异步 JavaScript 简介

在本文中，我们将解释什么是异步编程，为什么我们需要它，并简要讨论 JavaScript 历史上异步函数是怎样被实现的。

前提：	基本的计算机素养，以及对 JavaScript 基础知识的一定了解，包括函数和事件处理程序。
目标：	熟悉异步 JavaScript 的概念，了解它与同步 JavaScript 的不同，以及我们需要它的原因。

异步编程技术使你的程序可以在执行一个可能长期运行的任务的同时继续对其他事件做出反应而不必等待任务完成。与此同时，你的程序也将在任务完成后显示结果。

浏览器提供的许多功能（尤其是最有趣的那一部分）可能需要很长的时间来完成，因此需要异步完成，例如：

- 使用 `fetch()` 发起 HTTP 请求
- 使用 `getUserMedia()` 访问用户的摄像头和麦克风
- 使用 `showOpenFilePicker()` 请求用户选择文件以供访问

因此，即使你可能不需要经常*实现*自己的异步函数，你也很可能需要*正确使用*它们。

在这篇文章中，我们将从同步函数长时间运行时存在的问题开始，并以此进一步认识异步编程的必要性。

同步编程

观察下面的代码：

JS

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

这段代码：

1. 声明了一个叫做 `name` 的字符串常量
2. 声明了另一个叫做 `greeting` 的字符串常量（并使用了 `name` 常量的值）
3. 将 `greeting` 常量输出到 JavaScript 控制台中。

我们应该注意的是，实际上浏览器是按照我们书写代码的顺序一行一行地执行程序的。浏览器会等待代码的解析和工作，在上一行完成后才会执行下一行。这样做是很有必要的，因为每一行新的代码都是建立在前面代码的基础之上的。

这也使得它成为一个**同步程序**。

事实上，调用函数的时候也是同步的，就像这样：

JS

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

在这里 `makeGreeting()` 就是一个**同步函数**，因为在函数返回之前，调用者必须等待函数完成其工作。

一个耗时的同步函数

如果同步函数需要很长的时间怎么办？

当用户点击“生成素数”按钮时，这个程序将使用一种非常低效的算法生成一些大素数。你可以控制要生成的素数数量，这也会影响操作需要的时间。

HTML Play

```
<label for="quota">素数个数：</label>
<input type="text" id="quota" name="quota" value="1000000" />

<button id="generate">生成素数</button>
<button id="reload">重载</button>

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

JS Play

```
function generatePrimes(quota) {
  function isPrime(n) {
    for (let c = 2; c <= Math.sqrt(n); ++c) {
      if (n % c === 0) {
        return false;
      }
    }
    return true;
  }
  const primes = [];
  const maximum = 1000000;
  while (primes.length < quota) {
    const candidate = Math.floor(Math.random() * (maximum + 1));
    if (isPrime(candidate)) {
      primes.push(candidate);
    }
  }
  return primes;
}
document.querySelector("#generate").addEventListener("click", () => {
  const quota = document.querySelector("#quota").value;
  const primes = generatePrimes(quota);
  document.querySelector("#output").textContent =
    `完成！已生成素数${quota}个。`;
});
document.querySelector("#reload").addEventListener("click", () => {
  document.location.reload();
});
```

Play

素数个数： 1000000 生成素数 重载

试着点击“生成素数”按钮。在程序显示“完成！”信息之前可能需要几秒钟（取决于你的电脑性能）。

耗时同步函数的问题

接下来的示例和上一个一样，不过我们增加了一个文本框供你输入。这一次，试着点击“生成素数”，然后在文本框中输入。

你会发现，当我们的 `generatePrimes()` 函数运行时，我们的程序完全没有反应：用户不能输入任何东西，也不能点击任何东西，或做任何其他事情。

Play

素数个数 1000000 生成素数 重载

点击“生成素数”按钮后试着在这里输入

这就是耗时的同步函数的基本问题。在这里我们想要的是一种方法，以让我们的程序可以：

- 通过调用一个函数来启动一个长期运行的操作
- 让函数开始操作并立即返回，这样我们的程序就可以保持对其他事件做出反应的能力
- 当操作最终完成时，通知我们操作的结果。

这就是异步函数为我们提供的能力，本模块的其余部分将解释它们是如何在 JavaScript 中实现的。

事件处理程序

我们刚才看到的对异步函数的描述可能会让你想起事件处理程序，这么想是对的。事件处理程序实际上就是异步编程的一种形式：你提供的函数（事件处理程序）将在事件发生时被调用（而不是立即被调用）。如果“事件”是“异步操作已经完成”，那么你就可以看到事件如何被用来通知调用者异步函数调用的结果的。

一些早期的异步 API 正是以这种方式来使用事件的。 `XMLHttpRequest` API 可以让你用 JavaScript 向远程服务器发起 HTTP 请求。由于这样的操作可能需要很长的时间，所以它被设计成异步 API，你可以通过给 `XMLHttpRequest` 对象附加事件监听器来让程序在请求进展和最终完成时获得通知。

下面的例子展示了这样的操作。点击“点击发起请求”按钮来发送一个请求。我们将创建一个新的 `XMLHttpRequest` 并监听它的 `loadend` 事件。而我们的事件处理程序则会在控制台中输出一个“完成！”的消息和请求的状态代码。

我们在添加了事件监听器后发送请求。注意，在这之后，我们仍然可以在控制台中输出“请求已发起”，也就是说，我们的程序可以在请求进行的同时继续运行，而我们的事件处理程序将在请求完成时被调用。

HTML Play

```
<button id="xhr">点击发起请求</button>
<button id="reload">重载</button>

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

JS Play

```
const log = document.querySelector(".event-log");
document.querySelector("#xhr").addEventListener("click", () => {
  log.textContent = "";
  const xhr = new XMLHttpRequest();
  xhr.addEventListener("loadend", () => {
    log.textContent = `${log.textContent}完成！状态码：${xhr.status}`;
  });
  xhr.open(
```

```
    "GET",
    "https://raw.githubusercontent.com/mdn/content/main/files/en-us/_wikihistory.json",
  );
  xhr.send();
  log.textContent = `${log.textContent}请求已发起\n`;
});
document.querySelector("#reload").addEventListener("click", () => {
  log.textContent = "";
  document.location.reload();
});
```

Play

点击发起请求 重载

这就像我们在以前的模块中遇到的事件处理程序，只是这次的事件不是像点击按钮那样的用户行为，而是某个对象的状态变化。

回调

事件处理程序是一种特殊类型的回调函数。而回调函数则是一个被传递到另一个函数中的会在适当的时候被调用的函数。正如我们刚刚所看到的：回调函数曾经是 JavaScript 中实现异步函数的主要方式。

然而，当回调函数本身需要调用其他同样接受回调函数的函数时，基于回调的代码会变得难以理解。当你需要执行一些分解成一系列异步函数的操作时，这将变得十分常见。例如下面这种情况：

JS

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

```
function doStep3(init) {
  return init + 3;
}
function doOperation() {
  let result = 0;
  result = doStep1(result);
  result = doStep2(result);
  result = doStep3(result);
  console.log(`结果: ${result}`);
}
doOperation();
```

现在我们有一个被分成三步的操作，每一步都依赖于上一步。在这个例子中，第一步给输入的数据加 1，第二步加 2，第三步加 3。从输入 0 开始，最终结果是 6（0+1+2+3）。作为同步代码，这很容易理解。但是如果我们用回调来实现这些步骤呢？

JS

```
function doStep1(init, callback) {
  const result = init + 1;
  callback(result);
}
function doStep2(init, callback) {
  const result = init + 2;
  callback(result);
}
function doStep3(init, callback) {
  const result = init + 3;
  callback(result);
}
function doOperation() {
  doStep1(0, (result1) => {
    doStep2(result1, (result2) => {
      doStep3(result2, (result3) => {
        console.log(`结果: ${result3}`);
      });
    });
  });
}
doOperation();
```

因为必须在回调函数中调用回调函数，我们就得到了这个深度嵌套的 `doOperation()` 函数，这就更难阅读和调试了。在一些地方这被称为“回调地狱”或“厄运金字塔”（因为缩进看起来像一个金字

塔的侧面）。

面对这样的嵌套回调，处理错误也会变得非常困难：你必须在“金字塔”的每一级处理错误，而不是在最高一级一次完成错误处理。

由于以上这些原因，大多数现代异步 API 都不使用回调。事实上，JavaScript 中异步编程的基础是 `Promise`，这也是我们下一篇文章要讲述的主题。