> 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。

Promise

`Promise` 对象用于表示一个异步操作的最终完成（或失败）及其结果值。

> **备注：** 此特性在 Web Worker 中可用

若想了解 promise 的工作方式以及如何使用它们，我们建议你先阅读使用 promise。

描述

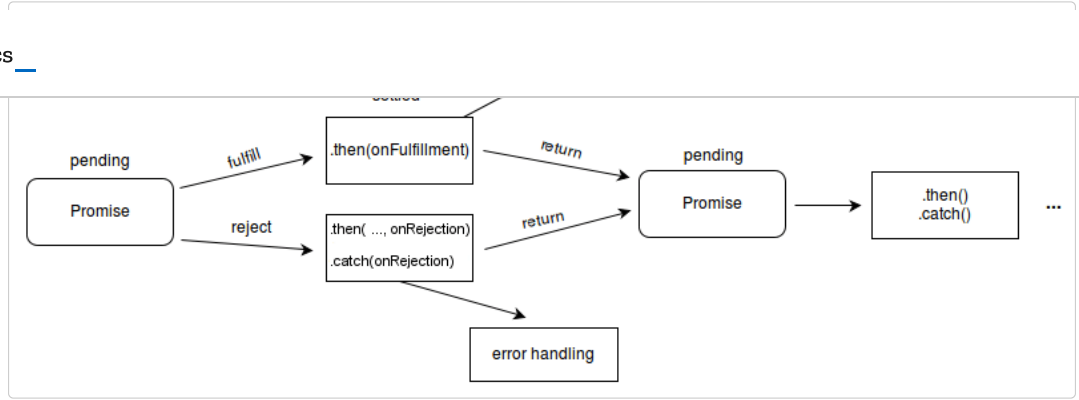
一个 `Promise` 对象代表一个在这个 promise 被创建出来时不一定已知值的代理。它让你能够把异步操作最终的成功返回值或者失败原因和相应的处理程序关联起来。这样使得异步方法可以像同步方法那样返回值：异步方法并不会立即返回最终的值，而是会返回一个 *promise*，以便在未来某个时候把值交给使用者。

一个 `Promise` 必然处于以下几种状态之一：

- *待定（pending）*：初始状态，既没有被兑现，也没有被拒绝。
- *已兑现（fulfilled）*：意味着操作成功完成。
- *已拒绝（rejected）*：意味着操作失败。

待定状态的 Promise 对象要么会通过一个值*被兑现*，要么会通过一个原因（错误）*被拒绝*。当这些情况之一发生时，我们用 promise 的 `then` 方法排列起来的相关处理程序就会被调用。如果 promise 在一个相应的处理程序被绑定时就已经被兑现或被拒绝了，那么这个处理程序也同样会被调用，因此在完成异步操作和绑定处理方法之间不存在竞态条件。

因为 `Promise.prototype.then` 和 `Promise.prototype.catch` 方法返回的是 promise，所以它们可以被链式调用。



> **备注：** 有一些语言中有惰性求值和延迟计算的特性，它们也被称为“promise”，例如 Scheme。JavaScript 中的 promise 代表的是已经在发生的进程，而且可以通过回调函数实现链式调用。如果你想对一个表达式进行惰性求值，就考虑一下使用无参数的箭头函数，如 `f = () =>`

> `expression` 来创建惰性求值的表达式，然后使用 `f()` 进行求值。

> **备注：** 如果一个 promise 已经被兑现或被拒绝，那么我们也可以说它处于 *已敲定（settled）* 状态。你还会听到一个经常跟 promise 一起使用的术语：*已决议（resolved）*，它表示 promise 已经处于已敲定状态，或者为了匹配另一个 promise 的状态被"锁定"了。Domenic Denicola 的 States and fates 中有更多关于 promise 术语的细节可以供你参考。

Promise 的链式调用

我们可以用 `Promise.prototype.then()` 、 `Promise.prototype.catch()` 和 `Promise.prototype.finally()` 这些方法将进一步的操作与一个变为已敲定状态的 promise 关联起来。

例如 `.then()` 方法需要两个参数，第一个参数作为处理已兑现状态的回调函数，而第二个参数则作为处理已拒绝状态的回调函数。每一个 `.then()` 方法还会返回一个新生成的 promise 对象，这个对象可被用作链式调用，就像这样：

```
const myPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve('foo');
  }, 300);
});

myPromise
  .then(handleResolvedA, handleRejectedA)
  .then(handleResolvedB, handleRejectedB)
  .then(handleResolvedC, handleRejectedC);
```

当 `.then()` 中缺少能够返回 promise 对象的函数时，链式调用就直接继续进行下一环操作。因此，链式调用可以在最后一个 `.catch()` 之前把所有的处理已拒绝状态的回调函数都省略掉。

过早地处理变为已拒绝状态的 promise 会对之后 promise 的链式调用造成影响。不过有时候我们因为需要马上处理一个错误也只能这样做。例如，外面必须抛出某种类型的错误以在链式调用中传递错误状态。另一方面，在没有迫切需要的情况下，可以在最后一个 `.catch()` 语句时再进行错误处理，这种做法更加简单。 `.catch()` 其实只是没有给处理已兑现状态的回调函数预留参数位置的 `.then()` 而已。

```
myPromise
  .then(handleResolvedA)
  .then(handleResolvedB)
  .then(handleResolvedC)
  .catch(handleRejectedAny);
```

使用箭头函数表达式作为 promise 回调函数的示例如下：

```
myPromise
  .then(value => { return value + ' and bar'; })
  .then(value => { return value + ' and bar again'; })
  .then(value => { return value + ' and again'; })
  .then(value => { return value + ' and again'; })
  .then(value => { console.log(value) })
  .catch(err => { console.log(err) });
```

这些函数的终止状态决定着链式调用中下一个 promise 的"已敲定"状态是什么。"已决议"状态意味着 promise 已经成功完成，而"已拒绝"则表示 promise 未成功完成。"已决议"状态的返回值会逐级传递到下一个 `.then()` 中，而"已拒绝"的理由则会被传递到链中的下一个已拒绝状态的处理函数。

链式调用中的 promise 们就像俄罗斯套娃一样，是嵌套起来的，但又像是一个栈，每个都必须从顶端被弹出。链式调用中的第一个 promise 是嵌套最深的一个，也将是第一个被弹出的。

```
(promise D, (promise C, (promise B, (promise A) ) ) )
```

当存在一个 `nextValue` 是 promise 时，就会出现一种动态的替换效果。`return` 会导致一个 promise 被弹出，但这个 `nextValue` promise 则会被推入被弹出 promise 原来的位置。对于上面所示的嵌套场景，假设与 "promise B" 相关的 `.then()` 返回了一个值为 "promise X" 的 `nextValue` 。那么嵌套的结果看起来就会是这样：

```
(promise D, (promise C, (promise X) ) )
```

一个 promise 可能会参与不止一次的嵌套。对于下面的代码， `promiseA` 向“已敲定”状态的过渡会导致两个实例的 `.then` 都被调用。

```
const promiseA = new Promise(myExecutorFunc);
const promiseB = promiseA.then(handleFulfilled1, handleRejected1);
const promiseC = promiseA.then(handleFulfilled2, handleRejected2);
```

一个已经处于“已敲定”状态的 promise 也可以接收操作。在那种情况下，（如果没有问题的话）这个操作会被作为第一个异步操作被执行。注意，所有的 promise 都一定是异步的。因此，一个已经处于“已敲定”状态的 promise 中的操作只有 promise 链式调用的栈被清空且一个时间片段过去之后才会被执行。这种效果跟 `setTimeout(action, 10)` 特别相似。

```
const promiseA = new Promise( (resolutionFunc,rejectionFunc) => {
    resolutionFunc(777);
});
// At this point, "promiseA" is already settled.
promiseA.then( (val) => console.log("asynchronous logging has val:",val) );
console.log("immediate logging");

// produces output in this order:
// immediate logging
// asynchronous logging has val: 777
```

追踪现有设置对象

设置对象（settings object）是 JavaScript 代码运行时用于提供附加信息的环境 。它包含了领域（realm）和模块映射（module map），以及 HTML 的特定信息，如来源（origin）等。对现有设置对象的追踪保证了浏览器知道用户给定的哪些代码片段需要使用。

为了更好地说明这一点，我们在这里进一步探讨领域是如何引发问题的。我们可以粗略地认为**领域**是一个全局对象。其独特之处在于，它拥有运行 JavaScript 代码所需的所有信息。这包括像 `Array` 和 `Error` 这样的对象。每一个设置对象都有自己的“副本”，而且它们与副本之间是不共享的。这可能会导致一些与 promise 相关的意外行为。为了解决这个问题，我们需要追踪**现有设置对象**（incumbent settings object）。它表示负责用户某个函数调用工作的特定信息。

我们可以尝试在文档中嵌入 `<iframe>` ，并让其与父级上下文通信。由于所有的 web API 都有现有设置对象，下面的代码能够在所有的浏览器中运行：

```
<!DOCTYPE html>
<iframe></iframe> <!-- we have a realm here -->
<script> // we have a realm here as well
  const bound = frames[0].postMessage.bind(
    frames[0], "some data", "*");
    // bound is a built-in function -- there is no user
    // code on the stack, so which realm do we use?
  window.setTimeout(bound);
  // this still works, because we use the youngest
  // realm (the incumbent) on the stack
</script>
```

同样的概念也适用与 promise。如果我们稍加修改上面的示例，我们就能得到这个：

```
<!DOCTYPE html>
<iframe></iframe> <!-- we have a realm here -->
<script> // we have a realm here as well
  const bound = frames[0].postMessage.bind(
    frames[0], "some data", "*");
    // bound is a built in function -- there is no user
    // code on the stack -- which realm do we use?
  Promise.resolve(undefined).then(bound);
  // this still works, because we use the youngest
  // realm (the incumbent) on the stack
</script>
```

如果我们修改代码，使用文档中的 `<iframe>` 来监听发送的消息，我们可以观察到现有设置对象的影响：

```
<!-- y.html -->
<!DOCTYPE html>
<iframe src="x.html"></iframe>
<script>
  const bound = frames[0].postMessage.bind(frames[0], "some data", "*");
  Promise.resolve(undefined).then(bound);
</script>
```

```
<!-- x.html -->
<!DOCTYPE html>
<script>
window.addEventListener("message", (event) => {
  document.querySelector("#text").textContent = "hello";
  // 这一部分代码仅在追踪现有设置对象的浏览器中会被运行
  console.log(event);
}, false);
</script>
```

在上面的示例中， `<iframe>` 仅在现有设置对象被追踪时才会被更新。这是因为在不追踪的情况下，我们可能会使用错误的环境发送消息。

> **备注：** 目前，Firefox 完全实现了现有领域追踪，Chrome 和 Safari 仅部分实现。

构造函数

`Promise()`

创建一个新的 `Promise` 对象。该构造函数主要用于包装还没有添加 promise 支持的函数。

静态方法

`Promise.all(iterable)`

这个方法返回一个新的 promise 对象，等到所有的 promise 对象都成功或有任意一个 promise 失败。

如果所有的 promise 都成功了，它会把一个包含 iterable 里所有 promise 返回值的数组作为成功回调的返回值。顺序跟 iterable 的顺序保持一致。

一旦有任意一个 iterable 里面的 promise 对象失败则立即以该 promise 对象失败的理由来拒绝这个新的 promise。

`Promise.allSettled(iterable)`

等到所有 promise 都已敲定（每个 promise 都已兑现或已拒绝）。

返回一个 promise，该 promise 在所有 promise 都敲定后完成，并兑现一个对象数组，其中的对象对应每个 promise 的结果。

`Promise.any(iterable)`

接收一个 promise 对象的集合，当其中的任意一个 promise 成功，就返回那个成功的 promise 的值。

`Promise.race(iterable)`

等到任意一个 promise 的状态变为已敲定。

当 iterable 参数里的任意一个子 promise 成功或失败后，父 promise 马上也会用子 promise 的成功返回值或失败详情作为参数调用父 promise 绑定的相应处理函数，并返回该 promise 对象。

`Promise.reject(reason)`

返回一个状态为已拒绝的 `Promise` 对象，并将给定的失败信息传递给对应的处理函数。

`Promise.resolve(value)`

返回一个状态由给定 value 决定的 `Promise` 对象。如果该值是 thenable（即，带有 `then` 方法的对象），返回的 Promise 对象的最终状态由 then 方法执行结果决定；否则，返回的 Promise 对象状态为已兑现，并且将该 value 传递给对应的 then 方法。

通常而言，如果你不知道一个值是否是 promise 对象，使用 `Promise.resolve(value)` 来返回一个 Promise 对象，这样就能将该 value 以 promise 对象形式使用。

实例方法

参阅微任务指南以了解有关这些方法如何使用为任务队列和服务。

`Promise.prototype.catch()`

为 promise 添加一个被拒绝状态的回调函数，并返回一个新的 promise，若回调函数被调用，则兑现其返回值，否则兑现原来的 promise 兑现的值。

`Promise.prototype.then()`

为 promise 添加被兑现和被拒绝状态的回调函数，其以回调函数的返回值兑现 promise。若不处理已兑现或者已拒绝状态（例如，`onFulfilled` 或 `onRejected` 不是一个函数），则返回 promise 被敲定时的值。

`Promise.prototype.finally()`

为 promise 添加一个回调函数，并返回一个新的 promise。这个新的 promise 将在原 promise 被兑现时兑现。而传入的回调函数将在原 promise 被敲定（无论被兑现还是被拒绝）时被调用。

示例

基础示例

```
let myFirstPromise = new Promise((resolve, reject) => {
  // We call resolve(...) when what we were doing asynchronously was successful, and reject(...) when it failed.
  // In this example, we use setTimeout(...) to simulate async code.
  // In reality, you will probably be using something like XHR or an HTML5 API.
  setTimeout( function() {
    resolve("Success!")  // Yay! Everything went well!
  }, 250)
})

myFirstPromise.then((successMessage) => {
  // successMessage is whatever we passed in the resolve(...) function above.
  // It doesn't have to be a string, but if it is only a succeed message, it probably will be.
```

```
  console.log("Yay! " + successMessage)
});
```

不同场景的示例

此示例展示了使用 promise 的多种方法，以及其可能发生的多种情况。要理解这一点，首先滚动到代码块的底部，然后查看 promise 调用链。在创建初始的 primise 后，可以接上一条 promise 调用链。该调用链由 `.then()` 组成，通常（但不一定）在末尾会有一个 `.catch()`，并可能会接上一个 `.finnaly()`。在本示例中，promise 调用链是由一个自定义的 `new Promise()` 构造并发起的；但在实践中，promise 调用链通常由一个 API 函数（由其他人编写的）返回的 promise 开始。

示例函数 `tetheredGetNumber()` 会在设置同步调用或者函数内部抛出异常时调用 `reject()`。函数 `promiseGetWord()` 展示了如何在 API 函数内部创建并返回一个 promise。

请注意，函数 `troubleWithGetNumber()` 以 `throw()` 结束。这是强制的做法，因为 ES6 的 promsie 会遍历所有的 `.then` promise，在遇到错误时，如果不使用 `throw()`，这个错误会被当作“已修复”。这很麻烦，因此，通常会在 `.then()` promise 调用链中忽略 `rejectionFunc`，而仅在最后的 `.catch()` 中保留一个 `rejectionFunc`。另一种方法是抛出一个特殊值（本例使用了 `-999`，但使用自定义错误类型更合适）。

示例代码可以在 NodeJS 下运行。请通过查看实际发生的错误来理解代码。若要提高错误发生的概率，请该改变 `threshold` 的值。

```
"use strict";

// To experiment with error handling, "threshold" values cause errors randomly
const THRESHOLD_A = 8; // can use zero 0 to guarantee error

function tetheredGetNumber(resolve, reject) {
  try {
    setTimeout(
      function() {
        const randomInt = Date.now();
        const value = randomInt % 10;
        try {
          if(value >= THRESHOLD_A) {
            throw new Error(`Too large: ${value}`);
          }
        } catch(msg) {
            reject(`Error in callback ${msg}`);
        }
      resolve(value);
      return;
    }, 500);
    // To experiment with error at set-up, uncomment the following 'throw'.
    // throw new Error("Bad setup");
  } catch(err) {
    reject(`Error during setup: ${err}`);
  }
  return;
}

function determineParity(value) {
  const isOdd = value % 2 ? true : false ;
  const parityInfo = { theNumber: value, isOdd: isOdd };
  return parityInfo;
}

function troubleWithGetNumber(reason) {
  console.error(`Trouble getting number: ${reason}`);
  throw -999; // must "throw" something, to maintain error state down the chain
}
```

```
function promiseGetWord(parityInfo) {
  // The "tetheredGetWord()" function gets "parityInfo" as closure variable.
  const tetheredGetWord = function(resolve,reject) {
    const theNumber = parityInfo.theNumber;
    const threshold_B = THRESHOLD_A - 1;
    if(theNumber >= threshold_B) {
      reject(`Still too large: ${theNumber}`);
    } else {
      parityInfo.wordEvenOdd = parityInfo.isOdd ? 'odd' : 'even';
      resolve(parityInfo);
    }
    return;
  }
  return new Promise(tetheredGetWord);
}

(new Promise(tetheredGetNumber))
  .then(determineParity,troubleWithGetNumber)
  .then(promiseGetWord)
  .then((info) => {
    console.log("Got: ",info.theNumber," , ", info.wordEvenOdd);
    return info;
  })
  .catch((reason) => {
    if(reason === -999) {
      console.error("Had previously handled error");
    }
    else {
      console.error(`Trouble with promiseGetWord(): ${reason}`);
    }
   })
  .finally((info) => console.log("All done"));
```

高级示例

本例展示了 `Promise` 的一些机制。`testPromise()` 方法在每次点击 `<button>` 按钮时被调用，该方法会创建一个 promise 对象，使用 `setTimeout()` 让 `Promise` 等待 1-3 秒不等的时间来兑现计数结果（从 1 开始的数字）。使用 `Promise` 构造函数来创建 promise。

promise 的值的兑现过程都被日志记录（logged，使用 `p1.then()`）下来。这些日志信息展示了方法中的同步代码和异步代码是如何通过 promise 完成解耦的。

通过在短时间内多次点击按钮，你可以看到不同的 promise 被一个接一个地兑现。

HTML

```
<button id="make-promise">Make a promise!</button>
<div id="log"></div>
```

JavaScript

```
"use strict";
let promiseCount = 0;

function testPromise() {
  let thisPromiseCount = ++promiseCount;
  let log = document.getElementById('log');
  // begin
  log.insertAdjacentHTML('beforeend', thisPromiseCount + ') Started<br>');
  // We make a new promise: we promise a numeric count of this promise, starting from 1 (after waiting 3s)
```

```
  let p1 = new Promise((resolve, reject) => {
    // The executor function is called with the ability to resolve or reject the promise
    log.insertAdjacentHTML('beforeend', thisPromiseCount + ') Promise constructor<br>');
    // This is only an example to create asynchronism
    window.setTimeout(function() {
        // We fulfill the promise !
        resolve(thisPromiseCount);
    }, Math.random() * 2000 + 1000);
  });

  // We define what to do when the promise is resolved with the then() call,
  // and what to do when the promise is rejected with the catch() call
  p1.then(function(val) {
    // Log the fulfillment value
    log.insertAdjacentHTML('beforeend', val + ') Promise fulfilled<br>');
  }).catch((reason) => {
    // Log the rejection reason
    console.log(`Handle rejected promise (${reason}) here.`);
  });
  // end
  log.insertAdjacentHTML('beforeend', thisPromiseCount + ') Promise made<br>');
}

if ("Promise" in window) {
  let btn = document.getElementById("make-promise");
  btn.addEventListener("click",testPromise);
} else {
  log = document.getElementById('log');
  log.textContent = "Live example not available as your browser doesn't support the <code>Promise<code> interface.";
}
```

结果

Make a promise!

使用 XHR 加载图像

另一个使用 `Promise` 和 `XMLHttpRequest` 加载一个图像的例子可在 MDN GitHub js-examples 仓库中找到。你也可以看它的实例 。每一步都有注释可以让你详细的了解 Promise 和 XHR 架构。

规范

Specification
ECMAScript Language Specification # sec-promise-objects

浏览器兼容性

Report problems with this compatibility data on GitHub

	Chrome	Edge	Firefox	Opera	Safari	Chrome Android	Firefox for Android	Opera Android
Promise	Chrome 32	Edge 12	Firefox 29	Opera 19	Safari 8	Chrome 32 Android	Firefox for Android 29	Opera Androic
Promise() constructor	Chrome 32	Edge 12	Firefox 29	Opera 19	Safari 8	Chrome 32 Android	Firefox for Android 29	Opera Androic
all()	Chrome 32	Edge 12	Firefox 29	Opera 19	Safari 8	Chrome 32 Android	Firefox for Android 29	Opera Androic
allSettled()	Chrome 76	Edge 79	Firefox 71	Opera 63	Safari 13	Chrome 76 Android	Firefox for Android 79	Opera Android
any	Chrome 85	Edge 85	Firefox 79	Opera 71	Safari 14	Chrome 85 Android	Firefox for Android 79	Opera Android
catch()	Chrome 32	Edge 12	Firefox 29	Opera 19	Safari 8	Chrome 32 Android	Firefox for Android 29	Opera Androic
finally()	Chrome 63	Edge 18	Firefox 58	Opera 50	Safari 11.1	Chrome 63 Android	Firefox for Android 58	Opera Android
Incumbent settings object tracking	Chrome No	Edge No	Firefox 50	Opera No	Safari No	Chrome No Android	Firefox for Android 50	Opera Android
race()	Chrome 32	Edge 12	Firefox 29	Opera 19	Safari 8	Chrome 32 Android	Firefox for Android 29	Opera Androic
reject()	Chrome 32	Edge 12	Firefox 29	Opera 19	Safari 8	Chrome 32 Android	Firefox for Android 29	Opera Androic
resolve()	Chrome 32	Edge 12	Firefox 29	Opera 19	Safari 8	Chrome 32 Android	Firefox for Android 29	Opera Androic
then()	Chrome 32	Edge 12	Firefox 29	Opera 19	Safari 8	Chrome 32 Android	Firefox for Android 29	Opera Androic

Tip: you can click/tap on a cell for more information.

Full support No support See implementation notes.

参见

- core-js 中 Promise 的 Polyfill
- 使用 promise
- Promises/A+ 规范
- JavaScript Promises：简介
- Domenic Denicola：回调、Promise 和协程——JavaScript 中的异步编程模式

Last modified: 2022年11月27日, by MDN contributors