JavaScript 基础第四天

函数

黑马程序员
www.itheima.com
传智教育旗下
高端IT教育品牌

目录

Contents

- 函数
- 综合案例

01

函数

- 函数基本使用
- 函数整体感知
- 函数传参
- 函数返回值
- 作用域
- 匿名函数

函数-Function

函数：是可以被重复使用的代码块

作用：函数可以把具有相同或相似逻辑的代码“包裹”起来，这么做的优势是有利于代码复用

```
for (let i = 1; i <= 9; i++) {
  for (let j = 1; j <= i; j++) {
    document.write(`<span>${j} x ${i} = ${j * i}</span>`)
  }
  document.write('<br>')
}
```

页面不同地方需要使用三次乘法表

```
for (let i = 1; i <= 9; i++) {
  for (let j = 1; j <= i; j++) {
    document.write(`<span>${j} x ${i} = ${j * i}</span>`)
  }
  document.write('<br>')
}
for (let i = 1; i <= 9; i++) {
  for (let j = 1; j <= i; j++) {
    document.write(`<span>${j} x ${i} = ${j * i}</span>`)
  }
  document.write('<br>')
}
for (let i = 1; i <= 9; i++) {
  for (let j = 1; j <= i; j++) {
    document.write(`<span>${j} x ${i} = ${j * i}</span>`)
  }
  document.write('<br>')
}
```

封装

```
function export99() {
  for (let i = 1; i <= 9; i++) {
    for (let j = 1; j <= i; j++) {
      document.write(`<span>${j} x ${i} = ${j * i}</span>`)
    }
    document.write('<br>')
  }
}
export99() // 第一次输出乘法表
export99() // 第二次输出乘法表
export99() // 第三次输出乘法表
```

计算逻辑封装到函数，代码复用

函数基本使用

1. 定义函数

利用关键字 function 定义函数 （声明函数）

```
function 函数名() {
    函数体
}
```

```
function sum() {
  // 函数体
}
```

函数名命名跟变量一致
采用小驼峰命名法

函数名经常采用 动词

2. 调用函数

定义一个函数并不会自动执行它，需要调用函数

```
函数名()  // 小括号是调用函数
```

```
sum() // 调用 sum 函数
```

函数可以多次调用，每次调用都会重新执行函数体里面代码

alert() parseInt() 名字后面跟小括号的本质都是函数的调用

总结

1. 函数是什么？有什么好处？
   - 可以被重复使用的代码块
   - 代码复用
2. 函数用哪个关键字声明？不调用会执行吗？
   - function
   - 函数不调用自己不执行，调用方式：函数名()
3. 函数的复用代码和循环重复代码有什么不同？
   - 循环代码写完立即执行，不能很方便控制执行位置
   - 需要的时候来调用，可重复调用

练习

函数课堂练习

需求：

1. 把99乘法表封装到函数里面，重复调用3次

案例

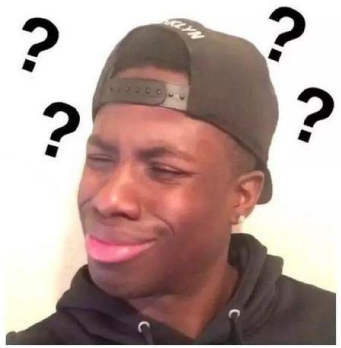
函数小练习

需求：

1. 封装一个函数，计算两个数的和
2. 封装一个函数，计算1-100之间的累加和

灵魂拷问：

这些函数有什么缺陷？



01

函数

- 函数基本使用
- 函数整体认知
- 函数参数
- 函数返回值
- 作用域
- 匿名函数

1.2 函数整体认知

问题：这样的函数只能求 10 + 20， 而且结果只能函数内部打印，这个函数功能局限非常大

```
function sum() {
  let x = 10
  let y = 20
  console.log(x + y)
}
sum()
```

函数：

1. 传递数据给函数
2. 函数内部处理
3. 返回一个结果（值）给调用者

用户决定传递什么原料

返回不同水果汁给用户

1.2 函数整体认知

- 函数语法

```
function sum(参数1, 参数2...) {
  return 结果
}
console.log(sum(1, 2)) // 输出函数返回的结果
```

用户决定传递什么原料

返回不同水果汁给用户

- 说明：
  - 函数参数，如果有多个则用逗号分隔，用于接受传递过来的数据
  - return 关键字可以把结果返回给调用者

练习

- **函数封装求和**

需求：采取函数封装的形式：输入2个数，计算两者的和，打印到页面中

01

函数

- 函数基本使用
- 函数整体认知
- 函数参数
- 函数返回值
- 作用域
- 匿名函数

1.3 函数参数

形参 形参

```
function sum(x, y) {
  return x + y
}
console.log(sum(1, 2))  // 输出函数返回的结果  3
```

x = 1 y = 2

实参 实参

- 形参：声明函数时小括号里的叫形参（形式上的参数）
- 实参：调用函数时小括号里的叫实参（实际上的参数）
- 执行过程： 会把实参的数据传递给形参，从而提供给函数内部使用，我们可以把形参理解为变量
- 我们曾经使用过的 alert('打印')，parseInt('11px')，Number('11') 本质上都是函数调用的传参

1.3 函数参数

- 在Javascript中 实参的个数和形参的个数可以不一致
  - 如果形参过多 会自动填上undefined
  - 如果实参过多 那么多余的实参会被忽略

建议：开发中尽量保持形参和实参个数一致

总结

1. 函数中实参在哪里？形参在哪里？
   - 函数调用时，小括号里面的是实参（实际上的参数）
   - 函数声明时，小括号里面的是形参（形式上的参数）
   - 执行过程：实参会传递给形参
2. 实参的个数和形参的个数不一致怎么执行的？
   - 如果形参过多 会自动填上undefined
   - 如果实参过多 那么多余的实参会被忽略
   - 开发中尽量形参和实参个数保持统一

形参 形参

```
function sum(x, y) {
  return x + y
}
console.log(sum(1, 2))  // 输出函数返回的结果  3
```

x = 1 y = 2

实参 实参

逻辑中断

逻辑中断： 存在于逻辑运算符 && 和 || 中，左边如果满足一定条件会中断代码执行，也称为逻辑短路

解释：

false && anything // 逻辑与左边false则中断，如果左边为true，则返回右边代码的值

true || anything // 逻辑或左边true则中断，如果左边为false，则返回右边代码的值

```
function sum(x, y) {
  return x + y
}
console.log(sum(1, 2))  // 3
console.log(sum()) // NaN ???
```

```
function sum(x, y) {
  x = x || 0
  y = y || 0
  return x + y
}
console.log(sum(1, 2))  // 3
console.log(sum()) // 0
```

1.3 函数参数 - 默认参数

默认参数：可以给形参设置默认值

```
function sum(x, y) {
  x = x || 0
  y = y || 0
  return x + y
}
console.log(sum(1, 2))  // 3
console.log(sum()) // 0
```

```
function sum(x = 0, y = 0) {
  return x + y
}
console.log(sum(1, 2))  // 3
console.log(sum()) // 0
```

说明：这个默认值只会在缺少实参传递或者实参是undefined才会被执行

默认参数和逻辑中断使用场景区别：

- 默认参数主要处理函数形参（处理参数要比逻辑中断更简单）
- 逻辑中断除了参数还可以处理更多的需求

```
(function flexible(window, document) {
  // window.devicePixelRatio 获取当前设备的 dpr
  // 获取不到，则默认取值为1
  // 移动端 获取为2，则执行2
  var dpr = window.devicePixelRatio || 1
}(window, document))
```

案例

函数封装-数组求和

需求：用户给不同数组（里面是数字型数据），求数组和并且返回

分析：

①： 封装一个求和函数，传递过去的参数是一个数组

②： 函数内部遍历数组求和，返回这个结果

01

函数

- 函数基本使用
- 函数整体感知
- 函数参数
- 函数返回值
- 作用域
- 匿名函数

1.4 函数返回值 return

- 返回值：把处理结果返回给调用者
- 其实我们前面已经接触了很多的函数具备返回值：

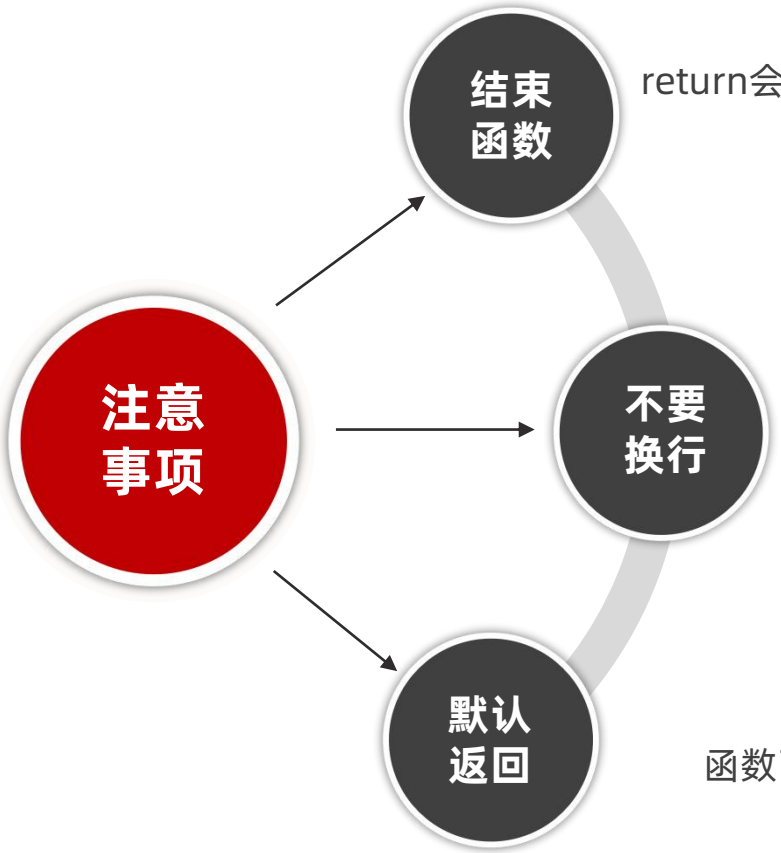
```
let result = prompt('请输入你的年龄?')
let result2 = parseInt('111')
```

- 只是这些函数是JS底层内置的. 我们直接就可以使用
- 当然有些函数，则没有返回值

```
alert('我是弹框，不需要返回值')
```

- 所以要根据需求，来设定需不需要返回值

1.4 函数返回值 return



return会立即结束当前函数，所以后面代码不会再被执行

在 return 关键字和被返回的表达式之间不允许使用换行符，否则内部执行相当于会自动补充分号

函数可以没有 return，这种情况函数默认返回值为 undefined

练习

- **函数封装练习（重点）**

1. 封装函数，求任意数组中的最大值并返回这个最大值
2. 封装函数，求任意数组中的最小值并返回这个最小值
3. 封装函数，判断数组是否存在某个元素，如果有则返回true，否则返回 false

练习

- **函数封装练习（重点）**

封装函数，判断数组是否存在某个元素，如果有则返回true，否则返回 false

思路：

①：函数封装 some，传递两个参数：元素和数组

②：可以设置一个初始变量 flag 为 false

③：如果能找到，则修改 falg 值为 true，则中断循环，找不到则不修改 flag

④：返回 falg

练习 • 函数封装练习（重点）

封装函数，查找元素在数组中的索引。

如果找到该元素（第一个元素即可），则返回该元素的索引号，找不到该元素则返回-1

思路：

①：函数封装 findIndex，传递两个参数：元素和数组

②：可以设置一个初始变量 index 为 -1

③：如果能找到，则修改 index 值为 当前索引号，则中断循环，找不到则不修改 index

④：返回 index

练习

- **函数封装练习（重点）**

1. 封装函数，求任意数组中的最大值并返回这个最大值

2. 封装函数，求任意数组中的最小值并返回这个最小值

3. 封装函数，判断数组是否存在某个元素，如果有则返回true，否则返回 false

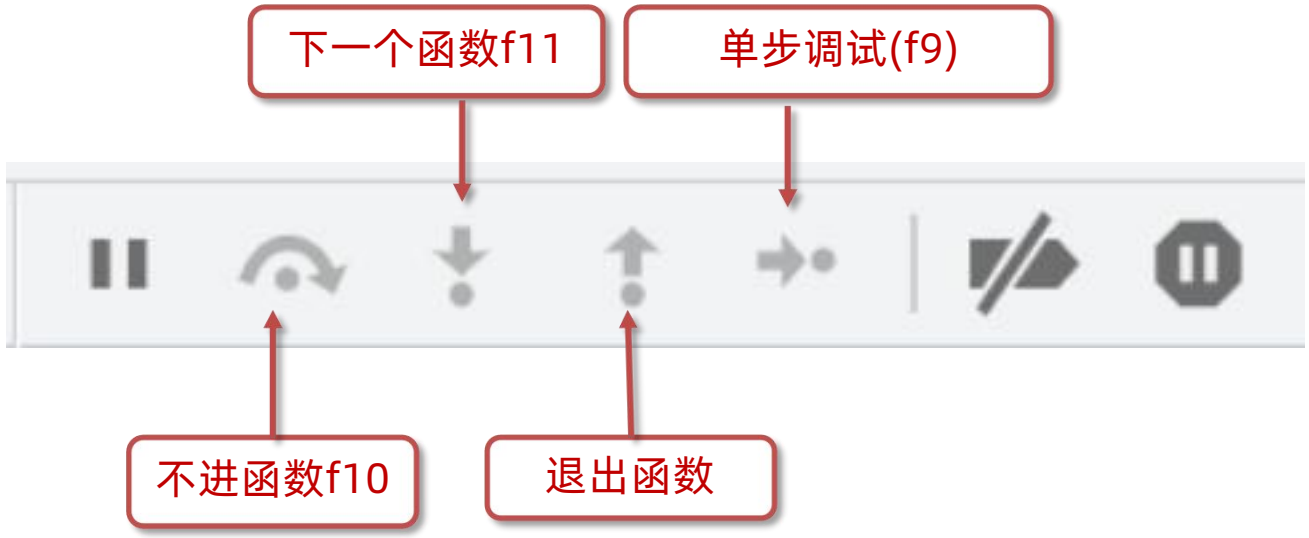
4. 封装函数，查找数组给定元素的索引，如果找到该元素（第一个元素即可），则返回该元素的索引号，否则返回-1

断点调试：

进入函数内部看执行过程 F11

断点调试 - 进入函数内部

- F11 可以进入函数内部调试



01

函数

- 函数基本使用
- 函数整体感知
- 函数参数
- 函数返回值
- 作用域
- 匿名函数

1.5 作用域（Scope）

作用域（scope）： 变量或者值在代码中可用性的范围

作用： 作用域的使用提高了程序逻辑的局部性，增强了程序的可靠性，减少了名字冲突。

```
for (let i = 0; i < 3; i++) {
    document.write('怎么回事呢？ ')
}
console.log(i)
```

```
function fun() {
    let num2 = 20
}
fun()
console.log(num2)
```

```
▶Uncaught ReferenceError: i is not defined
```

```
▶Uncaught ReferenceError: num2 is not defined
```

1.5 作用域

全局作用域

局部作用域

全局有效

作用于所有代码执行的环境(整个 script 标签内部)或者一个独立的 js 文件

局部有效

1. 函数作用域。作用于函数内的代码环境
2. 块级作用域。{ } 大括号内部

1.5 作用域

在JavaScript中，根据作用域的不同，变量可以分为：

全局变量

局部变量

全局作用域的变量

全局变量在任何区域都可以访问和修改

局部作用域内变量

局部变量只能在当前局部内部访问和修改

注意事项

- 如果函数内部，变量没有声明直接赋值，也当全局变量看，但是强烈不推荐
- 函数内部的形参可以看做是局部变量

```
for (let i = 0; i < 3; i++) {
    document.write('怎么回事呢？ ')
}
console.log(i)
```

总结

1. JS 中作用域分为哪两种？
   - 全局作用域。script 标签或者js文件
   - 局部作用域。函数作用域和块级作用域 {}
2. 根据作用域不同，变量分为哪两种？
   - 全局变量
   - 局部变量
3. 两个注意事项：
   - 函数内部不声明直接赋值的变量当全局变量，不提倡
   - 函数内部的形参可以当做局部变量看

```
function fun() {
    let num2 = 20
}
fun()
console.log(num2)
```

变量的访问原则

- 在不同作用域下，可能存在变量命名冲突的情况，到底该执行谁呢？

```
let num = 10
function fn() {
  let num = 20
  console.log(num)
}
fn()
```

- **访问原则**：在能够访问到的情况下先局部， 局部没有再找全局，总结： 就近原则

练习 • 变量访问原则

```
function f1() {
  let num = 123
  function f2() {
    console.log(num)
  }
  f2()
}
let num = 456
f1()
```

```
function f1() {
  let num = 123
  function f2() {
    let num = 0
    console.log(num)
  }
  f2()
}
let num = 456
f1()
```

练习

- **变量访问原则**

```
let a = 1
function fn1() {
  let a = 2
  let b = '22'
  fn2()

  function fn2() {
    let a = 3
    fn3()

    function fn3() {
      let a = 4
      console.log(a) //a的值 ?
      console.log(b) //b的值 ?
    }
  }
}

fn1()
```

总结

1. 变量访问原则是什么？

➢采取就近原则的方式来查找变量最终的值

01

函数

- 函数基本使用
- 函数整体感知
- 函数参数
- 函数返回值
- 作用域
- 匿名函数

1.6 匿名函数

函数

具名函数

- 带有名字的函数
- 声明：function sum() {}
- 调用：fn()

匿名函数

- 没有名字的函数
- function () {}

```
btn.onclick = function() {
    alert('我是匿名函数')
}
```

```
btn.addEventListener(function() {
    alert('弹出')
})
```

1.6 匿名函数

匿名函数两种使用方式：

- 函数表达式
- 立即执行函数

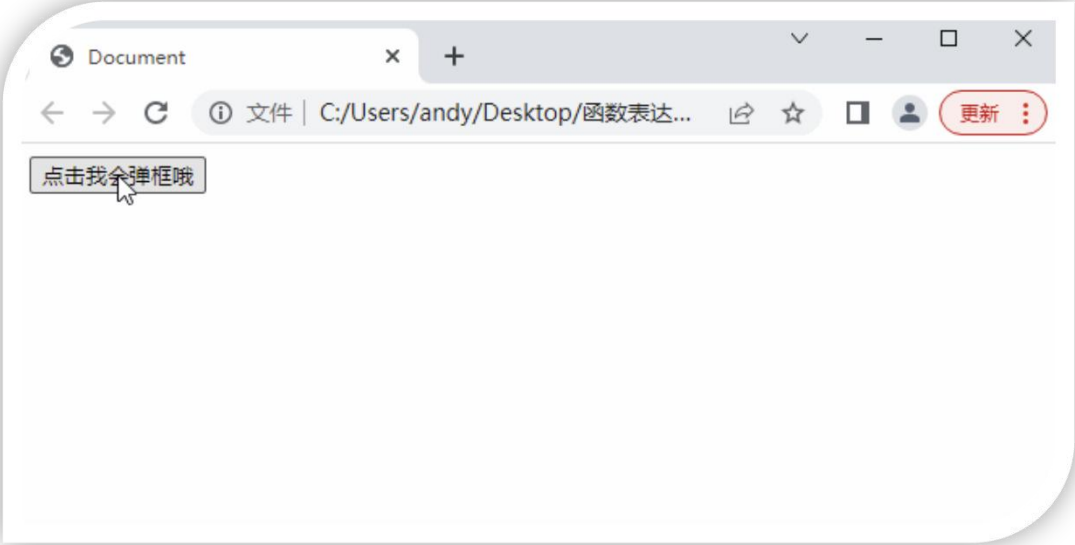
1.6 匿名函数

1. 函数表达式

将匿名函数赋值给一个变量，并且通过变量名称进行调用 我们将这个称为**函数表达式**

```
let fn = function() {}
```

使用场景： 后期web api阶段会使用，目前先认识




```
<body>
    <button>点击我</button>
    <script>
        let btn = document.querySelector('button')
        btn.onclick = function() {
            alert('我是匿名函数')
        }

    </script>
</body>
```

1.6 匿名函数

1. 函数表达式

语法：

```
let fn = function () {
    // 函数体
}
```

调用：

```
fn()  // 函数名()
```

- 其实函数也是一种数据类型
- 函数表达式必须先定义，后使用
- 函数的形参和实参使用跟具名函数一致

1.6 匿名函数

2. 立即执行函数（IIFE）

IIFE (立即执行函数表达式) (Immediately Invoked Function Expression)

语法：

```
// 方式1
(匿名函数)();

// 方式2
(匿名函数());

// 不需要调用立即执行
```

```
// 方式1
(function () { console.log(11) })();

// 方式2
(function () { console.log(11) }());

// 不需要调用，立即执行
```

场景介绍： 避免全局变量之间的污染

注意： 多个立即执行函数要用；隔开，要不然会报错

总结

1. 立即执行函数有什么作用？
   - 避免全局变量之间的污染
2. 立即执行函数需要调用吗？有什么注意事项呢？
   - 无需调用，立即执行，其实本质已经调用了
   - 多个立即执行函数之间用分号隔开

目录

Contents

- 函数
- 综合案例

案例

转换时间案例

需求：用户输入秒数，可以自动转换为时分秒

案例

转换时间案例

需求： 用户输入秒数，可以自动转换为时分秒

分析：

①：用户弹窗输入 总秒数

②：封装函数 getTime，计算时分秒，注意：里面包含数字补0

③：打印输出

计算公式：计算时分秒

小时： h = parseInt(总秒数 / 60 / 60 % 24)

分钟： m = parseInt(总秒数 / 60 % 60)

秒数： s = parseInt(总秒数 % 60)

黑马程序员

www.itheima.com

传智教育旗下高端IT教育品牌