

Promise 对象

1. Promise 的含义

Promise 是一个对象，从它可以获取异步操作的消息，它提供统一的接口，使得控制异步操作更加容易，避免了层层嵌套的回调函数（回调地域）。

Promise 对象有以下两个特点：

- 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。
- 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，再对 **Promise** 对象添加回调函数，也会立即得到这个结果。

Promise 也有一些缺点：

- 无法取消 **Promise**，一旦新建它就会立即执行，无法中途取消。
- 如果不设置回调函数，**Promise** 内部抛出的错误，不会反应到外部。
- 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）。

2. 基本用法

Promise 对象是一个构造函数，用来生成 **Promise** 实例。它接受一个函数作为参数，该函数的两个参数分别是 **resolve** 和 **reject**。**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 新建后就会立即执行。

```
let promise = new Promise(function(resolve, reject) {  
  console.log('Promise'); // 新建 Promise 实例后就会立即执行，甚至在同步代码之前  
  resolve();  
});  
promise.then(function() {  
  console.log('resolved.');
```

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
```

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，它的作用是为 **Promise** 实例添加状态改变时的回调函数，**then** 方法的第一个参数是 **resolved** 状态的回调函数，第二个参数是 **rejected** 状态的回调函数，它们都是可选的。

then 方法返回的是一个新的 **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。

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

4. 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。

```
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));
```

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
```

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。

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，它捕获前面 then 方法执行中的错误，也更接近同步的写法（try/catch）。

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

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

catch()方法之中，还能再抛出错误。

```
someAsyncThing().then(function() {
  return someOtherAsyncThing();
}).catch(function(error) {
```

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

第二个 `catch()` 方法用来捕获前一个 `catch()` 方法抛出的错误。

5. 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。`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。

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

`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;
    }
  );
```

6. Promise.all()

`Promise.all()` 方法用于将多个 `Promise` 实例，包装成一个新的 `Promise` 实例。记忆同数组的 `every` 方法，**`rejected` 对应 `false`，`fulfilled` 对应 `true`。**

- 只有参数实例的状态都变成 `fulfilled`，包装实例的状态才会变成 `fulfilled`，此时参数实例的返回值组成一个数组，传递给包装实例的回调函数。
- 只要参数实例之中有一个被 `rejected`，包装实例的状态就变成 `rejected`，此时第一个被 `reject` 的实例的返回值，会传递给包装实例的回调函数。

```
// 生成一个Promise对象的数组
const promises = [2, 3, 5, 7, 11, 13].map(function (id) {
  return getJSON('/post/' + id + ".json");
});

Promise.all(promises).then(function (posts) {
  // ...
}).catch(function (reason){
  // ...
});
```

`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。

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]
```

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
```

7. 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。

- 只要参数实例之中有一个实例率先改变状态，包装实例的状态就跟着改变。那个率先改变的 `Promise` 实例的返回值，就传递给包装实例的回调函数。

```
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(console.log).catch(console.error);
```

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。

8. Promise.allSettled()

`Promise.race()` 方法同样是将多个 `Promise` 实例，包装成一个新的 `Promise` 实例。用来确定一组异步操作是否都结束了（不管成功或失败）。

- 只有等到参数数组的所有 `Promise` 对象都发生状态变更（不管是 `fulfilled` 还是 `rejected`），返回的 `Promise` 对象才会发生状态变更。

一旦发生状态变更，状态总是 `fulfilled`，不会变成 `rejected`。状态变成 `fulfilled` 后，它的回调函数会接收到一个数组作为参数，该数组的每个成员对应前面数组的每个 `Promise` 对象。

```
const resolved = Promise.resolve(42);
const rejected = Promise.reject(-1);

const allSettledPromise = Promise.allSettled([resolved, rejected]);
allSettledPromise.then(function (results) {
  console.log(results);
});
// [
//   { status: 'fulfilled', value: 42 },
//   { status: 'rejected', reason: -1 }
// ]
```

成员对象的 `status` 属性的值只可能是字符串 `fulfilled` 或字符串 `rejected`，用来区分异步操作是成功还是失败。如果是成功（`fulfilled`），对象会有 `value` 属性，如果是失败（`rejected`），会有 `reason` 属性，对

应两种状态时前面异步操作的返回值。

9. Promise.any()

`Promise.race()` 方法同样是将多个 `Promise` 实例，包装成一个新的 `Promise` 实例。记忆同数组的 `some` 方法，`rejected` 对应 `false`，`fulfilled` 对应 `true`。

- 只要参数实例有一个变成 `fulfilled` 状态，包装实例就会变成 `fulfilled` 状态。
- 如果所有参数实例都变成 `rejected` 状态，包装实例就会变成 `rejected` 状态。

```
Promise.any([
  fetch('https://v8.dev/').then(() => 'home'),
  fetch('https://v8.dev/blog').then(() => 'blog'),
  fetch('https://v8.dev/docs').then(() => 'docs')
]).then((first) => { // 只要有一个 fetch() 请求成功
  console.log(first);
}).catch((error) => { // 所有三个 fetch() 全部请求失败
  console.log(error); // AggregateError: All promises were rejected
});
```

`Promise.any()` 抛出的错误是一个 `AggregateError` 实例

```
let resolved = Promise.resolve(42);
let rejected = Promise.reject(-1);
let alsoRejected = Promise.reject(Infinity);

Promise.any([resolved, rejected, alsoRejected]).then(function (result) {
  console.log(result); // 42
});

Promise.any([rejected, alsoRejected]).catch(function (results) {
  console.log(results instanceof AggregateError); // true
  console.log(results.errors); // [-1, Infinity]
});
```

10. 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。

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

立即 `resolve()` 的 `Promise` 对象，是在本轮“事件循环”（`event loop`）的结束时执行，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
```

`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, 因此输出顺序是 `one`, `two`, `three`。

`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。`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  
});
```

`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`。

```
const 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。

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。

```
const p = Promise.resolve(); // p就是一个 Promise 对象
p.then(function () {
  // ...
});
```

11. 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`。

```
const p = Promise.reject('出错了');
// 等同于
const p = new Promise((resolve, reject) => reject('出错了'));
p.then(null, function (s) {
  console.log(s)
});
// 出错了
```

上例生成一个 `Promise` 对象的实例 `p`，状态为 `rejected`，回调函数会立即执行。

`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。

```
Promise.reject('出错了')
  .catch(e => {
    console.log(e === '出错了')
  })
// true
```

上例中，`Promise.reject()` 方法的参数是一个字符串，后面 `catch()` 方法的参数 `e` 就是这个字符串。