js特性和差异性

调试:
document.write()

输入框:
var x = prompt('hellow world')
alert(x)

typeof 显示数据类型

toFixed(n) 四舍五入，保留小数，多出来的补零

null、NaN、Infinity、undefined

```
NaN; // NaN表示Not a Number，当无法计算结果时用NaN表示
Infinity; // Infinity表示无限大，当数值超过了JavaScript的Number所能表示的最大值时，就表示为Infinity
null // 表示空值
undefined // 未定义类型，例如key不存的时的obj[key].property
```

另一个例外是`NaN`这个特殊的Number与所有其他值都不相等，包括它自己：

```
NaN === NaN; // false
```

唯一能判断`NaN`的方法是通过`isNaN()`函数：

```
isNaN(NaN); // true
```

字符数字转换为数字的方法

‘1’*1=1

数组和obj比较问题

直接用===比较时，即使数值相同也返回false。

浮点数比较问题

最后要注意浮点数的相等比较：

```
1 / 3 === (1 - 2 / 3); // false
```

这不是JavaScript的设计缺陷。浮点数在运算过程中会产生误差，因为计算机无法精确表示无限循环小数。要比较两个浮点数是否相等，只能计算它们之差的绝对值，看是否小于某个阈值：

```
Math.abs(1 / 3 - (1 - 2 / 3)) < 0.0000001; // true
```

>> 和 >>>

带符号位移case:

1. -7>>1 = -4

    第一步：00000000 00000000 00000000 00000111 （无符号整数）

    第二步：11111111 11111111 11111111 11111001 （-7的表示，第一步求反+1）

    第三步：11111111 11111111 11111111 11111100（带符号位移）

    第四步：00000000 00000000 00000000 00000100 （-1 取反 ）

    第五步：10000000 00000000 00000000 00000100（符号位补1）答案是-4

无符号位移case:

1. -1>>>4 = ox0FFFFFFF

    第一步：00000000 00000000 00000000 00000001 （无符号整数）

    第二步：11111111 11111111 11111111 11111111（-1的表示，第一步求反+1）

    第三步：00001111 11111111 11111111 11111111 （无符号位移）答案是ox0FFFFFFF

===

- ==会隐式转换类型
    - 例如：false==0
- ===不转换类型，比较稳定
- **===不能比较数组类型数据**
- **===可以比较string**

数据结构

Map

这种类型的map和}{key:value}的不同, 其只能通过.get方法获取对应的value, 而不能通过[]获得.

```
// 传入二维数组初始化
let map = new Map([[key, value]]);

// 使用set设置key-value
map.set(key,value);

// 使用has判断是否存在key
map.has(key);

// 使用get获取某个key的value
map.get(key);

// 使用delete删除key
map.delete(key)
```

Set

```
let set = new Set(list);
set.add(key);
set.delete(key);
```

`Set`与`Array`类似，但`Set`没有索引，因此回调函数的前两个参数都是元素本身：

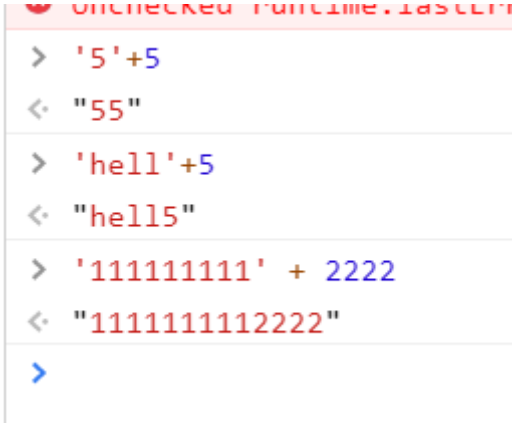
```
var s = new Set(['A', 'B', 'C']);
s.forEach(function (element, sameElement, set) {
    console.log(element);
});
```

`Map`的回调函数参数依次为`value`、`key`和`map`本身：

```
var m = new Map([[1, 'x'], [2, 'y'], [3, 'z']]);
m.forEach(function (value, key, map) {
    console.log(value);
});
```

字符串差异

- 字符和整数相加



- js支持类似python的字符串简易组合

```
var name = '小明';
var age = 20;
var message = `你好, ${name}, 你今年${age}岁了!`;
alert(message);
```

- js和c++不一样，不能直接用str[i]=X修改某个字符、考虑用replace修改字符：

```
一、替换字符串中所有为a的字符为e
let str = 'abcabcabc';
str = str.replace(/a/g, 'e');
console.log(str);
// 打印结果： ebcebcebc
二、替换字符串中第一个为a的字符为e
let str = 'abcabcabc';
str = str.replace(/a/, 'e');
console.log(str);
// 打印结果： ebcabcabc
```

获得字符串索引 indexOf charAt

解决方案一：使用 indexOf 方法（数组也适用）

```
let str = "中国伟大"
console.log(str.indexOf("国")); // 1
console.log(str.indexOf("洲")); // -1
```

解决方案二：使用 search 方法（数组不适用）

search方法 和 indexOf方法差不多，有就返回对应索引，没有就返回-1

```
let str = "qwertyuiop"
console.log(str.search('t')); // 4
console.log(str[str.search('t')]); //t  通过索引可以访问到对应的值
console.log(str.search("k")); // -1
```

解决方案三：使用 includes 方法（数组也适用）

判断字符串中是否包含某个字，包含就返回true，不包含就返回false；

```
let str = "你好世界"
console.log(str.includes("好")); // true
console.log( !str.includes("好") ); // false
console.log(str.includes("啊")); // false
```

补充：charAt查询方法-根据索引查对应下标的值

字符串的str.charAt(index)方法和直接使用str[index]方法很像。只需要传入一个索引，就能够获取访问到对应字符串索引的值。

```
let str = "早上好，又是新的一天"
console.log( str.charAt(2) ); // 好
console.log( str[2] ); // 好
```

字符串子串和切割 slice

字符串切割之-slice方法

indexOf和search可以得到对应的索引 --> 通过索引表示从第几位开始对相应的字符串进行切割。

slice[index1,index2)

- **提供两个index**
- **左闭右开**

```
let str = "helloWorld"
console.log(str.slice(5,str.length)); //World  截取第五位到最后一位
console.log(str.slice(5)); //World  如果只写一个索引参数，就默认从此索引开始截取，一直截取到最后一个
onsole.log(str.slice(5,-2)); //Wor 若参数为负数，就从结尾处开始计算
```

字符串切割之-substring方法

substring用法类似slice。

```
let str = "qwertyuiop"
console.log( str.substring(3,str.length) );// rtyuiop
console.log( str.substring(3)); // rtyuiop  同样的和slice一样，如果只传入一个索引就会从当前索引一直截取到最后
console.log( str.substring(5,-2) ); //qwert  官网上说substring不支持负数，貌似也支持，不过以后多用slice
```

字符串切割固定长度之-substr方法

如果想要从某个位置，截取固定长度的字符串.。

```
 let str = "打工人加油鸭"
console.log( str.substr(1,4) ); // 工人加油
```

- **str.substr(i)：获取从i到结尾的子串**

`substr()`、`substring()`和`slice()`都是用于从字符串中提取子串的方法，但它们有一些差异：

- `substr(startIndex, length)` 方法从字符串的 `startIndex` 位置开始，提取 `length` 个字符组成的子串。如果省略 `length` 参数，则会提取从 `startIndex` 位置开始的所有字符，直到字符串的结尾。`substr()` 方法的第一个参数可以是负数，表示从字符串的末尾开始计算的位置。
- `substring(startIndex, endIndex)` 方法从字符串的 `startIndex` 位置开始，提取到 `endIndex-1` 位置的所有字符组成的子串。`substring()` 方法的参数必须是正数，如果省略 `endIndex` 参数，则会提取从 `startIndex` 位置开始的所有字符，直到字符串的结尾。如果 `startIndex` 大于 `endIndex`，`substring()` 方法会自动调整两个参数的位置。
- `slice(startIndex, endIndex)` 方法从字符串的 `startIndex` 位置开始，提取到 `endIndex-1` 位置的所有字符组成的子串。`slice()` 方法的参数可以是负数，表示从字符串的末尾开始计算的位置。如果省略 `endIndex` 参数，则会提取从 `startIndex` 位置开始的所有字符，直到字符串的结尾。如果 `startIndex` 大于 `endIndex`，`slice()` 方法会返回一个空字符串。

另外，这些方法返回的都是新的字符串，而不是修改原字符串。

startsWith方法/endsWith方法

字符串以某某开头/某某结尾-startsWith方法/endsWith方法

```
        let str = "computer"
        console.log( str.startsWith("c") ); // true
        console.log( str.startsWith("C") ); // false
        console.log( str.endsWith("r") ); // true
        console.log( str.endsWith("R") ); // false
        /* 注意事项：
            startsWith以什么什么开头，endsWith以什么什么结尾
            这两个方法是区分大小写的，可以通过toUpperCase、toLowerCase
            统一转换成大写或者小写以后，再使用startsWith或endsWith方法
        */
```

字符串转数组方法之-split方法

```
let str = "美好的一天"
console.log(str.split()); // ["美好的一天"]
console.log(str.split(""));// ["美", "好", "的", "一", "天"]
let str1 = "热，爱，和，平"
console.log(str1.split("，"));// ["热", "爱", "和", "平"]
```

大小写转换-针对字母或全体 toUpperCase

```
// Get fieldname
function getFieldName(input) {
  return input.id.charAt(0).toUpperCase() + input.id.slice(1);
}
```

数组操作

尾增：push和pop

`push()` 向 `Array` 的末尾添加若干元素，`pop()` 则把 `Array` 的最后一个元素删除掉：

```
var arr = [1, 2];
arr.push('A', 'B'); // 返回Array新的长度: 4
arr; // [1, 2, 'A', 'B']
arr.pop(); // pop()返回'B'
arr; // [1, 2, 'A']
arr.pop(); arr.pop(); arr.pop(); // 连续pop 3次
arr; // []
arr.pop(); // 空数组继续pop不会报错，而是返回undefined
arr; // []
```

头增：unshift和shift

如果要往 `Array` 的头部添加若干元素，使用 `unshift()` 方法，`shift()` 方法则把 `Array` 的第一个元素删掉：

```
var arr = [1, 2];
arr.unshift('A', 'B'); // 返回Array新的长度: 4
arr; // ['A', 'B', 1, 2]
arr.shift(); // 'A'
arr; // ['B', 1, 2]
arr.shift(); arr.shift(); arr.shift(); // 连续shift 3次
arr; // []
arr.shift(); // 空数组继续shift不会报错，而是返回undefined
arr; // []
```

删1，改: splice

- **删除：splice(index,len)**
- 删除后再插入：splice(idx,len,a,b,c..)
- 插入：splice(idx,0,a,b,c,...)

```
var arr = ['Microsoft', 'Apple', 'Yahoo', 'AOL', 'Excite', 'Oracle'];
// 从索引2开始删除3个元素,然后再添加两个元素:
arr.splice(2, 3, 'Google', 'Facebook'); // 返回删除的元素 ['Yahoo', 'AOL',
'Excite']
arr; // ['Microsoft', 'Apple', 'Google', 'Facebook', 'Oracle']
// 只删除,不添加:
arr.splice(2, 2); // ['Google', 'Facebook']
arr; // ['Microsoft', 'Apple', 'Oracle']
// 只添加,不删除:
arr.splice(2, 0, 'Google', 'Facebook'); // 返回[],因为没有删除任何元素
arr; // ['Microsoft', 'Apple', 'Google', 'Facebook', 'Oracle']
```

删2：filter

例如，在一个Array中，删掉偶数，只保留奇数，可以这么写

```
var arr = [1, 2, 4, 5, 6, 9, 10, 15];
var r = arr.filter(function (x) {
    return x % 2 !== 0;
});

r; // [1, 5, 9, 15]
```

把一个Array中的空字符串删掉，可以这么写：

```
var arr = ['A', '', 'B', null, undefined, 'C', '  '];
var r = arr.filter(function (s) {
    return s && s.trim(); // 注意：IE9以下的版本没有trim()方法
});
arr; // ['A', 'B', 'C']
```

trim()函数去掉字符串首尾空白字符

利用filter，可以巧妙地去除Array的重复元素：

```
arr = ['apple', 'strawberry', 'banana', 'pear', 'apple', 'orange', 'orange',
'strawberry']
    arr = arr.filter(function (element, index, self) {
    return self.indexOf(element) === index;
});
```

indexOf总是返回第一个元素的位置，后续的重复元素位置与indexOf返回的位置不相等，因此被filter滤掉了，所以重复的元素仅会保留第一个位置的元素

查: indexOf

与String类似，`Array`也可以通过`indexOf()`来搜索**第一个**指定的元素的位置：

```
var arr = [10, 20, '30', 'xyz'];
arr.indexOf(10); // 元素10的索引为0
arr.indexOf(20); // 元素20的索引为1
arr.indexOf(30); // 元素30没有找到，返回-1
arr.indexOf('30'); // 元素'30'的索引为2
```

注意了，数字`30`和字符串`'30'`是不同的元素。

注意:

二维数组中, 无法用indexOf判断是否存在某个一维数组元素, 因为比较的是地址.

for in循环

遍历出来的i是数组的index

（一般用于对象，用于对key操作）

由于`Array`也是对象，而它的每个元素的索引被视为对象的属性，因此，`for ... in`循环可以直接循环出`Array`的索引：

```
var a = ['A', 'B', 'C'];
for (var i in a) {
    console.log(i); // '0', '1', '2'
    console.log(a[i]); // 'A', 'B', 'C'
}
```

请注意，`for ... in`对`Array`的循环得到的是`String`而不是`Number`。

for ... of`循环

遍历出来的i是数组的元素

具有`iterable`类型的集合可以通过新的`for ... of`循环来遍历。

用`for ... of`循环遍历集合，用法如下：

```
var a = ['A', 'B', 'C'];
var s = new Set(['A', 'B', 'C']);
var m = new Map([[1, 'x'], [2, 'y'], [3, 'z']]);
for (var x of a) { // 遍历Array
    console.log(x);
}
for (var x of s) { // 遍历Set
    console.log(x);
}
for (var x of m) { // 遍历Map
    console.log(x[0] + '=' + x[1]);
}
```

forEach遍历

```
a.forEach(function (element, index, array) {
    // element: 指向当前元素的值
    // index: 指向当前索引
    // array: 指向Array对象本身
    console.log(element + ', index = ' + index);
});
```

总结:

1. 对象 obj 用for key in
2. 数组 用for item of/ [..].forEach(item => {})

concat

`concat()` 方法把当前的 `Array` 和另一个 `Array` 连接起来，并返回一个新的 `Array`：

```
var arr = ['A', 'B', 'C'];
var added = arr.concat([1, 2, 3]);
added; // ['A', 'B', 'C', 1, 2, 3]
arr; // ['A', 'B', 'C']
```

请注意，`concat()` 方法并没有修改当前 `Array`，而是返回了一个新的 `Array`。

也可以先分别join为字符串，在使用+拼接，再split为数组

join

`join()` 方法是一个非常实用的方法，它把当前 `Array` 的每个元素都用指定的字符串连接起来，然后返回连接后的字符串：

```
var arr = ['A', 'B', 'C', 1, 2, 3];
arr.join('-'); // 'A-B-C-1-2-3'
```

使用join把数组连接成字符串，使用split把字符串拆分为

sort

`sort()` 可以对当前 `Array` 进行排序，**它会直接修改当前 `Array` 的元素位置**，直接调用时，按照默认顺序排序：

```
var arr = ['B', 'C', 'A'];
arr.sort();
arr; // ['A', 'B', 'C']
```

能否按照我们自己指定的顺序排序呢？完全可以，里面传递一个lambda表达式即可。

例子：

```
let order = ['论坛','微信','微博']
let tableData=[{name:'微博',source:'weibo'},{name:'微信',source:'weixin'},
{name:'论坛',source:'luntan'}]
tableData = tableData.sort((a, b) => {
        return order.indexOf(a.name) - order.indexOf(b.name)
 })
```

reverse

`reverse()`把整个`Array`的元素给调个个，也就是反转：

```
var arr = ['one', 'two', 'three'];
arr.reverse();
arr; // ['three', 'two', 'one']
```

对象操作

注意: 对象内部的是property, 而要获取这个property的value时需要用'property'来表示属性的值才行.

添加：obj['key']=value或者obj.key=value

JavaScript对象的所有属性都是字符串，不过属性对应的值可以是任意数据类型

添加：Object.defineProperty(obj,'key',{value: v})

(**相当于 obj['key']=value**)

writable： 设置该属性是否允许修改，true表示允许，false表示不允许。如果不写，默认为false。

Object.defineProperty(obj, "name", { value: "张三", "writable": false });

添加：Object.defineProperties()

Object.defineProperties(对象名, { 属性名1: { value1: 值1, writable: true }, 属性名2: { value2: 值2, writable: false } });

删除：delete

查：hasOwnProperty()

- 查询obj是否有某个属性；查询map是否有某个key
- 返回bool值
- 作用：初始化是如果判断key不存在，则不进行下面的操作

查: 'key' in obj

遍历：for in

把一个对象的所有属性依次循环出来

```
for (const key in obj) {
    console.log(key+":"+ obj[key]);
};
```

遍历：for of + Object.keys()

```
for (const key of Object.keys(obj)) {
    console.log(key + ":" + obj[key]);
};
```

遍历：for of + Object.entries()

```
for (const [key, value] of Object.entries(obj)) {
    console.log(key + ":" + value);
};
```

```
const obj = {
    a: "have",
    b: "some",
    c: "thing",
    d: "to",
    e: "do"
};
console.log(Object.entries(obj));//[['a', 'have'], ['b', 'some'], ['c', 'thing'],
['d', 'to'], ['e', 'do']]
```

遍历：for of + Object.values()

```
for (const val of Object.values(obj)) {
    console.log(value);
};
```

例子

```
var xiaoming = {
    name: '小明',
    birth: 1990,
    school: 'No.1 Middle School',
    height: 1.70,
    weight: 65,
    score: null
};
// 增加
xiaoming.age; // undefined
xiaoming.age = 18; // 新增一个age属性
// 删除
xiaoming.age; // 18
delete xiaoming.age; // 删除age属性
xiaoming.age; // undefined
delete xiaoming['name']; // 删除name属性
xiaoming.name; // undefined
delete xiaoming.school; // 删除一个不存在的school属性也不会报错
// 判断
'name' in xiaoming; // true
'grade' in xiaoming; // false
/*所有类继承于object类，里面实现了toString属性*/
'toString' in xiaoming; // true
xiaoming.hasOwnProperty('name'); // true
xiaoming.hasOwnProperty('toString'); // false
```

函数

arguments

JavaScript还有一个免费赠送的关键字`arguments`，它只在函数内部起作用，并且永远指向当前函数的调用者传入的所有参数。`arguments`类似`Array`但它不是一个`Array`：

```
'use strict'
function foo(x) {
    console.log('x = ' + x); // 10
    for (var i=0; i<arguments.length; i++) {
        console.log('arg ' + i + ' = ' + arguments[i]); // 10, 20, 30
    }
}
foo(10, 20, 30);

x = 10
arg 0 = 10
arg 1 = 20
arg 2 = 30
```

```
// foo(a[, b], c)
// 接收2~3个参数，b是可选参数，如果只传2个参数，b默认为null：
function foo(a, b, c) {
    if (arguments.length === 2) {
        // 实际拿到的参数是a和b，c为undefined
        c = b; // 把b赋给c
        b = null; // b变为默认值
    }
    // ...
}
```

根据传入参数的个数动态调整内部逻辑

function(a,b,...c)接受多余的参数为c

返回值的特点

- 默认返回值后面加 ';'

```
function foo() {
    return { name: 'foo' };
}
```

```
function foo() {
    return { // 这里不会自动加分号，因为{表示语句尚未结束
        name: 'foo'
    };
}
```

名字空间

全局变量会绑定到`window`上，不同的JavaScript文件如果使用了相同的全局变量，或者定义了相同名字的顶层函数，都会造成命名冲突，并且很难被发现。

减少冲突的一个方法是把自己的所有变量和函数全部绑定到一个全局变量中。例如：

```
// 唯一的全局变量MYAPP:
var MYAPP = {};

// 其他变量:
MYAPP.name = 'myapp';
MYAPP.version = 1.0;

// 其他函数:
MYAPP.foo = function () {
    return 'foo';
};
```

把自己的代码全部放入唯一的名字空间`MYAPP`中，会大大减少全局变量冲突的可能。

许多著名的JavaScript库都是这么干的：jQuery，YUI，underscore等等。

let/var/const

由于JavaScript的变量作用域实际上是函数内部，我们在`for`循环等语句块中是无法定义具有局部作用域的变量的：

```
'use strict';

function foo() {
    for (var i=0; i<100; i++) {
        //
    }
    i += 100; // 仍然可以引用变量i
}
```

为了解决块级作用域，ES6引入了新的关键字`let`，用`let`替代`var`可以申明一个块级作用域的变量：

```
'use strict';

function foo() {
    var sum = 0;
    for (let i=0; i<100; i++) {
        sum += i;
    }
    // SyntaxError:
    i += 1;
}
```

常量

由于`var`和`let`申明的是变量，如果要申明一个常量，在ES6之前是不行的，我们通常用全部大写的变量来表示“这是一个常量，不要修改它的值”：

```
var PI = 3.14;
```

ES6标准引入了新的关键字`const`来定义常量，`const`与`let`都具有块级作用域：

```
'use strict';

const PI = 3.14;
PI = 3; // 某些浏览器不报错，但是无效果！
PI; // 3.14
```

变量作用域与解构赋值

对数组元素进行解构赋值时，多个变量要用`[...]`括起来。

如果数组本身还有嵌套，也可以通过下面的形式进行解构赋值，注意嵌套层次和位置要保持一致：

```
let [x, [y, z]] = ['hello', ['JavaScript', 'ES6']];
x; // 'hello'
y; // 'JavaScript'
z; // 'ES6'
```

解构赋值还可以忽略某些元素：

```
let [, , z] = ['hello', 'JavaScript', 'ES6']; // 忽略前两个元素，只对z赋值第三个元素
z; // 'ES6'
```

如果需要从一个对象中取出若干属性，也可以使用解构赋值，便于快速获取对象的指定属性：

```
'use strict';

var person = {
    name: '小明',
    age: 20,
    gender: 'male',
    passport: 'G-12345678',
    school: 'No.4 middle school'
};
var {name, age, passport} = person;
```

对一个对象进行解构赋值时，同样可以直接对嵌套的对象属性进行赋值，只要保证对应的层次是一致的：

```
var person = {
    name: '小明',
    age: 20,
    gender: 'male',
    passport: 'G-12345678',
    school: 'No.4 middle school',
    address: {
        city: 'Beijing',
        street: 'No.1 Road',
        zipcode: '100001'
    }
};
var {name, address: {city, zip}} = person;
name; // '小明'
city; // 'Beijing'
zip; // undefined, 因为属性名是zipcode而不是zip
// 注意: address不是变量，而是为了让city和zip获得嵌套的address对象的属性:
address; // Uncaught ReferenceError: address is not defined
```

使用解构赋值对对象属性进行赋值时，如果对应的属性不存在，变量将被赋值为`undefined`，这和引用一个不存在的属性获得`undefined`是一致的。如果要使用的变量名和属性名不一致，可以用下面的语法获取：

```
var person = {
    name: '小明',
    age: 20,
```

```
    gender: 'male',
    passport: 'G-12345678',
    school: 'No.4 middle school'
};

// 把passport属性赋值给变量id:
let {name, passport:id} = person;
name; // '小明'
id; // 'G-12345678'
// 注意: passport不是变量，而是为了让变量id获得passport属性:
passport; // Uncaught ReferenceError: passport is not defined
```

解构赋值还可以使用默认值，这样就避免了不存在的属性返回`undefined`的问题：

```
var person = {
    name: '小明',
    age: 20,
    gender: 'male',
    passport: 'G-12345678'
};

// 如果person对象没有single属性，默认赋值为true:
var {name, single=true} = person;
name; // '小明'
single; // true
```

有些时候，如果变量已经被声明了，再次赋值的时候，正确的写法也会报语法错误：

```
// 声明变量:
var x, y;
// 解构赋值:
{x, y} = { name: '小明', x: 100, y: 200};
// 语法错误: Uncaught SyntaxError: Unexpected token =
```

这是因为JavaScript引擎把`{`开头的语句当作了块处理，于是`=`不再合法。解决方法是用小括号括起来：

```
({x, y} = { name: '小明', x: 100, y: 200});
```

使用场景

解构赋值在很多时候可以大大简化代码。例如，交换两个变量`x`和`y`的值，可以这么写，不再需要临时变量：

```
var x=1, y=2;
[x, y] = [y, x]
```

快速获取当前页面的域名和路径：

```
var {hostname:domain, pathname:path} = location;
```

如果一个函数接收一个对象作为参数，那么，可以使用解构直接把对象的属性绑定到变量中。例如，下面的函数可以快速创建一个`Date`对象：

```
function buildDate({year, month, day, hour=0, minute=0, second=0}) {
    return new Date(year + '-' + month + '-' + day + ' ' + hour + ':' + minute +
':' + second);
}
```

它的方便之处在于传入的对象只需要`year`、`month`和`day`这三个属性：

```
buildDate({ year: 2017, month: 1, day: 1 });
// Sun Jan 01 2017 00:00:00 GMT+0800 (CST)
```

也可以传入`hour`、`minute`和`second`属性：

```
buildDate({ year: 2017, month: 1, day: 1, hour: 20, minute: 15 });
// Sun Jan 01 2017 20:15:00 GMT+0800 (CST)
```

map、reduce、filter、sort

map:

- 是一个property
- 传入函数
- 返回加工过的结果

```
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9];
arr.map(String); // ['1', '2', '3', '4', '5', '6', '7', '8', '9']
```

例子:

```
输入：['adam', 'LISA', 'barT']，输出：['Adam', 'Lisa', 'Bart']。

'use strict';

function normalize(arr) {
return arr.map(x=>(x.substr(0,1).toUpperCase()+x.substr(1).toLowerCase()))
}
```

```
利用map和reduce操作实现一个string2int()函数：

'use strict';

function string2int(s) {
var x = s.split('');
return x.map(a=>a*1).reduce((a,b)=>a*10+b);
}
```

注意:

由于`map()`接收的回调函数可以有3个参数：`callback(currentValue, index, array)`，通常我们仅需要第一个参数，而忽略了传入的后面两个参数。不幸的是，`parseInt(string, radix)`没有忽略第二个参数，导致实际执行的函数分别是：

- parseInt('1', 0); // 1, 按十进制转换
- parseInt('2', 1); // NaN, 没有一进制
- parseInt('3', 2); // NaN, 按二进制转换不允许出现3

可以改为`r = arr.map(Number);`，因为`Number(value)`函数仅接收一个参数。

reduce

再看reduce的用法。Array的`reduce()`把一个函数作用在这个`Array`的`[x1, x2, x3...]`上，这个函数必须接收两个参数，`reduce()`把结果继续和序列的下一个元素做累积计算，其效果就是：

```
[x1, x2, x3, x4].reduce(f) = f(f(f(x1, x2), x3), x4)
```

比方说对一个`Array`求和，就可以用`reduce`实现：

```
var arr = [1, 3, 5, 7, 9];
arr.reduce(function (x, y) {
    return x + y;
}); // 25
```

filter

- **返回值: 想要保留下来的元素需要满足的条件**

filter也是一个常用的操作，它用于把`Array`的某些元素过滤掉，然后返回剩下的元素。

和`map()`类似，`Array`的`filter()`也接收一个函数。和`map()`不同的是，`filter()`把传入的函数依次作用于每个元素，然后根据返回值是`true`还是`false`决定保留还是丢弃该元素。

例如，在一个`Array`中，删掉偶数，只保留奇数，可以这么写：

```
var arr = [1, 2, 4, 5, 6, 9, 10, 15];
var r = arr.filter(function (x) {
    return x % 2 !== 0;
});
r; // [1, 5, 9, 15]
```

把一个`Array`中的空字符串删掉，可以这么写：

```
var arr = ['A', '', 'B', null, undefined, 'C', '  '];
var r = arr.filter(function (s) {
    return s && s.trim(); // 注意：IE9以下的版本没有trim()方法
});
r; // ['A', 'B', 'C']
```

可见用`filter()`这个高阶函数，关键在于正确实现一个“筛选”函数。

回调函数

`filter()`接收的回调函数，其实可以有多个参数。通常我们仅使用第一个参数，表示`Array`的某个元素。回调函数还可以接收另外两个参数，表示元素的位置和数组本身：

```
var arr = ['A', 'B', 'C'];
var r = arr.filter(function (element, index, self) {
    console.log(element); // 依次打印'A', 'B', 'C'
    console.log(index); // 依次打印0, 1, 2
    console.log(self); // self就是变量arr
    return true;
});
```

利用`filter`，可以巧妙地去除`Array`的重复元素：

过滤素数

```
return arr.filter(x=>{
     let count=0;
      for(let i=1;i<=x;i++){
       if(x%i===0){
        count++;
    }
}
return count===2?true:false;
});

return arr.filter((value, index, array) => {
    // return 素数
    for (let i = 2; i <= Math.floor(value / 2); i++) {
        if (value % i === 0) {
            return false
        }
    }
return value !== 1;
});
```

sort

`Array`的`sort()`方法默认把所有元素先转换为String再排序，结果`'10'`排在了`'2'`的前面，因为字符`'1'`比字符`'2'`的ASCII码小。

`sort()`方法也是一个高阶函数，它还可以接收一个比较函数来实现自定义**从大到小排序。**

```
arr.sort(function (x, y) {
    if (x < y) {
        return -1;
    }
    if (x > y) {
        return 1;
    }
    return 0;
});
```

```
arr.sort(function (s1, s2) {
    x1 = s1.toUpperCase();
    x2 = s2.toUpperCase();
    if (x1 < x2) {
        return -1;
    }
    if (x1 > x2) {
        return 1;
    }
    return 0;
}); // ['apple', 'Google', 'Microsoft']
```

every函数

every()方法可以判断数组的所有元素是否满足测试条件。

例如，给定一个包含若干字符串的数组，判断所有字符串是否满足指定的测试条件：

'use strict';

var arr = ['Apple', 'pear', 'orange'];
console.log(arr.every(function (s) {
    return s.length > 0;
})); // true, 因为每个元素都满足s.length>0

console.log(arr.every(function (s) {
    return s.toLowerCase() === s;
})); // false, 因为不是每个元素都全部是小写

dom操作

对html操作

查找 HTML 元素

方法	描述
document.getElementById(*id*)	通过元素 id 来查找元素-结果唯一
document.getElementsByTagName(*name*)	通过标签名来查找元素-返回数组
document.getElementsByClassName(*name*)	通过类名来查找元素-返回数组
document.querySelectorAll()	可以搜索p标签、id、class
document.querySelector()	

document.forms('表单的id')	配合.elements[i]得到表单中的所有dom项
	`<input type="text" id="fname" name="fname" value="Bill">`
	`<input type="text" id="lname" name="lname" value="Gates">`
	`<input type="submit" value="提交">`

elementNode.parentElement　　// 父节点标签元素

elementNode.children　　// 所有子标签

elementNode.firstElementChild　　// 第一个子标签元素

elementNode.lastElementChild　　// 最后一个子标签元素

elementNode.nextElementtSibling　　// 下一个兄弟标签元素

elementNode.previousElementSibling　// 上一个兄弟标签元素

使用getElementsByTagName获取container内的所有p标签：

```
let container = document.getElementById('container');
let allP = container.getElementByTagName('p');
let num = allP.length; // getElementByTagName返回的是数组
for (let i = 0; i < num; i++) {
    allP[i].innerHtml = i;
}
```

使用document.forms获取表单内的所有dom：

```
<!DOCTYPE html>
<html>
<body>
<h1>使用document.forms 查找 HTML 元素</h1>
<form id="frm1" action="/demo/demo_form.asp">
  <label for="fname">First name:</label>
  <input type="text" id="fname" name="fname" value="Bill"><br>
  <label for="lname">Last name:</label>
  <input type="text" id="lname" name="lname" value="Gates"><br><br>
```

```
  <input type="submit" value="提交">
</form>
<p>单击“试一试”按钮，显示表单中每个元素的值。</p>
<button onclick="myFunction()">试一试</button>
<p id="demo"></p>
<script>
function myFunction() {
  var x = document.forms["frm1"];
  var text = "";
  var i;
  for (i = 0; i < x.length ;i++) {
    text += x.elements[i].value + "<br>";
    console.log(x.elements[i].value)
  }
  document.getElementById("demo").innerHTML = text;
}
</script>
</body>
</html>
```

改变 HTML 元素

方法	描述
element.innerHTML = *new html content*	改变元素的 inner HTML
element.attribute = *new value*	改变 HTML 元素的属性值 <img id="myImage" src="smiley.gif"> document.getElementById("myImage").src = "landscape.jpg";
element.setAttribute(*attribute*, *value*)	改变 HTML 元素的属性值
element.style.property = *new style*	改变 HTML 元素的样式
element.classList.add('new-class-name')	添加新的属性名字-交互时满足条件的就添加一个类，用css设置不同格式

添加和删除元素

方法	描述
document.createElement(*element*)	创建 HTML 元素

方法	描述
document.removeChild(*element*)	删除 HTML 元素； const child = document.getElementById("p1");//得到孩子child.parentNode.removeChild(child); //先得到父亲，再删除孩子
document.appendChild(*element*)、document.insertBefore()	添加 HTML 元素
document.replaceChild(*element*)	替换 HTML 元素
document.write(*text*)	写入 HTML 输出流。参数是str
document.createTextNode("xxx")	生成可插入的文字，通过appendChild可插入p标签中
document.querySelector('xxx').remove();	删除已有节点

给span加上js点击交互

```
document.querySelector('.go-right').onclick = function() {
    // 是否为12月
    if (curMonth === 11) {
        curYear++;
    }
    curMonth = (curMonth + 1) % 12;
    generateCalendar(curMonth, curYear);
}
```

创建div并插入、修改属性

```
let generateCalendar = (month, year) => {
    let cal_month = document.querySelector('#month');
    let cal_year = document.querySelector('#year');
    let cal_days = document.querySelector('.calendar-days');
    cal_year.innerHTML = year;
    cal_month.innerHTML = month + 1;
    cal_days.innerHTML = '';
    let first_day = (new Date(year,month)).getDay();
    let days_of_month = [...]
    let loop_nums = first_day + days_of_month[month];
    let cur_date = new Date();
    for (let i = 0 ; i < loop_nums; i++) {
        let day = document.createElement('div');
        if (i >= first_day) {
            day.innerHTML = i - first_day + 1;
            if (i - first_day+1 === cur_date.getDate()
            && year === cur_date.getFullYear()
            && month === cur_date.getMonth()
```

```
            ) {
                day.classList.add('cur_day')
            }
        }
        cal_days.appendChild(day);
    }
}
```

修改表单

用JavaScript操作表单和操作DOM是类似的，因为表单本身也是DOM树。

不过表单的输入框、下拉框等可以接收用户输入，所以用JavaScript来操作表单，可以获得用户输入的内容，或者对一个输入框设置新的内容。

HTML表单的输入控件主要有以下几种：

- 文本框，对应的`<input type="text">`，用于输入文本；
- 口令框，对应的`<input type="password">`，用于输入口令；
- 单选框，对应的`<input type="radio">`，用于选择一项；
- 复选框，对应的`<input type="checkbox">`，用于选择多项；
- 下拉框，对应的`<select>`，用于选择一项；
- 隐藏文本，对应的`<input type="hidden">`，用户不可见，但表单提交时会把隐藏文本发送到服务器。

获取值

如果我们获得了一个`<input>`节点的引用，就可以直接调用`value`获得对应的用户输入值：

```
// <input type="text" id="email">
var input = document.getElementById('email');
input.value; // '用户输入的值'
```

这种方式可以应用于`text`、`password`、`hidden`以及`select`。但是，对于单选框和复选框，`value`属性返回的永远是HTML预设的值，而我们需要获得的实际是用户是否“勾上了”选项，所以应该用`checked`判断：

```
// <label><input type="radio" name="weekday" id="monday" value="1">
Monday</label>
// <label><input type="radio" name="weekday" id="tuesday" value="2">
Tuesday</label>
var mon = document.getElementById('monday');
var tue = document.getElementById('tuesday');
mon.value; // '1'
tue.value; // '2'
mon.checked; // true或者false
tue.checked; // true或者false
```

设置值

设置值和获取值类似，对于`text`、`password`、`hidden`以及`select`，直接设置`value`就可以：

```
// <input type="text" id="email">
var input = document.getElementById('email');
input.value = 'test@example.com'; // 文本框的内容已更新
```

对于单选框和复选框，设置`checked`为`true`或`false`即可。

HTML5控件

HTML5新增了大量标准控件，常用的包括`date`、`datetime`、`datetime-local`、`color`等，它们都使用`<input>`标签：

```
<input type="date" value="2021-12-02">
```

```
<input type="datetime-local" value="2021-12-02T20:21:12">
```

```
<input type="color" value="#ff0000">
```

不支持HTML5的浏览器无法识别新的控件，会把它们当做`type="text"`来显示。支持HTML5的浏览器将获得格式化的字符串。例如，`type="date"`类型的`input`的`value`将保证是一个有效的`YYYY-MM-DD`格式的日期，或者空字符串。

提交表单

最后，JavaScript可以以两种方式来处理表单的提交（AJAX方式在后面章节介绍）。

方式一是通过`<form>`元素的`submit()`方法提交一个表单，例如，响应一个`<button>`的`click`事件，在JavaScript代码中提交表单：

```
<!-- HTML -->
<form id="test-form">
    <input type="text" name="test">
    <button type="button" onclick="doSubmitForm()">Submit</button>
</form>

<script>
function doSubmitForm() {
    var form = document.getElementById('test-form');
    // 可以在此修改form的input...
    // 提交form:
    form.submit();
}
</script>
```

这种方式的缺点是扰乱了浏览器对form的正常提交。浏览器默认点击`<button type="submit">`时提交表单，或者用户在最后一个输入框按回车键。因此，第二种方式是响应`<form>`本身的`onsubmit`事件，在提交form时作修改：

```
```

```
<!-- HTML -->
<form id="test-form" onsubmit="return checkForm()">
    <input type="text" name="test">
    <button type="submit">Submit</button>
</form>

<script>
function checkForm() {
    var form = document.getElementById('test-form');
    // 可以在此修改form的input...
    // 继续下一步:
    return true;
}
</script>
```

注意要`return true`来告诉浏览器继续提交，如果`return false`，浏览器将不会继续提交form，这种情况通常对应用户输入有误，提示用户错误信息后终止提交form。

在检查和修改`<input>`时，要充分利用`<input type="hidden">`来传递数据。

例如，很多登录表单希望用户输入用户名和口令，但是，安全考虑，提交表单时不传输明文口令，而是口令的MD5。普通JavaScript开发人员会直接修改`<input>`：

```
<!-- HTML -->
<form id="login-form" method="post" onsubmit="return checkForm()">
    <input type="text" id="username" name="username">
    <input type="password" id="password" name="password">
    <button type="submit">Submit</button>
</form>

<script>
function checkForm() {
    var pwd = document.getElementById('password');
    // 把用户输入的明文变为MD5:
    pwd.value = toMD5(pwd.value);
    // 继续下一步:
    return true;
}
</script>
```

这个做法看上去没啥问题，但用户输入了口令提交时，口令框的显示会突然从几个`*`变成32个`*`（因为MD5有32个字符）。

要想不改变用户的输入，可以利用`<input type="hidden">`实现：

```
<!-- HTML -->
<form id="login-form" method="post" onsubmit="return checkForm()">
    <input type="text" id="username" name="username">
    <input type="password" id="input-password">
    <input type="hidden" id="md5-password" name="password">
    <button type="submit">Submit</button>
</form>

<script>
function checkForm() {
```

```
    var input_pwd = document.getElementById('input-password');
    var md5_pwd = document.getElementById('md5-password');
    // 把用户输入的明文变为MD5:
    md5_pwd.value = toMD5(input_pwd.value);
    // 继续下一步:
    return true;
}
</script>
```

注意到`id`为`md5-password`的`<input>`标记了`name="password"`，而用户输入的`id`为`input-password`的`<input>`没有`name`属性。**没有`name`属性的`<input>`的数据不会被提交。**

练习

利用JavaScript检查用户注册信息是否正确，在以下情况不满足时报错并阻止提交表单：

- 用户名必须是3-10位英文字母或数字；
- 口令必须是6-20位；
- 两次输入口令必须一致。

```
<!-- HTML结构 -->
<form id="test-register" action="#" target="_blank" onsubmit="return
checkRegisterForm()">
    <p id="test-error" style="color:red"></p>
    <p>
        用户名: <input type="text" id="username" name="username">
    </p>
    <p>
        口令: <input type="password" id="password" name="password">
    </p>
    <p>
        重复口令: <input type="password" id="password-2">
    </p>
    <p>
        <button type="submit">提交</button> <button type="reset">重置</button>
    </p>
</form>

let username = document.querySelector("#username");
let pwd1 = document.querySelector("#password");
let pwd2 = document.querySelector("#password-2");
let nameReg = /\w{3,10}/;
let nameIsLegal = nameReg.test(username.value);
let pwdReg = new RegExp(".{6,20}");
let pwdIsLegal = pwdReg.test(pwd1.value);
return nameIsLegal && pwdIsLegal && pwd1.value === pwd2.value;
```

使用js实现动画效果

```
<style>
```

```
#container {
  width: 400px;
  height: 400px;
  position: relative;
  background: yellow;
}
#animate {
  width: 50px;
  height: 50px;
  position: absolute;
  background-color: red;
}
</style>

<p><button onclick="myMove()">单击我</button></p>

<div id ="container">
  <div id ="animate"></div>
</div>

<script>
function myMove() {
  var elem = document.getElementById("animate");
  var pos = 0;
  var id = setInterval(frame, 5); // 每5ms执行一次
  function frame() {
    if (pos == 350) { // 边界
      clearInterval(id);
    } else {
      pos+=1;  // 每次移动的步幅
      elem.style.top = pos + "px";  // 修改位置
      elem.style.left = pos + "px";
    }
  }
}
</script>
```

猜数字游戏

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <style>
        .continer{
            display: flex;
```

```
            position: relative;
            justify-content: center;
        }
        .guess-game {
            margin-top: 40px;
        }
        .gussed-num {
            margin-top: 15px;
        }
        .btn-restart {
            margin-top: 20px;
        }
        .wrong{
            margin-top: 5px;
            background-color: red;
        }
        .right{
            margin-top: 5px;
            background-color: rgb(0, 255, 81);
        }
    </style>
</head>
<body>
    <div class="continer">
        <div class="guess-game">

            <label for="input">请输入：</label>
            <input class="input" id="input" type="text" />
            <button class="guess">我猜！</button>

            <div class="gussed-num">
                <div>
                    <span>你已经猜过</span>
                    <span class="guessNum">0</span>
                    <span>次：</span>
                    <span class="guessVal"></span>
                </div>
                <div>
                    <span>还剩：</span>
                    <span class="restNum">5</span>
                    <span>次。</span>
                </div>
            </div>

            <div class="wrong">
                <!-- 错误提示 -->
            </div>
            <div class="right">
                <!-- 正确提示 -->
            </div>
            <div class="restart">
                <!-- 重新开始 -->
            </div>
        </div>
    </div>
```

```
<script>
    const getRandom = ()=>parseInt(Math.random()*100);
    // 生成随机数
    let randNum = getRandom();

    let container = document.querySelector('.container');
    let input = document.querySelector('.input')
    let button = document.querySelector('.guess');
    let guessNum = document.querySelector('.guessNum')
    let guessVal = document.querySelector('.guessVal')
    let wrong = document.querySelector('.wrong');
    let right = document.querySelector('.right');
    let restNum = document.querySelector('.restNum');
    let restart = document.querySelector('.restart')
    // 次数限制
    let limit = 5;
    // 已经猜测的次数
    let curGuess = 0;
    console.log(randNum);
    // 添加监听
    button.onclick = startGuess;

    // 重新开始的按钮
    const generateRestartButton = () => {
        input.disabled = true;
        button.disabled = true;
        let newButton = document.createElement('button');
        newButton.innerHTML = "TRY AGAIN";
        newButton.onclick = startNewGame;
        newButton.classList.add('btn-restart');
        restart.appendChild(newButton);
    }

    // 开始新游戏，清除以前的变量
    function startNewGame() {
        input.value = '';
        input.disabled = false;
        button.disabled = false;
        // 次数限制
        limit = 5;
        // 已经猜测的次数
        curGuess = 0;
        guessNum.innerHTML = curGuess;
        restNum.innerHTML = limit - curGuess;
        randNum = getRandom();
        //alert(randNum);
        console.log(randNum);
        wrong.innerHTML = '';
        right.innerHTML = '';
        restart.firstElementChild.remove();
    }

    // 验证输入数字的合法性
    function isvaild(num) {
```

```
        var reg = /^((?!0)\d{1,2}|100)$/;
        if (!num.match(reg)) {
        return false;
        } else {
        return true;
        }
    }

    // 开始猜数字
    function startGuess() {
        let inputVal = input.value;
        input.focus();
        // 判断空
        if (inputVal === ''){
            alert('Enter null');
            return;
        }
        // 正则表达式匹配0-100
        if (!isvaild(inputVal)){
            alert('Enter number is not a integer in 0-100');
            return;
        }

        curGuess++;
        guessNum.innerHTML = curGuess;
        restNum.innerHTML = limit - curGuess;
        if (curGuess === 5) {
            generateRestartButton();
            return;
        }

        wrong.innerHTML = "";
        if (inputVal > randNum) {
            //alert('你猜大了')
            wrong.innerHTML = "你猜大了"
        }
        else if(inputVal < randNum){
            //alert('你猜小了');
            wrong.innerHTML = "你猜小了"
        }
        else{
            //alert('你猜对了');
            right.innerHTML = "恭喜你，猜对了！"
            generateRestartButton();
        }

    }
</script>
</body>
</html>
```

标准对象

Data

在JavaScript中，`Date`对象用来表示日期和时间。

要获取系统当前时间，用：

```
var now = new Date();
now; // Wed Jun 24 2015 19:49:22 GMT+0800 (CST)
now.getFullYear(); // 2015, 年份
now.getMonth(); // 5, 月份，注意月份范围是0~11，5表示六月
now.getDate(); // 24, 表示24号
now.getDay(); // 3, 表示星期三
now.getHours(); // 19, 24小时制
now.getMinutes(); // 49, 分钟
now.getSeconds(); // 22, 秒
now.getMilliseconds(); // 875, 毫秒数
now.getTime(); // 1435146562875, 以number形式表示的时间戳
```

注意，当前时间是浏览器从本机操作系统获取的时间，所以不一定准确，因为用户可以把当前时间设定为任何值。

如果要创建一个指定日期和时间的`Date`对象，可以用：

```
var d = new Date(2015, 5, 19, 20, 15, 30, 123);
d; // Fri Jun 19 2015 20:15:30 GMT+0800 (CST)
```

你可能观察到了一个*非常非常坑爹*的地方，就是JavaScript的月份范围用整数表示是0~11，`0`表示一月，`1`表示二月……，所以要表示6月，我们传入的是`5`！这绝对是JavaScript的设计者当时脑抽了一下，但是现在要修复已经不可能了。

JavaScript的Date对象月份值从0开始，牢记0=1月，1=2月，2=3月，……，11=12月。

第二种创建一个指定日期和时间的方法是解析一个符合ISO 8601格式的字符串：

```
var d = Date.parse('2015-06-24T19:49:22.875+08:00');
d; // 1435146562875
```

但它返回的不是`Date`对象，而是一个时间戳。不过有时间戳就可以很容易地把它转换为一个`Date`：

```
var d = new Date(1435146562875);
d; // Wed Jun 24 2015 19:49:22 GMT+0800 (CST)
d.getMonth(); // 5
```

使用Date.parse()时传入的字符串使用实际月份01~12，转换为Date对象后getMonth()获取的月份值为0~11。

时区

`Date`对象表示的时间总是按浏览器所在时区显示的，不过我们既可以显示本地时间，也可以显示调整后的UTC时间：

```
var d = new Date(1435146562875);
d.toLocaleString(); // '2015/6/24 下午7:49:22'，本地时间（北京时区+8:00），显示的字符串与操作系统设定的格式有关
d.toUTCString(); // 'Wed, 24 Jun 2015 11:49:22 GMT'，UTC时间，与本地时间相差8小时
```

那么在JavaScript中如何进行时区转换呢？实际上，只要我们传递的是一个`number`类型的时间戳，我们就不用关心时区转换。任何浏览器都可以把一个时间戳正确转换为本地时间。

时间戳是个什么东西？时间戳是一个自增的整数，它表示从1970年1月1日零时整的GMT时区开始的那一刻，到现在的毫秒数。假设浏览器所在电脑的时间是准确的，那么世界上无论哪个时区的电脑，它们此刻产生的时间戳数字都是一样的，所以，时间戳可以精确地表示一个时刻，并且与时区无关。

所以，我们只需要传递时间戳，或者把时间戳从数据库里读出来，再让JavaScript自动转换为当地时间就可以了。

要获取当前时间戳，可以用：

```
if (Date.now) {
    console.log(Date.now()); // 老版本IE没有now()方法
} else {
    console.log(new Date().getTime());
}
```

正则表达式RegExp

字符串是编程时涉及到的最多的一种数据结构，对字符串进行操作的需求几乎无处不在。比如判断一个字符串是否是合法的Email地址，虽然可以编程提取`@`前后的子串，再分别判断是否是单词和域名，但这样做不但麻烦，而且代码难以复用。

正则表达式是一种用来匹配字符串的强有力的武器。它的设计思想是用一种描述性的语言来给字符串定义一个规则，凡是符合规则的字符串，我们就认为它“匹配”了，否则，该字符串就是不合法的。

所以我们判断一个字符串是否是合法的Email的方法是：

1. 创建一个匹配Email的正则表达式；
2. 用该正则表达式去匹配用户的输入来判断是否合法。

因为正则表达式也是用字符串表示的，所以，我们要首先了解如何用字符来描述字符。

在正则表达式中，如果直接给出字符，就是精确匹配。用`\d`可以匹配一个数字，`\w`可以匹配一个字母或数字，所以：

- `'00\d'`可以匹配`'007'`，但无法匹配`'00A'`；
- `'\d\d\d'`可以匹配`'010'`；
- `'\w\w'`可以匹配`'js'`；

`.`可以匹配任意字符，所以：

- `'js.'`可以匹配`'jsp'`、`'jss'`、`'js!'`等等。

要匹配变长的字符，在正则表达式中，用`*`表示任意个字符（包括0个），用`+`表示至少一个字符，用`?`表示0个或1个字符，用`{n}`表示n个字符，用`{n,m}`表示n-m个字符：

来看一个复杂的例子：`\d{3}\s+\d{3,8}`。

我们来从左到右解读一下：

1. `\d{3}`表示匹配3个数字，例如`'010'`；
2. `\s`可以匹配一个空格（也包括Tab等空白符），所以`\s+`表示至少有一个空格，例如匹配`' '`，`'\t\t'`等；
3. `\d{3,8}`表示3-8个数字，例如`'1234567'`。

综合起来，上面的正则表达式可以匹配以任意个空格隔开的带区号的电话号码。

如果要匹配`'010-12345'`这样的号码呢？由于`'-'`是特殊字符，在正则表达式中，要用`'\'`转义，所以，上面的正则是`\d{3}\-\d{3,8}`。

但是，仍然无法匹配`'010 - 12345'`，因为带有空格。所以我们需要更复杂的匹配方式。

进阶

要做更精确地匹配，可以用`[]`表示范围，比如：

- `[0-9a-zA-Z\_]`可以匹配一个数字、字母或者下划线；
- `[0-9a-zA-Z\_]+`可以匹配至少由一个数字、字母或者下划线组成的字符串，比如`'a100'`，`'0_Z'`，`'js2015'`等等；
- `[a-zA-Z\_\$][0-9a-zA-Z\_\$]*`可以匹配由字母或下划线、开头，后接任意个由一个数字、字母或者下划线、组成的字符串，也就是JavaScript允许的变量名；
- `[a-zA-Z\_\$][0-9a-zA-Z\_\$]{0, 19}`更精确地限制了变量的长度是1-20个字符（前面1个字符+后面最多19个字符）。

`A|B`可以匹配A或B，所以`(J|j)ava(S|s)cript`可以匹配`'JavaScript'`、`'Javascript'`、`'javaScript'`或者`'javascript'`。

`^`表示行的开头，`^\d`表示必须以数字开头。

`$`表示行的结束，`\d$`表示必须以数字结束。

你可能注意到了，`js`也可以匹配`'jsp'`，但是加上`^js$`就变成了整行匹配，就只能匹配`'js'`了。

RegExp

有了准备知识，我们就可以在JavaScript中使用正则表达式了。

JavaScript有两种方式创建一个正则表达式：

第一种方式是直接通过`/正则表达式/`写出来，第二种方式是通过`new RegExp('正则表达式')`创建一个RegExp对象。

两种写法是一样的：

```
var re1 = /ABC\-001/;
var re2 = new RegExp('ABC\\-001');

re1; // /ABC\-001/
re2; // /ABC\-001/
```

注意，如果使用第二种写法，因为字符串的转义问题，字符串的两个`\\`实际上是一个`\`。

先看看如何判断正则表达式是否匹配：

```
var re = /^\d{3}\-\d{3,8}$/;
re.test('010-12345'); // true
re.test('010-1234x'); // false
re.test('010 12345'); // false
```

RegExp对象的`test()`方法用于测试给定的字符串是否符合条件。

切分字符串

用正则表达式切分字符串比用固定的字符更灵活，请看正常的切分代码：

```
'a b   c'.split(' '); // ['a', 'b', '', '', 'c']
```

嗯，无法识别连续的空格，用正则表达式试试：

```
'a b   c'.split(/\s+/); // ['a', 'b', 'c']
```

无论多少个空格都可以正常分割。加入`,`试试：

```
'a,b, c  d'.split(/[\s\,]+/); // ['a', 'b', 'c', 'd']
```

再加入`;`试试：

```
'a,b;; c  d'.split(/[\s\,\;]+/); // ['a', 'b', 'c', 'd']
```

如果用户输入了一组标签，下次记得用正则表达式来把不规范的输入转化成正确的数组。

分组

除了简单地判断是否匹配之外，正则表达式还有提取子串的强大功能。用`()`表示的就是要提取的分组（Group）。比如：

`^(\d{3})-(\d{3,8})$`分别定义了两个组，可以直接从匹配的字符串中提取出区号和本地号码：

```
var re = /^(\d{3})-(\d{3,8})$/;
re.exec('010-12345'); // ['010-12345', '010', '12345']
re.exec('010 12345'); // null
```

如果正则表达式中定义了组，就可以在`RegExp`对象上用`exec()`方法提取出子串来。

`exec()`方法在匹配成功后，会返回一个`Array`，第一个元素是正则表达式匹配到的整个字符串，后面的字符串表示匹配成功的子串。

`exec()`方法在匹配失败时返回`null`。

提取子串非常有用。来看一个更凶残的例子：

```
var re = /^(0[0-9]|1[0-9]|2[0-3]|[0-9])\:(0[0-9]|1[0-9]|2[0-9]|3[0-9]|4[0-9]|5[0-
9]|[0-9])\:(0[0-9]|1[0-9]|2[0-9]|3[0-9]|4[0-9]|5[0-9]|[0-9])$/;
re.exec('19:05:30'); // ['19:05:30', '19', '05', '30']
```

这个正则表达式可以直接识别合法的时间。但是有些时候，用正则表达式也无法做到完全验证，比如识别日期：

```
var re = /^(0[1-9]|1[0-2]|[0-9])-(0[1-9]|1[0-9]|2[0-9]|3[0-1]|[0-9])$/;
```

对于`'2-30'`，`'4-31'`这样的非法日期，用正则还是识别不了，或者说写出来非常困难，这时就需要程序配合识别了。

贪婪匹配

需要特别指出的是，正则匹配默认是贪婪匹配，也就是匹配尽可能多的字符。举例如下，匹配出数字后面的`0`：

```
var re = /^(\d+)(0*)$/;
re.exec('102300'); // ['102300', '102300', '']
```

由于`\d+`采用贪婪匹配，直接把后面的`0`全部匹配了，结果`0*`只能匹配空字符串了。

必须让`\d+`采用非贪婪匹配（也就是尽可能少匹配），才能把后面的`0`匹配出来，加个`?`就可以让`\d+`采用非贪婪匹配：

```
var re = /^(\d+?)(0*)$/;
re.exec('102300'); // ['102300', '1023', '00']
```

全局搜索

JavaScript的正则表达式还有几个特殊的标志，最常用的是`g`，表示全局匹配：

```
var r1 = /test/g;
// 等价于:
var r2 = new RegExp('test', 'g');
```

全局匹配可以多次执行`exec()`方法来搜索一个匹配的字符串。当我们指定`g`标志后，每次运行`exec()`，正则表达式本身会更新`lastIndex`属性，表示上次匹配到的最后索引：

```
var s = 'JavaScript, VBScript, JScript and ECMAScript';
var re=/[a-zA-Z]+Script/g;

// 使用全局匹配:
re.exec(s); // ['JavaScript']
re.lastIndex; // 10

re.exec(s); // ['VBScript']
re.lastIndex; // 20

re.exec(s); // ['JScript']
re.lastIndex; // 29

re.exec(s); // ['ECMAScript']
re.lastIndex; // 44

re.exec(s); // null，直到结束仍没有匹配到
```

全局匹配类似搜索，因此不能使用`/^...$/`，那样只会最多匹配一次。

正则表达式还可以指定`i`标志，表示忽略大小写，`m`标志，表示执行多行匹配。

JSON

JSON是JavaScript Object Notation的缩写，它是一种数据交换格式。

在JSON出现之前，大家一直用XML来传递数据。因为XML是一种纯文本格式，所以它适合在网络上交换数据。XML本身不算复杂，但是，加上DTD、XSD、XPath、XSLT等一大堆复杂的规范以后，任何正常的软件开发人员碰到XML都会感觉头大了，最后大家发现，即使你努力钻研几个月，也未必搞得清楚XML的规范。

终于，在2002年的一天，道格拉斯·克罗克福特（Douglas Crockford）同学为了拯救深陷水深火热同时又被某几个巨型软件企业长期愚弄的软件工程师，发明了JSON这种超轻量级的数据交换格式。

道格拉斯同学长期担任雅虎的高级架构师，自然钟情于JavaScript。他设计的JSON实际上是JavaScript的一个子集。在JSON中，一共就这么几种数据类型：

- number：和JavaScript的`number`完全一致；
- boolean：就是JavaScript的`true`或`false`；
- string：就是JavaScript的`string`；
- null：就是JavaScript的`null`；
- array：就是JavaScript的`Array`表示方式——`[]`；
- object：就是JavaScript的`{ ... }`表示方式。

以及上面的任意组合。

并且，JSON还定死了字符集必须是UTF-8，表示多语言就没有问题了。为了统一解析，JSON的字符串规定必须用双引号`""`，Object的键也必须用双引号`""`。

由于JSON非常简单，很快就风靡Web世界，并且成为ECMA标准。几乎所有编程语言都有解析JSON的库，而在JavaScript中，我们可以直接使用JSON，因为JavaScript内置了JSON的解析。

把任何JavaScript对象变成JSON，就是把这个对象序列化成一个JSON格式的字符串，这样才能够通过网络传递给其他计算机。

如果我们收到一个JSON格式的字符串，只需要把它反序列化成一个JavaScript对象，就可以在JavaScript中直接使用这个对象了。

序列化

让我们先把小明这个对象序列化成JSON格式的字符串：

```
'use strict';

var xiaoming = {
    name: '小明',
    age: 14,
    gender: true,
    height: 1.65,
    grade: null,
    'middle-school': '\"W3C\" Middle School',
```

```
    skills: ['JavaScript', 'Java', 'Python', 'Lisp']
};

var s = JSON.stringify(xiaoming, null, '  ');
console.log(s);
```

要输出得好看一些，可以加上参数，按缩进输出：

```
JSON.stringify(xiaoming, null, '  ');
```

结果：

```
{
  "name": "小明",
  "age": 14,
  "gender": true,
  "height": 1.65,
  "grade": null,
  "middle-school": "\"W3C\" Middle School",
  "skills": [
    "JavaScript",
    "Java",
    "Python",
    "Lisp"
  ]
}
```

第二个参数用于控制如何筛选对象的键值，如果我们只想输出指定的属性，可以传入`Array`：

```
JSON.stringify(xiaoming, ['name', 'skills'], '  ');
```

结果：

```
{
  "name": "小明",
  "skills": [
    "JavaScript",
    "Java",
    "Python",
    "Lisp"
  ]
}
```

还可以传入一个函数，这样对象的每个键值对都会被函数先处理：

```
function convert(key, value) {
    if (typeof value === 'string') {
        return value.toUpperCase();
    }
    return value;
}

JSON.stringify(xiaoming, convert, '  ');
```

上面的代码把所有属性值都变成大写：

```
{
  "name": "小明",
  "age": 14,
  "gender": true,
  "height": 1.65,
  "grade": null,
  "middle-school": "\"W3C\" MIDDLE SCHOOL",
  "skills": [
    "JAVASCRIPT",
    "JAVA",
    "PYTHON",
    "LISP"
  ]
}
```

如果我们还想要精确控制如何序列化小明，可以给`xiaoming`定义一个`toJSON()`的方法，直接返回JSON应该序列化的数据：

```
var xiaoming = {
    name: '小明',
    age: 14,
    gender: true,
    height: 1.65,
    grade: null,
    'middle-school': '\"W3C\" Middle School',
    skills: ['JavaScript', 'Java', 'Python', 'Lisp'],
    toJSON: function () {
        return { // 只输出name和age，并且改变了key：
            'Name': this.name,
            'Age': this.age
        };
    }
};

JSON.stringify(xiaoming); // '{"Name":"小明","Age":14}'
```

反序列化

拿到一个JSON格式的字符串，我们直接用`JSON.parse()`把它变成一个JavaScript对象：

```
JSON.parse('[1,2,3,true]'); // [1, 2, 3, true]
JSON.parse('{"name":"小明","age":14}'); // Object {name: '小明', age: 14}
JSON.parse('true'); // true
JSON.parse('123.45'); // 123.45
```

`JSON.parse()`还可以接收一个函数，用来转换解析出的属性：

```
'use strict';
var obj = JSON.parse('{"name":"小明","age":14}', function (key, value) {
    if (key === 'name') {
        return value + '同学';
    }
    return value;
});
console.log(JSON.stringify(obj)); // {name: '小明同学', age: 14}
```

高阶内容

generator

generator（生成器）是ES6标准引入的新的数据类型。**一个generator看上去像一个函数，但可以返回多次。**

ES6定义generator标准的哥们借鉴了Python的generator的概念和语法，如果你对Python的generator很熟悉，那么ES6的generator就是小菜一碟了。如果你对Python还不熟，赶快恶补Python教程！。

我们先复习函数的概念。一个函数是一段完整的代码，调用一个函数就是传入参数，然后返回结果：

```
function foo(x) {
    return x + x;
}

var r = foo(1); // 调用foo函数
```

函数在执行过程中，如果没有遇到`return`语句（函数末尾如果没有`return`，就是隐含的`return undefined;`），控制权无法交回被调用的代码。

generator跟函数很像，定义如下：

```
function* foo(x) {
    yield x + 1;
    yield x + 2;
    return x + 3;
}
```

generator和函数不同的是，generator由`function*`定义（注意多出的`*`号），并且，除了`return`语句，还可以用`yield`返回多次。

大多数同学立刻就晕了，generator就是能够返回多次的“函数”？返回多次有啥用？

还是举个栗子吧。

我们以一个著名的斐波那契数列为例，它由`0`，`1`开头：

```
0 1 1 2 3 5 8 13 21 34 ...
```

要编写一个产生斐波那契数列的函数，可以这么写：

```
function fib(max) {
    var
        t,
        a = 0,
        b = 1,
        arr = [0, 1];
    while (arr.length < max) {
        [a, b] = [b, a + b];
        arr.push(b);
    }
    return arr;
}

// 测试:
fib(5); // [0, 1, 1, 2, 3]
fib(10); // [0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

函数只能返回一次，所以必须返回一个`Array`。但是，如果换成generator，就可以一次返回一个数，不断返回多次。用generator改写如下：

```
function* fib(max) {
    var
        t,
        a = 0,
        b = 1,
        n = 0;
    while (n < max) {
        yield a;
        [a, b] = [b, a + b];
        n ++;
    }
    return;
}
```

直接调用试试：

```
fib(5); // fib {[[GeneratorStatus]]: "suspended", [[GeneratorReceiver]]: Window}
```

直接调用一个generator和调用函数不一样，`fib(5)`仅仅是创建了一个generator对象，还没有去执行它。

调用generator对象有两个方法，一是不断地调用generator对象的`next()`方法：

```
var f = fib(5);
f.next(); // {value: 0, done: false}
f.next(); // {value: 1, done: false}
f.next(); // {value: 1, done: false}
f.next(); // {value: 2, done: false}
f.next(); // {value: 3, done: false}
f.next(); // {value: undefined, done: true}
```

`next()`方法会执行generator的代码，然后，每次遇到`yield x;`就返回一个对象`{value: x, done: true/false}`，然后“暂停”。返回的`value`就是`yield`的返回值，`done`表示这个generator是否已经执行结束了。如果`done`为`true`，则`value`就是`return`的返回值。

当执行到`done`为`true`时，这个generator对象就已经全部执行完毕，不要再继续调用`next()`了。

第二个方法是直接用`for ... of`循环迭代generator对象，这种方式不需要我们自己判断`done`：

```
'use strict'

function* fib(max) {
    var
        t,
        a = 0,
        b = 1,
        n = 0;
    while (n < max) {
        yield a;
        [a, b] = [b, a + b];
        n ++;
    }
    return;
}
```

浏览器对象

window

- 浏览器窗口
- `window`对象有`innerWidth`和`innerHeight`属性，可以获取浏览器窗口的内部宽度和高度。内部宽高是指除去菜单栏、工具栏、边框等占位元素后，用于显示网页的净宽高。
- 对应的，还有一个`outerWidth`和`outerHeight`属性，可以获取浏览器窗口的整个宽高。

navigator

`navigator`对象表示浏览器的信息，最常用的属性包括：

- navigator.appName：浏览器名称；
- navigator.appVersion：浏览器版本；
- navigator.language：浏览器设置的语言；
- navigator.platform：操作系统类型；
- navigator.userAgent：浏览器设定的`User-Agent`字符串。

请注意，`navigator`的信息可以很容易地被用户修改，所以JavaScript读取的值不一定是正确的。很多初学者为了针对不同浏览器编写不同的代码，喜欢用`if`判断浏览器版本，例如：

```
var width;
if (getIEVersion(navigator.userAgent) < 9) {
    width = document.body.clientWidth;
} else {
    width = window.innerWidth;
}
```

但这样既可能判断不准确，也很难维护代码。正确的方法是充分利用JavaScript对不存在属性返回`undefined`的特性，直接用短路运算符`||`计算：

```
var width = window.innerWidth || document.body.clientWidth;
```

screen

`screen`对象表示屏幕的信息，常用的属性有：

- screen.width：屏幕宽度，以像素为单位；
- screen.height：屏幕高度，以像素为单位；
- screen.colorDepth：返回颜色位数，如8、16、24。

location

`location`对象表示当前页面的URL信息。例如，一个完整的URL：

```
http://www.example.com:8080/path/index.html?a=1&b=2#TOP
```

可以用`location.href`获取。要获得URL各个部分的值，可以这么写：

```
location.protocol; // 'http'
location.host; // 'www.example.com'
location.port; // '8080'
location.pathname; // '/path/index.html'
location.search; // '?a=1&b=2'
location.hash; // 'TOP'
```

要加载一个新页面，可以调用`location.assign()`。如果要重新加载当前页面，调用`location.reload()`方法非常方便。

document

`document`对象表示当前页面。由于HTML在浏览器中以DOM形式表示为树形结构，`document`对象就是整个DOM树的根节点。

`document`的`title`属性是从HTML文档中的`<title>xxx</title>`读取的，但是可以动态改变：

要查找DOM树的某个节点，需要从`document`对象开始查找。最常用的查找是根据ID和Tag Name。

我们先准备HTML数据：

```
<dl id="drink-menu" style="border:solid 1px #ccc;padding:6px;">
    <dt>摩卡</dt>
    <dd>热摩卡咖啡</dd>
    <dt>酸奶</dt>
    <dd>北京老酸奶</dd>
    <dt>果汁</dt>
    <dd>鲜榨苹果汁</dd>
</dl>
```

用`document`对象提供的`getElementById()`和`getElementsByTagName()`可以按ID获得一个DOM节点和按Tag名称获得一组DOM节点：

`document`对象还有一个`cookie`属性，可以获取当前页面的Cookie。

Cookie是由服务器发送的key-value标示符。因为HTTP协议是无状态的，但是服务器要区分到底是哪个用户发过来的请求，就可以用Cookie来区分。当一个用户成功登录后，服务器发送一个Cookie给浏览器，例如`user=ABC123XYZ(加密的字符串)...`，此后，浏览器访问该网站时，会在请求头附上这个Cookie，服务器根据Cookie即可区分出用户。

Cookie还可以存储网站的一些设置，例如，页面显示的语言等等。

JavaScript可以通过`document.cookie`读取到当前页面的Cookie：

```
document.cookie; // 'v=123; remember=true; prefer=zh'
```

由于JavaScript能读取到页面的Cookie，而用户的登录信息通常也存在Cookie中，这就造成了巨大的安全隐患，这是因为在HTML页面中引入第三方的JavaScript代码是允许的：

```
<!-- 当前页面在wwwexample.com -->
<html>
    <head>
        <script src="http://www.foo.com/jquery.js"></script>
    </head>
    ...
</html>
```

如果引入的第三方的JavaScript中存在恶意代码，则`www.foo.com`网站将直接获取到`www.example.com`网站的用户登录信息。

为了解决这个问题，服务器在设置Cookie时可以使用`httpOnly`，设定了`httpOnly`的Cookie将不能被JavaScript读取。这个行为由浏览器实现，主流浏览器均支持`httpOnly`选项，IE从IE6 SP1开始支持。

为了确保安全，服务器端在设置Cookie时，应该始终坚持使用`httpOnly`。

history

`history`对象保存了浏览器的历史记录，JavaScript可以调用`history`对象的`back()`或`forward()`，相当于用户点击了浏览器的“后退”或“前进”按钮。

这个对象属于历史遗留对象，对于现代Web页面来说，由于大量使用AJAX和页面交互，简单粗暴地调用`history.back()`可能会让用户感到非常愤怒。

新手开始设计Web页面时喜欢在登录页登录成功时调用`history.back()`，试图回到登录前的页面。这是一种错误的方法。

任何情况，你都不应该使用`history`这个对象了。

面向对象

Ajax

如果仔细观察一个Form的提交，你就会发现，一旦用户点击“Submit”按钮，表单开始提交，浏览器就会刷新页面，然后在新页面里告诉你操作是成功了还是失败了。如果不幸由于网络太慢或者其他原因，就会得到一个404页面。

这就是Web的运作原理：一次HTTP请求对应一个页面。

如果要让用户留在当前页面中，同时发出新的HTTP请求，就必须用JavaScript发送这个新请求，接收到数据后，再用JavaScript更新页面，这样一来，用户就感觉自己仍然停留在当前页面，但是数据却可以不断地更新。

最早大规模使用AJAX的就是Gmail，Gmail的页面在首次加载后，剩下的所有数据都依赖于AJAX来更新。

用JavaScript写一个完整的AJAX代码并不复杂，但是需要注意：AJAX请求是异步执行的，也就是说，要通过回调函数获得响应。

在现代浏览器上写AJAX主要依靠`XMLHttpRequest`对象：

```
function success(text) {
    var textarea = document.getElementById('test-response-text');
    textarea.value = text;
}

function fail(code) {
    var textarea = document.getElementById('test-response-text');
    textarea.value = 'Error code: ' + code;
}

var request = new XMLHttpRequest(); // 新建XMLHttpRequest对象

request.onreadystatechange = function () { // 状态发生变化时，函数被回调
    if (request.readyState === 4) { // 成功完成
        // 判断响应结果:
        if (request.status === 200) {
            // 成功，通过responseText拿到响应的文本:
            return success(request.responseText);
        } else {
            // 失败，根据响应码判断失败原因:
            return fail(request.status);
        }
    } else {
        // HTTP请求还在继续...
    }
```

```
}

// 发送请求:
request.open('GET', '/api/categories');
request.send();

alert('请求已发送，请等待响应...');
```

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <title>简单的天气查询</title>
</head>
<body>
        <p>输入当前城市:<input type="text" id="city"></p>
        <button type="submit" onclick="getWeather()">查询</button>
        <p id="weatherInfo"></p>
    <script>
        function getWeather(){
            let url = 'https://www.apiopen.top/weatherApi?city=';
            let city = document.getElementById('city');
            // 获取要查询的城市
            let newURL = url + city.value;
            console.log(newURL);
            var xhr = new XMLHttpRequest();
            xhr.onreadystatechange = function(){
                if(xhr.readyState === 4){
                    if(xhr.status === 200){
                        return success(xhr.responseText);
                    }else{
                        alert('失败！')
                    }
                }
            }
            xhr.open('GET',newURL);
            xhr.send();
        }

        function success(data){
            let weather = document.getElementById('weatherInfo');
            let weatherInfo = JSON.parse(data);
            if(weatherInfo.code ===  200){
                weather.innerHTML = '查询成功' + '<br>'+
                                    '当前城市：'+weatherInfo.data.city+ '<br>'+
                                    '当前温度：'+weatherInfo.data.wendu+'<br>'+
                                    '气
温：'+'最'+weatherInfo.data.forecast[0].high +',
最'+weatherInfo.data.forecast[0].low+'<br>'+
                                    '天
气：'+weatherInfo.data.forecast[0].type+'<br>'+
```

```
                                        '风
向：'+weatherInfo.data.forecast[0].fengxiang +
weatherInfo.data.forecast[0].fengli+'<br>' +
                                        '注意：'+weatherInfo.data.ganmao
            }else{
                weather.innerText  = weatherInfo.msg;
            }
        }
    </script>
</body>
</html>
```

异步编程

timer，回调函数

promise

await、async

事件循环

原型链