

JavaScript教程

JavaScript全栈教程

www.liaoxuefeng.com



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简介

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。

快速入门

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');
```

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。

基本语法

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');
```

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。

数据类型和变量

数据类型

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

Number

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

```
123; // 整数123
0.456; // 浮点数0.456
1.2345e3; // 科学计数法表示1.2345x1000，等同于1234.5
-99; // 负数
NaN; // NaN表示Not a Number，当无法计算结果时用NaN表示
Infinity; // Infinity表示无限大，当数值超过了JavaScript的Number所能表示的最大值时，就表示为Infinity
```

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

Number可以直接做四则运算，规则和数学一致：

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

注意`%`是求余运算。

字符串

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

布尔值

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

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

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

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

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


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

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

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

布尔值经常用在条件判断中，比如：

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

比较运算符

当我们对 **Number** 做比较时，可以通过比较运算符得到一个布尔值：

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

实际上，**JavaScript** 允许对任意数据类型做比较：

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

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

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

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

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

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

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

唯一能判断 **NaN** 的方法是通过 **isNaN()** 函数：

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

最后要注意浮点数的相等比较：

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

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

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

null和undefined

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

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

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

数组

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

```

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

```

上述数组包含6个元素。数组用`[]`表示，元素之间用`,`分隔。

另一种创建数组的方法是通过`Array()`函数实现：

```

new Array(1, 2, 3); // 创建了数组[1, 2, 3]

```

然而，出于代码的可读性考虑，强烈建议直接使用`[]`。

数组的元素可以通过索引来访问。请注意，索引的起始值为`0`：

```

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

```

对象

JavaScript的对象是一组由键-值组成的无序集合，例如：

```

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

```

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

要获取一个对象的属性，我们用`对象变量.属性名`的方式：

```

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

```

变量

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

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

```
var a; // 声明了变量a, 此时a的值为undefined
var $b = 1; // 声明了变量$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`声明一次，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);
```

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。

字符串

JavaScript的字符串就是用`' '`或`" "`括起来的字符表示。

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

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

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

表示的字符串内容是：`I'm "OK"!`

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

ASCII字符可以以`\x##`形式的十六进制表示，例如：

```
'\x41'; // 完全等同于 'A'
```

还可以用`\u####`表示一个Unicode字符：

```
'\u4e2d\u6587'; // 完全等同于 '中文'
```

多行字符串

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

```
`这是一个
多行
字符串`;
```

注意：反引号在键盘的ESC下方，数字键1的左边：

ESC	F1	F2	F3	F4
~	! 1	@ 2	# 3	\$ 4
tab	Q	W	E	
caps lock	A	S		

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`);
```

模板字符串

要把多个字符串连接起来，可以用`+`号连接：

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

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

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

练习: 测试你的浏览器是否支持**ES6**模板字符串, 如果不支持, 请把模板字符串改为`+`连接的普通字符串:

```
// 如果浏览器支持模板字符串, 将会替换字符串内部的变量:
----
var name = '小明';
var age = 20;
alert(`你好, ${name}, 你今年${age}岁了!`);
```

操作字符串

字符串常见的操作如下:

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

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

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

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

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

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

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

toUpperCase

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'
```

数组

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

要取得 `Array` 的长度，直接访问 `length` 属性：

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

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

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

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

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

请注意，如果通过索引赋值时，索引超过了范围，同样会引起 `Array` 大小的变化：

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

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

indexOf

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

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

注意了，数字 `30` 和字符串 `'30'` 是不同的元素。

slice

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

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

注意到 `slice()` 的起止参数包括开始索引，不包括结束索引。

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


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

push和pop

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

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

unshift和shift

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

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

sort

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

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

能否按照我们自己指定的顺序排序呢？完全可以，我们将在后面的函数中讲到。

reverse

`reverse()` 把整个 `Array` 的元素给掉个个，也就是反转：

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

splice

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

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

concat

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

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

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

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

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

join

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

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

如果 `Array` 的元素不是字符串,将自动转换为字符串后再连接。

多维数组

如果数组的某个元素又是一个 `Array`,则可以形成多维数组,例如:

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

上述 `Array` 包含3个元素,其中头两个元素本身也是 `Array`。

练习:如何通过索引取到 `500` 这个值:

```
'use strict';
var arr = [[1, 2, 3], [400, 500, 600], '-'];
----
var x = ??;
----
alert(x); // x应该为500
```

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('???');
```

对象

JavaScript的对象是一种无序的集合数据类型，它由若干键值对组成。

JavaScript的对象用于描述现实世界中的某个对象。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
```

条件判断

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');
}
```

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 ...
```

循环

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，因此它最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');
}
```

`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'];
----
```

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。

Map和Set

JavaScript的默认对象表示方式`{}`可以视为其他语言中的`Map`或`Dictionary`的数据结构，即一组键值对。

但是JavaScript的对象有个小问题，就是键必须是字符串。但实际上`Number`或者其他数据类型作为键也是非常合理的。

为了解决这个问题，最新的ES6规范引入了新的数据类型`Map`。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
```

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。

iterable

遍历 `Array` 可以采用下标循环，遍历 `Map` 和 `Set` 就无法使用下标。为了统一集合类型，ES6标准引入了新的 `iterable` 类型，`Array`、`Map` 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
```

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，因此回调函数的前两个参数都是元素本身：

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

函数

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：

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

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。

函数定义和调用

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!');
}
}
```

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');
}
```

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));
```

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。

变量作用域

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，把所有申明的变量“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!');
```

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
```


方法

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

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
```

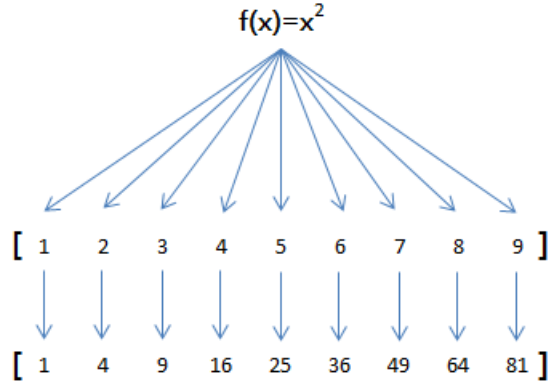
编写高阶函数，就是让函数的参数能够接收别的函数。

map/reduce

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]) === 44274384) {
    alert('测试通过!');
}
else {
    alert('测试失败!');
}
```

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!');
}
```

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!');
}
```

小明希望利用 `map()` 把字符串变成整数，他写的代码很简洁：

```
'use strict';

var arr = ['1', '2', '3'];
var r;
----
r = arr.map(parseInt);
----
alert([' ' + r[0] + ' ', ' ' + r[1] + ' ', ' ' + r[2] + ' ']);
```

结果竟然是 `[1, NaN, NaN]`，小明百思不得其解，请帮他找到原因并修正代码。

提示： 参考[Array.prototype.map\(\)](#)的文档。

原因分析

filter

`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];
var r = arr.filter(function (x) {
    return x % 2 !== 0;
});
r; // [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', ' '];
var r = arr.filter(function (s) {
    return s && s.trim(); // 注意：IE9以下的版本没有trim()方法
});
r; // ['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。

回调函数

`filter()` 接收的回调函数，其实可以有多个参数。通常我们仅使用第一个参数，表示 `Array` 的某个元素。回调函数还可以接收另外两个参数，表示元素的位置和数组本身：

```
var arr = ['A', 'B', 'C'];
var r = arr.filter(function (element, index, self) {
    console.log(element); // 依次打印'A', 'B', 'C'
    console.log(index); // 依次打印0, 1, 2
    console.log(self); // self就是变量arr
    return true;
});
```

利用 `filter`，可以巧妙地去掉 `Array` 的重复元素：

```
'use strict';

var
    r,
    arr = ['apple', 'strawberry', 'banana', 'pear', 'apple', 'orange', 'orange', 'strawberry'];
----
r = arr.filter(function (element, index, self) {
    return self.indexOf(element) === index;
});
----
alert(r.toString());
```

去除重复元素依靠的是 `indexOf` 总是返回第一个元素的位置，后续的重复元素位置与 `indexOf` 返回的位置不相等，因此被 `filter` 滤掉了。

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, 73, 79, 83, 89, 97].toString()) {
  alert('测试通过!');
} else {
  alert('测试失败: ' + r.toString());
}
```

sort

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。



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
```

闭包

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...
```

箭头函数

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**Arrow Function**：

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

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

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!');
}
```

标准对象

在JavaScript的世界里，一切都是对象。

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

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

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

包装对象

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

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

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

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

```
typeof new Number(123); // 'object'
new Number(123) === 123; // false

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

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

所以**闲的蛋疼也不要使用包装对象**！尤其是针对`string`类型！！

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

此时，`Number()`、`Boolean`和`String()`被当做普通函数，把任何类型的数据转换为`number`、`boolean`和`string`类型（注意不是其包装类型）：

```
var n = Number('123'); // 123, 相当于parseInt()或parseFloat()
typeof n; // 'number'

var b = Boolean('true'); // true
typeof b; // 'boolean'

var b2 = Boolean('false'); // true! 'false'字符串转换结果为true! 因为它非空字符串!
var b3 = Boolean(''); // false

var s = String(123.45); // '123.45'
typeof s; // 'string'
```

是不是感觉头大了？这就是JavaScript特有的催眠魅力！

总结一下，有这么几条规则需要遵守：

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

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

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

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

遇到这种情况，要特殊处理一下：

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

不要问为什么，这就是 **JavaScript** 代码的乐趣！

Date

在JavaScript中，`Date`对象用来表示日期和时间。

要获取系统当前时间，用：

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

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

如果要创建一个指定日期和时间的`Date`对象，可以用：

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

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

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



RegExp

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

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

所以我们判断一个字符串是否是合法的Email的方法是：

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

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

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

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

.可以匹配任意字符，所以：

- 'js.'可以匹配'jsp'、'jss'、'js!'等等。

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

来看一个复杂的例子：\d{3}\s+\d{3,8}。

我们来从左到右解读一下：

1. \d{3}表示匹配3个数字，例如'010'；
2. \s可以匹配一个空格（也包括Tab等空白符），所以\s+表示至少有一个空格，例如匹配' '，'\t\t'等；
3. \d{3,8}表示3-8个数字，例如'1234567'。

综合起来，上面的正则表达式可以匹配以任意个空格隔开的带区号的电话号码。

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

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

进阶

要做更精确地匹配，可以用[]表示范围，比如：

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

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

^表示行的开头，^d表示必须以数字开头。

`$`表示行的结束，`\d$`表示必须以数字结束。

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

RegExp

有了准备知识，我们就可以在JavaScript中使用正则表达式了。

JavaScript有两种方式创建一个正则表达式：

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

两种写法是一样的：

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

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

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

先看看如何判断正则表达式是否匹配：

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

RegExp对象的`test()`方法用于测试给定的字符串是否符合条件。

切分字符串

用正则表达式切分字符串比用固定的字符更灵活，请看正常的切分代码：

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

嗯，无法识别连续的空格，用正则表达式试试：

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

无论多少个空格都可以正常分割。加入`,`试试：

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

再加入`;`试试：

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

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

分组

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

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

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

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

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

`exec()` 方法在匹配失败时返回 `null`。

提取子串非常有用。来看一个更凶残的例子：

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

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

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

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

贪婪匹配

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

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

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

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

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

全局搜索

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

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

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

```

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

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

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

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

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

re.exec(s); // null, 直到结束仍没有匹配到

```

全局匹配类似搜索，因此不能使用 `/^...$/`，那样只会最多匹配一次。

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

小结

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

练习

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

```

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

```

版本二可以验证并提取出带名字的Email地址:

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

JSON

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

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

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

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

- number: 和JavaScript的 `number` 完全一致；
- boolean: 就是JavaScript的 `true` 或 `false`；
- string: 就是JavaScript的 `string`；
- null: 就是JavaScript的 `null`；
- array: 就是JavaScript的 `Array` 表示方式——`[]`；
- object: 就是JavaScript的 `{ ... }` 表示方式。

以及上面的任意组合。

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

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

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

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

序列化

让我们先把小明这个对象序列化成JSON格式的字符串：

```
var xiaoming = {
  name: '小明',
  age: 14,
  gender: true,
  height: 1.65,
  grade: null,
  'middle-school': '"W3C" Middle School',
  skills: ['JavaScript', 'Java', 'Python', 'Lisp']
};

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

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。

面向对象编程

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！

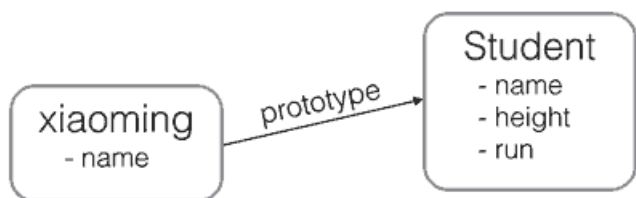
于是我们把它改名为 `Student`，然后创建出 `xiaoming`：

```
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。

创建对象

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 ↗
```

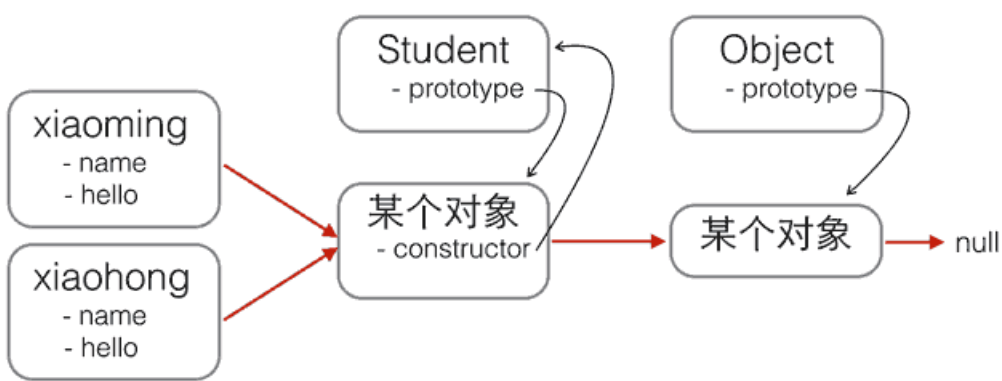
用 `new Student()` 创建的对象还从原型上获得了一个 `constructor` 属性，它指向函数 `Student` 本身：

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

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

不过还有一个小问题，注意观察：

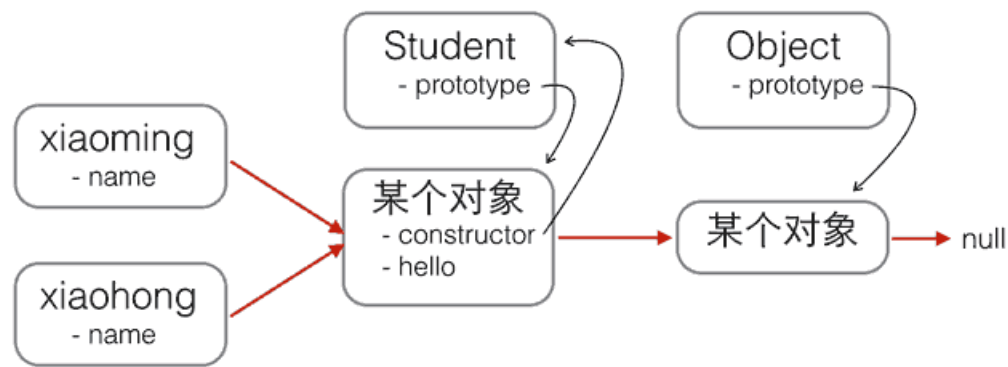
```
xiaoming.name; // '小明'
xiaohong.name; // '小红'
xiaoming.hello; // function: Student.hello()
xiaohong.hello; // function: Student.hello()
xiaoming.hello === xiaohong.hello; // false
```

`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 `eslint` 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() === 'Hello, Kitty!' && kitty.say === d
    alert('测试通过!');
} else {
    alert('测试失败!');
}
}
```

原型继承

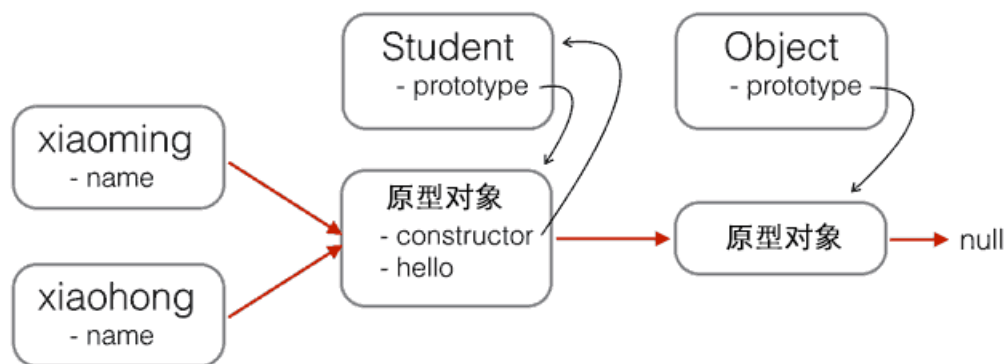
在传统的基于Class的语言如Java、C++中，继承的本质是扩展一个已有的Class，并生成新的Subclass。

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

但是办法还是有的。我们先回顾 `Student` 构造函数：

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

以及 `Student` 的原型链：



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

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

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

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

必须想办法把原型链修改为：

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

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

如果你想用最简单粗暴的方法这么干：

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

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

我们必须借助一个中间对象来实现正确的原型链，这个中间对象的原型要指向 `Student.prototype`。为了实现这一点，参考道爷（就是发明JSON的那个道格

拉斯)的代码,中间对象可以用一个空函数 **F** 来实现:

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

// 空函数F:
function F() {}

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

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

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

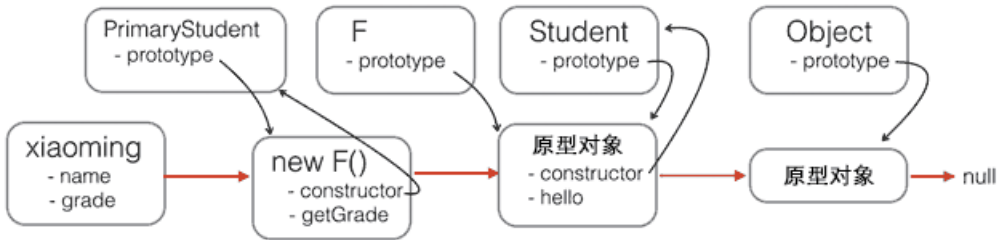
// 继续在PrimaryStudent原型(就是new F()对象)上定义方法:
PrimaryStudent.prototype.getGrade = function () {
  return this.grade;
};

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

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

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

用一张图来表示新的原型链:



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

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

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

这个 `inherits()` 函数可以复用：

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

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

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

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

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

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 `inherits` 函数完成；
3. 继续在新的构造函数的原型上定义新方法。

class继承

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

有没有更简单的写法？有！

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

我们先回顾用函数实现 `Student` 的方法：

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

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

如果用新的 `class` 关键字来编写 `Student`，可以这样写：

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

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

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

最后，创建一个 `Student` 对象代码和前面章节完全一样：

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

class继承

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

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

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

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

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

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

你一定会问，`class` 这么好用，能不能现在就用上？

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

练习

请利用 `class` 重新定义 `Cat`，并让它从已有的 `Animal` 继承，然后新增一个方法 `say()`，返回字符串 `'Hello, xxx!'`：

```
'use strict';

class Animal {
  constructor(name) {
    this.name = name;
  }
}
----
class Cat ???
----
// 测试：
var kitty = new Cat('Kitty');
var doraemon = new Cat('哆啦A梦');
if ((new Cat('x') instanceof Animal) && kitty && kitty.name === 'Kitty' && kitty.say && typeof kitty.say === 'function' && kitty.say() === 'Hello, xxx!') {
  alert('测试通过!');
} else {
  alert('测试失败!');
}
```

这个练习需要浏览器支持ES6的 `class`，如果遇到SyntaxError，则说明浏览器不支持 `class` 语法，请换一个最新的浏览器试试。

浏览器

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

目前主流的浏览器分这么几种：

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

其他浏览器如Opera等由于市场份额太小就被自动忽略了。

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

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

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

浏览器对象

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);
```

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);
```

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);
```

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
}
```

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';
----
document.title = '努力学习JavaScript!';
```

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>
```

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

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

摩卡
热摩卡咖啡
酸奶
北京老酸奶
果汁
鲜榨苹果汁

`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。

操作DOM

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.lastChild;
```

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 !== 'Python' || arr[1].innerText !== 'Ruby' || a
  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!');
}
```

更新DOM

拿到一个**DOM**节点后，我们可以对它进行更新。

可以直接修改节点的文本，方法有两种：

一种是修改 `innerHTML` 属性，这个方式非常强大，不但可以修改一个**DOM**节点的文本内容，还可以直接通过**HTML**片段修改**DOM**节点内部的子树：

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

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

第二种是修改 `innerText` 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.color === '#ff0000' || js.style.color ===
            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!');
}
```

插入DOM

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!');
  }
})();
```


删除DOM

删除一个DOM节点就比插入要容易得多。

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

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

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

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

例如，对于如下HTML结构：

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

当我们用如下代码删除子节点时：

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

浏览器报错：`parent.children[1]`不是一个有效的节点。原因就在于，当`<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!');
    }
})();
```

操作表单

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"> Tuesday</label>
var mon = document.getElementById('monday');
var tue = document.getElementById('tuesday');
mon.value; // '1'
tue.value; // '2'
mon.checked; // true或者false
tue.checked; // true或者false
```

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" value="2015-07-01">
```

```
<input type="datetime-local" value="2015-07-01T02:03:04">
```

2015-07-01T02:03:04

```
<input type="color" value="#ff0000">
```

#ff0000

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');
  // 可以在这里修改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');
  // 可以在这里修改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 `<input type="hidden">` 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 `<input>` 标记了 `name="password"`，而用户输入的 `id` 为 `input-password` 的 `<input>` 没有 `name` 属性。没有 `name` 属性的 `<input>` 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。

```
<!-- HTML结构 -->
<form id="test-register" action="#" target="_blank" onsubmit="return checkRegisterForm()" ">
  <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>
```

用户名:

口令:

重复口令:

提交

重置

操作文件

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

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：

AJAX

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。

如果要让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...');
```

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 ActiveXObject('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...');
```

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：

Promise

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.isSuccess(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!');
```

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.');
```

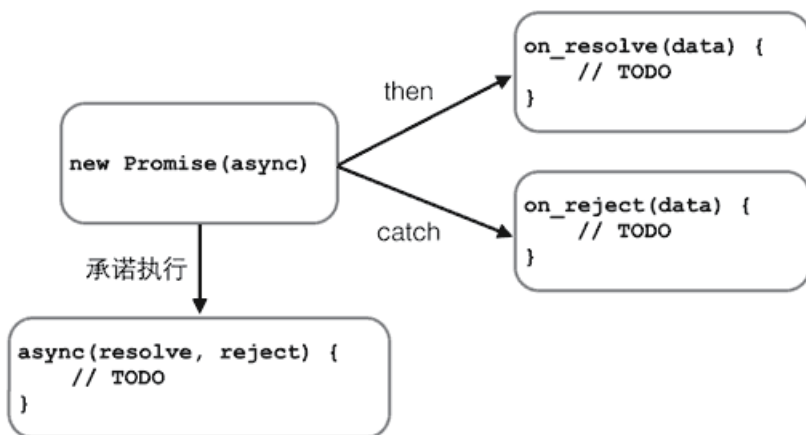
```
    setTimeout(function () {
        if (timeOut < 1) {
            log('call resolve()...');
            resolve('200 OK');
```

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

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

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);
  });
```

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;
});
```

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

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：

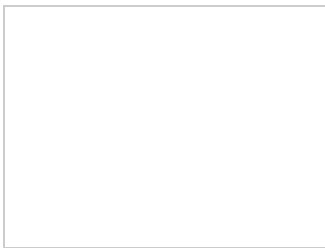
```
<canvas id="test-canvas" width="300" height="200"></canvas>
```

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!');
}
```

`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');
```

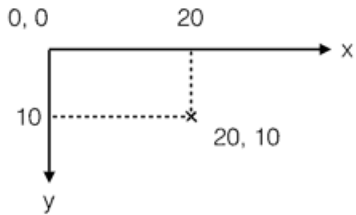
如果需要绘制3D怎么办？HTML5还有一个WebGL规范，允许在Canvas中绘制3D图形：

```
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：

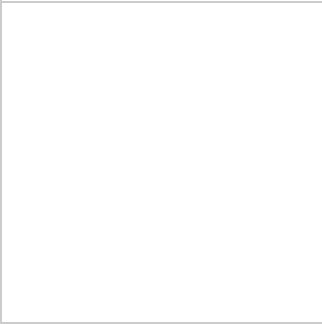


Canvas的坐标以左上角为原点，水平向右为X轴，垂直向下为Y轴，以像素为单位，所以每个点都是非负整数。

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);
```



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);
```



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

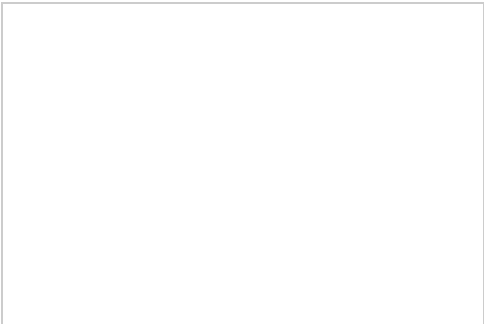
请根据从163获取的JSON数据绘制最近30个交易日的K线图，数据已处理为包含一组对象的数组：

```
'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}
    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=loadStockData&t=' + Date.now();
document.getElementsByTagName('head')[0].appendChild(js);
```



[下载为图片](#)

jQuery

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);
```

\$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

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');
```

JavaScript

Haskell

Erlang

Python

[注册新用户](#)

名字:

邮件:

口令:

层级选择器

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）

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

- JavaScript
- Python
- Lua

1. Swift
2. Java
3. C

查找和过滤

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

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

Name:

Email:

Password:

Gender: ☒ Male ☐ Female

City: Beijing ▼

操作DOM

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');
```

- 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');
```

- 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:
```

- 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。

获取DOM信息

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

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

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

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

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

```
// <div id="test-div" name="Test" start="1">...</div>
var div = $('#test-div');
div.attr('data'); // undefined, 属性不存在
div.attr('name'); // 'Test'
div.attr('name', 'Hello'); // div的name属性变为'Hello'
div.removeAttr('name'); // 删除name属性
div.attr('name'); // undefined
```

`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')`。

操作表单

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

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

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

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

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

可见, 一个 `val()` 就统一了各种输入框的取值和赋值的问题。

修改DOM结构

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);
  }
})();
```

- JavaScript
- Python
- Swift

事件

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：

动画

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
```

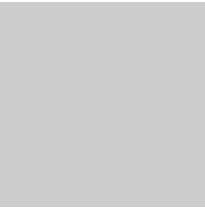
时间以毫秒为单位，但也可以是 `'slow'`，`'fast'` 这些字符串：

```
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：

AJAX

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

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');
});
```


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JSONP，可以在`ajax()`中设置`jsonp: 'callback'`，让jQuery实现JSONP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。

扩展

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()`：

错误处理

在执行JavaScript代码的时候，有些情况下会发生错误。

错误分两种，一种是程序写的逻辑不对，导致代码执行异常。例如：

```
var s = null;
var len = s.length; // TypeError: null变量没有length属性
```

对于这种错误，要修复程序。

一种是执行过程中，程序可能遇到无法预测的异常情况而报错，例如，网络连接中断，读取不存在的文件，没有操作权限等。

对于这种错误，我们需要处理它，并可能需要给用户反馈。

错误处理是程序设计时必须要考虑的问题。对于C这样贴近系统底层的语言，错误是通过错误码返回的：

```
int fd = open("/path/to/file", O_RDONLY);
if (fd == -1) {
    printf("Error when open file!");
} else {
    // TODO
}
```

通过错误码返回错误，就需要约定什么是正确的返回值，什么是错误的返回值。上面的`open()`函数约定返回`-1`表示错误。

显然，这种用错误码表示错误在编写程序时十分不便。

因此，高级语言通常都提供了更抽象的错误处理逻辑`try ... catch ... finally`，JavaScript也不例外。

try ... catch ... finally

使用`try ... catch ... finally`处理错误时，我们编写的代码如下：

```
'use strict';
----
var r1, r2, s = null;
try {
    r1 = s.length; // 此处应产生错误
    r2 = 100; // 该语句不会执行
} catch (e) {
    alert('出错了: ' + e);
} finally {
    console.log('finally');
}
console.log('r1 = ' + r1); // r1应为undefined
console.log('r2 = ' + r2); // r2应为undefined
----
// 直接运行
```

运行后可以发现，弹出的Alert提示类似“出错了: `TypeError: Cannot read property 'length' of null`”。

我们来分析一下使用`try ... catch ... finally`的执行流程。

当代码块被`try { ... }`包裹的时候，就表示这部分代码执行过程中可能会发生错误，一旦发生错误，就不再继续执行后续代码，转而跳到`catch`块。`catch (e) { ... }`包裹的代码就是错误处理代码，变量`e`表示捕获到的错误。最后，无论有没有错误，`finally`一定会被执行。

所以，有错误发生时，执行流程像这样：

1. 先执行`try { ... }`的代码；
2. 执行到出错的语句时，后续语句不再继续执行，转而执行`catch (e) { ... }`代码；

3. 最后执行`finally { ... }`代码。

而没有错误发生时，执行流程像这样：

1. 先执行`try { ... }`的代码；
2. 因为没有出错，`catch (e) { ... }`代码不会被执行；
3. 最后执行`finally { ... }`代码。

最后请注意，`catch`和`finally`可以不必都出现。也就是说，`try`语句一共有三种形式：

完整的`try ... catch ... finally`：

```
try {  
    ...  
} catch (e) {  
    ...  
} finally {  
    ...  
}
```

只有`try ... catch`，没有`finally`：

```
try {  
    ...  
} catch (e) {  
    ...  
}
```

只有`try ... finally`，没有`catch`：

```
try {  
    ...  
} finally {  
    ...  
}
```

错误类型

JavaScript有一个标准的`Error`对象表示错误，还有从`Error`派生的`TypeError`、`ReferenceError`等错误对象。我们在处理错误时，可以通过`catch(e)`捕获的变量`e`访问错误对象：

```
try {  
    ...  
} catch (e) {  
    if (e instanceof TypeError) {  
        alert('Type error!');  
    } else if (e instanceof Error) {  
        alert(e.message);  
    } else {  
        alert('Error: ' + e);  
    }  
}
```

使用变量`e`是一个习惯用法，也可以以其他变量命名，如`catch(ex)`。

抛出错误

程序也可以主动抛出一个错误，让执行流程直接跳转到`catch`块。抛出错误使用`throw`语句。

例如，下面的代码让用户输入一个数字，程序接收到的实际上是一个字符串，然后用`parseInt()`转换为整数。当用户输入不合法的时候，我们就抛出错误：

```
'use strict';
----
var r, n, s;
try {
    s = prompt('请输入一个数字');
    n = parseInt(s);
    if (isNaN(n)) {
        throw new Error('输入错误');
    }
    // 计算平方:
    r = n * n;
    alert(n + ' * ' + n + ' = ' + r);
} catch (e) {
    alert('出错了: ' + e);
}
----
// 直接运行
```

实际上，JavaScript允许抛出任意对象，包括数字、字符串。但是，最好还是抛出一个Error对象。

最后，当我们用catch捕获错误时，一定要编写错误处理语句：

```
var n = 0, s;
try {
    n = s.length;
} catch (e) {
    console.log(e);
}
console.log(n);
```

哪怕仅仅把错误打印出来，也不要什么也不干：

```
var n = 0, s;
try {
    n = s.length;
} catch (e) {
}
console.log(n);
```

因为catch到错误却什么都不执行，就不知道程序执行过程中到底有没有发生错误。

处理错误时，请不要简单粗暴地用`alert()`把错误显示给用户。教程的代码使用`alert()`是为了便于演示。

错误传播

如果代码发生了错误，又没有被`try ... catch`捕获，那么，程序执行流程会跳转到哪呢？

```
function getLength(s) {
    return s.length;
}

function printLength() {
    console.log(getLength('abc')); // 3
    console.log(getLength(null)); // Error!
}

printLength();
```

如果在一个函数内部发生了错误，它自身没有捕获，错误就会被抛到外层调用函数，如果外层函数也没有捕获，该错误会一直沿着函数调用链向上抛出，直到被JavaScript引擎捕获，代码终止执行。

所以，我们不必在每一个函数内部捕获错误，只需要在合适的地方来个统一捕获，一网打尽：

```
'use strict';
----
function main(s) {
    console.log('BEGIN main()');
    try {
        foo(s);
    } catch (e) {
        alert('出错了: ' + e);
    }
    console.log('END main()');
}

function foo(s) {
    console.log('BEGIN foo()');
    bar(s);
    console.log('END foo()');
}

function bar(s) {
    console.log('BEGIN bar()');
    console.log('length = ' + s.length);
    console.log('END bar()');
}

main(null);
----
// 直接运行，观察控制台输出
```

当`bar()`函数传入参数`null`时，代码会报错，错误会向上抛给调用方`foo()`函数，`foo()`函数没有`try ... catch`语句，所以错误继续向上抛给调用方`main()`函数，`main()`函数有`try ... catch`语句，所以错误最终在`main()`函数被处理了。

至于在哪些地方捕获错误比较合适，需要视情况而定。

异步错误处理

编写JavaScript代码时，我们要时刻牢记，JavaScript引擎是一个事件驱动的执行引擎，代码总是以单线程执行，而回调函数的执行需要等到下一个满足条件的事件出现后，才会被执行。

例如，`setTimeout()` 函数可以传入回调函数，并在指定若干毫秒后执行：

```
function printTime() {  
    console.log('It is time!');  
}  
  
setTimeout(printTime, 1000);  
console.log('done');
```

上面的代码会先打印 `done`，1秒后才会打印 `It is time!`。

如果 `printTime()` 函数内部发生了错误，我们试图用 `try` 包裹 `setTimeout()` 是无效的：

```
'use strict';  
----  
function printTime() {  
    throw new Error();  
}  
  
try {  
    setTimeout(printTime, 1000);  
    console.log('done');  
} catch (e) {  
    alert('error');  
}  
----  
// 直接运行，看是否会alert
```

原因就在于调用 `setTimeout()` 函数时，传入的 `printTime` 函数并未立刻执行！紧接着，JavaScript引擎会继续执行 `console.log('done');` 语句，而此时并没有错误发生。直到1秒钟后，执行 `printTime` 函数时才发生错误，但此时除了在 `printTime` 函数内部捕获错误外，外层代码并无法捕获。

所以，涉及到异步代码，无法在调用时捕获，原因就是捕获的当时，回调函数并未执行。

类似的，当我们处理一个事件时，在绑定事件的代码处，无法捕获事件处理函数的错误。

例如，针对以下的表单：

```
<form>  
    <input id="x"> + <input id="y">  
    <button id="calc" type="button">计算</button>  
</form>
```

+

计算

我们用下面的代码给 `button` 绑定 `click` 事件：

```
'use strict';

var $btn = $('#calc');

// 取消已绑定的事件:
$btn.off('click');
----
try {
    $btn.click(function () {
        var
            x = parseFloat($('#x').val()),
            y = parseFloat($('#y').val()),
            r;
        if (isNaN(x) || isNaN(y)) {
            throw new Error('输入有误');
        }
        r = x + y;
        alert('计算结果: ' + r);
    });
} catch (e) {
    alert('输入有误! ');
}
```

但是，用户输入错误时，处理函数并未捕获到错误。请修复错误处理代码。

underscore

前面我们已经讲过了，JavaScript是函数式编程语言，支持高阶函数和闭包。函数式编程非常强大，可以写出非常简洁的代码。例如Array的map()和filter()方法：

```
'use strict';
var a1 = [1, 4, 9, 16];
var a2 = a1.map(Math.sqrt); // [1, 2, 3, 4]
var a3 = a2.filter((x) => { return x % 2 === 0; }); // [2, 4]
```

现在问题来了，Array有map()和filter()方法，可是Object没有这些方法。此外，低版本的浏览器例如IE6~8也没有这些方法，怎么办？

方法一，自己把这些方法添加到Array.prototype中，然后给Object.prototype也加上mapObject()等类似的方法。

方法二，直接找一个成熟可靠的第三方开源库，使用统一的函数来实现map()、filter()这些操作。

我们采用方法二，选择的第三方库就是underscore。

正如jQuery统一了不同浏览器之间的DOM操作的差异，让我们可以简单地对DOM进行操作，underscore则提供了一套完善的函数式编程的接口，让我们更方便地在JavaScript中实现函数式编程。

jQuery在加载时，会把自身绑定到唯一的全局变量\$上，underscore与其类似，会把自身绑定到唯一的全局变量_上，这也是为啥它的名字叫underscore的原因。

用underscore实现map()操作如下：

```
'use strict';
_.map([1, 2, 3], (x) => x * x); // [1, 4, 9]
```

咋一看比直接用Array.map()要麻烦一点，可是underscore的map()还可以作用于Object:

```
'use strict';
_.map({ a: 1, b: 2, c: 3 }, (v, k) => k + '=' + v); // ['a=1', 'b=2', 'c=3']
```

后面我们会详细介绍underscore提供了一系列函数式接口。

Collections

underscore为集合类对象提供了一致的接口。集合类是指Array和Object，暂不支持Map和Set。

map/filter

和Array的map()与filter()类似，但是underscore的map()和filter()可以作用于Object。当作用于Object时，传入的函数为function (value, key)，第一个参数接收value，第二个参数接收key：

```
'use strict';

var obj = {
  name: 'bob',
  school: 'No.1 middle school',
  address: 'xueyuan road'
};
----
var upper = _.map(obj, function (value, key) {
  return ???;
});
----
alert(JSON.stringify(upper));
```

你也许会想，为啥对Object作map()操作的返回结果是Array？应该是Object才合理啊！把_.map换成_.mapObject再试试。

every / some

当集合的所有元素都满足条件时，_.every()函数返回true，当集合的至少一个元素满足条件时，_.some()函数返回true：

```
'use strict';
// 所有元素都大于0?
_.every([1, 4, 7, -3, -9], (x) => x > 0); // false
// 至少一个元素大于0?
_.some([1, 4, 7, -3, -9], (x) => x > 0); // true
```

当集合是Object时，我们可以同时获得value和key：

```
'use strict';
var obj = {
  name: 'bob',
  school: 'No.1 middle school',
  address: 'xueyuan road'
};
// 判断key和value是否全部是小写:
----
var r1 = _.every(obj, function (value, key) {
  return ???;
});
var r2 = _.some(obj, function (value, key) {
  return ???;
});
----
alert('every key-value are lowercase: ' + r1 + '\nsome key-value are lowercase: ' + r2);
```

max / min

这两个函数直接返回集合中最大和最小的数：

```
'use strict';
var arr = [3, 5, 7, 9];
_.max(arr); // 9
_.min(arr); // 3

// 空集合会返回-Infinity和Infinity，所以要先判断集合不为空：
_.max([])
-Infinity
_.min([])
Infinity
```

注意，如果集合是Object，`max()`和`min()`只作用于value，忽略掉key：

```
'use strict';
_.max({ a: 1, b: 2, c: 3 }); // 3
```

groupBy

`groupBy()`把集合的元素按照key归类，key由传入的函数返回：

```
'use strict';

var scores = [20, 81, 75, 40, 91, 59, 77, 66, 72, 88, 99];
var groups = _.groupBy(scores, function (x) {
  if (x < 60) {
    return 'C';
  } else if (x < 80) {
    return 'B';
  } else {
    return 'A';
  }
});
// 结果：
// {
//   A: [81, 91, 88, 99],
//   B: [75, 77, 66, 72],
//   C: [20, 40, 59]
// }
```

可见`groupBy()`用来分组是非常方便的。

shuffle / sample

`shuffle()`用洗牌算法随机打乱一个集合：

```
'use strict';
// 注意每次结果都不一样：
_.shuffle([1, 2, 3, 4, 5, 6]); // [3, 5, 4, 6, 2, 1]
```

`sample()`则是随机选择一个或多个元素：

```
'use strict';
// 注意每次结果都不一样：
// 随机选1个：
_.sample([1, 2, 3, 4, 5, 6]); // 2
// 随机选3个：
_.sample([1, 2, 3, 4, 5, 6], 3); // [6, 1, 4]
```

更多完整的函数请参考underscore的文档：<http://underscorejs.org/#collections>

Arrays

`underscore`为 `Array` 提供了许多工具类方法，可以更方便快捷地操作 `Array`。

first / last

顾名思义，这两个函数分别取第一个和最后一个元素：

```
'use strict';  
var arr = [2, 4, 6, 8];  
_.first(arr); // 2  
_.last(arr); // 8
```

flatten

`flatten()` 接收一个 `Array`，无论这个 `Array` 里面嵌套了多少个 `Array`，`flatten()` 最后都把它们变成一个一维数组：

```
'use strict';  
  
_.flatten([1, [2], [3, [[4], [5]]]]); // [1, 2, 3, 4, 5]
```

zip / unzip

`zip()` 把两个或多个数组的所有元素按索引对齐，然后按索引合并成新数组。例如，你有一个 `Array` 保存了名字，另一个 `Array` 保存了分数，现在，要把名字和分数给对上，用 `zip()` 轻松实现：

```
'use strict';  
  
var names = ['Adam', 'Lisa', 'Bart'];  
var scores = [85, 92, 59];  
_.zip(names, scores);  
// [['Adam', 85], ['Lisa', 92], ['Bart', 59]]
```

`unzip()` 则是反过来：

```
'use strict';  
var namesAndScores = [['Adam', 85], ['Lisa', 92], ['Bart', 59]];  
_.unzip(namesAndScores);  
// [['Adam', 'Lisa', 'Bart'], [85, 92, 59]]
```

object

有时候你会想，与其用 `zip()`，为啥不把名字和分数直接对应成 `Object` 呢？别急，`object()` 函数就是干这个的：

```
'use strict';  
  
var names = ['Adam', 'Lisa', 'Bart'];  
var scores = [85, 92, 59];  
_.object(names, scores);  
// {Adam: 85, Lisa: 92, Bart: 59}
```

注意 `_.object()` 是一个函数，不是 JavaScript 的 `Object` 对象。

range

`range()` 让你快速生成一个序列，不再需要用 `for` 循环实现了：

```
'use strict';

// 从0开始小于10:
_.range(10); // [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

// 从1开始小于11:
_.range(1, 11); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

// 从0开始小于30，步长5:
_.range(0, 30, 5); // [0, 5, 10, 15, 20, 25]

// 从0开始大于-10，步长-1:
_.range(0, -10, -1); // [0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
```

更多完整的函数请参考underscore的文档: <http://underscorejs.org/#arrays>

Functions

因为underscore本来就是为了充分发挥JavaScript的函数式编程特性，所以也提供了大量JavaScript本身没有的高阶函数。

bind

`bind()`有什么用？我们先看一个常见的错误用法：

```
'use strict';

console.log('Hello, world!');
// 输出'Hello, world!'

var log = console.log;
log('Hello, world!');
// Uncaught TypeError: Illegal invocation
```

如果你想用`log()`取代`console.log()`，按照上面的做法是不行的，因为直接调用`log()`传入的`this`指针是`undefined`，必须这么用：

```
'use strict';

var log = console.log;
// 调用call并传入console对象作为this:
log.call(console, 'Hello, world!')
// 输出Hello, world!
```

这样搞多麻烦！还不如直接用`console.log()`。但是，`bind()`可以帮我们把`console`对象直接绑定在`log()`的`this`指针上，以后调用`log()`就可以直接正常调用了：

```
'use strict';

var log = _.bind(console.log, console);
log('Hello, world!');
// 输出Hello, world!
```

partial

`partial()`就是为一个函数创建偏函数。偏函数是什么东东？看例子：

假设我们要计算 x^y ，这时只需要调用`Math.pow(x, y)`就可以了。

假设我们经常计算 2^y ，每次都写`Math.pow(2, y)`就比较麻烦，如果创建一个新的函数能直接这样写`pow2N(y)`就好了，这个新函数`pow2N(y)`就是根据`Math.pow(x, y)`创建出来的偏函数，它固定住了原函数的第一个参数（始终为2）：

```
'use strict';

var pow2N = _.partial(Math.pow, 2);
pow2N(3); // 8
pow2N(5); // 32
pow2N(10); // 1024
```

如果我们不想固定第一个参数，想固定第二个参数怎么办？比如，希望创建一个偏函数`cube(x)`，计算 x^3 ，可以用`_`作占位符，固定住第二个参数：

```
'use strict';

var cube = _.partial(Math.pow, _, 3);
cube(3); // 27
cube(5); // 125
cube(10); // 1000
```

可见，创建偏函数的目的是将原函数的某些参数固定住，可以降低新函数调用的难度。

memoize

如果一个函数调用开销很大，我们就可能希望能把结果缓存下来，以便后续调用时直接获得结果。举个例子，计算阶乘就比较耗时：

```
'use strict';

function factorial(n) {
  console.log('start calculate ' + n + '!...');
  var s = 1, i = n;
  while (i > 1) {
    s = s * i;
    i --;
  }
  console.log(n + '! = ' + s);
  return s;
}

factorial(10); // 3628800
// 注意控制台输出：
// start calculate 10!...
// 10! = 3628800
```

用 `memoize()` 就可以自动缓存函数计算的结果：

```
'use strict';

var factorial = _.memoize(function(n) {
  console.log('start calculate ' + n + '!...');
  var s = 1, i = n;
  while (i > 1) {
    s = s * i;
    i --;
  }
  console.log(n + '! = ' + s);
  return s;
});

// 第一次调用：
factorial(10); // 3628800
// 注意控制台输出：
// start calculate 10!...
// 10! = 3628800

// 第二次调用：
factorial(10); // 3628800
// 控制台没有输出
```

对于相同的调用，比如连续两次调用 `factorial(10)`，第二次调用并没有计算，而是直接返回上次计算后缓存的结果。不过，当你计算 `factorial(9)` 的时候，仍然会重新计算。

可以对 `factorial()` 进行改进，让其递归调用：

```
'use strict';

var factorial = _.memoize(function(n) {
  console.log('start calculate ' + n + '!...');
  if (n < 2) {
    return 1;
  }
  return n * factorial(n - 1);
});

factorial(10); // 3628800
// 输出结果说明factorial(1)~factorial(10)都已经缓存了:
// start calculate 10!...
// start calculate 9!...
// start calculate 8!...
// start calculate 7!...
// start calculate 6!...
// start calculate 5!...
// start calculate 4!...
// start calculate 3!...
// start calculate 2!...
// start calculate 1!...

factorial(9); // 362880
// console无输出
```

once

顾名思义，`once()` 保证某个函数执行且仅执行一次。如果你有一个方法叫 `register()`，用户在页面上点两个按钮的任何一个都可以执行的话，就可以用 `once()` 保证函数仅调用一次，无论用户点击多少次：

```
'use strict';
----
var register = _.once(function () {
  alert('Register ok!');
});
----
// 测试效果:
register();
register();
register();
```

delay

`delay()` 可以让一个函数延迟执行，效果和 `setTimeout()` 是一样的，但是代码明显简单了：

```
'use strict';

// 2秒后调用alert():
_.delay(alert, 2000);
```

如果要延迟调用的函数有参数，把参数也传进去：

```
'use strict';

var log = _.bind(console.log, console);
_.delay(log, 2000, 'Hello, ', 'world!');
// 2秒后打印'Hello, world!':
```

更多完整的函数请参考 [underscore](http://underscorejs.org/#functions) 的文档：<http://underscorejs.org/#functions>

Objects

和 `Array` 类似，`underscore` 也提供了大量针对 `Object` 的函数。

keys / allKeys

`keys()` 可以非常方便地返回一个 `object` 自身所有的 `key`，但不包含从原型链继承下来的：

```
'use strict';

function Student(name, age) {
  this.name = name;
  this.age = age;
}

var xiaoming = new Student('小明', 20);
_.keys(xiaoming); // ['name', 'age']
```

`allKeys()` 除了 `object` 自身的 `key`，还包含从原型链继承下来的：

```
'use strict';

function Student(name, age) {
  this.name = name;
  this.age = age;
}

Student.prototype.school = 'No.1 Middle School';
var xiaoming = new Student('小明', 20);
_.allKeys(xiaoming); // ['name', 'age', 'school']
```

values

和 `keys()` 类似，`values()` 返回 `object` 自身但不包含原型链继承的所有值：

```
'use strict';

var obj = {
  name: '小明',
  age: 20
};

_.values(obj); // ['小明', 20]
```

注意，没有 `allValues()`，原因我也不知道。

mapObject

`mapObject()` 就是针对 `object` 的 `map` 版本：

```
'use strict';

var obj = { a: 1, b: 2, c: 3 };
// 注意传入的函数签名，value在前，key在后：
_.mapObject(obj, (v, k) => 100 + v); // { a: 101, b: 102, c: 103 }
```

invert

`invert()` 把 `object` 的每个 `key-value` 来个交换，`key` 变成 `value`，`value` 变成 `key`：

```
'use strict';

var obj = {
  Adam: 90,
  Lisa: 85,
  Bart: 59
};
_.invert(obj); // { '59': 'Bart', '85': 'Lisa', '90': 'Adam' }
```

extend / extendOwn

`extend()` 把多个object的key-value合并到第一个object并返回：

```
'use strict';

var a = {name: 'Bob', age: 20};
_.extend(a, {age: 15}, {age: 88, city: 'Beijing'}); // {name: 'Bob', age: 88, city: 'Beijing'}
// 变量a的内容也改变了:
a; // {name: 'Bob', age: 88, city: 'Beijing'}
```

注意：如果有相同的key，后面的object的value将覆盖前面的object的value。

`extendOwn()` 和 `extend()` 类似，但获取属性时忽略从原型链继承下来的属性。

clone

如果我们要复制一个object对象，就可以用 `clone()` 方法，它会把原有对象的所有属性都复制到新的对象中：

```
'use strict';
var source = {
  name: '小明',
  age: 20,
  skills: ['JavaScript', 'CSS', 'HTML']
};
----
var copied = _.clone(source);
----
alert(JSON.stringify(copied, null, ' '));
```

注意， `clone()` 是“浅复制”。所谓“浅复制”就是说，两个对象相同的key所引用的value其实是同一对象：

```
source.skills === copied.skills; // true
```

也就是说，修改 `source.skills` 会影响 `copied.skills`。

isEqual

`isEqual()` 对两个object进行深度比较，如果内容完全相同，则返回 `true`：

```
'use strict';

var o1 = { name: 'Bob', skills: { Java: 90, JavaScript: 99 } };
var o2 = { name: 'Bob', skills: { JavaScript: 99, Java: 90 } };

o1 === o2; // false
_.isEqual(o1, o2); // true
```

`isEqual()` 其实对 `Array` 也可以比较：

```
'use strict';

var o1 = ['Bob', { skills: ['Java', 'JavaScript'] }];
var o2 = ['Bob', { skills: ['Java', 'JavaScript'] }];

o1 === o2; // false
_.isEqual(o1, o2); // true
```

更多完整的函数请参考underscore的文档: <http://underscorejs.org/#objects>

Chaining

还记得jQuery支持链式调用吗？

```
$('#a').attr('target', '_blank')
    .append(' <i class="uk-icon-external-link"></i>')
    .click(function () {});
```

如果我们有一组操作，用underscore提供的函数，写出来像这样：

```
_.filter(_.map([1, 4, 9, 16, 25], Math.sqrt), x => x % 2 === 1);
// [1, 3, 5]
```

能不能写成链式调用？

能！

underscore提供了把对象包装成能进行链式调用的方法，就是`chain()`函数：

```
_.chain([1, 4, 9, 16, 25])
  .map(Math.sqrt)
  .filter(x => x % 2 === 1)
  .value();
// [1, 3, 5]
```

因为每一步返回的都是包装对象，所以最后一步的结果需要调用`value()`获得最终结果。

小结

通过学习underscore，是不是对JavaScript的函数式编程又有了进一步的认识？

Node.js

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端设备绝缘。

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。

其次，JavaScript语言本身是完善的函数式语言，在前端开发时，开发人员往往写得比较随意，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。

安装Node.js和npm

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7.6.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，包括勾选 **Add to Path**。

安装完成后，在Windows环境下，请打开命令提示符，然后输入 **node -v**，如果安装正常，你应该看到 **v7.6.0** 这样的输出：

```
C:\Users\IEUser>node -v
v7.6.0
```

继续在命令提示符输入 **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，例如 **100+200**，回车后将得到输出结果。

要退出Node.js环境，连接两次Ctrl+C。

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
v7.6.0
```

如果版本号小于 **v7.6.0**，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。

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
4.1.2
```

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:
...
```

上面的一大堆文字告诉你，**npm**需要跟上命令。现在我们不用关心这些命令，后面会一一讲到。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。

第一个Node程序

在前面的所有章节中，我们编写的JavaScript代码都是在浏览器中运行的，因此，我们可以直接在浏览器中敲代码，然后直接运行。

从本章开始，我们编写的JavaScript代码将~~不能~~在浏览器环境中执行了，而是在Node环境中执行，因此，JavaScript代码将直接在你的计算机上以命令行的方式运行，所以，我们要先选择一个文本编辑器来编写JavaScript代码，并且把它保存到本地硬盘的某个目录，才能够执行。

那么问题来了：文本编辑器到底哪家强？

首先，请注意，**绝对不能用Word和写字板**。Word和写字板保存的不是纯文本文件。如果我们要用记事本来编写JavaScript代码，要务必注意，记事本以UTF-8格式保存文件时，会自作聪明地在文件开始的地方加上几个特殊字符（UTF-8 BOM），结果经常会导致程序运行出现莫名其妙的错误。

所以，用记事本写代码时请注意，保存文件时使用ANSI编码，并且暂时不要输入中文。

如果你的电脑上已经安装了Sublime Text，或者Notepad++，也可以用来编写JavaScript代码，注意用UTF-8格式保存。

输入以下代码：

```
'use strict';

console.log('Hello, world.');
```

第一行总是写上 `'use strict';` 是因为我们总是以严格模式运行JavaScript代码，避免各种潜在陷阱。

然后，选择一个目录，例如 `C:\Workspace`，把文件保存为 `hello.js`，就可以打开命令行窗口，把当前目录切换到 `hello.js` 所在目录，然后输入以下命令运行这个程序了：

```
C:\Workspace>node hello.js
Hello, world.
```

也可以保存为别的名字，比如 `first.js`，但是必须要以 `.js` 结尾。此外，文件名只能是英文字母、数字和下划线的组合。

如果当前目录下没有 `hello.js` 这个文件，运行 `node hello.js` 就会报错：

```
C:\Workspace>node hello.js
module.js:338
  throw err;
      ^

Error: Cannot find module 'C:\Workspace\hello.js'
    at Function.Module._resolveFilename
    at Function.Module._load
    at Function.Module.runMain
    at startup
    at node.js
```

报错的意思就是，没有找到 `hello.js` 这个文件，因为文件不存在。这个时候，就要检查一下当前目录下是否有这个文件了。

命令行模式和Node交互模式

请注意区分命令行模式和Node交互模式。

看到类似 `C:\>` 是在Windows提供的命令行模式：



```
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 hello.js
Hello, word.

C:\Users\Michael>
```

在命令行模式下，可以执行 `node` 进入Node交互式环境，也可以执行 `node hello.js` 运行一个 `.js` 文件。

看到 `>` 是在Node交互式环境下：



```
命令提示符 - node
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
>
```

在Node交互式环境下，我们可以输入JavaScript代码并立刻执行。

此外，在命令行模式运行 `.js` 文件和在Node交互式环境下直接运行JavaScript代码有所不同。Node交互式环境会把每一行JavaScript代码的结果自动打印出来，但是，直接运行JavaScript文件却不会。

例如，在Node交互式环境下，输入：

```
> 100 + 200 + 300;
600
```

直接可以看到结果 `600`。

但是，写一个 `calc.js` 的文件，内容如下：

```
100 + 200 + 300;
```

然后在命令行模式下执行：

```
C:\Workspace>node calc.js
```

发现什么输出都没有。

这是正常的。想要输出结果，必须自己用 `console.log()` 打印出来。把 `calc.js` 改造一下：

```
console.log(100 + 200 + 300);
```

再执行，就可以看到结果：

```
C:\Workspace>node calc.js
600
```

使用严格模式

如果在JavaScript文件开头写上 `'use strict';`，那么Node在执行该JavaScript时将使用严格模式。但是，在服务器环境下，如果有很多JavaScript文件，每

个文件都写上 `'use strict';` 很麻烦。我们可以给Nodejs传递一个参数，让Node直接为所有js文件开启严格模式：

```
node --use_strict calc.js
```

后续代码，如无特殊说明，我们都会直接给Node传递 `--use_strict` 参数来开启严格模式。

小结

用文本编辑器写JavaScript程序，然后保存为后缀为 `.js` 的文件，就可以用node直接运行这个程序了。

Node的交互模式和直接运行 `.js` 文件有什么区别呢？

直接输入 `node` 进入交互模式，相当于启动了Node解释器，但是等待你一行一行地输入源代码，每输入一行就执行一行。

直接运行 `node hello.js` 文件相当于启动了Node解释器，然后一次性把 `hello.js` 文件的源代码给执行了，你是没有机会以交互的方式输入源代码的。

在编写JavaScript代码的时候，完全可以一边在文本编辑器里写代码，一边开一个Node交互式命令窗口，在写代码的过程中，把部分代码粘到命令行去验证，事半功倍！前提是得有个27'的超大显示器！

参考源码

[hello.js](#)和[calc.js](#)

搭建Node开发环境

使用文本编辑器来开发Node程序，最大的缺点是效率太低，运行Node程序还需要在命令行单独敲命令。如果还需要调试程序，就更加麻烦了。

所以我们需要一个IDE集成开发环境，让我们能在一个环境里编码、运行、调试，这样就可以大大提升开发效率。

Java的集成开发环境有Eclipse，IntelliJ idea等，C#的集成开发环境有Visual Studio，那么问题又来了：Node.js的集成开发环境到底哪家强？

考察Node.js的集成开发环境，重点放在启动速度快，执行简单，调试方便这三点上。当然，免费使用是一个加分项。

综合考察后，我们隆重向大家推荐Node.js集成开发环境：

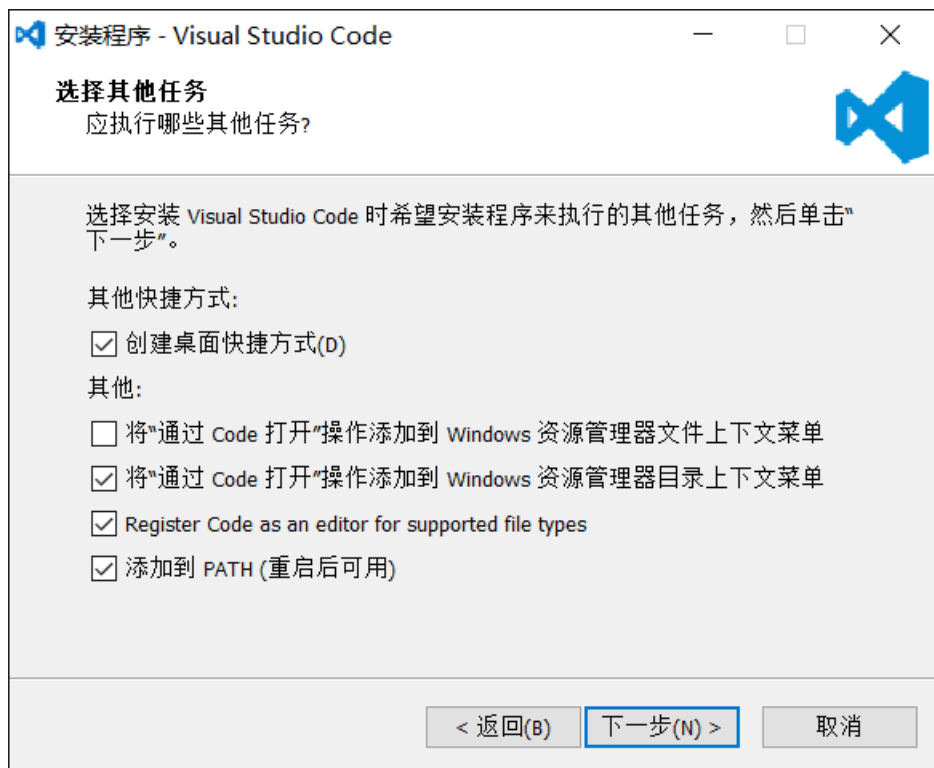
Visual Studio Code

Visual Studio Code由微软出品，但它不是那个大块头的Visual Studio，它是一个精简版的迷你Visual Studio，并且，Visual Studio Code可以跨！平！台！Windows、Mac和Linux通用。

安装Visual Studio Code

可以从Visual Studio Code的[官方网站](#)下载并安装最新的1.4版本。网速慢的童鞋请移步[国内镜像](#)。

安装过程中，请务必钩上以下选项：



☒ 将“通过Code打开”操作添加到Windows资源管理器目录上下文菜单

这将大大提升将来的操作快捷度。

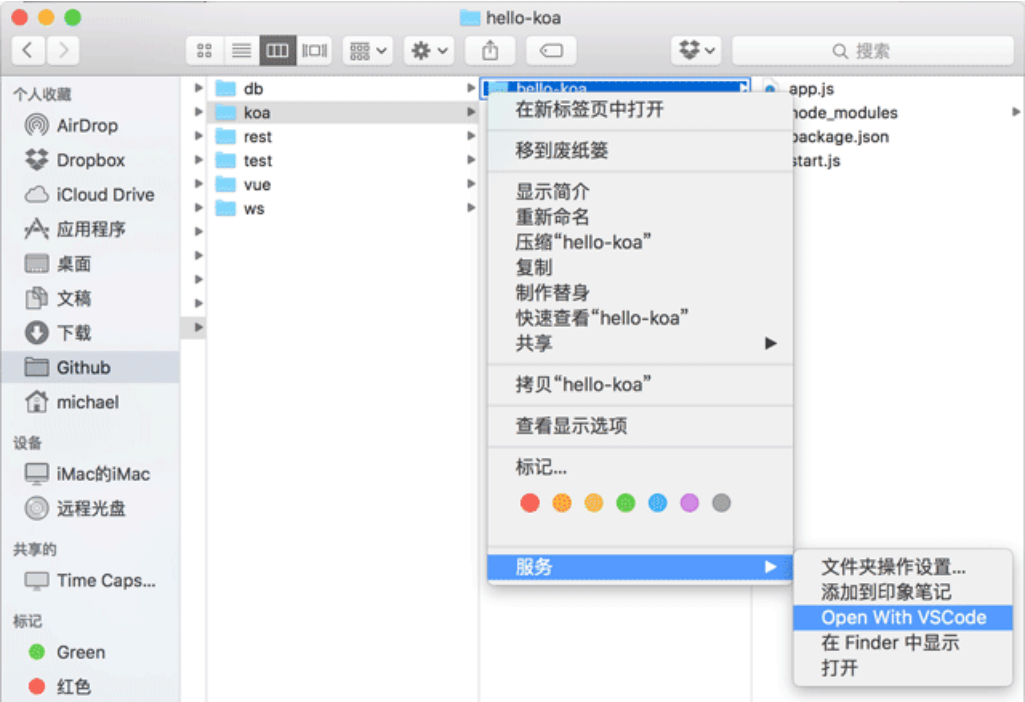
在Mac系统上，Finder选中一个目录，右键菜单并没有“通过Code打开”这个操作。不过我们可以通过Automator自己添加这个操作。

先运行Automator，选择“服务”：

1. 在右侧面板选择“服务”收到选定的“文件夹”，位于“Finder.app”，该选项是为了从Finder中接收一个文件夹；
2. 在左侧面板选择“实用工具”，然后找到“运行Shell脚本”，把它拽到右侧面板里；
3. 在右侧“运行Shell脚本”的面板里，选择Shell“/bin/bash”，传递输入“作为自变量”，然后修改Shell脚本如下：

```
for f in "$@"
do
    open -a "Visual Studio Code" "$f"
done
```

保存为“Open With VSCode”后，打开Finder，选中一个文件夹，点击右键，“服务”，就可以看到“Open With VSCode”菜单：



运行和调试JavaScript

在VS Code中，我们可以非常方便地运行JavaScript文件。

VS Code以文件夹作为工程目录（Workspace Dir），所有的JavaScript文件都存放在该目录下。此外，VS Code在工程目录下还需要一个`.vscode`的配置目录，里面存放里VS Code需要的配置文件。

假设我们在`C:\Work\`目录下创建了一个`hello`目录作为工程目录，并编写了一个`hello.js`文件，则该工程目录的结构如下：

```
hello/ <-- workspace dir
|
+- hello.js <-- JavaScript file
|
+- .vscode/ <-- VS Code config
|
|   +- launch.json <-- VS Code config file for JavaScript
```

可以用VS Code快速创建`launch.json`，然后修改如下：

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Run hello.js",
      "type": "node",
      "request": "launch",
      "program": "${workspaceRoot}/hello.js",
      "stopOnEntry": false,
      "args": [],
      "cwd": "${workspaceRoot}",
      "preLaunchTask": null,
      "runtimeExecutable": null,
      "runtimeArgs": [
        "--nolazy"
      ],
      "env": {
        "NODE_ENV": "development"
      },
      "externalConsole": false,
      "sourceMaps": false,
      "outDir": null
    }
  ]
}
```

有了配置文件，即可使用**VS Code**调试**JavaScript**。

视频演示：

参考源码

[hello.js](#)

模块

在计算机程序的开发过程中，随着程序代码越写越多，在一个文件里代码就会越来越长，越来越不容易维护。

为了编写可维护的代码，我们把很多函数分组，分别放到不同的文件里，这样，每个文件包含的代码就相对较少，很多编程语言都采用这种组织代码的方式。在Node环境中，一个.js文件就称之为一个模块（module）。

使用模块有什么好处？

最大的好处是大大提高了代码的可维护性。其次，编写代码不必从零开始。当一个模块编写完毕，就可以被其他地方引用。我们在编写程序的时候，也经常引用其他模块，包括Node内置的模块和来自第三方的模块。

使用模块还可以避免函数名和变量名冲突。相同名字的函数和变量完全可以分别存在不同的模块中，因此，我们自己在编写模块时，不必考虑名字会与其他模块冲突。

在上一节，我们编写了一个hello.js文件，这个hello.js文件就是一个模块，模块的名字就是文件名（去掉.js后缀），所以hello.js文件就是名为hello的模块。

我们把hello.js改造一下，创建一个函数，这样我们就可以在其他地方调用这个函数：

```
'use strict';

var s = 'Hello';

function greet(name) {
    console.log(s + ', ' + name + '!');
}

module.exports = greet;
```

函数greet()是我们在hello模块中定义的，你可能注意到最后一行是一个奇怪的赋值语句，它的意思是，把函数greet作为模块的输出暴露出去，这样其他模块就可以使用greet函数了。

问题是其他模块怎么使用hello模块的这个greet函数呢？我们再编写一个main.js文件，调用hello模块的greet函数：

```
'use strict';

// 引入hello模块：
var greet = require('./hello');

var s = 'Michael';

greet(s); // Hello, Michael!
```

注意到引入hello模块用Node提供的require函数：

```
var greet = require('./hello');
```

引入的模块作为变量保存在greet变量中，那greet变量到底是什么东西？其实变量greet就是在hello.js中我们用module.exports = greet;输出的greet函数。所以，main.js就成功地引用了hello.js模块中定义的greet()函数，接下来就可以直接使用它了。

在使用require()引入模块的时候，请注意模块的相对路径。因为main.js和hello.js位于同一个目录，所以我们用了当前目录.：

```
var greet = require('./hello'); // 不要忘了写相对目录！
```

如果只写模块名：

```
var greet = require('hello');
```


则Node会依次在内置模块、全局模块和当前模块下查找`hello.js`，你很可能会得到一个错误：

```
module.js
  throw err;
    ^
Error: Cannot find module 'hello'
    at Function.Module._resolveFilename
    at Function.Module._load
    ...
    at Function.Module._load
    at Function.Module.runMain
```

遇到这个错误，你要检查：

- 模块名是否写对了；
- 模块文件是否存在；
- 相对路径是否写对了。

CommonJS规范

这种模块加载机制被称为CommonJS规范。在这个规范下，每个`.js`文件都是一个模块，它们内部各自使用的变量名和函数名都互不冲突，例如，`hello.js`和`main.js`都申明了全局变量`var s = 'xxx'`，但互不影响。

一个模块想要对外暴露变量（函数也是变量），可以用`module.exports = variable;`，一个模块要引用其他模块暴露的变量，用`var ref = require('module_name');`就拿到了引用模块的变量。

结论

要在模块中对外输出变量，用：

```
module.exports = variable;
```

输出的变量可以是任意对象、函数、数组等等。

要引入其他模块输出的对象，用：

```
var foo = require('other_module');
```

引入的对象具体是什么，取决于引入模块输出的对象。

深入了解模块原理

如果你想详细地了解CommonJS的模块实现原理，请继续往下阅读。如果不了解，请直接跳到最后做练习。

当我们编写JavaScript代码时，我们可以申明全局变量：

```
var s = 'global';
```

在浏览器中，大量使用全局变量可不好。如果你在`a.js`中使用了全局变量`s`，那么，在`b.js`中也使用全局变量`s`，将造成冲突，`b.js`中对`s`赋值会改变`a.js`的运行逻辑。

也就是说，JavaScript语言本身并没有一种模块机制来保证不同模块可以使用相同的变量名。

那Node.js是如何实现这一点的？

其实要实现“模块”这个功能，并不需要语法层面的支持。Node.js也并不会增加任何JavaScript语法。实现“模块”功能的奥妙就在于JavaScript是一种函数式编程语言，它支持闭包。如果我们把一段JavaScript代码用一个函数包装起来，这段代码的所有“全局”变量就变成了函数内部的局部变量。

请注意我们编写的`hello.js`代码是这样的：

```
var s = 'Hello';
var name = 'world';

console.log(s + ' ' + name + '!');
```

Node.js加载了 `hello.js` 后，它可以把代码包装一下，变成这样执行：

```
(function () {
  // 读取的hello.js代码：
  var s = 'Hello';
  var name = 'world';

  console.log(s + ' ' + name + '!');
  // hello.js代码结束
})();
```

这样一来，原来的全局变量 `s` 现在变成了匿名函数内部的局部变量。如果Node.js继续加载其他模块，这些模块中定义的“全局”变量 `s` 也互不干扰。

所以，Node利用JavaScript的函数式编程的特性，轻而易举地实现了模块的隔离。

但是，模块的输出 `module.exports` 怎么实现？

这个也很容易实现，Node可以先准备一个对象 `module`：

```
// 准备module对象：
var module = {
  id: 'hello',
  exports: {}
};

var load = function (module) {
  // 读取的hello.js代码：
  function greet(name) {
    console.log('Hello, ' + name + '!');
  }

  module.exports = greet;
  // hello.js代码结束
  return module.exports;
};

var exported = load(module);
// 保存module：
save(module, exported);
```

可见，变量 `module` 是Node在加载js文件前准备的一个变量，并将其传入加载函数，我们在 `hello.js` 中可以直接使用变量 `module` 原因就在于它实际上是函数的一个参数：

```
module.exports = greet;
```

通过把参数 `module` 传递给 `load()` 函数，`hello.js` 就顺利地把一个变量传递给了Node执行环境，Node会把 `module` 变量保存到某个地方。

由于Node保存了所有导入的 `module`，当我们用 `require()` 获取module时，Node找到对应的 `module`，把这个 `module` 的 `exports` 变量返回，这样，另一个模块就顺利拿到了模块的输出：

```
var greet = require('./hello');
```

以上是Node实现JavaScript模块的一个简单的原理介绍。

module.exports vs exports

很多时候，你会看到，在Node环境中，有两种方法可以在一个模块中输出变量：

方法一：对`module.exports`赋值：

```
// hello.js

function hello() {
  console.log('Hello, world!');
}

function greet(name) {
  console.log('Hello, ' + name + '!');
}

module.exports = {
  hello: hello,
  greet: greet
};
```

方法二：直接使用`exports`：

```
// hello.js

function hello() {
  console.log('Hello, world!');
}

function greet(name) {
  console.log('Hello, ' + name + '!');
}

function hello() {
  console.log('Hello, world!');
}

exports.hello = hello;
exports.greet = greet;
```

但是你不可以直接对 `exports` 赋值：

```
// 代码可以执行，但是模块并没有输出任何变量：
exports = {
  hello: hello,
  greet: greet
};
```

如果你对上面的写法感到十分困惑，不要着急，我们来分析Node的加载机制：

首先，Node会把整个待加载的 `hello.js` 文件放入一个包装函数 `load` 中执行。在执行这个 `load()` 函数前，Node准备好了 `module` 变量：

```
var module = {
  id: 'hello',
  exports: {}
};
```

`load()` 函数最终返回 `module.exports`：

```
var load = function (exports, module) {
  // hello.js的文件内容
  ...
  // load函数返回:
  return module.exports;
};

var exported = load(module.exports, module);
```

也就是说，默认情况下，Node准备的`exports`变量和`module.exports`变量实际上是同一个变量，并且初始化为空对象`{}`，于是，我们可以写：

```
exports.foo = function () { return 'foo'; };
exports.bar = function () { return 'bar'; };
```

也可以写：

```
module.exports.foo = function () { return 'foo'; };
module.exports.bar = function () { return 'bar'; };
```

换句话说，Node默认给你准备了一个空对象`{}`，这样你可以直接往里面加东西。

但是，如果我们要输出的是一个函数或数组，那么，只能给`module.exports`赋值：

```
module.exports = function () { return 'foo'; };
```

给`exports`赋值是无效的，因为赋值后，`module.exports`仍然是空对象`{}`。

结论

如果要输出一个键值对象`{}`，可以利用`exports`这个已存在的空对象`{}`，并继续在上面添加新的键值；

如果要输出一个函数或数组，必须直接对`module.exports`对象赋值。

所以我们可以得出结论：直接对`module.exports`赋值，可以应对任何情况：

```
module.exports = {
  foo: function () { return 'foo'; }
};
```

或者：

```
module.exports = function () { return 'foo'; };
```

最终，我们**强烈建议**使用`module.exports = xxx`的方式来输出模块变量，这样，你只需要记忆一种方法。

练习

编写`hello.js`，输出一个或多个函数；

编写`main.js`，引入`hello`模块，调用其函数。

参考源码

[module](#)

基本模块

因为**Node.js**是运行在服务区端的**JavaScript**环境，服务器程序和浏览器程序相比，最大的特点是没有浏览器的安全限制了，而且，服务器程序必须能接收网络请求，读写文件，处理二进制内容，所以，**Node.js**内置的常用模块就是为了实现基本的服务器功能。这些模块在浏览器环境中是无法被执行的，因为它们的底层代码是用C/C++在**Node.js**运行环境中实现的。

global

在前面的**JavaScript**课程中，我们已经知道，**JavaScript**有且仅有一个全局对象，在浏览器中，叫**window**对象。而在**Node.js**环境中，也有唯一的全局对象，但不叫**window**，而叫**global**，这个对象的属性和方法也和浏览器环境的**window**不同。进入**Node.js**交互环境，可以直接输入：

```
> global.console
Console {
  log: [Function: bound ],
  info: [Function: bound ],
  warn: [Function: bound ],
  error: [Function: bound ],
  dir: [Function: bound ],
  time: [Function: bound ],
  timeEnd: [Function: bound ],
  trace: [Function: bound trace],
  assert: [Function: bound ],
  Console: [Function: Console] }
```

process

process也是**Node.js**提供的一个对象，它代表当前**Node.js**进程。通过**process**对象可以拿到许多有用信息：

```
> process === global.process;
true
> process.version;
'v5.2.0'
> process.platform;
'darwin'
> process.arch;
'x64'
> process.cwd(); //返回当前工作目录
'/Users/michael'
> process.chdir('/private/tmp'); // 切换当前工作目录
undefined
> process.cwd();
'/private/tmp'
```

JavaScript程序是由事件驱动执行的单线程模型，**Node.js**也不例外。**Node.js**不断执行响应事件的**JavaScript**函数，直到没有任何响应事件的函数可以执行时，**Node.js**就退出了。

如果我们想要在下一事件响应中执行代码，可以调用**process.nextTick()**：

```
// test.js

// process.nextTick() 将在下一轮事件循环中调用：
process.nextTick(function () {
  console.log('nextTick callback!');
});
console.log('nextTick was set!');
```

用**Node**执行上面的代码**node test.js**，你会看到，打印输出是：

```
nextTick was set!
nextTick callback!
```

这说明传入 `process.nextTick()` 的函数不是立刻执行，而是要等到下一次事件循环。

Node.js进程本身的事件就由 `process` 对象来处理。如果我们响应 `exit` 事件，就可以在程序即将退出时执行某个回调函数：

```
// 程序即将退出时的回调函数：
process.on('exit', function (code) {
    console.log('about to exit with code: ' + code);
});
```

判断JavaScript执行环境

有很多**JavaScript**代码既能在浏览器中执行，也能在**Node**环境执行，但有些时候，程序本身需要判断自己到底是在什么环境下执行的，常用的方式就是根据浏览器和**Node**环境提供的全局变量名称来判断：

```
if (typeof(window) === 'undefined') {
    console.log('node.js');
} else {
    console.log('browser');
}
```

后面，我们将介绍**Node.js**的常用内置模块。

参考源码

[gl.js](#)

fs

Node.js内置的 `fs` 模块就是文件系统模块，负责读写文件。

和所有其它JavaScript模块不同的是，`fs` 模块同时提供了异步和同步的方法。

回顾一下什么是异步方法。因为JavaScript的单线程模型，执行IO操作时，JavaScript代码无需等待，而是传入回调函数后，继续执行后续JavaScript代码。比如jQuery提供的 `getJSON()` 操作：

```
$.getJSON('http://example.com/ajax', function (data) {
    console.log('IO结果返回后执行...');
});
console.log('不等待IO结果直接执行后续代码...');
```

而同步的IO操作则需要等待函数返回：

```
// 根据网络耗时，函数将执行几十毫秒~几秒不等：
var data = getJSONSync('http://example.com/ajax');
```

同步操作的好处是代码简单，缺点是程序将等待IO操作，在等待时间内，无法响应其它任何事件。而异步读取不用等待IO操作，但代码较麻烦。

异步读文件

按照JavaScript的标准，异步读取一个文本文件的代码如下：

```
'use strict';

var fs = require('fs');

fs.readFile('sample.txt', 'utf-8', function (err, data) {
    if (err) {
        console.log(err);
    } else {
        console.log(data);
    }
});
```

请注意，`sample.txt` 文件必须在当前目录下，且文件编码为 `utf-8`。

异步读取时，传入的回调函数接收两个参数，当正常读取时，`err` 参数为 `null`，`data` 参数为读取到的String。当读取发生错误时，`err` 参数代表一个错误对象，`data` 为 `undefined`。这也是Node.js标准的回调函数：第一个参数代表错误信息，第二个参数代表结果。后面我们还会经常编写这种回调函数。

由于`err`是否为`null`就是判断是否出错的标志，所以通常的判断逻辑总是：

```
if (err) {
    // 出错了
} else {
    // 正常
}
```

如果我们要读取的文件不是文本文件，而是二进制文件，怎么办？

下面的例子演示了如何读取一个图片文件：

```
'use strict';

var fs = require('fs');

fs.readFile('sample.png', function (err, data) {
  if (err) {
    console.log(err);
  } else {
    console.log(data);
    console.log(data.length + ' bytes');
  }
});
```

当读取二进制文件时，不传入文件编码时，回调函数的 `data` 参数将返回一个 `Buffer` 对象。在Node.js中，`Buffer` 对象就是一个包含零个或多个任意字节的数组（注意和`Array`不同）。

`Buffer` 对象可以和`String`作转换，例如，把一个 `Buffer` 对象转换成`String`：

```
// Buffer -> String
var text = data.toString('utf-8');
console.log(text);
```

或者把一个`String`转换成 `Buffer`：

```
// String -> Buffer
var buf = new Buffer(text, 'utf-8');
console.log(buf);
```

同步读文件

除了标准的异步读取模式外，`fs` 也提供相应的同步读取函数。同步读取的函数和异步函数相比，多了一个 `Sync` 后缀，并且不接收回调函数，函数直接返回结果。

用 `fs` 模块同步读取一个文本文件的代码如下：

```
'use strict';

var fs = require('fs');

var data = fs.readFileSync('sample.txt', 'utf-8');
console.log(data);
```

可见，原异步调用的回调函数的 `data` 被函数直接返回，函数名需要改为 `readFileSync`，其它参数不变。

如果同步读取文件发生错误，则需要用 `try...catch` 捕获该错误：

```
try {
  var data = fs.readFileSync('sample.txt', 'utf-8');
  console.log(data);
} catch (err) {
  // 出错了
}
```

写文件

将数据写入文件是通过 `fs.writeFile()` 实现的：


```
'use strict';

var fs = require('fs');

var data = 'Hello, Node.js';
fs.writeFile('output.txt', data, function (err) {
  if (err) {
    console.log(err);
  } else {
    console.log('ok.');
```

`writeFile()` 的参数依次为文件名、数据和回调函数。如果传入的数据是String，默认按UTF-8编码写入文本文件，如果传入的参数是Buffer，则写入的是二进制文件。回调函数由于只关心成功与否，因此只需要一个err参数。

和readFile()类似，writeFile()也有一个同步方法，叫writeFileSync()：

```
'use strict';

var fs = require('fs');

var data = 'Hello, Node.js';
fs.writeFileSync('output.txt', data);
```

stat

如果我们要获取文件大小，创建时间等信息，可以使用fs.stat()，它返回一个Stat对象，能告诉我们文件或目录的详细信息：

```
'use strict';

var fs = require('fs');

fs.stat('sample.txt', function (err, stat) {
  if (err) {
    console.log(err);
  } else {
    // 是否是文件：
    console.log('isFile: ' + stat.isFile());
    // 是否是目录：
    console.log('isDirectory: ' + stat.isDirectory());
    if (stat.isFile()) {
      // 文件大小：
      console.log('size: ' + stat.size);
      // 创建时间，Date对象：
      console.log('birth time: ' + stat.birthtime);
      // 修改时间，Date对象：
      console.log('modified time: ' + stat.mtime);
    }
  }
});
```

运行结果如下：

```
isFile: true
isDirectory: false
size: 181
birth time: Fri Dec 11 2015 09:43:41 GMT+0800 (CST)
modified time: Fri Dec 11 2015 12:09:00 GMT+0800 (CST)
```

stat()也有一个对应的同步函数statSync()，请试着改写上述异步代码为同步代码。

异步还是同步

在 `fs` 模块中，提供同步方法是为了方便使用。那我们到底是应该用异步方法还是同步方法呢？

由于Node环境执行的JavaScript代码是服务器端代码，所以，绝大部分需要在服务器运行期反复执行业务逻辑的代码，*必须使用异步代码*，否则，同步代码在执行时期，服务器将停止响应，因为JavaScript只有一个执行线程。

服务器启动时如果需要读取配置文件，或者结束时需要写入到状态文件时，可以使用同步代码，因为这些代码只在启动和结束时执行一次，不影响服务器正常运行时的异步执行。

参考源码

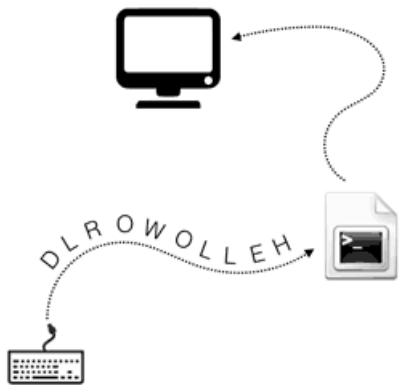
`fs`

stream

`stream` 是 Node.js 提供的又一个仅在服务区端可用的模块，目的是支持“流”这种数据结构。

什么是流？流是一种抽象的数据结构。想象水流，当在水管中流动时，就可以从某个地方（例如自来水厂）源源不断地到达另一个地方（比如你家的洗手池）。我们也可以把数据看成是数据流，比如你敲键盘的时候，就可以把每个字符依次连起来，看成字符流。这个流是从键盘输入到应用程序，实际上它还对应着一个名字：标准输入流（`stdin`）。

如果应用程序把字符一个一个输出到显示器上，这也可以看成是一个流，这个流也有名字：标准输出流（`stdout`）。流的特点是数据是有序的，而且必须依次读取，或者依次写入，不能像 `Array` 那样随机定位。



有些流用来读取数据，比如从文件读取数据时，可以打开一个文件流，然后从文件流中不断地读取数据。有些流用来写入数据，比如向文件写入数据时，只需要把数据不断地往文件流中写进去就可以了。

在 Node.js 中，流也是一个对象，我们只需要响应流的事件就可以了：`data` 事件表示流的数据已经可以读取了，`end` 事件表示这个流已经到了末尾了，没有数据可以读取了，`error` 事件表示出错了。

下面是一个从文件流读取文本内容的示例：

```
'use strict';

var fs = require('fs');

// 打开一个流：
var rs = fs.createReadStream('sample.txt', 'utf-8');

rs.on('data', function (chunk) {
  console.log('DATA:')
  console.log(chunk);
});

rs.on('end', function () {
  console.log('END');
});

rs.on('error', function (err) {
  console.log('ERROR: ' + err);
});
```

要注意，`data` 事件可能会有多次，每次传递的 `chunk` 是流的一部分数据。

要以流的形式写入文件，只需要不断调用 `write()` 方法，最后以 `end()` 结束：

```
'use strict';

var fs = require('fs');

var ws1 = fs.createWriteStream('output1.txt', 'utf-8');
ws1.write('使用Stream写入文本数据...\n');
ws1.write('END.');
```

```
var ws2 = fs.createWriteStream('output2.txt');
ws2.write(new Buffer('使用Stream写入二进制数据...\n', 'utf-8'));
ws2.write(new Buffer('END.', 'utf-8'));
ws2.end();
```

所有可以读取数据的流都继承自 `stream.Readable`，所有可以写入的流都继承自 `stream.Writable`。

pipe

就像可以把两个水管串成一个更长的水管一样，两个流也可以串起来。一个 `Readable` 流和一个 `Writable` 流串起来后，所有的数据自动从 `Readable` 流进入 `Writable` 流，这种操作叫 `pipe`。

在Node.js中，`Readable` 流有一个 `pipe()` 方法，就是用来干这件事的。

让我们用 `pipe()` 把一个文件流和另一个文件流串起来，这样源文件的所有数据就自动写入到目标文件里了，所以，这实际上是一个复制文件的程序：

```
'use strict';

var fs = require('fs');

var rs = fs.createReadStream('sample.txt');
var ws = fs.createWriteStream('copied.txt');

rs.pipe(ws);
```

默认情况下，当 `Readable` 流的数据读取完毕，`end` 事件触发后，将自动关闭 `Writable` 流。如果我们不希望自动关闭 `Writable` 流，需要传入参数：

```
readable.pipe(writable, { end: false });
```

参考源码

[stream](#)

http

Node.js开发的目的是为了用JavaScript编写Web服务器程序。因为JavaScript实际上已经统治了浏览器端的脚本，其优势就是有世界上数量最多的前端开发人员。如果已经掌握了JavaScript前端开发，再学习一下如何将JavaScript应用在后端开发，就是名副其实的**全栈**了。

HTTP协议

要理解Web服务器程序的工作原理，首先，我们要对HTTP协议有基本的了解。如果你对HTTP协议不太熟悉，先看一看[HTTP协议简介](#)。

HTTP服务器

要开发HTTP服务器程序，从头处理TCP连接，解析HTTP是不现实的。这些工作实际上已经由Node.js自带的 `http` 模块完成了。应用程序并不直接和HTTP协议打交道，而是操作 `http` 模块提供的 `request` 和 `response` 对象。

`request` 对象封装了HTTP请求，我们调用 `request` 对象的属性和方法就可以拿到所有HTTP请求的信息：

`response` 对象封装了HTTP响应，我们操作 `response` 对象的方法，就可以把HTTP响应返回给浏览器。

用Node.js实现一个HTTP服务器程序非常简单。我们来实现一个最简单的Web程序 `hello.js`，它对于所有请求，都返回 `Hello world!`：

```
'use strict';

// 导入http模块：
var http = require('http');

// 创建http server，并传入回调函数：
var server = http.createServer(function (request, response) {
  // 回调函数接收request和response对象，
  // 获得HTTP请求的method和url：
  console.log(request.method + '：' + request.url);
  // 将HTTP响应200写入response，同时设置Content-Type: text/html：
  response.writeHead(200, {'Content-Type': 'text/html'});
  // 将HTTP响应的HTML内容写入response：
  response.end('<h1>Hello world!</h1>');
});

// 让服务器监听8080端口：
server.listen(8080);

console.log('Server is running at http://127.0.0.1:8080/');
```

在命令提示符下运行该程序，可以看到以下输出：

```
$ node hello.js
Server is running at http://127.0.0.1:8080/
```

不要关闭命令提示符，直接打开浏览器输入 `http://localhost:8080`，即可看到服务器响应的内容：



同时，在命令提示符窗口，可以看到程序打印的请求信息：

```
GET: /  
GET: /favicon.ico
```

这就是我们编写的第一个HTTP服务器程序！

文件服务器

让我们继续扩展一下上面的Web程序。我们可以设定一个目录，然后让Web程序变成一个文件服务器。要实现这一点，我们只需要解析 `request.url` 中的路径，然后在本地找到对应的文件，把文件内容发送出去就可以了。

解析URL需要用到Node.js提供的 `url` 模块，它使用起来非常简单，通过 `parse()` 将一个字符串解析为一个 `Url` 对象：

```
'use strict';  
  
var url = require('url');  
  
console.log(url.parse('http://user:pass@host.com:8080/path/to/file?query=string#hash'));
```

结果如下：

```
Url {  
  protocol: 'http:',  
  slashes: true,  
  auth: 'user:pass',  
  host: 'host.com:8080',  
  port: '8080',  
  hostname: 'host.com',  
  hash: '#hash',  
  search: '?query=string',  
  query: 'query=string',  
  pathname: '/path/to/file',  
  path: '/path/to/file?query=string',  
  href: 'http://user:pass@host.com:8080/path/to/file?query=string#hash' }
```

处理本地文件目录需要使用Node.js提供的 `path` 模块，它可以方便地构造目录：

```
'use strict';

var path = require('path');

// 解析当前目录：
var workDir = path.resolve('.'); // '/Users/michael'

// 组合完整的文件路径:当前目录+'pub'+index.html':
var filePath = path.join(workDir, 'pub', 'index.html');
// '/Users/michael/pub/index.html'
```

使用 `path` 模块可以正确处理操作系统相关的文件路径。在Windows系统下，返回的路径类似于 `C:\Users\michael\static\index.html`，这样，我们就不关心怎么拼接路径了。

最后，我们实现一个文件服务器 `file_server.js`：

```
'use strict';

var
  fs = require('fs'),
  url = require('url'),
  path = require('path'),
  http = require('http');

// 从命令行参数获取root目录，默认是当前目录：
var root = path.resolve(process.argv[2] || '.');

console.log('Static root dir: ' + root);

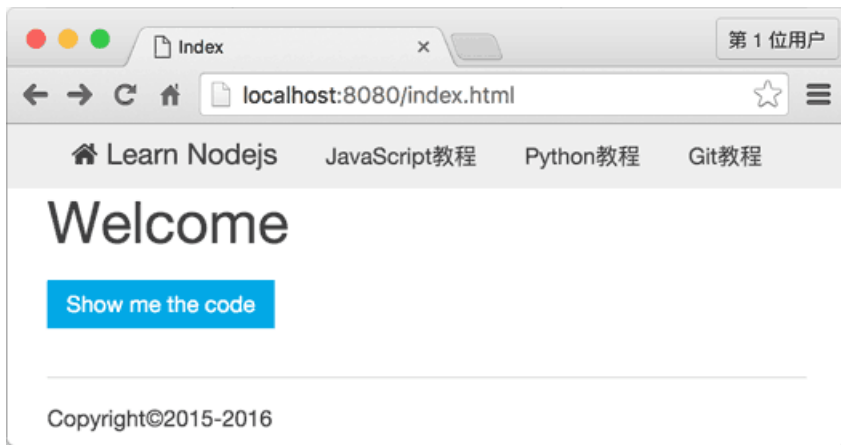
// 创建服务器：
var server = http.createServer(function (request, response) {
  // 获得URL的path，类似 '/css/bootstrap.css':
  var pathname = url.parse(request.url).pathname;
  // 获得对应的本地文件路径，类似 '/srv/www/css/bootstrap.css':
  var filepath = path.join(root, pathname);
  // 获取文件状态：
  fs.stat(filepath, function (err, stats) {
    if (!err && stats.isFile()) {
      // 没有出错并且文件存在：
      console.log('200 ' + request.url);
      // 发送200响应：
      response.writeHead(200);
      // 将文件流导向response：
      fs.createReadStream(filepath).pipe(response);
    } else {
      // 出错了或者文件不存在：
      console.log('404 ' + request.url);
      // 发送404响应：
      response.writeHead(404);
      response.end('404 Not Found');
    }
  });
});

server.listen(8080);

console.log('Server is running at http://127.0.0.1:8080/');
```

没有必要手动读取文件内容。由于 `response` 对象本身是一个 `Writable Stream`，直接用 `pipe()` 方法就实现了自动读取文件内容并输出到HTTP响应。

在命令行运行 `node file_server.js /path/to/dir`，把 `/path/to/dir` 改成你本地的一个有效的目录，然后在浏览器中输入 `http://localhost:8080/index.html`：



只要当前目录下存在文件`index.html`，服务器就可以把文件内容发送给浏览器。观察控制台输出：

```
200 /index.html
200 /css/uikit.min.css
200 /js/jquery.min.js
200 /fonts/fontawesome-webfont.woff2
```

第一个请求是浏览器请求`index.html`页面，后续请求是浏览器解析HTML后发送的其它资源请求。

练习

在浏览器输入`http://localhost:8080/`时，会返回404，原因是程序识别出HTTP请求的不是文件，而是目录。请修改`file_server.js`，如果遇到请求的路径是目录，则自动在目录下依次搜索`index.html`、`default.html`，如果找到了，就返回HTML文件的内容。

参考源码

[http服务器代码](#)（含静态网站）

crypto

crypto模块的目的是为了提供通用的加密和哈希算法。用纯**JavaScript**代码实现这些功能不是不可能，但速度会非常慢。**Nodejs**用**C/C++**实现这些算法后，通过**crypto**这个模块暴露为**JavaScript**接口，这样用起来方便，运行速度也快。

MD5和SHA1

MD5是一种常用的哈希算法，用于给任意数据一个“签名”。这个签名通常用一个十六进制的字符串表示：

```
const crypto = require('crypto');

const hash = crypto.createHash('md5');

// 可任意多次调用update():
hash.update('Hello, world!');
hash.update('Hello, nodejs!');

console.log(hash.digest('hex')); // 7e1977739c748beac0c0fd14fd26a544
```

update()方法默认字符串编码为**UTF-8**，也可以传入**Buffer**。

如果要计算**SHA1**，只需要把**'md5'**改成**'sha1'**，就可以得到**SHA1**的结果**1f32b9c9932c02227819a4151feed43e131aca40**。

还可以使用更安全的**sha256**和**sha512**。

Hmac

Hmac算法也是一种哈希算法，它可以利用**MD5**或**SHA1**等哈希算法。不同的是，**Hmac**还需要一个密钥：

```
const crypto = require('crypto');

const hmac = crypto.createHmac('sha256', 'secret-key');

hmac.update('Hello, world!');
hmac.update('Hello, nodejs!');

console.log(hmac.digest('hex')); // 80f7e22570...
```

只要密钥发生了变化，那么同样的输入数据也会得到不同的签名，因此，可以把**Hmac**理解为用随机数“增强”的哈希算法。

AES

AES是一种常用的对称加密算法，加解密都用同一个密钥。**crypto**模块提供了**AES**支持，但是需要自己封装好函数，便于使用：

```
const crypto = require('crypto');

function aesEncrypt(data, key) {
  const cipher = crypto.createCipher('aes192', key);
  var crypted = cipher.update(data, 'utf8', 'hex');
  crypted += cipher.final('hex');
  return crypted;
}

function aesDecrypt(encrypted, key) {
  const decipher = crypto.createDecipher('aes192', key);
  var decrypted = decipher.update(encrypted, 'hex', 'utf8');
  decrypted += decipher.final('utf8');
  return decrypted;
}

var data = 'Hello, this is a secret message!';
var key = 'Password!';
var encrypted = aesEncrypt(data, key);
var decrypted = aesDecrypt(encrypted, key);

console.log('Plain text: ' + data);
console.log('Encrypted text: ' + encrypted);
console.log('Decrypted text: ' + decrypted);
```

运行结果如下：

```
Plain text: Hello, this is a secret message!
Encrypted text: 8a944d97bdabc157a5b7a40cb180e7...
Decrypted text: Hello, this is a secret message!
```

可以看出，加密后的字符串通过解密又得到了原始内容。

注意到AES有很多不同的算法，如[aes192](#)，[aes-128-ecb](#)，[aes-256-cbc](#)等，AES除了密钥外还可以指定IV（Initial Vector），不同的系统只要IV不同，用相同的密钥加密相同的数据得到的加密结果也是不同的。加密结果通常有两种表示方法：[hex](#)和[base64](#)，这些功能Nodejs全部都支持，但是在应用中要注意，如果加解密双方一方用Nodejs，另一方用Java、PHP等其它语言，需要仔细测试。如果无法正确解密，要确认双方是否遵循同样的AES算法，字符串密钥和IV是否相同，加密后的数据是否统一为hex或base64格式。

Diffie-Hellman

DH算法是一种密钥交换协议，它可以让双方在不泄漏密钥的情况下协商出一个密钥来。DH算法基于数学原理，比如小明和小红想要协商一个密钥，可以这么做：

1. 小明先选一个素数和一个底数，例如，素数 $p=23$ ，底数 $g=5$ （底数可以任选），再选择一个秘密整数 $a=6$ ，计算 $A=g^a \bmod p=8$ ，然后大声告诉小红： $p=23, g=5, A=8$ ；
2. 小红收到小明发来的 p ， g ， A 后，也选一个秘密整数 $b=15$ ，然后计算 $B=g^b \bmod p=19$ ，并大声告诉小明： $B=19$ ；
3. 小明自己计算出 $s=B^a \bmod p=2$ ，小红也自己计算出 $s=A^b \bmod p=2$ ，因此，最终协商的密钥 s 为 2 。

在这个过程中，密钥 2 并不是小明告诉小红的，也不是小红告诉小明的，而是双方协商计算出来的。第三方只能知道 $p=23$ ， $g=5$ ， $A=8$ ， $B=19$ ，由于不知道双方选的秘密整数 $a=6$ 和 $b=15$ ，因此无法计算出密钥 2 。

用crypto模块实现DH算法如下：

```
const crypto = require('crypto');

// xiaoming's keys:
var ming = crypto.createDiffieHellman(512);
var ming_keys = ming.generateKeys();

var prime = ming.getPrime();
var generator = ming.getGenerator();

console.log('Prime: ' + prime.toString('hex'));
console.log('Generator: ' + generator.toString('hex'));

// xiaohong's keys:
var hong = crypto.createDiffieHellman(prime, generator);
var hong_keys = hong.generateKeys();

// exchange and generate secret:
var ming_secret = ming.computeSecret(hong_keys);
var hong_secret = hong.computeSecret(ming_keys);

// print secret:
console.log('Secret of Xiao Ming: ' + ming_secret.toString('hex'));
console.log('Secret of Xiao Hong: ' + hong_secret.toString('hex'));
```

运行后，可以得到如下输出：

```
$ node dh.js
Prime: a8224c...deead3
Generator: 02
Secret of Xiao Ming: 695308...d519be
Secret of Xiao Hong: 695308...d519be
```

注意每次输出都不一样，因为素数的选择是随机的。

证书

crypto模块也可以处理数字证书。数字证书通常用在**SSL**连接，也就是**Web**的**https**连接。一般情况下，**https**连接只需要处理服务器端的单向认证，如无特殊需求（例如自己作为**Root**给客户发认证证书），建议用反向代理服务器如**Nginx**等**Web**服务器去处理证书。

参考源码

[crypto常用算法](#)

Web开发

最早的软件都是运行在大型机上的，软件使用者通过“哑终端”登录到大型机上去运行软件。后来随着PC机的兴起，软件开始主要运行在桌面上，而数据库这样的软件运行在服务器端，这种Client/Server模式简称CS架构。

随着互联网的兴起，人们发现，CS架构不适合Web，最大的原因是Web应用程序的修改和升级非常迅速，而CS架构需要每个客户端逐个升级桌面App，因此，Browser/Server模式开始流行，简称BS架构。

在BS架构下，客户端只需要浏览器，应用程序的逻辑和数据都存储在服务器端。浏览器只需要请求服务器，获取Web页面，并把Web页面展示给用户即可。

当然，Web页面也具有极强的交互性。由于Web页面是用HTML编写的，而HTML具备超强的表现力，并且，服务器端升级后，客户端无需任何部署就可以使用到新的版本，因此，BS架构迅速流行起来。

今天，除了重量级的软件如Office，Photoshop等，大部分软件都以Web形式提供。比如，新浪提供的新闻、博客、微博等服务，均是Web应用。

Web应用开发可以说是目前软件开发中最重要的部分。Web开发也经历了好几个阶段：

静态Web页面：由文本编辑器直接编辑并生成静态的HTML页面，如果要修改Web页面的内容，就需要再次编辑HTML源文件，早期的互联网Web页面就是静态的；

CGI：由于静态Web页面无法与用户交互，比如用户填写了一个注册表单，静态Web页面就无法处理。要处理用户发送的动态数据，出现了Common Gateway Interface，简称CGI，用C/C++编写。

ASP/JSP/PHP：由于Web应用特点是修改频繁，用C/C++这样的低级语言非常不适合Web开发，而脚本语言由于开发效率高，与HTML结合紧密，因此，迅速取代了CGI模式。ASP是微软推出的用VBScript脚本编程的Web开发技术，而JSP用Java来编写脚本，PHP本身则是开源的脚本语言。

MVC：为了解决直接用脚本语言嵌入HTML导致的可维护性差的问题，Web应用也引入了Model-View-Controller的模式，来简化Web开发。ASP发展为ASP.Net，JSP和PHP也有一大堆MVC框架。

目前，Web开发技术仍在快速发展中，异步开发、新的MVVM前端技术层出不穷。

由于Node.js把JavaScript引入了服务器端，因此，原来必须使用PHP/Java/C#/Python/Ruby等其他语言来开发服务器端程序，现在可以使用Node.js开发了！

用Node.js开发Web服务器端，有几个显著的优势：

一是后端语言也是JavaScript，以前掌握了前端JavaScript的开发人员，现在可以同时编写后端代码；

二是前后端统一使用JavaScript，就没有切换语言的障碍了；

三是速度快，非常快！这得益于Node.js天生是异步的。

在Node.js诞生后的短短几年里，出现了无数种Web框架、ORM框架、模版引擎、测试框架、自动化构建工具，数量之多，即使是JavaScript老司机，也不免眼花缭乱。

常见的Web框架包括：[Express](#)，[Sails.js](#)，[koa](#)，[Meteor](#)，[DerbyJS](#)，[Total.js](#)，[restify](#).....

ORM框架比Web框架要少一些：[Sequelize](#)，[ORM2](#)，[Bookshelf.js](#)，[Objection.js](#).....

模版引擎PK：[Jade](#)，[EJS](#)，[Swig](#)，[Nunjucks](#)，[doT.js](#).....

测试框架包括：[Mocha](#)，[Expresso](#)，[Unit.js](#)，[Karma](#).....

构建工具有：[Grunt](#)，[Gulp](#)，[Webpack](#).....

目前，在npm上已发布的开源Node.js模块数量超过了30万个。

有选择恐惧症的朋友，看到这里可以洗洗睡了。

好消息是这个教程已经帮你选好了，你只需要跟着教程一条道走到黑就可以了。

koa

koa是Express的下一代基于Node.js的web框架，目前有1.x和2.0两个版本。

历史

1. Express

Express是第一代最流行的web框架，它对Node.js的http进行了封装，用起来如下：

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
});
```

虽然Express的API很简单，但是它是基于ES5的语法，要实现异步代码，只有一个方法：回调。如果异步嵌套层次过多，代码写起来就非常难看：

```
app.get('/test', function (req, res) {
  fs.readFile('/file1', function (err, data) {
    if (err) {
      res.status(500).send('read file1 error');
    }
    fs.readFile('/file2', function (err, data) {
      if (err) {
        res.status(500).send('read file2 error');
      }
      res.type('text/plain');
      res.send(data);
    });
  });
});
```

虽然可以用async这样的库来组织异步代码，但是用回调写异步实在是太痛苦了！

2. koa 1.0

随着新版Node.js开始支持ES6，Express的团队又基于ES6的generator重新编写了下一代web框架koa。和Express相比，koa 1.0使用generator实现异步，代码看起来像同步的：

```
var koa = require('koa');
var app = koa();

app.use('/test', function *() {
  yield doReadFile1();
  var data = yield doReadFile2();
  this.body = data;
});

app.listen(3000);
```

用generator实现异步比回调简单了不少，但是generator的本意并不是异步。Promise才是为异步设计的，但是Promise的写法.....想想就复杂。为了简化异步代码，ES7（目前是草案，还没有发布）引入了新的关键字`async`和`await`，可以轻松地把一个function变为异步模式：

```
async function () {  
  var data = await fs.read('/file1');  
}
```

这是JavaScript未来标准的异步代码，非常简洁，并且易于使用。

3. koa2

koa团队并没有止步于koa 1.0，他们非常超前地基于ES7开发了koa2，和koa 1相比，koa2完全使用Promise并配合 `async` 来实现异步。

koa2的代码看上去像这样：

```
app.use(async (ctx, next) => { await next(); var data = await doReadFile(); ctx.response.type = 'text/plain'; ctx.response.body = data; });
```

出于兼容性考虑，目前koa 2仍支持generator的写法，但下一个版本将会去掉。

选择哪个版本？

目前JavaScript处于高速进化中，ES7是大势所趋。为了紧跟时代潮流，教程将使用最新的koa 2开发！

koa入门

创建koa2工程

首先，我们创建一个目录 `hello-koa` 并作为工程目录用VS Code打开。然后，我们创建 `app.js`，输入以下代码：

```
// 导入koa，和koa 1.x不同，在koa2中，我们导入的是一个class，因此用大写的Koa表示：
const Koa = require('koa');

// 创建一个Koa对象表示web app本身：
const app = new Koa();

// 对于任何请求，app将调用该异步函数处理请求：
app.use(async (ctx, next) => {
  await next();
  ctx.response.type = 'text/html';
  ctx.response.body = '<h1>Hello, koa2!</h1>';
});

// 在端口3000监听：
app.listen(3000);
console.log('app started at port 3000...');
```

对于每一个http请求，koa将调用我们传入的异步函数来处理：

```
async (ctx, next) => {
  await next();
  // 设置response的Content-Type：
  ctx.response.type = 'text/html';
  // 设置response的内容：
  ctx.response.body = '<h1>Hello, koa2!</h1>';
}
```

其中，参数 `ctx` 是由koa传入的封装了request和response的变量，我们可以通过它访问request和response，`next` 是koa传入的将要处理的下一个异步函数。

上面的异步函数中，我们首先用 `await next();` 处理下一个异步函数，然后，设置response的Content-Type和内容。

由 `async` 标记的函数称为异步函数，在异步函数中，可以用 `await` 调用另一个异步函数，这两个关键字将在ES7中引入。

现在我们遇到第一个问题：koa这个包怎么装，`app.js` 才能正常导入它？

方法一：可以用npm命令直接安装koa。先打开命令提示符，务必把当前目录切换到 `hello-koa` 这个目录，然后执行命令：

```
C:\...\hello-koa> npm install koa@2.0.0
```

npm会把koa2以及koa2依赖的所有包全部安装到当前目录的node_modules目录下。

方法二：在 `hello-koa` 这个目录下创建一个 `package.json`，这个文件描述了我们的 `hello-koa` 工程会用到哪些包。完整的文件内容如下：

```
{
  "name": "hello-koa2",
  "version": "1.0.0",
  "description": "Hello Koa 2 example with async",
  "main": "app.js",
  "scripts": {
    "start": "node app.js"
  },
  "keywords": [
    "koa",
    "async"
  ],
  "author": "Michael Liao",
  "license": "Apache-2.0",
  "repository": {
    "type": "git",
    "url": "https://github.com/michaelliao/learn-javascript.git"
  },
  "dependencies": {
    "koa": "2.0.0"
  }
}
```

其中，`dependencies` 描述了我们的工程依赖的包以及版本号。其他字段均用来描述项目信息，可任意填写。

然后，我们在 `hello-koa` 目录下执行 `npm install` 就可以把所需包以及依赖包一次性全部装好：

```
C:\...\hello-koa> npm install
```

很显然，第二个方法更靠谱，因为我们只要在 `package.json` 正确设置了依赖，`npm` 就会把所有用到的包都装好。

注意，任何时候都可以直接删除整个 `node_modules` 目录，因为用 `npm install` 命令可以完整地重新下载所有依赖。并且，这个目录不应该被放入版本控制中。

现在，我们的工程结构如下：

```
hello-koa/
|
+- .vscode/
| |
| +- launch.json <-- VSCode 配置文件
|
+- app.js <-- 使用koa的js
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

紧接着，我们在 `package.json` 中添加依赖包：

```
"dependencies": {
  "koa": "2.0.0"
}
```

然后使用 `npm install` 命令安装后，在 VS Code 中执行 `app.js`，调试控制台输出如下：

```
node --debug-brk=40645 --nolazy app.js
Debugger listening on port 40645
app started at port 3000...
```

我们打开浏览器，输入 `http://localhost:3000`，即可看到效果：



还可以直接用命令 `node app.js` 在命令行启动程序，或者用 `npm start` 启动。`npm start` 命令会让npm执行定义在 `package.json` 文件中的 `start` 对应命令：

```
"scripts": {
  "start": "node app.js"
}
```

koa middleware

让我们再仔细看看koa的执行逻辑。核心代码是：

```
app.use(async (ctx, next) => {
  await next();
  ctx.response.type = 'text/html';
  ctx.response.body = '<h1>Hello, koa2!</h1>';
});
```

每收到一个http请求，koa就会调用通过 `app.use()` 注册的 `async` 函数，并传入 `ctx` 和 `next` 参数。

我们可以对 `ctx` 操作，并设置返回内容。但是为什么要调用 `await next()`？

原因是koa把很多 `async` 函数组成一个处理链，每个 `async` 函数都可以做一些自己的事情，然后用 `await next()` 来调用下一个 `async` 函数。我们把每个 `async` 函数称为 `middleware`，这些 `middleware` 可以组合起来，完成很多有用的功能。

例如，可以用以下3个 `middleware` 组成处理链，依次打印日志，记录处理时间，输出HTML：

```
app.use(async (ctx, next) => {
  console.log(`${ctx.request.method} ${ctx.request.url}`); // 打印URL
  await next(); // 调用下一个middleware
});

app.use(async (ctx, next) => {
  const start = new Date().getTime(); // 当前时间
  await next(); // 调用下一个middleware
  const ms = new Date().getTime() - start; // 耗时间
  console.log(`Time: ${ms}ms`); // 打印耗时间
});

app.use(async (ctx, next) => {
  await next();
  ctx.response.type = 'text/html';
  ctx.response.body = '<h1>Hello, koa2!</h1>';
});
```

`middleware` 的顺序很重要，也就是调用 `app.use()` 的顺序决定了 `middleware` 的顺序。

此外，如果一个 `middleware` 没有调用 `await next()`，会怎么办？答案是后续的 `middleware` 将不再执行了。这种情况也很常见，例如，一个检测用户权限的 `middleware` 可以决定是否继续处理请求，还是直接返回403错误：

```
app.use(async (ctx, next) => {
  if (await checkUserPermission(ctx)) {
    await next();
  } else {
    ctx.response.status = 403;
  }
});
```

理解了middleware，我们就已经会用koa了！

最后注意 `ctx` 对象有一些简写的方法，例如 `ctx.url` 相当于 `ctx.request.url`，`ctx.type` 相当于 `ctx.response.type`。

参考源码

[hello-koa](#)

处理URL

在hello-koa工程中，我们处理http请求一律返回相同的HTML，这样虽然非常简单，但是用浏览器一测，随便输入任何URL都会返回相同的网页。



正常情况下，我们应该对不同的URL调用不同的处理函数，这样才能返回不同的结果。例如像这样写：

```
app.use(async (ctx, next) => {
  if (ctx.request.path === '/') {
    ctx.response.body = 'index page';
  } else {
    await next();
  }
});

app.use(async (ctx, next) => {
  if (ctx.request.path === '/test') {
    ctx.response.body = 'TEST page';
  } else {
    await next();
  }
});

app.use(async (ctx, next) => {
  if (ctx.request.path === '/error') {
    ctx.response.body = 'ERROR page';
  } else {
    await next();
  }
});
```

这么写是可以运行的，但是好像有点蠢。

应该有一个能集中处理URL的middleware，它根据不同的URL调用不同的处理函数，这样，我们才能专心为每个URL编写处理函数。

koa-router

为了处理URL，我们需要引入 `koa-router` 这个middleware，让它负责处理URL映射。

我们把上一节的 `hello-koa` 工程复制一份，重命名为 `url-koa`。

先在 `package.json` 中添加依赖项：

```
"koa-router": "7.0.0"
```

然后用 `npm install` 安装。

接下来，我们修改 `app.js`，使用 `koa-router` 来处理URL：

```

const Koa = require('koa');

// 注意require('koa-router')返回的是函数：
const router = require('koa-router')();

const app = new Koa();

// log request URL:
app.use(async (ctx, next) => {
  console.log(`Process ${ctx.request.method} ${ctx.request.url}...`);
  await next();
});

// add url-route:
router.get('/hello/:name', async (ctx, next) => {
  var name = ctx.params.name;
  ctx.response.body = `

# Hello, ${name}!</h1>`; }); router.get('/', async (ctx, next) => { ctx.response.body = '<h1>Index</h1>'; }); // add router middleware: app.use(router.routes()); app.listen(3000); console.log('app started at port 3000...');


```

注意导入 `koa-router` 的语句最后的 `()` 是函数调用：

```
const router = require('koa-router')();
```

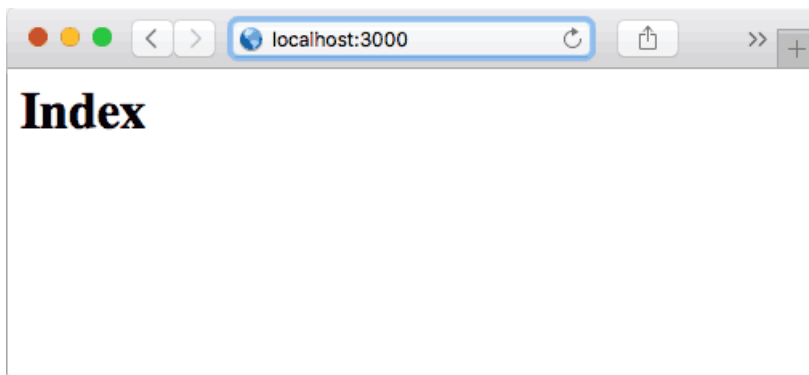
相当于：

```
const fn_router = require('koa-router');
const router = fn_router();
```

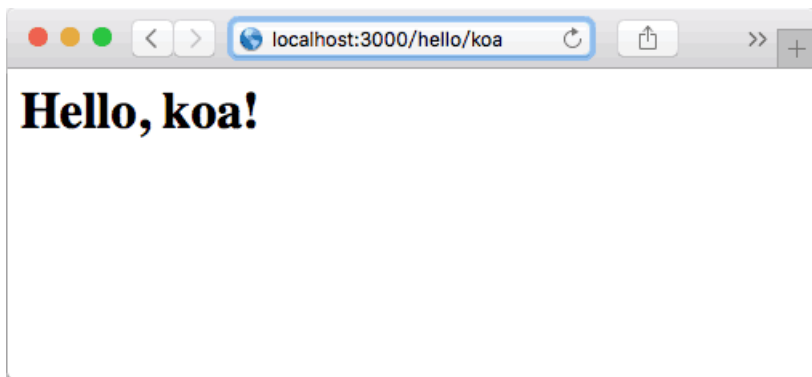
然后，我们使用 `router.get('/path', async fn)` 来注册一个GET请求。可以在请求路径中使用带变量的 `/hello/:name`，变量可以通过 `ctx.params.name` 访问。

再运行 `app.js`，我们就可以测试不同的URL：

输入首页：<http://localhost:3000/>



输入：<http://localhost:3000/hello/koa>



处理post请求

用 `router.get('/path', async fn)` 处理的是get请求。如果要处理post请求，可以用 `router.post('/path', async fn)`。

用post请求处理URL时，我们会遇到一个问题：post请求通常会发送一个表单，或者JSON，它作为request的body发送，但无论是Node.js提供的原始request对象，还是koa提供的request对象，都 **不提供** 解析request的body的功能！

所以，我们又需要引入另一个middleware来解析原始request请求，然后，把解析后的参数，绑定到 `ctx.request.body` 中。

`koa-bodyparser` 就是用来干这个活的。

我们在 `package.json` 中添加依赖项：

```
"koa-bodyparser": "3.2.0"
```

然后使用 `npm install` 安装。

下面，修改 `app.js`，引入 `koa-bodyparser`：

```
const bodyParser = require('koa-bodyparser');
```

在合适的位置加上：

```
app.use(bodyParser());
```

由于middleware的顺序很重要，这个 `koa-bodyparser` 必须在 `router` 之前被注册到 `app` 对象上。

现在我们可以处理post请求了。写一个简单的登录表单：

```
router.get('/', async (ctx, next) => {
  ctx.response.body = `

# 


```

注意到我们用 `var name = ctx.request.body.name || ''` 拿到表单的 `name` 字段，如果该字段不存在，默认值设置为 `''`。

类似的，`put`、`delete`、`head` 请求也可以由 `router` 处理。

重构

现在，我们已经可以处理不同的 URL 了，但是看看 `app.js`，总觉得还是有点不对劲。



所有的 URL 处理函数都放到 `app.js` 里显得很乱，而且，每加一个 URL，就需要修改 `app.js`。随着 URL 越来越多，`app.js` 就会越来越长。

如果能把 URL 处理函数集中到某个 js 文件，或者某几个 js 文件中就好了，然后让 `app.js` 自动导入所有处理 URL 的函数。这样，代码一分离，逻辑就显得清楚了。最好是这样：

```

url2-koa/
|
+- .vscode/
| |
| +- launch.json <-- VSCode 配置文件
|
+- controllers/
| |
| +- login.js <-- 处理login相关URL
| |
| +- users.js <-- 处理用户管理相关URL
|
+- app.js <-- 使用koa的js
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包

```

于是我们把 `url1-koa` 复制一份，重命名为 `url2-koa`，准备重构这个项目。

我们先在 `controllers` 目录下编写 `index.js`：

```

var fn_index = async (ctx, next) => {
  ctx.response.body = `<h1>Index</h1>
    <form action="/signin" method="post">
      <p>Name: <input name="name" value="koa"></p>
      <p>Password: <input name="password" type="password"></p>
      <p><input type="submit" value="Submit"></p>
    </form>`;
};

var fn_signin = async (ctx, next) => {
  var
    name = ctx.request.body.name || '',
    password = ctx.request.body.password || '';
  console.log(`signin with name: ${name}, password: ${password}`);
  if (name === 'koa' && password === '12345') {
    ctx.response.body = `<h1>Welcome, ${name}!</h1>`;
  } else {
    ctx.response.body = `<h1>Login failed!</h1>
    <p><a href="/">Try again</a></p>`;
  }
};

module.exports = {
  'GET /': fn_index,
  'POST /signin': fn_signin
};

```

这个 `index.js` 通过 `module.exports` 把两个URL处理函数暴露出来。

类似的，`hello.js` 把一个URL处理函数暴露出来：

```

var fn_hello = async (ctx, next) => {
  var name = ctx.params.name;
  ctx.response.body = `<h1>Hello, ${name}!</h1>`;
};

module.exports = {
  'GET /hello/:name': fn_hello
};

```

现在，我们修改 `app.js`，让它自动扫描 `controllers` 目录，找到所有 `js` 文件，导入，然后注册每个URL：

```

// 先导入fs模块，然后用readdirSync列出文件
// 这里可以用sync是因为启动时只运行一次，不存在性能问题：
var files = fs.readdirSync(__dirname + '/controllers');

// 过滤出.js文件：
var js_files = files.filter((f)=>{
  return f.endsWith('.js');
});

// 处理每个js文件：
for (var f of js_files) {
  console.log(`process controller: ${f}...`);
  // 导入js文件：
  let mapping = require(__dirname + '/controllers/' + f);
  for (var url in mapping) {
    if (url.startsWith('GET ')) {
      // 如果url类似"GET xxx":
      var path = url.substring(4);
      router.get(path, mapping[url]);
      console.log(`register URL mapping: GET ${path}`);
    } else if (url.startsWith('POST ')) {
      // 如果url类似"POST xxx":
      var path = url.substring(5);
      router.post(path, mapping[url]);
      console.log(`register URL mapping: POST ${path}`);
    } else {
      // 无效的URL:
      console.log(`invalid URL: ${url}`);
    }
  }
}
}

```

如果上面的大段代码看起来还是有点费劲，那就把它拆成更小单元的函数：

```

function addMapping(router, mapping) {
  for (var url in mapping) {
    if (url.startsWith('GET ')) {
      var path = url.substring(4);
      router.get(path, mapping[url]);
      console.log(`register URL mapping: GET ${path}`);
    } else if (url.startsWith('POST ')) {
      var path = url.substring(5);
      router.post(path, mapping[url]);
      console.log(`register URL mapping: POST ${path}`);
    } else {
      console.log(`invalid URL: ${url}`);
    }
  }
}

function addControllers(router) {
  var files = fs.readdirSync(__dirname + '/controllers');
  var js_files = files.filter((f) => {
    return f.endsWith('.js');
  });

  for (var f of js_files) {
    console.log(`process controller: ${f}...`);
    let mapping = require(__dirname + '/controllers/' + f);
    addMapping(router, mapping);
  }
}

addControllers(router);

```


确保每个函数功能非常简单，一眼能看明白，是代码可维护的关键。

Controller Middleware

最后，我们把扫描 `controllers` 目录和创建 `router` 的代码从 `app.js` 中提取出来，作为一个简单的 `middleware` 使用，命名为 `controller.js`：

```
const fs = require('fs');

function addMapping(router, mapping) {
  ...
}

function addControllers(router, dir) {
  ...
}

module.exports = function (dir) {
  let
    controllers_dir = dir || 'controllers', // 如果不传参数，扫描目录默认为 'controllers'
    router = require('koa-router')();
  addControllers(router, controllers_dir);
  return router.routes();
};
```

这样一来，我们在 `app.js` 的代码又简化了：

```
...

// 导入controller middleware:
const controller = require('./controller');

...

// 使用middleware:
app.use(controller());

...
```

经过重新整理后的工程 `url2-koa` 目前具备非常好的模块化，所有处理 `URL` 的函数按功能组存放在 `controllers` 目录，今后我们也只需要不断往这个目录下加东西就可以了，`app.js` 保持不变。

参考源码

[url-koa](#)

[url2-koa](#)

使用Nunjucks

Nunjucks

Nunjucks是什么东东？其实它是一个模板引擎。

那什么是模板引擎？

模板引擎就是基于模板配合数据构造出字符串输出的一个组件。比如下面的函数就是一个模板引擎：

```
function examResult (data) {  
    return `${data.name}同学一年级期末考试语文${data.chinese}分，数学${data.math}分，位于年级第${data.ranking}名。`  
}
```

如果我们输入数据如下：

```
examResult({  
    name: '小明',  
    chinese: 78,  
    math: 87,  
    ranking: 999  
});
```

该模板引擎把模板字符串里面对应的变量替换以后，就可以得到以下输出：

小明同学一年级期末考试语文78分，数学87分，位于年级第999名。

模板引擎最常见的输出就是输出网页，也就是**HTML**文本。当然，也可以输出任意格式的文本，比如**Text**，**XML**，**Markdown**等等。

有同学要问了：既然**JavaScript**的模板字符串可以实现模板功能，那为什么我们还需要另外的模板引擎？

因为**JavaScript**的模板字符串必须写在**JavaScript**代码中，要想写出新浪首页这样复杂的页面，是非常困难的。

输出**HTML**有几个特别重要的问题需要考虑：

转义

对特殊字符要转义，避免受到**XSS**攻击。比如，如果变量`name`的值不是**小明**，而是**小明<script>...</script>**，模板引擎输出的**HTML**到了浏览器，就会自动执行恶意**JavaScript**代码。

格式化

对不同类型的变量要格式化，比如，货币需要变成**12,345.00**这样的格式，日期需要变成**2016-01-01**这样的格式。

简单逻辑

模板还需要能执行一些简单逻辑，比如，要按条件输出内容，需要**if**实现如下输出：

```
{{ name }}同学，  
{% if score >= 90 %}  
    成绩优秀，应该奖励  
{% elif score >=60 %}  
    成绩良好，继续努力  
{% else %}  
    不及格，建议回家打屁股  
{% endif %}
```

所以，我们需要一个功能强大的模板引擎，来完成页面输出的功能。

Nunjucks

我们选择Nunjucks作为模板引擎。Nunjucks是Mozilla开发的一个纯JavaScript编写的模板引擎，既可以用在Node环境下，又可以运行在浏览器端。但是，主要还是运行在Node环境下，因为浏览器端有更好的模板解决方案，例如MVVM框架。

如果你使用过Python的模板引擎jinja2，那么使用Nunjucks就非常简单，两者的语法几乎是一模一样的，因为Nunjucks就是用JavaScript重新实现了jinja2。

从上面的例子我们可以看到，虽然模板引擎内部可能非常复杂，但是使用一个模板引擎是非常简单的，因为本质上我们只需要构造这样一个函数：

```
function render(view, model) {  
  // TODO:...\br/>}
```

其中，view是模板的名称（又称为视图），因为可能存在多个模板，需要选择其中一个。model就是数据，在JavaScript中，它就是一个简单的Object。render函数返回一个字符串，就是模板的输出。

下面我们来使用Nunjucks这个模板引擎来编写几个HTML模板，并且用实际数据来渲染模板并获得最终的HTML输出。

我们创建一个use-nunjucks的VS Code工程结构如下：

```
use-nunjucks/  
|  
+- .vscode/  
| |  
| +- launch.json <-- VSCode 配置文件  
|  
+- views/  
| |  
| +- hello.html <-- HTML模板文件  
|  
+- app.js <-- 入口js  
|  
+- package.json <-- 项目描述文件  
|  
+- node_modules/ <-- npm安装的所有依赖包
```

其中，模板文件存放在views目录中。

我们先在package.json中添加nunjucks的依赖：

```
"nunjucks": "2.4.2"
```

注意，模板引擎是可以独立使用的，并不需要依赖koa。用npm install安装所有依赖包。

紧接着，我们要编写使用Nunjucks的函数render。怎么写？方法是查看Nunjucks的官方文档，仔细阅读后，在app.js中编写代码如下：

```

const nunjucks = require('nunjucks');

function createEnv(path, opts) {
  var
    autoescape = opts.autoescape && true,
    noCache = opts.noCache || false,
    watch = opts.watch || false,
    throwOnUndefined = opts.throwOnUndefined || false,
    env = new nunjucks.Environment(
      new nunjucks.FileSystemLoader('views', {
        noCache: noCache,
        watch: watch,
      }), {
        autoescape: autoescape,
        throwOnUndefined: throwOnUndefined
      });
  if (opts.filters) {
    for (var f in opts.filters) {
      env.addFilter(f, opts.filters[f]);
    }
  }
  return env;
}

var env = createEnv('views', {
  watch: true,
  filters: {
    hex: function (n) {
      return '0x' + n.toString(16);
    }
  }
});

```

变量 `env` 就表示Nunjucks模板引擎对象，它有一个 `render(view, model)` 方法，正好传入 `view` 和 `model` 两个参数，并返回字符串。

创建 `env` 需要的参数可以查看文档获知。我们用 `autoescape = opts.autoescape && true` 这样的代码给每个参数加上默认值，最后使用 `new nunjucks.FileSystemLoader('views')` 创建一个文件系统加载器，从 `views` 目录读取模板。

我们编写一个 `hello.html` 模板文件，放到 `views` 目录下，内容如下：

```
<h1>Hello {{ name }}</h1>
```

然后，我们就可以用下面的代码来渲染这个模板：

```
var s = env.render('hello.html', { name: '小明' });
console.log(s);
```

获得输出如下：

```
<h1>Hello 小明</h1>
```

咋一看，这和使用JavaScript模板字符串没啥区别嘛。不过，试试：

```
var s = env.render('hello.html', { name: '<script>alert("小明")</script>' });
console.log(s);
```

获得输出如下：

```
<h1>Hello &lt;script&gt;alert("小明")&lt;/script&gt;</h1>
```

这样就避免了输出恶意脚本。

此外，可以使用Nunjucks提供的功能强大的tag，编写条件判断、循环等功能，例如：

```
<!-- 循环输出名字 -->
<body>
  <h3>Fruits List</h3>
  {% for f in fruits %}
  <p>{{ f }}</p>
  {% endfor %}
</body>
```

Nunjucks模板引擎最强大的功能在于模板的继承。仔细观察各种网站可以发现，网站的结构实际上是类似的，头部、尾部都是固定格式，只有中间页面部分内容不同。如果每个模板都重复头尾，一旦要修改头部或尾部，那就需要改动所有模板。

更好的方式是使用继承。先定义一个基本的网页框架base.html：

```
<html><body>
{% block header %} <h3>Unnamed</h3> {% endblock %}
{% block body %} <div>No body</div> {% endblock %}
{% block footer %} <div>copyright</div> {% endblock %}
</body>
```

base.html定义了三个可编辑的块，分别命名为header、body和footer。子模板可以有选择地对块进行重新定义：

```
{% extends 'base.html' %}

{% block header %}<h1>{{ header }}</h1>{% endblock %}

{% block body %}<p>{{ body }}</p>{% endblock %}
```

然后，我们对子模板进行渲染：

```
console.log(env.render('extend.html', {
  header: 'Hello',
  body: 'bla bla bla...'
}));
```

输出HTML如下：

```
<html><body>
<h1>Hello</h1>
<p>bla bla bla...</p>
<div>copyright</div> <-- footer没有重定义，所以仍使用父模板的内容
</body>
```

性能

最后我们要考虑一下Nunjucks的性能。

对于模板渲染本身来说，速度是非常非常快的，因为就是拼字符串嘛，纯CPU操作。

性能问题主要出现在从文件读取模板内容这一步。这是一个IO操作，在Node.js环境中，我们知道，单线程的JavaScript最不能忍受的就是同步IO，但Nunjucks默认就使用同步IO读取模板文件。

好消息是Nunjucks会缓存已读取的文件内容，也就是说，模板文件最多读取一次，就会放在内存中，后面的请求是不会再次读取文件的，只要我们指定了noCache: false这个参数。

在开发环境下，可以关闭cache，这样每次重新加载模板，便于实时修改模板。在生产环境下，一定要打开cache，这样就不会有性能问题。

Nunjucks也提供了异步读取的方式，但是这样写起来很麻烦，有简单的写法我们就不会考虑复杂的写法。保持代码简单是可维护性的关键。

参考源码

[use-nunjucks](#)

使用MVC

MVC

我们已经可以用koa处理不同的URL，还可以用Nunjucks渲染模板。现在，是时候把这两者结合起来了！

当用户通过浏览器请求一个URL时，koa将调用某个异步函数处理该URL。在这个异步函数内部，我们用一行代码：

```
ctx.render('home.html', { name: 'Michael' });
```

通过Nunjucks把数据用指定的模板渲染成HTML，然后输出给浏览器，用户就可以看到渲染后的页面了：



这就是传说中的MVC：Model-View-Controller，中文名“模型-视图-控制器”。

异步函数是C：Controller，Controller负责业务逻辑，比如检查用户名是否存在，取出用户信息等等；

包含变量`{{ name }}`的模板就是V：View，View负责显示逻辑，通过简单地替换一些变量，View最终输出的就是用户看到的HTML。

MVC中的Model在哪？Model是用来传给View的，这样View在替换变量的时候，就可以从Model中取出相应的数据。

上面的例子中，Model就是一个JavaScript对象：

```
{ name: 'Michael' }
```

下面，我们根据原来的`url2-koa`创建工程`view-koa`，把koa2、Nunjucks整合起来，然后，把原来直接输出字符串的方式，改为`ctx.render(view, model)`的方式。

工程`view-koa`结构如下：

```
view-koa/
|
+- .vscode/
| |
| +- launch.json <-- VSCode 配置文件
|
+- controllers/ <-- Controller
|
+- views/ <-- html模板文件
|
+- static/ <-- 静态资源文件
|
+- controller.js <-- 扫描注册Controller
|
+- app.js <-- 使用koa的js
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

在 `package.json` 中，我们将要用到的依赖包有：

```
"koa": "2.0.0",
"koa-bodyparser": "3.2.0",
"koa-router": "7.0.0",
"nunjucks": "2.4.2",
"mime": "1.3.4",
"mz": "2.4.0"
```

先用 `npm install` 安装依赖包。

然后，我们准备编写以下两个Controller：

处理首页 GET /

我们定义一个 `async` 函数处理首页 URL `/`：

```
async (ctx, next) => {
  ctx.render('index.html', {
    title: 'Welcome'
  });
}
```

注意到 `koa` 并没有在 `ctx` 对象上提供 `render` 方法，这里我们假设应该这么使用，这样，我们在编写Controller的时候，最后一步调用 `ctx.render(view, model)` 就完成了页面输出。

处理登录请求 POST /signin

我们再定义一个 `async` 函数处理登录请求 `/signin`：


```

async (ctx, next) => {
  var
    email = ctx.request.body.email || '',
    password = ctx.request.body.password || '';
  if (email === 'admin@example.com' && password === '123456') {
    // 登录成功:
    ctx.render('signin-ok.html', {
      title: 'Sign In OK',
      name: 'Mr Node'
    });
  } else {
    // 登录失败:
    ctx.render('signin-failed.html', {
      title: 'Sign In Failed'
    });
  }
}

```

由于登录请求是一个POST，我们就用 `ctx.request.body.<name>` 拿到POST请求的数据，并给一个默认值。

登录成功时我们用 `signin-ok.html` 渲染，登录失败时我们用 `signin-failed.html` 渲染，所以，我们一共需要以下3个View:

- index.html
- signin-ok.html
- signin-failed.html

编写View

在编写View的时候，我们实际上是在编写HTML页。为了让页面看起来美观大方，使用一个现成的CSS框架是非常有必要的。我们用Bootstrap这个CSS框架。从首页下载zip包后解压，我们把所有静态资源文件放到 `/static` 目录下：

```

view-koa/
|
+- static/
  |
  +- css/ <- 存放bootstrap.css等
  |
  +- fonts/ <- 存放字体文件
  |
  +- js/ <- 存放bootstrap.js等

```

这样我们在编写HTML的时候，可以直接用Bootstrap的CSS，像这样：

```
<link rel="stylesheet" href="/static/css/bootstrap.css">
```

现在，在使用MVC之前，第一个问题来了，如何处理静态文件？

我们把所有静态资源文件全部放入 `/static` 目录，目的就是能统一处理静态文件。在koa中，我们需要编写一个middleware，处理以 `/static/` 开头的URL。

编写middleware

我们来编写一个处理静态文件的middleware。编写middleware实际一点也不复杂。我们先创建一个 `static-files.js` 的文件，编写一个能处理静态文件的middleware:

```

const path = require('path');
const mime = require('mime');
const fs = require('mz/fs');

// url: 类似 '/static/'
// dir: 类似 __dirname + '/static'
function staticFiles(url, dir) {
  return async (ctx, next) => {
    let rpath = ctx.request.path;
    // 判断是否以指定的url开头:
    if (rpath.startsWith(url)) {
      // 获取文件完整路径:
      let fp = path.join(dir, rpath.substring(url.length));
      // 判断文件是否存在:
      if (await fs.exists(fp)) {
        // 查找文件的mime:
        ctx.response.type = mime.lookup(rpath);
        // 读取文件内容并赋值给response.body:
        ctx.response.body = await fs.readFile(fp);
      } else {
        // 文件不存在:
        ctx.response.status = 404;
      }
    } else {
      // 不是指定前缀的URL, 继续处理下一个middleware:
      await next();
    }
  };
}

module.exports = staticFiles;

```

`staticFiles` 是一个普通函数，它接收两个参数：URL前缀和一个目录，然后返回一个`async`函数。这个`async`函数会判断当前的URL是否以指定前缀开头，如果是，就把URL的路径视为文件，并发送文件内容。如果不是，这个`async`函数就不做任何事情，而是简单地调用 `await next()` 让下一个middleware去处理请求。

我们使用了一个 `mz` 的包，并通过 `require('mz/fs');` 导入。`mz` 提供的API和Node.js的 `fs` 模块完全相同，但 `fs` 模块使用回调，而 `mz` 封装了 `fs` 对应的函数，并改为`Promise`。这样，我们就可以非常简单的用 `await` 调用 `mz` 的函数，而不需要任何回调。

所有的第三方包都可以通过npm官网搜索并查看其文档：

<https://www.npmjs.com/>

最后，这个middleware使用起来也很简单，在 `app.js` 里加一行代码：

```

let staticFiles = require('./static-files');
app.use(staticFiles('/static/', __dirname + '/static'));

```

注意： 也可以去npm搜索能用于koa2的处理静态文件的包并直接使用。

集成Nunjucks

集成Nunjucks实际上也是编写一个middleware，这个middleware的作用是给 `ctx` 对象绑定一个 `render(view, model)` 的方法，这样，后面的Controller就可以调用这个方法渲染模板了。

我们创建一个 `templating.js` 来实现这个middleware：

```

const nunjucks = require('nunjucks');

function createEnv(path, opts) {
  var
    autoescape = opts.autoescape === undefined ? true : opts.autoescape,
    noCache = opts.noCache || false,
    watch = opts.watch || false,
    throwOnUndefined = opts.throwOnUndefined || false,
    env = new nunjucks.Environment(
      new nunjucks.FileSystemLoader(path || 'views', {
        noCache: noCache,
        watch: watch,
      }), {
        autoescape: autoescape,
        throwOnUndefined: throwOnUndefined
      });
  if (opts.filters) {
    for (var f in opts.filters) {
      env.addFilter(f, opts.filters[f]);
    }
  }
  return env;
}

function templating(path, opts) {
  // 创建Nunjucks的env对象：
  var env = createEnv(path, opts);
  return async (ctx, next) => {
    // 给ctx绑定render函数：
    ctx.render = function (view, model) {
      // 把render后的内容赋值给response.body：
      ctx.response.body = env.render(view, Object.assign({}, ctx.state || {}, model || {}));
      // 设置Content-Type：
      ctx.response.type = 'text/html';
    };
    // 继续处理请求：
    await next();
  };
}

module.exports = templating;

```

注意到 `createEnv()` 函数和前面使用Nunjucks时编写的函数是一模一样的。我们主要关心 `templating()` 函数，它会返回一个middleware，在这个middleware中，我们只给 `ctx` “安装”了一个 `render()` 函数，其他什么事情也没干，就继续调用下一个middleware。

使用的时候，我们在 `app.js` 添加如下代码：

```

const isProduction = process.env.NODE_ENV === 'production';

app.use(templating('view', {
  noCache: !isProduction,
  watch: !isProduction
}));

```

这里我们定义了一个常量 `isProduction`，它判断当前环境是否是production环境。如果是，就使用缓存，如果不是，就关闭缓存。在开发环境下，关闭缓存后，我们修改View，可以直接刷新浏览器看到效果，否则，每次修改都必须重启Node程序，会极大地降低开发效率。

Node.js在全局变量 `process` 中定义了一个环境变量 `env.NODE_ENV`，为什么要使用该环境变量？因为我们在开发的时候，环境变量应该设置为 `'development'`，而部署到服务器时，环境变量应该设置为 `'production'`。在编写代码的时候，要根据当前环境作不同的判断。

注意：生产环境上必须配置环境变量 `NODE_ENV = 'production'`，而开发环境不需要配置，实际上 `NODE_ENV` 可能是 `undefined`，所以判断的时候，不要用 `NODE_ENV === 'development'`。

类似的，我们在使用上面编写的处理静态文件的middleware时，也可以根据环境变量判断：

```
if (! isProduction) {
  let staticFiles = require('./static-files');
  app.use(staticFiles('/static/', __dirname + '/static'));
}
```

这是因为在生产环境下，静态文件是由部署在最前面的反向代理服务器（如Nginx）处理的，Node程序不需要处理静态文件。而在开发环境下，我们希望koa能顺带处理静态文件，否则，就必须手动配置一个反向代理服务器，这样会导致开发环境非常复杂。

编写View

在编写View的时候，非常有必要先编写一个base.html作为骨架，其他模板都继承自base.html，这样，才能大大减少重复工作。

编写HTML不在本教程的讨论范围之内。这里我们参考Bootstrap的官网简单编写了base.html。

运行

一切顺利的话，这个view-koa工程应该可以顺利运行。运行前，我们再检查一下app.js里的middleware的顺序：

第一个middleware是记录URL以及页面执行时间：

```
app.use(async (ctx, next) => {
  console.log(`Process ${ctx.request.method} ${ctx.request.url}...`);
  var
    start = new Date().getTime(),
    execTime;
  await next();
  execTime = new Date().getTime() - start;
  ctx.response.set('X-Response-Time', `${execTime}ms`);
});
```

第二个middleware处理静态文件：

```
if (! isProduction) {
  let staticFiles = require('./static-files');
  app.use(staticFiles('/static/', __dirname + '/static'));
}
```

第三个middleware解析POST请求：

```
app.use(bodyParser());
```

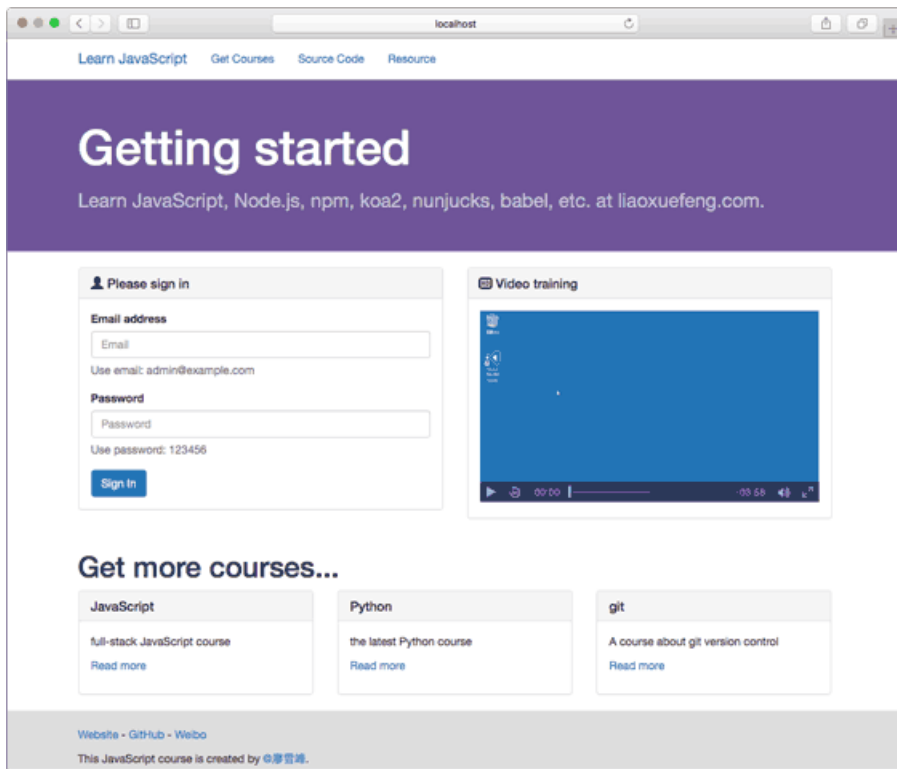
第四个middleware负责给ctx加上render()来使用Nunjucks：

```
app.use(templating('view', {
  noCache: !isProduction,
  watch: !isProduction
})));
```

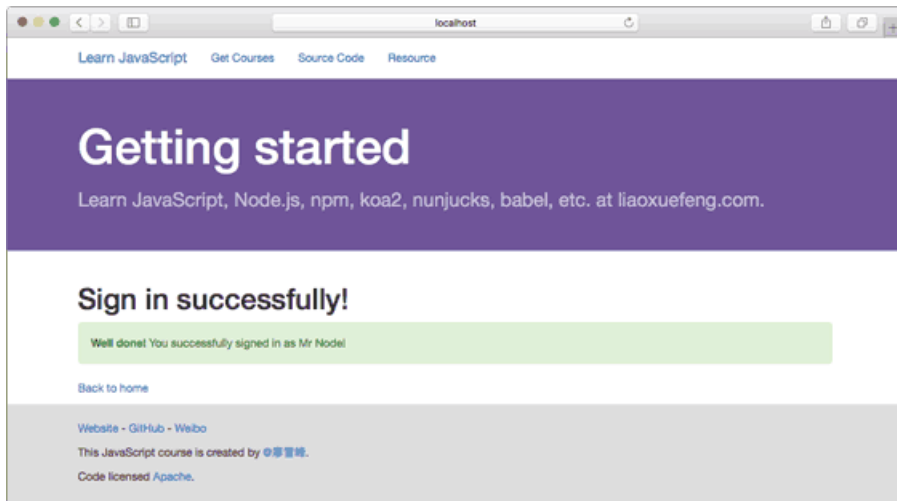
最后一个middleware处理URL路由：

```
app.use(controller());
```

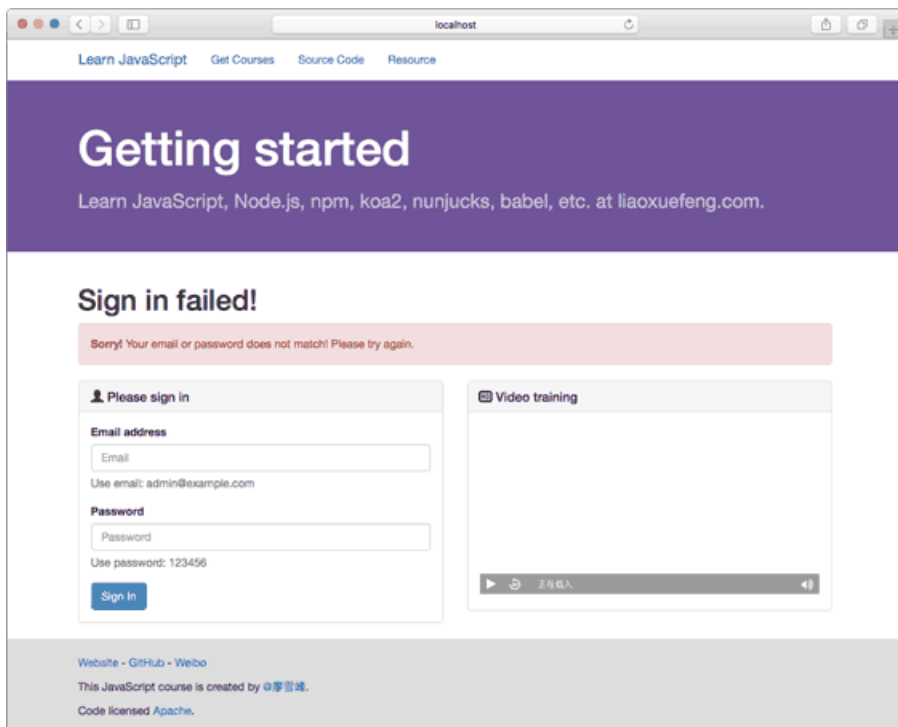
现在，在VS Code中运行代码，不出意外的话，在浏览器输入localhost:3000/，可以看到首页内容：



直接在首页登录，如果输入正确的Email和Password，进入登录成功的页面：



如果输入的Email和Password不正确，进入登录失败的页面：



怎么判断正确的Email和Password？目前我们在 `signin.js` 中是这么判断的：

```
if (email === 'admin@example.com' && password === '123456') {  
  ...  
}
```

当然，真实的网站会根据用户输入的Email和Password去数据库查询并判断登录是否成功，不过这需要涉及到Node.js环境如何操作数据库，我们后面再讨论。

扩展

注意到 `ctx.render` 内部渲染模板时，Model对象并不是传入的model变量，而是：

```
Object.assign({}, ctx.state || {}, model || {})
```

这个小技巧是为了扩展。

首先，`model || {}` 确保了即使传入 `undefined`，model也会变为默认值 `{}`。`Object.assign()` 会把除第一个参数外的其他参数的所有属性复制到第一个参数中。第二个参数是 `ctx.state || {}`，这个目的是为了能把一些公共的变量放入 `ctx.state` 并传给View。

例如，某个middleware负责检查用户权限，它可以把当前用户放入 `ctx.state` 中：

```
app.use(async (ctx, next) => {  
  var user = tryGetUserFromCookie(ctx.request);  
  if (user) {  
    ctx.state.user = user;  
    await next();  
  } else {  
    ctx.response.status = 403;  
  }  
});
```

这样就没有必要在每个Controller的async函数中都把user变量放入model中。

参考源码

[view-koa](#)

mysql

访问数据库

程序运行的时候，数据都是在内存中的。当程序终止的时候，通常都需要将数据保存到磁盘上，无论是保存到本地磁盘，还是通过网络保存到服务器上，最终都会将数据写入磁盘文件。

而如何定义数据的存储格式就是一个大问题。如果我们自己来定义存储格式，比如保存一个班级所有学生的成绩单：

名字	成绩
Michael	99
Bob	85
Bart	59
Lisa	87

你可以用一个文本文件保存，一行保存一个学生，用`,`隔开：

```
Michael,99
Bob,85
Bart,59
Lisa,87
```

你还可以用JSON格式保存，也是文本文件：

```
[
  {"name":"Michael","score":99},
  {"name":"Bob","score":85},
  {"name":"Bart","score":59},
  {"name":"Lisa","score":87}
]
```

你还可以定义各种保存格式，但是问题来了：

存储和读取需要自己实现，JSON还是标准，自己定义的格式就各式各样了；

不能做快速查询，只有把数据全部读到内存中才能自己遍历，但有时候数据的大小远远超过了内存（比如蓝光电影，40GB的数据），根本无法全部读入内存。

为了便于程序保存和读取数据，而且，能直接通过条件快速查询到指定的数据，就出现了数据库（Database）这种专门用于集中存储和查询的软件。

数据库软件诞生的历史非常久远，早在1950年数据库就诞生了。经历了网状数据库，层次数据库，我们现在广泛使用的关系数据库是20世纪70年代基于关系模型的基础上诞生的。

关系模型有一套复杂的数学理论，但是从概念上是十分容易理解的。举个学校的例子：

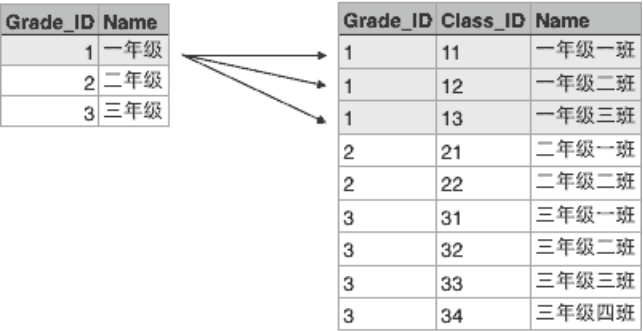
假设某个XX省YY市ZZ县第一实验小学有3个年级，要表示出这3个年级，可以在Excel中用一个表格画出来：

Grade_ID	Name
1	一年级
2	二年级
3	三年级

每个年级又有若干个班级，要把所有班级表示出来，可以在Excel中再画一个表格：

Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

这两个表格有个映射关系，就是根据Grade_ID可以在班级表中查找到对应的所有班级：



也就是Grade表的每一行对应Class表的多行，在关系数据库中，这种基于表（Table）的一对多的关系就是关系数据库的基础。

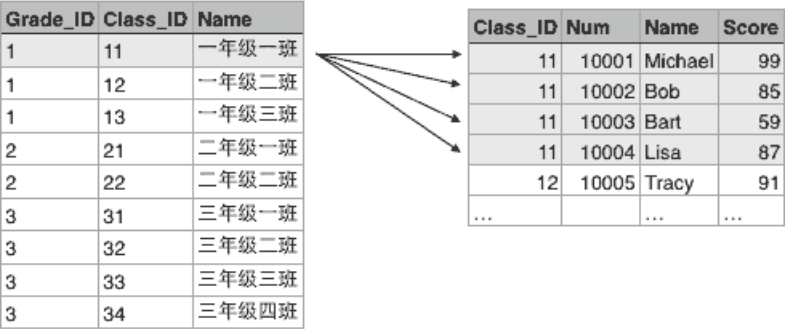
根据某个年级的ID就可以查找所有班级的行，这种查询语句在关系数据库中称为SQL语句，可以写成：

```
SELECT * FROM classes WHERE grade_id = '1';
```

结果也是一个表：

```
-----+-----+-----
grade_id | class_id | name
-----+-----+-----
1        | 11       | 一年级一班
-----+-----+-----
1        | 12       | 一年级二班
-----+-----+-----
1        | 13       | 一年级三班
-----+-----+-----
```

类似的，Class表的一行记录又可以关联到Student表的多行记录：



由于本教程不涉及到关系数据库的详细内容，如果你想从零学习关系数据库和基本的SQL语句，请自行搜索相关课程。

NoSQL

你也许还听说过**NoSQL**数据库，很多**NoSQL**宣传其速度和规模远远超过关系数据库，所以很多同学觉得有了**NoSQL**是否就不需要**SQL**了呢？千万不要被他们忽悠了，连**SQL**都不明白怎么可能搞明白**NoSQL**呢？

数据库类别

既然我们要使用关系数据库，就必须选择一个关系数据库。目前广泛使用的关系数据库也就这么几种：

付费的商用数据库：

- **Oracle**，典型的高富帅；
- **SQL Server**，微软自家产品，**Windows**定制专款；
- **DB2**，**IBM**的产品，听起来挺高端；
- **Sybase**，曾经跟微软是好基友，后来关系破裂，现在家境惨淡。

这些数据库都是不开源而且付费的，最大的好处是花了钱出了问题可以找厂家解决，不过在**Web**的世界里，常常需要部署成千上万的数据库服务器，当然不能把大把大把的银子扔给厂家，所以，无论是**Google**、**Facebook**，还是国内的**BAT**，无一例外都选择了免费的开源数据库：

- **MySQL**，大家都在用，一般错不了；
- **PostgreSQL**，学术气息有点重，其实挺不错，但知名度没有**MySQL**高；
- **sqlite**，嵌入式数据库，适合桌面和移动应用。

作为一个**JavaScript**全栈工程师，选择哪个免费数据库呢？当然是**MySQL**。因为**MySQL**普及率最高，出了错，可以很容易找到解决方法。而且，围绕**MySQL**有一大堆监控和运维的工具，安装和使用很方便。

安装MySQL

为了能继续后面的学习，你需要从**MySQL**官方网站下载并安装**MySQL Community Server 5.6**，这个版本是免费的，其他高级版本是要收钱的（请放心，收钱的功能我们用不上）。**MySQL**是跨平台的，选择对应的平台下载安装文件，安装即可。

安装时，**MySQL**会提示输入 `root` 用户的口令，请务必记清楚。如果怕记不住，就把口令设置为 `password`。

在**Windows**上，安装时请选择 `UTF-8` 编码，以便正确地处理中文。

在**Mac**或**Linux**上，需要编辑**MySQL**的配置文件，把数据库默认的编码全部改为**UTF-8**。**MySQL**的配置文件默认存放在 `/etc/my.cnf` 或者 `/etc/mysql/my.cnf`：

```
[client]
default-character-set = utf8

[mysqld]
default-storage-engine = INNODB
character-set-server = utf8
collation-server = utf8_general_ci
```

重启**MySQL**后，可以通过**MySQL**的客户端命令行检查编码：

```
$ mysql -u root -p
Enter password:
Welcome to the MySQL monitor...
...

mysql> show variables like '%char%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| character_set_client | utf8 |
| character_set_connection | utf8 |
| character_set_database | utf8 |
| character_set_filesystem | binary |
| character_set_results | utf8 |
| character_set_server | utf8 |
| character_set_system | utf8 |
| character_sets_dir | /usr/local/mysql-5.1.65-osx10.6-x86_64/share/charsets/ |
+-----+-----+
8 rows in set (0.00 sec)
```

看到 `utf8` 字样就表示编码设置正确。

注：如果MySQL的版本≥5.5.3，可以把编码设置为 `utf8mb4`，`utf8mb4` 和 `utf8` 完全兼容，但它支持最新的Unicode标准，可以显示emoji字符。

使用Sequelize

访问MySQL

当我们安装好MySQL后，Node.js程序如何访问MySQL数据库呢？

访问MySQL数据库只有一种方法，就是通过网络发送SQL命令，然后，MySQL服务器执行后返回结果。

我们可以在命令行窗口输入 `mysql -u root -p`，然后输入root口令后，就连接到了MySQL服务器。因为没有指定 `--host` 参数，所以我们连接到的 是 `localhost`，也就是本机的MySQL服务器。

在命令行窗口下，我们可以输入命令，操作MySQL服务器：

```
mysql> show databases;
+-----+
| Database          |
+-----+
| information_schema |
| mysql              |
| performance_schema |
| test               |
+-----+
4 rows in set (0.05 sec)
```

输入 `exit` 退出MySQL命令行模式。

对于Node.js程序，访问MySQL也是通过网络发送SQL命令给MySQL服务器。这个访问MySQL服务器的软件包通常称为MySQL驱动程序。不同的编程语言需要实现自己的驱动，MySQL官方提供了Java、.Net、Python、Node.js、C++和C的驱动程序，官方的Node.js驱动目前仅支持5.7以上版本，而我们上面使用的命令行程序实际上用的就是C驱动。

目前使用最广泛的MySQL Node.js驱动程序是开源的 `mysql`，可以直接使用npm安装。

ORM

如果直接使用 `mysql` 包提供的接口，我们编写的代码就比较底层，例如，查询代码：

```
connection.query('SELECT * FROM users WHERE id = ?', ['123'], function(err, rows) {
  if (err) {
    // error
  } else {
    for (let row in rows) {
      processRow(row);
    }
  }
});
```

考虑到数据库表是一个二维表，包含多行多列，例如一个 `pets` 的表：

```
mysql> select * from pets;
+----+-----+-----+
| id | name  | birth |
+----+-----+-----+
|  1 | Gaffey | 2007-07-07 |
|  2 | Odie   | 2008-08-08 |
+----+-----+-----+
2 rows in set (0.00 sec)
```

每一行可以用一个JavaScript对象表示，例如第一行：

```
{
  "id": 1,
  "name": "Gaffey",
  "birth": "2007-07-07"
}
```

这就是传说中的ORM技术：**Object-Relational Mapping**，把关系数据库的表结构映射到对象上。是不是很简单？

但是由谁来做这个转换呢？所以ORM框架应运而生。

我们选择**Node**的ORM框架**Sequelize**来操作数据库。这样，我们读写的都是**JavaScript**对象，**Sequelize**帮我们把对象变成数据库中的行。

用**Sequelize**查询 `pets` 表，代码像这样：

```
Pet.findAll()
  .then(function (pets) {
    for (let pet in pets) {
      console.log(`${pet.id}: ${pet.name}`);
    }
  }).catch(function (err) {
    // error
  });
```

因为**Sequelize**返回的对象是**Promise**，所以我们可以用 `then()` 和 `catch()` 分别异步响应成功和失败。

但是用 `then()` 和 `catch()` 仍然比较麻烦。有没有更简单的方法呢？

可以用**ES7**的**await**来调用任何一个**Promise**对象，这样我们写出来的代码就变成了：

```
var pets = await Pet.findAll();
```

真的就是这么简单！

await只有一个限制，就是必须在**async**函数中调用。上面的代码直接运行还差一点，我们可以改成：

```
(async () => {
  var pets = await Pet.findAll();
}) ();
```

考虑到**koa**的处理函数都是**async**函数，所以我们实际上将来在**koa**的**async**函数中直接写**await**访问数据库就可以了！

这也是为什么我们选择**Sequelize**的原因：只要**API**返回**Promise**，就可以用**await**调用，写代码就非常简单！

实战

在使用**Sequelize**操作数据库之前，我们先在**MySQL**中创建一个表来测试。我们可以在 `test` 数据库中创建一个 `pets` 表。`test` 数据库是**MySQL**安装后自动创建的用于测试的数据库。在**MySQL**命令行执行下列命令：

```
grant all privileges on test.* to 'www'@'%' identified by 'www';

use test;

create table pets (
  id varchar(50) not null,
  name varchar(100) not null,
  gender bool not null,
  birth varchar(10) not null,
  createdAt bigint not null,
  updatedAt bigint not null,
  version bigint not null,
  primary key (id)
) engine=innodb;
```

第一条`grant`命令是创建MySQL的用户名和口令，均为`www`，并赋予操作`test`数据库的所有权限。

第二条`use`命令把当前数据库切换为`test`。

第三条命令创建了`pets`表。

然后，我们根据前面的工程结构创建`hello-sequelize`工程，结构如下：

```
hello-sequelize/
|
+- .vscode/
| |
| +- launch.json <-- VSCode 配置文件
|
+- init.txt <-- 初始化SQL命令
|
+- config.js <-- MySQL配置文件
|
+- app.js <-- 使用koa的js
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

然后，添加如下依赖包：

```
"sequelize": "3.24.1",
"mysql": "2.11.1"
```

注意`mysql`是驱动，我们不直接使用，但是`sequelize`会用。

用`npm install`安装。

`config.js`实际上是一个简单的配置文件：

```
var config = {
  database: 'test', // 使用哪个数据库
  username: 'www', // 用户名
  password: 'www', // 口令
  host: 'localhost', // 主机名
  port: 3306 // 端口号，MySQL默认3306
};

module.exports = config;
```

下面，我们就可以在`app.js`中操作数据库了。使用Sequelize操作MySQL需要先做两件准备工作：

第一步，创建一个`sequelize`对象实例：

```
const Sequelize = require('sequelize');
const config = require('./config');

var sequelize = new Sequelize(config.database, config.username, config.password, {
  host: config.host,
  dialect: 'mysql',
  pool: {
    max: 5,
    min: 0,
    idle: 30000
  }
});
```

第二步，定义模型**Pet**，告诉**Sequelize**如何映射数据库表：

```
var Pet = sequelize.define('pet', {
  id: {
    type: Sequelize.STRING(50),
    primaryKey: true
  },
  name: Sequelize.STRING(100),
  gender: Sequelize.BOOLEAN,
  birth: Sequelize.STRING(10),
  createdAt: Sequelize.BIGINT,
  updatedAt: Sequelize.BIGINT,
  version: Sequelize.BIGINT
}, {
  timestamps: false
});
```

用 `sequelize.define()` 定义 **Model** 时，传入名称 `pet`，默认的表名就是 `pets`。第二个参数指定列名和数据类型，如果是主键，需要更详细地指定。第三个参数是额外的配置，我们传入 `{ timestamps: false }` 是为了关闭 **Sequelize** 的自动添加 `timestamp` 的功能。所有的 **ORM** 框架都有一种很不好的风气，总是自作聪明地加上所谓“自动化”的功能，但是会让人感到完全摸不着头脑。

接下来，我们就可以往数据库中塞一些数据了。我们可以用 **Promise** 的方式写：

```
var now = Date.now();

Pet.create({
  id: 'g-' + now,
  name: 'Gaffey',
  gender: false,
  birth: '2007-07-07',
  createdAt: now,
  updatedAt: now,
  version: 0
}).then(function (p) {
  console.log('created.' + JSON.stringify(p));
}).catch(function (err) {
  console.log('failed: ' + err);
});
```

也可以用 `await` 写：

```
(async () => {
  var dog = await Pet.create({
    id: 'd-' + now,
    name: 'Odie',
    gender: false,
    birth: '2008-08-08',
    createdAt: now,
    updatedAt: now,
    version: 0
  });
  console.log('created: ' + JSON.stringify(dog));
})();
```

显然`await`代码更胜一筹。

查询数据时，用`await`写法如下：

```
(async () => {
  var pets = await Pet.findAll({
    where: {
      name: 'Gaffey'
    }
  });
  console.log(`find ${pets.length} pets:`);
  for (let p of pets) {
    console.log(JSON.stringify(p));
  }
})();
```

如果要更新数据，可以对查询到的实例调用 `save()` 方法：

```
(async () => {
  var p = await queryFromSomewhere();
  p.gender = true;
  p.updatedAt = Date.now();
  p.version++;
  await p.save();
})();
```

如果要删除数据，可以对查询到的实例调用 `destroy()` 方法：

```
(async () => {
  var p = await queryFromSomewhere();
  await p.destroy();
})();
```

运行代码，可以看到Sequelize打印出的每一个SQL语句，便于我们查看：

```
Executing (default): INSERT INTO `pets` (`id`,`name`,`gender`,`birth`,`createdAt`,`updatedAt`,`version`) VALUES ('g-1471961204219','Gaffe
```

Model

我们把通过 `sequelize.define()` 返回的 `Pet` 称为Model，它表示一个数据模型。

我们把通过 `Pet.findAll()` 返回的一个或一组对象称为Model实例，每个实例都可以直接通过 `JSON.stringify` 序列化为JSON字符串。但是它们和普通JSON对象相比，多了一些由Sequelize添加的方法，比如 `save()` 和 `destroy()`。调用这些方法我们可以执行更新或者删除操作。

所以，使用Sequelize操作数据库的一般步骤就是：

首先，通过某个Model对象的 `findAll()` 方法获取实例；

如果要更新实例，先对实例属性赋新值，再调用 `save()` 方法；

如果要删除实例，直接调用 `destroy()` 方法。

注意 `findAll()` 方法可以接收 `where`、`order` 这些参数，这和将要生成的SQL语句是对应的。

文档

Sequelize的API可以参考[官方文档](#)。

参考源码

[hello-sequelize](#)

建立Model

直接使用Sequelize虽然可以，但是存在一些问题。

团队开发时，有人喜欢自己加timestamp：

```
var Pet = sequelize.define('pet', {
  id: {
    type: Sequelize.STRING(50),
    primaryKey: true
  },
  name: Sequelize.STRING(100),
  createdAt: Sequelize.BIGINT,
  updatedAt: Sequelize.BIGINT
}, {
  timestamps: false
});
```

有人又喜欢自增主键，并且自定义表名：

```
var Pet = sequelize.define('pet', {
  id: {
    type: Sequelize.INTEGER,
    autoIncrement: true,
    primaryKey: true
  },
  name: Sequelize.STRING(100)
}, {
  tableName: 't_pet'
});
```

一个大型Web App通常都有几十个映射表，一个映射表就是一个Model。如果按照各自喜好，那业务代码就不好写。Model不统一，很多代码也无法复用。

所以我们需要一个统一的模型，强迫所有Model都遵守同一个规范，这样不但实现简单，而且容易统一风格。

Model

我们首先要定义的就是Model存放的文件夹必须在models内，并且以Model名字命名，例如：Pet.js，User.js等等。

其次，每个Model必须遵守一套规范：

1. 统一主键，名称必须是id，类型必须是STRING(50)；
2. 主键可以自己指定，也可以由框架自动生成（如果为null或undefined）；
3. 所有字段默认为NOT NULL，除非显式指定；
4. 统一timestamp机制，每个Model必须有createdAt、updatedAt和version，分别记录创建时间、修改时间和版本号。其中，createdAt和updatedAt以BIGINT存储时间戳，最大的好处是无需处理时区，排序方便。version每次修改时自增。

所以，我们不要直接使用Sequelize的API，而是通过db.js间接地定义Model。例如，User.js应该定义如下：

```
const db = require('../db');

module.exports = db.defineModel('users', {
  email: {
    type: db.STRING(100),
    unique: true
  },
  passwd: db.STRING(100),
  name: db.STRING(100),
  gender: db.BOOLEAN
});
```

这样，**User**就具有`email`、`passwd`、`name`和`gender`这4个业务字段。`id`、`createdAt`、`updatedAt`和`version`应该自动加上，而不是每个**Model**都去重复定义。

所以，`db.js`的作用就是统一**Model**的定义：

```
const Sequelize = require('sequelize');

console.log('init sequelize...');

var sequelize = new Sequelize('dbname', 'username', 'password', {
  host: 'localhost',
  dialect: 'mysql',
  pool: {
    max: 5,
    min: 0,
    idle: 10000
  }
});

const ID_TYPE = Sequelize.STRING(50);

function defineModel(name, attributes) {
  var attrs = {};
  for (let key in attributes) {
    let value = attributes[key];
    if (typeof value === 'object' && value['type']) {
      value.allowNull = value.allowNull || false;
      attrs[key] = value;
    } else {
      attrs[key] = {
        type: value,
        allowNull: false
      };
    }
  }
  attrs.id = {
    type: ID_TYPE,
    primaryKey: true
  };
  attrs.createdAt = {
    type: Sequelize.BIGINT,
    allowNull: false
  };
  attrs.updatedAt = {
    type: Sequelize.BIGINT,
    allowNull: false
  };
  attrs.version = {
    type: Sequelize.BIGINT,
    allowNull: false
  };
  return sequelize.define(name, attrs, {
    tableName: name,
    timestamps: false,
    hooks: {
      beforeValidate: function (obj) {
        let now = Date.now();
        if (obj.isNewRecord) {
          if (!obj.id) {
            obj.id = generateId();
          }
          obj.createdAt = now;
          obj.updatedAt = now;
          obj.version = 0;
        } else {
          obj.updatedAt = Date.now();
        }
      }
    }
  });
}
```

```
        obj.version++;
    }
}
});
}
```

我们定义的 `defineModel` 就是为了强制实现上述规则。

`Sequelize` 在创建、修改 `Entity` 时会调用我们指定的函数，这些函数通过 `hooks` 在定义 `Model` 时设定。我们在 `beforeValidate` 这个事件中根据是否是 `isNewRecord` 设置主键（如果主键为 `null` 或 `undefined`）、设置时间戳和版本号。

这么一来，`Model` 定义的时候就可以大大简化。

数据库配置

接下来，我们把简单的 `config.js` 拆成3个配置文件：

- `config-default.js`: 存储默认的配置；
- `config-override.js`: 存储特定的配置；
- `config-test.js`: 存储用于测试的配置。

例如，默认的 `config-default.js` 可以配置如下：

```
var config = {
  dialect: 'mysql',
  database: 'nodejs',
  username: 'www',
  password: 'www',
  host: 'localhost',
  port: 3306
};

module.exports = config;
```

而 `config-override.js` 可应用实际配置：

```
var config = {
  database: 'production',
  username: 'www',
  password: 'secret-password',
  host: '192.168.1.199'
};

module.exports = config;
```

`config-test.js` 可应用测试环境的配置：

```
var config = {
  database: 'test'
};

module.exports = config;
```

读取配置的时候，我们用 `config.js` 实现不同环境读取不同的配置文件：

```
const defaultConfig = './config-default.js';
// 可设定为绝对路径, 如 /opt/product/config-override.js
const overrideConfig = './config-override.js';
const testConfig = './config-test.js';

const fs = require('fs');

var config = null;

if (process.env.NODE_ENV === 'test') {
  console.log(`Load ${testConfig}...`);
  config = require(testConfig);
} else {
  console.log(`Load ${defaultConfig}...`);
  config = require(defaultConfig);
  try {
    if (fs.statSync(overrideConfig).isFile()) {
      console.log(`Load ${overrideConfig}...`);
      config = Object.assign(config, require(overrideConfig));
    }
  } catch (err) {
    console.log(`Cannot load ${overrideConfig}.`);
  }
}

module.exports = config;
```

具体的规则是：

1. 先读取 `config-default.js`；
2. 如果不是测试环境，就读取 `config-override.js`，如果文件不存在，就忽略。
3. 如果是测试环境，就读取 `config-test.js`。

这样做的好处是，开发环境下，团队统一使用默认的配置，并且无需 `config-override.js`。部署到服务器时，由运维团队配置好 `config-override.js`，以覆盖 `config-override.js` 的默认设置。测试环境下，本地和CI服务器统一使用 `config-test.js`，测试数据库可以反复清空，不会影响开发。

配置文件表面上写起来很容易，但是，既要保证开发效率，又要避免服务器配置文件泄漏，还要能方便地执行测试，就需要一开始搭建出好的结构，才能提升工程能力。

使用Model

要使用Model，就需要引入对应的Model文件，例如： `User.js`。一旦Model多了起来，如何引用也是一件麻烦事。

自动化永远比手工做效率高，而且更可靠。我们写一个 `model.js`，自动扫描并导入所有Model：

```

const fs = require('fs');
const db = require('./db');

let files = fs.readdirSync(__dirname + '/models');

let js_files = files.filter((f)=>{
  return f.endsWith('.js');
}, files);

module.exports = {};

for (let f of js_files) {
  console.log(`import model from file ${f}...`);
  let name = f.substring(0, f.length - 3);
  module.exports[name] = require(__dirname + '/models/' + f);
}

module.exports.sync = () => {
  db.sync();
};

```

这样，需要用的时候，写起来就像这样：

```

const model = require('./model');

let
  Pet = model.Pet,
  User = model.User;

var pet = await Pet.create({ ... });

```

工程结构

最终，我们创建的工程 `model-sequelize` 结构如下：

```
model-sequelize/  
|  
+- .vscode/  
| |  
| +- launch.json <-- VSCode 配置文件  
|  
+- models/ <-- 存放所有Model  
| |  
| +- Pet.js <-- Pet  
| |  
| +- User.js <-- User  
|  
+- config.js <-- 配置文件入口  
|  
+- config-default.js <-- 默认配置文件  
|  
+- config-test.js <-- 测试配置文件  
|  
+- db.js <-- 如何定义Model  
|  
+- model.js <-- 如何导入Model  
|  
+- init-db.js <-- 初始化数据库  
|  
+- app.js <-- 业务代码  
|  
+- package.json <-- 项目描述文件  
|  
+- node_modules/ <-- npm安装的所有依赖包
```

注意到我们其实不需要创建表的SQL，因为Sequelize提供了一个`sync()`方法，可以自动创建数据库。这个功能在开发和生产环境中没有什么用，但是在测试环境中非常有用。测试时，我们可以用`sync()`方法自动创建出表结构，而不是自己维护SQL脚本。这样，可以随时修改Model的定义，并立刻运行测试。开发环境下，首次使用`sync()`也可以自动创建出表结构，避免了手动运行SQL的问题。

`init-db.js`的代码非常简单：

```
const model = require('./model.js');  
model.sync();  
  
console.log('init db ok.');
```

```
process.exit(0);
```

它最大的好处是避免了手动维护一个SQL脚本。

参考源码

[model-sequelize](#)

mocha

如果你听说过“测试驱动开发”（TDD: Test-Driven Development），单元测试就不陌生。

单元测试是用来对一个模块、一个函数或者一个类来进行正确性检验的测试工作。

比如对函数`abs()`，我们可以编写出以下几个测试用例：

输入正数，比如1、1.2、0.99，期待返回值与输入相同；

输入负数，比如-1、-1.2、-0.99，期待返回值与输入相反；

输入0，期待返回0；

输入非数值类型，比如`null`、`[]`、`{}`，期待抛出`Error`。

把上面的测试用例放到一个测试模块里，就是一个完整的单元测试。

如果单元测试通过，说明我们测试的这个函数能够正常工作。如果单元测试不通过，要么函数有bug，要么测试条件输入不正确，总之，需要修复使单元测试能够通过。

单元测试通过后有什么意义呢？如果我们对`abs()`函数代码做了修改，只需要再跑一遍单元测试，如果通过，说明我们的修改不会对`abs()`函数原有的行为造成影响，如果测试不通过，说明我们的修改与原有行为不一致，要么修改代码，要么修改测试。

这种以测试为驱动的开发模式最大的好处就是确保一个程序模块的行为符合我们设计的测试用例。在将来修改的时候，可以极大程度地保证该模块行为仍然是正确的。

mocha

`mocha`是JavaScript的一种单元测试框架，既可以在浏览器环境下运行，也可以在Node.js环境下运行。

使用`mocha`，我们就只需要专注于编写单元测试本身，然后，让`mocha`去自动运行所有的测试，并给出测试结果。

`mocha`的特点主要有：

1. 既可以测试简单的JavaScript函数，又可以测试异步代码，因为异步是JavaScript的特性之一；
2. 可以自动运行所有测试，也可以只运行特定的测试；
3. 可以支持`before`、`after`、`beforeEach`和`afterEach`来编写初始化代码。

我们会详细讲解如何使用`mocha`编写自动化测试，以及如何测试异步代码。

编写测试

假设我们编写了一个 `hello.js`，并且输出一个简单的求和函数：

```
// hello.js

module.exports = function (...rest) {
  var sum = 0;
  for (let n of rest) {
    sum += n;
  }
  return sum;
};
```

这个函数非常简单，就是对输入的任意参数求和并返回结果。

如果我们想对这个函数进行测试，可以写一个 `test.js`，然后使用Node.js提供的 `assert` 模块进行断言：

```
// test.js

const assert = require('assert');
const sum = require('./hello');

assert.strictEqual(sum(), 0);
assert.strictEqual(sum(1), 1);
assert.strictEqual(sum(1, 2), 3);
assert.strictEqual(sum(1, 2, 3), 6);
```

`assert` 模块非常简单，它断言一个表达式为`true`。如果断言失败，就抛出`Error`。可以在Node.js文档中查看 `assert` 模块的[所有API](#)。

单独写一个 `test.js` 的缺点是没法自动运行测试，而且，如果第一个`assert`报错，后面的测试也执行不了了。

如果有很多测试需要运行，就必须把这些测试全部组织起来，然后统一执行，并且得到执行结果。这就是我们为什么要用mocha来编写并运行测试。

mocha test

我们创建 `hello-test` 工程来编写 `hello.js` 以及相关测试。工程结构如下：

```
hello-test/
|
+- .vscode/
| |
| +- launch.json <-- VSCode 配置文件
|
+- hello.js <-- 待测试js文件
|
+- test/ <-- 存放所有test
| |
| +- hello-test.js <-- 测试文件
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

我们首先在 `package.json` 中添加mocha的依赖包。和其他依赖包不同，这次我们并没有把依赖包添加到 `"dependencies"` 中，而是 `"devDependencies"`：

```
{
  ...

  "dependencies": {},
  "devDependencies": {
    "mocha": "3.0.2"
  }
}
```

如果一个模块在运行的时候并不需要，仅仅在开发时才需要，就可以放到 `devDependencies` 中。这样，正式打包发布时，`devDependencies` 的包不会被包含进来。

然后使用 `npm install` 安装。

注意，很多文章会让你用命令 `npm install -g mocha` 把 `mocha` 安装到全局 `module` 中。这是不需要的。尽量不要安装全局模块，因为全局模块会影响到所有 `Node.js` 的工程。

紧接着，我们在 `test` 目录下创建 `hello-test.js` 来编写测试。

`mocha` 默认会执行 `test` 目录下的所有测试，不要去改变默认目录。

`hello-test.js` 内容如下：

```
const assert = require('assert');

const sum = require('../hello');

describe('#hello.js', () => {

  describe('#sum()', () => {
    it('sum() should return 0', () => {
      assert.strictEqual(sum(), 0);
    });

    it('sum(1) should return 1', () => {
      assert.strictEqual(sum(1), 1);
    });

    it('sum(1, 2) should return 3', () => {
      assert.strictEqual(sum(1, 2), 3);
    });

    it('sum(1, 2, 3) should return 6', () => {
      assert.strictEqual(sum(1, 2, 3), 6);
    });
  });
});
```

这里我们使用 `mocha` 默认的 `BDD-style` 的测试。`describe` 可以任意嵌套，以便把相关测试看成一组测试。

每个 `it("name", function() {...})` 就代表一个测试。例如，为了测试 `sum(1, 2)`，我们这样写：

```
it('sum(1, 2) should return 3', () => {
  assert.strictEqual(sum(1, 2), 3);
});
```

编写测试的原则是，一次只测一种情况，且测试代码要非常简单。我们编写多个测试来分别测试不同的输入，并使用 `assert` 判断输出是否是我们所期望的。

运行测试

下一步，我们就可以用 `mocha` 运行测试了。

如何运行？有三种方法。

方法一，可以打开命令提示符，切换到 `hello-test` 目录，然后执行命令：

```
C:\...\hello-test> node_modules\mocha\bin\mocha
```

`mocha` 就会自动执行所有测试，然后输出如下：

```
#hello.js
#sum()
  □ sum() should return 0
  □ sum(1) should return 1
  □ sum(1, 2) should return 3
  □ sum(1, 2, 3) should return 6
4 passing (7ms)
```

这说明我们编写的4个测试全部通过。如果没有通过，要么修改测试代码，要么修改 `hello.js`，直到测试全部通过为止。

方法二，我们在 `package.json` 中添加 `npm` 命令：

```
{
  ...

  "scripts": {
    "test": "mocha"
  },

  ...
}
```

然后在 `hello-test` 目录下执行命令：

```
C:\...\hello-test> npm test
```

可以得到和上面一样的输出。这种方式通过 `npm` 执行命令，输入的命令比较简单。

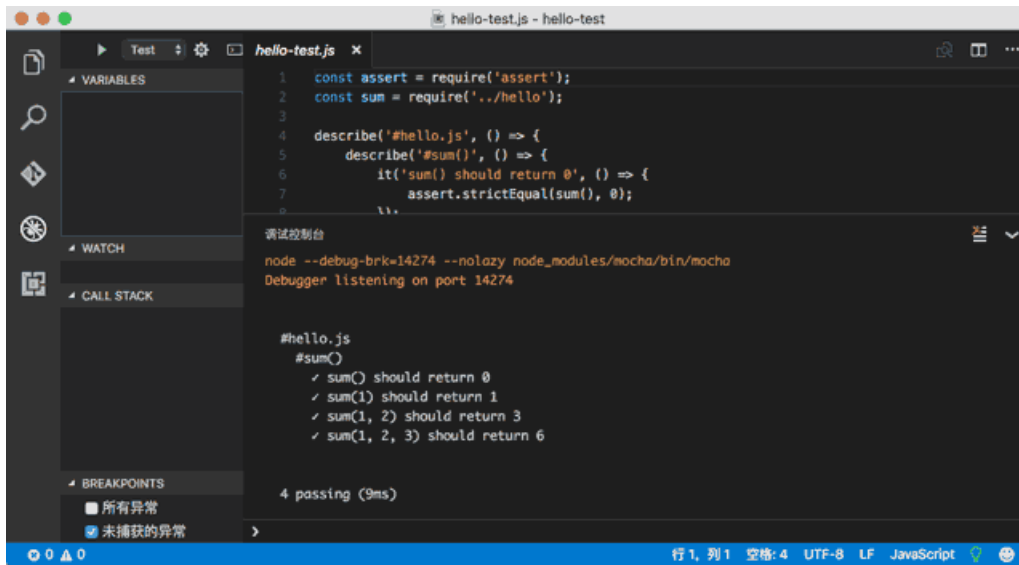
方法三，我们在VS Code中创建配置文件 `.vscode/launch.json`，然后编写两个配置选项：

```

{
  "version": "0.2.0",
  "configurations": [
    {
      "name": "Run",
      "type": "node",
      "request": "launch",
      "program": "${workspaceRoot}/hello.js",
      "stopOnEntry": false,
      "args": [],
      "cwd": "${workspaceRoot}",
      "preLaunchTask": null,
      "runtimeExecutable": null,
      "runtimeArgs": [
        "--nolazy"
      ],
      "env": {
        "NODE_ENV": "development"
      },
      "externalConsole": false,
      "sourceMaps": false,
      "outDir": null
    },
    {
      "name": "Test",
      "type": "node",
      "request": "launch",
      "program": "${workspaceRoot}/node_modules/mocha/bin/mocha",
      "stopOnEntry": false,
      "args": [],
      "cwd": "${workspaceRoot}",
      "preLaunchTask": null,
      "runtimeExecutable": null,
      "runtimeArgs": [
        "--nolazy"
      ],
      "env": {
        "NODE_ENV": "test"
      },
      "externalConsole": false,
      "sourceMaps": false,
      "outDir": null
    }
  ]
}

```

注意第一个配置选项 **Run** 是正常执行一个.js文件，第二个配置选项 **Test** 我们填入 `"program": "${workspaceRoot}/node_modules/mocha/bin/mocha"`，并设置 **env** 为 `"NODE_ENV": "test"`，这样，就可以在VS Code中打开Debug面板，选择 **Test**，运行，即可在Console面板中看到测试结果：



before和after

在测试前初始化资源，测试后释放资源是非常常见的。mocha提供了before、after、beforeEach和afterEach来实现这些功能。

我们把`hello-test.js`改为：

```
const assert = require('assert');
const sum = require('../hello');

describe('#hello.js', () => {
  describe('#sum()', () => {
    before(function () {
      console.log('before:');
    });

    after(function () {
      console.log('after:');
    });

    beforeEach(function () {
      console.log('  beforeEach:');
    });

    afterEach(function () {
      console.log('  afterEach:');
    });

    it('sum() should return 0', () => {
      assert.strictEqual(sum(), 0);
    });

    it('sum(1) should return 1', () => {
      assert.strictEqual(sum(1), 1);
    });

    it('sum(1, 2) should return 3', () => {
      assert.strictEqual(sum(1, 2), 3);
    });

    it('sum(1, 2, 3) should return 6', () => {
      assert.strictEqual(sum(1, 2, 3), 6);
    });
  });
});
```

再次运行，可以看到每个**test**执行前后会分别执行`beforeEach()`和`afterEach()`，以及一组**test**执行前后会分别执行`before()`和`after()`：

```
#hello.js
#sum()
before:
  beforeEach:
    □ sum() should return 0
  afterEach.
  beforeEach:
    □ sum(1) should return 1
  afterEach.
  beforeEach:
    □ sum(1, 2) should return 3
  afterEach.
  beforeEach:
    □ sum(1, 2, 3) should return 6
  afterEach.
after.
4 passing (8ms)
```

参考源码

[hello-test](#)

异步测试

用mocha测试一个函数是非常简单的，但是，在JavaScript的世界中，更多的时候，我们编写的是异步代码，所以，我们需要用mocha测试异步函数。

我们把上一节的

hello-test

工程复制一份，重命名为

async-test

，然后，把

hello.js

改造为异步函数：

```
const fs = require('mz/fs');

// a simple async function:
module.exports = async () => {
  let expression = await fs.readFile('./data.txt', 'utf-8');
  let fn = new Function('return ' + expression);
  let r = fn();
  console.log(`Calculate: ${expression} = ${r}`);
  return r;
};
```

这个async函数通过读取

data.txt

的内容获取表达式，这样它就变成了异步。我们编写一个

data.txt

文件，内容如下：

```
1 + (2 + 4) * (9 - 2) / 3
```

别忘了在

package.json

中添加依赖包：

```
"dependencies": {
  "mz": "2.4.0"
},
```

紧接着，我们在

test

目录中添加一个

await-test.js

，测试

hello.js

的async函数。

我们先看看mocha如何实现异步测试。

如果要测试同步函数，我们传入无参数函数即可：

```
it('test sync function', function () {
  // TODO:
  assert(true);
});
```

如果要测试异步函数，我们要传入的函数需要带一个参数，通常命名为

done

：

```
it('test async function', function (done) {
  fs.readFile('filepath', function (err, data) {
    if (err) {
      done(err);
    } else {
      done();
    }
  });
});
```

测试异步函数需要在函数内部手动调用

done()

表示测试成功，

done(err)

表示测试出错。

对于用ES7的async编写的函数，我们可以这么写：

```
it('#async with done', (done) => {
  (async function () {
    try {
      let r = await hello();
      assert.strictEqual(r, 15);
      done();
    } catch (err) {
      done(err);
    }
  })();
});
```

但是用`try...catch`太麻烦。还有一种更简单的写法，就是直接把`async`函数当成同步函数来测试：

```
it('#async function', async () => {
  let r = await hello();
  assert.strictEqual(r, 15);
});
```

这么写异步测试，太简单了有木有！

我们把上一个 `hello-test` 工程复制为 `async-test`，结构如下：

```
async-test/
|
+- .vscode/
| |
| +- launch.json <-- VSCode 配置文件
|
+- hello.js <-- 待测试js文件
|
+- data.txt <-- 数据文件
|
+- test/ <-- 存放所有test
| |
| +- await-test.js <-- 异步测试
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

现在，在命令行窗口运行命令 `node_modules\mocha\bin\mocha`，测试就可以正常执行：

```
#async hello
#asyncCalculate()
Calculate: 1 + (2 + 4) * (9 - 2) / 3 = 15
  □ #async function
  1 passing (11ms)
```

第二种方法是在 `package.json` 中把 `script` 改为：

```
"scripts": {
  "test": "mocha"
}
```

这样就可以在命令行窗口通过 `npm test` 执行测试。

第三种方法是在VS Code配置文件中把 `program` 改为：

```
"program": "${workspaceRoot}/node_modules/mocha/bin/mocha"
```


这样就可以在VS Code中直接运行测试。

编写异步代码时，我们要坚持使用 `async` 和 `await` 关键字，这样，编写测试也同样简单。

参考源码

[async-test](#)

Http测试

用mocha测试一个async函数是非常方便的。现在，当我们有了一个koa的Web应用程序时，我们怎么用mocha来自动化测试Web应用程序呢？

一个简单的想法就是在测试前启动koa的app，然后运行async测试，在测试代码中发送http请求，收到响应后检查结果，这样，一个基于http接口的测试就可以自动运行。

我们先创建一个最简单的koa应用，结构如下：

```
koa-test/
|
+- .vscode/
| |
| +- launch.json <-- VSCode 配置文件
|
+- app.js <-- koa app文件
|
+- start.js <-- app启动入口
|
+- test/ <-- 存放所有test
| |
| +- app-test.js <-- 异步测试
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

这个koa应用和前面的koa应用稍有不同的是，`app.js`只负责创建`app`实例，并不监听端口：

```
// app.js

const Koa = require('koa');

const app = new Koa();

app.use(async (ctx, next) => {
  const start = new Date().getTime();
  await next();
  const ms = new Date().getTime() - start;
  console.log(`${ctx.request.method} ${ctx.request.url}: ${ms}ms`);
  ctx.response.set('X-Response-Time', `${ms}ms`);
});

app.use(async (ctx, next) => {
  var name = ctx.request.query.name || 'world';
  ctx.response.type = 'text/html';
  ctx.response.body = `<h1>Hello, ${name}!</h1>`;
});

module.exports = app;
```

而`start.js`负责真正启动应用：

```
// start.js

const app = require('./app');

app.listen(3000);
console.log('app started at port 3000...');
```

这样做的目的是便于后面的测试。

紧接着，我们在 `test` 目录下创建 `app-test.js`，来测试这个koa应用。

在测试前，我们在 `package.json` 中添加 `devDependencies`，除了 `mocha` 外，我们还需要一个简单而强大的测试模块 `supertest`：

```
{
  ...
  "devDependencies": {
    "mocha": "3.0.2",
    "supertest": "3.0.0"
  }
}
```

运行 `npm install` 后，我们开始编写测试：

```
// app-test.js

const
  request = require('supertest'),
  app = require('../app');

describe('#test koa app', () => {

  let server = app.listen(9900);

  describe('#test server', () => {

    it('#test GET /', async () => {
      let res = await request(server)
        .get('/')
        .expect('Content-Type', /text\/html/)
        .expect(200, '<h1>Hello, world!</h1>');
    });

    it('#test GET /path?name=Bob', async () => {
      let res = await request(server)
        .get('/path?name=Bob')
        .expect('Content-Type', /text\/html/)
        .expect(200, '<h1>Hello, Bob!</h1>');
    });
  });
});
```

在测试中，我们首先导入 `supertest` 模块，然后导入 `app` 模块，注意我们已经在 `app.js` 中移除了 `app.listen(3000);` 语句，所以，这里我们用：

```
let server = app.listen(9900);
```

让 `app` 实例监听在 `9900` 端口上，并且获得返回的 `server` 实例。

在测试代码中，我们使用：

```
let res = await request(server).get('/');
```

就可以构造一个GET请求，发送给koa的应用，然后获得响应。

可以手动检查响应对象，例如，`res.body`，还可以利用 `supertest` 提供的 `expect()` 更方便地断言响应的HTTP代码、返回内容和HTTP头。断言HTTP头时可用使用正则表达式。例如，下面的断言：

```
.expect('Content-Type', /text\/html/)
```

可用成功匹配到 `Content-Type` 为 `text/html`、`text/html; charset=utf-8` 等值。

当所有测试运行结束后，`app` 实例会自动关闭，无需清理。

利用 `mocha` 的异步测试，配合 `supertest`，我们可以用简单的代码编写端到端的 HTTP 自动化测试。

参考源码

[koa-test](#)

WebSocket

WebSocket是HTML5新增的协议，它的目的是在浏览器和服务器之间建立一个不受限的双向通信的通道，比如说，服务器可以在任意时刻发送消息给浏览器。

为什么传统的HTTP协议不能做到**WebSocket**实现的功能？这是因为HTTP协议是一个请求—响应协议，请求必须先由浏览器发给服务器，服务器才能响应这个请求，再把数据发送给浏览器。换句话说，浏览器不主动请求，服务器是没法主动发数据给浏览器的。

这样一来，要在浏览器中搞一个实时聊天，在线炒股（不鼓励），或者在线多人游戏的话就没法实现了，只能借助Flash这些插件。

也有人说，HTTP协议其实也能实现啊，比如用轮询或者**Comet**。轮询是指浏览器通过JavaScript启动一个定时器，然后以固定的间隔给服务器发请求，询问服务器有没有新消息。这个机制的缺点一是实时性不够，二是频繁的请求会给服务器带来极大的压力。

Comet本质上也是轮询，但是在没有消息的情况下，服务器先拖一段时间，等到有消息了再回复。这个机制暂时地解决了实时性问题，但是它带来了新的问题：以多线程模式运行的服务器会让大部分线程大部分时间都处于挂起状态，极大地浪费服务器资源。另外，一个HTTP连接在长时间没有数据传输的情况下，链路上的任何一个网关都可能关闭这个连接，而网关是我们不可控的，这就要求**Comet**连接必须定期发一些ping数据表示连接“正常工作”。

以上两种机制都治标不治本，所以，HTML5推出了WebSocket标准，让浏览器和服务器之间可以建立无限制的全双工通信，任何一方都可以主动发消息给对方。

WebSocket协议

WebSocket并不是全新的协议，而是利用了HTTP协议来建立连接。我们来看看WebSocket连接是如何创建的。

首先，**WebSocket**连接必须由浏览器发起，因为请求协议是一个标准的HTTP请求，格式如下：

```
GET ws://localhost:3000/ws/chat HTTP/1.1
Host: localhost
Upgrade: websocket
Connection: Upgrade
Origin: http://localhost:3000
Sec-WebSocket-Key: client-random-string
Sec-WebSocket-Version: 13
```

该请求和普通的HTTP请求有几点不同：

1. GET请求的地址不是类似 `/path/`，而是以 `ws://` 开头的地址；
2. 请求头 `Upgrade: websocket` 和 `Connection: Upgrade` 表示这个连接将要被转换为WebSocket连接；
3. `Sec-WebSocket-Key` 是用于标识这个连接，并非用于加密数据；
4. `Sec-WebSocket-Version` 指定了WebSocket的协议版本。

随后，服务器如果接受该请求，就会返回如下响应：

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: server-random-string
```

该响应代码 `101` 表示本次连接的HTTP协议即将被更改，更改后的协议就是 `Upgrade: websocket` 指定的WebSocket协议。

版本号和子协议规定了双方能理解的数据格式，以及是否支持压缩等等。如果仅使用WebSocket的API，就不需要关心这些。

现在，一个WebSocket连接就建立成功，浏览器和服务器就可以随时主动发送消息给对方。消息有两种，一种是文本，一种是二进制数据。通常，我们可以发送JSON格式的文本，这样，在浏览器处理起来就十分容易。

为什么WebSocket连接可以实现全双工通信而HTTP连接不行呢？实际上HTTP协议是建立在TCP协议之上的，TCP协议本身就实现了全双工通信，但是HTTP协议的请求—应答机制限制了全双工通信。WebSocket连接建立以后，其实只是简单规定了一下：接下来，咱们通信就不使用HTTP协议了，直接互相发数据吧。

安全的WebSocket连接机制和HTTPS类似。首先，浏览器用 `wss://xxx` 创建WebSocket连接时，会先通过HTTPS创建安全的连接，然后，该HTTPS连接升级为WebSocket连接，底层通信走的仍然是安全的SSL/TLS协议。

浏览器

很显然，要支持WebSocket通信，浏览器得支持这个协议，这样才能发出`ws://xxx`的请求。目前，支持WebSocket的主流浏览器如下：

- Chrome
- Firefox
- IE >= 10
- Safari >= 6
- Android >= 4.4
- iOS >= 8

服务器

由于WebSocket是一个协议，服务器具体怎么实现，取决于所用编程语言和框架本身。**Node.js**本身支持的协议包括TCP协议和HTTP协议，要支持WebSocket协议，需要对Node.js提供的HTTPServer做额外的开发。已经有若干基于Node.js的稳定可靠的WebSocket实现，我们直接用npm安装使用即可。

使用ws

要使用WebSocket，关键在于服务器端支持，这样，我们才有可能用支持WebSocket的浏览器使用WebSocket。

ws模块

在Node.js中，使用最广泛的WebSocket模块是ws，我们创建一个hello-ws的VS Code工程，然后在package.json中添加ws的依赖：

```
"dependencies": {  
  "ws": "1.1.1"  
}
```

整个工程结构如下：

```
hello-ws/  
|  
+- .vscode/  
| |  
| +- launch.json <-- VSCode 配置文件  
|  
+- app.js <-- 启动js文件  
|  
+- package.json <-- 项目描述文件  
|  
+- node_modules/ <-- npm安装的所有依赖包
```

运行npm install后，我们就可以在app.js中编写WebSocket的服务器端代码。

创建一个WebSocket的服务器实例非常容易：

```
// 导入WebSocket模块：  
const WebSocket = require('ws');  
  
// 引用Server类：  
const WebSocketServer = WebSocket.Server;  
  
// 实例化：  
const wss = new WebSocketServer({  
  port: 3000  
});
```

这样，我们就在3000端口上打开了一个WebSocket Server，该实例由变量wss引用。

接下来，如果有WebSocket请求接入，wss对象可以响应connection事件来处理这个WebSocket：

```
wss.on('connection', function (ws) {  
  console.log(`[SERVER] connection()`);  
  ws.on('message', function (message) {  
    console.log(`[SERVER] Received: ${message}`);  
    ws.send(`ECHO: ${message}`, (err) => {  
      if (err) {  
        console.log(`[SERVER] error: ${err}`);  
      }  
    });  
  });  
});
```

在connection事件中，回调函数会传入一个WebSocket的实例，表示这个WebSocket连接。对于每个WebSocket连接，我们都要对它绑定某些事件方法来处理不同的事件。这里，我们通过响应message事件，在收到消息后再返回一个ECHO: xxx的消息给客户端。

创建WebSocket连接

现在，这个简单的服务器端WebSocket程序就编写好了。如何真正创建WebSocket并且给服务器发消息呢？方法是在浏览器中写JavaScript代码。

先在VS Code中执行 `app.js`，或者在命令行用 `npm start` 执行。然后，在当前页面下，直接打开可以执行JavaScript代码的浏览器Console，依次输入代码：

```
// 打开一个WebSocket：
var ws = new WebSocket('ws://localhost:3000/test');
// 响应onmessage事件：
ws.onmessage = function(msg) { console.log(msg); };
// 给服务器发送一个字符串：
ws.send('Hello!');
```

一切正常的话，可以看到Console的输出如下：

```
MessageEvent {isTrusted: true, data: "ECHO: Hello!", origin: "ws://localhost:3000", lastEventId: "", source: null...}
```

这样，我们就在浏览器中成功地收到了服务器发送的消息！

如果嫌在浏览器中输入JavaScript代码比较麻烦，我们还可以直接用 `ws` 模块提供的 `WebSocket` 来充当客户端。换句话说，`ws` 模块既包含了服务器端，又包含了客户端。

`ws` 的 `WebSocket` 就表示客户端，它其实就是WebSocketServer响应 `connection` 事件时回调函数传入的变量 `ws` 的类型。

客户端的写法如下：

```
let ws = new WebSocket('ws://localhost:3000/test');

// 打开WebSocket连接后立刻发送一条消息：
ws.on('open', function () {
  console.log(`[CLIENT] open()`);
  ws.send('Hello!');
});

// 响应收到的消息：
ws.on('message', function (message) {
  console.log(`[CLIENT] Received: ${message}`);
})
```

在Node环境下，`ws` 模块的客户端可以用于测试服务器端代码，否则，每次都必须在浏览器执行JavaScript代码。

同源策略

从上面的测试可以看出，WebSocket协议本身不要求同源策略（Same-origin Policy），也就是某个地址为 `http://a.com` 的网页可以通过WebSocket连接到 `ws://b.com`。但是，浏览器会发送 `Origin` 的HTTP头给服务器，服务器可以根据 `Origin` 拒绝这个WebSocket请求。所以，是否要求同源要看服务器端如何检查。

路由

还需要注意到服务器在响应 `connection` 事件时并未检查请求的路径，因此，在客户端打开 `ws://localhost:3000/any/path` 可以写任意的路径。

实际应用中还需要根据不同的路径实现不同的功能。

参考源码

hello-ws

编写聊天室

上一节我们用 `ws` 模块创建了一个 `WebSocket` 应用。但是它只能简单地响应 `ECHO: xxx` 消息，还属于 `Hello, world` 级别的应用。

要创建真正的 `WebSocket` 应用，首先，得有一个基于 `MVC` 的 `Web` 应用，也就是我们在前面用 `koa2` 和 `Nunjucks` 创建的 `Web`，在此基础上，把 `WebSocket` 添加进来，才算完整。

因此，本节的目标是基于 `WebSocket` 创建一个在线聊天室。

首先，我们把前面编写的 `MVC` 工程复制一份，先创建一个完整的 `MVC` 的 `Web` 应用，结构如下：

```
ws-with-koa/
|
+- .vscode/
| |
| +- launch.json <-- VSCode 配置文件
|
+- controllers/ <-- Controller
|
+- views/ <-- html模板文件
|
+- static/ <-- 静态资源文件
|
+- app.js <-- 使用koa的js
|
+- controller.js <-- 扫描注册Controller
|
+- static-files.js <-- 处理静态文件
|
+- templating.js <-- 模版引擎入口
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

然后，把我们需要的依赖包添加到 `package.json`：

```
"dependencies": {
  "ws": "1.1.1",
  "koa": "2.0.0",
  "koa-bodyparser": "3.2.0",
  "koa-router": "7.0.0",
  "nunjucks": "2.4.2",
  "mime": "1.3.4",
  "mz": "2.4.0"
}
```

使用 `npm install` 安装后，我们首先得到了一个标准的基于 `MVC` 的 `koa2` 应用。该应用的核心是一个代表 `koa` 应用的 `app` 变量：

```
const app = new Koa();

// TODO: app.use(...);

app.listen(3000);
```

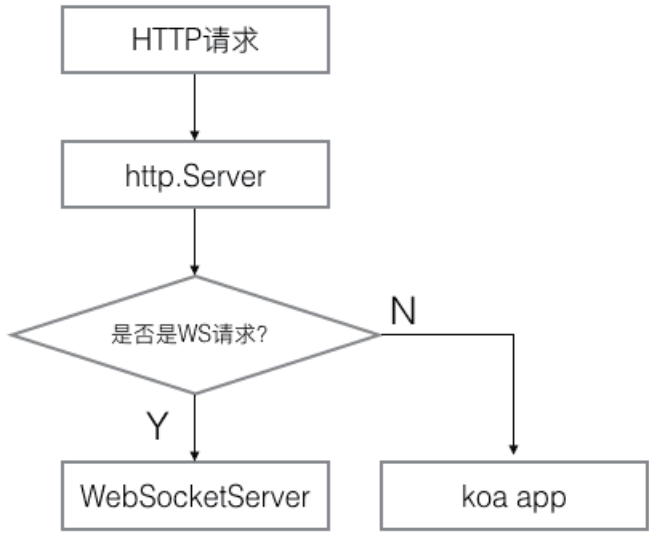
现在第一个问题来了：`koa` 通过 `3000` 端口响应 `HTTP`，我们要新加的 `WebSocketServer` 还能否使用 `3000` 端口？

答案是肯定的。虽然 `WebSocketServer` 可以使用别的端口，但是，统一端口有个最大的好处：

实际应用中，`HTTP` 和 `WebSocket` 都使用标准的 `80` 和 `443` 端口，不需要暴露新的端口，也不需要修改防火墙规则。

在 `3000` 端口被 `koa` 占用后，`WebSocketServer` 如何使用该端口？

实际上，3000端口并非由koa监听，而是koa调用Node标准的http模块创建的http.Server监听的。koa只是把响应函数注册到该http.Server中了。类似的，WebSocketServer也可以把自己的响应函数注册到http.Server中，这样，同一个端口，根据协议，可以分别由koa和ws处理：



把WebSocketServer绑定到同一个端口的关键代码是先获取koa创建的http.Server的引用，再根据http.Server创建WebSocketServer：

```
// koa app的listen()方法返回http.Server:
let server = app.listen(3000);

// 创建WebSocketServer:
let wss = new WebSocketServer({
  server: server
});
```

要始终注意，浏览器创建WebSocket时发送的仍然是标准的HTTP请求。无论是WebSocket请求，还是普通HTTP请求，都会被http.Server处理。具体的处理方式则是由koa和WebSocketServer注入的回调函数实现的。WebSocketServer会首先判断请求是不是WS请求，如果是，它将处理该请求，如果不是，该请求仍由koa处理。

所以，WS请求会直接由WebSocketServer处理，它根本不会经过koa，koa的任何middleware都没有机会处理该请求。

现在第二个问题来了：在koa应用中，可以很容易地认证用户，例如，通过session或者cookie，但是，在响应WebSocket请求时，如何识别用户身份？

一个简单可行的方案是把用户登录后的身份写入Cookie，在koa中，可以使用middleware解析Cookie，把用户绑定到ctx.state.user上。

WS请求也是标准的HTTP请求，所以，服务器也会把Cookie发送过来，这样，我们在用WebSocketServer处理WS请求时，就可以根据Cookie识别用户身份。

先把识别用户身份的逻辑提取为一个单独的函数：

```
function parseUser(obj) {
  if (!obj) {
    return;
  }
  console.log('try parse: ' + obj);
  let s = '';
  if (typeof obj === 'string') {
    s = obj;
  } else if (obj.headers) {
    let cookies = new Cookies(obj, null);
    s = cookies.get('name');
  }
  if (s) {
    try {
      let user = JSON.parse(Buffer.from(s, 'base64').toString());
      console.log(`User: ${user.name}, ID: ${user.id}`);
      return user;
    } catch (e) {
      // ignore
    }
  }
}
```

注意：出于演示目的，该Cookie并没有作Hash处理，实际上它就是一个JSON字符串。

在koa的middleware中，我们很容易识别用户：

```
app.use(async (ctx, next) => {
  ctx.state.user = parseUser(ctx.cookies.get('name') || '');
  await next();
});
```

在WebSocketServer中，就需要响应 `connection` 事件，然后识别用户：

```
wss.on('connection', function (ws) {
  // ws.upgradeReq是一个request对象：
  let user = parseUser(ws.upgradeReq);
  if (!user) {
    // Cookie不存在或无效，直接关闭WebSocket：
    ws.close(4001, 'Invalid user');
  }
  // 识别成功，把user绑定到该WebSocket对象：
  ws.user = user;
  // 绑定WebSocketServer对象：
  ws.wss = wss;
});
```

紧接着，我们要对每个创建成功的WebSocket绑定 `message`、`close`、`error` 等事件处理函数。对于聊天应用来说，每收到一条消息，就需要把该消息广播到所有WebSocket连接上。

先为 `wss` 对象添加一个 `broadcast()` 方法：

```
wss.broadcast = function (data) {
  wss.clients.forEach(function (client) {
    client.send(data);
  });
};
```

在某个WebSocket收到消息后，就可以调用 `wss.broadcast()` 进行广播了：

```
ws.on('message', function (message) {
  console.log(message);
  if (message && message.trim()) {
    let msg = createMessage('chat', this.user, message.trim());
    this.wss.broadcast(msg);
  }
});
```

消息有很多类型，不一定是聊天的消息，还可以有获取用户列表、用户加入、用户退出等多种消息。所以我们用 `createMessage()` 创建一个JSON格式的字符串，发送给浏览器，浏览器端的JavaScript就可以直接使用：

```
// 消息ID:
var messageIndex = 0;

function createMessage(type, user, data) {
  messageIndex ++;
  return JSON.stringify({
    id: messageIndex,
    type: type,
    user: user,
    data: data
  });
}
```

编写页面

相比服务器端的代码，页面的JavaScript代码会更复杂。

聊天室页面可以划分为左侧会话列表和右侧用户列表两部分：

<div>路人甲加入了聊天室</div> <div>路人乙说： 大家好！</div> <div>路人甲说： WebSocket是个好东西！</div> <div>路人丙说： 顶楼主！</div> <div><input type="text"/></div> <div>GO</div>	<ul style="list-style-type: none">• 路人甲• 路人乙• 路人丙
---	---

这里的DOM需要动态更新，因此，状态管理是页面逻辑的核心。

为了简化状态管理，我们用Vue控制左右两个列表：

```
var vmMessageList = new Vue({
  el: '#message-list',
  data: {
    messages: []
  }
});

var vmUserList = new Vue({
  el: '#user-list',
  data: {
    users: []
  }
});
```

会话列表和用户列表初始化为空数组。

紧接着，创建WebSocket连接，响应服务器消息，并且更新会话列表和用户列表：

```
var ws = new WebSocket('ws://localhost:3000/ws/chat');

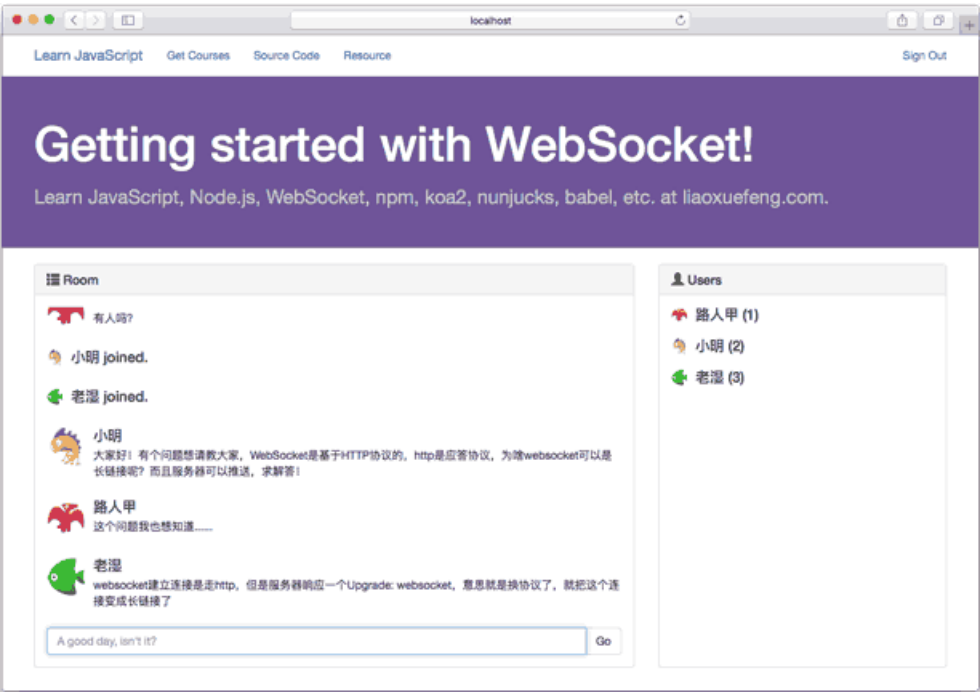
ws.onmessage = function(event) {
  var data = event.data;
  console.log(data);
  var msg = JSON.parse(data);
  if (msg.type === 'list') {
    vmUserList.users = msg.data;
  } else if (msg.type === 'join') {
    addToUserList(vmUserList.users, msg.user);
    addMessage(vmMessageList.messages, msg);
  } else if (msg.type === 'left') {
    removeFromUserList(vmUserList.users, msg.user);
    addMessage(vmMessageList.messages, msg);
  } else if (msg.type === 'chat') {
    addMessage(vmMessageList.messages, msg);
  }
};
```

这样，JavaScript负责更新状态，Vue负责根据状态刷新DOM。以用户列表为例，HTML代码如下：

```
<div id="user-list">
  <div class="media" v-for="user in users">
    <div class="media-left">
      
    </div>
    <div class="media-body">
      <h4 class="media-heading" v-text="user.name"></h4>
    </div>
  </div>
</div>
```

测试的时候，如果在本机测试，需要同时用几个不同的浏览器，这样Cookie互不干扰。

最终的聊天室效果如下：



配置反向代理

如果网站配置了反向代理，例如Nginx，则HTTP和WebSocket都必须通过反向代理连接Node服务器。HTTP的反向代理非常简单，但是要正常连接WebSocket，代理服务器必须支持WebSocket协议。

我们以Nginx为例，编写一个简单的反向代理配置文件。

详细的配置可以参考Nginx的官方博客：[Using NGINX as a WebSocket Proxy](#)

首先要保证Nginx版本>=1.3，然后，通过`proxy_set_header`指令，设定：

```
proxy_set_header Upgrade $http_upgrade;
proxy_set_header Connection "upgrade";
```

Nginx即可理解该连接将使用WebSocket协议。

一个示例配置文件内容如下：

```
server {
    listen      80;
    server_name localhost;

    # 处理静态资源文件：
    location ^~ /static/ {
        root /path/to/ws-with-koa;
    }

    # 处理WebSocket连接：
    location ^~ /ws/ {
        proxy_pass      http://127.0.0.1:3000;
        proxy_http_version 1.1;
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection "upgrade";
    }

    # 其他所有请求：
    location / {
        proxy_pass      http://127.0.0.1:3000;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

参考源码

[ws-with-koa](#)

REST

自从Roy Fielding博士在2000年他的博士论文中提出REST（Representational State Transfer）风格的软件架构模式后，REST就基本上迅速取代了复杂而笨重的SOAP，成为Web API的标准了。

什么是Web API呢？

如果我们想要获取某个电商网站的某个商品，输入 `http://localhost:3000/products/123`，就可以看到id为123的商品页面，但这个结果是HTML页面，它同时混合包含了Product的数据和Product的展示两个部分。对于用户来说，阅读起来没有问题，但是，如果机器读取，就很难从HTML中解析出Product的数据。

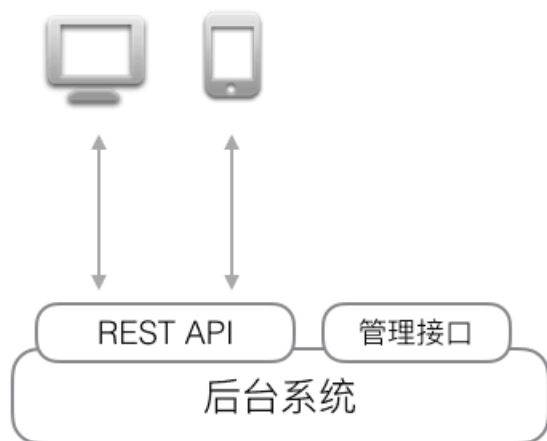
如果一个URL返回的不是HTML，而是机器能直接解析的数据，这个URL就可以看成是一个Web API。比如，读取 `http://localhost:3000/api/products/123`，如果能直接返回Product的数据，那么机器就可以直接读取。

REST就是一种设计API的模式。最常用的数据格式是JSON。由于JSON能直接被JavaScript读取，所以，以JSON格式编写的REST风格的API具有简单、易读、易用的特点。

编写API有什么好处呢？由于API就是把Web App的功能全部封装了，所以，通过API操作数据，可以极大地把前端和后端的代码隔离，使得后端代码易于测试，前端代码编写更简单。

此外，如果我们把前端页面看作是一种用于展示的客户端，那么API就是为客户端提供数据、操作数据的接口。这种设计可以获得极高的扩展性。例如，当用户需要在手机上购买商品时，只需要开发针对iOS和Android的两个客户端，通过客户端访问API，就可以完成通过浏览器页面提供的功能，而后端代码基本无需改动。

当一个Web应用以API的形式对外提供功能时，整个应用的结构就扩展为：



把网页视为一种客户端，是REST架构可扩展的一个关键。

编写REST API

REST API规范

编写REST API，实际上就是编写处理HTTP请求的`async`函数，不过，REST请求和普通的HTTP请求有几个特殊的地方：

1. REST请求仍然是标准的HTTP请求，但是，除了GET请求外，POST、PUT等请求的body是JSON数据格式，请求的`Content-Type`为`application/json`；
2. REST响应返回的结果是JSON数据格式，因此，响应的`Content-Type`也是`application/json`。

REST规范定义了资源的通用访问格式，虽然它不是一个强制要求，但遵守该规范可以让人易于理解。

例如，商品Product就是一种资源。获取所有Product的URL如下：

```
GET /api/products
```

而获取某个指定的Product，例如，id为`123`的Product，其URL如下：

```
GET /api/products/123
```

新建一个Product使用POST请求，JSON数据包含在body中，URL如下：

```
POST /api/products
```

更新一个Product使用PUT请求，例如，更新id为`123`的Product，其URL如下：

```
PUT /api/products/123
```

删除一个Product使用DELETE请求，例如，删除id为`123`的Product，其URL如下：

```
DELETE /api/products/123
```

资源还可以按层次组织。例如，获取某个Product的所有评论，使用：

```
GET /api/products/123/reviews
```

当我们只需要获取部分数据时，可通过参数限制返回的结果集，例如，返回第2页评论，每页10项，按时间排序：

```
GET /api/products/123/reviews?page=2&size=10&sort=time
```

koa处理REST

既然我们已经使用koa作为Web框架处理HTTP请求，因此，我们仍然可以在koa中响应并处理REST请求。

我们先创建一个`rest-hello`的工程，结构如下：


```
rest-hello/
|
+- .vscode/
| |
| +- launch.json <-- VSCode 配置文件
|
+- controllers/
| |
| +- api.js <-- REST API
|
+- app.js <-- 使用koa的js
|
+- controller.js <-- 扫描注册Controller
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
```

在 `package.json` 中，我们需要如下依赖包：

```
"dependencies": {
  "koa": "2.0.0",
  "koa-bodyparser": "3.2.0",
  "koa-router": "7.0.0"
}
```

运行 `npm install` 安装依赖包。

在 `app.js` 中，我们仍然使用标准的koa组件，并自动扫描加载 `controllers` 目录下的所有js文件：

```
const app = new Koa();

const controller = require('./controller');

// parse request body:
app.use(bodyParser());

// add controller:
app.use(controller());

app.listen(3000);
console.log('app started at port 3000...');
```

注意到 `app.use(bodyParser());` 这个语句，它给koa安装了一个解析HTTP请求body的处理函数。如果HTTP请求是JSON数据，我们就可以通过 `ctx.request.body` 直接访问解析后的JavaScript对象。

下面我们编写 `api.js`，添加一个GET请求：

```
// 存储Product列表，相当于模拟数据库：
var products = [{
  name: 'iPhone',
  price: 6999
}, {
  name: 'Kindle',
  price: 999
}];

module.exports = {
  'GET /api/products': async (ctx, next) => {
    // 设置Content-Type:
    ctx.response.type = 'application/json';
    // 设置Response Body:
    ctx.response.body = {
      products: products
    };
  }
}
```

在koa中，我们只需要给`ctx.response.body`赋值一个JavaScript对象，koa会自动把该对象序列化为JSON并输出到客户端。

我们在浏览器中访问 `http://localhost:3000/api/products`，可以得到如下输出：

```
{"products":[{"name":"iPhone","price":6999},{"name":"Kindle","price":999}]}
```

紧接着，我们再添加一个创建Product的API：

```
module.exports = {
  'GET /api/products': async (ctx, next) => {
    ...
  },

  'POST /api/products': async (ctx, next) => {
    var p = {
      name: ctx.request.body.name,
      price: ctx.request.body.price
    };
    products.push(p);
    ctx.response.type = 'application/json';
    ctx.response.body = p;
  }
};
```

这个POST请求无法在浏览器中直接测试。但是我们可以通过`curl`命令在命令提示符窗口测试这个API。我们输入如下命令：

```
curl -H 'Content-Type: application/json' -X POST -d '{"name":"XBox","price":3999}' http://localhost:3000/api/products
```

得到的返回内容如下：

```
{"name":"XBox","price":3999}
```

我们再次在浏览器中访问 `http://localhost:3000/api/products`，可以得到更新后的输出如下：

```
{"products":[{"name":"iPhone","price":6999},{"name":"Kindle","price":999},{"name":"XBox","price":3999}]}
```

可见，在koa中处理REST请求是非常简单的。`bodyParser()`这个middleware可以解析请求的JSON数据并绑定到`ctx.request.body`上，输出JSON时我们先指定`ctx.response.type = 'application/json'`，然后把JavaScript对象赋值给`ctx.response.body`就完成了REST请求的处理。

参考源码

[rest-hello](#)

开发REST API

在上一节中，我们演示了如何在koa项目中使用REST。其实，使用REST和使用MVC是类似的，不同的是，提供REST的Controller处理函数最后不调用`render()`去渲染模板，而是把结果直接用JSON序列化返回给客户端。

使用REST虽然非常简单，但是，设计一套合理的REST框架却需要仔细考虑很多问题。

问题一：如何组织URL

在实际工程中，一个Web应用既有REST，还有MVC，可能还需要集成其他第三方系统。如何组织URL？

一个简单的方法是通过固定的前缀区分。例如，`/static/`开头的URL是静态资源文件，类似的，`/api/`开头的URL就是REST API，其他URL是普通的MVC请求。

使用不同的子域名也可以区分，但对于中小项目来说配置麻烦。随着项目的扩大，将来仍然可以把单域名拆成多域名。

问题二：如何统一输出REST

如果每个异步函数都编写下面这样的代码：

```
// 设置Content-Type:
ctx.response.type = 'application/json';
// 设置Response Body:
ctx.response.body = {
  products: products
};
```

很显然，这样的重复代码很容易导致错误，例如，写错了字符串`'application/json'`，或者漏写了`ctx.response.type = 'application/json'`，都会导致浏览器得不到JSON数据。

回忆我们集成Nunjucks模板引擎的方法：通过一个middleware给`ctx`添加一个`render()`方法，Controller就可以直接使用`ctx.render('view', model)`来渲染模板，不必编写重复的代码。

类似的，我们也可以通过一个middleware给`ctx`添加一个`rest()`方法，直接输出JSON数据。

由于我们给所有REST API一个固定的URL前缀`/api/`，所以，这个middleware还需要根据path来判断当前请求是否是一个REST请求，如果是，我们才给`ctx`绑定`rest()`方法。

我们把这个middleware先写出来，命名为`rest.js`：

```
module.exports = {
  restify: (pathPrefix) => {
    // REST API前缀，默认为/api/
    pathPrefix = pathPrefix || '/api/';
    return async (ctx, next) => {
      // 是否是REST API前缀？
      if (ctx.request.path.startsWith(pathPrefix)) {
        // 绑定rest()方法：
        ctx.rest = (data) => {
          ctx.response.type = 'application/json';
          ctx.response.body = data;
        }
        await next();
      } else {
        await next();
      }
    };
  }
};
```

这样，任何支持REST的异步函数只需要简单地调用：

```
ctx.rest({
  data: 123
});
```

就完成了**REST**请求的处理。

问题三：如何处理错误

这个问题实际上有两部分。

第一，当**REST API**请求出错时，我们如何返回错误信息？

第二，当客户端收到**REST**响应后，如何判断是成功还是错误？

这两部分还必须统一考虑。

REST架构本身对错误处理并没有统一的规定。实际应用时，各种各样的错误处理机制都有。有的设计得比较合理，有的设计得不合理，导致客户端尤其是手机客户端处理**API**简直就是噩梦。

在涉及到**REST API**的错误时，我们必须先意识到，客户端会遇到两种类型的**REST API**错误。

一类是类似**403**，**404**，**500**等错误，这些错误实际上是**HTTP**请求可能发生的错误。**REST**请求只是一种请求类型和响应类型均为**JSON**的**HTTP**请求，因此，这些错误在**REST**请求中也会发生。

针对这种类型的错误，客户端除了提示用户“出现了网络错误，稍后重试”以外，并无法获得具体的错误信息。

另一类错误是业务逻辑的错误，例如，输入了不合法的**Email**地址，试图删除一个不存在的**Product**，等等。这种类型的错误完全可以通过**JSON**返回给客户端，这样，客户端可以根据错误信息提示用户“**Email**不合法”等，以便用户修复后重新请求**API**。

问题的关键在于客户端必须能区分出这两种类型的错误。

第一类的错误实际上客户端可以识别，并且我们也无法操控**HTTP**服务器的错误码。

第二类的错误信息是一个**JSON**字符串，例如：

```
{
  "code": "10000",
  "message": "Bad email address"
}
```

但是**HTTP**的返回码应该用啥？

有的**Web**应用使用**200**，这样客户端在识别出第一类错误后，如果遇到**200**响应，则根据响应的**JSON**判断是否有错误。这种方式对于动态语言（例如，**JavaScript**，**Python**等）非常容易：

```
var result = JSON.parse(response.data);
if (result.code) {
  // 有错误：
  alert(result.message);
} else {
  // 没有错误
}
```

但是，对于静态语言（例如，**Java**）就比较麻烦，很多时候，不得不做两次数列化：

```
APIError err = objectMapper.readValue(jsonString, APIError.class);
if (err.code == null) {
  // 没有错误，还需要重新转换：
  User user = objectMapper.readValue(jsonString, User.class);
} else {
  // 有错误：
}
```

有的Web应用对正确的REST响应使用[200]，对错误的REST响应使用[400]，这样，客户端即是静态语言，也可以根据HTTP响应码判断是否出错，出错时直接把结果反序列化为[APIError]对象。

两种方式各有优劣。我们选择第二种，[200]表示成功响应，[400]表示失败响应。

但是，要注意，**绝不能**混合其他HTTP错误码。例如，使用[401]响应“登录失败”，使用[403]响应“权限不够”。这会使客户端无法有效识别HTTP错误码和业务错误，其原因在于HTTP协议定义的错误码十分偏向底层，而REST API属于“高层”协议，不应该复用底层的错误码。

问题四：如何定义错误码

REST架构本身同样没有标准的错误码定义一说，因此，有的Web应用使用数字[1000]、[1001].....作为错误码，例如Twitter和新浪微博，有的Web应用使用字符串作为错误码，例如YouTube。到底哪一种比较好呢？

我们强烈建议使用字符串作为错误码。原因在于，使用数字作为错误码时，API提供者需要维护一份错误码代码说明表，并且，该文档必须时刻与API发布同步，否则，客户端开发者遇到一个文档上没有写明的错误码，就完全不知道发生了什么错误。

使用字符串作为错误码，最大的好处在于不用查表，根据字面意思也能猜个八九不离十。例如，YouTube API如果返回一个错误[authError]，基本上能猜到是因为认证失败。

我们定义的REST API错误格式如下：

```
{
  "code": "错误代码",
  "message": "错误描述信息"
}
```

其中，错误代码命名规范为[大类:子类]，例如，口令不匹配的登录错误代码为[auth:bad_password]，用户名不存在的登录错误代码为[auth:user_not_found]。这样，客户端既可以简单匹配某个类别的错误，也可以精确匹配某个特定的错误。

问题五：如何返回错误

如果一个REST异步函数想要返回错误，一个直观的想法是调用[ctx.rest()]：

```
user = processLogin(username, password);
if (user != null) {
  ctx.rest(user);
} else {
  ctx.response.status = 400;
  ctx.rest({
    code: 'auth:user_not_found',
    message: 'user not found'
  });
}
```

这种方式不好，因为控制流程会混乱，而且，错误只能在Controller函数中输出。

更好的方式是异步函数直接用[throw]语句抛出错误，让middleware去处理错误：

```
user = processLogin(username, password);
if (user != null) {
  ctx.rest(user);
} else {
  throw new APIError('auth:user_not_found', 'user not found');
}
```

这种方式可以在异步函数的任何地方抛出错误，包括调用的子函数内部。

我们只需要稍稍改写一个middleware就可以处理错误：

```

module.exports = {
  APIError: function (code, message) {
    this.code = code || 'internal:unknown_error';
    this.message = message || '';
  },
  restify: (pathPrefix) => {
    pathPrefix = pathPrefix || '/api/';
    return async (ctx, next) => {
      if (ctx.request.path.startsWith(pathPrefix)) {
        // 绑定rest()方法:
        ctx.rest = (data) => {
          ctx.response.type = 'application/json';
          ctx.response.body = data;
        }
        try {
          await next();
        } catch (e) {
          // 返回错误:
          ctx.response.status = 400;
          ctx.response.type = 'application/json';
          ctx.response.body = {
            code: e.code || 'internal:unknown_error',
            message: e.message || ''
          };
        }
      } else {
        await next();
      }
    };
  }
};

```

这个错误处理的好处在于，不但简化了Controller的错误处理（只需要throw，其他不管），并且，在遇到非APIError的错误时，自动转换错误码为 `internal:unknown_error`。

受益于async/await语法，我们在middleware中可以直接用 `try...catch` 捕获异常。如果是callback模式，就无法用 `try...catch` 捕获，代码结构将混乱得多。

最后，顺便把 `APIError` 这个对象export出去。

开发REST API

我们先根据 `rest-hello` 和 `view-koa` 来创建一个 `rest-hello` 的工程，结构如下：

```

rest-koa/
|
+-- .vscode/
| |
| +- launch.json <-- VSCode 配置文件
|
+-- controllers/
| |
| +- api.js <-- REST API
| |
| +- index.js <-- MVC Controllers
|
+-- products.js <-- 集中处理Product
|
+-- rest.js <-- 支持REST的middleware
|
+-- app.js <-- 使用koa的js
|
+-- controller.js <-- 扫描注册Controller
|
+-- static-files.js <-- 支持静态文件的middleware
|
+-- templating.js <-- 支持Nunjucks的middleware
|
+-- package.json <-- 项目描述文件
|
+-- views/ <-- Nunjucks模板
|
+-- static/ <-- 静态资源文件
|
+-- node_modules/ <-- npm安装的所有依赖包

```

在 `package.json` 中，我们需要如下依赖包：

```

"dependencies": {
  "koa": "2.0.0",
  "koa-bodyparser": "3.2.0",
  "koa-router": "7.0.0"
  "nunjucks": "2.4.2",
  "mime": "1.3.4",
  "mz": "2.4.0"
}

```

运行 `npm install` 安装依赖包。

我们在这个工程中约定了如下规范：

1. REST API的返回值全部是`object`对象，而不是简单的`number`、`boolean`、`null`或者数组；
2. REST API必须使用前缀 `/api/`。

第一条规则实际上是为了方便客户端处理结果。如果返回结果不是`object`，则客户端反序列化后还需要判断类型。以Objective-C为例，可以直接返回 `NSDictionary*`：

```

NSDictionary* dict = [NSJSONSerialization JSONObjectWithData:jsonData options:0 error:&err];

```

如果返回值可能是`number`、`boolean`、`null`或者数组，则客户端的工作量会大大增加。

Service

为了操作`Product`，我们用 `products.js` 封装所有操作，可以把它视为一个`Service`：


```

var id = 0;

function nextId() {
  id++;
  return 'p' + id;
}

function Product(name, manufacturer, price) {
  this.id = nextId();
  this.name = name;
  this.manufacturer = manufacturer;
  this.price = price;
}

var products = [
  new Product('iPhone 7', 'Apple', 6800),
  new Product('ThinkPad T440', 'Lenovo', 5999),
  new Product('LBP2900', 'Canon', 1099)
];

module.exports = {
  getProducts: () => {
    return products;
  },

  getProduct: (id) => {
    var i;
    for (i = 0; i < products.length; i++) {
      if (products[i].id === id) {
        return products[i];
      }
    }
    return null;
  },

  createProduct: (name, manufacturer, price) => {
    var p = new Product(name, manufacturer, price);
    products.push(p);
    return p;
  },

  deleteProduct: (id) => {
    var
      index = -1,
      i;
    for (i = 0; i < products.length; i++) {
      if (products[i].id === id) {
        index = i;
        break;
      }
    }
    if (index >= 0) {
      // remove products[index]:
      return products.splice(index, 1)[0];
    }
    return null;
  }
};

```

变量 `products` 相当于在内存中模拟了数据库，这里是为了简化逻辑。

API

紧接着，我们编写 `api.js`，并放到 `controllers` 目录下：

```
const products = require('../products');

const APIError = require('../rest').APIError;

module.exports = {
  'GET /api/products': async (ctx, next) => {
    ctx.rest({
      products: products.getProducts()
    });
  },

  'POST /api/products': async (ctx, next) => {
    var p = products.createProduct(ctx.request.body.name, ctx.request.body.manufacturer, parseFloat(ctx.request.body.price));
    ctx.rest(p);
  },

  'DELETE /api/products/:id': async (ctx, next) => {
    console.log(`delete product ${ctx.params.id}...`);
    var p = products.deleteProduct(ctx.params.id);
    if (p) {
      ctx.rest(p);
    } else {
      throw new APIError('product:not_found', 'product not found by id.');
```

该API支持GET、POST和DELETE这三个请求。当然，还可以添加更多的API。

编写API时，需要注意：

如果客户端传递了JSON格式的数据（例如，POST请求），则async函数可以通过`ctx.request.body`直接访问已经反序列化的JavaScript对象。这是由`bodyParser()`这个middleware完成的。如果`ctx.request.body`为`undefined`，说明缺少middleware，或者middleware没有正确配置。

如果API路径带有参数，参数必须用`:`表示，例如，`DELETE /api/products/:id`，客户端传递的URL可能就是`/api/products/A001`，参数`id`对应的值就是`A001`，要获得这个参数，我们用`ctx.params.id`。

类似的，如果API路径有多个参数，例如，`/api/products/:pid/reviews/:rid`，则这两个参数分别用`ctx.params.pid`和`ctx.params.rid`获取。

这个功能由koa-router这个middleware提供。

请注意：API路径的参数永远是字符串！

MVC

有了API以后，我们就可以编写MVC，在页面上调用API完成操作。

先在`controllers`目录下创建`index.js`，编写页面入口函数：

```
module.exports = {
  'GET /': async (ctx, next) => {
    ctx.render('index.html');
  }
};
```

然后，我们在`views`目录下创建`index.html`，编写JavaScript代码读取Products：

```
$(function () {
  var vm = new Vue({
    el: '#product-list',
    data: {
      products: []
    }
  });

  $.getJSON('/api/products').done(function (data) {
    vm.products = data.products;
  }).fail(function (jqXHR, textStatus) {
    alert('Error: ' + jqXHR.status);
  });
});
```

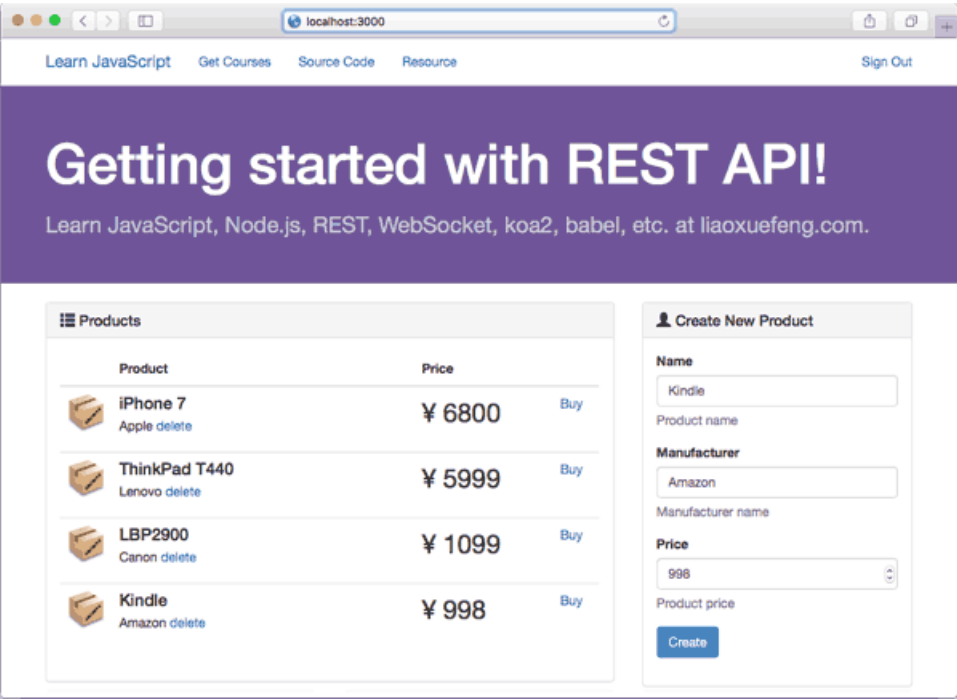
与VM对应的HTML如下：

```
<table id="product-list" class="table table-hover">
  <thead>
    <tr>
      <th style="width:50px"></th>
      <th>Product</th>
      <th style="width:150px">Price</th>
    </tr>
  </thead>
  <tbody>
    <tr v-for="p in products">
      <td>
        
      </td>
      <td>
        <h4 class="media-heading" v-text="p.name"></h4>
        <p><span v-text="p.manufacturer"></span></p>
      </td>
      <td>
        <p style="font-size:2em">¥ <span v-text="p.price"></span></p>
      </td>
    </tr>
  </tbody>
</table>
```

当products变化时，Vue会自动更新表格的内容。

类似的，可以添加创建和删除Product的功能，并且刷新变量products的内容，就可以实时更新Product列表。

最终的页面效果如下：



右侧可以通过[`POST /api/products`]创建新的Product，左侧可以通过[`GET /api/products`]列出所有Product，并且还可以通过[`DELETE /api/products/<id>`]来删除某个Product。

参考源码

rest-koa

MVVM

什么是MVVM? **MVVM**是Model-View-ViewModel的缩写。

要编写可维护的前端代码绝非易事。我们已经用MVC模式通过koa实现了后端数据、模板页面和控制器的分离，但是，对于前端来说，还不够。

这里有童鞋会问，不是讲Node后端开发吗？怎么又回到前端开发了？

对于一个全栈开发工程师来说，懂前端才会开发出更好的后端程序（不懂前端的后端工程师会设计出非常难用的API），懂后端才会开发出更好的前端程序。程序设计的基本思想在前后端都是通用的，两者并无本质的区别。这和“不想当厨子的裁缝不是好司机”是一个道理。

当我们用Node.js有了一整套后端开发模型后，我们对前端开发也会有新的认识。由于前端开发混合了HTML、CSS和JavaScript，而且页面众多，所以，代码的组织和维护难度其实更加复杂，这就是MVVM出现的原因。

在了解MVVM之前，我们先回顾一下前端发展的历史。

在上个世纪的1989年，欧洲核子研究中心的物理学家Tim Berners-Lee发明了超文本标记语言（HyperText Markup Language），简称HTML，并在1993年成为互联网草案。从此，互联网开始迅速商业化，诞生了一大批商业网站。

最早的HTML页面是完全静态的网页，它们是预先编写好的存放在Web服务器上的html文件。浏览器请求某个URL时，Web服务器把对应的html文件扔给浏览器，就可以显示html文件的内容了。

如果要针对不同的用户显示不同的页面，显然不可能给成千上万的用户准备好成千上万的不同的html文件，所以，服务器就需要针对不同的用户，动态生成不同的html文件。一个最直接的想法就是利用C、C++这些编程语言，直接向浏览器输出拼接后的字符串。这种技术被称为CGI: Common Gateway Interface。

很显然，像新浪首页这样的复杂的HTML是不可能通过拼字符串得到的。于是，人们又发现，其实拼字符串的时候，大多数字符串都是HTML片段，是不变的，变化的只有少数和用户相关的数据，所以，又出现了新的创建动态HTML的方式：ASP、JSP和PHP——分别由微软、SUN和开源社区开发。

在ASP中，一个asp文件就是一个HTML，但是，需要替换的变量用特殊的[<%=var%>]标记出来了，再配合循环、条件判断，创建动态HTML就比CGI要容易得多。

但是，一旦浏览器显示了一个HTML页面，要更新页面内容，唯一的方法就是重新向服务器获取一份新的HTML内容。如果浏览器想要自己修改HTML页面的内容，就需要等到1995年年底，JavaScript被引入到浏览器。

有了JavaScript后，浏览器就可以运行JavaScript，然后，对页面进行一些修改。JavaScript还可以通过修改HTML的DOM结构和CSS来实现一些动画效果，而这些功能没法通过服务器完成，必须在浏览器实现。

用JavaScript在浏览器中操作HTML，经历了若干发展阶段：

第一阶段，直接用JavaScript操作DOM节点，使用浏览器提供的原生API：

```
var dom = document.getElementById('name');
dom.innerHTML = 'Homer';
dom.style.color = 'red';
```

第二阶段，由于原生API不好用，还要考虑浏览器兼容性，jQuery横空出世，以简洁的API迅速俘获了前端开发者的芳心：

```
$('#name').text('Homer').css('color', 'red');
```

第三阶段，MVC模式，需要服务器端配合，JavaScript可以在前端修改服务器渲染后的数据。

现在，随着前端页面越来越复杂，用户对于交互性要求也越来越高，想要写出Gmail这样的页面，仅仅用jQuery是远远不够的。MVVM模型应运而生。

MVVM最早由微软提出来，它借鉴了桌面应用程序的MVC思想，在前端页面中，把Model用纯JavaScript对象表示，View负责显示，两者做到了最大限度的分离。

把Model和View关联起来的的就是ViewModel。ViewModel负责把Model的数据同步到View显示出来，还负责把View的修改同步回Model。

ViewModel如何编写？需要用JavaScript编写一个通用的ViewModel，这样，就可以复用整个MVVM模型了。

一个MVVM框架和jQuery操作DOM相比有什么区别？

我们先看用jQuery实现的修改两个DOM节点的例子：

```
<!-- HTML -->
<p>Hello, <span id="name">Bart</span>!</p>
<p>You are <span id="age">12</span>.</p>
```

Hello, **Bart**!

You are **12**.

用jQuery修改name和age节点的内容：

```
'use strict';
----
var name = 'Homer';
var age = 51;

$('#name').text(name);
$('#age').text(age);
----
// 执行代码并观察页面变化
```

如果我们使用MVVM框架来实现同样的功能，我们首先并不关心DOM的结构，而是关心数据如何存储。最简单的数据存储方式是使用JavaScript对象：

```
var person = {
  name: 'Bart',
  age: 12
};
```

我们把变量`person`看作Model，把HTML某些DOM节点看作View，并假定它们之间被关联起来了。

要把显示的name从 `Bart` 改为 `Homer`，把显示的age从 `12` 改为 `51`，我们并不操作DOM，而是直接修改JavaScript对象：

Hello, **Bart**!

You are **12**.

```
'use strict';
----
person.name = 'Homer';
person.age = 51;
----
// 执行代码并观察页面变化
```

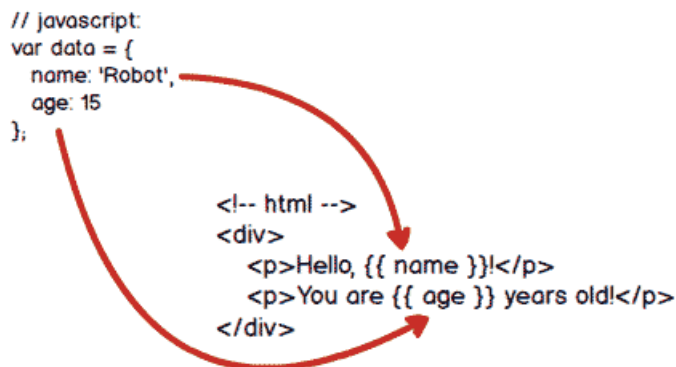
执行上面的代码，我们惊讶地发现，改变JavaScript对象的状态，会导致DOM结构作出对应的变化！这让我们关注点从如何操作DOM变成了如何更新JavaScript对象的状态，而操作JavaScript对象比DOM简单多了！

这就是MVVM的设计思想：关注Model的变化，让MVVM框架去自动更新DOM的状态，从而把开发者从操作DOM的繁琐步骤中解脱出来！

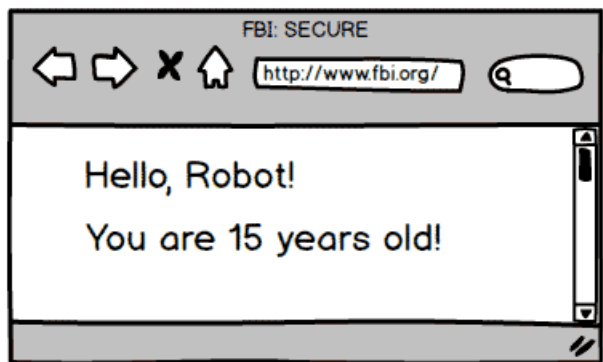
单向绑定

MVVM就是在前端页面上，应用了扩展的MVC模式，我们关心Model的变化，MVVM框架自动把Model的变化映射到DOM结构上，这样，用户看到的页面内容就会随着Model的变化而更新。

例如，我们定义好一个JavaScript对象作为Model，并且把这个Model的两个属性绑定到DOM节点上：



经过MVVM框架的自动转换，浏览器就可以直接显示Model的数据了：



现在问题来了：MVVM框架哪家强？

目前，常用的MVVM框架有：

Angular：Google出品，名气大，但是很难用；

Backbone.js：入门非常困难，因为自身API太多；

Ember：一个大而全的框架，想写个Hello world都很困难。

我们选择MVVM的目标应该是入门容易，安装简单，能直接在页面写JavaScript，需要更复杂的功能时又能扩展支持。

所以，综合考察，最佳选择是尤雨溪大神开发的MVVM框架：[Vue.js](#)

目前，Vue.js的最新版本是2.0，我们会使用2.0版本作为示例。

我们首先创建基于koa的Node.js项目。虽然目前我们只需要在HTML静态页面中编写MVVM，但是很快我们就需要和后端API进行交互，因此，要创建基于koa的项目结构如下：

```
hello-vue/
|
+- .vscode/
| |
| +- launch.json <-- VSCode 配置文件
|
+- app.js <-- koa app
|
+- static-files.js <-- 支持静态文件的koa middleware
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
|
+- static/ <-- 存放静态资源文件
|   |
|   +- css/ <-- 存放bootstrap.css等
|   |
|   +- fonts/ <-- 存放字体文件
|   |
|   +- js/ <-- 存放各种js文件
|   |
|   +- index.html <-- 使用MVVM的静态页面
```

这个Node.js项目的主要目的是作为服务器输出静态网页，因此，`package.json`仅需要如下依赖包：

```
"dependencies": {
  "koa": "2.0.0",
  "mime": "1.3.4",
  "mz": "2.4.0"
}
```

使用 `npm install` 安装好依赖包，然后启动 `app.js`，在 `index.html` 文件中随便写点内容，确保浏览器可以通过 `http://localhost:3000/static/index.html` 访问到该静态文件。

紧接着，我们在 `index.html` 中用Vue实现MVVM的一个简单例子。

安装Vue

安装Vue有很多方法，可以用npm或者webpack。但是我们现在的目标是尽快用起来，所以最简单的方法是直接在HTML代码中像引用jQuery一样引用Vue。可以直接使用CDN的地址，例如：

```
<script src="https://unpkg.com/vue@2.0.1/dist/vue.js"></script>
```

也可以把 `vue.js` 文件下载下来，放到项目的 `/static/js` 文件夹中，使用本地路径：

```
<script src="/static/js/vue.js"></script>
```

这里需要注意，`vue.js` 是未压缩的用于开发的版本，它会在浏览器console中输出很多有用的信息，帮助我们调试代码。当开发完毕，需要真正发布到服务器时，应该使用压缩过的 `vue.min.js`，它会移除所有调试信息，并且文件体积更小。

编写MVM

下一步，我们就可以在HTML页面中编写JavaScript代码了。我们的Model是一个JavaScript对象，它包含两个属性：

```
{
  name: 'Robot',
  age: 15
}
```


而负责显示的是DOM节点可以用 `{{ name }}` 和 `{{ age }}` 来引用Model的属性：

```
<div id="vm">
  <p>Hello, {{ name }}!</p>
  <p>You are {{ age }} years old!</p>
</div>
```

最后一步是用Vue把两者关联起来。**要特别注意的是**，在 `<head>` 内部编写的JavaScript代码，需要用jQuery把MVVM的初始化代码推迟到页面加载完毕后执行，否则，直接在 `<head>` 内执行MVVM代码时，DOM节点尚未被浏览器加载，初始化将失败。正确的写法如下：

```
<html>
<head>

<!-- 引用jQuery -->
<script src="/static/js/jquery.min.js"></script>

<!-- 引用Vue -->
<script src="/static/js/vue.js"></script>

<script>
// 初始化代码：
$(function () {
  var vm = new Vue({
    el: '#vm',
    data: {
      name: 'Robot',
      age: 15
    }
  });
  window.vm = vm;
});
</script>

</head>

<body>

  <div id="vm">
    <p>Hello, {{ name }}!</p>
    <p>You are {{ age }} years old!</p>
  </div>

</body>
</html>
```

我们创建一个VM的核心代码如下：

```
var vm = new Vue({
  el: '#vm',
  data: {
    name: 'Robot',
    age: 15
  }
});
```

其中，`el` 指定了要把Model绑定到哪个DOM根节点上，语法和jQuery类似。这里的 `'#vm'` 对应ID为 `vm` 的一个 `<div>` 节点：

```
<div id="vm">
  ...
</div>
```

在该节点以及该节点内部，就是Vue可以操作的View。Vue可以自动把Model的状态映射到View上，但是**不能**操作View范围之外的其他DOM节点。

然后，`data`属性指定了Model，我们初始化了Model的两个属性`name`和`age`，在View内部的`<p>`节点上，可以直接用`{{ name }}`引用Model的某个属性。

一切正常的话，我们在浏览器中访问`http://localhost:3000/static/index.html`，可以看到页面输出为：

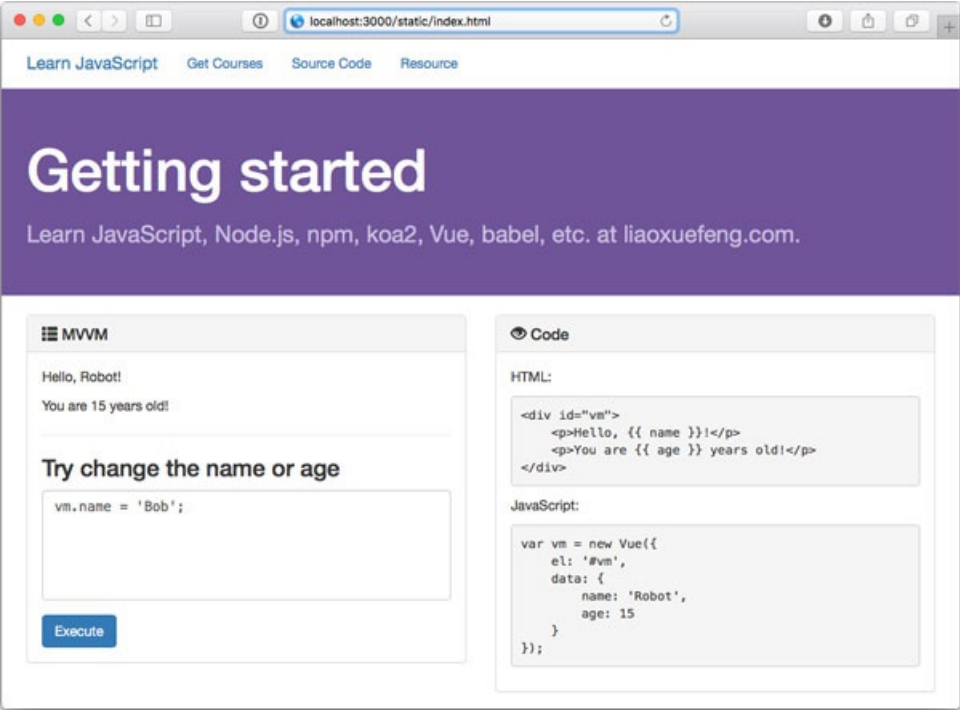
```
Hello, Robot!  
You are 15 years old!
```

如果打开浏览器console，因为我们用代码`window.vm = vm`，把VM变量绑定到了window对象上，所以，可以直接修改VM的Model：

```
window.vm.name = 'Bob'
```

执行上述代码，可以观察到页面立刻发生了变化，原来的`Hello, Robot!`自动变成了`Hello, Bob!`。Vue作为MVVM框架会自动监听Model的任何变化，在Model数据变化时，更新View的显示。这种Model到View的绑定我们称为单向绑定。

经过CSS修饰后的页面如下：



可以在页面直接输入JavaScript代码改变Model，并观察页面变化。

单向绑定

在Vue中，可以直接写`{{ name }}`绑定某个属性。如果属性关联的是对象，还可以用多个`.`引用，例如，`{{ address.zipcode }}`。

另一种单向绑定的方法是使用Vue的指令`v-text`，写法如下：

```
<p>Hello, <span v-text="name"></span>!</p>
```

这种写法是把指令写在HTML节点的属性上，它会被Vue解析，该节点的文本内容会被绑定为Model的指定属性，注意不能再写双花括号`{{ }}`。

参考源码

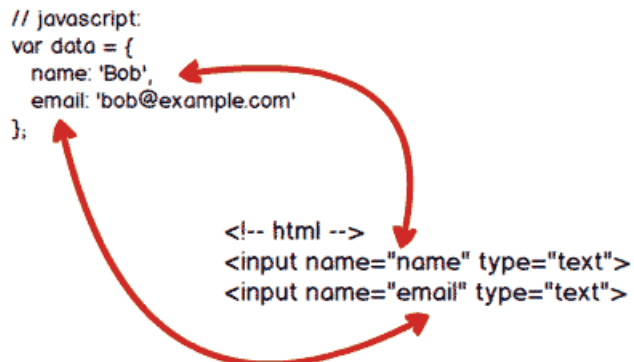
[hello-vue](#)

双向绑定

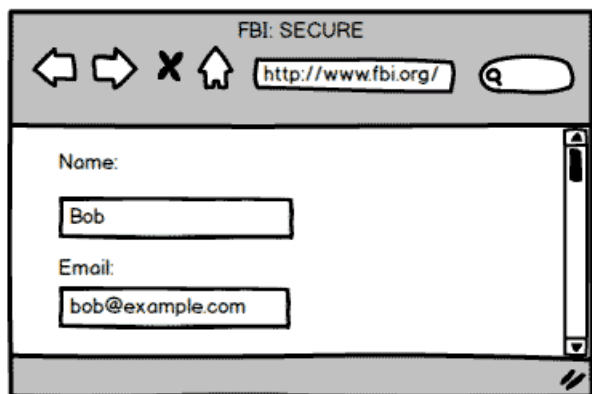
单向绑定非常简单，就是把Model绑定到View，当我们用JavaScript代码更新Model时，View就会自动更新。

有单向绑定，就有双向绑定。如果用户更新了View，Model的数据也自动被更新了，这种情况就是双向绑定。

什么情况下用户可以更新View呢？填写表单就是一个最直接的例子。当用户填写表单时，View的状态就被更新了，如果此时MVVM框架可以自动更新Model的状态，那就相当于我们把Model和View做了双向绑定：



在浏览器中，当用户修改了表单的内容时，我们绑定的Model会自动更新：



在Vue中，使用双向绑定非常容易，我们仍然先创建一个VM实例：

```
$(function () {
  var vm = new Vue({
    el: '#vm',
    data: {
      email: '',
      name: ''
    }
  });
  window.vm = vm;
});
```

然后，编写一个HTML FORM表单，并用 `v-model` 指令把某个 `<input>` 和Model的某个属性作双向绑定：

```
<form id="vm" action="#">
  <p><input v-model="email"></p>
  <p><input v-model="name"></p>
</form>
```

我们可以在表单中输入内容，然后在浏览器console中用 `window.vm.$data` 查看Model的内容，也可以用 `window.vm.name` 查看Model的 `name` 属性，它的值和FORM表单对应的 `<input>` 是一致的。

如果在浏览器console中用JavaScript更新Model，例如，执行`window.vm.name='Bob'`，表单对应的`<input>`内容就会立刻更新。

除了`<input type="text">`可以和字符串类型的属性绑定外，其他类型的`<input>`也可以和相应数据类型绑定：

多个checkbox可以和数组绑定：

```
<label><input type="checkbox" v-model="language" value="zh"> Chinese</label>
<label><input type="checkbox" v-model="language" value="en"> English</label>
<label><input type="checkbox" v-model="language" value="fr"> French</label>
```

对应的Model为：

```
language: ['zh', 'en']
```

单个checkbox可以和boolean类型变量绑定：

```
<input type="checkbox" v-model="subscribe">
```

对应的Model为：

```
subscribe: true; // 根据checkbox是否选中为true/false
```

下拉框`<select>`绑定的是字符串，但是要注意，绑定的是value而非用户看到的文本：

```
<select v-model="city">
  <option value="bj">Beijing</option>
  <option value="sh">Shanghai</option>
  <option value="gz">Guangzhou</option>
</select>
```

对应的Model为：

```
city: 'bj' // 对应option的value
```

双向绑定最大的好处是我们不再需要用jQuery去查询表单的状态，而是直接获得了用JavaScript对象表示的Model。

处理事件

当用户提交表单时，传统的做法是响应`onsubmit`事件，用jQuery获取表单内容，检查输入是否有效，最后提交表单，或者用AJAX提交表单。

现在，获取表单内容已经不需要了，因为双向绑定直接让我们获得了表单内容，并且获得了合适的数据类型。

响应`onsubmit`事件也可以放到VM中。我们在`<form>`元素上使用指令：

```
<form id="vm" v-on:submit.prevent="register">
```

其中，`v-on:submit="register"`指令就会自动监听表单的`submit`事件，并调用`register`方法处理该事件。使用`.prevent`表示阻止事件冒泡，这样，浏览器不再处理`<form>`的`submit`事件。

因为我们指定了事件处理函数是`register`，所以需要创建VM时添加一个`register`函数：

```
var vm = new Vue({
  el: '#vm',
  data: {
    ...
  },
  methods: {
    register: function () {
      // 显示JSON格式的Model:
      alert(JSON.stringify(this.$data));
      // TODO: AJAX POST...
    }
  }
});
```

在 `register()` 函数内部，我们可以用AJAX把JSON格式的Model发送给服务器，就完成了用户注册的功能。

使用CSS修饰后的页面效果如下：

The screenshot shows a web browser at `localhost:3000/static/index.html`. The page is divided into two main sections. On the left, titled "Register Vue Course", is a registration form. It includes fields for "Email address:" (with the value "bob@example.com"), "Password:" (with the placeholder "Password"), "Birth:" (with the value "1990-01-21"), "Gender:" (with radio buttons for "Male", "Female" (selected), and "Keep Secret"), "Language:" (with checkboxes for "Chinese" (selected), "English", and "French"), "City:" (a dropdown menu showing "Beijing"), "Introduce Yourself:" (a text area with the placeholder "Your background, interests, etc."), and a "Subscribe:" section with a checked checkbox for "Send me information of new courses". A "Register" button is at the bottom of the form. On the right, titled "Model", is a section for viewing the current data model. It contains a code block with the following JSON: `{vm: {email: 'bob@example.com', birth: '1990-01-21', gender: 'f'}}`. Below the code block is an "Execute" button.

参考源码

[form-vue](#)

同步DOM结构

除了简单的单向绑定和双向绑定，MVVM还有一个重要的用途，就是让Model和DOM的结构保持同步。

我们用一个TODO的列表作为示例，从用户角度看，一个TODO列表在DOM结构的表现形式就是一组节点：

```
<ol>
  <li>
    <dl>
      <dt>产品评审</dt>
      <dd>新款iPhone上市前评审</dd>
    </dl>
  </li>
  <li>
    <dl>
      <dt>开发计划</dt>
      <dd>与PM确定下一版Android开发计划</dd>
    </dl>
  </li>
  <li>
    <dl>
      <dt>VC会议</dt>
      <dd>敲定C轮5000万美元融资</dd>
    </dl>
  </li>
</ol>
```

而对应的Model可以用JavaScript数组表示：

```
todos: [
  {
    name: '产品评审',
    description: '新款iPhone上市前评审'
  },
  {
    name: '开发计划',
    description: '与PM确定下一版Android开发计划'
  },
  {
    name: 'VC会议',
    description: '敲定C轮5000万美元融资'
  }
]
```

使用MVVM时，当我们更新Model时，DOM结构会随着Model的变化而自动更新。当todos数组增加或删除元素时，相应的DOM节点会增加或者删除节点。

在Vue中，可以使用v-for指令来实现：

```
<ol>
  <li v-for="t in todos">
    <dl>
      <dt>{{ t.name }}</dt>
      <dd>{{ t.description }}</dd>
    </dl>
  </li>
</ol>
```

v-for指令把数组和一组元素绑定了。在元素内部，用循环变量t引用某个属性，例如，{{ t.name }}。这样，我们只关心如何更新Model，不关心如何增删DOM节点，大大简化了整个页面的逻辑。

我们可以在浏览器console中用window.vm.todos[0].name='计划有变'查看View的变化，或者通

过 `window.vm.todos.push({name: '新计划', description: 'blabla...' })` 来增加一个数组元素，从而自动添加一个 `` 元素。

需要注意的是，Vue之所以能够监听Model状态的变化，是因为JavaScript语言本身提供了Proxy或者Object.observe()机制来监听对象状态的变化。但是，对于数组元素的赋值，却没有办法直接监听，因此，如果我们直接对数组元素赋值：

```
vm.todos[0] = {
  name: 'New name',
  description: 'New description'
};
```

会导致Vue无法更新View。

正确的方法是不要对数组元素赋值，而是更新：

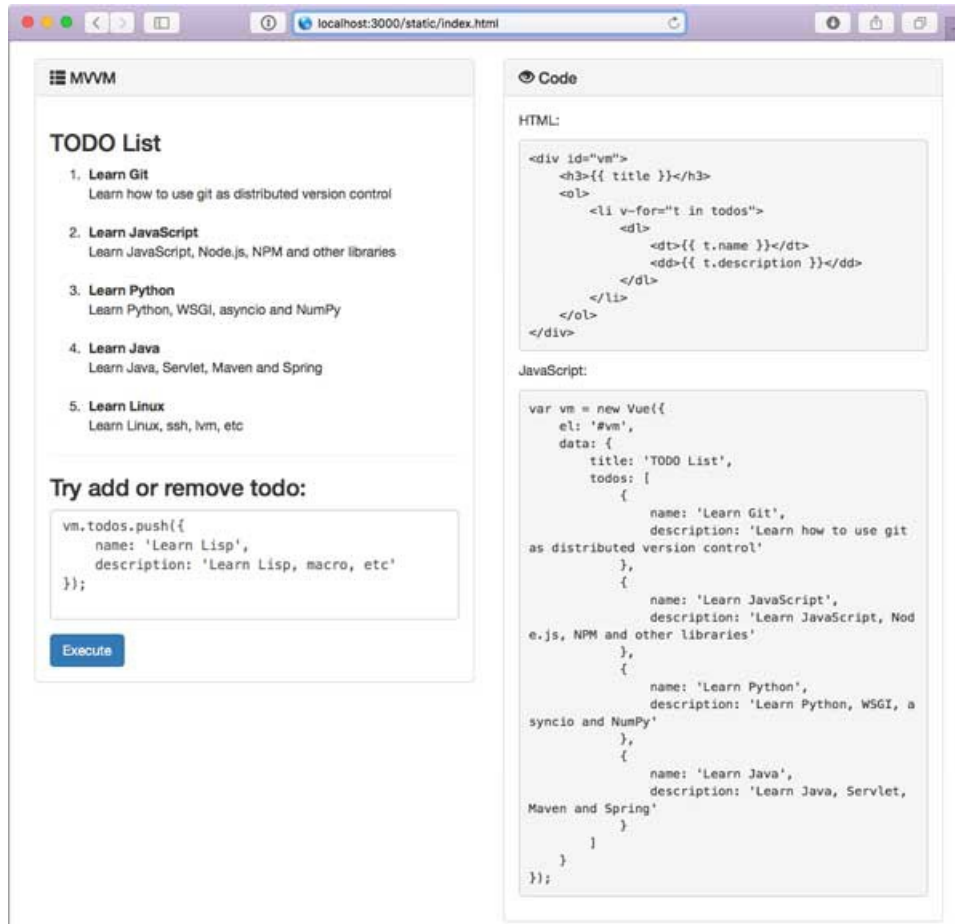
```
vm.todos[0].name = 'New name';
vm.todos[0].description = 'New description';
```

或者，通过 `splice()` 方法，删除某个元素后，再添加一个元素，达到“赋值”的效果：

```
var index = 0;
var newElement = {...};
vm.todos.splice(index, 1, newElement);
```

Vue可以监听数组的 `splice`、`push`、`unshift` 等方法调用，所以，上述代码可以正确更新View。

用CSS修饰后的页面效果如下：



参考源码

集成API

在上一节中，我们用Vue实现了一个简单的TODO应用。通过对Model的更新，DOM结构可以同步更新。

现在，如果要把这个简单的TODO应用变成一个用户能使用的Web应用，我们需要解决几个问题：

1. 用户的TODO数据应该从后台读取；
2. 对TODO的增删改必须同步到服务器后端；
3. 用户在View上必须能够修改TODO。

第1个和第2个问题都是和API相关的。只要我们实现了合适的API接口，就可以在Mvvm内部更新Model的同时，通过API把数据更新反映到服务器端，这样，用户数据就保存到了服务器端，下次打开页面时就可以读取TODO列表。

我们在 `vue-todo` 的基础上创建 `vue-todo-2` 项目，结构如下：

```
vue-todo-2/
|
+- .vscode/
| |
| +- launch.json <-- VSCode 配置文件
|
+- app.js <-- koa app
|
+- static-files.js <-- 支持静态文件的koa middleware
|
+- controller.js <-- 支持路由的koa middleware
|
+- rest.js <-- 支持REST的koa middleware
|
+- package.json <-- 项目描述文件
|
+- node_modules/ <-- npm安装的所有依赖包
|
+- controllers/ <-- 存放Controller
| |
| +- api.js <-- REST API
|
+- static/ <-- 存放静态资源文件
|
| +- css/ <-- 存放bootstrap.css等
|
| +- fonts/ <-- 存放字体文件
|
| +- js/ <-- 存放各种js文件
|
+- index.html <-- 使用Mvvm的静态页面
```

在 `api.js` 文件中，我们用数组在服务器端模拟一个数据库，然后实现以下4个API：

- GET /api/todos: 返回所有TODO列表；
- POST /api/todos: 创建一个新的TODO，并返回创建后的对象；
- PUT /api/todos/:id: 更新一个TODO，并返回更新后的对象；
- DELETE /api/todos/:id: 删除一个TODO。

和上一节的TODO数据结构相比，我们需要增加一个 `id` 属性，来唯一标识一个TODO。

准备好API后，在Vue中，我们如何把Model的更新同步到服务器端？

有两个方法，一是直接用jQuery的AJAX调用REST API，不过这种方式比较麻烦。

第二个方法是使用 `vue-resource` 这个针对Vue的扩展，它可以给VM对象加上一个 `$resource` 属性，通过 `$resource` 来方便地操作API。

使用 `vue-resource` 只需要在导入 `vue.js` 后，加一行 `<script>` 导入 `vue-resource.min.js` 文件即可。可以直接使用CDN的地址：

```
<script src="https://cdn.jsdelivr.net/vue.resource/1.0.3/vue-resource.min.js"></script>
```

我们给VM增加一个 `init()` 方法，读取TODO列表：

```
var vm = new Vue({
  el: '#vm',
  data: {
    title: 'TODO List',
    todos: []
  },
  created: function () {
    this.init();
  },
  methods: {
    init: function () {
      var that = this;
      that.$resource('/api/todos').get().then(function (resp) {
        // 调用API成功时调用json()异步返回结果：
        resp.json().then(function (result) {
          // 更新VM的todos：
          that.todos = result.todos;
        });
      }, function (resp) {
        // 调用API失败：
        alert('error');
      });
    }
  }
});
```

注意到创建VM时，`created` 指定了当VM初始化成功后的回调函数，这样，`init()` 方法会被自动调用。

类似的，对于添加、修改、删除的操作，我们也需要往VM中添加对应的函数。以添加为例：

```
var vm = new Vue({
  ...
  methods: {
    ...
    create: function (todo) {
      var that = this;
      that.$resource('/api/todos').save(todo).then(function (resp) {
        resp.json().then(function (result) {
          that.todos.push(result);
        });
      }, showError);
    },
    update: function (todo, prop, e) {
      ...
    },
    remove: function (todo) {
      ...
    }
  }
});
```

添加操作需要一个额外的表单，我们可以创建另一个VM对象 `vmAdd` 来对表单作双向绑定，然后，在提交表单的事件中调用 `vm` 对象的 `create` 方法：

```
var vmAdd = new Vue({
  el: '#vmAdd',
  data: {
    name: '',
    description: ''
  },
  methods: {
    submit: function () {
      vm.create(this.$data);
    }
  }
});
```

`vmAdd` 和FORM表单绑定:

```
<form id="vmAdd" action="#0" v-on:submit.prevent="submit">
  <p><input type="text" v-model="name"></p>
  <p><input type="text" v-model="description"></p>
  <p><button type="submit">Add</button></p>
</form>
```

最后, 把 `vm` 绑定到对应的DOM:

```
<div id="vm">
  <h3>{{ title }}</h3>
  <ol>
    <li v-for="t in todos">
      <dl>
        <dt contenteditable="true" v-on:blur="update(t, 'name', $event)">{{ t.name }}</dt>
        <dd contenteditable="true" v-on:blur="update(t, 'description', $event)">{{ t.description }}</dd>
        <dd><a href="#0" v-on:click="remove(t)">Delete</a></dd>
      </dl>
    </li>
  </ol>
</div>
```

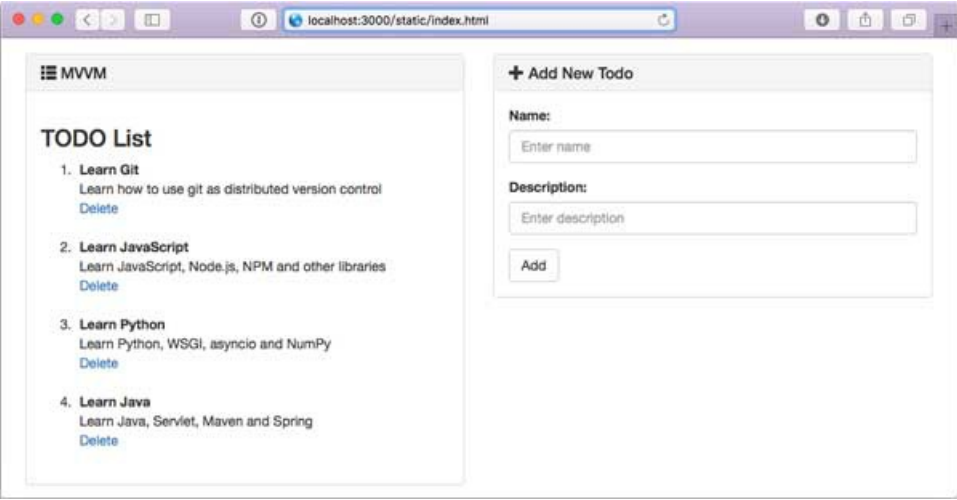
这里我们用 `contenteditable="true"` 让DOM节点变成可编辑的, 用 `v-on:blur="update(t, 'name', $event)"` 在编辑结束时调用 `update()` 方法并传入参数, 特殊变量 `$event` 表示DOM事件本身。

删除TODO是通过对 `<a>` 节点绑定 `v-on:click` 事件并调用 `remove()` 方法实现的。

如果一切无误, 我们就可以先启动服务器, 然后在浏览器中访问 `http://localhost:3000/static/index.html`, 对TODO进行增删改等操作, 操作结果会保存在服务器端。

通过Vue和vue-resource插件, 我们用简单的几十行代码就实现了一个真正可用的TODO应用。

使用CSS修饰后的页面效果如下:



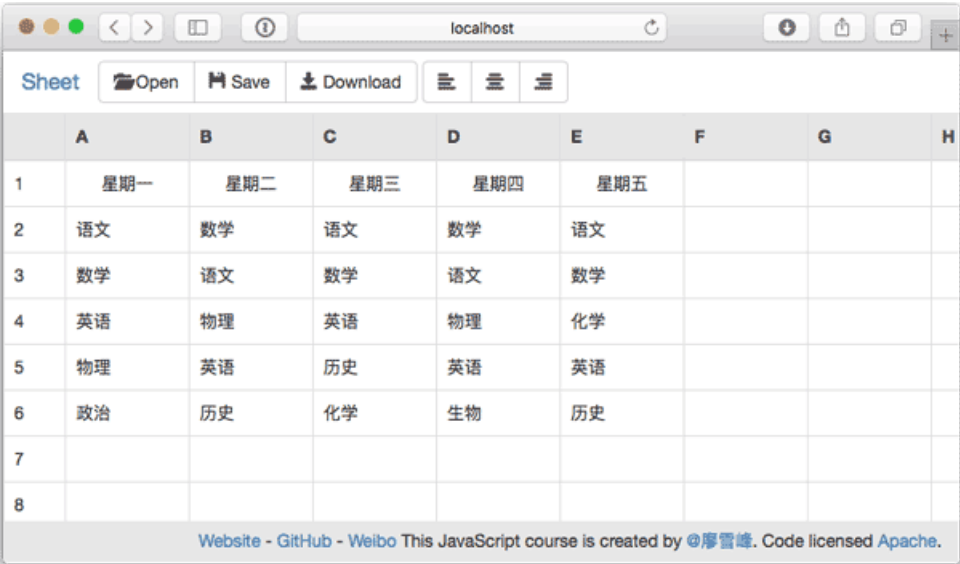
参考源码

[vue-todo-2](#)

在线电子表格

利用MVVM，很多非常复杂的前端页面编写起来就非常容易了。这得益于我们把注意力放在Model的结构上，而不怎么关心DOM的操作。

本节我们演示如何利用Vue快速创建一个在线电子表格：



首先，我们定义Model的结构，它的主要数据就是一个二维数组，每个单元格用一个JavaScript对象表示：

```

data: {
  title: 'New Sheet',
  header: [ // 对应首行 A, B, C...
    { row: 0, col: 0, text: '' },
    { row: 0, col: 1, text: 'A' },
    { row: 0, col: 2, text: 'B' },
    { row: 0, col: 3, text: 'C' },
    ...
    { row: 0, col: 10, text: 'J' }
  ],
  rows: [
    [
      { row: 1, col: 0, text: '1' },
      { row: 1, col: 1, text: '' },
      { row: 1, col: 2, text: '' },
      ...
      { row: 1, col: 10, text: '' },
    ],
    [
      { row: 2, col: 0, text: '2' },
      { row: 2, col: 1, text: '' },
      { row: 2, col: 2, text: '' },
      ...
      { row: 2, col: 10, text: '' },
    ],
    ...
    [
      { row: 10, col: 0, text: '10' },
      { row: 10, col: 1, text: '' },
      { row: 10, col: 2, text: '' },
      ...
      { row: 10, col: 10, text: '' },
    ]
  ],
  selectedRowIndex: 0, // 当前活动单元格的row
  selectedColIndex: 0 // 当前活动单元格的col
}

```

紧接着，我们就可以把Model的结构映射到一个 `<table>` 上：

```

<table id="sheet">
  <thead>
    <tr>
      <th v-for="cell in header" v-text="cell.text"></th>
    </tr>
  </thead>
  <tbody>
    <tr v-for="tr in rows">
      <td v-for="cell in tr" v-text="cell.text"></td>
    </tr>
  </tbody>
</table>

```

现在，用Vue把Model和View关联起来，这个电子表格的原型已经可以运行了！

下一步，我们想在单元格内输入一些文本，怎么办？

因为不是所有单元格都可以被编辑，首行和首列不行。首行对应的是 `<th>`，默认是不可编辑的，首列对应的是第一列的 `<td>`，所以，需要判断某个 `<td>` 是否可编辑，我们用 `v-bind` 指令给某个DOM元素绑定对应的HTML属性：

```

<td v-for="cell in tr" v-bind:contenteditable="cell.contentEditable" v-text="cell.text"></td>

```

在Model中给每个单元格对象加上 `contentEditable` 属性，就可以决定哪些单元格可编辑。

最后，给 `<td>` 绑定 `click` 事件，记录当前活动单元格的 `row` 和 `col`，再绑定 `blur` 事件，在单元格内容编辑结束后更新 `Model`：

```
<td v-for="cell in tr" v-on:click="focus(cell)" v-on:blur="change" ...></td>
```

对应的两个方法要添加到 `VM` 中：

```
var vm = new Vue({
  ...
  methods: {
    focus: function (cell) {
      this.selectedRowIndex = cell.row;
      this.selectedColIndex = cell.col;
    },
    change: function (e) {
      // change事件传入的e是DOM事件
      var
        rowIndex = this.selectedRowIndex,
        colIndex = this.selectedColIndex,
        text;
      if (rowIndex > 0 && colIndex > 0) {
        text = e.target.innerText; // 获取td的innerText
        this.rows[rowIndex - 1][colIndex].text = text;
      }
    }
  }
});
```

现在，单元格已经可以编辑，并且用户的输入会自动更新到 `Model` 中。

如果我们要给单元格的文本添加格式，例如，左对齐或右对齐，可以给 `Model` 对应的对象添加一个 `align` 属性，然后用 `v-bind:style` 绑定到 `<td>` 上：

```
<td v-for="cell in tr" ... v-bind:style="{ textAlign: cell.align }"></td>
```

然后，创建工具栏，给左对齐、居中对齐和右对齐按钮编写 `click` 事件代码，调用 `setAlign()` 函数：

```
function setAlign(align) {
  var
    rowIndex = vm.selectedRowIndex,
    colIndex = vm.selectedColIndex,
    row, cell;
  if (rowIndex > 0 && colIndex > 0) {
    row = vm.rows[rowIndex - 1];
    cell = row[colIndex];
    cell.align = align;
  }
}

// 给按钮绑定事件：
$('#cmd-left').click(function () { setAlign('left'); });
$('#cmd-center').click(function () { setAlign('center'); });
$('#cmd-right').click(function () { setAlign('right'); });
```

现在，点击某个单元格，再点击右对齐按钮，单元格文本就变成右对齐了。

类似的，可以继续添加其他样式，例如字体、字号等。

MVVM的适用范围

从几个例子我们可以看到，**MVVM**最大的优势是编写前端逻辑非常复杂的页面，尤其是需要大量 `DOM` 操作的逻辑，利用 **MVVM** 可以极大地简化前端页面的逻辑。

但是MVVM不是万能的，它的目的是为了解决复杂的前端逻辑。对于以展示逻辑为主的页面，例如，新闻，博客、文档等，**不能**使用MVVM展示数据，因为这些页面需要被搜索引擎索引，而搜索引擎无法获取使用MVVM并通过API加载的数据。

所以，需要SEO（Search Engine Optimization）的页面，不能使用MVVM展示数据。不需要SEO的页面，如果前端逻辑复杂，就适合使用MVVM展示数据，例如，工具类页面，复杂的表单页面，用户登录后才能操作的页面等等。

参考源码

[mini-excel](#)

自动化工具

即将推出，请耐心等待。等待不耐烦的，请关注微博[@廖雪峰](#)

React

即将推出，请耐心等待。等待不耐烦的，请关注微博[@廖雪峰](#)

期末总结

即将推出，请耐心等待。等待不耐烦的，请关注微博[@廖雪峰](#)