# Promise 对象

## 概述

Promise 对象是 JavaScript 的异步操作解决方案，为异步操作提供统一接口。它起到代理作用（proxy），充当异步操作与回调函数之间的中介，使得异步操作具备同步操作的接口。Promise 可以让异步操作写起来，就像在写同步操作的流程，而不必一层层地嵌套回调函数。

注意，本章只是 Promise 对象的简单介绍。为了避免与后续教程的重复，更完整的介绍请看[《ES6 标准入门》](http://es6.ruanyifeng.com/)的[《Promise 对象》](http://es6.ruanyifeng.com/#docs/promise)一章。

首先，Promise 是一个对象，也是一个构造函数。

```javascript
function f1(resolve, reject) {
  // 异步代码...
}

var p1 = new Promise(f1);
```

上面代码中，`Promise`构造函数接受一个回调函数`f1`作为参数，`f1`里面是异步操作的代码。然后，返回的`p1`就是一个 Promise 实例。

Promise 的设计思想是，所有异步任务都返回一个 Promise 实例。Promise 实例有一个`then`方法，用来指定下一步的回调函数。

```javascript
var p1 = new Promise(f1);
p1.then(f2);
```

上面代码中，`f1`的异步操作执行完成，就会执行`f2`。

传统的写法可能需要把`f2`作为回调函数传入`f1`，比如写成`f1(f2)`，异步操作完成后，在`f1`内部调用`f2`。Promise 使得`f1`和`f2`变成了链式写法。不仅改善了可读性，而且对于多层嵌套的回调函数尤其方便。

```javascript
// 传统写法
step1(function (value1) {
  step2(value1, function(value2) {
    step3(value2, function(value3) {
      step4(value3, function(value4) {
        // ...
      });
```

```
    });
  });
});

// Promise 的写法
(new Promise(step1))
  .then(step2)
  .then(step3)
  .then(step4);
```

从上面代码可以看到，采用 Promises 以后，程序流程变得非常清楚，十分易读。注意，为了便于理解，上面代码的`Promise`实例的生成格式，做了简化，真正的语法请参照下文。

总的来说，传统的回调函数写法使得代码混成一团，变得横向发展而不是向下发展。Promise 就是解决这个问题，使得异步流程可以写成同步流程。

Promise 原本只是社区提出的一个构想，一些函数库率先实现了这个功能。ECMAScript 6 将其写入语言标准，目前 JavaScript 原生支持 Promise 对象。

## Promise 对象的状态

Promise 对象通过自身的状态，来控制异步操作。Promise 实例具有三种状态。

- 异步操作未完成（pending）
- 异步操作成功（fulfilled）
- 异步操作失败（rejected）

上面三种状态里面，`fulfilled`和`rejected`合在一起称为`resolved`（已定型）。

这三种的状态的变化途径只有两种。

- 从“未完成”到“成功”
- 从“未完成”到“失败”

一旦状态发生变化，就凝固了，不会再有新的状态变化。这也是 Promise 这个名字的由来，它的英语意思是“承诺”，一旦承诺成效，就不得再改变了。这也意味着，Promise 实例的状态变化只可能发生一次。

因此，Promise 的最终结果只有两种。

- 异步操作成功，Promise 实例传回一个值（value），状态变为`fulfilled`。
- 异步操作失败，Promise 实例抛出一个错误（error），状态变为`rejected`。

## Promise 构造函数

JavaScript 提供原生的`Promise`构造函数，用来生成 Promise 实例。

```javascript
var promise = new Promise(function (resolve, reject) {
  // ...

  if (/* 异步操作成功 */){
    resolve(value);
  } else { /* 异步操作失败 */
    reject(new Error());
  }
});
```

上面代码中，`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，不用自己实现。

`resolve`函数的作用是，将`Promise`实例的状态从“未完成”变为“成功”（即从`pending`变为`fulfilled`），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`实例的状态从“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。

下面是一个例子。

```javascript
function timeout(ms) {
  return new Promise((resolve, reject) => {
    setTimeout(resolve, ms, 'done');
  });
}

timeout(100)
```

上面代码中，`timeout(100)`返回一个 Promise 实例。100毫秒以后，该实例的状态会变为`fulfilled`。

## Promise.prototype.then()

Promise 实例的`then`方法，用来添加回调函数。

`then`方法可以接受两个回调函数，第一个是异步操作成功时（变为`fulfilled`状态）的回调函数，第二个是异步操作失败（变为`rejected`）时的回调函数（该参数可以省略）。一旦状态改变，就调用相应的回调函数。

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

```
  resolve('成功');
});
p1.then(console.log, console.error);
// "成功"

var p2 = new Promise(function (resolve, reject) {
  reject(new Error('失败'));
});
p2.then(console.log, console.error);
// Error: 失败
```

上面代码中，`p1`和`p2`都是Promise 实例，它们的`then`方法绑定两个回调函数：成功时的回调函数`console.log`，失败时的回调函数`console.error`（可以省略）。`p1`的状态变为成功，`p2`的状态变为失败，对应的回调函数会收到异步操作传回的值，然后在控制台输出。

`then`方法可以链式使用。

```javascript
p1
  .then(step1)
  .then(step2)
  .then(step3)
  .then(
    console.log,
    console.error
  );
```

上面代码中，`p1`后面有四个`then`，意味依次有四个回调函数。只要前一步的状态变为`fulfilled`，就会依次执行紧跟在后面的回调函数。

最后一个`then`方法，回调函数是`console.log`和`console.error`，用法上有一点重要的区别。`console.log`只显示`step3`的返回值，而`console.error`可以显示`p1`、`step1`、`step2`、`step3`之中任意一个发生的错误。举例来说，如果`step1`的状态变为`rejected`，那么`step2`和`step3`都不会执行了（因为它们是`resolved`的回调函数）。Promise 开始寻找，接下来第一个为`rejected`的回调函数，在上面代码中是`console.error`。这就是说，Promise 对象的报错具有传递性。

## then() 用法辨析

Promise 的用法，简单说就是一句话：使用`then`方法添加回调函数。但是，不同的写法有一些细微的差别，请看下面四种写法，它们的差别在哪里？

```javascript
// 写法一
f1().then(function () {
  return f2();
```

```
});

// 写法二
f1().then(function () {
  f2();
});

// 写法三
f1().then(f2());

// 写法四
f1().then(f2);
```

为了便于讲解，下面这四种写法都再用`then`方法接一个回调函数`f3`。写法一的`f3`回调函数的参数，是`f2`函数的运行结果。

```javascript
f1().then(function () {
  return f2();
}).then(f3);
```

写法二的`f3`回调函数的参数是`undefined`。

```javascript
f1().then(function () {
  f2();
  return;
}).then(f3);
```

写法三的`f3`回调函数的参数，是`f2`函数返回的函数的运行结果。

```javascript
f1().then(f2())
  .then(f3);
```

写法四与写法一只有一个差别，那就是`f2`会接收到`f1()`返回的结果。

```javascript
f1().then(f2)
  .then(f3);
```

## 实例：图片加载

下面是使用 Promise 完成图片的加载。

```javascript
var preloadImage = function (path) {
  return new Promise(function (resolve, reject) {
    var image = new Image();
    image.onload  = resolve;
    image.onerror = reject;
    image.src = path;
  });
};
```

上面代码中，`image`是一个图片对象的实例。它有两个事件监听属性，`onload`属性在图片加载成功后调用，`onerror`属性在加载失败调用。

上面的`preloadImage()`函数用法如下。

```javascript
preloadImage('https://example.com/my.jpg')
  .then(function (e) { document.body.append(e.target) })
  .then(function () { console.log('加载成功') })
```

上面代码中，图片加载成功以后，`onload`属性会返回一个事件对象，因此第一个`then()`方法的回调函数，会接收到这个事件对象。该对象的`target`属性就是图片加载后生成的 DOM 节点。

## 小结

Promise 的优点在于，让回调函数变成了规范的链式写法，程序流程可以看得很清楚。它有一整套接口，可以实现许多强大的功能，比如同时执行多个异步操作，等到它们的状态都改变以后，再执行一个回调函数；再比如，为多个回调函数中抛出的错误，统一指定处理方法等等。

而且，Promise 还有一个传统写法没有的好处：它的状态一旦改变，无论何时查询，都能得到这个状态。这意味着，无论何时为 Promise 实例添加回调函数，该函数都能正确执行。所以，你不用担心是否错过了某个事件或信号。如果是传统写法，通过监听事件来执行回调函数，一旦错过了事件，再添加回调函数是不会执行的。

Promise 的缺点是，编写的难度比传统写法高，而且阅读代码也不是一眼可以看懂。你只会看到一堆`then`，必须自己在`then`的回调函数里面理清逻辑。

## 微任务

Promise 的回调函数属于异步任务，会在同步任务之后执行。

```javascript
new Promise(function (resolve, reject) {
  resolve(1);
}).then(console.log);
```

```
console.log(2);
// 2
// 1
```

上面代码会先输出2，再输出1。因为`console.log(2)`是同步任务，而`then`的回调函数属于异步任务，一定晚于同步任务执行。

但是，Promise 的回调函数不是正常的异步任务，而是微任务（microtask）。它们的区别在于，正常任务追加到下一轮事件循环，微任务追加到本轮事件循环。这意味着，微任务的执行时间一定早于正常任务。

```javascript
setTimeout(function() {
  console.log(1);
}, 0);

new Promise(function (resolve, reject) {
  resolve(2);
}).then(console.log);

console.log(3);
// 3
// 2
// 1
```

上面代码的输出结果是`321`。这说明`then`的回调函数的执行时间，早于`setTimeout(fn, 0)`。因为`then`是本轮事件循环执行，`setTimeout(fn, 0)`在下一轮事件循环开始时执行。

## 参考链接


- Sebastian Porto, [Asynchronous JS: Callbacks, Listeners, Control Flow Libs and Promises](http://sporto.github.com/blog/2012/12/09/callbacks-listeners-promises/)
- Rhys Brett-Bowen, [Promises/A+ - understanding the spec through implementation](http://modernjavascript.blogspot.com/2013/08/promisesa-understanding-by-doing.html)
- Matt Podwysocki, Amanda Silver, [Asynchronous Programming in JavaScript with “Promises”](http://blogs.msdn.com/b/ie/archive/2011/09/11/asynchronous-programming-in-javascript-with-promises.aspx)
- Marc Harter, [Promise A+ Implementation](https://gist.github.com//wavded/5692344)
- Bryan Klimt, [What’s so great about JavaScript Promises?](http://blog.parse.com/2013/01/29/whats-so-great-about-javascript-promises/)
- Jake Archibald, [JavaScript Promises There and back again](http://www.html5rocks.com/en/tutorials/es6/promises/)
- Mikito Takada, [7. Control flow, Mixu's Node book](http://book.mixu.net/node/ch7.html)