

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。

## 使用 Promise

[上一页](#)[下一页](#)

[Promise](#) 是一个对象，它代表了一个异步操作的最终完成或者失败。因为大多数人仅仅是使用已创建的 Promise 实例对象，所以本教程将首先说明怎样使用 Promise，再说明如何创建 Promise。

本质上 Promise 是一个函数返回的对象，我们可以在它上面绑定回调函数，这样我们就没有必要一开始把回调函数作为参数传入这个函数了。

假设现在有一个名为 `createAudioFileAsync()` 的函数，它接收一些配置和两个回调函数，然后异步地生成音频文件。一个回调函数在文件成功创建时被调用，另一个则在出现异常时被调用。

以下为使用 `createAudioFileAsync()` 的示例：

JS

```
// 成功的回调函数
function successCallback(result) {
  console.log("音频文件创建成功：" + result);
}

// 失败的回调函数
function failureCallback(error) {
  console.log("音频文件创建失败：" + error);
}

createAudioFileAsync(audioSettings, successCallback, failureCallback);
```

如果重写 `createAudioFileAsync()` 为返回 Promise 的形式，你可以把回调函数附加到它上面：

JS

---

```
createAudioFileAsync(audioSettings).then(successCallback, failureCallback);
```

这种形式有若干优点，下面我们将会逐一讨论。

## 链式调用

连续执行两个或者多个异步操作是一个常见的需求，在上一个操作执行成功之后，开始下一个的操作，并带着上一步操作所返回的结果。在旧的回调风格中，这种操作会导致经典的[回调地狱](#)：

JS

---

```
doSomething(function (result) {
  doSomethingElse(result, function (newResult) {
    doThirdThing(newResult, function (finalResult) {
      console.log(`得到最终结果: ${finalResult}`);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

有了 Promise，我们就可以通过一个 Promise 链来解决这个问题。这就是 Promise API 的优势，因为回调函数是附加到返回的 Promise 对象上的，而不是传入一个函数中。

见证奇迹的时刻：`then()` 函数会返回一个和原来不同的**新的 Promise**：

JS

---

```
const promise = doSomething();
const promise2 = promise.then(successCallback, failureCallback);
```

第二个 promise ( `promise2` ) 不仅表示 `doSomething()` 函数的完成，也代表了传入的 `successCallback` 或者 `failureCallback` 的完成，这两个函数也可以是返回 Promise 对象的异步函数。这样的话，在 `promise2` 上新增的排在该 promise 后面的回调函数会通过 `successCallback` 或 `failureCallback` 返回。

**备注：**如果你想要一个可以操作的示例，你可以使用下面的模板来创建任何返回 Promise 的函数：

JS

---

```
function doSomething() {
  return new Promise((resolve) => {
    setTimeout(() => {
      // 在完成 Promise 之前的其他操作
      console.log("完成了一些事情");
      // promise 的兑现值
      resolve("https://example.com/");
    }, 200);
  });
}
```

该实现会在下面的[在旧式回调 API 中创建 Promise](#)部分讨论。

就像这样，你可以创建一个更长的处理链，其中的每个 Promise 都代表了链中的一个异步过程的完成。此外，then 的参数是可选的，catch(failureCallback) 等同于 then(null, failureCallback) ——所以如果你的错误处理代码对所有步骤都是一样的，你可以把它附加到链的末尾：

JS

---

```
doSomething()
  .then(function (result) {
    return doSomethingElse(result);
  })
  .then(function (newResult) {
    return doThirdThing(newResult);
  })
  .then(function (finalResult) {
    console.log(`得到最终结果: ${finalResult}`);
  })
  .catch(failureCallback);
```

你或许会看到这种形式的[箭头函数](#)：

```
doSomething()  
  .then((result) => doSomethingElse(result))  
  .then((newResult) => doThirdThing(newResult))  
  .then((finalResult) => {  
    console.log(`得到最终结果: ${finalResult}`);  
  })  
  .catch(failureCallback);
```

**备注：**箭头函数表达式可以有隐式返回值；所以，`() => x` 是 `() => { return x; }` 的简写。

`doSomethingElse` 和 `doThirdThing` 可以返回任何值——如果它们返回的是 `Promise`，那么会首先等待这个 `Promise` 的敲定，然后下一个回调函数会接收到它的兑现值，而不是 `Promise` 本身。在 `then` 回调中始终返回 `Promise` 是非常重要的，即使 `Promise` 总是兑现为 `undefined`。如果上一个处理器启动了一个 `Promise` 但并没有返回它，那么就没有办法再追踪它的敲定状态了，这个 `Promise` 就是“漂浮”的。

```
doSomething()  
  .then((url) => {  
    // fetch(url) 前缺少 `return` 关键字。  
    fetch(url);  
  })  
  .then((result) => {  
    // result 是 undefined，因为上一个处理器没有返回任何东西。  
    // 无法得知 fetch() 的返回值，也无法知道它是否成功。  
  });
```

通过返回 `fetch` 调用的结果（一个 `Promise`），我们既可以追踪它的完成状态，也可以在它完成时接收到它的值。

```
doSomething()  
  .then((url) => {  
    // 添加 `return` 关键字
```

```
    return fetch(url);
  })
  .then((result) => {
    // result 是一个 Response 对象
  });
```

如果有竞态条件的话，使 Promise 漂浮的情况会更糟——如果上一个处理器的 Promise 没有返回，那么下一个 then 处理器会被提前调用，而它读取的任何值都可能是不完整的。

JS

---

```
const listOfIngredients = [];

doSomething()
  .then((url) => {
    // fetch(url) 前缺少 `return` 关键字。
    fetch(url)
      .then((res) => res.json())
      .then((data) => {
        listOfIngredients.push(data);
      });
  })
  .then(() => {
    console.log(listOfIngredients);
    // listOfIngredients 永远为 [], 因为 fetch 请求还没有完成。
  });
```

因此，一个经验法则是，每当你的操作遇到一个 Promise，就返回它，并把它处理推迟到下一个 then 处理器中。

JS

---

```
const listOfIngredients = [];

doSomething()
  .then((url) => {
    // fetch 调用前面现在包含了 `return` 关键字。
    return fetch(url)
      .then((res) => res.json())
      .then((data) => {
        listOfIngredients.push(data);
      });
  });
```

```
    });  
  })  
  .then(() => {  
    console.log(listOfIngredients);  
    // listOfIngredients 现在将包含来自 fetch 调用的数据。  
  });
```

更加好的解决方法是，你可以将嵌套链扁平化为单链，这样更简单，也更容易处理错误。具体细节将在下面的[嵌套](#)部分讨论。

JS

---

```
doSomething()  
  .then((url) => fetch(url))  
  .then((res) => res.json())  
  .then((data) => {  
    listOfIngredients.push(data);  
  })  
  .then(() => {  
    console.log(listOfIngredients);  
  });
```

使用 [async / await](#) 可以帮助你编写更直观、更类似同步代码的代码。下面是使用 `async / await` 的相同示例：

JS

---

```
async function logIngredients() {  
  const url = await doSomething();  
  const res = await fetch(url);  
  const data = await res.json();  
  listOfIngredients.push(data);  
  console.log(listOfIngredients);  
}
```

请注意，除了前面的 `await` 关键字外，这段代码看起来与同步代码一模一样。唯一的折衷是，可能很容易忘记 [await](#) 关键字，这只能在出现类型不匹配（例如试图将承诺作为值使用）时才能解决。

async / await 基于 promise，例如，doSomething() 与之前的函数相同，因此从 promise 到 async / await 所需的重构工作微乎其微。有关 async / await 语法的更多信息，请参阅[异步函数](#)和 [await](#) 参考。

**备注：** async/await 的并发语义与普通 Promise 链相同。异步函数中的 await 不会停止整个程序，只会停止依赖其值的部分，因此在 await 挂起时，其他异步任务仍可运行。

## 错误处理

你或许还有印象，在之前的回调地狱示例中，有 3 次 failureCallback 的调用，而在 Promise 链中只有尾部的一次调用。

JS

---

```
doSomething()
  .then((result) => doSomethingElse(result))
  .then((newResult) => doThirdThing(newResult))
  .then((finalResult) => console.log(`得到最终结果: ${finalResult}`))
  .catch(failureCallback);
```

通常，一遇到异常抛出，浏览器就会顺着 Promise 链寻找下一个 onRejected 失败回调函数或者由 .catch() 指定的回调函数。这和以下同步代码的工作原理（执行过程）非常相似。

JS

---

```
try {
  let result = syncDoSomething();
  let newResult = syncDoSomethingElse(result);
  let finalResult = syncDoThirdThing(newResult);
  console.log(`得到最终结果: ${finalResult}`);
} catch (error) {
  failureCallback(error);
}
```

这种异步代码的对称性在 [async / await](#) 语法中达到了极致：

JS

---

```
async function foo() {
  try {
    const result = await doSomething();
    const newResult = await doSomethingElse(result);
    const finalResult = await doThirdThing(newResult);
    console.log(`得到最终结果: ${finalResult}`);
  } catch (error) {
    failureCallback(error);
  }
}
```

## 嵌套

对比上述涉及 `listOfIngredients` 的两个例子，第一个例子中有一个 Promise 链嵌套在另一个 `then()` 处理器的返回值中；而第二个例子则是完全扁平化的链。简洁的 Promise 链式编程最好保持扁平化，不要嵌套 Promise，因为嵌套经常会是粗心导致的。

嵌套是一种可以限制 `catch` 语句的作用域的控制结构写法。明确来说，嵌套的 `catch` 只会捕获其作用域及以下的错误，而不会捕获链中更高层的错误。如果使用正确，可以实现细粒度的错误恢复。

JS

---

```
doSomethingCritical()
  .then((result) =>
    doSomethingOptional()
      .then((optionalResult) => doSomethingExtraNice(optionalResult))
      .catch((e) => {}),
  ) // 即便可选操作失败了，也会继续执行
  .then(() => moreCriticalStuff())
  .catch((e) => console.log(`严重失败: ${e.message}`));
```

注意，这里的可选操作是嵌套的——缩进并不是原因，而是因为可选操作被外层的 `(` 和 `)` 括号包裹起来了。

这个内部的 `catch` 语句仅能捕获到 `doSomethingOptional()` 和 `doSomethingExtraNice()` 的失败，并将该错误与外界屏蔽，之后就恢复到 `moreCriticalStuff()` 继续执行。值得注意的是，如

果 `doSomethingCritical()` 失败，这个错误仅会被最后的（外部）`catch` 语句捕获到，并不会被内部 `catch` 吞掉。

在 `async / await` 中，这段代码看起来像这样：

JS

---

```
async function main() {
  try {
    const result = await doSomethingCritical();
    try {
      const optionalResult = await doSomethingOptional(result);
      await doSomethingExtraNice(optionalResult);
    } catch (e) {
      // 忽略可选步骤的失败并继续执行。
    }
    await moreCriticalStuff();
  } catch (e) {
    console.error(`严重失败: ${e.message}`);
  }
}
```

**备注：**如果没有复杂的错误处理，则很可能不需要嵌套的 `then` 处理器。相反，可以使用扁平链，将错误处理逻辑放在最后。

## Catch 的后续链式操作

有可能会在一个回调失败之后继续使用链式操作，即，使用一个 `catch`，这对于在链式操作中抛出一个失败之后，再次进行新的操作会很有用。请阅读下面的例子：

JS

---

```
new Promise((resolve, reject) => {
  console.log("初始化");

  resolve();
})
.then(() => {
  throw new Error("哪里不对了");
});
```

```
    console.log("执行「这个」");
  })
  .catch(() => {
    console.log("执行「那个」");
  })
  .then(() => {
    console.log("执行「这个」, 无论前面发生了什么");
  });
```

输出结果如下：

初始化  
执行「那个」  
执行「这个」, 无论前面发生了什么

**备注：**并没有输出“执行「这个」”，因为在第一个 `then()` 中的 `throw` 语句导致其被拒绝。

在 `async / await` 中，这段代码看起来像这样：

JS

```
async function main() {
  try {
    await doSomething();
    throw new Error("有哪里不对了");
    console.log("执行「这个」");
  } catch (e) {
    console.error("执行「那个」");
  }
  console.log("执行「这个」, 无论前面发生了什么");
}
```

## Promise 拒绝事件

当一个 Promise 拒绝事件未被任何处理器处理时，它会冒泡到调用栈的顶部，主机需要将其暴露出来。在 Web 上，当 Promise 被拒绝时，会有下文所述的两个事件之一被派发到全局作用域（通常而言，就是 [window](#)；如果是在 web worker 中使用的话，就是 [Worker](#) 或者其他基于 worker 的接口）。这两个事件如下所示：

### [unhandledrejection](#)

当 promise 被拒绝，但没有可用的拒绝处理器时，会派发此事件。

### [rejectionhandled](#)

当一个被拒绝的 promise 在触发了 `unhandledrejection` 事件之后才附加处理器时，会派发此事件。

上述两种事件（类型为 [PromiseRejectionEvent](#)）都有两个属性，一个是 `promise` 属性，该属性指向被拒绝的 Promise，另一个是 `reason`（[英语](#)）属性，该属性用来说明 Promise 被拒绝的原因。

因此，我们可以通过以上事件为 Promise 失败时提供补偿处理，也有利于调试 Promise 相关的问题。在每一个上下文中，该处理都是全局的，因此不管源码如何，所有的错误都会在同一个处理函数中被捕捉并处理。

在 [Node.js](#) 中，对拒绝事件的处理稍有不同。你可以通过为 Node.js 的 `unhandledRejection` 事件添加处理器（注意名称的大小写不同）来捕获未处理的拒绝，就像这样：

JS

---

```
process.on("unhandledRejection", (reason, promise) => {
  /* 你可以在这里添加一些代码，以便检查 promise 和 reason */
});
```

对于 Node.js 来说，为了防止错误被记录到控制台（否则默认会发生），添加 `process.on()` 监听器就足够了；不需要类似浏览器运行时的 [preventDefault\(\)](#) 方法这样的等效操作。

然而，如果你添加了 `process.on` 监听器，但没有在其中添加代码来处理被拒绝的 Promise，那么它们就会被丢弃，而且不会有任何提示。因此，最好在监听器中添加代码来检查每个被拒绝的 Promise，并确保它们不是由于代码错误而导致的。

## 组合

有四个[组合工具](#)可用来并发异步操作：[Promise.all\(\)](#)、[Promise.allSettled\(\)](#)、[Promise.any\(\)](#)和 [Promise.race\(\)](#)。

我们可以同时启动所有操作，再等待它们全部完成，就像这样：

JS

---

```
Promise.all([func1(), func2(), func3()]).then(([result1, result2, result3]) =>
{
  /* 使用 result1、result2 和 result3 */
});
```

如果数组中的某个 Promise 被拒绝，`Promise.all()` 就会立即拒绝返回的 Promise，并终止其他操作。这可能会导致一些意外的状态或行为。[Promise.allSettled\(\)](#) 是另一个组合工具，它会等待所有操作完成后再处理返回的 Promise。

所有的这些方法都是并发运行 Promise 的——一系列 Promise 同时启动，而不是彼此等待。顺序执行也是可能的，这需要一些巧妙的 JavaScript 手段：

JS

---

```
[func1, func2, func3]
  .reduce((p, f) => p.then(f), Promise.resolve())
  .then((result3) => {
    /* 使用 result3 */
  });
```

在这个例子中，我们使用 [reduce](#) 把一个异步函数数组变为一个 Promise 链。上面的代码等同于：

JS

---

```
Promise.resolve()
  .then(func1)
  .then(func2)
  .then(func3)
  .then((result3) => {
    /* 使用 result3 */
  });
```

我们也可以写成可复用的函数形式，这在函数式编程中极为普遍：

JS

---

```
const applyAsync = (acc, val) => acc.then(val);
const composeAsync =
  (...funcs) =>
  (x) =>
    funcs.reduce(applyAsync, Promise.resolve(x));
```

`composeAsync()` 函数将会接受任意数量的函数作为其参数，并返回一个新的函数，而该函数又接受一个初始值，该组合的参数传递管线如下所示：

JS

---

```
const transformData = composeAsync(func1, func2, func3);
const result3 = transformData(data);
```

顺序组合还可以使用 `async/await` 更简洁地完成：

JS

---

```
let result;
for (const f of [func1, func2, func3]) {
  result = await f(result);
}
/* 使用最后的结果 (即 result3) */
```

然而，在你顺序组合 Promise 前，请考虑是否真的有必要——因为它们会阻塞彼此，除非一个 Promise 的执行依赖于另一个 Promise 的结果，否则最好并发运行 Promise。

## 在日式回调 API 中创建 Promise

可以通过 Promise 的构造函数从零开始创建 [Promise](#)。这种方式（通过构造函数的方式）应当只在封装旧 API 的时候用到。

理想状态下，所有的异步函数应该会返回 Promise。但有一些 API 仍然使用旧方式来传入成功（或者失败）的回调。最典型的例子就是 [setTimeout\(\).](#) 函数：

JS

---

```
setTimeout(() => saySomething("10 秒钟过去了"), 10 * 1000);
```

混用旧式回调和 Promise 可能会造成运行时序问题。如果 saySomething 函数失败了，或者包含了编程错误，那就没有办法捕获它了。这得怪 setTimeout()。

幸运的是，我们可以将 setTimeout() 封装入 Promise 内。最好的做法是，将这些有问题的函数封装起来，留在底层，并且永远不要再直接调用它们：

JS

---

```
const wait = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

wait(10 * 1000)
  .then(() => saySomething("10 秒钟"))
  .catch(failureCallback);
```

通常，Promise 的构造函数接收一个执行函数（executor），我们可以在这个执行函数里手动地解决（resolve）或拒绝（reject）一个 Promise。既然 setTimeout() 并不会真的执行失败，那么我们可以在这种情况下忽略拒绝的情况。你可以在 [Promise\(\).](#) 参考中查看更多关于执行函数的信息。

## 时序

最后，我们将深入了解更多技术细节——关于注册的回调函数何时被调用。

## 保证

在基于回调的 API 中，回调函数何时以及如何被调用取决于 API 的实现者。例如，回调可能是同步调用的，也可能是异步调用的：

JS

---

```
function doSomething(callback) {
  if (Math.random() > 0.5) {
    callback();
  } else {
    setTimeout(() => callback(), 1000);
  }
}
```

我们非常不建议使用上述这种设计，因为它会导致所谓的“Zalgo 状态”。在设计异步 API 的上下文中，这意味着回调在某些情况下是同步调用的，但在其他情况下是异步调用的，这为调用者带来的歧义。更多背景信息，请参见文章[为异步设计 API](#)，这是该术语首次被正式提出的地方。这种 API 设计使得副作用难以分析：

JS

---

```
let value = 1;
doSomething(() => {
  value = 2;
});
console.log(value); // 1 还是 2?
```

另一方面，Promise 是一种[控制反转](#)的形式——API 的实现者不控制回调何时被调用。相反，维护回调队列并决定何时调用回调的工作被委托给了 Promise 的实现者，这样一来，API 的使用者和开发者都会自动获得强大的语义保证，包括：

- 被添加到 [then\(\)](#) 的回调永远不会在 JavaScript 事件循环的[当前运行完成](#)之前被调用。
- 即使异步操作已经完成（成功或失败），在这之后通过 [then\(\)](#) 添加的回调函数也会被调用。
- 通过多次调用 [then\(\)](#) 可以添加多个回调函数，它们会按照插入顺序进行执行。

以防万一的提醒：传入 [then\(\)](#) 的函数永远不会被同步调用，即使 Promise 已经被解决了（resolved）：

JS

---

```
Promise.resolve().then(() => console.log(2));
console.log(1); // 1, 2
```

传入 `then()` 的函数不会立即运行，而是被放入微任务队列中，这意味着它会在稍后运行（仅在创建该函数的函数退出后，且 JavaScript 执行堆栈为空时），也就是在控制权返回事件循环之前。总而言之，不会等待太久：

JS

---

```
const wait = (ms) => new Promise((resolve) => setTimeout(resolve, ms));

wait().then(() => console.log(4));
Promise.resolve()
  .then(() => console.log(2))
  .then(() => console.log(3));
console.log(1); // 1, 2, 3, 4
```

## 任务队列 vs. 微任务

Promise 回调被处理为微任务，而 `setTimeout().` 回调被处理为任务队列。

JS

---

```
const promise = new Promise((resolve, reject) => {
  console.log("Promise 执行函数");
  resolve();
}).then((result) => {
  console.log("Promise 回调 (.then)");
});

setTimeout(() => {
  console.log("新一轮事件循环: Promise (已完成)", promise);
}, 0);

console.log("Promise (队列中)", promise);
```

上述代码的输出如下：

```
Promise 执行函数
Promise (队列中) Promise {<pending>}
Promise 回调 (.then)
新一轮事件循环: Promise (已完成) Promise {<fulfilled>}
```

详见[深入：微任务与 Javascript 运行时环境](#)。

## 当 Promise 与 任务冲突时

你可能遇到如下情况：你的一些 Promise 和任务（例如事件或回调）会以不可预测的顺序启动。此时，你或许可以通过使用微任务检查状态或平衡 Promise，并以此有条件地创建 Promise。

如果你认为微任务可能会帮助你解决问题，那么请阅读[微任务指南](#)，学习如何用 `queueMicrotask()` 来将一个函数作为微任务添加到队列中。

## 参见

- [Promise](#)
- [async function](#)
- [await](#)
- [Promises/A+ 规范](#)
- [我们遇到了 promise 的问题](#) pouchdb.com ( 2015 )

[上一页](#)

[下一页](#)

### Help improve MDN

Was this page helpful to you?

Yes

No

[Learn how to contribute.](#)

This page was last modified on 2025年3月8日 by [MDN contributors](#).

[View this page on GitHub](#) • [Report a problem with this content](#)



**mdn**

Your blueprint for a better internet.

MDN

About

Blog

Careers  
Advertise with us

Our communities

MDN Community

MDN Forum

MDN Chat

Support

Product help

Report an issue

Developers

Web Technologies

Learn Web Development

MDN Plus  
Hacks Blog



[Website Privacy Notice](#) [Cookies](#) [Legal](#) [Community Participation Guidelines](#)

Visit [Mozilla Corporation's](#) not-for-profit parent, the [Mozilla Foundation](#).

Portions of this content are ©1998–2025 by individual mozilla.org contributors. Content available under a [Creative Commons license](#).