**JavaScript Promise 对象**

*分类*[**编程技术**](https://www.runoob.com/w3cnote_genre/code)

ECMAscript 6 原生提供了 Promise 对象。

Promise 对象代表了未来将要发生的事件，用来传递异步操作的消息。

**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: 初始状态，不是成功或失败状态。
* fulfilled: 意味着操作成功完成。
* rejected: 意味着操作失败。

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。

**Promise 优缺点**

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）。

**Promise 创建**

要想创建一个 promise 对象、可以使用 new 来调用 Promise 的构造器来进行实例化。

下面是创建 promise 的步骤：

var promise = new Promise(function(resolve, reject) { // 异步处理 // 处理结束后、调用resolve 或 reject });

Promise 构造函数包含一个参数和一个带有 resolve（解析）和 reject（拒绝）两个参数的回调。在回调中执行一些操作（例如异步），如果一切都正常，则调用 resolve，否则调用 reject。

**实例**

var myFirstPromise = new Promise(function(resolve, reject){ //当异步代码执行成功时，我们才会调用resolve(...), 当异步代码失败时就会调用reject(...) //在本例中，我们使用setTimeout(...)来模拟异步代码，实际编码时可能是XHR请求或是HTML5的一些API方法. setTimeout(function(){ resolve("成功!"); //代码正常执行！ }, 250); }); myFirstPromise.then(function(successMessage){ //successMessage的值是上面调用resolve(...)方法传入的值. //successMessage参数不一定非要是字符串类型，这里只是举个例子 document.write("Yay! " + successMessage); });

[**尝试一下 »**](https://c.runoob.com/codedemo/3343)

对于已经实例化过的 promise 对象可以调用 promise.then() 方法，传递 resolve 和 reject 方法作为回调。

promise.then() 是 promise 最为常用的方法。

promise.then(onFulfilled, onRejected)

promise简化了对error的处理，上面的代码我们也可以这样写：

promise.then(onFulfilled).catch(onRejected)

**Promise Ajax**

下面是一个用 Promise 对象实现的 Ajax 操作的例子。

**实例**

function ajax(URL) { return new Promise(function (resolve, reject) { var req = new XMLHttpRequest(); req.open('GET', URL, true); req.onload = function () { if (req.status === 200) { resolve(req.responseText); } else { reject(new Error(req.statusText)); } }; req.onerror = function () { reject(new Error(req.statusText)); }; req.send(); }); } var URL = "/try/ajax/testpromise.php"; ajax(URL).then(function onFulfilled(value){ document.write('内容是：' + value); }).catch(function onRejected(error){ document.write('错误：' + error); });

[**尝试一下 »**](https://www.runoob.com/try/try.php?filename=tryjs_promise1)

上面代码中，resolve 方法和 reject 方法调用时，都带有参数。它们的参数会被传递给回调函数。reject 方法的参数通常是 Error 对象的实例，而 resolve 方法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){ // ... some code }); var p2 = new Promise(function(resolve, reject){ // ... some code 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，这时 p1 的状态就会传递给 p2。如果调用的时候，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 fulfilled 或者 rejected，那么 p2 的回调函数将会立刻执行。

**Promise.prototype.then方法：链式操作**

Promise.prototype.then 方法返回的是一个新的 Promise 对象，因此可以采用链式写法。

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

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。

如果前一个回调函数返回的是Promise对象，这时后一个回调函数就会等待该Promise对象有了运行结果，才会进一步调用。

getJSON("/post/1.json").then(function(post) { return getJSON(post.commentURL); }).then(function(comments) { // 对comments进行处理 });

这种设计使得嵌套的异步操作，可以被很容易得改写，从回调函数的"横向发展"改为"向下发展"。

**Promise.prototype.catch方法：捕捉错误**

Promise.prototype.catch 方法是 Promise.prototype.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) { // some code }).catch(function(error) { // 处理前一个回调函数运行时发生的错误 console.log('发生错误！', error); });

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) { // 处理前两个回调函数的错误 });

**Promise.all方法，Promise.race方法**

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 对象的实例。（Promise.all 方法的参数不一定是数组，但是必须具有 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){ // ... });

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的返回值。

如果Promise.all方法和Promise.race方法的参数，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。

**Promise.resolve 方法，Promise.reject 方法**

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 deferred 对象，转为一个新的 ES6 的 Promise 对象。

如果 Promise.resolve 方法的参数，不是具有 then 方法的对象（又称 thenable 对象），则返回一个新的 Promise 对象，且它的状态为fulfilled。

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，它的状态为fulfilled，所以回调函数会立即执行，Promise.resolve方法的参数就是回调函数的参数。

如果Promise.resolve方法的参数是一个Promise对象的实例，则会被原封不动地返回。

Promise.reject(reason)方法也会返回一个新的Promise实例，该实例的状态为rejected。Promise.reject方法的参数reason，会被传递给实例的回调函数。

var p = Promise.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，状态为rejected，回调函数会立即执行。