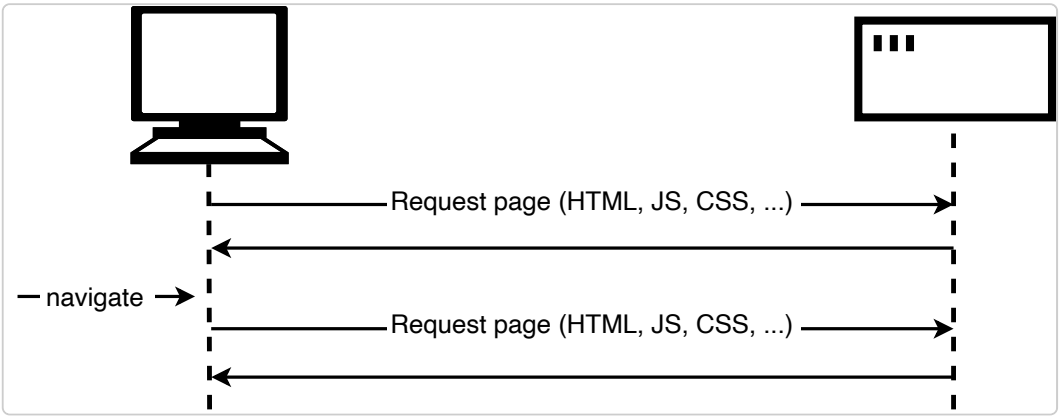
从服务器获取数据

现代网站和应用程序中另一个非常常见的任务是从服务器检索单个数据项以更新网页的各个部分，而无需加载整个新页面。这个看似很小的细节对网站的性能和行为产生了巨大的影响，因此在本文中，我们将解释这个概念并研究使其成为可能的技术：特别是 Fetch API。

先决条件：	JavaScript 基础知识（参见 第一步、 构建块、 JavaScript 对象） 、 客户端 API 基础知识
客观的：	了解如何从服务器获取数据并使用它来更新网页内容。

这里有什么问题?

网页由 HTML 页面和（通常）各种其他文件组成，例如样式表、脚本和图像。Web 上页面加载的基本模型是您的浏览器向服务器发出一个或多个 HTTP 请求以获取显示页面所需的文件，然后服务器以请求的文件进行响应。如果您访问另一个页面，浏览器会请求新文件，而服务器会用它们进行响应。

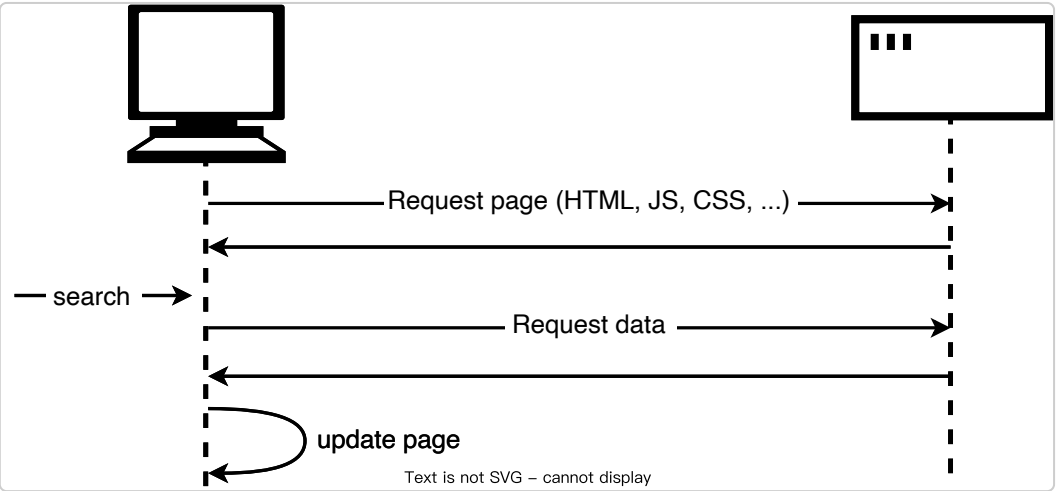


该模型适用于许多站点。但是考虑一个非常数据驱动的网站。例如，像温哥华公共图书馆 这样的图书馆网站。除其他外，您可以将这样的站点视为数据库的用户界面。它可以让您搜索特定类型的书籍，或者可以根据您之前借

过的书籍向您推荐您可能喜欢的书籍。当您这样做时，它需要用要显示的新帐套更新页面。但请注意，大部分页面内容（包括页眉、侧边栏和页脚等项目）保持不变。

传统模型的问题在于我们必须获取并加载整个页面，即使我们只需要更新页面的一部分也是如此。这是低效的，并且会导致糟糕的用户体验。

因此，与传统模型不同，许多网站使用 JavaScript API 从服务器请求数据并在不加载页面的情况下更新页面内容。因此，当用户搜索新产品时，浏览器仅请求更新页面所需的数据——例如要显示的新书集。



这里的主要 API 是Fetch API。这使在页面中运行的 JavaScript 能够向服务器发出HTTP请求以检索特定资源。当服务器提供它们时，JavaScript 可以使用数据来更新页面，通常是通过使用DOM 操作 API。请求的数据通常是JSON，这是传输结构化数据的良好格式，但也可以是 HTML 或纯文本。

这是数据驱动网站（如 Amazon、YouTube、eBay 等）的常见模式。使用此模型：

- 页面更新要快得多，您不必等待页面刷新，这意味着网站感觉更快、响应更快。
- 每次更新时下载的数据更少，这意味着浪费的带宽更少。这在使用宽带连接的台式机上可能不是什么大问题，但在移动设备上和在没有无处不在的快速互联网服务的国家/地区是一个主要问题。

> **注意：**在早期，这种通用技术被称为异步JavaScript 和 XML（Ajax），因为它倾向于请求 XML 数据。这几天通常不是这种情况（您更有可能请求 JSON），但结果仍然相同，并且仍然经常使用术语“Ajax”来描述该技术。

为了进一步加快速度，某些网站还会在首次请求时将资产和数据存储在用户的计算机上，这意味着在随后的访问中，他们会使用本地版本，而不是每次首次加载页面时都下载新副本。内容只有在更新后才会从服务器重新加载。

获取 API

让我们看一下 Fetch API 的几个示例。

获取文本内容

对于此示例，我们将从几个不同的文本文件中请求数据并使用它们来填充内容区域。

这一系列文件将充当我们的假数据库；在实际应用程序中，我们更有可能使用服务器端语言（如 PHP、Python 或 Node）从数据库请求数据。然而，在这里，我们希望保持简单并专注于客户端部分。

要开始此示例，请将fetch-start.html 和四个文本文件（verse1.txt 、verse2.txt 、verse3.txt 和verse4.txt ）复制到您计算机上的一个新目录中。在这个例子中，当在下拉菜单中选择它时，我们将获取这首诗的不同经文（您可能很熟悉）。

在元素内部 `<script>`，添加以下代码。这会存储对 `<select>` 和 `<pre>` 元素的引用，并向该元素添加一个侦听器 `<select>`，以便当用户选择一个新值时，新值将作为 `updateDisplay()` 参数传递给名为的函数。

```
const verseChoose = document.querySelector('select');
const poemDisplay = document.querySelector('pre');

verseChoose.addEventListener('change', () => {
  const verse = verseChoose.value;
```

```
  updateDisplay(verse);
});
```

让我们定义我们的 `updateDisplay()` 功能。首先，将以下内容放在您之前的代码块下方——这是该函数的空壳。

```
function updateDisplay(verse) {

}
```

我们将通过构建一个指向我们要加载的文本文件的相对 URL 来启动我们的函数，因为我们稍后会需要它。元素的值 `<select>` 在任何时候都与所选内容中的文本相同 `<option>` （除非您在值属性中指定不同的值）——例如“Verse 1”。对应的经文文件是“verse1.txt”，和HTML文件在同一个目录下，所以只需要文件名即可。

但是，网络服务器往往区分大小写，文件名中没有空格。要将“Verse 1”转换为“verse1.txt”，我们需要将“V”转换为小写，删除空格，并在末尾添加“.txt”。这可以通过 `replace()` , `toLowerCase()` 和template literal来完成。在您的函数中添加以下行 `updateDisplay()`：

```
verse = verse.replace(' ', '').toLowerCase();
const url = `${verse}.txt`;
```

最后，我们准备好使用 Fetch API：

```
// Call `fetch()`, passing in the URL.
fetch(url)
  // fetch() returns a promise. When we have received a response
from the server,
  // the promise's `then()` handler is called with the response.
  .then((response) => {
    // Our handler throws an error if the request did not
succeed.
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    // Otherwise (if the response succeeded), our handler
fetches the response
```

```
    // as text by calling response.text(), and immediately
returns the promise
    // returned by `response.text()`.
    return response.text();
  })
  // When response.text() has succeeded, the `then()` handler is
called with
  // the text, and we copy it into the `poemDisplay` box.
  .then((text) => poemDisplay.textContent = text)
  // Catch any errors that might happen, and display a message
  // in the `poemDisplay` box.
  .catch((error) => poemDisplay.textContent = `Could not fetch
verse: ${error}`);
```

这里有很多东西要打开。

首先，Fetch API 的入口点是一个名为 的全局函数 `fetch()`，它将 URL 作为参数（它采用另一个可选参数来进行自定义设置，但我们在这里没有使用它）。

接下来， `fetch()` 是一个异步 API，它返回一个 `Promise` . 如果您不知道那是什么，请阅读关于异步 JavaScript 的模块，尤其是关于promises的文章，然后回到这里。您会发现该文章还讨论了 `fetch()` API！

所以因为 `fetch()` 返回一个承诺，我们将一个函数传递给 `then()` 返回的承诺的方法。当 HTTP 请求收到服务器的响应时，将调用此方法。在处理程序中，我们检查请求是否成功，如果没有则抛出错误。否则，我们调用 `response.text()` , 以获取文本形式的响应正文。

原来这 `response.text()` 也是异步的，*所以*我们返回它返回的promise，并传递一个函数到 `then()` 这个新promise的方法中。当响应文本准备好时将调用此函数，在其中我们将 `<pre>` 用文本更新我们的块。

`catch()` 最后，我们在最后链接一个处理程序，以捕获我们调用的异步函数或其处理程序中抛出的任何错误。

这个例子的一个问题是它在第一次加载时不会显示任何诗歌。要解决此问题，请在代码底部（结束 `</script>` 标记上方）添加以下两行以默认加载第 1 节，并确保该 `<select>` 元素始终显示正确的值：

```
updateDisplay('Verse 1');
verseChoose.value = 'Verse 1';
```

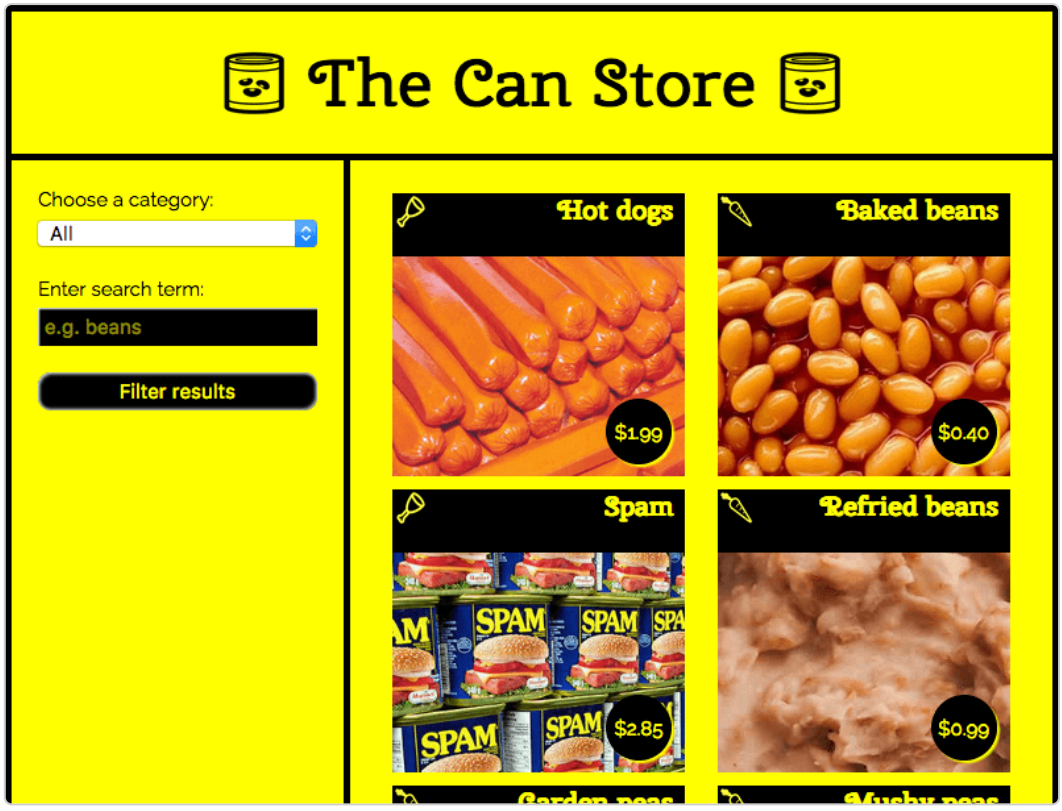
从服务器提供您的示例

如果您只是从本地文件运行示例，现代浏览器将不会运行 HTTP 请求。这是因为安全限制（有关网络安全的更多信息，请阅读网站安全）。

为了解决这个问题，我们需要通过本地 Web 服务器运行示例来测试它。要了解如何执行此操作，请阅读我们的设置本地测试服务器指南。

罐头店

在此示例中，我们创建了一个名为 The Can Store 的示例站点 — 它是一个虚构的超市，仅销售罐头食品。您可以在 GitHub 上找到这个实时示例 ，并查看源代码 。



默认情况下，网站会显示所有产品，但您可以使用左侧栏中的表单控件按类别或搜索词或两者进行过滤。

有相当多的复杂代码处理按类别和搜索词过滤产品、操作字符串以便数据在 UI 中正确显示等。我们不会在本文中讨论所有这些，但您可以找到广泛的代码中的注释（参见can-script.js ）。

但是，我们将解释 Fetch 代码。

使用 Fetch 的第一个块可以在 JavaScript 的开头找到：

```
fetch('products.json')
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    return response.json();
  })
  .then((json) => initialize(json))
  .catch((err) => console.error(`Fetch problem:
${err.message}`));
```

该 `fetch()` 函数返回一个承诺。如果成功完成，第一个块中的函数 `.then()` 包含 `response` 从网络返回的。

在这个函数中我们：

- 检查服务器是否未返回错误（例如 `404 Not Found` ）。如果是，我们抛出错误。
- 呼吁 `json()` 回应。这会将数据检索为JSON 对象。我们返回 返回的承诺 `response.json()` 。

接下来我们将一个函数传递给 `then()` 返回的 promise 的方法。该函数将传递一个对象，其中包含作为 JSON 的响应数据，我们将其传递给该 `initialize()` 函数。此函数启动在用户界面中显示所有产品的过程。

为了处理错误，我们将一个 `.catch()` 块链接到链的末端。如果 promise 由于某种原因失败了，它就会运行。在其中，我们包含一个作为参数传递的函数，一个 `err` 对象。该 `err` 对象可用于报告已发生错误的性质，在本例中我们使用简单的 `console.log()` .

然而，一个完整的网站会通过在用户屏幕上显示一条消息并可能提供补救这种情况的选项来更优雅地处理这个错误，但我们只需要一个简单的 `console.error()`.

您可以自己测试失败案例：

1. 制作示例文件的本地副本。
2. 通过 Web 服务器运行代码（如上所述，在从服务器提供示例中）。
3. 将正在获取的文件的路径修改为类似“produc.json”的内容（确保拼写错误）。
4. 现在在您的浏览器中加载索引文件（通过 `localhost:8000` ）并查看您的浏览器开发人员控制台。您将看到类似于“获取问题：HTTP 错误：404”的消息。

第二个 Fetch 块可以在函数内部找到 `fetchBlob()`：

```
fetch(url)
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP error: ${response.status}`);
    }
    return response.blob();
  })
  .then((blob) => showProduct(blob, product))
  .catch((err) => console.error(`Fetch problem:
${err.message}`));
```

这与前一个的工作方式大致相同，除了 `json()` 我们使用 `blob()`. 在这种情况下，我们希望将我们的响应作为图像文件返回，我们为此使用的数据格式是Blob（该术语是“Binary Large Object”的缩写，基本上可以用来表示类似文件的大型对象，例如作为图像或视频文件）。

一旦我们成功接收到我们的 blob，我们将它传递给我们的 `showProduct()` 函数，它会显示它。

XMLHttpRequest API

有时，尤其是在旧代码中，您会看到另一个名为（通常缩写为“XHR”）的 API `XMLHttpRequest` 用于发出 HTTP 请求。这早于 Fetch，并且确实是第一个广泛用于实现 AJAX 的 API。如果可以，我们建议您使用 Fetch：它是一个更简单的 API，并且比 `XMLHttpRequest` . 我们不会通过使用 的示例 `XMLHttpRequest` ，但我们将向您展示 `XMLHttpRequest` 我们的第一个 can store 请求的版本是什么样的：

```
const request = new XMLHttpRequest();

try {
  request.open('GET', 'products.json');

  request.responseType = 'json';
```



```
  request.addEventListener('error', () => console.error('XHR
error'));

  request.send();

} catch (error) {
  console.error(`XHR error ${request.status}`);
}
```

这有五个阶段：

1. 创建一个新 `XMLHttpRequest` 对象。
2. 调用它的 `open()` 方法来初始化它。
3. 向其 `load` 事件添加一个事件侦听器，该事件会在响应成功完成时触发。在侦听器中，我们调用 `initialize()` 数据。
4. 为其 `error` 事件添加一个事件监听器，当请求遇到错误时触发
5. 发送请求。

我们还必须将整个事情包装在try...catch块中，以处理 `open()` or抛出的任何错误 `send()` 。

希望您认为 Fetch API 是对此的改进。特别是，看看我们如何在两个不同的地方处理错误。

概括

本文介绍如何开始使用 Fetch 从服务器获取数据。

也可以看看

然而，本文讨论了许多不同的主题，这只是真正触及了表面。有关这些主题的更多详细信息，请尝试阅读以下文章：

- Ajax——入门
- 使用获取
- 承诺
- 使用 JSON 数据
- HTTP 概述
- 服务器端网站编程

此页面最后修改于 2023 年 2 月 24 日由MDN 贡献者提供。