如何使用承诺

Promises是现代 JavaScript 中异步编程的基础。Promise 是异步函数返回的对象，代表操作的当前状态。当 promise 返回给调用者时，操作通常还没有完成，但是 promise 对象提供了处理操作最终成功或失败的方法。

先决条件：	基本的计算机知识，对 JavaScript 基础知识的合理理解，包括事件处理。
客观的：	了解如何在 JavaScript 中使用承诺。

在上一篇文章中，我们谈到了使用回调来实现异步功能。使用该设计，您调用异步函数，传入您的回调函数。该函数立即返回并在操作完成时调用您的回调。

使用基于承诺的 API，异步函数启动操作并返回一个 `Promise` 对象。然后，您可以将处理程序附加到此 promise 对象，这些处理程序将在操作成功或失败时执行。

使用 fetch() API

> **注意：**在本文中，我们将通过将代码示例从页面复制到浏览器的 JavaScript 控制台来探索承诺。要进行设置：
>
> 1. 打开浏览器选项卡并访问https://example.org
> 2. 在该选项卡中，在浏览器的开发人员工具中打开 JavaScript 控制台

3. 当我们展示一个例子时，将它复制到控制台中。每次输入新示例时都必须重新加载页面，否则控制台会抱怨您已重新声明 `fetchPromise` 。

在此示例中，我们将从https://mdn.github.io/learning-area/javascript/apis/fetching-data/can-store/products.json 下载 JSON 文件，并记录一些相关信息。

为此，我们将向服务器发出**HTTP 请求**。在 HTTP 请求中，我们向远程服务器发送请求消息，它向我们发回响应。在这种情况下，我们将发送请求以从服务器获取 JSON 文件。还记得在上一篇文章中，我们使用 `XMLHttpRequest` API 发出 HTTP 请求吗？那么，在本文中，我们将使用 `fetch()` API，它是 `XMLHttpRequest` .

将其复制到浏览器的 JavaScript 控制台：

```
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(`Received response: ${response.status}`);
});

console.log("Started request…");
```

我们到了：

1. 调用 `fetch()` API，并将返回值赋给 `fetchPromise` 变量
2. 紧接着，记录 `fetchPromise` 变量。这应该输出如下内容： `Promise { <state>: "pending" }` ，告诉我们我们有一个 `Promise` 对象，它的 `state` 值为 `"pending"` 。该 `"pending"` 状态意味着获取操作仍在进行中。
3. 将处理函数传递给 Promise 的 **`then()`** 方法。当（如果）获取操作成功时，promise 将调用我们的处理程序，传入一个 `Response` 包含服务器

响应的对象。

4. 记录我们已经开始请求的消息。

完整的输出应该是这样的：

```
Promise { <state>: "pending" }
Started request…
Received response: 200
```

请注意， `Started request…` 在我们收到响应之前已记录。与同步函数不同， `fetch()` 在请求仍在进行时返回，使我们的程序能够保持响应。响应显示 `200` (OK)状态代码，表示我们的请求成功。

这可能看起来很像上一篇文章中的示例，我们在其中向 `XMLHttpRequest` 对象添加了事件处理程序。取而代之的是，我们将一个处理程序传递给 `then()` 返回的 promise 的方法。

链接承诺

使用 `fetch()` API，一旦获得一个 `Response` 对象，就需要调用另一个函数来获取响应数据。在这种情况下，我们希望获取 JSON 格式的响应数据，因此我们将调用对象 `json()` 的方法 `Response` 。原来 `json()` 也是异步的。所以在这种情况下，我们必须调用两个连续的异步函数。

尝试这个：

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

在此示例中，与之前一样，我们 `then()` 向 返回的承诺添加了一个处理程序 `fetch()` 。但是这一次，我们的处理程序调用 `response.json()` ，然后将一个新的 `then()` 处理程序传递给 返回的承诺 `response.json()` 。

这应该记录“baked beans”（“products.json”中列出的第一个产品的名称）。

可是等等！还记得上一篇文章，我们说通过在另一个回调中调用一个回调，我们会得到越来越多的代码嵌套层级吗？而我们说这个“回调地狱”让我们的代码难以理解？这不也一样，只是 `then()` 调用？

当然是。但是 promises 的优雅特性是它 *`then()` 本身返回一个 promise，它将用传递给它的函数的结果来完成*。这意味着我们可以（当然应该）像这样重写上面的代码：

```
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);
  });
```

我们可以*返回*由 返回的承诺 `then()` ，然后在该返回值上调用第二个，而不是在第一个的处理程序中调用第二个。这称为**promise 链**，意味着当我们需要进行连续的异步函数调用时，我们可以避免不断增加的缩进级别。

`then()` `json()` `then()`

在我们继续下一步之前，还有一点要添加。在我们尝试读取请求之前，我们需要检查服务器是否接受并能够处理该请求。我们将通过检查响应中的状态代码并在它不是“OK”时抛出错误来做到这一点：

```
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 error: ${response.status}`);
  }
  return response.json();
})
.then((data) => {
  console.log(data[0].name);
});
```

捕获错误

这将我们带到了最后一部分：我们如何处理错误？API `fetch()` 可能会因多种原因抛出错误（例如，因为没有网络连接或 URL 以某种方式格式错误），如果服务器返回错误，我们自己也会抛出错误。

在上一篇文章中，我们看到嵌套回调的错误处理变得非常困难，这让我们在每个嵌套级别处理错误。

为了支持错误处理， `Promise` 对象提供了一种 `catch()` 方法。这很像 `then()`：你调用它并传入一个处理函数。但是，当 `then()` 异步操作*成功* `catch()` 时调用传递给的处理程序，而当异步操作*失败*时调用传递给的处理程序。

如果你添加 `catch()` 到一个承诺链的末尾，那么它会在任何异步函数调用失败时被调用。因此，您可以将一个操作实现为多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 添加了错误处理程序 `catch()`，还修改了 URL，因此请求将失败。

```
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 error: ${response.status}`);
    }
    return response.json();
```

```
})
.then((data) => {
  console.log(data[0].name);
})
.catch((error) => {
  console.error(`Could not get products: ${error}`);
});
```

尝试运行这个版本：你应该看到我们的处理程序记录的错误 `catch()` 。

承诺术语

Promises 带有一些非常具体的术语，值得弄清楚。

首先，承诺可以处于以下三种状态之一：

- **pending**：promise 已经创建，并且它关联的异步函数尚未成功或失败。这是您的承诺在从对 的调用返回时所处的状态 `fetch()` ，并且仍在发出请求。
- **fulfilled**：异步函数已经成功。当一个承诺被履行时，它的 `then()` 处理程序被调用。
- **rejected**：异步函数失败。当一个承诺被拒绝时，它的 `catch()` 处理程序被调用。

请注意，此处“成功”或“失败”的含义取决于所讨论的 API：例如， `fetch()` 如果服务器返回404 Not Found之类的错误，则认为请求成功，但如果网络错误阻止请求发送，则认为请求成功。

有时，我们使用术语**settled**来涵盖**fulfilled**和**rejected**。

如果承诺已**解决**，或者如果它已“锁定”以遵循另一个承诺的状态，则承诺已解决。

文章让我们谈谈如何谈论承诺 对这个术语的细节给出了很好的解释。

组合多个承诺

当您的操作包含多个异步函数时，您需要使用承诺链，并且您需要在开始下一个函数之前完成每个函数。但是您可能需要通过其他方式组合异步函数调用，并且 `Promise` API 为它们提供了一些帮助程序。

有时候，你需要所有的承诺都实现，但它们并不相互依赖。在这种情况下，一起启动它们，然后在它们全部完成时收到通知会更有效率。该 `Promise.all()` 方法是您在这里需要的。它接受一系列承诺并返回一个承诺。

返回的承诺 `Promise.all()` 是：

- 当数组中的*所有*承诺都实现时以及是否实现。在这种情况下， `then()` 将使用所有响应的数组调用处理程序，其顺序与传递到 promise 的顺序相同 `all()` 。
- *当数组中的任何*承诺被拒绝时以及是否被拒绝。在这种情况下，`catch()` 调用处理程序时会出现被拒绝的 promise 抛出的错误。

例如：

```
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(`Failed to fetch: ${error}`)
  });
```

在这里，我们 `fetch()` 向三个不同的 URL 发出了三个请求。如果他们都成功了，我们将记录每一个的响应状态。如果其中任何一个失败，那么我们将

记录失败。

使用我们提供的 URL，所有请求都应该得到满足，尽管第二次，服务器将返回 `404` (Not Found) 而不是 `200` (OK)，因为请求的文件不存在。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
```

如果我们使用格式错误的 URL 尝试相同的代码，如下所示：

```
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(`Failed to fetch: ${error}`)
  });
```

然后我们可以期待 `catch()` 处理程序运行，我们应该看到类似的东西：

```
Failed to fetch: TypeError: Failed to fetch
```

有时，您可能需要履行一组承诺中的任何一个，而不关心是哪一个。在那种情况下，你想要 `Promise.any()`。这就像 `Promise.all()`，除了它会在任何

承诺数组中的任何一个得到满足时立即得到满足，或者如果所有承诺都被拒绝则被拒绝：

```
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(`Failed to fetch: ${error}`)
  });
```

请注意，在这种情况下，我们无法预测哪个提取请求将首先完成。

这些只是 `Promise` 组合多个承诺的额外功能中的两个。要了解其余部分，请参阅 `Promise` 参考文档。

异步和等待

关键字 `async` 为您提供了一种更简单的方法来处理基于承诺的异步代码。`async` 在函数的开头添加使其成为异步函数：

```
async function myFunction() {
  // This is an async function
}
```

在异步函数中，您可以 `await` 在调用返回承诺的函数之前使用关键字。这使得代码在那一点等待，直到 promise 被解决，此时 promise 的已实现值被视为返回值，或者被拒绝的值被抛出。

这使您能够编写使用异步函数但看起来像同步代码的代码。例如，我们可以用它来重写我们的获取示例：

```
async function fetchProducts() {
  try {
    // after this line, our function will wait for the `fetch()`
call to be settled
    // the `fetch()` call will either return a Response or throw
an error
    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 error: ${response.status}`);
    }
    // after this line, our function will wait for the
`response.json()` call to be settled
    // the `response.json()` call will either return the parsed
JSON object or throw an error
    const data = await response.json();
    console.log(data[0].name);
  }
  catch (error) {
    console.error(`Could not get products: ${error}`);
  }
}

fetchProducts();
```

在这里，我们调用 `await fetch()`，而不是获取一个 `Promise`，我们的调用者返回一个完整的 `Response` 对象，就像 `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，就像代码是同步的一样。

请注意，尽管异步函数总是返回一个承诺，因此您不能执行以下操作：

```
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 error: ${response.status}`);
    }
```

```
    const data = await response.json();
    return data;
  }
  catch (error) {
    console.error(`Could not get products: ${error}`);
  }
}

const promise = fetchProducts();
console.log(promise[0].name);   // "promise" is a Promise
object, so this will not work
```

相反，您需要执行以下操作：

```
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 error: ${response.status}`);
    }
```



```
  catch (error) {
    console.error(`Could not get products: ${error}`);
  }
}

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

另外请注意，您只能在函数 `await` 内部使用 `async`，除非您的代码位于 JavaScript 模块中。这意味着您不能在普通脚本中执行此操作：

```
try {
  // using await outside an async function is only allowed in a
module
  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 error: ${response.status}`);
  }
  const data = await response.json();
  console.log(data[0].name);
}
catch(error) {
  console.error(`Could not get products: ${error}`);
}
```

您可能会 `async` 在可能会使用 promise 链的地方大量使用函数，它们使使用 promise 更加直观。

请记住，就像承诺链一样， `await` 强制异步操作按顺序完成。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，但如果不是这种情况，那么类似的东西 `Promise.all()` 会更高效。

结论

Promises 是现代 JavaScript 中异步编程的基础。它们使得在没有深层嵌套回调的情况下更容易表达和推理异步操作序列，并且它们支持类似于同步语句的错误处理风格 `try...catch` 。

和关键字使从一系列连续的异步函数调用构建操作变得更加容易，避免了创建显式承诺链的需要，并允许您编写看起来就像同步代码的代码 `async` 。 `await`

Promises 适用于所有现代浏览器的最新版本；唯一会出现 promise 支持问题的地方是 Opera Mini 和 IE11 及更早版本。

我们没有在本文中触及 promises 的所有特性，只涉及最有趣和最有用的特性。当您开始更多地了解 promises 时，您会遇到更多的特性和技术。

许多现代 Web API 都是基于承诺的，包括WebRTC、Web Audio API、Media Capture 和 Streams API等等。

也可以看看

- `Promise()`

- 使用承诺
- 我们对诺兰劳森的承诺有疑问
- 让我们谈谈如何谈论承诺

此页面最后修改于 2023 年 2 月 26 日由MDN 贡献者提供。