JavaScript

参考网站：https://www.liaoxuefeng.com/wiki/1022910821149312

1、概述

1.1、前言

JavaScript是世界上最流行的脚本语言，因为你在电脑、手机、平板上浏览的所有的网页，以及无数基于HTML5的手机App，交互逻辑都是由JavaScript驱动的。

简单地说，JavaScript是一种运行在浏览器中的解释型的编程语言。

那么问题来了，为什么我们要学JavaScript？尤其是当你已经掌握了某些其他编程语言如Java、C++的情况下。

简单粗暴的回答就是：因为你没有选择。在Web世界里，只有JavaScript能跨平台、跨浏览器驱动网页，与用户交互。

Flash背后的ActionScript曾经流行过一阵子，不过随着移动应用的兴起，没有人用Flash开发手机App，所以它目前已经边缘化了。相反，随着HTML5在PC和移动端越来越流行，JavaScript变得更加重要了。并且，新兴的Node.js把JavaScript引入到了服务器端，JavaScript已经变成了全能型选手。

JavaScript一度被认为是一种玩具编程语言，它有很多缺陷，所以不被大多数后端开发人员所重视。很多人认为，写JavaScript代码很简单，并且JavaScript只是为了在网页上添加一点交互和动画效果。

但这是完全错误的理解。JavaScript确实很容易上手，但其精髓却不为大多数开发人员所熟知。编写高质量的JavaScript代码更是难上加难。

一个合格的开发人员应该精通JavaScript和其他编程语言。如果你已经掌握了其他编程语言，或者你还什么都不会，请立刻开始学习JavaScript，不要被Web时代所淘汰。

1.2、JavaScript历史

要了解JavaScript，我们首先要回顾一下JavaScript的诞生。

在上个世纪的1995年，当时的网景公司正凭借其Navigator浏览器成为Web时代开启时最著名的第一代互联网公司。

由于网景公司希望能在静态HTML页面上添加一些动态效果，于是叫Brendan Eich这哥们在两周之内设计出了JavaScript语言。你没看错，这哥们只用了10天时间。

为什么起名叫JavaScript？原因是当时Java语言非常红火，所以网景公司希望借Java的名气来推广，但事实上JavaScript除了语法上有点像Java，其他部分基本上没啥关系。

1.3、ECMAScript

因为网景开发了JavaScript，一年后微软又模仿JavaScript开发了JScript，为了让JavaScript成为全球标准，几个公司联合ECMA（European Computer Manufacturers Association）组织定制了JavaScript语言的标准，被称为ECMAScript标准。

所以简单说来就是，ECMAScript是一种语言标准，而JavaScript是网景公司对ECMAScript标准的一种实现。

那为什么不直接把JavaScript定为标准呢？因为JavaScript是网景的注册商标。

不过大多数时候，我们还是用JavaScript这个词。如果你遇到ECMAScript这个词，简单把它替换为JavaScript就行了。

1.4、JavaScript版本

JavaScript语言是在10天时间内设计出来的，虽然语言的设计者水平非常NB，但谁也架不住“时间紧，任务重”，所以，JavaScript有很多设计缺陷，我们后面会慢慢讲到。

此外，由于JavaScript的标准——ECMAScript在不断发展，最新版ECMAScript 6标准（简称ES6）已经在2015年6月正式发布了，所以，讲到JavaScript的版本，实际上就是说它实现了ECMAScript标准的哪个版本。

由于浏览器在发布时就确定了JavaScript的版本，加上很多用户还在使用IE6这种古老的浏览器，这就导致你在写JavaScript的时候，要照顾一下老用户，不能一上来就用最新的ES6标准写，否则，老用户的浏览器是无法运行新版本的JavaScript代码的。

不过，JavaScript的核心语法并没有多大变化。我们的教程会先讲JavaScript最核心的用法，然后，针对ES6讲解新增特性。

1.5、引入JavaScript

JavaScript代码可以直接嵌在网页的任何地方，不过通常我们都把JavaScript代码放到`head`中：

```
<html>
<head>
  <script>
    alert('Hello, world');
  </script>
</head>
<body>
  ...
</body>
</html>


```

由`<script>...</script>`包含的代码就是JavaScript代码，它将直接被浏览器执行。

第二种方法是把JavaScript代码放到一个单独的.js文件，然后在HTML中通过`<script src="..."></script>`引入这个文件：

```
<html>
<head>
  <script src="/static/js/abc.js"></script>
</head>
<body>
  ...
</body>
</html>
```

这样，`/static/js/abc.js`就会被浏览器执行。

把JavaScript代码放入一个单独的`.js`文件中更利于维护代码，并且多个页面可以各自引用同一份`.js`文件。

可以在同一个页面中引入多个`.js`文件

还可以在页面中多次编写`<script> js代码... </script>`，浏览器按照顺序依次执行。

有些时候你会看到`<script>`标签还设置了一个type属性：

```
<script type="text/javascript">
  ...
</script>
```

但这是没有必要的，因为默认的`type`就是JavaScript，所以不必显式地把`type`指定为JavaScript。

1.6、IDE推荐

可以用任何文本编辑器来编写JavaScript代码。这里我们推荐以下几种文本编辑器：

- Visual Studio Code

```
微软出的Visual Studio Code，可以看做迷你版Visual Studio，免费！跨平台！内置JavaScript支持，强烈推荐使用！
```

- Sublime Text

```
Sublime Text是一个好用的文本编辑器，免费，但不注册会不定时弹出提示框。
```

- Notepad++

```
Notepad++也是免费的文本编辑器，但仅限Windows下使用。
```

- WebStorm

```
WebStorm 是jetbrains公司旗下一款JavaScript 开发工具。目前已经被广大中国JS开发者誉为“Web前端开发神器”、“最强大的HTML5编辑器”、“最智能的JavaScript IDE”等。
```

- IDEA

```
Java开发人员必备，NB！
```

- HBuilder

```
HBuilder是DCloud（数字天堂）推出的一款支持HTML5的Web开发IDE。HBuilder的编写用到了Java、C、Web和Ruby。HBuilder本身主体是由Java编写。它基于Eclipse，所以顺其自然地兼容了Eclipse的插件。
```

1.7、运行JavaScript

要让浏览器运行JavaScript，必须先有一个HTML页面，在HTML页面中引入JavaScript，然后，让浏览器加载该HTML页面，就可以执行JavaScript代码。

你也许会想，直接在我的硬盘上创建好HTML和JavaScript文件，然后用浏览器打开，不就可以看到效果了吗？

这种方式运行部分JavaScript代码没有问题，但由于浏览器的安全限制，以`file://`开头的地址无法执行如联网等JavaScript代码，最终，你还是需要架设一个Web服务器，然后以`http://`开头的地址来正常执行所有JavaScript代码。

我的第一个javaScript程序

```
<html>
<head>
  <script>
// 以双斜杠开头直到行末的是注释，注释是给人看的，会被浏览器忽略
/* 在这中间的也是注释，将被浏览器忽略 */
        alert('Hello,world');
  </script>
</head>
<body>
</body>
</html>
```

浏览器将弹出一个对话框，显示“Hello, world”。你也可以修改两个单引号中间的内容，再试着运行。

1.8、调试

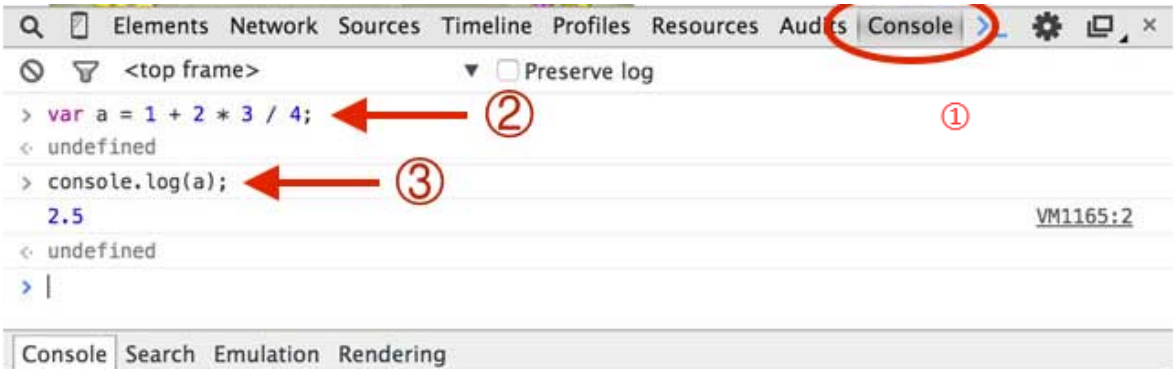
俗话说得好，“工欲善其事，必先利其器。”，写JavaScript的时候，如果期望显示`ABC`，结果却显示`XYZ`，到底代码哪里出了问题？不要抓狂，也不要泄气，作为小白，要坚信：JavaScript本身没有问题，浏览器执行也没有问题，有问题的一定是我的代码。

如何找出问题代码？这就需要调试。

怎么在浏览器中调试JavaScript代码呢？

首先，你需要安装Google Chrome浏览器，Chrome浏览器对开发者非常友好，可以让你方便地调试JavaScript代码。

安装后，随便打开一个网页，然后点击菜单“查看(View)”-“开发者(Developer)”-“开发者工具(Developer Tools)”，浏览器窗口就会一分为二，下方就是开发者工具：



先点击“控制台(Console)”，在这个面板里可以直接输入JavaScript代码，按回车后执行。

要查看一个变量的内容，在Console中输入`console.log(a);`，回车后显示的值就是变量的内容。

关闭Console请点击右上角的“×”按钮。请熟练掌握Console的使用方法，在编写JavaScript代码时，经常需要在Console运行测试代码。

如果你对自己还有更高的要求，可以研究开发者工具的“源码(Sources)”，掌握断点、单步执行等高级调试技巧。

2、快速入门

2.1、基本语法

JavaScript的语法和Java语言类似，每个语句以`;`结束，语句块用`{...}`。

但是，JavaScript并不强制要求在每个语句的结尾加`;`，浏览器中负责执行JavaScript代码的引擎会自动在每个语句的结尾补上`;`。

> 让JavaScript引擎自动加分号在某些情况下会改变程序的语义，导致运行结果与期望不一致。在本教程中，我们不会省略;，所有语句都会添加;。

例如，下面的一行代码就是一个完整的赋值语句：

```
var x = 1;
```

下面的一行代码是一个字符串，但仍然可以视为一个完整的语句：

```
'Hello, world';
```

下面的一行代码包含两个语句，每个语句用`;`表示语句结束：

```
var x = 1; var y = 2; // 不建议一行写多个语句!
```

语句块是一组语句的集合，例如，下面的代码先做了一个判断，如果判断成立，将执行`{...}`中的所有语句：

```
if (2 > 1) {
    x = 1;
    y = 2;
    z = 3;
}
```

注意花括号`{...}`内的语句具有缩进，通常是4个空格。缩进不是JavaScript语法要求必须的，但缩进有助于我们理解代码的层次，所以编写代码时要遵守缩进规则。很多文本编辑器具有“自动缩进”的功能，可以帮助整理代码。

`{...}`还可以嵌套，形成层级结构：

```
if (2 > 1) {
    x = 1;
    y = 2;
    z = 3;
    if (x < y) {
        z = 4;
    }
    if (x > y) {
        z = 5;
    }
}
```

JavaScript本身对嵌套的层级没有限制，但是过多的嵌套无疑会大大增加看懂代码的难度。遇到这种情况，需要把部分代码抽出来，作为函数来调用，这样可以减少代码的复杂度。

以`//`开头直到行末的字符被视为行注释，注释是给开发人员看到，JavaScript引擎会自动忽略：

```
// 这是一行注释
alert('hello'); // 这也是注释
```

另一种块注释是用`/*...*/`把多行字符包裹起来，把一大“块”视为一个注释：

```
/* 从这里开始是块注释
仍然是注释
仍然是注释
注释结束 */
```

请注意，JavaScript严格区分大小写，如果弄错了大小写，程序将报错或者运行不正常。

2.2、数据类型和变量

数据类型

计算机顾名思义就是可以做数学计算的机器，因此，计算机程序理所当然地可以处理各种数值。但是，计算机能处理的远不止数值，还可以处理文本、图形、音频、视频、网页等各种各样的数据，不同的数据，需要定义不同的数据类型。在JavaScript中定义了以下几种数据类型：

Number

JavaScript不区分整数和浮点数，统一用Number表示，以下都是合法的Number类型：

```
123; // 整数123
0.456; // 浮点数0.456
1.2345e3; // 科学计数法表示1.2345x1000，等同于1234.5
-99; // 负数
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
```

计算机由于使用二进制，所以，有时候用十六进制表示整数比较方便，十六进制用0x前缀和0-9，a-f表示，例如：`0xff00`，`0xa5b4c3d2`，等等，它们和十进制表示的数值完全一样。

Number可以直接做四则运算，规则和数学一致：

```
1 + 2; // 3
(1 + 2) * 5 / 2; // 7.5
2 / 0; // Infinity
0 / 0; // NaN
10 % 3; // 1
10.5 % 3; // 1.5
```

注意`%`是求余运算。

字符串

字符串是以单引号'或双引号"括起来的任意文本，比如`'abc'`，`"xyz"`等等。请注意，`''`或`""`本身只是一种表示方式，不是字符串的一部分，因此，字符串`'abc'`只有`a`，`b`，`c`这3个字符。

布尔值

布尔值和布尔代数的表示完全一致，一个布尔值只有`true`、`false`两种值，要么是`true`，要么是`false`，可以直接用`true`、`false`表示布尔值，也可以通过布尔运算计算出来：

```
true; // 这是一个true值
false; // 这是一个false值
2 > 1; // 这是一个true值
2 >= 3; // 这是一个false值
```

`&&`运算是与运算，只有所有都为`true`，`&&`运算结果才是`true`：

```
true && true; // 这个&&语句计算结果为true
true && false; // 这个&&语句计算结果为false
false && true && false; // 这个&&语句计算结果为false
```

`||`运算是或运算，只要其中有一个为`true`，`||`运算结果就是`true`：

```
false || false; // 这个||语句计算结果为false
true || false; // 这个||语句计算结果为true
false || true || false; // 这个||语句计算结果为true
```

`!`运算是非运算，它是一个单目运算符，把`true`变成`false`，`false`变成`true`：

```
! true; // 结果为false
! false; // 结果为true
! (2 > 5); // 结果为true
```

布尔值经常用在条件判断中，比如：

```
var age = 15;
if (age >= 18) {
    alert('adult');
} else {
    alert('teenager');
}
```

比较运算符

当我们对Number做比较时，可以通过比较运算符得到一个布尔值：

```
2 > 5; // false
5 >= 2; // true
7 == 7; // true
```

实际上，JavaScript允许对任意数据类型做比较：

```
false == 0; // true
false === 0; // false
```

要特别注意相等运算符`==`。JavaScript在设计时，有两种比较运算符：

第一种是`==`比较，它会自动转换数据类型再比较，很多时候，会得到非常诡异的结果；

第二种是`===`比较，它不会自动转换数据类型，如果数据类型不一致，返回`false`，如果一致，再比较。

由于JavaScript这个设计缺陷，*不要*使用`==`比较，始终坚持使用`===`比较。

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

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

唯一能判断`NaN`的方法是通过`isNaN()`函数：

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

最后要注意浮点数的相等比较：

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

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

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

null和undefined

`null`表示一个“空”的值，它和`0`以及空字符串`''`不同，`0`是一个数值，`''`表示长度为0的字符串，而`null`表示“空”。

在其他语言中，也有类似JavaScript的`null`的表示，例如Java也用`null`，Swift用`nil`，Python用`None`表示。但是，在JavaScript中，还有一个和`null`类似的`undefined`，它表示“未定义”。

JavaScript的设计者希望用`null`表示一个空的值，而`undefined`表示值未定义。事实证明，这并没有什么卵用，区分两者的意义不大。大多数情况下，我们都应该用`null`。`undefined`仅仅在判断函数参数是否传递的情况下有用。

数组

数组是一组按顺序排列的集合，集合的每个值称为元素。JavaScript的数组可以包括任意数据类型。例如：

```
[1, 2, 3.14, 'Hello', null, true];
```

上述数组包含6个元素。数组用`[]`表示，元素之间用`,`分隔。

另一种创建数组的方法是通过`Array()`函数实现：

```
new Array(1, 2, 3); // 创建了数组[1, 2, 3]
```

然而，出于代码的可读性考虑，强烈建议直接使用`[]`。

数组的元素可以通过索引来访问。请注意，索引的起始值为`0`：

```
var arr = [1, 2, 3.14, 'Hello', null, true];
arr[0]; // 返回索引为0的元素，即1
arr[5]; // 返回索引为5的元素，即true
arr[6]; // 索引超出了范围，返回undefined
```

对象

JavaScript的对象是一组由键-值组成的无序集合，例如：

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

JavaScript对象的键都是字符串类型，值可以是任意数据类型。上述`person`对象一共定义了6个键值对，其中每个键又称为对象的属性，例如，`person`的`name`属性为`'Bob'`，`zipcode`属性为`null`。

要获取一个对象的属性，我们用`对象变量.属性名`的方式：

```
person.name; // 'Bob'
person.zipcode; // null
```

变量

变量的概念基本上和初中代数的方程变量是一致的，只是在计算机程序中，变量不仅可以是数字，还可以是任意数据类型。

变量在JavaScript中就是用一个变量名表示，变量名是大小写英文、数字、`$`和`_`的组合，且不能用数字开头。变量名也不能是JavaScript的关键字，如`if`、`while`等。申明一个变量用`var`语句，比如：

```
var a; // 申明了变量a，此时a的值为undefined
var $b = 1; // 申明了变量$b，同时给$b赋值，此时$b的值为1
var s_007 = '007'; // s_007是一个字符串
var Answer = true; // Answer是一个布尔值true
var t = null; // t的值是null
```

变量名也可以用中文，但是，请不要给自己找麻烦。

在JavaScript中，使用等号`=`对变量进行赋值。可以把任意数据类型赋值给变量，同一个变量可以反复赋值，而且可以是不同类型的变量，但是要注意只能用`var`申明一次，例如：

```
var a = 123; // a的值是整数123
a = 'ABC'; // a变为字符串
```

这种变量本身类型不固定的语言称之为动态语言，与之对应的是静态语言。静态语言在定义变量时必须指定变量类型，如果赋值的时候类型不匹配，就会报错。例如Java是静态语言，赋值语句如下：

```
int a = 123; // a是整数类型变量，类型用int申明
a = "ABC"; // 错误：不能把字符串赋给整型变量
```

和静态语言相比，动态语言更灵活，就是这个原因。

请不要把赋值语句的等号等同于数学的等号。比如下面的代码：

```
var x = 10;
x = x + 2;
```

如果从数学上理解`x = x + 2`那无论如何是不成立的，在程序中，赋值语句先计算右侧的表达式`x + 2`，得到结果`12`，再赋给变量`x`。由于`x`之前的值是`10`，重新赋值后，`x`的值变成`12`。

打印变量X，要显示变量的内容，可以用`console.log(x)`，打开Chrome的控制台就可以看到结果。

```
var x = 100;
console.log(x);
```

使用`console.log()`代替`alert()`的好处是可以避免弹出烦人的对话框。

2.3、strict模式

JavaScript在设计之初，为了方便初学者学习，并不强制要求用`var`申明变量。这个设计错误带来了严重的后果：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

```
i = 10; // i现在是全局变量
```

在同一个页面的不同的JavaScript文件中，如果都不用`var`申明，恰好都使用了变量`i`，将造成变量`i`互相影响，产生难以调试的错误结果。

使用`var`申明的变量则不是全局变量，它的范围被限制在该变量被申明的函数体内（函数的概念将稍后讲解），同名变量在不同的函数体内互不冲突。

为了修补JavaScript这一严重设计缺陷，ECMA在后续规范中推出了strict模式，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。

启用strict模式的方法是在JavaScript代码的第一行写上：

```
'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。

测试一下你的浏览器是否能支持strict模式：

```
'use strict';
// 如果浏览器支持strict模式，下面的代码将报ReferenceError错误:
abc = 'Hello, world';
console.log(abc);
```

运行代码，如果浏览器报错，请修复后再运行。如果浏览器不报错，说明你的浏览器太古老了，需要尽快升级。

不用`var`申明的变量会被视为全局变量，为了避免这一缺陷，所有的JavaScript代码都应该使用strict模式。我们在后面编写的JavaScript代码将全部采用strict模式。

2.4、字符串

JavaScript的字符串就是用`''`或`""`括起来的字符表示。

如果`'`本身也是一个字符，那就可以用`""`括起来，比如`"I'm OK"`包含的字符是`I`，`'`，`m`，空格，`O`，`K`这6个字符。

如果字符串内部既包含`'`又包含`"`怎么办？可以用转义字符`\`来标识，比如：

```
'I\'m \"OK\"!';
```

表示的字符串内容是：`I'm "OK"!`

转义字符`\`可以转义很多字符，比如`\n`表示换行，`\t`表示制表符，字符`\`本身也要转义，所以`\\`表示的字符就是`\`。

ASCII字符可以以`\x##`形式的十六进制表示，例如：

```
'\x41'; // 完全等同于 'A'
```

还可以用`\u####`表示一个Unicode字符：

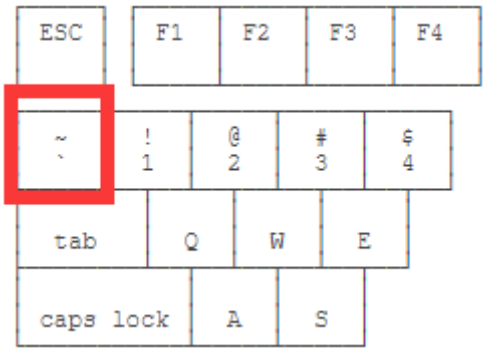
```
'\u4e2d\u6587'; // 完全等同于 '中文'
```

多行字符串

由于多行字符串用`\n`写起来比较费事，所以最新的ES6标准新增了一种多行字符串的表示方法，用反引号 ``表示：

```
`这是一个
多行
字符串`;
```

注意：反引号在键盘的ESC下方，数字键1的左边：



模板字符串

要把多个字符串连接起来，可以用`+`号连接：

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

如果有很多变量需要连接，用`+`号就比较麻烦。ES6新增了一种模板字符串，表示方法和上面的多行字符串一样，但是它会自动替换字符串中的变量：

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

操作字符串

字符串常见的操作如下：

```
var s = 'Hello, world!';
s.length; // 13
```

要获取字符串某个指定位置的字符，使用类似Array的下标操作，索引号从0开始：

```
var s = 'Hello, world!';

s[0]; // 'H'
s[6]; // ' '
s[7]; // 'w'
s[12]; // '!'
s[13]; // undefined 超出范围的索引不会报错，但一律返回undefined
```

需要特别注意的是，字符串是不可变的，如果对字符串的某个索引赋值，不会有任何错误，但是，也没有任何效果：

```
var s = 'Test';
s[0] = 'X';
alert(s); // s仍然为'Test'
```

JavaScript为字符串提供了一些常用方法，注意，调用这些方法本身不会改变原有字符串的内容，而是返回一个新字符串：

- `toUpperCase()`把一个字符串全部变为大写

```
var s = 'Hello';
s.toUpperCase(); // 返回'HELLO'
```

- `toLowerCase()`把一个字符串全部变为小写

```
var s = 'Hello';
var lower = s.toLowerCase(); // 返回'hello'并赋值给变量lower
lower; // 'hello'
```

- `indexOf()`会搜索指定字符串出现的位置

```
var s = 'hello, world';
s.indexOf('world'); // 返回7
s.indexOf('World'); // 没有找到指定的子串，返回-1
```

- `substring()`返回指定索引区间的子串

```
var s = 'hello, world'
s.substring(0, 5); // 从索引0开始到5（不包括5），返回'hello'
s.substring(7); // 从索引7开始到结束，返回'world'
```

2.5、数组

JavaScript的`Array`可以包含任意数据类型，并通过索引来访问每个元素。

要取得`Array`的长度，直接访问`length`属性：

```
var arr = [1, 2, 3.14, 'Hello', null, true];
arr.length; // 6
```

请注意，直接给`Array`的`length`赋一个新的值会导致`Array`大小的变化：

```
var arr = [1, 2, 3];
arr.length; // 3
arr.length = 6;
arr; // arr变为[1, 2, 3, undefined, undefined, undefined]
arr.length = 2;
arr; // arr变为[1, 2]
```

`Array`可以通过索引把对应的元素修改为新的值，因此，对`Array`的索引进行赋值会直接修改这个`Array`

```
var arr = ['A', 'B', 'C'];
arr[1] = 99;
arr; // arr现在变为['A', 99, 'C']
```

请注意，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：

```
var arr = [1, 2, 3];
arr[5] = 'x';
arr; // arr变为[1, 2, 3, undefined, undefined, 'x']
```

大多数其他编程语言不允许直接改变数组的大小，越界访问索引会报错。然而，JavaScript的`Array`却不会有任何错误。在编写代码时，不建议直接修改`Array`的大小，访问索引时要确保索引不会越界。

常用方法

- **indexOf**

  与String类似，`Array`也可以通过`indexOf()`来搜索一个指定的元素的位置：

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

  注意了，数字`30`和字符串`'30'`是不同的元素。

- **slice**

  `slice()`就是对应String的`substring()`版本，它截取`Array`的部分元素，然后返回一个新的`Array`

```
var arr = ['A', 'B', 'C', 'D', 'E', 'F', 'G'];
arr.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']
arr.slice(3); // 从索引3开始到结束: ['D', 'E', 'F', 'G']
```

  注意到`slice()`的起止参数包括开始索引，不包括结束索引。

  如果不给`slice()`传递任何参数，它就会从头到尾截取所有元素。利用这一点，我们可以很容易地复制一个`Array`：

```
var arr = ['A', 'B', 'C', 'D', 'E', 'F', 'G'];
var aCopy = arr.slice();
aCopy; // ['A', 'B', 'C', 'D', 'E', 'F', 'G']
aCopy === arr; // false
```

- **push和pop**

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

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

- **unshift和shift**

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

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

- **sort**

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

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

能否按照我们自己指定的顺序排序呢？完全可以，我们将在后面的函数中讲到。

- **reverse**

`reverse()`把整个`Array`的元素给掉个个，也就是反转

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

- **splice**

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

- **concat**

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

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

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

实际上，`concat()`方法可以接收任意个元素和`Array`，并且自动把`Array`拆开，然后全部添加到新的`Array`里：

```
var arr = ['A', 'B', 'C'];
arr.concat(1, 2, [3, 4]); // ['A', 'B', 'C', 1, 2, 3, 4]
```

- **join**

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

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

  如果`Array`的元素不是字符串，将自动转换为字符串后再连接。

- **多维数组**

  如果数组的某个元素又是一个`Array`，则可以形成多维数组，例如：

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

  上述`Array`包含3个元素，其中头两个元素本身也是`Array`。

  练习：如何通过索引取到`500`这个值：

  ```
  var x = arr[1][2];
  console.log(x); // x应该为500
  ```

2.6、对象

JavaScript的对象是一种无序的集合数据类型，它由若干键值对组成。

JavaScript的对象用于描述现实世界中的某个对象。

```
var person = {
    name: '小明',
    birth: 1990,
    school: 'No.1 Middle School',
    height: 1.70,
    weight: 65,
    score: null
}
```

1、定义一个对象

```
var 对象名 = {
    key: 'value',
    key: 'value',
    key: 'value'
}
```

2、获取对象的属性

```
对象.属性
```

3、由于JavaScript的对象是动态类型，你可以自由地给一个对象添加或删除属性：

```
var xiaoming = {
    name: '小明'
};
xiaoming.age; // undefined
xiaoming.age = 18; // 新增一个age属性
xiaoming.age; // 18
delete xiaoming.age; // 删除age属性
xiaoming.age; // undefined
delete xiaoming['name']; // 删除name属性
xiaoming.name; // undefined
delete xiaoming.school; // 删除一个不存在的school属性也不会报错
```

如果我们要检测对象是否拥有某一属性，可以用 in 操作符：

```
var xiaoming = {
    name: '小明',
    birth: 1990,
    school: 'No.1 Middle School',
    height: 1.70,
    weight: 65,
    score: null
};
'name' in xiaoming; // true
'grade' in xiaoming; // false
```

不过要小心，如果用 in 判断一个属性存在，这个属性不一定是 这个对象的，它可能是这个对象继承得到的：

```
'toString' in xiaoming; // true
```

因为 toString 定义在 object 对象中，而所有对象最终都会在原型链上指向 object，所以xiaoming 也拥有 toString 属性。

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：

```
var xiaoming = {
    name: '小明'
};
xiaoming.hasOwnProperty('name'); // true
xiaoming.hasOwnProperty('toString'); // false
```

2.7、流程控制

> if 判断

```
var age = 3;
if (age >= 18) {
    alert('adult');
} else if (age >= 6) {
    alert('teenager');
} else {
    alert('kid');
}
```

> for循环

基础语法

```
var x = 0;
var i;
for (i=1; i<=10000; i++) {
    x = x + i;
}
x; // 50005000
```

遍历数组

```
var arr = ['Apple', 'Google', 'Microsoft'];
var i, x;
for (i=0; i<arr.length; i++) {
    x = arr[i];
    console.log(x);
}
```

无限循环

```
var x = 0;
for (;;) { // 将无限循环下去
    if (x > 100) {
        break; // 通过if判断来退出循环
    }
    x ++;
}
```

for ... in， 它可以把一个对象的所有属性依次循环出来：

```
var o = {
    name: 'Jack',
    age: 20,
    city: 'Beijing'
};
for (var key in o) {
    if (o.hasOwnProperty(key)) {
        console.log(key); // 'name', 'age', 'city'
    }
}
```

由于 Array 也是对象，而它的每个元素的索引被视为对象的属性，所以遍历出来是下标

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

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

> while循环

基本操作

```
var x = 0;
var n = 99;
while (n > 0) {
    x = x + n;
    n = n - 2;
}
x; // 2500
```

do.....while

```
var n = 0;
do {
    n = n + 1;
} while (n < 100);
n; // 100
```

在编写循环代码时，务必小心编写初始条件和判断条件，尤其是边界值。

特别注意`i < 100`和`i <= 100`是不同的判断逻辑。

2.8、Map 和 Set

JavaScript的默认对象表示方式 { } 可以视为其他语言中的 Map 或 Dictionary 的数据结构，即一组键值对。但是JavaScript的对象有个小问题，就是键必须是字符串。但实际上Number或者其他数据类型作为键也是非常合理的。

为了解决这个问题，最新的ES6规范引入了新的数据类型`Map`。

> Map

`Map`是一组键值对的结构，具有极快的查找速度。

举个例子，假设要根据同学的名字查找对应的成绩，如果用`Array`实现，需要两个`Array`：

```
var names = ['Michael', 'Bob', 'Tracy'];
var scores = [95, 75, 85];
```

给定一个名字，要查找对应的成绩，就先要在names中找到对应的位置，再从scores取出对应的成绩，Array越长，耗时越长。

如果用Map实现，只需要一个“名字”-“成绩”的对照表，直接根据名字查找成绩，无论这个表有多大，查找速度都不会变慢。用JavaScript写一个Map如下：

```
var m = new Map([['Michael', 95], ['Bob', 75], ['Tracy', 85]]);
m.get('Michael'); // 95
```

初始化`Map`需要一个二维数组，或者直接初始化一个空`Map`。`Map`具有以下方法：

```
var m = new Map(); // 空Map
m.set('Adam', 67); // 添加新的key-value
m.set('Bob', 59);
m.has('Adam'); // 是否存在key 'Adam': true
m.get('Adam'); // 67
m.delete('Adam'); // 删除key 'Adam'
m.get('Adam'); // undefined
```

由于一个key只能对应一个value，所以，多次对一个key放入value，后面的值会把前面的值冲掉：

```
var m = new Map();
m.set('Adam', 67);
m.set('Adam', 88);
m.get('Adam'); // 88
```

> Set

`Set`和`Map`类似，也是一组key的集合，但不存储value。由于key不能重复，所以，在`Set`中，没有重复的key。

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

重复元素在`Set`中自动被过滤：

```
var s = new Set([1, 2, 3, 3, '3']);
s; // Set {1, 2, 3, "3"}
```

注意数字`3`和字符串`'3'`是不同的元素。

通过`add(key)`方法可以添加元素到`Set`中，可以重复添加，但不会有效果：

```
s.add(4);
s; // Set {1, 2, 3, 4}
s.add(4);
s; // 仍然是 Set {1, 2, 3, 4}
```

通过`delete(key)`方法可以删除元素：

```
var s = new Set([1, 2, 3]);
s; // Set {1, 2, 3}
s.delete(3);
s; // Set {1, 2}
```

2.9、Iterable

遍历 Array 可以采用下标循环，遍历Map 和 Set 就无法使用下标。

为了统一集合类型，ES6标准引入了新的`iterable`类型，Array，Map，Set 属于；

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

> 遍历集合

```
var a = ['A', 'B', 'C'];
var s = new Set(['A', 'B', 'C']);
var m = new Map([[1, 'x'], [2, 'y'], [3, 'z']]);

for (var x of a) { // 遍历Array
    console.log(x);
}
for (var x of s) { // 遍历Set
    console.log(x);
}
for (var x of m) { // 遍历Map
    console.log(x[0] + '=' + x[1]);
}
```

更好的方式是直接使用`iterable`内置的`forEach`方法，它接收一个函数，每次迭代就自动回调该函数。以`Array`为例

```
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);
});
```

`forEach()`方法是ES5.1标准引入的，你需要测试浏览器是否支持。

`Set`没有索引，因此回调函数的前两个参数都是元素本身：

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

`Map`的回调函数参数依次为`value`、`key`和`map`本身：

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

3、函数

3.1、函数定义和调用

> 1、定义函数方式一

```
function abs(x) {
    if (x >= 0) {
        return x;
    } else {
        return -x;
    }
}
```

一旦执行到`return`时，函数就执行完毕，并将结果返回。

如果没有`return`语句，函数执行完毕后也会返回结果，只是结果为`undefined`。

> 2、第二种定义函数的方式如下

```
var abs = function (x) {
    if (x >= 0) {
        return x;
    } else {
        return -x;
    }
};
```

在这种方式下，`function (x) { ... }`是一个匿名函数，它没有函数名。但是，这个匿名函数赋值给了变量`abs`，所以，通过变量`abs`就可以调用该函数。

上述两种定义*完全等价*，注意第二种方式按照完整语法需要在函数体末尾加一个`;`，表示赋值语句结束。

> 3、调用函数

调用函数时，按顺序传入参数即可：

```
abs(10); // 返回10
abs(-9); // 返回9
```

由于JavaScript允许传入任意个参数而不影响调用，因此传入的参数比定义的参数多也没有问题，虽然函数内部并不需要这些参数：

```
abs(10, 'blablabla'); // 返回10
abs(-9, 'haha', 'hehe', null); // 返回9
```

传入的参数比定义的少也没有问题：

```
abs(); // 返回NaN
```

此时`abs(x)`函数的参数`x`将收到`undefined`，计算结果为`NaN`。

要避免收到`undefined`，可以对参数进行检查：

```
function abs(x) {
    if (typeof x !== 'number') {
        throw 'Not a number';
    }
    if (x >= 0) {
        return x;
    } else {
        return -x;
    }
}
```

3、arguments

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

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

利用`arguments`，你可以获得调用者传入的所有参数。也就是说，即使函数不定义任何参数，还是可以拿到参数的值：

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

abs(); // 0
abs(10); // 10
abs(-9); // 9
```

实际上`arguments`最常用于判断传入参数的个数。你可能会看到这样的写法：

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

> rest参数

由于JavaScript函数允许接收任意个参数，于是我们就不得不用`arguments`来获取所有参数：

```
function foo(a, b) {
    var i, rest = [];
    if (arguments.length > 2) {
        for (i = 2; i<arguments.length; i++) {
            rest.push(arguments[i]);
        }
    }
    console.log('a = ' + a);
    console.log('b = ' + b);
    console.log(rest);
}
```

为了获取除了已定义参数`a`、`b`之外的参数，我们不得不用`arguments`，并且循环要从索引`2`开始以便排除前两个参数，这种写法很别扭，只是为了获得额外的`rest`参数，有没有更好的方法？

ES6标准引入了rest参数，上面的函数可以改写为：

```
function foo(a, b, ...rest) {
    console.log('a = ' + a);
    console.log('b = ' + b);
    console.log(rest);
}

foo(1, 2, 3, 4, 5);
// 结果:
// a = 1
// b = 2
// Array [ 3, 4, 5 ]

foo(1);
// 结果:
// a = 1
// b = undefined
// Array []
```

rest参数只能写在最后，前面用`...`标识，从运行结果可知，传入的参数先绑定`a`、`b`，多余的参数以数组形式交给变量`rest`，所以，不再需要`arguments`我们就获取了全部参数。

3.2、变量作用域

> 变量的作用域

在JavaScript中，用`var`申明的变量实际上是有作用域的。

如果一个变量在函数体内部申明，则该变量的作用域为整个函数体，在函数体外不可引用该变量：

```
'use strict';

function foo() {
    var x = 1;
    x = x + 1;
}

x = x + 2; // ReferenceError! 无法在函数体外引用变量x
```

如果两个不同的函数各自申明了同一个变量，那么该变量只在各自的函数体内起作用。换句话说，不同函数内部的同名变量互相独立，互不影响：

```
'use strict';

function foo() {
    var x = 1;
    x = x + 1;
}

function bar() {
    var x = 'A';
    x = x + 'B';
}
```

由于JavaScript的函数可以嵌套，此时，内部函数可以访问外部函数定义的变量，反过来则不行：

```
'use strict';

function foo() {
    var x = 1;
    function bar() {
        var y = x + 1; // bar可以访问foo的变量x!
    }
    var z = y + 1; // ReferenceError! foo不可以访问bar的变量y!
}
```

如果内部函数和外部函数的变量名重名怎么办？来测试一下：

```
function foo() {
    var x = 1;
    function bar() {
        var x = 'A';
        console.log('x in bar() = ' + x); // 'A'
    }
    console.log('x in foo() = ' + x); // 1
    bar();
}

foo();
```

这说明JavaScript的函数在查找变量时从自身函数定义开始，从“内”向“外”查找。如果内部函数定义了与外部函数重名的变量，则内部函数的变量将“屏蔽”外部函数的变量。

> 变量提升

JavaScript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：

```
'use strict';

function foo() {
    var x = 'Hello, ' + y;
    console.log(x);
    var y = 'Bob';
}

foo();
```

结果：Hello, undefined ， 说明 y 的值为 undefined；

这正是因为JavaScript引擎自动提升了变量`y`的声明，但不会提升变量`y`的赋值。

对于上述`foo()`函数，JavaScript引擎看到的代码相当于：

```
function foo() {
    var y; // 提升变量y的申明，此时y为undefined
    var x = 'Hello, ' + y;
    console.log(x);
    y = 'Bob';
}
```

由于JavaScript的这一怪异的“特性”，我们在函数内部定义变量时，请严格遵守“在函数内部首先申明所有变量”这一规则。最常见的做法是用一个`var`申明函数内部用到的所有变量：

```
function foo() {
    var
        x = 1, // x初始化为1
        y = x + 1, // y初始化为2
        z, i; // z和i为undefined
    // 其他语句:
    for (i=0; i<100; i++) {
        ...
    }
}
```

> 全局作用域

不在任何函数内定义的变量就具有全局作用域。实际上，JavaScript默认有一个全局对象`window`，全局作用域的变量实际上被绑定到`window`的一个属性：

```
'use strict';

var course = 'Learn JavaScript';
alert(course); // 'Learn JavaScript'
alert(window.course); // 'Learn JavaScript
```

因此，直接访问全局变量`course`和访问`window.course`是完全一样的。

因此，顶层函数的定义也被视为一个全局变量，并绑定到`window`对象：

```
'use strict';

function foo() {
    alert('foo');
}

foo(); // 直接调用foo()
window.foo(); // 通过window.foo()调用
```

进一步大胆地猜测，我们每次直接调用的`alert()`函数其实也是`window`的一个变量：

```
'use strict';

window.alert('调用window.alert()');
// 把alert保存到另一个变量:
var old_alert = window.alert;
// 给alert赋一个新函数:
window.alert = function () {}
alert('无法用alert()显示了!');
// 恢复alert:
window.alert = old_alert;
alert('又可以用alert()了!');
```

这说明JavaScript实际上只有一个全局作用域。任何变量（函数也视为变量），如果没有在当前函数作用域中找到，就会继续往上查找，最后如果在全局作用域中也没有找到，则报`ReferenceError`错误。

全局变量会绑定到`window`上，不同的JavaScript文件如果使用了相同的全局变量，或者定义了相同名字的顶层函数，都会造成命名冲突，并且很难被发现。减少冲突的一个方法是把自己的所有变量和函数全部绑定到一个全局变量中。例如：

```
// 唯一的全局变量MYAPP:
var MYAPP = {};

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

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

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

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

> 局部作用域

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

```
'use strict';

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

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

```
'use strict';

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

> 常量

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

```
var PI = 3.14;
```

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

```
'use strict';

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

3.3、方法

> 定义方法

在一个对象中绑定函数，称为这个对象的方法。

在JavaScript中，对象的定义是这样的：

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

但是，如果我们给`xiaoming`绑定一个函数，就可以做更多的事情。比如，写个`age()`方法，返回`xiaoming`的年龄：

```
var xiaoming = {
    name: '小明',
    birth: 1990,
    age: function () {
        var y = new Date().getFullYear();
        return y - this.birth;
    }
};

xiaoming.age; // function xiaoming.age()
xiaoming.age(); // 今年调用是25,明年调用就变成26了
```

绑定到对象上的函数称为方法，和普通函数也没啥区别，但是它在内部使用了一个`this`关键字，这个东东是什么？

在一个方法内部，`this`是一个特殊变量，它始终指向当前对象，也就是`xiaoming`这个变量。所以，`this.birth`可以拿到`xiaoming`的`birth`属性。

让我们拆开写：

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

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

xiaoming.age(); // 25, 正常结果
getAge(); // NaN
```

单独调用函数`getAge()`怎么返回了`NaN`？*请注意*，我们已经进入到了JavaScript的一个大坑里。

JavaScript的函数内部如果调用了`this`，那么这个`this`到底指向谁？

答案是，视情况而定！

如果以对象的方法形式调用，比如`xiaoming.age()`，该函数的`this`指向被调用的对象，也就是`xiaoming`，这是符合我们预期的。

如果单独调用函数，比如`getAge()`，此时，该函数的`this`指向全局对象，也就是`window`。

> apply

不过，我们还是可以控制`this`的指向的！

要指定函数的`this`指向哪个对象，可以用函数本身的`apply`方法，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。

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

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

xiaoming.age(); // 25
getAge.apply(xiaoming, []); // 25, this指向xiaoming, 参数为空
```

4、标准对象

4.1、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'
```

4.2、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。

时区

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

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

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

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

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

要获取当前时间戳，可以用：

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

4.3、JSON

- JSON(JavaScript Object Notation, JS 对象标记) 是一种轻量级的数据交换格式，目前使用特别广泛。
- 采用完全独立于编程语言的**文本格式**来存储和表示数据。
- 简洁和清晰的层次结构使得 JSON 成为理想的数据交换语言。
- 易于人阅读和编写，同时也易于机器解析和生成，并有效地提升网络传输效率。

在 JavaScript 语言中，一切都是对象。因此，任何JavaScript 支持的类型都可以通过 JSON 来表示，例如字符串、数字、对象、数组等。看看他的要求和语法格式：

- 对象表示为键值对，数据由逗号分隔
- 花括号保存对象
- 方括号保存数组

JSON 键值对是用来保存 JavaScript 对象的一种方式，和 JavaScript 对象的写法也大同小异，键/值对组合中的键名写在前面并用双引号 "" 包裹，使用冒号 : 分隔，然后紧接着值：

```
{"name": "QinJiang"}
{"age": "3"}
{"sex": "男"}
```

很多人搞不清楚 JSON 和 JavaScript 对象的关系，甚至连谁是谁都不清楚。其实，可以这么理解：

- JSON 是 JavaScript 对象的字符串表示法，它使用文本表示一个 JS 对象的信息，本质是一个字符串。

```
var obj = {a: 'Hello', b: 'World'}; //这是一个对象，注意键名也是可以使用引号包裹的
var json = '{"a": "Hello", "b": "World"}'; //这是一个 JSON 字符串，本质是一个字符串
```

JSON 和 JavaScript 对象互转

- 要实现从JSON字符串转换为JavaScript 对象，使用 JSON.parse() 方法：

```
var obj = JSON.parse('{"a": "Hello", "b": "World"}');
//结果是 {a: 'Hello', b: 'World'}
```

- 要实现从JavaScript 对象转换为JSON字符串，使用 JSON.stringify() 方法：

```
var json = JSON.stringify({a: 'Hello', b: 'World'});
//结果是 '{"a": "Hello", "b": "World"}'
```

代码测试

1. 新建一个 json-1.html ， 编写测试内容

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<title>JSON_秦疆</title>
</head>
<body>

<script type="text/javascript">
//编写一个js的对象
var user = {
 name:"秦疆",
 age:3,
 sex:"男"
};
//将js对象转换成json字符串
var str = JSON.stringify(user);
console.log(str);

//将json字符串转换为js对象
var user2 = JSON.parse(str);
console.log(user2.age,user2.name,user2.sex);

</script>

</body>
</html>
```

2. 在IDEA中使用浏览器打开，查看控制台输出！

5、面向对象编程

> 面向对象编程

JavaScript的面向对象编程和大多数其他语言如Java、C#的面向对象编程都不太一样。如果你熟悉Java或C#，很好，你一定明白面向对象的两个基本概念：

1. 类：类是对象的类型模板，例如，定义`Student`类来表示学生，类本身是一种类型，`Student`表示学生类型，但不表示任何具体的某个学生；
2. 实例：实例是根据类创建的对象，例如，根据`Student`类可以创建出`xiaoming`、`xiaohong`、`xiaojun`等多个实例，每个实例表示一个具体的学生，他们全都属于`Student`类型。

所以，类和实例是大多数面向对象编程语言的基本概念。

不过，在JavaScript中，这个概念需要改一改。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。

原型是指当我们想要创建`xiaoming`这个具体的学生时，我们并没有一个`Student`类型可用。那怎么办？恰好有这么一个现成的对象：

```
var robot = {
    name: 'Robot',
    height: 1.6,
    run: function () {
        console.log(this.name + ' is running...');
    }
};
```

我们看这个`robot`对象有名字，有身高，还会跑，有点像小明，干脆就根据它来“创建”小明得了！

于是我们把它改名为`Student`，然后创建出`xiaoming`：

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

var xiaoming = {
    name: '小明'
};

xiaoming.__proto__ = Student;
```

注意最后一行代码把`xiaoming`的原型指向了对象`Student`，看上去`xiaoming`仿佛是从`Student`继承下来的：

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

`xiaoming`有自己的`name`属性，但并没有定义`run()`方法。不过，由于小明是从`Student`继承而来，只要`Student`有`run()`方法，`xiaoming`也可以调用：

JavaScript的原型链和Java的Class区别就在，它没有“Class”的概念，所有对象都是实例，所谓继承关系不过是把一个对象的原型指向另一个对象而已。

如果你把`xiaoming`的原型指向其他对象：

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

xiaoming.__proto__ = Bird;
```

现在`xiaoming`已经无法`run()`了，他已经变成了一只鸟：

```
xiaoming.fly(); // 小明 is flying...
```

在JavaScrip代码运行时期，你可以把`xiaoming`从`Student`变成`Bird`，或者变成任何对象。

> class 继承

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;
}

// 现在要给这个Student新增一个方法

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);
    }
}
```

6、操作BOM

6.1、浏览器

由于JavaScript的出现就是为了能在浏览器中运行，所以，浏览器自然是JavaScript开发者必须要关注的。

目前主流的浏览器分这么几种：

- IE 6~11：国内用得最多的IE浏览器，历来对W3C标准支持差。从IE10开始支持ES6标准；
- Chrome：Google出品的基于Webkit内核浏览器，内置了非常强悍的JavaScript引擎——V8。由于Chrome一经安装就时刻保持自升级，所以不用管它的版本，最新版早就支持ES6了；
- Safari：Apple的Mac系统自带的基于Webkit内核的浏览器，从OS X 10.7 Lion自带的6.1版本开始支持ES6，目前最新的OS X 10.11 El Capitan自带的Safari版本是9.x，早已支持ES6；
- Firefox：Mozilla自己研制的Gecko内核和JavaScript引擎OdinMonkey。早期的Firefox按版本发布，后来终于聪明地学习Chrome的做法进行自升级，时刻保持最新；
- 移动设备上目前iOS和Android两大阵营分别主要使用Apple的Safari和Google的Chrome，由于两者都是Webkit核心，结果HTML5首先在手机上全面普及（桌面绝对是Microsoft拖了后腿），对JavaScript的标准支持也很好，最新版本均支持ES6。

其他浏览器如Opera等由于市场份额太小就被自动忽略了。

另外还要注意识别各种国产浏览器，如某某安全浏览器，某某旋风浏览器，它们只是做了一个壳，其核心调用的是IE，也有号称同时支持IE和Webkit的“双核”浏览器。

不同的浏览器对JavaScript支持的差异主要是，有些API的接口不一样，比如AJAX，File接口。对于ES6标准，不同的浏览器对各个特性支持也不一样。

在编写JavaScript的时候，就要充分考虑到浏览器的差异，尽量让同一份JavaScript代码能运行在不同的浏览器中。

JavaScript可以获取浏览器提供的很多对象，并进行操作。

6.2、window

`window`对象不但充当全局作用域，而且表示浏览器窗口。

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

兼容性：IE<=8不支持。

```
'use strict';

// 可以调整浏览器窗口大小试试:
console.log('window inner size: ' + window.innerWidth + ' x ' +
window.innerHeight);
```

对应的，还有一个`outerWidth`和`outerHeight`属性，可以获取浏览器窗口的整个宽高。

6.3、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。

```
'use strict';

console.log('appName = ' + navigator.appName);
console.log('appVersion = ' + navigator.appVersion);
console.log('language = ' + navigator.language);
console.log('platform = ' + navigator.platform);
console.log('userAgent = ' + navigator.userAgent);
```

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

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

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

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

6.4、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。

```
console.log('Screen size = ' + screen.width + ' x ' + screen.height);
```

6.5、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。

```
location.reload();
location.assign('https://blog.kuangstudy.com/'); // 设置一个新的URL地址
```

6.6、document

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

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

```
document.title = '狂神说Java!';
```

请观察浏览器窗口标题的变化。

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="code-menu" style="border:solid 1px #ccc;padding:6px;">
    <dt>Java</dt>
    <dd>Spring</dd>
    <dt>Python</dt>
    <dd>Django</dd>
    <dt>Linux</dt>
    <dd>Docker</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('code-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);
```

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

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

Cookie还可以存储网站的一些设置，例如，页面显示的语言等等。

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

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

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

```
<!--www.example.com-->
<html>
    <head>
        <script src="http://www.foo.com/jquery.js"></script>
    </head>
    ...
</html>
```

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

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

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

6.7、history

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

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

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

任何情况，你都不应该使用`history`这个对象了。

7、操作DOM

7.1、选择器

由于HTML文档被浏览器解析后就是一棵DOM树，要改变HTML的结构，就需要通过JavaScript来操作DOM。

始终记住DOM是一个树形结构。操作一个DOM节点实际上就是这么几个操作：

- 更新：更新该DOM节点的内容，相当于更新了该DOM节点表示的HTML的内容；
- 遍历：遍历该DOM节点下的子节点，以便进行进一步操作；
- 添加：在该DOM节点下新增一个子节点，相当于动态增加了一个HTML节点；
- 删除：将该节点从HTML中删除，相当于删掉了该DOM节点的内容以及它包含的所有子节点。

在操作一个DOM节点前，我们需要通过各种方式先拿到这个DOM节点。最常用的方法是`document.getElementById()`和`document.getElementsByTagName()`，以及CSS选择器`document.getElementsByClassName()`。

由于ID在HTML文档中是唯一的，所以`document.getElementById()`可以直接定位唯一的一个DOM节点。`document.getElementsByTagName()`和`document.getElementsByClassName()`总是返回一组DOM节点。要精确地选择DOM，可以先定位父节点，再从父节点开始选择，以缩小范围。

例如：

```
// 返回ID为'test'的节点：
var test = document.getElementById('test');

// 先定位ID为'test-table'的节点，再返回其内部所有tr节点：
var trs = document.getElementById('test-table').getElementsByTagName('tr');

// 先定位ID为'test-div'的节点，再返回其内部所有class包含red的节点：
var reds = document.getElementById('test-
div').getElementsByClassName('red');

// 获取节点test下的所有直属子节点：
var cs = test.children;

// 获取节点test下第一个、最后一个子节点：
var first = test.firstElementChild;
var last = test.lastElementChild;
```

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，更加方便：

```
// 通过querySelector获取ID为q1的节点：
var q1 = document.querySelector('#q1');

// 通过querySelectorAll获取q1节点内的符合条件的所有节点：
var ps = q1.querySelectorAll('div.highlighted > p');
```

注意：低版本的IE<8不支持`querySelector`和`querySelectorAll`。IE8仅有限支持。

7.2、更新DOM

拿到一个DOM节点后，我们可以对它进行更新。

可以直接修改节点的文本，方法有两种：

一种是修改`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
```

用`innerHTML`时要注意，是否需要写入HTML。如果写入的字符串是通过网络拿到了，要注意对字符编码来避免XSS攻击。

第二种是修改`innerText`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>
```

修改CSS也是经常需要的操作。DOM节点的`style`属性对应所有的CSS，可以直接获取或设置。因为CSS允许`font-size`这样的名称，但它并非JavaScript有效的属性名，所以需要在JavaScript中改写为驼峰式命名`fontSize`：

```
// 获取<p id="p-id">...</p>
var p = document.getElementById('p-id');
// 设置CSS:
p.style.color = '#ff0000';
p.style.fontSize = '20px';
p.style.paddingTop = '2em';
```

7.3、插入DOM

> appendChild

当我们获得了某个DOM节点，想在这个DOM节点内插入新的DOM，应该如何做？

如果这个DOM节点是空的，例如，`，那么，直接使用innerHTML = 'child'`就可以修改DOM节点的内容，相当于“插入”了新的DOM节点。

如果这个DOM节点不是空的，那就不能这么做，因为`innerHTML`会直接替换掉原来的所有子节点。

有两个办法可以插入新的节点。一个是使用`appendChild`，把一个子节点添加到父节点的最后一个子节点。例如：

```
<!-- 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>
```

把`JavaScript`添加到的最后一项：

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

现在，HTML结构变成了这样：

```
<!-- 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。

更多的时候我们会从零创建一个新的节点，然后插入到指定位置：

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

这样我们就动态添加了一个新的节点：

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

动态创建一个节点然后添加到DOM树中，可以实现很多功能。举个例子，下面的代码动态创建了一个`节点，然后把它添加到`节点的末尾，这样就动态地给文档添加了新的CSS定义：

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

可以在Chrome的控制台执行上述代码，观察页面样式的变化。

> insertBefore

如果我们要把子节点插入到指定的位置怎么办？可以使用`parentElement.insertBefore(newElement, referenceElement);`，子节点会插入到`referenceElement`之前。

还是以上面的HTML为例，假定我们要把`Haskell`插入到`Python`之前：

```
<!-- 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>
```

7.4、删除DOM

删除一个DOM节点就比插入要容易得多。

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
```

注意到删除后的节点虽然不在文档树中了，但其实它还在内存中，可以随时再次被添加到别的位置。

当你遍历一个父节点的子节点并进行删除操作时，要注意，`children`属性是一个只读属性，并且它在子节点变化时会实时更新。

例如，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。

8、操作表单

8.1、回顾

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

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

HTML表单的输入控件主要有以下几种：

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

8.2、获取值

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

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

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

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

8.3、设置值

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

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

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

8.4、提交表单

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

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

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

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

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

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

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

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

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

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

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

<script src="https://cdn.bootcss.com/blueimp-md5/2.10.0/js/md5.min.js">
</script>

<script>
function checkForm() {
    var pwd = document.getElementById('password');
    // 把用户输入的明文变为MD5:
    pwd.value = md5(pwd.value);
    // 继续下一步:
    return true;
}
</script>
```

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

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

```
<script src="https://cdn.bootcss.com/blueimp-md5/2.10.0/js/md5.min.js">
</script>

<!-- HTML -->
<form id="login-form" method="post" onsubmit="return checkForm()">
```

```
    <input type="text" id="username" name="username">
    <input type="password" id="input-password">
    <input type="hidden" id="md5-password" name="password">
    <button type="submit">Submit</button>
</form>

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

9、jQuery

9.1、什么是JQuery

你可能听说过jQuery，它名字起得很土，但却是JavaScript世界中使用最广泛的一个库。

江湖传言，全世界大约有80~90%的网站直接或间接地使用了jQuery。鉴于它如此流行，又如此好用，所以每一个入门JavaScript的前端工程师都应该了解和学习它。

jQuery这么流行，肯定是因为它解决了一些很重要的问题。实际上，jQuery能帮我们干这些事情：

- 消除浏览器差异：你不需要自己写冗长的代码来针对不同的浏览器来绑定事件，编写AJAX等代码；
- 简洁的操作DOM的方法：写`$('#test')`肯定比`document.getElementById('test')`来得简洁；
- 轻松实现动画、修改CSS等各种操作。

jQuery的理念“Write Less, Do More”，让你写更少的代码，完成更多的工作！

官网：https://jquery.com/

jQuery只是一个`jquery-xxx.js`文件，但你会看到有compressed（已压缩）和uncompressed（未压缩）两种版本，使用时完全一样，但如果你想深入研究jQuery源码，那就用uncompressed版本。

使用jQuery只需要在页面的`head`引入jQuery文件即可：

```
<html>
<head>
    <script src="//code.jquery.com/jquery-1.11.3.min.js"></script>
    ...
</head>
<body>
<a id="test-link" href="#0">点我试试</a>

<script>
    // 获取超链接的jQuery对象:
    var a = $('#test-link');
    a.on('click', function () {
        alert('Hello!');
    });
```

```
    // 方式二
    a.click(function () {
        alert('Hello!');
    });
</script>
</body>
</html>
```

公式：`$(selector).action()`

- 美元符号定义 jQuery
- 选择符（selector）"查询"和"查找" HTML 元素
- jQuery 的 action() 执行对元素的操作

9.2、选择器

> 了解

选择器是jQuery的核心。

为什么jQuery要发明选择器？回顾一下DOM操作中我们经常使用的代码：

```
// 按ID查找：
var a = document.getElementById('dom-id');

// 按tag查找：
var divs = document.getElementsByTagName('div');

// 查找<p class="red">：
var ps = document.getElementsByTagName('p');
// 过滤出class="red":
// TODO:

// 查找<table class="green">里面的所有<tr>：
var table = ...
for (var i=0; i<table.children; i++) {
    // TODO: 过滤出<tr>
}
```

这些代码实在太繁琐了！

jQuery的选择器就是帮助我们快速定位到一个或多个DOM节点。

按ID查找

如果某个DOM节点有`id`属性，利用jQuery查找如下：

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

按tag查找

按tag查找只需要写上tag名称就可以了：

```
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

按class查找注意在class名称前加一个`.`：

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

按属性查找

一个DOM节点除了`id`和`class`外还可以有很多属性，很多时候按属性查找会非常方便，比如在一个表单中按属性来查找：

```
var email = $('[name=email]'); // 找出<??? name="email">
var passwordInput = $('[type=password]'); // 找出<??? type="password">
var a = $('[items="A B"]'); // 找出<??? items="A B">
```

当属性的值包含空格等特殊字符时，需要用双引号括起来。

按属性查找还可以使用前缀查找或者后缀查找：

```
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"
```

9.3、操作DOM

> 修改Text和HTML

jQuery对象的`text()`和`html()`方法分别获取节点的文本和原始HTML文本，例如，如下的HTML结构：

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

分别获取文本和HTML：

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

如何设置文本或HTML？jQuery的API设计非常巧妙：无参数调用`text()`是获取文本，传入参数就变成设置文本，HTML也是类似操作，自己动手试试：

```
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'); // 是不是两个节点都变成了JS?
```

修改CSS

jQuery对象有“批量操作”的特点，这用于修改CSS实在是太方便了。考虑下面的HTML结构：

```
<!-- HTML结构 -->
<ul id="test-css">
    <li class="lang dy"><span>JavaScript</span></li>
    <li class="lang"><span>Java</span></li>
    <li class="lang dy"><span>Python</span></li>
    <li class="lang"><span>Swift</span></li>
    <li class="lang dy"><span>Scheme</span></li>
</ul>
```

要高亮显示动态语言，调用jQuery对象的`css('name', 'value')`方法，我们用一行语句实现：

```
$('#test-css li.dy>span').css('background-color', '#ffd351').css('color',
'red');
```

注意，jQuery对象的所有方法都返回一个jQuery对象（可能是新的也可能是自身），这样我们可以进行链式调用，非常方便。

jQuery对象的`css()`方法可以这么用：

```
var div = $('#test-div');
div.css('color'); // '#000033', 获取CSS属性
div.css('color', '#336699'); // 设置CSS属性
div.css('color', ''); // 清除CSS属性
```

`css()`方法将作用于DOM节点的`style`属性，具有最高优先级。如果要修改`class`属性，可以用jQuery提供的下列方法：

```
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
```

显示和隐藏DOM

要隐藏一个DOM，我们可以设置CSS的`display`属性为`none`，利用`css()`方法就可以实现。不过，要显示这个DOM就需要恢复原有的`display`属性，这就得先记下来原有的`display`属性到底是`block`还是`inline`还是别的值。

考虑到显示和隐藏DOM元素使用非常普遍，jQuery直接提供`show()`和`hide()`方法，我们不用关心它是如何修改`display`属性的，总之它能正常工作：

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

注意，隐藏DOM节点并未改变DOM树的结构，它只影响DOM节点的显示。这和删除DOM节点是不同的。

获取DOM信息

利用jQuery对象的若干方法，我们直接可以获取DOM的高宽等信息，而无需针对不同浏览器编写特定代码：

```
// 浏览器可视窗口大小:
$(window).width(); // 800
$(window).height(); // 600

// HTML文档大小:
$(document).width(); // 800
$(document).height(); // 3500

// 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
```

操作表单

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="">
    <select id="test-select" name="city">
        <option value="BJ" selected>Beijing</option>
        <option value="SH">Shanghai</option>
        <option value="SZ">Shenzhen</option>
    </select>
    <textarea id="test-textarea">Hello</textarea>
*/
var
    input = $('#test-input'),
```

```
    select = $('#test-select'),
    textarea = $('#test-textarea');

input.val(); // 'test'
input.val('abc@example.com'); // 文本框的内容已变为abc@example.com

select.val(); // 'BJ'
select.val('SH'); // 选择框已变为Shanghai

textarea.val(); // 'Hello'
textarea.val('Hi'); // 文本区域已更新为'Hi'
```

可见，一个`val()`就统一了各种输入框的取值和赋值的问题。

> 添加DOM

要添加新的DOM节点，除了通过jQuery的`html()`这种暴力方法外，还可以用`append()`方法，例如：

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

如何向列表新增一个语言？首先要拿到`<ul>`节点：

```
var ul = $('#test-div>ul');
```

然后，调用`append()`传入HTML片段：

```
ul.append('<li><span>Haskell</span></li>');
```

`append()`把DOM添加到最后，`prepend()`则把DOM添加到最前。

如果要把新节点插入到指定位置，例如，JavaScript和Python之间，那么，可以先定位到JavaScript，然后用`after()`方法：

```
var js = $('#test-div>ul>li:first-child');
js.after('<li><span>Lua</span></li>');
```

也就是说，同级节点可以用`after()`或者`before()`方法。

> 删除节点

要删除DOM节点，拿到jQuery对象后直接调用`remove()`方法就可以了。如果jQuery对象包含若干DOM节点，实际上可以一次删除多个DOM节点：

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

9.4、事件

jQuery能够绑定的事件主要包括：

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； hover：鼠标进入和退出时触发两个函数，相当于mouseenter加上mouseleave。

键盘事件

键盘事件仅作用在当前焦点的DOM上，通常是`和`。

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`、`或`的内容改变时触发； submit：当`提交时触发； ready：当页面被载入并且DOM树完成初始化后触发。

> 初始化事件

我们自己的初始化代码必须放到`document`对象的`ready`事件中，保证DOM已完成初始化：

```
<html>
<head>
    <script>
        $(document).on('ready', function () {
            $('#testForm).on('submit', function () {
                alert('submit!');
            });
        });
    </script>
</head>
<body>
    <form id="testForm">
        ...
    </form>
</body>
```

这样写就没有问题了。因为相关代码会在DOM树初始化后再执行。

由于`ready`事件使用非常普遍，所以可以这样简化：

```
$(document).ready(function () {
    // on('submit', function)也可以简化:
    $('#testForm).submit(function () {
        alert('submit!');
    });
});
```

甚至还可以再简化为：

```
$(function () {
    // init...
});
```

> 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：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
    <style>
        #testMouseMoveDiv{
            width: 300px;
            height: 300px;
            border: 1px solid black;
        }
    </style>
</head>
<body>

mousemove: <span id="testMouseMoveSpan"></span>

<div id="testMouseMoveDiv">
    在此区域移动鼠标试试
</div>

<script src="https://cdn.bootcss.com/jquery/3.4.1/jquery.js"></script>
<script>
    $(function () {
        $('#testMouseMoveDiv').mousemove(function (e) {
            $('#testMouseMoveSpan').text('pageX = ' + e.pageX + ', pageY = '
+ e.pageY);
        });
    });
</script>

</body>
</html>
```