03 标准对象

在JavaScript的世界里，一切都是对象。

但是某些对象还是和其他对象不太一样。为了区分对象的类型，我们用`typeof`操作符获取对象的类型，它总是返回一个字符串：

```
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'
```

可见，`number`、`string`、`boolean`、`function`和`undefined`有别于其他类型。特别注意`null`的类型是`object`，`Array`的类型也是`object`，如果我们用`typeof`将无法区分出`null`、`Array`和通常意义上的object——`{}`。

包装对象

除了这些类型外，JavaScript还提供了包装对象，熟悉Java的小伙伴肯定很清楚`int`和`Integer`这种暧昧关系。

`number`、`boolean`和`string`都有包装对象。没错，在JavaScript中，字符串也区分`string`类型和它的包装类型。包装对象用`new`创建：

```
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
```

虽然包装对象看上去和原来的值一模一样，显示出来也是一模一样，但他们的类型已经变为`object`了！所以，包装对象和原始值用`===`比较会返回`false`：

```
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
```

所以*闲的蛋疼也不要使用包装对象*！尤其是针对`string`类型！！！

如果我们在使用`Number`、`Boolean`和`String`时，没有写`new`会发生什么情况？

此时，`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'
```

是不是感觉头大了？这就是JavaScript特有的催眠魅力！

总结一下，有这么几条规则需要遵守：

- 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'`。

最后有细心的同学指出，任何对象都有`toString()`方法吗？`null`和`undefined`就没有！确实如此，这两个特殊值要除外，虽然`null`还伪装成了`object`类型。

更细心的同学指出，`number`对象调用`toString()`报SyntaxError：

```
123.toString(); // SyntaxError
```

遇到这种情况，要特殊处理一下：

```
123..toString(); // '123', 注意是两个点！
(123).toString(); // '123'
```

不要问为什么，这就是JavaScript代码的乐趣！

Date

在JavaScript中，`Date`对象用来表示日期和时间。

要获取系统当前时间，用：

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

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

如果要创建一个指定日期和时间的`Date`对象，可以用：

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

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

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

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

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

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

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

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

时区

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

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

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

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

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

要获取当前时间戳，可以用：

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

练习

小明为了和女友庆祝情人节，特意制作了网页，并提前预定了法式餐厅。小明打算用JavaScript给女友一个惊喜留言：

```
'use strict';
var today = new Date();
if (today.getMonth() === 2 && today.getDate() === 14) {
    alert('亲爱的，我预定了晚餐，晚上6点在餐厅见！');
}
```

结果女友并未出现。小明非常郁闷，请你帮忙分析他的JavaScript代码有何问题。

RegExp

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

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

所以我们判断一个字符串是否是合法的Email的方法是：

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

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

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

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

`.`可以匹配任意字符，所以：

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

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

来看一个复杂的例子：`\d{3}\s+\d{3,8}`。

我们来从左到右解读一下：

1. `\d{3}`表示匹配3个数字，例如`'010'`；

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

综合起来，上面的正则表达式可以匹配以任意个空格隔开的带区号的电话号码。

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

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

进阶

要做更精确地匹配，可以用`[]`表示范围，比如：

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

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

`^`表示行的开头，`^\d`表示必须以数字开头。

`$`表示行的结束，`\d$`表示必须以数字结束。

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

RegExp

有了准备知识，我们就可以在JavaScript中使用正则表达式了。

JavaScript有两种方式创建一个正则表达式：

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

两种写法是一样的：

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

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

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

先看看如何判断正则表达式是否匹配：

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

RegExp对象的`test()`方法用于测试给定的字符串是否符合条件。

切分字符串

用正则表达式切分字符串比用固定的字符更灵活，请看正常的切分代码：

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

嗯，无法识别连续的空格，用正则表达式试试：

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

无论多少个空格都可以正常分割。加入`,`试试：

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

再加入`;`试试：

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

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

分组

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

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

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

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

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

`exec()`方法在匹配失败时返回`null`。

提取子串非常有用。来看一个更凶残的例子：

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

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

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

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

贪婪匹配

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

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

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

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

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

全局搜索

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

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

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

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

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

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

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

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

re.exec(s); // null，直到结束仍没有匹配到
```

全局匹配类似搜索，因此不能使用`/^...$/`，那样只会最多匹配一次。

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

小结

正则表达式非常强大，要在短短的一节里讲完是不可能的。要讲清楚正则的所有内容，可以写一本厚厚的书了。如果你经常遇到正则表达式的问题，你可能需要一本正则表达式的参考书。

练习

请尝试写一个验证Email地址的正则表达式。版本一应该可以验证出类似的Email：

```
'use strict';
var re = /^$/;
// 测试:
var
    i,
    success = true,
    should_pass = ['someone@gmail.com',
'bill.gates@microsoft.com', 'tom@voyager.org',
'bob2015@163.com'],
    should_fail = ['test#gmail.com', 'bill@microsoft',
'bill%gates@ms.com', '@voyager.org'];
for (i = 0; i < should_pass.length; i++) {
    if (!re.test(should_pass[i])) {
```

```
            console.log('测试失败: ' + should_pass[i]);
            success = false;
            break;
        }
    }
    for (i = 0; i < should_fail.length; i++) {
        if (re.test(should_fail[i])) {
            console.log('测试失败: ' + should_fail[i]);
            success = false;
            break;
        }
    }
    if (success) {
        console.log('测试通过!');
    }
```

版本二可以验证并提取出带名字的Email地址：

```
'use strict';
var re = /^$/;
// 测试:
var r = re.exec('<Tom Paris> tom@voyager.org');
if (r === null || r.toString() !== ['<Tom Paris>
tom@voyager.org', 'Tom Paris',
'tom@voyager.org'].toString()) {
    console.log('测试失败!');
}
else {
    console.log('测试成功!');
}
```

JSON

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

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

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

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

- number：和JavaScript的`number`完全一致；
- boolean：就是JavaScript的`true`或`false`；
- string：就是JavaScript的`string`；

- null：就是JavaScript的`null`；
- array：就是JavaScript的`Array`表示方式——`[]`；
- object：就是JavaScript的`{ ... }`表示方式。

以及上面的任意组合。

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

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

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

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

序列化

让我们先把小明这个对象序列化成JSON格式的字符串：

```
'use strict';

var xiaoming = {
    name: '小明',
    age: 14,
    gender: true,
    height: 1.65,
    grade: null,
    'middle-school': '\"W3C\" Middle School',
    skills: ['JavaScript', 'Java', 'Python', 'Lisp']
};
var s = JSON.stringify(xiaoming);
console.log(s);
```

要输出得好看一些，可以加上参数，按缩进输出：

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

结果：

```
{
  "name": "小明",
  "age": 14,
  "gender": true,
  "height": 1.65,
  "grade": null,
```

```
  "middle-school": "\"W3C\" Middle School",
  "skills": [
    "JavaScript",
    "Java",
    "Python",
    "Lisp"
  ]
}
```

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

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

结果：

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

还可以传入一个函数，这样对象的每个键值对都会被函数先处理：

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

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

上面的代码把所有属性值都变成大写：

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

```
    "PYTHON",
    "LISP"
  ]
}
```

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

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

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

反序列化

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

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

`JSON.parse()`还可以接收一个函数，用来转换解析出的属性：

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

在JavaScript中使用JSON，就是这么简单！

练习

用浏览器访问OpenWeatherMap的天气API，查看返回的JSON数据，然后返回城市、天气预报等信息：

```
'use strict'

var url =
'https://api.openweathermap.org/data/2.5/forecast?
q=Beijing,cn&appid=800f49846586c3ba6e7052cfc89af16c';
$.getJSON(url, function (data) {
    var info = {
        city: data.city.name,
        weather: data.list[0].weather[0].main,
        time: data.list[0].dt_txt
    };
    alert(JSON.stringify(info, null, '  '));
});
```