

第三部分

其他文章

JS

Ilya Kantor

建于 2020年6月2日

最新版教程详见 <https://zh.javascript.info.>

我们在不断努力改进教程。如果你发现任何错误, 请在[我们的 GitHub](#) 上告诉我们。.

- [Frame 和 window](#)
 - 弹窗和 window 的方法
 - 跨窗口通信
 - 点击劫持攻击
- [二进制数据, 文件](#)
 - [ArrayBuffer](#), 二进制数组
 - [TextDecoder](#) 和 [TextEncoder](#)
 - [Blob](#)
 - [File](#) 和 [FileReader](#)
- [网络请求](#)
 - [Fetch](#)
 - [FormData](#)
 - [Fetch: 下载进度](#)
 - [Fetch: 中止 \(Abort\)](#)
 - [Fetch: 跨源请求](#)
 - [Fetch API](#)
 - [URL 对象](#)
 - [XMLHttpRequest](#)
 - [可恢复的文件上传](#)
 - [长轮询 \(Long polling\)](#)
 - [WebSocket](#)
 - [Server Sent Events](#)
- [在浏览器中存储数据](#)
 - [Cookie, document.cookie](#)
 - [LocalStorage, sessionStorage](#)
 - [IndexedDB](#)
- [动画](#)
 - 贝塞尔曲线
 - [CSS 动画](#)
 - [JavaScript 动画](#)
- [Web components](#)
 - 从星球轨道的高度讲起
 - [Custom elements](#)

- 影子 DOM (Shadow DOM)
- 模板元素
- Shadow DOM 插槽, 组成
- 给 Shadow DOM 添加样式
- Shadow DOM 和事件 (events)
- 正则表达式
 - 模式 (Patterns) 和修饰符 (flags)
 - 字符类
 - Unicode: 修饰符 "u" 和 class \p{...}
 - 锚点 (Anchors): 字符串开始 ^ 和末尾 \$
 - Flag "m" — 多行模式
 - 词边界: \b
 - 转义, 特殊字符
 - 集合和范围 [...]
 - 量词 `+,*,?` 和 `{n}`
 - 贪婪量词和惰性量词
 - 捕获组
 - 模式中的反向引用: \N 和 \k<name>
 - 选择 (OR) |
 - 前瞻断言与后瞻断言
 - Catastrophic backtracking
 - 粘性标志 "y", 在位置处搜索
 - 正则表达式 (RegExp) 和字符串 (String) 的方法

Frame 和 window

弹窗和 window 的方法

弹窗（popup）是向用户显示其他文档的最古老的方法之一。

基本上，你只需要运行：

```
window.open('https://javascript.info/')
```

.....它将打开一个具有给定 URL 的新窗口。大多数现代浏览器都配置为打开新选项卡，而不是单独的窗口。

弹窗自古以来就存在。最初的想法是，在不关闭主窗口的情况下显示其他内容。目前为止，还有其他方式可以实现这一点：我们可以使用 `fetch` 动态加载内容，并将其显示在动态生成的 `<div>` 中。弹窗并不是我们每天都会使用的东西。

并且，弹窗在移动设备上非常棘手，因为移动设备无法同时显示多个窗口。

但仍然有一些任务在使用弹窗，例如进行 OAuth 授权（使用 Google/Facebook/... 登陆），因为：

1. 弹窗是一个独立的窗口，具有自己的独立 JavaScript 环境。因此，使用弹窗打开一个不信任的第三方网站是安全的。
2. 打开弹窗非常容易。
3. 弹窗可以导航（修改 URL），并将消息发送到 `opener` 窗口（译注：即打开弹窗的窗口）。

阻止弹窗

在过去，很多恶意网站经常滥用弹窗。一个不好的页面可能会打开大量带有广告的弹窗。因此，现在大多数浏览器都会通过阻止弹窗来保护用户。

如果弹窗是在用户触发的事件处理程序（如 `onclick`）之外调用的，大多数浏览器都会阻止此类弹窗。

例如：

```
// 弹窗被阻止
window.open('https://javascript.info');

// 弹窗被允许
button.onclick = () => {
  window.open('https://javascript.info');
};
```

这种方式可以在某种程度上保护用户免受非必要的弹窗的影响，但是并没有完全阻止该功能。

如果弹窗是从 `onclick` 打开的，但是在 `setTimeout` 之后，该怎么办？这有点棘手。

试试运行一下这段代码：

```
// 3 秒后打开弹窗  
setTimeout(() => window.open('http://google.com'), 3000);
```

这个弹窗在 Chrome 中会被打开，但是在 Firefox 中会被阻止。

……如果我们减少延迟，则弹窗在 Firefox 中也会被打开：

```
// 1 秒后打开弹窗  
setTimeout(() => window.open('http://google.com'), 1000);
```

区别在于 Firefox 可以接受 2000ms 或更短的延迟，但是超过这个时间——则移除“信任”。所以，第一个弹窗被阻止，而第二个却没有。

window.open

打开一个弹窗的语法是 `window.open(url, name, params)`：

url

要在新窗口中加载的 URL。

name

新窗口的名称。每个窗口都有一个 `window.name`，在这里我们可以指定哪个窗口用于弹窗。如果已经有一个这样名字的窗口——将在该窗口打开给定的 URL，否则会打开一个新窗口。

params

新窗口的配置字符串。它包括设置，用逗号分隔。参数之间不能有空格，例如：

```
width:200,height=100
```

`params` 的设置项：

- 位置：

- `left/top` (数字) —— 屏幕上窗口的左上角的坐标。这有一个限制：不能将新窗口置于屏幕外 (`offscreen`)。

- `width/height` (数字) —— 新窗口的宽度和高度。宽度/高度的最小值是有限制的，因此不可能创建一个不可见的窗口。
- 窗口功能：
 - `menubar` (yes/no) —— 显示或隐藏新窗口的浏览器菜单。
 - `toolbar` (yes/no) —— 显示或隐藏新窗口的浏览器导航栏（后退，前进，重新加载等）。
 - `location` (yes/no) —— 显示或隐藏新窗口的 URL 字段。Firefox 和 IE 浏览器不允许默认隐藏它。
 - `status` (yes/no) —— 显示或隐藏状态栏。同样，大多数浏览器都强制显示它。
 - `resizable` (yes/no) —— 允许禁用新窗口大小调整。不建议使用。
 - `scrollbars` (yes/no) —— 允许禁用新窗口的滚动条。不建议使用。

还有一些不太受支持的特定于浏览器的功能，通常不使用。通常不使用这些功能。更多示例请见 [MDN 中的 window.open](#)。

示例：一个最简窗口

让我们打开一个包含最小功能集的新窗口，来看看哪些功能是浏览器允许禁用的：

```
let params = `scrollbars=no,resizable=no,status=no,location=no,toolbar=no,menubar=no,
width=0,height=0,left=-1000,top=-1000`;
open('/', 'test', params);
```

在这里，大多数“窗口功能”都被禁用了，并且窗口位于屏幕外。运行它，看看会发生什么。大多数浏览器都会“修复”奇怪的东西，例如 `width/height` 为零以及脱离屏幕 (offscreen) 的 `left/top` 设置。例如，Chrome 打开了一个全 `width/height` 的窗口，使其占满整个屏幕。

让我们添加正常的定位选项和合理的 `width`、`height`、`left` 和 `top` 坐标：

```
let params = `scrollbars=no,resizable=no,status=no,location=no,toolbar=no,menubar=no,
width=600,height=300,left=100,top=100`;
open('/', 'test', params);
```

大多数浏览器会根据要求显示上面的示例。

设置中的省略规则：

- 如果 `open` 调用中没有第三个参数，或者它是空的，则使用默认的窗口参数。

- 如果这里有一个参数字符串，但是某些 `yes/no` 功能被省略了，那么被省略的功能则被默认值为 `no`。因此，如果你指定参数，请确保将所有必需的功能明确设置为 `yes`。
- 如果参数中没有 `left/top`，那么浏览器会尝试在最后打开的窗口附近打开一个新窗口。
- 如果没有 `width/height`，那么新窗口的大小将与上次打开的窗口大小相同。

从窗口访问弹窗

`open` 调用会返回对新窗口的引用。它可以用来操纵弹窗的属性，更改位置，甚至更多操作。

在下面这个示例中，我们从 JavaScript 中生成弹窗：

```
let newWin = window.open("about:blank", "hello", "width=200,height=200");
newWin.document.write("Hello, world!");
```

这里，我们在其加载完成后，修改其中的内容：

```
let newWindow = open('/', 'example', 'width=300,height=300')
newWindow.focus();

alert(newWindow.location.href); // (*) about:blank, 加载尚未开始

newWindow.onload = function() {
  let html = `<div style="font-size:30px">Welcome!</div>`;
  newWindow.document.body.insertAdjacentHTML('afterbegin', html);
};
```

请注意：在刚刚进行了 `window.open` 的时候，新窗口还没有加载完成。我们可以通过 (*) 行中的 `alert` 证实这一点。因此，我们需要等待 `onload` 以对新窗口进行更改。我们也可以对 `newWin.document` 使用 `DOMContentLoaded` 处理程序。

⚠ 同源策略

只有在窗口是同源的时，窗口才能自由访问彼此的内容（相同的协议：`//domain:port`）。

否则，例如，如果主窗口来自于 `site.com`，弹窗来自于 `gmail.com`，则出于安全性考虑，这两个窗口不能访问彼此的内容。有关详细信息，请参见 [跨窗口通信](#) 一章。

从弹窗访问窗口

弹窗也可以使用 `window.opener` 来访问 `opener` 窗口。除了弹窗之外，对其他所有窗口来说，`window.opener` 均为 `null`。

如果你运行下面这段代码，它将用“Test”替换 `opener`（也就是当前的）窗口的内容：

```
let newWin = window.open("about:blank", "hello", "width=200,height=200");

newWin.document.write(
  "<script>window.opener.document.body.innerHTML = 'Test'</script>"
);
```

所以，窗口之间的连接是双向的：主窗口和弹窗之间相互引用。

关闭弹窗

关闭一个窗口：`win.close()`。

检查一个窗口是否被关闭：`win.closed`。

从技术上讲，`close()` 方法可用于任何 `window`，但是如果 `window` 不是通过 `window.open()` 创建的，那么大多数浏览器都会忽略 `window.close()`。因此，`close()` 只对弹窗起作用。

如果窗口被关闭了，那么 `closed` 属性则为 `true`。这对于检查弹窗（或主窗口）是否仍处于打开状态很有用。用户可以随时关闭它，我们的代码应该考虑到这种可能性。

这段代码加载并关闭了窗口：

```
let newWindow = open('/', 'example', 'width=300,height=300');

newWindow.onload = function() {
  newWindow.close();
  alert(newWindow.closed); // true
};
```

滚动和调整大小

有一些方法可以移动一个窗口，或者调整一个窗口的大小：

`win.moveTo(x, y)`

将窗口相对于当前位置向右移动 `x` 像素，并向下移动 `y` 像素。允许负值（向上/向左移动）。

`win.moveTo(x, y)`

将窗口移动到屏幕上的坐标 `(x, y)` 处。

`win.resizeBy(width, height)`

根据给定的相对于当前大小的 `width/height` 调整窗口大小。允许负值。

`win.resizeTo(width, height)`

将窗口调整为给定的大小。

还有 `window.onresize` 事件。

仅对于弹窗

为了防止滥用，浏览器通常会阻止这些方法。它们仅在我们打开的，没有其他选项卡的弹窗中能够可靠地工作。

没有最小化/最大化

JavaScript 无法最小化或者最大化一个窗口。这些操作系统级别的功能对于前端开发者而言是隐藏的。

移动或者调整大小的方法不适用于最小化/最大化的窗口。

滚动窗口

我们已经在 [Window 大小和滚动](#) 一章中讨论过了滚动窗口。

`win.scrollBy(x, y)`

相对于当前位置，将窗口向右滚动 `x` 像素，并向下滚动 `y` 像素。允许负值。

`win.scrollTo(x, y)`

将窗口滚动到给定坐标 `(x, y)`。

`elem.scrollIntoView(top = true)`

滚动窗口，使 `elem` 显示在 `elem.scrollIntoView(false)` 的顶部（默认）或底部。

这里也有 `window.onscroll` 事件。

弹窗的聚焦/失焦

从理论上讲，使用 `window.focus()` 和 `window.blur()` 方法可以使窗口获得或失去焦点。此外，这里还有 `focus/blur` 事件，可以聚焦窗口并捕获访问者切换到其他地方的瞬间。

在过去，恶意网站滥用了这些方法。例如，看这段代码：

```
window.onblur = () => window.focus();
```

当用户尝试从窗口切换出去（`blur`）时，这段代码又让窗口重新获得了焦点。目的是将用户“锁定”在 `window` 中。

因此，就有了禁用此类代码的措施。保护用户免受广告和恶意页面的侵害的限制有很多。这取决于浏览器。

例如，移动端浏览器通常会完全忽略这种调用。并且，当弹窗是在单独的选项卡而不是新窗口中打开时，也无法进行聚焦。

尽管如此，还是有一些事情可以使用它们来完成。

例如：

- 当我们打开一个弹窗时，在它上面执行 `newWindow.focus()` 是个好主意。以防万一，对于某些操作系统/浏览器组合（combination），它可以确保用户现在位于新窗口中。
- 如果我们想要跟踪访问者何时在实际使用我们的 Web 应用程序，我们可以跟踪 `window.onfocus/onblur`。这使我们可以暂停/恢复页面活动和动画等。但是请注意，`blur` 事件意味着访问者从窗口切换了出来，但他们仍然可以观察到它。窗口处在背景中，但可能仍然是可见的。

总结

弹窗很少使用，因为有其他选择：在页面内或在 `iframe` 中加载和显示信息。

如果我们要打开一个弹窗，将其告知用户是一个好的实践。在链接或按钮附近的“打开窗口”图标可以让用户免受焦点转移的困扰，并使用户知道点击它会弹出一个新窗口。

- 可以通过 `open(url, name, params)` 调用打开一个弹窗。它会返回对新打开的窗口的引用。
- 浏览器会阻止来自用户行为之外的代码中的 `open` 调用。通常会显示一条通知，以便用户可以允许它们。
- 默认情况下，浏览器会打开一个新标签页，但如果提供了窗口大小，那么浏览器将打开一个弹窗。
- 弹窗可以使用 `window.opener` 属性访问 `opener` 窗口（译注：即打开弹窗的窗口）。
- 如果主窗口和弹窗同源，那么它们可以彼此自由地读取和修改。否则，它们可以更改彼此的地址（`location`），[交换消息](#)。

要关闭弹窗：使用 `close()` 调用。用户也可以关闭弹窗（就像任何其他窗口一样）。关闭之后，`window.closed` 为 `true`。

- `focus()` 和 `blur()` 方法允许聚焦/失焦于窗口。但它们并不是一直都有效。
- `focus` 和 `blur` 事件允许跟踪窗口的切换。但是请注意，在 `blur` 之后，即使窗口在背景状态下，窗口仍有可能是可见的。

跨窗口通信

“同源（Same Origin）”策略限制了窗口（window）和 frame 之间的相互访问。

这个想法出于这样的考虑，如果一个用户有两个打开的页面：一个来自 `john-smith.com`，另一个是 `gmail.com`，那么用户将不希望 `john-smith.com` 的脚本可以读取 `gmail.com` 中的邮件。所以，“同源”策略的目的是保护用户免遭信息盗窃。

同源

如果两个 URL 具有相同的协议，域和端口，则称它们是“同源”的。

以下几个 URL 都是同源的：

- `http://site.com`
- `http://site.com/`
- `http://site.com/my/page.html`

但是下面这几个不是：

- `http://www.site.com` (另一个域：`www`，影响)
- `http://site.org` (另一个域：`.org` 影响)
- `https://site.com` (另一个协议：`https`)
- `http://site.com:8080` (另一个端口：`8080`)

“同源”策略规定：

- 如果我们有对另外一个窗口（例如，一个使用 `window.open` 创建的弹窗，或者一个窗口中的 `iframe`）的引用，并且该窗口是同源的，那么我们就具有对该窗口的全部访问权限。
- 否则，如果该窗口不是同源的，那么我们就无法访问该窗口中的内容：变量，文档，任何东西。唯一的例外是 `location`：我们可以修改它（进而重定向用户）。但是我们无法读取 `location`（因此，我们无法看到用户当前所处的位置，也就不会泄漏任何信息）。

实例：`iframe`

一个 `<iframe>` 标签承载了一个单独的嵌入的窗口，它具有自己的 `document` 和 `window`。

我们可以使用以下属性访问它们：

- `iframe.contentWindow` 来获取 `<iframe>` 中的 `window`。
- `iframe.contentDocument` 来获取 `<iframe>` 中的 `document`，是 `iframe.contentWindow.document` 的简写形式。

当我们访问嵌入的窗口中的东西时，浏览器会检查 `iframe` 是否具有相同的源。如果不是，则会拒绝访问（对 `location` 进行写入是一个例外，它是会被允许的）。

例如，让我们尝试对来自另一个源的 `<iframe>` 进行读取和写入：

```
<iframe src="https://example.com" id="iframe"></iframe>

<script>
  iframe.onload = function() {
    // 我们可以获取对内部 window 的引用
    let iframeWindow = iframe.contentWindow; // OK
    try {
      // ...但是无法获取其中的文档
      let doc = iframe.contentDocument; // ERROR
    } catch(e) {
      alert(e); // Security Error (另一个源)
    }

    // 并且，我们也无法 **读取** iframe 中页面的 URL
    try {
      // 无法从 location 对象中读取 URL
      let href = iframe.contentWindow.location.href; // ERROR
    } catch(e) {
      alert(e); // Security Error
    }

    // ...我们可以 **写入** location (所以，在 iframe 中加载了其他内容) !
    iframe.contentWindow.location = '/'; // OK

    iframe.onload = null; // 清空处理程序，在 location 更改后不要再运行它
  };
</script>
```

上述代码除了以下操作都会报错：

- 通过 `iframe.contentWindow` 获取对内部 `window` 的引用——这是被允许的。
- 对 `location` 进行写入

与此相反，如果 `<iframe>` 具有相同的源，我们可以使用它做任何事情：

```
<!-- 来自同一个网站的 iframe -->
<iframe src="/" id="iframe"></iframe>

<script>
  iframe.onload = function() {
    // 可以做任何事儿
    iframe.contentDocument.body.prepend("Hello, world!");
  };
</script>
```

i `iframe.onload` vs `iframe.contentWindow.onload`

`iframe.onload` 事件（在 `<iframe>` 标签上）与 `iframe.contentWindow.onload`（在嵌入的 `window` 对象上）基本相同。当嵌入的窗口的所有资源都完全加载完毕时触发。

.....但是，我们无法使用 `iframe.contentWindow.onload` 访问不同源的 `iframe`。因此，请使用 `iframe.onload`，

子域上的 `window: document.domain`

根据定义，两个具有不同域的 `URL` 具有不同的源。

但是，如果窗口的二级域相同，例如 `john.site.com`, `peter.site.com` 和 `site.com`（它们共同的二级域是 `site.com`），我们可以使浏览器忽略该差异，使得它们可以被作为“同源”的来对待，以便进行跨窗口通信。

为了做到这一点，每个这样的窗口都应该执行下面这行代码：

```
document.domain = 'site.com';
```

这样就可以了。现在它们可以无限制地进行交互了。但是再强调一遍，这仅适用于具有相同二级域的页面。

Iframe：错误文档陷阱

当一个 `iframe` 来自同一个源时，我们可能会访问其 `document`，但是这里有一个陷阱。它与跨源无关，但你一定要知道。

在创建 `iframe` 后，`iframe` 会立即就拥有了一个文档。但是该文档不同于加载到其中的文档！

因此，如果我们要立即对文档进行操作，就可能出问题。

看一下下面这段代码：

```
<iframe src="/" id="iframe"></iframe>

<script>
  let oldDoc = iframe.contentDocument;
  iframe.onload = function() {
    let newDoc = iframe.contentDocument;
    // 加载的文档与初始的文档不同!
    alert(oldDoc == newDoc); // false
  };
</script>
```

我们不应该对尚未加载完成的 `iframe` 的文档进行处理，因为那是 **错误的文档**。如果我们在其上设置了任何事件处理程序，它们将会被忽略。

如果检测文档加载完成的时刻呢？

正确的文档是在 `iframe.onload` 触发时就会。但是，只有在整个 `iframe` 的所有资源都加载完成时，`iframe.onload` 才会触发。

我们可以尝试通过在 `setInterval` 中进行检查，以更早地捕获该时刻：

```
<iframe src="/" id="iframe"></iframe>

<script>
  let oldDoc = iframe.contentDocument;

  // 每 100ms 检查一次文档是否为新文档
  let timer = setInterval(() => {
    let newDoc = iframe.contentDocument;
    if (newDoc == oldDoc) return;

    alert("New document is here!");

    clearInterval(timer); // 取消 setInterval，不再需要它做任何事儿
  }, 100);
</script>
```

集合：`window.frames`

获取 `<iframe>` 的 `window` 对象的另一个方式是从命名集合 `window.frames` 中获取：

- 通过索引获取：`window.frames[0]` —— 文档中的第一个 `iframe` 的 `window` 对象。
- 通过名称获取：`window.frames iframeName` —— 获取 `name="iframeName"` 的 `iframe` 的 `window` 对象。

例如：

```
<iframe src="/" style="height:80px" name="win" id="iframe"></iframe>

<script>
  alert(iframe.contentWindow == frames[0]); // true
  alert(iframe.contentWindow == frames.win); // true
</script>
```

一个 `iframe` 内可能嵌套了其他的 `iframe`。相应的 `window` 对象会形成一个层次结构 (`hierarchy`)。

可以通过以下方式获取：

- `window.frames` —— “子”窗口的集合（用于嵌套的 `iframe`）。
- `window.parent` —— 对“父”（外部）窗口的引用。
- `window.top` —— 对最顶级父窗口的引用。

例如：

```
window.frames[0].parent === window; // true
```

我们可以使用 `top` 属性来检查当前的文档是否是在 `iframe` 内打开的：

```
if (window == top) { // 当前 window == window.top?
  alert('The script is in the topmost window, not in a frame');
} else {
  alert('The script runs in a frame!');
```

“sandbox” `iframe` 特性

`sandbox` 特性（attribute）允许在 `<iframe>` 中禁止某些特定行为，以防止其执行不被信任的代码。它通过将 `iframe` 视为非同源的，或者应用其他限制来实现 `iframe` 的“沙盒化”。

对于 `<iframe sandbox src="...">`，有一个应用于其上的默认的限制集。但是，我们可以通过提供一个以空格分隔的限制列表作为特性的值，来放宽这些限制，该列表中的各项为不应该应用于这个 `iframe` 的限制，例如：`<iframe sandbox="allow-forms allow-popups">`。

换句话说，一个空的 `"sandbox"` 特性会施加最严格的限制，但是我们用一个以空格分隔的列表，列出要移除的限制。

以下是限制的列表：

allow-same-origin

默认情况下，"sandbox" 会为 `iframe` 强制实施“不同来源”的策略。换句话说，它使浏览器将 `iframe` 视为来自另一个源，即使其 `src` 指向的是同一个网站也是如此。具有所有隐含的脚本限制。此选项会移除这些限制。

allow-top-navigation

允许 `iframe` 更改 `parent.location`。

allow-forms

允许在 `iframe` 中提交表单。

allow-scripts

允许在 `iframe` 中运行脚本。

allow-popups

允许在 `iframe` 中使用 `window.open` 打开弹窗。

查看 [官方手册](#) 获取更多内容。

下面的示例演示了一个具有默认限制集的沙盒 `iframe`: `<iframe sandbox src="...>`。它有一些 JavaScript 代码和一个表单。

请注意，这里没有东西会运行。可见默认设置非常苛刻：

<https://plnkr.co/edit/ucOI96Jl5kfG8h2Y?p=preview>

i 请注意：

"sandbox" 特性的目的仅是 **添加更多** 限制。它无法移除这些限制。尤其是，如果 `iframe` 来自其他源，则无法放宽同源策略。

跨窗口通信

`postMessage` 接口允许窗口之间相互通信，无论它们来自什么源。

因此，这是解决“同源”策略的方式之一。它允许来自于 `john-smith.com` 的窗口与来自于 `gmail.com` 的窗口进行通信，并交换信息，但前提是它们双方必须均同意并调用相应的 JavaScript 函数。这可以保护用户的安全。

这个接口有两个部分。

postMessage

想要发送消息的窗口需要调用接收窗口的 `postMessage` 方法。换句话说，如果我们想把消息发送给 `win`，我们应该调用 `win.postMessage(data, targetOrigin)`。

参数：

data

要发送的数据。可以是任何对象，数据会被通过使用“结构化克隆算法”进行克隆。IE 浏览器只支持字符串，因此我们需要对复杂的对象调用 `JSON.stringify` 方法进行处理，以支持该浏览器。

targetOrigin

指定目标窗口的源，以便只有来自给定的源的窗口才能获得该消息。

`targetOrigin` 是一种安全措施。请记住，如果目标窗口是非同源的，我们无法在发送方窗口读取它的 `location`。因此，我们无法确定当前在预期的窗口中打开的是哪个网站：用户随时可以导航离开，并且发送方窗口对此一无所知。

指定 `targetOrigin` 可以确保窗口仅在当前仍处于正确的网站时接收数据。在有敏感数据时，这非常重要。

例如，这里的 `win` 仅在它拥有来自 `http://example.com` 这个源的文档时，才会接收消息：

```
<iframe src="http://example.com" name="example">

<script>
  let win = window.frames.example;

  win.postMessage("message", "http://example.com");
</script>
```

如果我们不希望做这个检查，可以将 `targetOrigin` 设置为 `*`。

```
<iframe src="http://example.com" name="example">

<script>
  let win = window.frames.example;

  win.postMessage("message", "*");
</script>
```

onmessage

为了接收消息，目标窗口应该在 `message` 事件上有一个处理程序。当 `postMessage` 被调用时触发该事件（并且 `targetOrigin` 检查成功）。

`event` 对象具有特殊属性：

data

从 `postMessage` 传递来的数据。

origin

发送方的源，例如 `http://javascript.info`。

source

对发送方窗口的引用。如果我们想，我们可以立即 `source.postMessage(...)` 回去。

要为 `message` 事件分配处理程序，我们应该使用 `addEventListener`，简短的语法 `window.onmessage` 不起作用。

这里有一个例子：

```
window.addEventListener("message", function(event) {
  if (event.origin != 'http://javascript.info') {
    // 来自未知的源的内容，我们忽略它
    return;
  }

  alert( "received: " + event.data );

  // 可以使用 event.source.postMessage(...) 向回发送消息
});
```

完整示例：

<https://plnkr.co/edit/mTT1G37hpkrYSBNt?p=preview>

总结

要调用另一个窗口的方法或者访问另一个窗口的内容，我们应该首先拥有对其的引用。

对于弹窗，我们有两个引用：

- 从打开窗口的 (`opener`) 窗口：`window.open` —— 打开一个新的窗口，并返回对它的引用，
- 从弹窗：`window.opener` —— 是从弹窗中对打开此弹窗的窗口 (`opener`) 的引用。

对于 `iframe`，我们可以使用以下方式访问父/子窗口：

- `window.frames` —— 一个嵌套的 `window` 对象的集合,
- `window.parent`, `window.top` 是对父窗口和顶级窗口的引用,
- `iframe.contentWindow` 是 `<iframe>` 标签内的 `window` 对象。

如果几个窗口的源相同（域，端口，协议），那么这几个窗口可以彼此进行所需的操作。

否则，只能进行以下操作：

- 更改另一个窗口的 `location` (只能写入)。
- 向其发送一条消息。

例外情况：

- 对于二级域相同的窗口：`a.site.com` 和 `b.site.com`。通过在这些窗口中均设置 `document.domain='site.com'`，可以使它们处于“同源”状态。
- 如果一个 `iframe` 具有 `sandbox` 特性 (attribute)，则它会被强制处于“非同源”状态，除非在其特性值中指定了 `allow-same-origin`。这可用于在同一网站的 `iframe` 中运行不受信任的代码。

`postMessage` 接口允许两个具有任何源的窗口之间进行通信：

1. 发送方调用 `targetWin.postMessage(data, targetOrigin)`。
2. 如果 `targetOrigin` 不是 '*'，那么浏览器会检查窗口 `targetWin` 是否具有源 `targetOrigin`。
3. 如果它具有，`targetWin` 会触发具有特殊的属性的 `message` 事件：
 - `origin` —— 发送方窗口的源（比如 `http://my.site.com`）。
 - `source` —— 对发送方窗口的引用。
 - `data` —— 数据，可以是任何对象。但是 IE 浏览器只支持字符串，因此我们需要对复杂的对象调用 `JSON.stringify` 方法进行处理，以支持该浏览器。

我们应该使用 `addEventListener` 来在目标窗口中设置 `message` 事件的处理程序。

点击劫持攻击

“点击劫持”攻击允许恶意页面 **以用户的名义** 点击“受害网站”。

许多网站都被黑客以这种方式攻击过，包括 Twitter、Facebook 和 Paypal 等许多网站。当然，它们都已经被修复了。

原理

原理十分简单。

我们以 Facebook 为例，解释点击劫持是如何完成的：

1. 访问者被恶意页面吸引。怎样吸引的不重要。
2. 页面上有一个看起来无害的链接（例如：“变得富有”或者“点我，超好玩！”）。
3. 恶意页面在该链接上方放置了一个透明的 `<iframe>`，其 `src` 来自于 `facebook.com`，这使得“点赞”按钮恰好位于该链接上面。这通常是通过 `z-index` 实现的。
4. 用户尝试点击该链接时，实际上点击的是“点赞”按钮。

示例

这是恶意页面看起来的样子。为了清楚起见，我们将 `<iframe>` 设置成了半透明的（在真正的恶意页面中，它是全透明的）：

```
<style>
  iframe { /* 来自受害网站的 iframe */
    width: 400px;
    height: 100px;
    position: absolute;
    top: 0; left: -20px;
    opacity: 0.5; /* 在实际中为 opacity:0 */
    z-index: 1;
  }
</style>

<div>点击即可变得富有: </div>

<!-- 来自受害网站的 url -->
<iframe src="/clickjacking/facebook.html"></iframe>

<button>点这里！</button>

<div>.....你很酷（我实际上是一名帅气的黑客）！</div>
```

完整的攻击示例如下：

[https://plnkr.co/edit/dXu26rm0sghlN94x?p=preview ↗](https://plnkr.co/edit/dXu26rm0sghlN94x?p=preview)

在上面这个示例中，我们有一个半透明的 `<iframe src="facebook.html">`，我们可以看到，它位于按钮之上。点击按钮实际上会点击在 `iframe` 上，但这对用户不可见，因为 `iframe` 是透明的。

结果，如果访问者登陆了 Facebook（“记住我”通常是打开的），那么这个行为就会点一个“赞”。Twitter 上是“Follow”按钮。

下面是相同的示例，但 `iframe` 的透明度设置为了 `opacity:0`，更符合实际情况：

[https://plnkr.co/edit/4UWTfdySPyn4riiH?p=preview ↗](https://plnkr.co/edit/4UWTfdySPyn4riiH?p=preview)

我们进行攻击所需要的——就是将 `<iframe>` 放置在恶意页面中，使得按钮恰好位于链接的正上方。这样当用户点击链接时，他们实际上点击的是按钮。这通常可以通过 CSS 实现。

➊ 点击劫持是对点击事件，而非键盘事件

此攻击仅影响鼠标行为（或者类似的行为，例如在手机上的点击）。

键盘输入很难重定向。从技术上讲，我们可以用 `iframe` 的文本区域覆盖原有的文本区域实现攻击。因此，当访问者试图聚焦页面中的输入时，实际上聚焦的是 `iframe` 中的输入。

但是这里有个问题。访问者键入的所有内容都会被隐藏，因为该 `iframe` 是不可见的。

当用户无法在屏幕上看到自己输入的字符时，通常会停止打字。

传统防御（弱 ⚡）

最古老的防御措施是一段用于禁止在 `frame` 中打开页面的 JavaScript 代码（所谓的“framebusting”）。

它看起来像这样：

```
if (top != window) {
  top.location = window.location;
}
```

意思是说：如果 `window` 发现它不在顶部，那么它将自动使其自身位于顶部。

这个方法并不可靠，因为有许多方式可以绕过这个限制。下面我们就介绍几个。

阻止顶级导航

我们可以阻止因更改 `beforeunload` 事件处理程序中的 `top.location` 而引起的过渡（transition）。

顶级页面（从属于黑客）在 `beforeunload` 上设置了一个用于阻止的处理程序，像这样：

```
window.onbeforeunload = function() {
  return false;
};
```

当 `iframe` 试图更改 `top.location` 时，访问者会收到一条消息，询问他们是否要离开页面。

在大多数情况下，访问者会做出否定的回答，因为他们并不知道还有这么一个 `iframe`，他们所看到的只有顶级页面，他们没有理由离开。所以 `top.location` 不会变化！

演示示例：

[https://plnkr.co/edit/zKtJHiM53OYIFZwJ?p=preview ↗](https://plnkr.co/edit/zKtJHiM53OYIFZwJ?p=preview)

Sandbox 特性

`sandbox` 特性的限制之一就是导航。沙箱化的 `iframe` 不能更改 `top.location`。

但我们可以添加具有 `sandbox="allow-scripts allow-forms"` 的 `iframe`。从而放开限制，允许脚本和表单。但我们没添加 `allow-top-navigation`，因此更改 `top.location` 是被禁止的。

代码如下：

```
<iframe sandbox="allow-scripts allow-forms" src="facebook.html"></iframe>
```

还有其他方式可以绕过这个弱鸡防御。

X-Frame-Options

服务器端 header `X-Frame-Options` 可以允许或禁止在 `frame` 中显示页面。

它必须被完全作为 HTTP-header 发送：如果浏览器在 HTML `<meta>` 标签中找到它，则会忽略它。因此，`<meta http-equiv="X-Frame-Options" ...>` 没有任何作用。

这个 header 可能包含 3 个值：

DENY

始终禁止在 `frame` 中显示此页面。

SAMEORIGIN

允许在和父文档同源的 `frame` 中显示此页面。

ALLOW-FROM domain

允许在来自给定域的父文档的 `frame` 中显示此页面。

例如，Twitter 使用的是 `X-Frame-Options: SAMEORIGIN`。

显示禁用的功能

`X-Frame-Options` 有一个副作用。其他的网站即使有充分的理由也无法在 `frame` 中显示我们的页面。

因此，还有其他解决方案……例如，我们可以用一个样式为 `height: 100%; width: 100%;` 的 `<div>` “覆盖”页面，这样它就能拦截所有点击。如果 `window == top` 或者我们确定不需要保护时，再将该 `<div>` 移除。

像这样：

```
<style>
#protector {
    height: 100%;
    width: 100%;
    position: absolute;
    left: 0;
    top: 0;
    z-index: 99999999;
}
</style>

<div id="protector">
    <a href="/" target="_blank">前往网站</a>
</div>

<script>
// 如果顶级窗口来自其他源，这里则会出现一个 error
// 但是在本例中没有问题
if (top.document.domain == document.domain) {
    protector.remove();
}
</script>
```

演示示例：

[https://plnkr.co/edit/GbBFZGhpcOzAqDuX?p=preview ↗](https://plnkr.co/edit/GbBFZGhpcOzAqDuX?p=preview)

Samesite cookie 特性

`samesite` cookie 特性也可以阻止点击劫持攻击。

具有 `samesite` 特性的 cookie 仅在网站是通过直接方式打开（而不是通过 `frame` 或其他方式）的情况下才发送到网站。更多细节请见 [Cookie, document.cookie](#)。

如果网站，例如 Facebook，在其身份验证 cookie 中具有 `samesite` 特性，像这样：

```
Set-Cookie: authorization=secret; samesite
```

.....那么，当在另一个网站中的 `iframe` 中打开 Facebook 时，此类 `cookie` 将不会被发送。因此，攻击将失败。

当不实用 `cookie` 时，`samesite` `cookie` 特性将不会有任何影响。这可以使其他网站能够轻松地在 `iframe` 中显示我们公开的、未进行身份验证的页面。

然而，这也可能会使得劫持攻击在少数情况下起作用。例如，通过检查 IP 地址来防止重复投票的匿名投票网站仍然会受到点击劫持的攻击，因为它不使用 `cookie` 对用户身份进行验证。

总结

点击劫持是一种“诱骗”用户在不知情的情况下点击恶意网站的方式。如果是重要的点击操作，这是非常危险的。

黑客可以通过信息发布指向他的恶意页面的链接，或者通过某些手段引诱访问者访问他的页面。当然还有很多其他变体。

一方面——这种攻击方式是“浅层”的：黑客所做的只是拦截一次点击。但另一方面，如果黑客知道在点击之后将出现另一个控件，则他们可能还会使用狡猾的消息来迫使用户也点击它们。

这种攻击相当危险，因为在设计交互界面时，我们通常不会考虑到可能会有黑客代表用户点击界面。所以，在许多意想不到的地方可能发现攻击漏洞。

- 建议在那些不希望被在 `frame` 中查看的页面上（或整个网站上）使用 `X-Frame-Options: SAMEORIGIN`。
- 如果我们希望允许在 `frame` 中显示我们的页面，那我们使用一个 `<div>` 对整个页面进行遮盖，这样也是安全的。

二进制数据，文件

使用 JavaScript 处理二进制数据和文件。

ArrayBuffer，二进制数组

在 Web 开发中，当我们处理文件时（创建，上传，下载），经常会遇到二进制数据。另一个典型的应用场景是图像处理。

这些都可以通过 JavaScript 进行处理，而且二进制操作性能更高。

不过，在 JavaScript 中有很多种二进制数据格式，会有点容易混淆。仅举几个例子：

- `ArrayBuffer`, `Uint8Array`, `DataView`, `Blob`, `File` 及其他。

与其他语言相比，JavaScript 中的二进制数据是以非标准方式实现的。但是，当我们理清楚以后，一切就会变得相当简单了。

基本的二进制对象是 `ArrayBuffer` —— 对固定长度的连续内存空间的引用。

我们这样创建它：

```
let buffer = new ArrayBuffer(16); // 创建一个长度为 16 的 buffer  
alert(buffer.byteLength); // 16
```

它会分配一个 16 字节的连续内存空间，并用 0 进行预填充。

⚠️ **ArrayBuffer** 不是某种东西的数组

让我们先澄清一个可能的误区。**ArrayBuffer** 与 **Array** 没有任何共同之处：

- 它的长度是固定的，我们无法增加或减少它的长度。
- 它正好占用了内存中的那么多空间。
- 要访问单个字节，需要另一个“视图”对象，而不是 `buffer[index]`。

ArrayBuffer 是一个内存区域。它里面存储了什么？无从判断。只是一个原始的字节序列。

如要操作 **ArrayBuffer**，我们需要使用“视图”对象。

视图对象本身并不存储任何东西。它是一副“眼镜”，透过它来解释存储在 **ArrayBuffer** 中的字节。

例如：

- **Uint8Array** —— 将 **ArrayBuffer** 中的每个字节视为 0 到 255 之间的单个数字（每个字节是 8 位，因此只能容纳那么多）。这称为“8 位无符号整数”。
- **Uint16Array** —— 将每 2 个字节视为一个 0 到 65535 之间的整数。这称为“16 位无符号整数”。
- **Uint32Array** —— 将每 4 个字节视为一个 0 到 4294967295 之间的整数。这称为“32 位无符号整数”。
- **Float64Array** —— 将每 8 个字节视为一个 5.0×10^{-324} 到 1.8×10^{308} 之间的浮点数。

因此，一个 16 字节 **ArrayBuffer** 中的二进制数据可以解释为 16 个“小数字”，或 8 个更大的数字（每个数字 2 个字节），或 4 个更大的数字（每个数字 4 个字节），或 2 个高精度的浮点数（每个数字 8 个字节）。

new ArrayBuffer(16)																
Uint8Array	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Uint16Array	0	1	2	3	4	5	6	7								
Uint32Array	0				1				2				3			
Float64Array	0								1							

`ArrayBuffer` 是核心对象，是所有的基础，是原始的二进制数据。

但是，如果我们要写入值或遍历它，基本上几乎所有操作——我们必须使用视图（`view`），例如：

```
let buffer = new ArrayBuffer(16); // 创建一个长度为 16 的 buffer

let view = new Uint32Array(buffer); // 将 buffer 视为一个 32 位整数的序列

alert(Uint32Array.BYTES_PER_ELEMENT); // 每个整数 4 个字节

alert(view.length); // 4, 它存储了 4 个整数
alert(view.byteLength); // 16, 字节中的大小

// 让我们写入一个值
view[0] = 123456;

// 遍历值
for(let num of view) {
  alert(num); // 123456, 然后 0, 0, 0 (一共 4 个值)
}
```

TypedArray

所有这些视图（`Uint8Array`，`Uint32Array` 等）的通用术语是 `TypedArray`。它们都享有同一组方法和属性。

请注意，没有名为 `TypedArray` 的构造器，它只是表示 `ArrayBuffer` 上的视图之一的通用总称术语：`Int8Array`，`Uint8Array` 及其他，很快就会有完整列表。

当你看到 `new TypedArray` 之类的内容时，它表示 `new Int8Array`、`new Uint8Array` 及其他中之一。

类型化数组的行为类似于常规数组：具有索引，并且是可迭代的。

一个类型化数组的构造器（无论是 `Int8Array` 或 `Float64Array`，都无关紧要），其行为各不相同，并且取决于参数类型。

参数有 5 种变体：

```
new TypedArray(buffer, [byteOffset], [length]);
new TypedArray(object);
new TypedArray(typedArray);
new TypedArray(length);
new TypedArray();
```

- 如果给定的是 `ArrayBuffer` 参数，则会在其上创建视图。我们已经用过该语法了。

可选，我们可以给定起始位置 `byteOffset`（默认为 0）以及 `length`（默认至 `buffer` 的末尾），这样视图将仅涵盖 `buffer` 的一部分。

- 如果给定的是 `Array`，或任何类数组对象，则会创建一个相同长度的类型化数组，并复制其内容。

我们可以使用它来预填充数组的数据：

```
let arr = new Uint8Array([0, 1, 2, 3]);
alert( arr.length ); // 4, 创建了相同长度的二进制数组
alert( arr[1] ); // 1, 用给定值填充了 4 个字节 (无符号 8 位整数)
```

- 如果给定的是另一个 `TypedArray`，也是如此：创建一个相同长度的类型化数组，并复制其内容。如果需要的话，数据在此过程中会被转换为新的类型。

```
let arr16 = new Uint16Array([1, 1000]);
let arr8 = new Uint8Array(arr16);
alert( arr8[0] ); // 1
alert( arr8[1] ); // 232, 试图复制 1000, 但无法将 1000 放进 8 位字节中 (详述见下文)。
```

- 对于数字参数 `length` —— 创建类型化数组以包含这么多元素。它的字节长度将是 `length` 乘以单个 `TypedArray.BYTES_PER_ELEMENT` 中的字节数：

```
let arr = new Uint16Array(4); // 为 4 个整数创建类型化数组
alert( Uint16Array.BYTES_PER_ELEMENT ); // 每个整数 2 个字节
alert( arr.byteLength ); // 8 (字节中的大小)
```

- 不带参数的情况下，创建长度为零的类型化数组。

我们可以直接创建一个 `TypedArray`，而无需提及 `ArrayBuffer`。但是，视图离不开底层的 `ArrayBuffer`，因此，除第一种情况（已提供 `ArrayBuffer`）外，其他所有情况都会自动创建 `ArrayBuffer`。

如要访问 `ArrayBuffer`，可以用以下属性：

- `arr.buffer` —— 引用 `ArrayBuffer`。
- `arr.byteLength` —— `ArrayBuffer` 的长度。

因此，我们总是可以从一个视图转到另一个视图：

```
let arr8 = new Uint8Array([0, 1, 2, 3]);  
  
// 同一数据的另一个视图  
let arr16 = new Uint16Array(arr8.buffer);
```

下面是类型化数组的列表：

- `Uint8Array`, `Uint16Array`, `Uint32Array` —— 用于 8、16 和 32 位的整数。
 - `Uint8ClampedArray` —— 用于 8 位整数，在赋值时便“固定”其值（见下文）。
- `Int8Array`, `Int16Array`, `Int32Array` —— 用于有符号整数（可以为负数）。
- `Float32Array`, `Float64Array` —— 用于 32 位和 64 位的有符号浮点数。

⚠ 没有 `int8` 或类似的单值类型

请注意，尽管有类似 `Int8Array` 这样的名称，但 JavaScript 中并没有像 `int`，或 `int8` 这样的单值类型。

这是合乎逻辑的，因为 `Int8Array` 不是这些单值的数组，而是 `ArrayBuffer` 上的视图。

越界行为

如果我们尝试将越界值写入类型化数组会出现什么情况？不会报错。但是多余的位被切除。

例如，我们尝试将 256 放入 `Uint8Array`。256 的二进制格式是 `100000000`（9 位），但 `Uint8Array` 每个值只有 8 位，因此可用范围为 0 到 255。

对于更大的数字，仅存储最右边的（低位有效）8 位，其余部分被切除：

8-bit integer
10000000 256

因此结果是 0。

257 的二进制格式是 100000001 (9 位)，最右边的 8 位会被存储，因此数组中会有 1：

8-bit integer
100000001 257

换句话说，该数字对 2^8 取模的结果被保存了下来。

示例如下：

```
let uint8array = new Uint8Array(16);

let num = 256;
alert(num.toString(2)); // 100000000 (二进制表示)

uint8array[0] = 256;
uint8array[1] = 257;

alert(uint8array[0]); // 0
alert(uint8array[1]); // 1
```

`Uint8ClampedArray` 在这方面比较特殊，它的表现不太一样。对于大于 255 的任何数字，它将保存为 255，对于任何负数，它将保存为 0。此行为对于图像处理很有用。

TypedArray 方法

`TypedArray` 具有常规的 `Array` 方法，但有个明显的例外。

我们可以遍历 (`iterate`)，`map`，`slice`，`find` 和 `reduce` 等。

但有几件事我们做不了：

- 没有 `splice` —— 我们无法“删除”一个值，因为类型化数组是缓冲区（`buffer`）上的视图，并且缓冲区（`buffer`）是固定的、连续的内存区域。我们所能做的就是分配一个零值。
- 无 `concat` 方法。

还有两种其他方法：

- `arr.set(fromArr, [offset])` 将 `fromArr` 中从 `offset`（默认为 0）开始的所有元素复制到 `arr`。
- `arr.subarray([begin, end])` 创建一个从 `begin` 到 `end`（不包括）相同类型的新视图。这类似于 `slice` 方法（同样也支持），但不复制任何内容——只是创建一个新视图，以对给定片段的数据进行操作。

有了这些方法，我们可以复制、混合类型化数组，从现有数组创建新数组，等。

DataView

`DataView` ↗ 是在 `ArrayBuffer` 上的一种特殊的超灵活“未类型化”视图。它允许以任何格式访问任何偏移量（`offset`）的数据。

- 对于类型的数组，构造器决定了其格式。整个数组应该是统一的。第 `i` 个数字是 `arr[i]`。
- 通过 `DataView`，我们可以使用 `.getUint8(i)` 或 `.getUint16(i)` 之类的方法访问数据。我们在调用方法时选择格式，而不是在构造的时候。

语法：

```
new DataView(buffer, [byteOffset], [byteLength])
```

- **buffer** —— 底层的 `ArrayBuffer`。与类型化数组不同，`DataView` 不会自行创建缓冲区（`buffer`）。我们需要事先准备好。
- **byteOffset** —— 视图的起始字节位置（默认为 0）。
- **byteLength** —— 视图的字节长度（默认至 `buffer` 的末尾）。

例如，这里我们从同一个 `buffer` 中提取不同格式的数字：

```
// 4 个字节的二进制数组，每个都是最大值 255
let buffer = new Uint8Array([255, 255, 255, 255]).buffer;

let dataView = new DataView(buffer);

// 在偏移量为 0 处获取 8 位数字
alert( dataView.getUint8(0) ); // 255
```

```
// 现在在偏移量为 0 处获取 16 位数字，它由 2 个字节组成，一起解析为 65535
alert( dataView.getInt16(0) ); // 65535 (最大的 16 位无符号整数)

// 在偏移量为 0 处获取 32 位数字
alert( dataView.getInt32(0) ); // 4294967295 (最大的 32 位无符号整数)

dataView.setInt32(0, 0); // 将 4 个字节的数字设为 0，即将所有字节都设为 0
```

当我们将混合格式的数据存储在同一缓冲区（`buffer`）中时，`DataView` 非常有用。例如，我们存储一个成对序列（16 位整数，32 位浮点数）。用 `DataView` 可以轻松访问它们。

总结

`ArrayBuffer` 是核心对象，是对固定长度的连续内存区域的引用。

几乎任何对 `ArrayBuffer` 的操作，都需要一个视图。

- 它可以是 `TypedArray`：
 - `Uint8Array`, `Uint16Array`, `Uint32Array` —— 用于 8 位、16 位和 32 位无符号整数。
 - `Uint8ClampedArray` —— 用于 8 位整数，在赋值时便“固定”其值。
 - `Int8Array`, `Int16Array`, `Int32Array` —— 用于有符号整数（可以为负数）。
 - `Float32Array`, `Float64Array` —— 用于 32 位和 64 位的有符号浮点数。
- 或 `DataView` —— 使用方法来指定格式的视图，例如，`getInt8(offset)`。

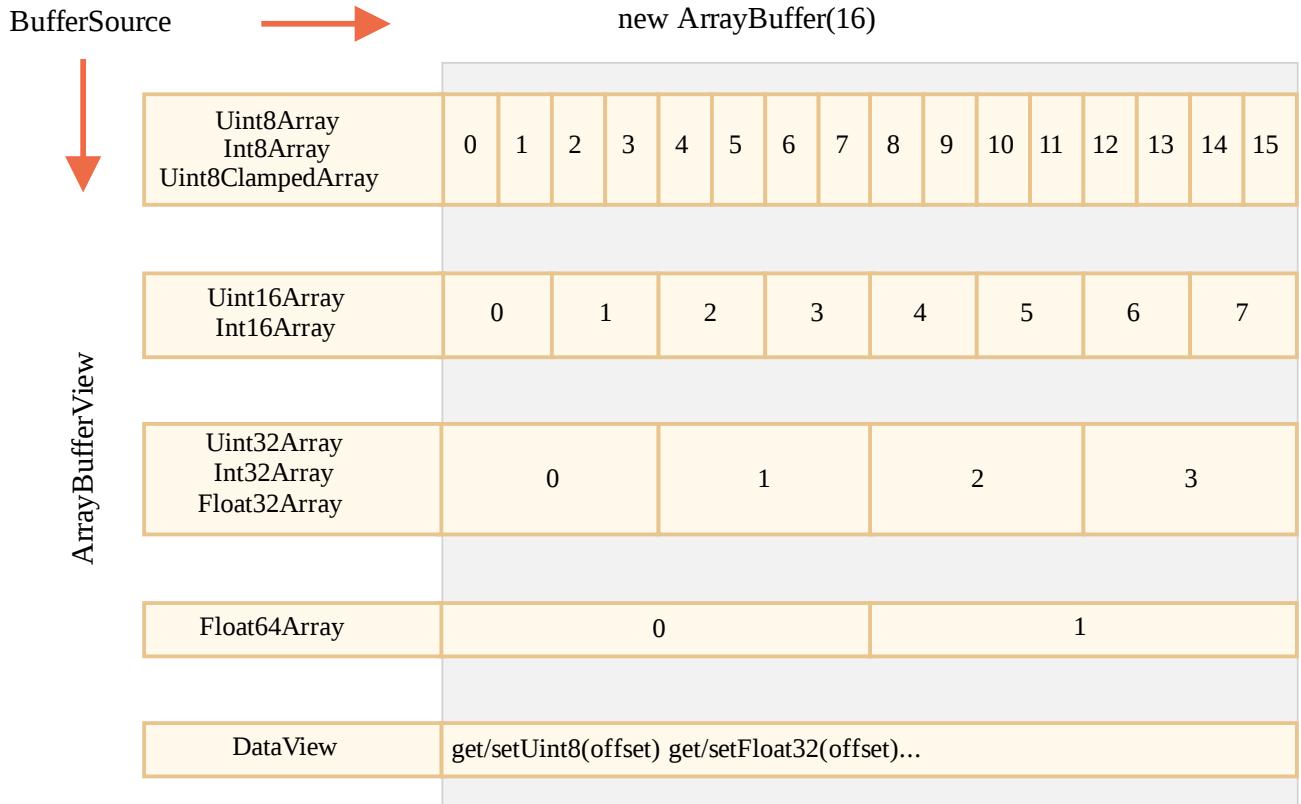
在大多数情况下，我们直接对类型化数组进行创建和操作，而将 `ArrayBuffer` 作为“通用标识符（common discriminator）”隐藏起来。我们可以通过 `.buffer` 来访问它，并在需要时创建另一个视图。

还有另外两个术语，用于对二进制数据进行操作的方法的描述：

- `ArrayBufferView` 是所有这些视图的总称。
- `BufferSource` 是 `ArrayBuffer` 或 `ArrayBufferView` 的总称。

我们将在下一章中学习这些术语。`BufferSource` 是最常用的术语之一，因为它的意思是“任何类型的二进制数据”—— `ArrayBuffer` 或其上的视图。

这是一份备忘单：



TextDecoder 和 TextEncoder

如果二进制数据实际上是一个字符串怎么办？例如，我们收到了一个包含文本数据的文件。

内建的 `TextDecoder` 对象在给定缓冲区（`buffer`）和编码格式（`encoding`）的情况下，能够将值读取到实际的 JavaScript 字符串中。

首先我们需要创建：

```
let decoder = new TextDecoder([label], [options]);
```

- **label** —— 编码格式，默认为 `utf-8`，但同时也支持 `big5`, `windows-1251` 等许多其他编码格式。
- **options** —— 可选对象：
 - **fatal** —— 布尔值，如果为 `true` 则为无效（不可解码）字符抛出异常，否则（默认）用字符 `\uFFFD` 替换无效字符。
 - **ignoreBOM** —— 布尔值，如果为 `true` 则 BOM（可选的字节顺序 unicode 标记），很少需要使用。

.....然后解码：

```
let str = decoder.decode([input], [options]);
```

- **input** —— 要被解码的 `BufferSource`。
- **options** —— 可选对象:
 - **stream** —— 对于解码流, 为 `true`, 则将传入的数据块 (`chunk`) 作为参数重复调用 `decoder`。在这种情况下, 多字节的字符可能偶尔会在块与块之间被分割。这个选项告诉 `TextDecoder` 记住“未完成”的字符, 并在下一个数据块来的时候进行解码。

例如:

```
let uint8Array = new Uint8Array([72, 101, 108, 108, 111]);  
alert( new TextDecoder().decode(uint8Array) ); // Hello
```

```
let uint8Array = new Uint8Array([228, 189, 160, 229, 165, 189]);  
alert( new TextDecoder().decode(uint8Array) ); // 你好
```

我们可以通过为其创建子数组视图来解码部分缓冲区:

```
let uint8Array = new Uint8Array([0, 72, 101, 108, 108, 111, 0]);  
  
// 该字符串位于中间  
// 在不复制任何内容的前提下, 创建一个新的视图  
let binaryString = uint8Array.subarray(1, -1);  
  
alert( new TextDecoder().decode(binaryString) ); // Hello
```

TextEncoder

`TextEncoder` ↗ 做相反的事情——将字符串转换为字节。

语法为:

```
let encoder = new TextEncoder();
```

只支持 `utf-8` 编码。

它有两种方法:

- **encode(str)** —— 从字符串返回 `Uint8Array`。
- **encodeInto(str, destination)** —— 将 `str` 编码到 `destination` 中，该目标必须为 `Uint8Array`。

```
let encoder = new TextEncoder();

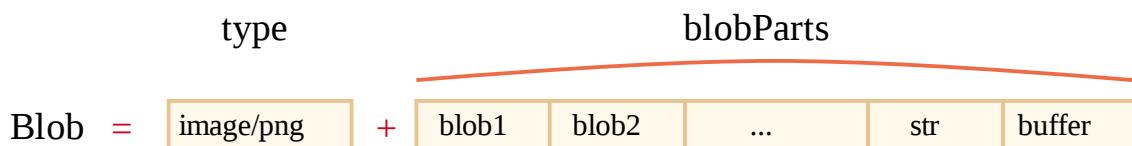
let uint8Array = encoder.encode("Hello");
alert(uint8Array); // 72,101,108,108,111
```

Blob

`ArrayBuffer` 和视图（`view`）都是 ECMA 标准的一部分，是 JavaScript 的一部分。

在浏览器中，还有其他更高级的对象，特别是 `Blob`，在 [File API ↗](#) 中有相关描述。

`Blob` 由一个可选的字符串 `type`（通常是 MIME 类型）和 `blobParts` 组成——一系列其他 `Blob` 对象，字符串和 `BufferSource`。



构造函数的语法为：

```
new Blob(blobParts, options);
```

- `blobParts` 是 `Blob/BufferSource/String` 类型的值的数组。
- `options` 可选对象：
 - `type` —— `Blob` 类型，通常是 MIME 类型，例如 `image/png`，
 - `endings` —— 是否转换换行符，使 `Blob` 对应于当前操作系统的换行符（`\r\n` 或 `\n`）。默认为 `"transparent"`（啥也不做），不过也可以是 `"native"`（转换）。

例如：

```
// 从字符串创建 Blob
let blob = new Blob(["<html>...</html>"], {type: 'text/html'});
// 请注意：第一个参数必须是一个数组 [...]
```

```
// 从类型化数组 (typed array) 和字符串创建 Blob
let hello = new Uint8Array([72, 101, 108, 108, 111]); // 二进制格式的 "hello"

let blob = new Blob([hello, ' ', 'world'], {type: 'text/plain'});
```

我们可以用 `slice` 方法来提取 `Blob` 片段:

```
blob.slice([byteStart], [byteEnd], [contentType]);
```

- `byteStart` —— 起始字节，默认为 0。
- `byteEnd` —— 最后一个字节（专有，默认为最后）。
- `contentType` —— 新 `blob` 的 `type`，默认与源 `blob` 相同。

参数值类似于 `array.slice`，也允许是负数。

i `Blob` 对象是不可改变的

我们无法直接在 `Blob` 中更改数据，但我们可以 `slice` 获得 `Blob` 的多个部分，从这些部分创建新的 `Blob` 对象，将它们组成新的 `Blob`，等。

这种行为类似于 JavaScript 字符串：我们无法更改字符串中的字符，但可以生成一个新的改动过的字符串。

Blob 用作 URL

`Blob` 可以很容易用作 `<a>`、`` 或其他标签的 `URL`，来显示它们的内容。

多亏了 `type`，让我们也可以下载/上传 `Blob` 对象，而在网络请求中，`type` 自然地变成了 `Content-Type`。

让我们从一个简单的例子开始。通过点击链接，你可以下载一个具有动态生成的内容为 `hello world` 的 `Blob` 的文件：

```
<!-- download 特性 (attribute) 强制浏览器下载而不是导航 -->
<a download="hello.txt" href="#" id="link">Download</a>

<script>
let blob = new Blob(["Hello, world!"], {type: 'text/plain'});

link.href = URL.createObjectURL(blob);
</script>
```

我们也可以在 **Javascript** 中动态创建一个链接，通过 `link.click()` 模拟一个点击，然后便自动下载了。

下面是类似的代码，此代码可以让用户无需任何 **HTML** 即可下载动态生成的 **Blob**（译注：也就是通过代码模拟用户点击，从而自动下载）：

```
let link = document.createElement('a');
link.download = 'hello.txt';

let blob = new Blob(['Hello, world!'], {type: 'text/plain'});

link.href = URL.createObjectURL(blob);

link.click();

URL.revokeObjectURL(link.href);
```

`URL.createObjectURL` 取一个 **Blob**，并为其创建一个唯一的 **URL**，形式为 `blob:<origin>/<uuid>`。

也就是 `link.href` 的值的样子：

```
blob:https://javascript.info/1e67e00e-860d-40a5-89ae-6ab0cbee6273
```

浏览器内部为每个通过 `URL.createObjectURL` 生成的 **URL** 存储了一个 **URL** → **Blob** 映射。因此，此类 **URL** 很短，但可以访问 **Blob**。

生成的 **URL**（即其链接）仅在当前文档打开的状态下才有效。它允许引用 ``、`<a>` 中的 **Blob**，以及基本上任何其他期望 **URL** 的对象。

不过它有个副作用。虽然这里有 **Blob** 的映射，但 **Blob** 本身只保存在内存中的。浏览器无法释放它。

在文档退出时（`unload`），该映射会被自动清除，因此 **Blob** 也相应被释放了。但是，如果应用程序寿命很长，那这个释放就不会很快发生。

因此，如果我们创建一个 **URL**，那么即使我们不再需要该 **Blob** 了，它也会被挂在内存中。

`URL.revokeObjectURL(url)` 从内部映射中移除引用，因此允许 **Blob** 被删除（如果没有其他引用的话），并释放内存。

在上面最后一个示例中，我们打算仅使用一次 **Blob**，来进行即时下载，因此我们立即调用 `URL.revokeObjectURL(link.href)`。

而在前一个带有可点击的 **HTML** 链接的示例中，我们不调用 `URL.revokeObjectURL(link.href)`，因为那样会使 **Blob** **URL** 无效。在调用

该方法后，由于映射被删除了，因此该 URL 也就不再起作用了。

Blob 转换为 base64

`URL.createObjectURL` 的一个替代方法是，将 `Blob` 转换为 `base64`-编码的字符串。

这种编码将二进制数据表示为一个由 0 到 64 的 ASCII 码组成的字符串，非常安全且“可读”。更重要的是——我们可以在“`data-url`”中使用此编码。

“`data-url`” 的形式为 `data:[<mediatype>][;base64],<data>`。我们可以在任何地方使用这种 `url`，和使用“常规” `url` 一样。

例如，这是一个笑脸：

```
` 元素来实现的：

- 使用 `canvas.drawImage` 在 `canvas` 上绘制图像（或图像的一部分）。
- 调用 `canvas` 方法 `.toBlob(callback, format, quality)` 创建一个 `Blob`，并在创建完成后使用其运行 `callback`。

在下面这个示例中，图像只是被复制了，不过我们可以在创建 `blob` 之前，从中裁剪图像，或者在 `canvas` 上对其进行转换：

```
// 获取任何图像
let img = document.querySelector('img');

// 生成同尺寸的 <canvas>
let canvas = document.createElement('canvas');
canvas.width = img.clientWidth;
canvas.height = img.clientHeight;

let context = canvas.getContext('2d');

// 向其中复制图像（此方法允许剪裁图像）
context.drawImage(img, 0, 0);
// 我们 context.rotate(), 并在 canvas 上做很多其他事情

// toBlob 是异步操作，结束后会调用 callback
canvas.toBlob(function(blob) {
 // blob 创建完成，下载它
 let link = document.createElement('a');
 link.download = 'example.png';

 link.href = URL.createObjectURL(blob);
 link.click();

 // 删除内部 blob 引用，这样浏览器可以从内存中将其清楚
 URL.revokeObjectURL(link.href);
}, 'image/png');
```

如果我们更喜欢 `async/await` 而不是 `callback`:

```
let blob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/png'));
```

对于页面截屏，我们可以使用诸如 <https://github.com/niklasvh/html2canvas> 之类的库。它所做的只是扫一遍浏览器页面，并将其绘制在 `<canvas>` 上。然后，我们就可以像上面一样获取一个它的 `Blob`。

## Blob 转换为 ArrayBuffer

`Blob` 构造器允许从几乎所有东西创建 `blob`，包括任何 `BufferSource`。

但是，如果我们需要执行低级别的操作的话，则可以使用 `FileReader` 从 `blob` 中获取最低级别的 `ArrayBuffer`：

```
// 从 blob 获取 arrayBuffer
let fileReader = new FileReader();

fileReader.readAsArrayBuffer(blob);

fileReader.onload = function(event) {
 let arrayBuffer = fileReader.result;
};

}
```

## 总结

`ArrayBuffer`, `Uint8Array` 及其他 `BufferSource` 是“二进制数据”，而 `Blob` 则表示“具有类型的二进制数据”。

这样可以方便 `Blob` 用于在浏览器中非常常见的上传/下载操作。

[XMLHttpRequest](#), 文章 "fetch-basics" 未找到 等进行 Web 请求的方法可以自然地使用 `Blob`，也可以使用其他类型的二进制数据。

我们可以轻松地在 `Blob` 和低级别的二进制数据类型之间进行转换：

- 我们可以使用 `new Blob(...)` 构造函数从一个类型化数组（`typed array`）创建 `Blob`。
- 我们可以使用 `FileReader` 从 `Blob` 中取回 `ArrayBuffer`，然后在其上创建一个视图（`view`），用于低级别的二进制处理。

## File 和 FileReader

`File` 对象继承自 `Blob`，并扩展了与文件系统相关的功能。

有两种方式可以获取它。

第一种，与 `Blob` 类似，有一个构造器：

```
new File(fileParts, fileName, [options])
```

- `fileParts` —— `Blob/BufferSource/String` 类型值的数组。
- `fileName` —— 文件名字符串。
- `options` —— 可选对象：
  - `lastModified` —— 最后一次修改的时间戳（整数日期）。

第二种，更常见的是，我们从 `<input type="file">` 或拖放或其他浏览器接口来获取文件。在这种情况下，`file` 将从操作系统（OS）获得 `this` 信息。

由于 `File` 是继承自 `Blob` 的，所以 `File` 对象具有相同的属性，附加：

- `name` —— 文件名，
- `lastModified` —— 最后一次修改的时间戳。

这就是我们从 `<input type="file">` 中获取 `File` 对象的方式：

```
<input type="file" onchange="showFile(this)">

<script>
function showFile(input) {
 let file = input.files[0];

 alert(`File name: ${file.name}`); // 例如 my.png
 alert(`Last modified: ${file.lastModified}`); // 例如 1552830408824
}
</script>
```

### ① 请注意：

输入 (`input`) 可以选择多个文件，因此 `input.files` 是一个类数组对象。这里我们只有一个文件，所以我们只取 `input.files[0]`。

## FileReader

`FileReader` 是一个对象，它的唯一目的是从 `Blob`（因此也从 `File`）对象中读取数据。

它使用事件来传递数据，因为从磁盘读取数据可能比较费时间。

构造函数：

```
let reader = new FileReader(); // 没有参数
```

主要方法:

- **readAsArrayBuffer(blob)** —— 将数据读取为二进制格式的 `ArrayBuffer`。
- **readAsText(blob, [encoding])** —— 将数据读取为给定编码（默认为 `utf-8` 编码）的文本字符串。
- **readAsDataURL(blob)** —— 读取二进制数据，并将其编码为 `base64` 的 `data url`。
- **abort()** —— 取消操作。

`read*` 方法的选择，取决于我们喜欢哪种格式，以及如何使用数据。

- **readAsArrayBuffer** —— 用于二进制文件，执行低级别的二进制操作。对于诸如切片（`slicing`）之类的高级别的操作，`File` 是继承自 `Blob` 的，所以我们可以直接调用它们，而无需读取。
- **readAsText** —— 用于文本文件，当我们想要获取字符串时。
- **readAsDataURL** —— 当我们想在 `src` 中使用此数据，并将其用于 `img` 或其他标签时。正如我们在 `Blob` 一章中所讲的，还有一种用于此的读取文件的替代方案：`URL.createObjectURL(file)`。

读取过程中，有以下事件:

- **loadstart** —— 开始加载。
- **progress** —— 在读取过程中出现。
- **load** —— 读取完成，没有 `error`。
- **abort** —— 调用了 `abort()`。
- **error** —— 出现 `error`。
- **loadend** —— 读取完成，无论成功还是失败。

读取完成后，我们可以通过以下方式访问读取结果:

- `reader.result` 是结果（如果成功）
- `reader.error` 是 `error`（如果失败）。

使用最广泛的事件无疑是 `load` 和 `error`。

这是一个读取文件的示例:

```
<input type="file" onchange="readFile(this)">

<script>
```

```
function readfile(input) {
 let file = input.files[0];

 let reader = new FileReader();

 reader.readAsText(file);

 reader.onload = function() {
 console.log(reader.result);
 };

 reader.onerror = function() {
 console.log(reader.error);
 };
}

</script>
```

### i `FileReader` 用于 blob

正如我们在 [Blob](#) 一章中所提到的，`FileReader` 不仅可读取文件，还可读取任何 blob。

我们可以使用它将 blob 转换为其他格式：

- `readAsArrayBuffer(blob)` —— 转换为 `ArrayBuffer`，
- `readAsText(blob, [encoding])` —— 转换为字符串（`TextDecoder` 的一个替代方案），
- `readAsDataURL(blob)` —— 转换为 base64 的 data url。

### i 在 Web Workers 中可以使用 `FileReaderSync`

对于 Web Worker，还有一种同步的 `FileReader` 变体，称为 `FileReaderSync` ↗。

它的读取方法 `read*` 不会生成事件，但是会像常规函数那样返回一个结果。

不过，这仅在 Web Worker 中可用，因为在读取文件的时候，同步调用会有延迟，而在 Web Worker 中，这种延迟并不是很重要。它不会影响页面。

## 总结

`File` 对象继承自 `Blob`。

除了 `Blob` 方法和属性外，`File` 对象还有 `name` 和 `lastModified` 属性，以及从文件系统读取的内部功能。我们通常从用户输入如 `<input>` 或拖放事件来获取 `File` 对象。

`FileReader` 对象可以从文件或 `blob` 中读取数据，可以读取为以下三种格式：

- 字符串 (`readAsText`) 。
- `ArrayBuffer` (`readAsArrayBuffer`) 。
- `data url, base-64 编码` (`readAsDataURL`) 。

但是，在很多情况下，我们不必读取文件内容。就像我们处理 `blob` 一样，我们可以使用 `URL.createObjectURL(file)` 创建一个短的 `url`，并将其赋给 `<a>` 或 `<img>`。这样，文件便可以下载文件或者将其呈现为图像，作为 `canvas` 等的一部分。

而且，如果我们要通过网络发送一个 `File`，那也很容易：像 `XMLHttpRequest` 或 `fetch` 等网络 API 本身就接受 `File` 对象。

## 网络请求

### Fetch

JavaScript 可以将网络请求发送到服务器，并在需要时加载新信息。

例如，我们可以使用网络请求来：

- 提交订单，
- 加载用户信息，
- 从服务器接收最新的更新，
- .....等。

.....所有这些都没有重新加载页面！

对于来自 JavaScript 的网络请求，有一个总称术语“**AJAX**”（**Asynchronous JavaScript And XML** 的简称）。但是，我们不必使用 `XML`：这个术语诞生于很久以前，所以这个词一直在那儿。

有很多方式可以向服务器发送网络请求，并从服务器获取信息。

`fetch()` 方法是一种现代通用的方法，那么我们就从它开始吧。旧版本的浏览器不支持它（可以 `polyfill`），但是它在现代浏览器中的支持情况很好。

基本语法：

```
let promise = fetch(url, [options])
```

- `url` —— 要访问的 URL。
- `options` —— 可选参数：`method`, `header` 等。

没有 `options`，那就是一个简单的 `GET` 请求，下载 `url` 的内容。

浏览器立即启动请求，并返回一个该调用代码应该用来获取结果的 `promise`。

获取响应通常需要经过两个阶段。

第一阶段，当服务器发送了响应头（**response header**），`fetch` 返回的 `promise` 就使用内建的 `Response ↗ class` 对象来对响应头进行解析。

在这个阶段，我们可以通过检查响应头，来检查 HTTP 状态以确定请求是否成功，当前还没有响应体（**response body**）。

如果 `fetch` 无法建立一个 HTTP 请求，例如网络问题，亦或是请求的网址不存在，那么 `promise` 就会 `reject`。异常的 HTTP 状态，例如 404 或 500，不会导致出现 `error`。

我们可以在 `response` 的属性中看到 HTTP 状态：

- `status` —— HTTP 状态码，例如 200。
- `ok` —— 布尔值，如果 HTTP 状态码为 200-299，则为 `true`。

例如：

```
let response = await fetch(url);

if (response.ok) { // 如果 HTTP 状态码为 200-299
 // 获取 response body (此方法会在下面解释)
 let json = await response.json();
} else {
 alert("HTTP-Error: " + response.status);
}
```

第二阶段，为了获取 **response body**，我们需要使用一个其他的方法调用。

`Response` 提供了多种基于 `promise` 的方法，来以不同的格式访问 `body`：

- `response.text()` —— 读取 `response`，并以文本形式返回 `response`，
- `response.json()` —— 将 `response` 解析为 JSON，
- `response.formData()` —— 以 `FormData` 对象（在 [下一章](#) 有解释）的形式返回 `response`，
- `response.blob()` —— 以 `Blob`（具有类型的二进制数据）形式返回 `response`，
- `response.arrayBuffer()` —— 以 `ArrayBuffer`（低级别的二进制数据）形式返回 `response`，
- 另外，`response.body` 是 [ReadableStream ↗ class](#) 对象，它允许你逐块读取 `body`，我们稍后会用一个例子解释它。

例如，我们从 GitHub 获取最新 commits 的 JSON 对象：

```
let url = 'https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits';
let response = await fetch(url);

let commits = await response.json(); // 读取 response body, 并将其解析为 JSON

alert(commits[0].author.login);
```

也可以使用纯 promise 语法，不使用 `await`：

```
fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits')
 .then(response => response.json())
 .then(commits => alert(commits[0].author.login));
```

要获取响应文本，可以使用 `await response.text()` 代替 `.json()`：

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits');
let text = await response.text(); // 将 response body 读取为文本

alert(text.slice(0, 80) + '...');
```

作为一个读取为二进制格式的演示示例，让我们 `fetch` 并显示一张 “[“fetch” 规范](#)” 中的图片（`Blob` 操作的有关内容请见 [Blob](#)）：

```
let response = await fetch('/article/fetch/logo-fetch.svg');

let blob = await response.blob(); // 下载为 Blob 对象

// 为其创建一个
let img = document.createElement('img');
img.style = 'position:fixed;top:10px;left:10px;width:100px';
document.body.append(img);

// 显示它
img.src = URL.createObjectURL(blob);

setTimeout(() => { // 3 秒后将其隐藏
 img.remove();
 URL.revokeObjectURL(img.src);
}, 3000);
```

## ⚠ 重要:

我们只能选择一种读取 `body` 的方法。

如果我们已经使用了 `response.text()` 方法来获取 `response`, 那么如果再用 `response.json()`, 则不会生效, 因为 `body` 内容已经被处理过了。

```
let text = await response.text(); // response body 被处理了
let parsed = await response.json(); // 失败 (已经被处理过了)
```

## Response header

Response header 位于 `response.headers` 中的一个类似于 Map 的 header 对象。

它不是真正的 Map, 但是它具有类似的方法, 我们可以按名称 (`name`) 获取各个 header, 或迭代它们:

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.ja
// 获取一个 header
alert(response.headers.get('Content-Type')); // application/json; charset=utf-8

// 迭代所有 header
for (let [key, value] of response.headers) {
 alert(` ${key} = ${value}`);
}
```

## Request header

要在 `fetch` 中设置 request header, 我们可以使用 `headers` 选项。它有一个带有输出 header 的对象, 如下所示:

```
let response = fetch(protectedUrl, {
 headers: {
 Authentication: 'secret'
 }
});
```

.....但是有一些我们无法设置的 header (详见 [forbidden HTTP headers ↗](#)) :

- `Accept-Charset`, `Accept-Encoding`
- `Access-Control-Request-Headers`

- `Access-Control-Request-Method`
- `Connection`
- `Content-Length`
- `Cookie`, `Cookie2`
- `Date`
- `DNT`
- `Expect`
- `Host`
- `Keep-Alive`
- `Origin`
- `Referer`
- `TE`
- `Trailer`
- `Transfer-Encoding`
- `Upgrade`
- `Via`
- `Proxy-*`
- `Sec-*`

这些 header 保证了 HTTP 的正确性和安全性，所以它们仅由浏览器控制。

## POST 请求

要创建一个 `POST` 请求，或者其他方法的请求，我们需要使用 `fetch` 选项：

- `method` —— HTTP 方法，例如 `POST`，
- `body` —— request body，其中之一：
  - 字符串（例如 JSON 编码的），
  - `FormData` 对象，以 `form/multipart` 形式发送数据，
  - `Blob`/`BufferSource` 发送二进制数据，
  - `URLSearchParams`，以 `x-www-form-urlencoded` 编码形式发送数据，很少使用。

JSON 形式是最常用的。

例如，下面这段代码以 JSON 形式发送 `user` 对象：

```
let user = {
 name: 'John',
```

```

 surname: 'Smith'
};

let response = await fetch('/article/fetch/post/user', {
 method: 'POST',
 headers: {
 'Content-Type': 'application/json; charset=utf-8'
 },
 body: JSON.stringify(user)
});

let result = await response.json();
alert(result.message);

```

请注意，如果请求的 `body` 是字符串，则 `Content-Type` 会默认设置为 `text/plain; charset=UTF-8`。

但是，当我们要发送 JSON 时，我们会使用 `headers` 选项来发送 `application/json`，这是 JSON 编码的数据的正确的 `Content-Type`。

## 发送图片

我们同样可以使用 `Blob` 或 `BufferSource` 对象通过 `fetch` 提交二进制数据。

例如，这里有一个 `<canvas>`，我们可以通过在其上移动鼠标来进行绘制。点击“submit”按钮将图片发送到服务器：

```

<body style="margin:0">
 <canvas id="canvasElem" width="100" height="80" style="border:1px solid"></canvas>

 <input type="button" value="Submit" onclick="submit()">

 <script>
 canvasElem.onmousemove = function(e) {
 let ctx = canvasElem.getContext('2d');
 ctx.lineTo(e.clientX, e.clientY);
 ctx.stroke();
 };

 async function submit() {
 let blob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/png'));
 let response = await fetch('/article/fetch/post/image', {
 method: 'POST',
 body: blob
 });

 // 服务器给出确认信息和图片大小作为响应
 let result = await response.json();
 alert(result.message);
 }
 </script>

```

```
</script>
</body>
```

请注意，这里我们没有手动设置 `Content-Type` header，因为 `Blob` 对象具有内建的类型（这里是 `image/png`，通过 `toBlob` 生成的）。对于 `Blob` 对象，这个类型就变成了 `Content-Type` 的值。

可以在不使用 `async/await` 的情况下重写 `submit()` 函数，像这样：

```
function submit() {
 canvasElem.toBlob(function(blob) {
 fetch('/article/fetch/post/image', {
 method: 'POST',
 body: blob
 })
 .then(response => response.json())
 .then(result => alert(JSON.stringify(result, null, 2)))
 }, 'image/png');
}
```

## 总结

典型的 `fetch` 请求由两个 `await` 调用组成：

```
let response = await fetch(url, options); // 解析 response header
let result = await response.json(); // 将 body 读取为 json
```

或者以 `promise` 形式：

```
fetch(url, options)
 .then(response => response.json())
 .then(result => /* process result */)
```

响应的属性：

- `response.status` —— `response` 的 HTTP 状态码，
- `response.ok` —— HTTP 状态码为 200-299，则为 `true`。
- `response.headers` —— 类似于 `Map` 的带有 HTTP header 的对象。

获取 response body 的方法:

- `response.text()` —— 读取 response，并以文本形式返回 response，
- `response.json()` —— 将 response 解析为 JSON 对象形式，
- `response.formData()` —— 以 `FormData` 对象（form/multipart 编码，参见下一章）的形式返回 response，
- `response.blob()` —— 以 `Blob`（具有类型的二进制数据）形式返回 response，
- `response.arrayBuffer()` —— 以 `ArrayBuffer`（低级别的二进制数据）形式返回 response。

到目前为止我们了解到的 `fetch` 选项:

- `method` —— HTTP 方法，
- `headers` —— 具有 `request header` 的对象（不是所有 `header` 都是被允许的）
- `body` —— 要以 `string`, `FormData`, `BufferSource`, `Blob` 或 `UrlSearchParams` 对象的形式发送的数据（`request body`）。

在下一章，我们将会看到更多 `fetch` 的选项和用例。

## FormData

这一章是关于发送 HTML 表单的：带有或不带文件，带有其他字段等。

`FormData` 对象可以提供帮助。你可能已经猜到了，它是表示 HTML 表单数据的对象。

构造函数是：

```
let formData = new FormData([form]);
```

如果提供了 HTML `form` 元素，它会自动捕获 `form` 元素字段。

`FormData` 的特殊之处在于网络方法（network methods），例如 `fetch` 可以接受一个 `FormData` 对象作为 `body`。它会被编码并发送出去，带有 `Content-Type: multipart/form-data`。

从服务器角度来看，它就像是一个普通的表单提交。

## 发送一个简单的表单

我们先来发送一个简单的表单。

正如你所看到的，它几乎就是一行代码：

```

<form id="formElem">
 <input type="text" name="name" value="John">
 <input type="text" name="surname" value="Smith">
 <input type="submit">
</form>

<script>
 formElem.onsubmit = async (e) => {
 e.preventDefault();

 let response = await fetch('/article/formdata/post/user', {
 method: 'POST',
 body: new FormData(formElem)
 });

 let result = await response.json();

 alert(result.message);
 };
</script>

```

在这个示例中，没有将服务器代码展示出来，因为它超出了我们当前的学习范围。服务器接受 POST 请求并回应“User saved”。

## FormData 方法

我们可以使用以下方法修改 `FormData` 中的字段：

- `formData.append(name, value)` —— 添加具有给定 `name` 和 `value` 的表单字段，
- `formData.append(name, blob, fileName)` —— 添加一个字段，就像它是 `<input type="file">`，第三个参数 `fileName` 设置文件名（而不是表单字段名），因为它是用户文件系统中文件的名称，
- `formData.delete(name)` —— 移除带有给定 `name` 的字段，
- `formData.get(name)` —— 获取带有给定 `name` 的字段值，
- `formData.has(name)` —— 如果存在带有给定 `name` 的字段，则返回 `true`，否则返回 `false`。

从技术上来讲，一个表单可以包含多个具有相同 `name` 的字段，因此，多次调用 `append` 将会添加多个具有相同名称的字段。

还有一个 `set` 方法，语法与 `append` 相同。不同之处在于 `.set` 移除所有具有给定 `name` 的字段，然后附加一个新字段。因此，它确保了只有一个具有这种 `name` 的字段，其他的和 `append` 一样：

- `formData.set(name, value)` ,
- `formData.set(name, blob, fileName)` 。

我们也可以使用 `for..of` 循环迭代 `formData` 字段:

```
let formData = new FormData();
formData.append('key1', 'value1');
formData.append('key2', 'value2');

// 列出 key/value 对
for(let [name, value] of formData) {
 alert(` ${name} = ${value}`); // key1=value1, 然后是 key2=value2
}
```

## 发送带有文件的表单

表单始终以 `Content-Type: multipart/form-data` 来发送数据，这个编码允许发送文件。因此 `<input type="file">` 字段也能被发送，类似于普通的表单提交。

这是具有这种形式的示例:

```
<form id="formElem">
 <input type="text" name="firstName" value="John">
 Picture: <input type="file" name="picture" accept="image/*">
 <input type="submit">
</form>

<script>
 formElem.onsubmit = async (e) => {
 e.preventDefault();

 let response = await fetch('/article/formdata/post/user-avatar', {
 method: 'POST',
 body: new FormData(formElem)
 });

 let result = await response.json();

 alert(result.message);
 };
</script>
```

<input type="text" value="John"/>	Picture: <input type="file"/>	No file chosen	<input type="button" value="Submit"/>
-----------------------------------	-------------------------------	----------------	---------------------------------------

## 发送具有 Blob 数据的表单

正如我们在 [Fetch](#) 一章中所看到的，以 `Blob` 发送一个动态生成的二进制数据，例如图片，是很简单的。我们可以直接将其作为 `fetch` 参数的 `body`。

但在实际中，通常更方便的发送图片的方式不是单独发送，而是将其作为表单的一部分，并带有附加字段（例如“`name`”和其他 `metadata`）一起发送。

并且，服务器通常更适合接收多部分编码的表单（`multipart-encoded form`），而不是原始的二进制数据。

下面这个例子使用 `FormData` 将一个来自 `<canvas>` 的图片和一些其他字段一起作为一个表单提交：

```
<body style="margin:0">
 <canvas id="canvasElem" width="100" height="80" style="border:1px solid"></canvas>

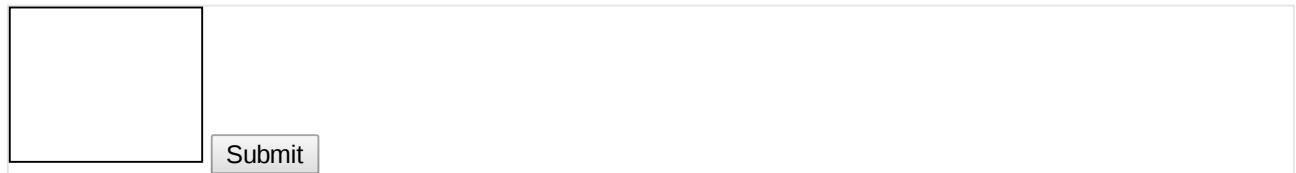
 <input type="button" value="Submit" onclick="submit()">

 <script>
 canvasElem.onmousemove = function(e) {
 let ctx = canvasElem.getContext('2d');
 ctx.lineTo(e.clientX, e.clientY);
 ctx.stroke();
 };

 async function submit() {
 let imageBlob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/png'));

 let formData = new FormData();
 formData.append("firstName", "John");
 formData.append("image", imageBlob, "image.png");

 let response = await fetch('/article/formdata/post/image-form', {
 method: 'POST',
 body: formData
 });
 let result = await response.json();
 alert(result.message);
 }
 </script>
</body>
```



请注意图片 `Blob` 是如何添加的：

```
formData.append("image", imageBlob, "image.png");
```

就像表单中有 `<input type="file" name="image">` 一样，用户从他们的文件系统中使用数据 `imageBlob`（第二个参数）提交了一个名为 `image.png`（第三个参数）的文件。

服务器读取表单数据和文件，就好像它是常规的表单提交一样。

## 总结

`FormData` 对象用于捕获 HTML 表单，并使用 `fetch` 或其他网络方法提交。

我们可以从 HTML 表单创建 `new FormData(form)`，也可以创建一个完全没有表单的对象，然后使用以下方法附加字段：

- `formData.append(name, value)`
- `formData.append(name, blob, fileName)`
- `formData.set(name, value)`
- `formData.set(name, blob, fileName)`

让我们在这里注意两个特点：

1. `set` 方法会移除具有相同名称（`name`）的字段，而 `append` 不会。
2. 要发送文件，需要使用三个参数的语法，最后一个参数是文件名，一般是通过 `<input type="file">` 从用户文件系统中获取的。

其他方法是：

- `formData.delete(name)`
- `formData.get(name)`
- `formData.has(name)`

这就是它的全貌！

## Fetch: 下载进度

`fetch` 方法允许去跟踪 **下载** 进度。

请注意：到目前为止，`fetch` 方法无法跟踪 **上传** 进度。对于这个目的，请使用 [XMLHttpRequest](#)，我们在后面章节会讲到。

要跟踪下载进度，我们可以使用 `response.body` 属性。它是 `ReadableStream` —— 一个特殊的对象，它可以逐块（`chunk`）提供 `body`。在 [Streams API](#) 规范中有对 `ReadableStream` 的详细描述。

与 `response.text()`, `response.json()` 和其他方法不同，`response.body` 给予了对进度读取的完全控制，我们可以随时计算下载了多少。

这是从 `response.body` 读取 `response` 的示例代码:

```
// 代替 response.json() 以及其他方法
const reader = response.body.getReader();

// 在 body 下载时，一直为无限循环
while(true) {
 // 当最后一块下载完成时，done 值为 true
 // value 是块字节的 Uint8Array
 const {done, value} = await reader.read();

 if (done) {
 break;
 }

 console.log(`Received ${value.length} bytes`)
}
```

`await reader.read()` 调用的结果是一个具有两个属性的对象:

- **done** —— 当读取完成时为 `true`，否则为 `false`。
- **value** —— 字节的类型化数组: `Uint8Array`。

**i** **请注意:**

Streams API 还描述了如果使用 `for await...of` 循环异步迭代 `ReadableStream`，但是目前为止，它还未得到很好的支持（参见 [浏览器问题](#)），所以我们使用了 `while` 循环。

我们在循环中接收响应块 (response chunk)，直到加载完成，也就是: 直到 `done` 为 `true`。

要将进度打印出来，我们只需要将每个接收到的片段 `value` 的长度 (`length`) 加到 `counter` 即可。

这是获取响应，并在控制台中记录进度的完整工作示例，下面有更多说明:

```
// Step 1: 启动 fetch，并获得一个 reader
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.ja
const reader = response.body.getReader();

// Step 2: 获得总长度 (length)
const contentLength = +response.headers.get('Content-Length');

// Step 3: 读取数据
let receivedLength = 0; // 当前接收到了这么多字节
```

```

let chunks = [] // 接收到的二进制块的数组（包括 body）
while(true) {
 const {done, value} = await reader.read();

 if (done) {
 break;
 }

 chunks.push(value);
 receivedLength += value.length;

 console.log(`Received ${receivedLength} of ${contentLength}`)
}

// Step 4: 将块连接到单个 Uint8Array
let chunksAll = new Uint8Array(receivedLength); // (4.1)
let position = 0;
for(let chunk of chunks) {
 chunksAll.set(chunk, position); // (4.2)
 position += chunk.length;
}

// Step 5: 解码成字符串
let result = new TextDecoder("utf-8").decode(chunksAll);

// 我们完成啦!
let commits = JSON.parse(result);
alert(commits[0].author.login);

```

让我们一步步解释下这个过程：

1. 我们像往常一样执行 `fetch`，但不是调用 `response.json()`，而是获得了一个流读取器（stream reader） `response.body.getReader()`。

请注意，我们不能同时使用这两种方法来读取相同的响应。要么使用流读取器，要么使用 `reponse` 方法来获取结果。

2. 在读取数据之前，我们可以从 `Content-Length` header 中得到完整的响应长度。

跨源请求中可能不存在这个 `header`（请参见 [Fetch: 跨源请求](#)），并且从技术上讲，服务器可以不设置它。但是通常情况下它都会在那里。

3. 调用 `await reader.read()`，直到它完成。

我们将响应块收集到数组 `chunks` 中。这很重要，因为在使用完（consumed）响应后，我们将无法使用 `response.json()` 或者其他方式（你可以试试，将会有出现 `error`）去“重新读取”它。

4. 最后，我们有了一个 `chunks` —— 一个 `Uint8Array` 字节块数组。我们需要将这些块合并成一个结果。但不幸的是，没有单个方法可以将它们串联起来，所以这里需要一些代码来实现：

1. 我们创建 `chunksAll = new Uint8Array(receivedLength)` —— 一个具有所有数据块合并后的长度的同类型数组。
2. 然后使用 `.set(chunk, position)` 方法，从数组中一个个地复制这些 `chunk`。
5. 我们的结果现在储存在 `chunksAll` 中。但它是一个字节数组，不是字符串。

要创建一个字符串，我们需要解析这些字节。可以使用内建的 `TextDecoder` 对象完成。然后，我们可以 `JSON.parse` 它，如果有必要的话。

如果我们需要的是二进制内容而不是字符串呢？这更简单。用下面这行代码替换掉第 4 和第 5 步，这行代码从所有块创建一个 `Blob`：

```
let blob = new Blob(chunks);
```

最后，我们得到了结果（以字符串或 `blob` 的形式表示，什么方便就用什么），并在过程中对进度进行了跟踪。

再强调一遍，这不能用于 `上传` 过程（现在无法通过 `fetch` 获取），仅用于 `下载` 过程。

## Fetch: 中止 (Abort)

正如我们所知道的，`fetch` 返回一个 `promise`。`JavaScript` 通常并没有“中止”`promise` 的概念。那么我们怎样才能中止 `fetch` 呢？

为此有一个特殊的内建对象：`AbortController`，它不仅可以中止 `fetch`，还可以中止其他异步任务。

用法很简单：

- Step 1: 创建一个控制器 (`controller`)：

```
let controller = new AbortController();
```

控制器是一个极其简单的对象。

- 它具有单个方法 `abort()`，和单个属性 `signal`。
- 当 `abort()` 被调用时：
  - `abort` 事件就会在 `controller.signal` 上触发
  - `controller.signal.aborted` 属性变为 `true`。

任何对 `abort()` 调用感兴趣的人，都可以在 `controller.signal` 上设置监听器来对其进行跟踪。

就像这样（目前还没有 `fetch`）：

```
let controller = new AbortController();
let signal = controller.signal;

// 当 controller.abort() 被调用时触发
signal.addEventListener('abort', () => alert("abort!"));

controller.abort(); // 中止!

alert(signal.aborted); // true
```

- Step 2: 将 `signal` 属性传递给 `fetch` 选项:

```
let controller = new AbortController();
fetch(url, {
 signal: controller.signal
});
```

`fetch` 方法知道如何与 `AbortController` 一起使用，它会监听 `signal` 上的 `abort`。

- Step 3: 调用 `controller.abort()` 来中止:

```
controller.abort();
```

我们完成啦： `fetch` 从 `signal` 获取了事件并中止了请求。

当一个 `fetch` 被中止，它的 `promise` 就会以一个 `error AbortError` `reject`，因此我们应该对其进行处理：

```
// 1 秒后中止
let controller = new AbortController();
setTimeout(() => controller.abort(), 1000);

try {
 let response = await fetch('/article/fetch-abort/demo/hang', {
 signal: controller.signal
 });
} catch(err) {
 if (err.name == 'AbortError') { // handle abort()
 alert("Aborted!");
 } else {
 throw err;
 }
}
```

```
 }
}
```

**AbortController** 是可扩展的，它允许一次取消多个 `fetch`。

例如，这里我们并行 `fetch` 多个 `urls`，然后 `controller` 将它们全部中止：

```
let urls = [...]; // 要并行 fetch 的 url 列表

let controller = new AbortController();

let fetchJobs = urls.map(url => fetch(url, {
 signal: controller.signal
}));

let results = await Promise.all(fetchJobs);

// 如果 controller.abort() 被从其他地方调用,
// 它将中止所有 fetch
```

如果我们有自己的与 `fetch` 不同的异步任务，我们可以使用单个 `AbortController` 中止这些任务以及 `fetch`。

我们只需要监听其 `abort` 事件：

```
let urls = [...];
let controller = new AbortController();

let ourJob = new Promise((resolve, reject) => { // 我们的任务
 ...
 controller.signal.addEventListener('abort', reject);
});

let fetchJobs = urls.map(url => fetch(url, { // fetches
 signal: controller.signal
}));

// 等待完成我们的任务和所有 fetch
let results = await Promise.all([...fetchJobs, ourJob]);

// 如果 controller.abort() 被从其他地方调用,
// 它将中止所有 fetch 和 ourJob
```

所以，`AbortController` 不仅适用于 `fetch`，它还是一个可以中止异步任务的通用对象，`fetch` 具有它的内建集成（译注：即 `fetch` 在内部集成了它）。

## Fetch: 跨源请求

如果我们向另一个网站发送 `fetch` 请求，则该请求可能会失败。

例如，让我们尝试向 `http://example.com` 发送 `fetch` 请求：

```
try {
 await fetch('http://example.com');
} catch(err) {
 alert(err); // fetch 失败
}
```

正如所料，获取失败。

这里的核心概念是 **源（origin）** —— 域（domain）/端口（port）/协议（protocol）的组合。

跨源请求 —— 那些发送到其他域（即使是子域）、协议或端口的请求 —— 需要来自远程端的特殊 `header`。

这个策略被称为“**CORS**”：跨源资源共享（Cross-Origin Resource Sharing）。

## 为什么需要 CORS? 跨源请求简史

CORS 的存在是为了保护互联网免受黑客攻击。

说真的，在这说点儿题外话，讲讲它的历史。

多年来，来自一个网站的脚本无法访问另一个网站的内容。

这个简单有力的规则是互联网安全的基础。例如，来自 `hacker.com` 的脚本无法访问 `gmail.com` 上的用户邮箱。基于这样的规则，人们感到很安全。

在那时候，`JavaScript` 并没有任何特殊的执行网络请求的方法。它只是一种用来装饰网页的玩具语言而已。

但是 `Web` 开发人员需要更多功能。人们发明了各种各样的技巧去突破该限制，并向其他网站发出请求。

## 使用表单

其中一种和其他服务器通信的方法是在那里提交一个 `<form>`。人们将它提交到 `<iframe>`，只是为了停留在当前页面，像这样：

```
<!-- 表单目标 -->
<iframe name="iframe"></iframe>

<!-- 表单可以由 JavaScript 动态生成并提交 -->
<form target="iframe" method="POST" action="http://another.com/...">
```

```
...
</form>
```

因此，即使没有网络方法，也可以向其他网站发出 GET/POST 请求，因为表单可以将数据发送到任何地方。但是由于禁止从其他网站访问 `<iframe>` 中的内容，因此就无法读取响应。

确切地说，实际上有一些技巧能够解决这个问题，这在 `iframe` 和页面中都需要添加特殊脚本。因此，与 `iframe` 的通信在技术上是可能的。现在我们没必要讲其细节内容，我们还是让这些古董代码不要再出现了吧。

## 使用 `script`

另一个技巧是使用 `script` 标签。`script` 可以具有任何域的 `src`，例如 `<script src="http://another.com/...">`。也可以执行来自任何网站的 `script`。

如果一个网站，例如 `another.com` 试图公开这种访问方式的数据，则会使用所谓的“JSONP (JSON with padding)”协议。

这是它的工作方式。

假设在我们的网站，需要以这种方式从 `http://another.com` 网站获取数据，例如天气：

- 首先，我们先声明一个全局函数来接收数据，例如 `gotWeather`。

```
// 1. 声明处理天气数据的函数
function gotWeather({ temperature, humidity }) {
 alert(`temperature: ${temperature}, humidity: ${humidity}`);
}
```

- 然后我们创建一个特性（attribute）为

`src="http://another.com/weather.json?callback=gotWeather"` 的 `<script>` 标签，使用我们的函数名作为它的 `callback` URL-参数。

```
let script = document.createElement('script');
script.src = `http://another.com/weather.json?callback=gotWeather`;
document.body.append(script);
```

- 远程服务器 `another.com` 动态生成一个脚本，该脚本调用 `gotWeather(...)`，发送它想让我们接收的数据。

```
// 我们期望来自服务器的回答看起来像这样:
gotWeather({
 temperature: 25,
```

```
humidity: 78
});
```

4. 当远程脚本加载并执行时，`gotWeather` 函数将运行，并且因为它是我们的函数，我们就有了需要的数据。

这是可行的，并且不违反安全规定，因为双方都同意以这种方式传递数据。而且，既然双方都同意这种行为，那这肯定不是黑客攻击了。现在仍然有提供这种访问的服务，因为即使是非常旧的浏览器它依然适用。

不久之后，网络方法出现在了浏览器 JavaScript 中。

起初，跨源请求是被禁止的。但是，经过长时间的讨论，跨源请求被允许了，但是任何新功能都需要服务器明确允许，以特殊的 `header` 表述。

## 简单的请求

有两种类型的跨源请求：

1. 简单的请求。
2. 所有其他请求。

顾名思义，简单的请求很简单，所以我们先从它开始。

一个 [简单的请求](#) 是指满足以下两个条件的请求：

1. [简单的方法](#)：GET, POST 或 HEAD
2. [简单的 header](#) —— 仅允许自定义下列 header:

- `Accept`,
- `Accept-Language`,
- `Content-Language`,
- `Content-Type` 的值为 `application/x-www-form-urlencoded`, `multipart/form-data` 或 `text/plain`。

任何其他请求都被认为是“非简单请求”。例如，具有 `PUT` 方法或 `API-Key` HTTP-header 的请求就不是简单请求。

本质区别在于，可以使用 `<form>` 或 `<script>` 进行“简单请求”，而无需任何其他特殊方法。

因此，即使是非常旧的服务器也能很好地接收简单请求。

与此相反，带有非标准 `header` 或者例如 `DELETE` 方法的请求，无法通过这种方式创建。在很长一段时间里，JavaScript 都不能进行这样的请求。所以，旧的服务器可能会认为此类请求来自具有特权的来源（privileged source），“因为网页无法发送它们”。

当我们尝试发送一个非简单请求时，浏览器会发送一个特殊的“预检（preflight）”请求到服务器——询问服务器，你接受此类跨源请求吗？

并且，除非服务器明确通过 `header` 进行确认，否则非简单请求不会被发送。

现在，我们来详细介绍它们。

## 用于简单请求的 CORS

如果一个请求是跨源的，浏览器始终会向其添加 `Origin header`。

例如，如果我们从 `https://javascript.info/page` 请求 `https://anywhere.com/request`，请求的 `header` 将会如下：

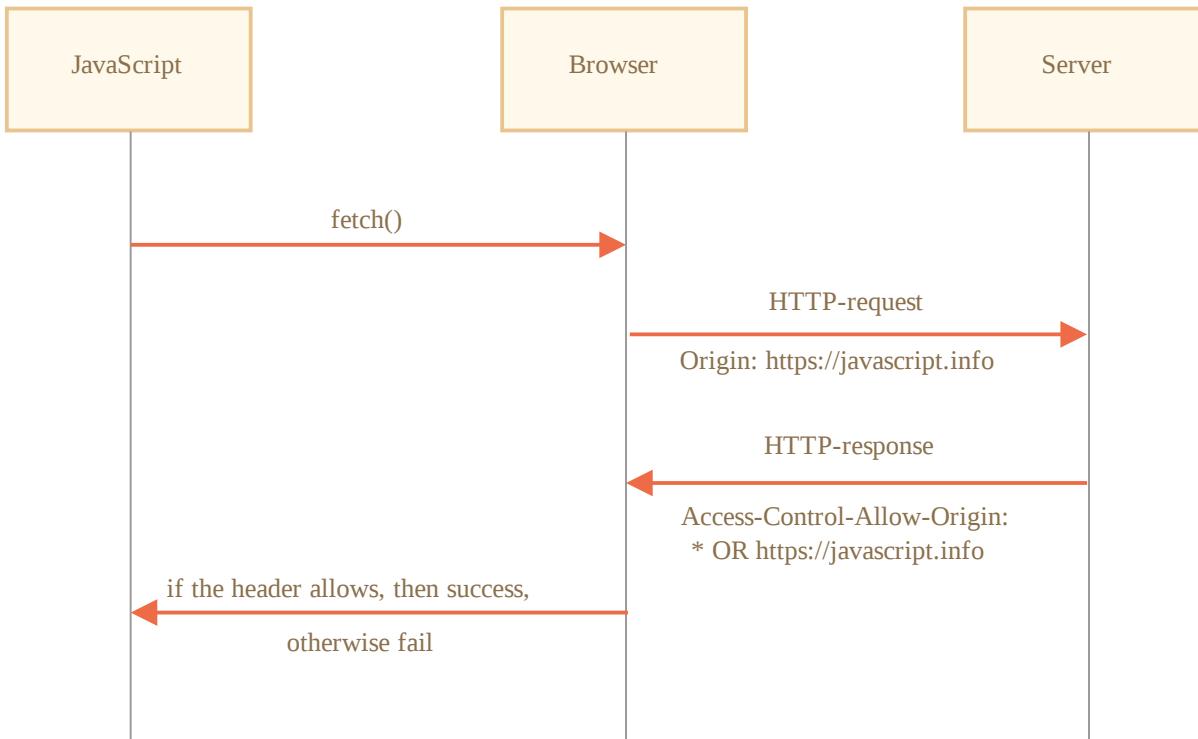
```
GET /request
Host: anywhere.com
Origin: https://javascript.info
...
...
```

正如你所见，`Origin` 包含了确切的源（domain/protocol/port），没有路径。

服务器可以检查 `Origin`，如果同意接受这样的请求，就会在响应中添加一个特殊的 `header Access-Control-Allow-Origin`。该 `header` 包含了允许的源（在我们的示例中是 `https://javascript.info`），或者一个星号 `*`。然后响应成功，否则报错。

浏览器在这里扮演受被信任的中间人的角色：

1. 它确保发送的跨源请求带有正确的 `Origin`。
2. 它检查响应中的许可 `Access-Control-Allow-Origin`，如果存在，则允许 JavaScript 访问响应，否则将失败并报错。



这是一个带有服务器许可的响应示例:

```

200 OK
Content-Type: text/html; charset=UTF-8
Access-Control-Allow-Origin: https://javascript.info

```

## Response header

对于跨源请求，默认情况下，JavaScript 只能访问“简单” response header:

- Cache-Control
- Content-Language
- Content-Type
- Expires
- Last-Modified
- Pragma

访问任何其他 response header 都将导致 error。

**i** 请注意:

请注意: 列表中没有 Content-Length header!

该 header 包含完整的响应长度。因此，如果我们正在下载某些内容，并希望跟踪进度百分比，则需要额外的权限才能访问该 header（请见下文）。

要授予 JavaScript 对任何其他 response header 的访问权限，服务器必须发送 `Access-Control-Expose-Headers` header。它包含一个以逗号分隔的应该被设置为可访问的非简单 header 名称列表。

例如：

```
200 OK
Content-Type:text/html; charset=UTF-8
Content-Length: 12345
API-Key: 2c9de507f2c54aa1
Access-Control-Allow-Origin: https://javascript.info
Access-Control-Expose-Headers: Content-Length, API-Key
```

有了这种 `Access-Control-Expose-Headers` header，此脚本就被允许读取响应的 `Content-Length` 和 `API-Key` header。

## “非简单”请求

我们可以使用任何 HTTP 方法：不仅仅是 `GET/POST`，也可以是 `PATCH`，`DELETE` 及其他。

之前，没有人能够设想网页能发出这样的请求。因此，可能仍然存在有些 Web 服务将非标准方法视为一个信号：“这不是浏览器”。它们可以在检查访问权限时将其考虑在内。

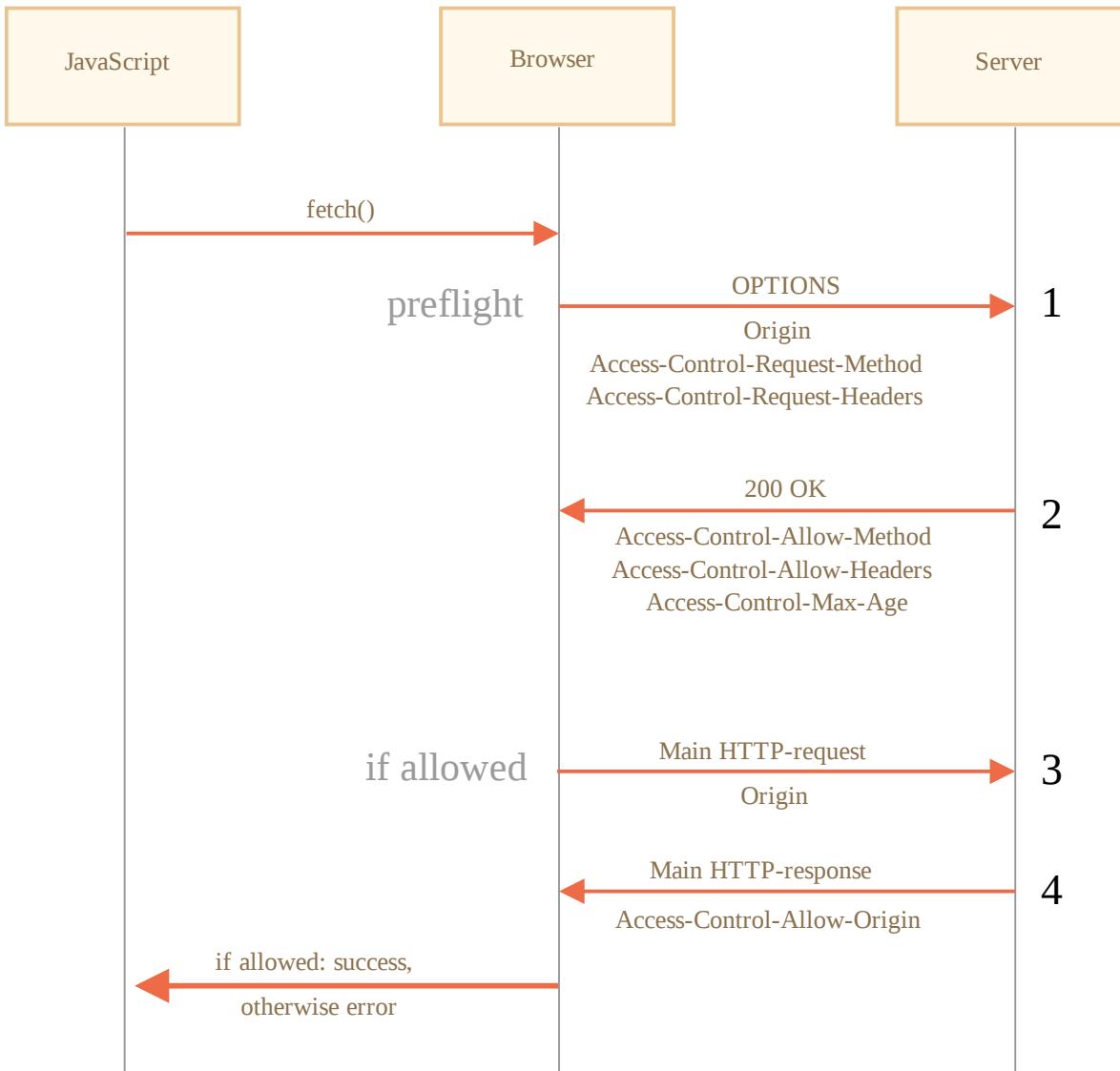
因此，为了避免误解，任何“非标准”请求——浏览器不会立即发出在过去无法完成的这类请求。即在它发送这类请求前，会先发送“预检（preflight）”请求来请求许可。

预检请求使用 `OPTIONS` 方法，它没有 `body`，但是有两个 `header`：

- `Access-Control-Request-Method` header 带有非简单请求的方法。
- `Access-Control-Request-Headers` header 提供一个以逗号分隔的非简单 HTTP-header 列表。

如果服务器同意处理请求，那么它会进行响应，此响应的状态码应该为 200，没有 `body`，具有 `header`：

- `Access-Control-Allow-Methods` 必须具有允许的方法。
- `Access-Control-Allow-Headers` 必须具有一个允许的 `header` 列表。
- 另外，`header Access-Control-Max-Age` 可以指定缓存此权限的秒数。因此，浏览器不是必须为满足给定权限的后续请求发送预检。



让我们用一个例子来一步步看一下它是怎么工作的，对于一个跨源的 **PATCH** 请求（此方法经常被用于更新数据）：

```
let response = await fetch('https://site.com/service.json', {
 method: 'PATCH',
 headers: {
 'Content-Type': 'application/json',
 'API-Key': 'secret'
 }
});
```

这里有三个理由解释为什么它不是一个简单请求（其实一个就够了）：

- 方法 **PATCH**
- **Content-Type** 不是这三个中之一：**application/x-www-form-urlencoded**, **multipart/form-data**, **text/plain**。
- “非简单” **API-Key header**。

## Step 1 预检请求 (preflight request)

在发送我们的请求前，浏览器会自己发送如下所示的预检请求：

```
OPTIONS /service.json
Host: site.com
Origin: https://javascript.info
Access-Control-Request-Method: PATCH
Access-Control-Request-Headers: Content-Type, API-Key
```

- 方法：`OPTIONS`。
- 路径 —— 与主请求完全相同：`/service.json`。
- 特殊跨源头：
  - `Origin` —— 来源。
  - `Access-Control-Request-Method` —— 请求方法。
  - `Access-Control-Request-Headers` —— 以逗号分隔的“非简单” header 列表。

## Step 2 预检响应 (preflight response)

服务应响应状态 200 和 header：

- `Access-Control-Allow-Methods: PATCH`
- `Access-Control-Allow-Headers: Content-Type, API-Key`。

这将允许后续通信，否则会触发错误。

如果服务器将来期望其他方法和 header，则可以通过添加到列表中来预先允许它们：

```
200 OK
Access-Control-Allow-Methods: PUT, PATCH, DELETE
Access-Control-Allow-Headers: API-Key, Content-Type, If-Modified-Since, Cache-Control
Access-Control-Max-Age: 86400
```

现在，浏览器可以看到 `PATCH` 在 `Access-Control-Allow-Methods` 中，`Content-Type, API-Key` 在列表 `Access-Control-Allow-Headers` 中，因此它将发送主请求。

此外，预检响应会缓存一段时间，该时间由 `Access-Control-Max-Age` header 指定（86400 秒，一天），因此，后续请求将不会导致预检。假设它们符合缓存的配额，则将直接发送它们。

## Step 3 实际请求 (actual request)

预检成功后，浏览器现在发出主请求。这里的算法与简单请求的算法相同。

主请求具有 `Origin` header（因为它是跨源的）：

```
PATCH /service.json
Host: site.com
Content-Type: application/json
API-Key: secret
Origin: https://javascript.info
```

## Step 4 实际响应 (actual response)

服务器不应该忘记在主响应中添加 `Access-Control-Allow-Origin`。成功的预检并不能免除此要求：

```
Access-Control-Allow-Origin: https://javascript.info
```

然后，JavaScript 可以读取主服务器响应了。

**i** **请注意:**

预检请求发生在“幕后”，它对 JavaScript 不可见。

JavaScript 仅获取对主请求的响应，如果没有服务器许可，则获得一个 `error`。

## 凭据 (Credentials)

默认情况下，跨源请求不会带来任何凭据（`cookies` 或者 HTTP 认证（HTTP authentication））。

这对于 HTTP 请求来说并不常见。通常，对 `http://site.com` 的请求附带有该域的所有 `cookie`。但是由 JavaScript 方法发出的跨源请求是个例外。

例如，`fetch('http://another.com')` 不会发送任何 `cookie`，即使那些 (!) 属于 `another.com` 域的 `cookie`。

为什么？

这是因为具有凭据的请求比没有凭据的请求要强大得多。如果被允许，它会使用它们的凭据授予 JavaScript 代表用户行为和访问敏感信息的全部权力。

服务器真的这么信任这种脚本吗？是的，它必须显式地带有允许请求的凭据和附加 `header`。

要在 `fetch` 中发送凭据，我们需要添加 `credentials: "include"` 选项，像这样：

```
fetch('http://another.com', {
 credentials: "include"
});
```

现在，`fetch` 将把源自 `another.com` 的 `cookie` 和我们的请求发送到该网站。

如果服务器同意接受 **带有凭据** 的请求，则除了 `Access-Control-Allow-Origin` 外，服务器还应该在响应中添加 header `Access-Control-Allow-Credentials: true`。

例如：

```
200 OK
Access-Control-Allow-Origin: https://javascript.info
Access-Control-Allow-Credentials: true
```

请注意：对于具有凭据的请求，禁止 `Access-Control-Allow-Origin` 使用星号`*`。如上所示，它必须有一个确切的源。这是另一项安全措施，以确保服务器真的知道它信任的发出此请求的是谁。

## 总结

从浏览器角度来看，有两种跨源请求：“简单”请求和其他请求。

**简单请求 ↗** 必须满足下列条件：

- 方法：`GET`, `POST` 或 `HEAD`。
- header —— 我们仅能设置：
  - `Accept`
  - `Accept-Language`
  - `Content-Language`
  - `Content-Type` 的值为 `application/x-www-form-urlencoded`, `multipart/form-data` 或 `text/plain`。

简单请求和其他请求的本质区别在于，自古以来使用 `<form>` 或 `<script>` 标签进行简单请求就是可行的，而长期以来浏览器都不能进行非简单请求。

所以，实际区别在于，简单请求会使用 `origin` header 并立即发送，而对于其他请求，浏览器会发出初步的“预检”请求，以请求许可。

**对于简单请求：**

- → 浏览器发送带有源的 `Origin` header。
- ← 对于没有凭据的请求（默认不发送），服务器应该设置：
  - `Access-Control-Allow-Origin` 为 `*` 或与 `Origin` 的值相同
- ← 对于具有凭据的请求，服务器应该设置：
  - `Access-Control-Allow-Origin` 值与 `Origin` 的相同
  - `Access-Control-Allow-Credentials` 为 `true`

此外，要授予 JavaScript 访问除 Cache-Control, Content-Language, Content-Type, Expires, Last-Modified 或 Pragma 外的任何 response header 的权限，服务器应该在 header Access-Control-Expose-Headers 中列出允许的那些 header。

\*\*对于非简单请求，会在请求之前发出初步“预检”请求：

- → 浏览器将具有以下 header 的 OPTIONS 请求发送到相同的 URL：
  - Access-Control-Request-Method 有请求方法。
  - Access-Control-Request-Headers 以逗号分隔的“非简单” header 列表。
- ← 服务器应该响应状态码为 200 和 header：
  - Access-Control-Allow-Methods 带有允许的方法的列表，
  - Access-Control-Allow-Headers 带有允许的 header 的列表，
  - Access-Control-Max-Age 带有指定缓存权限的秒数。
- 然后，发出实际请求，应用先前的“简单”方案。

## Fetch API

到目前为止，我们已经对 fetch 相当了解了。

现在让我们来看看 fetch 的剩余 API，来了解它的全部本领吧。

### ❶ 请注意：

请注意：这些选项 (option) 大多都很少使用。即使跳过本章，你也可以很好地使用 fetch。

但是，知道 fetch 可以做什么还是很好的，所以如果需要，你可以来看看这些细节内容。

这是所有可能的 fetch 选项及其默认值（注释中标注了可选值）的完整列表：

```
let promise = fetch(url, {
 method: "GET", // POST, PUT, DELETE, 等。
 headers: {
 // 内容类型 header 值通常是自动设置的
 // 取决于 request body
 "Content-Type": "text/plain;charset=UTF-8"
 },
 body: undefined // string, FormData, Blob, BufferSource, 或 URLSearchParams
 referrer: "about:client", // 或 "" 以不发送 Referer header,
 // 或者是当前源的 url
 referrerPolicy: "no-referrer-when-downgrade", // no-referrer, origin, same-origin.
 mode: "cors", // same-origin, no-cors
 credentials: "same-origin", // omit, include
```

```
cache: "default", // no-store, reload, no-cache, force-cache, 或 only-if-cached
redirect: "follow", // manual, error
integrity: "", // 一个 hash, 像 "sha256-abcdef1234567890"
keepalive: false, // true
signal: undefined, // AbortController 来中止请求
window: window // null
});
```

一个令人印象深刻的列表，对吧？

我们已经在 [Fetch](#) 一章中详细介绍过了 `method`, `headers` 和 `body`。

在 [Fetch: 中止 \(Abort\)](#) 一章中介绍了 `signal` 选项。

现在让我们学习其余的本领。

## referrer, referrerPolicy

这些选项决定了 `fetch` 如何设置 HTTP 的 `Referer` header。

通常来说，这个 `header` 是被自动设置的，并包含了发出请求的页面的 `url`。在大多数情况下，它一点也不重要，但有时出于安全考虑，删除或缩短它是有意义的。

`referrer` 选项允许设置在当前域的任何 `Referer`，或者移除它。

要不发送 `referrer`，可以将 `referrer` 设置为空字符串：

```
fetch('/page', {
 referrer: "" // 没有 Referer header
});
```

设置在当前域内的另一个 `url`：

```
fetch('/page', {
 // 假设我们在 https://javascript.info
 // 我们可以设置任何 Referer header，但必须是在当前域内的
 referrer: "https://javascript.info/anotherpage"
});
```

`referrerPolicy` 选项为 `Referer` 设置一般的规则。

请求分为 3 种类型：

1. 同源请求。
2. 跨域请求。
3. 从 HTTPS 到 HTTP 的请求 (从安全协议到不安全协议)。

与 `referrer` 选项允许设置确切的 `Referer` 值不同，`referrerPolicy` 告诉浏览器针对各个请求类型的一般的规则。

可能的值在 [Referrer Policy 规范](#) 中有详细描述：

- `"no-referrer-when-downgrade"` —— 默认值：除非我们从 HTTPS 发送请求到 HTTP（到安全性较低的协议），否则始终会发送完整的 `Referer`。
- `"no-referrer"` —— 从不发送 `Referer`。
- `"origin"` —— 只发送在 `Referer` 中的域，而不是完整的页面 URL，例如，只发送 `http://site.com` 而不是 `http://site.com/path`。
- `"origin-when-cross-origin"` —— 发送完整的 `Referer` 到相同的源，但对于跨源请求，只发送域部分（同上）。
- `"same-origin"` —— 发送完整的 `Referer` 到相同的源，但对于跨源请求，不发送 `Referer`。
- `"strict-origin"` —— 只发送域，对于  $\text{HTTPS} \rightarrow \text{HTTP}$  请求，则不发送中则不发送 `Referer`。
- `"strict-origin-when-cross-origin"` —— 对于同源情况下则发送完整的 `Referer`，对于跨源情况下，则只发送域，如果是  $\text{HTTPS} \rightarrow \text{HTTP}$  请求，则什么都不发送。
- `"unsafe-url"` —— 在 `Referer` 中始终发送完整的 url，即使是  $\text{HTTPS} \rightarrow \text{HTTP}$  请求。

这是一个包含所有组合的表格：

值	同源	跨源	$\text{HTTPS} \rightarrow \text{HTTP}$
<code>"no-referrer"</code>	-	-	-
<code>"no-referrer-when-downgrade"</code> 或 <code>""</code> (默认)	完整的 url	完整的 url	-
<code>"origin"</code>	仅域	仅域	仅域
<code>"origin-when-cross-origin"</code>	完整的 url	仅域	仅域
<code>"same-origin"</code>	完整的 url	-	-
<code>"strict-origin"</code>	仅域	仅域	-
<code>"strict-origin-when-cross-origin"</code>	完整的 url	仅域	-
<code>"unsafe-url"</code>	完整的 url	完整的 url	完整的 url

假如我们有一个带有 URL 结构的管理区域（admin zone），它不应该被从网站外看到。

如果我们发送了一个 `fetch`，则默认情况下，它总是发送带有页面完整 url 的 `Referer` header（我们从 HTTPS 向 HTTP 发送请求的情况下除外，这种情况下没有 `Referer`）。

例如 `Referer: https://javascript.info/admin/secret/paths`。

如果我们想让其他网站只知道域的部分，而不是 URL 路径，我们可以这样设置选项：

```
fetch('https://another.com/page', {
 // ...
 referrerPolicy: "origin-when-cross-origin" // Referer: https://javascript.info
});
```

我们可以将其置于所有 `fetch` 调用中，也可以将其集成到我们项目的执行所有请求并在内部使用 `fetch` 的 JavaScript 库中。

与默认行为相比，它的唯一区别在于，对于跨源请求，`fetch` 只发送 URL 域的部分（例如 `https://javascript.info`，没有路径）。对于同源请求，我们仍然可以获得完整的 `Referer`（可能对于调试目的是有用的）。

### Referrer policy 不仅适用于 `fetch`

在 规范 ↗ 中描述的 `referrer policy`，不仅适用于 `fetch`，它还具有全局性。

特别是，可以使用 `Referrer-Policy` HTTP header，或者为每个链接设置 `<a rel="noreferrer">`，来为整个页面设置默认策略（policy）。

## mode

`mode` 选项是一种安全措施，可以防止偶发的跨源请求：

- `"cors"` —— 默认值，允许跨源请求，如 [Fetch: 跨源请求](#) 一章所述，
- `"same-origin"` —— 禁止跨源请求，
- `"no-cors"` —— 只允许简单的跨源请求。

当 `fetch` 的 URL 来自于第三方，并且我们想要一个“断电开关”来限制跨源能力时，此选项可能很有用。

## credentials

`credentials` 选项指定 `fetch` 是否应该随请求发送 cookie 和 HTTP-Authorization header。

- `"same-origin"` —— 默认值，对于跨源请求不发送，
- `"include"` —— 总是发送，需要来自跨源服务器的 `Accept-Control-Allow-Credentials`，才能使 JavaScript 能够访问响应，详细内容在 [Fetch: 跨源请求](#) 一章有详细介绍，
- `"omit"` —— 不发送，即使对于同源请求。

## cache

默认情况下，`fetch` 请求使用标准的 HTTP 缓存。就是说，它遵从 `Expires`，`Cache-Control` header，发送 `If-Modified-Since`，等。就像常规的 HTTP 请求那样。

使用 `cache` 选项可以忽略 HTTP 缓存或者对其进行微调：

- `"default"` —— `fetch` 使用标准的 HTTP 缓存规则和 header，
- `"no-store"` —— 完全忽略 HTTP 缓存，如果我们设置 header `If-Modified-Since`，`If-None-Match`，`If-Unmodified-Since`，`If-Match`，或 `If-Range`，则此模式会成为默认模式，
- `"reload"` —— 不从 HTTP 缓存中获取结果（如果有），而是使用响应填充缓存（如果 response header 允许），
- `"no-cache"` —— 如果有一个已缓存的响应，则创建一个有条件的请求，否则创建一个普通的请求。使用响应填充 HTTP 缓存，
- `"force-cache"` —— 使用来自 HTTP 缓存的响应，即使该响应已过时（stale）。如果 HTTP 缓存中没有响应，则创建一个常规的 HTTP 请求，行为像正常那样，
- `"only-if-cached"` —— 使用来自 HTTP 缓存的响应，即使该响应已过时（stale）。如果 HTTP 缓存中没有响应，则报错。只有当 `mode` 为 `same-origin` 时生效。

## redirect

通常来说，`fetch` 透明地遵循 HTTP 重定向，例如 301，302 等。

`redirect` 选项允许对此进行更改：

- `"follow"` —— 默认值，遵循 HTTP 重定向，
- `"error"` —— HTTP 重定向时报错，
- `"manual"` —— 不遵循 HTTP 重定向，但 `response.url` 将是一个新的 URL，并且 `response.redirected` 将为 `true`，以便我们能够手动执行重定向到新的 URL（如果需要的话）。

## integrity

`integrity` 选项允许检查响应是否与已知的预先校验和相匹配。

正如 规范 ↗ 所描述的，支持的哈希函数有 SHA-256，SHA-384，和 SHA-512，可能还有其他的，这取决于浏览器。

例如，我们下载一个文件，并且我们知道它的 SHA-256 校验和为 “abcdef”（当然，实际校验和会更长）。

我们可以将其放在 `integrity` 选项中，就像这样：

```
fetch('http://site.com/file', {
 integrity: 'sha256-abcdef'
});
```

然后 `fetch` 将自行计算 SHA-256 并将其与我们的字符串进行比较。如果不匹配，则会触发错误。

## keepalive

`keepalive` 选项表示该请求可能会使发起它的网页“失活（`outlive`）”。

例如，我们收集有关当前访问者是如何使用我们的页面（鼠标点击，他查看的页面片段）的统计信息，以分析和改善用户体验。

当访问者离开我们的网页时——我们希望能够将数据保存到我们的服务器上。

我们可以使用 `window.onunload` 事件来实现：

```
window.onunload = function() {
 fetch('/analytics', {
 method: 'POST',
 body: "statistics",
 keepalive: true
 });
};
```

通常，当一个文档被卸载时（`unloaded`），所有相关的网络请求都会被中止。但是，`keepalive` 选项告诉浏览器，即使在离开页面后，也要在后台执行请求。所以，此选项对于我们的请求成功至关重要。

它有一些限制：

- 我们无法发送兆字节的数据：`keepalive` 请求的 `body` 限制为 **64kb**。
  - 如果我们收集了更多数据，我们可以定期将其以数据包的形式发送出去，这样就不会留下太多数据给最后的 `onunload` 请求了。
  - 该限制是对当前正在进行的所有请求的。因此，我们无法通过创建 **100** 个请求，每个 **64kb** 这样来作弊。
- 如果请求是在 `onunload` 中发起的，我们将无法处理服务器响应，因为文档在那个时候已经卸载了（`unloaded`），函数将无法工作。
  - 通常来说，服务器会向此类请求发送空响应，所以这不是问题。

## URL 对象

内建的 `URL` 类提供了用于创建和解析 URL 的便捷接口。

没有任何一个网络方法一定需要使用 `URL` 对象，字符串就足够了。所以从技术上讲，我们并不是必须使用 `URL`。但是有些时候 `URL` 对象真的很有用。

## 创建 URL 对象

创建一个新 `URL` 对象的语法：

```
new URL(url, [base])
```

- `url` —— 完整的 URL，或者仅路径（如果设置了 `base`），
- `base` —— 可选的 `base` URL：如果设置了此参数，且参数 `url` 只有路径，则会根据这个 `base` 生成 URL。

例如：

```
let url = new URL('https://javascript.info/profile/admin');
```

下面这两个 URL 是一样的：

```
let url1 = new URL('https://javascript.info/profile/admin');
let url2 = new URL('/profile/admin', 'https://javascript.info');

alert(url1); // https://javascript.info/profile/admin
alert(url2); // https://javascript.info/profile/admin
```

我们可以根据相对于现有 URL 的路径轻松创建一个新的 URL：

```
let url = new URL('https://javascript.info/profile/admin');
let newUrl = new URL('tester', url);

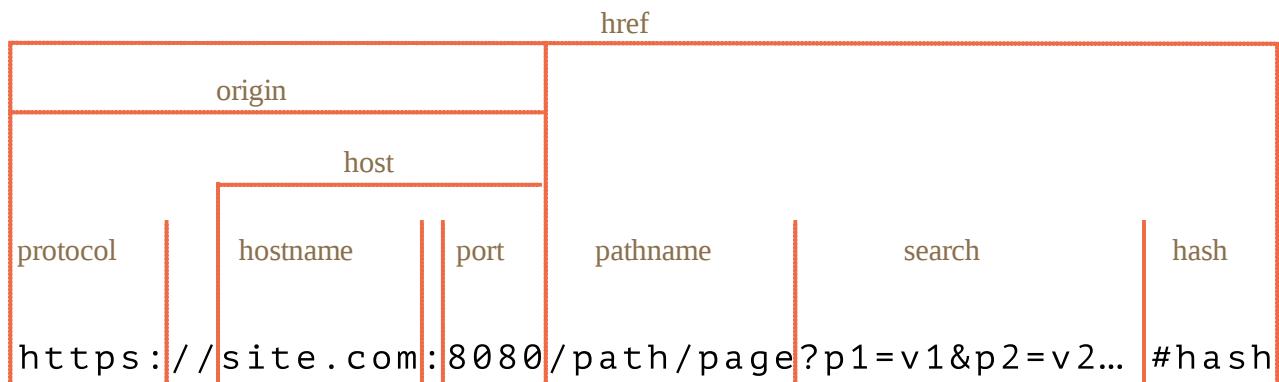
alert(newUrl); // https://javascript.info/profile/tester
```

`URL` 对象立即允许我们访问其组件，因此这是一个解析 `url` 的好方法，例如：

```
let url = new URL('https://javascript.info/url');

alert(url.protocol); // https:
alert(url.host); // javascript.info
alert(url.pathname); // /url
```

这是 URL 组件的备忘单:



- `href` 是完整的 URL，与 `url.toString()` 相同
- `protocol` 以冒号字符 `:` 结尾
- `search` —— 以问号 `?` 开头的一串参数
- `hash` 以哈希字符 `#` 开头
- 如果存在 HTTP 身份验证，则这里可能还会有 `user` 和 `password` 属性：  
`http://login:password@site.com`（图片上没有，很少被用到）。

❶ 我们可以将 `URL` 对象传递给网络（和大多数其他）方法，而不是字符串

我们可以在 `fetch` 或 `XMLHttpRequest` 中使用 `URL` 对象，几乎可以在任何需要 URL 字符串的地方都能使用 `URL` 对象。

通常，`URL` 对象可以替代字符串传递给任何方法，因为大多数方法都会执行字符串转换，这会将 `URL` 对象转换为具有完整 URL 的字符串。

## SearchParams “?...”

假设，我们想要创建一个具有给定搜索参数的 `url`，例如：

`https://google.com/search?query=JavaScript`。

我们可以在 `URL` 字符串中提供它们：

```
new URL('https://google.com/search?query=JavaScript')
```

.....但是，如果参数中包含空格，非拉丁字母等（具体参见下文），参数就需要被编码。

因此，有一个 `URL` 属性用于解决这个问题：`urlSearchParams`，[URLSearchParams](#) 类型的对象。

它为搜索参数提供了简便的方法：

- `append(name, value)` —— 按照 `name` 添加参数，
- `delete(name)` —— 按照 `name` 移除参数，
- `get(name)` —— 按照 `name` 获取参数，
- `getAll(name)` —— 获得相同 `name` 的所有参数（这是可行的，例如 `?user=John&user=Pete`），
- `has(name)` —— 按照 `name` 检查参数是否存在，
- `set(name, value)` —— set/replace 参数，
- `sort()` —— 按 `name` 对参数进行排序，很少使用，
- .....并且它是可迭代的，类似于 `Map`。

包含空格和标点符号的参数的示例：

```
let url = new URL('https://google.com/search');

url.searchParams.set('q', 'test me!'); // 添加带有一个空格和一个 ! 的参数

alert(url); // https://google.com/search?q=test+me%21

url.searchParams.set('tbs', 'qdr:y'); // 添加带有一个冒号 : 的参数

// 参数会被自动编码
alert(url); // https://google.com/search?q=test+me%21&tbs=qdr%3Ay

// 遍历搜索参数 (被解码)
for(let [name, value] of url.searchParams) {
 alert(` ${name}=${value}`); // q=test me!, 然后是 tbs=qdr:y
}
```

## 编码 (encoding)

[RFC3986](#) 标准定义了 `URL` 中允许哪些字符，不允许哪些字符。

那些不被允许的字符必须被编码，例如非拉丁字母和空格——用其 `UTF-8` 代码代替，前缀为 `%`，例如 `%20`（由于历史原因，空格可以用 `+` 编码，但这是一个例外）。

好消息是 `URL` 对象会自动处理这些。我们仅需提供未编码的参数，然后将 `URL` 转换为字符串：

```
// 在此示例中使用一些西里尔字符

let url = new URL('https://ru.wikipedia.org/wiki/Тест');
url.searchParams.set('key', 'ъ');
alert(url); //https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82?key=%D1%8A
```

正如你所看到的，`url` 路径中的 `Тест` 和 `ъ` 参数都被编码了。

`URL` 变长了，因为每个西里尔字母用 `UTF-8` 编码的两个字节表示，因此这里有两个 `%..` 实体（entities）。

## 编码字符串

在过去，在出现 `URL` 对象之前，人们使用字符串作为 `URL`。

而现在，`URL` 对象通常更方便，但是仍然可以使用字符串。在很多情况下，使用字符串可以使代码更短。

如果使用字符串，则需要手动编码/解码特殊字符。

下面是用于编码/解码 `URL` 的内建函数：

- `encodeURI` —— 编码整个 `URL`。
- `decodeURI` —— 解码为编码前的状态。
- `encodeURIComponent` —— 编码 `URL` 组件，例如搜索参数，或者 `hash`，或者 `pathname`。
- `decodeURIComponent` —— 解码为编码前的状态。

一个自然的问题：“`encodeURIComponent` 和 `encodeURI` 之间有什么区别？我们什么时候应该使用哪个？”

如果我们看一个 `URL`，就容易理解了，它被分解为本文上面图中所示的组件形式：

```
https://site.com:8080/path/page?p1=v1&p2=v2#hash
```

正如我们所看到的，在 `URL` 中 `:`，`?`，`=`，`&`，`#` 这类字符是被允许的。

.....另一方面，对于 `URL` 的单个组件，例如一个搜索参数，则必须对这些字符进行编码，以免破坏 `URL` 的格式。

- `encodeURI` 仅编码 `URL` 中完全禁止的字符。
- `encodeURIComponent` 也编码这类字符，此外，还编码 `#`，`$`，`&`，`+`，`,`，`/`，`:`，`;`，`=`，`?` 和 `@` 字符。

所以，对于一个 `URL` 整体，我们可以使用 `encodeURI`：

```
// 在 url 路径中使用西里尔字符
let url = encodeURIComponent('http://site.com/привет');

alert(url); // http://site.com/%D0%BF%D1%80%D0%B8%D0%B2%D0%B5%D1%82
```

.....而对于 URL 参数，我们应该改用 `encodeURIComponent`：

```
let music = encodeURIComponent('Rock&Roll');

let url = `https://google.com/search?q=${music}`;
alert(url); // https://google.com/search?q=Rock%26Roll
```

将其与 `encodeURI` 进行比较：

```
let music = encodeURI('Rock&Roll');

let url = `https://google.com/search?q=${music}`;
alert(url); // https://google.com/search?q=Rock&Roll
```

我们可以看到，`encodeURI` 没有对 `&` 进行编码，因为它对于整个 URL 来说是合法的字符。

但是，我们应该编码在搜索参数中的 `&` 字符，否则，我们将得到 `q=Rock&Roll` —— 实际上是 `q=Rock` 加上一些晦涩的参数 `Roll`。不符合预期。

因此，对于每个搜索参数，我们应该使用 `encodeURIComponent`，以将其正确地插入到 URL 字符串中。最安全的方式是对 `name` 和 `value` 都进行编码，除非我们能够绝对确保它只包含允许的字符。

### i encode\* 与 URL 之间的编码差异

类 `URL` 和 `URLSearchParams` 基于最新的 URL 规范: [RFC3986](#), 而 `encode*` 函数是基于过时的 [RFC2396](#)。

它们之间有一些区别, 例如对 IPv6 地址的编码方式不同:

```
// IPv6 地址的合法 url
let url = 'http://[2607:f8b0:4005:802::1007]/';
alert(encodeURI(url)); // http://%5B2607:f8b0:4005:802::1007%5D/
alert(new URL(url)); // http://[2607:f8b0:4005:802::1007]/
```

正如我们所看到的, `encodeURI` 替换了方括号 [...] , 这是不正确的, 原因是: 在 [RFC2396 \(August 1998\)](#) 时代, 还不存在 IPv6 url。

这种情况很少见, `encode*` 函数在大多数情况下都能正常工作。

## XMLHttpRequest

`XMLHttpRequest` 是一个内建的浏览器对象, 它允许使用 JavaScript 发送 HTTP 请求。

虽然它的名字里面有“XML”一词, 但它可以操作任何数据, 而不仅仅是 XML 格式。我们可以用它来上传/下载文件, 跟踪进度等。

现如今, 我们有一个更为现代的方法叫做 `fetch`, 它的出现使得 `XMLHttpRequest` 在某种程度上被弃用。

在现代 Web 开发中, 出于以下三种原因, 我们还在使用 `XMLHttpRequest`:

1. 历史原因: 我们需要支持现有的使用了 `XMLHttpRequest` 的脚本。
2. 我们需要兼容旧浏览器, 并且不想用 `polyfill` (例如为了使脚本更小)。
3. 我们需要做一些 `fetch` 目前无法做到的事情, 例如跟踪上传进度。

这些话听起来熟悉吗? 如果是, 那么请继续阅读下面的 `XMLHttpRequest` 相关内容吧。如果还不是很熟悉的话, 那么请先阅读 [Fetch](#) 一章的内容。

## XMLHttpRequest 基础

`XMLHttpRequest` 有两种执行模式: 同步 (`synchronous`) 和异步 (`asynchronous`)。

我们首先来看看最常用的异步模式:

要发送请求, 需要 3 个步骤:

## 1. 创建 XMLHttpRequest :

```
let xhr = new XMLHttpRequest();
```

此构造器没有参数。

## 2. 初始化它，通常就在 new XMLHttpRequest 之后:

```
xhr.open(method, URL, [async, user, password])
```

此方法指定请求的主要参数:

- `method` —— HTTP 方法。通常是 `"GET"` 或 `"POST"`。
- `URL` —— 要请求的 URL，通常是一个字符串，也可以是 `URL` 对象。
- `async` —— 如果显式地设置为 `false`，那么请求将会以同步的方式处理，我们稍后会讲到它。
- `user`, `password` —— HTTP 基本身份验证（如果需要的话）的登录名和密码。

请注意，`open` 调用与其名称相反，不会建立连接。它仅配置请求，而网络活动仅以 `send` 调用开启。

## 3. 发送请求。

```
xhr.send([body])
```

这个方法会建立连接，并将请求发送到服务器。可选参数 `body` 包含了 `request body`。

一些请求方法，像 `GET` 没有 `request body`。还有一些请求方法，像 `POST` 使用 `body` 将数据发送到服务器。我们稍后会看到相应示例。

## 4. 监听 xhr 事件以获取响应。

这三个事件是最常用的:

- `load` —— 当请求完成（即使 HTTP 状态为 400 或 500 等），并且响应已完全下载。
- `error` —— 当无法发出请求，例如网络中断或者无效的 URL。
- `progress` —— 在下载响应期间定期触发，报告已经下载了多少。

```
xhr.onload = function() {
 alert(`Loaded: ${xhr.status} ${xhr.response}`);
}
```

```

};

xhr.onerror = function() { // 仅在根本无法发出请求时触发
 alert(`Network Error`);
};

xhr.onprogress = function(event) { // 定期触发
 // event.loaded — 已经下载了多少字节
 // event.lengthComputable = true, 当服务器发送了 Content-Length header 时
 // event.total — 总字节数 (如果 lengthComputable 为 true)
 alert(`Received ${event.loaded} of ${event.total}`);
};

```

下面是一个完整的示例。它从服务器加载

`/article/xmlhttprequest/example/load`，并打印加载进度：

```

// 1. 创建一个 new XMLHttpRequest 对象
let xhr = new XMLHttpRequest();

// 2. 配置它：从 URL /article/.../load GET-request
xhr.open('GET', '/article/xmlhttprequest/example/load');

// 3. 通过网络发送请求
xhr.send();

// 4. 当接收到响应后，将调用此函数
xhr.onload = function() {
 if (xhr.status != 200) { // 分析响应的 HTTP 状态
 alert(`Error ${xhr.status}: ${xhr.statusText}`); // 例如 404: Not Found
 } else { // 显示结果
 alert(`Done, got ${xhr.responseText}`); // response 是服务器响应
 }
};

xhr.onprogress = function(event) {
 if (event.lengthComputable) {
 alert(`Received ${event.loaded} of ${event.total} bytes`);
 } else {
 alert(`Received ${event.loaded} bytes`); // 没有 Content-Length
 }
};

xhr.onerror = function() {
 alert("Request failed");
};

```

一旦服务器有了响应，我们可以在以下 `xhr` 属性中接收结果：

**status**

HTTP 状态码（一个数字）：`200`，`404`，`403` 等，如果出现非 HTTP 错误，则为`0`。

### statusText

HTTP 状态消息（一个字符串）：状态码为`200` 对应于`OK`，`404` 对应于`NotFound`，`403` 对应于`Forbidden`。

### response（旧脚本可能用的是`responseText`）

服务器 response body。

我们还可以使用相应的属性指定超时（`timeout`）：

```
xhr.timeout = 10000; // timeout 单位是 ms, 此处即 10 秒
```

如果在给定时间内请求没有成功执行，请求就会被取消，并且触发`timeout`事件。

## i URL 搜索参数（URL search parameters）

为了向 URL 添加像`?name=value`这样的参数，并确保正确的编码，我们可以使用`URL`对象：

```
let url = new URL('https://google.com/search');
url.searchParams.set('q', 'test me!');

// 参数 'q' 被编码
xhr.open('GET', url); // https://google.com/search?q=test+me%21
```

## 响应类型

我们可以使用`xhr.responseType`属性来设置响应格式：

- ""（默认）——响应格式为字符串，
- "text"——响应格式为字符串，
- "arraybuffer"——响应格式为`ArrayBuffer`（对于二进制数据，请参见`ArrayBuffer`, 二进制数组），
- "blob"——响应格式为`Blob`（对于二进制数据，请参见`Blob`），
- "document"——响应格式为`XML document`（可以使用`XPath`和其他`XML`方法），
- "json"——响应格式为`JSON`（自动解析）。

例如，我们以 JSON 格式获取响应：

```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/article/xmlhttprequest/example/json');

xhr.responseType = 'json';

xhr.send();

// 响应为 {"message": "Hello, world!"}
xhr.onload = function() {
 let responseObj = xhr.response;
 alert(responseObj.message); // Hello, world!
};
```

**i** **请注意:**

在旧的脚本中，你可能会看到 `xhr.responseText`，甚至会看到 `xhr.responseXML` 属性。

它们是由于历史原因而存在的，以获取字符串或 XML 文档。如今，我们应该在 `xhr.responseType` 中设置格式，然后就能获取如上所示的 `xhr.response` 了。

## readyState

`XMLHttpRequest` 的状态（state）会随着它的处理进度变化而变化。可以通过 `xhr.readyState` 来了解当前状态。

规范 [↗](#) 中提到的所有状态如下：

```
UNSENT = 0; // 初始状态
OPENED = 1; // open 被调用
HEADERS_RECEIVED = 2; // 接收到 response header
LOADING = 3; // 响应正在被加载（接收到一个数据包）
DONE = 4; // 请求完成
```

`XMLHttpRequest` 对象以 `0 → 1 → 2 → 3 → ... → 3 → 4` 的顺序在它们之间转变。每当通过网络接收到一个数据包，就会重复一次状态 `3`。

我们可以使用 `readystatechange` 事件来跟踪它们：

```
xhr.onreadystatechange = function() {
 if (xhr.readyState == 3) {
```

```
// 加载中
}
if (xhr.readyState == 4) {
 // 请求完成
}
};
```

你可能在非常老的代码中找到 `readystatechange` 这样的事件监听器，它的存在是有历史原因的，因为曾经有很长一段时间都没有 `load` 以及其他事件。如今，它已被 `load/error/progress` 事件处理程序所替代。

## 中止请求（Aborting）

我们可以随时终止请求。调用 `xhr.abort()` 即可：

```
xhr.abort(); // 终止请求
```

它会触发 `abort` 事件，且 `xhr.status` 变为 `0`。

## 同步请求

如果在 `open` 方法中将第三个参数 `async` 设置为 `false`，那么请求就会以同步的方式进行。

换句话说，JavaScript 执行在 `send()` 处暂停，并在收到响应后恢复执行。这有点儿像 `alert` 或 `prompt` 命令。

下面是重写的示例，`open` 的第三个参数为 `false`：

```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/article/xmlhttprequest/hello.txt', false);

try {
 xhr.send();
 if (xhr.status != 200) {
 alert(`Error ${xhr.status}: ${xhr.statusText}`);
 } else {
 alert(xhr.response);
 }
} catch(err) { // 代替 onerror
 alert("Request failed");
}
```

这看起来好像不错，但是很少使用同步调用，因为它们会阻塞页面内的 JavaScript，直到加载完成。在某些浏览器中，滚动可能无法正常进行。如果一个同步调用执行时间过长，浏览器可能会建议关闭“挂起（*hanging*）”的网页。

`XMLHttpRequest` 的很多高级功能在同步请求中都不可用，例如向其他域发起请求或者设置超时。并且，正如你所看到的，没有进度指示。

基于这些原因，同步请求使用的非常少，几乎从不使用。在这我们就不再讨论它了。

## HTTP-header

`XMLHttpRequest` 允许发送自定义 `header`，并且可以从响应中读取 `header`。

`HTTP-header` 有三种方法：

### `setRequestHeader(name, value)`

使用给定的 `name` 和 `value` 设置 `request header`。

例如：

```
xhr.setRequestHeader('Content-Type', 'application/json');
```

#### Header 的限制

一些 `header` 是由浏览器专门管理的，例如 `Referer` 和 `Host`。完整列表请见[规范](#)。

为了用户安全和请求的正确性，`XMLHttpRequest` 不允许更改它们。

#### 不能移除 `header`

`XMLHttpRequest` 的另一个特点是不能撤销 `setRequestHeader`。

一旦设置了 `header`，就无法撤销了。其他调用会向 `header` 中添加信息，但不会覆盖它。

例如：

```
xhr.setRequestHeader('X-Auth', '123');
xhr.setRequestHeader('X-Auth', '456');

// header 将是:
// X-Auth: 123, 456
```

## **getResponseHeader(name)**

获取具有给定 name 的 header ( Set-Cookie 和 Set-Cookie2 除外) 。

例如:

```
xhr.getResponseHeader('Content-Type')
```

## **getAllResponseHeaders()**

返回除 Set-Cookie 和 Set-Cookie2 外的所有 response header 。

header 以单行形式返回，例如:

```
Cache-Control: max-age=31536000
Content-Length: 4260
Content-Type: image/png
Date: Sat, 08 Sep 2012 16:53:16 GMT
```

header 之间的换行符始终为 "\r\n" (不依赖于操作系统)，所以我们可以很容易地将其拆分为单独的 header。name 和 value 之间总是以冒号后跟一个空格 ": " 分隔。这是标准格式。

因此，如果我们想要获取具有 name/value 对的对象，则需要用一点 JavaScript 代码来处理它们。

像这样（假设如果两个 header 具有相同的名称，那么后者就会覆盖前者）：

```
let headers = xhr
 .getAllResponseHeaders()
 .split('\r\n')
 .reduce((result, current) => {
 let [name, value] = current.split(': ');
 result[name] = value;
 return result;
 }, {});
// headers['Content-Type'] = 'image/png'
```

## **POST, FormData**

要建立一个 POST 请求，我们可以使用内建的 [FormData](#) 对象。

语法为:

```
let formData = new FormData([form]); // 创建一个对象，可以选择从 <form> 中获取数据
formData.append(name, value); // 附加一个字段
```

我们创建它，可以选择从一个表单中获取数据，如果需要，还可以 `append` 更多字段，然后：

1. `xhr.open('POST', ...)` —— 使用 `POST` 方法。
2. `xhr.send(formData)` 将表单发送到服务器。

例如：

```
<form name="person">
 <input name="name" value="John">
 <input name="surname" value="Smith">
</form>

<script>
 // 从表单预填充 FormData
 let formData = new FormData(document.forms.person);

 // 附加一个字段
 formData.append("middle", "Lee");

 // 将其发送出去
 let xhr = new XMLHttpRequest();
 xhr.open("POST", "/article/xmlhttprequest/post/user");
 xhr.send(formData);

 xhr.onload = () => alert(xhr.response);
</script>
```

以 `multipart/form-data` 编码发送表单。

或者，如果我们更喜欢 `JSON`，那么可以使用 `JSON.stringify` 并以字符串形式发送。

只是，不要忘记设置 `header Content-Type: application/json`，只要有了它，很多服务端框架都能自动解码 `JSON`：

```
let xhr = new XMLHttpRequest();

let json = JSON.stringify({
 name: "John",
 surname: "Smith"
});

xhr.open("POST", '/submit')
```

```
xhr.setRequestHeader('Content-type', 'application/json; charset=utf-8');

xhr.send(json);
```

`.send(body)` 方法就像一个非常杂食性的动物。它几乎可以发送任何 `body`，包括 `Blob` 和 `BufferSource` 对象。

## 上传进度

`progress` 事件仅在下载阶段触发。

也就是说：如果我们 `POST` 一些内容，`XMLHttpRequest` 首先上传我们的数据（`request body`），然后下载响应。

如果我们要上传的东西很大，那么我们肯定会对跟踪上传进度感兴趣。但是 `xhr.onprogress` 在这里并不起作用。

这里有另一个对象，它没有办法，它专门用于跟踪上传事件：`xhr.upload`。

它会生成事件，类似于 `xhr`，但是 `xhr.upload` 仅在上传时触发它们：

- `loadstart` —— 上传开始。
- `progress` —— 上传期间定期触发。
- `abort` —— 上传中止。
- `error` —— 非 HTTP 错误。
- `load` —— 上传成功完成。
- `timeout` —— 上传超时（如果设置了 `timeout` 属性）。
- `loadend` —— 上传完成，无论成功还是 `error`。

`handler` 示例：

```
xhr.upload.onprogress = function(event) {
 alert(`Uploaded ${event.loaded} of ${event.total} bytes`);
};

xhr.upload.onload = function() {
 alert(`Upload finished successfully.`);
};

xhr.upload.onerror = function() {
 alert(`Error during the upload: ${xhr.status}`);
};
```

这是一个真实示例：带有进度指示的文件上传：

```
<input type="file" onchange="upload(this.files[0])">

<script>
function upload(file) {
 let xhr = new XMLHttpRequest();

 // 跟踪上传进度
 xhr.upload.onprogress = function(event) {
 console.log(`Uploaded ${event.loaded} of ${event.total}`);
 };

 // 跟踪完成: 无论成功与否
 xhr.onloadend = function() {
 if (xhr.status == 200) {
 console.log("success");
 } else {
 console.log("error " + this.status);
 }
 };
}

xhr.open("POST", "/article/xmlhttprequest/post/upload");
xhr.send(file);
}
</script>
```

## 跨源请求

`XMLHttpRequest` 可以使用和 `fetch` 相同的 CORS 策略进行跨源请求。

就像 `fetch` 一样, 默认情况下不会将 `cookie` 和 `HTTP` 授权发送到其他域。要启用它们, 可以将 `xhr.withCredentials` 设置为 `true`:

```
let xhr = new XMLHttpRequest();
xhr.withCredentials = true;

xhr.open('POST', 'http://anywhere.com/request');
...
```

有关跨源 `header` 的详细信息, 请见 [Fetch: 跨源请求](#) 一章。

## 总结

使用 `XMLHttpRequest` 的 `GET` 请求的典型代码:

```
let xhr = new XMLHttpRequest();
```

```
xhr.open('GET', '/my/url');

xhr.send();

xhr.onload = function() {
 if (xhr.status != 200) { // HTTP error?
 // 处理 error
 alert('Error: ' + xhr.status);
 return;
 }

 // 获取来自 xhr.response 的响应
};

xhr.onprogress = function(event) {
 // 报告进度
 alert(`Loaded ${event.loaded} of ${event.total}`);
};

xhr.onerror = function() {
 // 处理非 HTTP error (例如网络中断)
};
```

实际上还有很多事件，在[现代规范](#) 中有详细列表（按生命周期排序）：

- `loadstart` —— 请求开始。
- `progress` —— 一个响应数据包到达，此时整个 `response body` 都在 `response` 中。
- `abort` —— 调用 `xhr.abort()` 取消了请求。
- `error` —— 发生连接错误，例如，域错误。不会发生诸如 404 这类的 HTTP 错误。
- `load` —— 请求成功完成。
- `timeout` —— 由于请求超时而取消了该请求（仅发生在设置了 `timeout` 的情况下）。
- `loadend` —— 在 `load`, `error`, `timeout` 或 `abort` 之后触发。

`error`, `abort`, `timeout` 和 `load` 事件是互斥的。其中只有一种可能发生。

最常用的事件是加载完成（`load`），加载失败（`error`），或者我们可以使用单个 `loadend` 处理程序并检查请求对象 `xhr` 的属性，以查看发生了什么。

我们还了解了另一个事件：`readystatechange`。由于历史原因，它早在规范制定之前就出现了。如今我们已经无需使用它了，我们可以用新的事件代替它，但通常可以在旧的代码中找到它。

如果我们需要专门跟踪上传，那么我们应该在 `xhr.upload` 对象上监听相同的事件。

# 可恢复的文件上传

使用 `fetch` 方法来上传文件相当容易。

连接断开后如何恢复上传？这里没有对此的内建选项，但是我们有实现它的一些方式。

对于大文件（如果我们可能需要恢复），可恢复的上传应该带有上传进度提示。由于 `fetch` 不允许跟踪上传进度，我们将会使用 [XMLHttpRequest](#)。

## 不太实用的进度事件

要恢复上传，我们需要知道在连接断开前已经上传了多少。

我们有 `xhr.upload.onprogress` 来跟踪上传进度。

不幸的是，它不会帮助我们在此处恢复上传，因为它会在数据 **被发送** 时触发，但是服务器是否接收到？浏览器并不知道。

或许它是由本地网络代理缓冲的（**buffered**），或者可能是远程服务器进程刚刚终止而无法处理它们，亦或是它在中间丢失了，并没有到达服务器。

这就是为什么此事件仅适用于显示一个好看的进度条。

要恢复上传，我们需要 **确切地** 知道服务器接收的字节数。而且只有服务器能告诉我们，因此，我们将发出一个额外的请求。

## 算法

1. 首先，创建一个文件 `id`，以唯一地标识我们要上传的文件：

```
let fileId = file.name + '-' + file.size + '-' + file.lastModifiedDate;
```

在恢复上传时需要用到它，以告诉服务器我们要恢复的内容。

如果名称，或大小，或最后一次修改事件发生了更改，则将有另一个 `fileId`。

2. 向服务器发送一个请求，询问它已经有了多少字节，像这样：

```
let response = await fetch('status', {
 headers: {
 'X-File-Id': fileId
 }
});

// 服务器已有的字节数
let startByte = +await response.text();
```

这假设服务器通过 `X-File-Id` header 跟踪文件上传。应该在服务端实现。

如果服务器上尚不存在该文件，则服务器响应应为 `0`。

3. 然后，我们可以使用 `Blob` 和 `slice` 方法来发送从 `startByte` 开始的文件：

```
xhr.open("POST", "upload", true);

// 文件 id, 以便服务器知道我们要恢复的是哪个文件
xhr.setRequestHeader('X-File-Id', fileId);

// 发送我们要从哪个字节开始恢复, 因此服务器知道我们正在恢复
xhr.setRequestHeader('X-Start-Byte', startByte);

xhr.upload.onprogress = (e) => {
 console.log(`Uploaded ${startByte + e.loaded} of ${startByte + e.total}`);
};

// 文件可以是来自 input.files[0], 或者另一个源
xhr.send(file.slice(startByte));
```

这里我们将文件 `id` 作为 `X-File-Id` 发送给服务器，所以服务器知道我们正在上传哪个文件，并且，我们还将起始字节作为 `X-Start-Byte` 发送给服务器，所以服务器知道我们不是重新上传它，而是恢复其上传。

服务器应该检查其记录，如果有一个上传的该文件，并且当前已上传的文件大小恰好是 `X-Start-Byte`，那么就将数据附加到该文件。

这是用 `Node.js` 写的包含客户端和服务端代码的示例。

在本网站上，它只有部分能工作，因为 `Node.js` 位于另一个服务 `Nginx` 后面，该服务器缓冲（`buffer`）上传的内容，当完全上传后才将其传递给 `Node.js`。

但是你可以下载这些代码，在本地运行以进行完整演示：

<https://plnkr.co/edit/uxj1jt1eMi25C7q1?p=preview>

正如我们所看到的，现代网络方法在功能上已经与文件管理器非常接近——控制 `header`，进度指示，发送文件片段等。

我们可以实现可恢复的上传等。

## 长轮询（Long polling）

长轮询是与服务器保持持久连接的最简单的方式，它不使用任何特定的协议，例如 `WebSocket` 或者 `Server Sent Event`。

它很容易实现，在很多场景下也很好用。

## 常规轮询

从服务器获取新信息的最简单的方式是定期轮询。也就是说，定期向服务器发出请求：“你好，我在这儿，你有关于我的任何信息吗？”例如，每 10 秒一次。

作为响应，服务器首先通知自己，客户端处于在线状态，然后——发送目前为止的消息包。

这可行，但是也有些缺点：

1. 消息传递的延迟最多为 10 秒（两个请求之间）。
2. 即使没有消息，服务器也会每隔 10 秒被请求轰炸一次，即使用户切换到其他地方或者处于休眠状态，也是如此。就性能而言，这是一个很大的负担。

因此，如果我们讨论的是一个非常小的服务，那么这种方式可能可行，但总的来说，它需要改进。

## 长轮询

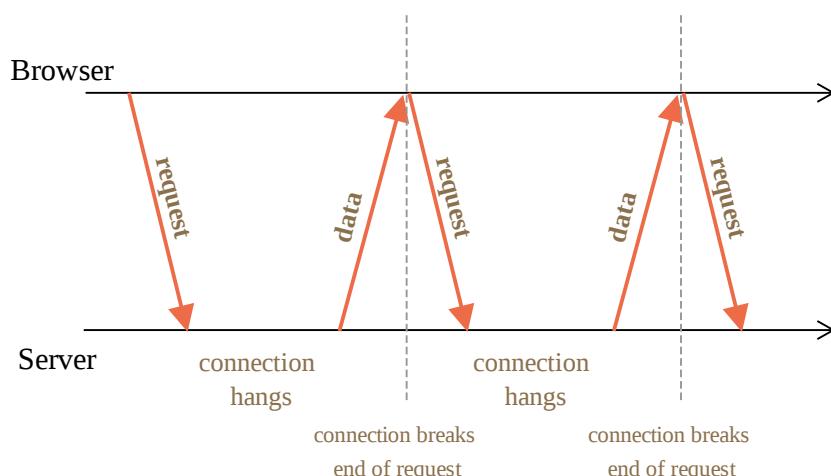
所谓“长轮询”是轮询服务器的一种更好的方式。

它也很容易实现，并且可以无延迟地传递消息。

其流程为：

1. 请求发送到服务器。
2. 服务器在有消息之前不会关闭连接。
3. 当消息出现时——服务器将对其请求作出响应。
4. 浏览器立即发出一个新的请求。

对此方法，浏览器发出一个请求并与服务器之间建立起一个挂起的（pending）连接的情况是标准的。仅在有消息被传递时，才会重新建立连接。



如果连接丢失，可能是因为网络错误，浏览器会立即发送一个新请求。

实现长轮询的客户端 `subscribe` 函数的示例代码：

```
async function subscribe() {
 let response = await fetch("/subscribe");

 if (response.status == 502) {
 // 状态 502 是连接超时错误,
 // 连接挂起时间过长时可能会发生,
 // 远程服务器或代理会关闭它
 // 让我们重新连接
 await subscribe();
 } else if (response.status != 200) {
 // 一个 error — 让我们显示它
 showMessage(response.statusText);
 // 一秒后重新连接
 await new Promise(resolve => setTimeout(resolve, 1000));
 await subscribe();
 } else {
 // 获取并显示消息
 let message = await response.text();
 showMessage(message);
 // 再次调用 subscribe() 以获取下一条消息
 await subscribe();
 }
}

subscribe();
```

正如你所看到的，`subscribe` 函数发起了一个 `fetch`，然后等待响应，处理它，并再次调用自身。

### ⚠ 服务器应该可以处理许多挂起的连接

服务器架构必须能够处理许多挂起的连接。

某些服务器架构是每个连接对应一个进程。对于许多连接的情况，将会有许多进程，并且每个进程占用大量内存。因此，过多的连接会消耗掉全部内存。

使用 `PHP`, `Ruby` 语言编写的后端程序会经常遇到这个问题，但是从技术上讲，它不是语言问题，而是实现问题。大多数现代编程语言都允许实现适当的后端，但是其中一些语言比其他语言更容易实现。

使用 `Node.js` 写的后端通常不会出现这样的问题。

## 示例：聊天

这是一个聊天演示，你可以下载它并在本地运行（如果你熟悉 Node.js 并且可以安装模块）：

<https://plnkr.co/edit/iVZCSORDTwZkQmbR?p=preview>

浏览器代码在 `browser.js` 中。

## 使用场景

在消息很少的情况下，长轮询很有效。

如果消息比较频繁，那么上面描绘的请求-接收（requesting-receiving）消息的图表就会变成锯状状（saw-like）。

每个消息都是一个单独的请求，并带有 `header`, 身份验证开销（authentication overhead）等。

因此，在这种情况下，首选另一种方法，例如：[WebSocket](#) 或 [Server Sent Events](#)。

## WebSocket

在 [RFC 6455](#) 规范中描述的 `WebSocket` 协议提供了一种在浏览器和服务器之间建立持久连接来交换数据的方法。数据可以作为“数据包”在两个方向上传递，而不会断开连接和其他 `HTTP` 请求。

对于需要连续数据交换的服务，例如网络游戏，实时交易系统等，`WebSocket` 尤其有用。

## 一个简单例子

要打开一个 `WebSocket` 连接，我们需要在 `url` 中使用特殊的协议 `ws` 创建 `new WebSocket`：

```
let socket = new WebSocket("ws://javascript.info");
```

同样也有一个加密的 `wss://` 协议。类似于 `WebSocket` 中的 `HTTPS`。

## i 始终使用 `wss://`

`wss://` 协议不仅能加密，而且更可靠。

因为 `ws://` 数据不是加密的，对于任何中间人来说其数据都是可见的。并且，旧的代理服务器不了解 WebSocket，它们可能会因为看到“奇怪的” header 而中止连接。

另一方面，`wss://` 是基于 TLS 的 WebSocket，类似于 HTTPS 是基于 TLS 的 HTTP），传输安全层在发送方对数据进行了加密，在接收方进行解密。因此，数据包是通过代理加密传输的。它们看不到传输的里面的内容，且会让这些数据通过。

一旦 socket 被建立，我们就应该监听 socket 上的事件。一共有 4 个事件：

- `open` —— 连接已建立，
- `message` —— 接收到数据，
- `error` —— WebSocket 错误，
- `close` —— 连接已关闭。

.....如果我们想发送一些东西，那么可以使用 `socket.send(data)`。

这是一个示例：

```
let socket = new WebSocket("wss://javascript.info/article/websocket/demo/hello");

socket.onopen = function(e) {
 alert("[open] Connection established");
 alert("Sending to server");
 socket.send("My name is John");
};

socket.onmessage = function(event) {
 alert(`[message] Data received from server: ${event.data}`);
};

socket.onclose = function(event) {
 if (event.wasClean) {
 alert(`[close] Connection closed cleanly, code=${event.code} reason=${event.reason}`);
 } else {
 // 例如服务器进程被杀死或网络中断
 // 在这种情况下，event.code 通常为 1006
 alert('[close] Connection died');
 }
};

socket.onerror = function(error) {
 alert(`[error] ${error.message}`);
};
```

出于演示目的，在上面的示例中，运行着一个用 Node.js 写的小型服务器 `server.js`。它响应为“Hello from server, John”，然后等待 5 秒，关闭连接。

所以你看到的事件顺序为：`open` → `message` → `close`。

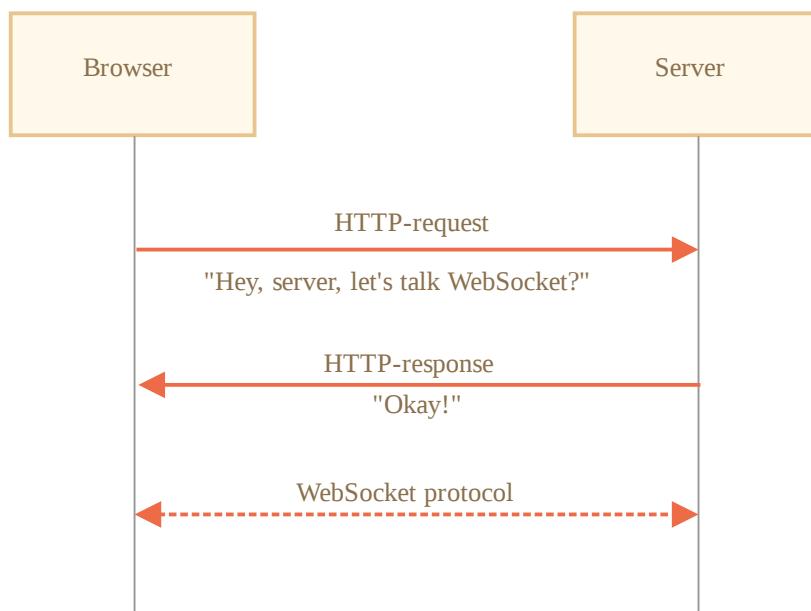
这就是 WebSocket，我们已经可以使用 WebSocket 通信了。很简单，不是吗？

现在让我们更深入地学习它。

## 建立 WebSocket

当 `new WebSocket(url)` 被创建后，它将立即开始连接。

在连接期间，浏览器（使用 `header`）问服务器：“你支持 WebSocket 吗？”如果服务器回复说“我支持”，那么通信就以 WebSocket 协议继续进行，该协议根本不是 HTTP。



这是由 `new WebSocket("wss://javascript.info/chat")` 发出的请求的浏览器 `header` 示例。

```
GET /chat
Host: javascript.info
Origin: https://javascript.info
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==
Sec-WebSocket-Version: 13
```

- `Origin` —— 客户端页面的源，例如 `https://javascript.info`。WebSocket 对象是原生支持跨源的。没有特殊的 `header` 或其他限制。旧的服务器无法处理 WebSocket，因此不存在兼容性问题。但是 `Origin header` 很重要，因为它允许服务器决定是否使用 WebSocket 与该网站通信。
- `Connection: Upgrade` —— 表示客户端想要更改协议。
- `Upgrade: websocket` —— 请求的协议是“websocket”。
- `Sec-WebSocket-Key` —— 浏览器随机生成的安全密钥。
- `Sec-WebSocket-Version` —— WebSocket 协议版本，当前为 13。

### 无法模拟 WebSocket 握手

我们不能使用 `XMLHttpRequest` 或 `fetch` 来进行这种 HTTP 请求，因为不允许 JavaScript 设置这些 `header`。

如果服务器同意切换为 WebSocket 协议，服务器应该返回响应码 101：

```
101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: hsB1buDTkk24srzE0TBu1ZA1C2g=
```

这里 `Sec-WebSocket-Accept` 是 `Sec-WebSocket-Key`，是使用特殊的算法重新编码的。浏览器使用它来确保响应与请求相对应。

然后，就使用 WebSocket 协议传输数据，我们很快就会看到它的结构（“frames”）。它根本不是 HTTP。

## 扩展和子协议

WebSocket 可能还有其他 `header`，`Sec-WebSocket-Extensions` 和 `Sec-WebSocket-Protocol`，它们描述了扩展和子协议。

例如：

- `Sec-WebSocket-Extensions: deflate-frame` 表示浏览器支持数据压缩。扩展与传输数据有关，扩展了 WebSocket 协议的功能。`Sec-WebSocket-Extensions header` 有浏览器自动发送，其中包含其支持的所有扩展的列表。
- `Sec-WebSocket-Protocol: soap, wamp` 表示我们不仅要传输任何数据，还要传输 [SOAP](#) 或 WAMP（“The WebSocket Application Messaging Protocol”）协议中的数据。WebSocket 子协议已经在 [IANA catalogue](#) 中注册。

这个可选的 `header` 是使用 `new WebSocket` 的第二个参数设置的。它是子协议数组，例如，如果我们想使用 SOAP 或 WAMP：

```
let socket = new WebSocket("wss://javascript.info/chat", ["soap", "wamp"]);
```

服务器应该使用同意使用的协议和扩展的列表进行响应。

例如，这个请求：

```
GET /chat
Host: javascript.info
Upgrade: websocket
Connection: Upgrade
Origin: https://javascript.info
Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==
Sec-WebSocket-Version: 13
Sec-WebSocket-Extensions: deflate-frame
Sec-WebSocket-Protocol: soap, wamp
```

响应：

```
101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: hsBlbuDTkk24srzE0TBu1ZAlC2g=
Sec-WebSocket-Extensions: deflate-frame
Sec-WebSocket-Protocol: soap
```

在这里服务器响应——它支持扩展“deflate-frame”，并且仅支持所请求的子协议中的 SOAP。

## 数据传输

WebSocket 通信由“frames”（即数据片段）组成，可以从任何一方发送，并且有以下几种类型：

- “text frames”——包含各方发送给彼此的文本数据。
- “binary data frames”——包含各方发送给彼此的二进制数据。
- “ping/pong frames”被用于检查从服务器发送的连接，浏览器会自动响应它们。
- 还有“connection close frame”以及其他服务 frames。

在浏览器里，我们仅直接使用文本或二进制 frames。

**WebSocket .send()** 方法可以发送文本或二进制数据。

`socket.send(body)` 调用允许 `body` 是字符串或二进制格式，包括 `Blob`，`ArrayBuffer` 等。不需要额外的设置：直接发送它们就可以了。

当我们收到数据时，文本总是以字符串形式呈现。而对于二进制数据，我们可以在 **Blob** 和 **ArrayBuffer** 格式之间进行选择。

它是由 `socket.bufferType` 属性设置的，默认为 "blob"，因此二进制数据通常以 **Blob** 对象呈现。

**Blob** 是高级的二进制对象，它直接与 `<a>`，`<img>` 及其他标签集成在一起，因此，默认以 **Blob** 格式是一个明智的选择。但是对于二进制处理，要访问单个数据字节，我们可以将其改为 "arraybuffer"：

```
socket.bufferType = "arraybuffer";
socket.onmessage = (event) => {
 // event.data 可以是文本（如果是文本），也可以是 arraybuffer（如果是二进制数据）
};
```

## 限速

想象一下：我们的应用程序正在生成大量要发送的数据。但是用户的网速却很慢，可能是在乡下的移动设备上。

我们可以反复地调用 `socket.send(data)`。但是数据将会缓冲（储存）在内存中，并且只能在网速允许的情况下尽快将数据发送出去。

`socket.bufferedAmount` 属性储存目前已缓冲的字节数，等待通过网络发送。

我们可以检查它以查看 `socket` 是否真的可用于传输。

```
// 每 100ms 检查一次 socket
// 仅当所有现有的数据都已被发送出去时，再发送更多数据
setInterval(() => {
 if (socket.bufferedAmount == 0) {
 socket.send(moreData());
 }
}, 100);
```

## 连接关闭

通常，当一方想要关闭连接时（浏览器和服务器都具有相同的权限），它们会发送一个带有数字码（numeric code）和文本形式的原因的“connection close frame”。

它的方法是：

```
socket.close([code], [reason]);
```

- `code` 是一个特殊的 WebSocket 关闭码（可选）

- `reason` 是一个描述关闭原因的字符串（可选）

然后，另外一方通过 `close` 事件处理器获取了关闭码和关闭原因，例如：

```
// 关闭方:
socket.close(1000, "Work complete");

// 另一方
socket.onclose = event => {
 // event.code === 1000
 // event.reason === "Work complete"
 // event.wasClean === true (clean close)
};
```

最常见的数字码：

- `1000` —— 默认，正常关闭（如果没有指明 `code` 时使用它），
- `1006` —— 没有办法手动设定这个数字码，表示连接丢失（没有 `close frame`）。

还有其他数字码，例如：

- `1001` —— 一方正在离开，例如服务器正在关闭，或者浏览器离开了该页面，
- `1009` —— 消息太大，无法处理，
- `1011` —— 服务器上发生意外错误，
- ..... 等。

完整列表请见 [RFC6455, §7.4.1](#)。

WebSocket 码有点像 HTTP 码，但它们是不同的。特别是，小于 `1000` 的码都是被保留的，如果我们尝试设置这样的码，将会出现错误。

```
// 再连接断开的情况下
socket.onclose = event => {
 // event.code === 1006
 // event.reason === ""
 // event.wasClean === false (未关闭 frame)
};
```

## 连接状态

要获取连接状态，可以通过带有值的 `socket.readyState` 属性：

- `0` —— “CONNECTING”：连接还未建立，
- `1` —— “OPEN”：通信中，

- **2** —— “CLOSING”: 连接关闭中,
- **3** —— “CLOSED”: 连接已关闭。

## 聊天示例

让我们来看一个使用浏览器 WebSocket API 和 Node.js 的 WebSocket 模块 <https://github.com/websockets/ws> 的聊天示例。我们将主要精力放在客户端上，但是服务端也很简单。

**HTML:** 我们需要一个 `<form>` 来发送消息，并且需要一个 `<div>` 来接收消息：

```
<!-- 消息表单 -->
<form name="publish">
 <input type="text" name="message">
 <input type="submit" value="Send">
</form>

<!-- 带有消息的 div -->
<div id="messages"></div>
```

在 JavaScript 中，我们想要做三件事：

1. 打开连接。
2. 在表单提交中 —— `socket.send(message)` 用于消息。
3. 对于传入的消息 —— 将其附加（append）到 `div#messages` 上。

代码如下

```
let socket = new WebSocket("wss://javascript.info/article/websocket/chat/ws");

// 从表单发送消息
document.forms.publish.onsubmit = function() {
 let outgoingMessage = this.message.value;

 socket.send(outgoingMessage);
 return false;
};

// 收到消息 — 在 div#messages 中显示消息
socket.onmessage = function(event) {
 let message = event.data;

 let messageElem = document.createElement('div');
 messageElem.textContent = message;
 document.getElementById('messages').prepend(messageElem);
}
```

服务端代码有点超出我们的范围。在这里，我们将使用 Node.js，但你不必这样做。其他平台也有使用 WebSocket 的方法。

服务器端的算法为：

1. 创建 `clients = new Set()` —— 一系列 socket。
2. 对于每个被接受的 WebSocket，将其添加到 `clients.add(socket)`，并为其设置 `message` 事件侦听器以获取其消息。
3. 当接收到消息：便利客户端，并将消息发送给所有人。
4. 当连接被关闭：`clients.delete(socket)`。

```
const ws = new require('ws');
const wss = new ws.Server({noServer: true});

const clients = new Set();

http.createServer((req, res) => {
 // 在这里，我们仅处理 WebSocket 连接
 // 在实际项目中，我们在这里还会有其他代码，来处理非 WebSocket 请求
 wss.handleUpgrade(req, req.socket, Buffer.alloc(0), onSocketConnect);
});

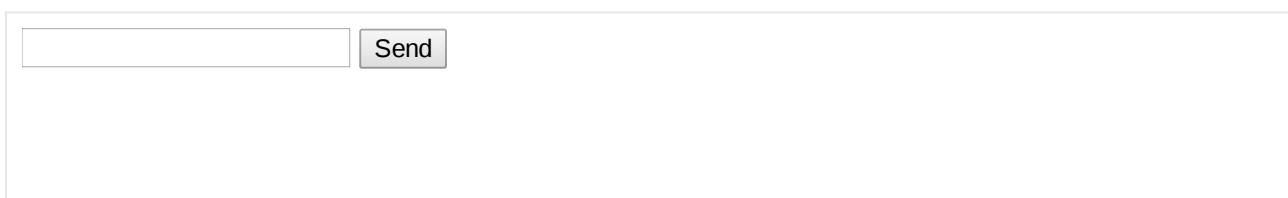
function onSocketConnect(ws) {
 clients.add(ws);

 ws.on('message', function(message) {
 message = message.slice(0, 50); // message 的最大长度为 50

 for(let client of clients) {
 client.send(message);
 }
 });

 ws.on('close', function() {
 clients.delete(ws);
 });
}
```

这是运行示例：



你也可以下载它（点击 iframe 右上角的按钮）然后在本地运行。运行之前请记得安装 [Node.js](#) 和 [npm install ws](#)。

## 总结

WebSocket 是一种在浏览器和服务器之间建立持久连接的现代方式。

- WebSocket 没有跨源限制。
- 浏览器对 WebSocket 支持很好。
- 可以发送/接收字符串和二进制数据。

WebSocket 的 API 很简单。

WebSocket 方法:

- `socket.send(data)`,
- `socket.close([code], [reason])`。

WebSocket 事件:

- `open`,
- `message`,
- `error`,
- `close`。

WebSocket 自身并不包含重新连接 (reconnection)，身份验证 (authentication) 和很多其他高级机制。因此，有针对于此的客户端/服务端的库，并且也可以手动实现这些功能。

有时为了将 WebSocket 集成到现有项目中，人们将主 HTTP 服务器与 WebSocket 服务器并行运行，并且它们之间共享同一个数据库。对于 WebSocket 请求使用一个通向 WebSocket 服务器的子域 `wss://ws.site.com`，而 `https://site.com` 则通向主 HTTP 服务器。

当然，其他集成方式也是可行的。

## Server Sent Events

[Server-Sent Events ↗](#) 规范描述了一个内建的类 `EventSource`，它能保持与服务器的连接，并允许从中接收事件。

与 `WebSocket` 类似，其连接是持久的。

但是两者之间有几个重要的区别:

WebSocket	EventSource
双向：客户端和服务端都能交换消息	单向：仅服务端能发送消息
二进制和文本数据	仅文本数据
WebSocket 协议	常规 HTTP 协议

与 `WebSocket` 相比，`EventSource` 是与服务器通信的一种不那么强大的方式。

我们为什么要使用它？

主要原因：简单。在很多应用中，`WebSocket` 有点大材小用。

我们需要从服务器接收一个数据流：可能是聊天消息或者市场价格等。这正是 `EventSource` 所擅长的。它还支持自动重新连接，而在 `WebSocket` 中这个功能需要我们手动实现。此外，它是一个普通的旧的 `HTTP`，不是一个新协议。

## 获取消息

要开始接收消息，我们只需要创建 `new EventSource(url)` 即可。

浏览器将会连接到 `url` 并保持连接打开，等待事件。

服务器响应状态码应该为 `200`，`header` 为 `Content-Type: text/event-stream`，然后保持此连接并以一种特殊的格式写入消息，就像这样：

```
data: Message 1
data: Message 2
data: Message 3
data: of two lines
```

- `data:` 后为消息文本，冒号后面的空格是可选的。
- 消息以双换行符 `\n\n` 分隔。
- 要发送一个换行 `\n`，我们可以在要换行的位置立即再发送一个 `data:`（上面的第三条消息）。

在实际开发中，复杂的消息通常是用 `JSON` 编码后发送。换行符在其中编码为 `\n`，因此不需要多行 `data:` 消息。

例如：

```
data: {"user": "John", "message": "First line\n Second line"}
```

.....因此，我们可以假设一个 `data:` 只保存了一条消息。

对于每个这样的消息，都会生成 `message` 事件：

```
let eventSource = new EventSource("/events/subscribe");
eventSource.onmessage = function(event) {
```

```
console.log("New message", event.data);
// 对于上面的数据流将打印三次
};

// 或 eventSource.addEventListener('message', ...)
```

## 跨源请求

`EventSource` 支持跨源请求，就像 `fetch` 任何其他网络方法。我们可以使用任何 URL：

```
let source = new EventSource("https://another-site.com/events");
```

远程服务器将会获取到 `Origin` header，并且必须以 `Access-Control-Allow-Origin` 响应来处理。

要传递凭证（`credentials`），我们应该设置附加选项 `withCredentials`，就像这样：

```
let source = new EventSource("https://another-site.com/events", {
 withCredentials: true
});
```

更多关于跨源 `header` 的详细内容，请参见 [Fetch: 跨源请求](#)。

## 重新连接

创建之后，`new EventSource` 连接到服务器，如果连接断开——则重新连接。

这非常方便，我们不用去关心重新连接的事情。

每次重新连接之间有一点小的延迟，默认为几秒钟。

服务器可以使用 `retry:` 来设置需要的延迟响应时间（以毫秒为单位）。

```
retry: 15000
data: Hello, I set the reconnection delay to 15 seconds
```

`retry:` 既可以与某些数据一起出现，也可以作为独立的消息出现。

在重新连接之前，浏览器需要等待那么多毫秒。甚至更长，例如，如果浏览器知道（从操作系统）此时没有网络连接，它会等到连接出现，然后重试。

- 如果服务器想要浏览器停止重新连接，那么它应该使用 `HTTP` 状态码 `204` 进行响应。

- 如果浏览器想要关闭连接，则应该调用 `eventSource.close()`：

```
let eventSource = new EventSource(...);

eventSource.close();
```

并且，如果响应具有不正确的 `Content-Type` 或者其 HTTP 状态码不是 301, 307, 200 和 204，则不会进行重新连接。在这种情况下，将会发出 `"error"` 事件，并且浏览器不会重新连接。

**i** **请注意:**

当连接最终被关闭时，就无法“重新打开”它。如果我们想要再次连接，只需要创建一个新的 `EventSource`。

## 消息 id

当一个连接由于网络问题而中断时，客户端和服务器都无法确定哪些消息已经收到哪些没有收到。

为了正确地恢复连接，每条消息都应该有一个 `id` 字段，就像这样：

```
data: Message 1
id: 1

data: Message 2
id: 2

data: Message 3
data: of two lines
id: 3
```

当收到具有 `id` 的消息时，浏览器会：

- 将属性 `eventSource.lastEventId` 设置为其值。
- 重新连接后，发送带有 `id` 的 header `Last-Event-ID`，以便服务器可以重新发送后面的消息。

**i** **把 `id:` 放在 `data:` 后**

请注意：`id` 被服务器附加到 `data` 消息后，以确保在收到消息后 `lastEventId` 会被更新。

## 连接状态: readyState

`EventSource` 对象有 `readyState` 属性, 该属性具有下列值之一:

```
EventSource.CONNECTING = 0; // 连接中或者重连中
EventSource.OPEN = 1; // 已连接
EventSource.CLOSED = 2; // 连接已关闭
```

对象创建完成或者连接断开后, 它始终是 `EventSource.CONNECTING` (等于 `0`)。

我们可以查询该属性以了解 `EventSource` 的状态。

## Event 类型

默认情况下 `EventSource` 对象生成三个事件:

- `message` —— 收到消息, 可以用 `event.data` 访问。
- `open` —— 连接已打开。
- `error` —— 无法建立连接, 例如, 服务器返回 HTTP 500 状态码。

服务器可以在事件开始时使用 `event: ...` 指定另一种类型事件。

例如:

```
event: join
data: Bob

data: Hello

event: leave
data: Bob
```

要处理自定义事件, 我们必须使用 `addEventListener` 而非 `onmessage`:

```
eventSource.addEventListener('join', event => {
 alert(`Joined ${event.data}`);
});

eventSource.addEventListener('message', event => {
 alert(`Said: ${event.data}`);
});

eventSource.addEventListener('leave', event => {
```

```
 alert(`Left ${event.data}`);
});
```

## 完整示例

服务器依次发送 `1`, `2`, `3`, 最后发送 `bye` 并断开连接。

然后浏览器会自动重新连接。

<https://plnkr.co/edit/NnYjoCPNrx1ujFfc?p=preview>

## 总结

`EventSource` 对象自动建立一个持久的连接，并允许服务器通过这个连接发送消息。

它提供了：

- 在可调的 `retry` 超时内自动重新连接。
- 用于恢复事件的消息 `id`, 重新连接后, 最后接收到的标识符被在 `Last-Event-ID` header 中发送出去。
- 当前状态位于 `readyState` 属性中。

这使得 `EventSource` 成为 `WebSocket` 的一个可行的替代方案，因为 `WebSocket` 更低级 (`low-level`)，且缺乏这样的内建功能（尽管它们可以被实现）。

在很多实际应用中，`EventSource` 的功能就已经够用了。

`EventSource` 在所有现代浏览器（除了 IE）中都得到了支持。

语法：

```
let source = new EventSource(url, [credentials]);
```

第二个参数只有一个可选项：`{ withCredentials: true }`，它允许发送跨源凭证。

总体跨源安全性与 `fetch` 以及其他网络方法相同。

## EventSource 对象的属性

### readyState

当前连接状态：为 `EventSource.CONNECTING (=0)`, `EventSource.OPEN (=1)`, `EventSource.CLOSED (=2)` 三者之一。

## **lastEventId**

最后接收到的 `id`。重新连接后，浏览器在 header `Last-Event-ID` 中发送此 `id`。

## **EventSource 对象的方法**

### **close()**

关闭连接。

## **EventSource 对象的事件**

### **message**

接收到的消息，消息数据在 `event.data` 中。

### **open**

连接已建立。

### **error**

如果发生错误，包括连接丢失（将会自动重连）以及其他致命错误。我们可以检查 `readyState` 以查看是否正在尝试重新连接。

服务器可以在 `event:` 中设置自定义事件名称。应该使用 `addEventListener` 来处理此类事件，而不是使用 `on<event>`。

## **服务器响应格式**

服务器发送由 `\n\n` 分隔的消息。

一条消息可能有以下字段：

- `data:` —— 消息体（body），一系列多个 `data` 被解释为单个消息，各个部分之间由 `\n` 分隔。
- `id:` —— 更新 `lastEventId`，重连时以 `Last-Event-ID` 发送此 `id`。
- `retry:` —— 建议重连的延迟，以 ms 为单位。无法通过 JavaScript 进行设置。
- `event:` —— 事件名，必须在 `data:` 之前。

一条消息可以按任何顺序包含一个或多个字段，但是 `id:` 通常排在最后。

## **在浏览器中存储数据**

### **Cookie, document.cookie**

**Cookie** 是直接存储在浏览器中的一小串数据。它们是 **HTTP** 协议的一部分，由 [RFC 6265](#) 规范定义。

**Cookie** 通常是由 **Web** 服务器使用响应 `Set-Cookie` **HTTP-header** 设置的。然后浏览器使用 `Cookie` **HTTP-header** 将它们自动添加到（几乎）每个对相同域的请求

中。

最常见的用处之一就是身份验证:

1. 登录后，服务器在响应中使用 `Set-Cookie` HTTP-header 来设置具有唯一“会话标识符（session identifier）”的 cookie。
2. 下次如果请求是由相同域发起的，浏览器会使用 `Cookie` HTTP-header 通过网络发送 cookie。
3. 所以服务器知道是谁发起了请求。

我们还可以使用 `document.cookie` 属性从浏览器访问 cookie。

关于 cookie 及其选项，有很多棘手的事情。在本章中，我们将详细介绍它们。

## 从 `document.cookie` 中读取

假设你在一个网站上，则可以看到来自该网站的 cookie，像这样:

```
// 在 javascript.info，我们使用谷歌分析来进行统计，
// 所以应该存在一些 cookie
alert(document.cookie); // cookie1=value1; cookie2=value2;...
```

`document.cookie` 的值由 `name=value` 对组成，以 ; 分隔。每一个都是独立的 cookie。

为了找到一个特定的 cookie，我们可以以 ; 作为分隔，将 `document.cookie` 分开，然后找到对应的名字。我们可以使用正则表达式或者数组函数来实现。

我们把这个留给读者当作练习。此外，在本章的最后，你可以找到一些操作 cookie 的辅助函数。

## 写入 `document.cookie`

我们可以写入 `document.cookie`。但这不是一个数据属性，它是一个访问器（getter/setter）。对其的赋值操作会被特殊处理。

对 `document.cookie` 的写入操作只会更新其中提到的 cookie，而不会涉及其他 cookie。

例如，此调用设置了一个名称为 `user` 且值为 `John` 的 cookie:

```
document.cookie = "user=John"; // 只会更新名称为 user 的 cookie
alert(document.cookie); // 展示所有 cookie
```

如果你运行了上面这段代码，你会看到多个 cookie。这是因为 `document.cookie=` 操作不是重写整所有 cookie。它只设置代码中提到的 cookie user。

从技术上讲，cookie 的名称和值可以是任何字符，为了保持有效的格式，它们应该使用内建的 `encodeURIComponent` 函数对其进行转义：

```
// 特殊字符（空格），需要编码
let name = "my name";
let value = "John Smith"

// 将 cookie 编码为 my%20name=John%20Smith
document.cookie = encodeURIComponent(name) + '=' + encodeURIComponent(value);

alert(document.cookie); // ...; my%20name=John%20Smith
```

### ⚠ 限制

存在一些限制：

- `encodeURIComponent` 编码后的 `name=value` 对，大小不能超过 4kb。因此，我们不能在一个 cookie 中保存大的东西。
- 每个域的 cookie 总数不得超过 20+ 左右，具体限制取决于浏览器。

Cookie 有几个选项，其中很多都很重要，应该设置它。

选项被列在 `key=value` 之后，以 ; 分隔，像这样：

```
document.cookie = "user=John; path=/; expires=Tue, 19 Jan 2038 03:14:07 GMT"
```

## path

- `path=/mypath`

`url` 路径前缀，该路径下的页面可以访问该 cookie。必须是绝对路径。默认为当前路径。

如果一个 cookie 带有 `path=/admin` 设置，那么该 cookie 在 `/admin` 和 `/admin/something` 下都是可见的，但是在 `/home` 或 `/adminpage` 下不可见。

通常，我们应该将 `path` 设置为根目录： `path=/`，以使 cookie 对此网站的所有页面可见。

## domain

- **domain=site.com**

可访问 cookie 的域。但是在实际中，有一些限制。我们无法设置任何域。

默认情况下，cookie 只有在设置的域下才能被访问到。所以，如果 cookie 设置在 site.com 下，我们在 other.com 下就无法获取它。

.....但是棘手的是，我们在子域 forum.site.com 下也无法获取它！

```
// 在 site.com
document.cookie = "user=John"

// 在 forum.site.com
alert(document.cookie); // 没有 user
```

无法使 cookie 可以被从另一个二级域访问，因此，other.com 将永远不会收到设置在 site.com 的 cookie。

这是一项安全限制，为了允许我们可以将敏感信息保存在 cookie 中。

.....但是，如果我们想要批准像 forum.site.com 这样的子域访问 cookie，这是可以做到的。当我们设置一个在 site.com 的 cookie 时，我们应该将 domain 选项显式地设置为根域：domain=site.com：

```
// 在 site.com
// 使 cookie 可以被在任何子域 *.site.com 访问:
document.cookie = "user=John; domain=site.com"

// 之后

// 在 forum.site.com
alert(document.cookie); // 有 cookie user=John
```

出于历史原因，domain=.site.com（site.com 前面有一个点符号）也以相同的方式工作，允许从子域访问 cookie。这是一个旧的表示法，如果我们需要支持非常旧的浏览器，则应该使用它。

所以，domain 选项允许设置一个可以在子域访问的 cookie。

## expires, max-age

默认情况下，如果一个 cookie 没有设置这两个参数中的任何一个，那么在关闭浏览器之后，它就会消失。此类 cookie 被称为 "session cookie"。

为了让 cookie 在浏览器关闭后仍然存在，我们可以设置 `expires` 或 `max-age` 选项中的一个。

- **`expires=Tue, 19 Jan 2038 03:14:07 GMT`**

cookie 的到期日期，那时浏览器会自动删除它。

日期必须完全采用 `GMT` 时区的这种格式。我们可以使用 `date.toUTCString` 来获取它。例如，我们可以将 cookie 设置为 1 天后过期。

```
// 当前时间 +1 天
let date = new Date(Date.now() + 86400e3);
date = date.toUTCString();
document.cookie = "user=John; expires=" + date;
```

如果我们将 `expires` 设置为过去的时间，则 cookie 会被删除。

- **`max-age=3600`**

`expires` 的替代选项，具指明 cookie 的过期时间距离当前时间的秒数。

如果为 0 或负数，则 cookie 会被删除：

```
// cookie 会在一小时后失效
document.cookie = "user=John; max-age=3600";

// 删除 cookie (让它立即过期)
document.cookie = "user=John; max-age=0";
```

## secure

- **`secure`**

Cookie 应只能被通过 HTTPS 传输。

默认情况下，如果我们在 `http://site.com` 上设置了 cookie，那么该 cookie 也会出现在 `https://site.com` 上，反之亦然。

也就是说，cookie 是基于域的，它们不区分协议。

使用此选项，如果一个 cookie 是通过 `https://site.com` 设置的，那么它不会在相同域的 `HTTP` 环境下出现，例如 `http://site.com`。所以，如果一个 cookie 包含绝不应该通过未加密的 `HTTP` 协议发送的敏感内容，那么就应该设置这个选项。

```
// 假设我们现在在 HTTPS 环境下
// 设置 cookie secure (只在 HTTPS 环境下可访问)
```

```
document.cookie = "user=John; secure";
```

## samesite

这是另外一个关于安全的特性。它旨在防止 **XSRF**（跨网站请求伪造）攻击。

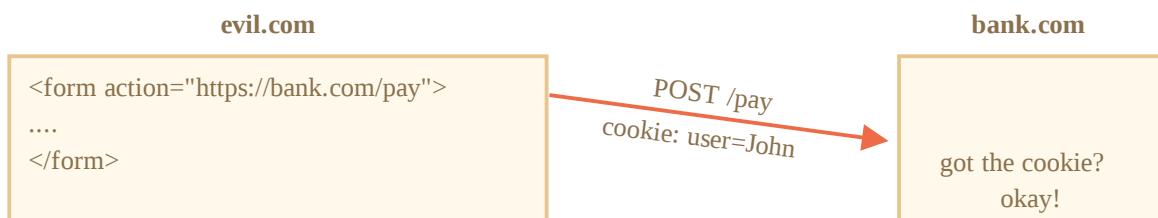
为了了解它是如何工作的，以及何时有用，让我们看一下 **XSRF** 攻击。

### XSRF 攻击

想象一下，你登录了 `bank.com` 网站。此时：你有了来自该网站的身份验证 `cookie`。你的浏览器会在每次请求时将其发送到 `bank.com`，以便识别你，并执行所有敏感的财务上的操作。

现在，在另外一个窗口中浏览网页时，你不小心访问了另一个网站 `evil.com`。该网站具有向 `bank.com` 网站提交一个具有启动与黑客账户交易的字段的表单 `<form action="https://bank.com/pay">` 的 JavaScript 代码。

你每次访问 `bank.com` 时，浏览器都会发送 `cookie`，即使该表单是从 `evil.com` 提交过来的。因此，银行会识别你的身份，并执行真实的付款。



这就是“跨网站请求伪造（Cross-Site Request Forgery，简称 XSRF）”攻击。

当然，实际的银行会防止出现这种情况。所有由 `bank.com` 生成的表单都具有一个特殊的字段，即所谓的“**XSRF 保护 token**”，恶意页面既不能生成，也不能从远程页面提取它（它可以在那里提交表单，但是无法获取数据）。并且，网站 `bank.com` 会对收到的每个表单都进行这种 `token` 的检查。

但是，实现这种防护需要花费时间：我们需要确保每个表单都具有 `token` 字段，并且还必须检查所有请求。

### 输入 `cookie samesite` 选项

`Cookie` 的 `samesite` 选项提供了另一种防止此类攻击的方式，（理论上）不需要要求“**XSRF 保护 token**”。

它有两个可能的值：

- `samesite=strict`（和没有值的 `samesite` 一样）

如果用户来自同一网站之外，那么设置了 `samesite=strict` 的 `cookie` 永远不会被发送。

换句话说，无论用户是通过邮件链接还是从 `evil.com` 提交表单，或者进行了任何来自其他域下的操作，`cookie` 都不会被发送。

如果身份验证 `cookie` 具有 `samesite` 选项，那么 `XSRF` 攻击是没有机会成功的，因为来自 `evil.com` 的提交没有 `cookie`。因此，`bank.com` 将无法识别用户，也就不会继续进行付款。

这种保护是相当可靠的。只有来自 `bank.com` 的操作才会发送 `samesite` `cookie`，例如来自 `bank.com` 的另一页面的表单提交。

虽然，这样有一些不方便。

当用户通过合法的链接访问 `bank.com` 时，例如从他们自己的笔记，他们会感到惊讶，`bank.com` 无法识别他们的身份。实际上，在这种情况下不会发送 `samesite=strict` `cookie`。

我们可以通过使用两个 `cookie` 来解决这个问题：一个 `cookie` 用于“一般识别”，仅用于说“Hello, John”，另一个带有 `samesite=strict` 的 `cookie` 用于进行数据更改的操作。这样，从网站外部来的用户会看到欢迎信息，但是支付操作必须是从银行网站启动的，这样第二个 `cookie` 才能被发送。

- `samesite=lax`

一种更轻松的方法，该方法还可以防止 `XSRF` 攻击，并且不会破坏用户体验。

宽松（`lax`）模式，和 `strict` 模式类似，当从外部来到网站，则禁止浏览器发送 `cookie`，但是增加了一个例外。

如果以下两个条件均成立，则会发送 `samesite=lax` `cookie`：

1. `HTTP` 方法是“安全的”（例如 `GET` 方法，而不是 `POST`）。

所有安全的 `HTTP` 方法详见 [RFC7231 规范](#)。基本上，这些都是用于读取而不是写入数据的方法。它们不得执行任何更改数据的操作。跟随链接始终是 `GET`，是安全的方法。

2. 该操作执行顶级导航（更改浏览器地址栏中的 `URL`）。

这通常是成立的，但是如果导航是在一个 `<iframe>` 中执行的，那么它就不是顶级的。此外，用于网络请求的 `JavaScript` 方法不会执行任何导航，因此它们不适合。

所以，`samesite=lax` 所做的是基本上允许最常见的“去往 `URL`”操作具有 `cookie`。例如，从笔记本中打开网站链接就满足这些条件。

但是，任何更复杂的事儿，例如来自另一网站的网络请求或表单提交都会丢失 `cookie`。

如果这种情况适合你，那么添加 `samesite=lax` 将不会破坏用户体验并且可以增加保护。

总体而言，`samesite` 是一个很好的选项，但是它有一个重要的缺点：

- `samesite` 会被到 2017 年左右的旧版本浏览器忽略（不兼容）。

因此，如果我们仅依靠 `samesite` 提供保护，那么在旧版本的浏览器上将很容易受到攻击。

但是，我们肯定可以将 `samesite` 与其他保护措施（例如 `XSRF token`）一起使用，例如 `xsrf token`，这样可以多增加一层保护，将来，当旧版本的浏览器淘汰时，我们可能就可以删除 `xsrf token` 这种方式了。

## httpOnly

这个选项和 `JavaScript` 没有关系，但是我们必须为了完整性也提一下它。

Web 服务器使用 `Set-Cookie` header 来设置 cookie。并且，它可以设置 `httpOnly` 选项。

这个选项禁止任何 `JavaScript` 访问 cookie。我们使用 `document.cookie` 看不到此类 cookie，也无法对此类 cookie 进行操作。

这是一种预防措施，当黑客将自己的 `JavaScript` 代码注入网页，并等待用户访问该页面时发起攻击，而这个选项可以防止此时的这种攻击。这应该是不可能发生的，黑客应该无法将他们的代码注入我们的网站，但是网站有可能存在 `bug`，使得黑客能够实现这样的操作。

通常来说，如果发生了这种情况，并且用户访问了带有黑客 `JavaScript` 代码的页面，黑客代码将执行并通过 `document.cookie` 获取到包含用户身份验证信息的 cookie。这就很糟糕了。

但是，如果 cookie 设置了 `httpOnly`，那么 `document.cookie` 则看不到 cookie，所以它受到了保护。

## 附录: Cookie 函数

这里有一组有关 cookie 操作的函数，比手动修改 `document.cookie` 方便得多。

有很多这种 cookie 库，所以这些函数只用于演示。虽然它们都能正常使用。

### getCookie(name)

获取 cookie 最简短的方式是使用 [正则表达式](#)。

`getCookie(name)` 函数返回具有给定 `name` 的 cookie:

```
// 返回具有给定 name 的 cookie,
// 如果没找到，则返回 undefined
function getCookie(name) {
 let matches = document.cookie.match(new RegExp(
 "(?:^|;)"+ name.replace(/([\$.?*|\{\}\(\)\[\]\\\\\/\^\+])/g, '\\$1') + "=([^\;]*")"
```

```
));
return matches ? decodeURIComponent(matches[1]) : undefined;
}
```

这里的 `new RegExp` 是动态生成的，以匹配 `; name=<value>`。

请注意 `cookie` 的值是经过编码的，所以 `getCookie` 使用了内建方法 `decodeURIComponent` 函数对其进行解码。

### **setCookie(name, value, options)**

将 `cookie name` 设置为具有默认值 `path=/`（可以修改以添加其他默认值）和给定值 `value`：

```
function setCookie(name, value, options = {}) {

 options = {
 path: '/',
 // 如果需要，可以在这里添加其他默认值
 ...options
 };

 if (options.expires instanceof Date) {
 options.expires = options.expires.toUTCString();
 }

 let updatedCookie = encodeURIComponent(name) + "=" + encodeURIComponent(value);

 for (let optionKey in options) {
 updatedCookie += "; " + optionKey;
 let optionValue = options[optionKey];
 if (optionValue !== true) {
 updatedCookie += "=" + optionValue;
 }
 }

 document.cookie = updatedCookie;
}

// 使用范例:
setCookie('user', 'John', {secure: true, 'max-age': 3600});
```

### **deleteCookie(name)**

要删除一个 `cookie`，我们可以给它设置一个负的过期时间来调用它：

```
function deleteCookie(name) {
 setCookie(name, "", {
 'max-age': -1
 })
```

```
 })
}
```

### ⚠ 更新或删除必须使用相同的路径和域

请注意：当我们更新或删除一个 cookie 时，我们应该使用和设置 cookie 时相同的路径和域选项。

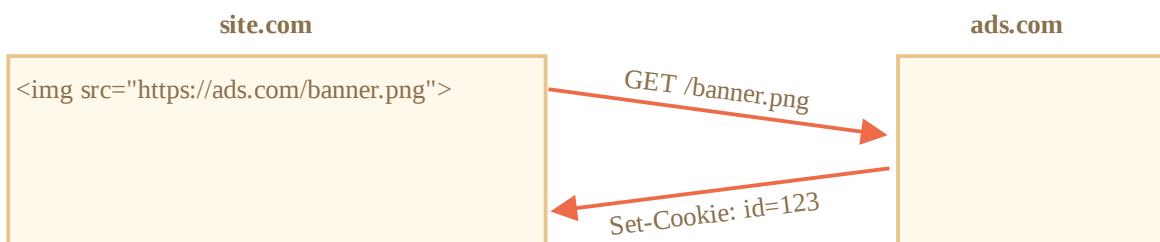
代码放在：[cookie.js](#)。

## 附录：第三方 cookie

如果 cookie 是由用户所访问的页面的域以外的域放置的，则称其为第三方 cookie。

例如：

1. `site.com` 网站的一个页面加载了另外一个网站的 banner: ``。
2. 与 banner 一起，`ads.com` 的远程服务器可能会设置带有 `id=1234` 这样的 cookie 的 `Set-Cookie` header。此类 cookie 源自 `ads.com` 域，并且仅在 `ads.com` 中可见：



3. 下次访问 `ads.com` 网站时，远程服务器获取 cookie `id` 并识别用户：



4. 更为重要的是，当用户从 `site.com` 网站跳转至另一个也带有 banner 的网站 `other.com` 时，`ads.com` 会获得该 cookie，因为它属于 `ads.com`，从而识别

用户并在他在网站之间切换时对其进行跟踪：



由于它的性质，第三方 `cookie` 通常用于跟踪和广告服务。它们被绑定在原始域上，因此 `ads.com` 可以在不同网站之间跟踪同一用户，如果这些网站都可以访问 `ads.com` 的话。

当然，有些人不喜欢被跟踪，因此浏览器允许禁止此类 `cookie`。

此外，一些现代浏览器对此类 `cookie` 采取特殊策略：

- Safari 浏览器完全不允许第三方 `cookie`。
- Firefox 浏览器附带了一个第三方域的黑名单，它阻止了来自名单内的域的第三方 `cookie`。

#### ❶ 请注意：

如果我们加载了一个来自第三方域的脚本，例如 `<script src="https://google-analytics.com/analytics.js">`，并且该脚本使用 `document.cookie` 设置了 `cookie`，那么此类 `cookie` 就不是第三方的。

如果一个脚本设置了一个 `cookie`，那么无论脚本来自何处——这个 `cookie` 都属于当前网页的域。

## 附录: GDPR

本主题和 JavaScript 无关，只是设置 `cookie` 时的一些注意事项。

欧洲有一项名为 **GDPR** 的立法，该法规针对网站尊重用户实施了一系列规则。其中之一就是需要明确的许可才可以跟踪用户的 `cookie`。

请注意，这仅与跟踪/识别/授权 `cookie` 有关。

所以，如果我们设置一个只保存了一些信息的 `cookie`，但是既不跟踪也不识别用户，那么我们可以自由地设置它。

但是，如果我们要设置带有身份验证会话（**session**）或跟踪 `id` 的 `cookie`，那么必须得到用户的允许。

网站为了遵循 **GDPR** 通常有两种做法。你一定已经在网站中看到过它们了：

## 1. 如果一个网站想要仅为已经经过身份验证的用户设置跟踪的 cookie。

为此，注册表单中必须要有一个复选框，例如“接受隐私政策”（描述怎么使用 cookie），用户必须勾选它，然后网站就可以自由设置身份验证 cookie 了。

## 2. 如果一个网站想要为所有人设置跟踪的 cookie。

为了合法地这样做，网站为每个新用户显示一个模态“初始屏幕”，并要求他们同意设置 cookie。之后网站就可以设置 cookie，并可以让用户看到网站内容了。不过，这可能会使新用户感到反感。没有人喜欢看到“必须点击”的模态初始屏幕而不是网站内容。但是 GDPR 要求必须得到用户明确地准许。

GDPR 不仅涉及 cookie，还涉及其他与隐私相关的问题，但这超出了我们的讨论范围。

## 总结

`document.cookie` 提供了对 cookie 的访问

- 写入操作只会修改其中提到的 cookie。
- name/value 必须被编码。
- 一个 cookie 最大为 4kb，每个网站最多有 20+ 个左右的 cookie（具体取决于浏览器）。

Cookie 选项：

- `path=/`，默认为当前路径，使 cookie 仅在该路径下可见。
- `domain=site.com`，默认 cookie 仅在当前域下可见，如果显式设置了域，可以使 cookie 在子域下也可见。
- `expires` 或 `max-age` 设置 cookie 过期时间，如果没有设置，则当浏览器关闭时 cookie 就失效了。
- `secure` 使 cookie 仅在 HTTPS 下有效。
- `samesite`，如果请求来自外部网站，禁止浏览器发送 cookie，这有助于防止 XSRF 攻击。

另外：

- 浏览器可能会禁用第三方 cookie，例如 Safari 浏览器默认禁止所有第三方 cookie。
- 在为欧盟公民设置跟踪 cookie 时，GDPR 要求必须得到用户明确许可。

## LocalStorage, sessionStorage

Web 存储对象 `localStorage` 和 `sessionStorage` 允许我们在浏览器上保存键/值对。

它们有趣的是，在页面刷新后（对于 `sessionStorage`）甚至浏览器完全重启（对于 `localStorage`）后，数据仍然保留在浏览器中。我们很快就会看到。

我们已经有了 `cookie`。为什么还要其他存储对象呢？

- 与 `cookie` 不同，Web 存储对象不会随每个请求被发送到服务器。因此，我们可以保存更多数据。大多数浏览器都允许保存至少 2MB 的数据（或更多），并且具有用于配置数据的设置。
- 还有一点和 `cookie` 不同，服务器无法通过 `HTTP header` 操纵存储对象。一切都是在 `JavaScript` 中完成的。
- 存储绑定到源（域/协议/端口三者）。也就是说，不同协议或子域对应不同的存储对象，它们之间无法访问彼此数据。

两个存储对象都提供相同的方法和属性：

- `setItem(key, value)` —— 存储键/值对。
- `getItem(key)` —— 按照键获取值。
- `removeItem(key)` —— 删除键及其对应的值。
- `clear()` —— 删除所有数据。
- `key(index)` —— 获取该索引下的键名。
- `length` —— 存储的内容的长度。

正如你所看到的，它就像一个 `Map` 集合（`setItem/getItem/removeItem`），但也允许通过 `key(index)` 来按索引访问。

让我们看看它是如何工作的吧。

## localStorage 示例

`localStorage` 最主要的特点是：

- 在同源的所有标签页和窗口之间共享数据。
- 数据不会过期。它在浏览器重启甚至系统重启后仍然存在。

例如，如果你运行此代码……

```
localStorage.setItem('test', 1);
```

……然后关闭/重新打开浏览器，或者只是在不同的窗口打开同一页面，然后你可以这样获取它：

```
alert(localStorage.getItem('test')); // 1
```

我们只需要在同一个源（域/端口/协议），URL 路径可以不同。

在所有同源的窗口之间，`localStorage` 数据可以共享。因此，如果我们在一个窗口中设置了数据，则在另一个窗口中也可以看到数据变化。

## 类对象形式访问

我们还可以像使用一个普通对象那样，读取/设置键，像这样：

```
// 设置 key
localStorage.test = 2;

// 获取 key
alert(localStorage.test); // 2

// 删除 key
delete localStorage.test;
```

这是历史原因造成的，并且大多数情况下都可行，但通常不建议这样做，因为：

1. 如果键是由用户生成的，那么它可以是任何内容，例如 `length` 或 `toString`，也可以是 `localStorage` 的另一种内建方法。在这种情况下，`getItem/setItem` 可以正常工作，而类对象访问的方式则会失败：

```
let key = 'length';
localStorage[key] = 5; // Error, 无法对 length 进行赋值
```

2. 有一个 `storage` 事件，在我们更改数据时会触发。但以类对象方式访问时，不会触发该事件。我们将在本章的后面看到。

## 遍历键

正如我们所看到的，这些方法提供了“按照键获取/设置/删除”的功能。但我们如何获取所有保存的值或键呢？

不幸的是，存储对象是不可迭代的。

一种方法是像遍历数组那样遍历它们：

```
for(let i = 0; i < localStorage.length; i++) {
 let key = localStorage.key(i);
 alert(`#${key}: ${localStorage.getItem(key)}`);
}
```

另一个方式是使用 `for key in localStorage` 循环，就像处理常规对象一样。它会遍历所有的键，但也会输出一些我们不需要的内建字段。

```
// 不好的尝试
for(let key in localStorage) {
 alert(key); // 显示 getItem, setItem 和其他内建的东西
}
```

.....因此，我们需要使用 `hasOwnProperty` 检查来过滤掉原型中的字段：

```
for(let key in localStorage) {
 if (!localStorage.hasOwnProperty(key)) {
 continue; // 跳过像 "setItem", "getItem" 等这样的键
 }
 alert(`#${key}: ${localStorage.getItem(key)}`);
}
```

.....或者，使用 `Object.keys` 获取只属于“自己”的键，然后如果需要，可以遍历它们：

```
let keys = Object.keys(localStorage);
for(let key of keys) {
 alert(`#${key}: ${localStorage.getItem(key)}`);
}
```

后者有效，因为 `Object.keys` 只返回属于对象的键，会忽略原型上的。

## 仅字符串

请注意，键和值都必须是字符串。

如果是任何其他类型，例数字或对象，它会被自动转换为字符串。

```
sessionStorage.user = {name: "John"};
alert(sessionStorage.user); // [object Object]
```

我们可以使用 `JSON` 来存储对象：

```
sessionStorage.user = JSON.stringify({name: "John"});
// sometime later
```

```
let user = JSON.parse(sessionStorage.user);
alert(user.name); // John
```

也可以对整个存储对象进行字符串化处理，例如出于调试目的：

```
// 为 JSON.stringify 增加了格式设置选项，以使对象看起来更美观
alert(JSON.stringify(localStorage, null, 2));
```

## sessionStorage

`sessionStorage` 对象的使用频率比 `localStorage` 对象低得多。

属性和方法是相同的，但是它有更多的限制：

- `sessionStorage` 的数据只存在于当前浏览器标签页。
  - 具有相同页面的另一个标签页中将会有不同的存储。
  - 但是，它在同一标签页下的 `iframe` 之间是共享的（假如它们来自相同的源）。
- 数据在页面刷新后仍然保留，但在关闭/重新打开浏览器标签页后不会被保留。

让我们看看它的运行效果。

运行此代码.....

```
sessionStorage.setItem('test', 1);
```

.....然后刷新页面。这时你仍然可以获取到数据：

```
alert(sessionStorage.getItem('test')); // after refresh: 1
```

.....但是，如果你在另一个新的标签页中打开此页面，然后在新页面中再次运行上面这行代码，则会得到 `null`，表示“未找到数据”。

这是因为 `sessionStorage` 不仅绑定到源，还绑定在同一浏览器标签页。因此，`sessionStorage` 很少被使用。

## Storage 事件

当 `localStorage` 或 `sessionStorage` 中的数据更新后，`storage` 事件就会触发，它具有以下属性：

- `key` —— 发生更改的数据的 `key`（如果调用的是 `.clear()` 方法，则为 `null`）。

- `oldValue` —— 旧值（如果是新增数据，则为 `null`）。
- `newValue` —— 新值（如果是删除数据，则为 `null`）。
- `url` —— 发生数据更新的文档的 `url`。
- `storageArea` —— 发生数据更新的 `localStorage` 或 `sessionStorage` 对象。

重要的是：该事件会在所有可访问到存储对象的 `window` 对象上触发，导致当前数据改变的 `window` 对象除外。

我们来详细解释一下。

想象一下，你有两个窗口，它们具有相同的页面。所以 `localStorage` 在它们之间是共享的。

如果两个窗口都在监听 `window.onstorage` 事件，那么每个窗口都会对另一个窗口中发生的更新作出反应。

```
// 在其他文档对同一存储进行更新时触发
window.onstorage = event => { // 等同于 window.addEventListener('storage', () => {
 if (event.key != 'now') return;
 alert(event.key + ':' + event.newValue + " at " + event.url);
};

localStorage.setItem('now', Date.now());
```

请注意，该事件还包含：`event.url` —— 发生数据更新的文档的 `url`。

并且，`event.storageArea` 包含存储对象 —— `sessionStorage` 和 `localStorage` 具有相同的事件，所以 `event.storageArea` 引用了被修改的对象。我们可能会想设置一些东西，以“响应”更改。

**这允许同源的不同窗口交换消息。**

现代浏览器还支持 [Broadcast channel API ↗](#)，这是用于同源窗口之间通信的特殊 API，它的功能更全，但被支持的情况不好。有一些库基于 `localStorage` 来 polyfill 该 API，使其可以用在任何地方。

## 总结

Web 存储对象 `localStorage` 和 `sessionStorage` 允许我们在浏览器中保存键/值对。

- `key` 和 `value` 都必须为字符串。
- 存储大小限制为 2MB+，具体取决于浏览器。
- 它们不会过期。
- 数据绑定到源（域/端口/协议）。

localStorage	sessionStorage
在同源的所有标签页和窗口之间共享数据	在当前浏览器标签页中可见，包括同源的 iframe
浏览器重启后数据仍然保留	页面刷新后数据仍然保留（但标签页关闭后数据则不再保留）

API:

- `setItem(key, value)` —— 存储键/值对。
- `getItem(key)` —— 按照键获取值。
- `removeItem(key)` —— 删除键及其对应的值。
- `clear()` —— 删除所有数据。
- `key(index)` —— 获取该索引下的键名。
- `length` —— 存储的内容的长度。
- 使用 `Object.keys` 来获取所有的键。
- 我们将键作为对象属性来访问，在这种情况下，不会触发 `storage` 事件。

Storage 事件:

- 在调用 `setItem`, `removeItem`, `clear` 方法后触发。
- 包含有关操作的所有数据 (`key/oldValue/newValue`)，文档 `url` 和存储对象 `storageArea`。
- 在所有可访问到存储对象的 `window` 对象上触发，导致当前数据改变的 `window` 对象除外（对于 `sessionStorage` 是在当前标签页下，对于 `localStorage` 是在全局，即所有同源的窗口）。

## IndexedDB

IndexedDB 是一个内置的数据库，它比 `localStorage` 强大得多。

- 键/值 储存：值（几乎）可以是任何类型，键有多种类型。
- 支撑事务的可靠性。
- 支持键范围查询、索引。
- 和 `localStorage` 相比，它可以存储更多数据。

对于传统的 客户端-服务器 应用，这些功能通常是没有必要的。IndexedDB 适用于离线应用，可与 ServiceWorkers 和其他技术相结合使用。

根据规范 <https://www.w3.org/TR/IndexedDB> 中的描述，IndexedDB 的本机接口是基于事件的。

我们还可以在基于 promise 的包装器（wrapper），如 <https://github.com/jakearchibald/idb> 的帮助下使用 `async/await`。这要方便的

多，但是包装器并不完美，它并不能替代所有情况下的事件。因此，我们先练习事件（events），在理解 IndexedDB 之后，我们将使用包装器。

## 打开数据库

要想使用 IndexedDB，首先需要打开一个数据库。

语法：

```
let openRequest = indexedDB.open(name, version);
```

- `name` —— 字符串，即数据库名称。
- `version` —— 一个正整数版本，默认为 1（下面解释）。

数据库可以有许多不同的名称，但是必须存在于当前的源（域/协议/端口）中。不同的网站不能相互访问对方的数据库。

调用之后，需要监听 `openRequest` 对象上的事件：

- `success`：数据库准备就绪，`openRequest.result` 中有了一个数据库对象“Database Object”，使用它进行进一步的调用。
- `error`：打开失败。
- `upgradeneeded`：数据库已准备就绪，但其版本已过时（见下文）。

IndexedDB 具有内建的“模式（scheme）版本控制”机制，这在服务器端数据库中是不存在的。

与服务器端数据库不同，IndexedDB 存在于客户端，数据存储在浏览器中。因此开发人员不能直接访问它。但当新版本的应用程序发布之后，我们可能需要更新数据库。

如果本地数据库版本低于 `open` 中指定的版本，会触发一个特殊事件 `upgradeneeded`。我们可以根据需要比较版本并升级数据结构。

当数据库还不存在的时候，也会触发这个事件。因此，我们应该先执行初始化。

当我们第一次发布应用程序时，使用版本 1 打开它，并在 `upgradeneeded` 处理程序中执行初始化：

```
let openRequest = indexedDB.open("store", 1);

openRequest.onupgradeneeded = function() {
 // 如果客户端没有数据库则触发
 // ...执行初始化...
};

openRequest.onerror = function() {
```

```
 console.error("Error", openRequest.error);
};

openRequest.onsuccess = function() {
 let db = openRequest.result;
 // 继续使用 db 对象处理数据库
};
```

当我们发布第二个版本时：

```
let openRequest = indexedDB.open("store", 2);

openRequest.onupgradeneeded = function() {
 // 现有的数据库版本小于 2 (或不存在)
 let db = openRequest.result;

 switch(db.version) { // 现有的 db 版本
 case 0:
 // 版本 0 表示客户端没有数据库
 // 执行初始化
 case 1:
 // 客户端版本为 1
 // 更新
 }
};
```

因此，需要在 `openRequest.onupgradeneeded` 中更新数据库，很快我们就能知道运行结果。只有当程序处理完且不报错，才会触发 `openRequest.onsuccess`。

在 `openRequest.onsuccess` 之后，`openRequest.result` 中有一个数据库对象，将用于我们的进一步操作。

删除数据库：

```
let deleteRequest = indexedDB.deleteDatabase(name)
// deleteRequest.onsuccess/onerror 追踪 (tracks) 结果
```

## ⚠ 我们可以打开旧版本吗?

如果我们想打开一个比当前版本更低的数据库，该怎么办？例如，现有的数据库版本是 3，但我想打开版本 2 `open(..., 2)`。

报错，触发 `openRequest.onerror`。

当用户加载了旧代码（例如，代理缓存），可能会发生这种情况。这时我们应该检查 `db.version`，并建议用户重新加载页面。重新检查缓存标头，以确保用户永远不会获取旧代码。

## 并行更新问题

提到版本控制，有一个相关的小问题。

一个用户在网页中打开了数据库为版本 1 的网站。

这时网站更新到版本 2，这个用户在另一网页下打开了网站。这时两个网页都是我们的网站，但一个与数据库版本 1 有开放连接，而另一个试图在 `upgradeneeded` 处理程序中更新。

问题是，这两个网页是同一个站点，同一个来源，共享同一个数据库。而数据库不能同时为版本 1 和版本 2。要执行版本 2 的更新，必须关闭版本 1 的所有连接。

为了完成这些，当尝试并行更新时，`versionchange` 事件会触发一个打开的数据对象。我们应该监听这个对象，关闭数据库（还应该建议访问者重新加载页面，获取最新的代码）。

如果旧连接不关闭，新连接会被 `blocked` 事件阻塞，而不是 `success`。

下面是执行此操作的代码：

```
let openRequest = indexedDB.open("store", 2);

openRequest.onupgradeneeded = ...;
openRequest.onerror = ...;

openRequest.onsuccess = function() {
 let db = openRequest.result;

 db.onversionchange = function() {
 db.close();
 // 数据库已过时，请重新加载页面
 alert("Database is outdated, please reload the page.");
 };
}

//数据库已经准备好，请使用它.....
};

openRequest.onblocked = function() {
 // 到同一数据库的另一个开放连接
}
```

```
// 触发 db.onversionchange 后没有关闭
};
```

在这我们做两件事：

1. 成功打开后添加 `db.onversionchange` 监听器，以得到尝试并行更新的消息。
2. 添加 `openRequest.onblocked` 监听器来处理旧连接未关闭的情况。如果在 `db.onversionchange` 中关闭，就不会发生这种情况。

还有其他方案。例如，我们可以在 `db.onversionchange` 中优雅地关闭一些东西，关闭连接之前提示用户保存数据。如果 `db.onversionchange` 完成但没有关闭，新的连接将立即阻塞。可以要求用户只保留新的网页，关闭旧网页，以此更新数据。

这种更新冲突很少发生，但我们至少应该处理一下。例如使用 `onblocked` 处理程序，以防程序卡死影响用户体验。

## 对象库（object store）

要在 `IndexedDB` 中存储某些内容，我们需要一个**对象库**。

对象库是 `IndexedDB` 的核心概念，在其他数据库中对应的对象称为“表”或“集合”。它是储存数据的地方。一个数据库可能有多个存储区：一个用于存储用户数据，另一个用于商品，等等。

尽管被命名为“对象库”，但也可以存储原始类型。

**几乎可以存储任何值，包括复杂的对象。**

`IndexedDB` 使用[标准序列化算法](#) 来克隆和存储对象。类似于 `JSON.stringify`，不过功能更加强大，能够存储更多的数据类型。

有一种对象不能被存储：循环引用的对象。此类对象不可序列化，也不能进行 `JSON.stringify`。

### 库中的每个值都必须有唯一的键 key

键的类型必须为数字、日期、字符串、二进制或数组。它是唯一的标识符：通过键来搜索/删除/更新 值。

## Database



类似于 `localStorage`，我们向存储区添加值时，可以提供一个键。但当我们存储对象时，`IndexedDB` 允许设置一个对象属性作为键，这就更加方便了。或者，我们可以自动生成键。

但我们需要先创建一个对象库。

创建对象库的语法：

```
db.createObjectStore(name[, keyOptions]);
```

请注意，操作是同步的，不需要 `await`。

- `name` 是存储区名称，例如 "`books`" 表示书。
- `keyOptions` 是具有以下两个属性之一的可选对象：
  - `keyPath` —— 对象属性的路径，`IndexedDB` 将以此路径作为键，例如 `id`。
  - `autoIncrement` —— 如果为 `true`，则自动生成新存储的对象的键，键是一个不断递增的数字。

如果我们不提供 `keyOptions`，那么以后需要在存储对象时，显式地提供一个键。

例如，此对象库使用 `id` 属性作为键：

```
db.createObjectStore('books', {keyPath: 'id'});
```

在 `upgradeneeded` 处理程序中，只有在创建数据库版本时，对象库才能被 创建/修改。

这是技术上的限制。在 `upgradeneedHandler` 之外，可以 添加/删除/更新数据，但是只能在版本更新期间 创建/删除/更改对象库。

要执行数据库版本升级，主要有两种方法：

1. 我们实现每个版本的升级功能：从 1 到 2，从 2 到 3，从 3 到 4，等等。在 `upgradeneeded` 中，可以进行版本比较（例如，老版本是 2，需要升级到 4），并针对每个中间版本（2 到 3，然后 3 到 4）逐步运行每个版本的升级。
2. 或者我们可以检查数据库：以 `db.objectStoreNames` 的形式获取现有对象库的列表。该对象是一个 [DOMStringList](#) 提供 `contains(name)` 方法来检查 `name` 是否存在，再根据存在和不存在的内容进行更新。

对于小型数据库，第二种方法可能更简单。

下面是第二种方法的演示：

```
let openRequest = indexedDB.open("db", 2);

// 创建/升级 数据库而无需版本检查
openRequest.onupgradeneeded = function() {
 let db = openRequest.result;
 if (!db.objectStoreNames.contains('books')) { // 如果没有 "books" 数据
 db.createObjectStore('books', {keyPath: 'id'}); // 创建它
 }
};
```

删除对象库：

```
db.deleteObjectStore('books')
```

## 事务

术语“事务”是通用的，许多数据库中都有用到。

事务是一组操作，要么全部成功，要么全部失败。

例如，当一个人买东西时，我们需要：

1. 从他们的账户中扣除这笔钱。
2. 将该项目添加到他们的清单中。

如果完成了第一个操作，但是出了问题，比如停电。这时无法完成第二个操作，这非常糟糕。两件时应该要么都成功（购买完成，好！）或同时失败（这个人保留了钱，可以重新尝试）。

事务可以保证同时完成。

**所有数据操作都必须在 IndexedDB 中的事务内进行。**

启动事务：

```
db.transaction(store[, type]);
```

- `store` 是事务要访问的库名称，例如 `"books"`。如果我们要访问多个库，则是库名称的数组。
- `type` – 事务类型，以下类型之一：
  - `readonly` —— 只读，默认值。
  - `readwrite` —— 只能读取和写入数据，而不能 创建/删除/更改 对象库。

还有 `versionchange` 事务类型：这种事务可以做任何事情，但不能被手动创建。`IndexedDB` 在打开数据库时，会自动为 `updateneeded` 处理程序创建 `versionchange` 事务。这就是它为什么可以更新数据库结构、创建/删除 对象库的原因。

### i 为什么存在不同类型的事务？

性能是事务需要标记为 只读 (`readonly`) 和 读写 (`readwrite`) 的原因。

许多只读事务能够同时访问同一存储区，但读写事务不能。因为读写事务会“锁定”存储区进行写操作。下一个事务必须等待前一个事务完成，才能访问相同的存储区。

创建事务后，我们可以将项目添加到库，就像这样：

```
let transaction = db.transaction("books", "readwrite"); // (1)

// 获取对象库进行操作
let books = transaction.objectStore("books"); // (2)

let book = {
 id: 'js',
 price: 10,
 created: new Date()
};

let request = books.add(book); // (3)

request.onsuccess = function() { // (4)
 // 书已添加到存储区
 console.log("Book added to the store", request.result);
};

request.onerror = function() {
 console.log("Error", request.error);
};
```

基本有四个步骤:

1. 创建一个事务，在（1）表明要访问的所有存储。
2. 使用 `transaction.objectStore(name)`，在（2）中获取存储对象。
3. 在（3）执行对对象库 `books.add(book)` 的请求。
4. .....处理请求成功/错误（4），还可以根据需要发出其他请求。

对象库支持两种存储值的方法:

- **put(value, [key])** 将 `value` 添加到存储区。仅当对象库没有 `keyPath` 或 `autoIncrement` 时，才提供 `key`。如果已经存在具有相同键的值，则将替换该值。
- **add(value, [key])** 与 `put` 相同，但是如果已经有一个值具有相同的键，则请求失败，并生成一个名为 "`ConstraintError`" 的错误。

与打开数据库类似，我们可以发送一个请求: `books.add(book)`，然后等待 `success/error` 事件。

- `add` 的 `request.result` 是新对象的键。
- 错误在 `request.error` (如果有的话) 中。

## 事务的自动提交

在上面的示例中，我们启动了事务并发出了 `add` 请求。但正如前面提到的，一个事务可能有多个相关的请求，这些请求必须全部成功或全部失败。那么如何标记事务为已完成，并不再请求呢？

简短的回答是：没有。

在下一个版本 3.0 规范中，可能会有一种手动方式来完成事务，但目前在 2.0 中还没有。

**当所有事务的请求完成，并且 微任务队列 为空时，它将自动提交。**

通常，我们可以假设事务在其所有请求完成时提交，并且当前代码完成。

因此，在上面的示例中，不需要任何特殊调用即可完成事务。

事务自动提交原则有一个重要的副作用。不能在事务中间插入 `fetch`, `setTimeout` 等异步操作。`IndexedDB` 不会让事务等待这些操作完成。

在下面的代码中，`request2` 中的行 (\*) 失败，因为事务已经提交，不能在其中发出任何请求:

```
let request1 = books.add(book);

request1.onsuccess = function() {
```

```
fetch('/').then(response => {
 let request2 = books.add(anotherBook); // (*)
 request2.onerror = function() {
 console.log(request2.error.name); // TransactionInactiveError
 };
});
});
```

这是因为 `fetch` 是一个异步操作，一个宏任务。事务在浏览器开始执行宏任务之前关闭。

`IndexedDB` 规范的作者认为事务应该是短期的。主要是性能原因。

值得注意的是，`readwrite` 事务将存储“锁定”以进行写入。因此，如果应用程序的一部分启动了 `books` 对象库上的 `readwrite` 操作，那么希望执行相同操作的另一部分必须等待新事务“挂起”，直到第一个事务完成。如果事务处理需要很长时间，将会导致奇怪的延迟。

那么，该怎么办？

在上面的示例中，我们可以在新请求 (\*) 之前创建一个新的 `db.transaction`。

如果需要在一个事务中把所有操作保持一致，更好的做法是将 `IndexedDB` 事务和“其他”异步内容分开。

首先，执行 `fetch`，并根据需要准备数据。然后创建事务并执行所有数据库请求，然后就正常了。

为了检测到成功完成的时刻，我们可以监听 `transaction.oncomplete` 事件：

```
let transaction = db.transaction("books", "readwrite");

//执行操作......

transaction.oncomplete = function() {
 console.log("Transaction is complete"); // 事务执行完成
};
```

只有 `complete` 才能保证将事务作为一个整体保存。个别请求可能会成功，但最终的写入操作可能会出错（例如 I/O 错误或其他错误）。

要手动中止事务，请调用：

```
transaction.abort();
```

取消请求里所做的所有修改，并触发 `transaction.onabort` 事件。

## 错误处理

写入请求可能会失败。

这是意料之中的事，不仅是我们可能会犯的粗心失误，还有与事务本身相关的其他原因。例如超过了存储配额。因此，必须做好请求失败的处理。

**失败的请求将自动中止事务，并取消所有的更改。**

在一些情况下，我们会想自己去处理失败事务（例如尝试另一个请求）并让它继续执行，而不是取消现有的更改。可以调用 `request.onerror` 处理程序，在其中调用 `event.preventDefault()` 防止事务中止。

在下面的示例中，添加了一本新书，键 (`id`) 与现有的书相同。`store.add` 方法生成一个 `"ConstraintError"`。可以在不取消事务的情况下进行处理：

```
let transaction = db.transaction("books", "readwrite");

let book = { id: 'js', price: 10 };

let request = transaction.objectStore("books").add(book);

request.onerror = function(event) {
 // 有相同 id 的对象存在时，发生 ConstraintError
 if (request.error.name == "ConstraintError") {
 console.log("Book with such id already exists"); // 处理错误
 event.preventDefault(); // 不要中止事务
 // 这个 book 用另一个键？
 } else {
 // 意外错误，无法处理
 // 事务将中止
 }
};

transaction.onabort = function() {
 console.log("Error", transaction.error);
};
```

## 事件委托

每个请求都需要调用 `onerror/onsuccess`？并不，可以使用事件委托来代替。

**IndexedDB 事件冒泡：请求 → 事务 → 数据库。**

所有事件都是 DOM 事件，有捕获和冒泡，但通常只使用冒泡阶段。

因此，出于报告或其他原因，我们可以使用 `db.onerror` 处理程序捕获所有错误：

```
db.onerror = function(event) {
 let request = event.target; // 导致错误的请求
```

```
 console.log("Error", request.error);
};
```

.....但是错误被完全处理了呢？这种情况不应该被报告。

我们可以通过在 `request.onerror` 中使用 `event.stopPropagation()` 来停止冒泡，从而停止 `db.onerror` 事件。

```
request.onerror = function(event) {
 if (request.error.name == "ConstraintError") {
 console.log("Book with such id already exists"); // 处理错误
 event.preventDefault(); // 不要中止事务
 event.stopPropagation(); // 不要让错误冒泡，停止它的传播
 } else {
 // 什么都不做
 // 事务将中止
 // 我们可以解决 transaction.onabort 中的错误
 }
};
```

## 通过键搜索

对象库有两种主要的搜索类型：

1. 通过一个键或一个键范围。即：通过在“books”中存储的 `book.id`。
2. 另一个对象字段，例如 `book.price`。

首先，让我们来处理键和键范围（1）。

涉及到的搜索方法，包括支持精确键，也包括所谓的“范围查询”——[IDBKeyRange ↗](#) 对象指定一个“键范围”。

使用以下调用函数创建范围：

- `IDBKeyRange.lowerBound(lower, [open])` 表示：`≥lower`（如果 `open` 是 `true`，表示 `>lower`）
- `IDBKeyRange.upperBound(upper, [open])` 表示：`≤upper`（如果 `open` 是 `true`，表示 `<upper`）
- `IDBKeyRange.bound(lower, upper, [lowerOpen], [upperOpen])` 表示：在 `lower` 和 `upper` 之间。如果 `open` 为 `true`，则相应的键不包括在范围内。
- `IDBKeyRange.only(key)` —— 仅包含一个键的范围 `key`，很少使用。

所有搜索方法都接受一个查询参数 `query`，该参数可以是精确键或者键范围：

- `store.get(query)` —— 按键或范围搜索第一个值。

- `store.getAll([query], [count])` —— 搜索所有值。如果 `count` 给定，则按 `count` 进行限制。
- `store.getKey(query)` —— 搜索满足查询的第一个键，通常是一个范围。
- `store.getAllKeys([query], [count])` —— 搜索满足查询的所有键，通常是一个范围。如果 `count` 给定，则最多为 `count`。
- `store.count([query])` —— 获取满足查询的键的总数，通常是一个范围。

例如，我们存储区里有很多书。因为 `id` 字段是键，因此所有方法都可以按 `id` 进行搜索。

请求示例：

```
// 获取一本书
books.get('js')

// 获取 'css' <= id <= 'html' 的书
books.getAll(IDBKeyRange.bound('css', 'html'))

// 获取 id < 'html' 的书
books.getAll(IDBKeyRange.upperBound('html', true))

// 获取所有书
books.getAll()

// 获取所有 id > 'js' 的键
books.getAllKeys(IDBKeyRange.lowerBound('js', true))
```

### ❶ 对象库始终是有序的

对象库按键对值进行内部排序。

因此，请求的返回值，是按照键的顺序排列的。

## 通过带索引的字段搜索

要根据其他对象字段进行搜索，我们需要创建一个名为“索引（index）”的附加数据结构。

索引是存储的“附加项”，用于跟踪给定的对象字段。对于该字段的每个值，它存储有该值的对象的键列表。下面会有更详细的图片。

语法：

```
objectStore.createIndex(name, keyPath, [options]);
```

- **name** —— 索引名称。
- **keyPath** —— 索引应该跟踪的对象字段的路径（我们将根据该字段进行搜索）。
- **option** —— 具有以下属性的可选对象：
  - **unique** —— 如果为true，则存储中只有一个对象在 **keyPath** 上具有给定值。如果我们尝试添加重复项，索引将生成错误。
  - **multiEntry** —— 只有 **keyPath** 上的值是数组时使用。这时，默认情况下，索引将默认把整个数组视为键。但是如果 **multiEntry** 为 true，那么索引将为该数组中的每个值保留一个存储对象的列表。所以数组成员成为了索引键。

在我们的示例中，是按照 **id** 键存储图书的。

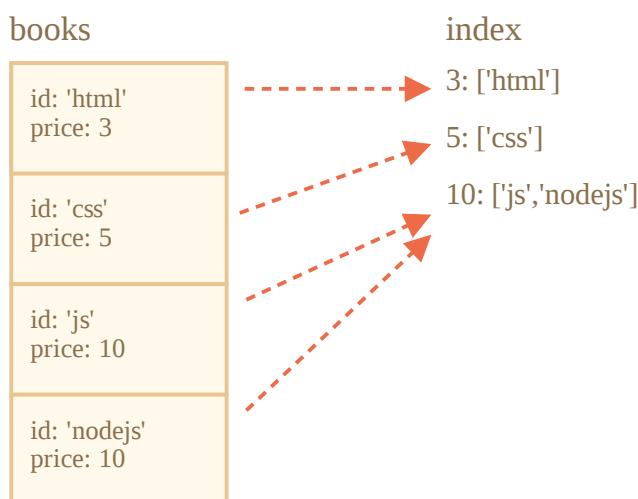
假设我们想通过 **price** 进行搜索。

首先，我们需要创建一个索引。它像对象库一样，必须在 **upgradeneeded** 中创建完成：

```
openRequest.onupgradeneeded = function() {
 // 在 versionchange 事务中，我们必须在这里创建索引
 let books = db.createObjectStore('books', {keyPath: 'id'});
 let index = inventory.createIndex('price_idx', 'price');
};
```

- 该索引将跟踪 **price** 字段。
- 价格不是唯一的，可能有多本书价格相同，所以我们不设置唯一 **unique** 选项。
- 价格不是一个数组，因此不适用多入口 **multiEntry** 标志。

假设我们的库存里有4本书。下面的图片显示了该索引 **index** 的确切内容：



如上所述，每个 **price** 值的索引（第二个参数）保存具有该价格的键的列表。索引自动保持最新，所以我们不必关心它。

现在，当我们想要搜索给定的价格时，只需将相同的搜索方法应用于索引：

```
let transaction = db.transaction("books"); // 只读
let books = transaction.objectStore("books");
let priceIndex = books.index("price_idx");

let request = priceIndex.getAll(10);

request.onsuccess = function() {
 if (request.result !== undefined) {
 console.log("Books", request.result); // 价格为 10 的书的数组
 } else {
 console.log("No such books");
 }
};
```

我们还可以使用 `IDBKeyRange` 创建范围，并查找便宜/贵的书：

```
// 查找价格 <=5 的书籍
let request = priceIndex.getAll(IDBKeyRange.upperBound(5));
```

在我们的例子中，索引是按照被跟踪对象字段价格 `price` 进行内部排序的。所以当我们进行搜索时，搜索结果也会按照价格排序。

## 从存储中删除

`delete` 方法查找要由查询删除的值，调用格式类似于 `getAll`

- `delete(query)` —— 通过查询删除匹配的值。

例如：

```
// 删除 id='js' 的书
books.delete('js');
```

如果要基于价格或其他对象字段删除书。首先需要在索引中找到键，然后调用 `delete`：

```
// 找到价格 = 5 的钥匙
let request = priceIndex.getKey(5);

request.onsuccess = function() {
 let id = request.result;
```

```
let deleteRequest = books.delete(id);
};
```

删除所有内容:

```
books.clear(); // 清除存储。
```

## 光标 (Cursors)

像 `getAll/getAllKeys` 这样的方法，会返回一个 键/值 数组。

但是一个对象库可能很大，比可用的内存还大。这时，`getAll` 就无法将所有记录作为一个数组获取。

该怎么办呢？

光标提供了解决这一问题的方法。

光标是一种特殊的对象，它在给定查询的情况下遍历对象库，一次返回一个键/值，从而节省内存。

由于对象库是按键在内部排序的，因此光标按键顺序（默认为升序）遍历存储。

语法:

```
// 类似于 getAll，但带有光标:
let request = store.openCursor(query, [direction]);

// 获取键，而不是值（例如 getAllKeys）: store.openKeyCursor
```

- **query** 是一个键或键范围，与 `getAll` 相同。
- **direction** 是一个可选参数，使用顺序是：
  - `"next"` —— 默认值，光标从有最小索引的记录向上移动。
  - `"prev"` —— 相反的顺序：从有最大的索引的记录开始下降。
  - `"nextunique"`，`"prevunique"` —— 同上，但是跳过键相同的记录（仅适用于索引上的光标，例如，对于价格为 5 的书，仅返回第一本）。

光标对象的主要区别在于 `request.onSuccess` 多次触发：每个结果触发一次。

这有一个如何使用光标的例子：

```
let transaction = db.transaction("books");
let books = transaction.objectStore("books");
```

```
let request = books.openCursor();

// 为光标找到的每本书调用
request.onsuccess = function() {
 let cursor = request.result;
 if (cursor) {
 let key = cursor.key; // 书的键 (id字段)
 let value = cursor.value; // 书本对象
 console.log(key, value);
 cursor.continue();
 } else {
 console.log("No more books");
 }
};
```

主要的光标方法有：

- `advance(count)` —— 将光标向前移动 `count` 次，跳过值。
- `continue([key])` —— 将光标移至匹配范围中的下一个值（如果给定键，紧接键之后）。

无论是否有更多的值匹配光标 —— 调用 `onsuccess`。结果中，我们可以获得指向下一条记录的光标，或者 `undefined`。

在上面的示例中，光标是为对象库创建的。

也可以在索引上创建一个光标。索引是允许按对象字段进行搜索的。在索引上的光标与在对象存储上的光标完全相同 — 它们通过一次返回一个值来节省内存。

对于索引上的游标，`cursor.key` 是索引键（例如：价格），我们应该使用 `cursor.primaryKey` 属性作为对象的键：

```
let request = priceIdx.openCursor(IDBKeyRange.upperBound(5));

// 为每条记录调用
request.onsuccess = function() {
 let cursor = request.result;
 if (cursor) {
 let key = cursor.primaryKey; // 下一个对象存储键 (id 字段)
 let value = cursor.value; // 下一个对象存储对象 (book 对象)
 let key = cursor.key; // 下一个索引键 (price)
 console.log(key, value);
 cursor.continue();
 } else {
 console.log("No more books"); // 没有书了
 }
};
```

## Promise 包装器

将 `onsuccess/onerror` 添加到每个请求是一项相当麻烦的任务。我们可以通过使用事件委托（例如，在整个事务上设置处理程序）来简化我们的工作，但是 `async/await` 要方便的多。

在本章，我们会进一步使用一个轻便的承诺包装器

<https://github.com/jakearchibald/idb>。它使用 `promisified` IndexedDB 方法创建全局 `idb` 对象。

然后，我们可以不使用 `onsuccess/onerror`，而是这样写：

```
let db = await idb.openDb('store', 1, db => {
 if (db.oldVersion == 0) {
 // 执行初始化
 db.createObjectStore('books', {keyPath: 'id'});
 }
});

let transaction = db.transaction('books', 'readwrite');
let books = transaction.objectStore('books');

try {
 await books.add(...);
 await books.add(...);

 await transaction.complete;

 console.log('jsbook saved');
} catch(err) {
 console.log('error', err.message);
}
```

现在我们有了可爱的“简单异步代码”和「`try...catch`」捕获的东西。

## 错误处理

如果我们没有捕获到错误，那么程序将一直失败，直到外部最近的 `try..catch` 捕获到为止。

未捕获的错误将成为 `window` 对象上的“`unhandled promise rejection`”事件。

我们可以这样处理这种错误：

```
window.addEventListener('unhandledrejection', event => {
 let request = event.target; // IndexedDB 本机请求对象
 let error = event.reason; // 未处理的错误对象，与 request.error 相同
 //报告错误.....
});
```

## “非活跃事务”陷阱

我们都知道，浏览器一旦执行完成当前的代码和**微任务**之后，事务就会自动提交。因此，如果我们在事务中间放置一个类似 `fetch` 的宏任务，事务只是会自动提交，而不会等待它执行完成。因此，下一个请求会失败。

对于 `promise` 包装器和 `async/await`，情况是相同的。

这是在事务中间进行 `fetch` 的示例：

```
let transaction = db.transaction("inventory", "readwrite");
let inventory = transaction.objectStore("inventory");

await inventory.add({ id: 'js', price: 10, created: new Date() });

await fetch(...); // (*)

await inventory.add({ id: 'js', price: 10, created: new Date() }); // 错误
```

`fetch (*)` 后的下一个 `inventory.add` 失败，出现“非活动事务”错误，因为这时事务已经被提交并且关闭了。

解决方法与使用本机 IndexedDB 时相同：进行新事务，或者将事情分开。

1. 准备数据，先获取所有需要的信息。
2. 然后保存在数据库中。

## 获取本机对象

在内部，包装器执行本机 IndexedDB 请求，并添加 `onerror/onsuccess` 方法，并返回 `rejects/resolves` 结果的 `promise`。

在大多数情况下都可以运行，示例在这 <https://github.com/jakearchibald/idb>。

极少数情况下，我们需要原始的 `request` 对象。可以将 `promise` 的 `promise.request` 属性，当作原始对象进行访问：

```
let promise = books.add(book); // 获取 promise 对象(不要 await 结果)

let request = promise.request; // 本地请求对象
let transaction = request.transaction; // 本地事务对象

//做些本地的 IndexedDB 的处理......

let result = await promise; // 如果仍然需要
```

## 总结

IndexedDB 可以被认为是“localStorage on steroids”。这是一个简单的键值对数据库，功能强大到足以支持离线应用，而且用起来比较简单。

最好的指南是官方文档。目前的版本 [是2.0](#)，但是[3.0](#) 版本中的一些方法（差别不大）也得到部分支持。

基本用法可以用几个短语来描述：

1. 获取一个 promise 包装器，比如 `idb`。
2. 打开一个数据库： `idb.openDb(name, version, onupgradeneeded)`
  - 在 `onupgradeneeded` 处理程序中创建对象存储和索引，或者根据需要执行版本更新。
3. 对于请求：
  - 创建事务 `db.transaction('books')`（如果需要的话，设置 `readwrite`）。
  - 获取对象存储 `transaction.objectStore('books')`。
4. 按键搜索，可以直接调用对象库上的方法。
  - 要按对象字段搜索，需要创建索引。
5. 如果内存中容纳不下数据，请使用光标。

这里有一个小应用程序示例：

<https://plnkr.co/edit/q43Zgtwr54BGYnYK?p=preview>

## 动画

CSS 和 JavaScript 动画。

## 贝塞尔曲线

贝塞尔曲线用于计算机图形绘制形状，CSS 动画和许多其他地方。

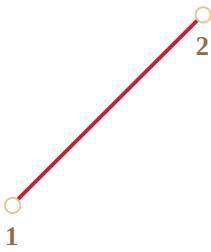
他们其实非常简单，值得学习一次并且在矢量图形和高级动画的世界里非常受用。

## 控制点

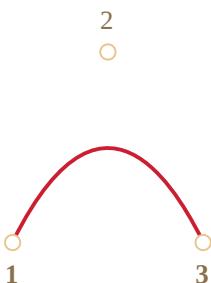
贝塞尔曲线 [由控制点定义](#)。

这些点可能有 2、3、4 或更多。

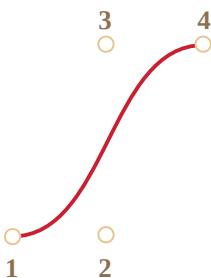
例如，两点曲线：



三点曲线:

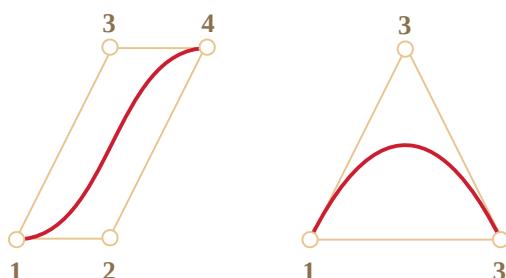


四点曲线:



如果仔细观察这些曲线，你会立即注意到：

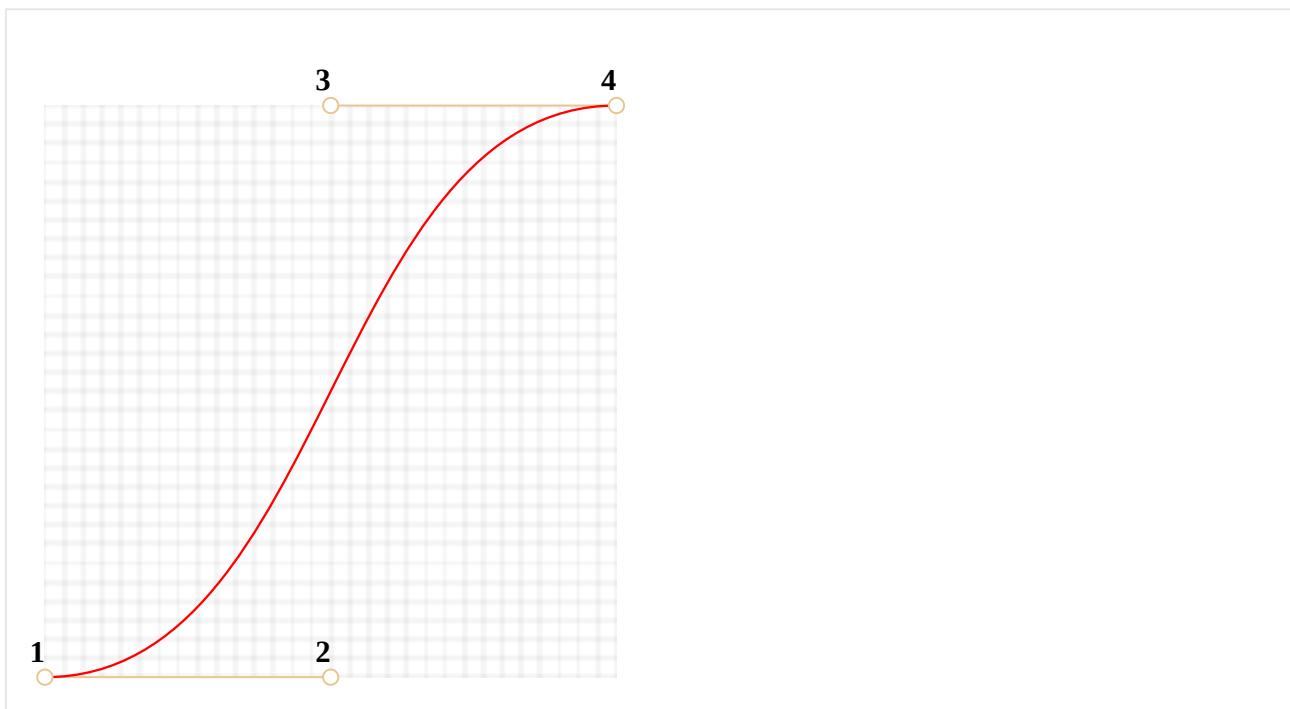
1. **控制点不总是在曲线上**这是非常正常的，稍后我们将看到曲线是如何构建的。
2. **曲线的阶次等于控制点的数量减一**。对于两个点我们能得到一条线性曲线（直线），三个点 — 一条二阶曲线，四个点 — 一条三阶曲线。
3. **曲线总是在控制点的凸包 内部**:



由于最后一个属性，在计算机图形学中，可以优化相交测试。如果凸包不相交，则曲线也不相交。因此，首先检查凸包的交叉点可以非常快地给出“无交叉”结果。检查交叉区域或凸包更容易，因为它们是矩形，三角形等（见上图），比曲线简单的多。

贝塞尔曲线绘制的主要重点——通过移动曲线，曲线以直观明显的方式变化。

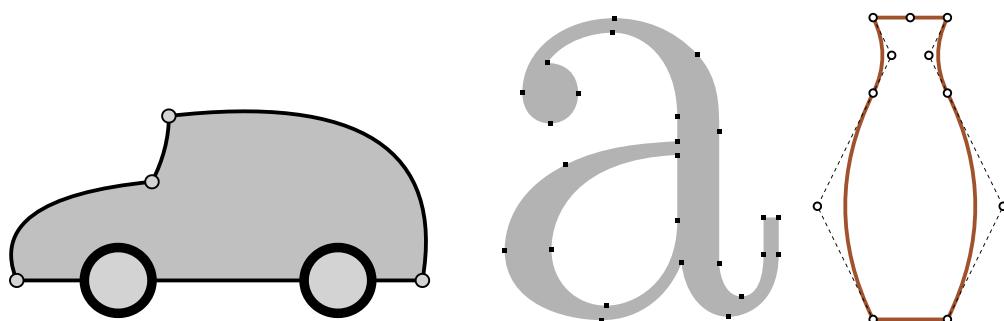
尝试在下面的示例中使用鼠标移动控制点：



可以注意到，曲线沿切线  $1 \rightarrow 2$  和  $3 \rightarrow 4$  延伸。

经过一些练习后，很明显我们知道怎样通过放置控制点来获得所需要的曲线。通过连接几条曲线，我们几乎可以得到任何东西。

这里有一些例子：



## 数学

贝塞尔曲线可以使用数学方程式来描述。

很快我们就能看到——没必要知道它。但是为了完整性——请看这里。

给定控制点  $P_i$  的坐标: 第一个控制点的坐标为  $P_1 = (x_1, y_1)$ , 第二个控制点的坐标为  $P_2 = (x_2, y_2)$ , 以此类推, 曲线坐标由方程式描述, 这个方程式依赖属于区间  $[0, 1]$  的参数  $t$ 。

- 有两个控制点的曲线方程:

$$P = (1-t)P_1 + tP_2$$

- 有三个控制点的曲线方程:

$$P = (1-t)^2P_1 + 2(1-t)tP_2 + t^2P_3$$

- 有四个控制点的曲线方程:

$$P = (1-t)^3P_1 + 3(1-t)^2tP_2 + 3(1-t)t^2P_3 + t^3P_4$$

这些是矢量方程。

我们可以逐坐标重写它们, 例如 3 点曲线:

- $x = (1-t)^2x_1 + 2(1-t)tx_2 + t^2x_3$
- $y = (1-t)^2y_1 + 2(1-t)ty_2 + t^2y_3$

我们应该放置 3 个控制点的坐标, 而不是  $x_1, y_1, x_2, y_2, x_3$  和  $y_3$ 。

例如, 如果控制点是  $(0, 0)$ 、 $(0.5, 1)$  和  $(1, 0)$ , 则方程式为:

- $x = (1-t)^2 * 0 + 2(1-t)t * 0.5 + t^2 * 1 = (1-t)t + t^2 = t$
- $y = (1-t)^2 * 0 + 2(1-t)t * 1 + t^2 * 0 = 2(1-t)t = -t^2 + 2t$

现在随着  $t$  从  $0$  到  $1$  变化, 每个  $t$  对应的  $(x, y)$  集合可以构成曲线。

这可能太学术化了, 对于曲线为什么看起来像这样以及它们如何依赖于控制点的描述并不是很明显。

所以绘制算法可能更容易理解。

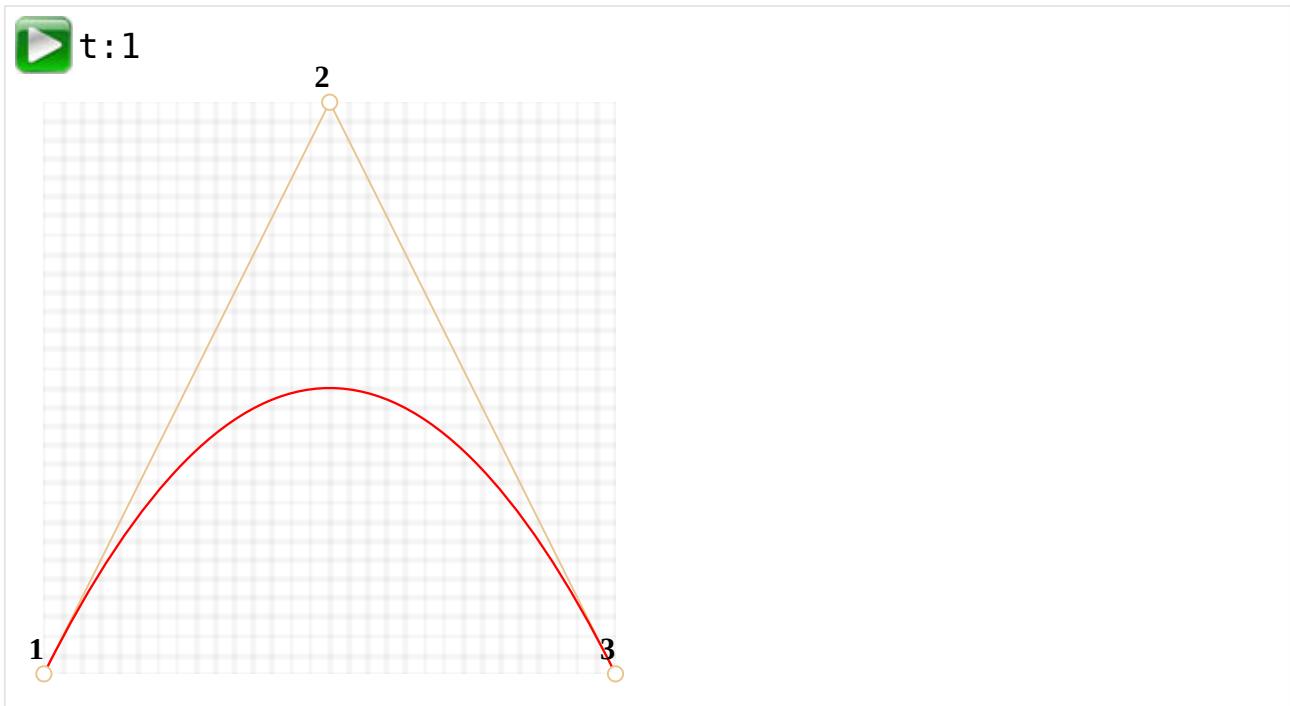
## 德卡斯特里奥算法

德卡斯特里奥算法  与曲线的数学定义相同, 但直观地显示了曲线是如何被建立的。

让我们看看 3 个控制点的例子。

这里是一个演示, 随后会有解释。

控制点可以用鼠标移动, 点击“play”运行演示。



### 德卡斯特里奥算法构造三点贝塞尔曲线：

1. 绘制控制点。在上面的演示中，它们标有： **1**、**2** 和 **3**。
2. 创建控制点 **1 → 2 → 3** 间的线段. 在上面的演示中它们是**棕色的**。
3. 参数 **t** 从 **0 to 1** 变化。在上面的演示中取值 **0.05**: 循环遍历 **0, 0.05, 0.1, 0.15, ... 0.95, 1**。

对于每一个 **t** 的取值:

- 在每一个**棕色**线段上我们取一个点，这个点距起点的距离按比例 **t** 取值。由于有两条线段，我们能得到两个点。

例如，当 **t=0** — 所有点都在线段起点处，当 **t=0.25** — 点到起点的距离为线段长度的 25%，当 **t=0.5** — 50%（中间），当 **t=1** — 线段终点。

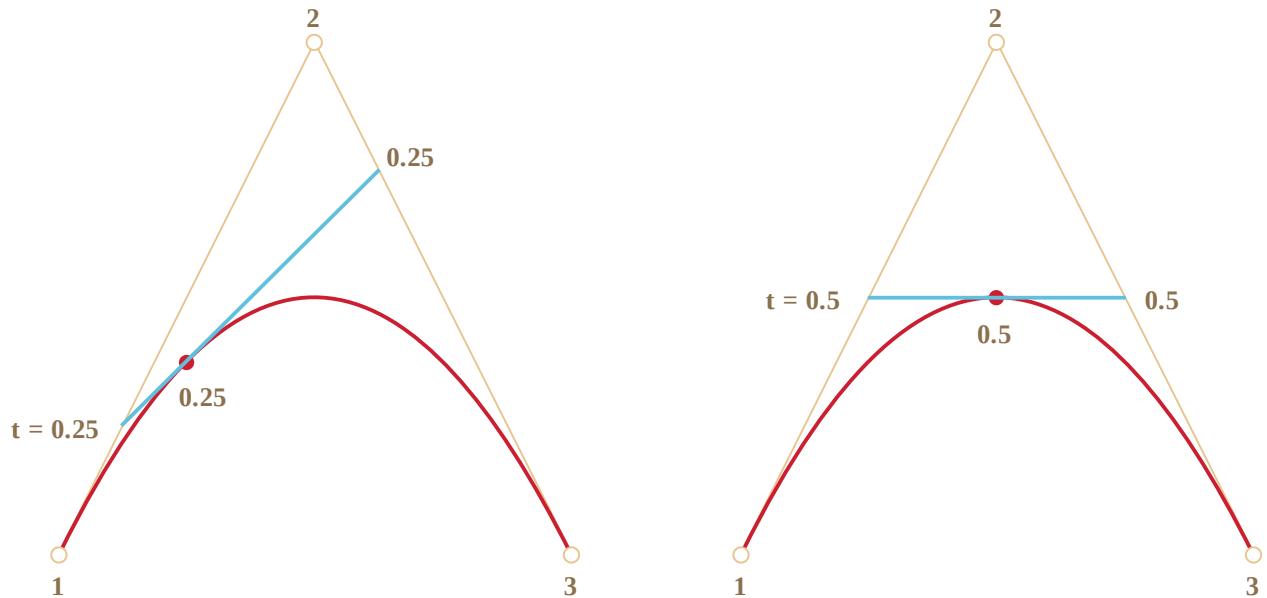
- 连接这些点，下面这张图中连好的线被绘制成**蓝色**。

当 **t=0.25**

当 **t=0.5**

当  $t=0.25$

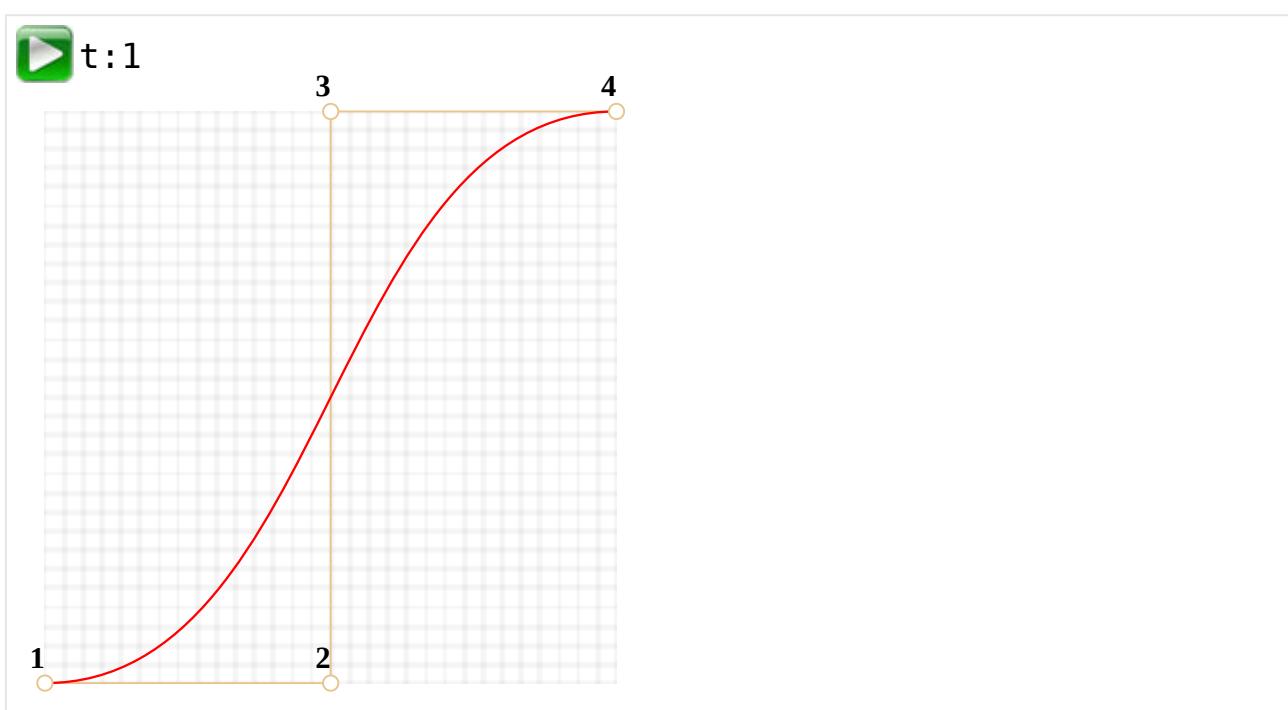
当  $t=0.5$



4. 现在在蓝色线段上取一个点，距离比例取相同数值的  $t$ 。也就是说，当  $t=0.25$  (左图) 时，我们取到的点位于线段的左  $1/4$  终点处，当  $t=0.5$  (右图) 时 — 线段中间。在上图中这一点是红色的。
5. 随着  $t$  从  $0$  to  $1$  变化，每一个  $t$  的值都会添加一个点到曲线上。这些点的集合就形成的贝塞尔曲线。它在上面的图中是红色的，并且是抛物线状的。

这是三控制点的处理过程，但是对于 4 个点同样适用。

4 个控制点的演示（点可以被鼠标移动）：



算法:

- 控制点通过线段连接:  $1 \rightarrow 2$ 、 $2 \rightarrow 3$  和  $3 \rightarrow 4$ 。我们能得到 3 条棕色的线段。
- 对于  $0$  to  $1$  之间的每一个  $t$ :
  - 我们在这些线段上距起点距离比例为  $t$  的位置取点。把这些点连接起来, 然后得到两条绿色线段。
  - 在这些线段上同样按比例  $t$  取点, 得到一条蓝色线段。
  - 在蓝色线段按比例  $t$  取点。在上面的例子中是红色的。
- 这些点在一起组成了曲线。

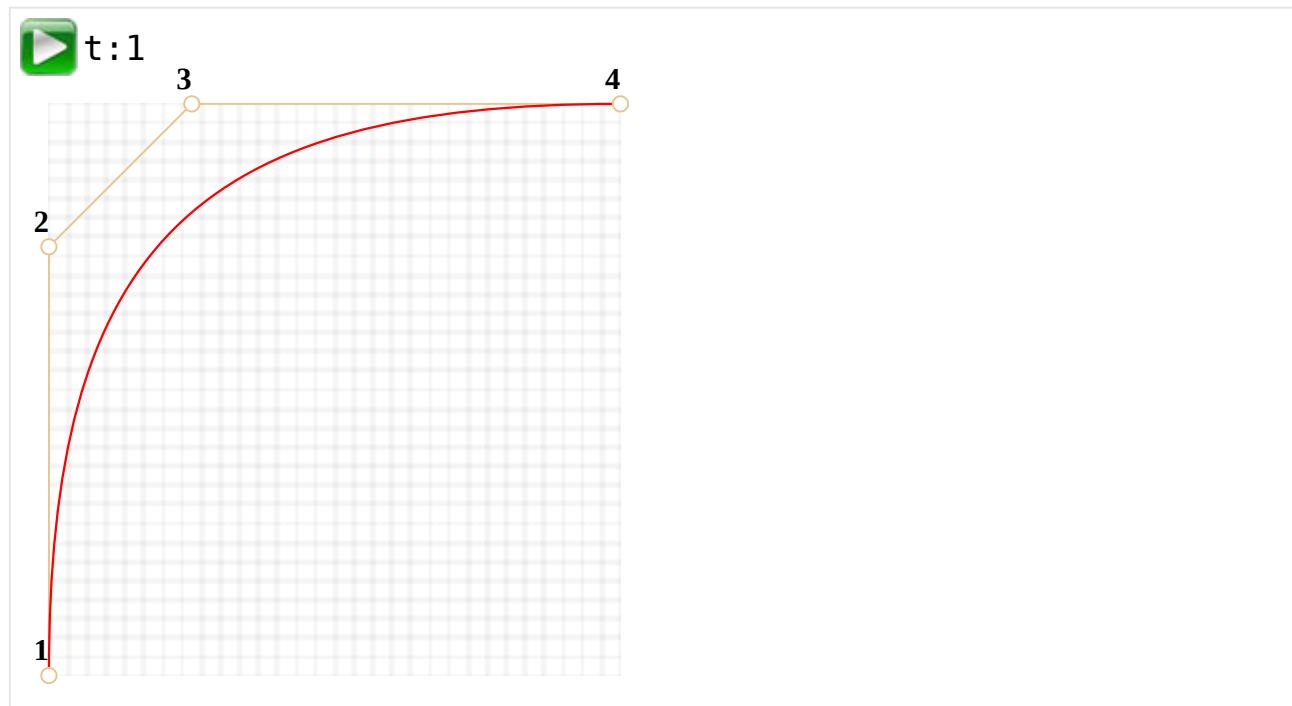
该算法是递归的, 并且可以适应于任意数量的控制点。

给定  $N$  个控制点, 我们将它们连接起来以获得初始的  $N-1$  个线段。

然后对从  $0$  到  $1$  的每一个  $t$ :

- 在每条线段上按  $t$  比例距离取一个点并且连接 —— 会得到  $N-2$  个线段。
- 在上面得到的每条线段上按  $t$  比例距离取一个点并且连接 —— 会得到  $N-3$  个线段, 以此类推.....
- 直到我们得到一个点。得到的这些点就形成了曲线。

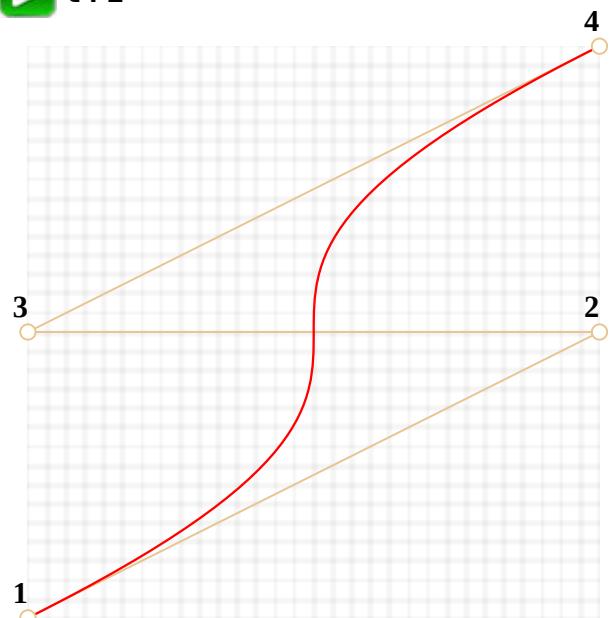
曲线的移动演示:



和其它的点:



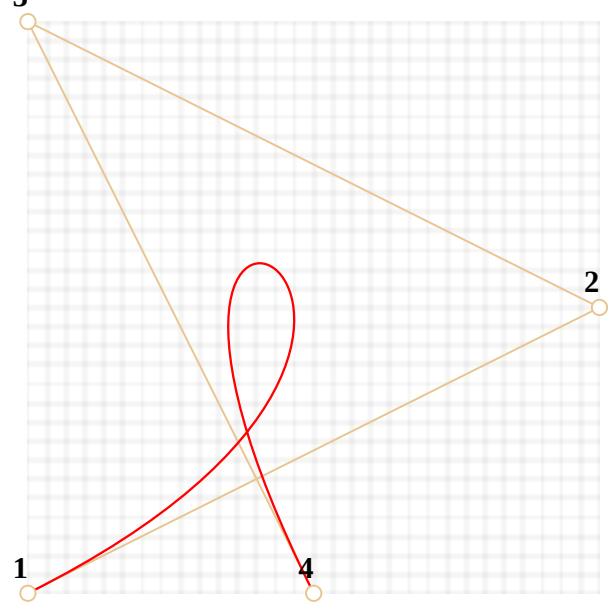
t:1



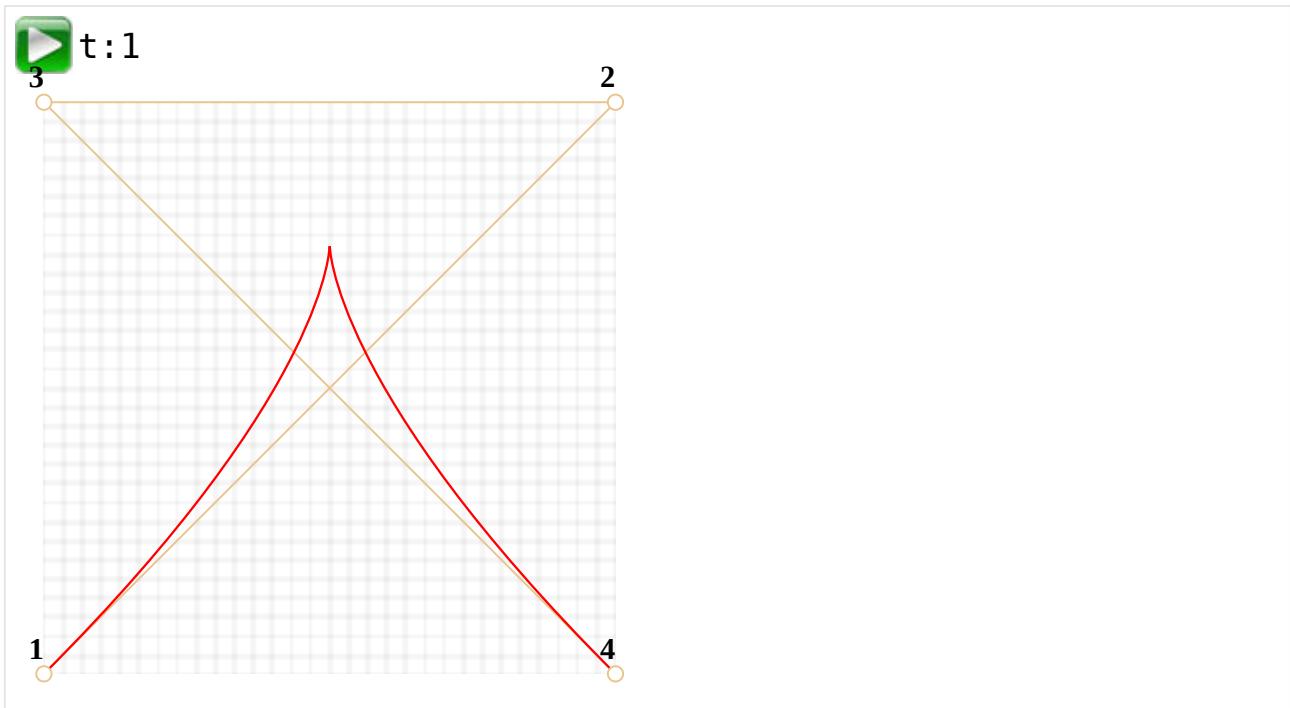
环形:



t:1



非平滑贝塞尔曲线:



由于算法是递归的，我们可以构建任何顺序的贝塞尔曲线：使用 5 个、6 个或更多个控制点。但在实践中它们没那么有用。通常我们取 2-3 个点，对于复杂的线条，将几条曲线拼接在一起。这更容易开发和计算。

### i 如何通过给定点绘制曲线？

我们使用控制点制作贝塞尔曲线。正如我们所见，它们并不在曲线上。或者更准确地说，第一个和最后一个在曲线上，但其它的不在。

有时我们有另一种任务：绘制一条曲线 **通过几个点**，让它们都在一条平滑曲线上。这种任务叫[插值](#)，这里我们不覆盖讲解它。

这些曲线有数学方程式，例如[拉格朗日多项式](#)。

在计算机图形中[样条插值](#)通常用于构建连接多个点的平滑曲线。

## 总结

贝塞尔曲线由其控制点定义。

贝塞尔曲线的两种定义方法：

1. 使用数学方程式。
2. 使用绘图过程：德卡斯特里奥算法

贝塞尔曲线的优点：

- 我们可以通过控制点移动来用鼠标绘制平滑线条。
- 复杂的形状可以由多条贝塞尔曲线组成。

用途:

- 在计算机图形学, 建模, 矢量图形编辑器中。字体由贝塞尔曲线描述。
- 在 Web 开发中 — 用于 Canvas 上的图形和 SVG 格式。顺便说一下, 上面的“实时”示例是用 SVG 编写的。它们实际上是一个 SVG 文档, 被赋予不同的控制点做参数。你可以在单独的窗口中打开它并查源码: [demo.svg](#)。
- 在 CSS 动画中描述动画的路径和速度。

## CSS 动画

CSS 动画可以在不借助 Javascript 的情况下做出一些简单的动画效果。

你也可以通过 Javascript 控制 CSS 动画, 使用少量的代码, 就能让动画表现更加出色。

### CSS 过渡 (transition) [[#css-transition](#)]

CSS 过渡的理念非常简单, 我们只需要定义某一个属性以及如何动态地表现其变化。当属性变化时, 浏览器将会绘制出相应的过渡动画。

也就是说: 我们只需要改变某个属性, 然后所有流畅的动画都由浏览器生成。

举个例子, 以下 CSS 会为 background-color 的变化生成一个 3 秒的过渡动画:

```
.animated {
 transition-property: background-color;
 transition-duration: 3s;
}
```

现在, 只要一个元素拥有名为 `.animated` 的类, 那么任何背景颜色的变化都会被渲染为 3 秒钟的动画。

单击以下按钮以演示动画:

```
<button id="color">Click me</button>

<style>
 #color {
 transition-property: background-color;
 transition-duration: 3s;
 }
</style>

<script>
 color.onclick = function() {
 this.style.backgroundColor = 'red';
 }
</script>
```

```
};
</script>
```

Click me

CSS 提供了四个属性来描述一个过渡:

- `transition-property`
- `transition-duration`
- `transition-timing-function`
- `transition-delay`

之后我们会详细介绍它们，目前我们需要知道，我们可以在 `transition` 中以 `property duration timing-function delay` 的顺序一次性定义它们，并且可以同时为多个属性设置过渡动画。

请看以下例子，点击按钮生成 `color` 和 `font-size` 的过渡动画:

```
<button id="growing">Click me</button>

<style>
#growing {
 transition: font-size 3s, color 2s;
}
</style>

<script>
growing.onclick = function() {
 this.style.fontSize = '36px';
 this.style.color = 'red';
};
</script>
```

Click me

现在让我们一个一个展开看这些属性。

## transition-property

在 `transition-property` 中我们可以列举要设置动画的所有属性，如：`left`、`margin-left`、`height` 和 `color`。

不是所有的 CSS 属性都可以使用过渡动画，但是它们中的[大多数](#)都是可以的。

`all` 表示应用在所有属性上。

## transition-duration

`transition-duration` 允许我们指定动画持续的时间。时间的格式参照[CSS 时间格式](#)：单位为秒 `s` 或者毫秒 `ms`。

## transition-delay

`transition-delay` 允许我们设定动画开始前的延迟时间。例如，对于 `transition-delay: 1s`，动画将会在属性变化发生 1 秒后开始渲染。

你也可以提供一个负值。那么动画将会从整个过渡的中间时刻开始渲染。例如，对于 `transition-duration: 2s`，同时把 `delay` 设置为 `-1s`，那么这个动画将会持续 1 秒钟，并且从正中间开始渲染。

这里演示了数字从 `0` 到 `9` 的动画，使用了 CSS `translate` 方法：

<https://plnkr.co/edit/zBb6B4MI1WYsUJ0g?p=preview>

如下在 `transform` 属性上应用动画：

```
#stripe.animate {
 transform: translate(-90%);
 transition-property: transform;
 transition-duration: 9s;
}
```

在以上的例子中，JavaScript 把 `.animate` 类添加到了元素上，由此触发了动画：

```
stripe.classList.add('animate');
```

我们也可以『从中间』开始，也就是说从某个特定数字开始，比方说，从当前的秒数开始。这就要用到负的 `transition-delay`。

此处，如果你单击这个数字，那么它会从当前的秒数开始渲染：

<https://plnkr.co/edit/NQLS2apfmeMhkJD2?p=preview>

只需添加一行 JavaScript 代码：

```
stripe.onclick = function() {
 let sec = new Date().getSeconds() % 10;
 // for instance, -3s here starts the animation from the 3rd second
```

```
stripe.style.transitionDelay = '-' + sec + 's';
stripe.classList.add('animate');
};
```

## transition-timing-function

时间函数描述了动画进程在时间上的分布。它是先慢后快还是先快后慢？

乍一看，这可能是最复杂的属性了，但是稍微花点时间，你就会发现其实也很简单。这个属性接受两种值：一个贝塞尔曲线（**Bezier curve**）或者阶跃函数（**steps**）。我们先从贝塞尔曲线开始，这也是较为常用的。

### 贝塞尔曲线（**Bezier curve**）

时间函数可以用**贝塞尔曲线**描述，通过设置四个满足以下条件的控制点：

1. 第一个应为：`(0, 0)`。
2. 最后一个应为：`(1, 1)`。
3. 对于中间值，`x` 必须位于 `0..1` 之间，`y` 可以为任意值。

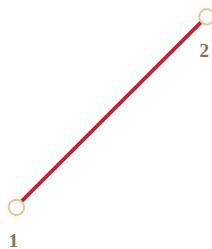
CSS 中设置一贝塞尔曲线的语法为：`cubic-bezier(x2, y2, x3, y3)`。这里我们只需要设置第二个和第三个值，因为第一个点固定为 `(0, 0)`，第四个点固定为 `(1, 1)`。

时间函数描述了动画进行的快慢。

- `x` 轴表示时间：`0` —— 开始时刻，`1` —— `transition-duration` 的结束时刻。
- `y` 轴表示过程的完成度：`0` —— 属性的起始值，`1` —— 属性的最终值。

最简单的一种情况就是动画匀速进行，可以通过设置曲线为 `cubic-bezier(0, 0, 1, 1)` 来实现。

看上去就像这样：



...正如我们所见，这就是条直线。随着时间 `x` 推移，完成度 `y` 稳步从 `0` 增长到 `1`。

例子中的列车匀速地从左侧移动到右侧：

[https://plnkr.co/edit/NZdbZoW6BjFfCRgf?p=preview ↗](https://plnkr.co/edit/NZdbZoW6BjFfCRgf?p=preview)

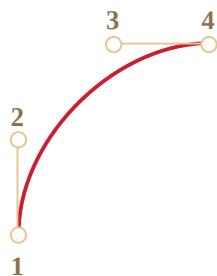
这个里面的 CSS 就是基于刚才那条曲线的：

```
.train {
 left: 0;
 transition: left 5s cubic-bezier(0, 0, 1, 1);
 /* JavaScript sets left to 450px */
}
```

...那么，我们如果表现出减速行驶的列车呢？

我们可以使用另一条贝塞尔曲线：`cubic-bezier(0.0, 0.5, 0.5, 1.0)`。

图像如下：



正如我们所见，这个过程起初很快：曲线开始迅速升高，然后越来越慢。

这是实际的效果演示：

[https://plnkr.co/edit/trgYHdLvWpuYGiXe?p=preview ↗](https://plnkr.co/edit/trgYHdLvWpuYGiXe?p=preview)

CSS:

```
.train {
 left: 0;
 transition: left 5s cubic-bezier(0, .5, .5, 1);
 /* JavaScript sets left to 450px */
}
```

CSS 提供几条内置的曲线：`linear`、`ease`、`ease-in`、`ease-out` 和 `ease-in-out`。

`linear` 其实就是 `cubic-bezier(0, 0, 1, 1)` 的简写——一条直线，刚刚我们已经看过了。

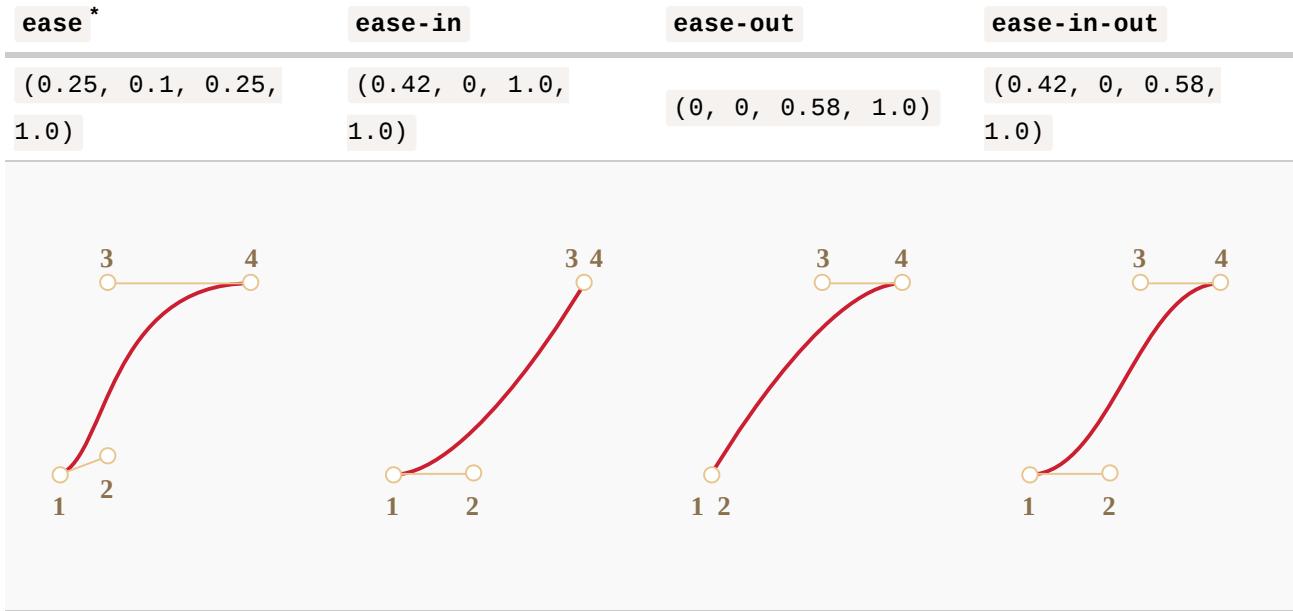
其它的名称是以下贝塞尔曲线的简写：

`ease`\*

`ease-in`

`ease-out`

`ease-in-out`



\* —— 默认值，如果没有指定时间函数，那么将使用 `ease` 作为默认值。

所以，我们可以使用 `ease-out` 来表现减速行驶的列车：

```
.train {
 left: 0;
 transition: left 5s ease-out;
 /* transition: left 5s cubic-bezier(0, .5, .5, 1); */
}
```

但是这看起来有点怪怪的。

**贝塞尔曲线可以使动画『超出』其原本的范围。**

曲线上的控制点的 `y` 值可以使任意的：不管是负值还是一个很大的值。如此，贝塞尔曲线就会变得很低或者很高，让动画超出其正常的范围。

在一下的例子中使用的代码：

```
.train {
 left: 100px;
 transition: left 5s cubic-bezier(.5, -1, .5, 2);
 /* JavaScript sets left to 400px */
}
```

`left` 本该在 `100px` 到 `400px` 之间变化。

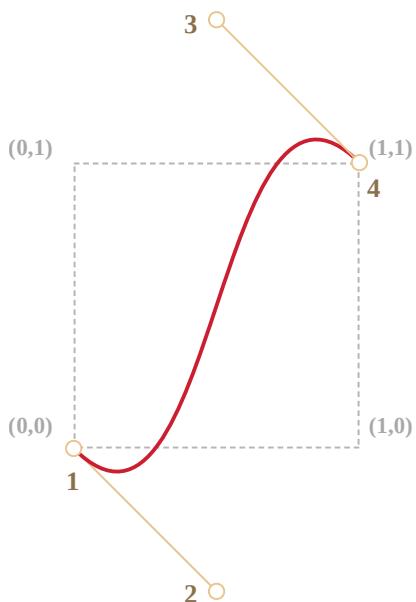
但是如果你点击列车，你会发现：

- 起初，列车会**反向**运动：`left` 会变得小于 `100px`。
- 然后，它会变回往前运动，并且超过 `400px`。

- 最后再返回——回到 `400px`。

<https://plnkr.co/edit/a4PvPk9aQYFlmZFI?p=preview>

为什么会这样？看一眼给定的贝塞尔曲线的图像你就会明白了。



我们把第二个点的 `y` 坐标移动到了小于 `0` 的位置，同时把第三个点的 `y` 坐标移动到了大于 `1` 的位置，因此曲线已经不再像一个四分之一圆了。`y` 坐标超出了常规的 `0..1` 的范围。

正如我们所知，`y` 表示『动画进程的完成度』。`y = 0` 表示属性的初始值，`y = 1` 则表示属性的最终值。因此，`y < 0` 意味着属性值要比初始值小，而 `y > 1` 则表明属性值要比最终值大。

当然了，`-1` 和 `2` 还是比较缓和的值。如果我们把 `y` 设为 `-99` 和 `99`，那么列车将会偏离地更远。

但是，如何针对特定的任务寻找到合适的贝塞尔曲线呢？事实上，有很多工具可以帮助到你。比方说，我们可以利用这个网站：<http://cubic-bezier.com/>。

## 阶跃函数 (Steps)

时间函数 `steps(number of steps[, start/end])` 允许你让动画分段进行，`number of steps` 表示需要拆分为多少段。

让我们通过一个数字的例子来演示一下。我们将会让数字以离散的方式变化，而不是以连续的方式。

为了达到效果，我们把动画拆分为 `9` 段：

```
#stripe.animate {
 transform: translate(-90%);
 transition: transform 9s steps(9, start);
}
```

`step(9, start)` 生效时:

[https://plnkr.co/edit/t6F7X2MxDV5wvI77?p=preview ↗](https://plnkr.co/edit/t6F7X2MxDV5wvI77?p=preview)

`steps` 的第一个参数表示段数。这个过渡动画将会被拆分为 9 个部分（每个占 10%）。时间间隔也会以同样的方式被拆分：9 秒会被分割为多个时长 1 秒的间隔。

第二个参数可以取 `start` 或 `end` 两者其一。

`start` 表示在动画开始时，我们需要立即开始第一段的动画。

可以观察到，在动画过程中：当我们单击数字之后，它会立马变为 1（即第一段），然后在下一秒开始的时候继续变化。

具体的流程如下：

- `0s —— -10%` (在第一秒开始的时候立即变化)
- `1s —— -20%`
- ...
- `8s —— -80%`
- (最后一秒，显示最终值)

另一个值 `end` 表示：改变不应该在最开始的时候发生，而是发生在每一段的最后时刻。

其流程如下：

- `0s —— 0`
- `1s —— -10%` (在第一秒结束时第一次变化)
- `2s —— -20%`
- ...
- `9s —— -90%`

`step(9, end)` 生效时:

[https://plnkr.co/edit/HZo8KsINYhiAjmcR?p=preview ↗](https://plnkr.co/edit/HZo8KsINYhiAjmcR?p=preview)

另外还有一些简写值：

- `step-start` —— 等同于 `steps(1, start)`。即：动画立刻开始，并且只有一段。也就是说，会立刻开始，紧接着就结束了，宛如没有动画一样。
- `step-end` —— 等同于 `steps(1, end)`。即：在 `transition-duration` 结束时生成一段动画。

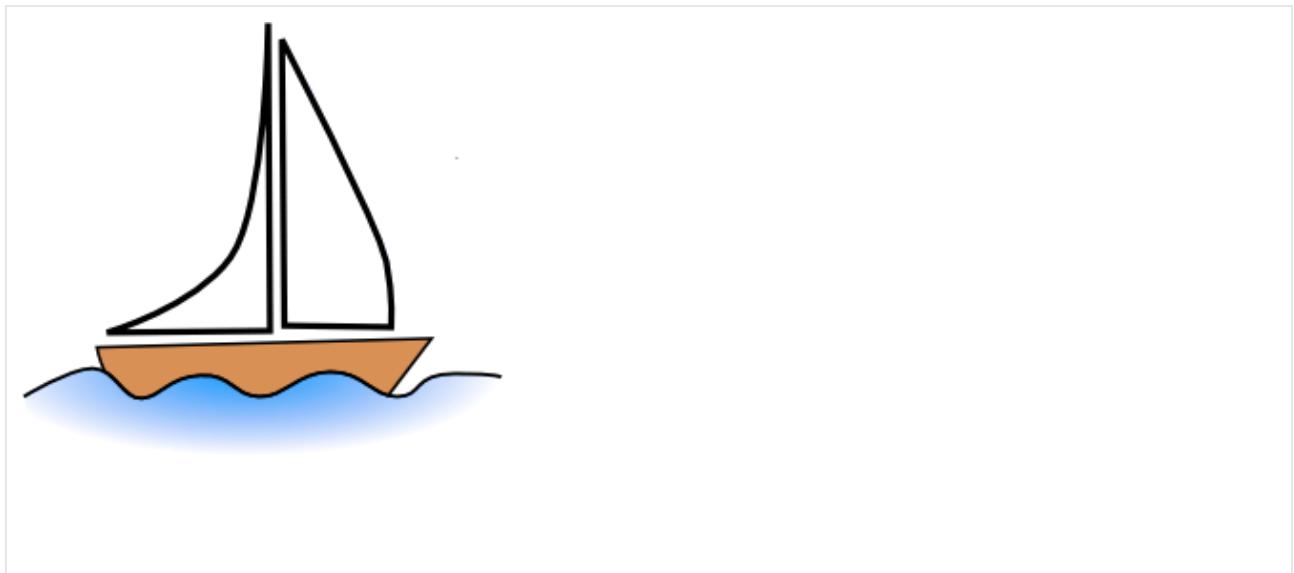
这些值很少会被用到，因为它们并不算是真正的动画，而是单步的变化。

## transitionend 事件

CSS 动画完成后，会触发 `transitionend` 事件。

这被广泛用于在动画结束后执行某种操作。我们也可以用它来串联动画。

举例来说，下面的小船会在点击后向右浮动，然后再回来。而且，每一次都会向右移动地更远一点：



这个动画通过 `go` 函数初始化，并且在每次动画完成后都会重复执行，并转变方向：

```
boat.onclick = function() {
 //...
 let times = 1;

 function go() {
 if (times % 2) {
 // 向右移动
 boat.classList.remove('back');
 boat.style.marginLeft = 100 * times + 200 + 'px';
 } else {
 // 向左移动
 boat.classList.add('back');
 boat.style.marginLeft = 100 * times - 200 + 'px';
 }
 }

 go();

 boat.addEventListener('transitionend', function() {
 times++;
 go();
 });
};
```

`transitionend` 的事件对象有几个特定的属性:

`event.propertyName` : 当前完成动画的属性, 这在我们同时为多个属性加上动画时会很有用。

`event.elapsedTime` : 动画完成的时间 (按秒计算), 不包括 `transition-delay`。

## 关键帧动画 (`Keyframes`)

我们可以通过 CSS 提供的 `@keyframes` 规则整合多个简单的动画。

它会指定某个动画的名称以及相应的规则: 哪个属性, 何时以及何地渲染动画。然后使用 `animation` 属性把动画绑定到相应的元素上, 并为其添加额外的参数。

这里有个详细的例子:

```
<div class="progress"></div>

<style>
 @keyframes go-left-right { /* 指定一个名字: "go-left-right" */
 from { left: 0px; } /* 从 left: 0px 开始 */
 to { left: calc(100% - 50px); } /* 移动至 left: 100%-50px */
 }

 .progress {
 animation: go-left-right 3s infinite alternate;
 /* 把动画 "go-left-right" 应用到元素上
 持续 3 秒
 持续次数: infinite
 每次都改变方向
 */
 position: relative;
 border: 2px solid green;
 width: 50px;
 height: 20px;
 background: lime;
 }
</style>
```



有许多关于 `@keyframes` 的文章以及一个[详细的规范说明](#)。

很可能你并不需要经常用到 `@keyframes`, 除非你的网站上有一直在运动的元素。

## 总结

CSS 动画允许你为一个或者多个属性的变化创建丝滑流畅（也可能不是）的过渡动画。

它们适用于大多数的动画需求。我们也可以使用 JavaScript 创建动画，下一章将会详细讲解相关内容。

相对于 JavaScript 动画，CSS 动画存在的特点如下：

## 优点

- 简单的事，简单地做。
- 快速，而且对 CPU 造成的影响很小。

## 不足

- JavaScript 动画更加灵活。它们可以实现任何动画逻辑，比如某个元素的爆炸效果。
- 不仅仅只是属性的变化。我们还可以在 JavaScript 中生成新元素用于动画。

本节已经介绍了可以使用 CSS 实现的主要动画类型，而且 `transitionend` 还允许在动画结束后执行 JavaScript 代码，因此它可以方便得与代码结合起来。

但是在下一节，我们将会学习一些 JavaScript 动画来实现更加复杂的效果。

## JavaScript 动画

JavaScript 动画可以处理 CSS 无法处理的事情。

例如，沿着具有与 Bezier 曲线不同的时序函数的复杂路径移动，或者实现画布上的动画。

## 使用 `setInterval`

从 HTML/CSS 的角度来看，动画是 `style` 属性的逐渐变化。例如，将 `style.left` 从 `0px` 变化到 `100px` 可以移动元素。

如果我们用 `setInterval` 每秒做 50 次小变化，看起来会更流畅。电影也是这样的原理：每秒 24 帧或更多帧足以使其看起来流畅。

伪代码如下：

```
let delay = 1000 / 50; // 每秒 50 帧
let timer = setInterval(function() {
 if (animation complete) clearInterval(timer);
```

```
 else increase style.left
}, delay)
```

更完整的动画示例：

```
let start = Date.now(); // 保存开始时间

let timer = setInterval(function() {
 // 距开始过了多长时间
 let timePassed = Date.now() - start;

 if (timePassed >= 2000) {
 clearInterval(timer); // 2 秒后结束动画
 return;
 }

 // 在 timePassed 时刻绘制动画
 draw(timePassed);

}, 20);

// 随着 timePassed 从 0 增加到 2000
// 将 left 的值从 0px 增加到 400px
function draw(timePassed) {
 train.style.left = timePassed / 5 + 'px';
}
```

点击演示：

<https://plnkr.co/edit/xS2LwvelKh2W7MKr?p=preview>

## 使用 requestAnimationFrame

假设我们有几个同时运行的动画。

如果我们单独运行它们，每个都有自己的 `setInterval(..., 20)`，那么浏览器必须以比 **20ms** 更频繁的速度重绘。

每个 `setInterval` 每 **20ms** 触发一次，但它们相互独立，因此 **20ms** 内将有多个独立运行的重绘。

这几个独立的重绘应该组合在一起，以便浏览器更加容易处理。

换句话说，像下面这样：

```
setInterval(function() {
 animate1();
 animate2();
})
```

```
animate3();
}, 20)
```

.....比这样更好：

```
setInterval/animate1, 20);
setInterval/animate2, 20);
setInterval/animate3, 20);
```

还有一件事需要记住。有时当 CPU 过载时，或者有其他原因需要降低重绘频率。例如，如果浏览器选项卡被隐藏，那么绘图完全没有意义。

有一个标准[动画时序](#) 提供了 `requestAnimationFrame` 函数。

它解决了所有这些问题，甚至更多其它的问题。

语法：

```
let requestId = requestAnimationFrame(callback);
```

这会让 `callback` 函数在浏览器每次重绘的最近时间运行。

如果我们对 `callback` 中的元素进行变化，这些变化将与其他 `requestAnimationFrame` 回调和 CSS 动画组合在一起。因此，只会有一次几何重新计算和重绘，而不是多次。

返回值 `requestId` 可用来取消回调：

```
// 取消回调的周期执行
cancelAnimationFrame(requestId);
```

`callback` 得到一个参数——从页面加载开始经过的毫秒数。这个时间也可通过调用 [performance.now\(\)](#) 得到。

通常 `callback` 很快就会运行，除非 CPU 过载或笔记本电量消耗殆尽，或者其他原因。

下面的代码显示了 `requestAnimationFrame` 的前 10 次运行之间的时间间隔。通常是 10-20ms：

```
<script>
let prev = performance.now();
let times = 0;
```

```
requestAnimationFrame(function measure(time) {
 document.body.insertAdjacentHTML("beforeEnd", Math.floor(time - prev) + " ");
 prev = time;

 if (times++ < 10) requestAnimationFrame(measure);
});
</script>
```

## 结构化动画

现在我们可以在 `requestAnimationFrame` 基础上创建一个更通用的动画函数:

```
function animate({timing, draw, duration}) {

 let start = performance.now();

 requestAnimationFrame(function animate(time) {
 // timeFraction 从 0 增加到 1
 let timeFraction = (time - start) / duration;
 if (timeFraction > 1) timeFraction = 1;

 // 计算当前动画状态
 let progress = timing(timeFraction);

 draw(progress); // 绘制

 if (timeFraction < 1) {
 requestAnimationFrame(animate);
 }
 });
}
```

`animate` 函数接受 3 个描述动画的基本参数:

### duration

动画总时间, 比如 `1000`。

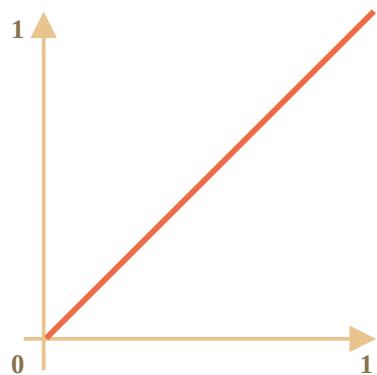
### timing(`timeFraction`)

时序函数, 类似 CSS 属性 `transition-timing-function`, 传入一个已过去的时间与总时间之比的小数 (`0` 代表开始, `1` 代表结束), 返回动画完成度 (类似 Bezier 曲线中的 `y`)。

例如, 线性函数意味着动画以相同的速度均匀地进行:

```
function linear(timeFraction) {
 return timeFraction;
}
```

图像如下：



它类似于 `transition-timing-function: linear`。后文有更多有趣的变体。

### **draw(progress)**

获取动画完成状态并绘制的函数。值 `progress = 0` 表示开始动画状态，`progress = 1` 表示结束状态。

这是实际绘制动画的函数。

它可以移动元素：

```
function draw(progress) {
 train.style.left = progress + 'px';
}
```

.....或者做任何其他事情，我们可以以任何方式为任何事物制作动画。

让我们使用我们的函数将元素的 `width` 从 `0` 变化为 `100%`。

点击演示元素：

[https://plnkr.co/edit/mtdDQ89NcD1flbGv?p=preview ↗](https://plnkr.co/edit/mtdDQ89NcD1flbGv?p=preview)

它的代码如下：

```
animate({
 duration: 1000,
 timing(timeFraction) {
 return timeFraction;
```

```
},
draw(progress) {
 elem.style.width = progress * 100 + '%';
}
});
```

与 CSS 动画不同，我们可以在这里设计任何时序函数和任何绘图函数。时序函数不受 Bezier 曲线的限制。并且 `draw` 不局限于操作 CSS 属性，还可以为类似烟花动画或其他动画创建新元素。

## 时序函数

上文我们看到了最简单的线性时序函数。

让我们看看更多。我们将尝试使用不同时序函数的移动动画来查看它们的工作原理。

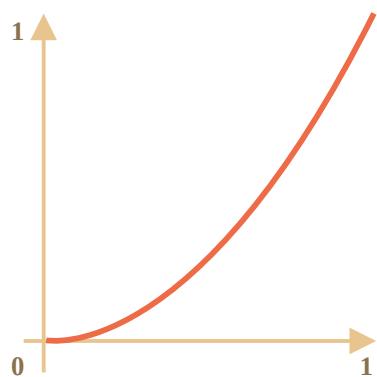
### n 次幂

如果我们想加速动画，我们可以让 `progress` 为  $n$  次幂。

例如，抛物线：

```
function quad(timeFraction) {
 return Math.pow(timeFraction, 2)
}
```

图像如下：

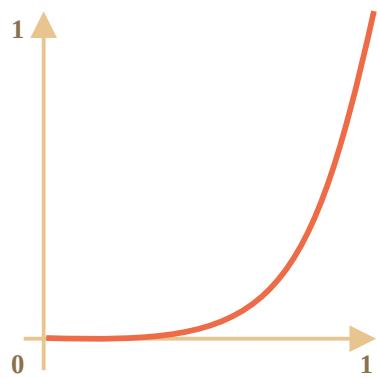


看看实际效果（点击激活）：

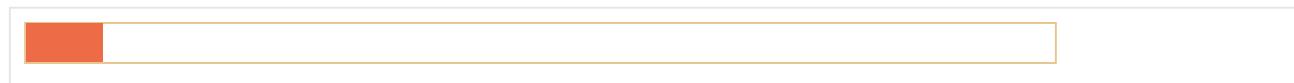


.....或者三次曲线甚至使用更大的  $n$ 。增大幂会让动画加速得更快。

下面是 `progress` 为  $5$  次幂的图像：



实际效果:

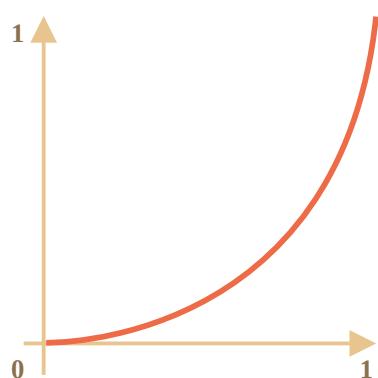


## 圆弧

函数:

```
function circ(timeFraction) {
 return 1 - Math.sin(Math.acos(timeFraction));
}
```

图像:



## 反弹: 弓箭射击

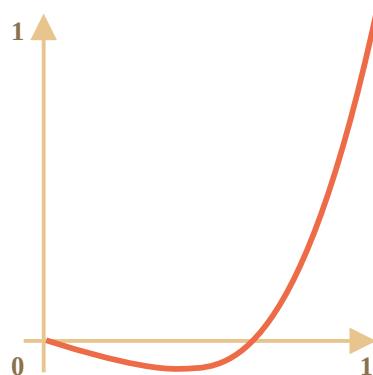
此函数执行“弓箭射击”。首先，我们“拉弓弦”，然后“射击”。

与以前的函数不同，它取决于附加参数  $x$ ，即“弹性系数”。“拉弓弦”的距离由它定义。

代码如下：

```
function back(x, timeFraction) {
 return Math.pow(timeFraction, 2) * ((x + 1) * timeFraction - x);
}
```

$x = 1.5$  时的图像：



在动画中我们使用特定的  $x$  值。下面是  $x = 1.5$  时的例子：



## 弹跳

想象一下，我们正在抛球。球落下之后，弹跳几次然后停下来。

`bounce` 函数也是如此，但顺序相反：“bouncing”立即启动。它使用了几个特殊的系数：

```
function bounce(timeFraction) {
 for (let a = 0, b = 1, result; 1; a += b, b /= 2) {
 if (timeFraction >= (7 - 4 * a) / 11) {
 return -Math.pow((11 - 6 * a - 11 * timeFraction) / 4, 2) + Math.pow(b, 2)
 }
 }
}
```

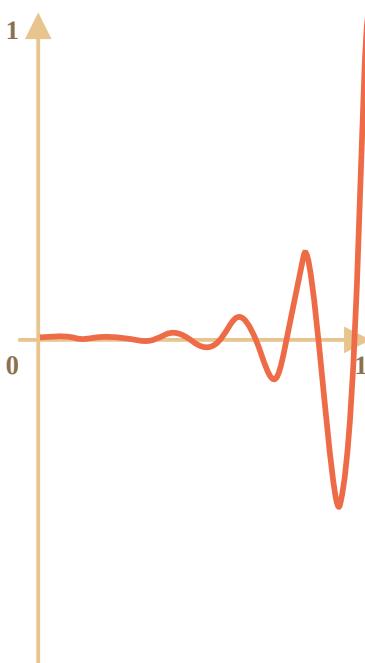
演示：



## 伸缩动画

另一个“伸缩”函数接受附加参数  $x$  作为“初始范围”。

```
function elastic(x, timeFraction) {
 return Math.pow(2, 10 * (timeFraction - 1)) * Math.cos(20 * Math.PI * x / 3 * timeFraction);
}
```



x=1.5 时的图像:

x=1.5 时的演示



## 逆转: ease\*

我们有一组时序函数。它们的直接应用称为“easeIn”。

有时我们需要以相反的顺序显示动画。这是通过“easeOut”变换完成的。

### easeOut

在“easeOut”模式中，我们将 timing 函数封装到 timingEaseOut 中：

```
timingEaseOut(timeFraction) = 1 - timing(1 - timeFraction);
```

换句话说，我们有一个“变换”函数 makeEaseOut，它接受一个“常规”时序函数 timing 并返回一个封装器，里面封装了 timing 函数：

```
// 接受时序函数，返回变换后的变体
function makeEaseOut(timing) {
 return function(timeFraction) {
```

```
 return 1 - timing(1 - timeFraction);
}
}
```

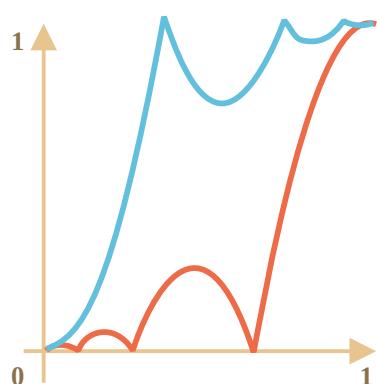
例如，我们可以使用上面描述的 `bounce` 函数：

```
let bounceEaseOut = makeEaseOut(bounce);
```

这样，弹跳不会在动画开始时执行，而是在动画结束时。这样看起来更好：

[https://plnkr.co/edit/CMtBB5AZhMaBWHu8?p=preview ↗](https://plnkr.co/edit/CMtBB5AZhMaBWHu8?p=preview)

在这里，我们可以看到变换如何改变函数的行为：



如果在开始时有动画效果，比如弹跳——那么它将在最后显示。

上图中常规弹跳为红色，`easeOut` 弹跳为蓝色。

- 常规弹跳——物体在底部弹跳，然后突然跳到顶部。
- `easeOut` 变换之后——物体跳到顶部之后，在那里弹跳。

## `easeInOut`

我们还可以在动画的开头和结尾都显示效果。该变换称为“`easeInOut`”。

给定时序函数，我们按下面的方式计算动画状态：

```
if (timeFraction <= 0.5) { // 动画前半部分
 return timing(2 * timeFraction) / 2;
} else { // 动画后半部分
 return (2 - timing(2 * (1 - timeFraction))) / 2;
}
```

封装器代码：

```

function makeEaseInOut(timing) {
 return function(timeFraction) {
 if (timeFraction < .5)
 return timing(2 * timeFraction) / 2;
 else
 return (2 - timing(2 * (1 - timeFraction))) / 2;
 }
}

bounceEaseInOut = makeEaseInOut(bounce);

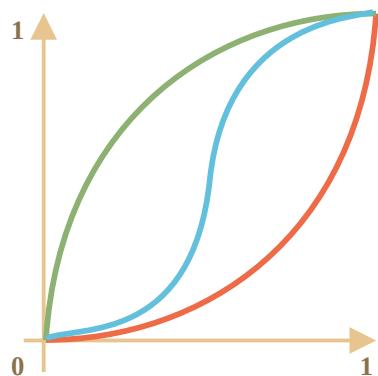
```

`bounceEaseInOut` 演示如下:

[https://plnkr.co/edit/6UyJGES6NkcXMRmW?p=preview ↗](https://plnkr.co/edit/6UyJGES6NkcXMRmW?p=preview)

“easeInOut” 变换将两个图像连接成一个：动画的前半部分为“easeIn”（常规），后半部分为“easeOut”（反向）。

如果我们比较 `circ` 时序函数的 `easeIn`、`easeOut` 和 `easeInOut` 的图像，就可以清楚地看到效果：



- 红色是 `circ` (`easeIn`) 的常规变体。
- 绿色 —— `easeOut`。
- 蓝色 —— `easeInOut`。

正如我们所看到的，动画前半部分的图形是缩小的“easeIn”，后半部分是缩小的“easeOut”。结果是动画以相同的效果开始和结束。

## 更有趣的“draw”

除了移动元素，我们还可以做其他事情。我们所需要的只是写出合适的 `draw`。

这是动画形式的“弹跳”文字输入：

[https://plnkr.co/edit/2xgcs2upgj9hZoi6?p=preview ↗](https://plnkr.co/edit/2xgcs2upgj9hZoi6?p=preview)

## 总结

JavaScript 动画应该通过 `requestAnimationFrame` 实现。该内置方法允许设置回调函数，以便在浏览器准备重绘时运行。那通常很快，但确切的时间取决于浏览器。

当页面在后台时，根本没有重绘，因此回调将不会运行：动画将被暂停并且不会消耗资源。那很棒。

这是设置大多数动画的 helper 函数 `animate`：

```
function animate({timing, draw, duration}) {

 let start = performance.now();

 requestAnimationFrame(function animate(time) {
 // timeFraction 从 0 增加到 1
 let timeFraction = (time - start) / duration;
 if (timeFraction > 1) timeFraction = 1;

 // 计算当前动画状态
 let progress = timing(timeFraction);

 draw(progress); // 绘制

 if (timeFraction < 1) {
 requestAnimationFrame(animate);
 }
 });
}
```

参数：

- `duration` —— 动画运行的总毫秒数。
- `timing` —— 计算动画进度的函数。获取从 0 到 1 的小数时间，返回动画进度，通常也是从 0 到 1。
- `draw` —— 绘制动画的函数。

当然我们可以改进它，增加更多花里胡哨的东西，但 JavaScript 动画不是经常用到。它们用于做一些有趣和不标准的事情。因此，您大可在必要时再添加所需的功能。

JavaScript 动画可以使用任何时序函数。我们介绍了很多例子和变换，使它们更加通用。与 CSS 不同，我们不仅限于 Bezier 曲线。

`draw` 也是如此：我们可以将任何东西动画化，而不仅仅是 CSS 属性。

## Web components

Web components is a set of standards to make self-contained components: custom HTML-elements with their own properties and methods, encapsulated DOM and styles.

## 从星球轨道的高度讲起

这一部分我们将会讲述关于「Web Components」的一系列现代标准。

到目前为止，这些标准仍然在制定中。其中一些特性已经被很好地支持并集成到了现代 HTML/DOM 标准中，但是还有部分特性仍然处在草案阶段。你可以在任何浏览器中尝试一些例子，Google Chrome 可能是对这些新特性支持得最好的浏览器。猜测可能是因为 Google 公司的人本身就是很多相关标准的支持者。

## 共通之处在于.....

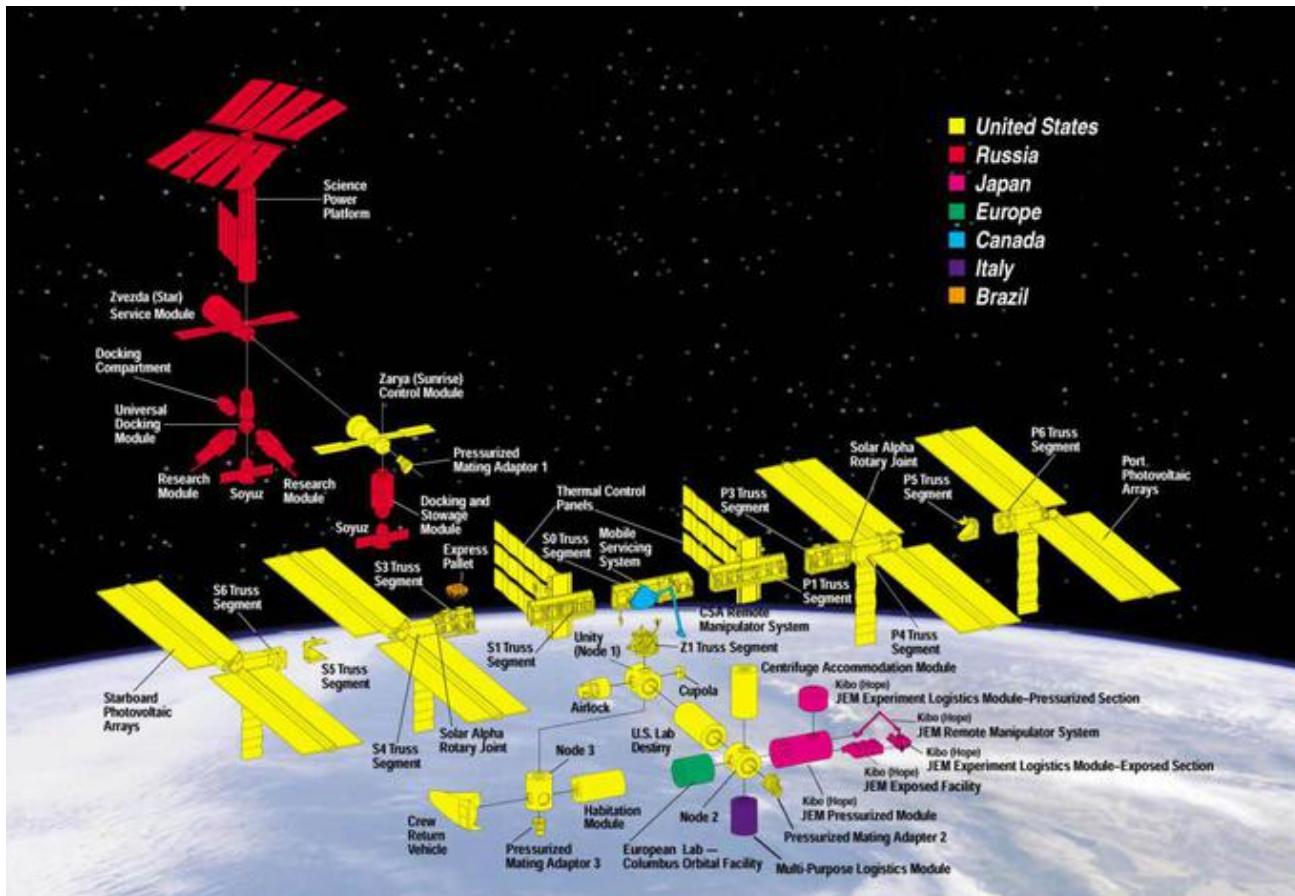
整个组件化的概念并不是最新才提出的。很多框架和其他地方已经广泛地应用了组件化的设计。

在我们开始探讨实现细节之前，先让我们看看人类的伟大成就：



这是国际空间站（ISS）。

这是其组成结构（大致的）：



这个国际空间站：

- 由许多组件构成。
- 各个组件都由很多的更小的部分组成，
- 组件都非常复杂，远比大部分网站更复杂。
- 国际化的组件开发团队，整个工作由不同国家、说着不同语言的人共同完成。

.....并且这个家伙能飞，它让人类在太空中能够生存！

这些复杂的设备是如何被创建的？

我们可以从中借鉴哪些原则，让我们的开发项目同样的可靠并且可大规模化呢？或者至少让我们可以接近这些目标。

## 组件化架构

众所周知，开发复杂软件的原则是：不要让软件复杂。

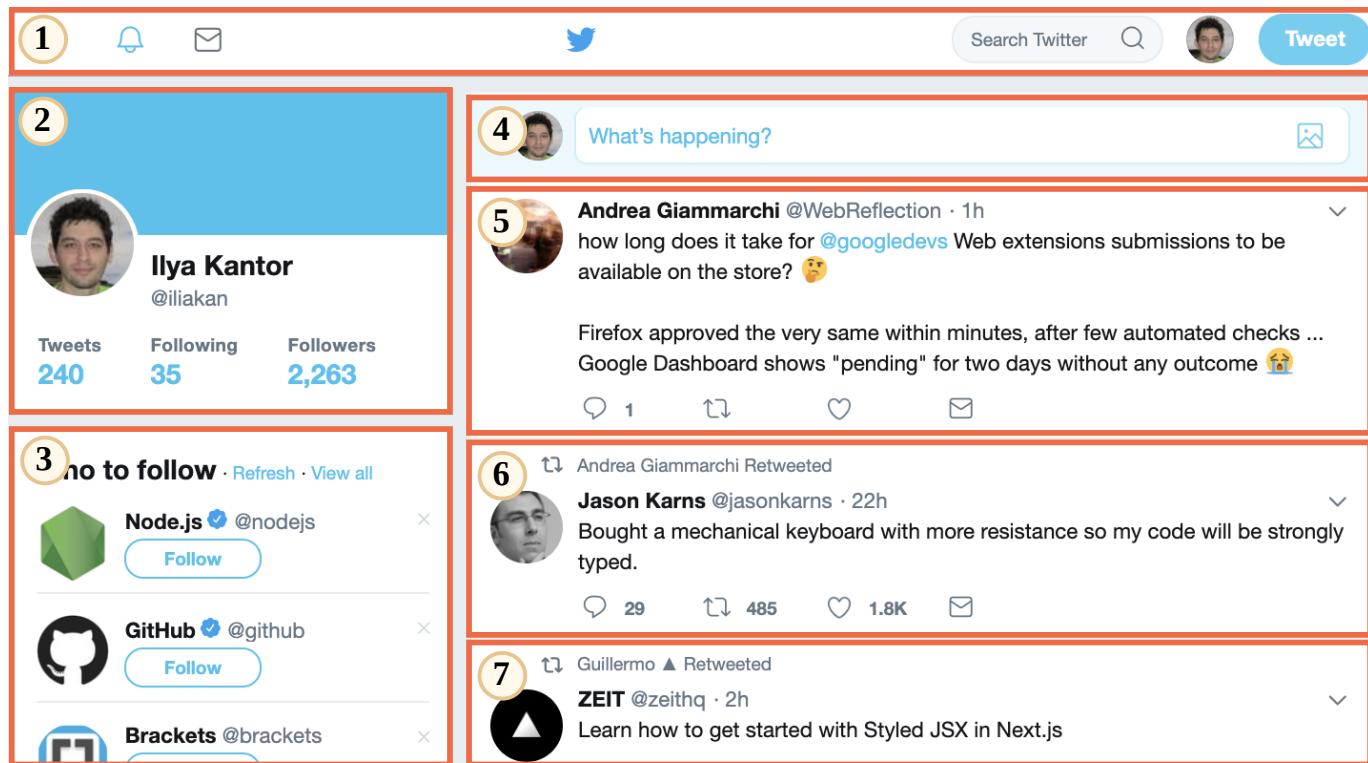
如果某个部分变得复杂了——将其拆分成更简单的部分，再以最简明的方式组合起来。

只有让复杂的事情简单化的架构才是好架构。

我们可以把用户界面拆分为若干可视化组件：每个组件都在页面上占有一块位置，可以执行一个明确的任务，并且可以和其他组件区分开。

接下来看一个实际的网站的例子，比如 Twitter。

非常自然地，可以拆分为几个组件：



1. 顶部导航栏。
2. 用户信息。
3. 关注推荐。
4. 提交表格。
5. (6, 7也是) —— 消息。

组件也可以包含子组件，比如消息组件可能是更高阶组件「消息列表」的子组件。可点击的用户头像可能也是一个组件，这样的例子还有很多。

我们如何划分一个组件呢？直觉、经验和常识可以帮助我们完成这件事。通常情况下，如果一个独立可视化实体，我们可以描述其可以做什么和如何在页面上交互，那么就可以将其划分为一个组件。在上面的例子中，这个页面存在几个模块，每个模块都有自己的角色，所以把它们划分为组件是合理的。

一个组件有：

- 自己的 JavaScript 类。
- DOM 结构，并且只由自己的类管理，无法被外部代码操作。（「封装」原则）。
- CSS 样式，作用在这个组件上。
- API：事件，类方法等等，让组件可以与其他组件交互。

再说一遍，整个「组件化」的概念并不是什么特别的东西。

现在已经有了很多框架和开发方法论可以实现组件化，它们各个都有自己的卖点。通常情况下，采用特殊的 CSS 类命名和一些规范，已经可以带来「组件化的感觉」——即 CSS 作用域和 DOM 封装。

而现在浏览器已经原生支持了「Web Components」，我们就可以不用再自己去模拟组件化的结构了。

- [Custom elements ↗](#) —— 用于自定义 HTML 元素。
- [Shadow DOM ↗](#) —— 为组件创建内部 DOM，它对外部是不可见的。
- [CSS Scoping ↗](#) —— 申明仅应用于组件的 Shadow DOM 内的样式。
- [Event retargeting ↗](#) 以及更多的小东西，让自定义组件更适用于开发工作。

在下一篇中我们将会更细致地讲述「Custom Elements」——一个已经被很广泛支持的 Web Components 重要组成部分。

## Custom elements

我们可以通过描述带有自己的方法、属性和事件等的类来创建自定义 HTML 元素。

在 custom elements（自定义标签）定义完成之后，我们可以将其和 HTML 的内置标签一同使用。

这是一件好事，因为虽然 HTML 有非常多的标签，但仍然是有穷尽的。如果我们需要像 `<easy-tabs>`、`<sliding-carousel>`、`<beautiful-upload>` …… 这样的标签，内置标签并不能满足我们。

我们可以把上述的标签定义为特殊的类，然后使用它们，就好像它们本来就是 HTML 的一部分一样。

Custom elements 有两种：

1. **Autonomous custom elements**（自主自定义标签）——“全新的”元素，继承自 `HTMLElement` 抽象类。
2. **Customized built-in elements**（自定义内置元素）——继承内置的 HTML 元素，比如自定义 `HTMLButtonElement` 等。

我们将会先创建 autonomous 元素，然后再创建 customized built-in 元素。

在创建 custom elements 的时候，我们需要告诉浏览器一些细节，包括：如何展示它，以及在添加元素到页面和将其从页面移除的时候需要做什么，等等。

通过创建一个带有几个特殊方法的类，我们可以完成这件事。这非常容易实现，我们只需要添加几个方法就行了，同时这些方法都不是必须的。

下面列出了这几个方法的概述：

```
class MyElement extends HTMLElement {
 constructor() {
 super();
 // 元素在这里创建
 }

 connectedCallback() {
 // 在元素被添加到文档之后，浏览器会调用这个方法
 // （如果一个元素被反复添加到文档／移除文档，那么这个方法会被多次调用）
 }

 disconnectedCallback() {
 // 在元素从文档移除到时候，浏览器会调用这个方法
 // （如果一个元素被反复添加到文档／移除文档，那么这个方法会被多次调用）
 }

 static get observedAttributes() {
 return /* 属性数组，这些属性的变化会被被监视 */;
 }

 attributeChangedCallback(name, oldValue, newValue) {
 // 当上面数组里面的属性变化的时候，这个方法会被调用
 }

 adoptedCallback() {
 // 在元素被移动到新的文档的时候，这个方法会被调用
 // （document.adoptNode 会用到，非常少见）
 }

 // 还可以添加更多的元素方法和属性
}
```

在申明了上面几个方法之后，我们需要注册元素：

```
// 让浏览器知道我们新定义的类是为 <my-element> 服务的
customElements.define("my-element", MyElement);
```

现在当任何带有 `<my-element>` 标签的元素被创建的时候，一个 `MyElement` 的实例也会被创建，并且前面提到的方法也会被调用。我们同样可以使用 `document.createElement('my-element')` 在 JavaScript 里创建元素。

### ❶ Custom element 名称必须包括一个短横线 -

Custom element 名称必须包括一个短横线 `-`，比如 `my-element` 和 `super-button` 都是有效的元素名，但 `myelement` 并不是。

这是为了确保 custom element 和内置 HTML 元素之间不会发生命名冲突。

## 例子：“time-formatted”

举个例子，HTML 里面已经有 `<time>` 元素了，用于显示日期／时间。但是这个标签本身并不会对时间进行任何格式化处理。

让我们来创建一个可以展示适用于当前浏览器语言的时间格式的 `<time-formatted>` 元素：

```
<script>
class TimeFormatted extends HTMLElement { // (1)

 connectedCallback() {
 let date = new Date(this.getAttribute('datetime') || Date.now());

 this.innerHTML = new Intl.DateTimeFormat("default", {
 year: this.getAttribute('year') || undefined,
 month: this.getAttribute('month') || undefined,
 day: this.getAttribute('day') || undefined,
 hour: this.getAttribute('hour') || undefined,
 minute: this.getAttribute('minute') || undefined,
 second: this.getAttribute('second') || undefined,
 timeZoneName: this.getAttribute('time-zone-name') || undefined,
 }).format(date);
 }

}

customElements.define("time-formatted", TimeFormatted); // (2)
</script>

<!-- (3) -->
<time-formatted datetime="2019-12-01"
 year="numeric" month="long" day="numeric"
 hour="numeric" minute="numeric" second="numeric"
 time-zone-name="short"
></time-formatted>
```

December 1, 2019, 3:00:00 AM GMT+3

1. 这个类只有一个方法 `connectedCallback()` —— 在 `<time-formatted>` 元素被添加到页面的时候，浏览器会调用这个方法（或者当 HTML 解析器检测到它的时候），它使用了内置的时间格式化工具 [Intl.DateTimeFormat](#)，这个工具可以非常好地展示格式化之后的时间，在各浏览器中兼容性都非常好。
2. 我们需要通过 `customElements.define(tag, class)` 来注册这个新元素。
3. 接下来在任何地方我们都可以使用这个新元素了。

## i Custom elements 升级

如果浏览器在 `customElements.define` 之前的任何地方见到了 `<time-formatted>` 元素，并不会报错。但会把这个元素当作未知元素，就像任何非标准标签一样。

`:not(:defined)` CSS 选择器可以对这样「未定义」的元素加上样式。

当 `customElement.define` 被调用的时候，他们被「升级」了：一个新的 `TimeFormatted` 元素为每一个标签创建了，并且 `connectedCallback` 被调用。他们变成了 `:defined`。

我们可以通过这些方法来获取更多的自定义标签的信息：

- `customElements.get(name)` —— 返回指定 custom element `name` 的类。
- `customElements.whenDefined(name)` – 返回一个 promise，将会在这个具有给定 `name` 的 custom element 变为已定义状态的时候 `resolve`（不带值）。

## i 在 `connectedCallback` 中渲染，而不是 `constructor` 中

在上面的例子中，元素里面的内容是在 `connectedCallback` 中渲染（创建）的。

为什么不在 `constructor` 中渲染？

原因很简单：在 `constructor` 被调用的时候，还为时过早。虽然这个元素实例已经被创建了，但还没有插入页面。在这个阶段，浏览器还没有处理／创建元素属性：调用 `getAttribute` 将会得到 `null`。所以我们并不能在那里渲染元素。

而且，如果你仔细考虑，这样作对于性能更好——推迟渲染直到真正需要的时候。

在元素被添加到文档的时候，它的 `connectedCallback` 方法会被调用。这个元素不仅仅是被添加为了另一个元素的子元素，同样也成为了页面的一部分。因此我们可以构建分离的 DOM，创建元素并且让它们为之后的使用准备好。它们只有在插入页面的时候才会真的被渲染。

## 监视属性

我们目前的 `<time-formatted>` 实现中，在元素渲染以后，后续的属性变化并不会带来任何影响。这对于 HTML 元素来说有点奇怪。通常当我们改变一个属性的时候，比如 `a.href`，我们会预期立即看到变化。我们将会在下面修正这一点。

为了监视这些属性，我们可以在 `observedAttributes()` static getter 中提供属性列表。当这些属性发生变化的时候，`attributeChangedCallback` 会被调用。出于性能优化的考虑，其他属性变化的时候并不会触发这个回调方法。

以下是 `<time-formatted>` 的新版本，它会在属性变化的时候自动更新：

```
<script>
class TimeFormatted extends HTMLElement {

 render() { // (1)
 let date = new Date(this.getAttribute('datetime') || Date.now());

 this.innerHTML = new Intl.DateTimeFormat("default", {
 year: this.getAttribute('year') || undefined,
 month: this.getAttribute('month') || undefined,
 day: this.getAttribute('day') || undefined,
 hour: this.getAttribute('hour') || undefined,
 minute: this.getAttribute('minute') || undefined,
 second: this.getAttribute('second') || undefined,
 timeZoneName: this.getAttribute('time-zone-name') || undefined,
 }).format(date);
 }

 connectedCallback() { // (2)
 if (!this.rendered) {
 this.render();
 this.rendered = true;
 }
 }

 static get observedAttributes() { // (3)
 return ['datetime', 'year', 'month', 'day', 'hour', 'minute', 'second', 'time-zo
 }

 attributeChangedCallback(name, oldValue, newValue) { // (4)
 this.render();
 }

}

customElements.define("time-formatted", TimeFormatted);
</script>

<time-formatted id="elem" hour="numeric" minute="numeric" second="numeric"></time-f
<script>
setInterval(() => elem.setAttribute('datetime', new Date()), 1000); // (5)
</script>
```

10:57:27 AM

1. 渲染逻辑被移动到了 `render()` 这个辅助方法里面。
2. 这个方法在元素被插入到页面的时候调用。
3. `attributeChangedCallback` 在 `observedAttributes()` 里的属性改变的时候被调用。
4. ..... 然后重渲染元素。
5. 最终，一个计时器就这样被我们轻松地实现了。

## 渲染顺序

在 `HTML` 解析器构建 `DOM` 的时候，会按照先后顺序处理元素，先处理父级元素再处理子元素。例如，如果我们有 `<outer><inner></inner></outer>`，那么 `<outer>` 元素会首先被创建并接入到 `DOM`，然后才是 `<inner>`。

这对 `custom elements` 产生了重要影响。

比如，如果一个 `custom element` 想要在 `connectedCallback` 内访问 `innerHTML`，它什么也拿不到：

```
<script>
customElements.define('user-info', class extends HTMLElement {

 connectedCallback() {
 alert(this.innerHTML); // empty (*)
 }
});

</script>

<user-info>John</user-info>
```

如果你运行上面的代码，`alert` 出来的内容是空的。

这正是因为在那个阶段，子元素还不存在，`DOM` 还没有完成构建。`HTML` 解析器先连接 `custom element` `<user-info>`，然后再处理子元素，但是那时候子元素还没有加载上。

如果我们要给 `custom element` 传入信息，我们可以使用元素属性。它们是即时生效的。

或者，如果我们需要子元素，我们可以使用延迟时间为零的 `setTimeout` 来推迟访问子元素。

这样是可行的：

```
<script>
customElements.define('user-info', class extends HTMLElement {
```

```
connectedCallback() {
 setTimeout(() => alert(this.innerHTML)); // John (*)
}

});

</script>

<user-info>John</user-info>
```

现在 `alert` 在 `(*)` 行展示了「John」，因为我们是在 `HTML` 解析完成之后，才异步执行了这段程序。我们在这个时候处理必要的子元素并且结束初始化过程。

另一方面，这个方案并不是完美的。如果嵌套的 `custom element` 同样使用了 `setTimeout` 来初始化自身，那么它们会按照先后顺序执行：外层的 `setTimeout` 首先触发，然后才是内层的。

这样外层元素还是早于内层元素结束初始化。

让我们用一个例子来说明：

```
<script>
customElements.define('user-info', class extends HTMLElement {
 connectedCallback() {
 alert(`#${this.id} 已连接。`);
 setTimeout(() => alert(`#${this.id} 初始化完成。`));
 }
});
</script>

<user-info id="outer">
 <user-info id="inner"></user-info>
</user-info>
```

输出顺序：

1. `outer` 已连接。
2. `inner` 已连接。
3. `outer` 初始化完成。
4. `inner` 初始化完成。

我们可以很明显地看到外层元素并没有等待内层元素。

并没有任何内置的回调方法可以在嵌套元素渲染好之后通知我们。但我们可以自己实现这样的回调。比如，内层元素可以分派像 `initialized` 这样的事件，同时外层的元素监听这样的事件并做出响应。

## Customized built-in elements

我们创建的 `<time-formatted>` 这些新元素，并没有任何相关的语义。搜索引擎并不知晓它们的存在，同时无障碍设备也无法处理它们。

但上述两点同样是非常重要的。比如，搜索引擎会对这些事情感兴趣，比如我们真的展示了时间。或者如果我们创建了一个特别的按钮，为什么不复用已有的 `<button>` 功能呢？

我们可以通过继承内置元素的类来扩展和定制它们。

比如，按钮是 `HTMLButtonElement` 的实例，让我们在这个基础上创建元素。

1. 我们的类继承自 `HTMLButtonElement`：

```
class HelloButton extends HTMLButtonElement { /* custom element 方法 */ }
```

2. 给 `customElements.define` 提供定义标签的第三个参数：

```
customElements.define('hello-button', HelloButton, {extends: 'button'});
```

这一步是必要的，因为不同的标签会共享同一个类。

3. 最后，插入一个普通的 `<button>` 标签，但添加 `is="hello-button"` 到这个元素，这样就可以使用我们的 custom element：

```
<button is="hello-button">...</button>
```

下面是一个完整的例子：

```
<script>
// 这个按钮在被点击的时候说 "hello"
class HelloButton extends HTMLButtonElement {
 constructor() {
 super();
 this.addEventListener('click', () => alert("Hello!"));
 }
}

customElements.define('hello-button', HelloButton, {extends: 'button'});
</script>

<button is="hello-button">Click me</button>

<button is="hello-button" disabled>Disabled</button>
```

```
Click me Disabled
```

我们新定义的按钮继承了内置按钮，所以它拥有和内置按钮相同的样式和标准特性，比如 `disabled` 属性。

## 引用参考

- HTML 现行标准: <https://html.spec.whatwg.org/#custom-elements>。
- 兼容性: <https://caniuse.com/#feat=custom-elements>。

## 总结

有两种 custom element:

- “Autonomous”——全新的标签，继承 `HTMLElement`。

定义方式:

```
class MyElement extends HTMLElement {
 constructor() { super(); /* ... */ }
 connectedCallback() { /* ... */ }
 disconnectedCallback() { /* ... */ }
 static get observedAttributes() { return /* ... */; }
 attributeChangedCallback(name, oldValue, newValue) { /* ... */ }
 adoptedCallback() { /* ... */ }
}
customElements.define('my-element', MyElement);
/* <my-element> */
```

- “Customized built-in elements”——已有元素的扩展。

需要多一个 `.define` 参数，同时 `is="..."` 在 HTML 中:

```
class MyButton extends HTMLButtonElement { /*...*/ }
customElements.define('my-button', MyElement, {extends: 'button'});
/* <button is="my-button"> */
```

Custom element 在各浏览器中的兼容性已经非常好了。Edge 支持地相对较差，但是我们可以使用 polyfill <https://github.com/webcomponents/webcomponentsjs>。

## 影子 DOM (Shadow DOM)

Shadow DOM 为封装而生。它可以让一个组件拥有自己的「影子」DOM 树，这个 DOM 树不能在主文档中被任意访问，可能拥有局部样式规则，还有其他特性。

## 内建 shadow DOM

你是否曾经思考过复杂的浏览器控件是如何被创建和添加样式的？

比如 `<input type="range">`：



浏览器在内部使用 DOM/CSS 来绘制它们。这个 DOM 结构一般来说对我们是隐藏的，但我们可以开发者工具里面看见它。比如，在 Chrome 里，我们需要打开「Show user agent shadow DOM」选项。

然后 `<input type="range">` 看起来会像这样：

```
▼<input type="range"> == $0
 ▼#shadow-root (user-agent)
 ▼<div>
 ▼<div pseudo="-webkit-slider-runnable-track" id="track">
 <div id="thumb"></div>
 </div>
 </div>
 </input>
```

你在 `#shadow-root` 下看到的就是被称为「shadow DOM」的东西。

我们不能使用一般的 JavaScript 调用或者选择器来获取内建 shadow DOM 元素。它们不是常规的子元素，而是一个强大的封装手段。

在上面的例子中，我们可以看到一个有用的属性 `pseudo`。这是一个因为历史原因而存在的属性，并不在标准中。我们可以使用它来给子元素加上 CSS 样式，像这样：

```
<style>
/* 让滑块轨道变红 */
input::-webkit-slider-runnable-track {
 background: red;
}
</style>

<input type="range">
```



重申一次，`pseudo` 是一个非标准的属性。按照时间顺序来说，浏览器首先实验了使用内部 DOM 结构来实现控件，然后，在一段时间之后，shadow DOM 才被标准化来让我们，开发者们，做类似的事。

接下来，我们将要使用现代 shadow DOM 标准，它在 [DOM spec](#) 和其他相关标准中可以被找到。

## Shadow tree

一个 DOM 元素可以有以下两类 DOM 子树：

1. Light tree（光明树）——一个常规 DOM 子树，由 HTML 子元素组成。我们在之前章节看到的所有子树都是「光明的」。
2. Shadow tree（影子树）——一个隐藏的 DOM 子树，不在 HTML 中反映，无法被察觉。

如果一个元素同时有以上两种子树，那么浏览器只渲染 shadow tree。但是我们同样可以设置两种树的组合。我们将会在后面的章节 [Shadow DOM 插槽，组成](#) 中看到更多细节。

影子树可以在自定义元素中被使用，其作用是隐藏组件内部结构和添加只在组件内有效的样式。

比如，这个 `<show-hello>` 元素将它的内部 DOM 隐藏在了影子里面：

```
<script>
customElements.define('show-hello', class extends HTMLElement {
 connectedCallback() {
 const shadow = this.attachShadow({mode: 'open'});
 shadow.innerHTML = `<p>
 Hello, ${this.getAttribute('name')}
 </p>`;
 }
});
</script>

<show-hello name="John"></show-hello>
```

Hello, John

这就是在 Chrome 开发者工具中看到的最终样子，所有的内容都在「#shadow-root」下：

```
▼<show-hello name="John"> == $0
 ▼#shadow-root (open)
 | <p>Hello, John!</p>
 </show-hello>
```

首先，调用 `elem.attachShadow({mode: ...})` 可以创建一个 shadow tree。

这里有两个限制：

- 在每个元素中，我们只能创建一个 shadow root。
- elem 必须是自定义元素，或者是以下元素的其中一个：「article」、「aside」、「blockquote」、「body」、「div」、「footer」、「h1...h6」、「header」、「main」、「nav」、「p」、「section」或者「span」。其他元素，比如 `<img>`，不能容纳 shadow tree。

`mode` 选项可以设定封装层级。他必须是以下两个值之一：

- 「open」 —— shadow root 可以通过 `elem.shadowRoot` 访问。  
任何代码都可以访问 elem 的 shadow tree。
- 「closed」 —— `elem.shadowRoot` 永远是 `null`。

我们只能通过 `attachShadow` 返回的指针来访问 shadow DOM（并且可能隐藏在一个 class 中）。浏览器原生的 shadow tree，比如 `<input type="range">`，是封闭的。没有任何方法可以访问它们。

`attachShadow` 返回的 shadow root ↗，和任何元素一样：我们可以使用 `innerHTML` 或者 DOM 方法，比如 `append` 来扩展它。

我们称有 shadow root 的元素叫做「shadow tree host」，可以通过 shadow root 的 `host` 属性访问：

```
// 假设 {mode: "open"}, 否则 elem.shadowRoot 是 null
alert(elem.shadowRoot.host === elem); // true
```

## 封装

Shadow DOM 被非常明显地和主文档分开：

- Shadow DOM 元素对于 light DOM 中的 `querySelector` 不可见。实际上，Shadow DOM 中的元素可能与 light DOM 中某些元素的 id 冲突。这些元素必须在 shadow tree 中独一无二。
- Shadow DOM 有自己的样式。外部样式规则在 shadow DOM 中不产生作用。

比如：

```
<style>
/* 文档样式对 #elem 内的 shadow tree 无作用 (1) */
p { color: red; }
</style>

<div id="elem"></div>

<script>
```

```

elem.attachShadow({mode: 'open'});
 // shadow tree 有自己的样式 (2)
elem.shadowRoot.innerHTML =
<style> p { font-weight: bold; } </style>
<p>Hello, John!</p>
`;

// <p> 只对 shadow tree 里面的查询可见 (3)
alert(document.querySelectorAll('p').length); // 0
alert(elem.shadowRoot.querySelectorAll('p').length); // 1
</script>

```

1. 文档里面的样式对 shadow tree 没有任何效果。
2. .....但是内部的样式是有效的。
3. 为了获取 shadow tree 内部的元素，我们可以从树的内部查询。

## 参考

- DOM: <https://dom.spec.whatwg.org/#shadow-trees>
- 兼容性: <https://caniuse.com/#feat=shadowdomv1>
- Shadow DOM 在很多其他标准中被提到，比如: [DOM Parsing](#) 指定了 shadow root 有 `innerHTML`。

## 总结

Shadow DOM 是创建组件级别 DOM 的一种方法。

1. `shadowRoot = elem.attachShadow({mode: open|closed})` —— 为 `elem` 创建 shadow DOM。如果 `mode="open"`，那么它通过 `elem.shadowRoot` 属性被访问。
2. 我们可以使用 `innerHTML` 或者其他 DOM 方法来扩展 `shadowRoot`。

Shadow DOM 元素:

- 有自己的 `id` 空间。
- 对主文档的 JavaScript 选择器隐身，比如 `querySelector`。
- 只使用 shadow tree 内部的样式，不使用主文档的样式。

Shadow DOM，如果存在的话，会被浏览器渲染而不是所谓的「light DOM」（普通子元素）。在 [Shadow DOM 插槽](#)，组成 章节中我们将会看到如何组织它们。

## 模板元素

内建的 `<template>` 元素用来存储 HTML 模板。浏览器将忽略它的内容，仅检查语法的有效性，但是我们可以在 JavaScript 中访问和使用它来创建其他元素。

从理论上讲，我们可以在 HTML 中的任何位置创建不可见元素来储存 HTML 模板。那 `<template>` 元素有什么优势？

首先，其内容可以是任何有效的HTML，即使它通常需要特定的封闭标签。

例如，我们可以在其中放置一行表格 `<tr>`：

```
<template>
 <tr>
 <td>Contents</td>
 </tr>
</template>
```

通常，如果我们在 `<tr>` 内放置类似 `<div>` 的元素，浏览器会检测到无效的 DOM 结构并对其进行“修复”，然后用 `<table>` 封闭 `<tr>`，那不是我们想要的。而 `<template>` 则完全保留我们储存的内容。

我们也可以将样式和脚本放入 `<template>` 元素中：

```
<template>
 <style>
 p { font-weight: bold; }
 </style>
 <script>
 alert("Hello");
 </script>
</template>
```

浏览器认为 `<template>` 的内容“不在文档中”：样式不会被应用，脚本也不会被执行，`<video autoplay>` 也不会运行，等。

当我们将内容插入文档时，该内容将变为活动状态（应用样式，运行脚本等）。

## 插入模板

模板的 `content` 属性可看作 [DocumentFragment](#) —— 一种特殊的 DOM 节点。

我们可以将其视为普通的DOM节点，除了它有一个特殊属性：将其插入某个位置时，会被插入的则是其子节点。

例如：

```

<template id="tmpl">
 <script>
 alert("Hello");
 </script>
 <div class="message">Hello, world!</div>
</template>

<script>
 let elem = document.createElement('div');

 // Clone the template content to reuse it multiple times
 elem.append(tmpl.content.cloneNode(true));

 document.body.append(elem);
 // Now the script from <template> runs
</script>

```

让我们用 `<template>` 重写上一章的 Shadow DOM 示例：

```

<template id="tmpl">
 <style> p { font-weight: bold; } </style>
 <p id="message"></p>
</template>

<div id="elem">Click me</div>

<script>
 elem.onclick = function() {
 elem.attachShadow({mode: 'open'});

 elem.shadowRoot.append(tmpl.content.cloneNode(true)); // (*)

 elem.shadowRoot.getElementById('message').innerHTML = "Hello from the shadows!";
 };
</script>

```

Click me

在 (\*) 行，我们将 `tmpl.content` 作为 `DocumentFragment` 克隆和插入，它的子节点（`<style>`，`<p>`）将代为插入。

它们会变成一个 Shadow DOM：

```

<div id="elem">
 #shadow-root
 <style> p { font-weight: bold; } </style>

```

```
<p id="message"></p>
</div>
```

## 总结

总结一下：

- `<template>` 的内容可以是任何语法正确的 HTML。
- `<template>` 内容被视为“超出文档范围”，因此它不会产生任何影响。
- 我们可以在 JavaScript 中访问 `template.content`，将其克隆以在新组件中重复使用。

`<template>` 标签非常独特，因为：

- 浏览器将检查其中的 HTML 语法（与在脚本中使用模板字符串不同）。
- 但允许使用任何顶级 HTML 标签，即使没有适当包装元素的无意义的元素（例如 `<tr>`）。
- 其内容是交互式的：插入其文档后，脚本会运行，`<video autoplay>` 会自动播放。

`<template>` 元素不具有任何迭代机制，数据绑定或变量替换的功能，但我们可以在其基础上实现这些功能。

## Shadow DOM 插槽，组成

许多类型的组件，例如标签、菜单、照片库等等，需要内容去渲染。

就像浏览器内建的 `<select>` 需要 `<option>` 子项，我们的 `<custom-tabs>` 可能需要实际的标签内容来起作用。并且一个 `<custom-menu>` 可能需要菜单子项。

使用了 `<custom-menu>` 的代码如下所示：

```
<custom-menu>
 <title>Candy menu</title>
 <item>Lollipop</item>
 <item>Fruit Toast</item>
 <item>Cup Cake</item>
</custom-menu>
```

.....之后，我们的组件应该正确地渲染成具有给定标题和项目、处理菜单事件等的漂亮菜单。

如何实现呢？

我们可以尝试分析元素内容并动态复制重新排列 DOM 节点。这是可能的，但是如果我们要将元素移动到 Shadow DOM，那么文档的 CSS 样式不能在那里应用，因此文档的视觉样式可能会丢失。看起来还需要做一些事情。

幸运的是我们不需要去做。Shadow DOM 支持 `<slot>` 元素，由 light DOM 中的内容自动填充。

## 具名插槽

让我们通过一个简单的例子看下插槽是如何工作的。

在这里 `<user-card>` shadow DOM 提供两个插槽，从 light DOM 填充：

```
<script>
customElements.define('user-card', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML = `
 <div>Name:
 <slot name="username"></slot>
 </div>
 <div>Birthday:
 <slot name="birthday"></slot>
 </div>
 `;
 }
});
</script>

<user-card>
 John Smith
 01.01.2001
</user-card>
```

Name: John Smith  
Birthday: 01.01.2001

在 shadow DOM 中，`<slot name="X">` 定义了一个“插入点”，一个带有 `slot="X"` 的元素被渲染的地方。

然后浏览器执行“组合”：它从 light DOM 中获取元素并且渲染到 shadow DOM 中的对应插槽中。最后，正是我们想要的——一个能被填充数据的通用组件。

这是编译后，不考虑组合的 DOM 结构：

```
<user-card>
#shadow-root
```

```

<div>Name:
 <slot name="username"></slot>
</div>
<div>Birthday:
 <slot name="birthday"></slot>
</div>
John Smith
01.01.2001
</user-card>

```

我们创建了 shadow DOM，所以它当然就存在了，位于 `#shadow-root` 之下。现在元素同时拥有 light DOM 和 shadow DOM。

为了渲染 shadow DOM 中的每一个 `<slot name="...">` 元素，浏览器在 light DOM 中寻找相同名字的 `slot="..."`。这些元素在插槽内被渲染：

```

<user-card>
 #shadow-root
 <div>Name:
 <slot name="username"></slot>
 </div>
 <div>Birthday:
 <slot name="birthday"></slot>
 </div>
 John Smith
 01.01.2001
 </user-card>

```

结果被叫做扁平化 (flattened) DOM：

```

<user-card>
 #shadow-root
 <div>Name:
 <slot name="username">
 <!-- slotted element is inserted into the slot -->
 John Smith
 </slot>
 </div>
 <div>Birthday:
 <slot name="birthday">
 01.01.2001
 </slot>
 </div>
 </user-card>

```

.....但是“flattened” DOM 仅仅被创建用来渲染和事件处理，是“虚拟”的。虽然是渲染出来了，但文档中的节点事实上并没有到处移动！

如果我们调用 `querySelector` 那就很容易验证：节点仍在它们的位置。

```
// light DOM 节点位置依然不变，在 `<user-card>` 里
alert(document.querySelector('user-card span').length); // 2
```

因此，扁平化 DOM 是通过插入插槽从 shadow DOM 派生出来的。浏览器渲染它并且用于样式继承、事件传播。但是 JavaScript 在展平前仍按原样看到文档。

### ⚠ 仅顶层子元素可以设置 `slot="..."` 特性

`slot="..."` 属性仅仅对 shadow host 的直接子代 (在我们的例子中的 `<user-card>` 元素) 有效。对于嵌套元素它将被忽略。

例如，这里的第二个 `<span>` 被忽略了(因为它不是 `<user-card>` 的顶层子元素):

```
<user-card>
 John Smith
 <div>
 <!-- invalid slot, must be direct child of user-card -->
 01.01.2001
 </div>
</user-card>
```

如果在 light DOM 里有多个相同插槽名的元素，那么它们会被一个接一个地添加到插槽中。

例如这样：

```
<user-card>
 John
 Smith
</user-card>
```

给这个扁平化 DOM 两个元素，插入到 `<slot name="username">` 里：

```
<user-card>
 #shadow-root
 <div>Name:
 <slot name="username">
 John
```

```
Smith
</slot>
</div>
<div>Birthday:
 <slot name="birthday"></slot>
</div>
</user-card>
```

## 插槽后备内容

如果我们在一个 `<slot>` 内部放点什么，它将成为后备内容。如果 light DOM 中没有相应填充物的话浏览器就展示它。

例如，在这里的 shadow DOM 中，如果 light DOM 中没有 `slot="username"` 的话 `Anonymous` 就被渲染。

```
<div>Name:
 <slot name="username">Anonymous</slot>
</div>
```

## 默认插槽：第一个不具名的插槽

shadow DOM 中第一个没有名字的 `<slot>` 是一个默认插槽。它从 light DOM 中获取没有放置在其他位置的所有节点。

例如，让我们把默认插槽添加到 `<user-card>`，该位置可以收集有关用户的所有未开槽（`unslotted`）的信息：

```
<script>
customElements.define('user-card', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML =
`<div>Name:
 <slot name="username"></slot>
</div>
<div>Birthday:
 <slot name="birthday"></slot>
</div>
<fieldset>
 <legend>Other information</legend>
 <slot></slot>
</fieldset>
`;
 }
});
```

```
<user-card>
 <div>I like to swim.</div>
 John Smith
 01.01.2001
 <div>...And play volleyball too!</div>
</user-card>
```

Name: John Smith  
Birthday: 01.01.2001

Other information

I like to swim.  
...And play volleyball too!

所有未被插入的 light DOM 内容进入“其他信息”字段集。

元素一个接一个的附加到插槽中，因此这两个未插入插槽的信息都在默认插槽中。

扁平化的 DOM 看起来像这样：

```
<user-card>
 #shadow-root
 <div>Name:
 <slot name="username">
 John Smith
 </slot>
 </div>
 <div>Birthday:
 <slot name="birthday">
 01.01.2001
 </slot>
 </div>
 <fieldset>
 <legend>About me</legend>
 <slot>
 <div>Hello</div>
 <div>I am John!</div>
 </slot>
 </fieldset>
</user-card>
```

## Menu example

现在让我们回到在本章开头提到的 `<custom-menu>`。

我们可以使用插槽来分配元素。

这是 `<custom-menu>`：

```
<custom-menu>
 Candy menu
 <li slot="item">Lollipop
 <li slot="item">Fruit Toast
 <li slot="item">Cup Cake
</custom-menu>
```

带有适当插槽的 shadow DOM 模版:

```
<template id="tmpl">
 <style> /* menu styles */ </style>
 <div class="menu">
 <slot name="title"></slot>
 <slot name="item"></slot>
 </div>
</template>
```

1. `<span slot="title">` 进入 `<slot name="title">`。
2. 模版中有许多 `<li slot="item">`，但是只有一个 `<slot name="item">`。因此所有带有 `slot="item"` 的元素都一个接一个地附加到 `<slot name="item">` 上，从而形成列表。

扁平化的 DOM 变为:

```
<custom-menu>
 #shadow-root
 <style> /* menu styles */ </style>
 <div class="menu">
 <slot name="title">
 Candy menu
 </slot>

 <slot name="item">
 <li slot="item">Lollipop
 <li slot="item">Fruit Toast
 <li slot="item">Cup Cake
 </slot>

 </div>
</custom-menu>
```

可能会注意到，在有效的 DOM 中，`<li>` 必须是 `<ul>` 的直接子代。但这是扁平化的 DOM，它描述了组件的渲染方式，这样的事情在这里自然发生。

我们只需要添加一个 `click` 事件处理程序来打开/关闭列表，并且 `<custom-menu>` 准备好了：

```
customElements.define('custom-menu', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});

 // tmpl is the shadow DOM template (above)
 this.shadowRoot.append(tmpl.content.cloneNode(true));

 // we can't select light DOM nodes, so let's handle clicks on the slot
 this.shadowRoot.querySelector('slot[name="title"]').onclick = () => {
 // open/close the menu
 this.shadowRoot.querySelector('.menu').classList.toggle('closed');
 };
 }
});
```

这是完整的演示：

## □ Candy menu

Lollipop  
Fruit Toast  
Cup Cake

当然我们可以为它添加更多的功能：事件、方法等。

## 更新插槽

如果外部代码想动态添加/移除菜单项怎么办？

**如果添加/删除了插槽元素，浏览器将监视插槽并更新渲染。**

另外，由于不复制 light DOM 节点，而是仅在插槽中进行渲染，所以内部的变化是立即可见的。

因此我们无需执行任何操作即可更新渲染。但是如果组件想知道插槽的更改，那么可以用 `slotchange` 事件。

例如，这里的菜单项在 1 秒后动态插入，而且标题在 2 秒后改变。

```
<custom-menu id="menu">
 Candy menu
</custom-menu>
```

```

<script>
customElements.define('custom-menu', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML = `<div class="menu">
 <slot name="title"></slot>
 <slot name="item"></slot>
 </div>`;

 // shadowRoot can't have event handlers, so using the first child
 this.shadowRoot.firstChild.addEventListener('slotchange',
 e => alert("slotchange: " + e.target.name)
);
 }
});

setTimeout(() => {
 menu.insertAdjacentHTML('beforeEnd', '<li slot="item">Lollipop')
}, 1000);

setTimeout(() => {
 menu.querySelector('[slot="title"]').innerHTML = "New menu";
}, 2000);
</script>

```

菜单每次都会更新渲染而无需我们干预。

这里有两个 `slotchange` 事件:

1. 在初始化时:

`slotchange: title` 立即触发, 因为来自 light DOM 的 `slot="title"` 进入了相应的插槽。

2. 1 秒后:

`slotchange: item` 触发, 当一个新的 `<li slot="item">` 被添加。

请注意: 2 秒后, 如果修改了 `slot="title"` 的内容, 则不会发生 `slotchange` 事件。因为没有插槽更改。我们修改了 `slotted` 元素的内容, 这是另一回事。

如果我们想通过 JavaScript 跟踪 light DOM 的内部修改, 也可以使用更通用的机制: [MutationObserver](#)。

## 插槽 API

最后让我们来谈谈与插槽相关的 JavaScript 方法。

正如我们之前所见, JavaScript 会查看真实的 DOM, 不展开。但是如果 shadow 树有 `{mode: 'open'}`, 那么我们可以找出哪个元素被放进一个插槽, 反之亦然, 哪个插槽分配了给这个元素:

- `node.assignedSlot` – 返回 `node` 分配给的 `<slot>` 元素。
- `slot.assignedNodes({flatten: true/false})` – 分配给插槽的 DOM 节点。默认情况下，`flatten` 选项为 `false`。如果显式地设置为 `true`，则它将更深入地查看扁平化 DOM，如果嵌套了组件，则返回嵌套的插槽，如果未分配节点，则返回备用内容。
- `slot.assignedElements({flatten: true/false})` – 分配给插槽的 DOM 元素（与上面相同，但仅元素节点）。

当我们不仅需要显示已插入内容的内容，还需要在 JavaScript 中对其进行跟踪时，这些方法非常有用。

例如，如果 `<custom-menu>` 组件想知道它所显示的内容，那么它可以跟踪 `slotchange` 并从 `slot.assignedElements` 获取：

```

<custom-menu id="menu">
 Candy menu
 <li slot="item">Lollipop
 <li slot="item">Fruit Toast
</custom-menu>

<script>
customElements.define('custom-menu', class extends HTMLElement {
 items = []

 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML = `<div class="menu">
 <slot name="title"></slot>
 <slot name="item"></slot>
 </div>`;

 // 插槽能被添加/删除/代替
 this.shadowRoot.firstChild.addEventListener('slotchange', e => {
 let slot = e.target;
 if (slot.name == 'item') {
 this.items = slot.assignedElements().map(elem => elem.textContent);
 alert("Items: " + this.items);
 }
 });
 }
});

// items 在 1 秒后更新
setTimeout(() => {
 menu.insertAdjacentHTML('beforeEnd', '<li slot="item">Cup Cake');
}, 1000);
</script>

```

## 小结

通常，如果一个元素含有 shadow DOM，那么其 light DOM 就不会被展示出来。插槽允许在 shadow DOM 中显示 light DOM 子元素。

插槽有两种：

- 具名插槽：`<slot name="X">...</slot>` – 使用 `slot="X"` 获取 light 子元素。
- 默认插槽：第一个没有名字的 `<slot>`（随后的未命名插槽将被忽略） - 接受不是插槽的 light 子元素。
- 如果同一插槽中有很多元素 – 它们会被一个接一个地添加。
- `<slot>` 元素的内容作为备用。如果插槽没有 light 型的子元素，就会显示。

在其槽内渲染插槽元素的过程称为“组合”。结果称为“扁平化 DOM”。

组合不会真实的去移动节点，从 JavaScript 的视角看 DOM 仍然是相同的。

JavaScript 可以使用以下的方法访问插槽：

- `slot.assignedNodes/Elements()` – 返回插槽内的 节点/元素。
- `node.assignedSlot` – 相反的方法，返回一个节点的插槽。

如果我们想知道显示的内容，可以使用以下方法跟踪插槽位的内容：

- `slotchange` 事件 – 在插槽第一次填充时触发，并且在插槽元素的 添加/删除/替换 操作（而不是其子元素）时触发，插槽是 `event.target`。
- 使用 [MutationObserver](#) 来深入了解插槽内容，并查看其中的更改。

现在，在 shadow DOM 中有来自 light DOM 的元素时，让我们看看如何正确的设置样式。基本规则是 shadow 元素在内部设置样式，light 元素在外部设置样式，但是有一些例外。

我们将在下一章中看到详细内容。

## 给 Shadow DOM 添加样式

shadow DOM 可以包含 `<style>` 和 `<link rel="stylesheet" href="...">` 标签。在后一种情况下，样式表是 HTTP 缓存的，因此不会为使用同一模板的多个组件重新下载样式表。

一般来说，局部样式只在 shadow 树内起作用，文档样式在 shadow 树外起作用。但也有少数例外。

### :host

`:host` 选择器允许选择 shadow 宿主（包含 shadow 树的元素）。

例如，我们正在创建 `<custom-dialog>` 元素，并且想使它居中。为此，我们需要对 `<custom-dialog>` 元素本身设置样式。

这正是 `:host` 所能做的：

```
<template id="tmpl">
<style>
 /* 这些样式将从内部应用到 custom-dialog 元素上 */
 :host {
 position: fixed;
 left: 50%;
 top: 50%;
 transform: translate(-50%, -50%);
 display: inline-block;
 border: 1px solid red;
 padding: 10px;
 }
</style>
<slot></slot>
</template>

<script>
customElements.define('custom-dialog', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'}).append(tmpl.content.cloneNode(true));
 }
});
</script>

<custom-dialog>
 Hello!
</custom-dialog>
```



Hello!

## 级联

shadow 宿主（`<custom-dialog>` 本身）驻留在 light DOM 中，因此它受到文档 CSS 规则的影响。

如果在局部的 `:host` 和文档中都给一个属性设置样式，那么文档样式优先。

例如，如果在文档中我们有如下样式：

```
<style>
 custom-dialog {
 padding: 0;
```

```
}
```

```
</style>
```

.....那么 `<custom-dialog>` 将没有 padding。

这是非常有利的，因为我们可以在其 `:host` 规则中设置“默认”组件样式，然后在文档中轻松地覆盖它们。

唯一的例外是当局部属性被标记 `!important` 时，对于这样的属性，局部样式优先。

## :host(selector)

与 `:host` 相同，但仅在 shadow 宿主与 `selector` 匹配时才应用样式。

例如，我们希望仅当 `<custom-dialog>` 具有 `centered` 属性时才将其居中：

```
<template id="tmpl">
<style>
:host([centered]) {
 position: fixed;
 left: 50%;
 top: 50%;
 transform: translate(-50%, -50%);
 border-color: blue;
}

:host {
 display: inline-block;
 border: 1px solid red;
 padding: 10px;
}
</style>
<slot></slot>
</template>
```

```
<script>
customElements.define('custom-dialog', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'}).append(tmpl.content.cloneNode(true));
 }
});
</script>
```

```
<custom-dialog centered>
 Centered!
</custom-dialog>
```

```
<custom-dialog>
```

```
Not centered.
</custom-dialog>
```

Not centered.

Centered!

现在附加的居中样式只应用于第一个对话框: `<custom-dialog centered>`。

## :host-context(selector)

与 `:host` 相同, 但仅当外部文档中的 `shadow` 宿主或它的任何祖先节点与 `selector` 匹配时才应用样式。

例如, `:host-context(.dark-theme)` 只有在 `<custom-dialog>` 或者 `<custom-dialog>` 的任何祖先节点上有 `dark-theme` 类时才匹配:

```
<body class="dark-theme">
 <!--
 :host-context(.dark-theme) 只应用于 .dark-theme 内部的 custom-dialog
 -->
 <custom-dialog>...</custom-dialog>
</body>
```

总之, 我们可以使用 `:host -family` 系列的选择器来对组件的主元素进行样式设置, 具体取决于上下文。这些样式 (除 `!important` 外) 可以被文档样式覆盖。

## 给占槽（slotted）内容添加样式

现在让我们考虑有插槽的情况。

占槽元素来自 `light DOM`, 所以它们使用文档样式。局部样式不会影响占槽内容。

在下面的例子中, 按照文档样式, 占槽的 `<span>` 是粗体, 但是它不从局部样式中获取 `background`:

```
<style>
 span { font-weight: bold }
</style>

<user-card>
 <div slot="username">John Smith</div>
</user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
```

```
connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML =
 `<style>
 span { background: red; }
 </style>
 Name: <slot name="username"></slot>
 `;
}
});
</script>
```

Name:  
**John Smith**

结果是粗体，但不是红色。

如果我们想要在我们的组件中设置占槽元素的样式，有两种选择。

首先，我们可以对 `<slot>` 本身进行样式化，并借助 CSS 继承：

```
<user-card>
 <div slot="username">John Smith</div>
</user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML =
 `<style>
 slot[name="username"] { font-weight: bold; }
 </style>
 Name: <slot name="username"></slot>
 `;
 }
});
</script>
```

Name:  
**John Smith**

这里 `<p>John Smith</p>` 变成粗体，因为 CSS 继承在 `<slot>` 和它的内容之间有效。但是在 CSS 中，并不是所有的属性都是继承的。

另一个选项是使用 `::slotted(selector)` 伪类。它根据两个条件来匹配元素：

1. 这是一个占槽元素，来自于 light DOM。插槽名并不重要，任何占槽元素都可以，但只能是元素本身，而不是它的子元素。
2. 该元素与 selector 匹配。

在我们的例子中，`::slotted(div)` 正好选择了 `<div slot="username">`，但是没有选择它的子元素：

```
<user-card>
 <div slot="username">
 <div>John Smith</div>
 </div>
</user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML = `
 <style>
 ::slotted(div) { border: 1px solid red; }
 </style>
 Name: <slot name="username"></slot>
 `;
 }
});
</script>
```

Name:  
John Smith

请注意，`::slotted` 选择器不能用于任何插槽中更深层的内容。下面这些选择器是无效的：

```
::slotted(div span) {
 /* 我们插入的 <div> 不会匹配这个选择器 */
}

::slotted(div) p {
 /* 不能进入 light DOM 中选择元素 */
}
```

此外，`::sloated` 只能在 CSS 中使用，不能在 `querySelector` 中使用。

## 用自定义 CSS 属性作为勾子

如何在主文档中设置组件的内建元素的样式？

像 `:host` 这样的选择器应用规则到 `<custom-dialog>` 元素或 `<user-card>`，但是如何设置在它们内部的 shadow DOM 元素的样式呢？

没有选择器可以从文档中直接影响 shadow DOM 样式。但是，正如我们暴露用来与组件交互的方法那样，我们也可以暴露 CSS 变量（自定义 CSS 属性）来对其进行样式设置。

自定义 CSS 属性存在于所有层次，包括 light DOM 和 shadow DOM。

例如，在 shadow DOM 中，我们可以使用 `--user-card-field-color` CSS 变量来设置字段的样式，而外部文档可以设置它的值：

```
<style>
 .field {
 color: var(--user-card-field-color, black);
 /* 如果 --user-card-field-color 没有被声明过，则取值为 black */
 }
</style>
<div class="field">Name: <slot name="username"></slot></div>
<div class="field">Birthday: <slot name="birthday"></slot></div>
</style>
```

然后，我们可以在外部文档中为 `<user-card>` 声明此属性：

```
user-card {
 --user-card-field-color: green;
}
```

自定义 CSS 属性穿透 shadow DOM，它们在任何地方都可见，因此内部的 `.field` 规则将使用它。

以下是完整的示例：

```
<style>
 user-card {
 --user-card-field-color: green;
 }
</style>

<template id="tmpl">
 <style>
 .field {
 color: var(--user-card-field-color, black);
 }
 </style>
```

```

<div class="field">Name: <slot name="username"></slot></div>
<div class="field">Birthday: <slot name="birthday"></slot></div>
</template>

<script>
customElements.define('user-card', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.append(document.getElementById('tmp1').content.cloneNode(true));
 }
});
</script>

<user-card>
 John Smith
 01.01.2001
</user-card>

```

Name: John Smith  
 Birthday: 01.01.2001

## 小结

shadow DOM 可以引入样式，如 `<style>` 或 `<link rel="stylesheet">`。

局部样式可以影响：

- shadow 树，
- shadow 宿主（通过 `:host -family` 系列伪类），
- 占槽元素（来自 light DOM），`::slotted(selector)` 允许选择占槽元素本身，但不能选择它们的子元素。

文档样式可以影响：

- shadow 宿主（因为它位于外部文档中）
- 占槽元素及占槽元素的内容（因为它们同样位于外部文档中）

当 CSS 属性冲突时，通常文档样式具有优先级，除非属性被标记为 `!important`，那么局部样式优先。

CSS 自定义属性穿透 shadow DOM。它们被用作“勾子”来设计组件的样式：

1. 组件使用自定义 CSS 属性对关键元素进行样式设置，比如 `var(--component-name-title, <default value>)`。
2. 组件作者为开发人员发布这些属性，它们和其他公共的组件方法一样重要。
3. 当开发人员想要对一个标题进行样式设计时，他们会为 shadow 宿主或宿主上层的元素赋值 `--component-name-title` CSS 属性。

## 4. 奥力给！

### Shadow DOM 和事件 (events)

Shadow tree 背后的思想是封装组件的内部实现细节。

假设，在 `<user-card>` 组件的 shadow DOM 内触发一个点击事件。但是主文档内部的脚本并不了解 shadow DOM 内部，尤其是当组件来自于第三方库。

所以，为了保持细节简单，浏览器会\*重新定位（retargets）\*事件。

当事件在组件外部捕获时，shadow DOM 中发生的事件将会以 host 元素作为目标。

这里有个简单的例子：

```
<user-card></user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML = `<p>
 <button>Click me</button>
 </p>`;
 this.shadowRoot.firstChild.onclick =
 e => alert("Inner target: " + e.target.tagName);
 }
});

document.onclick =
 e => alert("Outer target: " + e.target.tagName);
</script>
```

Click me

如果你点击了 button，就会出现以下信息：

1. Inner target: BUTTON —— 内部事件处理程序获取了正确的目标，即 shadow DOM 中的元素。
2. Outer target: USER-CARD —— 文档事件处理程序以 shadow host 作为目标。

事件重定向是一件很棒的事情，因为外部文档并不需要知道组件的内部情况。从它的角度来看，事件是发生在 `<user-card>`。

如果事件发生在 slotted 元素上，实际存在于轻型 (light) DOM 上，则不会发生重定向。

例如，在下面的例子中，如果用户点击了 `<span slot="username">`，那么对于 shadow 和 light 处理程序来说，事件目标就是当前这个 `span` 元素。

```
<user-card id="userCard">
 John Smith
</user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
 connectedCallback() {
 this.attachShadow({mode: 'open'});
 this.shadowRoot.innerHTML = `<div>
 Name: <slot name="username"></slot>
 </div>`;
 this.shadowRoot.firstChild.onclick =
 e => alert("Inner target: " + e.target.tagName);
 }
});

userCard.onclick = e => alert(`Outer target: ${e.target.tagName}`);
</script>
```

**Name:** John Smith

如果单击事件发生在 `"John Smith"` 上，则对于内部和外部处理程序来说，其目标是 `<span slot="username">`。这是 light DOM 中的元素，所以没有重定向。

另一方面，如果单击事件发生在源自 shadow DOM 的元素上，例如，在 `<b>Name</b>` 上，然后当它冒泡出 shadow DOM 后，其 `event.target` 将重置为 `<user-card>`。

## 冒泡（bubbling）, `event.composedPath()`

出于事件冒泡的目的，使用扁平 DOM（flattened DOM）。

所以，如果我们有一个 `slot` 元素，并且事件发生在它的内部某个地方，那么它就会冒泡到 `<slot>` 并继续向上。

使用 `event.composedPath()` 获得原始事件目标的完整路径以及所有 shadow 元素。正如我们从方法名称中看到的那样，该路径是在组合（composition）之后获取的。

在上面的例子中，扁平 DOM 是：

```
<user-card id="userCard">
#shadow-root
```

```
<div>
 Name:
 <slot name="username">
 John Smith
 </slot>
</div>
</user-card>
```

因此，对于 `<span slot="username">` 上的点击事件，会调用 `event.composedPath()` 并返回一个数组：`[span, slot, div, shadow-root, user-card, body, html, document, window]`。在组合之后，这正是扁平 DOM 中目标元素的父链。

### ⚠️ Shadow 树详细信息仅提供给 `{mode: 'open'}` 树

如果 shadow 树是用 `{mode: 'closed'}` 创建的，那么组合路径就从 host 开始：`user-card` 及其更上层。

这与使用 shadow DOM 的其他方法的原理类似。`closed` 树内部是完全隐藏的。

## event.composed

大多数事件能成功冒泡到 shadow DOM 边界。很少有事件不能冒泡到 shadow DOM 边界。

这由 `composed` 事件对象属性控制。如果 `composed` 是 `true`，那么事件就能穿过边界。否则它仅能在 shadow DOM 内部捕获。

如果你浏览一下 [UI 事件规范](#) 就知道，大部分事件都是 `composed: true`：

- `blur`, `focus`, `focusin`, `focusout`,
- `click`, `dblclick`,
- `mousedown`, `mouseup` `mousemove`, `mouseout`, `mouseover`,
- `wheel`,
- `beforeinput`, `input`, `keydown`, `keyup`。

所有触摸事件（touch events）及指针事件（pointer events）都是 `composed: true`。

但也有些事件是 `composed: false` 的：

- `mouseenter`, `mouseleave`（它们根本不会冒泡），
- `load`, `unload`, `abort`, `error`,
- `select`,
- `slotchange`。

这些事件仅能在事件目标所在的同一 DOM 中的元素上捕获,

## 自定义事件 (Custom events)

当我们发送 (dispatch) 自定义事件, 我们需要设置 `bubbles` 和 `composed` 属性都为 `true` 以使其冒泡并从组件中冒泡出来。

例如, 我们在 `div#outer` shadow DOM 内部创建 `div#inner` 并在其上触发两个事件。只有 `composed: true` 的那个自定义事件才会让该事件本身冒泡到文档外面:

```
<div id="outer"></div>

<script>
outer.attachShadow({mode: 'open'});

let inner = document.createElement('div');
outer.shadowRoot.append(inner);

/*
div(id=outer)
 #shadow-dom
 div(id=inner)
*/

document.addEventListener('test', event => alert(event.detail));

inner.dispatchEvent(new CustomEvent('test', {
 bubbles: true,
 composed: true,
 detail: "composed"
}));

inner.dispatchEvent(new CustomEvent('test', {
 bubbles: true,
 composed: false,
 detail: "not composed"
});
</script>
```

## 总结

事件仅仅是在它们的 `composed` 标志设置为 `true` 的时候才能通过 shadow DOM 边界。

内建事件大部分都是 `composed: true` 的, 正如相关规范所描述的那样:

- UI 事件 <https://www.w3.org/TR/uievents>。

- Touch 事件 <https://w3c.github.io/touch-events>。
- Pointer 事件 <https://www.w3.org/TR/pointerevents>。
- .....等等。

也有些内建事件它们是 `composed: false` 的:

- `mouseenter`, `mouseleave` (也不冒泡) ,
- `load`, `unload`, `abort`, `error`,
- `select`,
- `slotchange`。

这些事件仅能在同一 DOM 中的元素上捕获。

如果我们发送一个 `CustomEvent`，那么我们应该显示设置 `composed: true`。

请注意，如果是嵌套组件，一个 shadow DOM 可能嵌套到另外一个 shadow DOM 中。在这种情况下合成事件冒泡到所有 shadow DOM 边界。因此，如果一个事件仅用于直接封闭组件，我们也可以在 shadow host 上发送它并设置 `composed: false`。这样它就不在组件 shadow DOM 中，也不会冒泡到更高级别的 DOM。

## 正则表达式

正则表达式是一个查找和替换字符串的强有力的工具。

## 模式（Patterns）和修饰符（flags）

正则表达式是搜索和替换字符串的一种强大方式。

在 JavaScript 中，正则表达式通过内置的“`RegExp`”类的对象来实现，并与字符串集成。

请注意，在各编程语言之间，正则表达式是有所不同的。在本教程中，我们只专注于 JavaScript。当然，它们有很多共同点，但在 Perl、Ruby 和 PHP 等语言下会有所不同。

## 正则表达式

正则表达式（可叫作“`regexp`”或者“`reg`”）包含 模式 和可选的 修饰符。

创建一个正则表达式对象有两种语法。

较长一点的语法:

```
regexp = new RegExp("pattern", "flags");
```

...较短一点的语法，使用斜杠 `"/"`：

```
regexp = /pattern/; // 没有修饰符
regexp = /pattern/gmi; // 伴随修饰符 g、m 和 i (后面会讲到)
```

斜杠 `"/"` 会告诉 JavaScript 我们正在创建一个正则表达式。它的作用类似于字符串的引号。

## 用法

如果要在字符串中进行搜索，可以使用 `search ↗` 方法。

下面是示例：

```
let str = "I love JavaScript!"; // 将在这里搜索

let regexp = /love/;
alert(str.search(regexp)); // 2
```

`str.search` 方法会查找模式 `/love/`，然后返回匹配项在字符串中的位置。我们可以猜到，`/love/` 是最简单的模式。它所做的就是简单的子字符串的查找。

上面的代码等同于：

```
let str = "I love JavaScript!"; // 将在这里搜索

let substr = 'love';
alert(str.search(substr)); // 2
```

所以搜索 `/love/` 与搜索 `"love"` 是等价的。

但这只是暂时的。很快我们就会接触更复杂的正则表达式，其搜索功能将更强大。

### ➊ 配色

本文中的配色方案如下：

- `regexp` – `red`
- `string` (我们要搜索的) -- `blue`
- `result` – `green`

## i 什么时候使用 new RegExp ?

通常我们使用的都是简短语法 `/.../`。但是它不接受任何变量插入，所以我们必须在写代码的时候就知道确切的 regexp。

另一方面，`new RegExp` 允许从字符串中动态地构造模式。

所以我们可以找出需要搜索的字段，然后根据搜索字段创建 `new RegExp`：

```
let search = prompt("What you want to search?", "love");
let regexp = new RegExp(search);

// 找到用户想要的任何东西
alert("I love JavaScript".search(regexp));
```

## 修饰符

正则表达式的修饰符可能会影响搜索结果。

在 JavaScript 中，有 5 个修饰符：

**i**

使用此修饰符后，搜索时不区分大小写：`A` 和 `a` 没有区别（具体看下面的例子）。

**g**

使用此修饰符后，搜索时会查找所有的匹配项，而不只是第一个（在下一章会讲到）。

**m**

多行模式（详见章节 文章 "regexp-multiline" 未找到）。

**u**

开启完整的 `unicode` 支持。该修饰符能够修正对于代理对的处理。更详细的内容见章节 [Unicode：修饰符“u”和 class \p{...}](#)。

**y**

粘滞模式（详见 [下一章节](#)）

## “i”修饰符

最简单的修饰符就是 `i` 了。

示例代码如下：

```
let str = "I love JavaScript!";

alert(str.search(/LOVE/)); // -1 (没找到)
alert(str.search(/LOVE/i)); // 2
```

1. 第一个搜索返回的是 `-1` (也就是没找到)，因为搜索默认是区分大小写的。
2. 使用修饰符 `/LOVE/i`，在字符串的第 `2` 个位置上搜索到了 `love`。

相比与简单的子字符串查找，`i` 修饰符已经让正则表达式变得更加强大了。但是这还不够。我们会在下一章节讲述其它修饰符和特性。

## 总结

- 一个正则表达式包含模式和可选修饰符：`g`、`i`、`m`、`u`、`y`。
- 如果不使用我们在后面将要学到的修饰符和特殊标志，正则表达式的搜索就等同于子字符串查找。
- `str.search(regexp)` 方法返回的是找到的匹配项的索引位置，如果没找到则返回 `-1`。

## 字符类

考虑一个实际的任务 – 我们有一个电话号码，例如 `"+7(903)-123-45-67"`，我们需要将其转换为纯数字：`79035419441`。

为此，我们可以查找并删除所有非数字的内容。字符类可以帮助解决这个问题。

**字符类 (Character classes)** 是一个特殊的符号，匹配特定集中的任何符号。

首先，让我们探索“数字”类。它写为 `\d`，对应于“任何一个数字”。

例如，让我们找到电话号码的第一个数字：

```
let str = "+7(903)-123-45-67";

let regexp = /\d/;

alert(str.match(regexp)); // 7
```

如果没有标志 `g`，则正则表达式仅查找第一个匹配项，即第一个数字 `\d`。

让我们添加 `g` 标志来查找所有数字：

```
let str = "+7(903)-123-45-67";
```

```
let regexp = /\d/g;

alert(str.match(regexp)); // array of matches: 7,9,0,3,1,2,3,4,5,6,7

// let's make the digits-only phone number of them:
alert(str.match(regexp).join('')); // 79035419441
```

这是数字的字符类。还有其他字符类。

最常用的是：

### \d (“d” 来自 “digit”)

数字：从 **0** 到 **9** 的字符。

### \s (“s” 来自 “space”)

空格符号：包括空格，制表符 **\t**，换行符 **\n** 和其他少数稀有字符，例如 **\v**，**\f** 和 **\r**。

### \w (“w” 来自 “word”)

“单字”字符：拉丁字母或数字或下划线 **\_**。非拉丁字母（如西里尔字母或印地文）不属于 \w。

例如，\d\s\w 表示“数字”，后跟“空格字符”，后跟“单字字符”，例如 **1 a**。

正则表达式可能同时包含常规符号和字符类。

例如，CSS\d 匹配字符串 **CSS** 与后面的数字：

```
let str = "Is there CSS4?";
let regexp = /CSS\d/

alert(str.match(regexp)); // CSS4
```

我们还可以使用许多字符类：

```
alert("I love HTML5!".match(/\s\w\w\w\w\d/)); // ' HTML5'
```

匹配项（每个正则表达式字符类都有对应的结果字符）：

I **l** **o** **v** **e** H**T****M****L** 5

## 反向类

对于每个字符类，都有一个“反向类”，用相同的字母表示，但要以大写书写形式。

“反向”表示它与所有其他字符匹配，例如：

### \D

非数字：除 \d 以外的任何字符，例如字母。

### \S

非空格符号：除 \s 以外的任何字符，例如字母。

### \W

非单字字符：除 \w 以外的任何字符，例如非拉丁字母或空格。

在这一章的开头，我们看到了如何从 +7(903)-123-45-67 这样的字符串中创建一个只包含数字的电话号码：找到所有的数字并将它们连接起来。

```
let str = "+7(903)-123-45-67";

alert(str.match(/\d/g).join('')); // 79031234567
```

另一种快捷的替代方法是查找非数字 \D 并将其从字符串中删除：

```
let str = "+7(903)-123-45-67";

alert(str.replace(/\D/g, '')); // 79031234567
```

## 点 (.) 是匹配“任何字符”

点 . 是一种特殊字符类，它与“除换行符之外的任何字符”匹配。

例如：

```
alert("Z".match(/./)); // Z
```

或在正则表达式中间：

```
let regexp = /CS.4/;

alert("CSS4".match(regexp)); // CSS4
alert("CS-4".match(regexp)); // CS-4
alert("CS 4".match(regexp)); // CS 4 (space is also a character)
```

请注意，点表示“任何字符”，而不是“缺少字符”。必须有一个与之匹配的字符：

```
alert("CS4".match(/CS.4/)); // null, no match because there's no character for the .
```

## 带有“s”标志时点字符类严格匹配任何字符

默认情况下，点与换行符 `\n` 不匹配。

例如，正则表达式 `A.B` 匹配 `A`，然后匹配 `B` 和它们之间的任何字符，除了换行符 `\n`：

```
alert("A\nB".match(/A.B/)); // null (no match)
```

在许多情况下，当我们希望用点来表示“任何字符”（包括换行符）时。

这就是标志 `s` 所做的。如果有一个正则表达式，则点 `.` 实际上匹配任何字符：

```
alert("A\nB".match(/A.B/s)); // A\nB (match!)
```

### ⚠ 不支持 Firefox、IE、Edge

使用前可从 <https://caniuse.com/#search=dotall> 确认以获得最新的支持状态。  
在撰写本文时，它不包括 Firefox、IE、Edge。

幸运的是，有一种替代方法可以在任何地方使用。我们可以使用诸如 `[\s\S]` 之类的正则表达式来匹配“任何字符”。

```
alert("A\nB".match(/A[\s\S]B/)); // A\nB (match!)
```

模式 `[\s\S]` 从字面上说：“空格字符或非空格字符”。换句话说，“任何东西”。我们可以使用另一对互补的类，例如 `[\d\D]`。甚至是 `[^]` —— 意思是匹配任何字符，除了什么都没有。

如果我们希望两种“点”都使用相同的模式，也可以使用此技巧：实际的点 `.` 具有常规方式（“不包括换行符”）以及一种使用 `[\s\S]` 或类似形式匹配“任何字符”。

## ⚠ 注意空格

通常我们很少注意空格。对我们来说，字符串 `1-5` 和 `1 - 5` 几乎相同。

但是，如果正则表达式未考虑空格，则可能无法正常工作。

让我们尝试查找由连字符（-）分隔的数字：

```
alert("1 - 5".match(/\d-\d/)); // null, no match!
```

让我们修复一下，在正则表达式中添加空格：`\d-\d``：

```
alert("1 - 5".match(/\d - \d/)); // 1 - 5, now it works
// or we can use \s class:
alert("1 - 5".match(/\d\s-\s\d/)); // 1 - 5, also works
```

空格是一个字符。与其他字符同等重要。

我们无法在正则表达式中添加或删除空格，并且期望能正常工作。

换句话说，在正则表达式中，所有字符都很重要，空格也很重要。

## 总结

存在以下字符类：

- `\d` —— 数字。
- `\D` —— 非数字。
- `\s` —— 空格符号，制表符，换行符。
- `\S` —— 除了 `\s`。
- `\w` —— 拉丁字母，数字，下划线 `'_'`。
- `\W` —— 除了 `\w`。
- `.` —— 任何带有 `'s'` 标志的字符，否则为除换行符 `\n` 之外的任何字符。

.....但这还不是全部！

JavaScript 用于字符串的 **Unicode** 编码提供了许多字符属性，例如：这个字母属于哪一种语言（如果它是一个字母）？它是标点符号吗？等等。

我们也可以通过这些属性进行搜索。这需要标志 `u`，在下一篇文章中介绍。

## Unicode：修饰符“u”和 class `\p{...}`

JavaScript 使用 [Unicode 编码](#)（[Unicode encoding](#)）对字符串进行编码。大多数字符使用 2 个字节编码，但这种方式只能编码最多 65536 个字符。

这个范围不足以对所有可能的字符进行编码，这就是为什么一些罕见的字符使用 4 个字节进行编码，比如  $\chi$ （数学符号 X）或者  $\circledcirc$ （笑脸），一些象形文字等等。

以下是一些字符对应的 `unicode` 编码：

字符	Unicode	unicode 中的字节数
a	0x0061	2
=	0x2248	2
$\chi$	0x1d4b3	4
$\gamma$	0x1d4b4	4
$\circledcirc$	0x1f604	4

所以像 `a` 和 `=` 这样的字符占用 2 个字节，而  $\chi$ ， $\gamma$  和  $\circledcirc$  的对应编码则更长，它们具有 4 个字节的长度。

很久以前，当 JavaScript 被发明出来的时候，`Unicode` 的编码要更加简单：当时并没有 4 个字节长的字符。所以，一部分语言特性在现在仍旧无法对 `unicode` 进行正确的处理。

比如 `length` 认为这里的输入有 2 个字符：

```
alert('☺'.length); // 2
alert('χ'.length); // 2
```

...但我们可以清楚地认识到输入的字符只有一个，对吧？关键在于 `length` 把 4 个字节当成了 2 个 2 字节长的字符。这是不对的，因为它们必须被当作一个整体来考虑。（即所谓的“代理伪字符”（surrogate pair），你可以在这里进一步阅读有关的信息 [字符串](#)）。

默认情况下，正则表达式同样把一个 4 个字节的“长字符”当成一对 2 个字节长的字符。正如在字符串中遇到的情况，这将导致一些奇怪的结果。我们将很快在后面的文章中遇到 [集合和范围](#) [...]。

与字符串有所不同的是，正则表达式有一个修饰符 `u` 被用以解决此类问题。当一个正则表达式使用这个修饰符后，4 个字节长的字符将被正确地处理。同时也能够用上 `Unicode 属性`（`Unicode property`）来进行查找了。我们接下来就来了解这方面的内容。

## Unicode 属性（`Unicode properties`）\p{...}

## ⚠ 在 Firefox 和 Edge 中缺乏支持

尽管 `unicode` property 从 2018 年以来便作为标准的一部分，但 `unicode` 属性在 Firefox ([bug ↗](#)) 和 Edge ([bug ↗](#)) 中并没有相应的支持。

目前 [XRegExp ↗](#) 这个库提供“扩展”的正则表达式，其中包括对 `unicode` property 的跨平台支持。

`Unicode` 中的每一个字符都具有很多的属性。它们描述了一个字符属于哪个“类别”，包含了各种关于字符的信息。

例如，如果一个字符具有 `Letter` 属性，这意味着这个字符归属于（任意语言的）一个字母表。而 `Number` 属性则表示这是一个数字：也许是阿拉伯语，亦或者是中文，等等。

我们可以查找具有某种属性的字符，写作 `\p{...}`。为了顺利使用 `\p{...}`，一个正则表达式必须使用修饰符 `u`。

举个例子，`\p{Letter}` 表示任何语言中的一个字母。我们也可以使用 `\p{L}`，因为 `L` 是 `Letter` 的一个别名（alias）。对于每种属性而言，几乎都存在对应的缩写别名。

在下面的例子中 3 种字母将会被查找出：英语、格鲁吉亚语和韩语。

```
let str = "A ბ ㅁ";

alert(str.match(/\p{L}/gu)); // A, ბ, ㅁ
alert(str.match(/\p{L}/g)); // null (没有匹配的文本，因为没有修饰符"u")
```

以下是主要的字符类别和它们对应的子类别：

- 字母（`Letter`） `L`：
  - 小写（`lowercase`） `Ll`
  - 修饰（`modifier`） `Lm`,
  - 首字母大写（`titlecase`） `Lt`,
  - 大写（`uppercase`） `Lu`,
  - 其它（`other`） `Lo`。
- 数字（`Number`） `N`：
  - 十进制数字（`decimal digit`） `Nd`,
  - 字母数字（`letter number`） `Nl`,
  - 其它（`other`） `No`。
- 标点符号（`Punctuation`） `P`：
  - 链接符（`connector`） `Pc`,

- 横杠 (dash) `Pd`,
- 起始引用号 (initial quote) `Pi`,
- 结束引用号 (final quote) `Pf`,
- 开 (open) `Ps`,
- 闭 (close) `Pe`,
- 其它 (other) `Po`。
- 标记 (Mark) `M` (accents etc):
  - 间隔合并 (spacing combining) `Mc`,
  - 封闭 (enclosing) `Me`,
  - 非间隔 (non-spacing) `Mn`。
- 符号 (Symbol) `S`:
  - 货币 (currency) `Sc`,
  - 修饰 (modifier) `Sk`,
  - 数学 (math) `Sm`,
  - 其它 (other) `So`。
- 分隔符 (Separator) `Z`:
  - 行 (line) `Zl`,
  - 段落 (paragraph) `Zp`,
  - 空格 (space) `Zs`。
- 其它 (Other) `C`:
  - 控制符 (control) `Cc`,
  - 格式 (format) `Cf`,
  - 未分配 (not assigned) `Cn`,
  - 私有 (private use) `Co`,
  - 代理伪字符 (surrogate) `Cs`。

因此，比如说我们需要小写的字母，就可以写成 `\p{Ll}`，标点符号写作 `\p{P}` 等等。

也有其它派生的类别，例如：

- `Alphabetic` (`Alpha`)，包含了字母 `L`，加上字母数字 `Nl`（例如 `XII` – 罗马数字 12），加上一些其它符号 `Other_Alphabetic` (`OAlpha`)。
- `Hex_Digit` 包括 16 进制数字 `0-9`, `a-f`。
- ...等等

`Unicode` 支持相当数量的属性，列出整个清单需要占用大量的空间，因此在这里列出相关的链接：

- 列出一个字符的所有属性 <https://unicode.org/cldr/utility/character.jsp> .
- 按照属性列出所有的字符 <https://unicode.org/cldr/utility/list-unicodeset.jsp> .
- 属性的对应缩写形式:  
<https://www.unicode.org/Public/UCD/latest/ucd/PropertyValueAliases.txt> .
- 以文本格式整理的所有 Unicode 字符，包含了所有的属性:  
<https://www.unicode.org/Public/UCD/latest/ucd/> .

## 实例：16 进制数字

举个例子，让我们来查找 16 进制数字，写作 `xFF` 其中 `F` 是一个 16 进制的数字（0...1 或者 A...F）。

一个 16 进制数字可以表示为 `\p{Hex_Digit}` :

```
let regexp = /x\p{Hex_Digit}\p{Hex_Digit}/u;
alert("number: xAF".match(regexp)); // xAF
```

## 实例：中文字符

让我们再来考虑中文字符。

有一个 `unicode` 属性 `Script` （一个书写系统），这个属性可以有一个值：`Cyrillic`，`Greek`，`Arabic`，`Han`（中文）等等，[这里是一个完整的列表](#)。

为了实现查找一个给定的书写系统中的字符，我们需要使用 `Script=<value>`，例如对于西里尔字符: `\p{sc=Cyrillic}`，中文字符: `\p{sc=Han}`，等等。

```
let regexp = /\p{sc=Han}/gu; // returns Chinese hieroglyphs
let str = `Hello Привет 你好 123_456`;
alert(str.match(regexp)); // 你,好
```

## 实例：货币

表示货币的字符，例如 `$`，`€`，`¥`，具有 `unicode` 属性 `\p{Currency_Symbol}`，缩写为 `\p{Sc}`。

让我们使用这一属性来查找符合“货币，接着是一个数字”的价格文本:

```
let regexp = /\p{Sc}\d/gu;
let str = `Prices: $2, €1, ¥9`;
alert(str.match(regexp)); // $2,€1,¥9
```

之后，在文章 [量词`+,\\*,?`和`{n}`](#) 中我们将会了解如何查找包含很多位的数字。

## 总结

修饰符 u 在正则表达式中提供对 **Unicode** 的支持。

这意味着两件事：

1. 4个字节长的字符被以正确的方式处理：被看成单个的字符，而不是2个2字节长的字符。
2. **Unicode** 属性可以被用于查找中 `\p{...}`。

有了 `unicode` 属性我们可以查找给定语言中的词，特殊字符（引用，货币）等等。

## 锚点 (Anchors): 字符串开始 ^ 和末尾 \$

插入符号 ^ 和美元符号 \$ 在正则表达式中具有特殊的含义。它们被称为“锚点”。

插入符号 ^ 匹配文本开头，而美元符号 \$ 则匹配文本末尾。

举个例子，让我们测试一下文本是否以 `Mary` 开头：

```
let str1 = "Mary had a little lamb";
alert(/^Mary/.test(str1)); // true
```

该模式 ^Mary 的意思是：字符串开始，接着是“Mary”。

与此类似，我们可以用 snow\$ 来测试文本是否以 `snow` 结尾：

```
let str1 = "it's fleece was white as snow";
alert(/snow$/.test(str1)); // true
```

在以上这些具体的例子中我们实际上可以用 `startsWith/endsWith` 来代替。正则表达式应该被用于更加复杂的测试中。

## 测试完全匹配

这两个锚点 ^...\$ 放在一起常常被用于测试一个字符串是否完全匹配一个模式。比如，测试用户的输入是否符合正确的格式。

让我们测试一下一个字符串是否属于 `12:34` 格式的时间。即，两个数字，然后一个冒号，接着是另外两个数字。

用正则表达式来表示就是 \d\d:\d\d：

```
let goodInput = "12:34";
let badInput = "12:345";

let regexp = /\d\d:\d\d/;
alert(regexp.test(goodInput)); // true
alert(regexp.test(badInput)); // false
```

在这个例子中 `\d\d:\d\d` 所对应的匹配文本必须正好在文本开头 `^` 之后，而在这之后必须紧跟文本末尾 `$`。

整个字符串必须准确地符合这一个格式。如果其中有任何偏差或者额外的字符，结果将为 `false`。

当修饰符 `m` 出现时，锚点将会有不同的行为。我们将在后面学习到。

### ❶ 锚点具有“零宽度”

锚点 `^` 和 `$` 属于测试。它们的宽度为零。

换句话说，它们并不匹配一个具体的字符，而是让正则引擎测试所表示的条件（文本开头/文本末尾）。

## Flag "m" — 多行模式

通过 flag `/.../m` 可以开启多行模式。

这仅仅会影响 `^` 和 `$` 锚符的行为。

在多行模式下，它们不仅仅匹配文本的开始与结束，还匹配每一行的开始与结束。

### 行的开头 `^`

在这个有多行文本的例子中，正则表达式 `/^\d+/gm` 将匹配每一行的开头数字：

```
let str = `1st place: Winnie
2nd place: Piglet
33rd place: Eeyore`;

alert(str.match(/^\d+/gm)); // 1, 2, 33
```

没有 flag `/.../m` 时，仅仅是第一个数字被匹配到：

```
let str = `1st place: Winnie
2nd place: Piglet
33rd place: Eeyore`;
```

```
alert(str.match(/^\d+/g)); // 1
```

这是因为默认情况下，锚符 `^` 仅仅匹配文本的开头，在多行模式下，它匹配行的开头。

正则表达式引擎将会在文本中查找以锚符 `^` 开始的字符串，我们找到之后继续匹配 `\d+` 模式。

## 行的结尾 `$`

美元符 `$` 行为也相似。

正则表达式 `\w+\$` 会找到每一行的最后一个单词：

```
let str = `1st place: Winnie
2nd place: Piglet
33rd place: Eeyore`;

alert(str.match(/\w+$/gim)); // Winnie,Piglet,Eeyore
```

没有 `/.../m` flag 的话，美元符 `$` 将会仅仅匹配整个文本的结尾，所以只有最后的一个单词会被找到。

## 锚符 `^$` 对比 `\n`

要寻找新的一行的话，我们不仅可以使用锚符 `^` 和 `$`，也可以使用换行符 `\n`。

它和锚符 `^` 和 `$` 的第一个不同点是它不像锚符那样，它会“消耗”掉 `\n` 并且将其 (`\n`) 加入到匹配结果中。

举个例子，我们在下面的代码中用它来替代 `$`：

```
let str = `1st place: Winnie
2nd place: Piglet
33rd place: Eeyore`;

alert(str.match(/\w+\n/gim)); // Winnie\n,Piglet\n
```

这里，我们每次匹配到的时候都会被添加一个换行符。

还有一个不同点——换行符 `\n` 不会匹配字符串结尾。这就是为什么在上面的例子中 `Eeyore` 没有匹配到。

所以，通常情况下使用锚符更棒，用它匹配出来的结果更加接近我们想要的结果。

## 词边界: \b

词边界 `\b` 是一种检查, 就像 `\^` 和 `\$` 一样。

当正则表达式引擎 (实现搜索正则表达式的程序模块) 遇到 `\b` 时, 它会检查字符串中的位置是否是词边界。

有三种不同的位置可作为词边界:

- 在字符串开头, 如果第一个字符是单词字符 `\w`。
- 在字符串中的两个字符之间, 其中一个是单词字符 `\w`, 另一个不是。
- 在字符串末尾, 如果最后一个字符是单词字符 `\w`。

例如, 可以在 `Hello, Java!` 中找到匹配 `\bJava\b` 的单词, 其中 `Java` 是一个独立的单词, 而在 `Hello, JavaScript!` 中则不行。

```
alert("Hello, Java!".match(/\bJava\b/)); // Java
alert("Hello, JavaScript!".match(/\bJava\b/)); // null
```

在字符串 `Hello, Java!` 中, 以下位置对应于 `\b`:

↓      ↓      ↓      ↓  
Hello, Java!

因此, 它与模式 `\bHello\b` 相匹配, 因为:

1. 字符串的开头符合第一种检查 `\b`。
2. 然后匹配了单词 `Hello`。
3. 然后与 `\b` 再次匹配, 因为我们在 `o` 和一个空格之间。

模式 `\bJava\b` 也同样匹配。但 `\bHell\b` (因为 `l` 之后没有词边界) 和 `Java!\b` (因为感叹号不是单词 `\w`, 所以其后没有词边界) 却不匹配。

```
alert("Hello, Java!".match(/\bHello\b/)); // Hello
alert("Hello, Java!".match(/\bJava\b/)); // Java
alert("Hello, Java!".match(/\bHell\b/)); // null (no match)
alert("Hello, Java!".match(/\bJava!\b/)); // null (no match)
```

`\b` 既可以用于单词, 也可以用于数字。

例如, 模式 `\b\d\d\b` 查找独立的两位数。换句话说, 它查找的是两位数, 其周围是与 `\w` 不同的字符, 例如空格或标点符号 (或文本开头/结尾)。

```
alert("1 23 456 78".match(/\b\d\d\b/g)); // 23,78
alert("12,34,56".match(/\b\d\d\b/g)); // 12,34,56
```

### ⚠ 词边界 \b 不适用于非拉丁字母

词边界测试 `\b` 检查位置的一侧是否匹配 `\w`，而另一侧则不匹配“`\w`”。

但是，`\w` 表示拉丁字母 `a-z`（或数字或下划线），因此此检查不适用于其他字符，如西里尔字母（cyrillic letters）或象形文字（hieroglyphs）。

## 转义，特殊字符

正如我们所看到的，一个反斜杠 `"\"` 是用来表示匹配字符类的。所以它是一个特殊字符。

还存在其它的特殊字符，这些字符在正则表达式中有特殊的含义。它们可以被用来做更加强大的搜索。

这里是包含所有特殊字符的列表： `[ \ ^ $ . | ? * + ( ) ]`。

现在并不需要尝试去记住它们——当我们分别处理其中的每一个时，你自然而然就会记住它们。

## 转义

如果要把特殊字符作为常规字符来使用，只需要在它前面加个反斜杠。

这种方式也被叫做“转义一个字符”。

比如说，我们需要找到一个点号 `'.'`。在一个正则表达式中一个点号意味着“除了换行符以外的任意字符”，所以如果我们想真正表示对“一个点号”查询的时候，可以在点号前加一个反斜杠。

```
alert("Chapter 5.1".match(/\d\.\d/)); // 5.1
```

括号也是特殊字符，所以如果我们想要在正则中查找它们，我们应该使用 `\()`。下面的例子会查找一个字符串 `"g()"`：

```
alert("function g()".match(/g\(\)/)); // "g()"
```

如果我们想查找反斜杠 `\`，我们就应该使用两个反斜杠来查找：

```
alert("1\\2".match(/\\/)); // '\'
```

## 一个斜杠

斜杠符号 `'/'` 并不是一个特殊符号，但是它被用于在 **Javascript** 中开启和关闭正则匹配: /...pattern.../，所以我们也应该转义它。

下面是查询斜杠 `'/'` 的表达式:

```
alert("/" .match(/\/)/)); // '/'
```

从另一个方面看，如果使用另一种 `new RegExp` 方式就不需要转义斜杠:

```
alert("/" .match(new RegExp("/")))); // '/'
```

## 使用 `new RegExp` 创建正则实例

如果我们使用 `new RegExp` 来创建一个正则表达式实例，那么我们需要对其做一些额外的转义。

比如说，考虑下面的示例:

```
let reg = new RegExp("\d.\d");

alert("Chapter 5.1".match(reg)); // null
```

它并没有正常发挥作用，但是为什么呢？

原因就在于字符串转义规则。看下面的例子:

```
alert("\d.\d"); // d.d
```

在字符串中的反斜杠表示转义或者类似 `\n` 这种只能在字符串中使用的特殊字符。这个引用会“消费”并且解释这些字符，比如说:

- `\n` —— 变成一个换行字符，
- `\u1234` —— 变成包含该码位的 **Unicode** 字符，
- 。。。其它有些并没有特殊的含义，就像 `\d` 或者 `\z`，碰到这种情况的话会把反斜杠移除。

所以调用 `new RegExp` 会获得一个没有反斜杠的字符串。

如果要修复这个问题，我们需要双斜杠，因为引用会把 `\\"` 变为 `\`：

```
let regStr = "\\d\\.\\d";
alert(regStr); // \d.\d (correct now)

let regexp = new RegExp(regStr);

alert("Chapter 5.1".match(regexp)); // 5.1
```

## Summary

- 要在字面（意义）上搜索特殊字符 `[ \ ^ $ . | ? * + ( )]`，我们需要在它们前面加上反斜杠 `\`（“转义它们”）。
- 如果我们在 `/.../` 内部（但不在 `new RegExp` 内部），还需要转义 `/`。
- 传递一个字符串（参数）给 `new RegExp` 时，我们需要双倍反斜杠 `\\"`，因为字符串引号会消费其中的一个。

## 集合和范围 [...]

在方括号 `[...]` 中的几个字符或者字符类意味着“搜索给定的字符中的任意一个”。

### 集合

比如说，`[eao]` 意味着查找在 3 个字符 `'a'`、`'e'` 或者 `'o'` 中的任意一个。

这被叫做一个 **集合**。集合可以在正则表达式中和其它常规字符一起使用。

```
// 查找 [t 或者 m]，然后再匹配 "op"
alert("Mop top".match(/t|m]op/gi)); // "Mop", "top"
```

请注意尽管在集合中有多个字符，但它们在匹配中只会对应其中的一个。

所以下面的示例并不会匹配上：

```
// 查找 "V"，然后匹配 [o 或者 i]，之后再匹配 "la"
alert("Voila".match(/V[o|i]la/)); // null，并没有匹配上
```

这个模式会做以下假设：

- `V`，

- 然后匹配其中的一个字符 oi ,
- 然后匹配 la ,

所以可以匹配上 Vola 或者 Vila 。

## 范围

方括号也可以包含字符范围。

比如说, [a-z] 会匹配从 a 到 z 范围内的字母, [0-5] 表示从 0 到 5 的数字。

在下面的示例中, 我们会查询首先匹配 "x" 字符, 再匹配两个数字或者位于 A 到 F 范围内的字符。

```
alert("Exception 0xAF".match(/x[0-9A-F][0-9A-F]/g)); // xAF
```

[0-9A-F] 表示两个范围: 它搜索一个字符, 满足数字 0 到 9 或字母 A 到 F。

如果我们还想查找小写字母, 则可以添加范围 a-f : [0-9A-Fa-f] 。或添加标志 i。

我们也可以在 [...] 里面使用字符类。

例如, 如果我们想要查找单词字符 \w 或连字符 -, 则该集合为 [\w-] 。

也可以组合多个类, 例如 [\s\d] 表示“空格字符或数字”。

### ❶ 字符类是某些字符集的简写

例如:

- \d —— 和 [0-9] 相同,
- \w —— 和 [a-zA-Z0-9\_] 相同,
- \s —— 和 [\t\n\v\f\r ] 外加少量罕见的 unicode 空格字符相同。

## 示例: 多语言 \w

由于字符类 \w 是简写的 [a-zA-Z0-9\_] , 因此无法找到中文象形文字, 西里尔字母等。

我们可以编写一个更通用的模式, 该模式可以查找任何语言中的文字字符。这很容易想到就 Unicode 属性: [\p{Alpha}\p{M}\p{Nd}\p{Pc}\p{Join\_C}] 。

让我们理解它。类似于 \w , 我们在制作自己的一套字符集, 包括以下 unicode 字符:

- `Alphabetic (Alpha)`——字母,
- `Mark (M)`——重读,
- `Decimal_Number (Nd)`——数字,
- `Connector_Punctuation (Pc)`——下划线 `'_'` 和类似的字符,
- `Join_Control (Join_C)`——两个特殊代码 `200c` and `200d`, 用于连字, 例如阿拉伯语。

使用示例:

```
let regexp = /[^\p{Alpha}\p{M}\p{Nd}\p{Pc}\p{Join_C}]/gu;
let str = `Hi 你好 12`;
// finds all letters and digits:
alert(str.match(regexp)); // H,i,你,好,1,2
```

当然, 我们可以编辑此模式: 添加 `unicode` 属性或删除它们。文章 [Unicode: 修饰符“u”和 class \p{...}](#) 中包含了更多 `Unicode` 属性的细节。

### Edge 和 Firefox 不支持 Unicode 属性

Edge 和 Firefox 尚未实现 `Unicode` 属性 `\p{...}`。如果确实需要它们, 可以使用库 [XRegExp](#)。

或者只使用我们想要的语言范围的字符, 例如西里尔字母 `[а-я]`。

## 排除范围

除了普通的范围匹配, 还有类似 `[^...]` 的“排除”范围匹配。

它们通过在匹配查询的开头添加插入符号 `^` 来表示, 它会匹配所有除了给定的字符之外的任意字符。

比如说:

- `[^aeyo]` —— 匹配任何除了 `'a'`、`'e'`、`'y'` 或者 `'o'` 之外的字符。
- `[^0-9]` —— 匹配任何除了数字之外的字符, 也可以使用 `\D` 来表示。
- `[^\s]` —— 匹配任何非空字符, 也可以使用 `\S` 来表示。

下面的示例查询除了字母, 数字和空格之外的任意字符:

```
alert("alice15@gmail.com".match(/[^\\d\\sA-Z]/gi)); // @ and .
```

## 在 [...] 中不转义

通常当我们的确需要查询点字符时，我们需要把它转义成像 `\.` 这样的形式。如果我们需要查询一个反斜杠，我们需要使用 `\\\`。

在方括号表示中，绝大多数特殊字符可以在不转义的情况下使用：

- 表示一个点符号 `'.'`。
- 表示一个加号 `'+'`。
- 表示一个括号 `'( )'`。
- 在开头或者结尾表示一个破折号（在这些位置该符号表示的就不是一个范围）`'pattern:'-'`。
- 在不是开头的位置表示一个插入符号（在开头位置该符号表示的是排除）`'^'`。
- 表示一个开口的方括号符号 `'[ '`。

换句话说，除了在方括号中有特殊含义的字符外，其它所有特殊字符都是允许不添加反斜杠的。

一个在方括号中的点符号 `"."` 表示的就是一个点字符。查询模式 `[.,]` 将会寻找一个为点或者逗号的字符。

在下面的示例中，`[-().^.+]` 会查找 `-().^.+` 的其中任意一个字符：

```
// 并不需要转义
let reg = /[-().^.+]/g;

alert("1 + 2 - 3".match(reg)); // 匹配 +, -
```

。。。但是如果你为了“以防万一”转义了它们，这也不会有任何问题：

```
// 转义其中的所有字符
let reg = /[\\-\\\\(\\)\\\\.\\\\^\\\\+]/g;

alert("1 + 2 - 3".match(reg)); // 仍能正常工作: +, -
```

## 范围和标志“u”

如果集合中有代理对（surrogate pairs），则需要标志 `u` 以使其正常工作。

例如，让我们在字符串 `X` 中查找 `[XY]`：

```
alert('X'.match(/[XY]/u)); // 显示一个奇怪的字符, 像 [?]
// (搜索执行不正确, 返回了半个字符)
```

结果不正确，因为默认情况下正则表达式“不知道”代理对。

正则表达式引擎认为 `[XY]` —— 不是两个，而是四个字符：

1. `X` (1) 的左半部分，
2. `X` (2) 的右半部分，
3. `Y` (3) 的左半部分，
4. `Y` (4) 的右半部分。

我们可以看到他们的代码，如下所示：

```
for(let i=0; i<'XY'.length; i++) {
 alert('XY'.charCodeAt(i)); // 55349, 56499, 55349, 56500
}
```

因此，以上示例查找并显示了 `X` 的左半部分。

如果我们添加标志 `u`，那么行为将是正确的：

```
alert('X'.match(/[XY]/u)); // X
```

当我们查找范围时也会出现类似的情况，就像 `[X-Y]`。

如果我们忘记添加标志 `u`，则会出现错误：

```
'X'.match(/[X-Y]/); // 错误：无效的正则表达式
```

原因是，没有标志 `u` 的代理对被视为两个字符，因此 `[X-Y]` 被解释为 `[<55349><56499>-<55349><56500>]`（每个代理对都替换为其代码）。现在很容易看出范围 `56499-55349` 是无效的：其起始代码 `56499` 大于终止代码 `55349`。这就是错误的原因。

使用标志 `u`，该模式可以正常匹配：

```
// 查找字符从 X 到 Z
alert('Y'.match(/[X-Z]/u)); // Y
```

## 量词 `+,\*,?` 和 `{n}`

假设我们有一个字符串 `+7(903)-123-45-67`，并且想要找到它包含的所有数字。但与之前不同的是，我们对单个数字不感兴趣，只对全数感兴趣：`7, 903, 123,`

45, 67。

数字是一个或多个 `\d` 的序列。用来形容我们所需要的数量的词被称为**量词**。

## 数量 {n}

最明显的量词便是一对引号间的数字: `{n}`。在一个字符 (或一个字符类等等) 后跟着一个量词, 用来指出我们具体需要的数量。

它有更高级的格式, 用一个例子来说明:

### 确切的位数: `{5}`

`\d{5}` 表示 5 位的数字, 如同 `\d\d\d\d\d`。

接下来的例子将会查找一个五位数的数字:

```
alert("I'm 12345 years old".match(/\d{5}/)); // "12345"
```

我们可以添加 `\b` 来排除更多位数的数字: `\b\d{5}\b`。

### 某个范围的位数: `{3,5}`

我们可以将限制范围的数字放入括号中, 来查找位数为 3 至 5 位的数字: `\d{3,5}`

```
alert("I'm not 12, but 1234 years old".match(/\d{3,5}/)); // "1234"
```

我们可以省略上限。那么正则表达式 `\d{3,}` 就会查找位数大于或等于 3 的数字:

```
alert("I'm not 12, but 345678 years old".match(/\d{3,}/)); // "345678"
```

对于字符串 `+7(903)-123-45-67` 来说, 我们如果需要一个或多个连续的数字, 就使用 `\d{1,}`:

```
let str = "+7(903)-123-45-67";

let numbers = str.match(/\d{1,}/g);

alert(numbers); // 7,903,123,45,67
```

## 缩写

大多数常用的量词都可以有缩写:

+

代表“一个或多个”，相当于 `{1,}`。

例如，`\d+` 用来查找所有数字：

```
let str = "+7(903)-123-45-67";

alert(str.match(/\d+/g)); // 7, 903, 123, 45, 67
```

?

代表“零个或一个”，相当于 `{0, 1}`。换句话说，它使得符号变得可选。

例如，模式 `ou?r` 查找 `o`，后跟零个或一个 `u`，然后是 `r`。

所以他能够在 `color` 中找到 `or`，以及在 `colour` 中找到 `our`：

```
let str = "Should I write color or colour?";

alert(str.match(/colou?r/g)); // color, colour
```

\*

代表着“零个或多个”，相当于 `{0,}`。也就是说，这个字符可以多次出现或不出现。

接下来的例子将要寻找一个后跟任意数量的 `0` 的数字：

```
alert("100 10 1".match(/\d0*/g)); // 100, 10, 1
```

将它与 `'+'`（一个或多个）作比较：

```
alert("100 10 1".match(/\d0+/g)); // 100, 10
```

## 更多示例

量词是经常被使用的。它们是构成复杂的正则表达式的主要模块之一，我们接着来看更多的例子。

正则表达式“浮点数”（带浮点的数字）：`\d+\.\d+`

实现：

```
alert("0 1 12.345 7890".match(/\d+\.\d+/g)); // 12.345
```

正则表达式“打开没有属性的 HTML 标记”，比如 `<span>` 或 `<p>`： `/<[a-z]+>/i` 实现：

```
alert("<body> ... </body>".match(/<[a-z]+>/gi)); // <body>
```

我们查找字符 '<' 后跟一个或多个英文字母，然后是 '>'。

正则表达式“打开没有属性的HTML标记”（改进版）：`/<[a-z][a-z0-9]*>/i`

更好的表达式：根据标准，HTML 标记名称可以在除了第一个位置以外的任意一个位置有一个数字，比如 `<h1>`。

```
alert("<h1>Hi!</h1>".match(/<[a-z][a-z0-9]*>/gi)); // <h1>
```

正则表达式“打开没有属性的HTML标记”：`/<\/?[a-z][a-z0-9]*>/i`

我们在标记前加上了一个可选的斜杆 /?。必须用一个反斜杠来转义它，否则 JavaScript 就会认为它是这个模式的结束符。

```
alert("<h1>Hi!</h1>".match(/<\/?[a-z][a-z0-9]*>/gi)); // <h1>, </h1>
```

### i 更精确意味着更复杂

我们能够从这些例子中看到一个共同的规则：正则表达式越精确——它就越长且越复杂。

例如，HTML 标记能用一个简单的正则表达式：`<\w+>`。

因为 \w 代表任意英文字母或数字或 '\_'，这个正则表达式也能够匹配非标注的内容，比如 `<_>`。但它要比 `/<[a-z][a-z0-9]*>/i` 简单很多。

我们能够接受 `<\w+>` 或者我们需要 `/<[a-z][a-z0-9]*>/i`？

在现实生活中，两种方式都能接受。取决于我们对于“额外”匹配的宽容程度以及是否难以通过其他方式来过滤掉它们。

## 贪婪量词和惰性量词

量词，看上去十分简单，但实际上它可能会很棘手。

如果我们打算寻找比 `/\d+/` 更加复杂的东西，就需要理解搜索工作是如何进行的。

以接下来的问题为例。

有一个文本，我们需要用书名号: «...» 来代替所有的引号 "..."。在许多国家，它们是排版的首选。

例如: "Hello, world" 将会变成 «Hello, world»。

一些国家偏爱 „Witam, świat!” (波兰语) 甚至 「你好，世界」 (汉语) 引号。对于不同的语言环境，我们可以选择不同的替代方式，但它们都是一样的，那我们就以书名号 «...» 开始。

为了进行替换，我们首先要找出所有被引号围起来的子串。

正则表达式看上去可能是这样的: /".+"/g。这个表达式的意思是：我们要查找这样一个句子，一个引号后跟一个或多个字符，然后以另一个引号结尾。

…但如果我们试着在一个如此简单的例子中去应用它…

```
let reg = /.+$/g;

let str = 'a "witch" and her "broom" is one';

alert(str.match(reg)); // "witch" and her "broom"
```

…我们会发现它的运行结果与预期不同！

它直接找到了一个匹配结果: "witch" and her "broom"，而不是找到两个匹配结果 "witch" 和 "broom"。

这可被称为“贪婪是万恶之源”。

## 贪婪搜索

为了查找到一个匹配项，正则表达式引擎采用了以下算法：

- 对于字符串中的每一个字符
  - 用这个模式来匹配此字符。
  - 若无匹配，移至下一个字符

这些简单的词语没有说清楚为什么这个正则表达式匹配失败了，因此，让我们详细说明一下模式 ".+" 是如何进行搜索工作的。

1. 该模式的第一个字符是一个引号 "。

正则表达式引擎企图在字符串 a "witch" and her "broom" is one 的第一个位置就匹配到目标，但这个位置是 subject:a，所以匹配失败。

然后它进行下一步：移至字符串中的下一个位置，并试图匹配模式中的第一个字符，最终在第三个位置匹配到了引号：



a "witch" and her "broom" is one

- 检测到了引号后，引擎就尝试去匹配模式中的剩余字符。它试图查看剩余的字符串主体是否符合 .+"。

在我们的用例中，模式中的下一个字符为 .（一个点）。它表示匹配除了换行符之外的任意字符，所以将会匹配下一个字符 'w'：



a "witch" and her "broom" is one

- 然后因为量词 .+，模式中的点（.）将会重复。正则表达式引擎逐一读取字符，当该字符可能匹配时就用它来构建匹配项。

...什么时候会不匹配？点（.）能够匹配所有字符，所以只有在移至字符串末尾时才停止匹配：



- 现在引擎完成了对重复模式 .+ 的搜索，并且试图寻找模式中的下一个字符。这个字符是引号 "。但还有一个问题，对字符串的遍历已经结束，已经没有更多的字符了！

正则表达式引擎明白它已经为 .+ 匹配了太多项了，所以开始回溯了。

换句话说，它去掉了量词的匹配项的最后一个字符：



现在它假设在结束前，.+ 会匹配一个字符，并尝试匹配剩余的字符。

如果出现了一个引号，就表示到达了末尾，但最后一个字符是 'e'，所以无法匹配。

5. ...所以引擎会再去掉一个字符，以此来减少 .+ 的重复次数：



'n' 并不会匹配 'n'。

6. 引擎不断进行回溯：它减少了 '..' 的重复次数，直到模式的其它部分（在我们的用例中是 '"）匹配到结果：



7. 匹配完成。

8. 所以，第一次匹配是 "witch" and her "broom"。接下来的搜索的起点位于第一次搜索的终点，但在 is one 中没有更多的引号了，所以没有其它的结果了。

这可能不是我们所想要的，但这就是它的工作原理。

**在贪婪模式下（默认情况下），量词都会尽可能地重复多次。**

正则表达式引擎尝试用 .+ 去获取尽可能多的字符，然后再一步步地筛选它们。

对于这个问题，我们想要另一种结果，这也就是懒惰量词模式的用途。

## 懒惰模式

懒惰模式中的量词与贪婪模式中的是相反的。它想要“重复最少次数”。

我们能够通过在量词之后添加一个问号 '?' 来启用它，所以匹配模式变为 \*? 或 +?，甚至将 '?' 变为 ??。

这么说吧：通常，一个问号 ? 就是一个它本身的量词（0 或 1），但如果添加另一个量词（甚至可以是它自己），就会有不同的意思——它将匹配的模式从贪婪转为懒惰。

正则表达式 "/".+?"/g 正如预期工作：它找到了 "witch" 和 "broom"：

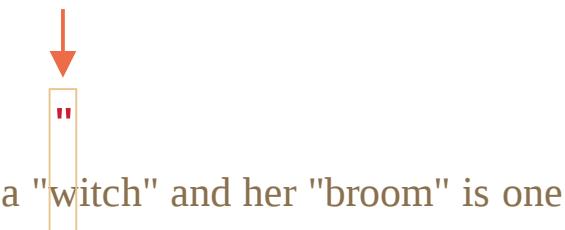
```
let reg = /.+?"/g;

let str = 'a "witch" and her "broom" is one';

alert(str.match(reg)); // witch, broom
```

为了更清楚地理解这个变化，我们来一步步解析这个搜索过程。

1. 第一步依然相同：它在第三个位置开始 `'"`：



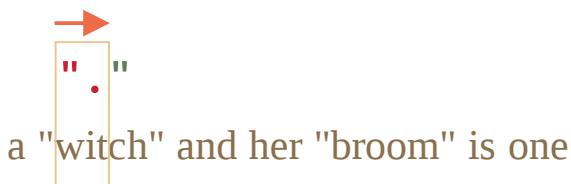
a "witch" and her "broom" is one

2. 下一步也是类似的：引擎为 `'. '` 找到了一个匹配项：



a "witch" and her "broom" is one

3. 接下来就是搜索过程出现不同的时候了。因为我们对 `+?` 启用了懒惰模式，引擎不会去尝试多匹配一个点，并且开始了对剩余的 `'"'` 的匹配：



a "witch" and her "broom" is one

如果有一个引号，搜索就会停止，但是有一个 `'i'`，所以没有匹配到引号。

4. 接着，正则表达式引擎增加对点的重复搜索次数，并且再次尝试：



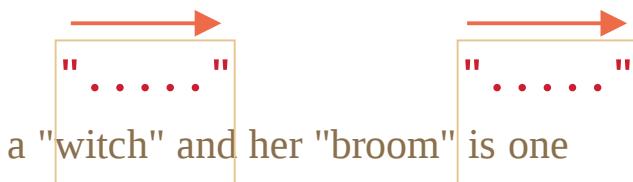
a "witch" and her "broom" is one

又失败了。然后重复次数一次又一次的增加...

5. ...直到模式中的剩余部分找到匹配项:



6. 接下来的搜索工作从当前匹配结束的那一项开始，就会再产生一个结果:



在这个例子中，我们看到了懒惰模式 `+?` 是怎样工作的。量词 `*?` 和 `??` 也有类似的效果——只有在模式的剩余部分无法在给定位置匹配时，正则表达式引擎才会增加重复次数。

**懒惰模式只能通过带 ? 的量词启用**

其它的量词依旧保持贪婪模式。

例如:

```
alert("123 456".match(/\d+ \d+?/g)); // 123 4
```

1. 模式 `\d+` 尝试匹配尽可能多的数字（贪婪模式），因此在它找到 `123` 时停止，因为下一个字符为空格 `' '`。
2. 匹配到一个空格。
3. 由于 `\d+?`。量词是出于懒惰模式的，所以它匹配一个数字 `4` 并且尝试去检测模式的剩余部分是否匹配。  
。。。但是在 `\d+?` 之后没有其它的匹配项了。

懒惰模式不会在不必要的的情况下重复任何事情。模式结束，所以我们找到了匹配项 `123 4`。

4. 接下来的搜索工作从字符 `5` 开始。

## i Optimizations

当代的正则表达式引擎会通过优化内部算法来提升效率。所以它们的工作流程和所描述的算法可能略有不同。

但如果只是为了理解正则表达式是如何工作以及如何构建的，我们不需要知道这些，它们仅用于内部优化。

复杂的正则表达式是难以优化的，所以搜索的过程可能会完全按照描述进行。

## 替代方法

在正则表达式中，通常有多种方法来达到某个相同目的。

在用例中，我们能够在不启用懒惰模式的情况下用 `"[^"]+"` 找到带引号的字符串：

```
let reg = /"[^"]+/"g;

let str = 'a "witch" and her "broom" is one';

alert(str.match(reg)); // witch, broom
```

`"[^"]+"` 得到了正确的答案，因为它查找一个引号 `'"`，后跟一个或多个非引号字符 `[^"]`，然后是结束的引号。

当引擎寻找 `[^"]+` 时，它会在匹配到结束的引号时停止重复，这样就完成了。

请注意，这个逻辑并不能取代惰性量词！

这是不同的，我们有时需要这一个，有时却需要另一个。

让我们再来看一个使用惰性量词失败而使用这种方式正确的例子。

例如，我们想要找到 `<a href=". . ." class="doc">` 形式的链接，或是任意 `href`。

该使用哪个正则表达式呢？

首先可能会想到： `/<a href=". *" class="doc">/g`。

验证一下：

```
let str = '......';
let reg = //g;

// Works!
alert(str.match(reg)); //
```

...但如果文本中有多个链接呢?

```
let str = '...... ...';
let reg = //g;

// Whoops! Two links in one match!
alert(str.match(reg)); // ...
```

现在这个结果和我们的“witches”用例结果的错误原因是一样的。量词 .\* 占用太多字符了。

匹配结果如下:

```

...
```

让我们启用惰性量词 .\*? 来修改模式:

```
let str = '...... ...';
let reg = //g;

// 有效!
alert(str.match(reg)); // ,
```

现在能成功了，有两个匹配项:

```

...
```

它的工作原理是——在上述的解释之后，这应该是显而易见的。所以我们不停留在这些细节上，来再尝试一个例子:

```
let str = '...... <p style="" class="doc">...';
let reg = //g;

// 错误!
alert(str.match(reg)); // ... <p style="" class="doc">
```

我们会发现，这个正则表达式不仅匹配了一个链接，还匹配了包含 <p ...> 的一段文本。

为什么？

1. 首先，正则表达式发现一个链接标签: `<a href=""`。
2. 然后它寻找 `.*?`，我们取一个字符，检查其是否与模式的剩余部分匹配，然后再取另一个。。。  
量词 `.*?` 检测字符，直到 `class="doc">`。  
...在哪里可以找到它呢？我们如果查看文本，就可以看到唯一的 `class="doc">` 是在链接之后的，在 `<p>` 中。
3. 所以有了如下匹配项：

```

... <p style="" class="doc">
```

所以，懒惰模式在这里不起作用。

我们需要寻找 `<a href="...something..." class="doc">`，但贪婪和懒惰模式都有一些问题。

正确的做法应该是这样的： `href="[^"]*"`。它会获取 `href` 属性中的所有字符，正好符合我们的需求。

一个实例：

```
let str1 = '...... <p style="" class="doc">...';
let str2 = '...... ...';
let reg = //g;

// Works!
alert(str1.match(reg)); // 没有匹配项，是正确的
alert(str2.match(reg)); // , <a href="link2" class="do
```

## 总结

量词有两种工作模式：

### 贪婪模式

默认情况下，正则表达式引擎会尝试尽可能多地重复量词。例如，`\d+` 检测所有可能的字符。当不可能检测更多（没有更多的字符或到达字符串末尾）时，然后它再匹配模式的剩余部分。如果没有匹配，则减少重复的次数（回溯），并再次尝试。

### 懒惰模式

通过在量词后添加问号 `?` 来启用。在每次重复量词之前，引擎会尝试去匹配模式的剩余部分。

正如我们所见，懒惰模式并不是针对贪婪搜索的灵丹妙药。另一种方式是“微调”贪婪搜索，我们很快就会见到更多的例子。

## 捕获组

模式的一部分可以用括号括起来 `(...)`。这称为“捕获组 (capturing group) ”。

这有两个影响：

1. 它允许将匹配的一部分作为结果数组中的单独项。
2. 如果我们将量词放在括号后，则它将括号视为一个整体。

## 示例

让我们看看在示例中的括号是如何工作的。

### 示例：gogogo

不带括号，模式 `go+` 表示 `g` 字符，其后 `o` 重复一次或多次。例如 `goooo` 或 `oooooooooooo`。

括号将字符组合，所以 `(go)+` 匹配 `go`，`gogo`，`gogogo` 等。

```
alert('Gogogo now!'.match(/(go)+/i)); // "Gogogo"
```

### 示例：域名

让我们做些更复杂的事——搜索域名的正则表达式。

例如：

```
mail.com
users.mail.com
smith.users.mail.com
```

正如我们所看到的，一个域名由重复的单词组成，每个单词后面有一个点，除了最后一个单词。

在正则表达式中是 `(\w+\. )+\w+`：

```
let regexp = /(\w+\.)+\w+/g;

alert("site.com my.site.com".match(regexp)); // site.com,my.site.com
```

搜索有效，但是该模式无法匹配带有连字符的域名，例如 `my-site.com`，因为连字符不属于 `\w` 类。

我们可以通过用 `[\w-]` 替换 `\w` 来匹配除最后一个的每个单词：  
`([\w-]+\. )+\w+`。

## 示例：email

前面的示例可以扩展。我们可以基于它为电子邮件创建一个正则表达式。

`email` 格式为：`name@domain`。名称可以是任何单词，可以使用连字符和点。在正则表达式中为 `[-.\w]+`。

模式：

```
let regexp = /[-.\w]+@[\w-]+[.\w]+[\w-]+/g;
alert("my@mail.com @ his@site.com.uk".match(regexp)); // my@mail.com, his@site.com.
```

该正则表达式并不完美的，但多数情况下都可以工作，并且有助于修复意外的错误类型。唯一真正可靠的 `email` 检查只能通过发送 `email` 来完成。

## 匹配括号中的内容

括号从左到右编号。正则引擎会记住它们各自匹配的内容，并允许在结果中获得它。

方法 `str.match(regexp)`，如果 `regexp` 没有 `g` 标志，将查找第一个匹配并将它作为一个数组返回：

1. 在索引 `0` 处：完全匹配。
2. 在索引 `1` 处：第一个括号的内容。
3. 在索引 `2` 处：第二个括号的内容。
4. ...等等...

例如，我们想找到 `HTML` 标记 `<.*?>` 并进行处理。这将很方便的把标签内容（尖括号内的内容）放在单独的变量中。

让我们将内部内容包装在括号中，像这样：`<(.*?)>`。

现在，我们能在结果数组中获取标签的整体 `<h1>` 及其内容 `h1`：

```
let str = '<h1>Hello, world!</h1>';
let tag = str.match(/<(.*?)>/);
alert(tag[0]); // <h1>
alert(tag[1]); // h1
```

## 嵌套组

括号可以嵌套。在这种情况下，编号也从左到右。

例如，在搜索标签 `<span class="my">` 时我们可能会对以下内容感兴趣：

1. 整个标签内容: `span class="my"`。
2. 标签名称: `span`。
3. 标签属性: `class="my"`。

让我们为它们添加括号: `<(([a-z]+)\s*([^>]*))>`。

这是它们的编号方式（从左到右，由左括号开始）：

span class="my"  
1 ┌─────────┐  
 <(( [a-z]+ ) \s\* ([^>]\* )) >  
 2 ┌─────────┐ 3 ┌─────────┐  
 span class="my"

实际上：

```
let str = '';

let regexp = /<(([a-z]+)\s*([^\>]*))>/;

let result = str.match(regexp);
alert(result[0]); //
alert(result[1]); // span class="my"
alert(result[2]); // span
alert(result[3]); // class="my"
```

`result` 的零索引始终保持完全匹配。

然后按左括号将组从左到右编号。第一组返回为 `result[1]`。它包含了整个标签内容。

然后 `result[2]` 从第二个开始的括号中进入该组 `([a-z]+)` —— 标签名称，然后在 `result[3]` 标签中: `([^>]*)`。

字符串中每个组的内容：

span class="my"  
1 ┌─────────┐  
 <(( [a-z]+ ) \s\* ([^>]\* )) >  
 2 ┌─────────┐ 3 ┌─────────┐  
 span class="my"

## 可选组

即使组是可选的并且在匹配项中不存在（例如，具有数量词 `(...)?`），也存在相应的 `result` 数组项，并且等于 `undefined`。

例如，让我们考虑正则 `a(z)?(c)?`。它寻找 "a"，然后是可选的 "z"，然后是可选的 "c"。

如果我们在单个字母的字符串上运行 `a`，则结果为：

```
let match = 'a'.match(/a(z)?(c)?/);

alert(match.length); // 3
alert(match[0]); // a (完全匹配)
alert(match[1]); // undefined
alert(match[2]); // undefined
```

数组的长度为 3，但所有组均为空。

这是字符串的一个更复杂的匹配 `ac`：

```
let match = 'ac'.match(/a(z)?(c)?/)

alert(match.length); // 3
alert(match[0]); // ac (完全匹配)
alert(match[1]); // undefined, 因为 (z)? 没匹配项
alert(match[2]); // c
```

数组长度是恒定的：3。但是对于组 `(z)?` 而言，什么都没有，所以结果是 `["ac", undefined, "c"]`。

## 搜索所有具有组的匹配项：`matchAll`

 `matchAll` 是一个新方法，可能需要使用 `polyfill`

旧的浏览器不支持 `matchAll`。

可能需要一个 `polyfill`，例如

[https://github.com/ljharb/String.prototype.matchAll ↗](https://github.com/ljharb/String.prototype.matchAll)

当我们搜索所有匹配项（标志 `g`）时，`match` 方法不会返回组的内容。

例如，让我们查找字符串中的所有标签：

```
let str = '<h1> <h2>';
```

```
let tags = str.match(/<(.*)?>/g);

alert(tags); // <h1>,<h2>
```

结果是一个匹配数组，但没有每个匹配项的详细信息。但是实际上，我们通常需要在结果中获取捕获组的内容。

要获取它们，我们应该使用方法 `str.matchAll(regexp)` 进行搜索。

在使用 `match` 很长一段时间后，它作为“新的改进版本”被加入到 JavaScript 中。

就像 `match` 一样，它寻找匹配项，但有 3 个区别：

1. 它返回的不是数组，而是一个可迭代的对象。
2. 当标志 `g` 存在时，它将每个匹配组作为一个数组返回。
3. 如果没有匹配项，则不返回 `null`，而是返回一个空的可迭代对象。

例如：

```
let results = '<h1> <h2>'.matchAll(/<(.*)?>/gi);

// results - is not an array, but an iterable object
alert(results); // [object RegExp String Iterator]

alert(results[0]); // undefined (*)
```

results = Array.from(results); // let's turn it into array

```
alert(results[0]); // <h1>,h1 (1st tag)
alert(results[1]); // <h2>,h2 (2nd tag)
```

我们可以看到，第一个区别非常重要，如 (\*) 行所示。我们无法获得 `results[0]` 的匹配内容，因为该对象是伪数组。我们可以使用 `Array.from` 把它变成一个真正的 `Array`。在 `Iterable`（可迭代对象）`Iterable object`（可迭代对象）一文中有关于伪数组和可迭代对象的更多详细信息。

如果我们不需要遍历结果，则 `Array.from` 没有必要：

```
let results = '<h1> <h2>'.matchAll(/<(.*)?>/gi);

for(let result of results) {
 alert(result);
 // 第一个结果: <h1>,h1
 // 第二个结果: <h2>,h2
}
```

.....或使用解构:

```
let [tag1, tag2] = '<h1> <h2>'.matchAll(/<(.*)>/gi);
```

由 `matchAll` 所返回的每个匹配，其格式与不带标志 `g` 的 `match` 所返回的格式相同：它是一个具有额外的 `index`（字符串中的匹配索引）属性和 `input`（源字符串）的数组：

```
let results = '<h1> <h2>'.matchAll(/<(.*)>/gi);

let [tag1, tag2] = results;

alert(tag1[0]); // <h1>
alert(tag1[1]); // h1
alert(tag1.index); // 0
alert(tag1.input); // <h1> <h2>
```

### i 为什么 `matchAll` 的结果是可迭代对象而不是数组？

为什么这个方法这样设计？原因很简单 — 为了优化。

调用 `matchAll` 不会执行搜索。相反，它返回一个可迭代的对象，最初没有结果。每当我们对它进行迭代时才会执行搜索，例如在循环中。

因此，这将根据需要找到尽可能多的结果，而不是全部。

例如，文本中可能有 100 个匹配项，但是在在一个 `for..of` 循环中，我们已经找到了 5 个匹配项，然后觉得足够了并做出一个 `break`。这时引擎就不会花时间查找其他 95 个匹配。

## 命名组

用数字记录组很困难。对于简单模式，它是可行的，但对于更复杂的模式，计算括号很不方便。我们有一个更好的选择：给括号起个名字。

这是通过在开始括号之后立即放置 `?<name>` 来完成的。

例如，让我们查找“year-month-day”格式的日期：

```
let dateRegexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/;
let str = "2019-04-30";

let groups = str.match(dateRegexp).groups;

alert(groups.year); // 2019
```

```
alert(groups.month); // 04
alert(groups.day); // 30
```

如您所见，匹配的组在 `.groups` 属性中。

要查找所有日期，我们可以添加标志 `g`。

We'll also need `matchAll` to obtain full matches, together with `groups`: 我们还需要 `matchAll` 获取完整的组匹配:

```
let dateRegexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/g;

let str = "2019-10-30 2020-01-01";

let results = str.matchAll(dateRegexp);

for(let result of results) {
 let {year, month, day} = result.groups;

 alert(` ${day}.${month}.${year}`);
 // 第一个 alert: 30.10.2019
 // 第二个: 01.01.2020
}
```

## 替换捕获组

方法 `str.replace(regexp, replacement)` 用 `replacement` 替换 `str` 中匹配 `regexp` 的所有捕获组。这使用 `$n` 来完成，其中 `n` 是组号。

例如，

```
let str = "John Bull";
let regexp = /(\w+) (\w+)/;

alert(str.replace(regexp, '$2, $1')); // Bull, John
```

对于命名括号，引用为 `$<name>`。

例如，让我们将日期格式从“year-month-day”更改为“day.month.year”：

```
let regexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/g;

let str = "2019-10-30, 2020-01-01";

alert(str.replace(regexp, '$<day>.$<month>.$<year>'));
// 30.10.2019, 01.01.2020
```

## 非捕获组 ?:

有时我们需要括号才能正确应用量词，但我们不希望它们的内容出现在结果中。

可以通过在开头添加 `?:` 来排除组。

例如，如果我们要查找 `(go)+`，但不希望括号内容（`go`）作为一个单独的数组项，则可以编写：`(?:go)+`。

在下面的示例中，我们仅将名称 `John` 作为匹配项的单独成员：

```
let str = "Gogogo John!";

// ?: 从捕获组中排除 'go'
let regexp = /(?:go)+ (\w+)/i;

let result = str.match(regexp);

alert(result[0]); // Gogogo John (完全匹配)
alert(result[1]); // John
alert(result.length); // 2 (数组中没有更多项)
```

## 总结

括号将正则表达式的一部分组合在一起，以便量词可以整体应用。

括号组从左到右编号，可以选择用 `(?<name>...)` 命名。

可以在结果中获得按组匹配的内容：

- 方法 `str.match` 仅当不带标志 `g` 时返回捕获组。
- 方法 `str.matchAll` 始终返回捕获组。

如果括号没有名称，则匹配数组按编号提供其内容。命名括号还可使用属性 `groups`。

我们还可以使用 `str.replace` 来替换括号内容中的字符串：使用 `$n` 或者名称 `$<name>`。

可以通过在组的开头添加 `?:` 来排除编号组。当我们需要对整个组应用量词，但不希望将其作为结果数组中的单独项时这很有用。我们也不能在替换字符串时引用此类括号。

## 模式中的反向引用：`\n` 和 `\k<name>`

我们不仅可以在结果或替换字符串中使用捕获组 (...) 的内容，还可以在模式本身中使用它们。

## 按编号反向引用: \N

可以使用 \N 在模式中引用一个组，其中 N 是组号。

为了弄清那为什么有帮助，让我们考虑一项任务。

我们需要找到带引号的字符串：单引号 '...' 或双引号 "..." – 应匹配两种变体。

如何找到它们？

我们可以将两种引号放在方括号中：["](.\*)["]，但它会找到带有混合引号的字符串，例如 "..." 和 '..."'。当一种引号出现在另一种引号内，比如在字符串 "She's the one!" 中时，便会导致不正确的匹配：

```
let str = `He said: "She's the one!.`;
let regexp = /["](.*?)["]/g;
// 不是我们想要的结果
alert(str.match(regexp)); // "She'
```

如我们所见，该模式找到了一个开头的引号 "，然后文本被匹配，直到另一个引号 "，该匹配结束。

为了确保模式查找的结束引号与开始的引号完全相同，我们可以将其包装到捕获组中并对其进行反向引用：(["])(.\*?)\1。

这是正确的代码：

```
let str = `He said: "She's the one!.`;
let regexp = /(【】)(.*?)\1/g;
alert(str.match(regexp)); // "She's the one!"
```

现在可以了！正则表达式引擎会找到第一个引号 (["]) 并记住其内容。那是第一个捕获组。

\1 在模式中进一步的含义是“查找与第一（捕获）分组相同的文本”，在我们的示例中为完全相同的引号。

与此类似，\2 表示第二（捕获）分组的内容，\3 – 第三分组，依此类推。

### 请注意:

如果我们在组中使用 `?:`，那么我们将无法引用它。用 `(?:...)` 捕获的组被排除，引擎不会存储。

### 不要搞混了：在模式中用 `\1`，在替换项中用： `$1`

在替换字符串中我们使用美元符号：`$1`，而在模式中 – 使用反斜杠 `\1`。

## 按命名反向引用： `\k<name>`

如果正则表达式中有很多括号对（注：捕获组），给它们起个名字方便引用。

要引用命名组，我们可以使用：`\k<name>`。

在下面的示例中引号组命名为 `?<quote>`，因此反向引用为 `\k<quote>`：

```
let str = `He said: "She's the one!"`;
let regexp = /(?<quote>[""])(.*?)\k<quote>/g;
alert(str.match(regexp)); // "She's the one!"
```

## 选择 (OR) |

选择是正则表达式中的一个术语，实际上是一个简单的“或”。

在正则表达式中，它用竖线 `|` 表示。

例如，我们需要找出编程语言：HTML、PHP、Java 或 JavaScript。

对应的正则表达式为：`html|php|java(script)?`。

用例如下：

```
let reg = /html|php|css|java(script)?/gi;
let str = "First HTML appeared, then CSS, then JavaScript";
alert(str.match(reg)); // 'HTML', 'CSS', 'JavaScript'
```

我们已知的一个相似符号——方括号。就允许在许多字符中进行选择，例如 `gr[ae]y` 匹配 `gray` 或 `grey`。

选择符号并非在字符级别生效，而是在表达式级别。正则表达式 A|B|C 意思是命中 A、B 或 C 其一均可。

例如：

- gr(a|e)y 严格等同 gr[ae]y。
- gra|ey 匹配“gra” or “ey”。

我们通常用圆括号把模式中的选择部分括起来，像这样 before(XXX|YYY)after。

## 时间正则表达式

在之前的章节中有个任务是构建用于查找形如 hh:mm 的时间字符串，例如 12:00。但是简单的 \d\d:\d\d 过于模糊。它同时匹配 25:99。

如何构建更优的正则表达式？

我们可以应用到更多的严格匹配结果中：

- 首个匹配数字必须是 0 或 1，同时其后还要跟随任一数字。
- 或者是数字 2 之后跟随 [0-3]。

构建正则表达式： [01]\d|2[0-3]。

接着可以添加冒号和分钟的部分。

分钟的部分必须在 0 到 59 区间，在正则表达式语言中含义为首个匹配数字 [0-5] 其后跟随任一数字 \d。

把他们拼接在一起形成最终的模式 [01]\d|2[0-3]:[0-5]\d。

快大功告成了，但仍然存在一个问题。选择符 | 在 [01]\d 和 2[0-3]:[0-5]\d 之间。这是错误的，因为它只匹配符号左侧或右侧任一表达式。

```
let reg = /[01]\d|2[0-3]:[0-5]\d/g;

alert("12".match(reg)); // 12 (matched [01]\d)
```

这个错误相当明显，但也是初学正则表达式的常见错误。

我们需要添加一个插入语用于匹配时钟： [01]\d 或 2[0-3]。

以下为正确版本：

```
let reg = /([01]\d|2[0-3]):[0-5]\d/g;

alert("00:00 10:10 23:59 25:99 1:2".match(reg)); // 00:00,10:10,23:59
```

## 前瞻断言与后瞻断言

有时候我们需要匹配后面跟着特定模式的一段模式。比如，我们要从 1 turkey costs 30€ 这段字符中匹配价格数值。

我们需要获取 € 符号前面的数值（假设价格是整数）。

那就是前瞻断言要做的事情。

### 前瞻断言

语法为: x(?=y)，它表示“匹配 x，仅在后面是 y 的情况”。

那么对于一个后面跟着 € 的整数金额，它的正则表达式应该为: \d+(?=€)。

```
let str = "1 turkey costs 30€";
alert(str.match(/\d+(?=€)/)); // 30 (正确地跳过了单个的数字 1)
```

让我们来看另一种情况：这次我们想要一个数量，它是一个不被 € 跟着的数值。

这里就要用到前瞻否定断言了。

语法为: x(?!=y)，意思是“查找 x，但是仅在不被 y 跟随的情况下匹配成功”。

```
let str = "2 turkeys cost 60€";
alert(str.match(/\d+(?!€)/)); // 2 (正确地跳过了价格)
```

### 后瞻断言

前瞻断言允许添加一个“后面要跟着什么”的条件判断。

后瞻断言也是类似的，只不过它是在相反的方向上进行条件判断。也就是说，它只允许匹配前面有特定字符串的模式。

语法为：

- 后瞻肯定断言: (?<=y)x，匹配 x，仅在前面是 y 的情况。
- 后瞻否定断言: (?<!y)x，匹配 x，仅在前面不是 y 的情况。

举个例子，让我们把价格换成美元。美元符号通常在数字之前，所以要查找 \$30 我们将使用 (?<=\\$)\d+ ——一个前面带 \$ 的数值：

```
let str = "1 turkey costs $30";

alert(str.match(/(?<=\$)\d+/)); // 30 (跳过了单个的数字 1)
```

另外，为了找到数量 —— 一个前面不带 \$ 的数字，我们可以使用否定后瞻断言：  
(?<! \\$)\d+

```
let str = "2 turkeys cost $60";

alert(str.match(/(?<! \$)\d+/)); // 2 (跳过了价格)
```

## 捕获组

一般来说，环视断言括号中（前瞻和后瞻的通用名称）的内容不会成为匹配到的一部分结果。

例如：在模式 \d+(?!€) 中，€ 符号就不会出现在匹配结果中。

但是如果我们要捕捉整个环视表达式或其中的一部分，那也是有可能的。只需要将其包裹在另加的括号中。

例如，这里货币符号 (€|kr) 和金额一起被捕获了：

```
let str = "1 turkey costs 30€";
let reg = /\d+(?=(€|kr))/; // €|kr 两边有额外的括号

alert(str.match(reg)); // 30, €
```

后瞻断言也一样：

```
let str = "1 turkey costs $30";
let reg = /(?<=(\$|£))\d+/;

alert(str.match(reg)); // 30, $
```

请注意，对于后瞻断言，顺序保持不变，尽管前瞻括号在主模式之前。

通常括号是从左到右编号，但是后瞻断言是一个例外，它总是在主模式之后被捕获。所以 \d+ 的匹配会首先进入结果数组，然后是 (\\$|£)。

## 总结

当我们想根据前面/后面的上下文筛选出一些东西的时候，前瞻断言和后瞻断言（通常被称为“环视断言”）对于简单的正则表达式就很有用。

有时我们可以手动处理来得到相同的结果，即：匹配所有，然后在循环中按上下文进行筛选。请记住，`str.matchAll` 和 `reg.exec` 返回的匹配结果有 `.index` 属性，因此我们能知道它在文本中的确切位置。但通常正则表达式可以做得更好。

环视断言类型：

模式	类型	匹配
<code>x(?=y)</code>	前瞻肯定断言	<code>x</code> ，仅当后面跟着 <code>y</code>
<code>x(?!y)</code>	前瞻否定断言	<code>x</code> ，仅当后面不跟 <code>y</code>
<code>(?&lt;=y)x</code>	后瞻肯定断言	<code>x</code> ，仅当跟在 <code>y</code> 后面
<code>(?&lt;!y)x</code>	后瞻否定断言	<code>x</code> ，仅当不跟在 <code>y</code> 后面

前瞻断言也可用于禁用回溯。为什么它是需要的 – 请看下一章。

## Catastrophic backtracking

Some regular expressions are looking simple, but can execute veeeeeeeery long time, and even “hang” the JavaScript engine.

Sooner or later most developers occasionally face such behavior, because it's quite easy to create such a regexp.

The typical symptom – a regular expression works fine sometimes, but for certain strings it “hangs”, consuming 100% of CPU.

In such case a web-browser suggests to kill the script and reload the page. Not a good thing for sure.

For server-side JavaScript it may become a vulnerability if regular expressions process user data.

## Example

Let's say we have a string, and we'd like to check if it consists of words `\w+` with an optional space `\s?` after each.

We'll use a regexp `^( \w+\s?)*$`, it specifies 0 or more such words.

In action:

```
let regexp = /^(\w+\s?)*$/;
```

```
alert(regexp.test("A good string")); // true
alert(regexp.test("Bad characters: $@#")); // false
```

It seems to work. The result is correct. Although, on certain strings it takes a lot of time. So long that JavaScript engine “hangs” with 100% CPU consumption.

If you run the example below, you probably won’t see anything, as JavaScript will just “hang”. A web-browser will stop reacting on events, the UI will stop working. After some time it will suggest to reload the page. So be careful with this:

```
let regexp = /^(w+\s?)*$/;
let str = "An input string that takes a long time or even makes this regexp to hang

// will take a very long time
alert(regexp.test(str));
```

Some regular expression engines can handle such search, but most of them can’t.

## Simplified example

What’s the matter? Why the regular expression “hangs”?

To understand that, let’s simplify the example: remove spaces \s?. Then it becomes ^(w+)\*\$.

And, to make things more obvious, let’s replace \w with \d. The resulting regular expression still hangs, for instance:

```
let regexp = /^(d+)*$/;
let str = "012345678901234567890123456789!";
// will take a very long time
alert(regexp.test(str));
```

So what’s wrong with the regexp?

First, one may notice that the regexp (\d+)\* is a little bit strange. The quantifier \* looks extraneous. If we want a number, we can use \d+.

Indeed, the regexp is artificial. But the reason why it is slow is the same as those we saw above. So let’s understand it, and then the previous example will become obvious.

What happens during the search of ^(d+)\*\$ in the line 123456789! (shortened a bit for clarity), why does it take so long?

1. First, the regexp engine tries to find a number `\d+`. The plus `+` is greedy by default, so it consumes all digits:

```
\d+.....
(123456789)z
```

Then it tries to apply the star quantifier, but there are no more digits, so it the star doesn't give anything.

The next in the pattern is the string end `$`, but in the text we have `!`, so there's no match:

```
X
\d+.....$
(123456789)!
```

2. As there's no match, the greedy quantifier `+` decreases the count of repetitions, backtracks one character back.

Now `\d+` takes all digits except the last one:

```
\d+.....
(12345678)9!
```

3. Then the engine tries to continue the search from the new position (9).

The star `(\d+)*` can be applied – it gives the number `9`:

```
\d+.....\d+
(12345678)(9)!
```

The engine tries to match `$` again, but fails, because meets `!`:

```
X
\d+.....\d+
(12345678)(9)z
```

4. There's no match, so the engine will continue backtracking, decreasing the number of repetitions. Backtracking generally works like this: the last greedy quantifier decreases the number of repetitions until it can. Then the previous greedy quantifier decreases, and so on.

All possible combinations are attempted. Here are their examples.

The first number `\d+` has 7 digits, and then a number of 2 digits:

```
x
\d+.....\d+
(1234567)(89)!
```

The first number has 7 digits, and then two numbers of 1 digit each:

```
x
\d+.....\d+\d+
(1234567)(8)(9)!
```

The first number has 6 digits, and then a number of 3 digits:

```
x
\d+.....\d+
(123456)(789)!
```

The first number has 6 digits, and then 2 numbers:

```
x
\d+....\d+ \d+
(123456)(78)(9)!
```

...And so on.

There are many ways to split a set of digits `123456789` into numbers. To be precise, there are  $2^n - 1$ , where `n` is the length of the set.

For `n=20` there are about 1 million combinations, for `n=30` – a thousand times more. Trying each of them is exactly the reason why the search takes so long.

What to do?

Should we turn on the lazy mode?

Unfortunately, that won't help: if we replace `\d+` with `\d+?`, the regexp will still hang. The order of combinations will change, but not their total count.

Some regular expression engines have tricky tests and finite automations that allow to avoid going through all combinations or make it much faster, but not all engines, and not in all cases.

## Back to words and strings

The similar thing happens in our first example, when we look words by pattern

$^(\w+\s?)^*$ \$ in the string An input that hangs!.

The reason is that a word can be represented as one  $\w+$  or many:

```
(input)
(inpu)(t)
(inp)(u)(t)
(in)(p)(ut)
...
```

For a human, it's obvious that there may be no match, because the string ends with an exclamation sign !, but the regular expression expects a wordly character  $\w$  or a space  $\s$  at the end. But the engine doesn't know that.

It tries all combinations of how the regexp  $(\w+\s?)^*$  can "consume" the string, including variants with spaces  $(\w+\s)^*$  and without them  $(\w+)^*$  (because spaces  $\s?$  are optional). As there are many such combinations, the search takes a lot of time.

## How to fix?

There are two main approaches to fixing the problem.

The first is to lower the number of possible combinations.

Let's rewrite the regular expression as  $^(\w+\s)^*\w^*$  – we'll look for any number of words followed by a space  $(\w+\s)^*$ , and then (optionally) a word  $\w^*$ .

This regexp is equivalent to the previous one (matches the same) and works well:

```
let regexp = /^(\w+\s)^*\w^$/;
let str = "An input string that takes a long time or even makes this regex to hang!"

alert(regexp.test(str)); // false
```

Why did the problem disappear?

Now the star \* goes after  $\w+\s$  instead of  $\w+\s?$ . It became impossible to represent one word of the string with multiple successive  $\w+$ . The time needed to try such combinations is now saved.

For example, the previous pattern  $(\w+\s?)^*$  could match the word string as two  $\w+$ :

```
\w+\w+
string
```

The previous pattern, due to the optional `\s` allowed variants `\w+`, `\w+\s`, `\w+\w+` and so on.

With the rewritten pattern `(\w+\s)*`, that's impossible: there may be `\w+\s` or `\w+\s\w+\s`, but not `\w+\w+`. So the overall combinations count is greatly decreased.

## Preventing backtracking

It's not always convenient to rewrite a regexp. And it's not always obvious how to do it.

The alternative approach is to forbid backtracking for the quantifier.

The regular expressions engine tries many combinations that are obviously wrong for a human.

E.g. in the regexp `(\d+)*$` it's obvious for a human, that `+` shouldn't backtrack. If we replace one `\d+` with two separate `\d+\d+`, nothing changes:

```
\d+.....
(123456789)!

\d+.\.\.\.\d+.....
(1234)(56789)!
```

And in the original example `^(\w+\s?)*$` we may want to forbid backtracking in `\w+`. That is: `\w+` should match a whole word, with the maximal possible length. There's no need to lower the repetitions count in `\w+`, try to split it into two words `\w+\w+` and so on.

Modern regular expression engines support possessive quantifiers for that. They are like greedy ones, but don't backtrack (so they are actually simpler than regular quantifiers).

There are also so-called “atomic capturing groups” – a way to disable backtracking inside parentheses.

Unfortunately, in JavaScript they are not supported. But there's another way.

## Lookahead to the rescue!

We can prevent backtracking using lookahead.

The pattern to take as much repetitions of `\w` as possible without backtracking is:

(?=(\w+))\1.

Let's decipher it:

- Lookahead `?=` looks forward for the longest word `\w+` starting at the current position.
- The contents of parentheses with `?=...` isn't memorized by the engine, so wrap `\w+` into parentheses. Then the engine will memorize their contents
- ...And allow us to reference it in the pattern as `\1`.

That is: we look ahead – and if there's a word `\w+`, then match it as `\1`.

Why? That's because the lookahead finds a word `\w+` as a whole and we capture it into the pattern with `\1`. So we essentially implemented a possessive plus `+` quantifier. It captures only the whole word `\w+`, not a part of it.

For instance, in the word `JavaScript` it may not only match `Java`, but leave out `Script` to match the rest of the pattern.

Here's the comparison of two patterns:

```
alert("JavaScript".match(/\w+Script/)); // JavaScript
alert("JavaScript".match(/(?=(\w+))\1Script/)); // null
```

1. In the first variant `\w+` first captures the whole word `JavaScript` but then `+` backtracks character by character, to try to match the rest of the pattern, until it finally succeeds (when `\w+` matches `Java`).
2. In the second variant `(?=(\w+))` looks ahead and finds the word `JavaScript`, that is included into the pattern as a whole by `\1`, so there remains no way to find `Script` after it.

We can put a more complex regular expression into `(?=(\w+))\1` instead of `\w`, when we need to forbid backtracking for `+` after it.

**i** 请注意:

There's more about the relation between possessive quantifiers and lookahead in articles [Regex: Emulate Atomic Grouping \(and Possessive Quantifiers\) with LookAhead ↗](#) and [Mimicking Atomic Groups ↗](#).

Let's rewrite the first example using lookahead to prevent backtracking:

```
let regexp = /^(?=(\w+))\2\s?)*$/;

alert(regexp.test("A good string")); // true

let str = "An input string that takes a long time or even makes this regex to hang!"

alert(regexp.test(str)); // false, works and fast!
```

Here `\2` is used instead of `\1`, because there are additional outer parentheses. To avoid messing up with the numbers, we can give the parentheses a name, e.g. `(?\w+)`.

```
// parentheses are named ?<word>, referenced as \k<word>
let regexp = /^(?=(?<word>\w+))\k<word>\s?)*$/;

let str = "An input string that takes a long time or even makes this regex to hang!"

alert(regexp.test(str)); // false

alert(regexp.test("A correct string")); // true
```

The problem described in this article is called “catastrophic backtracking”.

We covered two ways how to solve it:

- Rewrite the regexp to lower the possible combinations count.
- Prevent backtracking.

## 粘性标志 "y"，在位置处搜索

y 标志允许在源字符串中的指定位置执行搜索。

为了掌握 y 标志的用例，看看它有多好，让我们来探讨一个实际的用例。

regexp 的常见任务之一是“词法分析”：比如我们在程序设计语言中得到一个文本，然后分析它的结构元素。

例如，HTML 有标签和属性，JavaScript 代码有函数、变量等。

编写词法分析器是一个特殊的领域，有自己的工具和算法，所以我们就不深究了，但有一个共同的任务：在给定的位置读出一些东西。

例如，我们有一个代码字符串 let varName = "value"，我们需要从其中读取变量名，这个变量名从位置 4 开始。

我们用 regexp \w+ 来查找变量名。实际上，JavaScript 的变量名需要更复杂的 regexp 来进行准确的匹配，但在这里并不重要。

调用 `str.match(/\w+/)` 将只找到该行中的第一个单词。或者是所有带标记 `g` 的单词。但我们只需要在位置 `4` 的一个词。

要从给定位置搜索，我们可以使用方法 `regexp.exec(str)`。

如果 `regexp` 没有标志 `g` 或 `y`，那么这个方法就可以寻找字符串 `str` 中的第一个匹配，就像 `str.match(regexp)` 一样。这种简单的无标志的情况我们在这里并不感兴趣。

如果有标志 `g`，那么它就会在字符串 `str` 中执行搜索，从存储在 `regexp.lastIndex` 属性中的位置开始。如果发现匹配，则将 `regexp.lastIndex` 设置为匹配后的索引。

当一个 `regexp` 被创建时，它的 `lastIndex` 是 `0`。

因此，连续调用 `regexp.exec(str)` 会一个接一个地返回匹配。

一个例子（用标志 `g`）：

```
let str = 'let varName';

let regexp = /\w+/g;
alert(regexp.lastIndex); // 0 (最初 lastIndex=0)

let word1 = regexp.exec(str);
alert(word1[0]); // let (第一个单词)
alert(regexp.lastIndex); // 3 (匹配后的位置)

let word2 = regexp.exec(str);
alert(word2[0]); // varName (第二个单词)
alert(regexp.lastIndex); // 11 (匹配后的位置)

let word3 = regexp.exec(str);
alert(word3); // null (没有更多的匹配)
alert(regexp.lastIndex); // 0 (搜索结束时重置)
```

每个匹配都会以数组形式返回，包含分组和附加属性。

我们可以在循环中得到所有的匹配。

```
let str = 'let varName';
let regexp = /\w+/g;

let result;

while (result = regexp.exec(str)) {
 alert(`Found ${result[0]} at position ${result.index}`);
 // 在位置 0 发现 let，然后
 // 在位置 4 发现 varName
}
```

`regexp.exec` 是 `str.matchAll` 方法的替代方法。

与其他方法不同，我们可以设置自己的 `lastIndex`，从给定位置开始搜索。

例如，让我们从位置 `4` 开始寻找一个单词。

```
let str = 'let varName = "value"';
let regexp = /\w+/g; // 如果没有标志 "g"，属性 lastIndex 会被忽略
regexp.lastIndex = 4;
let word = regexp.exec(str);
alert(word); // varName
```

我们从位置 `regexp.lastIndex = 4` 开始搜索 `w+`。

请注意：搜索从位置 `lastIndex` 开始，然后再往前走。如果在 `lastIndex` 位置上没有词，但它在后面的某个地方，那么它就会被找到：

```
let str = 'let varName = "value"';
let regexp = /\w+/g;
regexp.lastIndex = 3;
let word = regexp.exec(str);
alert(word[0]); // varName
alert(word.index); // 4
```

.....所以，用标志 `g` 属性 `lastIndex` 设置搜索的起始位置。

\*\*标记 `y` 使 `regexp.exec` 正好在 `lastIndex` 位置，而不是在它之前，也不是在它之后。

下面是使用标志 `y` 进行同样的搜索。

```
let str = 'let varName = "value"';
let regexp = /\w+/y;
regexp.lastIndex = 3;
alert(regexp.exec(str)); // null (位置 3 有一个空格，不是单词)
regexp.lastIndex = 4;
alert(regexp.exec(str)); // varName (在位置 4 的单词)
```

我们可以看到，`regexp /\w+/y` 在位置 3 处不匹配(不同于标志 `g` )，而是在位置 4 处匹配。

想象一下，我们有一个长的文本，而里面根本没有匹配。那么用标志 `g` 搜索将一直到文本的最后，这将比用标志 `y` 搜索要花费更多的时间。

在像词法分析这样的任务中，通常在一个确切的位置会有很多搜索。使用标志 `y` 是获得良好性能的关键。

## 正则表达式（RegExp）和字符串（String）的方法

在本文中，我们将深入探讨与正则表达式配合使用的各种方法。

### str.match(regexp)

`str.match(regexp)` 方法在字符串 `str` 中找到匹配 `regexp` 的字符。

它有 3 种模式：

1. 如果 `regexp` 不带有 `g` 标记，则它以数组的形式返回第一个匹配项，其中包含分组和属性 `index` (匹配项的位置)、`input` (输入字符串，等于 `str`)：

```
let str = "I love JavaScript";

let result = str.match(/Java(Script)/);

alert(result[0]); // JavaScript (完全匹配)
alert(result[1]); // Script (第一个分组)
alert(result.length); // 2

// 其他信息:
alert(result.index); // 7 (匹配位置)
alert(result.input); // I love JavaScript (源字符串)
```

2. 如果 `regexp` 带有 `g` 标记，则它将所有匹配项的数组作为字符串返回，而不包含分组和其他详细信息。

```
let str = "I love JavaScript";

let result = str.match(/Java(Script)/g);

alert(result[0]); // JavaScript
alert(result.length); // 1
```

3. 如果没有匹配项，则无论是否带有标记 `g`，都将返回 `null`。

这是一个重要的细微差别。如果没有匹配项，我们得到的不是一个空数组，而是 `null`。忘记这一点很容易出错，例如：

```
let str = "I love JavaScript";

let result = str.match(/HTML/);

alert(result); // null
alert(result.length); // Error: Cannot read property 'length' of null
```

如果我们希望结果是一个数组，我们可以这样写：

```
let result = str.match(regexp) || [];
```

## str.matchAll(regexp)



### A recent addition

This is a recent addition to the language. Old browsers may need polyfills.

方法 `str.matchAll(regexp)` 是 `str.match` “新改进的”变体。

它主要用来搜索所有组的所有匹配项。

与 `match` 相比有 3 个区别：

1. 它返回包含匹配项的可迭代对象，而不是数组。我们可以用 `Array.from` 从中得到一个常规数组。
2. 每个匹配项均以包含分组的数组形式返回（返回格式与不带 `g` 标记的 `str.match` 相同）。
3. 如果没有结果，则返回的不是 `null`，而是一个空的可迭代对象。

用法示例：

```
let str = '<h1>Hello, world!</h1>';
let regexp = /<(.*)?>/g;

let matchAll = str.matchAll(regexp);

alert(matchAll); // [object RegExp String Iterator], 不是数组，而是一个可迭代对象

matchAll = Array.from(matchAll); // 现在返回的是数组
```

```
let firstMatch = matchAll[0];
alert(firstMatch[0]); // <h1>
alert(firstMatch[1]); // h1
alert(firstMatch.index); // 0
alert(firstMatch.input); // <h1>Hello, world!</h1>
```

如果我们用 `for..of` 来循环 `matchAll` 的匹配项，那么我们就不需要 `Array.from` 了，`разумеется`，`ненужен`。

## str.split(regexp|substr, limit)

使用正则表达式（或子字符串）作为分隔符来分割字符串。

我们可以用 `split` 来分割字符串，如下所示：

```
alert('12-34-56'.split('-')) // 数组 ['12', '34', '56']
```

但同样，我们也可以用正则表达式来做：

```
alert('12, 34, 56'.split(/,/)) // 数组 ['12', '34', '56']
```

## str.search(regexp)

方法 `str.search(regexp)` 返回第一个匹配项的位置，如果未找到，则返回 `-1`：

```
let str = "A drop of ink may make a million think";
alert(str.search(/ink/i)); // 10 (第一个匹配位置)
```

**重要限制：** `search` 仅查找第一个匹配项。

如果需要其他匹配项的位置，则应使用其他方法，例如用 `str.matchAll(regexp)` 查找所有位置。

## str.replace(str|regexp, str|func)

这是用于搜索和替换的通用方法，是最有用的方法之一。它是搜索和替换字符串的瑞士军刀。

我们可以不用正则表达式来搜索和替换子字符串：

```
// 用冒号替换连字符
alert('12-34-56'.replace("-", ":")) // 12:34-56
```

不过有一个陷阱。

当 `replace` 的第一个参数是字符串时，它仅替换第一个匹配项。

您可以在上面的示例中看到：只有第一个 `"-"` 被 ":" 替换了。

如要找到所有的连字符，我们不应该用字符串 `"-"`，而应使用带 `g` 标记的正则表达式 `/-/g`：

```
// 将连字符替换为冒号
alert('12-34-56'.replace(/-/g, ":")) // 12:34:56
```

第二个参数是一个替代字符串。我们可以在其中使用特殊字符：

符号	替换字符串中的操作
<code>\$&amp;</code>	插入整个匹配项
<code>\$`</code>	在匹配项之前插入字符串的一部分
<code>\$'</code>	在匹配项之后插入字符串的一部分
<code>\$n</code>	如果 <code>n</code> 是一个 1 到 2 位的数字，则插入第 <code>n</code> 个分组的内容，详见 <a href="#">捕获组</a>
<code>\$&lt;name&gt;</code>	插入带有给定 <code>name</code> 的括号内的内容，详见 <a href="#">捕获组</a>
<code>\$\$</code>	插入字符 <code>\$</code>

例如：

```
let str = "John Smith";

// 交换名字和姓氏
alert(str.replace(/(john) (smith)/i, '$2, $1')) // Smith, John
```

对于需要“智能”替换的场景，第二个参数可以是一个函数。

每次匹配都会调用这个函数，并且返回的值将作为替换字符串插入。

该函数 `func(match, p1, p2, ..., pn, offset, input, groups)` 带参数调用：

1. `match` – 匹配项，
2. `p1, p2, ..., pn` – 分组的内容（如有），
3. `offset` – 匹配项的位置，

4. `input` – 源字符串,
5. `groups` – 所指定分组的对象。

如果正则表达式中没有括号, 则只有 3 个参数: `func(str, offset, input)`。

例如, 将所有匹配项都大写:

```
let str = "html and css";

let result = str.replace(/html|css/gi, str => str.toUpperCase());

alert(result); // HTML and CSS
```

按其在字符串中的位置来替换每个匹配项:

```
alert("Ho-Ho-ho".replace(/ho/gi, (match, offset) => offset)); // 0-3-6
```

在下面的示例中, 有两对括号, 因此将使用 5 个参数调用替换函数: 第一个是完全匹配项, 然后是 2 对括号, 然后是匹配位置 (在示例中未使用) 和源字符串:

```
let str = "John Smith";

let result = str.replace(/(\w+) (\w+)/, (match, name, surname) => `${surname}, ${name}`);

alert(result); // Smith, John
```

如果有许多组, 用 `rest` 参数 (...) 可以很方便的访问:

Если в регулярном выражении много скобочных групп, то бывает удобно использовать остаточные аргументы для обращения к ним:

```
let str = "John Smith";

let result = str.replace(/(\w+) (\w+)/, (...match) => `${match[2]}, ${match[1]}`);

alert(result); // Smith, John
```

或者, 如果我们使用的是命名组, 则带有它们的 `groups` 对象始终是最后一个对象, 因此我们可以这样获得它:

```
let str = "John Smith";
```

```
let result = str.replace(/(？<name>\w+) (？<surname>\w+)/, (...match) => {
 let groups = match.pop();

 return `${groups.surname}, ${groups.name}`;
});

alert(result); // Smith, John
```

使用函数可以为我们提供终极替代功能，因为它可以获取匹配项的所有信息，可以访问外部变量，可以做任何事。

## regexp.exec(str)

`regexp.exec(str)` 方法返回字符串 `str` 中的 `regexp` 匹配项。与以前的方法不同，它是在正则表达式而不是字符串上调用的。

根据正则表达式是否带有标志 `g`，它的行为有所不同。

如果没有 `g`，那么 `regexp.exec(str)` 返回的第一个匹配与 `str.match(regexp)` 完全相同。这没什么新的变化。

但是，如果有标记 `g`，那么：

- 调用 `regexp.exec(str)` 会返回第一个匹配项，并将紧随其后的位置保存在属性 `regexp.lastIndex` 中。-下一次同样的调用会从位置 `regexp.lastIndex` 开始搜索，返回下一个匹配项，并将其后的位置保存在 `regexp.lastIndex` 中。
- ...以此类推。-如果没有匹配项，则 `regexp.exec` 返回 `null`，并将 `regexp.lastIndex` 重置为 `0`。

因此，重复调用会挨个返回所有的匹配项，属性 `regexp.lastIndex` 用来跟踪当前的搜索位置。

过去，在将 `str.matchAll` 方法添加到 `JavaScript` 之前，在循环中是通过调用 `regexp.exec` 来获取分组的所有匹配项：

```
let str = 'More about JavaScript at https://javascript.info';
let regexp = /javascript/ig;

let result;

while (result = regexp.exec(str)) {
 alert(`Found ${result[0]} at position ${result.index}`);
 // Found JavaScript at position 11, 然后
 // Found javascript at position 33
}
```

这个现在也可以使用，尽管对于较新的浏览器来说，`str.matchAll` 通常更方便。

我们可以通过手动设置 `lastIndex`，用 `regexp.exec` 从给定位置进行搜索。

例如：

```
let str = 'Hello, world!';

let regexp = /\w+/g; // 带有标记 "g"，lastIndex 属性被忽略
regexp.lastIndex = 5; // 从第 5 个位置搜索（从逗号开始）

alert(regexp.exec(str)); // world
```

如果正则表达式带有标记 `y`，则搜索将精确地在 `regexp.lastIndex` 位置执行，不会再继续了。

让我们将上例中的 `g` 标记替换为 `y`。现在没有找到匹配项了，因为在位置 `5` 处没有单词：

```
let str = 'Hello, world!';

let regexp = /\w+/y;
regexp.lastIndex = 5; // 在位置 5 精确查找

alert(regexp.exec(str)); // null
```

这个方法在某些场景下很方便，例如需要用正则表达式从字符串的精确位置来“读取”字符串（而不是其后的某处）。

## regexp.test(str)

方法 `regexp.test(str)` 查找匹配项，然后返回 `true/false` 表示是否存在。

例如：

```
let str = "I love JavaScript";

// 这两个测试相同
alert(/love/i.test(str)); // true
alert(str.search(/love/i) != -1); // true
```

一个反例：

```
let str = "Bla-bla-bla";

alert(/love/i.test(str)); // false
alert(str.search(/love/i) != -1); // false
```

如果正则表达式带有标记 `g`，则 `regexp.test` 从 `regexp.lastIndex` 属性中查找，并更新此属性，就像 `regexp.exec` 一样。

因此，我们可以用它从给定位置进行搜索：

```
let regexp = /love/gi;

let str = "I love JavaScript";

// 从位置 10 开始:
regexp.lastIndex = 10;
alert(regexp.test(str)); // false (无匹配)
```

### ⚠ 相同的全局正则表达式在不同的源字符串上测试可能会失败

如果我们在不同的源字符串上应用相同的全局表达式，可能会出现错误的结果，因为 `regexp.test` 的调用会增加 `regexp.lastIndex` 属性值，因此在另一个字符串中的搜索可能是从非 0 位置开始的。

例如，这里我们在同一文本上调用 `regexp.test` 两次，而第二次调用失败了：

```
let regexp = /javascript/g; // (新建 regexp: regexp.lastIndex=0)

alert(regexp.test("javascript")); // true (现在 regexp.lastIndex=10)
alert(regexp.test("javascript")); // false
```

这正是因为在第二个测试中 `regexp.lastIndex` 不为零。

如要解决这个问题，我们可以在每次搜索之前设置 `regexp.lastIndex = 0`。或者，不调用正则表达式的方法，而是使用字符串方法 `str.match/search/...`，这些方法不用 `lastIndex`。