00-Vue3阶段必会的前置知识

版权说明

代码 & 板书链接

https://gitee.com/sinerry1314/bitedu-vue3

一、变量和常量

1. 变量

```
let name = 'Jack'
let age = 18

name = 'Bit'
age = 20
```

2. 常量

```
const PI = 3.14

const articleList = []

const user = {
  name: 'vue3',
  age: 9
}
```

3. 思考

- const 声明的数组可以添加或删除么?
- const 声明的对象可以添加或修改属性么?

答：可以的，因为数组和对象在JS中属于引用类型, 对其做添加、删除等操作, 并不改变其内存地址。

```
const arr = [1,2,3]
// 添加
arr.push(4)
console.log(arr)
// 删除
arr.shift()
console.log(arr)
const obj = {
  name: 'vue3',
  age: 9
}

// 添加属性
obj.birth = 2015
console.log(obj)
// 修改属性
obj.age = 10
console.log(obj)
```

二、模版字符串

1. 普通字符串

用一对单引号或双引号声明，基本都用单引号

```
// 普通字符串
let name = 'Jack'
let msg = "Hello World"
```

2. 模版字符串

用一对反引号声明

```
let name = `Jack`
let msg = `Hello World`
```

3. 好处

3.1 可任意换行

```
const htmlStr = '<div class="hot-goods-box"><h1>热门商品</h1><p>卖爆了卖爆了卖爆了</p></div>'

const htmlStr = `<div class="hot-goods-box">
  <h1>热门商品</h1>
  <p>卖爆了卖爆了卖爆了</p>
</div>`
```

3.2 可嵌入表达式

避免了繁琐的 加号 做字符串拼接；嵌入的语法 `${表达式}`

```
let name = 'Bit'
let age = 9

// 把 name,age 变量的值放到 xxx 的位置，并且展示是否成年
// let str = 'My name is xxx, I am xxx years old, 未成年 or 已成年'
// 传统的加号拼接
```

```
// let str = 'My name is ' + name + ', I am ' + age + ' years old' + (age >= 18 ? '已成年' : '未成年')
// 模版字符串嵌入表达式
let str = `My name is ${name}, I am ${age} years old, ${age >= 18 ? '已成年' : '未成年'}`
```

三、对象

1. 取值

1.1 点取值

```
const obj = {
  name: 'vue3',
  age: 9
}

console.log(obj.name)
console.log(obj.age)

// 等同于
console.log(obj['name']
console.log(obj['age']
```

1.2 中括号取值

```
const obj = {
  name: 'vue3',
  age: 9
}

let x = 'name'
let y = 'age'

// 正确的
console.log(obj[x])
console.log(obj[y])

// 错误的
console.log(obj.x)
console.log(obj.y)
```

1.3 注意

当属性名是变量的时候, 只能用中括号取值; 否则既可以用点, 也可以用中括号

2. 简写

2.1 属性

当属性名和属性值的名字一样时, 并且配合变量时，可以简写

```
let min = 1
let max = 99

// 以前的写法(不简写)
const obj = {
  min: min,
  max: max
}

// 现在的写法
const obj = {
  min,
  max
}
```

2.2 方法

```
const obj = {
  // 不简写
  fn: function() {

  }
}

// 等同于

const obj = {
  // 简写: 连同 :和function 一起省略
  fn() {

  }
}
```

四、解构赋值

1. 针对的目标

数组 或 对象

2. 作用

让数组和对象的取值更便捷

3. 代码示例

3.1 数组解构

```
const arr = [11, 22, 33]

// eg1: 把 arr 中的3个值分别赋值给变量 a, b, c

// 以前的写法
let a = arr[0]
let b = arr[1]
let c = arr[2]

// 现在的写法
let [a, b, c] = arr

// eg2: 把 arr 中的后两个值赋值给变量 b,c
let [,b,c] = arr

// eg3: 把 arr 中的第1个值赋值给变量 a, 剩余的全部给到变量 rest
let [a, ...rest] = arr
```

3.2 对象解构

```
const obj = {
  name: '比特教育科技',
  age: 9,
  address: '陕西省西安市高新区沣惠南路34号'
}

// eg1: 把 obj 中的3个属性值分别赋值给变量 name, age, address

// 以前的写法
```

```
const name = obj.name
const age = obj.age
const address = obj.address

// 现在的写法
const { name, age, address } = obj

// eg2: 把 obj 的 age, address 属性值赋值给 age, address
const { age, address } = obj

// eg3: 把 obj 的 name 属性值赋值给变量 name, 剩余的全部给到变量 rest
const { name, ...rest } = obj

// eg4: 把 obj 的 name 属性值赋值给变量 uname
const { name: uname } = obj
```

4. 练习

```
const arr = [2,[3, 4],5]

// 1、把 arr 中的3, 4赋值给变量 a,b

const obj = {
  data: {
    code: 10000,
    message: '频道列表获取成功',
    result: ['HTML', 'CSS', 'JavaScript', 'Vue', 'SprintBoot']
  },
  status: 200,
  statusText: 'Ok'
}

// 2、把 obj 中的 code, message, result 的值取出来赋值给变量 code, message, list
```

五、箭头函数

1. 非箭头函数

```
// 有名函数
function fn() {
  // some code...
}
```

```
// 函数表达式
const fn = function() {
  // some code...
}
```

2. 语法

```
const fn = () => {
  // some code...
}

const add = (x, y) => {
  return x + y
}
```

3. 特点

- 参数一个可以小括号、函数体一句话可以省略大括号

```
const log = arg => {
  console.log(arg)
}
```

- 当省略了大括号时, 函数自带 return

```
const add = (x, y) => x + y

// 等同于
const add = (x, y) => {
  return x + y
}
```

- 当函数体直接返回一个对象, 如果简写, 需要给对象加一对小括号

```
const state = () => ({
  token: 'xxxx',
  userInfo: {
    name: 'admin',
    id: 1
```

```
  }
})

// 等同于
const state = () => {
  return {
    token: 'xxxx',
    userInfo: {
      name: 'admin',
      id: 1
    }
  }
}
```

4. 应用

可用于普通函数的声明, 也多用于回调函数传参

```
setTimeout(() => {

}, 2000)
```

六、数组的重要方法

1. 概述

数组是js的重要数据结构, 掌握对数组的操作显得格外重要

2. 添加

`push()和unshift()`

```
const arr = [11, 22, 33]

// 尾部添加
const len = arr.push(44)
console.log(len)// 4

// 头部添加
const len = arr.unshift(44)
console.log(len)// 4
```

3. 删除

`pop()和shift()`

```
const arr = [11, 22, 33]

// 尾部删除
const last = arr.pop()
console.log(last)// 33

// 头部删除
const first = arr.shift()
coonsole.log(first)// 11
```

4. 任意位置删除或添加

`splice()`

```
const arr = [11, 22, 33, 44]


// 语法
// arr.splice(startIndex:number, delCount:number, ...addItem)

// 删除 22
const temp =  arr.splice(1, 1)
console.log(temp)// [22]
console.log(arr)// [11, 33, 44]

// 33后面添加55
arr.splice(2, 0, 55)
console.log(arr)// [11, 22, 33, 55, 44]
```

5. 包含

`includes()`

```
const arr = [11, 22, 33, 44]

console.log(arr.includes(33))// true

console.log(arr.includes(55))// false
```

6. 遍历

`forEach()`

```
const arr = [11, 22, 33, 44]

// for循环遍历
for(let i = 0; i < arr.length; i++) {
  console.log(arr[i])
}

// forEach遍历
arr.forEach((item, index, array) => {
  // item: 每次遍历的元素
  // index: 元素的下标
  // array: 数组本身
})
```

7. 过滤

`filter()`

保留满足条件的、去掉不满足条件的

```
const arr = [11, 22, 33, 44]

// 语法
const filteredArr = arr.filter((item, index, array) => {
  // 内部会遍历数组，没遍历一次都会执行回调一次
  // 如果返回true,则当前元素会保留；否则去掉
  return 布尔值
})

// 保留所有的偶数
const eventArr = arr.filter((item) => {
  if(item % 2 === 0) {
     return true
  } else {
    return false
  }
})

// 简写
const eventArr = arr.filter((item) => item % 2 === 0)
```

8. 映射

`map()`

由一个数组得到另一个数组, 并且二者长度相同、每个元素存在一一对应关系

```
const arr = [11, 22, 33, 44]
// 得到 [22, 44, 66, 88]

// 语法
const mappedArr = arr.map((item, index, array) => {
  return 新值
})

// 得到每个元素翻倍的新数组
const doubleArr = arr.map((item) => {
  // 原来每个元素乘 2
  return item * 2
})

// 简写
const doubleArr = arr.map((item) => item * 2)


// 思考如何由arr得到如下数组
[
  { index: 0, value: 11 },
  { index: 1, value: 22 },
  { index: 2, value: 33 },
  { index: 3, value: 44 }
]


const newArr = arr.map((value, index) =>({ index, value }))
```

9. 检测每一个

`every()`

```
const arr = [11, 22, 33, 44]

// 语法
arr.every((item, index, array) => {
```

```
  // 1、如果返回true，说明当前元素满足条件，则继续检测下一次；
  //    若都满足条件，则最终返回true
  // 2、如果返回false，说明当前元素不满足条件；
  //    立即停止检测，最终返回false
  return 布尔值
})

// 判断arr中的元素是否都是奇数
const bool = arr.every((item) => {
  console.log(item)
  return item % 2 === 1
})

// 判断arr中的元素是否都大于10
const bool = arr.every((item) => {
  console.log(item)
  return item > 10
})
```

10. 汇总

`reduce()`

```
const arr = [11, 22, 33, 44]

// 对arr求和
let sum = 0
arr.forEach((item) => {
  sum += item
})
console.log(sum)

// 语法
const result = arr.reduce((prev, item, index, array) => {
  return 结果
}, 初始值)

// 对arr求和
const sum = arr.reduce((prev, item) => {
  return prev + item
}, 0)

// 简写
const sum = arr.reduce((prev, item) => prev + item, 0)

```

```
23
24 // 商品列表数组
25 const goodsList = [
26   { id: 1, name: '篮球', num: 1 },
27   { id: 2, name: '玩具', num: 3 },
28   { id: 3, name: '书籍', num: 2 }
29 ]
30 // 求总数量?
31 const totalNum = goodsList.reduce((prev, item) => prev + item.num, 0)
```

七、对象的重要方法

`Object.keys()`

```
1  const obj = {
2    id: 10001,
3    name: 'Bit',
4    age: 9,
5    address: '陕西省西安市高新区沣惠南路34号'
6  }
7  // 之前遍历对象: for-in循环
8  for(let key in obj) {
9    console.log(key, obj[key])
10 }
11
12 // 获取对象键的数组
13 const keys = Object.keys(obj)
14 console.log(keys)// ['id', 'name', 'age']
15
16 // 可用来遍历对象
17 Object.keys(obj).forEach(key => {
18   console.log(obj[key])
19 })
20
21 // Object.keys()拿到的是键的数组，可以对数组做很多处理在进行遍历对象
22
23 // 需求: 获取 obj 所有以 a 开头的属性值
24 Object
25     .keys(obj)
26     .filter(key => key.startsWith('a'))
27     .forEach(key => {
28       console.log(key, obj[key])
29     })
```

八、扩展运算符

`...`

1. 复制数组或对象

```
const arr1 = [11, 22, 33]
// 赋值
const arr2 = arr1
arr2.push(44)
console.log(arr1)// 受影响了

// 正确的做法，把 arr1 复制一份给到 arr2
const arr2 = [...arr1]


const obj1 = {
  id: 10001,
  name: 'Bit',
  age: 9
}
// 赋值
const obj2 = obj1
obj2.age = 10
console.log(obj1)// 受影响了

// 正确的做法，把 obj1 复制一份给到 obj2
const obj2 = {...obj1}
```

2. 合并数组或对象

```
const arr1 = [1,2,3]
const arr2 = [4,5,6]
// 把 arr1 和 arr2 合并起来给到 arr
const arr = [...arr1, ...arr2]


const obj1 = {
  name: 'Jack',
  height: 176
}
const obj2 = {
  height: 180,
  age: 18
```

```
}
// 把 obj1 和 obj2 合并起来给到 obj

// 注意：同名属性会覆盖
const obj = {
 ...obj1,
 ...obj2
}
```

九、序列化和反序列化

1. 序列化

把对象转换为 json 格式字符串

```
// 对象
const json = {
  id: 10001,
  name: 'Bit',
  age: 9
}
// 序列化
const jsonStr = JSON.stringify(json)
```

2. 反序列化

把 json 字符串转换为 json 数据

```
// json字符串
const jsonStr = '{"id": 10001, "name": "Bit", "age": 9}'

// 反序列化
const json = JSON.parse(jsonStr)
```

十、Web存储

1. 介绍

Web Storage 包含如下两种机制：

- `sessionStorage` 该存储区域在页面会话期间可用（即只要浏览器处于打开状态，包括页面重新加载和恢复）。

    - 仅为会话存储数据，这意味着数据将一直存储到浏览器（或选项卡）关闭。
    - 数据永远不会被传输到服务器。
    - 存储限额大于 Cookie（最大 5MB）。
- `localStorage` 即使浏览器关闭并重新打开也仍然存在。
    - 存储的数据没有过期日期，只能通过 JavaScript、清除浏览器缓存或本地存储的数据来清除。

2. 用法

以 localStorage 为例, 学习存、取、删

2.1 存

```
// 存
localStorage.setItem(key:string, value:string)

// eg:
localStorage.setItem('uname', 'Bit')

// 存对象和数组，需要序列化
```

2.2 取

如果 key 存在, 则取出相应的值; 否则取出的值为 null

```
// 取
localStorage.getItem(key:string)

// eg:
const uname = localStorage.getItem('uname')

// 取出对象和数组，需要反序列化
```

2.3 删

```
// 删
localStorage.removeItem(key:string)

localStorage.removeItem('uname')
```

2.4 注意

存储对象和数组需要进行序列化和反序列化

```
const obj = {
  id: 10001,
  name: 'Bit',
  age: 9
}

// 存：序列化
localStorage.setItem('bit', JSON.stringify(obj))
// 取：反序列化
const local = JSON.parse(localStorage.getItem('bit'))
```

十一、Promise+Aysnc/Await

1. 为什么需要Promise

是为了消除回调地狱的

```
// 需求：延迟2秒之后输出1，完了之后延迟1秒输出2，完了之后延迟1秒输出3
setTimeout(() => {
  console.log(1)
  setTimeout(() => {
    console.log(2)
    setTimeout(() => {
      console.log(3)
    }, 1000)
  }, 1000)
}, 2000)

// 上述代码存在的问题：回调套回调，代码的可读性差
```

为了解决上述问题、Promise就应运而生了

2. Promise介绍

Promise是一个类, 用来包装异步操作, 根据异步操作的结果, 是成功还是失败,

进而决定Promise是成功还是失败;Promise支持链式调用, 从而消除回调地狱

3. Promise的3种状态

```
Pending:    进行中
Fulfilled:  成功
Rejected:   失败
```

Promise的状态

1. 只能由 `Pending -> Fulfilled`,或 `Pening -> Rejected`
2. Promise的状态`一旦确定，就不可改变了`

4. 基本使用

```
const p = new Promose((resolve, reject) => {
  // 这里编写异步代码：比如定时器、ajax请求等
  setTimeout(() => {
    // 2秒后，Promise标记为成功
    resolve('ok')

    reject('error')
  }, 2000)
})

p.then((msg) => {
  // 成功回调
  console.log(msg)// ok
}, (err) => {
  // 失败回调
  console.log(err)// error
})
```


5. 消除上述回调地狱

```
// 封装延迟函数
function delay(duration, n) {
  return new Promise((resolve) => {
   setTimeout(() => {
     resolve(n)
   }, duration)
 })
}
```

```
9
10 // 链式调用消除回调地狱
11 delay(2000, 1)
12  .then(n1 => {
13    console.log(n1)
14    return delay(1000, 2)
15  }).then(n2 => {
16    console.log(n2)
17    return delay(1000, 3)
18  }).then(n3 => {
19    console.log(n3)
20  })
```

上述代码虽然消除了回调地狱、但链式调用过长、也不利于阅读

为了继续优化、Aysnc+Await就应运而生了

6. Async+Await异步终极解决方案

```
1 // 封装延迟函数
2 function delay(duration, n) {
3   return new Promise((resolve) => {
4    setTimeout(() => {
5      resolve(n)
6    }, duration)
7  })
8
9
10 // 语法
11 async function log() {
12   // 1、在Promise实例前添加 await 关键字，那么await的返回值就是当前
13      // Promise的resolve参数
14   // 2、await所在的函数必须被async修饰
15   // 3、async函数内，当前await执行结束了,代码才会继续往后执行(同步的方式执行)
16   const n1 = await delay(2000, 1)
17   console.log(n1)
18 }
19 log()
20
21
22 // 最终代码
23 async function log() {
24   const n1 = await delay(2000, 1)
25   console.log(n1)
26   const n2 = await delay(1000, 2)
```

```
  console.log(n2)
  const n3 = await delay(1000, 3)
  console.log(n3)
}

log()
```

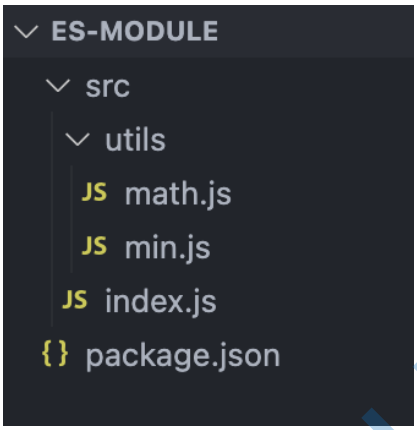
十二、模块化

1. 概述

模块化是指将一个复杂程序划分为一系列独立、可互操作的模块的过程。每个模块负责特定的功能或任务，并通过定义好的接口与其他模块进行通信。简单来说, 就是将代码进行分解、按功能进行管理。

模块化的目的是提高代码的可维护性、可重用性、可测试性和可扩展性。开发者能够更容易地处理大型JavaScript项目。

2. 目录结构准备



1. 新建 12-es-module 目录
2. 命令行进入 12-es-module 目录
3. 执行 `npm init` , 初始化得到 `package.json` 文件
4. 给 package.json 添加 `"type": "module"`
5. 根目录下新建 `src/index.js` 、作为代码的入口文件
6. src 目录下新建 utils 目录

3. 默认导出与导入

新建 `utils/min.js`

```
// 定义并默认导出求最小值函数
export default function min(m, n) {
```

```
  return m > n ? n : m
}

// 注意: 默认导出 export default 在一个模块中最多出现1次
```

`src/index.js`

```
// 默认导入
import min from './utils/min.js'

console.log(min(12, 45))
```

4. 按需导出与导入

新建 `utils/math.js`

```
// 定义求和函数并按需导出
export function add(x, y) {
  return x + y
}

// 定义作差函数并按需导出
export function sub(x, y) {
  return x - y
}

// 注意: 按需导出 export 在一个模块中可以出现多次
```


`src/index.js`

```
// 按需导入
import { add, sub } from './utils/math.js'

console.log(add(33, 18))
console.log(sub(33, 18))
```