> 此页面由社区从英文翻译而来。了解更多并加入 MDN Web Docs 社区。

# 如何使用 Promise

**Promise** 是现代 JavaScript 中异步编程的基础。它是一个由异步函数返回的对象，可以指示操作当前所处的状态。在 Promise 返回给调用者的时候，操作往往还没有完成，但 Promise 对象提供了方法来处理操作最终的成功或失败。

| | |
|---|---|
| **前提：** | 对 JavaScript 基础知识的一定了解，包括事件处理程序。 |
| **目标：** | 了解如何在 JavaScript 中使用 Promise。 |

在上一篇文章中，我们谈到使用回调实现异步函数的方法。在这种设计中，我们需要在调用异步函数的同时传入回调函数。这个异步函数会立即返回，并在操作完成后调用传入的回调。

在基于 Promise 的 API 中，异步函数会启动操作并返回一个 `Promise` 对象。然后，你可以将处理函数附加到 Promise 对象上，当操作完成时（成功或失败），这些处理函数将被执行。

## 使用 fetch() API

> **备注：** 在这篇文章中，我们将通过复制页面上的代码示例到浏览器的 JavaScript 控制台中运行的方式来学习 Promise。因此在正式开始学习之前你需要进行以下设置：
>
> 1. 在浏览器中打开一个新标签页并访问 https://example.org 。
> 2. 在该标签页中，打开浏览器开发者工具中的 JavaScript 控制台
> 3. 把我们展示的代码示例复制到控制台中运行。值得注意的是，你必须在每次输入新的示例之前重新加载页面，否则控制台会报错“重新定义了 `fetchPromise` ”。

在这个例子中，我们将从 https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json 下载 JSON 文件，并打印一些相关信息。

要做到这一点，我们将向服务器发出一个 **HTTP 请求**。在 HTTP 请求中，我们向远程服务器发送一个请求消息，然后它向我们发送一个响应。在这里，我们将发送一个请求，从服务器上获得一个 JSON 文件。还记得在上一篇文章中，我们使用 `XMLHttpRequest` API 进行 HTTP 请求吗？那么，在这篇文章中，我们将使用 `fetch()` API，一个现代的、基于 Promise 的、用于替代 `XMLHttpRequest` 的方法。

把下列代码复制到你的浏览器 JavaScript 控制台中：

JS

```
const fetchPromise = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);

console.log(fetchPromise);

fetchPromise.then((response) => {
  console.log(`已收到响应：${response.status}`);
});

console.log("已发送请求……");
```

我们在这里：

1. 调用 `fetch()` API，并将返回值赋给 `fetchPromise` 变量。
2. 紧接着，输出 `fetchPromise` 变量，输出结果应该像这样：`Promise { <state>: "pending" }`。这告诉我们有一个 `Promise` 对象，它有一个 `state` 属性，值是 `"pending"`。`"pending"` 状态意味着操作仍在进行中。
3. 将一个处理函数传递给 Promise 的 **`then()`** 方法。当（如果）获取操作成功时，Promise 将调用我们的处理函数，传入一个包含服务器的响应的 `Response` 对象。
4. 输出一条信息，说明我们已经发送了这个请求。

完整的输出结果应该是这样的：

```
Promise { <state>: "pending" }
已发送请求……
已收到响应：200
```

请注意，`已发送请求……` 的消息在我们收到响应之前就被输出了。与同步函数不同，`fetch()` 在请求仍在进行时返回，这使我们的程序能够保持响应性。响应显示了 `200` （OK）的状态码，意味着我们的请求成功了。

可能这看起来很像上一篇文章中的例子中我们把事件处理程序添加到 `XMLHttpRequest` 对象中。但不同的是，我们这一次将处理程序传递到返回的 Promise 对象的 `then()` 方法中。

## 链式使用 Promise

在你通过 `fetch()` API 得到一个 `Response` 对象的时候，你需要调用另一个函数来获取响应数据。在这里，我们想获得 JSON 格式的响应数据，所以我们会调用 `Response` 对象的 `json()` 方法。事实上，`json()` 也是异步的，因此我们必须连续调用两个异步函数。

试试这个：

JS

```
const fetchPromise = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);

fetchPromise.then((response) => {
  const jsonPromise = response.json();
  jsonPromise.then((json) => {
    console.log(json[0].name);
  });
});
```

在这个示例中，就像我们之前做的那样，我们给 `fetch()` 返回的 Promise 对象添加了一个 `then()` 处理程序。但这次我们的处理程序调用 `response.json()` 方法，然后将一个新的 `then()` 处理程序传递到 `response.json()` 返回的 Promise 中。

执行代码后应该会输出“baked beans”（“products.json”中第一个产品的名称）。

等等！还记得上一篇文章吗？我们好像说过，在回调中调用另一个回调会出现多层嵌套的情况？我们是不是还说过，这种“回调地狱”使我们的代码难以理解？这不是也一样吗，只不过变成了用 `then()` 调用而已？

当然如此。但 Promise 的优雅之处在于 *`then()` 本身也会返回一个 Promise，这个 Promise 将指示 `then()` 中调用的异步函数的完成状态*。这意味着我们可以（当然也应该）把上面的代码改写成这样：

JS

```
const fetchPromise = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);

fetchPromise
  .then((response) => response.json())
  .then((data) => {
    console.log(data[0].name);
  });
```

不必在第一个 `then()` 的处理程序中调用第二个 `then()`，我们可以直接*返回* `json()` 返回的 Promise，并在该返回值上调用第二个 `then()`。这被称为 **Promise 链**，意味着当我们需要连续进行异步函数调用时，我们就可以避免不断嵌套带来的缩进增加。

在进入下一步之前，还有一件事要补充：我们需要在尝试读取请求之前检查服务器是否接受并处理了该请求。我们将通过检查响应中的状态码来做到这一点，如果状态码不是“OK”，就抛出一个错误：

JS

```
const fetchPromise = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);

fetchPromise
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP 请求错误：${response.status}`);
    }
    return response.json();
```

```
  })
  .then((json) => {
    console.log(json[0].name);
  });
```

## 错误捕获

这给我们带来了最后一个问题：我们如何处理错误？ `fetch()` API 可能因为很多原因抛出错误（例如，没有网络连接或 URL 本身存在问题），我们也会在服务器返回错误消息时抛出一个错误。

在上一篇文章中，我们看到在嵌套回调中进行错误处理非常困难，我们需要在每一个嵌套层中单独捕获错误。

为了支持错误处理， `Promise` 对象提供了一个 `catch()` 方法。这很像 `then()`：你调用它并传入一个处理函数。然后，当异步操作*成功*时，传递给 `then()` 的处理函数被调用，而当异步操作*失败*时，传递给 `catch()` 的处理函数被调用。

如果将 `catch()` 添加到 Promise 链的末尾，它就可以在任何异步函数失败时被调用。于是，我们就可以将一个操作实现为几个连续的异步函数调用，并在一个地方处理所有错误。

试试这个版本的 `fetch()` 代码。我们使用 `catch()` 添加了一个错误处理函数，并修改了 URL（这样请求就会失败）。

JS

```
const fetchPromise = fetch(
  "bad-scheme://mdn.github.io/learning-area/javascript/apis/fetching-data/can-
store/products.json",
);

fetchPromise
  .then((response) => {
    if (!response.ok) {
      throw new Error(`HTTP 请求错误：${response.status}`);
    }
    return response.json();
  })
  .then((json) => {
    console.log(json[0].name);
  })
  .catch((error) => {
```

```
  console.error(`无法获取产品列表：${error}`);
});
```

尝试运行这个版本：你应该会看到 `catch()` 处理函数输出的错误。

## Promise 术语

Promise 中有一些具体的术语值得我们弄清楚。

首先，Promise 有三种状态：

- **待定（pending）**：初始状态，既没有被兑现，也没有被拒绝。这是调用 `fetch()` 返回 Promise 时的状态，此时请求还在进行中。
- **已兑现（fulfilled）**：意味着操作成功完成。当 Promise 完成时，它的 `then()` 处理函数被调用。
- **已拒绝（rejected）**：意味着操作失败。当一个 Promise 失败时，它的 `catch()` 处理函数被调用。

注意，这里的“成功”或“失败”的含义取决于所使用的 API：例如，`fetch()` 认为服务器返回一个错误（如 404 Not Found）时请求成功，但如果网络错误阻止请求被发送，则认为请求失败。

有时我们用**已敲定**（settled）这个词来同时表示**已兑现**（fulfilled）和**已拒绝**（rejected）两种情况。

如果一个 Promise 已敲定，或者如果它被“锁定”以跟随另一个 Promise 的状态，那么它就是**已解决**（resolved）的。

文章 Let's talk about how to talk about promises 对这些术语的细节做了很好的解释。

## 合并使用多个 Promise

当你的操作由几个异步函数组成，而且你需要在开始下一个函数之前完成之前每一个函数时，你需要的就是 Promise 链。但是在其他的一些情况下，你可能需要合并多个异步函数的调用，`Promise` API 为解决这一问题提供了帮助。

有时你需要所有的 Promise 都得到实现，但它们并不相互依赖。在这种情况下，将它们一起启动然后在它们全部被兑现后得到通知会更有效率。这里需要 `Promise.all()` 方法。它接收一个 Promise 数组，并返回一个单一的 Promise。

由 `Promise.all()` 返回的 Promise：

- 当且仅当数组中*所有*的 Promise 都被兑现时，才会通知 `then()` 处理函数并提供一个包含所有响应的数组，数组中响应的顺序与被传入 `all()` 的 Promise 的顺序相同。
- 会被拒绝——如果数组中有*任何一个* Promise 被拒绝。此时， `catch()` 处理函数被调用，并提供被拒绝的 Promise 所抛出的错误。

譬如：

JS

```
const fetchPromise1 = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-
store/products.json",
);
const fetchPromise2 = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/not-found",
);
const fetchPromise3 = fetch(
  "https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json",
);

Promise.all([fetchPromise1, fetchPromise2, fetchPromise3])
  .then((responses) => {
    for (const response of responses) {
      console.log(`${response.url}: ${response.status}`);
    }
  })
  .catch((error) => {
    console.error(`获取失败: ${error}`);
  });
```

这里我们向三个不同的 URL 发出三个 `fetch()` 请求。如果它们都被兑现了，我们将输出每个请求的响应状态。如果其中任何一个被拒绝了，我们将输出失败的情况。

根据我们提供的 URL，应该所有的请求都会被兑现，尽管因为第二个请求中请求的文件不存在，服务器将返回 `404` （Not Found）而不是 `200` （OK）。所以输出应该是：

```
https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json: 200
https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/not-found: 404
https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json: 200
```

如果我们用一个错误编码的 URL 尝试同样的代码，就像这样：

JS

```
const fetchPromise1 = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json",
);
const fetchPromise2 = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/not-found",
);
const fetchPromise3 = fetch(
  "bad-scheme://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json",
);

Promise.all([fetchPromise1, fetchPromise2, fetchPromise3])
  .then((responses) => {
    for (const response of responses) {
      console.log(`${response.url}: ${response.status}`);
    }
  })
  .catch((error) => {
    console.error(`获取失败：${error}`);
  });
```

然后 `catch()` 处理程序将被运行，我们应该看到像这样的输出：

```
获取失败：TypeError: Failed to fetch
```

有时，你可能需要一组 Promise 中的某一个 Promise 的兑现，而不关心是哪一个。在这种情况下，你需要 `Promise.any()`。这就像 `Promise.all()`，不过在 Promise 数组中的任何一个被兑现时它就会被兑现，如果所有的 Promise 都被拒绝，它也会被拒绝。

JS

```
const fetchPromise1 = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-
store/products.json",
);
const fetchPromise2 = fetch(
  "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/not-found",
);
const fetchPromise3 = fetch(
  "https://mdn.github.io/learning-area/javascript/oojs/json/superheroes.json",
);

Promise.any([fetchPromise1, fetchPromise2, fetchPromise3])
  .then((response) => {
    console.log(`${response.url}: ${response.status}`);
  })
  .catch((error) => {
    console.error(`获取失败: ${error}`);
  });
```

值得注意的是，在这种情况下，我们无法预测哪个获取请求会先被兑现。

这两个用于组合多个承诺的函数只是额外的 `Promise` 函数中的两个。要了解其余的内容，参见 `Promise` 参考文档。

## async 和 await

`async` 关键字为你提供了一种更简单的方法来处理基于异步 Promise 的代码。在一个函数的开头添加 `async`，就可以使其成为一个异步函数。

JS

```
async function myFunction() {
  // 这是一个异步函数
}
```

在异步函数中，你可以在调用一个返回 Promise 的函数之前使用 `await` 关键字。这使得代码在该点上等待，直到 Promise 被完成，这时 Promise 的响应被当作返回值，或者被拒绝的响应被作为错误抛出。

这使你能够编写像同步代码一样的异步函数。例如，我们可以用它来重写我们的 fetch 示例。

JS

```
async function fetchProducts() {
  try {
    // 在这一行之后，我们的函数将等待 `fetch()` 调用完成
    // 调用 `fetch()` 将返回一个“响应”或抛出一个错误
    const response = await fetch(
      "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-
store/products.json",
    );
    if (!response.ok) {
      throw new Error(`HTTP 请求错误：${response.status}`);
    }
    // 在这一行之后，我们的函数将等待 `response.json()` 的调用完成
    // `response.json()` 调用将返回 JSON 对象或抛出一个错误
    const json = await response.json();
    console.log(json[0].name);
  } catch (error) {
    console.error(`无法获取产品列表：${error}`);
  }
}

fetchProducts();
```

这里我们调用 `await fetch()`，我们的调用者得到的并不是 `Promise`，而是一个完整的 `Response` 对象，就好像 `fetch()` 是一个同步函数一样。

我们甚至可以使用 `try...catch` 块来处理错误，就像我们在写同步代码时一样。

但请注意，这个写法只在异步函数中起作用。异步函数总是返回一个 Pomise，所以你不能做这样的事情：

JS

```
  async function fetchProducts() {
    try {
      const response = await fetch(
        "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-
  store/products.json",
      );
      if (!response.ok) {
        throw new Error(`HTTP 请求错误：${response.status}`);
      }
```

```
    const data = await response.json();
    return data;
  } catch (error) {
    console.error(`无法获取产品列表：${error}`);
  }
}

const promise = fetchProducts();
console.log(promise[0].name); // “promise”是一个 Promise 对象，因此这句代码无法正常工作
```

相反，你需要做一些事情，比如：

JS

```
async function fetchProducts() {
  try {
    const response = await fetch(
      "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-
store/products.json",
    );
    if (!response.ok) {
      throw new Error(`HTTP 请求错误：${response.status}`);
    }
    const data = await response.json();
    return data;
  } catch (error) {
    console.error(`无法获取产品列表：${error}`);
  }
}

const promise = fetchProducts();
promise.then((data) => console.log(data[0].name));
```

同样地，请注意你只能在 `async` 函数中使用 `await`，除非你的代码是 JavaScript 模块。这意味着你不能在普通脚本中这样做：

JS

```
try {
  // 只有在模块中才能在异步函数之外使用 await
  const response = await fetch(
    "https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-
store/products.json",
```

```
  );
  if (!response.ok) {
    throw new Error(`HTTP 请求错误：${response.status}`);
  }
  const data = await response.json();
  console.log(data[0].name);
} catch (error) {
  console.error(`无法获取产品列表：${error}`);
}
```

你可能会在需要使用 Promise 链地方使用 `async` 函数，这也使得 Promise 的工作更加直观。

请记住，就像一个 Promise 链一样， `await` 强制异步操作以串联的方式完成。如果下一个操作的结果取决于上一个操作的结果，这是必要的，但如果不是这样，像 `Promise.all()` 这样的操作会有更好的性能。

## 小结

Promise 是现代 JavaScript 异步编程的基础。它避免了深度嵌套回调，使表达和理解异步操作序列变得更加容易，并且它们还支持一种类似于同步编程中 `try...catch` 语句的错误处理方式。

`async` 和 `await` 关键字使得从一系列连续的异步函数调用中建立一个操作变得更加容易，避免了创建显式 Promise 链，并允许你像编写同步代码那样编写异步代码。



---

在这篇文章中，我们没有涉及到所有的 Promise 功能，只是介绍了最有趣和最有用的那一部分。随着你开始学习更多关于 Promise 的知识，你会遇到更多有趣的特性。

许多现代 Web API 是基于 Promise 的，包括 WebRTC、Web Audio API、媒体捕捉与媒体流等等。

## 参见

- `Promise()`
- 使用 Promise

- We have a problem with promises by Nolan Lawson
- Let's talk about how to talk about promises

## Help improve MDN

Was this page helpful to you?

Yes No

Learn how to contribute.

This page was last modified on 2024年4月22日 by MDN contributors.