JavaScript 高级第一天

作用域&解构&箭头函数

黑马程序员
www.itheima.com

传智教育旗下
高端IT教育品牌

学习目标

Learning Objectives

1. 掌握作用域等概念加深对JS理解
2. 学习ES6新特性让代码书写更加简洁便利

目录

Contents

作用域

- 局部作用域
- 全局作用域
- 作用域链
- JS垃圾回收机制
- 闭包
- 变量提升

1. 作用域

目标：了解作用域对程序执行的影响及作用域链的查找机制，使用闭包函数创建隔离作用域避免全局变量污染。

- 作用域（scope）规定了变量能够被访问的“范围”，离开了这个“范围”变量便不能被访问，
- 作用域分为：
  - 局部作用域
  - 全局作用域

1.1 局部作用域

局部作用域分为函数作用域和块作用域。

1. **函数作用域：**

在函数内部声明的变量只能在函数内部被访问，外部无法直接访问。

```
<script>
  function getSum() {
    // 函数内部是函数作用域 属于局部变量
    const num = 10
  }
  console.log(num) // 此处报错 函数外部不能使用局部作用域变量
</script>
```

总结：

1. 函数内部声明的变量，在函数外部无法被访问
2. 函数的参数也是函数内部的局部变量
3. 不同函数内部声明的变量无法互相访问
4. 函数执行完毕后，函数内部的变量实际被清空了

1.1 局部作用域

局部作用域分为函数作用域和块作用域。

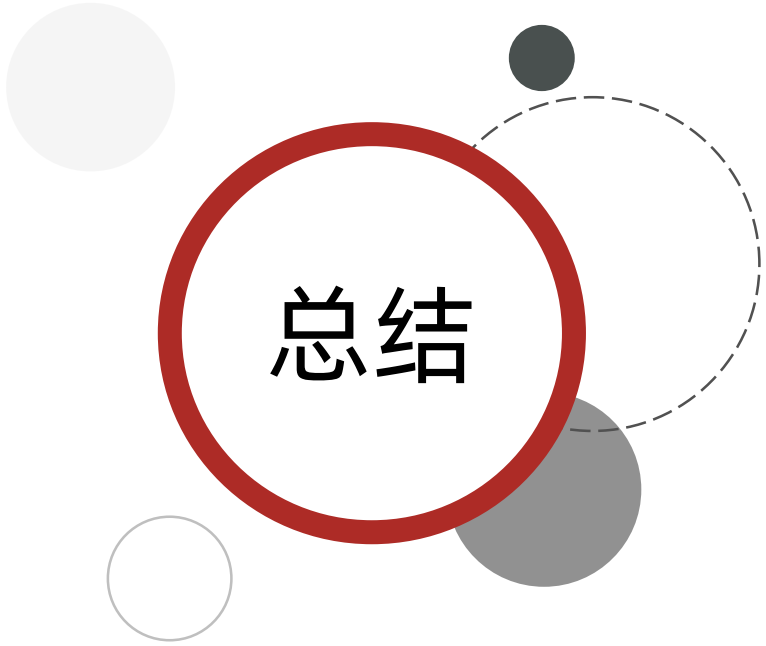
2. 块作用域：

在 JavaScript 中使用 {} 包裹的代码称为代码块，代码块内部声明的变量外部将【有可能】无法被访问。

```
for (let t = 1; t <= 6; t++) {
  // t 只能在该代码块中被访问
  console.log(t) // 正常
}
// 超出了 t 的作用域
console.log(t) // 报错
```

总结：

1. let 声明的变量会产生块作用域，var 不会产生块作用域
2. const 声明的常量也会产生块作用域
3. 不同代码块之间的变量无法互相访问
4. 推荐使用 let 或 const



1. 局部作用域分为哪两种？
   - 函数作用域 函数内部
   - 块级作用域 {}
2. 局部作用域声明的变量外部能使用吗？
   - 不能

作用域

- 局部作用域
- 全局作用域
- 作用域链
- JS垃圾回收机制
- 闭包
- 变量提升

1.2 全局作用域

<script> 标签 和 .js 文件 的【最外层】就是所谓的全局作用域，在此声明的变量在函数内部也可以被访问。

全局作用域中声明的变量，任何其它作用域都可以被访问

```
<script>
  // 全局作用域
  // 全局作用域下声明了 num 变量
  const num = 10
  function fn() {
    // 函数内部可以使用全局作用域的变量
    console.log(num)
  }
  // 此处全局作用域
</script>
```

注意：

1. 为 window 对象动态添加的属性默认也是全局的，不推荐！

2. 函数中未使用任何关键字声明的变量为全局变量，不推荐！！！

3. 尽可能少的声明全局变量，防止全局变量被污染

总结

1. 全局作用域有哪些？
   - <script> 标签内部
   - .js 文件
2. 全局作用域声明的变量其他作用域能使用吗？
   - 相当能

JavaScript 中的作用域是程序被执行时的底层机制，了解这一机制有助于规范代码书写习惯，避免因作用域导致的语法错误。

作用域

- 局部作用域
- 全局作用域
- 作用域链
- 闭包
- 变量提升

思考

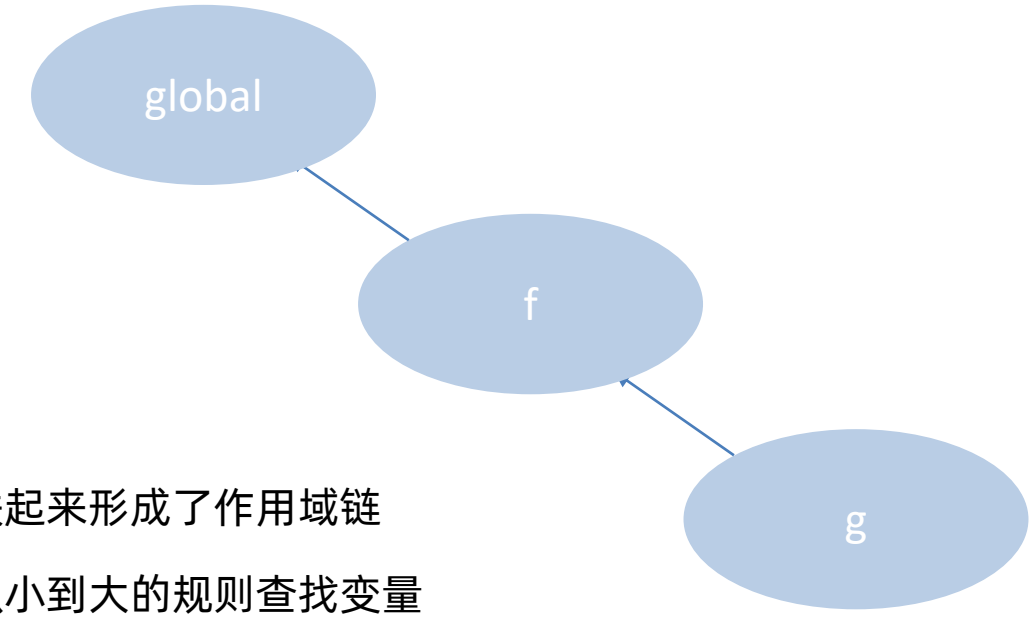
1. 代码有错误吗？如果没有错误结果是几？

```
<script>
  // 全局作用域
  let a = 1
  let b = 2
  // 局部作用域
  function f() {
    let a = 1
    // 局部作用域
    function g() {
      a = 2
      console.log(a)
    }
    g() // 调用g
  }
  f() //  调用f
</script>
```

1.3 作用域链

作用域链本质上是底层的变量查找机制。

- 在函数被执行时，会优先查找当前函数作用域中查找变量
- 如果当前作用域查找不到则会依次逐级查找父级作用域直到全局作用域




```
<script>
  // 全局作用域
  let a = 1
  let b = 2
  // 局部作用域
  function f() {
    let a = 1
    // 局部作用域
    function g() {
      a = 2
      console.log(a)
    }
    g() // 调用g
  }
  f() //  调用f
</script>
```

总结：

1. 嵌套关系的作用域串联起来形成了作用域链
2. 相同作用域链中按着从小到大的规则查找变量
3. 子作用域能够访问父作用域，父级作用域无法访问子级作用域

总结

1. 作用域链本质是什么？
   - 作用域链本质上是底层的变量查找机制
2. 作用域链查找的规则是什么？
   - 会优先查找当前函数作用域中查找变量
   - 查找不到则会依次逐级查找父级作用域直到全局作用域

作用域

- 局部作用域
- 全局作用域
- 作用域链
- JS垃圾回收机制
- 闭包
- 变量提升

1.4 JS垃圾回收机制

目标： 了解JS垃圾回收机制的执行过程

学习目的： 为了闭包做铺垫

学习路径：

1. 什么是垃圾回收机制
2. 内存的声明周期
3. 垃圾回收的算法说明

1.4 JS垃圾回收机制

1. 什么是垃圾回收机制？

垃圾回收机制(Garbage Collection) 简称 GC

JS中内存的分配和回收都是自动完成的，内存在不使用的时候会被垃圾回收器自动回收。

正因为垃圾回收器的存在，许多人认为JS不用太关心内存管理的问题

但如果不了解JS的内存管理机制，我们同样非常容易成内存泄漏（内存无法被回收）的情况

不再用到的内存，没有及时释放，就叫做内存泄漏

1.4 JS垃圾回收机制

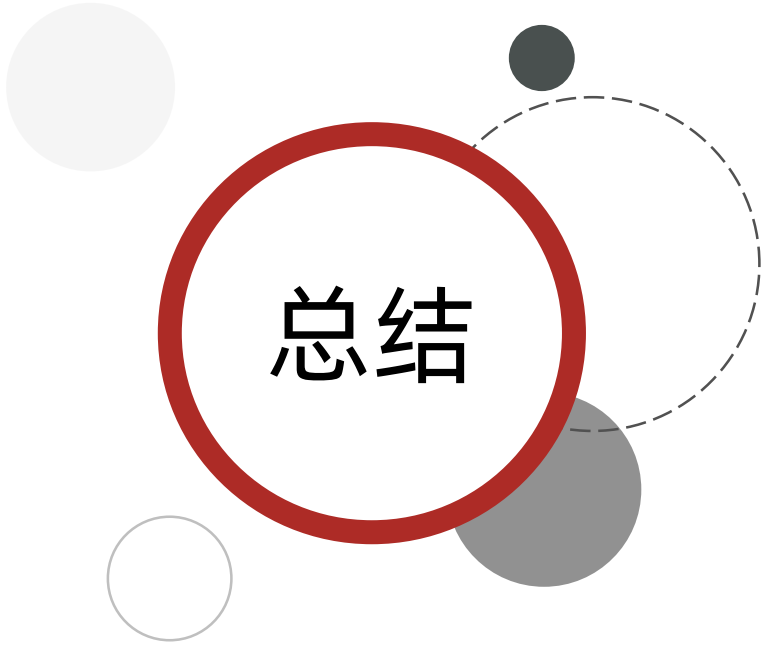
2.内存的生命周期

JS环境中分配的内存，一般有如下生命周期：

1. 内存分配：当我们声明变量、函数、对象的时候，系统会自动为他们分配内存
2. 内存使用：即读写内存，也就是使用变量、函数等
3. 内存回收：使用完毕，由垃圾回收自动回收不再使用的内存
4. 说明：

- 全局变量一般不会回收(关闭页面回收)；
- 一般情况下局部变量的值，不用了，会被自动回收掉

```
// 为变量分配内存
const i = 11
const str = 'pink老师'
// 为对象分配内存
const person = {
  age: 18,
  uname: 'pink老师'
}
// 为函数分配内存
function sum(a, b) {
  return a + b
}
```



1. 什么是垃圾回收机制？
   - 简称 GC
   - JS中内存的分配和回收都是自动完成的，内存在不使用的时候会被垃圾回收器自动回收
2. 什么是内存泄漏？
   - 不再用到的内存，没有及时释放，就叫做内存泄漏
3. 内存的生命周期是什么样的？
   - 内存分配、内存使用、内存回收
   - 全局变量一般不会回收； 一般情况下局部变量的值, 不用了, 会被自动回收掉

1.4 JS垃圾回收机制

3.垃圾回收算法说明

所谓垃圾回收，核心思想就是如何判断内存是否已经不再会被使用了，如果是，就视为垃圾，释放掉

下面介绍两种常见的浏览器垃圾回收算法：引用计数法 和 标记清除法

- **引用计数**

IE采用的引用计数算法，定义“内存不再使用”的标准很简单，就是看一个对象是否有指向它的引用。

算法：

1. 跟踪记录每个值被引用的次数。
2. 如果这个值的被引用了一次，那么就记录次数1
3. 多次引用会累加。
4. 如果减少一个引用就减1。
5. 如果引用次数是0 ，则释放内存。

1.4 JS垃圾回收机制

- **引用计数**

```
const person = {
    age: 18,
    name: 'pink老师'
}
const p = person
person = 1
p = null
```

由上面可以看出，引用计数算法是个简单有效的算法。

但它却存在一个致命的问题：**嵌套引用**。

如果两个对象相互引用，尽管他们已不再使用，垃圾回收器不会进行回收，导致内存泄露。

1.4 JS垃圾回收机制

- **引用计数**

```
function fn() {
    let o1 = {}
    let o2 = {}
    o1.a = o2
    o2.a = o1
    return '引用计数无法回收'
}
fn()
```

因为他们的引用次数永远不会是0。这样的相互引用如果说很大量的存在就会导致大量的内存泄露

1.4 JS垃圾回收机制

3.垃圾回收算法说明

所谓垃圾回收，核心思想就是如何判断内存是否已经不再会被使用了，如果是，就视为垃圾，释放掉

下面介绍两种常见的浏览器垃圾回收算法：引用计数法 和 标记清除法

- **标记清除法**

现代的浏览器已经不再使用引用计数算法了。

现代浏览器通用的大多是基于标记清除算法的某些改进算法，总体思想都是一致的。

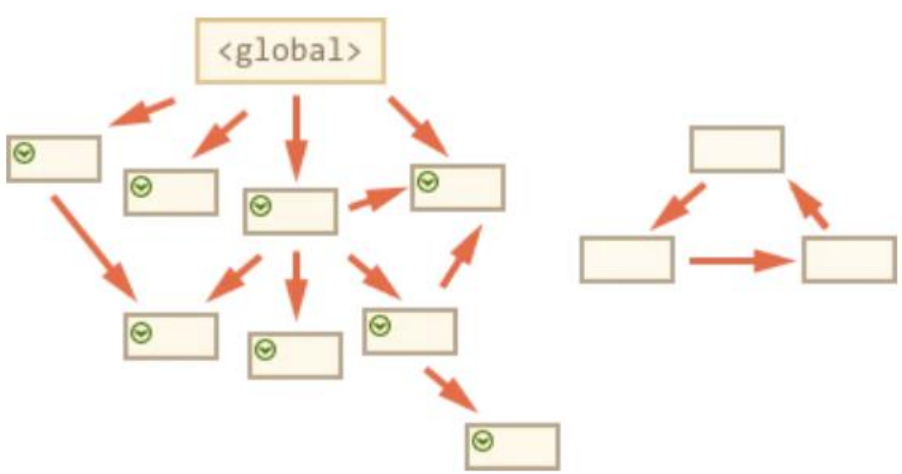
核心：

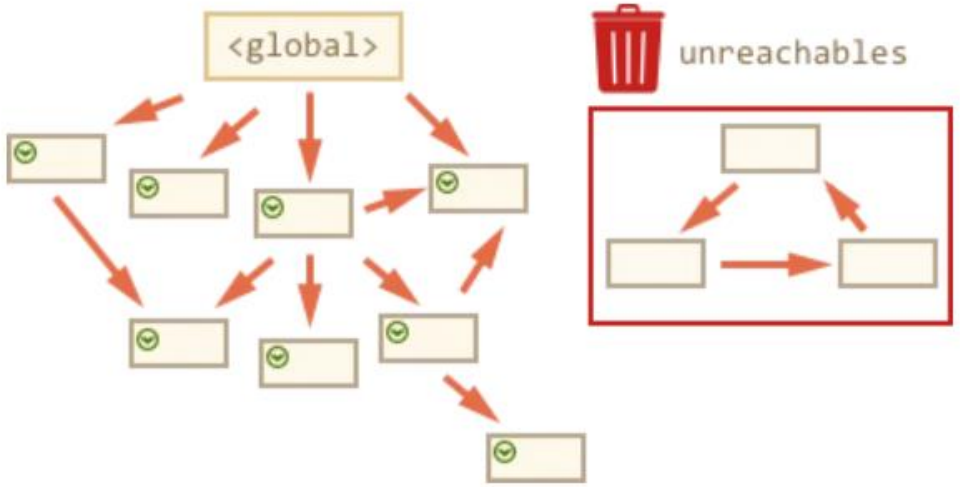
1. 标记清除算法将“不再使用的对象”定义为“无法达到的对象”。
2. 就是从根部（在JS中就是全局对象）出发定时扫描内存中的对象。 凡是能从根部到达的对象，都是还需要使用的。
3. 那些无法由根部出发触及到的对象被标记为不再使用，稍后进 行回收。

1.4 JS垃圾回收机制

- **标记清除法**

标记所有的引用





1.4 JS垃圾回收机制

- **标记清除**

```
function fn() {
    let o1 = {}
    let o2 = {}
    o1.a = o2
    o2.a = o1
    return '引用计数无法回收'
}
fn()
```

根部已经访问不到，所以自动清除

作用域

- 局部作用域
- 全局作用域
- 作用域链
- JS垃圾回收机制
- 闭包
- 变量提升

1.5 闭包

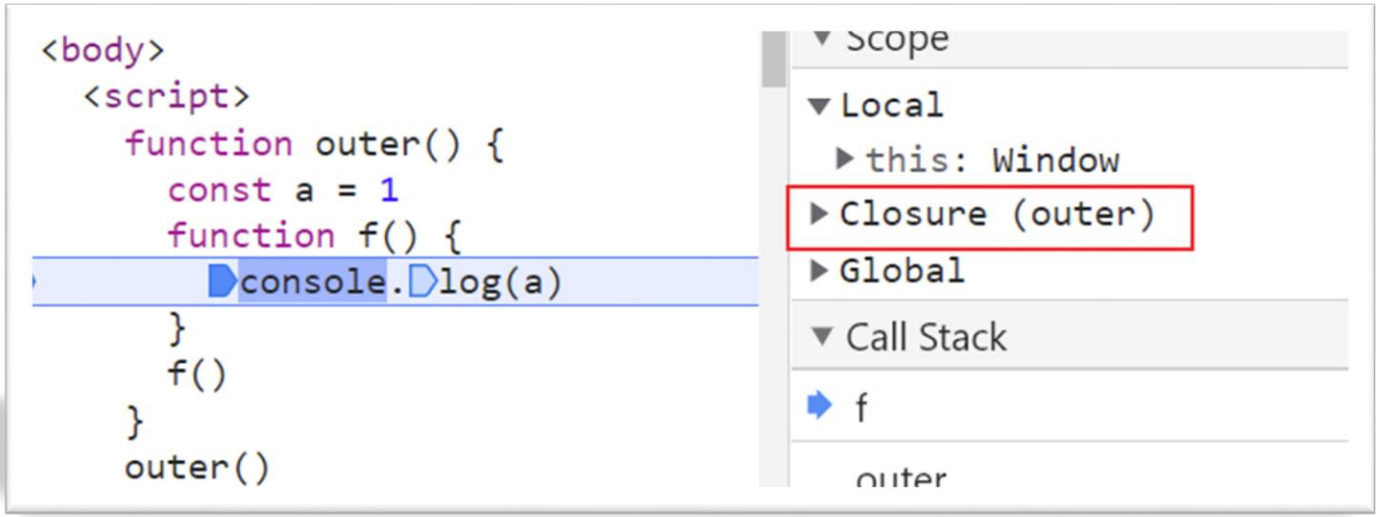
目标： 能说出什么是闭包，闭包的作用以及注意事项

概念：一个函数对周围状态的引用捆绑在一起，内层函数中访问到其外层函数的作用域

简单理解：**闭包 = 内层函数 + 外层函数的变量**

先看个简单的代码：

```
function outer() {
  const a = 1
  function f() {
    console.log(a)
  }
  f()
}
outer()
```



1.5 闭包

闭包作用：封闭数据，提供操作，外部也可以访问函数内部的变量

闭包的基本格式：

```
function outer() {
  let i = 1
  function fn() {
    console.log(i)
  }
  return fn
}
const fun = outer()
fun()  //  1
//外层函数使用内部函数的变量
```

```
// 简约写法
function outer() {
  let i = 1
  return function () {
    console.log(i)
  }
}
const fun = outer()
fun() // 调用fun  1
//外层函数使用内部函数的变量
```

1.5 闭包

闭包应用：实现数据的私有

比如，我们要做个统计函数调用次数，函数调用一次，就++

```
let count = 1
function fn() {
  count++
  console.log(`函数被调用${count}次`)
}
fn()  // 2
fn()  // 3
```

但是，这个count 是个全局变量，很容易被修改

```
function fn() {
  let count = 1
  function fun() {
    count++
    console.log(`函数被调用${count}次`)
  }
  return fun
}
const result = fn()
result() // 2
result() // 3
```

这样实现了数据私有，无法直接修改count

总结

1. 怎么理解闭包？
   - 闭包 = 内层函数 + 外层函数的变量
2. 闭包的作用？
   - 封闭数据，实现数据私有，外部也可以访问函数内部的变量
   - 闭包很有用，因为它允许将函数与其所操作的某些数据（环境）关联起来
3. 闭包可能引起的问题？
   - 内存泄漏

作用域

- 局部作用域
- 全局作用域
- 作用域链
- JS垃圾回收机制
- 闭包
- 变量提升

1.6 变量提升

目标：了解什么是变量提升

变量提升是 JavaScript 中比较“奇怪”的现象，它允许在变量声明之前即被访问（仅存在于var声明变量）

注意：

1. 变量在未声明即被访问时会报语法错误
2. 变量在var声明之前即被访问，变量的值为 undefined
3. let/const 声明的变量不存在变量提升
4. 变量提升出现在相同作用域当中
5. 实际开发中推荐先声明再访问变量

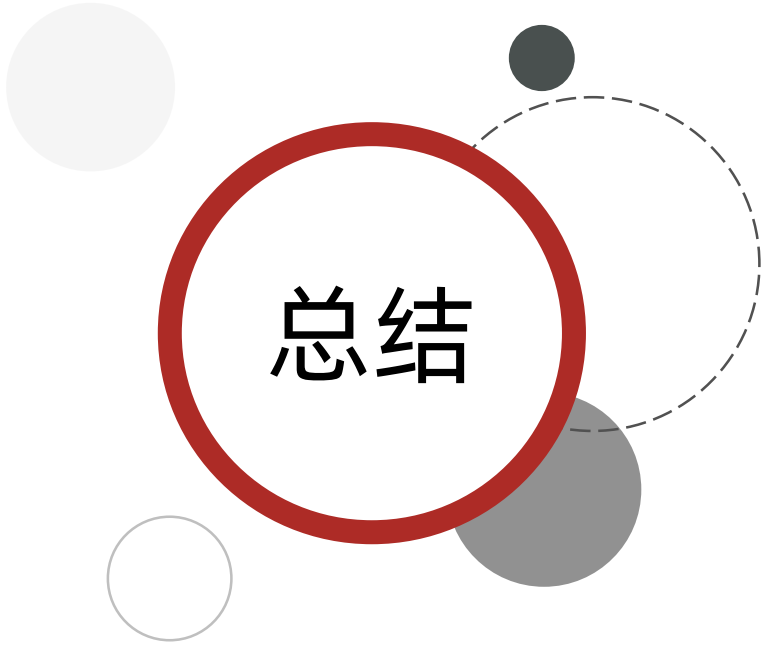
```
<script>
  // 访问变量 str
  console.log(str + 'world!')
  // 声明变量 str
  var str = 'hello '
</script>
```

1.6 变量提升

目标：了解什么是变量提升

说明：

JS初学者经常花很多时间才能习惯变量提升，还经常出现一些意想不到的bug，正因为如此，ES6 引入了块级作用域，用let 或者 const声明变量，让代码写法更加规范和人性化。



1. 用哪个关键字声明变量会有变量提升？
   - var
2. 变量提升是什么流程？
   - 先把var 变量提升到当前作用域于最前面
   - 只提升变量声明， 不提升变量赋值
   - 然后依次执行代码

我们不建议使用var声明变量

目录

Contents

02

函数进阶

- 函数提升
- 函数参数
- 箭头函数

知道函数参数默认值、动态参数、剩余参数的使用细节，提升函数应用的灵活度，知道箭头函数的语法及与普通函数的差异。

2.1 函数提升

目标：能说出函数提升的过程

函数提升与变量提升比较类似，是指函数在声明之前即可被调用。

```
// 调用函数
foo()
// 声明函数
function foo() {
  console.log('声明之前即被调用...')
}
```

```
// 不存在提升现象
bar()  // 错误
var bar = function () {
  console.log('函数表达式不存在提升现象...')
}
```

总结：

1. 函数提升能够使函数的声明调用更灵活
2. 函数表达式不存在提升的现象
3. 函数提升出现在相同作用域当中

02

函数进阶

- 函数提升
- 函数参数
- 箭头函数

知道函数参数默认值、动态参数、剩余参数的使用细节，提升函数应用的灵活度，知道箭头函数的语法及与普通函数的差异。

2.2 函数参数

函数参数的使用细节，能够提升函数应用的灵活度。

学习路径：

1. 动态参数
2. 剩余参数

2.2 函数参数

产品需求： 写一个求和函数

不管用户传入几个实参，都要把和求出来

```
getSum(2, 3)
getSum(1, 2, 3)
getSum(1, 2, 3, 4, 5, 6)
```

形参我改咋写？

```
getSum(???)
```

2.2 函数参数

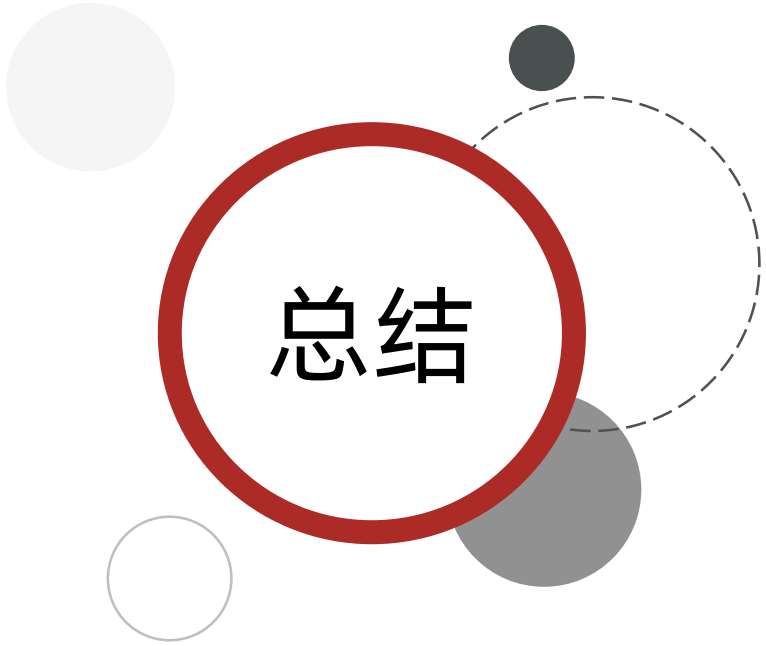
1. 动态参数

arguments 是函数内部内置的伪数组变量，它包含了调用函数时传入的所有实参

```
// 求生函数，计算所有参数的和
function sum() {
  // console.log(arguments)
  let s = 0
  for(let i = 0; i < arguments.length; i++) {
    s += arguments[i]
  }
  console.log(s)
}
// 调用求和函数
sum(5, 10) // 两个参数
sum(1, 2, 4) // 两个参数
```

总结：

1. arguments 是一个伪数组，只存在于函数中
2. arguments 的作用是动态获取函数的实参
3. 可以通过for循环依次得到传递过来的实参



1. 当不确定传递多少个实参的时候，我们怎么办？
   - arguments 动态参数
2. arguments是什么？
   - 伪数组
   - 它只存在函数中

2.2 函数参数

函数参数的使用细节，能够提升函数应用的灵活度。

学习路径：

1. 动态参数
2. 剩余参数

2.2 函数参数

产品需求： 写一个求和函数

不管用户传入几个实参，都要把和求出来

```
getSum(2, 3)
getSum(1, 2, 3)
getSum(1, 2, 3, 4, 5, 6)
```

形参我改咋写？

```
getSum(???)
```

2.2 函数参数

2. 剩余参数

目标： 能够使用剩余参数

剩余参数允许我们将一个不定数量的参数表示为一个数组

```
function getSum(...other) {
  // other 得到 [1,2,3]
  console.log(other)
}
getSum(1, 2, 3)
```

那和arguments 有什么不同吗？

2.2 函数参数

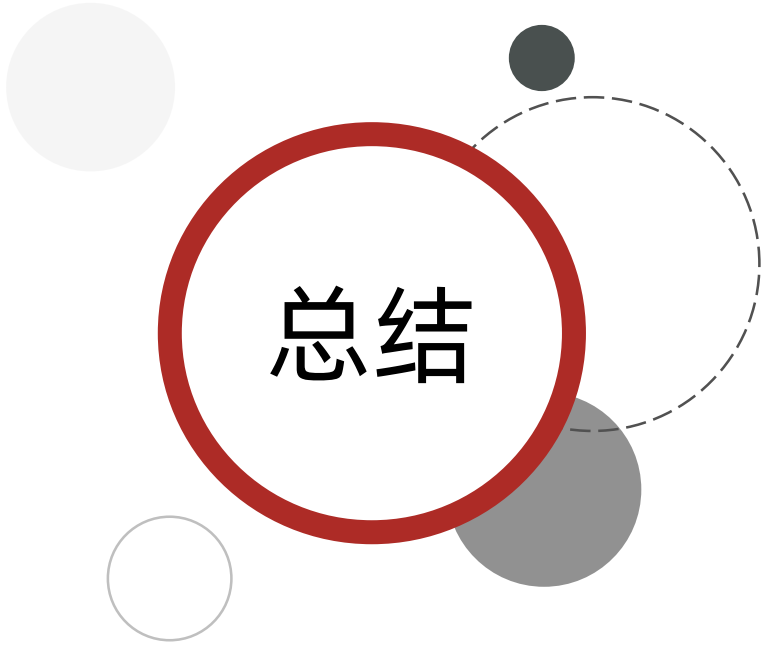
2. 剩余参数

1. ... 是语法符号，置于最末函数形参之前，用于获取多余的实参

2. 借助 ... 获取的剩余实参，是个真数组

```
function config(baseURL, ...other) {
  console.log(baseURL) // 得到 'http://baidu.com'
  console.log(other)  // other  得到 ['get', 'json']
}
// 调用函数
config('http://baidu.com', 'get', 'json');
```

开发中，还是提倡多使用 **剩余参数。**



1. 剩余参数主要的使用场景是？
   - 用于获取多余的实参
2. 剩余参数和动态参数区别是什么？开发中提倡使用哪一个？
   - 动态参数是伪数组
   - 剩余参数是真数组
   - 开发中使用剩余参数想必也是极好的

展开运算符

目标：能够使用展开运算符并说出常用的使用场景

展开运算符(...),将一个数组进行展开

```
const arr = [1, 5, 3, 8, 2]
console.log(...arr) // 1 5 3 8 2
```

说明：

1. 不会修改原数组

展开运算符

目标：能够使用展开运算符并说出常用的使用场景

展开运算符(...),将一个数组进行展开

典型运用场景： 求数组最大值(最小值)、合并数组等

```
const arr = [1, 5, 3, 8, 2]
// console.log(...arr) // 1 5 3 8 2
console.log(Math.max(...arr)) // 8
console.log(Math.min(...arr)) // 1
```

```
// 合并数组
const arr1 = [1, 2, 3]
const arr2 = [4, 5, 6]
const arr3 = [...arr1, ...arr2]
console.log(arr3) // [1,2,3,4,5,6]
```

展开运算符 or 剩余参数

剩余参数：函数参数使用，得到真数组

展开运算符：数组中使用，数组展开

```
function getSum(...other) {
  // other 得到 [1,2,3]
  console.log(other)
}
getSum(1, 2, 3)
```

```
const arr = [1, 5, 3, 8, 2]
console.log(...arr) // 1 5 3 8 2
```

总结

1. 展开运算符主要的作用是？
   - 可以把数组展开,可以利用求数组最大值以及合并数组等操作
2. 展开运算符和剩余参数有什么区别？
   - 展开运算符主要是 数组展开
   - 剩余参数 在函数内部使用

02

函数进阶

- 函数提升
- 函数参数
- 箭头函数

知道函数参数默认值、动态参数、剩余参数的使用细节，提升函数应用的灵活度，知道箭头函数的语法及与普通函数的差异。

2.3 箭头函数（重要）

目标： 能够熟悉箭头函数不同写法

目的：引入箭头函数的目的是更简短的函数写法并且不绑定this，箭头函数的语法比函数表达式更简洁

使用场景：箭头函数更适用于那些本来需要匿名函数的地方

学习路径：

1. 基本语法
2. 箭头函数参数
3. 箭头函数this

2.3 箭头函数（重要）

语法1：基本写法

```
// 普通函数
const fn = function () {
  console.log('我是普通函数')
}
fn()
```

```
// 箭头函数
const fn = () => {
  console.log('俺是箭头函数')
}
fn()
```

2.3 箭头函数（重要）

语法2：只有一个参数可以省略小括号

```
// 普通函数
const fn = function (x) {
  return x + x
}
console.log(fn(1)) // 2
```

```
// 箭头函数
const fn = x => {
  return x + x
}
console.log(fn(1)) // 2
```

2.3 箭头函数（重要）

语法3：如果函数体只有一行代码，可以写到一行上，并且无需写 return 直接返回值

```
// 普通函数
const fn = function (x, y) {
  return x + y
}
console.log(fn(1, 2)) // 3
```

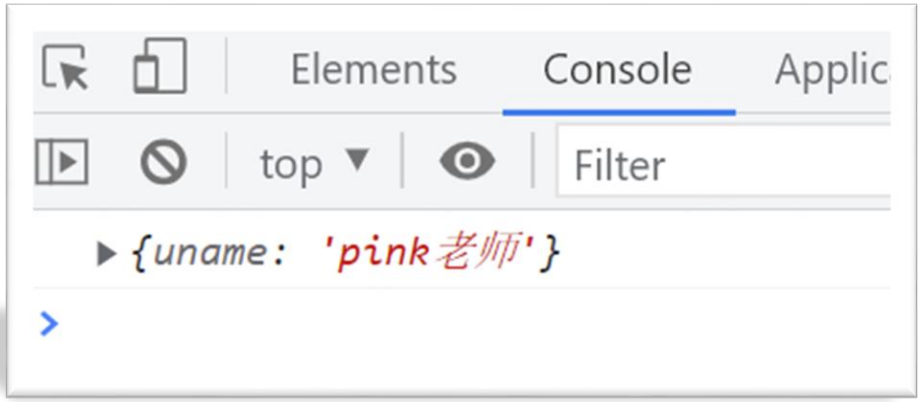
```
// 箭头函数
const fn = (x, y) => x + y

console.log(fn(1, 2)) // 3
```

```
// 更简洁的语法
const form = document.querySelector('form')
form.addEventListener('click', ev => ev.preventDefault())
```

2.3 箭头函数（重要）

语法4：加括号的函数体返回对象字面量表达式

```
const fn1 = uname => ({ uname: uname })
console.log(fn1('pink老师'))
```



总结

1. 箭头函数属于表达式函数，因此不存在函数提升

2. 箭头函数只有一个参数时可以省略圆括号 ()

3. 箭头函数函数体只有一行代码时可以省略花括号 {}，并自动做为返回值被返回

4. 加括号的函数体返回对象字面量表达式

```
// 箭头函数
const fn = x => {
  return x + x
}
console.log(fn(1)) // 2
```

```
// 箭头函数
const fn = (x, y) => x + y

console.log(fn(1, 2)) // 3
```

```
const fn1 = uname => ({ uname: uname })
console.log(fn1('pink老师'))
```

2.3 箭头函数（重要）

目标： 能够熟悉箭头函数不同写法

目的：引入箭头函数的目的是更简短的函数写法并且不绑定this，箭头函数的语法比函数表达式更简洁

使用场景：箭头函数更适用于那些本来需要匿名函数的地方

学习路径：

1. 基本语法
2. 箭头函数参数
3. 箭头函数this

2.3 箭头函数（重要）

2. 箭头函数参数

1. 普通函数有arguments 动态参数
2. 箭头函数没有 arguments 动态参数，但是有 剩余参数 ..args

```
const getSum = (...args) => {
  let sum = 0
  for (let i = 0; i < args.length; i++) {
    sum += args[i]
  }
  return sum    // 注意函数体有多行代码需要return
}
console.log(getSum(1, 2, 3)) // 6
```

总结

1. 箭头函数里面有arguments动态参数吗？可以使用什么参数？
   - 没有arguments动态参数
   - 可以使用剩余参数

```
const getSum = (...args) => {
  let sum = 0
  for (let i = 0; i < args.length; i++) {
    sum += args[i]
  }
  return sum    // 注意函数体有多行代码需要return
}
console.log(getSum(1, 2, 3)) // 6
```

2.3 箭头函数（重要）

目标： 能够熟悉箭头函数不同写法

目的：引入箭头函数的目的是更简短的函数写法并且不绑定this，箭头函数的语法比函数表达式更简洁

使用场景：箭头函数更适用于那些本来需要匿名函数的地方

学习路径：

1. 基本语法
2. 箭头函数参数
3. 箭头函数this

2.3 箭头函数（重要）

3. 箭头函数 this

在箭头函数出现之前，每一个新函数根据它是被如何调用的来定义这个函数的this值， 非常令人讨厌。

箭头函数不会创建自己的this,它只会从自己的作用域链的上一层沿用this。

```
console.log(this) // 此处为window
const sayHi = function () {
  console.log(this) // 普通函数指向调用者 此处为window
}
btn.addEventListener('click', function () {
  console.log(this) // 当前this 指向 btn
})
```

2.3 箭头函数（重要）

3. 箭头函数 this

箭头函数不会创建自己的this,它只会从自己的作用域链的上一层沿用this。

```
console.log(this) // 此处为window
// 箭头函数
const sayHi = () => {
  console.log(this) // 箭头函数此处为window
}
btn.addEventListener('click', () => {
  console.log(this) // 当前this 指向 window
})
```

```
const user = {
  name: '小明',
  // 该箭头函数中的 this 为函数声明环境中 this 一致
  walk: () => {
    console.log(this)  // 指向window 不是user
  }
}
user.walk()
```

2.3 箭头函数（重要）

3. 箭头函数 this

箭头函数不会创建自己的this,它只会从自己的作用域链的上一层沿用this。

```
const user = {
  name: '小明',
  sleep: function () {
    console.log(this) // 指向 user
    const fn = () => {
      console.log(this) // 指向user 该箭头函数中的 this 与 sleep 中的 this 一致
    }
    // 调用箭头函数
    fn()
  }
}
user.sleep()
```

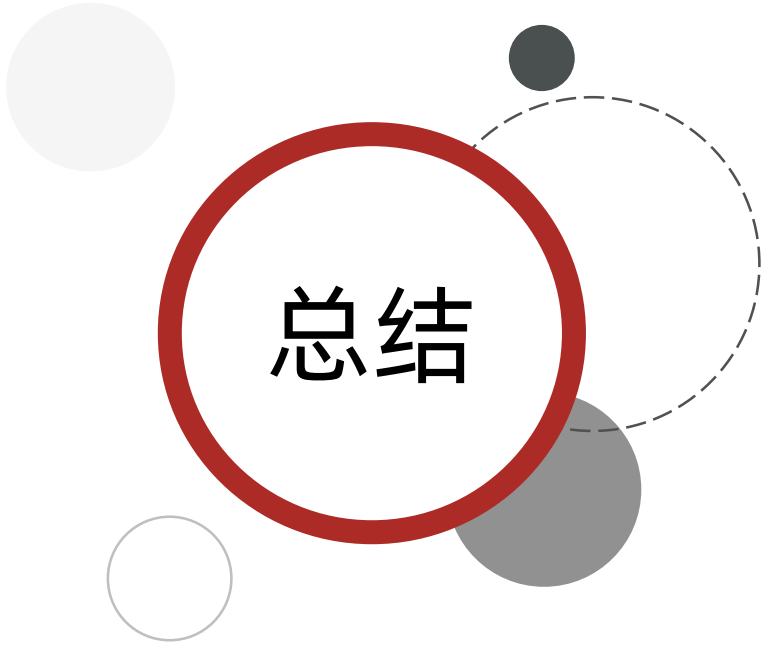
2.3 箭头函数（重要）

3. 箭头函数 this

在开发中【使用箭头函数前需要考虑函数中 this 的值】，事件回调函数使用箭头函数时，this 为全局的 window，因此DOM事件回调函数为了简便，还是不太推荐使用箭头函数

```
<script>
  // DOM 节点
  const btn = document.querySelector('.btn')

  // 箭头函数 此时 this 指向了 window
  btn.addEventListener('click', () => {
    console.log(this)
  })

  // 普通函数 此时 this 指向了 DOM 对象
  btn.addEventListener('click', function () {
    console.log(this)
  })
</script>
```

箭头函数更多this问题，我们第四天再进行讲解



1. 箭头函数里面有this吗？
   - 箭头函数不会创建自己的this,它只会从自己的作用域链的上一层沿用this
2. DOM事件回调函数推荐使用箭头函数吗？
   - 不太推荐，特别是需要用到this的时候
   - 事件回调函数使用箭头函数时，this 为全局的 window

目录

Contents

解构赋值

- 数组解构
- 对象解构

3. 解构赋值

目标：知道解构的语法及分类，使用解构简洁语法快速为变量赋值

```
const arr = [100, 60, 80]
console.log(arr[0]) // 最大值
console.log(arr[1]) // 最小值
console.log(arr[2]) // 平均值
```

```
const arr = [100, 60, 80]
const max = arr[0]
const min = arr[1]
const avg = arr[2]
console.log(max) //最大值 100
console.log(min) // 最小值 60
console.log(avg) // 最小值 80
```

以上要么不好记忆，要么书写麻烦，此时可以使用解构赋值的方法让代码更简洁

```
const [max, min, avg] = [100, 60, 80]
console.log(max) //最大值 100
console.log(min) // 最小值 60
console.log(avg) // 最小值 80
```

3. 解构赋值

解构赋值是一种快速为变量赋值的简洁语法，本质上仍然是为变量赋值。

分为：

- 数组解构
- 对象解构

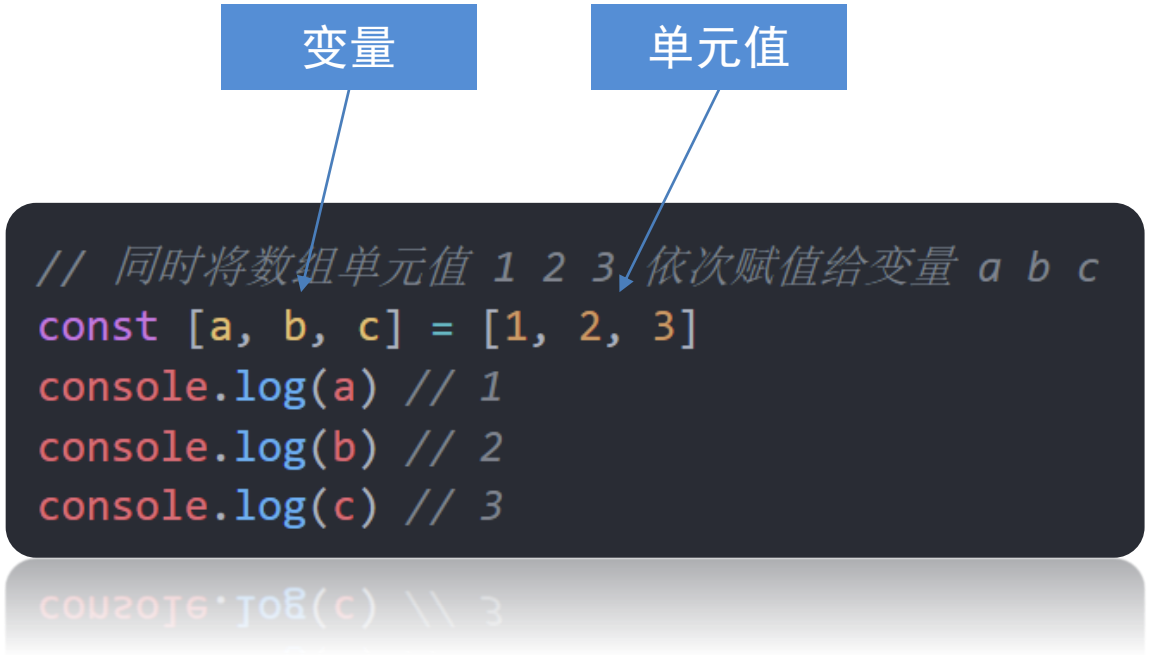
3.1 数组解构

数组解构是将数组的单元值快速批量赋值给一系列变量的简洁语法。

基本语法：

1. 赋值运算符 = 左侧的 [] 用于批量声明变量，右侧数组的单元值将被赋值给左侧的变量
2. 变量的顺序对应数组单元值的位置依次进行赋值操作

```
// 普通的数组
const arr = [1, 2, 3]
// 批量声明变量 a b c
// 同时将数组单元值 1 2 3 依次赋值给变量 a b c
const [a, b, c] = arr
console.log(a) // 1
console.log(b) // 2
console.log(c) // 3
```



3.1 数组解构

数组解构是将数组的单元值快速批量赋值给一系列变量的简洁语法。

基本语法：典型应用交互2个变量

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

这里必须有分号

```
<script>
  let arr = [2, 6, 4, 3, 5, 1]
  // 1. 外层循环控制  趟数   循环 4次   arr.length - 1
  for (let i = 0; i < arr.length - 1; i++) {
    // 2. 里层的循环 控制 一趟交换几次  arr.length - i - 1 次序
    for (let j = 0; j < arr.length - i - 1; j++) {
      // 交换两个变量
      // arr[j]   arr[j + 1]
      if (arr[j] > arr[j + 1]) {
        [arr[j + 1], arr[j]] = [arr[j], arr[j + 1]]
      }
    }
  }
  console.log(arr)
```

3.1 数组解构

数组解构是将数组的单元值快速批量赋值给一系列变量的简洁语法。

基本语法：典型应用交互2个变量

注意： js 前面必须加分号情况

1. 立即执行函数

```
(function t() { })();
// 或者
;(function t() { })()
```

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

2. 数组解构

```
// 数组开头的，特别是前面有语句的一定注意加分号
;[b, a] = [a, b]
```

总结

1. 数组解构赋值的作用是什么？
   - 是将数组的单元值快速批量赋值给一系列变量的简洁语法
2. Js 前面有两哪种情况需要加分号的？
   - 立即执行函数
   - 数组解构

```
(function t() { })();
// 或者
;(function t() { })()
```

```
// 数组开头的，特别是前面有语句的一定注意加分号
;[b, a] = [a, b]
```

练习

- **独立完成数组解构赋值**

需求①： 有个数组： const pc = ['海尔', '联想', '小米', '方正']

解构为变量：hr lx mi fz

需求②：请将最大值和最小值函数返回值解构 max 和min 两个变量

```
function getValue() {
  return [100, 60]
}

// 要求 max 变量里面存100  min 变量里面存 60
```

3.1 数组解构

数组解构是将数组的单元值快速批量赋值给一系列变量的简洁语法。

2. 变量多 单元值少的情况：

```
// 变量多，单元值少
const [a, b, c, d] = ['小米', '苹果', '华为']
console.log(a) // 小米
console.log(b) // 苹果
console.log(c) // 华为
console.log(d) // undefined
```

变量的数量大于单元值数量时，多余的变量将被赋值为 undefined

3.1 数组解构

数组解构是将数组的单元值快速批量赋值给一系列变量的简洁语法。

2. 变量少 单元值多的情况：

```
// 变量少，单元值多
const [a, b, c] = ['小米', '苹果', '华为', '格力']
console.log(a) // 小米
console.log(b) // 苹果
console.log(c) // 华为
```

3.1 数组解构

数组解构是将数组的单元值快速批量赋值给一系列变量的简洁语法。

3. 利用剩余参数解决变量少 单元值多的情况：

```
// 利用剩余参数 变量少，单元值多
const [a, b, ...tel] = ['小米', '苹果', '华为', '格力', 'vivo']
console.log(a) // 小米
console.log(b) // 苹果
console.log(tel) // ['华为', '格力', 'vivo']
```

剩余参数返回的还是一个数组

3.1 数组解构

数组解构是将数组的单元值快速批量赋值给一系列变量的简洁语法。

4. 防止有undefined传递单元值的情况，可以设置默认值：

```
const [a = '手机', b = '华为'] = ['小米']
console.log(a) // 小米
console.log(b) // 华为
```

允许初始化变量的默认值，且只有单元值为 undefined 时默认值才会生效

3.1 数组解构

数组解构是将数组的单元值快速批量赋值给一系列变量的简洁语法。

5. 按需导入，忽略某些返回值：

```
// 按需导入，忽略某些值
const [a, , c, d] = ['小米', '苹果', '华为', '格力']
console.log(a) // 小米
console.log(c) // 华为
console.log(d) // 格力
```

3.1 数组解构

数组解构是将数组的单元值快速批量赋值给一系列变量的简洁语法。

6. 支持多维数组的结构：

```
const [a, b] = ['苹果', ['小米', '华为']]
console.log(a) // 苹果
console.log(b) // ['小米', '华为']
```

```
// 想要拿到 小米和华为怎么办？
const [a, [b, c]] = ['苹果', ['小米', '华为']]
console.log(a) // 苹果
console.log(b) // 小米
console.log(c) // 华为
```

总结

1. 变量的数量大于单元值数量时，多余的变量将被赋值为？
   - undefined
2. 变量的数量小于单元值数量时，可以通过什么剩余获取所有的值？
   - 剩余参数... 获取剩余单元值，但只能置于最末位

3. 解构赋值

解构赋值是一种快速为变量赋值的简洁语法，本质上仍然是为变量赋值。

分为：

- 数组解构
- 对象解构

3.2 对象解构

对象解构是将对象属性和方法快速批量赋值给一系列变量的简洁语法

1. 基本语法：

1. 赋值运算符 = 左侧的 {} 用于批量声明变量，右侧对象的属性值将被赋值给左侧的变量

2. 对象属性的值将被赋值给与属性名相同的变量

3. 注意解构的变量名不要和外面的变量名冲突否则报错

4.对象中找不到与变量名一致的属性时变量值为 undefined

```
<script>
  // 普通对象
  const user = {
    name: '小明',
    age: 18
  };
  // 批量声明变量 name age
  // 同时将数组单元值 小明  18 依次赋值给变量 name  age
  const {name, age} = user

  console.log(name) // 小明
  console.log(age) // 18
</script>
```

3.2 对象解构

对象解构是将对象属性和方法快速批量赋值给一系列变量的简洁语法

2.给新的变量名赋值：

可以从一个对象中提取变量并同时修改新的变量名

```
// 普通对象
const user = {
  name: '小明',
  age: 18
};
// 把 原来的name 变量重新命名为 uname
const { name: uname, age } = user
console.log(uname) // 小明
console.log(age) // 18
```

冒号表示“什么值：赋值给谁”

3.2 对象解构

对象解构是将对象属性和方法快速批量赋值给一系列变量的简洁语法

2. 数组对象解构

```
const pig = [
  {
    name: '佩奇',
    age: 6
  }
]
```

```
const [{ name, age }] = pig
console.log(name, age)
```

练习

- **独立完成对象解构赋值**

需求①： 有个对象： const pig = { name: '佩奇',age: 6 }

结构为变量：完成对象解构，并以此打印出值

需求②：请将pig对象中的name，通过对象解构的形式改为 uname，并打印输出

需求③：请将 数组对象，完成 商品名和价格的解构

```
const goods = [
    {
      goodsName: '小米',
      price: 1999
    }
]
```

3.2 对象解构

对象解构是将对象属性和方法快速批量赋值给一系列变量的简洁语法

3. 多级对象解构：

```
const pig = {
    name: '佩奇',
    family: {
      mother: '猪妈妈',
      father: '猪爸爸',
      sister: '乔治'
    },
    age: 6
}
```

```
// 依次打印家庭成员
const pig = {
  name: '佩奇',
  family: {
    mother: '猪妈妈',
    father: '猪爸爸',
    sister: '乔治'
  },
  age: 6
}
const { name, family: { mother, father, sister } } = pig
console.log(name) // 佩奇
console.log(mother) // 猪妈妈
console.log(father) // 猪爸爸
console.log(sister) // 乔治
```

3.2 对象解构

对象解构是将对象属性和方法快速批量赋值给一系列变量的简洁语法

3. 多级对象解构：

```
const people = [
  {
    name: '佩奇',
    family: {
      mother: '猪妈妈',
      father: '猪爸爸',
      sister: '乔治'
    },
    age: 6
  }
]
```

```
const [{ name, family: { mother, father, sister } }] = people
console.log(name) // 佩奇
console.log(mother) // 猪妈妈
console.log(father) // 猪爸爸
console.log(sister) // 乔治
```

3.2 对象解构

对象解构是将对象属性和方法快速批量赋值给一系列变量的简洁语法

3. 多级对象解构：

```
// 这是后台传递过来的数据，请选出data里面的数据方便后面渲染W
const msg = {
  "code": 200,
  "msg": "获取新闻列表成功",
  "data": [
    {
      "id": 1,
      "title": "5G商用自己，三大运用商收入下降",
      "count": 58
    },
    {
      "id": 2,
      "title": "国际媒体头条速览",
      "count": 56
    },
    {
      "id": 3,
      "title": "乌克兰和俄罗斯持续冲突",
      "count": 1669
    },

  ]
}
```

```
const { data } = msg
console.log(data)
```

```
// 2. 需求， 请将上面对象只选出 data数据，传递给另外一个函数
function render({ data }) {
  // 内部处理
  console.log(data)
}
render(msg)
```

```
// 3. 需求， 为了防止msg里面的data名字混淆，要求渲染函数里面的数据名改为 myData
function render({ data: myData }) {
  // 内部处理
  console.log(myData)
}
render(msg)
```

练习

- **独立完成对象解构赋值**

请将刚才数据完成3个需求

案例

渲染商品列表案例

请根据数据渲染以下效果

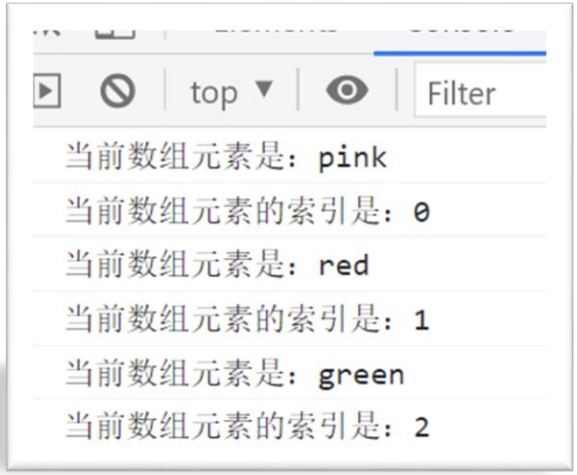
遍历数组 forEach 方法（重点）

- forEach() 方法用于调用数组的每个元素，并将元素传递给回调函数
- 主要使用场景： **遍历数组的每个元素**
- **语法：**

```
被遍历的数组.forEach(function (当前数组元素, 当前元素索引号) {
  // 函数体
})
```

- **例如：**

```
const arr = ['pink', 'red', 'green']
arr.forEach(function (item, index) {
  console.log(`当前数组元素是：${item}`) // 依次打印数组每一个元素
  console.log(`当前数组元素的索引是：${index}`) // 依次打印数组每一个元素的索引
})
```



遍历数组 forEach 方法（重点）

- forEach() 方法用于调用数组的每个元素，并将元素传递给回调函数

```
被遍历的数组.forEach(function (当前数组元素, 当前元素索引号) {
    // 函数体
})
```

注意：

1. forEach 主要是遍历数组
2. 参数当前数组元素是必须要写的， 索引号可选。

步骤

- **渲染商品列表案例**

核心思路：有多少条数据，就渲染多少模块，然后 生成对应的 html结构标签， 赋值给 list标签即可

①：利用forEach 遍历数据里面的 数据

②：拿到数据，利用**字符串拼接**生成结构添加到页面中

③：注意：传递参数的时候，可以使用对象解构

目录

Contents

案例

商品列表价格筛选

需求：

①：渲染数据列表

②：根据选择不同条件显示不同商品

筛选数组 filter 方法（重点）

- filter() 方法创建一个新的数组，新数组中的元素是通过检查指定数组中符合条件的所有元素
- 主要使用场景： **筛选数组符合条件的元素，并返回筛选之后元素的新数组**
- **语法：**

```
被遍历的数组.filter(function (currentValue, index) {
    return 筛选条件
})
```

- **例：**

```
// 筛选数组中大于30的元素
const score = [10, 50, 3, 40, 33]
const re = score.filter(function (item) {
  return item > 30
})
console.log(re)  // [50, 40, 33]
```

筛选数组 filter 方法（重点）

- filter() 筛选数组
- **返回值：**返回数组，包含了符合条件的所有元素。如果没有符合条件的元素则返回空数组
- **参数：**currentValue 必须写， index 可选
- 因为返回新数组，所以不会影响原数组

步骤

• 商品列表价格筛选

业务分析：

①： 页面初始渲染

②： 点击不同需求显示不同的数据

步骤

- **商品列表价格筛选**

分析：

①：渲染页面 利用forEach 遍历数据里面的 数据，并渲染数据列表

②：根据 filter 选择不同条件显示不同商品

步骤

• 商品列表价格筛选

步骤：

①：渲染页面模块

(1) 初始化需要渲染页面，同时，点击不同的需求，还会重新渲染页面，所以渲染做成一个函数

(2) 做法基本跟前面案例雷同，就是封装到了一个函数里面

步骤

• 商品列表价格筛选

步骤：

②：点击不同需求，显示不同页面内容

(1) 点击采取事件委托方式 .filter

(2) 利用过滤函数 filter 筛选出符合条件的数据，因为生成的是一个数组，传递给渲染函数即可

(3) 筛选条件是根据点击的 data-index 来判断

(4) 可以使用对象解构，把 事件对象 解构

(5) 因为 全部区间不需要筛选，直接 把goodList渲染即可

今日作业

1. 整理笔记
2. 综合案例至少写三遍
3. 开始做测试题： PC端地址： https://ks.wjx.top/vj/wOeHuEk.aspx
4. 预习第二天内容

传智教育旗下高端IT教育品牌