

这是小白的零基础JavaScript全栈教程。

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。

等等，你会问道，现在有这么多在线JavaScript教程和各种从入门到精通的JavaScript书籍，为什么我要选择这个教程？

原因是，这个教程：

是JavaScript全栈教程！

可以在线免费学习！

可以在线编写JavaScript代码并直接运行！

不要再犹豫了，立刻从现在开始，零基础迈向全栈开发工程师！

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。

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。

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。

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。

如何编写JavaScript

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：

Sublime Text

免费，但不注册会不定时弹出提示框。

Notepad++

免费

注意：不可以用Word或写字板来编写JavaScript或HTML，因为带格式的文本保存后不是纯文本文件，无法被浏览器正常读取。

如何运行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。

不过，开始学习阶段，你无须关心如何搭建开发环境的问题，我们提供在页面输入JavaScript代码并直接运行的功能，让你专注于JavaScript的学习。

试试直接点击“Run”按钮执行下面的JavaScript代码：

```
// 以//开头直到行末的是注释，将被浏览器忽略
// 第一个JavaScript代码：
```

```
alert('Hello, world');
```

Run

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。

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。从这里[下载Chrome浏览器](#)。打开网页出问题的童鞋请移步[国内镜像](#)。

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“x”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。

练习

打开[新浪首页](#)，然后查看页面源代码，找一找引入的JavaScript文件和直接编写在页面中的JavaScript代码。然后在Chrome中打开开发者工具，在控制台输入 `console.log('Hello');`，回车查看JavaScript代码执行结果。

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`;`。

注意：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。

注释

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 */
```

练习：

分别利用行注释和块注释把下面的语句注释掉，使它不再执行：

// 请注释掉下面的语句：

```
alert('我不想执行');
alert('我也不想执行');
```

Run

大小写

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。

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',
  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`。

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。

来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';  
alert(abc);
```

Run

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。

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'
```

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``...``表示：

```
`这是一个  
多行  
字符串`;
```

练习：测试你的浏览器是否支持**ES6**标准，如果不支持，请把多行字符串用`\n`重新表示出来：

```
// 如果浏览器不支持ES6，将报SyntaxError错误：  
  
alert(`多行  
字符串  
测试`);
```

Run

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

`toUpperCase()` 把一个字符串全部变为大写：

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

toLowerCase

`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

`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

`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'
```

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。

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10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
```

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2个元素, 其中头两个元素又是一个 `Array` 。

练习: 如何通过索引取到 `500` 这个值:

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


Run

小结

Array 提供了一种顺序存储一组元素的功能，并可以按索引来读写。

练习：在新生欢迎会上，你已经拿到了新同学的名单，请排序后显示： 欢迎XXX，XXX，XXX和XXX同学！：

```
'use strict';  
var arr = ['小明', '小红', '大军', '阿黄'];  
  
alert('???');
```

Run

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。例如，为了描述“小明”这个淘气的小朋友，我们可以用若干键值对来描述他：

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

JavaScript用一个 `{...}` 表示一个对象，键值对以 `xxx: xxx` 形式申明，用 `,` 隔开。注意，最后一个键值对不需要在末尾加 `,`，如果加了，有的浏览器（如低版本的**IE**）将报错。

上述对象申明了一个 `name` 属性，值是 `'小明'`，`birth` 属性，值是 `1990`，以及其他一些属性。最后，把这个对象赋值给变量 `xiaoming` 后，就可以通过变量 `xiaoming` 来获取小明的属性了：

```
xiaoming.name; // '小明'
xiaoming.birth; // 1990
```

访问属性是通过 `.` 操作符完成的，但这要求属性名必须是一个有效的变量名。如果属性名包含特殊字符，就必须用 `''` 括起来：

```
var xiaohong = {
  name: '小红',
  'middle-school': 'No.1 Middle School'
};
```

`xiaohong` 的属性名 `middle-school` 不是一个有效的变量，就需要用 `''` 括起来。访问这个属性也无法使用 `.` 操作符，必须用 `['xxx']` 来访问：

```
xiaohong['middle-school']; // 'No.1 Middle School'
xiaohong['name']; // '小红'
xiaohong.name; // '小红'
```

也可以用 `xiaohong['name']` 来访问 `xiaohong` 的 `name` 属性，不过 `xiaohong.name` 的写法更简洁。我们在编写**JavaScript**代码的时候，属性名尽量使用标准的变量名，这样就可以直接通过 `object.prop` 的形式访问一个属性了。

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

如果访问一个不存在的属性会返回什么呢？**JavaScript**规定，访问不存在的属性不报错，而是返回 `undefined`：

```
var xiaoming = {
  name: '小明'
};
xiaoming.age; // undefined
```

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 `xiaoming` 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 `in` 判断一个属性存在，这个属性不一定是 `xiaoming` 的，它可能是 `xiaoming` 继承得到的：

```
'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
```

JavaScript使用 `if () { ... } else { ... }` 来进行条件判断。例如，根据年龄显示不同内容，可以用 `if` 语句实现如下：

```
var age = 20;
if (age >= 18) { // 如果age >= 18为true，则执行if语句块
  alert('adult');
} else { // 否则执行else语句块
  alert('teenager');
}
```

其中 `else` 语句是可选的。如果语句块只包含一条语句，那么可以省略 `{ }`：

```
var age = 20;
if (age >= 18)
  alert('adult');
else
  alert('teenager');
```

省略 `{ }` 的危险之处在于，如果后来想添加一些语句，却忘了写 `{ }`，就改变了 `if...else...` 的语义，例如：

```
var age = 20;
if (age >= 18)
    alert('adult');
else
    console.log('age < 18'); // 添加一行日志
    alert('teenager'); // <- 这行语句已经不在else的控制范围了
```

上述代码的 `else` 子句实际上只负责执行 `console.log('age < 18');`，原有的 `alert('teenager');` 已经不属于 `if...else...` 的控制范围了，它每次都会执行。

相反地，有 `{}` 的语句就不会出错：

```
var age = 20;
if (age >= 18) {
    alert('adult');
} else {
    console.log('age < 18');
    alert('teenager');
}
```

这就是为什么我们建议永远都要写上 `{}`。

多行条件判断

如果还要更细致地判断条件，可以使用多个 `if...else...` 的组合：

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

上述多个 `if...else...` 的组合实际上相当于两层 `if...else...`：

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

但是我们通常把 `else if` 连写在一起，来增加可读性。这里的 `else` 略掉了 `{}` 是没有问题的，因为它只包含一个 `if` 语句。注意最后一个单独的 `else` 不要略掉 `{}`。

请注意，`if...else...` 语句的执行特点是二选一，在多个 `if...else...` 语句中，如果某个条件成立，则后续就不再继续判断了。

试解释为什么下面的代码显示的是 `teenager`：

```
'use strict';
var age = 20;

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

Run

由于 `age` 的值为 `20`，它实际上同时满足条件 `age >= 6` 和 `age >= 18`，这说明条件判断的顺序非常重要。请修复后让其显示 `adult`。

如果 `if` 的条件判断语句结果不是 `true` 或 `false` 怎么办？例如：

```
var s = '123';
if (s.length) { // 条件计算结果为3
    //
}
```

JavaScript把 `null`、`undefined`、`0`、`NaN` 和空字符串 `''` 视为 `false`，其他值一概视为 `true`，因此上述代码条件判断的结果是 `true`。

练习

小明身高1.75，体重80.5kg。请根据BMI公式（体重除以身高的平方）帮小明计算他的BMI指数，并根据BMI指数：

- 低于18.5：过轻
- 18.5-25：正常
- 25-28：过重
- 28-32：肥胖
- 高于32：严重肥胖

用 `if...else...` 判断并显示结果：

```
'use strict';

var height = parseFloat(prompt('请输入身高(m):'));
var weight = parseFloat(prompt('请输入体重(kg):'));
```

```
var bmi = ???;
if ...
```

Run

循环

要计算**1+2+3**，我们可以直接写表达式：

```
1 + 2 + 3; // 6
```

要计算**1+2+3+...+10**，勉强也能写出来。

但是，要计算**1+2+3+...+10000**，直接写表达式就不可能了。

为了让计算机能计算成千上万次的重复运算，我们就需要循环语句。

JavaScript的循环有两种，一种是 `for` 循环，通过初始条件、结束条件和递增条件来循环执行语句块：

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

让我们来分析一下 `for` 循环的控制条件：

- **i=1** 这是初始条件，将变量*i*置为**1**；
- **i<=10000** 这是判断条件，满足时就继续循环，不满足就退出循环；
- **i++** 这是每次循环后的递增条件，由于每次循环后变量*i*都会加**1**，因此它终将在若干次循环后不满足判断条件 `i<=10000` 而退出循环。

练习

利用 `for` 循环计算 `1 * 2 * 3 * ... * 10` 的结果：

```
'use strict';
```

```
var x = ?;  
var i;  
for ...
```

```
if (x === 3628800) {  
    alert('1 x 2 x 3 x ... x 10 = ' + x);  
}  
else {  
    alert('计算错误');  
}
```

Run

`for` 循环最常用的地方是利用索引来遍历数组：

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

`for` 循环的3个条件都是可以省略的，如果没有退出循环的判断条件，就必须使用 `break` 语句退出循环，否则就是死循环：

```
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

`for` 循环的一个变体是 `for ... in` 循环，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) {
    alert(key); // 'name', 'age', 'city'
}
```

要过滤掉对象继承的属性，用 `hasOwnProperty()` 来实现：

```
var o = {
    name: 'Jack',
    age: 20,
    city: 'Beijing'
};
for (var key in o) {
    if (o.hasOwnProperty(key)) {
        alert(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，因此，`for ... in` 循环可以直接循环出 `Array` 的索引：

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

`for` 循环在已知循环的初始和结束条件时非常有用。而上述忽略了条件的 `for` 循环容易让人看不清循环的逻辑，此时用 `while` 循环更佳。

`while` 循环只有一个判断条件，条件满足，就不断循环，条件不满足时则退出循环。比如我们要计算100以内所有奇数之和，可以用 `while` 循环实现：

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

在循环内部变量 `n` 不断自减，直到变为 `-1` 时，不再满足 `while` 条件，循环退出。

do ... while

最后一种循环是 `do { ... } while()` 循环，它和 `while` 循环的唯一区别在于，不是在每次循环开始的时候判断条件，而是在每次循环完成的时候判断条件：

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

用 `do { ... } while()` 循环要小心，循环体会至少执行1次，而 `for` 和 `while` 循环则可能一次都不执行。

练习

请利用循环遍历数组中的每个名字，并显示 `Hello, xxx!`：

```
'use strict';
var arr = ['Bart', 'Lisa', 'Adam'];
```

Run

请尝试 `for` 循环和 `while` 循环，并以正序、倒序两种方式遍历。

小结

循环是让计算机做重复任务的有效的方法，有些时候，如果代码写得有问题，会让程序陷入“死循环”，也就是永远循环下去。**JavaScript**的死循环会让浏览器无法正常显示或执行当前页面的逻辑，有的浏览器会直接挂掉，有的浏览器会在一段时间后提示你强行终止**JavaScript**的执行，因此，要特别注意死循环的问题。

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。

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`。要测试你的浏览器是否支持**ES6**规范，请执行以下代码，如果浏览器报**ReferenceError**错误，那么你需要换一个支持**ES6**的浏览器：

```
'use strict';
var m = new Map();
var s = new Set();
alert('你的浏览器支持Map和Set! ');

// 直接运行测试
```

Run

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**。

要创建一个 `Set`，需要提供的一个 `Array` 作为输入，或者直接创建一个空 `Set`：

```
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
{1, 2, 3, 4}
>>> s.add(4)
>>> s
{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}
```

小结

`Map` 和 `Set` 是ES6标准新增的数据类型，请根据浏览器的支持情况决定是否要使用。

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` 和 `Set` 都属于 `iterable` 类型。

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

`for ... of` 循环是ES6引入的新的语法，请测试你的浏览器是否支持：

```
'use strict';
var a = [1, 2, 3];
for (var x of a) {
}
alert('你的浏览器支持for ... of');

// 请直接运行测试
```

Run

用 `for ... of` 循环遍历集合，用法如下：

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

你可能会有疑问，`for ... of` 循环和 `for ... in` 循环有何区别？

`for ... in` 循环由于历史遗留问题，它遍历的实际上是对对象的属性名称。一个 `Array` 数组实际上也是一个对象，它的每个元素的索引被视为一个属性。

当我们手动给 `Array` 对象添加了额外的属性后，`for ... in` 循环将带来意想不到的意外效果：

```
var a = ['A', 'B', 'C'];
a.name = 'Hello';
for (var x in a) {
    alert(x); // '0', '1', '2', 'name'
}
```

`for ... in` 循环将把 `name` 包括在内，但 `Array` 的 `length` 属性却不包括在内。

`for ... of` 循环则完全修复了这些问题，它只循环集合本身的元素：

```
var a = ['A', 'B', 'C'];
a.name = 'Hello';
for (var x of a) {
    alert(x); 'A', 'B', 'C'
}
```

这就是为什么要引入新的 `for ... of` 循环。

然而，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：

```
var a = ['A', 'B', 'C'];
a.forEach(function (element, index, array) {
    // element: 指向当前元素的值
    // index: 指向当前索引
    // array: 指向Array对象本身
    alert(element);
});
```

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

`Set` 与 `Array` 类似，但 `Set` 没有索引，因此回调函数最多两个参数：

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

如果对某些参数不感兴趣，由于JavaScript的函数调用不要求参数必须一致，因此可以忽略它们。例如，只需要获得 `Array` 的 `element`：

```
var a = ['A', 'B', 'C'];
a.forEach(function (element) {
    alert(element);
});
```

我们知道圆的面积计算公式为：

$$S = \pi r^2$$

当我们知道半径 `r` 的值时，就可以根据公式计算出面积。假设我们需要计算3个不同大小的圆的面积：

```
var r1 = 12.34;
var r2 = 9.08;
var r3 = 73.1;
var s1 = 3.14 * r1 * r1;
var s2 = 3.14 * r2 * r2;
var s3 = 3.14 * r3 * r3;
```

当代码出现有规律的重复的时候，你就需要当心了，每次写 `3.14 * x * x` 不仅很麻烦，而且，如果要把 `3.14` 改成 `3.14159265359` 的时候，得全部替换。

有了函数，我们就不再每次写 `s = 3.14 * x * x`，而是写成更有意义的函数调用 `s = area_of_circle(x)`，而函数 `area_of_circle` 本身只需要写一次，就可以多次调用。

基本上所有的高级语言都支持函数，JavaScript也不例外。JavaScript的函数不但是“头等公民”，而且可以像变量一样使用，具有非常强大的抽象能力。

抽象

抽象是数学中非常常见的概念。举个例子：

计算数列的和，比如：`1 + 2 + 3 + ... + 100`，写起来十分不方便，于是数学家发明了求和符号 \sum ，可以把 `1 + 2 + 3 + ... + 100` 记作：

$$\sum_{n=1}^{100} n$$

这种抽象记法非常强大，因为我们看到 \sum 就可以理解成求和，而不是还原成低级的加法运算。

而且，这种抽象记法是可扩展的，比如：

$$\sum_{n=1}^{100} (n^2+1)$$

还原成加法运算就变成了：

$$(1 \times 1 + 1) + (2 \times 2 + 1) + (3 \times 3 + 1) + \dots + (100 \times 100 + 1)$$

可见，借助抽象，我们才能不关心底层的具体计算过程，而直接在更高的层次上思考问题。

写计算机程序也是一样，函数就是最基本的一种代码抽象的方式。

定义函数

在JavaScript中，定义函数的方式如下：

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

上述 `abs()` 函数的定义如下：

- `function` 指出这是一个函数定义；
- `abs` 是函数的名称；
- `(x)` 括号内列出函数的参数，多个参数以 `,` 分隔；
- `{ ... }` 之间的代码是函数体，可以包含若干语句，甚至可以没有任何语句。

请注意，函数体内部的语句在执行时，一旦执行到 `return` 时，函数就执行完毕，并将结果返回。因此，函数内部通过条件判断和循环可以实现非常复杂的逻辑。

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`。

由于 **JavaScript** 的函数也是一个对象，上述定义的 `abs()` 函数实际上是一个函数对象，而函数名 `abs` 可以视为指向该函数的变量。

因此，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。

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;
  }
}
```

arguments

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

```
function foo(x) {
  alert(x); // 10
  for (var i=0; i<arguments.length; i++) {
    alert(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
  }
  // ...
}
```

要把中间的参数**b**变为“可选”参数，就只能通过 **arguments** 判断，然后重新调整参数并赋值。

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。

如果传入的参数连正常定义参数都没填满，也不要紧，**rest**参数会接收一个空数组（注意不是 `undefined`）。

因为**rest**参数是ES6新标准，所以你需要测试一下浏览器是否支持。请用**rest**参数编写一个 `sum()` 函数，接收任意个参数并返回它们的和：

```
'use strict';

function sum(...rest) {
    ???
}

// 测试：
var i, args = [];
for (i=1; i<=100; i++) {
    args.push(i);
}
if (sum() !== 0) {
    alert('测试失败: sum() = ' + sum());
} else if (sum(1) !== 1) {
    alert('测试失败: sum(1) = ' + sum(1));
} else if (sum(2, 3) !== 5) {
    alert('测试失败: sum(2, 3) = ' + sum(2, 3));
} else if (sum.apply(null, args) !== 5050) {
    alert('测试失败: sum(1, 2, 3, ..., 100) = ' + sum.apply(null, args));
} else {
    alert('测试通过!');
}
```

Run

小心你的**return**语句

前面我们讲到了JavaScript引擎有一个在行末自动添加分号的机制，这可能让你栽到**return**语句的一个大坑：

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

foo(); // { name: 'foo' }
```

如果把**return**语句拆成两行：

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

要小心了，由于JavaScript引擎在行末自动添加分号的机制，上面的代码实际上变成了：

```
function foo() {  
    return; // 自动添加了分号，相当于return undefined;  
    { name: 'foo' }; // 这行语句已经没法执行到了  
}
```

所以正确的多行写法是：

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

练习

定义一个计算圆面积的函数`area_of_circle()`，它有两个参数：

- `r`: 表示圆的半径；
- `pi`: 表示 π 的值，如果不传，则默认3.14

```
'use strict';
```

```
function area_of_circle(r, pi) {
```

```
    return 0;
```

```
}  
// 测试：  
if (area_of_circle(2) === 12.56 && area_of_circle(2, 3.1416) === 12.5664) {  
    alert('测试通过');  
} else {  
    alert('测试失败');  
}
```

Run

Max是一个JavaScript新手，他写了一个`max()`函数，返回两个数中较大的那个：

```
'use strict';

function max(a, b) {

    if (a > b) {
        return
            a;
    } else {
        return
            b;
    }

}

alert(max(15, 20));
```

Run

但是Max抱怨他的浏览器出问题了，无论传入什么数，`max()`函数总是返回`undefined`。请帮他指出问题并修复。

在JavaScript中，用`var`声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？

```
'use strict';

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

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，把所有声明的变量“提升”到函数顶部：

```
'use strict';

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

foo();
```

虽然是**strict**模式，但语句 `var x = 'Hello, ' + y;` 并不报错，原因是变量 `y` 在稍后申明了。但是 `alert` 显示 `Hello, undefined`，说明变量 `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
    var x = 'Hello, ' + y;
    alert(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。

你可能猜到了，由于函数定义有两种方式，以变量方式 `var foo = function () {}` 定义的函数实际上也是一个全局变量，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!');
```

Run

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。

名字空间

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

减少冲突的一个方法是把自己的所有变量和函数全部绑定到一个全局变量中。例如：

```
// 唯一的全局变量MYAPP:
var MYAPP = {};

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

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

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

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

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;
    }
    i += 1; // SyntaxError
}
```

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
```

在一个对象中绑定函数，称为这个方法。

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`。

坑爹啊！

更坑爹的是，如果这么写：

```
var fn = xiaoming.age; // 先拿到xiaoming的age函数
fn(); // NaN
```

也是不行的！要保证 `this` 指向正确，必须用 `obj.xxx()` 的形式调用！

由于这是一个巨大的设计错误，要想纠正可没那么简单。ECMA决定，在strict模式下让函数的 `this` 指向 `undefined`，因此，在strict模式下，你会得到一个错误：

```
'use strict';

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

var fn = xiaoming.age;
fn(); // Uncaught TypeError: Cannot read property 'birth' of undefined
```

这个决定只是让错误及时暴露出来，并没有解决 `this` 应该指向的正确位置。

有些时候，喜欢重构的你把方法重构了一下：

```
'use strict';

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

xiaoming.age(); // Uncaught TypeError: Cannot read property 'birth' of undefined
```

结果又报错了！原因是 `this` 指针只在 `age` 方法的函数内指向 `xiaoming`，在函数内部定义的函数，`this` 又指向 `undefined` 了！（在非strict模式下，它重新指向全局对象 `window` ！）

修复的办法也不是没有，我们用一个 `that` 变量首先捕获 `this`：

```
'use strict';

var xiaoming = {
  name: '小明',
  birth: 1990,
  age: function () {
    var that = this; // 在方法内部一开始就捕获this
    function getAgeFromBirth() {
      var y = new Date().getFullYear();
      return y - that.birth; // 用that而不是this
    }
    return getAgeFromBirth();
  }
};

xiaoming.age(); // 25
```

用 `var that = this;`，你就可以放心地在方法内部定义其他函数，而不是把所有语句都堆到一个方法中。

apply

虽然在一个独立的函数调用中，根据是否是`strict`模式，`this` 指向 `undefined` 或 `window`，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。

用 `apply` 修复 `getAge()` 调用：

```
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
```

另一个与 `apply()` 类似的方法是 `call()`，唯一区别是：

- `apply()` 把参数打包成 `Array` 再传入；
- `call()` 把参数按顺序传入。

比如调用 `Math.max(3, 5, 4)`，分别用 `apply()` 和 `call()` 实现如下：

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

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

对普通函数调用，我们通常把 `this` 绑定为 `null`。

装饰器

利用 `apply()`，我们还可以动态改变函数的行为。

JavaScript 的所有对象都是动态的，即使内置的函数，我们也可以重新指向新的函数。

现在假定我们想统计一下代码一共调用了多少次 `parseInt()`，可以把所有的调用都找出来，然后手动加上 `count += 1`，不过这样做太傻了。最佳方案是用我们自己的函数替换掉默认的 `parseInt()`：

```
var count = 0;
var oldParseInt = parseInt; // 保存原函数

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

// 测试：
parseInt('10');
parseInt('20');
parseInt('30');
count; // 3
```

高阶函数英文叫 **Higher-order function**。那么什么是高阶函数？

JavaScript 的函数其实都指向某个变量。既然变量可以指向函数，函数的参数能接收变量，那么一个函数就可以接收另一个函数作为参数，这种函数就称之为高阶函数。

一个最简单的高阶函数：

```
function add(x, y, f) {
    return f(x) + f(y);
}
```

当我们调用 `add(-5, 6, Math.abs)` 时，参数 `x`，`y` 和 `f` 分别接收 `-5`，`6` 和函数 `Math.abs`，根据函数定义，我们可以推导计算过程为：

```
x = -5;
y = 6;
f = Math.abs;
f(x) + f(y) ==> Math.abs(-5) + Math.abs(6) ==> 11;
return 11;
```

用代码验证一下：

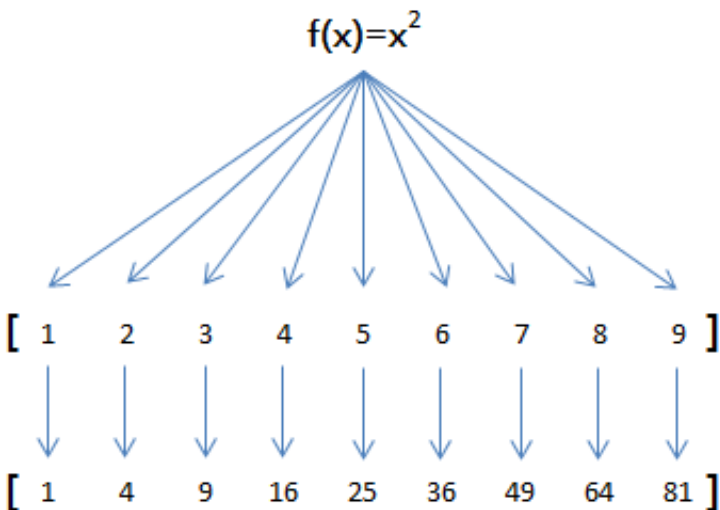
```
add(-5, 6, Math.abs); // 11
```

编写高阶函数，就是让函数的参数能够接收别的函数。

如果你读过Google的那篇大名鼎鼎的论文“[MapReduce: Simplified Data Processing on Large Clusters](#)”，你就能大概明白map/reduce的概念。

map

举例说明，比如我们有一个函数 $f(x)=x^2$ ，要把这个函数作用在一个数组`[1, 2, 3, 4, 5, 6, 7, 8, 9]`上，就可以用`map`实现如下：



由于`map()`方法定义在JavaScript的`Array`中，我们调用`Array`的`map()`方法，传入我们自己的函数，就得到了一个新的`Array`作为结果：

```
function pow(x) {  
    return x * x;  
}  
  
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9];  
arr.map(pow); // [1, 4, 9, 16, 25, 36, 49, 64, 81]
```

`map()`传入的参数是`pow`，即函数对象本身。

你可能会想，不需要`map()`，写一个循环，也可以计算出结果：

```
var f = function (x) {  
    return x * x;  
};  
  
var arr = [1, 2, 3, 4, 5, 6, 7, 8, 9];  
var result = [];  
for (var i=0; i<arr.length; i++) {  
    result.push(f(arr[i]));  
}
```

的确可以，但是，从上面的循环代码，我们无法一眼看明白“把 $f(x)$ 作用在`Array`的每一个元素并把结果生成一个新的`Array`”。

所以，`map()`作为高阶函数，事实上它把运算规则抽象了，因此，我们不但可以计算简单的 $f(x)=x^2$ ，还可以计算任意复杂的函数，比如，把`Array`的所有数字转为字符串：

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

只需要一行代码。

reduce

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

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

比方说对一个 `Array` 求和，就可以用 `reduce` 实现：

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

练习：利用 `reduce()` 求积：

```
'use strict';
```

```
function product(arr) {
```

```
    return 0;
```

```
}
```

```
// 测试：
```

```
if (product([1, 2, 3, 4]) === 24 && product([0, 1, 2]) === 0 && product([99, 88, 77, 66]) === 44) {
    alert('测试通过!');
}
```

```
else {
    alert('测试失败!');
}
```

Run

要把 `[1, 3, 5, 7, 9]` 变换成整数13579，`reduce()` 也能派上用场：

```
var arr = [1, 3, 5, 7, 9];
arr.reduce(function (x, y) {
    return x * 10 + y;
}); // 13579
```

如果我们继续改进这个例子，想办法把一个字符串 `13579` 先变成 `Array` —— `[1, 3, 5, 7, 9]`，再利用 `reduce()` 就可以写出一个把字符串转换为 `Number` 的函数。

练习：不要使用 `JavaScript` 内置的 `parseInt()` 函数，利用 `map` 和 `reduce` 操作实现一个 `string2int()` 函数：

```
'use strict';

function string2int(s) {

    return 0;

}

// 测试：
if (string2int('0') === 0 && string2int('12345') === 12345 && string2int('12300') === 12300) {
    if (string2int.toString().indexOf('parseInt') !== -1) {
        alert('请勿使用parseInt()!');
    } else if (string2int.toString().indexOf('Number') !== -1) {
        alert('请勿使用Number()!');
    } else {
        alert('测试通过!');
    }
}
else {
    alert('测试失败!');
}
```

Run

练习

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

```
'use strict';

function normalize(arr) {
```

```
return [];
```

```
}

// 测试:
if (normalize(['adam', 'LISA', 'barT']).toString() === ['Adam', 'Lisa', 'Bart'].toString()) {
    alert('测试通过!');
}
else {
    alert('测试失败!');
}
```

Run

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

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

例如，在一个 `Array` 中，删掉偶数，只保留奇数，可以这么写：

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

把一个 `Array` 中的空字符串删掉，可以这么写：

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

可见用 `filter()` 这个高阶函数，关键在于正确实现一个“筛选”函数。

练习

请尝试用 `filter()` 筛选出素数：

```
'use strict';

function get_primes(arr) {
```



```
return [];
```

```
}

// 测试:
var
    x,
    r,
    arr = [];
for (x = 1; x < 100; x++) {
    arr.push(x);
}
r = get_primes(arr);
if (r.toString() === [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71]) {
    alert('测试通过!');
} else {
    alert('测试失败: ' + r.toString());
}
```

Run

排序算法

排序也是在程序中经常用到的算法。无论使用冒泡排序还是快速排序，排序的核心是比较两个元素的大小。如果是数字，我们可以直接比较，但如果是字符串或者两个对象呢？直接比较数学上的大小是没有意义的，因此，比较的过程必须通过函数抽象出来。通常规定，对于两个元素 `x` 和 `y`，如果认为 `x < y`，则返回 `-1`，如果认为 `x == y`，则返回 `0`，如果认为 `x > y`，则返回 `1`，这样，排序算法就不用关心具体的比较过程，而是根据比较结果直接排序。

JavaScript的 `Array` 的 `sort()` 方法就是用于排序的，但是排序结果可能让你大吃一惊：

```
// 看上去正常的结果:
['Google', 'Apple', 'Microsoft'].sort(); // ['Apple', 'Google', 'Microsoft'];

// apple排在了最后:
['Google', 'apple', 'Microsoft'].sort(); // ['Google', 'Microsoft', 'apple']

// 无法理解的结果:
[10, 20, 1, 2].sort(); // [1, 10, 2, 20]
```

第二个排序把 `apple` 排在了最后，是因为字符串根据 **ASCII** 码进行排序，而小写字母 `a` 的 **ASCII** 码在大写字母之后。

第三个排序结果是什么鬼？简单的数字排序都能错？

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

坑爹啊！

如果不知道 `sort()` 方法的默认排序规则，直接对数字排序，绝对栽进坑里！

幸运的是，`sort()` 方法也是一个高阶函数，它还可以接收一个比较函数来实现自定义的排序。

要按数字大小排序，我们可以这么写：

```
var arr = [10, 20, 1, 2];
arr.sort(function (x, y) {
  if (x < y) {
    return -1;
  }
  if (x > y) {
    return 1;
  }
  return 0;
}); // [1, 2, 10, 20]
```

如果要倒序排序，我们可以把大的数放前面：

```
var arr = [10, 20, 1, 2];
arr.sort(function (x, y) {
  if (x < y) {
    return 1;
  }
  if (x > y) {
    return -1;
  }
  return 0;
}); // [20, 10, 2, 1]
```

默认情况下，对字符串排序，是按照 `ASCII` 的大小比较的，现在，我们提出排序应该忽略大小写，按照字母序排序。要实现这个算法，不必对现有代码大加改动，只要我们能定义出忽略大小写的比较算法就可以：

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

忽略大小写来比较两个字符串，实际上就是先把字符串都变成大写（或者都变成小写），再比较。

从上述例子可以看出，高阶函数的抽象能力是非常强大的，而且，核心代码可以保持得非常简洁。

最后友情提示，`sort()` 方法会直接对 `Array` 进行修改，它返回的结果仍是当前 `Array`：

```
var a1 = ['B', 'A', 'C'];
var a2 = a1.sort();
a1; // ['A', 'B', 'C']
a2; // ['A', 'B', 'C']
a1 === a2; // true, a1和a2是同一对象
```

函数作为返回值

高阶函数除了可以接受函数作为参数外，还可以把函数作为结果值返回。

我们来实现一个对 `Array` 的求和。通常情况下，求和的函数是这样定义的：

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

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

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

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

当我们调用 `lazy_sum()` 时，返回的并不是求和结果，而是求和函数：

```
var f = lazy_sum([1, 2, 3, 4, 5]); // function sum()
```

调用函数 `f` 时，才真正计算求和的结果：

```
f(); // 15
```

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

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

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

`f1()` 和 `f2()` 的调用结果互不影响。

闭包

注意到返回的函数在其定义内部引用了局部变量 `arr`，所以，当一个函数返回了一个函数后，其内部的局部变量还被新函数引用，所以，闭包用起来简单，实现起来可不容易。

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

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

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

在上面的例子中，每次循环，都创建了一个新的函数，然后，把创建的3个函数都添加到一个 `Array` 中返回了。

你可能认为调用 `f1()`，`f2()` 和 `f3()` 结果应该是 `1`，`4`，`9`，但实际结果是：

```
f1(); // 16
f2(); // 16
f3(); // 16
```

全部都是 `16`！原因就在于返回的函数引用了变量 `i`，但它并非立刻执行。等到3个函数都返回时，它们所引用的变量 `i` 已经变成了 `4`，因此最终结果为 `16`。

返回闭包时牢记的一点就是：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

如果一定要引用循环变量怎么办？方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变：

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

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

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

注意这里用了一个“创建一个匿名函数并立刻执行”的语法：

```
(function (x) {
    return x * x;
})(3); // 9
```

理论上讲，创建一个匿名函数并立刻执行可以这么写：

```
function (x) { return x * x } (3);
```

但是由于JavaScript语法解析的问题，会报SyntaxError错误，因此需要用括号把整个函数定义括起来：

```
(function (x) { return x * x }) (3);
```

通常，一个立即执行的匿名函数可以把函数体拆开，一般这么写：

```
(function (x) {
    return x * x;
})(3);
```

说了这么多，难道闭包就是为了返回一个函数然后延迟执行吗？

当然不是！闭包有非常强大的功能。举个栗子：

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

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

```
'use strict';

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

它用起来像这样：

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

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

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

闭包还可以把多参数的函数变成单参数的函数。例如，要计算 x^y 可以用 `Math.pow(x, y)` 函数，不过考虑到经常计算 x^2 或 x^3 ，我们可以利用闭包创建新的函数 `pow2` 和 `pow3`：

```
function make_pow(n) {
  return function (x) {
    return Math.pow(x, n);
  }
}

// 创建两个新函数：
var pow2 = make_pow(2);
var pow3 = make_pow(3);

pow2(5); // 25
pow3(7); // 343
```

脑洞大开

很久很久以前，有个叫阿隆佐·邱奇的帅哥，发现只需要用函数，就可以用计算机实现运算，而不需要 `0`、`1`、`2`、`3` 这些数字和 `+`、`-`、`*`、`/` 这些符号。

JavaScript 支持函数，所以可以用 JavaScript 用函数来写这些计算。来试试：

```
'use strict';

// 定义数字0:
var zero = function (f) {
    return function (x) {
        return x;
    }
};

// 定义数字1:
var one = function (f) {
    return function (x) {
        return f(x);
    }
};

// 定义加法:
function add(n, m) {
    return function (f) {
        return function (x) {
            return m(f)(n(f)(x));
        }
    }
}

// 计算数字2 = 1 + 1:
var two = add(one, one);

// 计算数字3 = 1 + 2:
var three = add(one, two);

// 计算数字5 = 2 + 3:
var five = add(two, three);

// 你说它是3就是3，你说它是5就是5，你怎么证明？

// 呵呵，看这里：

// 给3传一个函数,会打印3次:
(three(function () {
    console.log('print 3 times');
})))();

// 给5传一个函数,会打印5次:
(five(function () {
    console.log('print 5 times');
})))();

// 继续接着玩一会...
```

Run

ES6标准新增了一种新的函数：**Arrow Function**（箭头函数）。

为什么叫**Arrow Function**？因为它的定义用的就是一个箭头：

```
x => x * x
```

上面的箭头函数相当于：

```
function (x) {  
  return x * x;  
}
```

在继续学习箭头函数之前，请测试你的浏览器是否支持ES6的Array Function：

```
'use strict';  
  
var fn = x => x * x;  
  
alert('你的浏览器支持ES6的Array Function!');
```

Run

箭头函数相当于匿名函数，并且简化了函数定义。箭头函数有两种格式，一种像上面的，只包含一个表达式，连 `{ ... }` 和 `return` 都省略掉了。还有一种可以包含多条语句，这时候就不能省略 `{ ... }` 和 `return`：

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

如果参数不是一个，就需要用括号 `()` 括起来：


```
// 两个参数:
(x, y) => x * x + y * y
```

```
// 无参数:
() => 3.14
```

```
// 可变参数:
(x, y, ...rest) => {
  var i, sum = x + y;
  for (i=0; i<rest.length; i++) {
    sum += rest[i];
  }
  return sum;
}
```

如果要返回一个对象，就要注意，如果是单表达式，这么写的话会报错：

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

因为和函数体的 `{ ... }` 有语法冲突，所以要改为：

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

this

箭头函数看上去是匿名函数的一种简写，但实际上，箭头函数和匿名函数有个明显的区别：箭头函数内部的 `this` 是词法作用域，由上下文确定。

回顾前面的例子，由于JavaScript函数对 `this` 绑定的错误处理，下面的例子无法得到预期结果：

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

现在，箭头函数完全修复了 `this` 的指向，`this` 总是指向词法作用域，也就是外层调用者 `obj`：

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

如果使用箭头函数，以前的那种**hack**写法：

```
var that = this;
```

就不再需要了。

由于 **this** 在箭头函数中已经按照词法作用域绑定了，所以，用 **call()** 或者 **apply()** 调用箭头函数时，无法对 **this** 进行绑定，即传入的第一个参数被忽略：

```
var obj = {
  birth: 1990,
  getAge: function (year) {
    var b = this.birth; // 1990
    var fn = (y) => y - this.birth; // this.birth仍是1990
    return fn.call({birth:2000}, year);
  }
};
obj.getAge(2015); // 25
```

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

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

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

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

var r = foo(1); // 调用foo函数
```

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

generator跟函数很像，定义如下：

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

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

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

还是举个栗子吧。

我们以一个著名的斐波那契数列为例，它由 `0`，`1` 开头：

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

要编写一个产生斐波那契数列的函数，可以这么写：

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

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

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

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

直接调用试试：

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

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

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

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

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

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

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

```
for (var x of fib(5)) {
  console.log(x); // 依次输出0, 1, 1, 2, 3
}
```

generator和普通函数相比，有什么用？

因为generator可以在执行过程中多次返回，所以它看上去就像一个可以记住执行状态的函数，利用这一点，写一个generator就可以实现需要用面向对象才能实现的功能。例如，用一个对象来保存状态，得这么写：

```
var fib = {
  a: 0,
  b: 1,
  n: 0,
  max: 5,
  next: function () {
    var
      r = this.a,
      t = this.a + this.b;
    this.a = this.b;
    this.b = t;
    if (this.n < this.max) {
      this.n ++;
      return r;
    } else {
      return undefined;
    }
  }
};
```

用对象的属性来保存状态，相当繁琐。

generator还有另一个巨大的好处，就是把异步回调代码变成“同步”代码。这个好处要等到后面学了AJAX以后才能体会到。

没有generator之前的黑暗时代，用AJAX时需要这么写代码：

```

ajax('http://url-1', data1, function (err, result) {
  if (err) {
    return handle(err);
  }
  ajax('http://url-2', data2, function (err, result) {
    if (err) {
      return handle(err);
    }
    ajax('http://url-3', data3, function (err, result) {
      if (err) {
        return handle(err);
      }
      return success(result);
    });
  });
});

```

回调越多，代码越难看。

有了generator的美好时代，用AJAX时可以这么写：

```

try {
  r1 = yield ajax('http://url-1', data1);
  r2 = yield ajax('http://url-2', data2);
  r3 = yield ajax('http://url-3', data3);
  success(r3);
}
catch (err) {
  handle(err);
}

```

看上去是同步的代码，实际执行是异步的。

练习

要生成一个自增的ID，可以编写一个 `next_id()` 函数：

```

var current_id = 0;

function next_id() {
  current_id++;
  return current_id;
}

```

由于函数无法保存状态，故需要一个全局变量 `current_id` 来保存数字。

不用闭包，试用generator改写：

```

'use strict';
function* next_id() {

```

```

}

// 测试:
var
    x,
    pass = true,
    g = next_id();
for (x = 1; x < 100; x ++) {
    if (g.next().value !== x) {
        pass = false;
        alert('测试失败!');
        break;
    }
}
if (pass) {
    alert('测试通过!');
}

```

Run

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！

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。

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)
```

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) {  
    alert(Date.now()); // 老版本IE没有now()方法  
} else {  
    alert(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。

JavaScript学艺不精



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，上面的正则表达式是 `\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_\$\s][0-9a-zA-Z_\$\s]*` 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_\$\s][0-9a-zA-Z_\$\s]{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`，第一个元素始终是原始字符串本身，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@'],
    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])) {
        alert('测试失败: ' + should_pass[i]);
        success = false;
        break;
    }
}
for (i = 0; i < should_fail.length; i++) {
    if (re.test(should_fail[i])) {
        alert('测试失败: ' + should_fail[i]);
        success = false;
        break;
    }
}
if (success) {
    alert('测试通过!');
}
```

Run

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']) {
    alert('测试失败!');
}
else {
    alert('测试成功!');
}
```

Run

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·克洛克福特（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：

```
var xiaoming = {
  name: '小明',
  age: 14,
  gender: true,
  height: 1.65,
  grade: null,
  'middle-school': '\uW3C\u Middle School',
  skills: ['JavaScript', 'Java', 'Python', 'Lisp']
};
```

```
JSON.stringify(xiaoming); // '{"name":"小明","age":14,"gender":true,"height":1.65,"grade":null,"middle-school": "\uW3C\u Mi
```

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：

```
JSON.parse('{"name":"小明","age":14}', function (key, value) {  
    // 把number * 2:  
    if (key === 'name') {  
        return value + '同学';  
    }  
    return value;  
}); // Object {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Yahoo的天气API，查看返回的JSON数据。

JavaScript的所有数据都可以看成对象，那是不是我们已经在使用面向对象编程了呢？

当然不是。如果我们只使用 `Number`、`Array`、`string` 以及基本的 `{...}` 定义的对象，还无法发挥出面向对象编程的威力。

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...');
  }
};

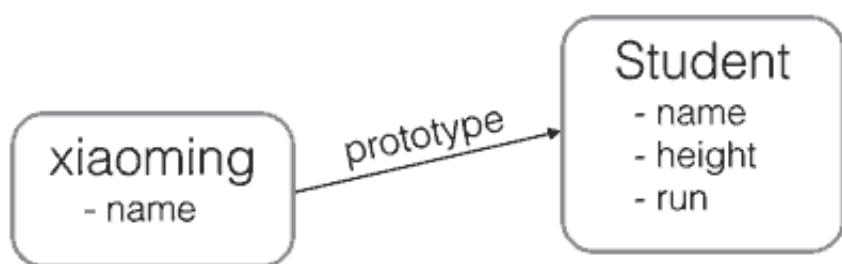
var xiaoming = {
  name: '小明'
};

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

请注意，上述代码仅用于演示目的。在编写JavaScript代码时，不要直接用 `obj.__proto__` 去改变一个对象的原型，并且，低版本的IE也无法使用 `__proto__`。 `Object.create()` 方法可以传入一个原型对象，并创建一个基于该原型的新对象，但是新对象什么属性都没有，因此，我们可以编写一个函数来创建 `xiaoming`：

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

function createStudent(name) {
  // 基于Student原型创建一个新对象：
  var s = Object.create(Student);
  // 初始化新对象：
  s.name = name;
  return s;
}

var xiaoming = createStudent('小明');
xiaoming.run(); // 小明 is running...
xiaoming.__proto__ === Student; // true
```

这是创建原型继承的一种方法，JavaScript还有其他方法来创建对象，我们在后面会一一讲到。

JavaScript对每个创建的对象都会设置一个原型，指向它的原型对象。

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

例如，创建一个 `Array` 对象：

```
var arr = [1, 2, 3];
```

其原型链是：

```
arr ----> Array.prototype ----> Object.prototype ----> null
```

`Array.prototype` 定义了 `indexOf()`、`shift()` 等方法，因此你可以在所有的 `Array` 对象上直接调用这些方法。

当我们创建一个函数时：

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

函数也是一个对象，它的原型链是：

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

由于 `Function.prototype` 定义了 `apply()` 等方法，因此，所有函数都可以调用 `apply()` 方法。

很容易想到，如果原型链很长，那么访问一个对象的属性就会因为花更多的时间查找而变得更慢，因此要注意不要把原型链搞得太长。

构造函数

除了直接用 `{ ... }` 创建一个对象外，JavaScript 还可以用一种构造函数的方法来创建对象。它的用法是，先定义一个构造函数：

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

你会问，咦，这不是一个普通函数吗？

这确实是一个普通函数，但是在 JavaScript 中，可以用关键字 `new` 来调用这个函数，并返回一个对象：

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

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

新创建的 `xiaoming` 的原型链是：

```
xiaoming ----> Student.prototype ----> Object.prototype ----> null
```

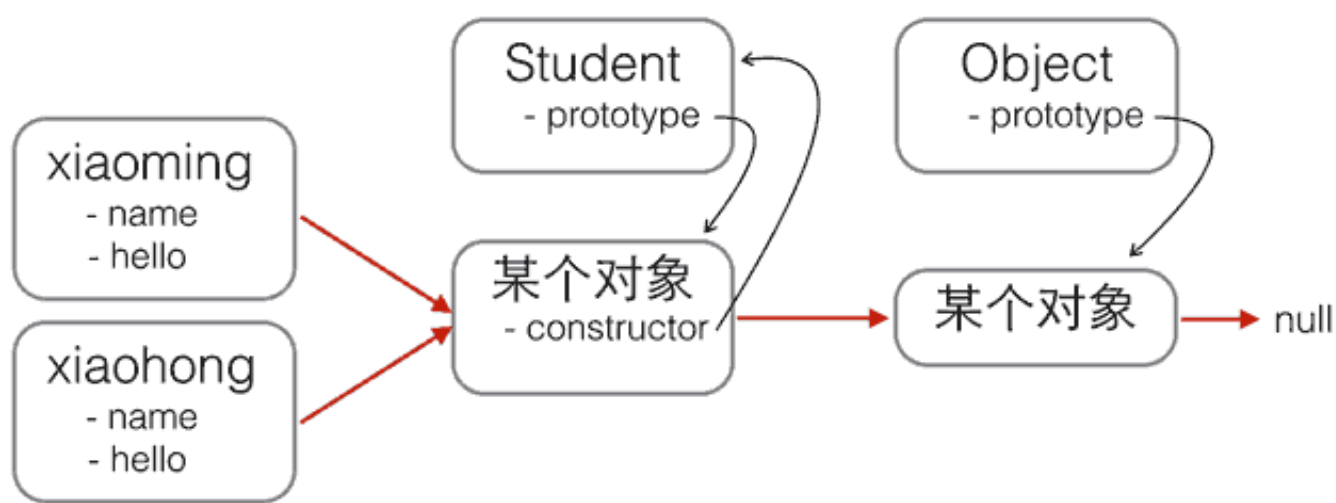
也就是说，`xiaoming` 的原型指向函数 `Student` 的原型。如果你又创建了 `xiaohong`、`xiaojun`，那么这些对象的原型与 `xiaoming` 是一样的：

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

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

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

看晕了吧？用一张图来表示这些乱七八糟的关系就是：



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

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

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

不过还有一个小问题，注意观察：

```

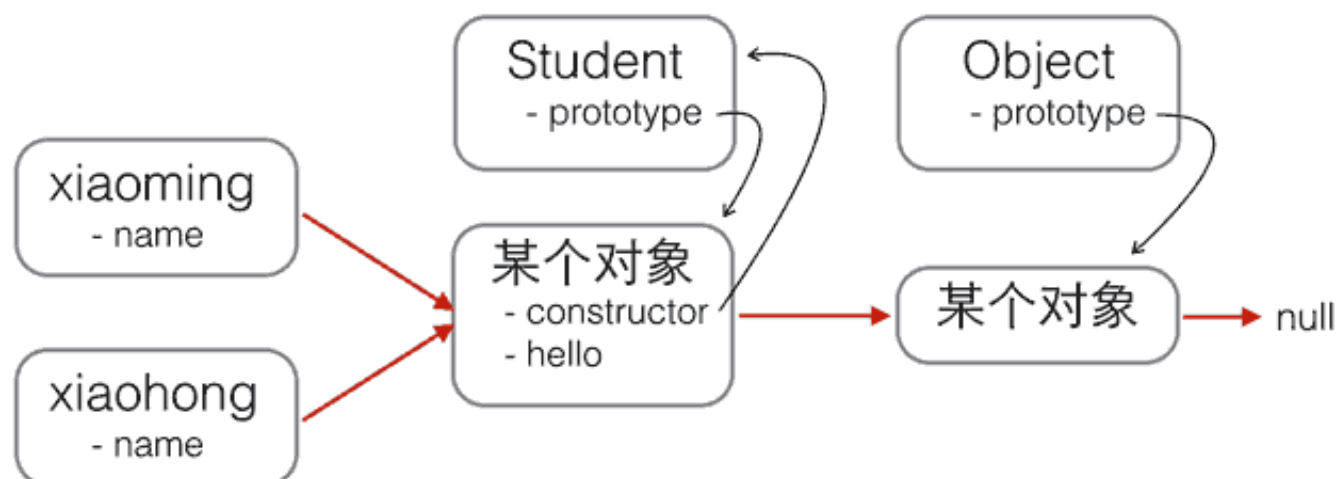
xiaoming.name; // '小明'
xiaohong.name; // '小红'
xiaoming.hello; // function: Student.hello()
xiaohong.hello; // function: Student.hello()
xiaoming.hello === xiaohong.hello; // false
  
```

`xiaoming` 和 `xiaohong` 各自的 `name` 不同，这是对的，否则我们无法区分谁是谁了。

`xiaoming` 和 `xiaohong` 各自的 `hello` 是一个函数，但它们是两个不同的函数，虽然函数名称和代码都是相同的！

如果我们通过 `new Student()` 创建了很多对象，这些对象的 `hello` 函数实际上只需要共享同一个函数就可以了，这样可以节省很多内存。

要让创建的对象共享一个 `hello` 函数，根据对象的属性查找原则，我们只要把 `hello` 函数移动到 `xiaoming`、`xiaohong` 这些对象共同的原型上就可以了，也就是 `Student.prototype`：



修改代码如下：

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

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

用 `new` 创建基于原型的JavaScript的对象就是这么简单！

忘记写new怎么办

如果一个函数被定义为用于创建对象的构造函数，但是调用时忘记了写 `new` 怎么办？

在strict模式下，`this.name = name` 将报错，因为 `this` 绑定为 `undefined`，在非strict模式下，`this.name = name` 不报错，因为 `this` 绑定为 `window`，于是无意间创建了全局变量 `name`，并且返回 `undefined`，这个结果更糟糕。

所以，调用构造函数千万不要忘记写 `new`。为了区分普通函数和构造函数，按照约定，构造函数首字母应当大写，而普通函数首字母应当小写，这样，一些语法检查工具如[jslint](#)将可以帮你检测到漏写的 `new`。

最后，我们还可以编写一个 `createStudent()` 函数，在内部封装所有的 `new` 操作。一个常用的编程模式像这样：

```
function Student(props) {
    this.name = props.name || '匿名'; // 默认值为'匿名'
    this.grade = props.grade || 1; // 默认值为1
}

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

function createStudent(props) {
    return new Student(props || {})
}
```

这个 `createStudent()` 函数有几个巨大的优点：一是不需要 `new` 来调用，二是参数非常灵活，可以不传，也可以这么传：

```
var xiaoming = createStudent({
    name: '小明'
});

xiaoming.grade; // 1
```

如果创建的对象有很多属性，我们只需要传递需要的某些属性，剩下的属性可以用默认值。由于参数是一个Object，我们无需记忆参数的顺序。如果恰好从 `JSON` 拿到了一个对象，就可以直接创建出 `xiaoming`。

练习

请利用构造函数定义 `Cat`，并让所有的Cat对象有一个 `name` 属性，并共享一个方法 `say()`，返回字符串 `'Hello, xxx!'`：

```
'use strict';
```



```
function Cat(name) {  
    //  
}  
  
// 测试:  
var kitty = new Cat('Kitty');  
var doraemon = new Cat('哆啦A梦');  
if (kitty && kitty.name === 'Kitty' && kitty.say && typeof kitty.say === 'function' && kitty.say === '喵喵')  
    alert('测试通过!');  
} else {  
    alert('测试失败!');  
}
```

Run

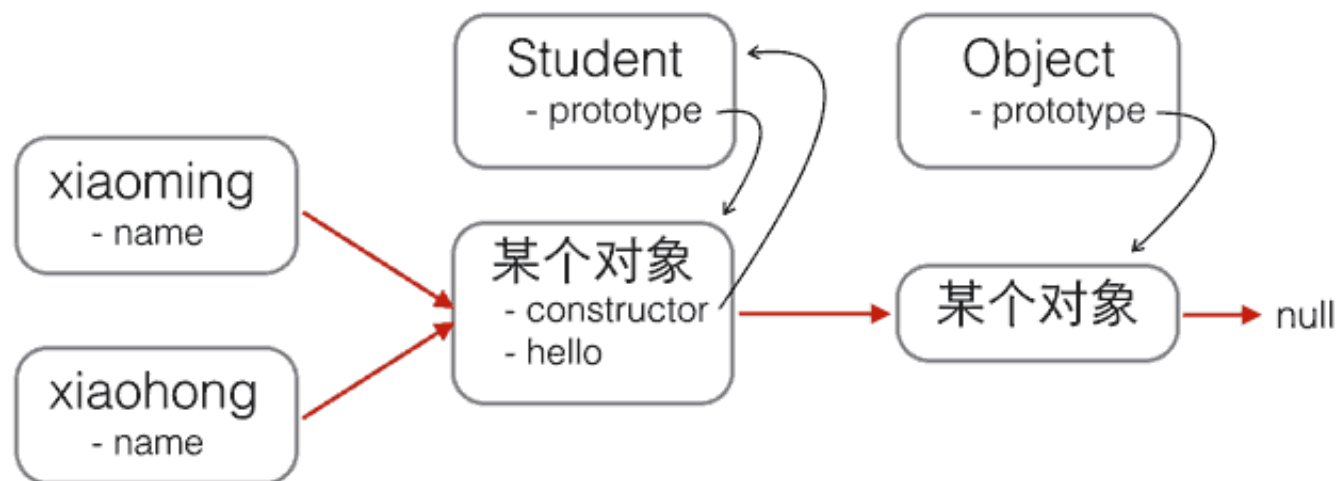
在传统的基于**Class**的语言如**Java**、**C++**中，继承的本质是扩展一个已有的**Class**，并生成新的**Subclass**。

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

但是办法还是有的。我们先回顾 `Student` 构造函数：

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

以及 `Student` 的原型链：



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

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

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

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

必须想办法把原型链修改为：

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

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

如果你想用最简单粗暴的方法这么干：

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

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

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

```

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

// 空函数F：
function F() {}

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

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

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

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

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

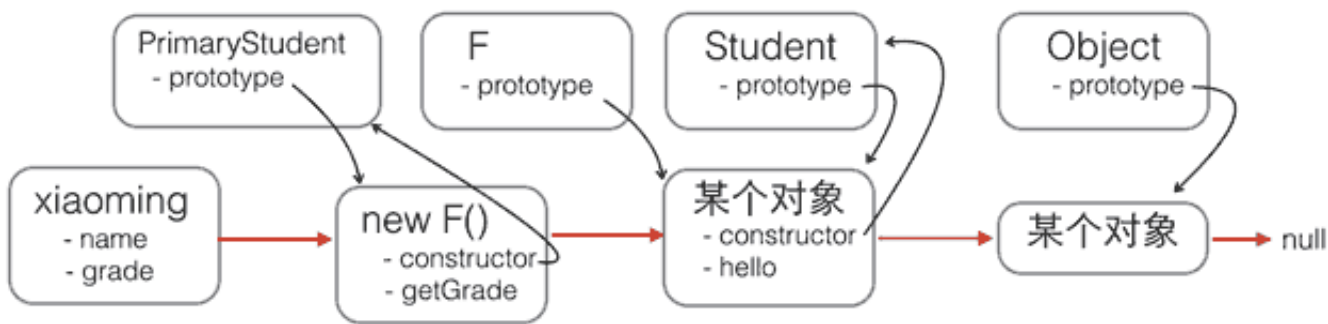
xiaoming.name; // '小明'
xiaoming.grade; // 2

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

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

```

用一张图来表示新的原型链：



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

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

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

这个 `extends()` 函数可以复用：

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

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

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

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

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

小结

JavaScript的原型继承实现方式就是：

1. 定义新的构造函数，并在内部用 `call()` 调用希望“继承”的构造函数，并绑定 `this`；
2. 借助中间函数 `F` 实现原型链继承，最好通过封装的 `extends` 函数完成；
3. 继续在新的构造函数的原型上定义新方法。

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 10.10 Yosemite自带的Safari版本是8.x，早已支持ES6；
- **Firefox**：Mozilla自己研制的Gecko内核和JavaScript引擎OdinMonkey。早起的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上。

JavaScript可以获取浏览器提供的很多对象，并进行操作。

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：  
alert('window inner size: ' + window.innerWidth + ' x ' + window.innerHeight);
```

Run

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

navigator

`navigator`对象表示浏览器的信息，最常用的属性包括：

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

```
'use strict';
```

```
alert('appName = ' + navigator.appName + '\n' +  
      'appVersion = ' + navigator.appVersion + '\n' +  
      'language = ' + navigator.language + '\n' +  
      'platform = ' + navigator.platform + '\n' +  
      'userAgent = ' + navigator.userAgent);
```

Run

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

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

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

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

screen

`screen` 对象表示屏幕的信息，常用的属性有：

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

```
'use strict';  
  
alert('Screen size = ' + screen.width + ' x ' + screen.height);
```

Run

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。

```
'use strict';

if (confirm('重新加载当前页' + location.href + '?')) {
    location.reload();
} else {
    location.assign('/discuss'); // 设置一个新的URL地址
}
```

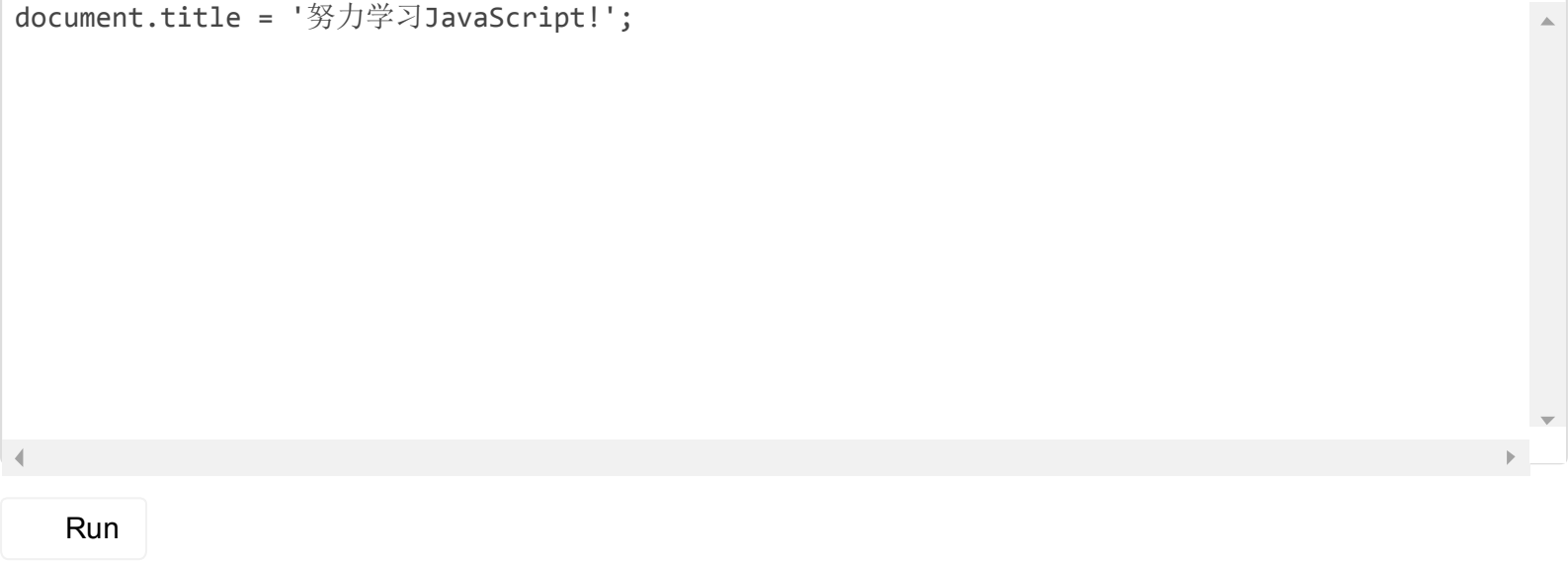
Run

document

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

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

```
'use strict';
```



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

摩卡
热摩卡咖啡
酸奶
北京老酸奶
果汁
鲜榨苹果汁

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

```
'use strict';
```



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

menu = document.getElementById('drink-menu');
menu.tagName; // 'DL'

drinks = document.getElementsByTagName('dt');
s = '提供的饮料:';
for (i=0; i<drinks.length; i++) {
    s = s + drinks[i].innerHTML + ',';
}
alert(s);
```

Run

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

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

Cookie还可以存储网站的一些设置，例如，页面显示的语言等等。

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

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

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

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

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

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

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

history

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

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

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

任何情况，你都不应该使用 `history` 这个对象了。

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.firstChild;
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。

严格地讲，我们这里的DOM节点是指 `Element`，但是DOM节点实际上是 `Node`，在HTML中，`Node` 包括 `Element`、`Comment`、`CDATA_SECTION` 等很多种，以及根节点 `Document` 类型，但是，绝大多数时候我们只关心 `Element`，也就是实际控制页面结构的 `Node`，其他类型的 `Node` 忽略即可。根节点 `Document` 已经自动绑定为全局变量 `document`。

练习

如下的HTML结构：

JavaScript

Java

Python

Ruby

Swift

Scheme

Haskell

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

请选择出指定条件的节点：

```
'use strict';
```

```
// 选择<p>JavaScript</p>:
var js = ???;

// 选择<p>Python</p>,<p>Ruby</p>,<p>Swift</p>:
var arr = ???;

// 选择<p>Haskell</p>:
var haskell = ???;

// 测试:
if (!js || js.innerText !== 'JavaScript') {
    alert('选择JavaScript失败!');
} else if (!arr || arr.length !== 3 || !arr[0] || !arr[1] || !arr[2] || arr[0].innerText !== 'Py
    alert('选择Python,Ruby,Swift失败!');
} else if (!haskell || haskell.innerText !== 'Haskell') {
    alert('选择Haskell失败!');
} else {
    alert('测试通过!');
}
```

Run

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

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

两者的区别在于读取属性时，`innerText` 不返回隐藏元素的文本，而 `textContent` 返回所有文本。另外注意IE<9不支持 `textContent`。

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';
```

练习

有如下的HTML结构：

javascript

Java

```
<!-- HTML结构 -->
<div id="test-div">
  <p id="test-js">javascript</p>
  <p>Java</p>
</div>
```

请尝试获取指定节点并修改：

```
'use strict';

// 获取<p>javascript</p>节点:
var js = ???;

// 修改文本为JavaScript:
// TODO:

// 修改CSS为: color: #ff0000, font-weight: bold
// TODO:
```

```
// 测试:
if (js && js.parentNode && js.parentNode.id === 'test-div' && js.id === 'test-js') {
    if (js.innerText === 'JavaScript') {
        if (js.style && js.style.fontWeight === 'bold' && (js.style.color === 'red' || js.style.
            alert('测试通过!');
        } else {
            alert('CSS样式测试失败!');
        }
    } else {
        alert('文本测试失败!');
    }
} else {
    alert('节点测试失败!');
}
```

Run

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，`<div></div>`，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 `<p id="js">JavaScript</p>` 添加到 `<div id="list">` 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 `<style>` 节点，然后把它添加到 `<head>` 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 }';
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>
```

可见，使用 `insertBefore` 重点是要拿到一个“参考子节点”的引用。很多时候，需要循环一个父节点的所有子节点，可以通过迭代 `children` 属性实现：

```
var
  i, c,
  list = document.getElementById('list');
for (i = 0; i < list.children.length; i++) {
  c = list.children[i]; // 拿到第i个子节点
}
```

练习

对于一个已有的HTML结构：

1. Scheme
2. JavaScript
3. Python
4. Ruby
5. Haskell

```
<!-- HTML结构 -->
<ol id="test-list">
  <li class="lang">Scheme</li>
  <li class="lang">JavaScript</li>
  <li class="lang">Python</li>
  <li class="lang">Ruby</li>
  <li class="lang">Haskell</li>
</ol>
```

按字符串顺序重新排序DOM节点：

```
'use strict';
```



```
// sort list:
```

```
// 测试:
;(function () {
    var
        arr, i,
        t = document.getElementById('test-list');
    if (t && t.children && t.children.length === 5) {
        arr = [];
        for (i=0; i<t.children.length; i++) {
            arr.push(t.children[i].innerText);
        }
        if (arr.toString() === ['Haskell', 'JavaScript', 'Python', 'Ruby', 'Scheme'].toString())
            alert('测试通过!');
        else {
            alert('测试失败: ' + arr.toString());
        }
    }
    else {
        alert('测试失败!');
    }
})();
```

Run

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 `<p>First</p>` 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。

练习

- JavaScript
- Swift
- HTML
- ANSI C
- CSS
- DirectX

```
<!-- HTML结构 -->
<ul id="test-list">
  <li>JavaScript</li>
  <li>Swift</li>
  <li>HTML</li>
  <li>ANSI C</li>
  <li>CSS</li>
  <li>DirectX</li>
</ul>
```

把与Web开发技术不相关的节点删掉：

```
'use strict';
```

```
// TODO
```

```
// 测试:
;(function () {
    var
        arr, i,
        t = document.getElementById('test-list');
    if (t && t.children && t.children.length === 3) {
        arr = [];
        for (i = 0; i < t.children.length; i++) {
            arr.push(t.children[i].innerText);
        }
        if (arr.toString() === ['JavaScript', 'HTML', 'CSS'].toString()) {
            alert('测试通过!');
        }
        else {
            alert('测试失败: ' + arr.toString());
        }
    }
    else {
        alert('测试失败!');
    }
})();
```

Run

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

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

HTML表单的输入控件主要有以下几种：

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

获取值

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

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

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

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

设置值

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

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

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

HTML5控件

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

`<input type="date">`:

`<input type="datetime-local">`:

`<input type="color">`:

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

提交表单

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

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

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

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

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

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

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

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

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

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

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

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

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

要想不改变用户的输入，可以利用实现：

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

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

注意到id为md5-password的标记了name="password"，而用户输入的id为input-password的没有name属性。没有name属性的的数据不会被提交。

练习

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

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

用户名:

口令:

重复口令:

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

```
'use strict';
function checkRegisterForm() {

  // TODO:
  return true;
}

// 测试:
;(function () {
  var form = document.getElementById('test-register');
  clearFormListeners(form);
  gListener = checkRegisterForm;
  form.addEventListener('submit', checkRegisterForm);
  form.submit();
})();
```

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

注意：当一个表单包含 `<input type="file">` 时，表单的 `enctype` 必须指定为 `multipart/form-data`，`method` 必须指定为 `post`，浏览器才能正确编码并以 `multipart/form-data` 格式发送表单的数据。

出于安全考虑，浏览器只允许用户点击 `<input type="file">` 来选择本地文件，用JavaScript对 `<input type="file">`

的 `value` 赋值是没有任何效果的。当用户选择了上传某个文件后，**JavaScript**也无法获得该文件的真实路径：

未选择任何文件

待上传文件：

通常，上传的文件都由后台服务器处理，**JavaScript**可以在提交表单时对文件扩展名做检查，以便防止用户上传无效格式的文件：

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

File API

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

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

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

下面的例子演示了如何读取用户选取的图片文件，并在一个 `<div>` 中预览图像：

图片预览：

未选择任何文件


```
var
    fileInput = document.getElementById('test-image-file'),
    info = document.getElementById('test-file-info'),
    preview = document.getElementById('test-image-preview');
// 监听change事件:
fileInput.addEventListener('change', function () {
    // 清除背景图片:
    preview.style.backgroundImage = '';
    // 检查文件是否选择:
    if (!fileInput.value) {
        info.innerHTML = '没有选择文件';
        return;
    }
    // 获取File引用:
    var file = fileInput.files[0];
    // 获取File信息:
    info.innerHTML = '文件: ' + file.name + '<br>' +
        '大小: ' + file.size + '<br>' +
        '修改: ' + file.lastModifiedDate;
    if (file.type !== 'image/jpeg' && file.type !== 'image/png' && file.type !== 'image/gif') {
        alert('不是有效的图片文件!');
        return;
    }
    // 读取文件:
    var reader = new FileReader();
    reader.onload = function(e) {
        var
            data = e.target.result; // 'data:image/jpeg;base64,/9j/4AAQSk...(base64编码)...'
        preview.style.backgroundImage = 'url(' + data + ')';
    };
    // 以DataURL的形式读取文件:
    reader.readAsDataURL(file);
});
```

上面的代码演示了如何通过HTML5的File API读取文件内容。以DataURL的形式读取到的文件是一个字符串，类似于data:image/jpeg;base64,/9j/4AAQSk...(base64编码)...，常用于设置图像。如果需要服务器端处理，把字符串base64,后面的字符发送给服务器并用Base64解码就可以得到原始文件的二进制内容。

回调

上面的代码还演示了JavaScript的一个重要的特性就是单线程执行模式。在JavaScript中，浏览器的JavaScript执行引擎在执行JavaScript代码时，总是以单线程模式执行，也就是说，任何时候，JavaScript代码都不可能同时有多于1个线程在执行。

你可能会问，单线程模式执行的JavaScript，如何处理多任务？

在JavaScript中，执行多任务实际上都是异步调用，比如上面的代码：

```
reader.readAsDataURL(file);
```

就会发起一个异步操作来读取文件内容。因为是异步操作，所以我们在JavaScript代码中就不知道什么时候操作结束，因此需要先设置一个回调函数：

```
reader.onload = function(e) {  
    // 当文件读取完成后，自动调用此函数：  
};
```

当文件读取完成后，**JavaScript**引擎将自动调用我们设置的回调函数。执行回调函数时，文件已经读取完毕，所以我们可以从回调函数内部安全地获得文件内容。

AJAX不是**JavaScript**的规范，它只是一个哥们“发明”的缩写：**Asynchronous JavaScript and XML**，意思就是用**JavaScript**执行异步网络请求。

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

这就是**Web**的运作原理：一次**HTTP**请求对应一个页面。

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

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

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

在现代浏览器上写**AJAX**主要依靠 `XMLHttpRequest` 对象：

```
'use strict';
```

```

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

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

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

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

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

alert('请求已发送，请等待响应...');

```

Run

响应结果：

对于低版本的IE，需要换一个 `ActiveXObject` 对象：

```
'use strict';
```

```

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

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

var request = new XMLHttpRequest('Microsoft.XMLHTTP'); // 新建Microsoft.XMLHTTP对象

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

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

alert('请求已发送，请等待响应...');

```

Run

IE响应结果：

如果你想把标准写法和IE写法混在一起，可以这么写：

```
var request;
if (window.XMLHttpRequest) {
    request = new XMLHttpRequest();
} else {
    request = new ActiveXObject('Microsoft.XMLHTTP');
}
```

通过检测 `window` 对象是否有 `XMLHttpRequest` 属性来确定浏览器是否支持标准的 `XMLHttpRequest`。注意，不要根据浏览器的 `navigator.userAgent` 来检测浏览器是否支持某个JavaScript特性，一是因为这个字符串本身可以伪造，二是通过IE版本判断JavaScript特性将非常复杂。

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

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

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

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

安全限制

上面代码的URL使用的是相对路径。如果你把它改为 `'http://www.sina.com.cn/'`，再运行，肯定报错。在Chrome的控制台里，还可以看到错误信息。

这是因为浏览器的同源策略导致的。默认情况下，JavaScript在发送AJAX请求时，URL的域名必须和当前页面完全一致。

完全一致的意思是，域名要相同（`www.example.com` 和 `example.com` 不同），协议要相同（`http` 和 `https` 不同），端口号要相同（默认是 `:80` 端口，它和 `:8080` 就不同）。有的浏览器口子松一点，允许端口不同，大多数浏览器都会严格遵守这个限制。

那是不是用JavaScript无法请求外域（就是其他网站）的URL了呢？方法还是有的，大概有这么几种：

一是通过Flash插件发送HTTP请求，这种方式可以绕过浏览器的安全限制，但必须安装Flash，并且跟Flash交互。不过Flash用起来麻烦，而且现在用得也越来越少了。

二是通过在同源域名下架设一个代理服务器来转发，JavaScript负责把请求发送到代理服务器：

```
'/proxy?url=http://www.sina.com.cn'
```

代理服务器再把结果返回，这样就遵守了浏览器的同源策略。这种方式麻烦之处在于需要服务器端额外做开发。

第三种方式称为JSONP，它有个限制，只能用GET请求，并且要求返回JavaScript。这种方式跨域实际上是利用了浏览器允许跨域引用JavaScript资源：

```
<html>
<head>
  <script src="http://example.com/abc.js"></script>
  ...
</head>
<body>
  ...
</body>
</html>
```

JSONP通常以函数调用的形式返回，例如，返回JavaScript内容如下：

```
foo('data');
```

这样一来，我们如果在页面中先准备好 `foo()` 函数，然后给页面动态加一个 `<script>` 节点，相当于动态读取外域的JavaScript资源，最后就等着接收回调了。

以163的股票查询URL为例，对于URL：<http://api.money.126.net/data/feed/0000001,1399001?callback=refreshPrice>，你将得到如下返回：

```
refreshPrice({"0000001":{"code": "0000001", ... }});
```

因此我们需要首先在页面中准备好回调函数：

```
function refreshPrice(data) {
  var p = document.getElementById('test-jsonp');
  p.innerHTML = '当前价格：' +
    data['0000001'].name + '：' +
    data['0000001'].price + '；' +
    data['1399001'].name + '：' +
    data['1399001'].price;
}
```

当前价格：

刷新

最后用 `getPrice()` 函数触发：

```
function getPrice() {
  var
    js = document.createElement('script'),
    head = document.getElementsByTagName('head')[0];
  js.src = 'http://api.money.126.net/data/feed/0000001,1399001?callback=refreshPrice';
  head.appendChild(js);
}
```

就完成了跨域加载数据。

CORS

如果浏览器支持HTML5，那么就可以一劳永逸地使用新的跨域策略：CORS了。

CORS全称Cross-Origin Resource Sharing，是HTML5规范定义的如何跨域访问资源。

了解CORS前，我们先搞明白概念：

Origin表示本域，也就是浏览器当前页面的域。当JavaScript向外域（如sina.com）发起请求后，浏览器收到响应后，首先检查Access-Control-Allow-Origin是否包含本域，如果是，则此次跨域请求成功，如果不是，则请求失败，JavaScript将无法获取到响应的任何数据。

用一个图来表示就是：



假设本域是my.com，外域是sina.com，只要响应头Access-Control-Allow-Origin为http://my.com，或者是*，本次请求就可以成功。

可见，跨域能否成功，取决于对方服务器是否愿意给你设置一个正确的Access-Control-Allow-Origin，决定权始终在对方手中。

上面这种跨域请求，称之为“简单请求”。简单请求包括GET、HEAD和POST（POST的Content-Type类型仅限application/x-www-form-urlencoded、multipart/form-data和text/plain），并且不能出现任何自定义头（例如，X-Custom: 12345），通常能满足90%的需求。

无论你是否需要用JavaScript通过CORS跨域请求资源，你都要了解CORS的原理。最新的浏览器全面支持HTML5。在引用外域资源时，除了JavaScript和CSS外，都要验证CORS。例如，当你引用了某个第三方CDN上的字体文件时：

```
/* CSS */
@font-face {
  font-family: 'FontAwesome';
  src: url('http://cdn.com/fonts/fontawesome.ttf') format('truetype');
}
```

如果该CDN服务商未正确设置Access-Control-Allow-Origin，那么浏览器无法加载字体资源。

对于PUT、DELETE以及其他类型如application/json的POST请求，在发送AJAX请求之前，浏览器会先发送一个OPTIONS请求（称为preflighted请求）到这个URL上，询问目标服务器是否接受：

```
OPTIONS /path/to/resource HTTP/1.1
Host: bar.com
Origin: http://bar.com
Access-Control-Request-Method: POST
```

服务器必须响应并明确指出允许的Method：

```
HTTP/1.1 200 OK
Access-Control-Allow-Origin: http://foo.com
Access-Control-Allow-Methods: POST, GET, PUT, OPTIONS
Access-Control-Max-Age: 86400
```

浏览器确认服务器响应的 `Access-Control-Allow-Methods` 头确实包含将要发送的 **AJAX** 请求的 **Method**，才会继续发送 **AJAX**，否则，抛出一个错误。

由于以 `POST`、`PUT` 方式传送 **JSON** 格式的数据在 **REST** 中很常见，所以要跨域正确处理 `POST` 和 `PUT` 请求，服务器端必须正确响应 `OPTIONS` 请求。

需要深入了解 **CORS** 的童鞋请移步 [W3C 文档](#)。

在 **JavaScript** 的世界中，所有代码都是单线程执行的。

由于这个“缺陷”，导致 **JavaScript** 的所有网络操作，浏览器事件，都必须是异步执行。异步执行可以用回调函数实现：

```
function callback() {
    console.log('Done');
}
console.log('before setTimeout()');
setTimeout(callback, 1000); // 1秒钟后调用callback函数
console.log('after setTimeout()');
```

观察上述代码执行，在 **Chrome** 的控制台输出可以看到：

```
before setTimeout()
after setTimeout()
(等待1秒后)
Done
```

可见，异步操作会在将来的某个时间点触发一个函数调用。

AJAX 就是典型的异步操作。以上一节的代码为例：

```
request.onreadystatechange = function () {
    if (request.readyState === 4) {
        if (request.status === 200) {
            return success(request.responseText);
        } else {
            return fail(request.status);
        }
    }
}
```

把回调函数 `success(request.responseText)` 和 `fail(request.status)` 写到一个 **AJAX** 操作里很正常，但是不好看，而且不利于代码复用。

有没有更好的写法？比如写成这样：

```
var ajax = ajaxGet('http://...');
ajax.ifSuccess(success)
    .ifFail(fail);
```


这种链式写法的好处在于，先统一执行AJAX逻辑，不关心如何处理结果，然后，根据结果是成功还是失败，在将来的某个时候调用 `success` 函数或 `fail` 函数。

古人云：“君子一诺千金”，这种“承诺将来会执行”的对象在JavaScript中称为Promise对象。

Promise有各种开源实现，在ES6中被统一规范，由浏览器直接支持。先测试一下你的浏览器是否支持Promise：

```
'use strict';

new Promise(function () {});

// 直接运行测试：
alert('支持Promise!');
```

Run

我们先看一个最简单的Promise例子：生成一个0-2之间的随机数，如果小于1，则等待一段时间后返回成功，否则返回失败：

```
function test(resolve, reject) {
  var timeout = Math.random() * 2;
  log('set timeout to: ' + timeout + ' seconds.');
```

```
  setTimeout(function () {
    if (timeout < 1) {
      log('call resolve()...');
      resolve('200 OK');
```

```
    }
    else {
      log('call reject()...');
      reject('timeout in ' + timeout + ' seconds.');
```

```
    }
  }, timeout * 1000);
}
```

这个 `test()` 函数有两个参数，这两个参数都是函数，如果执行成功，我们将调用 `resolve('200 OK')`，如果执行失败，我们将调用 `reject('timeout in ' + timeout + ' seconds.')`。可以看出，`test()` 函数只关心自身的逻辑，并不关心具体的 `resolve` 和 `reject` 将如何处理结果。

有了执行函数，我们就可以用一个Promise对象来执行它，并在将来某个时刻获得成功或失败的结果：

```
var p1 = new Promise(test);
var p2 = p1.then(function (result) {
    console.log('成功: ' + result);
});
var p3 = p2.catch(function (reason) {
    console.log('失败: ' + reason);
});
```

变量 `p1` 是一个 **Promise** 对象，它负责执行 `test` 函数。由于 `test` 函数在内部是异步执行的，当 `test` 函数执行成功时，我们告诉 **Promise** 对象：

```
// 如果成功，执行这个函数：
p1.then(function (result) {
    console.log('成功: ' + result);
});
```

当 `test` 函数执行失败时，我们告诉 **Promise** 对象：

```
p2.catch(function (reason) {
    console.log('失败: ' + reason);
});
```

Promise 对象可以串联起来，所以上述代码可以简化为：

```
new Promise(test).then(function (result) {
    console.log('成功: ' + result);
}).catch(function (reason) {
    console.log('失败: ' + reason);
});
```

实际测试一下，看看 **Promise** 是如何异步执行的：

```
'use strict';

// 清除log:
var logging = document.getElementById('test-promise-log');
while (logging.children.length > 1) {
    logging.removeChild(logging.children[logging.children.length - 1]);
}

// 输出log到页面:
function log(s) {
    var p = document.createElement('p');
    p.innerHTML = s;
    logging.appendChild(p);
}
```

```

new Promise(function (resolve, reject) {
  log('start new Promise...');
  var timeout = Math.random() * 2;
  log('set timeout to: ' + timeout + ' seconds.');
```

```

  setTimeout(function () {
    if (timeout < 1) {
      log('call resolve()...');
      resolve('200 OK');
    }
    else {
      log('call reject()...');
      reject('timeout in ' + timeout + ' seconds.');
```

```

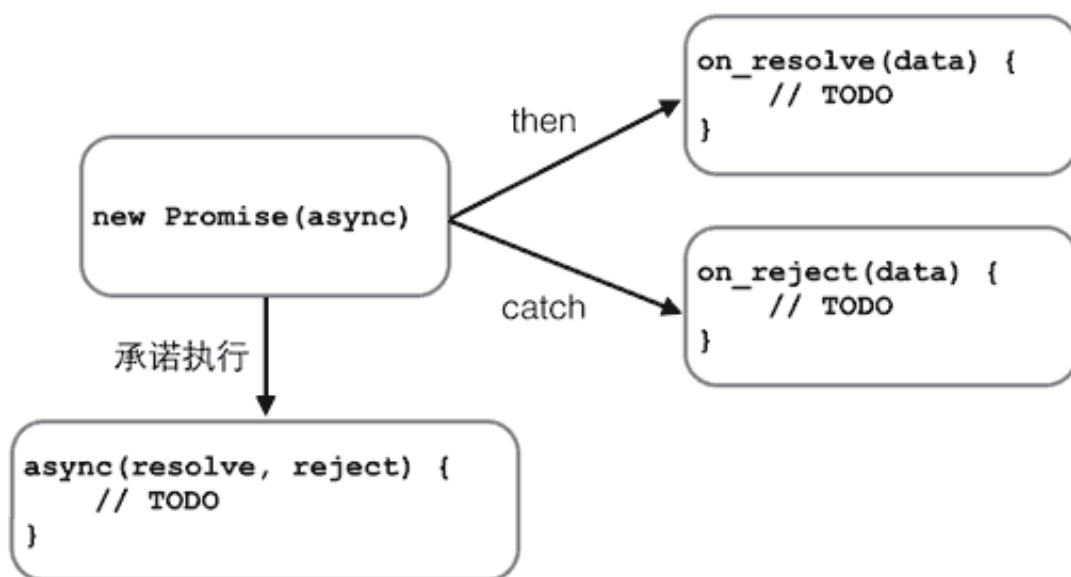
    }
  }, timeout * 1000);
}).then(function (r) {
  log('Done: ' + r);
}).catch(function (reason) {
  log('Failed: ' + reason);
});

```

Run

Log:

可见**Promise**最大的好处是在异步执行的流程中，把执行代码和处理结果的代码清晰地分离了：



Promise还可以做更多的事情，比如，有若干个异步任务，需要先做任务1，如果成功后再做任务2，任何任务失败则不再继续并执行错误处理函数。

要串行执行这样的异步任务，不用**Promise**需要写一层一层的嵌套代码。有了**Promise**，我们只需要简单地写：

```
job1.then(job2).then(job3).catch(handleError);
```

其中，`job1`、`job2` 和 `job3` 都是 `Promise` 对象。

下面的例子演示了如何串行执行一系列需要异步计算获得结果的任务：

```
'use strict';

var logging = document.getElementById('test-promise2-log');
while (logging.children.length > 1) {
    logging.removeChild(logging.children[logging.children.length - 1]);
}

function log(s) {
    var p = document.createElement('p');
    p.innerHTML = s;
    logging.appendChild(p);
}

// 0.5秒后返回input*input的计算结果:
function multiply(input) {
    return new Promise(function (resolve, reject) {
        log('calculating ' + input + ' x ' + input + '...');
        setTimeout(resolve, 500, input * input);
    });
}

// 0.5秒后返回input+input的计算结果:
function add(input) {
    return new Promise(function (resolve, reject) {
        log('calculating ' + input + ' + ' + input + '...');
        setTimeout(resolve, 500, input + input);
    });
}

var p = new Promise(function (resolve, reject) {
    log('start new Promise...');
    resolve(123);
});

p.then(multiply)
  .then(add)
  .then(multiply)
  .then(add)
  .then(function (result) {
    log('Got value: ' + result);
  });
```

Run

Log:

`setTimeout` 可以看成是一个模拟网络等异步执行的函数。现在，我们把上一节的AJAX异步执行函数转换为Promise对象，看看用Promise如何简化异步处理：

```
'use strict';

// ajax函数将返回Promise对象：
function ajax(method, url, data) {
    var request = new XMLHttpRequest();
    return new Promise(function (resolve, reject) {
        request.onreadystatechange = function () {
            if (request.readyState === 4) {
                if (request.status === 200) {
                    resolve(request.responseText);
                } else {
                    reject(request.status);
                }
            }
        };
        request.open(method, url);
        request.send(data);
    });
}

var log = document.getElementById('test-promise-ajax-result');
var p = ajax('GET', '/api/categories');
p.then(function (text) { // 如果AJAX成功，获得响应内容
    log.innerText = text;
}).catch(function (status) { // 如果AJAX失败，获得响应代码
    log.innerText = 'ERROR: ' + status;
});
```

Run

Result:

除了串行执行若干异步任务外，**Promise**还可以并行执行异步任务。

试想一个页面聊天系统，我们需要从两个不同的URL分别获得用户的个人信息和好友列表，这两个任务是可以并行执行的，用 `Promise.all()` 实现如下：

```
var p1 = new Promise(function (resolve, reject) {
    setTimeout(resolve, 500, 'P1');
});
var p2 = new Promise(function (resolve, reject) {
    setTimeout(resolve, 600, 'P2');
});
// 同时执行p1和p2, 并在它们都完成后执行then:
Promise.all([p1, p2]).then(function (results) {
    console.log(results); // 获得一个Array: ['P1', 'P2']
});
```

有些时候, 多个异步任务是为了容错。比如, 同时向两个URL读取用户的个人信息, 只需要获得先返回的结果即可。这种情况下, 用 `Promise.race()` 实现:

```
var p1 = new Promise(function (resolve, reject) {
    setTimeout(resolve, 500, 'P1');
});
var p2 = new Promise(function (resolve, reject) {
    setTimeout(resolve, 600, 'P2');
});
Promise.race([p1, p2]).then(function (result) {
    console.log(result); // 'P1'
});
```

由于 `p1` 执行较快, `Promise` 的 `then()` 将获得结果 `'P1'`。 `p2` 仍在继续执行, 但执行结果将被丢弃。

如果我们组合使用 `Promise`, 就可以把很多异步任务以并行和串行的方式组合起来执行。

`Canvas` 是 `HTML5` 新增的组件, 它就像一块幕布, 可以用 `JavaScript` 在上面绘制各种图表、动画等。

没有 `Canvas` 的年代, 绘图只能借助 `Flash` 插件实现, 页面不得不用 `JavaScript` 和 `Flash` 进行交互。有了 `Canvas`, 我们就再也不需要 `Flash` 了, 直接使用 `JavaScript` 完成绘制。

一个 `Canvas` 定义了一个指定尺寸的矩形框, 在这个范围内我们可以随意绘制:

```
<canvas id="test-canvas" width="300" height="200"></canvas>
```

由于浏览器对 `HTML5` 标准支持不一致, 所以, 通常在 `<canvas>` 内部添加一些说明性 `HTML` 代码, 如果浏览器支持 `Canvas`, 它将忽略 `<canvas>` 内部的 `HTML`, 如果浏览器不支持 `Canvas`, 它将显示 `<canvas>` 内部的 `HTML`:

```
<canvas id="test-stock" width="300" height="200">
    <p>Current Price: 25.51</p>
</canvas>
```

在使用 `Canvas` 前, 用 `canvas.getContext` 来测试浏览器是否支持 `Canvas`:

```
<!-- HTML代码 -->
<canvas id="test-canvas" width="200" height="100">
    <p>你的浏览器不支持Canvas</p>
</canvas>
```

```
'use strict';

var canvas = document.getElementById('test-canvas');
if (canvas.getContext) {
    alert('你的浏览器支持Canvas!');
} else {
    alert('你的浏览器不支持Canvas!');
}
```

Run

`getContext('2d')` 方法让我们拿到一个 `CanvasRenderingContext2D` 对象，所有的绘图操作都需要通过这个对象完成。

```
var ctx = canvas.getContext('2d');
```

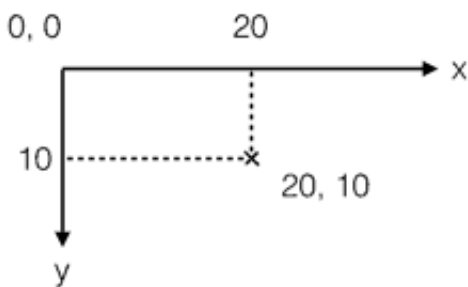
如果需要绘制3D怎么办？HTML5还有一个WebGL规范，允许在Canvas中绘制3D图形：

```
gl = canvas.getContext("webgl");
```

本节我们只专注于绘制2D图形。

绘制形状

我们可以在Canvas上绘制各种形状。在绘制前，我们需要先了解一下Canvas的坐标系统：



Canvas的坐标以左上角为原点，水平向右为X轴，垂直向下为Y轴，以像素为单位，所以每个点都是非负整数。

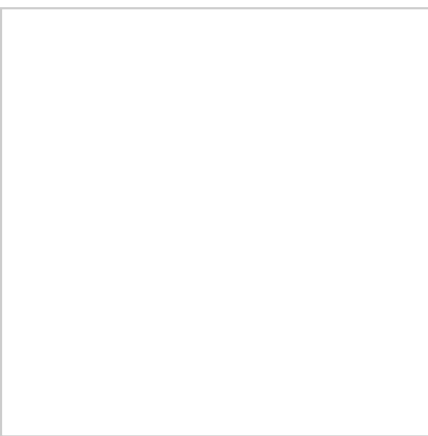
`CanvasRenderingContext2D` 对象有若干方法来绘制图形：

```
'use strict';

var
    canvas = document.getElementById('test-shape-canvas'),
    ctx = canvas.getContext('2d');

ctx.clearRect(0, 0, 200, 200); // 擦除(0,0)位置大小为200x200的矩形，擦除的意思是把该区域变为透明
ctx.fillStyle = '#dddddd'; // 设置颜色
ctx.fillRect(10, 10, 130, 130); // 把(10,10)位置大小为130x130的矩形涂色
// 利用Path绘制复杂路径：
var path=new Path2D();
path.arc(75, 75, 50, 0, Math.PI*2, true);
path.moveTo(110,75);
path.arc(75, 75, 35, 0, Math.PI, false);
path.moveTo(65, 65);
path.arc(60, 65, 5, 0, Math.PI*2, true);
path.moveTo(95, 65);
path.arc(90, 65, 5, 0, Math.PI*2, true);
ctx.strokeStyle = '#0000ff';
ctx.stroke(path);
```

Run



绘制文本

绘制文本就是在指定的位置输出文本，可以设置文本的字体、样式、阴影等，与CSS完全一致：

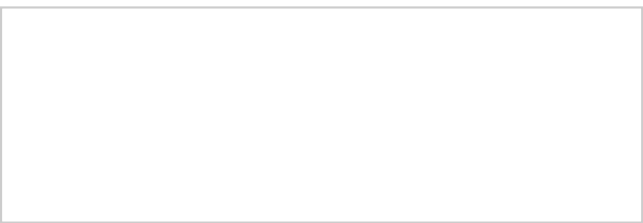
```
'use strict';

var
    canvas = document.getElementById('test-text-canvas'),
    ctx = canvas.getContext('2d');
```



```
ctx.clearRect(0, 0, canvas.width, canvas.height);
ctx.shadowOffsetX = 2;
ctx.shadowOffsetY = 2;
ctx.shadowBlur = 2;
ctx.shadowColor = '#666666';
ctx.font = '24px Arial';
ctx.fillStyle = '#333333';
ctx.fillText('带阴影的文字', 20, 40);
```

Run



Canvas除了能绘制基本的形状和文本，还可以实现动画、缩放、各种滤镜和像素转换等高级操作。如果要实现非常复杂的操作，考虑以下优化方案：

- 通过创建一个不可见的**Canvas**来绘图，然后将最终绘制结果复制到页面的可见**Canvas**中；
- 尽量使用整数坐标而不是浮点数；
- 可以创建多个重叠的**Canvas**绘制不同的层，而不是在一个**Canvas**中绘制非常复杂的图；
- 背景图片如果不变可以直接用 `` 标签并放到最底层。

练习

请根据从**163**获取的**JSON**数据绘制最近**30**个交易日的**K**线图，数据已处理为包含一组对象的数组：

```
use strict';
```

```
window.loadStockData = function (r) {  
    var  
        NUMS = 30,  
        data = r.data;  
    if (data.length > NUMS) {  
        data = data.slice(data.length - NUMS);  
    }  
    data = data.map(function (x) {  
        return {  
            date: x[0],  
            open: x[1],  
            close: x[2],  
            high: x[3],  
            low: x[4],  
            vol: x[5],  
            change: x[6]  
        };  
    });  
    window.drawStock(data);  
}
```

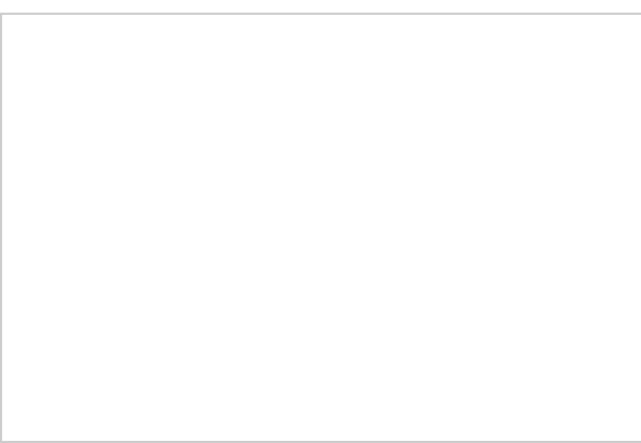
```
window.drawStock = function (data) {
```

```
    var  
        canvas = document.getElementById('stock-canvas'),  
        width = canvas.width,  
        height = canvas.height,  
        ctx = canvas.getContext('2d');  
    console.log(JSON.stringify(data[0])); //  
{"date":"20150602","open":4844.7,"close":4910.53,"high":4911.57,"low":4797.55,"vol":62374809  
900,"change":1.69}  
    ctx.clearRect(0, 0, width, height);  
    ctx.fillText('Test Canvas', 10, 10);  
};
```

// 加载最近30个交易日的K线图数据:

```
var js = document.createElement('script');  
js.src = 'http://img1.money.126.net/data/hs/kline/day/history/2015/0000001.json?callback=loadSto  
document.getElementsByTagName('head')[0].appendChild(js);
```

Run



[下载为图片](#)

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

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！

jQuery版本

目前jQuery有1.x和2.x两个主要版本，区别在于2.x移除了对古老的IE 6、7、8的支持，因此2.x的代码更精简。选择哪个版本主要取决于你是否想支持IE 6~8。

从[jQuery官网](#)可以下载最新版本。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

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>
  ...
</body>
</html>
```

好消息是，当你在学习这个教程时，由于网站本身已经引用了jQuery，所以你可以直接使用：

```
'use strict';
```

```
alert('jQuery版本: ' + $.fn.jquery);
```

Run

\$符号

`$` 是著名的jQuery符号。实际上，jQuery把所有功能全部封装在一个全局变量 `jQuery` 中，而 `$` 也是一个合法的变量名，它是变量 `jQuery` 的别名：

```
window.jQuery; // jQuery(selector, context)
window.$; // jQuery(selector, context)
$ === jQuery; // true
typeof($); // 'function'
```

`$` 本质上就是一个函数，但是函数也是对象，于是 `$` 除了可以直接调用外，也可以有很多其他属性。

注意，你看到的 `$` 函数名可能不是 `jQuery(selector, context)`，因为很多JavaScript压缩工具可以对函数名和参数改名，所以压缩过的jQuery源码 `$` 函数可能变成 `a(b, c)`。

绝大多数时候，我们都直接用 `$`（因为写起来更简单嘛）。但是，如果 `$` 这个变量不幸地被占用了，而且还不能改，那我们就只能让 `jQuery` 把 `$` 变量交出来，然后就只能使用 `jQuery` 这个变量：

```
$. // jQuery(selector, context)
jQuery.noConflict();
$. // undefined
jQuery; // jQuery(selector, context)
```

这种黑魔法的原理是jQuery在占用 `$` 之前，先在内部保存了原来的 `$`，调用 `jQuery.noConflict()` 时会把原来保存的变量还原。

选择器是jQuery的核心。一个选择器写出来类似 `$('#dom-id')`。

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，并且，在层级关系中，例如，查找 `<table class="green">` 里面的所有 `<tr>`，一层循环实际上是错的，因为 `<table>` 的标准写法是：

```
<table>
  <tbody>
    <tr>...</tr>
    <tr>...</tr>
  </tbody>
</table>
```

很多时候，需要递归查找所有子节点。

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');
```

注意，`#abc` 以 `#` 开头。返回的对象是jQuery对象。

什么是jQuery对象？jQuery对象类似数组，它的每个元素都是一个引用了DOM节点的对象。

以上面的查找为例，如果 `id` 为 `abc` 的 `<div>` 存在，返回的jQuery对象如下：

```
[<div id="abc">...</div>]
```

如果 `id` 为 `abc` 的 `<div>` 不存在，返回的jQuery对象如下：

```
[]
```

总之jQuery的选择器不会返回 `undefined` 或者 `null`，这样的好处是你不必在下一行判断 `if (div === undefined)`。

jQuery对象和DOM对象之间可以互相转化:

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

通常情况下你不需要获取DOM对象, 直接使用jQuery对象更加方便。如果你拿到了一个DOM对象, 那可以简单地调用 `$(aDomObject)` 把它变成jQuery对象, 这样就可以方便地使用jQuery的API了。

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>
```

通常很多节点有多个class, 我们可以查找同时包含 `red` 和 `green` 的节点:

```
var a = $('.red.green'); // 注意没有空格!
// 符合条件的节点:
// <div class="red green">...</div>
// <div class="blue green red">...</div>
```

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"
```

这个方法尤其适合通过class属性查找，且不受class包含多个名称的影响：

```
var icons = $(' [class^="icon-"]'); // 找出所有class包含至少一个以`icon-`开头的DOM  
// 例如: class="icon-clock", class="abc icon-home"
```

组合查找

组合查找就是把上述简单选择器组合起来使用。如果我们查找 `$('[name=email]')`，很可能把表单外的 `<div name="email">` 也找出来，但我们只希望查找 `<input>`，就可以这么写：

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

同样的，根据tag和class来组合查找也很常见：

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

多项选择器

多项选择器就是把多个选择器用 `,` 组合起来一块选：

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

要注意的是，选出来的元素是按照它们在HTML中出现的顺序排列的，而且不会有重复元素。例如，`<p class="red green">` 不会被上面的 `$('p.red,p.green')` 选择两次。

练习

使用jQuery选择器分别选出指定元素：

- 仅选择JavaScript
- 仅选择Erlang
- 选择JavaScript和Erlang
- 选择所有编程语言
- 选择名字input
- 选择邮件和名字input

```
<!-- HTML结构 -->
<div id="test-jquery">
  <p id="para-1" class="color-red">JavaScript</p>
  <p id="para-2" class="color-green">Haskell</p>
  <p class="color-red color-green">Erlang</p>
  <p name="name" class="color-black">Python</p>
  <form class="test-form" target="_blank" action="#0" onsubmit="return false;">
    <legend>注册新用户</legend>
    <fieldset>
      <p><label>名字: <input name="name"></label></p>
      <p><label>邮件: <input name="email"></label></p>
      <p><label>口令: <input name="password" type="password"></label></p>
      <p><button type="submit">注册</button></p>
    </fieldset>
  </form>
</div>
```

运行查看结果:

```
'use strict';

var selected = null;

selected = ???;

// 高亮结果:
if (!(selected instanceof jQuery)) {
  return alert('不是有效的jQuery对象!');
}
$('#test-jquery').find('*').css('background-color', '');
selected.css('background-color', '#ffd351');
```

Run

JavaScript

Haskell

Erlang

Python

注册新用户

名字:

邮件:

口令:

除了基本的选择器外，jQuery的层级选择器更加灵活，也更强大。

因为DOM的结构就是层级结构，所以我们经常要根据层级关系进行选择。

层级选择器（Descendant Selector）

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

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

要选出JavaScript，可以用层级选择器：

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

因为 `<div>` 和 `` 都是 `` 的祖先节点，所以上面两种方式都可以选出相应的 `` 节点。

要选择所有的 `` 节点，用：

```
$('.ul.lang li');
```

这种层级选择器相比单个的选择器好处在于，它缩小了选择范围，因为首先要定位父节点，才能选择相应的子节点，这样避免了页面其他不相关的元素。

例如：

```
$('.form[name=upload] input');
```

就把选择范围限定在 `name` 属性为 `upload` 的表单里。如果页面有很多表单，其他表单的 `<input>` 不会被选择。

多层选择也是允许的：

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

子选择器（Child Selector）

选择器 `$('parent>child')` 类似层级选择器，但是限定了层级关系必须是父子关系，就是 `<child>` 节点必须是 `<parent>` 节点的直属子节点。还是以上面的例子：

```
$( 'ul.lang>li.lang-javascript' ); // 可以选出[<li class="lang-javascript">JavaScript</li>]  
$( 'div.testing>li.lang-javascript' ); // [], 无法选出，因为<div>和<li>不构成父子关系
```

过滤器 (Filter)

过滤器一般不单独使用，它通常附加在选择器上，帮助我们更精确地定位元素。观察过滤器的效果：

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

表单相关

针对表单元素，jQuery还有一组特殊的选择器：

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

此外，jQuery还有很多有用的选择器，例如，选出可见的或隐藏的元素：

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

练习

针对如下HTML结构：

<!-- HTML结构 -->

```
<div class="test-selector">
  <ul class="test-lang">
    <li class="lang-javascript">JavaScript</li>
    <li class="lang-python">Python</li>
    <li class="lang-lua">Lua</li>
  </ul>
  <ol class="test-lang">
    <li class="lang-swift">Swift</li>
    <li class="lang-java">Java</li>
    <li class="lang-c">C</li>
  </ol>
</div>
```

选出相应的内容并观察效果：

```
'use strict';
var selected = null;

// 分别选择所有语言，所有动态语言，所有静态语言，JavaScript，Lua，C等：
selected = ???

// 高亮结果：
if (!(selected instanceof jQuery)) {
  return alert('不是有效的jQuery对象!');
}
$('#test-jquery').find('*').css('background-color', '');
selected.css('background-color', '#ffd351');
```

Run

- JavaScript
 - Python
 - Lua
1. Swift
 2. Java
 3. C

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

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

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

用 `find()` 查找：

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

如果要从当前节点开始向上查找，使用 `parent()` 方法：

```
var swf = $('#swift'); // 获得Swift
var parent = swf.parent(); // 获得Swift的上层节点<ul>
var a = swf.parent('div.red'); // 从Swift的父节点开始向上查找，直到找到某个符合条件的节点并返回
```

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

当我们已经拿到 `Swift` 节点后：

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

swift.next(); // Scheme
swift.next('[name=haskell]'); // Haskell，因为Haskell是后续第一个符合选择器条件的节点

swift.prev(); // Python
swift.prev('.js'); // JavaScript，因为JavaScript是往前第一个符合选择器条件的节点
```

过滤

和函数式编程的`map`、`filter`类似，`jQuery`对象也有类似的方法。

`filter()` 方法可以过滤掉不符合选择器条件的节点：

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

或者传入一个函数，要特别注意函数内部的 `this` 被绑定为DOM对象，不是`jQuery`对象：

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

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

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

此外，一个jQuery对象如果包含了不止一个DOM节点，`first()`、`last()`和`slice()`方法可以返回一个新的jQuery对象，把不需要的DOM节点去掉：

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

练习

对于下面的表单：

```
<form id="test-form" action="#0" onsubmit="return false;">
  <p><label>Name: <input name="name"></label></p>
  <p><label>Email: <input name="email"></label></p>
  <p><label>Password: <input name="password" type="password"></label></p>
  <p>Gender: <label><input name="gender" type="radio" value="m" checked> Male</label> <label><input name="gender" type="radio" value="f" checked> Female</label></p>
  <p><label>City: <select name="city">
    <option value="BJ" selected>Beijing</option>
    <option value="SH">Shanghai</option>
    <option value="CD">Chengdu</option>
    <option value="XM">Xiamen</option>
  </select></label></p>
  <p><button type="submit">Submit</button></p>
</form>
```

输入值后，用jQuery获取表单的JSON字符串，**key**和**value**分别对应每个输入的**name**和相应的**value**，例如：`{"name":"Michael","email":...}`

```
'use strict';
var json = null;

json = ???;
```

```
// 显示结果:
if (typeof(json) === 'string') {
    alert(json);
}
else {
    alert('json变量不是string!');
}
```

Run

Name:

Email:

Password:

Gender: ☒ Male ☐ Female

City:

Submit

jQuery的选择器很强大，用起来又简单又灵活，但是搞了这么久，我拿到了jQuery对象，到底要干什么？

答案当然是操作对应的DOM节点啦！

回顾一下修改DOM的CSS、文本、设置HTML有多么麻烦，而且有的浏览器只有innerHTML，有的浏览器支持innerText，有了jQuery对象，不需要考虑浏览器差异了，全部统一操作！

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 & 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 & 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：

```
'use strict';
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');
```

Run

- JavaScript
- Java & JavaScript

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？
```

所以jQuery对象的另一个好处是我们可以执行一个操作，作用在对应的一组DOM节点上。即使选择器没有返回任何DOM节点，调用jQuery对象的方法仍然不会报错：

```
// 如果不存在id为not-exist的节点：  
$('#not-exist').text('Hello'); // 代码不报错，没有节点被设置为'Hello'
```

这意味着jQuery帮你免去了许多if语句。

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：

```
'use strict';
```

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

Run

- JavaScript
- Java
- Python
- Swift
- Scheme

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
```

为了和JavaScript保持一致，CSS属性可以用`'background-color'`和`'backgroundColor'`两种格式。

`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
```

练习：分别用`css()`方法和`addClass()`方法高亮显示JavaScript：


```
<!-- HTML结构 -->
<style>
.highlight {
    color: #dd1144;
    background-color: #ffd351;
}
</style>

<div id="test-highlight-css">
    <ul>
        <li class="py"><span>Python</span></li>
        <li class="js"><span>JavaScript</span></li>
        <li class="sw"><span>Swift</span></li>
        <li class="hk"><span>Haskell</span></li>
    </ul>
</div>
```

```
'use strict';
```

```
var div = $('#test-highlight-css');
// TODO:
```

Run

- Python
- JavaScript
- Swift
- Haskell

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。

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.hasAttr('name'); // true
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
```

`prop()` 方法和 `attr()` 类似，但是HTML5规定有一种属性在DOM节点中可以没有值，只有出现与不出现两种，例如：

```
<input id="test-radio" type="radio" name="test" checked value="1">
```

等价于：

```
<input id="test-radio" type="radio" name="test" checked="checked" value="1">
```

`attr()` 和 `prop()` 对于属性 `checked` 处理有所不同：

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

`prop()` 返回值更合理一些。不过，用 `is()` 方法判断更好：

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

类似的属性还有 `selected`，处理时最好用 `is(':selected')`。

对于表单元素，jQuery对象还提供了 `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。

直接使用浏览器提供的API对DOM结构进行修改，不但代码复杂，而且要针对浏览器写不同的代码。

有了jQuery，我们就专注于操作jQuery对象本身，底层的DOM操作由jQuery完成就可以了，这样一来，修改DOM也大大简化了。

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 `` 节点：

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

然后，调用 `append()` 传入HTML片段：

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

除了接受字符串，`append()` 还可以传入原始的DOM对象，jQuery对象和函数对象：

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

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

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

传入函数时，要求返回一个字符串、DOM对象或者jQuery对象。因为jQuery的 `append()` 可能作用于一组DOM节点，只有传入函数才能针对每个DOM生成不同的子节点。

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

另外注意，如果要添加的DOM节点已经存在于HTML文档中，它会首先从文档移除，然后再添加，也就是说，用 `append()`，你可以移动一个DOM节点。

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
```

练习

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

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

```
use strict';
```

```
// 测试：
;(function () {
    var s = $('#test-div>ul>li').map(function () {
        return $(this).text();
    }).get().join(',');
    if (s === 'JavaScript,Lua,Pascal,Python,Ruby,Swift') {
        alert('测试通过!');
    } else {
        alert('测试失败： ' + s);
    }
})();
```

Run

- JavaScript
- Python
- Swift

因为**JavaScript**在浏览器中以单线程模式运行，页面加载后，一旦页面上所有的**JavaScript**代码被执行完后，就只能依赖触发事件来执行**JavaScript**代码。

浏览器在接收到用户的鼠标或键盘输入后，会自动在对应的**DOM**节点上触发相应的事件。如果该节点已经绑定了对应的**JavaScript**处理函数，该函数就会自动调用。

由于不同的浏览器绑定事件的代码都不太一样，所以用**jQuery**来写代码，就屏蔽了不同浏览器的差异，我们总是编写相同的代码。

举个例子，假设要在用户点击了超链接时弹出提示框，我们用**jQuery**这样绑定一个 `click` 事件：

```
/* HTML：
 *
 * <a id="test-link" href="#0">点我试试</a>
 *
 */

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

案例：[点击试试](#)

`on` 方法用来绑定一个事件，我们需要传入事件名称和对应的处理函数。

另一种更简化的写法是直接调用 `click()` 方法：

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

两者完全等价。我们通常用后面的写法。

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 `<input>` 和 `<textarea>`。

keydown: 键盘按下时触发； **keyup**: 键盘松开时触发； **keypress**: 按一次键后触发。

其他事件

focus: 当DOM获得焦点时触发； **blur**: 当DOM失去焦点时触发； **change**: 当 `<input>`、`<select>` 或 `<textarea>` 的内容改变时触发； **submit**: 当 `<form>` 提交时触发； **ready**: 当页面被载入并且DOM树完成初始化后触发。

其中，`ready` 仅作用于 `document` 对象。由于 `ready` 事件在DOM完成初始化后触发，且只触发一次，所以非常适合用来写其他的初始化代码。假设我们想给一个 `<form>` 表单绑定 `submit` 事件，下面的代码没有预期的效果：

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

因为JavaScript在此执行的时候，`<form>` 尚未载入浏览器，所以 `$('#testForm')` 返回 `[]`，并没有绑定事件到任何DOM上。

所以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!');
  });
});

```

甚至可以再简化为：

```

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

```

上面的这种写法最为常见。如果你遇到 `$(function () {...})` 的形式，牢记这是 `document` 对象的 `ready` 事件处理函数。

完全可以反复绑定事件处理函数，它们会依次执行：

```

$(function () {
  console.log('init A...');
});
$(function () {
  console.log('init B...');
});
$(function () {
  console.log('init C...');
});

```

事件参数

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

```
$(function () {
    $('#testMouseMoveDiv').mousemove(function (e) {
        $('#testMouseMoveSpan').text('pageX = ' + e.pageX + ', pageY = ' + e.pageY);
    });
});
```

效果实测：

mousemove:



取消绑定

一个已被绑定的事件可以解除绑定，通过 `off('click', function)` 实现：

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

a.click(hello); // 绑定事件

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

需要特别注意的是，下面这种写法是无效的：

```
// 绑定事件：
a.click(function () {
    alert('hello!');
});

// 解除绑定：
a.off('click', function () {
    alert('hello!');
});
```

这是因为两个匿名函数虽然长得一模一样，但是它们是两个不同的函数对象，`off('click', function () {...})` 无法移除已绑定的第一个匿名函数。

为了实现移除效果，可以使用 `off('click')` 一次性移除已绑定的 `click` 事件的所有处理函数。

同理，无参数调用 `off()` 一次性移除已绑定的所有类型的事件处理函数。

事件触发条件

需要注意的是，事件的触发总是由用户操作引发的。例如，我们监控文本框的内容改动：

```
var input = $('#test-input');
input.change(function () {
    console.log('changed...');
});
```

当用户在文本框中输入时，就会触发 `change` 事件。但是，如果用JavaScript代码去改动文本框的值，将不会触发 `change` 事件：

```
var input = $('#test-input');
input.val('change it!'); // 无法触发change事件
```

有些时候，我们希望用代码触发 `change` 事件，可以直接调用无参数的 `change()` 方法来触发该事件：

```
var input = $('#test-input');
input.val('change it!');
input.change(); // 触发change事件
```

`input.change()` 相当于 `input.trigger('change')`，它是 `trigger()` 方法的简写。

为什么我们希望手动触发一个事件呢？如果不这么做，很多时候，我们就得写两份一模一样的代码。

浏览器安全限制

在浏览器中，有些JavaScript代码只有在用户触发下才能执行，例如，`window.open()` 函数：

```
// 无法弹出新窗口，将被浏览器屏蔽：
$(function () {
    window.open('/');
});
```

这些“敏感代码”只能由用户操作来触发：

```
var button1 = $('#testPopupButton1');
var button2 = $('#testPopupButton2');

function popupTestWindow() {
    window.open('/');
}

button1.click(function () {
    popupTestWindow();
});

button2.click(function () {
    // 不立刻执行popupTestWindow()，100毫秒后执行：
    setTimeout(popupTestWindow, 100);
});
```

当用户点击 `button1` 时，`click` 事件被触发，由于 `popupTestWindow()` 在 `click` 事件处理函数内执行，这是浏览器允许的，而 `button2` 的 `click` 事件并未立刻执行 `popupTestWindow()`，延迟执行的 `popupTestWindow()` 将被浏览器拦截。

需求实现:

Button 1

Button 2

练习

对如下的Form表单:

```
<!-- HTML结构 -->
<form id="test-form" action="test">
  <legend>请选择想要学习的编程语言: </legend>
  <fieldset>
    <p><label class="selectAll"><input type="checkbox"> <span class="selectAll">全选</span><span class="deselectAll">全不选</span></p>
    <p><label><input type="checkbox" name="lang" value="javascript"> JavaScript</label></p>
    <p><label><input type="checkbox" name="lang" value="python"> Python</label></p>
    <p><label><input type="checkbox" name="lang" value="ruby"> Ruby</label></p>
    <p><label><input type="checkbox" name="lang" value="haskell"> Haskell</label></p>
    <p><label><input type="checkbox" name="lang" value="scheme"> Scheme</label></p>
    <p><button type="submit">Submit</button></p>
  </fieldset>
</form>
```

绑定合适的事件处理函数, 实现以下逻辑:

当用户勾选“全选”时, 自动选中所有语言, 并把“全选”变成“全不选”;

当用户去掉“全不选”时, 自动不选中所有语言;

当用户点击“反选”时, 自动把所有语言状态反转(选中的变为未选, 未选的变为选中);

当用户把所有语言都手动勾选上时, “全选”被自动勾选上, 并变为“全不选”;

当用户手动去掉选中至少一种语言时, “全不选”自动被去掉选中, 并变为“全选”。

```
'use strict';

var
  form = $('#test-form'),
  langs = form.find('[name=lang]'),
  selectAll = form.find('label.selectAll :checkbox'),
  selectAllLabel = form.find('label.selectAll span.selectAll'),
  deselectAllLabel = form.find('label.selectAll span.deselectAll'),
  invertSelect = form.find('a.invertSelect');

// 重置初始化状态:
form.find('*').show().off();
form.find(':checkbox').prop('checked', false).off();
deselectAllLabel.hide();
// 拦截form提交事件:
form.off().submit(function (e) {
  e.preventDefault();
  alert(form.serialize());
});
```

// TODO: 绑定事件

```
// 测试：  
alert('请测试功能是否正常。');
```

Run

请选择想要学习的编程语言：

☐ 全选 ☐ 全不选 [反选](#)

☐ JavaScript

☐ Python

☐ Ruby

☐ Haskell

☐ Scheme

Submit

用JavaScript实现动画，原理非常简单：我们只需要以固定的时间间隔（例如，0.1秒），每次把DOM元素的CSS样式修改一点（例如，高宽各增加10%），看起来就像动画了。

但是要用JavaScript手动实现动画效果，需要编写非常复杂的代码。如果想要把动画效果用函数封装起来便于复用，那考虑的事情就更多了。

使用jQuery实现动画，代码已经简单得不能再简化了：只需要一行代码！

让我们先来看看jQuery内置的几种动画样式：

show / hide

直接以无参数形式调用 `show()` 和 `hide()`，会显示和隐藏DOM元素。但是，只要传递一个时间参数进去，就变成了动画：

```
var div = $('#test-show-hide');  
div.hide(3000); // 在3秒钟内逐渐消失
```

时间以毫秒为单位，但也可以是 `'slow'`，`'fast'` 这些字符串：

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

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

效果实测：

hide('slow')

show('slow')

toggle('slow')

slideUp / slideDown

你可能已经看出来了，`show()` 和 `hide()` 是从左上角逐渐展开或收缩的，而 `slideUp()` 和 `slideDown()` 则是在垂直方向逐渐展开或收缩的。

`slideUp()` 把一个可见的DOM元素收起来，效果跟拉上窗帘似的，`slideDown()` 相反，而 `slideToggle()` 则根据元素是否可见来决定下一步动作：

```
var div = $('#test-slide');  
div.slideUp(3000); // 在3秒钟内逐渐向上消失
```

效果实测：

slideUp('slow')

slideDown('slow')

slideToggle('slow')

fadeIn / fadeOut

`fadeIn()` 和 `fadeOut()` 的动画效果是淡入淡出，也就是通过不断设置DOM元素的 `opacity` 属性来实现，而 `fadeToggle()` 则根据元素是否可见来决定下一步动作：

```
var div = $('#test-fade');  
div.fadeOut('slow'); // 在0.6秒内淡出
```

`fadeOut('slow')``fadeIn('slow')``fadeToggle('slow')`

自定义动画

如果上述动画效果还不能满足你的要求，那就祭出最后大招：`animate()`，它可以实现任意动画效果，我们需要传入的参数就是DOM元素最终的CSS状态和时间，jQuery在时间段内不断调整CSS直到达到我们设定的值：

```
var div = $('#test-animate');
div.animate({
  opacity: 0.25,
  width: '256px',
  height: '256px'
}, 3000); // 在3秒钟内CSS过渡到设定值
```

`animate()` 还可以再传入一个函数，当动画结束时，该函数将被调用：

```
var div = $('#test-animate');
div.animate({
  opacity: 0.25,
  width: '256px',
  height: '256px'
}, 3000, function () {
  console.log('动画已结束');
  // 恢复至初始状态：
  $(this).css('opacity', '1.0').css('width', '128px').css('height', '128px');
});
```

实际上这个回调函数参数对于基本动画也是适用的。

有了`animate()`，你就可以实现各种自定义动画效果了：

`animate()`

串行动画

jQuery的动画效果还可以串行执行，通过 `delay()` 方法还可以实现暂停，这样，我们可以实现更复杂的动画效果，而代码却相当简单：

```
var div = $('#test-animates');
// 动画效果：slideDown - 暂停 - 放大 - 暂停 - 缩小
div.slideDown(2000)
  .delay(1000)
  .animate({
    width: '256px',
    height: '256px'
  }, 2000)
  .delay(1000)
  .animate({
    width: '128px',
    height: '128px'
  }, 2000);
</script>
```

因为动画需要执行一段时间，所以jQuery必须不断返回新的Promise对象才能后续执行操作。简单地把动画封装在函数中是不够的。

效果实测：

animate

为什么有的动画没有效果

你可能会遇到，有的动画如 `slideUp()` 根本没有效果。这是因为jQuery动画的原理是逐渐改变CSS的值，如 `height` 从 `100px` 逐渐变为 `0`。但是很多不是block性质的DOM元素，对它们设置 `height` 根本就不起作用，所以动画也就没有效果。

此外，jQuery也没有实现对 `background-color` 的动画效果，用 `animate()` 设置 `background-color` 也没有效果。这种情况下可以使用CSS3的 `transition` 实现动画效果。

练习

在执行删除操作时，给用户显示一个动画比直接调用 `remove()` 要更好。请在表格删除一行时添加一个淡出的动画效果：

```
'use strict';

function deleteFirstTR() {
    var tr = $('#test-table>tbody>tr:visible').first();

    // ...

}

deleteFirstTR();
```

Run

Name	Email	Address	Status
Bart Simpson	bart.s@primary.school	Springfield	Active
Michael Scofield	m.scofield@escape.org	Fox River	Locked
Optimus Prime	prime@cybertron.org	Cybertron	Active
Peter Parker	spider@movie.org	New York	Active
Thor Odinson	thor@asgard.org	Asgard	Active

Add

用JavaScript写AJAX前面已经介绍过了，主要问题就是不同浏览器需要写不同代码，并且状态和错误处理写起来很麻烦。

用jQuery的相关对象来处理AJAX，不但不需要考虑浏览器问题，代码也能大大简化。

ajax

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

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

下面的例子发送一个GET请求，并返回一个JSON格式的数据：

```
var jqxhr = $.ajax('/api/categories', {
    dataType: 'json'
});
// 请求已经发送了
```

不过，如何用回调函数处理返回的数据和出错时的响应呢？

还记得Promise对象吗？jQuery的jqXHR对象类似一个Promise对象，我们可以用链式写法来处理各种回调：

```
'use strict';

function ajaxLog(s) {
    var txt = $('#test-response-text');
    txt.val(txt.val() + '\n' + s);
}

$('#test-response-text').val('');

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

Run

响应结果：

get

对常用的AJAX操作，jQuery提供了一些辅助方法。由于GET请求最常见，所以jQuery提供了`get()`方法，可以这么写：

```
var jqxhr = $.get('/path/to/resource', {
  name: 'Bob Lee',
  check: 1
});
```

第二个参数如果是object，jQuery自动把它变成query string然后加到URL后面，实际的URL是：

```
/path/to/resource?name=Bob%20Lee&check=1
```

这样我们就不用关心如何用URL编码并构造一个query string了。

post

`post()`和`get()`类似，但是传入的第二个参数默认被序列化为`application/x-www-form-urlencoded`：

```
var jqxhr = $.post('/path/to/resource', {
  name: 'Bob Lee',
  check: 1
});
```

实际构造的数据`name=Bob%20Lee&check=1`作为POST的body被发送。

getJSON

由于JSON用得越来越普遍，所以jQuery也提供了`getJSON()`方法来快速通过GET获取一个JSON对象：

```
var jqxhr = $.getJSON('/path/to/resource', {
  name: 'Bob Lee',
  check: 1
}).done(function (data) {
  // data已经被解析为JSON对象了
});
```

安全限制

jQuery的AJAX完全封装的是JavaScript的AJAX操作，所以它的安全限制和前面讲的用JavaScript写AJAX完全一样。

如果需要使用JSON，可以在 `ajax()` 中设置 `jsonp: callback`，让jQuery实现JSON跨域加载数据。

关于跨域的设置请参考[浏览器 - AJAX](#)一节中CORS的设置。

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

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

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

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

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

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

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

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

编写jQuery插件

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

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

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

对于如下的HTML结构：

```
<!-- HTML结构 -->
<div id="test-highlight1">
    <p>什么是<span>jQuery</span></p>
    <p><span>jQuery</span>是目前最流行的<span>JavaScript</span>库。</p>
</div>
```

来测试一下 `highlight1()` 的效果：

```
'use strict';
```

```
$( '#test-highlight1 span').highlight1();
```

Run

什么是jQuery

jQuery是目前最流行的JavaScript库。

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

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

不然，用户调用的时候，就不得不把上面的代码拆成两行。

但是这个版本并不完美。有的用户希望高亮的颜色能自己来指定，怎么办？

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

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

对于如下HTML结构：

```
<!-- HTML结构 -->  
<div id="test-highlight2">  
    <p>什么是<span>jQuery</span> <span>Plugin</span></p>  
    <p>编写<span>jQuery</span> <span>Plugin</span>可以用来扩展<span>jQuery</span>的功能。</p>  
</div>
```

来实测一下带参数的 `highlight2()`：

```
'use strict';
```

```
$( '#test-highlight2 span' ).highlight2({
  backgroundColor: '#00a8e6',
  color: '#ffffff'
});
```

Run

什么是jQuery Plugin

编写jQuery Plugin可以用来扩展jQuery的功能。

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

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

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

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

也就是说，我们设定的默认值应该能允许用户修改。

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

于是最终版的 `highlight()` 终于诞生了：

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

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

默认用户终于满意了。用户使用的时候，无需一次性设定默认值：

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

然后就可以非常简单地调用 `highlight()` 了。

对如下的HTML结构：

```
<!-- HTML结构 -->  
<div id="test-highlight">  
  <p>如何编写<span>jQuery</span> <span>Plugin</span></p>  
  <p>编写<span>jQuery</span> <span>Plugin</span>，要设置<span>默认值</span>，并允许用户修改<span>默认值</span>，或者运行  
</div>
```

实测一下修改默认值的效果：

```
'use strict';  
  
$.fn.highlight.defaults.color = '#659f13';  
$.fn.highlight.defaults.backgroundColor = '#f2fae3';  
  
$('#test-highlight p:first-child span').highlight();  
  
$('#test-highlight p:last-child span').highlight({  
  color: '#dd1144'  
});
```

Run

如何编写jQuery Plugin

编写jQuery Plugin，要设置默认值，并允许用户修改默认值，或者运行时传入其他值。

最终，我们得出编写一个jQuery插件的原则：

1. 给 `$.fn` 绑定函数，实现插件的代码逻辑；
2. 插件函数最后要 `return this;` 以支持链式调用；
3. 插件函数要有默认值，绑定在 `$.fn.<pluginName>.defaults` 上；
4. 用户在调用时可传入设定值以便覆盖默认值。

针对特定元素的扩展

我们知道jQuery对象的有些方法只能作用在特定DOM元素上，比如 `submit()` 方法只能针对 `form`。如果我们编写的扩展只能针对某些类型的DOM元素，应该怎么写？

还记得jQuery的选择器支持 `filter()` 方法来过滤吗？我们可以借助这个方法来实现针对特定元素的扩展。

例子，现在我们要给所有指向外链的超链接加上跳转提示，怎么做：

先写出用户调用的代码：

```
$('#main a').external();
```

然后按照上面的方法编写一个 `external` 扩展：

```
$.fn.external = function () {  
    // return返回的each()返回结果，支持链式调用：  
    return this.filter('a').each(function () {  
        // 注意：each()内部的回调函数的this绑定为DOM本身！  
        var a = $(this);  
        var url = a.attr('href');  
        if (url && (url.indexOf('http://')==0 || url.indexOf('https://')==0)) {  
            a.attr('href', '#0')  
            .removeAttr('target')  
            .append(' <i class="uk-icon-external-link"></i>')  
            .click(function () {  
                if(confirm(' 你确定要前往' + url + '? ')) {  
                    window.open(url);  
                }  
            });  
        }  
    });  
};
```

对如下的HTML结构：

```
<!-- HTML结构 -->  
<div id="test-external">  
    <p>如何学习<a href="http://jquery.com">jQuery</a>? </p>  
    <p>首先，你要学习<a href="/wiki/001434446689867b27157e896e74d51a89c25cc8b43bdb3000">JavaScript</a>，并了解基本的<a href="http://www.w3schools.com/js/default.asp">JavaScript教程</a>。</p>  
</div>
```

实测外链效果：

```
'use strict';  
  
$('#test-external a').external();
```

Run

如何学习jQuery?

首先，你要学习JavaScript，并了解基本的HTML。

小结

扩展jQuery对象的功能十分简单，但是我们要遵循jQuery的原则，编写的扩展方法能支持链式调用、具备默认值和过滤特定元素，使得扩展方法看上去和jQuery本身的方法没有什么区别。

从本章开始，我们就正式开启JavaScript的后端开发之旅。

Node.js是目前非常火热的技术，但是它的诞生经历却很奇特。

众所周知，在Netscape设计出JavaScript后的短短几个月，JavaScript事实上已经是前端开发的唯一标准。

后来，微软通过IE击败了Netscape后一统桌面，结果几年时间，浏览器毫无进步。（2001年推出的古老的IE 6到今天仍然有人在使用！）

没有竞争就没有发展。微软认为IE6浏览器已经非常完善，几乎没有可改进之处，然后解散了IE6开发团队！而Google却认为支持现代Web应用的新一代浏览器才刚刚起步，尤其是浏览器负责运行JavaScript的引擎性能还可提升10倍。

先是Mozilla借助已壮烈牺牲的Netscape遗产在2002年推出了Firefox浏览器，紧接着Apple于2003年在开源的KHTML浏览器的基础上推出了WebKit内核的Safari浏览器，不过仅限于Mac平台。

随后，Google也开始创建自家的浏览器。他们也看中了WebKit内核，于是基于WebKit内核推出了Chrome浏览器。

Chrome浏览器是跨Windows和Mac平台的，并且，Google认为要运行现代Web应用，浏览器必须有一个性能非常强劲的JavaScript引擎，于是Google自己开发了一个高性能JavaScript引擎，名字叫V8，以BSD许可证开源。

现代浏览器大战让微软的IE浏览器远远地落后了，因为他们解散了最有经验、战斗力最强的浏览器团队！回过头再追赶却发现，支持HTML5的WebKit已经成为手机端的标准了，IE浏览器从此与主流移动设备绝缘。

浏览器大战和Node有何关系？

话说有个叫Ryan Dahl的歪果仁，他的工作是用C/C++写高性能Web服务。对于高性能，异步IO、事件驱动是基本原则，但是用C/C++写就太痛苦了。于是这位仁兄开始设想用高级语言开发Web服务。他评估了很多种高级语言，发现很多语言虽然同时提供了同步IO和异步IO，但是开发人员一旦用了同步IO，他们就再也懒得写异步IO了，所以，最终，Ryan瞄向了JavaScript。

因为JavaScript是单线程执行，根本不能进行同步IO操作，所以，JavaScript的这一“缺陷”导致了它只能使用异步IO。

选定了开发语言，还要有运行时引擎。这位仁兄曾考虑过自己写一个，不过明智地放弃了，因为V8就是开源的JavaScript引擎。让Google投资去优化V8，咱只负责改造一下拿来用，还不用付钱，这个买卖很划算。

于是在2009年，Ryan正式推出了基于JavaScript语言和V8引擎的开源Web服务器项目，命名为Node.js。虽然名字很土，但是，Node第一次把JavaScript带入到后端服务器开发，加上世界上已经有无数的JavaScript开发人员，所以Node一下子就火了起来。

在Node上运行的JavaScript相比其他后端开发语言有何优势？

最大的优势是借助JavaScript天生的事件驱动机制加V8高性能引擎，使编写高性能Web服务轻而易举。

误，JavaScript语言本身是纯粹的函数式语言，在前端开发时，开发人员往往与它比较随意，让人感觉JavaScript就是个“玩具语言”。但是，在Node环境下，通过模块化的JavaScript代码，加上函数式编程，并且无需考虑浏览器兼容性问题，直接使用最新的ECMAScript 6标准，可以完全满足工程上的需求。

我还听说过io.js，这又是什么鬼？

因为Node.js是开源项目，虽然由社区推动，但幕后一直由Joyent公司资助。由于一群开发者对Joyent公司的策略不满，于2014年从Node.js项目fork出了io.js项目，决定单独发展，但两者实际上是兼容的。

然而中国有句古话，叫做“分久必合，合久必分”。分家后没多久，Joyent公司表示要和解，于是，io.js项目又决定回归Node.js。

具体做法是将来io.js将首先添加新的特性，如果大家测试用得爽，就把新特性加入Node.js。io.js是“尝鲜版”，而Node.js是线上稳定版，相当于Fedora Linux和RHEL的关系。

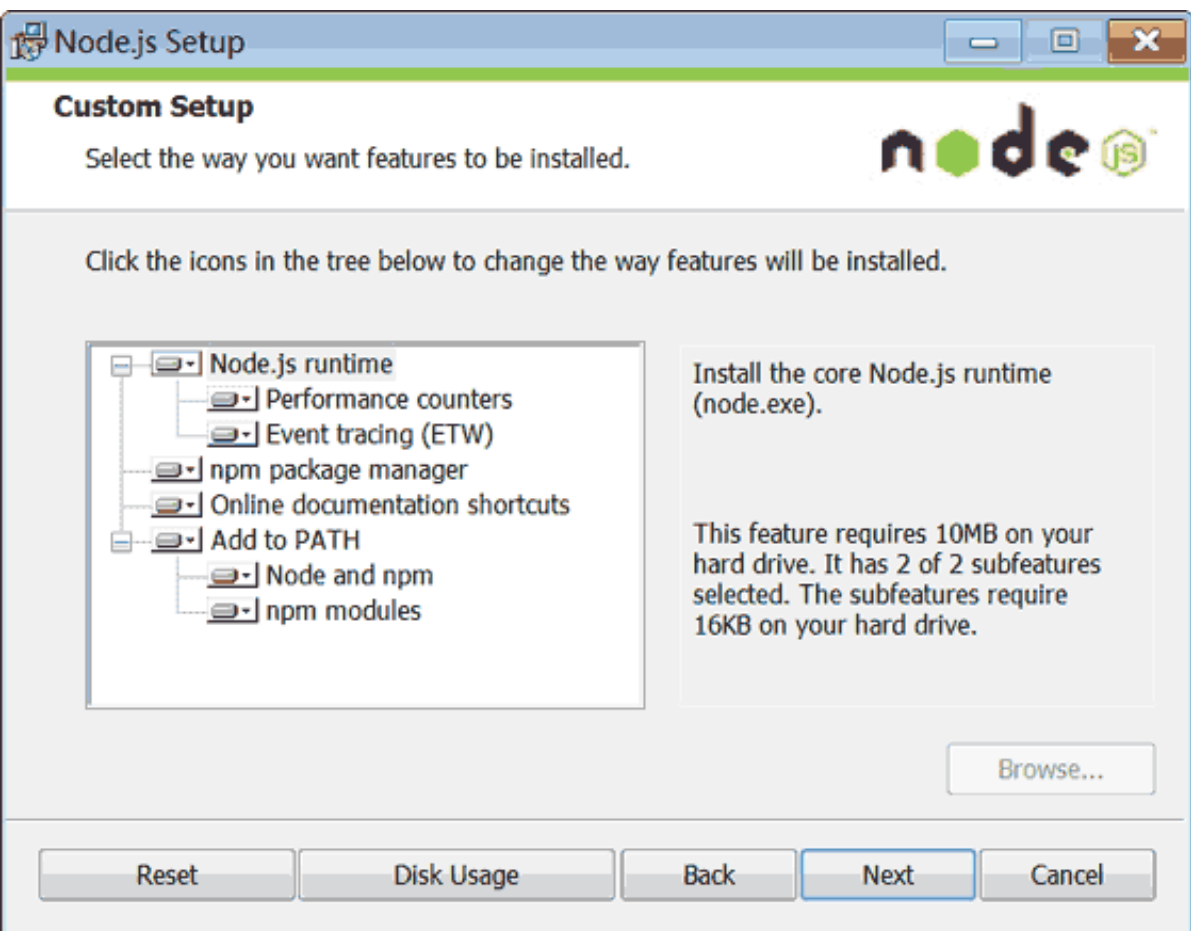
本章教程的所有代码都在Node.js上调试通过。如果你要尝试io.js也是可以的，不过两者如果遇到一些区别请自行查看io.js的文档。

由于Node.js平台是在后端运行JavaScript代码，所以，必须首先在本机安装Node环境。

安装Node.js

目前Node.js的最新版本是0.12.x。首先，从[Node.js官网](#)下载对应平台的安装程序，网速慢的童鞋请移步[国内镜像](#)。

在Windows上安装时务必选择全部组件：



安装完成后，在Windows环境下，请打开命令提示符，然后输入 `node -v`：



```
命令提示符
Microsoft Windows [版本 6.1.7601]
版权所有 (c) 2009 Microsoft Corporation。保留所有权利。

C:\Users\Michael>node -v
v0.12.7

C:\Users\Michael>
```

如果安装正常，你应该看到 `v0.12.7` 这样的输出。

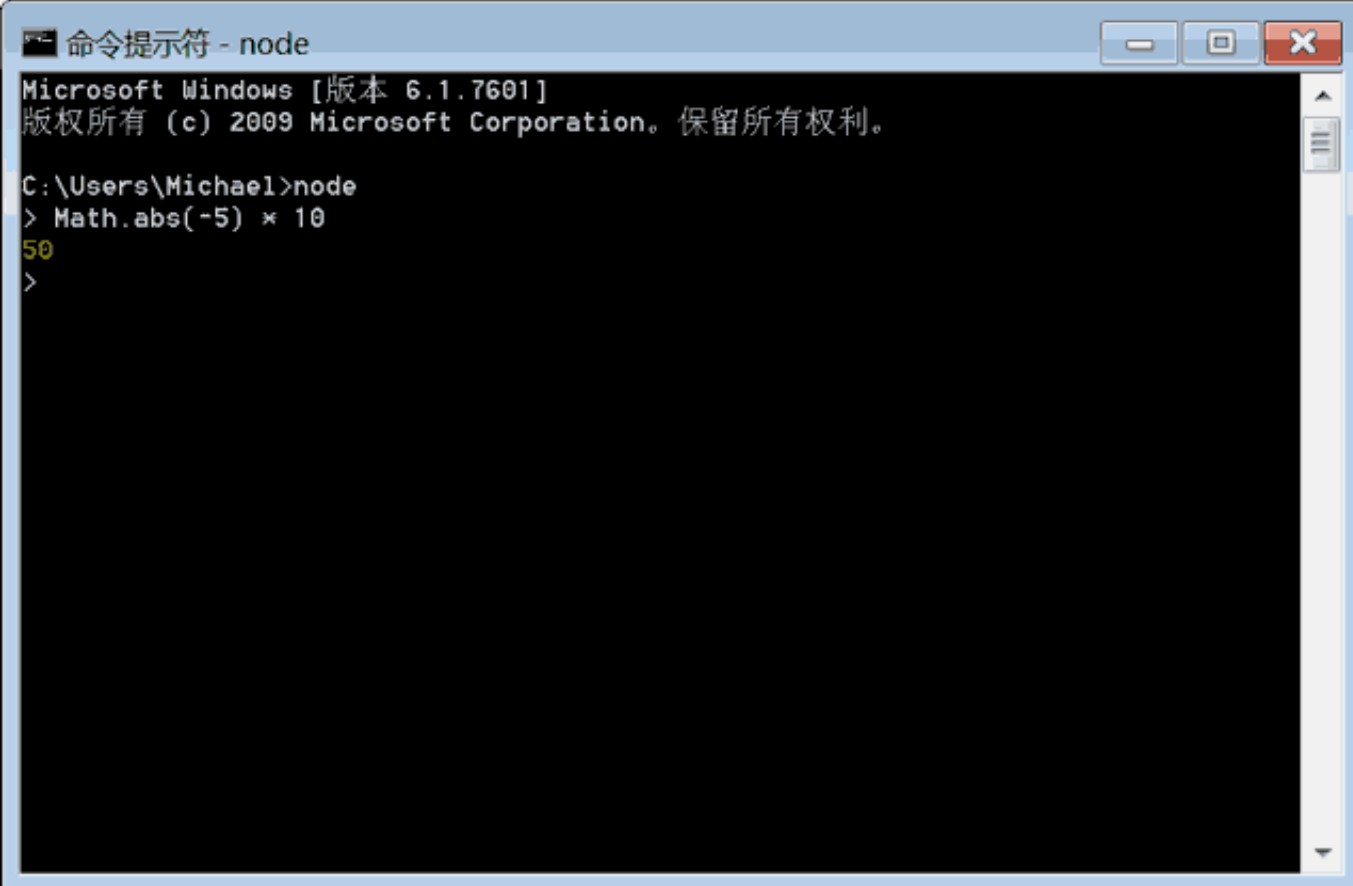
在Mac或Linux环境下，请打开终端，然后输入 `node -v`，你应该看到如下输出：



```
$ node -v
v0.12.7
```

如果版本号不是 `0.12.x`，说明Node.js版本不对，后面章节的代码不保证能正常运行，请重新安装最新版本。

继续在命令提示符或者终端输入 `node`，此刻你将进入Node.js的交互环境。在交互环境下，你可以输入任意JavaScript语句，回车后得到输出：



要退出Node.js环境，连接两次Ctrl+C。

npm

在正式开始Node.js学习之前，我们先认识一下npm。

npm是什么东东？npm其实是Node.js的包管理工具（package manager）。

为啥我们需要一个包管理工具呢？因为我们在Node.js上开发时，会用到很多别人写的JavaScript代码。如果我们要使用别人写的某个包，每次都根据名称搜索一下官方网站，下载代码，解压，再使用，非常繁琐。于是一个集中管理的工具应运而生：大家都把自己开发的模块打包后放到npm官网上，如果要使用，直接通过npm安装就可以直接用，不用管代码存在哪，应该从哪下载。

更重要的是，如果我们要使用模块A，而模块A又依赖于模块B，模块B又依赖于模块X和模块Y，npm可以根据依赖关系，把所有依赖的包都下载下来并管理起来。否则，靠我们自己手动管理，肯定既麻烦又容易出错。

讲了这么多，npm究竟在哪？

其实npm已经在Node.js安装的时候顺带装好了。我们在命令提示符或者终端输入 `npm -v`，应该看到类似的输出：

```
C:\> npm -v
2.11.3
```

如果直接输入 `npm`，你会看到类似下面的输出：

```
C:\> npm

Usage: npm <command>

where <command> is one of:
  ...
```

上面的一大堆关于告诉你，`npm`需要跟上命令。现在我们将加入它这三命令，后面会讲到。目前，你只需要确保`npm`正确安装了，能运行就行。

小结

请在本机安装Node.js环境，并确保`node`和`npm`能正常运行。