

1: promise

可以将异步的操作以同步的流程表达出来，避免了层层嵌套

## 2: ES6 promise对象

### Promise的含义

Promise是异步编程的一种解决方案，**比传统的解决方案——回调函数和事件——更合理和更强大**。它由社区最早提出和实现，ES6将其写进了语言标准，统一了用法，原生提供了Promise对象。

所谓Promise，**简单说就是一个容器，里面保存着某个未来才会结束的事件**（通常是一个异步操作）的结果。从语法上说，**Promise是一个对象，从它可以获取异步操作的消息**。Promise提供统一的API，各种异步操作都可以用同样的方法进行处理。

Promise对象有以下两个特点。

(1) 对象的状态不受外界影响。Promise对象代表一个异步操作，**有三种状态**：Pending（进行中）、Resolved（已完成，又称Fulfilled）和Rejected（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是Promise这个名字的由来，它的英语意思就是“承诺”，表示其他手段无法改变。

(2) 一旦状态改变，就不会再变，任何时候都可以得到这个结果。Promise对象的状态改变，只有两种可能：从Pending变为Resolved和从Pending变为Rejected。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果。就算改变已经发生了，你再对Promise对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

有了Promise对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，Promise对象提供统一的接口，使得控制异步操作更加容易。

Promise也有一些缺点。首先，无法取消Promise，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，Promise内部抛出的错误，不会反应到外

部。第三，当处于Pending状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

如果某些事件不断地反复发生，一般来说，使用stream模式是比部署Promise更好的选择。

## 基本用法：

0: Promise的实例一创建就会执行，如果把他放在函数里面，就可以按需执行

1: Promise构造函数接受一个函数作为参数，该函数的两个参数分别是resolve和reject。它们是两个函数，由JavaScript引擎提供，不用自己部署。

2: 可以用then方法分别指定resolved状态和rejected状态的回调函数。

3: then方法可以接受两个回调函数作为参数。第一个回调函数是Promise对象的状态变为Resolved时调用，第二个回调函数是Promise对象的状态变为Reject时调用。其中，第二个函数是可选的，不一定要提供。

4: 这两个回调函数的参数是----Promise对象传出的值作为参数。

5: Promise 新建后就会立即执行。

6: then方法指定的回调函数，将在当前脚本所有同步任务执行完才会执行

```
7 : // good
    promise
    .then(function(data) { //cb
      // success
    })
    .catch(function(err) {
      // error
    });
```

```
promise.then(function (param) { })
    .catch(function (param) { })
    .finally(()=>{})
```

上面代码中，不管promise最后的状态，在执行完then或catch指定的回调函数以后，都会执行finally方法指定的回调函数。

finally方法的回调函数不接受任何参数，这意味着没有办法知道，前面的 Promise 状态到底是fulfilled还是rejected。这表明，finally方法里面的操作，应该是与状态无

关的，不依赖于 Promise 的执行结果。

`finally` 本质上是 `then` 方法的特例。

```
promise
  .finally(() => {
    // 语句
  });

// 等同于
promise
  .then(
    result => {
      // 语句
      return result;
    },
    error => {
      // 语句
      throw error;
    }
  );
```

1.

[Symbol](#)

2. [Set 和 Map 数据结构](#)
3. [Proxy](#)
4. [Reflect](#)
5. [Promise 对象](#)
6. [Iterator 和 for...of 循环](#)
7. [Generator 函数的语法](#)
8. [Generator 函数的异步应用](#)
9. [async 函数](#)
10. [Class 的基本语法](#)
11. [Class 的继承](#)
12. [Module 的语法](#)
13. [Module 的加载实现](#)
14. [编程风格](#)
15. [读懂规格](#)
16. [异步遍历器](#)
17. [ArrayBuffer](#)
18. [最新提案](#)
19. [Decorator](#)
20. [参考链接](#)

其他

- [源码](#)
- [修订历史](#)
- [反馈意见](#)

# Promise 对象

1. [Promise 的含义](#)
2. [基本用法](#)
3. [Promise.prototype.then\(\)](#)
4. [Promise.prototype.catch\(\)](#)

5. [Promise.prototype.finally\(\)](#)

6. [Promise.all\(\)](#)

7. [Promise.race\(\)](#)

8. [Promise.resolve\(\)](#)

9. [Promise.reject\(\)](#)

10. [应用](#)

11. [Promise.try\(\)](#)

## Promise 的含义

Promise 是异步编程的一种解决方案，比传统的解决方案——回调函数和事件——更合理和更强大。它由社区最早提出和实现，ES6 将其写进了语言标准，统一了用法，原生提供了 `Promise` 对象。

所谓 `Promise`，简单说就是一个容器，里面保存着某个未来才会结束的事件（通常是一个异步操作）的结果。从语法上说，`Promise` 是一个对象，从它可以获取异步操作的消息。`Promise` 提供统一的 API，各种异步操作都可以用同样的方法进行处理。

`Promise` 对象有以下两个特点。

(1) 对象的状态不受外界影响。`Promise` 对象代表一个异步操作，有三种状态：

`pending`（进行中）、`fulfilled`（已成功）和 `rejected`（已失败）。只有异步操作的结果，可以决定当前是哪一种状态，任何其他操作都无法改变这个状态。这也是 `Promise` 这个名字的由来，它的英语意思就是“承诺”，表示其他手段无法改变。

(2) 一旦状态改变，就不会再变，任何时候都可以得到这个结果。`Promise` 对象的状态改变，只有两种可能：从 `pending` 变为 `fulfilled` 和从 `pending` 变为 `rejected`。只要这两种情况发生，状态就凝固了，不会再变了，会一直保持这个结果，这时就称为 `resolved`（已定型）。如果改变已经发生了，你再对 `Promise` 对象添加回调函数，也会立即得到这个结果。这与事件（Event）完全不同，事件的特点是，如果你错过了它，再去监听，是得不到结果的。

注意，为了行文方便，本章后面的 `resolved` 统一只指 `fulfilled` 状态，不包含 `rejected` 状态。

有了 `Promise` 对象，就可以将异步操作以同步操作的流程表达出来，避免了层层嵌套的回调函数。此外，`Promise` 对象提供统一的接口，使得控制异步操作更加容易。

`Promise` 也有一些缺点。首先，无法取消 `Promise`，一旦新建它就会立即执行，无法中途取消。其次，如果不设置回调函数，`Promise` 内部抛出的错误，不会反应到外部。第三，当处于 `pending` 状态时，无法得知目前进展到哪一个阶段（刚刚开始还是即将完成）。

如果某些事件不断地反复发生，一般来说，使用 [Stream](#) 模式是比部署 `Promise` 更好的选择。

## 基本用法

ES6 规定，`Promise` 对象是一个构造函数，用来生成 `Promise` 实例。

下面代码创造了一个 `Promise` 实例。

```
const promise = new Promise(function(resolve, reject) {  
  // ... some code  
  
  if (/* 异步操作成功 */) {  
    resolve(value);  
  } else {  
    reject(error);  
  }  
});
```

`Promise` 构造函数接受一个函数作为参数，该函数的两个参数分别是 `resolve` 和 `reject`。它们是两个函数，由 JavaScript 引擎提供，不用自己部署。

`resolve` 函数的作用是，将 `Promise` 对象的状态从“未完成”变为“成功”（即从 `pending` 变为 `resolved`），在异步操作成功时调用，并将异步操作的结果，作为参数传递出去；`reject` 函数的作用是，将 `Promise` 对象的状态从“未完成”变为“失败”（即从 `pending` 变为 `rejected`），在异步操作失败时调用，并将异步操作报出的错误，作为参数传递出去。

`Promise` 实例生成以后，可以用 `then` 方法分别指定 `resolved` 状态和 `rejected` 状态的回调函数。

```
promise.then(function(value) {  
  // success  
}, function(error) {  
  // failure  
});
```

`then` 方法可以接受两个回调函数作为参数。第一个回调函数是 `Promise` 对象的状态变为 `resolved` 时调用，第二个回调函数是 `Promise` 对象的状态变为 `rejected` 时调用。其中，第二个函数是可选的，不一定要提供。这两个函数都接受 `Promise` 对象传出的值作为参数。

下面是一个 `Promise` 对象的简单例子。

```
function timeout(ms) {  
  return new Promise((resolve, reject) => {  
    setTimeout(resolve, ms, 'done');  
  });  
}  
  
timeout(100).then((value) => {  
  console.log(value);  
});
```

上面代码中，`timeout` 方法返回一个 `Promise` 实例，表示一段时间以后才会发生的结果。过了指定的时间（`ms` 参数）以后，`Promise` 实例的状态变为 `resolved`，就会触发 `then` 方法绑定的回调函数。

`Promise` 新建后就会立即执行。

---

```

let promise = new Promise(function(resolve, reject) {
  console.log('Promise');
  resolve();
});

promise.then(function() {
  console.log('resolved.');
```

```

});

console.log('Hi!');

// Promise
// Hi!
// resolved
```

上面代码中，Promise 新建后立即执行，所以首先输出的是 Promise。然后，then 方法指定的回调函数，将在当前脚本所有同步任务执行完才会执行，所以 resolved 最后输出。

下面是异步加载图片的例子。

```

function loadImageAsync(url) {
  return new Promise(function(resolve, reject) {
    const image = new Image();

    image.onload = function() {
      resolve(image);
    };

    image.onerror = function() {
      reject(new Error('Could not load image at ' + url));
    };

    image.src = url;
  });
}
```

上面代码中，使用 Promise 包装了一个图片加载的异步操作。如果加载成功，就调用 resolve 方法，否则就调用 reject 方法。

下面是一个用 Promise 对象实现的 Ajax 操作的例子。

```

const getJSON = function(url) {
  const promise = new Promise(function(resolve, reject){
    const handler = function() {
      if (this.readyState !== 4) {
        return;
      }
      if (this.status === 200) {
        resolve(this.response);
      } else {
        reject(new Error(this.statusText));
      }
    };
    const client = new XMLHttpRequest();
    client.open("GET", url);
    client.onreadystatechange = handler;
    client.responseType = "json";
    client.setRequestHeader("Accept", "application/json");
    client.send();

  });

  return promise;
};
```

```

getJSON("/posts.json").then(function(json) {
  console.log('Contents: ' + json);
});
```

```

}, function(error) {
  console.error('出错了', error);
});

```

上面代码中，`getJSON`是对 `XMLHttpRequest` 对象的封装，用于发出一个针对 JSON 数据的 HTTP 请求，并且返回一个 `Promise` 对象。需要注意的是，在 `getJSON` 内部，`resolve` 函数和 `reject` 函数调用时，都带有参数。

如果调用 `resolve` 函数和 `reject` 函数时带有参数，那么它们的参数会被传递给回调函数。`reject` 函数的参数通常是 `Error` 对象的实例，表示抛出的错误；`resolve` 函数的参数除了正常的值以外，还可能是另一个 `Promise` 实例，比如像下面这样。

```

const p1 = new Promise(function (resolve, reject) {
  // ...
});

```

```

const p2 = new Promise(function (resolve, reject) {
  // ...
  resolve(p1);
});

```

上面代码中，`p1`和`p2`都是 `Promise` 的实例，但是`p2`的`resolve`方法将`p1`作为参数，即一个异步操作的结果是返回另一个异步操作。

注意，这时`p1`的状态就会传递给`p2`，也就是说，`p1`的状态决定了`p2`的状态。如果`p1`的状态是`pending`，那么`p2`的回调函数就会等待`p1`的状态改变；如果`p1`的状态已经是`resolved`或者`rejected`，那么`p2`的回调函数将会立刻执行。

```

const p1 = new Promise(function (resolve, reject) {
  setTimeout(() => reject(new Error('fail')), 3000)
});

```

```

const p2 = new Promise(function (resolve, reject) {
  setTimeout(() => resolve(p1), 1000)
});

```

```

p2
  .then(result => console.log(result))
  .catch(error => console.log(error))
// Error: fail

```

上面代码中，`p1`是一个 `Promise`，3 秒之后变为`rejected`。`p2`的状态在 1 秒之后改变，`resolve`方法返回的是`p1`。由于`p2`返回的是另一个 `Promise`，导致`p2`自己的状态无效了，由`p1`的状态决定`p2`的状态。所以，后面的`then`语句都变成针对后者（`p1`）。又过了 2 秒，`p1`变为`rejected`，导致触发`catch`方法指定的回调函数。

注意，调用`resolve`或`reject`并不会终结 `Promise` 的参数函数的执行。

```

new Promise((resolve, reject) => {
  resolve(1);
  console.log(2);
}).then(r => {
  console.log(r);
});
// 2
// 1

```

上面代码中，调用`resolve(1)`以后，后面的`console.log(2)`还是会执行，并且会首先打印出来。这是因为立即 `resolved` 的 `Promise` 是在本轮事件循环的末尾执行，总是晚

于本轮循环的同步任务。

一般来说，调用 `resolve` 或 `reject` 以后，`Promise` 的使命就完成了，后继操作应该放到 `then` 方法里面，而不应该直接写在 `resolve` 或 `reject` 的后面。所以，最好在它们前面加上 `return` 语句，这样就不会有意外。

```
new Promise((resolve, reject) => {  
  return resolve(1);  
  // 后面的语句不会执行  
  console.log(2);  
}))
```

## Promise.prototype.then()

`Promise` 实例具有 `then` 方法，也就是说，`then` 方法是定义在原型对象 `Promise.prototype` 上的。它的作用是为 `Promise` 实例添加状态改变时的回调函数。前面说过，`then` 方法的第一个参数是 `resolved` 状态的回调函数，第二个参数（可选）是 `rejected` 状态的回调函数。

`then` 方法返回的是一个新的 `Promise` 实例（注意，不是原来那个 `Promise` 实例）。因此可以采用链式写法，即 `then` 方法后面再调用另一个 `then` 方法。

```
getJSON("/posts.json").then(function(json) {  
  return json.post;  
}).then(function(post) {  
  // ...  
});
```

上面的代码使用 `then` 方法，依次指定了两个回调函数。第一个回调函数完成以后，会将返回结果作为参数，传入第二个回调函数。

采用链式的 `then`，可以指定一组按照次序调用的回调函数。这时，前一个回调函数，有可能返回的还是一个 `Promise` 对象（即有异步操作），这时后一个回调函数，就会等待该 `Promise` 对象的状态发生变化，才会被调用。

```
getJSON("/post/1.json").then(function(post) {  
  return getJSON(post.commentURL);  
}).then(function (comments) {  
  console.log("resolved: ", comments);  
}, function (err) {  
  console.log("rejected: ", err);  
});
```

上面代码中，第一个 `then` 方法指定的回调函数，返回的是另一个 `Promise` 对象。这时，第二个 `then` 方法指定的回调函数，就会等待这个新的 `Promise` 对象状态发生变化。如果变为 `resolved`，就调用第一个回调函数，如果状态变为 `rejected`，就调用第二个回调函数。

如果采用箭头函数，上面的代码可以写得更简洁。

```
getJSON("/post/1.json").then(  
  post => getJSON(post.commentURL)  
)  
.then(  
  comments => console.log("resolved: ", comments),  
  err => console.log("rejected: ", err)  
);
```



## Promise.prototype.catch()

`Promise.prototype.catch`方法是`.then(null, rejection)`或`.then(undefined, rejection)`的别名，用于指定发生错误时的回调函数。

```
getJSON('/posts.json').then(function(posts) {  
  // ...  
}).catch(function(error) {  
  // 处理 getJSON 和 前一个回调函数运行时发生的错误  
  console.log('发生错误!', error);  
});
```

上面代码中，`getJSON`方法返回一个 `Promise` 对象，如果该对象状态变为`resolved`，则会调用`then`方法指定的回调函数；如果异步操作抛出错误，状态就会变为`rejected`，就会调用`catch`方法指定的回调函数，处理这个错误。另外，`then`方法指定的回调函数，如果运行中抛出错误，也会被`catch`方法捕获。

```
p.then((val) => console.log('fulfilled:', val))  
  .catch((err) => console.log('rejected', err));
```

```
// 等同于  
p.then((val) => console.log('fulfilled:', val))  
  .then(null, (err) => console.log("rejected:", err));
```

下面是一个例子。

```
const promise = new Promise(function(resolve, reject) {  
  throw new Error('test');  
});  
promise.catch(function(error) {  
  console.log(error);  
});  
// Error: test
```

上面代码中，`promise`抛出一个错误，就被`catch`方法指定的回调函数捕获。注意，上面的写法与下面两种写法是等价的。

```
// 写法一  
const promise = new Promise(function(resolve, reject) {  
  try {  
    throw new Error('test');  
  } catch(e) {  
    reject(e);  
  }  
});  
promise.catch(function(error) {  
  console.log(error);  
});
```

```
// 写法二  
const promise = new Promise(function(resolve, reject) {  
  reject(new Error('test'));  
});  
promise.catch(function(error) {  
  console.log(error);  
});
```

比较上面两种写法，可以发现`reject`方法的作用，等同于抛出错误。

如果 `Promise` 状态已经变成`resolved`，再抛出错误是无效的。

```
const promise = new Promise(function(resolve, reject) {  
  resolve('ok');  
  throw new Error('test');  
});  
promise  
  .then(function(value) { console.log(value) })
```

```
.catch(function(error) { console.log(error) });
// ok
```

上面代码中，Promise 在 `resolve` 语句后面，再抛出错误，不会被捕获，等于没有抛出。因为 Promise 的状态一旦改变，就永久保持该状态，不会再变了。

Promise 对象的错误具有“冒泡”性质，会一直向后传递，直到被捕获为止。也就是说，错误总是会被下一个 `catch` 语句捕获。

```
getJSON('/post/1.json').then(function(post) {
  return getJSON(post.commentURL);
}).then(function(comments) {
  // some code
}).catch(function(error) {
  // 处理前面三个Promise产生的错误
});
```

上面代码中，一共有三个 Promise 对象：一个由 `getJSON` 产生，两个由 `then` 产生。它们之中任何一个抛出的错误，都会被最后一个 `catch` 捕获。

一般来说，不要在 `then` 方法里面定义 Reject 状态的回调函数（即 `then` 的第二个参数），总是使用 `catch` 方法。

```
// bad
promise
  .then(function(data) {
    // success
  }, function(err) {
    // error
  });
```

```
// good
promise
  .then(function(data) { //cb
    // success
  })
  .catch(function(err) {
    // error
  });
```

上面代码中，第二种写法要好于第一种写法，理由是第二种写法可以捕获前面 `then` 方法执行中的错误，也更接近同步的写法（`try/catch`）。因此，建议总是使用 `catch` 方法，而不使用 `then` 方法的第二个参数。

跟传统的 `try/catch` 代码块不同的是，如果没有使用 `catch` 方法指定错误处理的回调函数，Promise 对象抛出的错误不会传递到外层代码，即不会有任何反应。

```
const someAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    // 下面一行会报错，因为x没有声明
    resolve(x + 2);
  });
};

someAsyncThing().then(function() {
  console.log('everything is great');
});

setTimeout(() => { console.log(123) }, 2000);
// Uncaught (in promise) ReferenceError: x is not defined
// 123
```

上面代码中，`someAsyncThing`函数产生的 Promise 对象，内部有语法错误。浏览器运行到这一行，会打印出错误提示`ReferenceError: x is not defined`，但是不会退出进程、终止脚本执行，2 秒之后还是会输出`123`。这就是说，Promise 内部的错误不会影响到 Promise 外部的代码，通俗的说法就是“Promise 会吃掉错误”。

这个脚本放在服务器执行，退出码就是`0`（即表示执行成功）。不过，Node 有一个 `unhandledRejection` 事件，专门监听未捕获的 `reject` 错误，上面的脚本会触发这个事件的监听函数，可以在监听函数里面抛出错误。

```
process.on('unhandledRejection', function (err, p) {  
  throw err;  
});
```

上面代码中，`unhandledRejection`事件的监听函数有两个参数，第一个是错误对象，第二个是报错的 Promise 实例，它可以用来了解发生错误的环境信息。

注意，Node 有计划在未来废除`unhandledRejection`事件。如果 Promise 内部有未捕获的错误，会直接终止进程，并且进程的退出码不为 `0`。

再看下面的例子。

```
const promise = new Promise(function (resolve, reject) {  
  resolve('ok');  
  setTimeout(function () { throw new Error('test') }, 0)  
});  
promise.then(function (value) { console.log(value) });  
// ok  
// Uncaught Error: test
```

上面代码中，Promise 指定在下一轮“事件循环”再抛出错误。到了那个时候，Promise 的运行已经结束了，所以这个错误是在 Promise 函数体外抛出的，会冒泡到最外层，成了未捕获的错误。

一般总是建议，Promise 对象后面要跟`catch`方法，这样可以处理 Promise 内部发生的错误。`catch`方法返回的还是一个 Promise 对象，因此后面还可以接着调用`then`方法。

```
const someAsyncThing = function() {  
  return new Promise(function(resolve, reject) {  
    // 下面一行会报错，因为x没有声明  
    resolve(x + 2);  
  });  
};
```

```

someAsyncThing()
.catch(function(error) {
  console.log('oh no', error);
})
.then(function() {
  console.log('carry on');
});
// oh no [ReferenceError: x is not defined]
// carry on

```

上面代码运行完`catch`方法指定的回调函数，会接着运行后面那个`then`方法指定的回调函数。如果没有报错，则会跳过`catch`方法。

```

Promise.resolve()
.catch(function(error) {
  console.log('oh no', error);
})
.then(function() {
  console.log('carry on');
});
// carry on

```

上面的代码因为没有报错，跳过了`catch`方法，直接执行后面的`then`方法。此时，要是`then`方法里面报错，就与前面的`catch`无关了。

`catch`方法之中，还能再抛出错误。

```

const someAsyncThing = function() {
  return new Promise(function(resolve, reject) {
    // 下面一行会报错，因为x没有声明
    resolve(x + 2);
  });
};

```

```

someAsyncThing().then(function() {
  return someOtherAsyncThing();
}).catch(function(error) {
  console.log('oh no', error);
  // 下面一行会报错，因为 y 没有声明
  y + 2;
}).then(function() {
  console.log('carry on');
});
// oh no [ReferenceError: x is not defined]

```

上面代码中，`catch`方法抛出一个错误，因为后面没有别的`catch`方法了，导致这个错误不会被捕获，也不会传递到外层。如果改写一下，结果就不一样了。

```
someAsyncThing().then(function() {  
  return someOtherAsyncThing();  
}).catch(function(error) {  
  console.log('oh no', error);  
  // 下面一行会报错，因为y没有声明  
  y + 2;  
}).catch(function(error) {  
  console.log('carry on', error);  
});  
// oh no [ReferenceError: x is not defined]  
// carry on [ReferenceError: y is not defined]
```

上面代码中，第二个`catch`方法用来捕获前一个`catch`方法抛出的错误。

## Promise.prototype.finally()

`finally`方法用于指定不管 Promise 对象最后状态如何，都会执行的操作。该方法是 ES2018 引入标准的。

```
promise  
  .then(result => { ... })  
  .catch(error => { ... })  
  .finally(() => { ... });
```

上面代码中，不管`promise`最后的状态，在执行完`then`或`catch`指定的回调函数以后，都会执行`finally`方法指定的回调函数。

下面是一个例子，服务器使用 Promise 处理请求，然后使用`finally`方法关掉服务器。

```
server.listen(port)  
  .then(function () {  
    // ...  
  })  
  .finally(server.stop);
```

`finally`方法的回调函数不接受任何参数，这意味着没有办法知道，前面的 Promise 状态到底是`fulfilled`还是`rejected`。这表明，`finally`方法里面的操作，应该是与状态无关的，不依赖于 Promise 的执行结果。

`finally`本质上是`then`方法的特例。

```
promise  
  .finally(() => {  
    // 语句
```

```
});
```

```
// 等同于
promise
.then(
  result => {
    // 语句
    return result;
  },
  error => {
    // 语句
    throw error;
  }
);
```

上面代码中，如果不使用`finally`方法，同样的语句需要为成功和失败两种情况各写一次。有了`finally`方法，则只需要写一次。

它的实现也很简单。

```
Promise.prototype.finally = function (callback) {
  let P = this.constructor;
  return this.then(
    value => P.resolve(callback()).then(() => value),
    reason => P.resolve(callback()).then(() => { throw
reason })
  );
};
```

上面代码中，不管前面的 Promise 是`fulfilled`还是`rejected`，都会执行回调函数`callback`。

从上面的实现还可以看到，`finally`方法总是会返回原来的值。

```
// resolve 的值是 undefined
Promise.resolve(2).then(() => {}, () => {})
```

```
// resolve 的值是 2
Promise.resolve(2).finally(() => {})
```

```
// reject 的值是 undefined
Promise.reject(3).then(() => {}, () => {})
```

```
// reject 的值是 3
Promise.reject(3).finally(() => {})
```

**Promise.all()**

`Promise.all`方法用于将多个 Promise 实例，包装成一个新的 Promise 实例。

```
// console.log(hw.next())
let i = Promise.resolve("aaa");
let i2=Promise.resolve("bbb");
let i3=Promise.resolve("ccc");
Promise.all([i,i2,i3]).then(res=>{
  let [res3,res1,res2]=res;
  console.log(res1,res2,res3);
})
```