# 05 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:
console.log('window inner size: ' + window.innerWidth + '
x ' + window.innerHeight);
```

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

## navigator

`navigator`对象表示浏览器的信息，最常用的属性包括：

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

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

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

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

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

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

## 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';
console.log('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('/'); // 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`xxx`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;

s = '提供的饮料有:';
for (i=0; i<drinks.length; i++) {
    s = s + drinks[i].innerHTML + ',';
}
console.log(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.firstElementChild;
var last = test.lastElementChild;
```

第二种方法是使用`querySelector()`和`querySelectorAll()`，需要了解selector语法，然后使用条件来获取节点，更加方便：

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

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

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

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' || arr[2].innerText !==
'Swift') {
    console.log('选择Python,Ruby,Swift失败!');
} else if (!haskell || haskell.innerText !== 'Haskell') {
    console.log('选择Haskell失败!');
} else {
    console.log('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 === '#f00' || js.style.color ===
'rgb(255, 0, 0)')) {
            console.log('测试通过!');
        } else {
            console.log('CSS样式测试失败!');
        }
    } else {
        console.log('文本测试失败!');
    }
} else {
    console.log('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`JavaScript`添加到``的最后一项：

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

现在，HTML结构变成了这样：

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

因为我们插入的`js`节点已经存在于当前的文档树，因此这个节点首先会从原先的位置删除，再插入到新的位置。

更多的时候我们会从零创建一个新的节点，然后插入到指定位置：

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

这样我们就动态添加了一个新的节点：

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

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

```
var d = document.createElement('style');
d.setAttribute('type', 'text/css');
d.innerHTML = 'p { color: red }';
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()) {
            console.log('测试通过!');
        }
        else {
            console.log('测试失败: ' + arr.toString());
        }
    }
    else {
        console.log('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`First`节点被删除后，`parent.children`的节点数量已经从2变为了1，索引`[1]`已经不存在了。

因此，删除多个节点时，要注意`children`属性时刻都在变化。

## 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()) {
            console.log('测试通过!');
        }
        else {
            console.log('测试失败: ' + arr.toString());
        }
    }
    else {
        console.log('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``，用于输入文本；
- 口令框，对应的``，用于输入口令；
- 单选框，对应的``，用于选择一项；
- 复选框，对应的``，用于选择多项；
- 下拉框，对应的``，用于选择一项；
- 隐藏文本，对应的``，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`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
```

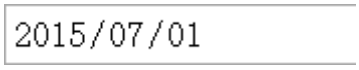
对于单选框和复选框，设置`checked`为`true`或`false`即可。

## HTML5控件

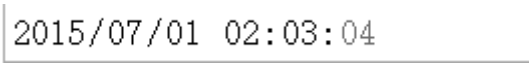
HTML5新增了大量标准控件，常用的包括`date`、`datetime`、`datetime-local`、`color`等，它们都使用``标签：

html

```
<input type="date" value="2015-07-01">
```




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




```
<input type="color" value="#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`元素的`submit()`方法提交一个表单，例如，响应一个`的`click`事件，在JavaScript代码中提交表单：

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

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

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

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

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

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

在检查和修改时，要充分利用来传递数据。

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``：

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

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

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

要想不改变用户的输入，可以利用``实现：

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

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

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

## 练习

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

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

```
<!-- 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>
```


```
'use strict';
var checkRegisterForm = function () {
    // TODO:
    return false;
}
// 测试:
;(function () {
    window.testFormHandler = checkRegisterForm;
    var form = document.getElementById('test-register');
    if (form.dispatchEvent) {
        var event = new Event('submit', {
            bubbles: true,
            cancelable: true
        });
        form.dispatchEvent(event);
    } else {
        form.fireEvent('onsubmit');
    }
})();
```

## 操作文件

在HTML表单中，可以上传文件的唯一控件就是``。

*注意*：当一个表单包含`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来选择本地文件，用JavaScript对的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：

待上传文件：

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

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

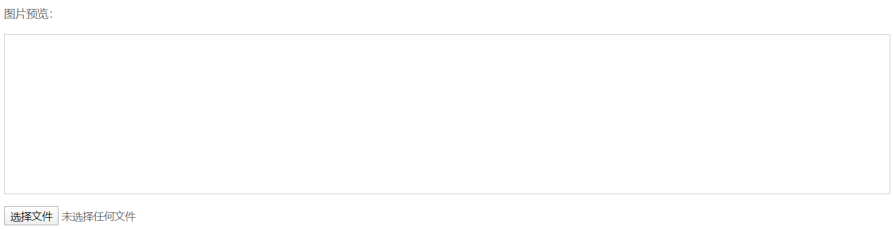
## File API

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

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

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

下面的例子演示了如何读取用户选取的图片文件，并在一个``中预览图像：




图片预览：

```
var
    fileInput = document.getElementById('test-image-
file'),
    info = document.getElementById('test-file-info'),
    preview = document.getElementById('test-image-
preview');
// 监听change事件:
fileInput.addEventListener('change', function () {
    // 清除背景图片:
    preview.style.backgroundImage = '';
    // 检查文件是否选择:
    if (!fileInput.value) {
        info.innerHTML = '没有选择文件';
        return;
    }
    // 获取File引用:
    var file = fileInput.files[0];
```

```
    // 获取File信息:
    info.innerHTML = '文件: ' + file.name + '<br>' +
                     '大小: ' + file.size + '<br>' +
                     '修改: ' + file.lastModifiedDate;
    if (file.type !== 'image/jpeg' && file.type !==
'image/png' && file.type !== 'image/gif') {
        alert('不是有效的图片文件!');
        return;
    }
    // 读取文件:
    var reader = new FileReader();
    reader.onload = function(e) {
        var
            data = e.target.result; //
'...(base64编码)...'

        preview.style.backgroundImage = 'url(' + data +
')';
    };
    // 以DataURL的形式读取文件:
    reader.readAsDataURL(file);
});
```

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

## 回调

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

你可能会问，单线程模式执行的JavaScript，如何处理多任务？

在JavaScript中，执行多任务实际上都是异步调用，比如上面的代码：

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

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

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

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

## 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...');
```


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``节点，相当于动态读取外域的JavaScript资源，最后就等着接收回调了。

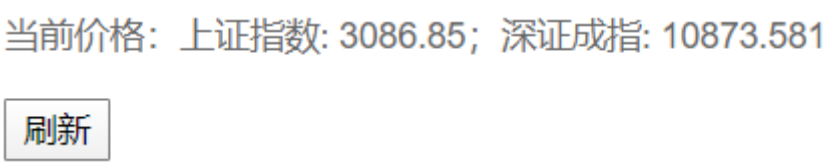
以163的股票查询URL为例，对于URL：http://api.money.126.net/data/feed/0000001,1399001?callback=refreshPrice，你将得到如下返回：

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

因此我们需要首先在页面中准备好回调函数：

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

当前价格：




最后用`getPrice()`函数触发：

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

就完成了跨域加载数据。

## CORS

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

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

了解CORS前，我们先搞明白概念：

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

用一个图来表示就是：


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

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

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

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

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

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

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

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

服务器必须响应并明确指出允许的Method：

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

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

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

需要深入了解CORS的童鞋请移步W3C文档。

## 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.ifSuccess(success)
    .ifFail(fail);
```

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

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

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

```
'use strict';

new Promise(function () {});
// 直接运行测试:
console.log('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：

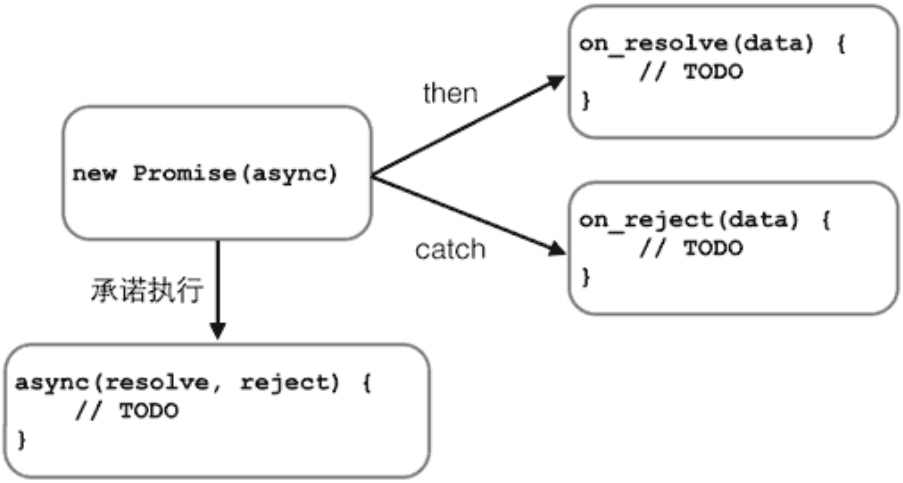
```
new Promise(test).then(function (result) {
    console.log('成功: ' + result);
}).catch(function (reason) {
    console.log('失败: ' + reason);
});
```

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.len
gth - 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.len
gth - 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);
    });
}
```

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`内部添加一些说明性HTML代码，如果浏览器支持Canvas，它将忽略`内部的HTML，如果浏览器不支持Canvas，它将显示``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" heigth="100">
    <p>你的浏览器不支持Canvas</p>
</canvas>
```



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

```
你的浏览器支持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');
```

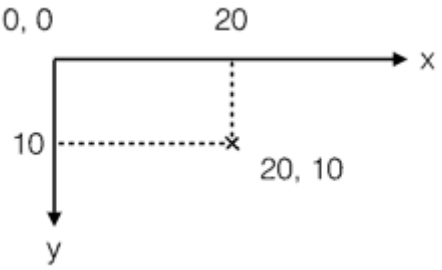
如果需要绘制3D怎么办？HTML5还有一个WebGL规范，允许在Canvas中绘制3D图形：

```
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":49
11.57,"low":4797.55,"vol":62374809900,"change":1.69}
    ctx.clearRect(0, 0, width, height);
    ctx.fillText('Test Canvas', 10, 10);
};

// 加载最近30个交易日的K线图数据:
var js = document.createElement('script');
js.src =
'http://img1.money.126.net/data/hs/kline/day/history/2015/
0000001.json?callback=loadStockData&t=' + Date.now();
document.getElementsByTagName('head')[0].appendChild(js);
```


Test Canvas