

My Note (JS)

1.变量

▼ var

定义的变量会成为包含它的函数的局部变量，变量将在函数退出时销毁。在函数内定义变量时省略var操作符可以创建全局变量。var具有变量提升。

▼ let

声明的范围是块作用域，无变量提升。

▼ const

与let行为相似，唯一不同是变量不可修改。

2.for-in、for-of、with语句

▼ for-in

- 迭代语句，用于枚举对象中的非符号键属性

for(property in expression) statement

▼ for-of

- 迭代语句，用于遍历可迭代对象的元素

for(property of expression) statement

```

Object.prototype.objCustom = function() {};
Array.prototype.arrCustom = function() {};

let iterable = [3, 5, 7];
iterable.foo = 'hello';

for (let i in iterable) {
  console.log(i); // logs 0, 1, 2, "foo", "arrCustom", "objCustom" // 以任意顺序迭代对象的可枚举属性
}

for (let i in iterable) {
  if (iterable.hasOwnProperty(i)) {
    console.log(i); // logs 0, 1, 2, "foo"
  }
}

for (let i of iterable) {
  console.log(i); // logs 3, 5, 7 //遍历可迭代对象定义要迭代的数据
}

```

▼ with

- 将代码作用域设置为特定的对象

with(expression) statement;

eg:let qs = location.search.substring(1);

let hostName = location.hostname;

let url = location.href;

with(location){

let qs = search.substring(1);

let hostName = hostname;

let url = href;

}

3.变量、作用域与内存

▼ 原始值与引用值

1. 创建一个变量然后给它赋值。原始值是最简单的数据，而对引用值，可以随时添加、修改和删除属性和方法。

2. 变量复制时，原始值会被复制到新的变量位置，两个变量相互独立，互不干扰。引用值复制则两个变量指向同一个对象。

▼ typeof与instanceof

1. typeof用于判断一个变量是否为原始类型，如果值是对象或null，则typeof返回“object”
2. instanceof用于变量是否是引用类型，是返回“true”，否返回“false”

eg.console.log(person instanceof Object);

A instanceof B：A是否是B这个构造函数的实例对象

A：实例对象

B：构造函数

如果函数B的显式原型在A对象的原型链上，返回true，否则返回false。

▼ 垃圾回收机制

通过自动内存管理实现内存分配和闲置资源回收。基本思路：确定哪个变量不会再使用，然后释放其占用内存。该过程周期性，即垃圾回收程序每隔一段时间就自动运行。

1. 标记清理
2. 引用计数（不好，不推荐）

```
/*
垃圾回收（GC）
就像人生活的事件长了会产生垃圾一样，程序运行过程中也会产生垃圾
这些垃圾积攒过多以后，会导致程序运行的速度过慢
所以我们需要一个垃圾回收的机制，来处理程序运行过程中产生垃圾
当一个对象没有任何的变量或属性对它进行引用，此时我们将永远无法操作该对象
此时这种对象就是一个垃圾，这种对象过多会占用大量的内存空间，导致程序运行变慢
所以这种垃圾必须进行清理。
-在JS中拥有自动的垃圾回收机制，会自动将这些垃圾对象从内存中销毁，
  我们不需要也不能进行垃圾回收的操作
  我们需要做的只是要将不再使用的对象设置为null即可
*/

var obj = new Object();
//对对象进行各种操作...

obj = null;
```

▼ 内存管理

1. 解除引用：如果数据不再必要，把它设为null，从而释放其引用。

4.集合引用类型（Object等）

Array类型

▼ 1.创建数组

- (1) 使用Array：let colors = new Array(20);
- (2) 数组字面量：let colors = ["red","blue","green"]
- (3) **ES6新增：from()** 将类数组结构转换为数组实例
 of() 将一组参数转换为数组实例

▼ 2.数组索引

数组length属性不是只读的，通过修改length属性，可以从数组末尾删除或添加属性。

▼ 3.监测数组

```
if(Array.isArray(value)){  
    //操作数组  
}
```

▼ 4.迭代器方法

三个用于检索数组内容的方法：keys()、values() 和entries()

keys()返回数组索引的迭代器，values()返回数组元素的迭代器，而entries()返回索引/值对的迭代器

eg.

```
const a = ["foo","bar","baz","qux"];  
const aKeys = Array.from(a.keys());  
const aValues = Array.from(a.values());  
const aEntries = Array.from(a.entries());
```

使用ES6的解构可以非常容易地在循环中拆分键/值对：

```
const a = ["foo","bar","baz","qux"];  
for(const [idx,element] of a.entries()) {  
    alert(idx);  
    alert(element);  
}
```

```
}
```

▼ 5. 复制和填充方法

- 批量复制方法fill()：向一个已有的数组中插入全部或部分相同的值

eg.

```
const zeroes = [0,0,0,0,0];
```

```
zeroes.fill(7,1,3); //用7填充所以大于等于1小于4的元素
```

```
zeroes.fill(7,-4,-1); //用7填充所以大于等于（-4+5=1）小于（-1+5=4）的元素
```

fill()静默忽略超出数组边界、零长度及方向相反的索引范围。

- copyWithin()按照指定范围浅复制数组中的部分内容，然后将它们插入到指定索引开始的位置。

eg.

```
let ints,reset = () => ints = [0,1,2,3,4,5,6,7,8,9];
```

```
reset();
```

```
//从ints中复制索引0开始到索引3结束的内容，插入到索引4开始的位置
```

```
ints.copyWithin(4,0,3);
```

▼ 6. 栈方法和队列方法

(1) 栈：后进先出（LIFO） push()和pop()方法

(2) 队列：先进先出（FIFO）

队列在列表末尾添加数据，但从列表开头获取数据。 shift()和push()

队列在列表开头添加数据，但从列表末尾获取数据。 unshift()和pop()

```
//创建一个数组
var arr = ["孙悟空","猪八戒","沙和尚"];

/*
push()
- 该方法可以在数组的末尾添加一个或多个元素，并返回数组的新长度
- 可以将要添加的元素作为方法的参数传递，这样这些元素将会自动添加到数组的末尾
- 该方法会将数组新的长度作为返回值返回
*/
var result = arr.push("唐僧","蜘蛛精","白骨精");
console.log(arr);    //[ '孙悟空', '猪八戒', '沙和尚', '唐僧', '蜘蛛精', '白骨精' ]
console.log("result = "+result);    //result = 6

/*
pop()
*/
```

```

- 该方法可以删除数组的最后一个元素，并将删除的元素作为返回值返回
*/
var result = arr.pop();
console.log(arr);    //[ '孙悟空', '猪八戒', '沙和尚', '唐僧', '蜘蛛精' ]
console.log("result = "+result);    //result = 白骨精

/*
unshift()
- 向数组开头添加一个或多个元素，并返回新的数组长度
- 向前插入元素以后，其他的元素索引会依次调整
*/
var result = arr.unshift("牛魔王","二郎神");
console.log(arr);    //[ '牛魔王', '二郎神', '孙悟空', '猪八戒', '沙和尚', '唐僧', '蜘蛛精' ]
console.log("result = "+result);    //result = 7

/*
shift()
- 可以删除数组的第一个元素，并将被删除的元素作为返回值返回
*/
var result = arr.shift();
console.log(arr);    //[ '二郎神', '孙悟空', '猪八戒', '沙和尚', '唐僧', '蜘蛛精' ]
console.log("result = "+result);    //result = 牛魔王

```

▼ 7.排序方法

- (1) reverse() 将数组元素反向排列
- (2) sort() 按照**升序**重新排列数组元素 可以接收一个**比较函数**

```

function compare(value1, value2) {
    if (value1 < value2) {
        return -1;
    } else if (value1 > value2) {
        return 1;
    } else {
        return 0;
    }
}

let values = [0, 1, 5, 10, 15];
values.sort(compare);
console.log(values);    //[ 0, 1, 5, 10, 15 ]

```

补充：

sort()

可以对一个数组中的内容进行排序，默认是按照Unicode编码进行排序调用以后，会直接修改原数组。可以自己指定排序的规则，需要一个回调函数作为参数：

我们可以自己来指定排序的规则我们可以在`sort()`添加一个回调函数，来指定排序规则，回调函数中需要定义两个形参,浏览器将会分别使用数组中的元素作为实参去调用回调函数使用哪个元素调用不确定，但是肯定的是在数组中a一定在b前边

- 浏览器会根据回调函数的返回值来决定元素的顺序，如果返回一个大于0的值，则元素会交换位置，如果返回一个小于0的值，则元素位置不变，如果返回一个0，则认为两个元素相等，也不交换位置。
- 如果需要升序排列，则返回 $a-b$ ，如果需要降序排列，则返回 $b-a$

```
function(a,b){  
    //升序排列  
    //return a-b;  
    //降序排列  
    return b-a;  
}
```

▼ 对于谷歌浏览器分析相反，结果相同

```
/*  
sort()  
- 可以用来对数组中的元素进行排序  
- 也会影响原数组，默认会按照Unicode编码进行排序  
*/  
arr = ["s", "a", "d", "t", "d", "h", "e", "r"];  
arr.sort();  
console.log(arr); //['a', 'd', 'd', 'e', 'h', 'r', 's', 't']  
  
/*  
即使对于纯数字的数组，使用sort()排序时，也会按照Unicode编码来排序  
所以对数字进行排序时，可能会得到错误的结果。  
  
我们可以自己来指定排序的规则  
    我们可以在sort()中添加一个回调函数，来指定排序规则  
        回调函数中需要定义两个形参。  
        浏览器将会分别使用数组中的元素作为实参去调用回调函数  
        使用哪个元素调用不确定，但是肯定的是谷歌浏览器在数组中a一定在b后面  
- 谷歌浏览器会根据回调函数的返回值来决定元素的顺序，  
    如果返回一个大于等于0的值，则元素位置不变  
    如果返回一个小于0的值，则元素会交换位置  
- 如果需要升序排列，则返回 $a-b$   
- 如果需要降序排列，则返回 $b-a$   
  
*/  
arr = [1,24,4,3,6];  
arr.sort();  
console.log(arr); //[1, 24, 3, 4, 6]
```

```
arr = [5,4];
//升序排列，前小后大
arr.sort(function(a,b){
    console.log("a = "+a);    //a = 4
    console.log("b = "+b);    //b = 5
    return a-b;
});
console.log(arr);    //[4,5]
```

▼ 8.操作方法

(1) concat () 在现有数组全部元素基础上创建一个新数组

```
let colors = ["red", "green", "blue"];
let colors2 = colors.concat("yellow", ["black", "brown"]);
console.log(colors);    //[ 'red', 'green', 'blue' ]
console.log(colors2);    //[ 'red', 'green', 'blue', 'yellow', 'black', 'brown' ]
```

Symbol.isConcatSeparate能阻止concat()打平参数数组。设置为true可以强制打平类数组对象

```
//强制打平
let colors = ["red", "green", "blue"];
let newColors = ["black", "brown"];
newColors[Symbol.isConcatSpreadable] = true;
let colors2 = colors.concat("yellow", newColors);
console.log(colors2);    //[ 'red', 'green', 'blue', 'yellow', 'black', 'brown' ]
```

(2) join()

- 该方法可以将数组转换为一个字符串
 - 该方法不会对原数组产生影响，而是将转换后的字符串作为结果返回
 - **在join()中可以指定一个字符串作为参数，这个字符串将会成为数组中元素的连接符**
- 如果不指定连接符，则默认使用","作为连接符；如果不想要连接符就传一个空串

```
var arr = ["唐僧", "白骨精", "蜘蛛精"];
result = arr.join();
console.log(result);    //"唐僧,白骨精,蜘蛛精"
console.log(typeof result);    //string
console.log(arr);    //['唐僧', '白骨精', '蜘蛛精']

result = arr.join("hello");
console.log(result);    //唐僧hello白骨精hello蜘蛛精
```



```
result = arr.join("");
console.log(result);    //唐僧白骨精蜘蛛精
```

(3) **slice ()** 返回从开始索引到结束索引对应的所有元素，其中**不包含结束索引对应的元素**。

```
let colors = ["red", "green", "blue", "black", "brown"];
let colors2 = colors.slice(1);
let colors3 = colors.slice(1, 4);
console.log(colors2);    //[ 'green', 'blue', 'black', 'brown' ]
console.log(colors3);    //[ 'green', 'blue', 'black' ]
```

补充：

slice(start,[end])

可以从一个数组中截取指定的元素

该方法不会影响原数组，而是将截取到的内容封装为一个新的数组并返回

参数：

1. 截取开始位置的索引（包括开始位置）
2. 截取结束位置的索引（不包括结束位置）

第二个参数可以省略不写，如果不写则一直截取到最后

参数可以传递一个负值，如果是负值，则从后往前数

- 1 倒数第一个
- 2 倒数第二个

(4) **splice ()** 在数组中间插入元素

删除：2个参数，要删除的第一个元素的位置和要删除的元素数量

插入：3个参数，开始位置，0（要删除元素数量），要插入的元素

替换：3个参数，开始位置，要替换元素数量，要插入元素

```

let colors = ["blue", "purple", "black"];
let removed = colors.splice(0, 1);
console.log(colors);    //[ 'purple', 'black' ]
console.log(removed);   //[ 'blue' ]

removed = colors.splice(1, 0, "yellow", "orange");
console.log(colors);    //[ 'purple', 'yellow', 'orange', 'black' ]
console.log(removed);   //[ ]

removed = colors.splice(1, 2, "red", "pink");
console.log(colors);    //[ 'purple', 'red', 'pink', 'black' ]
console.log(removed);   //[ 'yellow', 'orange' ]

```

补充：

splice()

可以用来删除数组中指定元素，并使用新的元素替换

该方法会将删除的元素封装到新数组中返回

参数：

1. 删除开始位置的索引
2. 删除的个数
3. 三个以后，都是替换的元素，这些元素将会插入到开始位置索引的前边

▼ 9. 搜索和位置方法

(1) 严格相等

- indexOf () 从前往后，返回查找元素在数组中的位置
- lastIndexOf () 从数组末尾（最后一项）开始向前搜索，返回查找元素在数组中的位置
- includes () 从前往后，返回布尔值，表示是否至少找到与指定元素匹配的项

(2) 断言函数

- find () 返回第一个匹配的元素
- findIndex () 返回第一个匹配元素的索引

找到匹配后，这两种方法都不继续搜索

▼ 10. 迭代方法

接受三个参数：数组元素 (items)、元素索引 (index) 和数组本身(array)

- **every()** 对数组每一项都运行传入的函数，如果对每一项函数都返回true，则返回true
- **filter()** 对数组每一项都运行传入的函数，函数返回true的项目会组成数组之后返回

```
//需求2 从数组中返回偶数的元素
const arr = [1, 6, 9, 10, 100, 25];
// const result = arr.filter(function (item) {
//     if (item % 2 === 0) {
//         return true;
//     }
//     else {
//         return false;
//     }
// });
// console.log(result);

// const result = arr.filter(item => {
//     if (item % 2 === 0) {
//         return true;
//     }
//     else {
//         return false;
//     }
// });
// console.log(result);

//或者简化
const result = arr.filter(item => item % 2 === 0);
```

- **forEach()** 对数组每一项都运行传入的函数，没有返回值

补充：

使用forEach()方法来遍历数组（不兼容IE8）

```
数组.forEach(function(value , index , obj){
});
```

forEach()方法需要一个回调函数作为参数，

数组中有几个元素，回调函数就会被调用几次，

每次调用时，都会将遍历到的信息以实参的形式传递进来，

我们可以定义形参来获取这些信息。

value:正在遍历的元素

index:正在遍历元素的索引

obj:被遍历对象

- **map()** 对数组每一项都运行传入的函数，返回由每次函数调用的结果构成的数组

```
let numbers = [1, 2, 3, 4, 5, 4, 3, 2, 1];
let mapResult = numbers.map((items, index, array) => items * 2);
console.log(mapResult); // [2, 4, 6, 8, 10, 8, 6, 4, 2]
```

- **some()** 对数组每一项都运行传入的函数，如果有一项返回true,则返回true

▼ 11. 归并方法

接受4个参数：上一个归并值(prev)、当前值(cur)、当前项的索引(index)和数组本身(array)

- **reduce()** 从数组第一项开始遍历到最后一项
- **reduceRight()** 从最后一项遍历到第一项

函数返回任何值都会作为下一次调用同一函数的第一个参数。

```
let values = [1, 2, 3, 4, 5];
let sum = values.reduce((prev, cur, index, array) => prev + cur);
console.log(sum); // 15
```

Map



Set



Date类型

▼ Date对象与方法

- Date对象
 - 在JS中使用Date对象来表示一个时间

```
//创建一个Date对象
//如果直接使用构造函数创建一个Date对象，则会封装为当前代码执行的时间
var d = new Date();
console.log(d);    //Wed Jan 19 2022 10:24:39 GMT+0800 (中国标准时间)

//创建一个指定的时间对象
//需要在构造函数中传递一个表示时间的字符串作为参数
//日期的格式：  月份/日/年  时:分:秒
var d2 = new Date("12/26/1997 11:10:30"); //Fri Dec 26 1997 11:10:30 GMT+0800 (中国标准时间)
console.log(d2);
```

- Date方法：

getDate() 获取当前对象日期是几日(0 ~ 31)

getDay() 获取当前对象是周几(0 ~ 6) 0:周日

getMonth() 获取当前对象的月份(0 ~ 11) 0：1月

getFullYear() 获取当前对象的年份

getTime()

- 获取当前日期对象的时间戳

- 时间戳，指的是从格林威治标准时间的1970年1月1日，0时0分0秒到当前日期所花费的毫秒数

- 计算机底层在保存时间时使用都是时间戳

now() 获取当前的时间戳，利用时间戳来测试代码的执行性能

```
var d2 = new Date("12/26/1997 11:10:30");
var date = d2.getDate();
var day = d2.getDay();
var month = d2.getMonth();
var year = d2.getFullYear();
var time = d2.getTime();
console.log("date = "+date);    //date = 26
console.log("day = "+day);    //day = 5
console.log("month = "+month);    //month = 11
console.log("year = "+year);    //year = 1997
console.log("time = "+time);    //time = 883105830000

// var d3 = new Date("1/1/1970 0:0:0");
// time = d3.getTime();
// console.log(time);    //-28800000 北京时间有8h时差

/*
获取当前的时间戳
利用时间戳来测试代码的执行性能
*/
var start = Date.now();
for(let i=0;i<100;i++){
```

```
console.log(i);  
}  
var end = Date.now();  
console.log(end - start); //1 (单位：毫秒)
```

RegExp类型

▼ 正则表达式

正则表达式(regular expression)描述了一种**字符串匹配的模式** (pattern)，可以用来检查一个串是否含有某种子串、将匹配的子串替换或者从某个串中取出符合某个条件的子串等。

1、创建正则表达式对象

- 语法：

`var 变量 = new RegExp("正则表达式","匹配模式");`

使用typeof检查正则对象，会返回Object

这个正则表达式可以用来检查一个字符串中是否含有a

在构造函数中可以传递一个匹配模式作为第二个参数，

可以是

i 忽略大小写

g 全局匹配模式

2、正则表达式方法

3、使用字面量创建正则表达式

- 语法：

`var 变量 = new RegExp("正则表达式","匹配模式");`

使用typeof检查正则对象，会返回Object

这个正则表达式可以用来检查一个字符串中是否含有a

在构造函数中可以传递一个匹配模式作为第二个参数，

可以是

i 忽略大小写

g 全局匹配模式

2、正则表达式方法

- **test()**

-使用这个方法可以用来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

```
var reg = new RegExp("a","i");
var str = "a";
var result = reg.test(str);
console.log(reg);    // /a/i
console.log(typeof reg); //object
console.log(result); //true
console.log(reg.test("abc"));    //true
console.log(reg.test("csferg")); //false
console.log(reg.test("Asferg")); //true
```

3、使用字面量创建正则表达式

- 语法：

var 变量 = /正则表达式/匹配模式

使用字面量的方式更加简单使用构造函数创建更加灵活

使用 | 表示或者的意思

[]里的内容也是或的关系

[ab] == a|b

[a-z] 任意小写字母

[A-Z] 任意大写字母

[A-z] 任意字母

[0-9] 任意数字

[^] 除了

```
var reg = /a/i;
console.log(reg.test("abc"));    //true
console.log(typeof reg); //object

//创建一个正则表达式，检查一个字符串中是否有a或b
reg = /a|b|c/;
console.log(reg.test("bcd"));    //true

//创建一个正则表达式检查一个字符串中是否含有字母
reg = /[A-z]/;
console.log(reg.test("d"));    //true

//检查一个字符串中是否含有abc 或 adc 或 aec
reg = /a[bde]c/;
```

```

console.log(reg.test("aec")); //true

//除了[ ^ ]
reg = /^[^ab]/;
console.log(reg.test("abc")); //true
console.log(reg.test("ab")); //false

```

▼ 正则表达式语法

1、量词/开头结尾

- 通过量词可以设置一个内容出现的次数
 - {n} 正好出现n次
 - {m,n} 出现m~n次
 - {m,} 出现m次以上
 - + 至少一个，相当于{1,}
 - * 0个或多个，相当于{0,}
 - ? 0个或1个，相当于{0,1}

```

var reg = /a{3}/;
console.log(reg.test("aaabc")); //true
reg = /(ab){3}c/;
console.log(reg.test("abababc")); //true
reg = /ab{1,3}c/;
console.log(reg.test("abc")); //true
console.log(reg.test("abbc")); //true
console.log(reg.test("abbbc")); //true
reg = /ab{3,}c/;
console.log(reg.test("abc")); //false
console.log(reg.test("abbc")); //false
console.log(reg.test("abbbc")); //true
console.log(reg.test("abbbbc")); //true
reg = /ab+c/;
console.log(reg.test("abc")); //true
console.log(reg.test("abbc")); //true
console.log(reg.test("abbbc")); //true
console.log(reg.test("abbbbc")); //true
reg = /ab*c/;
console.log(reg.test("ac")); //true
console.log(reg.test("abc")); //true
console.log(reg.test("abbc")); //true
console.log(reg.test("abbbc")); //true
console.log(reg.test("abbbbc")); //true
reg = /ab?c/;
console.log(reg.test("ac")); //true
console.log(reg.test("abc")); //true
console.log(reg.test("abbc")); //false

```


- 字符串开头结尾 (^和\$)

```
/*
检查一个字符串是否以a开头
*/
reg = /^a/; //匹配开头的a
console.log(reg.test("abc")); //true
console.log(reg.test("bac")); //false
reg = /a$/; //匹配结尾的a
console.log(reg.test("acdva")); //true
/*
如果在正在表达式中同时使用^ $则要求字符串必须完全符合正则表达式
*/
reg = /^a$/;
console.log(reg.test("aaa")); //false
reg = /^a|a$/; //以a开头或者以a结尾
console.log(reg.test("abc")); //true
```

2、转义符

- 在正则表达式中使用\作为转义字符

\. 来表示.

\\ 表示\

注意：使用构造函数时，由于它的参数是一个字符串，而\是字符串的转义字符，

如果要使用\则需要使用\\来代替

\w 任意字母、数字、_ [A-z0-9_]

\W 除了字母、数字、_ [^A-z0-9_]

\d 任意数字 [0-9]

\D 除了数字 [^0-9]

\s 空格

\S 除了空格

\b 单词边界

\B 除了单词边界

```
var reg = ./; //匹配所有字符
console.log(reg.test("abcdfs")); //true
reg = /\../;
console.log(reg.test("ab.cd fs")); //true
console.log(reg.test("abcd fs")); //false
reg = /\./;
console.log(reg.test("b\\.")); //true
console.log("b\\."); // b\
reg = new RegExp("\\.");
```

```

console.log(reg.test("aarpewv")); //true
console.log(reg); // /.
reg = new RegExp("\\.");
console.log(reg.test("aarpewv")); //false
console.log(reg.test("aarp.ewv")); //true
console.log(reg); // /\./

/*
创建一个正则表达式检查一个字符串中是否含有单词child
*/
reg = /child/;
console.log(reg.test("hello children")); //true
reg = /\bchild\b/;
console.log(reg.test("hello children")); //false

//接收一个用户的输入
// var str = prompt("请输入你的用户名：");
str = "      he llo      ";
console.log(str);
//去除字符串中前后的空格
// 去除空格就是使用""来替换空格（但是前、后、中间的空格都被去除了）
// str = str.replace(/\s/g, "");
// console.log(str);
//去除开头的空格
// str = str.replace(/^\s*/, "");
// console.log(str);
//去除结尾的空格
// str = str.replace(/\s$/, "");
// console.log(str);
//去除开头和结尾的空格
str = str.replace(/^\s*|\s$/g, "");
console.log(str);

```

▼ 邮件/电话正则表达式

1、电话正则

- 手机号规则：11位

1、以1开头

2、第二位3-9任意数字

3、三位以后任意数字9个

`^1 [3-9] [0-9]{9}$`

```

var phoneStr = "18196614333";
var phoneReg = /^1[3-9][0-9]{9}$/;
console.log(phoneReg.test(phoneStr)); //true

```

2、邮件正则

- 电子邮件

hello.nihao@abc.com.cn

hello 任意字母、数字、下划线

.nihao .任意字母、数字、下划线(可有可无)

@

abc 任意字母、数字（长度不限）

.com .任意字母（2-5位）

.cn .任意字母（2-5位）

`\w{3,} (\.\w+)* @[A-z0-9]+ (\.[A-z]{2,5}){1,2}`

```
var emailReg = /^\\w{3,}(\\.\\w+)*@[A-z0-9]+(\\.\\[A-z]{2,5}){1,2}$/;
var email = "abc@abc.com";
console.log(emailReg.test(email)); //true
```

Function类型

▼ call()和apply()（函数对象的方法）

- call()和apply() **改变this指向**
 - 这两个方法都是函数对象的方法，需要通过函数对象来调用
 - 当对函数调用call()和apply()都会调用函数执行
 - **在调用call()和apply()可以将一个对象指定为第一个参数**
此时这个对象将会成为函数执行时的this
 - call()方法可以将实参在对象之后一次传递
 - apply()方法需要将实参封装到一个数组中统一传递
 - this情况
 - 1、以函数形式调用时，this永远都是window
 - 2、以方法的形式调用时，this是调用方法的对象
 - 3、以构造函数的形式调用时，this是新创建的对象
 - 4、以call和apply调用时，this是指定的那个对象

```

// function fun(){
//     alert(this);
// }
//     fun();
//     console.log(fun);
//     console.log(typeof fun); //function
//     fun.call();
//     fun.apply();

// fun(); // [object Window]
// var obj = {};
// fun.call(obj); // [object Object]
// fun.apply(obj); // [object Object]

// function fun(){
//     alert(this.name);
// }
// var obj = {
//     name: 'obj',
//     sayName: function(){
//         alert(this.name);
//     }
// };
// var obj2 = {name: 'obj2'};
// fun.call(obj); //obj
// fun.apply(obj2); //obj2
// obj.sayName(); //obj
// obj.sayName.apply(obj2); //obj2

function fun(a,b){
    console.log("a = "+a);
    console.log("b = "+b);
}
var obj = {
    name: 'obj',
    sayName: function(){
        alert(this.name);
    }
};
var obj2 = {name: 'obj2'};
fun.call(obj, 2, 3); //a=2 b=3
fun.apply(obj, [2, 3]); //a=2 b=3

```

高级

JS考点

call()和apply()方法的区别和用法详解

▼ arguments

- 在调用函数时，浏览器每次都会传递进两个隐含的参数：

1、函数的上下文对象this

2、封装实参的对象 arguments

- arguments是一个类数组对象，它也可以通过索引来操作数据，也可以获取长度
 - 在调用函数时，我们所传的实参都会在arguments中保存
 - arguments.length可以用来获取实参的长度
 - 我们即使不定义实参，也可以通过arguments来使用实参，只不过比较麻烦
- arguments[0] 表示第一个实参
arguments[1] 表示第二个实参...
- 它里面有一个属性叫做callee
- 这个属性对应一个函数对象，就是正在指向的函数的对象

```
function fun(a,b){
  console.log(arguments);
  console.log(arguments instanceof Array); //false
  console.log(Array.isArray(arguments)); //false
  console.log(arguments.length); //2
  console.log(arguments[0]); // "hello"
  console.log(arguments.callee); //函数对象
}

fun("hello",true);
```

基本包装类型

▼ String()、Number()、Boolean()

基本数据类型

String Null Number Boolean Undefined

引用数据类型

Object

在JS中为我们提供了三个包装类，通过这三个基本包装类可以将基本数据类型的数据转换为对象

String()

- 可以将基本数据类型字符串转换为String对象

Number()

- 可以将基本数据类型的数组转换为Number对象

Boolean()

- 可以将基本数据类型的布尔值转换为Boolean对象

但是注意，我们在实际应用中不会使用基本数据类型的对象

如果使用基本数据类型的对象，在做一些比较和判断时会得到一些不可预期的结果

```
var a = 123;
console.log(typeof a); //number

//创建一个Number类型的对象
var num = new Number(3);
var num2 = new Number(3);
var str = new String("hello");
var bool = new Boolean(true);
var bool2 = true;
console.log(num);
console.log(typeof num); //object
console.log(typeof str); //object
console.log(typeof bool); //object
console.log(num == num2); //false 比较两个对象的内存地址
console.log(bool == bool2); //true
console.log(bool === bool2); //false

//向num中添加一个属性
num.hello = "alsjfaeg";
console.log(num.hello);
```

方法和属性只能添加给对象，不能添加给基本数据类型

但我们对一些基本数据类型的值去调用属性和方法时，

浏览器会临时使用包装类将其转换为对象，然后再调用对象的属性和方法

调用完以后，再将其转换为基本数据类型（临时转，其实没有赋值）

实际上console.log(s.hello);放在s.hello = "你好！";后面的任何位置，都返回undefined，因为就是临时转成对象，一旦转换完，在语句结束后又会回到基本数据类型。

```
var s = 123;
s = s.toString();
s.hello = "你好!";
console.log(s); //123
console.log(typeof s); //string
console.log(s.hello); //undefined 临时赋值hello属性的对象在复制完成后已经被销毁了，
//打印时有一次调用属性，因此再一次转换为对象，由于此时的对象hello属性没赋值，因此返回undefined
```

▼ 字符串的相关方法（String对象）

常用属性：length 可以用来获取字符串的长度

常用方法：

1、charAt()

可以返回字符串中指定位置的字符，根据索引获取指定的字符

```
//创建一个字符串
var str = "hello Sophie";
var result = str.charAt(0);
console.log(str);
console.log("result = "+result);
```

2、charCodeAt()

获取指定位置字符的字符编码（Unicode编码）

```
//创建一个字符串
var str = "hello Sophie";
var result = str.charCodeAt(0);
console.log(str);
console.log("result = "+result);
```

3、concat()

可以用来连接两个或多个字符串，作用和+一样

```
//创建一个字符串
var str = "hello Sophie";
result = str.concat("下次再见");
console.log(str);
console.log("result = "+result);
```

4、indexOf()

- 该方法可以检索一个字符串中是否含有指定内容
 - 如果字符串中含有该内容，则会返回其第一次出现的索引
 - 如果没有找到指定的内容，则返回-1
 - 可以指定一个第二个参数，指定开始查找的位置

5、lastIndexOf()

- 该方法的用法和indexOf()一样，不同的是indexOf是从前往后找而lastIndexOf是从后往前找
 - 也可以指定开始查找的位置

```
//创建一个字符串
var str = "hello Sophie";
result = str.indexOf('h',10);
result1 = str.lastIndexOf('h',1);
console.log(result);    //-1
console.log(result1);   //0
```

6、slice()

- 可以从字符串中截取指定的内容
 - 不会影响原字符串，而是将截取到的内容返回
 - 参数：
 - 第一个，开始位置的索引（包括开始位置）
 - 第二个，结束位置的索引（不包括结束位置）
 - 如果省略第二个参数，则会截取到后边所有的
 - 也可以传递一个负数作为参数，负数的话将会从后边计算

```
str = "abcdefghijklmn";
result = str.slice(1,4);
result = str.slice(1);    //bcdefghijklmn
result = str.slice(1,-1); //bcdefghijklm
console.log(result);
```

7、substring()

- 可以用来截取一个字符串，和slice()类似
 - 参数：
 - 第一个：开始截取位置的索引（包括开始位置）
 - 第二个：结束位置的索引（不包括结束位置）
 - 不同的是这个方法不能接受负数作为参数，
 - 如果传递了一个负值，则默认使用0
 - 而且他会自动调整参数的位置，如果第二个参数小于第一个，则自动交换

```
str = "abcdefghijklmn";
result = str.substring(0,2); //ab
result = str.substring(1,-1); //a 先把-1变0，再和1交换位置，取第一个字符
console.log(result);
```

8、split()

- 可以将一个字符串拆分为一个数组
 - 参数：
 - 需要一个字符串作为参数，将会根据该字符串去拆分数组
 - 如果传递一个空串作为参数，则会将每个字符都拆分为数组中的一个元素

```
str = "abc,bcd,efg,hij";
result = str.split(',');
console.log(result);    //[ 'abc', 'bcd', 'efg', 'hij' ]
console.log(typeof result);    //Object
console.log(Array.isArray(result));    //true

str = "abcdefghjkos";
result = str.split("");
console.log(result);    //[ 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'j', 'k', 'o', 's' ]
```

9、toUpperCase() -将一个字符串转换为大写并返回

10、toLowerCase() -将一个字符串转换为小写并返回

```
str = 'abcdefg';
result = str.toUpperCase();
console.log(result);    //ABCDEFGF
result = result.toLowerCase();
console.log(result);    //abcdefg
```

▼ 字符串和正则相关的方法

1、split()

- 可以将一个字符串拆分为一个数组
 - 方法中可以传递一个正则表达式作为参数，这样方法将会根据正则表达式去拆分字符串
 - 这个方法即使不指定全局匹配，也会全都拆分

```
var str = "1a2b3c4d5e6f7";
/*
根据任意字母来将字符串拆分
*/
var result = str.split(/[A-z]/);
console.log(result);    //[ '1', '2', '3', '4', '5', '6', '7' ]
```

2、search()

- 可以搜索字符串中是否含有指定内容
 - 如果搜索到指定内容，则会返回第一次出现的索引，如果没有搜索到返回-1
 - 它可以接受到一个正则表达式作为参数，然后会根据正则表达式去检索字符串
 - search()只会查找第一个，即使设置全局匹配也没用

```
str = "Hello adc aec abc";
result = str.search("abc"); //14
result = str.search(/a[bde]c/); //6
```

3、match()

- 可以根据正则表达式，从一个字符串中将符合条件的内容提取出来
 - 默认情况下我们的match只会找到第一个符合要求的内容，找到以后就会停止检索
 - 我们可以设置正则表达式为全局匹配模式，这样就会匹配到所有的内容
 - 可以为一个正则表达式设置多个匹配模式，且顺序无所谓
 - match() 会将匹配到的内容封装到一个数组中返回，即使只查询到一个结果

```
str = "1a2b3c4d5e6f7A8B8C";
result = str.match(/[A-z]/gi);
console.log(result); //['a', 'b', 'c', 'd', 'e', 'f', 'A', 'B', 'C']
console.log(typeof result); //Object
console.log(Array.isArray(result)); //true
```

4、replace()

- 可以将字符串中指定内容替换为新的内容
 - 参数：
 - 1、被替换的内容，可以接受一个正则表达式作为参数
 - 2、新的内容
 - 默认只会替换第一个

```
str = "1a2b3c4d5e6f7A8B8C";
result = str.replace(/a/gi, "@_@"); //1@_@2b3c4d5e6f7@_@8B8C
result = str.replace(/[A-z]/gi, ""); //123456788
```

， 5.迭代器和生成器

▼ 迭代器

迭代器是一个定义了序列并可能返回一个值直到它迭代结束的对象。更具体的说，它是通过`next()`方法实现迭代器协议的任意对象。`next()`是一个返回两个属性的对象，这两个属性是：

- (1) `value`，序列中的下一个值
- (2) `done`，这个值如果已经迭代到序列中的最后一个值，则为`true`

```
function makeRangeIterator(start = 0, end = Infinity, step = 1){
  let nextIndex = start;
  let iterationCount = 0;
  const rangeIterator = {
    next: function(){
      let result;
      if(nextIndex < end){
        result = {
          value: nextIndex,
          done: false
        };
        nextIndex += step;
        iterationCount++;
        return result;
      }
      return {value: iterationCount, done: true};
    }
  };
  return rangeIterator;
}

let it = makeRangeIterator(1, 10, 2);
let result = it.next();
while(!result.done){
  console.log(result.value);
  result = it.next();
}
```

▼ 生成器

定义一个执行不连续的简单函数的迭代器。生成器Generator使用`function*`创建。初始化的时候，生成器函数会返回一个被称作为Generator的迭代器来替代执行代码。当一个值调用生成器的`next()`方法时，生成器函数将会执行直到遇到`yield`关键字。

function* name([param[,param[, ...param]]]){statements}

当这个迭代器的next()方法首次或者继续调用时，内部语句会执行到第一个出现yield的位置位置，yield后紧跟迭代器要返回的值。或者，如果使用的是yield*，则表示将执行权移交给另一个生成器函数，而当前生成器暂停执行。

其内部的next()方法返回一个对象，这个对象包含两个属性：

value：表示本次yield表达式的返回值

done：Boolean，表示生成器后续是否还有yield语句，即生成器函数是否已经执行完毕并返回。

调用next方法是，如果传入参数，那么这参数会传给上一条执行的yield语句左边的变量

```
function* sendProp(){
  yield 10;
  x = yield 'foo';
  yield x;
}
var obj = sendProp();
console.log(obj.next());
//Output: {value: 10, done: false}
console.log(obj.next());
//Output: {value: "foo", done: false}
console.log(obj.next(100));
//Output: {value: 100, done: false}
console.log(obj.next());
//Output: {value: undefined, done: true}
```

不过，当在生成器函数中显式return时，会导致生成器立即变为完成状态，即调用next()方法返回的对象的done为true。若return后面跟了一个值，那么这个值会作为当前调用的next()方法返回的value值。

▼ yield

yield关键字是生成器函数执行暂停，yield后面的表达式的值返回给生成器的调用者。实际上，yield关键字返回的是一个IteratorResult对象，一样也是value和done。此处的value表示的是对yield表达式求值的结果，而done是表示生成器函数是否完全完成。生成器函数在遇到yield表达式的时候，生成器的代码奖杯暂停运行，直到生成器的next()方法被调用。每次调用生成器的next()方法是，生成器都会恢复执行，直到遇到以下的某个值

yield：导致生成器再次暂停并返回生成器的新值。下一次调用next()时，在yield之后紧接的语句继续执行。

throw：用于从生成器中抛出异常。这个让生成器完全停止执行，并在调用者中继续执行，正如通常情况下抛出异常一样。

到达生成器函数的结尾。在这种情况下，生成器的执行结束，并且IteratorResult给调用者返回undefined并且done为true

到达return语句。在这种情况下，生成器的执行结束，并将IteratorResult返回给调用者，其值value是由return语句指定的，并且done为true。

如果将参数传递给生成器的next()方法，则该值将成为生成器当前yield操作返回的值。

```
function* idMaker(){
  var index = arguments[0] || 0;
  while(true){
    yield index++;
    if(index > 100){
      return;
    }
  }
}

var gen = idMaker(5);
console.log(gen.next().value); //5
console.log(gen.next().value); //6
console.log(gen.next().value); //7
console.log(gen.next().value); //8
console.log(gen.next().value); //9
```

▼ yield*

该表达式用于委托给另一个生成器Generator或可迭代对象。也就是在生成器的输出序列中插入另一个生成器的输出序列。

yield* [[expression]];

expression：返回一个可迭代对象的表达式。

(1) yield*委托给其他生成器

```
function* g1(){
  yield 2;
  yield 3;
  yield 4;
}
function* g2(){
  yield 1;
  yield* g1();
  yield 5;
}
var iterator = g2();
console.log(iterator.next());
//Output: {value: 1, done: false}
console.log(iterator.next());
//Output: {value: 2, done: false}
console.log(iterator.next());
//Output: {value: 3, done: false}
console.log(iterator.next());
//Output: {value: 4, done: false}
console.log(iterator.next());
//Output: {value: 5, done: false}
console.log(iterator.next());
//Output: {value: undefined, done: true}
```

(2) yield*委托给其他可迭代对象

```
function* g3(){
  yield* [1,2];
  yield* "34";
  yield* arguments;
}
var iterator = g3(5,6);

console.log(iterator.next());
//Output: {value: 1, done: false}
console.log(iterator.next());
//Output: {value: 2, done: false}
console.log(iterator.next());
//Output: {value: "3", done: false}
console.log(iterator.next());
//Output: {value: "4", done: false}
console.log(iterator.next());
//Output: {value: 5, done: false}
console.log(iterator.next());
//Output: {value: 6, done: false}
console.log(iterator.next());
//Output: {value: undefined, done: true}
```

6.对象、类与面向对象编程

对象

▼ 对象的分类

1.内建对象

- 由ES标准中定义的对象,在任何的ES实现中都可以使用
- 比如: Math String Number Boolean Function Object...

2.宿主对象

- 由JS的运行环境提供的对象,目前来讲主要指由浏览器提供的对象
- 比如:BOM DOM

3.自定义对象

- 由开发人员自己创建的对象

▼ 创建对象及获取\增\删属性 (new Object())

```
//创建对象
/*使用new关键字调用的函数，是构造函数constructor，
构造函数是专门用来创建对象的函数
使用typeof检查一个对象时，会返回object*/
var obj = new Object();
//console.log(obj, typeof obj);

/*
在对象中保存的值成为属性
向对象添加属性
语法：对象.属性值 = 属性值;
*/
obj.name = '孙悟空';
obj.gender = '男';
obj.age = 18;
console.log(obj);

/*
读取对象中的属性
语法：对象.属性名
如果读取对象中没有的属性，不会报错而回返回undefined
*/
console.log(obj.name);
console.log(obj.gender);
console.log(obj.age);
console.log(obj.hello); //undefined

/*
修改对象的属性值
语法：对象.属性名 = 新值;
*/
obj.name = "tom";
console.log(obj.name); //tom

/*
删除对象的属性
语法：delete 对象.属性名
*/
delete obj.name;
console.log(obj.name);
```

▼ 属性名与属性值 (in运算符检查对象中是否含有某属性)

```
var obj = new Object();

/*
向对象中添加属性
属性名：
    - 对象属性名不强制要求遵守标识符的规范, 但尽量按照标识符规范
*/
```



```
obj.name = '孙悟空';
obj.var = "hello";
console.log(obj.var); //hello

/*
如果要使用特殊的属性名，不能采用.的方式来操作，
需要使用另一种方式：
    语法：对象["属性名"] = 属性值
读取时也需要采用这种方式

使用[]这种形式取操作属性，更加灵活
在[]中可以直接传递一个变量，这样变量值是多少就会读取该属性
*/
// obj["123"] = 789;
// obj["hello"] = "你好!";
// var n = "hello";
// console.log(obj["123"]);
// console.log(obj[n]); //你好!
```

```
/*
属性值：
JS对象的属性值，可以是任意的数据类型
*/
let obj2 = new Object();
obj2.name = "猪八戒";
obj.test = obj2;
//obj.test = "hello";
// console.log(obj.test);
console.log(obj);
```

```
/*
in运算符
-通过该运算符可以检查一个对象中是否含有指定的属性
    如果有返回true，没有则返回false

-语法：
    "属性名" in 对象
*/

//检查obj中是否含有test2属性
console.log("test2" in obj);
```

```
> {name: '孙悟空', var: 'hello', test: {...}}
  name: '孙悟空'
  > test: {name: '猪八戒'}
    var: 'hello'
  > [[Prototype]]: Object
false
```

▼ 对象的方法（匿名函数使用）

对象的属性可以是一个函数，如果一个函数作为一个对象的属性保存，那么称这个函数是该对象的方法，调用函数就说成调用对象的方法（method）。但是它只是名称上的区别。

```
13     var obj = new Object();
14     obj.name = "孙悟空";
15     obj.age = 18;
16     //对象的属性值可以是任何的数据类型，也可以是一个函数
17     obj.sayName = function () {
18         |     console.log(obj.name);
19     }
20
21     var fun = function () {
22         |     console.log(obj.name);
23     }
24
25     console.log(obj.sayName); //打印的是对象的属性值，是一个函数
26     //调用方法
27     obj.sayName();
28     //调用函数
29     fun();
```

```
f () {                                     对象2.html:25
    console.log(obj.name);
}
```

```
孙悟空                                   对象2.html:18
```

```
孙悟空                                   对象2.html:22
```

还可以用对象字面量也来定义对象的方法：

```
//对象字面量定义对象方法
var obj2 = {
    name: "猪八戒",
    age: 18,
    sayName: function () {
        console.log(obj2.name);
    }
};

obj2.sayName();
```

▼ 基本数据类型和引用数据类型

栈内存

```
/*
    基本数据类型
    String Number Boolean Null Undefined Symbol
    引用数据类型
    Object
*/

/*
    对于基本数据类型，a的值变化并不会影响b的值
    JS中的变量都是保存在栈内存中存储，
    值与值之间是独立存在，修改一个变量不会影响其他的变量
*/
var a = 123;
var b = a;
a++;
console.log("a=" + a);//a=124
console.log("b=" + b);//b=123
```

(1)var a = 123;

(2)var b = a;

(3)a++;

变量名	值
b	123
a	124

```
/*
    对于引用数据类型
    对象是保存在堆内存中的，每创建一个新的对象，就会在堆内存中开辟出一个新的空间
    而变量保存的是对象的内存地址（对象引用），如果两个变量保存的是同一个对象引用，当一个
    对象的属性改变时，另一个对象的属性跟着改变
*/
var obj = new Object();
obj.name = "孙悟空";

var obj2 = obj;

//修改obj的name属性
obj.name = "Sophie";
```

```

console.log(obj.name); //Sophie
console.log(obj2.name); //Sophie

obj2 = null;
console.log(obj2);
console.log(obj);

```

堆内存

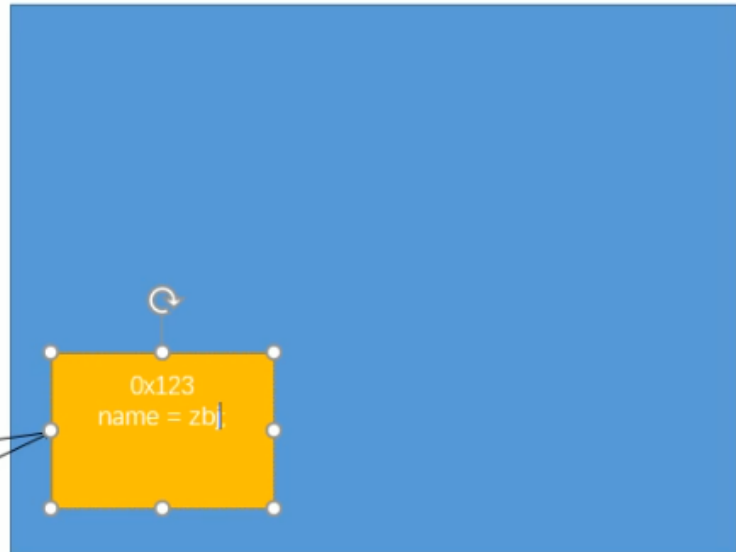
```

var obj = new Object();
obj.name = "swk";
var obj2 = obj;
obj.name = "zbj"

```

变量	值
obj2	0x123
obj	0x123

栈内存



堆内存

```

/*
对于引用数据类型
对象是保存在堆内存中的，没创建一个新的对象，就会在堆内存中开辟出一个新的空间
而变量保存的是对象的内存地址（对象引用），如果两个变量保存的是同一个对象引用，当一个通过
*/
var obj = new Object();
obj.name = "孙悟空";

var obj2 = obj;

//修改obj的name属性
obj.name = "Sophie";
console.log(obj.name); //Sophie
console.log(obj2.name); //Sophie

obj2 = null;
console.log(obj2);
console.log(obj);

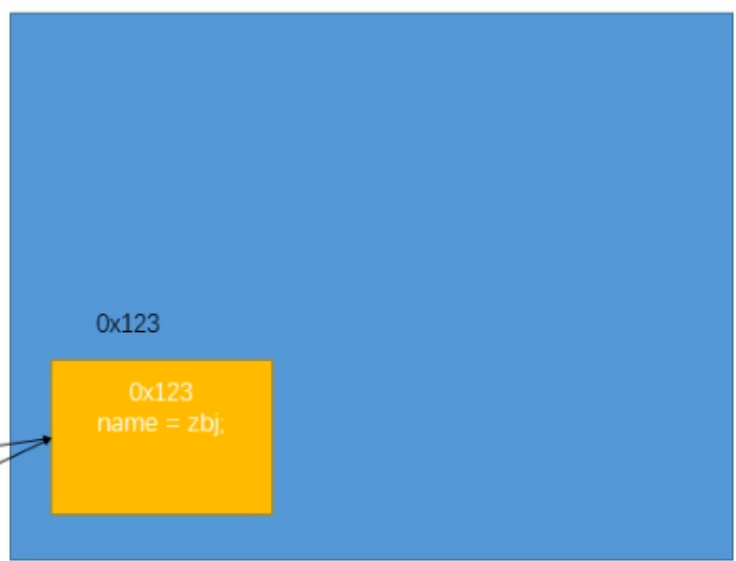
```

Sophie	基本和引用数据类型.html:44
Sophie	基本和引用数据类型.html:45
null	基本和引用数据类型.html:48
▼ Object i	基本和引用数据类型.html:49
name: "Sophie"	
▶ [[Prototype]]: Object	

```
var obj = new Object();
obj.name = "swk";
var obj2 = obj;
obj.name = "zbj"
obj2 = null;
```

变量	值
obj2	null
obj	0x123

栈内存

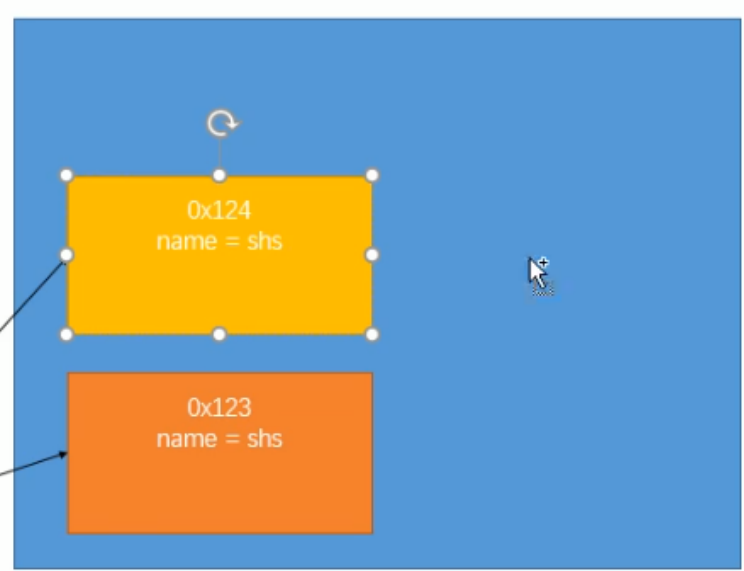


堆内存

```
var obj3 = new Object();
var obj4 = new Object();
obj3.name = "shs";
obj4.name = "shs";
```

变量	值
obj4	0x124
obj3	0x123

栈内存



堆内存

```

/*
当比较两个基本数据类型的值时，就是比较值。
而比较两个引用数据类型时，它是比较对象的内存地址，
    如果两个对象是一模一样的，但是地址不同，也会返回false
*/
var c = 10;
var d = 10;
console.log(c === d); //true
var obj3 = new Object();
var obj4 = new Object();
obj3.name = "沙和尚";
obj4.name = "沙和尚";
console.log(obj3);
console.log(obj4);
console.log(obj3 === obj4); //false

```

▼ 对象字面量（常用的创建方法）

创建对象有两种方式:new创建 和 对象字面量.开发中多使用对象字面量.

```

//创建一个对象
var obj = new Object();

//使用对象字面量来创建一个对象
var obj2 = {};
obj2.name = "Sophie";

/*
使用对象字面量，可以在创建对象时直接指定对象中的属性
语法：{属性名：属性值, 属性名：属性值....}
对象字面量的属性名可以加引号也可以不加，建议不加
如果使用一些特殊的名字，必须加引号

属性值和属性名时一组一组的名值对结构，名和名之间使用“：”连接，多个名值对之间使用“，”隔开
*/
var obj3 = {
    name: "猪八戒",
    age: 26,
    gender: "男",
    "aifu@#p-08-": "oouiou",
    test: { name: "沙和尚" }
};
console.log(obj3);

```

▼ 枚举对象中的属性（for...in...）

```

var obj = {
    name: "孙悟空",
    age: 18,
    gender: "男",

```

```

    address: "花果山"
  };

  //枚举对象中的属性
  //for...in 语句
  /*
  for(var 变量 in 对象){

  }
  for...in语句 对象中有几个属性，循环体就会执行几次
  */
  for (var n in obj) {
    console.log("属性名:" + n);
    console.log("属性值:" + obj[n]);
  }

```

属性名: name	枚举对象中的属性.html:29
属性值: 孙悟空	枚举对象中的属性.html:30
属性名: age	枚举对象中的属性.html:29
属性值: 18	枚举对象中的属性.html:30
属性名: gender	枚举对象中的属性.html:29
属性值: 男	枚举对象中的属性.html:30
属性名: address	枚举对象中的属性.html:29
属性值: 花果山	枚举对象中的属性.html:30

▼ 使用工厂函数创建对象（缺点：都是Object类）

使用字面量创建对象，会存在大量冗余代码

```

/*
创建一个对象
*/
var obj = {
  name: "孙悟空",
  age: 18,
  gender: "男",
  sayName: function () {
    alert(this.name);
  }
};

var obj2 = {
  name: "猪八戒",
  age: 38,
  gender: "男",
  sayName: function () {
    alert(this.name);
  }
};

```

```

var obj3 = {
  name: "沙和尚",
  age: 28,
  gender: "男",
  sayName: function () {
    alert(this.name);
  }
};

obj.sayName();
obj2.sayName();
obj3.sayName();

```

通过工厂函数创建对象，可以批量创建

```

/*
使用工厂方法创建对象
通过该方法可以大批量的创建对象
*/
function createPerson(name, age, gender) {
  //创建一个新的对象
  var obj = new Object();
  //向对象中添加属性
  obj.name = name;
  obj.age = age;
  obj.gender = gender;
  obj.sayName = function () {
    alert(this.name);
  }
  //将新的对象返回
  return obj;
}

var obj2 = createPerson("孙悟空", 18, "男");
var obj3 = createPerson("猪八戒", 38, "男");
var obj4 = createPerson("沙和尚", 28, "男");
console.log(obj2);
console.log(obj3);
console.log(obj4);
obj2.sayName();
obj3.sayName();
obj4.sayName();

```

依次弹出“孙悟空”、“猪八戒”和“沙和尚”对话框，控制台打印创建的三个对象，都是Object类型

<div>▼ Object ⓘ</div> <div>age: 18</div> <div>gender: "男"</div> <div>name: "孙悟空"</div> <div>▶ sayName: f ()</div> <div>▶ [[Prototype]]: Object</div>	工厂函数创建对象.html:68
<div>▼ Object ⓘ</div> <div>age: 38</div> <div>gender: "男"</div> <div>name: "猪八戒"</div> <div>▶ sayName: f ()</div> <div>▶ [[Prototype]]: Object</div>	工厂函数创建对象.html:69
<div>▼ Object ⓘ</div> <div>age: 28</div> <div>gender: "男"</div> <div>name: "沙和尚"</div> <div>▶ sayName: f ()</div> <div>▶ [[Prototype]]: Object</div>	工厂函数创建对象.html:70

缺点：使用工厂函数创建对象，类型都是Object，没有做区分，比如下面创建狗的对象。

```

/*
使用工厂方法创建的对象，使用的构造函数都是Object
所以创建的对象都是Object这个类型
就导致我们无法区分出多种不同类型的对象
*/

//用来创建狗的对象

function createDog(name, age) {
  var obj = new Object();
  obj.name = name;
  obj.age = age;
  obj.sayHello = function () {
    alert("汪汪");
  }
  return obj;
}

var dog = createDog("发财", 3);
console.log(dog);

```

▼ 构造函数创建对象/instanceof/方法改写（引入原型）

创建一个构造函数，专门用来创建Person对象的

构造函数就是一个普通的函数，创建方式和普通函数没有区别

不同的是构造函数习惯上首字母大写

构造函数和普通函数的区别就是调用方式的不同

普通函数是直接调用，而**构造函数需要使用new关键字来调用**

构造函数的执行流程：

- 1.立刻创建一个新的对象
- 2.将新建的对象设置为函数中的this,在构造函数中可以使用this来引用新建的对象
- 3.逐行执行函数中的代码
- 4.将新的对象作为返回值返回

使用同一个构造函数创建的对象，我们称为一类对象，也将一个构造函数称为一个类

我们将通过一个构造函数创建的对象，称为是该类的实例

this的情况：

- 1.当以函数的形式调用时，this是window
- 2.当以方法的形式调用时，谁调用函数this就是谁
- 3.**当以构造函数的形式调用时，this就是新创建的那个对象**

```
function Person(name, age, gender) {  
    //alert(this);  
    this.name = name;  
    this.age = age;  
    this.gender = gender;  
    this.sayName = function () {  
        alert(this.name);  
    }  
}  
  
var per = new Person("孙悟空", 18, "男");  
var per2 = new Person("玉兔精", 15, "女");  
var per3 = new Person("沙和尚", 28, "男");  
console.log(per);  
console.log(per2);  
console.log(per3);  
  
function Dog() {  
  
}  
var dog = new Dog();  
console.log(dog);
```

<div>▼ Person ⓘ</div> <div> age: 18 gender: "男" name: "孙悟空" ▶ sayName: f () ▶ [[Prototype]]: Object </div>	构造函数创建对象.html:47
<div>▼ Person ⓘ</div> <div> age: 15 gender: "女" name: "玉兔精" ▶ sayName: f () ▶ [[Prototype]]: Object </div>	构造函数创建对象.html:48
<div>▼ Person ⓘ</div> <div> age: 28 gender: "男" name: "沙和尚" ▶ sayName: f () ▶ [[Prototype]]: Object </div>	构造函数创建对象.html:49
<div>▼ Dog ⓘ</div> <div> ▶ [[Prototype]]: Object </div>	构造函数创建对象.html:55

instanceof:检查一个对象是否是一个类的实例

所有对象都是Object的后代

```

/*
使用instanceof可以检查一个对象是否是一个类的实例
语法：
    对象 instanceof 构造函数
如果是，则返回true，否则返回false
*/
console.log(per instanceof Person); //true
console.log(dog instanceof Person); //false

/*
所有的对象都是Object的后代
所以任何对象和Object做instanceof检查是都会返回true
*/
console.log(per instanceof Object); //true
console.log(dog instanceof Object); //true

```

创建方法（解决代码冗余问题：构造函数+向原型添加方法）

创建一个Person构造函数

- 在Person构造函数中，为每一个对象都添加了一个sayName方法，
 目前我们的方法是在构造函数内部创建，
 也就是构造函数每执行一次就会创建一个新的sayName方法
 也就是所有实例的sayName都是唯一的
 这样就导致了构造函数执行一次就会创建一个新的方法，

执行10000次就会创建10000个新的方法，而10000个方法都是一模一样的，这是完全没有必要的，完全可以使所有的对象共享同一个方法

```
function Person(name, age, gender) {
    this.name = name;
    this.age = age;
    this.gender = gender;
    //向对象中添加一个方法
    this.sayName = function () {
        alert("Hello大家好，我是：" + this.name);
    };
}
//创建一个Person实例
var per = new Person("Sophie", 23, "女");
var per2 = new Person("Sophia", 23, "女");
per.sayName();
per2.sayName();
console.log(per.sayName == per2.sayName); //true
```

根据上述缺点，考虑在全局作用域定义方法，但是**不安全且会污染全局作用域的命名空间**

```
function Person(name, age, gender) {
    this.name = name;
    this.age = age;
    this.gender = gender;
    this.sayName = fun;
}

//将sayName方法在全局作用域中定义
/*
将函数定义在全局作用域，污染了全局作用域的命名空间
而且定义在全局作用域中也很不安全
*/
function fun() {
    alert("Hello大家好，我是：" + this.name);
}
//创建一个Person实例
var per = new Person("Sophie", 23, "女");
var per2 = new Person("Sophia", 23, "女");
per.sayName();
per2.sayName();
console.log(per.sayName == per2.sayName); //true
```

因此考虑向原型中添加sayName方法

```
function Person(name, age, gender) {
    this.name = name;
    this.age = age;
    this.gender = gender;
```

```
}

Person.prototype.sayName = function () {
    alert("Hello大家好, 我是:" + this.name);
}

//创建一个Person实例
var per = new Person("Sophie", 23, "女");
var per2 = new Person("Sophia", 23, "女");
per.sayName();
per2.sayName();
console.log(per.sayName == per2.sayName); //true
```

▼ 原型对象

原型prototype

我们所创建的每一个函数，解析器都会向函数中添加一个属性prototype

这个属性对应着一个对象，这个对象就是我们所谓的原型对象

如果函数作为普通函数调用，prototype没有任何作用

当函数通过 构造函数调用时，它所创建的对象中都会有一个隐含的属性，

指向该构造函数的原型对象,可以通过下划线__proto__来访问

原型对象就相当于一个公共的区域，所有同一个类的实例都可以访问到这个原型对象。

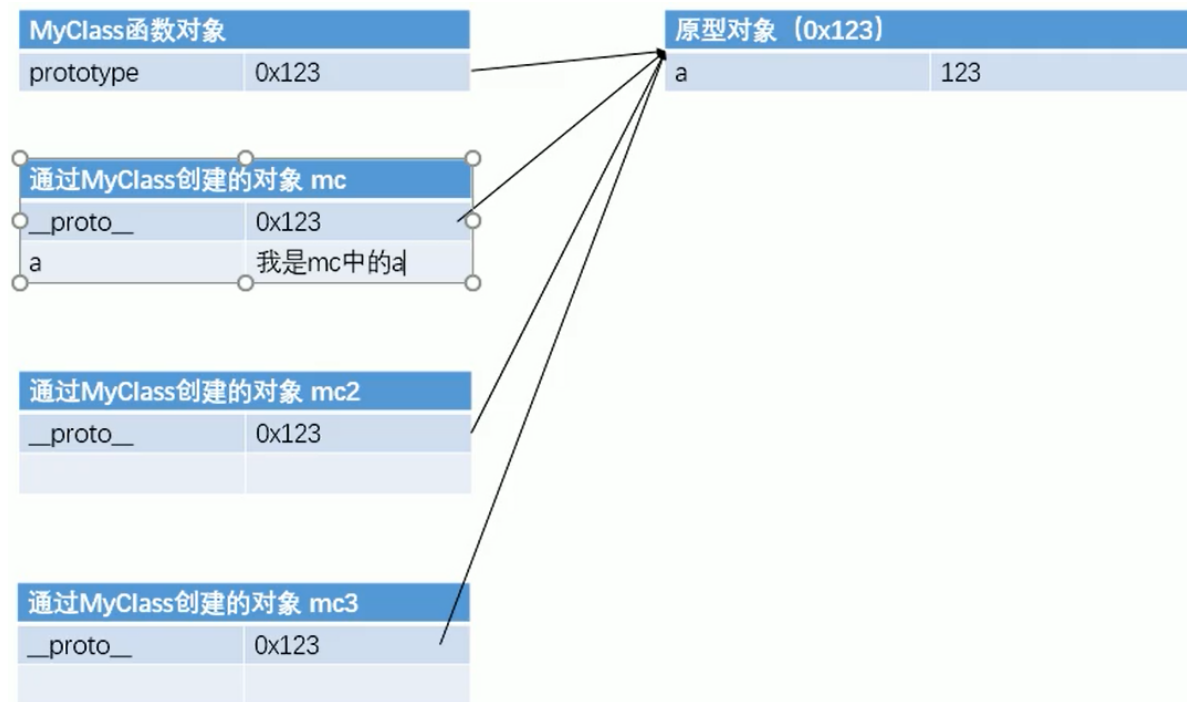
我们可以将对象中共有的内容，统一设置到原型对象中

当我们访问对象的一个属性或者方法时，它会现在对象自身中寻找，如果有则直接使用，

如果没有则会去原型对象中寻找，如果找到则直接使用

以后我们创建构造函数时。可以将这些对象共有的属性和方法，统一添加到构造函数的原型对象中

这样不用分别为每一个对象添加，也不会影响到全局作用域，就可以使每个对象都具有这些属性和方法



```
function MyClass() {
}
//向MyClass的原型中添加一个属性a
MyClass.prototype.a = 123;

//向MyClass的原型中添加一个方法
MyClass.prototype.sayHello = function () {
    alert("Hello");
}
var mc = new MyClass();
var mc2 = new MyClass();
mc.a = "我是mc中的a";
console.log(MyClass.prototype);
console.log(mc.__proto__);
console.log(mc.__proto__ == MyClass.prototype); //true

console.log(mc.a); //我是mc中的a
console.log(mc2.a); //123

mc.sayHello();//mc中没有，但是原型对象中有该方法
```

显式原型&隐式原型

每个函数都有一个prototype属性，它默认指向一个Object实例空对象（原型对象），原型对象中有一个属性constructor，它指向函数对象。

每个函数function都有一个prototype，即显式原型（属性）；
每个实例对象都有一个 __proto__，可称为隐式原型（属性）。

实例对象的隐式原型的值为其构造函数的显式原型的值。

hasOwnProperty() 是Object的方法

```
//创建一个构造函数
function MyClass() {

}

//向MyClass的原型中添加一个Name属性
MyClass.prototype.name = "我是原型中的名字";
var mc = new MyClass();
mc.age = 18;
console.log(mc.name);    //我是原型中的名字

//使用in检查对象中是否含有某个属性时，如果对象中没有但是原型中有，也会返回true
console.log("name" in mc);    //true
//可以使用对象的hasOwnProperty()来检查对象自身中是否含有该属性
//使用该方法只有当对象自身中含有属性时，才会返回true
console.log(mc.hasOwnProperty("name"));    //false
console.log(mc.hasOwnProperty("age"));    //true
```

原型链

- 1、访问一个对象的属性时，先在自身属性中查找， 如果找到就返回，如果没找到，沿着 __proto__ 这条链向上找，直到找到就返回，如果没找到，返回undefined
- 2、别名：隐式原型链
- 3、作用：查找对象属性（方法）
- 4、Function的prototype与 __proto__ 是指向一个地方。
- 5、所有函数的 __proto__ 都是相等的，因为都是New Function () 创建的，都等于Function.prototype。
- 6、函数的显式原型指向的对象默认是空的Object实例对象（Object不满足）
- 7、Object的原型对象是原型链尽头！（Object.prototype.__proto__=null）

```
/*
原型对象也是对象，所以它也有原型
当我们使用一个对象的属性或方法时，会先在自身寻找
    自身如果有，则直接使用，
    如果没有则取原型对象中寻找，如果原型对象中有，则使用，
    如果没有则去原型的原型中寻找，直到找到Object对象的原型
    Object对象的原型没有原型，如果在Object中依然没有找到，则返回undefined
*/
//创建一个构造函数
function MyClass() {
```

```

}


//向MyClass的原型中添加一个Name属性
MyClass.prototype.name = "我是原型中的名字";
var mc = new MyClass();
mc.age = 18;

console.log(mc.hasOwnProperty("hasOwnProperty")); //false, 代表是原型的方法
console.log(mc.__proto__.hasOwnProperty("hasOwnProperty")); //false
console.log(mc.__proto__.__proto__); //Object
console.log(mc.__proto__.__proto__.hasOwnProperty("hasOwnProperty")); //true
console.log(mc.__proto__.__proto__.__proto__); //null

```

继承

类

 js运行机制(线程)

7.函数

▼ 函数简介(创建)

```

13      /*
14      函数Function
15      函数也是一个对象
16      函数可以封装一些功能（代码），在需要时可以执行这些代码
17      函数中可以保存一些代码在需要的时候调用
18      */
19
20      //创建一个函数对象
21      //可以将封装的代码以字符串的形式传递给构造函数
22      var fun = new Function("console.log('hello 这是我的第一个函数');");
23      console.log(typeof fun);
24      console.log(fun);
25
26      //封装到函数中的代码不会立即执行，
27      //函数中的代码会在函数调用时执行
28      //调用函数方式：函数对象（）
29      //当调用函数时，函数中封装的代码会按照顺序执行
30      fun();

```


function	函数.html:23
f anonymous() { console.log('hello 这是我的第一个函数'); }	函数.html:24
hello 这是我的第一个函数	VM11:3

我们在实际开发中很少使用构造函数来创建一个函数对象,通常使用**函数声明创建函数**

```

32      /*
33      使用 函数声明来创建一个函数
34      语法: function 函数名([形参1,形参2,形参3...形参N]){
35      |   语句...
36      |
37      */
38      function fun2() {
39      |   console.log("这是我的第二个函数");
40      |
41      console.log(fun2);
42      fun2();
43  
```

f fun2() { console.log("这是我的第二个函数"); }	函数.html:41
这是我的第二个函数	函数.html:39

第三种方法：**使用函数表达式创建一个函数（结合匿名函数，将匿名函数赋值给一个对象）**

```

45      /*
46      使用函数表达式来创建一个函数（匿名函数）
47      将匿名函数赋值给一个对象
48      var 函数名 = function([形参1,形参2,形参3...形参N]){
49      |   语句...
50      |
51      */
52      var fun3 = function () {
53      |   console.log("我是匿名函数中封装的代码");
54      |
55      console.log(fun3);
56      fun3();
57  
```

```
f() {
    console.log("我是匿名函数中封装的代码");
}
```

函数.html:55

我是匿名函数中封装的代码

函数.html:53

▼ 函数参数及返回值

▼ 全局作用域

▼ 函数作用域（局部作用域）

```
/*
函数作用域
-调用函数时创建的作用域，函数执行完毕后，函数作用域销毁
-每调用一次函数就会创建一个新的函数作用域，他们之间是互相独立的
-在函数作用域中可以访问到全局作用域的变量
  在局部作用域中无法访问到函数作用域的变量
-当在函数作用域操作一个变量时，它会先在自身作用域中寻找，如果有就直接使用
  如果没有则向上一级作用域中寻找，直到找到全局作用域
  如果全局作用域中依然没有找到，则会报错ReferenceError
-在函数中要访问全局变量可以使用window对象
*/
//创建一个变量
var a = 10;
function fun() {
    var a = "我是函数中的变量a";
    var b = 20;
    console.log("a=" + a); //a=我是函数中的变量a
}
fun();
//console.log("b=" + b);//Uncaught ReferenceError: b is not defined
console.log("a=" + a); //a=10
```

```
/*
在函数作用域也有声明提前的特性
使用var 关键字声明的变量，会在函数中所有的代码执行之前被声明
函数声明也会在函数中所有的代码执行之前执行
*/
function fun3() {
    fun4();
    console.log("a=" + a);
    var a = 35;
    function fun4() {
        alert("我是函数fun4");
    }
}
fun3();
```

```
var c = 33;
function fun5() {
    console.log("c=" + c); //c=undefined 变量c提前声明
    var c = 10;
}
fun5();
```

上述代码浏览器页面会率先弹出"我是函数fun4"提示框，说明函数声明会在函数种所有的代码执行之前执行，因此函数声明在后，依然可以提前调用。控制台显示a=undefined。原因：var关键字声明的变量提前了，因此找得到变量a，但是赋值在打印之后，因此返回a=undefined.同样的控制台显示c=undefined，因为变量c提前声明，但赋值在打印之后。

```
var c = 33;
function fun5() {
    console.log("c=" + c); //c=33 搜索到全局作用域的c值
    c = 10; //没有用var关键字声明变量，则会设置为全局变量
}
fun5();
console.log("c=" + c); //c=10
```

```
//形参就相当于函数作用域中的变量
var e = 23;
function fun6(e) {
    alert(e);
}
fun6(); //undefined
```

▼ 立即执行函数（匿名函数使用） IIFE（Immediately-Invoked Function Expression）

```
//调用函数：函数对象()
/*
立即执行函数
函数定义完，立即被调用，这种函数叫做立即执行函数
立即执行函数往往只会执行一次

作用：
    隐藏实现
    不会污染全局命名空间
    用它来编写js模块
*/
(function () {
    alert("我是一个匿名函数");
})();

(function (a, b) {
    console.log("a=" + a);
```

```
    console.log("b=" + b);
  })(123, 456);
```

▼ this(上下文对象)

解析器/浏览器在调用函数时，每次都会向函数内部传递进一个隐含的参数

这个隐含的参数就是this, **this指向的是一个对象**

这个对象我们成为函数执行的上下文对象

根据函数的调用方式不同,this会指向不同的对象

1.以函数的形式调用时, this永远都是Window

2.以方法的形式调用时, this就是调用方法的那个对象

```
function fun(a, b) {
  console.log("a=" + a, "b=" + b);
  console.log(this); //Window(Object)
}
fun(123, 456);

//创建一个对象
var obj = {
  name: "孙悟空",
  sayName: fun
};
obj.sayName(); //a=undefined b=undefined Object
console.log(obj.sayName == fun); //true
fun(); //a=undefined b=undefined Window

var obj2 = {
  name: "沙和尚",
  sayName: fun2
};
obj2.sayName(); //沙和尚 this指的是obj2


function fun2() {
  console.log(this.name);
}
var name = "全局的名称属性";
//以函数形式调用, this是window, this.name找的是全局变量
fun2(); //全局的名称属性
```

```
//创建一个name变量
var name = "全局";

//创建一个fun()函数
function fun() {
  console.log(this.name);
}

//创建两个对象
```

```
var obj = {
  name: "孙悟空",
  sayName: fun
};
var obj2 = {
  name: "沙和尚",
  sayName: fun
};
fun(); //以函数形式调用，this指向window 结果：全局
obj.sayName(); //孙悟空
obj2.sayName(); //沙和尚
```

 JavaScript中闭包的概念、原理、作用及应用

8.期约与异步函数

见ES6

9.DOM (Document Object Model)

▼ DOM简介

文档对象模型，通过DOM可以来任意来修改网页中各个内容

文档：文档指的是网页，一个网页就是一个文档

对象：对象指将网页中的每一个节点都转换为对象，转换完对象以后，就可以以一种纯面向对象的形式来操作网页了

模型：模型用来表示节点和节点之间的关系，方便操作页面

节点（Node）：节点是构成网页的最基本的单元，网页中的每一个部分都可以称为是一个节点，虽然都是节点，但是节点的类型却是不同的

常用的节点

- (1) **文档节点** (Document)，代表整个网页
- (2) **元素节点** (Element)，代表网页中的标签
- (3) **属性节点** (Attribute)，代表标签中的属性
- (4) **文本节点** (Text)，代表网页中的文本内容

▼ DOM操作

DOM查询在网页中浏览器已经为我们提供了**document**对象，它代表的是整个网页，它是**window**对象的属性，可以在页面中直接使用。

1、document查询方法：

根据元素的id属性查询一个元素节点对象：

```
document.getElementById("id属性值");
```

根据元素的name属性值查询一组元素节点对象：

```
document.getElementsByName("name属性值");
```

根据标签名来查询一组元素节点对象：

```
document.getElementsByTagName("标签名");
```

2、读取元素的属性：

语法：元素.属性名例子：

```
ele.name
```

```
ele.id
```

```
ele.value
```

```
ele.className
```

注意：class属性不能采用这种方式，**读取class属性时需要使用 元素.className**

3、修改元素的属性：

语法：元素.属性名 = 属性值

```
innerHTML
```

使用该属性可以获取或设置元素内部的HTML代码

▼ 事件（Event）

事件指的是用户和浏览器之间的交互行为。比如：点击按钮、关闭窗口、鼠标移动...

我们可以为事件来绑定回调函数来响应事件。

绑定事件的方式：

1、可以在标签的事件属性中设置相应的JS代码，这样当事件被触发时，这些代码将会执行。

这种写法称为结构和行为耦合，不方便维护，一般不用

```
<button id="btn" onclick="alert('讨厌，你点我干嘛');">我是一个按钮</button>
```

2、可以为按钮的对应事件绑定处理函数的形式来响应事件，这样当事件被触发时，其对应的函数将会被调用（回调函数）

```
<button id="btn">我是一个按钮</button>
//获取按钮对象
var btn = document.getElementById('btn');
//绑定一个单击事件
//像这种为单击事件绑定的函数，我们称为单击响应函数
btn.onclick = function () {
    alert("你还点");
};
```

▼ DOM查询

通过具体的元素节点来查询

元素.getElementsByTagName()

通过标签名查询当前元素的指定后代元素

子节点包括标签元素中的文本，子元素包含标签元素

元素.childNodes

获取当前元素的**所有子节点**，会获取到空白的文本子节点

childNodes属性会获取包括文本节点在内的所有节点

根据DOM标签标签间空白也会当成文本节点

注意：在IE8及以下的浏览器中，不会将空白文本当成子节点，所以该属性在IE8中会返回4个子元素而其他浏览器是9个

元素.children获取当前元素的**所有子元素**

元素.firstChild获取当前元素的**第一个子节点**，会获取到空白的文本子节点

元素.lastChild

获取当前元素的**最后一个子节点**

元素.parentNode

获取当前元素的父元素

元素.previousSibling

获取当前元素的前一个兄弟节点

元素.previousElementSibling

获取前一个兄弟元素，IE8及以下不支持

元素.nextSibling

获取当前元素的后一个兄弟节点

firstElementChild

获取当前元素的第一个子元素

firstElementChild不支持IE8及以下的浏览器，如果需要兼容他们尽量不要使用

innerHTML和innerText

这两个属性并没有在DOM标准定义，但是大部分浏览器都支持这两个属性

两个属性作用类似，都可以获取到标签内部的内容，**不同是innerHTML会获取到html标签，而innerText会自动去除标签**

如果使用这两个属性来设置标签内部的内容时，没有任何区别的

读取标签内部的文本内容

h1中的文本内容

元素.firstChild.nodeValue

```
<body>
  <div id="content">
    <div id="formlist">
      <div class="inner">
        <p>你喜欢哪个城市</p>
        <ul id="city">
```



```

        <li id="bj">北京</li>
        <li>上海</li>
        <li>广州</li>
        <li>杭州</li>
    </ul>
    <br />
    <br />
    <p>你喜欢哪款游戏</p>
    <ul id="game">
        <li id="gta">gta</li>
        <li>csgo</li>
        <li>data2</li>
        <li>destiny2</li>
    </ul>
    <br />
    <br />
    <p>你的手机操作系统是</p>
    <ul id="phone">
        <li>iOS</li>
        <li id="android">Android</li>
        <li>鸿蒙</li>
    </ul>
    <br />
    <br />
</div>
<br />
<div class="inner">
    <p>gender:</p>
    <input class="test" type="radio" name="gender" value="male">
    male
    <input type="radio" name="gender" value="female">
    female
    <br>
    <p>name:</p>
    <input type="text" name="name" id="username" value="">
    <br>
    <br>
</div>
</div>
<div id="btnlist">
    <div><button id="btn01">查找#bj节点</button></div>
    <div><button id="btn02">查找所有li节点</button></div>
    <div><button id="btn03">查找name=gender所有节点</button></div>
    <div><button id="btn04">查找#city下所有li节点</button></div>
    <div><button id="btn05">返回#city下所有子节点</button></div>
    <div><button id="btn06">返回#phone下第一个节点</button></div>
    <div><button id="btn07">返回#bj父亲节点</button></div>
    <div><button id="btn08">返回#android前一个兄弟节点</button></div>
    <div><button id="btn09">返回#username的value属性值</button></div>
    <div><button id="btn10">设置#username的value属性值</button></div>
    <div><button id="btn11">返回#bj的文本值</button></div>
</div>
</div>
<script>
    /*
    定义一个函数，专门用来为指定元素绑定单击响应事件

```

```

参数：
    idStr 要绑定单击响应函数的对象的id属性值
    fun 事件的回调函数，当单击元素时，该函数将会被触发
*/
function myClick(idStr, fun) {
    var btn = document.getElementById(idStr);
    btn.onclick = fun;
}
//查找#bj节点
var btn01 = document.getElementById("btn01");
btn01.onclick = function () {
    var bj = document.getElementById("bj");
    //打印bj
    //innerHTML 通过这个属性可以获取到元素内部的html代码
    alert(bj.innerHTML);
};

//查找所有li节点
//为id为btn02的按钮绑定一个单击响应函数
var btn02 = document.getElementById("btn02");
btn02.onclick = function () {
    //getElementsByTagName()可以根据标签名来获取一组元素节点对象
    //这个方法会给我们返回一个类数组对象，所有查询到的元素都会封装到对象中
    //即使查询到的元素只有一个，也会封装到数组中返回
    var lis = document.getElementsByTagName("li");
    //打印lis
    alert(lis);
    alert(lis.length);
    //遍历数组
    for (var i = 0; i < lis.length; i++) {
        alert(lis[i].innerHTML);
    }
};

//查找name=gender所有节点
var btn03 = document.getElementById("btn03");
btn03.onclick = function () {
    var inputs = document.getElementsByName("gender");
    alert(inputs);
    alert(inputs.length); //2
    for (var i = 0; i < inputs.length; i++) {
        /*
        innerHTML用于获取元素内部的HTML代码
        对于自结束标签，这个属性没有意义
        */
        /*
        如果需要读取元素节点属性，直接使用 元素.属性名
        例子：元素.id 元素.name 元素.value
        注意：class属性不能采用这种方式
        读取class属性时需要使用元素.className
        */
        alert(inputs[i].value);
    }
};

//查找#city下所有li节点

```

```

var btn04 = document.getElementById("btn04");
btn04.onclick = function () {
    // alert("店家");
    var city = document.getElementById("city");
    var lis = city.getElementsByTagName("li");
    // alert(lis.length);
    for (var i = 0; i < lis.length; i++) {
        alert(lis[i].innerHTML);
    }
};

//返回#city下所有子节点
var btn05 = document.getElementById("btn05");
btn05.onclick = function () {
    var city = document.getElementById("city");
    /*
    childNodes属性会获取包括文本节点在内的所有节点
    根据DOM标准，标签间的空白也会当成文本节点
    */
    var cns = city.childNodes;
    alert(cns.length); //9
    /*
    children属性可以获取当前元素的所有子元素
    */
    var cns2 = city.children;
    alert(cns2.length); //4
};

//返回#phone下第一个节点
var btn06 = document.getElementById("btn06");
btn06.onclick = function () {
    var phone = document.getElementById("phone");
    //firstChild可以获取到当前元素的第一个子节点（包括空白文本节点）
    // var fir = phone.firstChild;
    //firstElementChild获取当前元素的第一个子元素（仅支持IE8以上）
    //如果兼容IE8，尽量避免使用
    var fir = phone.firstElementChild;
    alert(fir);
};

//返回#bj父亲节点
myClick("btn07", function () {
    var bj = document.getElementById("bj");
    var pn = bj.parentNode;
    alert(pn.innerHTML);
    /*
    innerText
    该属性可以获取到元素内部的文本内容
    它和innerHTML类似，不同的是他会自动将html去除
    */
    alert(pn.innerText);
});

//返回#android前一个兄弟节点
myClick("btn08", function () {

```

```

        var android = document.getElementById("android");
        var psn = android.previousSibling; //Text空白
        //previousElementSibling获取前一个兄弟元素，IE8及以下不支持
        //var psn = android.previousElementSibling;
        alert(psn.innerText);
    });

    //返回#username的value属性值
    myClick("btn09", function () {
        var user = document.getElementById("username");
        //读取user的username属性值
        //文本框的value属性值，就是文本框中填写的内容
        var value = user.value;
        alert(value);
    });

    //设置#username的value属性值
    myClick("btn10", function () {
        var user = document.getElementById("username");
        user.value = "今天天气真不错";
    });

    //返回#bj的文本值
    myClick("btn11", function () {
        var bj = document.getElementById("bj");
        // alert(bj.innerText);
        //获取bj中的文本节点,用的少
        // var fc = bj.firstChild;
        // alert(fc.nodeValue);
        alert(bj.firstChild.nodeValue);
    });
</script>
</body>

```

document.all

获取页面中的所有元素，相当于document.getElementsByTagName("*");

document.documentElement

获取页面中html根元素

document.body

获取页面中的body元素

document.getElementsByClassName()

根据元素的class属性值查询一组元素节点对象这个方法不支持IE8及以下的浏览器

document.querySelector()

根据CSS选择器去页面中查询一个元素如果匹配到的元素有多个，则它会返回查询到的第一个元素

document.querySelectorAll()

根据CSS选择器去页面中查询一组元素会将匹配到所有元素封装到一个数组中返回，即使只匹配到一个

```
<body>
  <div id="box2"></div>
  <div class="box1">
    我是第一个box1
    <div>我是box1中的div</div>
  </div>
  <div class="box1">
    <div>我是box1中的div</div>
  </div>
  <div class="box1">
    <div>我是box1中的div</div>
  </div>
</div>
<script>
  //获取body标签
  // var body = document.getElementsByTagName("body")[0];
  /*
  在document中有一个属性body，它保存的是body的引用
  */
  var body = document.body;
  /*
  document.documentElement保存的是html根标签
  */
  var html = document.documentElement;

  /*
  document.all代表的是页面中所有的元素
  */
  // var all = document.all;
  // for (var i = 0; i < all.length; i++) {
  //   console.log(all[i]);
  // }
  // console.log(all); //HTMLCollection(8)

  all = document.getElementsByTagName("");
  console.log(all);

  /*
```

```

根据元素的class属性值查询一组元素节点对象
getElementsByClassName(), 可以根据class属性值获取一组元素节点对象
但是该方法不支持IE8及以下的浏览器
*/
// var box1 = document.getElementsByClassName("box1");
// console.log(box1);

//获取页面中所有的div
// var divs = document.getElementsByTagName("div");
//获取class为box1中的所有的div
//.box1 div
/*
document.querySelector()
- 需要一个选择器的字符串作为参数, 可以根据一个CSS选择器来查询一个元素节点
- 虽然IE8中没有getElementsByClassName()但是可以使用querySelector()代替
- 使用该方法总会返回唯一的一个元素, 如果满足条件的元素有多个, 那么只会返回第一个
*/
var div = document.querySelector(".box1 div");
var box1 = document.querySelector(".box1");
console.log(box1.innerHTML);
console.log(div.innerHTML);

/*
document.querySelectorAll()
- 该方法与 querySelector()用法类似, 不同的是它会将符合条件的元素封装到一个数组中返回
- 即使符合条件的元素只有一个, 它也会返回数组
*/
var box1 = document.querySelectorAll(".box1");
box1 = document.querySelectorAll("#box2");
console.log(box1.length); //1
console.log(box1);

```

▼ DOM增删改

`document.createElement("TagName");`

可以用于创建一个元素节点对象, 它需要一个标签名作为参数, 将会根据该标签名创建元素节点对象, 并将创建好的对象作为返回值返回

`document.createTextNode("textContent");`

可以根据文本内容创建一个文本节点对象

父节点.appendChild(子节点);

向父节点中添加指定的子节点

父节点.insertBefore(新节点,旧节点);

将一个新的节点插入到旧节点的前边

父节点.replaceChild(新节点,旧节点);

使用一个新的节点去替换旧节点

父节点.removeChild(子节点);

删除指定的子节点推荐方式：

子节点.parentNode.removeChild(子节点) 自删除

以上方法，实际就是改变了相应元素（标签）的innerHTML的值。

使用innerHTML也可以完成DOM的增删改的相关操作。一般我们会两种方式结合使用。参见btn07.

```
<body>
  <div id="total">
    <div id="formlist">
      <div class="inner">
        <p>你喜欢哪个城市？</p>
        <ul id="city">
          <li id="bj">北京</li>
          <li>上海</li>
          <li>杭州</li>
          <li>武汉</li>
        </ul>
        <br>
        <br>
      </div>
    </div>

    <div id="btnList">
      <div><button type="button" id="btn01">创建一个“广州”节点添加到#city下</button></div>
      <div><button type="button" id="btn02">将“广州”节点插入到#bj前面</button></div>
      <div><button type="button" id="btn03">使用“广州”节点替换#bj节点</button></div>
      <div><button type="button" id="btn04">删除#bj节点</button></div>
      <div><button type="button" id="btn05">读取#city内的HTML代码</button></div>
      <div><button type="button" id="btn06">设置#bj内的HTML代码</button></div>
      <div><button type="button" id="btn07">创建一个“广州”节点添加到#city下</button></div>
    </div>
  </div>

  <script>
    function myClick(idStr, fun) {
      var btn = document.getElementById(idStr);
      btn.onclick = fun;
    }

    //获取bj节点
    var bj = document.getElementById("bj");

    //获取id为city的节点
    var city = document.getElementById("city");

    //创建一个“广州”节点添加到#city下
    myClick("btn01", function () {
      //创建“广州”节点<li>广州</li>
      //创建一个li元素节点
```

```

/*
document.createElement()
    可以用于创建一个元素节点对象
    它需要一个标签名作为参数，将会根据标签名创建元素节点对象
    并将创建好的对象作为返回值返回
*/
var li = document.createElement("li");
// alert(li);
//创建“广州”文本节点
/*
document.createTextNode()
    可以用来创建一个文本节点对象
    需要一个文本内容作为参数，将会根据内容创建文本节点，并将新的节点返回
*/
var gzText = document.createTextNode("广州");
// console.log(gzText);
//将gzText设置li子节点
/*
appendChild()
    -向一个父节点中添加一个新的子节点
    -用法：父节点.appendChild(子节点)
*/
li.appendChild(gzText);

//将广州添加到city下
city.appendChild(li);

});

//将“广州”节点插入到#bj前面
myClick("btn02", function () {
    //创建一个广州
    var li = document.createElement("li");
    var gzText = document.createTextNode("广州");
    li.appendChild(gzText);
    // //获取id为bj的节点
    // var bj = document.getElementById("bj");
    // //获取city
    // var city = document.getElementById("city");

    /*
    insertBefore()
        可以在指定的字节带前插入新的子节点
        语法：
        父节点.insertBefore(新节点, 旧节点)
    */
    city.insertBefore(li, bj);
});

//使用“广州”节点替换#bj节点
myClick("btn03", function () {
    //创建广州节点
    var li = document.createElement("li");
    var gzText = document.createTextNode("广州");

```



```

        li.appendChild(gzText);
        city.replaceChild(li, bj);
    });

    //删除#bj节点
    myClick("btn04", function () {
        // city.removeChild(bj);
        bj.parentNode.removeChild(bj);
    });

    //读取#city内的HTML代码
    myClick("btn05", function () {
        alert(city.innerHTML);
    });

    //设置#bj内的HTML代码
    myClick("btn06", function () {
        bj.innerHTML = "昌平";
    });

    //
    myClick("btn07", function () {
        //向city中获取广州
        /*
        使用innerHTML也可以完成DOM的增删改的相关操作
        一般我们会两种方式结合使用
        */
        // city.innerHTML += "<li>广州</li>";
        //创建一个li
        var li = document.createElement("li");
        //向li中设置文本
        li.innerHTML = "广州";
        //将li添加到city中
        city.appendChild(li);
    });
</script>
</body>

```

▼ DOM对CSS的操作 (getStyle())

修改元素样式： 元素.style.样式名 = 样式值；

获取元素当前显示的样式： 元素.currentStyle.样式名（只支持IE浏览器）

getComputedStyle() 正常浏览器，返回包含样式信息的对

象

```

<style>
    #box1 {
        width: 200px;
        height: 200px;
        background-color: red;
    }

```

```

    }
</style>

<button id="btn01">点我一下</button>
<button id="btn02">点我一下</button>
<div id="box1"></div>

//点击按钮以后，修改box的大小
var box1 = document.getElementById("box1");
var btn01 = document.getElementById("btn01");
btn01.onclick = function () {
    //修改box1的宽度
    /*
    通过JS修改元素的样式
    语法：元素.style.样式名 = 样式值；
    注意，如果CSS的样式中含有-
    这种名称在JS是不合法的比如background-color
    需要将这种样式名修改为驼峰命名法，
    去掉-，然后将-后的字母大写

    我们通过style属性设置的样式都是内联样式，
    而内联样式具有较高的优先级，所以通过JS修改的样式往往会立刻显示

    但是如果在样式中写了!important，则此时样式会有更高的优先级，
    即使通过JS也不能覆盖该样式，此时将会导致JS修改样式失效，
    因此尽量不写!important
    */
    box1.style.width = "300px";
    box1.style.height = "300px";
    box1.style.backgroundColor = "yellow";
};

//点击按钮2以后，读取元素样式
var btn02 = document.getElementById("btn02");

/*
通过style属性设置和读取的都是内联样式
无法读取样式表中的样式
所以上来点按钮2，不会弹出样式
*/
// btn02.onclick = function () {
//     //读取box1内联样式 元素.style.样式名
//     alert(box1.style.backgroundColor);
// };

/*
获取元素的当前显示的样式
语法：元素.currentStyle.样式名
它可以用来读取当前元素正在显示的样式

currentStyle只有IE浏览器支持，其他的浏览器都不支持
*/
btn02.onclick = function () {
    // alert(box1.currentStyle.backgroundColor);
    /*

```

在其他浏览器中可以使用

getComputedStyle() 这个方法来获取元素当前的样式

这个方法是window的方法，可以直接使用

需要两个参数：

第一个：要获取样式的元素

第二个：可以传递一个伪元素，一般都传null

该方法会返回一个对象，对象中封装了当前元素对应的样式

可以通过对象.样式名来读取样式，

如果获取的样式没有设置，则会获取到真实的值，而不是默认值

比如：没有设置width，它不会获取到auto，而是一个长度

但是该方法不支持IE8及以下的浏览器

```
*/
// var obj = getComputedStyle(box1, null);
// alert(obj.width);
alert(getStyle(box1, "backgroundColor"));
};

/*
定义一个函数，用来获取指定元素的当前的样式
参数：
    obj 要获取样式的元素
    name 要获取的样式名

通过currentStyle()和getComputedStyle()读取到的样式是只读的不能修改
*/
function getStyle(obj, name) {
    //正常浏览器的方式
    if (window.getComputedStyle) {
        return getComputedStyle(obj, null)[name];
    } else {
        //IE8方式，没有getComputedStyle()方法
        return obj.currentStyle[name];
    }
}
```

▼ 其他样式相关的属性

注意：以下样式都是只读的,未指明偏移量都是相对于当前窗口左上角

clientHeight元素的可见高度，包括元素的内容区和内边距的高度

clientWidth元素的可见宽度，包括元素的内容区和内边距的宽度

offsetHeight整个元素的高度，包括内容区、内边距、边框

offsetWidth整个元素的宽度，包括内容区、内边距、边框

offsetParent当前元素的定位父元素离他最近的开启了定位的祖先元素，如果所有的元素都没有开启定位，则返回body

offsetLeft / offsetTop当前元素和定位父元素之间的水平/垂直方向的偏移量

scrollHeight / scrollWidth获取元素滚动区域的高度和宽度

scrollTop / scrollLeft获取元素垂直和水平滚动条滚动的距离

判断滚动条是否滚动到底垂：

垂直滚动条scrollHeight - scrollTop = clientHeight

水平滚动scrollWidth - scrollLeft = clientWidth

10.事件

▼ 事件

事件，就是用户和浏览器之间的交互行为，比如：点击按钮，鼠标移动，关闭窗口....

可以为按钮的对应事件绑定处理函数的形式来响应事件，这样当事件被触发时，其对应的函数将会被调用（回调函数）。

```
<button id="btn">我是一个按钮</button>

//获取按钮对象
var btn = document.getElementById('btn');
//绑定一个单击事件
//像这种为单击事件绑定的函数，我们称为单击响应函数
btn.onclick = function () {
    alert("你还点");
};
```

事件源：绑定事件的元素

事件目标：触发事件的元素

事件分类：键盘类、鼠标类、表单类、浏览器类（location href scroll load）

▼ 事件的绑定

- DOM0级（赋值级）

使用 **对象.事件 = 函数** 的形式绑定响应函数，
它只能同时为一个元素的一个事件绑定一个响应函数
不能绑定多个，如果绑定了多个，则后边会覆盖掉前边的

```
<button id="btn01">点我一下</button>

//该例子只弹出2
var btn01 = document.getElementById("btn01");
//为btn01绑定一个单击响应函数
```

```
btn01.onclick = function () {  
    alert(1);  
};  
//为btn01绑定第二个单击响应函数  
btn01.onclick = function () {  
    alert(2);  
};
```

- **DOM2级（监听式 addEventListener()）**

通过这个方法也可以为元素绑定响应函数，**一般封装某个功能需要用到绑定事件都是用监听式事件绑定，防止覆盖**

参数：

- 1.事件的字符串，不要on
- 2.回调函数，当事件触发时，该函数会被调用
- 3.是否在捕获阶段触发事件，需要一个布尔值，一般都是false
 - 布尔：
 - false表示**冒泡状态**，默认值
 - true表示**捕获状态**，IE不支持

使用addEventListener()可以同时为一个元素的相同事件同时绑定多个响应函数，这样当事件被触发时，响应函数将会按照函数的绑定顺序执行

```
<button id="btn01">点我一下</button>  
  
//该例子依次弹出1, 2, 3  
var btn01 = document.getElementById("btn01");  
btn01.addEventListener("click", function () {  
    alert(1);  
}, false);  
btn01.addEventListener("click", function () {  
    alert(2);  
}, false);  
btn01.addEventListener("click", function () {  
    alert(3);  
}, false);
```

对于IE8使用attachEvent()来绑定事件

参数：

- 1.事件的字符串，要on
- 2.回调函数

这个方法也可以同时为一个事件绑定多个处理函数

不同的是它是**后绑定先执行**，执行顺序和addEventListener()相反

```
<button id="btn01">点我一下</button>

//该例子依次弹出3,2,1
var btn01 = document.getElementById("btn01");
btn01.attachEvent("onclick", function () {
    alert(1);
});
btn01.attachEvent("onclick", function () {
    alert(2);
});
btn01.attachEvent("onclick", function () {
    alert(3);
});
```

- **考虑浏览器类型的绑定函数**

```
//定义一个函数，用来为指定元素绑定响应函数
/*
addEventListener()中的this，是绑定事件的对象
attachEvent()中的this，是window
需要统一两个方法this
*/
/*
参数：
    obj 要绑定事件的对象
    eventStr 事件的字符串(不要on)
    callback 回调函数
*/
function bind(obj, eventStr, callback) {
    if (obj.addEventListener) {
        //大部分浏览器兼容的方式
        obj.addEventListener(eventStr, callback, false);
    } else {
        /*
        this是谁由调用方式决定
        callback.call(obj)
        */
        //IE8及以下
        obj.attachEvent("on" + eventStr, function () {
            //在匿名函数中调用回调函数
            callback.call(obj);
        });
    }
}
```

▼ 事件的传播（事件流）

- 关于事件的传播，网景公司和微软公司有不同的理解

-微软公司认为事件应该由内向外传播，也就是当事件触发时，应该先触发当前元素上的事件，然后再向当前元素的祖先元素上传播，也就是说事件应该在冒泡阶段执行

-网景公司认为事件应该是由外向内传播的，也就是当前事件触发时，应该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综合了两个公司的方案，将事件的传播分成了三个阶段：

1.捕获阶段

-在捕获阶段时，从最外层的祖先元素，向目标元素进行事件的捕获，但是默认此时不会触发事件

2.目标阶段

-事件捕获到目标元素，捕获结束，开始在目标元素上触发事件

3.冒泡阶段

-事件从目标元素向他的祖先元素传递，依次触发祖先元素上的事件，直至到达document对象

在这个阶段，一般是会触发所经过的元素的相同类型的事件函数

如果希望在捕获阶段就触发事件，可以将addEventListener()的第三个参数设置为true

一般情况下，我们不会希望在捕获阶段触发事件，所以第三个参数通常设为false

-IE8及以下的浏览器中没有捕获阶段

```
<style>
#box1 {
  width: 300px;
  height: 300px;
  background-color: yellowgreen;
}

#box2 {
  width: 200px;
  height: 200px;
  background-color: yellow;
}

#box3 {
  width: 100px;
  height: 100px;
  background-color: green;
}
</style>

<div id="box1">
```

```

<div id="box2">
  <div id="box3"></div>
</div>
</div>

//分别为三个div绑定单击响应函数
var box1 = document.getElementById("box1");
var box2 = document.getElementById("box2");
var box3 = document.getElementById("box3");

/*
当点击绿色方框，依次弹出"我是box3的响应函数", "我是box2的响应函数", "我是box1的响应函数"。
当点击黄色区域时，依次弹出 "我是box2的响应函数", "我是box1的响应函数"。
当点击黄绿色区域时，弹出"我是box1的响应函数"。
*/
bind(box1, "click", function () {
  alert("我是box1的响应函数");
});
bind(box2, "click", function () {
  alert("我是box2的响应函数");
});
bind(box3, "click", function () {
  alert("我是box3的响应函数");
});

function bind(obj, eventStr, callback) {
  if (obj.addEventListener) {
    //大部分浏览器兼容的方式
    obj.addEventListener(eventStr, callback, false);
  } else {
    /*
    this是谁由调用方式决定
    callback.call(obj)
    */
    //IE8及以下
    obj.attachEvent("on" + eventStr, function () {
      //在匿名函数中调用回调函数
      callback.call(obj);
    });
  }
}

```

• 原因分析

当我们点击绿色的div时，浏览器就开始工作了

1. 首先，浏览器要找到当前被点击的对象。从最外层的document开始寻找，一直沿着DOM树找到了被触发事件的对象。在这过程中是不会触发事件的(此阶段属于事件捕获过程)
2. 然后，浏览器找到被触发事件的对象，执行该对象相应的事件函数。在这里，浏览器找到了被点击的对象div.innerInner,然后执行该对象的点击事件函数(此阶段属于处于目标阶段)

3. 接下来，就以这个被触发的对象为起点，沿着DOM树向上冒泡，直到body元素为止。
在过程中所遇到的元素都会触发它们相同类型的事件函数

在这里，浏览器找到了id为box3的div，执行该对象的点击事件函数，然后事件冒泡，沿着DOM树向上冒泡。在冒泡过程中，遇到了div#box2,div#box1.,body,于是就执行它们的点击事件函数

- **事件的冒泡**

所谓的冒泡指的是事件的向上传导，当后代元素上的事件被触发时，其祖先元素的相同事件也会被触发

在开发中，冒泡中都是有用的，如果不希望发生事件冒泡，可以通过事件对象来取消冒泡：

event.cancelBubble = true;

```
<style>
  #box1 {
    width: 300px;
    height: 300px;
    background-color: yellowgreen;
  }

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

<div id="box1">
  我是box1
  <span id="s1">我是span</span>
</div>
<script>
  //为s1绑定单击响应函数
  var s1 = document.getElementById("s1");
  s1.onclick = function (event) {
    alert("我是span的单击响应函数");
    //取消冒泡
    //可以将事件对象的cancelBubble设置为true，即可取消冒泡
    event.cancelBubble = true;
  };

  //为box1绑定单击响应函数
  var box1 = document.getElementById("box1");
  box1.onclick = function (event) {
    alert("我是box1的单击响应函数");
    event.cancelBubble = true;
  };

  //为body绑定单击响应函数
  var body = document.body;
  body.onclick = function () {
```

```

        alert("我是body的单击响应函数");
    };
</script>

```

▼ 事件的委派

指将事件统一绑定给元素的共同的祖先元素，这样当后代元素上的事件触发时，会一直冒泡到祖先元素，从而通过祖先元素的响应函数来处理事件。

事件委派是利用了冒泡，通过委派可以减少事件绑定的次数，提高程序的性能。

```

<button id="btn01">添加超链接</button>
<ul id="u1" style="background-color: aquamarine;">
    <li><a href="javascript:;" class="link">超链接1</a></li>
    <li><a href="javascript:;" class="link">超链接2</a></li>
    <li><a href="javascript:;" class="link">超链接3</a></li>
    <li><a href="javascript:;" class="link">超链接4</a></li>
    <li><a href="javascript:;" class="link">超链接5</a></li>
    <li><a href="javascript:;" class="link">超链接6</a></li>
    <li><a href="javascript:;" class="link">超链接7</a></li>
    <li><a href="javascript:;" class="link">超链接8</a></li>
</ul>

/*
点击按钮，添加超链接
*/
var u1 = document.getElementById("u1");
var btn = document.getElementById("btn01");
btn.onclick = function () {
    //创建一个li
    var li = document.createElement("li");
    li.innerHTML = "<a href='javascript: ; ' class = 'link'>新建的超链接</a>";
    //将li添加到ul中
    u1.appendChild(li);

};
/*
为每一个超链接绑定一个单击响应函数
这里我们为每一个超链接都绑定了一个单击响应函数，这种操作比较麻烦
而且这种操作只能为已有的超链接设置事件，而新添加的超链接必须重新绑定
*/
//获取所有的a
var allA = document.getElementsByTagName("a");
//遍历
// for (var i = 0; i < allA.length; i++) {
//     allA[i].onclick = function () {
//         alert("我是单击响应函数");
//     };
// }

/*
我们希望，只绑定一次事件，即可应用到多个元素上，即使元素是后添加的
我们可以尝试将其绑定给元素的共同的祖先元素

```

事件委派是利用了冒泡，通过委派可以减少事件绑定的次数，提高程序的性能

```
*/
//为ul绑定一个单击响应函数
u1.onclick = function (event) {
    event = event || window.event;
    /*
    target
    event中的target表示的是触发事件的对象
    */
    // alert(event.target);

    //如果触发事件的对象是我们期望的元素，则执行，否则不执行
    if (event.target.className == "link") {
        alert("我是ul的单击响应函数");
    }
};
```

▼ 事件对象

当事件的响应函数被触发时，浏览器每次都会将一个事件对象作为实参传递进响应函数。在事件对象中封装了当前事件相关的一切信息，比如：鼠标的坐标 键盘哪个按键被按下 鼠标滚轮滚动方向等。

target

event中的target表示的是触发事件的对象

- 处理兼容性问题
在IE8中，响应函数被触发时，浏览器不会传递事件对象
在IE8及以下的浏览器中，是将事件对象作为window对象的属性保存的

```
元素.事件 = function(event){
    event = event || window.event;
};

元素.事件 = function(e){
    e = e || event;
};
```

比较常见的两种事件：

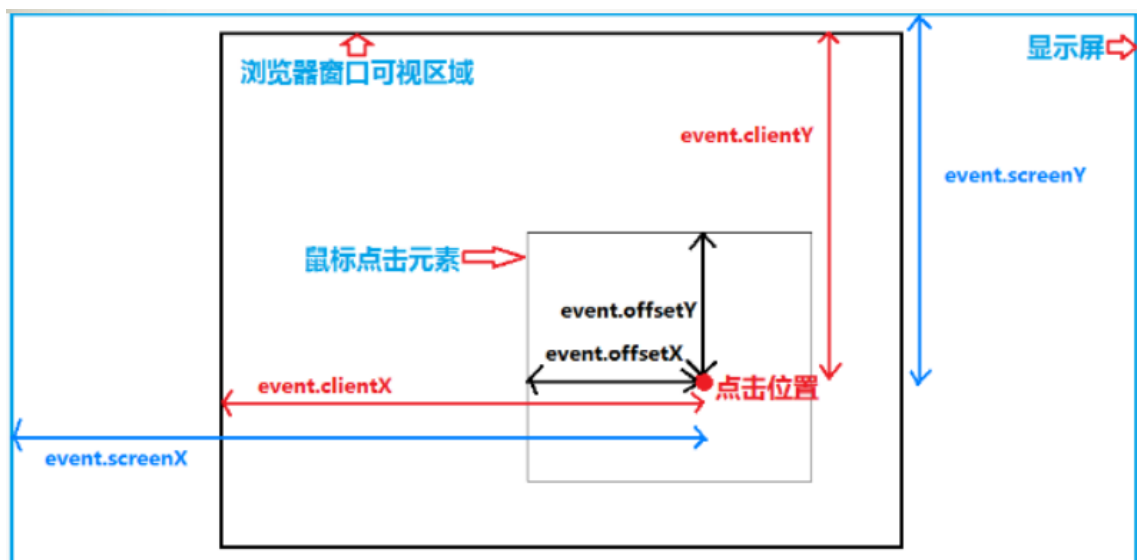
- 事件对象身上的属性（鼠标事件对象）
 - e.offsetX/Y 相对于事件目标的坐标
 - e.clientX/Y 相对于可视区域的坐标
 - e.pageX/Y 相对于页面的坐标

- e.button 鼠标的按键
- e.type 事件类型
- e.target/e.srcElement 事件目标
- 事件对象身上的属性（键盘事件对象）
 - 键盘事件添加给谁，怎么触发
 - 键盘事件，需要提前获取，焦点
 - 谁具有焦点，键盘事件就添加给谁，默认情况下document具有焦点
 - 事件对象的属性
 - e.keyCode 键码
 - e.which 键码
 - e.ctrlKey 返回当事件被触发时，"CTRL" 键是否被按下。
 - e.shiftKey 返回当事件被触发时，"SHIFT" 键是否被按下。
 - e.altKey 返回当事件被触发时，"ALT" 是否被按下。
 - e.metaKey 返回当事件被触发时，"meta" 键是否被按下。

▼ 常用事件

- 事件坐标

▼ 示意图



▼ 视口坐标 (clientX/clientY)

事件发生时鼠标指针在视口中的水平和垂直位置。

▼ 页面坐标 (pageX/pageY)

事件发生时鼠标指针在页面中的水平和垂直位置。

在页面没有滚动的情况下，pageX和pageY的值与clientX和clientY的值相等。

IE8及以下版本不支持页面坐标，使用视口坐标和滚动信息计算出来。

```
if(pageX===undefined){
    pageX = event.clientX + (document.body.scrollLeft ||
        document.documentElement.scrollLeft);
}

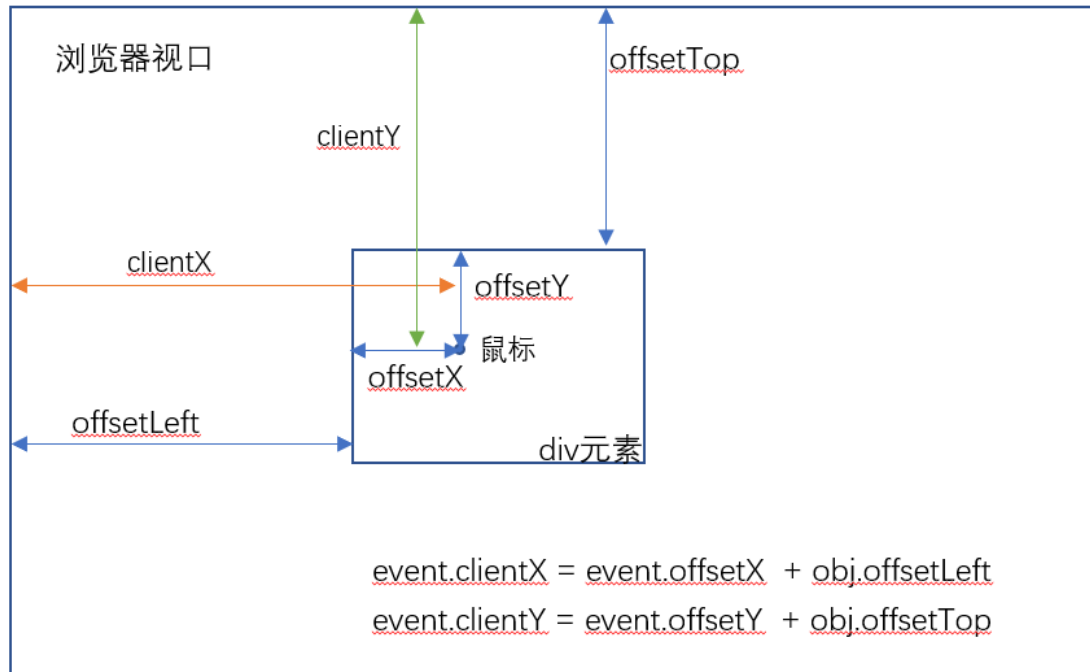
if(pageY===undefined){
    pageY = event.clientY + (document.body.scrollTop ||
        document.documentElement.scrollTop);
}
```

▼ 屏幕坐标 (screenX/screenY)

事件发生时鼠标指针相对于整个屏幕的坐标信息。

• 鼠标事件

▼ 鼠标拖拽



```

<style>
  #box1 {
    width: 100px;
    height: 100px;
    background-color: red;
    position: absolute;
  }

  #box2 {
    width: 100px;
    height: 100px;
    background-color: yellow;
    position: absolute;
    left: 200px;
    top: 200px;
  }
</style>
</head>

<body>
  我是一段文字
  <div id="box1"></div>
  <div id="box2"></div>
  <script>
    /*
     拖拽box1元素
     -拖拽流程
     1. 当鼠标在拖拽元素上按下时, 开始拖拽  onmousedown
     2. 当鼠标移动时, 被拖拽元素跟随鼠标移动  onmousemove
     3. 当鼠标松开时, 被拖拽元素被固定当前位置  onmouseup
    */
  
```

```

var box1 = document.getElementById("box1");
var box2 = document.getElementById("box2");

//开启box1的拖拽
drag(box1);
//开启box2的拖拽
drag(box2);

/*
提取一个专门用来设置拖拽的函数
参数：开启拖拽的元素
*/
function drag(obj) {
    //当鼠标在拖拽元素上按下时，开始拖拽
    obj.onmousedown = function (event) {

        //对IE8设置box1捕获所有的鼠标按下的事件
        // if (box1.setCapture) {
        //     box1.setCapture();
        // }
        obj.setCapture && obj.setCapture();

        event = event || window.event;
        //div的偏移量 鼠标.clientX - 元素.offsetLeft
        //div的偏移量 鼠标.clientY - 元素.offsetTop
        var ol = event.clientX - obj.offsetLeft;
        var ot = event.clientY - obj.offsetTop;

        // alert("鼠标按下，开始拖拽");
        //为document绑定一个onmousemove事件
        document.onmousemove = function (event) {
            //当鼠标移动时，被拖拽元素跟随鼠标移动
            event = event || window.event;
            var left = event.clientX - ol;
            var top = event.clientY - ot;

            //修改box1的位置
            obj.style.left = left + "px";
            obj.style.top = top + "px";
        };

        //当鼠标松开时，被拖拽元素被固定当前位置 onmouseup
        document.onmouseup = function () {
            //取消document.onmousemove事件
            document.onmousemove = null;

            //取消document.onmouseup事件
            document.onmouseup = null;

            //当鼠标松开时，取消对事件的捕获
            // if (box1.releaseCapture) {
            //     box1.releaseCapture();
            // }
            obj.releaseCapture && obj.releaseCapture();
        };
    };
};

```

```

        /*
        当我们拖拽一个网页的内容时，浏览器会默认去搜索引擎中搜索内容
        此时会导致拖拽功能的异常，这个时浏览器提供的默认行为，
        如果不希望发生这个行为，则可以通过return false来取消这个默认行为
        但是这招对IE8不起作用
        */
        return false;
    };
}
</script>
</body>

```

▼ 鼠标滚轮事件

```

<div id="box1"></div>

/*
当鼠标滚轮向下滚动时，box1变长
当鼠标滚轮向上滚动时，box1变短
*/
var box1 = document.getElementById("box1");

//为box1绑定一个鼠标滚轮滚动事件
/*
onwheel鼠标滚轮滚动的事件，会在滚轮滚动时触发
*/
box1.onwheel = function (event) {
    event = event || window.event;
    //判断鼠标滚轮滚动的方向
    //event.wheelDelta可以获取鼠标滚轮滚动的方向
    //向上滚180 向下滚-180
    //wheelDelta这个值我们不看大小只看正负
    //alert(event.wheelDelta);
    // alert("滚动");
    if (event.wheelDelta > 0) {
        // alert("向上滚");
        // 当鼠标滚轮向上滚动时，box1变短，clientHeight：元素的可见高度
        box1.style.height = box1.clientHeight - 10 + "px";
    } else {
        // alert("向下滚");
        // 当鼠标滚轮向下滚动时，box1变长
        box1.style.height = box1.clientHeight + 10 + "px";
    }
}

/*
当滚轮滚动时，如果浏览器有滚动条，滚动条会随之滚动，
这是浏览器的默认行为，如果不希望发生则可以取消默认行为
*/
return false;
};

```


- 键盘事件

- ▼ 键盘事件

onkeydown：按键被按下

-对于onkeydown来说如果一直按着某个按键不松手，则事件会一直触发

-当onkeydown连续触发时，第一次和第二次之间会间隔稍微长一点，其他的会非常快，这种设计是为了**防止误操作**

onkeyup：按键被松开

键盘事件一般都会绑定给一些可以获取到焦点的对象或者是document

```
<input type="text">

document.onkeydown = function (event) {
    event = event || window.event;
    /*
    可以通过keyCode来获取按键的编码
    通过它可以判断哪个按键被按下
    除了keyCode，事件对象中还提供了几个属性
        altKey
        ctrlKey
        shiftKey
        -这三个用来判断alt ctrl shift是否被按下
        如果按下则返回true，否则返回false
    */
    //判断一个y和ctrl是否同时被按下
    if (event.keyCode === 89 && event.ctrlKey) {
        console.log("y和ctrl被按下");
    }
};

document.onkeyup = function () {
    console.log("按键被松开");
};

//获取input
var input = document.getElementsByTagName("input")[0];
input.onkeydown = function (event) {
    event = event || window.event;
    console.log(event.keyCode);
    //是文本框中不能输入数字:48-57
    if (event.keyCode >= 48 && event.keyCode <= 57) {
        return false;
    }

    //在文本框中输入内容，是以onkeydown的默认行为
    //如果在onkeydown中取消了默认行为，则输入的内容不会出现在文本框中
    // return false;
};
```

▼ 键盘移动div

```
<div id="box1"></div>

//使div可以根据不同的方向键向不同的方向移动
/*
按左键, div向左移
按右键, div向右移
*/
var box1 = document.getElementById("box1");
//为document绑定一个键盘按下事件
document.onkeydown = function (event) {
    event = event || window.event;
    // console.log(event.keyCode);
    /*
    37  左
    38  上
    39  右
    40  下
    */
    switch (event.keyCode) {
        case 37:
            // alert("向左"); left值减小
            box1.style.left = box1.offsetLeft - 10 + "px";
            break;
        case 38:
            // alert("向上");
            box1.style.top = box1.offsetTop - 10 + "px";
            break;
        case 39:
            // alert("向右");
            box1.style.left = box1.offsetLeft + 10 + "px";
            break;
        case 40:
            // alert("向下");
            box1.style.top = box1.offsetTop + 10 + "px";

            break;
    }
};
```

11.BOM

▼ 定时器

▼ 定时调用 (setInterval())

- 可以将一个函数，每隔一段时间执行一次
 - 参数：
 - 1.回调函数，该函数会每隔一段时间被调用一次
 - 2.每次调用间隔的时间，单位是毫秒
 - 返回值：
 - 返回一个Number类型的数据
 - 这个数字用来作为定时器的唯一标识

```
<h1 id="count">1</h1>

var count = document.getElementById("count");
var num = 1;
var timer = setInterval(function () {
    count.innerHTML = num++;
    if (num == 11) {
        //clearInterval()可以用来关闭一个定时器
        //方法中需要一个定时器的标识作为参数，这样将关闭标识对应的定时器
        clearInterval(timer);
    }
}, 1000);
console.log(timer); //1
```

▼ 延时调用 (setTimeout())

延时调用一个函数不马上执行，而是隔一段时间以后再执行，而且只会执行一次

延时调用和定时调用的区别：定时调用会执行多次，而延时调用只会执行一次

延时调用和定时调用实际上是可以互相代替的，在开发中可以根据自己需要去选择

```
var timer = setTimeout(function () {
    console.log(num++);
}, 3000);

//使用clearTimeout()来关闭一个延时调用
clearTimeout(timer);
```

▼ 定时器的应用

- 轮播图

```
<script src="./js/tools.js"></script>
<style>
    * {
        margin: 0;
        padding: 0;
```

```

}

#outer {
    /*设置宽和高*/
    width: 1044px;
    height: 1448px;
    /*居中*/
    margin: 50px auto;
    /*设置背景颜色*/
    background-color: yellow;
    /*设置padding*/
    padding: 10px 0;
    /*开启相对定位*/
    position: relative;
    /*裁剪溢出的内容*/
    overflow: hidden;
}

/*设置imgList*/
#imgList {
    /*去除项目符号*/
    list-style: none;
    /*设置ul的宽度*/
    /* width: 5300px; */
    /*开启绝对定位*/
    position: absolute;
    /*设置偏移量*/
    left: 0px;
}

/*设置li浮动*/
#imgList li {
    /*设置浮动*/
    float: left;
    /*设置左右间距*/
    margin: 0 10px;
}

/*设置导航按钮*/
#navDiv {
    /*开启绝对定位*/
    position: absolute;
    /*设置位置*/
    bottom: 25px;
    /*设置left值
       outer 1044px
       navDiv: 5*25 = 125px
       1044-125=919
       919/2=459.5
    */
    /* left: 459px; */
}

#navDiv a {
    /*设置超链接浮动*/
    float: left;

```

```

        /*设置超链接宽高*/
        width: 15px;
        height: 15px;
        background-color: red;
        /*设置左右外边距*/
        margin: 0 5px;
        /*设置透明*/
        opacity: 0.5;
    }

    #navDiv a:hover {
        background-color: black;
    }
</style>
<!-- 引入move工具 -->
<script src="./js/tools.js"></script>

<body>
    <!-- 创建一个外部的div, 来作为大的容器 -->
    <div id="outer">
        <!-- 创建一个ul, 用于放置图片 -->
        <ul id="imgList">
            <li>
                
            </li>
            <li>
                
            </li>
            <li>
                
            </li>
            <li>
                
            </li>
            <li>
                
            </li>
        </ul>
        <!-- 创建导航按钮 -->
        <div id="navDiv">
            <a href="javascript:;"></a>
            <a href="javascript:;"></a>
            <a href="javascript:;"></a>
            <a href="javascript:;"></a>
            <a href="javascript:;"></a>
        </div>
    </div>
    <script>
        var imgList = document.getElementById("imgList");
        //获取页面中所有的图片
        var imgArr = document.getElementsByTagName("img");
        //设置imgList的宽度
        imgList.style.width = 1044 * imgArr.length + "px";

        //设置导航按钮居中

```

```

//获取navDiv
var navDiv = document.getElementById("navDiv");
//获取outer
var outer = document.getElementById("outer");
//设置navDiv的left值
navDiv.style.left = (outer.offsetWidth - navDiv.offsetWidth) / 2 + "px";

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

/*
点击超链接切换到指定图片
    点击第一个超链接显示第一个图片
*/
//为所有的超链接绑定单击响应函数
for (var i = 0; i < allA.length; i++) {
    //为每一个超链接添加一个num属性
    allA[i].num = i;

    allA[i].onclick = function () {
        //获取点击超链接的索引，并将其设置为index
        index = this.num;
        //切换图片
        /*
        第一张 0  0
        第二张 1  -1044
        第三张 2  -1044*2
        */
        // imgList.style.left = -1044 * index + "px";

        //使用move()移动移动图片
        move(imgList, "left", -1044 * index, 50);
        setA(index);
    };
}

//创建一个方法用来设置选中的a
function setA(index) {
    //遍历所有的a，并将背景颜色变为红褐色，并把选中的a设置为黑色
    for (var i = 0; i < allA.length; i++) {
        //去掉就是上面的默认样式
        allA[i].style.backgroundColor = "";
    }
    allA[index].style.backgroundColor = "black";
}

</script>
</body>

```

```

tools.js

//尝试创建一个可以执行简单动画的函数
/*
参数：
    obj:要执行动画的对象
    attr:要执行动画的样式，比如：left,height,width...
    target:执行动画的目标位置
    speed:移动的速度（正数向右移动，负数向左移动）
    callback:回调函数，这个函数将会在动画执行完毕以后执行
*/
function move(obj, attr, target, speed, callback) {
    //关闭上一个定时器
    /*
    避免同一个元素开启多个定时器
    */
    clearInterval(obj.timer);

    //获取元素目前的位置
    var current = parseInt(getStyle(obj, attr));

    //判断速度的正负值
    //如果从0到800移动，则speed为正
    //如果从800到0移动，则speed为负
    if (current > target) {
        //此时速度应为负值
        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，用来保存它自己的定时器的标识
    obj.timer = setInterval(function () {
        //获取box1原来的left的值
        var oldValue = parseInt(getStyle(obj, attr));
        //在旧值基础上增加
        var newValue = oldValue + speed;
        //判断newValue是否大于800
        //从800向0移动
        //向左移动时，需要判断newValue是否小于target
        //向右移动时，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box1
        obj.style[attr] = newValue + "px";

        //当元素移动到0px时，使其停止执行动画
        if (newValue == target) {
            clearInterval(obj.timer);
            //动画执行完毕，调用回调函数，且指挥执行一次
            callback && callback();
        }
    }, 30);
}

```

```

/*
定义一个函数，用来获取指定元素的当前的样式
参数：
    obj 要获取样式的元素
    name 要获取的样式名

通过currentStyle()和getComputedStyle()读取到的样式是只读的不能修改
*/
function getStyle(obj, name) {
    //正常浏览器的方式
    if (window.getComputedStyle) {
        return getComputedStyle(obj, null)[name];
    } else {
        //IE8方式，没有getComputedStyle()方法
        return obj.currentStyle[name];
    }
}

//定义一个函数，用来向一个元素中添加指定的class属性值
/*
参数：
    obj 要添加class属性的元素
    cn 要添加的class值
*/
function addClass(obj, cn) {
    //检查obj中是否含有cn
    if (!hasClass(obj, cn))
        obj.className += " " + cn;
}

/*
判断一个元素中是否含有指定的class属性值
如果有该class，怎返回true，没有则返回false
参数：
    obj
    cn
*/
function hasClass(obj, cn) {
    //判断obj中是否有cn class
    //创建一个正则表达式
    // var reg = /\bb2\b/;
    var reg = new RegExp("\\b" + cn + "\\b");
    return reg.test(obj.className);
}

/*
删除一个元素中的指定的class属性
*/
function removeClass(obj, cn) {
    //创建一个正则表达式
    var reg = new RegExp("\\b" + cn + "\\b");
    //删除class
    obj.className = obj.className.replace(reg, "");
}

/*

```



```

toggleClass可以用来切换一个类
    如果元素中具有该类，则删除
    如果元素中没有该类，则添加
*/
function toggleClass(obj, cn) {
    //判断obj中是否含有cn
    if (hasClass(obj, cn)) {
        removeClass(obj, cn);
    } else {
        addClass(obj, cn);
    }
}

```

- 定时切换图片

重点：开启定时器之前，需要将上一个定时器关闭。

```


<button id="btn01">开始</button>
<button id="btn02">停止</button>

/*
使图片自动切换
*/
//获取img标签
var img1 = document.getElementById("img1");
//创建一个数组来保存图片的路径
var imgArr = ["img/BB.jpg", "img/巴妈.jpg", "img/宫本.jpg", "img/狐狸.jpg", "img/旧剑.jpg"];
//创建一个变量，用来保存当前图片的索引
var index = 0;

//定义一个变量，用来保存定时器的标识
var timer;

//为btn01绑定一个单击响应函数
var btn01 = document.getElementById("btn01");
btn01.onclick = function () {
    /*
    开启一个定时器，自动切换图片
    */
    /*
    目前，我们每点击一次按钮，就会开启一个定时器，
    点击多次就会开启多个定时器，这就导致图片的切换速度越来越快，
    并且我们只能关闭最后一次开启的定时器
    */

    /*
    在开启定时器之前，需要将上一个定时器关闭
    */
    clearInterval(timer);
    timer = setInterval(function () {

        //索引自增
        index++;

```

```

        // if (index >= imgArr.length) {
        //     index = 0;
        // }
        index = index % imgArr.length;
        //修改img1
        img1.src = imgArr[index];
    }, 1000);
};

var btn02 = document.getElementById("btn02");
btn02.onclick = function () {
    //点击按钮以后停止图片自动切换，关闭定时器
    /*
    clearInterval()可以接受任意参数，
        如果参数是一个有效的定时器的标识，则停止对应的定时器
        如果参数不是一个有效的标识，则什么也不做
    */
    clearInterval(timer);
};

```

- 类的操作

通过style属性来修改元素的样式，每修改一个样式，浏览器就需要重新渲染一次页面。

这样的执行性能非常差，而且这种形式当我们需要修改多个样式时，也不太方便。

```

//渲染三次
box.style.width = "100px";
box.style.height = "100px";
box.style.backgroundColor = "yellow";

```

我们希望一行代码同时修改多个样式

修改box的class属性

我们可以通过修改元素的class属性来间接修改样式。这样，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页面一次，性能较高。

并且这种方式，可以使表现和行为进一步分离，class里面的元素要用空格隔开

四种方法（结合正则表达式）：

hasClass 是否有xx类

addClass 添加类（如果没有xx类，添加，防止类重复）

removeClass 删除类

toggleClass 切换类：如果有xx类，则删除；否则添加。

```

<div id="box" class="b1 b2"></div>

var btn01 = document.getElementById("btn01");
var box = document.getElementById("box");

btn01.onclick = function () {
    //修改box样式
    /*
    通过style属性来修改元素的样式，每修改一个样式，浏览器就需要重新渲染一次页面
    这样的执行性能非常差，而且这种形式当我们需要修改多个样式时，也不太方便
    */
    // box.style.width = "100px";
    // box.style.height = "100px";
    // box.style.backgroundColor = "yellow";

    /*
    我们希望一行代码同时修改多个样式
    */
    //修改box的class属性
    /*
    我们可以通过修改元素的class属性来间接修改样式
    这样，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页面一次，性能较高
    并且这种方式，可以使表现和行为进一步分离
    class里面的元素要用空格隔开
    */
    // box.className += " b2";
    // addClass(box, "b2");
    // alert(hasClass(box, "b2"));
    // removeClass(box, "b2");
    toggleClass(box, "b2");
};

//定义一个函数，用来向一个元素中添加指定的class属性值
/*
参数：
    obj 要添加class属性的元素
    cn 要添加的class值
*/
function addClass(obj, cn) {
    //检查obj中是否含有cn
    if (!hasClass(obj, cn))
        obj.className += " " + cn;
}

/*
判断一个元素中是否含有指定的class属性值
如果有该class，则返回true，没有则返回false
参数：
    obj
    cn
*/
function hasClass(obj, cn) {
    //判断obj中是否有cn class

```

```

    //创建一个正则表达式
    // var reg = /\bb2\b/;
    var reg = new RegExp("\\b" + cn + "\\b");
    return reg.test(obj.className);
}

/*
删除一个元素中的指定的class属性
*/
function removeClass(obj, cn) {
    //创建一个正则表达式
    var reg = new RegExp("\\b" + cn + "\\b");
    //删除class
    obj.className = obj.className.replace(reg, "");
}

/*
toggleClass可以用来切换一个类
    如果元素中具有该类，则删除
    如果元素中没有该类，则添加
*/
function toggleClass(obj, cn) {
    //判断obj中是否含有cn
    if (hasClass(obj, cn)) {
        removeClass(obj, cn);
    } else {
        addClass(obj, cn);
    }
}

```

- 动态轮播图

```

<style>
    * {
        margin: 0;
        padding: 0;
    }

    #outer {
        /*设置宽和高*/
        width: 1044px;
        height: 1448px;
        /*居中*/
        margin: 50px auto;
        /*设置背景颜色*/
        background-color: yellow;
        /*设置padding*/
        padding: 10px 0;
        /*开启相对定位*/
        position: relative;
        /*裁剪溢出的内容*/
        overflow: hidden;
    }

```

```

/*设置imgList*/
#imgList {
    /*去除项目符号*/
    list-style: none;
    /*设置ul的宽度*/
    /* width: 5300px; */
    /*开启绝对定位*/
    position: absolute;
    /*设置偏移量*/
    left: 0;
}

/*设置li浮动*/
#imgList li {
    /*设置浮动*/
    float: left;
    /*设置左右间距*/
    margin: 0 10px;
}

/*设置导航按钮*/
#navDiv {
    /*开启绝对定位*/
    position: absolute;
    /*设置位置*/
    bottom: 25px;
    /*设置left值
        outer 1044px
        navDiv: 5*25 = 125px
        1044-125=919
        919/2=459.5
    */
    /* left: 459px; */
}

#navDiv a {
    /*设置超链接浮动*/
    float: left;
    /*设置超链接宽高*/
    width: 15px;
    height: 15px;
    background-color: red;
    /*设置左右外边距*/
    margin: 0 5px;
    /*设置透明*/
    opacity: 0.5;
}

#navDiv a:hover {
    background-color: black;
}
</style>
<!-- 引入move工具 -->
<script src="./js/tools.js"></script>

<body>

```

```

<!-- 创建一个外部的div，来作为大的容器 -->
<div id="outer">
  <!-- 创建一个ul，用于放置图片 -->
  <ul id="imgList">
    <li>
      
    </li>
    <li>
      
    </li>
    <li>
      
    </li>
    <li>
      
    </li>
    <li>
      
    </li>
    <li>
      
    </li>
  </ul>
  <!-- 创建导航按钮 -->
  <div id="navDiv">
    <a href="javascript:;"></a>
    <a href="javascript:;"></a>
    <a href="javascript:;"></a>
    <a href="javascript:;"></a>
    <a href="javascript:;"></a>
  </div>
</div>
<script>

  var imgList = document.getElementById("imgList");
  //获取页面中所有的图片
  var imgArr = document.getElementsByTagName("img");
  //设置imgList的宽度
  imgList.style.width = 1044 * imgArr.length + "px";

  //设置导航按钮居中
  //获取navDiv
  var navDiv = document.getElementById("navDiv");
  //获取outer
  var outer = document.getElementById("outer");
  //设置navDiv的left值
  navDiv.style.left = (outer.offsetWidth - navDiv.offsetWidth) / 2 + "px";

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

  autoChange();

```

```

//创建一个方法用来设置选中的a
function setA(index) {
    //判断当前索引是否是最后一张图片
    if (index >= imgArr.length - 1) {
        //则将index设为0
        index = 0;
        //此时显示的最后一张图片，而最后一图片和第一张图片相同
        imgList.style.left = 0;
    }

    //遍历所有的a，并将背景颜色变为红色，并把选中的a设置为黑色
    for (var i = 0; i < allA.length; i++) {
        //去掉就是上面的默认样式
        allA[i].style.backgroundColor = "";
    }
    allA[index].style.backgroundColor = "black";
}

var timer;
//创建一个函数，用来开启自动切换图片
function autoChange() {
    //开启定时器，定时切换图片
    timer = setInterval(function () {
        //索引自增index
        index++;

        //判断index值
        index %= imgArr.length;
        //执行动画，切换图片
        move(imgList, "left", -1044 * index, 20, function () {
            //修改导航按钮
            setA(index);
        });
    }, 1000);
}
</script>
</body>

```

9.ES6

 ES6