# Promise 对象

1. **[Promise 的含义](http://es6.ruanyifeng.com/" \l "docs/promise#Promise 的含义)**
2. **[基本用法](http://es6.ruanyifeng.com/" \l "docs/promise#基本用法)**
3. **[Promise.prototype.then()](http://es6.ruanyifeng.com/" \l "docs/promise#Promise.prototype.then())**
4. **[Promise.prototype.catch()](http://es6.ruanyifeng.com/" \l "docs/promise#Promise.prototype.catch())**
5. **[Promise.all()](http://es6.ruanyifeng.com/" \l "docs/promise#Promise.all())**
6. **[Promise.race()](http://es6.ruanyifeng.com/" \l "docs/promise#Promise.race())**
7. **[Promise.resolve()](http://es6.ruanyifeng.com/" \l "docs/promise#Promise.resolve())**
8. **[Promise.reject()](http://es6.ruanyifeng.com/" \l "docs/promise#Promise.reject())**
9. **[两个有用的附加方法](http://es6.ruanyifeng.com/" \l "docs/promise#两个有用的附加方法)**
10. **[应用](http://es6.ruanyifeng.com/" \l "docs/promise#应用)**
11. **[Promise.try()](http://es6.ruanyifeng.com/" \l "docs/promise#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](https://nodejs.org/api/stream.html) 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。

var 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) {

var 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。

var getJSON = function(url) {

var promise = new Promise(function(resolve, reject){

var client = new XMLHttpRequest();

client.open("GET", url);

client.onreadystatechange = handler;

client.responseType = "json";

client.setRequestHeader("Accept", "application/json");

client.send();

function handler() {

if (this.readyState !== 4) {

return;

}

if (this.status === 200) {

resolve(this.response);

} else {

reject(new Error(this.statusText));

}

};

});

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。

var p1 = new Promise(function (resolve, reject) {

// ...});

var 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。

var p1 = new Promise(function (resolve, reject) {

setTimeout(() => reject(new Error('fail')), 3000)})

var 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 funcA(comments) {

console.log("resolved: ", comments);}, function funcB(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funcA，如果状态变为rejected，就调用funcB。

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)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。

var 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var 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var 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。

var 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。

// badpromise

.then(function(data) {

// success }, function(err) {

// error });

// goodpromise

.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。

var 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。

var 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。

var 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.all()

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

var p = Promise.all([p1, p2, p3]);

上面代码中，Promise.all方法接受一个数组作为参数，p1、p2、p3都是 Promise 实例，如果不是，就会先调用下面讲到的Promise.resolve方法，将参数转为 Promise 实例，再进一步处理。（Promise.all方法的参数可以不是数组，但必须具有 Iterator 接口，且返回的每个成员都是 Promise 实例。）

p的状态由p1、p2、p3决定，分成两种情况。

（1）只有p1、p2、p3的状态都变成fulfilled，p的状态才会变成fulfilled，此时p1、p2、p3的返回值组成一个数组，传递给p的回调函数。

（2）只要p1、p2、p3之中有一个被rejected，p的状态就变成rejected，此时第一个被reject的实例的返回值，会传递给p的回调函数。

下面是一个具体的例子。

// 生成一个Promise对象的数组var promises = [2, 3, 5, 7, 11, 13].map(function (id) {

return getJSON('/post/' + id + ".json");});

Promise.all(promises).then(function (posts) {

// ...}).catch(function(reason){

// ...});

上面代码中，promises是包含6个 Promise 实例的数组，只有这6个实例的状态都变成fulfilled，或者其中有一个变为rejected，才会调用Promise.all方法后面的回调函数。

下面是另一个例子。

const databasePromise = connectDatabase();

const booksPromise = databasePromise

.then(findAllBooks);

const userPromise = databasePromise

.then(getCurrentUser);

Promise.all([

booksPromise,

userPromise]).then(([books, user]) => pickTopRecommentations(books, user));

上面代码中，booksPromise和userPromise是两个异步操作，只有等到它们的结果都返回了，才会触发pickTopRecommentations这个回调函数。

注意，如果作为参数的 Promise 实例，自己定义了catch方法，那么它一旦被rejected，并不会触发Promise.all()的catch方法。

const p1 = new Promise((resolve, reject) => {

resolve('hello');}).then(result => result).catch(e => e);

const p2 = new Promise((resolve, reject) => {

throw new Error('报错了');}).then(result => result).catch(e => e);

Promise.all([p1, p2]).then(result => console.log(result)).catch(e => console.log(e));

// ["hello", Error: 报错了]

上面代码中，p1会resolved，p2首先会rejected，但是p2有自己的catch方法，该方法返回的是一个新的 Promise 实例，p2指向的实际上是这个实例。该实例执行完catch方法后，也会变成resolved，导致Promise.all()方法参数里面的两个实例都会resolved，因此会调用then方法指定的回调函数，而不会调用catch方法指定的回调函数。

如果p2没有自己的catch方法，就会调用Promise.all()的catch方法。

const p1 = new Promise((resolve, reject) => {

resolve('hello');}).then(result => result);

const p2 = new Promise((resolve, reject) => {

throw new Error('报错了');}).then(result => result);

Promise.all([p1, p2]).then(result => console.log(result)).catch(e => console.log(e));

// Error: 报错了

## Promise.race()

Promise.race方法同样是将多个Promise实例，包装成一个新的Promise实例。

var p = Promise.race([p1, p2, p3]);

上面代码中，只要p1、p2、p3之中有一个实例率先改变状态，p的状态就跟着改变。那个率先改变的 Promise 实例的返回值，就传递给p的回调函数。

Promise.race方法的参数与Promise.all方法一样，如果不是 Promise 实例，就会先调用下面讲到的Promise.resolve方法，将参数转为 Promise 实例，再进一步处理。

下面是一个例子，如果指定时间内没有获得结果，就将Promise的状态变为reject，否则变为resolve。

const p = Promise.race([

fetch('/resource-that-may-take-a-while'),

new Promise(function (resolve, reject) {

setTimeout(() => reject(new Error('request timeout')), 5000)

})]);

p.then(response => console.log(response));

p.catch(error => console.log(error));

上面代码中，如果5秒之内fetch方法无法返回结果，变量p的状态就会变为rejected，从而触发catch方法指定的回调函数。

## Promise.resolve()

有时需要将现有对象转为Promise对象，Promise.resolve方法就起到这个作用。

var jsPromise = Promise.resolve($.ajax('/whatever.json'));

上面代码将jQuery生成的deferred对象，转为一个新的Promise对象。

Promise.resolve等价于下面的写法。

Promise.resolve('foo')

// 等价于new Promise(resolve => resolve('foo'))

Promise.resolve方法的参数分成四种情况。

****（1）参数是一个Promise实例****

如果参数是Promise实例，那么Promise.resolve将不做任何修改、原封不动地返回这个实例。

****（2）参数是一个****thenable****对象****

thenable对象指的是具有then方法的对象，比如下面这个对象。

let thenable = {

then: function(resolve, reject) {

resolve(42);

}};

Promise.resolve方法会将这个对象转为Promise对象，然后就立即执行thenable对象的then方法。

let thenable = {

then: function(resolve, reject) {

resolve(42);

}};

let p1 = Promise.resolve(thenable);

p1.then(function(value) {

console.log(value); // 42});

上面代码中，thenable对象的then方法执行后，对象p1的状态就变为resolved，从而立即执行最后那个then方法指定的回调函数，输出42。

****（3）参数不是具有****then****方法的对象，或根本就不是对象****

如果参数是一个原始值，或者是一个不具有then方法的对象，则Promise.resolve方法返回一个新的Promise对象，状态为resolved。

var p = Promise.resolve('Hello');

p.then(function (s){

console.log(s)});

// Hello

上面代码生成一个新的Promise对象的实例p。由于字符串Hello不属于异步操作（判断方法是字符串对象不具有then方法），返回Promise实例的状态从一生成就是resolved，所以回调函数会立即执行。Promise.resolve方法的参数，会同时传给回调函数。

****（4）不带有任何参数****

Promise.resolve方法允许调用时不带参数，直接返回一个resolved状态的Promise对象。

所以，如果希望得到一个Promise对象，比较方便的方法就是直接调用Promise.resolve方法。

var p = Promise.resolve();

p.then(function () {

// ...});

上面代码的变量p就是一个Promise对象。

需要注意的是，立即resolve的Promise对象，是在本轮“事件循环”（event loop）的结束时，而不是在下一轮“事件循环”的开始时。

setTimeout(function () {

console.log('three');}, 0);

Promise.resolve().then(function () {

console.log('two');});

console.log('one');

// one// two// three

上面代码中，setTimeout(fn, 0)在下一轮“事件循环”开始时执行，Promise.resolve()在本轮“事件循环”结束时执行，console.log('one')则是立即执行，因此最先输出。

## Promise.reject()

Promise.reject(reason)方法也会返回一个新的 Promise 实例，该实例的状态为rejected。

var p = Promise.reject('出错了');

// 等同于var p = new Promise((resolve, reject) => reject('出错了'))

p.then(null, function (s) {

console.log(s)});

// 出错了

上面代码生成一个Promise对象的实例p，状态为rejected，回调函数会立即执行。

注意，Promise.reject()方法的参数，会原封不动地作为reject的理由，变成后续方法的参数。这一点与Promise.resolve方法不一致。

const thenable = {

then(resolve, reject) {

reject('出错了');

}};

Promise.reject(thenable).catch(e => {

console.log(e === thenable)})

// true

上面代码中，Promise.reject方法的参数是一个thenable对象，执行以后，后面catch方法的参数不是reject抛出的“出错了”这个字符串，而是thenable对象。

## 两个有用的附加方法

ES6的Promise API提供的方法不是很多，有些有用的方法可以自己部署。下面介绍如何部署两个不在ES6之中、但很有用的方法。

### done()

Promise对象的回调链，不管以then方法或catch方法结尾，要是最后一个方法抛出错误，都有可能无法捕捉到（因为Promise内部的错误不会冒泡到全局）。因此，我们可以提供一个done方法，总是处于回调链的尾端，保证抛出任何可能出现的错误。

asyncFunc()

.then(f1)

.catch(r1)

.then(f2)

.done();

它的实现代码相当简单。

Promise.prototype.done = function (onFulfilled, onRejected) {

this.then(onFulfilled, onRejected)

.catch(function (reason) {

// 抛出一个全局错误 setTimeout(() => { throw reason }, 0);

});};

从上面代码可见，done方法的使用，可以像then方法那样用，提供fulfilled和rejected状态的回调函数，也可以不提供任何参数。但不管怎样，done都会捕捉到任何可能出现的错误，并向全局抛出。

### 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。它与done方法的最大区别，它接受一个普通的回调函数作为参数，该函数不管怎样都必须执行。

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(0)

.then(function () {

// run test })

.finally(server.stop);

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。

## 应用

### 加载图片

我们可以将图片的加载写成一个Promise，一旦加载完成，Promise的状态就发生变化。

const preloadImage = function (path) {

return new Promise(function (resolve, reject) {

var image = new Image();

image.onload = resolve;

image.onerror = reject;

image.src = path;

});};

### Generator函数与Promise的结合

使用Generator函数管理流程，遇到异步操作的时候，通常返回一个Promise对象。

function getFoo () {

return new Promise(function (resolve, reject){

resolve('foo');

});}

var g = function\* () {

try {

var foo = yield getFoo();

console.log(foo);

} catch (e) {

console.log(e);

}};

function run (generator) {

var it = generator();

function go(result) {

if (result.done) return result.value;

return result.value.then(function (value) {

return go(it.next(value));

}, function (error) {

return go(it.throw(error));

});

}

go(it.next());}

run(g);

上面代码的Generator函数g之中，有一个异步操作getFoo，它返回的就是一个Promise对象。函数run用来处理这个Promise对象，并调用下一个next方法。

## Promise.try()

实际开发中，经常遇到一种情况：不知道或者不想区分，函数f是同步函数还是异步操作，但是想用 Promise 来处理它。因为这样就可以不管f是否包含异步操作，都用then方法指定下一步流程，用catch方法处理f抛出的错误。一般就会采用下面的写法。

Promise.resolve().then(f)

上面的写法有一个缺点，就是如果f是同步函数，那么它会在本轮事件循环的末尾执行。

const f = () => console.log('now');

Promise.resolve().then(f);

console.log('next');

// next// now

上面代码中，函数f是同步的，但是用 Promise 包装了以后，就变成异步执行了。

那么有没有一种方法，让同步函数同步执行，异步函数异步执行，并且让它们具有统一的 API 呢？回答是可以的，并且还有两种写法。第一种写法是用async函数来写。

const f = () => console.log('now');(async () => f())();

console.log('next');

// now// next

上面代码中，第二行是一个立即执行的匿名函数，会立即执行里面的async函数，因此如果f是同步的，就会得到同步的结果；如果f是异步的，就可以用then指定下一步，就像下面的写法。

(async () => f())().then(...)

需要注意的是，async () => f()会吃掉f()抛出的错误。所以，如果想捕获错误，要使用promise.catch方法。

(async () => f())().then(...).catch(...)

第二种写法是使用new Promise()。

const f = () => console.log('now');(

() => new Promise(

resolve => resolve(f())

))();

console.log('next');

// now// next

上面代码也是使用立即执行的匿名函数，执行new Promise()。这种情况下，同步函数也是同步执行的。

鉴于这是一个很常见的需求，所以现在有一个[提案](https://github.com/ljharb/proposal-promise-try)，提供Promise.try方法替代上面的写法。

const f = () => console.log('now');

Promise.try(f);

console.log('next');

// now// next

事实上，Promise.try存在已久，Promise 库[Bluebird](http://bluebirdjs.com/docs/api/promise.try.html)、[Q](https://github.com/kriskowal/q/wiki/API-Reference" \l "promisefcallargs)和[when](https://github.com/cujojs/when/blob/master/docs/api.md" \l "whentry)，早就提供了这个方法。

由于Promise.try为所有操作提供了统一的处理机制，所以如果想用then方法管理流程，最好都用Promise.try包装一下。这样有[许多好处](http://cryto.net/~joepie91/blog/2016/05/11/what-is-promise-try-and-why-does-it-matter/)，其中一点就是可以更好地管理异常。

function getUsername(userId) {

return database.users.get({id: userId})

.then(function(user) {

return user.name;

});}

上面代码中，database.users.get()返回一个 Promise 对象，如果抛出异步错误，可以用catch方法捕获，就像下面这样写。

database.users.get({id: userId}).then(...).catch(...)

但是database.users.get()可能还会抛出同步错误（比如数据库连接错误，具体要看实现方法），这时你就不得不用try...catch去捕获。

try {

database.users.get({id: userId})

.then(...)

.catch(...)} catch (e) {

// ...}

上面这样的写法就很笨拙了，这时就可以统一用promise.catch()捕获所有同步和异步的错误。

Promise.try(database.users.get({id: userId}))

.then(...)

.catch(...)

事实上，Promise.try就是模拟try代码块，就像promise.catch模拟的是catch代码块。