# 浏览器环境概述

JavaScript 是浏览器的内置脚本语言。也就是说，浏览器内置了 JavaScript 引擎，并且提供各种接口，让 JavaScript 脚本可以控制浏览器的各种功能。一旦网页内嵌了 JavaScript 脚本，浏览器加载网页，就会去执行脚本，从而达到操作浏览器的目的，实现网页的各种动态效果。

本章开始介绍浏览器提供的各种 JavaScript 接口。首先，介绍 JavaScript 代码嵌入网页的方法。

## 代码嵌入网页的方法

网页中嵌入 JavaScript 代码，主要有三种方法。

- `<script>`元素直接嵌入代码。
- `<script>`标签加载外部脚本
- 事件属性
- URL 协议

### script 元素嵌入代码

`<script>`元素内部可以直接写 JavaScript 代码。

```html
<script>
  var x = 1 + 5;
  console.log(x);
</script>
```

`<script>`标签有一个`type`属性，用来指定脚本类型。对 JavaScript 脚本来说，`type`属性可以设为两种值。

- `text/javascript`：这是默认值，也是历史上一贯设定的值。如果你省略`type`属性，默认就是这个值。对于老式浏览器，设为这个值比较好。
- `application/javascript`：对于较新的浏览器，建议设为这个值。

```html
<script type="application/javascript">
  console.log('Hello World');
</script>
```

由于`<script>`标签默认就是 JavaScript 代码。所以，嵌入 JavaScript 脚本时，`type`属性可以省略。

如果`type`属性的值，浏览器不认识，那么它不会执行其中的代码。利用这一点，可以在`<script>`标签之中嵌入任意的文本内容，只要加上一个浏览器不认识的`type`属性即可。

```html
<script id="mydata" type="x-custom-data">
  console.log('Hello World');
</script>
```

上面的代码，浏览器不会执行，也不会显示它的内容，因为不认识它的`type`属性。但是，这个`<script>`节点依然存在于 DOM 之中，可以使用`<script>`节点的`text`属性读出它的内容。

```javascript
document.getElementById('mydata').text
//   console.log('Hello World');
```

### script 元素加载外部脚本

`<script>`标签也可以指定加载外部的脚本文件。

```html
<script src="https://www.example.com/script.js"></script>
```

如果脚本文件使用了非英语字符，还应该注明字符的编码。

```html
<script charset="utf-8" src="https://www.example.com/script.js"></script>
```

所加载的脚本必须是纯的 JavaScript 代码，不能有`HTML`代码和`<script>`标签。

加载外部脚本和直接添加代码块，这两种方法不能混用。下面代码的`console.log`语句直接被忽略。

```html
<script charset="utf-8" src="example.js">
  console.log('Hello World!');
</script>
```

为了防止攻击者篡改外部脚本，`script`标签允许设置一个`integrity`属性，写入该外部脚本的 Hash 签名，用来验证脚本的一致性。

```html
<script src="/assets/application.js"
  integrity="sha256-TvVUHzSfftWg1rcfL6TlJ0XKEGrgLyEq6lEpcmrG9qs=">
</script>
```

```

上面代码中，`script`标签有一个`integrity`属性，指定了外部脚本`/assets/application.js`的 SHA256 签名。一旦有人改了这个脚本，导致 SHA256 签名不匹配，浏览器就会拒绝加载。

### 事件属性

网页元素的事件属性（比如`onclick`和`onmouseover`），可以写入 JavaScript 代码。当指定事件发生时，就会调用这些代码。

```html
<button id="myBtn" onclick="console.log(this.id)">点击</button>
```

上面的事件属性代码只有一个语句。如果有多个语句，使用分号分隔即可。

### URL 协议

URL 支持`javascript:`协议，即在 URL 的位置写入代码，使用这个 URL 的时候就会执行 JavaScript 代码。

```html
<a href="javascript:console.log('Hello')">点击</a>
```

浏览器的地址栏也可以执行`javascript:`协议。将`javascript:console.log('Hello')`放入地址栏，按回车键也会执行这段代码。

如果 JavaScript 代码返回一个字符串，浏览器就会新建一个文档，展示这个字符串的内容，原有文档的内容都会消失。

```html
<a href="javascript: new Date().toLocaleTimeString();">点击</a>
```

上面代码中，用户点击链接以后，会打开一个新文档，里面有当前时间。

如果返回的不是字符串，那么浏览器不会新建文档，也不会跳转。

```javascript
<a href="javascript: console.log(new Date().toLocaleTimeString())">点击</a>
```

上面代码中，用户点击链接后，网页不会跳转，只会在控制台显示当前时间。
```

`javascript:`协议的常见用途是书签脚本 Bookmarklet。由于浏览器的书签保存的是一个网址，所以`javascript:`网址也可以保存在里面，用户选择这个书签的时候，就会在当前页面执行这个脚本。为了防止书签替换掉当前文档，可以在脚本前加上`void`，或者在脚本最后加上`void 0`。

```html
<a href="javascript: void new Date().toLocaleTimeString();">点击</a>
<a href="javascript: new Date().toLocaleTimeString();void 0;">点击</a>
```

上面这两种写法，点击链接后，执行代码都不会网页跳转。

## script 元素

### 工作原理

浏览器加载 JavaScript 脚本，主要通过`<script>`元素完成。正常的网页加载流程是这样的。

1. 浏览器一边下载 HTML 网页，一边开始解析。也就是说，不等到下载完，就开始解析。
2. 解析过程中，浏览器发现`<script>`元素，就暂停解析，把网页渲染的控制权转交给 JavaScript 引擎。
3. 如果`<script>`元素引用了外部脚本，就下载该脚本再执行，否则就直接执行代码。
4. JavaScript 引擎执行完毕，控制权交还渲染引擎，恢复往下解析 HTML 网页。

加载外部脚本时，浏览器会暂停页面渲染，等待脚本下载并执行完成后，再继续渲染。原因是 JavaScript 代码可以修改 DOM，所以必须把控制权让给它，否则会导致复杂的线程竞赛的问题。

如果外部脚本加载时间很长（一直无法完成下载），那么浏览器就会一直等待脚本下载完成，造成网页长时间失去响应，浏览器就会呈现“假死”状态，这被称为“阻塞效应”。

为了避免这种情况，较好的做法是将`<script>`标签都放在页面底部，而不是头部。这样即使遇到脚本失去响应，网页主体的渲染也已经完成了，用户至少可以看到内容，而不是面对一张空白的页面。如果某些脚本代码非常重要，一定要放在页面头部的话，最好直接将代码写入页面，而不是连接外部脚本文件，这样能缩短加载时间。

脚本文件都放在网页尾部加载，还有一个好处。因为在 DOM 结构生成之前就调用 DOM 节点，JavaScript 会报错，如果脚本都在网页尾部加载，就不存在这个问题，因为这时 DOM 肯定已经生成了。

```html
<head>
  <script>
    console.log(document.body.innerHTML);
  </script>
```

```
</head>
<body>
</body>
```

上面代码执行时会报错，因为此时`document.body`元素还未生成。

一种解决方法是设定`DOMContentLoaded`事件的回调函数。

```html
<head>
  <script>
    document.addEventListener(
      'DOMContentLoaded',
      function (event) {
        console.log(document.body.innerHTML);
      }
    );
  </script>
</head>
```

上面代码中，指定`DOMContentLoaded`事件发生后，才开始执行相关代码。
`DOMContentLoaded`事件只有在 DOM 结构生成之后才会触发。

另一种解决方法是，使用`<script>`标签的`onload`属性。当`<script>`标签指定的外部脚本文件下载和解析完成，会触发一个`load`事件，可以把所需执行的代码，放在这个事件的回调函数里面。

```html
<script src="jquery.min.js" onload="console.log(document.body.innerHTML)">
</script>
```

但是，如果将脚本放在页面底部，就可以完全按照正常的方式写，上面两种方式都不需要。

```html
<body>
  <!-- 其他代码  -->
  <script>
    console.log(document.body.innerHTML);
  </script>
</body>
```

如果有多个`script`标签，比如下面这样。

```html
<script src="a.js"></script>
<script src="b.js"></script>
```

浏览器会同时并行下载`a.js`和`b.js`，但是，执行时会保证先执行`a.js`，然后再执行`b.js`，即使后者先下载完成，也是如此。也就是说，脚本的执行顺序由它们在页面中的出现顺序决定，这是为了保证脚本之间的依赖关系不受到破坏。当然，加载这两个脚本都会产生“阻塞效应”，必须等到它们都加载完成，浏览器才会继续页面渲染。

解析和执行 CSS，也会产生阻塞。Firefox 浏览器会等到脚本前面的所有样式表，都下载并解析完，再执行脚本；Webkit则是一旦发现脚本引用了样式，就会暂停执行脚本，等到样式表下载并解析完，再恢复执行。

此外，对于来自同一个域名的资源，比如脚本文件、样式表文件、图片文件等，浏览器一般有限制，同时最多下载6～20个资源，即最多同时打开的 TCP 连接有限制，这是为了防止对服务器造成太大压力。如果是来自不同域名的资源，就没有这个限制。所以，通常把静态文件放在不同的域名之下，以加快下载速度。

### defer 属性

为了解决脚本文件下载阻塞网页渲染的问题，一个方法是对`<script>`元素加入`defer`属性。它的作用是延迟脚本的执行，等到 DOM 加载生成后，再执行脚本。

```html
<script src="a.js" defer></script>
<script src="b.js" defer></script>
```

上面代码中，只有等到 DOM 加载完成后，才会执行`a.js`和`b.js`。

`defer`属性的运行流程如下。

1. 浏览器开始解析 HTML 网页。
2. 解析过程中，发现带有`defer`属性的`<script>`元素。
3. 浏览器继续往下解析 HTML 网页，同时并行下载`<script>`元素加载的外部脚本。
4. 浏览器完成解析 HTML 网页，此时再回过头执行已经下载完成的脚本。

有了`defer`属性，浏览器下载脚本文件的时候，不会阻塞页面渲染。下载的脚本文件在`DOMContentLoaded`事件触发前执行（即刚刚读取完`</html>`标签），而且可以保证执行顺序就是它们在页面上出现的顺序。

对于内置而不是加载外部脚本的`script`标签，以及动态生成的`script`标签，`defer`属性不起作用。另外，使用`defer`加载的外部脚本不应该使用`document.write`方法。

### async 属性

解决“阻塞效应”的另一个方法是对`<script>`元素加入`async`属性。

```html
<script src="a.js" async></script>
<script src="b.js" async></script>
```

`async`属性的作用是，使用另一个进程下载脚本，下载时不会阻塞渲染。

1. 浏览器开始解析 HTML 网页。
2. 解析过程中，发现带有`async`属性的`script`标签。
3. 浏览器继续往下解析 HTML 网页，同时并行下载`<script>`标签中的外部脚本。
4. 脚本下载完成，浏览器暂停解析 HTML 网页，开始执行下载的脚本。
5. 脚本执行完毕，浏览器恢复解析 HTML 网页。

`async`属性可以保证脚本下载的同时，浏览器继续渲染。需要注意的是，一旦采用这个属性，就无法保证脚本的执行顺序。哪个脚本先下载结束，就先执行那个脚本。另外，使用`async`属性的脚本文件里面的代码，不应该使用`document.write`方法。

`defer`属性和`async`属性到底应该使用哪一个?

一般来说，如果脚本之间没有依赖关系，就使用`async`属性，如果脚本之间有依赖关系，就使用`defer`属性。如果同时使用`async`和`defer`属性，后者不起作用，浏览器行为由`async`属性决定。

### 脚本的动态加载

`<script>`元素还可以动态生成，生成后再插入页面，从而实现脚本的动态加载。

```javascript
['a.js', 'b.js'].forEach(function(src) {
  var script = document.createElement('script');
  script.src = src;
  document.head.appendChild(script);
});
```

这种方法的好处是，动态生成的`script`标签不会阻塞页面渲染，也就不会造成浏览器假死。但是问题在于，这种方法无法保证脚本的执行顺序，哪个脚本文件先下载完成，就先执行哪个。

如果想避免这个问题，可以设置async属性为`false`。

```javascript
['a.js', 'b.js'].forEach(function(src) {
  var script = document.createElement('script');
  script.src = src;
  script.async = false;
  document.head.appendChild(script);
```

```
});
```

上面的代码不会阻塞页面渲染，而且可以保证`b.js`在`a.js`后面执行。不过需要注意的是，在这段代码后面加载的脚本文件，会因此都等待`b.js`执行完成后再执行。

如果想为动态加载的脚本指定回调函数，可以使用下面的写法。

```javascript
function loadScript(src, done) {
  var js = document.createElement('script');
  js.src = src;
  js.onload = function() {
    done();
  };
  js.onerror = function() {
    done(new Error('Failed to load script ' + src));
  };
  document.head.appendChild(js);
}
```

### 加载使用的协议

如果不指定协议，浏览器默认采用 HTTP 协议下载。

```html
<script src="example.js"></script>
```

上面的`example.js`默认就是采用 HTTP 协议下载，如果要采用 HTTPS 协议下载，必需写明。

```html
<script src="https://example.js"></script>
```

但是有时我们会希望，根据页面本身的协议来决定加载协议，这时可以采用下面的写法。

```html
<script src="//example.js"></script>
```

## 浏览器的组成

浏览器的核心是两部分：渲染引擎和 JavaScript 解释器（又称 JavaScript 引擎）。

### 渲染引擎

渲染引擎的主要作用是，将网页代码渲染为用户视觉可以感知的平面文档。

不同的浏览器有不同的渲染引擎。

- Firefox：Gecko 引擎
- Safari：WebKit 引擎
- Chrome：Blink 引擎
- IE: Trident 引擎
- Edge: EdgeHTML 引擎

渲染引擎处理网页，通常分成四个阶段。

1. 解析代码：HTML 代码解析为 DOM，CSS 代码解析为 CSSOM（CSS Object Model）。
2. 对象合成：将 DOM 和 CSSOM 合成一棵渲染树（render tree）。
3. 布局：计算出渲染树的布局（layout）。
4. 绘制：将渲染树绘制到屏幕。

以上四步并非严格按顺序执行，往往第一步还没完成，第二步和第三步就已经开始了。所以，会看到这种情况：网页的 HTML 代码还没下载完，但浏览器已经显示出内容了。

### 重流和重绘

渲染树转换为网页布局，称为“布局流”（flow）；布局显示到页面的这个过程，称为“绘制”（paint）。它们都具有阻塞效应，并且会耗费很多时间和计算资源。

页面生成以后，脚本操作和样式表操作，都会触发“重流”（reflow）和“重绘”（repaint）。用户的互动也会触发重流和重绘，比如设置了鼠标悬停（`a:hover`）效果、页面滚动、在输入框中输入文本、改变窗口大小等等。

重流和重绘并不一定一起发生，重流必然导致重绘，重绘不一定需要重流。比如改变元素颜色，只会导致重绘，而不会导致重流；改变元素的布局，则会导致重绘和重流。

大多数情况下，浏览器会智能判断，将重流和重绘只限制到相关的子树上面，最小化所耗费的代价，而不会全局重新生成网页。

作为开发者，应该尽量设法降低重绘的次数和成本。比如，尽量不要变动高层的 DOM 元素，而以底层 DOM 元素的变动代替；再比如，重绘`table`布局和`flex`布局，开销都会比较大。

```javascript
var foo = document.getElementById('foobar');

foo.style.color = 'blue';
foo.style.marginTop = '30px';
```

上面的代码只会导致一次重绘，因为浏览器会累积 DOM 变动，然后一次性执行。

下面是一些优化技巧。

- 读取 DOM 或者写入 DOM，尽量写在一起，不要混杂。不要读取一个 DOM 节点，然后立刻写入，接着再读取一个 DOM 节点。
- 缓存 DOM 信息。
- 不要一项一项地改变样式，而是使用 CSS class 一次性改变样式。
- 使用`documentFragment`操作 DOM
- 动画使用`absolute`定位或`fixed`定位，这样可以减少对其他元素的影响。
- 只在必要时才显示隐藏元素。
- 使用`window.requestAnimationFrame()`，因为它可以把代码推迟到下一次重流时执行，而不是立即要求页面重流。
- 使用虚拟 DOM（virtual DOM）库。

下面是一个`window.requestAnimationFrame()`对比效果的例子。

```javascript
// 重绘代价高
function doubleHeight(element) {
  var currentHeight = element.clientHeight;
  element.style.height = (currentHeight * 2) + 'px';
}

all_my_elements.forEach(doubleHeight);

// 重绘代价低
function doubleHeight(element) {
  var currentHeight = element.clientHeight;

  window.requestAnimationFrame(function () {
    element.style.height = (currentHeight * 2) + 'px';
  });
}

all_my_elements.forEach(doubleHeight);
```

上面的第一段代码，每读一次 DOM，就写入新的值，会造成不停的重排和重流。第二段代码把所有的写操作，都累积在一起，从而 DOM 代码变动的代价就最小化了。

### JavaScript 引擎

JavaScript 引擎的主要作用是，读取网页中的 JavaScript 代码，对其处理后运行。

JavaScript 是一种解释型语言，也就是说，它不需要编译，由解释器实时运行。这样的好处是运行和修改都比较方便，刷新页面就可以重新解释；缺点是每次运行都要调用解释器，系统开销较大，运行速度慢于编译型语言。

为了提高运行速度，目前的浏览器都将 JavaScript 进行一定程度的编译，生成类似字节码（bytecode）的中间代码，以提高运行速度。

早期，浏览器内部对 JavaScript 的处理过程如下：

1. 读取代码，进行词法分析（Lexical analysis），将代码分解成词元（token）。
2. 对词元进行语法分析（parsing），将代码整理成“语法树”（syntax tree）。
3. 使用“翻译器”（translator），将代码转为字节码（bytecode）。
4. 使用“字节码解释器”（bytecode interpreter），将字节码转为机器码。

逐行解释将字节码转为机器码，是很低效的。为了提高运行速度，现代浏览器改为采用“即时编译”（Just In Time compiler，缩写 JIT），即字节码只在运行时编译，用到哪一行就编译哪一行，并且把编译结果缓存（inline cache）。通常，一个程序被经常用到的，只是其中一小部分代码，有了缓存的编译结果，整个程序的运行速度就会显著提升。

字节码不能直接运行，而是运行在一个虚拟机（Virtual Machine）之上，一般也把虚拟机称为 JavaScript 引擎。并非所有的 JavaScript 虚拟机运行时都有字节码，有的 JavaScript 虚拟机基于源码，即只要有可能，就通过 JIT（just in time）编译器直接把源码编译成机器码运行，省略字节码步骤。这一点与其他采用虚拟机（比如 Java）的语言不尽相同。这样做的目的，是为了尽可能地优化代码、提高性能。下面是目前最常见的一些 JavaScript 虚拟机：

- [Chakra](https://en.wikipedia.org/wiki/Chakra_(JScript_engine)) (Microsoft Internet Explorer)
- [Nitro/JavaScript Core](http://en.wikipedia.org/wiki/WebKit#JavaScriptCore) (Safari)
- [Carakan](http://dev.opera.com/articles/view/labs-carakan/) (Opera)
- [SpiderMonkey](https://developer.mozilla.org/en-US/docs/SpiderMonkey) (Firefox)
- [V8](https://en.wikipedia.org/wiki/Chrome_V8) (Chrome, Chromium)

## 参考链接


- John Dalziel, [The race for speed part 2: How JavaScript compilers work](http://creativejs.com/2013/06/the-race-for-speed-part-2-how-javascript-compilers-work/)
- Jake Archibald, [Deep dive into the murky waters of script loading](http://www.html5rocks.com/en/tutorials/speed/script-loading/)
- Mozilla Developer Network, [window.setTimeout](https://developer.mozilla.org/en-US/docs/Web/API/window.setTimeout)
- Remy Sharp, [Throttling function calls](http://remysharp.com/2010/07/21/throttling-function-calls/)
- Ayman Farhat, [An alternative to JavaScript's evil setInterval](http://www.thecodeship.com/web-development/alternative-to-javascript-evil-setinterval/)
- Ilya Grigorik, [Script-injected "async scripts" considered harmful](https://www.igvita.com/2014/05/20/script-injected-async-scripts-considered-harmful/)

- Axel Rauschmayer, [ECMAScript 6 promises (1/2): foundations](http://www.2ality.com/2014/09/es6-promises-foundations.html)
- Daniel Imms, [async vs defer attributes](http://www.growingwiththeweb.com/2014/02/async-vs-defer-attributes.html)
- Craig Buckler, [Load Non-blocking JavaScript with HTML5 Async and Defer](http://www.sitepoint.com/non-blocking-async-defer/)
- Domenico De Felice, [How browsers work](http://domenicodefelice.blogspot.sg/2015/08/how-browsers-work.html?t=2)