1 JavaScript简介

JavaScript 是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。

脚本语言：解释执行。
动态语言：解释执行 JavaScript 代码的时候才确定数据类型。
弱类型语言：变量的类型由实际的数据来确定，任何变量可以保存任意类型的数据；不同类型的数据之间可以进行运算。
基于原型：基于原型对象继承。

JavaScript 用于制作 web 页面交互效果，提升用户体验。

前端三层：

语言	层级	作用
HTML	结构层	写页面结构内容
CSS	样式层	写页面样式
JavaScript	行为层	写页面动态特效

JavaScript 的组成部分：

1. ECMAScript：JavaScript 的核心语法。
2. DOM：文档对象模型，规定了一套管理 HTML 文档的机制。
3. BOM：浏览器对象模型，规定了操作浏览器的语法。

2 引入JavaScript的方式

JavaScript 程序要引入到 HTML 页面中，运行页面的时候，这个页面上的 JavaScript 也一起运行。

JavaScript 有两种常见的引入方式：

1. 外部引入：在 `<script></script>` 标签中，使用 `src` 属性引入外部的 `.js` 文件。可以写在任意位置，一个页面可以有任意数量的 `script` 标签。推荐写在 `head` 标签中。例如：

```
<head>
    <script src="myScript.js"></script>
</head>
```

2. 内部引入：在 `<script></script>` 标签中写 JavaScript 代码。可以写在任意位置，一个页面可以有任意数量的 `script` 标签。推荐写在 `</body>` 之前。例如：

```
<body>
    <!-- HTML 文档 -->
    <script>
        // JavaScript 代码
    </script>
</body>
```

外部引入和内部引入不能共用同一个 `script` 标签。当共用时，外部引入生效，内部引入不生效。

`script` 标签有一个 `type` 属性，用于指定引入脚本的类型。JavaScript 脚本的 `type` 属性值为 `text/javascript`，引入 JavaScript 时此属性可以省略。

3 JavaScript输出方式

3.1 输出到控制台

```
console.log(); // 在括号中写输出的内容
```

每次调用输出一行。可以有多个参数，多个参数之间用逗号隔开，同一条语句中的多个参数输出在同一行。

此语句不解析 HTML 标签。

3.2 输出到HTML页面

```
document.write(); // 在括号中写输出的内容
```

此语句中的内容输出到当前`script`标签后面。

此语句可以解析 HTML 标签。

3.3 弹出框

警告框：`alert()`函数。显示带有一段信息和一个确认按钮的警告框，信息内容由参数指定。

动态输入框：`prompt()`函数。弹出框内有一个输入框，将用户输入的内容作为返回值返回，返回类型为字符串。第一个参数是提示信息，第二个参数是输入框中的默认值，默认值可以省略。

提示框：`confirm`函数。显示带有提示信息以及确认按钮和取消按钮的对话框，提示信息的内容由参数指定。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`。

4 JavaScript语法

1. JavaScript 大小写敏感。
2. JavaScript 语句以分号或换行结尾。推荐一行只写一条语句，以分号结尾。
3. JavaScript 不识别多余的空格。

5 标识符

标识符的命名规则：

1. 标识符必须由字母、数字、下划线`_`、美元符号`$`组成。
2. 不能以数字开头。
3. 不能用 JavaScript 中的关键字和保留字。
4. 标识符大小写敏感。

标识符的命名规范：

1. 使用有意义的名字。
2. 使用驼峰命名法：当标识符由多个单词构成时，第一个单词首字母小写，其他单词首字母大写。
3. 不要使用单字符标识符。只允许在循环中使用单字符标识符。
4. 不建议名字太长。

6 变量

值可以改变的量称为变量。变量的作用是存储数据。

使用`var`关键字声明变量，并指定变量的名称。例如：

```
var a;
```

声明变量后可以给变量赋值，使用`=`将右侧的值赋给左侧的变量。例如：

```
var a;
a = 100;
```

可以在声明变量的同时给变量赋值，称为变量的初始化。例如：

```
var a = 100;
```

一条语句可以声明多个变量，多个变量之间用逗号隔开。例如：

```
var a = 100, b, c;
```

声明变量但不赋值时，变量的值为`undefined`。

变量必须先声明后使用，未声明的变量不能使用，使用未声明的变量会报错。

重复声明变量时，变量的值不会丢失。但重新赋值时，变量的值会改变，原来的值丢失。

变量声明会在代码执行之前提升到所在作用域的最前面（声明提升），变量赋值不提升。因此，下面的代码不会报错：

```
console.log(x); // x 为 undefined
var x = 10; // x 为 10
```

上述代码相当于：

```
var x; // 变量声明提升到所在作用域的最前面
console.log(x);
x = 10; // 变量赋值不提升
```

也可以使用`let`关键词声明变量。`let`与`var`的区别如下：

1. `var`声明的变量只有全局变量和局部变量，没有块级作用域变量；`let`声明的变量有全局变量和局部变量，还有块级作用域变量。
2. `var`可以重复声明变量，`let`不可以。
3. 用`var`定义的全局变量会作为`window`对象的属性，用`let`定义的全局变量不会作为`window`对象的属性。

7 常量

字面常量：直接在代码中出现的字面量。例如：

```
var num = 10; // 数值型字面量
var name = '李白'; // 字符串字面量
var bool = true; // boolean 字面量
```

用`const`关键字声明常量。定义的常量只能访问、不能修改，因此需要在声明的同时初始化。

常量的命名规范：所有字母全部大写，多个单词用下划线分割。例如：

```
const MAX_AGE = 60;
```

常量不能修改，修改常量会报错。

常量没有声明提升，因此调用不能写在声明之前。

8 注释

单行注释：

```
// 单行注释
```

多行注释：

```
/* 多行注释 */
```

文档注释：

```
/**
 *  文档注释
 */
```

9 数据类型

JavaScript 的数据类型分为基本类型和引用类型。

基本类型：

1. `number`：数字类型。
2. `string`：字符串类型。
3. `boolean`：布尔类型。
4. `undefined`：未定义类型。
5. `null`：空类型。

引用类型：`object`，对象类型。

可以使用`typeof`运算符得到当前值的数据类型，返回值的类型是字符串，`typeof`后面跟变量或值。例如：

```
var a = 10;
console.log(typeof a);
console.log(typeof 3);
```

9.1 `number`类型

JavaScript 中所有数字都是`number`类型，不区分整型、浮点型。

`number`类型特殊值：

1. `NaN`：Not a Number，表示不是一个数值。任何数值与`NaN`计算，结果都是`NaN`。`NaN`不等于自身。
2. `Infinity`：正无穷。分母为 0，分子为正数时，结果为`Infinity`。
3. `-Infinity`：负无穷。分母为 0，分子为负数时，结果为`-Infinity`。

9.2 `string`类型

由引号包含的任意内容都是`string`类型。引号可以是单引号，也可以是双引号，但要成对出现。

字符串之间可以相互嵌套，单引号内可以包含双引号，双引号内可以包含单引号。但是单引号内不能包含单引号，双引号内不能包含双引号。例如：

```
var str1 = "hello 'world'!";
var str2 = 'hello "world"!';
```

如果想在单引号内使用单引号，可以使用转义字符。转义字符由`\`引导，代表一个特定的字符。常见的转义字符如下所示：

转义字符	含义
`\'`	单引号
`\"`	双引号
`\\`	反斜杠
`\n`	换行符
`\t`	制表符，长度为 1-4 个空格
`\b`	退格

9.3 `boolean`类型

`boolean`类型只有`true`、`false`两个值。`true`表示真，可以转换为数字 1；`false`表示假，可以转换为数字 0。

9.4 `undefined`类型

变量已声明但没有赋值时，其类型为`undefined`，其值也是`undefined`。

`undefined`类型值与任何`number`类型值做计算，结果为`NaN`。

9.5 `null`类型

`null`类型只有一个值`null`，表示引用类型变量没有指向任何一个对象。

`null`类型使用`typeof`返回`"object"`。

`null`做计算时转换为 0。

9.6 引用数据类型

引用数据类型只有`object`一种，`object`类型又可以细分：

1. `array`：数组类型。
2. `function`：函数类型。
3. `object`：对象类型。

`array`类型使用`typeof`返回`"object"`，`function`类型使用`typeof`返回`"function"`。

10 运算符

10.1 算数运算符

运算符	描述	操作数
+	加法	只有一个操作数时，表示正号。 两个操作数都是 number 类型时，表示加号。操作数中有 NaN 时，返回值为 NaN 。 操作数中有 string 类型时，将两个操作数都转换成 string 类型，进行字符串拼接，返回值为 string 类型。 操作数中有 boolean 类型时， true 转换成数字1， false 转换成数字0。 操作数中有 undefined 类型时，返回值为 NaN 。 操作数中有 null 类型时， null 类型值转换成数字0。 操作数中有数组或对象时，将两个操作数都转换成 string 类型，进行字符串拼接。
-	减法	只有一个操作数时，表示负号。 两个操作数都是 number 类型时，表示减号。操作数中有 NaN 时，返回值为 NaN 。 操作数中有 string 类型时，将 string 类型值转换成 number 类型，做减法。如果字符串不能转换成数字，返回值为 NaN 。 操作数中有 boolean 类型时， true 转换成数字1， false 转换成数字0。 操作数中有 undefined 类型时，返回值为 NaN 。 操作数中有 null 类型时， null 类型值转换成数字0。 操作数中有数组时，将数组转换为 number 类型进行计算。数组中只有一个元素时正常运算；数组中有多个元素时，返回值为 NaN 。 操作数中有对象时，返回值为 NaN 。
*	乘法	二元运算符 两个操作数都是 number 类型时，表示乘号。操作数中有 NaN 时，返回值为 NaN 。 操作数中有 string 类型时，将 string 类型值转换成 number 类型，做乘法。如果字符串不能转换成数字，返回值为 NaN 。 操作数中有 boolean 类型时， true 转换成数字1， false 转换成数字0。 操作数中有 undefined 类型时，返回值为 NaN 。 操作数中有 null 类型时， null 类型值转换成数字0。 操作数中有数组时，将数组转换为 number 类型进行计算。数组中只有一个元素时正常运算；数组中有多个元素时，返回值为 NaN 。 操作数中有对象时，返回值为 NaN 。
/	除法	二元运算符 两个操作数都是 number 类型时，表示除号。操作数中有 NaN 时，返回值为 NaN 。 操作数中有 string 类型时，将 string 类型值转换成 number 类型，做除法。如果字符串不能转换成数字，返回值为 NaN 。 操作数中有 boolean 类型时， true 转换成数字1， false 转换成数字0。 操作数中有 undefined 类型时，返回值为 NaN 。 操作数中有 null 类型时， null 类型值转换成数字0。 操作数中有数组时，将数组转换为 number 类型进行计算。数组中只有一个元素时正常运算；数组中有多个元素时，返回值为 NaN 。 操作数中有对象时，返回值为 NaN 。

运算符	描述	操作数
%	取余	二元运算符 两个操作数都是number类型时，表示取余。操作数中有NaN时，返回值为NaN。 操作数中有string类型时，将string类型值转换成number类型，做取余运算。如果字符串不能转换成数字，返回值为NaN。 操作数中有boolean类型时，true转换成数字1，false转换成数字0。 操作数中有undefined类型时，返回值为NaN。 操作数中有null类型时，null类型值转换成数字0。 操作数中有数组时，将数组转换为number类型进行计算。数组中只有一个元素时正常运算；数组中有多个元素时，返回值为NaN。 操作数中有对象时，返回值为NaN。
++	自增	一元运算符，可以在操作数前面，也可以在操作数后面。 无论在操作数前面还是后面，都会将操作数加1。 放在操作数前面时，先加1后返回，返回的是加1后的值。 放在操作数后面时，先返回后加1，返回的是加1前的值。
--	自减	一元运算符，可以在操作数前面，也可以在操作数后面。 无论在操作数前面还是后面，都会将操作数减1。 放在操作数前面时，先减1后返回，返回的是减1后的值。 放在操作数后面时，先返回后减1，返回的是减1前的值。

10.2 关系运算符

关系运算符比较两个值，返回boolean值，true表示满足比较条件，false表示不满足比较条件。

运算符	描述
==	等于（只判断值）
===	绝对等于（既判断值又判断类型）
!=	不等于
!==	不绝对等于
>	大于
<	小于
>=	大于等于
<=	小于等于

当两个操作数都是number类型时，正常比较数值大小。

当两个操作数都是string类型时，将各个字符转换为 Unicode 编码，逐位比较。

特殊值的比较：

- null == undefined返回值为true，null === undefined返回值为false。
- null == 0与undefined == 0返回值为false。

- `false == 0`、`false == ''`、`'' == 0`返回值为`true`。
- `NaN`与任何值做比较都返回`false`。`NaN == NaN`也返回`false`。

10.3 逻辑运算符

逻辑运算符的操作数是`boolean`类型值，返回值是`boolean`类型。

运算符	描述
`&&`	逻辑与
`||`	逻辑或
`!`	逻辑非

优先级：`!`最高，`&&`次之，`||`最低。

`&&`和`||`存在短路逻辑。对于`&&`，当第一个表达式的值为`false`，就不会计算第二个表达式。对于`||`，当第一个表达式的值为`true`，就不会计算第二个表达式。

当`&&`和`||`的两个操作数中有非布尔值时，将其转换成布尔值进行计算，并且返回值为操作数原值：

1. `&&`运算：如果第一个值为`true`，则返回第二个值；如果第一个值为`false`，则返回第一个值。
2. `||`运算：如果第一个值为`true`，则返回第一个值；如果第一个值为`false`，则返回第二个值。

下面 5 种值转换成布尔值时为`false`，除此之外都转换成`true`：

1. `undefined`
2. `null`
3. 数字0
4. `NaN`
5. 空字符串：`''`或`""`

例如：

```
/* 第一个表达式为 true，逻辑与运算返回第二个值 */
console.log('ok' && null); // null
console.log(1 && undefined); // undefined
/* 第一个表达式为 false，逻辑与运算返回第一个值 */
console.log(0 && 6); // 0
console.log(null && 6); // null
/* 第一个表达式为 true，逻辑或运算返回第一个值 */
console.log('false' || 6); // 'false'
/* 第一个表达式为 false，逻辑或运算返回第二个值 */
console.log(undefined || 6); // 6
console.log(NaN || 6);  // 6
```

10.4 赋值运算符

赋值运算符有：`=`、`+=`、`-=`、`*=`、`/=`、`%=`、`&=`、`|=`、`^=`、`>>=`。

10.5 位运算符

位运算符直接对底层的二进制数操作，效率较高。

位运算只能操作整数。如果对小数操作，先将小数转换为整数，再做操作。位运算具有取整的功能。

运算符	描述
`&`	按位与
`|`	按位或
`~`	按位取反（本质是取负减一）
`^`	按位异或
`>>`	带符号右移（本质是除以 2 的 n 次幂，并取整）
`<<`	左移（本质是乘以 2 的 n 次幂）
`>>>`	无符号右移

连续两次按位取反可以起到取整的作用。例如：

```
var num = 3.2;
console.log(~~num); // 3
```

按位异或可以用于交换两个变量的值。例如：

```
var num1 = 1;
var num2 = 2;
num1 = num1 ^ num2;
num2 = num1 ^ num2;
num1 = num1 ^ num2;
```

10.6 条件运算符

语法：`布尔值 : 表达式1 ? 表达式2;`

当布尔值为`true`时，执行表达式1并返回其值；当布尔值为`false`时，执行表达式2并返回其值。

条件运算符可以嵌套。例如：

```
const MIN = 1;
const MAX = 100;
// 生成三个 1 到 100 之间的随机数
var ran1 = parseInt(Math.random() * (MAX - MIN + 1) + MIN);
var ran2 = parseInt(Math.random() * (MAX - MIN + 1) + MIN);
var ran3 = parseInt(Math.random() * (MAX - MIN + 1) + MIN);
// 计算这三个随机数中的最大值
var max = ran1 > ran2 ? (ran1 > ran3 ? ran1 : ran3) : (ran2 > ran3 ? ran2 :
ran3);
```

10.7 运算符优先级

优先级	运算符	说明	结合性
1	`[]`、`.`、`()`	数组索引、字段访问、函数调用、表达式分组	从左向右
2	`++`、`--`、`-`、`~`、`!`、`delete`、`new`、`typeof`、`void`	自增、自减、负号、按位取反、逻辑非、对象创建、返回数据类型、未定义的值	从右向左
3	`*`、`/`、`%`	乘、除、取余	从左向右
4	`+`、`-`	加、减	从左向右
5	`<<`、`>>`、`>>>`	左移、带符号右移、无符号右移	从左向右
6	`<`、`<=`、`>`、`>=`、`instanceof`	小于、小于等于、大于、大于等于、是否为特定类的实例	从左向右
7	`==`、`!=`、`===`、`!==`	相等、不等、全等、不全等	从左向右
8	`&`	按位与	从左向右
9	`^`	按位异或	从左向右
10	`|`	按位或	从左向右
11	`&&`	逻辑与	从左向右
12	`||`	按位或	从左向右
13	`?:`	条件运算符	从右向左
14	`=`、`+=`、`-=`、`*=`、`/=`、`%=`、`&=`、`|=`、`^=`、`>>=`	赋值运算符	从右向左

优先级	运算符	说明	结合性
15	,	分割多个表达式	按优先级计算，然后从右向左

总结：

1. 优先级相同时从左向右计算。
2. 乘除优先级大于加减。
3. 小括号优先级最高。
4. 一元运算符优先级高于二元运算符，二元运算符优先级高于条件运算符。
5. 赋值运算符优先级最低。
6. 逻辑与的优先级大于逻辑或。

11 数据类型转换

11.1 隐式类型转换

转换为`number`类型：`-`、`*`、`/`、`%`运算符的操作数中，非`number`类型会隐式转换成`number`类型。

转换为`string`类型：`+`运算符遇到字符串，进行字符串拼接。

转换为`boolean`类型：条件运算符中的条件、逻辑运算符的操作数、`if`语句的条件，类型不是`boolean`类型时，会自动转换为`boolean`类型。

11.2 强制类型转换

11.2.1 `string`类型转换为`number`类型

`parseInt`函数将字符串转换为整数。

注意点：

1. 小数部分直接去掉，不会四舍五入。
2. 如果字符串以空格开头，忽略空格。
3. 如果字符串以字符 0 开头，忽略 0。
4. 当包含数字和其他字符时，如果以数字开头，返回数字部分；如果以其他字符开头，返回`NaN`。

例如：

```
console.log(parseInt('6.779')); // 6
console.log(parseInt('6.779.998')); // 6
console.log(parseInt('6ok')); // 6
console.log(parseInt('ok6')); // NaN
console.log(parseInt('  6  6  ')); // 6
console.log(parseInt('060')); // 60
```

`parseFloat`函数将字符串转换为浮点数。

11.2.2 `Number`方法

`Number`方法将参数转换为`number`类型。

注意点：

1. 当参数为字符串时，只要字符串中包含不是数字的字符，返回`NaN`。
2. 当参数为字符串时，忽略前后的空格，忽略数字前面的 0。
3. 布尔值`true`转换为 1，`false`转换为 0。
4. 空字符串、`null`转换为 0。
5. `undefined`转换为`NaN`。

例如：

```
Number('66.789'); // 66.789
Number('66ok'); // NaN
Number('ok66'); // NaN
Number('  6  6  '); // NaN
Number('  66  '); // 66
Number('0660'); // 660
Number(true); // 1
Number(false); // 0
Number(undefined); // NaN
Number(null); // 0
Number(''); // 0
```

11.2.3 `String`方法

`String`方法将参数转换为`string`类型。

注意点：

1. 对象转换成字符串为`[object Object]`。
2. 数组转换为字符串，将方括号去掉，直接取出数组元素，中间用逗号隔开。

例如：

```
String(66); // '66'
String(true); // 'true'
String(false); // 'false'
String(undefined); // 'undefined'
String(null); // 'null'
String([1, 2, 3]); // '1,2,3'
String({name:Alice}); // '[object Object]'
```

11.2.4 `Boolean`方法

`Boolean`方法将参数转换为`boolean`类型。

转换为`false`的有：`undefined`、`null`、数字0、`NaN`、空字符串。

其余都转换为`true`。

12 流程控制

12.1 分支

12.1.1 `if`：单分支结构

语法：

```
if (条件)
{
    // 代码
}
```

条件为`boolean`类型值。若不是`boolean`类型，则将其隐式转换为`boolean`类型。

当条件为`true`时，进入`if`分支，执行大括号中的代码；当条件为`false`时，跳过`if`分支，执行大括号后面的代码。

当大括号中只有一条语句时，大括号可以省略。不建议省略大括号。

12.1.2 `if-else`：双分支结构

语法：

```
if (条件)
{
    // 代码1
}
else
{
    // 代码2
}
```

当条件为`true`时，执行代码1；当条件为`false`时，执行代码2。

`else`必须与`if`配对，不能单独使用。

12.1.3 `if-else if`：多分支结构

语法：

```
if (条件1)
{
    // 代码1
}
else if (条件2)
{
    // 代码2
}
// ...
else if (条件n)
{
    // 代码n
}
else
{
    // 代码
}
```

从上到下依次判断条件，当某个条件为`true`时，执行相应的代码，跳过剩余的条件和代码；如果所有条件都为`false`，执行`else`对应的代码。

条件的个数没有限制。

`else`可以省略。

分支结构可以相互嵌套。

12.1.4 `switch-case`：多分支结构

语法：

```
switch (表达式)
{
    case 表达式1:
        // 代码1
    case 表达式2:
        // 代码2
    // ...
    case 表达式n:
        // 代码n
    default:
        // 代码块
}
```

括号里的表达式可以返回任何数据类型，与`case`后面的表达式返回值做等值比较（`===`）。从上到下依次做比较，当某个`case`分支匹配成功时，执行相应分支的代码，并依次执行此后的所有分支而不需要匹配表达式。如果所有`case`分支都匹配失败，则执行`default`分支。

`default`分支可以省略。

switch-case 结构可以执行多个分支。

使用`break`关键词可以退出 switch-case 结构。如果只想执行一个分支，可以在每个`case`的代码最后添加`break`语句。

12.2 循环

12.2.1 `for`循环

语法：

```
for (初始化循环变量;循环条件;改变循环变量)
{
    // 循环体
}
```

循环条件为`boolean`类型值，如果不是`boolean`类型，则隐式转换为`boolean`类型。

执行顺序：首先初始化循环变量，然后判断循环条件。当循环条件为`true`时，执行循环体，之后改变循环变量，然后再次判断循环条件。如此循环往复，直到循环条件为`false`时，退出循环。

小括号中的三个表达式都可以省略，但是两个分号不能省略。

12.2.2 `while`循环

语法：

```
while（条件）
{
    // 循环体
}
```

当条件为`true`时，执行循环体；当条件为`false`时，退出循环。

执行顺序：首先判断条件，如果条件为`true`，则执行循环体。执行完一次循环体后，再次判断条件，如果条件为`true`，再次执行循环体。如此循环往复，直到条件为`false`，退出循环。

`for`循环适用于循环次数确定的情况，`while`循环适用于循环次数不确定的情况。

12.2.3 `do-while`循环

语法：

```
do
{
    // 循环体
} while（条件）;
```

执行顺序：首先执行循环体，执行完循环体后判断条件，当条件为`true`时，再次执行循环体；当条件为`false`时，退出循环。

12.2.4 `break`关键字

`break`关键字在`switch`语句和循环体中使用。在`switch`语句中使用时，用于跳出`switch`语句。在循环体中使用时，用于结束当前循环。

12.2.5 `continue`关键字

`continue`关键字在循环体中使用，用于结束本次循环、继续下一次循环。

13 函数

13.1 函数的概念

函数是事件驱动的或者被调用时执行的可重复使用的代码块。

函数的作用：

1. 控制程序执行时间。
2. 重复使用代码。
3. 功能模块化，易于维护和修改。

13.2 函数的声明

`function`命令法：

```
function 函数名(参数)
{
    // 函数体
}
```

函数表达式法：

```
// 声明了一个匿名函数对象，存在于内存中，变量保存的是函数对象的地址
// 通过变量保存的函数对象的地址去访问内存中的匿名函数
var 变量名 = function(参数) {
    // 函数体
};
```

构造函数法：

```
var 变量名 = new Function('参数1','参数2',...,'参数n','函数体');
```

参数可以没有，可以只有一个，也可以有多个。多个参数之间用逗号隔开，最后一个参数后面不加逗号。

函数声明有提升性，代码执行之前会把函数声明提升到作用域的最前面。所以在`function`命令法中，函数调用可以写在函数声明前面。

函数表达式法和构造函数法中，函数调用不可以写在函数声明前面。因为变量声明提升，而赋值不提升。

JavaScript 中没有函数重载。函数名相同时，后面的函数会覆盖前面的函数。

13.3 函数的调用

语法：

```
函数名(参数);
```

13.4 函数的参数

函数声明时的参数叫做**形式参数**，简称**形参**。形参用于接收外部的数据，供函数内部使用。形参是局部变量，不用`var`声明，其作用域为当前函数，只能在当前函数中使用，函数外不能使用。

函数调用时的参数叫做**实际参数**，简称**实参**。函数调用时，把实参的值赋给形参，让实参的值在函数中参与运算，这一过程叫做**参数传递**，简称**传参**。

实参与形参最好一一对应。当实参个数多于形参个数时，多余的实参直接忽略；当实参个数少于形参个数时，未赋值的形参为`undefined`。

形参列表中形参的名字最好不要重复。如果形参名重复，在函数调用时，相当于对同一个变量先后赋值两次，后面的值生效。

13.5 函数的返回值

函数执行完毕后会产生一个结果，该结果称为函数的**返回值**。函数的返回值可以赋值给变量，可以作为表达式的操作数使用，可以作为其他函数的实参，也可以作为其他函数的返回值。

函数可以有返回值，也可以没有返回值。当函数没有返回值时，默认返回`undefined`。

当函数有返回值时，在函数体中使用`return`关键字返回相应的值。语法：

```
return 表达式;
```

一旦执行了`return`语句，函数立即返回，不再执行函数中的其他代码。

没有返回值时也可以使用`return`关键字来结束函数，此时不需要指定返回值。语法：

```
return;
```

函数的返回值可以是任意类型。

13.6 深入理解函数调用

JavaScript 的内存模型：

1. 堆（heap）：保存对象数据，由 JavaScript 引擎动态管理。数据在堆中是不连续的。
2. 栈（stack）：保存局部变量，静态管理。数据在栈中是连续的，后进先出。

函数调用的过程：

1. 将函数的栈帧压入栈中。栈帧中保存当前函数中所有的局部变量。
2. 参数传递。将实参的值赋给形参。
3. 执行函数体。
4. 函数执行完毕，弹出对应的栈帧。

基本数据类型作为参数时，参数传递过程采用值传递，将实参的值赋给形参；引用数据类型作为参数时，参数传递过程采用引用传递。

13.7 局部变量与全局变量

函数的形参以及在函数内声明的变量属于局部变量。局部变量的作用域为声明它的函数内部，只能在所在函数内使用。局部变量的生命周期依赖于函数调用，函数调用开始时局部变量被创建，函数结束后局部变量被销毁。局部变量存在于栈中。

函数外声明的变量属于全局变量。全局变量的作用域为整个`script`代码块。全局变量的生命周期依赖于`window`对象，浏览器窗口开启时全局变量被创建，窗口关闭后全局变量被销毁。全局变量存在于堆中。

当局部变量和全局变量重名时，在函数内局部变量的优先级更高。要想在函数内访问重名的全局变量，可以使用`window.全局变量名`或`this.全局变量名`。

全局变量不利于维护，可能导致变量之间数据的覆盖、污染等问题。因此优先使用局部变量。

13.8 函数的执行符

函数名、函数变量名代表了函数对象自身，通过指针指向函数对象。

在函数名后面加`()`就可以调用函数，将`()`叫做函数的执行符。函数的执行符可以让函数对象执行函数体中的内容。

13.9 回调函数

可以将函数 A 的函数名作为参数传递给函数 B，在函数 B 中调用函数 A。这样的函数 A 叫做**回调函数**。例如：

```
function fn1()
{
```

```
    // 函数体
}

function test(fn)
{
    fn();
}

// 传入 fn1，fn1 为回调函数
test(fn1);

// 传入匿名函数，该匿名函数为回调函数
test(function() {
   // 函数体
});
```

13.10 匿名自执行函数

语法：

```
(function() {
    // 函数体
})();

(function() {
    // 函数体
}());

// 在 function 关键字之前加任何运算符都可以
!function() {
    // 函数体
}();

var num = function() {
    // 函数体
}();
```

匿名自执行函数也叫做**匿名立即执行函数**，它作为函数存在，只执行一次。

匿名自执行函数的作用：创建块级作用域，可以在其中定义局部变量，减少全局变量的数量。

13.11 局部函数

定义的所有全局变量和全局函数都会作为`window`对象的属性和功能存在，而局部变量和局部函数不会。

定义在函数体内的函数叫做**局部函数**。局部函数的作用域在定义它的函数体内，只能在定义它的函数体内使用。例如：

```
function fn()
{
    // 定义局部函数
    function fn1()
    {
        // 函数体
    }
    // 调用局部函数
    fn1();
}
```

可以将局部函数返回，在函数体外执行局部函数。例如：

```
function fn()
{
    var fn1 = function() {
        // 函数体
    }
    // 返回局部函数
    return fn1;
}

// 接收局部函数
var fn2 = fn();
// 调用局部函数
fn2();
```

13.12 函数的属性

函数是一种对象，有属性和方法。函数的属性通过`函数名.属性名`引用。

13.12.1 `name`

`name`属性表示函数名，`string`类型。

对于有名函数，`name`属性值为函数名。对于匿名函数，如果用一个变量指向匿名函数，`name`属性值为变量名；如果没有变量指向匿名函数，`name`属性值为空字符串。

`name`属性的应用：根据函数名的不同，给出不同的回调方案。例如：

```
function test(fn)
{
    var funName = fn.name;
    if (funName === 'fn1')
    {
        fn();
        fn();
    }
    else
    {
        fn();
    }
}
```

13.12.2 `length`

`length`属性表示函数的形参个数，`number`类型。

13.12.3 `arguments`

`arguments`属性是函数体中内置的属性，是一个对象，表示实参列表。

`arguments`对象的属性：

1. `arguments.length`：实参个数。
2. `arguments.callee`：函数本身。在严格模式下不能使用。

当函数的实参个数可变时，不能使用形参来接收实参。这时可以不写形参，用`arguments`属性来获取实参。例如：

```
// 求任意个数字的和
function sum()
{
    var result = 0;
    var count = arguments.length; // 获取实参个数
    for (var i = 0; i < count; i++)
    {
        result += arguments[i]; // 用序号访问 arguments 对象中的实参，序号从 0 开始
    }
    return result;
}
```

13.13 函数的方法

13.13.1 `toString`方法

`toString`方法返回当前函数对象的字符串表示形式。例如：

```
function test()
{
    console.log("test");
}

console.log(test.toString());
// 输出：
// "function test()
// {
//     console.log("test");
// }"
```

13.13.2 `call`方法

如果两个对象具有完全一致的功能，函数完全相同，但这两个函数是不同的函数对象。同样的函数在内存中保存两次，造成代码冗余和内存浪费。例如：

```
var gang = {
    name: "小刚",
    age: 16,
```

```
    study: function() {
        console.log(this.name + "喜欢学习");
    }
}

var ming = {
    name: "小明",
    age: 16,
    study: function() {
        console.log(this.name + "喜欢学习");
    }
}

console.log(gang.study === ming.study); // false
```

通过`call`方法可以解决这一问题。`call`方法将当前函数借给其他对象使用。如果不同的对象具有相同的功能，可以只保留一个对象的函数，其他对象通过`call`方法借用该对象的函数。例如：

```
// 创建小刚对象，包含 study 函数
var gang = {
    name: "小刚",
    age: 16,
    study: function() {
        console.log(this.name + "喜欢学习");
    }
}
// 创建小明对象，不包含 study 函数
var ming = {
    name: "小明",
    age: 16
}
// 使用 call 方法，将 gang 的 study 方法借给 ming 使用
gang.study.call(ming);
```

如果借用的函数有参数，将参数依次写在借用者后面。例如：

```
var obj1 = {
    name : "tom",
    sum : function(num1, num2) {
        console.log(num1 + num2);
    }
}

var obj2 = {
    name : "jack"
}

obj1.sum.call(obj2, 1, 2); // 3
```

`call`方法的本质：改变函数体内的`this`的指向，指向借用者对象。通过参数指定函数的调用者。

13.13.3 `apply`方法

`apply`方法将当前函数借给其他对象使用。`apply`方法的使用与`call`方法基本一致，区别在于，如果借用的函数有参数，`apply`方法要求将这些参数写在数组中。例如：

```
var obj1 = {
    name : "tom",
    sum : function(num1, num2) {
        console.log(num1 + num2);
    }
}

var obj2 = {
    name : "jack"
}

obj1.sum.apply(obj2, [1, 2]); // 3
```

13.13.4 `bind`方法

`bind`方法利用当前对象的函数，帮助其他对象创建一个新函数。例如：

```
var obj1 = {
    name : "tom",
    sum : function(num1, num2) {
        console.log(num1 + num2);
    },
    printName : function() {
        console.log(this.name);
    }
}

var obj2 = {
    name : "jack"
}

// 有参数的函数
var sum = obj1.sum.bind(obj2); // 利用 obj1.bind 函数，为 obj2 创建 sum 函数
sum(1, 2); // 3

// 无参函数
var printName = obj1.printName.bind(obj2);
printName(); // jack
```

`bind`方法的本质：生成新的函数对象，这个函数体中的`this`指向`bind`函数的实参对象。

13.14 函数的递归调用

函数直接或间接地调用自身的过程，称为递归调用。

递归调用解决的问题的特点：

1. 问题可以分解为若干个子问题。
2. 子问题的解决方式和问题本身的解决方式一致。
3. 最终问题的解决依赖于子问题的解决。

4. 必须存在一个子问题能够直接解决。（存在递归出口）

优点：思路简单，代码实现简单。

缺点：效率比较低，非常消耗栈内存。

大部分递归调用实现的功能都可以用循环来实现。但是，如果递归的层数不确定，就不能使用循环了。

可以用函数的`arguments`属性实现递归，以避免递归调用的安全隐患。例如：

```
// 不使用 arguments 属性，存在安全隐患
var fibo = function(n) {
    if (n === 1 || n === 2)
    {
        return 1;
    }
    return fibo(n - 1) + fibo(n - 2);
}
var fibo2 = fibo;
fibo = null;
fibo2(5); // 报错，因为函数体中使用了 fibo 函数，而 fibo 为 null

// 使用 arguments 属性以避免上述情况
var fibonacci = function(n) {
    if (n === 1 || n === 2)
    {
        return 1;
    }
    // arguments.callee 表示函数本身
    // arguments.callee 在严格模式下不能使用，所以也要慎用
    return arguments.callee(n - 1) + arguments.callee(n - 2);
}
```

14 数组

14.1 数组的概念

数组是一组有序的变量的集合。数组中的变量称为**元素**。

数组是一种特殊的对象，是一组键值对的集合。数组中的键叫做**下标**或**索引**，是`number`类型，从 0 开始，依次递增。数组中的值即元素，可以是任意类型。

数组存在于堆内存中，数组元素在内存中连续存放。这使得随机访问数组元素的效率很高，但删除和插入元素的效率比较低。

访问数组元素的语法：`数组变量名[下标]`或`数组变量名['下标']`。

14.2 数组的定义

语法：

```
// 表达式方式
var 变量名 = []; // 空数组
var 变量名 = [1, true, "1", null, undefined, function(){}, {}, [1, 2]]; // 有初始值的数组
// 构造函数法
var 变量名 = new Array(元素1, 元素2, ...); // 多个参数时，为数组元素赋值
var 变量名 = new Array(长度); // 只有一个参数时，指定数组的长度
```

变量名保存的是数组第一个元素的首地址。

14.3 数组的`length`属性

数组的`length`属性表示数组中元素的个数，取值为大于等于 0 的整数，可读可写。通过`数组名.length`访问该属性。

`length`属性的取值范围：$[0, 2^{32} - 1]$

对数组元素赋值的时候，下标可以不连续，未赋值的元素为`undefined`。`length`属性统计到最后一个非`undefined`的元素为止。例如：

```
var arr = [];
arr[0] = 0;
arr[10] = 10;
console.log(arr[3]); // undefined
console.log(arr.length); // 11
```

如果数组元素的下标超过`length`属性的取值范围，该元素就不会被`length`属性统计。例如：

```
var arr = []
arr[5] = 5;
arr[9999999999] = 'hello';
console.log(arr.length); // 6
```

`length`属性的值可以修改。如果`length`属性的值变大，则数组扩容，在数组后面添加值为`undefined`的元素；如果`length`属性的值变小，则数组缩容，多余的元素直接截断。

可以给数组添加非数值型的键，非数值型的键不会被`length`属性统计。例如：

```
var arr = [];
arr[0] = 'hello';
arr.name = '数组';
console.log(arr.length); // 1
```

14.4 `in`操作符

`in`操作符用来判断某个字符串是否是某个对象中的键。如果是则返回`true`，如果不是则返回`false`。

语法：`字符串 in 对象`

例如：

```
var arr = [0, 1, 2, 3, 4];
console.log('0' in arr); // true
console.log(0 in arr); // true
console.log('5' in arr); // false
```

14.5 数组元素的操作

访问数组元素：`数组名[下标]`或`数组名['下标']`

删除数组元素：`delete 数组名[下标]`，将相应位置的元素设置为`undefined`，`length`属性不受影响。

14.6 数组的遍历

```
var array = [0, 1, 2, 3, 4];
const LENGTH = array.length;

// 1.使用 for 循环
for (let i = 0; i < LENGTH; i++)
{
    console.log(array[i]);
}

// 2.使用 for in 循环
for (var i in array) // i 代表下标
{
    console.log(array[i]);
}

// 3.使用 for of 循环
for (var element of array) // element 代表数组元素
{
    console.log(element);
}

// 4.使用数组对象的实例函数 forEach，forEach 函数的参数是一个回调函数
// 这个回调函数的第一个参数为数组元素，第二个参数为数组下标，第三个参数为被遍历的数组
array.forEach(function(value, index) {
    console.log('第' + (index + 1) + '个元素为：' + val);
});

function fn(value, index, arr)
{
    console.log('第' + (index + 1) + '个元素为：' + val);
}
array.forEach(fn);
```

14.7 数组的函数

14.7.1 `push`、`pop`函数

`push`函数：向数组尾部添加元素，返回数组新的长度。添加的元素数量和类型不限。例如：

```
var array = [0];
array.push(1, true, 'a'); // [0, 1, true, 'a']
```

`pop`函数：删除数组的最后一个元素，返回被删除的元素。如果数组为空，返回`undefined`。

14.7.2 `shift`、`unshift`函数

`shift`函数：删除数组的第一个元素，返回被删除的元素。如果数组为空，返回`undefined`。

`unshift`函数：向数组头部添加元素，返回数组新的长度。添加的元素数量和类型不限。

14.7.3 `join`函数

`join`函数：将数组的所有元素用指定的字符进行连接，形成一个字符串。参数为连接符，返回值为连接成的字符串。参数可选，当省略参数时，使用逗号作为默认的连接符。

例如：

```
var array = [0, 1, 2, 3, 4];
console.log(array.join('-')); // '0-1-2-3-4'
console.log(array.join()); // '0,1,2,3,4'
```

14.7.4 `indexOf`、`lastIndexOf`函数

`indexOf`函数：从前往后查找指定的元素。第一个参数为待查找的元素，第二个参数为起始索引。第二个参数可选，如果不指定起始索引，默认起始索引为 0。如果找到，返回第一个匹配的元素的下标；如果没找到，返回 -1。

`lastIndexOf`函数：从后往前查找指定的元素。第一个参数为待查找的元素，第二个参数为起始索引。第二个参数可选，如果不指定起始索引，默认从最后一个元素开始。如果找到，返回第一个匹配的元素的下标；如果没找到，返回 -1。

例如：

```
var array = [1, 2, 3, 4, 5, 6, 7];
console.log(array.indexOf(2)); // 1
console.log(array.indexOf(2, 2)); // -1
console.log(array.lastIndexOf(2, 3)); // 1
```

14.7.5 `slice`函数

`slice`函数：从数组中截取子数组，返回子数组。第一个参数为起始索引（包含），第二个参数为结束索引（不包含），两个参数均可省略。如果只有一个参数，则从该位置截取到末尾；如果都省略，复制当前数组。

例如：

```
var array = [1, 2, 3, 4, 5, 6, 7];
console.log(array.slice(2, 5)); // [3, 4, 5]
console.log(array.slice(5)); // [6, 7]
console.log(array.slice()); // [1, 2, 3, 4, 5, 6, 7]
```

参数可以为负数， -1 表示最后一个元素， -2 表示倒数第二个元素，以此类推。例如：

```
var array = [1, 2, 3, 4, 5, 6, 7];
console.log(array.slice(-5, -1)); // [3, 4, 5, 6]
console.log(array.slice(-3)); // [5, 6, 7]
```

如果起始索引位于结束索引后面，则返回空数组。

14.7.6 `splice`函数

`splice`函数：实现对数组的增、删、改操作。

插入元素：第一个参数为插入位置，第二个参数为 0，之后是插入的元素。插入的元素数量不限。返回空数组。例如：

```
var array = [1, 2, 3, 4, 5];
array.splice(3, 0, 7, 8); // [1, 2, 3, 4, 7, 8, 5]
```

删除元素：第一个参数为起始索引，第二个参数为删除的元素个数。返回被删除的元素组成的数组。例如：

```
var array = [1, 2, 3, 4, 7, 8, 5]
array.splice(4, 2); // [1, 2, 3, 4, 5]
```

修改操作：第一个参数为修改的位置，第二个参数为修改的个数，之后是用于替换的元素。返回被替换的元素组成的数组。例如：

```
var array = [1, 2, 3, 4, 5];
array.splice(3, 1, 7, 8); // [1, 2, 3, 7, 8, 5]
```

14.7.7 `reverse`函数

`reverse`函数：翻转当前数组。

例如：

```
var array = [1, 2, 3];
console.log(array.reverse()); // [3, 2, 1]
```

14.7.8 `sort`函数

`sort`函数：对数组进行排序。默认的排序规则为，将所有数组元素转换为字符串，再对字符串升序排序。

例如：

```
var array = [19, 15, 17, 9, 10, 0, 7, 15, 4, 12];
console.log(array.sort()); // [0, 10, 12, 15, 15, 17, 19, 4, 7, 9]
```

可以向`sort`函数提供回调函数，指定排序规则。回调函数有两个参数，代表数组中的元素，在回调函数中指定元素的比较规则。

升序规则：如果第一个参数小于第二个参数，则返回负数；如果二者相等，则返回 0；如果第一个参数大于第二个参数，则返回正数。

降序规则：如果第一个参数小于第二个参数，则返回正数；如果二者相等，则返回 0；如果第一个参数大于第二个参数，则返回负数。

例如：

```
function compareFn(val1, val2)
{
    if (val1 > val2)
    {
        return 1;
    }
    else if (val1 < val2)
    {
        return -1;
    }
    return 0;
}

var array = [19, 15, 17, 9, 10, 0, 7, 15, 4, 12];
console.log(array.sort(compareFn)); // [0, 4, 7, 9, 10, 12, 15, 15, 17, 19]
```

14.7.9 `map`函数

`map`函数：对数组中的所有元素进行某个操作，返回操作后的新数组。

向`map`函数提供回调函数，指定对元素的操作。回调函数有 3 个参数，第一个参数为数组元素，第二个元素为元素的索引，第三个参数为数组。回调函数的返回值为用于替换的元素。

例如：

```
var array = [1, 2, 3, 4, 5, 6];

// 将偶数转换成字符串
function fn(value, index, arr)
{
    if (value % 2 === 0)
    {
        return String(value);
    }
    return value;
}
console.log(array.map(fn)); // [1, '2', 3, '4', 5, '6']
```

14.7.10 `filter`函数

`filter`函数：过滤数组元素，满足条件的元素留下，不满足条件的元素去掉。返回过滤后的新数组。

向`filter`函数提供回调函数，指定筛选条件。回调函数有 3 个参数，第一个参数为数组元素，第二个元素为元素的索引，第三个参数为数组。回调函数的返回值为`boolean`类型，返回`true`的元素保留，返回`false`的元素去掉。

例如：

```
var array = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

// 判断质数
function isPrime(num)
{
    if (num < 2)
    {
        return false;
    }
    for (let i = 2; i < num / 2; i++)
    {
        if (num % i === 0)
        {
            return false;
        }
    }
    return true;
}

// 选择数组中的质数
var arr = array.filter(function(value, index) {
    return isPrime(value);
});
// [2, 3, 5, 7, 11]
```

14.7.11 `find`、`findIndex`函数

`find`函数：查找满足指定条件的元素，返回元素本身。

`findIndex`函数：查找满足指定条件的元素，返回元素索引。

向函数提供回调函数，指定查找条件。回调函数有 3 个参数，第一个参数为数组元素，第二个元素为元素的索引，第三个参数为数组。回调函数的返回值为`boolean`类型。

第一个使回调函数返回`true`的元素为满足条件的元素，`find`函数返回元素本身，`findIndex`函数返回元素索引。如果没有满足条件的元素，则返回`undefined`。

例如：

```
var array = [1, 2, 3, 4, 5, 6, 7, 8, 9];

// 查找是 5 的倍数的元素
array.find(function(value, index) {
    return value % 5 === 0;
});
// 返回值：5
```

14.7.12 some、every函数

some函数：判断数组中是否存在满足条件的元素。返回值为boolean类型，如果存在满足条件的元素，则返回true；如果不存在满足条件的元素，则返回false。

every函数：判断数组中的所有元素是否都满足条件。返回值为boolean类型，如果所有元素都满足条件，则返回true；如果存在不满足条件的元素，则返回false。

向函数提供回调函数，指定条件。回调函数有 3 个参数，第一个参数为数组元素，第二个元素为元素的索引，第三个参数为数组。回调函数的返回值为boolean类型。

14.7.13 includes函数

includes函数：判断数组中是否包含指定的值。第一个参数为待判断的值，第二个参数为起始索引，第二个参数可选。返回值为boolean类型，存在则返回true，不存在则返回false。

14.7.14 concat函数

concat函数：将数组元素和实参的内容进行连接，形成新的数组，返回这个新数组。参数的数量和类型不限。

例如：

```
var array = [1, 2, 3, 4];
var arr = array.concat(true, ['a', 'b']); // [1, 2, 3, 4, true, 'a', 'b']
```

14.7.15 Array.isArray静态函数

Array.isArray静态函数：判断参数是否为数组。如果是数组，则返回true；如果不是数组，则返回false。

14.8 引用传递

基本类型数据的赋值是值传递，将一个变量的值传递给另一个变量，这两个变量之间没有关系，修改其中的一个变量时，不会影响另一个变量。

引用类型数据的赋值是引用传递，将一个变量中保存的对象引用传递给另一个变量，这两个变量指向同一个对象。通过其中一个变量修改对象时，另一个变量也会受影响。

函数的参数传递也服从上述规律。基本数据类型作为实参，不会改变实参；引用数据类型作为实参，可以改变实参所指向的对象的内容，但是不能改变实参指向哪个对象。

14.9 多维数组

二维数组：数组中的元素是一维数组。

三维数组：数组中的元素是二维数组。

n维数组：数组中的元素是 n - 1 维数组。

例如：

```
// 二维数组，元素为一维数组
var arr2 = [
    [1, 2, 3],
    [4, 5, 6],
```

```
    [7, 8, 9]
];

// 访问二维数组中的元素
console.log(arr2[0]); // [1, 2, 3]
console.log(arr2[0][0]); // arr2[0] 是一个数组，这个数组中下标为 0 的元素为：1

// 遍历二维数组
for (let i = 0; i < arr2.length; i++)
{
    for (let j = 0; j < arr2[i].length; j++)
    {
        console.log(arr2[i][j]);
    }
}

// 三维数组，元素为二维数组
var arr3 = [
    [
        [1, 2],
        [3, 4, 5]
    ],
    [
        [6, 7, 8, 9]
    ],
    [
        [11, 12, 13, 14],
        [15, 16],
        [17]
    ]
];

// 访问三维数组中的元素
console.log(arr3[0][1][1]); // 4
console.log(arr3[1][0][1]); // 7

// 遍历三维数组
for (let i = 0; i < arr3.length; i++)
{
    for (let j = 0; j < arr3[i].length; j++)
    {
        for (let k = 0; k < arr3[i][j].length; k++)
        {
            console.log(arr3[i][j][k]);
        }
    }
}
```

15 对象

15.1 对象的定义

JavaScript 通过键值对描述对象。键和值之间用冒号隔开，多个键值对之间用逗号隔开。对象中的键值对是无序的。

定义对象的语法：

```
// 1.表达式方式
var 变量名 = {
    key1 : value1,
    key2 : value2,
    ...
    key_n : value_n
};

// 2.构造函数方式
var 变量名 = new Object(); // 空对象
var 变量名 = new Object(obj);

// 3.通过 Object 的静态函数创建
var 变量名 = Object.create(null); // 空对象
var 变量名 = Object.create(obj);
```

所有的键都是字符串，可以使用双引号或单引号包围键。如果键的字符串遵循标识符的命名规则，就可以省略引号；如果键的字符串不满足标识符的命名规则，就必须使用引号。例如：

```
var obj = {
    "name" : "小明", // 可以使用引号包围键
    age : 30, // 键的字符串遵循标识符的命名规则，可以省略引号
    "1a" : true // 键的字符串不满足标识符的命名规则，必须使用引号
};
```

如果键是数字，可以省略引号。例如：

```
var obj = {
    0 : "a", // 键是数字，可以省略引号
    1 : true,
    length : 2
};
```

键是唯一的。如果出现重名的键，后定义的键会覆盖前面的。

值可以是任意类型。

15.2 访问对象

访问对象的方式：

1. 如果键符合标识符的命名规则，可以用`对象名.键名`或`对象名['键名']`访问键。
2. 如果键不符合标识符的命名规则，只能用`对象名['键名']`访问键。
3. 如果键是数字，可以用`对象名[数字]`或`对象名['数字']`访问键。

例如：

```
var obj = {
    "name" : "小明",
    age : 30,
    "1a" : true,
    1 : null
};

// 键符合标识符的命名规则
var key1 = "age";
```

```
console.log(obj.name);
console.log(obj['age']);
console.log(obj[key1]);
// 键不符合标识符的命名规则
var key2 = "1a"
console.log(obj["1a"]);
console.log(obj[key2]);
// 键是数字
var key3 = "1";
console.log(obj[1]);
console.log(obj["1"]);
console.log(obj[key3]);
```

15.3 对对象的操作

添加和修改属性值：直接对属性赋值。

删除属性：`delete 对象名.属性名`。

例如：

```
// 创建空对象
var obj = {};

// 添加属性
obj.name = "tom";
obj["age"] = 12;
var key = "gender";
obj[key] = "boy";

// 修改属性
obj.age = 13;

// 删除属性
delete obj.gender;
```

15.4 `this`关键字

`this`关键字在对象中使用，是当前对象的引用，代表了当前对象。

当`this`关键字在函数中使用时，`this`代表调用该函数的对象。例如：

```
var obj = {
    name: "tom",
    age : 12,
    show : function() {
        // this 代表当前对象自身
        console.log("name=" + name + ",age=" + "age");
    }
};

obj.show(); // "name=tom,age=12"

var num = 100;
function test()
{
```

```
    var num = 10;
    console.log(this.num); // test 函数的调用者是 window 对象，this 代表 window 对象
}
test(); // 100
```

15.5 对象的遍历

```
var obj = {
    name : "tom",
    age : 17,
    gender : "man"
    study : function() {
        console.log("I love study.");
    }
}

// 1. for in 循环
for (var key in obj) // 变量 key 保存键名
{
    console.log(key,obj[key]);
}

// 2. Object 的静态函数 keys(obj)
// 返回一个数组，数组元素为对象的所有的键
var keys = Object.keys(obj);
for (let i = 0; i < keys.length; i++)
{
    console.log(keys[i],obj[keys[i]]);
}

// 3. Object 的静态函数 values(obj)
// 返回一个数组，数组元素为对象的所有的值
var values = Object.values(obj);
for (let i = 0; i < values.length; i++)
{
    console.log(values[i]);
}

// 4. Object 的静态函数 entries(obj)
// 得到一个二维数组，每个元素数组是一个键值对
var entries = Object.entries(obj);
for (let i = 0; i < entries.length; i++)
{
    console.log("key:" + entries[i][0] + ",value:" + entries[i][1]);
}
```

15.6 JavaScript的内置对象

15.6.1 `Object`对象

`Object`是 JavaScript 中的一个构造函数，使用`new`关键字调用，用来创建对象。

`Object()`可以作为工具方法使用，将任意类型的数据转换为对象。例如：

```
console.log(Object(null)); // 空对象
console.log(Object(undefined)); // 空对象
console.log(Object(1)); // Number {1}
console.log(Object("a")); // String {"a"}
console.log(Object(true)); // Boolean {true}
```

`Object`对象的静态函数：

1. `Object.keys(obj)`：得到对象所有的键，以数组形式返回。
2. `Object.values(obj)`：得到对象所有的值，以数组形式返回。
3. `Object.entries(obj)`：得到对象所有的键值对，以二维数组形式返回。
4. `Object.create(obj)`：创建对象。
5. `Object.getOwnPropertyName(obj)`：得到对象所有的键，包括不可枚举的属性，以数组形式返回。

`Object`对象的实例函数：

1. `toString(obj)`：得到当前对象的字符串表示形式，返回值为`"[object Object]"`。
2. `toLocalString(obj)`：得到当前对象的具有本地特性的字符串表示形式。
3. `hasOwnProperty(key)`函数：判断参数是否为当前对象的私有属性，是则返回`true`，否则返回`false`。
4. `valueOf(obj)`：返回当前对象的原始值。主要用于从`Number`、`String`、`Boolean`对象中获取原始值。

JavaScript 是一种基于原型的语言，任何对象都是在一个原型对象的基础上创建的，这个原型对象提供了所有对象可能需要使用的函数。这个原型对象是`Object`构造函数的原型，并依赖于`Object`对象。JavaScript 中的所有对象都是以`Object`构造函数的原型为基础创建的。`Object`是 JavaScript 中任何类型的父类型，JavaScript 中任何类型的数据都以`Object`为父类型。

`instanceof`关键字用于判断一个对象是否是某种类型的实例，是则返回`true`，否则返回`false`。语法：`对象 instanceof 类型`。例如：

```
console.log([] instanceof Array); // true
console.log(function(){} instanceof Function); // true
console.log({} instanceof Object); // true

console.log([] instanceof Object); // true
console.log(function(){} instanceof Object); // true
```

15.6.2 `Number`对象

`Number`对象是`number`类型的包装类型对象，通过`new`关键字调用`Number`构造函数可以将`number`类型数据包装成对象，这个过程叫做装箱。例如：

```
console.log(new Number(1)); // Number {1}
```

如果对一个`number`类型值调用实例函数，底层会进行自动装箱。例如：

```
(1).toString() // 自动装箱
```

作为工具类使用，可以将任意类型的数据强制转换为`number`类型。

`Number`对象中的常量：

1. `Number.MAX_VALUE`：最大值，取值为`1.7976931348623157e+308`
2. `Number.MIN_VALUE`：最小值，取值为`5e-324`
3. `Number.NEGATIVE_INFINITY`：`-Infinity`
4. `Number.POSITIVE_INFINITY`：`Infinity`
5. `Number.MIN_SAFE_INTEGER`：最小整数，取值为`-9007199254740991`
6. `Number.MAX_SAFE_INTEGER`：最大整数，取值为`9007199254740991`
7. `Number.NaN`：`NaN`

`Number`对象的静态函数：

1. `Number.isNaN(x)`：判断参数是否为`NaN`，是`NaN`则返回`true`，否则返回`false`。
2. `Number.parseInt(x)`：将字符串转换成数字。
3. `Number.parseFloat(x)`：将字符串转换成浮点数。
4. `Number.isFinite(x)`：判断实参是否是有穷数。是有穷数则返回`true`，是无穷数则返回`false`。

`Number`对象的实例函数：

1. `valueOf()`：拆箱，将对象转换成`number`类型。
2. `toString(redix)`：将当前对象转换为指定进制的字符串表示形式。
3. `toFixed(digit)`：小数点后保留指定的位数，四舍五入。

15.6.3 `Boolean`对象

作为构造函数使用，可以将布尔类型数据装箱为对象。例如：

```
console.log(new Boolean(true)); // Boolean {true}
console.log(new Boolean(1)); // Boolean {true}
```

作为工具函数使用，可以将任意类型值转换为布尔类型。

15.6.4 `Math`对象

`Math`不能作为构造函数使用，不能创建`Math`类型对象。

`Math`对象中的成员都是静态成员，不需要通过实例对象调用。

`Math`对象中的常量：

1. `Math.E`：自然对数的底数 e，约等于 2.71828。
2. `Math.LN2`：$\ln 2$，约等于 0.693。
3. `Math.LN10`：$\ln 10$，约等于 2.302。
4. `Math.LOG2E`：$\log_2 e$，约等于 1.414。
5. `LOG10E`：$\log_{10} e$，约等于 0.434。
6. `Math.PI`：圆周率π，约等于 3.1415926。
7. `Math.SQRT1_2`，$\frac{1}{\sqrt{2}}$，约等于 0.707。
8. `Math.SQRT2`：$\sqrt{2}$，约等于 1.414。

`Math`对象的静态方法：

1. `Math.abs(x)`：求绝对值。
2. `Math.acos(x)`：反余弦函数。
3. `Math.asin(x)`：反正弦函数。
4. `Math.atan(x)`：反正切函数，取值范围为$[-\frac{\pi}{2}, \frac{\pi}{2}]$。
5. `Math.atan2(y,x)`：返回从 x 轴到点 (x, y) 的角度，取值范围为$[-\frac{\pi}{2}, \frac{\pi}{2}]$。

6. `Math.ceil(x)`：向上取整。
7. `Math.floor(x)`：向下取整。
8. `Math.round(x)`：把数四舍五入为最接近的整数。
9. `Math.cos(x)`：余弦函数。
10. `Math.sin(x)`：正弦函数。
11. `Math.tan(x)`：正切函数。
12. `Math.exp(x)`：返回 e^x。
13. `Math.log(x)`：返回 $\ln x$。
14. `Math.max(x,y)`：返回 x 和 y 中的最大值。
15. `Math.min(x,y)`：返回 x 和 y 中的最小值。
16. `Math.pow(x,y)`：返回 x^y。
17. `Math.random()`：返回 $[0,1)$ 区间上的随机数。
18. `Math.sqrt(x)`：返回 $\sqrt{x}$。
19. `Math.toSource()`：返回该对象的源代码。
20. `Math.valueOf()`：返回`Math`对象的原始值。

随机数的用法：

```
const MIN = ...;
const MAX = ...;

// [0,1) 范围内的随机数
var ran1 = Math.random();
// [MIN,MAX) 范围内的随机数
var ran2 = Math.random() * (MAX - MIN) + MIN;
// [MIN,MAX+1) 范围内的随机数
var ran3 = Math.random() * (MAX - MIN + 1) + MIN;
// [MIN,MAX] 范围内的整数随机数
var ran4 = ~~(Math.random() * (MAX - MIN + 1) + MIN);
// [MIN,MAX) 范围内的整数随机数
var ran5 = ~~(Math.random() * (MAX - MIN) + MIN);
```

15.6.5 `Date`对象

`Date`是 JavaScript 的内置时间对象。

JavaScript 的时间原点：标准世界时（1970-01-01 00:00:00）。

`Date()`作为工具函数使用，以字符串形式返回当前系统时间。例如：

```
console.log(Date()); // 'Fri Sep 17 2021 10:11:55 GMT+0800 (中国标准时间)'
```

`Date()`作为构造函数使用，创建日期对象。没有参数时，用系统时间创建日期对象。例如：

```
var date = new Date();
console.log(typeof date); // object
console.log(date); // Fri Sep 17 2021 10:11:55 GMT+0800 (中国标准时间)
console.log(Number(date)); // 当前时间与时间原点的时间差，单位为毫秒
```

利用`Date()`构造函数可以用于计算代码块的执行时间。例如：

```
var date1 = new Date();
// 代码块
var date2 = new Date();
var time = date2 - date1; // 计算时间差
```

`Date()` 构造函数有参数时，用指定的时间创建日期对象。例如：

```
var date1 = new Date(2021, 0, 1, 0, 0, 0); // 2021-01-01 00:00:00
var date2 = new Date("2021-1-1"); // 2021-01-01 00:00:00
var date3 = new Date("2021/1/1 12:12:12"); // 2021-01-01 12:12:12
```

`Date` 对象的静态函数：

1. `Date.now()`：返回当前时间与时间原点的时间差，单位为毫秒。
2. `Date.parse(str)`：解析一个时间字符串，返回指定时间与时间原点的时间差，单位为毫秒。

`Date` 对象的实例函数：

1. `toString()`：将日期对象转换为字符串，默认为西方风格。
2. `toLocalString()`：将日期对象转换为本地风格的字符串。
3. `toTimeString()`：将日期对象的时分秒部分转换为字符串。
4. `toLocalTimeString()`：将日期对象的时分秒部分转换为本地风格的字符串。
5. `toDateString()`：将日期对象的年月日部分转换为字符串。
6. `toLocalDateString()`：将日期对象的年月日部分转换为本地风格的字符串。
7. `toUTCString()`：将日期对象转换为 UTC 标准的字符串形式。
8. `getFullYear()`：返回日期对象的年数。
9. `getMonth()`：返回日期对象的月数，从 0 开始。
10. `getDate()`：返回日期对象的日期书。
11. `getDay()`：返回日期对象的星期数。0 为星期日。
12. `getHours()`：返回日期对象的小时数。
13. `getMinutes()`：返回日期对象的分钟数。
14. `getSeconds()`：返回日期对象的秒数。
15. `getMilliseconds()`：返回日期对象的毫秒数。
16. `setFullYear()`：设置日期对象的年数。
17. `setMonth()`：设置日期对象的月数。
18. `setDate()`：设置日期对象的日期数。
19. `setHours()`：设置日期对象的小时数。
20. `setMinutes()`：设置日期对象的分钟数。
21. `setSeconds()`：设置日期对象的秒数。
22. `setMilliseconds()`：设置日期对象的毫秒数。
23. `getTime()`：返回日期对象距离时间原点的毫秒数。
24. `getTimezoneOffset()`：时区偏差，距离标准世界时的时差，单位为分钟。

15.6.6 `String` 对象

`String` 对象是 `string` 类型的包装对象。当 `string` 类型值调用实例函数时，会进行自动装箱。例如：

```
var str = "abc";
str.charAt(); // str 自动装箱
```

`String()` 作为工具函数使用，可以将任意类型值转换为 `string` 类型。

`String` 作为构造函数使用，可以将 `string` 类型值转换为对象。例如：

```
var str = new String("abc");
```

任何一个字符串常量都可以看做一个 `String` 对象。

字符串是由若干个字符序列组成的，底层使用字符数组管理这些字符序列，所以字符串具有一些数组的特点。例如：

```
// 使用下标访问字符
var str = "abc";
var key = 2;
console.log(str[0]); // a
console.log(str["1"]); // b
console.log(str[key]); // c
```

`String` 对象一旦创建就不能修改。因此通过下标只能获取字符，不能修改字符。例如：

```
// 1.通过下标只能获取字符，不能修改字符
var str = "abc";
str[0] = "A";
console.log(str); // abc

// 2.字符串的拼接并不修改字符串，而是创建新的字符串
var str1 = "abc";
var str2 = str1; // str1 与 str2 指向同一个对象
str2 += "d"; // str2 指向新建的字符串
console.log(str1); // abc
console.log(str2); // abcd
```

`String` 对象的静态函数：

1. `String.fromCharCode(codes)`：接收多个 `number` 类型值，将每个数字转换为字符，返回这些字符组成的字符串。

`String` 对象的实例属性：

1. `length`：表示字符串中的字符个数，是一个大于等于 0 的整数。该属性是只读属性，不能修改。在严格模式下，修改 `length` 属性会报错。

`String` 对象的实例函数：

1. `big()`：当该字符串显示在页面上时，用大号字体显示。
2. `blink()`：当该字符串显示在页面上时，产生闪动效果。
3. `bold()`：当该字符串显示在页面上时，用粗体显示。
4. `italics()`：当该字符串显示在页面上时，用斜体显示。
5. `charAt(num)`：返回指定下标的字符。
6. `charCodeAt(num)`：返回指定下标的字符的编码。
7. `concat(str)`：连接字符串，返回连接后的字符串。
8. `indexOf(str,num)`：从指定位置开始查找给定的字符串。第二个参数可以省略。
9. `lastIndexOf(str,num)`：从指定位置开始从后向前查找给定的字符串。第二个参数可以省略。
10. `localeCompare(str)`：用本地特定的顺序比较两个字符串。如果当前字符串大于参数，则返回 1；如果当前字符串小于参数，则返回 -1；如果二者相等，则返回 0。
11. `match(reg)`：从字符串中提取出符合正则表达式的内容。默认情况下只返回第一个符合要求的内容。如果要返回所有符合要求的内容，可以设置正则表达式为全局匹配模式，将匹配到的内容封装

到数组中。

12. `replace(str,newSubstring)`：替换与给定字符串匹配的子串，第一个参数可以是正则表达式。默认只替换第一个匹配的子串。如果要替换所有匹配的子串，可以设置正则表达式为全局匹配模式。
13. `search(str)`：搜索字符串中是否含有指定内容。如果搜索到指定内容，则返回第一次出现的索引；如果没有搜索到，则返回 -1。可以接收正则表达式作为参数。
14. `slice(start,end)`：提取字符串的片段，返回被提取的部分。
15. `substr(from,length)`：从指定位置开始，提取指定数目的字符。
16. `substring(start,end)`：提取两个指定的索引号之间的字符。不支持负数作为参数。
17. `split()`：把字符串分割为数组。参数为分隔符，如果参数为空，则每个字符作为一个数组元素。
18. `toLowerCase()`：将所有字母转换为小写。
19. `toUpperCase()`：将所有字母转换为大写。
20. `valueOf()`：返回字符串对象的原始值。

字符串中字符的遍历：

```
var str = "abc";

// 1. for 循环
for (let i = 0; i < str.length; i++)
{
    console.log(str[i]);
}

// 2. for of 循环
for (var ch of str)
{
    console.log(ch);
}

// 3. for in 循环
for (var i in str)
{
    console.log(str[i]);
}

// 4. for 循环中使用 charAt 函数
for (let i = 0; i < str.length; i++)
{
    console.log(str.charAt(i));
}
```

15.6.7 `Global` 对象

JavaScript 中任何一个函数必须属于某个对象。JavaScript 中有一些内置的全局函数，它们归属于 `Global` 对象。

`Global` 对象的函数（全局函数）：

1. `isNaN(num)`：判断参数是否是 `NaN`，是 `NaN` 则返回 `true`，否则返回 `false`。
2. `isFinite(num)`：判断参数是否是有穷数，是有穷数则返回 `true`，否则返回 `false`。
3. `parseInt(str)`：将字符串转换为整数。
4. `parseFloat(str)`：将字符串转换为浮点数。

编码函数：

1. `encodeURI(str)`：对字符串进行编码。只对中文、空格等字符进行编码，不会对常见字符进行编码。
2. `encodeURIComponent(str)`：对字符串进行编码。只对数字、字母不进行编码，其他字符都会编码。

解码函数：

1. `decondeURI(uri)`：对 `encodeURI` 函数编码后的字符串进行解码。
2. `decodeURIComponent(uri)`：对 `encodeURIComponent` 函数编码后的字符串进行解码。

`Global` 对象的 `eval(str)` 函数将字符串转换为 JavaScript 代码执行，返回执行的结果。例如：

```
// 1.执行 JavaScript 代码
eval("console.log(1 + 1)"); // 2

// 2. eval 函数内部定义的变量和函数，可以在外部访问
eval("var value = 10");
console.log(value); // 10

// 3. 外部定义的变量和函数，可以在 eval 函数内部访问
function test()
{
    console.log("hello world");
}
eval("test()"); // hello world
```

如果字符串中含有 `{}`，它会将 `{}` 当做代码块。如果不希望把 `{}` 当成代码块解析，需要在 `{}` 前后加上小括号 `()`。

15.7 类数组对象

如果一个对象的所有键都是数值类型、从 0 开始依次递增，还有 `length` 属性表示元素个数，称这样的对象为**类数组对象**。

类数组对象的 `length` 需要程序员自己维护，而数组的 `length` 属性是数组底层自动维护的。

15.8 正则表达式

15.8.1 正则表达式简介

正则表达式用于定义一些字符串的规则。计算机可以根据正则表达式来检查一个字符串是否满足规则，或者将字符串中符合规则的内容提取出来。

15.8.2 创建正则表达式对象

要使用正则表达式，首先要创建正则表达式对象。使用 `RegExp()` 构造函数创建正则表达式对象。它有 2 个参数，第一个参数表示正则表达式，第二个参数表示匹配模式，两个参数都是字符串。例如：

```
var reg = new RegExp("a");
console.log(reg); // "/a/"
console.log(typeof reg); // object
```

`RegExp()` 构造函数的第二个参数表示匹配模式，可以省略。可能的取值为：

1. `i`：忽略大小写。
2. `g`：全局匹配模式。

创建正则表达式对象的简写语法：`/正则表达式/匹配模式`。例如：

```
var reg = /a/i;
console.log(typeof reg); // object
```

正则表达式对象的`test(str)`方法用于检查一个字符串是否符合正则表达式的规则，如果符合则返回`true`，否则返回`false`。例如：

```
var reg = new RegExp("a"); // 检查字符串中是否含有 a
console.log(reg.test("a")); // true
console.log(reg.test("bcbc")); // false
```

15.8.3 正则表达式语法

如果正则表达式中只含有字符序列，表示查找字符串中是否包含指定的字符序列。例如：

```
var reg = /ab/; // 检测字符串中是否包含 "ab"
console.log(reg.test(abc)); // true
console.log(reg.test(bac)); // false
```

`|`表示“或者”关系。例如：

```
var reg = /a|b/; // 检测字符串中是否包含 a 或 b
console.log(reg.test("ac")); // true
console.log(reg.test("bc")); // true
console.log(reg.test("c")); // false
```

`[]`表示查找方括号之间的任何字符。例如：

```
var reg = /[ab]/; // []中的内容是“或者”关系，查找给定集合内的任何字符
reg = /[a-z]/; // 查找任意小写字母
reg = /[A-Z]/; // 查找任意大写字母
reg = /[A-z]/; // 查找任意字母
reg = /a[bde]c/; // 查找 "abc" 或 "adc" 或 "aec"
reg = /[0-9]/; // 查找任意数字
```

`[^]`表示查找给定集合之外的字符。例如：

```
var reg = /[^ab]/; // 查找 a 和 b 之外的字符
console.log(reg.test("a")); // false
console.log(reg.test("abc")); // true
```

量词：通过量词可以设置一个内容出现的次数。常用的量词有：

1. `(内容){n}`：指定的内容连续出现 n 次。
2. `(内容){m,n}`：指定的内容连续出现 m 到 n 次。
3. `(内容){n,}`：指定的内容至少连续出现 n 次。
4. `(内容)+`：指定的内容至少出现一次。
5. `(内容)*`：指定的内容出现 0 次或多次。
6. `(内容)?`：指定的内容出现 0 次或 1 次。
7. `^(内容)`：指定的内容作为开头。
8. `(内容)$`：指定的内容作为结尾。

9. `^(内容)$`：指定的内容既是开头也是结尾，整个字符串必须完全符合正则表达式。

元字符是拥有特殊含义的字符。常用的元字符有：

1. `.`：查找单个字符，除了换行和行结束符。
2. `\.`：查找点号 `.`。
3. `\\`：查找反斜杠 `\`。
4. `\w`：查找任意字母、数字、下划线。
5. `\W`：查找除了字母、数字、下划线的字符。
6. `\d`：查找数字。
7. `\D`：查找非数字字符。
8. `\s`：查找空白字符。
9. `\S`：查找非空白字符。
10. `\b`：表示单词边界。
11. `\B`：表示非单词边界。
12. `\0`：查找 NULL 字符。
13. `\n`：查找换行符。
14. `\f`：查找换页符。
15. `\r`：查找回车符。
16. `\t`：查找制表符。
17. `\v`：查找垂直制表符。
18. `\xxx`：查找以八进制数 xxx 规定的字符。
19. `\xdd`：查找以十六进制数 dd 规定的字符。
20. `\uxxxx`：查找以十六进制数 xxxx 规定的 Unicode 字符。

常用的正则表达式：

```
// 1.检查字符串是否为合法的手机号
// 手机号的规则：
//（1）第一位是 1
//（2）第二位是 3-9 的任意数字
//（3）从第三位开始，各个位是任意数字，共 9 个
var phoneReg = /^1[3-9][0-9]{9}$/;

// 2.去除字符串首尾的空格
var str = ...;
str = str.replace(/^\s*|\s*$/g, "");

// 3.检查字符串是否为合法的电子邮件
// 电子邮件的规则：
// 任意字母、数字、下划线 + .任意字母、数字、下划线（可选） + @ + 任意字母、数字 + .任意字母（2-5位） + .任意字母（2-5位，可选）
var emailReg = /^\w{3,}(\.\w+)*@[A-zO-9]+(\.[A-z]{2,5}){1,2}$/;
```

15.8.4 支持正则表达式的`String`对象的方法

1. `search(str)`：搜索字符串中是否含有指定内容。如果搜索到指定内容，则返回第一次出现的索引；如果没有搜索到，则返回 -1。可以接收正则表达式作为参数。
2. `match(reg)`：从字符串中提取出符合正则表达式的内容。默认情况下只返回第一个符合要求的内容。如果要返回所有符合要求的内容，可以设置正则表达式为全局匹配模式，将匹配到的内容封装到数组中。
3. `replace(str,newSubstring)`：替换与给定字符串匹配的子串，第一个参数可以是正则表达式。默认只替换第一个匹配的子串。如果要替换所有匹配的子串，可以设置正则表达式为全局匹配模

式。
4. `split(str)`：把字符串分割为字符串数组。

例如：

```
// 1. search
var str1 = "hello abc hello aec afc";
console.log(str1.search("abc")); // 6
console.log(str1.search("abcd")); // -1
result1 = str1.search(/a[bef]c/); // 搜索字符串中是否含有 abc 或 aec 或 afc
console.log(result1); // 6

// 2. match
var str2 = "1a2b3c4d5e6f7";
var result2 = str2.match(/[A-z]/); // 提取任意字母
console.log(result2); // "a"
result2 = str2.match(/[A-z]/g); // 全局匹配模式
console.log(result2); // [a,b,c,d,e,f]

// 3. replace
var str3 = "1a2a3a4a";
var result3 = str3.replace("a", "@_@"); // 只替换第一个 a
console.log(result3); // "1@_@2a3a4a"
result3 = str3.replace(/a/g, "@_@"); // 将所有 a 替换成 "@_@"
console.log(result3); // "1@_@2@_@3@_@4@_@"
result3 = str3.replace(/[a-z]/ig, ""); // 将所有字母替换为空字符串，即去掉所有字母
console.log(result3); //"1234"

// 4. split
var str4 = "1a2b3c4d5e6f7";
var result4 = str4.split(/[A-z]/); // 以任意字母为分隔符
console.log(result4); // [1,2,3,4,5,6,7]
```

16 DOM

16.1 DOM介绍

DOM 的全称为**文档对象模型**（Document Object Model）。整个 HTML 文档和文档中的节点可以看做对象，通过 JavaScript 可以访问这些对象。

在 JavaScript 中，浏览器对象`window`提供了`document`对象来代表整个文档，通过`document`对象可以访问页面中的所有元素。`document`对象是 DOM 的核心。

节点是构成网页的最基本的组成部分，网页中的每一个部分都可以看做一个节点。例如，`html`标签、属性、文本、注释、整个文档都是节点。

文档中节点的类型：

1. 文档节点：`document`对象。
2. 元素节点：HTML 标签。
3. 属性节点：HTML 标签的属性。
4. 文本节点：HTML 文档中的文本。
5. 注释节点：HTML 注释。

在 DOM 中，任何节点对象都有下列 3 个属性：

1. `nodeName`：节点名称。

2. `nodeType`：节点类型。
3. `nodeValue`：节点值。

不同类型的节点，其属性值也不同，如下表所示。

节点类型	`nodeName`	`nodeType`	`nodeValue`
文档节点	`#document`	9	`null`
元素节点	标签名	1	`null`
属性节点	属性名	2	属性值
文本节点	`#text`	3	文本内容

16.2 元素节点

获得元素节点的方式：

1. `document.getElementById(id)`：在整个文档中搜索指定`id`属性值的元素对象，如果存在则返回元素对象，否则返回`null`。
2. `getElementsByTagName(tag)`：在指定的范围内搜索指定标签的元素对象，返回类数组对象`HTMLCollection`，通过下标访问元素对象。参数为`*`时返回文档中所有标签的元素对象。
3. `document.getElementsByName(name)`：在整个文档中搜索指定`name`属性值的元素对象，返回类数组对象`NodeList`，通过下标访问元素对象。
4. `getElementsByClassName(class)`：在指定的范围内搜索指定`class`属性值的元素对象，返回类数组对象`HTMLCollection`，通过下标访问元素对象。不支持 IE 8 及以下版本。
5. `querySelector(selector)`：在指定的范围内用选择器选择元素，返回第一个满足条件的元素对象。
6. `querySelectorAll(selector)`：在指定的范围内用选择器选择元素，返回所有满足条件的元素对象。返回类数组对象`NodeList`，通过下标访问元素对象。

例如：

```
<div id="box">hello</div>
<div>world</div>

<input type="checkbox" name="hobby">唱
<input type="checkbox" name="hobby">跳
<input type="checkbox" name="hobby">rap
<input type="checkbox" name="hobby">篮球

<ul>
    <li>1</li>
    <li class="sty">2</li>
    <li>3</li>
    <li class="sty">4</li>
    <li>5</li>
</ul>

<script>
    // 1. getElementById
    var box = document.getElementById("box");
    console.log(box); // <div id="box">hello</div>

```

```
    // 2. getElementsByTagName
    var divs = document.getElementsByTagName("div"); // 在整个文档中搜索 div 标签
    console.log(divs); // HTMLCollection(2) [div#box, div, box: div#box]
    console.log(divs[0]); // <div id="box">hello</div>
    var div = document.body.getElementsByTagName("div")[1]; // 在 body 中搜索 div 标签
    console.log(div); // <div>world</div>

    // 3. getElementsByName
    var hobbies = document.getElementsByName("hobby");
    console.log(hobbies); // NodeList(4) [input, input, input, input]
    console.log(hobbies[1]); // <input type="checkbox" name="hobby">

    // 4. getElementsByClassName
    var ul = document.getElementsByTagName("ul")[0];
    var lis = ul.getElementsByClassName("sty"); // 在 ul 中搜索 class 属性值为 "sty" 的元素
    console.log(lis); // HTMLCollection(2) [li.sty, li.sty]

    // 5. querySelector
    var element = document.querySelector(".sty"); // 通过类选择器选择元素
    console.log(element); // <li class="sty">2</li>

    // 6. querySelectorAll
    var list = document.querySelectorAll(".sty");
    console.log(list); // NodeList(2) [li.sty, li.sty]
</script>
```

`HTMLCollection`与`NodeList`的区别：`HTMLCollection`会根据内容的改变而动态更新集合中的内容；而`NodeList`是静态的，不会动态更新。

操作元素节点的方式：

1. `innerHTML`：访问元素节点的内容，包含节点中的所有子元素及其内容。修改时会覆盖所有子元素。设置的内容中如果包含标签，标签会生效。
2. `innerText`：访问元素节点的文本内容，不包含节点中的子元素。设置的内容中如果包含标签，标签会显示为普通的文本。
3. `textContent`：用法和作用与`innerText`相同，但兼容性更好。
4. `value`：访问元素的`value`属性。

例如：

```
<div>
    content
    <span>内容</span>
</div>

<script>
    var div = document.querySelector("div");

    // 获得 innerHTML
    console.log(div.innerHTML); /*
                                 * content
                                 * <span>内容</span>
                                 */
    console.log(typeof div.innerHTML); // string
```

```
    // 修改 innerHTML
    div.innerHTML = "hello world"; // 覆盖原有内容
    div.innerHTML += "<span>你好</span>"; // 在原有内容的基础上追加内容
    console.log(div.innerHTML); /*
                                 * hello world
                                 * <span>你好</span>
                                 */

    // 获得 innerText
    console.log(div.innerText); // hello world
</script>
```

16.3 属性节点

通过元素节点获取属性节点的方式：

1. `元素对象.attributes`：返回类数组对象`NamedNodeMap`，其中包含元素对象的所有属性。可以通过数字索引或属性名访问属性对象。
2. `元素对象.getAttributeNode(attr)`：返回指定属性名的属性对象。

例如：

```
<div id="box" title="div1">content</div>

<script>
    var div = document.querySelector("div");

    // 1. attributes 属性
    var attributes = div.attributes;
    console.log(attributes); // NamedNodeMap{0:id,
1:title,id:id,title:title,length:2}
    console.log(attributes.id); // id="box"
    console.log(attributes.title); // title="div1"

    // 2. getAttributeNode(attr) 函数
    var node = div.getAttributeNode("id");
    console.log(node); // id="box"
</script>
```

获取和设置属性值的方式：

1. `元素对象.getAttribute(attr)`：返回指定属性的属性值。如果属性不存在，则返回`null`。
2. `元素对象.setAttribute(attr,value)`：为指定属性设置属性值。
3. `元素对象.setAttributeNode(node)`：参数为属性节点对象，为元素对象设置相应属性。
4. `属性对象.nodeValue`属性：返回属性值。
5. `元素对象.属性名`或`元素对象["属性名"]`：返回属性值。不支持自定义属性。

例如：

```
<div id="box" title="div1">content</div>

<script>
    var div = document.querySelector("div");

    // 1. getAttribute(attr)
```

```
    console.log(div.getAttribute("id")); // box

    // 2. setAttribute(attr,value)
    div.setAttribute("id", "div1"); // 已设置 id 属性值，修改其属性值
    div.setAttribute("name", "tom"); // 未设置 name 属性值，设置其属性值

    // 3. 元素对象.setAttributeNode(node)
    var node = div.getAttributeNode("title"); // 获得属性节点对象
    div.setAttributeNode(node); // 设置 title 属性

    // 4. 属性对象.nodeValue
    console.log(node.nodeValue); // div1
    node.nodeValue = "title"; // 修改属性值
    console.log(node.nodeValue); // title

    // 5. 元素对象.属性名
    console.log(div.id); // div1
    div["id"] = "container"; // 修改属性值
</script>
```

特殊属性的获取和设置：

1. `class` 属性：如果使用 `元素对象.属性名` 或 `元素对象["属性名"]` 访问 `class` 属性，属性名为 `className`。
2. `checked` 属性：返回值为 `boolean` 类型。如果是选中状态，则返回 `true`，否则返回 `false`。设置 `checked` 属性值时也要使用 `boolean` 类型值。
3. `selected` 属性：同 `checked` 属性。

例如：

```
<div class="box">content</div>

<input type="radio" name="gender" checked>男
<input type="radio" name="gender">女

<script>
    var div = document.querySelector("div");
    var gender = document.getElementsByName("gender");

    // 1. class 属性
    console.log(div.className); // box
    div.className = "wrapper"; // 修改 class 属性值

    // 2. checked 属性
    console.log(gender[0].checked); // true
    console.log(gender[1].checked); // false
    gender[1].checked = true; // 修改 checked 属性值
</script>
```

添加属性节点的方式：`document.createAttribute(attr)`。

删除属性节点的方式：`元素对象.removeAttribute(attr)`。

16.4 节点对象之间的关系

1. 当前节点的所有子节点：`节点对象.childNodes`，返回类数组对象`NodeList`，通过下标访问子节点对象。
2. 当前节点的第一个子节点：`节点对象.firstChild`。
3. 当前节点的最后一个子节点：`节点对象.lastChild`。
4. 当前节点的父节点：`节点对象.parentNode`。
5. 当前节点的下一个兄弟节点：`节点对象.nextSibling`。
6. 当前节点的上一个兄弟节点：`节点对象.previousSibling`。

例如：

```
<ul>
    <li>1</li>
    <li>2</li>
    <!-- comment -->
    <li id="li3">3</li>
    <li>4</li>
    <li>5</li>
</ul>

<script>
    var parent = document.querySelector("ul");

    // 1. childNodes
    console.log(parent.childNodes); // NodeList(13)，包括换行符和注释
                                    // 在 IE 8 及以下版本中，不会将换行符当成文本节点

    // 2. firstChild
    console.log(parent.firstChild); // 换行符

    // 3. lastChild
    console.log(parent.lastChild); // 换行符

    // 4. parentNode
    var li3 = document.getElementById("li3");
    console.log(li3.parentNode); // <ul>...</ul>

    // 5. nextSibling
    console.log(li3.nextSibling.nextSibling); // <li>4</li>

    // 6. previousSibling
    console.log(parent.lastChild.previousSibling); // <li>5</li>
</script>
```

16.5 元素节点之间的关系

1. 当前元素的所有子元素：`元素对象.children`，返回类数组对象`HTMLConnection`，通过下标访问子元素对象。
2. 当前元素的第一个子元素：`元素对象.firstElementChild`。不支持 IE 8 及以下版本。
3. 当前元素的最后一个子元素：`元素对象.lastElementChild`。
4. 当前元素的父元素：`元素对象.parentElement`。
5. 当前元素的下一个兄弟元素：`元素对象.nextElementSibling`。
6. 当前元素的上一个兄弟元素：`元素对象.previousElementSibling`。不支持 IE 8 及以下版本。

例如：

```html
<ul>
    <li>1</li>
    <li>2</li>
    <!-- comment -->
    <li id="li3">3</li>
    <li>4</li>
    <li>5</li>
</ul>

<script>
    var parent = document.querySelector("ul");

    // 1. children
    console.log(parent.children); // HTMLCollection(5)，只包括 li 标签

    // 2. firstElementChild
    console.log(parent.firstElementChild); // <li>1</li>

    // 3. lastElementChild
    console.log(parent.lastElementChild); // <li>5</li>

    // 4. parentElement
    var li3 = document.getElementById("li3");
    console.log(li3.parentElement); // <ul>...</ul>

    // 5. nextElementSibling
    console.log(li3.nextElementSibling); // <li>4</li>

    // 6. previousElementSibling
    console.log(li3.previousElementSibling); // <li>2</li>
</script>
```

16.6 DOM的增、删、改操作

1. `document.createElement(tagName)`：以标签名为参数，根据标签名创建元素节点对象并返回。
2. `document.createTextNode(text)`：创建文本节点对象并返回该对象，通过参数指定文本内容。
3. `父节点.appendChild(child)`：向节点对象中添加子节点。
4. `父节点.removeChild(child)`：从节点对象中删除子节点。
5. `父节点.replaceChild(newChild, oldChild)`：替换子节点。
6. `父节点.insertBefore(newChild, refChild)`：在指定的子节点对象前面插入新的子节点。

16.7 `document`对象的属性

1. `documentElement`：`html`标签元素对象。
2. `body`：`body`节点对象。
3. `head`：`head`节点对象。
4. `title`：`title`节点对象的内容。
5. `doctype`：文档类型说明节点对象。
6. `readyState`：文档的加载状态。取值有 3 种：`loading`表示正在加载文档，`interactive`表示正在加载外部资源文件，`complete`表示加载完毕。
7. `documentURI`：地址栏中的地址。
8. `domain`：域名。
9. `location`：地址栏对象。
   地址栏对象的属性：

(1) `location.href`：地址栏中的地址。
(2) `location.protocol`：协议。
(3) `location.host`：主机号。
(4) `location.hostname`：主机名。
(5) `location.port`：端口号。
(6) `location.pathname`：文件路径。
(7) `location.search`：请求参数。

10. `links`：文档中所有 `a` 标签的集合，返回值为类数组对象 `HTMLCollection`。
11. `images`：文档中所有 `img` 标签的集合，返回值为类数组对象 `HTMLCollection`。
12. `forms`：文档中所有 `form` 标签的集合，返回值为类数组对象 `HTMLCollection`。
13. `scripts`：文档中所有 `script` 标签的集合，返回值为类数组对象 `HTMLCollection`。
14. `all`：文档中所有标签元素的集合，返回值为类数组对象 `HTMLAllCollectiono`。

16.8 操作样式

16.8.1 通过 `style` 属性获得和修改行内样式

1. `元素对象.style`：访问元素的 `style` 属性。
2. `元素对象.style.样式属性`：访问特定的样式属性，样式属性的值为字符串。样式属性名使用驼峰命名法。
3. `元素对象.style.cssText`：获得 `style` 属性中的所有样式属性。
4. `元素对象.style.setProperty(key,value)`：为特定样式属性设置样式属性值。样式属性使用中划线的命名方式。
5. `元素对象.style.getProperty(key)`：获得特定样式属性的样式属性值。
6. `元素对象.style.removeProperty(key)`：删除特定样式属性。

例如：

```
<div></div>

<script>
    // 获取元素对象
    var div = document.querySelector("div");

    // 通过 style.样式属性 设置行内样式
    div.style.width = "200px";
    div.style.height = "200px";
    div.style.backgroundColor = "red";
    div.style.border = "1px solid";
    div.style.fontSize = "30px";
    div.style.lineHeight = "200px";
    div.style.textAlign = "center";

    // 设置元素内容
    div.innerHTML = "hello world";
</script>
```

通过 `style` 属性读取和设置的是行内样式，无法读取和设置样式表中的样式。

通过 `style` 属性设置的样式是行内样式，具有较高的优先级，所以通过这种方式设置的样式往往会立即显示。

如果在样式中有 `!important`，则该样式具有最高的优先级，此时通过 JavaScript 修改样式会失效。因此尽量不要为样式添加 `!important`。

16.8.2 获得内部样式和外部样式

1. `元素对象.currentStyle.样式名`：访问元素当前生效的样式。如果未设置样式值，则返回该样式的默认值。只支持 IE 浏览器。
2. `window.getComputedStyle(element)`：获得指定元素对象当前生效的样式，返回`CSSStyleDeclaration`对象。获得对象后，通过访问样式属性获得样式值。如果未设置样式值，则该样式的值为当前真实值。IE 8 及以下版本不支持`window.getComputedStyle()`方法。

这两种方式获得的样式都是只读的，不能修改。

例如：

```
<style>
    div {
        width: 200px;
        height: 200px;
        background-color: orange;
        border: 1px solid;
        text-align: center;
        line-height: 200px;
        border-radius: 50%;
    }
</style>

<div></div>

<script>
    var div = document.querySelector("div");

    // 1. 元素对象.currentStyle.样式名
    console.log(div.currentStyle.width); // 200px
    console.log(div.currentStyle.backgroundColor); // orange

    // 2. window.getComputedStyle()
    var styles = window.getComputedStyle(div, null);
    console.log(styles); // CSSStyleDeclaration
    console.log(styles.width); // 200px
</script>
```

`currentStyle`属性只在 IE 浏览器中支持，而`getComputedStyle()`函数不支持 IE 8 及以下版本，二者只有一个可以使用。为了解决兼容性问题，可以自定义一个函数，如下所示：

```
/*
 * 获取指定元素的指定样式值
 * element：要获取样式的元素
 * style：要获取样式名
 */
function getStyle(element, style)
{
    if (window.getComputedStyle) // 如果 window.getComputedStyle 存在，就使用它
    {
        return getComputedStyle(element, null)[style];
    }
    else // 否则，该函数不存在，就使用 currentStyle
    {
        return element.currentStyle[style];
```

```
    }
}
```

16.8.3 其他样式相关的属性

1. `element.clientHeight`：返回元素的可见高度，包括元素的`content`和`padding`部分。返回值是不带单位的数字，可以直接计算。
2. `element.clientWidth`：返回元素的可见宽度，包括元素的`content`和`padding`部分。返回值是不带单位的数字，可以直接计算。
3. `element.offsetHeight`：返回元素的高度，包括`content`、`padding`和`border`。
4. `element.offsetWidth`：返回元素的宽度，包括`content`、`padding`和`border`。
5. `element.offsetParent`：返回元素的偏移容器。用于获取最近的开启了定位的祖先元素。
6. `element.offsetLeft`：返回元素的水平偏移位置。
7. `element.offsetTop`：返回元素的垂直偏移位置。
8. `element.scrollHeight`：返回元素的整体高度。用于获取元素整个滚动区域的高度。
9. `element.scrollWidth`：返回元素的整体宽度。用于获取元素整个滚动区域的宽度。
10. `element.scrollLeft`：返回元素左边缘与视图之间的距离。用于获取水平滚动条的滚动距离。
11. `element.scrollTop`：返回元素上边缘与视图之间的距离。用于获取垂直滚动条的滚动距离。

以上所有属性都是只读的，不可修改。

当满足`scrollHeight - scrollTop == clientHeight`时，说明垂直滚动条滚动到底；当满足`scrollWidth - scrollLeft == clientWidth`时，说明水平滚动条滚动到底。

16.9 事件

16.9.1 事件简介

事件就是文档或浏览器窗口中发生的一些特定的交互瞬间。JavaScript 与 HTML 之间的交互是通过事件实现的。

可以为节点对象绑定事件响应函数，当事件发生时，执行对应的响应函数。

绑定事件有两种方法。第一种方法是在 HTML 标签的事件属性中设置 JavaScript 代码。这种写法造成结构和行为耦合，不方便维护，因此不推荐使用。例如：

```
<!-- 为 button 元素设置单击事件 -->
<button onclick="alert('hello world');">按钮</button>
```

第二种方法是为节点对象的事件属性绑定响应函数。例如：

```
<button id="btn">按钮</button>
<script>
    var btn = document.getElementById("btn");
    // 为按钮绑定单击响应函数
    btn.onclick = function() {
        alert('hello world');
    };
</script>
```

事件响应函数的调用者是被绑定的节点对象，事件响应函数中的`this`就是被绑定的节点对象。

16.9.2 页面加载事件

HTML 页面的加载方式是从上到下逐句加载，读到一行就加载一行。如果把 `script` 标签写在 `body` 上面，就会在页面加载完成之前执行 JavaScript 代码，可能导致 JavaScript 的执行出错。例如：

```
<head>
    <script>
        // 获取 id 为 btn 的按钮
        var btn = document.getElementById("btn"); // 按钮还未加载，获取不到
        // 为按钮绑定单击响应函数
        btn.onclick = function() { // btn 为 null，无法设置 onload 属性，报错
            alert("hello");
        };
    </script>
</head>
<body>
    <button id="btn">点击</button>
</body>
```

`onload` 事件在对象加载完成后触发。支持 `onload` 事件的 JavaScript 对象有：`image`、`layer`、`window`。

为 `window` 对象绑定 `onload` 响应函数后，将在整个页面加载完成后执行相应代码。例如：

```
<head>
    <script>
        // 为 window 对象绑定 onload 事件响应函数
        window.onload = function() {
            // 获取 id 为 btn 的按钮
            var btn = document.getElementById("btn");
            // 为按钮绑定单击响应函数
            btn.onclick = function() {
                alert("hello");
            };
        };
    </script>
</head>
<body>
    <button id="btn">点击</button>
</body>
```

16.9.3 事件对象

当事件的响应函数被触发时，浏览器会将一个事件对象作为实参传入响应函数。可以在响应函数中定义形参，以便在函数体中使用该事件对象。例如：

```
var div = document.getElementById("div");
// 鼠标移动事件
div.onmousemove = function(event) {
    alert(event); // [object MouseEvent]
};
```

在事件对象中封装了当前事件相关的一切信息，可以通过事件对象的属性获取相关信息。例如，事件对象的鼠标/键盘属性如下表所示。

属性	描述
`altKey`	当事件被触发时，ALT 键是否被按下
`ctrlKey`	当事件被触发时，CTRL 键是否被按下
`metaKey`	当事件被触发时，meta 键是否被按下
`shiftKey`	当事件被触发时，SHIFT 键是否被按下
`clientX`	当事件被触发时，鼠标指针在可见窗口中的水平坐标
`clientY`	当事件被触发时，鼠标指针在可见窗口中的垂直坐标
`pageX`	当事件被触发时，鼠标指针在整个页面中的水平坐标 (IE 8 及以下版本不支持)
`pageY`	当事件被触发时，鼠标指针在整个页面中的垂直坐标 (IE 8 及以下版本不支持)

例如，可以在事件被触发时获取鼠标指针的坐标并显示：

```
var div = document.getElementById("div");
div.onmousemove = function(event) {
    var x = event.clientX;
    var y = event.clientY;
    alert("x = " + x + ", y = " + y);
};
```

在 IE 8 及以下版本，当响应函数被触发时，浏览器不会传递事件对象，而是将事件对象作为`window`对象的一个属性，使用`window.event`访问。例如：

```
var div = document.getElementById("div");
div.onmousemove = function() {
    var x = window.event.clientX;
    var y = window.event.clientY;
    alert("x = " + x + ", y = " + y);
};
```

火狐浏览器不支持`window.event`属性。解决事件对象兼容性问题的方案如下：

```
var div = document.getElementById("div");
div.onmousemove = function(event) {
    // 如果传入了 event 实参，就直接使用 event
    // 如果没有传入实参，就使用 window.event
    // 这里利用了逻辑或运算的短路特性
    event = event || window.event;

    var x = event.clientX;
    var y = event.clientY;
    alert("x = " + x + ", y = " + y);
};
```

16.9.4 事件的冒泡

当元素的事件被触发时，其祖先元素的相同事件也会被触发，这种现象叫做事件的冒泡（bubble）。例如：

```
<head>
    <style type="text/css">
        #box1 {
            width: 200px;
            height: 200px;
            background-color: yellowgreen;
        }

        #s1 {
            background-color: yellow;
        }
    </style>

    <script type="text/javascript">
        window.onload = function() {
            var s1 = document.getElementById("s1");
            s1.onclick = function() {
                alert("我是span");
            };

            var box1 = document.getElementById("box1");
            box1.onclick = function() {
                alert("我是div");
            };

            document.body.onclick = function() {
                alert("我是body");
            };
        };
    </script>
</head>
<body>
    <div id="box1">
        我是div
        <span id="s1">我是span</span>
    </div>
</body>
```

在上面的代码中，分别为`span`、`div`和`body`绑定了单击响应函数。当单击`span`时，触发`span`、`div`、`body`的单击事件；当单击`div`时，触发`div`和`span`的单击事件。

如果不希望发生事件冒泡，可以通过事件对象取消冒泡。事件对象的`cancelBubble`属性用于取消冒泡，当取值为`true`时取消冒泡。

16.9.5 事件的委派

观察下列代码：

```
<head>
    <script type="text/javascript">
        window.onload = function() {
```

```
            // 为每个超链接绑定一个单击响应函数
            var allA = document.getElementByTagName("a");
            for (var i = 0; i < allA.length; i++)
            {
                allA[i].onclick = function() {
                    alert("我是a的单击响应函数");
                }
            }

            var u1 = document.getElementById("u1");

            // 点击按钮后添加超链接
            var btn01 = document.getElementById("btn01");
            btn01.onclick = function() {
                // 创建一个 li
                var li = document.createElement("li");
                li.innerHTML = "<a href='javascript:;'>新建的超链接</a>"
                // 将 li 添加到 ul 中
                u1.appendChild(li);
            };
        };
    </script>
</head>
<body>
    <button id="btn01">添加超链接</button>
    <ul id="u1">
        <li><a href="javascript:;">超链接1</a></li>
        <li><a href="javascript:;">超链接2</a></li>
        <li><a href="javascript:;">超链接3</a></li>
    </ul>
</body>
```

为每个超链接绑定一个单击响应函数，这样的操作成本较高、影响性能，而且只能为已有的超链接绑定响应函数，新添加的超链接必须重新绑定。

目标：只绑定一次即可应用到多个元素上，并且后添加的元素能自动绑定响应函数。

解决方案：事件的委派。将事件的响应函数绑定给共同的祖先元素，当后代元素上的事件触发时，通过冒泡触发祖先元素的响应函数。

利用事件的委派对上述代码进行改进，如下所示：

```
<head>
    <script type="text/javascript">
        window.onload = function() {
            // 为所有超链接的共同父元素绑定一个单击响应函数
            var u1 = document.getElementById("u1");
            u1.onclick = function(event) {
                // 如果触发事件的对象是期望的元素则执行，否则不执行
                if (event.target.className == "link")
                {
                    alert("我是ul的单击响应函数");
                }
            };

            // 点击按钮后添加超链接
            var btn01 = document.getElementById("btn01");
            btn01.onclick = function() {
```

```
                // 创建一个 li
                var li = document.createElement("li");
                li.innerHTML = "<a href='javascript:;' className='link'>新建的超链接</a>"
                // 将 li 添加到 ul 中
                ul.appendChild(li);
            };
        };
    </script>
</head>
<body>
    <button id="btn01">添加超链接</button>
    <ul id="ul">
        <li><a href="javascript:;" className="link">超链接1</a></li>
        <li><a href="javascript:;" className="link">超链接2</a></li>
        <li><a href="javascript:;" className="link">超链接3</a></li>
    </ul>
</body>
```

事件对象的`target`属性返回触发此事件的元素。这里使用`target`属性判断触发事件的元素是否为期望的元素，如果是则执行，否则不执行。

16.9.6 事件的绑定

使用`object.event = function`的形式绑定响应函数时，只能为元素的相同事件绑定一个响应函数，不能绑定多个响应函数，后面的会覆盖前面的。

`addEventListener()`函数可以为对象绑定响应函数，其参数为：

1. 事件名称字符串（去掉 on）。例如，`onclick`事件在参数中写成`"click"`。
2. 响应函数。
3. 是否在捕获阶段触发事件。布尔值，一般用`false`。

`addEventListener()`函数可以为对象的相同事件绑定多个响应函数，当事件被触发时，响应函数按照绑定的顺序从先到后依次执行。

IE 8 及以下版本不支持`addEventListener()`函数。在 IE 8 及以下版本中，可以使用`attachEvent()`函数，其参数为：

1. 事件名称字符串（带 on）。
2. 响应函数。

`attachEvent()`函数也可以为对象的相同事件绑定多个响应函数。与`addEventListener()`函数不同的是，`attachEvent()`函数的响应函数执行顺序相反，后绑定的先执行。

`addEventListener()`函数的响应函数中的`this`是绑定事件的对象，`attachEvent()`函数的响应函数中的`this`是`window`，需要统一不同的`this`。

事件的绑定存在兼容性问题，解决方案如下：

```
/*
 * 为指定的对象绑定指定事件的响应函数
 * obj: 要绑定事件的对象
 * eventStr: 事件名称字符串(不带 on)
 * callback: 回调函数
 */
function bind(obj, eventStr, callback)
{
```

```
    if (obj.addEventListener)
    {
        obj.addEventListener(eventStr, callback, false);
    }
    else
    {
        obj.attachEvent("on" + eventStr, function(){
            // 在匿名函数中调用回调函数
            // 用 call 方法指定 this 为 obj，解决 this 不统一的问题
            callback.call(obj);
        });
    }
}
```

16.9.7 事件的传播

网景公司和微软公司对事件的传播有不同的理解：

- 微软公司认为事件应该由内向外传播，当事件触发时，先触发当前元素的事件，再向祖先元素传播。
- 网景公司认为事件应该由外向内传播，当事件触发时，先触发当前元素的最外层的祖先元素的事件，然后再向内传播给后代元素。

W3C 综合了两种方案，将事件的传播分成 3 个阶段：

1. 捕获阶段：从最外层的祖先元素开始，向目标元素进行事件的捕获，但不触发事件。
2. 目标阶段：捕获到目标元素的事件，捕获阶段结束，开始在目标元素上触发事件。
3. 冒泡阶段：从目标元素开始，向祖先元素传递，依次触发祖先元素上的事件。

如果希望在捕获阶段触发事件，可以将`addEventListener()`函数的第三个参数设置为`true`。一般情况下不会在捕获阶段触发事件，因此一般都把这个参数设置为`false`。

IE 8 及以下版本的浏览器中没有捕获阶段。

16.9.8 滚轮事件

`onmousewheel`是滚轮滚动事件。火狐不支持该属性，在火狐中滚动事件的名称为`DOMMouseScroll`，并且需要使用`addEventListener()`函数来绑定。

事件对象的`wheelDelta`属性用于获取滚轮的滚动方向，向上滚动返回 120，向下滚动返回 -120。火狐不支持该属性，在火狐中使用`detail`属性获取滚轮的滚动方向，向上滚动返回 -3，向下滚动返回 3。

如果浏览器有滚动条，那么当滚轮滚动时，滚动条会随之滚动，这是浏览器的默认行为。我们希望当滚轮滚动时，只触发特定元素的滚轮事件，而不影响浏览器的滚动条。可以在响应函数最后使用`return false`取消默认行为。

在火狐中通过`addEventListener()`函数来绑定滚轮事件，取消默认行为时不能使用`return false`，而应该使用事件对象的`preventDefault()`函数。IE 8 及以下版本不支持`preventDefault()`函数，因此使用该函数时需要判断函数是否存在。

综上，解决滚轮事件兼容性问题的方法如下：

```
// 其他浏览器中绑定滚轮滚动事件
obj.onmousewheel = function(event) {
    event = event || window.event;

    if (event.wheelDelta > 0 || event.detail < 0)
    {
```

```
        alert("向上滚动");
    }
    else
    {
        alert("向下滚动");
    }

    // 火狐浏览器中取消浏览器默认行为
    // 利用逻辑与运算的短路特性，当该函数存在时才调用
    event.preventDefault && event.preventDefault();

    // 其他浏览器中取消默认行为
    return false;
};

// 火狐浏览器中绑定滚轮滚动事件
// bind 函数在 16.10.6 节定义
bind(obj, "DOMMouseScroll", obj.onmousewheel);
```

16.9.9 键盘事件

`onkeydown`是键盘按键按下事件，`onkeyup`是键盘按键松开事件。

键盘事件一般都会绑定给可以获取焦点的元素，或者是`document`。

对于`onkeydown`事件，如果一直按着按键不松手，事件会一直被触发。当`onkeydown`事件连续触发时，第一次和第二次之间间隔较长，之后间隔较短，这种设计是为了防止误操作。

事件对象的`keyCode`属性返回按键的 Unicode 编码，通过它可以判断哪个键被按下。

除了`keyCode`，事件对象中还有`altKey`、`ctrlKey`和`shiftKey`属性，分别用于判断 alt、ctrl 和 shift 是否被按下，如果按下则返回`true`，否则返回`false`。

在文本框中输入内容属于`onkeydown`的默认行为，如果在`onkeydown`的响应函数中使用`return false`取消了默认行为，则输入的内容不会出现在文本框中。借助这一特性可以使文本框不能输入某些字符，例如：

```
input.onkeydown = function(event) {
    event = event || window.event;

    // 使文本框不能输入数字
    // 数字的编码范围是 48-57
    if (event.keyCode >= 48 && event.keyCode <= 57)
    {
        return false;
    }
};
```

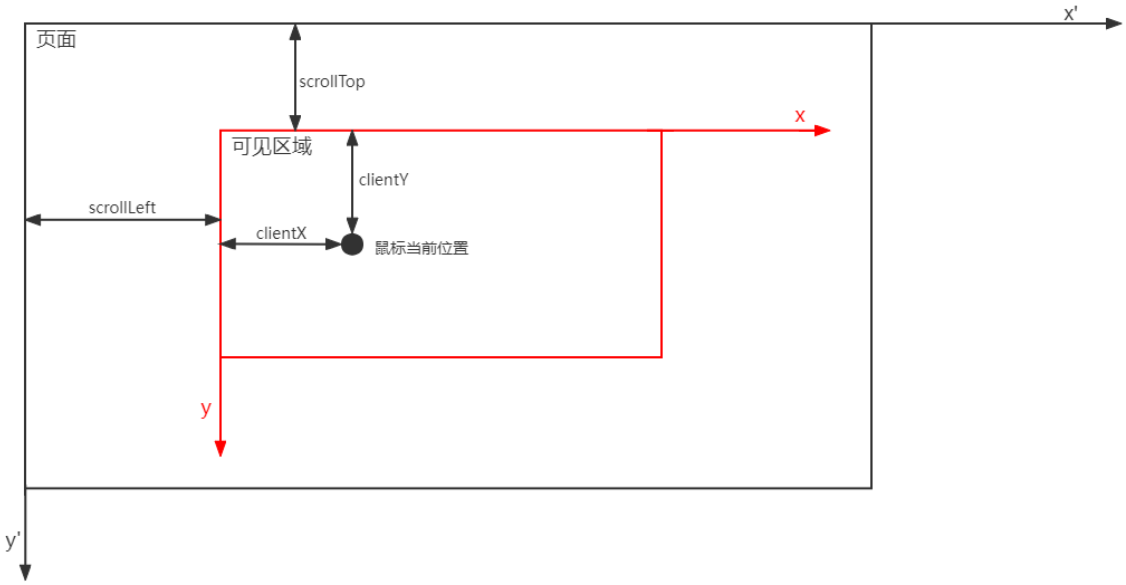
16.9.10 练习1：div跟随鼠标移动

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title></title>

```

```
        <style type="text/css">
            #box1 {
                width: 100px;
                height: 100px;
                background-color: red;
                position: absolute; /* 设置绝对定位 */
            }
        </style>

        <script type="text/javascript">
            window.onload = function() {
                var box1 = document.getElementById("box1");
                // 为整个页面绑定鼠标移动事件
                document.onmousemove = function(event) {
                    // 解决事件对象的兼容问题
                    event = event || window.event;

                    // 获取鼠标的坐标
                    var left = event.clientX;
                    var top = event.clientY;

                    // 获取滚动条的滚动距离
                    /*
                     * 在 chrome 中，滚动条是 body 对象的
                     * 在其他浏览器中，滚动条是 html 对象的
                     * 此处需要解决兼容性问题
                     */
                    var scrollTop = document.body.scrollTop ||
                                    document.documentElement.scrollTop;
                    var scrollLeft = document.body.scrollLeft ||
                                     document.documentElement.scrollLeft;

                    // 设置 div 的偏移量
                    /*
                     * clientX 和 clientY 是鼠标相对于可见窗口的坐标
                     * div 的偏移量是相对于整个页面的
                     * 如果有滚动条，会使得鼠标和div的坐标系不一致，导致div与鼠标分离
                     * 因此需要使用滚动条的滚动距离来修正，使鼠标始终在div的左上角
                     */
                    box1.style.left = left + scrollLeft + "px";
                    box1.style.top = top + scrollTop + "px";
                };
            };
        </script>
    </head>
    <body>
        <div id="box1"></div>
    </body>
</html>
```

要点：

1. 鼠标移动事件要绑定到 `document` 上。
2. 整个页面和可见区域的坐标系并不一致。通过 `clientX` 和 `clientY` 获取到的鼠标位置是相对于可见区域的，而 div 的偏移量是相对于整个页面的。计算 div 的偏移量时，需要考虑可见区域的滚动距离，如下图所示。



3. 获取滚动条的滚动距离时，需要解决兼容性问题。

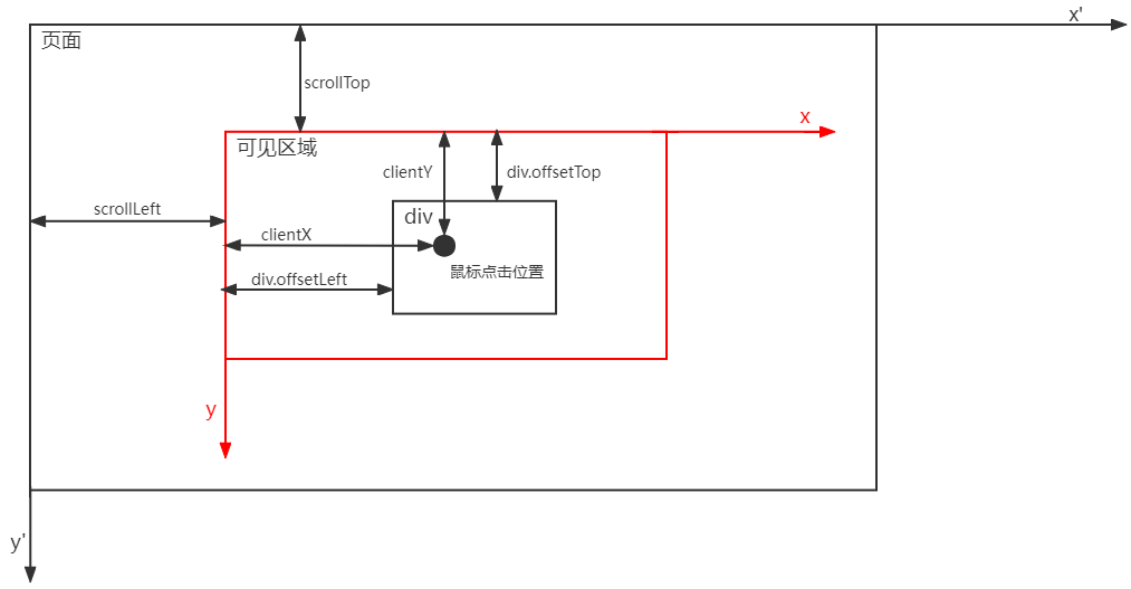
16.9.11 练习2：拖拽

```
/*
 * 拖拽的流程:
 * 1.当鼠标在被拖拽元素上按下时(onmousedown)，开始拖拽
 * 2.当鼠标移动时(onmousemove)，被拖拽元素跟随鼠标移动
 * 3.当鼠标松开时(onmouseup)，被拖拽元素固定在当前位置
 */

/*
 * 为指定的元素开启拖拽
 * obj: 要开启拖拽的元素
 */
function drag(obj)
{
    // 当鼠标在 obj 上按下时，开始拖拽
    obj.onmousedown = function(event) {
        event = event || window.event;

        // 设置 obj 捕获所有鼠标按下相关事件
        // 用于在 IE 8 及以下版本中取消打开搜索引擎的默认行为
        // 利用逻辑与运算的短路特性，当该函数存在时才调用，解决兼容性问题
        obj.setCapture && obj.setCapture();

        // 计算鼠标点击位置相对于 obj 的偏移量
        var offsetX = event.clientX - obj.offsetLeft;
        var offsetY = event.clientY - obj.offsetTop;

        // 为 document 绑定鼠标移动事件
        // 当鼠标移动时，obj 跟随鼠标移动
        document.onmousemove = function(event) {
            event = event || window.event;
            // 获取鼠标位置
            // 修正坐标，使得鼠标与 box1 的相对位置保持不变
            var left = event.clientX - offsetX;
            var top = event.clientY - offsetY;
            // 获取滚动条的滚动距离
```

```
            var scrollTop = document.body.scrollTop ||
                            document.documentElement.scrollTop;
            var scrollLeft = document.body.scrollLeft ||
                             document.documentElement.scrollLeft;
            // 修改 obj 的位置
            obj.style.left = left + scrollLeft + "px";
            obj.style.top = top + scrollTop + "px";
        };

        // 为 document 绑定鼠标松开事件
        // 当鼠标松开时，obj 固定在当前位置
        document.onmouseup = function() {
            // 取消 document 的 onmousemove 事件
            document.onmousemove = null;
            // 取消 document 的 onmouseup 事件
            document.onmouseup = null;
            // 取消鼠标按下相关事件的捕获
            obj.releaseCapture && obj.releaseCapture();
        };

        // 取消打开搜索引擎的默认行为
        return false;
    };
}
```

要点：

1. 鼠标松开事件要绑定到`document`上，否则会使得当鼠标在其他元素内部松开时，被拖拽元素无法固定。
2. 当鼠标松开时，拖拽过程就结束了，此时要将鼠标移动事件和鼠标松开事件取消。
3. 鼠标移动和鼠标松开事件的响应函数要写在鼠标按下事件响应函数的内部。
4. 为了使鼠标与被拖拽元素的相对位置保持不变，需要计算鼠标点击位置相对于被拖拽元素的偏移量，如下图所示。水平方向上，`offsetX = clientX - div.offsetLeft`，则 div 的偏移量为`div.style.left = scrollLeft + clientX - offsetX`。垂直方向同理。



5. 当拖拽网页中的内容时，浏览器会默认打开搜索引擎搜索内容，此时会导致拖拽功能的异常。如果不希望发生这个行为，可以在鼠标按下事件的响应函数最后添加`return false`来取消默认行为。这种方法对于 IE 8 及以下版本不起作用，在 IE 8 及以下版本中使用`setCapture()`函数，将下一

次所有的鼠标按下相关事件捕获到被拖拽元素上，在鼠标松开时使用`releaseCapture()`函数释放捕获。

17 BOM

17.1 BOM简介

BOM 全称为浏览器对象模型（Broser Object Model），BOM 可以使我们通过 JavaScript 来操作浏览器。

BOM 提供了一组对象，用来完成对浏览器的操作。BOM 对象有：

1. `Window`：代表整个浏览器的窗口。`window`是网页中的全局对象。
2. `Navigator`：代表当前浏览器的信息。通过该对象可以识别不同的浏览器。
3. `Location`：代表当前浏览器的地址栏信息。通过该对象可以获取地址栏信息，或者操作浏览器跳转页面。
4. `History`：代表浏览器的历史记录。通过该对象可以操作浏览器的历史记录。由于隐私原因，该对象不能获取到具体的历史记录，只能操作浏览器向前或向后翻页，而且该操作只在当次访问有效。
5. `Screen`：代表用户屏幕的信息。通过该对象可以获取用户的显示器的相关信息。

这些 BOM 对象在浏览器中都是作为`window`对象的属性保存的，可以通过`window`对象来使用，也可以省略`window`直接使用。例如，`Navigator`对象可以使用`window.navigator`来访问，也可以直接使用`navigator`来访问。

17.2 `Navigator`

`Navigator`对象的属性如下表所示。

属性	描述
`appCodeName`	浏览器的代码名
`appName`	浏览器的名称
`appVersion`	浏览器的平台和版本信息
`appMinorVersion`	浏览器的次级版本
`browserLanguage`	当前浏览器的语言
`cookieEnabled`	浏览器是否启用 cookie
`cpuClass`	浏览器系统的 CPU 等级
`onLine`	系统是否处于脱机模式
`platform`	运行浏览器的操作系统平台
`systemLanguage`	操作系统使用的默认语言
`userAgent`	客户机发送给服务器的 user-agent 头部的值
`userLanguage`	操作系统的自然语言设置

由于历史原因，`Navigator`对象中的大部分属性已经不能帮助我们识别浏览器了。一般我们只用`userAgent`属性来判断浏览器的信息。

`userAgent`的值是一个字符串，其中包含用来描述浏览器的内容，不同的浏览器有不同的值，据此可以使用`userAgent`判断是哪个浏览器。例如：

```
if (/firefox/i.test(navigator.userAgent))
{   // 火狐浏览器的 userAgent 中包含 "Firefox"
    alert("火狐");
}
else if (/chrome/i.test(navigator.userAgent))
{   // Chrome 的 userAgent 中包含 "Chrome"
    alert("Chrome")
}
else if (/msie/i.test(navigator.userAgent))
{   // IE 11 以下版本的 userAgent 中包含 "MSIE"
    alert("IE")
}
```

在 IE 11 中已经将微软和 IE 相关的标识都去除了，所以我们已经基本不能通过`userAgent`来识别浏览器是否是 IE 了。如果通过`userAgent`不能判断，还可以通过一些浏览器中特有的对象来判断浏览器的信息。例如，`window.ActiveXObjevt`只在 IE 浏览器中存在，可以用它来判断是否是 IE 浏览器。

```
/*
 * 在 IE 11 以下版本中，window.ActiveXObject 转换为 true，可以正常判断
 * 在 IE 11 中，window.ActiveXObject 转换为 false，使判断失效
 * 因此必须使用 "ActiveXObject" in window 作为判断条件
 */
if ("ActiveXObject" in window)
{
    alert("是IE");
}
else
{
    alert("不是IE");
}
```

17.3 History

`History`对象可以用来操作浏览器向前或向后翻页。

`History`对象只有一个属性`length`，可以获取浏览器历史列表中的 URL 数量。

`History`对象的方法如下表所示。

方法	描述
`back()`	加载历史列表中的上一个 URL
`forward()`	加载历史列表中的下一个 URL
`go()`	加载历史列表中的某个具体页面

`go()`方法的参数为整数，n 表示向下跳转 n 个页面，-n 表示向上跳转 n 个页面。

17.4 Location

`Location`对象中封装了浏览器的地址栏信息。使用`alert(location)`可以直接获取到地址栏中的完整路径。

如果将`location`属性修改为一个完整的路径，则页面会自动跳转到该路径，并且会生成相应的历史记录。

`Location`对象的属性如下表所示。

属性	描述
`hash`	设置或返回从 # 开始的 URL（锚）
`host`	设置或返回主机名和当前 URL 的端口号
`hostname`	设置或返回当前 URL 的主机名
`href`	设置或返回完整的 URL
`pathname`	设置或返回当前 URL 的路径部分
`port`	设置或返回当前 URL 的端口号
`protocol`	设置或返回当前 URL 的协议
`search`	设置或返回从 ? 开始的 URL 查询部分

`Location`对象的方法如下表所示。

方法	描述
`assign()`	加载新的文档，用于跳转到其他页面。参数为页面的路径。
`reload()`	重新加载当前文档，相当于刷新页面。 如果传递`true`作为参数，则会强制清空缓存刷新页面。
`replace()`	用新的文档替换当前文档，并且不生成历史记录。

17.5 定时器

17.5.1 定时调用

`window.setInterval(callback, millis)`方法：

1. 它是`window`对象的方法，是一个全局方法。
2. 第一个参数是一个回调函数。第二个参数是时间，单位为毫秒。
3. 功能：每隔 millis 毫秒执行一次回调函数。
4. 返回值：一个整数，代表该定时器的编号。
5. 停止定时器：`window.clearInterval(定时器编号)`。该函数可以接收任意参数。如果参数是一个有效的定时器编号，则停止对应的定时器；如果参数不是一个有效的编号，则什么也不做。
6. 可以同时开启多个定时器，各个定时器的编号各不相同。

当在一个元素上开启多个定时器，并且这些定时器的回调函数相同时，要在启动新的定时器之前将上一个定时器关闭。例如：

```
var timer; // 定时器编号
var btn = document.getElementByTagName("button")[0];

// 当点击按钮时，开启一个定时器
btn.onclick = function() {
    // 关闭上一个定时器
    clearInterval(timer);
    // 开启新的定时器
    timer = setInterval(function(){
        // 回调函数
    }, 1000);
};
```

17.5.2 延时调用

`window.setTimeout(callback, millis)`方法：

1. 它是`window`对象的方法，是一个全局方法。
2. 第一个参数是一个回调函数。第二个参数是时间，单位为毫秒。
3. 功能：间隔 millis 毫秒后执行回调函数，且只执行一次。
4. 返回值：一个整数，代表该延时器的编号。
5. 停止延时器：`window.clearTimeout(延时器编号)`。该函数可以接收任意参数。如果参数是一个有效的编号，则停止对应的延时器；如果参数不是一个有效的编号，则什么也不做。
6. 可以同时开启多个延时器，各个延时器的编号各不相同。

定时调用和延时调用可以相互代替，在开发中可以根据需要进行选择。

17.5.3 定时器的应用：元素动画效果

```
/**
 * 为指定的元素开启动画效果
 * obj: 要执行动画的对象
 * attr: 要执行动画的样式
 * target: 目标样式值
 * speed: 动画速度(正值)
 * callback: 回调函数，在动画执行完毕后执行
 */
function movie(obj, attr, target, speed, callback)
{
    // 关闭上一个定时器
    clearInterval(obj.timer);

    // 获取元素的当前样式值
    // getStyle() 函数定义在 16.8.2 节
    var current = parseInt(getStyle(obj, attr));

    // 判断速度的正负
    // 如果 current < target，则速度为正
    // 如果 current > target，则速度为负
    if (current > target)
    {
        speed = -speed;
    }

    // 开启一个定时器，执行动画效果
    // 向执行动画的对象中添加一个 timer 属性，用于保存它自己的定时器编号
```

```
    obj.timer = setInterval(function(){
        // 获取 obj 原来的样式值
        // getStyle() 函数定义在 16.8.2 节
        var oldValue = parseInt(getStyle(obj, attr));

        // 在旧值的基础上计算新值
        var newValue = oldValue + speed;

        // 判断是否到达目标
        // 速度为负时，需要判断 newValue 是否小于 target
        // 速度为正时，需要判断 newValue 是否大于 target
        if ((speed < 0 && newValue < target) ||
           (speed > 0 && newValue > target))
        {
            newValue = target;
        }

        // 将新值设置给 obj
        obj.style[attr] = newValue + "px";

        // 当元素到达目标位置时，关闭定时器
        if (newValue == target)
        {
            clearInterval(obj.timer);
            // 动画执行完毕，调用回调函数
            // 当用户传入回调函数时才调用
            callback && callback();
        }
    }, 30);
}
```

18 应用实例

18.1 轮播图

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>轮播图</title>

        <style type="text/css">
            * {
                margin: 0;
                padding: 0;
            }

            /* 外部容器 */
            #outer {
                /* 根据图片大小设置宽高 */
                width: 520px;
                height: 333px;
                /* 水平居中 */
                margin: 50px auto;
                /* 设置背景颜色 */
                background-color: greenyellow;
```

```
            /* 设置上下padding */
            padding: 10px 0;
            /* 开启相对定位 */
            position: relative;
            /* 裁剪溢出内容 */
            overflow: hidden;
        }

        #imgList {
            /* 去除项目符号 */
            list-style: none;
            /* 开启绝对定位 */
            position: absolute;
            /* 设置偏移量 */
            left: 0;
        }

        #imgList li {
            /* 向左浮动 */
            float: left;
            /* 设置左右外边距 */
            margin: 0 10px;
        }

        /* 导航按钮容器 */
        #navDiv {
            /* 开启绝对定位 */
            position: absolute;
            /* 设置位置 */
            bottom: 15px;
        }

        /* 导航按钮超链接 */
        #navDiv a {
            /* 向左浮动 */
            float: left;
            /* 设置宽高 */
            width: 15px;
            height: 15px;
            /* 设置左右外边距 */
            margin: 0 5px;
            /* 设置背景颜色 */
            background-color: red;
            /* 设置透明度 */
            opacity: 0.5;
            /* 兼容 IE8 透明 */
            filter: alpha(opacity=50);
        }

        /* 设置鼠标移入效果 */
        #navDiv a:hover {
            background-color: black;
        }
    </style>

    <script type="text/javascript">
        window.onload = function() {
            // 获取页面中所有的 img 标签
```

```
                var imgArr = document.getElementByTagName("img");
                // 设置 imgList 的宽度
                var imgList = document.getElementById("imgList");
                imgList.style.width = 520 * imgArr.length + "px";

                // 设置导航按钮水平居中
                var outer = document.getElementById("outer");
                var navDiv = document.getElementById("navDiv");
                navDiv.style.left = (outer.offsetWidth -
navDiv.offsetWidth) / 2 + "px";

                // 当前显示图片的索引
                var index = 0;
                // 获取所有的 a 标签
                var allA = document.getElementByTagName("a");
                // 设置默认选中的效果
                allA[index].style.backgroundColor = "black";

                // 自动切换的定时器标识
                var timer;

                // 点击超链接切换到指定的图片
                for (var i = 0; i < allA.length; i++)
                {
                    // 关闭自动切换的定时器
                    clearInterval(timer);
                    // 为每个超链接添加一个索引属性
                    allA[i].index = i;
                    // 为每个超链接绑定单击响应函数
                    allA[i].onclick = function() {
                        // 获取点击超链接的索引
                        index = this.index;
                        // 切换图片
                        // movie() 函数定义在 17.5.3 节
                        movie(imgList, "left", -520 * index, 20, function()
{
                            // 动画执行结束后开启自动切换图片
                            autoChange();
                        });
                        // 修改超链接的样式
                        setA();
                    };
                }

                // 开启自动切换图片
                autoChange();

                function setA()
                {
                    // 到达最后一张图片
                    if (index >= imgArr - 1)
                    {
                        // 将索引设置为 0
                        index = 0;
                        // 最后一张图片与第一张相同，将最后一张切换到第一张
                        imgList.style.left = 0;
                    }

```

```
                    for (var i = 0; i < allA.length; i++)
                    {
                        // 将行内样式值清空，内部样式生效
                        // 如果在这里设置颜色值，将使得 hover 失效
                        allA[i].style.backgroundColor = "";
                    }
                    allA[i].style.backgroundColor = "black";
                }

                // 自定义函数，用来开启自动切换图片
                function autoChange()
                {
                    // 开启定时器
                    timer = setInterval(function(){
                        // 索引自增
                        index++;
                        index = index % imgArr.length;
                        // 执行切换
                        movie(imgList, "left", -520 * index, 20);
                        // 修改导航按钮样式
                        setA();
                    }, 3000);
                }
            };
        </script>
    </head>
    <body>
        <!-- 创建一个外部div作为容器 -->
        <div id="outer">
            <!-- 创建一个ul用于放置图片 -->
            <ul id="imgList">
                <li><img src="img/1.jpg"></li>
                <li><img src="img/2.jpg"></li>
                <li><img src="img/3.jpg"></li>
                <!-- 在最后一张图后面放第一张图 -->
                <li><img src="img/1.jpg"></li>
            </ul>
            <!-- 创建导航按钮 -->
            <div id="navDiv">
                <a href="javascript:;"></a>
                <a href="javascript:;"></a>
                <a href="javascript:;"></a>
            </div>
        </div>
    </body>
</html>
```

18.2 `class`属性的操作

将元素的不同样式设置为不同的类名，这样可以通过修改元素的`class`属性来间接地修改样式。当需要向元素增加新的样式时，可以为元素添加新的`class`属性值。例如：

```
<head>
    <style type="text/css">
        .b1 {
            width: 100px;
```

```
            height: 100px;
            background-color: red;
        }

        .b2 {
            width: 200px;
            height: 200px;
            background-color: yellow;
        }
    </style>

    <script type="text/javascript">
        window.onload = function() {
            var box = document.getElementById("box");
            var btn01 = document.getElementById("btn01");
            btn01.onclick = function() {
                // 通过修改 box 的 class 属性来间接地修改样式
                box.className = "b2";
            };
        };
    </script>
</head>
<body>
    <button id="btn01">切换样式</button>
    <div id="box" class="b1"></div>
</body>
```

这样做的好处在于：

1. 只需要修改一次即可同时修改多个样式，浏览器只需要重新渲染一次，性能比较好。
2. 使表现和行为进一步地分离。

下面的工具函数提供了修改元素 `class` 属性值的功能。

```
/*
 * 判断元素中是否含有指定的 class 属性值，如果有则返回 true，否则返回 false
 * obj: 元素
 * className: class 属性值
 */
function hasClassName(obj, className)
{
    // 创建一个正则表达式
    var reg = new RegExp("\\b" + className + "\\b");
    // 返回匹配结果
    return reg.test(obj.className);
}

/*
 * 为元素添加指定的 class 属性值
 * obj: 元素
 * className: class 属性值
 */
function addClassName(obj, className)
{
    if (!hasClassName(obj, className))
    {   // 当 obj 中没有此 class 属性值时才添加
        obj.className += " " + className;
```

```
    }
}

/*
 * 删除元素中指定的 class 属性值
 * obj: 元素
 * className: class 属性值
 */
function removeClassName(obj, className)
{
    // 创建一个正则表达式
    var reg = new RegExp("\\b" + className + "\\b");
    // 将匹配到的子串替换为空字符串
    obj.className = obj.className.replace(reg, "");
}

/*
 * 切换 class 属性值，如果有该属性值则删除，如果没有则添加
 * obj: 元素
 * className: class 属性值
 */
function toggleClassName(obj, className)
{
    if (hasClassName(obj, className))
    {   // 有该属性值，则删除
        removeClassName(obj, className);
    }
    else
    {   // 没有该属性值，则添加
        addClassName(obj, className);
    }
}
```

18.3 二级菜单

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>二级菜单</title>

        <style type="text/css">
            /* 省略 */
        </style>

        <script type="text/javascript">
            window.onload = function() {
                /*
                 * 当菜单 div 具有 collapsed 类名时，处于折叠状态
                 * 当菜单 div 没有 collapsed 类名时，处于展开状态
                 */

                var menuSpan = document.querySelectorAll(".menuSpan");

                // 当前打开的菜单
                var openedDiv = menuSpan[0].parentNode;

```

```
                for (var i = 0; i < menuSpan.length; i++)
                {
                    menuSpan[i].onclick = function() {
                        // this 是 span, this.parentNode 是 div
                        var parentDiv = this.parentNode;
                        // 切换菜单的显示状态
                        toggleMenu(parentDiv);
                        // 关闭之前打开的菜单
                        if (openedDiv != parentDiv &&
!hasClassName(openedDiv, "collapsed"))
                        {
                            /**
                             * 这里本来应该使用 addClassName()
                             * 但是为了统一动画处理效果，改为 toggleClassName()
                             * 这样就可以使用 toggleMenu() 统一处理
                             * 为此，需要增加判断条件，保证只增加 class 属性值
                             * 所以，只有当 collapsed 类名不存在时才执行
                             */
                            toggleMenu(openedDiv);
                        }
                        // 修改当前打开的菜单
                        openedDiv = parentDiv;
                    };
                }

                /**
                 * 切换菜单的折叠和打开状态
                 * obj: 要切换的菜单对象
                 */
                function toggleMenu(obj)
                {
                    // 切换前的高度
                    var begin = obj.offsetHeight;
                    // 切换菜单显示状态
                    // toggleClassName() 函数定义在 18.2 节
                    toggleClassName(obj, "collapsed");
                    // 切换后的高度
                    var end = obj.offsetHeight;
                    // 将高度重置为 begin
                    obj.style.height = begin + "px";
                    // 执行动画
                    // movie() 函数定义在 16.8.2 节
                    movie(obj, "height", end, 10, function(){
                        // 动画执行完毕后，清空行内样式
                        obj.style.height = "";
                    });
                }
            };
        </script>
    </head>
    <body>
        <div id="my_menu" class="sdmenu">
            <div>
                <span class="menuSpan">在线工具</span>
                <a href="#">图像优化</a>
                <a href="#">收藏夹图标生成器</a>
                <a href="#">邮件</a>
                <a href="#">htaccess密码</a>
```

```
                <a href="#">梯度图像</a>
                <a href="#">按钮生成器</a>
            </div>
            <div class="collapsed">
                <span class="menuSpan">支持我们</span>
                <a href="#">推荐我们</a>
                <a href="#">链接我们</a>
                <a href="#">网络资源</a>
            </div>
            <div class="collapsed">
                <span class="menuSpan">合作伙伴</span>
                <a href="#">JavaScript工具包</a>
                <a href="#">CSS驱动</a>
                <a href="#">CodingForums</a>
                <a href="#">CSS例子</a>
            </div>
            <div class="collapsed">
                <span class="menuSpan">测试电流</span>
                <a href="#">Current or not</a>
                <a href="#">Current or not</a>
                <a href="#">Current or not</a>
                <a href="#">Current or not</a>
            </div>
        </div>
    </body>
</html>
```