引入异步 JavaScript

在本文中，我们将解释什么是异步编程，我们为什么需要它，并简要讨论异步函数历来在 JavaScript 中实现的一些方式。

先决条件：	基本的计算机知识，对 JavaScript 基础知识的合理理解，包括函数和事件处理程序。
客观的：	熟悉什么是异步 JavaScript，它与同步 JavaScript 有何不同，以及我们为什么需要它。

异步编程是一种技术，它使您的程序能够启动一个可能长时间运行的任务，并且仍然能够在该任务运行时响应其他事件，而不必等到该任务完成。一旦该任务完成，您的程序就会显示结果。

浏览器提供的许多功能，尤其是最有趣的功能，可能需要很长时间，因此是异步的。例如：

- 使用 `fetch()`
- 使用访问用户的摄像头或麦克风 `getUserMedia()`
- 要求用户选择文件使用 `showOpenFilePicker()`

因此，即使您可能不必经常*实现*自己的异步函数，您也很可能需要正确*使用它们*。

在本文中，我们将从研究长时间运行的同步函数的问题开始，这使得异步编程成为必要。

同步编程

考虑以下代码：

```
const name = 'Miriam';
const greeting = `Hello, my name is ${name}!`;
console.log(greeting);
// "Hello, my name is Miriam!"
```

这段代码：

1. 声明一个名为 `name` .
2. 声明另一个名为 的字符串 `greeting` ， 它使用 `name` .
3. 将问候语输出到 JavaScript 控制台。

在这里我们应该注意，浏览器实际上是按照我们编写的顺序一次一行地执行程序。在每一点，浏览器都会等待该行完成其工作，然后再继续下一行。它必须这样做，因为每一行都取决于前面几行所做的工作。

这使它成为一个**同步程序**。即使我们调用一个单独的函数，它仍然是同步的，如下所示：

```
function makeGreeting(name) {
  return `Hello, my name is ${name}!`;
}

const name = 'Miriam';
const greeting = makeGreeting(name);
console.log(greeting);
// "Hello, my name is Miriam!"
```

这里， `makeGreeting()` 是一个**同步函数**，因为调用者必须等待函数完成其工作并返回一个值，然后调用者才能继续。

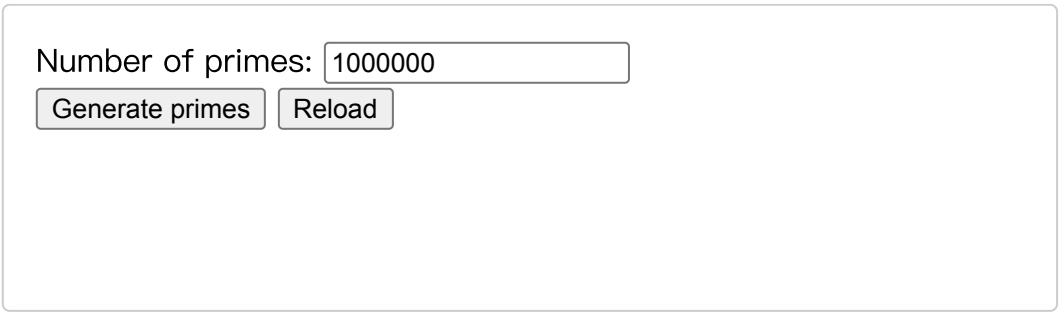
一个长时间运行的同步函数

如果同步功能耗时很长怎么办?

当用户单击“生成素数”按钮时，下面的程序使用了一种非常低效的算法来生成多个大素数。用户指定的素数越多，操作所需的时间就越长。

```
<label for="quota">Number of primes:</label>
<input type="text" id="quota" name="quota" value="1000000" />

<button id="generate">Generate primes</button>
<button id="reload">Reload</button>

<div id="output"></div>


const MAX_PRIME = 1000000;

function isPrime(n) {
  for (let i = 2; i <= Math.sqrt(n); i++) {
    if (n % i === 0) {
      return false;
    }
  }
  return n > 1;
}

const random = (max) => Math.floor(Math.random() * max);

function generatePrimes(quota) {
  const primes = [];
  while (primes.length < quota) {
    const candidate = random(MAX_PRIME);
    if (isPrime(candidate)) {
      primes.push(candidate);
    }
  }
  return primes;
}

const quota = document.querySelector('#quota');
const output = document.querySelector('#output');

document.querySelector('#generate').addEventListener('click', ()
=> {
  const primes = generatePrimes(quota.value);
  output.textContent = `Finished generating ${quota.value}
primes!`;
});

document.querySelector('#reload').addEventListener('click', ()
```

```
=> {
  document.location.reload();
});
```

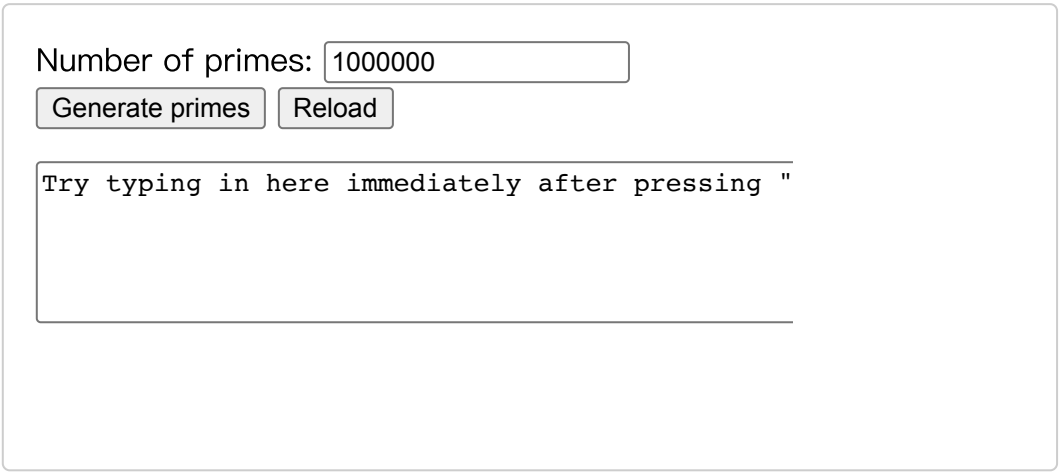


尝试单击“生成素数”。根据您的计算机的速度，程序可能需要几秒钟才能显示“已完成！”。信息。

长时间运行的同步函数的问题

下一个示例与上一个示例一样，只是我们添加了一个文本框供您输入。这次，单击“生成素数”，然后尝试在文本框中立即输入。

您会发现，当我们的 `generatePrimes()` 函数运行时，我们的程序完全没有响应：您不能键入任何内容、单击任何内容或执行任何其他操作。



这是长时间运行的同步函数的基本问题。我们需要的是让我们的程序能够：

1. 通过调用函数启动长时间运行的操作。
2. 让那个函数开始操作并立即返回，这样我们的程序仍然可以响应其他事件。
3. 当操作最终完成时通知我们操作结果。

这正是异步函数可以做的。本模块的其余部分解释了它们是如何在 JavaScript 中实现的。

事件处理器

我们刚才看到的异步函数的描述可能会让您想起事件处理程序，如果是这样，您就对了。事件处理程序实际上是异步编程的一种形式：您提供一个将被调用的函数（事件处理程序），不是立即调用，而是在事件发生时调用。如果“事件”是“异步操作已完成”，则该事件可用于通知调用者异步函数调用的结果。

一些早期的异步 API 就是以这种方式使用事件的。该 `XMLHttpRequest` API 使您能够使用 JavaScript 向远程服务器发出 HTTP 请求。由于这可能需要很长时间，因此它是一个异步 API，您可以通过将事件侦听器附加到对象来获得有关请求的进度和最终完成的通知 `XMLHttpRequest` 。

以下示例显示了这一点。按“点击开始请求”发送请求。我们创建一个新的 `XMLHttpRequest` 并监听它的 `loadend` 事件。处理程序记录“完成！”消息以及状态代码。

添加事件侦听器后，我们发送请求。注意，在此之后，我们可以记录“Started XHR request”：也就是说，我们的程序可以在请求进行的同时继续运行，当请求完成时，我们的事件处理程序将被调用。

```
<button id="xhr">Click to start request</button>
<button id="reload">Reload</button>

<pre readonly class="event-log"></pre>


const log = document.querySelector('.event-log');

document.querySelector('#xhr').addEventListener('click', () => {
  log.textContent = '';

  const xhr = new XMLHttpRequest();

  xhr.addEventListener('loadend', () => {
    log.textContent = `${log.textContent}Finished with status:
```

```
${xhr.status}`;
  });

  xhr.open('GET',
'https://raw.githubusercontent.com/mdn/content/main/files/en-
us/_wikihistory.json');
  xhr.send();
  log.textContent = `${log.textContent}Started XHR
request\n`;});

document.querySelector('#reload').addEventListener('click', ()
=> {
  log.textContent = '';
  document.location.reload();
});
```

Click to start request Reload

这就像我们在前面的模块中遇到的事件处理程序一样，只是事件不是用户操作（例如用户单击按钮），而是某个对象状态的更改。

回调

事件处理程序是一种特殊类型的回调。回调只是一个传递给另一个函数的函数，期望回调将在适当的时间被调用。正如我们刚刚看到的，回调曾经是 JavaScript 中实现异步函数的主要方式。

但是，当回调本身必须调用接受回调的函数时，基于回调的代码可能会变得难以理解。如果您需要执行一些分解为一系列异步函数的操作，这是一种常见的情况。例如，请考虑以下内容：

```
function doStep1(init) {
  return init + 1;
}

function doStep2(init) {
  return init + 2;
```

```
}

function doStep3(init) {
  return init + 3;
}

function doOperation() {
  let result = 0;
  result = doStep1(result);
  result = doStep2(result);
  result = doStep3(result);
  console.log(`result: ${result}`);
}

doOperation();
```

这里我们有一个分为三个步骤的操作，其中每个步骤都取决于最后一步。在我们的示例中，第一步将输入加 1，第二步将输入加 2，第三步将输入加 3。从输入 0 开始，最终结果为 6 (0 + 1 + 2 + 3)。作为同步程序，这非常简单。但是如果我们使用回调来实现这些步骤呢?


```
function doStep1(init, callback) {
  const result = init + 1;
  callback(result);
}

function doStep2(init, callback) {
  const result = init + 2;
  callback(result);
}

  const result = init + 3;
  callback(result);
}

function doOperation() {
  doStep1(0, (result1) => {
    doStep2(result1, (result2) => {
      doStep3(result2, (result3) => {
        console.log(`result: ${result3}`);
      });
    });
```

```
  });
}

doOperation();
```

因为我们必须在回调中调用回调，所以我们得到了一个深度嵌套的 `doOperation()` 函数，这更难阅读和调试。这有时被称为“回调地狱”或“厄运金字塔”（因为缩进在其侧面看起来像金字塔）。

当我们像这样嵌套回调时，处理错误也会变得非常困难：通常你必须在“金字塔”的每一层处理错误，而不是在顶层只处理一次错误。

由于这些原因，大多数现代异步 API 不使用回调。相反，JavaScript 中异步编程的基础是 `Promise`，这就是下一篇文章的主题。

此页面最后修改于 2023 年 2 月 23 日由MDN 贡献者提供。