《JavaScript 闯关记》之 BOM

javascript 劼哥stone 11月7日发布

ECMAScript 是 JavaScript 的核心，但如果要在 Web 中使用 JavaScript，那么 BOM（浏览器对象模型）则无疑才是真正的核心。BOM 提供了很多对象，用于访问浏览器的功能，这些功能与任何网页内容无关。多年来，缺少事实上的规范导致 BOM 有很多问题，因为浏览器提供商会按照各自的想法随意去扩展它。W3C 为了把浏览器中 JavaScript 最基本的部分标准化，已经将 BOM 的主要方面纳入了 HTML5 的规范中。

`window` 对象

BOM 的核心对象是 `window`，它表示浏览器的一个实例。在浏览器中，`window` 对象有双重角色，它既是通过 JavaScript 访问浏览器窗口的一个接口，又是 ECMAScript 规定的 `Global` 对象。这意味着在网页中定义的任何一个对象、变量和函数，都以 `window` 作为其 `Global` 对象，因此有权访问 `isNaN()`、`isFinite()`、`parseInt()`、`parseFloat()` 等方法。

全局作用域

由于 `window` 对象同时扮演着 ECMAScript 中 `Global` 对象的角色，因此所有在全局作用域中声明的变量、函数都会变成 `window` 对象的属性和方法。来看下面的例子。

```
var age = 29;
function sayAge(){
    console.log(this.age);
}

console.log(window.age);    // 29
sayAge();                   // 29
window.sayAge();            // 29
```

抛开全局变量会成为 `window` 对象的属性不谈，定义全局变量与在 `window` 对象上直接定义属性还是有一点差别：全局变量不能通过 `delete` 运算符删除，而直接在 `window` 对象上的定义的属性可以。例如：

```
var age = 29;
window.color = "red";

// 在 IE < 9 时抛出错误，在其他所有浏览器中都返回 false
delete window.age;

// 在 IE < 9 时抛出错误，在其他所有浏览器中都返回 true
delete window.color;        // return true

console.log(window.age);    // 29
console.log(window.color);  // undefined
```

使用 `var` 语句添加的 `window` 属性有一个名为 `Configurable` 的特性，这个特性的值被默认设置为 `false`，因此这样定义的属性不可以通过 `delete` 运算符删除。IE8 及更早版本在遇到使用 `delete` 删除 `window` 属性的语句时，不管该属性最初是如何创建的，都会抛出错误，以示警告。IE9 及更高版本不会抛出错误。

另外，还要记住一件事：尝试访问未声明的变量会抛出错误，但是通过查询 `window` 对象，可以知道某个可能未声明的变量是否存在。例如：

```
// 这里会抛出错误，因为 oldValue 未定义
var newValue = oldValue;

// 这里不会抛出错误，因为这是一次属性查询
// newValue 的值是 undefined
var newValue = window.oldValue;
```

窗口关系及框架

如果页面中包含框架，则每个框架都拥有自己的 `window` 对象，并且保存在 `frames` 集合中。在 `frames` 集合中，可以通过数值索引（从0开始，从左至右，从上到下）或者框架名称来访问相应的 `window` 对象。每个 `window` 对象都有一个 `name` 属性，其中包含框架的名称。下面是一个包含框架的页面：

```
<html>
    <head>
        <title>Frameset Example</title>
    </head>
    <frameset rows="160,*">
        <frame src="frame.htm" name="topFrame">
        <frameset cols="50%,50%">
            <frame src="anotherframe.htm" name="leftFrame">
            <frame src="yetanotherframe.htm" name="rightFrame">
        </frameset>
    </frameset>
</html>
```

对这个例子而言，可以通过 `window.frames[0]` 或者 `window.frames["topFrame"]` 来引用上方的框架。不过最好使用 `top` 而非 `window` 来引用这些框架（例如 `top.frames[0]`），因为 `top` 对象始终指向最高（最外）层的框架，也就是浏览器窗口。使用它可以确保在一个框架中正确地访问另一个框架。因为对于在一个框架中编写的任何代码来说，其中的 `window` 对象指向的都是那个框架的特定实例，而非最高层的框架。

与 `top` 相对的另一个 `window` 对象是 `parent`。顾名思义，`parent`（父）对象始终指向当前框架的直接上层框架。在某些情况下，`parent` 有可能等于 `top`；但在没有框架的情况下，`parent` 一定等于 `top`（此时它们都等于 `window`）。

与框架有关的最后一个对象是 `self`，它始终指向 `window`；实际上，`self` 和 `window` 对象可以互换使用。引入 `self` 对象的目的只是为了与 `top` 和 `parent` 对象对应起来，因此它不格外包含其他值。

所有这些对象都是 `window` 对象的属性，可以通过 `window.parent`、`window.top` 等形式来访问。同时，这也意味着可以将不同层次的 `window` 对象连缀起来，例如 `window.parent.parent.frames[0]`。

在使用框架的情况下，浏览器中会存在多个 `Global` 对象。在每个框架中定义的全局变量会自动成为框架中 `window` 对象的属性。由于每个 `window` 对象都包含原生类型的构造函数，因此每个框架都有一套自己的构造函数，这些构造函数一一对应，但并不相等。例如，`top.Object` 并不等于 `top.frames[0].Object`。这个问题会影响到对跨框架传递的对象使用 `instanceof` 运算符。

导航和打开窗口

使用 `window.open()` 方法既可以导航到一个特定的 URL，也可以打开一个新的浏览器窗口。这个方法可以接收4个参数：要加载的URL、窗口目标、一个特性字符串以及一个表示新页面是否取代浏览器历史记录中当前加载页面的布尔值。通常只须传递第一个参数，最后一个参数只在不打开新窗口的情况下使用。

如果为 `window.open()` 传递了第二个参数，而且该参数是已有窗口或框架的名称，那么就会在具有该名称的窗口或框架中加载第一个参数指定的URL。看下面的例子。

```
// 等同于 <a href="http://shijiajie.com" target="newWindow"></a>
window.open("http://shijiajie.com/", "newWindow");
```

弹出窗口

如果给 `window.open()` 传递的第二个参数并不是一个已经存在的窗口或框架，那么该方法就会根据在第三个参数位置上传入的字符串创建一个新窗口或新标签页。如果没有传入第三个参数，那么就会打开一个带有全部默认设置（工具栏、地址栏和状态栏等）的新浏览器窗口（或者打开一个新标签页）。在不打开新窗口的情况下，会忽略第三个参数。

第三个参数是一个逗号分隔的设置字符串，表示在新窗口中都显示哪些特性。下表列出了可以出现在这个字符串中的设置选项。

设置	值	说明
fullscreen	yes或no	表示浏览器窗口是否最大化。仅限IE
height	数值	表示新窗口的高度。不能小于100
left	数值	表示新窗口的左坐标。不能是负值
location	yes或	表示是否在浏览器窗口中显示地址栏。不同浏览器的默认值不同。如果设置为no，地址栏可能会隐藏，也可能会被禁用（取

	no	决于浏览器）
menubar	yes或no	表示是否在浏览器窗口中显示菜单栏。默认值为no
resizable	yes或no	表示是否可以通过拖动浏览器窗口的边框改变其大小。默认值为no
scrollbars	yes或no	表示如果内容在视口中显示不下，是否允许滚动。默认值为no
status	yes或no	表示是否在浏览器窗口中显示状态栏。默认值为no
toolbar	yes或no	表示是否在浏览器窗口中显示工具栏。默认值为no
top	数值	表示新窗口的上坐标。不能是负值
width	数值	表示新窗口的宽度。不能小于100

这行代码会打开一个新的可以调整大小的窗口，窗口初始大小为400×400像素，并且距屏幕上沿和左边各10像素。

```
window.open("http://shijiajie.com/","newWindow",
    "height=400,width=400,top=10,left=10,resizable=yes");
```

`window.open()` 方法会返回一个指向新窗口的引用。引用的对象与其他 `window` 对象大致相似，但我们可以对其进行更多控制。例如，有些浏览器在默认情况下可能不允许我们针对主浏览器窗口调整大小或移动位置，但却允许我们针对通过window.open()创建的窗口调整大小或移动位置。通过这个返回的对象，可以像操作其他窗口一样操作新打开的窗口，如下所示。

```
var win = window.open("http://shijiajie.com/","newWindow",
    "height=400,width=400,top=10,left=10,resizable=yes");

// 调整大小
win.resizeTo(500,500);

// 移动位置
win.moveTo(100,100);

// 关闭窗口
win.close();
```

但是，`close()` 方法仅适用于通过 `window.open()` 打开的弹出窗口。对于浏览器的主窗口，如果没有得到用户的允许是不能关闭它的。

新创建的 `window` 对象有一个 `opener` 属性，其中保存着打开它的原始窗口对象。这个属性只在弹出窗口中的最外层 `window` 对象（top）中有定义，而且指向调用 `window.open()` 的窗口或框架。例如：

```
var win = window.open("http://shijiajie.com/","newWindow",
    "height=400,width=400,top=10,left=10,resizable=yes");

console.log(win.opener === window);   // true
```

虽然弹出窗口中有一个指针指向打开它的原始窗口，但原始窗口中并没有这样的指针指向弹出窗口。窗口并不跟踪记录它们打开的弹出窗口，因此我们只能在必要的时候自己来手动实现跟踪。

弹出窗口屏蔽程序

曾经有一段时间，广告商在网上使用弹出窗口达到了肆无忌惮的程度。他们经常把弹出窗口打扮成系统对话框的模样，引诱用户去点击其中的广告。由于看起来像是系统对话框，一般用户很难分辨是真是假。为了解决这个问题，大多数浏览器内置有弹出窗口屏蔽程序，将绝大多数用户不想看到弹出窗口屏蔽掉。

于是，在弹出窗口被屏蔽时，就应该考虑两种可能性。如果是浏览器内置的屏蔽程序阻止的弹出窗口，那么 `window.open()` 很可能会返回 `null`，如果是浏览器扩展或其他程序阻止的弹出窗口，那么 `window.open()` 通常会抛出一个错误。因此，要想准确地检测出弹出窗口是否被屏蔽，必须在检测返回值的同时，将对 `window.open()` 的调用封装在一个 `try-catch` 块中，如下所示。

```
var blocked = false;

try {
    var win = window.open("http://shijiajie.com", "_blank");
    if (win == null){
        blocked = true;
    }
} catch (ex){
    blocked = true;
}
if (blocked){
    console.log("The popup was blocked!");
}
```

间歇调用和超时调用

JavaScript 是单线程语言，但它允许通过设置超时值和间歇时间值来调度代码在特定的时刻执行。前者是在指定的时间过后执行代码，而后者则是每隔指定的时间就执行一次代码。

超时调用需要使用 `window` 对象的 `setTimeout()` 方法，它接受两个参数：要执行的代码和以毫秒表示的时间（即在执行代码前需要等待多少毫秒）。其中，第一个参数可以是一个包含 JavaScript 代码的字符串（就和在 `eval()` 函数中使用的字符串一样），也可以是一个函数。例如，下面对 `setTimeout()` 的两次调用都会在一秒钟后显示一个警告框。

```
// 不建议传递字符串
setTimeout("console.log('Hello world!') ", 1000);

// 推荐的调用方式
setTimeout(function() {
    console.log("Hello world!");
}, 1000);
```

虽然这两种调用方式都没有问题，但由于传递字符串可能导致性能损失，因此不建议以字符串作为第一个参数。

第二个参数是一个表示等待多长时间的毫秒数，但经过该时间后指定的代码不一定会执行。JavaScript 是一个单线程序的解释器，因此一定时间内只能执行一段代码。为了控制要执行的代码，就有一个 JavaScript 任务队列。这些任务会按照将它们添加到队列的顺序执行。`setTimeout()` 的第二个参数告诉 `JavaScript` 再过多长时间把当前任务添加到队列中。如果队列是空的，那么添加的代码会立即执行；如果队列不是空的，那么它就要等前面的代码执行完了以后再执行。

调用 `setTimeout()` 之后，该方法会返回一个数值 `ID`，表示超时调用。这个超时调用 `ID` 是计划执行代码的唯一标识符，可以通过它来取消超时调用。要取消尚未执行的超时调用计划，可以调用 `clearTimeout()` 方法并将相应的超时调用 `ID` 作为参数传递给它，如下所示。

```
// 设置超时调用
var timeoutId = setTimeout(function() {
    console.log("Hello world!");
}, 1000);

// 注意：把它取消
clearTimeout(timeoutId);
```

只要是在指定的时间尚未过去之前调用 `clearTimeout()`，就可以完全取消超时调用。前面的代码在设置超时调用之后马上又调用了 `clearTimeout()`，结果就跟什么也没有发生一样。

间歇调用与超时调用类似，只不过它会按照指定的时间间隔重复执行代码，直至间歇调用被取消或者页面被卸载。设置间歇调用的方法是 `setInterval()`，它接受的参数与 `setTimeout()` 相同：要执行的代码（字符串或函数）和每次执行之前需要等待的毫秒数。下面来看一个例子。

```
// 不建议传递字符串
setInterval ("console.log('Hello world!') ", 10000);

// 推荐的调用方式
setInterval (function() {
    console.log("Hello world!");
}, 10000);
```

调用 `setInterval()` 方法同样也会返回一个间歇调用 `ID`，该 `ID` 可用于在将来某个时刻取消间歇调用。要取消尚未执行的间歇调用，可以使用 `clearInterval()` 方法并传入相应的间歇调用 `ID`。取消间歇调用的重要性要远远高于取消超时调用，因为在不加干涉的情况下，间歇调用将会一直执行到页面卸载。以下是一个常见的使用间歇调用的例子。

```
var num = 0;
var max = 10;
var intervalId = null;

function incrementNumber() {
    num++;
    // 如果执行次数达到了max设定的值，则取消后续尚未执行的调用
    if (num == max) {
        clearInterval(intervalId);
        console.log("Done");
    }
}

intervalId = setInterval(incrementNumber, 500);
```

在这个例子中，变量num每半秒钟递增一次，当递增到最大值时就会取消先前设定的间歇调用。这个模式也可以使用超时调用来实现，如下所示。

```
var num = 0;
var max = 10;

function incrementNumber() {
    num++;

    // 如果执行次数未达到max设定的值，则设置另一次超时调用
    if (num < max) {
        setTimeout(incrementNumber, 500);
    } else {
        console.log("Done");
    }
}

setTimeout(incrementNumber, 500);
```

可见，在使用超时调用时，没有必要跟踪超时调用 `ID`，因为每次执行代码之后，如果不再设置另一次超时调用，调用就会自行停止。一般认为，使用超时调用来模拟间歇调用的是一种最佳模式。在开发环境下，很少使用真正的间歇调用，原因是后一个间歇调用可能会在前一个间歇调用结束之前启动。而像前面示例中那样使用超时调用，则完全可以避免这一点。所以，最好不要使用间歇调用。

系统对话框

浏览器通过 `alert()`、`confirm()` 和 `prompt()` 方法可以调用系统对话框向用户显示消息。系统对话框与在浏览器中显示的网页没有关系，也不包含 HTML。它们的外观由操作系统及（或）浏览器设置决定，而不是由 CSS 决定。此外，通过这几个方法打开的对话框都是同步和模态的。也就是说，显示这些对话框的时候代码会停止执行，而关掉这些对话框后代码又会恢复执行。

第一种对话框是调用 `alert()` 方法生成的。它向用户显示一个系统对话框，其中包含指定的文本和一个 OK（“确定”）按钮。通常使用 `alert()` 生成的“警告”对话框向用户显示一些他们无法控制的消息，例如错误消息。而用户只能在看完消息后关闭对话框。

第二种对话框是调用 `confirm()` 方法生成的。从向用户显示消息的方面来看，这种“确认”对话框很像是一个“警告”对话框。但二者的主要区别在于“确认”对话框除了显示OK按钮外，还会显示一个 Cancel（“取消”）按钮，两个按钮可以让用户决定是否执行给定的操作。

为了确定用户是单击了OK还是Cancel，可以检查 `confirm()` 方法返回的布尔值：`true` 表示单击了OK，`false` 表示单击了Cancel或单击了右上角的 X 按钮。确认对话框的典型用法如下。

```
if (confirm("Are you sure?")) {
    alert("I'm so glad you're sure! ");
} else {
    alert("I'm sorry to hear you're not sure.");
}
```

最后一种对话框是通过调用 `prompt()` 方法生成的，这是一个“提示”框，用于提示用户输入一些文本。提示框中除了显示 OK 和 Cancel 按钮之外，还会显示一个文本输入域，以供用户在其中输入内容。`prompt()` 方法接受两个参数：要显示给用户的文本提示和文本输入域的默认值（可以是一个空字符串）。

如果用户单击了 OK 按钮，则 `promp()` 返回文本输入域的值；如果用户单击了 Cancel 或没有单击 OK 而是通过其他方式关闭了对话框，则该方法返回 `null`。下面是一个例子。

```
var result = prompt("What is your name? ", "");
if (result !== null) {
    alert("Welcome, " + result);
}
```

综上所述，这些系统对话框很适合向用户显示消息并请用户作出决定。由于不涉及 HTML、CSS 或 JavaScript，因此它们是增强 Web 应用程序的一种便捷方式。

`location` 对象

`location` 对象提供了与当前窗口中加载的文档有关的信息，还提供了一些导航功能。事实上，`location` 对象是很特别的一个对象，因为它既是 `window` 对象的属性，也是 `document` 对象的属性；换句话说，`window.location` 和 `document.location` 引用的是同一个对象。`location` 对象的用处不只表现在它保存着当前文档的信息，还表现在它将 URL 解析为独立的片段，让开发人员可以通过不同的属性访问这些片段。下表列出了 `location` 对象的所有属性。

属性名	例子	说明
hash	"#contents"	返回 URL 中的 hash（#号后跟零或多个字符），如果 URL 中不包含散列，则返回空字符串
host	"shijiajie.com:80"	返回服务器名称和端口号（如果有）
hostname	"shijiajie.com"	返回不带端口号的服务器名称
href	"http:/shijiajie.com"	返回当前加载页面的完整URL。而 `location` 对象的 `toString()` 方法也返回这个值
pathname	"/WileyCDA/"	返回URL中的目录和（或）文件名
port	"8080"	返回 URL 中指定的端口号。如果 URL 中不包含端口号，则这个属性返回空字符串
protocol	"http:"	返回页面使用的协议。通常是 http: 或 https:
search	"?q=javascript"	返回URL的查询字符串。这个字符串以问号开头

查询字符串参数

虽然通过上面的属性可以访问到 `location` 对象的大多数信息，但其中访问URL包含的查询字符串的属性并不方便。尽管 `location.search` 返回从问号到 URL 末尾的所有内容，但却没有办法逐个访问其中的每个查询字符串参数。为此，可以像下面这样创建一个函数，用以解析查询字符串，然后返回包含所有参数的一个对象：

```
/*
 * 这个函数用来解析来自URL的查询串中的name=value参数对
 * 它将name=value对存储在一个对象的属性中，并返回该对象
 * 这样来使用它
 *
 * var args = urlArgs(); // 从URL中解析参数
 * var q = args.q || ""; // 如果参数定义了的话就使用参数；否则使用一个默认值
 * var n = args.n ? parseInt(args.n) : 10;
 */
function urlArgs() {
    var args = {};                                  // 定义一个空对象
    var query = location.search.substring(1);       // 查找到查询串，并去掉'? '
    var pairs = query.split("&");                   // 根据"&"符号将查询字符串分隔开
    for (var i = 0; i < pairs.length; i++) {        // 对于每个片段
        var pos = pairs[i].indexOf('=');            // 查找"name=value"
        if (pos == -1) continue;                    // 如果没有找到的话，就跳过
        var name = pairs[i].substring(0, pos);      // 提取name
        var value = pairs[i].substring(pos + 1);    // 提取value
        value = decodeURIComponent(value);          // 对value进行解码
        args[name] = value;                         // 存储为属性
    }
    return args;                                    // 返回解析后的参数
}
```

位置操作

使用 `location` 对象可以通过很多方式来改变浏览器的位置。首先，也是最常用的方式，就是使用 `assign()` 方法并为其传递一个 URL，如下所示。

```
location.assign("http://shijiajie.com");
```

这样，就可以立即打开新URL并在浏览器的历史记录中生成一条记录。如果是将 `location.href` 或 `window.location` 设置为一个URL值，也会以该值调用 `assign()` 方法。例如，下列两行代码与显式调用 `assign()` 方法的效果完全一样。

```
window.location = "http://shijiajie.com";
location.href = "http://shijiajie.com";
```

在这些改变浏览器位置的方法中，最常用的是设置 `location.href` 属性。

另外，修改 `location` 对象的其他属性也可以改变当前加载的页面。下面的例子展示了通过将 `hash`、`search`、`hostname`、`pathname` 和 `port` 属性设置为新值来改变 URL。

```
// 假设初始 URL 为 http://shijiajie.com/about/
location.href = "http://shijiajie.com/about/"

// 将 URL 修改为 "http://shijiajie.com/about/#ds-thread"
location.hash = "#ds-thread";

// 将 URL 修改为 "http://shijiajie.com/about/?args=123"
location.search = "?args=123";

// 将 URL 修改为 "https://segmentfault.com/"
location.hostname = "segmentfault.com";

// 将 URL 修改为 "http://segmentfault.com/u/stone0090/"
location.pathname = "u/stone0090";

// 将 URL 修改为 "https://segmentfault.com:8080/"
location.port = 8080;
```

当通过上述任何一种方式修改URL之后，浏览器的历史记录中就会生成一条新记录，因此用户通过单击“后退”按钮都会导航到前一个页面。要禁用这种行为，可以使用 `replace()` 方法。这个方法只接受一个参数，即要导航到的 URL；结果虽然会导致浏览器位置改变，但不会在历史记录中生成新记录。在调用 `replace()` 方法之后，用户不能回到前一个页面，来看下面的例子：

```
<!DOCTYPE html>
<html>
<head>
    <title>You won't be able to get back here</title>
</head>
    <body>
    <p>Enjoy this page for a second, because you won't be coming back here.</p>
    <script type="text/javascript">
        setTimeout(function () {
            location.replace("http://shijiajie.com/");
        }, 1000);
    </script>
</body>
</html>
```

如果将这个页面加载到浏览器中，浏览器就会在1秒钟后重新定向到 `shijiajie.com`。然后，“后退”按钮将处于禁用状态，如果不重新输入完整的URL，则无法返回示例页面。

与位置有关的最后一个方法是 `reload()`，作用是重新加载当前显示的页面。如果调用 `reload()` 时不传递任何参数，页面就会以最有效的方式重新加载。也就是说，如果页面自上次请求以来并没有改变过，页面就会从浏览器缓存中重新加载。如果要强制从服务器重新加载，则需要像下面这样为该方法传递参数 `true`。

```
location.reload();        // 重新加载（有可能从缓存中加载）
location.reload(true);    // 重新加载（从服务器重新加载）
```

位于 `reload()` 调用之后的代码可能会也可能不会执行，这要取决于网络延迟或系统资源等因素。为此，最好将 `reload()` 放在代码的最后一行。

`history` 对象

`history` 对象保存着用户上网的历史记录，从窗口被打开的那一刻算起。因为 `history` 是 `window` 对象的属性，因此每个浏览器窗口、每个标签

页乃至每个框架，都有自己的 `history` 对象与特定的 `window` 对象关联。出于安全方面的考虑，开发人员无法得知用户浏览过的 URL。不过，借由用户访问过的页面列表，同样可以在不知道实际 URL 的情况下实现后退和前进。

使用 `go()` 方法可以在用户的历史记录中任意跳转，可以向后也可以向前。这个方法接受一个参数，表示向后或向前跳转的页面数的一个整数值。负数表示向后跳转（类似于单击浏览器的“后退”按钮），正数表示向前跳转（类似于单击浏览器的“前进”按钮）。来看下面的例子。

```
// 后退一页
history.go(-1);

// 前进一页
history.go(1);

// 前进两页
history.go(2);
```

也可以给 `go()` 方法传递一个字符串参数，此时浏览器会跳转到历史记录中包含该字符串的第一个位置——可能后退，也可能前进，具体要看哪个位置最近。如果历史记录中不包含该字符串，那么这个方法什么也不做，例如：

```
// 跳转到最近的 shijiajie.com 页面
history.go("shijiajie.com");
```

另外，还可以使用两个简写方法 `back()` 和 `forward()` 来代替 `go()`。顾名思义，这两个方法可以模仿浏览器的“后退”和“前进”按钮。

```
// 后退一页
history.back();

// 前进一页
history.forward();
```

除了上述几个方法外，`history` 对象还有一个 `length` 属性，保存着历史记录的数量。这个数量包括所有历史记录，即所有向后和向前的记录。对于加载到窗口、标签页或框架中的第一个页面而言，`history.length` 等于0。通过像下面这样测试该属性的值，可以确定用户是否一开始就打开了你的页面。

```
if (history.length == 0){
    //这应该是用户打开窗口后的第一个页面
}
```

虽然 `history` 并不常用，但在创建自定义的“后退”和“前进”按钮，以及检测当前页面是不是用户历史记录中的第一个页面时，还是必须使用它。

小结

BOM（浏览器对象模型）以 `window` 对象为依托，表示浏览器窗口以及页面可见区域。同时，`window` 对象还是 ECMAScript 中的 `Global` 对象，因而所有全局变量和函数都是它的属性，且所有原生的构造函数及其他函数也都存在于它的命名空间下。本章讨论了下列 BOM 的组成部分。

- 在使用框架时，每个框架都有自己的 `window` 对象以及所有原生构造函数及其他函数的副本。每个框架都保存在 `frames` 集合中，可以通过位置或通过名称来访问。
- 有一些窗口指针，可以用来引用其他框架，包括父框架。
- `top` 对象始终指向最外围的框架，也就是整个浏览器窗口。
- `parent` 对象表示包含当前框架的框架，而 `self` 对象则回指 `window`。
- 使用 `location` 对象可以通过编程方式来访问浏览器的导航系统。设置相应的属性，可以逐段或整体性地修改浏览器的 URL。
- 调用 `replace()` 方法可以导航到一个新 URL，同时该 URL 会替换浏览器历史记录中当前显示的页面。
- `navigator` 对象提供了与浏览器有关的信息。到底提供哪些信息，很大程度上取决于用户的浏览器；不过，也有一些公共的属性（如 `userAgent`）存在于所有浏览器中。

BOM中还有两个对象：`screen` 和 `history`，但它们的功能有限。`screen` 对象中保存着与客户端显示器有关的信息，这些信息一般只用于站点分析。`history` 对象为访问浏览器的历史记录开了一个小缝隙，开发人员可以据此判断历史记录的数量，也可以在历史记录中向后或向前导航到任意页面。

关卡

```
// 挑战一
setTimeout(function () {
    console.log("1");
}, 0)
console.log("2");   // ???
```

```
// 挑战二
for (var i = 0;i<5;i++) {
    setTimeout(function () {
        console.log(i);     // ???
    }, 0)
};
```

```
// 挑战三
var a = 1;
var obj = {
    a : 2,
    b : function(){
        setTimeout(function () {
            console.log(this.a);
        }, 0)
    }
}
obj.b();    // ???
```

```
// 挑战四
var a = 1;
var obj = {
    a : 2,
    b : function(){
        setTimeout(function () {
            console.log(this.a);
        }.call(this), 0);
    }
}
obj.b();    // ???
```

更多

> 关注微信公众号「劼哥舍」回复「答案」，获取关卡详解。
> 关注 https://github.com/stone0090/javascript-lessons，获取最新动态。

11月7日发布　更多

你可能感兴趣的文章

第一章 JavaScript简介　1 收藏，392 浏览

前端Javascript与Nodejs的异同　18 收藏，605 浏览

JavaScript BOM——“location 对象”的注意要点　3 收藏，607 浏览

讨论区

很细致，赞一个

一点灵犀 · 11月8日

很细致，赞一个

爱喝可乐 · 4 天前

使用评论询问更多信息或提出修改意见，请不要在评论里回答问题

提交评论

评论支持部分 Markdown 语法：`**bold**` `_italic_` `[link](http://example.com)` `> 引用` `` `code` `` `- 列表`。
同时，被你 @ 的用户也会收到通知

本文隶属于专栏

劼哥舍

欢迎来到「劼哥舍」，您非要念成「劫个色」也行。这里坚持原创，分享随笔和学习心得，主要涉及 前端 / .NET / Java 等方面的内容。欢迎交流，欢迎提问，欢迎转载，但需注明出处。

劼哥stone

作者 关注作者

关注专栏

系列文章

《JavaScript 闯关记》之对象 6 收藏， 164 浏览

《JavaScript 闯关记》之数组 6 收藏， 232 浏览

《JavaScript 闯关记》之函数 3 收藏， 161 浏览

《JavaScript 闯关记》之正则表达式 30 收藏， 808 浏览

《JavaScript 闯关记》之基本包装类型 18 收藏， 861 浏览

《JavaScript 闯关记》之单体内置对象 11 收藏， 419 浏览

《JavaScript 闯关记》之 DOM（上） 3 收藏， 231 浏览

相关收藏夹

换一组

闭包和作用域
4 个条目 | 1 人关注

rxjs
5 个条目 | 0 人关注

other
5 个条目 | 0 人关注

分享扩散：