**目录**

[10.1 编程简介 2](#_Toc17995)

[10.2 Hello World! 4](#_Toc7595)

[10.3 语句与注释 6](#_Toc25293)

[10.4 互动方法 7](#_Toc8723)

[10.5 变量 9](#_Toc9966)

[10.6 数据类型 11](#_Toc17481)

[10.7 算术运算符 13](#_Toc19620)

[10.8 表达式 14](#_Toc9657)

[第11章 判断 16](#_Toc18875)

[11.1 逻辑运算符 16](#_Toc32488)

[11.2 if语句 18](#_Toc12022)

[11.3 switch语句 21](#_Toc22137)

[第12章 循环 23](#_Toc1436)

[12.1 自增/自减运算符 23](#_Toc24859)

[12.2 循环语句 24](#_Toc17945)

[12.3 break or continue? 30](#_Toc23067)

[第13章 数组 32](#_Toc6350)

[13.1 数组 32](#_Toc24957)

[13.2 二维数组 34](#_Toc23116)

[13.3 数组操作 37](#_Toc5049)

[第14章 函数 41](#_Toc26259)

[14.1 函数 41](#_Toc28823)

[14.2 本地变量与全局变量 44](#_Toc2820)

[14.3 递归 46](#_Toc4547)

[第15章 事件 52](#_Toc15655)

[15.1 事件的概念 52](#_Toc30468)

[15.2 鼠标单击事件 53](#_Toc25728)

[15.3 鼠标经过/移开事件 54](#_Toc7493)

[15.4 光标聚焦/失焦事件 56](#_Toc14326)

[15.5 内容选中/改变事件 58](#_Toc32466)

[15.6 加载/卸载事件 60](#_Toc29603)

[第16章 对象 61](#_Toc20077)

[16.1 对象的概念 61](#_Toc4620)

[16.2 Date对象 62](#_Toc3116)

[16.3 String对象 64](#_Toc19459)

[16.4 Math对象 68](#_Toc19016)

[第17章 浏览器对象模型BOM 70](#_Toc31949)

[17.1 window对象 71](#_Toc26934)

[17.2 计时器 72](#_Toc13374)

[17.3 Screen对象 76](#_Toc3532)

[第18章 文档对象模型DOM 78](#_Toc20198)

[18.1 DOM的概念 79](#_Toc26937)

[18.2 获取结点对象 79](#_Toc27789)

[18.3 结点属性 82](#_Toc27163)

[18.4 结点操作 84](#_Toc8067)

**10.1 编程简介**

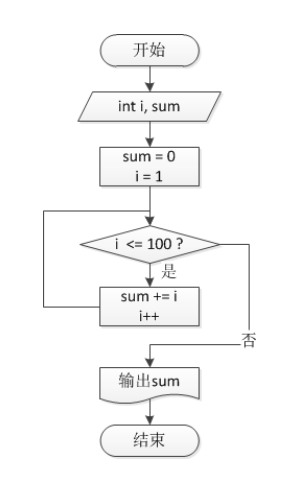
**程序(Program)**

程序是为了让计算机执行某些操作或者解决问题而编写的一系列**有序**指令的集合。由于计算机只能够识别二进制数字**0和1**，因此需要使用特殊的编程语言来描述如何解决问题过程和方法。

**算法(Algorithm)**

算法是可完成特定任务的一系列步骤，算法的计算过程定义明确，通过一些值作为**输入**并产生一些值作为**输出**。

**流程图(Flow Chart)**

流程图是算法的一种**图形化**表示方式，使用一组预定义的符号来说明如何执行特定任务：

1. 开始和结束用**圆角矩形**表示

2. 数据处理用**矩形**表示

3. 输入/输出用**平行四边形**表示

4. **菱形**用于表示分支判断条件

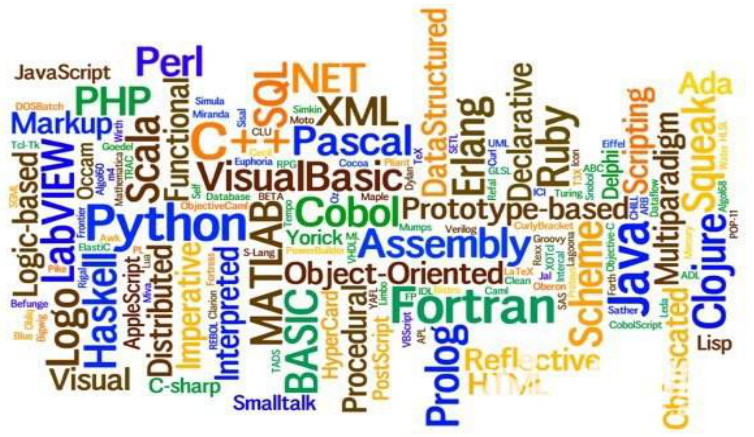
5. 各步骤间用**流程线**连接

**编程语言(Programming Language)**

编程语言主要分为**面向机器、面向过程和面向对象**三类。

只会使用HTML和CSS创建一个漂亮的页面还不够，它只是静态页面而已。还需要使用JavaScript增加行为，为网页添加**动态**效果。

JavaScript可以增强页面动态效果，如下拉菜单、图片轮播、信息滚动等，也可以实现页面与用户之间的实时、动态交互，如用户注册、登录验证等。



JavaScript是一种**直译式脚本语言**，是一种**动态类型**、**弱类型**的语言，是一种运行在浏览器中的**解释性**的编程语言。JavaScript是当今最流行的脚本语言，生活中看到的网页和基于HTML5的APP里面的**交互逻辑**都是由JavaScript驱动的。

**10.2 Hello World!**

**Hello World!**

使用<script>标签在HTML网页中插入JavaScript代码。语法如下：

|  |
| --- |
| <script type="text/JavaScript">      // JavaScript代码  </script> |

type属性表示在**<script>**标签之间的是文本类型，告诉了浏览器里面的文本属于JavaScript语言。

JS作为一种脚本语言可以放在HTML页面中任何位置，但是一般放在网页的head或body部分。最常用的方式是在页面head部分放置<script>元素，浏览器解析HTML时是按先后顺序的，所以前面的script就先被执行。

**引入外部JS文件**

除了在HTML文件中使用<script>标签添加JS代码，还能把HTML和JS分开，单独创建一个JS文件，其文件后缀为.js，然后直接将JS代码写在JS文件中。注意在JS文件中，不需要<script>标签，直接编写JS代码即可。

JS文件不能直接运行，需要嵌入到HTML文件中执行，因此需要在HTML中引入外部JS文件，语法如下：

|  |
| --- |
| <script src="JS文件.js"></script> |

|  |
| --- |
| **范例：Hello World!** |
| **hello\_world.html** |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <title>Hello World!</title>      <script src="hello\_world.js"></script>  </head>  <body>    </body>  </html> |
| **hello\_world.js** |
| document.write("Hello World!"); |

**10.3 语句与注释**

**语句**

JS语句是发给浏览器的命令，这些命令的作用是告诉浏览器要做的事情。一行语句通常在结尾加上一个“**;**”表示语句的**结束**，虽然分号也可以不写，但是要养成编程的好习惯，记得在语句末尾加上分号。

**注释(Comment)**

注释的作用是提高代码的**可读性**，帮助自己和别人阅历和理解所编写的代码，注释的内容不会在网页中显示。

注释可分为单行注释与多行注释两种：

1. **单行注释**：将一行内“**//**”之后的内容视为注释

2. **多行注释**：以“**/\***”开始，“**\*/**”结束，中间的内容视为注释

|  |
| --- |
| **范例：注释** |
| document.write("单行注释");     // 我是单行注释  /\*      我是      多行      注释  \*/  document.write("多行注释"); |

**10.4 互动方法**

**document.write()**

**document.write()**可用于直接向HTML输出流写内容，简单地说就是直接在网页中输出内容。

|  |
| --- |
| **范例：document.write()** |
| document.write("互动方法：document.write()"); |

**console.log()**

**console.log()**方法用于在**控制台**输出信息，该方法对于开发过程进行**测试**很有帮助。

|  |
| --- |
| **范例：console.log()** |
| console.log("互动方法：console.log()"); |

**alert()**

在访问网站的时候，有时会突然弹出一个小窗口，上面写着一段提示信息文字，如果不点击“确定”，就不能对网页做任何操作，这个小窗口就是使用**alert()**实现的。

|  |
| --- |
| **范例：alert()** |
| alert("互动方法：alert()"); |

**confirm()**

**confirm()**消息对话框通常用于允许用户做选择的动作，弹出对话框包含一个**确定按钮**和一个**取消按钮**。语法如下：

|  |
| --- |
| confirm(str); |

其中**str**表示在消息对话框中要**显示的文本**，confirm()的**返回值**为**Boolean**值，当用户点击“**确定**”按钮时返回**true**，当用户点击“**取消**”按钮时返回**false**。

|  |
| --- |
| **范例：confirm()** |
| confirm("互动方法：confirm()"); |

**prompt()**

**prompt()**弹出消息对话框，通常用于询问一些需要与用户**交互**的信息，弹出的消息对话框包含一个**确定按钮**、**取消按钮**和一个**文本输入框**。语法如下：

|  |
| --- |
| prompt(str1, str2); |

其中**str1**表示要**显示在消息对话框中的文本**，**不可修改**。**str2**表示文本框中的**内容**，**可以修改**。点击“**确定**”按钮，文本框中的**内容**将作为函数的**返回值**，点击“**取消**”按钮，将返回**null**。

|  |
| --- |
| **范例：prompt()** |
| prompt("互动方法：prompt()", "文本内容"); |

**10.5 变量**

**变量(Variable)**

变量是计算机中一块特定的**内存空间**，由一个或多个连续的字节组成，不同数据存入具有不同内存地址的空间，相互独立，通过变量名可以简单快速地找到在内存中存储的数据。

定义变量使用关键字**var**，语法如下：

|  |
| --- |
| *var* 变量名; |

**变量名/标识符**

变量名需要符合以下的要求：

1. 必须以**字母**、**下划线**或**美元符号**开头，后面可以跟字母、下划线、美元符号和数字。

2. 变量名区分**大小写**。

3. 不可以使用**保留字/关键字**

4. 准确、顾名思义，**不要**使用汉语拼音

**关键字/保留字(Key Words)**

关键字是编程语言内置的一些名称，具有特殊的用处和意义。

|  |  |  |  |
| --- | --- | --- | --- |
| **关键字** | | | |
| break | else | new | var |
| case | finally | return | void |
| catch | for | switch | while |
| default | if | throw | delete |
| in | try | do | instanceof |
| typeof | abstract | enum | int |
| short | boolean | export | interface |
| static | byte | extends | long |
| super | char | final | native |
| synchronized | class | float | package |
| throws | const | goto | private |
| transient | debugger | implements | protected |
| volatile | double | import | public |

**初始化(initialization)**

变量可以在定义时初始化：

|  |
| --- |
| *var* num = 1; |

也可以在定义后初始化：

|  |
| --- |
| *var* num;  num = 1; |

变量虽然也可以不声明直接使用，但是这样不规范。

**赋值运算符(Assignment)**

在编程中，“**=**”不是数学中的“等于”符号，而是表示“**赋值**”，即将“=”右边的值赋给左边的变量。

**10.6 数据类型**

**数据类型**

JS中变量主要有以下几种类型：

**1. 数字number：整型、浮点型**

**2. 字符串string**

**3. 布尔boolean**

**4. 空对象null**

**5. 未定义undefined**

**6. 复杂数据类型：数组Array、对象Object**

使用**typeof()**方法可以检测一个变量的**数据类型**。

|  |  |
| --- | --- |
| **范例：数据类型** | |
| *var* a = 12;  *var* b = 3.1415;  *var* c = "hello";  *var* d = false;  console.log(typeof(a))  console.log(typeof(b))  console.log(typeof(c))  console.log(typeof(d)) | |
| **运行结果** | number  number  string  boolean |

**类型转换**

类型转换是把变量从一种类型**转换**为另一种数据类型。类型转换可以是**隐式**的，也可以是**显式**的，通过使用强制类型转换的方法来指定。

|  |  |
| --- | --- |
| **范例：类型转换** | |
| console.log(*Number*("123"));  console.log(parseInt("456"));  console.log(parseFloat("78.9"));  console.log(*String*(123456789));  console.log(*Boolean*(0));  console.log(*Boolean*(1)); | |
| **运行结果** | 123  456  78.9  123456789  false  true |

**10.7 算术运算符**

**四则运算**

|  |  |  |
| --- | --- | --- |
| **数学符号** | **JS符号** | **意义** |
| + | + | 加 |
| - | - | 减 |
| × | \* | 乘 |
| ÷ | / | 除 |
|  | % | 取模 |
| () | () | 括号 |

取模%(mod)表示求两个数相除之后的**余数**，如：22 % 3的结果为1、4 % 7的结果为4。

**复合赋值运算符**

|  |  |  |
| --- | --- | --- |
| **赋值运算符** | **描述** | **实例** |
| += | 加法赋值运算符 | c += a等价于c = c + a |
| -= | 减法赋值运算符 | c -= a等价于c = c - a |
| \*= | 乘法赋值运算符 | c \*= a等价于c = c \* a |
| /= | 除法赋值运算符 | c /= a等价于c = c / a |
| %= | 取模赋值运算符 | c %= a等价于c = c % a |

**10.8 表达式**

**字符串连接**

“+”运算符不只代表加法，还可以用于**连接**两个字符串。

|  |  |
| --- | --- |
| **范例：字符串连接** | |
| console.log("Hello" + "World"); | |
| **运行结果** | HelloWorld |

**转义字符**

在一个**字符串**描述的过程中，有可能会有一些**特殊字符**的信息。

|  |  |
| --- | --- |
| **符号** | **描述** |
| \ | 续行符，实现字符串多行定义 |
| \\ | 表示一个反斜杠“\” |
| \' | 表示一个单引号“'” |
| \" | 表示一个双引号“"” |
| \n | 换行 |
| \v | 纵向制表符 |
| \t | 横向制表符 |
| \r | 回车 |
| \f | 换页 |
| \b | 退格 |

|  |  |
| --- | --- |
| **范例：转义字符** | |
| console.log("全球最大同性交友网站\n\'https://github.com\'"); | |
| **运行结果** | 全球最大同性交友网站  'https://github.com' |

**表达式**

表达式与数学中的定义相似，表达式是值具有一定的值，用操作符把常数和变量连接起来的代数式。

使用**prompt()**可以获取用户**输入**的信息，返回值为字符串类型。

|  |  |
| --- | --- |
| **范例：计算圆的面积** | |
| *var* PI = 3.14159;  *var* r = parseFloat(prompt("输入半径"));  *var* area = PI \* r \*\* 2;  console.log("面积 = " + area); | |
| **运行结果** | 输入半径：5  面积 = 78.53975 |

**Math.floor()**的作用**向下取整**，**Math.ceil()**的作用是**向上取整**。

|  |  |
| --- | --- |
| **范例：逆序三位数** | |
| *var* num = parseInt(prompt("输入一个正三位数"));  *var* a = Math.floor(num / 100);  *var* b = Math.floor(num / 10) % 10;  *var* c = num % 10;  num = c \* 100 + b \* 10 + a;  console.log("逆序：" + num); | |
| **运行结果** | 输入一个正三位数：520  逆序：25 |

**第11章 判断**

**11.1 逻辑运算符**

**关系运算符**

|  |  |
| --- | --- |
| **数学符号** | **JS关系运算符** |
| ＜ | < |
| ≤ | <= |
| ＞ | > |
| ≥ | >= |
| ≠ | != |
| = | == |

**逻辑运算符**

JS中逻辑运算符有三种，分别是：

1. **逻辑与&&** (logical AND)：当两个条件同时为真，结果为真。

2. **逻辑或||** (logical OR)：两个条件有一个为真时，结果为真。

3. **逻辑非!** (logical NOT)：条件为真时，结果为假；条件为假时，结果为真。

|  |  |
| --- | --- |
| **运算符** | **表达式** |
| && | 条件1 && 条件2 |
| || | 条件1 || 条件2 |
| ! | !条件 |

|  |  |  |
| --- | --- | --- |
| **逻辑与** | | |
| 条件1 | 条件2 | 条件1 && 条件2 |
| T | T | T |
| T | F | F |
| F | T | F |
| F | F | F |

|  |  |  |
| --- | --- | --- |
| **逻辑或** | | |
| 条件1 | 条件2 | 条件1 || 条件2 |
| T | T | T |
| T | F | T |
| F | T | T |
| F | F | F |

|  |  |
| --- | --- |
| **逻辑非** | |
| 条件 | !条件 |
| T | F |
| F | T |

**11.2 if语句**

**if语句**

当if语句的条件为**真**时，进入花括号执行内部的代码；若条件为**假**，则跳过花括号执行后面的代码。

if语句主要有以下几种形式：

|  |
| --- |
| if(条件) {      //code  } |

|  |  |
| --- | --- |
| **范例：使用if语句** | |
| *var* age = 15;  if(age > 0 && age < 18) {      console.log("未成年")  } | |
| **运行结果** | 未成年 |

|  |
| --- |
| timgif(条件) {      //code  } else {      //code  } |

|  |  |
| --- | --- |
| **范例：使用if-else语句** | |
| *var* age = 30;  if(age > 0 && age < 18) {      console.log("未成年人")  } else {      console.log("成年人");  } | |
| **运行结果** | 成年人 |

|  |
| --- |
| if(条件) {      //code  } else if(条件) {      //code  } else {      //code  } |

|  |  |
| --- | --- |
| **范例：多条件判断** | |
| *var* score = 76;  if(score >= 90 && score <= 100) {      console.log("优秀");  } else if(score >= 60 && score < 90) {      console.log("合格");  } else {      console.log("不合格");  } | |
| **运行结果** | 合格 |

if语句也可以嵌套使用：

|  |
| --- |
| if(条件1) {      if(条件2) {          //code      }  } |

|  |  |
| --- | --- |
| **范例：判断整数奇偶** | |
| *var* num = parseInt(prompt("输入一个正整数"));  if(num > 0) {      if(num % 2 == 0) {          console.log(num + "是偶数");      } else {          console.log(num + "是奇数");      }  } | |
| **运行结果** | 输入一个正整数：66  66是偶数 |

**11.3 switch语句**

**switch语句**

switch-case结构可以对**整数值**的表达式进行判断。语法如下：

|  |
| --- |
| switch(表达式) {      case label:          //code          break;      // ...      default:          //code          break;  } |

根据表达式的值，跳转到对应的case处进行执行。需要注意的是，当对应的case中的代码被执行完后，并**不会跳出**switch，而是会继续执行后面的代码，所以需要使用**break跳出**switch结构。

当所有case都不满足表达式的值时，会执行**default**语句中的代码，相当于if-else结构中的else。

|  |  |
| --- | --- |
| **范例：根据月份输出对应的英语简写** | |
| *var* month = parseInt(prompt("输入月份"));  switch(month) {      case 1:          console.log("Jan.");          break;      case 2:          console.log("Feb.");          break;      case 3:          console.log("Mar.");          break;      case 4:          console.log("Apr.");          break;      case 5:          console.log("May");          break;      case 6:          console.log("Jun.");          break;      case 7:          console.log("Jul.");          break;      case 8:          console.log("Aug.");          break;      case 9:          console.log("Sep.");          break;      case 10:          console.log("Oct.");          break;      case 11:          console.log("Nov.");          break;      case 12:          console.log("Dec.");          break;      default:          alert("输入有误");          break;  } | |
| **运行结果** | 输入月份：5  May |

**第12章 循环**

**12.1 自增/自减运算符**

**自增/自减运算符**

单目运算符中自增++、自减--运算符可以将变量的值进行加1、减1，但是++和--可以出现在变量之前或之后，即有四种情况：

1. **前缀自增**

2. **前缀自减**

3. **后缀自增**

4. **后缀自减**

|  |  |  |
| --- | --- | --- |
| **表达式** | **运算** | **含义** |
| count++ | 给count加1 | 执行完所在语句后自增1 |
| ++count | 给count加1 | 执行所在语句前自增1 |
| count-- | 给count减1 | 执行完所在语句后自减1 |
| --count | 给count减1 | 执行所在语句前自减1 |

**12.2 循环语句**

**while循环**

在while循环中，当条件满足时**重复**循环体内的语句。如果条件**永远**为真，循环会永无止境的进行下去（**死循环**），因此循环体内要有**改变条件**的机会。

控制循环次数的方法就是设置**循环变量**：**初值、判断、更新**。

while循环的特点是**先判断、再执行**，所以循环体有可能会进入**一次或多次**，也有可能**一次也不会**进入。

|  |
| --- |
| while(条件) {      //code  } |

|  |  |
| --- | --- |
| **范例：计算5个人的平均身高** | |
| *var* height;  *var* total = 0;  *var* average;  *var* i = 1;  while(i <= 5) {      height = parseFloat(prompt("输入第" + i + "个人的身高"));      total += height;      i++;  }  average = total / 5;  console.log("平均身高：" + average); | |
| **运行结果** | 输入第1个人的身高：160.8  输入第2个人的身高：175.2  输入第3个人的身高：171.2  输入第4个人的身高：181.3  输入第5个人的身高：164  平均身高：170.5 |

**do-while循环**

do-while循环在进入循环的时候**不做检查**，而是在执行完一轮循环体的代码之后，再来检查循环的条件是否满足，如果满足则继续下一轮循环，不满足则结束循环，即**至少执行一次**循环。

do-while循环的主要特点是**先执行、再判断**。

|  |
| --- |
| do {      //code  } while(条件); |

|  |  |
| --- | --- |
| **范例：计算整数位数** | |
| *var* num = 123;  *var* n = 0;  do {      num = Math.floor(num / 10);      n++;  } while(num != 0);  console.log("位数：" + n); | |
| **运行结果** | 输入整数：123  位数：3 |

do-while循环与while循环的区别：

1. 执行顺序不同

2. 初始情况不满足循环条件时：

* while循环一次都不会执行
* do-while循环不管任何情况都至少执行一次

3. do-while循环的while语句后有“**;**”



|  |  |
| --- | --- |
| **timg (1)范例：猜数字** | |
| //产生1-100之间的随机数  *var* answer = Math.floor(Math.random() \* 100) + 1;  *var* num = 0;  *var* cnt = 0;  do {      num = parseInt(prompt("猜一个1-100之间的数字"));      cnt++;      if(num > answer) {          alert("猜大了！");      } else if(num < answer) {          alert("猜小了！");      }  } while(num != answer);  alert("猜对了！你一共用了" + cnt + "次猜对！"); | |
| **运行结果** | 猜一个1-100之间的数字：50  猜大了！  猜一个1-100之间的数字：25  猜小了！  猜一个1-100之间的数字：37  猜小了！  猜一个1-100之间的数字：43  猜小了！  猜一个1-100之间的数字：46  猜小了！  猜一个1-100之间的数字：48  猜小了！  猜一个1-100之间的数字：49  猜对了！你一共用了7次猜对！ |

**for循环**

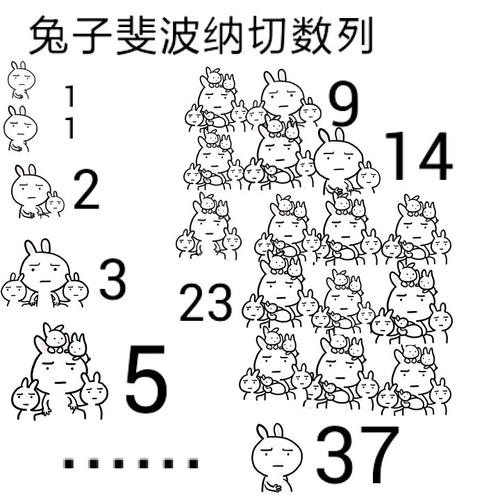
for循环有**三个**表达式，中间用**分号分隔，分号不可省略**：

* 表达式1通常是为循环变量赋**初值**，可省略。
* 表达式2是循环**条件**，是否继续执行循环，可省略。
* 表达式3为**更新**循环变量的值，可省略。

|  |
| --- |
| for(表达式1; 表达式2; 表达式3) {      //code  } |

|  |  |
| --- | --- |
| **u=2556415990,2055694793&fm=26&gp=0范例：计算1-100的累加和** | |
| *var* sum = 0;  for(*var* i = 1; i <= 100; i++) {      sum += i;  }  console.log("累加：" + sum); | |
| **运行结果** | 累加：5050 |

|  |  |
| --- | --- |
| **范例：计算** | |
| *var* n = 10;  *var* sum = 0;  for(*var* i = 1; i <= n; i++) {      sum += 1 / i;  }  console.log(sum); | |
| **运行结果** | 输入n：10  2.9289682539682538 |

****

|  |  |
| --- | --- |
| **范例：兔子数列（斐波那契数列）** | |
| *var* n = 10;  *var* num1, num2, val;  *var* str = "";  if(n == 1) {      console.log("1");  } else if(n == 2) {      console.log("1, 1");  } else {      num1 = 1;      num2 = 1;      str = "1, 1";      for(*var* i = 3; i <= n; i++) {          val = num1 + num2;          str += ", " + val;          num1 = num2;          num2 = val;      }      console.log(str);  } | |
| **运行结果** | 输入斐波那契数列长度：10  1, 1, 2, 3, 5, 8, 13, 21, 34, 55 |

循环也可以进行嵌套使用。

|  |  |
| --- | --- |
| **范例：九九乘法表** | |
| *var* str = "";  for(*var* i = 1; i <= 9; i++) {      for(*var* j = 1; j <= 9; j++) {          str += i + "\*" + j + "=" + i\*j + "\t";      }      str += "\n";  }  console.log(str); | |
| **运行结果** | 1\*1=1 1\*2=2 1\*3=3 1\*4=4 1\*5=5 1\*6=6 1\*7=7 1\*8=8 1\*9=9  2\*1=2 2\*2=4 2\*3=6 2\*4=8 2\*5=10 2\*6=12 2\*7=14 2\*8=16 2\*9=18  3\*1=3 3\*2=6 3\*3=9 3\*4=12 3\*5=15 3\*6=18 3\*7=21 3\*8=24 3\*9=27  4\*1=4 4\*2=8 4\*3=12 4\*4=16 4\*5=20 4\*6=24 4\*7=28 4\*8=32 4\*9=36  5\*1=5 5\*2=10 5\*3=15 5\*4=20 5\*5=25 5\*6=30 5\*7=35 5\*8=40 5\*9=45  6\*1=6 6\*2=12 6\*3=18 6\*4=24 6\*5=30 6\*6=36 6\*7=42 6\*8=48 6\*9=54  7\*1=7 7\*2=14 7\*3=21 7\*4=28 7\*5=35 7\*6=42 7\*7=49 7\*8=56 7\*9=63  8\*1=8 8\*2=16 8\*3=24 8\*4=32 8\*5=40 8\*6=48 8\*7=56 8\*8=64 8\*9=72  9\*1=9 9\*2=18 9\*3=27 9\*4=36 9\*5=45 9\*6=54 9\*7=63 9\*8=72 9\*9=81 |

|  |  |
| --- | --- |
| **范例：输出图案**  \*  \*\*  \*\*\*  \*\*\*\*  \*\*\*\*\* | |
| *var* str = "";  for(*var* i = 1; i <= 5; i++) {      for(*var* j = 1; j <= i; j++) {          str += "\*";      }      str += "\n";  }  console.log(str); | |
| **运行结果** | \*  \*\*  \*\*\*  \*\*\*\*  \*\*\*\*\* |

**12.3 break or continue?**

**循环控制**

循环控制语句的作用是控制当前的循环结构是否继续向下执行，如果不进行控制，那么会根据既定的结构重复执行。如果有一些特殊的情况导致循环的执行**中断**，就称为循环的控制语句。循环控制语句的关键字有**break**和**continue**。

break的作用是**跳出当前循环**，执行当前循环之后的语句。break**只能**跳出一层循环，如果是嵌套循环，那么需要按照嵌套的层次，逐步使用break来跳出。break语句只能在**循环体内**和**switch语句**内使用。

continue的作用是**跳过本轮循环**，**开始下一轮循环**的条件判断。continue终止当前轮的循环过程，但它并**不跳出**循环。

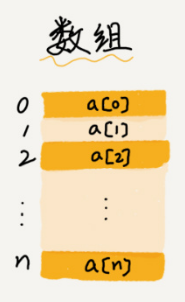
|  |  |
| --- | --- |
| **范例：break语句** | |
| *var* str = "";  for(*var* i = 1; i <= 10; i++) {      if(i == 5) {          break;      }      str += i + " ";  }  console.log(str); | |
| **运行结果** | 1 2 3 4 |

|  |  |
| --- | --- |
| **范例：continue语句** | |
| *var* str = "";  for(*var* i = 1; i <= 10; i++) {      if(i == 5) {          continue;      }      str += i + " ";  }  console.log(str); | |
| **运行结果** | 1 2 3 4 6 7 8 9 10 |

**第13章 数组**

**13.1 数组**

**数组(Array)**

一个变量只能存储一个内容，如果需要存储更多数据，就需要使用数组解决问题。一个数组变量可以存放多个数据，数组是一个值的集合，它们共享同一个名字，数组中的每个变量都能被其**下标**所访问。

关于数组的一些术语：

1. **元素**：数组中的每个变量

2. **大小**：数组的容量

3. **下标/索引**(index)：元素的位置，下标从**0**开始，必须为非负整数

使用数组之前首先要创建数组，数组的创建有2种方式：

1. **直接创建**

|  |
| --- |
| *var* arr1 = [];          //创建一个空数组  *var* arr2 = [1, 2, 3];   //创建有内容的数组 |

2. 利用**构造函数**创建

|  |
| --- |
| *var* arr1 = new *Array*();                 //创建空数组  *var* arr2 = new *Array*(10);               //创建一个长度为10的数组  *var* arr3 = new *Array*(5, 4, 3, 2, 1);    //创建数组并初始化 |

虽然创建数组时指定了长度，但实际上数组都是**变长**的，也就是说即使指定了长度，仍然可以将元素存储在规定长度以外。

**一维数组初始化**

很多时候在使用数组之前需要将数组的内容全部清空，这可以利用循环来实现。

|  |
| --- |
| **范例：初始化数组** |
| *var* arr = new *Array*(100);  for(*var* i = 0; i < arr.length; i++) {      arr[i] = 0;  } |

|  |  |
| --- | --- |
| **范例：找出数组中的最大值和最小值** | |
| *var* num = [7, 6, 2, 9, 3, 1, 4, 0, 5, 8];  *var* max = num[0];  *var* min = num[0];  for(*var* i = 1; i < num.length; i++) {      if(num[i] > max) {          max = num[i];      } else if(num[i] < min) {          min = num[i];      }  }  console.log("max = " + max);  console.log("min = " + min); | |
| **运行结果** | max = 9  min = 0 |

**13.2 二维数组**

**二维数组(2D Array)**

二维数组可以看成是由多个一维数组组成的。二维数组包括**行**和**列**两个维度。

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| a[0][0] | a[0][1] | a[0][2] | a[0][3] | a[0][4] |
| a[1][0] | a[1][1] | a[1][2] | a[1][3] | a[1][4] |
| a[2][0] | a[2][1] | a[2][2] | a[2][3] | a[2][4] |

二维数组的可以直接创建：

|  |  |
| --- | --- |
| **范例：直接创建二维数组** | |
| *var* arr = [[1, 2], [3, 4]];  console.log(arr); | |
| **运行结果** | (2) [Array(2), Array(2)]  0: (2) [1, 2]  1: (2) [3, 4]  length: 2 |

也可以使用**构造函数**循环创建：

|  |  |
| --- | --- |
| **范例：构造函数创建二维数组** | |
| *var* arr = new *Array*(3);  for(*var* i = 0; i < arr.length; i++) {      arr[i] = new *Array*(4);  } | |
| **运行结果** | (3) [Array(4), Array(4), Array(4)]  0: (4) [empty× 4]  1: (2) [empty× 4]  2: (4) [empty× 4]  length: 3 |

利用两层循环来初始化二维数组。

|  |
| --- |
| **范例：初始化二维数组** |
| *var* arr = new *Array*(3);  for(*var* i = 0; i < arr.length; i++) {      arr[i] = new *Array*(4);      for(*var* j = 0; j < arr[i].length; j++) {          arr[i][j] = 0;      }  } |

**矩阵运算**

矩阵的加法/减法是指两个[矩阵](https://baike.baidu.com/item/%E7%9F%A9%E9%98%B5/18069" \t "https://baike.baidu.com/item/%E7%9F%A9%E9%98%B5%E5%8A%A0%E6%B3%95/_blank)把其相对应元素加/减一起的运算。

矩阵加法：两个m×n矩阵A和B的和，标记为A+B，结果为一个m×n的矩阵，其内的各元素为其相对应元素相加后的值。

1

矩阵的减法：两个m×n矩阵A和B的差，标记为A-B，结果为一个m×n的矩阵，其内的各元素为其相对应元素相减后的值。

2

|  |  |
| --- | --- |
| **范例：矩阵运算** | |
| *var* A = [      [1, 3],      [1, 0],      [1, 2]  ];  *var* B = [      [0, 0],      [7, 5],      [2, 1]  ];  *var* C = new *Array*(3);  for (*var* i = 0; i < C.length; i++) {      C[i] = new *Array*(2);  }  console.log("矩阵加法");  for(*var* i = 0; i < C.length; i++) {      for (*var* j = 0; j < C[i].length; j++) {          C[i][j] = A[i][j] + B[i][j];      }  }  console.log(C);  console.log("矩阵减法");  for(*var* i = 0; i < C.length; i++) {      for(*var* j = 0; j < C[i].length; j++) {          C[i][j] = A[i][j] - B[i][j];      }  }  console.log(C); | |
| **运行结果** | 矩阵加法  [ [ 1, 3 ], [ 8, 5 ], [ 3, 3 ] ]  矩阵减法  [ [ 1, 3 ], [ -6, -5 ], [ -1, 1 ] ] |

**13.3 数组操作**

**数组操作**

数组的常见操作主要包括：

1. **计算数组长度**：引用数组的**length**属性获取数组长度，需要注意的是，JS数组的length属性是**可变**的。

|  |  |
| --- | --- |
| **范例：计算数组长度** | |
| *var* arr = [0, 1, 2, 3, 4];  console.log(arr.length);  arr.length = 10;  console.log(arr.length); | |
| **运行结果** | 5  10 |

2. **增加元素**：

* 使用下一个**未使用**的索引，任何时刻可以不断向数组增加新元素。

|  |  |
| --- | --- |
| **范例：增加元素** | |
| *var* arr = [0, 1, 2, 3, 4];  arr[5] = 5;  console.log(arr); | |
| **运行结果** | [0, 1, 2, 3, 4, 5] |

* 使用**unshift()**方法可以向数组第一个元素前面添加一个元素，返回值为数组长度。

|  |  |
| --- | --- |
| **范例：头部增加元素** | |
| *var* arr = [0, 1, 2, 3, 4];  arr.unshift(5);  console.log(arr); | |
| **运行结果** | [5, 0, 1, 2, 3, 4] |

* **push()**方法可以向数组最后一个元素后面添加一个元素，返回值为数组长度。

|  |  |
| --- | --- |
| **范例：尾部增加元素** | |
| *var* arr = [0, 1, 2, 3, 4];  arr.push(5);  console.log(arr); | |
| **运行结果** | [0, 1, 2, 3, 4, 5] |

3. **删除元素**：

* **shift()**方法可以删除数组的第一个元素，返回值为被删除元素。

|  |  |
| --- | --- |
| **范例：删除头部元素** | |
| *var* arr = [0, 1, 2, 3, 4];  arr.shift();  console.log(arr); | |
| **运行结果** | [1, 2, 3, 4] |

* **pop()**方法可以删除数组的最后一个元素，返回值为被删除元素。

|  |  |
| --- | --- |
| **范例：删除尾部元素** | |
| *var* arr = [0, 1, 2, 3, 4];  arr.pop();  console.log(arr); | |
| **运行结果** | [0, 1, 2, 3] |

4. **合并数组**：使用**concat()**方法可以合并两个或多个数组，该方法不会改变原有数组，而是返回一个新的合并完的数组。

|  |  |
| --- | --- |
| **范例：合并数组** | |
| *var* arr1 = [1, 2, 3, 4, 5];  *var* arr2 = [6, 7, 8, 9, 10];  console.log(arr1.concat(arr2));  console.log(arr1);  console.log(arr2); | |
| **运行结果** | [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  [1, 2, 3, 4, 5]  [6, 7, 8, 9, 10] |

5. **数组转字符串**：使用**join(separator)**方法可以将数组转换为字符串，其中separator参数可选，用于指定要使用的分隔符，如果该参数省略，则使用逗号作为分隔符。

|  |  |
| --- | --- |
| **范例：数组转字符串** | |
| *var* arr = [0, 1, 2, 3, 4];  console.log(arr.join()); | |
| **运行结果** | 0,1,2,3,4 |

6. **字符串转数组**：使用**split(separator, n)**方法可以将字符串转换为数组，其中参数separator必选，用于指定将字符串按某个字符切割成若干个子字符串，并以数组的形式返回。参数n可选，用于指定返回的数组的最大长度，如果设置了该参数，返回的子串数量不会多于n；如果没有设置该参数，整个字符串都会被分隔。

|  |  |
| --- | --- |
| **范例：字符串转数组** | |
| *var* str = "hello HTML hello CSS hello JavaScript";  *var* arr = str.split(' ');  console.log(arr); | |
| **运行结果** | ["hello", "HTML", "hello", "CSS", "hello", "JavaScript"] |

7. **翻转数组**：使用**reverse()**方法可以颠倒数组中元素的顺序。

|  |  |
| --- | --- |
| **范例：翻转数组** | |
| *var* arr = [1, 2, 3, 4, 5];  console.log(arr.reverse()); | |
| **运行结果** | [5, 4, 3, 2, 1] |

8. **数组排序**：使用**sort(sortfun)**方法可以将数组进行排序，其中参数sortfun可选，用于指定排序规则，而且必须是函数，该参数省略则按照字符编码顺序排序。

|  |  |
| --- | --- |
| **范例：数组排序** | |
| *var* arr = [98, 1, 21, 8, 12, 2, 10, 25];  arr.sort(*function*(*a*, *b*) {      return *a* > *b* ? 1 : -1;  });  console.log(arr); | |
| **运行结果** | [1, 2, 8, 10, 12, 21, 25, 98] |

9. **数组切分**：使用**slice(start, end)**方法可以返回数组中被选定的元素，不包含下标为end的元素。其中参数start必选，用于指定开始位置，如果是负数则从数组尾部开始算起。参数end可选，用于指定结束位置，没有该参数省略，则切分到数组结束为止，如果是负数则从数组尾部开始算起。

|  |  |
| --- | --- |
| **范例：数组切分** | |
| *var* arr = [0, 1, 2, 3, 4, 5, 6];  console.log(arr.slice(2, 5)); | |
| **运行结果** | [2, 3, 4] |

10. **查找元素**：使用**indexOf(item, start)**方法可以查找指定元素，如果查找成功则返回该元素的下标，如果查找失败者返回-1。其中参数item必选，用于指定需要查找的元素，参数start可选，用于指定在数组中开始检索的位置，如省略则从第一个元素开始检索。

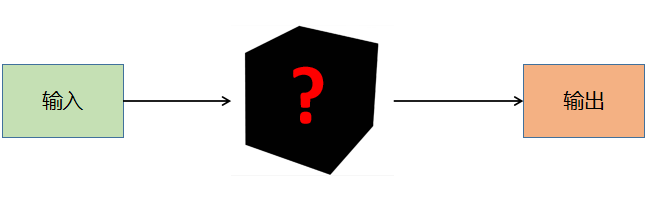
|  |  |
| --- | --- |
| **范例：查找元素** | |
| *var* arr = [0, 1, 2, 3, 4, 5, 6];  console.log(arr.indexOf(4)); | |
| **运行结果** | 4 |

**第14章 函数**

**14.1 函数**

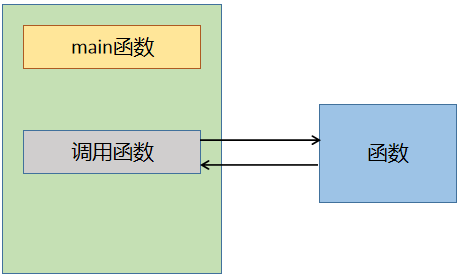
**函数(Function)**

函数执行一个**特定**的任务，JS提供了大量**内置函数**，例如alert()用来显示警告对话框、parseInt()用来将字符串转换为整型等。



**函数调用**

当**调用**函数时，程序**控制权**会转移给**被调用**的函数，当函数执行**结束**后，函数会把程序序控制权交还给其**调用者**。



**函数定义**

函数的定义需要使用关键字**function**，语法格式如下：

|  |
| --- |
| *function* funcName(*parameterList*) {      //code  } |

**为什么要使用函数？**

为什么不把所有的代码全部按顺序写，还需要自定义函数呢？使用函数有以下好处：

1. 避免“**代码复制**”：“代码复制”是程序质量不良的表现

2. 便于代码的**维护**

3. 避免重复制造“**轮子**”，提高开发效率

**函数的设计方法**

在设计函数的时候需要考虑以下的几点要素：

1. 确定函数的**功能**

2. 确定函数的**参数**（是否需要参数、参数的个数、参数的类型）

3. 确定函数的**返回值**（是否需要返回值、返回值的类型）

|  |  |
| --- | --- |
| **范例：函数实现返回最大值** | |
| *function* max(*num1*, *num2*) {      // if(num1 > num2) {      //     return num1;      // } else {      //     return num2;      // }      return *num1* > *num2* ? *num1* : *num2*;  }  console.log(max(4, 12));  console.log(max(54, 33));  console.log(max(0, -12));  console.log(max(-999, -774)); | |
| **运行结果** | 12  54  0  -774 |

|  |  |
| --- | --- |
| **范例：函数实现累加和** | |
| *function* sum(*start*, *end*) {  *var* total = 0;      for(*var* i = *start*; i <= *end*; i++) {          total += i;      }      return total;  }  console.log("1-100的累加和 = " + sum(1, 100));  console.log("1024-2048的累加和 = " + sum(1024, 2048)); | |
| **运行结果** | 1-100的累加和 = 5050  1024-2048的累加和 = 1574400 |

|  |  |
| --- | --- |
| **范例：函数实现输出i行j列由自定义字符组成的图案** | |
| *function* print\_chars(*row*, *col*, *c*) {  *var* str = "";      for(*var* i = 0; i < *row*; i++) {          for(*var* j = 0; j < *col*; j++) {              str += *c*;          }          str += "\n";      }      console.log(str);  }  print\_chars(5, 10, '?'); | |
| **运行结果** | ??????????  ??????????  ??????????  ??????????  ?????????? |

**14.2 本地变量与全局变量**

**本地变量(Local Variable)**

JS的本地变量是在**函数**里面被声明的，这些变量的**作用域**在**本地**，也就是说这些变量只能在函数内部可用。本地变量在函数**调用时被创造**，在函数**结束时被销毁**。

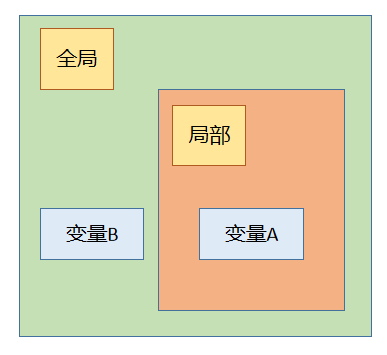
在函数中，函数的每次调用就会产生一个**独立**的空间，在这个空间中的变量，是函数的这次运行所独有的，函数的参数也是本地变量。

|  |  |
| --- | --- |
| **范例：本地变量** | |
| *function* test(*a*) {  *a* = 2;      console.log("a = " + *a*);  }  *var* a = 1;  console.log("a = " + a);  test(a);  console.log("a = " + a); | |
| **运行结果** | a = 1  a = 2  a = 1 |

**全局变量(Global Variable)**

JS的全局变量就是在**函数外**被声明的变量，作用域为**全局**，**所有**的在页面上的**脚本**和**函数**都可以获取这些变量。全局变量在其被声明时创建，在**页面被关闭时被销毁**。

全局变量的优先级**低于**[局部变量](https://baike.baidu.com/item/%E5%B1%80%E9%83%A8%E5%8F%98%E9%87%8F/9844788" \t "https://baike.baidu.com/item/%E5%85%A8%E5%B1%80%E5%8F%98%E9%87%8F/_blank)，当全局变量与局部变量重名的时候，起作用的是局部变量，全局变量会被暂时**屏蔽**掉。



|  |  |
| --- | --- |
| **范例：全局变量** | |
| *var* a = 1;          // 全局变量  *function* test() {  *var* a = 2;      // 本地变量      console.log("a = " + a);  }  test(); | |
| **运行结果** | a = 2 |

**14.3 递归**

**递归(Recursive)**

要理解递归，先得理解递归（见14.3章节）。

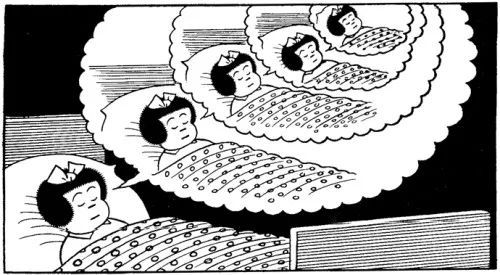
在函数的内部，直接或者间接的**调用自己**的过程就叫作递归。对于一些问题，使用递归可以简洁易懂的解决问题，但是递归的缺点是**性能低**，占用大量系统**栈空间**。

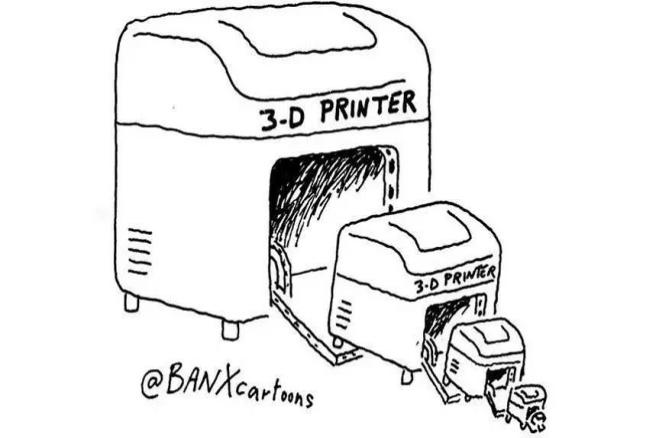
递归算法很多时候可以处理一些特别复杂、难以直接解决的问题，比如：

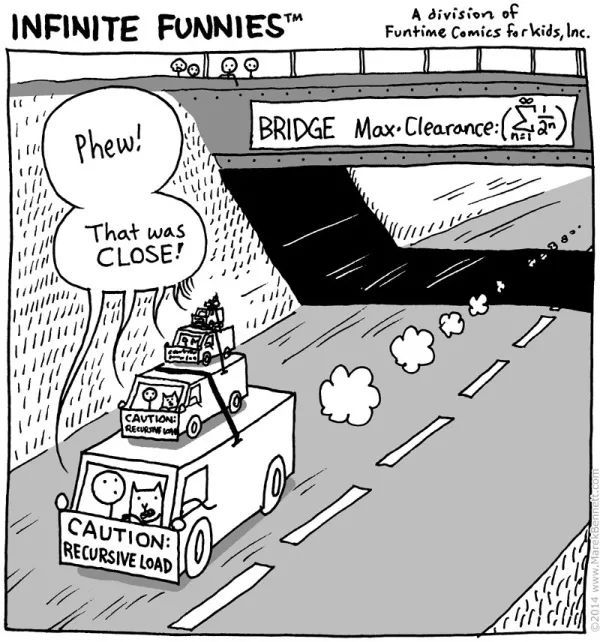
1. 迷宫问题

2. 汉诺塔问题

在定义递归函数时，一定要确定一个“**结束条件**”，否则会造成无限递归的情况，最终会导致**栈溢出**。



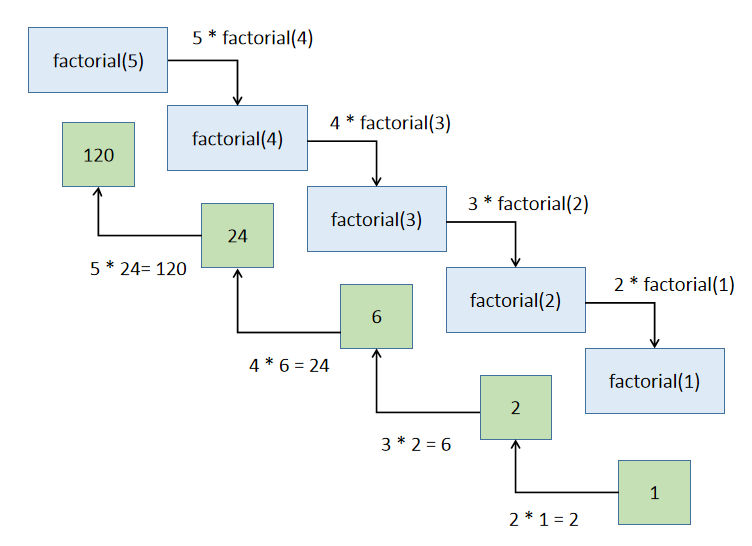




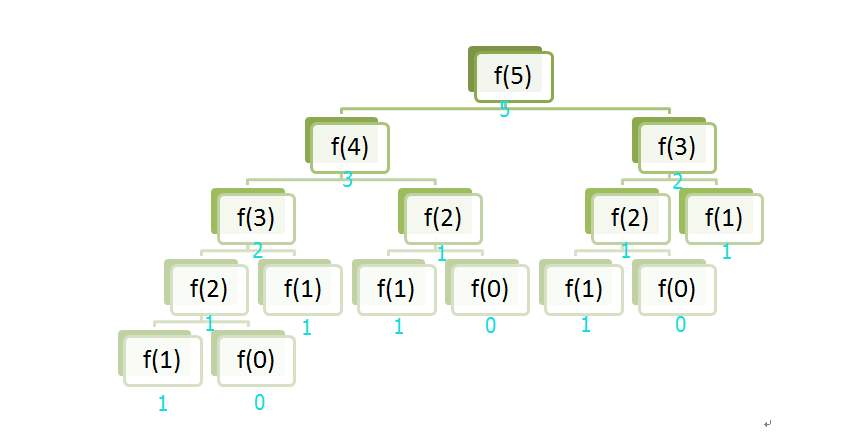
|  |  |
| --- | --- |
| **范例：无限递归** | |
| *function* tell\_story() {      console.log("从前有座山");      console.log("山里有座庙");      console.log("庙里有个老和尚和小和尚");      console.log("老和尚对小和尚在讲故事");      console.log("他讲的故事是：");      tell\_story();  }  tell\_story(); | |
| **运行结果** | 从前有座山  山里有座庙  庙里有个老和尚和小和尚  老和尚对小和尚在讲故事  他讲的故事是：  从前有座山  山里有座庙  庙里有个老和尚和小和尚  老和尚对小和尚在讲故事  他讲的故事是：  ... |

递归函数一般需要定义递归的**出口**，即“**结束条件**”，确保递归能够在适合的地方退出。

|  |  |
| --- | --- |
| **范例：阶乘** | |
| *function* factorial(*n*) {      if(*n* == 0 || *n* == 1) {          return 1;      }      return *n* \* factorial(*n* - 1);  }  console.log("5! = " + factorial(5)); | |
| **运行结果** | 5! = 120 |

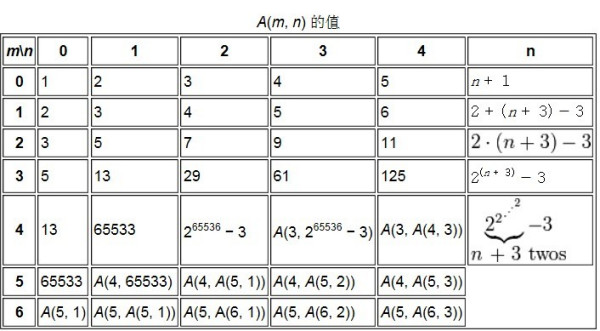


|  |  |
| --- | --- |
| **范例：斐波那契数列（递归）** | |
| *function* fibonacci(*n*) {      if(*n* == 1 || *n* == 2) {          return 1;      }      return fibonacci(*n* - 2) + fibonacci(*n* - 1);  }  n = 7;  console.log("斐波那契数列第" + n + "位：" + fibonacci(7)); | |
| **运行结果** | 斐波那契数列第7位：13 |



|  |  |
| --- | --- |
| **范例：斐波那契数列（迭代）** | |
| *function* fibonacci(*n*) {  *var* f = new *Array*(*n* + 1);      f[1] = f[2] = 1;      for(*var* i = 3; i <= *n*; i++) {          f[i] = f[i - 2] + f[i - 1];      }      return f[*n*];  }  n = 7;  console.log("斐波那契数列第" + n + "位：" + fibonacci(7)); | |
| **运行结果** | 斐波那契数列第7位：13 |

|  |  |
| --- | --- |
| **1165-1范例：阿克曼函数** | |
| *function* A(*m*, *n*) {      if(*m* == 0) {          return *n* + 1;      } else if(*m* > 0 && *n* == 0) {          return A(*m*-1, 1);      } else if(*m* > 0 && *n* > 0) {          return A(*m*-1, A(*m*, *n*-1));      }  }  console.log(A(3, 4)); | |
| **运行结果** | 125 |





|  |  |
| --- | --- |
| **timg (1)范例：汉诺塔** | |
| *var* move = 0;       // 移动次数  /\*\*   \* *@brief*  汉诺塔算法   \* *@note*   把 n 个盘子从 src 借助 mid 移到 dst   \* *@param*  n: 层数   \* *@param*  src: 起点柱子   \* *@param*  mid: 临时柱子   \* *@param*  dst: 目标柱子   \*/  *function* hanoi(*n*, *src*, *mid*, *dst*) {      if(*n* == 1) {          console.log(*n* + "号盘：" + *src* + " -> " + *dst*);          move++;      } else {          // 把前 n-1 个盘子从 src 借助 dst 移到 mid          hanoi(*n*-1, *src*, *dst*, *mid*);          // 移动第 n 个盘子          console.log(*n* + "号盘：" + *src* + " -> " + *dst*);          move++;          // 把刚才的 n-1 个盘子从 mid 借助 src 移到 dst          hanoi(*n*-1, *mid*, *src*, *dst*);      }  }  hanoi(4, 'A', 'B', 'C');  console.log("步数 ==> " + move); | |
| **运行结果** | 1号盘：A -> B  2号盘：A -> C  1号盘：B -> C  3号盘：A -> B  1号盘：C -> A  2号盘：C -> B  1号盘：A -> B  4号盘：A -> C  1号盘：B -> C  2号盘：B -> A  1号盘：C -> A  3号盘：B -> C  1号盘：A -> B  2号盘：A -> C  1号盘：B -> C  步数 ==> 15 |

**第15章 事件**

**15.1 事件的概念**

**事件**

JS创建**动态页面**时，事件是可以被JS**侦测**到的**行为**。网页中的每个元素都可以产生某些可以出发JS函数或程序的事件。例如当用户单击或者提交表单数据时，就发生一个鼠标单击事件，需要**浏览器做出处理**，返回给用户一个结果。

JS主要的事件包括：

|  |  |
| --- | --- |
| **事件** | **说明** |
| onclick | 鼠标单击事件 |
| onmouseover | 鼠标经过事件 |
| onmouseout | 鼠标移开事件 |
| onchange | 文本框内容改变事件 |
| onselect | 文本框内容被选中事件 |
| onfocus | 光标聚集 |
| onblur | 光标离开 |
| onload | 网页导入 |
| onunload | 关闭网页 |

**15.2 鼠标单击事件**

**鼠标单击事件onclick**

当在网页上**单击鼠标**时，就会发生该事件，同时**onclick**事件调用的程序块就会被执行，onclick通常与**按钮**一起使用。

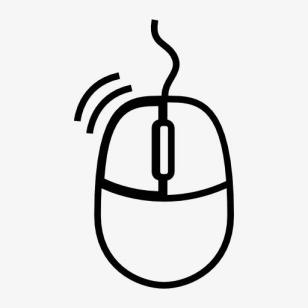


|  |
| --- |
| **范例：鼠标单击事件onclick** |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <title>鼠标单击事件onclick</title>      <script type="text/JavaScript">  *var* cnt = 0;  *function* feedback() {              cnt++;              console.log("我被点击了"+ cnt + "次");          }      </script>  </head>  <body>      <form action="get/post">          <input type="button" value="点击" onclick="feedback();">      </form>  </body>  </html> |

**15.3 鼠标经过/移开事件**

**鼠标经过事件onmouseover**

当鼠标**移到一个对象上**时，该对象就会触发**onmouseover**事件，并执行onmouseover事件调用的程序。



|  |
| --- |
| **范例：鼠标经过事件onmouseover** |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <title>鼠标经过事件onmouseover</title>      <script type="text/JavaScript">  *function* feedback() {              alert("卸载之前再想想吧...");          }      </script>  </head>  <body>      <form action="get/post">          <input type="button" value="卸载"              onmouseover="feedback();">      </form>  </body>  </html> |

**鼠标移开事件onmouseout**

当鼠标**移开当前对象**时，执行**onmouseout**事件调用的程序。

|  |
| --- |
| **范例：鼠标移开事件onmouseout** |
| <!DOCTYPE html>  <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <title>鼠标移开事件onmouseout</title>      <script type="text/JavaScript">  *function* feedback() {              alert("不要离开！只要输入密码，再点击登录就OK啦！");          }      </script>  </head>  <body>      <form action="get/post">          密码：<input type="password"                  onmouseout="feedback();">          <input type="button" value="登录">      </form>  </body>  </html> |

**15.4 光标聚焦/失焦事件**

**光标聚焦事件onfocus**

当网页中的对象**获得聚点**时，执行**onfocus**事件调用的程序。

|  |
| --- |
| **范例：光标聚焦事件onfocus** |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <title>光标聚焦事件onfocus</title>      <script type="text/JavaScript">  *var* flag = true;  *function* feedback() {              if(flag) {                  alert("不要填错啦！");                  flag = false;              }          }      </script>  </head>  <body>      <form action="get/post">          密码：<input type="password"                  onfocus="feedback();">          <input type="button" value="登录">      </form>  </body>  </html> |

**失焦事件onblur**

onblur事件与onfocus事件是**相对事件**，当**光标离开**当前获得聚焦对象的时候，就会触发**onblur**事件，同时执行被调用的程序。

|  |
| --- |
| **范例：失焦事件onblur** |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <title>失焦事件onblur</title>      <script type="text/JavaScript">  *function* feedback() {              alert("确定输对了再点登录哟！");          }      </script>  </head>  <body>      <form action="get/post">          密码：<input type="password"                  onblur="feedback();">          <input type="button" value="登录">      </form>  </body>  </html> |

**15.5 内容选中/改变事件**

**内容选中事件onselect**

当文本框或者文本域中的文本**被选中**时，触发**onselect**事件，同时调用的程序就会被执行。

|  |
| --- |
| **范例：内容选中事件onselect** |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <title>内容选中事件onselect</title>      <script type="text/JavaScript">  *function* feedback() {              console.log("文本内容被选中");          }      </script>  </head>  <body>      <form action="get/post">          <textarea rows="10" cols="30" onselect="feedback();">填写个人信息，不少于20个字。</textarea>      </form>  </body>  </html> |

**内容改变事件onchange**

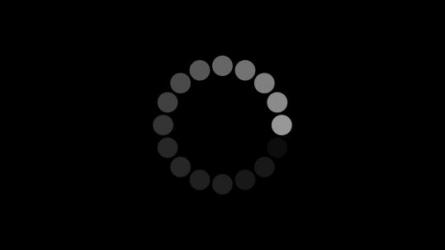
通过**改变文本框的内容**可以触发**onchange**事件，同时执行被调用的程序。

|  |
| --- |
| **范例：内容改变事件onchange** |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <title>内容改变事件onchange</title>      <script type="text/JavaScript">  *function* feedback() {              console.log("文本内容被修改");          }      </script>  </head>  <body>      <form action="get/post">          <textarea rows="10" cols="30" onchange="feedback();">填写个人信息，不少于20个字。</textarea>      </form>  </body>  </html> |

**15.6 加载/卸载事件**

**加载事件onload**

**加载事件会在页面加载完成后立即发生**，同时执行被调用的程序。注意，**加载事件需要写在<body>标签内**。



|  |
| --- |
| **范例：加载事件onload** |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <title>加载事件onload</title>      <script type="text/JavaScript">  *function* feedback() {              alert("页面加载完成");          }      </script>  </head>  <body onload="feedback();">      <p>Hello World!</p>  </body>  </html> |

**卸载事件onunload**

当用户**退出页面**时（页面**关闭**、页面**刷新**等），就会触发**onunload**事件，同时执行被调用的程序。注意，**不同浏览器对onunload事件的支持不同**。

**第16章 对象**

**16.1 对象的概念**

**对象(Object)**

JS中的所有事物都是**对象**，如字符串、数值、数组、函数等，每个对象带有**属性**和**方法**。对象的**属性**反映了该对象的某些**特定性质**，如字符串的长度、图像的长宽等，对象的**方法**指的是能够在对象上执行的**动作**，如表单的提交、时间的获取等。

JS提供了多个**内建对象**，如String、Date、Array等，使用对象前需要先使用**new**关键字进行**定义**。

|  |  |
| --- | --- |
| **范例：访问对象的属性** | |
| *var* arr = new *Array*(1, 2, 3, 4, 5);  console.log(arr.length); | |
| **运行结果** | 5 |

|  |  |
| --- | --- |
| **范例：访问对象的方法** | |
| *var* str = "Hello World";  console.log(str.toUpperCase()); | |
| **运行结果** | HELLO WORLD |

**16.2 Date对象**

**Date对象**

**Date对象**可以存储任意一个**日期**，并且可以精确到毫秒数（1/1000秒）。使用**默认构造函数**创建的日期对象有初始值，为**当前电脑系统时间**。



|  |  |
| --- | --- |
| **范例：定义Date对象** | |
| *var* date1 = new *Date*();  console.log(date1);  *var* date2 = new *Date*(2021, 3, 19);      //此处月份从0开始  console.log(date2); | |
| **运行结果** | 2021-03-19T05:09:47.713Z  2021-04-18T16:00:00.000Z |

Date对象中处理时间和日期的常用方法包括：

|  |  |
| --- | --- |
| **方法名称** | **功能描述** |
| getDate()/setDate() | 返回/设置日期 |
| getFullYear()/setFullYear() | 返回/设置年份，用四位数表示 |
| getYear()/setYear() | 返回/设置年份 |
| getMonth()/setMonth() | 返回/设置月份，月份从0开始 |
| getHours()/setHours() | 返回/设置小时，24小时制 |
| getMinutes()/setMinutes() | 返回/设置分钟数 |
| getSeconds()/setSeconds() | 返回/设置秒钟数 |
| getTime()/setTime() | 返回/设置时间（毫秒为单位） |
| getDay() | 返回0-6的数字表示星期，0表示星期天 |

|  |  |
| --- | --- |
| **范例：获取今日星期** | |
| *var* date = new *Date*();  *var* weekday = [      "星期天",      "星期一",      "星期二",      "星期三",      "星期四",      "星期五",      "星期六"  ];  console.log("今天是" + weekday[date.getDay()]); | |
| **运行结果** | 今天是星期五 |

**16.3 String对象**

**String对象**

定义**String对象**后就可以访问它的属性和方法。

|  |  |
| --- | --- |
| **范例：计算字符串长度** | |
| *var* str = "Hello World!"  console.log(str.length); | |
| **运行结果** | 12 |

**大小写转换**

使用String对象的**toUpperCase()**和**toLowerCase()**方法可以将字符串进行**大小写字母转换**。

|  |  |
| --- | --- |
| **范例：大小写字母转换** | |
| *var* str = "Hello World!"  console.log(str.toUpperCase());  console.log(str.toLowerCase()); | |
| **运行结果** | HELLO WORLD!  hello world! |



**返回指定的字符**

使用**charAt()**方法可**返回指定位置的字符**，返回的字符是长度为1的字符串。语法如下：

|  |
| --- |
| stringObj.charAt(index); |

字符串第一个字符的下标是0，最后一个字符的下标为length-1，如果参数index不在0与length-1之间，该方法将返回一个空字符串。

|  |  |
| --- | --- |
| **范例：返回指定位置的字符** | |
| *var* str = "Hello World!"  console.log(str.charAt(6)); | |
| **运行结果** | W |

**返回指定的字符串首次出现的位置**

**indexOf()**方法可以**返回某个指定的字符串值在字符串中首次出现的位置**。语法如下：

|  |
| --- |
| stringObj.indexOf(substring, startPos); |

该方法将从头到尾地检索字符串，检查是否含有需检索的子串。参数startPos为可选参数，用于规定开始查找的位置，如果没有设置此参数将从头开始查找。如果找到了子串，在返回子串的第一次出现位置。如果要检索的字符串值没有出现，则该方法返回-1。

|  |  |
| --- | --- |
| **范例：查找子串** | |
| *var* str = "Hello World!";  console.log(str.indexOf("Hell"));  console.log(str.indexOf("o", 6));  console.log(str.indexOf("JS")); | |
| **运行结果** | 0  7  -1 |

**字符串分隔**

**split()**方法可以**将字符串分割为字符串数组**，并返回此数组。语法如下：

|  |
| --- |
| stringObj.split(separator, limit); |

其中参数separator必选，用于指定将字符串按某个字符切割成若干个子字符串，并以数组的形式返回。参数limit可选，用于指定分隔的次数。

如果把空字符串作为separator，那么字符串的每个字符之间都会被分隔。

|  |  |
| --- | --- |
| **范例：字符串分隔** | |
| *var* str = "hello HTML hello CSS hello JavaScript";  console.log(str.split(" "));  console.log(str.split(" ", 4)); | |
| **运行结果** | ["hello", "HTML", "hello", "CSS", "hello", "JavaScript"]  ["hello", "HTML", "hello", "CSS"] |

**提取字符串**

**substring()**方法用于**提取字符串中介于两个指定下标之间的字符**。语法如下：

|  |
| --- |
| stringObj.substring(startPos, stopPos); |

该方法返回的内容是从startPos开始到stopPos-1处的所有内容，其长度为stopPos-startPos。如果startPos和stopPos相等，那么返回的就是一个空串（长度为0的字符串）。如果startPos比stopPos大，那么该方法在提取子串之间会先交换这两个参数。

|  |  |
| --- | --- |
| **范例：提取字符串** | |
| *var* str = "HelloWorld";  console.log(str.substring(2, 8));  console.log(str.substring(3, 3));  console.log(str.substring(7, 3)); | |
| **运行结果** | lloWor  loWo |

**提取指定数目的字符**

**substr()**方法用于**从字符串中提取从startPos位置开始的指定数目的字符串**。语法如下：

|  |
| --- |
| stringObj.substr(startPos, length); |

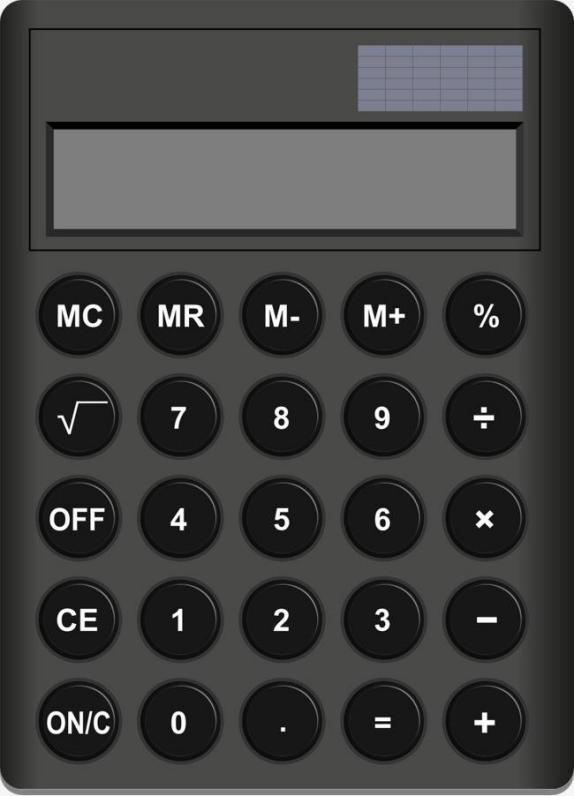
如果参数startPos是负数，则从字符串的尾部开始算起，如果startPos为负数且绝对值大于字符串长度，则startPos会被视为0。

|  |  |
| --- | --- |
| **范例：提取指定数目的字符** | |
| *var* str = "HelloWorld";  console.log(str.substr(2, 3));  console.log(str.substr(-5, 4)); | |
| **运行结果** | llo  Worl |

**16.4 Math对象**

**Math对象**

**Math对象**提供对数据的**数学计算**。需要注意的是，Math对象是一个**固有**的对象，**无需创建**它，直接把Math作为对象使用就可以调用其所有属性和方法，这是它与其它对象的区别。



Math对象的属性：

|  |  |
| --- | --- |
| **属性** | **说明** |
| E | 返回算术常量e，即自然对数的底数（约等于2.718） |
| LN2 | 返回2的自然对数（约等于0.693） |
| LN10 | 返回10的自然对数（约等于2.302） |
| LOG2E | 返回以2为底e的对数（约等于1.442） |
| LOG10E | 返回以10为底e的对数（约等于0.434） |
| PI | 返回圆周率（约等于3.14159） |
| SQRT1\_2 | 返回2的平方根的倒数（约等于0.707） |
| SQRT2 | 返回2的平方根（约等于1.414） |

Math对象的方法：

|  |  |
| --- | --- |
| **方法** | **描述** |
| sin(x) | 返回x的正弦 |
| cos(x) | 返回x的余弦 |
| tan(x) | 返回x的正切 |
| acos(x) | 返回x的反余弦值 |
| asin(x) | 返回x的反正弦值 |
| atan(x) | 返回x的反正切值 |
| **ceil(x)** | **对x进行上取整** |
| **floor(x)** | **对x进行下取整** |
| **abs(x)** | **返回x的绝对数** |
| **exp(x)** | **返回e的x次幂** |
| log(x) | 返回x的自然对数（底为e） |
| **pow(x, y)** | **返回x的y次幂** |
| **max(x, y)** | **返回x和y中的最大值** |
| **min(x, y)** | **返回x和y中的最小值** |
| **round(x)** | **返回x的四舍五入最接近的整数** |
| **sqrt(x)** | **返回x的平方根** |
| **random()** | **返回0-1之间的随机数** |

**第17章 浏览器对象模型BOM**

**17.1 window对象**

**window对象**

**window**对象是**浏览器对象模型BOM(Browser Object Model)**的核心。

window对象方法有：

|  |  |
| --- | --- |
| **方法** | **描述** |
| alert() | 显示带有一段消息和一个确认按钮的警告框 |
| prompt() | 显示可提示用户输入的对话框 |
| confirm() | 显示带有一段消息以及确认按钮和取消按钮的对话框 |
| open() | 打开一个新的浏览器窗口或查找一个已命名的窗口 |
| close() | 关闭浏览器窗口 |
| print() | 打印当前窗口的内容 |
| focus() | 把键盘焦点给与一个窗口 |
| blur() | 把键盘焦点从顶层窗口移开 |
| moveBy() | 可相对窗口的当前坐标把它移动指定的像素 |
| moveTo() | 把窗口的左上角移动到一个指定的坐标 |
| resizeBy() | 按照指定的像素调整窗口的大小 |
| resizeTo() | 把窗口的大小调整到指定的宽度和高度 |
| scrollBy() | 按照指定的像素值来滚动内容 |
| scrollTo() | 把内容滚动到指定的坐标 |
| setInterval() | 每隔指定的时间执行代码 |
| setTimeout() | 在指定的延迟时间之后来执行代码 |
| clearInterval() | 取消setInterval()的设置 |
| clearTimeout() | 取消setTimeout()的设置 |

**17.2 计时器**

**计时器**

在JS中，可以在设定的时间间隔之后执行代码，而不是在函数被调用后立即执行。

计时器的类型分为2种：

1. **一次性计时器：仅在指定的延迟时间之后触发一次**。

2. **间隔性触发计时器：每隔一定的时间间隔就触发一次**。

计时器的方法有：

|  |  |
| --- | --- |
| **方法** | **描述** |
| **setTimeout()** | **在指定的延迟时间之后来执行代码** |
| **clearTimeout()** | **取消setTimeout()的设置** |
| **setInterval()** | **每隔指定的时间执行代码** |
| **clearInterval()** | **取消setInterval()的设置** |

**setTimeout()**

**setTimeout()**计时器，在**载入后延迟指定时间**后，去执行一次表达式，**仅执行一次**。语法如下：

|  |
| --- |
| setTimeout(代码, 延迟时间); |

其中，第一个参数为要调用的函数或要执行的代码串，第二个参数表示在执行代码前需等待的时间，以**毫秒**为单位（**1s = 1000ms**）。

|  |
| --- |
| **范例：setTimeout()计时器** |
| //网页打开2秒后弹出提示框  setTimeout("alert('Welcome')", 2000); |

clearTimeout()和setTimeout()一起使用，用于**停止计时器**。语法如下：

|  |
| --- |
| clearTimeout(id\_of\_setTimeout); |

**参数id\_of\_setTimeout是setTimeout()返回的ID值**，该值标识要取消的延迟执行代码块。

|  |
| --- |
| **范例：计数器** |
| **counter.html** |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <title>计数器</title>      <script src="counter.js"></script>  </head>  <body>      <form action="get/post">          <input type="text" id="num">          <input type="button" value="stop" onclick="stopCount();">      </form>  </body>  </html> |
| **counter.js** |
| *var* cnt = 0;        // 计数  *var* counter;        // 计数器  /\*\*   \* 每隔1000毫秒计数加1   \*/  *function* count() {      document.getElementById('num').value = cnt;      cnt++;      counter = setTimeout(count, 1000);  }  /\*\*   \* 停止计数器   \*/  *function* stopCount() {      clearTimeout(counter);  }  setTimeout(count, 1000);        // 启动计数器 |

**setInterval()**

**setInterval()**在执行时，**从载入页面后每隔指定的时间执行代码**。语法如下：

|  |
| --- |
| setInterval(代码, 交互时间); |

其中，第一个参数表示要代用的函数或要执行的代码串，第二个参数表示周期性执行或调用表达式之间的时间间隔，以**毫秒**为单位（**1s = 1000ms**）。

**clearInterval()**方法可**取消由setInterval()设置的交互时间**。语法如下：

|  |
| --- |
| clearInterval(id\_of\_setInterval); |

**参数id\_of\_setInterval是setInterval()返回的ID值**。

|  |
| --- |
| **范例：实时显示当前时间** |
| **current\_time.html** |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <title>显示当前时间</title>      <script src="current\_time.js"></script>  </head>  <body>      <form action="get/post">          <input type="text" id="time" size="50">      </form>  </body>  </html> |
| **current\_time.js** |
| // function clock() {  //     var date = new Date();  //     document.getElementById("time").value = date;  // }  // setInterval(clock, 1000);  // 箭头函数  setInterval(() *=>* {  *var* date = new *Date*();      document.getElementById("time").value = date;  }, 1000); |

**17.3 Screen对象**

**Screen对象**

Screen对象用于获取用户的**屏幕信息**。window.screen对象在编写时**可以不使用**window前缀。语法如下：

|  |
| --- |
| window.screen.属性; |

|  |  |
| --- | --- |
| **属性** | **描述** |
| availHeight | 窗口可以使用的屏幕高度，单位像素 |
| availWidth | 窗口可以使用的屏幕宽度，单位像素 |
| colorDepth | 用户浏览器表示的颜色位数，通常为32位（每像素的位数） |
| pixelDepth | 用户浏览器表示的颜色位数，通常为32位（每像素的位数），IE不支持此属性 |
| height | 屏幕的高度，单位像素 |
| width | 屏幕的宽度，单位像素 |

**screen.availWidth**和**screen.availHeight**属性返回访问者屏幕的宽度和高度，单位为像素，**减去界面特性**，比如**任务栏**等。不同系统的任务栏**默认高度**不一样，及任务栏的位置可在屏幕上下左右任何位置，所以有可能可用宽度和高度不一样。

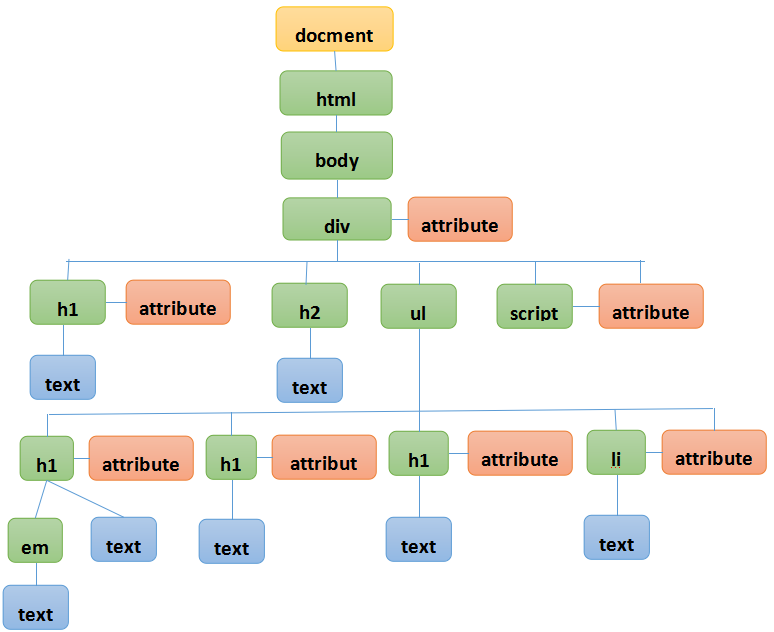
|  |  |
| --- | --- |
| **范例：屏幕信息** | |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <title>屏幕信息</title>      <script type="text/JavaScript">          console.log("屏幕分辨率：" + screen.width + "\*" + screen.height)          console.log("屏幕可用宽高：" + screen.availWidth + "\*" + screen.availHeight);      </script>  </head>  <body>    </body>  </html> | |
| **运行结果** | 屏幕分辨率：1536\*864  屏幕可用宽高：1536\*824 |

**第18章 文档对象模型DOM**

**18.1 DOM的概念**

**DOM (Document Object Model)**

**文档对象模型DOM**定义访问和处理HTML文档的标准方法，DOM将HTML文档呈现为带有元素、属性和文本的**树结构**（结点树）。



**18.2 获取结点对象**

**getElementById()**

**getElementById()**方法可返回拥有**指定ID的第一个对象的引用**。语法如下：

|  |
| --- |
| document.getElementById(elementID); |

如果没有指定ID的元素则返回null，如果存在多个指定ID的元素则返回第一个。

|  |
| --- |
| **范例：定时变换颜色** |
| **random\_color.html** |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <title>随机颜色</title>      <script src="random\_color.js"></script>  </head>  <body>      <div id="square" style="width: 100px; height: 100px"></div>  </body>  </html> |
| **random\_color.js** |
| /\*\*   \* 随机生成RGB颜色代码   \* *@returns* rgb颜色   \*/  *function* randomRGB() {  *var* r = Math.floor(Math.random() \* 256);  *var* g = Math.floor(Math.random() \* 256);  *var* b = Math.floor(Math.random() \* 256);      return "rgb(" + r + ", " + g + ", " + b + ")";  }  /\*\*   \* 获取元素结点，设置背景颜色   \*/  *function* changeColor() {  *var* obj = document.getElementById("square");      obj.style.background = randomRGB();  }  // 每隔300ms改变颜色  setInterval(*function* () {      changeColor();  }, 300); |

**getElementsByClassName()**

**getElementByClassName()**方法返回文档中**所有指定类名**的元素集合，作为NodeList对象。NodeList对象代表一个有顺序的结点列表，可以通过索引来访问列表中的结点。使用NodeList的length属性可以确定指定类名的元素个数，并循环各个元素来获取某个元素。语法如下：

|  |
| --- |
| document.getElementsByClassName(className); |

**getElementByName()**

**getElementByName()**方法返回带有**指定名称**的结点对象的集合。语法如下：

|  |
| --- |
| document.getElementsByName(name); |

与getElementById()方法不同的是，通过元素的name属性查询元素，而不是通过id属性。文档中的name属性可能不唯一，所以getElementByName()方法返回的是元素的数组，而不是一个元素。

**getElementsByTagName()**

**getElementsByTagName()**方法返回带有**指定标签名**的结点对象的集合，返回元素的顺序是它们在文档中的顺序。语法如下：

|  |
| --- |
| document.getElementsByTagName(tagName); |

**18.3 结点属性**

**结点属性**

**getAttribute()**方法可以通过元素结点的属性名称获取**属性的值**。语法如下：

|  |
| --- |
| elementNode.getAttribute(name); |

其中，elementNode可以使用getElementById()、getElementsByTagName()等方法获取到元素结点，参数name为需要查询的元素结点的属性名称。

**setAttribute()**方法可以**增加**一个指定名称和值的新属性，或者把一个**现有**的属性设定为指定的值。语法如下：

|  |
| --- |
| elementNode.setAttribute(name, value); |

其中，name表示要设置的属性名，value表示要设置的属性值。

|  |  |
| --- | --- |
| **范例：设置结点属性值** | |
| **getAttribute.html** | |
| <!DOCTYPE html>  <html lang="en">  <head>      <meta charset="UTF-8">      <title>设置结点属性</title>      <script src="getAttribute.js"></script>  </head>  <body>      <a class="link" href="https://www.baidu.com">百度</a>      <a class="link" href="https://www.bilibili.com">哔哩哔哩</a>  </body>  </html> | |
| **getAttribute.js** | |
| window.onload = *function*() {  *var* links = document.getElementsByClassName("link");      for(*var* i = 0; i < links.length; i++) {          console.log(links[i].getAttribute("href"));          links[i].setAttribute("target", "\_blank");      }  }; | |
| **运行结果** | https://www.baidu.com  https://www.bilibili.com |

**18.4 结点操作**

**创建结点createElement()**

**createElement()**方法可以**创建**结点元素，此方法可返回一个Element对象。语法如下：

|  |
| --- |
| document.createElement(tagName); |

其中参数tagName为字符串值，用来指明创建元素的类型。createElement()方法要与appendChild()或insertBefore()方法联合使用，将元素显示在页面中。

**插入结点appendChild()**

**appendChild()**方法可以在指定结点的最后一个子结点列表之后**添加**一个新的子结点。语法如下：

|  |
| --- |
| elementNode.appendChild(newNode); |

**插入结点insertBefore()**

**insertBefore()**方法可以在已有的子结点前**插入**一个新的子结点。语法如下：

|  |
| --- |
| elementNode.insertBefore(newNode, node); |

**删除结点removeChild()**

**removeChild()**方法可以从子结点列表中**删除**某个结点，如删除成功，此方法可返回被删除的结点，如失败则返回null。语法如下：

|  |
| --- |
| elementNode.removeChild(node); |

**替换结点replaceChild()**

**replaceChild()**方法可以实现子结点的**替换**，返回被替换对象的引用。语法如下：

|  |
| --- |
| elementNode.replaceChild(newNode, oldNode); |

当oldNode被替换时，所有与之相关的属性内容都将被移出。