# JavaScript语句、数组、函数

# 表达式和语句

表达式：

- 表达式是一组代码的集合，JavaScript解释器会将其计算出一个结果

```
x = 7
3 + 4
num++
```

- js 语句是以分号结束（可以省略）
- 比如： if语句　　for 循环语句

区别：

表达式计算出一个值，但语句用来执行以使某件事发生(做什么事)

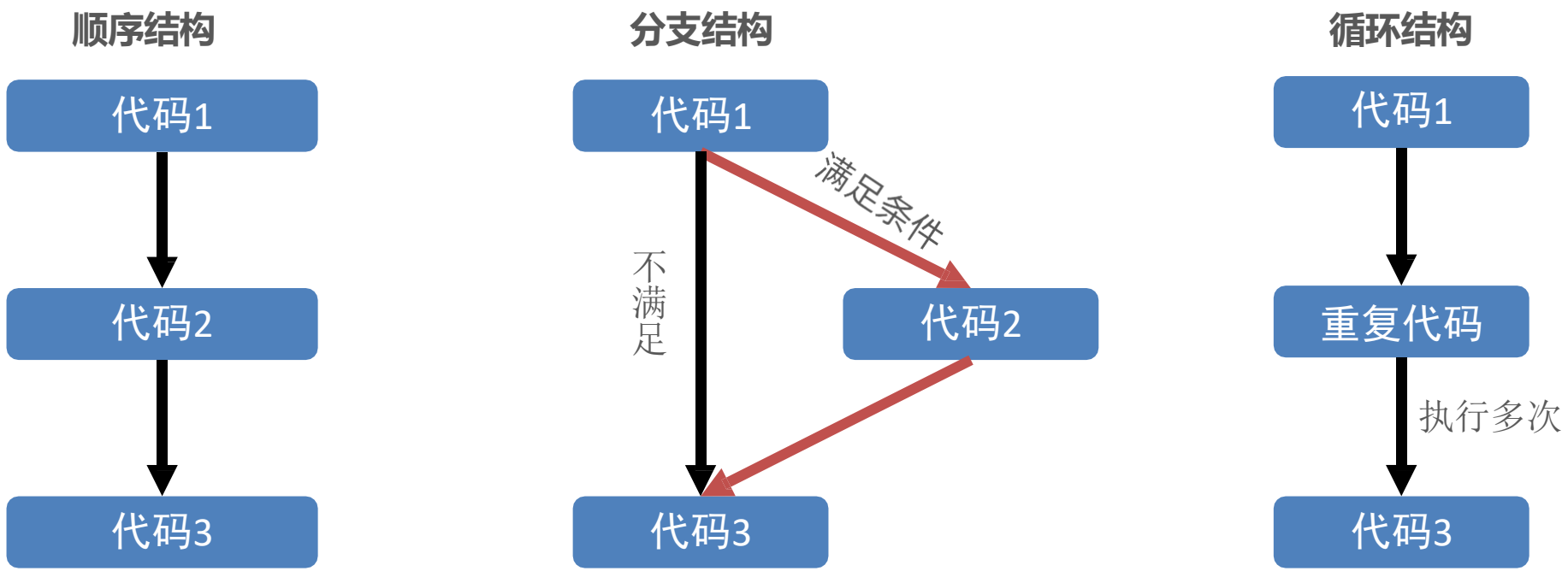
- 表达式　　3 + 4
- 语句alert() 弹出对话框  console.log()控制台打印

其实某些情况，也可以把表达式理解为语句，因为它是在计算结果，也是做事。

# 分支语句

**程序三大流程控制语句**

- 从上往下执行代码的结构，叫顺序结构；
- 根据条件选择执行代码，叫分支结构；
- 某段代码被重复执行，叫循环结构；



# 分支语句

- 分支语句可以让我们有选择性的执行想要的代码
- 分支语句包含：
  - if分支语句
  - 三元运算符
  - switch 语句

分支语句的条件部分可以是变量或者表达式，其他类型按照如下规律转换为Boolean类型。

① String：空字符串为false，所有非空字符串为true；

② Number：0为false，一切非0数字为true；

③ Null/Undefined/NaN：全为false；

④ Object：全为true

# 分支语句

- 双分支if 语法：

```
if (condition) {
  /* 条件为真时运行的代码 */
} else {
  /* 否则，运行其他的代码 */
}
```

```
if((year%4===0&&year%100!==0) || (year%400===0))
{
    alert( ‘闰年’ )
}else{
    alert( ‘平年’ )
}
```

- 让用户输入年份，判断这一年是闰年还是平年并输出
  - 能被4整除但不能被100整除，或者被400整除的年份是闰年，否则都是平年
  - 需要逻辑运算符

# 分支语句

- 多分支if 语法：

```
if (choice === 'sunny') {
  para.textContent = '阳光明媚。穿上短裤吧！去海滩，或公园，吃个冰淇淋。';
} else if (choice === 'rainy') {
  para.textContent = '外面下着雨；带上雨衣和雨伞，不要在外面呆太久。';
} else if (choice === 'snowing') {
  para.textContent = '大雪纷飞，天寒地冻！最好呆在家里喝杯热巧克力，或者去堆个雪人。';
} else if (choice === 'overcast') {
  para.textContent = '虽然没有下雨，但天空灰蒙蒙的，随时都可能变天，所以要带一件雨衣以防万一。';
} else {
  para.textContent = '';
}
```

# 分支语句

- 嵌套 if...else：

```
if (choice === 'sunny') {
  if (temperature < 86) {
    para.textContent = `外面现在是 ${temperature} 华氏度——风和日丽。让我们去海滩或公园，吃个冰激凌。
`;
  } else if (temperature >= 86) {
    para.textContent = `外面现在是 ${temperature} 华氏度——很热！如果你想出去，一定要涂抹一些防晒霜。
`;
  }
}
```

# 分支语句

- 不写嵌套 if...else，可以使用逻辑运算符与、或、非：

```
if (choice === 'sunny' && temperature < 86) {
  para.textContent = '外面现在是' + temperature + ' 华氏度—风和日丽。让我们去海滩或公园，吃个冰激凌。';
} else if (choice === 'sunny' && temperature >= 86) {
  para.textContent = '外面现在是' + temperature + ' 华氏度—很热！如果你想出去，一定要涂抹一些防晒霜。';
}
```

```
if (iceCreamVanOutside || houseStatus === 'on fire') {
  console.log('你需要赶紧离开这间房子。');
} else {
  console.log('或许你应该呆在这里。');
}
```

```
if (!(iceCreamVanOutside || houseStatus === 'on fire')) {
  console.log('或许你应该呆在这里。');
} else {
  console.log('你需要赶紧离开这间房子。);
}
```

# 分支语句

三元运算符

```
条件 ? 满足条件执行的代码 : 不满足条件执行的代码
```

## 例：判断2个数的最大值

需求：用户输入2个数，控制台输出最大的值

分析：

①：用户输入2个数

②：利用三元运算符输出最大值

```
let n1 = prompt()
let n2 = prompt()
let max = Number(n1) > Number(n2) ? Number(n1) : Number(n2)
console.log(max)
```

# 分支语句

## 数字补0

需求：用户输入1个数，如果数字小于10，则前面进行补0， 比如 09　　03 等

分析：利用三元运算符 补 0 计算

```
let num = prompt()
let outputStr = Number(num) < 10 ? '0' + num : num
document.write(outputStr)
```

# 分支语句

三元运算符示例：

```
const select = document.querySelector('select');
const html = document.querySelector('html');
document.body.style.padding = '10px';

function update(bgColor, textColor) {
  html.style.backgroundColor = bgColor;
  html.style.color = textColor;
}

select.addEventListener('change', () => select.value === 'black'
  ? update('black', 'white')
  : update('white', 'black')
);
```

# switch语句

- if...else 语句能够很好地实现条件代码，但是它们不是没有缺点。它们主要适用于只有几个选择的情况，每个都需要相当数量的代码来运行，或条件很复杂的情况（例如多个逻辑运算符）。对于只想将变量设置一系列为特定值的选项或根据条件打印特定语句的情况，语法可能会很麻烦，特别是如果有大量选择的时候。

```
switch (表达式) {
  case 选择1:
    运行这段代码
    break;

  case 选择2:
    否则，运行这段代码
    break;

  // 包含尽可能多的情况

  default:
    实际上，仅仅运行这段代码
}
```

注意事项

1. switch case语句一般用于等值判断,不适合于区间判断
2. switch case一般需要配合break关键字使用，没有break会造成case穿透
3. 当出现case穿透后，只有当执行第一个break语句时，才会退出switch语句。

# switch语句

```
function setWeather() {
  const choice = select.value;

  switch (choice) {
    case 'sunny':
      para.textContent = '阳光明媚。穿上短裤吧！去海滩，或公园，吃个冰淇淋。';
      break;
    case 'rainy':
      para.textContent = '外面下着雨；带上雨衣和雨伞，不要在外面呆太久。';
      break;
    case 'snowing':
      para.textContent = '大雪纷飞，天寒地冻！最好呆在家里喝杯热巧克力，或者去堆个雪人。';
      break;
    case 'overcast':
      para.textContent = '虽然没有下雨，但天空灰蒙蒙的，随时都可能变天，所以要带一件雨衣以防万一。';
      break;
    default:
      para.textContent = '';
  }
}
```

# switch语句

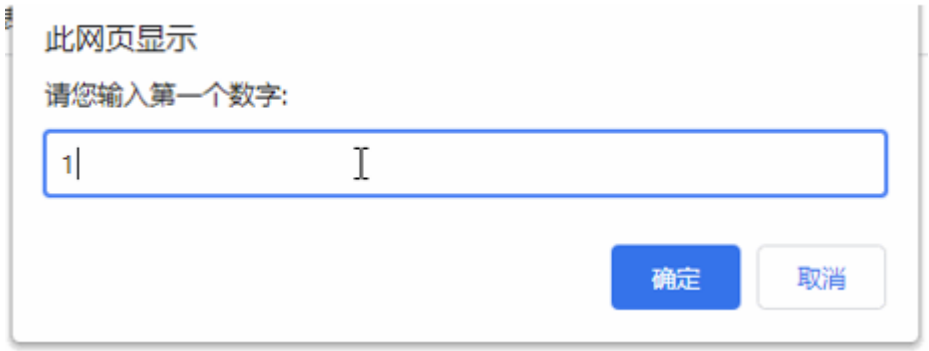
```
<script>
    let styType = prompt('请输入学生类型：')
    switch (styType) {
        case 'underGraduate':
            alert('本科生')
            break
        case 'master':
            alert('硕士研究生')
            break
        case 'ph.D':
            alert('博士研究生')
            break
        default:
            alert('不合理取值')
            break
    }
</script>
```

```
<script>
    let styType = prompt('请输入学生类型：')
    switch (styType) {
        case 'underGraduate':
            alert('本科生')
            break
        case 'master':
        case 'ph.D':
            alert('研究生')
            break
        default:
            alert('不合理取值')
            break
    }
</script>
```

案例

# 简单计算器

需求：用户输入2个数字，然后输入 + - * / 任何一个，可以计算结果




分析：

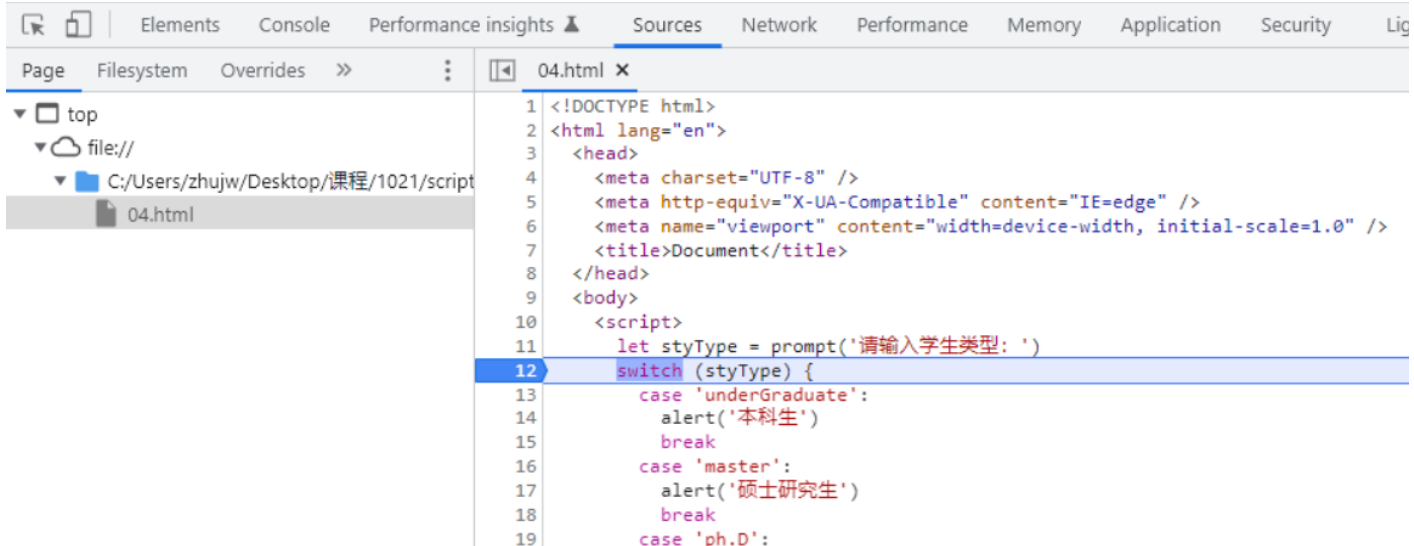
①：用户输入数字

②：用户输入不同算术运算符，可以去执行不同的运算（switch）

# 循环结构—断点调试

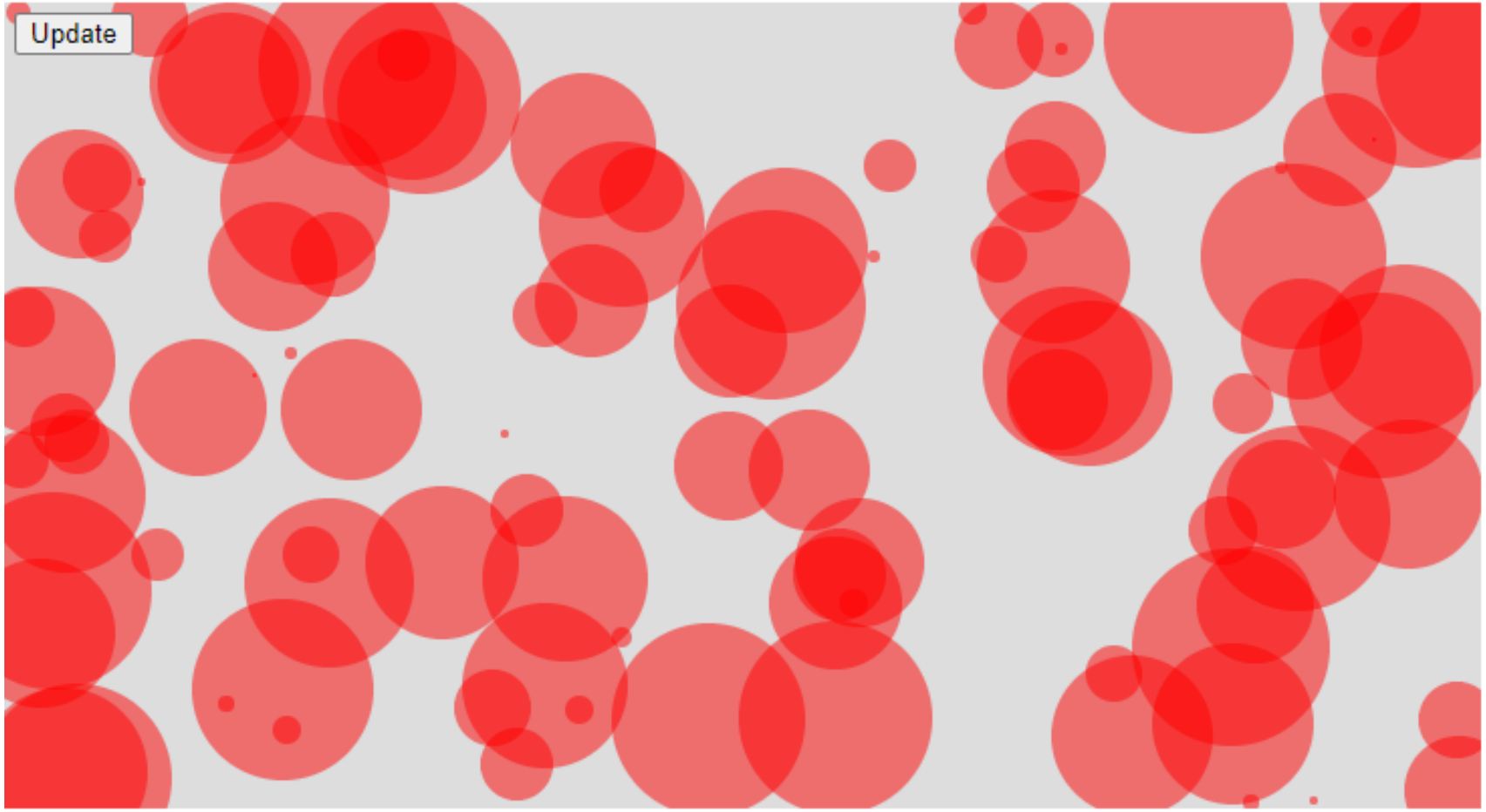
**掌握断点调试方法，学会通过调试检查代码**

- 作用：学习时可以帮助更好的理解代码运行，工作时可以更快找到bug
- 浏览器打开调试界面
  1. 按F12打开开发者工具
  2. 点到sources一栏
  3. 选择代码文件
- 断点：在某句代码上加的标记就叫断点，当程序执行到这句有标记的代码时会暂停下来



# 循环结构

编程语言可以很迅速方便地帮我们完成一些重复性的任务。看一个例子来完美地说明为什么循环是一件好事。假设我们想在<canvas>元素上绘制 100 个随机圆。



# 循环结构—for循环

**Cats数组**

```
<p></p>
<script>
  const cats = ['Bill', 'Jeff', 'Pete', 'Biggles', 'Jasmin'];
  let info = 'My cats are called ';
  const para = document.querySelector('p');

  for (let i = 0; i < cats.length; i++) {
    info += cats[i] + ', ';
  }

  para.textContent = info;

</script>
```

# 循环结构—for循环

如果要在所有迭代完成之前退出循环，可以使用 break 语句。

```
const contacts = ['Chris:2232322', 'Sarah:3453456', 'Bill:7654322', 'Mary:9998769',
'Dianne:9384975'];
const para = document.querySelector('p');
const input = document.querySelector('input');
const btn = document.querySelector('button');

btn.addEventListener('click', function() {
  let searchName = input.value.toLowerCase();
  input.value = '';
  input.focus();
  for (let i = 0; i < contacts.length; i++) {
    let splitContact = contacts[i].split(':');
    if (splitContact[0].toLowerCase() === searchName) {
      para.textContent = splitContact[0] + '\'s number is ' + splitContact[1] + '.';
      break;
    } else if (i === contacts.length - 1) {
      para.textContent = 'Contact not found.';
    }
  }
});
```

# 循环结构—for循环

continue 语句以类似的方式工作，而不是完全跳出循环，而是跳过当前循环而执行下一个循环。

```
var num = input.value;

for (var i = 1; i <= num; i++) {
  var sqRoot = Math.sqrt(i);
  if (Math.floor(sqRoot) !== sqRoot) {
    continue;
  }

  para.textContent += i + ' ';
}
```

使用Math.sqrt(i)找到每个数字的平方根，然后测试平方根是否是一个整数，通过判断当它被向下取整时，它是否与自身相同。

# 循环结构—while 和do while语句

for 不是 JavaScript 中唯一可用的循环类型。

```
initializer
while (exit-condition) {
  // code to run

  final-expression
}
```

释义：

- 跟if语句很像，都要满足小括号里的条件为true才会进入执行代码
- while大括号里代码执行完毕后不会跳出，而是继续回到小括号里判断条件是否满足，若满足又执行大括号里的代码，然后再回到小括号判断条件，直到括号内条件不满足，即跳出

# 循环结构—while 和do while语句

```
<p></p>
<script>
  const cats = ['Bill', 'Jeff', 'Pete', 'Biggles', 'Jasmin'];
  let info = 'My cats are called ';
  const para = document.querySelector('p');

  // for (let i = 0; i < cats.length; i++) {
  //   info += cats[i] + ', ';
  // }
  var i = 0;
  while (i < cats.length) {
    if (i === cats.length - 1) {
      info += 'and ' + cats[i] + '.';
    } else {
      info += cats[i] + ', ';
    }

    i++;
  }

  para.textContent = info;

</script>
```

# 循环结构—while 和do while语句

do...while循环非常类似但在 while 后提供了终止条件：

```
var i = 0;
do {
      if (i === cats.length - 1) {
         info += 'and ' + cats[i] + '.';
      } else {
         info += cats[i] + ', ';
      }

      i++;
} while (i < cats.length);
```

必须保证初始变量是迭代的，那么它才会逐渐地达到退出条件。不然，它将会永远循环下去，要么浏览器会强制终止它，要么它自己会崩溃。这就是无限循环。

# While 练习

需求：使用while循环，页面中打印，可以添加换行效果

**1. 页面输出1-100**

- 核心思路： 利用 i,因为正好和数字对应

**2. 计算从1加到100的总和并输出**

- 核心思路：
  - 声明累加和的变量 sum
  - 每次把 i 加到 sum 里面

**3. 计算1-100之间的所有偶数和**

- 核心思路：
  - 声明累加和的变量 sum
  - 首先利用if语句把 i 里面是偶数筛选出来
  - 把 筛选的 i 加到 sum 里面

# 简易ATM取款机案例

需求：用户可以选择存钱、取钱、查看余额和退出功能


分析：

①：循环的时候，需要反复提示输入框，所以提示框写到循环里面

②：退出的条件是用户输入了 4，如果是4，则结束循环，不再弹窗

③：提前准备一个金额预先存储一个数额

④：取钱则是减法操作， 存钱则是加法操作，查看余额则是直接显示金额

⑤：输入不同的值，可以使用switch来执行不同的操作

# 数组

一种将一组数据存储在单个变量名下的优雅方式。接下来学习如何来创建一个数组，检索、添加和删除存储在数组中的元素。

简单来说，数组是一个包含了多个值的对象。

## 创建数组

**数组由方括号构成，其中包含用逗号分隔的元素列表。**

```
let arr = [] //创建空数组
let arr = [1, 2, 3] //创建长度为3的数组，数组元素取值分别为1,2,3
let arr = [
        [0, 1],
        [2, 3],
        [4, 5]
    ] //创建二维数组，长度为3，包含3行2列共6个元素
```

# 数组

可以将任何类型的元素存储在数组中 - 字符串，数字，对象，另一个变量，甚至另一个数组。可以混合各数据类型。

```
let sequence = [1, 1, 2, 3, 5, 8, 13];
let random = ['tree', 795, [0, 1, 2]];
```

# 数组

## 访问和修改数组元素

```
let shopping = ['bread', 'milk', 'cheese', 'hummus', 'noodles'];
shopping[0];
// returns "bread"
```

为单个数组元素提供新值来修改数组中的元素：

```
shopping[0] = 'tahini';
shopping;
// shopping will now return [ "tahini", "milk", "cheese", "hummus", "noodles" ]
```

# 数组

## 获取数组长度

通过使用 length 属性获取数组的长度。虽然 length 属性也有其他用途，但最常用于循环（循环遍历数组中的所有项）。

```
let sequence = [1, 1, 2, 3, 5, 8, 13];
for (let i = 0; i < sequence.length; i++) {
  console.log(sequence[i]);
}
```

# 数组

创建数组还可以用Array()，格式如下：

let 数组名 = new Array();

```
let arr = new Array() //创建空数组
let arr = new Array(3) //创建长度为3的数组
let arr = new Array(1, 2, 3) //创建长度为3的数组，数组元素取值分别为1,2,3
let arr = new Array([], []) //创建二维空数组，长度为2
let arr = new Array([1, 2, 3], [4, 5, 6]) //创建二维数组，长度为2，包含2行3列共6个元素
```

# 数组

## 数组求最大值和最小值

需求：求数组 [2,6,1,77,52,25,7] 中的最大值。分析

①：声明一个保存最大元素的变量 max。

②：默认最大值可以取数组中的第一个元素。

③：遍历这个数组，把里面每个数组元素和 max 相比较。

④：如果这个数组元素大于max 就把这个数组元素存到 max 里面，否则继续下一轮比较。

⑤：最后输出这个 max

# 数组

## 一些有用的数组方法

### 1、字符串和数组之间的转换

```
let myData = 'Manchester,London,Liverpool,Birmingham,Leeds,Carlisle';
let myArray = myData.split(',');
myArray;
myArray.length;
myArray[0]; // the first item in the array
myArray[1]; // the second item in the array
myArray[myArray.length-1]; // the last item in the array
```

# 数组

## 一些有用的数组方法

**也可以使用 join() 方法进行相反的操作。**

```
let myNewString = myArray.join(',');
myNewString;
```

将数组转换为字符串的另一种方法是使用 toString() 方法。 toString() 可以比 join() 更简单，因为它不需要一个参数，但更有限制。使用 join() 可以指定不同的分隔符。

```
let dogNames = ["Rocket","Flash","Bella","Slugger"];
dogNames.toString(); //Rocket,Flash,Bella,Slugger
```

# 数组

## 一些有用的数组方法

**2、添加和删除数组项，可以使用 push() 和 pop()**

```
myArray.push('Cardiff');
myArray;
myArray.push('Bradford', 'Brighton');
myArray;
```

从数组中删除最后一个元素的话直接使用 **pop()**

```
myArray.pop();
```

# 数组

## 一些有用的数组方法

**3、unshift() 和 shift() 从功能上与 push() 和 pop() 完全相同，只是它们分别作用于数组的开始，而不是结尾。**

```
myArray.unshift('Edinburgh');
myArray;

let removedItem = myArray.shift();
myArray;
removedItem;
```

# 数组筛选

将数组 [2, 0, 6, 1, 77, 0, 52, 0, 25, 7] 中大于等于 10 的元素选出来，放入新数组。

分析：

①：声明一个新的数组用于存放新数据newArr

②：遍历原来的旧数组， 找出大于等于 10 的元素

③：依次追加给新数组 newArr

# 数组

## 4、数组删除元素：splice

语法：

1. splice(start)
2. splice(start, deleteCount)
3. splice(start, deleteCount, item1)

**start**

指定修改的开始位置（从 0 计数）。如果超出了数组的长度，则从数组末尾开始添加内容；

**deleteCount**

整数，表示要移除的数组元素的个数。如果 deleteCount 是 0 或者负数，表示添加元素。

https://developer.mozilla.org/zh-CN/docs/Web/JavaScript/Reference/Global_Objects/Array/splice

# 数组

## 根据数据生成柱形图

需求： 用户输入四个季度的数据，可以生成柱形图


分析：

①：需要输入4次，所以可以把4个数据放到一个数组里面，利用循环，弹出4次框，同时存到数组里面。

②：遍历该数组，根据数据生成4个柱形图，渲染打印到页面中，柱形图就是div盒子，设置宽度固定，高度是用户输入的数据。

div里面包含显示的数字和第n季度。

# 数组

## 5、sort()方法

用于对数组进行升序排序。在排序时，sort方法会调用每个数组元素的toString()方法，然后比较得到的字符串，以确定如何排序。

例如：

```
let arr = [2,1,3]
arr.sort() //sort升序排序
console.log(arr) //1,2,3
let arr2=["java","hello","script"]
arr2.sort()
console.log(arr2) //"hello","java","script"
```

# 数组

## 6、reverse()方法

用于对数组进行逆序。

```
let arr = [2, 1, 3]
arr.reverse() //reverse()逆序
console.log(arr) //3,2,1
```

## 7、concat()方法

用于数组连接。这个方法会先创建当前数组的一个副本进行连接，

并返回新的副本，原始数组内容不变。

```
let arr = [1, 3, 5, 7]
let arrCopy = arr.concat(2, 4, 6, 8)
console.log(arrCopy) //[1, 3, 5, 7, 2, 4, 6, 8]
console.log(arr) //[1, 3, 5, 7]
```

# 数组

## 8、slice()方法

用于对数组进行切片，原数组不变。格式如下：数组.slice(开始下标, 结束下标)

slice()返回从原数组中截取下来的新数组，新数组包含开始下标元素，不包含结束下标元素。结束下标可以省略，下标可以为正序下标或逆序下标。例如定义数组arr后，其正序和逆序下标：

数组arr ：

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|

正序下标： 0 1 2 3 4 5 6 7 8 9

逆序下标： -10 -9 -8 -7 -6 -5 -4 -3 -2 -1

使用slice()对arr进行切片，代码如下：

# 数组

## 8、slice()方法

用于对数组进行切片，原数组不变。格式如下：数组.slice(开始下标, 结束下标)

slice()返回从原数组中截取下来的新数组，新数组包含开始下标元素，不包含结束下标元素。结束

序和逆序

数组arr ：

正序下标:

逆序下标:

```
let arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
// 截取下标3以后所有元素[3,4,5,6,7,8,9]
let arrSub1 = arr.slice(3)
// 截取下标3~5的元素，不包含结束下标元素[3,4]
let arrSub2 = arr.slice(3, 5)
// 截取下标3~倒数第2个的元素，不包含结束下标元素，[3,4,5,6,7]
let arrSub3 = arr.slice(3, -2)
//截取下标倒数第4个~倒数第1个的元素，不包含结束下标元素，[6,7,8]
let arrSub4 = arr.slice(-4, -1)
```

使用slice()对arr进行切片，代码如下：

# 数组

## 9、indexOf()方法和lastIndexOf()方法

两个方法用于在数组中查找数据。格式如下：

数组.indexOf(待查找项，查找起点的下标)

数组.lastIndexOf(待查找项，查找起点的下标)

indexOf()用于从前向后查找数组元素，默认从第一个元素开始查找。

lastIndexOf()用于从后向前查找数组元素，默认从最后一个元素开始查找。

数据找到后返回元素下标，未找到返回-1；查找过程中采用===进行比较。

```
let arr = [6, 3, 2, 4, 5, 7, 9]
arr.indexOf(3) //从开头查找数字3，返回下标1
arr.indexOf(5, 2) //从下标为2的位置开始查找数字5，返回下标4
arr.lastIndexOf(7, -2) //从倒数第2个元素开始查找数字7，返回下标5
arr.lastIndexOf(7, -4) //从倒数第4个元素开始查找数字7，未找到，返回-1
arr.indexOf('9') //未找到，返回-1
```

# 数组

## 10、map()方法

用于对每个数组元素运行指定函数，映射结果组成新的数组并返回。

例如，使用map()方法实现求数组中每个元素的平方。

```
let arr = [1, 2, 3, 4]
let newArr = arr.map(func1) //返回新的数组[1, 4, 9, 16]
//映射函数func1(), 求平方
function func1(x) {
    return x * x
}
```

# 数组

## 11、reduce()方法

可以对数组每个元素运行指定函数作为累计器，数组元素依次汇集，最终成为一个值。例如，使用reduce()方法求数组元素的和，如图所示：

```
let arr = [1, 2, 3, 4]
let sum = arr.reduce(func1) //sum结果为10
function func1(x, y) {
    return x + y
}
```

# 数组

## 12、filter()方法

可以对数组每个元素运行指定函数，过滤结果组成新的数组并返回。

例如：

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
let newArr = arr.filter(func1) //newArr=[2,4,6,8,10]
//过滤函数func1，求偶数元素
function func1(x) {
    return x % 2 == 0
}
```

# 数组

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
arr.every(func1) //结果为false
function func1(x) {
    return x > 6
}
```

## 13、every()方法

用于检测是否数组每个元素都符合函数中的条件。例如：

## 14、some()方法

用于检测是否有一些数组元素符合函数中的条件。例如：

```
let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
arr.some(func1) //结果为true
function func1(x) {
    return x > 6
}
```

# 数组

## 15、forEach()方法

为每个数组元素调用一次函数。

```
let sum = 0
let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
arr.forEach(func1)
console.log(sum) // 输出求和结果为55
function func1(x) {
    sum += x
}
```

# 函数

是被设计为执行特定任务的代码块。

**说明：**

函数可以把具有相同或相似逻辑的代码“包裹”起来，通过函数调用执行这些被“包裹”的代码逻辑，这么做的优势：有利于精简代码方便复用。

--它允许你在一个代码块中存储一段用于处理单任务的代码，然后在任何你需要的时候用一个简短的命令来调用，而不是把相同的代码写很多次。

# 函数

**函数的声明和调用：**

```
function 函数名(参数列表){
    //函数体
    return 返回值;
}
```

```
function draw() {
  ctx.clearRect(0,0,WIDTH,HEIGHT);
  for (var i = 0; i < 100; i++) {
    ctx.beginPath();
    ctx.fillStyle = 'rgba(255,0,0,0.5)';
    ctx.arc(random(WIDTH), random(HEIGHT), random(50), 0, 2 * Math.PI);
    ctx.fill();
  }
}
```

```
draw();
```

无返回值的函数可以直接函数名调用；有返回值的函数，其返回值可以直接输出或赋值给其他变量。

执行到return语句时，函数将停止执行，将返回值返回给主调函数。

# 函数

**匿名函数：一个没有名称的函数**

```
function() {
  alert('hello');
}
```

它也不会自己做任何事情。通常将匿名函数与事件处理程序一起使用，例如，如果单击相关按钮，以下操作将在函数内运行代码

```
var myButton = document.querySelector('button');

myButton.onclick = function() {
  alert('hello');
}
```

# 函数

**匿名函数：一个没有名称的函数**

还可以将匿名函数赋值给变量，

```
var myGreeting = function() {
  alert('hello');
}
```

调用通过如下：

```
myGreeting();
```

- 匿名函数也称为函数表达式。函数表达式与函数声明有一些区别。函数声明会进行声明提升（declaration hoisting），而函数表达式不会。

# 函数

## 函数传参

一些函数需要在调用它们时指定参数——这些参数值需要放在函数括号内，才能正确地完成其工作。例如：浏览器的内置字符串replace()函数需要两个参数：在主字符串中查找的子字符串，以及用以下替换该字符串的子字符串：

```
var myText = 'I am a string';

var newString = myText.replace('string', 'sausage');
```

# 函数

```
function getSquare(num1) {
    document.write(num1 * num1)
}
```

```
getSquare(8)
```

```
function getSum(num1, num2) {
    document.write(num1 + num2)
}
```

```
getSum(10, 20)
```

# 函数　　函数返回值

除基本类型外，函数返回值还可以是数组、对象和函数。

（1）返回数组

例如，电话号码是由区号district和座机号number组成，定义getPhone()函数，以数组形式返回电话号码的两个部分，代码如下：

```
function getPhone() {
    let district = '010',
        number = '87654321'
    return [district, number] //返回数组
}
let phone = getPhone() //调用函数
console.log(phone[0] + '-' + phone[1]) //输出010-87654321

//也可以直接用数组结构接收返回值
let [pdistrict, pnumber] = getPhone()
console.log(pdistrict + '-' + pnumber) //输出010-87654321
```

# 函数　　函数返回值

函数返回值还可以是对象。

（2）返回对象

例如：

```
function getPhone() {
    let district = '010',
        number = '87654321'

    return { district, number } //返回对象
}
let phone = getPhone() //调用函数
console.log(phone.district + '-' + phone.number) //输出010-87654321
```

# 函数　　函数返回值

函数返回值还可以是函数。

（3）返回函数

例如：

```
function getPhone() {
    let district = '010',
        number = '87654321'
    function printPhone() {
        console.log(district + '-' + number)
    }
    return printPhone //返回函数
}
let phone = getPhone() //phone指向getPhone()的返回值，即printPhone()函数
phone() //输出010-87654321
```

# 函数　　函数表达式

函数表达式是以赋值形式使函数成为表达式的一部分，并可以作为更强大表达式的一部分，此时可以带函数名或者不带函数名。带函数名的称为命名函数。不带函数名的则为匿名函数。

格式：

let 变量|对象属性=function 函数名(){ }; //命名函数的函数表达式

let 变量|对象属性=function (){ }; //匿名函数的函数表达式

## 1、赋值给变量

（1）命名函数

```
let show = function f() {
    alert('Hello~')
}
show() //用show变量调用函数
```

(2)匿名函数：可以将匿名函数赋值给一个变量，再通过变量进行函数调用

```
let show = function () {
    alert('Hello~')
}
show() //用show变量调用匿名函数
```

# 函数　　函数表达式

2、赋值给对象属性

将匿名函数赋值给对象属性也很常见，如果赋值给事件属性，则事件发生时调用该函数。例如，定义匿名函数，单击“显示”时，调用该匿名函数，显示警告框，给出提示信息。

```
<button id="show">显示</button>
<script>
    // 找到id为show的按钮
    let showBtn = document.getElementById('show')
    // 单击按钮时，调用匿名函数显示信息
    showBtn.onclick = function () {
        alert('Hello~')
    }
</script>
```

# 函数

## 函数表达式

3、立即执行函数

匿名函数可以通过自调用方式来执行，此时也被称为立即执行函数。立即执行函数执行后被释放，通常适用于只被执行一次的函数，也常用于初始化工作。立即执行函数函数格式如下：

```
(function (形参列表) {
    //函数体
    return 返回值
})(实参列表)
```

例如，定义无参数和返回值的立即执行函数输出字符串，代码如下：

```
(function () {
    alert('Hello~')
})()
```

也可以带有参数和返回值。

```
let mul = (function (x, y) {
    return x * y
})(3, 6)
console.log(mul) // 输出18
```

# 函数　　函数表达式

4、箭头函数

箭头函数是更简洁的表达方式，声明函数时用=>代替function关键字，格式如下：

(形参列表)=>{函数体语句}

(形参列表)=>{return 表达式}

(形参列表)=>{表达式}

例如，求三个数的和，可以用箭头函数表示，代码如下：

```
let sum = (a, b, c) => {
    return a + b + c
}
//等价于  let sum = (a,b,c)=>a+b+c
console.log(sum(3, 5, 7)) //输出15
```

# 函数

## 函数嵌套

JavaScript允许在一个函数体内嵌套定义另一个函数。在函数嵌套中，按照嵌套关系分为外部函数和内部函数。

（1）定义函数嵌套

定义函数嵌套时，内部函数是外部函数私有的，内部函数只能在外部函数中调用。例如，定义外部函数isSumLess()，比较两个数组和的大小，第一个数组和小于第二个数组和返回true，反之，返回false。外部函数通过调用内部函数sum()求得任意数组的和。代码如下：

```
let arrA = [1, 2, 3, 4, 5]
let arrB = [6, 7, 9, 10]
console.log(isSumLess(arrA, arrB)) //true
//外部函数
function isSumLess(ax, ay) {
    //内部函数，对数组求和，只能在isSumLess()内调用
    function sum(arr) {
        let s = 0
        for (let i = 0; i < arr.length; i++) {
            s += arr[i]
        }
        return s
    }
    return sum(ax) < sum(ay)
}
```

# 函数　　函数嵌套

（2）内部函数访问外部函数变量

在函数嵌套时，内部函数可以直接访问外部函数的变量，反之则不可以。例如：

```
// 外部函数声明
function outter() {
    //内部函数也可以访问这个变量
    let outData = 10
    // 内部函数声明
    function inner() {
        // 外部函数不可以访问这个变量
        let inData = 22
        outData = 0
        //输出0
        console.log('inner():' + outData)
    }
    // 调用内部函数
    inner()
    // 输出0
    console.log('outter():' + outData)
}
// 调用外部函数
outter()
```

# 函数

## 作用域和声明提升

1、作用域

变量作用域就是变量可以被引用的有效范围。JavaScript作用域分为全局作用域、局部作用域、块作用域。

A. 全局作用域：由整个JavaScript执行环境构成。

B. 局部作用域：也称为函数作用域，由函数声明{}构成。

C. 块作用域：由{}构成，比如if语句和for语句的{}也属于块作用域。

2、声明提升

JavaScript代码在运行时分为语法分析、预编译和解释执行阶段。在预编译阶段，函数声明和变量声明会被提升，并且函数提升优先级高于变量提升，提升后，函数声明会位于变量声明之前。

（1）变量声明提升

在预编译阶段，全局作用域中声明的变量会提升至全局最顶层，局部作用域即函数内声明的变量只会提升至该局部作用域最顶层。例如，如下代码在if语句中定义变量msg，被提升到if之前成为全局变量。

```
console.log(msg)
if (true) {
  var msg = 'Hello' //var定义变量会被提升至全局作用域
}
```

变量提升后，
等价于右侧代码

```
var msg //变量声明被提升至全局作用域
console.log(msg) //输出undefined
if (true) {
  var msg = 'Hello'
}
```

# 函数　　作用域和声明提升

（2）函数声明提升

在预编译阶段，当前作用域的函数声明提升到整个作用域的最前面。例如，showMsg()函数的调用位于函数声明之前，可以正确输出结果，代码如下：

```
function show() {
    showMsg() //输出Hello~
    function showMsg() {
        console.log('Hello~')
    }
}
show()
```

本质上函数声明已经提升到函数调用之前，等同于如下代码：

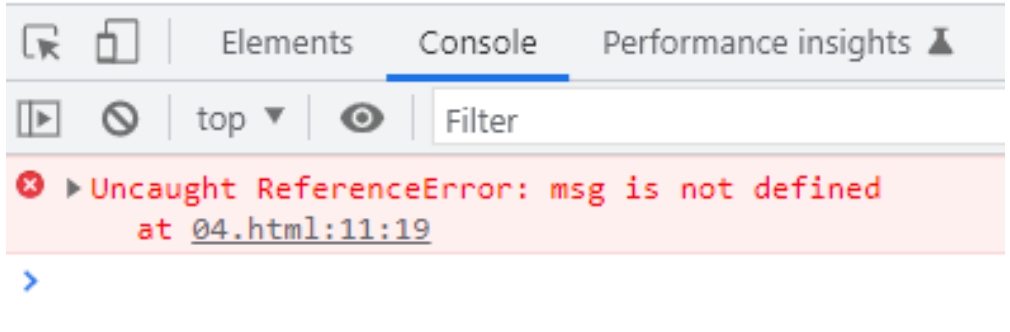
```
function show() {
    // 函数声明提升到作用域最前面
    function showMsg() {
        console.log('Hello~')
    }
    showMsg() //输出Hello~
}
show()
```

# 函数　　作用域和声明提升

3、let定义变量

let和var定义变量在全局作用域和局部作用域中是类似的，但是let定义变量可以拥有块作用域。例如，以下代码用let声明msg变量，在全局作用域中访问时，将提示变量未定义错误：

```
console.log(msg)
if (true) {
    let msg = 'Hello~'
}
```




对比一下，var关键字在块中声明的变量将被提升为全局变量，造成全局污染，例如：

```
var i = 6 //全局作用域i=6
for (var i = 0; i < 5; i++) {
    console.log('Hello~')
}
console.log(i) //i为5，for中的i污染全局变量i
```

为了避免全局污染，在块作用域中，应该是用let定义变量。

```
var i = 6 //全局作用域i=6
for (let i = 0; i < 5; i++) {
    console.log('Hello~')
}
console.log(i) //i为6
```

# 函数　　闭包

**1、概念**

闭包就是能够读取其他函数内部变量的函数。由于在JavaScript语言中，只有函数内部的子函数才能读取局部变量，因此可以把闭包简单理解成“定义在一个函数内部的函数”。所以，在本质上，闭包就是将函数内部和函数外部连接起来的一座桥梁。

**2、特性**

① 内嵌函数：函数嵌套函数，内嵌函数对函数中的局部变量进行访问

② 局部变量：在函数内定义有共享意义的局部变量

③ 外部使用：函数向外返回此内嵌函数，外部可通过此内嵌函数访问声明在函数中的局部变量，而此变量在外部是通过其他路径无法访问的

④ 参数和变量不会立即被垃圾回收机制回收

# 函数　　闭包

### 3、闭包的应用

闭包常用于生成计数器、缓存等，下面以计数器问题为例阐述闭包的应用。

例如，我们可以用全局变量完成一个计数器：

```
let num=0
function counter(){
    num++
    return num
}
counter() //num=1
counter() //num=2
counter() //num=3
```

这种方式下，计数器变量num是全局变量，如果改为局部变量，就无法实现计数器效果，代码如下：

```
//计数器，每执行一次，num加1
function counter() {
    let num = 0 //计数器变量，局部变量
    num++
    return num
}
counter() //num=1
counter() //num=1
counter() //num=1
```

# 函数　　闭包

## 3、闭包的应用

这个时候我们可以采用闭包来完成num的局部化，解决上边的问题：

```
//外部函数
function outterplus() {
  let num = 0 //局部变量
  //内部函数，与num一起形成闭包
  function plus() {
    num++
    return num
  }
  return plus //返回内部函数
}
//调用外部函数outterplus()之后，无法再通过outterplus()访问局部变量num
//但外部函数outterplus()返回内部函数plus()
let counter = outterplus() //counter指向内部函数plus()
counter() //调用counter(),就执行plus()，就可以访问num，num=1
counter() //num=2
counter() //num=3
```

# 函数　　闭包

**4、使用闭包的注意点**

由于闭包会使得函数中的变量都被保存在内存中，内存消耗很大，所以不能滥用闭包，否则会造成网页的性能问题，在IE中可能导致内存泄露。解决方法是，在退出函数之前，将不使用的局部变量全部删除。

**5、闭包的优点**

可读取函数内部的变量；

局部变量可以保存在内存中，实现数据共享；

执行过程中所有变量都匿名在函数内部；

**6、闭包的缺点**

使函数内部变量存在于内存中，内存消耗大；

滥用闭包可能导致内存泄露；

闭包可以在父函数外部改变父函数内部的值，慎操作；

# 总结

数组包含多个数组元素，数组元素类型通常是一致的，但是JavaScript支持数组元素类型不一致。可以通过元素下标访问元素，并结合for循环或forEach()方法遍历数组。JavaScript提供了一系列操作数组的方法。

| 方法 | 说明 | 方法 | 说明 |
| --- | --- | --- | --- |
| toString() | 转换为字符串 | concat() | 数组连接 |
| join() | 转换带分隔符的字符串 | slice() | 切片，截取新数组 |
| push() | 添加数组元素到数组末尾 | indexOf() | 从前向后查找 |
| pop() | 删除数组元素最后一项 | lastIndexOf() | 从后向前查找 |
| shift() | 删除数组元素第一项 | map() | 函数映射形成新数组 |
| unshift() | 添加元素到数组开头 | reduce() | 迭代每个元素汇总为一个值 |
| sort() | 升序排序 | filter() | 过滤数组元素 |
| reverse() | 逆序 | every() | 是否每个元素都符合条件 |
| forEach() | 遍历数组 | some() | 是否一些元素都符合条件 |

# 总结

函数是逻辑独立的代码段，JavaScript支持命名函数、匿名函数。声明函数有函数语句和函数表达式两种形式，函数表达式可以是命名函数、匿名函数、立即执行函数和箭头函数。

函数可以嵌套定义，内部函数可以访问外部函数的变量，所以内部函数作为外部函数返回值时会将内部函数和这些变量绑定在一起，形成闭包。闭包可用于计数器、缓存等用途。

JavaScript中有全局作用域、局部作用域、块作用域。在函数内部或者块内部用var定义的变量会被提升到全局作用域；函数和块内用let定义变量则不会被提升，应尽量采用let定义变量。