数是一组无次序的值
function f({x, y, z}) { ... }
f({z: 3, y: 2, x: 1});
```

4）场景 4：提取 JSON 数据

解构赋值对提取 JSON 对象中的数据尤其有用，代码如下：

```
let jsonData = {
  id: 42,
  status: "OK",
  data: [867, 5309]
};

let { id, status, data: number } = jsonData;

console.log(id, status, number);
//42, "OK", [867, 5309]
```

上面的代码可以快速提取 JSON 数据的值。

3.1.4　模板字符串

模板字符串替换＋操作符来拼接字符串，并且支持换行。

先来看一段传统字符串写法，代码如下：

```
let name = "Tom";
let occupation = "doctor";
//传统字符串拼接
let str = "He is " + name + ",he is a " + occupation;
```

而用 ES6 简洁的字符串拼接，可以写成这样：

```
let name = "Tom";
let occupation = "doctor";
//模板字符串拼接
let str = `He is ${name},he is a ${occupation}`;
```

对比两段拼接的代码，模板字符串使我们不再需要反复使用双引号（或者单引号）了，而是改用反引号标识符（`），插入变量时也不需要再使用加号（＋）了，而是把变量放入 ${} 即可。

模板字符串是增强版的字符串，提示以下两点。

1. 可以定义多行字符串

传统的多行字符串，写法如下：

```
let str = "write once ," +
          "run anywhere";
```

模板字符串的写法如下：

```
let str = `write once ,
          run anywhere`;
```

直接换行即可，但是需要注意的是模板字符串中所有的空格和缩进都会被保留在输出中。也就是如果控制台输出字符串 str，代码上换了行，则控制台输出的内容也会换行。

2. ${}中可以放任意的 JavaScript 表达式

花括号内部可以放入任意的 JavaScript 表达式，可以进行运算，以及引用对象属性或函数调用。

(1) ${}中可以是运算表达式，代码如下：

```
var a = 1;
var b = 2;
var str = `the result is ${a + b}`;
//进行加法运算,结果:the result is 3
```

(2) ${}中可以是对象的属性，代码如下：

```
var obj = {"a":1,"b":2};
var str = `the result is ${obj.a + obj.b}`;
//对象 obj 的属性
//结果:the result is 3.
```

(3) ${}中可以是函数的调用，代码如下：

```
function fn() {
      return 3;
  }
var str = `the result is ${ fn() }`;
//函数 fn 的调用,结果:the result is 3
```

3.1.5 标签模板

标签模板其实不是模板，而是函数调用的一种特殊形式。“标签”指的就是函数，是一个专门处理模板字符串的函数，紧跟在后面的模板字符串就是它的参数，示例代码如下：

```
var name = "张三";
var height = 1.8;

fn`他叫 ${name},身高 ${height}米.`;
//标签 + 模板字符串

//定义一个函数,作为标签
```

```
function fn(arr,v1,v2){
    console.log(arr);
    //结果:[ "他叫",",身高","米." ]
    console.log(v1);
    //结果:张三
    console.log(v2);
    //结果:1.8
}
```

以上代码有两处大家要注意,首先是 fn 函数,这是一个自定义的函数,它有 3 个参数,分别是 arr、v1、v2。函数 fn()的调用方式跟以往的调用方式不太一样,以往使用括号()表示函数调用执行,这一次在函数名后面直接加上一个模板字符串,代码如下:

```
fn`他叫 ${name},身高 ${height}米.`;
```

这样就是标签模板,可以理解为标签函数+模板字符串,这是一种新的语法规范。

接下来看函数的 3 个参数,从代码的打印结果可以看到它们运行后对应的结果,arr 的值是一个数组:["他叫",",身高","米。"],而 v1 的值是变量 name 的值:“张三”,v2 的值是变量 height 的值:1.8。

观察结果可以发现:第 1 个参数 arr 是数组类型,它的内容是模板字符串中除了 ${}以外的其他字符,按顺序组成了数组的内容,所以 arr 的值是["他叫",",身高","米。"];第 2 个和第 3 个参数则是模板字符串中对应次序的变量 name 和 height 的值。

标签模板是 ES6 带来的一种新语法,掌握了标签模板的用法后,可以利用这一特性更好、更快捷地实现各种功能。

3.1.6 字符串新增方法

1. raw()转义

String.raw()是一个模板字符串的标签函数,往往用于模板字符串的处理方法,返回值是自动转义的字符串,代码如下:

```
//常用形式
let name = "Bob";
String.raw `Hi\n ${name}!`;
//"Hi\nBob!"

//正常函数形式
//`foo ${1 + 2}bar`
//等同于
String.raw({ raw: ['foo', 'bar'] }, 1 + 2)            //"foo3bar"
```

2. includes()、startsWidth()、endsWidth()包含

之前由 indexOf 方法判断一个字符串是否包含在另一个字符串中,ES6 又提供了 3 种新方法。

(1) includes(searchStr[, position])：返回布尔值，判断字符串中是否含有指定的子字符串，如果返回值为 true，则表示含有，如果返回值为 false，则表示未含有。第 2 个参数选填，表示开始搜索的位置。

(2) startsWith(searchStr[, position])：返回布尔值，判断指定的子字符串是否出现在目标字符串的开头位置，第 2 个参数选填，表示开始搜索的位置。

(3) endsWith(searchStr[, length])：返回布尔值，判断子字符串是否出现在目标字符串的尾部位置，第 2 个参数选填，表示针对前 n 个字符。

示例代码如下：

```
let s = 'Hello world!';

s.startsWith('Hello')              //true
s.endsWith('!')                    //true
s.includes('o')                    //true

s.startsWith('world', 6)           //true 从 index 为 n 开始
s.endsWith('Hello', 5)             //true 前 n 个字符
s.includes('Hello', 6)             //false 从 index 为 n 开始
```

上面代码表示，使用第 2 个参数 n 时，endsWith 的行为与其他两种方法有所不同。它针对前 n 个字符，而其他两种方法针对从第 n 个位置直到字符串结束。

3. repeat()重复

repeat()方法返回一个新字符串，表示将原字符串重复 n 次，代码如下：

```
'x'.repeat(3)                      //"xxx"
'hello'.repeat(2)                  //"hellohello"
'na'.repeat(0)                     //""
'na'.repeat(NaN)                   //"" 参数 NaN 等同于 0
```

注意：如果参数是小数，则会被取整。如果 repeat 的参数是负数或者 Infinity，则会报错。

4. padStart()、padEnd()补全

如果某个字符串不够指定长度，则会在头部或尾部补全。padStart()用于头部补全，而 padEnd()用于尾部补全。

(1) str.padStart(targetLength [, padString])用另一个字符串填充当前字符串(如果需要，则重复填充)，以便产生的字符串达到给定的长度。从当前字符串的首部(左侧)开始填充。

(2) str.padEnd(targetLength [, padString])用一个字符串填充当前字符串(如果需要，则重复填充)，返回填充后达到指定长度的字符串。从当前字符串的末尾(右侧)开始填充。

示例代码如下：

```
'x'.padStart(5, 'ab')              //'ababx'
'x'.padStart(4, 'ab')              //'abax'

'x'.padEnd(5, 'ab')                //'xabab'
'x'.padEnd(4, 'ab')                //'xaba'
```

在上面的代码中，padStart()和 padEnd()一共接受两个参数，第 1 个参数是字符串补全生效的最大长度，第 2 个参数是用来补全的字符串。

如果省略第 2 个参数，则默认使用空格补全长度，代码如下：

```
'x'.padStart(4)                    //' x'
'x'.padEnd(4)                      //'x '
```

5. trim()、trimRight()(trimEnd())、trimLeft()(trimStart())去空

trimLeft()是 trimStart()的别名，trimRight()是 trimEnd()的别名。

(1) trim()方法会从一个字符串的两端删除空白字符。在这个上下文中空白字符是所有的空白字符(Space、Tab、no-break Space 等)及所有行终止符字符(如 LF、CR)。

(2) trimRight()方法从一个字符串的右端移除空白字符，又称 trimEnd。

(3) trimLeft()方法从一个字符串的左端移除空白字符，又称 trimStart。

示例代码如下：

```
const h = ' hello world! ';

h.trim();                          //'hello world!'

h.trimLeft();                      //'hello world! '
h.trimStart();                     //'hello world! '

h.trimEnd();                       //' hello world!'
h.trimRight();                     //' hello world!'
```

除了空格键，对字符串头部(或尾部)的 Tab 键、换行符等不可见的空白符号也有效。

6. substring()和 slice()截取

substring()方法返回一个字符串在开始索引到结束索引之间的一个子集，或从开始索引直到字符串的末尾的一个子集。返回新的字符串，不改变原来的字符串。

(1) str.substring(indexStart[, indexEnd])：indexStart 需要截取的第 1 个字符的索引，该字符作为返回的字符串的首字母。[,indexEnd]可选，一个 0 到字符串长度之间的整数，以该数字为索引的字符不包含在截取的字符串内(左闭右开)。

示例代码如下：

```
const h = 'helloworld!';
//如果省略 indexEnd,则 substring 提取字符,一直到字符串末尾
h.substring(5)                          //'world!'
//如果任一参数小于 0 或为 NaN,则被当作 0
```

```
h.substring(0,5)              //'hello'
h.substring(-1,5)             //'hello'
//如果 indexStart 等于 indexEnd,则 substring 返回一个空字符串
h.substring(5,5)              //''
//如果任一参数大于 stringName.length,则被当作 stringName.length
h.substring(0,100)            //'helloworld!'
//如果 indexStart 大于 indexEnd,则 substring 的执行效果就像两个参数调换了一样
h.substring(-5,5);            //'hello';
h.substring(5,-5);            //'hello';
```

slice()方法用于提取某个字符串的一部分,并返回一个新的字符串,并且不会改动原字符串。

(2) str.slice(beginIndex[, endIndex])参数和 substring 一样,只不过有差异。

和上面的示例作对比,代码如下:

```
const h = 'helloworld!';
h.slice(5);                    //'world!'同 substring
h.slice(0,5);                  //"hello" 同 substring

h.slice(-1,5);                 //'' 不同 substring

h.slice(5,5);                  //'' 同 substring

h.slice(0,100);                //"helloworld!" 同 substring

h.slice(-5,5);                 //'' 不同 substring

h.slice(5,-5);                 //'w' 不同 substring

h.slice(-6);                   //"world!"
```

可见 slice 方式的索引是可以倒数的,强烈推荐使用 slice 方式截取字符串,更好理解,不易出错。

3.1.7 函数的扩展

1. 参数的默认值

在开发中,经常需要给函数的参数指定默认值,ES6 之前给函数设置默认参数是这样实现的,示例代码如下:

```
function person(n,a){
    var name = n || 'admin';
    var age = a || 18;
    console.log(name,age);
}
person('beixi');               //输出:beixi 18
person(false,0);               //输出:admin 18
```

或运算符左侧为 true，右侧为 false。传入实参，直接返回左侧的值，否则返回右侧的值。如在 person 函数内，如果参数 n 没有传参，则变量 name 得到的值就是 admin，如果传参了，变量 name 的值就为参数 n 的值。

但是，前提是参数对应的布尔值不能为 false(数字 0，空字符串等转换成布尔值也是 false)，这就使这种方式存在一定的不足和缺陷。如在上面的示例中 person(false,0)，在调用函数时传入实参 false，结果都为或运算符的右侧值。

在 ES6 中允许直接在函数的参数后面设置默认值，示例代码如下：

```
function person(name = 'admin',age = 18){
  console.log(name,age);
}
person();                          //输出:admin 18
person(false,20);                  //输出:false 20
```

这样就不必再担心函数的实际传参与预期不相符了。

如果函数中有多个参数，有需要指定默认值的，有不需要指定默认值，则必须把设定默认值的参数放在后面，示例代码如下：

```
//错误写法
function person(age = 18,name){
    console.log(name,age);
}

//正确写法
function person(name,age = 18){
    console.log(name,age);
}
```

另外还有一点需要注意，由于参数变量是默认声明的，所以在函数体内不能再使用 let 或者 const 再次声明，示例代码如下：

```
function person(age = 18) {
    let age = 20;              //错误
}
person();
```

上面的示例代码，在函数被调用后就会报错，提示 age 已经被声明过了，如图 3-4 所示。

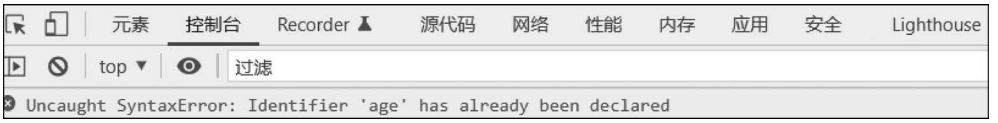

图 3-4　age 及被声明的错误信息

2. 与解构赋值默认值结合使用

参数默认值可以与解构赋值的默认值结合起来使用，代码如下：

```
function fn({a,b = 2}){
  console.log(a,b);
}
fn({})                    //输出:undefined 2
fn({a:1})                 //输出:1 2
fn({a:1,b:3})             //输出:1 3
```

3. rest 参数

rest 参数是一个新的概念,中文意思是剩下的部分。

ES6 之前,如果要获取函数中传入的多余参数,则可使用 arguments,代码如下:

```
function fn(a) {
    console.log(a);
    for(var i = 0; i < arguments.length; i ++){
        console.log(arguments[i]);
    }
}
fn(1, 2, 3);              //输出:1 1 2 3
```

从输出结果看,arguments 对象其实包含的是所有参数,并不是多余参数。因为在调用函数时,给 a 变量赋值 1,多余的参数是 2 和 3,所有输出结果应为 1 2 3 才符合预期。

ES6 中,新增了 rest 参数,可以用来获取多余的参数。语法为"…变量名",代码如下:

```
function fn(a,...values) {
  values.forEach(function (item) {
    console.log(item);
 });
}
fn(1, 2, 3);              //输出: 2 3
```

示例中,values 变量就是一个数组,包含传入的两个多余的参数 2、3。

这里需要注意一点,rest 参数只能放在最后面,否则会报错,代码如下:

```
//错误写法
function sum(a, ...values, b){

}
//正确的写法
function sum(a, b, ...values){

    }
```

4. 箭头函数

ES6 介绍了一种全新的定义函数的方式,也就是用箭头符号(⇒)来定义函数,故得名箭头函数。这里定义一个最简单的函数,代码如下:

```
//传统写法
function sayHi() {
    alert('hi');
}
//箭头函数写法
var sayHi = () => {
    alert('hi');
}
```

如果需要传参，则可把参数写在圆括号里，代码如下：

```
//传统写法
function sayHi(a,b) {
  console.log(a + b);
}
//箭头函数写法
var sayHi = (a,b) => {
  console.log(a + b);
}
sayHi(10,20);                    //输出:30
```

箭头函数的最大作用就是简化函数的实现，大大地减少了代码。如果只有一个参数，则可以不使用圆括号，如果只有一条语句，则花括号也可以省略，代码如下：

```
//箭头函数写法
var sayHi = a => console.log(a);
sayHi(10);                  //输出:10
```

由于花括号被解释为代码块，所以如果箭头函数返回的是一个对象，则必须在对象外面加上括号，否则会报错，示例代码如下：

```
//错误写法
var fn = () => {username: 'tom', age: 24};

//正确写法
var fn = () => ({username: 'tom', age: 24});
```

箭头函数有以下几个注意点：

(1) 箭头函数 this 为父作用域的 this，不是调用时的 this。

箭头函数的 this 永远指向其父作用域，任何方法都改变不了，代码如下：

```
  var name = 'admin';
 var teacher = {
    name: 'zhangsan',
    showName: function () {
        let showTest = () => alert(this.name);
        showTest();
```

```
    }
};
teacher.showName(); //弹出:zhangsan
```

箭头函数中的 this 其实是父级作用域中的 this。箭头函数引用了父级的变量词法作用域,也就是一个变量的作用在定义时就已经被定义好了,当在本作用域中找不到变量时就会一直向父作用域中查找,直到找到为止,代码如下:

```
var name = 'admin';
  var teacher = {
    name: 'zhangsan',
    showName: () => {
        let showTest = () => alert(this.name);
        showTest();
    }
};
teacher.showName();                    //弹出:admin
```

上例中,showName 也为箭头函数,其内部的 this 为全局 window,showTest 的 this 也就是 showName 函数的 this,也是 window,得到的 this.name 就为 admin。

(2) 箭头函数不能作为构造函数,不能使用 new,代码如下:

```
//构造函数如下
function Person(p){
    this.name = p.name;
}
//如果用箭头函数作为构造函数,则代码如下
var Person = (p) => {
    this.name = p.name;
}
```

由于 this 必须是对象实例,而箭头函数是没有实例的,所以此处的 this 指向别处,不能产生 Person 实例,自相矛盾。

(3) 箭头函数没有原型属性,代码如下:

```
var a = ()=>{
  return 1;
}

function b(){
  return 2;
}

console.log(a.prototype);                    //undefined
console.log(b.prototype);                    //{constructor: ƒ}
```

（4）多重箭头函数就是一个高阶函数，相当于内嵌函数，代码如下：

```
const add = x => y => y + x;
//相当于
function add(x){
  return function(y){
    return y + x;
  };
}
```

3.2　模板语法

由于所有的 Vue.js 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。在底层的实现上，Vue 将模板编译成虚拟 DOM 渲染函数。结合响应系统，Vue 能够智能地计算出最少需要重新渲染多少组件，并把 DOM 操作次数减到最少。

简单地说，Vue 模板语法采用的是前端渲染，前端渲染即是把数据填充到 HTML 标签中：数据（来自服务器）＋模板（HTML 标签）＝前端渲染（产物是静态 HTML 内容）。

3.2.1　插值

插值就是将数据插入 HTML 文档中，包含文本、HTML 元素、元素属性、JavaScript 表达式等。

1. 文本插值

文本插值中用得最多的就是用双花括号的形式，如例 3-1 所示。

【例 3-1】 文本插值方式

```
//第 3 章/文本插值.html
<!DOCTYPE html>
<html>
<head lang="en">
<meta charset="UTF-8">
<title></title>
  <!-- 引入 Vue 3.x 文件 -->
  <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <!-- 会与实例中 data 的属性绑定在一起，并且数据实现同步 -->
    <h1>{{message}}</h1>
</div></body>
<script>
    //创建一个程序实例
    const vm = Vue.createApp({
      //data 模型数据，值是一个对象
```

```
        data(){
            return{
                message: "I LOVE YOU"
            }
        }
    }).mount("#app");
</script>
</body>
</html>
```

在浏览器中的显示效果如图 3-5 所示。

图 3-5　文本插值

无论何时,绑定的数据对象上的 message 属性发生了改变,插值处的内容都会更新,如图 3-6 所示。在 Chrome 浏览器中按 F12 键打开控制台并切换到 Elements 选项,更改 message 的值,然后查看渲染的结果。

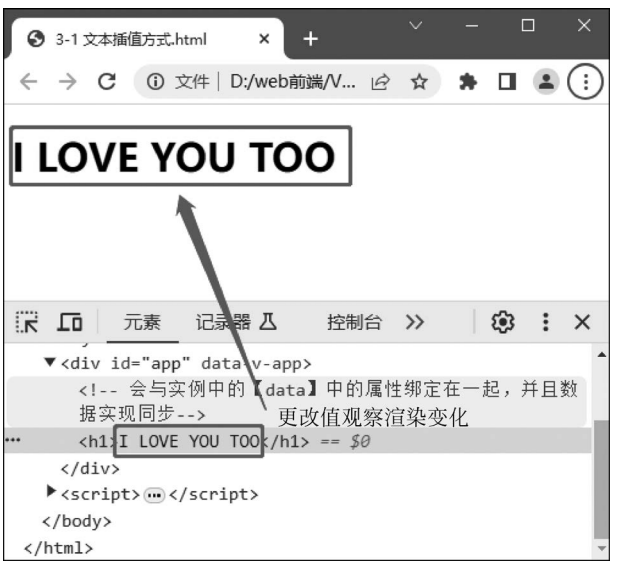

图 3-6　数据的双向性

在上面的代码中当 data 中的值更新之后我们不需要操作 html,页面中会自动更新数据。

也可以通过 v-once 指令让数据只绑定一次,当数据改变时,插值处的内容不会更新,如

例 3-2 所示。

【例 3-2】 使用 v-once 指令插值

```
//第 3 章/使用 v - once 指令插值.html
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF - 8">
    <title></title>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
    <div id="app">
        <!-- v - once 只编译一次。显示内容之后不再具有响应式功能。-->
        <h1 v - once>不可改变:{{message}}</h1>
        <h1>可改变:{{message}}</h1>
        <input type="text" v - model="message">
    </div>
    <script>
      const vm = Vue.createApp({
         data(){
            return{
              message: 'I LOVE YOU'
            }
         }
      }).mount("#app");
    </script>
</body>
</html>
```

在 Chrome 浏览器中运行程序,然后在输入框中输入“我爱你”,可以看到添加 v-once 指令表情内容不发生改变,效果如图 3-7 所示。

图 3-7　v-once 指令应用

2. html 插值

双花括号会将数据解释为普通文本,而非 HTML 代码。为了输出真正的 HTML,需要使用 v-html 指令,如例 3-3 所示。

【例 3-3】 使用 v-html 指令将数据解释成 HTML 代码

```
//第 3 章/v - html 指令.html
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF - 8">
    <title></title>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
<!-- 内容按普通 HTML 插入, 不会作为 Vue 模板进行编译,在网站上动态渲染任意 HTML 是非常危险
的,因为容易导致 XSS 攻击 -->
<h1 v - html="msg"></h1>
</div>
</body>
<script>
    const vm = Vue.createApp({
          data(){
             return{
                /* 可以使用 v - html 指令展示 HTML 代码 */
                msg: "<span style='color:blue'> BLUE </span>"
             }
          }
    }).mount("#app");
</script>
</html>
```

上面的代码将 msg 属性值以< span >标签插入< h1 >标签中。

3. 属性插值

在开发时,有时属性不是写死的,有可能是根据一些数据动态地决定的,例如图片标签(< img >)的 src 属性,可能从后端请求了一个包含图片地址的数组,需要将地址动态地绑定到 src 上面,这时就不能简单地将 src 写死。或如 a 标签的 href 属性。这时可以使用 v-bind:指令来解决这种问题(v-bind: 可以缩写成:),如例 3-4 所示。

【例 3-4】 使用 v-bind: 指令绑定属性

```
//第 3 章/使用 v - bind 指令.html
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF - 8">
    <title></title>
    <script src="https://unpkg.com/vue@next"></script>
</head>
```

```
<body>
    <div id="app">
        <img v-bind:src="imgUrl" alt=""/>
        <a :href="searchUrl">百度一下</a>
    </div>
    </body>
    <script>
       const vm = Vue.createApp({
          data(){
             return{
                imgUrl:'images/1.jpg',
                searchUrl:'http://www.baidu.com'
             }
          }
       }).mount("#app");
    </script>
</html>
```

从服务器请求来的数据，一般会在 data 中中转，中转后再把需要的变量绑定到对应的属性上面。

v-bind 除了在开发中用在有特殊意义的属性（src、href 等）外，也可以绑定其他一些属性，如 Class 与 Style 绑定，如例 3-5 和例 3-6 所示。

【例 3-5】 v-bind 动态绑定属性 class

```
//第 3 章/v-bind 动态绑定属性.html
<!DOCTYPE html>
<head>
    <meta charset="UTF-8">
    <title>v-bind 动态绑定属性 class</title>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
    <div id="app">
        <p:class="{fontCol:isName,setBack:!isAge}"
                class="weight">{{name}}</p>
        <i :class="addClass">{{name}}真好看!</i>
    </div>
    <script>
       const vm = Vue.createApp({
          data(){
            return{
            isName:true,
            isAge:false,
            name:"功守道"
            }
       },
      /*当 v-bind:class 的表达式过长或者逻辑复杂(一般当条件多于两个时),
      可以考虑采用计算属性,返回一个对象*/
      computed:{
```

```
            addClass:function(){
                return {
                    checked:this.isName&&!this.isAge
                }
            }
        }
        }).mount("#app");
    </script>
</body>
</html>
```

既然是一个对象,那么该对象内的属性可能不唯一,但总是每项为真时,对应的类名就会存在。通过 v-bind 更新的类名和元素本身存在的类名不冲突,可以优雅地共存,如图 3-8 所示。

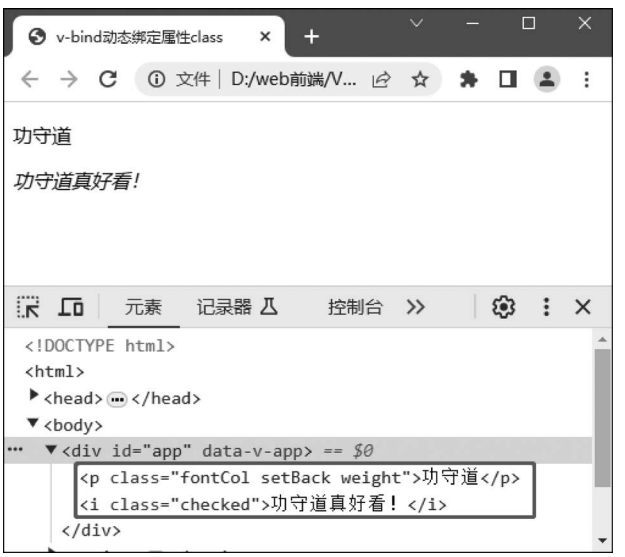

图 3-8 v-bind 动态绑定属性 class

【例 3-6】 v-bind 动态绑定属性 style

```
//第 3 章/v-bind 动态绑定属性 style.html
<!DOCTYPE html>
<head>
    <meta charset="UTF-8">
    <title>v-bind 动态绑定属性 style</title>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <div :style="{'color': color,'fontSize':fontSize + 'px'}">修饰文本</div>
</div>
<script>
```

```
    const vm = Vue.createApp({
        data(){
            return{
                color: 'red',
                fontSize: 24
            }
        }
    }).mount("#app");
</script>
</body>
</html>
```

在大多数情况下，由于标签中直接写一长串的样式不便于阅读和维护，所以一般写在dada 或 computed 里，代码如下：

```
<div id = "app">
    <div :style = "styles">修饰文本</div>
</div>
<script>
    const vm = Vue.createApp({
        data(){
            return{
                styles:{
                    color: 'red',
                    fontSize: 24 + 'px'
                }
            }
        }
    }).mount("#app");
</script>
```

当应用多个样式对象时，可以使用数组语法，如例 3-7 所示。

【例 3-7】 v-bind 绑定多个 style 属性

```
//第 3 章/v - bind 绑定多个 style 属性.html
<!DOCTYPE html>
<head>
    <meta charset = "UTF - 8">
    <title> v - bind 动态绑定属性 style </title>
    <script src = "https://unpkg.com/vue@next"></script>
</head>
<body>
<div id = "app">
    <div :style = "[styleA,styleB]">文本</div>
</div>
<script>
    const vm = Vue.createApp({
        data(){
            return{
```

```
                styleA:{
                    color: 'red',
                    fontSize: 24 + 'px'
                },
                styleB: {
                    width: 100 + 'px',
                    border: 1 + 'px ' + 'black ' + 'solid'
                }
            }
        }
    }).mount("#app");
</script>
</body>
</html>
```

4. 插值中使用 JavaScript 表达式

迄今为止,在我们的模板中,我们一直都只绑定简单的 property 键值,但实际上,对于所有的数据绑定,Vue.js 都提供了完全的 JavaScript 表达式支持,如例 3-8 所示。部分表达式格式的代码如下:

```
{{ number + 1 }}
{{ ok ? 'YES' : 'NO' }}
{{ message.split('').reverse().join('') }}
<div v-bind:id="'list-' + id"></div>
```

这些表达式会在所属 Vue 实例的数据作用域下作为 JavaScript 被解析。有个限制,也就是每个绑定都只能包含单个表达式,所以下面的表达式都不会生效,代码如下:

```
<!-- 这是语句,不是表达式 -->
{{ var a = 1 }}
<!-- 流控制也不会生效,请使用三元表达式 -->
{{ if (ok) { return message } }}
```

【例 3-8】 使用 JavaScript 表达式

```
//第 3 章/使用 JavaScript 表达式.html
<!DOCTYPE html>
<head>
    <meta charset="UTF-8">
    <title></title>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <p>{{ number + 1 }}</p>
    <hr>
    <p>{{msg + '~~~~~~'}}</p>
    <hr>
```

```
    <p>{{flag ? '条件为真' : '条件为假'}}</p>
    <hr>
</div>
<script>
    const vm = Vue.createApp({
        data(){
            return{
                msg:'Hello beixi!',
                flag:true,
                number:2
            }
        }
    }).mount("#app");
</script>
</body>
</html>
```

在浏览器中的显示效果如图 3-9 所示。

图 3-9　使用 JavaScript 表达式效果图

3.2.2　指令

指令是带有“v-”前缀的特殊属性，如前面使用的 v-bind、v-once 等指令。指令属性的值预期是单个 JavaScript 表达式(除了 v-for)。指令的职责就是当其表达式的值改变时相应地将某些行为应用到 DOM 上。

1. 参数

一些指令能够接收一个“参数”，在指令名称之后以冒号表示。如 v-bind 指令可以用于响应式地更新 HTML 特性，代码如下：

```
<a v-bind:href = "url">...</a>
```

在这里 href 是参数，告知 v-bind 指令将该元素的 href 属性与表达式 url 的值绑定。

v-on 指令用于监听 DOM 事件，代码如下：

```
<a v-on:click = "test">...</a>
```

其中,参数 click 是监听的事件。

2. 动态参数

从 Vue 2.6.0 开始,可以用方括号括起来的 JavaScript 表达式作为一个指令的参数,响应式使 Vue 更加灵活多变,其动态参数也有其含义,代码如下:

```
<a v-bind:[attributeName] = 'url'>...</a>
```

这里的 attributeName 会被作为一个 JavaScript 表达式进行动态求值,求得的值将作为最终的参数来使用。例如,如果你的 Vue 实例有一个 data 属性 attributeName,其值为'href',则这个绑定将等价于 v-bind:href。

同样地,可以使用动态参数作为一个动态的事件名绑定处理函数,格式代码如下:

```
<a v-on:[eventName] = 'doSomething'>...</a>
```

同样地,当 eventName 的值为'focus'时,v-on:[eventName]将等价于 v-on:focus。

【例 3-9】 动态参数

```
//第 3 章/动态参数.html
<!DOCTYPE html>
<html lang = "en">
<head>
    <meta charset = "UTF-8">
    <title>动态参数</title>
    <script src = "https://unpkg.com/vue@next"></script>
</head>
<body>
    <div id = "app">
    <p><a v-bind:[attr] = "url">百度链接</a></p>
    <p><button v-on:[things] = "doSomething">单击事件</button></p>
    </div>
    <script>
       const vm = Vue.createApp({
            data(){
               return{
                   attr: 'href',
                   things:'click',
                   url:'http://baidu.com'
               }
            },//在选项对象的 methods 属性中定义方法
            methods:{
               doSomething :function(){
                  alert("触发了单击事件")
               }
            }
        }).mount("#app");
    </script>
</body>
</html>
```

在浏览器中运行程序，在页面中单击“单击事件”按钮，结果如图 3-10 所示。

图 3-10　动态参数

当然动态参数的值也是有约束的，动态参数预期会求出一个字符串，异常情况下值为 null。这个特殊的 null 值可以被显式地用于移出绑定。任何其他非字符串类型的值都将会触发一个警告。

3. 修饰符

修饰符(Modifiers)是以半角句号“.”指明的特殊后缀，用于指出一个指令应该以特殊方式绑定。例如，.prevent 修饰符告诉 v-on 指令对于触发的事件调用 event.preventDefault()，代码如下：

```
<form v-on:submit.prevent="onSubmit">...</form>
```

在后面章节对 v-on 和 v-for 等功能的探索中，可以看到修饰符的其他例子。

3.3　实例及选项

在一个使用 Vue 框架的页面应用程序中，最终都会创建一个应用程序的实例对象并挂载到指定的 DOM 上。这个实例将提供应用程序上下文，应用程序实例装载的整个组件树将共享相同的上下文。

创建应用程序的实例后，可以调用实例的 mount()方法制定一个 DOM 元素，在该 DOM 元素上装载应用程序的根组件，这样这个 DOM 元素中的所有数据变化都会被 Vue 框架所监控，从而实现数据的双向绑定。

```
Vue.createApp().mount("#app");
```

3.3.1　数据选项

组件的 data 选项是一个函数。Vue 在创建新组件实例的过程中会自动调用此函数。data 选项通常返回一个对象，然后 Vue 会通过响应性系统将其包裹起来，并以 $data 的形式存储在组件实例中。

为了方便起见，作为语法糖，该对象的任何顶级属性也可以通过组件实例直接调用，示

例代码如下：

```
<div id = "app">

</div>
<script>
    const vm = Vue.createApp({
        data() {
           return { count: 4 }
        }
    }).mount('#app')

    console.log(vm.$data.count)                    //=> 4
    console.log(vm.count)                          //=> 4

    //修改 vm.count 的值也会更新 $data.count
    vm.count = 5
    console.log(vm.$data.count)                    //=> 5

    //反之亦然
    vm.$data.count = 6
console.log(vm.count)                              //=> 6
</script>
```

由于这些实例属性仅在实例首次创建时被自动添加，所以大家需要确保它们都在 data 函数返回的对象中。必要时，要对尚未提供所需值的属性使用 null、undefined 或其他占位的值。

在 Vue 实例被初始化后，再往 data 中手动添加新属性虽然是可以操作的，但该新属性不具备响应式功能，只是个傻瓜属性。

Vue 使用“$”前缀通过组件实例暴露自己的内置 API。Vue 还为内部属性保留“_”前缀。也就是说，给自定义的变量命名一定不能用这两个符号开头。

3.3.2 方法选项

Vue 通过 methods 选项为组件实例添加方法，我们在实现组件的方法时，可以放心地在其中使用 this 关键字，Vue 会自动将其绑定到当前组件实例本身。

1. 方法的调用方式

1）使用插值{{}}方式

【例 3-10】 使用插值方式

```
//第 3 章/使用插值方式.html
<!DOCTYPE html>
<html lang = "en">
<head>
    <meta charset = "UTF - 8">
    <title>Document</title>
    <script src = "https://unpkg.com/vue@next"></script>
```

```
</head>
<body>
    <div id="app">
        输入内容:<input type="text" v-model="message"><br>
        反转内容:{{ reversedMessage()}}
    </div>
    <script>
        const vm = Vue.createApp({
            data() {
               return { message: ''}
            },
            methods: {
                reversedMessage() {
                    //`this`指向该组件实例
                    return this.message.split('').reverse().join('')
                }
            }
        }).mount('#app')
    </script>
</body>
</html>
```

在浏览器中的显示效果如图 3-11 所示。

图 3-11　使用插值方式

2）使用事件调用

例如使用 v-on 指令来监听 DOM 事件，使用事件来调用方法，如例 3-11 所示。

【例 3-11】 事件调用方法

```
    //第 3 章/事件调用方法.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>事件调用方法</title>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
    <div id="app">
        <button v-on:click="test">点我</button>
    </div>
    <!-- 测试组件 -->
    <script>
```

```
        const vm = Vue.createApp({
            methods:{
              /*定义了一个 test 函数*/
                test: function () {
                    console.log(new Date().toLocaleTimeString());
                }
            }
        }).mount('#app')
    </script>
</body>
</html>
```

在浏览器中的显示效果如图 3-12 所示。

图 3-12　事件调用方法

2. 传递参数

传递参数和正常的 JavaScript 传递参数的方法一样，如例 3-12 所示。

【例 3-12】 传递参数

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>传递参数</title>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
    <div id="app">
        {{num}}<br>
        <button v-on:click="add(2)">增加</button>
    </div>
    <script>
```

```
        const vm = Vue.createApp({
          data(){
              return{
                  num:1
              }
          },
          methods:{
              add: function (a) {
                 this.num += a
              }
          }
      }).mount('#app')
    </script>
</body>
</html>
```

在浏览器中运行程序，单击 1 次“增加”按钮，可以发现 num 值自增 2，显示效果如图 3-13 所示。

图 3-13　传递参数

3.3.3　计算属性和侦听器

在大多数情况下可以将 Vue 组件中定义的属性数据直接渲染到 HTML 元素上，但是有些场景下，属性的数据并不适合直接渲染，需要处理后再进行渲染。在 Vue 中，通常使用计算属性或侦听器实现这种逻辑。

有时可能需要先在{{}}里进行一些计算，然后展示出来数据，如例 3-13 所示。

【例 3-13】　在页面中展示学生的成绩、总分和平均分

```
//第 3 章/学生成绩信息.html
<!DOCTYPE html>
<head>
    <title></title>
    <meta charset="utf-8"/>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <table border="1">
        <thead>
            <th>学科</th>
```

```
            <th>分数</th>
        </thead>
        <tbody>
            <tr>
                <td>数学</td>
                <td><input type="text" v-model="math"></td>
            </tr>
            <tr>
                <td>英语</td>
                <td><input type="text" v-model="english"></td>
            </tr>
            <tr>
                <td>语文</td>
                <td><input type="text" v-model="chinese"></td>
            </tr>
            <tr>
                <td>总分</td>
                <td>{{math + english + chinese}}</td>
            </tr>
            <tr>
                <td>平均分</td>
                <td>{{(math + english + chinese)/3}}</td>
            </tr>
        </tbody>
    </table>
</div>
    <script>
        const vm = Vue.createApp({
          data(){
              return{
                  math:66,
                  english :77,
                  chinese:88
              }
          }
        }).mount('#app')
    </script>
</body>
</html>
```

在浏览器中的运行效果如图 3-14 所示。

学科	分数
数学	66
英语	77
语文	88
总分	231
平均分	77

图 3-14　学生成绩、总分和平均分的展示

虽然通过{{}}运算可以满足我们的需求，但是代码结构不清晰，特别是当运算比较复杂时，不能复用功能代码。这时，大家不难想到使用 methods 来封装功能代码，但事实上，Vue 向我们提供了一个更好的解决方案，即计算属性。计算属性是 Vue 实例中的一个配置选项：computed，通常是计算相关的函数，返回最后计算出来的值。也就是可以把这些计算的过程写到一个计算属性中去，然后让它动态地计算，如例 3-14 所示。

【例 3-14】 使用计算属性展示学生的成绩、总分和平均分

```
//第 3 章/使用计算属性展示学生的成绩、总分和平均分.html
<!DOCTYPE html>
<head>
    <title></title>
    <meta charset="utf-8"/>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <table border="1">
        <thead>
            <th>学科</th>
            <th>分数</th>
        </thead>
        <tbody>
            <tr>
                <td>数学</td>
                <td><input type="text" v-model.number="math"></td>
            </tr>
            <tr>
                <td>英语</td>
                <td><input type="text" v-model.number="english"></td>
            </tr>
            <tr>
                <td>语文</td>
                <td><input type="text" v-model.number="chinese"></td>
            </tr>
            <tr>
                <td>总分</td>
                <td>{{sum}}</td>
            </tr>
            <tr>
                <td>平均分</td>
                <td>{{average}}</td>
            </tr>
        </tbody>
    </table>
</div>
    <script>
       const vm = Vue.createApp({
          data(){
             return{
                math:66,
                english:77,
                chinese:88
             }
          },
          computed:{
              <!-- 一个计算属性的 getter -->
```

```
                sum:function(){
                    return this.math + this.english + this.chinese;
                },
                average:function(){
                    return Math.round(this.sum/3);
                }
            }
        }).mount('#app')
    </script>
</body>
</html>
```

计算属性一般通过其他的数据计算出一个新数据,而且它有一个好处就是能把新的数据缓存下来,当其他的依赖数据没有发生改变时,它调用的是缓存的数据,这就极大地提高了我们程序的性能和数据提取的速度,而如果写在 methods 里,则数据不会被缓存下来,所以每次都会重新计算。这也是为什么这里没用 methods 的原因,如例 3-15 所示。

【例 3-15】 计算属性和方法的比较

```
//第 3 章/计算属性和方法的比较.html
<!DOCTYPE html>
<head>
    <title></title>
    <meta charset="utf-8"/>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <p>原始字符串: "{{ message }}"</p>
    <p>计算属性反向字符串: "{{ reversedMessage1 }}"</p>
    <p>methods 方法反向字符串: "{{ reversedMessage2() }}"</p>
</div>
<script>
    const vm = Vue.createApp({
        data() {
            return{
                message: 'beixi'
            }
        },
        computed:{
            reversedMessage1: function () {
                return this.message.split('').reverse().join('')
            }
        },
        methods: {
            reversedMessage2: function () {
                return this.message.split('').reverse().join('')
            }
        }
    }).mount("#app");
```

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

在浏览器中的显示效果如图 3-15 所示。

图 3-15　计算属性和方法的比较

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时才会重新求值。这就意味着只要 message 没有发生改变，多次访问 reversedMessage1 计算属性会立即返回之前的计算结果，而不必再次执行函数。相比而言，只要发生重新渲染，methods 总会调用执行该函数。

假设有一个性能开销比较大的计算属性 A，它需要遍历一个极大的数组和做大量的计算。可能有其他的计算属性依赖于 A。如果没有缓存，则将不可避免地多次执行 A 的 getter，极大地降低了数据的提取速度，对于用户体验感不强。如果不希望有缓存，则用 methods 替代。

例 3-14 和例 3-15 只提供了 getter 读取值的方法，实际上除了 getter，还可以设置计算属性的 setter，如例 3-16 所示。

【例 3-16】　计算属性的读取和设置

```
//第 3 章/计算属性的读取和设置.html
<!DOCTYPE html>
<head>
    <title></title>
    <meta charset = "utf - 8"/>
    <script src = "https://unpkg.com/vue@next"></script>
</head>
<body>
    <div id = "app"></div>
<script>
    const vm = Vue.createApp({
       data(){
          return{
             a: 1
          }
       },
       computed: {
          //改函数只能读取数据，vm.aDouble 即可读取数据
```

```
            aDouble: function () {
                return this.a * 2
            },
            //读取和设置数据
            aPlus: {
                get: function () {
                    return this.a + 1
                },
                set: function (v) {
                    this.a = v - 1
                }
            }
        }
    }).mount("#app");
    console.log(vm.aPlus);                    //2
    vm.aPlus = 3;
    console.log(vm.a);                        //2
    console.log(vm.aDouble);                  //4
</script>
</body>
</html>
```

属性侦听是 Vue 非常强大的功能之一。使用属性侦听器可以方便地监听某个属性的变化,以完成复杂的业务逻辑。大家都用过搜索引擎,以百度搜索引擎为例,当向搜索框中写入关键字后,网页上会自动关联一些推荐词供用户选择,如图 3-16 所示,这些场景非常适合使用监听器实现。

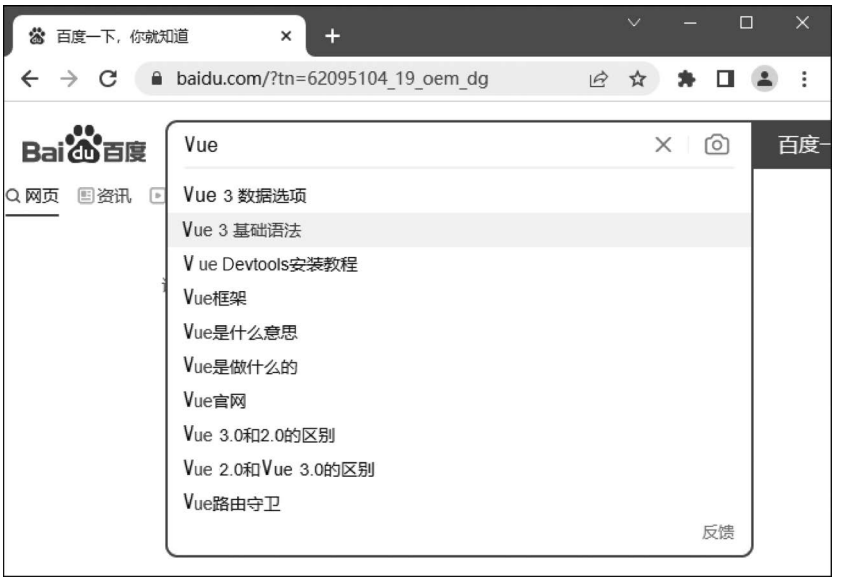

图 3-16　搜索引擎

在定义 Vue 组件时，可以通过 watch 选项来定义属性侦听器，如例 3-17 所示。

【例 3-17】 侦听器应用

```
//第 3 章/侦听器应用.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Document</title>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
    <div id="app">
        <input v-model="searchText">
    </div>
    <script>
        const vm = Vue.createApp({
            data(){
                return{
                   searchText:'',
                }
          },
          watch:{
              searchText(oldValue,newValue){
                  if(newValue.length>10){
                      alert("文本太长了")
                  }
              }
          }
       }).mount("#app");
    </script>
</body>
</html>
```

运行以上程序，在文本框中输入一些字符，当字符超过 10 个时，就会弹出警告框，如图 3-17 所示。

图 3-17　侦听器应用

从一些特性上看，侦听器和计算属性有类似的应用场景，使用计算属性的 set 方法也可以实现与上面的示例代码类似的功能。

3.3.4 表单数据的双向绑定

1. 基础用法

可以用 v-model 指令在表单< input >、< textarea >及< select >元素上创建双向数据绑定,如图 3-18 所示。它会根据控件类型自动选取正确的方法来更新元素。尽管有些神奇,但 v-model 本质上不过是语法糖。它负责监听用户的输入事件以更新数据,并对一些极端场景进行一些特殊处理。

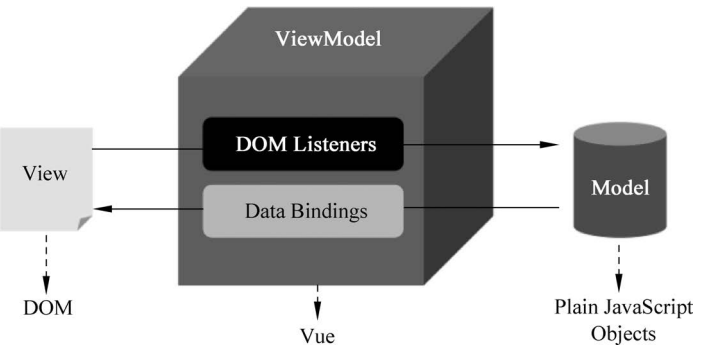

图 3-18 Vue 双向数据绑定

注意:v-model 会忽略所有表单元素的 value、checked、selected attribute 的初始值而总是将 Vue 实例的数据作为数据来源。应该通过 JavaScript 在组件的 data 选项中声明初始值。

1) 文本输入框

input 元素中使用 v-model 实现双向数据绑定,代码如下:

```
<div id = "app" class = "demo">
    <input v - model = "message" placeholder = "请输入信息...">
    <p > Message is: {{ message }}</p >
</div >
<script >
    const vm = Vue.createApp({
       data(){
          return{
             message: ''
          }
       }
    }).mount("#app");
</script >
```

2) 多行文本输入框

textarea 元素中使用 v-model 实现双向数据绑定,代码如下:

```
<div id = "app" class = "demo">
    <span > Multiline message is:</span >
```

```
    <p style="white-space: pre">{{ message }}</p><br>
    <textarea v-model="message" placeholder="请输入..."></textarea>
</div>
<script>
    const vm = Vue.createApp({
       data(){
          return{
             message: ''
          }
       }
    }).mount("#app");
</script>
```

注意：在文本区域插值(<textarea></textarea>)并不会生效，应用 v-model 来代替。

3）复选框

单个复选框绑定的是布尔值，如果选中，则值为 true，如果未选中，则值为 false，代码如下：

```
<div id="app" class="demo">
    <input type="checkbox" id="checkbox" v-model="checked">
    <label for="checkbox">{{ checked }}</label>
</div>
<script>
    const vm = Vue.createApp({
       data(){
          return{
             checked: false
          }
       }
    }).mount("#app");
</script>
```

多个复选框，绑定到同一个数组，代码如下：

```
<div id="app" class="demo">
    <input type="checkbox" id="jack" value="Jack" v-model="checkedNames">
    <label for="jack">Jack</label>
    <input type="checkbox" id="john" value="John" v-model="checkedNames">
    <label for="john">John</label>
    <input type="checkbox" id="mike" value="Mike" v-model="checkedNames">
    <label for="mike">Mike</label>
    <br>
    <span>Checked names: {{ checkedNames }}</span>
</div>
<script>
    const vm = Vue.createApp({
       data(){
          return{
             checkedNames: []
```

```
            }
        }
    }).mount("#app");
</script>
```

4）单选按钮

单选按钮的双向数据绑定，代码如下：

```
<div id = "app">
    <input type = "radio" value = "Runoob" v-model = "picked">
    <label for = "runoob">Runoob</label>
    <br>
    <input type = "radio" id = "google" value = "Google" v-model = "picked">
    <label for = "google">Google</label>
    <br>
    <span>选中值为：{{ picked }}</span>
</div>
<script>
    const vm = Vue.createApp({
        data(){
            return{
                picked : 'Runoob'
            }
        }
    }).mount("#app");
</script>
```

5）选择列表

下拉列表的双向数据绑定。

单选时，代码如下：

```
<div id = "app">
    <select v-model = "selected">
        <option disabled value = "">请选择</option>
        <option>A</option>
        <option>B</option>
        <option>C</option>
    </select>
    <span>Selected: {{ selected }}</span>
</div>
<script>
    const vm = Vue.createApp({
        data(){
            return{
                selected: ''
            }
        }
    }).mount("#app");
</script>
```

注意：如果 v-model 表达式的初始值未能匹配任何选项，则< select >元素将被渲染为“未选中”状态。在 iOS 中，这会使用户无法选择第 1 个选项。因为在这样的情况下，iOS 不会触发 change 事件，因此，更推荐像上面那样提供一个值为空的禁用选项。

多选列表(绑定到一个数组)，代码如下：

```
<div id = "app">
    <select v - model = "selected" multiple style = "width: 50px">
        <option > A </option >
        <option > B </option >
        <option > C </option >
    </select >
    <br >
    <span > Selected: {{ selected }}</span >
</div >
<script >
    const vm = Vue.createApp({
       data(){
          return{
             selected: [ ]
          }
       }
    }).mount("#app");
</script >
```

动态选项，用 v-for 渲染，代码如下：

```
<div id = "app" class = "demo">
    <select v - model = "selected">
        <option v - for = "option in options" v - bind:value = "option.value">
                  {{ option.text }}
        </option >
    </select >
    <span > Selected: {{ selected }}</span >
</div >
<script >
    const vm = Vue.createApp({
       data(){
          return{
             selected: 'A',
             options: [
                { text: 'One', value: 'A' },
                { text: 'Two', value: 'B' },
                { text: 'Three', value: 'C' }
             ]
          }
       }
    }).mount("#app");
</script >
```

2. 绑定 value

对于单选按钮,勾选框及选择列表选项,v-model 绑定的 value 通常是静态字符串,代码如下:

```
<div id = "app">
    <!-- 当选中时,picked 为字符串 "a" -->
    <input type = "radio" v - model = "picked" value = "a">
    <!-- 当选中时,toggle 为 true 或 false -->
    <input type = "checkbox" v - model = "toggle">
    <!-- 当选中时,selected 为字符串 "abc" -->
    <select v - model = "selected">
        <option value = "abc"> ABC </option>
    </select>
</div>
<script>
    const vm = Vue.createApp({
       data(){
          return{
             picked:'',
             toggle:'',
             selected:''
          }
       }
    }).mount("#app");
</script>
```

有时想将 value 绑定到 Vue 实例的一个动态属性上,这时可以用 v-bind 实现,并且这个属性的值可以不是字符串,代码如下:

```
<div id = "app">
    <input type = "checkbox" v - model = "toggle"
        v - bind:true - value = "a"    v - bind:false - value = "b">
        {{toggle}}
</div>
<script>
    const vm = Vue.createApp({
        data(){
          return{
              toggle:'',
              a:'a',
              b:'b'
          }
        }
    }).mount("#app");
</script>
```

当选中时输出 a;当没有选中时输出 b。

单选按钮,代码如下:

```
<input type = "radio" v - model = "pick" v - bind:value = "a">
```

当选中时 vm.pick ＝＝＝ vm.a。

单选列表设置，代码如下：

```
<select v-model="selected">
    <!-- 内联对象字面量 -->
    <option v-bind:value="{ number: 123 }">123</option>
</select>
```

当选中时在页面控制台输入 typeof vm.selected，输出'object'；输入 vm.selected.number，输出 123。

3. 修饰符

对于 v-model 指令，还有 3 个常用的修饰符 lazy、number 和 trim。

1）lazy

在默认情况下，v-model 默认为同步数据，可以使用 lazy 修饰符，此种情况会转换为在 change 事件中进行同步，也就是在失去焦点或者按 Enter 键时更新，代码如下：

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

2）number

如果想自动将用户的输入值转换为数值类型，则可以给 v-model 添加 number 修饰符，代码如下：

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

这通常很有用，因为即使在 type＝"number" 时，HTML 输入元素的值也总会返回字符串。如果这个值无法被 parseFloat() 解析，则会返回原始的值。

3）trim

如果要自动过滤用户输入的首尾空白字符，则可以给 v-model 添加 trim 修饰符，代码如下：

```
<input v-model.trim="msg">
```

3.3.5　生命周期

Vue 的生命周期用于描述组件从创建到销毁的全过程。在 Vue 中，组件生命周期的节点会被定义为一系列方法，这些方法称为生命周期钩子函数。Vue 3 和 Vue 2 的生命周期钩子非常像，我们仍然可以在相同的场景下使用相同的钩子函数。

Vue 3 在设计时对先前的版本进行了向下兼容，如果你的项目还在使用选项式 API 进行构建，则不需要修改生命周期相关的代码。如果使用组合式 API 进行项目构建，则生命周期钩子函数会略微不一样。Vue 的生命周期如图 3-19 所示。

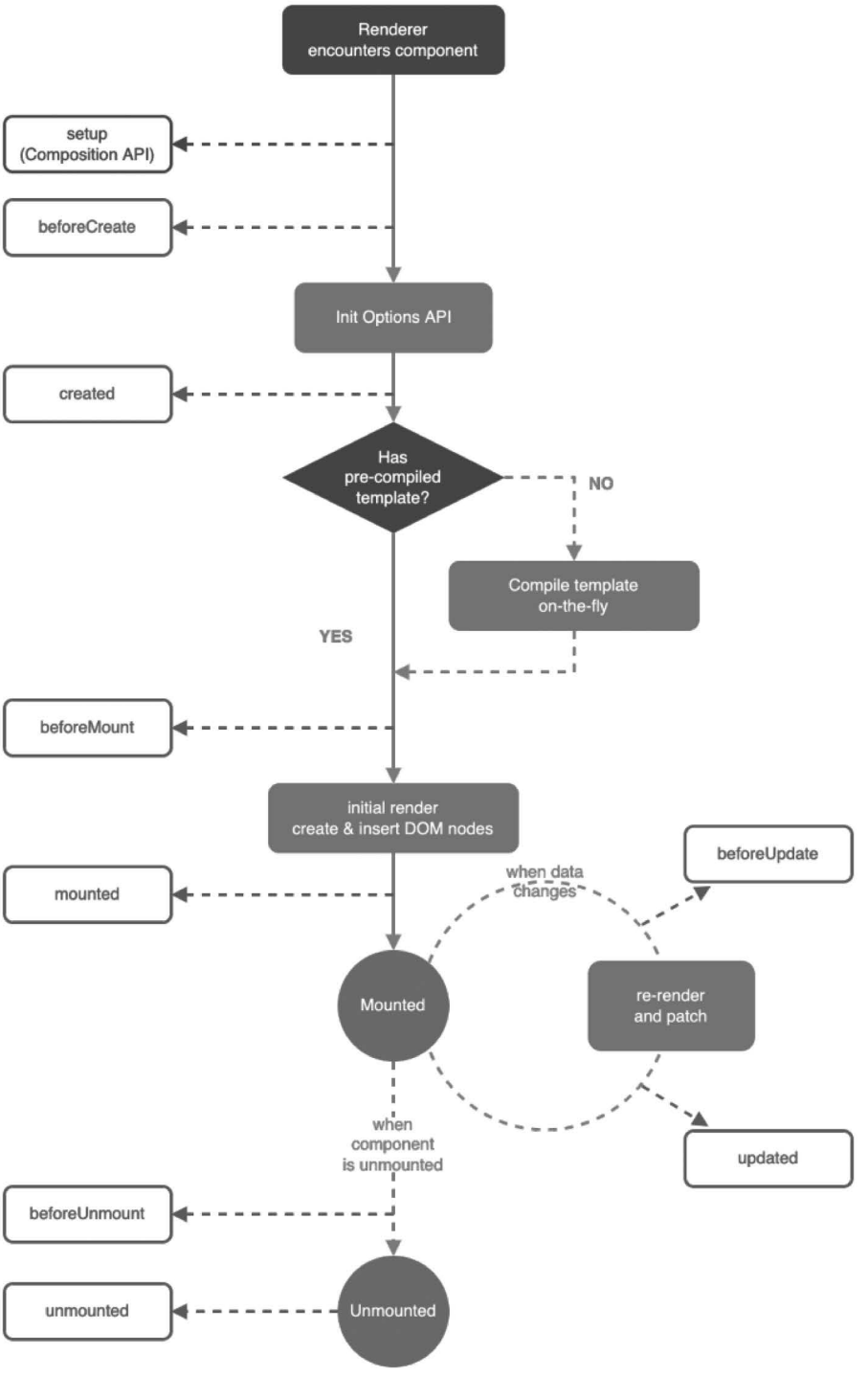

图 3-19　Vue 生命周期

一个 Vue 组件的生命周期分为 4 个阶段，即组件创建阶段、组件挂载阶段、数据更新阶段和组件销毁阶段，对应 8 个钩子函数。

（1）创建阶段：beforeCreate 和 created。

（2）挂载阶段：beforeMount 和 mounted。

（3）更新阶段：beforeUpdate 和 updated。

（4）销毁阶段：beforeUnmount 和 unmounted。

Vue 2 中的生命周期和 Vue 3 的生命周期对比见表 3-2。

表 3-2　生命周期中的钩子函数

Vue 2 生命周期	Vue 3 生命周期	功 能 描 述
beforeCreate	setup	在实例初始化之后，数据观测（Data Observer）和 event/watcher 事件配置之前被调用
created	setup	在实例创建完成后被立即调用。在这一步，实例已完成以下配置：数据观测（Data Observer），属性和方法的运算，watch/event 事件回调，然而，挂载阶段还没开始，$el 属性目前还不可见
beforeMount	onBeforeMount	在挂载开始之前被调用：相关的 render 函数首次被调用
mounted	onMounted	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。如果 root 实例挂载了一个文档内的元素，则当 mounted 被调用时 vm. $el 也在文档内
beforeUpdate	onBeforeUpdate	数据更新时调用，发生在虚拟 DOM 打补丁之前。这里适合在更新之前访问现有的 DOM，例如手动移除已添加的事件监听器
updated	onUpdated	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
beforeDestroy	onBeforeUnmount	在实例销毁之前调用。在这一步，实例仍然完全可用
destroyed	onUnmounted	在实例销毁之后调用。调用后，所有的事件监听器都会被移除，所有的子实例也都会被销毁

钩子函数中最常用的是 created、mounted、updated 和 unmounted。

在代码中使用生命周期函数有两种方式：选项式和组合式。

1. 选项式

Vue 3 中的选项式生命周期钩子基本上与 Vue 2 保持一致，它们都是定义在 Vue 实例的对象参数中的函数，它们在 Vue 实例的生命周期的不同阶段被调用。简单来讲，生命周期钩子就是 Vue.js 特别提供的一些函数，这是在我们的实例挂载、更新、卸载等过程中被调用的函数。如例 3-18 所示。

【例 3-18】 选项式生命周期

```
//第 3 章/选项式生命周期.html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>选项式</title>
```

```
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
</div>
<script>
    const vm = Vue.createApp({
        //组件实例化之前
        //可以访问 props 的数据,但不能访问组件的实例 this 中的数据源和函数等
        //不能访问组件中的视图 DOM 元素
        beforeCreate() {
            console.log('beforeCreate 组件实例化之前')
        },
        //组件实例化之后
        //可以访问组件中的数据、函数、自定义的属性等
        //不能访问组件中的视图 DOM 元素
        created() {
          console.log('created 组件实例化之后')
        },
        //组件视图渲染之前
        //可以访问组件中的数据、函数、自定义的属性等
        //不能访问组件中的视图 DOM 元素
        beforeMount() {
            console.log('beforeMount 组件视图渲染之前')
        },
        //组件视图渲染之后
        //可以访问组件中的数据、函数、自定义的属性等
        //不能访问组件中的视图 DOM 元素
        mounted() {
            console.log('mounted 组件视图渲染之后')
        },
        //数据源发生改变,视图重新渲染前
        //可以访问组件中的数据、函数、自定义的属性等
        //可以访问重新渲染的 DOM 元素之前的状态
        beforeUpdate() {
            console.log('beforeUpdate 数据源发生改变,视图重新渲染前')
        },
        //数据源发生改变,视图重新渲染后
        //可以访问组件中的数据、函数、自定义的属性等
        //可以访问重新渲染的 DOM 元素之后的状态
        updated() {
            console.log('updated 数据源发生改变,视图重新渲染后')
        },
        //组件在卸载之前
        //可以访问组件中的数据、函数、自定义的属性等
        //可以访问组件视图的 DOM 元素
        beforeUnmount() {
            console.log('beforeUnmount 组件在卸载之前')
```

```
        },
        //组件已卸载
        //可以访问组件中的数据、函数、自定义的属性等
        //不可以访问组件视图的 DOM 元素
        unmounted(){
            console.log('unmounted 组件已卸载')
        }
    }).mount("#app");
</script>
</body>
</html>
```

在浏览器中的显示效果如图 3-20 所示。

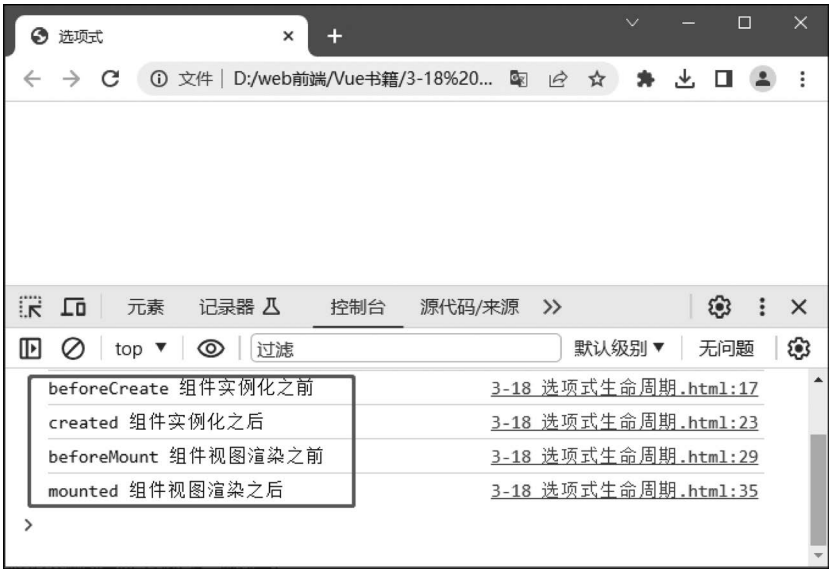

图 3-20　选项式生命周期

从图 3-20 可以看到，在本次页面渲染过程中只执行了 4 个组件的生命周期方法，这是由于我们使用的是 Vue 根组件，所以在页面渲染的过程中只执行了组件的创建和挂载过程，并没有执行卸载的过程。

2. 组合式

Vue 3 的组合式 API 提供了一套新的生命周期钩子，与 Vue 2 中的选项式生命周期钩子有着对应关系。

但是需要注意的是，Vue 3 的组合式 API 中的生命周期钩子函数是基于函数调用的方式，而不再作为组件选项的一部分。这意味着在使用组合式 API 时，不再像 Vue 2 中那样直接定义 beforeCreate、created 等钩子函数，而是通过 setup 函数返回一个对象，对象中包含相应的生命周期钩子函数。这些内容将在后面的章节中讲解。

3.4 模板渲染

获取后台数据之后会按照一定的规则加载到前端写好的模板中,显示在浏览器中,这个过程称为渲染。

3.4.1 条件渲染

1. v-if、v-else-if 和 v-else

v-if、v-else-if、v-else 这 3 个指令根据表达式的真假有条件地渲染元素。Vue 的条件指令可以根据表达式的值在 DOM 中渲染或销毁元素/组件,如例 3-19 所示。

【例 3-19】 v-if、v-else-if 、v-else 应用

```
//第 3 章/条件指令.html
<!DOCTYPE html>
<head>
    <title></title>
    <meta charset="utf-8"/>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <!-- if、else 指令 -->
    <p v-if="status == 1">当 status 为 1 时,显示该行</p>
    <p v-else-if="status == 2">当 status 为 2 时,显示该行</p>
    <p v-else-if="status == 3">当 status 为 3 时,显示该行</p>
    <p v-else>否则显示该行</p>
</div>
<!-- script 脚本 -->
<script>
    const vm = Vue.createApp({
       data(){
          return{
             status: 2
          }
       }
    }).mount("#app");
</script>
</body>
</html>
```

需要注意多个 v-if、v-else-if 和 v-else 之间必须紧挨着,例如下面代码的写法是错误的。

```
<p v-if="status == 1">当 status 为 1 时,显示该行</p>
<span></span>
<p v-else-if="status == 2">当 status 为 2 时,显示该行</p>
<p v-else-if="status == 3">当 status 为 3 时,显示该行</p>
<p v-else>否则显示该行</p>
```

以上代码会报错，因为多个判断语句被 span 标签隔开了，浏览器显示的内容如图 3-21 所示。

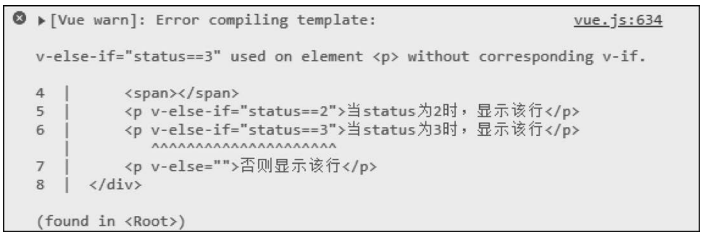

图 3-21　错误信息

2. v-show

v-show 指令的基本用法与 v-if 类似，也是根据条件的真假来决定元素的渲染情况，代码如下：

```
<div id = "app">
    <!-- if、else 指令 -->
    <p v-show = "status == 1">当 status 为 1 时，显示该行</p>
    <p v-show = "status == 2">当 status 为 2 时，显示该行</p>
</div>
<script>
    const vm = Vue.createApp({
        data(){
          return{
              status: 2
          }
        }
    }).mount("#app");
</script>
```

3. v-if 和 v-show 的区别

（1）控制显示的方法不同：该方法和 v-if 的区别在于，v-show 实际上是通过修改 DOM 元素的 display 属性实现节点的显示和隐藏的，而 v-if 则是通过添加/删除 DOM 节点实现的。

（2）编译条件：v-if 是惰性的，如果初始条件为假，则什么也不做，不会去渲染该元素；只有在条件第 1 次变为真时才开始局部编译；v-show 是不管初始条件是什么都被编译，然后被缓存，而且 DOM 元素保留，只是简单地基于 CSS 进行切换；即 v-if 中当为 true 时才会被加载渲染，当为 false 时不加载。v-show 不管是 true 还是 false 都会加载渲染。

（3）性能消耗：v-if 有更高的切换消耗；v-show 有更高的初始渲染消耗。

（4）使用场景：因此，如果需要非常频繁地切换，则使用 v-show 较好；如果在运行时条件很少改变，当只需显示一次或隐藏时，则使用 v-if 较好。

4. Key

Vue 会尽可能高效地渲染元素，通常会复用已有元素而不是从头开始渲染，因此可能造成下面这样的问题，如例 3-20 所示。

【例 3-20】 Vue 高效地渲染元素

```
//第 3 章/Vue 高效地渲染元素.html
<!DOCTYPE html>
<head>
    <title></title>
    <meta charset="utf-8"/>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<div id="app">
    <p v-if="ok">
        <label>Username</label>
        <input placeholder="Enter your username">
    </p>
    <p v-else>
        <label>Email</label>
        <input placeholder="Enter your email address">
    </p>
    <button @click="ok=!ok">切换</button>
</div>
<script>
     const vm=Vue.createApp({
        data(){
           return{
              ok: true,
           }
        }
    }).mount("#app");
</script>
</body>
</html>
```

当在页面中输入信息后单击“切换”按钮时,部分浏览器会出现文本框里的内容没有清空的现象。

Vue 提供了一种声明方式,只需添加一个具有唯一值的 key 属性便可以解决此问题,代码如下:

```
<div id="app">
    <p v-if="ok">
        <label>Username</label>
        <input placeholder="Enter your username" key="username-input">
    </p>
    <p v-else>
        <label>Email</label>
        <input placeholder="Enter your email address" key="email-input">
    </p>
    <button @click="ok=!ok">切换</button>
</div>
```

3.4.2　循环渲染

v-for 指令可以对数组、对象进行循环，以获取其中的每个值。

1. v-for 指令遍历数组

v-for 指令根据一组数组的选项列表进行渲染。

可以用 v-for 指令基于一个数组来渲染一个列表。v-for 指令需要使用 item in items 形式的特殊语法，其中 items 是源数据数组，而 item 则是被迭代的数组元素的别名（为当前遍历的元素提供别名，可以任意起名），如例 3-21 所示。

【例 3-21】 对数组选项进行列表渲染

```
//第 3 章/对数组选项进行列表渲染.html
<!DOCTYPE html>
<head>
    <title></title>
    <meta charset="utf-8"/>
    <script src="https://unpkg.com/vue@next"></script>
</head>
<body>
<ul id="app">
    <li v-for="item in items">
        {{ item.name }}
    </li>
</ul>
<script>
    const vm = Vue.createApp({
       data(){
          return{
             items: [
                { name: 'beixi' },
                { name: 'jzj' }
             ]
          }
       }
    }).mount("#app");
</script>
</body>
</html>
```

定义一个数组类型的数据 items，用 v-for 指令对其循环渲染，在浏览器中的显示效果如图 3-22 所示。

v-for 还支持一个可选的第 2 个参数作为当前项的索引，索引从 0 开始，代码如下：

```
<ul id="app">
    <li v-for="(item,index) in items">
        {{index}} -- {{ item.name }}
    </li>
</ul>
```

分隔符 in 前的语句使用括号,第 2 项就是 items 当前项的索引,在浏览器中的显示效果如图 3-23 所示。

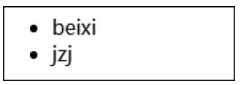

图 3-22　列表循环结果

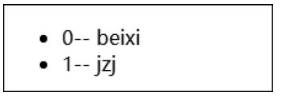

图 3-23　含有 index 选项的列表渲染结果

注意:可以用 of 代替 in 作为分隔符。

2. v-for 指令遍历对象

遍历对象的语法和遍历数组的语法是一样的:

```
value in object
```

其中,object 是被迭代的对象,value 是被迭代的对象属性的别名。

【例 3-22】 v-for 来遍历对象

```
//第 3 章/v - for 来遍历对象.html
<!DOCTYPE html>
<head>
    <title></title>
    <meta charset = "utf - 8"/>
    <script src = "https://unpkg.com/vue@next"></script>
</head>
<body>
<ul id = "app">
    <li v - for = "value in object">
          {{ value }}
    </li>
</ul>
<script>
    const vm = Vue.createApp({
       data(){
          return{
             object: {
                name: 'beixi',
                gender: '男',
                age: 30
             }
          }
       }
    }).mount("#app");
</script>
</body>
</html>
```

渲染后的结果如图 3-24 所示。

遍历对象属性时,有两个可选参数,分别是键名和索引,代码如下:

```
<ul id="app">
    <li v-for="(value, key, index) in object">
        {{ index }}-- {{ key }}: {{ value }}
    </li>
</ul>
```

渲染后的结果如图 3-25 所示。

图 3-24　遍历对象结果

图 3-25　遍历对象的渲染结果

3. v-for 指令遍历整数

v-for 指令还可以遍历整数，示例代码如下：

```
<ul id="app">
    <li v-for="n in 10">
        {{n}}
    </li>
</ul>
<script>
    const vm = Vue.createApp({
    }).mount("#app");
</script>
```

3.4.3　template 标签用法

类似于 v-if，也可以利用带有 v-for 的<template>标签来循环渲染一段包含多个元素的内容，代码如下：

```
<ul id="app">
    <template v-for="item in items">
          {{ item.name }}
          {{ item.age }}
    </template>
</ul>
<script>
    const vm = Vue.createApp({
       data(){
          return{
             items: [
                { name: 'beixi' },
                { age: 18 }
             ]
          }
```

```
        }
    }).mount("#app");
</script>
```

3.5 事件绑定

Vue 提供了 v-on 指令(通常使用@符号代替)来监听 DOM 事件,在事件绑定上,类似原生 JavaScript 的 onclick 事件写法,也是在 HTML 上进行监听的。

3.5.1 基本用法

Vue 中的事件绑定是使用 v-on:事件名=函数名来完成的,这里函数名是定义在 Vue 实例中的 methods 对象中的,Vue 实例可以直接访问其中的方法。

语法规则:

```
v-on:事件名.修饰符 = 方法名() | 方法名 | 简单的 JavaScript 表达式
简写: @事件名.修饰符 = 方法名() | 方法名 | 简单的 JavaScript 表达式
```

常用的交互事件见表 3-3。

表 3-3 Vue 中常用事件列表

事 件	描 述
click	在元素上按下并释放任意鼠标按键
dblclick	在元素上双击鼠标按钮
mousedown	在元素上按下任意鼠标按钮
mouseenter	指针移到有事件监听的元素内
mouseleave	指针移出元素范围外(不冒泡)
mousemove	指针在元素内移动时持续触发
mouseover	指针移到有事件监听的元素或者它的子元素内
mouseout	指针移出元素,或者移到它的子元素上
mouseup	在元素上释放任意鼠标按键
focus	元素获得焦点(不会冒泡)
blur	元素失去焦点(不会冒泡)
reset	单击“重置”按钮时
submit	单击提交按钮
keydown	按下任意按键
keyup	释放任意按键

1. 直接使用

直接在标签中书写 JS 方法,代码如下:

```
<div id = "app">
    单击次数{{count}}
<button @click = "count++">单击 + 1 </button>
</div>
<script>
    const vm = Vue.createApp({
        data(){
          return{
              count:0
          }
        }
    }).mount("#app");
</script>
```

注意：@click 等同于 v-on:click，这是一个语法糖。

2. 在 methods 设置方法

通过 v-on 绑定实例选项属性 methods 中的方法作为事件的处理器，如例 3-23 所示。

【例 3-23】 v-on 绑定事件

```
//第 3 章/v - on 绑定事件.html
<!DOCTYPE html>
<head>
    <title></title>
    <meta charset = "utf - 8"/>
    <script src = "https://unpkg.com/vue@next"></script>
</head>
<body>
<div id = "app">
    <button @click = "say">单击</button>
</div>
<script>
     const vm = Vue.createApp({
        data(){
           return{
              msg:'Say Hello'
           }
        },
        methods:{
           say: function () {
              alert(this.msg)
           }
        }
    }).mount("#app");
</script>
</body>
</html>
```

单击 button 按钮，即可触发 say 函数，在浏览器中的显示效果如图 3-26 所示。

图 3-26　单击事件

(1) 方法传参,直接在方法内传入参数,代码如下:

```
<div id="app">
    <button @click="say('Hello beixi')">单击</button>
</div>
<script>
     const vm = Vue.createApp({
       data(){
          return{
             msg:'Say Hello'
          }
       },
       methods:{
          say: function (val) {
             alert(val)
          }
       }
    }).mount("#app");
</script>
```

(2) 传入事件对象,代码如下:

```
<div id="app">
    <button data-aid='123' @click="eventFn( $event)">事件对象</button>
</div>
<script>
     const vm = Vue.createApp({
       methods:{
          eventFn:function(e){
             console.log(e);
             //e.srcElement dom 节点
             e.srcElement.style.background = 'red';
             console.log(e.srcElement.dataset.aid);          /*获取自定义属性的值*/
          }
       }
    }).mount("#app");
</script>
```

当单击时间对象按钮时背景颜色变为红色。

3.5.2　修饰符

Vue 中提供了多个修饰符，方便我们处理一些 DOM 事件的细节，让我们不再需要花费大量的时间去处理这些烦恼的事情，而将更多的精力用于程序的逻辑处理，Vue 常用的修饰符如下。

（1）stop 用于阻止事件继续传播，即阻止它的捕获和冒泡过程，代码如下：

```
@click.stop = 'handle()'                    //只要在事件后面加.stop 就可以阻止事件冒泡
```

如例 3-24 单击“内部单击”按钮，阻止了冒泡过程，即只执行 inner 这种方法，如果不加 .stop，则先执行 inner 方法，后执行 outer 方法，即通过了冒泡这个过程。

【例 3-24】 stop 修饰符应用

```
  //第 3 章/stop 修饰符应用.html
  <!DOCTYPE html>
  <head>
      <title></title>
      <meta charset = "utf - 8"/>
      <script src = "https://unpkg.com/vue@next"></script>
  </head>
  <body>
  <div id = "app">
      <div style = "background - color: aqua;width: 100px;height: 100px"
            v - on:click = "outer"> 外部单击
      <div style = "background - color: red;width: 50px;height: 50px"
            v - on:click.stop = "inner">内部单击</div>
      </div>
  </div>
  <script>
      const vm = Vue.createApp({
         methods:{
            outer: function () {
               console.log("外部单击")
            },
            inner: function () {
               console.log("内部单击")
            }
       }
    }).mount("#app");
</script>
</body>
</html>
```

（2）prevent 用于阻止默认事件，代码如下：

```
@click.prevent = 'handle()'          //只要在事件后面加.prevent 就可以阻止默认事件。
```

如下阻止了 a 标签的默认刷新：

```
<a href = "" v - on:click.prevent >单击</a >
```

(3) capture 用于添加事件监听器时使用事件捕获模式,即在捕获模式下触发,代码如下:

```
@click.capture = 'handle()'
```

如下实例在单击最里层的单击 6 时,outer 方法先执行,因为 outer 方法是在捕获模式执行的,先于冒泡事件。按下列顺序执行 outer→set→inner,因为后两个还是冒泡模式下触发的事件,代码如下:

```
<div v - on:click.capture = "outer">外部单击 5
    <div v - on:click = "inner">内部单击 5
        <div v - on:click = "set">单击 6 </div >
    </div >
</div >
```

(4) 当 self 是当前元素自身时触发处理函数时才会触发函数。

原理是根据 event.target 确定是否是当前元素本身,以此来决定是否触发事件/函数。

如果单击“内部单击”按钮,则冒泡不会执行 outer 方法,因为 event.target 指的是内部单击的 DOM 元素,而不是外部单击的,所以不会触发自己的单击事件,代码如下:

```
<div v - on:click.self = "outer">
    外部单击
    <div v - on:click = "inner">内部单击</div >
</div >
```

(5) once 表示只触发一次,代码如下:

```
<div id = "app">
    <div v - on:click.once = "once">单击 once </div >
</div >
<script >
    const vm = Vue.createApp({
       methods:{
          once: function () {
             console.log("单击 once")
          }
       }
    }).mount("#app");
</script >
```

(6) 键盘事件。

方式一,使用@keydown='show($event)',代码如下:

```
<div id = "app">
    <input type = "text" @keydown = 'show( $event)'   />
</div >
```

```
<script type="text/javascript">
    const vm = Vue.createApp({
        methods:{
            show: function (ev) {
                /*在函数中获取 ev.keyCode */
                console.log(ev.keyCode);
                if(ev.keyCode == 13){
                    alert('你按了 Enter 键!')
                }
            }
        }
    }).mount("#app");
</script>
```

方式二，代码如下：

```
<input type="text" @keyup.enter="show()">                    //Enter 键执行
<input type="text" @keydown.up='show()'>                     //上键执行
<input type="text" @keydown.down='show()'>                   //下键执行
<input type="text" @keydown.left='show()'>                   //左键执行
<input type="text" @keydown.right='show()'>                  //右键执行
```

3.6　图书管理系统

前面几个章节介绍了一些 Vue 的基础知识，结合以上知识可以实现图书管理系统，图书管理系统主要实现了数据的添加、删除、列表渲染等功能。最终实现的效果如图 3-27 所示。

图 3-27　图书管理系统效果图

这个示例是基于 Bootstrap 来快速搭建的，所以对 Bootstrap 不是很了解的读者，可以先自行到官网 http://www.bootcss.com/进行学习。开始实现此系统之前大家需下载 Bootstrap 文件，这里笔者采用的是 Bootstrap-3.3.7.css。

3.6.1　列表渲染

代码如下：

```
<!DOCTYPE html>
<head>
```

```
    <title></title>
    <meta charset="utf-8"/>
    <script src="https://unpkg.com/vue@next"></script>
    <!-- 引入 Bootstrap -->
    <link rel="stylesheet" href="./lib/Bootstrap-3.3.7.css">
</head>
<body>
<div id="app">
    <div class="panel panel-primary">
        <div class="panel-heading">
            <h2>图书管理系统</h2>
        </div>
        <div class="panel-body form-inline">
            <label for="">id:<input type="text" class="form-control"
                        v-model="id"></label>
            <label for="">图书名称:<input type="text" class="form-control"
                        v-model="name"></label>
            <input type="button" value="添加" class="btn btn-primary" @click="add">
        </div>
</div>
<table class="table table-bordered table-hover">
    <thead>
        <tr>
            <th>id</th>
            <th>图书名称</th>
            <th>添加时间</th>
            <th>操作</th>
        </tr>
    </thead>
    <tbody>
        <tr v-for="item in arr" :key="item.id">
            <td v-text="item.id"></td>
            <td v-text="item.name"></td>
            <td v-text="item.time"></td>
            <td><a href="" @click.prevent="del(item.id)">删除</a></td>
        </tr>
    </tbody>
  </table>
</div>
<script>
     const vm=Vue.createApp({
        data(){
            return{
               arr:[
                  {'id':1,'name':'三国演义','time':new Date()},
                  {'id':2,'name':'红楼梦','time':new Date()},
                  {'id':3,'name':'西游记','time':new Date()},
                  {'id':4,'name':'水浒传','time':new Date()}
               ], //创建一些初始数据与格式
```

```
                    id:'',
                    name:''
                }
        }
    }).mount("#app");
</script>
</body>
</html>
```

3.6.2　功能实现

添加功能的代码如下：

```
  add(){
        this.arr.push({'id':this.id,'name':this.name,'time':new Date()});
        this.id = this.name = '';
    }
```

删除功能的代码如下：

```
del(id){
    var index = this.arr.findIndex(item => {
     if(item.id == id) {
            return true;
         }
     })
 //findIndex方法用于查找索引以实现列表的删除功能
  this.arr.splice(index,1)
 }
```

完整的图书管理系统的代码如例 3-25 所示。

【例 3-25】 图书管理系统

```
<!DOCTYPE html>
<head>
    <title></title>
    <meta charset="utf-8"/>
    <script src="https://unpkg.com/vue@next"></script>
    <!-- 引入 Bootstrap -->
    <link rel="stylesheet" href="./lib/Bootstrap-3.3.7.css">
</head>
<body>
<div id="app">
    <div class="panel panel-primary">
        <div class="panel-heading">
            <h2>图书管理系统</h2>
        </div>
```

```
        <div class = "panel - body form - inline">
        <label for = ""> id:< input type = "text" class = "form - control"
        v - model = "id"></label >
        <label for = "">图书名称:< input type = "text" class = "form - control"
        v - model = "name"></label >
        < input type = "button" value = "添加" class = "btn btn - primary" @click = "add">
        </div >
    </div >
    < table class = "table table - bordered table - hover">
        < thead >
            < tr >
                < th > id </th >
                < th >图书名称</th >
                < th >添加时间</th >
                < th >操作</th >
            </tr >
        </thead >
        < tbody >
            < tr v - for = "item in arr" :key = "item.id">
                < td v - text = "item.id"></td >
                < td v - text = "item.name"></td >
                < td v - text = "item.time"></td >
                < td >< a href = "" @click.prevent = "del(item.id)">删除</a ></td >
            </tr >
        </tbody >
    </table >
</div >
< script >
    const vm = Vue.createApp({
      data(){
         return{
            arr:[
                {'id':1,'name':'三国演义','time':new Date()},
                {'id':2,'name':'红楼梦','time':new Date()},
                {'id':3,'name':'西游记','time':new Date()},
                {'id':4,'name':'水浒传','time':new Date()}
            ], //创建一些初始数据与格式
                id:'',
                name:''
        }
      },
      methods:{
          add(){
          this.arr.push({'id':this.id,'name':this.name,'time':new Date()});
                  this.id = this.name = '';
        },//add 方法实现列表的输入功能
        del(id){
            var index = this.arr.findIndex(item => {
```

```
                    if(item.id == id) {
                        return true;
                    }
                })
            /*findIndex方法用于查找索引以实现列表的删除功能*/
                this.arr.splice(index,1)
                }
        }
    }).mount("#app");
</script>
</body>
</html>
```