

# JS(1)

## 1.初识 JavaScript

### 1.1 JavaScript 发展史

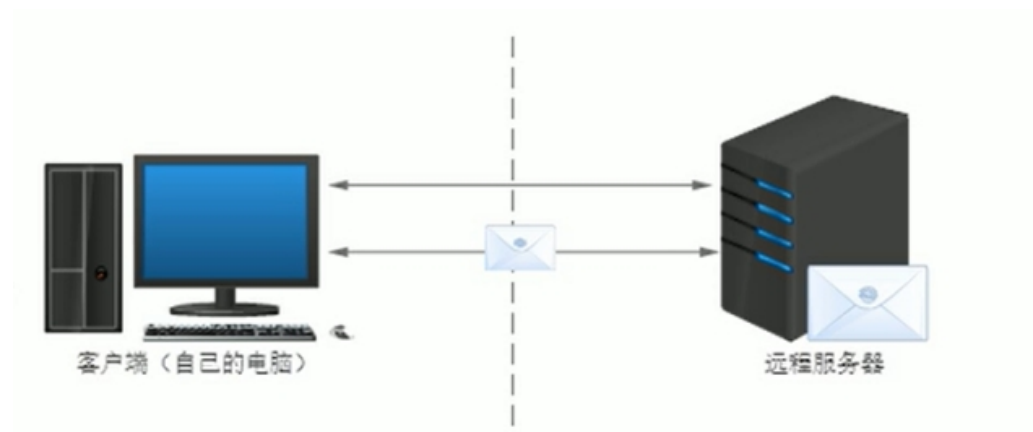
它最初由 Netscape 的 Brendan Eich(布兰登·艾奇)设计。JavaScript 是甲骨文公司的注册商标。Ecma 国际以 JavaScript 为基础制定了 ECMAScript 标准。JavaScript 也可以用于其他场合,如服务器端编程。完整的 JavaScript 实现包含三个部分: ECMAScript, 文档对象模型, 浏览器对象模型。

Netscape 在最初将其脚本语言命名为 LiveScript, 后来 Netscape 在与 Sun 合作之后将其改名为 JavaScript。JavaScript 最初受 Java 启发而开始设计的, 目的之一就是“看上去像 Java”, 因此语法上有类似之处, 一些名称和命名规范也借自 Java。但 JavaScript 的主要设计原则源自 Self 和 Scheme。JavaScript 与 Java 名称上的近似, 是当时 Netscape 为了营销考虑与 Sun 微系统达成协议的结果。为了取得技术优势, 微软推出了 JScript 来迎战 JavaScript 的脚本语言。为了互用性, Ecma 国际(前身为欧洲计算机制造商协会)创建了 ECMA-262 标准 (ECMAScript)。两者都属于 ECMAScript 的实现。尽管 JavaScript 作为给非程序人员的脚本语言, 而非作为给程序人员的脚本语言来推广和宣传, 但是 JavaScript 具有非常丰富的特性。

### 1.2 JavaScript 是什么

JavaScript 一种直译式脚本语言, 是一种动态类型、弱类型、基于原型的语言, 内置支持类型。它的解释器被称 JavaScript 引擎, 为浏览器的一部分, 广泛用于客户端的脚本语言, 最早是在 HTML (标准通用标记语言下的一个应用) 网页上使用, 用来给 HTML 网页增加动态功能。

Script : 是脚本的意思 脚本语言: 不需要编译, 运行过程中由 js 解释器 (js 引擎) 逐行来进行解释并执行 为了阅读方便, 我们后面把 JavaScript 简称为 JS。



### 1.3 JavaScript 的作用

- 表单动态校验(密码强度检测) ( JS 产生最初的目的 )
- 网页特效
- 服务端开发(Node.js)
- 桌面程序
- App
- 控制硬件-物联网
- 游戏开发

## 1.4 HTML CSS JS 之间的关系

HTML/CSS 标记语言--描述类语言

- HTML 决定网页结构和内容( 决定看到什么 ), 相当于人的身体
- CSS 决定网页呈现给用户的模样( 决定好不好看 ), 相当于给人穿衣服、化妆

JS 脚本语言--编程类语言

- 实现业务逻辑和页面控制( 决定功能 ), 相当于人的各种动作

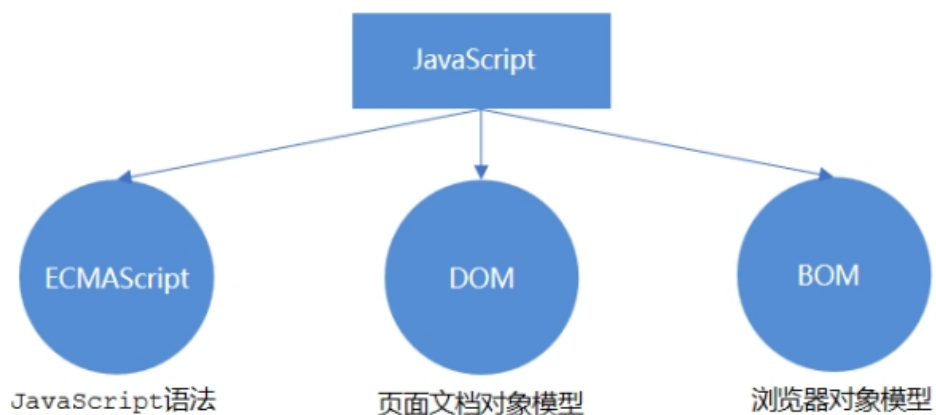
## 1.5 浏览器执行 JS 简介

浏览器分成两部分: 渲染引擎和 JS 引擎

- 渲染引擎: 用来解析 HTML 与 CSS, 俗称内核
- JS 引擎: 也称为 JS 解释器。用来读取网页中的 JavaScript 代码, 对其处理后运行

浏览器本身并不会执行 JS 代码, 而是通过内置 JavaScript 引擎(解释器) 来执行 JS 代码。JS 引擎执行代码时逐行解释每一句源码(转换为机器语言), 然后由计算机去执行, 所以 JavaScript 语言归为脚本语言, 会逐行解释执行。

## 1.6 JS 组成



(1)ECMAScript ECMAScript 是由 ECMA 国际（原欧洲计算机制造商协会）进行标准化的一门编程语言，这种语言在万维网上应用广泛，它往往被称为 JavaScript 或 JScript，但实际上后两者是 ECMAScript 语言的实现和扩展。

(2)DOM ——文档对象模型 文档对象模型（Document Object Model, 简称 DOM），是 W3C 组织推荐的处理可扩展标记语言的标准编程接口。通过 DOM 提供的接口可以对页面上的各种元素进行操作（大小、位置、颜色等）。

(3)BOM ——浏览器对象模型 BOM（Browser Object Model, 简称 BOM）是指浏览器对象模型，它提供了独立于内容的、可以与浏览器窗口进行互动的对象结构。通过 BOM 可以操作浏览器窗口，比如弹出框、控制浏览器跳转、获取分辨率等。

## 1.7 JS 书写位置

JS 有 3 种书写位置，分别为行内、内嵌和外部。

(1)行内式 JS

```
<input type="button" value="点我试试" onclick="alert('Hello World')" />
```

- 可以将单行或少量 JS 代码写在 HTML 标签的事件属性中（以 on 开头的属性），如：onclick
- 注意单双引号的使用：在 HTML 中我们推荐使用双引号，JS 中我们推荐使用单引号
- 可读性差，在 html 中编写 JS 大量代码时，不方便阅读；
- 引号易错，引号多层嵌套匹配时，非常容易弄混；

(2) 内嵌 JS

```
<script>

alert('Hello World~!');

</script>
```

- 可以将多行 JS 代码写到 `<script>` 标签中
- 内嵌 JS 是学习时常用的方式

### (3) 外部 JS 文件

```
<script src="my.js"></script>
```

- 利于 HTML 页面代码结构化, 把大段 JS 代码独立到 HTML 页面之外, 既美观, 也方便文件级别的复用
- 引用外部 JS 文件的 `script` 标签中间不可以写代码
- 适合于 JS 代码量比较大的情况

## 2. JavaScript 注释

---

### 2.1 单行注释

为了提高代码的可读性, JS 与 CSS 一样, 也提供了注释功能。JS 中的注释主要有两种, 分别是单行注释和多行注释。

单行注释的注释方式如下:

```
// 我是一行文字, 不想被 JS 引擎 执行, 所以 注释起来  
  
// 用来注释单行文字 ( 快捷键 ctrl + / )
```

### 2.2 多行注释

多行注释的注释方式如下:

```
/* 获取用户年龄和姓名 并通过提示框显示出来 */  
  
/* */ 用来注释多行文字 ( 默认快捷键 alt + shift + a )
```

## 3. JavaScript 输入输出语句

---

备注: `alert()` 主要用来显示消息给用户, `console.log()` 用来给程序员自己看运行时的消息。

方法	说明	归属
alert(msg)	浏览器弹出警示框	浏览器
console.log(msg)	浏览器控制台打印输出信息	浏览器
prompt(info)	浏览器弹出输入框，用户可以输入	浏览器

## 4.变量

### 4.1.1 什么是变量,变量提升

变量是用于存放数据的容器。我们通过 变量名 获取数据，甚至数据可以修改。

var 命令会发生“变量提升”现象，即变量可以在声明之前使用，值为 undefined。这种现象多多少少是有些奇怪的，按照一般的逻辑，变量应该在声明语句之后才可以使用。为了纠正这种现象，let 命令改变了语法行为，它所声明的变量一定要在声明后使用，否则报错。var 不建议用，这个是 es6 语法的，在查文档时,很可能看见 var。建议使用 let。

### 4.1.2 变量在内存中的存储

变量是程序在内存中申请的一块用来存放数据的空间。类似我们酒店的房间，一个房间就可以看做是一个变量。

### 4.2.1 变量的使用

变量在使用时分为两步：(1)声明变量 (2)赋值

(1)声明变量

· let age; // 声明一个 名称为 age 的变量

· age 是程序员定义的变量名，我们要通过变量名来访问内存中分配的空间

(2) 赋值 age = 10; // 给 age 这个变量赋值为 10

## 4.3 变量的初始化

```
let age = 18; // 声明变量同时赋值为 18
```

声明一个变量并赋值，我们称之为变量的初始化。

小练习 1: 变量的使用

(1)弹出一个输入框，提示用户输入姓名。

(2)弹出一个对话框，输出用户刚才输入的姓名。

## 4.4 变量语法扩展

### 4.4.1 更新变量

一个变量被重新赋值后，它原有的值就会被覆盖，变量值将以最后一次赋的值为准。

```
let age = 18;

age = 81; // 最后的结果就是 81 因为 18 被覆盖掉了
```

### 4.4.2 同时声明多个变量

同时声明多个变量时，只需要写一个 let，多个变量名之间使用英文逗号隔开。

```
let age = 10, name = 'zs', sex = 2;
```

## 4.5 变量命名规范

- 由字母(A-Z a-z)、数字(0-9)、下划线(\_)、美元符号( \$ )组成，如：usrAge, num01, \_name
- 严格区分大小写。let app; 和 let App; 是两个变量
- 不能 以数字开头。18age 是错误的
- 不能 是关键字、保留字。例如：let、for、while
- 变量名必须有意义。
- 遵守驼峰命名法。首字母小写，后面单词的首字母需要大写。myFirstName

以下哪些是合法的变量名？

第一组	第二组	第三组
let a	let <u>userName</u>	let <u>theWorld</u>
let 1	let \$name	let the world
let age18	let _sex	let the_world
let 18age	let &sex	let for

小练习 2: 交换两个变量的值 ( 实现思路: 使用一个 临时变量 用来做中间存储 )

## 4.6 小结

### 驼峰命名法

规范中标识符采用驼峰大小写格式, 驼峰命名法由小(大)写字母开始, 后续每个单词首字母都大写。根据首字母是否大写, 分为两种方式:

1. Pascal Case 大驼峰式命名法: 首字母大写。eg: StudentInfo、UserInfo、ProductInfo
2. Camel Case 小驼峰式命名法: 首字母小写。eg: studentInfo、userInfo、productInfo

备注:写英文, 不要写拼音

## 5.数据类型

### 5.1.1 为什么需要数据类型

在计算机中, 不同的数据所需占用的存储空间是不同的, 为了便于把数据分成所需内存大小不同的数据, 充分利用存储空间, 于是定义了不同的数据类型。

简单来说, 数据类型就是数据的类别型号。比如姓名“张三”, 年龄 18, 这些数据的类型是不一样的。

### 5.1.2 变量的数据类型

变量是用来存储值的所在处, 它们有名字和数据类型。变量的数据类型决定了如何将代表这些值的位存储到计算机的内存中。JavaScript 是一种弱类型或者说动态语言。这意味着不用提前声明变量的类型, 在程序运行过程中, 类型会被自动确定。

```
let age = 10; // 这是一个数字型

let areYouOk = '是的'; // 这是一个字符串
```

在代码运行时, 变量的数据类型是由 JS 引擎 根据 = 右边变量值的数据类型来判断 的, 运行完毕之后, 变量就确定了数据类型。

JavaScript 拥有动态类型，同时也意味着相同的变量可用作不同的类型：

```
let x = 6;    // x 为数字

let x = "Bill"; // x 为字符串
```

### 5.1.3 数据类型的分类

JS 把数据类型分为两类：

- 简单数据类型（Number,String,Boolean,Undefined,Null）
- 复杂数据类型（object）

### 5.2.1 简单数据类型（基本数据类型）

JavaScript 中的简单数据类型及其说明如下：

简单数据类型	说明	默认值
Number	数字型，包含 整型值和浮点型值，如 21、0.21	0
Boolean	布尔值类型，如 true 、 false，等价于 1 和 0	false
String	字符串类型，如 "张三" 注意咱们js 里面，字符串都带引号	""
Undefined	var a; 声明了变量 a 但是没有给值，此时 a = undefined	undefined
Null	var a = null; 声明了变量 a 为空值	null

### 5.2.2 数字型 Number

JavaScript 数字类型既可以用来保存整数值，也可以保存小数(浮点数)。

```
let age = 21;    // 整数

let Age = 21.3747; // 小数
```

(1)数字型进制

最常见的进制有二进制、八进制、十进制、十六进制。

// 1.八进制数字序列范围：0~7

```
let num1 = 07; // 对应十进制的 7

let num2 = 019; // 对应十进制的 19
```



```
let num3 = 08; // 对应十进制的 8
```

// 2.十六进制数字序列范围: 0~9 以及 A~F

```
let num = 0xA;
```

· 现阶段我们只需要记住, 在 JS 中八进制前面加 0, 十六进制前面加 0x

## (2)数字型范围

JavaScript 中数值的最大和最小值

```
alert(Number.MAX_VALUE); // 1.7976931348623157e+308
```

```
alert(Number.MIN_VALUE); // 5e-324
```

· 最大值: Number.MAX\_VALUE, 这个值为: 1.7976931348623157e+308

· 最小值: Number.MIN\_VALUE, 这个值为: 5e-32

## (3)数字型三个特殊值

```
alert(Infinity); // Infinity
```

```
alert(-Infinity); // -Infinity
```

```
alert(NaN); // NaN
```

· Infinity , 代表无穷大, 大于任何数值

· -Infinity , 代表无穷小, 小于任何数值

· NaN , Not a number, 代表一个非数值

## (4)isNaN()

用来判断一个变量是否为非数字的类型, 返回 true 或者 false



## 5.2.3 字符串型 String

字符串型可以是引号中的任意文本, 其语法为 双引号 "" 和 单引号''

```
let strMsg = "我爱北京天安门~"; // 使用双引号表示字符串
```

```
let strMsg2 = '我爱吃猪蹄~'; // 使用单引号表示字符串
```

// 常见错误

```
let strMsg3 = 我爱大肘子; // 报错, 没使用引号, 会被认为是 js 代码, 但 js 没有这些语法
```

因为 HTML 标签里面的属性使用的是双引号, JS 这里我们更推荐使用单引号。

#### (1) 字符串引号嵌套

JS 可以用单引号嵌套双引号, 或者用双引号嵌套单引号 (外双内单, 外单内双)

```
let strMsg = '我是"高帅富"程序猿'; // 可以用"包含""
```

```
let strMsg2 = "我是'高帅富'程序猿"; // 也可以用"" 包含"
```

// 常见错误

```
let badQuotes = 'What on earth?'; // 报错, 不能 单双引号搭配
```

#### (2) 字符串转义符

类似 HTML 里面的特殊字符, 字符串中也有特殊字符, 我们称之为转义符。

转义符都是 \ 开头的, 常用的转义符及其说明如下:

转义符	解释说明
\n	换行符, n 是 newline 的意思
\\	斜杠 \
\'	' 单引号
\"	" 双引号
\t	tab 缩进
\b	空格, b 是 blank 的意思

小练习 3: 弹出网页警示框

### 此网页显示

酷热难耐，火辣的太阳底下，我挺拔的身姿，成为了最为独特的风景。  
我审视四周，这里，是我的舞台，我就是天地间的王者。  
这一刻，我豪气冲天，终于大喊一声：“收破烂啦~”

确定

酷热难耐，火辣的太阳底下，我挺拔的身姿，成为了最为独特的风景。我审视四周，这里，是我的舞台，我就是天地间的王者。这一刻，我豪气冲天，终于大喊一声：“收破烂啦~”

### (3)字符串长度

字符串是由若干字符组成的，这些字符的数量就是字符串的长度。通过字符串的 `length` 属性可以获取整个字符串的长度。

```
let strMsg = "我是帅气多金的程序猿！";
```

```
alert(strMsg.length); // 显示 11
```

### (4)字符串拼接

- 多个字符串之间可以使用 `+` 进行拼接,其拼接方式为 字符串 + 任何类型 = 拼接之后的新字符串
- 拼接前会把与字符串相加的任何类型转成字符串,再拼接成一个新的字符串

//1.1 字符串 "相加"

```
alert('hello' + ' ' + 'world'); // hello world
```

//1.2 数值字符串 "相加"

```
alert('100' + '100'); // 100100
```

//1.3 数值字符串 + 数值

```
alert('11' + 12); // 1112
```

总结口诀：数值相加，字符相连

### (5)字符串拼接加强

```
console.log('同学' + 18); // 只要有字符就会相连
```

```
let age = 18;
```

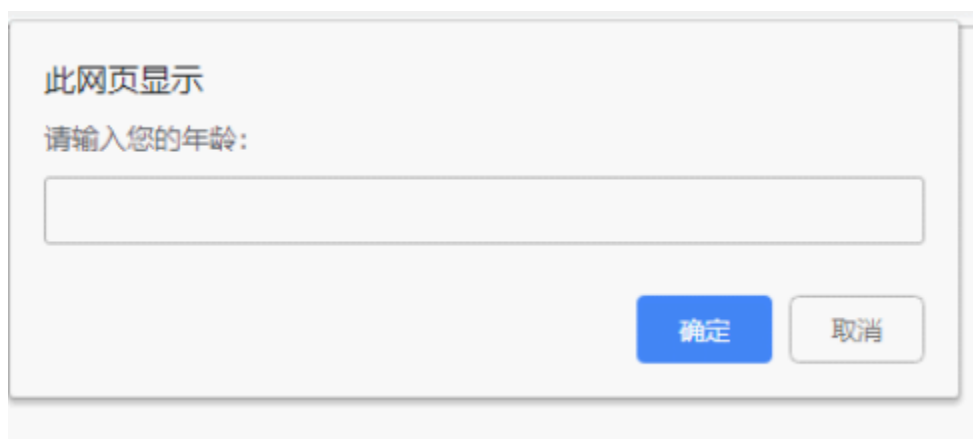
```
// console.log('同学 age 岁啦'); // 这样不行哦

console.log('同学' + age); // 同学 18

console.log('同学' + age + '岁啦'); // 同学 18 岁啦
```

- 我们经常会将字符串和变量来拼接，因为变量可以很方便地修改里面的值
- 变量是不能添加引号的，因为加引号的变量会变成字符串
- 如果变量两侧都有字符串拼接，口诀“引引加加”，删掉数字，变量写加中间

案例 1：显示年龄



弹出一个输入框，需要用户输入年龄，之后弹出一个警示框显示“您今年 xx 岁啦”（xx 表示刚才输入的年龄）

案例分析 这是利用 JS 编写的一个非常简单的交互效果程序。



你喜欢我吗



我答应啦~



交互编程的三个基本要素：

1. 你喜欢我吗？ → 这是用户输入
2. 女孩想了想 → 这是程序内部处理
3. 最后给了你一巴掌 → 这是输出结果

那么在程序中要如何实现呢？

- ①弹出一个输入框 (prompt), 让用户输入年龄 (用户输入)
- ②把用户输入的值用变量保存起来,把刚才输入的年龄与所要输出的字符串拼接 (程序内部处理)
- ③使用 alert 语句弹出警示框 (输出结果)

案例代码

```
// 弹出一个输入框 (prompt), 让用户输入年龄 (用户输入)

// 把用户输入的值用变量保存起来,把刚才输入的年龄与所要输出的字符串拼接 (程序内部处理)

// 使用 alert 语句弹出警示框 (输出结果)
```

```
let age = prompt('请输入您的年龄');

let str = '您今年已经' + age + '岁了';

alert(str);
```

## 5.2.4 布尔型 Boolean

布尔类型有两个值: true 和 false , 其中 true 表示真 (对), 而 false 表示假 (错)。

布尔型和数字型相加的时候, true 的值为 1 , false 的值为 0。

```
console.log(true + 1); // 2

console.log(false + 1); // 1
```

## 5.2.5 Undefined 和 Null

一个声明后没有被赋值的变量会有一个默认值 undefined ( 如果进行相连或者相加时, 注意结果)

一个声明变量给 null 值, 里面存的值为空 (学习对象时, 我们继续研究 null)

```
let leti = null;

console.log('你好' + leti); // 你好 null

console.log(11 + leti); // 11
```

```
console.log(true + leti); // 1
```

## 5.3 获取变量数据类型

### 5.3.1 获取检测变量的数据类型

typeof 可用来获取检测变量的数据类型

```
let num = 18;  
  
console.log(typeof num) // 结果 number
```

不同类型的返回值

类型	例	结果
String	typeof "小白"	"string"
Number	typeof 18	"number"
Boolean	typeof true	"boolean"
Undefined	typeof undefined	"undefined"
Null	typeof null	"object"

### 5.3.2 字面量

字面量是在源代码中一个固定值的表示法，通俗来说，就是字面量表示如何表达这个值。

## 5.4 数据类型转换

### 5.4.1 什么是数据类型转换

使用表单、prompt 获取过来的数据默认是字符串类型的，此时就不能直接简单的进行加法运算，而需要转换变量的数据类型。通俗来说，就是把一种数据类型的变量转换成另外一种数据类型。

我们通常会实现 3 种方式的转换：

- 转换为字符串类型
- 转换为数字型
- 转换为布尔型

### 5.4.2 转换为字符串

方式	说明	案例
<code>toString()</code>	转换成字符串	<code>let num = 1; alert(num.toString());</code>
<code>String()</code> 强制转换	转换成字符串	<code>let num = 1; alert(String(num));</code>
加号拼接字符串	和字符串拼接的结果都是字符串	<code>let num = 1; alert(num + '我是字符串');</code>

- `toString()` 和 `String()` 使用方式不一样。
- 三种转换方式，我们更喜欢用第三种加号拼接字符串转换方式， 这一种方式也称之为隐式转换。

### 5.4.3 转换为数字型（重点）

方式	说明	案例
<code>parseInt(string)</code> 函数	将string类型转成整数数值型	<code>parseInt('78')</code>
<code>parseFloat(string)</code> 函数	将string类型转成浮点数数值型	<code>parseFloat('78.21')</code>
<code>Number()</code> 强制转换函数	将string类型转换为数值型	<code>Number('12')</code>
js 隐式转换( - * / )	利用算术运算隐式转换为数值型	<code>'12' - 0</code>

- 注意 `parseInt` 和 `parseFloat` 单词的大小写，这 2 个是重点
- 隐式转换是我们在进行算数运算的时候，JS 自动转换了数据类型

案例 2：计算年龄

此网页显示

请输入您的出生年份：

确定

取消

此案例要求在页面中弹出一个输入框，我们输入出生年份后， 能计算出我们的年龄。

案例分析

①弹出一个输入框 (prompt)，让用户输入出生年份 （用户输入）

②把用户输入的值用变量保存起来，然后用今年的年份减去变量值，结果就是现在的年龄（程序内部处理）

③弹出警示框（alert），把计算的结果输出（输出结果）

案例代码

// 1. 弹出输入框，输入出生年份，并存储在变量中

```
let year = prompt('请输入您的出生年份: '); // 用户输入
```

// 2. 用今年减去刚才输入的年份

```
let result = 2023 - year; // 程序内部处理
```

// 3. 弹出提示框

```
alert('您的年龄是:' + result + '岁'); // 输出结果
```

案例 3:简单加法器

计算两个数的值，用户输入第一个值后，继续弹出第二个输入框并输入第二个值，最后通过弹出窗口显示出两次输入值相加的结果。

案例分析

①先弹出第一个输入框，提示用户输入第一个值 保存起来

②再弹出第二个框，提示用户输入第二个值 保存起来

③把这两个值相加，并将结果赋给新的变量（注意数据类型转换）

④弹出警示框（alert），把计算的结果输出（输出结果）

案例代码

// 1. 先弹出第一个输入框，提示用户输入第一个值

```
let num1 = prompt('请输入第一个值: ');
```

// 2. 再弹出第二个框，提示用户输入第二个值



```
let num2 = prompt('请输入第二个值: ');
```

// 3. 将输入的值转换为数字型后，把这两个值相加，并将结果赋给新的变量

```
let result = parseFloat(num1) + parseFloat(num2);
```

// 4. 弹出结果

```
alert('结果是:' + result);
```

## 5.4.4 转换为布尔型

方式	说明	案例
Boolean()函数	其他类型转成布尔值	Boolean('true');

- 代表空、否定的值会被转换为 false，如 ''、0、NaN、null、undefined
- 其余值都会被转换为 true

```
console.log(Boolean('')); // false
console.log(Boolean(0)); // false
console.log(Boolean(NaN)); // false
console.log(Boolean(null)); // false
console.log(Boolean(undefined)); // false
console.log(Boolean('小白')); // true
console.log(Boolean(12)); // true
```

## 5.5.1 标识符、关键字、保留字

### (1) 标识符

标识(zhi)符：就是指开发人员为变量、属性、函数、参数取的名字。

标识符不能是关键字或保留字。

### (2) 关键字

关键字：是指 JS 本身已经使用了的字，不能再用它们充当变量名、方法名。

包括: break、case、catch、continue、default、delete、do、else、finally、for、function、if、in、instanceof、new、return、switch、this、throw、try、typeof、let、void、while、with 等。

### (3) 保留字

保留字: 实际上就是预留的“关键字”, 意思是现在虽然还不是关键字, 但是未来可能会成为关键字, 同样不能使用它们当变量名或方法名。

包括: boolean、byte、char、class、const、debugger、double、enum、export、extends、final、float、goto、implements、import、int、interface、long、native、package、private、protected、public、short、static、super、synchronized、throws、transient、volatile 等。

注意: 如果将保留字用作变量名或函数名, 那么除非将来的浏览器实现了该保留字, 否则很可能收不到任何错误消息。当浏览器将其实现后, 该单词将被看做关键字, 如此将出现关键字错误。

## JS(2)

# JavaScript 运算符

## 1.运算符

---

运算符 (operator) 也被称为操作符, 是用于实现赋值、比较和执行算数运算等功能的符号。

JavaScript 中常用的运算符有:

- 算数运算符
- 递增和递减运算符
- 比较运算符
- 逻辑运算符
- 赋值运算符

## 2.算数运算符

---

### 2.1 算术运算符概述

概念: 算术运算使用的符号, 用于执行两个变量或值的算术运算。

运算符	描述	实例
+	加	10 + 20 = 30
-	减	10 - 20 = -10
*	乘	10 * 20 = 200
/	除	10 / 20 = 0.5
%	取余数(取模)	返回除法的余数 9 % 2 = 1

## 2.2 浮点数的精度问题

浮点数值的最高精度是 17 位小数，但在进行算术计算时其精确度远远不如整数。

```
let result = 0.1 + 0.2; // 结果不是 0.3, 而是: 0.30000000000000004
```

```
console.log(0.07 * 100); // 结果不是 7, 而是: 7.000000000000001
```

所以：不要直接判断两个浮点数是否相等！

## 2.3 小问题

(1)我们怎么判断 一个数能够被整除呢？

它的余数是 0 就说明这个数能被整除，这就是 % 取余运算符的主要用途

(2)请问 1 + (2 \* 3) 结果是？

结果是 7，注意算术运算符优先级的，先乘除，后加减，有小括号先算小括号里面的

## 2.4 表达式和返回值

表达式：是由数字、运算符、变量等以能求得数值的有意义排列方法所得的组合

简单理解：是由数字、运算符、变量等组成的式子

表达式最终都会有一个结果，返回给我们，我们成为返回值

## 3.递增和递减运算符

### 3.1 递增和递减运算符概述

如果需要反复给数字变量添加或减去 1，可以使用递增（++）和递减（--）运算符来完成。

在 JavaScript 中，递增（++）和递减（--）既可以放在变量前面，也可以放在变量后面。放在变量前面时，我们可以称为前置递增（递减）运算符，放在变量后面时，我们可以称为后置递增（递减）运算符。

注意：递增和递减运算符必须和变量配合使用。

## 3.2 递增运算符

### (1) 前置递增运算符

`++num` 前置递增，就是自加 1，类似于 `num = num + 1`，但是 `++num` 写起来更简单。

使用口诀：先自加，后返回值

```
let num = 10;

alert(++num + 10); // 21
```

### (2) 后置递增运算符

`num++` 后置递增，就是自加 1，类似于 `num = num + 1`，但是 `num++` 写起来更简单。

使用口诀：先返回原值，后自加

```
let num = 10;

alert(10 + num++); // 20
```

### 练习 1

```
let a = 10;
  ++a; // ++a 11  a=11
let b = ++a + 2; // ++a 12 a 12  b 14

console.log(b);

let c = 10;

c++; // c++ 10 c 11

let d = c++ + 2;

console.log(d); 12

let e = 10;

let f = e++ + ++e; // 1.e++ 10 e=11 2.++e 12 e 12
```

```
console.log(f); //f 22
```

### 3.3 前置递增和后置递增小结

- 前置递增和后置递增运算符可以简化代码的编写，让变量的值 + 1 比以前写法更简单
- 单独使用时，运行结果相同
- 与其他代码联用时，执行结果会不同
- 后置：先原值运算，后自加（先人后己）
- 前置：先自加，后运算（先己后人）
- 开发时，大多使用后置递增/减，并且代码独占一行，例如：num++; 或者 num--;

## 4.比较运算符

### 4.1 比较运算符概述

概念：比较运算符（关系运算符）是两个数据进行比较时所使用的运算符，比较运算后，会返回一个布尔值（true / false）作为比较运算的结果。

运算符名称	说明	案例	结果
<	小于号	1 < 2	true
>	大于号	1 > 2	false
>=	大于等于号 (大于或者等于)	2 >= 2	true
<=	小于等于号 (小于或者等于)	3 <= 2	false
==	判等号 (会转型)	37 == 37	true
!=	不等号	37 != 37	false
===    !==	全等 要求值和 数据类型都一致	37 === '37'	false

符号	作用	用法
=	赋值	把右边给左边
==	判断	判断两边值是否相等 (注意此时有隐式转换)
===	全等	判断两边的值和数据类型是否完全相同

```
console.log(18 == '18');
```

```
console.log(18 === '18');
```

## 5. 逻辑运算符

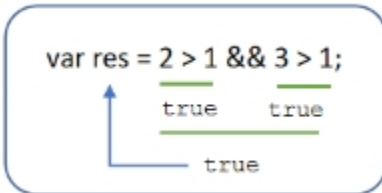
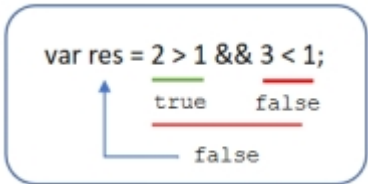
### 5.1 逻辑运算符概述

概念：逻辑运算符是用来进行布尔值运算的运算符，其返回值也是布尔值。后面开发中经常用于多个条件的判断

逻辑运算符	说明	案例
&&	"逻辑与", 简称 "与" and	true && false
	"逻辑或", 简称 "或" or	true    false
!	"逻辑非", 简称 "非" not	! true

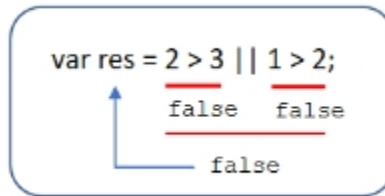
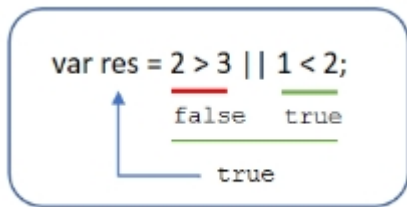
### 5.2 逻辑与 &&

两边都是 true 才返回 true, 否则返回 false



### 5.3 逻辑或 ||

两边都为 false 才返回 false, 否则都为 true



## 5.4 逻辑非 !

逻辑非 (!) 也叫作取反符, 用来取一个布尔值相反的值, 如 true 的相反值是 false

```
let isOk = !true;  
  
console.log(isOk); // false
```

练习 2

```
let num = 7;  
  
let str = "我爱你~中国~";  
  
console.log(num > 5 && str.length >= num);  
  
console.log(num < 5 && str.length >= num);  
  
console.log(!(num < 10));  
  
console.log(!(num < 10 || str.length == num) );
```

## 5.4 短路运算 (逻辑中断)

短路运算的原理: 当有多个表达式 (值) 时, 左边的表达式值可以确定结果时, 就不再继续运算右边的表达式的值;

### 5.4.1 逻辑与

- 语法: 表达式 1 && 表达式 2
- 如果第一个表达式的值为真, 则返回表达式 2
- 如果第一个表达式的值为假, 则返回表达式 1

```
console.log( 123 && 456 ); // 456
```

```
console.log( 0 && 456 );    // 0

console.log( 123 && 456 && 789 ); // 789
```

## 5.4.2 逻辑或

- 语法：表达式 1 || 表达式 2
- 如果第一个表达式的值为真，则返回表达式 1
- 如果第一个表达式的值为假，则返回表达式 2

```
console.log( 123 || 456 );    // 123

console.log( 0 || 456 );      // 456

console.log( 123 || 456 || 789 ); // 123
```

## 5.4.3 逻辑中断（短路操作）

```
let num = 0;

console.log(123 || num++);

console.log(num);
```

# 6. 赋值运算符

概念：用来把数据赋值给变量的运算符。

赋值运算符	说明	案例
=	直接赋值	let <u>userNamer</u> = '我是值';
+=、-=	加、减一个数后再赋值	let age = 10;age+=5;//15
*=、/=、%=	乘、除、取模后再赋值	let age = 2;age*=5;//10

```
let age = 10;

age += 5; // 相当于 age = age + 5;

age -= 5; // 相当于 age = age - 5;

age *= 10; // 相当于 age = age * 10;
```



# 7.运算符优先级

优先级	运算符	顺序
1	小括号	()
2	一元运算符	++ -- !
3	算数运算符	先 * / % 后 + -
4	关系运算符	> >= < <=
5	相等运算符	== != === !==
6	逻辑运算符	先 && 后
7	赋值运算符	=
8	逗号运算符	,

- 一元运算符里面的逻辑非优先级很高
- 逻辑与比逻辑或优先级高

## 练习 3

```
console.log( 4 >= 6 || '人' != '阿凡达' && !(12 * 2 == 144) && true)

let num = 10;

console.log( 5 == num / 2 && (2 + 2 * num).toString() === '22');
```

## 练习 4

```
let a = 3 > 5 && 2 < 7 && 3 == 4;

console.log(a);

let b = 3 <= 4 || 3 > 1 || 3 != 2;

console.log(b);

let c = 2 === "2";

console.log(c);

let d = !c || b && a ;

console.log(d);
```

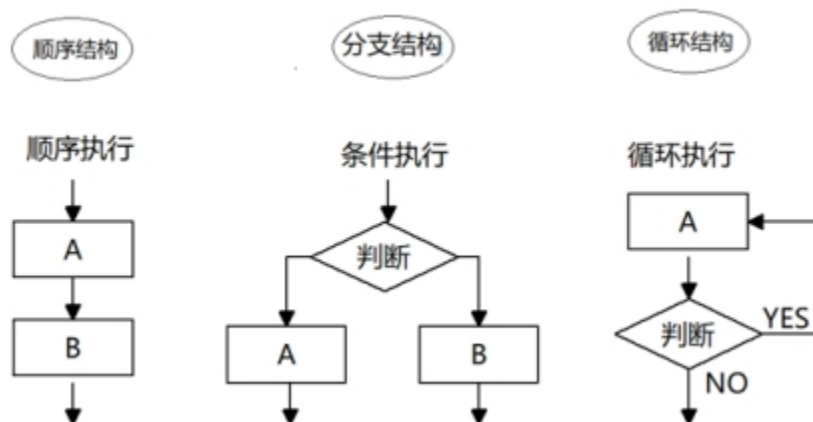
# JavaScript 流程控制-分支

## 1. 流程控制

在一个程序执行的过程中，各条代码的执行顺序对程序的结果是有直接影响的。很多时候我们要通过控制代码的执行顺序来实现我们要完成的功能。

简单理解：流程控制就是来控制我们的代码按照什么结构顺序来执行

流程控制主要有三种结构，分别是顺序结构、分支结构和循环结构，这三种结构代表三种代码执行的顺序。



## 2. 顺序流程控制

顺序结构是程序中最简单、最基本的流程控制，它没有特定的语法结构，程序会按照代码的先后顺序，依次执行，程序中大多数的代码都是这样执行的。

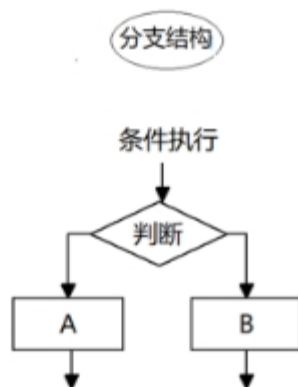


## 3.分支流程控制 if 语句

---

### 3.1 分支结构

由上到下执行代码的过程中，根据不同的条件，执行不同的路径代码（执行代码多选一的过程），从而得到不同的结果



## JS 语言提供了两种分支结构语句

---

- if 语句
- switch 语句

### 3.2 if 语句

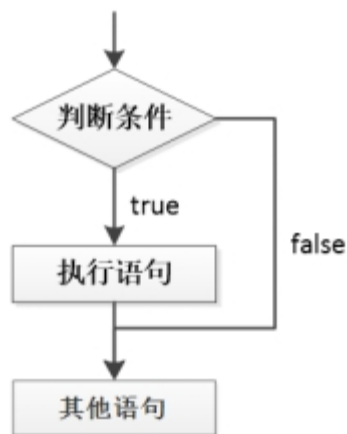
### 3.2.1 语法结构

// 条件成立执行代码，否则什么也不做

```
if (条件表达式) {  
  
    // 条件成立执行的代码语句  
  
}
```

语句可以理解为一个行为，循环语句和分支语句就是典型的语句。一个程序由很多个语句组成，一般情况下，会分割成一个一个的语句。

### 3.2.2 执行流程



案例 1:进入网吧

弹出一个输入框，要求用户输入年龄，如果年龄大于等于 16 岁，允许进网吧。

案例分析

弹出 prompt 输入框，用户输入年龄，程序把这个值取过来保存到变量中

使用 if 语句来判断年龄，如果年龄大于 16 就执行 if 大括号里面的输出语句

案例代码

```
let usrAge = prompt('请输入您的年龄: ');  
  
if(usrAge >= 16){  
  
    alert('您的年龄合法，欢迎来天际网吧享受学习的乐趣! ');  
}
```

```
}
```

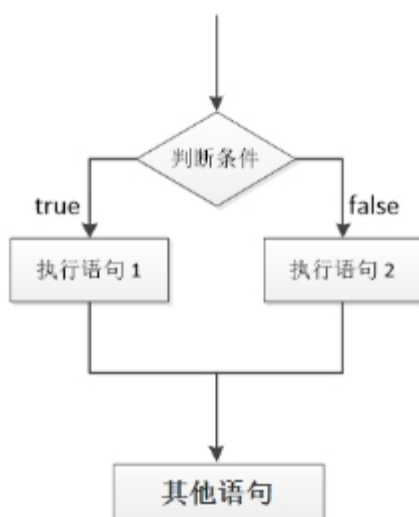
## 3.3 if else 语句（双分支语句）

### 3.3.1 语法结构

// 条件成立 执行 if 里面代码, 否则执行 else 里面的代码

```
if (条件表达式) {  
    // [如果] 条件成立执行的代码  
}  
else {  
    // [否则] 执行的代码  
}
```

### 3.3.2 执行流程



案例 2:判断闰年

接收用户输入的年份, 如果是闰年就弹出闰年, 否则弹出是平年

此网页显示

请您输入要检测的年份:

确定

取消

### 案例分析

- ①算法:能被 4 整除且不能整除 100 的为闰年(如 2004 年就是闰年,1901 年不是闰年)或者能够被 400 整除的就是闰年
- ②弹出 prompt 输入框, 让用户输入年份, 把这个值取过来保存到变量中
- ③使用 if 语句来判断是否是闰年, 如果是闰年, 就执行 if 大括号里面的输出语句, 否则就执行 else 里面的输出语句
- ④一定要注意里面的且 && 还有或者 || 的写法, 同时注意判断整除的方法是取余为 0

### 案例代码

```
if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0) {  
  
    alert("这个年份是闰年");  
  
} else { // 剩下的是平年  
  
    alert("这个年份是平年");  
  
}
```

### 案例 3: 判断是否中奖

接收用户输入的姓名, 来判断是否中奖, 如果输入的是韩梅梅, 则提示中了 5 块钱, 否则提示没有中奖。

### 案例分析

- ①弹出 prompt 输入框, 让用户输入姓名, 把这个值取过来保存到变量中
- ②使用 if 语句来判断是否存在这个姓名, 如果存在, 就执行 if 大括号里面的输出语句, 否则就执行 else 里面的输出语句
- ③一定要注意判断是否相等, 用 == 或者 ===

### 案例代码

```
// 算法 如果你叫韩梅梅 恭喜您中奖了, 否则没有中奖  
  
// 获得用户名
```

```
let username = prompt("请输入您的姓名: ");

if( username == "韩梅梅" ) {

    alert("恭喜发财");

} else {

    alert("谢谢惠顾,欢迎下次再来");

}
```

## 3.4 if else if 语句(多分支语句)

### 3.4.1 语法结构

```
// 适合于检查多重条件。

if (条件表达式 1) {

    语句 1;

} else if (条件表达式 2) {

    语句 2;

} else if (条件表达式 3) {

    语句 3;

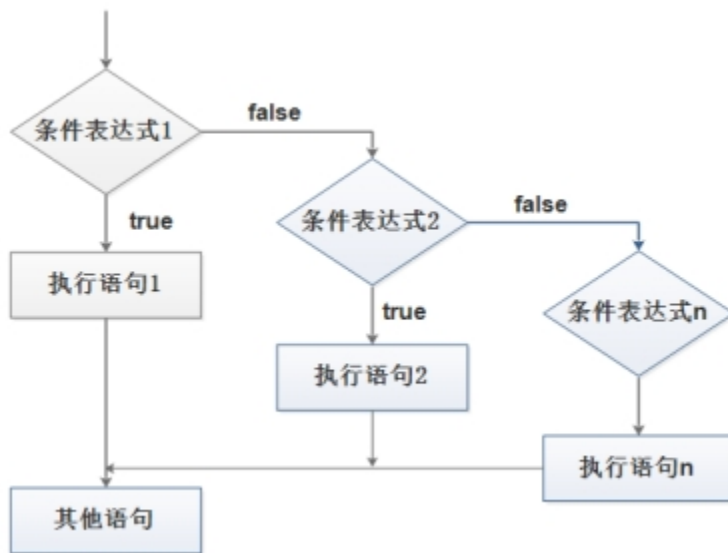
    ....

} else {

    // 上述条件都不成立执行此处代码

}
```

### 3.4.2 执行流程



#### 案例 4: 判断成绩级别

要求: 接收用户输入的分数, 根据分数输出对应的等级字母 A、B、C、D、E。

其中:

- (1) 90 分(含)以上 , 输出: A
- (2) 80 分(含)~ 90 分(不含), 输出: B
- (3) 70 分(含)~ 80 分(不含), 输出: C
- (4) 60 分(含)~ 70 分(不含), 输出: D
- (5) 60 分(不含) 以下, 输出: E

#### 案例分析

- ①按照从大到小判断的思路
- ②弹出 prompt 输入框, 让用户输入分数, 把这个值取过来保存到变量中
- ③使用多分支 if else if 语句来分别判断输出不同的值

#### 案例代码

```
let score = prompt('请您输入分数:');

if (score >= 90) {

    alert('宝贝, 你是我的骄傲');

} else if (score >= 80) {
```



```
alert('宝贝，你已经很出色了');

} else if (score >= 70) {

alert('你要继续加油喽');

} else if (score >= 60) {

alert('孩子，你很危险');

} else {

alert('熊孩子，我不想和你说话，我只想用鞭子和你说话');

}
```

## 4. 三元表达式

三元表达式也能做一些简单的条件选择。有三元运算符组成的式子称为三元表达式

### 4.1 语法结构

表达式 1 ? 表达式 2 : 表达式 3;

### 4.2 执行思路

- 如果表达式 1 为 true，则返回表达式 2 的值，如果表达式 1 为 false，则返回表达式 3 的值
- 简单理解：就类似于 if else（双分支）的简写

案例 5: 数字补 0

用户输入数字，如果数字小于 10，则在前面补 0，比如 01, 09，如果数字大于 10，则不需要补，比如 20。案例分析

- ①用户输入 0~59 之间的一个数字
- ②如果数字小于 10，则在这个数字前面补 0, (加 0) 否则 不做操作
- ③用一个变量接受这个返回值，输出

案例代码

```
let time = prompt('请您输入一个 0 ~ 59 之间的一个数字');
```

```
// 三元表达式 表达式 ? 表达式 1 : 表达式 2

let result = time < 10 ? '0' + time : time; // 把返回值赋值给一个变量

alert(result);
```

## 5. 分支流程控制 switch 语句

### 5.1 语法结构

switch 语句也是多分支语句，它用于基于不同的条件来执行不同的代码。当要针对变量设置一系列的特定值的选项时，就可以使用 switch。

```
switch( 表达式 ){

case value1:

    // 表达式 等于 value1 时要执行的代码

    break;

case value2:

    // 表达式 等于 value2 时要执行的代码

    break;

default:

    // 表达式 不等于任何一个 value 时要执行的代码

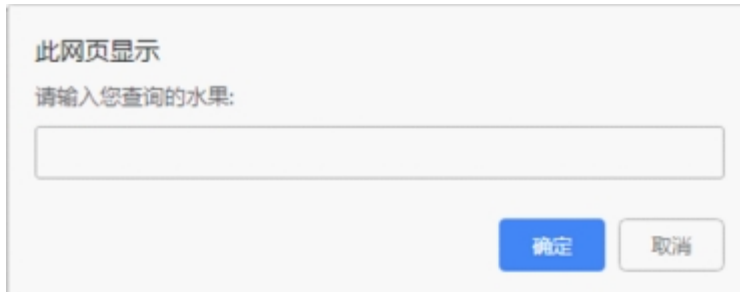
}
```

- switch：开关 转换，case：小例子 选项
- 关键字 switch 后面括号内可以是表达式或值，通常是一个变量
- 关键字 case，后跟一个选项的表达式或值，后面跟一个冒号
- switch 表达式的值会与结构中的 case 的值做比较
- 如果存在匹配全等(===)，则与该 case 关联的代码块会被执行，并在遇到 break 时停止，整个 switch 语句代码执行结束
- 如果所有的 case 的值都和表达式的值不匹配，则执行 default 里的代码

备注:执行 case 里面的语句时,如果没有 break,则继续执行下一个 case 里面的语句。

#### 案例 6:查询水果

用户在弹出框里面输入一个水果,如果有就弹出该水果的价格,如果没有该水果就弹出“没有此水果”。



#### 案例分析

- ①弹出 prompt 输入框,让用户输入水果名称,把这个值取过来保存到变量中。
- ②将这个变量作为 switch 括号里面的表达式。
- ③case 后面的值写几个不同的水果名称,注意一定要加引号,因为必须是全等匹配。
- ④弹出不同价格即可。同样注意每个 case 之后加上 break,以便退出 switch 语句。
- ⑤将 default 设置为没有此水果。

#### 案例代码

```
let fruit = prompt('请您输入查询的水果:');

switch (fruit) {

case '苹果':

alert('苹果的价格是 3.5/斤');

break;

case '榴莲':

alert('榴莲的价格是 35/斤');

break;

default:

alert('没有此水果');
```

```
}
```

## 5.2 switch 语句和 if else if 语句的区别

- ①一般情况下，它们两个语句可以相互替换
- ②switch...case 语句通常处理 case 为比较确定值的情况，而 if...else...语句更加灵活，常用于范围判断(大于、等于某个范围)
- ③switch 语句进行条件判断后直接执行到程序的条件语句，效率更高。而 if...else 语句有几种条件，就得判断多少次。
- ④当分支比较少时，if... else 语句的执行效率比 switch 语句高。
- ⑤当分支比较多时，switch 语句的执行效率比较高，而且结构更清晰。

# JS(3)

## JavaScript 流程控制-循环

### 1. 循环

目的:在实际问题中，有许多具有规律性的重复操作，因此在程序中要完成这类操作就需要重复执行某些语句

#### 1.1JS 中的循环

在 Js 中，主要有三种类型的循环语句：

- for 循环
- while 循环
- do...while 循环

### 2. for 循环

在程序中，一组被重复执行的语句被称之为循环体，能否继续重复执行，取决于循环的终止条件。由循环体及循环的终止条件组成的语句，被称之为循环语句

## 2.1 语法结构

for 循环主要用于把某些代码循环若干次，通常跟计数有关系。其语法结构如下：

```
for(初始化变量; 条件表达式; 操作表达式 ){  
  
    //循环体  
  
}
```

- 初始化变量：通常被用于初始化一个计数器，该表达式可以使用 let 关键字声明新的变量，这个变量帮我们来记录次数。
- 条件表达式：用于确定每一次循环是否能被执行。如果结果是 true 就继续循环，否则退出循环。
- 操作表达式：每次循环的最后都要执行的表达式。通常被用于更新或者递增计数器变量。当然，递减变量也是可以的。

```
for( 初始化变量; 条件表达式; 操作表达式 ){  
  
    //循环体语句  
  
}
```

执行过程：

- (1) 初始化变量，初始化操作在整个 for 循环只会执行一次。
- (2) 执行条件表达式，如果为 true，则执行循环体语句，否则退出循环，循环结束。
- (3) 执行操作表达式，此时第一轮结束。
- (4) 第二轮开始，直接去执行条件表达式（不再初始化变量），如果为 true，则去执行循环体语句，否则退出循环。
- (5) 继续执行操作表达式，第二轮结束。
- (6) 后续跟第二轮一致，直至条件表达式为假，结束整个 for 循环。

断点调试：

断点调试是指自己在程序的某一行设置一个断点，调试时，程序运行到这一行就会停住，然后你可以一步一步往下调试，调试过程中可以看各个变量当前的值，出错的话，调试到出错的代码行即显示错误，停下。

断点调试可以帮我们观察程序的运行过程

浏览器中按 F12--> sources --> 找到需要调试的文件-->在程序的某一行设置断点(刷新网页)

Watch: 监视，通过 watch 可以监视变量的值的变化，非常的常用。

代码调试的能力非常重要，只有学会了代码调试，才能学会自己解决 bug 的能力。初学者不要觉得调试代码麻烦就不去调试，知识点花点功夫肯定学的会，但是代码调试这个东西，自己不去练，永远都学不会。

## 2.2 for 循环重复相同的代码

for 循环可以重复相同的代码，比如我们要输出 10 句“修勾好可爱”

```
// 基本写法

for(let i = 1; i <= 10; i++){

  console.log('修勾好可爱~');

}

// 用户输入次数

let num = prompt('请输入次数:');

for ( let i = 1; i <= num; i++) {

  console.log('修勾好可爱~');

}
```

## 2.3 for 循环重复不相同的代码

for 循环还可以重复不同的代码，这主要是因为使用了计数器，计数器在每次循环过程中都会有变化。

例如，求输出一个人 1 到 100 岁：

```
// 基本写法

for (let i = 1; i <= 100; i++) {

  console.log('这个人今年' + i + '岁了');

}
```

## 2.4 for 循环重复某些相同操作

for 循环因为有了计数器的存在，我们还可以重复的执行某些操作，比如做一些算术运算。

案例 1: 求 1-100 之间所有整数的累加和

案例分析:

- ①需要循环 100 次，我们需要一个计数器 i
- ②我们需要一个存储结果的变量 sum，但是初始值一定是 0
- ③核心算法:  $1 + 2 + 3 + 4 \dots$  ,  $sum = sum + i$ ;

案例代码

```
let sum = 0;

for(let i = 1; i <= 100; i++){

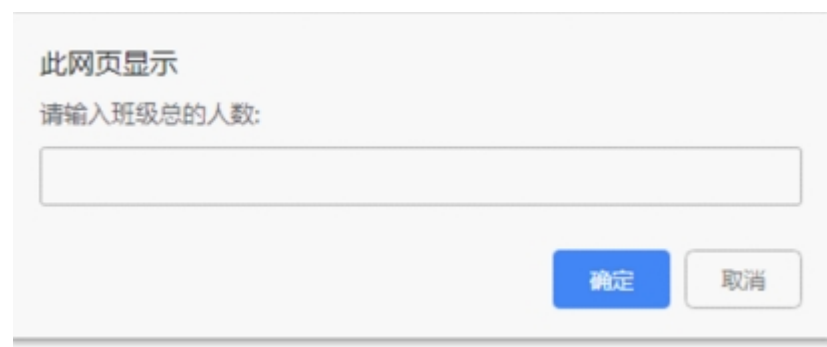
    sumNum += i;

}

console.log('1-100 之间整数的和 = ' + sum);
```

案例 2: 求学生成绩

要求用户输入班级人数，之后依次输入每个学生的成绩，最后打印出该班级总的的成绩以及平均成绩。



案例分析

- ①弹出输入框输入总的班级人数 ( num )
- ②依次输入学生的成绩 (保存起来 score)，此时我们需要用到 for 循环，弹出的次数跟班级总人数有关系 条件表达式  $i \leq num$
- ③进行业务处理: 计算成绩。先求总成绩 (sum)，之后求平均成绩 (average)
- ④弹出结果

案例代码

```
let num = prompt('请输入班级总的人数:'); // num 班级总的人数

let sum = 0; // 总成绩

let average = 0; // 平均成绩

for (let i = 1; i <= num; i++) {

    let score = prompt('请输入第' + i + '个学生的成绩');

    sum = sum + parseFloat(score);

}

average = sum / num;

alert('班级总的成績是: ' + sum);

alert('班级总的平均成績是: ' + average);
```

## 3. 双重 for 循环

### 3.1 双重 for 循环概述

很多情况下, 单层 for 循环并不能满足我们的需求, 比如我们要打印一个 5 行 5 列的图形、打印一个倒 直角三角形等, 此时就可以通过循环嵌套来实现。



```
☆☆☆☆☆
☆☆☆☆☆
☆☆☆☆☆
☆☆☆☆☆
☆☆☆☆☆

☆☆☆☆☆
☆☆☆☆
☆☆☆☆
☆☆☆
☆☆
☆☆
☆☆
☆☆
```

循环嵌套是指在一个循环语句中再定义一个循环语句的语法结构, 例如在 for 循环语句中, 可以再嵌套一个 for 循环, 这样的 for 循环语句我们称之为双重 for 循环。

### 3.2 双重 for 循环语法

```
for (外循环的初始; 外循环的条件; 外循环的操作表达式) {
```



```
for (内循环的初始; 内循环的条件; 内循环的操作表达式) {
```

需执行的代码;

```
}
```

```
}
```

- 内层循环可以看做外层循环的语句
- 内层循环执行的顺序也要遵循 for 循环的执行顺序
- 外层循环执行一次，内层循环要执行全部次数

### 3.3 打印五行五列星星



```
☆☆☆☆☆
☆☆☆☆☆
☆☆☆☆☆
☆☆☆☆☆
☆☆☆☆☆
```

思路：

- 内层循环负责一行打印五个星星
- 外层循环负责打印五行

代码实现

```
let star = '';

for (let j = 1; j <= 5; j++) {

  for (let i = 1; i <= 5; i++) {

    star += '☆'

  }

  // 每次满 5 个星星 就 加一次换行

  star += '\n'

}
```

```
console.log(star);
```

## 3.4 for 循环小结

- for 循环可以重复执行某些相同代码
- for 循环可以重复执行些许不同的代码，因为我们有计数器
- for 循环可以重复执行某些操作，比如算术运算符加法操作
- 随着需求增加，双重 for 循环可以做更多、更好看的效果
- 双重 for 循环，外层循环一次，内层 for 循环全部执行
- for 循环是循环条件和数字直接相关的循环
- 分析要比写代码更重要
- 一些核心算法想不到，但是要学会，分析它执行过程
- 举一反三，自己经常总结，做一些相似的案例

## 4. while 循环

while 语句可以在条件表达式为真的前提下，循环执行指定的一段代码，直到表达式不为真时结束循环。

while 语句的语法结构如下：

```
while (条件表达式) {  
  
    // 循环体代码  
  
}
```

执行思路：

①先执行条件表达式，如果结果为 true，则执行循环体代码；如果为 false，则退出循环，执行后面代码

②执行循环体代码

③循环体代码执行完毕后，程序会继续判断执行条件表达式，如条件仍为 true，则会继续执行循环体，直到循环条件为 false 时，整个循环过程才会结束

while 语句可以在条件表达式为真的前提下，循环执行指定的一段代码，直到表达式不为真时结束循环。

while 语句的语法结构如下：

```
while (条件表达式) {  
  
    // 循环体代码  
  
}
```

备注

①使用 while 循环时一定要注意到，它必须要有退出条件，否则会成为死循环

②while 循环和 for 循环的不同之处在于 while 循环可以做较为复杂的条件判断，比如判断用户名和密码

小练习 1: 询问你喜欢喝橙汁吗

弹出一个提示框，你喜欢喝橙汁吗？如果输入我喜欢喝橙汁，就提示结束，否则，一直询问。

## 5. do while 循环

do... while 语句其实是 while 语句的一个变体。该循环会先执行一次代码块，然后对条件表达式进行判断，如果条件为真，就会重复执行循环体，否则退出循环。

do... while 语句的语法结构如下：

```
do {  
  
    // 循环体代码 - 条件表达式为 true 时重复执行循环体代码  
  
} while(条件表达式);
```

执行思路：

①先执行一次循环体代码

②再执行条件表达式，如果结果为 true，则继续执行循环体代码，如果为 false，则退出循环，继续执行后面代码

备注

先执行循环体，再判断，我们会发现 do...while 循环语句至少会执行一次循环体代码

小练习 2: 询问你喜欢吃鸡公煲吗？

弹出一个提示框，你喜欢吃鸡公煲吗？如果输入我喜欢吃鸡公煲，就提示结束，否则，一直询问。

案例分析

①弹出输入框，要求用户输入。

②判断条件我们使用 do...while 循环。

③do... while 循环语句中的条件表达式只要输入的不是我喜欢吃鸡公煲，就一直循环。

案例代码

```
do {  
  
let love = prompt('你喜欢吃鸡公煲吗? ');  
  
} while (love !== '我喜欢吃鸡公煲')  
  
alert('交个朋友吧, 我也喜欢吃鸡公煲');
```

## 循环小结

- JS 中循环有 for 、while 、 do while
- 三个循环很多情况下都可以相互替代使用
- 如果是用来计次数，跟数字相关的，三者使用基本相同，但是我们更喜欢用 for
- while 和 do...while 可以做更复杂的判断条件，比 for 循环灵活一些
- while 和 do...while 执行顺序不一样，while 先判断后执行，do...while 先执行一次，再判断执行
- while 和 do...while 执行次数不一样，do...while 至少会执行一次循环体，而 while 可能一次也不执行
- 实际工作中，我们更常用 for 循环语句，它写法更简洁直观， 所以这个要重点学习

## 6.continue 和 break

### 6.1 continue 关键字

continue 关键字用于立即跳出本次循环，继续下一次循环（本次循环体中 continue 之后的代码就会少执行一次）。

例如，吃 5 个包子，第 3 个有虫子，就扔掉第 3 个，继续吃第 4 个第 5 个包子，其代码实现如下：

```
for (let i = 1; i <= 5; i++) {  
  
if (i == 3) {  
  
console.log('这个包子有虫子, 扔掉');
```

```
    continue; // 跳出本次循环, 跳出的是第 3 次循环

}

console.log('我正在吃第' + i + '个包子呢');

}
```

## 6.2 break 关键字

break 关键字用于立即跳出整个循环（循环结束）。

例如, 吃 5 个包子, 吃到第 3 个发现里面有半个虫子, 其余的不吃了, 其代码实现如下:

```
for (let i = 1; i <= 5; i++) {

    if (i == 3) {

        break; // 直接退出整个 for 循环, 跳到整个 for 下面的语句

    }

    console.log('我正在吃第' + i + '个包子呢');

}
```

# JavaScript 命名规范以及语法格式

## 1. 标识符命名规范

- 变量、函数的命名必须要有意义
- 变量的名称一般用名词
- 函数的名称一般用动词

## 2. 操作符规范

// 操作符的左右两侧各保留一个空格

```
for (let i = 1; i <= 5; i++) {
```

```
if (i == 3) {  
  
    break; // 直接退出整个 for 循环, 跳到整个 for 循环下面的语句  
  
}  
  
console.log('我正在吃第' + i + '个包子呢');  
  
}
```

### 3. 单行注释规范

```
for (let i = 1; i <= 5; i++) {  
  
    if (i == 3) {  
  
        break; // 单行注释前面注意有个空格  
  
    }  
  
    console.log('我正在吃第' + i + '个包子呢');  
  
}
```

## JS(4)

## JavaScript 数组

### 1. 数组的概念

---

(1)之前学习的数据类型, 只能存储一个值。如果我们想存储班级中所有学生的姓名, 那么该如何存储呢?

可以使用数组(Array)。数组可以把一组相关的数据一起存放, 并提供方便的访问(获取)方式。

(2)什么是数组呢?

数组是指一组数据的集合, 其中的每个数据被称作元素, 在数组中可以存放任意类型的元素。数组是一种将一组数据存储在一个变量名下的优雅方式。

// 普通变量一次只能存储一个值

```
let num = 10;
```

// 数组一次可以存储多个值

```
let arr = [1,2,3,4,5];
```

## 2. 创建数组

### 2.1 数组的创建方式

JS 中创建数组有两种方式：

- 利用 new 创建数组
- 利用数组字面量创建数组

### 2.2 利用 new 创建数组

```
let 数组名 = new Array();
```

```
let arr = new Array(); // 创建一个新的空数组
```

### 2.3 利用数组字面量创建数组

//1. 使用数组字面量方式创建空的数组

```
let 数组名 = [];
```

//2. 使用数组字面量方式创建带初始值的数组

```
let 数组名 = ['小白','小黑','大黄','瑞奇'];
```

- 数组的字面量是方括号 [ ]
- 声明数组并赋值称为数组的初始化
- 这种字面量方式也是我们以后最多使用的方式

### 2.4 数组元素的类型

数组中可以存放任意类型的数据，例如字符串，数字，布尔值等。

```
let arrStus = ['小白',12,true,28.9];
```

## 3. 获取数组元素

### 3.1 数组的索引

索引（下标）：用来访问数组元素的序号（数组下标从 0 开始）。

```
let arr = ['小白', '小黑', '大黄', '瑞奇'];
```

数组可以通过索引来访问、设置、修改对应的数组元素，我们可以通过“数组名[索引]”的形式来获取数组中的元素。

这里的访问就是获取得到的意思

// 定义数组

```
let arrStus = [1,2,3];
```

// 获取数组中的第 2 个元素

```
alert(arrStus[1]);
```

## 4. 遍历数组

数组中的每一项我们怎么取出来？

可以通过“数组名[索引号]”的方式一项项的取出来。

```
let arr = ['red', 'green', 'blue'];
```

```
console.log(arr[0]) // red
```

```
console.log(arr[1]) // green
```

```
console.log(arr[2]) // blue
```

遍历：就是把数组中的每个元素从头到尾都访问一次（类似我们每天早上学生的点名）。

我们可以通过 for 循环索引遍历数组中的每一项

```
let arr = ['red', 'green', 'blue'];
```

```
for(let i = 0; i < arr.length; i++){
```

```
console.log(arrStus[i]);
```

```
}
```



## 4.1 数组的长度

使用“数组名.length”可以访问数组元素的数量（数组长度）。

```
let arrStus = [1,2,3];  
  
alert(arrStus.length); // 3
```

备注

- ①此处数组的长度是数组元素的个数，不要和数组的索引号混淆。
- ②当我们数组里面的元素个数发生了变化，这个 length 属性跟着一起变化。

案例 1: 数组求和及平均值

求数组 [2,6,1,7, 4] 里面所有元素的和以及平均值。

案例分析

- ①声明一个求和变量 sum。
- ②遍历这个数组，把里面每个数组元素加到 sum 里面。
- ③用求和变量 sum 除以数组的长度就可以得到数组的平均值。

案例代码

```
let arr = [2, 6, 1, 7, 4];  
  
let sum = 0;  
  
let average = 0;  
  
for (let i = 0; i < arr.length; i++) {  
  
    sum += arr[i];  
  
}  
  
average = sum / arr.length;  
  
console.log('这组数的和是: ' + sum);  
  
console.log('这组数的平均值是: ' + average);
```

案例 2: 数组转换为字符串

要求: 将数组 ['red', 'green', 'blue', 'pink'] 里面的元素转换为字符串

输出: 'redgreenbluepink'

案例分析

- 1.思路: 就是把里面的元素相加就好了, 但是注意保证是字符相加。
- 2.遍历原来的数组, 分别把里面数据取出来, 加到字符串变量 `str` 里面。

案例代码

```
let arr = ['red', 'green', 'blue', 'pink'];

let str = '';

for (let i = 0; i < arr.length; i++) {

  str += arr[i];

}

console.log(str);
```

## 5. 数组中新增元素

### 5.1 通过修改 `length` 长度新增数组元素

- 可以通过修改 `length` 长度来实现数组扩容的目的
- `length` 属性是可读写的

```
let arr = ['red', 'green', 'blue', 'pink'];

arr.length = 7;

console.log(arr);

console.log(arr[4]);

console.log(arr[5]);

console.log(arr[6]);
```

其中索引号是 4, 5, 6 的空间没有给值, 就是声明变量未给值, 默认值就是 `undefined`。

```
▶ (7) ["red", "green", "blue", "pink", empty × 3]
undefined
undefined
undefined
```

## 5.2 通过修改数组索引新增数组元素

- 可以通过修改数组索引的方式追加数组元素
- 不能直接给数组名赋值，否则会覆盖掉以前的数据

```
let arr = ['red', 'green', 'blue', 'pink'];

arr[4] = 'hotpink';

console.log(arr);
```

这种方式也是我们最常用的一种方式。

案例 3: 数组新增元素

新建一个数组，里面存放 10 个整数（1~10），要求使用循环追加的方式输出：[1,2,3,4,5,6,7,8,9,10]

案例分析

- ①使用循环来追加数组。
- ②声明一个空数组 arr。
- ③循环中的计数器 i 可以作为数组元素存入。
- ④由于数组的索引号是从 0 开始的，因此计数器从 0 开始更合适，存入的数组元素要+1。

案例代码

```
let arr = [];

for (let i = 0; i < 10; i++) {

  arr[i] = i + 1;

}

console.log(arr);
```

## 6. 数组案例

### 案例 4: 删除指定数组元素

要求: 将数组[2, 0, 6, 1, 77, 0, 52, 0, 25, 7]中的 0 去掉后, 形成一个不包含 0 的新数组。

#### 案例分析

- ①需要一个新数组用于存放筛选之后的数据。
- ②遍历原来的数组,把不是 0 的数据添加到新数组里面(此时要注意采用数组名+索引的格式接收数据)。
- ③新数组里面的个数,用 length 不断累加。

#### 案例代码

```
let arr = [2, 0, 6, 1, 77, 0, 52, 0, 25, 7];

let newArr = []; // 空数组的默认的长度为 0

// 定义一个变量 i 用来计算新数组的索引号

for (let i = 0; i < arr.length; i++) {

  if (arr[i] !== 0) {

    // 给新数组

    // 每次存入一个值, newArr 长度都会 +1

    newArr[newArr.length] = arr[i];

  }

}

console.log(newArr);
```

#### 小练习:翻转数组

要求: 将数组 ['red', 'green', 'blue', 'pink', 'purple'] 的内容反过来存放。

输出: ['purple', 'pink', 'blue', 'green', 'red']

## 案例分析

核心： 把 arr 的 最后一个元素 取出来 给 新数组 作为第一个

```
['red', 'green', 'blue', 'pink', 'purple'];
```

```
['purple', 'pink', 'blue', 'green', 'red'];
```

把第 4 个 给 新数组 第 0 个

把第 3 个 给 新数组 第 1 个

把第 2 个 给 新数组 第 2 个

把第 1 个 给 新数组 第 3 个

把第 0 个 给 新数组 第 4 个

43210 是 长度(5) - 1

01234 正好就是

newArr.length

案例代码

```
let arr = ['red', 'green', 'blue', 'pink', 'purple'];

let newArr = [];

for (let i = 0; i < arr.length; i++) {

  // newArr 是接收方, arr 是输送方

  newArr[i] = arr[arr.length - i - 1];

}

console.log(newArr);
```

# JS(5)

## JavaScript 函数

### 1. 函数的概念

在 JS 里面,可能会定义非常多的相同代码或者功能相似的代码,这些代码可能需要大量重复使用。

虽然 for 循环语句也能实现一些简单的重复操作,但是比较具有局限性,此时我们就可以使用 JS 中的函数。

函数:就是封装了一段可被重复调用执行的代码块。通过此代码块可以实现大量代码的重复使用。

### 2. 函数的使用

函数在使用时分为两步:声明函数和调用函数。

#### 2.1 声明函数

```
// 声明函数

function 函数名() {

  //函数体代码
```

```
}
```

- `function` 是声明函数的关键字,必须小写
  - 由于函数一般是为了实现某个功能才定义的, 所以通常我们将函数名命名为动词, 比如 `getSum`
- 函数在使用时分为两步: 声明函数和调用函数。

## 2.2 调用函数

// 调用函数

函数名(); // 通过调用函数名来执行函数体代码

- 调用的时候千万不要忘记添加小括号
- 口诀: 函数不调用, 自己不执行。

备注:声明函数本身并不会执行代码, 只有调用函数时才会执行函数体代码。

## 2.3 函数的封装

- 函数的封装是把一个或者多个功能通过函数的方式封装起来, 对外只提供一个简单的函数接口
- 简单理解: 封装类似于将电脑配件整合组装到机箱中 ( 类似快递打包)

案例 1: 利用函数计算 1-100 之间的累加和

/\* 计算 1-100 之间值的函数 \*/

```
// 声明函数

function getSum(){

    let sumNum = 0; // 准备一个变量, 保存数字和

    for (let i = 1; i <= 100; i++) {

        sumNum += i; // 把每个数值 都累加 到变量中

    }

    alert(sumNum);

}
```

```
// 调用函数
```

```
getSum();
```

## 3. 函数的参数

### 3.1 形参和实参

在声明函数时，可以在函数名称后面的小括号中添加一些参数，这些参数被称为形参，而在调用该函数时，同样也需要传递相应的参数，这些参数被称为实参。

参数	说明
形参	形式上的参数 函数定义的时候 传递的参数 当前并不知道是什么
实参	实际上的参数 函数调用的时候传递的参数 实参是传递给形参的

参数的作用：在函数内部某些值不能固定，我们可以通过参数在调用函数时传递不同的值进去。

在声明函数时，可以在函数名称后面的小括号中添加一些参数，这些参数被称为形参，而在调用该函数时，同样也需要传递相应的参数，这些参数被称为实参。

```
// 带参数的函数声明
```

```
function 函数名(形参 1, 形参 2, 形参 3...) { // 可以定义任意多的参数, 用逗号分隔
```

```
    // 函数体
```

```
}
```

```
// 带参数的函数调用
```

```
函数名(实参 1, 实参 2, 实参 3...);
```

案例 2: 利用函数求任意两个数的和

```
function getSum(num1, num2) {
```

```
    console.log(num1 + num2);
```

```
}
```

```
getSum(1, 3); // 4
```



```
getSum(6, 5); // 11
```

## 3.2 函数参数的传递过程

```
// 声明函数

function getSum(num1, num2) {

  console.log(num1 + num2);

}

// 调用函数

getSum(1, 3); // 4

getSum(6, 5); // 11
```

- (1)调用的时候实参值是传递给形参的
- (2)形参简单理解为：不用声明的变量
- (3)实参和形参的多个参数之间用逗号(,) 分隔

## 3.3 函数形参和实参个数不匹配问题

参数个数	说明
实参个等于形参个数	输出正确结果
实参个数多于形参个数	只取到形参的个数
实参个数小于形参个数	多的形参定义为undefined，结果为NaN

//实参个数

```
function sum(num1, num2) {

  console.log(num1 + num2);

}
```

```
sum(100, 200);    // 形参和实参个数相等，输出正确结果

sum(100, 400, 500, 700); // 实参个数多于形参，只取到形参的个数

sum(200);        // 实参个数少于形参，多的形参定义为 undefined，结果为 NaN
```

备注:在 JavaScript 中，形参的默认值是 undefined。

## 3.4 小结

- 函数可以带参数也可以不带参数
- 声明函数的时候，函数名括号里面的是形参，形参的默认值为 undefined
- 调用函数的时候，函数名括号里面的是实参
- 多个参数中间用逗号分隔
- 形参的个数可以和实参个数不匹配，但是结果不可预计，我们尽量要匹配

## 4. 函数的返回值

### 4.1 return 语句

有的时候，我们会希望函数将值返回给调用者，此时通过使用 return 语句就可以实现。

return 语句的语法格式如下：

```
// 声明函数

function 函数名 () {

...

return 需要返回的值;

}
```

// 调用函数

函数名(); // 此时调用函数就可以得到函数体内 return 后面的值

- 在使用 return 语句时，函数会停止执行，并返回指定的值
- 如果函数没有 return，返回的值是 undefined

有的时候，我们会希望函数将值返回给调用者，此时通过使用 `return` 语句就可以实现。

例如，声明了一个 `sum()` 函数，该函数的返回值为 666，其代码如下：

```
// 声明函数

function sum () {

...

return 666;

}

// 调用函数

sum(); // 此时 sum 的值就等于 666, 因为 return 语句会把自身后面的值返回给调用者
```

案例 3: 利用函数求任意两个数的最大值

```
function getMax(num1, num2) {

return num1 > num2 ? num1 : num2;

}

console.log(getMax(1, 2));

console.log(getMax(11, 2));
```

案例 4: 利用函数求任意一个数组中的最大值

求数组 `[5,2,99,101,67,77]` 中的最大数值。

// 定义一个获取数组中最大数的函数

```
function getMaxFromArr(numArray){

let maxNum = 0;

for(let i =0;i < numArray.length;i++){

    if(numArray[i] > maxNum){

        maxNum = numArray[i];

    }

}
```

```
}

return maxNum;

}

let arrNum = [5,2,99,101,67,77];

let maxN = getMaxFromArr(arrNum); // 这个实参是个数组

alert('最大值为: ' + maxN);
```

## 4.2 return 终止函数

return 语句之后的代码不被执行。

```
function add(num1, num2){

//函数体

return num1 + num2; // 注意: return 后的代码不执行

alert('我不会被执行, 因为前面有 return');

}

let resNum = add(21,6); // 调用函数, 传入两个实参, 并通过 resNum 接收函数返回值

alert(resNum); // 27
```

## 4.3 return 的返回值

return 只能返回一个值。如果用逗号隔开多个值, 以最后一个为准。

```
function add(num1, num2){

//函数体

return num1, num2;

}
```

```
let resNum = add(21,6); // 调用函数, 传入两个实参, 并通过 resNum 接收函数返回值

alert(resNum);    // 6
```

案例 5: 创建一个函数, 实现两个数之间的加减乘除运算, 并将结果返回

```
let a = parseFloat(prompt('请输入第一个数'));

let b = parseFloat(prompt('请输入第二个数'));

function count(a, b) {

let arr = [a + b, a - b, a * b, a / b];

return arr;

}

let result = count(a, b);

console.log(result);
```

## 4.4 函数没有 return 返回 undefined

函数都是有返回值的

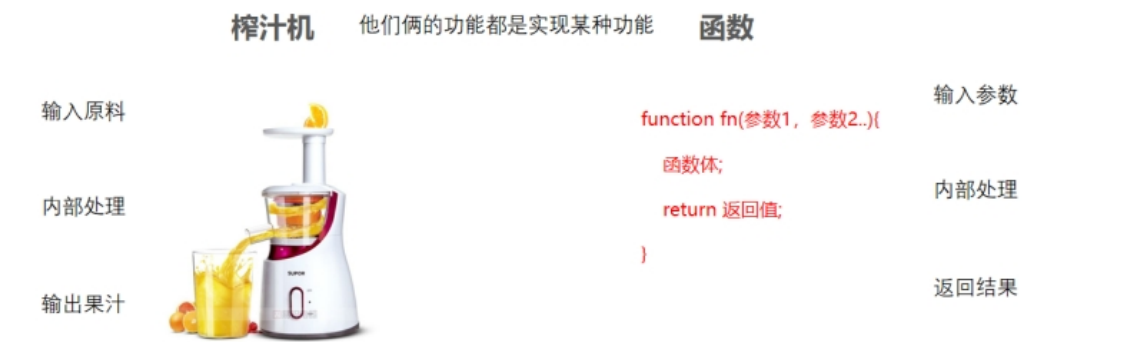
(1)如果有 return 则返回 return 后面的值

(2)如果没有 return 则返回 undefined

## 4.5 break ,continue ,return 的区别

- break : 结束当前的循环体 (如 for、while)
- continue : 跳出本次循环, 继续执行下次循环 (如 for、while)
- return : 不仅可以退出循环, 还能够返回 return 语句中的值, 同时还可以结束当前的函数体内的代码

## 4.6 巧妙看函数



## 5. arguments 的使用

当我们不确定有多少个参数传递的时候, 可以用 `arguments` 来获取。在 JavaScript 中, `arguments` 实际上它是当前函数的一个内置对象。所有函数都内置了一个 `arguments` 对象, `arguments` 对象中存储了传递的所有实参。

`arguments` 展示形式是一个伪数组, 因此可以进行遍历。伪数组具有以下特点:

- 具有 `length` 属性
- 按索引方式储存数据
- 不具有数组的 `push`, `pop` 等方法

案例 6: 利用函数求任意个数的最大值

```
function maxValue() {  
  
    let max = arguments[0];  
  
    for (let i = 0; i < arguments.length; i++) {  
  
        if (max < arguments[i]) {  
  
            max = arguments[i];  
  
        }  
  
    }  
  
    return max;  
  
}  
  
console.log(maxValue(2, 4, 5, 9));
```

```
console.log(maxValue(12, 4, 9));
```

## 6. 函数案例

案例 7: 利用函数封装方式, 翻转任意一个数组

```
function reverse(arr) {  
  
  let newArr = [];  
  
  for (let i = arr.length - 1; i >= 0; i--) {  
  
    newArr[newArr.length] = arr[i];  
  
  }  
  
  return newArr;  
  
}  
  
let arr1 = reverse([1, 3, 4, 6, 9]);  
  
console.log(arr1);
```

案例 8: 判断闰年

要求: 输入一个年份, 判断是否是闰年 (闰年: 能被 4 整除并且不能被 100 整数, 或者能被 400 整除)

```
function isRun(year) {  
  
  let flag = false;  
  
  if (year % 4 === 0 && year % 100 !== 0 || year % 400 === 0) {  
  
    flag = true;  
  
  }  
  
  return flag;  
  
}  
  
console.log(isRun(2010));
```

```
console.log(isRun(2012));
```

函数可以调用另外一个函数

因为每个函数都是独立的代码块，用于完成特殊任务，因此经常会用到函数相互调用的情况。

```
function fn1() {  
  
  console.log(111);  
  
  fn2();  
  
  console.log('fn1');  
  
}  
  
function fn2() {  
  
  console.log(222);  
  
  console.log('fn2');  
  
}  
  
fn1();
```

## 7. 函数的两种声明方式

### 1. 自定义函数方式(命名函数)

利用函数关键字 `function` 自定义函数方式。

// 声明定义方式

```
function fn() {...}
```

// 调用

```
fn();
```

- 因为有名字，所以也被称为命名函数
- 调用函数的代码既可以放到声明函数的前面，也可以放在声明函数的后面



## 2. 函数表达式方式(匿名函数)

利用函数表达式方式的写法如下：

// 这是函数表达式写法，匿名函数后面跟分号结束

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

// 调用的方式，函数调用必须写到函数体下面

```
fn();
```

- 因为函数没有名字，所以也被称为匿名函数
- 这个 fn 里面存储的是一个函数
- 函数表达式方式原理跟声明变量方式是一致的
- 函数调用的代码必须写到函数体后面

# JS 作用域

## 1. 作用域

---

### 1.1 作用域概述

通常来说，一段程序代码中所用到的名字并不总是有效和可用的，而限定这个名字的可用性的代码范围就是这个名字的作用域。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。

## JavaScript (es6 前) 中的作用域有两种：

---

- 全局作用域
- 局部作用域（函数作用域）

### 1.2 全局作用域

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。

### 1.3 局部作用域（函数作用域）

作用于函数内的代码环境，就是局部作用域。因为跟函数有关系，所以也称为函数作用域。

## 1.4 JS 没有块级作用域

- 块作用域由 { } 包括。

- 在其他编程语言中（如 java、c#等），在 if 语句、循环语句中创建的变量，仅仅只能在本 if 语句、本循环语句中使用，如下面的 Java 代码：

```
if(true){  
  
    int num = 123;  
  
    system.out.print(num); // 123  
  
}  
  
system.out.print(num); // 报错
```

JS 中没有块级作用域（在 ES6 之前）。

```
if(true){  
  
    let num = 123;  
  
    console.log(123); //123  
  
}  
  
console.log(123); //123
```

## 2. 变量的作用域

### 2.1 变量作用域的分类

在 JavaScript 中，根据作用域的不同，变量可以分为两种：

- 全局变量

- 局部变量

### 2.2 全局变量

在全局作用域下声明的变量叫做全局变量（在函数外部定义的变量）。

- 全局变量在代码的任何位置都可以使用
- 在全局作用域下 `let` 声明的变量 是全局变量
- 特殊情况下，在函数内不使用 `let` 声明的变量也是全局变量（不建议使用）

## 2.3 局部变量

在局部作用域下声明的变量叫做局部变量（在函数内部定义的变量）

- 局部变量只能在该函数内部使用
- 在函数内部 `let` 声明的变量是局部变量
- 函数的形参实际上就是局部变量

## 2.4 全局变量和局部变量的区别

- 全局变量：在任何一个地方都可以使用，只有在浏览器关闭时才会被销毁，因此比较占内存
- 局部变量：只在函数内部使用，当其所在的代码块被执行时，会被初始化；当代码块运行结束后，就会被销毁，因此更节省内存空间

## 3. 作用域链

- 只要是代码，就至少有一个作用域
- 写在函数内部的局部作用域
- 如果函数中还有函数，那么在这个作用域中又可以诞生一个作用域
- 根据在内部函数可以访问外部函数变量的这种机制，用链式查找决定哪些数据能被内部函数访问，就称作作用域链

案例 1:结果是几？

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

```

f2();

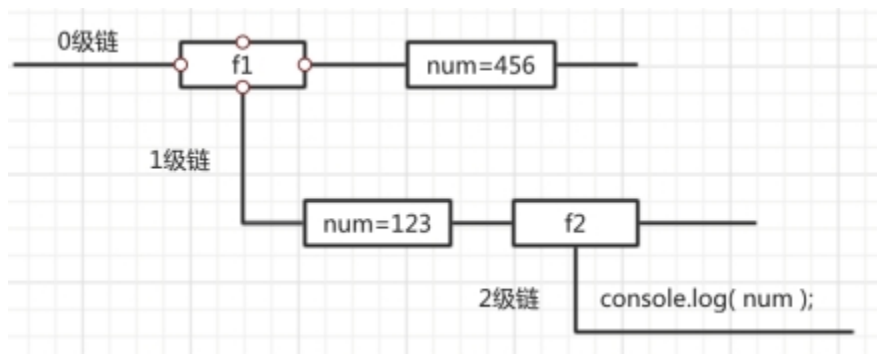
}

let num = 456;

f1();

```

案例分析



作用域链：采取就近原则的方式来查找变量最终的值。

案例 2：结果是几？

```

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();
```

# JavaScript 预解析

## 1. 预解析

JavaScript 代码是由浏览器中的 JavaScript 解析器来执行的。JavaScript 解析器在运行 JavaScript 代码的时候分为两步：预解析和代码执行。

- 预解析：在当前作用域下，JS 代码执行之前，浏览器会默认把带有 let 和 function 声明的变量在内存中进行提前声明或者定义。

- 代码执行：从上到下执行 JS 语句。

预解析只会发生在通过 let 定义的变量和 function 上。学习预解析能够让我们知道为什么在变量声明之前访问变量的值是 undefined，为什么在函数声明之前就可以调用函数。

## 2. 变量预解析和函数预解析

### 2.1 变量预解析（变量提升）

预解析也叫做变量、函数提升。

变量提升：变量的声明会被提升到当前作用域的最上面，变量的赋值不会提升。

### 2.2 函数预解析（函数提升）

函数提升：函数的声明会被提升到当前作用域的最上面，但是不会调用函数。

```
fn();

function fn() {
```

```
console.log('打印');
```

```
}
```