〇.基本语法

```
var x = 1; var y = 2;
if(x < y){
    x = y;
}
```

一.数据类型

1.Number

> ***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。
>
> 其余整数、浮点数、科学计数法不做介绍

2.字符串

```
var a = 'bc';
var b = "bc";
```

3.布尔型

4.运算符

> JS可对任意数据类型作比较

```
false == 0;//True. ==比较会自动转换数据类型再比较
false === 0;//False. ===不会转换数据类型，如果数据类型不一致返回False,如果一致再比较
NaN === NaN;//False. NaN与其他所有值都不相等
isNaN(NaN);//True. 唯一能判断NaN的函数
Math.abs(1 / 3 - (1 - 2 / 3)) < 0.0000001; // true. 比较浮点数
```

5.数组

> 数组:JavaScript的数组可以包括任意数据类型

```
var arr = [1,2,3.14,'Hello', null, true];//建议使用
new Array(1,2,3);
arr[3];// 'Hello'
```

6.对象

```
var person = {
    name: 'Bob',
    age: 20,
    tags: ['js', 'web', 'mobile'],
    city: 'Beijing',
    hasCar: true,
   zipcode: null
};
```

7.变量

> 变量：变量名是大小写英文、数字、$和_的组合，且不能用数字开头.

8.strict 模式

> - 如果一个变量没有通过`var`申明就被使用，那么该变量就自动被申明为全局变量.
> - 在strict模式下运行的JavaScript代码，强制通过`var`申明变量，未使用`var`申明变量就使用的，将导致运行错误。

```
i = 10; // i现在是全局变量
'use strict' // 这是一个字符串，不支持strict模式的浏览器会把它当做一个字符串语句执行，支持strict模式的浏览器将开启strict模式运行JavaScript。
```

二.字符串

1. 转义字符 `\`

```
'\x41';//等同于 'A'
//类似可以用\u####表示Unicode字符
'\u4e2d\u6587';//等同于'中文'
```

2.多行字符串

```
`
多
行
字符串
`
```

3.字符串拼接 `+` 或 `$`

```
var name = 'Tom';
var age = 20;
var message1 = '你好，' + name + ',今年' + age + '岁。';
var message2 = `你好，  ${name},今年${age}岁。`;//注意是使用的``符号
```

4.操作字符串

```
var s = 'Hello world!';
s.length;// 13
s[0];//H
s[13];//undefined 超出范围
s[0] = 'h';//不会改变字符串
//常用方法:
// toUpperCase()字符串变大写
var upper = s.toUpperCase();// HELLO WORLD!
// toLowerCase()把一个字符串全部变为小写
// indexOf()会搜索指定字符串出现的位置
s.indexOf('world');//7
s.indexOf('World');//-1
// substring()返回指定索引区间的子串
var str = 'hello, world';
str.substring(0,5);//返回区间[0,5）字符, 'hello'
str.substring(7);//从索引7开始到结束，返回'world'
```

三. 数组

> Array也是一个对象，它的每个元素索引也被视为一个属性（可用`for in`遍历）

1.`length`属性:可直接给该属性赋值导致Array大小变化

```
var arr0 = [1, 2, 3.14, 'Hello', null, true];
arr0.length;//6
arr0.length = 3;
arr0;//变为 [1,2,3.14]
arr0.length = 5;
arr0;//变为[1,2, 3.14, undefined, undefined]

//如果通过索引赋值时，索引超过了范围，同样会引起Array大小的变化：
arr0[10] = 'x';
arr0.length;//11
```

2.`indexOf()`：搜索一个指定元素的位置，无则返回-1

3.`slice()`:数组切片

```
var arr1 = ['A', 'B', 'C', 'D', 'E', 'F', 'G'];
arr1.slice(0, 3); // 从索引0开始，到索引3结束，但不包括索引3: ['A', 'B', 'C']
arr1.slice(3); // 从索引3开始到结束: ['D', 'E', 'F', 'G']
var arrCopy = arr.slice();//常用来复制数组
arrCopy;//['A', 'B', 'C', 'D', 'E', 'F', 'G']
```

4.`pop()` `push()`:

```
var arr2 = [1,2];
arr.push("A", "B");
arr;//[1,2,"A","B"]
arr.pop();//返回'B'
// 空数组继续pop不会报错，而是返回undefined
```

5.`unshift()` `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 arr3 = [1,2];
arr3.unshift('A', 'B');
arr3;//['A', 'B', 1, 2]
arr3.shift();//'A'
//空数组继续shift不会报错，而是返回undefined
```

6.`sort()`排序

7.`reverse()`:数组翻转

8.`splice()`:该方法是修改Array的“万能方法”，它可以从指定的索引开始删除若干元素，然后再从该位置添加若干元素.

```
var arr4 = ['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:
arr4.splice(2, 3, 'Google', 'Facebook'); // 返回删除的元素 ['Yahoo', 'AOL',
'Excite']
arr4; // ['Microsoft', 'Apple', 'Google', 'Facebook', 'Oracle']

// 只删除,不添加:
arr4.splice(2, 2); // ['Google', 'Facebook']
arr4; // ['Microsoft', 'Apple', 'Oracle']

// 只添加,不删除:
arr4.splice(2, 0, 'Google', 'Facebook'); // 返回[],因为没有删除任何元素
arr4; // ['Microsoft', 'Apple', 'Google', 'Facebook', 'Oracle']
```

9.`concat()`:该方法连接另一个数组起来，返回连接后的数组.

```
var arr5 = [1];
var added = arr5.concat([2,3,4]);
added;//[1,2,3,4]
arr5;//[1]
```

10.`join()`: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 arr6 = ['A', 'B', 'C'];
arr6.join('-');//'A-B-C'
//如果Array的元素不是字符串，将自动转换为字符串后再连接。
```

11.多维数组.

```
var arr7 = [[1, 2, 3], [400, 500, 600], '-'];
arr7[1][1];//500
```

四.对象

1.示例

```
var xiaoming={
    name:'小明',//访问 xiaoming.name或xiaoming['name']
    birth:1997,
    'middle-school':'No.1 Middle School'//访问xiaoming['middle-school']
}
xiaoming.age;//undefined

//为xiaoming对象增加一个age属性
xiaoming.age = 18;

//删除birth属性
delete xiaoming.birth;

//删除'middle-school'属性
delete xiaoming['middle-school'];

//in判断小明是否拥有某属性。
'name' in xiaoming;//true
'middle-school' in xiaoming;// false
//不过要小心，如果in判断一个属性存在，这个属性不一定是xiaoming的，它可能是xiaoming继承得到的
//要判断一个属性是否是xiaoming自身拥有的，而不是继承得到的，可以用hasOwnProperty()方法
```

2.遍历对象属性示例

```
for(var i in xiaoming){
    console.log(`${i}:${xiaoming[i]}`);
}
```

五. 条件判断

> 条件判断:JavaScript把`null`、`undefined`、`0`、`NaN`和空字符串''视为***false***，其他值一概视为***true***.

六. 循环

1.`for in`循环

> 一般遍历对象的每一个属性.

```
var o= {
    name:'xiaoming',
    age: 20,
    city:'ShanDong'
};
for(var key in o){
    console.log(key);//'name', 'age', 'city'
}

var a = ['A', 'B', 'C'];
for (var i in a) {
```

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

//因为Array也被视作对象
a.name = 'Array';
for(var i in a){
    console.log(i);//'0', '1', '2', 'name'
}
```

2. for of循环

> 循环迭代器的每一个元素(貌似无法获取索引)

```
var arr = ['A', 'B', 'C'];
var s = new Set(['A', 'B', 'C']);
var m = new Map([[1, 'x'], [2, 'y'], [3, 'z']]);
for (var x of arr) { // 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]);
}
```

七.Map 和 Set

1. Map()

```
//二维数组初始化Map
var m1 = new Map([['Michael', 34], ['Bob', 75]]);

//使用set方法初始化Map
var m2 = new Map();
m2.set('Michael', 34);
m2.set('Bob', 75);
m2.set(12, 'Xiaoming')
m2.has('Adam');//false
m2.delete('Bob');//删除键'Bob'
m2.get('Bob');//undefined
```

2.Set()

```
var s1 = new Set();//空Set
var s2 = new Set([1,2,'3']);

//添加元素
s1.add(4);

//删除元素
s1.delete(4);
```

八.迭代

1.使用forEach()迭代数组

```
var a = ['A', 'B', 'C'];
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);
});
```

2.使用forEach()迭代Set

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

3.使用forEach()迭代Map

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

九.函数

1.定义函数

```
function foo0(x){
    //Do something
}

//定义一个匿名函数（函数也是一个对象）
var foo1 = function(x){
    //Do something
}
```

2.调用函数

```
foo(1);
foo(1,2,'abc');//可传入任意参
```

3.arguments

> 即使函数不定义任何参数，还是可以通过`arguments`拿到参数的值.

```
function abs(){
    if(arguments.length === 0){
        return 0;
    }
    var x = arguments[0];
    return x >= 0 ? x: -x;
}
abs();// 0
abs(10);//10
abs(12, -1);//12
```

4.rest 获取多余参数

```
function foo1(x, y, ...rest){
    console.log(x);
    console.log(y);
    for(var i of rest){
        console.log(i);
    }
}
foo1();//undefined undefined undedined
foo1(1,2);//1 2 undefined
foo1(1,2,3,4);//1 2 3 4
```

十. 变量作用域和命名空间

1.变量作用空间

> 未被定义在任何函数里的为全局变量，与`window`对象绑定

```
var x = 12;
x;//12
window.x;//12
```

2.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

```
// 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';
}
```

3.声明常量

```
const PI = 3.14
```

4.变量提升

> JS的函数定义有个特点，它会先扫描整个函数体的语句，把所有申明的变量“提升”到函数顶部.

```
function foo0(){
    var sum = 0;
    for(var i = 0; i <= 10; i++){
        sum += i;
    }
    i += 1;//可访问
}

function foo1(){
    var sum = 0;
    for(let i = 0; i <= 10; i++){
        sum += i;
    }
    i += 1;//SyntaxError:
}
```

5. 解构赋值

```
var [x, y, z] = ['hello', 'JavaScript', 'ES6'];//x,y,z分别被赋值数组相应元素

//嵌套结构赋值
let [x, [y, z]] = ['hello', ['JavaScript', 'ES6']];

//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

//从一个对象中抽取若干属性
var person = {
    name: '小明',
    age: 20,
    gender: 'male',
```

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

//使用的变量名与属性名不一致
var {name,age, passport:id} = person;//让变量id获得person.passport的值

//解构赋值使用默认值
var {name, age, passport, single=true} = person;

//解决变量已经被声明时再使用解构赋值一般写法会报错
var x, y;
({x, y} = { name: '小明', x: 100, y: 200});
```

十一. 方法

```
function getAge() {
    var y = new Date().getFullYear();
    return y - this.birth;
}

var xiaoming = {
    name: '小明',
    birth: 1990,
    age: getAge
};
```

1.方法内的`this`指针指代的对象视情况而定：在对象内指代对象，在对象外指代`window`。

```
var foo = xiangming.age;//拿到对象方法
foo();//NaN错误，因为对象外this指向window(strict 模式中指向undefined)
```

2.函数对象的`apply()`方法可以控制`this`指向：

```
//它接收两个参数，第一个参数就是需要绑定的this变量，第二个参数是Array，表示函数本身的参数
xiaoming.age(); // 25
getAge.apply(xiaoming, []); // 25, this指向xiaoming, 参数为空
```

3.类似的，函数对象`call()`方法的使用

```
Math.max.apply(null, [3, 5, 4]); // 5
Math.max.call(null, 3, 5, 4); // 5
```

4.装饰器例子：统计`parseInt()`方法使用次数

```
var count = 0;
var oldParseInt = parseInt; // 保存原函数
window.parseInt = function () {
 count += 1;
 return oldParseInt.apply(null, arguments); // 调用原函数
};
```

十二.高阶函数

> 一个函数接收另一个函数作为参数

1. `map(function(currentValue, index, arr))` 方法

> ***currentValue***：当前元素
>
> ***index***: 当前元素下标
>
> ***arr***: 迭代的数组

`map()` 方法应用:

> 计算f(x) = x*x

```
    var f = function (x) {
        return x * x;
    };

    var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9];
    var result0 = [];
    for (var i=0; i<arr.length; i++) {
        result0.push(f(arr[i]));
    }
    var result1 = arr.map(f); // [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

更多参数详见相关文档

2. `reduce()` 方法

> [x1, x2, x3, x4].reduce(f) = f(f(f(x1, x2), x3), x4)
>
> reduce()方法必须接收两个参数

`reduce()` 方法应用:

> 计算数组和

```
    var arr = [1, 2, 3, 4, 5];
    arr.reduce(function(x, y){
        return x + y;
    })
```

3.`filter(function(element, index, self))`方法

> Array的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。

`filter()`方法应用：

> 1.把一个字符串13579先变成Array——[1, 3, 5, 7, 9]，再利用reduce()就可以写出一个把字符串转换为Number的函数。

```
function myIntegerParser(s){
        var arr = s.split('').map(function(x){return x - '0'});
        return arr.reduce(function(x, y){return x * 10 + y});
    }
```

> 2.请把用户输入的不规范的英文名字，变为首字母大写，其他小写的规范名字。输入：['adam', 'LISA', 'barT']，输出：['Adam', 'Lisa', 'Bart']。

```
//解1
function fooO(s){
    return s.map(function(currentValue){
        var temp = currentValue.toLowerCase().split('');
        temp[0] = temp[0].toUpperCase();
        return temp.join('');
    })
 }
```

```
//解2
function foo1(s){
    return s.map(function(currentValue){
        return currentValue[0].toUpperCase() +
currentValue.substring(1).toLowerCase();
    })
 }
```

> 3.使用filter()筛选出数组里的素数：

```
function getPrimes(arr){
        return arr.filter(function(currentValue){
            if(currentValue <= 1){
                return false;
            }
            else if(currentValue == 2){
                return true;
            }
            else{
                for(let i = 2; i <= Math.sqrt(currentValue); i++){
                    if(currentValue % i === 0){
                        return false;
                    }
                }
                return true;
            }
```

```
        });
    }
```

3.Sort()

> - `Array`的`sort()`方法默认把所有元素先转换为String再排序(所以有时会出现一些神奇的结果)。
> - 通常规定，对于两个元素`x`和`y`，如果认为`x < y`，则返回`-1`，如果认为`x == y`，则返回`0`，如果认为`x > y`，则返回`1`。

`sort`方法排序示例：

```
var arr = [10, 20, 1, 2];

var arr1 = arr.sort(function (x, y){
    if(x > y){
        return 1;
    }
    if(y > x){
        return -1;
    }
    return 0;
});

console.log(arr);//[1,2,10,20]
console.log(arr1);//[1,2,10,20],  和arr是同一对象
arr1 === arr;//true
```

4. every()

> `every()`方法可以判断数组的所有元素是否满足测试条件。

```
var arr = ['Apple', 'pear', 'orange'];
console.log(arr.every(function(s){
    return s.length > 0;
}));//true

console.log(arr.every(function(s){
    return s.toLowerCase() === s;
}));//false
```

5.find()

> `find()`方法用于查找符合条件的第一个元素，如果找到了，返回这个元素，否则，返回`undefined`

```
var arr = ['Apple', 'pear', 'orange'];
console.log(arr.find(function(s){
    return s.toLowerCase() === s;
})); //'pear'

console.log(arr.find(function(s){
    return s.toUpperCase() === s;
}));// undefined, 因为没有全部是大小写的元素
```

6.findIndex()

> `findIndex()`和`find()`类似，也是查找符合条件的第一个元素，不同之处在于`findIndex()`会返回这个元素的索引，如果没有找到，返回`-1`

```
var arr = ['Apple', 'pear', 'orange'];
console.log(arr.findIndex(function (s) {
    return s.toLowerCase() === s;
})); // 1, 因为'pear'的索引是1

console.log(arr.findIndex(function (s) {
    return s.toUpperCase() === s;
})); // -1
```

7.forEach()

> 见**八.迭代**。

8.闭包

> 高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

例如定义一个求Array和的函数

```
function sum(arr) {
    return arr.reduce(function (x, y) {
        return x + y;
    });
}

sum([1, 2, 3, 4, 5]); // 15
```

但是，如果不需要立刻求和，而是在后面的代码中，根据需要再计算怎么办？可以不返回求和的结果，而是返回求和的函数！

```
function lazy_sum(arr) {
    var sum = function () {
        return arr.reduce(function (x, y) {
            return x + y;
        });
    }
    return sum;
}

var f = lazy_sum([1,2,4,5]);
f();//15
```

> 在这个例子中，我们在函数`lazy_sum`中又定义了函数`sum`，并且，内部函数`sum`可以引用外部函数`lazy_sum`的参数和局部变量，当`lazy_sum`返回函数`sum`时，相关参数和变量都保存在返回的函数中，这种称为“闭包（Closure）”的程序结构拥有极大的威力。

请再注意一点，当我们调用`lazy_sum()`时，每次调用都会返回一个新的函数，即使传入相同的参数：

```
var f1 = lazy_sum([1, 2, 3, 4, 5]);
var f2 = lazy_sum([1, 2, 3, 4, 5]);
f1 === f2; // false
```

另一个需要注意的问题是，返回的函数并没有立刻执行，而是直到调用了`f()`才执行。我们来看一个例子：

```
function count() {
    var arr = [];
    for (var i=1; i<=3; i++) {
        arr.push(function () {
            return i * i;
        });
    }
    return arr;
}

var results = count();
var f1 = results[0];
var f2 = results[1];
var f3 = results[2];
f1();//16
f2();//16
f3();//16
//这是因为返回的函数引用了变量i，f1~f3这三个函数并没立刻执行，而是等for循环完毕，三个函数都返回后才执行
//此时i的值为4，所以f1~f3的值都为16
```

使用“ 创建一个匿名函数并立刻执行 ”语法解决：

```
function count() {
    var arr = [];
    for (var i=1; i<=3; i++) {
        arr.push((function (n) {
            return function () {
                return n * n;
            }
```

```
        })(i));
    }
    return arr;
}

var results = count();
var f1 = results[0];
var f2 = results[1];
var f3 = results[2];

f1(); // 1
f2(); // 4
f3(); // 9
```

```
//创建匿名函数立刻执行
(function (x) {
 return x * x;
})(3); // 9
//以参数为3立刻执行函数
```

说了这么多，难道闭包就是为了返回一个函数然后延迟执行吗？

当然不是！闭包有非常强大的功能。举个栗子：

在面向对象的程序设计语言里，比如Java和C++，要在对象内部封装一个私有变量，可以用`private`修饰一个成员变量。

在没有`class`机制，只有函数的语言里，借助闭包，同样可以封装一个私有变量。我们用JavaScript创建一个计数器：

```
'use strict';

function create_counter(initial) {
var x = initial || 0;
return {
   inc: function () {
       x += 1;
       return x;
    }
}
}
```

它用起来像这样：

```
var c1 = create_counter();
c1.inc(); // 1
c1.inc(); // 2
c1.inc(); // 3

var c2 = create_counter(10);
c2.inc(); // 11
c2.inc(); // 12
c2.inc(); // 13
```

在返回的对象中，实现了一个闭包，该闭包携带了局部变量`x`，并且，从外部代码根本无法访问到变量`x`。换句话说，闭包就是携带状态的函数，并且它的状态可以完全对外隐藏起来。

9.箭头函数

> 1）箭头函数内只有一条语句写法：
>
> ```
> x => x * x;
> //多参数
> (x, y) => x > y ? x : y;
> ```
>
> 等同于
>
> ```
> function (x){
>     return x *x;
> }
> ```

> 2）箭头函数内多条语句写法
>
> ```
> x => {
>     //Do Something
>     return 0;
> }
> ```
>
> 等同于
>
> ```
> function(x){
>  //Do Something
>  return 0;
> }
> ```

3）如果要返回一个对象，就要注意，如果是单表达式，这么写的话会报错：

```
// SyntaxError:
x => { foo: x }
```

因为和函数体的`{ ... }`有语法冲突，所以要改为：

```
// ok:
x => ({ foo: x })
```

4）之前提到过由于JavaScript函数对this的错误绑定，下面例子无法得到预期结果

```
var obj = {
    birth: 1990,
    getAge: function () {
        var b = this.birth; // 1990
        var fn = function () {
            return new Date().getFullYear() - this.birth; // this指向window或
undefined
        };
        return fn();
    }
};可以用箭头函数解决：
```

```
var obj = {
    birth: 1990,
    getAge: function () {
        var b = this.birth; // 1990
        var fn = () => new Date().getFullYear() - this.birth; // this指向obj对象
        return fn();
    }
};
obj.getAge(); // 25
```

10.generator

> 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。

用generator写一个斐波那契数列函数：

```
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。

1) 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。

2）使用`for of`循环迭代generator对象：

```
for (var x of fib(10)) {
    console.log(x); // 依次输出0, 1, 1, 2, 3, ...
}
```

> generator的优点：
>
> 1） 因为generator可以在执行过程中多次返回，所以它看上去就像一个可以记住执行状态的函数；
>
> 2） 就是把异步回调代码变成“同步”代码 。（AJAX中体现）

十三、标准对象

```
typeof 123; // 'number'
typeof NaN; // 'number'
typeof 'str'; // 'string'
typeof true; // 'boolean'
typeof undefined; // 'undefined'
typeof Math.abs; // 'function'
typeof null; // 'object'
typeof []; // 'object'
typeof {}; // 'object'
```

> 注意`null`是`object`类型。

1)包装对象

```
var n = new Number(123); // 123,生成了新的包装类型
var b = new Boolean(true); // true,生成了新的包装类型
var s = new String('str'); // 'str',生成了新的包装类型

//此时的包装对象类型已经变为了object类型
typeof new Number(123); // 'object'
new Number(123) === 123; // false

typeof new Boolean(true); // 'object'
new Boolean(true) === true; // false

typeof new String('str'); // 'object'
new String('str') === 'str'; // false
```

> `Number()`、`Boolean`和`String()`被当做普通函数，把任何类型的数据转换为`number`、`boolean`和`string`类型（注意不是其包装类型）
>
> ```
> var n = Number('123'); // 123，相当于parseInt()或parseFloat()
> typeof n; // 'number'
>
> var b = Boolean('true'); // true
> typeof b; // 'boolean'
>
> var b2 = Boolean('false'); // true! 'false'字符串转换结果为true! 因为它是非空字符串!
> var b3 = Boolean(''); // false
>
> var s = String(123.45); // '123.45'
> typeof s; // 'string'
> ```

总结来说：

- 不要使用`new Number()`、`new Boolean()`、`new String()`创建包装对象；
- 用`parseInt()`或`parseFloat()`来转换任意类型到`number`；
- 用`String()`来转换任意类型到`string`，或者直接调用某个对象的`toString()`方法；
- 通常不必把任意类型转换为`boolean`再判断，因为可以直接写`if (myVar) {...}`；
- `typeof`操作符可以判断出`number`、`boolean`、`string`、`function`和`undefined`；
- 判断`Array`要使用`Array.isArray(arr)`；
- 判断`null`请使用`myVar === null`；
- 判断某个全局变量是否存在用`typeof window.myVar === 'undefined'`；
- 函数内部判断某个变量是否存在用`typeof myVar === 'undefined'`。

> Number对象调用toString()方法特例：
>
> ```
> 123.toString();//SyntaxError
>
> 123..toString();//正确方法
> ```

1.Date

```
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
```

获取指定日期时间的`Date`对象

```
var d = new Date(2015, 5, 19, 20, 15, 30, 123);//此处传入的月份为5，表示六月！
d; // Fri Jun 19 2015 20:15:30 GMT+0800 (CST)
```

解析一个符合 ISO 8601格式的字符串来获取Date对象：

```
var raw_date = Date.parse('2015-06-24T19:49:22.875+08:00');//解析获得的是一个时间戳
var d = new Date(raw_date);
d;// Wed Jun 24 2015 19:49:22 GMT+0800 (CST)
```

时区：

```
var d = new Date(1435146562875);
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
```

2.正则表达式

1）创建正则表达式

> 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。

```
var re1 = '/ABC\-001/';//类似原生字符串写法，反斜杠不用转义
var re2 = newRegExp('ABC\\-001');//反斜杠需要转义
```

2)正则表达式使用

> 判断是否匹配

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

> 切分字符串

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

> 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）

```
var re = /^(\d{3})-(\d{3,8})$/;//提取-两边的数字

re.exec('010-12345'); // ['010-12345', '010', '12345']
re.exec('010 12345'); // null
```

`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`。

> 贪婪匹配

详见正则表达式速查表

3）全局匹配

> 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
```

> 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。

3.JSON

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。

> JSON的字符串规定必须用双引号`""`，Object的键也必须用双引号`""`

JSON示例

```
{
    "employees": [
    { "firstName":"Bill" , "lastName":"Gates" },
    { "firstName":"George" , "lastName":"Bush" },
    { "firstName":"Thomas" , "lastName":"Carter" }
    ]
}
```

十四、面向对象编程

> JavaScript不区分类和实例的概念，而是通过原型（prototype）来实现面向对象编程。

```
var Student = {
 name: 'Robot',
 height: 1.2,
 run: function () {
     console.log(this.name + ' is running...');
 }
};

var xiaoming = {
 name: '小明'
```

```
};

xiaoming.__proto__ = Student;

xiaoing.name;//'小明'
xiaoming.run();//小明 is running

var Bird = {
 fly: function () {
     console.log(this.name + ' is flying...');
 }
};

xiaoming.__proto__ = Bird;
xiaoming.fly();//小明 is flying...
```

当我们用`obj.xxx`访问一个对象的属性时，JavaScript引擎先在当前对象上查找该属性，如果没有找到，就到其原型对象上找，如果还没有找到，就一直上溯到`Object.prototype`对象，最后，如果还没有找到，就只能返回`undefined`。

1.创建对象

1）直接用`{}`声明

```
var arr = [1, 2, 3];
//原型链为 arr----> Array.prototype ----> Object.prototype ----> null
```

当我们创建一个函数时：

```
function foo() {
    return 0;
}
```

函数也是一个对象，它的原型链是：

```
foo ----> Function.prototype ----> Object.prototype ----> null
```

2)构造函数

```
function Student(name) {
    this.name = name;
    this.hello = function () {
        alert('Hello, ' + this.name + '!');
    }
}

var xiaoming = new Student('小明');
xiaoming.name; // '小明'
xiaoming.hello(); // Hello, 小明!
```

> *注意*，如果不写`new`，这就是一个普通函数，它返回`undefined`。但是，如果写了`new`，它就变成了一个构造函数，它绑定的`this`指向新创建的对象，并默认返回`this`，也就是说，不需要在最后写`return this;`。

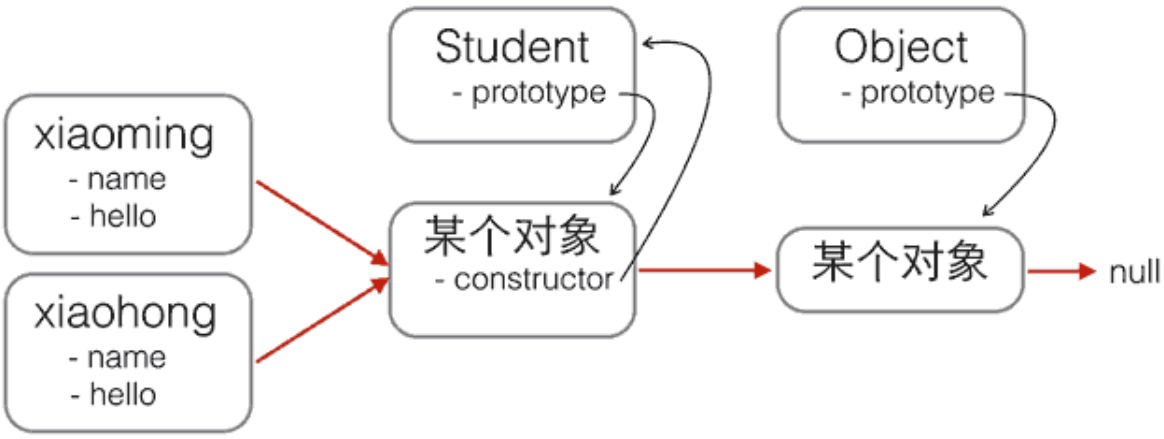
如果你又创建了`xiaohong`、`xiaojun`，那么这些对象的原型与`xiaoming`是一样的：

```
xiaoming ↘
xiaohong -→ Student.prototype ----> Object.prototype ----> null
xiaojun  ↗
```

用`new Student()`创建的对象还从原型上获得了一个`constructor`属性，它指向函数`Student`本身：

```
xiaoming.constructor === Student.prototype.constructor; // true
Student.prototype.constructor === Student; // true

Object.getPrototypeOf(xiaoming) === Student.prototype; // true

xiaoming instanceof Student; // true
```

看晕了吧？用一张图来表示这些乱七八糟的关系就是：



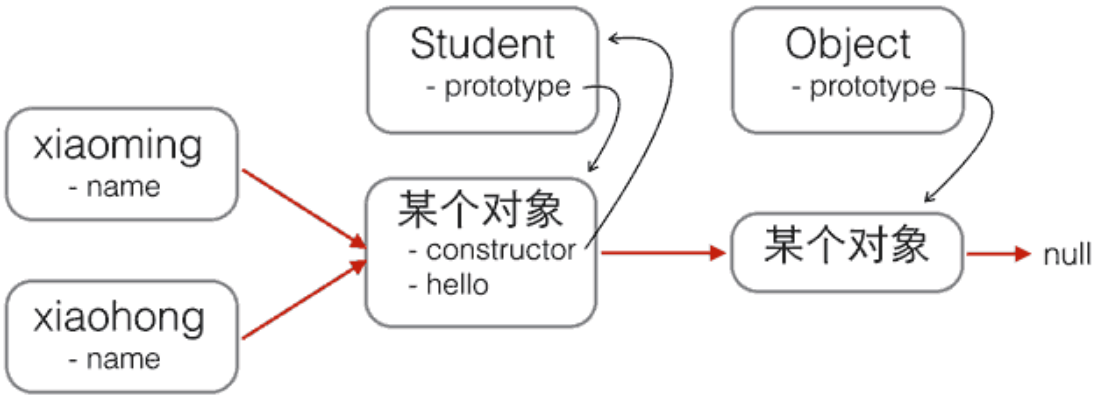
红色箭头是原型链。注意，`Student.prototype`指向的对象就是`xiaoming`、`xiaohong`的原型对象，这个原型对象自己还有个属性`constructor`，指向`Student`函数本身。

另外，函数`Student`恰好有个属性`prototype`指向`xiaoming`、`xiaohong`的原型对象，但是`xiaoming`、`xiaohong`这些对象可没有`prototype`这个属性，不过可以用`__proto__`这个非标准用法来查看。

现在我们就认为`xiaoming`、`xiaohong`这些对象“继承”自`Student`。

> 我的理解：xiaohong/xiaoming对象是某个对象的示例，这个对象的constructor属性指向该对象的构造函数，构造函数的prototype属性指向该对象，这样就形成了一个对象和它的构造函数的绑定。

> 让创建的对象共享一个hello函数：
>
> ```
> function Student(name) {
>  this.name = name;
> }
>
> Student.prototype.hello = function () {
>  alert('Hello, ' + this.name + '!');
> };
> ```
>
> 此时原型之间的关系：



> 如果一个函数被定义为用于创建对象的构造函数，但是调用时忘记了写`new`怎么办？
>
> 在strict模式下，`this.name = name`将报错，因为`this`绑定为`undefined`，在非strict模式下，`this.name = name`不报错，因为`this`绑定为`window`，于是无意间创建了全局变量`name`，并且返回`undefined`，这个结果更糟糕。

3.原型继承（未整理）

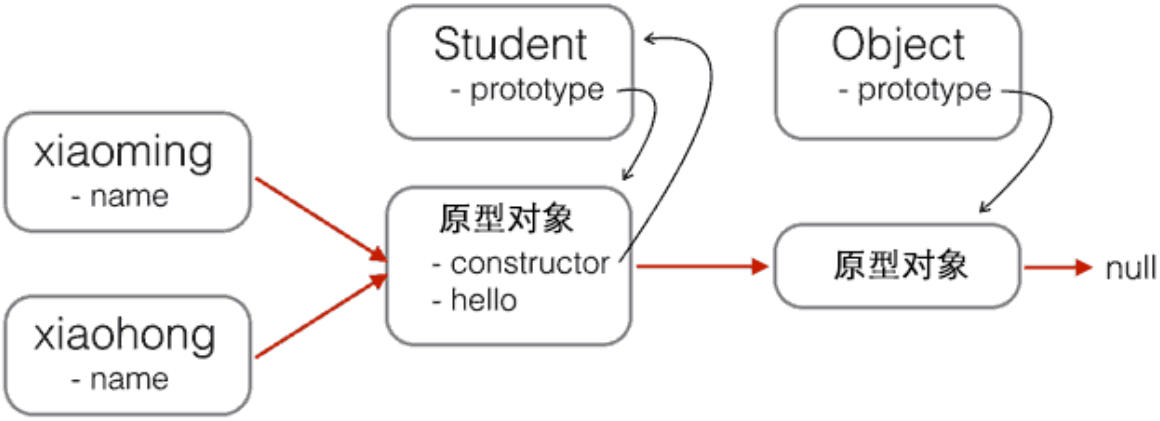
在传统的基于Class的语言如Java、C++中，继承的本质是扩展一个已有的Class，并生成新的Subclass。

由于这类语言严格区分类和实例，继承实际上是类型的扩展。但是，JavaScript由于采用原型继承，我们无法直接扩展一个Class，因为根本不存在Class这种类型。

但是办法还是有的。我们先回顾`Student`构造函数：

```
function Student(props) {
    this.name = props.name || 'Unnamed';
}

Student.prototype.hello = function () {
    alert('Hello, ' + this.name + '!');
}
```

以及`Student`的原型链：



现在，我们要基于`Student`扩展出`PrimaryStudent`，可以先定义出`PrimaryStudent`：

```
function PrimaryStudent(props) {
    // 调用Student构造函数，绑定this变量:
    Student.call(this, props);
    this.grade = props.grade || 1;
}
```

但是，调用了`Student`构造函数不等于继承了`Student`，`PrimaryStudent`创建的对象的原型是：

```
new PrimaryStudent() ----> PrimaryStudent.prototype ----> Object.prototype ----> null
```

必须想办法把原型链修改为：

```
new PrimaryStudent() ----> PrimaryStudent.prototype ----> Student.prototype ----> Object.prototype ----> null
```

这样，原型链对了，继承关系就对了。新的基于`PrimaryStudent`创建的对象不但能调用`PrimaryStudent.prototype`定义的方法，也可以调用`Student.prototype`定义的方法。

如果你想用最简单粗暴的方法这么干：

```
PrimaryStudent.prototype = Student.prototype;
```

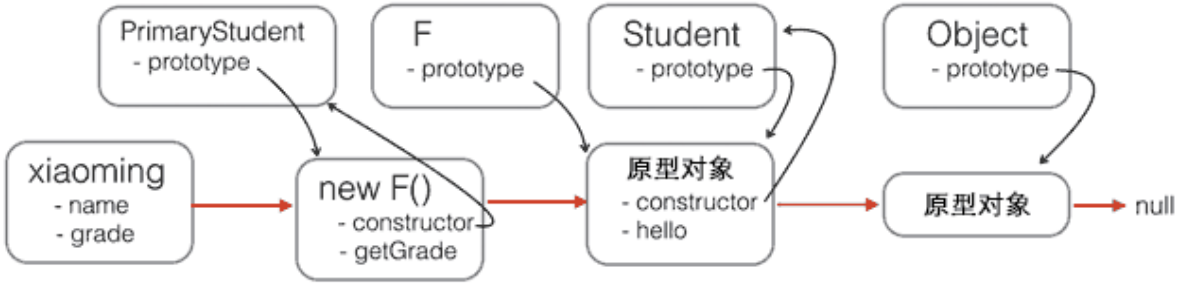
是不行的！如果这样的话，`PrimaryStudent`和`Student`共享一个原型对象，那还要定义`PrimaryStudent`干啥？

我们必须借助一个中间对象来实现正确的原型链，这个中间对象的原型要指向`Student.prototype`。为了实现这一点，参考道爷（就是发明JSON的那个道格拉斯）的代码，中间对象可以用一个空函数`F`来实现：

```
// PrimaryStudent构造函数:
function PrimaryStudent(props) {
    Student.call(this, props);
    this.grade = props.grade || 1;
}

// 空函数F:
function F() {
}

// 把F的原型指向Student.prototype:
F.prototype = Student.prototype;

// 把PrimaryStudent的原型指向一个新的F对象，F对象的原型正好指向Student.prototype:
PrimaryStudent.prototype = new F();

// 把PrimaryStudent原型的构造函数修复为PrimaryStudent:
PrimaryStudent.prototype.constructor = PrimaryStudent;

// 继续在PrimaryStudent原型（就是new F()对象）上定义方法：
PrimaryStudent.prototype.getGrade = function () {
    return this.grade;
};
```

```
// 创建xiaoming:
var xiaoming = new PrimaryStudent({
    name: '小明',
    grade: 2
});
xiaoming.name; // '小明'
xiaoming.grade; // 2

// 验证原型:
xiaoming.__proto__ === PrimaryStudent.prototype; // true
xiaoming.__proto__.__proto__ === Student.prototype; // true

// 验证继承关系:
xiaoming instanceof PrimaryStudent; // true
xiaoming instanceof Student; // true
```

用一张图来表示新的原型链：



注意，函数`F`仅用于桥接，我们仅创建了一个`new F()`实例，而且，没有改变原有的`Student`定义的原型链。

如果把继承这个动作用一个`inherits()`函数封装起来，还可以隐藏`F`的定义，并简化代码：

```
function inherits(Child, Parent) {
    var F = function () {};
    F.prototype = Parent.prototype;
    Child.prototype = new F();
    Child.prototype.constructor = Child;
}
```

这个`inherits()`函数可以复用：

```
function Student(props) {
    this.name = props.name || 'Unnamed';
}

Student.prototype.hello = function () {
    alert('Hello, ' + this.name + '!');
}

function PrimaryStudent(props) {
    Student.call(this, props);
    this.grade = props.grade || 1;
}

// 实现原型继承链:
inherits(PrimaryStudent, Student);
```

```
// 绑定其他方法到PrimaryStudent原型:
PrimaryStudent.prototype.getGrade = function () {
    return this.grade;
};
```

3.class继承（未整理）

在上面的章节中我们看到了JavaScript的对象模型是基于原型实现的，特点是简单，缺点是理解起来比传统的类 - 实例模型要困难，最大的缺点是继承的实现需要编写大量代码，并且需要正确实现原型链。

有没有更简单的写法？有！

新的关键字`class`从ES6开始正式被引入到JavaScript中。`class`的目的就是让定义类更简单。

我们先回顾用函数实现`Student`的方法：

```
function Student(name) {
    this.name = name;
}

Student.prototype.hello = function () {
    alert('Hello, ' + this.name + '!');
}
```

如果用新的`class`关键字来编写`Student`，可以这样写：

```
class Student {
    constructor(name) {
        this.name = name;
    }

    hello() {
        alert('Hello, ' + this.name + '!');
    }
}
```

比较一下就可以发现，`class`的定义包含了构造函数`constructor`和定义在原型对象上的函数`hello()`（注意没有`function`关键字），这样就避免了`Student.prototype.hello = function () {...}`这样分散的代码。

最后，创建一个`Student`对象代码和前面章节完全一样：

```
var xiaoming = new Student('小明');
xiaoming.hello();
```

class继承

用`class`定义对象的另一个巨大的好处是继承更方便了。想一想我们从`Student`派生一个`PrimaryStudent`需要编写的代码量。现在，原型继承的中间对象，原型对象的构造函数等等都不需要考虑了，直接通过`extends`来实现：

```
class PrimaryStudent extends Student {
    constructor(name, grade) {
        super(name); // 记得用super调用父类的构造方法!
        this.grade = grade;
    }

    myGrade() {
        alert('I am at grade ' + this.grade);
    }
}
```

注意`PrimaryStudent`的定义也是class关键字实现的，而`extends`则表示原型链对象来自`Student`。子类的构造函数可能会与父类不太相同，例如，`PrimaryStudent`需要`name`和`grade`两个参数，并且需要通过`super(name)`来调用父类的构造函数，否则父类的`name`属性无法正常初始化。

`PrimaryStudent`已经自动获得了父类`Student`的`hello`方法，我们又在子类中定义了新的`myGrade`方法。

ES6引入的`class`和原有的JavaScript原型继承有什么区别呢？实际上它们没有任何区别，`class`的作用就是让JavaScript引擎去实现原来需要我们自己编写的原型链代码。简而言之，用`class`的好处就是极大地简化了原型链代码。

你一定会问，`class`这么好用，能不能现在就用上？

现在用还早了点，因为不是所有的主流浏览器都支持ES6的class。如果一定要现在就用上，就需要一个工具把`class`代码转换为传统的`prototype`代码，可以试试Babel这个工具。

十五、浏览器

1.浏览器对象

1)window

`window`对象不但充当全局作用域，而且表示浏览器窗口。

> window对象的一些属性：
>
> - innerwidth`:浏览器窗口的内部宽度
> - innerHeight:浏览器窗口的内部高度
> - outerWidth：浏览器窗口的整个宽度
> - outerHeight:浏览器窗口的整个高度

2)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。

3)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。

4)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。

5)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。

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：

```
var menu = document.getElementById('drink-menu');
var drinks = document.getElementsByTagName('dt');
var i, s;

s = '提供的饮料有:';
for (i=0; i<drinks.length; i++) {
    s = s + drinks[i].innerHTML + ',';
}
console.log(s);//提供的饮料有:摩卡,酸奶,果汁,
```

2.操作DOM

1)获取DOM

> `document.getElementById()`:通过id获取DOM;
>
> `document.getElementsByTagName()`:通过标签名获取**一组**DOM
>
> `document.getElementsByClassName()`:通过class属性获取**一组**DOM

> 第二种方法是使用`querySelector()`和`querySelectorAll()`，需要了解selector语法，然后使用条件来获取节点.(略)

练习:

```
<!-- HTML结构 -->
<div id="test-div">
<div class="c-red">
    <p id="test-p">JavaScript</p>
    <p>Java</p>
  </div>
  <div class="c-red c-green">
    <p>Python</p>
    <p>Ruby</p>
    <p>Swift</p>
  </div>
  <div class="c-green">
    <p>Scheme</p>
    <p>Haskell</p>
  </div>
</div>
```

请选择出指定条件的节点:

```
// 选择<p>JavaScript</p>:
var js = document.getElementById('test-p');

// 选择<p>Python</p>,<p>Ruby</p>,<p>Swift</p>:
var arr = document.getElementsByClassName('c-red')[1].children;

// 选择<p>Haskell</p>:
var haskell = document.getElementsByClassName('c-green')[1].children[1];
```

2) 更新DOM

方法①： 修改`innerHTML`属性，这个方式非常强大，不但可以修改一个DOM节点的文本内容，还可以直接通过HTML片段修改DOM节点内部的子树：

```
// 获取<p id="p-id">...</p>
var p = document.getElementById('p-id');
// 设置文本为abc:
p.innerHTML = 'ABC'; // <p id="p-id">ABC</p>
// 设置HTML:
p.innerHTML = 'ABC <span style="color:red">RED</span> XYZ';
// <p>...</p>的内部结构已修改
```

方法②： 修改`innerText`或`textContent`属性，这样可以自动对字符串进行HTML编码，保证无法设置任何HTML标签

```
// 获取<p id="p-id">...</p>
var p = document.getElementById('p-id');
// 设置文本:
p.innerText = '<script>alert("Hi")</script>';
// HTML被自动编码，无法设置一个<script>节点:
// <p id="p-id">&lt;script&gt;alert("Hi")&lt;/script&gt;</p>
```

> DOM节点的`style`属性对应所有的CSS，可以直接获取或设置。
>
> ```
> // 获取 ...
> var p = document.getElementById('p-id');
> // 设置CSS:
> p.style.color = '#ff0000';
> p.style.fontSize = '20px';
> p.style.paddingTop = '2em';
> ```

3）插入DOM

①使用`appendChild`将子节点添加到父节点的最后一个子节点：

```
<!-- HTML结构 -->
<p id="js">JavaScript</p>
<div id="list">
    <p id="java">Java</p>
    <p id="python">Python</p>
    <p id="scheme">Scheme</p>
</div>
```

执行以下JS代码：

```
var
    js = document.getElementById('js'),
    list = document.getElementById('list');
list.appendChild(js);
```

执行后的HTML：

```
<!-- HTML结构 -->
<div id="list">
    <p id="java">Java</p>
    <p id="python">Python</p>
    <p id="scheme">Scheme</p>
    <p id="js">JavaScript</p>
</div>
```

> 因为我们插入的`js`节点已经存在于当前的文档树，因此这个节点首先会从原先的位置删除，再插入到新的位置。

插入一个新建的节点示例：

```
var d = document.createElement('style');
d.setAttribute('type', 'text/css');
d.innerHTML = 'p { color: red }';
document.getElementsByTagName('head')[0].appendChild(d);
```

②使用`insertBefore`将一个节点插入到某个节点前面：

```
<!-- HTML结构 -->
<div id="list">
    <p id="java">Java</p>
    <p id="python">Python</p>
    <p id="scheme">Scheme</p>
</div>
```

可以这么写：

```
var
    list = document.getElementById('list'),
    ref = document.getElementById('python'),
    haskell = document.createElement('p');
haskell.id = 'haskell';
haskell.innerText = 'Haskell';
list.insertBefore(haskell, ref);
```

新的HTML结构如下：

```
<!-- HTML结构 -->
<div id="list">
    <p id="java">Java</p>
    <p id="haskell">Haskell</p>
    <p id="python">Python</p>
    <p id="scheme">Scheme</p>
</div>
```

4）删除节点

> 要删除一个节点，首先要获得该节点本身以及它的父节点，然后，调用父节点的`removeChild`把自己删掉 .

```
// 拿到待删除节点:
var self = document.getElementById('to-be-removed');
// 拿到父节点:
var parent = self.parentElement;
// 删除:
var removed = parent.removeChild(self);
removed === self; // true
```

需要注意的地方：

对于如下HTML结构：

```
<div id="parent">
    <p>First</p>
    <p>Second</p>
</div>
```

当我们用如下代码删除子节点时：

```
var parent = document.getElementById('parent');
parent.removeChild(parent.children[0]);
parent.removeChild(parent.children[1]); // <-- 浏览器报错
```

浏览器报错：`parent.children[1]`不是一个有效的节点。原因就在于，当`First`节点被删除后，`parent.children`的节点数量已经从2变为了1，索引`[1]`已经不存在了。

因此，删除多个节点时，要注意`children`属性时刻都在变化。

3.操作表单

1） 获取值

> 获取`input`、`text`、`password`、`hidden`以及`select`等值

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

> 获取单选框和复选框的值
>
> ```
> // <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者false
> ```

2)设置值

基本同获取值，直接改变value就行。

3）提交表单

> 通过元素的`submit()`方法提交一个表单 .

```
<!-- 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。 -->
```

> 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 。

4.操作文件

在HTML表单中，可以上传文件的唯一控件就是`<input type="file">`。

出于安全考虑，浏览器只允许用户点击`来选择本地文件，用JavaScript对`的`value`赋值是没有任何效果的。当用户选择了上传某个文件后，JavaScript也无法获得该文件的真实路径：

```
var f = document.getElementById('test-file-upload');
var filename = f.value; // 'C:\fakepath\test.png'
if (!filename || !(filename.endsWith('.jpg') || filename.endsWith('.png') ||
filename.endsWith('.gif'))) {
    alert('Can only upload image file.');
    return false;
}
```

File API

由于JavaScript对用户上传的文件操作非常有限，尤其是无法读取文件内容，使得很多需要操作文件的网页不得不用Flash这样的第三方插件来实现。

随着HTML5的普及，新增的File API允许JavaScript读取文件内容，获得更多的文件信息。

HTML5的File API提供了`File`和`FileReader`两个主要对象，可以获得文件信息并读取文件。

5.AJAX

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 ActiveXObject('Microsoft.XMLHTTP'); // 新建Microsoft.XMLHTTP
对象
//var request = new XMLHttpRequest(); // 新建XMLHttpRequest对象
var request;
if (window.XMLHttpRequest) {
    request = new XMLHttpRequest();
} else {
    request = new ActiveXObject('Microsoft.XMLHTTP');
}
//通过检测window对象是否有XMLHttpRequest属性来确定浏览器是否支持标准的XMLHttpRequest。

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');//URL地址为相对地址
request.send();

alert('请求已发送，请等待响应...');
```

当创建了`XMLHttpRequest`对象后，要先设置`onreadystatechange`的回调函数。在回调函数中，通常我们只需通过`readyState === 4`判断请求是否完成，如果已完成，再根据`status === 200`判断是否是一个成功的响应。

`XMLHttpRequest`对象的`open()`方法有3个参数，第一个参数指定是`GET`还是`POST`，第二个参数指定URL地址，第三个参数指定是否使用异步，默认是`true`，所以不用写。

注意，千万不要把第三个参数指定为`false`，否则浏览器将停止响应，直到AJAX请求完成。如果这个请求耗时10秒，那么10秒内你会发现浏览器处于“假死”状态。

最后调用`send()`方法才真正发送请求。`GET`请求不需要参数，`POST`请求需要把body部分以字符串或者`FormData`对象传进去。

> 跨域JSONP方法见博客：https://www.cnblogs.com/dowinning/archive/2012/04/19/json-jsonp-jquery.html

示例写法

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
    <script type="text/javascript">
    // 得到航班信息查询结果后的回调函数
    var flightHandler = function(data){
        alert('你查询的航班结果是：票价 ' + data.price + ' 元，' + '余票 ' +
data.tickets + ' 张。');
    };
    // 提供jsonp服务的url地址（不管是什么类型的地址，最终生成的返回值都是一段javascript代
码）
    var url = "http://flightQuery.com/jsonp/flightResult.aspx?
code=CA1998&callback=flightHandler";
    // 创建script标签，设置其属性
    var script = document.createElement('script');
    script.setAttribute('src', url);
    // 把script标签加入head，此时调用开始
    document.getElementsByTagName('head')[0].appendChild(script);
    </script>
</head>
<body>
</body>
</html>
```

6.Promise

略

7.Canvas

略

十六、 jQuery

1.选择器

按ID查找DOM节点:

```
// 查找<div id="abc">:
var div = $('#abc');
```

按tag查找:

```
var ps = $('p'); // 返回所有<p>节点
ps.length; // 数一数页面有多少个<p>节点
```

按class查找:

```
// <div class="red">...</div>
// <div class="red green">...</div>
var a = $('.red'); // 所有节点包含`class="red"`都将返回
var b = $('.red.green');//
```

按属性查找:

```
var email = $('[name=email]'); // 找出<??? name="email">
var icons = $('[name^=icon]'); // 找出所有name属性值以icon开头的DOM
// 例如: name="icon-1", name="icon-2"
var names = $('[name$=with]'); // 找出所有name属性值以with结尾的DOM
// 例如: name="startswith", name="endswith"
```

组合查找:

```
//查找属性name= email的input节点
var emailInput = $('input[name=email]'); // 不会找出<div name="email">

var tr = $('tr.red'); // 找出<tr class="red ...">...</tr>
```

多项查找:

```
$('p,div'); // 把<p>和<div>都选出来
$('p.red,p.green'); // 把<p class="red">和<p class="green">都选出来

//多项查找出的元素是按照HTML中的出现顺序排列，并且不会重复
```

> *注意*，`#abc`以`#`开头。
>
> 各个选择器返回的对象是jQuery对象。
>
> 什么是jQuery对象？jQuery对象类似数组，它的每个元素都是一个引用了DOM节点的对象。如果不存在，则返回一个[]。

jQuery对象与DOM对象相互转化：

```
var div = $('#abc'); // jQuery对象
var divDom = div.get(0); // 假设存在div，获取第1个DOM元素
var another = $(divDom); // 重新把DOM包装为jQuery对象
```

1）层级选择器: 如果两个DOM元素具有层级关系，就可以用`$('ancestor descendant')`来选择，层级之间用空格隔开

```
<!-- HTML结构 -->
<div class="testing">
    <ul class="lang">
        <li class="lang-javascript">JavaScript</li>
        <li class="lang-python">Python</li>
        <li class="lang-lua">Lua</li>
    </ul>
</div>
```

```
$('ul.lang li.lang-javascript'); // [<li class="lang-
javascript">JavaScript</li>]
$('div.testing li.lang-javascript'); // [<li class="lang-
javascript">JavaScript</li>]
```

多层选择示例：

```
$('form.test p input'); // 在form表单选择被<p>包含的<input>
```

2) 子选择器

子选择器`$('parent>child')`类似层级选择器，但是限定了层级关系必须是父子关系，就是`child`节点必须是`parent`节点的直属子节点。

3） 过滤器一般不单独使用，它通常附加在选择器上，帮助我们更精确地定位元素。

```
<!-- HTML结构 -->
<div class="testing">
    <ul class="lang">
        <li class="lang-javascript">JavaScript</li>
        <li class="lang-python">Python</li>
        <li class="lang-lua">Lua</li>
    </ul>
</div>

$('ul.lang li'); // 选出JavaScript、Python和Lua 3个节点
$('ul.lang li:first-child'); // 仅选出JavaScript
$('ul.lang li:last-child'); // 仅选出Lua
$('ul.lang li:nth-child(2)'); // 选出第N个元素，N从1开始
$('ul.lang li:nth-child(even)'); // 选出序号为偶数的元素
```

```
$('ul.lang li:nth-child(odd)'); // 选出序号为奇数的元素
```

4）表单相关

针对表单元素，jQuery还有一组特殊的选择器：

- `:input`：可以选择`input`，`textarea`，`select`和`button`；
- `:file`：可以选择`<input type="file"`，和`input[type=file]`一样；
- `:checkbox`：可以选择复选框，和`input[type=checkbox]`一样；
- `:radio`：可以选择单选框，和`input[type=radio]`一样；
- `:focus`：可以选择当前输入焦点的元素，例如把光标放到一个`<input>`上，用`$('input:focus')`就可以选出；
- `:checked`：选择当前勾上的单选框和复选框，用这个选择器可以立刻获得用户选择的项目，如`$('input[type=radio]:checked')`；
- `:enabled`：可以选择可以正常输入的`<input>`、`<select>`等，也就是没有灰掉的输入；
- `:disabled`：和`:enabled`正好相反，选择那些不能输入的。

此外，jQuery还有很多有用的选择器，例如，选出可见的或隐藏的元素：

```
$('div:visible'); // 所有可见的div
$('div:hidden'); // 所有隐藏的div
```

2.查找和过滤

通常情况下选择器可以直接定位到我们想要的元素，但是，当我们拿到一个jQuery对象后，还可以以这个对象为基准，进行查找和过滤。

最常见的查找是在某个节点的所有子节点中查找，使用`find()`方法，它本身又接收一个任意的选择器。

```
<!-- HTML结构 -->
<ul class="lang">
    <li class="js dy">JavaScript</li>
    <li class="dy">Python</li>
    <li id="swift">Swift</li>
    <li class="dy">Scheme</li>
    <li name="haskell">Haskell</li>
</ul>
```

1) 用`find()`查找：

```
var ul = $('ul.lang'); // 获得<ul>
var dy = ul.find('.dy'); // 获得JavaScript, Python, Scheme
var swf = ul.find('#swift'); // 获得Swift
var hsk = ul.find('[name=haskell]'); // 获得Haskell
```

2) 如果要从当前节点开始向上查找，使用`parent()`方法：

```
var swf = $('#swift'); // 获得Swift
var parent = swf.parent(); // 获得Swift的上层节点<ul>
var a = swf.parent('.red'); // 获得Swift的上层节点<ul>，同时传入过滤条件。如果ul不符合条件，返回空jQuery对象
```

对于位于同一层级的节点，可以通过`next()`和`prev()`方法，例如：

当我们已经拿到`Swift`节点后：

```
var swift = $('#swift');

swift.next(); // Scheme
swift.next('[name=haskell]'); // 空的jQuery对象，因为Swift的下一个元素Scheme不符合条件[name=haskell]

swift.prev(); // Python
swift.prev('.dy'); // Python，因为Python同时符合过滤器条件.dy
```

3) `filter()`方法可以过滤掉不符合选择器条件的节点：

```
var langs = $('ul.lang li'); // 拿到JavaScript, Python, Swift, Scheme和Haskell
var a = langs.filter('.dy'); // 拿到JavaScript, Python, Scheme

var langs = $('ul.lang li'); // 拿到JavaScript, Python, Swift, Scheme和Haskell
langs.filter(function () {
    return this.innerHTML.indexOf('S') === 0; // 返回S开头的节点
}); // 拿到Swift, Scheme
```

4) `map()`方法把一个jQuery对象包含的若干DOM节点转化为其他对象

```
var langs = $('ul.lang li'); // 拿到JavaScript, Python, Swift, Scheme和Haskell
var arr = langs.map(function () {
    return this.innerHTML;
}).get(); // 用get()拿到包含string的Array：['JavaScript', 'Python', 'Swift', 'Scheme', 'Haskell']
```

```
var langs = $('ul.lang li'); // 拿到JavaScript, Python, Swift, Scheme和Haskell
var js = langs.first(); // JavaScript，相当于$('ul.lang li:first-child')
var haskell = langs.last(); // Haskell，相当于$('ul.lang li:last-child')
var sub = langs.slice(2, 4); // Swift, Scheme, 参数和数组的slice()方法一致
```

3.操作DOM

```
<!-- HTML结构 -->
<ul id="test-ul">
    <li class="js">JavaScript</li>
    <li name="book">Java &amp;amp; JavaScript</li>
</ul>
```

1）修改文本

`text()`/`html()`方法无参为获取文本 /原始HTML文本， 传入参数就变成设置文本 。

```
//获取文本
$('#test-ul li[name=book]').text(); // 'Java & JavaScript'
$('#test-ul li[name=book]').html(); // 'Java &amp;amp; JavaScript'
```

```
//修改文本
var j1 = $('#test-ul li.js');
var j2 = $('#test-ul li[name=book]');
j1.html('<span style="color: red">JavaScript</span>');
j2.text('JavaScript & ECMAScript');
```

一个jQuery对象可以包含0个或任意个DOM对象，它的方法实际上会作用在对应的每个DOM节点上。在上面的例子中试试：

```
$('#test-ul li').text('JS'); // 两个节点都变成了JS
```

2) 修改CSS

> 修改CSS使用jQuery对象的`css('name', 'value')`方法。

```
$('#test-css li.dy span').css('background-color', '#ffd351').css('color',
'red');
//一次修改多个CSS属性

//判断是否含有、添加、修改CSS样式
var div = $('#test-div');
div.hasClass('highlight'); // false， class是否包含highlight
div.addClass('highlight'); // 添加highlight这个class
div.removeClass('highlight'); // 删除highlight这个class
```

3) 显示和隐藏DOM

```
var a = $('a[target=_blank]');
a.hide(); // 隐藏
a.show(); // 显示
```

4) 获取DOM信息

```
// 某个div的大小:
var div = $('#test-div');
div.width(); // 600
div.height(); // 300
div.width(400); // 设置CSS属性 width: 400px，是否生效要看CSS是否有效
div.height('200px'); // 设置CSS属性 height: 200px，是否生效要看CSS是否有效
```

`attr()`和`removeAttr()`方法用于操作DOM节点的属性：

```
// <div id="test-div" name="Test" start="1">...</div>
var div = $('#test-div');
div.attr('data'); // undefined, 属性不存在
div.attr('name'); // 'Test'
div.attr('name', 'Hello'); // div的name属性变为'Hello'
div.removeAttr('name'); // 删除name属性
div.attr('name'); // undefined
//设置复选框时常用prop("checked", true)
```

> 判断单选/复选框是否选择：

```
var radio = $('#test-radio');
radio.is(':checked'); // true
```

对于表单元素，jQuery对象统一提供`val()`方法获取和设置对应的`value`属性：

```
/*
    <input id="test-input" name="email" value="">
*/
var input = $('#test-input'),
input.val(); // 'test'
input.val('abc@example.com'); // 文本框的内容已变为abc@example.com
```

5) 使用`append()`添加DOM

```
<div id="test-div">
    <ul>
        <li><span>JavaScript</span></li>
        <li><span>Python</span></li>
        <li><span>Swift</span></li>
    </ul>
</div>
```

```
// 创建DOM对象:
var ps = document.createElement('li');
ps.innerHTML = '<span>Pascal</span>';
// 添加DOM对象:
ul.append(ps);

// 添加jQuery对象:
ul.append($('#scheme'));

// 添加函数对象:
ul.append(function (index, html) {
    return '<li><span>Language - ' + index + '</span></li>';
});

//同级节点可以用after()或者before()方法。
var js = $('#test-div>ul>li:first-child');
js.after('<li><span>Lua</span></li>');
```

> 另外**注意**，如果要添加的DOM节点已经存在于HTML文档中，它会首先从文档移除，然后再添加。

6) 删除节点

```
var li = $('#test-div>ul>li');
li.remove(); // 所有<li>全被删除
```

练习：除了列出的3种语言外，请再添加Pascal、Lua和Ruby，然后按字母顺序排序节点：

```
<!-- HTML结构 -->
<div id="test-div">
    <ul>
        <li><span>JavaScript</span></li>
        <li><span>Python</span></li>
        <li><span>Swift</span></li>
    </ul>
</div>
```

```
var ul = $('#test-div ul');
var lan = ['Pascal', 'Lua', 'Ruby'];
lan.map(function(x){
    ul.append( `<li><span>${x}</span></li>`);
});
var lis = ul.find('li');
lis.sort(function(li1, li2){
    return li1.innerHTML > li2.innerHTML ? 1: -1;
});
ul.append(lis);
```

4. 事件

为一个jQuery对象绑定事件：

```
var a = $('#test-link');
a.on('click', function () {
    alert('Hello!');
});
```

1）事件列举

鼠标事件：

> - click: 鼠标单击时触发
> - dblclick：鼠标双击时触发
> - mouseenter：鼠标进入时触发
> - mouseleave：鼠标移出时触发
> - mousemove：鼠标在DOM内部移动时触发

键盘事件

> - keydown：键盘按下时触发
> - keyup：键盘松开时触发
> - keypress：按一次键后触发

其他事件

> - focus：当DOM获得焦点时触发
> - blur：当DOM失去焦点时触发
> - change：当`<input>`、`<select>`或`<textarea>`的内容改变时触发

两个可简写的事件：

①点击事件：

```
a.click(function () {
    alert('Hello!');
});
```

②页面初始化事件：

```
$(document).on('ready', function () {
     $('#testForm).on('submit', function () {
           alert('submit!');
      });
});
```

甚至还可以再简化为：

```
$(function () {
    // init...
});
//可连续绑定多个事件，它们会逐个执行
```

2） 事件参数

有些事件，如`mousemove`和`keypress`，我们需要获取鼠标位置和按键的值，否则监听这些事件就没什么意义了。所有事件都会传入`Event`对象作为参数，可以从`Event`对象上获取到更多的信息：

```
$(function () {
    $('#testMouseMoveDiv').mousemove(function (e) {
        $('#testMouseMoveSpan').text('pageX = ' + e.pageX + ', pageY = ' +
e.pageY);
        //page.x：鼠标X坐标
        //page.y: 鼠标Y坐标
    });
});
```

3）取消绑定: `off('click', function)`

```
function hello() {
    alert('hello!');
}

a.click(hello); // 绑定事件

// 10秒钟后解除绑定:
setTimeout(function () {
    a.off('click', hello);
}, 10000);
```

4)事件触发条件

```
var input = $('#test-input');
input.change(function () {
    console.log('changed...');
});
//代码改变input的值时，是无法触发change事件的
//想要用代码触发change事件，可无参调用change()来触发
//input.change();
```

5.动画

1)show / hide

```
var div = $('#test-show-hide');
div.show(600); // 在0.6秒钟内逐渐显示,等同于 div.show('slow')
div.hide(600);// 在0.6秒钟内逐渐消失

//toggle()方法则根据当前状态决定是show()还是hide()。
```

2) slideUp / slideDown

```
var div = $('#test-slide');
div.slideUp(3000); // 在3秒钟内逐渐向上消失
div.slideDown(3000); //在3秒钟内向下显示
//slideToggle()则根据元素是否可见来决定下一步动作
```

3) fadeIn / fadeOut

```
var div = $('#test-fade');
div.fadeOut('slow'); // 在0.6秒内淡出
//fadeToggle()则根据元素是否可见来决定下一步动作
```

4) 自定义动画 `animate()`

略

6.AJAX

jQuery在全局对象`jQuery`（也就是`$`）绑定了`ajax()`函数，可以处理AJAX请求。`ajax(url, settings)`函数需要接收一个URL和一个可选的`settings`对象，常用的选项如下：

- async：是否异步执行AJAX请求，默认为`true`，千万不要指定为`false`；
- method：发送的Method，缺省为`'GET'`，可指定为`'POST'`、`'PUT'`等；
- contentType：发送POST请求的格式，默认值为`'application/x-www-form-urlencoded; charset=UTF-8'`，也可以指定为`text/plain`、`application/json`；
- data：发送的数据，可以是字符串、数组或object。如果是GET请求，data将被转换成query附加到URL上，如果是POST请求，根据contentType把data序列化成合适的格式；
- headers：发送的额外的HTTP头，必须是一个object；
- dataType：接收的数据格式，可以指定为`'html'`、`'xml'`、`'json'`、`'text'`等，缺省情况下根据响应的`Content-Type`猜测。

一个AJAX示例

```
var jqxhr = $.ajax('/api/categories', {
    dataType: 'json'
}).done(function (data) {
    ajaxLog('成功, 收到的数据: ' + JSON.stringify(data));
}).fail(function (xhr, status) {
    ajaxLog('失败: ' + xhr.status + ', 原因: ' + status);
}).always(function () {
    ajaxLog('请求完成: 无论成功或失败都会调用');
});
```

自己写的Ajax请求模板

```
<script type="text/javascript">
    function a1() {
        $.ajax({
            url:"${pageContext.request.contextPath}" + "/Servlet1",
            dataType:'json',//数据类型
            type:'GET',//类型
            timeout:2000,//超时
            data:{"name":$("#txtName").val()},//后台可获取参数
            success:function (data, status) {
                console.log("请求成功：" + status);
                $('#sp').html("名字：" + data.name + ",年龄：" + data.age);
            },
            error:function(XMLHttpRequest,textStatus,errorThrown){
                if(textStatus==='timeout'){
                    alert('請求超時');
                    setTimeout(function(){
                        alert('重新请求');
                    },2000);
                 }
                    //alert(errorThrown);
            }
         })
    }
</script>

/* 后台Servlet程序
@WebServlet(name = "Servlet1", urlPatterns = "/Servlet1")
public class Servlet1 extends HttpServlet {
    protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        String name = request.getParameter("name");
        if("xiaoming".equals(name)){
            response.setCharacterEncoding("utf-8");
            response.getWriter().write("{\"name\":\"小明\", \"age\": 12}");
        }
        else
        {
            response.getWriter().write("false");
        }
    }
    protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
        doPost(request, response);
    }
```

```
}
*/
```

7.拓展 （未整理）

当我们使用jQuery对象的方法时，由于jQuery对象可以操作一组DOM，而且支持链式操作，所以用起来非常方便。

但是jQuery内置的方法永远不可能满足所有的需求。比如，我们想要高亮显示某些DOM元素，用jQuery可以这么实现：

```
$('span.hl').css('backgroundColor', '#fffceb').css('color', '#d85030');

$('p a.hl').css('backgroundColor', '#fffceb').css('color', '#d85030');
```

总是写重复代码可不好，万一以后还要修改字体就更麻烦了，能不能统一起来，写个`highlight()`方法?

```
$('span.hl').highlight();

$('p a.hl').highlight();
```

答案是肯定的。我们可以扩展jQuery来实现自定义方法。将来如果要修改高亮的逻辑，只需修改一处扩展代码。这种方式也称为编写jQuery插件。

编写jQuery插件

给jQuery对象绑定一个新方法是通过扩展`$.fn`对象实现的。让我们来编写第一个扩展——`highlight1()`：

```
$.fn.highlight1 = function () {
    // this已绑定为当前jQuery对象:
    this.css('backgroundColor', '#fffceb').css('color', '#d85030');
    return this;
}
```

注意到函数内部的`this`在调用时被绑定为jQuery对象，所以函数内部代码可以正常调用所有jQuery对象的方法。

细心的童鞋可能发现了，为什么最后要`return this;`？因为jQuery对象支持链式操作，我们自己写的扩展方法也要能继续链式下去：

```
$('span.hl').highlight1().slideDown();
```

不然，用户调用的时候，就不得不把上面的代码拆成两行。

但是这个版本并不完美。有的用户希望高亮的颜色能自己来指定，怎么办?

我们可以给方法加个参数，让用户自己把参数用对象传进去。于是我们有了第二个版本的`highlight2()`：

```
$.fn.highlight2 = function (options) {
    // 要考虑到各种情况:
    // options为undefined
    // options只有部分key
    var bgcolor = options && options.backgroundColor || '#fffceb';
    var color = options && options.color || '#d85030';
    this.css('backgroundColor', bgcolor).css('color', color);
    return this;
}
```

对于默认值的处理，我们用了一个简单的`&&`和`||`短路操作符，总能得到一个有效的值。

另一种方法是使用jQuery提供的辅助方法`$.extend(target, obj1, obj2, ...)`，它把多个object对象的属性合并到第一个target对象中，遇到同名属性，总是使用靠后的对象的值，也就是越往后优先级越高：

```
// 把默认值和用户传入的options合并到对象{}中并返回:
var opts = $.extend({}, {
    backgroundColor: '#00a8e6',
    color: '#ffffff'
}, options);
```

紧接着用户对`highlight2()`提出了意见：每次调用都需要传入自定义的设置，能不能让我自己设定一个缺省值，以后的调用统一使用无参数的`highlight2()`？

也就是说，我们设定的默认值应该能允许用户修改。

那默认值放哪比较合适？放全局变量肯定不合适，最佳地点是`$.fn.highlight2`这个函数对象本身。

于是最终版的`highlight()`终于诞生了：

```
$.fn.highlight = function (options) {
    // 合并默认值和用户设定值:
    var opts = $.extend({}, $.fn.highlight.defaults, options);
    this.css('backgroundColor', opts.backgroundColor).css('color', opts.color);
    return this;
}

// 设定默认值:
$.fn.highlight.defaults = {
    color: '#d85030',
    backgroundColor: '#fff8de'
}
```

这次用户终于满意了。用户使用时，只需一次性设定默认值：

```
$.fn.highlight.defaults.color = '#fff';
$.fn.highlight.defaults.backgroundColor = '#000';
```

然后就可以非常简单地调用`highlight()`了。

最终，我们得出编写一个jQuery插件的原则：

1. 给`$.fn`绑定函数，实现插件的代码逻辑；
2. 插件函数最后要`return this;`以支持链式调用；
3. 插件函数要有默认值，绑定在`$.fn..defaults`上；

4. 用户在调用时可传入设定值以便覆盖默认值。

十七、错误处理

十八、underscore

十九、 Node.js

二十、平时用到的东西

```
//防止浏览器回退
$(document).ready(function(e) {
    //禁止回退
    var counter = 0;
    if (window.history && window.history.pushState) {
        $(window).on('popstate', function () {
            window.history.pushState('forward', null, '#');
            //window.location.href="ad.html";   // 回退时跳转到新页面
            window.history.forward(1);
            msg(0, '我看到你了',"不要动小心思呦(^_-)☆");
        });
    }
    window.history.pushState('forward', null, '#'); //在IE中必须得有这两行
    window.history.forward(1);
});
```

```
//浏览器本地暂存东西
localStorage.setItem('key', 1);
localStorage.getItem("key");//1
```