# 1. 基本语法

## 1.1 编写位置

```
<!-- 1.编写位置一: 编写在html内部(了解) -->
<a href="#" onclick="alert('百度一下')">百度一下</a>
<a href="javascript: alert('百度一下')">百度一下</a>

<!-- 2.编写位置二: 编写在script元素之内 -->
<a class="google" href="#">Google一下</a>

<script>
  var googleAEl = document.querySelector(".google")
  googleAEl.onclick = function() {
    alert("Google一下")
  }
</script>

<!-- 3.编写位置三: 独立的js文件 -->
<a class="bing" href="#">bing一下</a>
<script src="./js/bing.js"></script>
```

## 1.2 noscript标签的使用

> 当浏览器禁用script脚本的时候，noscript标签中的内容则会被渲染到界面中

```
<noscript>
  <h1>您的浏览器不支持JavaScript, 请打开或者更换浏览器~</h1>
</noscript>

<script>
  alert("您的浏览器正在运行JavaScript代码")
</script>
```

## 1.3 js脚本的加载顺序

> 浏览器解析**html**时，是从上往下依次解析，当遇到script脚本时，会阻塞**dom**的解析（该脚本没有加**defer**和**async**属性时），会去下载js脚本，当js脚本下载完并执行完后才会继续构建dom。一般script元素是
>
> 作为body元素的最后子元素的。

## 1.4 交互方式

> 1. alert
> 2. console.log
> 3. document.write
> 4. prompt

```
<script>
  // 1.交互方式一: alert函数
  alert("Hello World");

  // 2.交互方式二: console.log函数, 将内容输出到控制台中(console)
  // 使用最多的交互方式
  console.log("Hello Coderwhy");

  // 编写的JavaScript代码出错了
  // message.length

  // 3.交互方式三: document.write()
  document.write("Hello Kobe");

  // 4.交互方式四: prompt函数, 作用获取用户输入的内容
  var result = prompt("请输入你的名字: ");
  alert("您刚才输入的内容是:" + result);

</script>
```

## 1.5 注释

1. 单行注释
2. 多行注释
3. 文档注释

```
<script>
  // 1.单行注释

  // 2.多行注释
  /*
     我是一行注释
     我是另外一行注释
    */

  // 3.文档注释
  /**
     * 和某人打招呼
     * @param {string} name 姓名
     * @param {number} age 年龄
     */
  function sayHello(name, age) {
```

```
  }

  sayHello()
</script>
```

# 2. 变量和数据结构

## 2.1 变量

> 变量：变量可以看作存放数据的容器，可以想象成一个盒子。
>
> 补充: **js**中的变量可以存放任意类型的数据（在程序运行中）。变量拥有这种行为时，这种编程语言称为"**动态类型的编程语言**"
>
> 在计算机程序中，数据需要存储到变量当中。计算机程序 = 数据结构 + 算法；数据结构相当于数据，算法相当于处理数据的一些逻辑。

## 2.2 变量声明

> 变量必须先声明，后使用（但js中是有部分不同的）
>
> 使用var 、let、const 关键字进行声明

```
<script>

  // 定义一个变量
  // 第一步： 变量的声明（告诉js引擎接下来我要定义一个变量）
  // var关键字
  // 第二步: 变量的赋值(使用=赋值即可)
  // var currentTime = "16:00"

  // 其他的写法一:
  // var currentTime;
  // currentTime = "16:02";
  // currentTime = "17:00";

  // 其他的写法二: 同时声明多个变量(不推荐, 阅读性比较差)
  // var name, age, height
  // name = "why"
  // age = 18
  // height = 1.88
  var name = "why", age = 18, height = 1.88;

  // 补充:
  // 1.当我们打印变量时, 实际上是在打印变量中保存的值
  // 2.console.log(参数1, 参数2, 参数3..........)
  console.log(name, age, height);

</script>
```

## 2.3 变量的命名规则和命名规范

### 命名规则

> 命名规则是必须要遵守的：表示在声明变量时，我们需要给这个变量（盒子）起一个名字，后续访问的时候可以根据这个变量名来访问。

- 变量名以**数字**、**字母**、**美元符号（$）**、**下划线（_）**组成
- 不能以**数字**开头
- 不能使用**关键字**和**保留字**作为变量名
- 严格区分大小写

### 命名规范

> 命名规范只是大家普遍认定比较好的约束，建议遵守

- 多个单词使用驼峰标识；
- 赋值 = 两边都加上一个空格；
- 一条语句结束后加上分号; 也有很多人的习惯是不加；
- 变量应该做到见名知意；

```
<script>
  // 规则: 必须遵守

  // 规范: 建议遵守
  // 1.多个单词, 建议使用驼峰标识(小驼峰)
  // 小驼峰(currentTime)/大驼峰(CurrentTime)
  // 2.推荐=的左右两边加上空格
  var currentTime = "16:18"
  var currentPrice = "¥100"

</script>
```

## 2.4 变量使用注意事项

- 如果一个变量未声明（declaration）就直接使用，那么会报错；
- 如果一个变量有声明，但是没有赋值，那么默认值是undefined
- 如果没有使用var声明变量也可以，但是不推荐（事实上会被添加到window对象上）

```
<script>
  // 注意事项一: 如果一个变量未声明就直接使用, 那么会直接报错
  // var currentAge = age
  // var name = "why"
  // console.log(message) // 浏览器报错

  // 注意事项二: 如果一个变量有声明, 但是没有赋值操作
```

```
  // var info
  // console.log(info) // undefined

  // 注意事项三: 在JavaScript中也可以不使用var在全局声明一个变量(不推荐)
  // 如果不使用var来声明一个变量, 也是可以声明成功的, 并且这个变量会被加入window对象
  address = "广州市"
  console.log(address)

</script>
```

## 2.5 数据类型

> 什么是数据类型，js中的值都具有特定的类型。不同的数据类型有不同的用途

### js中的数据类型

- Number
- String
- Boolean
- Undefined
- Null
- Symbol
- BigInt
- Object

### typeof 运算符

> 因为ES的类型系统是松散的，所以需要一种手段来确定任意变量的数据类型。**typeof**运算符就运应而生了

### 对一个值使用 typeof 操作符会返回下列字符串之一

- "undefined"表示值未定义
- "boolean"表示值为布尔值
- "string"表示值为字符串
- "number"表示值为数值
- "object"表示值为对象(而不是函数)或 null
- "function"表示值为函数
- “symbol”表示值为符号

## Number类型

> 数字类型的值，表示整数和小数

- 基本使用

```
// 1.Number类型的基本使用
var age = 18
var height = 1.88
```

- 特殊值

```
// 2.特殊的数值
// Infinity 表示正无穷大，-Infinity表示负无穷大
var num1 = Infinity
var num2 = 1 / 0
console.log(num1, num2)
```

- NaN 不是一个数字

> typeof NaN 为 'number'，isNaN函数用来检测一个值是不是NaN

```
// NaN: not a number(不是一个数字)
var result = 3 * "abc"
console.log(result) // NaN
console.log(isNaN(result)) // true
```

- 进制表示

```
// 3.进制表示
var num3 = 100 // 十进制
// 了解
var num4 = 0x100 // 十六进制
var num5 = 0o100 // 八进制
var num6 = 0b100 // 二进制
console.log(num4, num5, num6) // 256 64 4
```

- 值表示范围

```
// 5.数字可以表示的范围
var max = Number.MAX_VALUE
var min = Number.MIN_VALUE
console.log(max, min) // 1.7976931348623157e+308 5e-324
```

- 运算

> 数字类型可以值可以做各种运算
>
> 例如：+ - * / % **等

## String类型

> 在开发中我们经常会有一些文本需要表示，这个时候我们会使用字符串String
>
> 例如：人的姓名：dd。地址：德阳市。简介：认真是一种可怕的力量

- **JavaScript 中的字符串必须被括在引号里，有三种包含字符串的方式。**
  - 单引号 ''
  - 双引号 ""
  - 反引号 ``（es6之后）
- **前后的引号类型必须一致：**
  - 如果在字符串里面本身包括单引号，可以使用双引号；
  - 如果在字符串里面本身包括双引号，可以使用单引号；
- **除了普通的可打印字符以外，一些有特殊功能的字符可以通过转义字符的形式放入字符串中：**
  - | 转义字符 | 表示符号 |
    | --- | --- |
    | \' | 单引号 |
    | \" | 双引号 |
    | \\ | 反斜杠 |
    | \n | 换行符 |
    | \r | 回车符 |
    | \t | 制表符 |
    | \b | 退格符 |
- **<字符串>本身有的方法和属性**
  - **length**属性：表示字符串的长度
  - 字符串拼接：使用 + 符号

```
// 1.String基本使用
var name = "coderwhy"
var address = "广州市"
var intro = "认真是一种可怕的力量!"

// 2.别的引号的使用
// 单引号
var message1 = 'Hello World'
// 双引号
var message2 = "Hello World"
// 反引号(ES6新增语法)
// ${变量/表达式}
var message3 = `Hello World, ${name}, ${2 + 3}`

// 3.转义字符: 字符串本身中包含引号
var message4 = 'my name is "coderwhy"'
console.log(message4)

var message5 = 'my name \\ \'\' is "coderwhy"'
```

```
console.log(message5)

// 4.<字符串>本身有的方法和属性
var message = "Hello World"
console.log(message.length)

// 字符串操作
var nickname = "coderwhy"
var info = "my name is "
var infoStr = `my name is ${nickname}` // 推荐(babel)
var infoStr2 = info + nickname
console.log(infoStr, infoStr2)
```

## Boolean类型

> 布尔类型的值一般用来做逻辑判断使用

- 布尔类型仅包含两个值：**true** 和 **false**
- Boolean（布尔）类型用于表示**真假**
- 布尔（英语：**Boolean**）是计算机科学中的逻辑数据类型，以发明布尔代数的数学家**乔治·布尔**为名

```
// 1.Boolean基本使用
var isLogin = false
var isAdmin = true
```

## Undefined类型

> 如果我们声明一个变量，但是没有对其进行初始化时，它默认就是undefined；

```
var message
var info = undefined // 不推荐

console.log(message, info)
```

**注意事项：**

- 最好在变量**定义的时候进行初始化**，而不只是声明一个变量；
- **不要显式的将一个变量赋值为undefined**
  - 如果变量刚开始什么都没有，我们可以初始化为0、空字符串、null等值；

## Object类型

> Object 类型是一个特殊的类型，我们通常把它称为**引用类型或者复杂类型**
>
> 1. 其他的数据类型我们通常称之为 “原始类型”，因为它们的值质保函一个单独的内容（字符串、数字或者其他）；
> 2. Object往往可以表示一组数据，是其他数据的一个集合；

> 3. 在JavaScript中我们可以使用 花括号{} 的方式来表示一个对象；

```
// 1.Object类型的基本使用
// var name = "why"
// var age = 18
// var height = 1.88
var person = {
  name: "why",
  age: 18,
  height: 1.88
}
console.log(person)

// 2.对象类型中某一个属性
console.log(person.name)
```

## Null类型

> null类型通常用来表示一个对象为空，所以通常我们在给一个对象进行初始化时，会赋值为null；

- Null 类型同样只有一个值，即特殊值 null。
- null和undefined的关系

> 1. undefined通常只有在一个变量声明但是未初始化时，它的默认值是undefined才会用到；
> 2. 并且我们不推荐直接给一个变量赋值为undefined，所以很少主动来使用；
> 3. null值非常常用，当一个变量准备保存一个对象，但是这个对象不确定时，我们可以先赋值为null；

```
// 3.Null类型
// 3.1. 其他类型的初始化
var age = 0
var num = 0
var message = "" // 空字符串
var isAdmin = false

// Null存在的意义就是对 对象进行初始化的, 并且在转成Boolean类型时, 会转成false
var book = null
console.log(typeof book) // object
```

## 总结

> JavaScript 中有八种基本的数据类型（前七种为**基本数据类型，也称为原始类型**，而 object 为**复杂数据类型，也称为引用类型**）。

- number 用于任何类型的数字：整数或浮点数。
- string 用于字符串：一个字符串可以包含 0 个或多个字符，所以没有单独的单字符类型。

- boolean 用于 true 和 false。
- undefined 用于未定义的值 —— 只有一个 undefined 值的独立类型。
- object 用于更复杂的数据结构。
- null 用于未知的值 —— 只有一个 null 值的独立类型。

## 2.6 数据类型转换

**在开发中，我们可能会在不同的数据类型之间进行某些操作**

- 比如把一个String类型的数字和另外一个Number类型的数字进行运算；
- 比如把一个String类型的文本和另外一个Number类型的数字进行相加；
- 比如把一个String类型或者Number类型的内容，当做一个Boolean类型来进行判断；

**也就是在开发中，我们会经常需要对数据类型进行转换**

- 大多数情况下，**运算符**和**函数**会自动将赋予它们的值转换为正确的类型，这是一种**隐式转换**；
- 我们也可以，通过**显式**的方式来对数据类型进行转换；

### String类型的转换

> 其他类型经常需要转换成字符串类型，比如和字符串拼接在一起或者使用字符串中的方法。

- **隐式转换**
  - 一个字符串和另一个字符串进行+操作；
    - 如果+运算符左右两边有一个是字符串，那么另一边会自动转换成字符串类型进行拼接；
  - 某些函数的执行也会自动将参数转为字符串类型；
    - 比如console.log函数；
- **显式转换**
  - 调用String()函数；
  - 调用toString()方法（面向对象再深入）

```
var num1 = 123
var age = 18
var isAdmin = true

// 1.转换方式一: 隐式转换(用的非常多)
var num1Str = num1 + ""
var ageStr = age + ""
var isAdminStr = isAdmin + ""
console.log(typeof num1Str, typeof ageStr, typeof isAdminStr)

// 2.转换方式一: 显示转换
var num1Str2 = String(num1)
console.log(typeof num1Str2)
```

## Number类型的转换

> 其他类型也可能会转成数字类型。

- 隐式转换
  - 在**算数运算**中，通常**会将其他类型转换成数字类型**来进行运算；
    > 比如 "6" / "2"；
    >
    > 但是如果是+运算，并且其中一边有字符串，那么还是按照字符串来连接的；
- 显式转换
  - **使用Number()函数**来进行显式的转换；
- 其他类型转换数字的规则

| 值 | 转换后的值 |
| --- | --- |
| undefined | NaN |
| null | 0 |
| true 和 false | 1 and 0 |
| string | 去掉首尾空格后的纯数字字符串中含有的数字。如果剩余字符串为空，则转换结果为 0。否则，将会从剩余字符串中“读取”数字。当类型转换出现 error 时返回 NaN。 |

```
// 方式一: 隐式转换(用的很少)
var num1 = "8"
var num2 = "4"
var result1 = num1 + num2 // 84
console.log(typeof result1) // string

var result2 = num1 * num2
console.log(result2) // 32
```

```
// 方式二: 显示转换(Number())
var result3 = Number(num1) // 8
console.log(typeof result3) // number

// 其他类型转成数字类型的规则:
console.log(Number(undefined)) // NaN
console.log(Number(true)) // 1
console.log(Number(false)) // 0
console.log(Number(null)) // 0
console.log(Number("abc123")) // NaN
console.log(Number("")) // 0
console.log(Number("")) // 0
```

### Boolean类型的转换

> 布尔（boolean）类型转换是最简单的，它发生在逻辑运算中，但是也可以通过调用Boolean(value)显式地进行转换。

- 转换规则

| 值 | 转换后 |
| --- | --- |
| 0，null，undefined，NaN，"" | false |
| 其他值 | true |

- **注意：包含0的字符串 "0" 是 true**

```
// 方式一: 隐式转换
// 分支语句
var isAdmin = true
var num1 = 123 // true

// 方式二: 显示转换
console.log(Boolean(num1), Boolean(undefined))

// 转换有如下的规则:
// 直观上为空的值, 转成Boolean类型都是false
// 直观上为空的值: 0/""/undefined/null/NaN -> false

// 注意事项
console.log(Boolean("")) // false
console.log(Boolean("0")) // true
```

## 3. 运算符

> **运算符**就是用来做运算的符号。计算机最基本的操作就是执行运算，执行运算时就需要使用运算符来操作，编程语言按使用场景将运算符分成了以下类型：算术运算符/赋值运算符/关系(比较)运算符/逻辑运算符。

## 3.1 运算元

> 运算元就是运算符应用的对象，如果一个运算符有**n个运算元**，那么这个运算符称为**n元运算符。**

- 比如说乘法运算 5 * 2，有**两个运算元**；
- 左运算元 5 和右运算元 2；
- 有时候人们也称其为“参数”；

## 3.2 算数运算符

> 算术运算符用在数学表达式中, 它的使用方式和数学中也是一致的。
>
> 算术运算符是对数据进行计算的符号。

| 运算符 | 运算规则 | 范例 | 结果 |
| --- | --- | --- | --- |
| + | 加法 | 2 + 3 | 5 |
| + | 连接字符串 | '中' + '国' | '中国' |
| - | 减法 | 2 - 3 | -1 |
| * | 乘法 | 2 * 3 | 6 |
| / | 除法 | 6 / 2 | 3 |
| % | 取余 | 5 % 2 | 1 |
| ** | 幂（ES7） | 2 ** 3 | 8 |

- **取余运算符是 %，尽管它看起来很像百分数，但实际并无关联**

```
//  a % b 的结果是 a 整除 b 的余数
console.log(10 % 3) // 1
```

- **求幂运算 a ** b 将 a 提升至 a 的 b 次幂。（ES7中的语法，也叫做ES2016）**

```
// 在数学中我们将其表示为 a 的 b 次方。
console.log(2**3) // 8
console.log(2**4) // 16
```

## 3.3 赋值运算符

> = 被称之为赋值（ assignments ）运算符。

- **= 是一个运算符，而不是一个有着“魔法”作用的语言结构。**

```
var name; // undefined

// 语句 x = value 将值 value 写入 x 然后返回 x。
name = 'dd' // 'dd'
```

- **链式赋值（Chaining assignments）**

```
let a, b, c;
a = b = c = 2 + 2;
console.log(a, b, c); // 4 4 4
```

  - 链式赋值**从右到左进行计算**；
  - 首先，对最右边的表达式 2 + 2 求值，然后将其赋给左边的变量：c、b 和 a。
  - 最后，所有的变量共享一个值。

### 3.4 原地修改（Modify-in-place）

> 我们经常需要对一个变量做运算，并将新的结果存储在同一个变量中。

```
// 原写法
var n = 10;
n = n + 5;
n = n * 2;

// 原地修改
var n = 10;
n += 5;
n *= 2;

// 所有算术和位运算符都有简短的“修改并赋值”运算符：/= 和 -= 等。
```

| 运算符 | 运算规则 | 范例 | 结果 |
| --- | --- | --- | --- |
| = | 赋值 | a = 5 | 5 |
| += | 加后赋值 | a = 5, a += 2 | 7 |
| -= | 减后赋值 | a = 5, a -= 2 | 3 |
| *= | 乘后赋值 | a = 5; a *= 2 | 10 |
| /= | 除后赋值 | a = 5; a /= 2 | 2.5 |
| %= | 取模 (余数)后赋值 | a = 5; a %= 2 | 1 |
| **= | 幂后赋值 | a = 5; a **= 2 | 25 |

### 3.5 自增、自减运算符

> 对一个数进行**加一、减一**是最常见的数学运算符之一。应用于变量上的操作符
>
> ++ 和 --

- 将一个变量进行**自加1**或**自减1**的操作

```
var currentIndex = 5

// 方法一:
// currentIndex = currentIndex + 1

// 方法二:
// currentIndex += 1

// 方法三: 自增
currentIndex++
console.log(currentIndex)

// 自减
// currentIndex -= 1
currentIndex--
console.log(currentIndex)
```

#### ++ 和 -- 的位置

> ++和--运算符可以在变量前面，也可以在变量后面。

相同点：都是将变量进行自加一或者自减1。

不同点：当自增自减表达式在**其他表达式中参与运算时**，**前置形式**返回**变量自增或自减后的值**，**后置形式**返回**原始值**。

```
var currentIndex = 5
// 自己自增或者自减是没有区别
// ++currentIndex
// console.log(currentIndex) // 6
// --currentIndex
// console.log(currentIndex) // 5

// 自增和自减表达式本身又在其他的表达式中, 那就有区别
// var result1 = 100 + currentIndex++ // 100 + 5
// console.log(currentIndex)    // 6
// console.log("result1:" + result1)    // 105

var result2 = 100 + ++currentIndex // 100 + 6
console.log(currentIndex) // 6
console.log("result2:" + result2) // 106
```

### 3.6 比较运算符

> 用来比较大小的运算符

- 比较运算符的结果都是Boolean类型的

| 运算符 | 运算规则 | 范例 | 结果 |
| --- | --- | --- | --- |
| == | 相等 | 1 == '1' | true |
| === | 严格相等 | 1 === '1' | false |
| != | 不相等 | 3 != '3' | false |
| !== | 严格不相等 | 3 !== '3' | true |
| > | 大于 | 4 > 3 | true |
| < | 小于 | 4 < 3 | false |
| >= | 大于等于 | 4 >= 3 | true |
| <= | 小于等于 | 4 <= 3 | false |

注意：严格相等和严格不相等运算时，当两边的值的类型不一致时，直接返回false；普通相等和不相等运算时，当两边类型不一致时，首先会将两侧的值先转化为数字，再做比较。

```
var num1 = 20
var num2 = 30

// 1.比较运算符
var isResult = num1 > num2
console.log(isResult)

// 2.==判断
console.log(num1 == num2) // false
console.log(num1 != num2) // true

// 需求: 获取到比较大的那个值
// var result = 0
// if (num1 > num2) {
//   result = num1
// } else {
//   result = num2
// }



var foo1 = 0
var foo2 = ""

// ==运算符, 在类型不相同的情况下, 会将运算元先转成Number的值, 再进行比较(隐式转换)
// null比较特殊: null在进行比较的时候, 应该是会被当成一个对象和原生类型进行比较的
console.log(Number(foo1))
console.log(Number(foo2))
```

```
console.log(foo1 == foo2)

// ===运算符，在类型不相同的情况，直接返回false
console.log(foo1 === foo2)
```

### 3.7 逻辑运算符

> 逻辑运算符主要有三个，都是用来做逻辑运算，它可以将**多个表达式或者值放到一起**来获取到一个**最终的结果**；
>
> &&（与）、||（或）、！（非）

| 运算符 | 运算规则 | 范例 | 结果 |
| --- | --- | --- | --- |
| && | 与：同时为真 | false && true | false |
| \|\| | 或：一个为真 | false \|\| true | true |
| ! | 非：取反 | !false | true |

```
var chineseScore = 88
var mathScore = 99

// 1.逻辑与: &&, 并且
// 条件1 && 条件2 && 条件3.....
// 所有的条件都为true的时候，最终结果才为true
// 案例: 小明语文考试90分以上，并且数学考试90分以上，才能去游乐场
if (chineseScore > 90 && mathScore > 90) {
    console.log("去游乐场玩~")
}

// 2.逻辑或: ||, 或者
// 条件1 || 条件2 || 条件3....
// 只要有一个条件为true，最终结果就为true
// 案例: 如果有一门成绩大于90，那么可以吃打1小时游戏
if (chineseScore > 90 || mathScore > 90) {
    console.log("打1个小时游戏~")
}

// 3.逻辑非: !, 取反
var isLogin = true
if (!isLogin) {
    console.log("跳转到登录页面")
    console.log("进行登录~")
}

console.log("正常的访问页面")
```

- **逻辑与（&&）的本质**

1. 拿到第一个运算元, 将运算元转成Boolean类型
2. 对运算元的Boolean类型进行判断
    - 如果false, 返回运算元(原始值)
    - 如果true, 查找下一个继续来运算
    - 以此类推
3. 如果查找了所有的都为true, 那么返回最后一个运算元(原始值)

```
// 本质推导一: 逻辑与, 称之为短路与
var chineseScore = 80
var mathScore = 99
if (chineseScore > 90 && mathScore > 90) {}

// 本质推导二: 对一些对象中的方法进行有值判断
var obj = {
    name: "why",
    friend: {
        name: "kobe",
        eating: function() {
            console.log("eat something")
        }
    }
}

// 调用eating函数
// obj.friend.eating()
obj && obj.friend && obj.friend.eating && obj.friend.eating()
```

- **逻辑或（||）的本质**
    1. 先将运算元转成Boolean类型
    2. 对转成的boolean类型进行判断
        - 如果为true, 直接将结果(原始值)返回
        - 如果为false, 进行第二个运算元的判断
        - 以此类推
    3. 如果找到最后, 也没有找到为真值的运算元, 那么返回最后一个运算元

```
// 本质推导一: 之前的多条件是如何进行判断的
var chineseScore = 95
var mathScore = 99
// chineseScore > 90为true, 那么后续的条件都不会进行判断
if (chineseScore > 90 || mathScore > 90) {}

// 本质推导二: 获取第一个有值的结果
var info = "abc"
var obj = {name: "why"}
var message = info || obj || "我是默认值"
console.log(message.length)
```

- 逻辑非（！）的补充
    - 逻辑非运算符接受一个参数，并按如下运算：

1. 将操作数转化为布尔类型：true/false；
2. 返回相反的值；

- 两个非运算!! 有时候用来将某个值转化为布尔类型：
  - 第一个非运算将该值转化为布尔类型并取反，第二个非运算再次取反。
  - 最后我们就得到了一个任意值到布尔值的转化。

### 3.8 条件运算符（也叫三元运算符）

> **条件（三元）运算符**是 JavaScript 唯一使用三个操作数的运算符：一个条件后跟一个问号（`?`），如果条件为真值，则执行冒号（`:`）前的表达式；若条件为假值，则执行最后的表达式。该运算符经常当作 `if...else` 语句的简捷形式来使用
>
> 语法：condition ? exprIfTrue : exprIfFalse
>
> - `condition`
>   - 计算结果用作条件的表达式。
> - `exprIfTrue`
>   - 如果 `condition` 的计算结果为真值（等于或可以转换为 `true` 的值），则执行该表达式。
> - `exprIfFalse`
>   - 如果 `condition` 为假值（等于或可以转换为 `false` 的值）时执行的表达式。
>
> 描述：除了 `false`，可能的假值表达式还有：`null`、`NaN`、`0`、空字符串（`""`）和 `undefined`。如果 `condition` 是其中任何一个，那么条件表达式的结果就是 `exprIfFalse` 表达式执行的结果。

```
// 案例一: 比较两个数字
var num1 = 12*6 + 7*8 + 7**4
var num2 = 67*5 + 24**2

// 三元运算符
var result = num1 > num2 ? num1: num2
console.log(result)

// 案例二: 给变量赋值一个默认值(了解)
var info = {
    name: "why"
}
var obj = info ? info: {}
console.log(obj)

// 案例三: 让用户输入一个年龄, 判断是否成年人
var age = prompt("请输入您的年龄:")
age = Number(age)
// if (age >= 18) {
```

```
//   alert("成年人")
// } else {
//   alert("未成年人")
// }
var message = age >= 18 ? "成年人": "未成年人"
alert(message)
```

### 3.9 运算符的优先级

> 当一个表达式中有多个运算符时，不同运算符的优先级不一样；**运算符的优先级**决定了表达式中运算执行的先后顺序。优先级高的运算符会作为优先级低的运算符的操作数。MDN

```
var num = 5
var result = 2 + 3 * ++num // 2 + 3 * 6
console.log(result) // 20
```

ps：想要优先运算：添加 **()** 分组运算符，优先级最高

## 4. 分支语句

> 在程序开发中，程序有三种不同的执行方式：顺序、分支、循环

- 顺序：**从上向下**，顺序执行代码
- 分支：根据**条件判断**，决定执行代码的**分支**
- 循环：让 **特定代码 重复** 执行

```
// 1.顺序执行
var num1 = 10
var num2 = 20

var result = num1 + num2
var result2 = num1 * num2

// 2.分支语句
var isLogin = true
if (isLogin) {
    console.log("访问购物车")
    console.log("访问个人中心")
} else {
    console.log("跳转到登录页面")
}

// 3.循环语句
var i = 0;
while (i < 10) {
```

```
    console.log("执行循环语句")
    i++
}
```

## 4.1 代码块

> **代码块**是多行执行代码的集合，通过一个**花括号{}**放到了一起。
>
> 在开发中，一行代码很难完成某一个特定的功能，我们就会将这些代码放到一个**代码块**中
>
> **在JavaScript中，我们可以通过流程控制语句来决定如何执行一个代码块：**
>
> - 通常会通过一些关键字来告知js引擎代码要如何被执行;
> - 比如**分支语句**、**循环语句**对应的关键字等;

```
// 一个代码块
{
    var num1 = 10;
    var num2 = 20;
    var result = num1 + num2;
}

// 一个对象
var info = {
    name: "why",
    age: 18
}
```

## 4.2 分支结构

> - 分支结构的代码就是让我们**根据条件**来决定**代码的执行**
> - 分支结构的语句被称为**判断结构**或者**选择结构**
> - 几乎**所有的编程语言都有分支结构**（C、C++、OC、JavaScript等等）
> - **JavaScript中常见的分支结构有**：**if**分支结构、**switch**分支结构

- if分支结构
  - 单分支结构

    > if(...) 语句计算括号里的条件表达式，如果计算结果是 true，就会执行对应的代码块。

    ```
    // 如果条件成立，那么执行代码块
    // if(条件判断) {
    //     // 执行代码块
    // }


    // 案例一：如果小明考试超过90分，就可以去游乐场
    // 1.定义一个变量来保存小明的分数
    var score = 99
    ```

```
// 2.如果分数超过90分，去游乐场
if (score > 90) {
    console.log("去游乐场玩~")
}


// 案例二：苹果单价5元/斤，如果购买的数量超过5斤，那么立减8元
// 1.定义一些变量保存数据
var price = 5
var weight = 7
var totalPrice = price * weight

// 2.根据购买的重量，决定是否 -8
if (weight > 5) {
    totalPrice -= 8
}

console.log("总价格:", totalPrice)


// 案例三：播放列表中 currentIndex
// ["鼓楼", "理想", "阿刁"]
var currentIndex = 2

// 对currentIndex++完操作之后
currentIndex++
if (currentIndex === 3) {
    currentIndex = 0
}



// 补充一：如果if语句对应的代码块，只有一行代码，那么{}可以省略
// 案例二：苹果单价5元/斤，如果购买的数量超过5斤，那么立减8元
// 定义一些变量保存数据
var price = 5
var weight = 7
var totalPrice = price * weight

// 2.根据购买的重量，决定是否 -8
if (weight > 5) totalPrice -= 8

console.log("总价格:", totalPrice)

// 补充二：if (…) 语句会计算圆括号内的表达式，并将计算结果转换为布尔型（Boolean）
// 转换规则和Boolean函数的规则一致；
// 数字 0、空字符串 “”、null、undefined 和 NaN 都会被转换成 false。
// 其他值被转换为 true，所以它们被称为“真值（truthy）”；
var num = 123 // true
if (num) {
    console.log("num判断的代码执行")
}
```

ps: **如果if语句对应的代码块，只有一行代码，那么{}可以省略**

- 多分支结构

> if..else..
>
> if 语句有时会包含一个可选的 “else” 块
>
> 如果判断条件不成立，就会执行它内部的代码

```
var score = 80
// 条件成立, 专属的代码块
// 条件不成立, 专属的代码块
// if (score > 90) {
//   console.log("去游乐场玩~")
// } else {
//   console.log("哈哈哈哈")
// }

// 案例一: 小明超过90分去游乐场, 否则去上补习班
if (score > 90) {
    console.log("去游乐场玩~")
} else {
    console.log("去上补习班~")
}

// 案例二: 有两个数字, 进行比较, 获取较大的数字
var num1 = 12*6 + 7*8 + 7**4
var num2 = 67*5 + 24**2
console.log(num1, num2)

var result = 0
if (num1 > num2) {
    result = num1
} else {
    result = num2
}
console.log(result)
```

> if..else if..else..
>
> 有时我们需要判断多个条件；我们可以通过使用 else if 子句实现；

```
// 案例: score评级
// 1.获取用户输入的分数
var score = prompt("请输入您的分数:")
score = Number(score)

// 2.判断等级
// 使用if else的方式来实现
// if (score > 90) {
//   alert("您的评级是优秀!")
// } else {
//   if (score > 80) {
//     alert("您的评级是良好!")
//   } else {
```

```
//   }
// }

// edge case
// if (score > 100 || score < 0) {
//   alert("您输入的分数超过了正常范围")
// }

if (score > 90) {
    alert("您的评级是优秀!")
} else if (score > 80) {
    alert("您的评级是良好!")
} else if (score >= 60) {
    alert("您的评级是合格!")
} else {
    alert("不及格!!!")
}
```

- switch分支结构

> 它是通过判断表达式的结果（或者变量）是否**等于**（这里使用 === 进行比较的）case语句的常量，来执行相应的分支体的；
>
> 与if语句不同的是，switch语句只能做值的相等判断（使用全等运算符 ===），而if语句可以做值的范围判断；
>
> switch 语句有至少一个 case 代码块和一个可选的 default 代码块。

```
// 语法
// switch (表达式/变量) {
//   case 常量1:
//     // 语句
// }


// 案例:
// 上一首的按钮: 0
// 播放/暂停的按钮: 1
// 下一首的按钮: 2
// var btnIndex = 100
// if (btnIndex === 0) {
//   console.log("点击了上一首")
// } else if (btnIndex === 1) {
//   console.log("点击了播放/暂停")
// } else if (btnIndex === 2) {
//   console.log("点击了下一首")
// } else {
//   console.log("当前按钮的索引有问题~")
// }

var btnIndex = 0
switch (btnIndex) {
    case 0:
        console.log("点击了上一首")
        break
    case 1:
        console.log("点击了播放/暂停")
```

```
        // 默认情况下是有case穿透
        break
    case 2:
        console.log("点击了下一首停")
        break
    default:
        console.log("当前按钮的索引有问题~")
        break
}
```

- **case穿透问题：**（不会对后续的case **表达式**进行求值）

```
switch (undefined) {
  case console.log(1):
  case console.log(2):
}
// 仅输出 1
```

  - 一条case语句结束后，会自动执行下一个case的语句；
  - 这种现象被称之为case穿透；
- **break关键字**
  - 通过在每个case的代码块后添加break关键字来**解决case穿透**这个问题；
- **这里的相等是严格相等（===）。**