第6章 对象

在前面，我们有介绍过，在JavaScript中，数据类型整体上来讲可以分为两大类：简单数据类型和复杂数据类型。

简单数据类型一共有6种，分别是我们前面介绍过的
`string`， `number`， `boolean`， `null`， `undefined`以及本章将会介绍的`symbol`

而复杂数据类型就只有1种
`object`
也就是我们本章将会向大家所介绍的对象。事实上，我们前面已经遇到过一些对象了。例如之前所介绍过的数组就是一种对象，不过它是JavaScript语言中内置的对象。在本章中我们将学习到如何自定义对象，以及一些其他的内置对象。

本章将学习如下内容：

- 创建对象的方式
- 访问对象属性的方式
- 对象常用的属性和方法
- 对象的嵌套与解构
- this关键字
- JSON对象
- Math对象
- Date对象
- 正则表达式

6-1 对象基础介绍

6-1-1 JS对象概述

JavaScript里面的对象就是一组键值对的集合。这些键一般由字符串构成，而值可以是任意数据类型。比如字符串，数字，布尔，数组或者函数。一般来讲，如果一个键映射的是一个非函数的值，我们将这个值称之为该对象的属性，而如果一个键映射的是一个函数的值，那么我们将其称之为方法。

6-1-2 创建对象

要创建一个对象，我们只需要输入一对大括号即可。这样我们就可以创建一个空的对象，如下：

```
let objName = {};
```

创建好对象以后，我们就可以给该对象添加相应的属性，例如这里我们给`xiejie`这个对象添加相应的属性

```
let xiejie = {};
xiejie.name = "xiejie";
xiejie.age = 18;
xiejie.gender = "male";
xiejie.score = 100;
```

当然，和前面为大家介绍过的数组一样，我们可以在创建对象时就给对象添加好属性信息，如下：

```
let xiejie = {
    name : "xiejie",
    age : 18,
    gender : "male",
    score : 100
};
```

可以看到，当我们创建包含属性的对象的时候，属性与属性之间是以逗号隔开的。这里我们可以将属性名称之为键，属性对应的称之为值。所以，正如开头我们所介绍的那样，对象是由一个一个**键值对**组成的。

6-1-3 访问对象属性

访问对象的属性的方法有3种：点访问法，中括号访问法，symbol访问法

1. 点访问法

我们可以通过一个点`.`来访问到对象的属性，如下：

```
let xiejie = {
    name : "xiejie",
    age : 18,
    gender : "male",
    score : 100
};
```

```
console.log(xiejie.name);//xiejie
console.log(xiejie.age);//18
console.log(xiejie.gender);//male
console.log(xiejie.score);//100
```

2. 中括号访问法

第二种方法，是使用中括号法来访问对象的属性，如下：

```
let xiejie = {
    name : "xiejie",
    age : 18,
    gender : "male",
    score : 100
};
console.log(xiejie["name"]);//xiejie
console.log(xiejie["age"]);//18
console.log(xiejie["gender"]);//male
console.log(xiejie["score"]);//100
```

一般来讲，访问对象属性的时候使用点访问法的情况要多一些，那什么时候使用中括号访问方法呢？当我们的属性名来自于变量的时候，这个时候中括号就要比点要灵活许多。来看下面的例子：

```
let xiejie = {
    name : "xiejie",
    age : 18,
    gender : "male",
    score : 100
};
let str = "name";
console.log(xiejie[str]);//xiejie
```

既然讲到了对象属性的中括号访问法，那我们就顺带介绍一下伪数组对象的原理。前面给大家介绍过的arguments就是一个伪数组对象。伪数组对象的原理就在于对象的键都是数字。如果属性名是数字的话，通过中括号法来访问时可以不用添加引号，如下：

```
let obj = {
    1 : "Bill",
    2 : "Lucy",
    3 : "David"
}
console.log(obj[1]);//Bill
```

```
console.log(obj[2]);//Lucy
console.log(obj[3]);//David
```

这样，就形成了给人感觉像是数组，但是并不是数组的伪数组对象。

3. symbol访问法

在ES6之前，对象的属性名都只能是字符串。但是这样很容易造成属性名的冲突。比如我们使用了一个别人提供的对象，然后我们想在这个对象的基础上进行一定的扩展，添加新的属性，这个时候由于并不知道原来的对象里面包含哪些属性名，所以很容易就把别人的对象所具有的属性给覆盖掉了。示例如下：

```
//假设person对象是从外部库引入的一个对象
let person = {
    name : "xiejie"
}
console.log(person.name);//xiejie
person.name = "yajing";
console.log(person.name);//yajing
```

可以看到，这里两个name就产生了冲突，下面的name就把上面的name给覆盖掉了。

从ES6开始，新增了symbol这种数据类型，专门来解决这样的问题。创建symbol，需要使用 `Symbol()` 函数，其语法如下：

```
let sym = Symbol(描述信息);
```

示例：

```
let name = Symbol("这是一个名字");
console.log(name);//Symbol(这是一个名字)
console.log(typeof name);//symbol
```

这里的描述信息是可选的，是对我们自己创建的symbol的一个描述。接下来我们来用symbol作为对象的属性，示例如下：

```
let person = {
    name : "xiejie"
}
let name = Symbol("这是一个名字");
person[name] = "yajing";
```

```
console.log(person.name);//xiejie
console.log(person[name]);//yajing
```

可以看到，使用symbol来作为对象的属性，避免了同名的属性名发生冲突。

有些时候我们希望在不同的代码中共享一个symbol，那么这个时候可以使用`Symbol.for()`方法来创建一个共享的symbol。ES6提供了一个可以随时访问的全局symbol注册表。当我们使用`Symbol.for()`方法注册一个symbol的时候，系统会首先在全局表里面查找对应的参数的symbol是否存在，如果存在，直接返回已经有的symbol，如果不存在，则在全局表里面创建一个新的symbol

```
let obj = {};
let name = Symbol.for("test");
obj[name] = "xiejie";
let name2 = Symbol.for("test");
console.log(obj[name2]);//xiejie
```

如果使用`Symbol.for()`方法创建symbol的时候没有传递任何参数，那么也会将undefined作为全局表里面的键来进行注册，证明如下：

```
let obj = {};
let name = Symbol.for();
obj[name] = "xiejie";
let name2 = Symbol.for(undefined);
console.log(obj[name2]);//xiejie
```

ES6里面还提供了`Symbol.keyFor()`方法来查找一个symbol的键是什么。但是需要注意的是，该方法只能找到注册到全局表里面的symbol的键。如果是通过Symbol()方法创建的symbol，是无法找到的。这其实也很好理解，通过`Symbol()`方法创建的symbol都不存在有键。

```
let obj = {};
let name1 = Symbol("test1");
let name2 = Symbol.for("test2");
let i = Symbol.keyFor(name1);
let j = Symbol.keyFor(name2);
console.log(i);//udnefined
console.log(j);//test2
```

前面有提到，如果一个对象的属性对应的是一个函数，那么这个函数被称之为对象的方法。访问对象方法的方式和上面介绍的访问对象属性的方式是一样的，可以通过点访问法，中括号访问法以及symbol访问法来进行对象方法的调用。

```
let walk = Symbol("this is a test");
let person = {
    name : "xiejie",
    walk : function(){
        console.log("I'm walking");
    },
    [walk] : function(){
        console.log("I'm walking,too");
    }
}
person.walk();//I'm walking
person["walk"]();//I'm walking
person[walk]();//I'm walking,too
```

6-1-4 删除对象属性

对象的任何属性都可以通过`delete`运算符来从对象中删除。示例如下：

```
let person = {
    name : "xiejie",
    age : 18,
    walk : function(){
        console.log("I'm walking");
    }
}
console.log(person.age);//18
delete person.age;//删除age这个属性
console.log(person.age);//undefined
person.walk();//I'm walking
delete person.walk;//删除walk方法
person.walk();//报错
//TypeError: person.walk is not a function
```

如果是删除的是属性，那么再次访问值为变为undefined，而如果删除的是方法，那么调用时会直接报错

6-1-5 对象常用属性和方法

1. in操作符

该操作符用于判断一个对象是否含有某一个属性，如果有返回true，没有返回false。需要注意的是目前为止还无法判断对象的symbol属性的包含情况，如果属性是symbol，那么会直接报错

```
let person = {
    name : "xiejie",
    age : 18,
    walk : function(){
        console.log("I'm walking");
    }
}
let gender = Symbol("person's gender");
person[gender] = "male";
console.log("name" in person);//true
console.log("age" in person);//true
console.log([gender] in person);//报错
//TypeError: Cannot convert a Symbol value to a string
```

2. for..in

这个for..in我们在前面讲解遍历数组的时候已经见到过了。可以使用for..in来取出数组的键。除此之外，我们还可以使用for..in来循环遍历一个对象的所有属性，示例如下：

```
let person = {
    name : "xiejie",
    age : 18,
    walk : function(){
        console.log("I'm walking");
    }
}
for(let i in person)
{
    console.log(i);
}
// name
// age
// walk
```

需要注意的是，使用for..in虽然说可以遍历出一个对象的所有的属性和方法(包括继承的，关于继承后面会介绍)，但是无法遍历出用symbol来定义的属性，证明如下：

```
let person = {
    name : "xiejie",
    age : 18,
    walk : function(){
        console.log("I'm walking");
    }
}
```

```
let gender = Symbol("person's gender");
person[gender] = "male";
for(let i in person)
{
    console.log(i);
}
// name
// age
// walk
```

那么，这个时候可能有人会问了，那我如何才能遍历出一个对象的symbol属性呢？这里介绍两种方式。
第一种是使用`Object.getOwnPropertySymbols()`来返回一个对象所有的symbol属性，如下：

```
let person = {
    name : "xiejie",
    age : 18,
    walk : function(){
        console.log("I'm walking");
    }
}
let gender = Symbol("person's gender");
person[gender] = "male";
console.log(Object.getOwnPropertySymbols(person));
//[ Symbol(person's gender) ]
```

除了上面的方法以外，ES6中还提供了一个叫做`Reflect.ownkeys()`方法。该方法可以遍历出一个对象的所有类型的键名，包括字符串的键名以及symbol键名

```
let person = {
    name : "xiejie",
    age : 18,
    walk : function(){
        console.log("I'm walking");
    }
}
let gender = Symbol("person's gender");
person[gender] = "male";
console.log(Reflect.ownKeys(person));
//[ 'name', 'age', 'walk', Symbol(person's gender) ]
```

3. keys()，values()，entries()

前面在介绍遍历数组，集合以及映射的时候，有介绍过这3个方法，分别用于找出可迭代对象的键，值，以及键和值。实际上，我们的对象也是属于可迭代对象的一种，所以也可以使用这3个方法来找出对象的键和值

Object.key()

```
let person = {
    name : "xiejie",
    age : 18,
    walk : function(){
        console.log("I'm walking");
    }
}
let gender = Symbol("person's gender");
person[gender] = "male";
for(let i of Object.keys(person))
{
    console.log(i);
}
// name
// age
// walk
```

Object.values()

```
let person = {
    name : "xiejie",
    age : 18,
    walk : function(){
        console.log("I'm walking");
    }
}
let gender = Symbol("person's gender");
person[gender] = "male";
for(let i of Object.values(person))
{
    console.log(i);
}
// xiejie
// 18
// [Function: walk]
```

Object.entries()

```
let person = {
    name : "xiejie",
    age : 18,
    walk : function(){
        console.log("I'm walking");
    }
}
let gender = Symbol("person's gender");
person[gender] = "male";
for(let i of Object.entries(person))
{
    console.log(i);
}
// [ 'name', 'xiejie' ]
// [ 'age', 18 ]
// [ 'walk', [Function: walk] ]
```

6-1-6 嵌套对象

一个对象里面可以包含其他的对象，这个我们称之为对象的嵌套。示例如下：

```
let family = {
    xiejie : {
        age : 18,
        gender : "male"
    },
    song : {
        age : 20,
        gender : "female"
    }
};
```

当我们访问嵌套对象里面的值的时候，和访问单个对象的方式是一样的。

```
let family = {
    xiejie : {
        age : 18,
        gender : "male"
    },
    song : {
        age : 20,
        gender : "female"
    }
};
```

```
console.log(family.xiejie.gender);//male 点访问法
console.log(family["song"]["age"]);//20 中括号访问法
```

对象的解构

在前面介绍解构的时候，我们有介绍过数组的解构。实际上对象我们也是可以将其解构的。解构的方式也是和解构数组是类似的，示例如下：

```
let a = {name:"xiejie",age:18};
let b = {name:"song",age:20};
let {name:aName,age:aAge} = a;
let {name:bName,age:bAge} = b;
console.log(aName);//xiejie
console.log(aAge);//18
console.log(bName);//song
console.log(bAge);//20
```

当属性名和变量名一致的时候，可以进行简写，示例如下：

```
let a = {name:"xiejie",age:18};
let {name,age} = a;
console.log(name);//xiejie
console.log(age);//18
```

和数组一样，同样可以解构嵌套的对象，如下：

```
let family = {
    xiejie : {
        age : 18,
        gender : "male"
    },
    song : {
        age : 20,
        gender : "female"
    }
};
let {xiejie,song} = family;
console.log(xiejie);//{ age: 18, gender: 'male' }
console.log(song);//{ age: 20, gender: 'female' }
```

顺便一提的是，解构我们也是可以像函数一样设置一个默认值的，这里一起来看一下，如下：

```
let {name="xiejie",age} = {};
console.log(name);//xiejie
console.log(age);//undefined
```

这里我们为name变量设置了一个默认值为xiejie，当我们解构一个空对象的时候，name变量的值就使用了默认的xiejie这个值，而age这个变量由于没有设置默认值，所以值为undefined。

当然，既然叫做默认值，和函数一样，如果有解构的值传过来的话，肯定就是使用解构传过来的值，如下：

```
let {name="xiejie",age} = {name:"song",age:10};
console.log(name);//song
console.log(age);//10
```

6-1-7 对象作为函数参数

对象字面量也可以作为函数的参数进行传递。这在有很多形式参数的时候非常有用，因为它允许我们在调用函数时不用记住参数的顺序。
我们先来看一下我们一般函数调用的例子，如下：

```
let test = function(name,age){
    console.log(`我叫${name},我今年${age}岁`);
}
test("xiejie",18);//我叫xiejie,我今年18岁
test(18,"xiejie");//我叫18,我今年xiejie岁
```

可以看到，以前我们调用函数传递参数时，参数的顺序是必须要和形式参数的顺序一致的。否则就会出现上面第2次调用函数的情况，输出不符合预期的结果。

当我们使用对象字面量来最为形式参数时，就可以不用按照定义函数时形式参数的顺序，只要名字相同即可，如下：

```
let test = function({name,age}){
    console.log(`我叫${name},我今年${age}岁`);
}
test({name:"xiejie",age:18});//我叫xiejie,我今年18岁
test({age:18,name:"xiejie"});//我叫xiejie,我今年18岁
```

可以看到，只要参数名能对上号，位置是可以随意放置的。

和前面介绍函数的参数默认值一样，当对象字面量作为参数的时候，我们也可以为字面量的每一项设置一个默认值，甚至我们还可以给整个对象字面量一个默认值，示例如下：

```
//给整个对象字面量一个默认值{name:"Bill",age:20}
let test = function({name = "xiejie",age = 18} = {name:"Bill",age:20}){
    console.log(`我叫${name},我今年${age}岁`);
}
test();//我叫Bill,我今年20岁
test({});//我叫xiejie,我今年18岁
test({name:"yajing"});//我叫yajing,我今年18岁
test({age:1,name:"xizhi"});//我叫xizhi,我今年1岁
```

这种技术被称之为命名参数，经常被用在函数有很多可选参数的时候。

6-1-8 this关键字

弄清楚 `this` 的指向是非常有必要的。跟别的语言大相庭径的是，JavaScript的 `this` 总是指向一个对象，而具体指向哪个对象是在运行时基于函数的执行环境动态绑定的，而非函数被声明时的环境。除去不常用的 `with` 和 `eval` 的情况，具体到实际应用中， `this` 的指向大致可以分为以下2种：

- 作为对象的方法调用
- 作为普通函数调用

1. 作为对象的方法调用

当函数作为对象的方法被调用时， `this` 指向该对象，示例如下：

```
let person = {
    name : "xiejie",
    intro : function(){
        console.log(this === person); // true
        console.log(`My name is ${this.name}`);//this指向person对象
    }
}
person.intro();//My name is xiejie
```

在上面的例子中， `intro` 这个函数是作为 `person` 对象的方法被调用的，所以 `this` 指向 `person` 这个对象。

2. 作为普通函数调用

当函数不作为对象的属性被调用时，也就是我们常说的普通函数方式。此时的`this`总是指向全局对象。在浏览器的JavaScript中，这个全局对象是window对象。示例如下

```
let test = function(){
    console.log(this);//global对象
    //在node里面,全局对象就是global对象
    //如果是在浏览器里面,那么全局对象代表window对象
}
test(); //  普通函数调用，this将会指向全局对象
```

以普通函数的方式调用的时候，无论嵌套多少层函数，this都是指向全局对象

```
let test = function(){
    let test2 = function(){
        let test3 = function(){
            console.log(this);//global对象
        }
        test3();
    }
    test2();
}
test();
```

接下来我们再来看一个例子，如下：

```
var name = "PHP";
let obj = {
    name : "JavaScript"
}
let show = function(){
    console.log("Hello," + this.name);
}
obj.fn = show;//将show()方法赋值给obj对象的fn属性
obj.fn();//Hello,JavaScript
show();//Hello,undefined 如果是浏览器环境则是 Hello,PHP
```

这里我们将`show()`方法赋值给obj对象的fn属性，所以在执行`obj.fn()`时是以对象的形式来调用的，`this`将会指向obj对象。而后面的`show()`方法则是单纯的以普通函数的形式来进行调用的。所以`this`指向全局对象。

6-1-9 命名空间

当相同的变量和函数名被共享在同一作用域的时候，就会发生命名冲突。这看起来不太可能，但是我们可以想象一下，随着时间的推移，我们已经写了很多的代码，可能不知不觉就重用了一个变量名。如果是使用的其他开发者的代码库，这种问题就变得更加有可能。

解决命名冲突的方式，就是使用对象字面量来为一组相关函数创建一个命名空间。这样在调用这些函数的时候需要先写上对象名，这里的对象名就充当了命名空间的角色。示例如下：

```
let myMaths = {
    //求平方函数
    square : function(x){
        return x * x;
    },
    //传入数组求平均值函数
    avg : function(arr){
        let total = arr.reduce((a,b) => a + b);
        return total / arr.length;
    }
}
let arr = [1,2,3,4,5];
console.log(myMaths.avg(arr));//3
console.log(myMaths.square(5));//25
```

这里我们的`myMaths`就是我们的命名空间，这样就不用担心和其他人的变量或者函数名发生命名冲突。

6-2 JSON

JSON，英语全称为JavaScript Object Notation。是Douglas Crockford与2001年发明的一种轻量级数据存储格式，被很多服务用于数据序列化以及配置。JSON经常被用于Web服务之间交换信息，也被很多网站用来共享信息，例如twitter，facebook等。JSON最大的优点在于它在人、机可读性方面达到了一个最佳的临界点。

经常有人会把对象字面量和JSON相混淆，认为是同一个东西。但是实际上它们之间还是有几个关键性的区别：

- 属性名必须用双引号引起来
- 允许的值包括数字，true，false，null，数组，对象以及双引号引起来的字符串
- 函数是不允许的

例如：蝙蝠侠的JSON字符串可以表示为如下

```
let batman = {
    "name" : "Batman",
    "real name" : "Bruce Wayne",
    "height" : 74,
    "weight" : 210,
    "hero" : true,
    "villain" : false,
    "allies" : ["Robin","Batgirl","Superman"]
}
```

JSON正作为一种数据存储格式变得日益的流行，很多编程语言现在都有现成的库来解析和生成JSON。从ES5开始，就已经有了全局的JSON对象，该对象存在方法，可以将JavaScript中的字符串转为JSON或者将JSON转换为字符串。

JSON对象转为字符串

使用的方法为JSON.stringify()，示例如下：

```
let person = {
    "name" : "xiejie",
    "age" : 18,
    "gender" : "male",
}
let str = JSON.stringify(person);
console.log(str);
```

```
//{"name":"xiejie","age":18,"gender":"male"}
```

如果一个字面量对象里面包含了方法，那么在使用`JSON.stringify()`方法将其转为字符串时，会直接忽略掉对象里面的方法，如下：

```
let person = {
    name : "xiejie",
    age : 18,
    walk : function(){
        console.log("I'm walking");
    }
}
let str = JSON.stringify(person);
console.log(str);//{"name":"xiejie","age":18}
```

字符串转为JSON对象

使用的方法为`JSON.parse()`方法，但是需要注意的一个问题是，当我们要将一个字符串转为JSON对象时，必须要保证字符串的格式要和JSON的格式一模一样，否则无法进行转换，示例如下：

```
let person = '{"name":"xiejie","age":18,"gender":"male"}';
let obj = JSON.parse(person);
console.log(obj);
//{ name: 'xiejie', age: 18, gender: 'male' }
```

6-3 Math对象

在ECMAScript中，Math对象是一个比较特殊的对象，它是一个静态对象。什么意思呢？就是说这个Math和前面讲的其他类型不一样，不需要实例化，直接拿来用就可以了。

6-3-1 Math对象常见的属性

Math对象常见的属性如下表：

属性	说明
Math.E	自然对数的底数，即常量e的值
Math.LN10	10的自然对数
Math.LN2	2的自然对数
Math.LOG2E	以2为底e的对数
Math.LOG10E	以10为底e的对象
Math.PI	数学里面PI的值
Math.SQRT1_2	1/2的平方根(即2的平方根的倒数)
Math.SQRT2	2的平方根

这里面用得稍微多一点的就是PI，直接拿来用即可。

```
console.log(Math.PI);//3.141592653589793
```

6-3-2 Math对象常见的方法

1. min()和max()

这两个方法很简单，就是求一组数值的最大值和最小值

```
let max = Math.max(3,5,8,1);
let min = Math.min(3,5,8,1);
console.log(max);//8
console.log(min);//1
```

2. 舍入方法ceil()，floor()和round()

ceil()：执行向上舍入
floor()：执行向下舍入
round()：四舍五入

```
let num = 3.14;
console.log(Math.ceil(num));//4
console.log(Math.floor(num));//3
console.log(Math.round(num));//3
```

3. 随机数方法

Math.random()方法返回0-1之间的随机数，如果想显示固定范围的随机数，可以套用下面的公式。
值 = Math.floor(Math.random()*可能值的总数+第一个可能的值)

```
let num = Math.random();
console.log(num);//0.24003779065523112
//生成25-50之间的随机数
//可能值的计算：50-25+1
let rand = Math.floor(Math.random()*26 + 25);
console.log(rand);//41
```

练习：封装一个函数，这个函数接收两个参数，然后可以返回这两个参数之间的随机数

```
let rand = function(x,y){
    let choice = y - x + 1;
    return Math.floor(Math.random() * choice + x);
}
console.log(rand(1,10));//3
```

4. 其他方法

在Math对象里面还有诸如下表所示的其他方法，这里不再做一一演示

方法	说明
Math.abs(num)	返回num的绝对值

Math.exp(num)	返回Math.E的num次幂
Math.log(num)	返回num的自然对数
Math.pow(num,power)	返回num的power次幂
Math.sqrt(num)	返回num的平方根
Math.acos(x)	返回x的反余弦值
Math.asin(x)	返回x的反正弦值
Math.atan(x)	返回x的反正切值
Math.atan2(x)	返回y/x的反正切值
Math.cos(x)	返回x的余弦值
Math.sin(x)	返回x的正弦值
Math.tan(x)	返回x的正切值

6-4 Date对象

Date类型主要是用于处理和时间相关的操作。

6-4-1 时间戳

在学习Date类型之前，有一个概念必须要了解，那就是时间戳。
所谓时间戳，就是从1970年1月1日0时0分0秒到现在为止的秒数。在计算机里面，进行时间的计算都是通过时间戳来进行计算的。计算完成以后再将时间戳转换为表示时间的字符串。

获取时间戳

在ECMAScript中比较特殊的是获取时间的精度更高一些，可以获取到的时间戳精确到了毫秒，通过以下方式可以获取到

```
let now = Date.now();
console.log(now);//1511767644238
```

如果想要得到秒数，可以使用得到的毫秒数除以1000，然后四舍五入，如下：

```
let now = Date.now();
now = Math.round(now / 1000);//毫秒除以1000,得到描述
console.log(now);//1511767768
```

6-4-2 静态方法

可以看到，上面的`Date.now()`就是一个静态方法，除了这个方法以外，这里还要介绍两个静态方法，分别是`Date.parse()`和`Date.UTC()`

Date.parse()

该方法用于解析一个日期字符串，参数是一个包含待解析的日期和时间的字符串，返回从1970年1月1日0点到给定日期的毫秒数
该方法会根据日期时间字符串格式规则来解析字符串的格式，除了标准格式外，以下格式也支持。如果字符串无法识别，将返回NaN
1、'月/日/年' 如6/13/2004
2、'月 日,年' 如January 12,2004或Jan 12,2004
3、'星期 月 日 年 时:分:秒 时区' Tue May 25 2004 00:00:00 GMT-0700

注意：浏览器不支持不表示日期只表示时间的字符串格式

```
console.log(Date.parse("1990/03/23"));//638121600000
console.log(Date.parse("March 23,1990"));//638121600000
console.log(Date.parse("2017"));//1483228800000
console.log(Date.parse("Hello"));//NaN
```

> 注意：在ECMAScript5中，如果使用标准的日期时间字符串格式规则的字符串中，数学前有前置0，则会解析为UTC时间，时间没有前置0，则会解析为本地时间。其他情况一般都会解析为本地时间

Date.UTC()

Date.UTC()同样返回给定日期的毫秒数，但其参数并不是一个字符串，而是分别代表年、月、日、时、分、秒、毫秒的数字参数，说白了就是参数的形式和上面不一样。Date.UTC()方法的语法如下：
Date.UTC(year,month,day,hours,minutes,seconds,ms)，其中year参数是固定的，其余参数都是可选的，我们可以通过函数的length属性来查看该函数的形式参数个数

```
console.log(Date.UTC.length);//7
```

> 注意：该方法使用的是UTC时间，而不是本地时间

```
console.log(Date.UTC("1990/03/23"));//NaN
console.log(Date.UTC(1990,3,23));//640828800000
console.log(Date.UTC(2017));//1483228800000
console.log(Date.UTC("Hello"));//NaN
```

6-4-3 日期对象构造函数

日期对象的构造函数为Date()。该构造函数根据使用的不同效果也不尽相同。

不使用new关键字

如果不使用new关键字，那么就只是单纯的函数调用。会返回一个当前的日期和时间的字符串表示。并且被当作函数调用时，会忽略所有传递进去的参数，如下：

```
console.log(Date());
//Mon Nov 27 2017 16:03:33 GMT+0800 (CST)
console.log(Date("1990-03-23"));
```

```
//Mon Nov 27 2017 16:03:33 GMT+0800 (CST)
```

使用new关键字

如果使用new关键字，那么这个时候就会返回一个对象。关于这种使用new关键字创建对象的方式，我们会在后面进行详细的介绍，这是只是作为了解即可。

使用new关键字但是没有传入任何参数，则会根据当前的日期时间来创建一个date对象

```
let date = new Date();
console.log(date);//2017-11-27T08:05:44.025Z
console.log(typeof date);//object
```

如果传入数字参数，则该参数表示与1970年1月1日0时0分0秒之间的毫秒数，如下：

```
let date = new Date(638121600000);
console.log(date);//1990-03-22T16:00:00.000Z
```

可以接收多个数字参数，这个时候形式有点类似于Date.UTC()这个方法，不过返回的是一个对象，而不是毫秒数。

```
let date = new Date(1990,3,23);
console.log(date);//1990-04-22T15:00:00.000Z
```

如果传入的是字符串参数，则返回该日期对象。如果字符串不能被解析为日期，则返回Invalid Date

```
let date = new Date("1990-03-23");
console.log(date);//1990-03-23T00:00:00.000Z
let date2 = new Date("Hello");
console.log(date2);//Invalid Date
```

6-4-5 实例方法

Date对象没有可以直接读写的属性，所有对日期和时间的访问都需要通过方法。
Date对象的大多数方法分为两种形式：一种是使用本地时间，一种是使用UTC时间，这些方法在下面一起列出。例如，`get[UTC]Day()`同时代表`getDay()`和`getUTCDay()`
Date对象一共有46个实例方法，可以分为以下3类：`to类`、`get类`和`set类`。因为Date对象的实例方法个数太多，而大多数实例方法在使用的时候都是非常相似的，所以我们这里只选择个别

方法进行演示

1. to类

to类方法从Date对象返回一个字符串，表示指定的时间

- toString()：返回本地时区的日期字符串
- toUTCString()：返回UTC时间的日期字符串
- toISOString()：返回Date对象的标准的日期时间字符串格式的字符串
- toDateString()：返回Date对象的日期部分的字符串
- toTimeString()：返回Date对象的时间部分的字符串
- toJSON()：返回一个符合JSON格式的日期字符串，与toISOString方法的返回结果完全相同
- toLocaleString()：toString()方法的本地化转换
- toLocaleTimeString()：toTimeString()方法的本地化转换
- toLocaleDateString()：toDateString()方法的本地化转换

个别方法演示：

```
console.log(new Date("1990-03-23").toString());
//Fri Mar 23 1990 08:00:00 GMT+0800 (CST)
console.log(new Date("1990-03-23").toDateString());//Fri Mar 23 1990
console.log(new Date("1990-03-23").toTimeString());//08:00:00 GMT+0800 (CST)
console.log(new Date("1990-03-23").toLocaleString());//1990-3-23 08:00:00
```

2. get类

Date对象提供了一系列get类方法，用来获取实例对象某个方面的值
在介绍get类方法之前，首先要介绍valueOf()方法
valueOf()：返回距离1970年1月1日0点的毫秒数。因此，可以方便地使用比较运算符来比较日期值

```
let date1 = new Date(1990,3,23);
let date2 = new Date(1988,8,21);
console.log(date1 > date2);//true
```

- getTime()：返回距离1970年1月1日0点的毫秒数，同valueOf()。在ECMAScript5之前，可以使用getTime()方法实现Date.now()
- getTimezoneOffset()：返回当前时间与UTC的时区差异，以分钟表示(8*60=480分钟)，返回结果考虑到了夏令时因素
- getYear()：返回距离1900年的年数(已过时)

- get[UTC]FullYear()：返回年份(4位数)
- get[UTC]Month()：返回月份(0-11)
- get[UTC]Date()：返回第几天(1-31)
- get[UTC]Day()：返回星期几(0-6)
- get[UTC]Hours()：返回小时值(0-23)
- get[UTC]Minutes()：返回分钟值(0-59)
- get[UTC]Seconds()：返回秒值(0-59)
- get[UTC]Milliseconds()：返回毫秒值(0-999) >注意：通过标准日期时间格式字符串，且有前置0的形式的参数设置，设置的是UTC时间

个别方法演示：

```
console.log(new Date("1990-03-23").valueOf());//638150400000
console.log(new Date("1990-03-23").getTime());//638150400000
console.log(new Date("1990-03-23").getDay());//5
console.log(new Date("1990-03-23").getMonth());//2
```

3. set类

Date对象提供了一系列set类方法，用来设置实例对象的各个方面
set方法基本与get方法相对应，set方法传入类似于Date.UTC()的参数，返回调整后的日期的内部毫秒数

> 注意：星期只能获取，不能设置

- setTime()：使用毫秒的格式，设置一个Date对象的值
- setYear()：设置年份(已过时)
- set[UTC]FullYear()：设置年份(4位数)，以及可选的月份值和日期值
- set[UTC]Month()：设置月份(0-11)，以及可选的日期值
- set[UTC]Date()：设置第几天(1-31)
- set[UTC]Hours()：设置小时值(0-23)，以及可选的分钟值、秒值及毫秒值
- set[UTC]Minutes()：设置分钟值(0-59)，以及可选的秒值及毫秒值
- set[UTC]Seconds()：设置秒值(0-59)，以及可选的毫秒值
- set[UTC]Milliseconds()：设置毫秒值(0-999)

个别方法演示：

```
let date = new Date("1990-03-23");
console.log(date.setFullYear(1992),date.getFullYear());
//701308800000 1992
console.log(date.setMonth(4),date.getMonth());
```

```
//706579200000 4
```

6-5 正则表达式

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

6-5-1 正则表达式基本介绍

JavaScript中的正则表达式用RegExp对象表示，有两种写法：一种是字面量写法；另一种是构造函数写法

字面量写法

正则表达式字面量写法，又叫Perl写法，因为JavaScript的正则表达式特性借鉴自Perl。正则表达式字面量定义为包含在一对斜杠 `/` 之间的字符，并且可以有3个模式修正符

```
let expression = /pattern/flags;
```

这里的 `pattern` 就是指我们的字符串模式，而后面的 `flags` 则是指带的模式修正符。
JavaScript中的正则表达式支持下列3个模式修正符：
g：表示全局(global)模式，即模式将被应用于所有字符串，而并非在发现第一个匹配项时立即停止
i：表示不区分大小写(case-insensitive)模式，即在确定匹配项时忽略模式与字符串的大小写
m：表示多行(multiline)模式，即在到达一行文本末尾时还会继续查找下一行中是否存在与模式匹配的项
后面我们会对这些模式修正符做详细的介绍。

RegExp构造函数

和普通的内置对象一样，RegExp正则表达式对象也支持new RegExp()构造函数的形式来创建正则。RegExp构造函数接收两个参数：要匹配的字符串模式(pattern)和可选的模式修正符(flags)。

```
let reg1 = /at/i;
//等同于
let reg2 = new RegExp("at","i");
```

需要注意无论是字面量，还是构造函数，返回的类型都为object

```
let reg1 = /at/i;
//等同于
let reg2 = new RegExp("at","i");
console.log(typeof reg1);//object
console.log(typeof reg2);//object
```

一般来讲，如果是字面量构建的正则表达式，那么大致可以分为4个部分：定界符，简单字符，元字符和模式修正符。而如果是构造函数构建的正则表达式，则少了一个定界符，由简单字符，元字符以及模式修正符组成。

简单字符在正则表达式中，就是字面的含义，比如/a/匹配a，/b/匹配b，这样的字符被称之为简单字符，示例如下：

```
let reg = /dog/;
let str = "This is a big dog";
console.log(reg.test(str));//true
```

这里我们调用了正则表达式的实例方法`test()`，该方法如果和传入的字符串能够匹配，则返回true，否则返回false。

除了简单字符以外，还有一些字符，它们除了字面意思外，还有着特殊的含义，这些字符就是元字符。
JavaScript的正则中的元字符如下：

元字符	名称	匹配对象
`.`	点号	单个任意字符(除回车\r、换行\n、行分隔符\u2028和段分隔符\u2029外)
`[]`	字符组	列出的单个任意字符
`[^]`	排除型字符组	未列出的单个任意字符
`?`	问号	匹配0次或1次
`*`	星号	匹配0交或多次
`+`	加号	匹配1次或多次
`{a,b}`	区间量词	匹配至少a次，最多b次
`^`	脱字符	行的起始位置

`$`	美元符	行的结束位置
`|`	竖线	分隔两边的任意一个表达式
`()`	括号	限制多选结构的范围，标注量词作用的元素，为反向引用捕获文本
`\1,\2...`	反向引用	匹配之前的第一、第二...组括号内的表达式匹配的文本

在后面我们会慢慢的来介绍这些元字符。

转义字符

转义字符(escape)表示为反斜线`\`+字符的形式，共有以下3种情况
1.因为元字符有特殊的含义，所以无法直接匹配。如果要匹配它们本身，则需要在它们前面加上反斜杠`\`

```
console.log(/1+1/.test("1+1"));//false
console.log(/1\+1/.test("1+1"));//true
```

但实际上，并非14个元字符都需要转义，右方括号]和右花括号}可以不用写转义字符，当然写了也不会算你错

```
console.log(/]/.test("]"));//true
console.log(/\]/.test("]"));//true
```

2.`\`加非元字符，表示一些不能打印的特殊字符，具体如下表：

符号	含义
\0	NUL字符\u0000
[\b]	匹配退格符\u0008，不要与\b混淆
\t	制表符\u0009
\n	换行符\u000A
\v	垂直制表符\u000B
\f	换页符\u000C
\r	回车符\u000D

\xnn	由十六进制数nn指定的拉丁字符
\uxxxx	由十六进制数xxxx指定的Unicode字符(\u4e00-\u9fa5代表中文)
\cX	控制字符X，表示ctrl-[X]，其中的X是A-Z之中任一个英文字母，用来匹配控制字符

3. `\` 加任意其他字符，默认情况就是匹配此字符，也就是说，反斜线 `\` 被忽略了

```
console.log(/\x/.test("x"));//true
console.log(/\yat/.test("yat"));//true
```

6-5-2 字符组相关元字符

从这一小节开始我们来介绍一下正则表达式中的元字符。元字符可以分为好几个大类，例如字符组，量词等内容。其实可以这么说，学习正则表达式，主要就是学习元字符。所以，接下来我们就来一起认识一下这些在正则表达式中拥有特殊含义的符号。

1. 字符组介绍

字符组(Character Class)，有的翻译成字符类或字符集。简单而言，就是指用方括号表示的一组字符，它匹配若干字符之一

```
//匹配0-9这10个数字之一
let reg = /[0123456789]/;
console.log(reg.test("1"));//true
console.log(reg.test("a"));//false
```

注意：字符组中的字符排列顺序并不影响字符组的功能，出现重复字符也不会影响

```
[0123456789]
//等价于
[9876543210]
//等价于
[12345678901234567 89]
```

2. 范围

正则表达式通过连字符 `-` 提供了范围表示法，可以简化字符组

```
[0123456789]
```

```
//等价于
[0-9]
[abcdefghijklmnopqrstuvwxyz]
//等价于
[a-z]
```

连字符`-`表示的范围是根据ASCII编码的码值来确定的，码值小的在前，码值大的在后
所以[0-9]是合法的，而[9-0]会报错

```
//匹配0-9这10个数字之一
let reg1 = /[0-9]/;
console.log(reg1.test("1"));//true
let reg2 = /[9-0]/;//报错
console.log(reg2.test("1"));
//SyntaxError: Invalid regular expression/[9-0]/:
//Range out of order in character class
```

在字符组中可以同时并列多个`-`范围

```
//[0-9a-zA-Z];匹配数字、大写字母和小写字母
//[0-9a-fA-F];匹配数字，大、小写形式的a-f，用来验证十六进制字符
console.log(/[0-9a-fA-F]/.test('d'));//true
console.log(/[0-9a-fA-F]/.test('x'));//false
```

只有在字符组内部，连字符`-`才是元字符，表示一个范围，否则它就只能匹配普通的连字符号。
注意：如果连字符出现在字符组的开头或末尾，它表示的也是普通的连字符号，而不是一个范围

```
//匹配中划线
console.log(/-/.test('-'));//true
console.log(/[-]/.test('-'));//true
//匹配0-9的数字或中划线
console.log(/[0-9]/.test('-'));//false
console.log(/[0-9-]/.test('-'));//true
console.log(/[0-9\-]/.test('-'));//true
console.log(/[-0-9]/.test('-'));//true
console.log(/[\-0-9]/.test('-'));//true
```

3. 排除

字符组的另一个类型是排除型字符组，在左方括号后紧跟一个脱字符`^`表示，表示在当前位置匹配一个没有列出的字符
所以`[^0-9]`表示0-9以外的字符

```
//匹配第一个是数字字符，第二个不是数字字符的字符串
console.log(/[0-9][^0-9]/.test('1e'));//true
console.log(/[0-9][^0-9]/.test('q2'));//false
```

注意：在字符组内部，脱字符 `^` 表示排除，而在字符组外部，脱字符 `^` 表示一个行锚点
`^` 符号是元字符，在字符组中只要 `^` 符号不挨着左方括号就可以表示其本身含义，不转义也可以

```
//匹配abc和^符号
console.log(/[a-c^]/.test('^'));//true
console.log(/[a-c\^]/.test('^'));//true
console.log(/[\^a-c]/.test('^'));//true
```

在字符组中，只有^ - [ ] 这4个字符可能被当做元字符，其他有元字符功能的字符都只表示其本身，示例如下：在[ ]中的* + ？等元字符都是被当作其本身

```
//部分元字符示例
//这里会被当作是字面意思
console.log(/[1$]/.test('$'));//true
console.log(/[1|2]/.test('|'));//true
console.log(/[1?]/.test('?'));//true
console.log(/[1*]/.test('*'));//true
console.log(/[1+]/.test('+'));//true
console.log(/[1.]/.test('.'));//true
```

4. 简记

用[0-9]、[a-z]等字符组，可以很方便地表示数字字符和小写字母字符。对于这类常用字符组，正则表达式提供了更简单的记法，这就是字符组简记(shorthands)
常见的字符组简记有 `\d` 、 `\w` 、 `\s` 。其中d表示(digit)数字，w表示(word)单词，s表示(space)空白
正则表达式也提供了对应排除型字符组的简记法： `\D` 、 `\W` 、 `\S` 。字母完全相同，只是改为大写。它们和小写的简记符在含义上刚好相反。

```
\d      数字，等同于[0-9]
\D      非数字，等同于[^0-9]
\s      空白字符，等同于[\f\n\r\t\u000B\u0020\u00A0\u2028\u2029]
\S      非空白字符，等同于[^\f\n\r\t\u000B\u0020\u00A0\u2028\u2029]
\w      字母、数字、下划线，等同于[0-9A-Za-z_](汉字不属于\w)
\W      非字母、数字、下划线，等同于[^0-9A-Za-z_]
```

> 注意：\w不仅包括字母、数字，还包括下划线。在进行数字验证时，只允许输入字母和数字时，不可以使用\w，因为还包含了下划线。所以应该使用[0-9a-zA-Z]

5. 任意字符

经常有人有一个误区就是认为点可以代表任意字符，其实并不是。点号`.`代表除回车(\r)，换行(\n)，行分隔符(\u2028)和段分隔符(\u2029)以外的任意字符。
妥善的利用互补属性，可以得到一些巧妙的效果。比如，[\s\S]、[\w\W]、[\d\D]都可以表示任意字符

```
//匹配任意字符
console.log(/./.test('\r'));//false
console.log(/[\s\S]/.test('\r'));//true
console.log(/[\d\D]/.test('\r'));//true
```

6-5-3 量词相关元字符

根据字符组的介绍，可以用字符组[0-9]或\d来匹配单个数字字符，如果用正则表达式表示更复杂的字符串，则不太方便

```
//表示邮政编码6位数字
/[0-9][0-9][0-9][0-9][0-9][0-9]/;
//等价于
/\d\d\d\d\d\d/;
```

正则表达式提供了量词，用来设定某个模式出现的次数

```
{n}       匹配n次
{n,m}     匹配至少n次，最多m次
{n,}      匹配至少n次
?         相当于{0,1}
*         相当于{0,}
+         相当于{1,}
```

美国英语和英国英语有些词的写法不一样，如果traveler和traveller，favor和favour，color和colour

```
//同时匹配美国英语和英国英语单词
/travell?er/;
/favou?r/;
```

```
/colou?r/;
```

协议名有http和https两种

```
/https?/;
```

贪婪模式

默认情况下，量词都是贪婪模式(greedy quantifier)，即匹配到下一个字符不满足匹配规则为止

```
let reg = /a+/;
let str = "aaabbcc";
console.log(reg.exec(str));
//[ 'aaa', index: 0, input: 'aaabbcc' ]
```

这里我们使用了正则表达式的另外一个常用的实例方法`exec()`。该方法会返回一个数组，数组里面有匹配上的字符，匹配上的索引值以及原始的字符串等更加详细的信息。

懒惰模式

懒惰模式(lazy quantifier)和贪婪模式相对应，在量词后加问号`?`表示，表示尽可能少的匹配，一旦条件满足就再不往下匹配

```
{n}?        匹配n次
{n,m}?      匹配至少n次，最多m次
{n,}?       匹配至少n次
??          相当于{0,1}
*?          相当于{0,}
+?          相当于{1,}
```

示例如下：

```
let reg = /a+?/;
let str = "aaabbcc";
console.log(reg.exec(str));
//[ 'a', index: 0, input: 'aaabbcc' ]
```

6-5-4 括号相关元字符

括号有两个功能，分别是分组和引用。具体而言，用于限定量词或选择项的作用范围，也可以用于捕获文本并进行引用或反向引用

分组

量词控制之前元素的出现次数，而这个元素可能是一个字符，也可能是一个字符组，也可以是一个表达式。如果把一个表达式用括号包围起来，这个元素就是括号里的表达式，被称为子表达式
如果希望字符串"ab"重复出现2次，应该写为(ab){2}，而如果写为ab{2}，则{2}只限定b，如下：

```
console.log(/ab{2}/.test("abab"));//false
console.log(/(ab){2}/.test("abab"));//true
```

身份证长度有15位和18位两种，如果只匹配长度，可能会想当然地写成\d{15,18}，实际上这是错误的，因为它包括15、16、17、18这四种长度。因此，正确的写法应该是

```
/\d{15}(\d{3})?/
```

捕获

括号不仅可以对元素进行分组，还会保存每个分组匹配的文本，等到匹配完成后，引用捕获的内容。因为捕获了文本，这种功能叫捕获分组
比如，要匹配诸如2016-06-23这样的日期字符串

```
/(\d{4})-(\d{2})-(\d{2})/
```

与以往不同的是，年、月、日这三个数值被括号括起来了，从左到右为第1个括号、第2个括号和第3个括号，分别代表第1、2、3个捕获组。ES有9个用于存储捕获组的构造函数属性，如果使用的是`test()`方法，那么通过正则对象的`$1-$9`属性可以访问到

```
//RegExp.$1\RegExp.$2\RegExp.$3……到RegExp.$9
//分别用于存储第一、第二……第九个匹配的捕获组。
//在调用exec()或test()方法时，这些属性会被自动填充
console.log(/(\d{4})-(\d{2})-(\d{2})/.test('2016-06-23'));//true
console.log(RegExp.$1);//2016
console.log(RegExp.$2);//06
console.log(RegExp.$3);//23
console.log(RegExp.$4);//""
```

再例如：

```
let reg = /(a+)(b*)xj/;
let str = "aabbbxj";
console.log(reg.test(str));//true
console.log("$1的值:",RegExp.$1);//$1的值: aa
console.log("$2的值:",RegExp.$2);//$2的值: bbb
console.log("$3的值:",RegExp.$3);//$3的值:
```

而`exec()`方法是专门为捕获组而设计的，返回的数组中，第一项是与整个模式匹配的字符串，其他项是与模式中的捕获组匹配的字符串，如果要获取，那么可以通过指定数组的下标来进行获取，如下：

```
let reg = /(\d{4})-(\d{2})-(\d{2})/;
let str = "2017-03-21";
let i = reg.exec(str);
console.log(i);
//[ '2017-03-21', '2017', '03', '21', index: 0, input: '2017-03-21' ]
console.log(i[1]);//2017
console.log(i[2]);//03
console.log(i[3]);//21
```

捕获分组捕获的文本，不仅可以用于数据提取，也可以用于替换
字符串的`replace()`方法就是用于进行数据替换的，该方法接收两个参数，第一个参数为待查找的内容，而第二个参数为替换的内容

```
let str = "2017-12-12";
console.log(str.replace(/-/g,"."));
//2017.12.12
```

注意这里书写正则表达式的时候必须要写上模式修正符`g`，这样才能够进行全局匹配。
在`replace()`方法中也可以引用分组，形式还是用$num，其中num是对应分组的编号

```
//把2017-03-23的形式变成03-23-2017
let reg = /(\d{4})-(\d{2})-(\d{2})/;
let str = "2017-03-23";
console.log(str.replace(reg,"$2-$3-$1"));
//03-23-2017
```

反向引用

英文中不少单词都有重叠出现的字母，如shoot或beep。若想检查某个单词是否包含重叠出现的字母，则需要引入反向引用(back-reference)

反向引用允许在正则表达式内部引用之前捕获分组匹配的文本，形式是\num，num表示所引用分组的编号

```
//重复字母
let reg = /([a-z])\1/;
console.log(reg.test('aa'));//true
console.log(reg.test('ab'));//false
```

接下来我们再来看一个反向引用的例子

```
let reg = /(ab)(cd)\1xj/;
let str = "abcdabxj";
console.log(reg.test(str));//true
```

如果我们要跳过某一个子表达式，那么可以使用`?:`来跳过(后面会提到的非捕获)

```
let reg = /(?:ab)(cd)\1xj/;
let str = "abcdabxj";
let str2 = "abcdcdxj";
console.log(reg.test(str));//false
console.log(reg.test(str2));//true
```

反向引用可以用于建立前后联系。例如HTML标签的开始标签和结束标签是对应着的，这个时候我们就可以使用反向引用

```
//开始标签
// <([^>]+)>
//标签内容
// [\s\S]*?
//匹配成对的标签
///<([^>]+)>[\s\S]*?<\/\1>/
console.log(/<([^>]+)>[\s\S]*?<\/\1>/.test('<a>123</a>'));//true
console.log(/<([^>]+)>[\s\S]*?<\/\1>/.test('<a>123</b>'));//false
```

非捕获

除了捕获分组，正则表达式还提供了非捕获分组(non-capturing group)，以`?:`的形式表示，它只用于限定作用范围，而不捕获任何文本
比如，要匹配abcabc这个字符，一般地，可以写为(abc){2}，但由于并不需要捕获文本，只是限定了量词的作用范围，所以应该写为(?:abc){2}

```
console.log(/(abc){2}/.test('abcabc'));//true
console.log(/(?:abc){2}/.test('abcabc'));//true
```

由于非捕获分组不捕获文本，对应地，也就没有捕获组编号

```
console.log(/(abc){2}/.test('abcabc'));//true
console.log(RegExp.$1);//abc
console.log(/(?:abc){2}/.test('abcabc'));//true
console.log(RegExp.$1);//""
```

非捕获分组也不可以使用反向引用

```
console.log(/(?:123)\1/.test('123123'));//false
console.log(/(123)\1/.test('123123'));//true
```

捕获分组和非捕获分组可以在一个正则表达式中同时出现

```
console.log(/(\d)(\d)(?:\d)(\d)(\d)/.exec('12345'));
//[ '12345', '1', '2', '4', '5', index: 0, input: '12345' ]
```

6-5-5 选择相关元字符

竖线 `|` 在正则表达式中表示或(OR)关系的选择，以竖线 `|` 分隔开的多个子表达式也叫选择分支或选择项。在一个选择结构中，选择分支的数目没有限制。
在选择结构中，竖线 `|` 用来分隔选择项，而括号 `()` 用来规定整个选择结构的范围。如果没有出现括号，则将整个表达式视为一个选择结构。
选择项的尝试匹配次序是从左到右，直到发现了匹配项，如果某个选择项匹配就忽略右侧其他选择项，如果所有子选择项都不匹配，则整个选择结构匹配失败。

```
console.log(/12|23|34/.exec('1'));//null
console.log(/12|23|34/.exec('12'));//[ '12', index: 0, input: '12' ]
console.log(/12|23|34/.exec('23'));//[ '23', index: 0, input: '23' ]
console.log(/12|23|34/.exec('2334'));//[ '23', index: 0, input: '2334' ]
```

在选择结构中，应该尽量避免选择分支中存在重复匹配，因为这样会大大增加回溯的计算量

```
//不良的选择结构
a|[ab]
```

```
[0-9]|\w
```

6-5-6 断言相关元字符

在正则表达式中，有些结构并不真正匹配文本，而只负责判断在某个位置左/右侧是否符合要求，这种结构被称为断言(assertion)，也称为锚点(anchor)，常见的断言有3种：单词边界、行开头结尾、环视

单词边界

在文本处理中可能会经常进行单词替换，比如把row替换成line。但是，如果直接替换，不仅所有单词row都被替换成line，单词内部的row也会被替换成line。要想解决这个问题，必须有办法确定单词row，而不是字符串row
为了解决这类问题，正则表达式提供了专用的单词边界(word boundary)，记为`\b`，它匹配的是'单词边界'位置，而不是字符。`\b`匹配的是一边是单词字符`\w`，一边是非单词字符`\W`的位置
与`\b`对应的还有`\B`，表示非单词边界，但实际上`\B`很少使用

```
let reg = /\bis\b/;
let str = "this is a test";
console.log(reg.exec(str));
//[ 'is', index: 5, input: 'this is a test' ]
console.log(reg.exec("is"));
//[ 'is', index: 0, input: 'is' ]
```

起始结束

常见的断言还有`^`和`$`，它们分别匹配字符串的开始位置和结束位置，所以可以用来判断整个字符串能否由表达式匹配

```
let reg = /^\d\w*/;
let str1 = "1asd";
let str2 = "qwe2";
console.log(reg.test(str1));//true
console.log(reg.test(str2));//false
```

`^`和`$`的常用功能是删除字符串首尾多余的空白，类似于字符串String对象的`trim()`方法

```
let fnTrim = function(str){
    return str.replace(/^\s+|\s+$/,'')
```

```
}
console.log(fnTrim('      hello world   '));//'hello world'
```

环视

环视(look-around)，可形象地解释为停在原地，四处张望。环视类似于单词边界，在它旁边的文本需要满足某种条件，而且本身不匹配任何字符
环视分为正序环视和逆序环视，而JavaScript只支持正序环视，相当于只支持向前看，不支持往回看。而正序环视又分为肯定正序环视和否定正序环视。
肯定正序环视的记法是`?=n`，表示后面必须是n才匹配；否定正序环视的记忆法是`?!n`，表示后面必须不是n才匹配

示例：
一般来讲，`?=`是写在正则的最后的，例如我要匹配文件名以.js结束的文件

```
let reg = /\w+(?=\.js)/;
let str = "test.js";
console.log(reg.test(str));//true
```

但是，如果在`?=`后面想要继续书写字符，那么必须先写一个`?=`后面的字符，如下

```
let reg = /a(?=b)bz/;//a后面必须是b
let str = "abzc";
console.log(reg.test(str));//true
```

注意括号后面的地方第一个字符必须写成b，因为`?=`后面就写的b，这样才能够继续在后面书写字符
如果像下面这样写，那么是不可能匹配上的

```
let reg = /a(?=b)c/;
let str = "abc";
let str2 = "acc";
let str3 = "abb";
let str4 = "abbc";
console.log(reg.test(str));//false
console.log(reg.test(str2));//false
console.log(reg.test(str3));//false
console.log(reg.test(str4));//false
```

接下来我们来看`?!`，其实就和`?=`刚好相反，后面的字符不能是某一个字符，如下：

```
let reg = /a(?!b)c/;
let str = "ac";
console.log(reg.test(str));//true
```

同样需要注意的是，既然指定了后面不能是某一个字符，那么如果想要往后面继续书写字符，首先需要写一个不是`?!`后面的，如下：

```
let reg = /a(?!b)b/;
let str1 = "axb";
let str2 = "abb";
let str3 = "acb";
console.log(reg.test(str1));//false
console.log(reg.test(str2));//false
console.log(reg.test(str3));//false
```

注意：环视虽然也用到括号，却与捕获型分组编号无关；但如果环视结构出现捕获型括号，则会影响分组

```
console.log(/ab(?=cd)/.exec("abcd"));
//[ 'ab', index: 0, input: 'abcd' ]
console.log(/ab(?=(cd))/.exec("abcd"));
//[ 'ab', 'cd', index: 0, input: 'abcd' ]
```

6-5-7 模式修正符

匹配模式(match mode)又被称之为模式修正符。指的是匹配时使用的规则。设置特定的模式，可能会改变对正则表达式的识别。前面已经介绍过创建正则表达式对象时，可以设置`m`、`i`、`g`这三个标志，分别对应多行模式、不区分大小模式和全局模式三种

i

默认地，正则表达式是区分大小写的，通过设置标志'i'，可以忽略大小写(ignore case)

```
console.log(/ab/.test("aB"));//false
console.log(/ab/i.test("aB"));//true
```

m

默认地，正则表达式中的`^`和`$`匹配的是整个字符串的起始位置和结束位置，而通过设置标志'm'，开启多行模式，它们也能匹配字符串内部某一行文本的起始位置和结束位置

```
console.log(/^b/.test('a\nb')); //false
console.log(/^b/m.test('a\nb')); //true
```

g

默认地，第一次匹配成功后，正则对象就停止向下匹配了。g修饰符表示全局匹配(global)，设置'g'标志后，正则对象将匹配全部符合条件的结果，主要用于搜索和替换

```
console.log('1a,2a,3a'.replace(/a/,'b'));//1b,2a,3a
console.log('1a,2a,3a'.replace(/a/g,'b'));//1b,2b,3b
```

6-5-8 优先级

正则表达式千变万化，但是大多都是由之前介绍过的字符组、括号、量词等基本结构组合而成的。这些元字符，和运算符一样拥有一个优先级关系，如下：

```
//从上到下，优先级逐渐降低
\                           转义符
() (?!) (?=) []             括号、字符组、环视
* + ? {n} {n,} {n,m}        量词
^ $                         起始结束位置
|                           选择
```

由于括号的用途之一就是为量词限定作用范围，所以优先级比量词高

```
console.log(/ab{2}/.test('abab'));//false
console.log(/(ab){2}/.test('abab'));//true
```

> 注意：选择符`|`的优先级最低，比起始和结束位置都要低

```
console.log(/^ab|cd$/.test('abc'));//true
console.log(/^(ab|cd)$/.test('abc'));//false
console.log(/^(ab|cd)$/.test('ab'));//true
console.log(/^(ab|cd)$/.test('cd'));//true
```

6-5-9 局限性

尽管JavaScript中的正则表达式功能比较完备，但与其他语言相比，缺少某些特性
下面列出了JavaScript正则表达式不支持的特性

- POSIX字符组(只支持普通字符组和排除型字符组)
- Unicode支持(只支持单个Unicode字符)
- 匹配字符串开始和结尾的\A和\Z锚(只支持^和$)
- 逆序环视(只支持顺序环视)
- 命名分组(只支持0-9编号的捕获组)
- 单行模式和注释模式(只支持m、i、g)
- 模式作用范围
- 纯文本模式

6-5-10 正则表达式属性和方法

前面有提到，当我们使用`typeof`运算符来打印正则表达式的类型时，返回的是`object`，这说明正则表达式在JavaScript中也是一种对象。那么既然是对象，就应该有相应的属性和方法。

1. 实例属性

每个RegExp实例对象都包含如下5个属性

```
global:      布尔值，表示是否设置了g标志
ignoreCase:   布尔值，表示是否设置了i标志
lastIndex:    整数，表示开始搜索下一个匹配项的字符位置，从0算起
multiline:    布尔值，表示是否设置了标志m
source:      正则表达式的字符串表示，按照字面量形式而非传入构造函数中的字符串模式返回
```

示例如下：

```
let reg = /test/gi;
console.log(reg.global);//true
console.log(reg.ignoreCase);//true
console.log(reg.multiline);//false
console.log(reg.lastIndex);//0
console.log(reg.source);//test
```

如果使用RegExp的exec()或test()函数，并且设定了全局模式`g`，正则表达式的匹配就会从`lastIndex`的位置开始，并且在每次匹配成功之后重新设定`lastIndex`，继续往后匹配。这样，就可以在字符串中重复迭代，依次寻找各个匹配结果。但是，如果需要对不同字符串调用

同一个RegExp的`exec()`或`test()`方法，这个变量也可能会带来意料之外的匹配结果，所以在更换字符串时，要显式地将RegExp的`lastIndex`置为0

```
//exec()方法以数组形式返回匹配项
var reg = /\w/g;
var str = "abcd";
console.log(reg.lastIndex);//0
console.log(reg.exec(str));//[ 'a', index: 0, input: 'abcd' ]
console.log(reg.lastIndex);//1
console.log(reg.exec(str));//[ 'b', index: 1, input: 'abcd' ]
console.log(reg.lastIndex);//2
console.log(reg.exec(str));//[ 'c', index: 2, input: 'abcd' ]
console.log(reg.lastIndex);//3
console.log(reg.exec(str));//[ 'd', index: 3, input: 'abcd' ]
console.log(reg.lastIndex);//4
console.log(reg.exec(str));//null
console.log(reg.lastIndex);//0
console.log(reg.exec(str));//[ 'a', index: 0, input: 'abcd' ]
```

2. 构造函数属性

RegExp构造函数属性被看成静态属性，这些属性基于所执行的最近一次正则表达式操作而变化有两种方式访问它们，即长属性名和短属性名。短属性名大都不是有效的ECMAScript标识符，所以必须通过方括号语法来访问它们

长属性名	短属性名	说明
input	$_	最近一次要匹配的字符串
lastMatch	$&	最近一次的匹配项
lastParen	$+	最近一次匹配的捕获组
leftContext	$`	input字符串中lastMatch之前的文本
multiline	$*	布尔值，表示是否所有表达式都使用多行模式
rightContext	$'	input字符串中lastMatch之后的文本

使用这些属性，可以从`exec()`方法或`test()`方法执行的操作中提取出更具体的信息

```
//test()用于测试一个字符串是否匹配某个正则表达式，并返回一个布尔值
var text = 'this has been a short summer';
var pattern = /(.)hort/g;
if(pattern.test(text)){
    console.log(RegExp.input);//'this has been a short summer'
    console.log(RegExp.leftContext);//'this has been a '
    console.log(RegExp.rightContext);//' summer'
    console.log(RegExp.lastMatch);//'short'
```

```
    console.log(RegExp.lastParen);//'s'
    console.log(RegExp.multiline);//undefined
    console.log(RegExp['$_']);//'this has been a short summer'
    console.log(RegExp['$`']);//'this has been a '
    console.log(RegExp["$'"]);//' summer'
    console.log(RegExp['$&']);//'short'
    console.log(RegExp['$+']);//'s'
    console.log(RegExp['$*']);//undefined
}
```

ES有9个用于存储捕获组的构造函数属性，在调用`exec()`或`test()`方法时，这些属性会被自动填充
注意：理论上，应该保存整个表达式匹配文本的RegExp.$0并不存在，值为undefined

```
//RegExp.$1\RegExp.$2\RegExp.$3……到RegExp.$9分别用于存储第一、第二……第九个匹配的捕获组
var text = 'this has been a short summer';
var pattern = /(..)or(.)/g;
if(pattern.test(text)){
    console.log(RegExp.$1);//sh
    console.log(RegExp.$2);//t
}
```

3. 实例方法

RegExp对象的实例方法共5个，分为两类。包括`toString()`、`toLocalString()`、`valueOf()`这3种对象通用方法和`test()`、`exec()`这2种正则匹配方法

对象通用方法

RegExp对象继承了Object对象的通用方法`toString()`、`toLocaleString()`、`valueOf()`这三个方法
`toString()`：`toString()`方法返回正则表达式的字面量
`toLocaleString()`：`toLocaleString()`方法返回正则表达式的字面量
`valueOf()`：`valueOf()`方法返回返回正则表达式对象本身
注意：不论正则表达式的创建方式是哪种，这三个方法都只返回其字面量形式

```
let pattern1 = new RegExp('[bc]at','gi');
console.log(pattern1.toString()); // '/[bc]at/gi'
console.log(pattern1.toLocaleString()); // '/[bc]at/gi'
console.log(pattern1.valueOf()); // /[bc]at/gi
```

```
let pattern2 = /[bc]at/gi;
console.log(pattern2.toString()); // '/[bc]at/gi'
console.log(pattern2.toLocaleString()); // '[bc]at/gi'
console.log(pattern2.valueOf()); // /[bc]at/gi
```

正则匹配方法

正则表达式RegExp对象的正则匹配方法只有两个：分别是`test()`和`exec()`

test()

`test()`方法用来测试正则表达式能否在字符串中找到匹配文本，接收一个字符串参数，匹配时返回true，否则返回false

```
let reg = /test/;
let str = "this is a test";
console.log(reg.test(str));//true
```

在调用`test()`方法时，会造成RegExp对象的`lastIndex`属性的变化。如果指定了全局模式，每次执行`test()`方法时，都会从字符串中的`lastIndex`偏移值开始尝试匹配，所以用同一个RegExp多次验证不同字符串，必须在每次调用之后，将`lastIndex`值置为0

```
var pattern = /^\d{4}-\d{2}-\d{2}$/g;
console.log(pattern.test('2016-06-23'));//true
console.log(pattern.test('2016-06-23'));//false

//正确的做法应该是在验证不同字符串前，先将lastIndex重置为0
var pattern = /^\d{4}-\d{2}-\d{2}$/g;
console.log(pattern.test('2016-06-23'));//true
pattern.lastIndex = 0;
console.log(pattern.test('2016-06-23'));//true
```

前面有介绍过，JS有9个用于存储捕获组的构造函数属性，在调用`exec()`或`test()`方法时，这些属性会被自动填充。注意：理论上，应该保存整个表达式匹配文本的`RegExp.$0`并不存在，值为`undefined`

```
if(/^(\d{4})-(\d{2})-(\d{2})$/.test('2016-06-23')){
    console.log(RegExp.$1);//'2016'
    console.log(RegExp.$2);//'06'
    console.log(RegExp.$3);//'23'
    console.log(RegExp.$0);//undefined
}
```

exec()

`exec()`方法专门为捕获组而设计，接受一个参数，即要应用模式的字符串。然后返回包含匹配项信息的数组，在没有匹配项的情况下返回`null`
在匹配项数组中，第一项是与整个模式匹配的字符串，其他项是与模式中的捕获组匹配的字符串，如果模式中没有捕获组，则该数组只包含一项
返回的数组包含两个额外的属性：`index`和`input`。`index`表示匹配项在字符串的位置，`input`表示应用正则表达式的字符串

```
var text = 'mom and dad and baby and others';
var pattern = /mom( and dad( and baby)?)?/gi;
var matches = pattern.exec(text);
console.log(pattern,matches);
// /mom( and dad( and baby)?)?/gi [ 'mom and dad and baby',
// ' and dad and baby',
// ' and baby',
// index: 0,
// input: 'mom and dad and baby and others' ]
```

对于`exec()`方法而言，即使在模式中设置了全局标志`g`，它每次也只会返回一个匹配项。在不设置全局标志的情况下，在同一个字符串上多次调用`exec()`，将始终返回第一个匹配项的信息

```
var text = 'cat,bat,sat,fat';
var pattern1 = /.at/;
var matches = pattern1.exec(text);
console.log(pattern1,matches);
///.at/ [ 'cat', index: 0, input: 'cat,bat,sat,fat' ]

var text = 'cat,bat,sat,fat';
matches = pattern1.exec(text);
console.log(pattern1,matches);
///.at/ [ 'cat', index: 0, input: 'cat,bat,sat,fat' ]
```

而在设置全局标志的情况下，每次调用exec()都会在字符串中继续查找新匹配项

```
var text = 'cat,bat,sat,fat';
var pattern2 = /.at/g;
var matches = pattern2.exec(text);
console.log(pattern2,matches);
///.at/g [ 'cat', index: 0, input: 'cat,bat,sat,fat' ]

var text = 'cat,bat,sat,fat';
```

```
matches = pattern2.exec(text);
console.log(pattern2,matches);
///.at/g [ 'bat', index: 4, input: 'cat,bat,sat,fat' ]
```

> 注意：用exec()方法找出匹配的所有位置和所有值

```
var string = 'j1h342jg24g234j 3g24j1';
var pattern = /\d/g;
var valueArray = [];//值
var indexArray = [];//位置
var temp;
while((temp=pattern.exec(string)) != null){
    valueArray.push(temp[0]);
    indexArray.push(temp.index);
}
//["1", "3", "4", "2", "2", "4", "2", "3", "4", "3", "2", "4", "1"]
//[1, 3, 4, 5, 8, 9, 11, 12, 13, 16, 18, 19, 21]
console.log(valueArray,indexArray);
```

总结

1. JavaScript里面对象就是一组键值的集合。
2. 访问对象的属性的方式有3种：点访问法，中括号访问法和symbol访问法。
3. symbol访问法是ES6新添加的对象属性访问方式，主要就是为了避免属性名冲突的。
4. 对象也可以进行嵌套和解构。
5. 使用对象作为函数参数的技术被称之为命名参数，好处在于传参时参数顺序可以随意。
6. this的指向默认可以分为2种：指向全局对象和指向当前对象。
7. 使用对象字面量来为一组函数创建一个命名空间可以解决命名冲突的问题。
8. JSON是一种轻量级的数据存储格式，它在人、机可读性方面达到了一个最佳的临界点。
9. Math对象是一个数学对象，提供了很多有用的属性和方法供我们使用。
10. Date类型主要用于处理和时间相关的操作。
11. 正则表达式描述了一种字符串匹配的模式，可以用来检查一个串是否含有某种子串、将匹配的子串替换或者从某个串中取出符合某个条件的子串等。