

JavaScript 进阶第一天

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
- ◆ 函数进阶
- ◆ 解构赋值
- ◆ 综合案例

01

作用域

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

1. 作用域

- **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。

```
function sum(num) {  
    // 形参num相当于局部变量  
    const num = 10  
}  
sum(1)  
console.log(num) // 外部无法使用函数内部变量
```

总结

1. 函数内部声明的变量，在函数外部无法被访问
2. 函数的形参也可以看做是函数内部的局部变量

1.1 局部作用域

局部作用域分为函数作用域和块作用域。

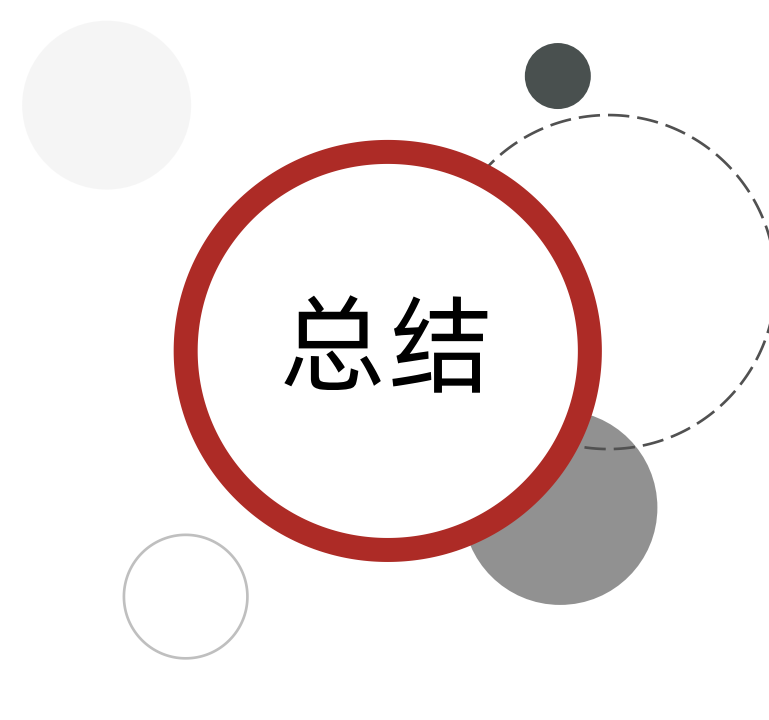
2. 块作用域:

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/const` 声明会产生块作用域，`var` 不会产生块作用域
2. 不同代码块之间的变量无法互相访问
3. 推荐使用 `let` 或 `const`



总结

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



01

作用域

- 局部作用域
- 全局作用域
- 作用域链
- 内存泄漏
- 闭包
- 变量提升

1.2 全局作用域

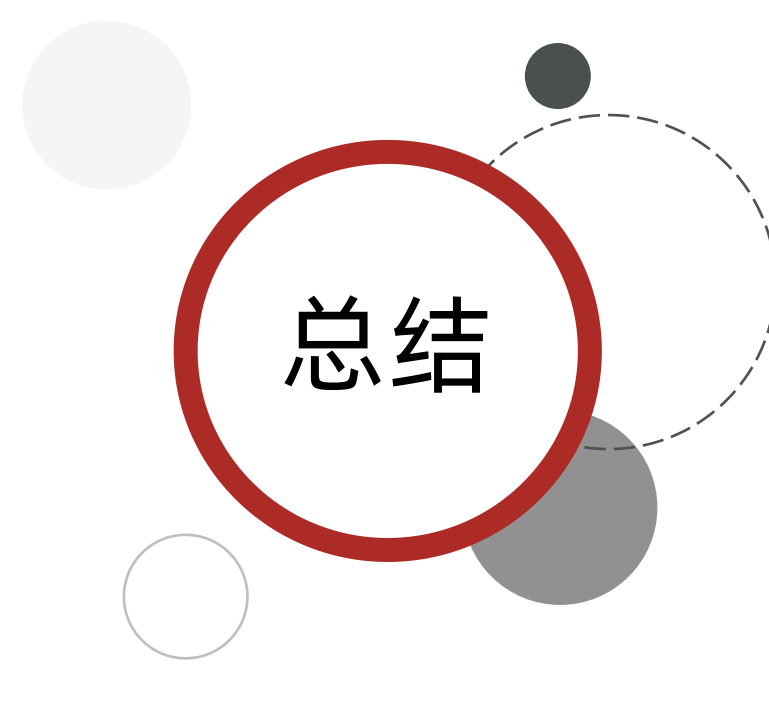
`<script>` 标签 和 `.js` 文件的【最外层】就是所谓的全局作用域，在此声明的变量在函数内部也可以被访问。

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>
```

总结

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?

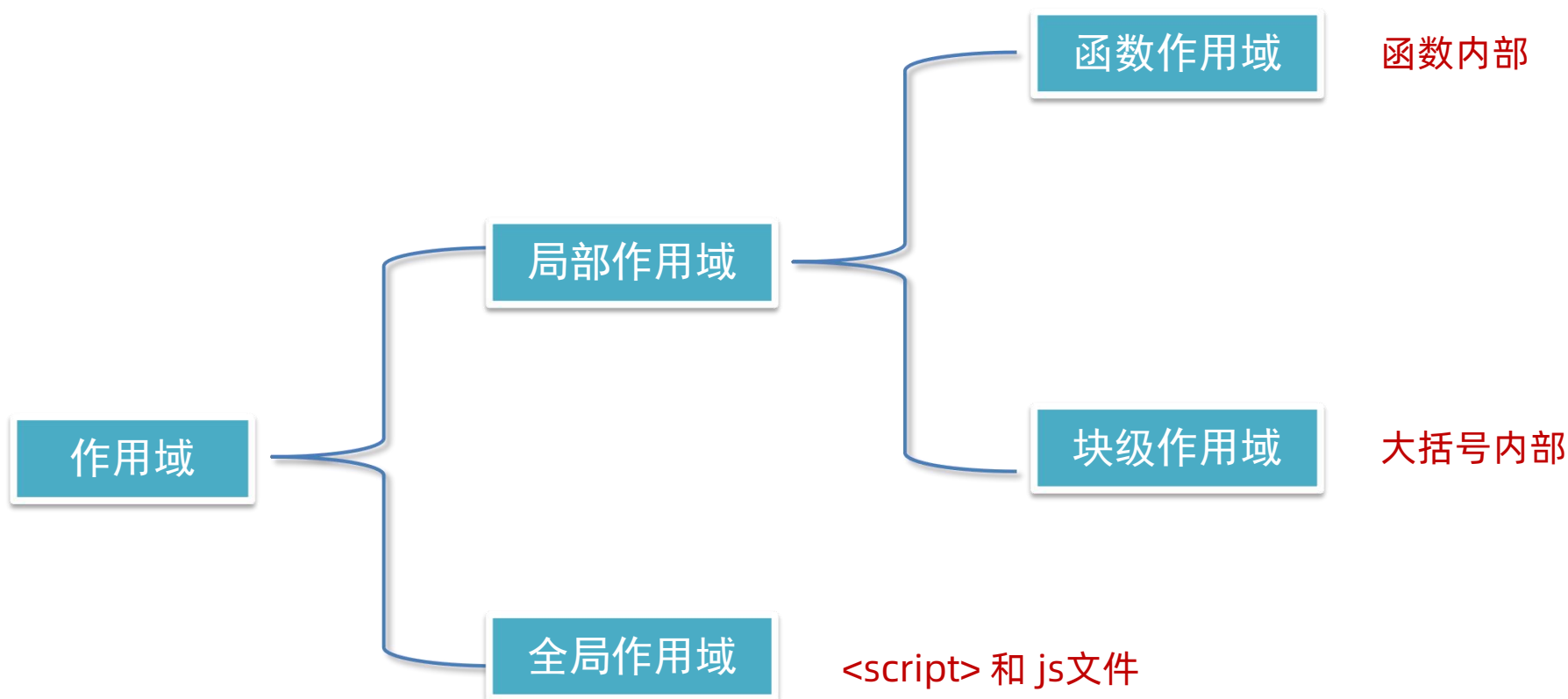
- `<script>` 标签内部
- `.js` 文件

2. 全局作用域声明的变量其他作用域能使用吗?

- 相当能

作用域总结

- **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





01

作用域

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

1.3 作用域链

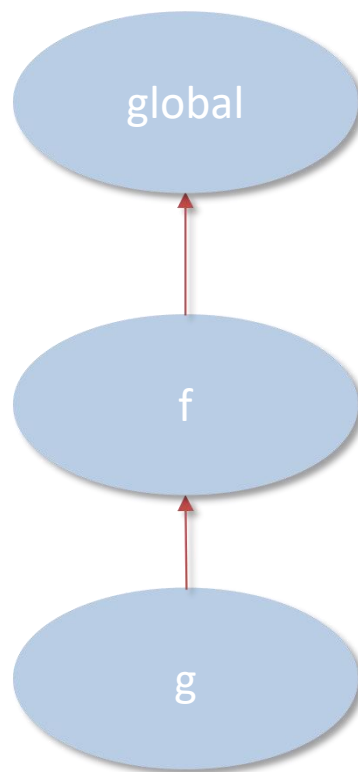
嵌套关系的**作用域串联**起来形成了**作用域链**

作用：作用域链本质上是底层的**变量查找机制**（就近原则）

- 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**逐级向上查找**父级作用域直到全局作用域

总结

1. 嵌套关系的**作用域串联**起来形成了**作用域链**
2. 查找规则：**就近原则**
 - 当前作用域用找不到，则会**逐级查找父级作用域**直到全局作用域
 - 都找不到则提示错误，这个**变量没有被定义过**
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



```
// 全局作用域
let a = 11
let b = 22
// 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
```

```
f() // 调用 f
}
g() // 调用 g
```

01

作用域

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

1.4 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，内存不使用时会被垃圾回收器自动回收

1.4 垃圾回收机制

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**垃圾回收器**自动回收不再使用的内存

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**

内存泄漏：程序中分配的**内存**由于某种原因程序**未释放**或**无法释放**叫做**内存泄漏**

```
// 为变量分配内存
const age = 18

// 为对象分配内存
const obj = {
  age: 19
}

// 为函数分配内存
function fn() {
  const age = 18
  console.log(age)
}

// 为函数分配内存
function fn() {
  const age = 18
  console.log(age)
}
```



01

作用域

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

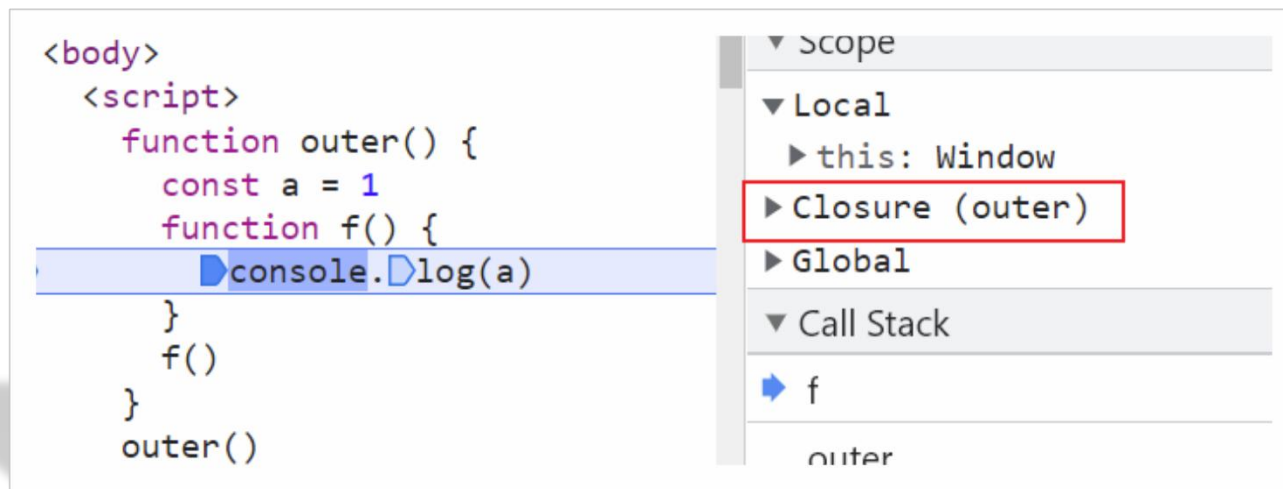
1.5 闭包

概念：一个函数对周围状态的引用捆绑在一起，闭包让开发者可以从内部函数访问外部函数的作用域

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()  
outer()  
}
```



1.5 闭包

闭包作用：实现数据的私有，避免全局污染，外层函数也可以访问里层函数变量

比如，我们要做个统计函数调用次数，函数调用一次，就++

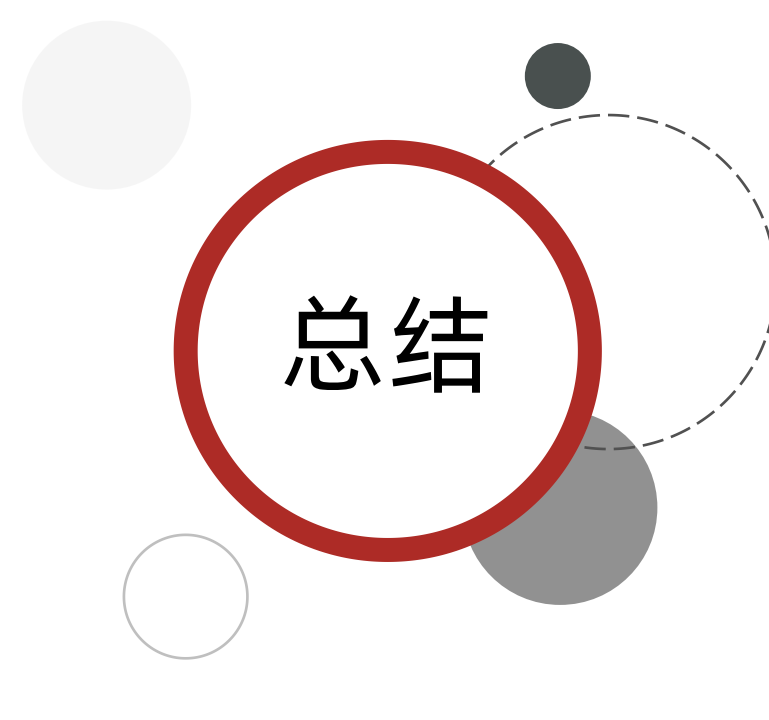
```
let count = 1
function fn() {
  count++
  console.log(`函数被调用${count}次`)
}
fn() // 2
fn() // 3
```



```
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
```

但是，这个count 是个全局变量，很容易被修改

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?

- 作用：实现数据私有，防止全局变量污染，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

01

作用域

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

1.6 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）

说明：

1. 变量提升出现在当前作用域的最前面
2. 提升时，只提升变量声明，不提升变量赋值
3. let/const 声明的变量不存在变量提升
4. 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>
```

注意事项

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。



总结

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

- ◆ 作用域
- ◆ 函数进阶
- ◆ 解构赋值
- ◆ 综合案例



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

函数提升与变量提升比较类似。

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

说明：

1. 函数提升提升到当前作用域最前面
2. 函数提升只提升声明，不提升调用
3. 函数表达式不存在提升的现象
4. 函数提升能够使函数的声明调用更灵活

```
// 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...')
}
```



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. arguments对象（了解）
2. 剩余参数（重点）

2.2 函数参数

1. arguments对象（了解）

`arguments` 是函数内部内置的对象(伪数组)，它包含了调用函数时传入的**所有实参**

使用场景：

写一个求和函数，不管多少实参都可以求结果。 问题是形参怎么写？

```
sum(1, 2)
sum(1, 2, 3)
sum(1, 2, 3, 4)
```

```
function sum(形参怎么写) {
    // 业务代码
}
```

2.2 函数参数

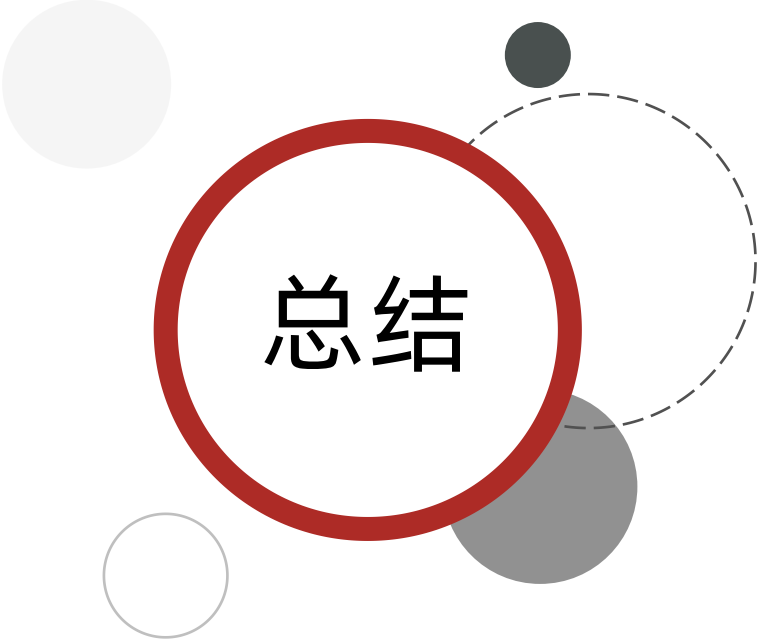
1. arguments对象（了解）

`arguments` 是函数内部内置的对象(伪数组)，它包含了调用函数时传入的所有实参

```
// arguments对象获取所有实参
function sum() {
  // 1. arguments 只存在于函数中 伪数组
  // 2. arguments 可以得到传递过来的所有实参 [1, 2]
  // console.log(arguments)
  let sum = 0
  for (let i = 0; i < arguments.length; i++) {
    sum += arguments[i]
  }
  console.log(sum)
}
sum(1, 2)
sum(1, 2, 3)
sum(1, 2, 3, 4)
// console.log(arguments) 外面无法使用
```

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



总结

1. 当不确定传递多少个实参的时候，我们怎么办?
 - `arguments` 对象，它可以得到传递过来的所有实参
2. `arguments`是什么?
 - 对象，表现形式是伪数组
 - 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. arguments对象（了解）
2. 剩余参数（重点）

2.2 函数参数

2. 剩余参数（重点）

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

简单理解: 用于获取多余的实参，并形成真数组

使用场景:

也可以解决形参和实参个数不匹配的问题

```
function sum(...other) {  
  // 可以得到 [1, 2, 3, 4]  
  console.log(other)  
}  
sum(1, 2, 3, 4)
```



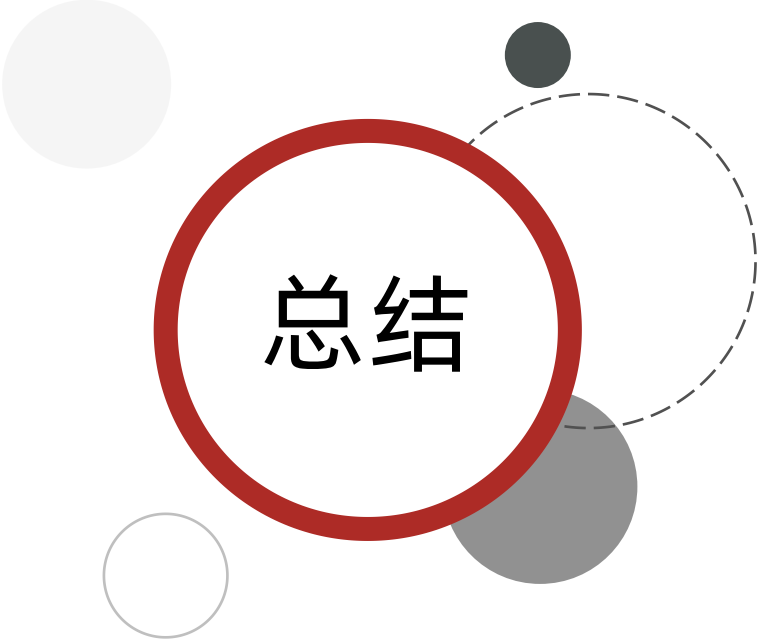
那和arguments 有什么不同吗?

2.2 函数参数

2. 剩余参数和arguments区别

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**
3. 箭头函数不支持arguments，但是可以使用剩余参数

开发中，还是提倡多使用 **剩余参数**



总结

1. 剩余参数主要的使用场景是?
 - 用于获取多余的实参，返回一个真数组
2. 剩余参数和arguments区别是什么？开发中提倡使用哪一个？
 - arguments 是伪数组
 - 剩余参数是真数组
 - 箭头函数不支持arguments，但是可以使用剩余参数
 - 开发中使用剩余参数想必也是极好的

展开运算符

展开运算符(...), 将一个数组/对象进行展开

语法:

1. 不会修改原数组

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

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，把多个元素收集起来生成一个真数组（凝聚）

展开运算符：将数组展开成各个元素（拆散）

```
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（拆散）
- 剩余参数 在函数内部使用，把多个元素收集起来生成一个真数组（凝聚）

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

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



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）

箭头函数比函数表达式更简洁的一种写法

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，写法更简单

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）

用法细节：

1. 当箭头函数只有一个参数时，可以省略参数的小括号，其余个数不能省略（没有参数也需要写小括号）
2. 当箭头函数的函数体只有一句代码 可以省略函数体大括号，这句代码就是返回值（可以不用写return）
3. 如果返回的是个对象，则需要把对象用小括号包裹
4. 箭头函数里面没有arguments，但是有剩余参数

```
const sum = x => {  
  console.log(x + x)  
}
```

```
const sum = x => x + x  
console.log(sum(5))
```

```
const fn = () => ({ age: 18 })  
console.log(fn())
```

```
const sum = (...other) => {  
  // console.log(arguments) // arguments is not defined  
  console.log(other)  
}  
sum(1, 2)
```

2.3 箭头函数（重要）

2. 箭头函数 this

以前函数中的this指向是根据如何调用来确定的。简单理解就是this指向调用者

箭头函数本身没有this, 它只会沿用上一层作用域的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
})
```

```
})
```

```
console.log(this) // 此处为window
```

2.3 箭头函数（重要）

2. 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】

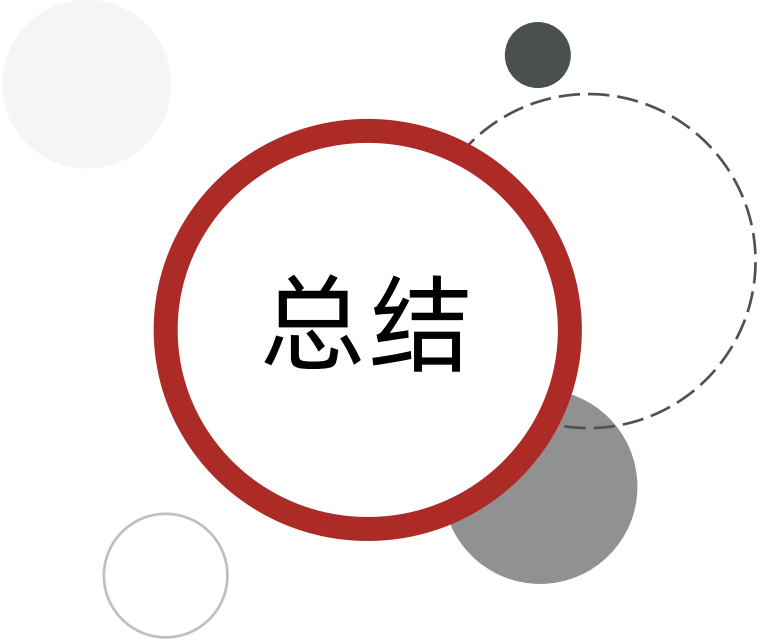
1. 事件回调函数使用箭头函数时，this 为全局的 window，不太推荐使用箭头函数
2. 定时器里面的如果需要this，可以使用箭头函数

```
<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>
```

```
</script>
))
console.log(this)
```



总结

1. 箭头函数里面有this吗?

- 箭头函数本身没有this,它只会沿用上一层作用域的this

2. 我们如何选择用不用箭头函数呢?

- 根据需求来选择是否需要
- dom元素注册事件，需要用this，就不要箭头函数
- 定时器里面需要用到this，就需要箭头函数
- 但是要明白箭头函数的this指向谁

2.4 ES6中对象属性和方法的简写

1. 在对象中，如果属性名和属性值一致，可以简写只写属性名即可

```
const uname = 'andy'  
const age = 18  
const obj = {  
  uname: uname,  
  age: age  
}  
console.log(obj)
```



```
const uname = 'andy'  
const age = 18  
const obj = {  
  uname, // 只写属性名即可  
  age  
}  
console.log(obj)
```

2. 在对象中，方法（函数）可以简写

```
const obj = {  
  sayHi: function() {  
    console.log('hi~')  
  }  
}  
console.log(obj)
```



```
const obj = {  
  sayHi() { // 函数简写  
    console.log('hi~')  
  }  
}  
console.log(obj)
```



目录

Contents

- ◆ 作用域
- ◆ 函数进阶
- ◆ 解构赋值
- ◆ 综合案例

03

解构赋值

- 数组解构
- 对象解构

3. 解构赋值

解构赋值：可以将数组中的值或对象的属性取出，赋值给其他变量

解构：其实就是把一个事物的结构进行拆解

使用场景：

```
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) // 平均值 80
```

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

解构写法

3.1 数组解构

基本语法:

1. 右侧数组的**值**将被**赋值**给左侧的**变量**
2. 变量的顺序对应数组值的位置**依次**进行赋值操作

变量

值

```
// 同时将数组值 1 2 3 依次赋值给变量 a b c
const [a, b, c] = [1, 2, 3]
console.log(a) // 1
console.log(b) // 2
console.log(c) // 3
```

```
console.log(c) // 3
```

3.1 数组解构

典型应用：交换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

3.1 数组解构

注意： js 前面必须加分号情况

1. 小括号开头

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

2. 中括号开头

```
// 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]
```

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

总结

```
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
```

1. 解构赋值有什么作用？

- 可以将数组中的值或对象的属性取出，赋值给其他变量

2. Js 前面有哪两种情况需要加分号的？

- 小括号开头
- 中括号开头

```
(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：变量和值不匹配的情况

2. 变量多值少的情况：

```
// 变量多, 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
```

变量的数量大于值数量时，多余的变量将被赋值为 undefined

3.1 数组解构

解构赋值：可以将数组中的值或对象的属性取出，赋值给其他变量

2. 防止有undefined传递值的情况，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
```

3.1 数组解构

解构赋值：可以将数组中的值或对象的属性取出，赋值给其他变量

3. 变量少值多的情况：

```
// 变量少, 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
console.log(c) // 格力
```

3.1 数组解构

解构赋值：可以将数组中的值或对象的属性取出，赋值给其他变量

4. 利用剩余参数解决变量少值多的情况：

```
// 利用剩余参数 变量少, 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']
```

3.1 数组解构

解构赋值：可以将数组中的值或对象的属性取出，赋值给其他变量

5. 按需导入，忽略某些值：

```
// 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
```

```
console.log(q) // 格力
```

3.1 数组解构

解构赋值：可以将数组中的值或对象的属性取出，赋值给其他变量

6. 支持多维数组的解构：

```
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
```

3. 解构赋值

解构赋值：可以将数组中的值或对象的属性取出，赋值给其他变量

分为：

- 数组解构
- 对象解构

3.2 对象解构

对象解构赋值：可以将对象的属性取出，赋值给其他变量

使用场景：

以前使用对象属性提取比较麻烦

有了对象解构赋值语法就简洁了很多

```
const user = {  
  username: '小明',  
  age: 18,  
  gender: '男'  
}
```

```
// 使用属性  
user.username  
user.age  
user.gender
```

```
// 对象解构赋值  
const user = {  
  username: '小明',  
  age: 18,  
  gender: '男'  
}
```

```
const { username, age, gender } = user  
// 使用属性  
username // 小明  
age // 18  
gender // 男
```

3.2 对象解构

1. 基本语法:

右侧对象的属性值将被赋值给左侧的变量

注意:

- 对象的属性名一定要和变量名相同
- 变量名如果没有和对象属性名相同的则默认是 `undefined`
- 注意解构的变量名不要和外面的变量名冲突否则报错

```
// 对象解构赋值
const user = {
  username: '小明',
  age: 18,
  gender: '男'
}
const { username, age, gender } = user
// 使用属性
console.log(username) // 小明
console.log(age)      // 18
console.log(gender)   // 男
```

```
console.log(gender) // 男
console.log(age)     // 18
console.log(username) // 小明
```

 练习

独立完成对象数组解构赋值

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

3.2 对象解构

2. 更改解构变量名:

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
```

格式：变量名：新变量名

3.2 对象解构

3. 对象数组解构

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

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

 练习**独立完成对象数组解构赋值**

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

解构赋值：可以将数组中的值或对象的属性取出，赋值给其他变量

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  
console.log(age) // 6
```

3.2 对象解构

解构赋值：可以将数组中的值或对象的属性取出，赋值给其他变量

3. 多级对象解构：

```
const person = [  
  {  
    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

解构赋值：可以将数组中的值或对象的属性取出，赋值给其他变量

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)
```



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



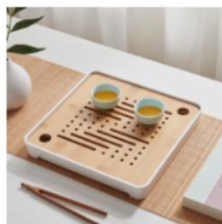
称心如意手摇咖啡磨豆机
咖啡豆研磨机

¥289.00



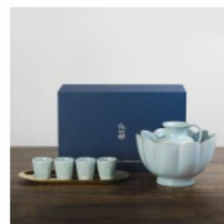
日式黑陶功夫茶组双侧
把茶具礼盒装

¥288.00



竹制干泡茶盘正方形沥
水茶台品茶盘

¥109.00



古法温酒汝瓷酒具套装
白酒杯莲花温酒器

¥488.00



大师监制龙泉青瓷茶叶
罐

¥139.00



与众不同的口感汝瓷白
酒杯套组1壶4杯

¥108.00



手工吹制更厚实白酒杯
壶套装6壶6杯

¥99.00



德国百年工艺高端水晶
玻璃红酒杯2支装

¥139.00

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：拿到数据，利用 **map + join 字符串拼接** 生成结构添加到页面中

②：填充数据的时候使用 **解构赋值**



称心如意手摇咖啡磨豆机
机咖啡豆研磨机

¥289.00



日式黑陶功夫茶组双侧
把茶具礼盒装

¥288.00



竹制干泡茶盘正方形沥
水茶台品茶盘

¥109.00



古法温酒汝瓷酒具套装
白酒杯莲花温酒器

¥488.00



大师监制龙泉青瓷茶叶
罐

¥139.00



与众不同的口感汝瓷白
酒杯套组1壶4杯

¥108.00



手工吹制更厚实白酒杯
壶套装6壶6杯

¥99.00



德国百年工艺高端水晶
玻璃红酒杯2支装

¥139.00



目录

Contents

- ◆ 作用域
- ◆ 函数进阶
- ◆ 解构赋值
- ◆ 综合案例

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

0-100元 100-300元 300元以上 全部区间



手工吹制更厚实白酒杯
壶套装6壶6杯

¥99.00

案例

商品列表价格筛选

0-100元

100-300元

300元以上

全部区间



称心如意手摇咖啡磨豆机
¥289.00



日式黑陶功夫茶具组双侧把茶具礼盒装
¥288.00



竹制干泡茶盘正方形沥水茶台品茶盘
¥109.00



古法温酒汝瓷酒具套装白酒杯莲花温酒器
¥488.00



大师监制龙泉青瓷茶叶罐
¥139.00



与众不同的口感汝瓷白酒杯套组1壶4杯
¥108.00



手工吹制更厚实白酒杯壶套装6壶6杯
¥100.00



德国百年工艺高端水晶玻璃红酒杯2支装
¥139.00

渲染业务

筛选业务 (filter)

步骤

商品列表价格筛选

渲染业务

业务分析：页面初始渲染

- ①：封装 render 渲染函数
- ②：利用 数组 `map + join` 方法渲染页面
- ③：后期筛选业务，本次要求渲染函数增加参数，传递不同的数组

案例

商品列表价格筛选

0-100元

100-300元

300元以上

全部区间



称心如意手摇咖啡磨豆机
¥289.00



日式黑陶功夫茶具组双侧把茶具礼盒装
¥288.00



竹制干泡茶盘正方形沥水茶台品茶盘
¥109.00



古法温酒汝瓷酒具套装
白酒杯莲花温酒器
¥488.00



大师监制龙泉青瓷茶叶罐
¥139.00



与众不同的口感汝瓷白酒杯套组1壶4杯
¥108.00



手工吹制更厚实白酒杯壶套装6壶6杯
¥100.00

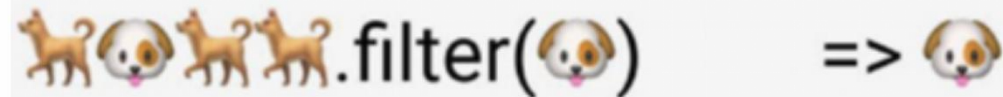


德国百年工艺高端水晶玻璃红酒杯2支装
¥139.00

渲染业务

筛选业务 (filter)

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，不影响原数组
- 语法：

```
const newArr = arr.filter(function (element, index) {  
  return 筛选条件  
})
```

- `element` 是数组元素
- `index` 是数组元素的索引号

案例

商品列表价格筛选

0-100元

100-300元

300元以上

全部区间



称心如意手摇咖啡磨豆机
¥289.00



日式黑陶功夫茶组双侧把茶具礼盒装
¥288.00



竹制干泡茶盘正方形沥水茶台品茶盘
¥109.00



古法温酒汝瓷酒具套装
白酒杯莲花温酒器
¥488.00



大师监制龙泉青瓷茶叶罐
¥139.00



与众不同的口感汝瓷白酒杯套组1壶4杯
¥108.00



手工吹制更厚实白酒杯壶套装6壶6杯
¥100.00



德国百年工艺高端水晶玻璃红酒杯2支装
¥139.00

渲染业务

筛选业务

 步骤

商品列表价格筛选

筛选业务

业务2：点击链接，筛选不同商品

(1) 点击采取事件委托方式

- 先拿到当前点击的序号（利用自定义属性 `data-index`）
- 因为里面事件对象用了多次，可以使用对象解构，把事件对象解构

(2) 筛选的核心思路：

根据不同序号 利用 `filter` 筛选数组，符合条件的新数组传递给渲染函数重新渲染页面即可

- 全部区间不需要筛选，直接把 `goodList` 渲染即可

05

JS垃圾回收机制-算法说明

拓展-JS垃圾回收机制-算法说明

堆栈空间分配区别：

1. 栈（操作系统）：由操作系统自动分配释放函数的参数值、局部变量等，基本数据类型放到栈里面。
2. 堆（操作系统）：一般由程序员分配释放，若程序员不释放，由垃圾回收机制回收。复杂数据类型放到堆里面。

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

拓展-JS垃圾回收机制-算法说明

- 引用计数

IE采用的引用计数算法，定义“内存不再使用”，就是看一个对象是否有指向它的引用，没有引用了就回收对象
算法：

1. 跟踪记录被引用的次数
2. 如果被引用了一次，那么就记录次数1, 多次引用会累加 ++
3. 如果减少一个引用就减1 --
4. 如果引用次数是0，则释放内存

拓展-JS垃圾回收机制-算法说明

- 引用计数

```
const arr = [1, 2, 3, 4]
arr = null
```

```
let person = {
  age: 18,
  name: '佩奇'
}
let 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。

拓展-JS垃圾回收机制-算法说明

- 引用计数

但它却存在一个致命的问题：**嵌套引用**（循环引用）

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。

```
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

拓展-JS垃圾回收机制-算法说明

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。

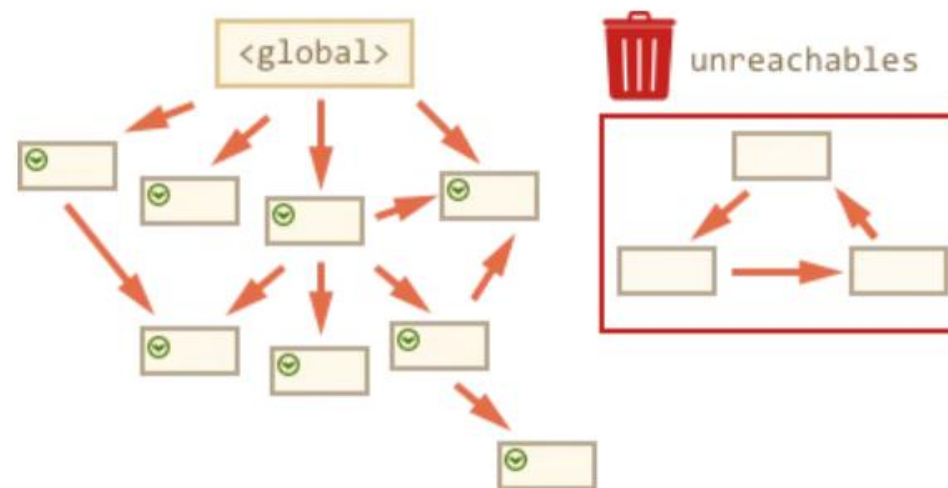
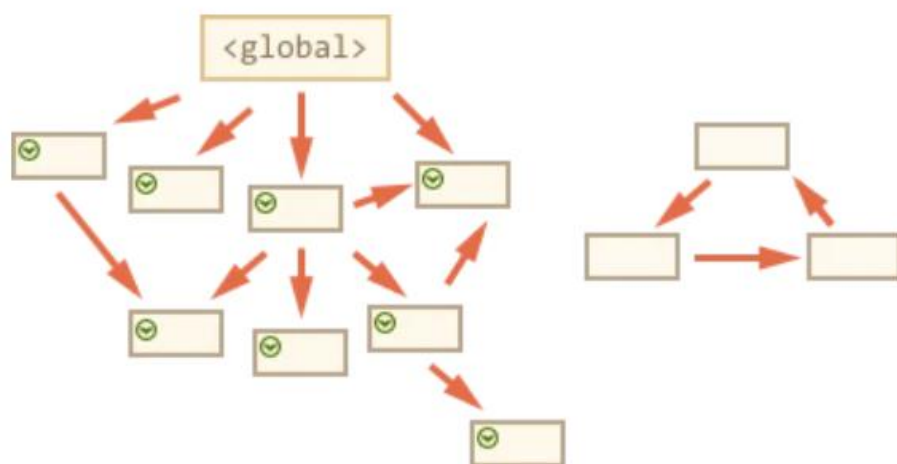
核心：

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**。

拓展-JS垃圾回收机制-算法说明

- 标记清除法

标记所有的引用

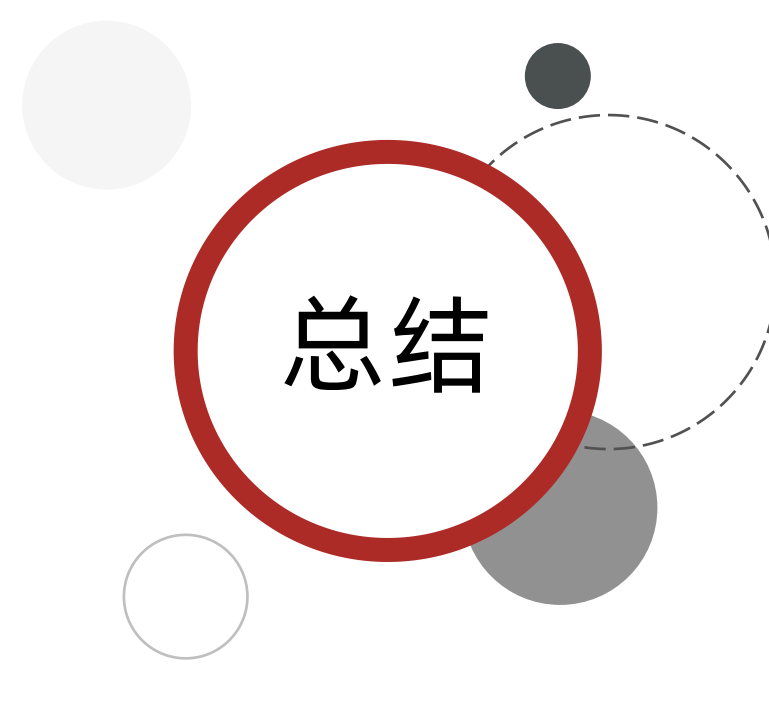


拓展-JS垃圾回收机制-算法说明

- 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



总结

1. 标记清除法核心思路是什么?

- 从**根部**扫描对象，能查找到的就是使用的，查找不到的就要回收



传智教育旗下高端IT教育品牌